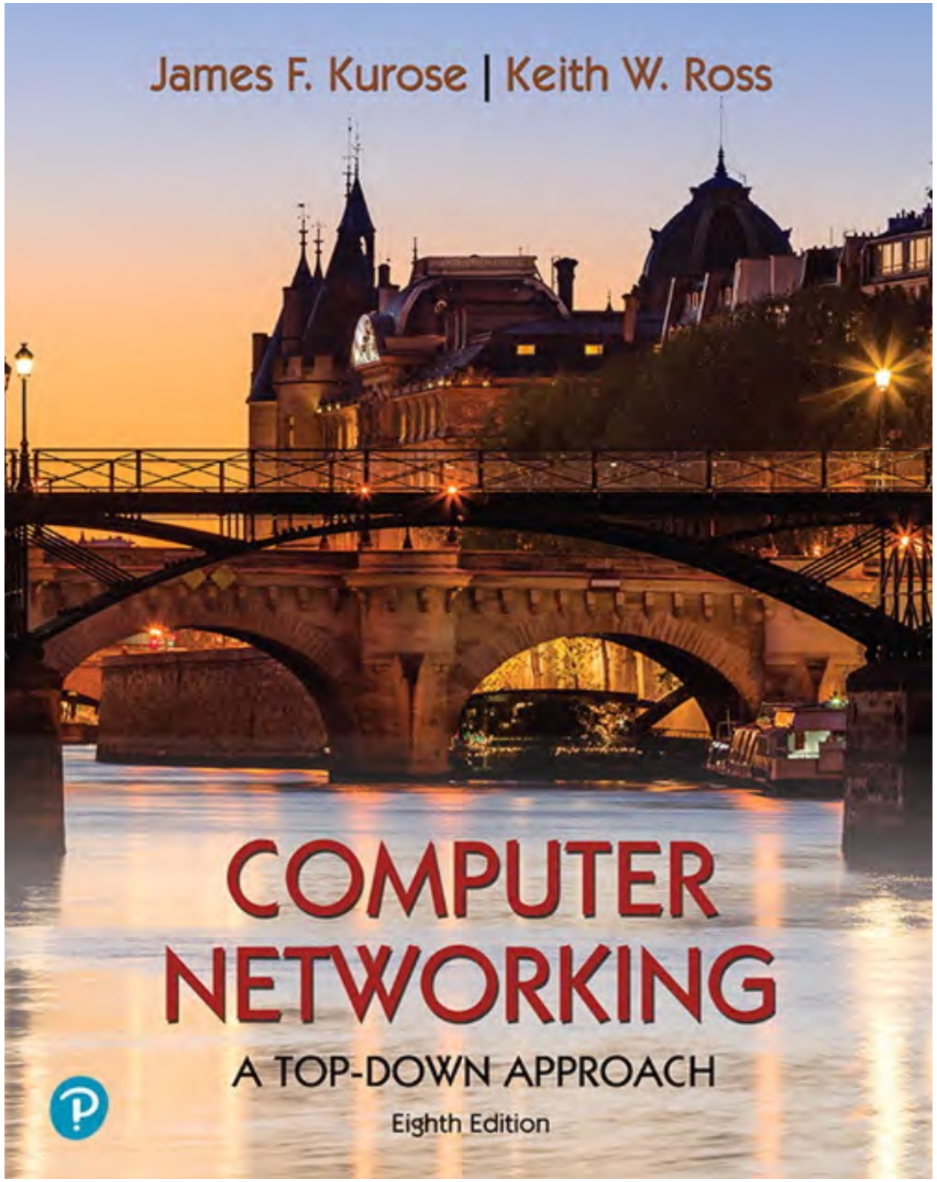
Chapter 3
Transport Layer

Computer Networking: A Top-Down Approach
8th Edition
Jim Kurose, Keith Ross
Pearson, 2021



Modified form the following
James Kurose and Keith Ross, Computer Networking: A Top-Down Approach, 8th Edition, Pearson Education Limited, 2021.

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

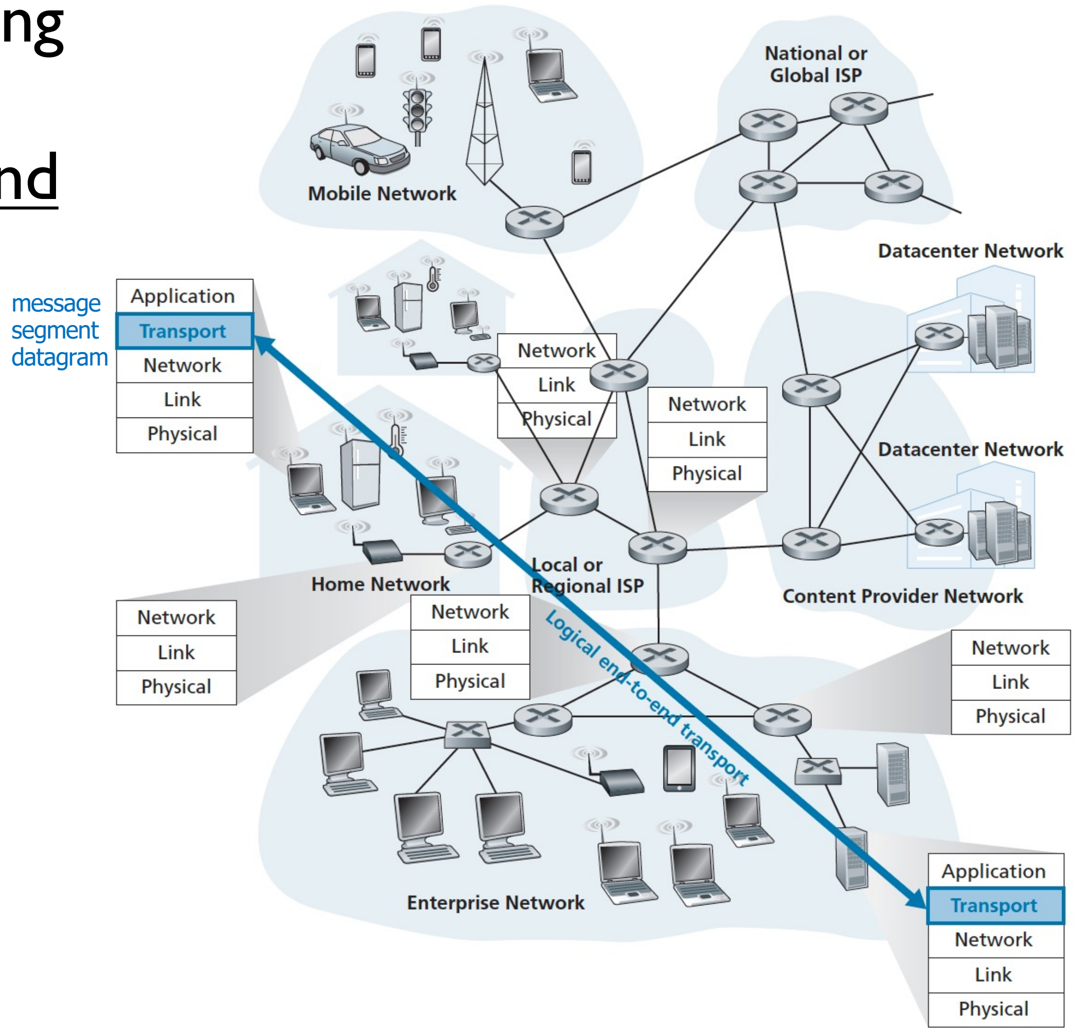
3.5 Connection-oriented transport: TCP
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Transport services and protocols

- Provide *logical communication* between app processes running on different hosts
- Transport protocols run in end systems
  - Send side: breaks app messages into *segments*, passes to network layer
  - Rcv side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps (Internet)
  - TCP (byte stream)
  - UDP (datagram)



Transport vs. network layer

- *Network layer:* logical communication between hosts
  - IP
- *Transport layer:* logical communication between processes
  - Relies on and enhances network layer services
  - IP + port number

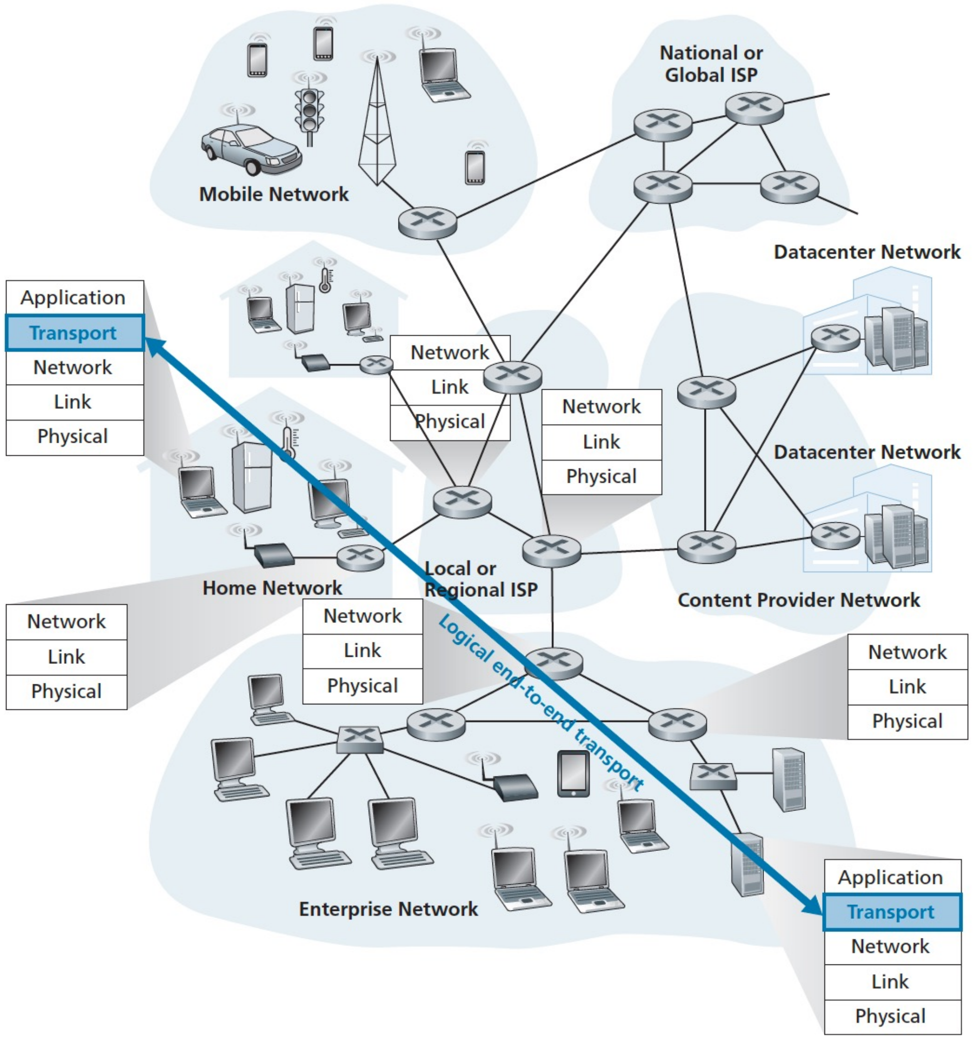
Household analogy

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

- Hosts = houses
- Processes = kids
- App messages = letters in envelopes
- Network-layer protocol = postal service
- Transport protocol = Ann and Bill who demux to in-house siblings

Internet transport-layer protocols

- Reliable, in-order delivery (TCP)
  - Connection setup
  - Flow control (sender / receiver)
  - Congestion control (network)
- Unreliable, unordered delivery (UDP)
  - No-frills extension of "best-effort" IP service
- IP services <u>not available</u>
  - <u>Delay</u> guarantees
  - <u>Bandwidth</u> guarantees



Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

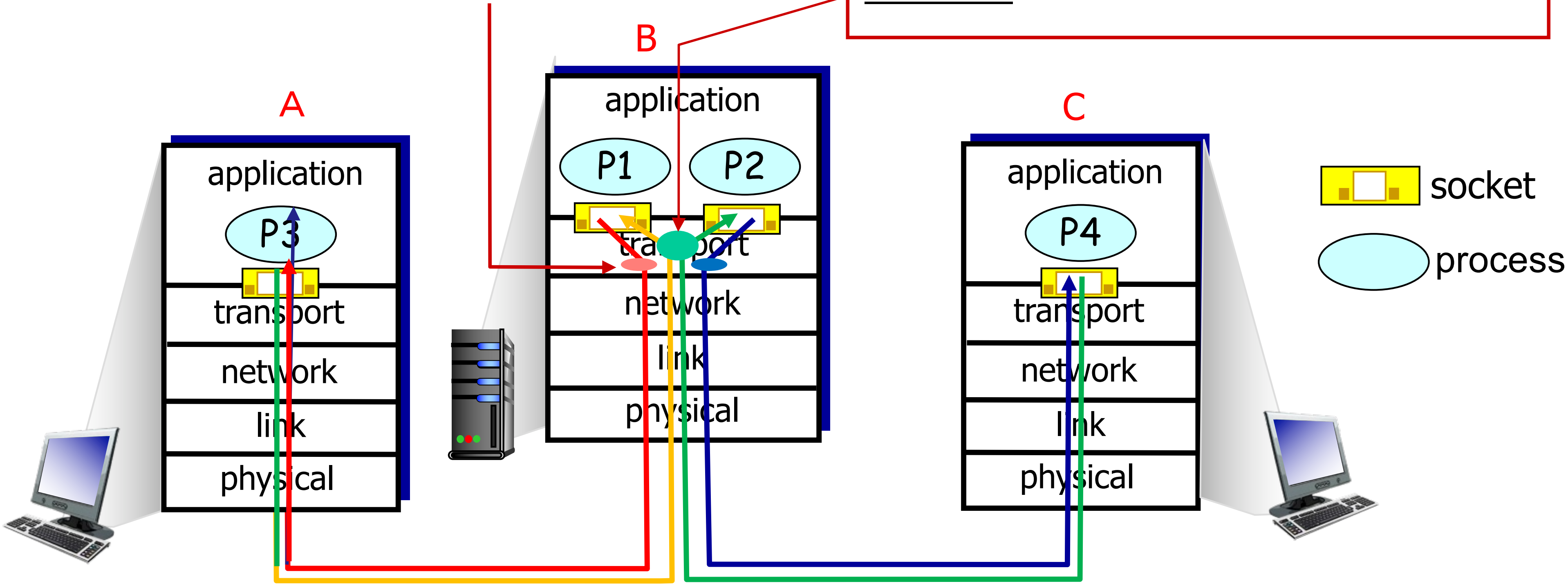
Multiplexing/demultiplexing

Multiplexing at sender

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

Demultiplexing at receiver

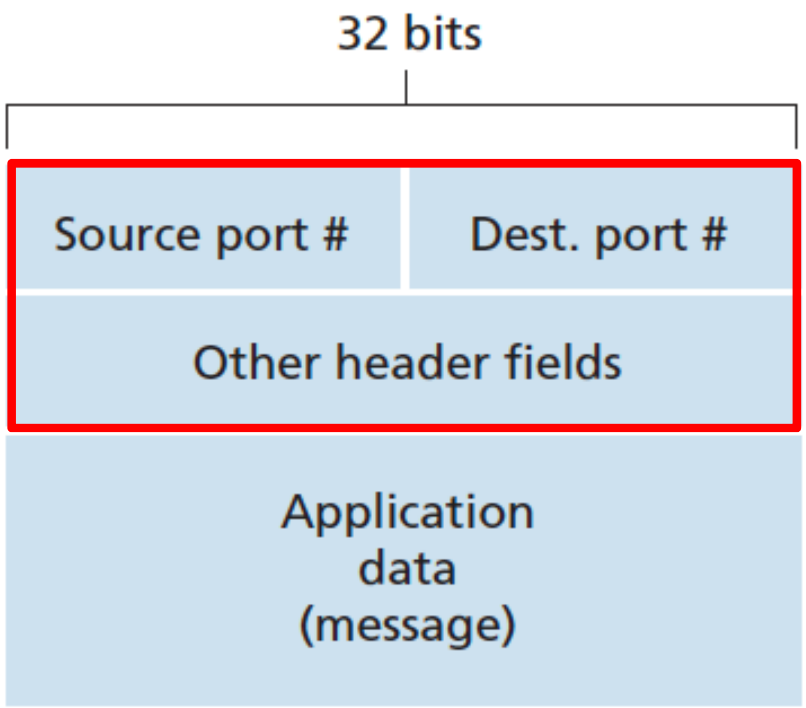
Use header info to deliver received segments to correct socket



How demultiplexing works

- Host receives IP datagrams
  - Each datagram has network-layer 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

Segment: transport layer (TCP, UDP) data unit
Datagram: network layer (IP) data unit
Datagram = IP header + Segment



Source and destination port-number fields in a transport-layer **segment**

Connectionless demultiplexing (UPD)

- Created socket has host-local port #:

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

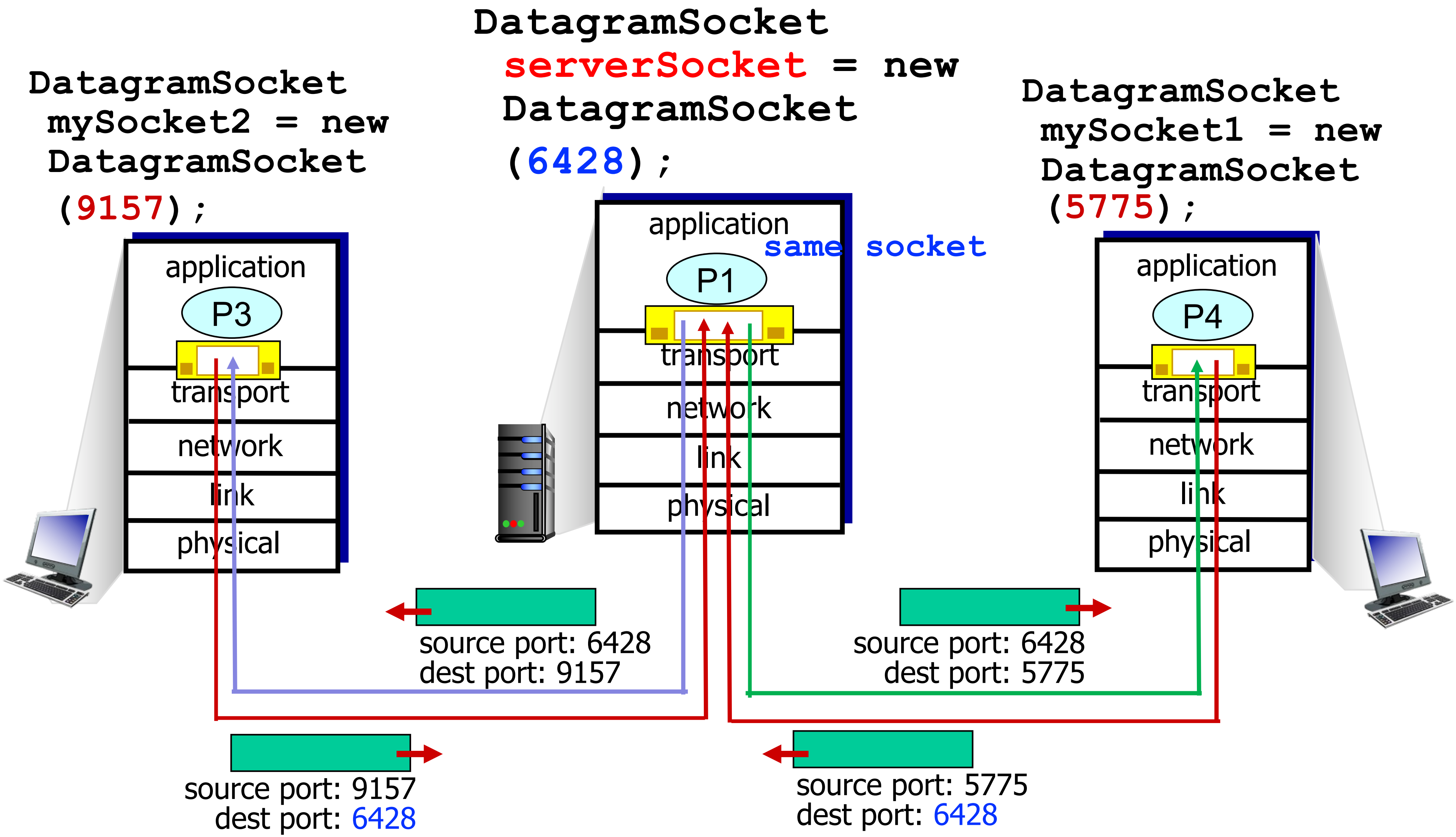
host-local port#

- When creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

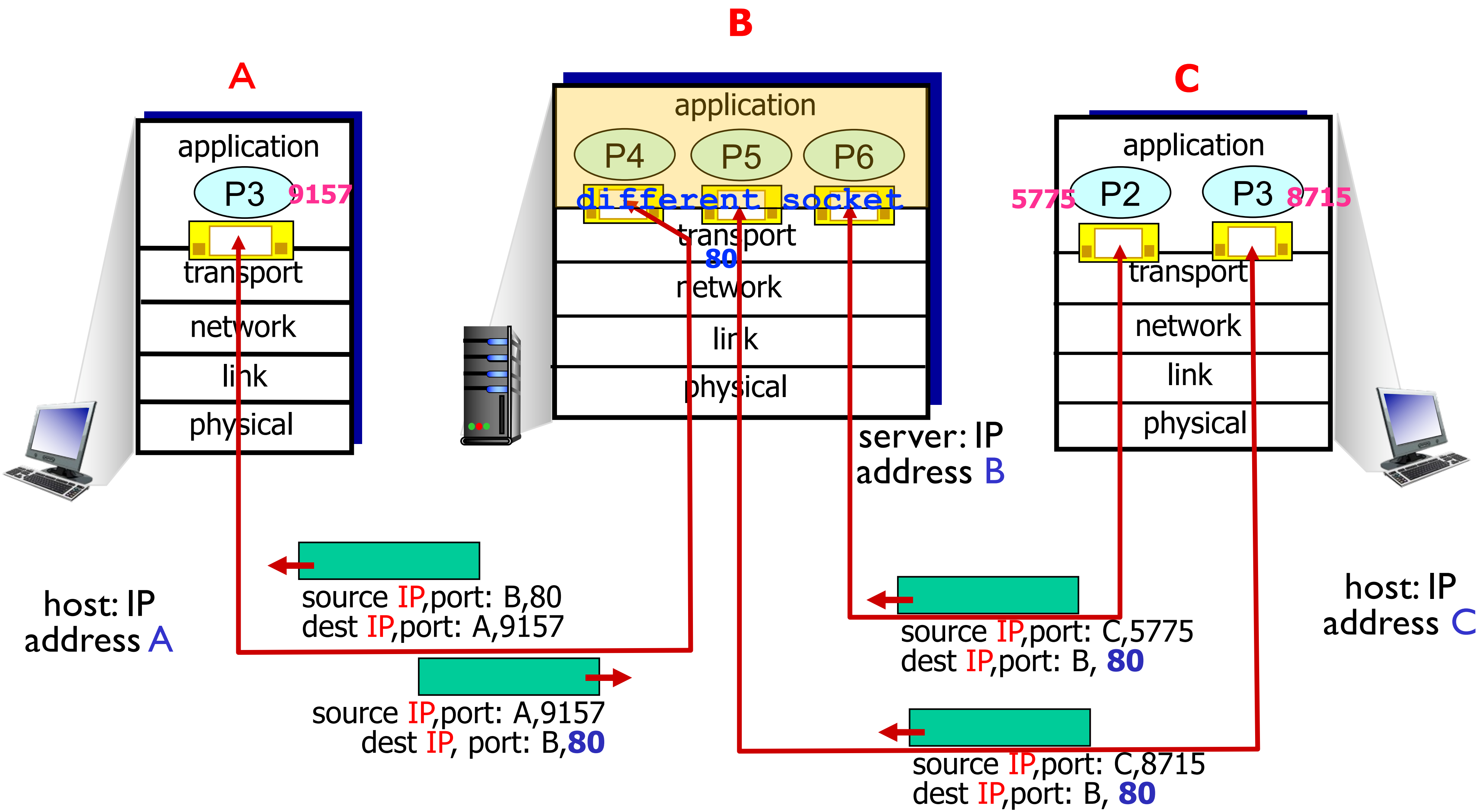
- When host receives UDP segment
  - Checks destination port # in segment
  - Directs UDP segment to socket with that port #

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

Connectionless demux: example



Connection-oriented demux: example

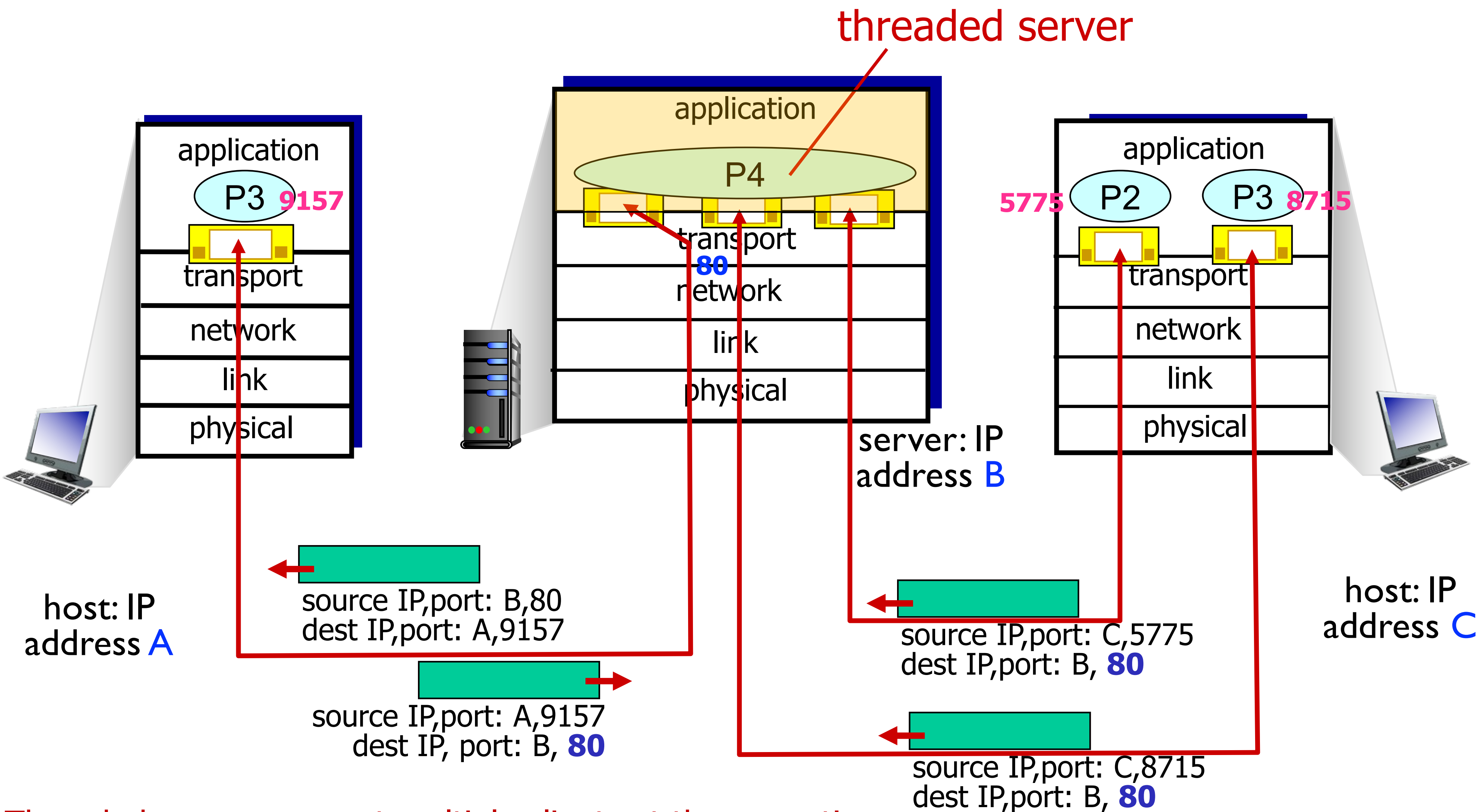


* Three segments, all destined to IP address: B
* Dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux (TCP)

- 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



Threaded server: support multiple clients at the same time (whenever a client request comes, a separate thread can be assigned for handling each request)

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

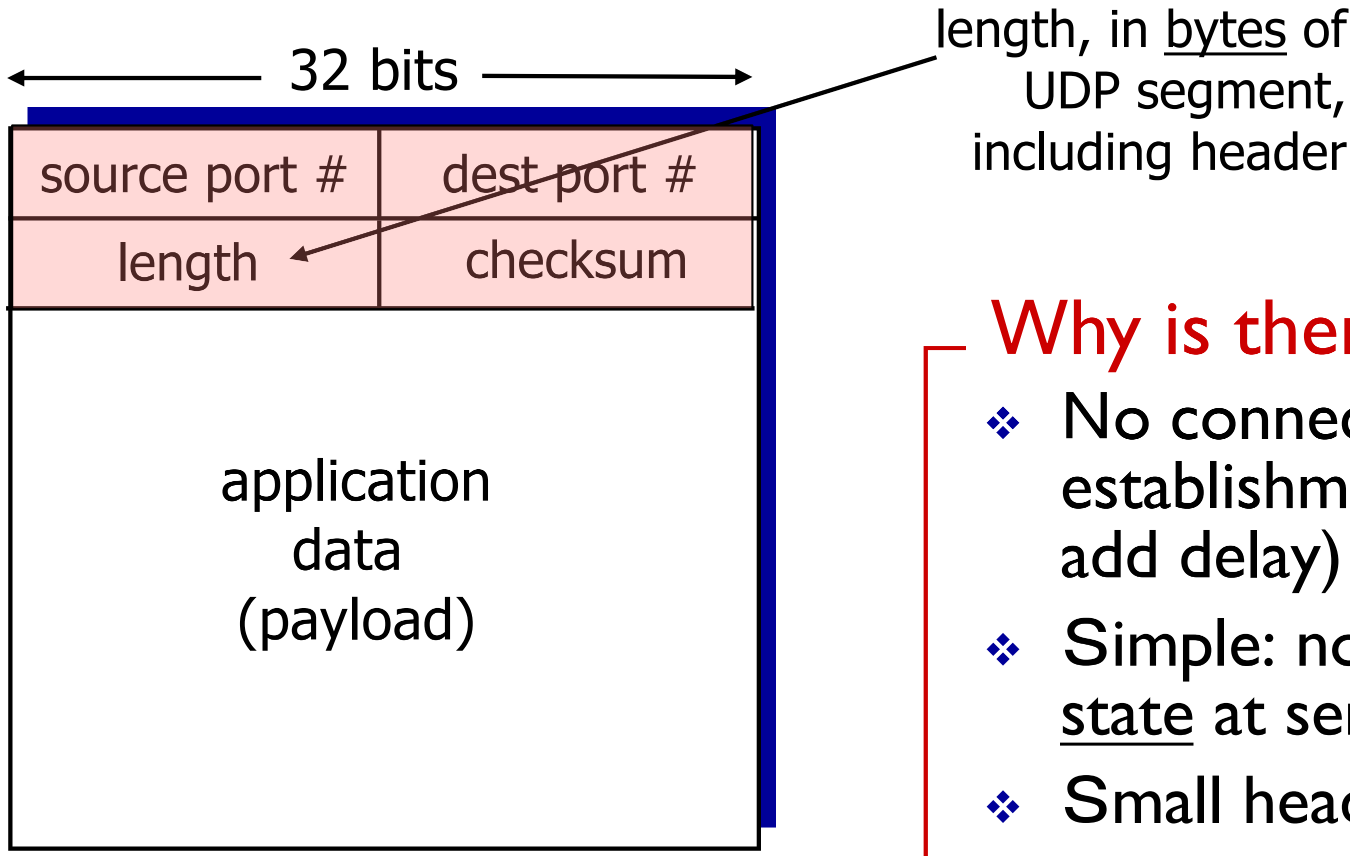
UDP: User Datagram Protocol [RFC 768]

- "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 apps
  - Streaming multimedia apps (loss tolerant, data rate sensitive)
  - DNS
  - SNMP (Simple Network Management Protocol)
    - Eg., query no. of ports, network interface of port, turn off port, etc.
- Reliable transfer over UDP
  - Add reliability (simple) at application layer
  - Application-specific error recovery!

No IP address fields
(network layer header)

UDP: segment header



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
																	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

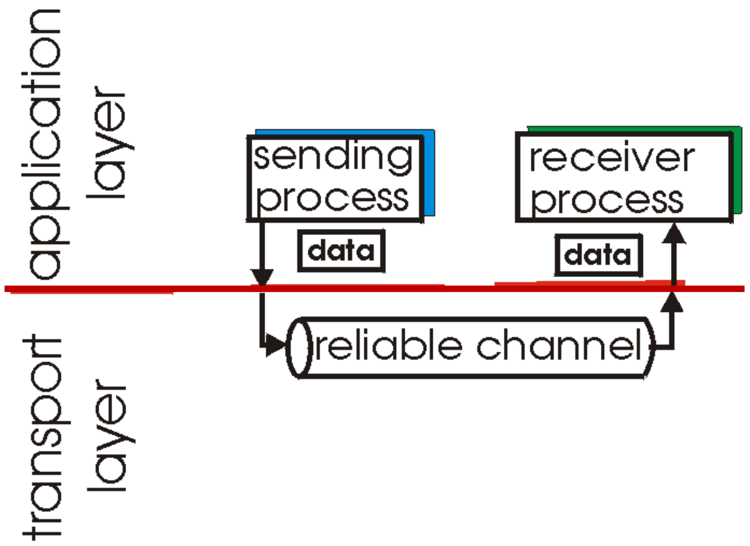
3.5 Connection-oriented transport: TCP
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

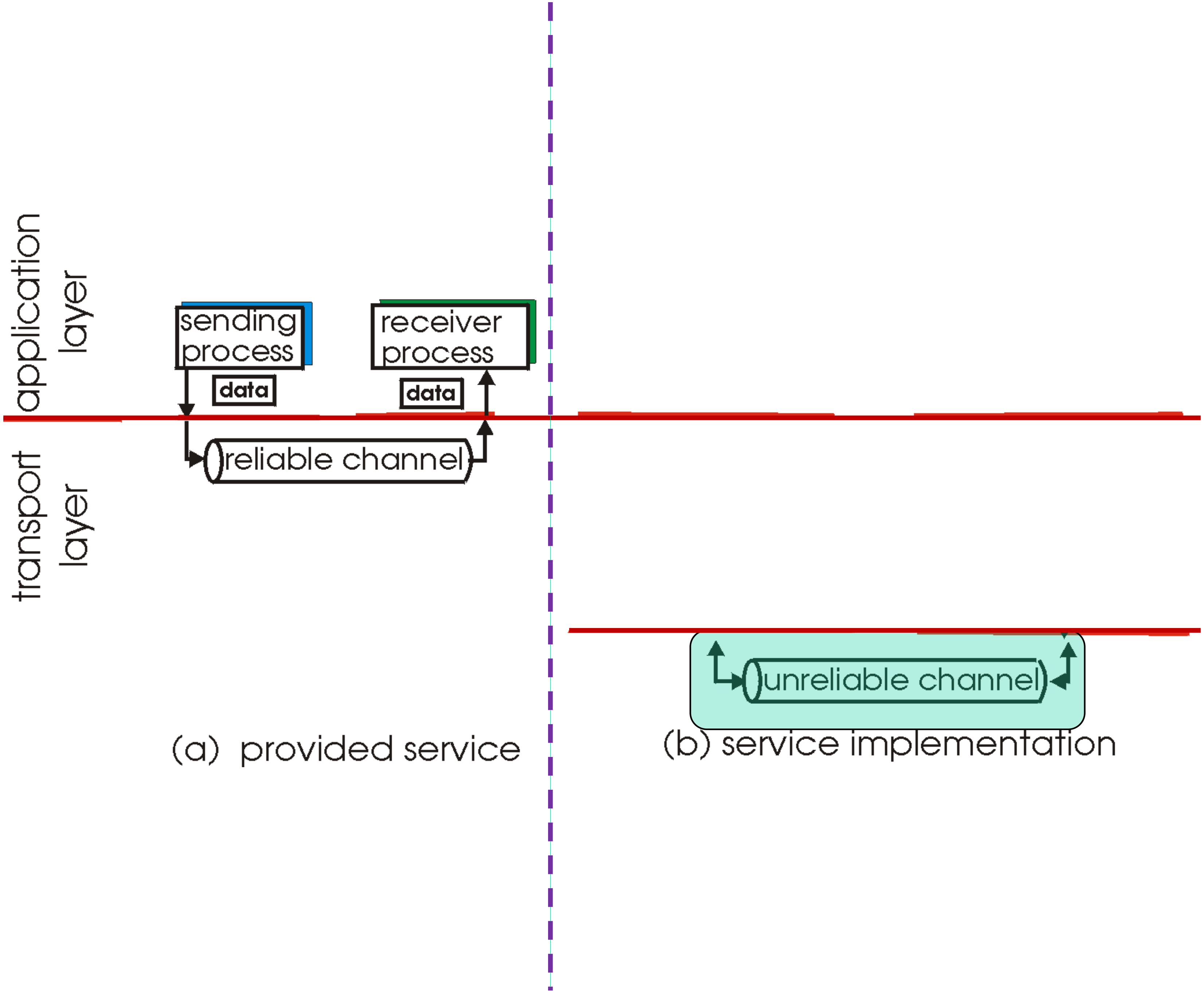
Principles of reliable data transfer

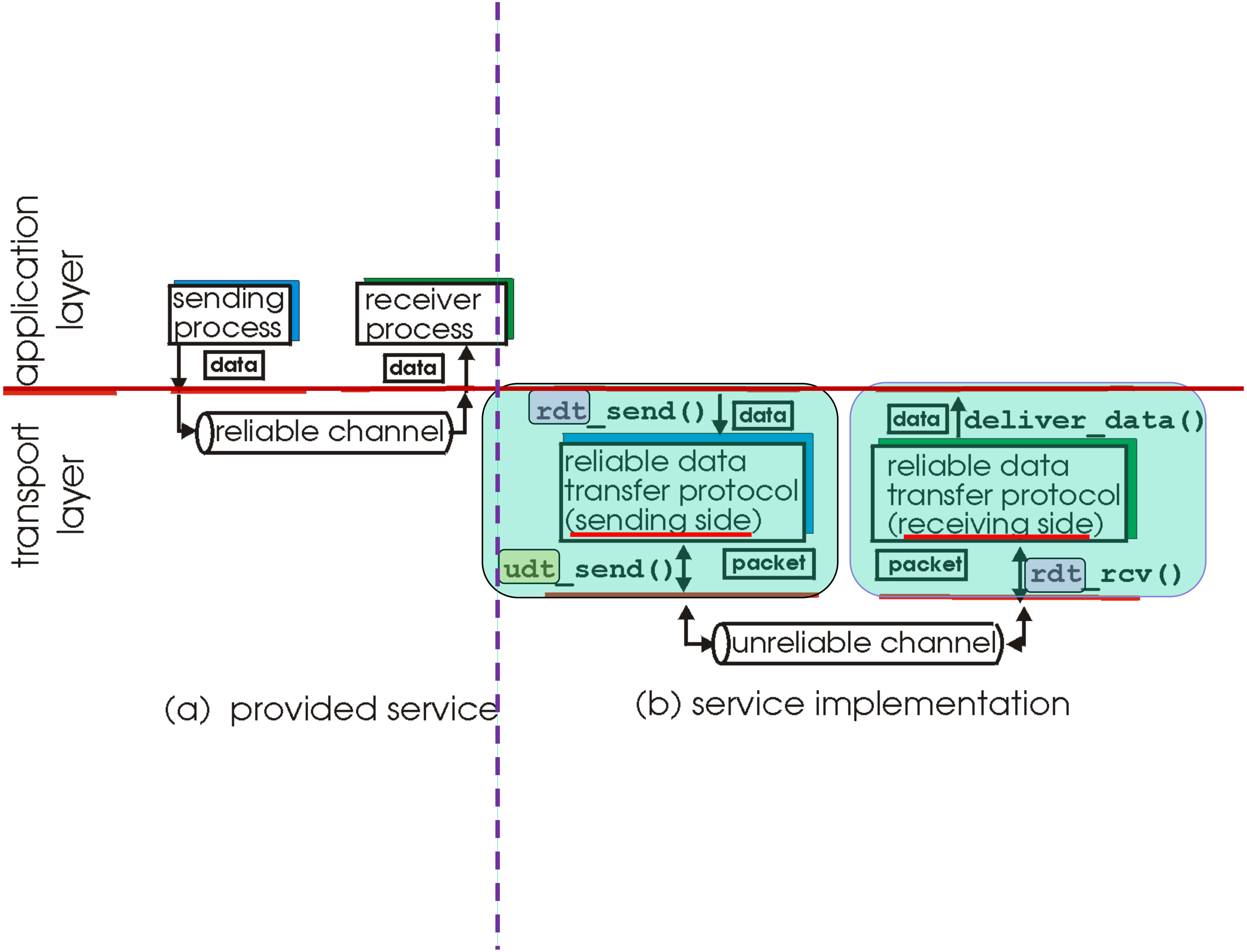
- Important in application, transport, link layers
  - Top-10 list of important networking topics!



(a) provided service

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

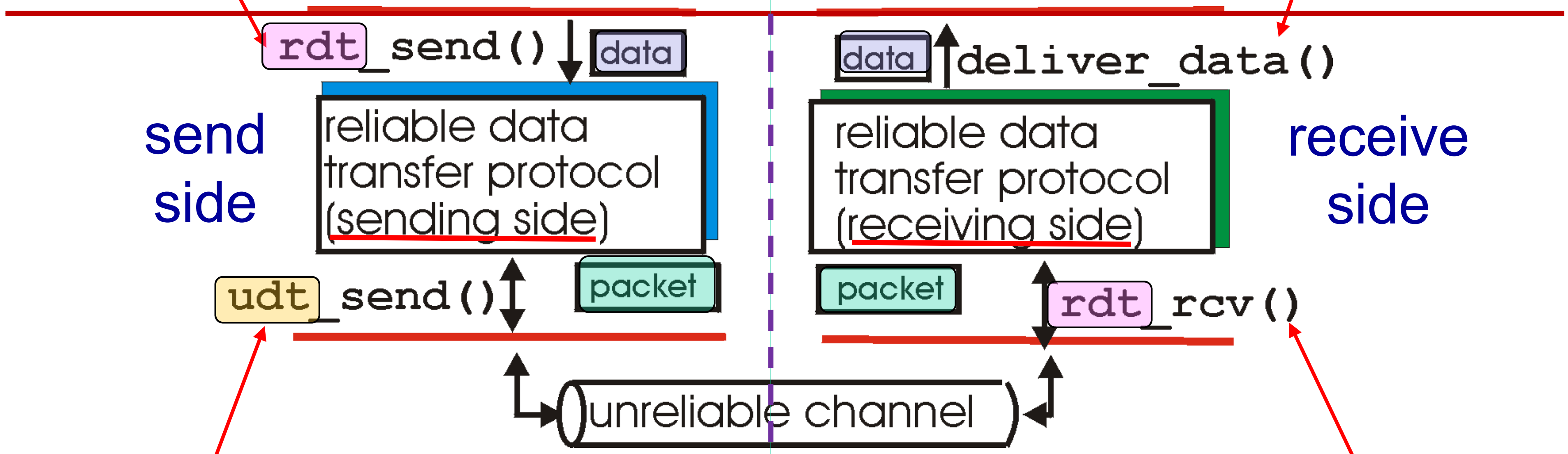

application layer
transport layer
sending process
data
receiver process
data
reliable channel
unreliable channel
(a) provided service
(b) service implementation


application layer
transport layer
sending process
data
receiver process
data
reliable channel
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
(a) provided service
(b) service implementation

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 layer

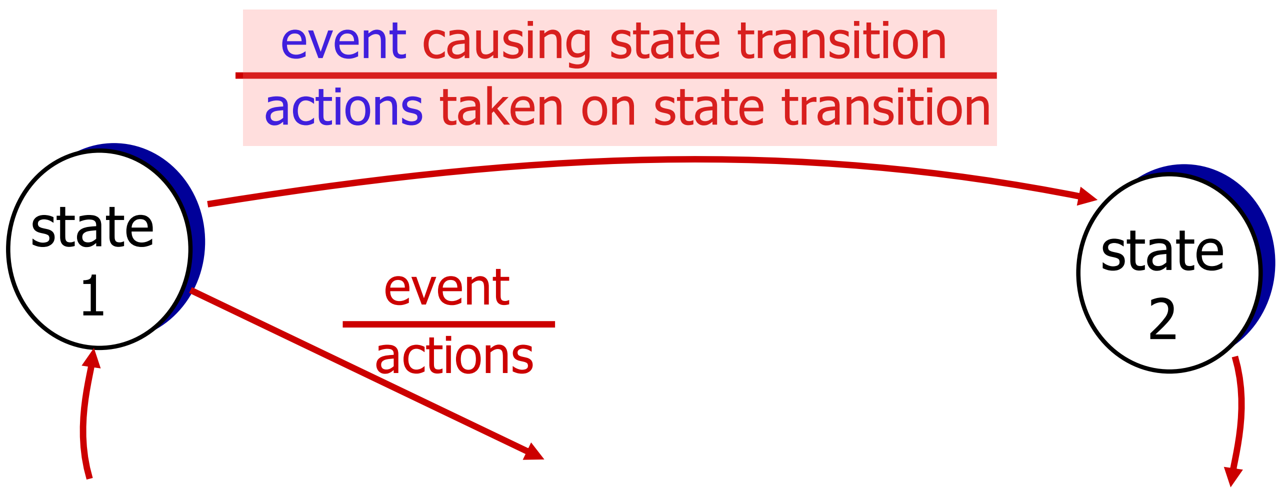


udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

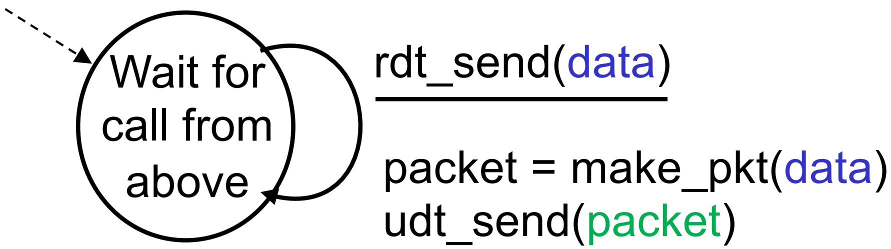
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - But control info (for connection) will flow on both directions
- Use finite state machines (FSM) to specify sender, receiver

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

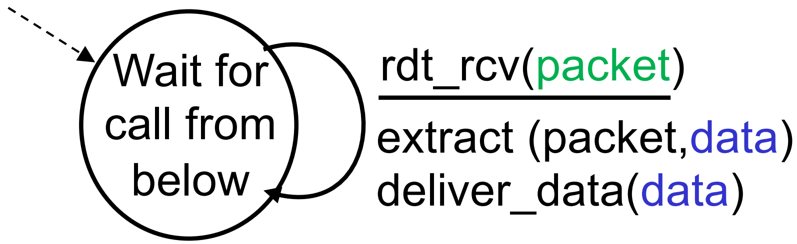


rdt1.0: reliable transfer over a reliable channel

- Underlying channel <u>perfectly reliable</u>
  - 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

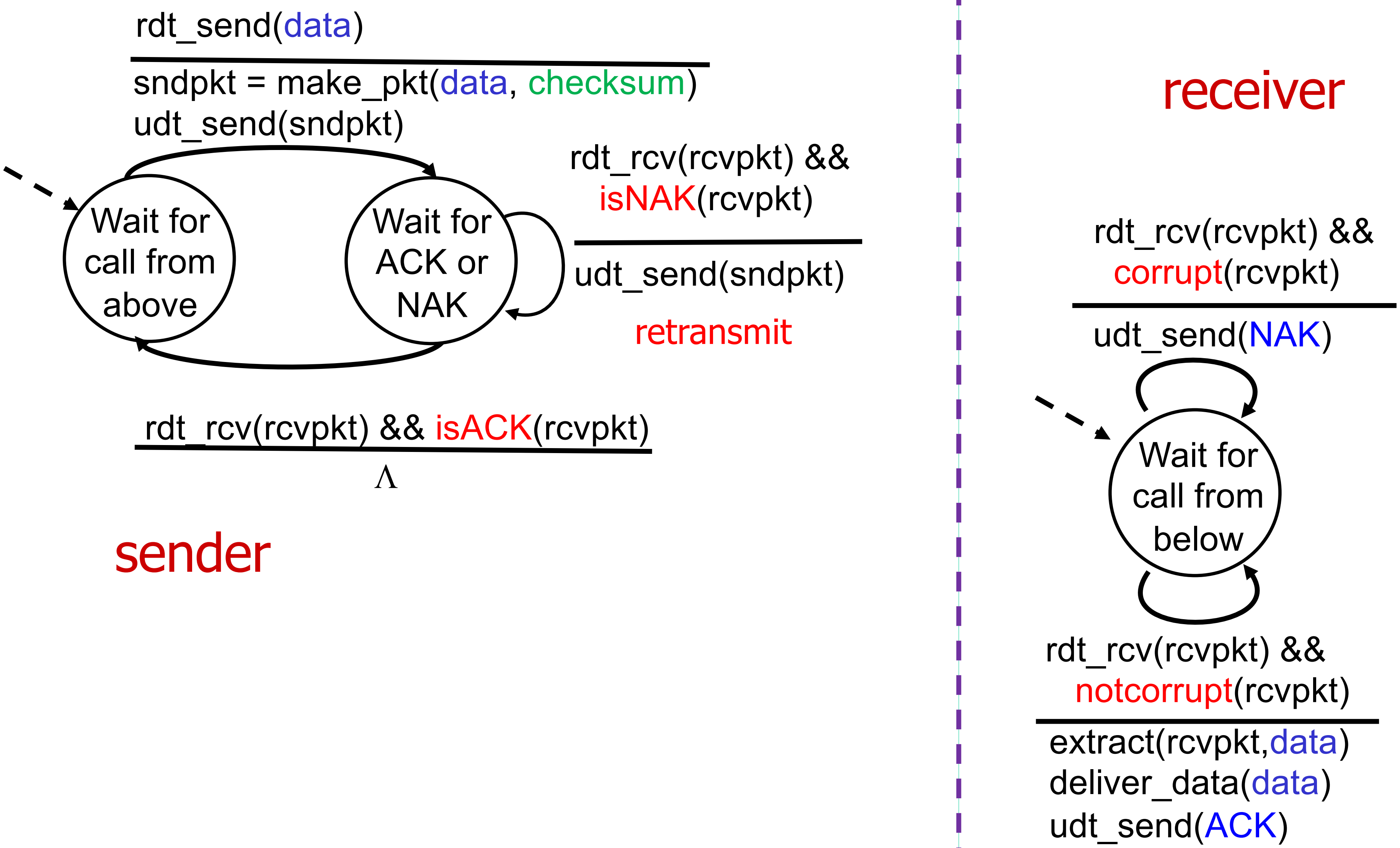


receiver

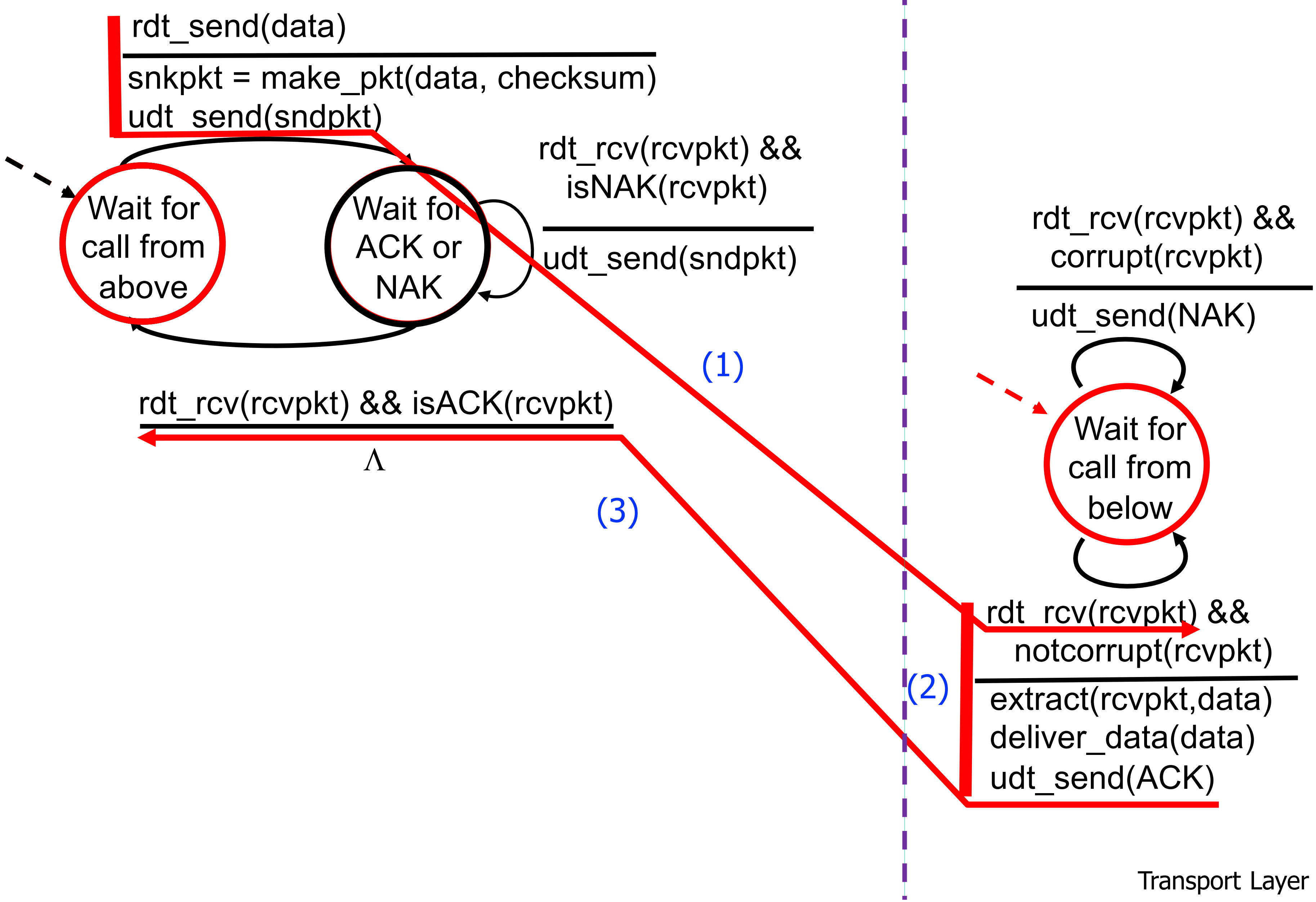
rdt2.0: channel with bit errors

- Underlying channel may flip bits in packet
  - Checksum to detect bit errors
- 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
  - Receiver feedback: control msgs (ACK, NAK) from receiver to sender

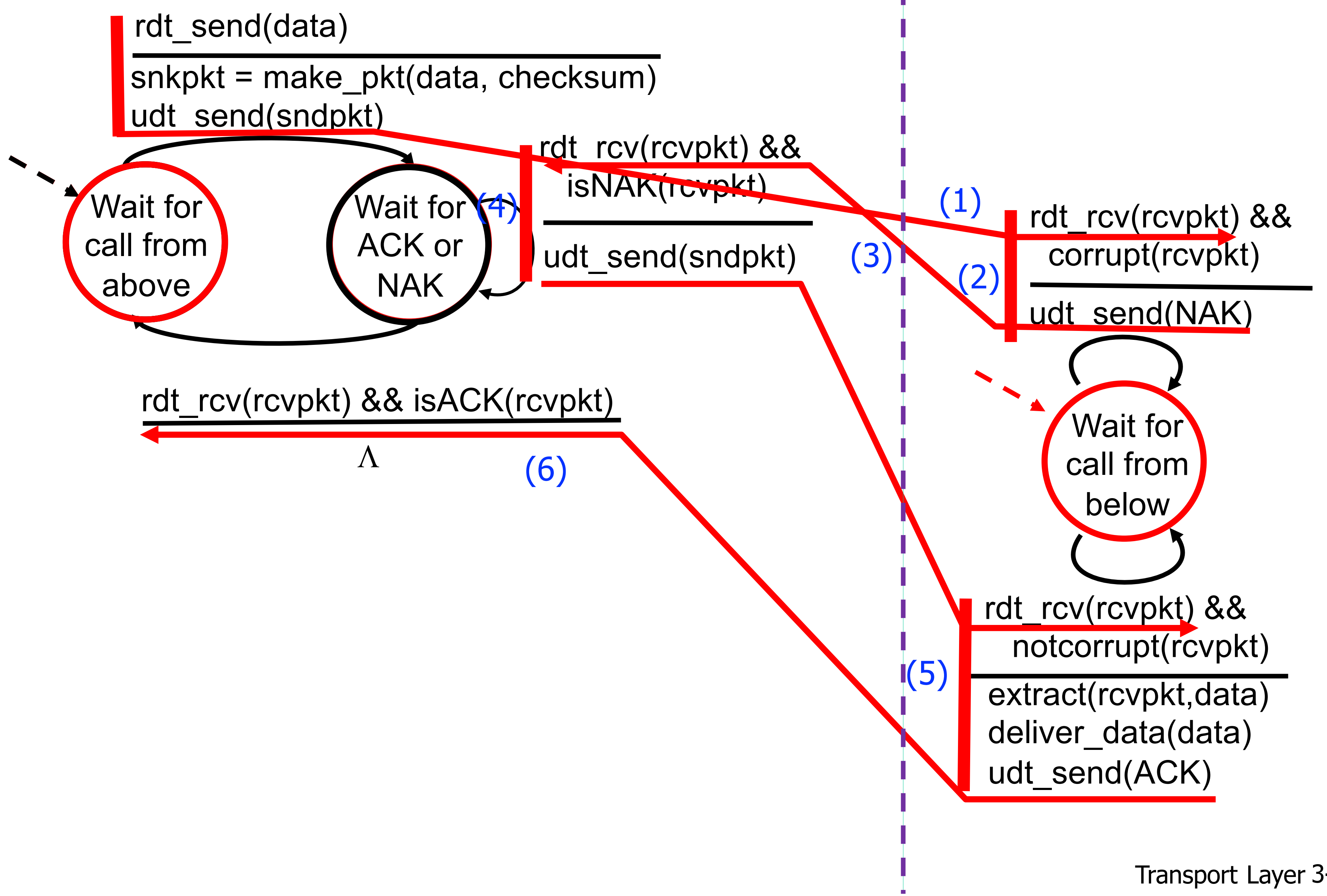
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- Sender doesn't know what happened at receiver!
- Can't just retransmit: possible duplicate

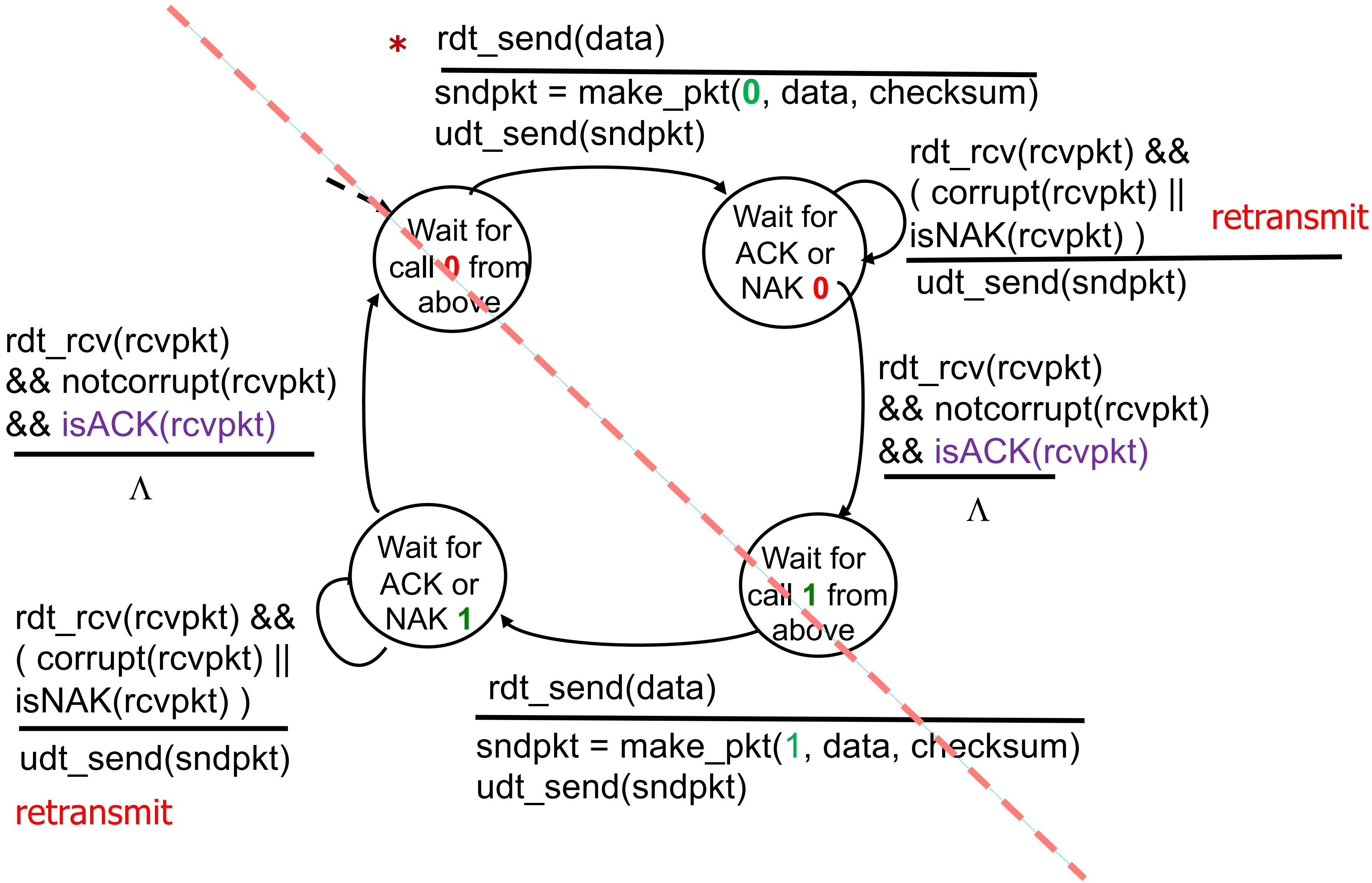
Handling duplicates

- Sender retransmits current pkt if ACK/NAK corrupted
- Sender adds *sequence number* (0 or 1) to each pkt to prevent duplicate pkt due to receiving garbled ACK/NAK
- Receiver discards (doesn't deliver up) duplicate pkt

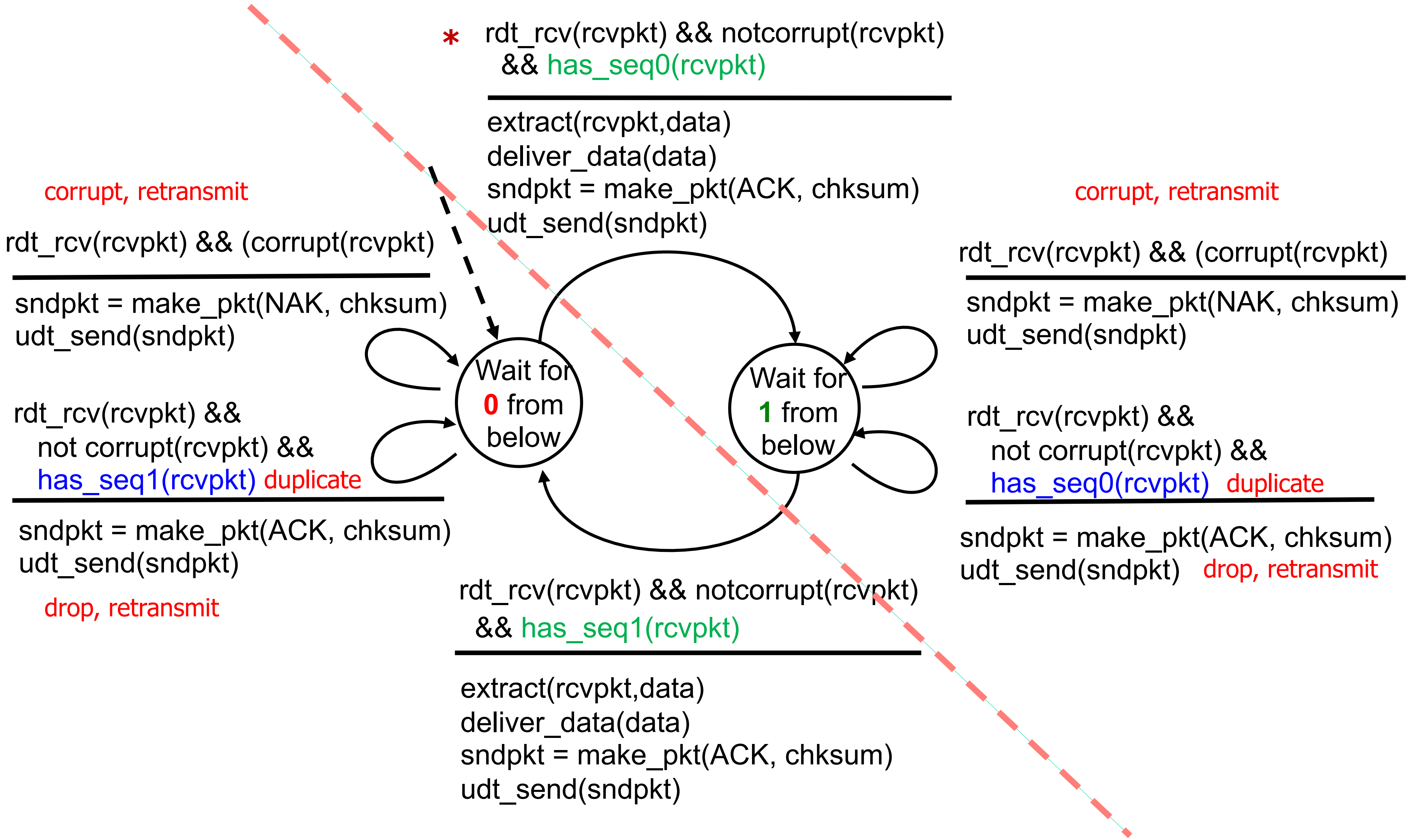
Stop and wait

Sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender

- seq # added to pkt
- Two seq. #'s (0,1) is suffice
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must "remember" whether "expected" pkt should have seq # of 0 or 1

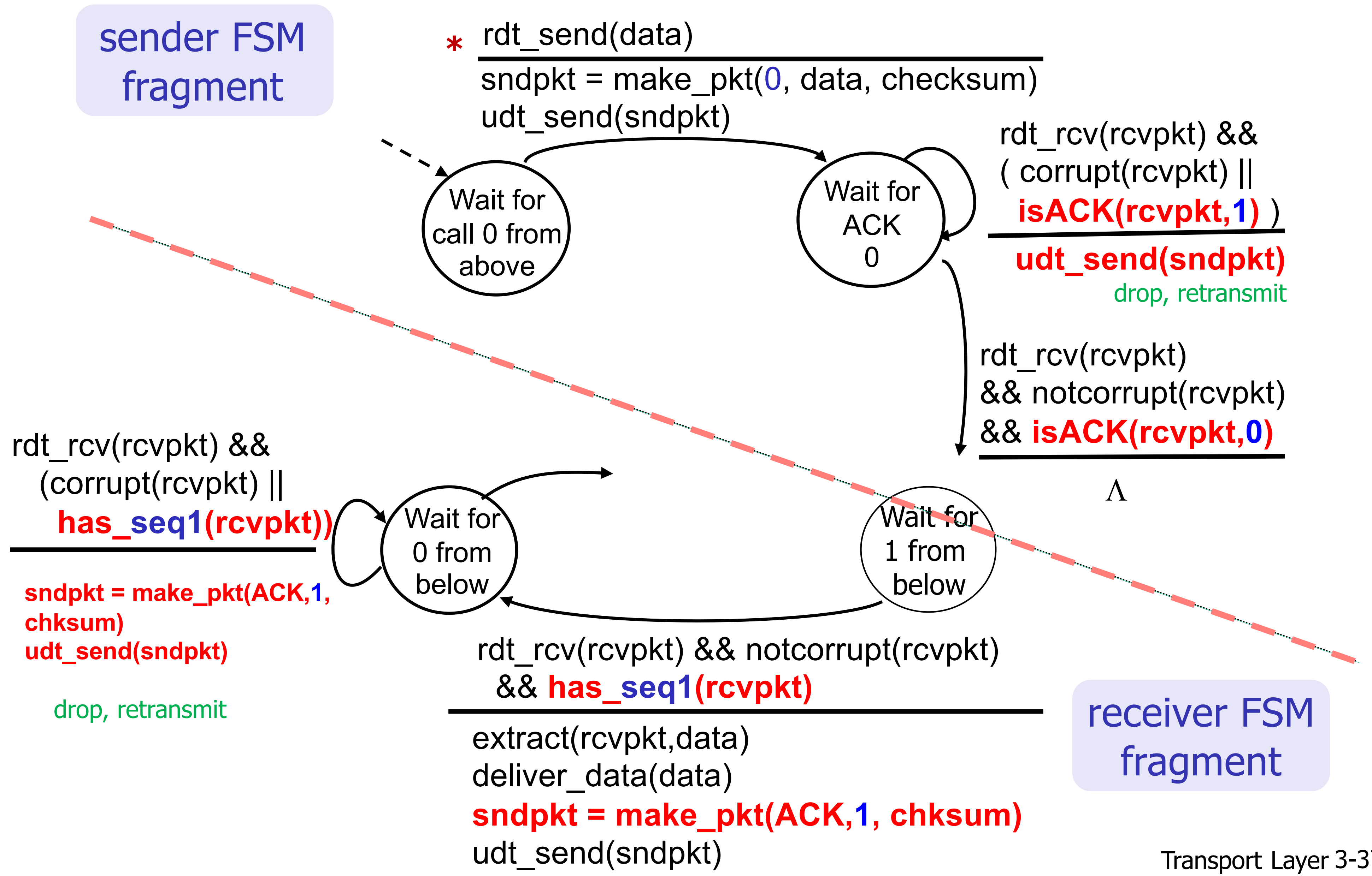
Receiver

- 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 successfully 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, recever fragments



rdt3.0: channels with errors *and* loss

New assumption

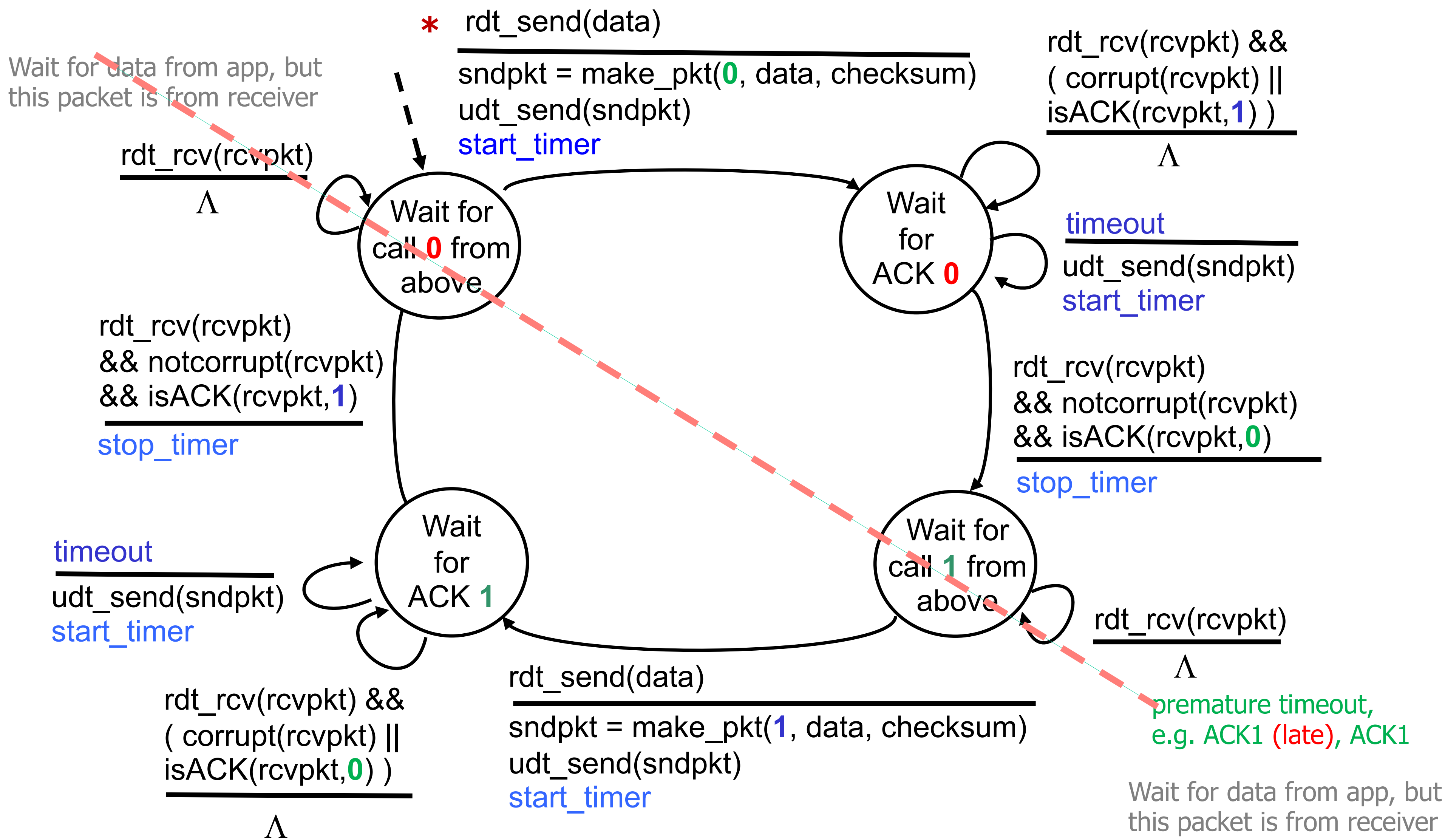
underlying channel can also lose packets (data, ACKs)

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

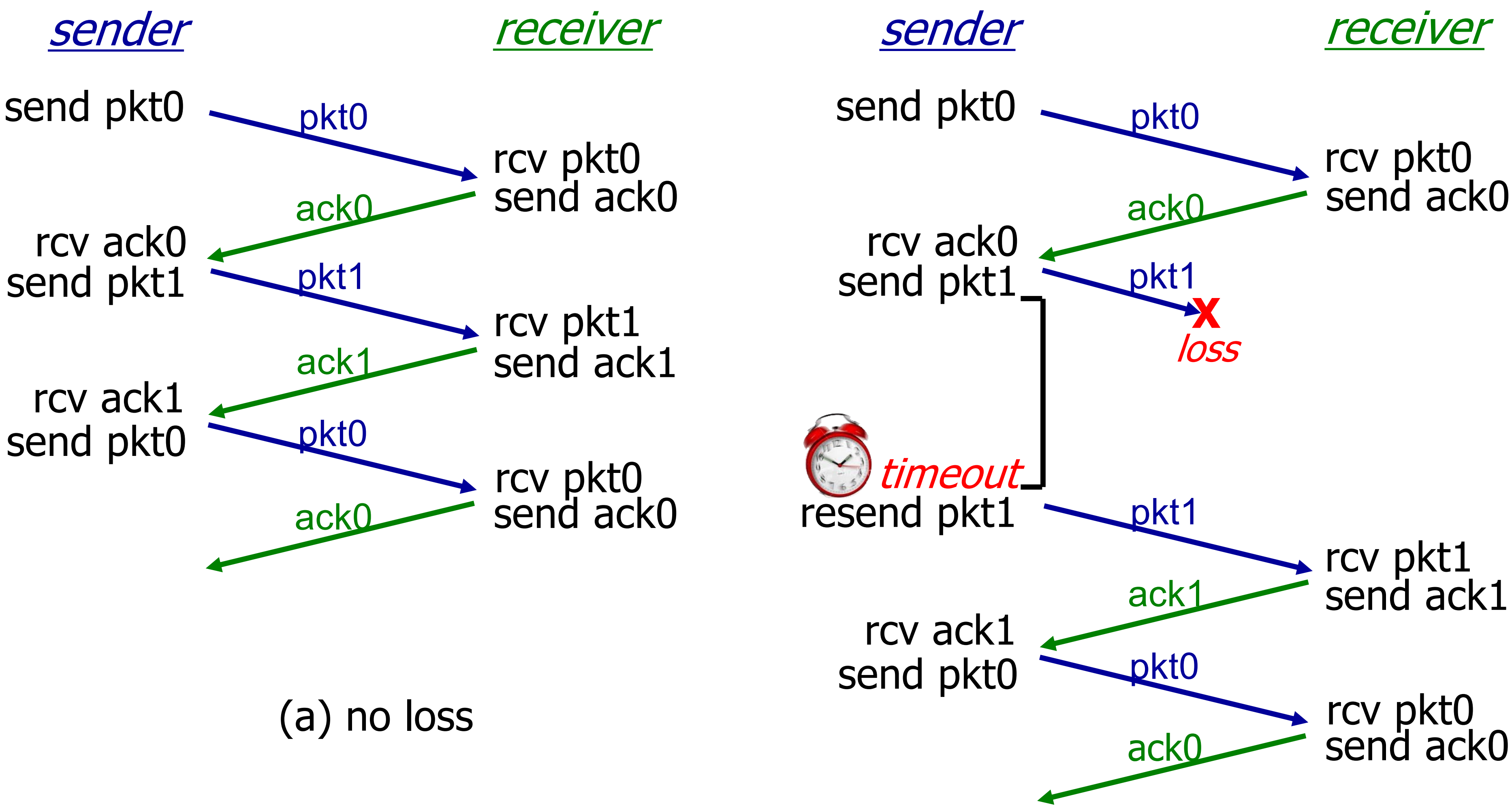
Approach: sender waits "reasonable" amount of time for ACK (round trip time + process time)

- 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 (RTT+ tolerate time)

rdt3.0 sender

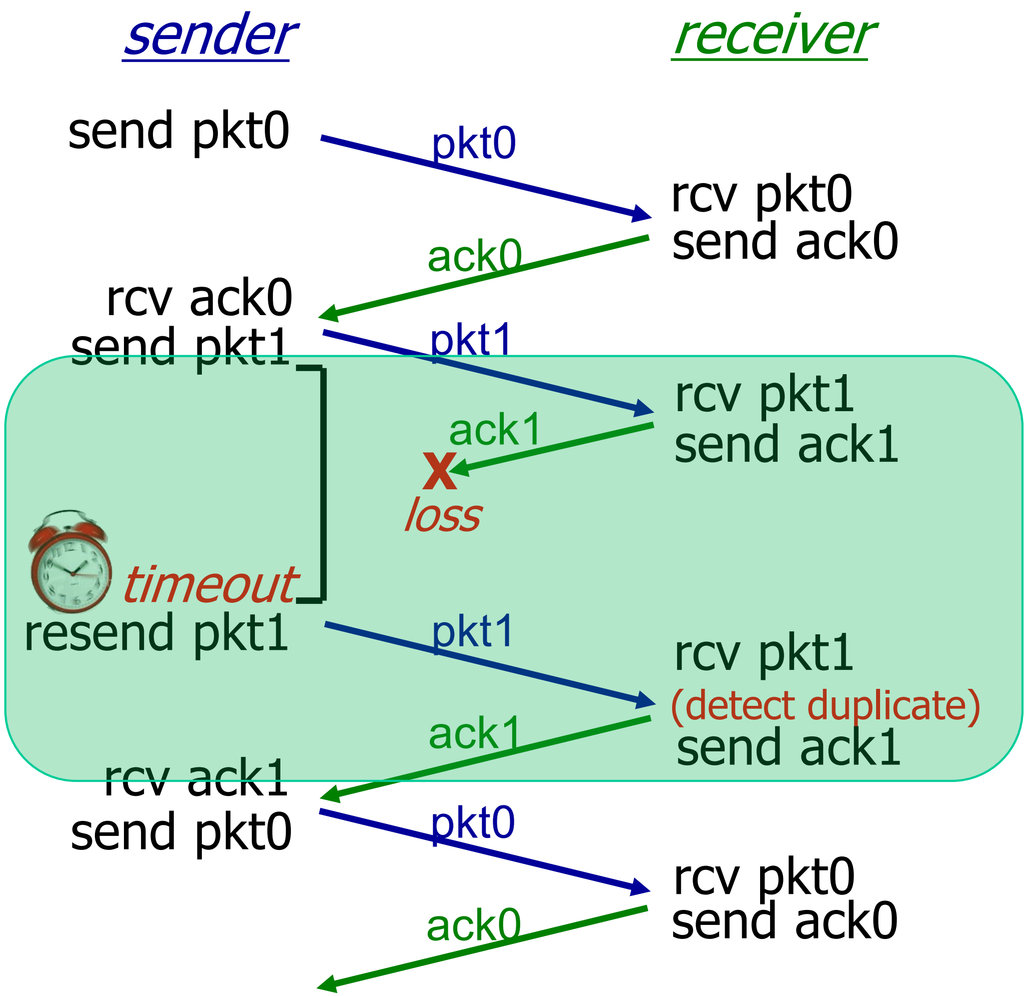


rdt3.0 in action

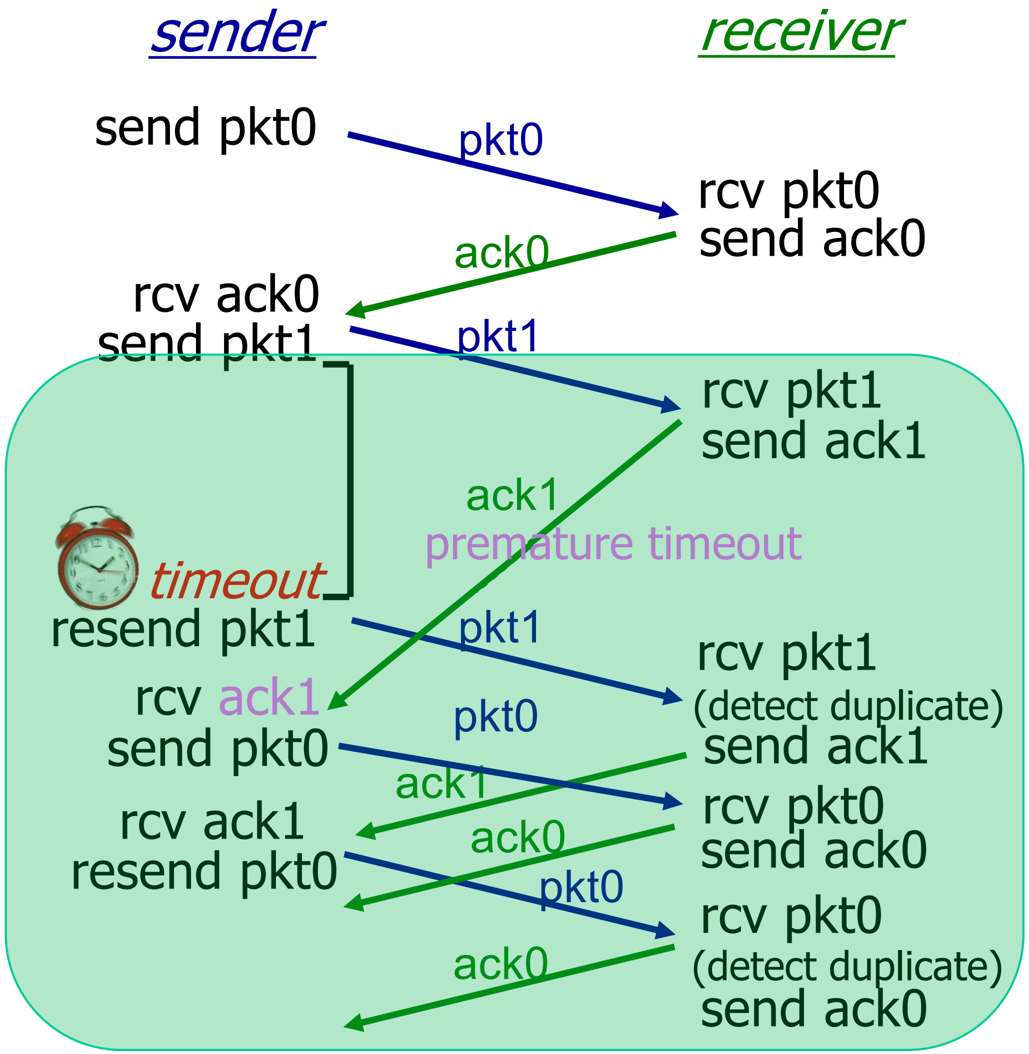


(a) no loss

(b) packet loss



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link (*R*), 15 ms prop. delay, 8000 bit packet (*L*), RTT=30 msec :

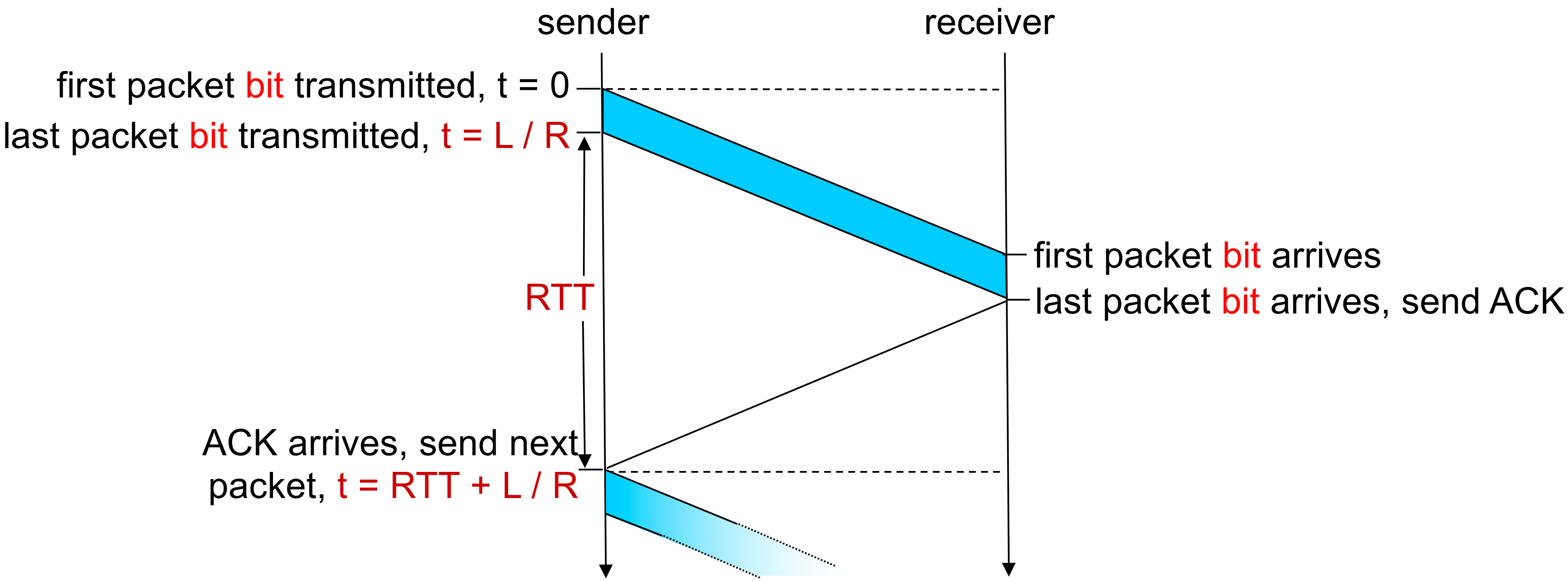
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs} \quad (0.008 \text{ msec})$$

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

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.000267$$

  - If RTT=30 msec, 8000 bits every 30.008 msec equals to 267 kbps thruput over 1 Gbps link
- Network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

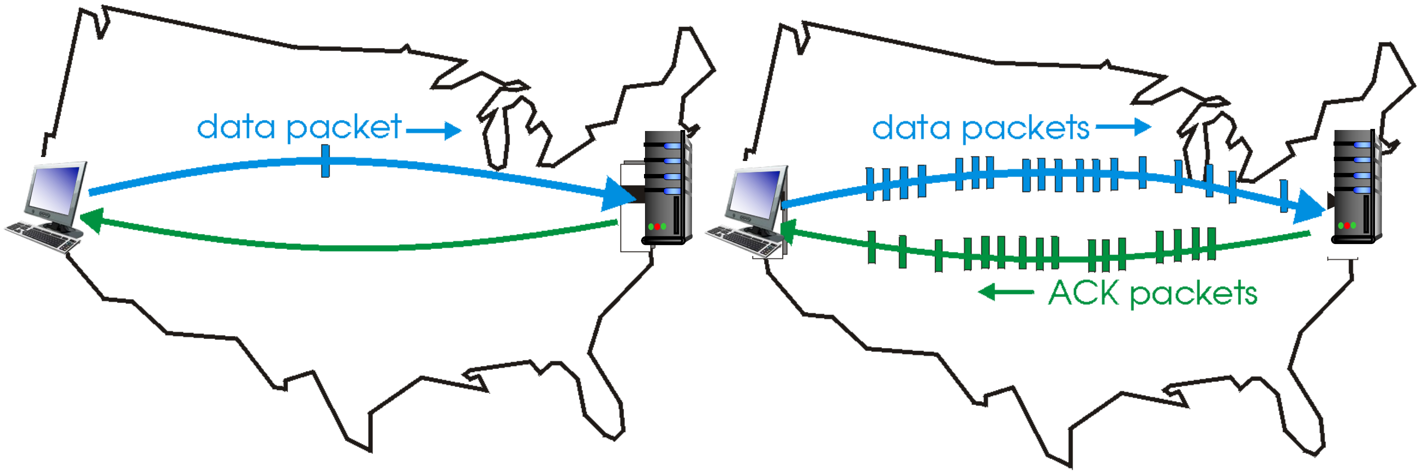


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.000267$$

Pipelined protocols

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

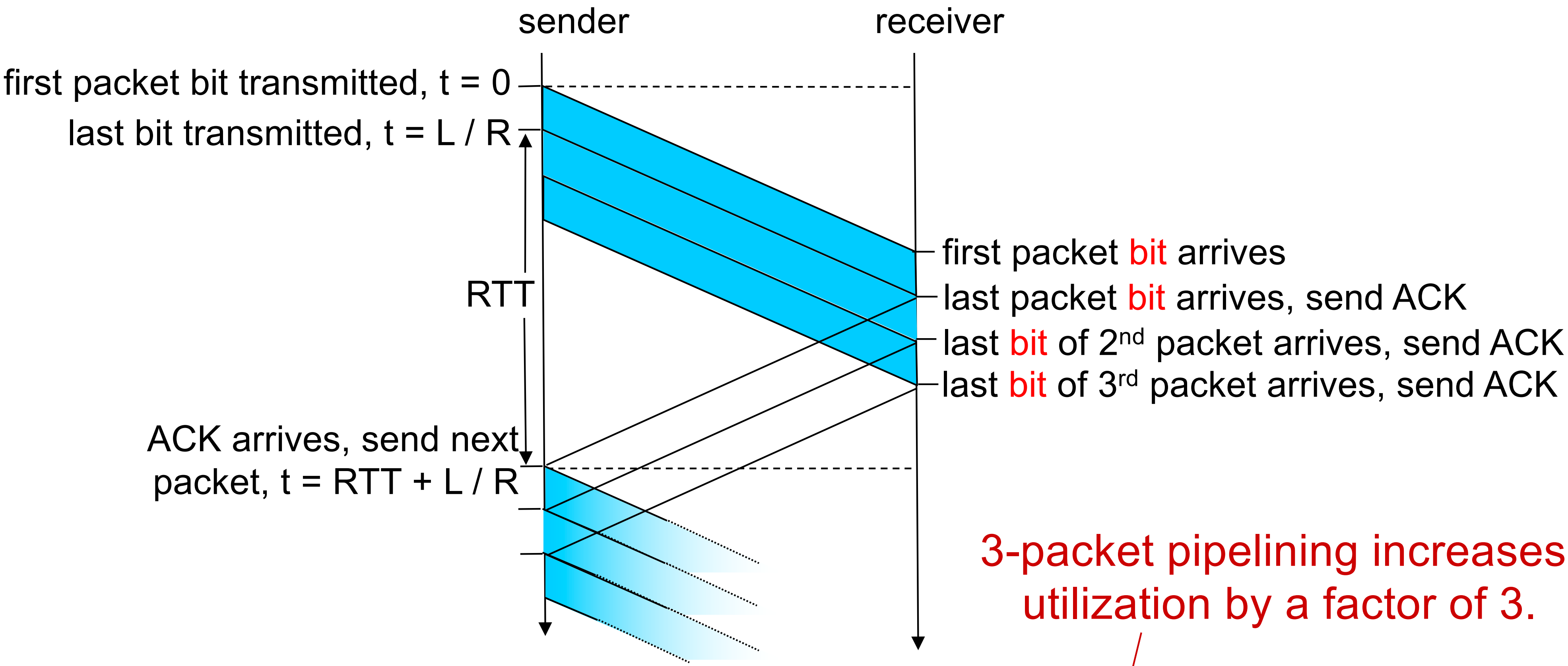
- Range of sequence numbers must be increased [more than 2]
- Buffering at sender and/or receiver



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

- Two generic forms of pipelined protocols
  - *Go-Back-N (one timer, timout then resend all unacked pkts)*
  - *Selective repeat (each timer per packet, timeout only resend the timeout pkt)*

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3.

$$U_{sender} = \frac{3*L/R}{RTT + L/R} = \frac{.024}{30.008} = 0.0008$$

Pipelined protocols: overview

Go-back-N

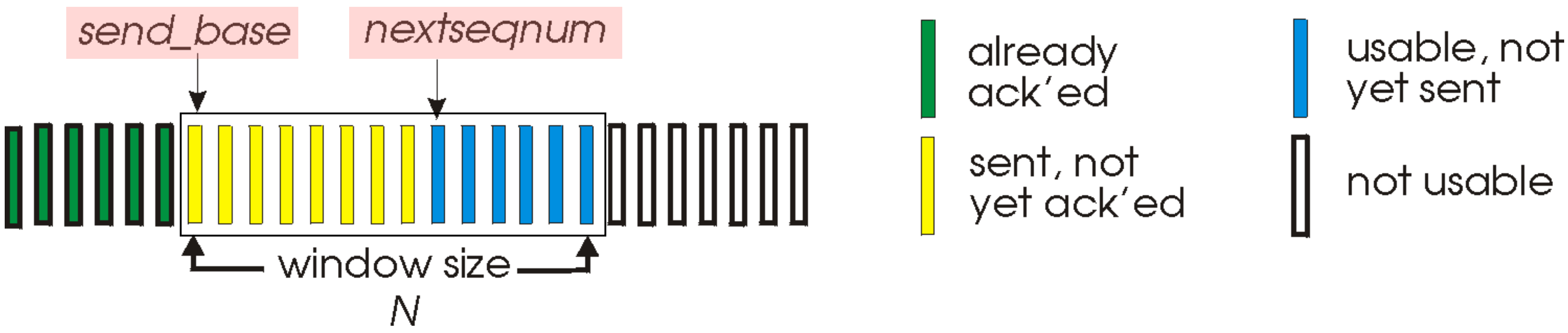
- Sender can have up to *N* unacked packets in pipeline
- Receiver only sends *cumulative ack*
  - Doesn't ack packet if there's a gap (e.g. 0, 1, 2, 4, 5)
- Sender has only one timer for the oldest unacked packet
  - When timer expires, retransmit *all* unacked packets

Selective Repeat

- Sender can have up to *N* unacked packets in pipeline
- Receiver sends *individual ack* for each packet
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only that unacked packet

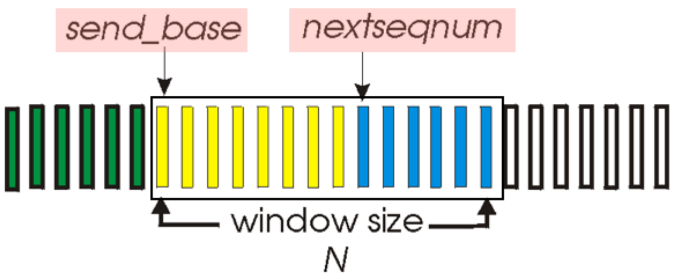
Go-Back-N: sender

- *K*-bit seq # in pkt header *(each pkt is assigned a seq# (n), length k = $\log_2 n$)*
- "Window size" of up to *N*, consecutive unack'ed pkts allowed



- ACK(*n*): ACKs all pkts up to, including seq # *n* - *"cumulative ACK"* (eg. *n*=7, ACK(7) means pkts with seq#0, 1,2,…7 are received)
  - May receive duplicate ACKs (see receiver) (eg. ACK(5), ACK(5), …pkt might be lost)
- Timer for the oldest in-flight pkt (only one timer)
- *Timeout*(*n*): retransmit packet *n* and all higher seq # pkts (yellow part) in window

GBN: sender extended FSM



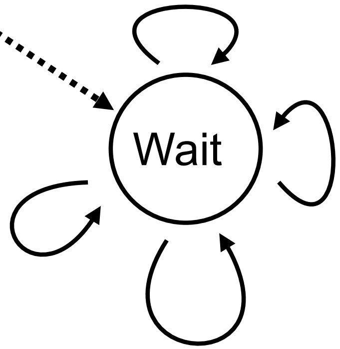
rdt_send(data)

```
if (nextseqnum < send_base+N) {
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (send_base == nextseqnum)
     start_timer (the oldest in-flight pkt)
   nextseqnum++
   }
else
  refuse_data(data)
```

* Λ

---

send_base=1
nextseqnum=1



timeout

```
start_timer
udt_send(sndpkt[send_base])
udt_send(sndpkt[send_base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

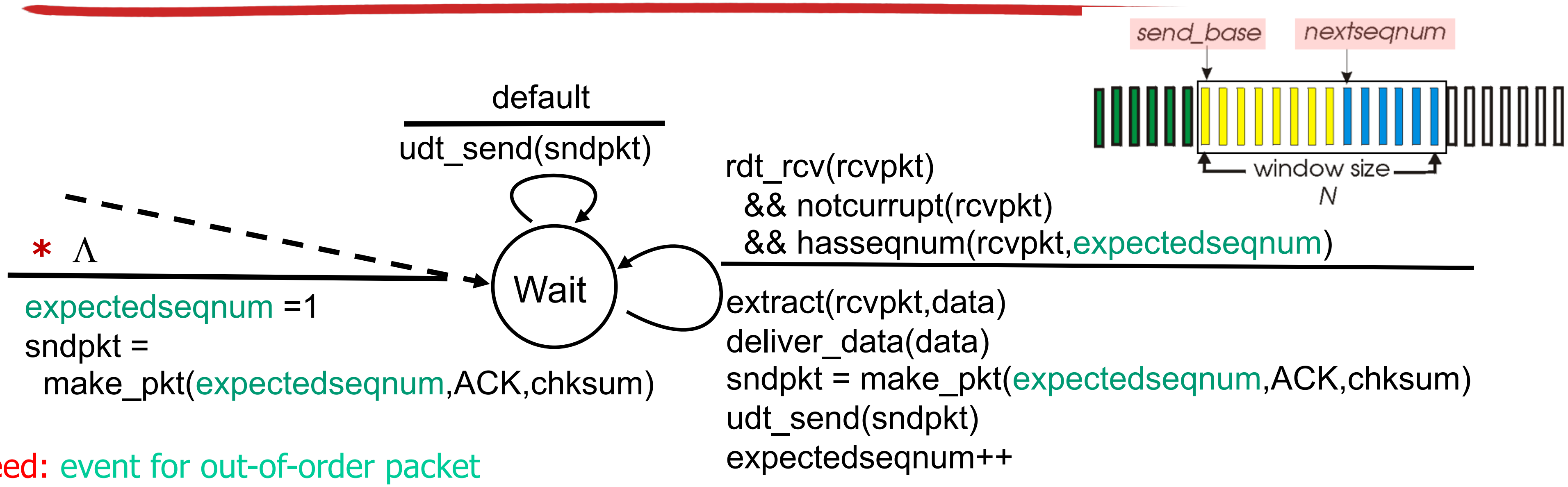
---

drop

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

```
send_base = getacknum(rcvpkt)+1
If (send_base == nextseqnum)
   stop_timer (all in-flight pkts arrived)
 else
   start_timer (the oldest in-flight pkt)
```

GBN: receiver extended FSM



Need: event for out-of-order packet
(standard not specified)

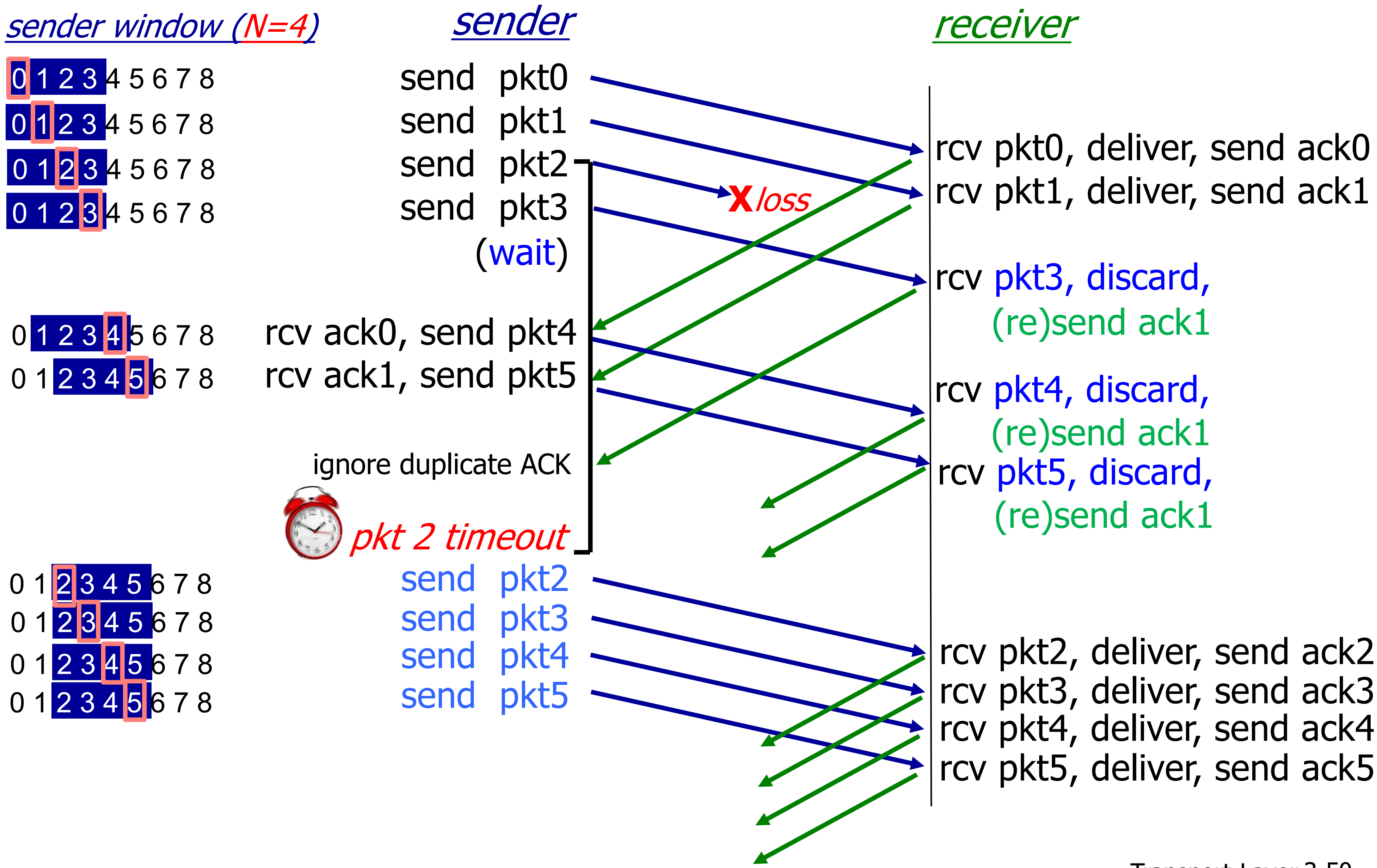
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- May generate duplicate ACKs (eg. ACK(5), ACK(5), …pkt might be lost)
- Receiver needs only remember **expectedseqnum**

❖ Out-of-order pkt

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

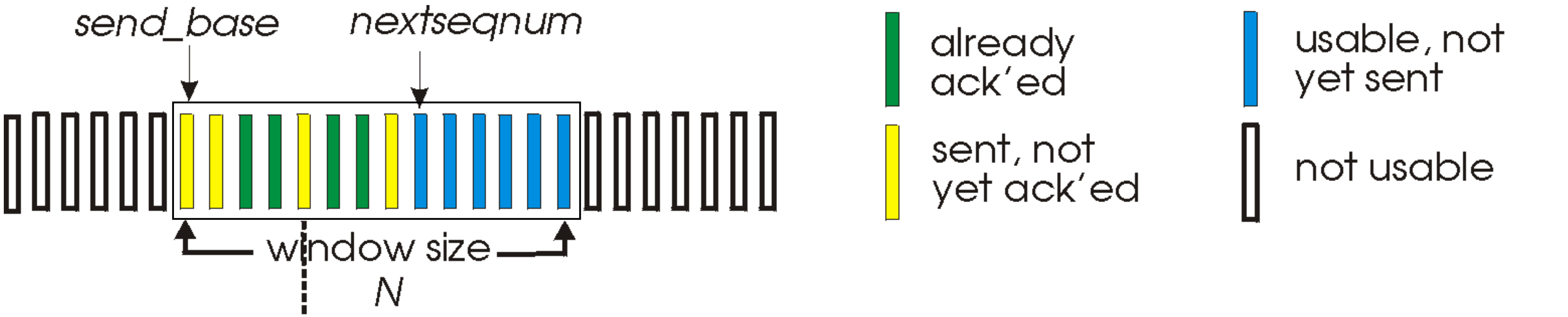
GBN in action



Selective repeat

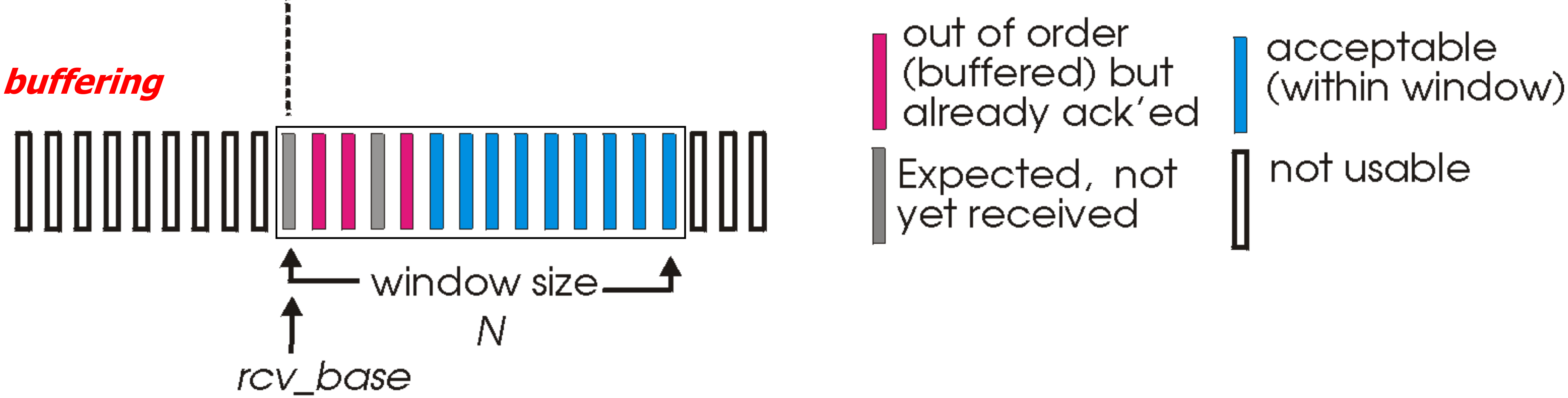
- Receiver *individually* acknowledges all correctly received pkts
  - Buffer pkts, as needed, for eventual in-order delivery to upper layer
- Sender only resends pkts for which ACK not received
  - Sender timer for each unACKed pkt
- Sender window
  - *N* consecutive seq #'s
  - Limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



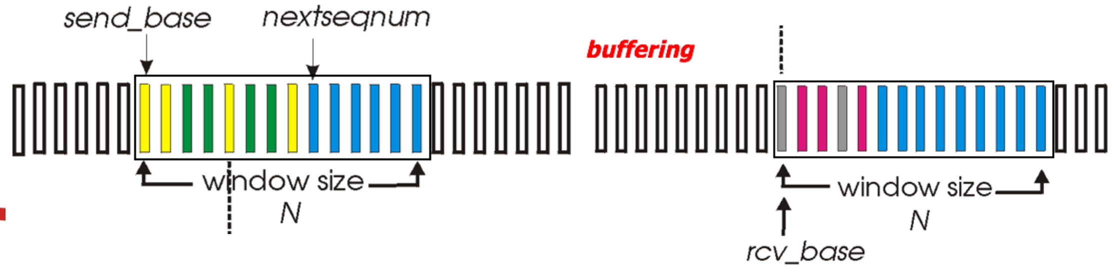
(a) sender view of sequence numbers

buffering



(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 [*send_base, send_base+N*]

- Mark pkt *n* as received
- [Sliding sender window] if *n* is the smallest unACKed pkt, advance window base to next unACKed seq #

Receiver

Rceive pkt *n* in [*rcv_base, rcv_base+N-1*]

- Send ACK(*n*)
- Out-of-order: buffer
- [Sliding receiver window] in-order: deliver (also deliver buffered, in-order pkts), advance window base to next not-yet-received pkt

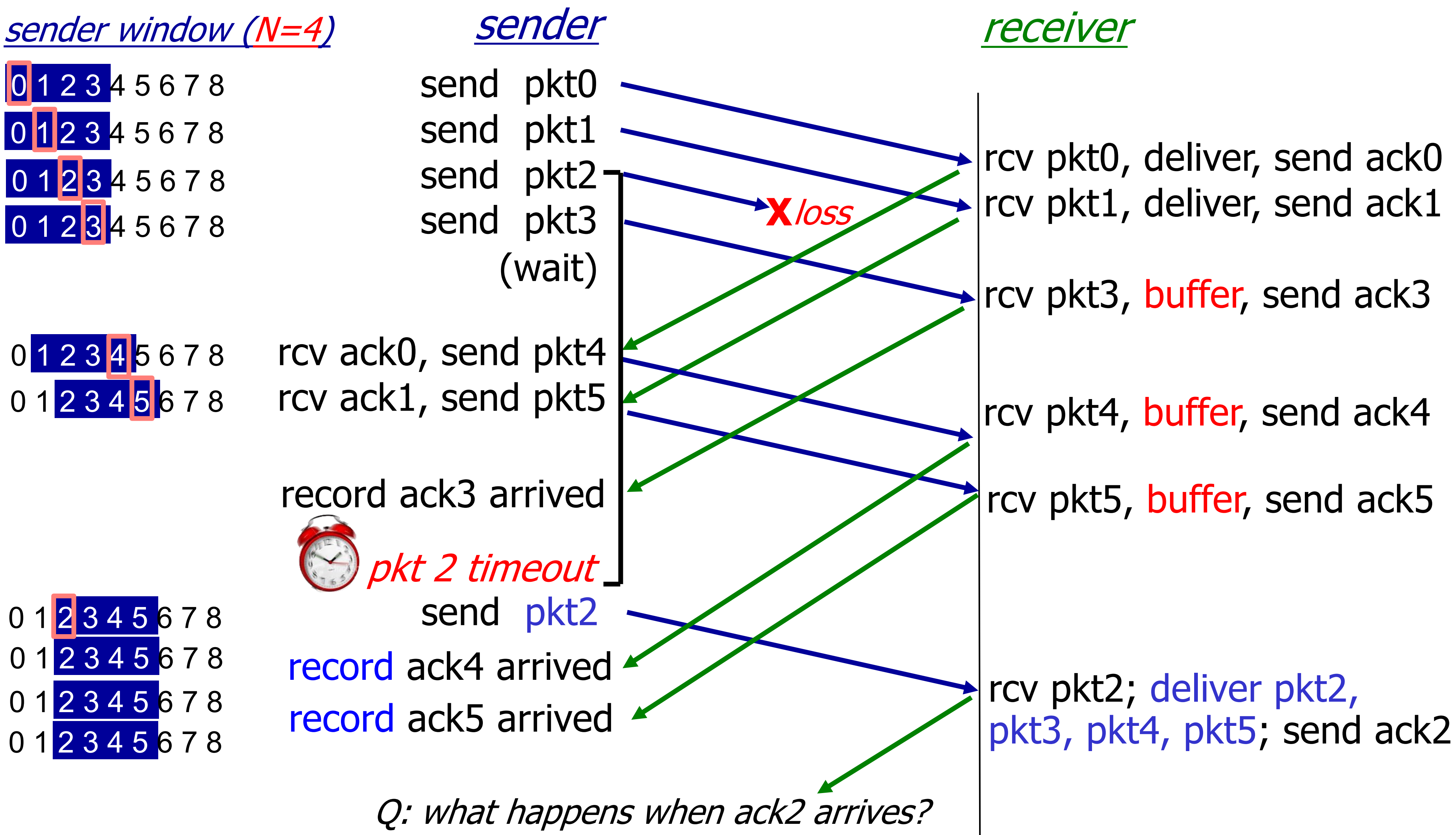
Pkt *n* in [*rcv_base-N,rcv_base-1*]

- ACK(*n*) [delay packet]

Otherwise

- Ignore

Selective repeat in action



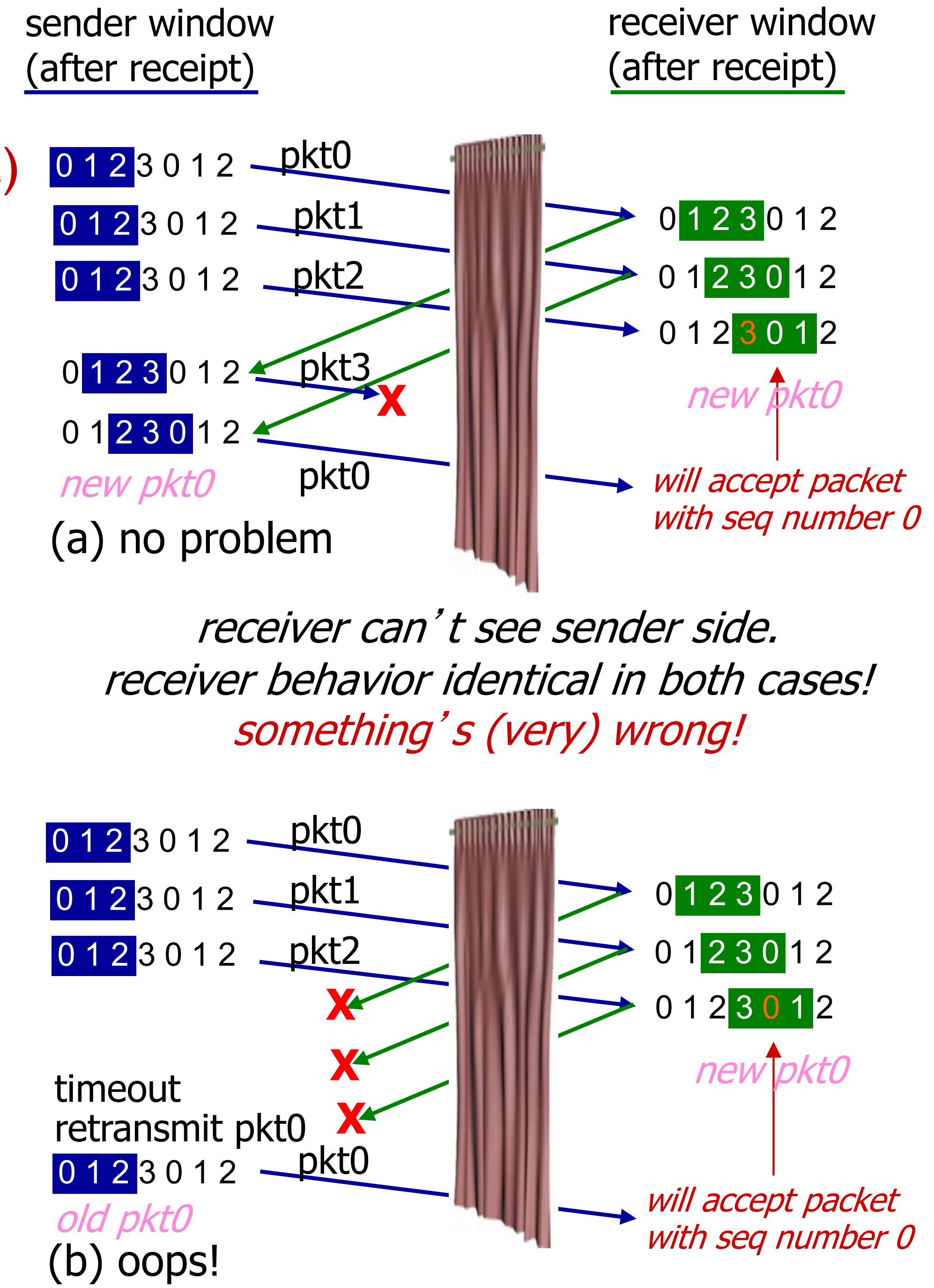
Selective repeat: dilemma (win size vs. seq. no.)

Example:

- Seq #'s: 0, 1, 2, 3
- Window size=3
- Receiver sees no difference in two scenarios!
- Take retransmitted data accepted as new data in (b)

Q: What relationship between seq # size and window size to avoid problem in (b)?

sequence number space ≧ 2 x window size



this is old *pkt0*, which is not expected by receiver (new *pkt0*)

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 stream*
  - No "message boundaries" (doesn't matter how many packets)
  - Byte stream based, each byte is assigned a seq#
- Pipelined
  - TCP congestion and flow control set window size (bytes)
- Send & receive buffers (selective repeat)
- Full duplex data
  - Bi-directional data flow in same connection
  - MSS: maximum segment size (TCP transmission data unit) – MSS should be determined at the time the connection is setup
- Connection-oriented
  - Handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- Flow controlled
  - Sender will not overwhelm receiver

TCP segment structure

No IP address fields
(Network Layer header)

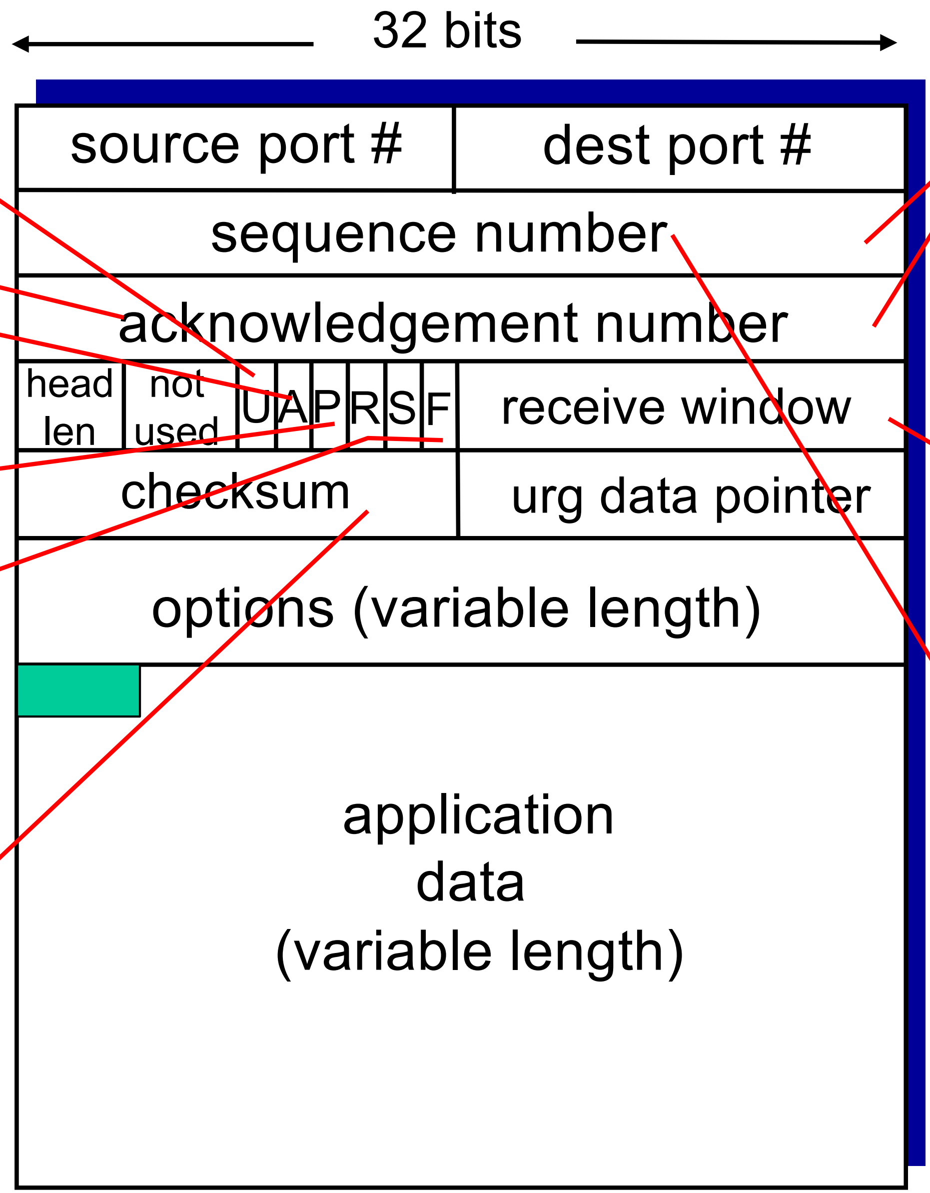
URG: urgent data
(generally not used)

ACK by byte seq#
ACK: “1” :ACK # is valid

PSH: push data to
the upper layer now
(otherwise buffer bytes)

RST, SYN, FIN:
(R) reset, (S)
connection estab
(sync (setup)), (F)
teardown commands

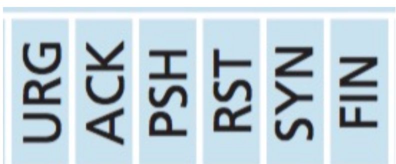
Internet
checksum
(as in UDP)



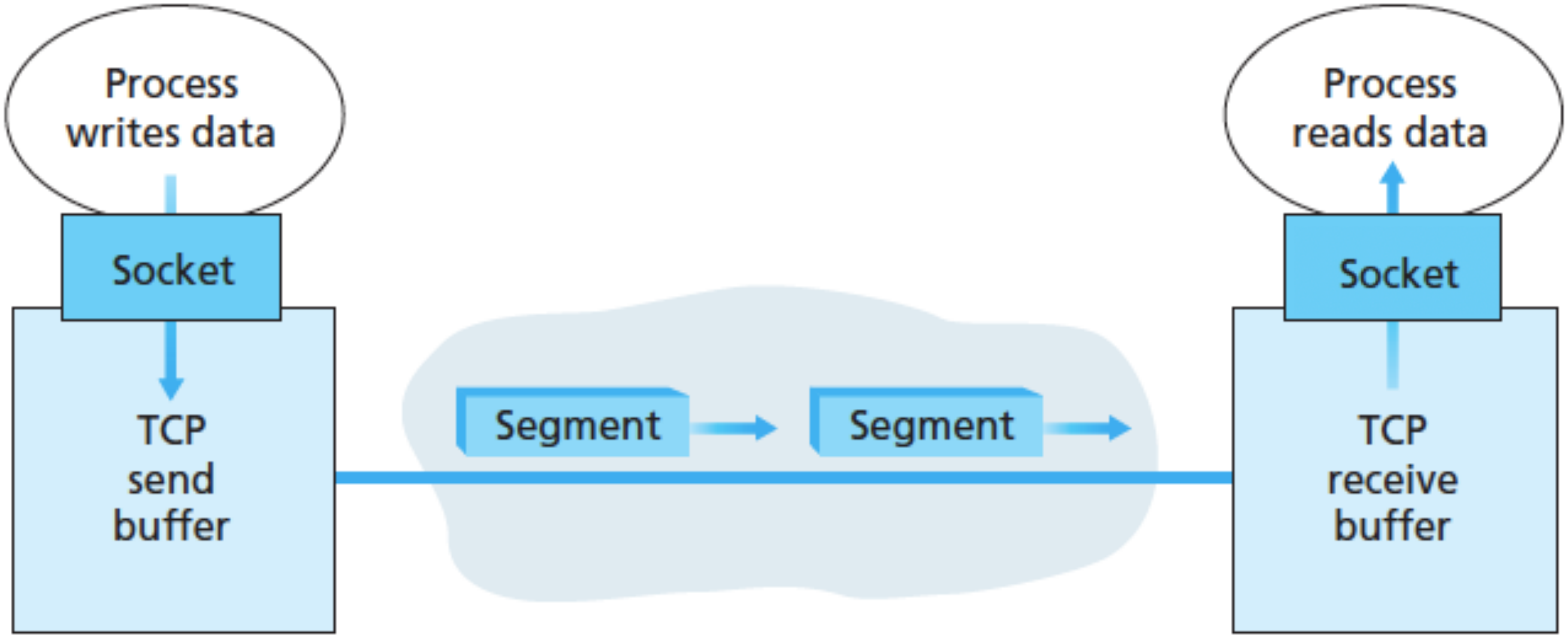
counting
by bytes
of data
(not segments!)

bytes
rcvr willing
to accept
(0-64K,
flow control)

the seq# of the
first data byte



TCP send and receive buffers

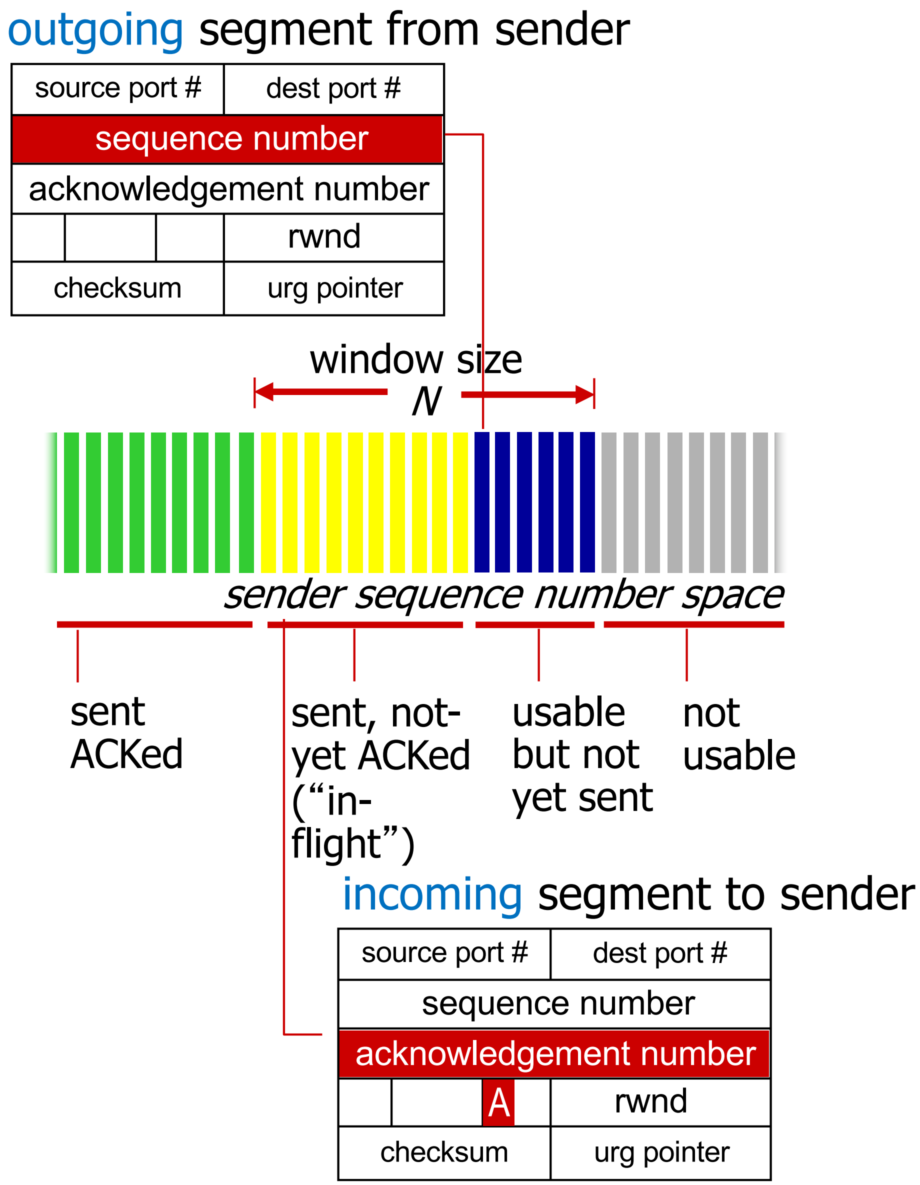


TCP seq. numbers, ACKs

Sequence numbers
- Byte stream "number" of first byte in segment's data
- Each byte is associated with a seq #

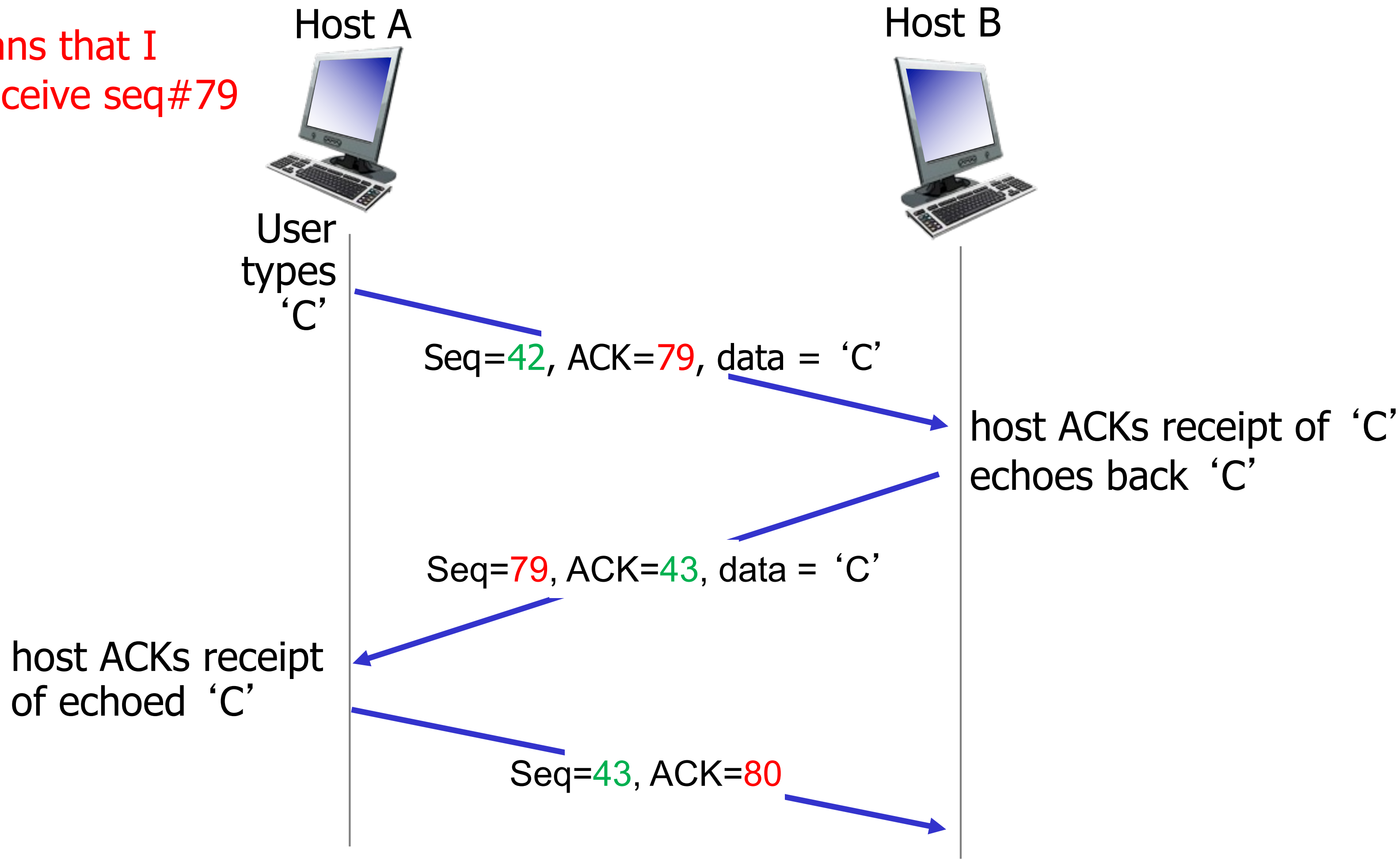
Acknowledgements
- Seq # of <u>next byte expected</u> from other side
- Cumulative ACK (Go-Back-N)

Q: How receiver handles <u>out-of-order</u> segments
- A: TCP spec <u>doesn't</u> say, - up to implementer



TCP seq. numbers, ACKs

ACK-79 means that I expect to receive seq#79



simple telnet scenario

TCP round trip time, timeout

Q: How to set TCP timeout value?

- Longer than RTT
  - But RTT varies
- *Timer too short*: premature timeout, unnecessary retransmissions
- *Timer too long*: slow reaction to segment loss

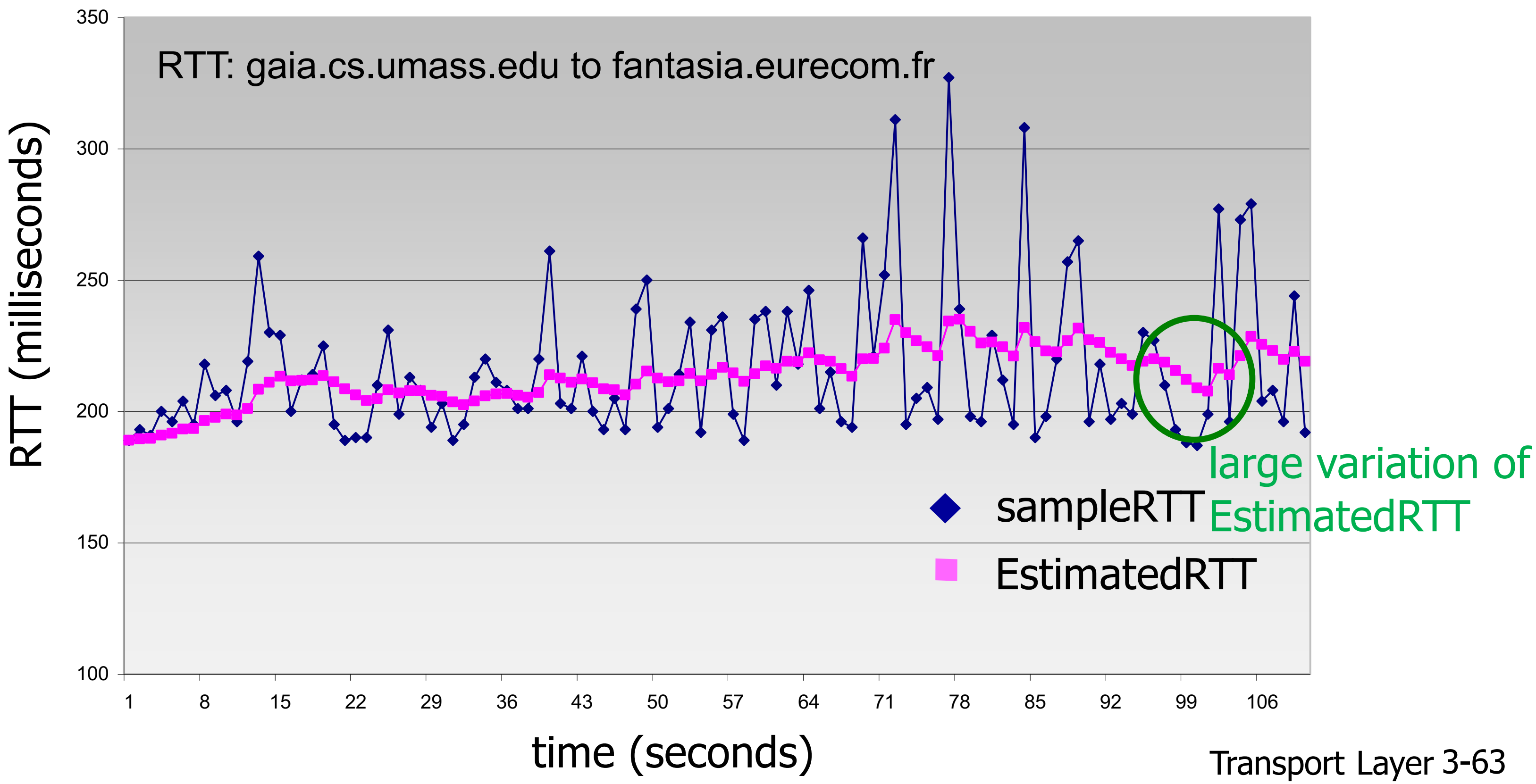
Q: How to estimate RTT?

- **`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

EstimatedRTT = (1-α)*EstimatedRTT + α*SampleRTT

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: α = 0.125



TCP round trip time, timeout

- Timeout interval: **`EstimatedRTT`** plus "safety margin"
  - Large variation in **`EstimatedRTT`** → larger safety margin
- Estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|

              (typically, β = 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 [not stop and wait]
  - Cumulative acks [ack(*n*) is "expecting" pkt *n*]
  - Single retransmission timer [retransmit only one pkt]
- Retransmissions triggered by
  - Timeout events
  - Duplicate acks [pkts may be lost]

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 the byte-stream number of the first data byte in segment
- Start timer if not already running (only one timer)
  - Think of timer as for oldest unacked segment
  - Expiration interval: `TimeOutInterval`

Timeout

- Retransmit the oldest unacked segment
- Restart timer

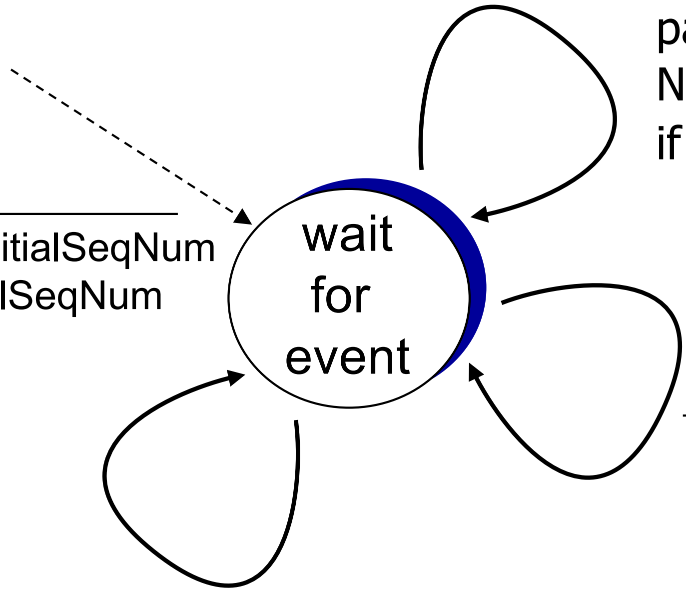
Ack rcvd

- If ack acknowledges previously unacked segments
  - Update what is known to be ACKed
  - Start timer if there are still unacked segments

Simplified TCP sender

```
/* Assume sender is not constrained by TCP flow or congestion control, that data from above is less
than MSS in size, and that data transfer is in one direction only. */

NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

    } /* end of loop forever */
```

TCP sender (simplified)



* Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

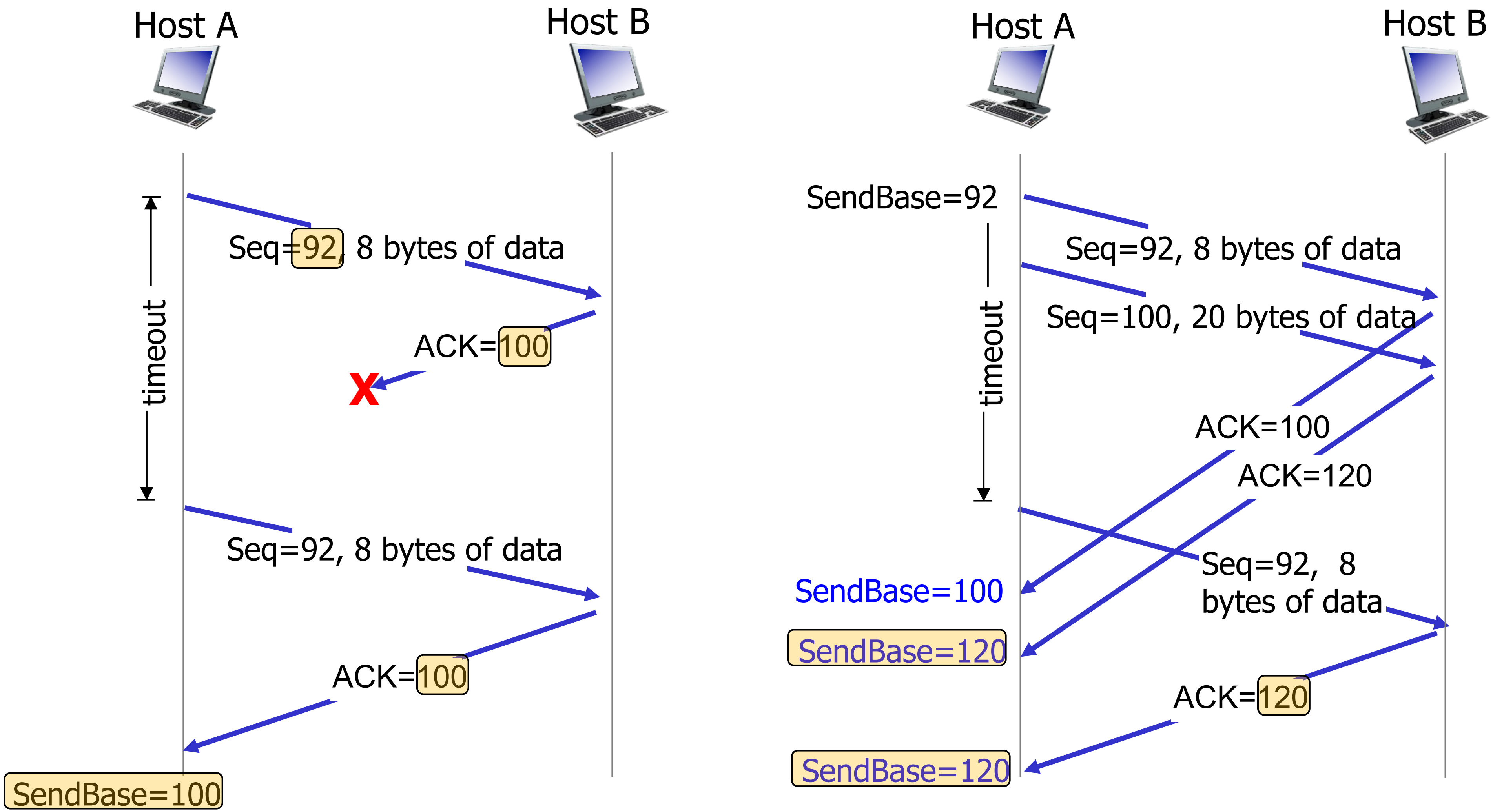
timeout

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

ACK received, with ACK field value y

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

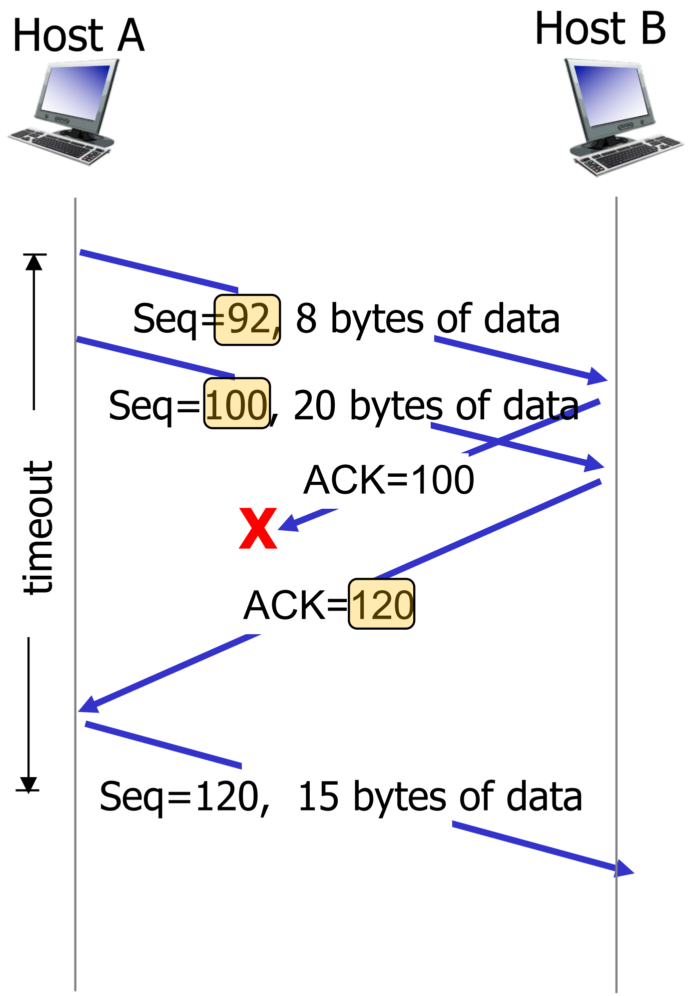
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



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. [delayed ACK]	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 [imply pkts lost]
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

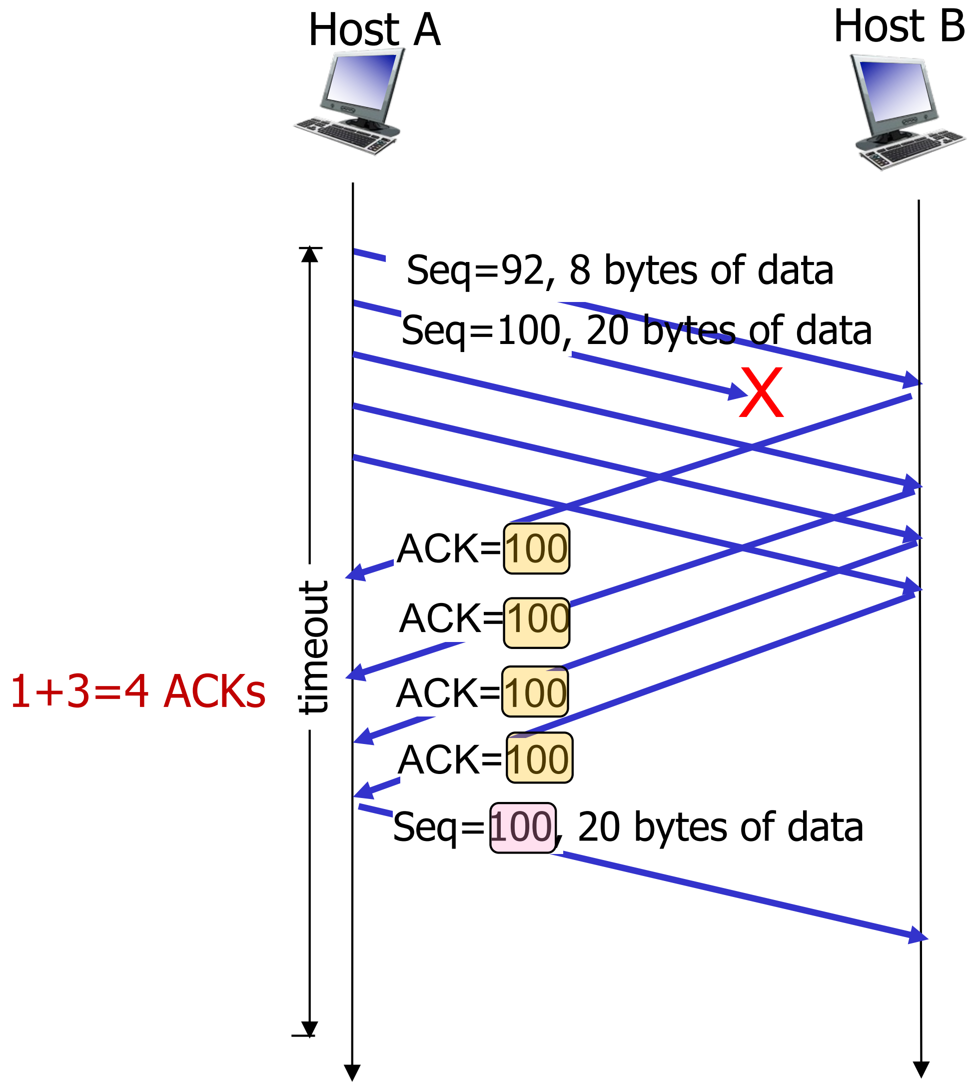
If sender receives 3 ACKs for same data ("triple duplicate ACKs" - (1+3=4 ACKs)), resend unacked segment with smallest seq #

- Likely that unacked segment lost, so don't wait for timeout

Fast retransmit algorithm

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
                SendBase=y
                if (there are currently any not yet
                           acknowledged segments)
                   start timer
                }
          else { /* a duplicate ACK for already ACKed
                segment */
             increment number of duplicate ACKs
                received for y
             if (number of duplicate ACKS received
                for y==3)
                /* TCP fast retransmit */
                resend segment with sequence number y
             }
          break;
```

TCP fast retransmit



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

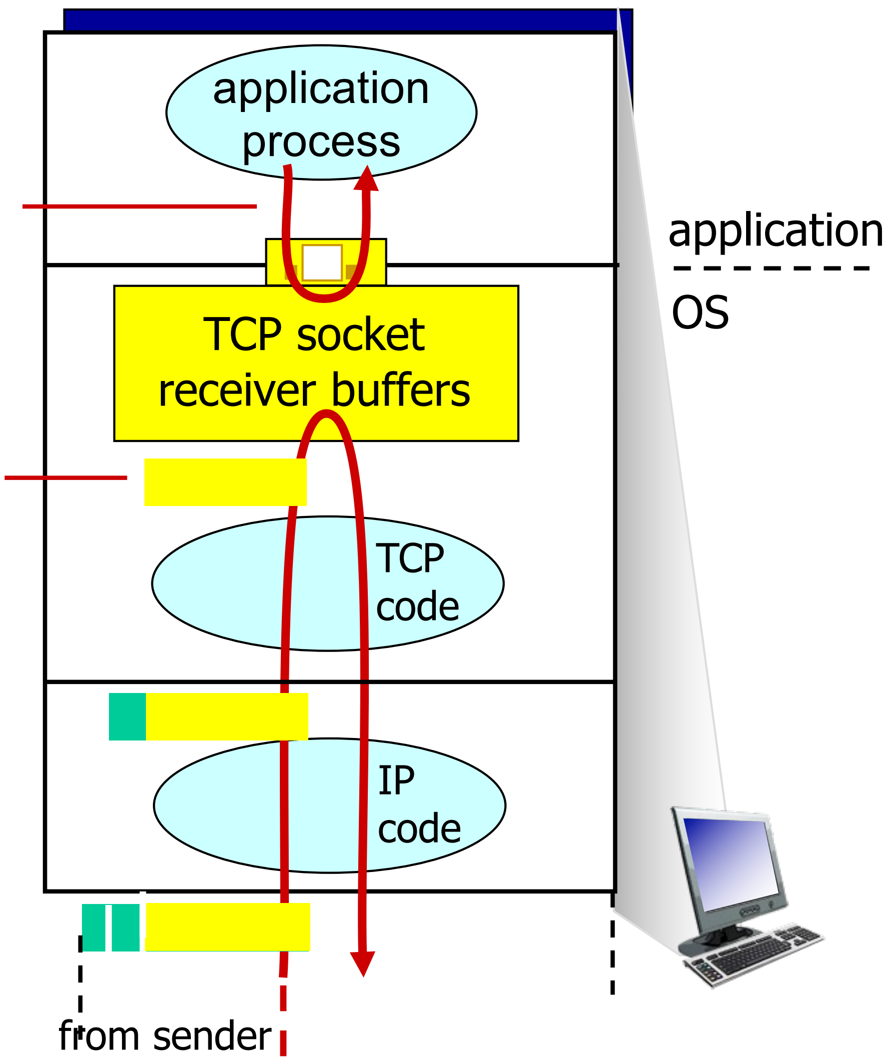
3.6 Principles of congestion control

3.7 TCP congestion control

TCP flow control

application may remove data from TCP socket buffers ….

sender is sending slower than TCP receiver is delivering



receiver protocol stack

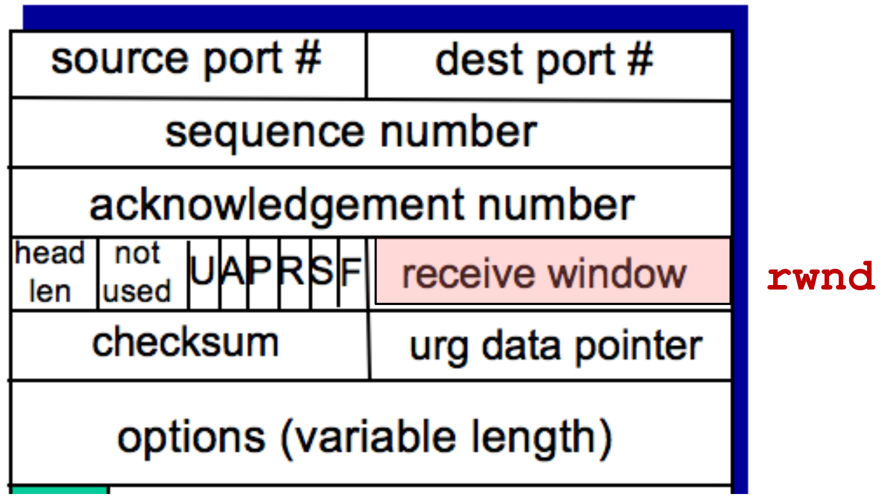
Flow control

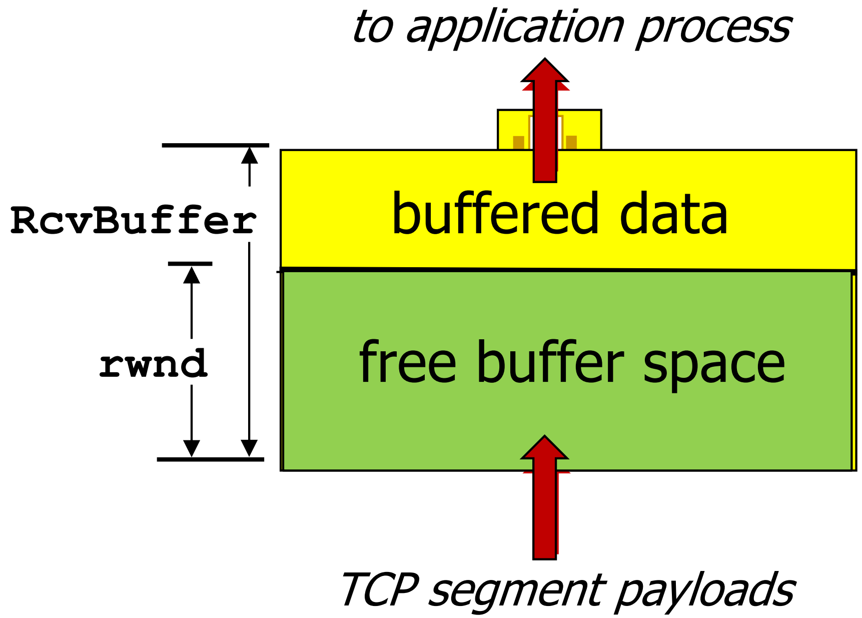
Receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

- Receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of <u>receiver-to-sender</u> 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

receiver segment





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

TCP Connection Management

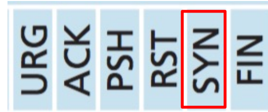
Recall: TCP sender, receiver establish "connection" before exchanging data segments

- Initialize TCP variables
  - Seq #s
  - Buffers, flow control info (e.g. RecWindow - rwnd)
- *Client*: connection initiator

  Socket ***clientSocket*** = *new Socket("hostname", "port number");*
- *Server*: connected by client

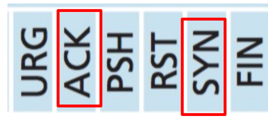
  Socket ***connectionSocket*** = *welcomeSocket.accept();*

Three way handshake



Step 1: client host sends TCP SYN segment to server (set SYN = 1)

- Specifies initial seq#
- No data



Step 2: server host receives SYN, replies with SYNACK (set SYN = 1, ACK = 1)

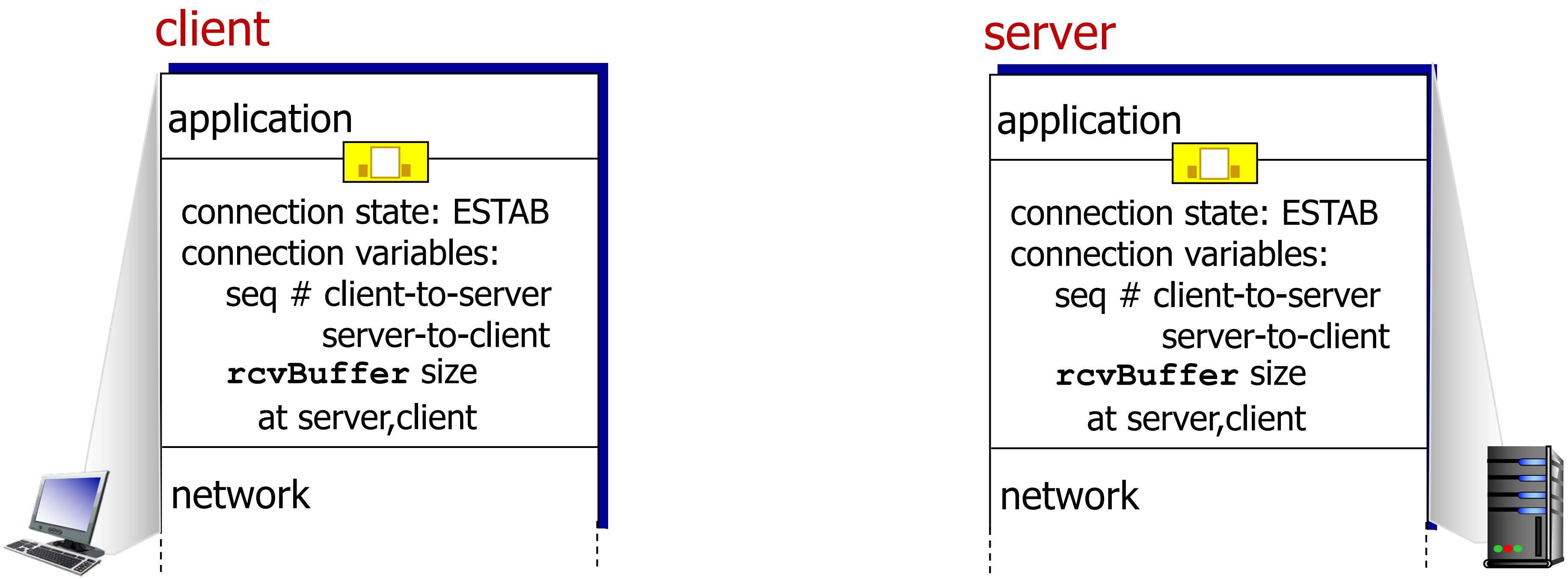
- Server allocates buffers [DOS attack – SYN flooding]
- Specifies server initial seq#

Step 3: client receives SYNACK, replies with ACK segment, which may contain **data**

Connection Management

Before exchanging data, sender/receiver "handshake"

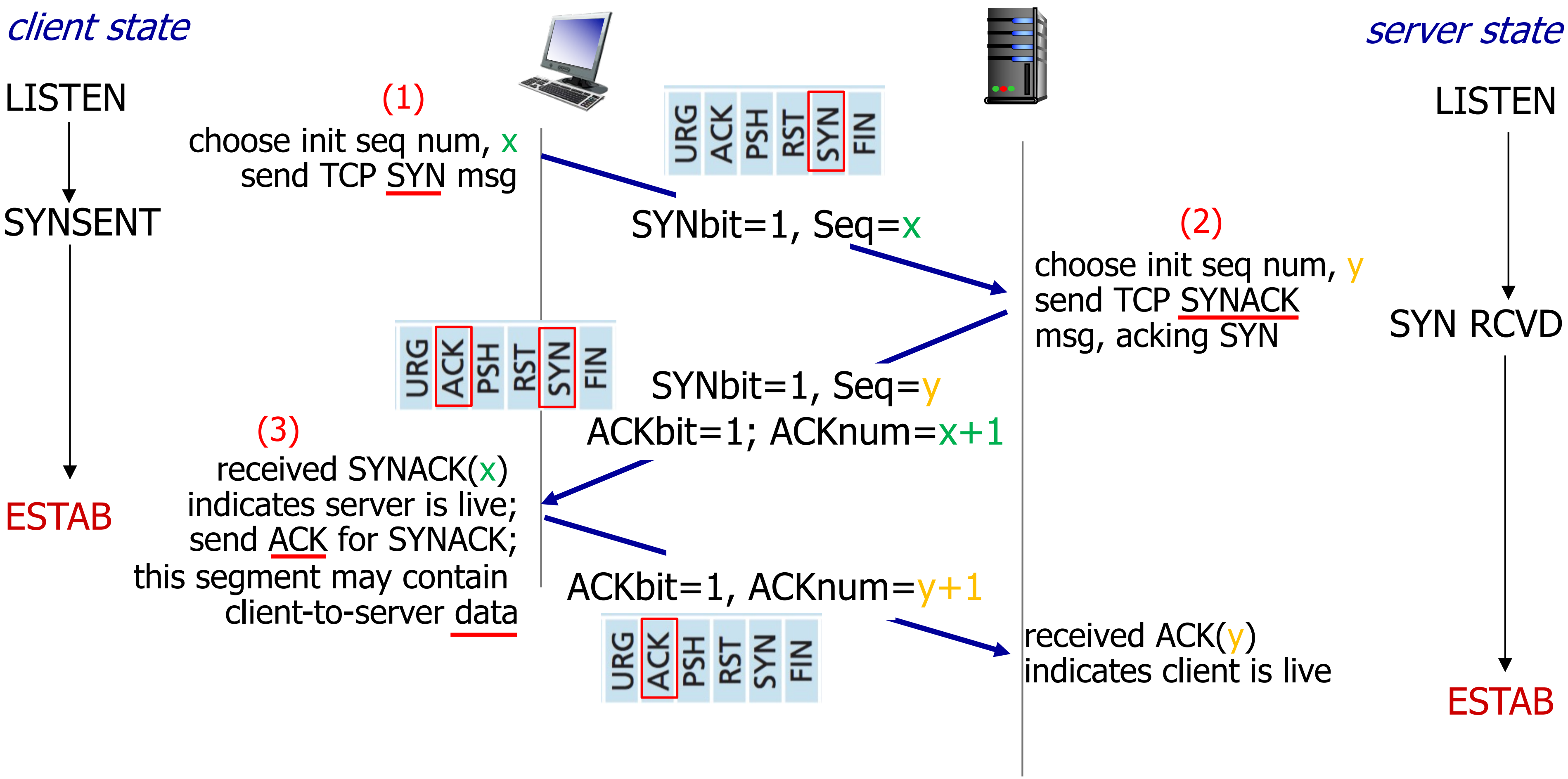
- Agree to establish connection (each knowing the other willing to establish connection)
- Agree on connection parameters



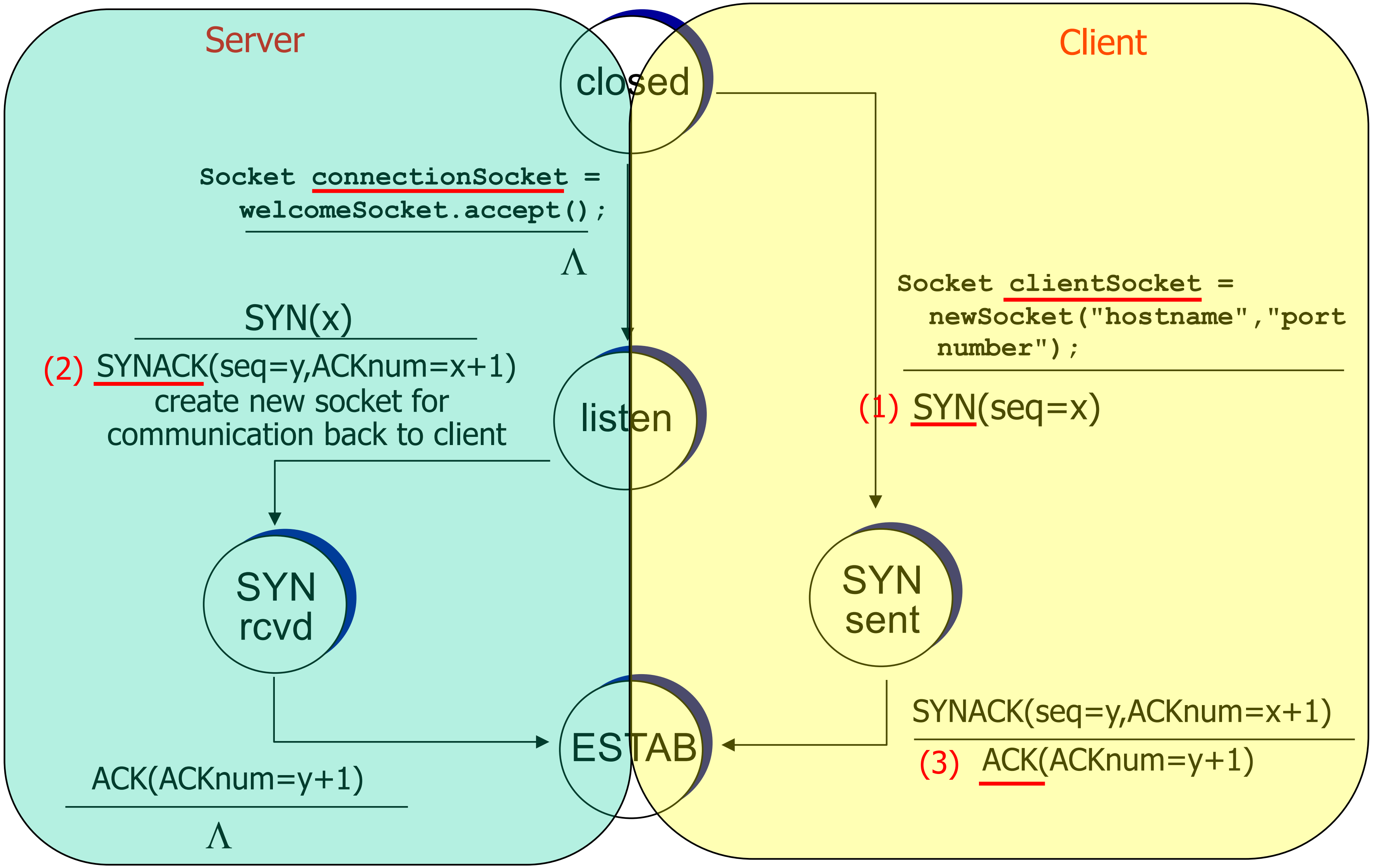

```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

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

TCP 3-way handshake



TCP 3-way handshake: FSM



TCP: closing a connection

- Client, server each close their side of connection
  - Send TCP segment with FIN bit = 1
- Respond to received FIN with ACK
  - On receiving FIN, ACK can be combined with own FIN
- Closing a connection

  Client closes socket: clientSocket.close();

  Step 1: client ends system sends TCP FIN control segment to server

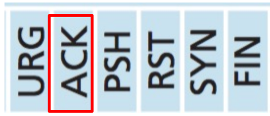
  

  Step 2 :server receives FIN replies with ACK closes connection sends FIN

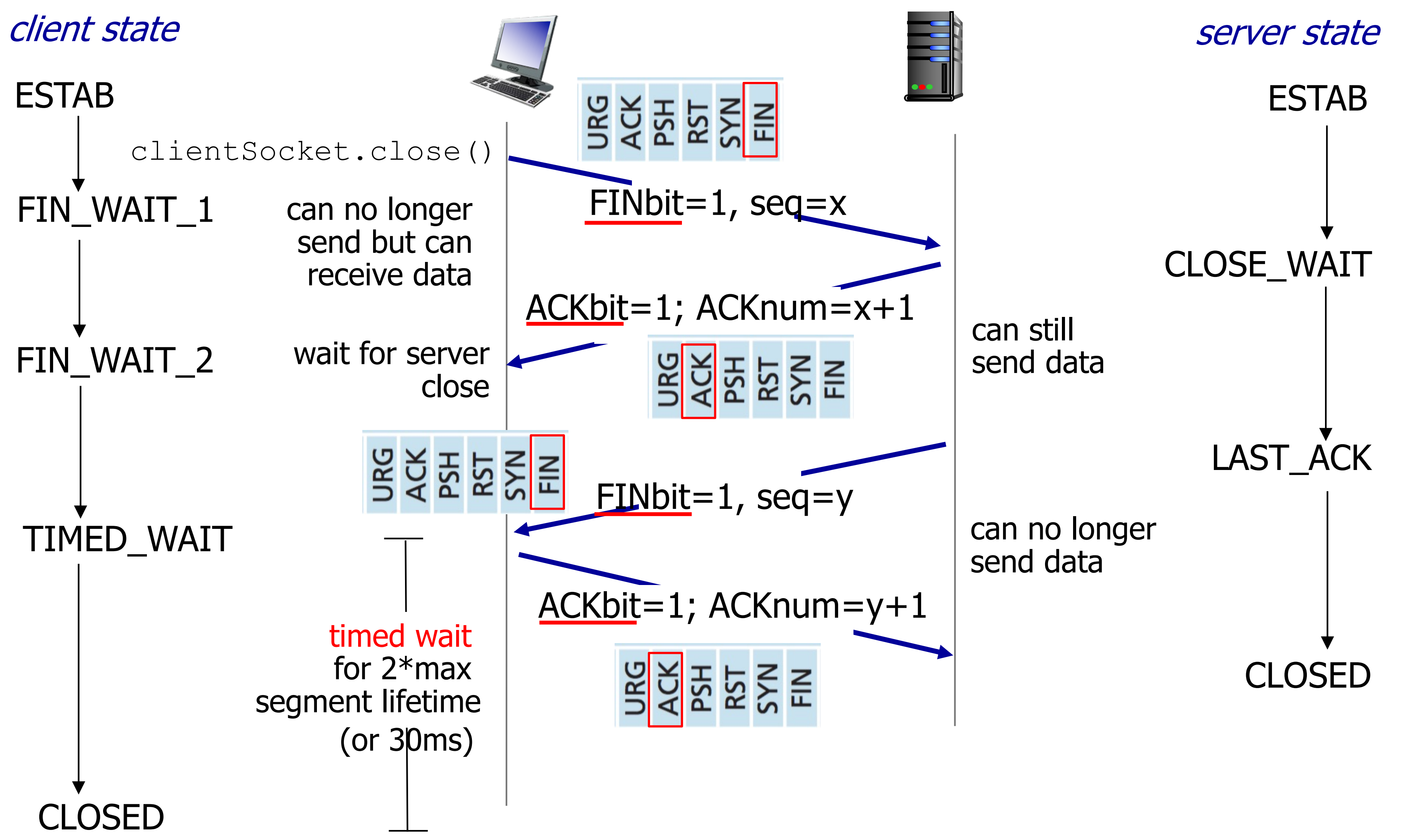
Step 3: client receives FIN
replies with ACK



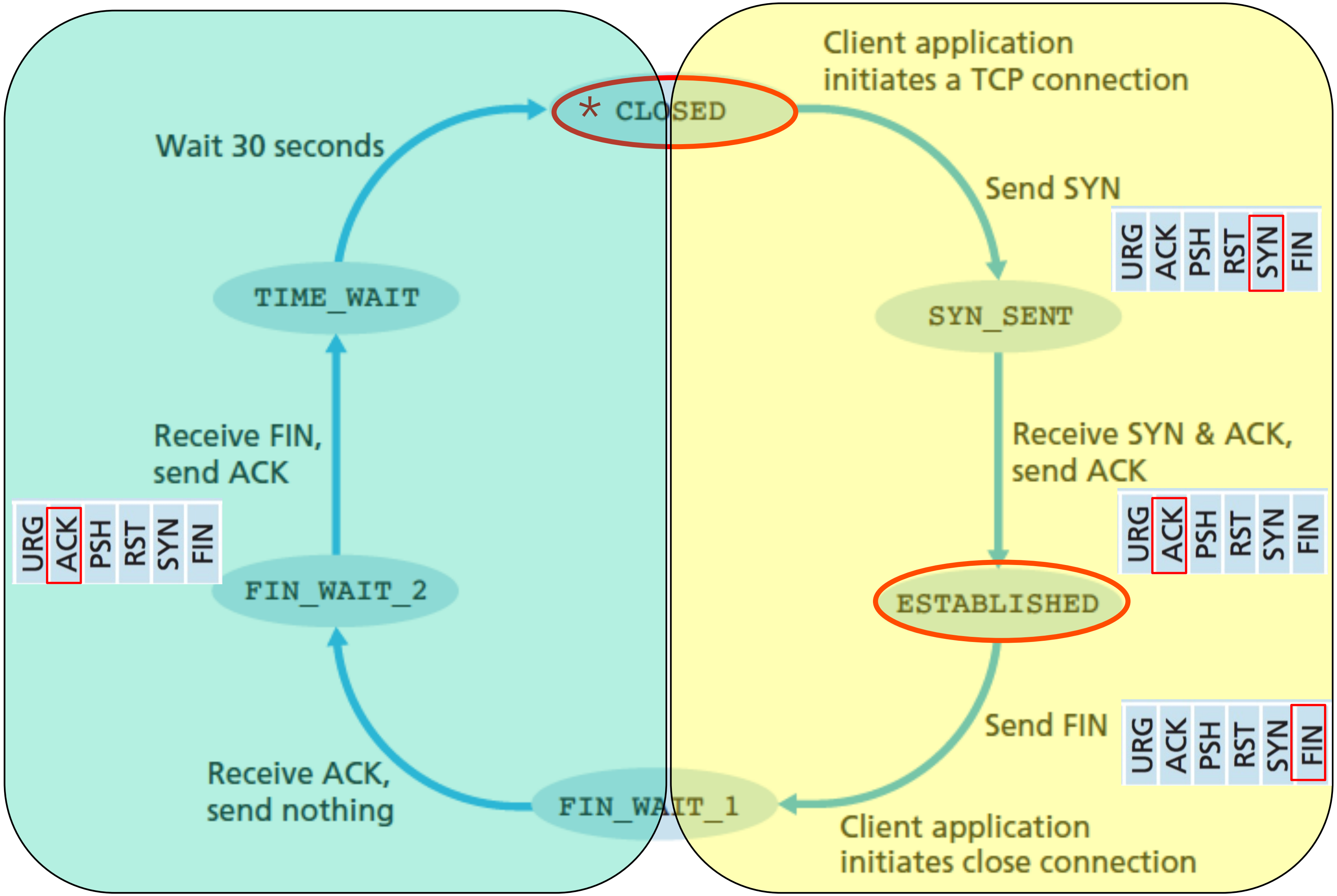
enters "timed wait" – will respond with ACK to received FIN

Step 4 :server receives ACK
connection closed

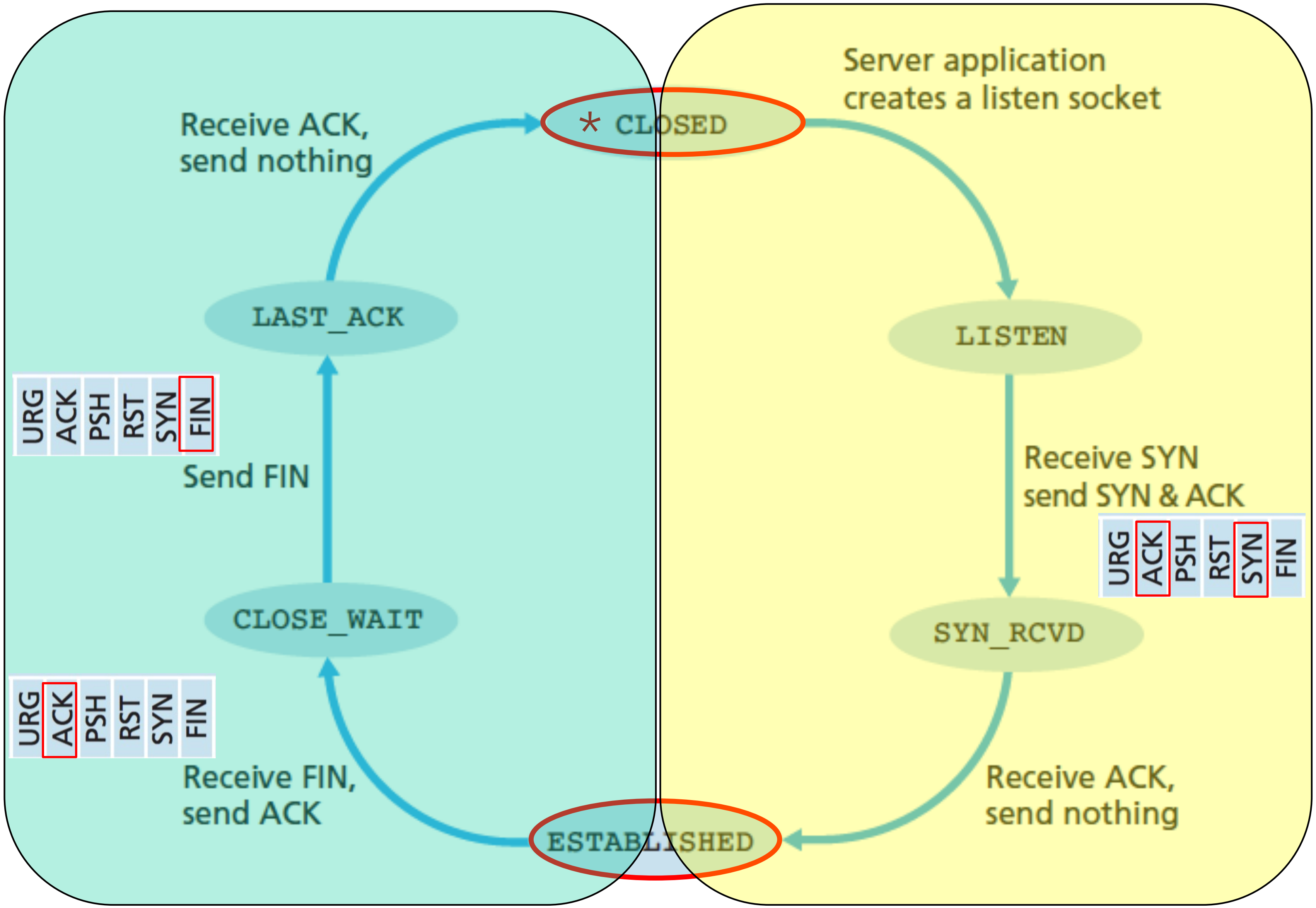
TCP: closing a connection



A typical sequence of TCP states visited by a client TCP



A typical sequence of TCP states visited by a server-side TCP



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

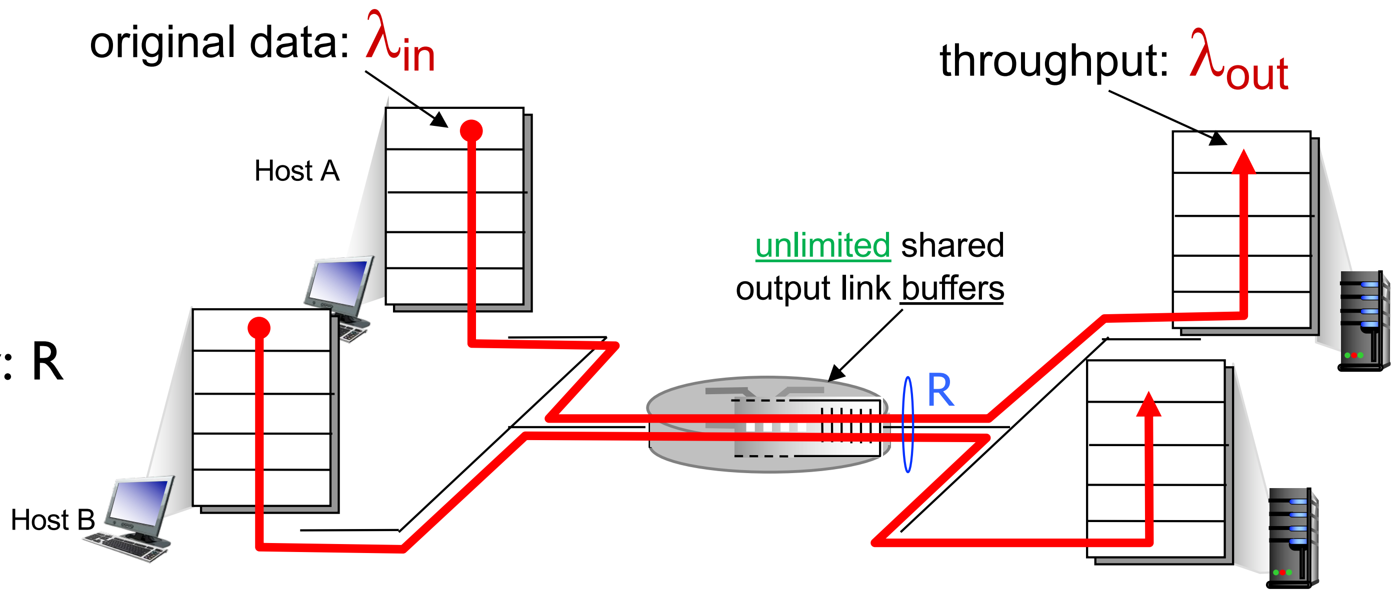
Principles of congestion control

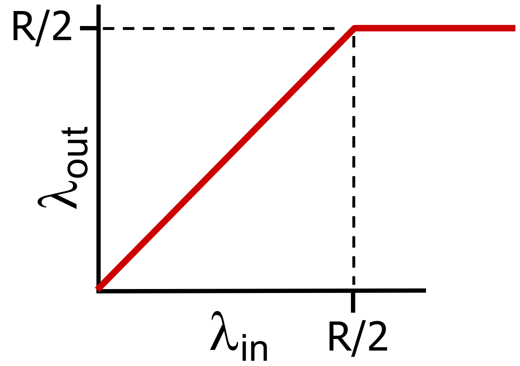
Congestion

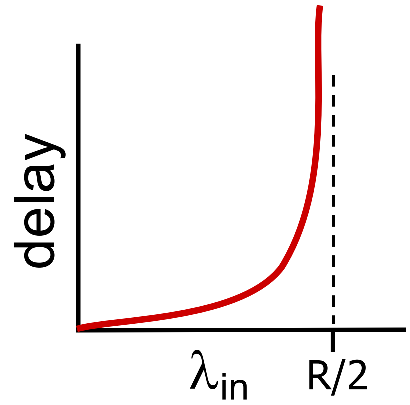
- Informally: "too many sources sending too much data too fast for *network* to handle"
- Different from flow control!
- Manifestations
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)
- A top-10 problem!

Causes/costs of congestion: scenario 1

- Two senders, two receivers
- One router, infinite buffers
- Output link capacity: R
- No retransmission



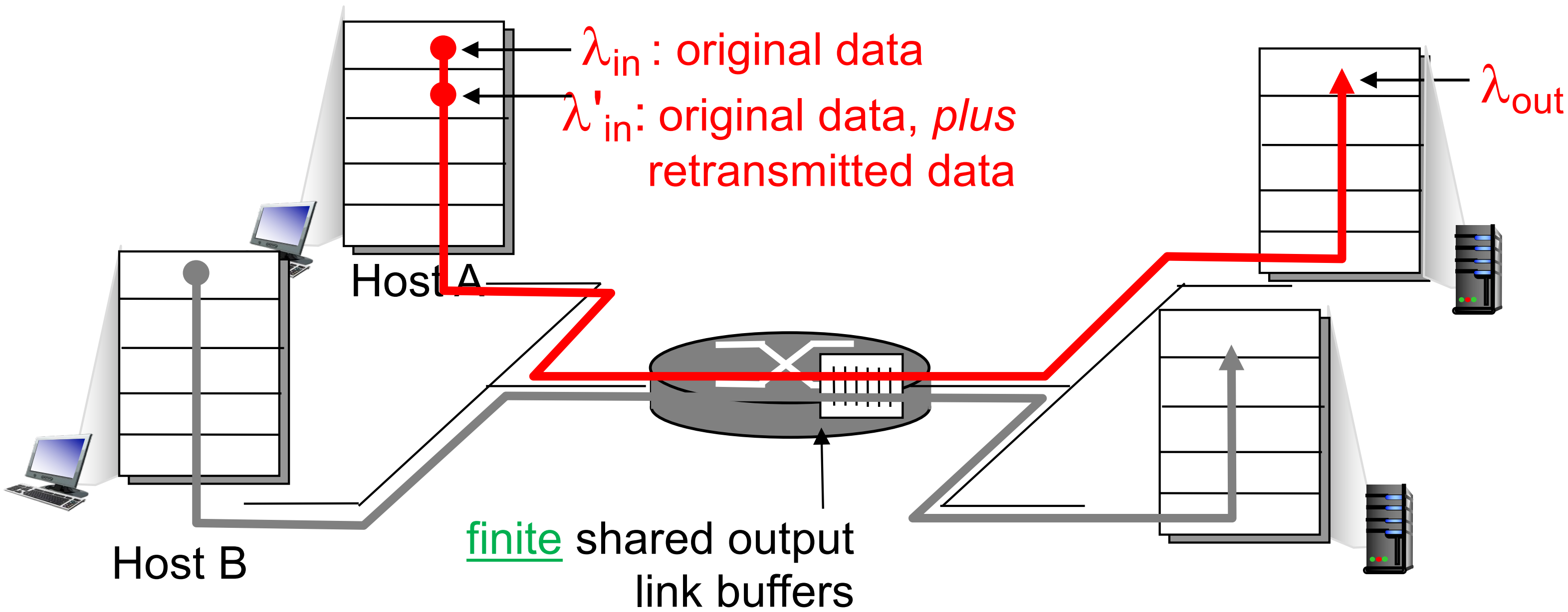




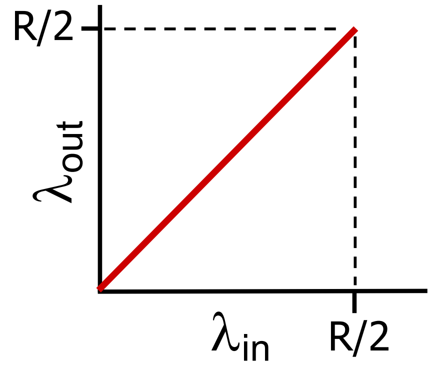
- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, approaches capacity

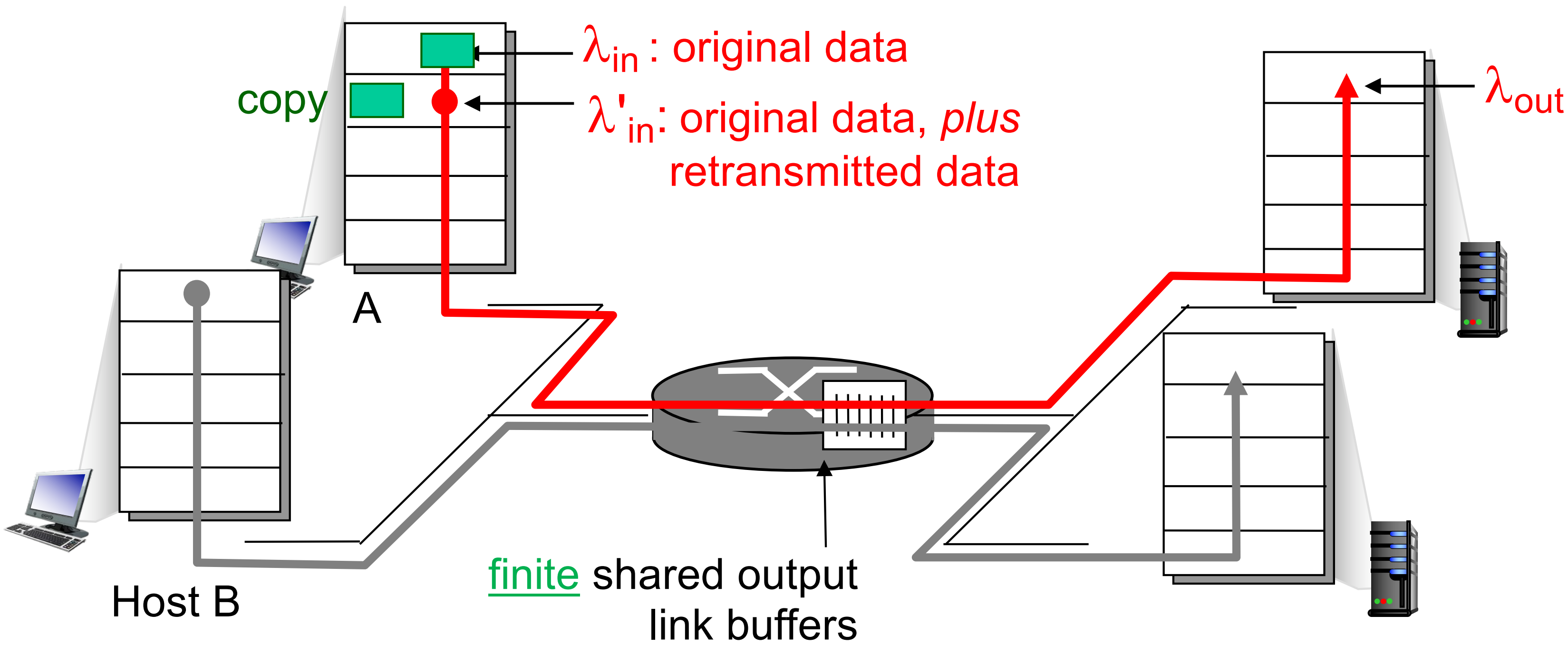
Causes/costs of congestion: scenario 2

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

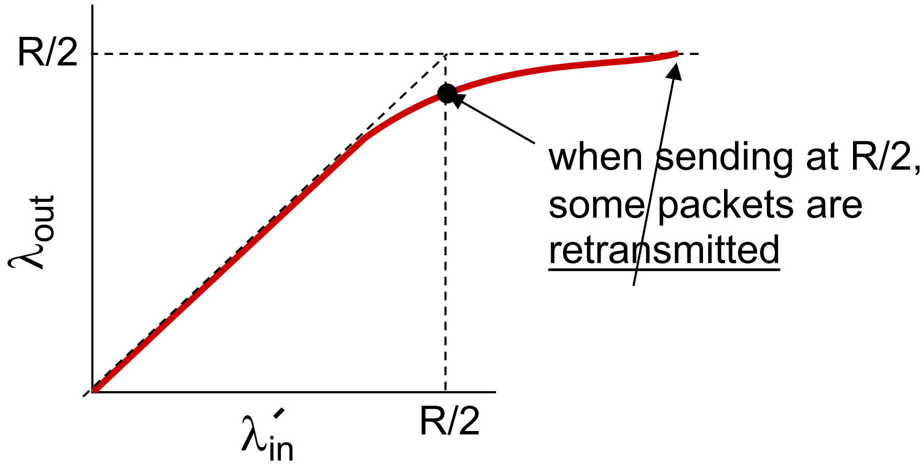


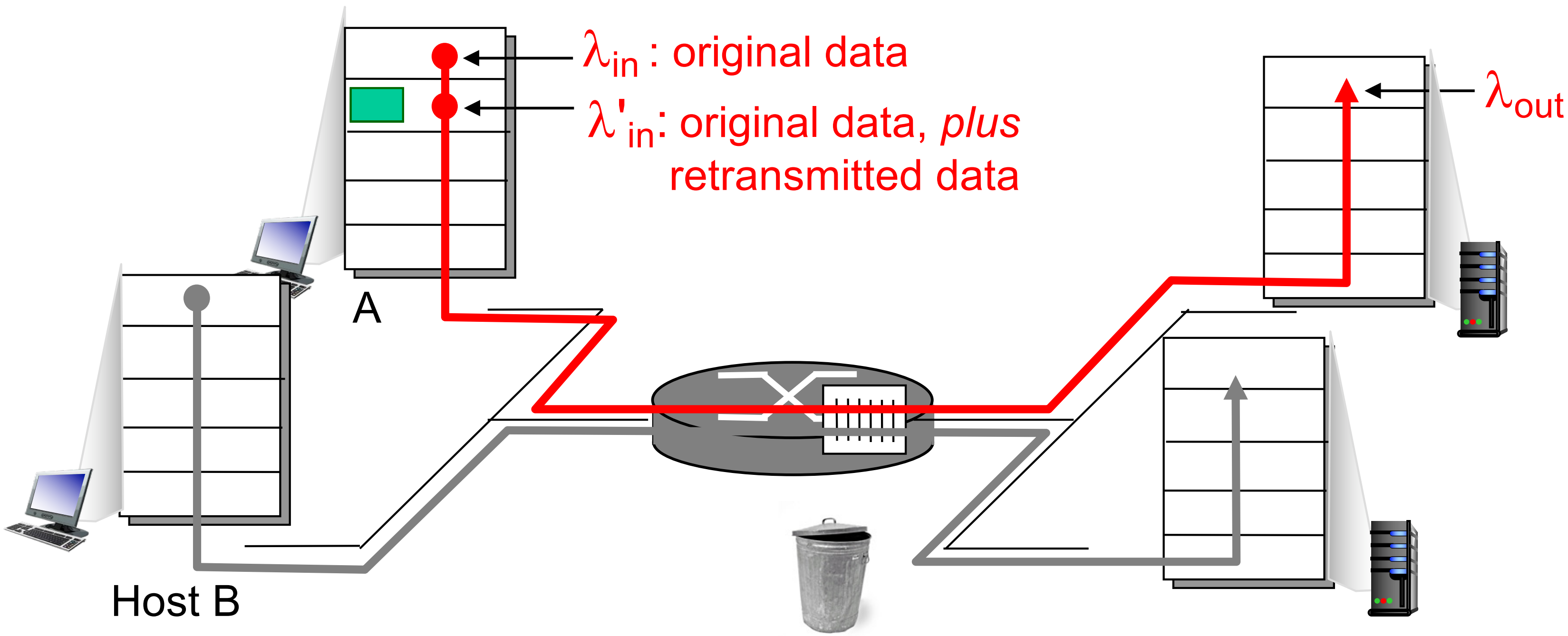
- Sender sends only when router buffers available



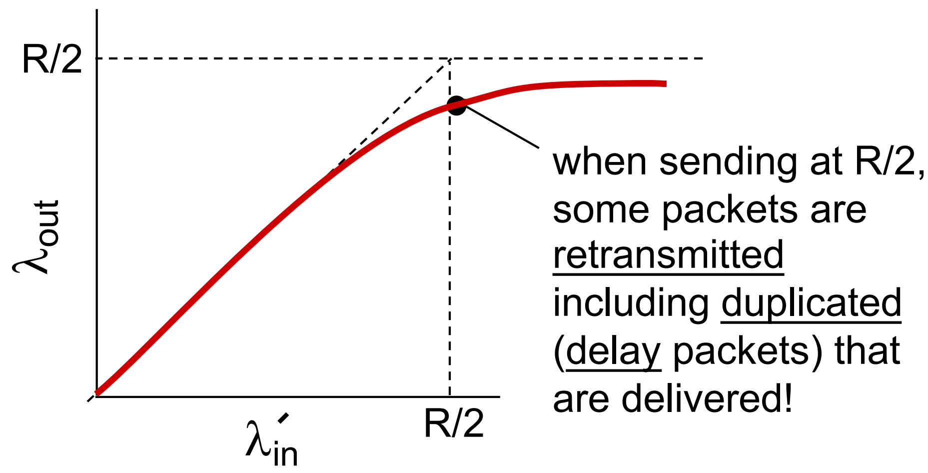


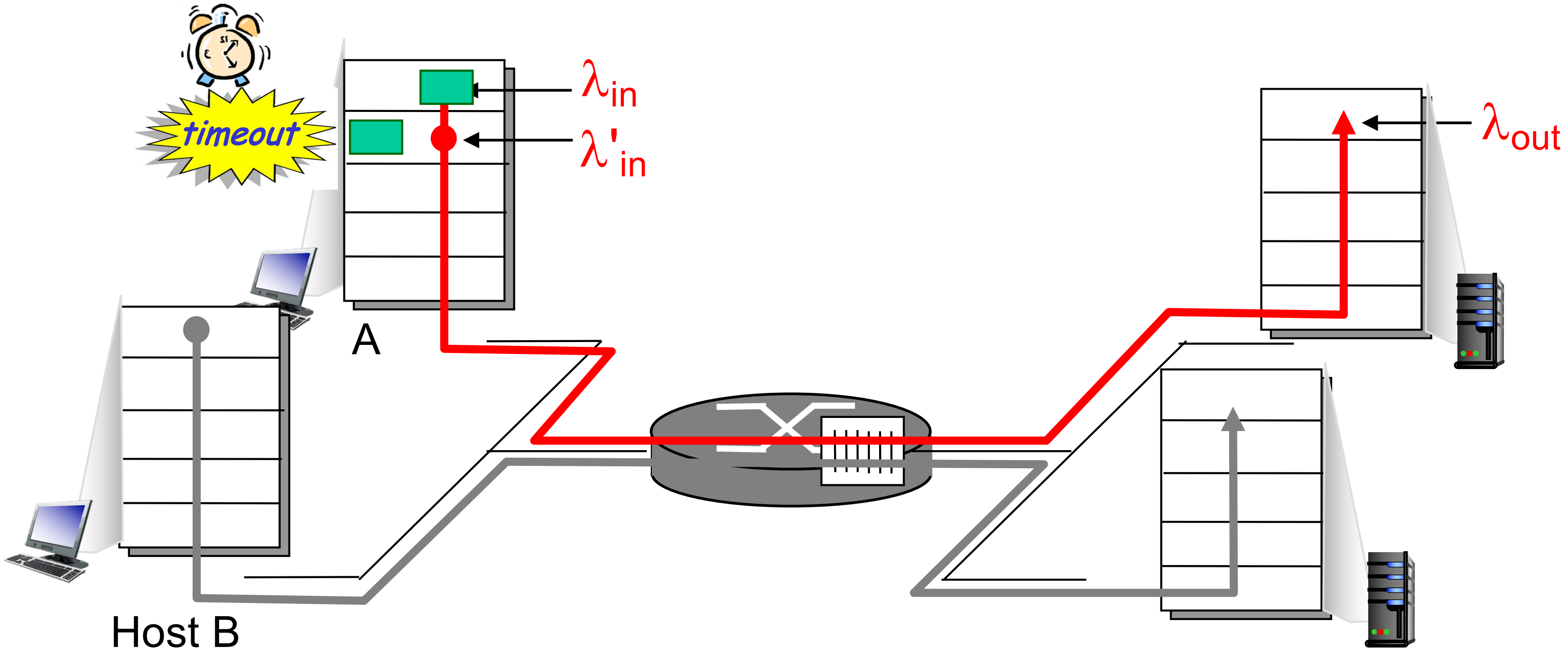
- Packets can be lost, dropped at router due to full buffers
- Sender only resends if packet *known* to be lost





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





"Costs" of congestion

- More work (<u>retrans</u>) for given "goodput"
- <u>Unneeded retransmissions</u>: link carries <u>multiple copies</u> of pkt
  - decreasing goodput

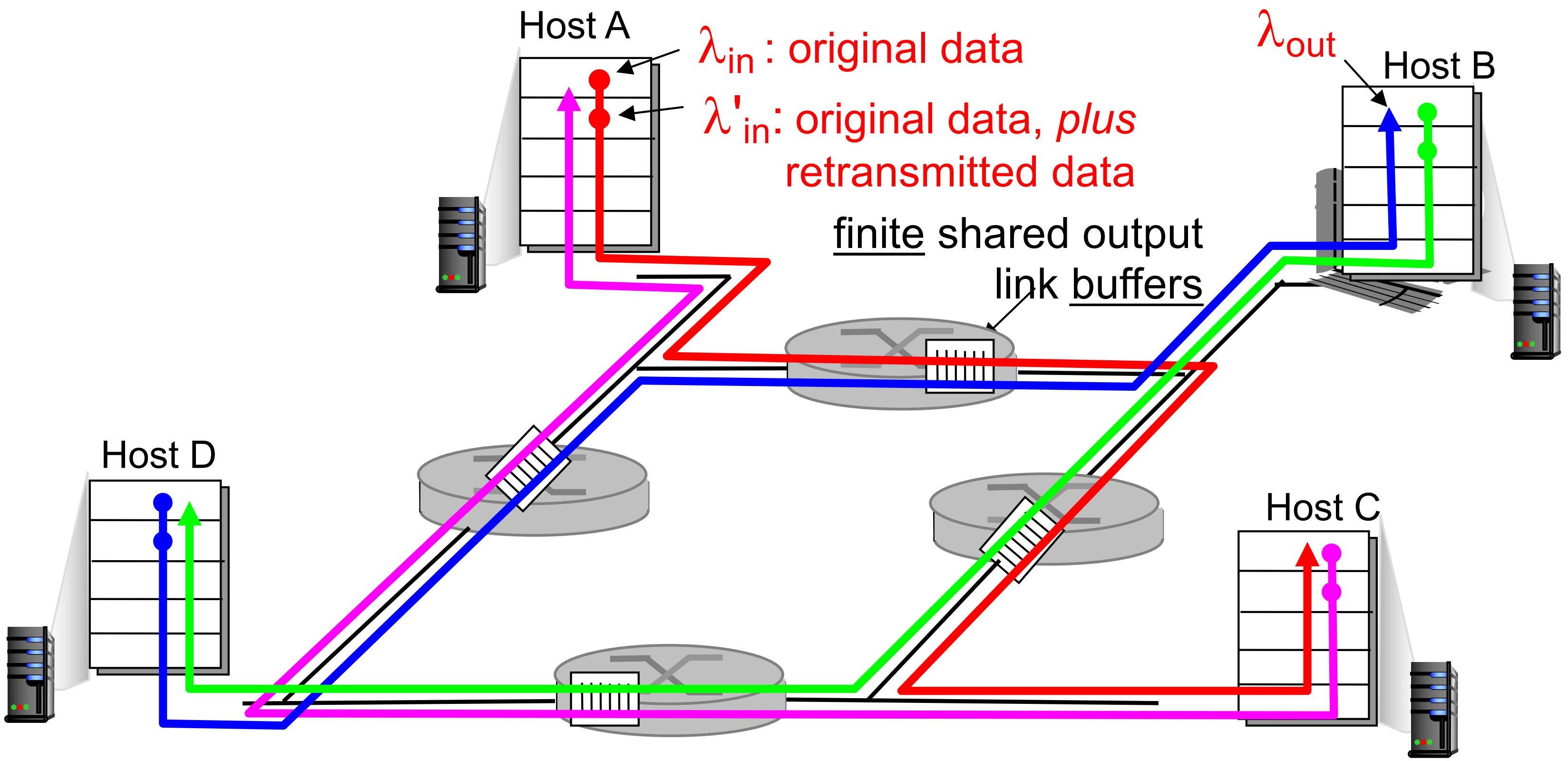
Goodput : the **application level** throughput, i.e. the number of useful information bits delivered by the network to a certain destination per unit of time

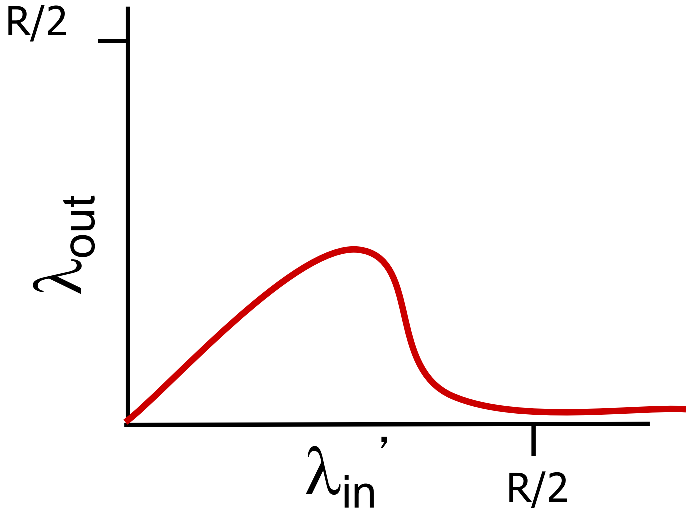
Causes/costs of congestion: scenario 3

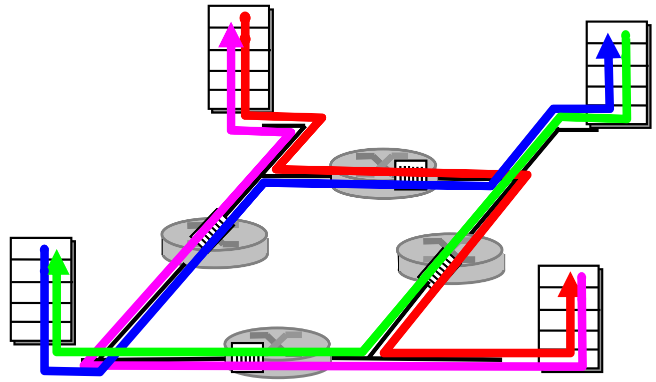
- Four senders
- Multihop paths
- Timeout/retransmit

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

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







Another "cost" of congestion:

- When packet <u>dropped</u>, any "<u>upstream</u> 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

ECN: Explicit Congestion Notification

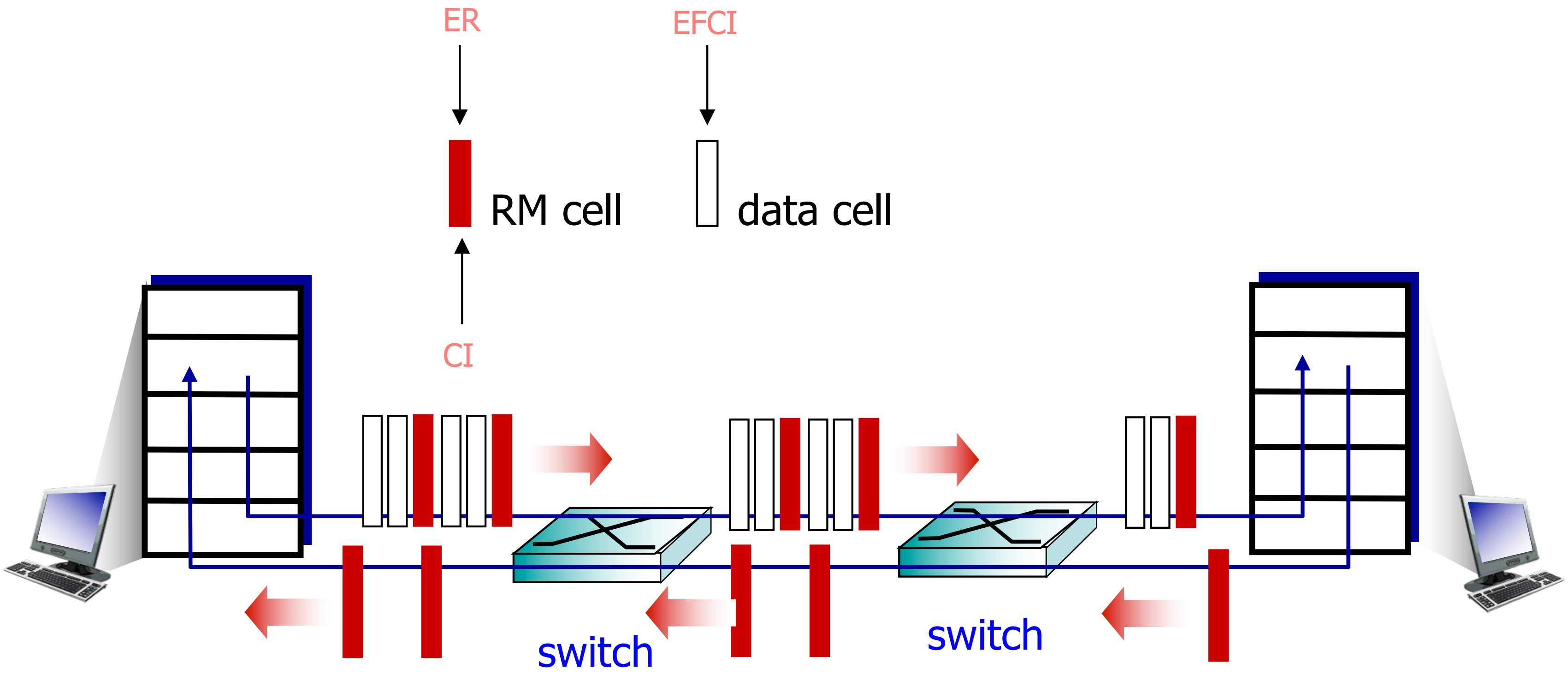
Case study: ATM ABR congestion control

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



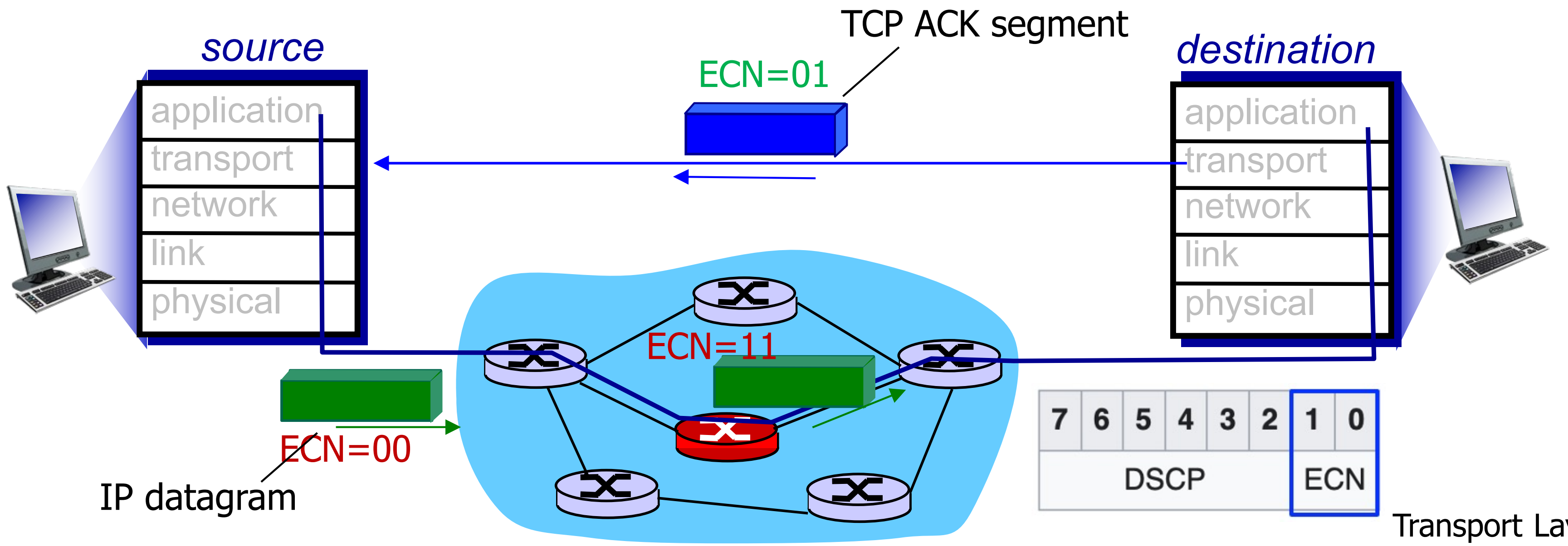
- Two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell [CI bit in RM cell, EFCI bit in data cell]
  - EFCI bit set is useful since there might have congestion between the last switch and destination

EFCI : Explicit Forward Congestion Indication

Explicit Congestion Notification (ECN)

Network-assisted congestion control

- Two bits in IP header (ToS field) marked *by network router* to indicate congestion
- Congestion indication carried to receiving host
- Receiver (seeing congestion indication in IP datagram) sets ECE (ECN-Echo) bit on receiver-to-sender ACK segment to notify sender of congestion



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

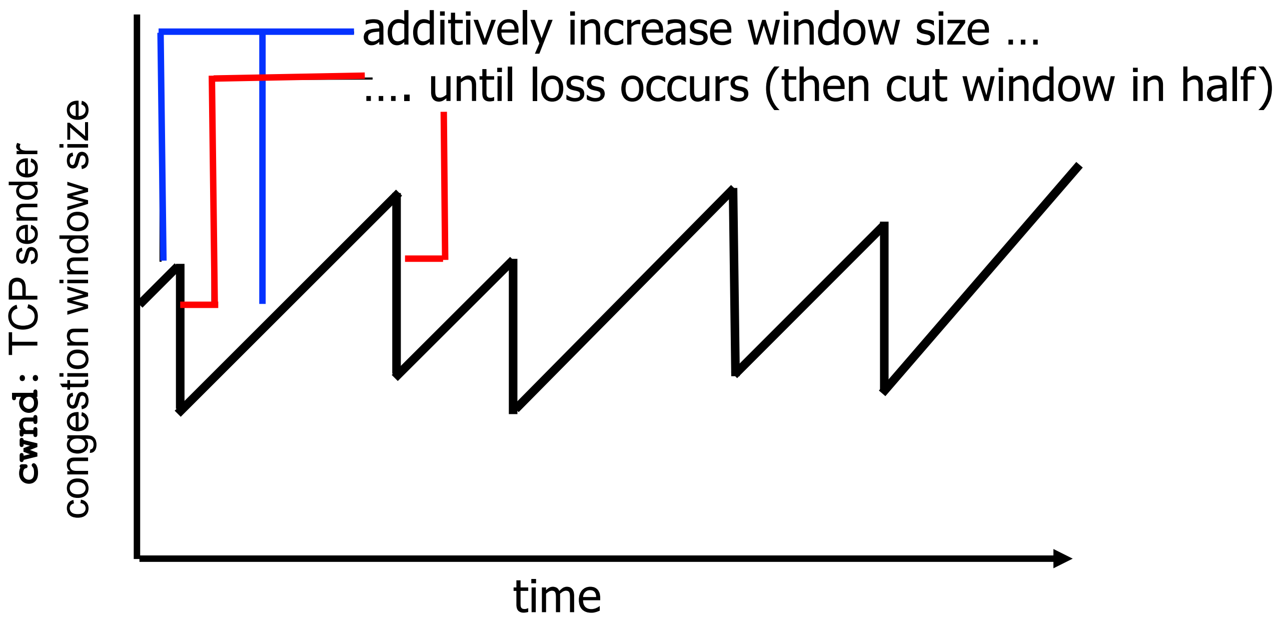
3.6 Principles of congestion control

3.7 TCP congestion control

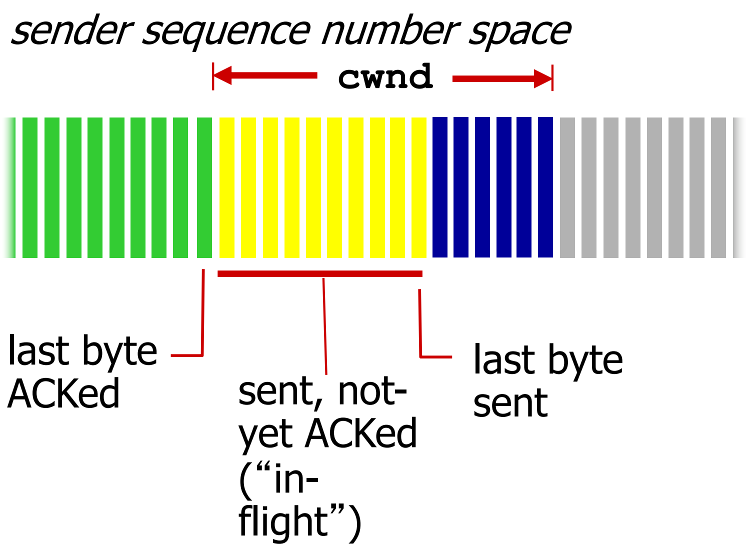
TCP congestion control: Additive Increase Multiplicative Decrease (AIMD)

- *Approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS (Max Segment Size) every RTT until loss detected
  - *multiplicative decrease:* cut **`cwnd`** in half after loss

AIMD
- saw tooth behavior
- probing for bandwidth



TCP Congestion Control: details



- Sender limits transmission:

$$\texttt{LastByteSent-}\ \texttt{LastByteAcked} \leq \texttt{cwnd}$$

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

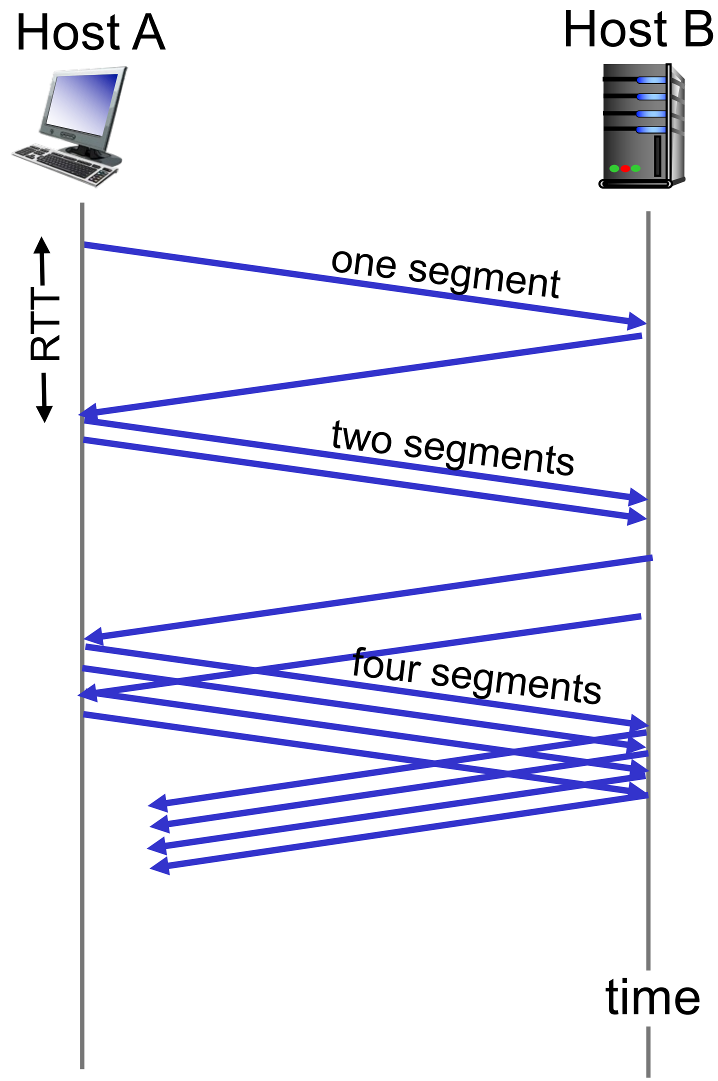
TCP ***sending rate***

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

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

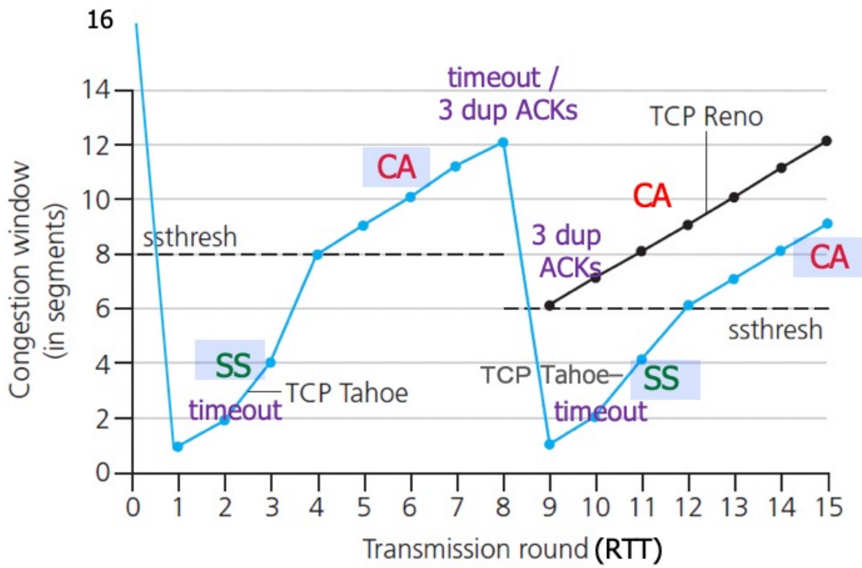
TCP Slow Start

- When connection begins, increase rate exponentially (double) until first loss event
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** (1 MSS) for every ACK received
  - eg,: 1 MSS = 500 bytes, RTT = 200 msce then initial rate = 20 kbps [cwnd/RTT = 500 x 8 (bits) / 0.2s = 20kbps]
- *Summary:* initial rate is slow but ramps up exponentially fast (double)



TCP: detecting, reacting to loss

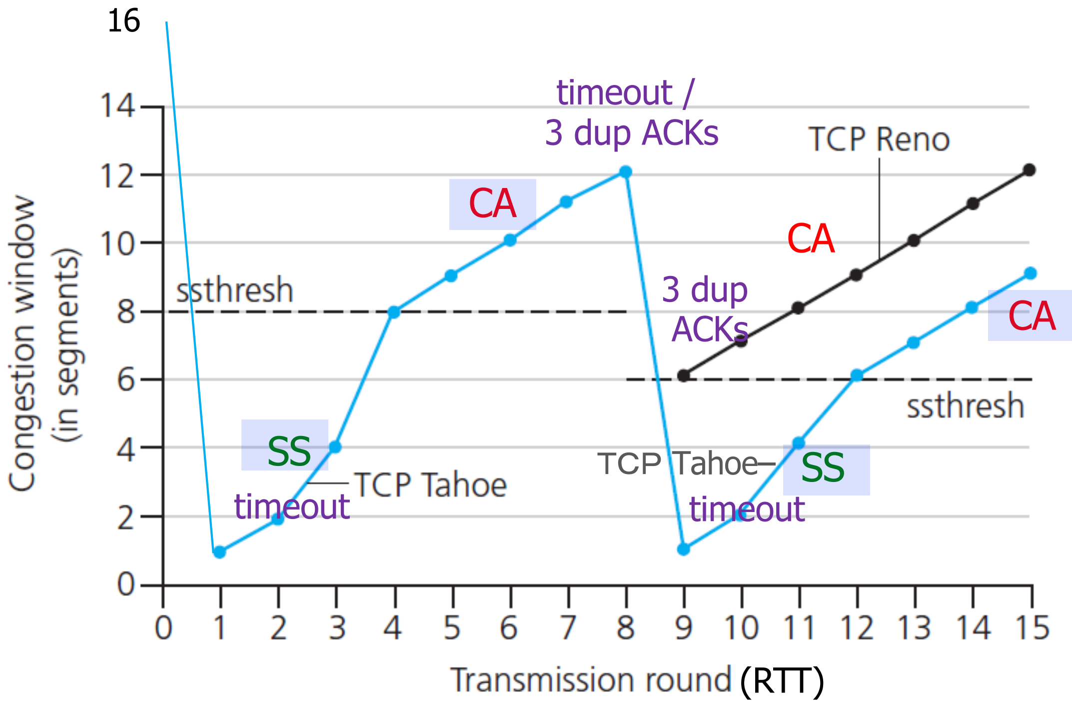
- Loss indicated by timeout : TCP Tahoe
  - **`cwnd`** set to 1 MSS (TCP Tahoe always sets cwnd to 1)
  - window then grows exponentially (as in slow start) to threshold (**ssthresh**, cut in half), then grows linearly (CA)
- Loss indicated by 3 duplicate ACKs: TCP RENO
  - duplicate ACKs indicate network is still capable of delivering some segments
  - **`cwnd`** is cut in half window then grows linearly (CA)



TCP: switching from slow start to CA

Q: When should the exponential increase (SS) switch to linear (CA)? [TCP Tahoe]

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



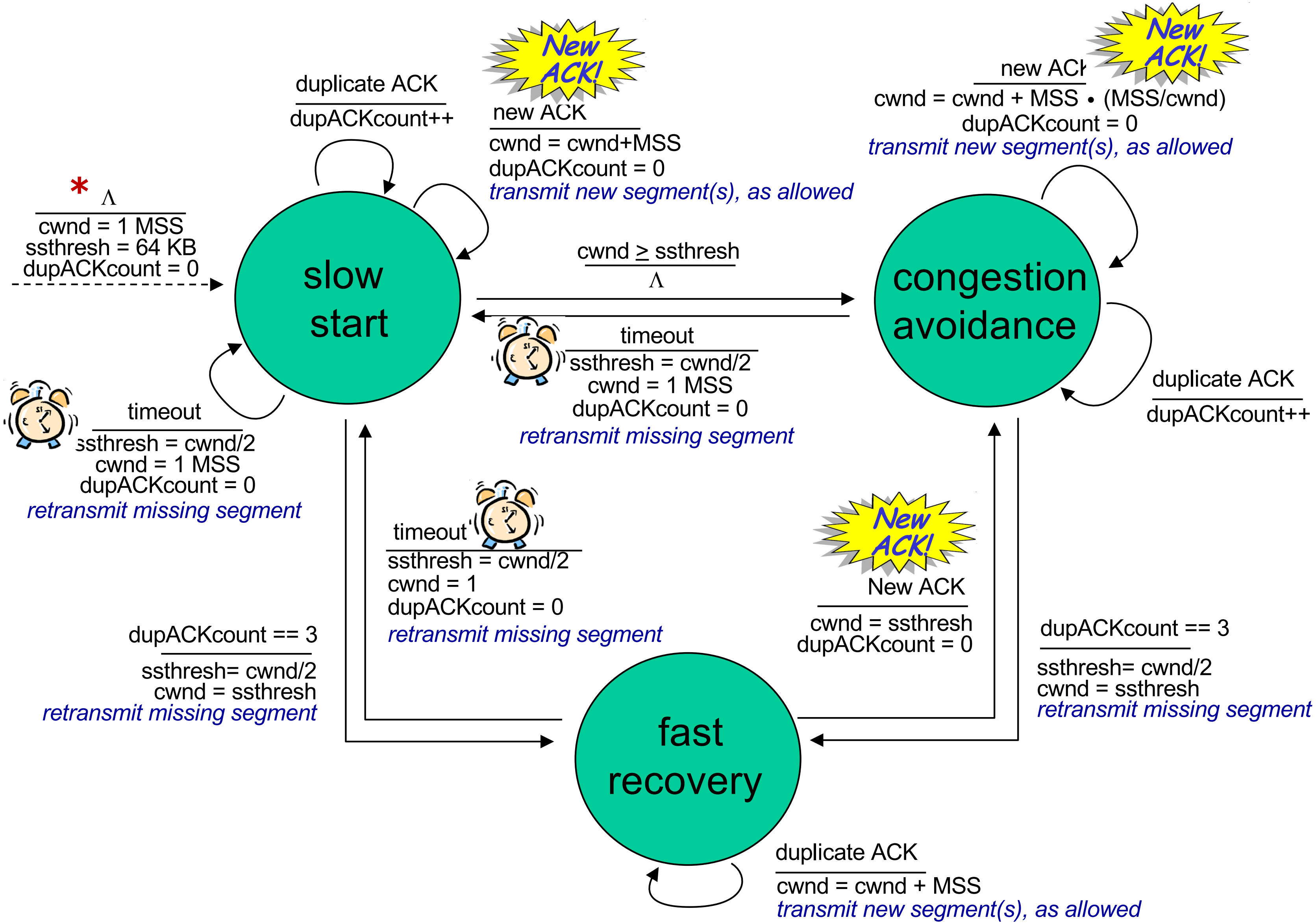
Implementation

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

TCP sender congestion control

State	Event	TCP sender action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	cwnd = cwnd + MSS, if (cwnd > ssthresh) set state to CA	resulting in a doubling of cwnd every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	cwnd = cwnd + MSS * (MSS / cwnd)	additive increase, resulting in increase of cwnd by 1 MSS every RTT
SS or CA	loss event detected by 3 duplicate ACK	ssthresh = cwnd / 2, cwnd = ssthresh, set state to CA	fast recovery, implementing multipliactive decrease, cwnd will not drop below 1 MSS
SS or CA	timeout	ssthresh = cwnd / 2, cwnd = 1 MSS, set state to SS	enter slow start
SS or CA	duplicate ACK	increment duplicate ACK count for segment being acked	cwnd and ssthresh not changed

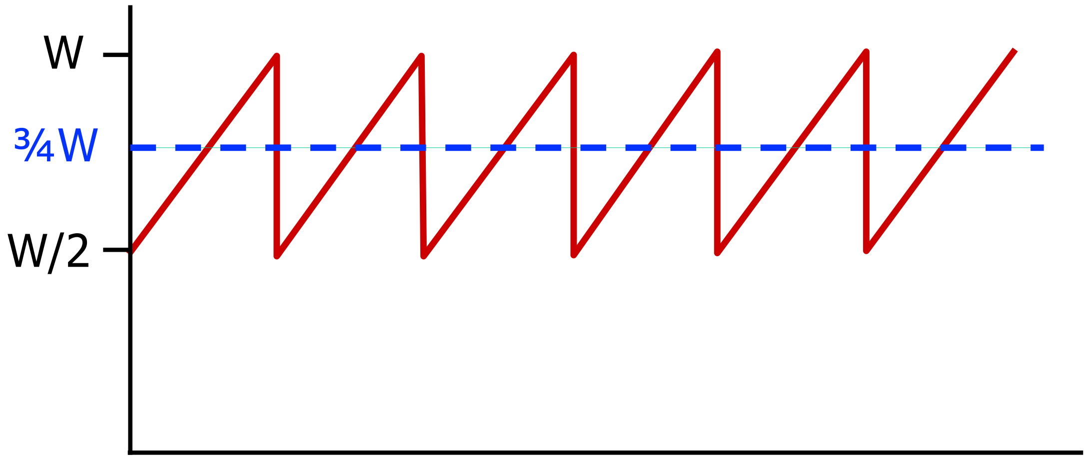
Summary: TCP Congestion Control



TCP throughput

- Avg. TCP throughput as function of window size and 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 ¾W [= (W + 0.5W) / 2]
  - avg. thruput [cwnd/RTT] is ¾W 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 [cwnd/RTT]
- Requires window size W = 83,333 in-flight <u>segments</u> [83,333 (seg.) * 1500 (byte) * 8 (bit)] / 0.1 (s) = 10 Gbps
- Throughput in terms of segment loss probability, L [Mathis 1997]:

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

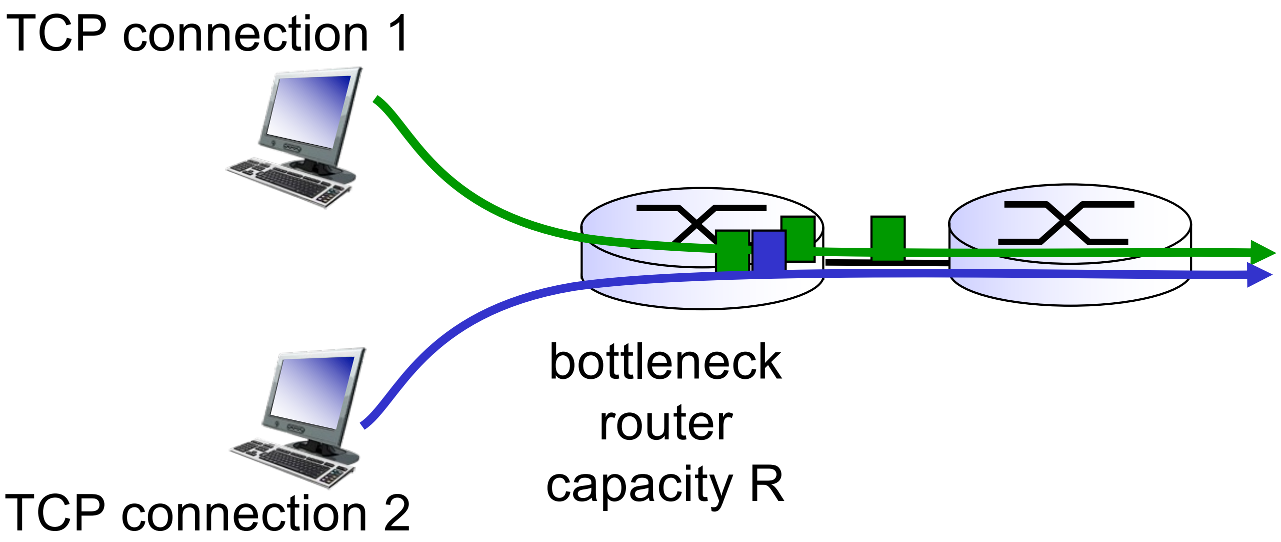
[Problem P45]

➜ to achieve 10 Gbps throughput, need a loss rate of L = $2 \cdot 10^{-10}$ – *a very small loss rate!*

- New versions of TCP for high-speed

TCP Fairness

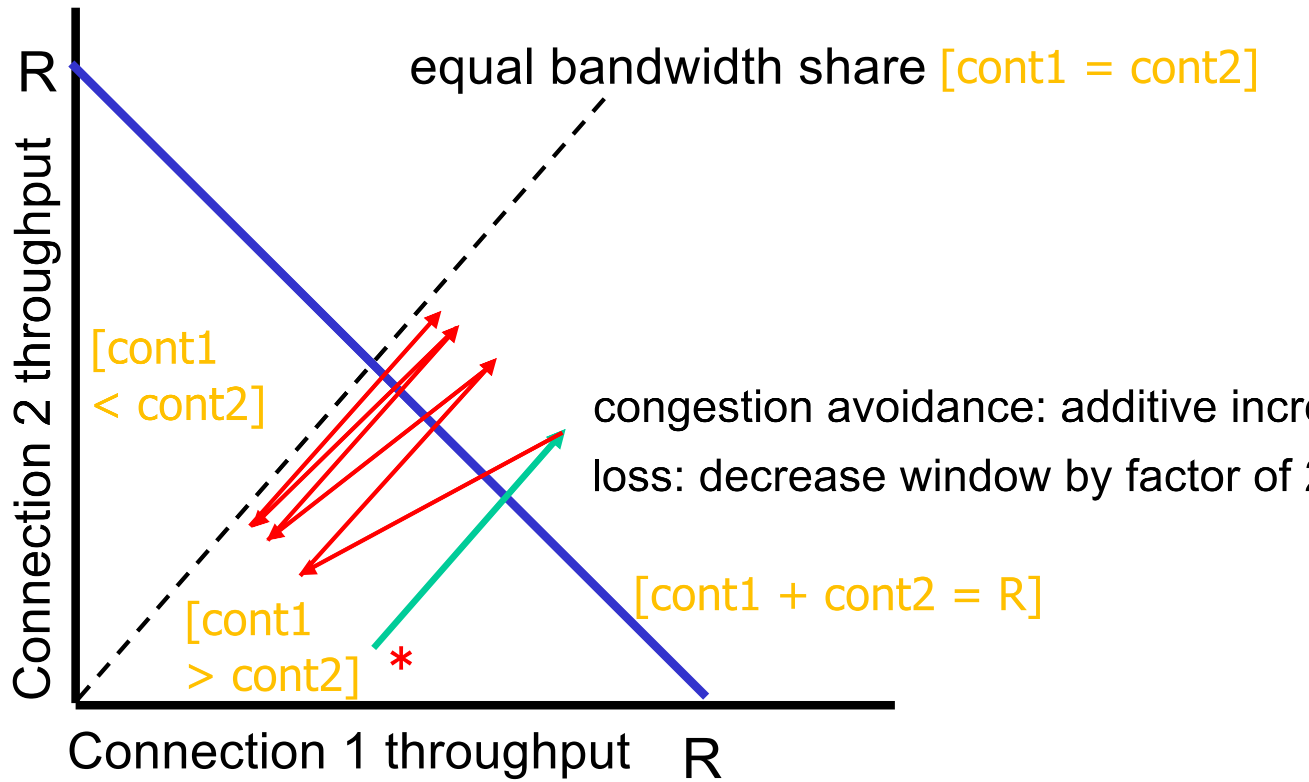
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



R is the router capacity.

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
- e.g., link of rate *R* with 9 existing connections
  - new app asks for 1 TCP, gets rate R/10
    $R/(1+9) = R/10$
  - new app asks for 2 TCPs, gets 2R/11
    $2R/(2+9) = 2R/11 > R/10$

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"