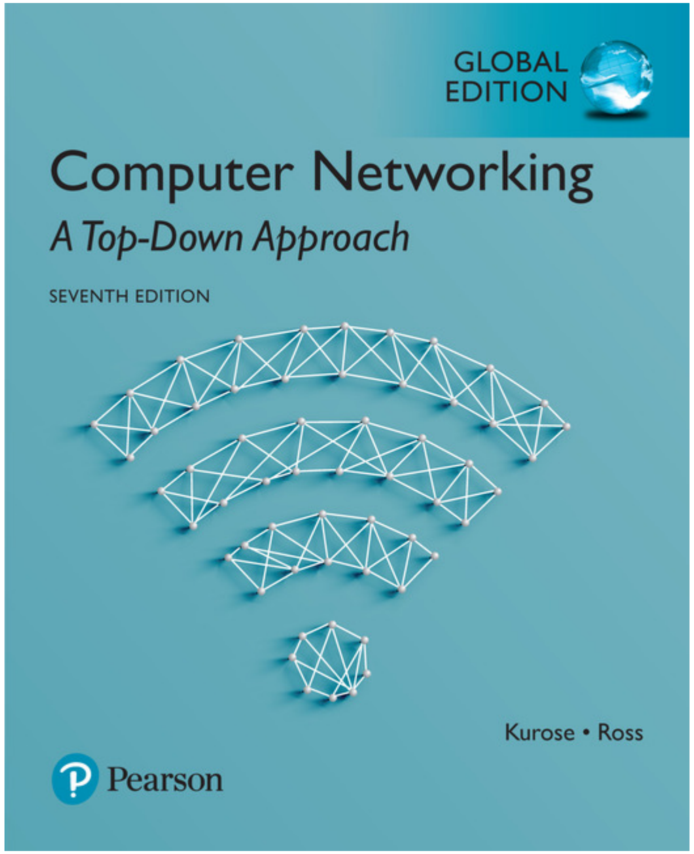
Chapter 3
Transport Layer



A note on the use of these Powerpoint slides:

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

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

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach

7th Edition, Global Edition
Jim Kurose, Keith Ross
Pearson
April 2016

Chapter 3: Transport Layer

our goals:

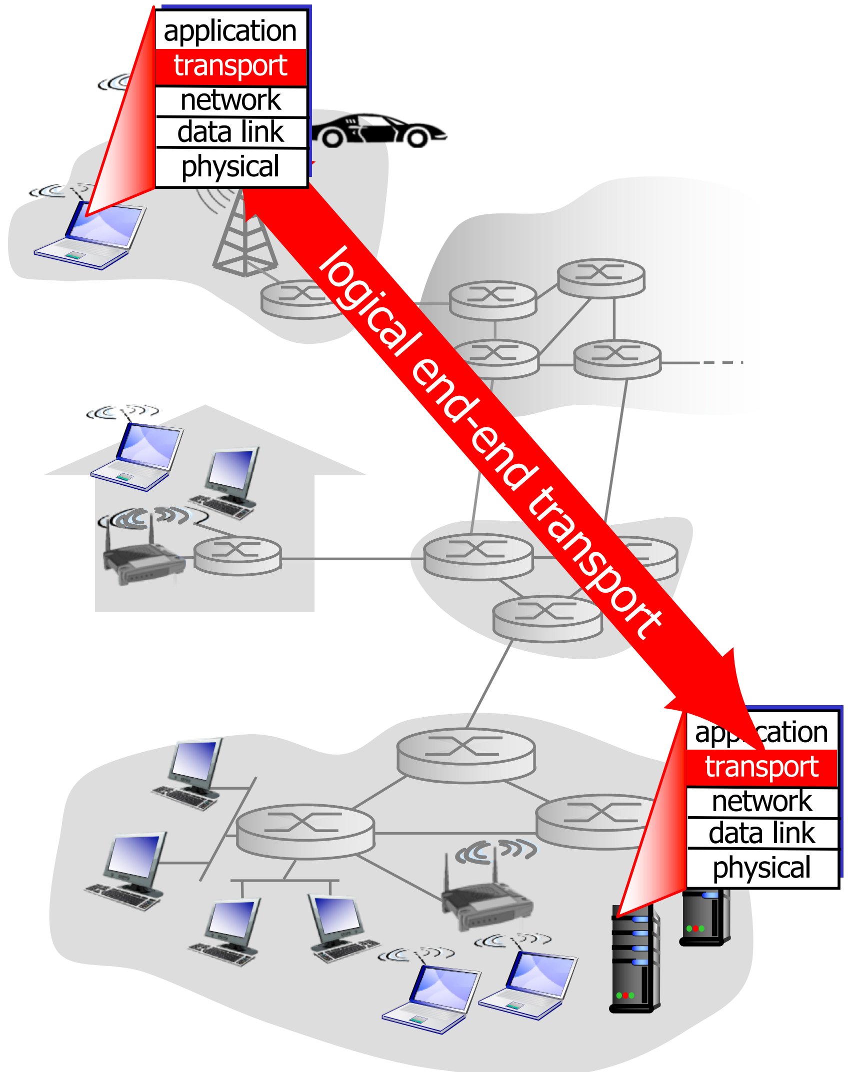
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion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

Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer
- 3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

Transport services and protocols

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



Transport vs. network layer

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

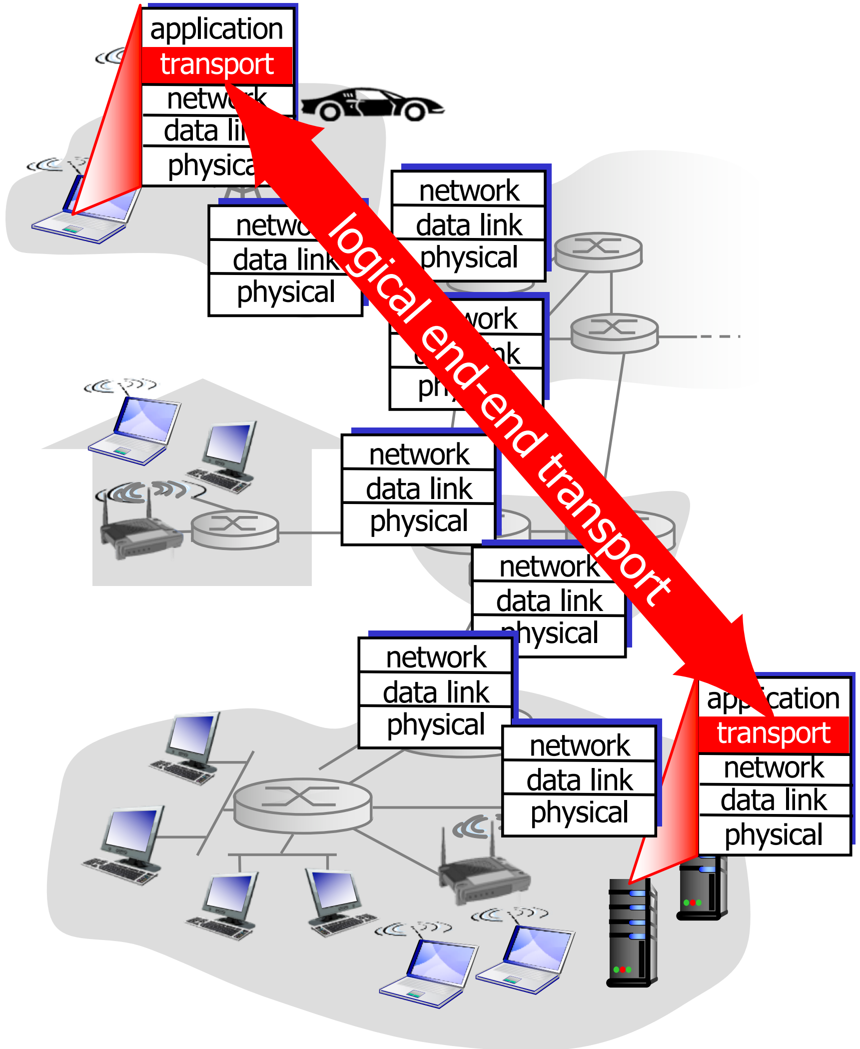
household analogy:

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

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

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



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

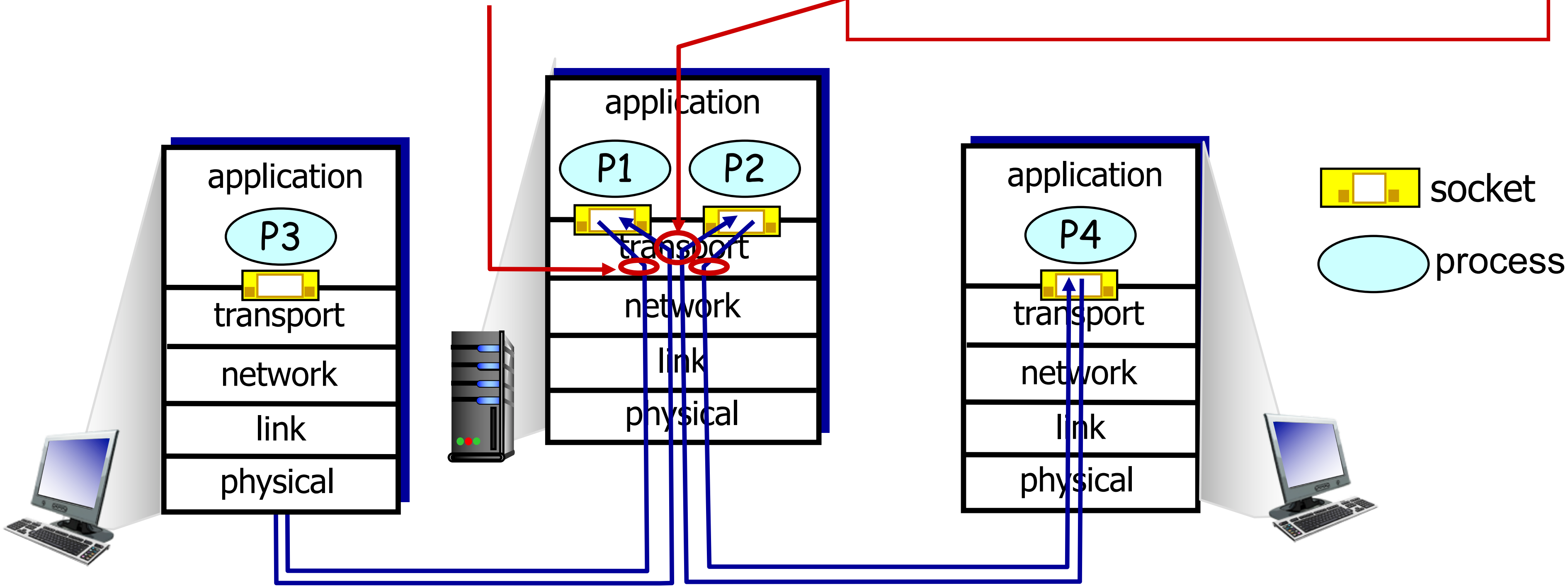
3.6 principles of congestion control

3.7 TCP congestion control

Multiplexing/demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

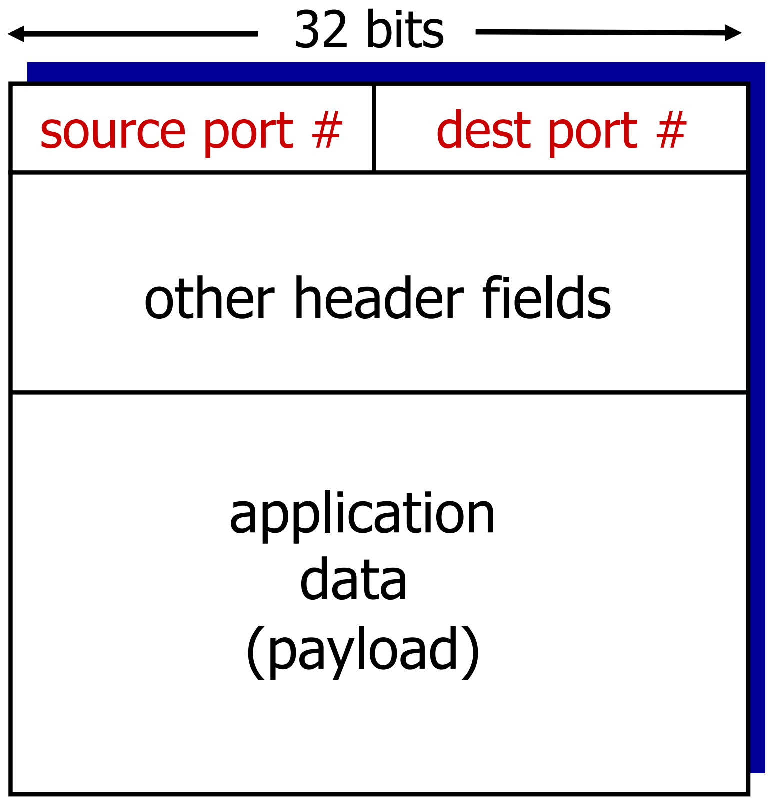
demultiplexing at receiver:
use header info to deliver received segments to correct socket



Quiz Time!

How demultiplexing works

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



TCP/UDP segment format

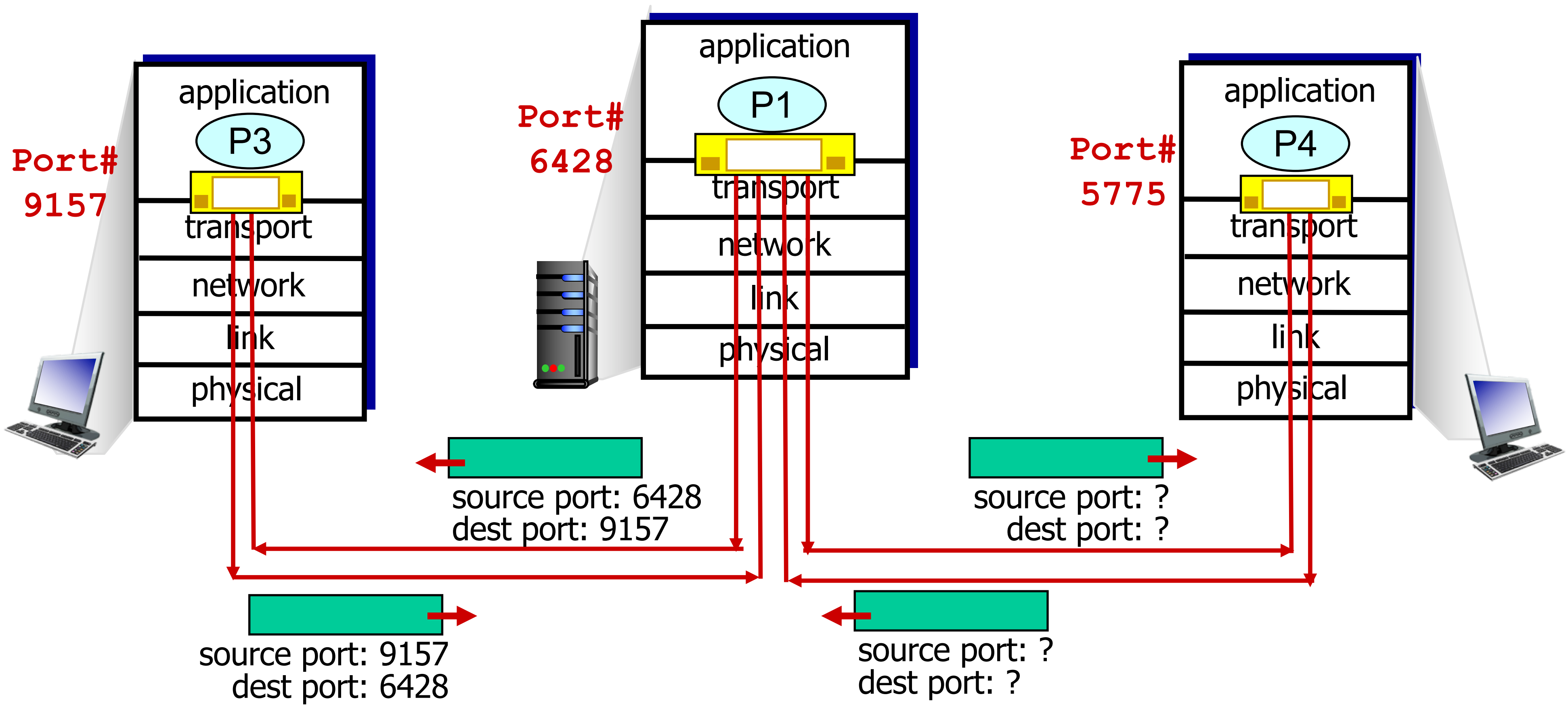
Connectionless demultiplexing

- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux: example

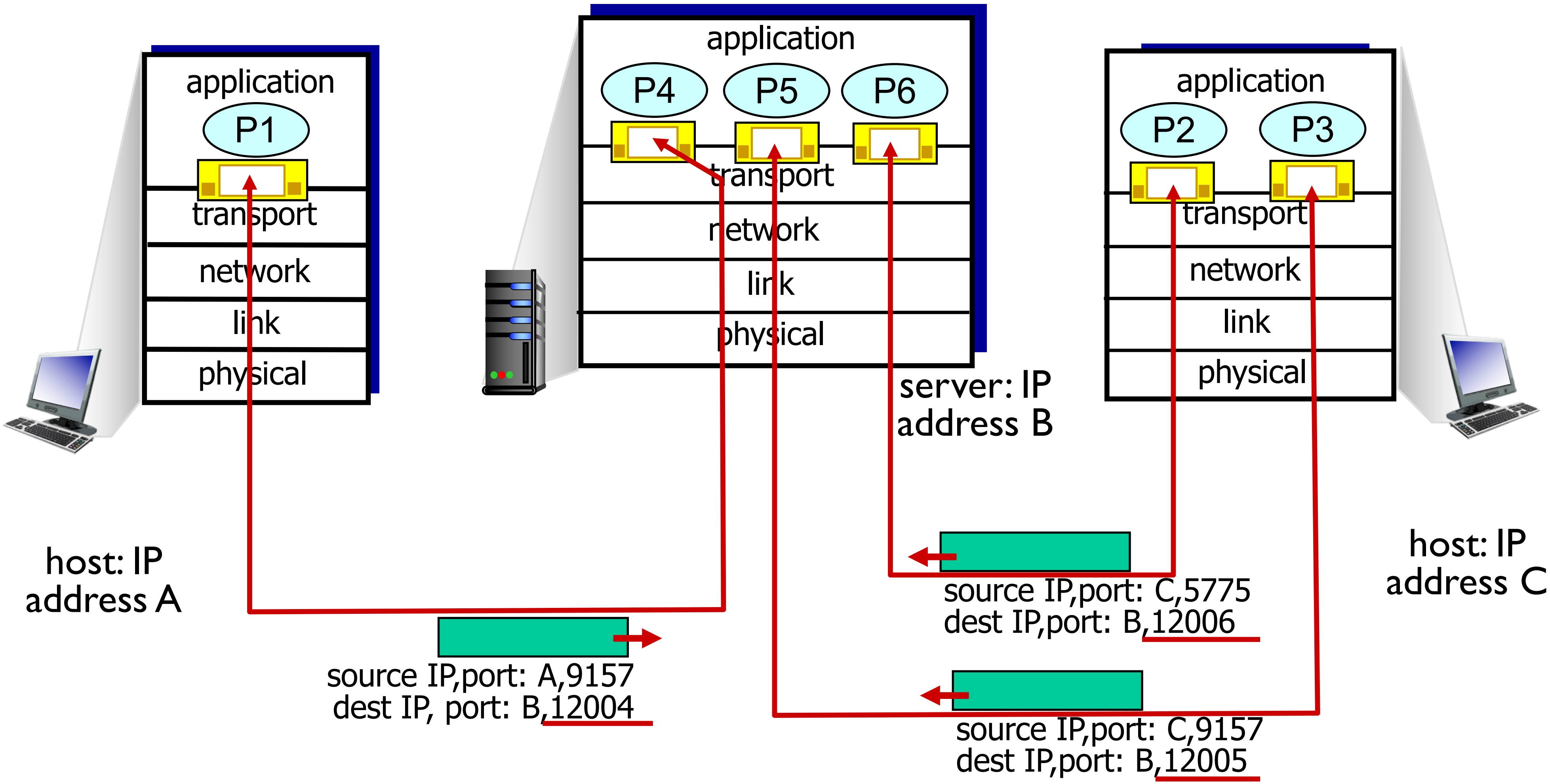


Connection-oriented demux

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

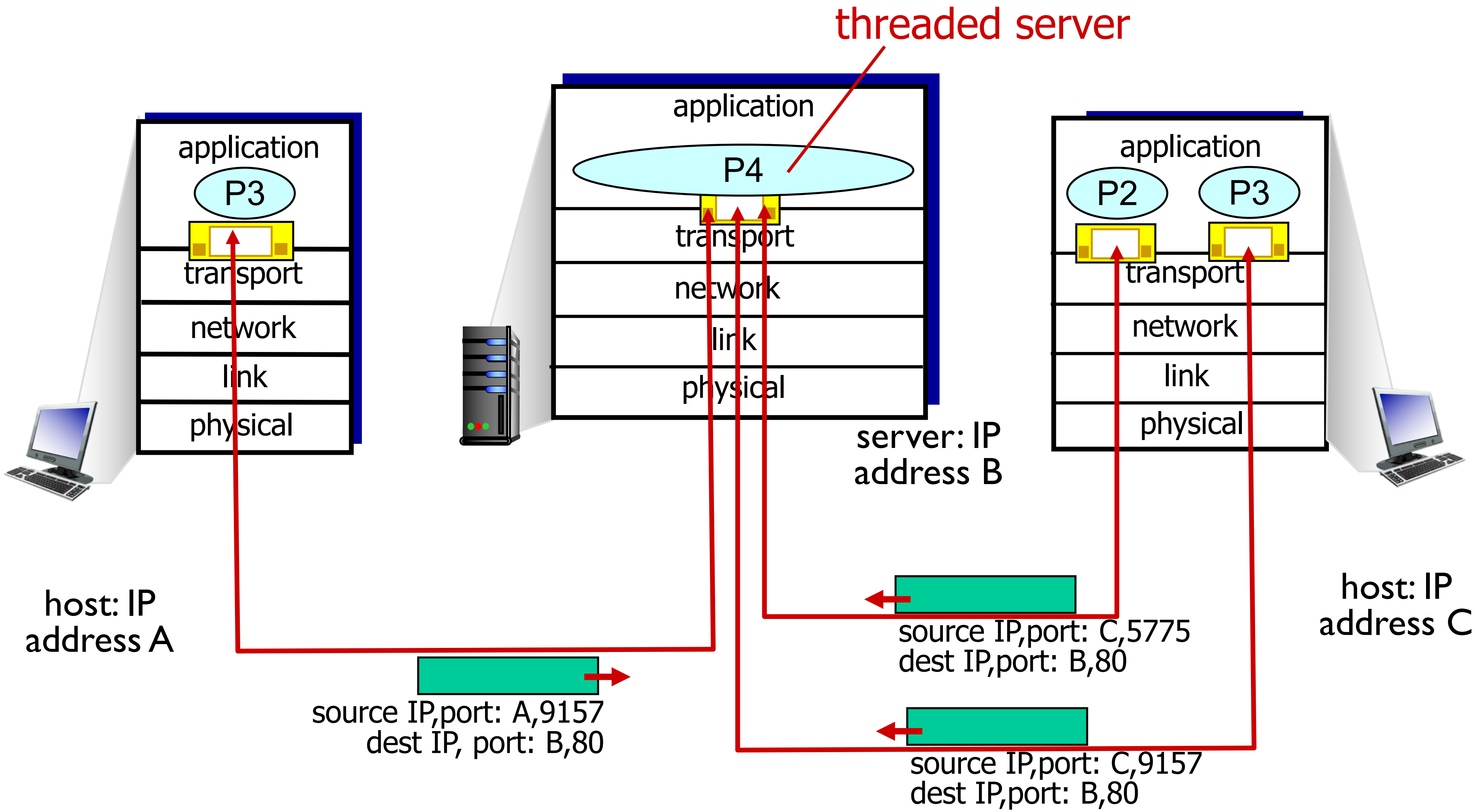
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



three segments, all destined to IP address: B,
demultiplexed to *different* sockets

Connection-oriented demux: example



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

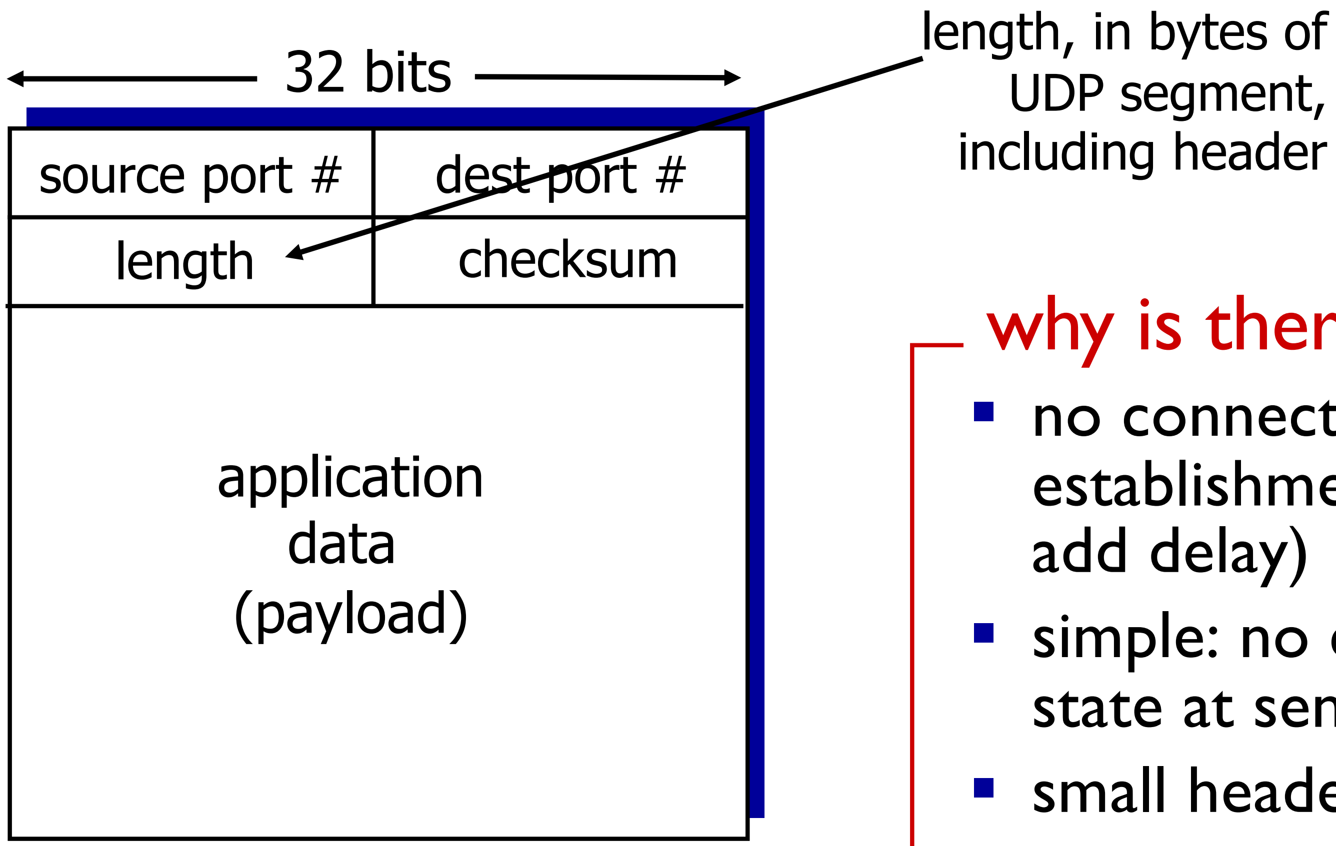
3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

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

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

UDP: segment header



UDP segment format

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

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

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

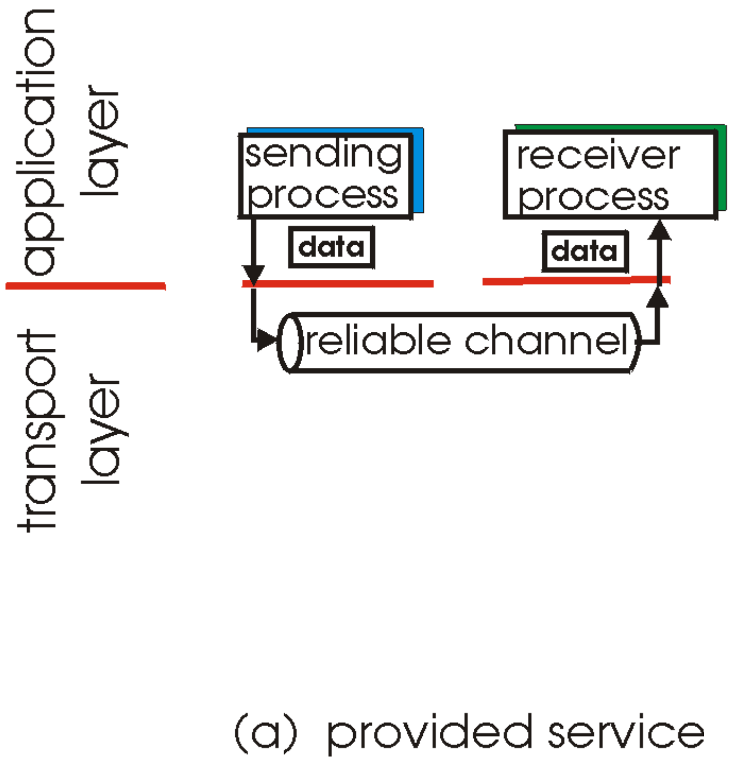
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

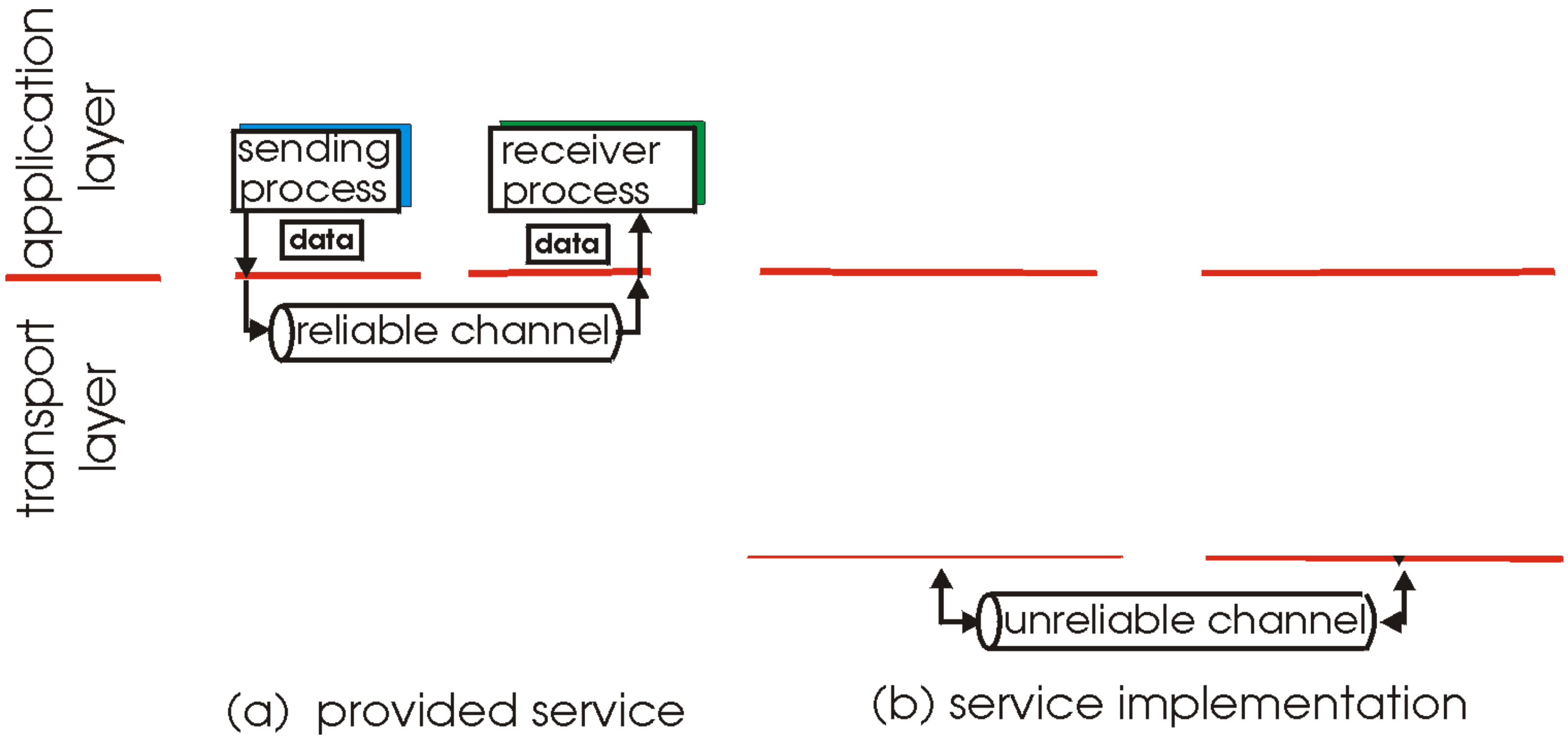
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

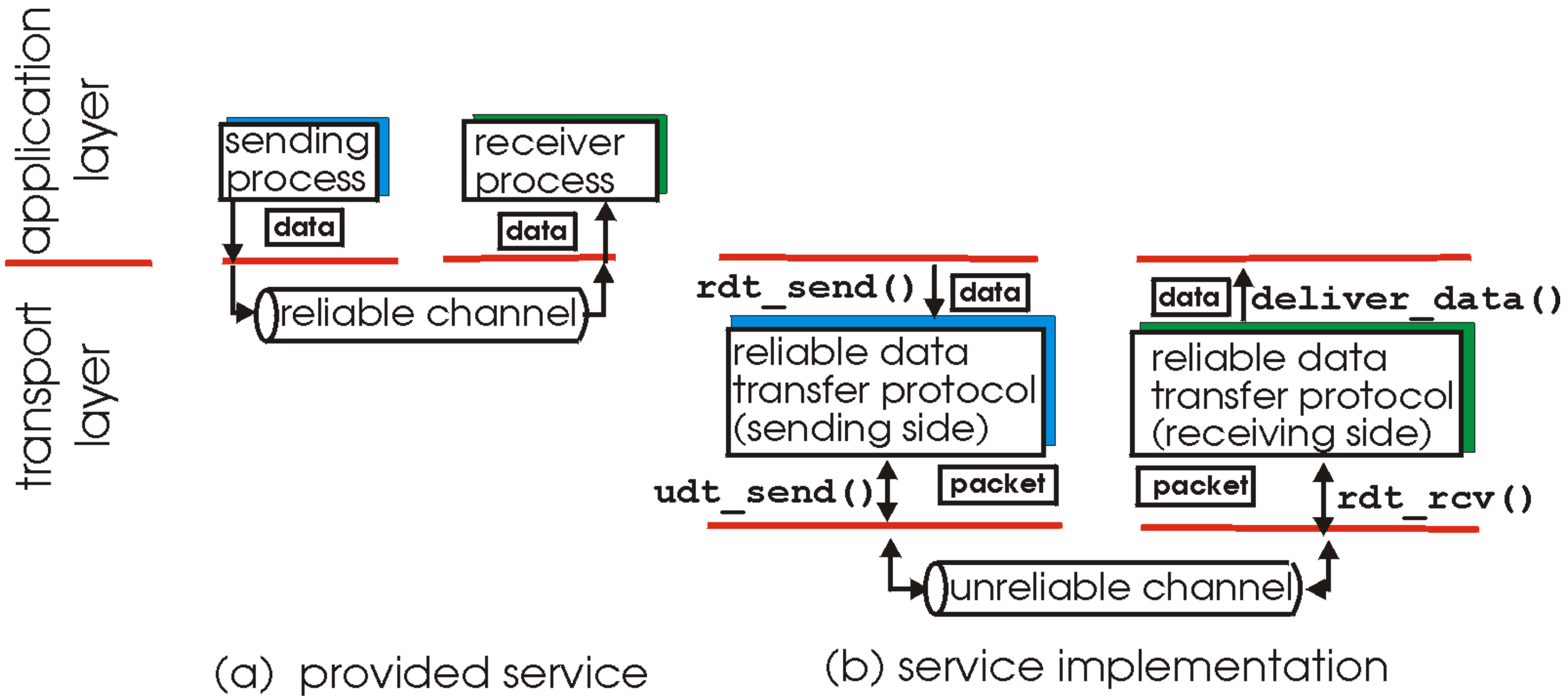
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

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

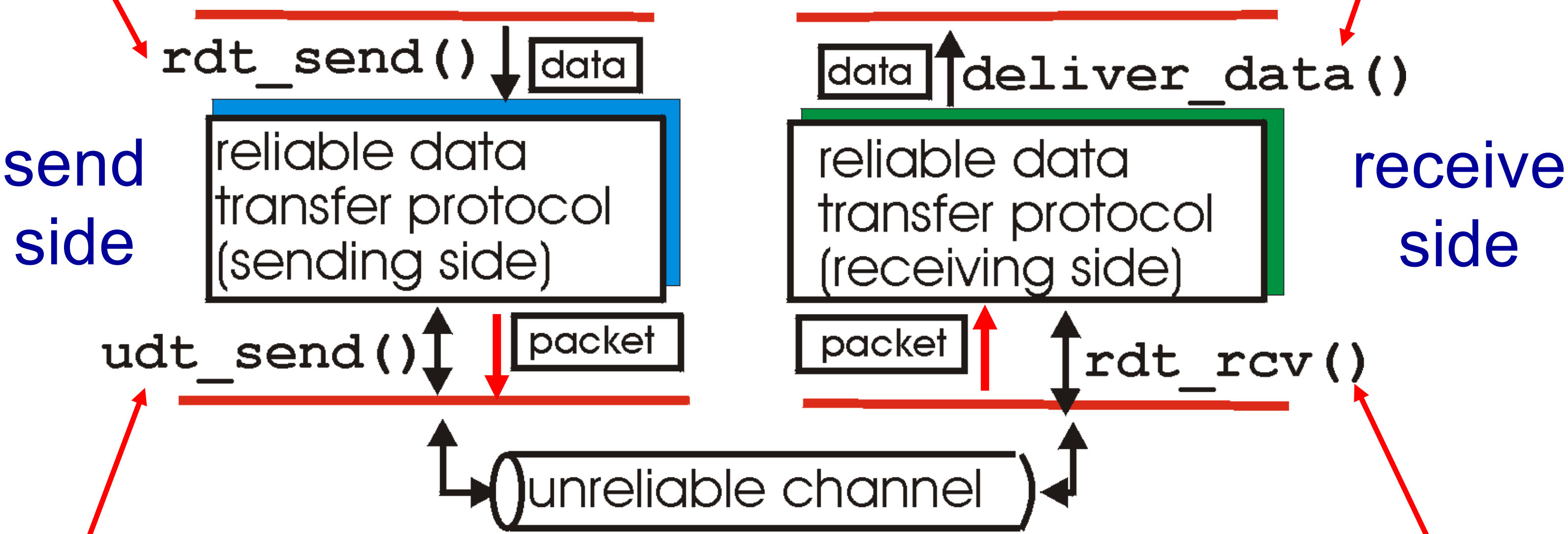


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

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

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



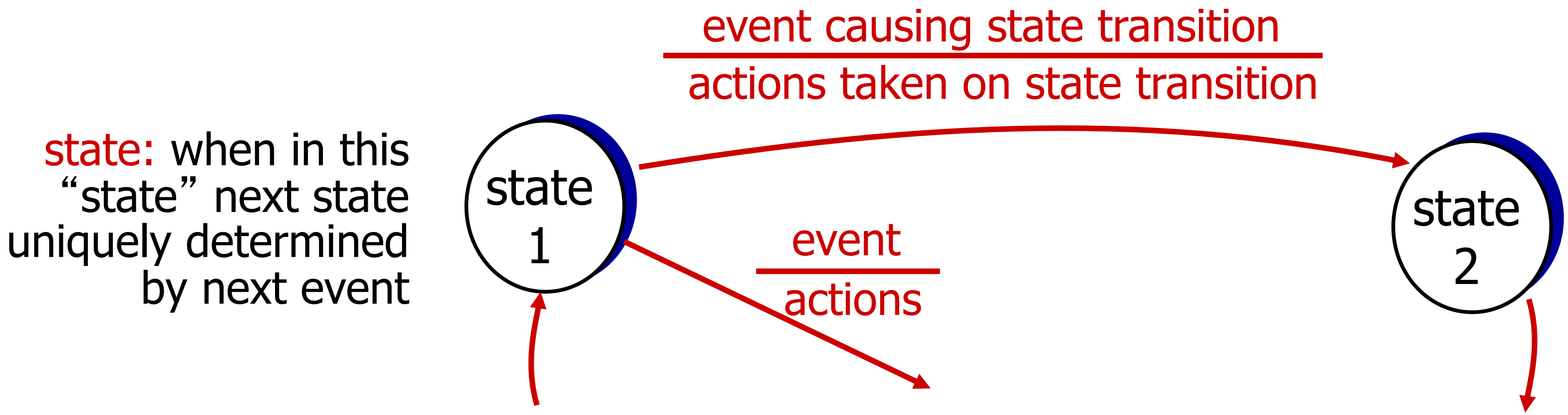
udt_send(): called by **rdt**, to transfer packet over unreliable channel to receiver

rdt_rcv(): called from **below** when packet arrives

Reliable data transfer: getting started

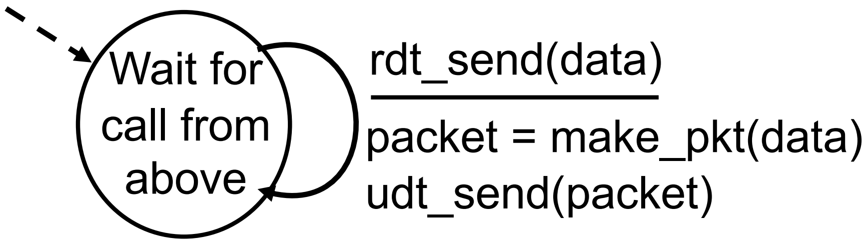
we' ll:

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

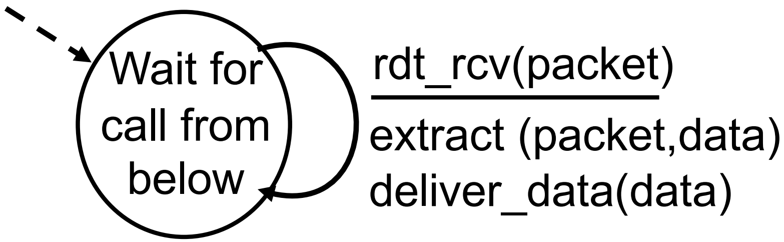


rdt1.0: reliable transfer over a reliable channel

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



sender



receiver

rdt2.0: channel with bit errors

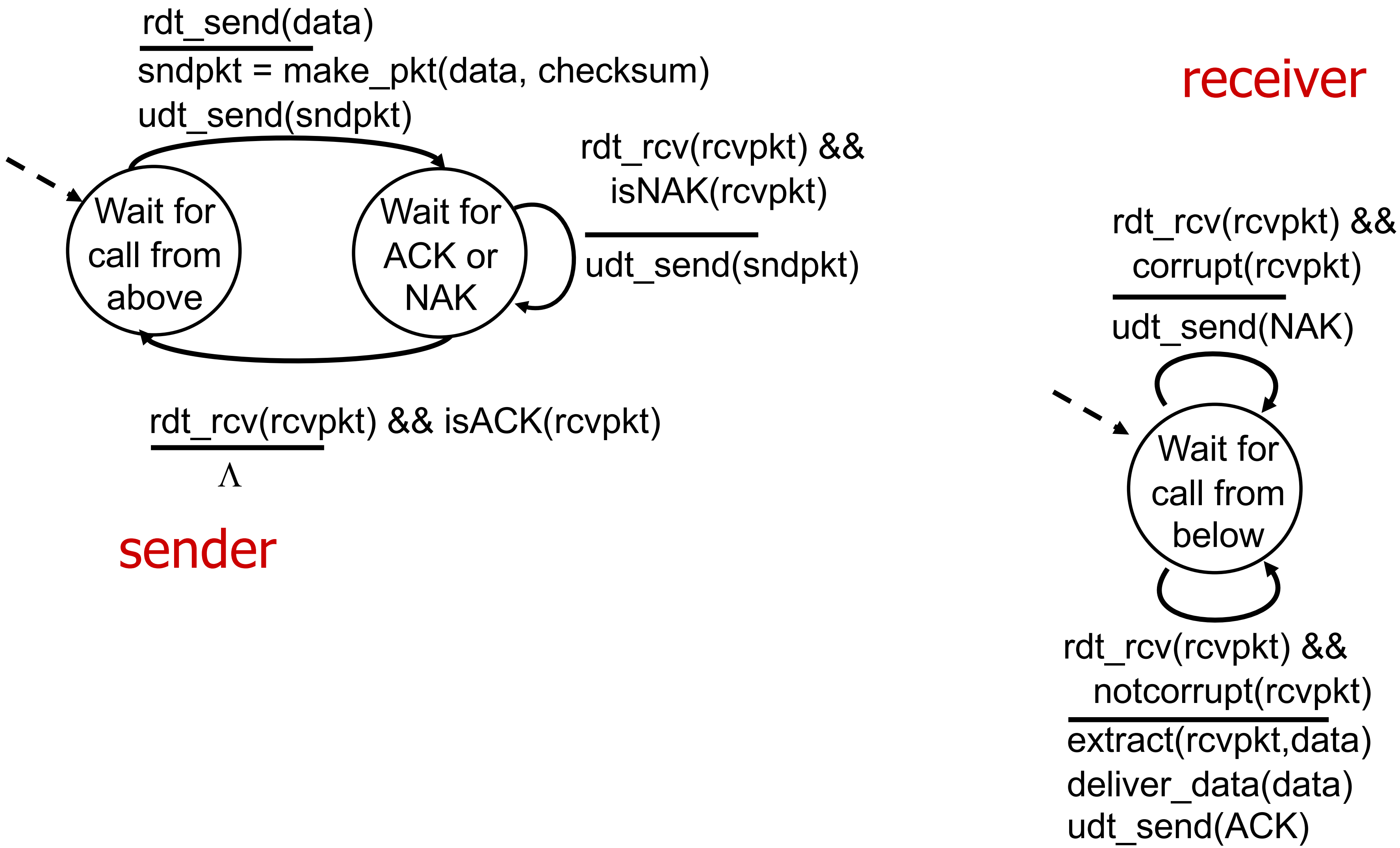
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

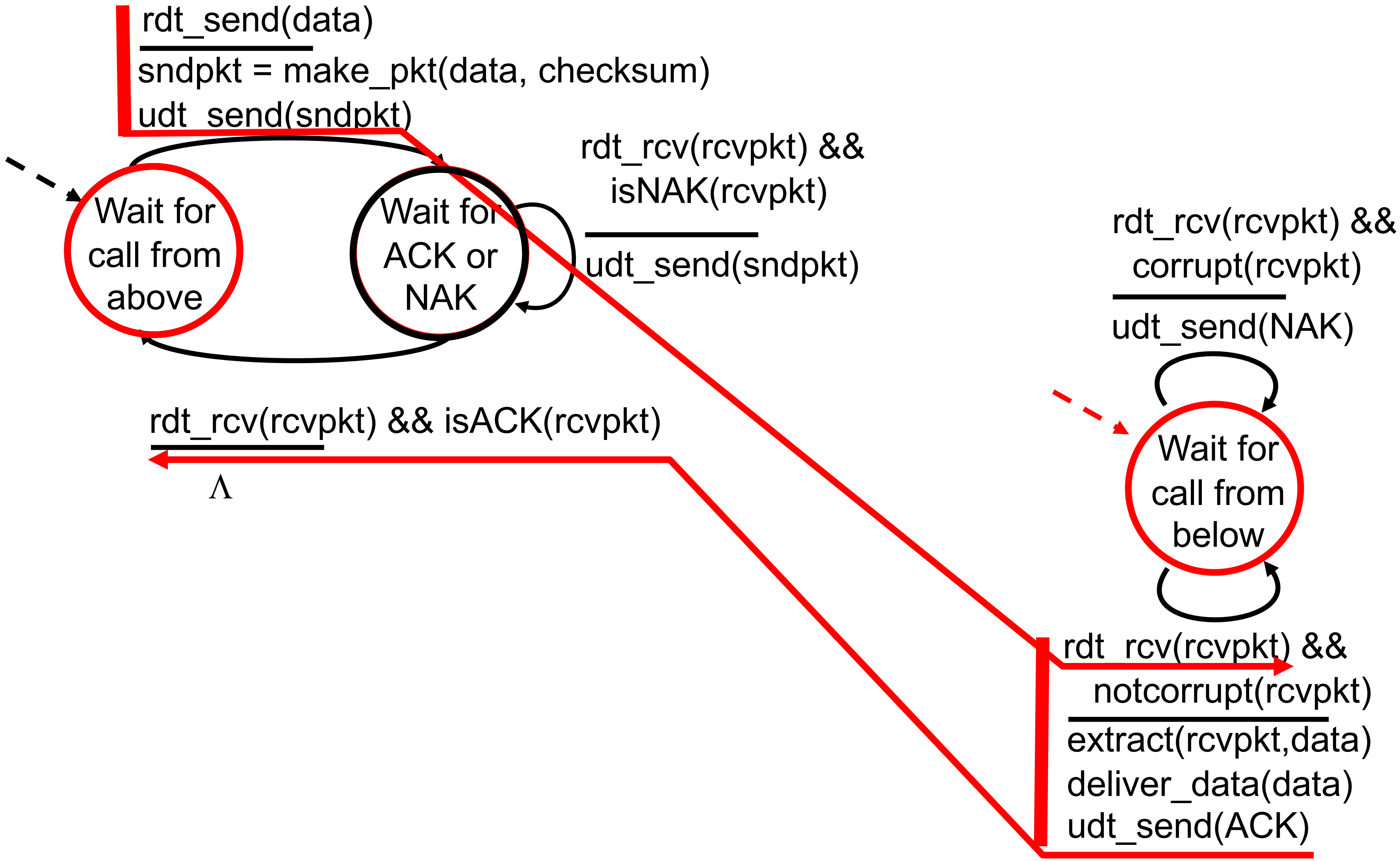
rdt2.0: channel with bit errors

- 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
  - retransmit in case of NAK

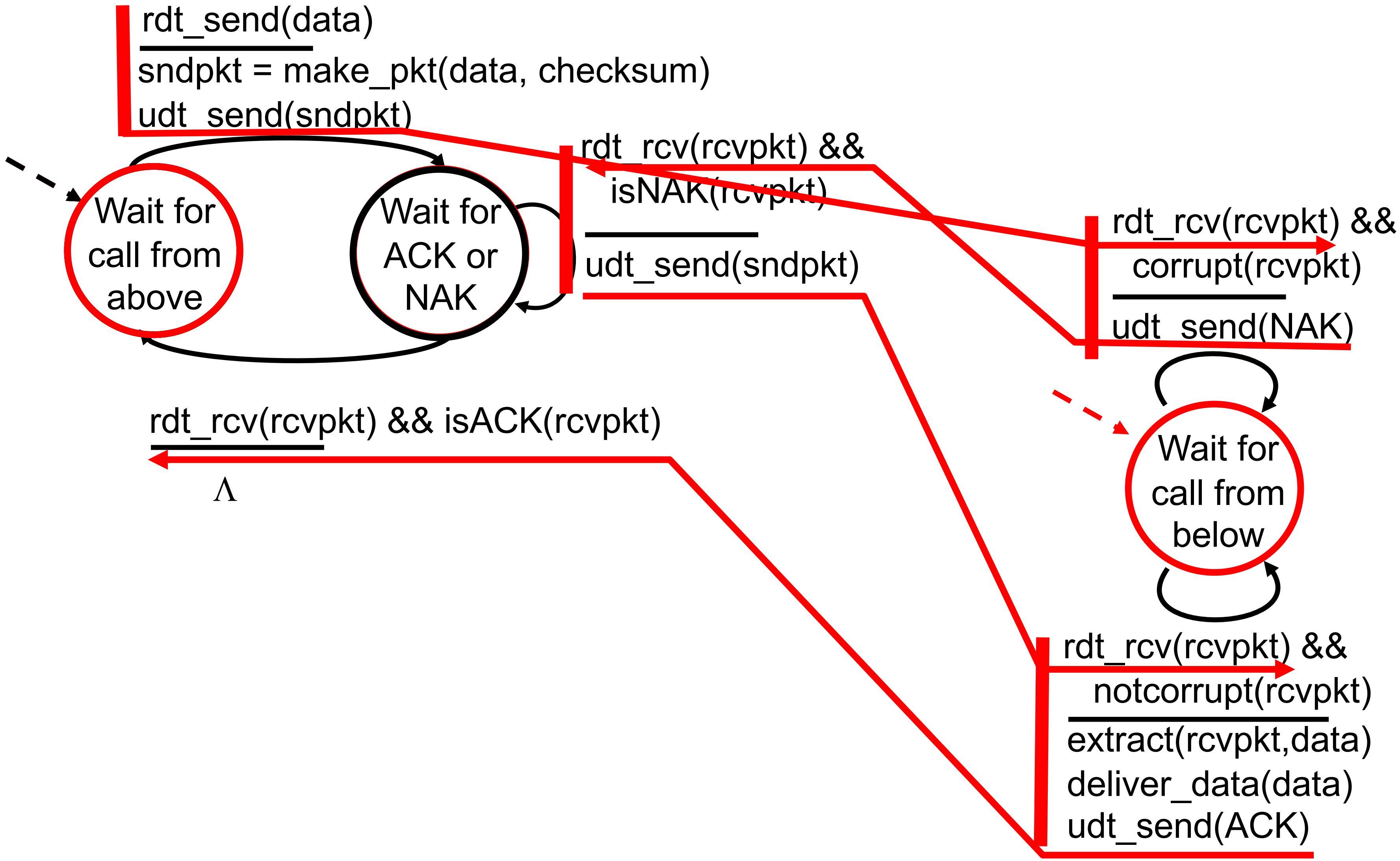
rdt2.0: FSM specification



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!

What to do?

- Q?
- Checksum the ACK/NAK?
- ACK/NAK the ACK/NAK?
- Retransmit the ACK/NAK?

What happens if ACK/NAK of ACK/NAK corrupted?

- >”<

How did we deal with corrupted data?

- Checksum data
- ACK/NAK the data
- Retransmit data

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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

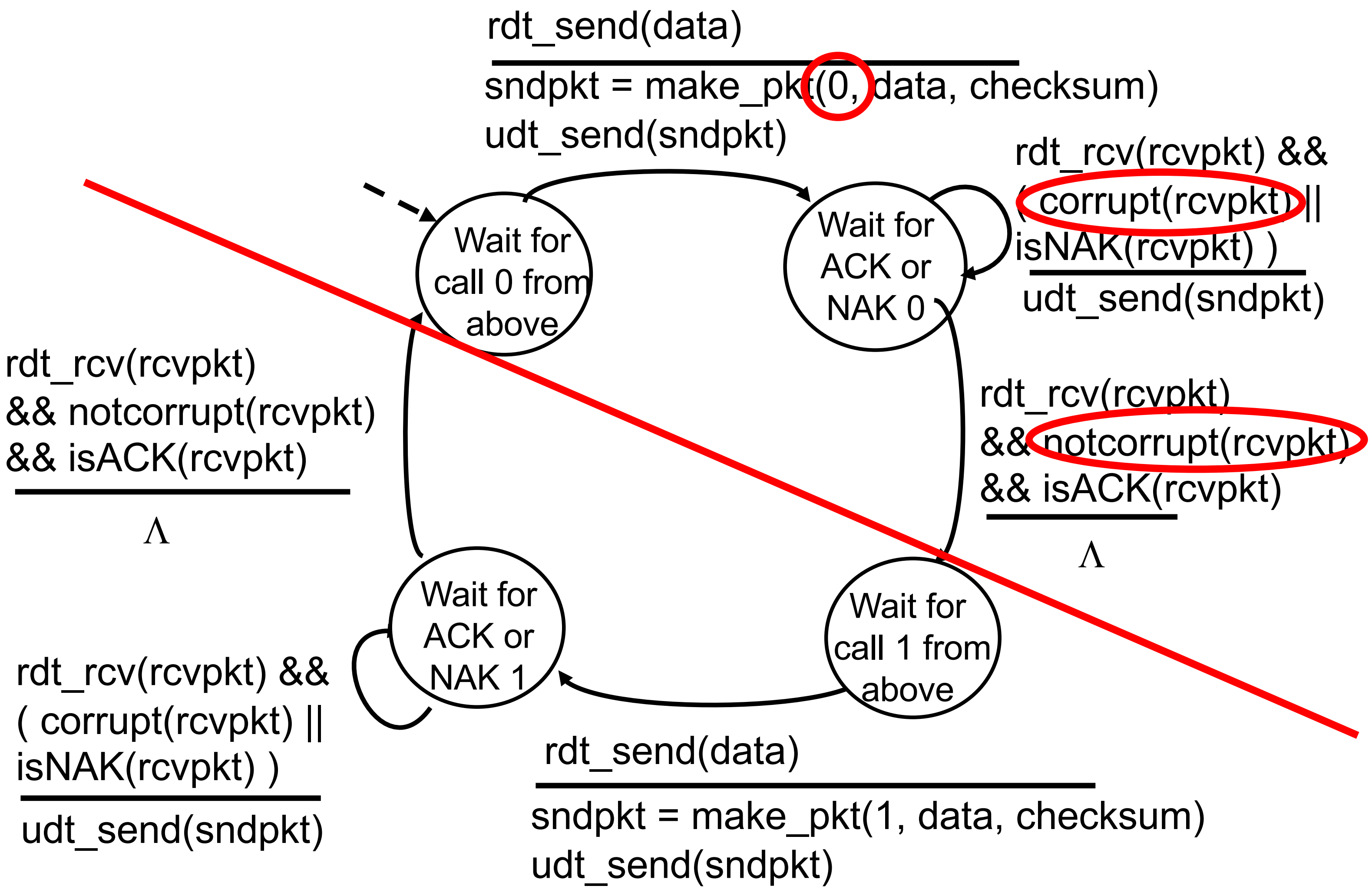
Handling error on the ACK/NAK direction in rdt2.1

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

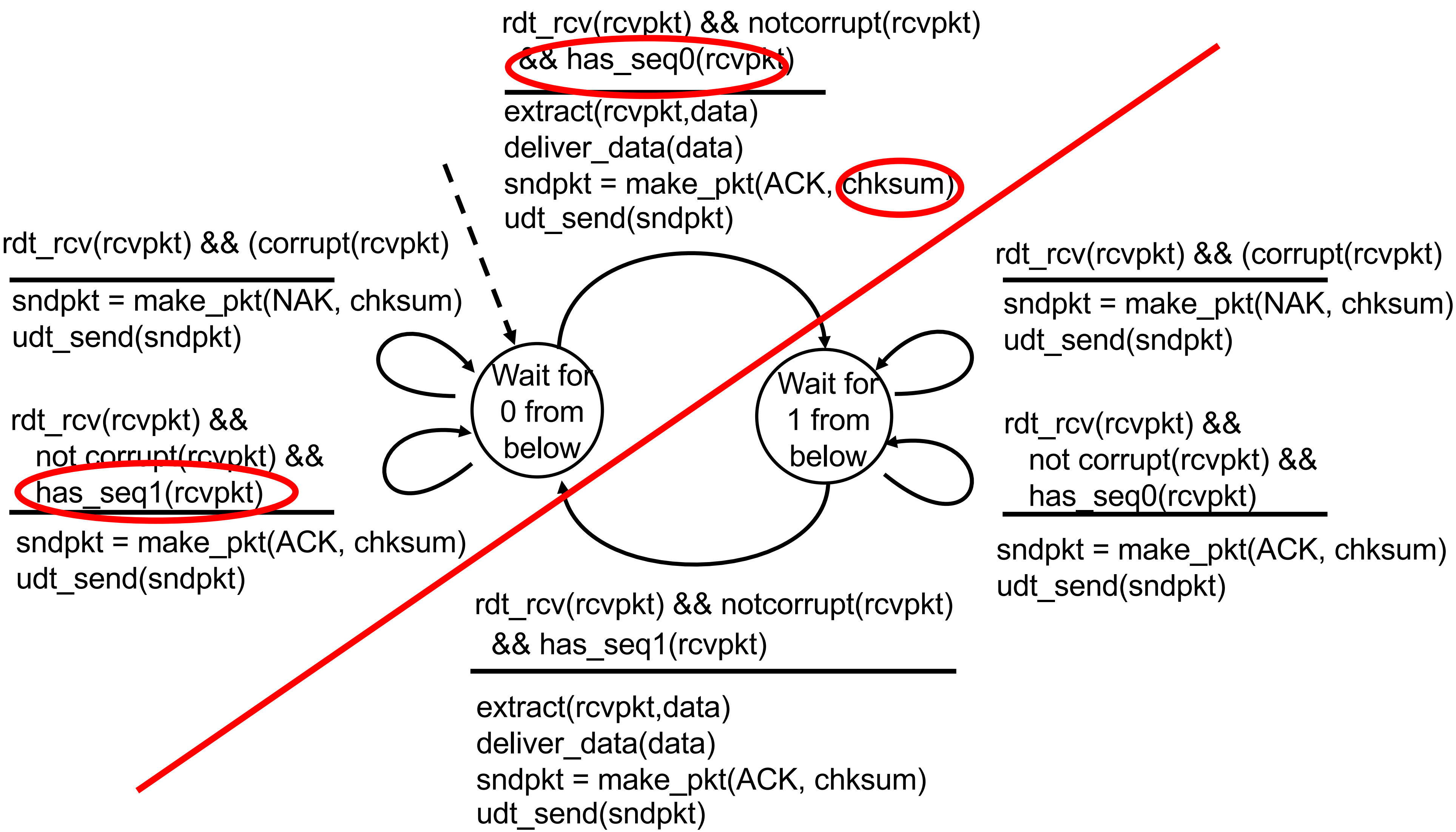
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



Double Quiz Time!

rdt2.1: discussion

sender:

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

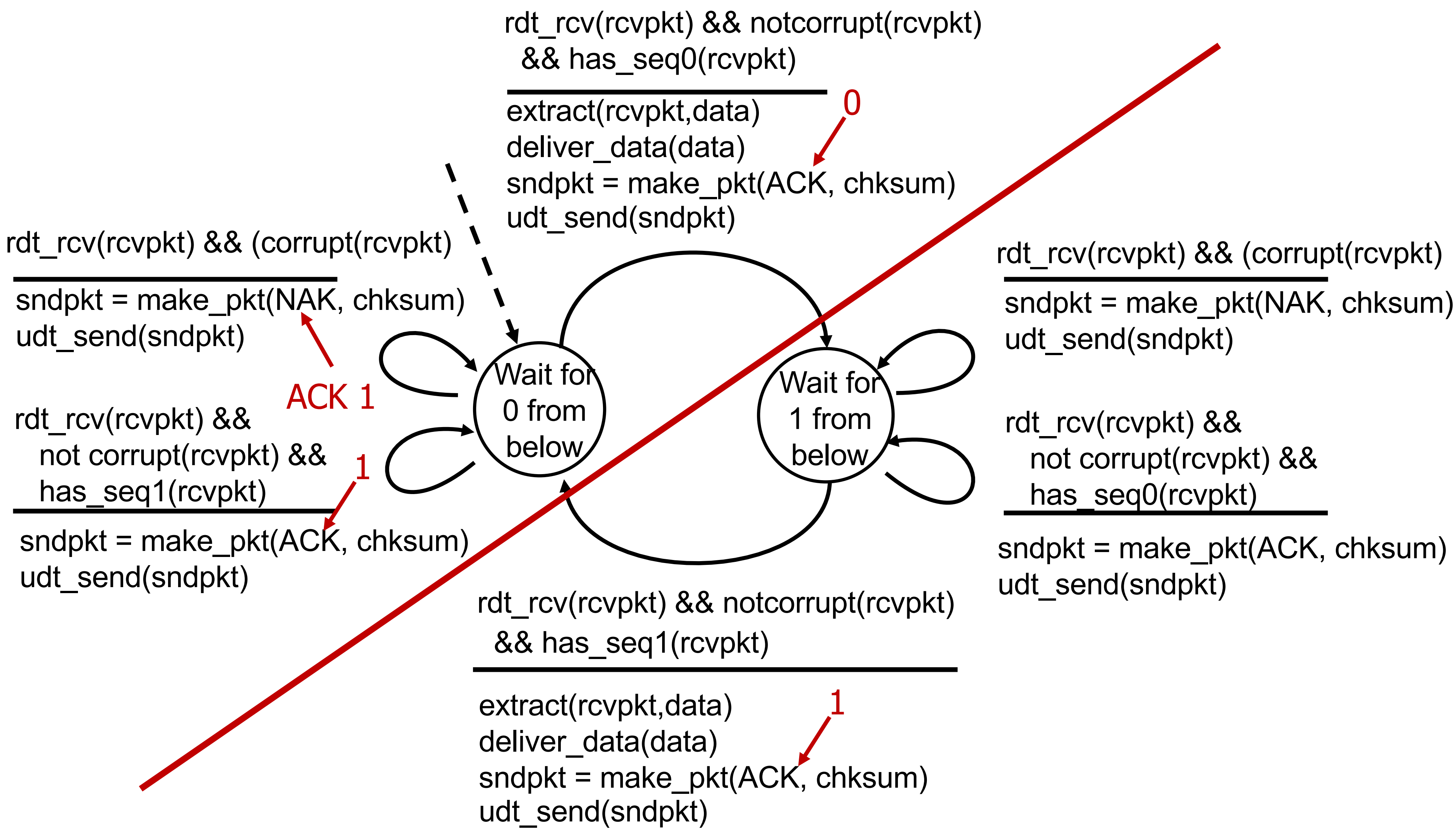
receiver:

- 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
  - If duplicate, not pass the data up, but sends ACK again

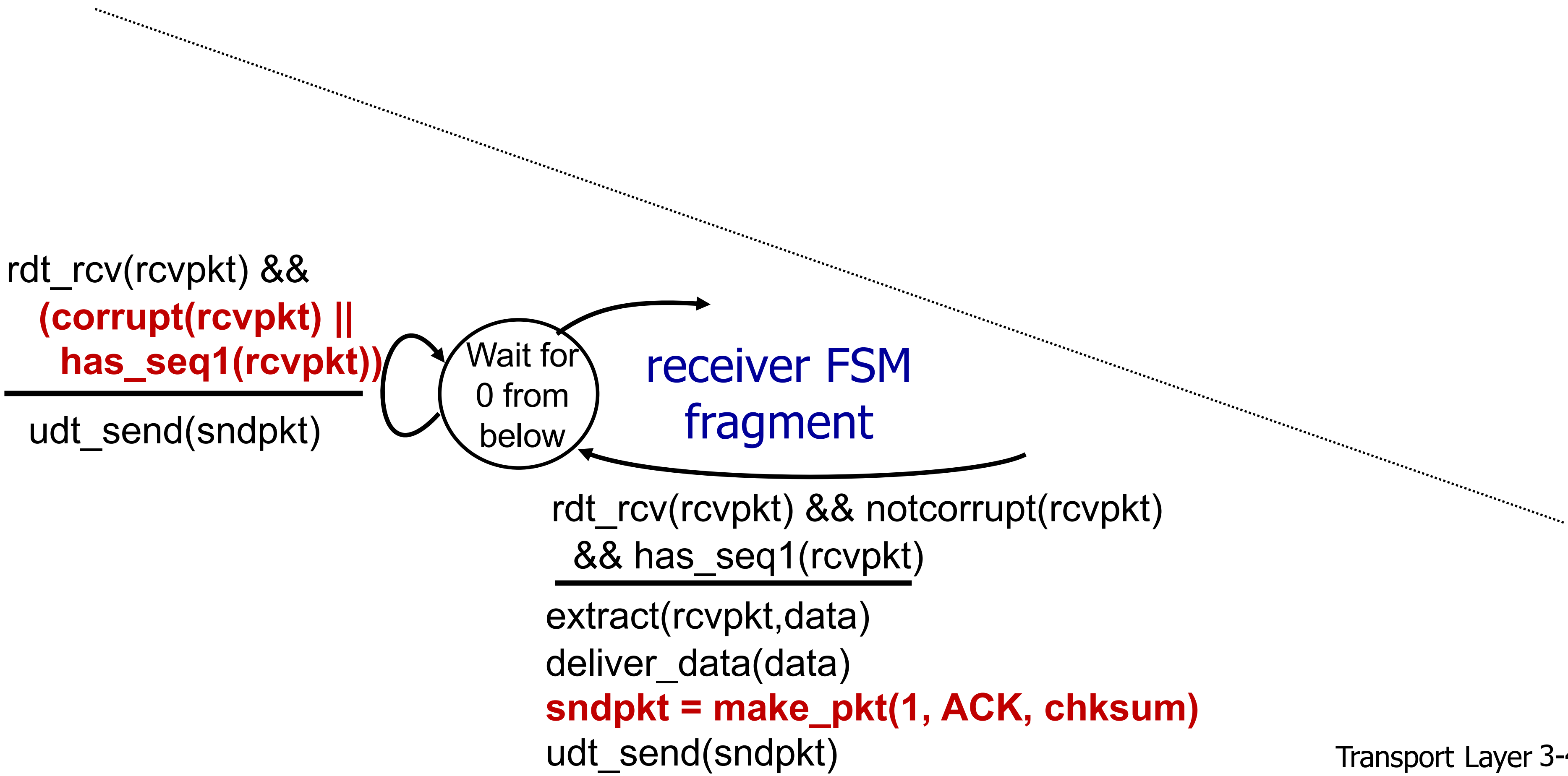
rdt2.2: a NAK-free protocol

- 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*

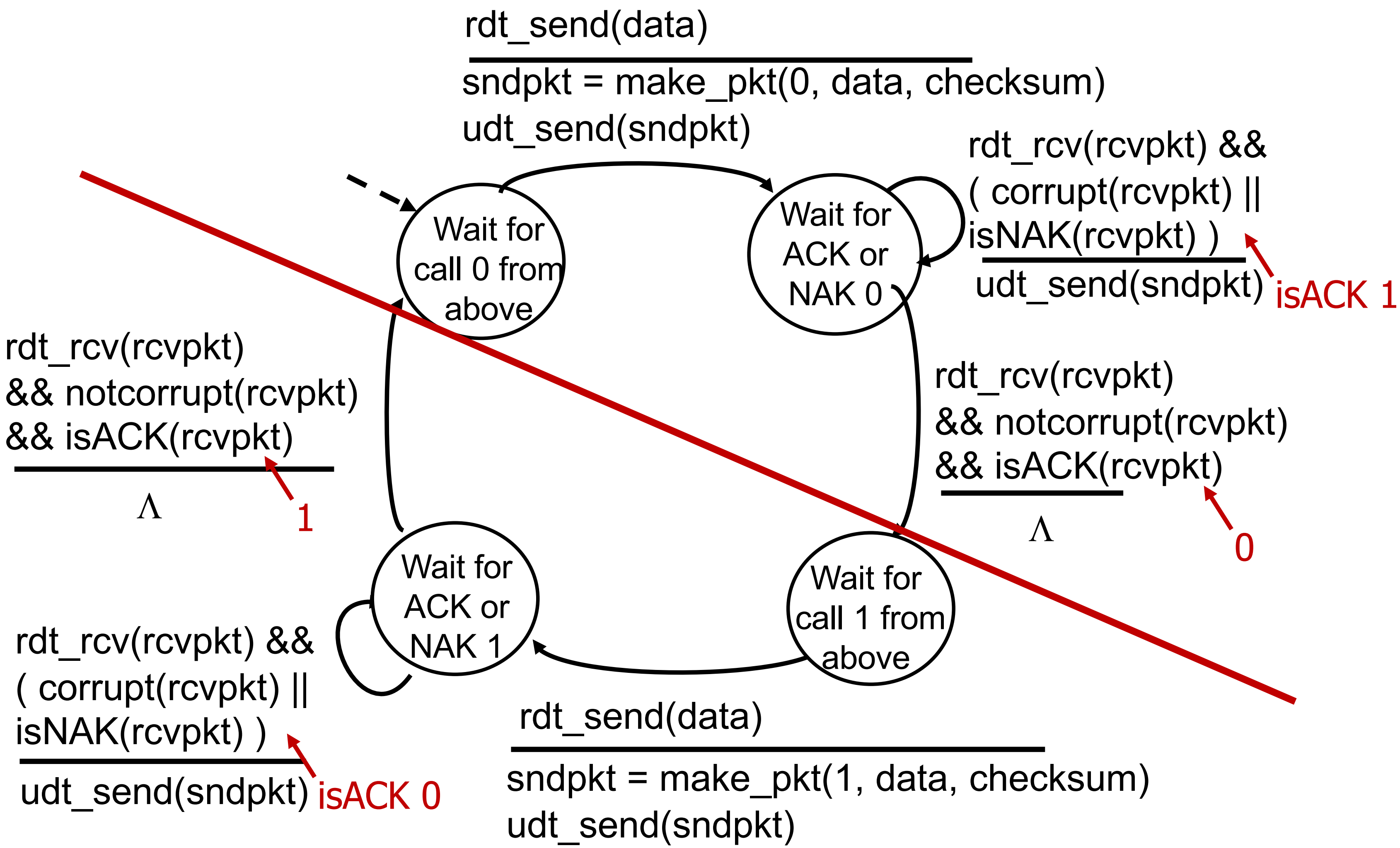
rdt2.1: receiver, handles garbled ACK/NAKs



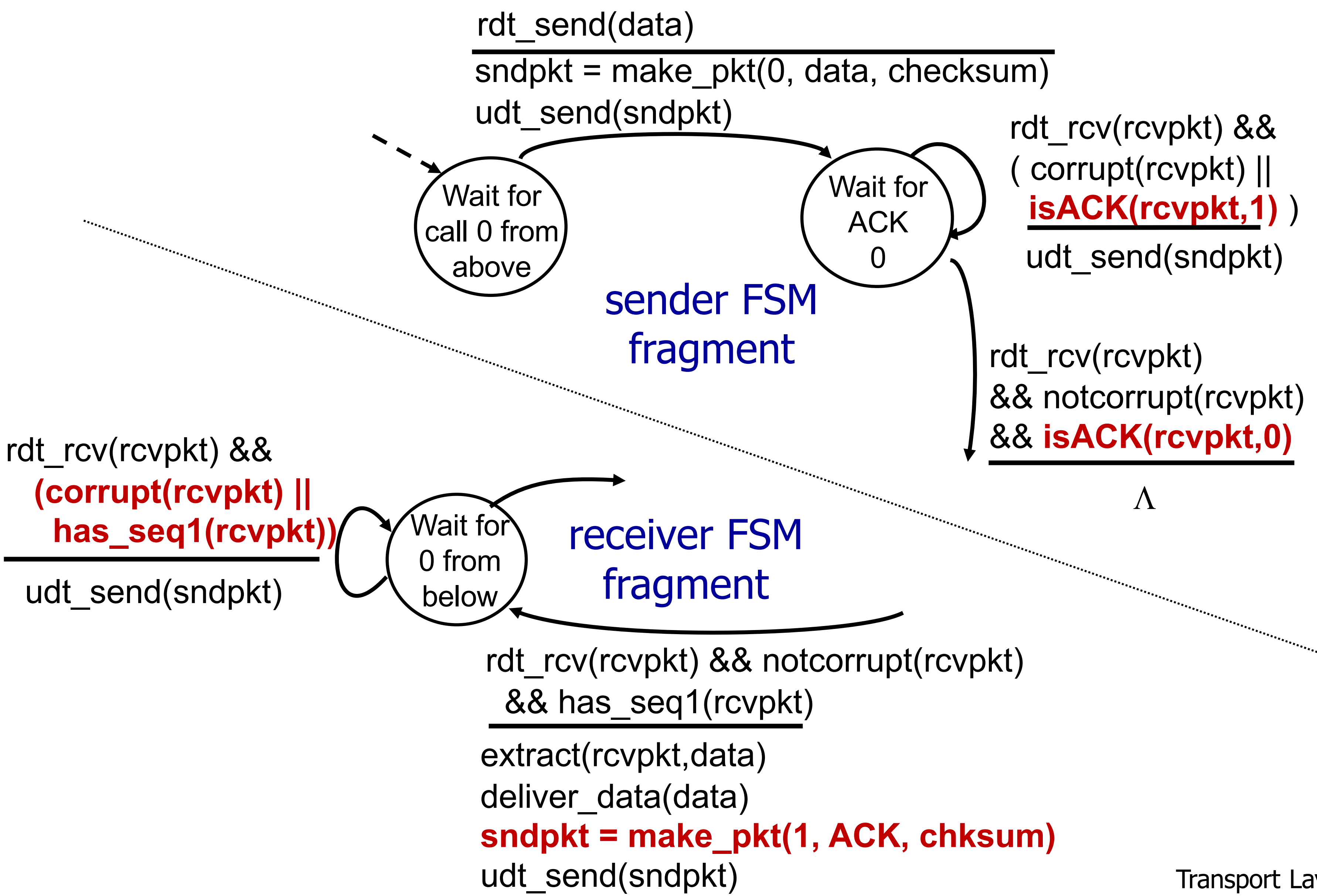
rdt2.2: receiver fragments



rdt2.1: sender, handles garbled ACK/NAKs



rdt2.2: sender, receiver fragments



Quiz Time!

rdt3.0: channels with errors *and* loss

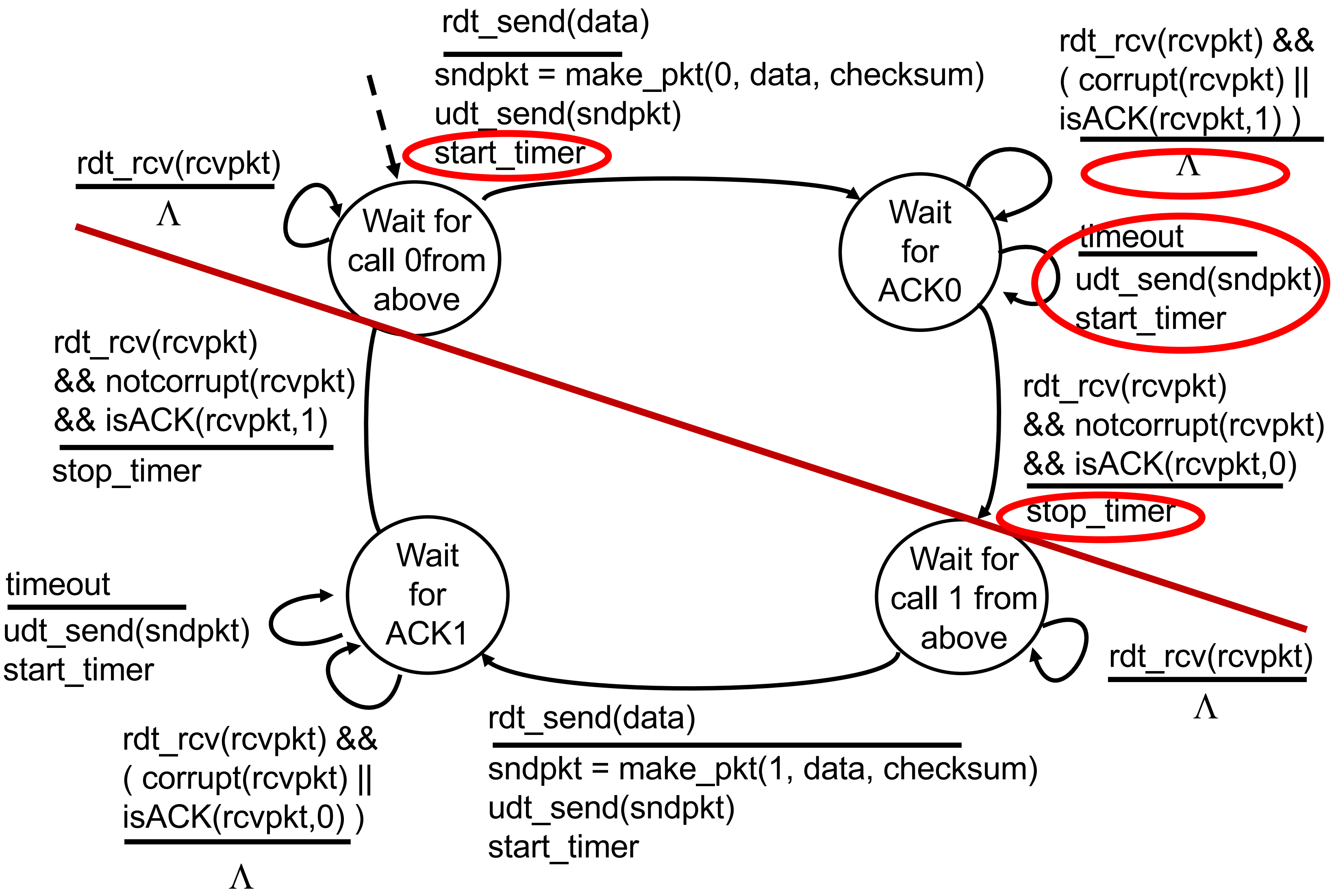
new assumption: underlying channel can also lose packets (data, ACKs)

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

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

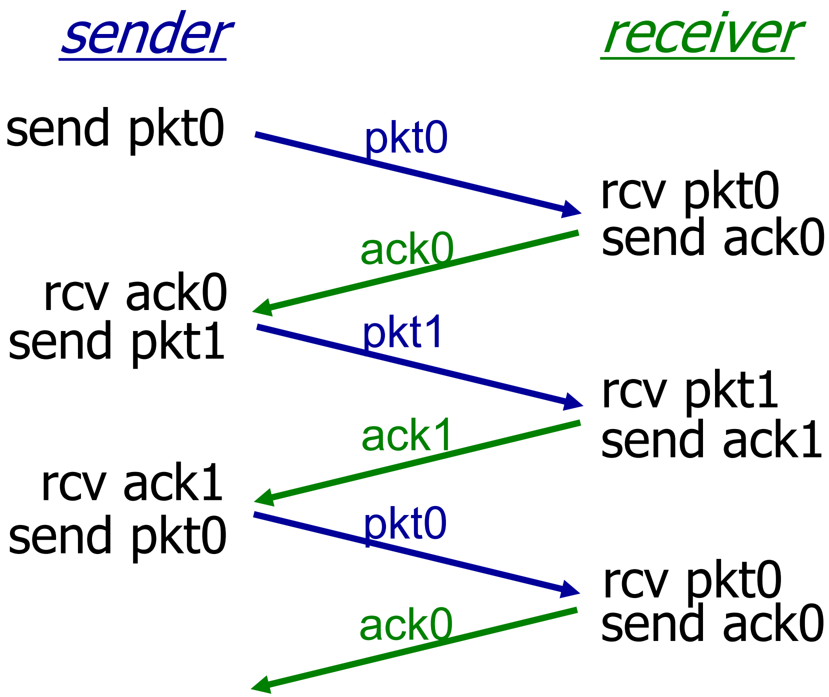
- retransmits if no ACK received in this time
- requires countdown timer
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate. Good that seq. #'s already handling this
  - receiver must specify seq # of pkt being ACKed

rdt3.0 sender

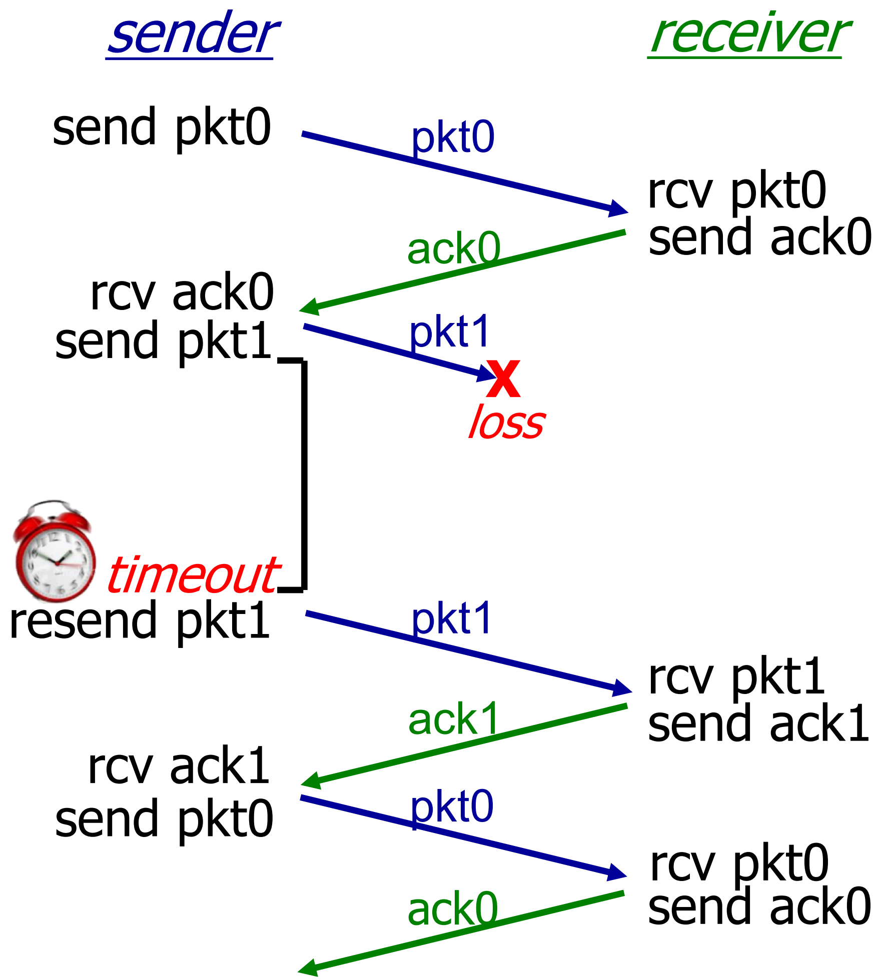


Quiz Time!

rdt3.0 in action

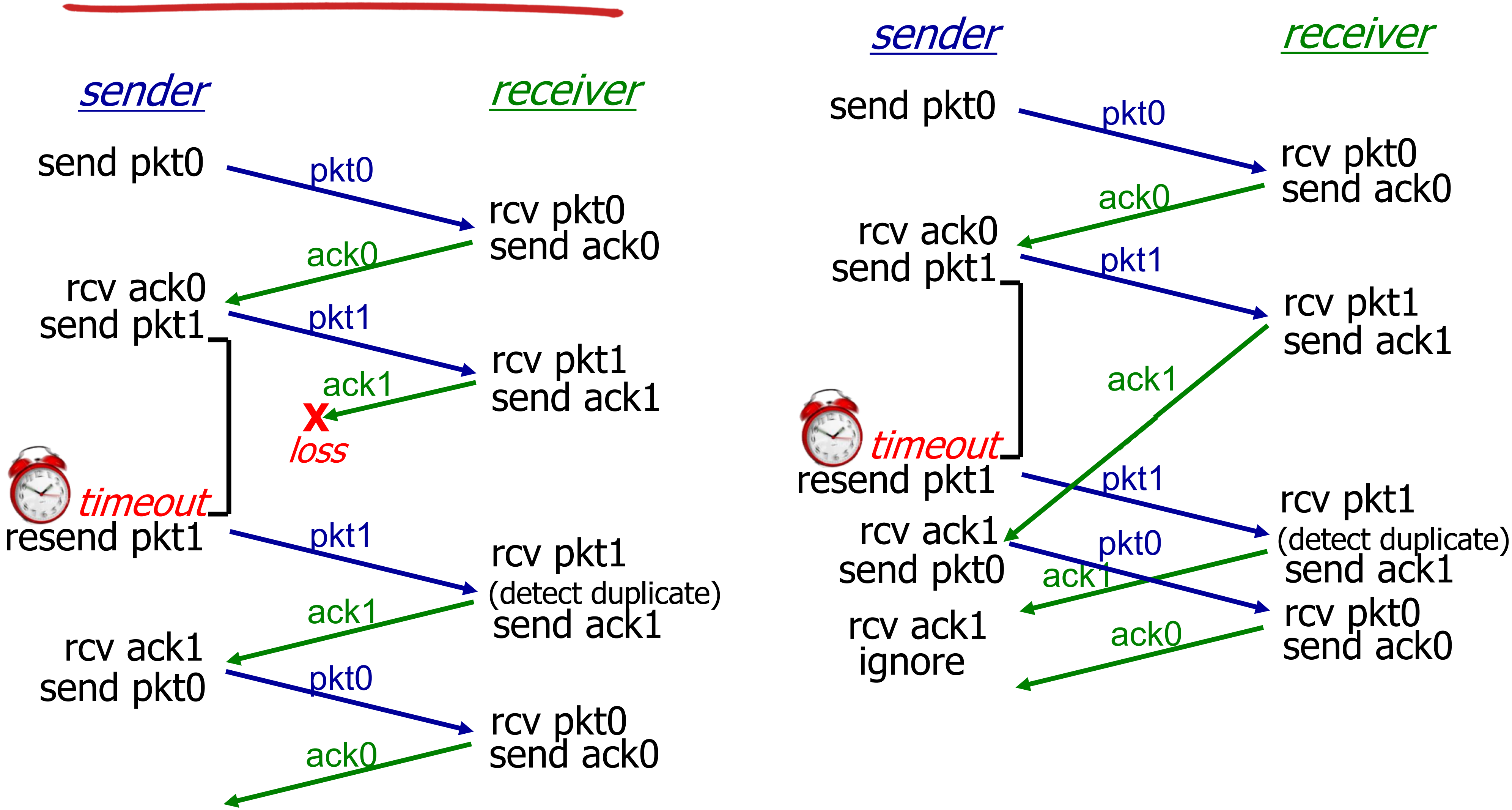


(a) no loss



(b) packet loss

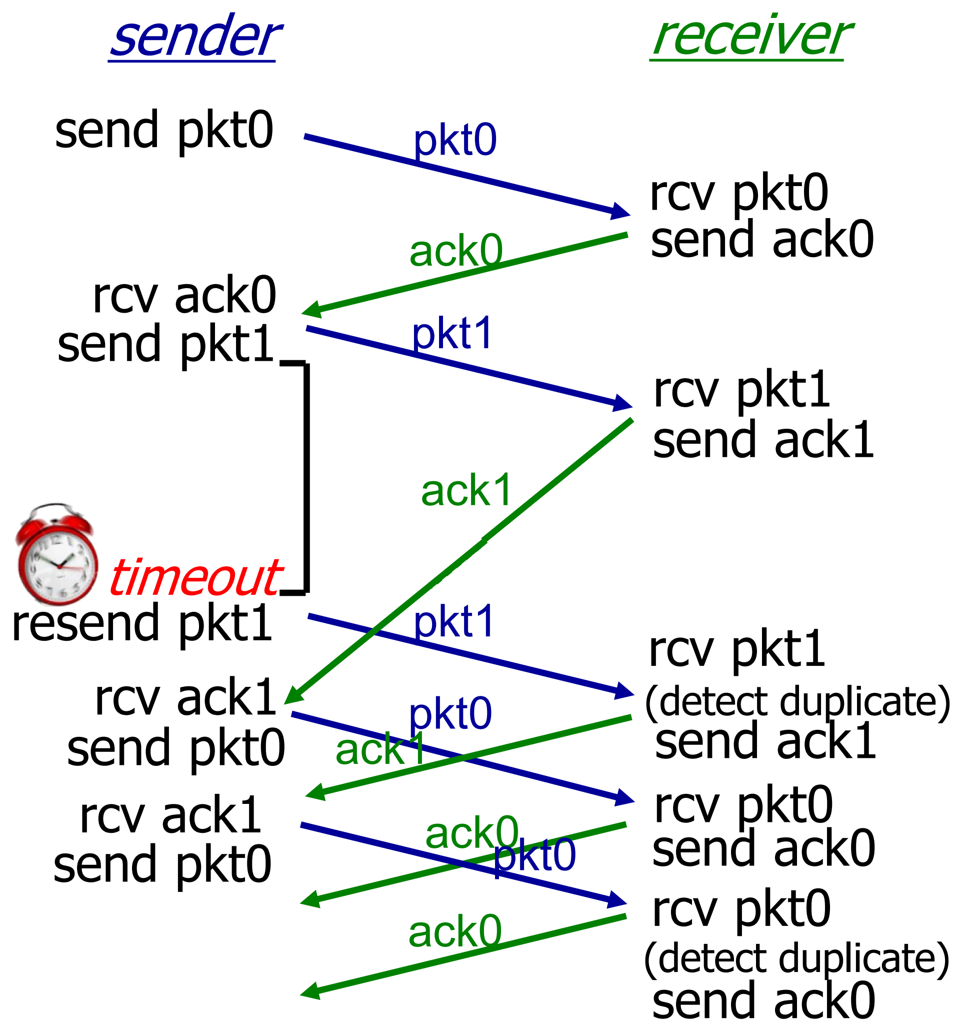
rdt3.0 in action



(c) ACK loss

(d) premature timeout/ delayed ACK

rdt3.0 in action



(e) premature timeout/ delayed ACK
with rdt2.2-like rdt3.0

Performance of rdt3.0

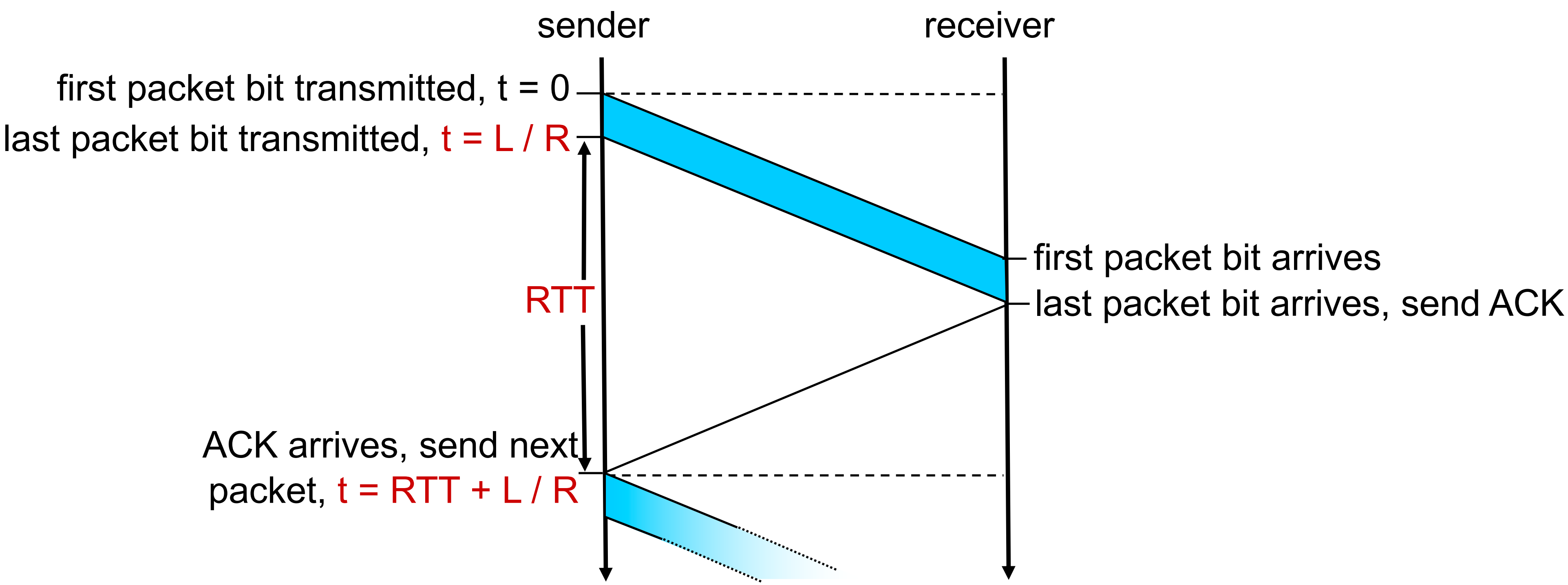
- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

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

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

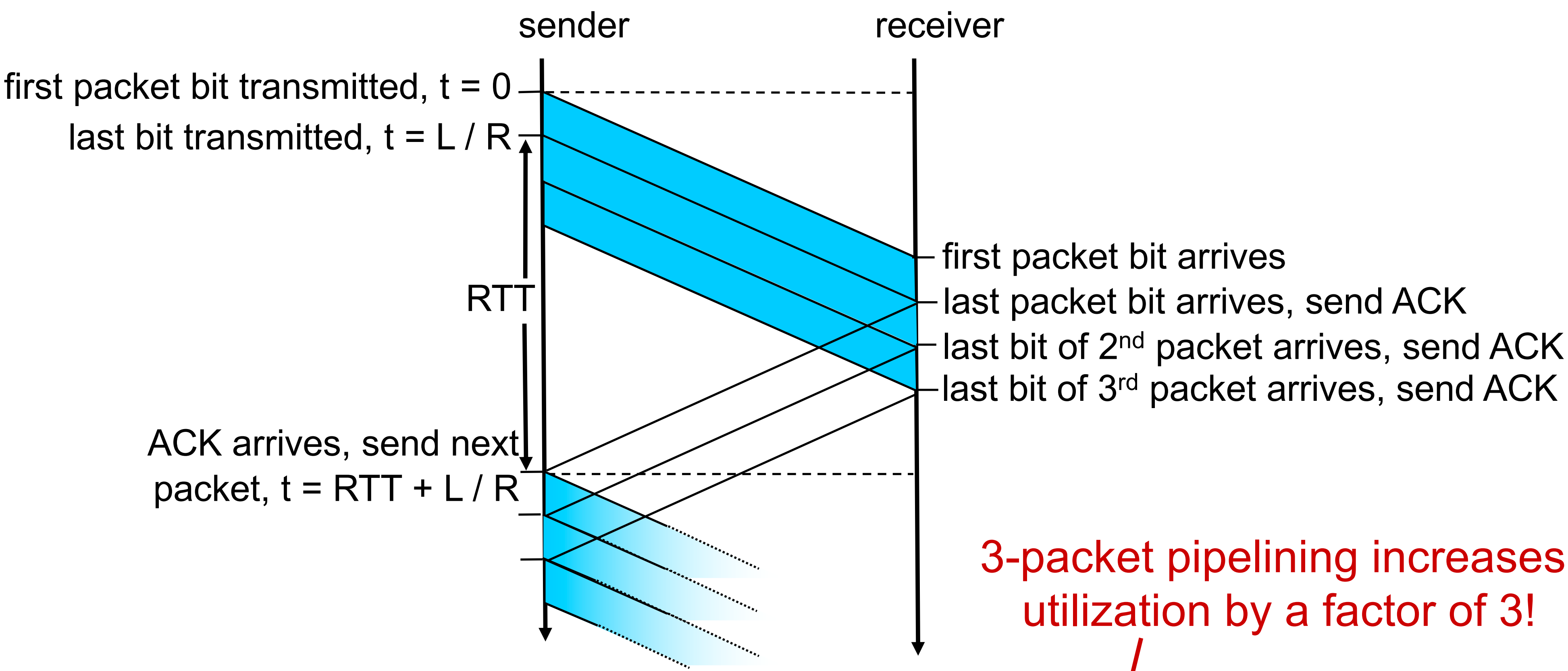
rdt3.0: stop-and-wait operation



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

- Throughput on 1 Gbps link = 0.00027 of 1Gbps = 270 kbps
- Network protocol limits use of physical resources!

Pipelining: increased utilization

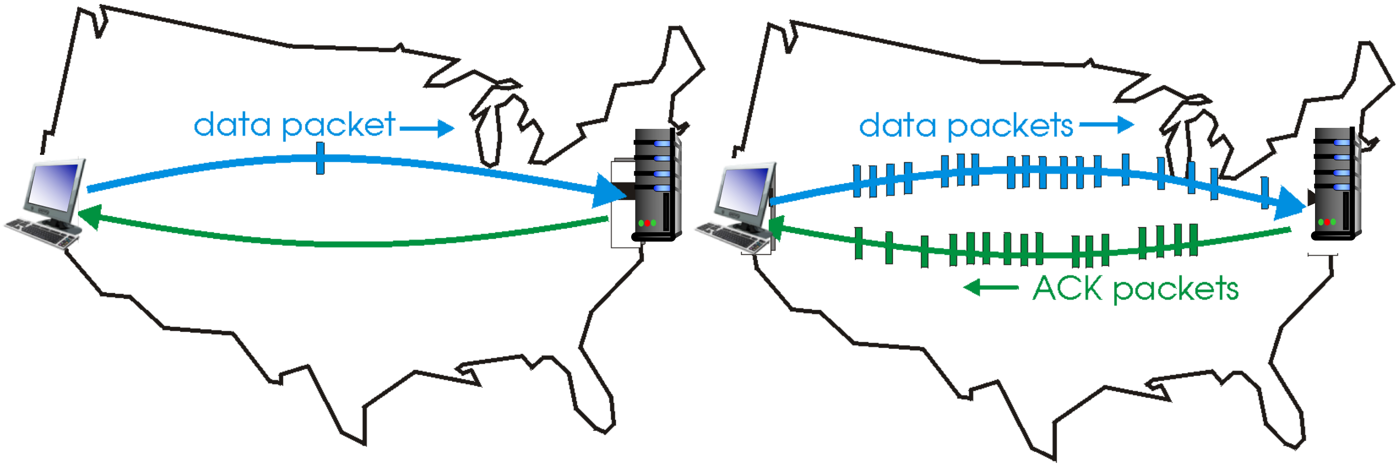


3-packet pipelining increases utilization by a factor of 3!

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

Pipelined protocols

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*

Pipelined protocols: overview

Go-back-N:

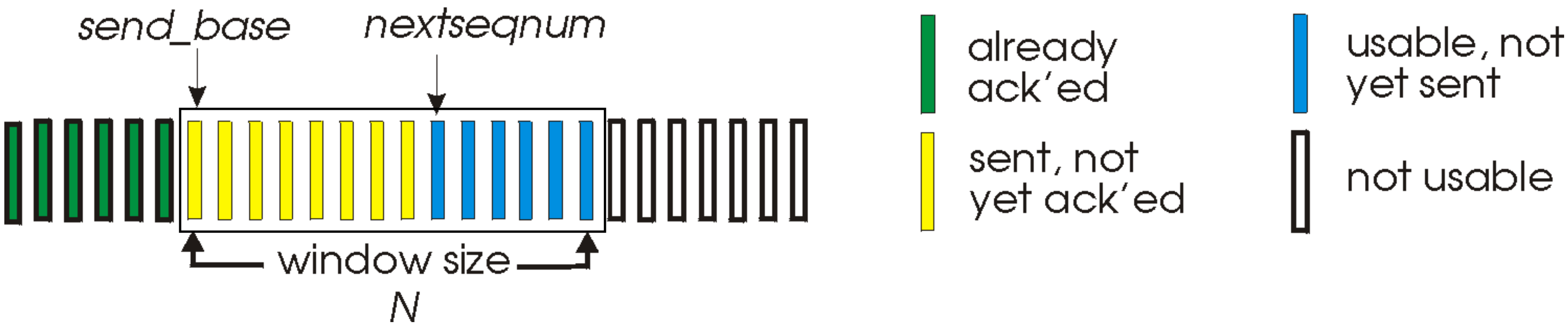
- sender can have up to N unack'ed packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

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

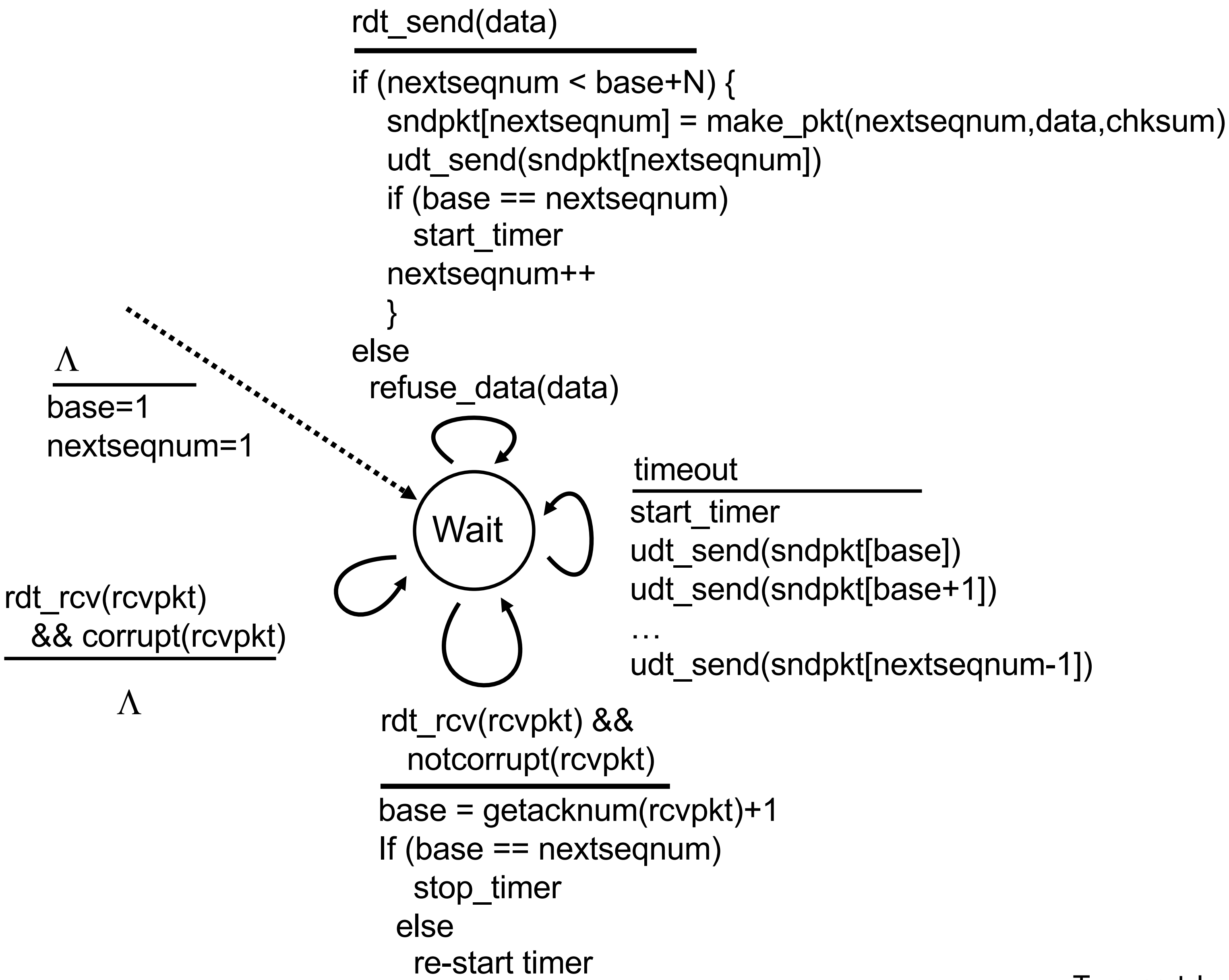
Go-Back-N: sender

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

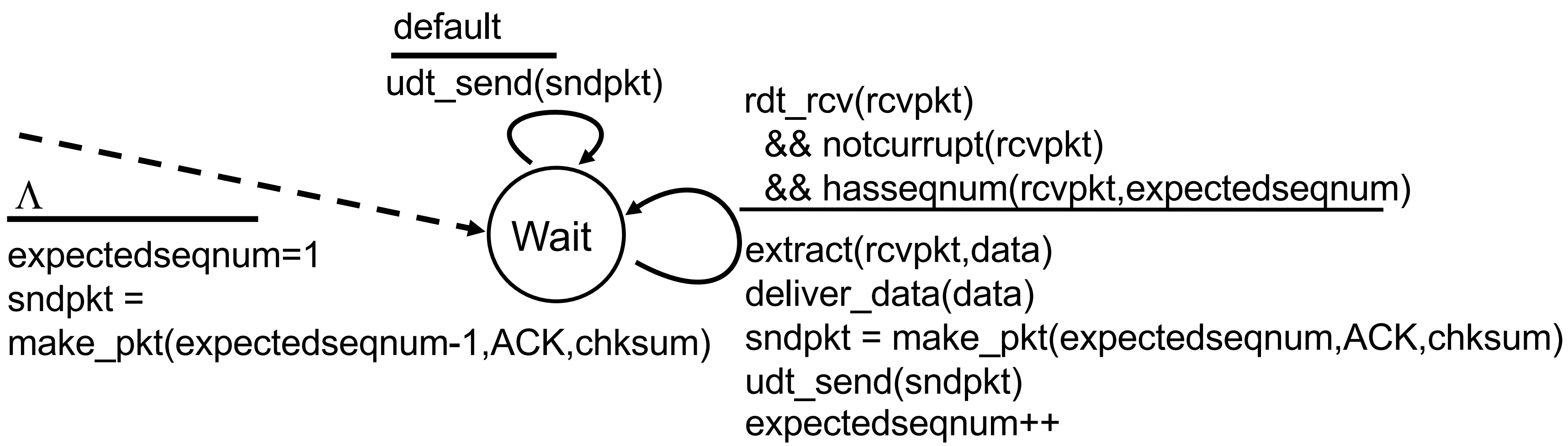


- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight, unack'ed pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM

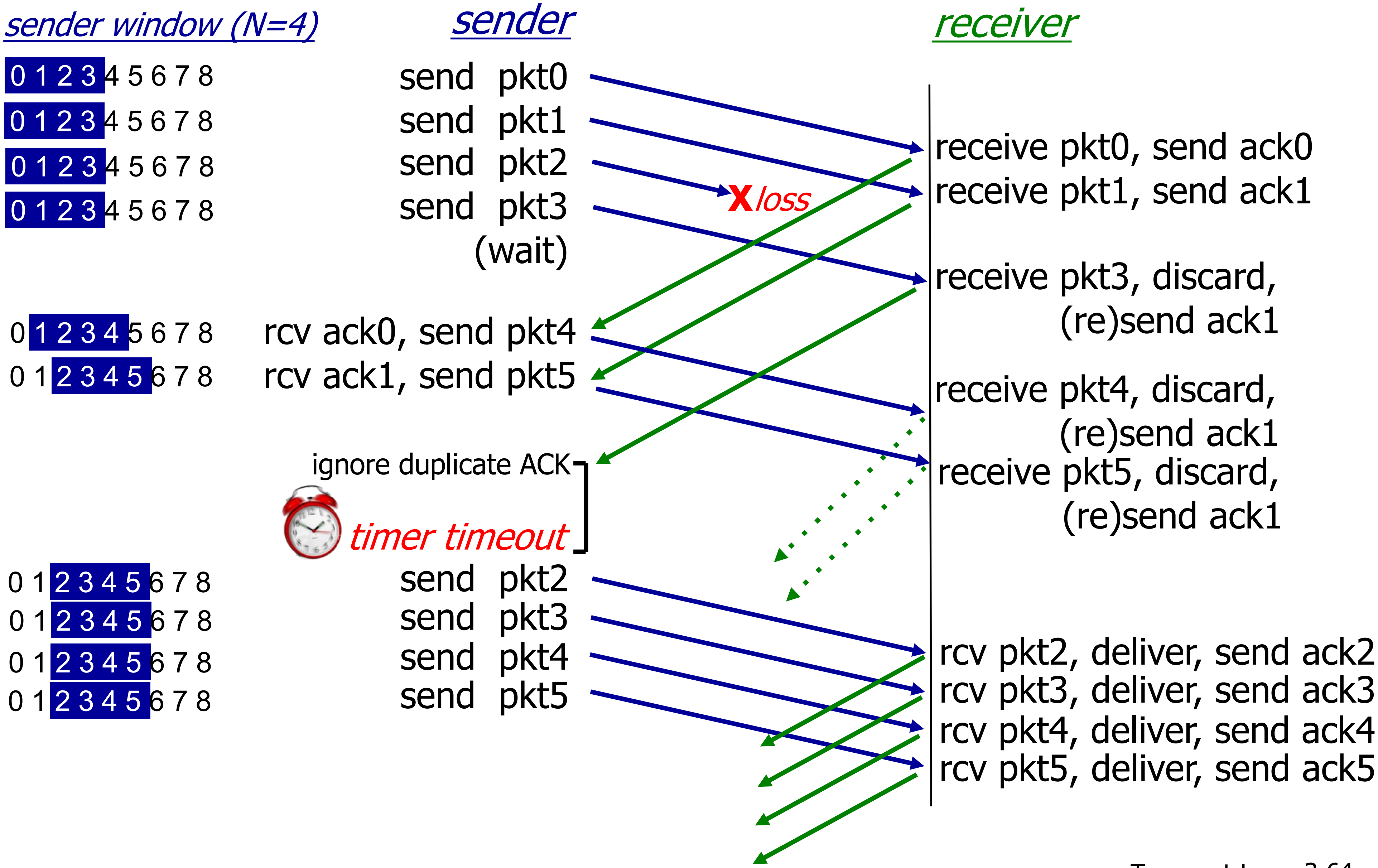


GBN: receiver extended FSM



- Corrupted data or out-of-order data pkt
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #
- Odd but simple
  - ACK for highest *in-order* seq #
  - only need to remember **`expectedseqnum`**

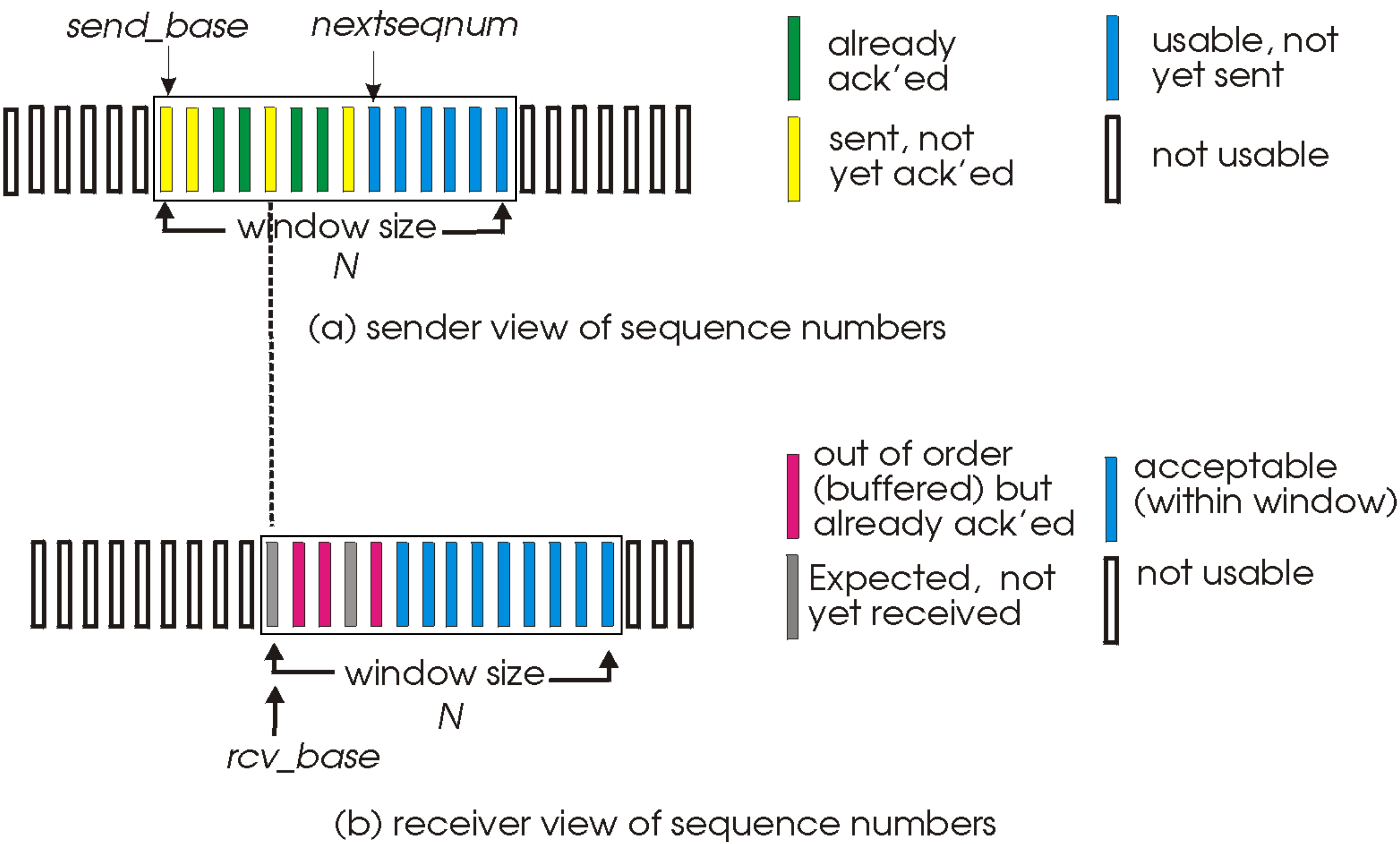
GBN in action



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



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-1]:

- 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

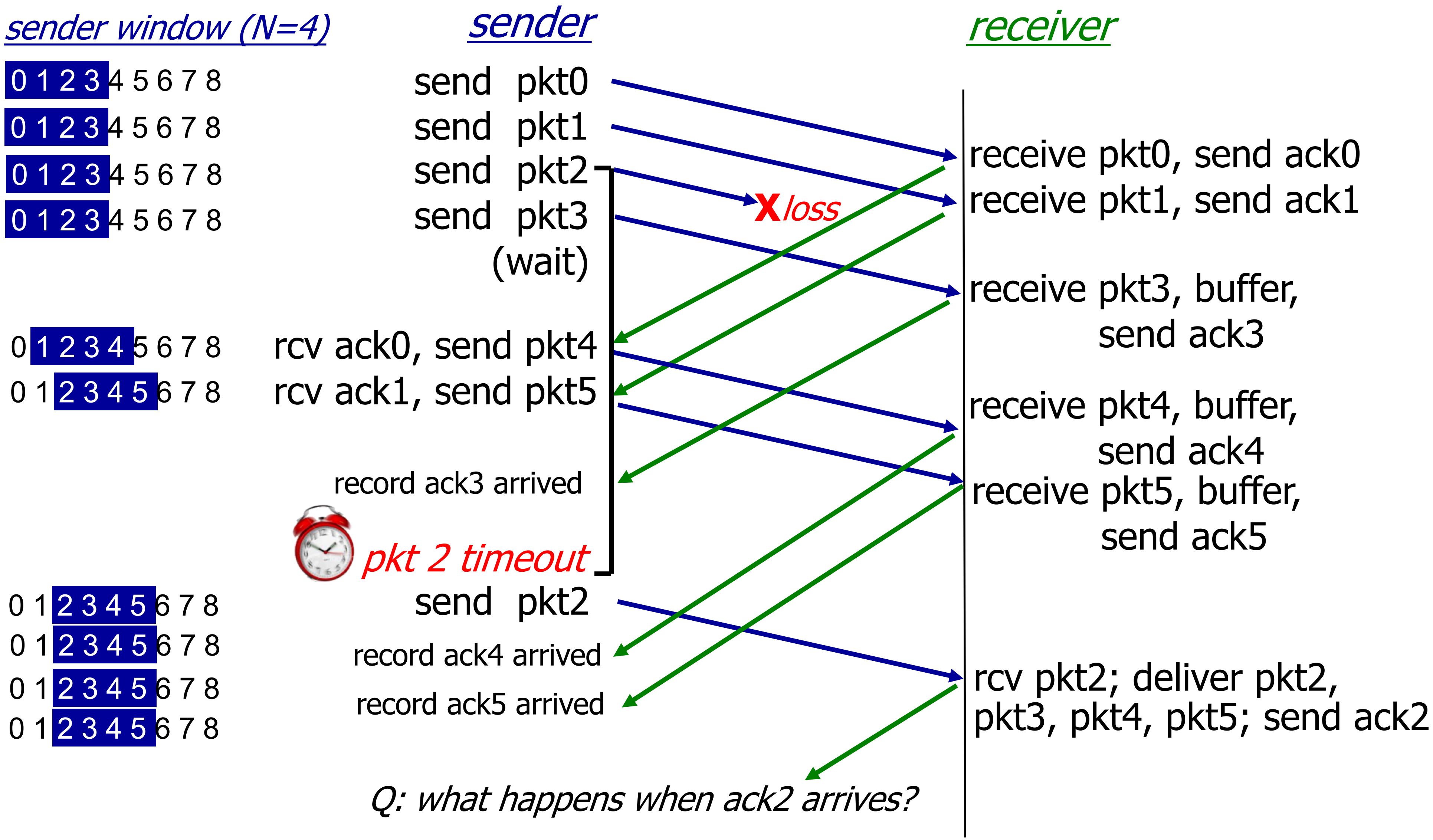
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action

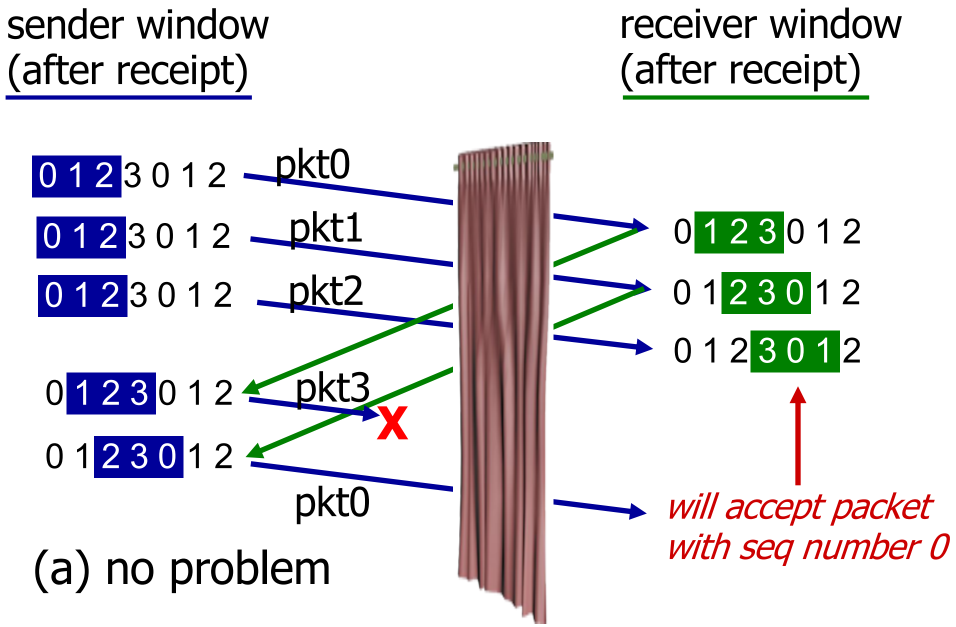


Selective repeat: dilemma

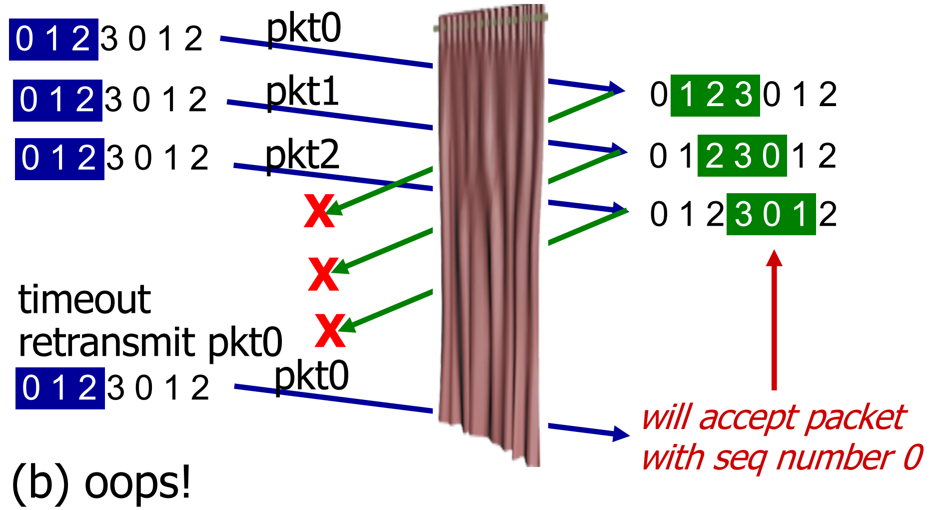
example:

- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

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



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



Pipelined protocols: summary

Go-back-N:

- sender can have up to N unack'ed 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

Quiz Time!

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

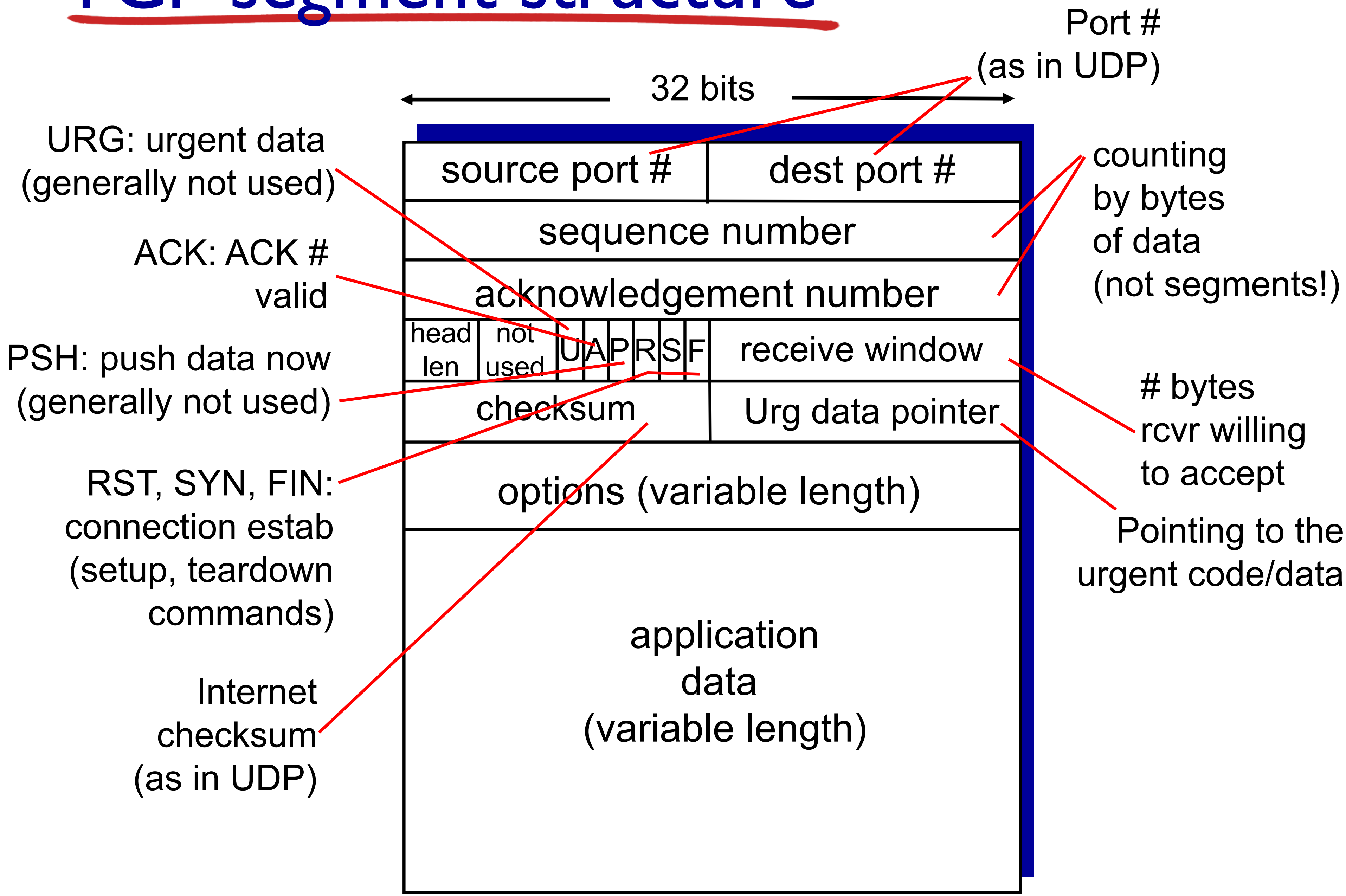
3.6 principles of congestion control

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"
- 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



TCP seq. numbers, ACKs

sequence numbers:

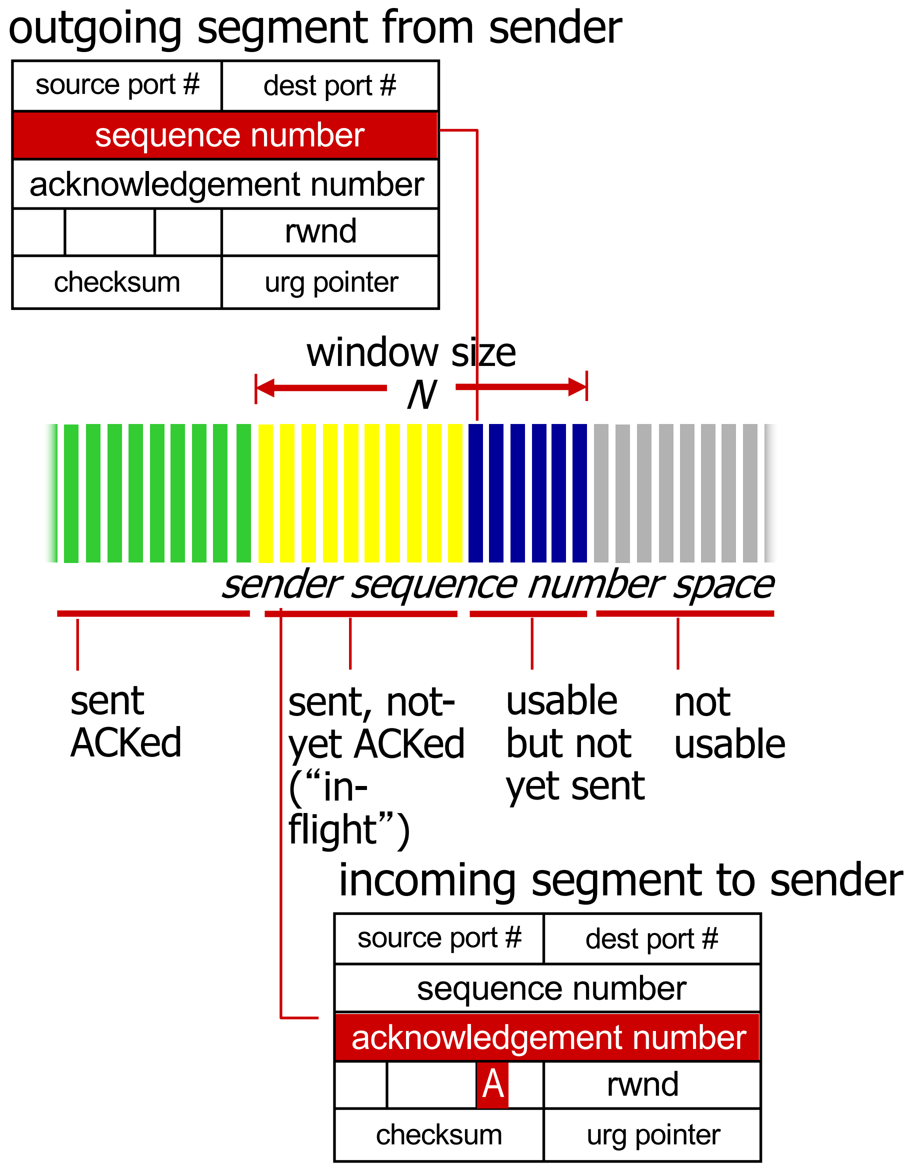
- byte stream "number" of first byte in segment's data

acknowledgements:

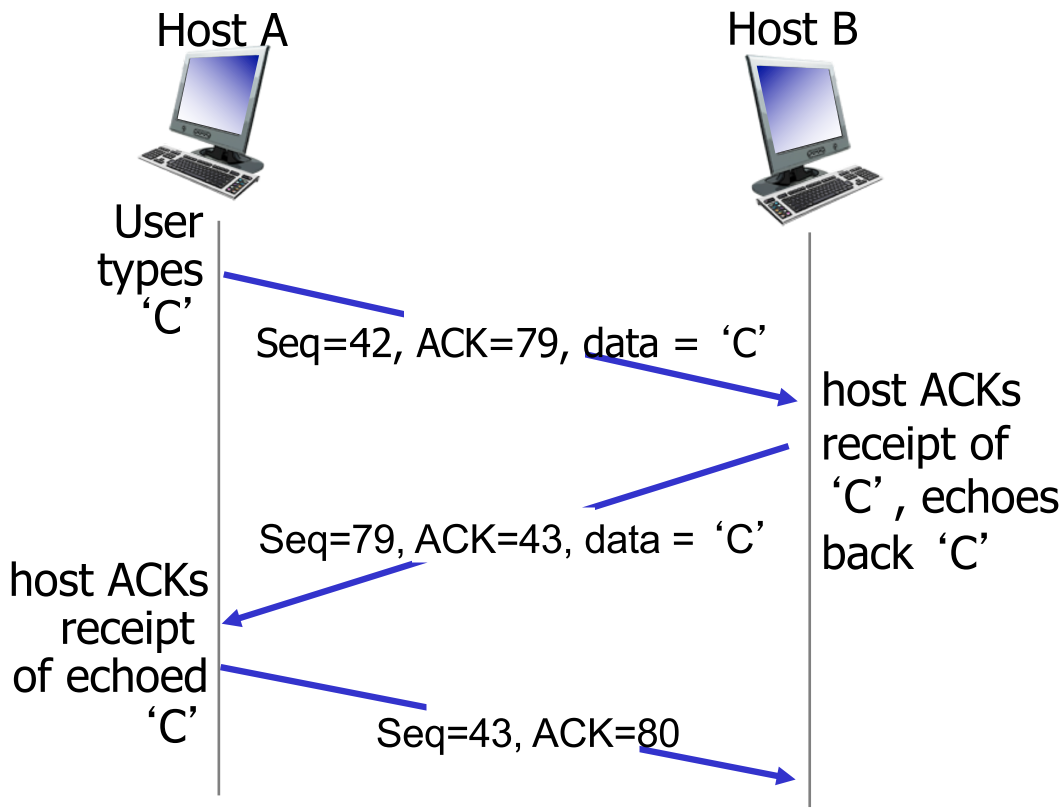
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACKs



simple telnet scenario

Quiz Time!

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 round trip time, timeout

Q: how to set TCP timeout value?

- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

TCP round trip time, timeout

Q: how to set TCP timeout value?

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

Q: how to estimate RTT?

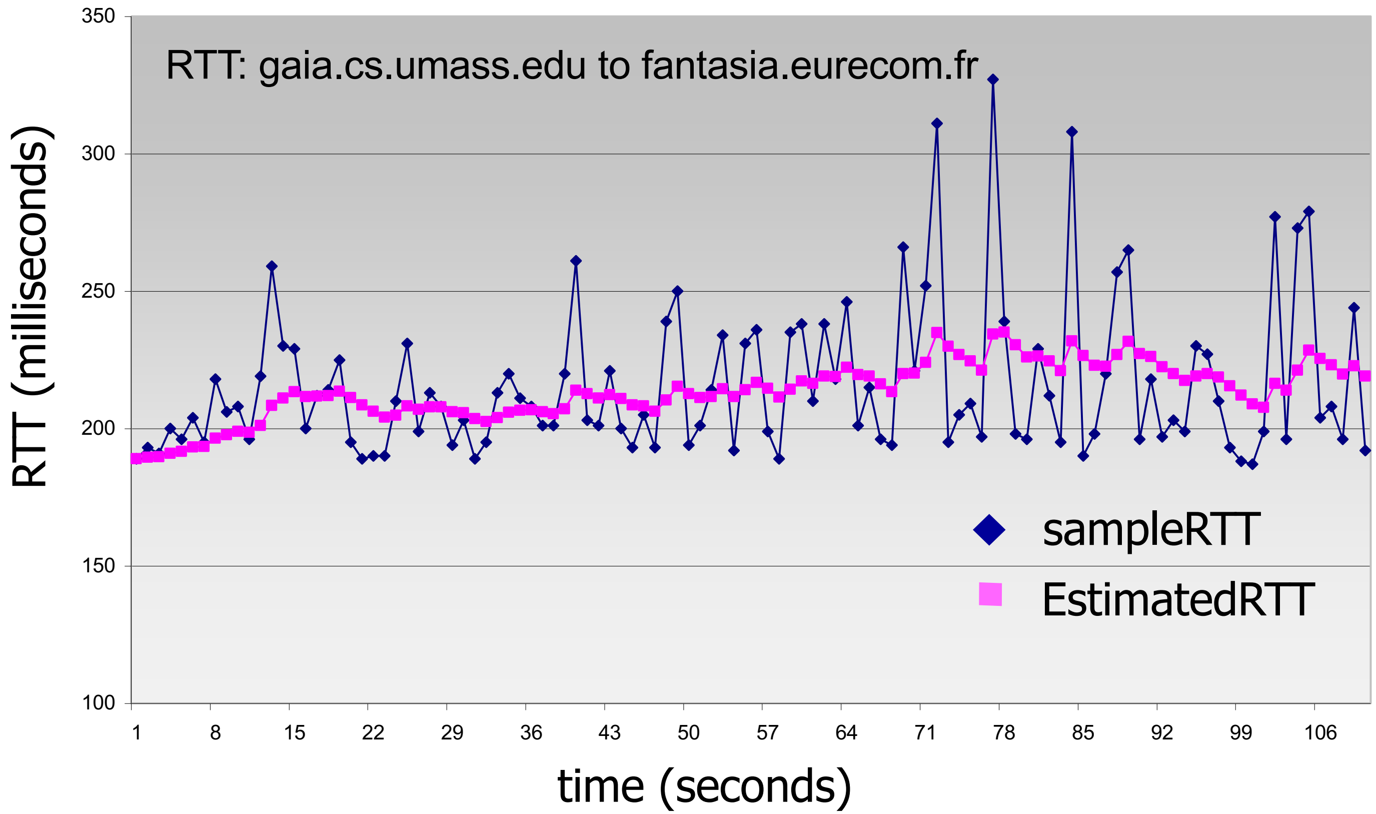
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

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

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

TCP round trip time, timeout



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`SampleRTT`** -> 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"

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

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

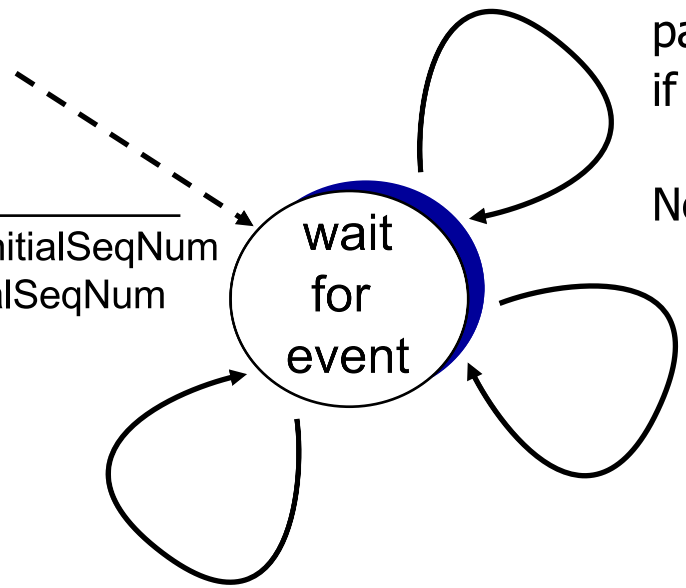
3.7 TCP congestion control

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

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

TCP sender (simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

---

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

timeout

---

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

ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase: oldest unACKed byte */
   if (there are currently not-yet-acked segments)
       re-start timer
   else
       stop timer
}
```

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
- start timer

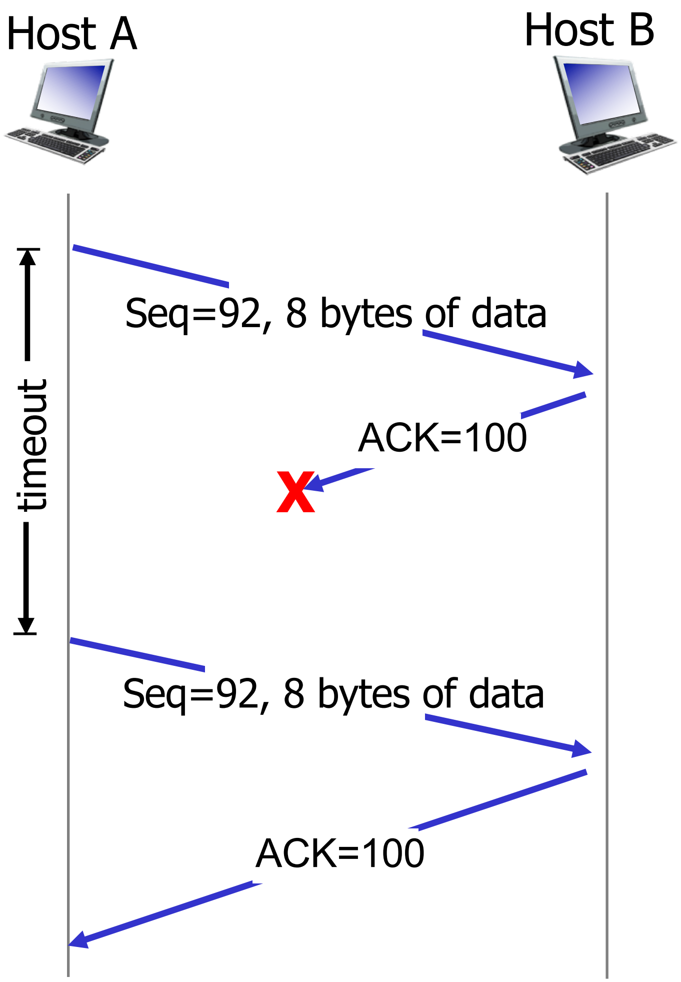
ack rcvd:

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

TCP sender (simplified)

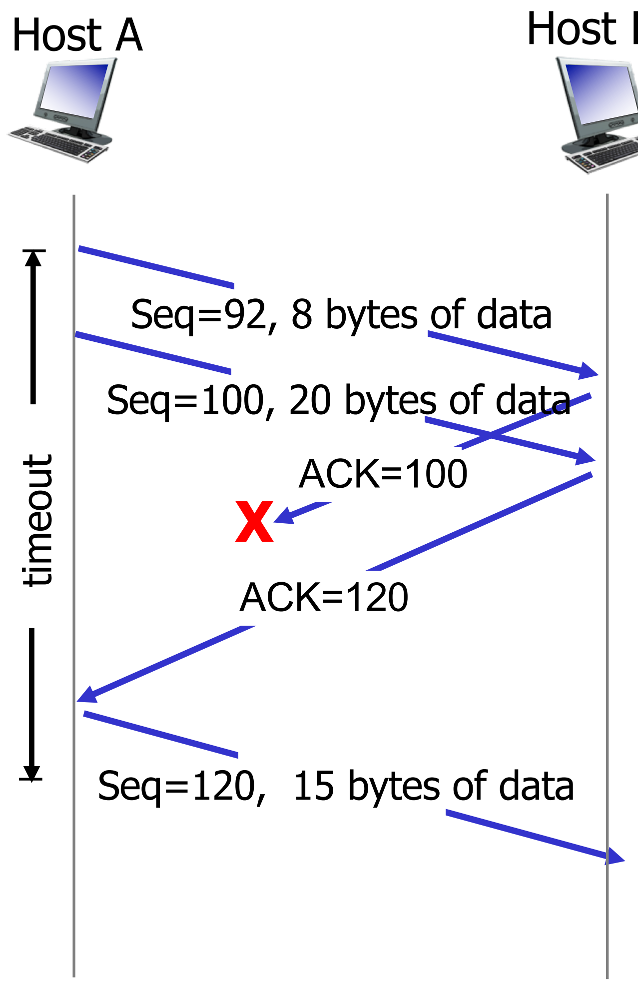
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

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

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

    event: ACK received, with ACK field value of y
        if (y > SendBase) {
             SendBase = y
             if (there are currently not-yet-acknowledged segments)
                       re-start timer
             else   stop timer
        }
 }  /* end of loop forever */
```

TCP: retransmission scenarios

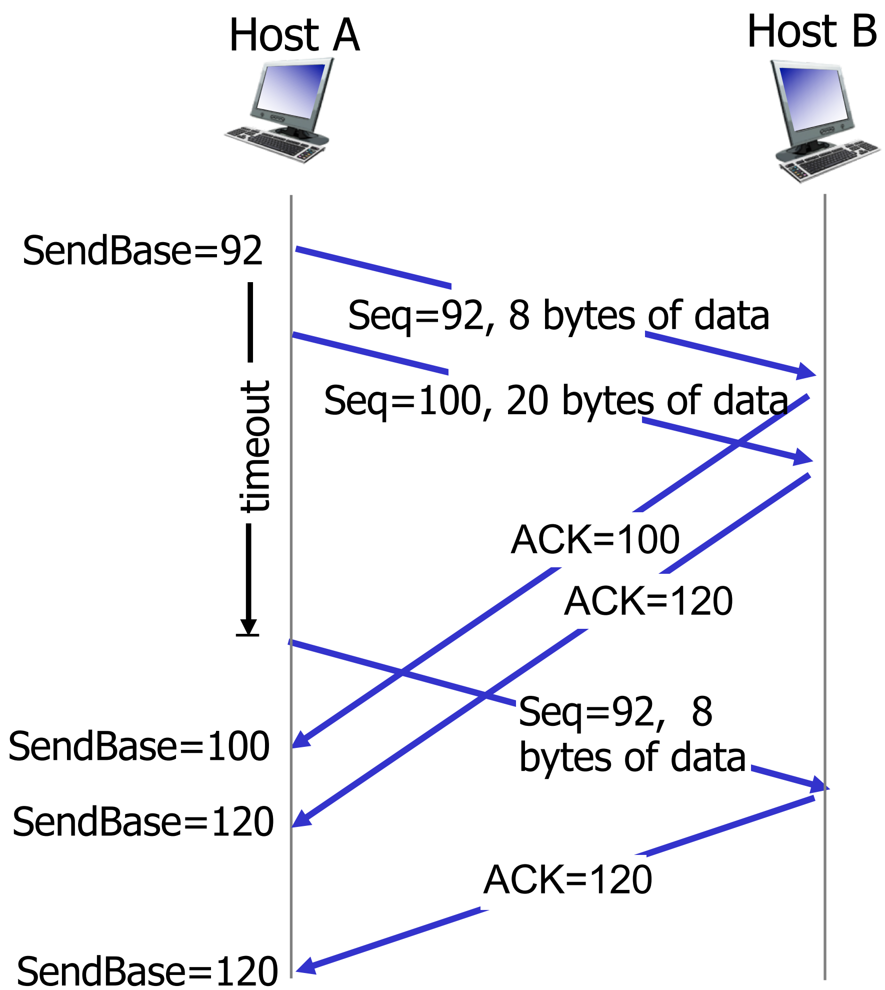


lost ACK scenario

TCP: retransmission scenarios

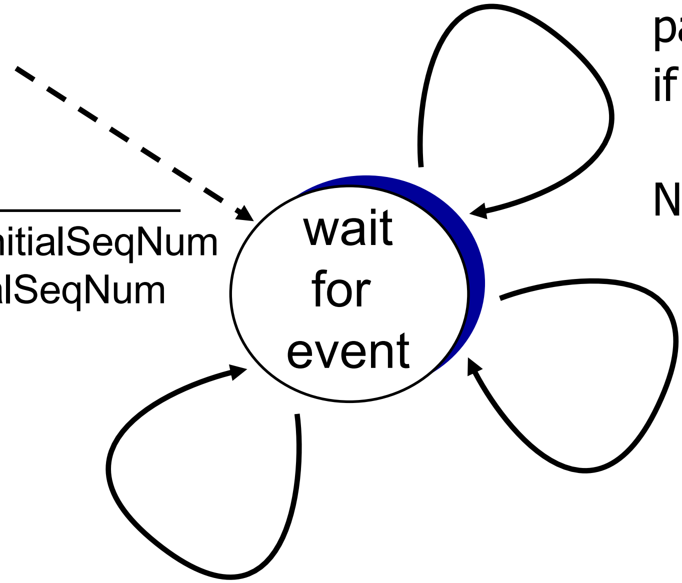


cumulative ACK



premature timeout

TCP sender (simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

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

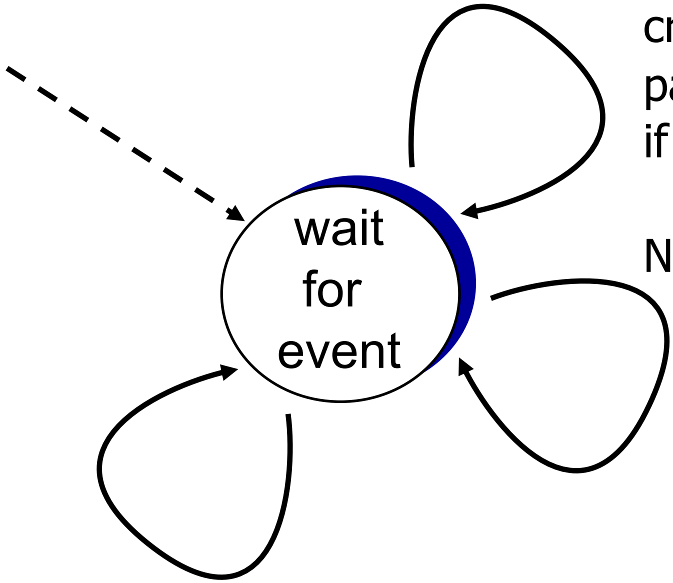
timeout

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

ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase: oldest unACKed byte */
   if (there are currently not-yet-acked segments)
        re-start timer
   else
        stop timer
}
```

TCP sender (not so simplified)



data received from application above

TimeoutInterval = EstimatedRTT + 4*DevRTT
create segment, seq#=NextSeqNum
pass segment to IP (i.e., "udt_send")
if (timer currently not running)
    start timer
NextSeqNum = NextSeqNum + length(data)

timeout

---

wnd = 1
retransmit one not-yet-acked segment
    with smallest seq#
TimeoutInterval = 2 * TimeoutInterval
start timer

ACK received, with ACK field value y

TimeoutInterval = EstimatedRTT + 4*DevRTT
if (y > SendBase) {
  SendBase = y
  /* SendBase: oldest unACKed byte */
  if (there are currently not-yet-acked segments)
    re-start timer
  else
    stop timer }

TCP's rdt vs. GBN

- Similarity
  - pipelined segments
  - cumulative acks
  - single retransmission timer
  - retransmission on timeout
- Unique in TCP
  - timeout interval well defined
  - only new ack refresh timer
- Also unique in TCP
  - retransmit only one segment
    - sending window = 1 segment
  - timeout interval doubled
    - exponentially slow in case of back-to-back timeout events

TCP's rdt vs. GBN

- Similarity
  - pipelined segments
  - cumulative acks
  - single retransmission timer
  - retransmission on timeout
- Unique in TCP
  - timeout interval well defined
  - only new ack refresh timer
- Also unique in TCP
  - retransmit only one segment
    - sending window = 1 segment
  - timeout interval doubled
    - exponentially slow in case of back-to-back timeout events
- More being unique in TCP
  - delayed ack
  - fast retransmission

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 cumulative ACK, provided that segment starts at lower end of gap

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 cumulative ACK, provided that segment starts at lower end of gap

Quiz Time!

TCP's rdt vs. GBN

- Similarity
  - pipelined segments
  - cumulative acks
  - single retransmission timer
  - retransmission on timeout
- Unique in TCP
  - timeout interval well defined
  - only new ack refresh timer
- Also unique in TCP
  - retransmit only one segment
    - sending window = 1 segment
  - timeout interval doubled
    - exponentially slow in case of back-to-back timeout events
- More being unique in TCP
  - delayed ack
    - receiver buffers out-of-order packets
  - fast retransmission

TCP fast retransmit

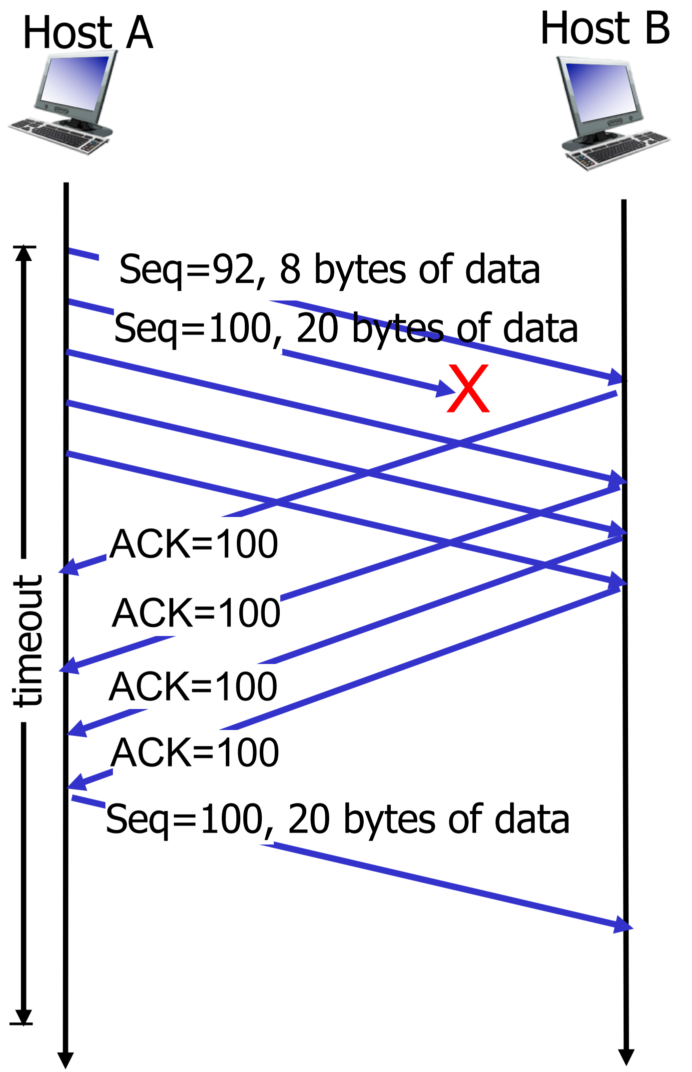
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

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

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

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  re-start timer
              else
                  stop timer
          }
          else {
              increment count of dup ACKs received for y
              if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                    count of dup ACK = 0
          }
```

a duplicate ACK for already ACKed segment

fast retransmit

TCP's rdt vs. GBN

- Similarity
  - pipelined segments
  - cumulative acks
  - single retransmission timer
  - retransmission on timeout
- Unique in TCP
  - timeout interval well defined
  - only new ack refresh timer
- Also unique in TCP
  - retransmit only one segment
    - sending window = 1 segment
  - timeout interval doubled
    - exponentially slow in case of back-to-back timeout events
- More being unique in TCP
  - delayed ack
    - receiver buffers out-of-order packets
  - fast retransmission
    - retransmission on 3 dup acks

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

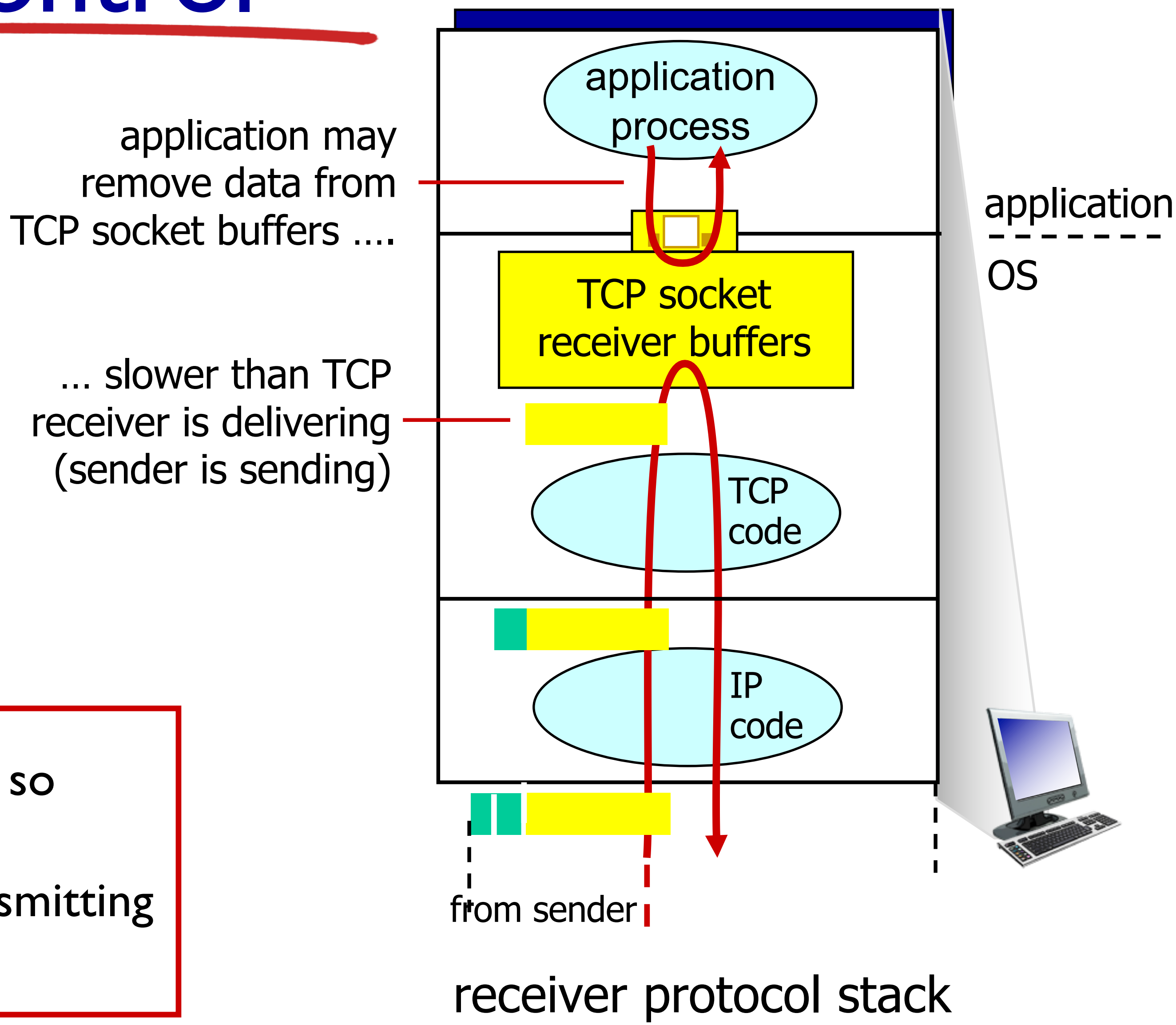
3.4 principles of reliable data transfer

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

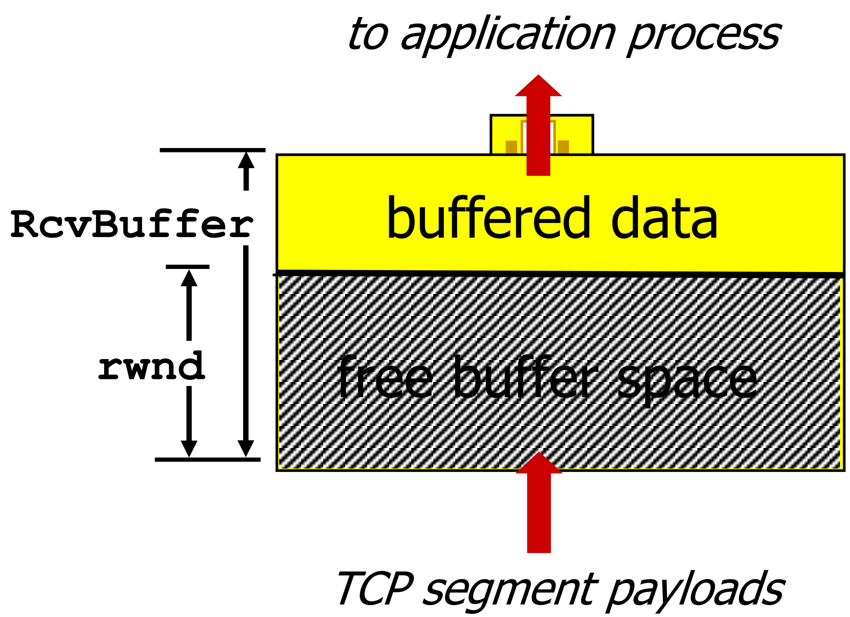


flow control

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

TCP flow control

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



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

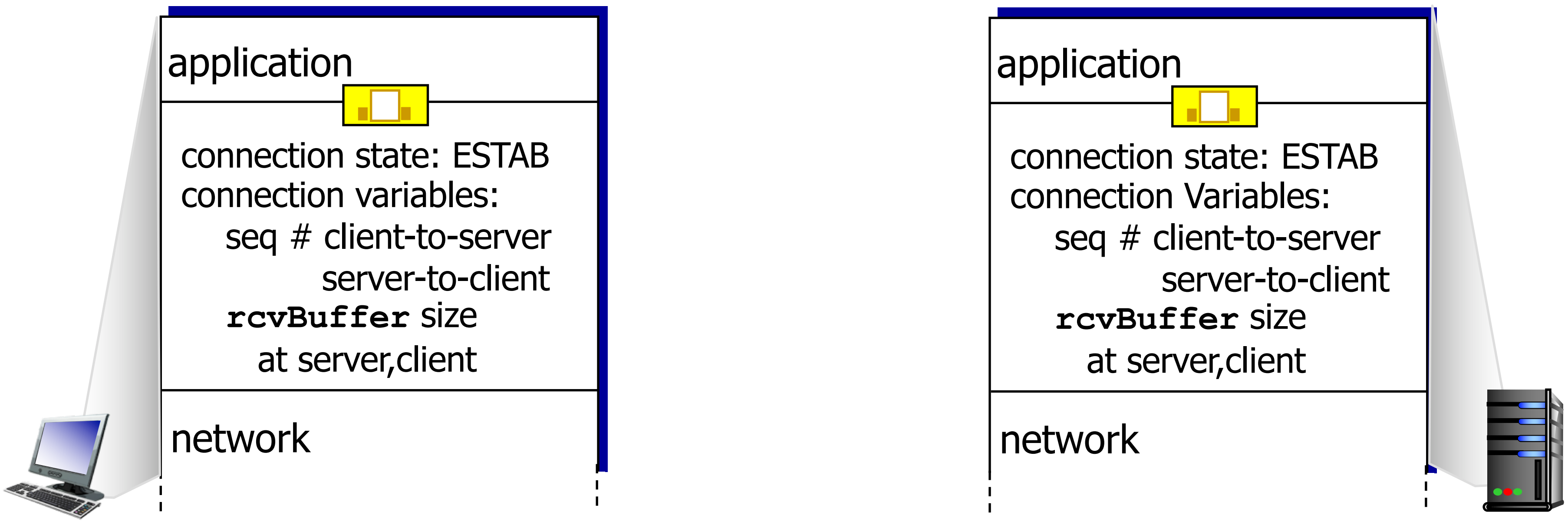
3.6 principles of congestion control

3.7 TCP congestion control

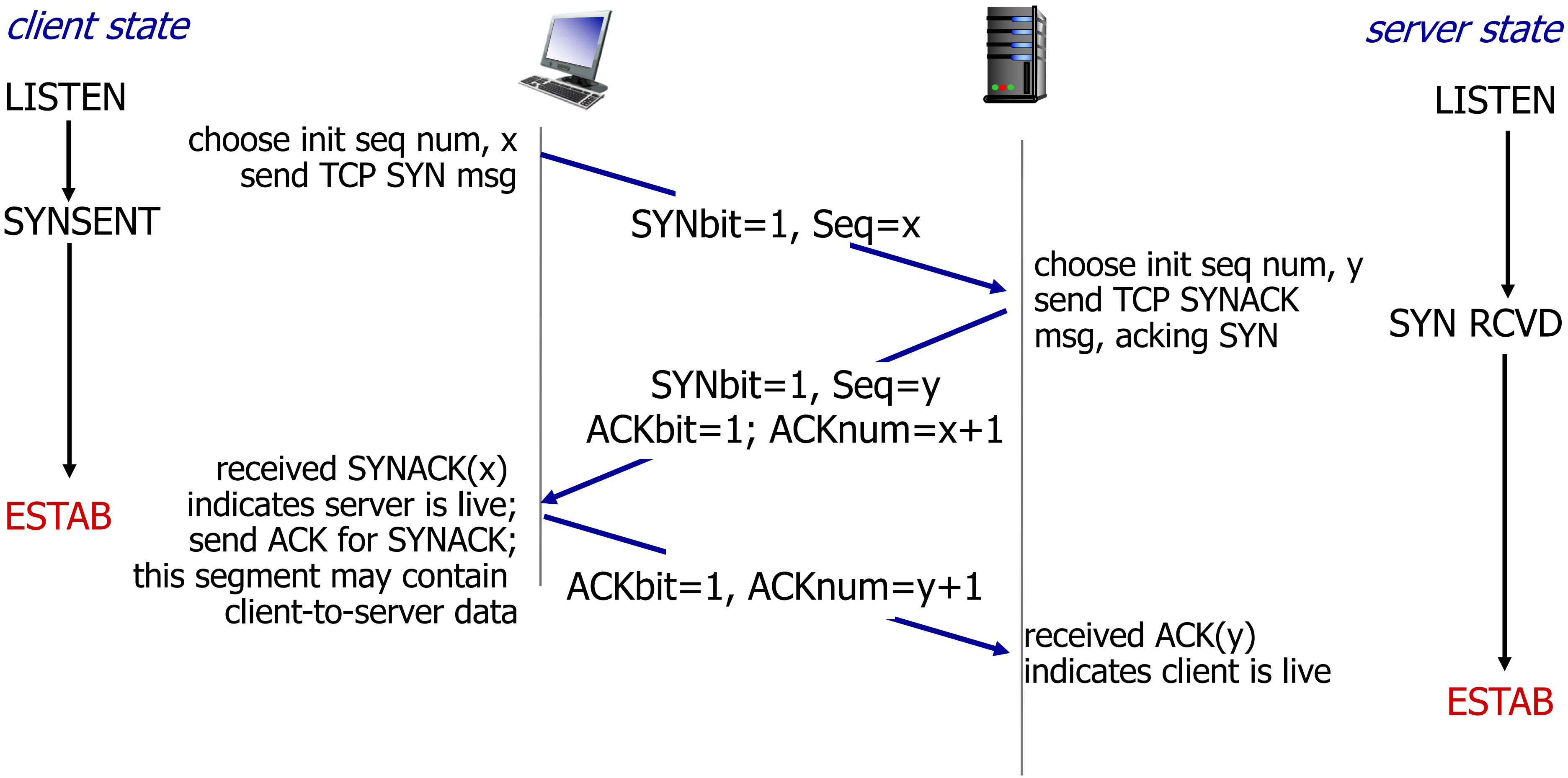
Connection Management

before exchanging data, sender/receiver "handshake":

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



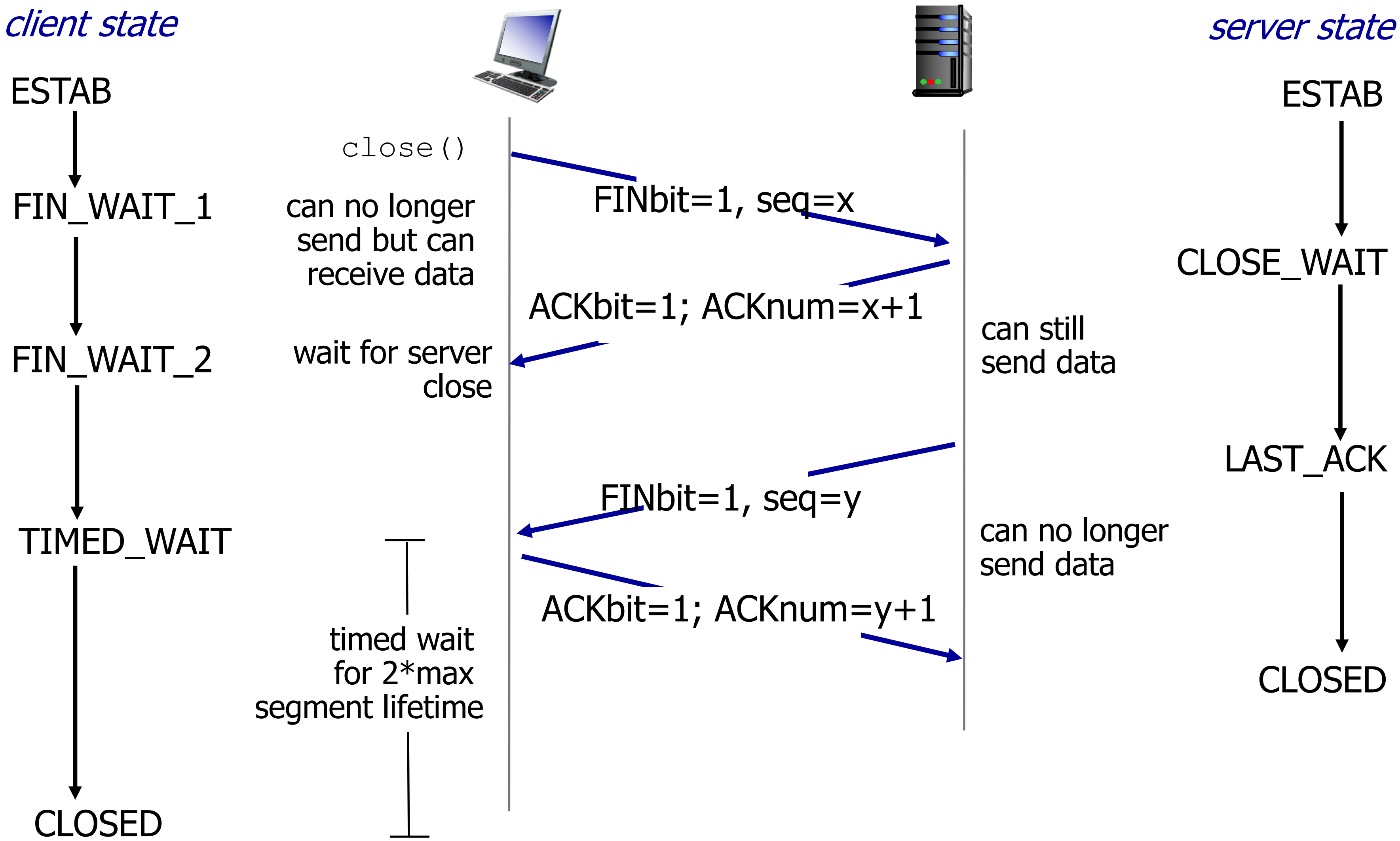
TCP 3-way handshake



TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

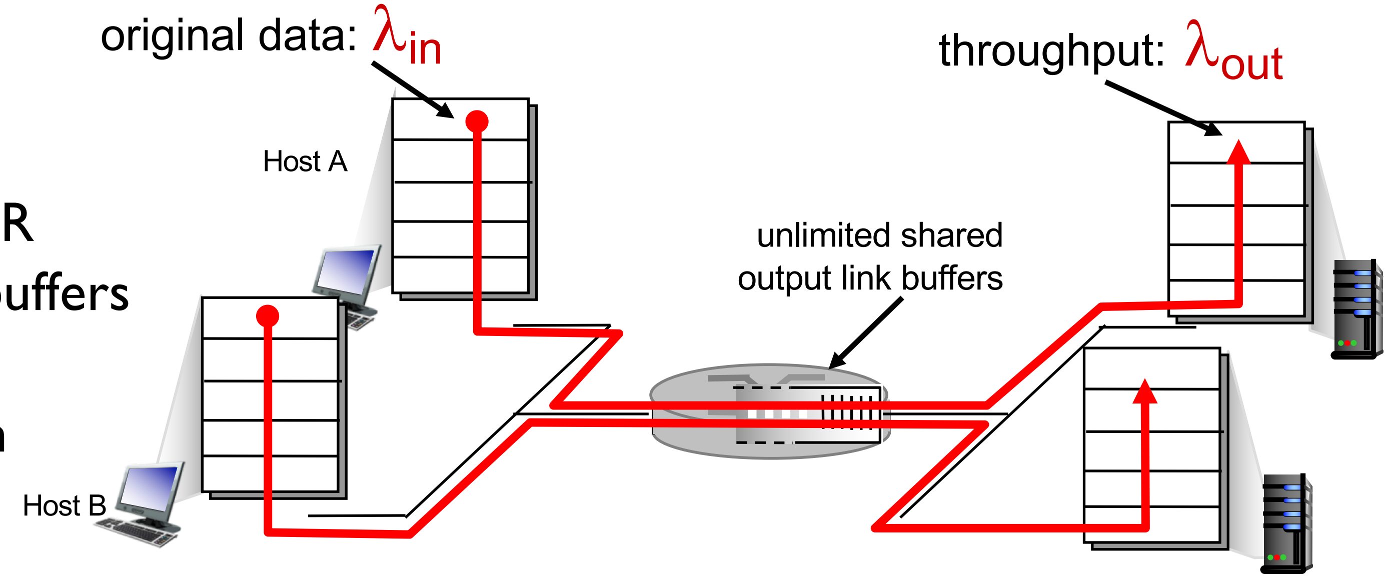
Principles of congestion control

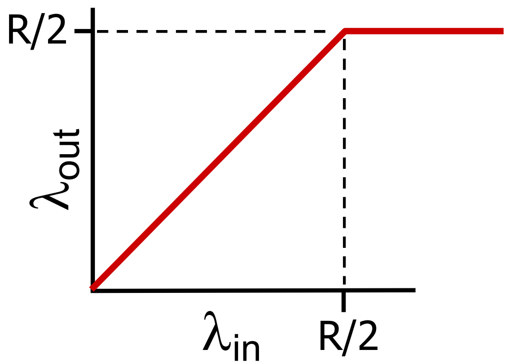
congestion:

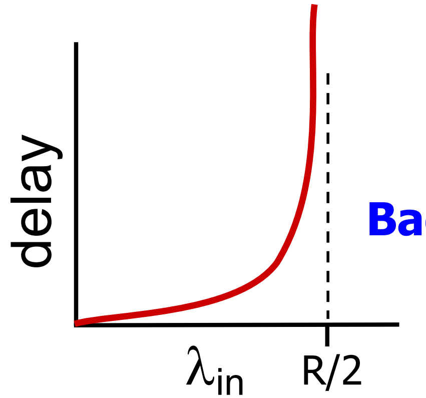
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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







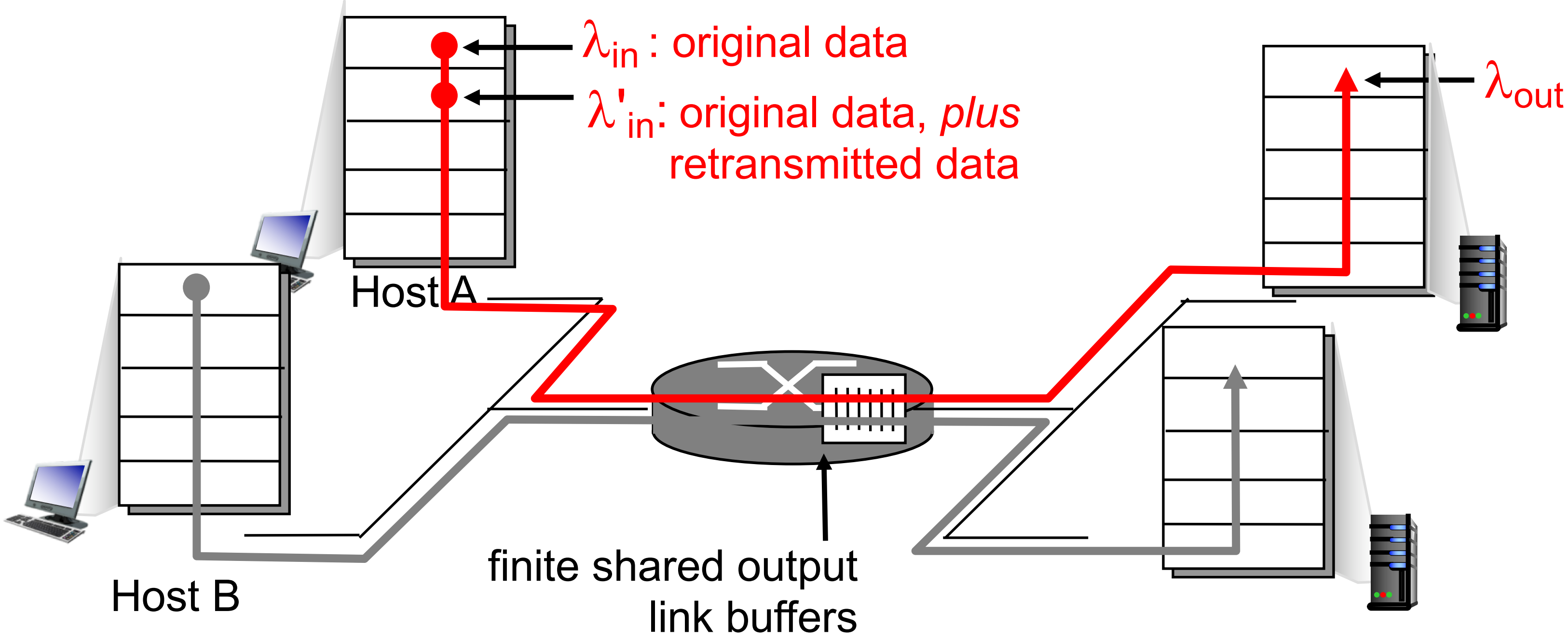
Bad - 1. long delay

- maximum per-connection throughput: R/2

- large delays as arrival rate, λ_{in}, approaches capacity

Causes/costs of congestion: scenario 2

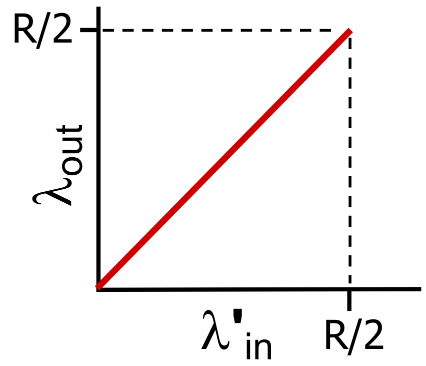
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output
    - $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions*
    - $\lambda'_{in} \geq \lambda_{in}$

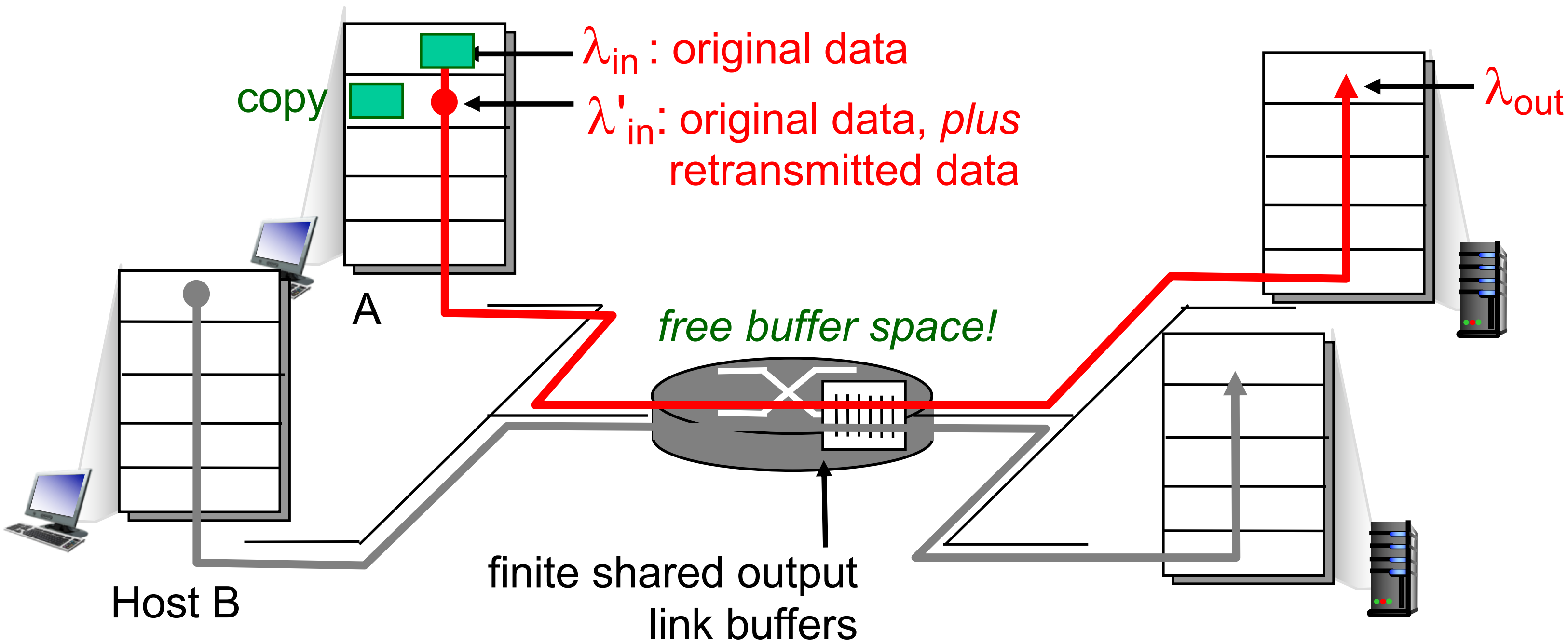


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available

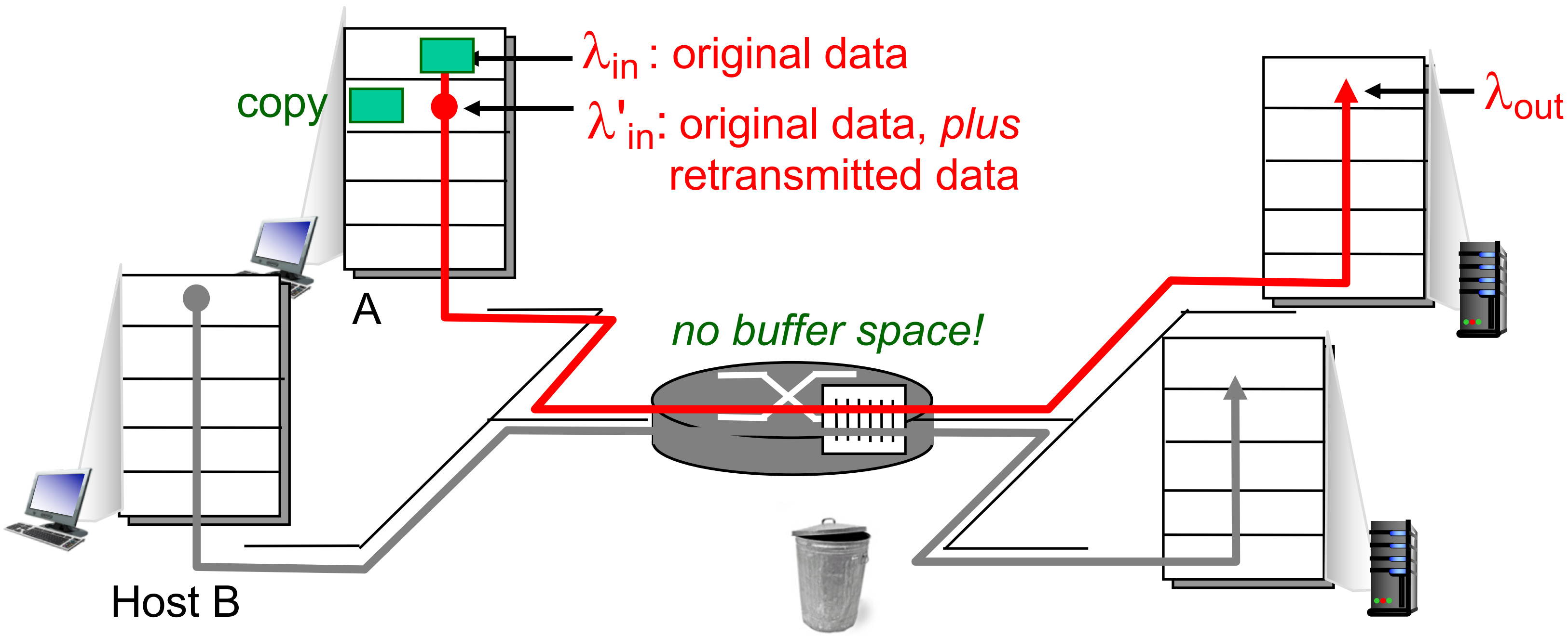




Causes/costs of congestion: scenario 2

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

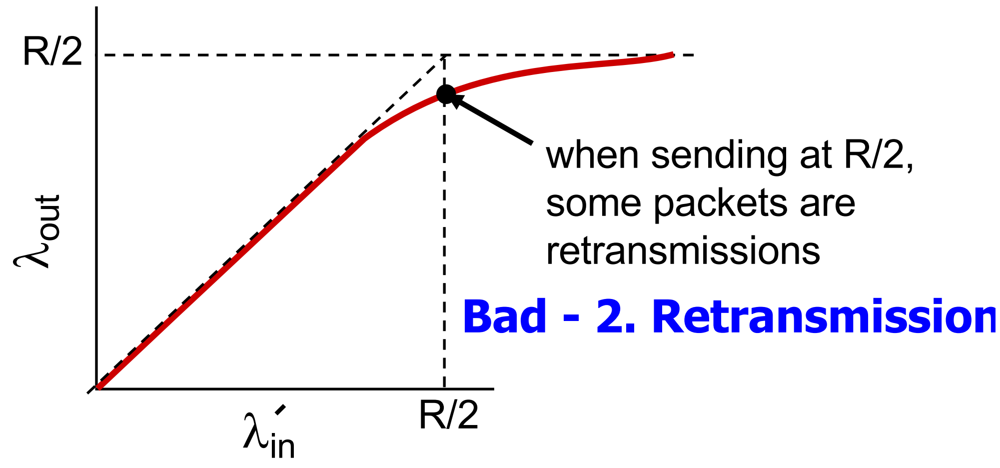
- sender only resends if packet *known* to be lost

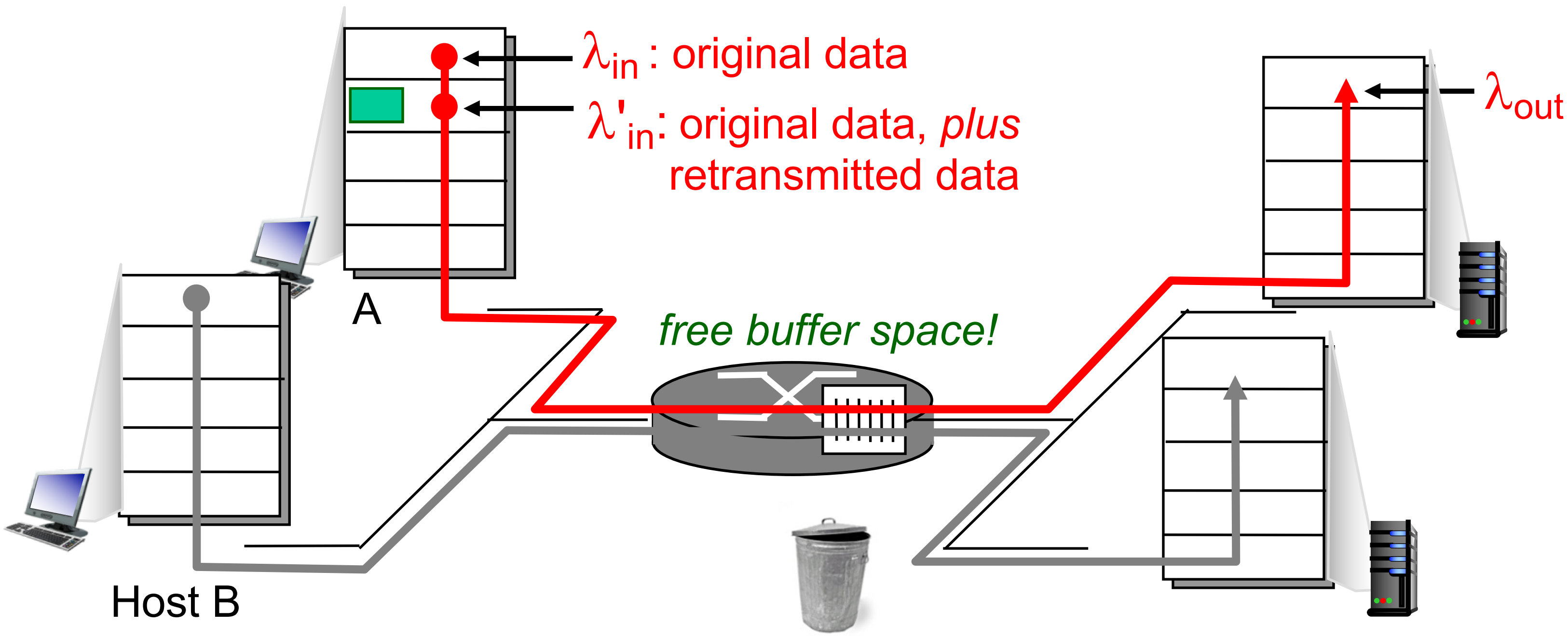


Causes/costs of congestion: scenario 2

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

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

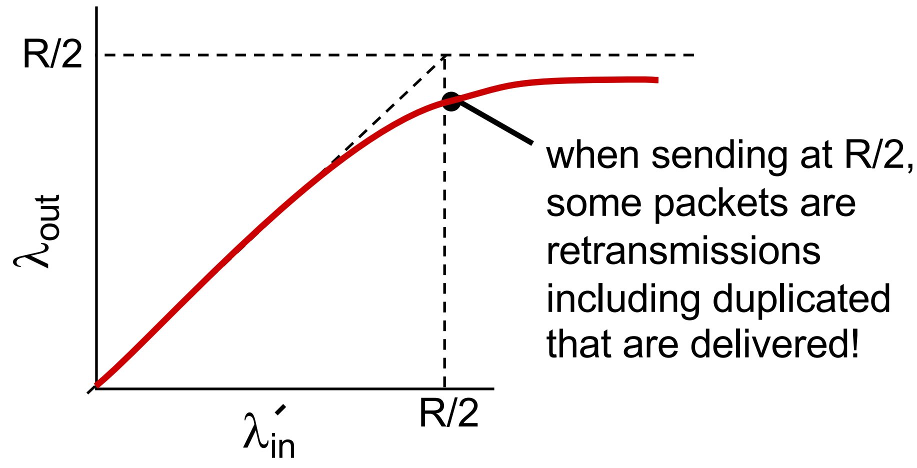


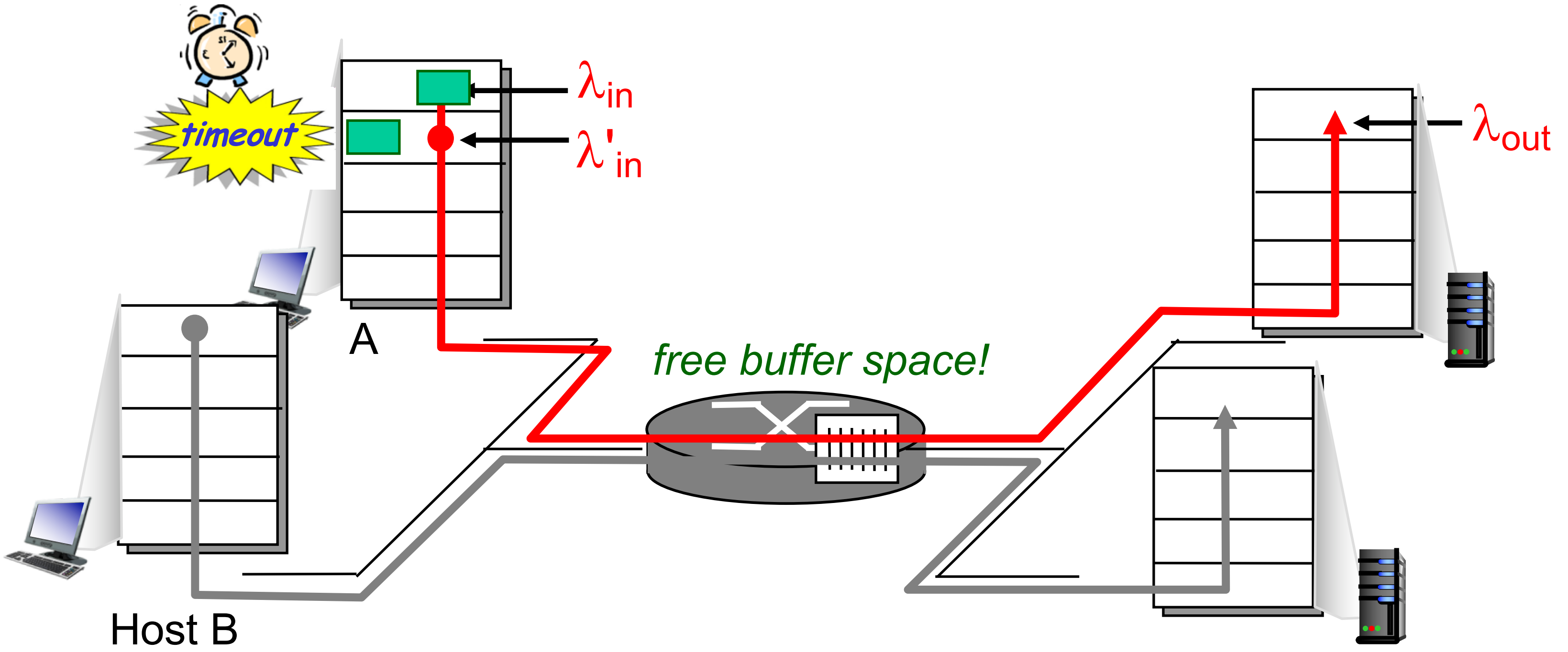


Causes/costs of congestion: scenario 2

Realistic: duplicates

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

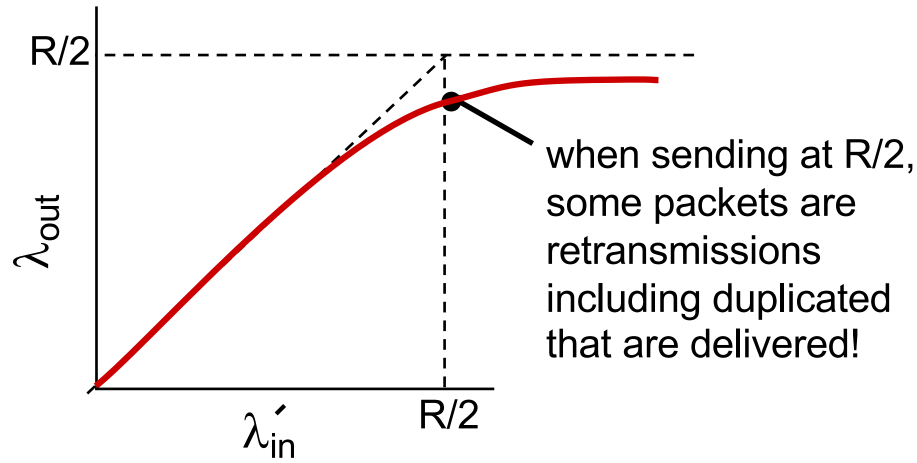




Causes/costs of congestion: scenario 2

Realistic: duplicates

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



"costs" of congestion:

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

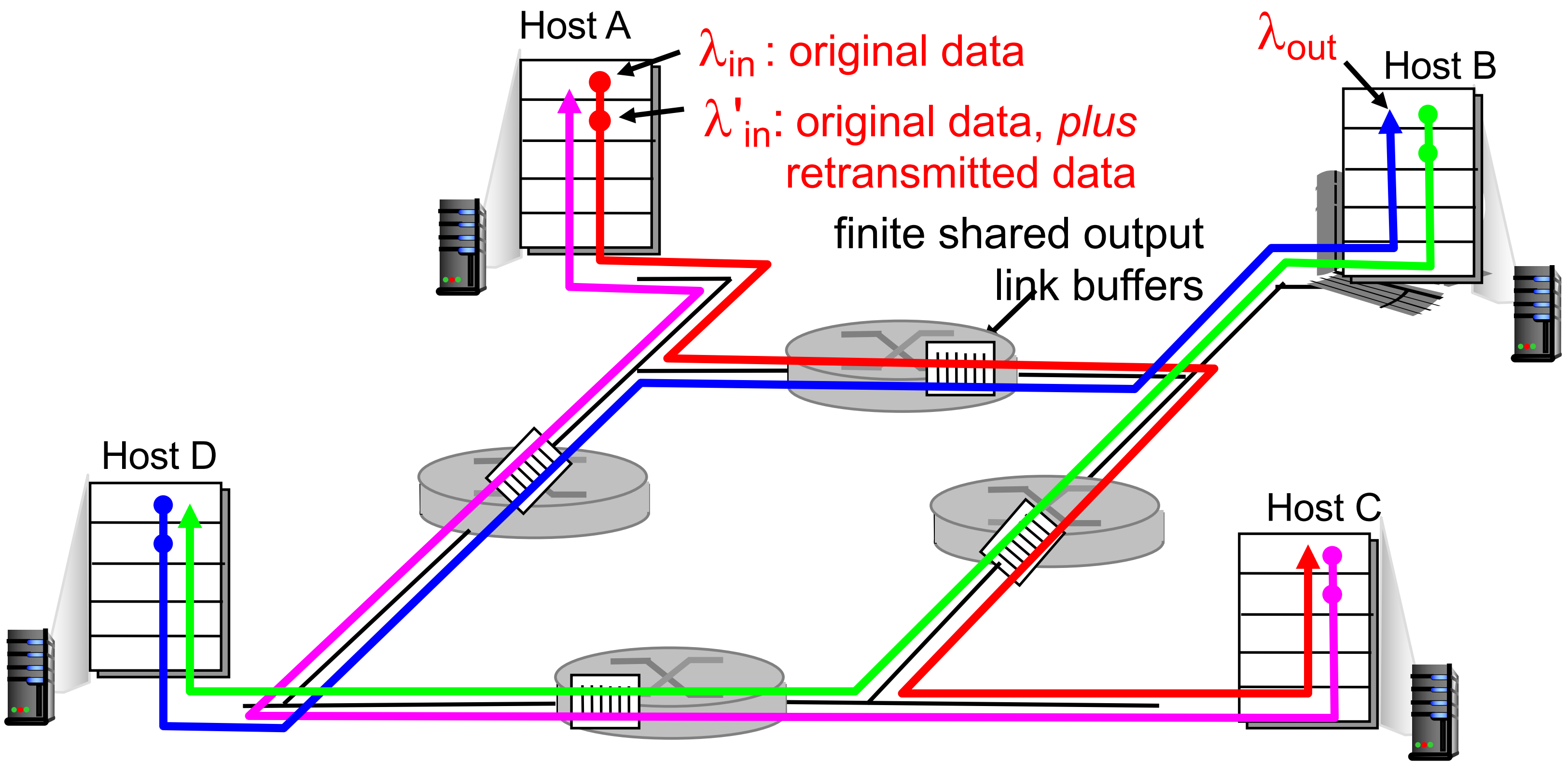
Bad - 3. Unnecessary retransmission

Causes/costs of congestion: scenario 3

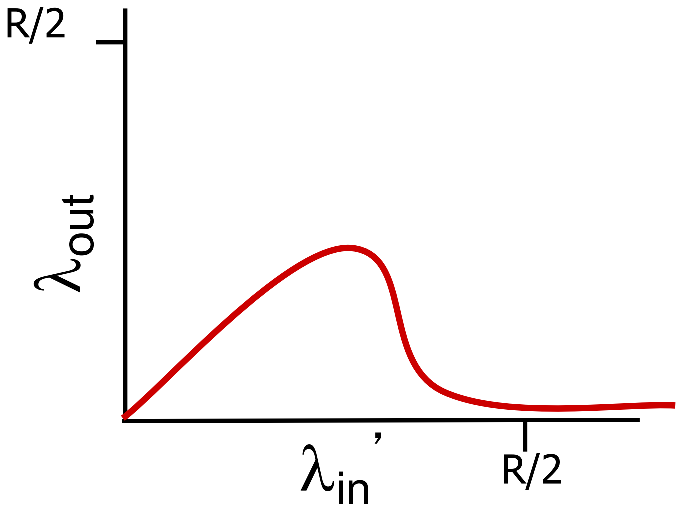
- four senders
- multihop paths
- timeout/retransmit

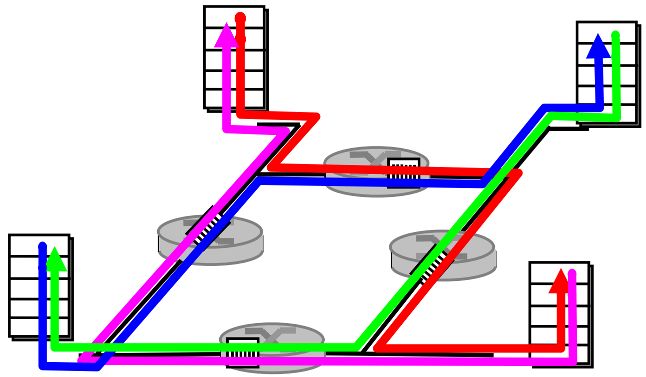
Q: what happens as λ_{in} and λ_{in}' increase ?

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



Causes/costs of congestion: scenario 3





another "cost" of congestion:

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

Bad - 4. Waste of upstream bandwidth resource

Message: Congestion is bad

But what can we do about it?

Quiz Time!

Approaches: congestion control

End-to-end:

- end-systems infer congestion
  - from observed loss, delay
  - no explicit feedback from network (routers)
- end-systems infer available rate/bandwidth
  - by trying and failing
- approach taken by TCP

Network-assisted:

- routers provide feedback to end systems
  - from observed queue size, free buffer space
  - single bit in pkt indicating congestion (e.g. TCP/IP ECN)
- routers tells explicit rate
  - sender sends at the rate

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

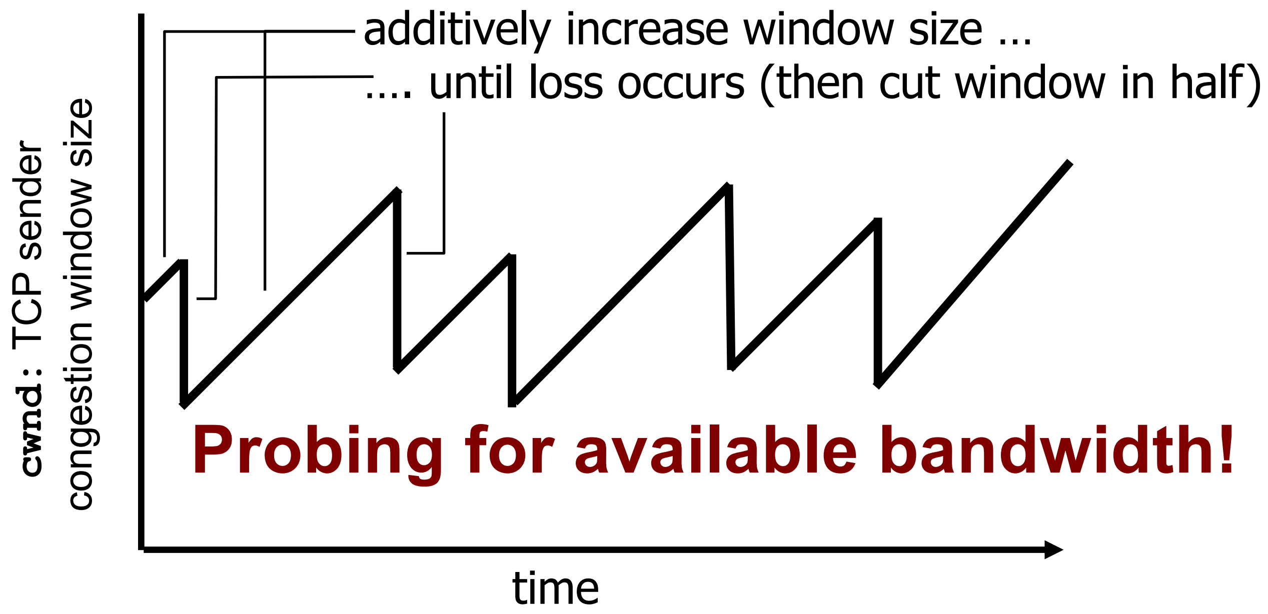
3.6 principles of congestion control

3.7 TCP congestion control

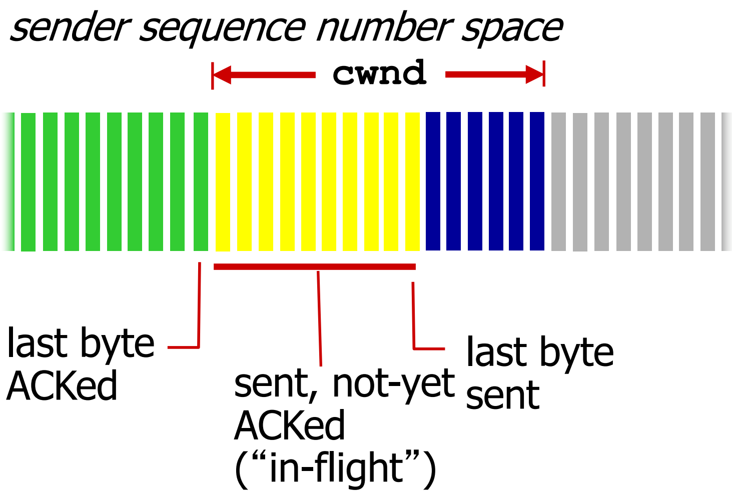
TCP Congestion Avoidance: AIMD

- *approach:* sender increases transmission rate (window size), until loss occurs
  - *Additive Increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
  - *Multiplicative Decrease:* cut **`cwnd`** in half after loss

Saw tooth behavior



TCP cwnd



- sender limits transmission:

```
LastByteSent-
  LastByteAcked   ≤  cwnd
```

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

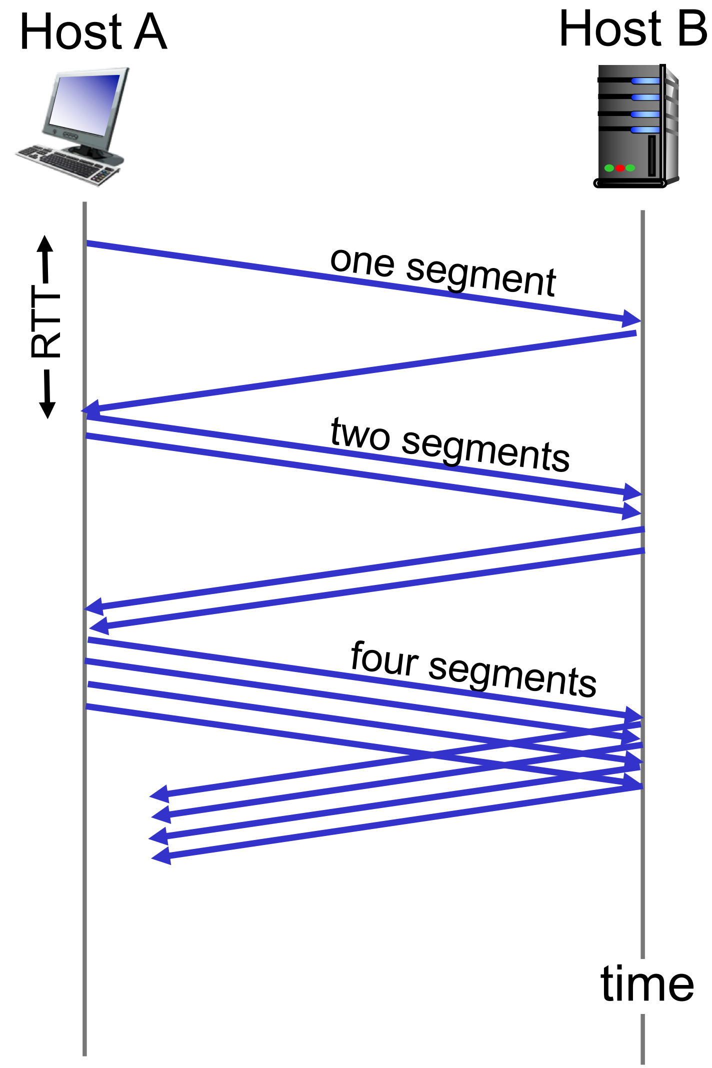
TCP sending rate:

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

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

TCP Initialization: Slow Start

- when connection begins, increase **cwnd** exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast

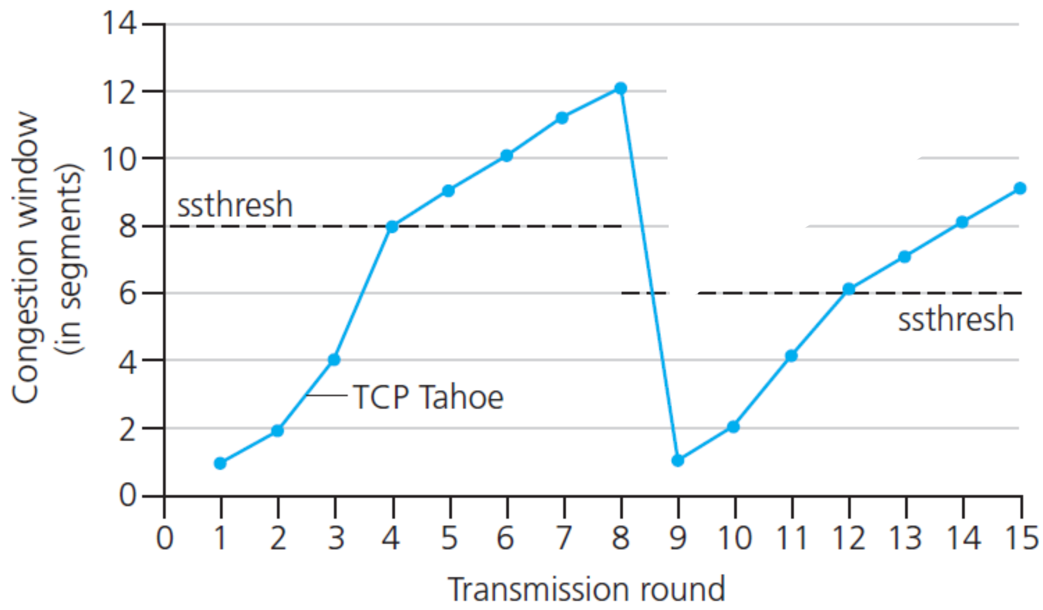


TCP: detecting, reacting to loss

- loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to a threshold (ssthresh), 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 SS to CA

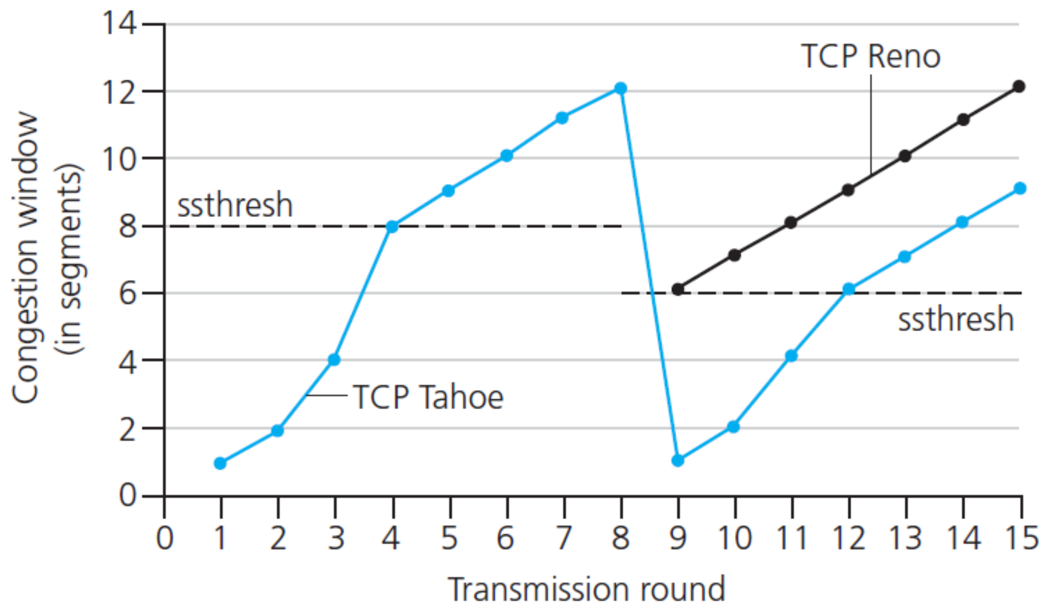


* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP: switching from SS to CA

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to **ssthresh**.

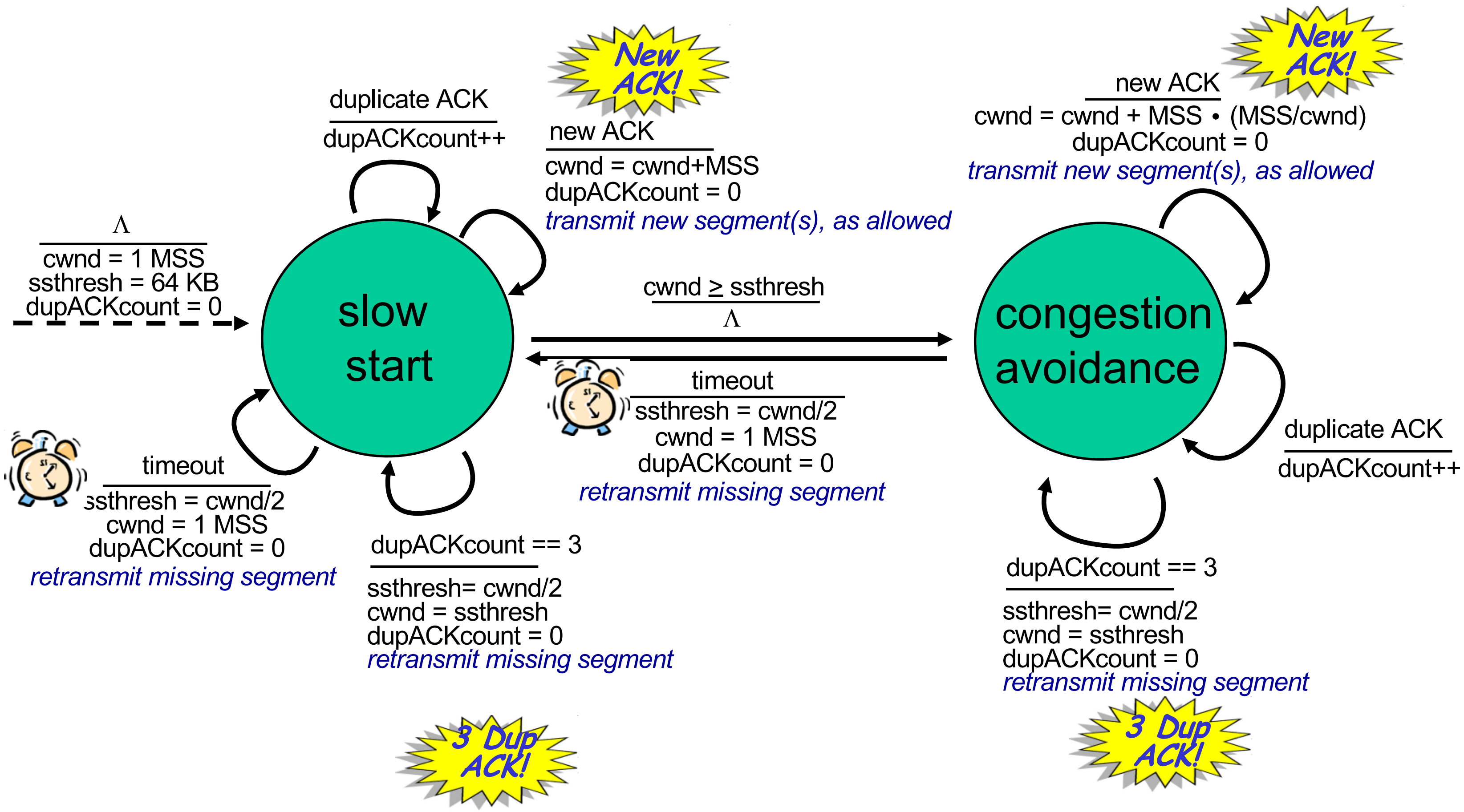


Implementation:

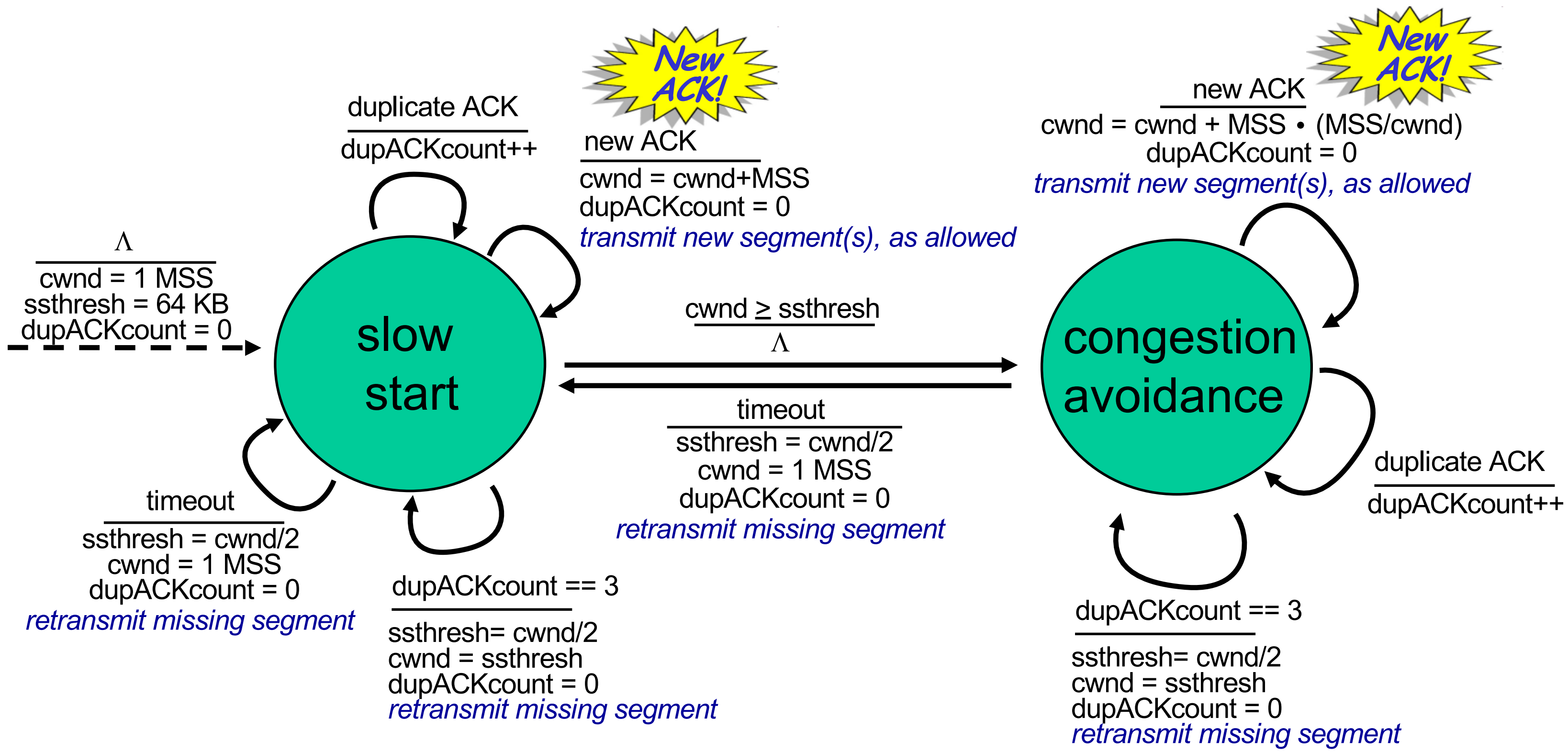
- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

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

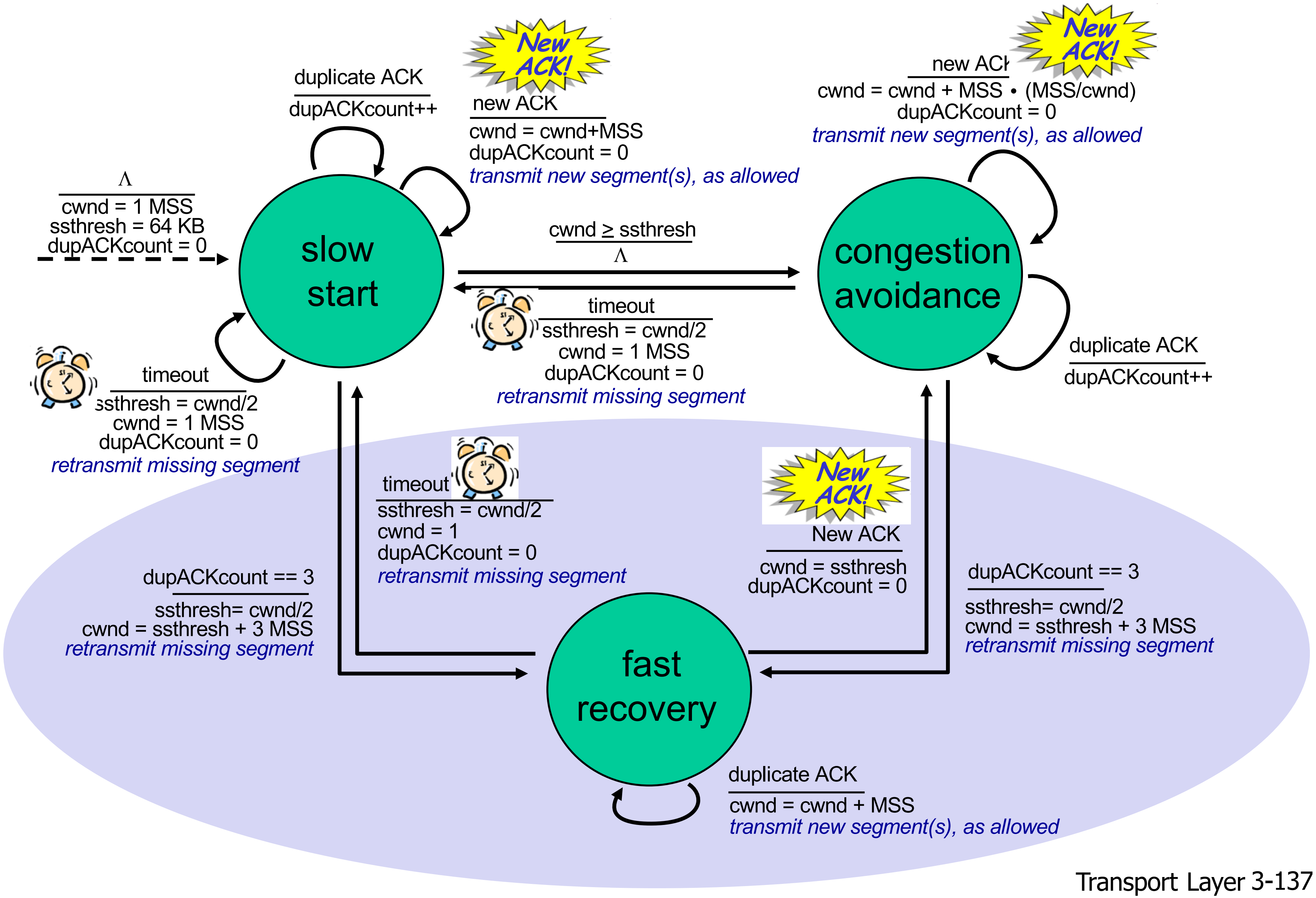
Summary: TCP Congestion Control



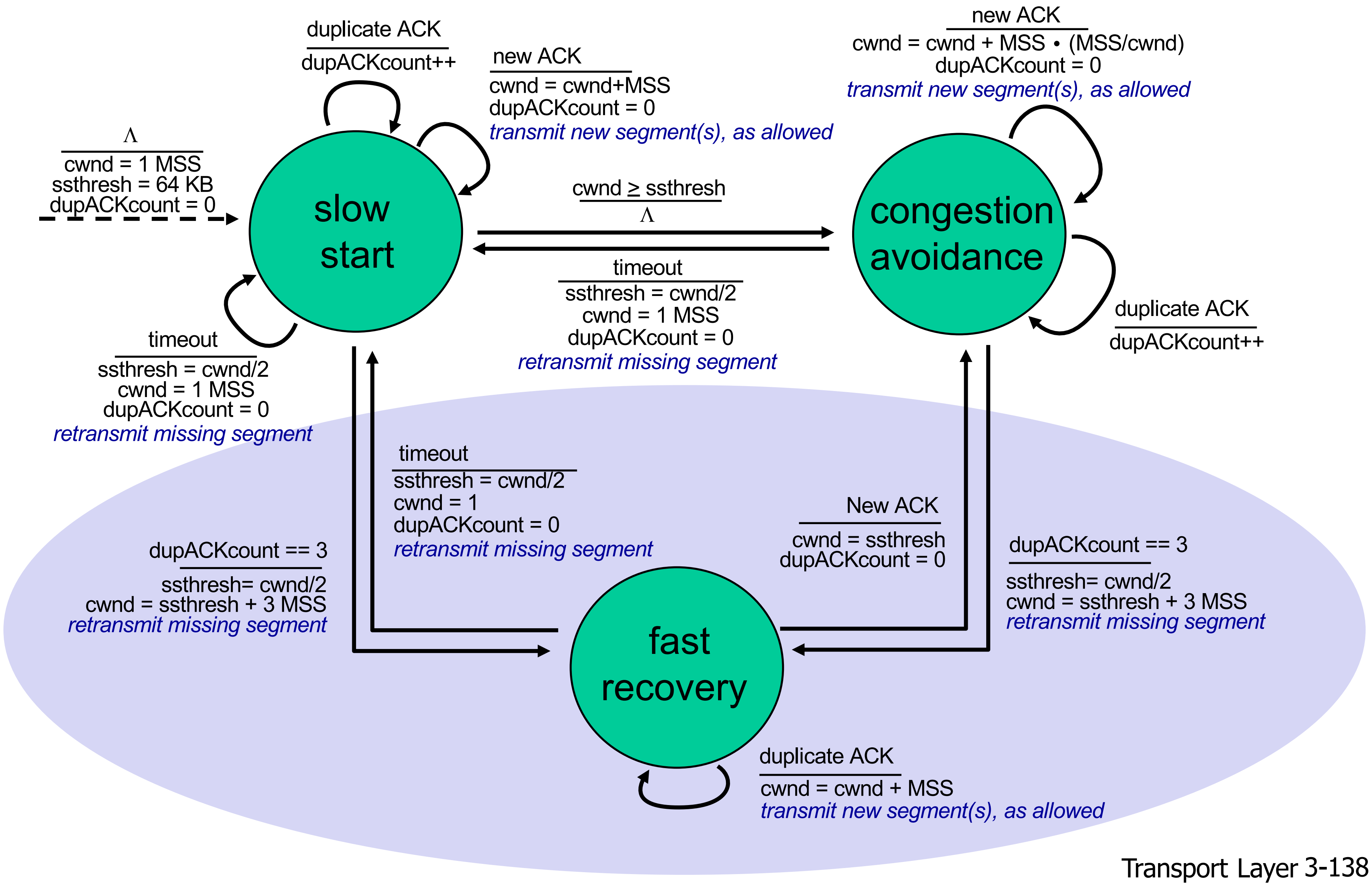
Summary: TCP Congestion Control



Summary: Full TCP Congestion Control



Summary: Full TCP Congestion Control



Double Quiz Time!

TCP Futures: TCP over "long, fat pipes"

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

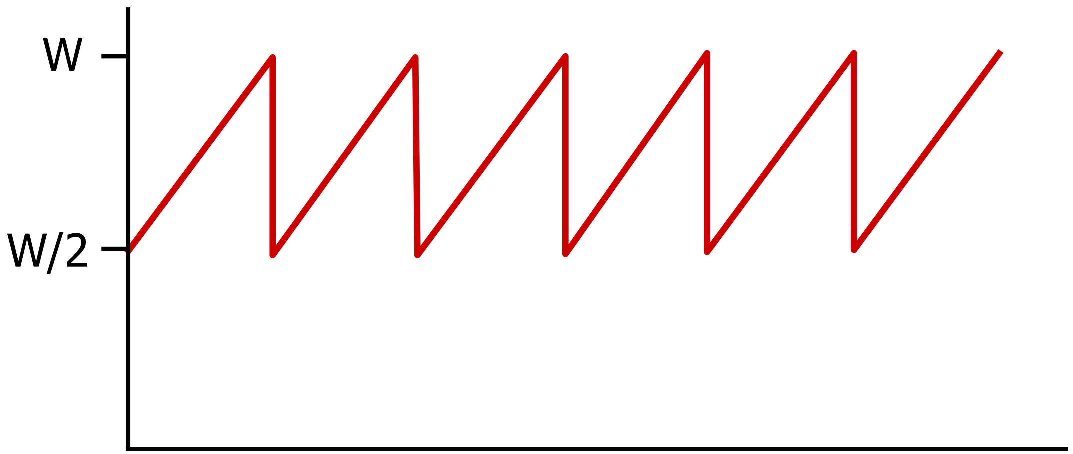
$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- requires 83,333 in-flight segments

TCP throughput with loss

- 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"

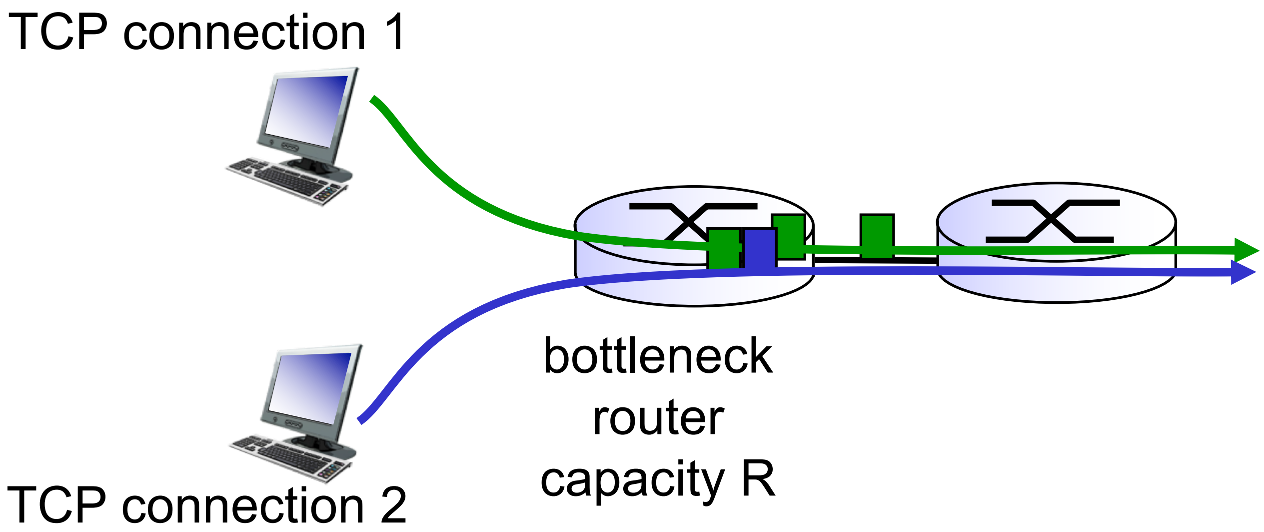
- throughput in terms of segment loss probability, L [Mathis 1997]:

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

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
  - ➜ to achieve 10 Gbps throughput, need a loss rate of L = $2 \cdot 10^{-10}$ *– a very small loss rate!*
- Needing new versions of TCP for high-speed

TCP Fairness

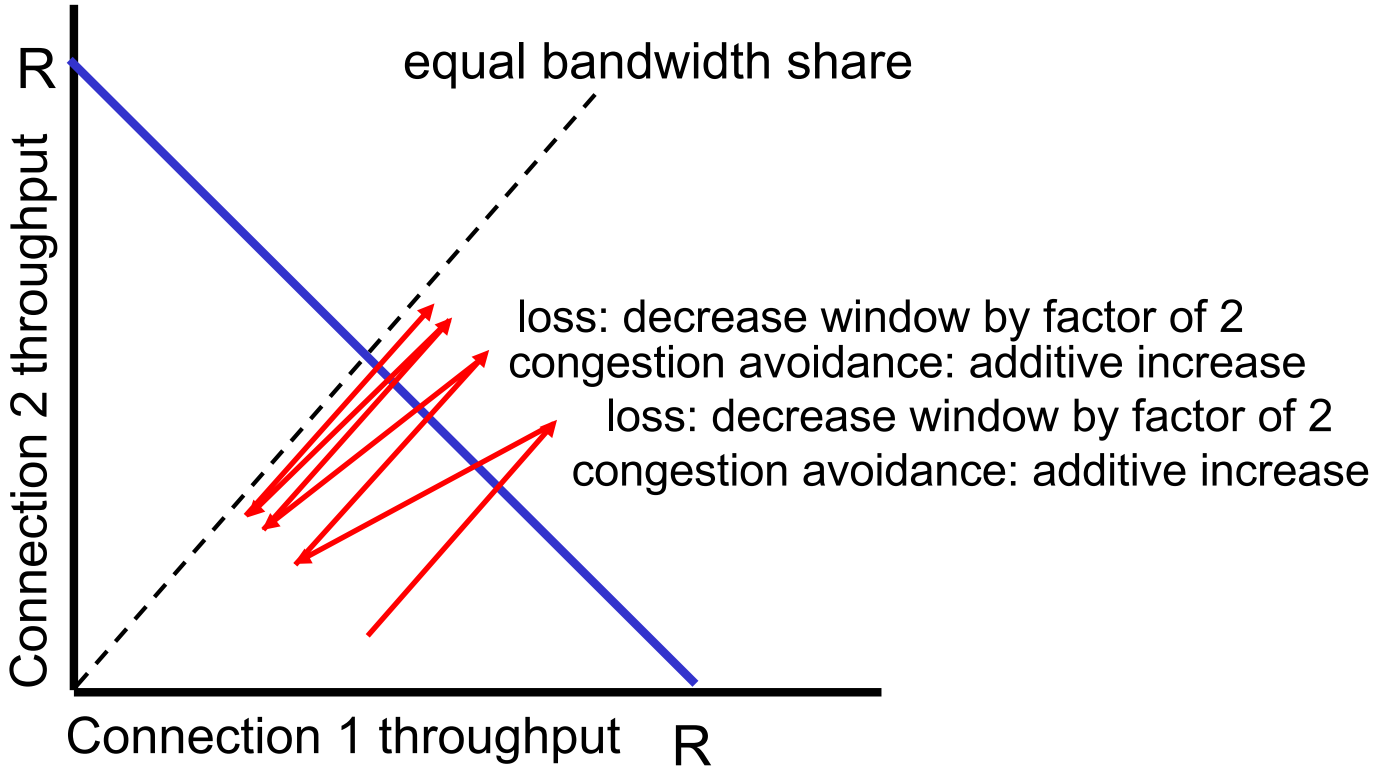
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions:

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



Quiz Time!

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 9 TCPs, gets R/2

Approaches: congestion control

End-to-end:

- end-systems infer congestion
  - from observed loss, delay
  - no explicit feedback from network (routers)
- end-systems infer available rate/bandwidth
  - by trying and failing
- approach taken by TCP

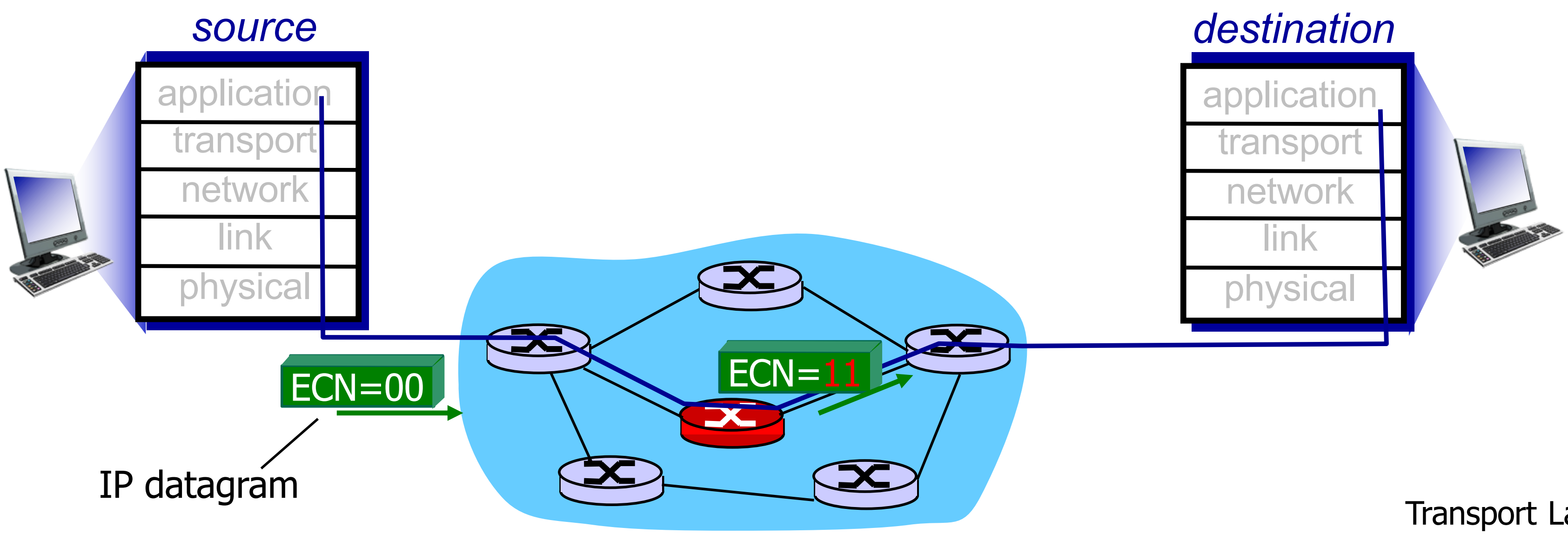
Network-assisted:

- routers provide feedback to end systems
  - from observed queue size, free buffer space
  - single bit in pkt indicating congestion (e.g. TCP/IP ECN)
- routers tells explicit rate
  - sender sends at the rate

Explicit Congestion Notification (ECN)

network-assisted congestion control:

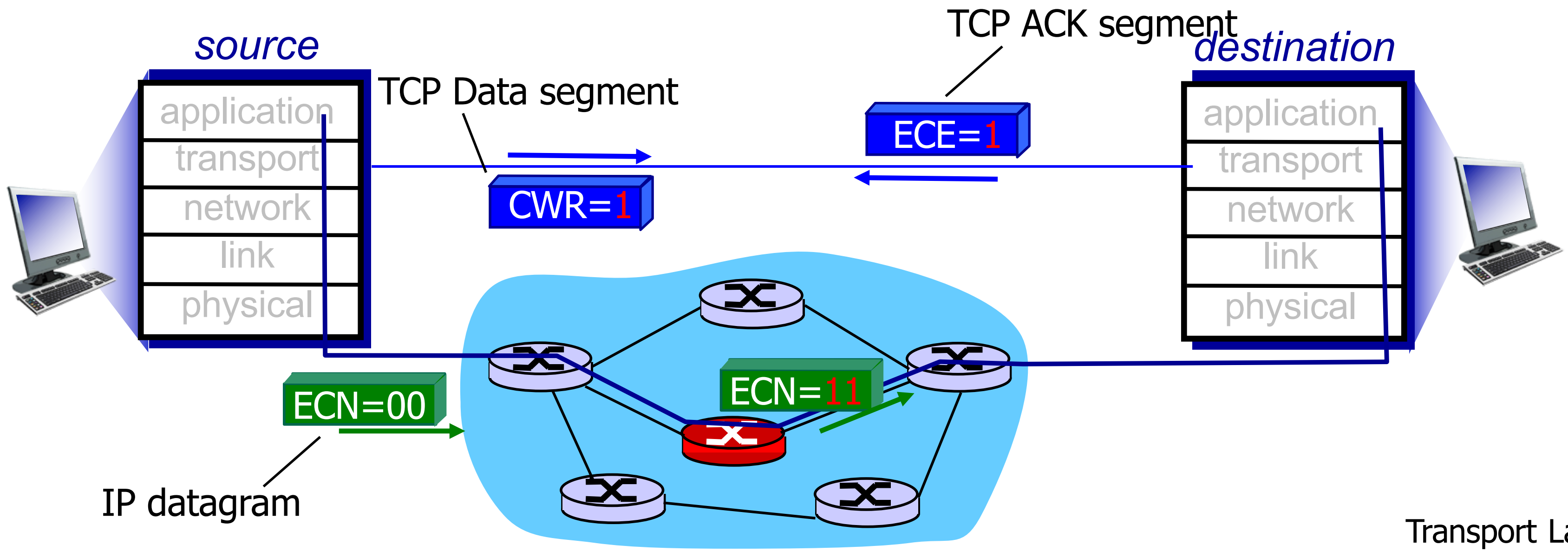
- two bits in IP header (ToS field) marked *by network router* to indicate congestion
- congestion indication carried to receiving host



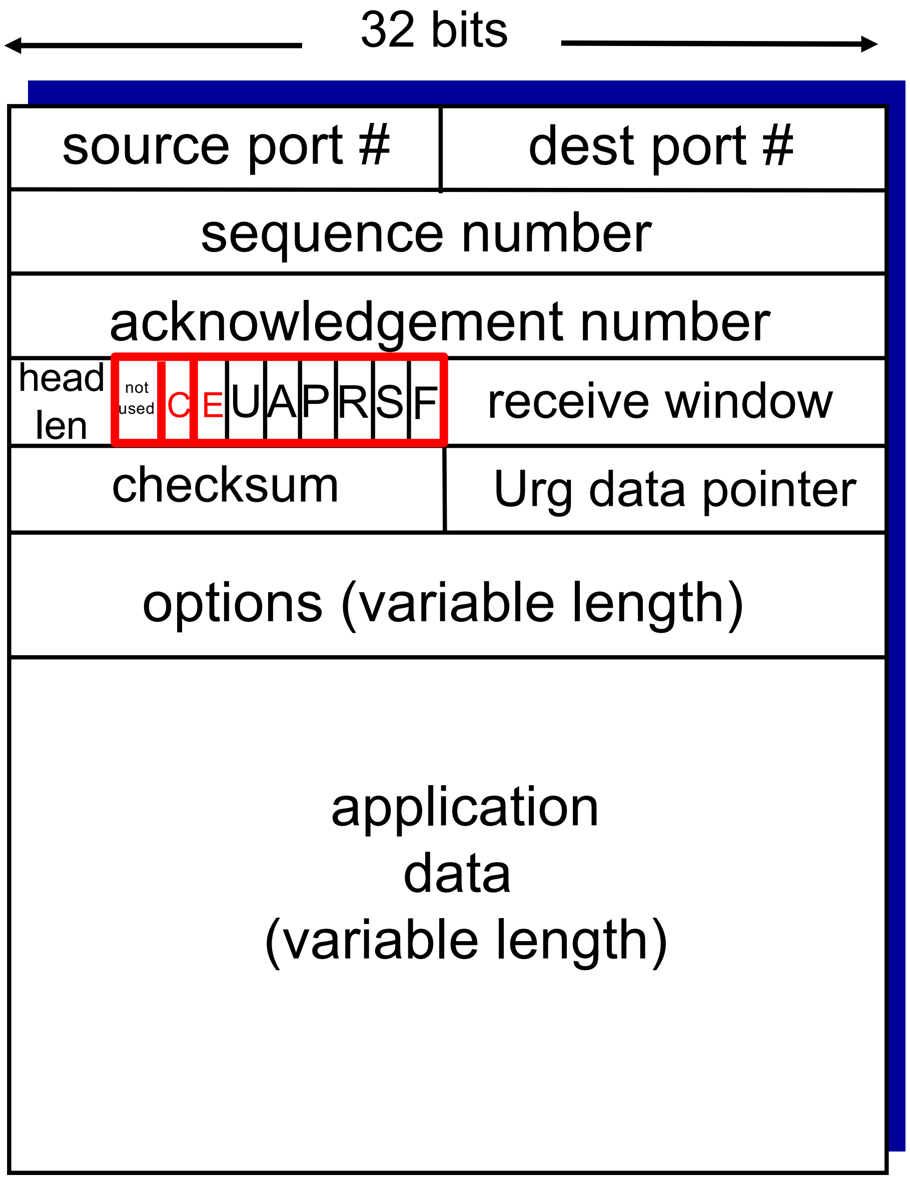
Explicit Congestion Notification (ECN)

network-assisted congestion control:

- receiver sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion
- sender sets CWR bit on sender-to receiver Data segment to confirm **`cwnd`** being reduced



TCP segment structure



Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network layer chapters:
  - data plane
  - control plane