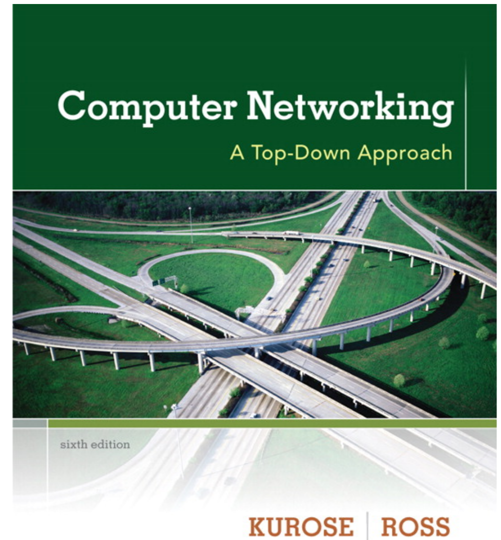
3. Transport Layer



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

3. Transport Layer: Goals

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

3. Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

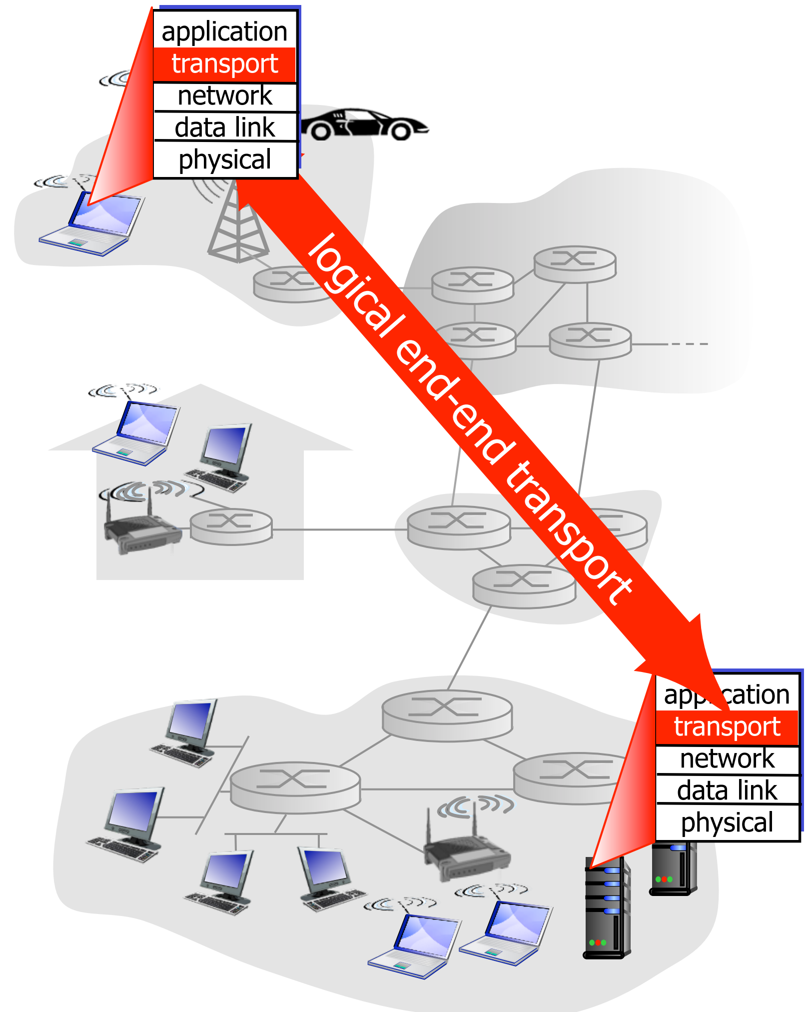
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

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
  - recv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

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

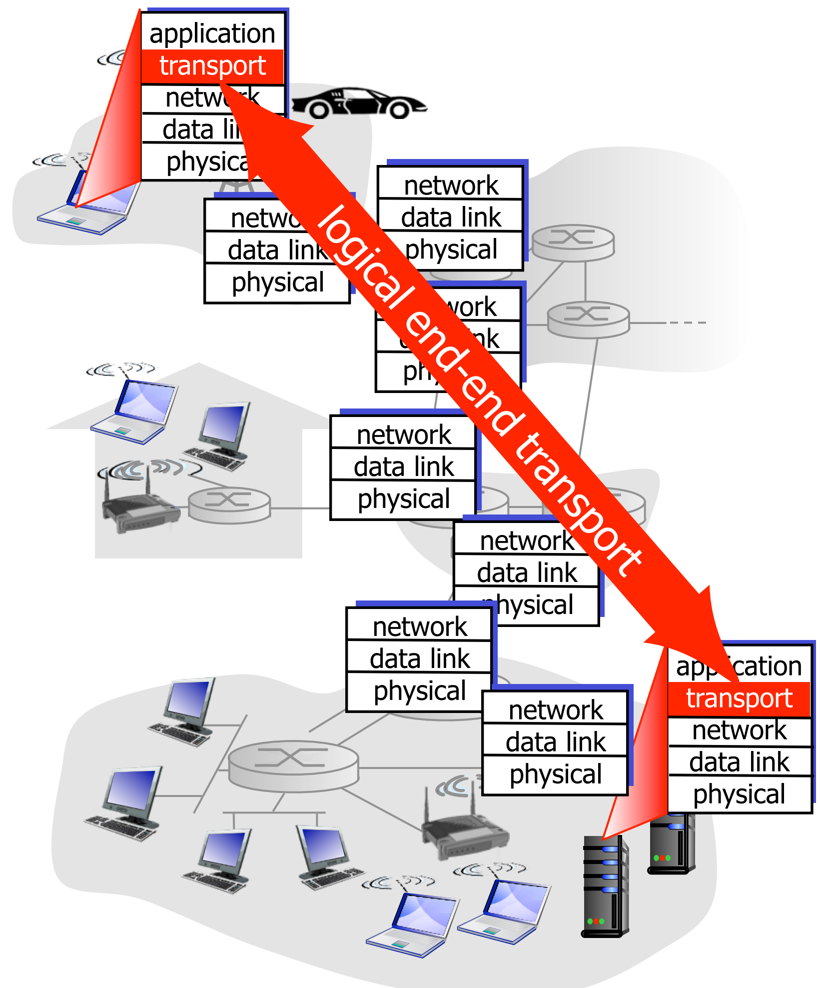
household analogy:

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

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

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



3. Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

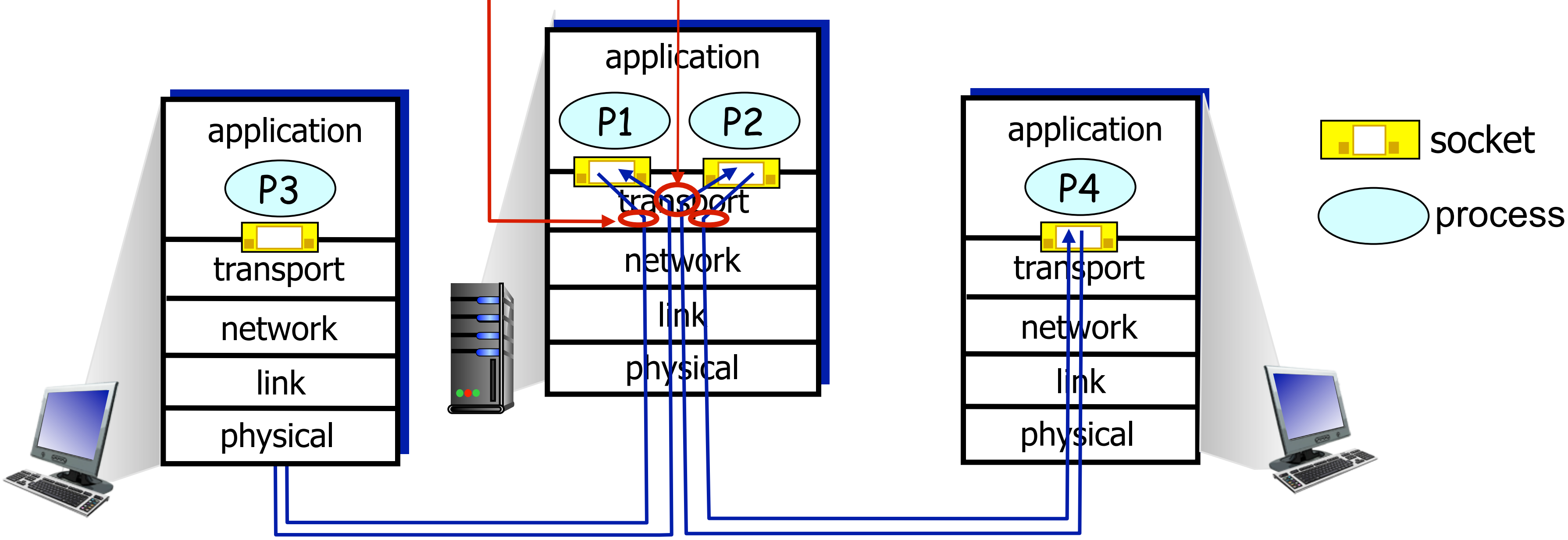
3.6 principles of congestion control

3.7 TCP congestion control

Multiplexing/demultiplexing

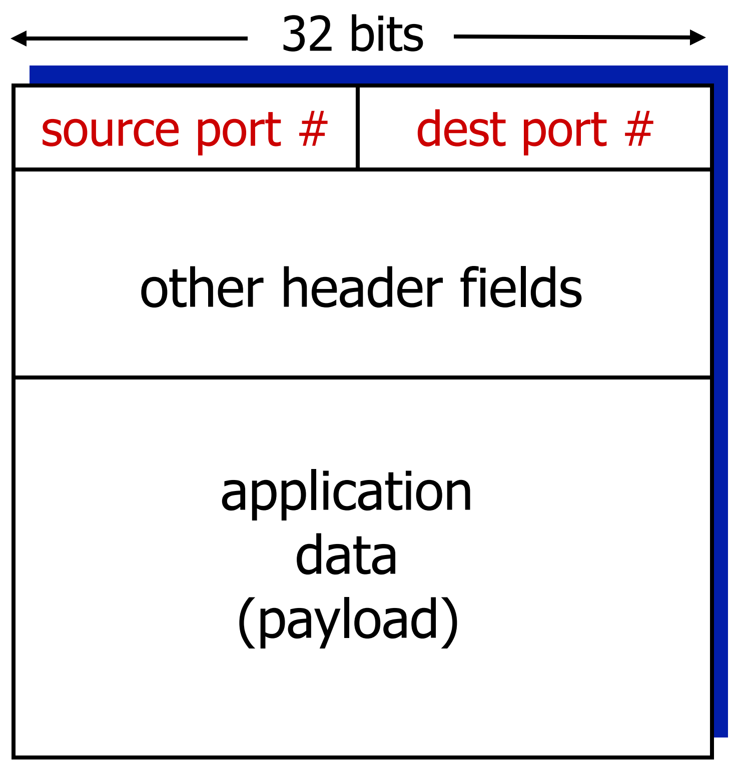
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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



How demultiplexing works

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



TCP/UDP segment format

Connectionless demultiplexing

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

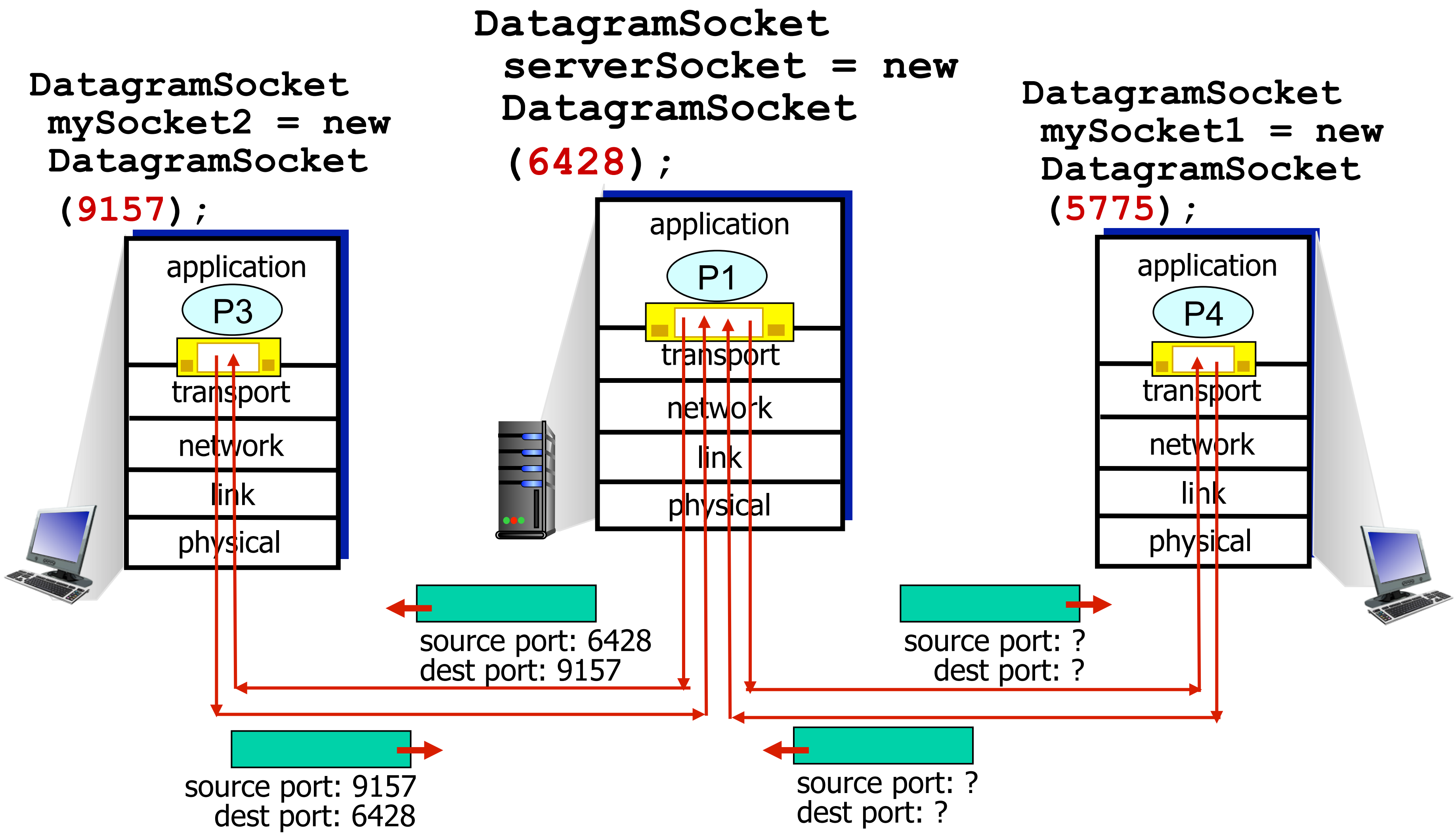
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

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

- when host receives UDP segment:
  - checks destination IP and port # in segment
  - directs UDP segment to socket bound to that (IP,port)

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

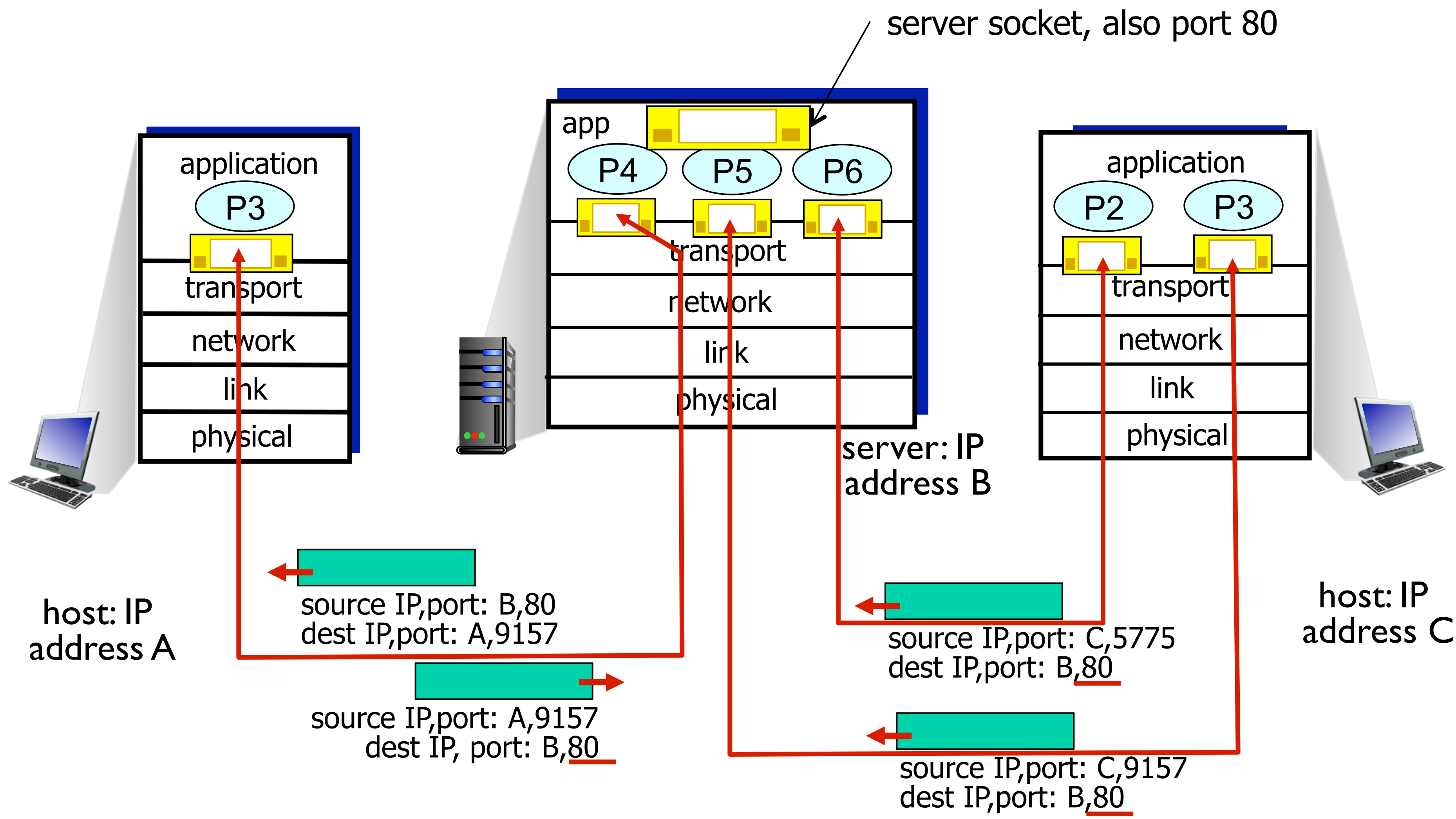
Connectionless demux: example



Connection-oriented demux

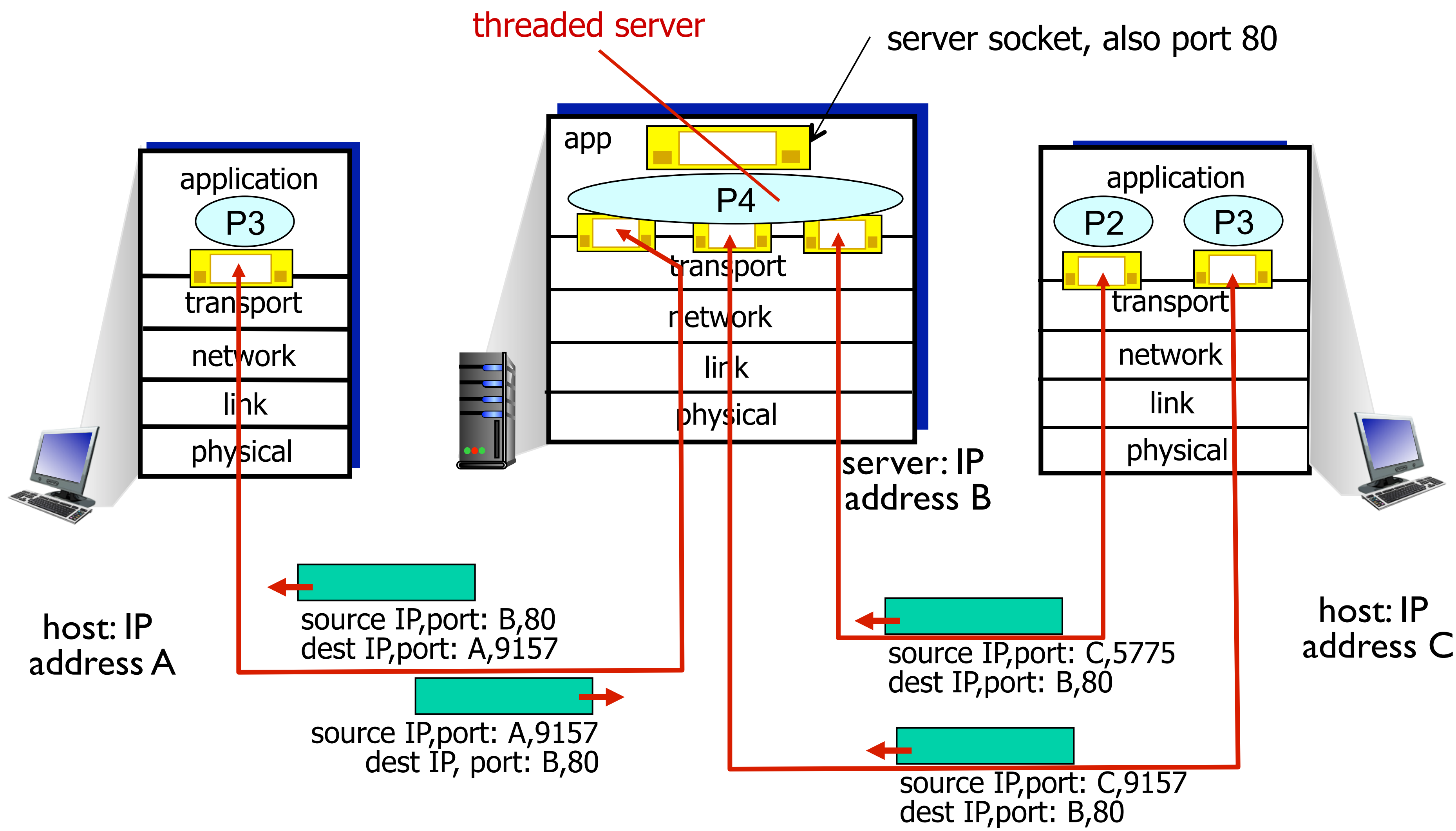
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to right socket
- server host has many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different socket each client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



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

Connection-oriented demux: example



3. Transport Layer: 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

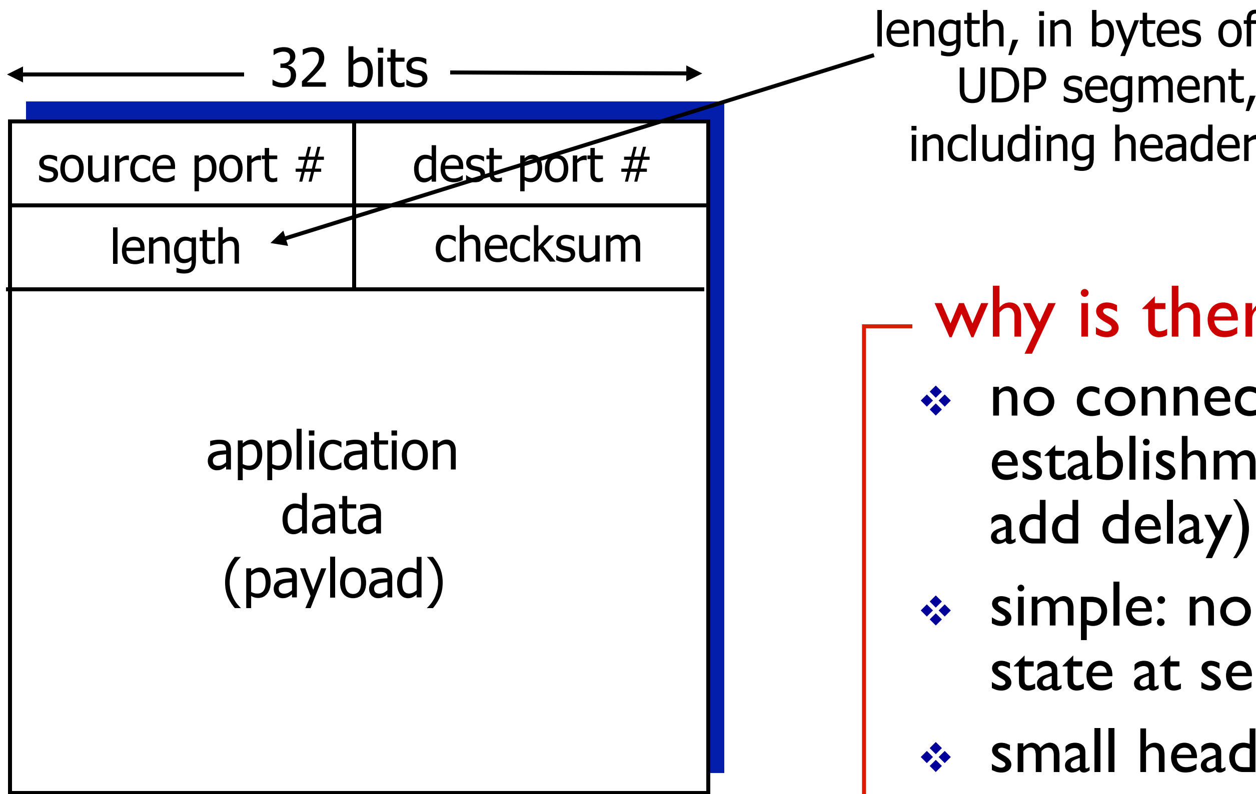
3.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

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

- UDP uses:
  - 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" (flipped bits) in segments

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

Q1: Sockets and multiplexing

- TCP uses more information in packet headers in order to demultiplex packets compared to UDP.
  A. True
  B. False

Q2: Sockets UDP

- Suppose we use UDP instead of TCP under HTTP for designing a web server where all requests and responses fit in a single packet. Suppose a 100 clients are simultaneously communicating with this web server. How many sockets are respectively at the server and at each client?
  A. 1,1
  B. 2,1
  C. 200,2
  D. 100,1
  E. 101, 1

Q3: Sockets TCP

- Suppose a 100 clients are simultaneously communicating with (a traditional HTTP/TCP) web server. How many sockets are respectively at the server and at each client?
  A. 1,1
  B. 2,1
  C. 200,2
  D. 100,1
  E. 101, 1

Q4: Sockets TCP

- Suppose a 100 clients are simultaneously communicating with (a traditional HTTP/TCP) web server. Do all of the sockets at the server have the same server-side port number?

  A. Yes

  B. No

Q5: UDP checksums

- Let's denote a UDP packet as (checksum, data) ignoring other fields for this question. Suppose a sender sends (0010, 1110) and the receiver receives (0011,1110). Which of the following is true of the receiver?
  A. Thinks the packet is corrupted and discards the packet.
  B. Thinks only the checksum is corrupted and delivers the correct data to the application.
  C. Can possibly conclude that nothing is wrong with the packet.
  D. A and C

3. Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

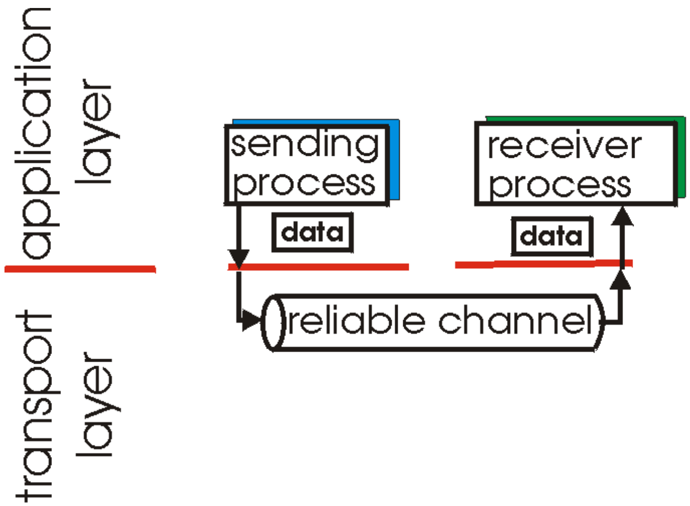
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

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!

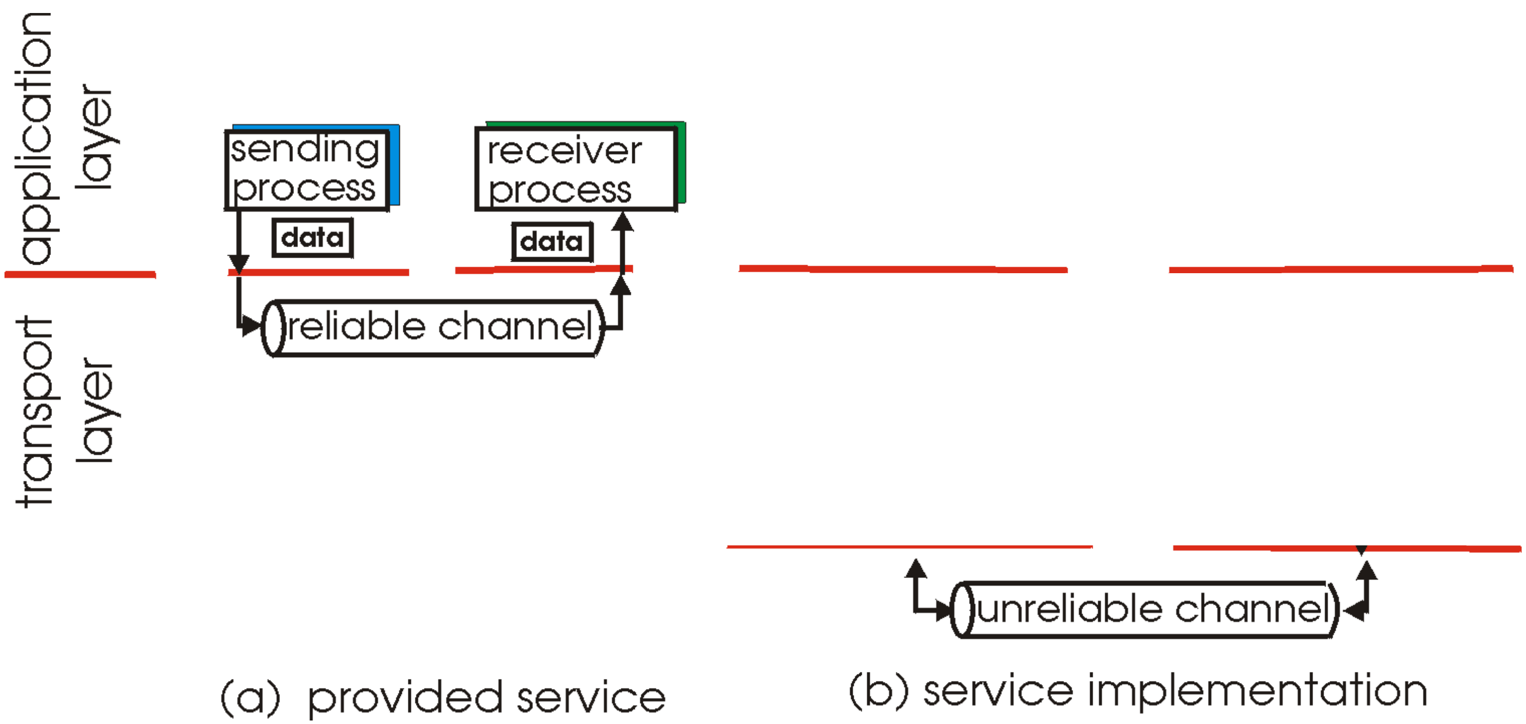


(a) provided service

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

Principles of reliable data transfer

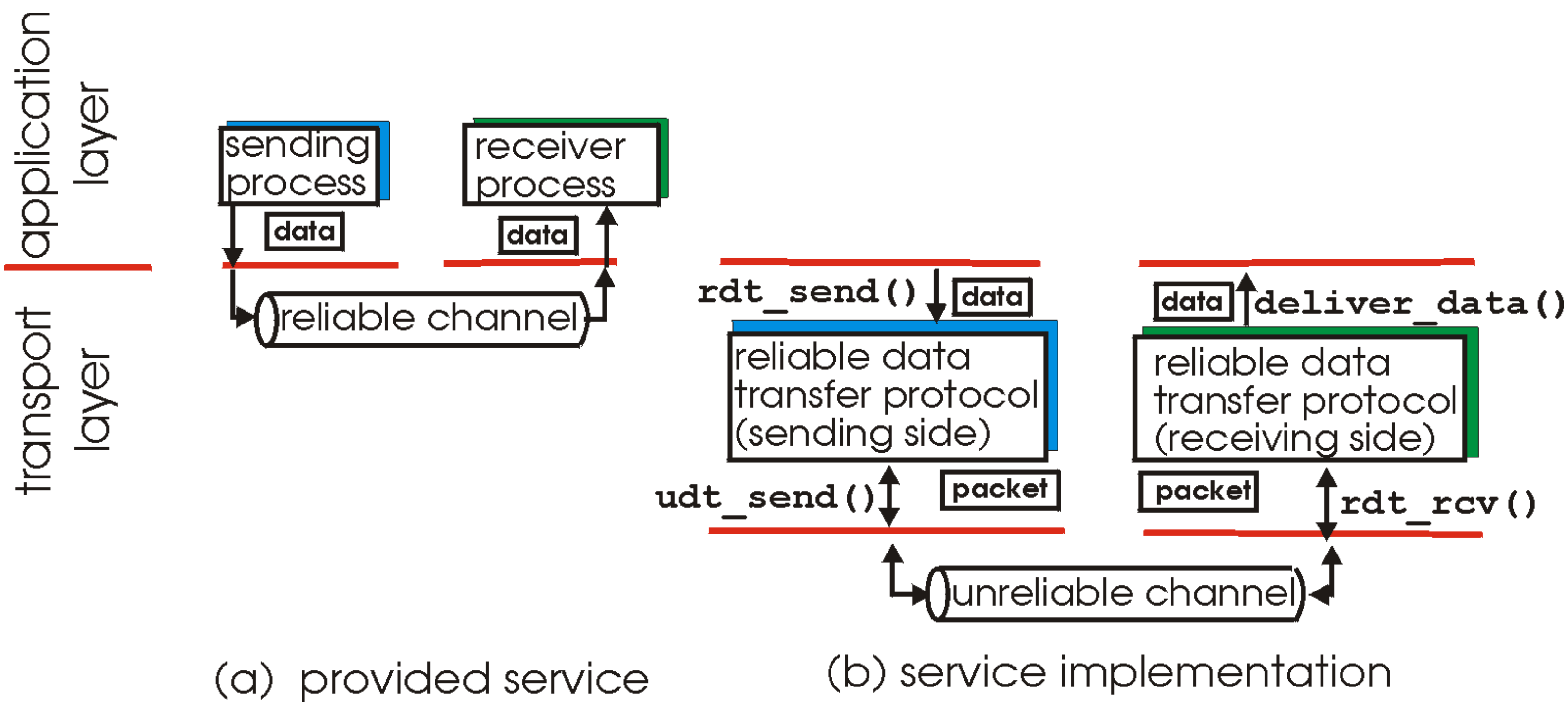
- important in application, transport, link layers
  - top-10 list of important networking topics!



- 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!

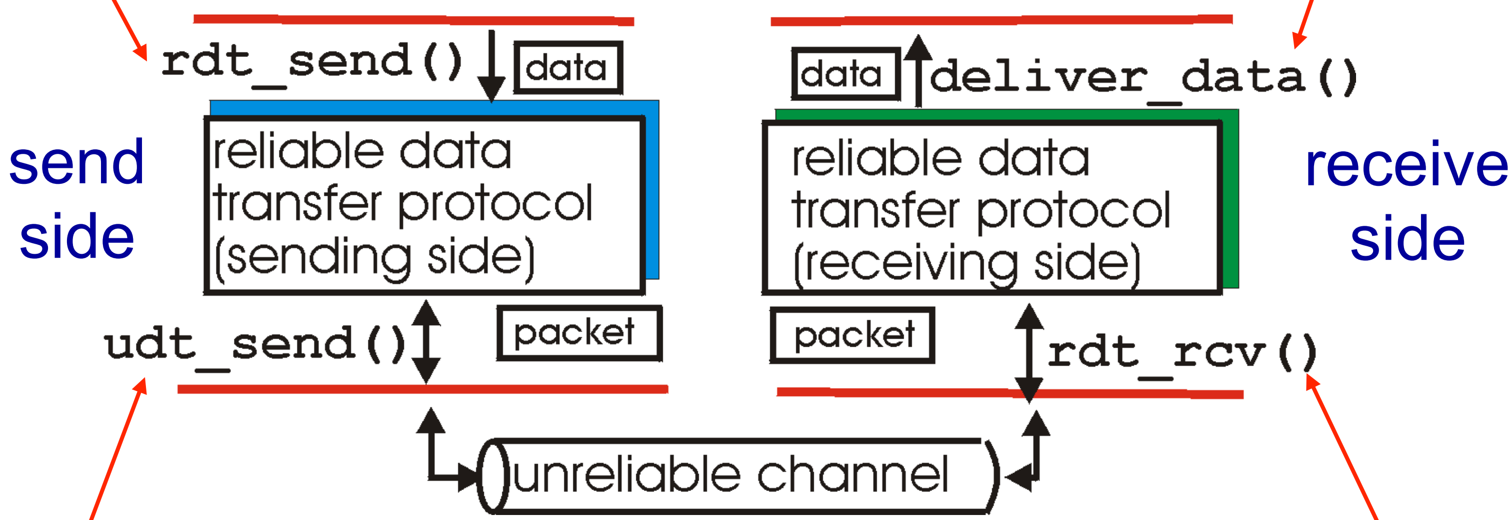


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



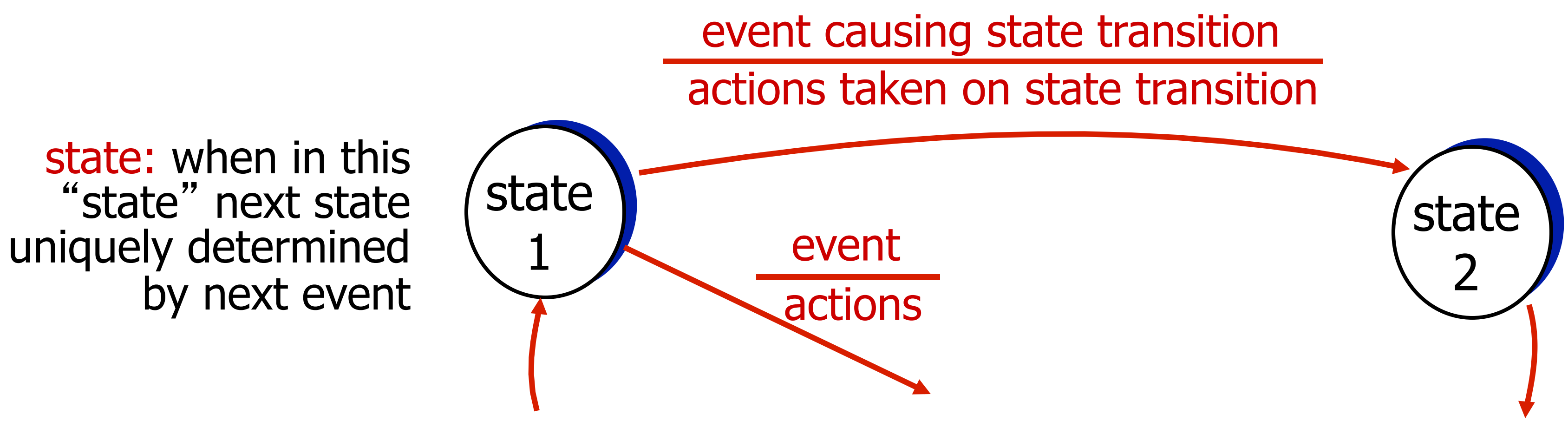
`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

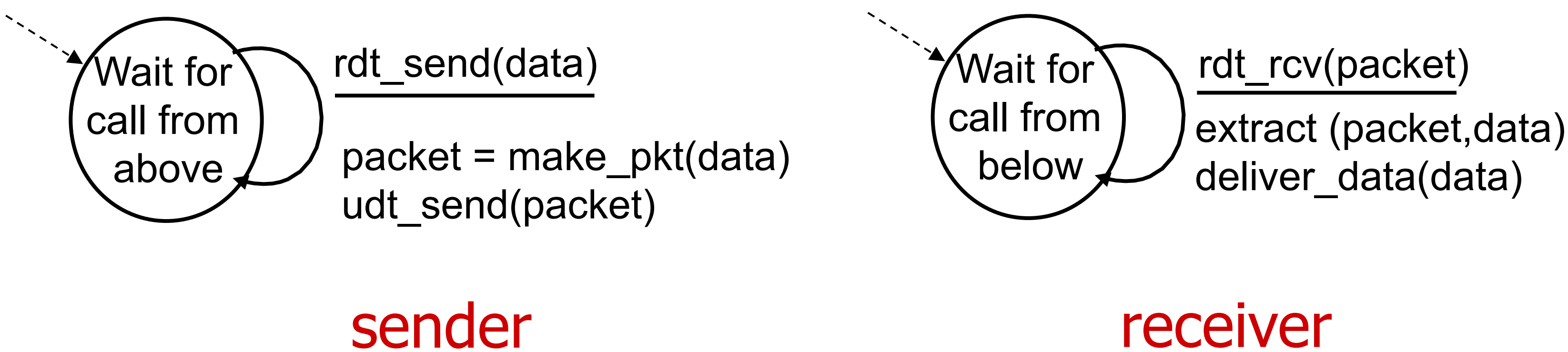
we' ll:

- incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



rdt1.0: reliable transfer over a reliable channel

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



rdt2.0: channel with bit errors

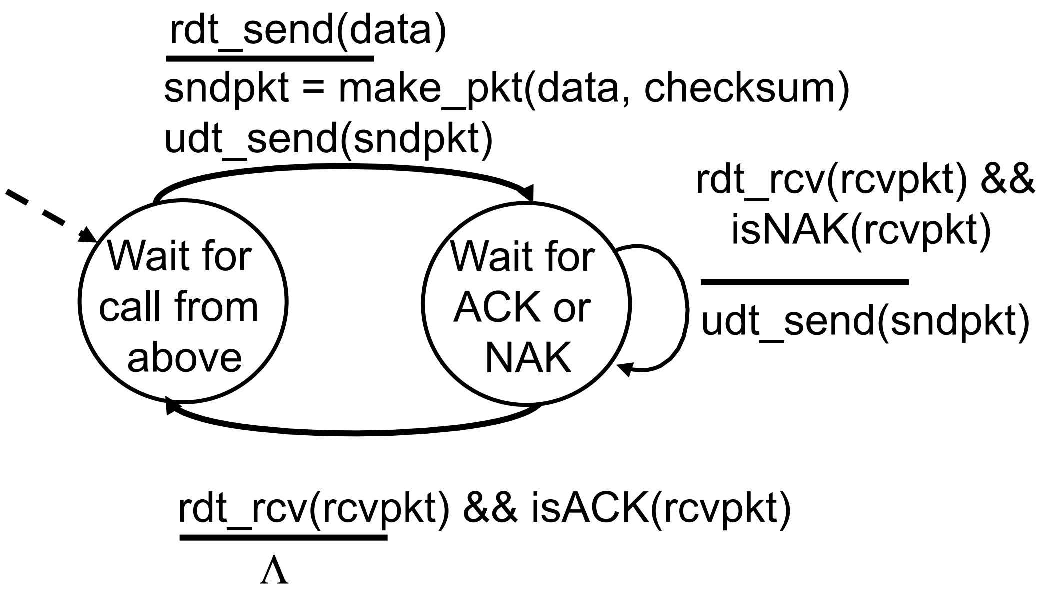
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

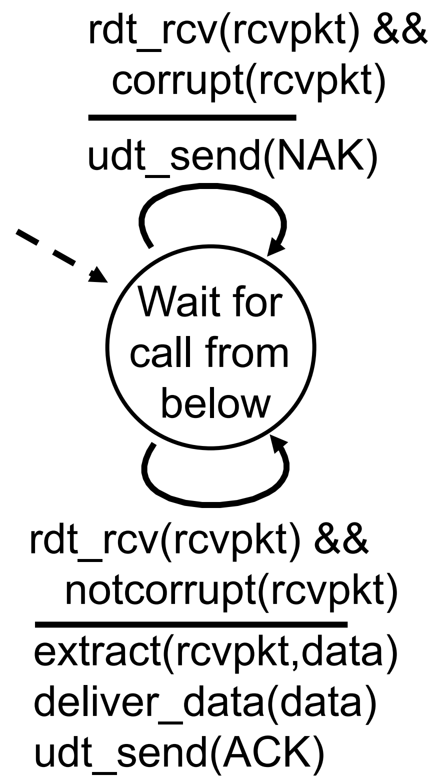
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

rdt2.0: FSM specification

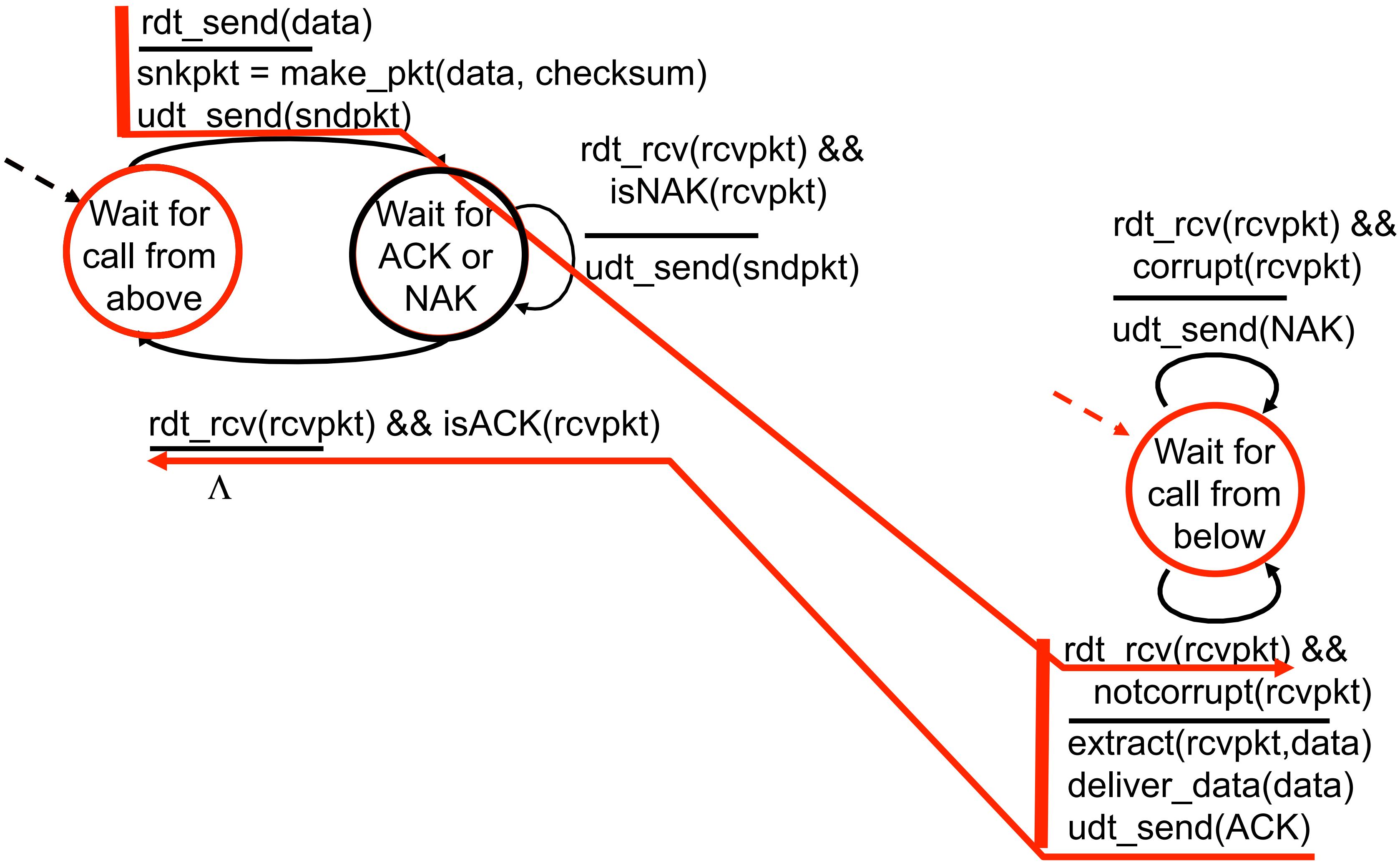


sender

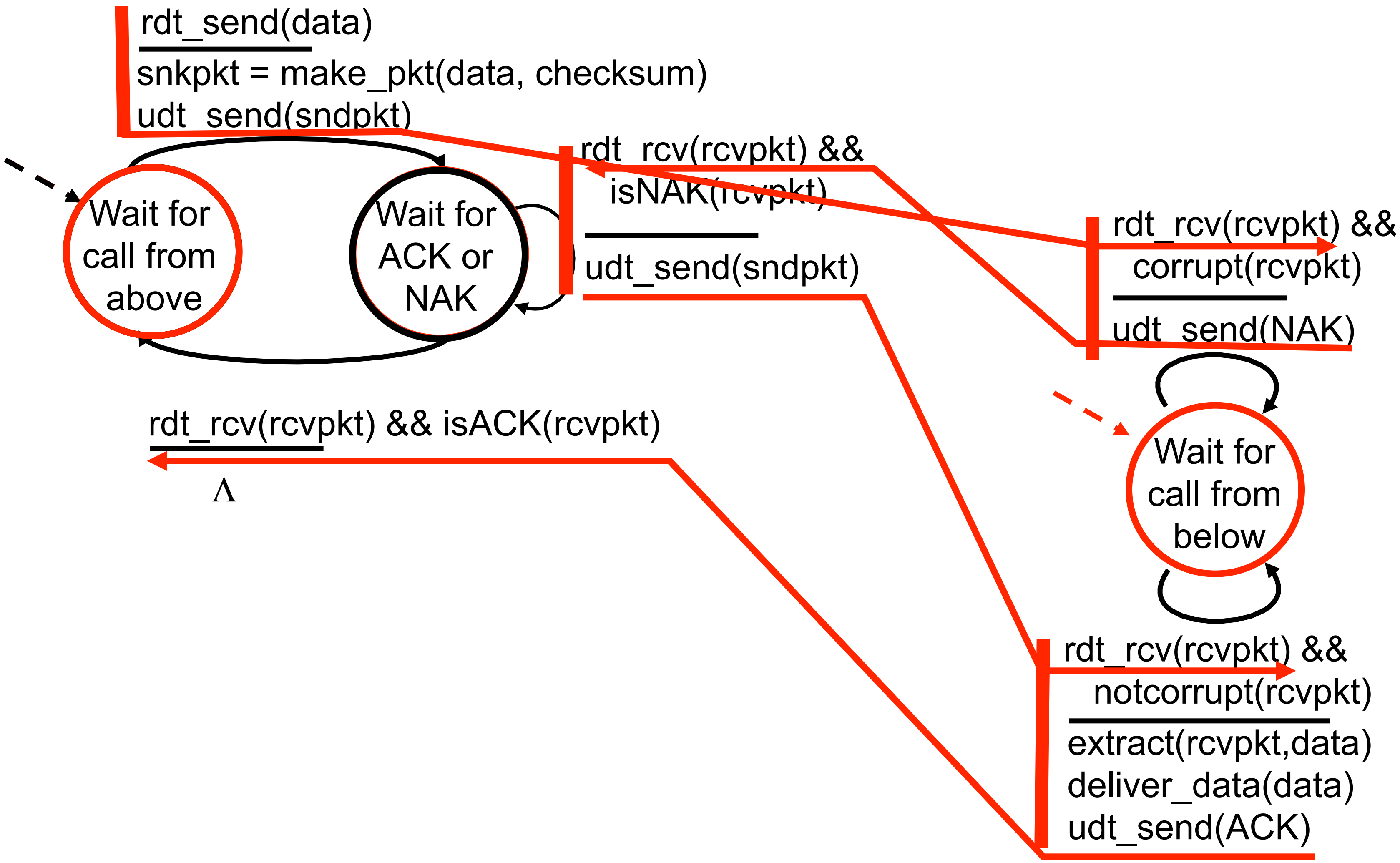
receiver



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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

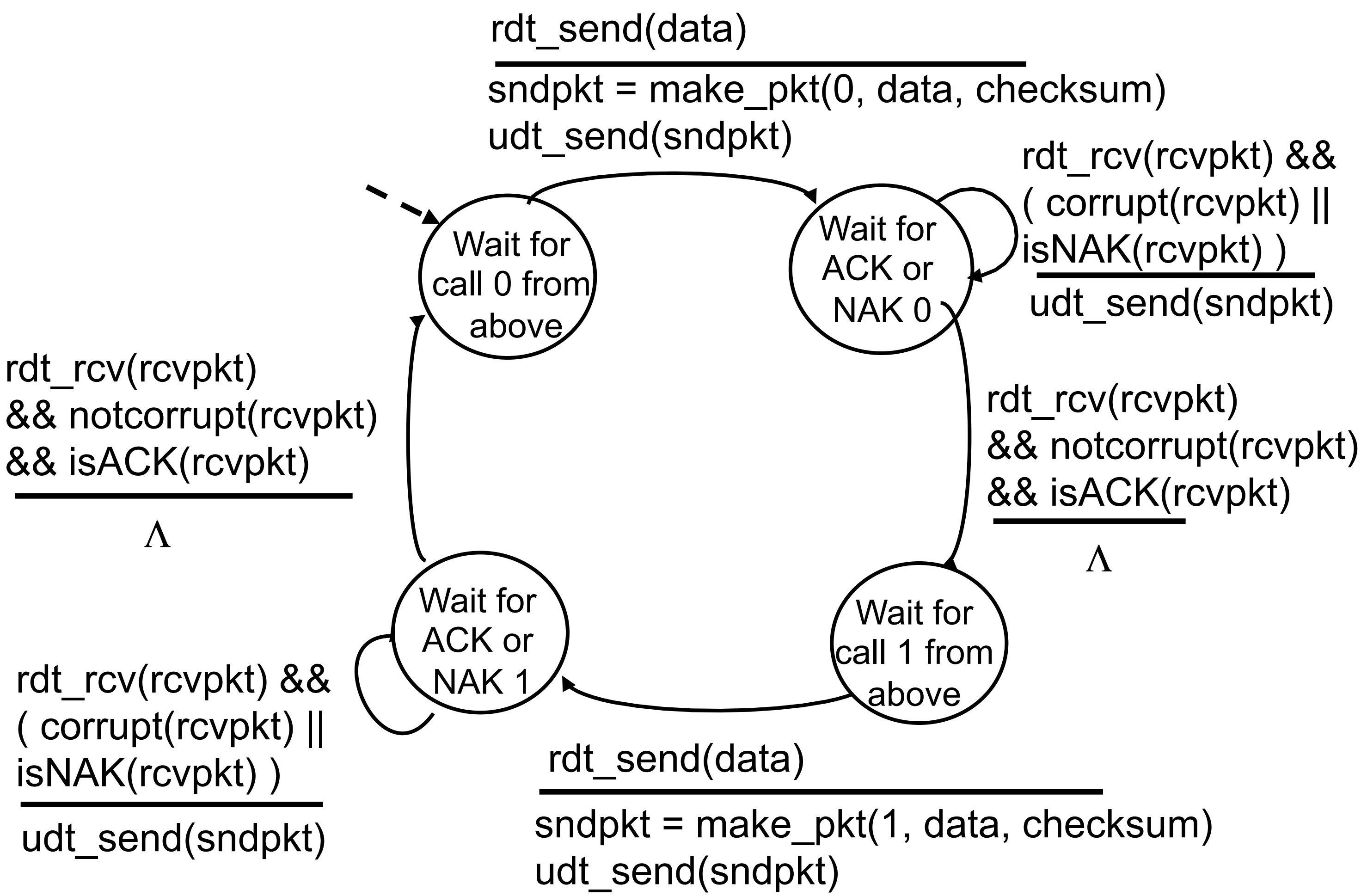
handling duplicates:

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

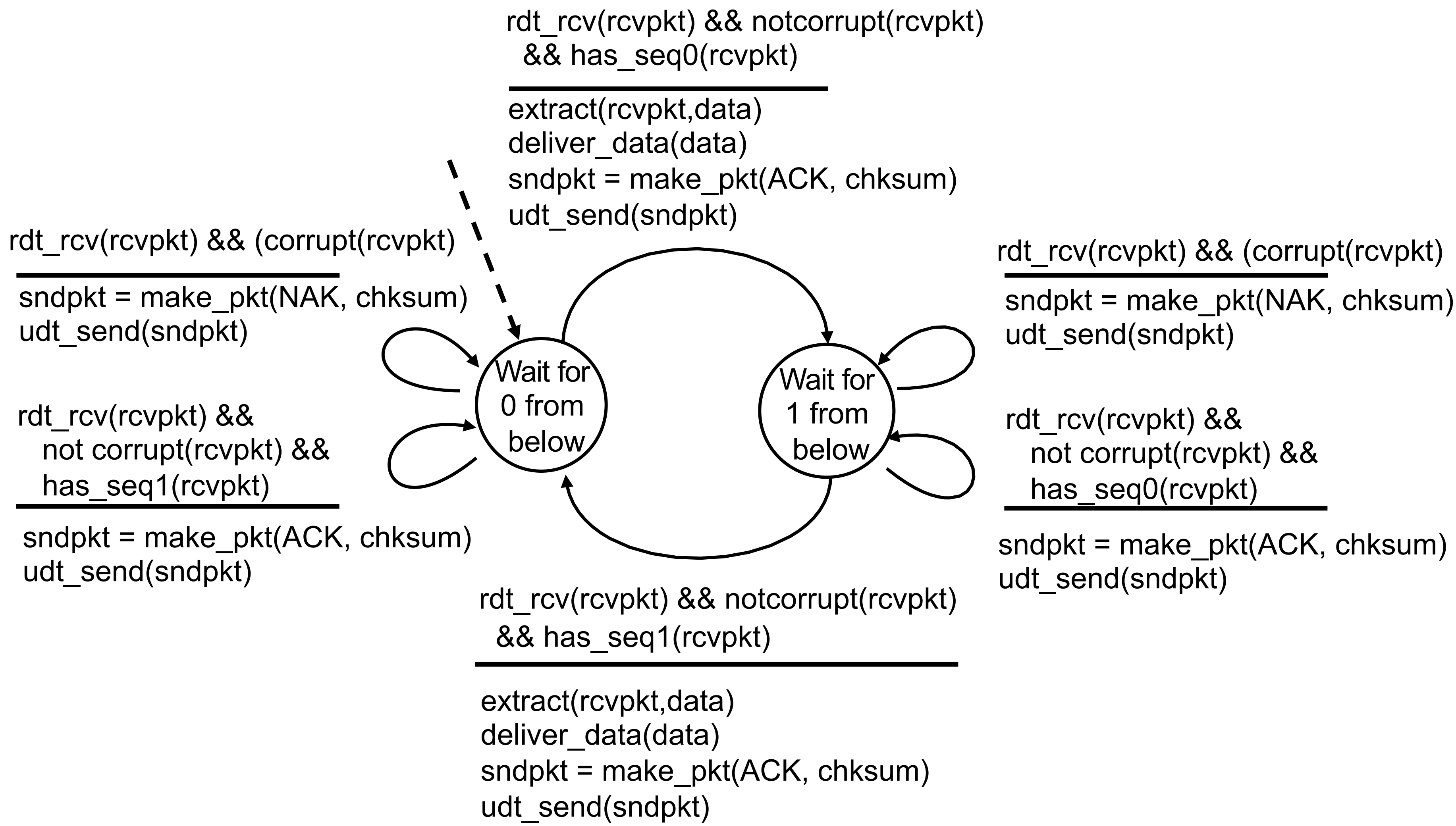
stop and wait

sender sends one packet, then waits for receiver response

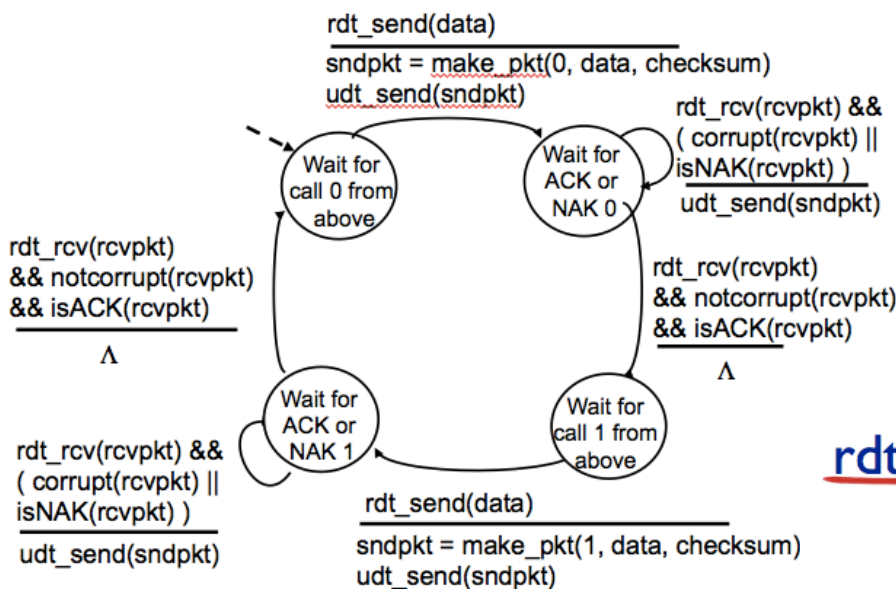
rdt2.1: sender, handles garbled ACK/NAKs



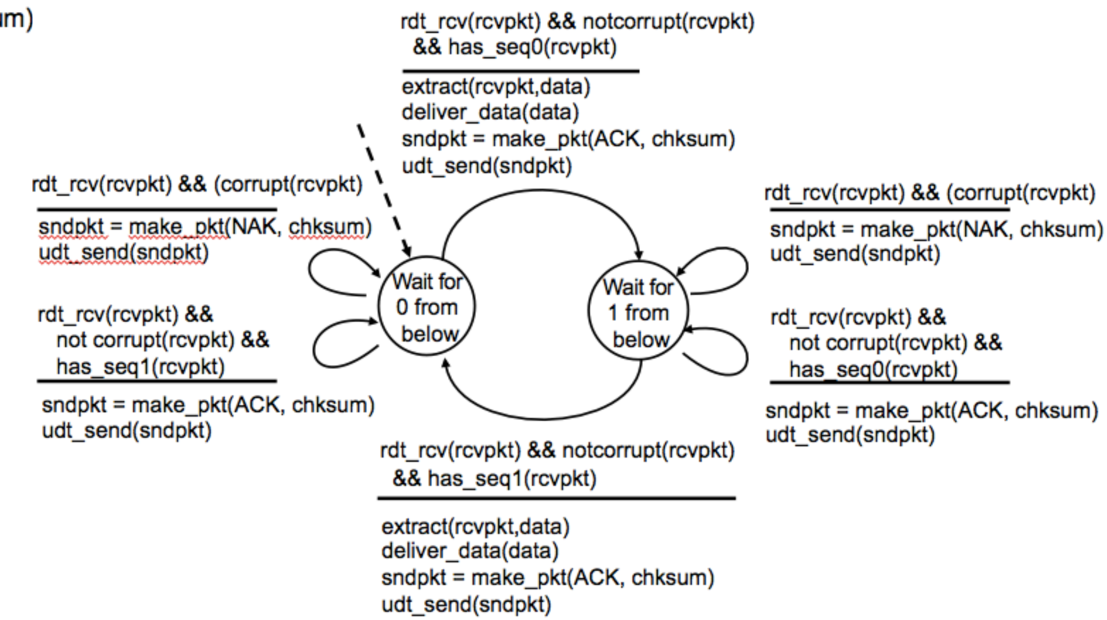
rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Q: Do we really need both ACKs and NACKs?

sender:

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

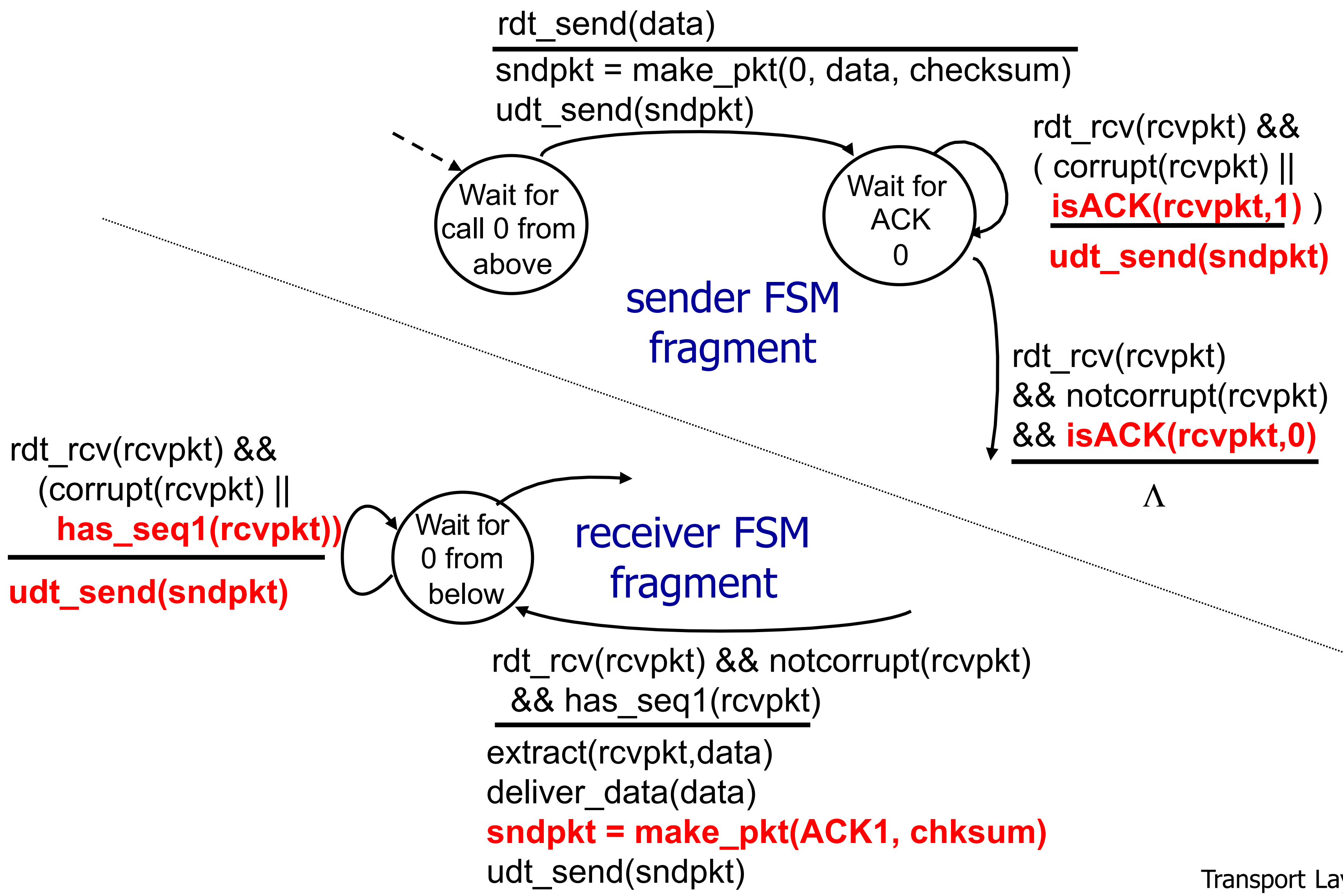
receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

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

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

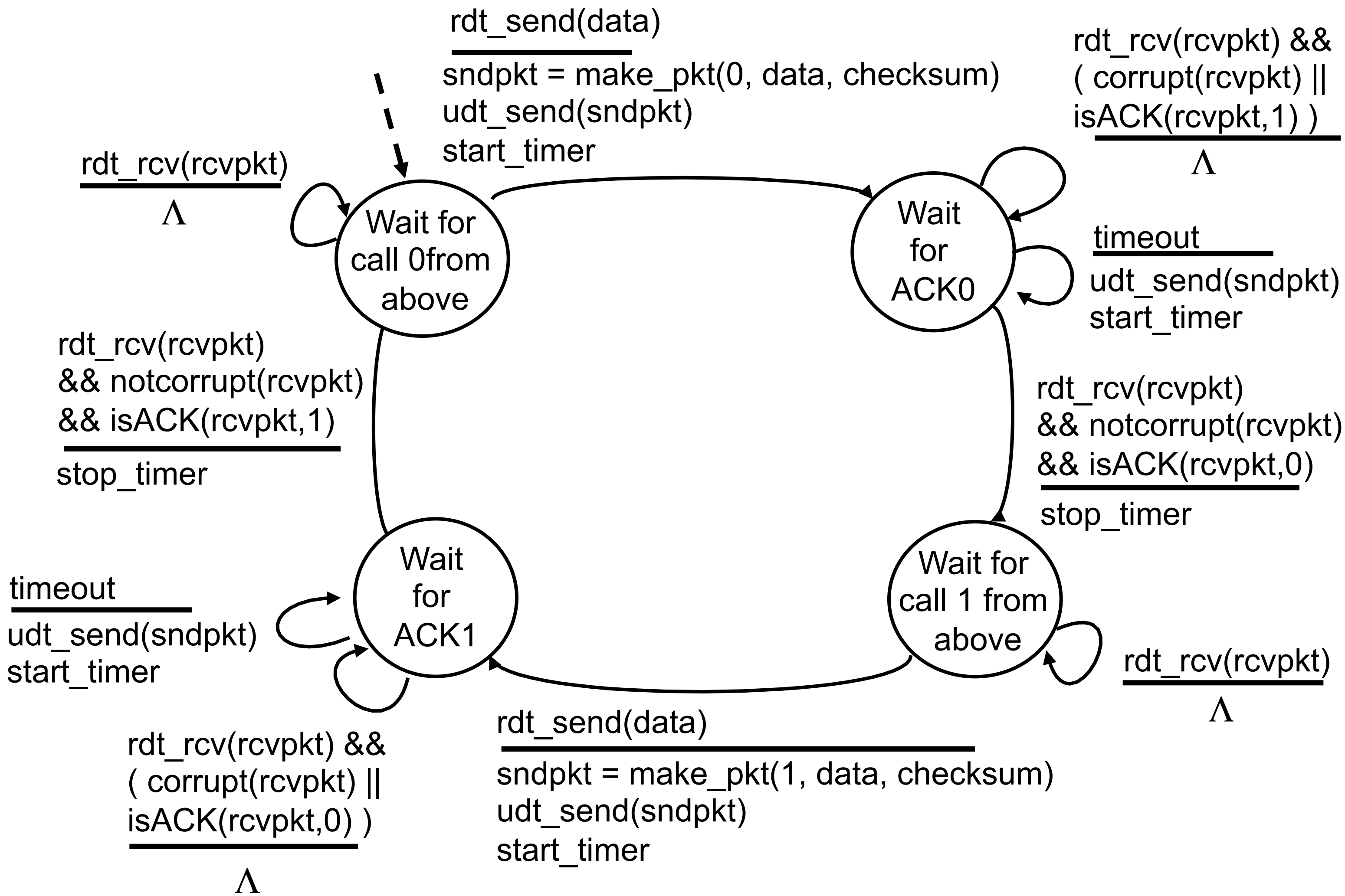
new assumption: underlying channel can also lose packets (data, ACKs)

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

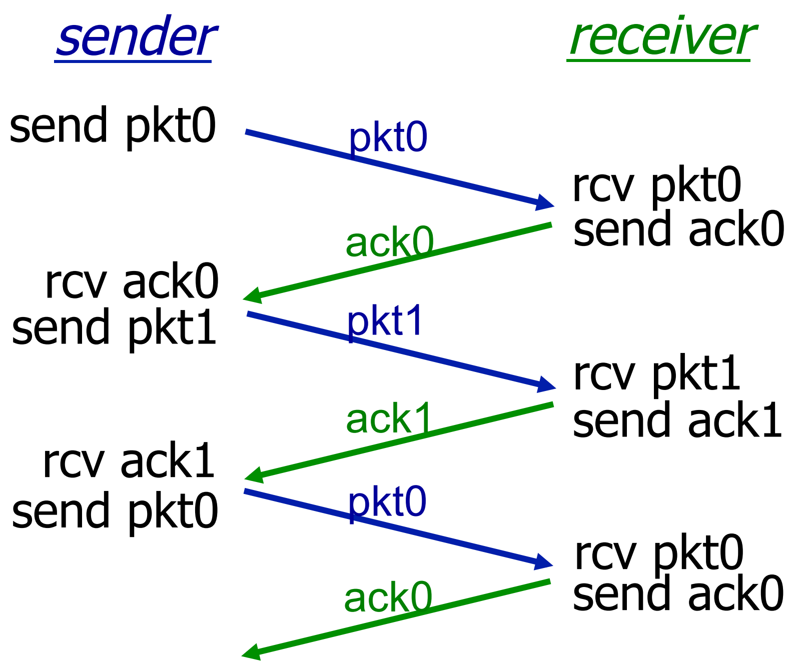
approach: sender waits "reasonable" amount of time for ACK

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

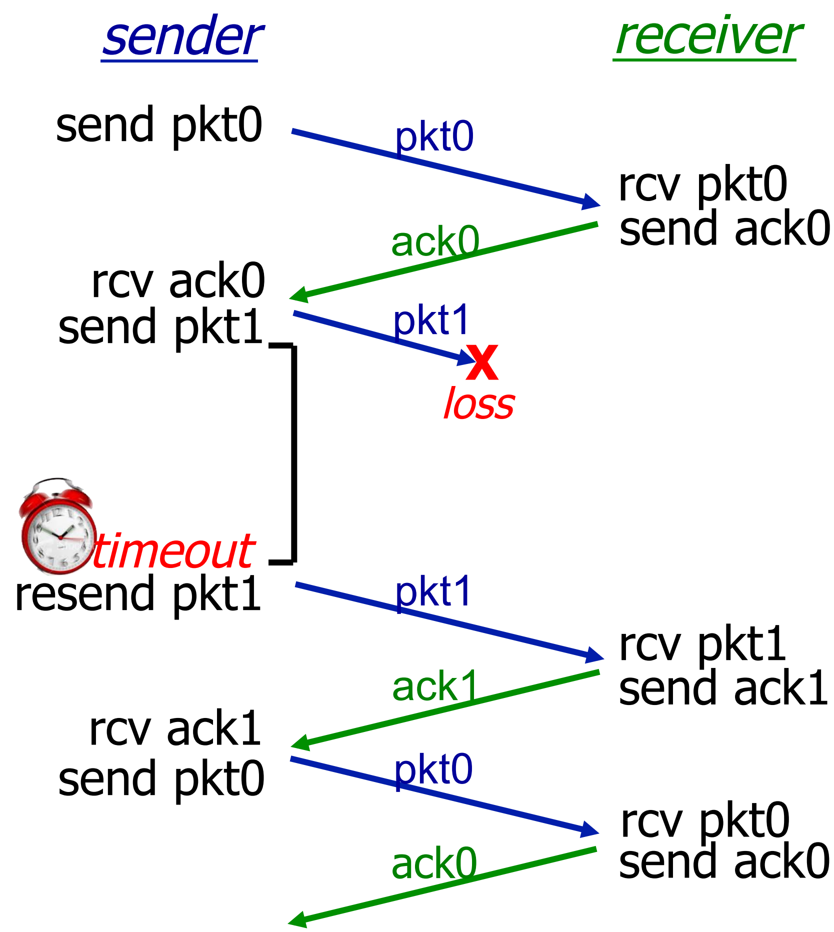
rdt3.0 sender



rdt3.0 in action

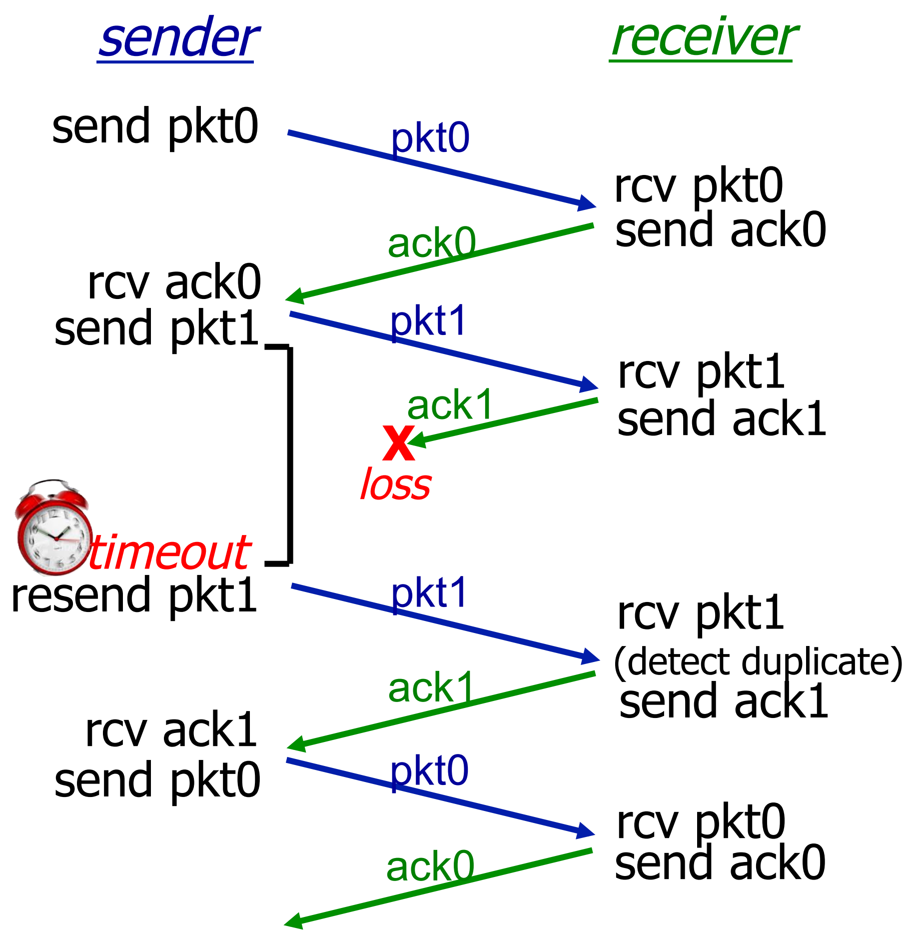


(a) no loss

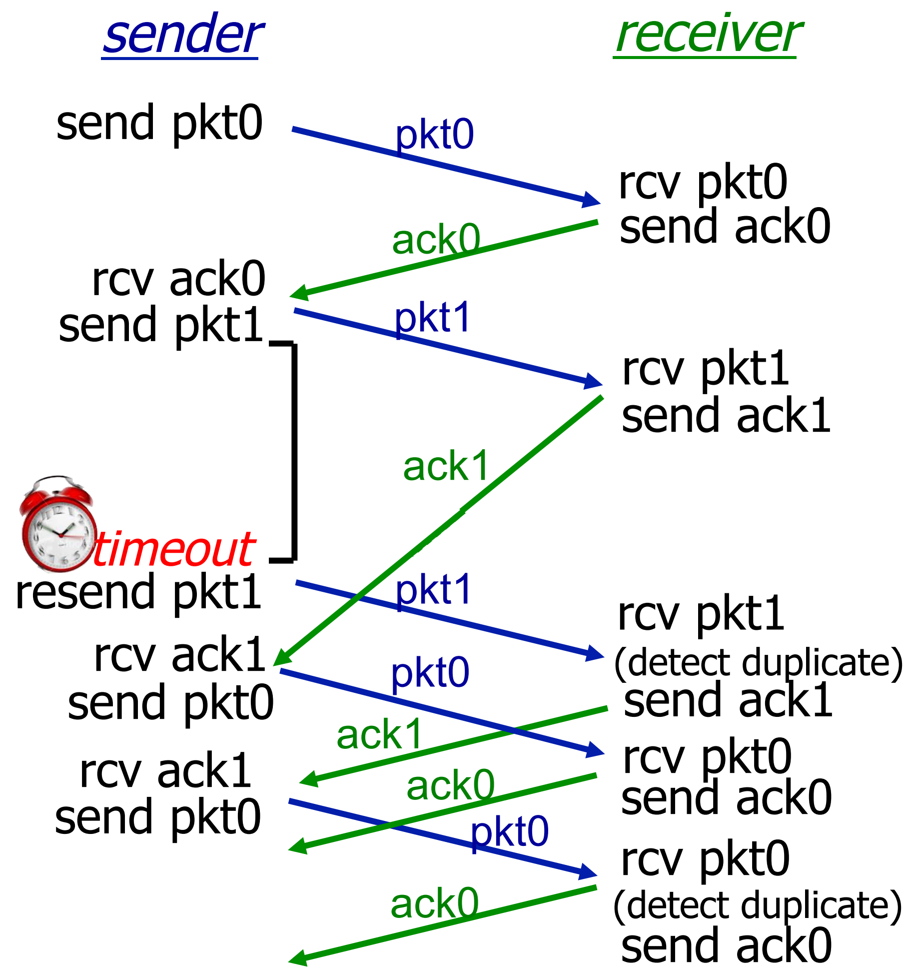


(b) packet loss

rdt3.0 in action



(c) ACK loss

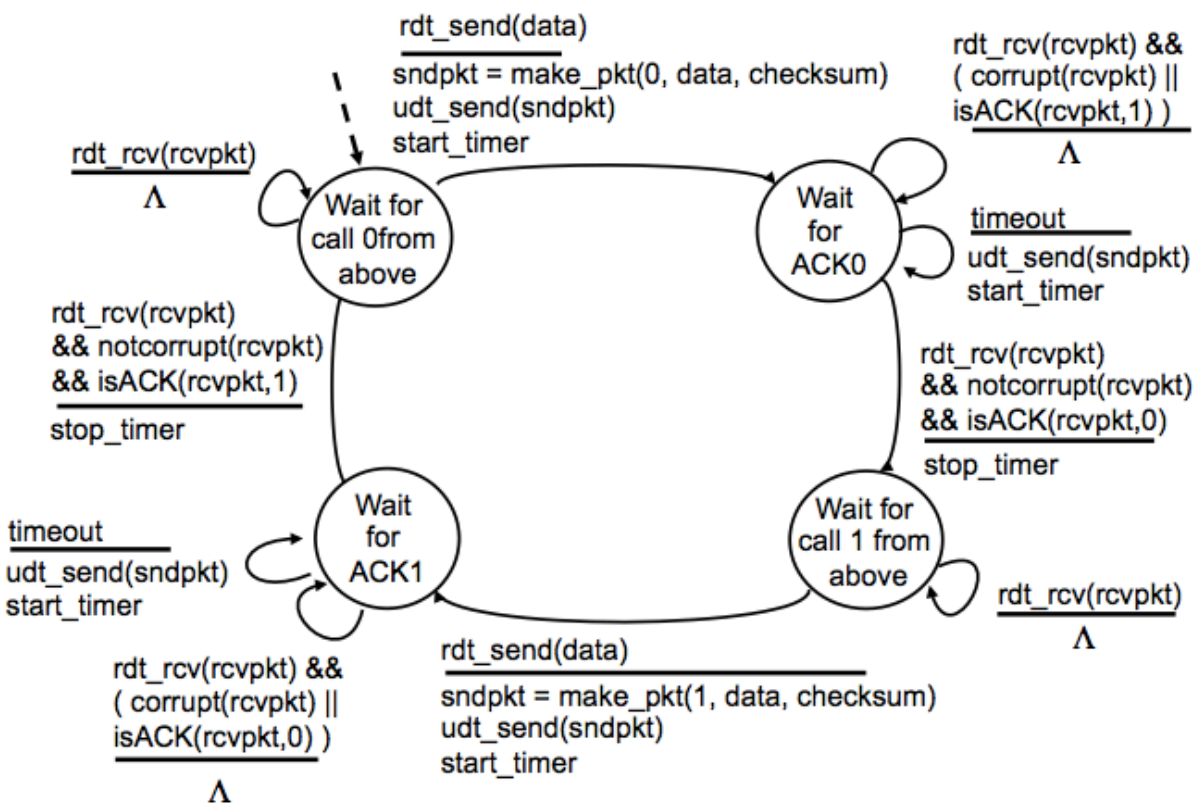


(d) premature timeout/ delayed ACK

Try writing rdt 3.0 receiver?

- Use rdt_rcv(), isCorrupt(), udt_send(pkt), extract(.), deliver(.), make_pkt(.), isAck(.), hasSeq(.)

rdt3.0 sender



Performance of rdt3.0

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

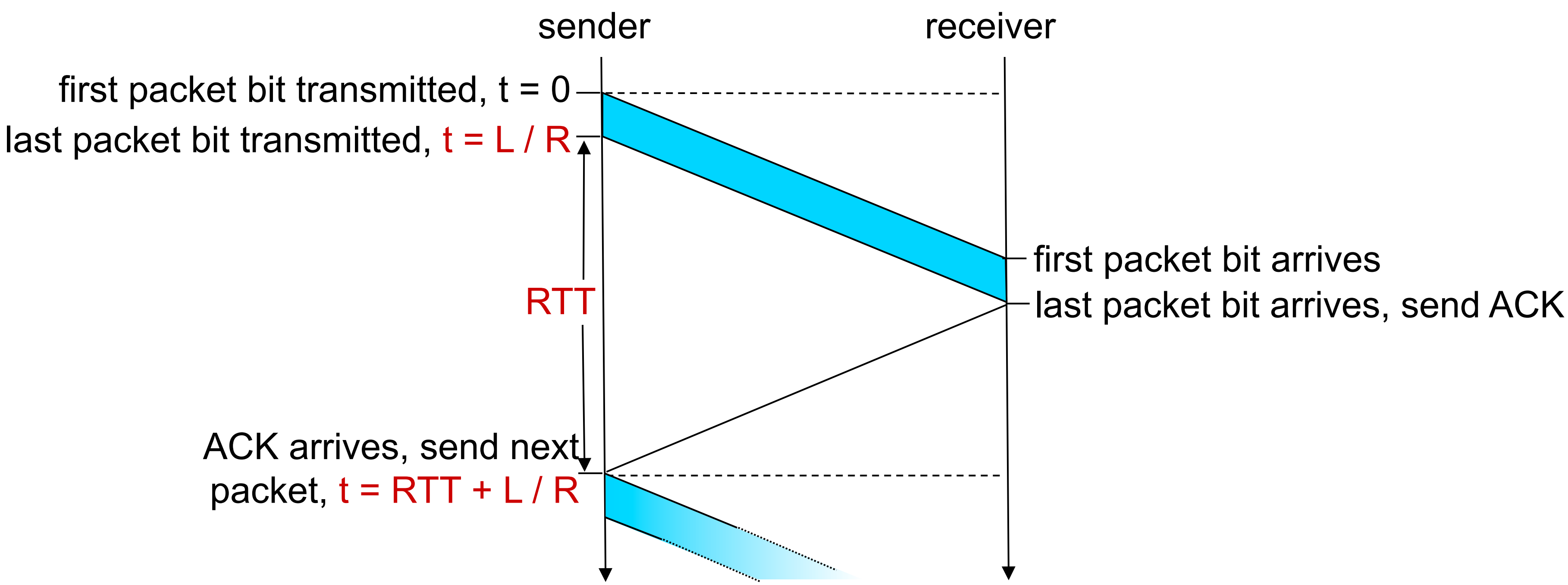
$$D = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U : *utilization* – fraction of time sender busy sending

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

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

rdt3.0: stop-and-wait operation

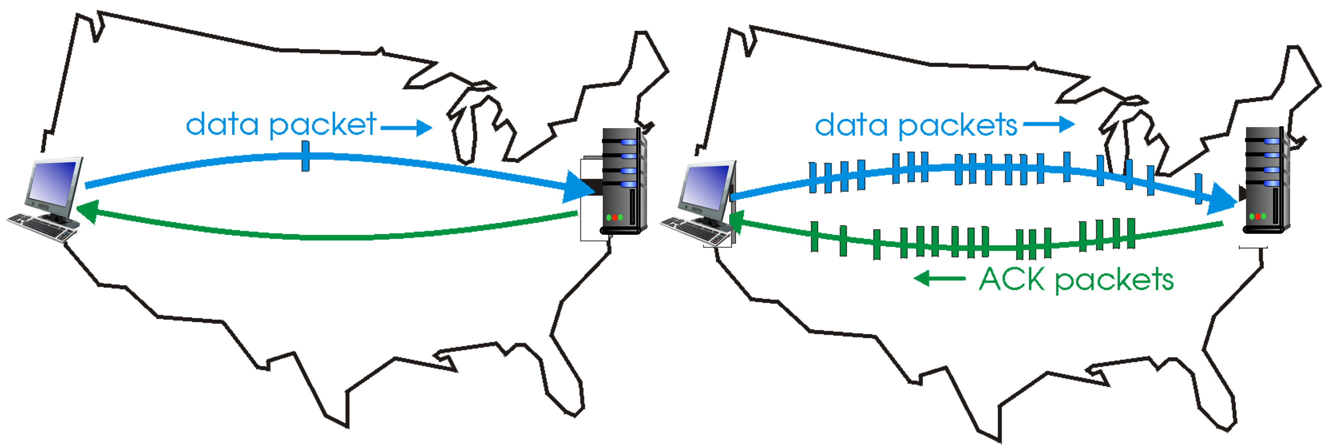


$$U = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

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

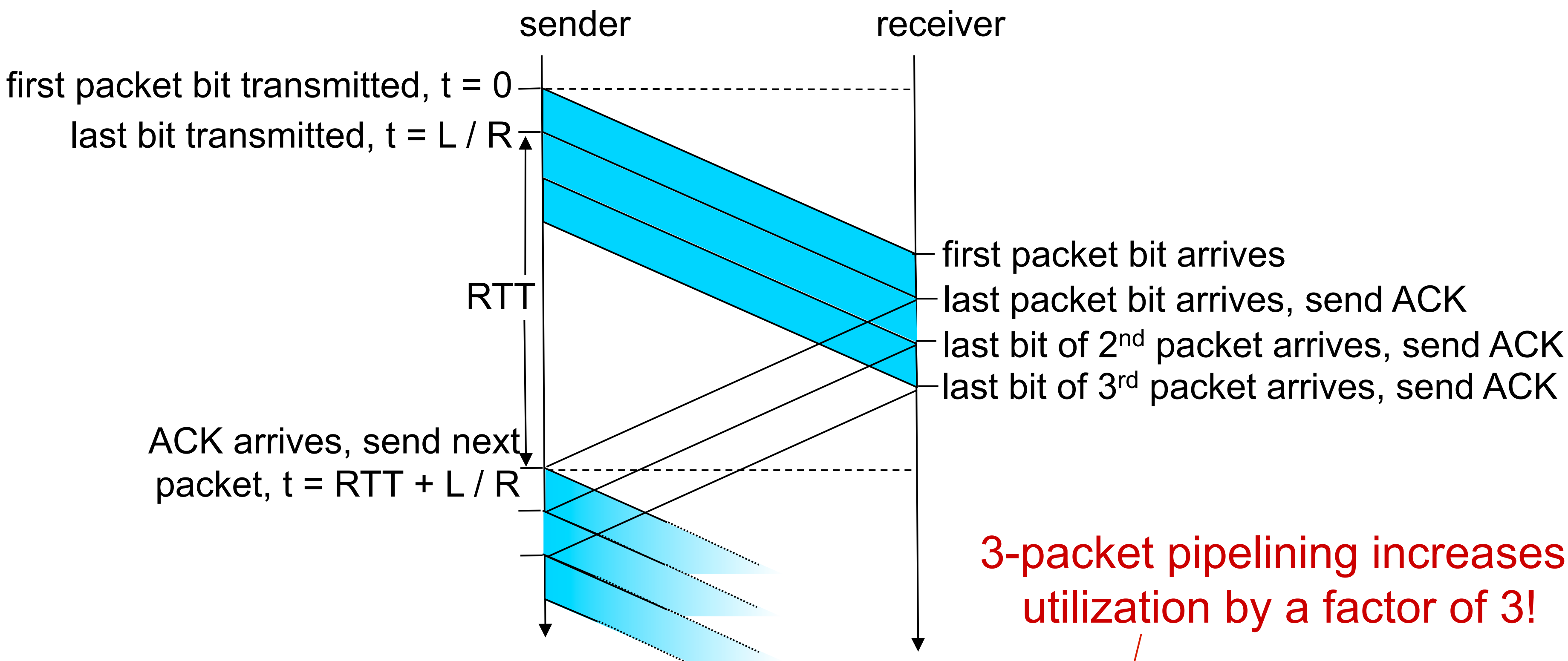
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

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

Pipelining: increased utilization



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

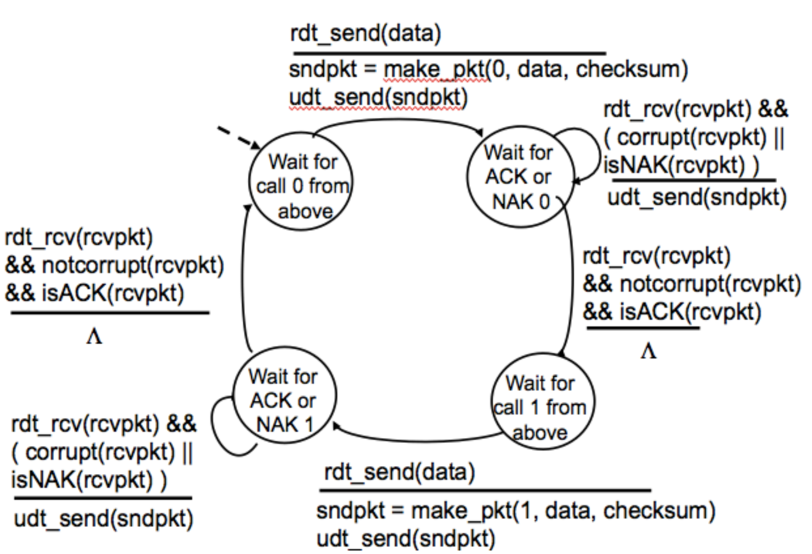
Q1: Reliable data transfer

- Which of the following are needed for reliable data transfer with only packet corruption (and no loss or reordering)? Use only as much as is strictly needed.
  A. Checksums
  B. Checksums, ACKs, NACKs
  C. Checksums, ACKs
  D. Checksums, ACKs, sequence numbers
  E. Checksums, ACKs, NACKs, sequence numbers

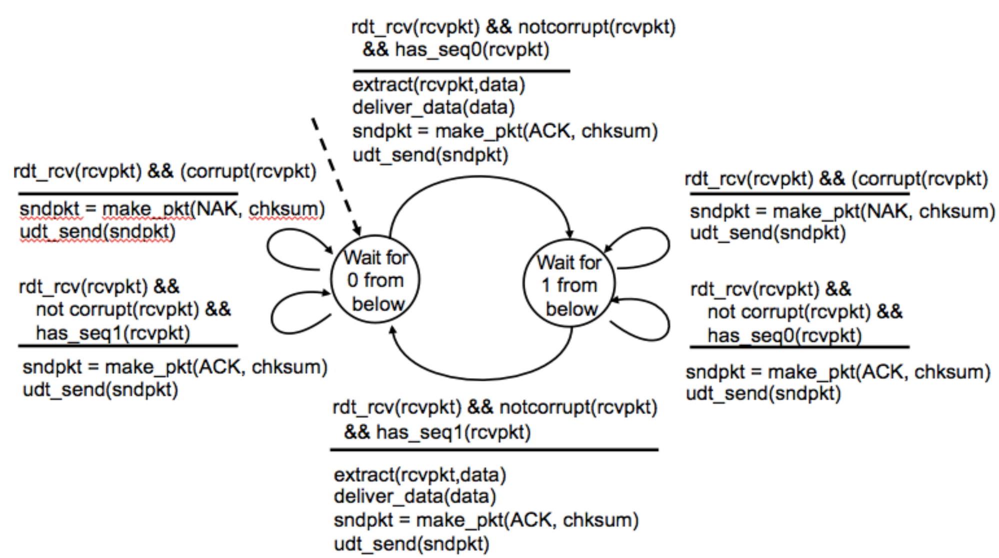
Q2: Reliable data transfer

- If packets (and ACKs and NACKs) could be lost, which of the following is true of rdt 2.1 (or 2.2)?
  A. Reliable, in-order delivery is still achieved.
  B. The protocol will get get stuck.
  C. The protocol will continue making progress but may skip delivering some messages.

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



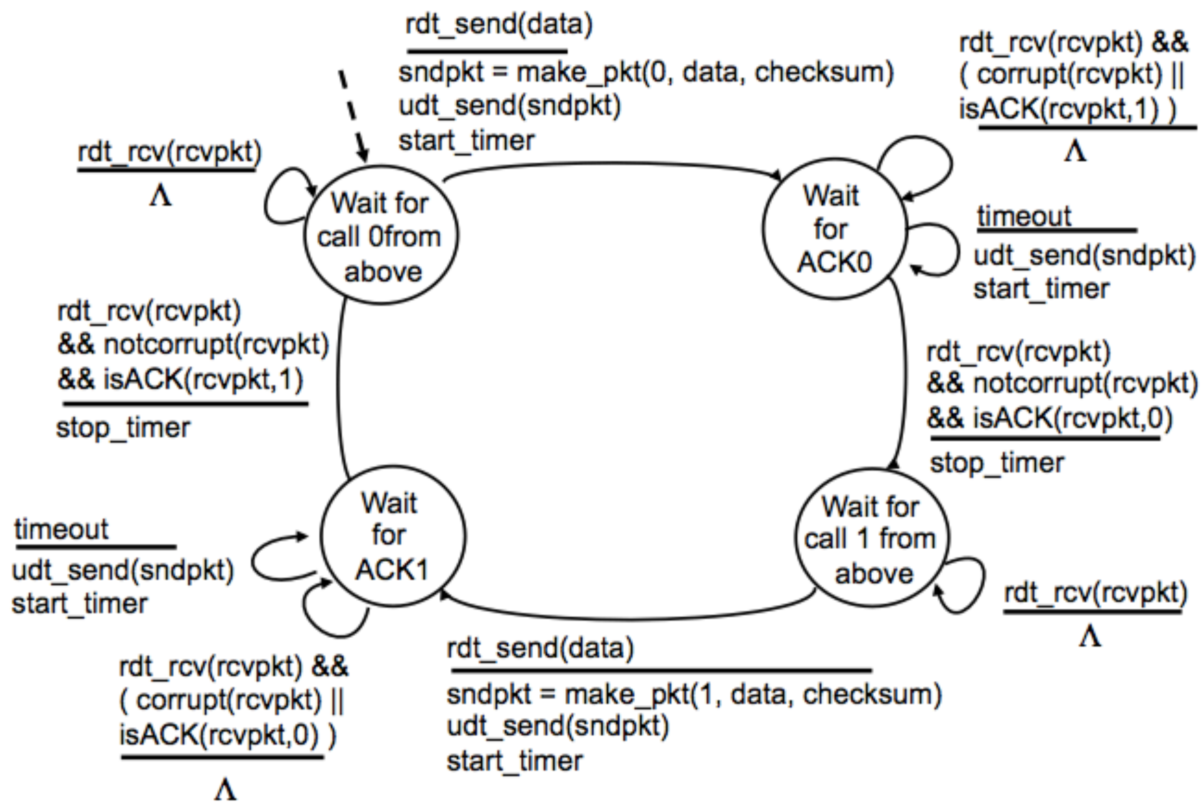
Q3: Reliable data transfer

- Which of the following are needed for reliable data transfer to handle packet corruption and loss? Use only as much as is strictly needed.
  A. Checksums, timeouts, and fries with that
  B. Checksums, ACKs, sequence numbers
  C. Checksums, ACKs, timeouts, pipelined protocol
  D. Checksums, ACKs, sequence numbers, timeouts
  E. Checksums, ACKs, NACKs, sequence numbers, timeouts

Q4: Reliable data transfer

- rdt 3.0 handles corruption and loss but not reordering. True or false?
  A. True
  B. False

rdt3.0 sender



Pipelined protocols: overview

Go-back-N:

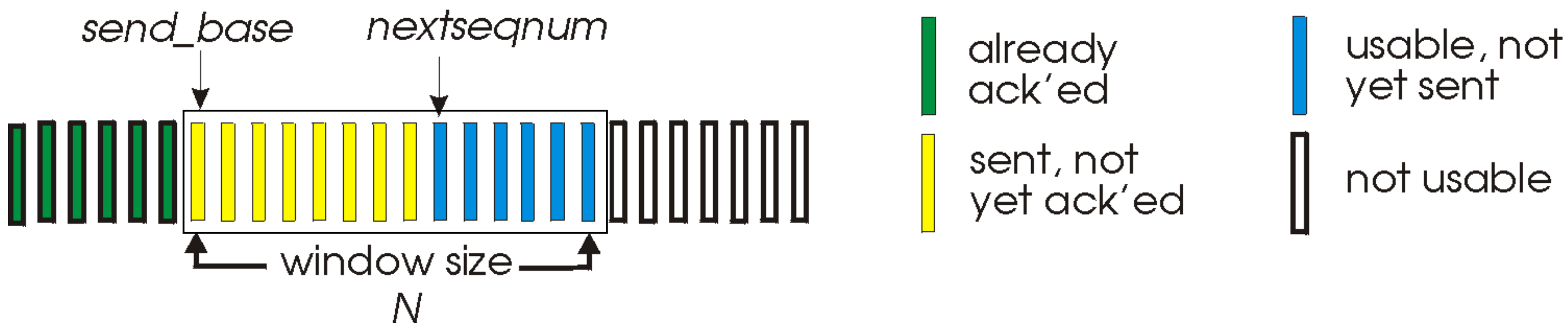
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

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

Go-Back-N: sender

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



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

GBN: sender extended FSM

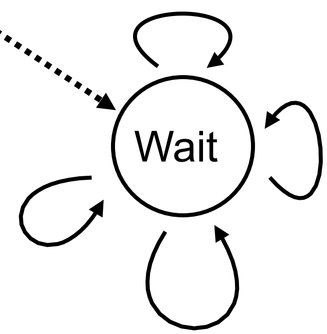
rdt_send(data)

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

Λ

---

base=1
nextseqnum=1



timeout

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

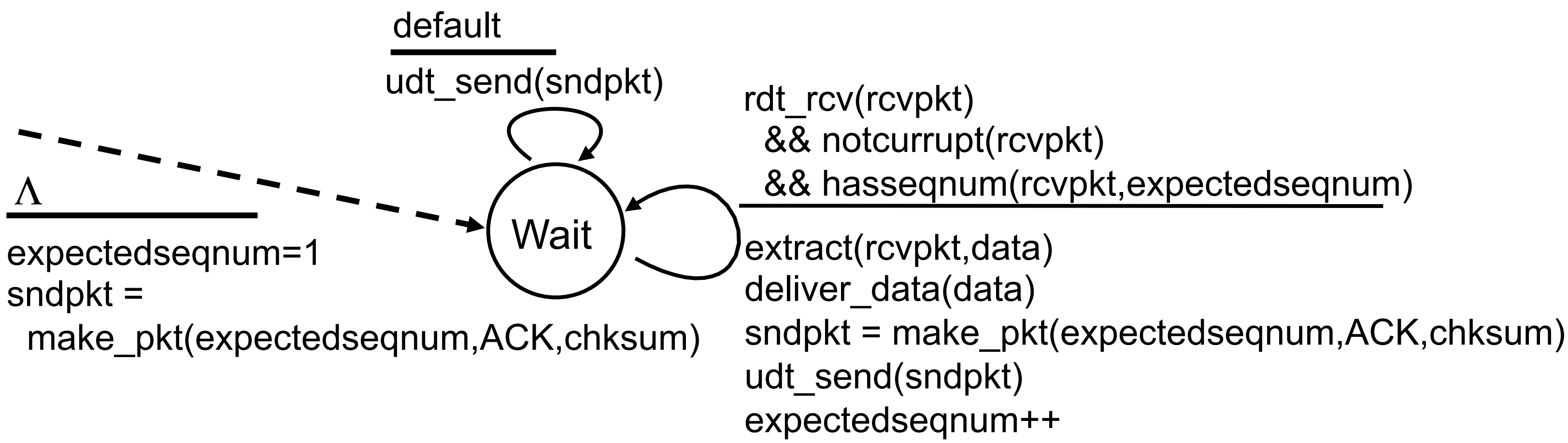
rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

---

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    restart_timer
```

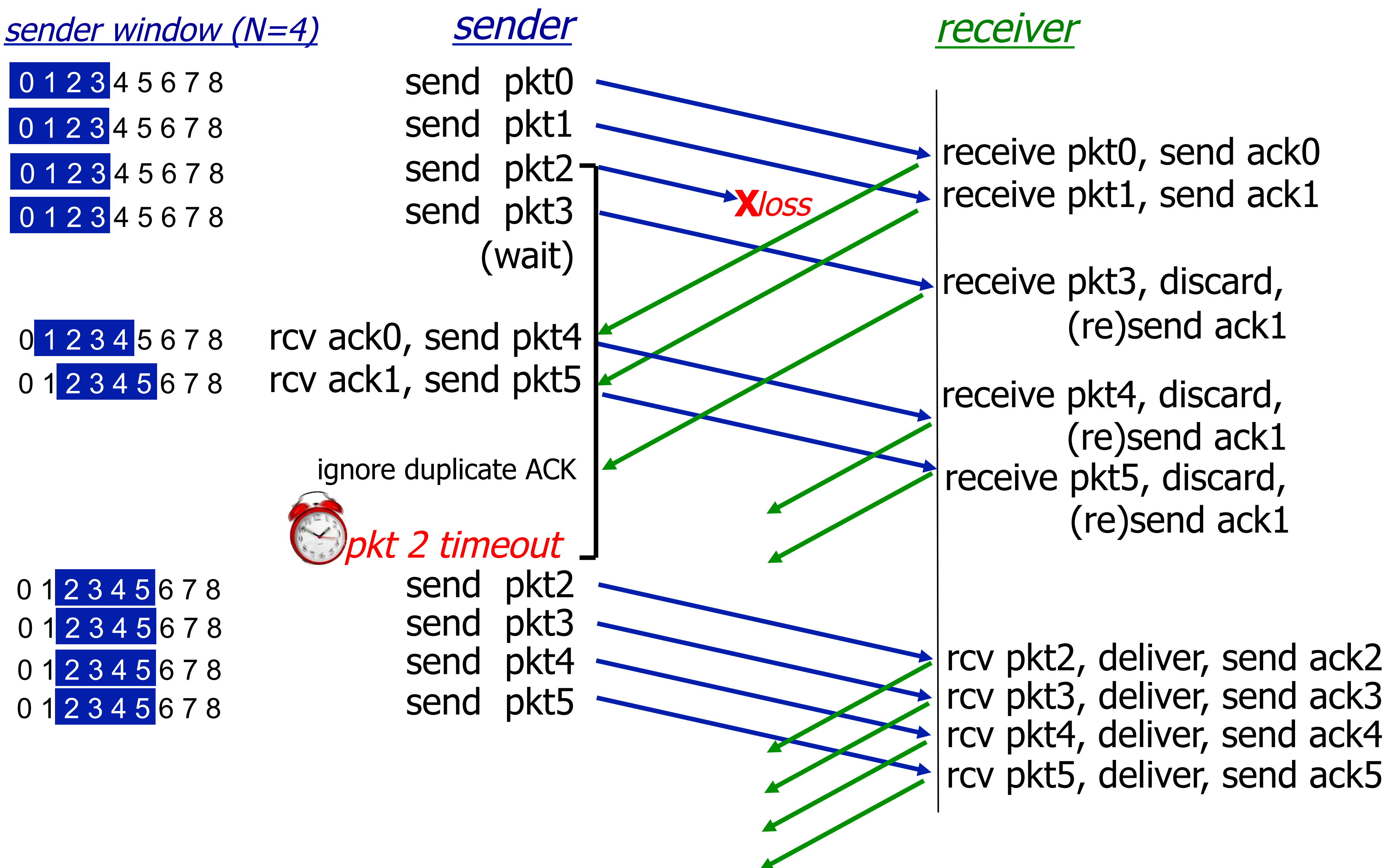
GBN: receiver extended FSM



ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

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

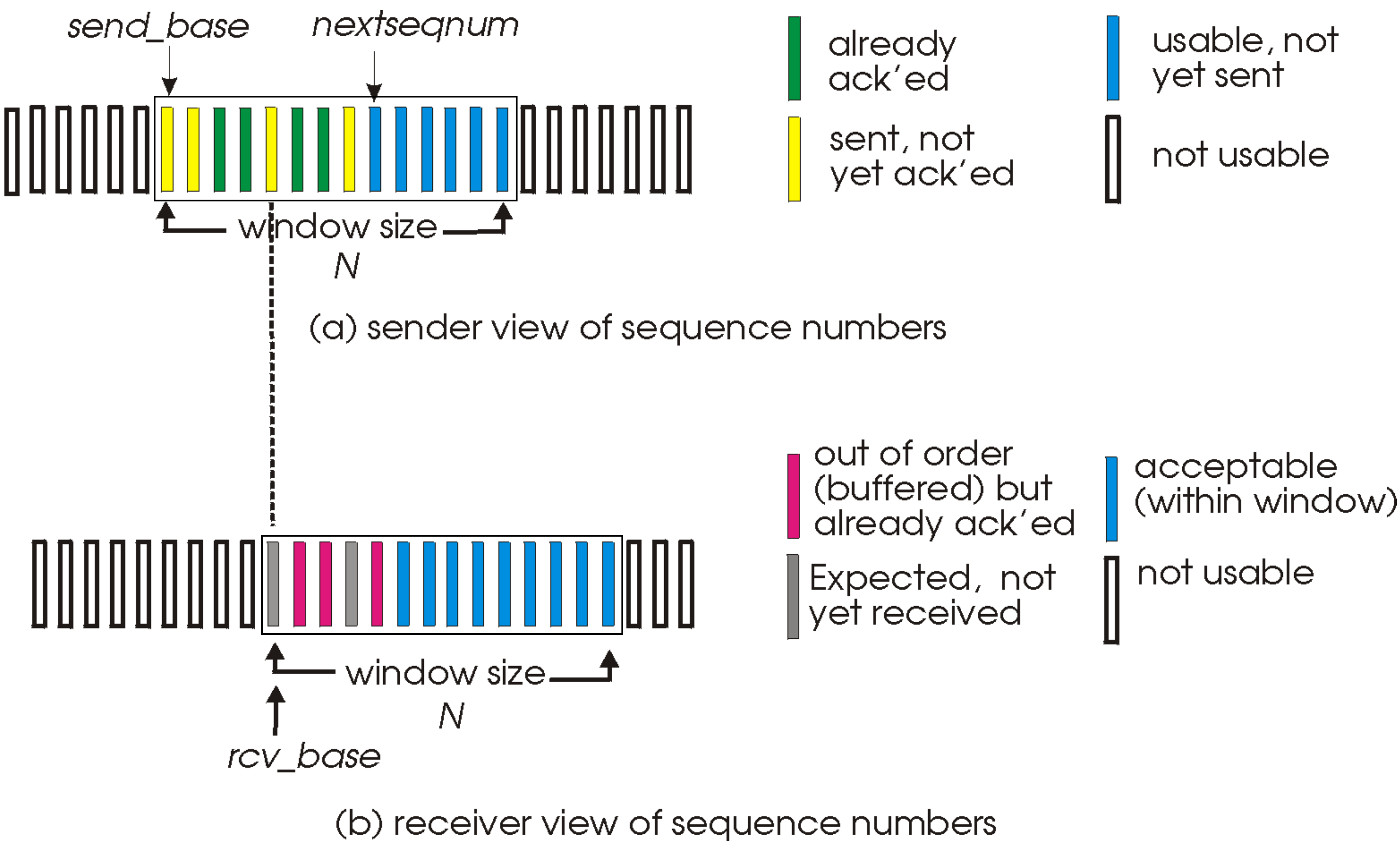
GBN in action



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]:

- 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

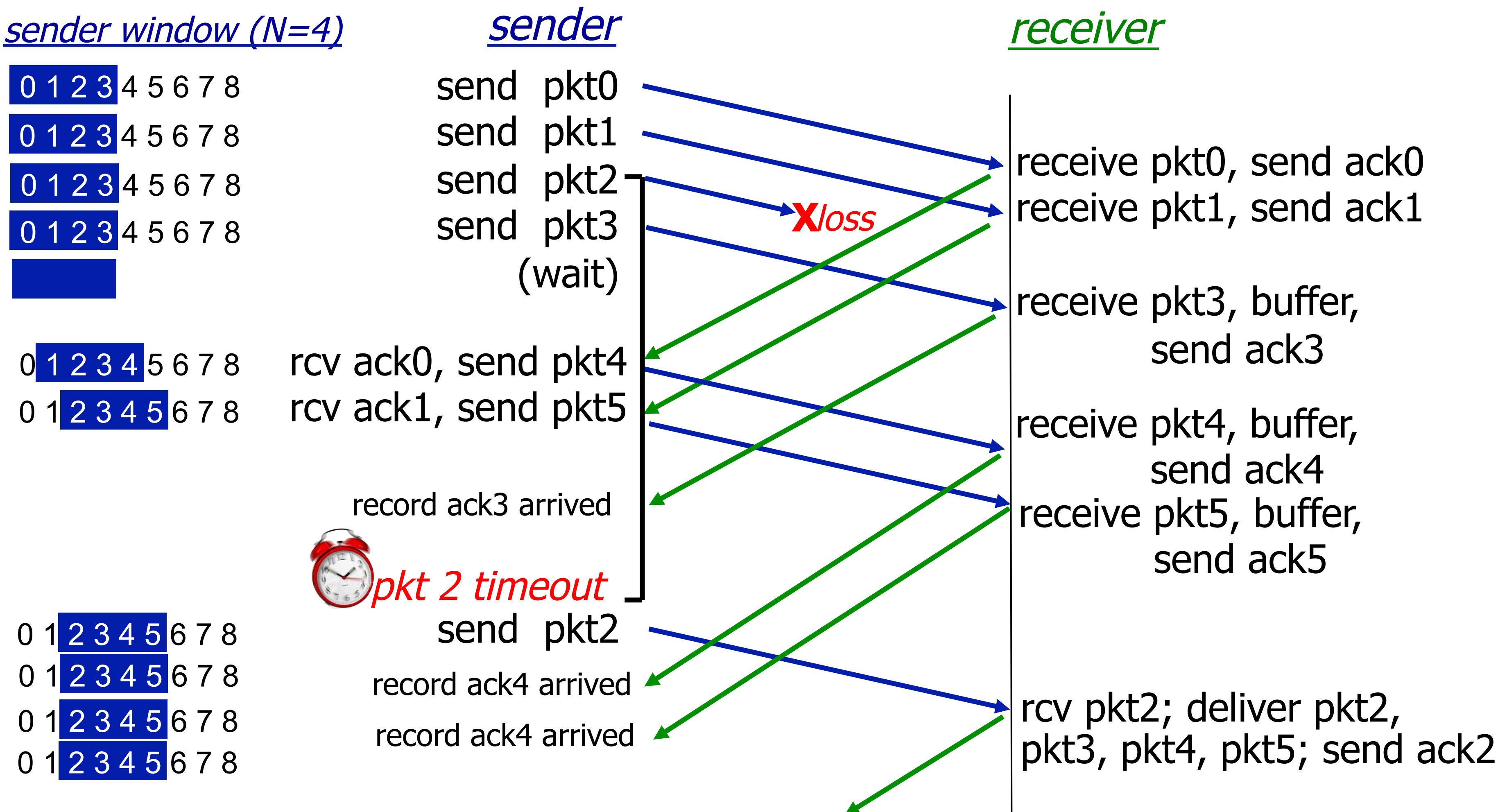
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



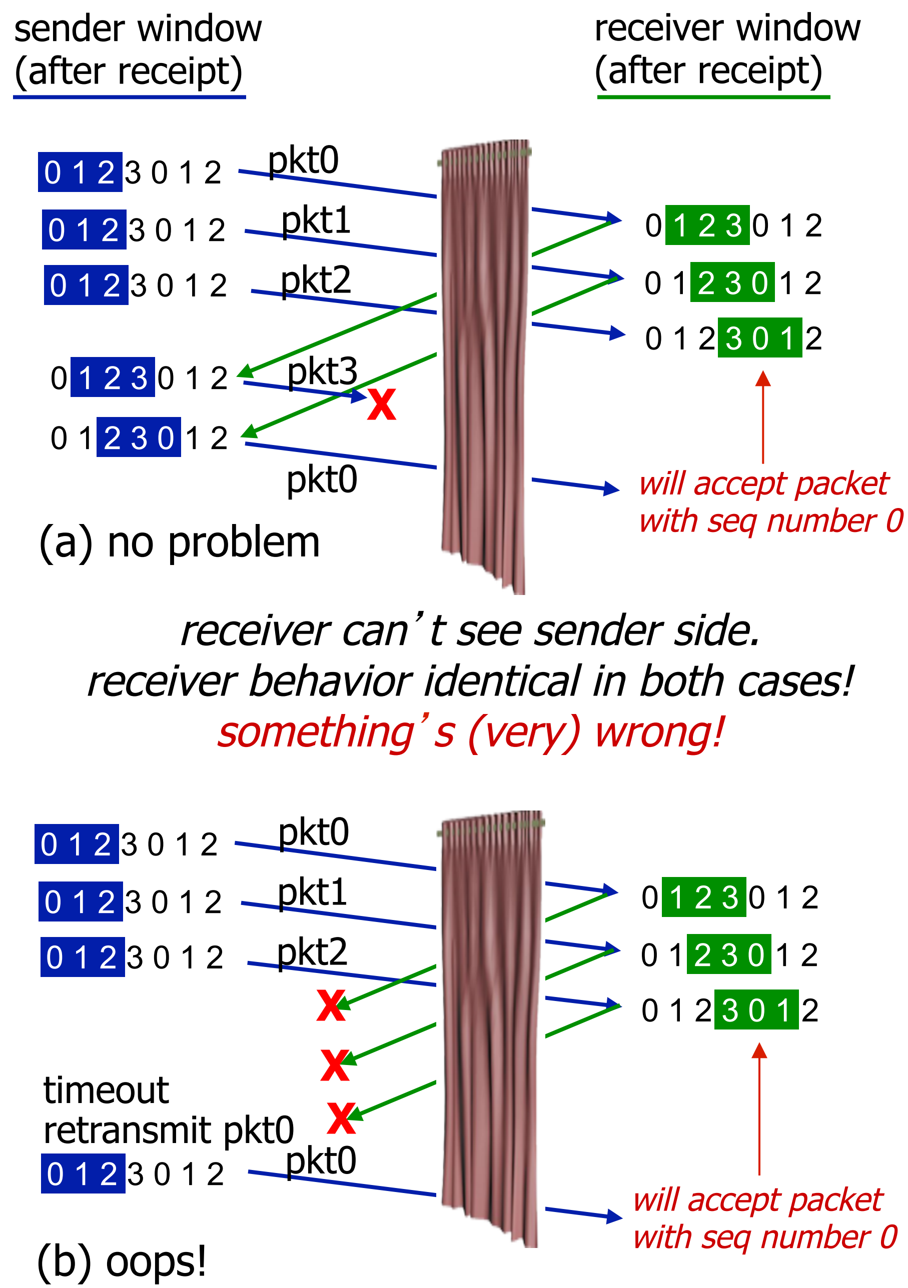
***Q**: what happens when ack2 arrives?*

Selective repeat: dilemma

example:

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

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



Q1: RDT pipelining

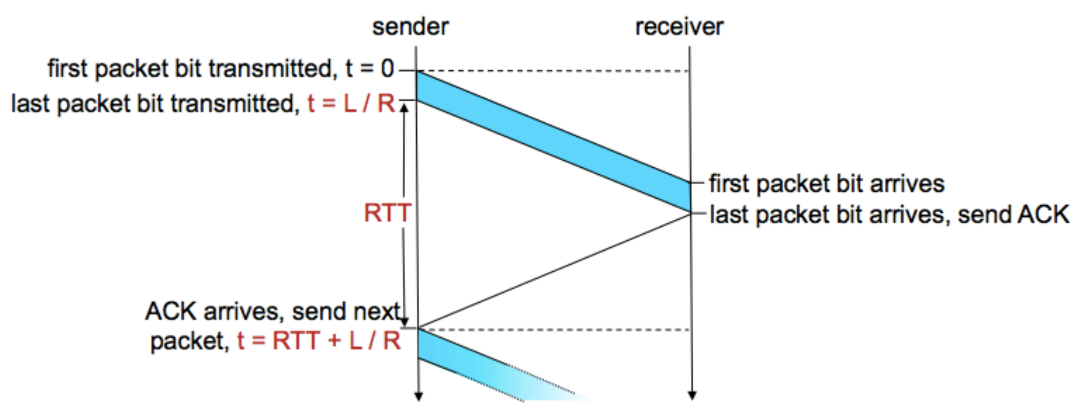
- Consider a path of bottleneck capacity C, round-trip time T, and maximum segment size S. If a pipelined rdt protocol maintains a window of N outstanding packets, how much does it improve throughput compared to a stop-and-wait rdt protocol (when no losses are actually happening)? Assume $NS/C < T$.
  A. N
  B. $NS/(CT+S)$
  C. $(NS/C)/(T+NS/C)$
  D. NTC/S

Q2: RDT pipelining

- Consider a path of bottleneck capacity C, round-trip time T, and maximum segment size S. What is the greatest throughput improvement factor that an ideal pipelined protocol (assuming corruptions and loss are negligible) can provide compared to a stop-and-wait protocol?

  A. (CT+S)/S
  B. 2S/(CT+S)
  C. (S/C)/(T+S/C)
  D. (TC/S)^2

rdt3.0: stop-and-wait operation



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

Q3 UDP & TCP

- Which of the following is true?

A. UDP does not maintain connection state and does not have error detection

B. TCP is a connectionless protocol with reliable, in-order delivery and error detection

C. UDP has error detection but no connection state

D. UDP only has error detection but TCP also has error correction

Q4 Go-back-N, selective repeat

- Which of the following is *not* true?

A. GBN uses cumulative ACKs, SR uses individual ACKs
B. Both GBN and SR use timeouts to address packet loss
C. GBN maintains a separate timer for each outstanding packet
D. SR maintains a separate timer for each outstanding packet
E. Neither GBN nor SR use NACKs

Q5 Go-back-N, selective repeat

- Suppose a receiver that has received all packets up to and including sequence number 24 and next receives packet 27 and 28. In response, what are the sequence numbers in the ACK(s) sent out by the GBN and SR receiver respectively?

A. [27,28], [28]
B. [24,24], [27,28]
C. [27,28], [27,28]
D. [25], [25]
E. [nothing], [27, 28]

Q6 Go-back-N

- Consider a GBN protocol with a sender window of 6 and a large sequence # space. Suppose the next in-order sequence number the receiver is expecting is **M**. At this time instant, which of the following sequence #'s can *not* be part of the sender's window? Assume no reordering.

A. M
B. M+1
C. M+5
D. M-6
E. M-7

Go-Back-N: sender

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

- ACK(n): ACKs all pkts up to, including # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)

Q7 Go-back-N

- Consider a GBN protocol with a sender window of 6 and a large sequence # space. Suppose the next in-order sequence number the receiver is expecting is **M**. At this instant, which of the following can *not* be the sequence # in an in-flight ACK from the receiver? Assume no reordering.

A. M-1
B. M-6
C. M-7
D. M-8
E. M-11

Go-Back-N: sender

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

- ACK(n): ACKs all pkts up to, including # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)

3. Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

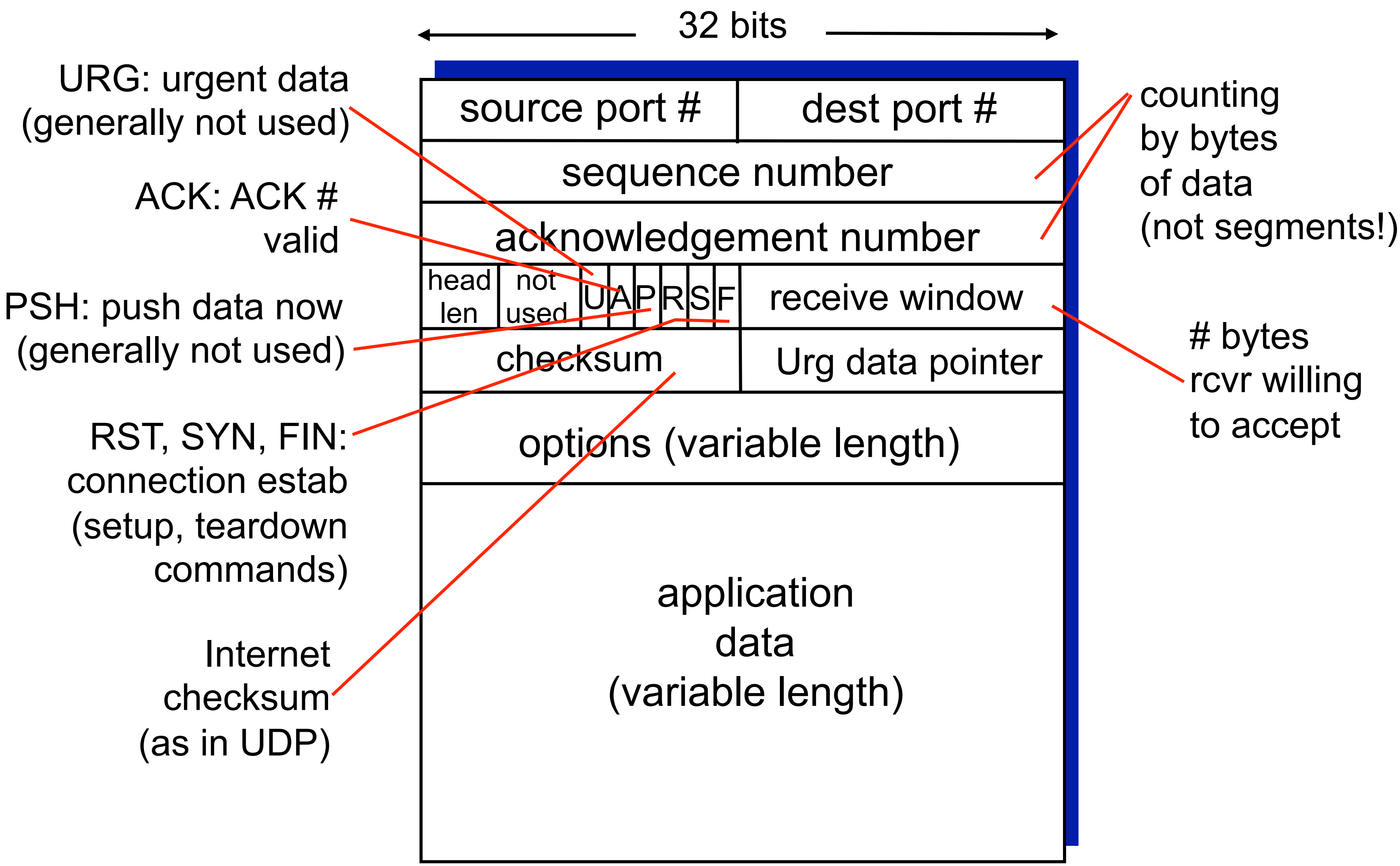
3.6 principles of congestion control

3.7 TCP congestion control

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure

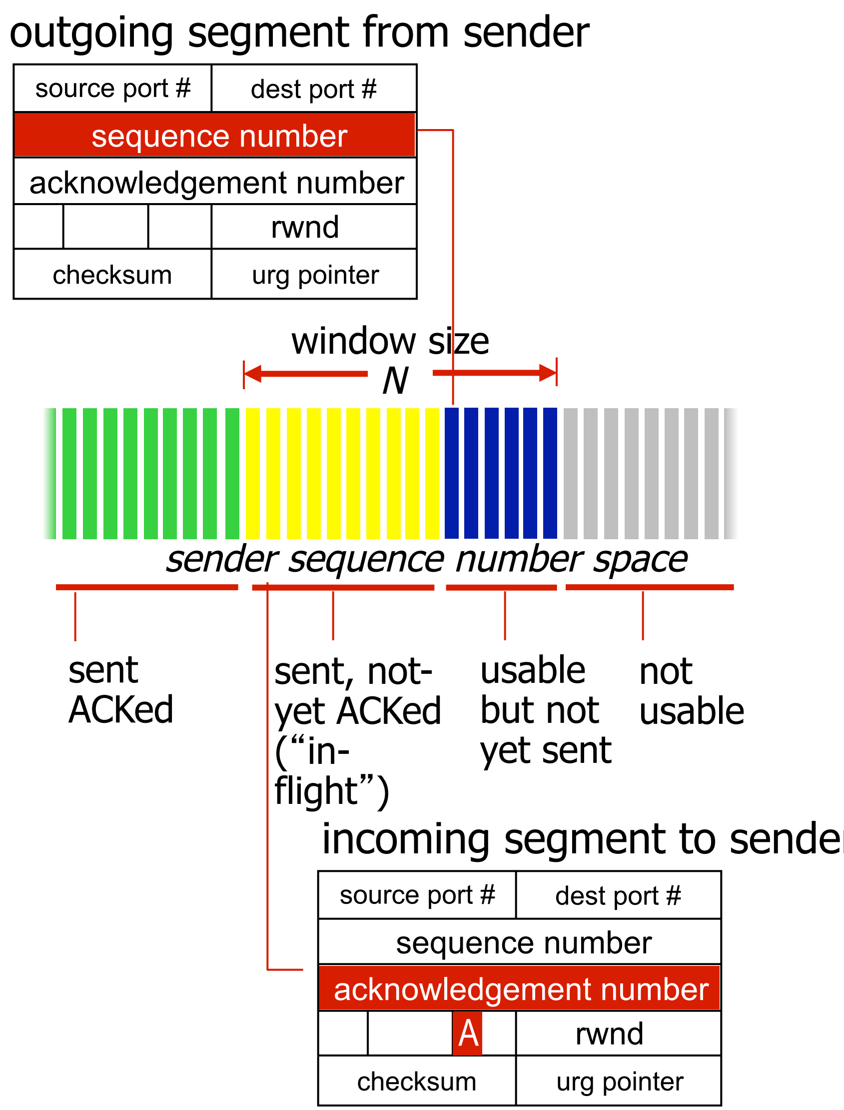


TCP seq. numbers, ACKs

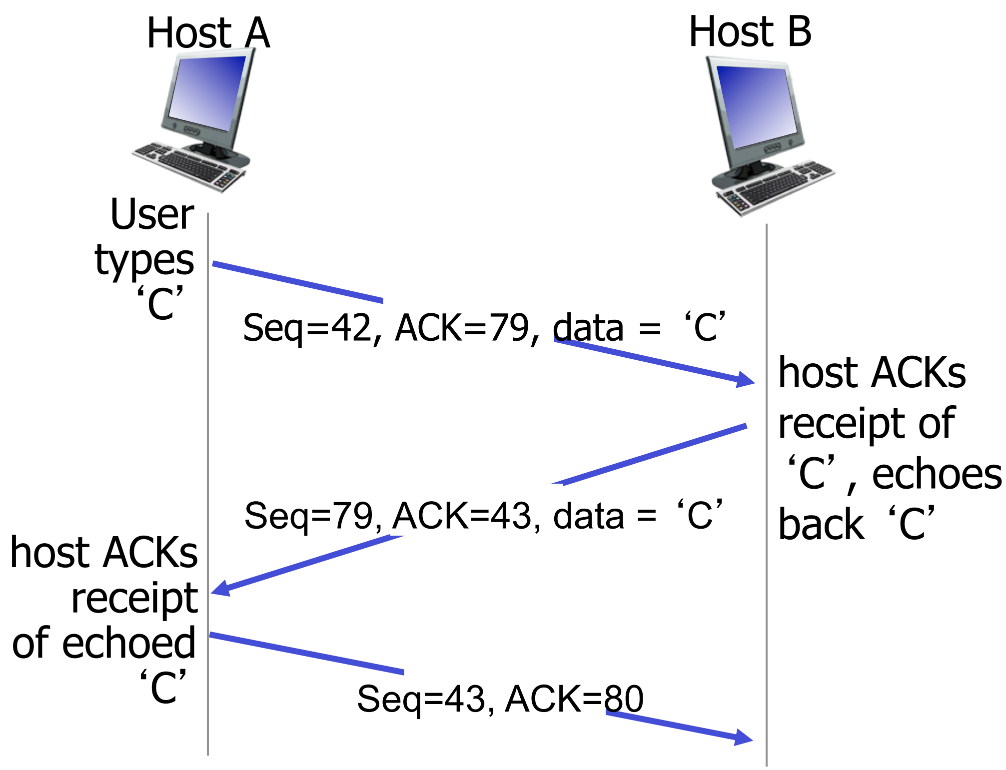
sequence number:
- byte stream # of first byte in segment's data

acknowledgement number:
- 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 character echo application

TCP round trip time, timeout

Q: how to set TCP timeout value?

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

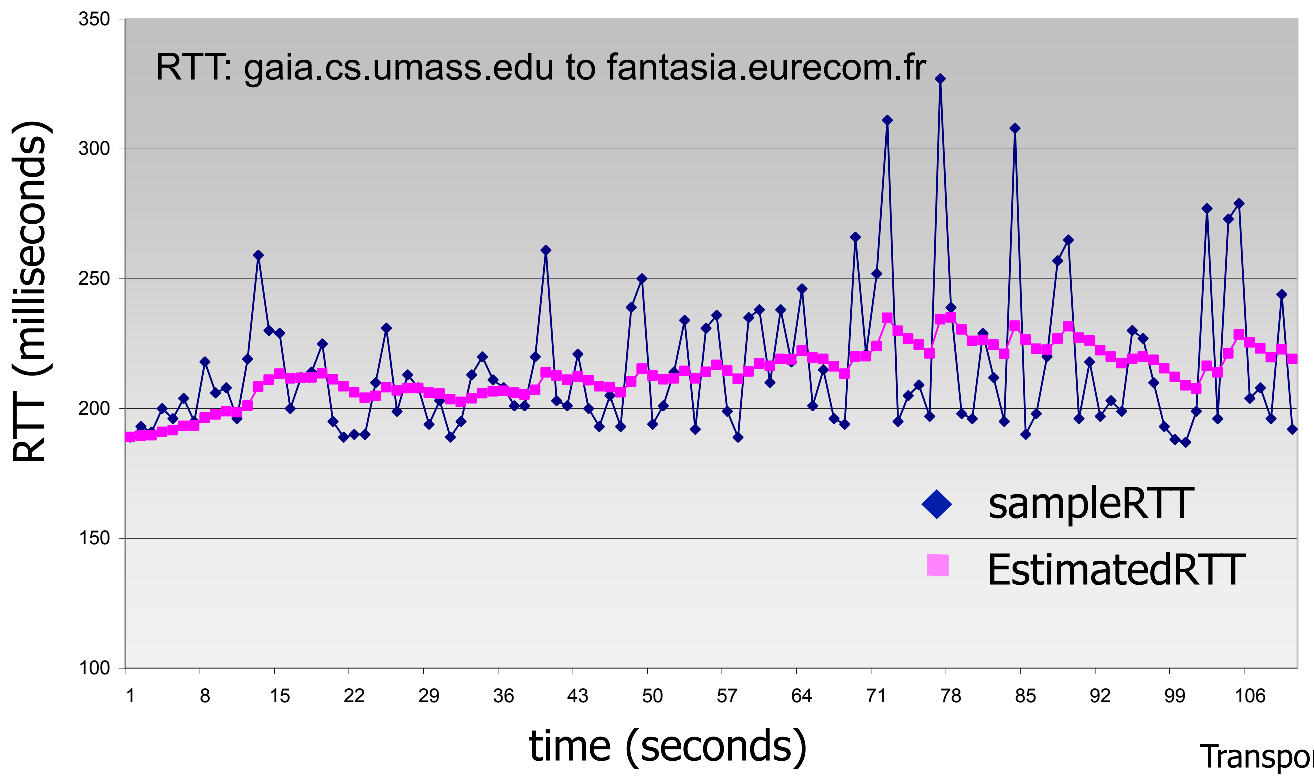
Q: how to estimate RTT?

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

TCP round trip time, timeout

$$\texttt{SmoothedRTT}_i = (1-\alpha)*\texttt{SmoothedRTT}_{i-1} + \alpha*\texttt{SampleRTT}_i$$

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



TCP round trip time, timeout

- timeout interval: **`SmoothedRTT`** plus "safety margin"
  - large variation in **`SmootedRTT`** ➔ larger safety margin
- estimate SampleRTT deviation from SmoothedRTT:

$$\mathtt{DevRTT}_i = (1-\beta)*\mathtt{DevRTT}_{i-1} + \beta*|\mathtt{SampleRTT}_i-\mathtt{SmoothedRTT}_i|$$

(typically, β = 0.25)

$$\mathtt{TimeoutInterval} = \underbrace{\mathtt{SmoothedRTT}}_{\text{"average RTT"}} + \underbrace{4*\mathtt{DevRTT}}_{\text{"safety margin"}}$$

3. Transport Layer: 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
    - selective acks often supported as an option
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

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

TCP sender events:

data rcvd from app:

- create segment with seq # (= byte-stream number of first data byte in segment)
- start timer (for oldest unacked segment) if not already running
  - `TimeOutInterval` = smoothed_RTT + 4*deviation_RTT

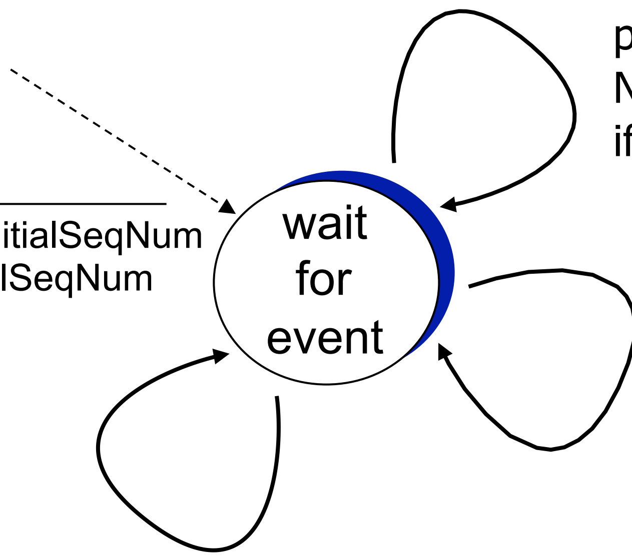
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

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

TCP sender (simplified)



data received from application above

---

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

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

timeout

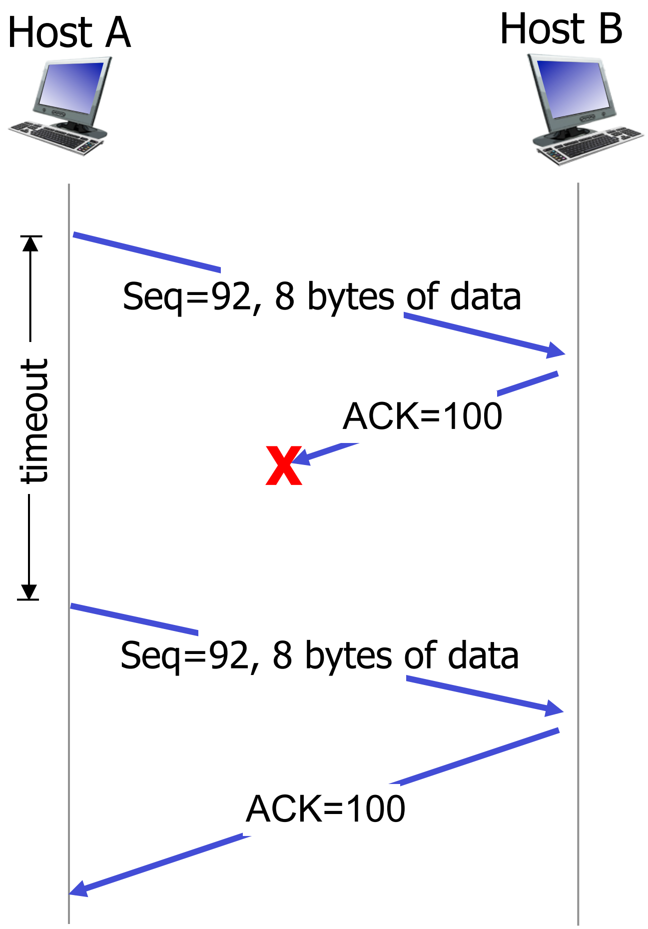
---

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

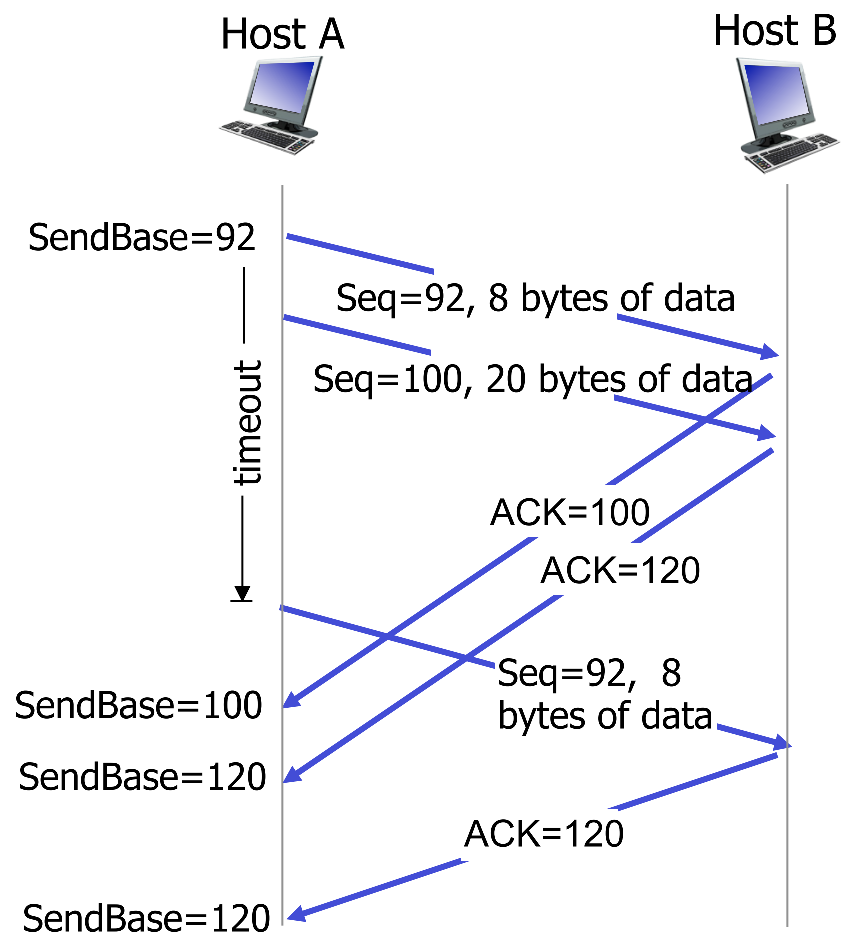
ACK received, with ACK field value y

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

TCP: retransmission scenarios

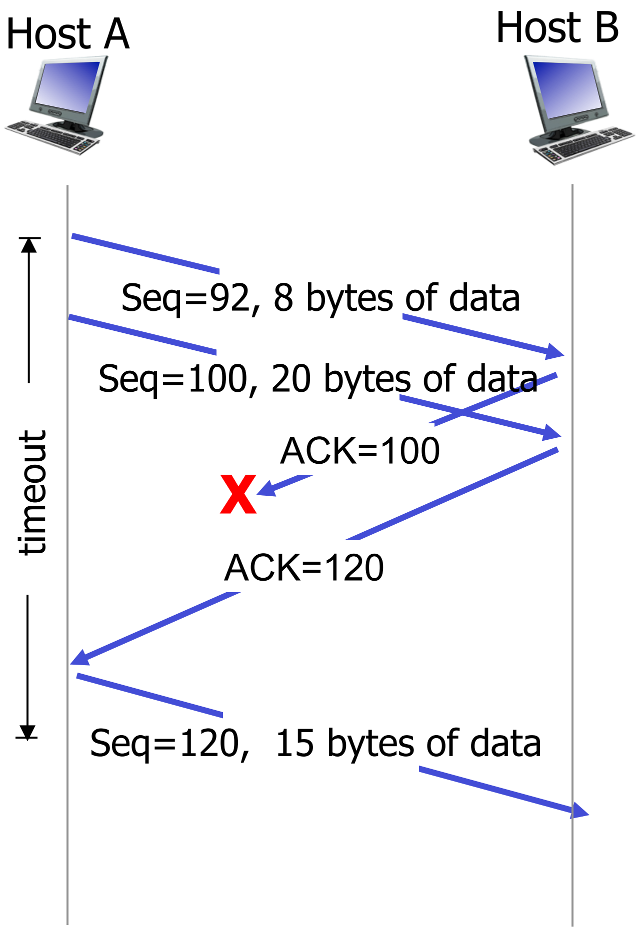


lost ACK scenario



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	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

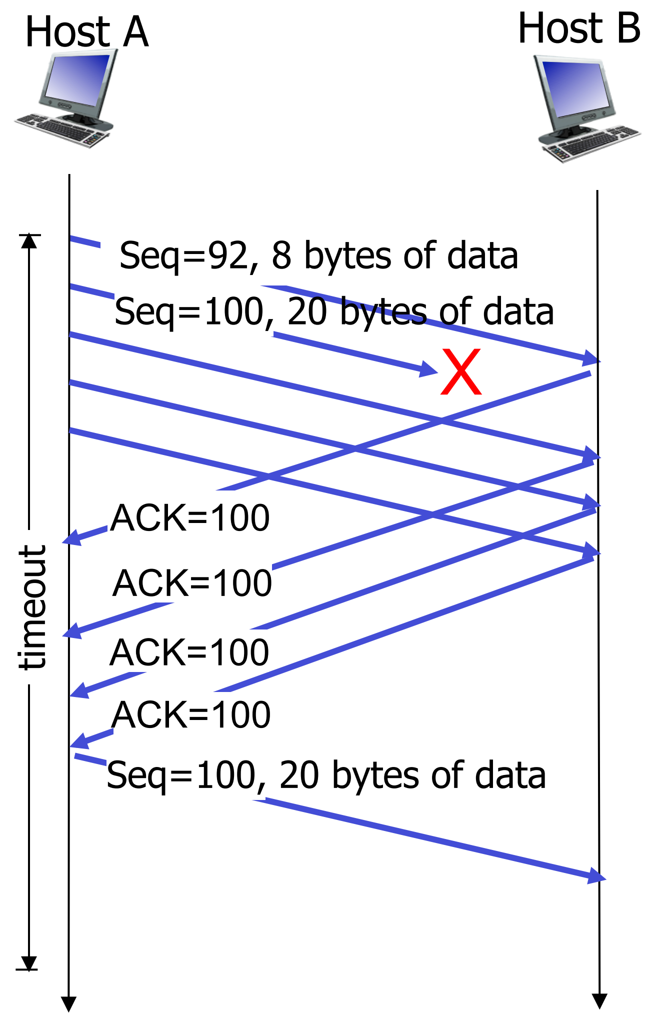
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sending many segments back-to-back plus occasional segment loss ➔ duplicate ACKs

TCP fast retransmit

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

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

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

3. Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

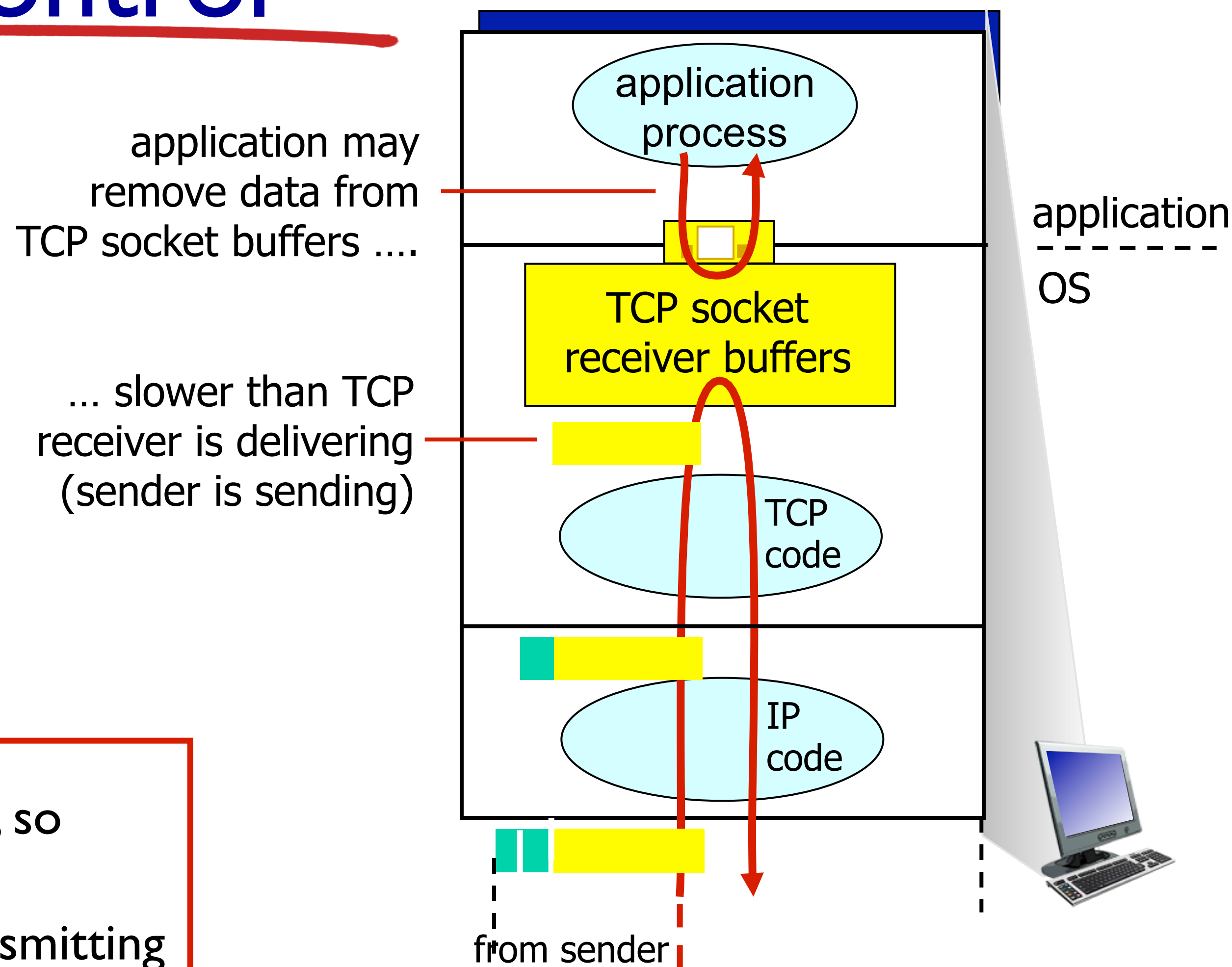
3.4 principles of reliable data transfer

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



receiver protocol stack

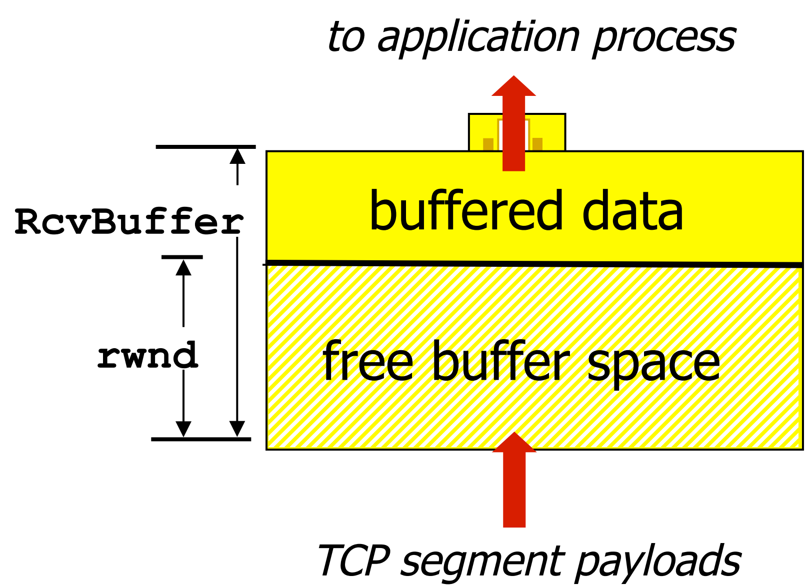
flow control

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 rcvr-to-sndr segments
  - **RcvBuffer** size can be set via socket options
  - most operating systems auto-adjust **RcvBuffer**

- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value to ensure receive buffer will not overflow



receiver-side buffering

3. Transport Layer: 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

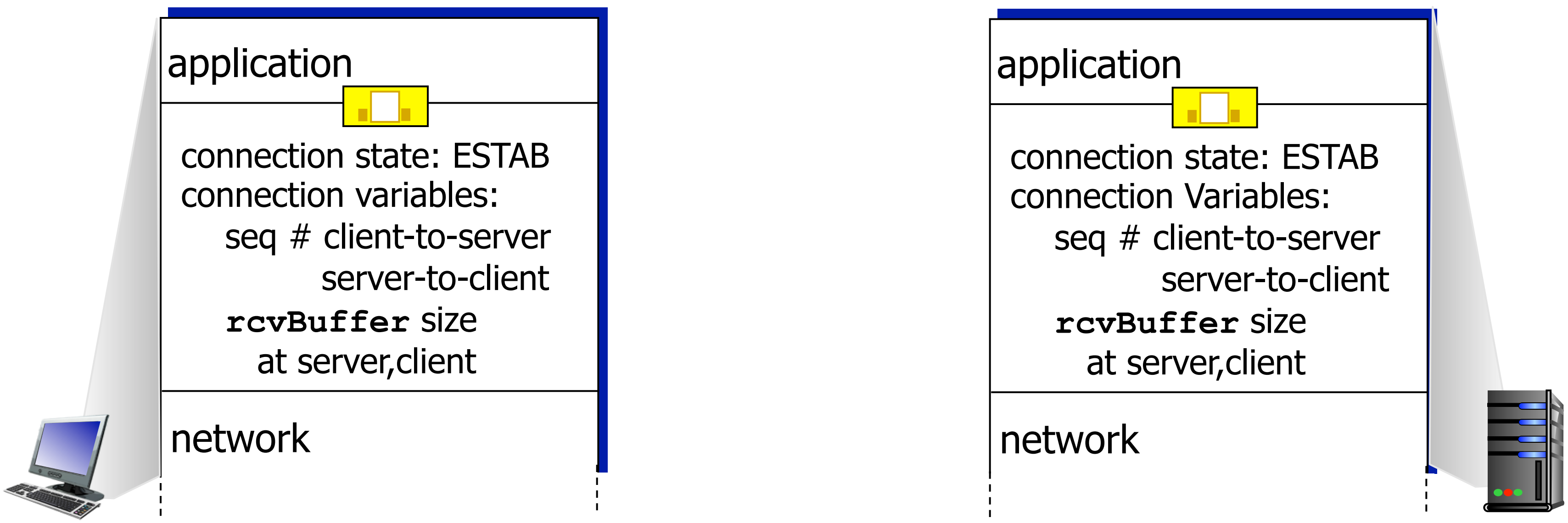
3.6 principles of congestion control

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

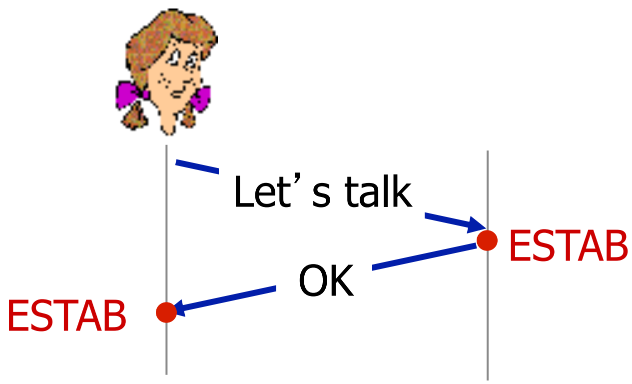


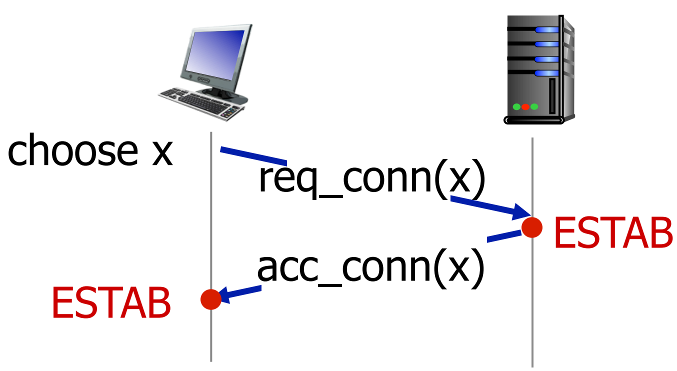

```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

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

Agreeing to establish a connection

2-way handshake:



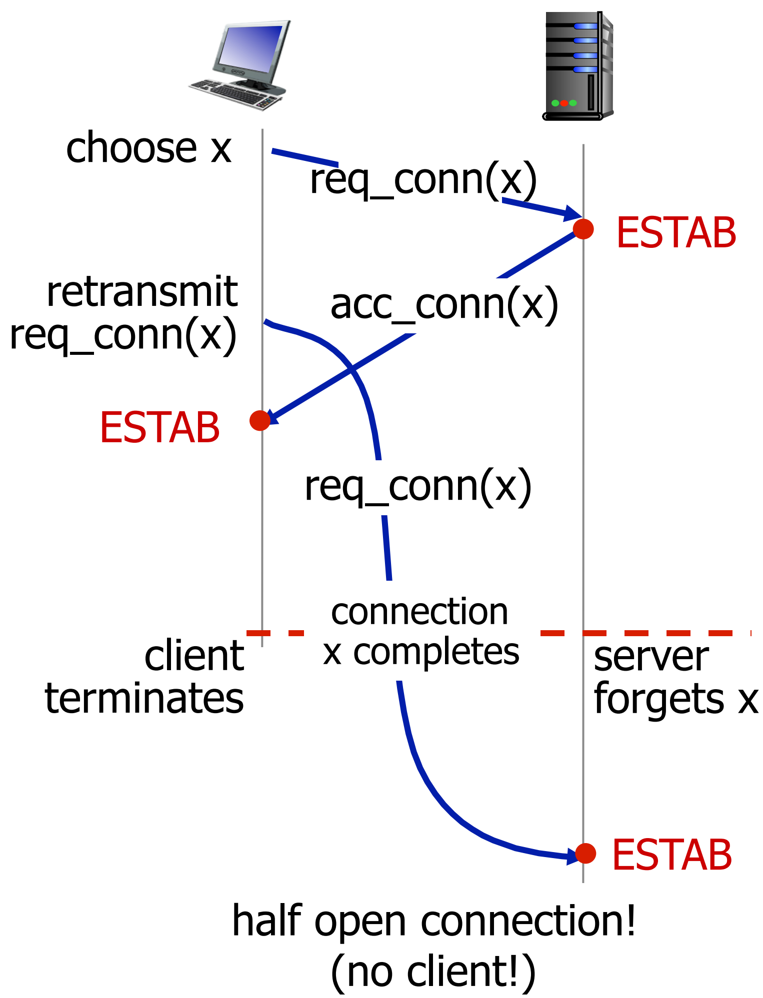


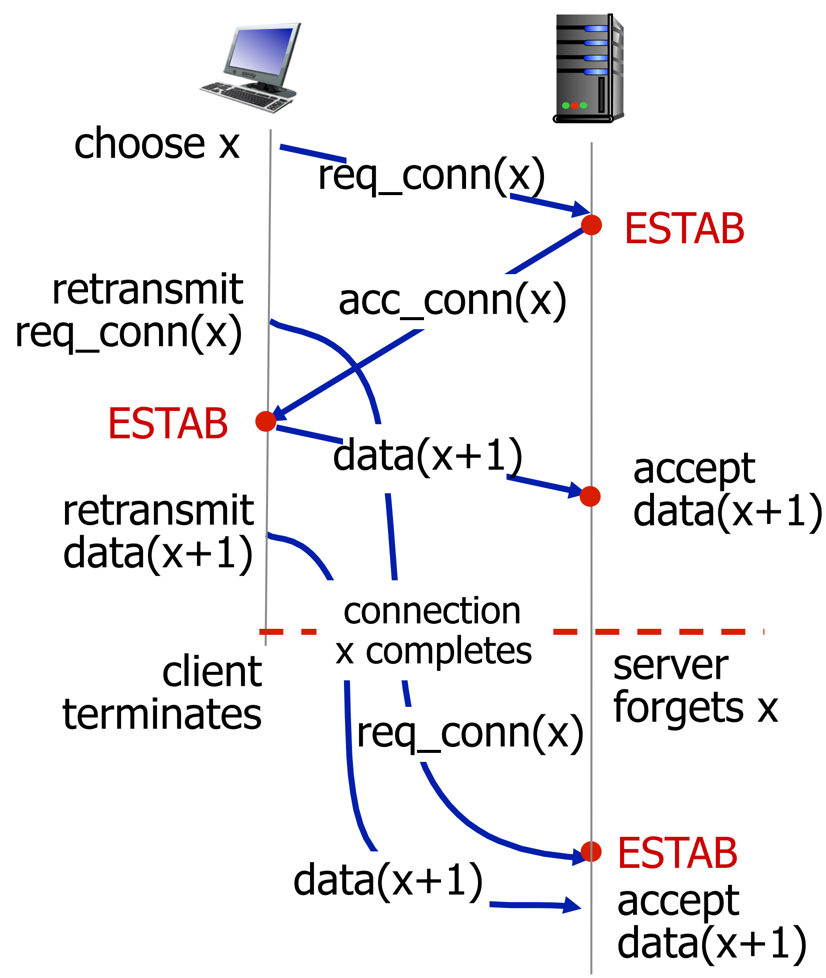
Q: will 2-way handshake always work in network?

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

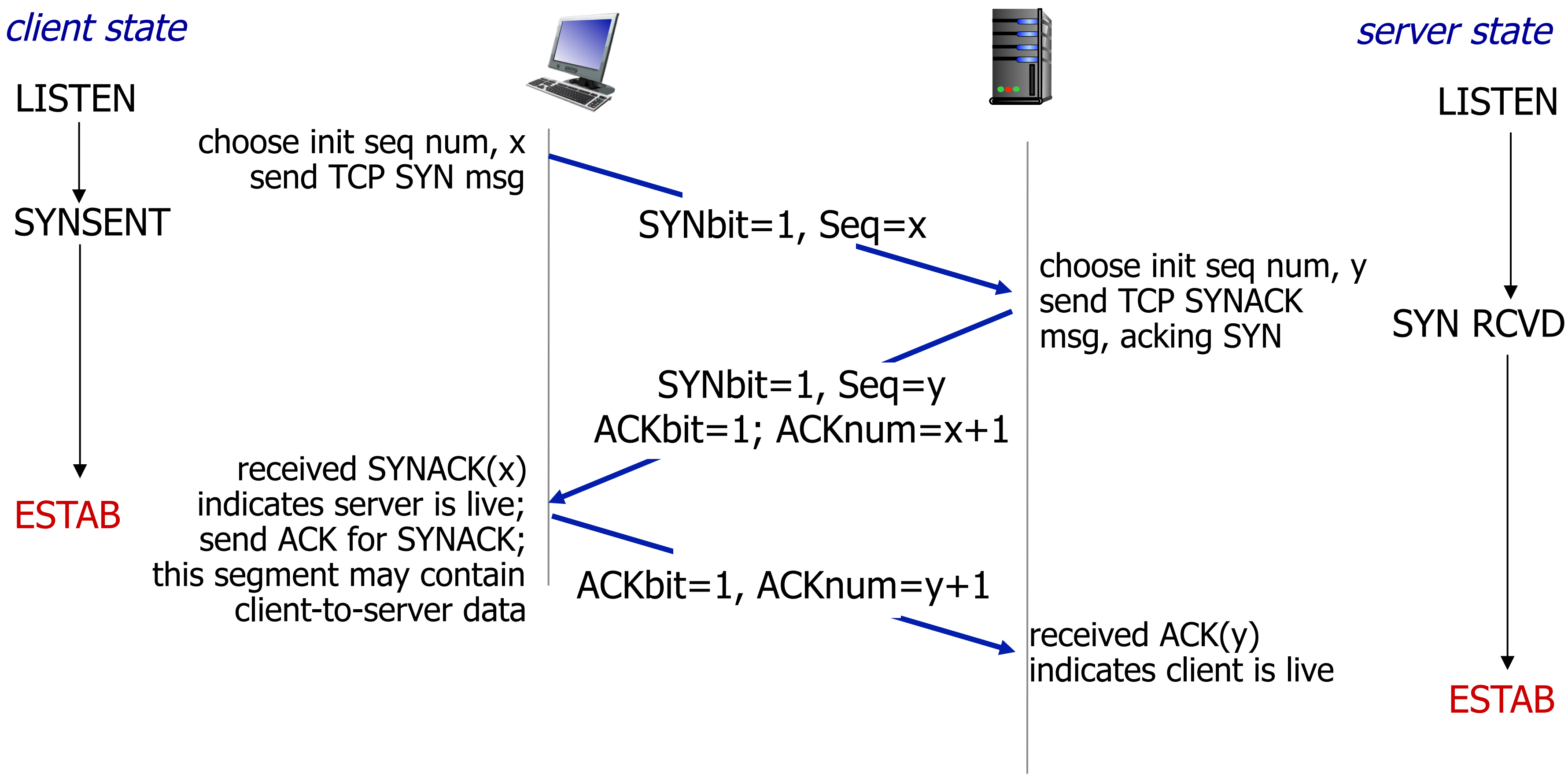
Agreeing to establish a connection

2-way handshake failure scenarios:

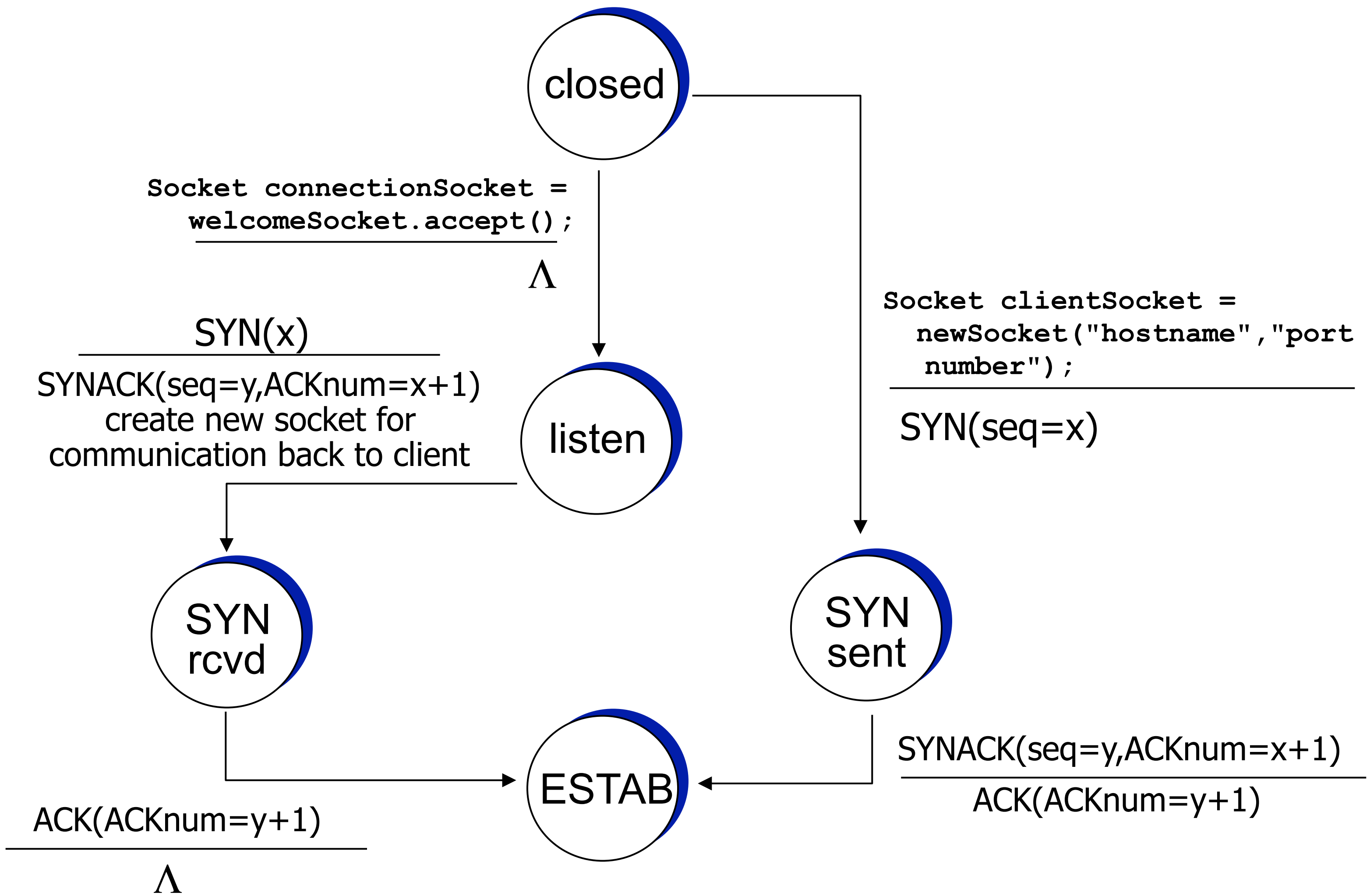




TCP 3-way handshake



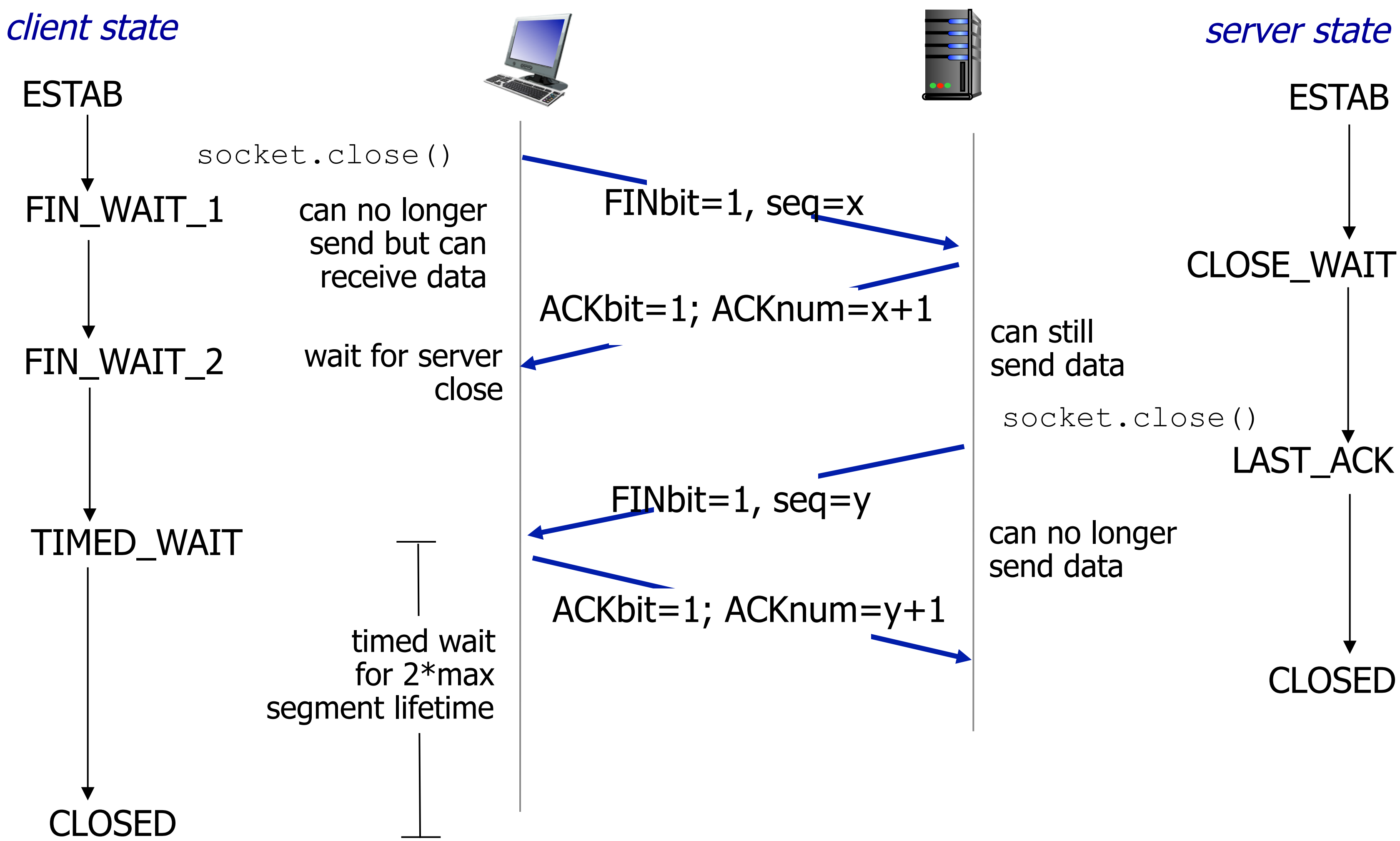
TCP 3-way handshake: FSM



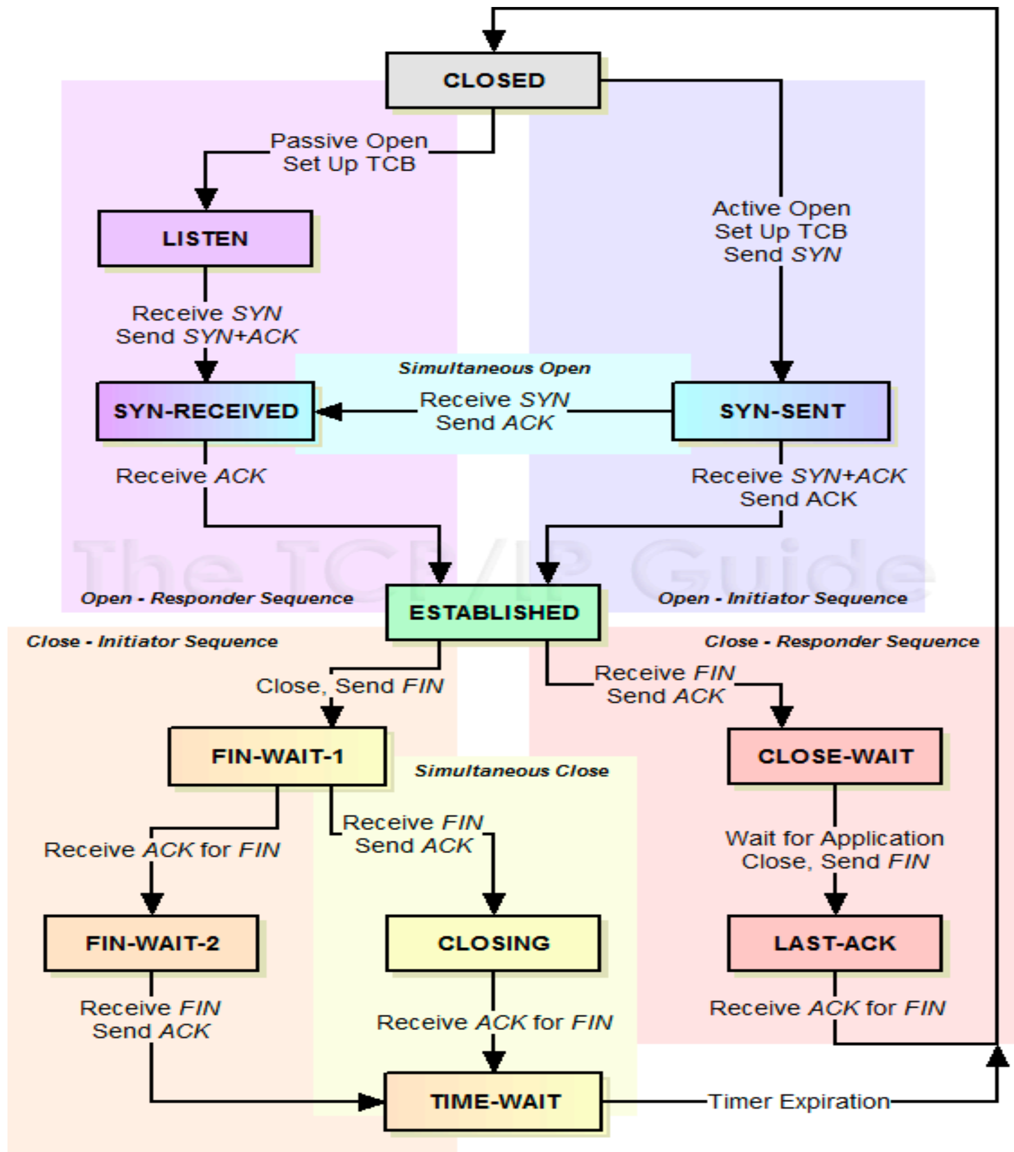
TCP: closing a connection

1. client and server should each close their side of connection
   - by sending FIN (TCP segment with FIN flag = 1)
2. should respond to received FIN with ACK
   - on receiving FIN, ACK can be combined with own FIN
3. simultaneous FIN exchanges should be handled

TCP: closing a connection



TCP: Overall state machine



Q1 TCP sequence numbers

- A TCP sender is just about to send a segment of size 100 bytes with sequence number 1234 and ack number 436 in the TCP header. What is the highest sequence number up to (and including) which this sender has received all bytes from the receiver?

  A. 1233
  B. 436
  C. 435
  D. 1334
  E. 536

Q2 TCP sequence numbers

- A TCP sender is just about to send a segment of size 100 bytes with sequence number 1234 and ack number 436 in the TCP header. Is it possible that the receiver has received byte number 1335?
  1. Yes
  2. No

Q3 TCP timeout

- A TCP sender maintains a SmoothedRTT of 100ms. Suppose the next SampleRTT is 108ms. Which of the following is true of the sender?
  1. Will increase SmoothedRTT but leave the timeout unchanged
  2. Will increase timeout
  3. Whether it increases SmoothedRTT depends on the deviation.
  4. Whether it increases the timeout depends on the deviation
  5. Will chomp on fries left over from the rdt question earlier

Q4 TCP timeout

- A TCP sender maintains a SmoothedRTT of 100ms and DevRTT of 8ms. Suppose the next SampleRTT is 108ms. What is the new value of the timeout in milliseconds? (Numerical question)

Q5 TCP header fields

- Which is the purpose of the receive window field in a TCP header?
    A. Reliability
    B. In-order delivery
    C. Flow control
    D. Congestion control
    E. Pipelining

Q6 TCP connection mgmt

- Roughly how much time does it take for both the TCP sender and receiver to establish connection state since the connect() call?
  A. RTT
  B. 1.5RTT
  C. 2RTT
  D. 3RTT

Q7 TCP reliability

- TCP uses cumulative ACKs like Go-back-N, but does not retransmit the entire window of outstanding packets upon a timeout. What mechanism TCP get away with this?
  A. Per-byte sequence and ack numbers
  B. Triple duplicate ACKs
  C. Receive window-based flow control
  D. Using a better timeout estimation method
  E. Ketchup (for the fries)

3. Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

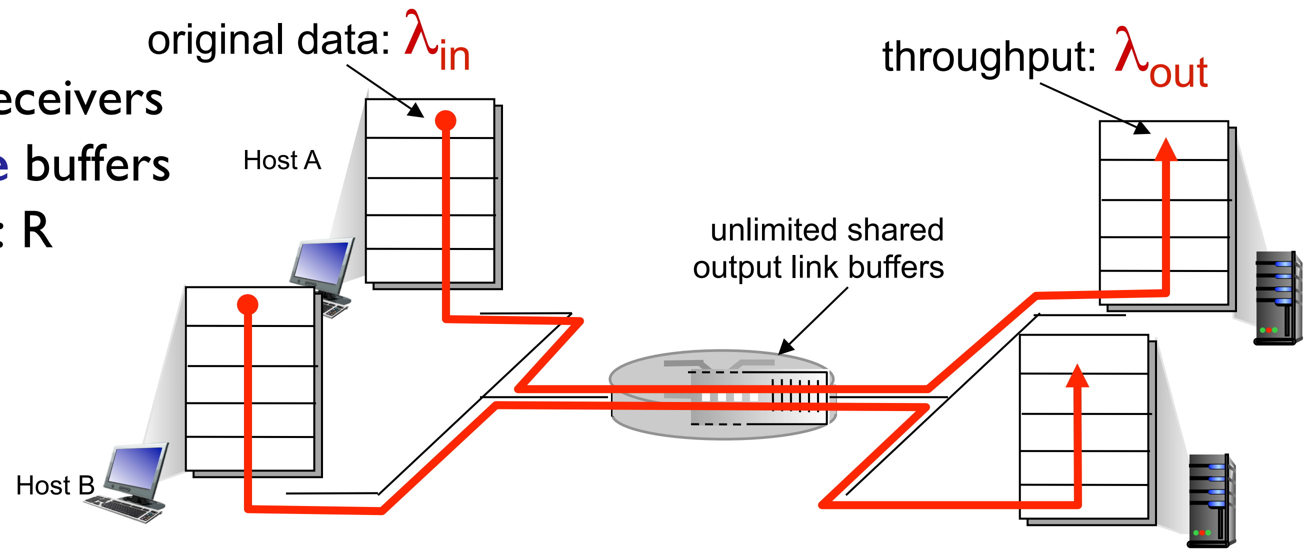
Principles of congestion control

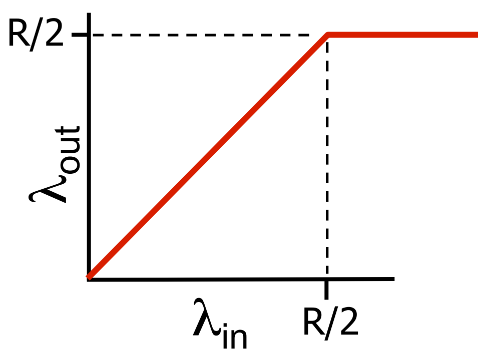
congestion:

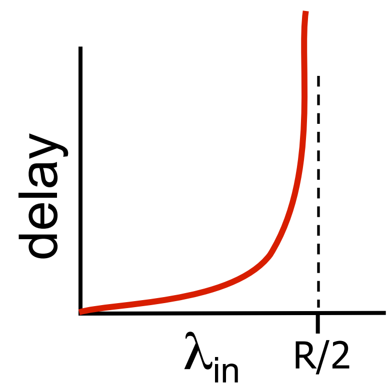
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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



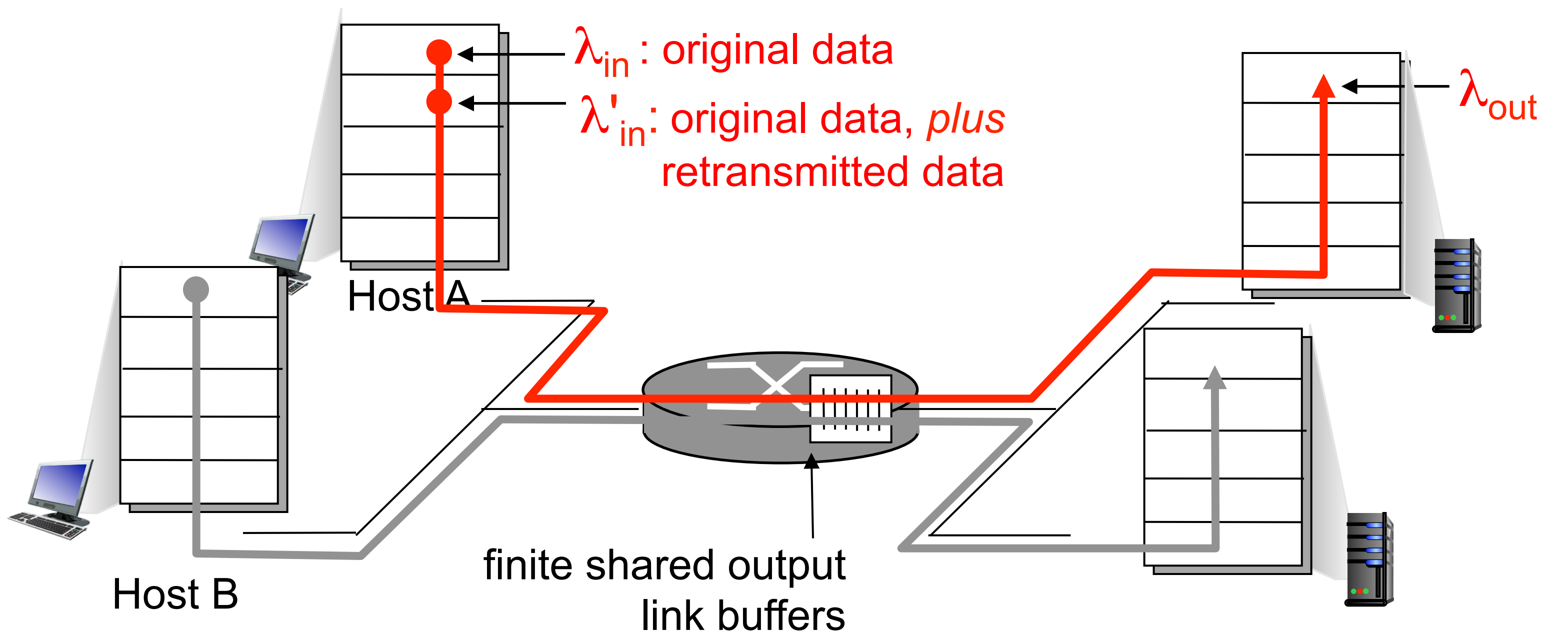




- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, approaches capacity

Causes/costs of congestion: scenario 2

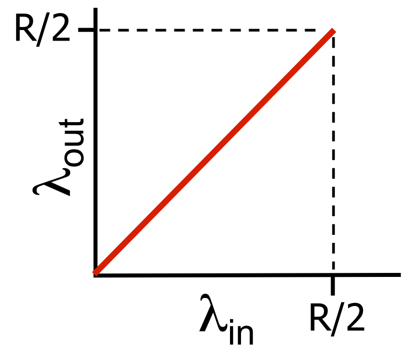
- one router, ***finite*** buffers
- sender retransmission of timed-out packet
  - app-layer input = app-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

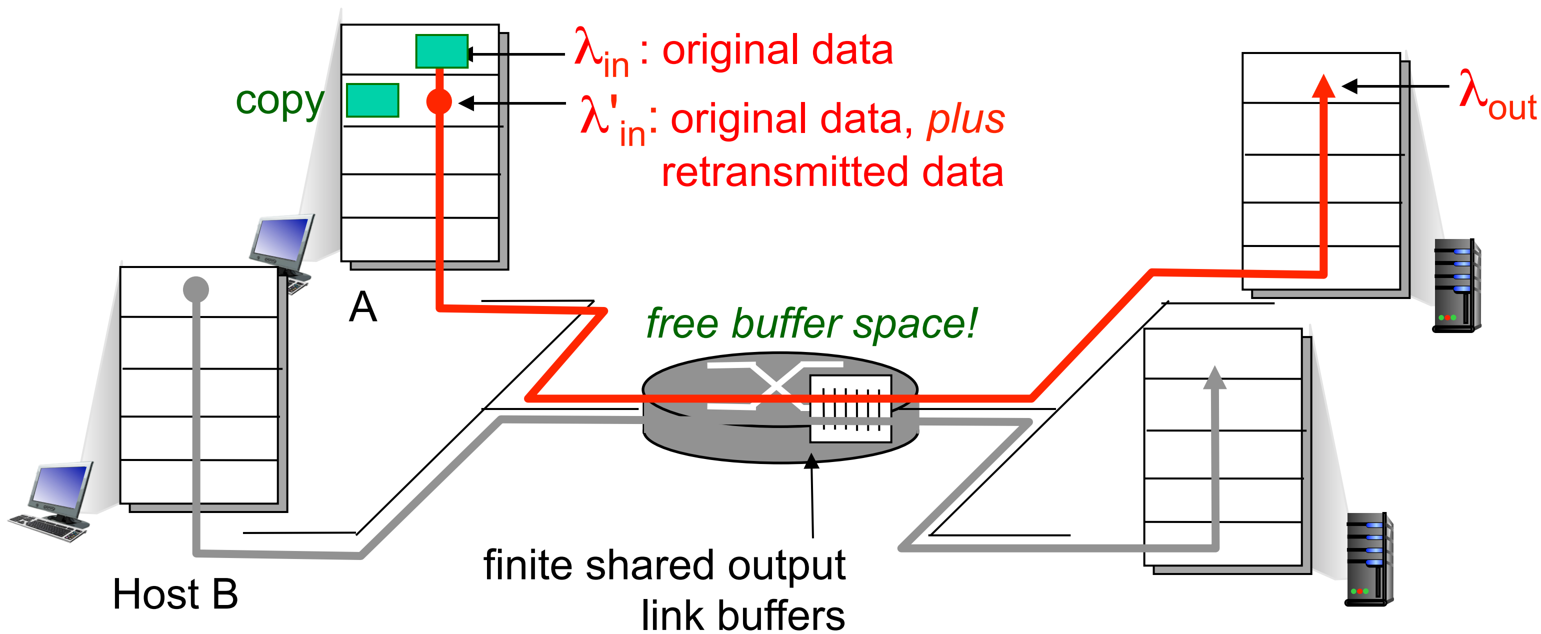


Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

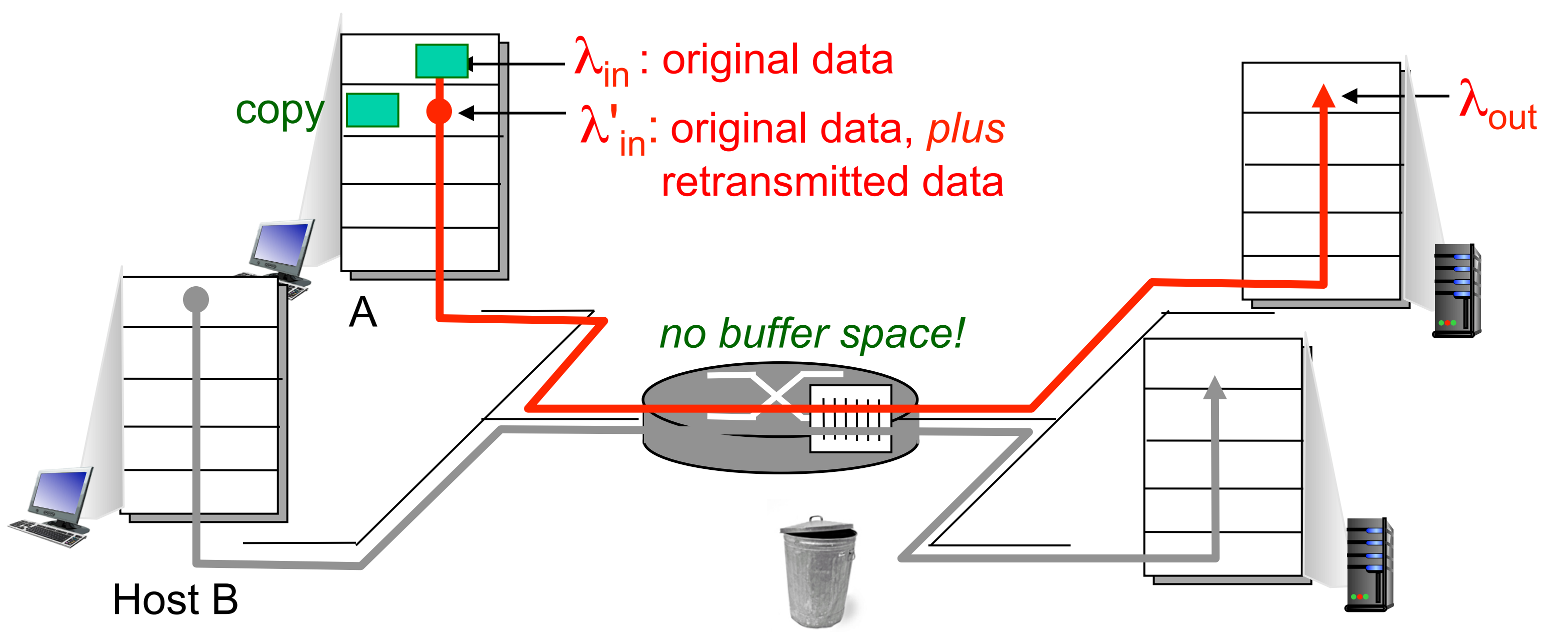




Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost at router with full buffer

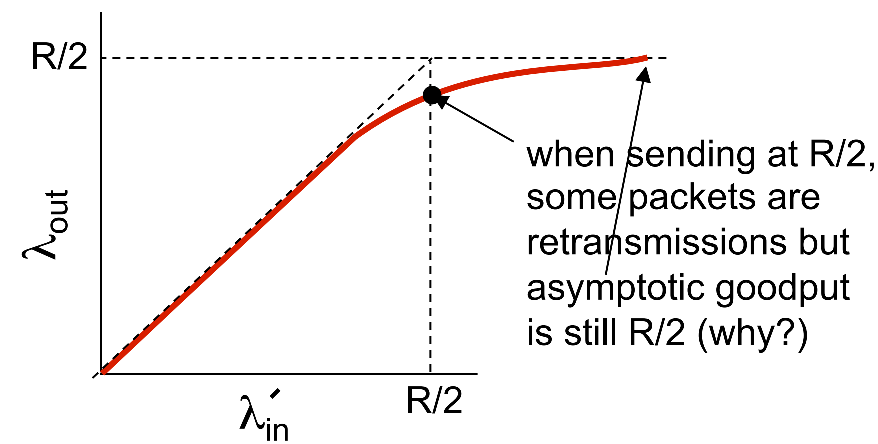
- sender only resends if packet *known* to be lost

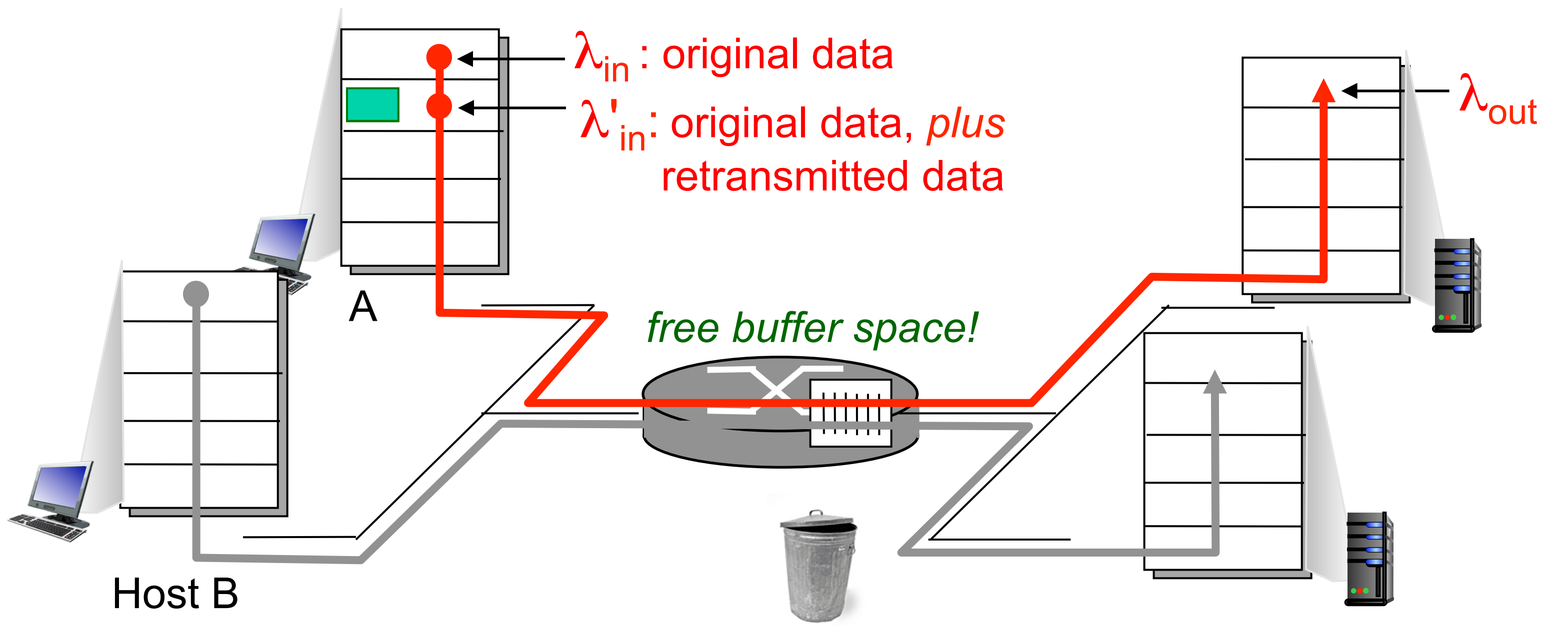


Causes/costs of congestion: scenario 2

Idealization: known loss
packets can be lost at router with full buffer

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

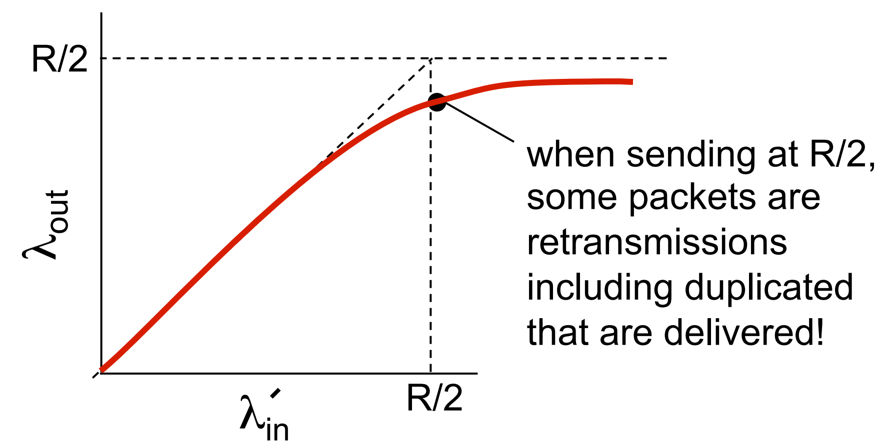


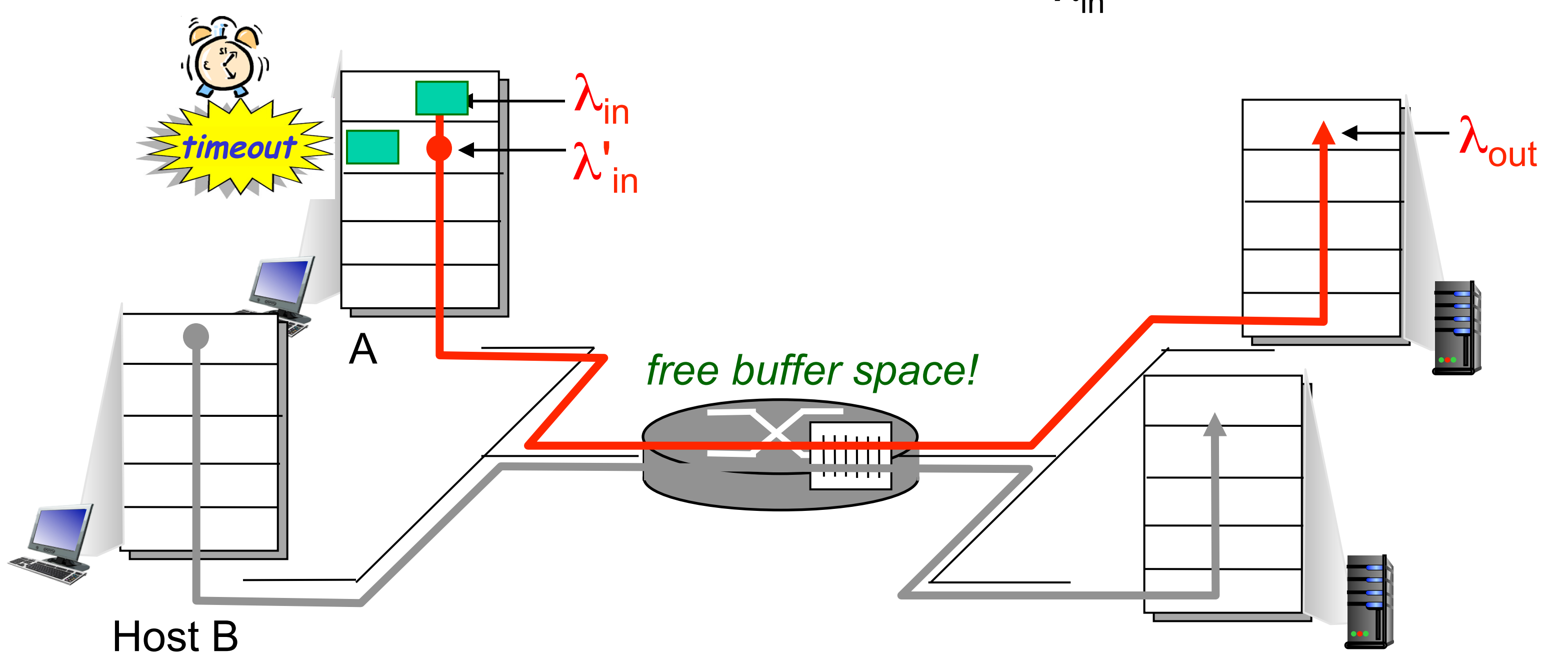


Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost at routers with full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

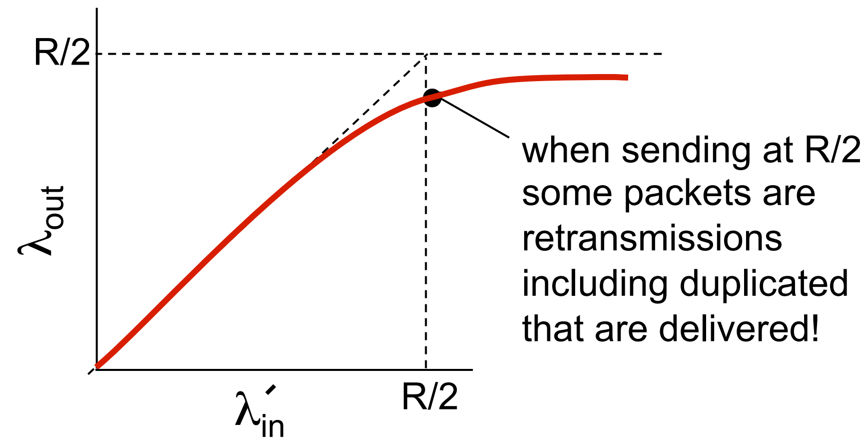




Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost at routers with full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

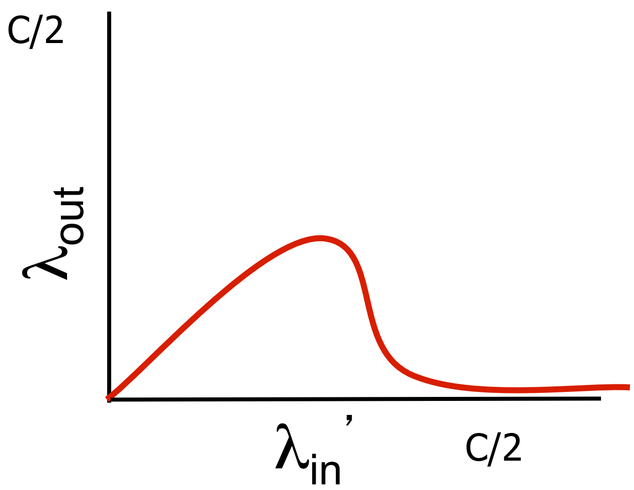


"costs" of congestion:

- more work for same "goodput"
- unnecessary retransmission (link carries multiple copies of packet) decreases goodput

Causes/costs of congestion: scenario 3

- Congestion collapse: dramatic reduction in throughput (how?)



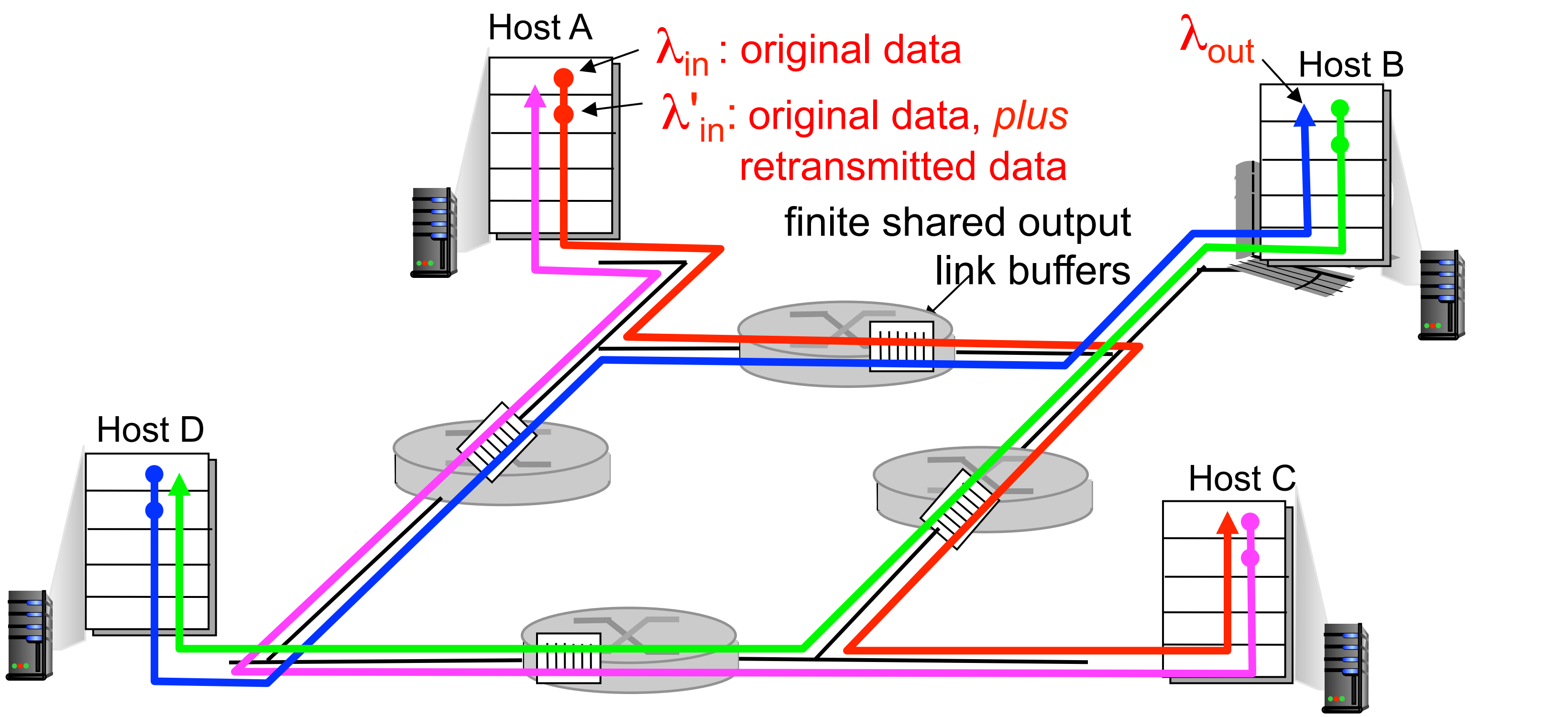
History: In the late 80s, we learned this lesson the hard way.

Causes/costs of congestion: scenario 3

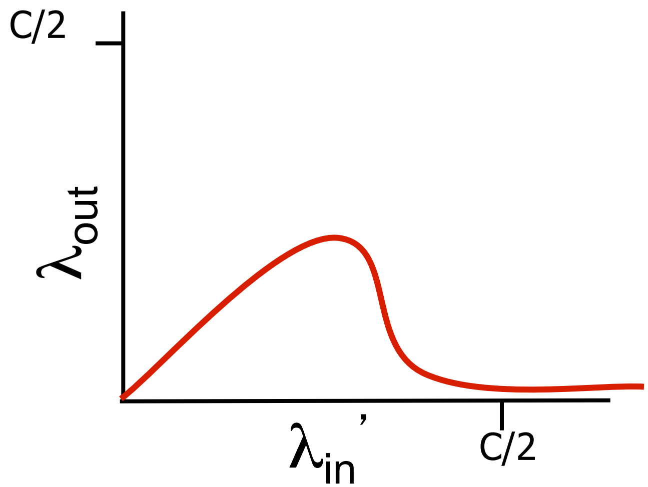
- four senders
- multihop paths
- timeout/retransmit

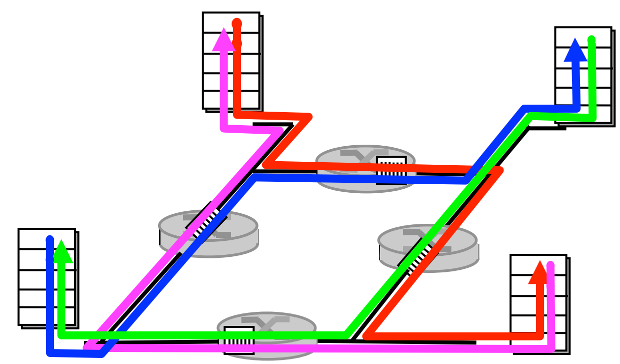
Q: what happens as λ_{in} and λ'_{in} increase ?

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





most important "cost" of congestion:

- when packet dropped, any upstream bandwidth used for that packet wasted.
- wastage can ripple into a "collapse"!

Approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:

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

network-assisted congestion control:

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

Case study: ATM ABR congestion control

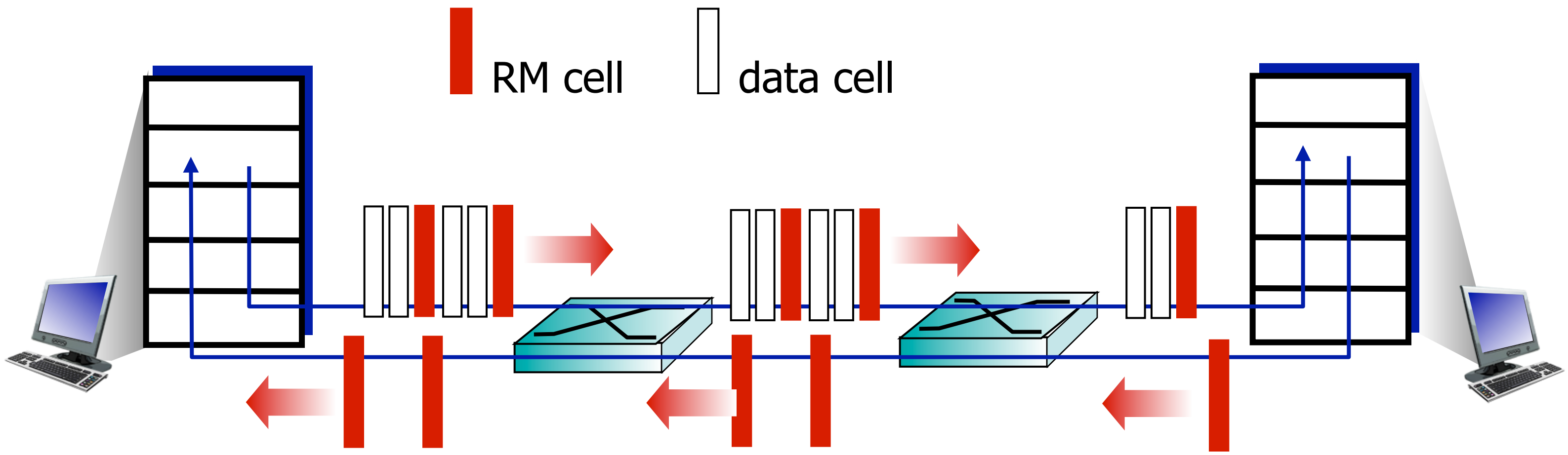
ABR: available bit rate:

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

RM (resource management) cells:

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

Case study: ATM ABR congestion control



- 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

3. Transport Layer: 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

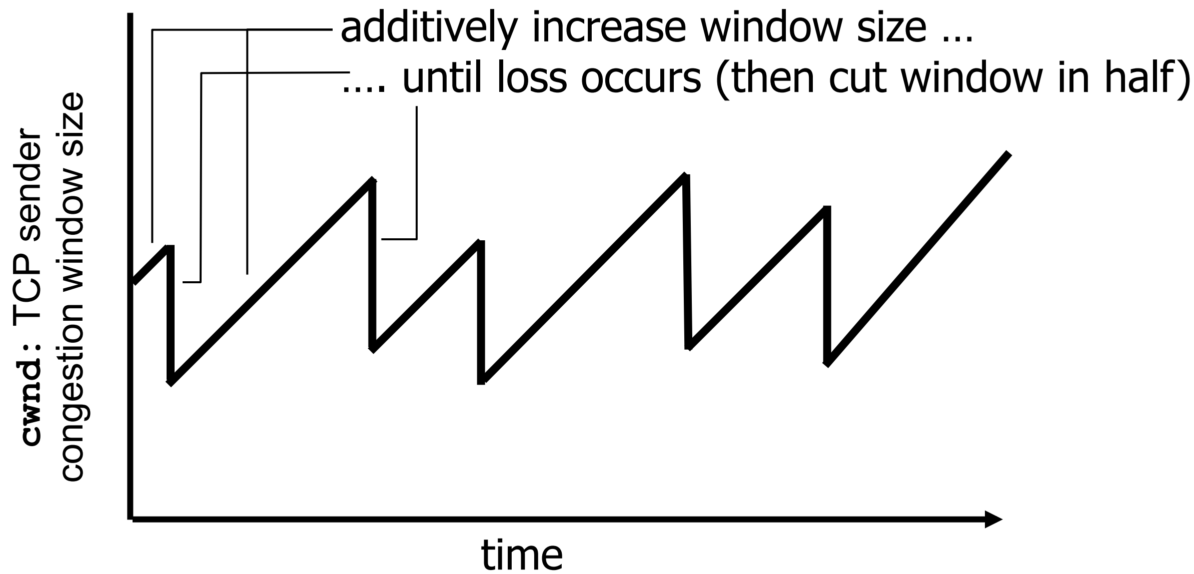
TCP congestion control

1. Congestion avoidance using AIMD
2. Slow start upon a timeout
3. Fast recovery to patch occasional loss

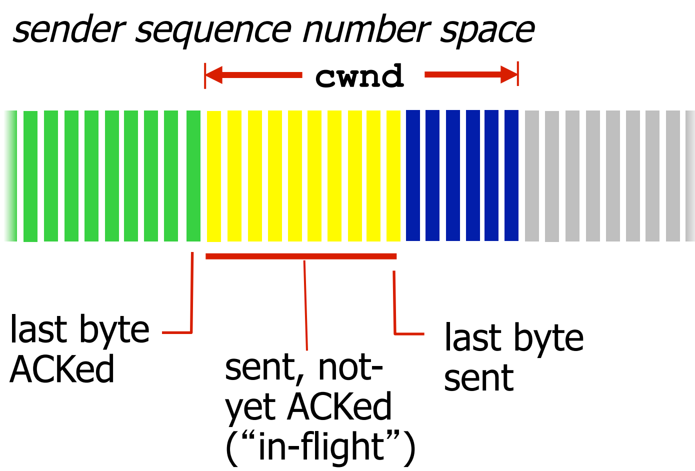
Congestion avoidance: AIMD

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

AIMD saw tooth behavior: probing for bandwidth



TCP congestion control window



- sender limits transmission:

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

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

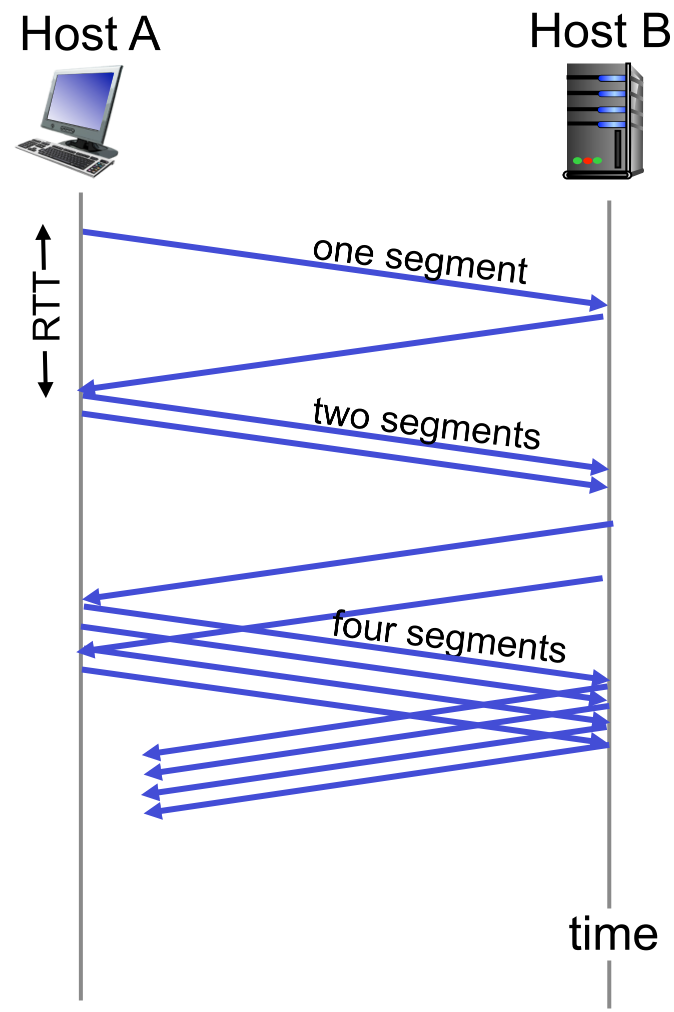
TCP sending rate:

- *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 until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** upon every ACK
- *summary:* initial rate is slow but ramps up exponentially fast



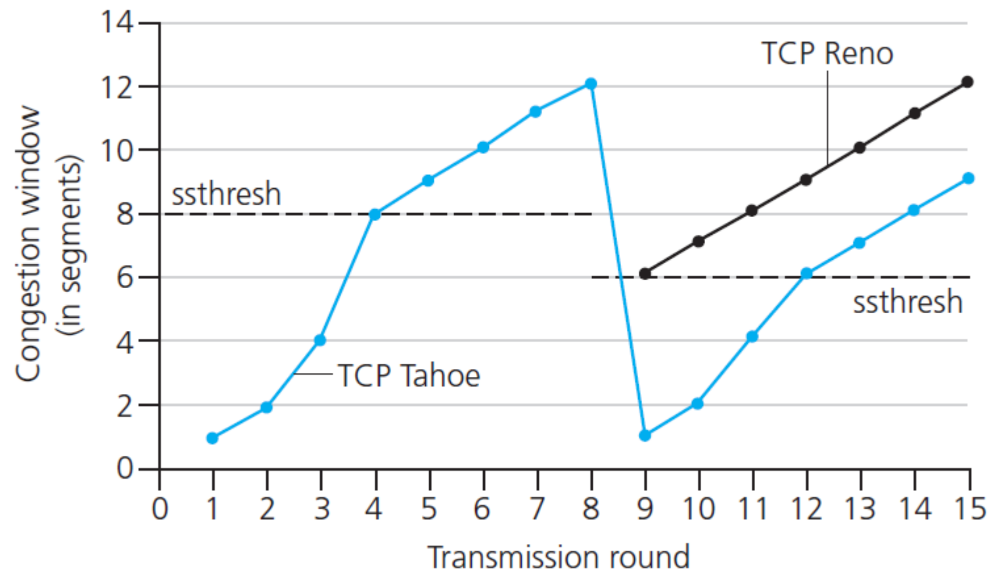
TCP: detecting, reacting to loss

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

TCP: slow start → cong. avoidance

Q: when should the exponential increase switch to linear?

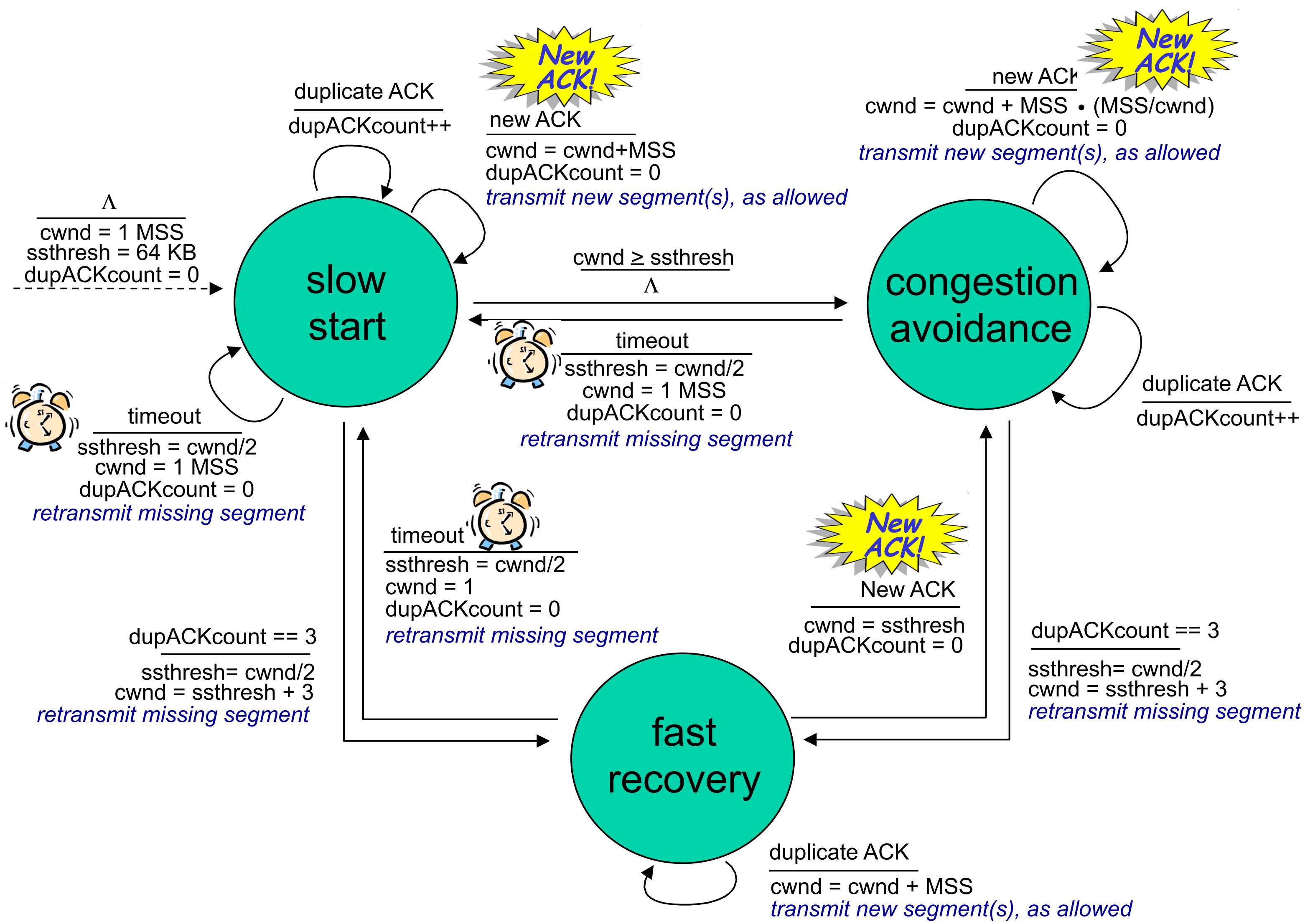
A: when **cwnd** gets to 1/2 of its value before timeout.



Implementation:

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

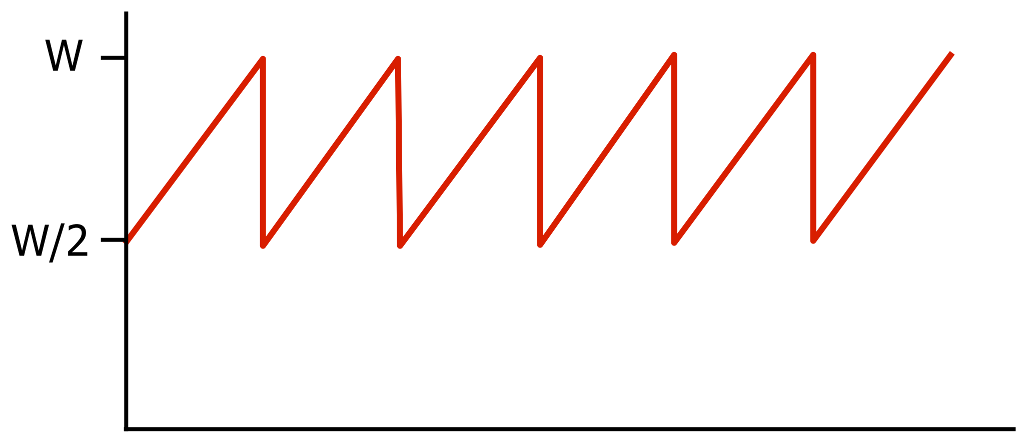
Summary: TCP Congestion Control



TCP throughput: Simplistic model

- 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. throughput is 3/4W per RTT

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



In practice, W not known or fixed, so this model is too simplistic to be useful

TCP throughput: More practical model

- Throughput in terms of segment loss probability, L, round-trip time T, and maximum segment size M [Mathis et al. 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot M}{T\sqrt{L}}$$

TCP futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments as per the throughput formula

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

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

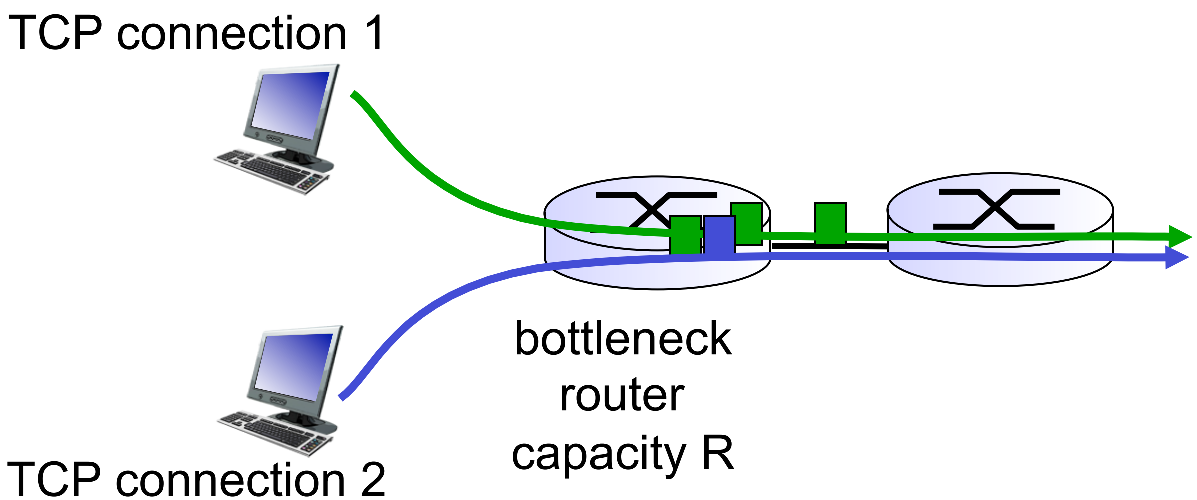
- new versions of TCP for high-speed

TCP throughput wrap-up

- Suppose
  - sender window `cwnd`,
  - receiver window `rwnd`
  - bottleneck capacity C
  - round-trip time T
  - path loss rate L
  - max segment size MSS
- Instantaneous TCP throughput =
  - min(C, `cwnd/T`, `rwnd/T`)
- Steady-state TCP throughput =
  - min(C, $1.22M/(T\sqrt{L})$)

TCP Fairness

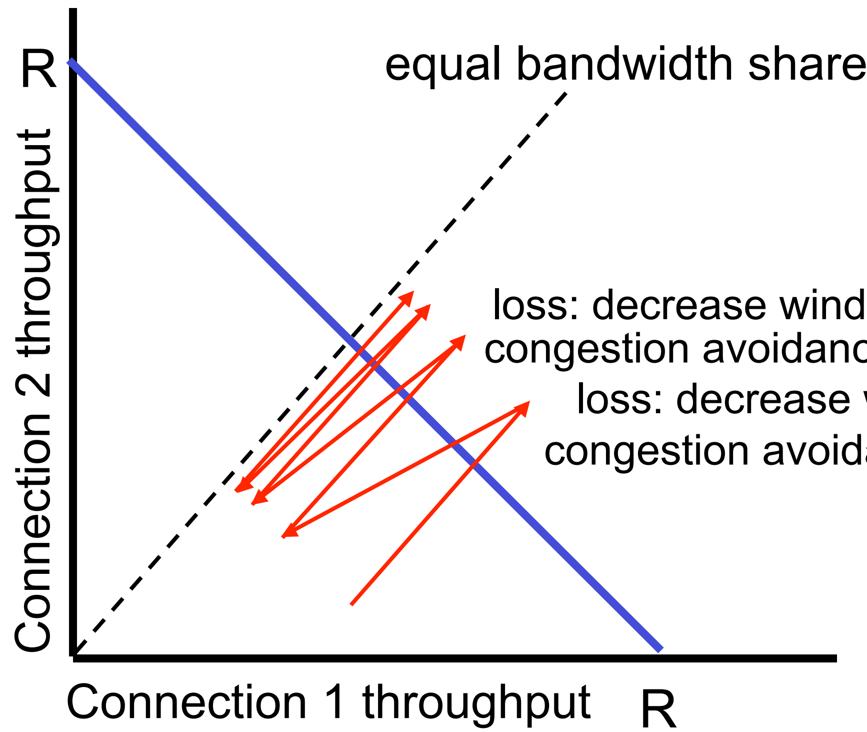
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions:

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



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - rate throttling by congestion control can hurt streaming quality
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open many 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 R/10
  - new app asks for 11 TCPs, gets R/2

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"