CS 3800: Computer Networks

Lecture 5: Transport Layer

Instructor: John Korah

Acknowledgement

- The following slides include material from author resources for:
  - KR Text book
  - "Data and computer communications," William Stallings, Tenth edition

Learning Goals

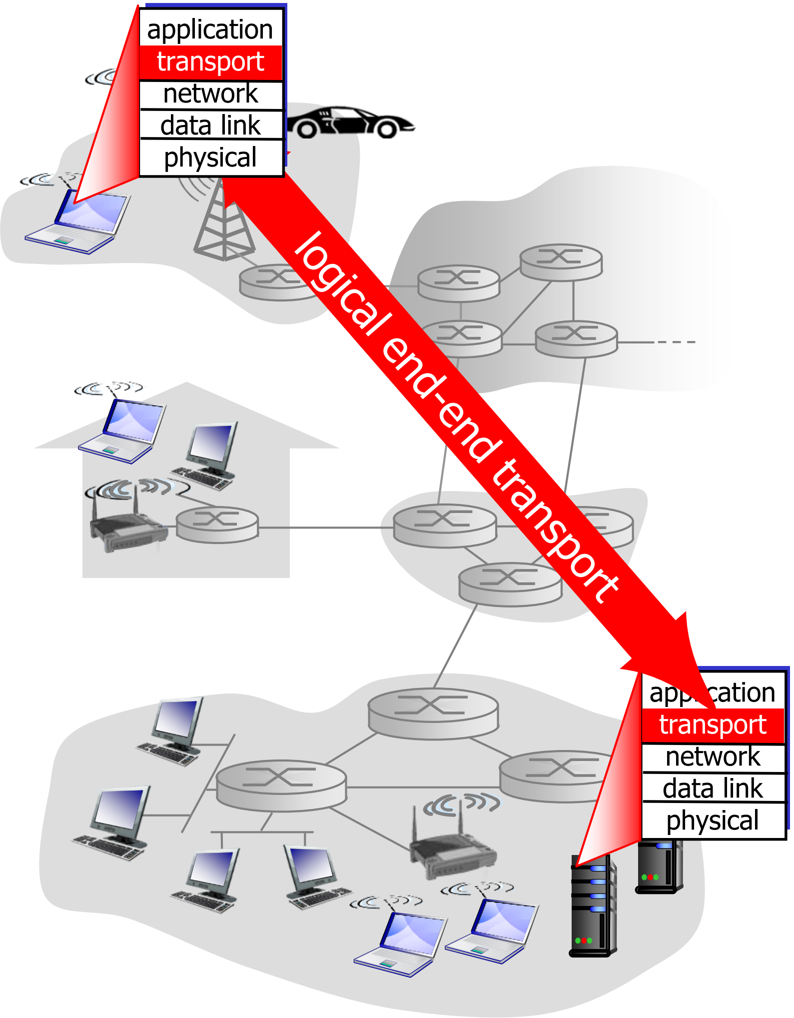
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

Topics

- Transport-layer services
- Multiplexing and demultiplexing
- UDP: Connectionless transport
- Principles of reliable data transfer
- TCP: Connection-oriented transport
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- 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:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

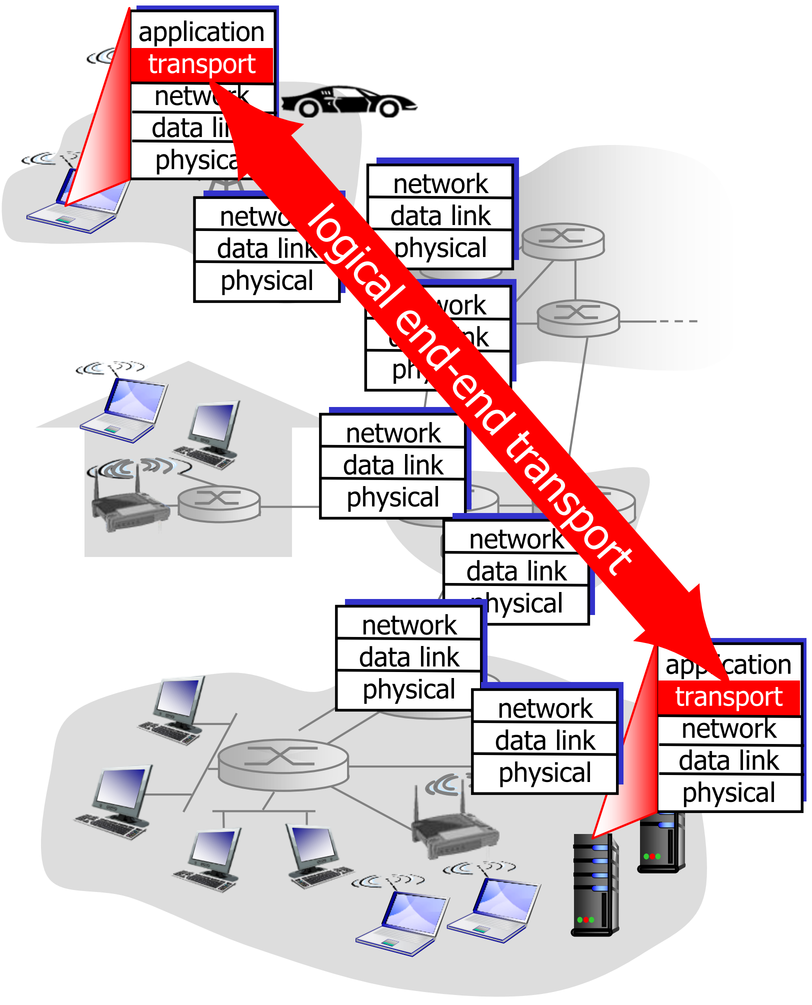
household analogy:

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

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

Internet transport-layer protocols

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



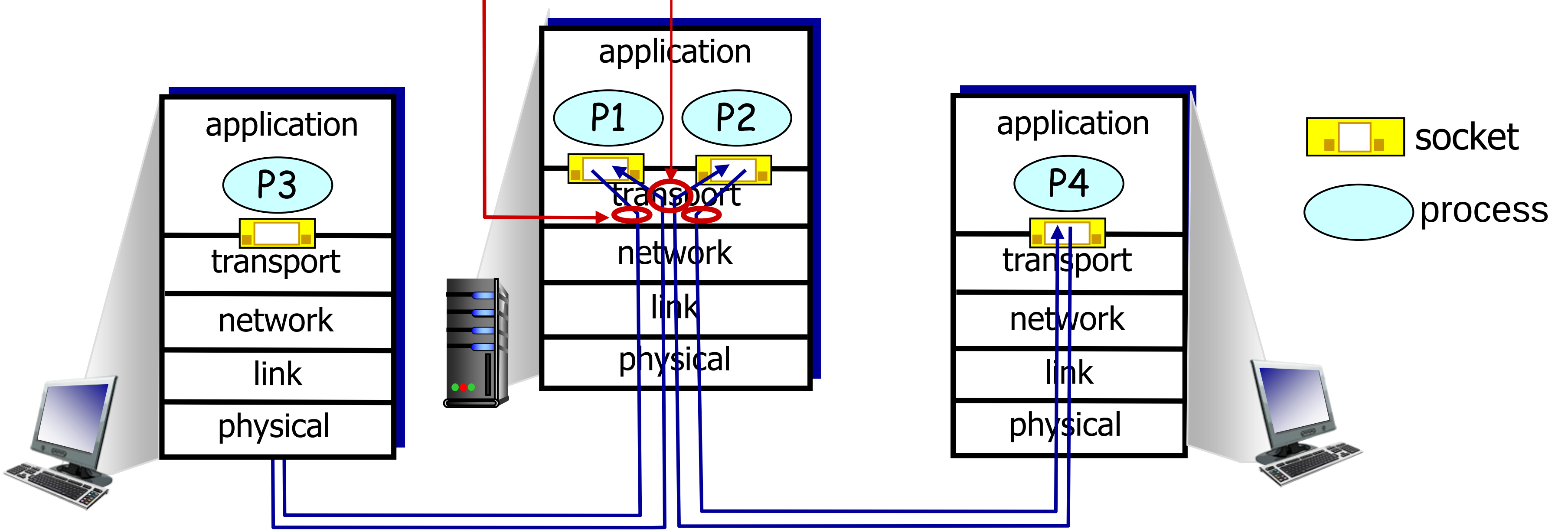
Topics

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

Multiplexing/demultiplexing

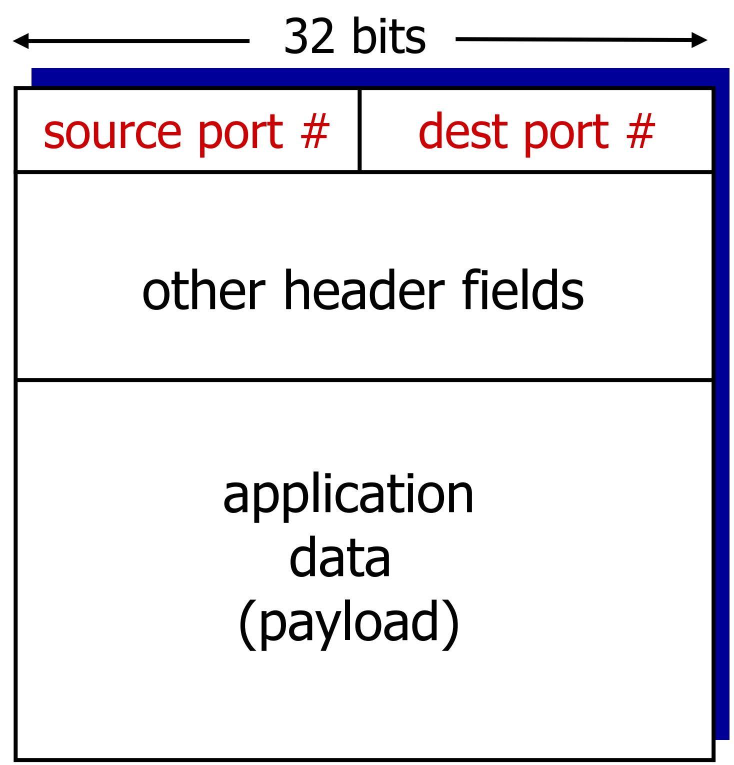
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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



How demultiplexing works

- host receives IP datagrams
  - each datagram has source 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

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

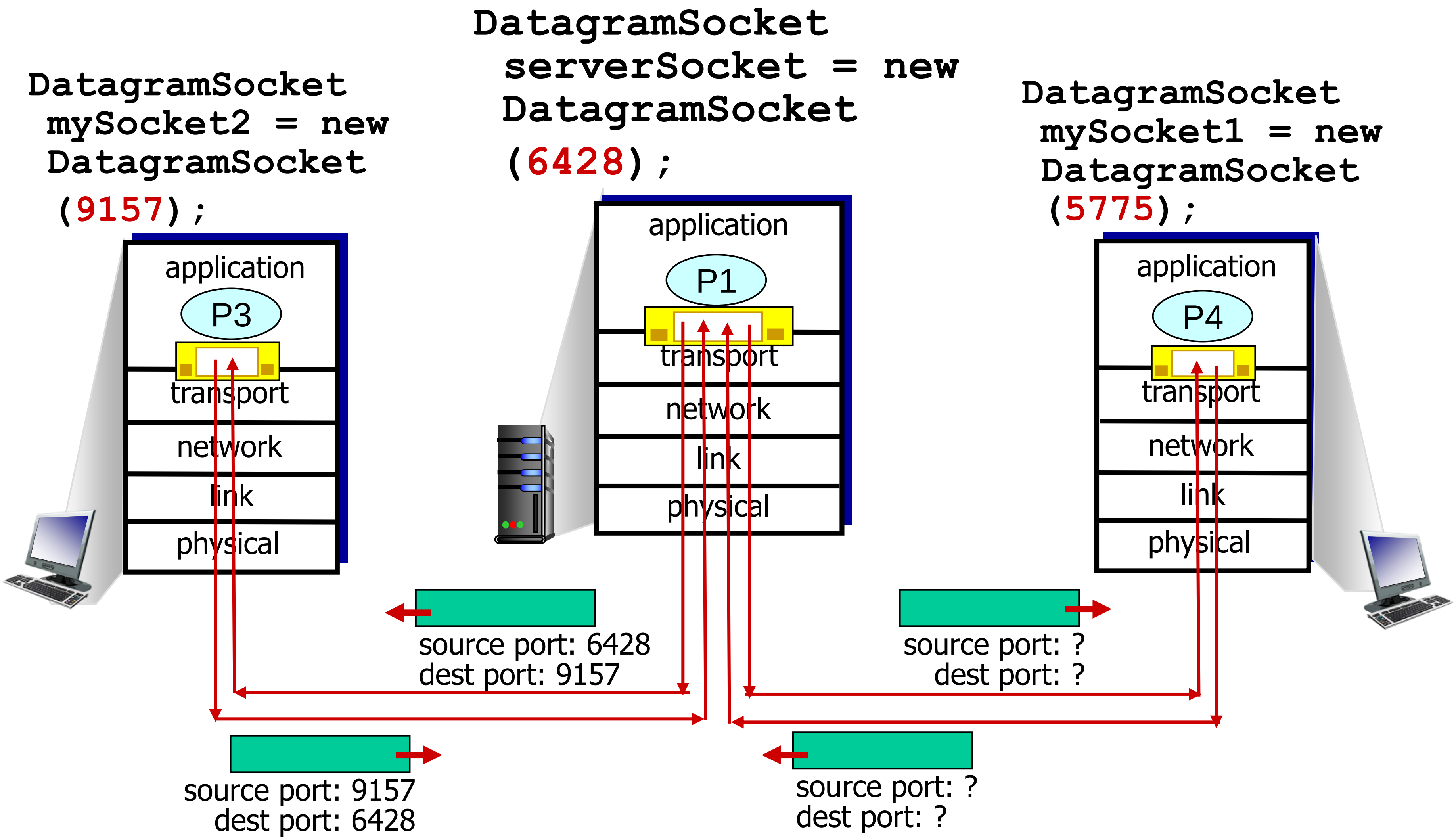
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

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

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

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

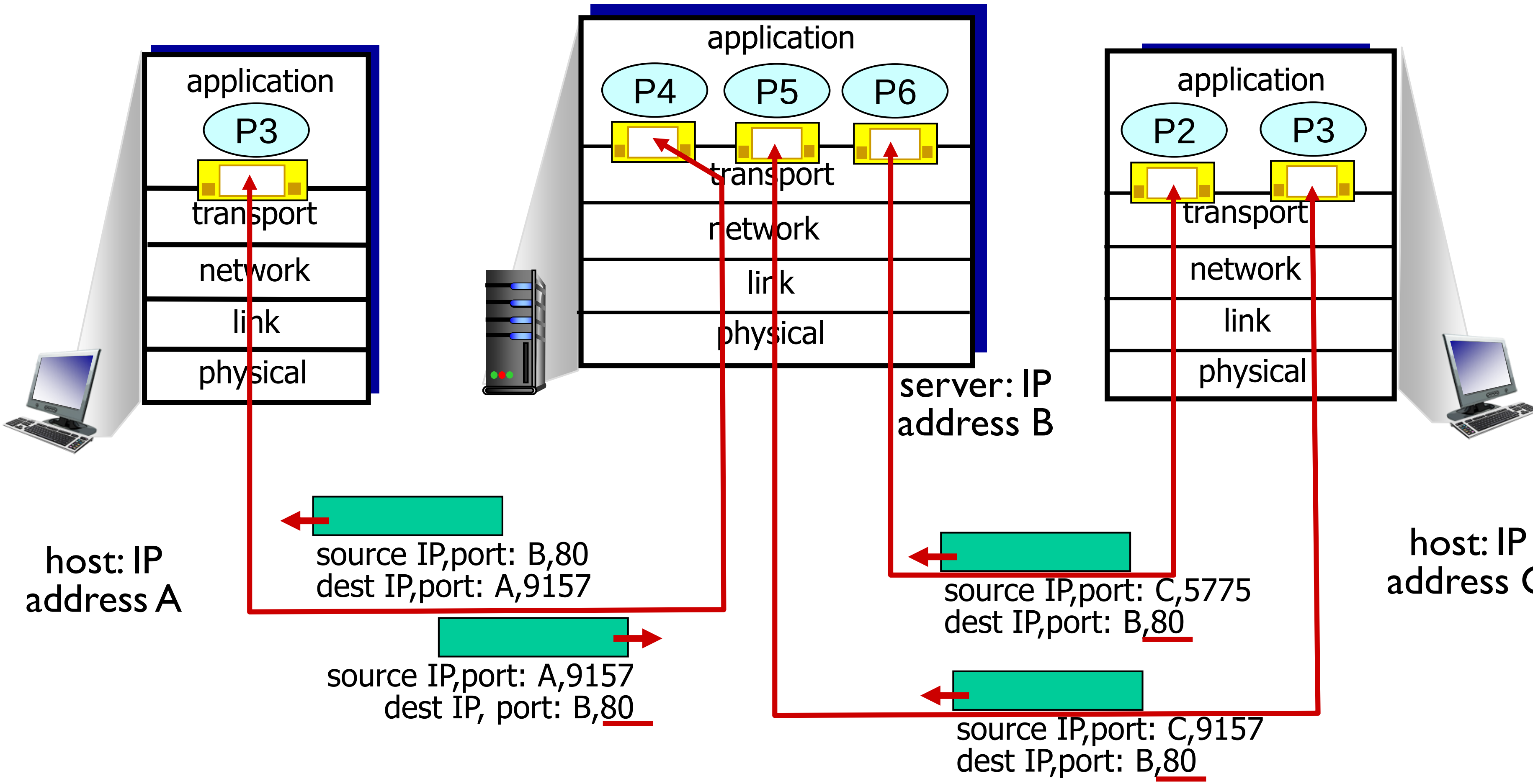
Connectionless demux: example



Connection-oriented demux

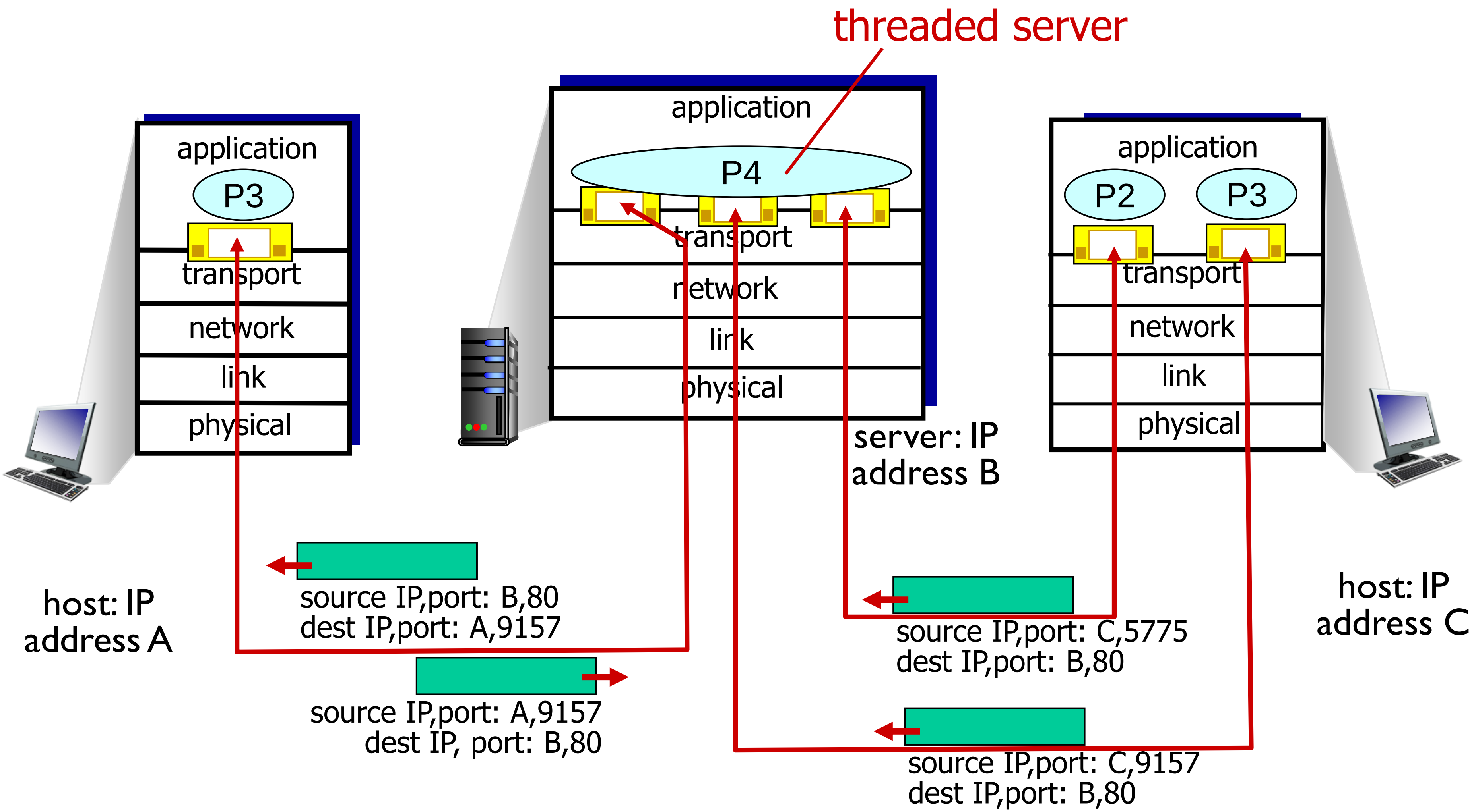
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- 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,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



Topics

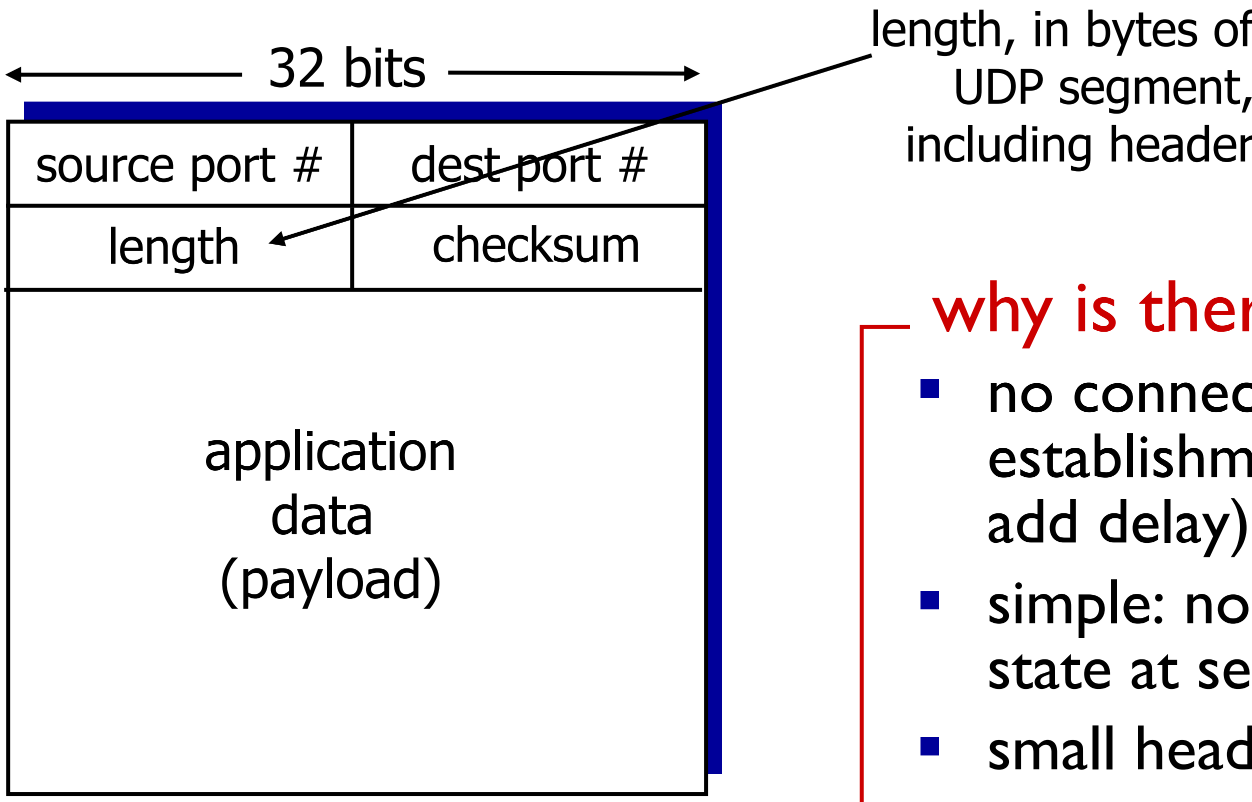
- Transport-layer services
- Multiplexing and demultiplexing
- UDP: Connectionless transport
- Principles of reliable data transfer
- TCP: Connection-oriented transport
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP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

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/

Exercise

- Compute the Internet checksum value for these 16-bit words:

1000 0110 0101 1110
1010 1100 0110 0000
0111 0001 0010 1010
1000 0001 1011 0101

Exercise solution

First, we add the 16-bit values 2 at a time:

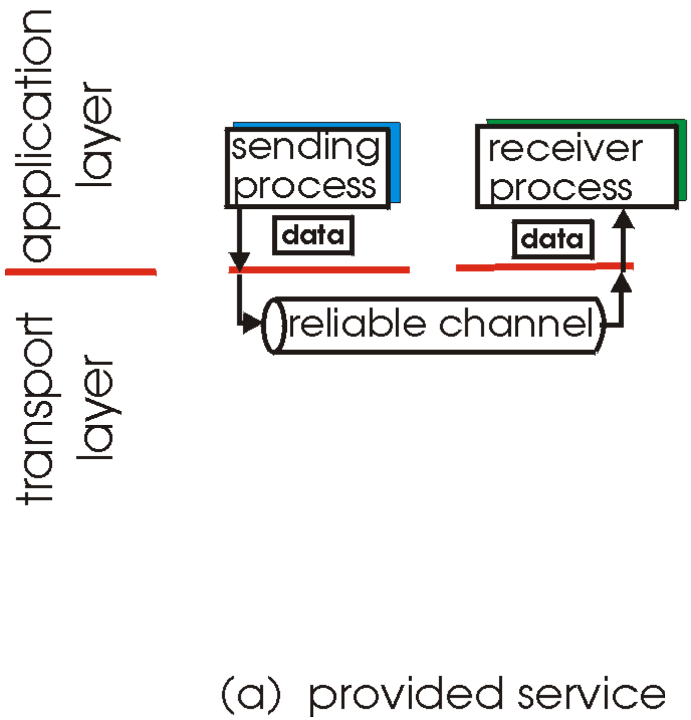
```
    1000 0110 0101 1110   First 16-bit value
 +  1010 1100 0110 0000   Second 16-bit value
  ----------------------------
  1 0011 0010 1011 1110   Produced a carry-out, which gets added
 +  \---------------------> 1   back into LBb
  ------------------------------
    0011 0010 1011 1111
 +  0111 0001 0010 1010   Third 16-bit value
  ------------------------------
  01010 0011 1110 1001   No carry to swing around (**)
 +  1000 0001 1011 0101   Fourth 16-bit value
  ------------------------------
  1 0010 0101 1001 1110   Produced a carry-out, which gets added
 +  \---------------------> 1   back into LBb
  -------------------------------
    0010 0101 1001 1111   Our "one's complement sum“

    1101 1010 0110 0000  Checksum
```

Topics

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

Principles of reliable data transfer

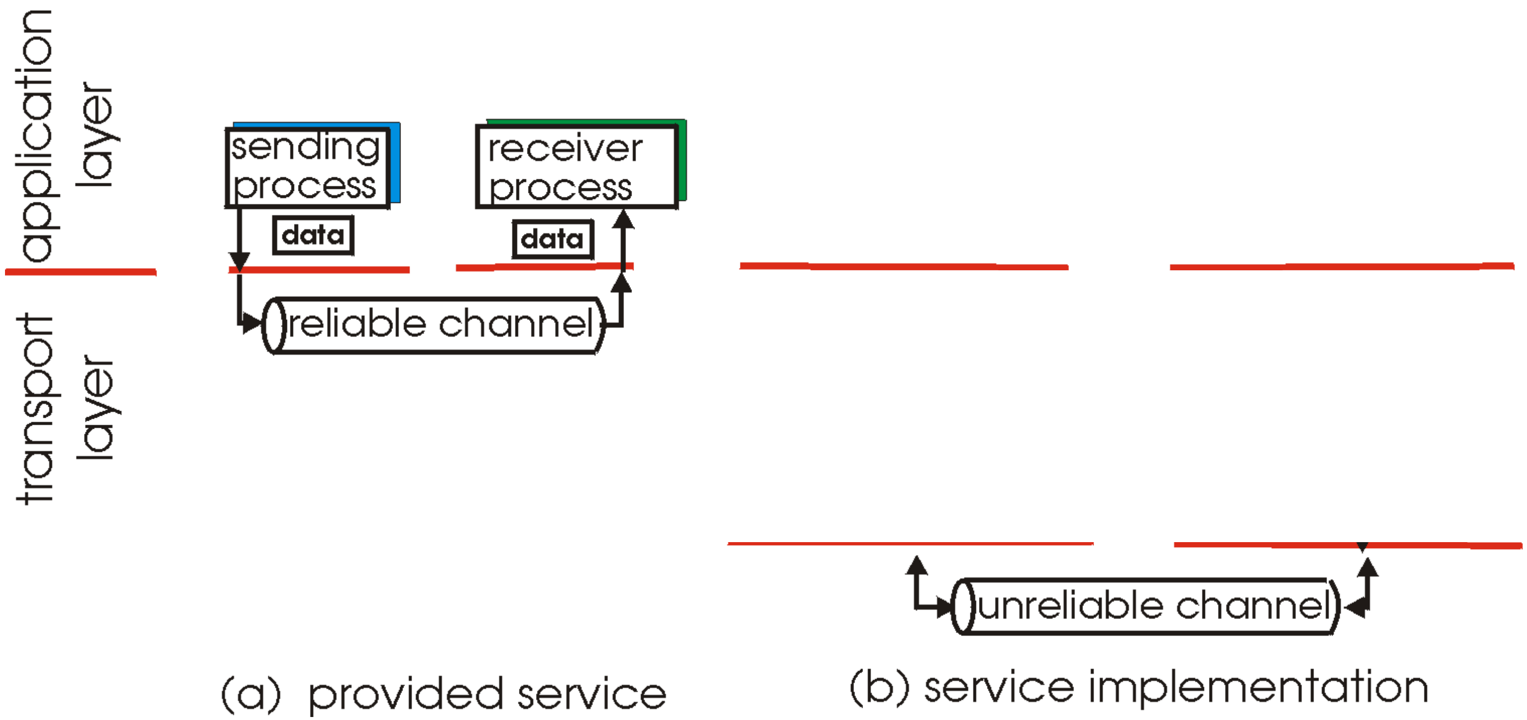
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

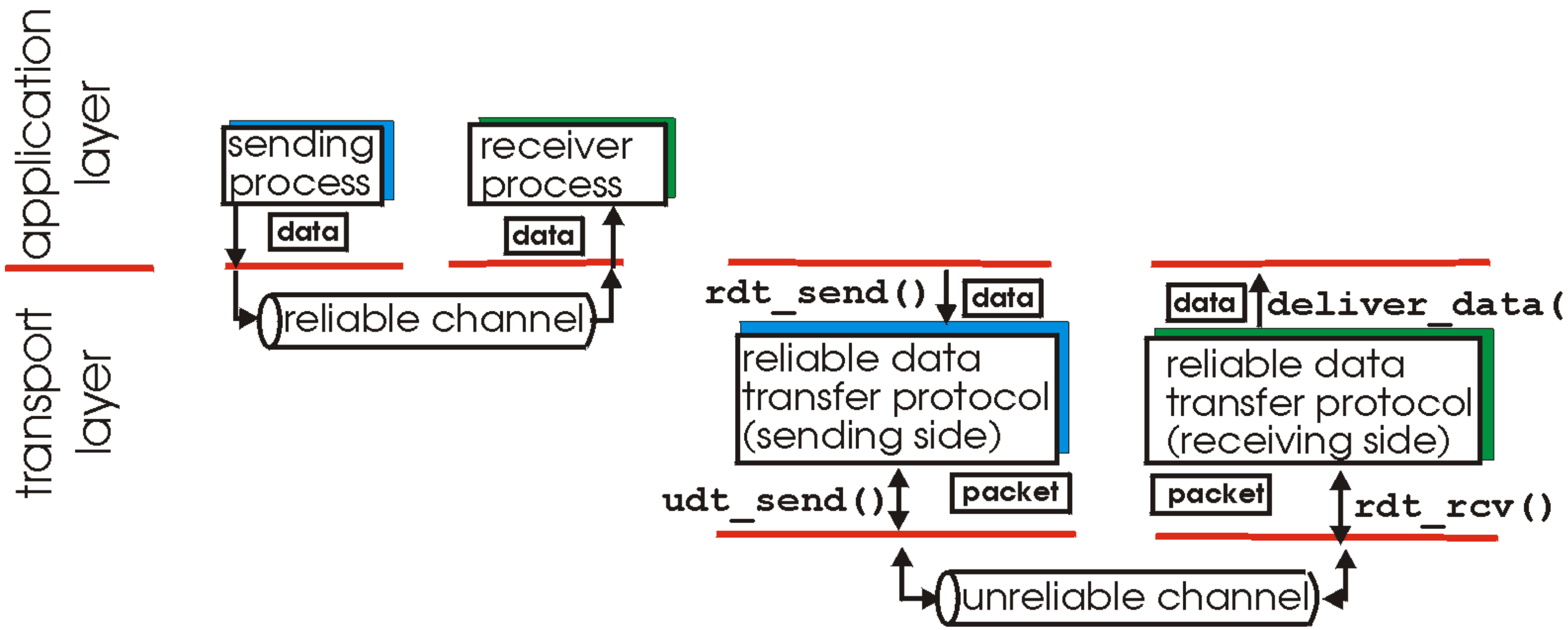
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer (rdt)

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



(a) provided service

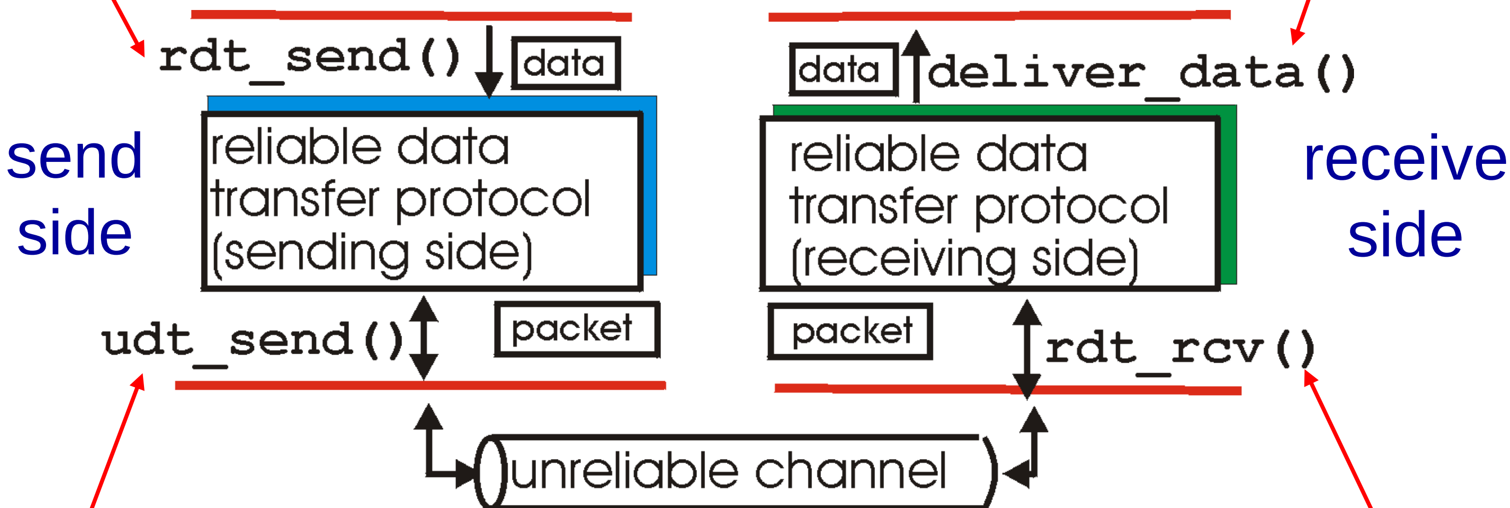
(b) service implementation

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

Reliable data transfer: getting started

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

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



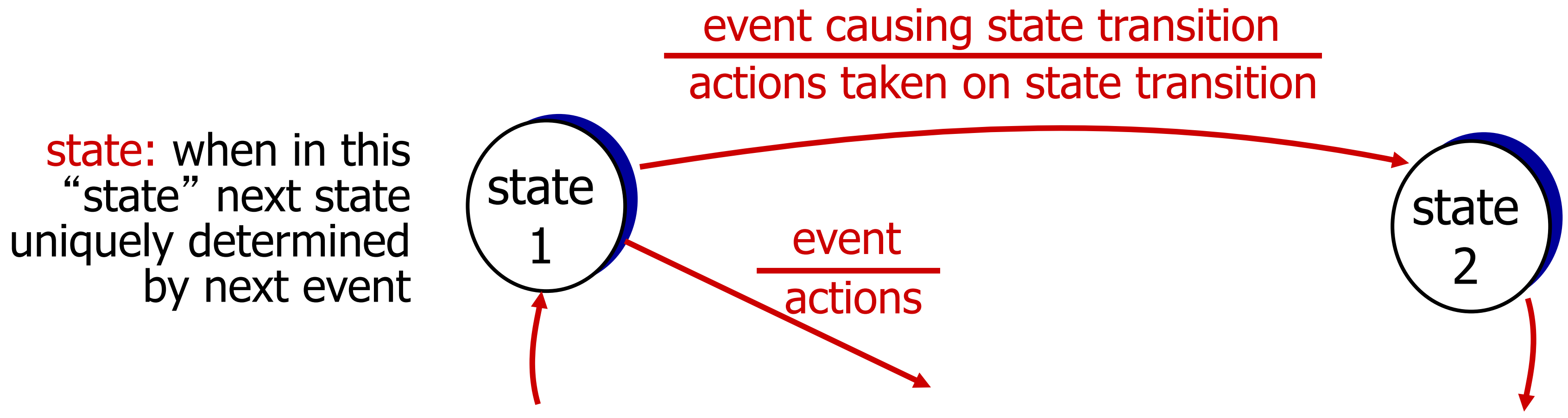
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

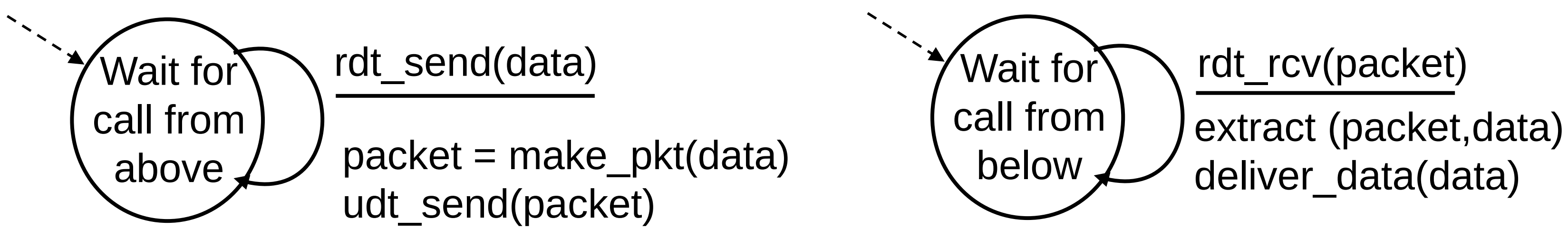
We'll:

- 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



sender receiver

rdt2.0: channel with bit errors

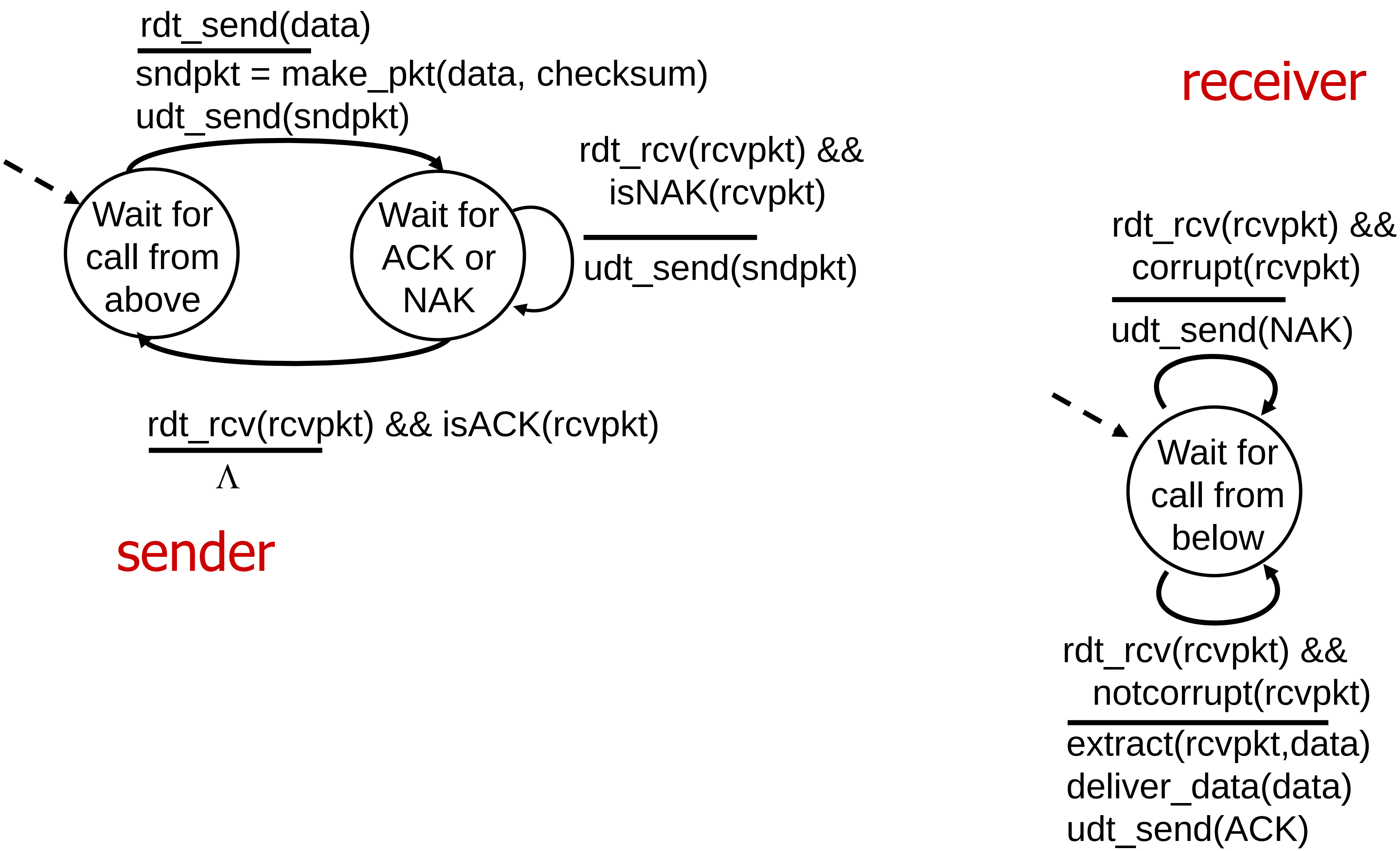
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

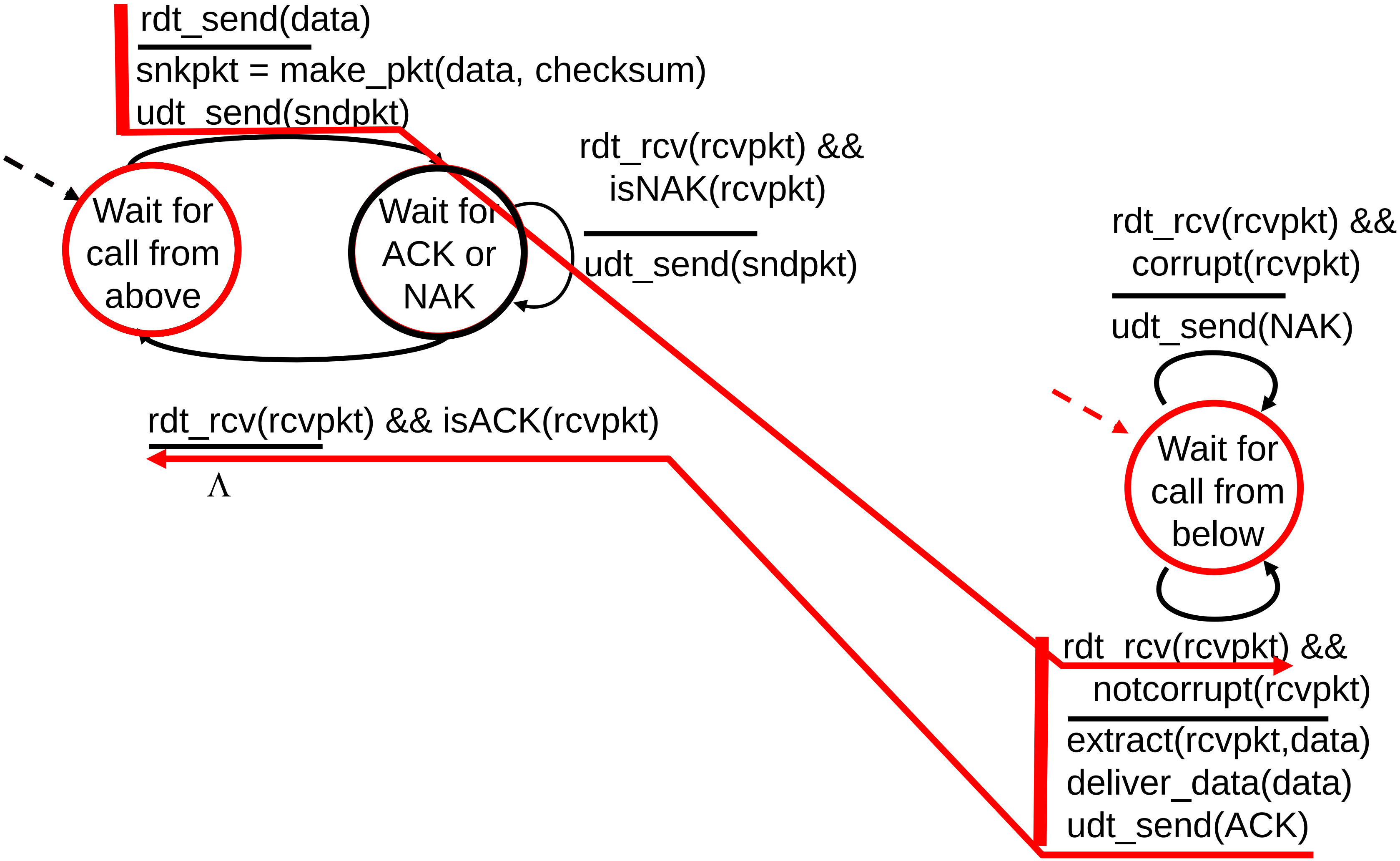
rdt2.0: channel with bit errors

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

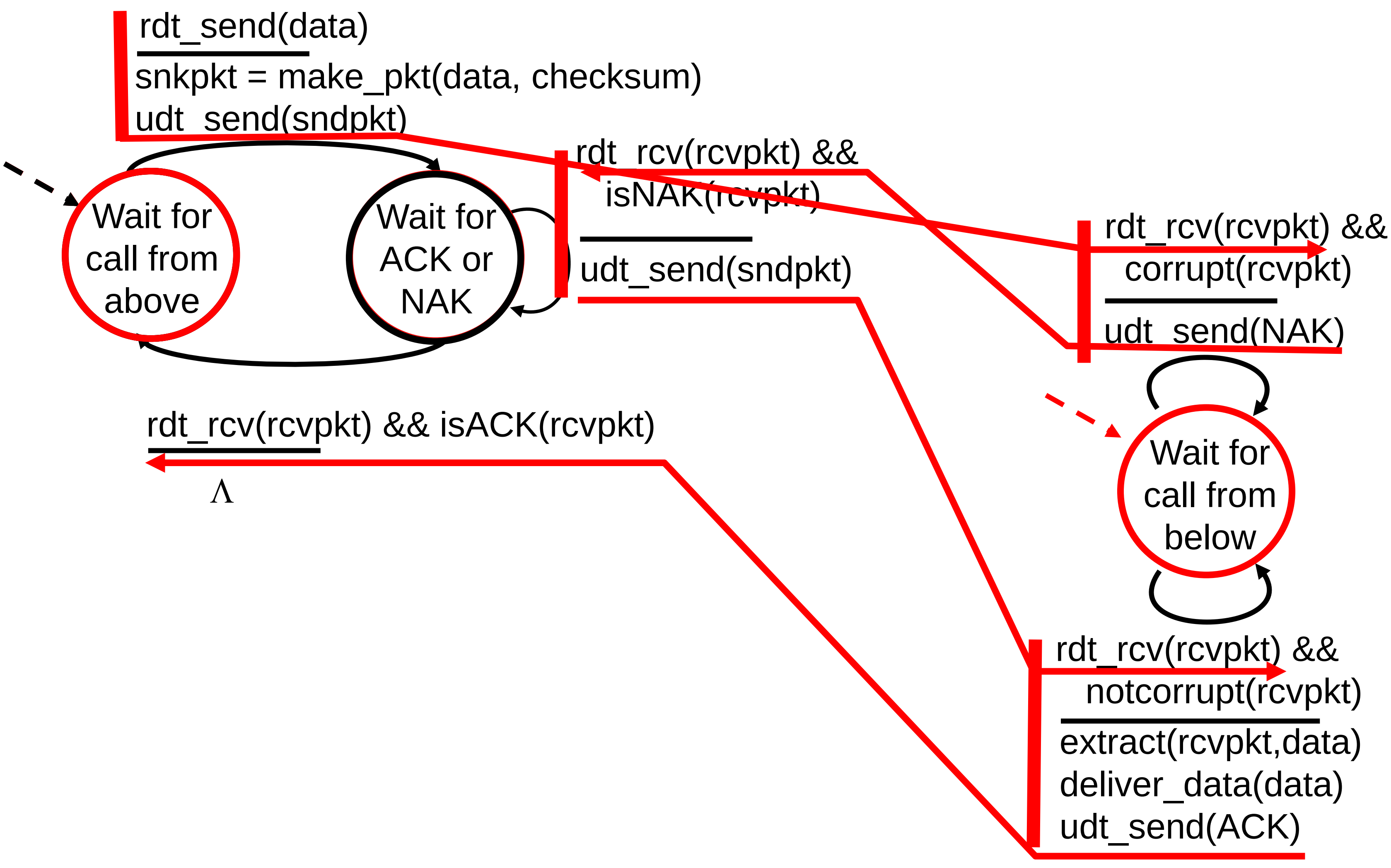
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

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

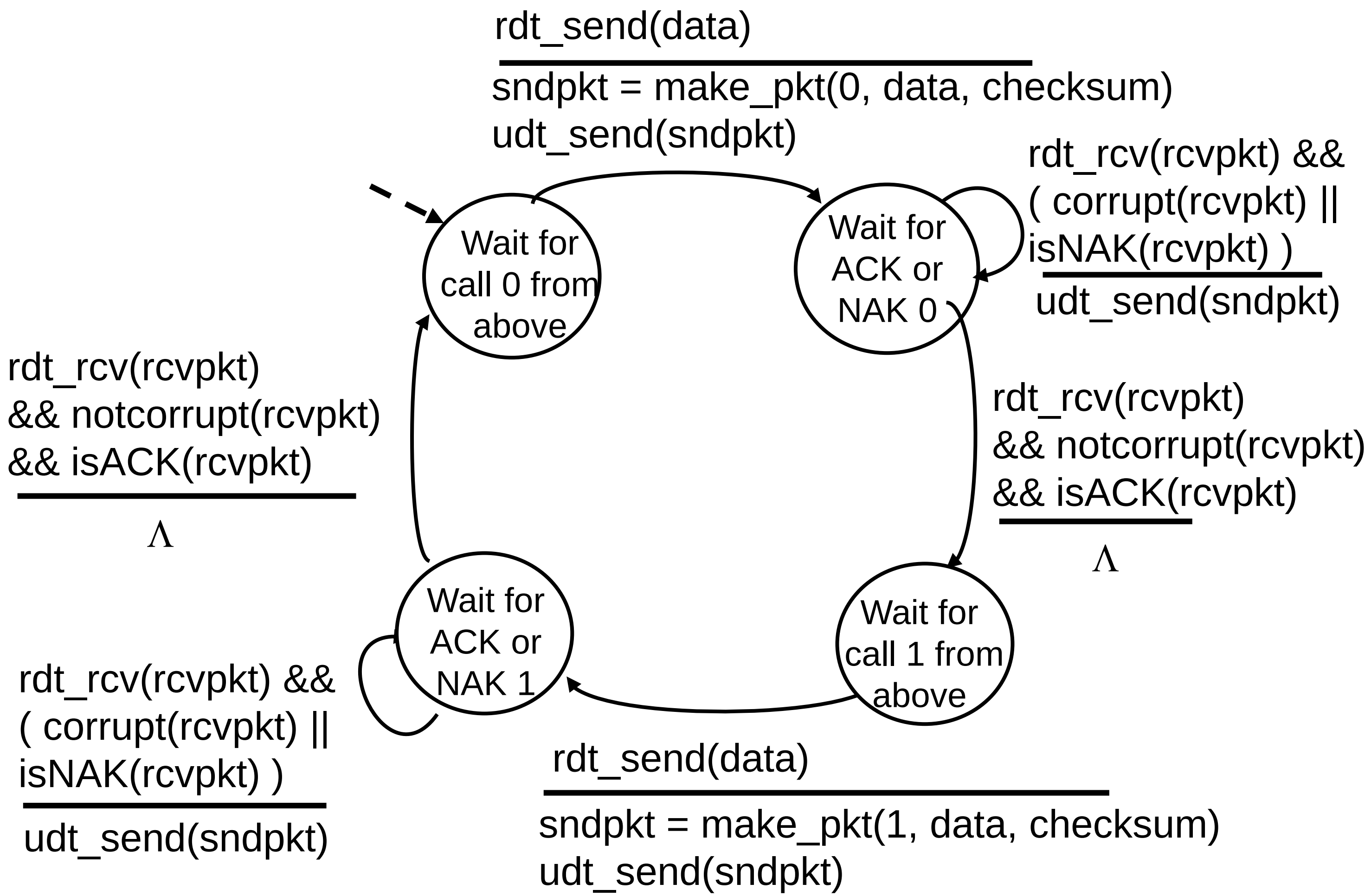
handling duplicates:

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

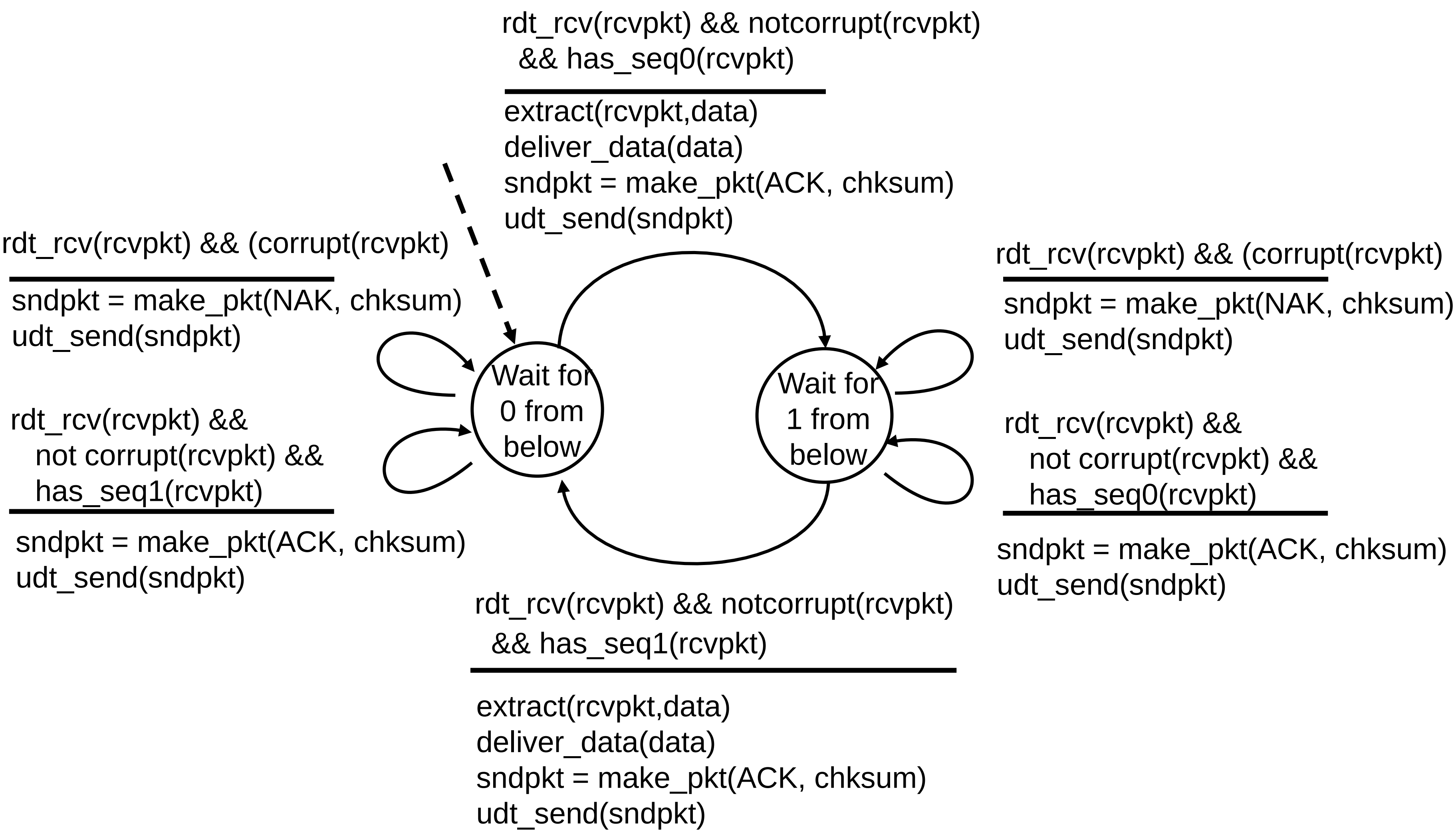
stop and wait

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: discussion

<u>sender:</u>

- 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

<u>receiver:</u>

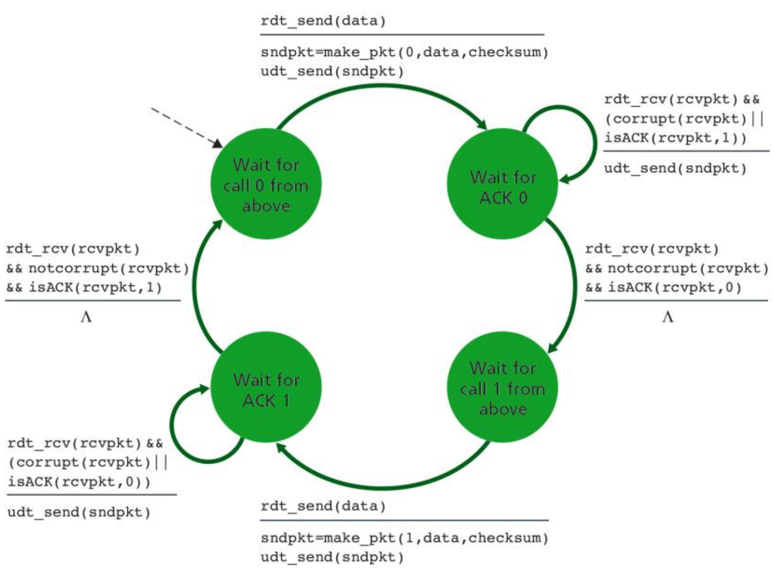
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

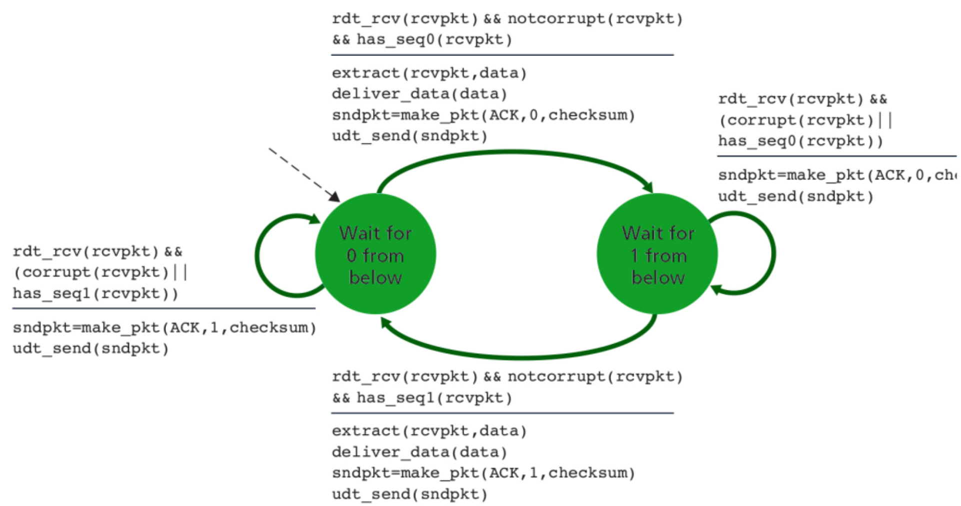
- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: a NAK-free protocol

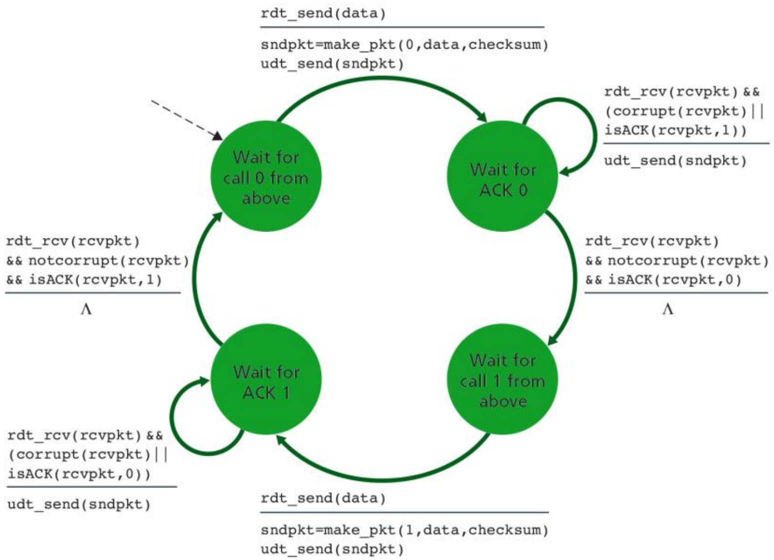
- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed



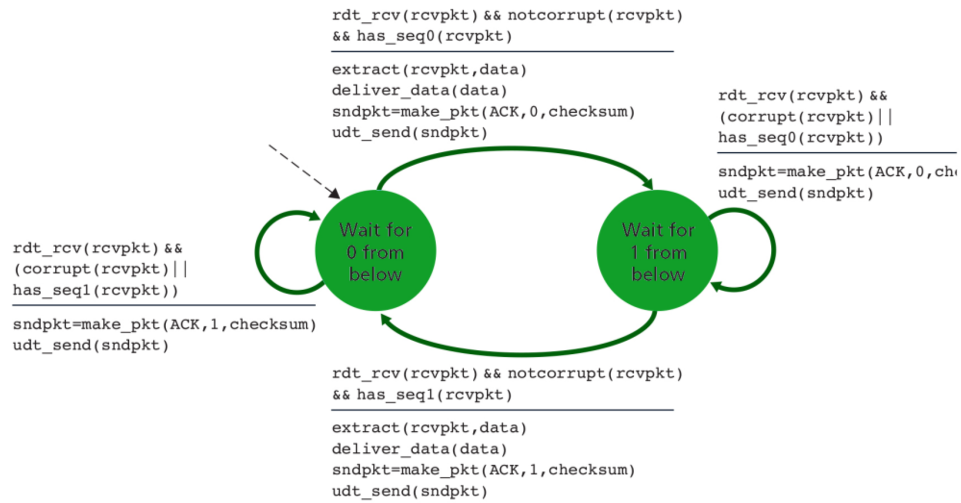
rdt2.2 sender



rdt2.2 receiver

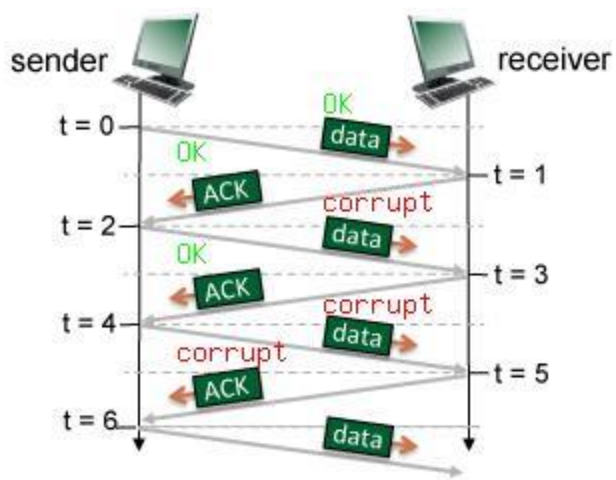


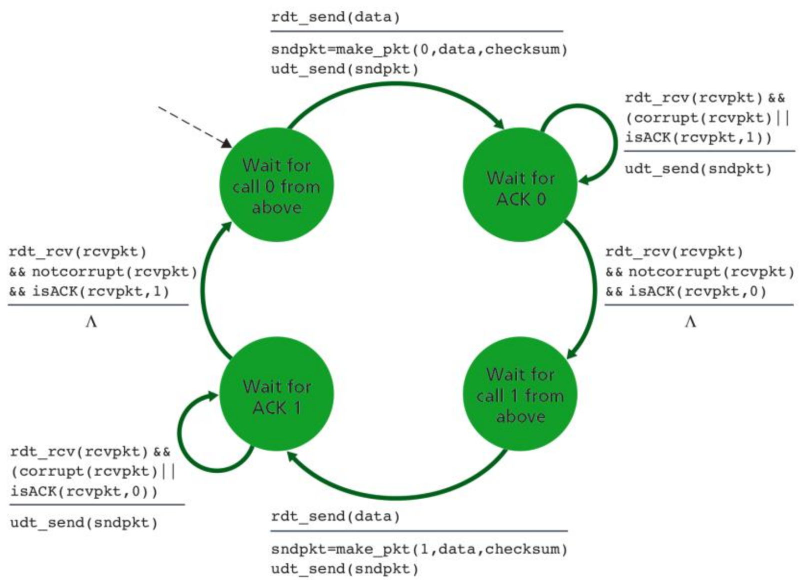
rdt2.2 sender

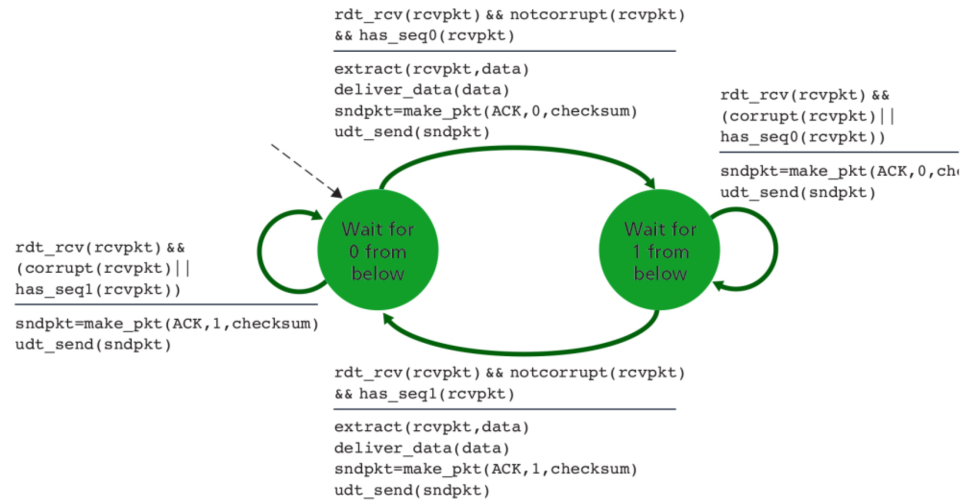


rdt2.2 receiver

Suppose that the channel connecting the sender and receiver can corrupt but not lose or reorder packets. Now consider the figure below, which shows four data packets and three corresponding ACKs being exchanged between an rdt 2.2 sender and receiver. The actual corruption or successful transmission/reception of a packet is indicated by the corrupt and OK labels, respectively, shown above the packets in the figure below.



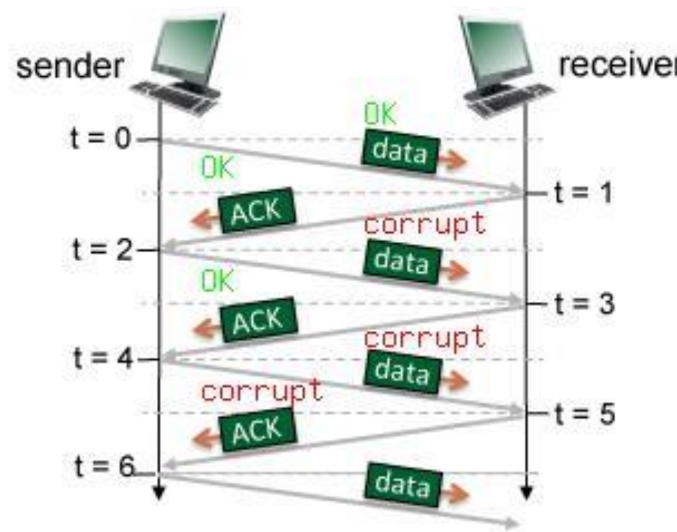


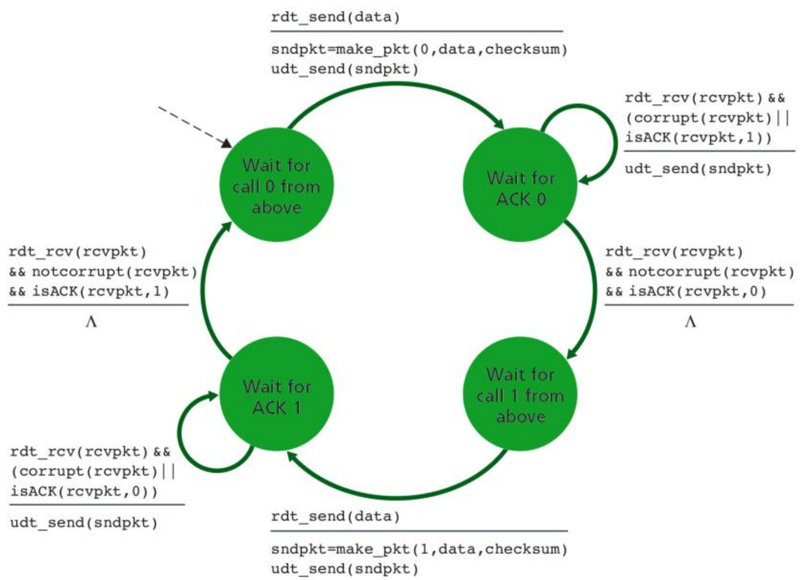


rdt2.2 receiver

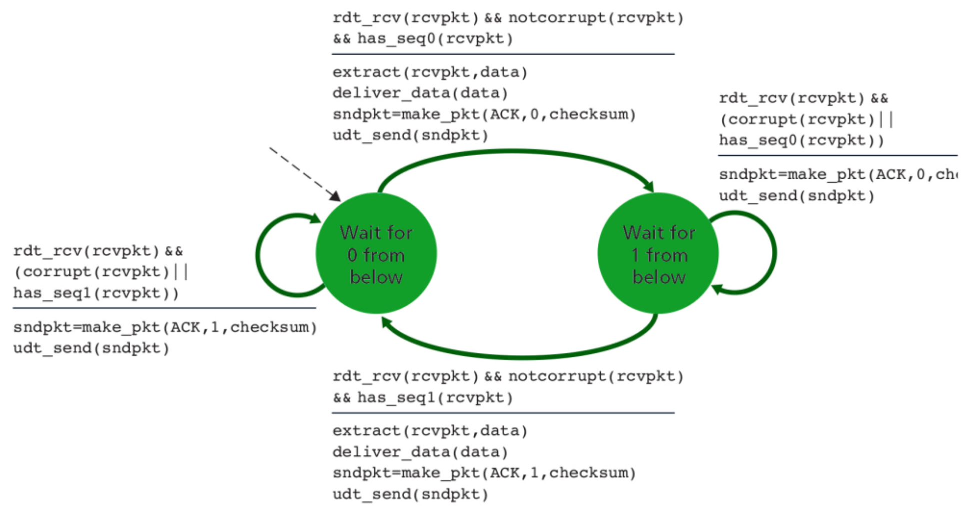
rdt2.2 sender

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0		Wait0 from below	data	
1			ACK	
2			data	
3			ACK	
4			data	
5			ACK	
6			data	



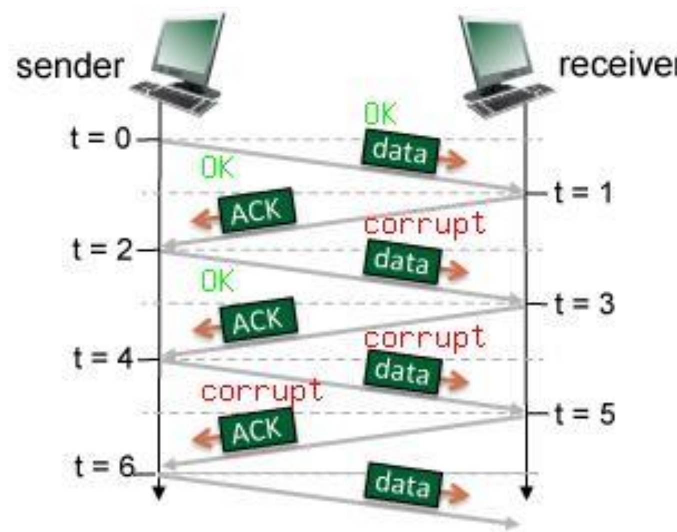


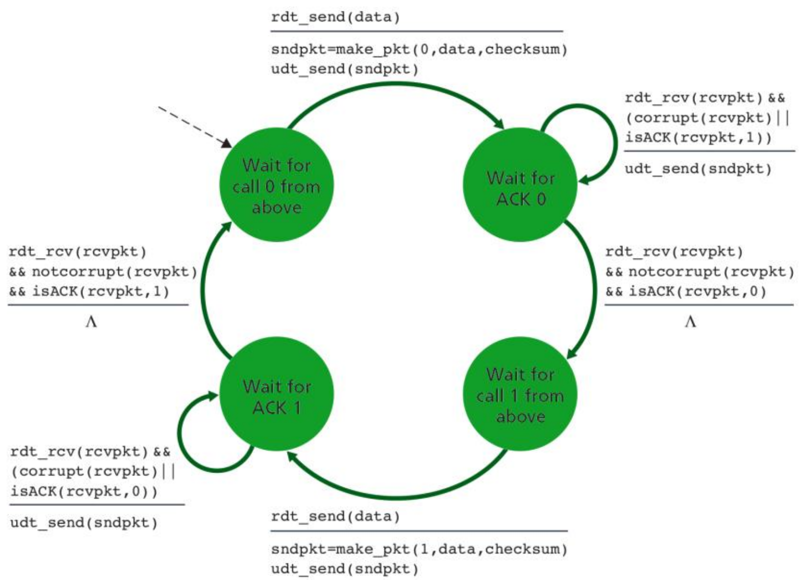
rdt2.2 sender



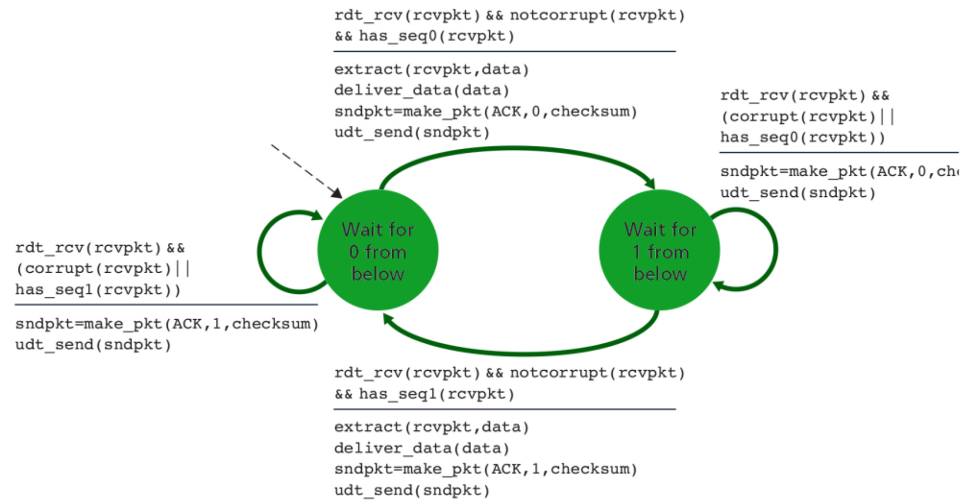
rdt2.2 receiver

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1			ACK	
2			data	
3			ACK	
4			data	
5			ACK	
6			data	



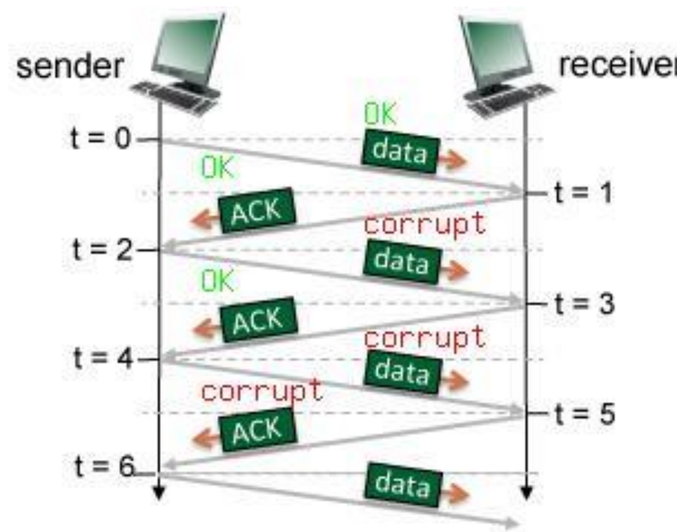


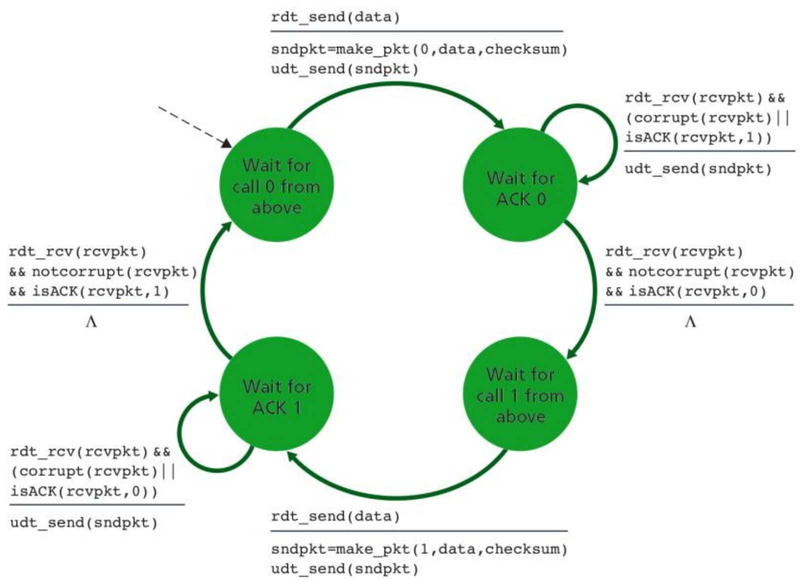
rdt2.2 sender



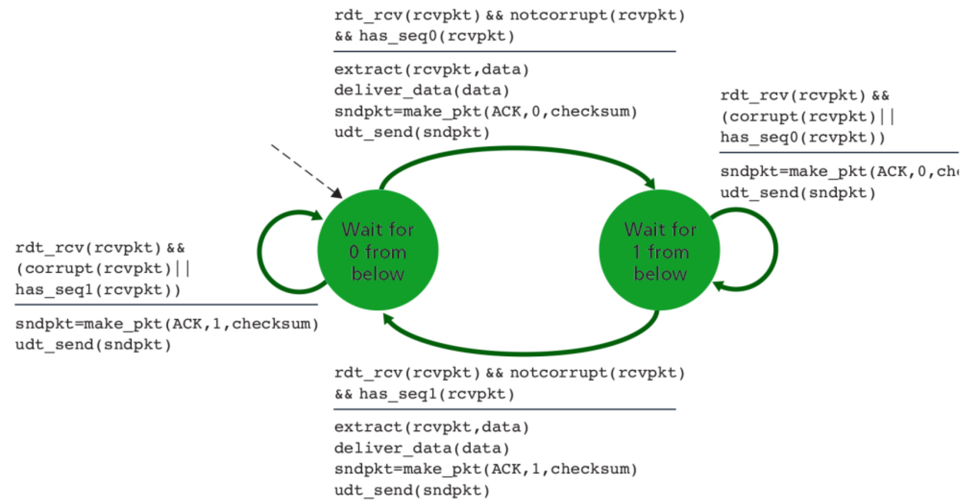
rdt2.2 receiver

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2			data	
3			ACK	
4			data	
5			ACK	
6			data	



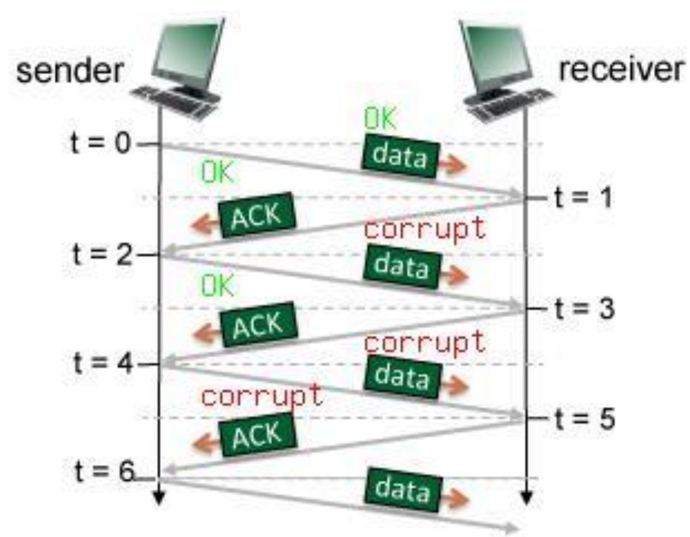


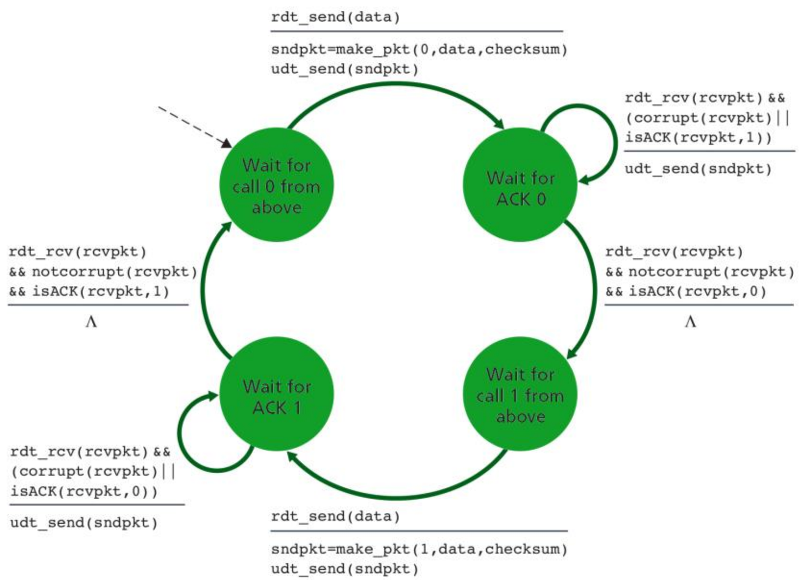
rdt2.2 sender



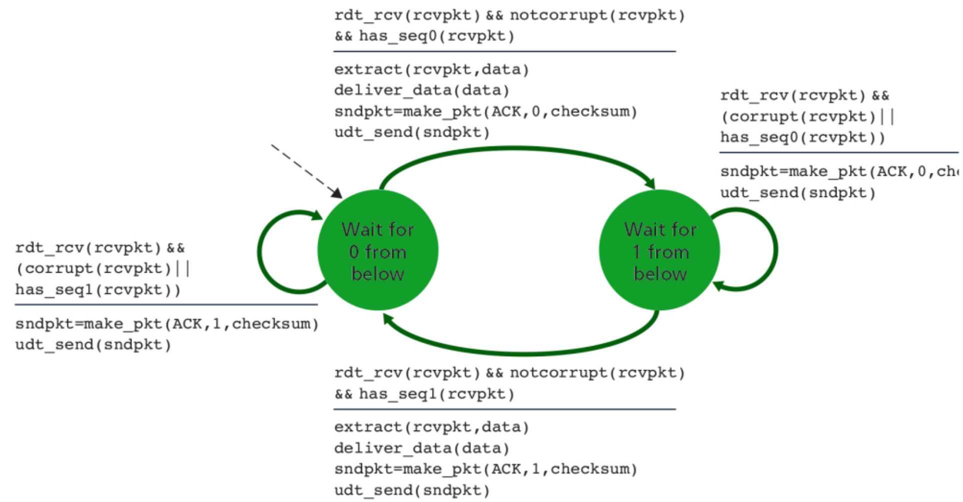
rdt2.2 receiver

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2	Wait ACK1	Wait1 from below	data	1
3			ACK	
4			data	
5			ACK	
6			data	



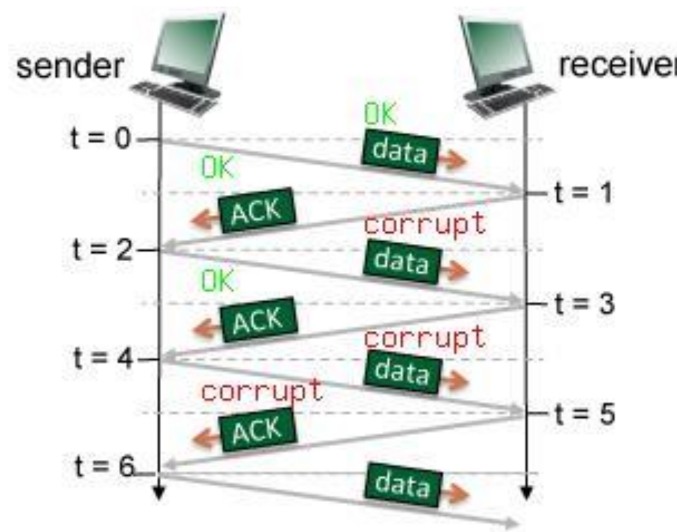


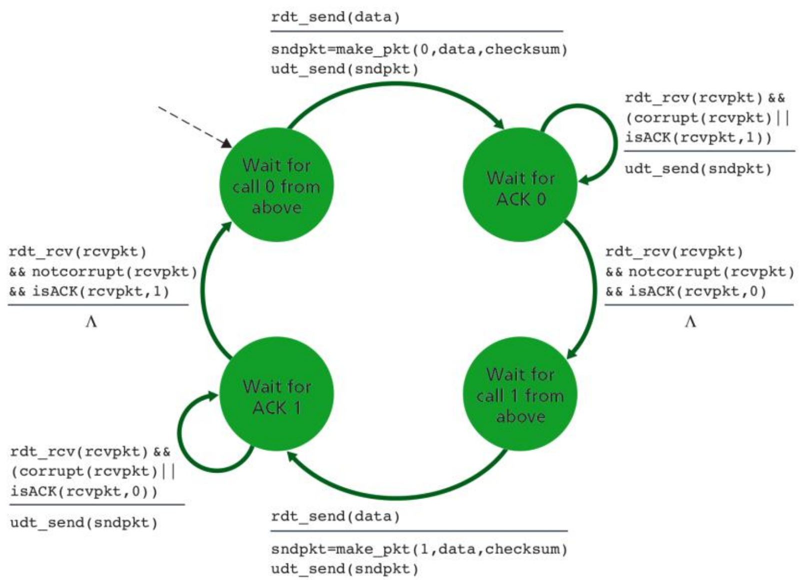
rdt2.2 sender



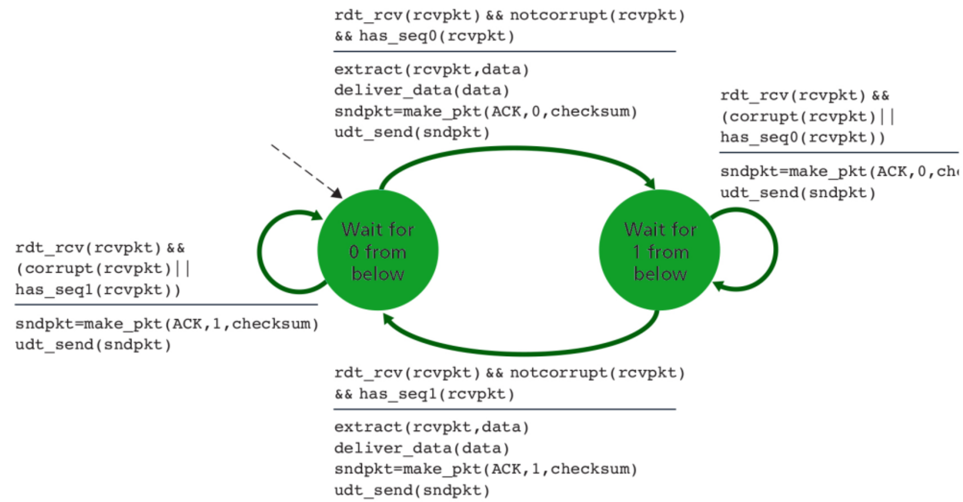
rdt2.2 receiver

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2	Wait ACK1	Wait1 from below	data	1
3	Wait ACK1	Wait1 from below	ACK	0
4			data	
5			ACK	
6			data	



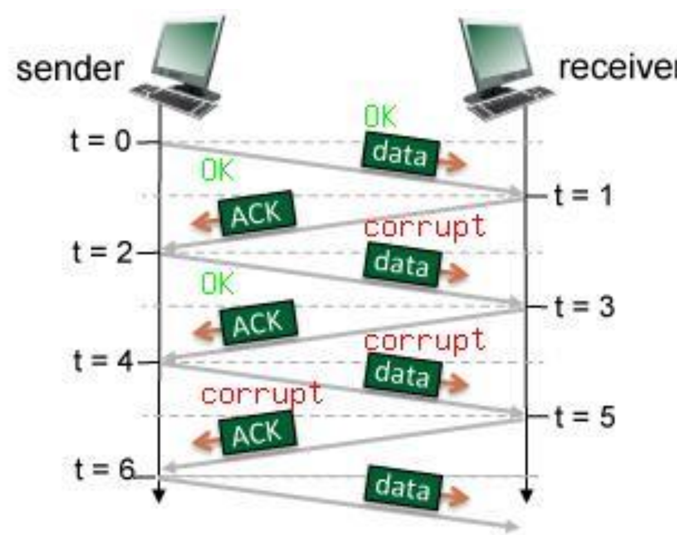


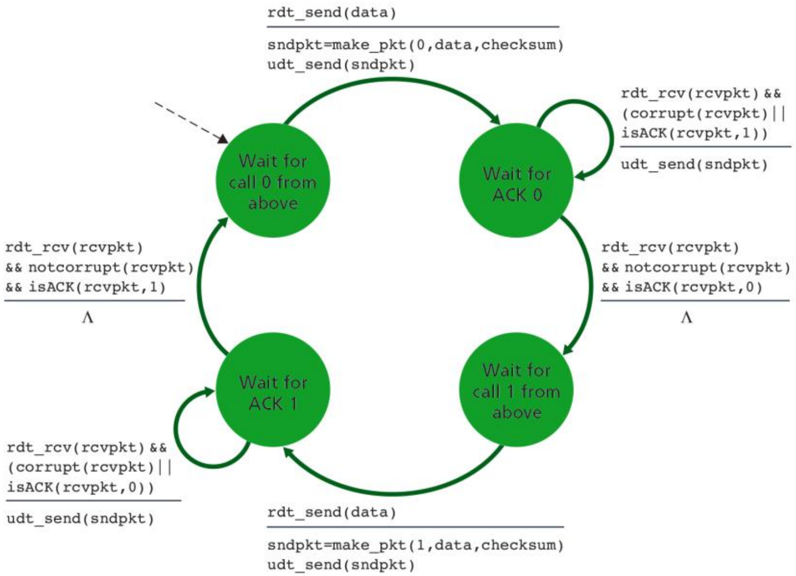
rdt2.2 sender



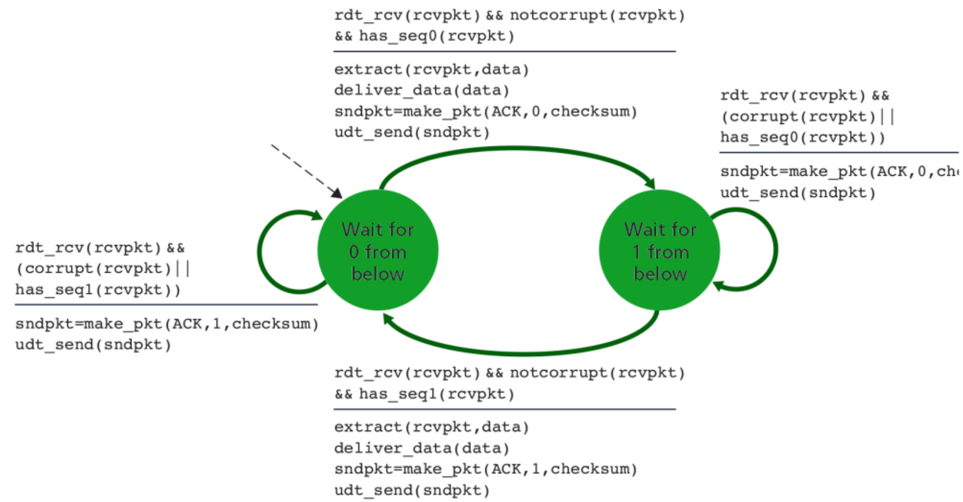
rdt2.2 receiver

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2	Wait ACK1	Wait1 from below	data	1
3	Wait ACK1	Wait1 from below	ACK	0
4	Wait ACK1	Wait1 from below	data	1
5			ACK	
6			data	



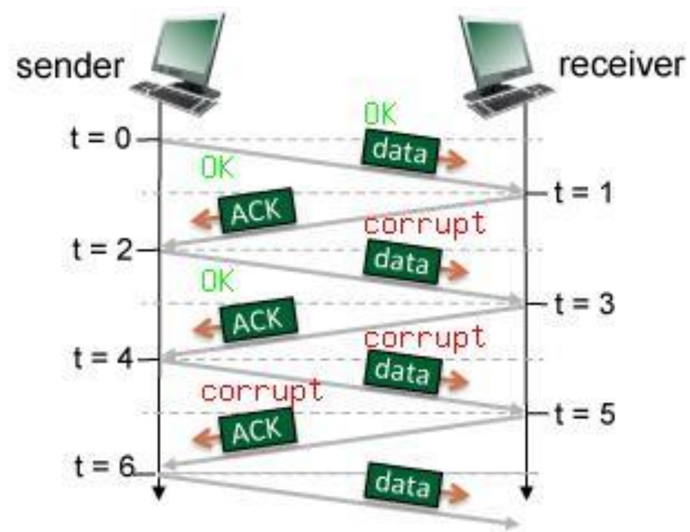


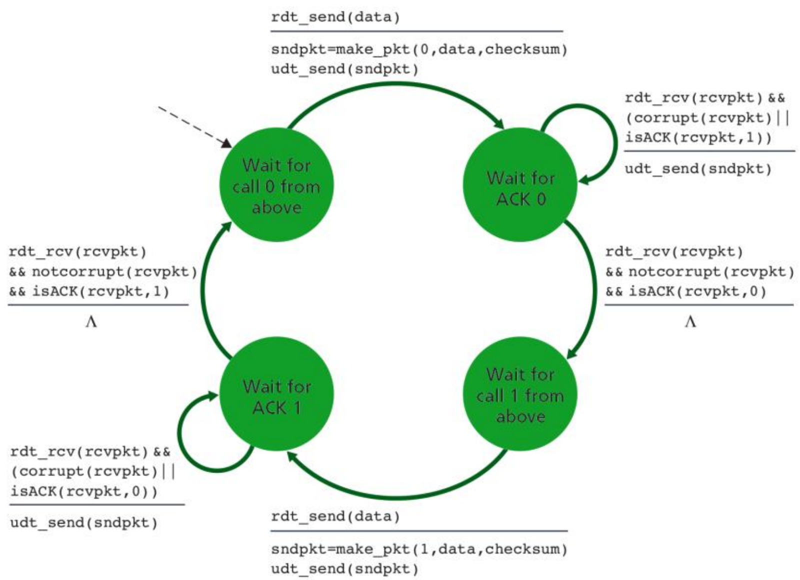
rdt2.2 sender



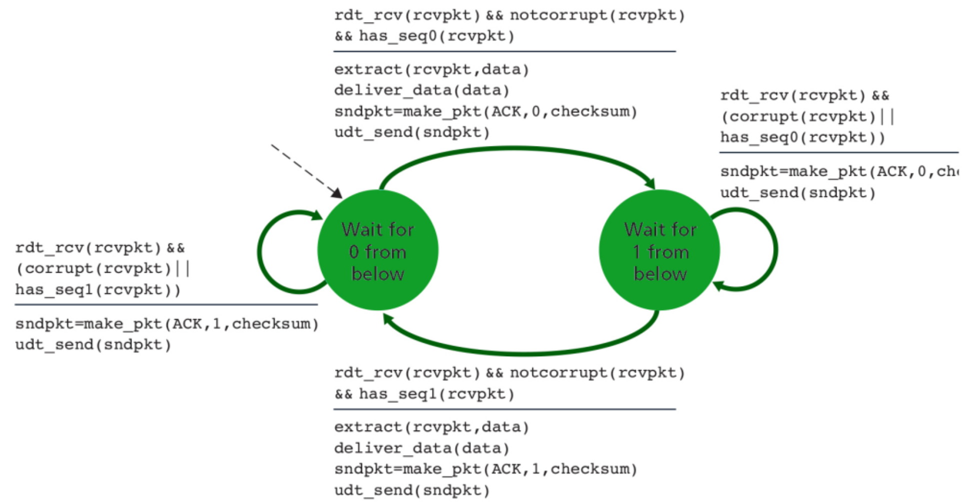
rdt2.2 receiver

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2	Wait ACK1	Wait1 from below	data	1
3	Wait ACK1	Wait1 from below	ACK	0
4	Wait ACK1	Wait1 from below	data	1
5	Wait ACK1	Wait1 from below	ACK	0
6			data	



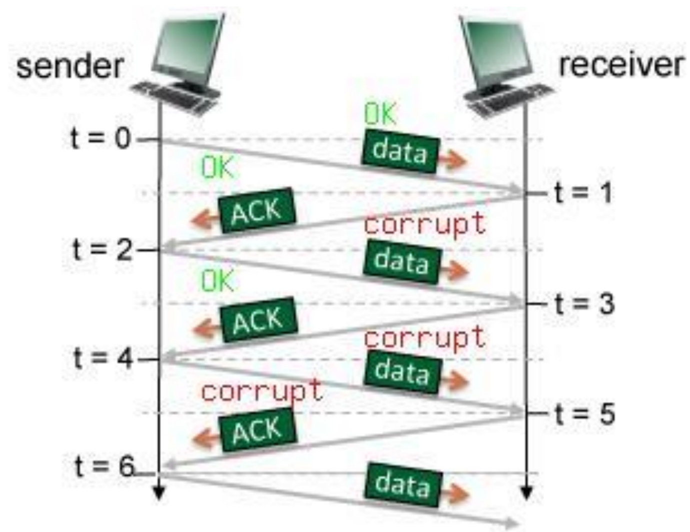


rdt2.2 sender

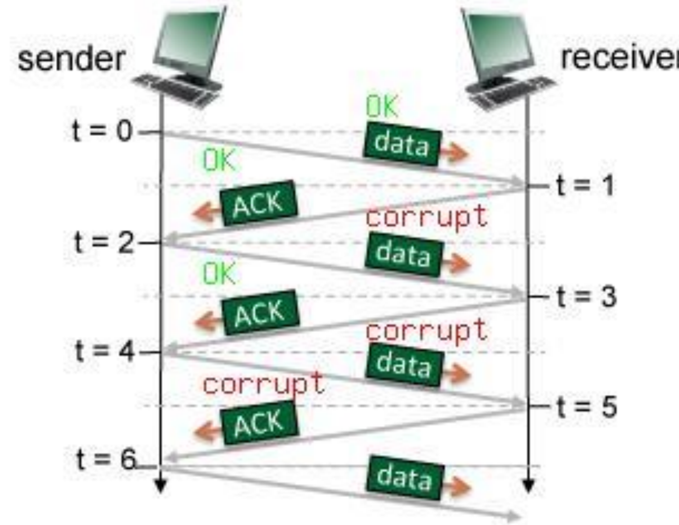


rdt2.2 receiver

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2	Wait ACK1	Wait1 from below	data	1
3	Wait ACK1	Wait1 from below	ACK	0
4	Wait ACK1	Wait1 from below	data	1
5	Wait ACK1	Wait1 from below	ACK	0
6	Wait ACK1	Wait1 from below	data	1

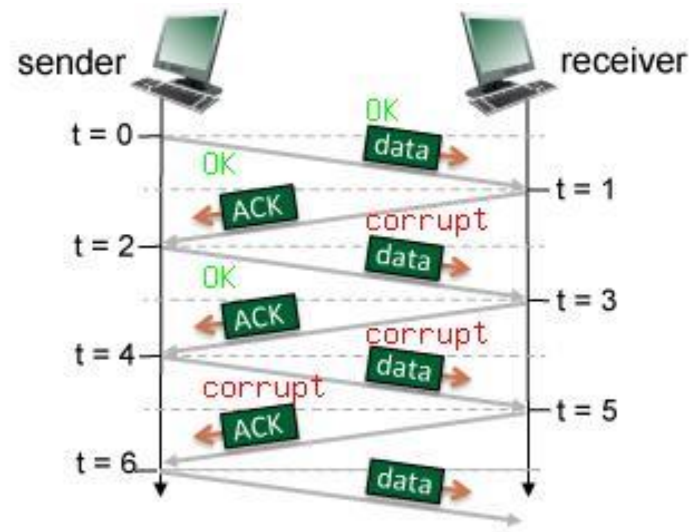


t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2	Wait ACK1	Wait1 from below	data	1
3	Wait ACK1	Wait1 from below	ACK	0
4	Wait ACK1	Wait1 from below	data	1
5	Wait ACK1	Wait1 from below	ACK	0
6	Wait ACK1	Wait1 from below	data	1



How many times is the payload of the received packet passed up to the higher layer at the receiver in the above example? At what times is the payload data passed up?

t	sender state	receiver state	packet type sent	seq. # or ACK # sent
0	Wait ACK0	Wait0 from below	data	0
1	Wait ACK0	Wait1 from below	ACK	0
2	Wait ACK1	Wait1 from below	data	1
3	Wait ACK1	Wait1 from below	ACK	0
4	Wait ACK1	Wait1 from below	data	1
5	Wait ACK1	Wait1 from below	ACK	0
6	Wait ACK1	Wait1 from below	data	1



How many times is the payload of the received packet passed up to the higher layer at the receiver in the above example? At what times is the payload data passed up?

One packet was passed up to the higher layer at the receiver at time t = 1.

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: ?

rdt3.0: channels with errors *and* loss

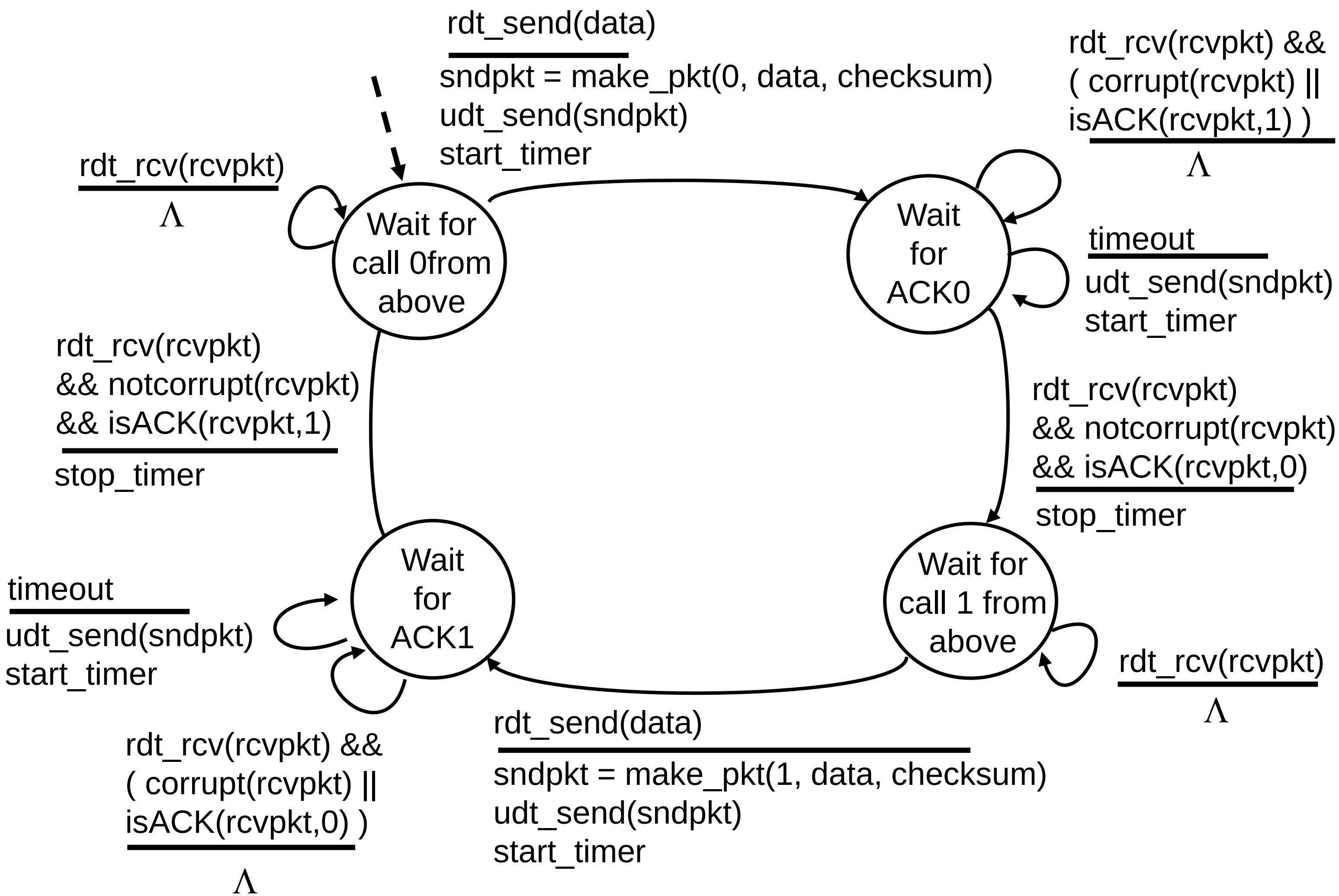
new assumption: underlying channel can also lose packets (data, ACKs)

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

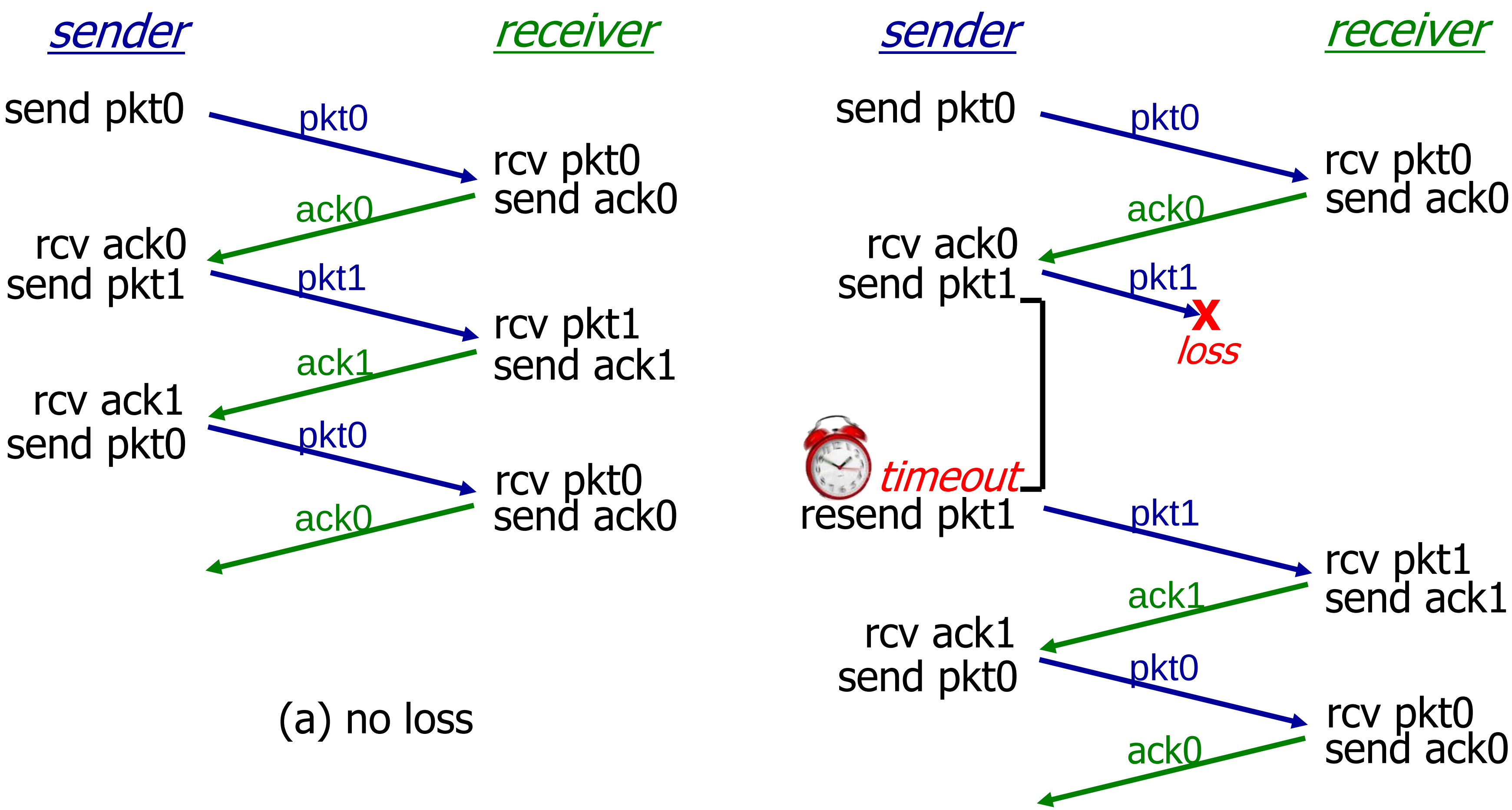
approach: sender waits "reasonable" amount of time for ACK

- 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



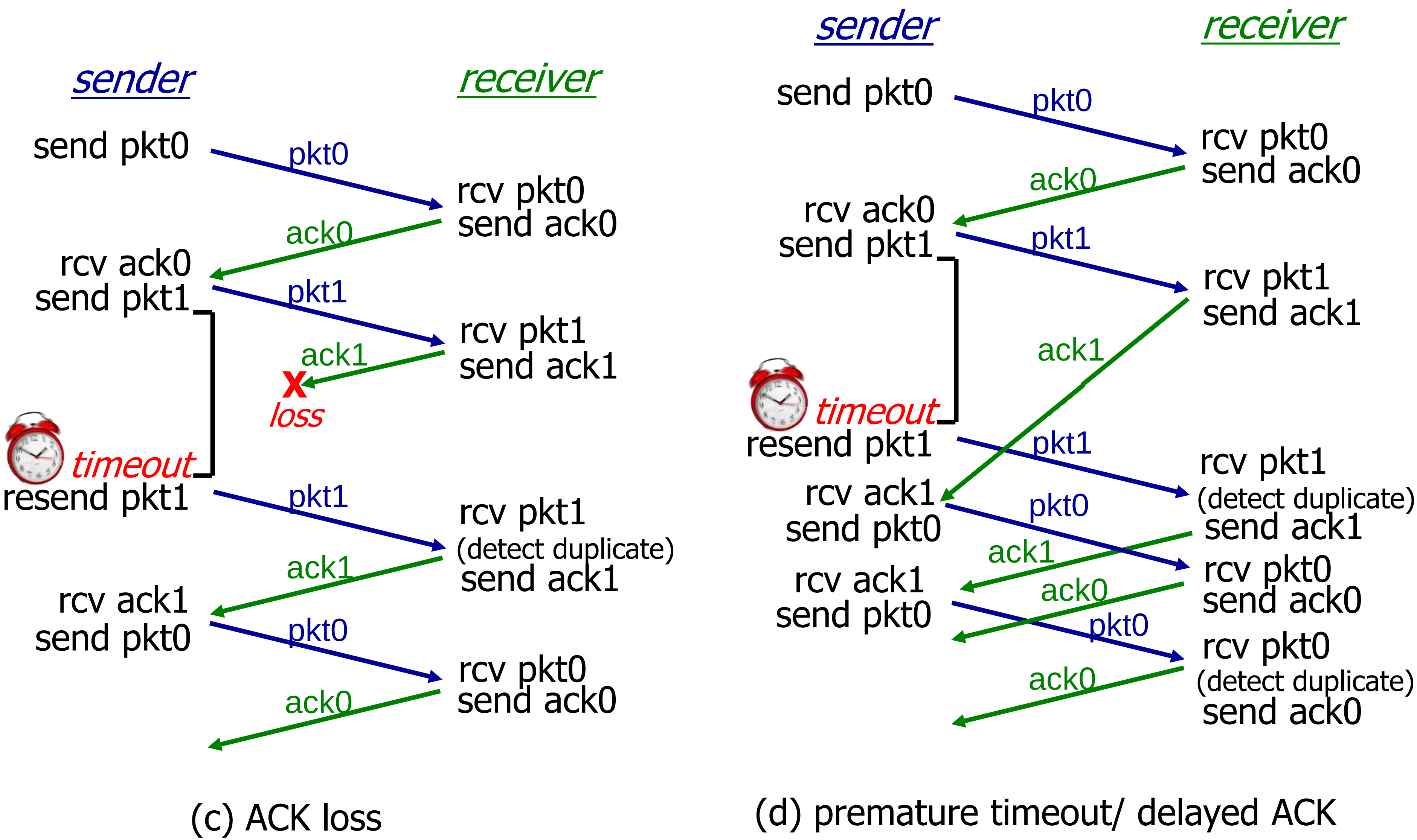
rdt3.0 in action



(a) no loss

(b) packet loss

rdt3.0 in action



Performance of rdt3.0

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

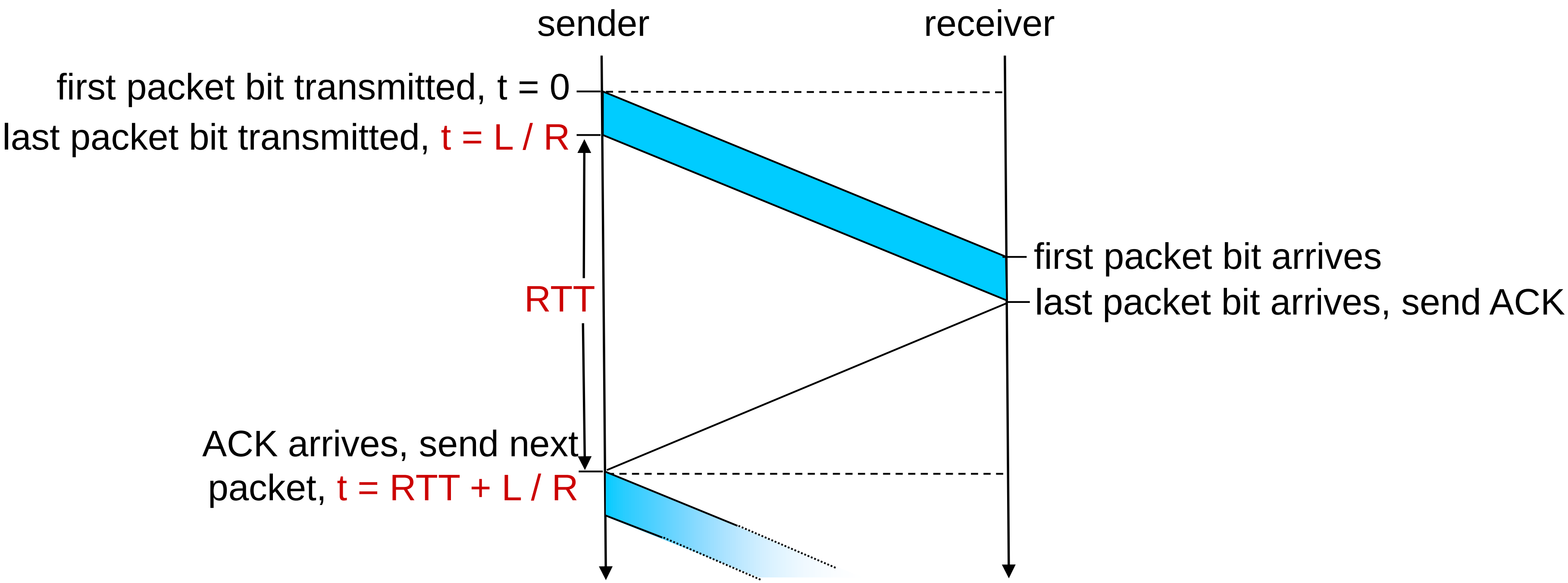
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

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

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

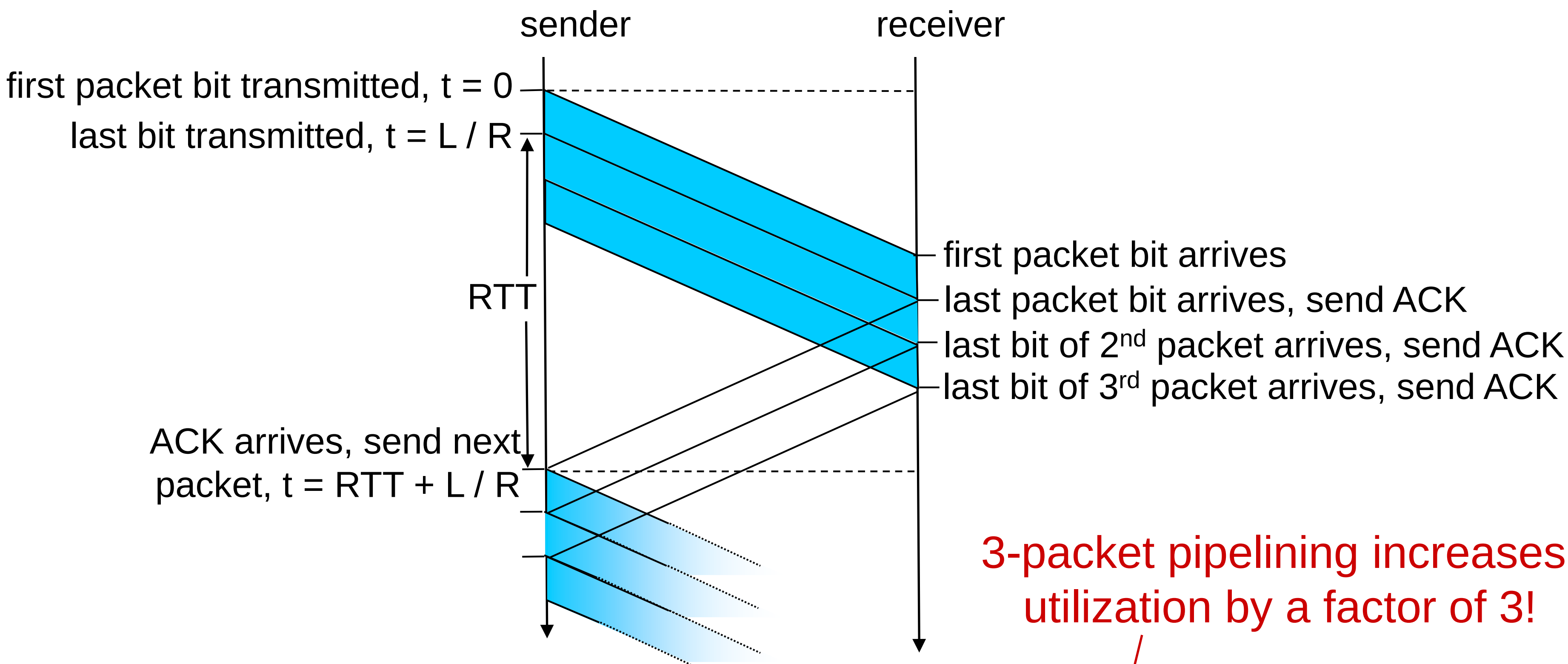
  - if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation



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

Pipelining: increased utilization



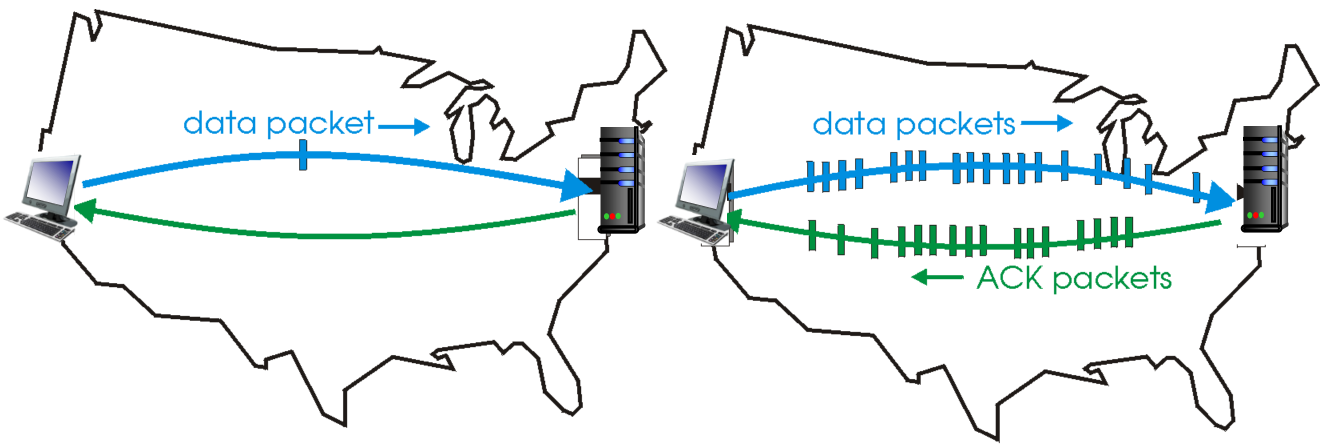
3-packet pipelining increases utilization by a factor of 3!

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

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

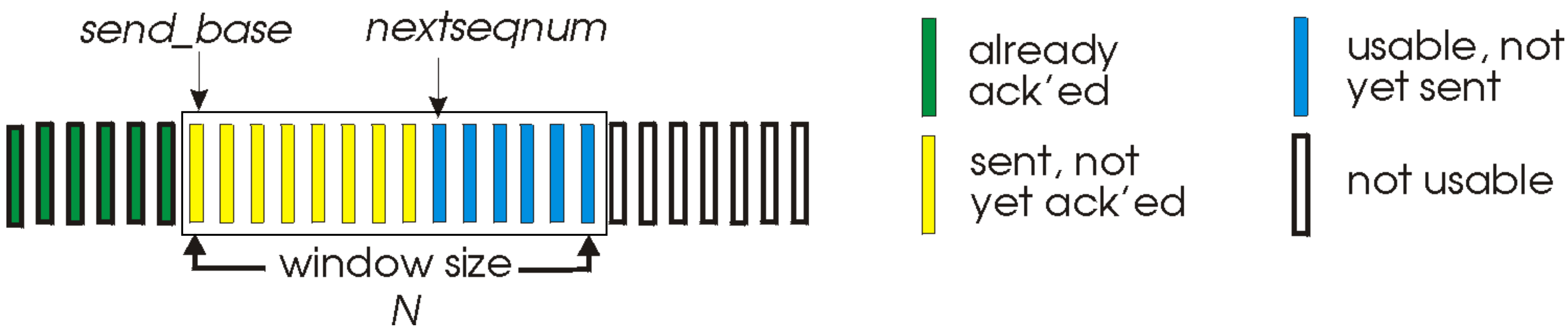


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

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

Go-Back-N: sender

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

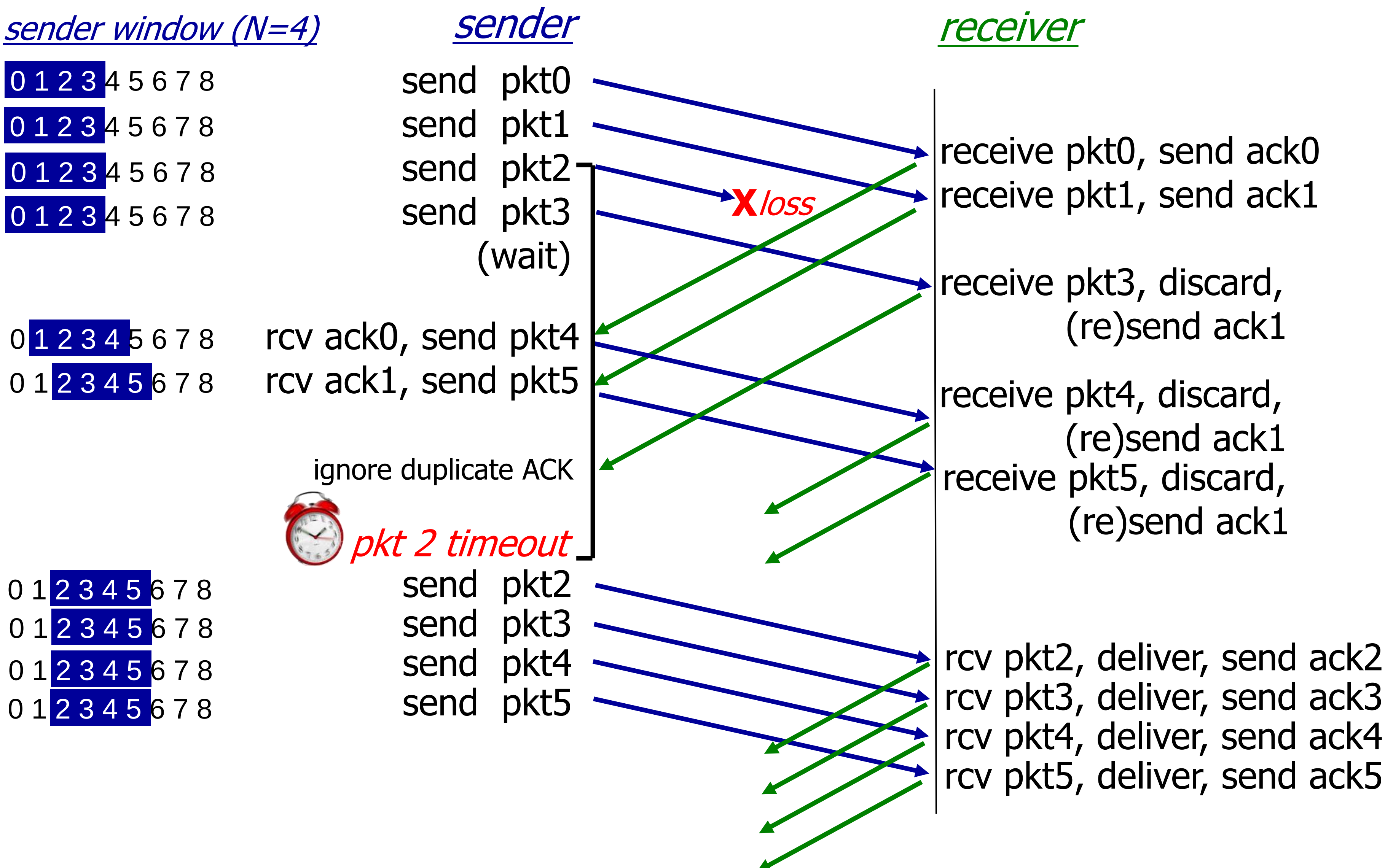


- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

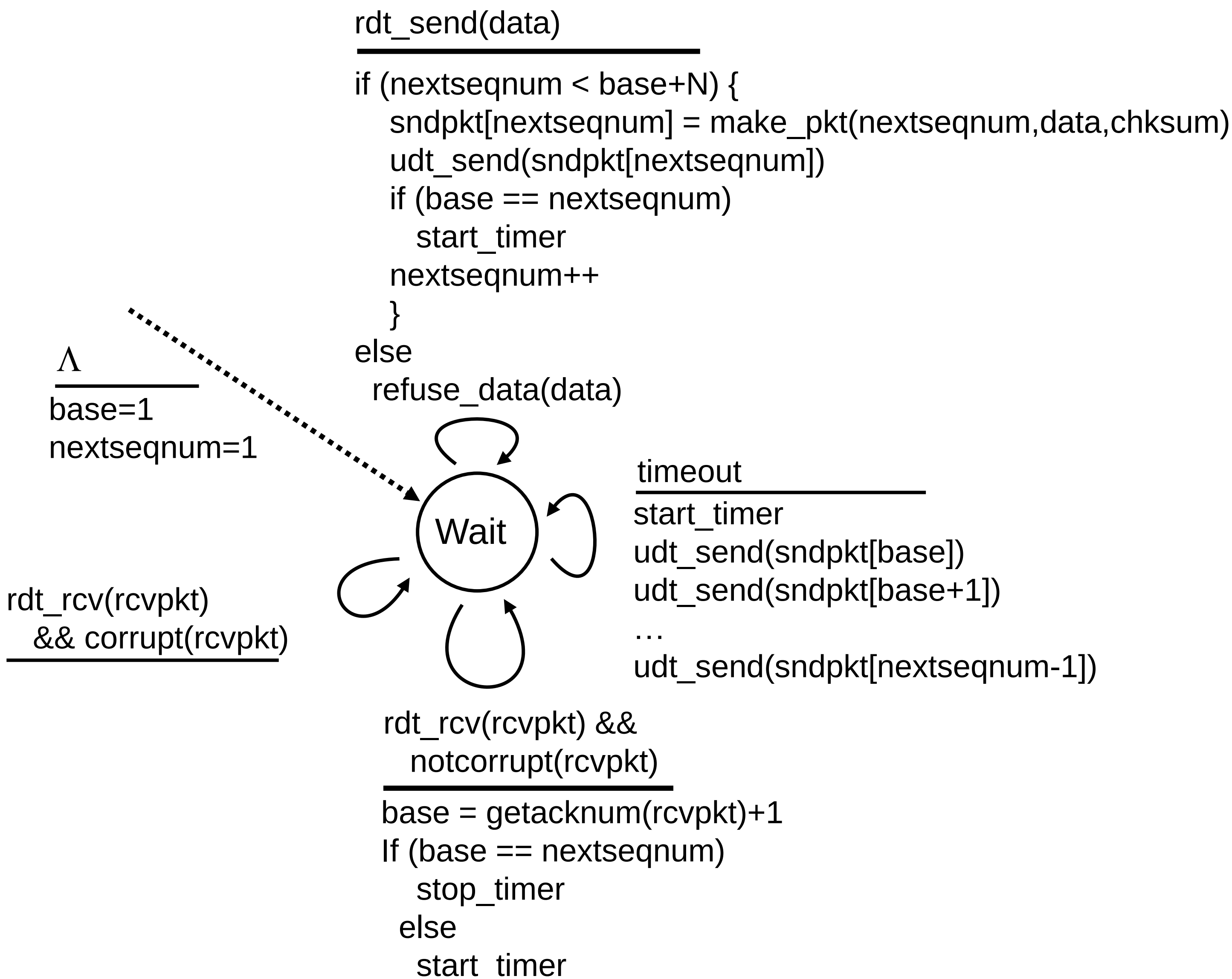
GBN in action

- http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

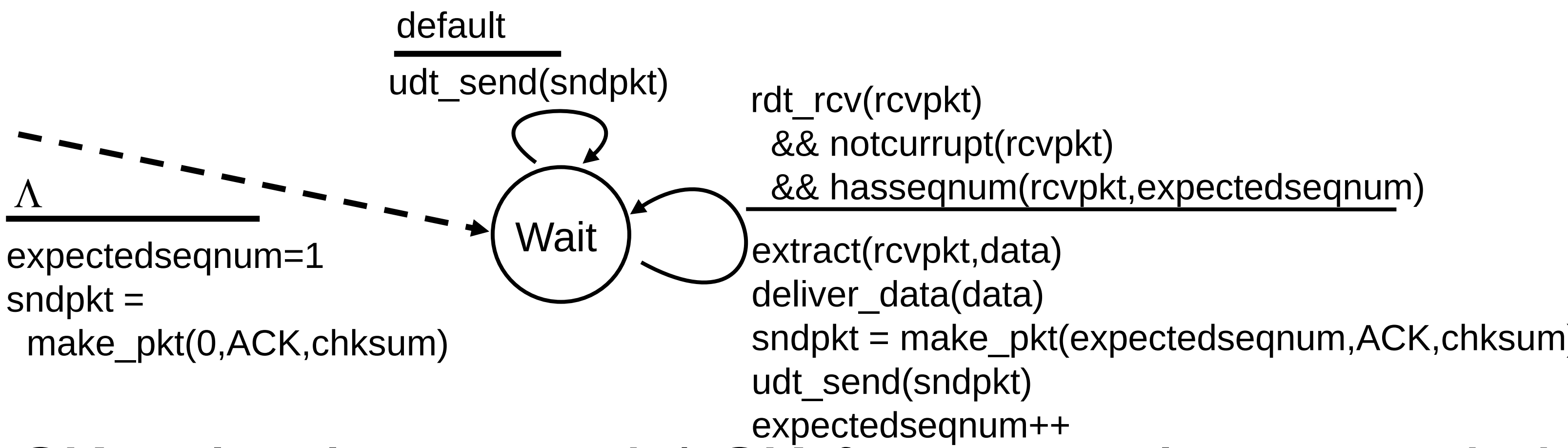
GBN in action



GBN: sender extended FSM



GBN: receiver extended FSM



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

- may generate duplicate ACKs
- Cummulative acknowledgement: if a sender receiver an ACK for seq #x, then all packets less than x-1 is said to have been received.

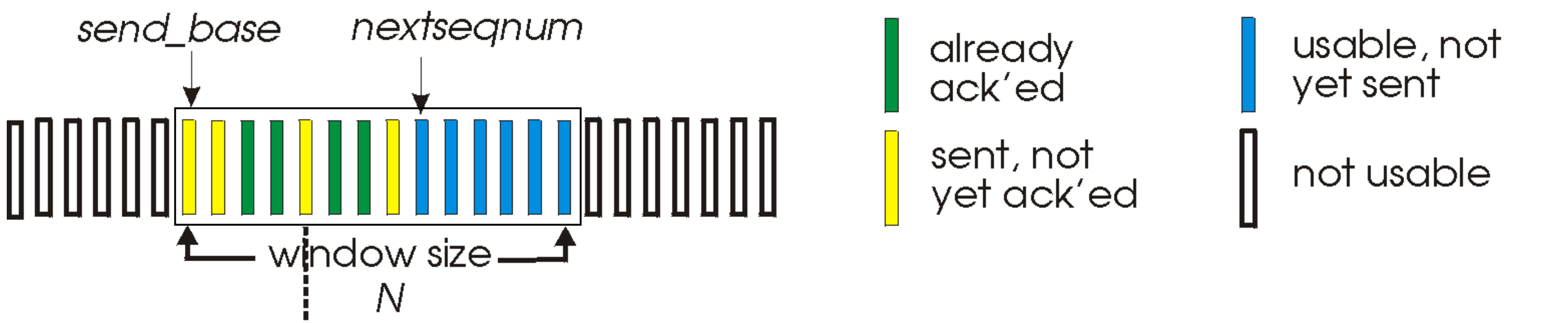
▪ out-of-order pkt:

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

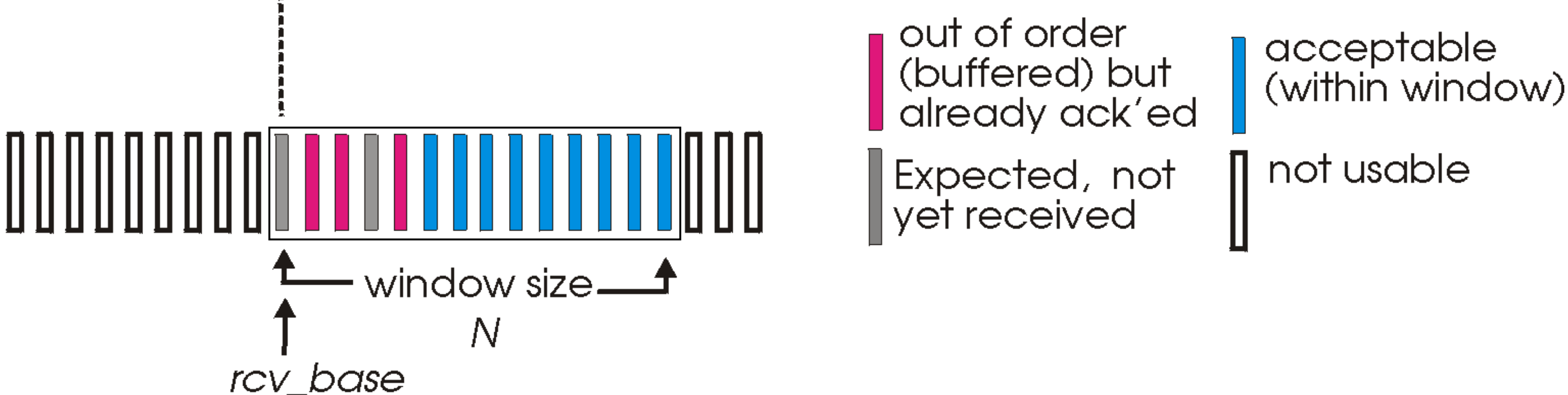
Selective repeat

- 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



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective repeat

sender

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

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

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

receiver

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

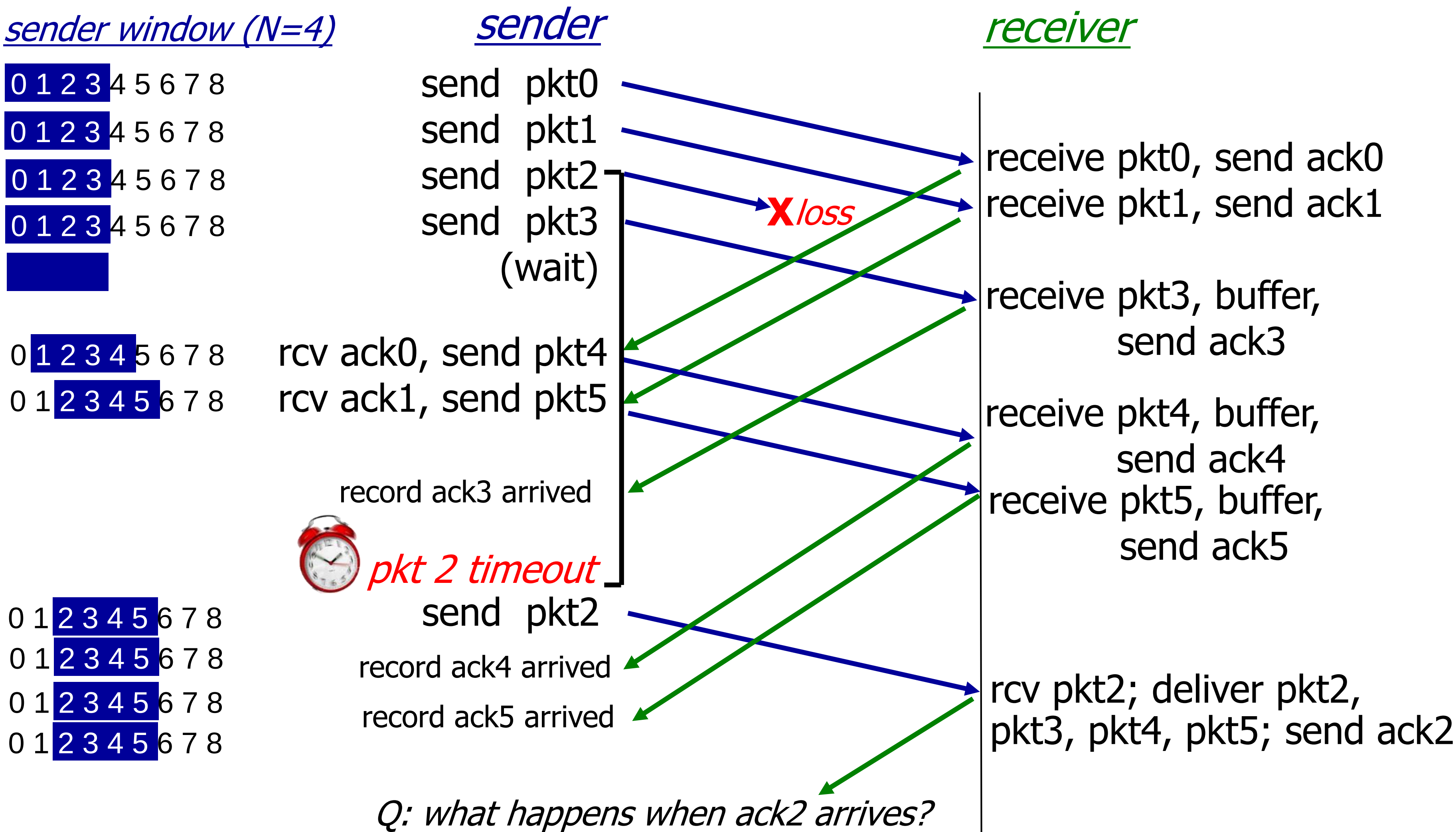
pkt n in [rcvbase-N,rcvbase-1]
- ACK(n)

otherwise:
- ignore

Selective Repeat in action

- http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

Selective repeat in action

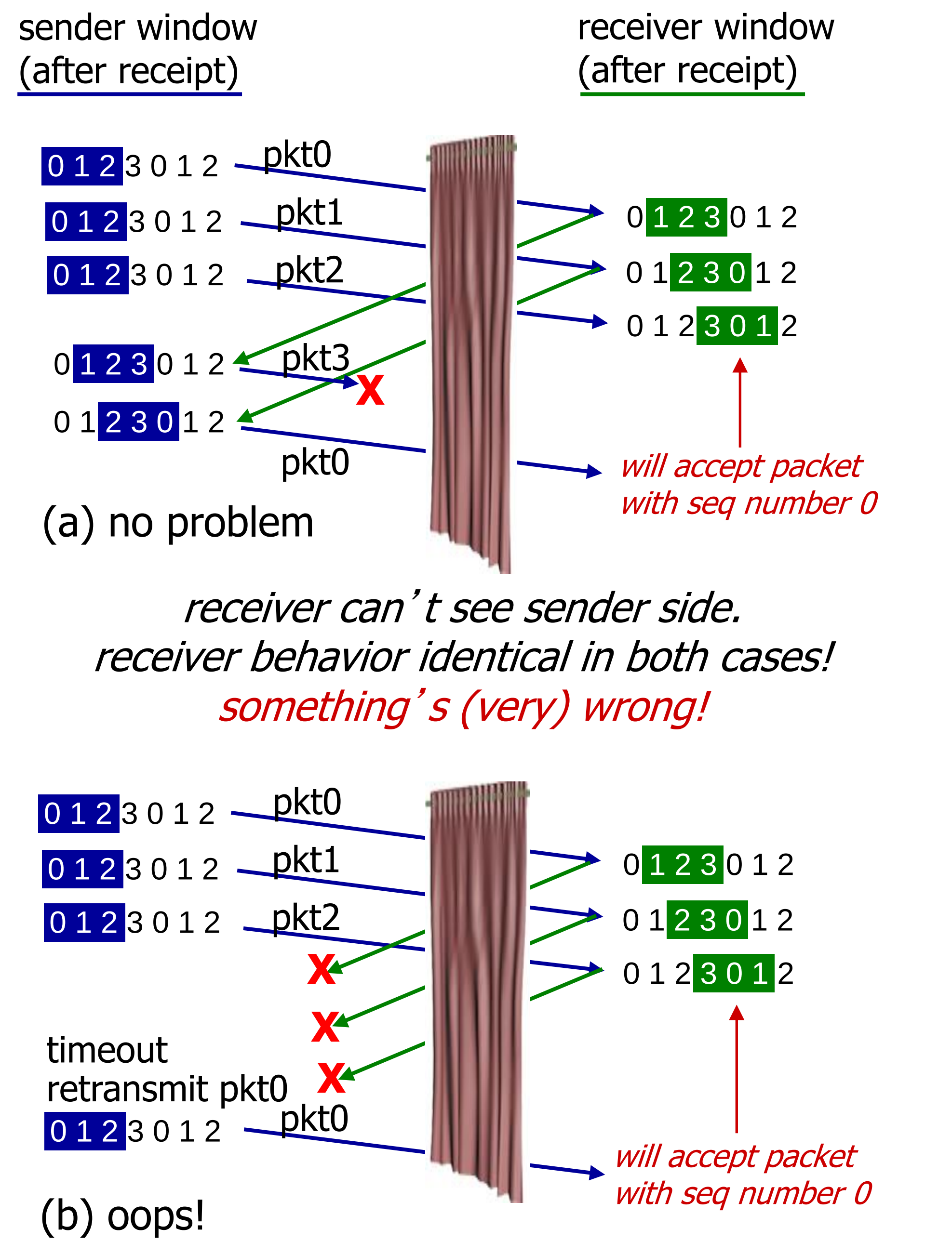


Selective repeat: dilemma

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)?



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 un-ACKed packet
  - when timer expires, retransmit *all* un-ACKed packets

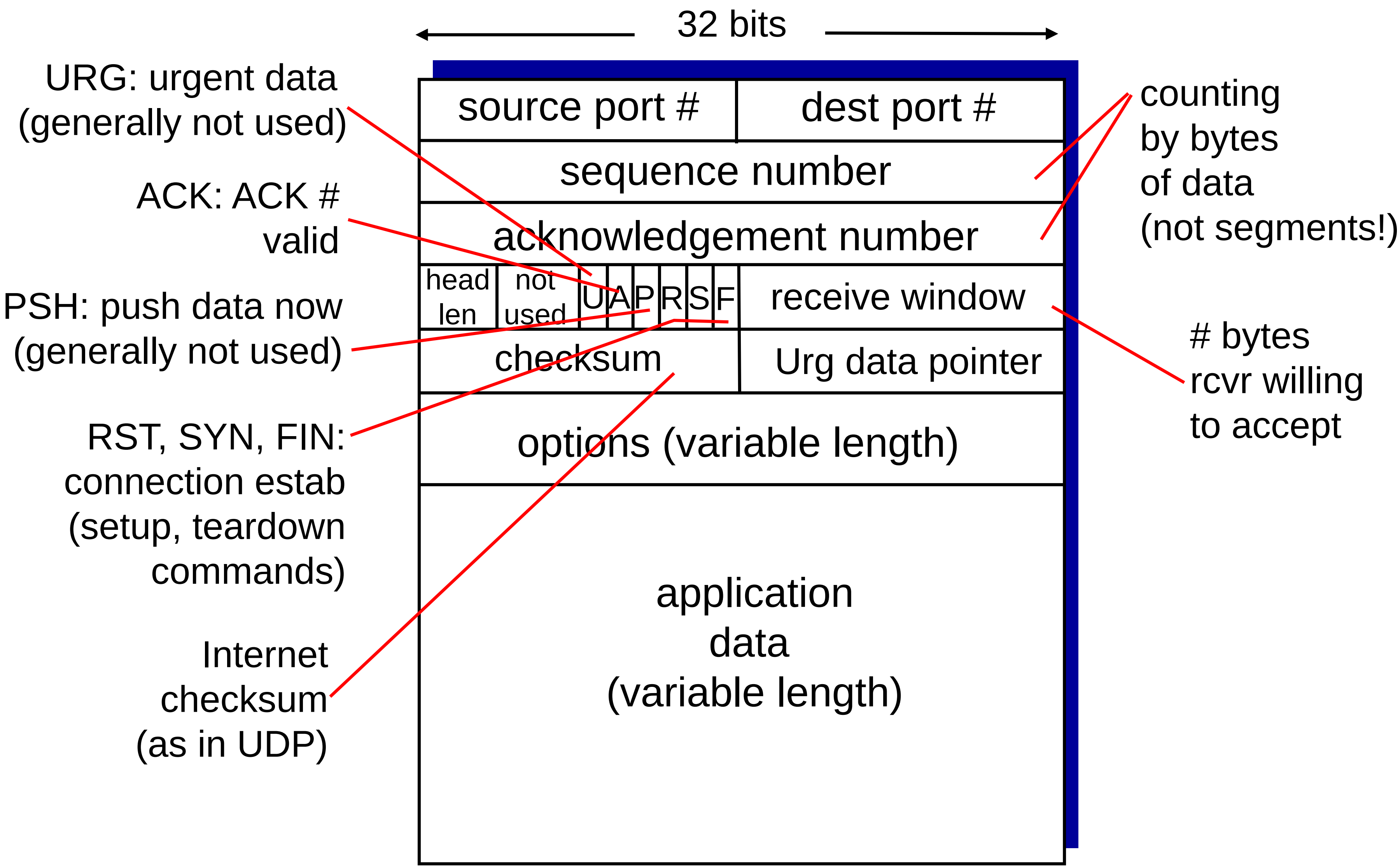
Selective Repeat:

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

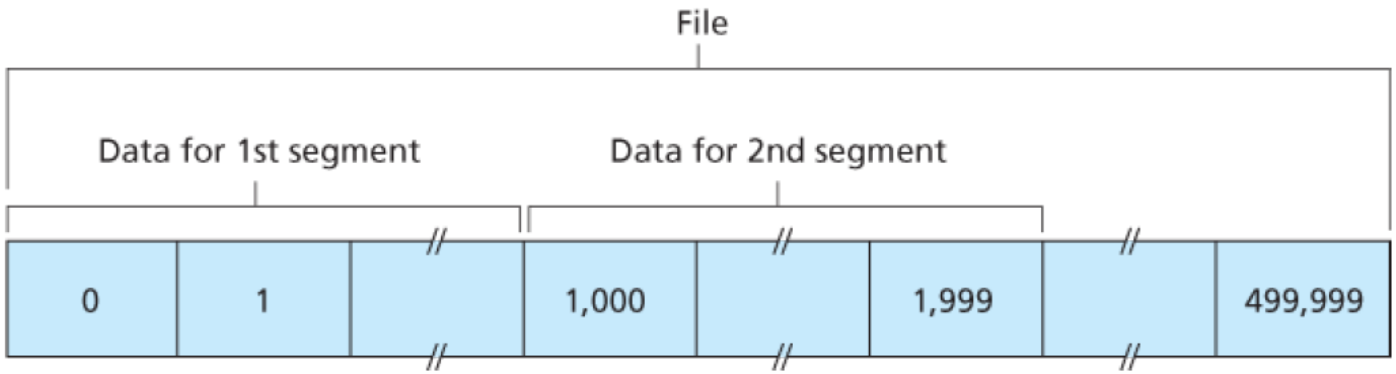
Topics

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

TCP segment structure



Sequence Nos and Acknowledgments



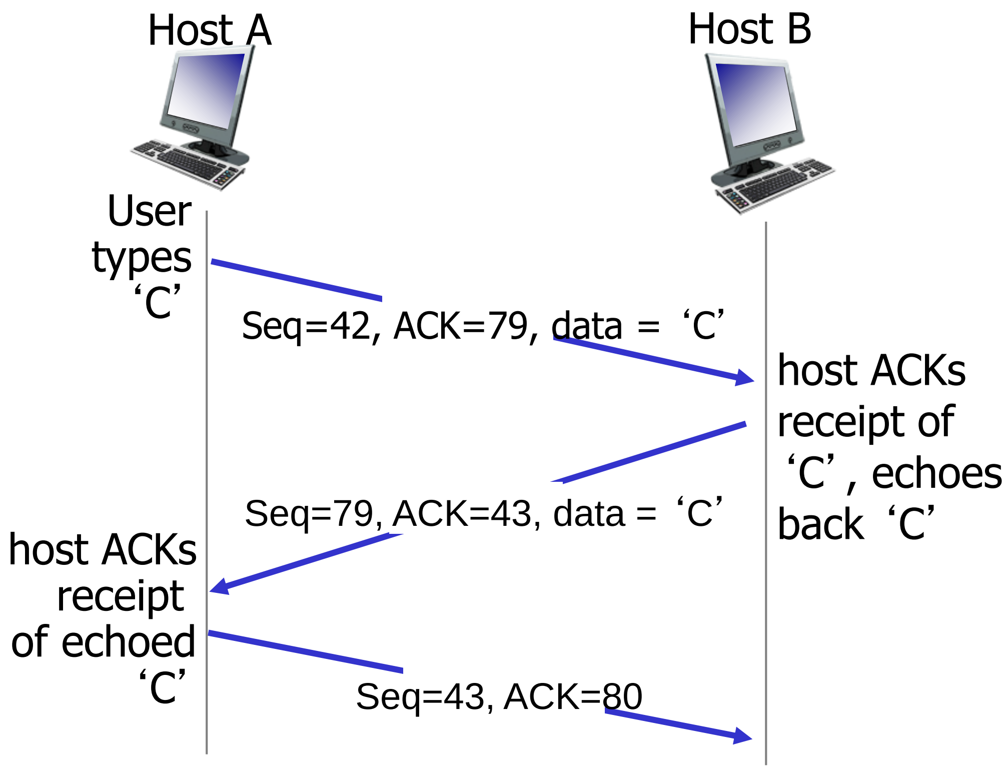
sequence numbers:

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

acknowledgements:

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

TCP seq. numbers, ACKs



simple telnet scenario

TCP seq. numbers, ACKs

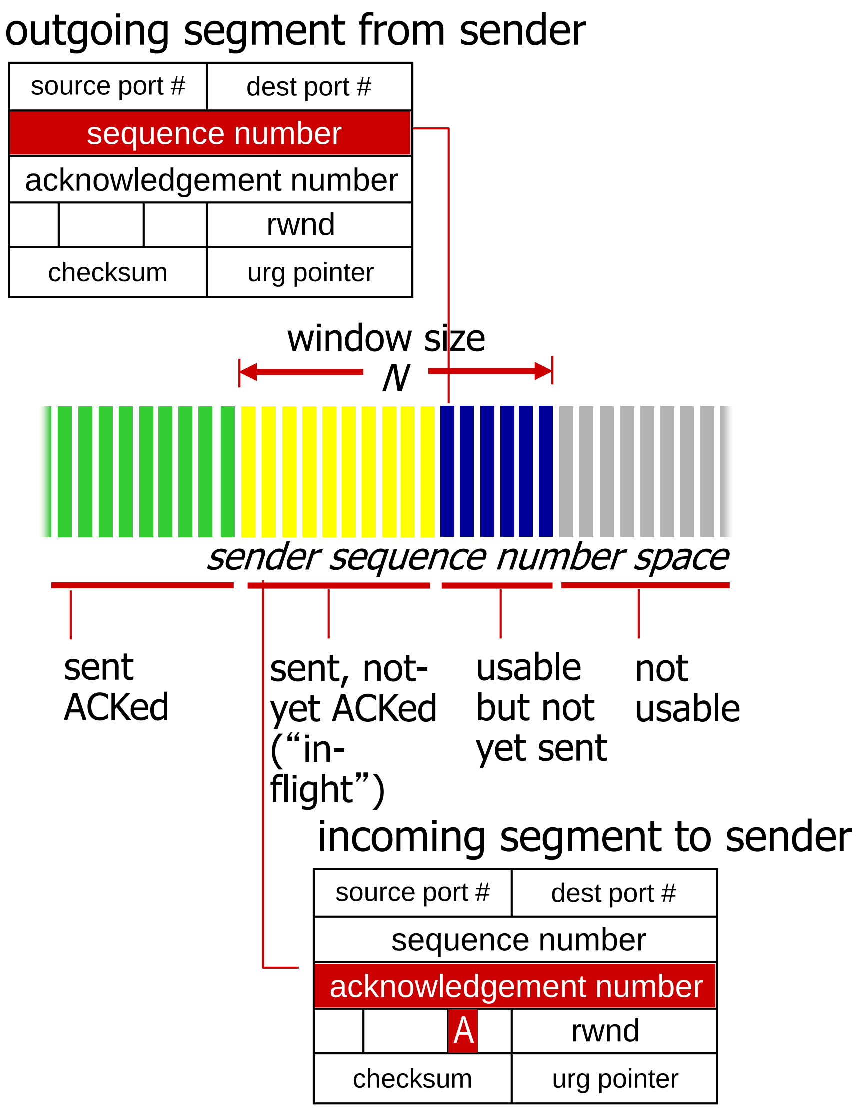
sequence numbers:

- 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



TCP seq. numbers, ACKs

sequence numbers:

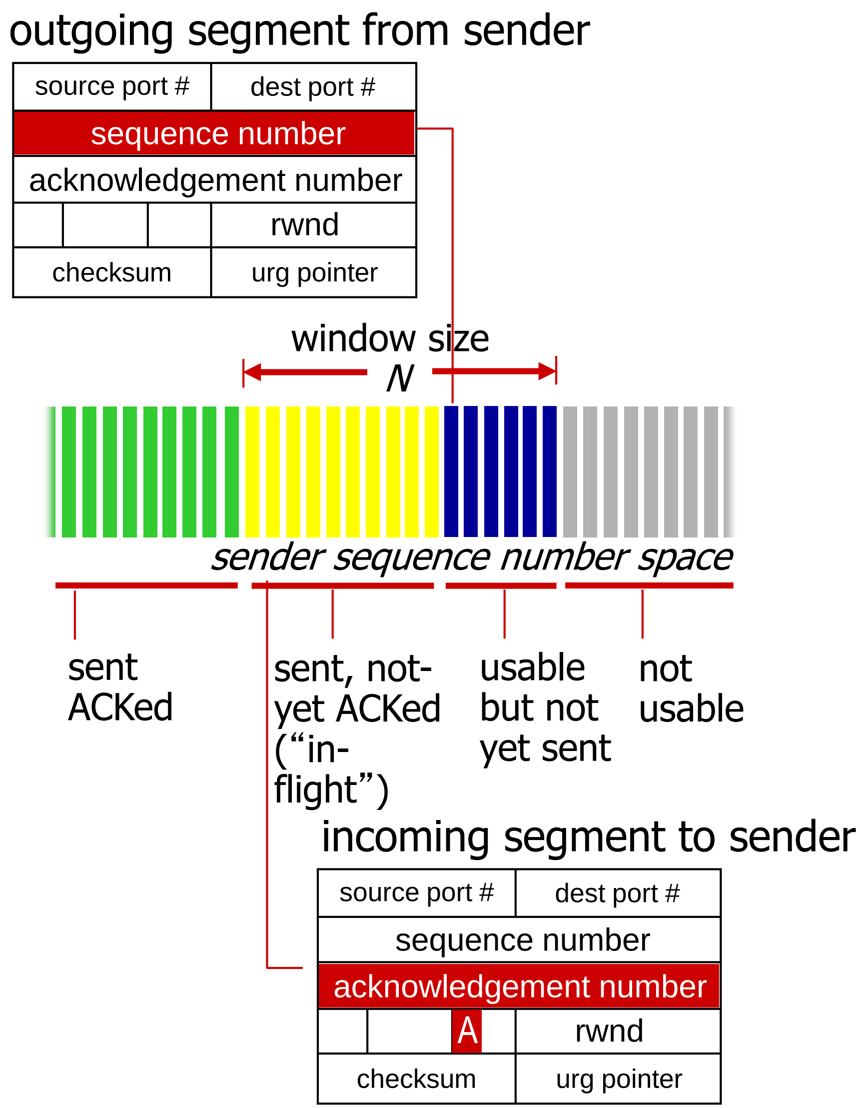
- byte stream "number" of first byte in segment's data

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 round trip time, timeout

Q: how to set TCP timeout value?

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

TCP round trip time, timeout

Q: how to estimate RTT?

TCP round trip time, timeout

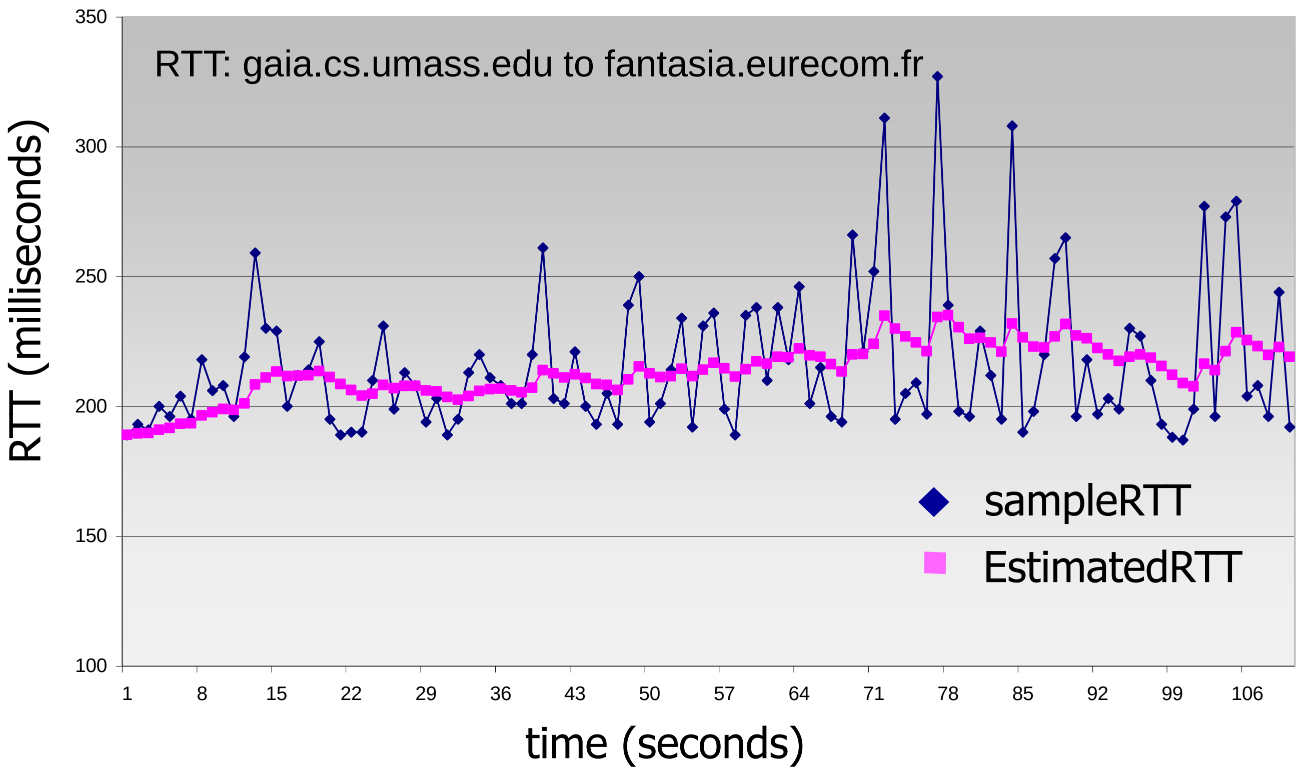
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

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



TCP round trip time, timeout

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

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

(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/

TCP round trip time, timeout

Suppose that TCP's current estimated values for the round trip time (`estimatedRTT`) and deviation in the RTT (`DevRTT`) are 360 msec and 39 msec, respectively (see Section 3.5.3 for a discussion of these variables). Suppose that the next three measured values of the RTT are 260, 340, and 260 respectively.

Compute TCP's new value of `estimatedRTT, DevRTT`, and the TCP timeout value after each of these three measured RTT values is obtained.Use the values of $\alpha = 0.125$ and $\beta = 0.25$.

TCP round trip time, timeout cont.

Solution:

After the first RTT estimate is made:

```
estimatedRTT = 0.875*360 + 0.125*260 = 347.5 ms
DevRTT = 0.75*39 + 0.25*(abs(260 - 347.5)) = 51.125 ms
TimeoutInterval = 347.5 + 4*51.125 = 552 ms
```

TCP round trip time, timeout cont.

Solution:

After the first RTT estimate is made:

```
estimatedRTT = 0.875*360 + 0.125*260 = 347.5 ms
DevRTT = 0.75*39 + 0.25*(abs(260 - 347.5)) = 51.125 ms
TimeoutInterval = 347.5 + 4*51.125 = 552 ms
```

After the second RTT estimate is made:

```
estimatedRTT = 0.875*347.5 + 0.125*340 = 346.5625 ms
DevRTT = 0.75*51.125 + 0.25*(abs(340 - 346.5625)) =
39.984375 ms
TimeoutInterval = 346.5625 + 4*39.984375 = 506.5 ms
```

TCP round trip time, timeout cont.

Solution:

After the first RTT estimate is made:

```
estimatedRTT = 0.875*360 + 0.125*260 = 347.5 ms
DevRTT = 0.75*39 + 0.25*(abs(260 - 347.5)) = 51.125 ms
TimeoutInterval = 347.5 + 4*51.125 = 552 ms
```

After the second RTT estimate is made:

```
estimatedRTT = 0.875*347.5 + 0.125*340 = 346.5625 ms
DevRTT = 0.75*51.125 + 0.25*(abs(340 - 346.5625)) = 
39.984375 ms
TimeoutInterval = 346.5625 + 4*39.984375 = 506.5 ms
```

After the third RTT estimate is made:

```
estimatedRTT = 0.875*346.5625 + 0.125*340 = 335.7421875 ms
DevRTT = 0.75*39.984375 + 0.25*(abs(260 - 335.7421875)) = 
39.984375 ms
TimeoutInterval = 335.7421875 + 4*48.923828125 = 531.4375 ms
```

TCP reliable data transfer

- TCP creates reliable data transfer 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) events:

data rcvd from app:

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

timeout:

- retransmit segment that caused timeout
- restart timer

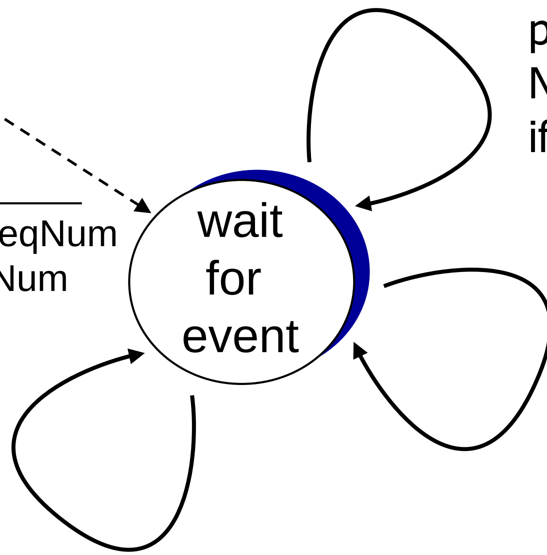
ack rcvd:

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

TCP sender (simplified)

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



data received from application above

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

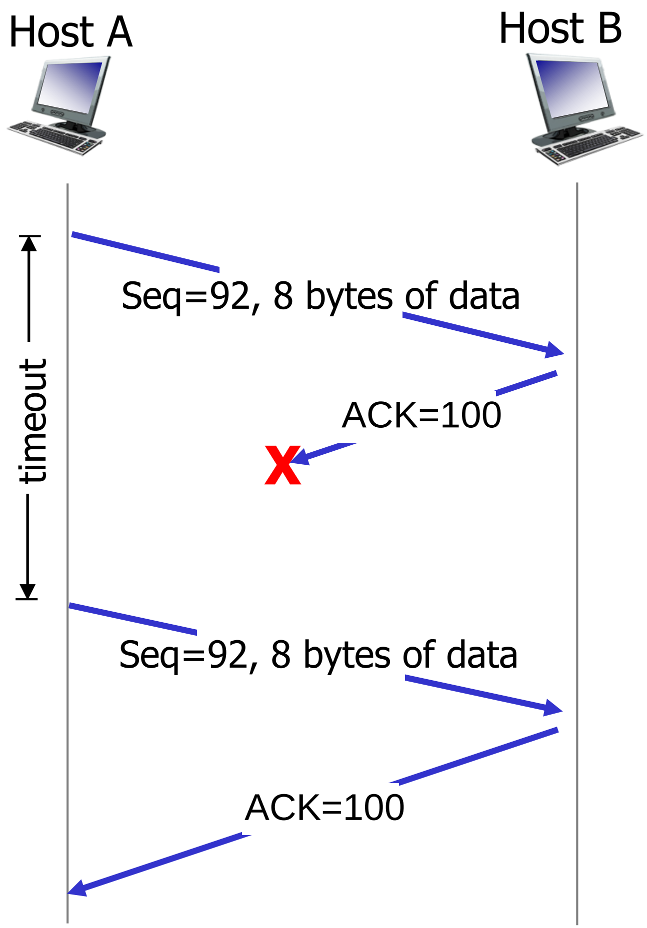
timeout

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

ACK received, with ACK field value y

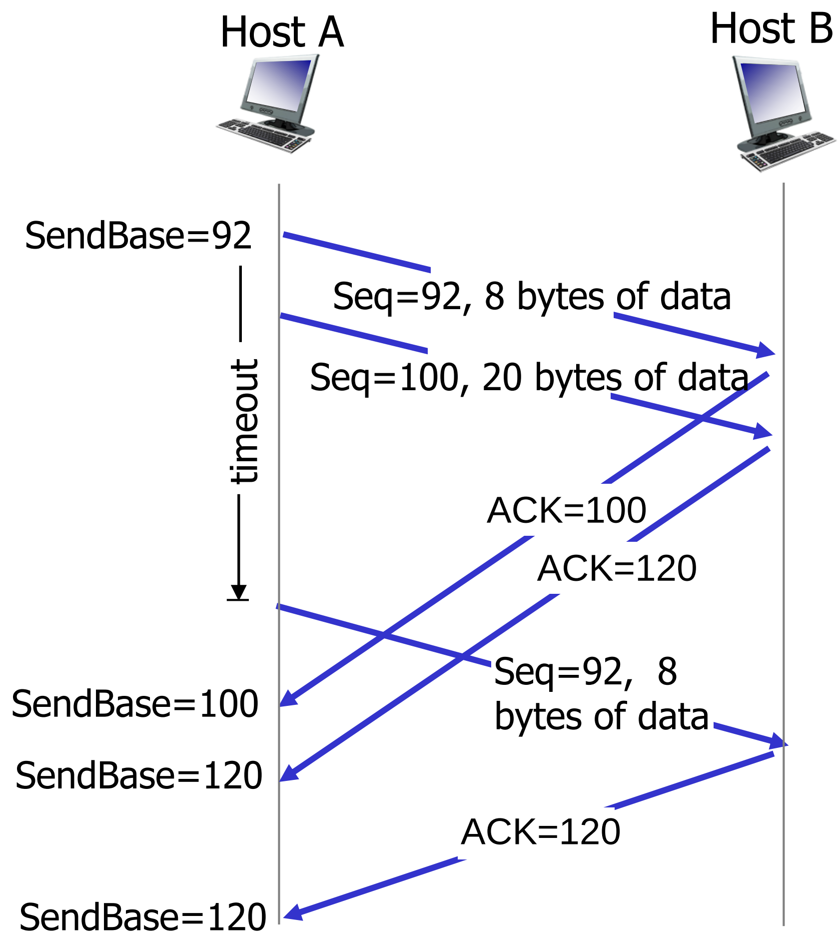
```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
 }
```

TCP: retransmission scenarios



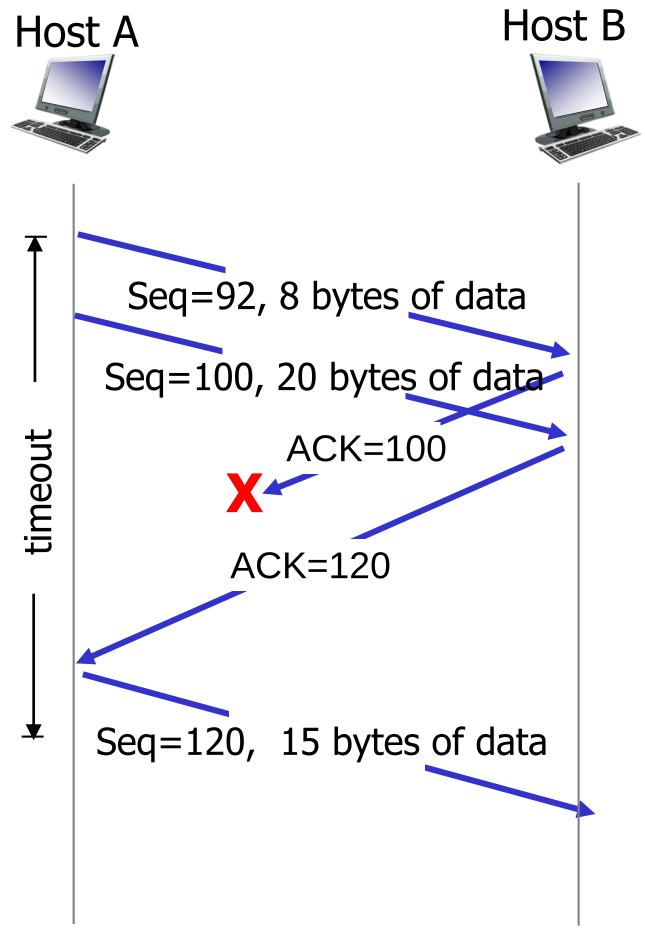
lost ACK scenario

TCP: retransmission scenarios



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

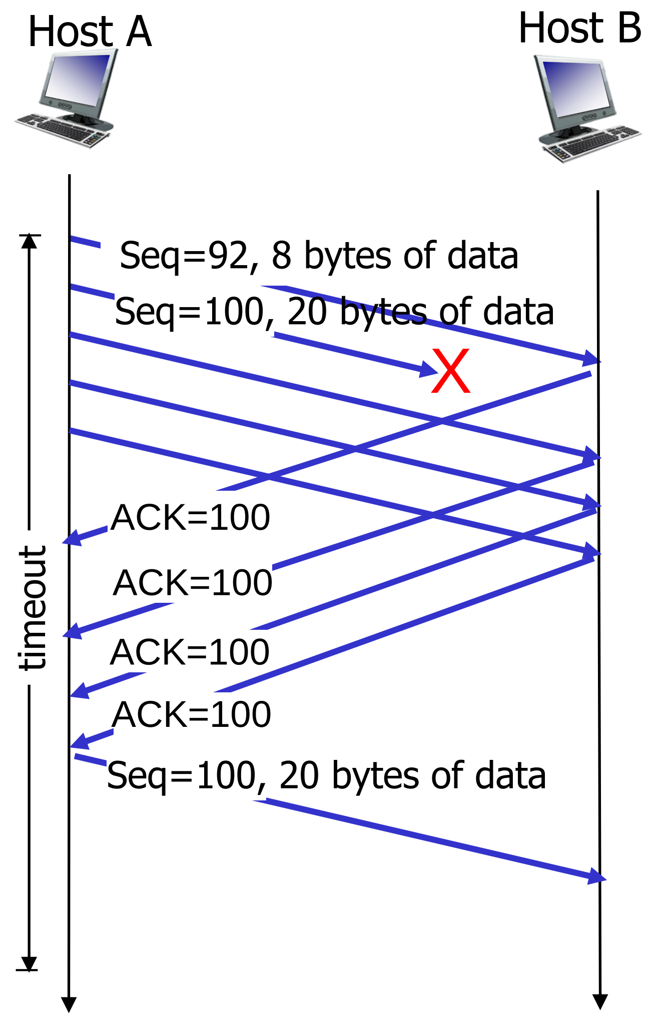
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

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

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

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

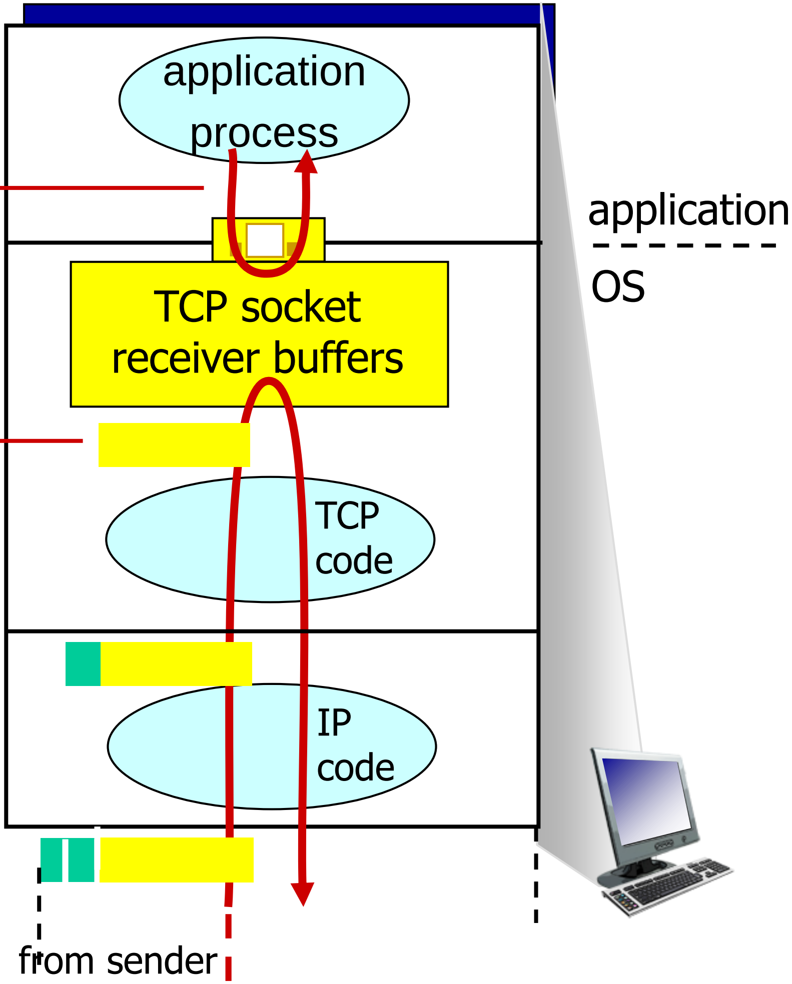
Topics

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

TCP flow control

application may
remove data from
TCP socket buffers ….

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



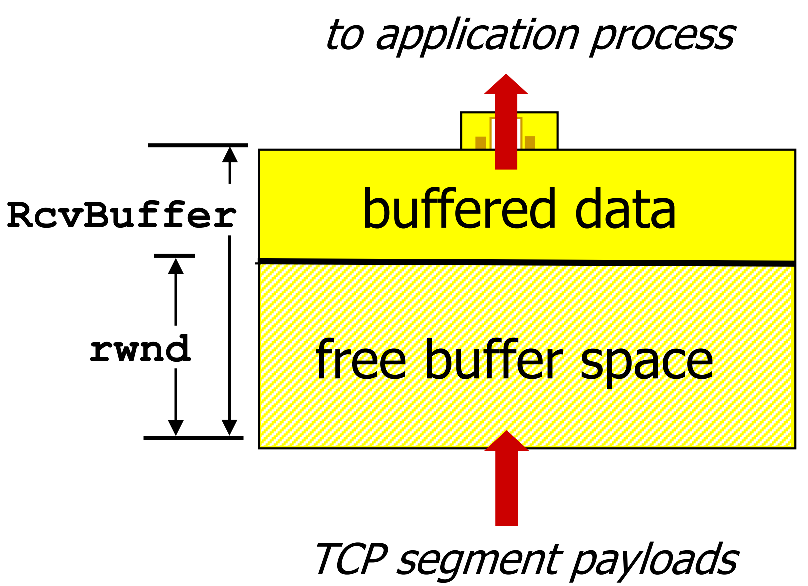
receiver protocol stack

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 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

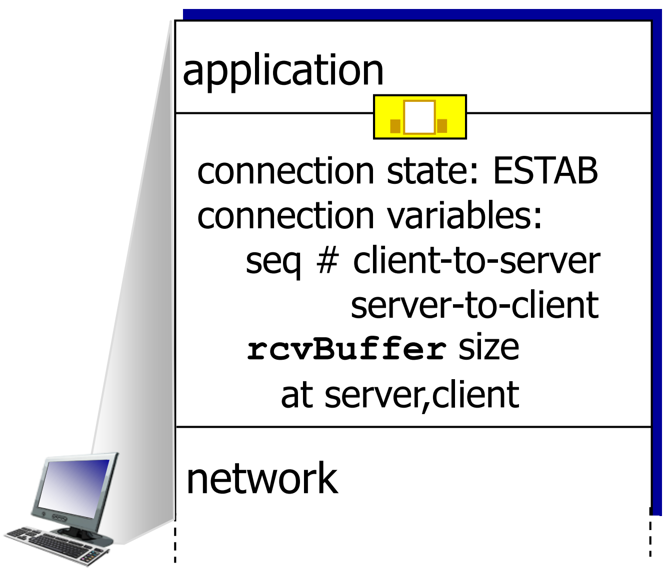
Topics

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

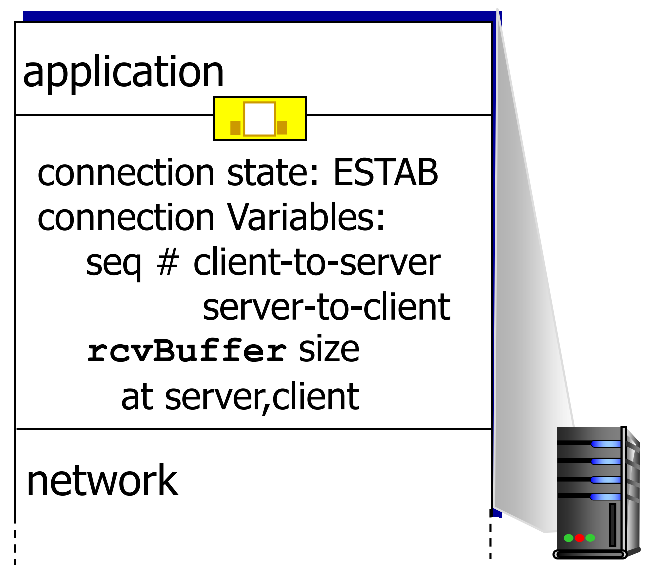
Connection Management

before exchanging data, sender/receiver "handshake":

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



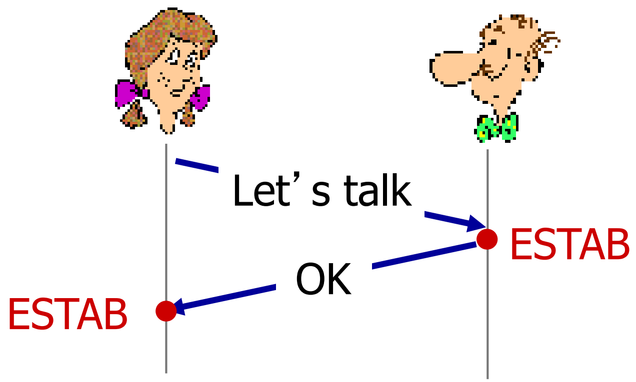

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

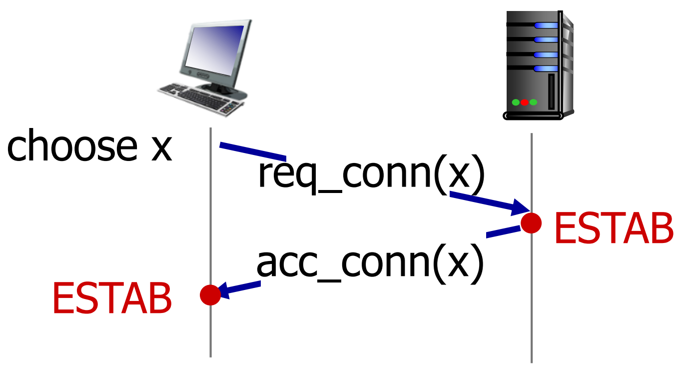



```
Socket connectionSocket =
   welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:



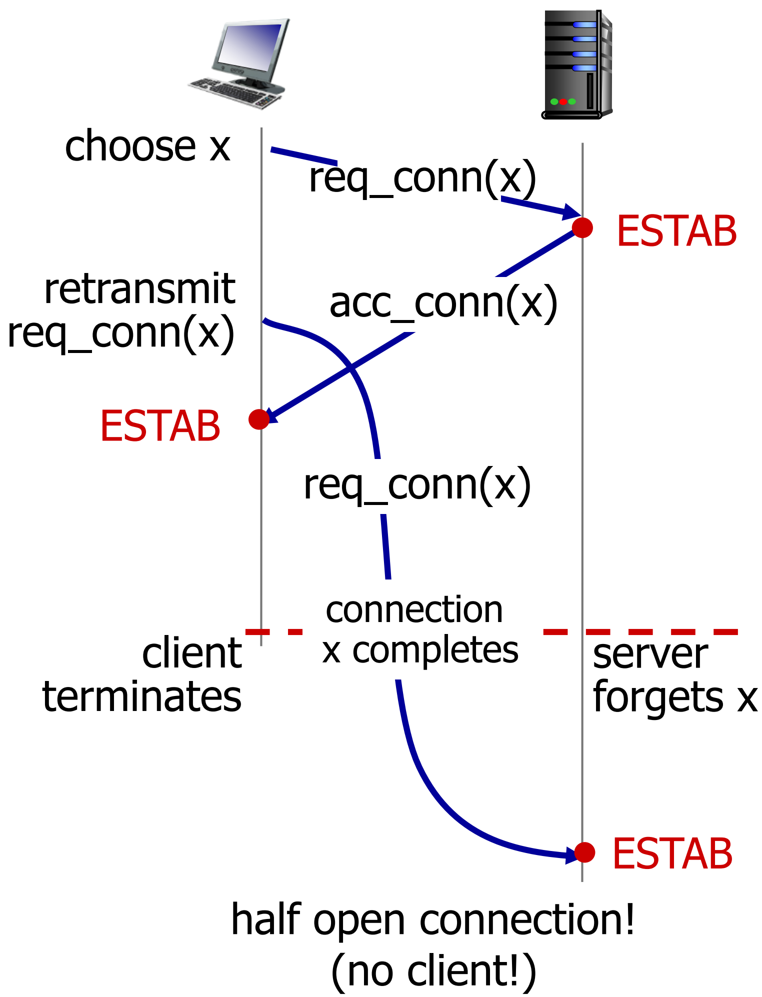


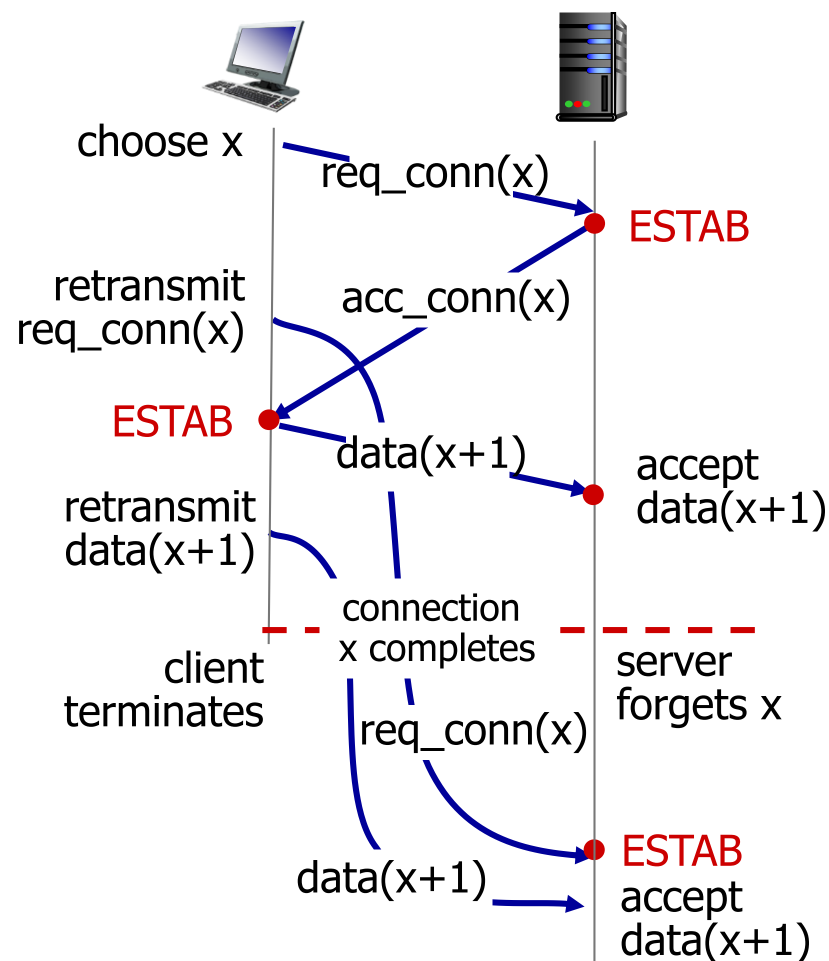
Q: will 2-way handshake always work in network?

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

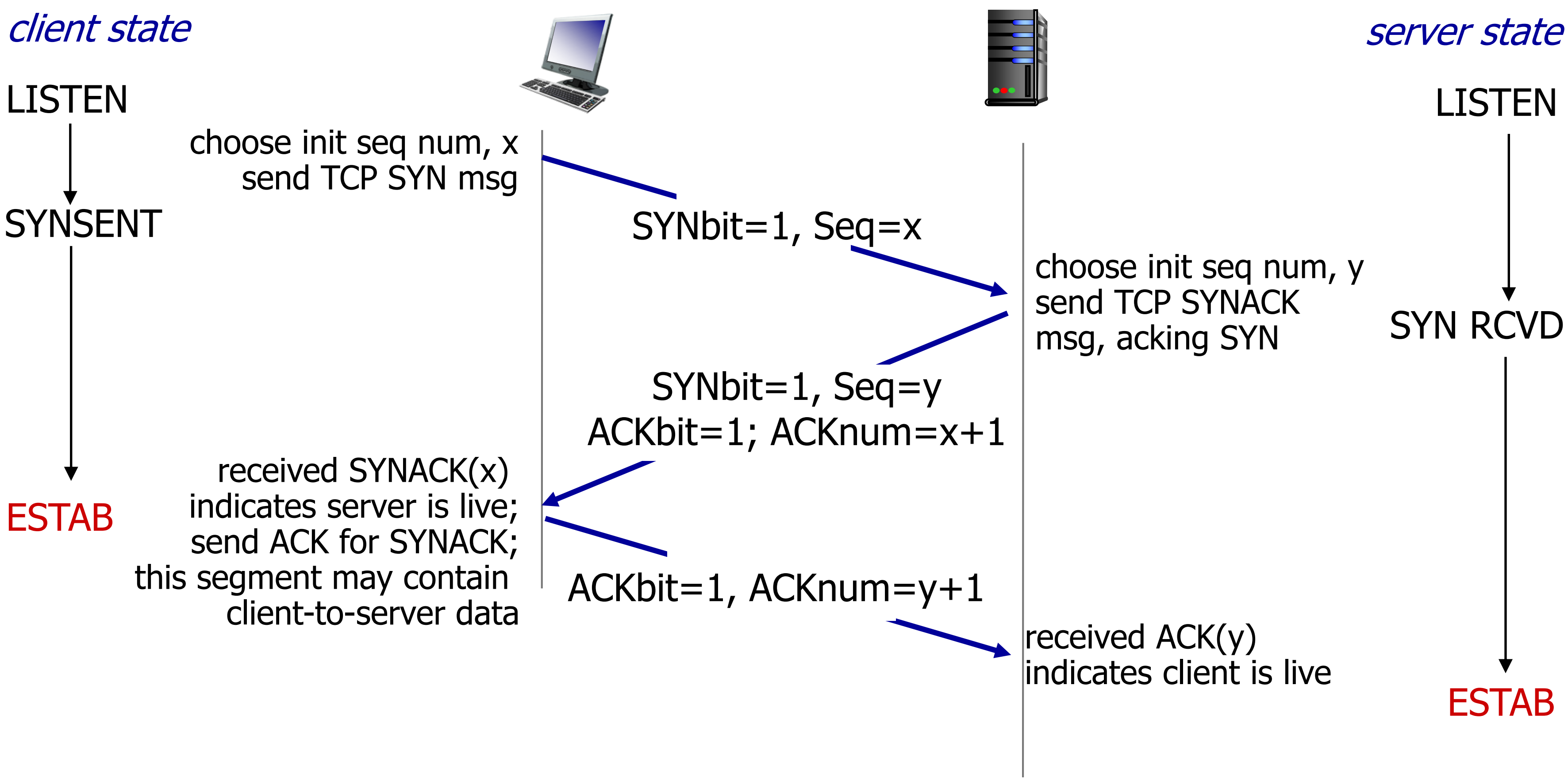
Agreeing to establish a connection

2-way handshake failure scenarios:





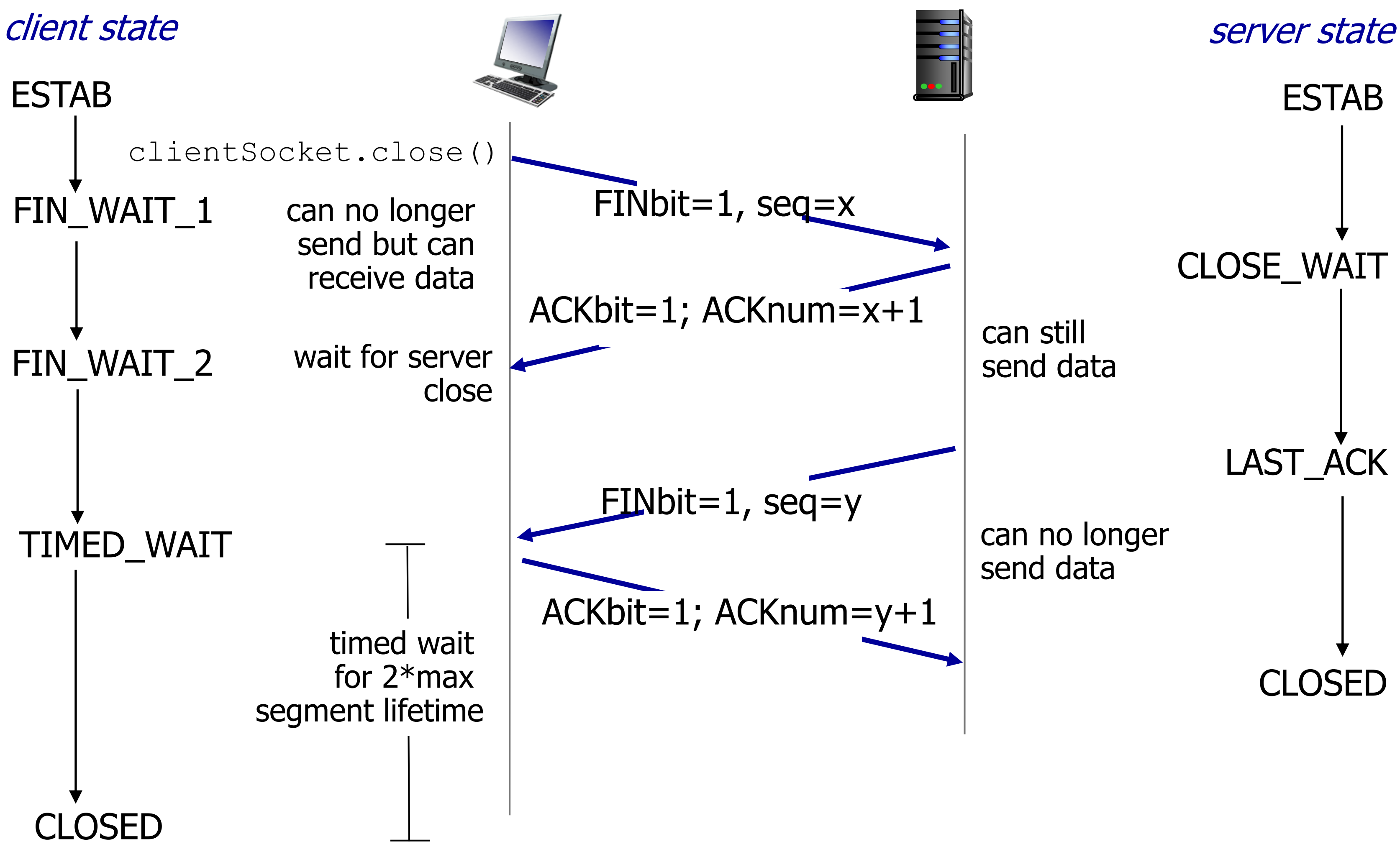
TCP 3-way handshake



TCP: 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



Topics

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

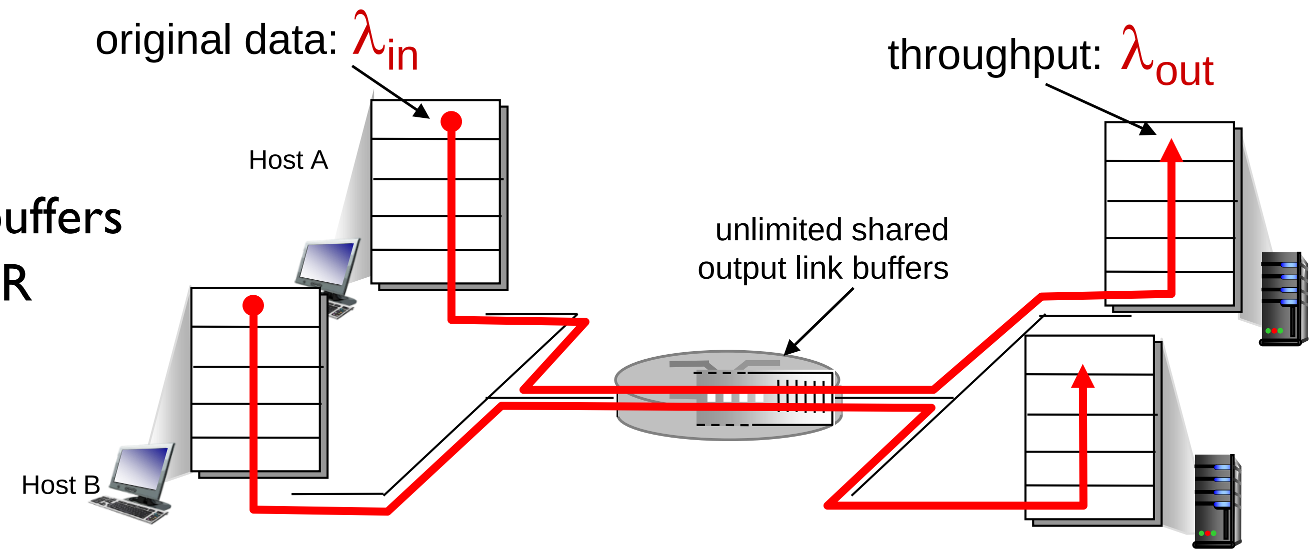
Principles of congestion control

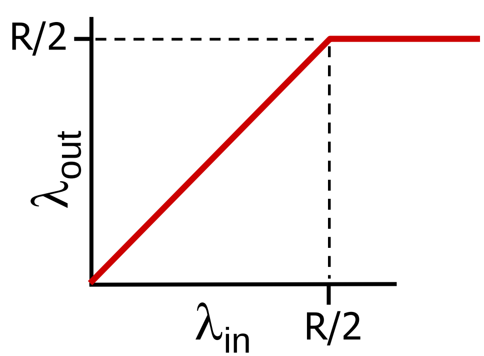
congestion:

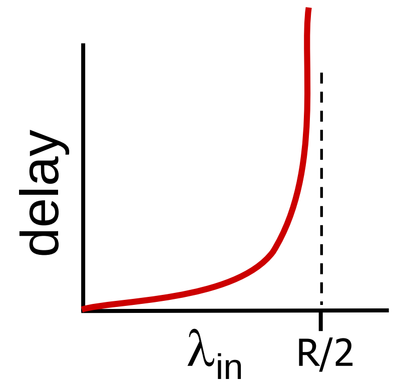
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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





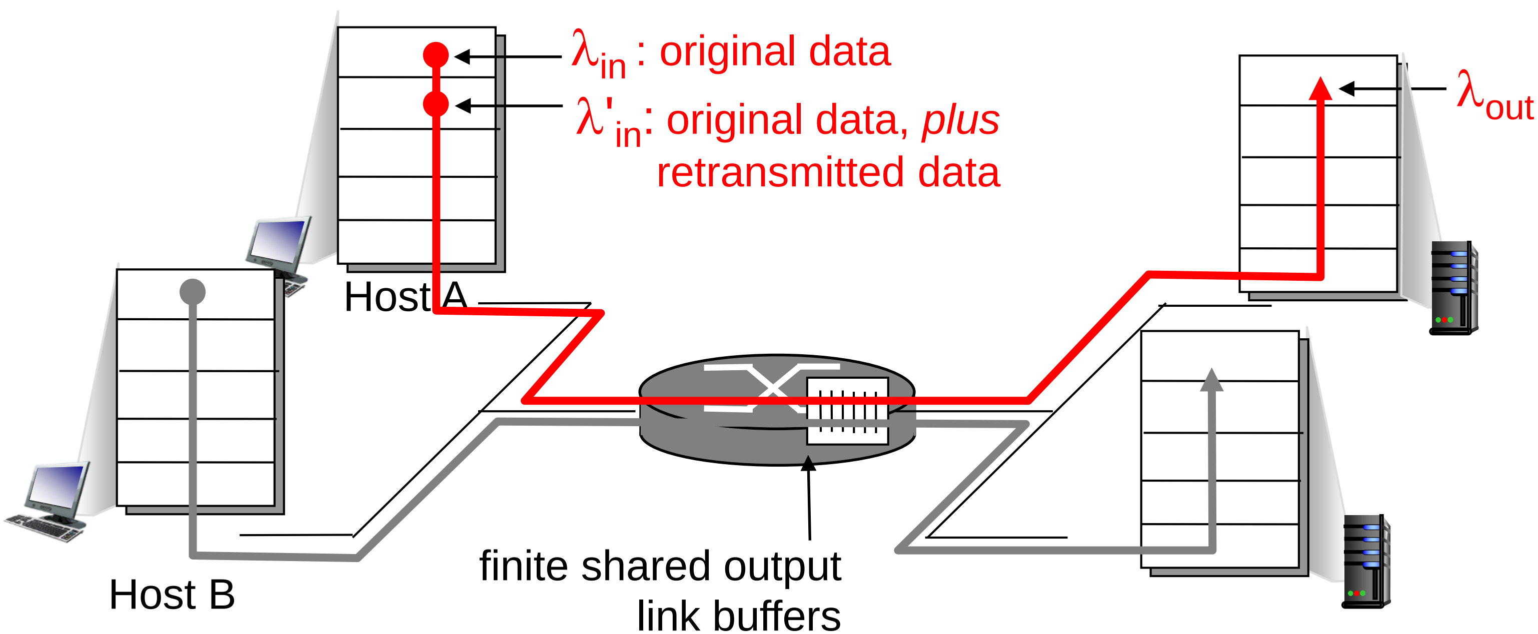


- maximum per-connection throughput: R/2

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

Causes/costs of congestion: scenario 2

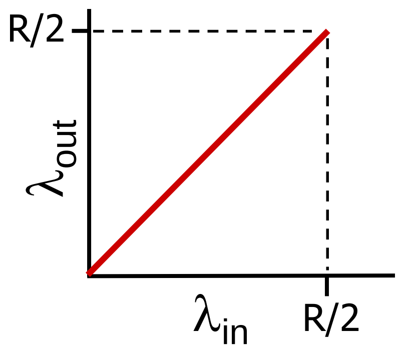
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

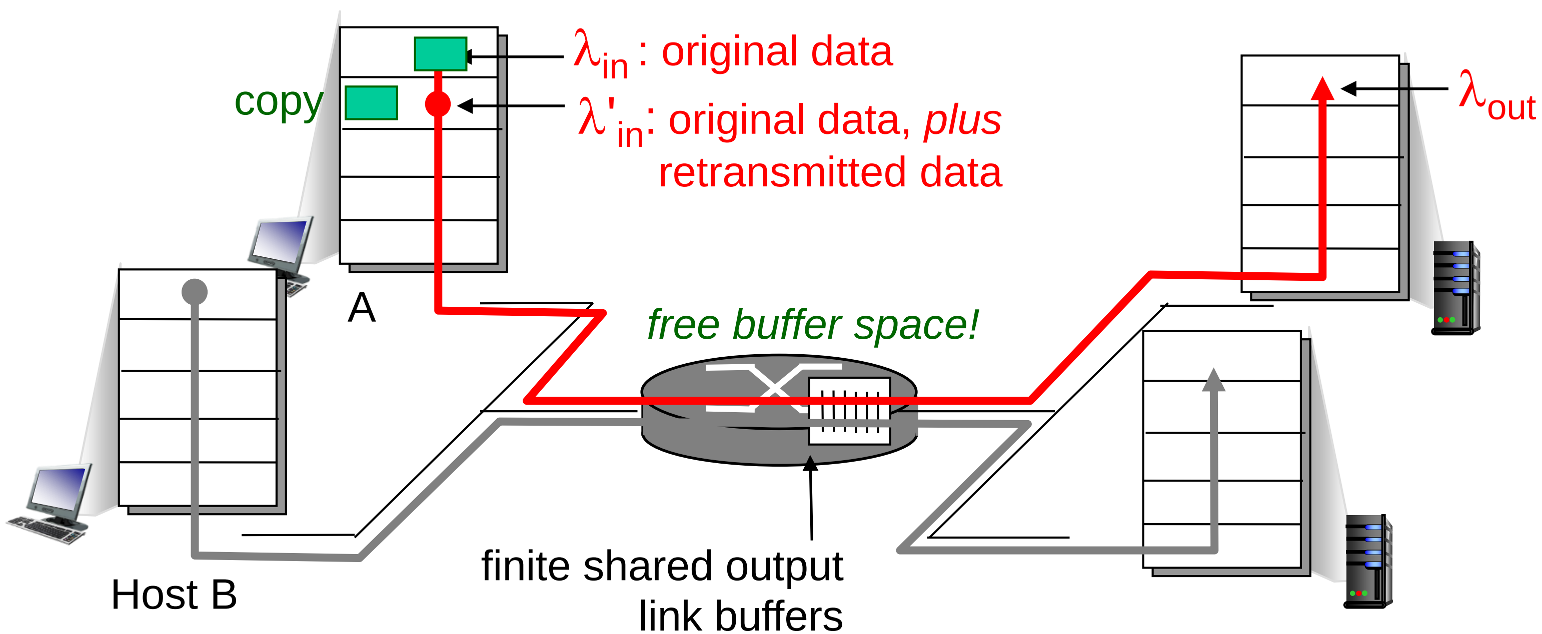


Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

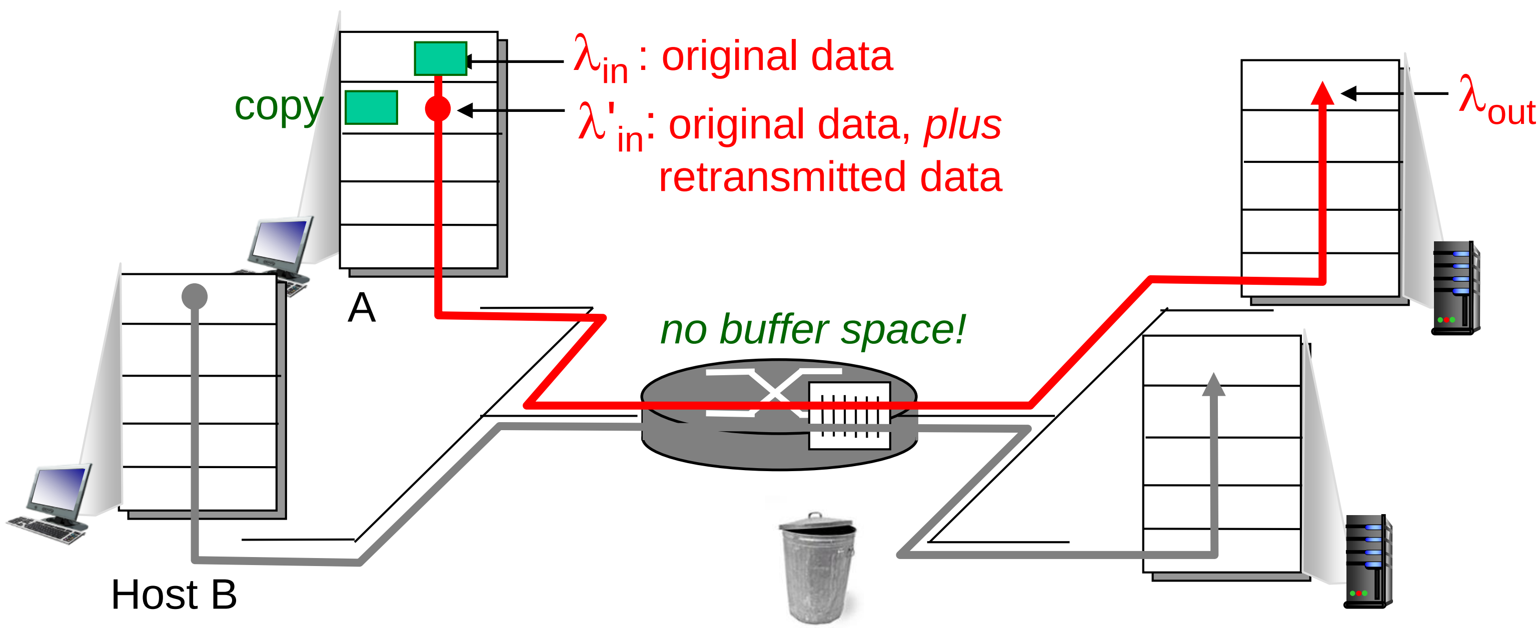




Causes/costs of congestion: scenario 2

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

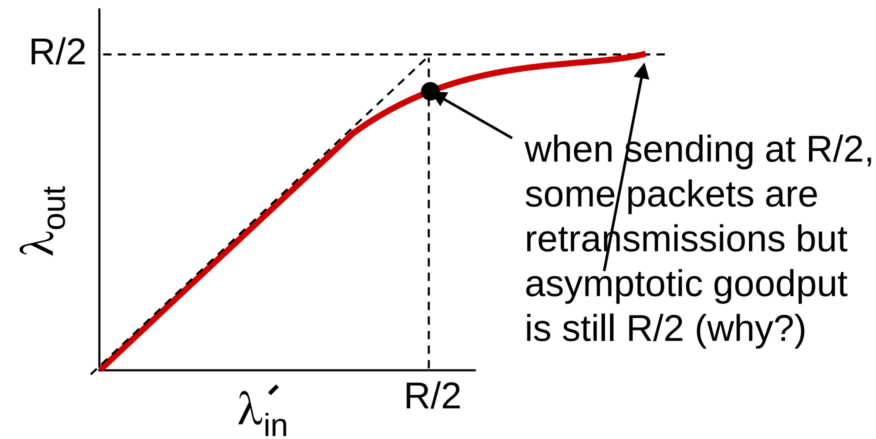
- sender only resends if packet *known* to be lost

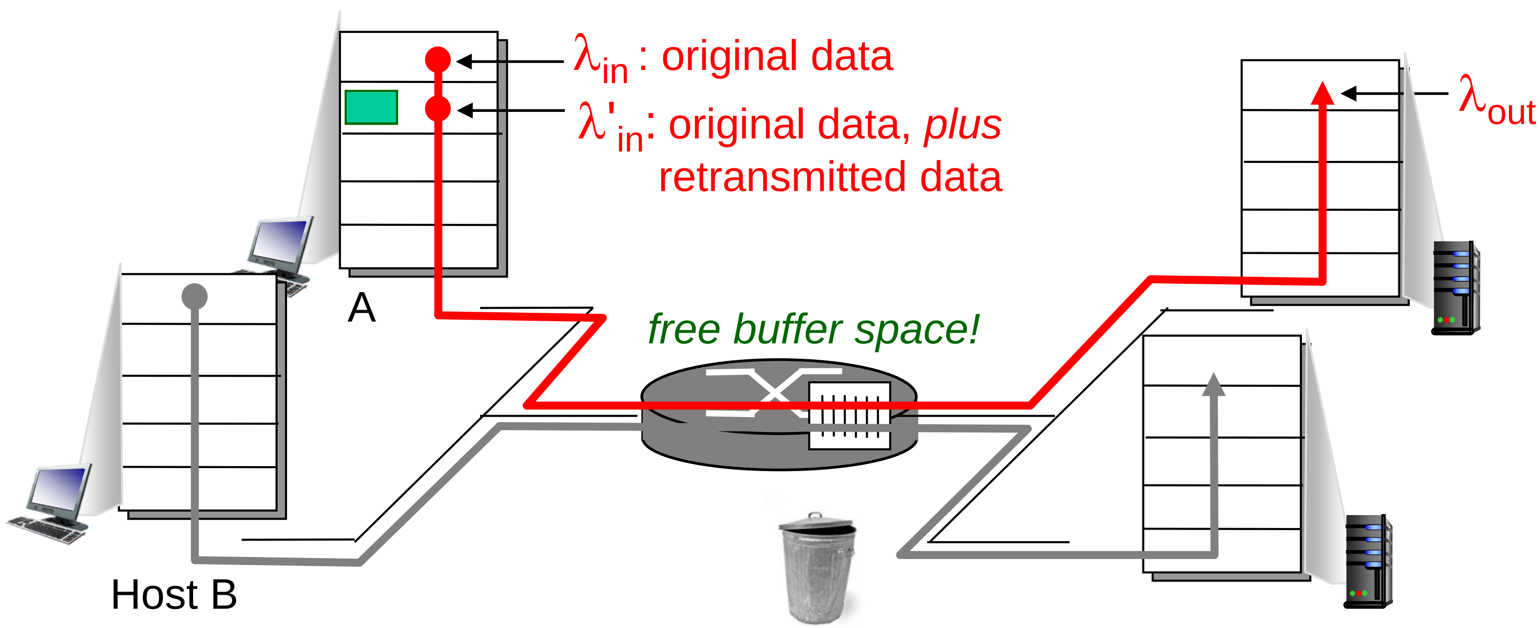


Causes/costs of congestion: scenario 2

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

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

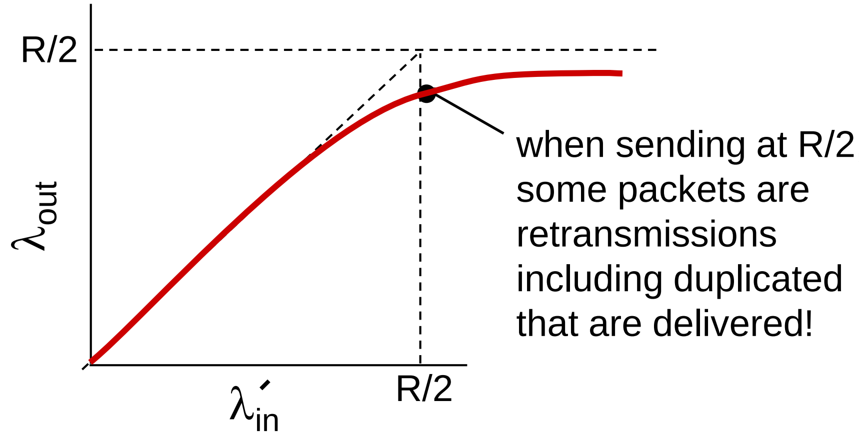


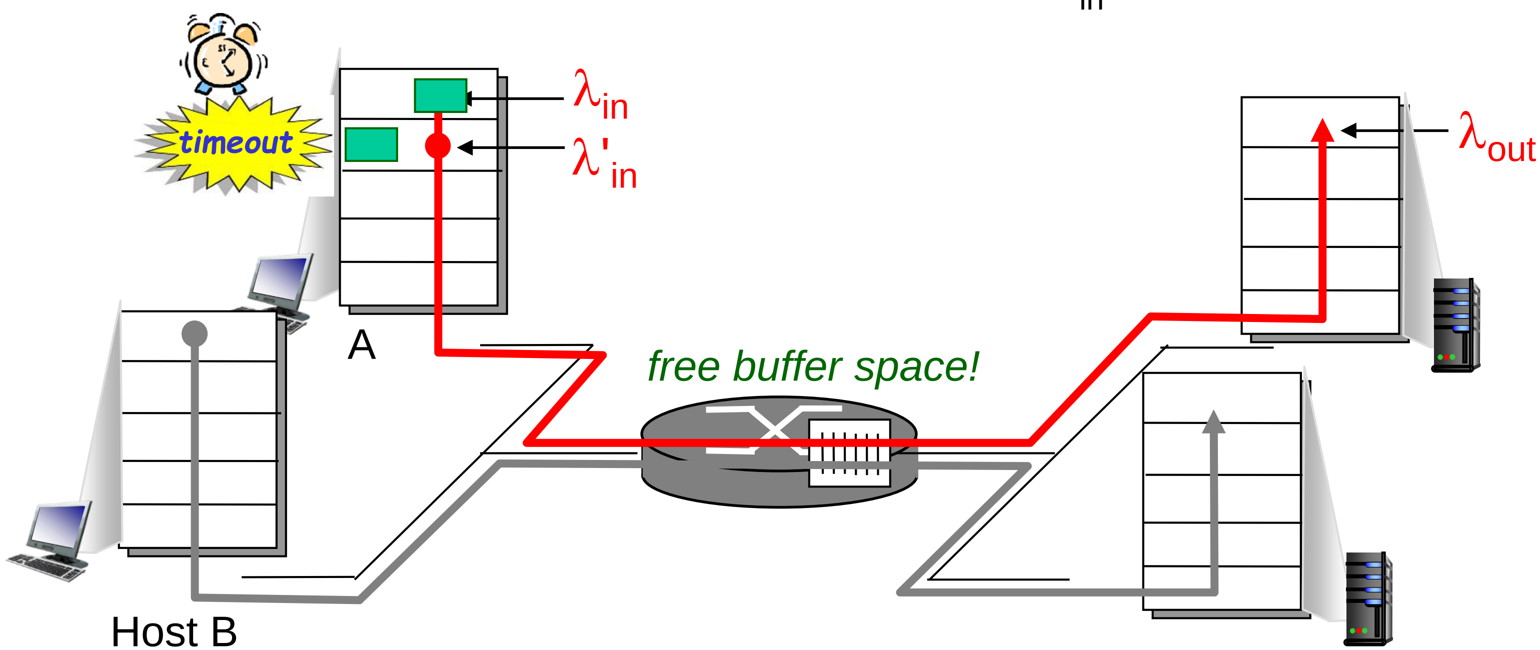


Causes/costs of congestion: scenario 2

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

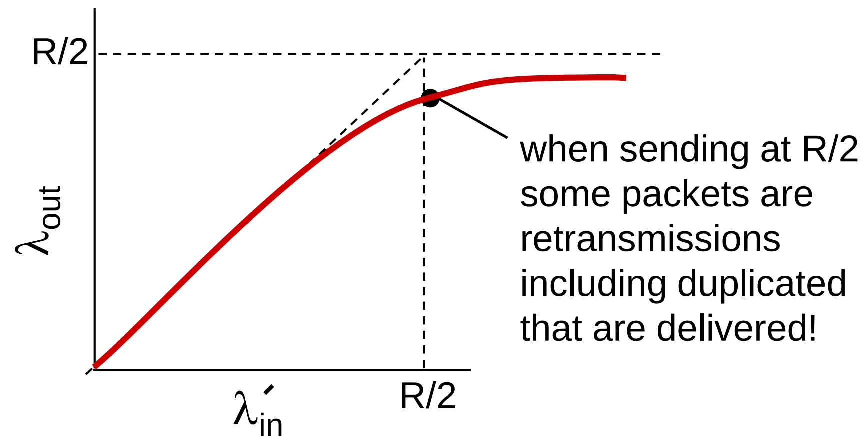




Causes/costs of congestion: scenario 2

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:

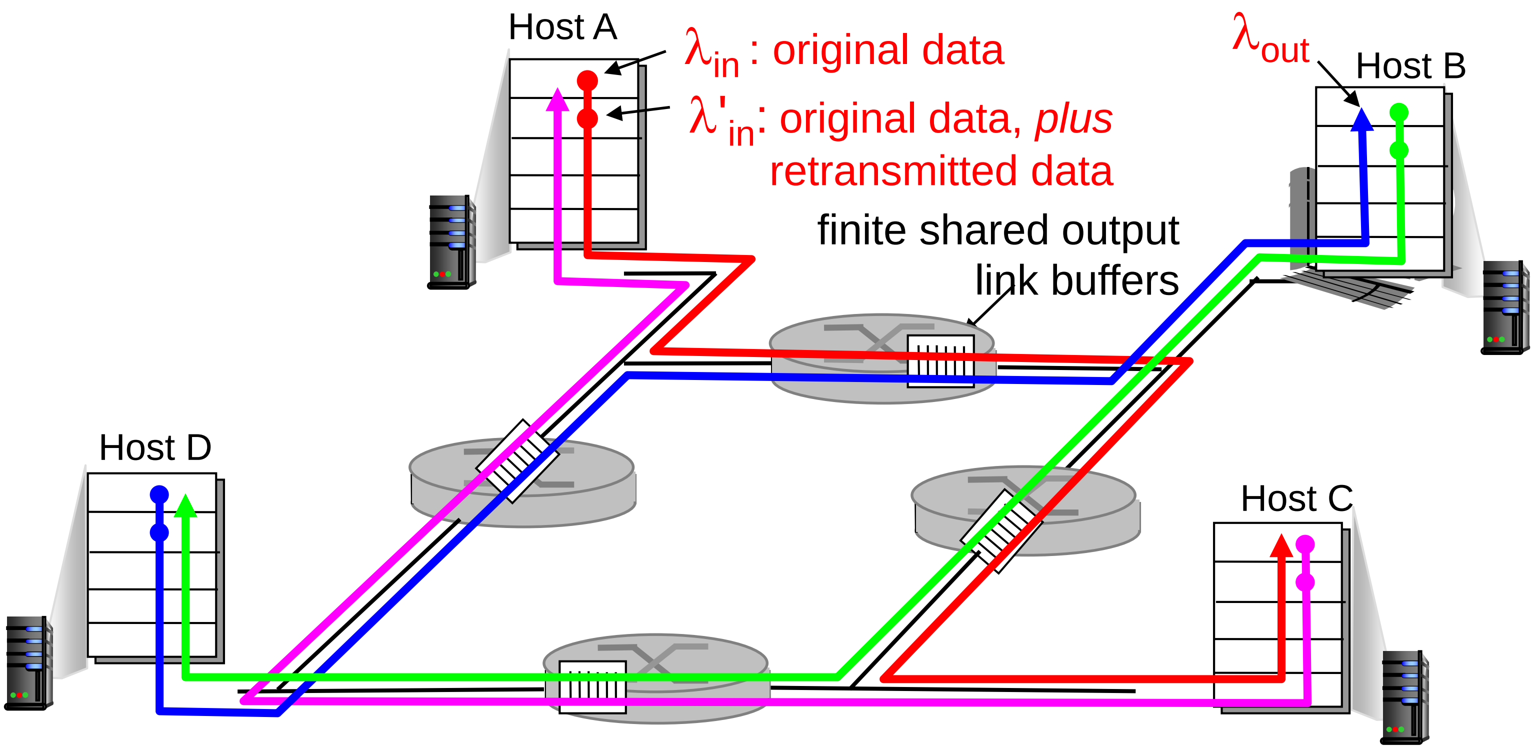
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

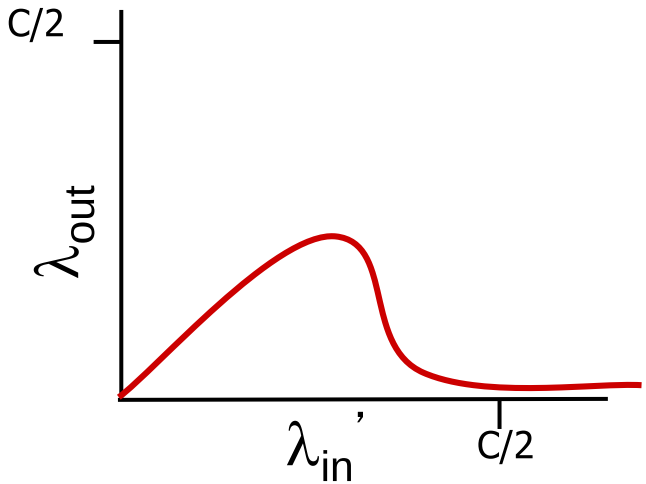
- four senders
- multihop paths
- timeout/retransmit

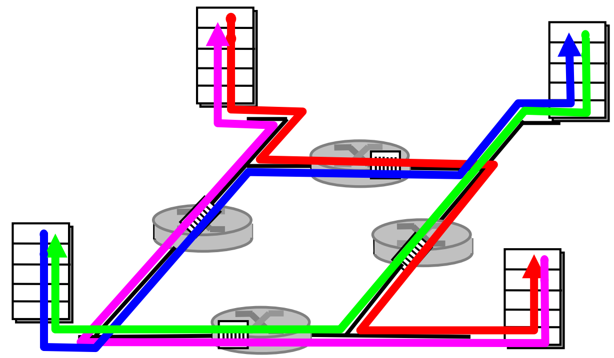
Q: what happens as λ_{in} and λ_{in}' increase ?

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!

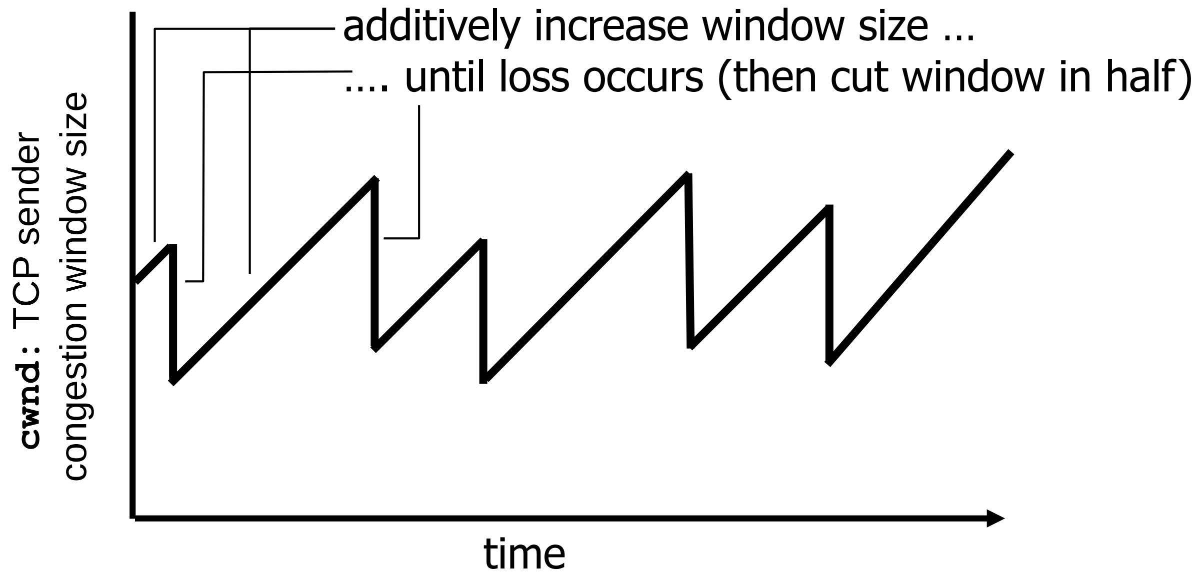
Topics

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

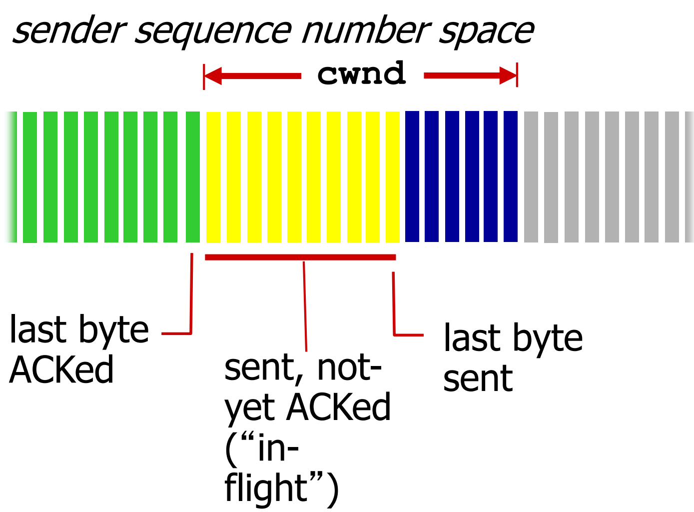
TCP congestion control: additive increase multiplicative decrease

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

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: details



- sender limits transmission:

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

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

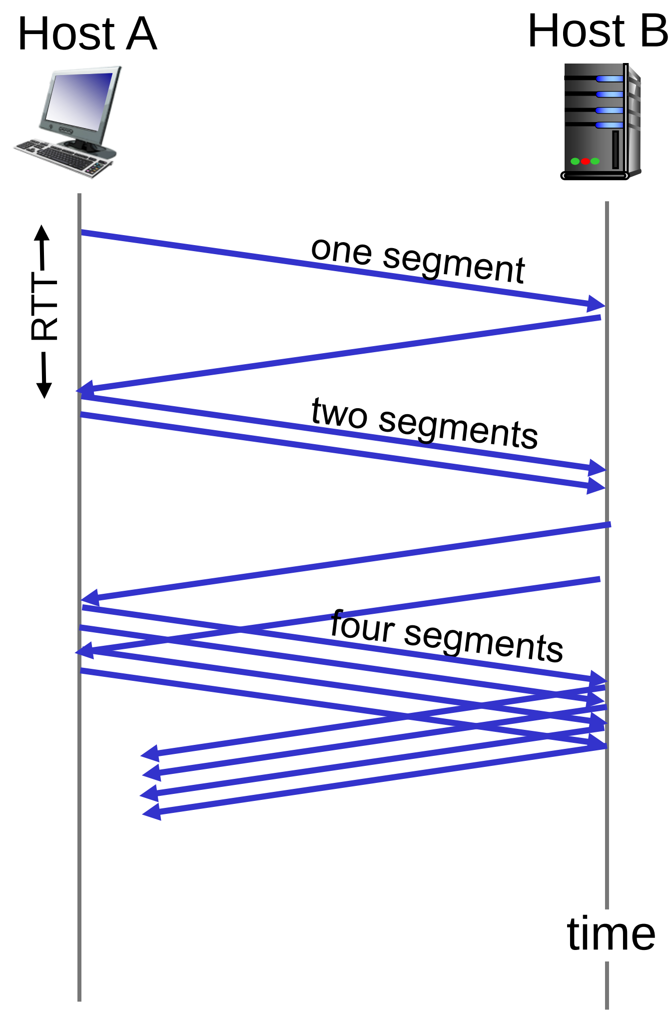
TCP sending rate:

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

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

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`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



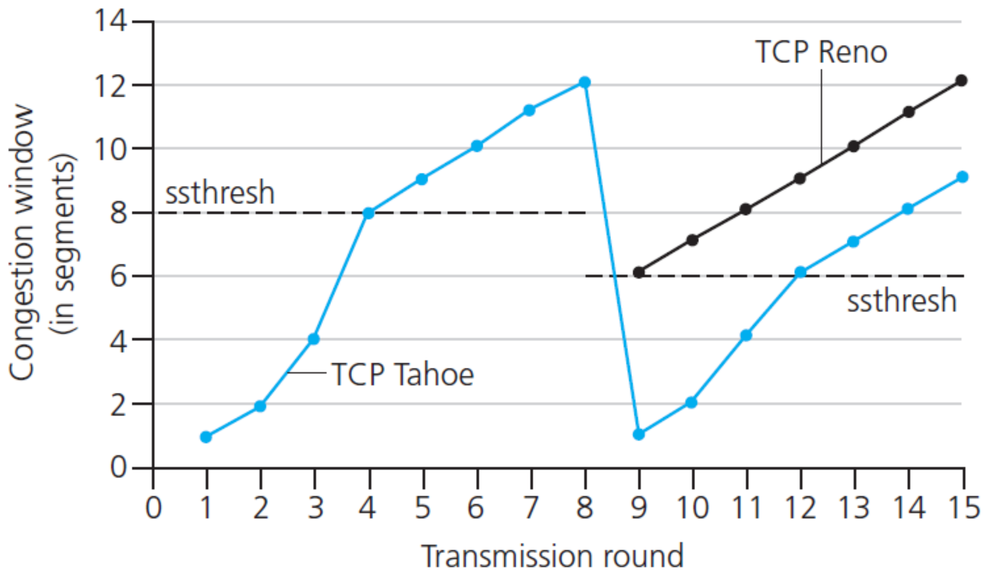
TCP: detecting, reacting to loss

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

TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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

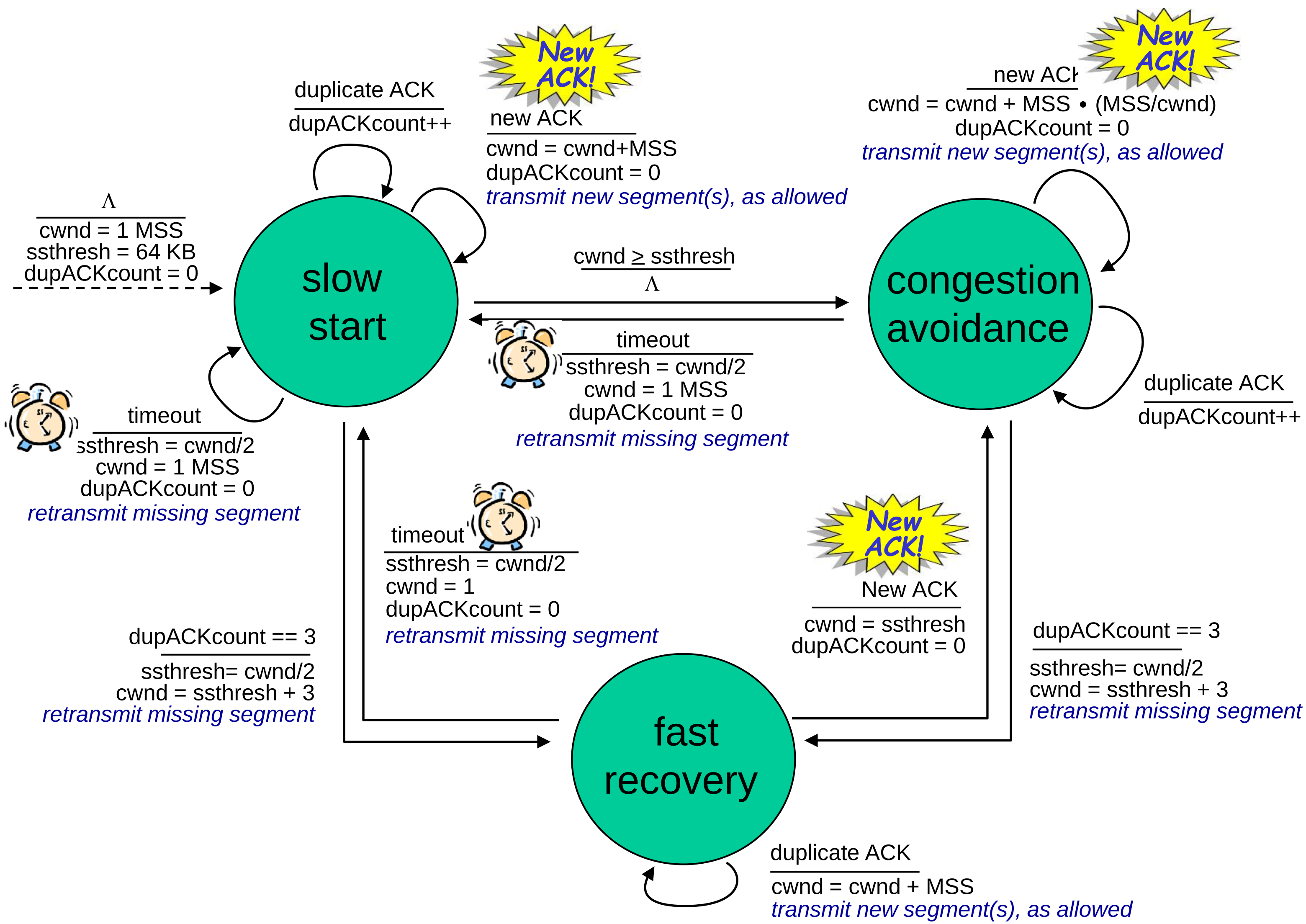


Implementation:

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

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

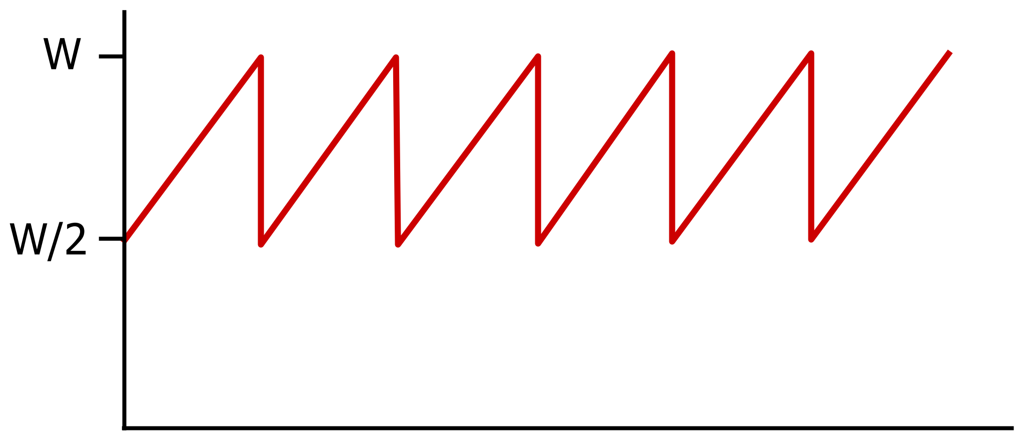
Summary: TCP Congestion Control



TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

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



TCP Futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

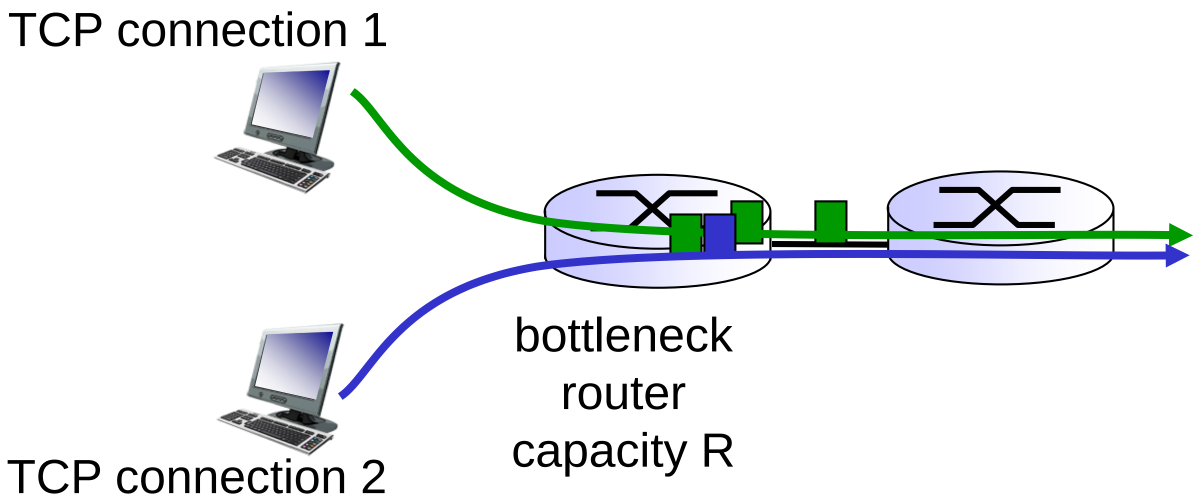
$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

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

- new versions of TCP for high-speed

TCP Fairness

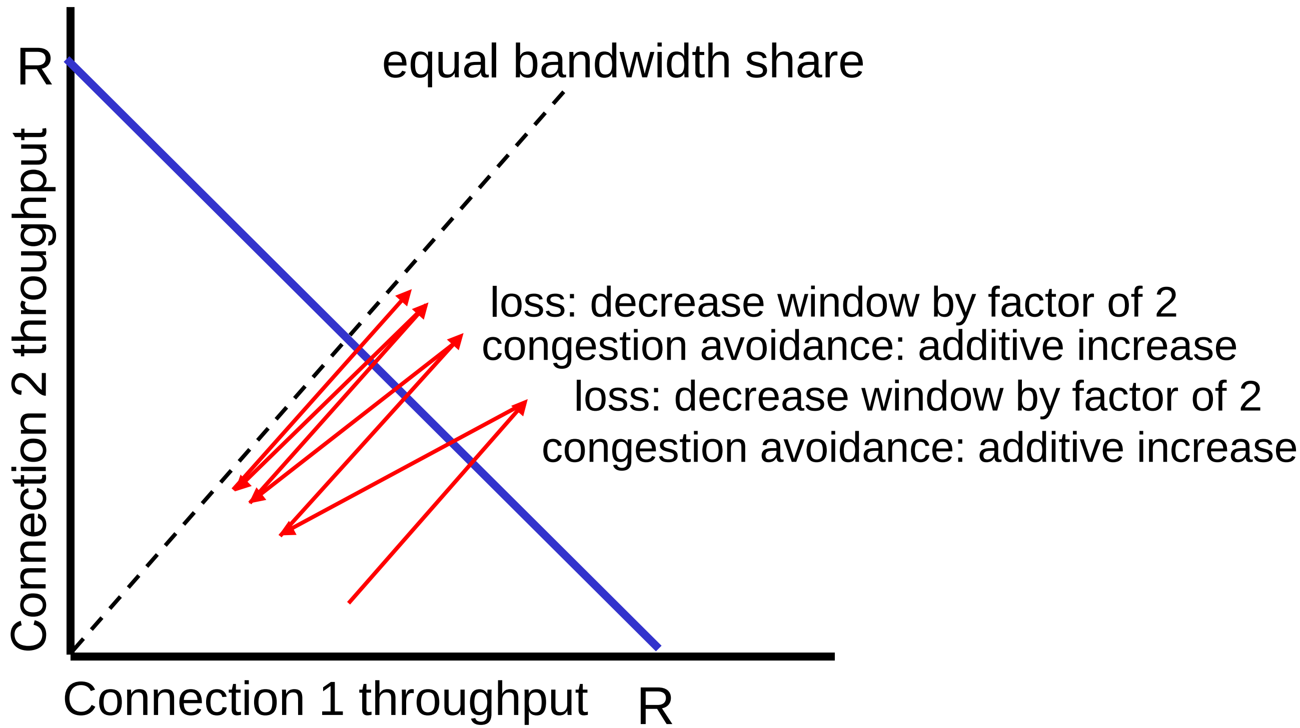
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



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
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

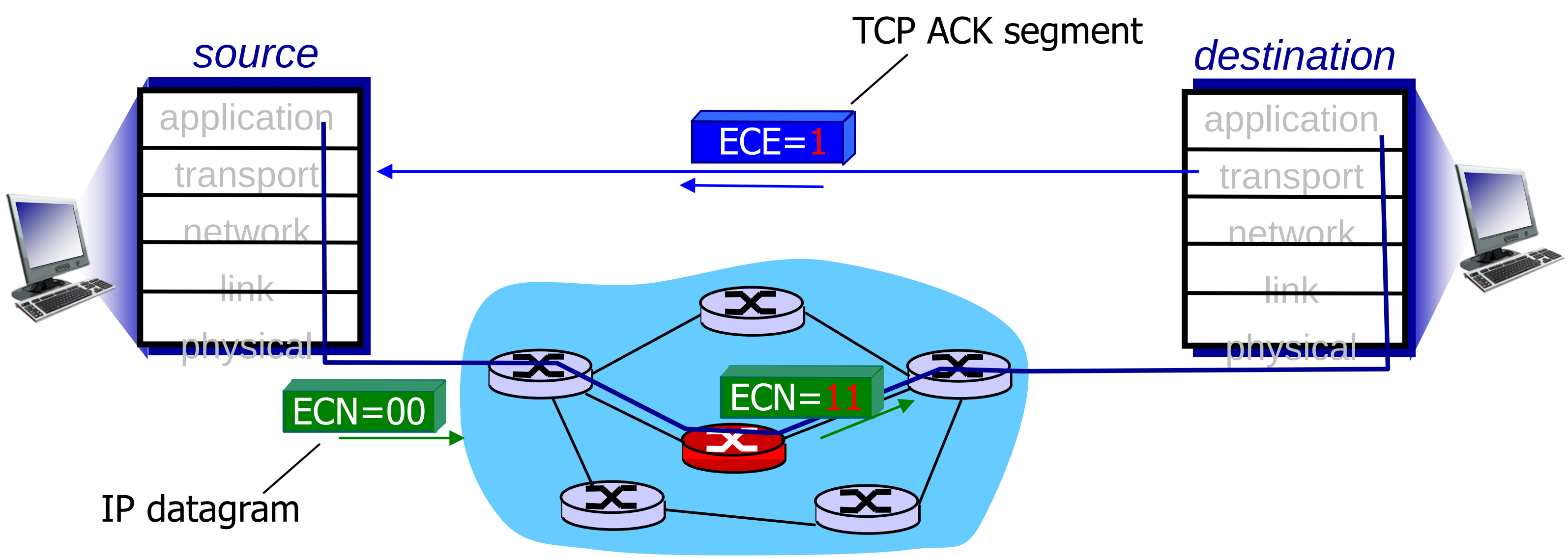
Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Explicit Congestion Notification (ECN)

network-assisted congestion control:

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



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