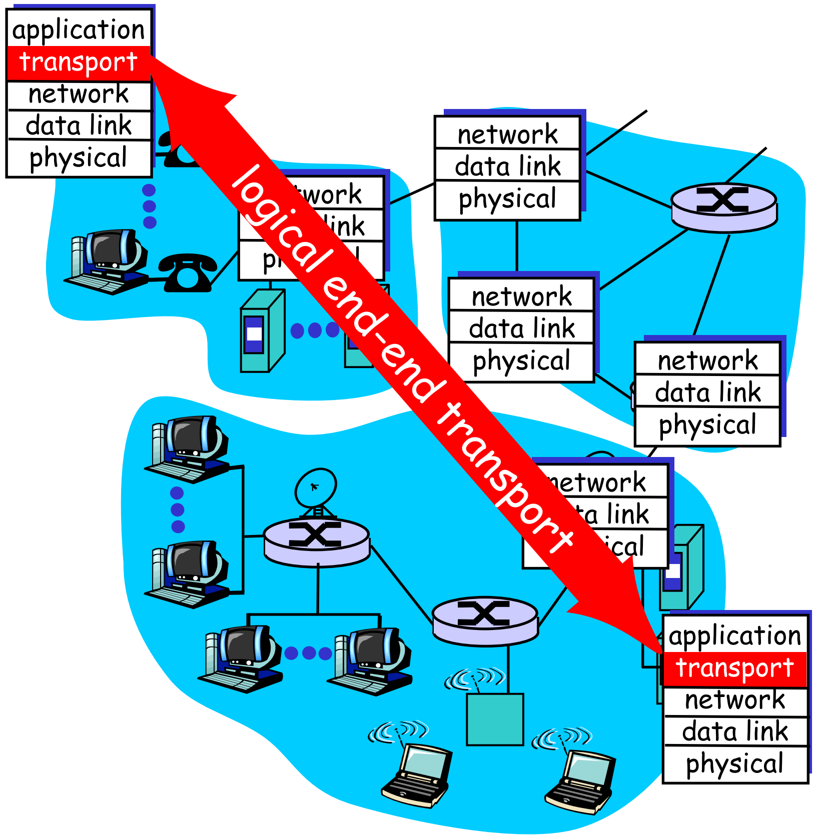
Chapter 3: Transport Layer

Our goals:

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

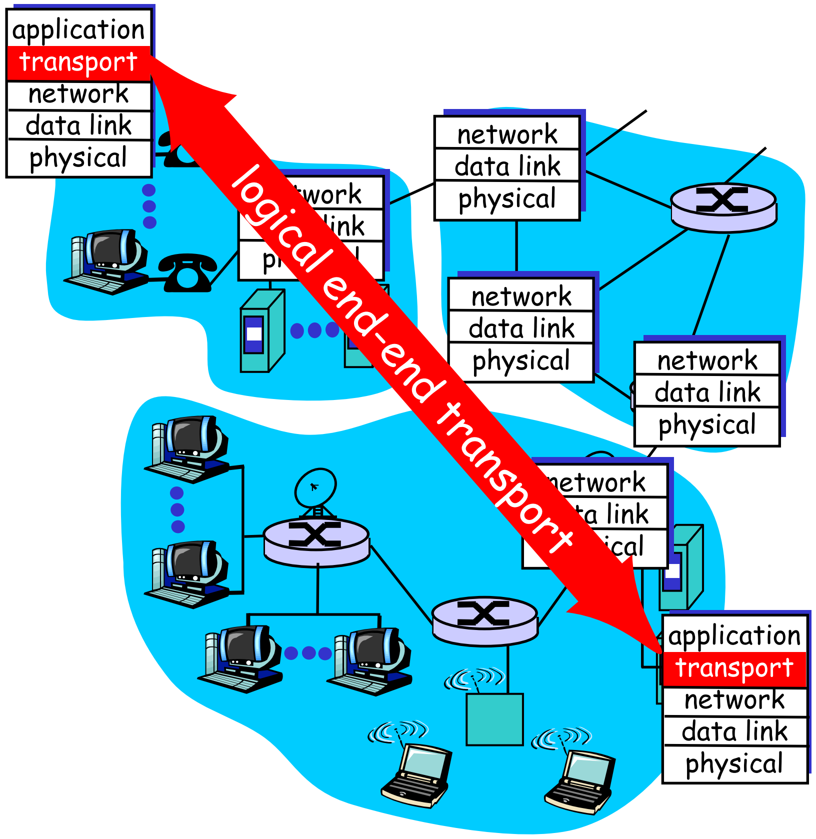
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ayşe and Bülent
- 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



Multiplexing/demultiplexing

Demultiplexing at rcv host:

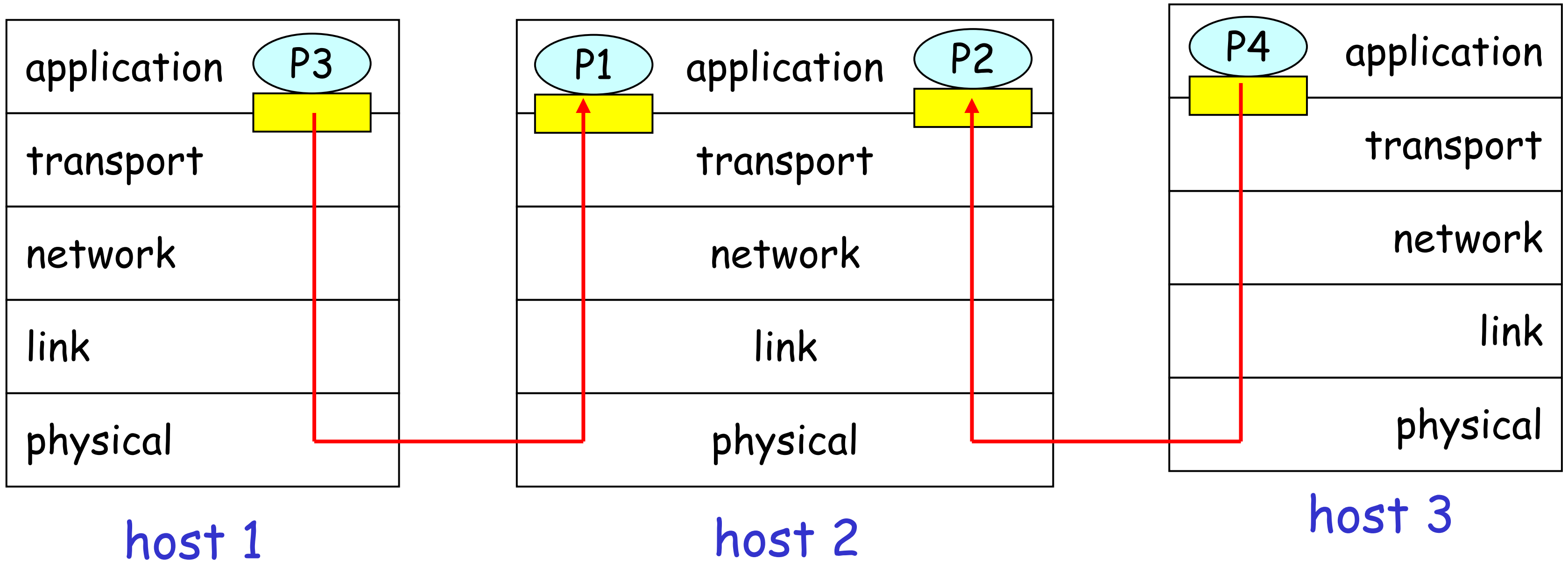
delivering received segments to correct socket

Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

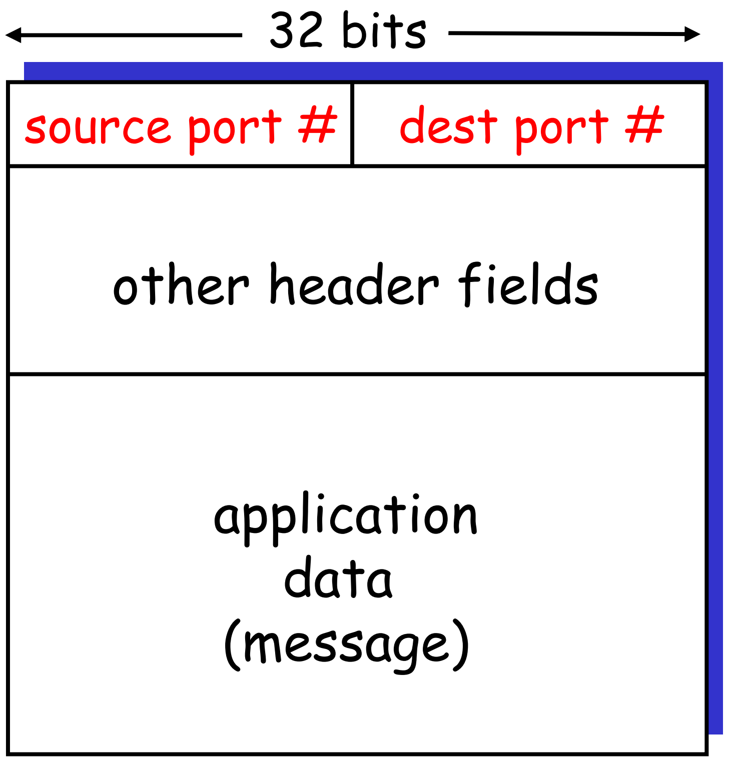
🟨 = socket

⬭ = process



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(9111);
DatagramSocket mySocket2 = new
   DatagramSocket(9222);
```

- UDP socket identified by two-tuple:

(dest IP address, dest port number)

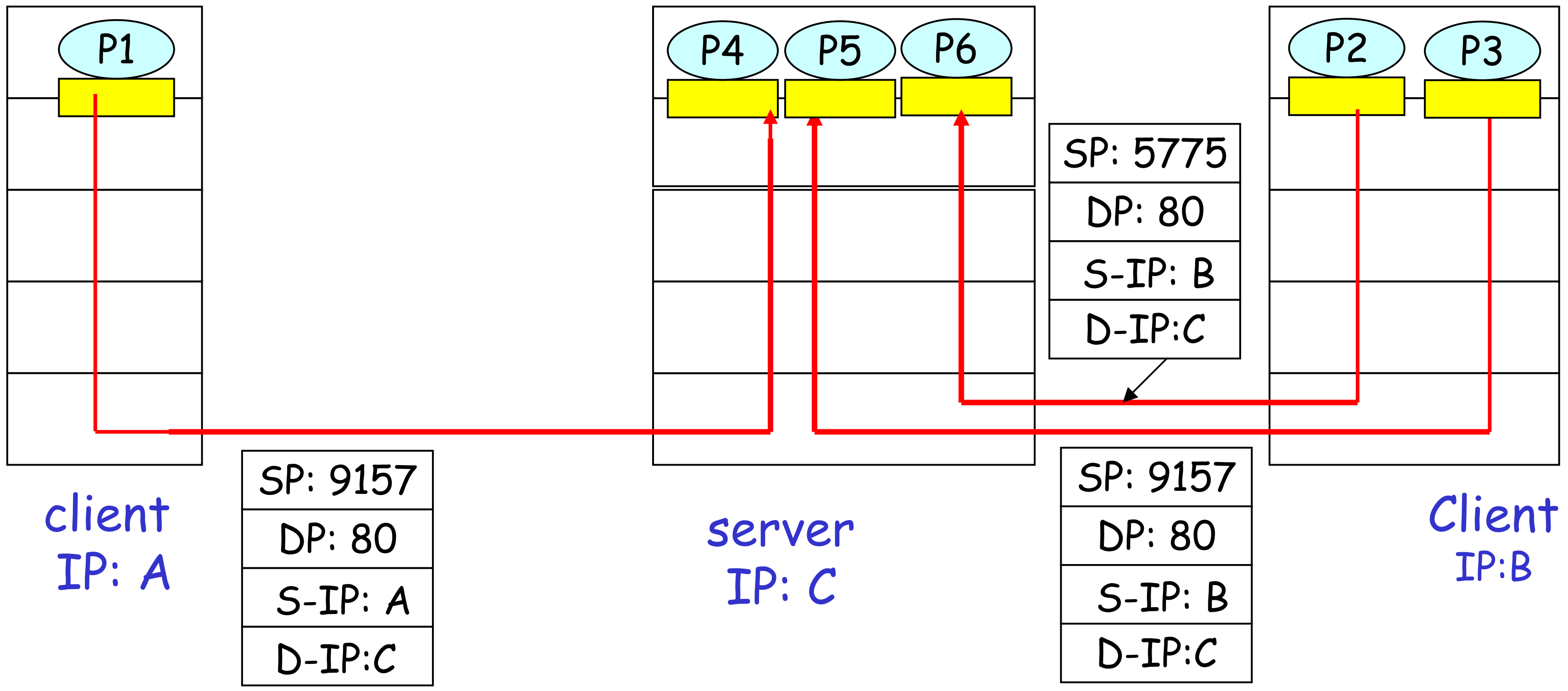
- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connection-oriented demux

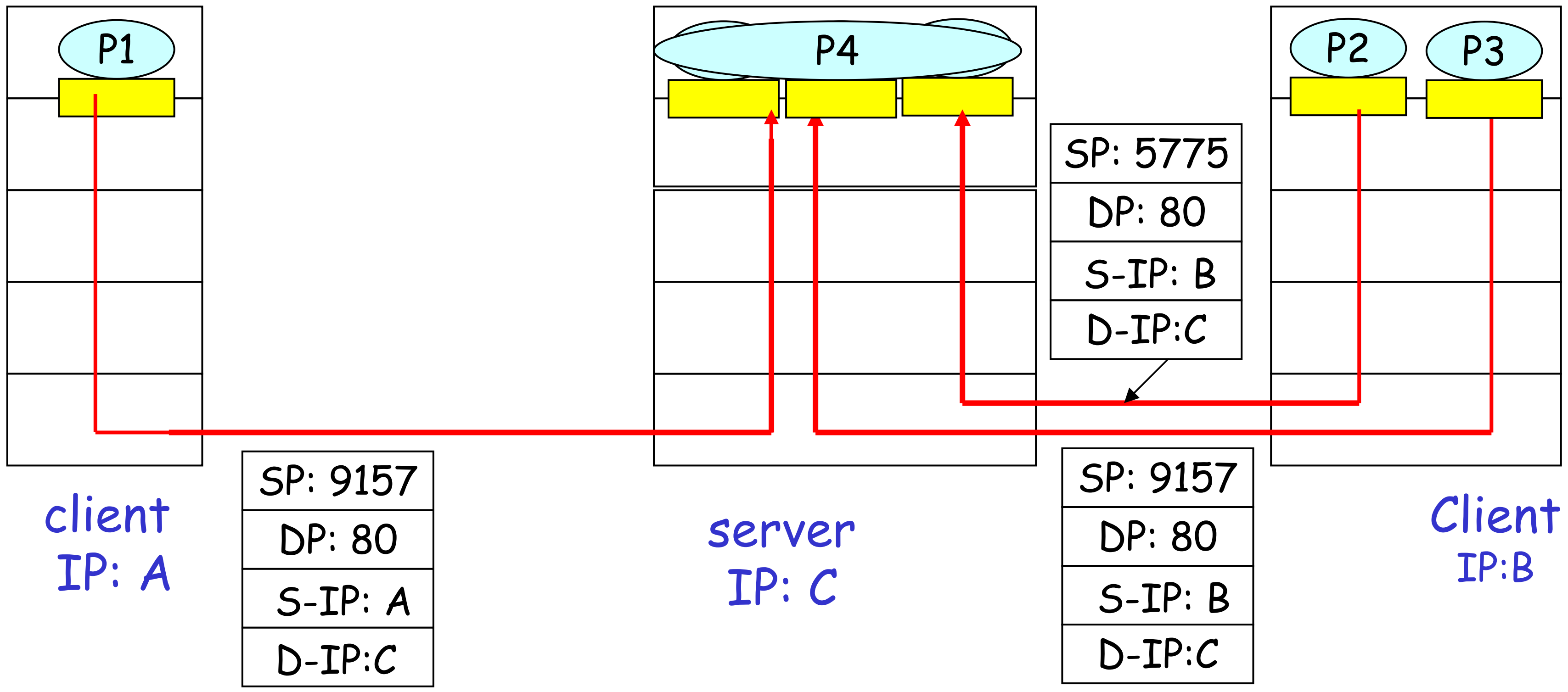
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket

- 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 (cont)



Connection-oriented demux Threaded Web Server



UDP: User Datagram Protocol [RFC 768]

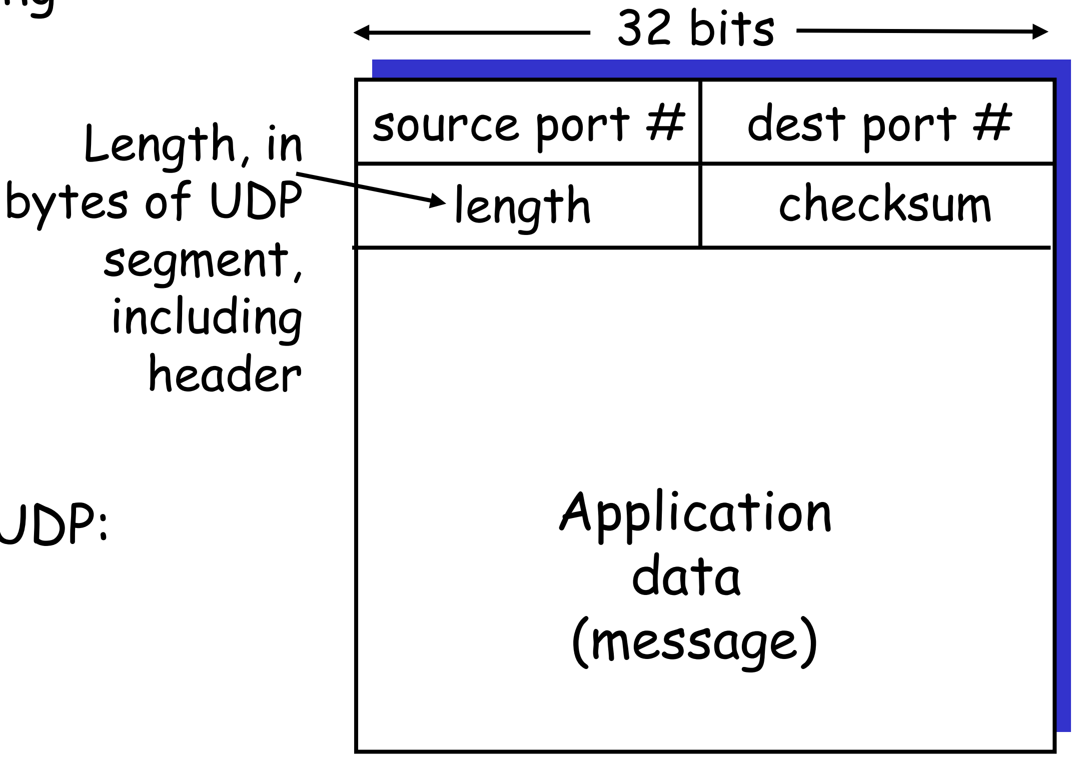
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!


32 bits
Length, in bytes of UDP segment, including header
source port #
dest port #
length
checksum
Application data (message)

UDP segment format

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

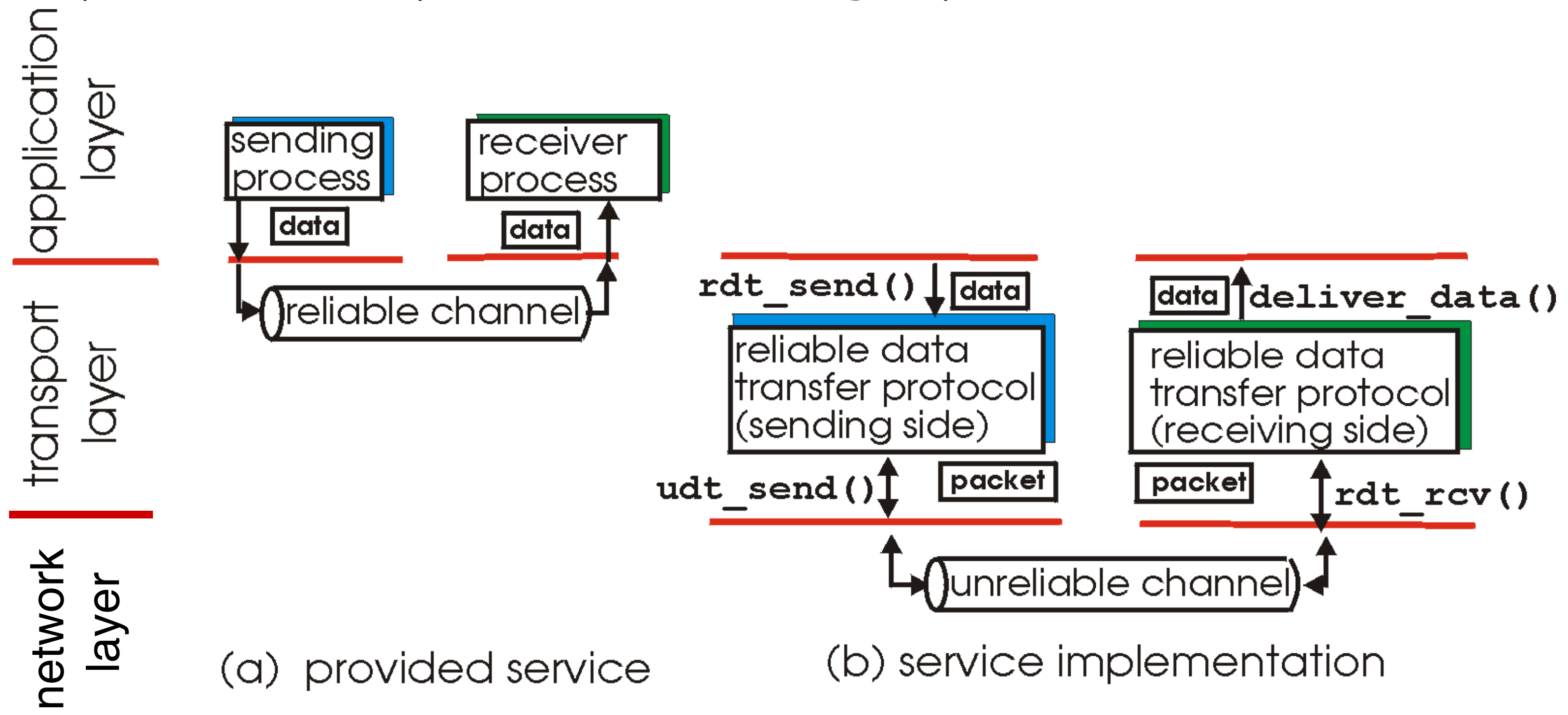
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents with wraparound of carry out bit
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

Principles of Reliable data transfer

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

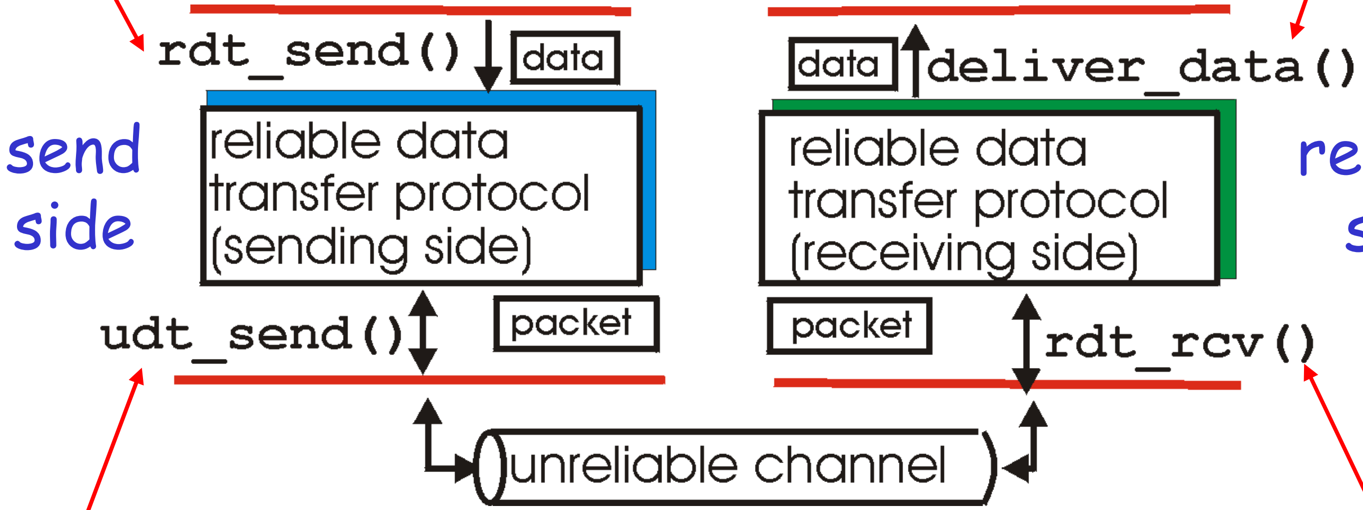


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



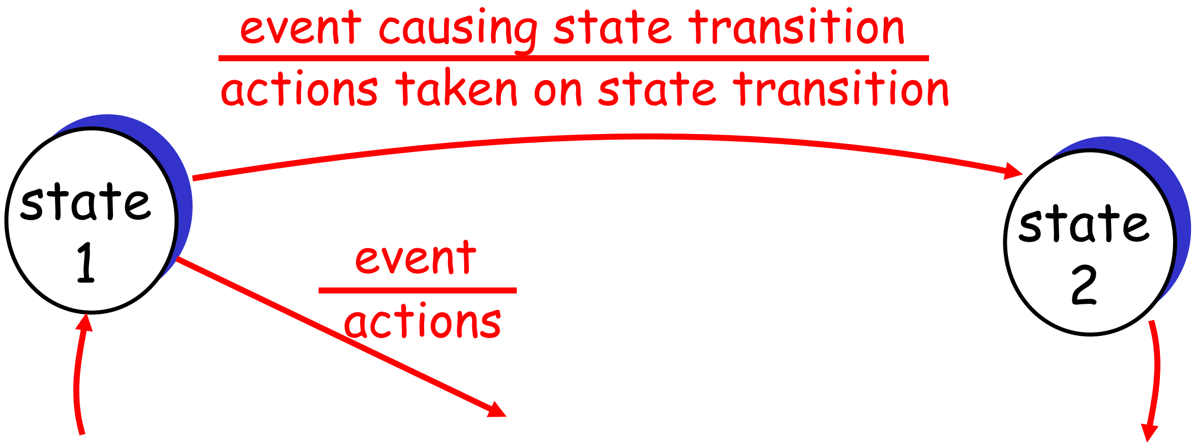
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

We'll:

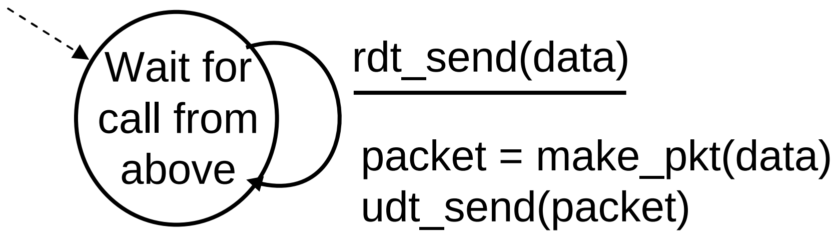
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver


event causing state transition
actions taken on state transition
state 1
event
actions
state 2

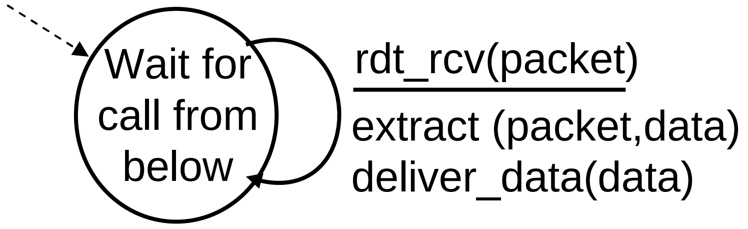
state: when in this "state" next state uniquely determined by next event

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 read data from underlying channel



sender

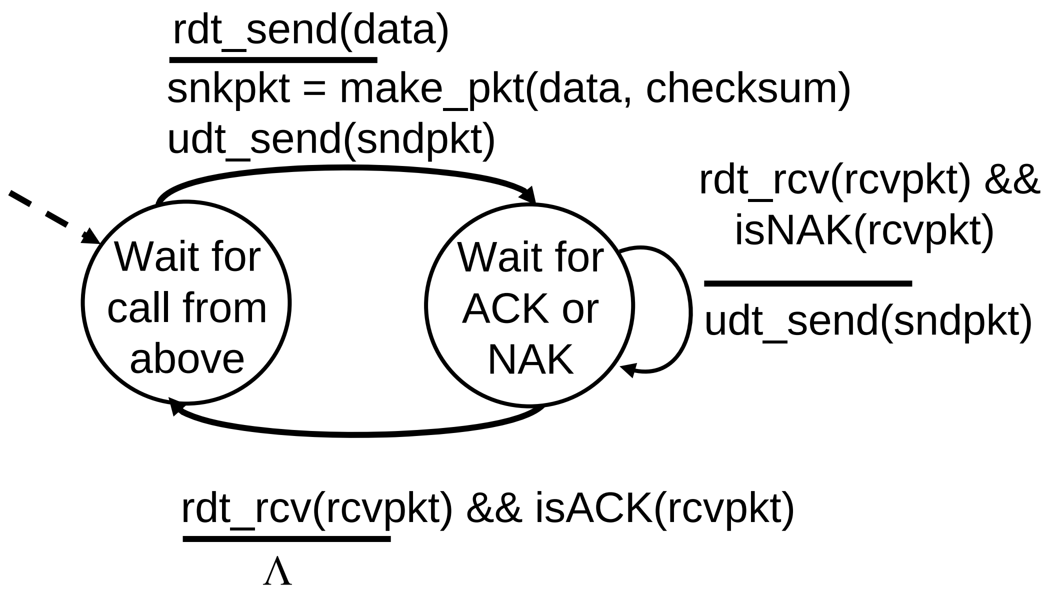


receiver

Rdt2.0: channel with bit errors

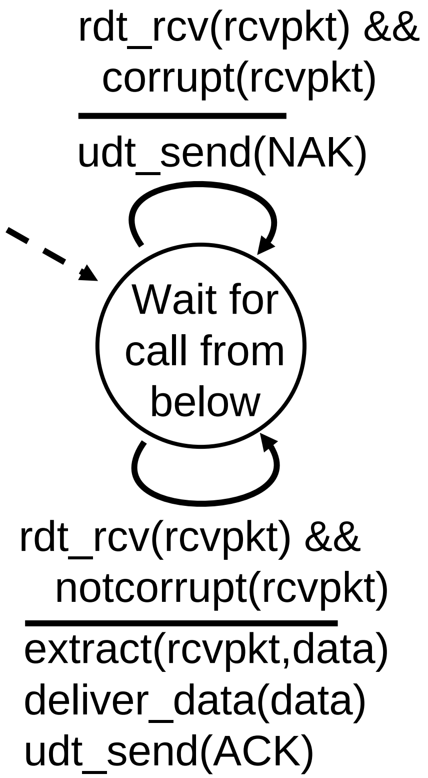
- underlying channel may flip bits in packet
  - recall: checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had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

rdt2.0: FSM specification

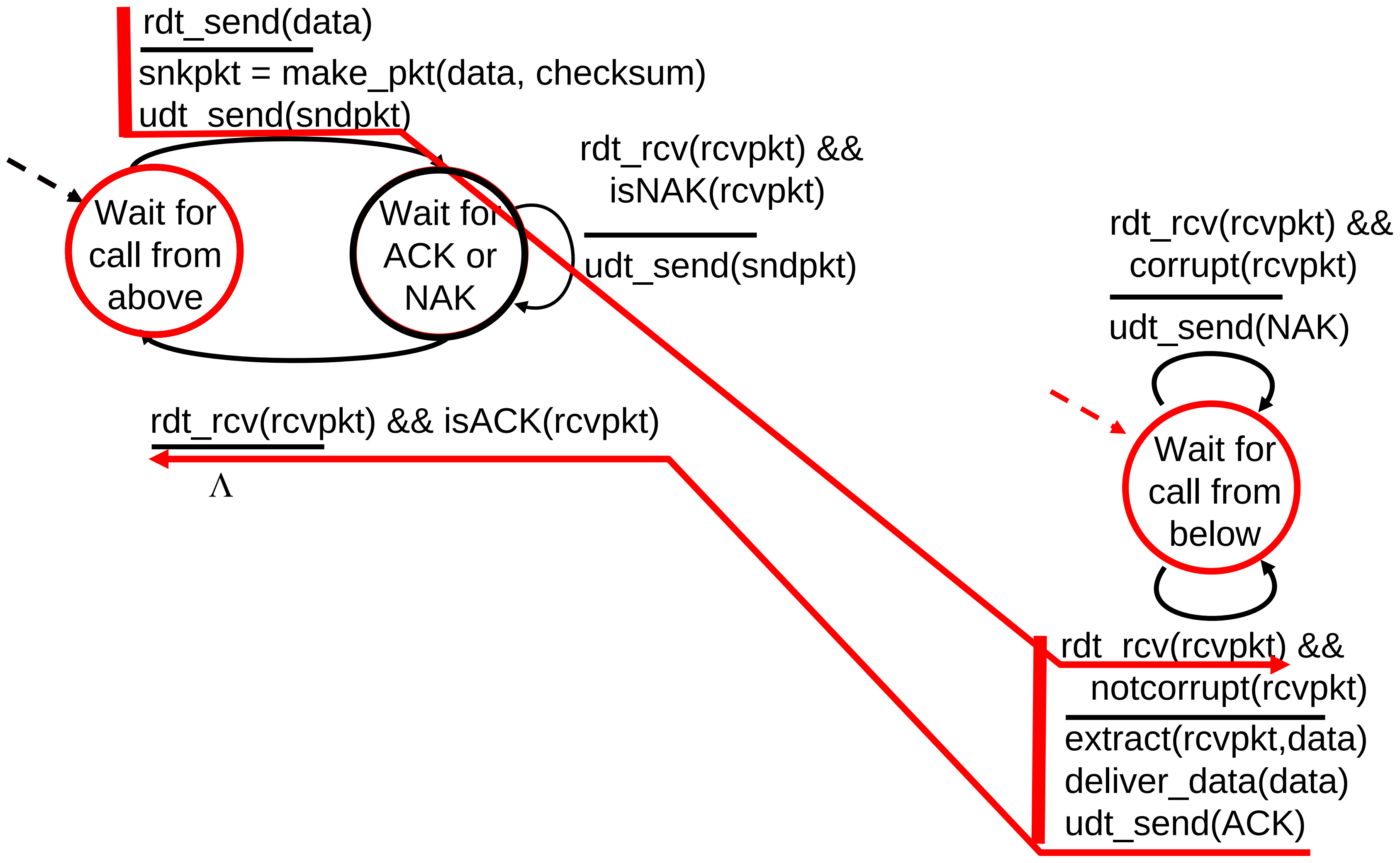

rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ

sender

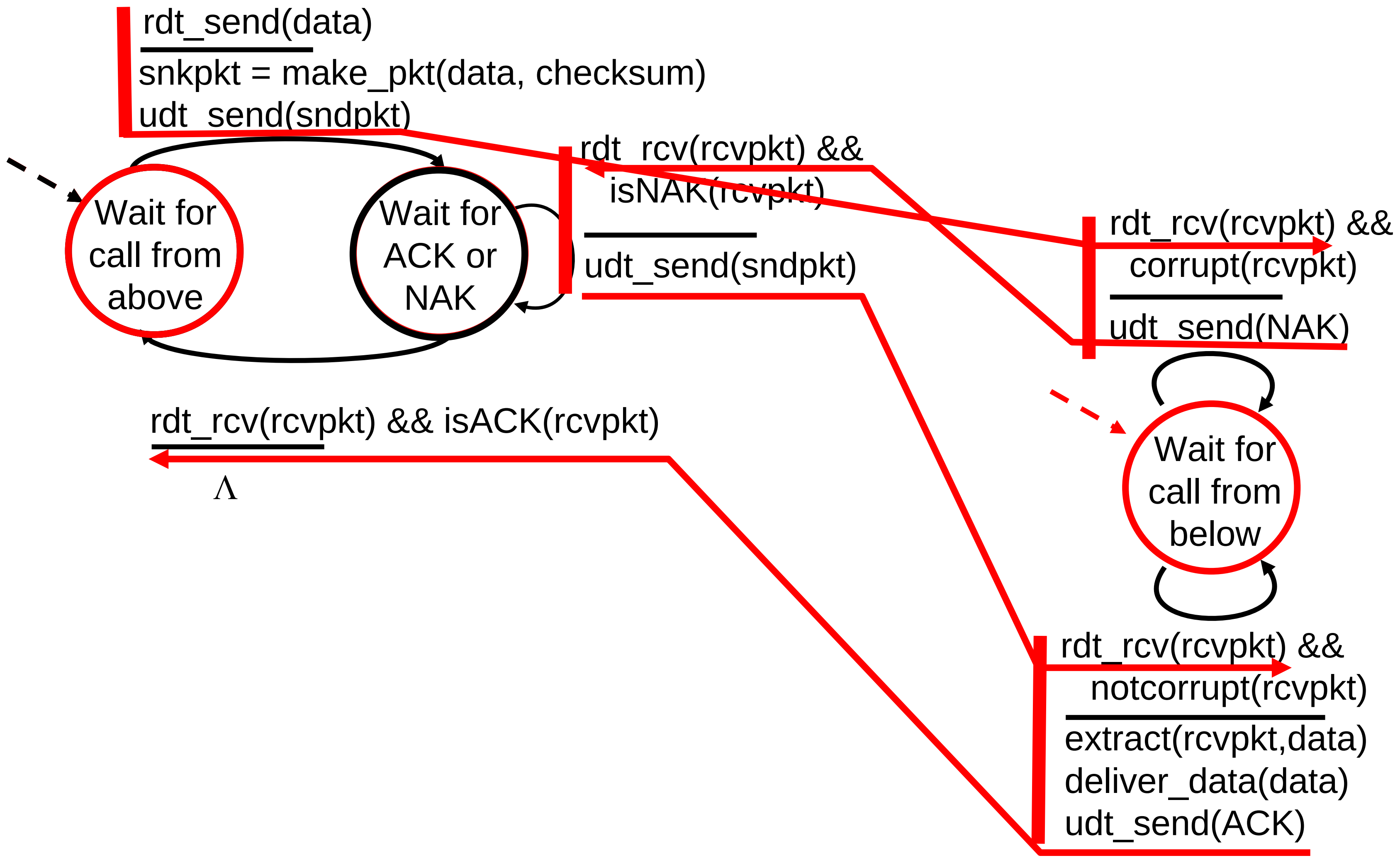
receiver



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

What to do?

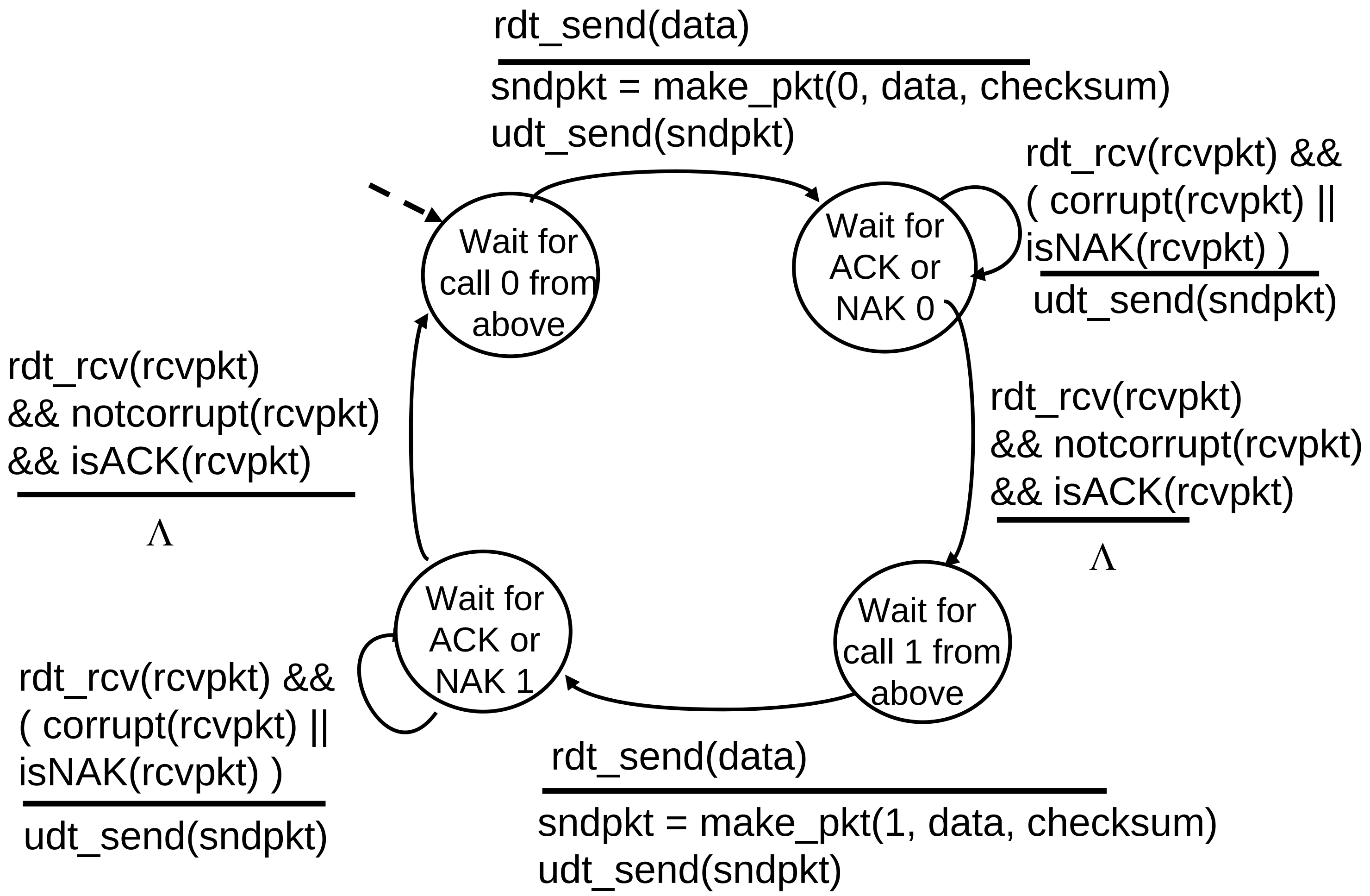
- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

Handling duplicates:

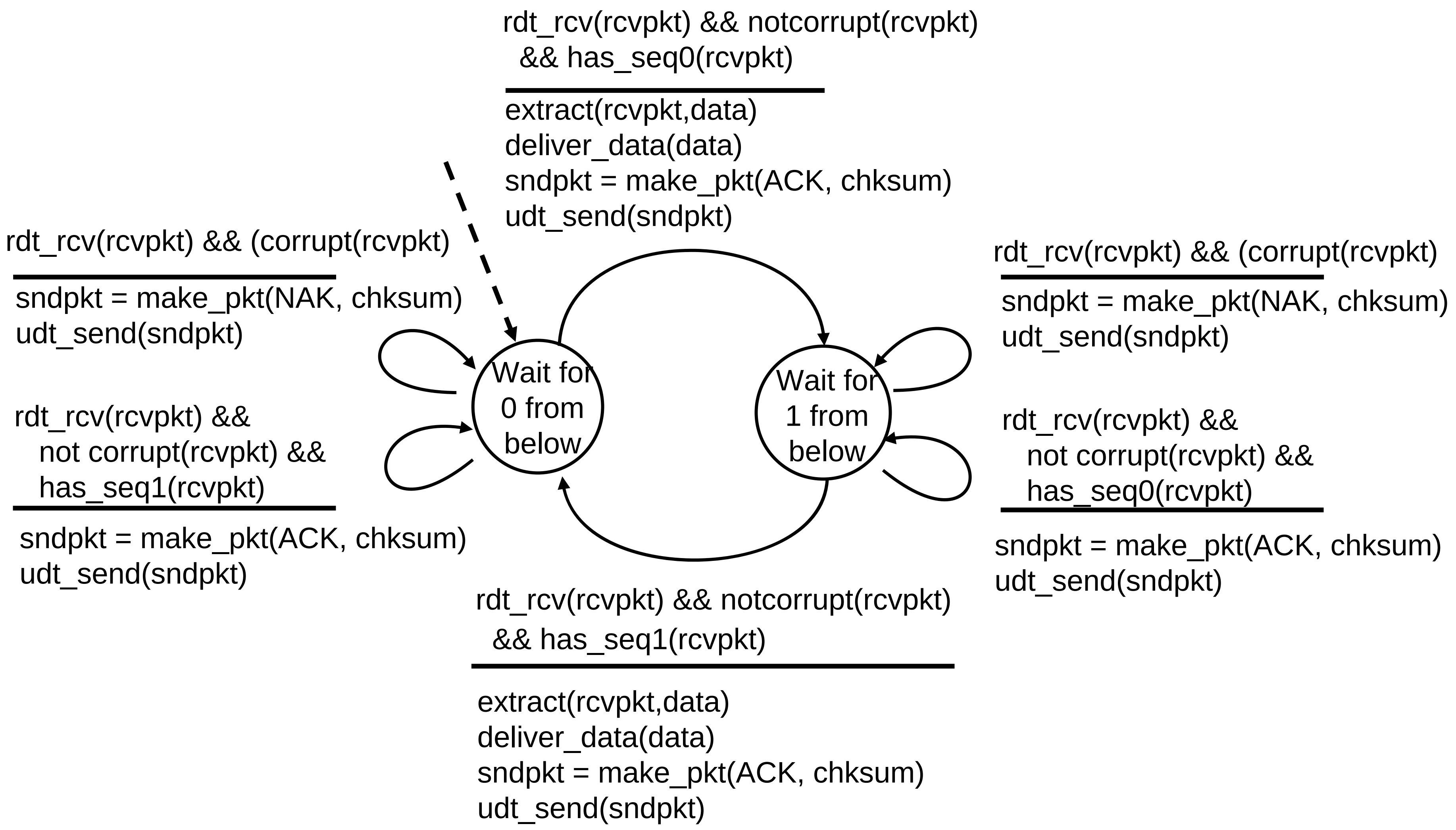
- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

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

rdt2.1: sender, handles garbled ACK/NAKs


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for call 1 from above
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
Wait for ACK or NAK 1
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ

rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender:

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

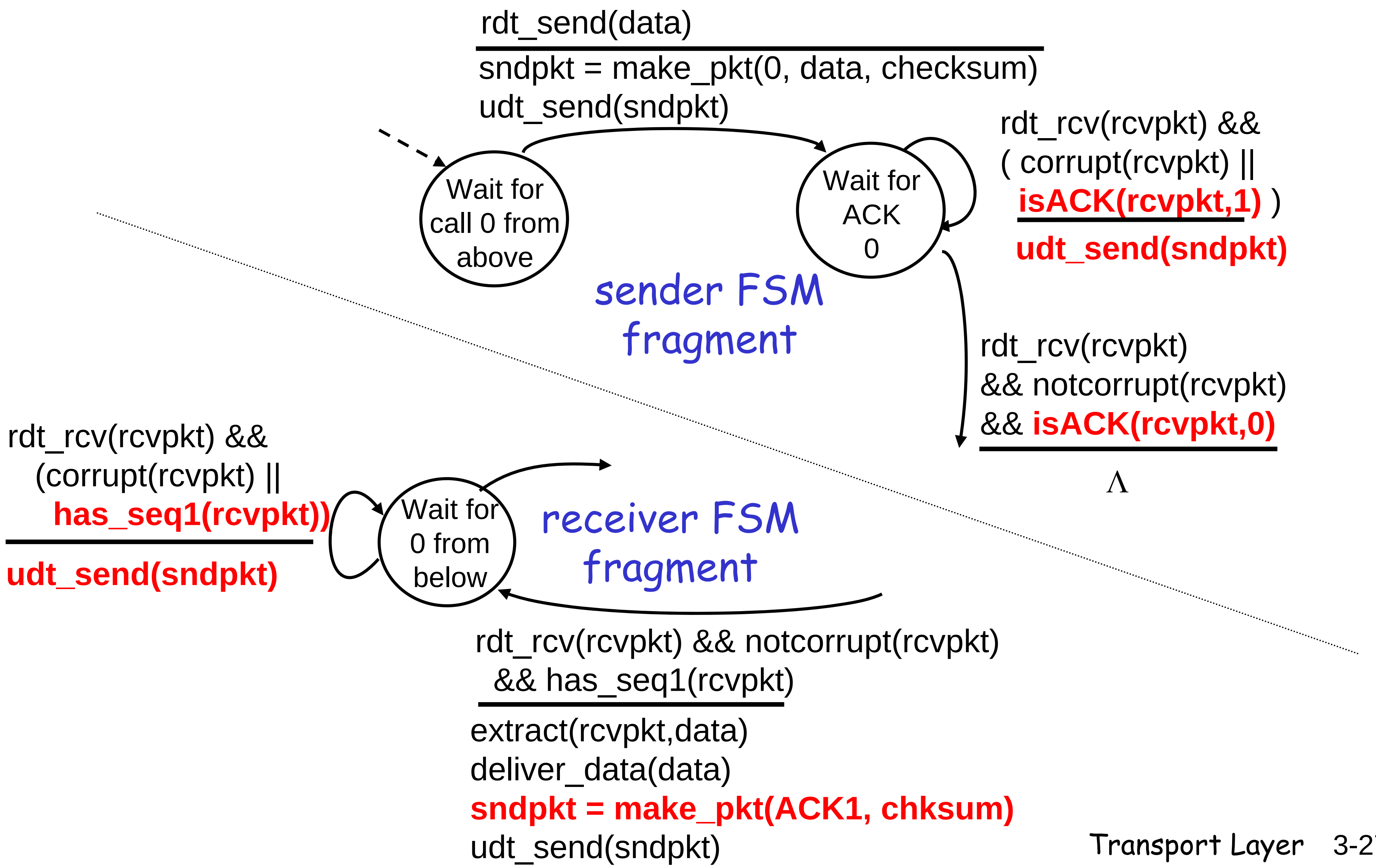
Receiver:

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

rdt2.2: a NAK-free protocol

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

rdt2.2: sender, receiver fragments


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
udt_send(sndpkt)
sender FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
udt_send(sndpkt)
Wait for 0 from below
receiver FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

rdt3.0: channels with errors *and* loss

New assumption: underlying channel can also lose packets (data or ACKs)

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

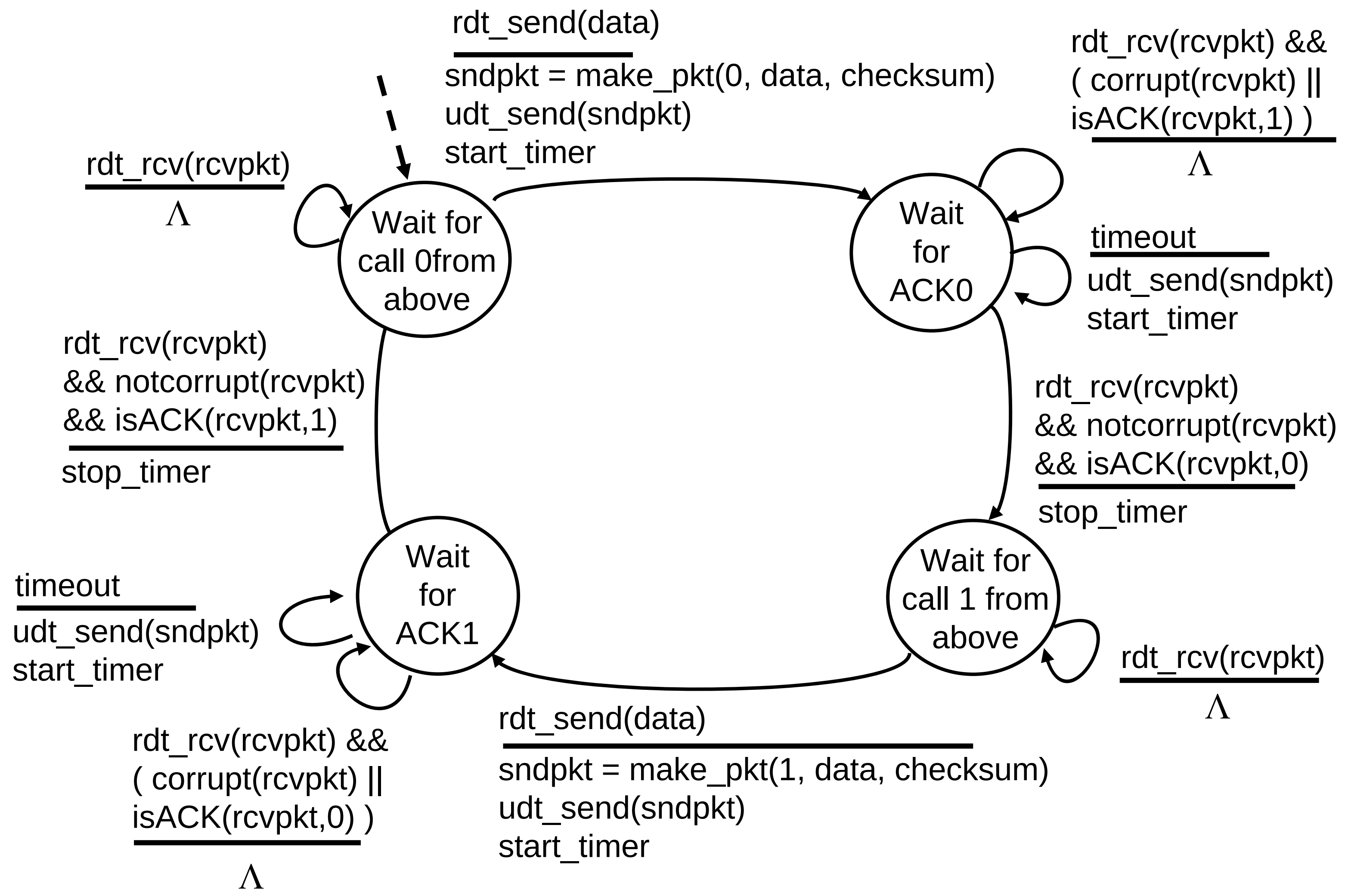
Q: how to deal with loss?

- sender waits until certain data or ACK lost, then retransmits
- drawbacks?

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

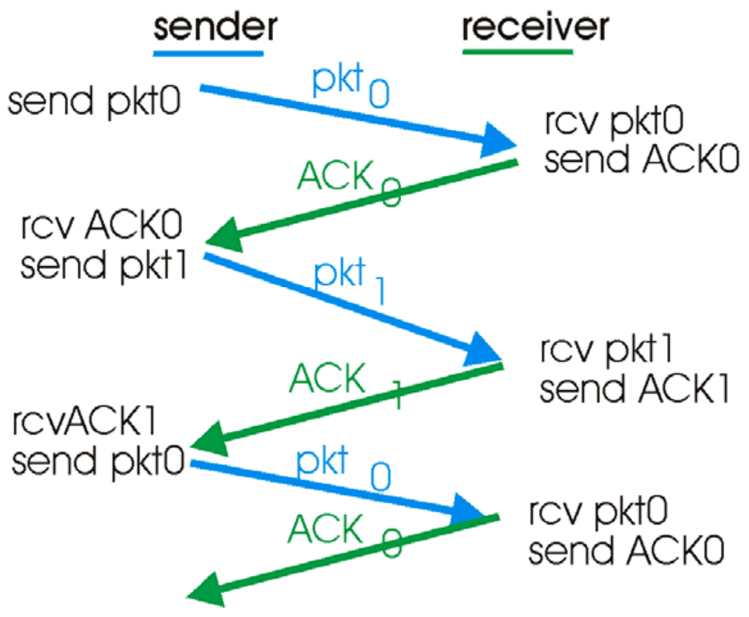
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 sender

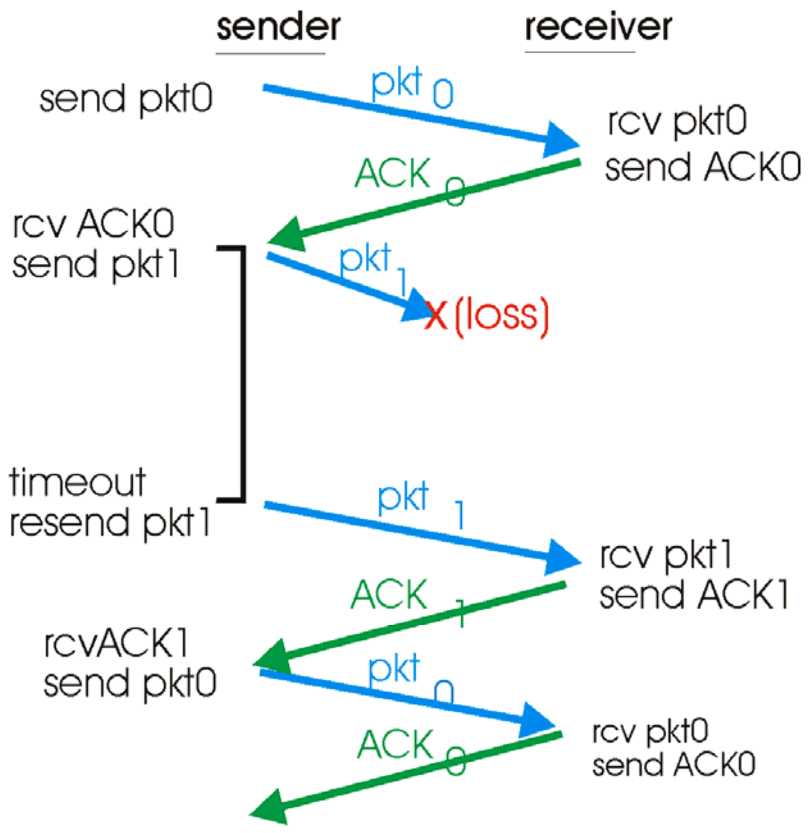


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for ACK1
Wait for call 1 from above
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 in action

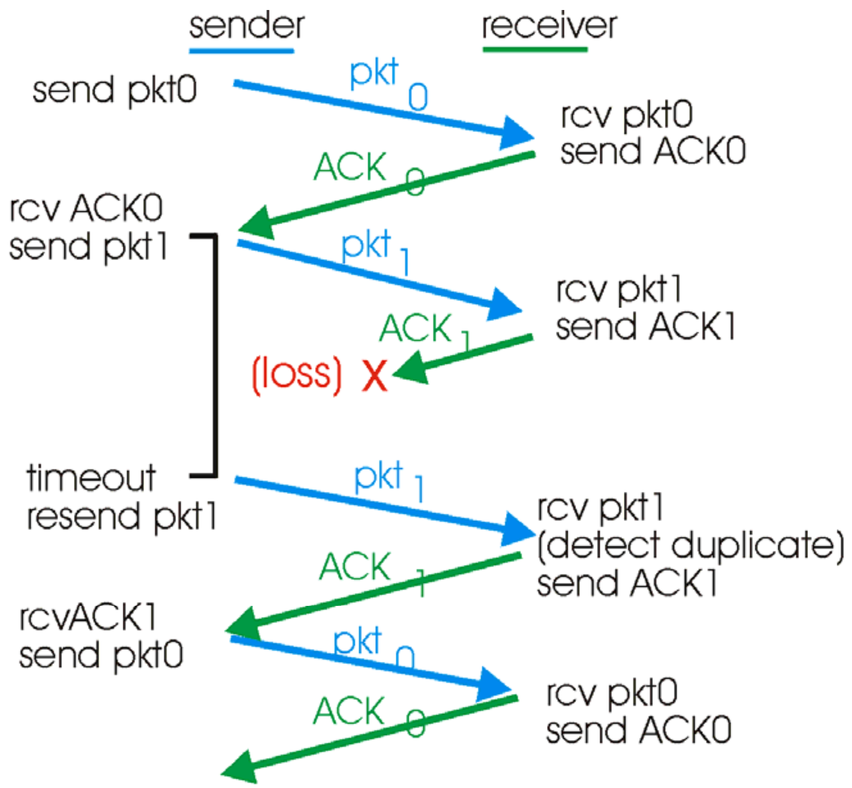


(a) operation with no loss

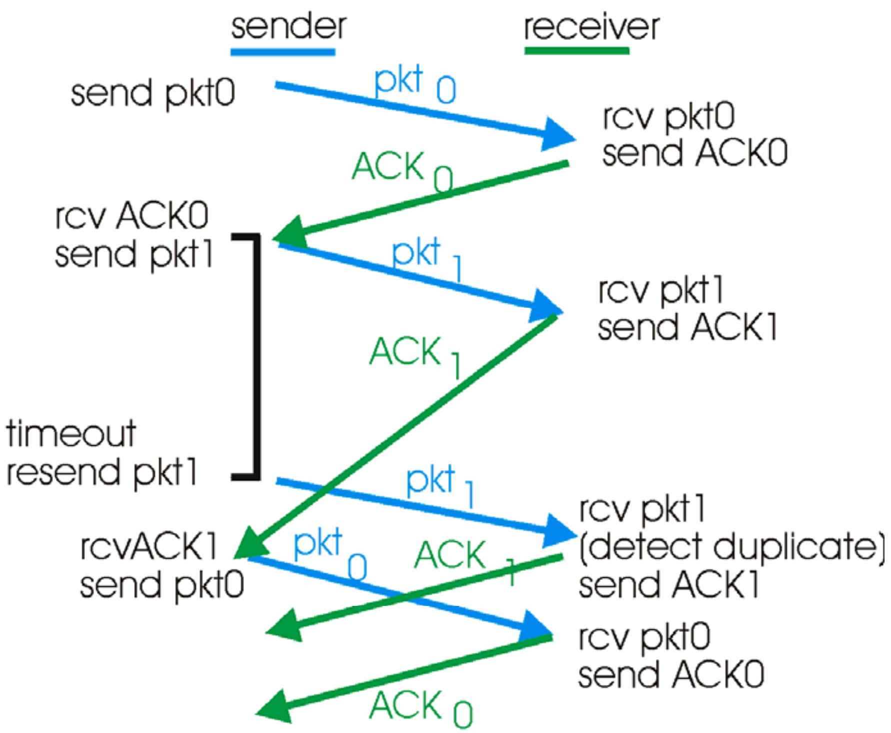


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

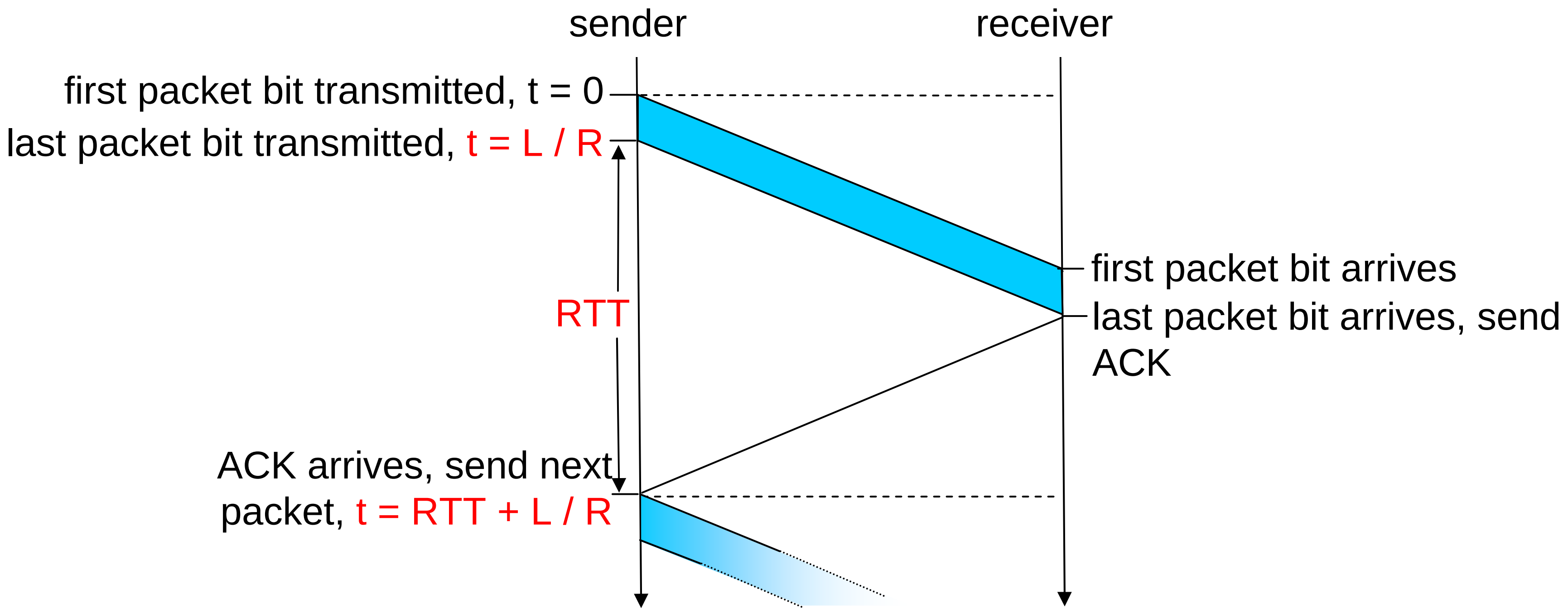
- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

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

  - U_{sender}: utilization – fraction of time sender busy sending
  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

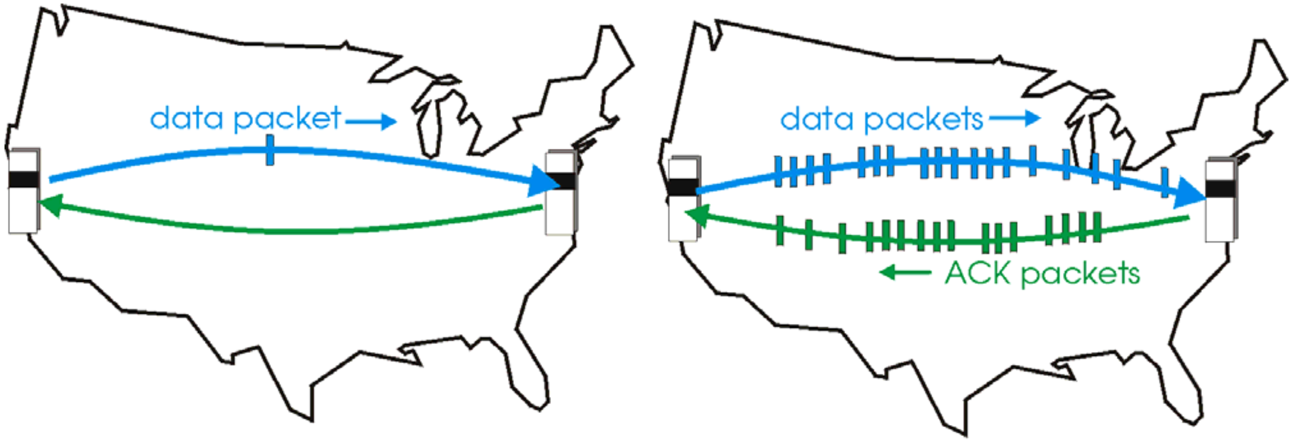

sender
receiver
first packet bit transmitted, t = 0
last packet bit transmitted, t = L / R
RTT
first packet bit arrives
last packet bit arrives, send ACK
ACK arrives, send next packet, t = RTT + L / R

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

Pipelined protocols

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

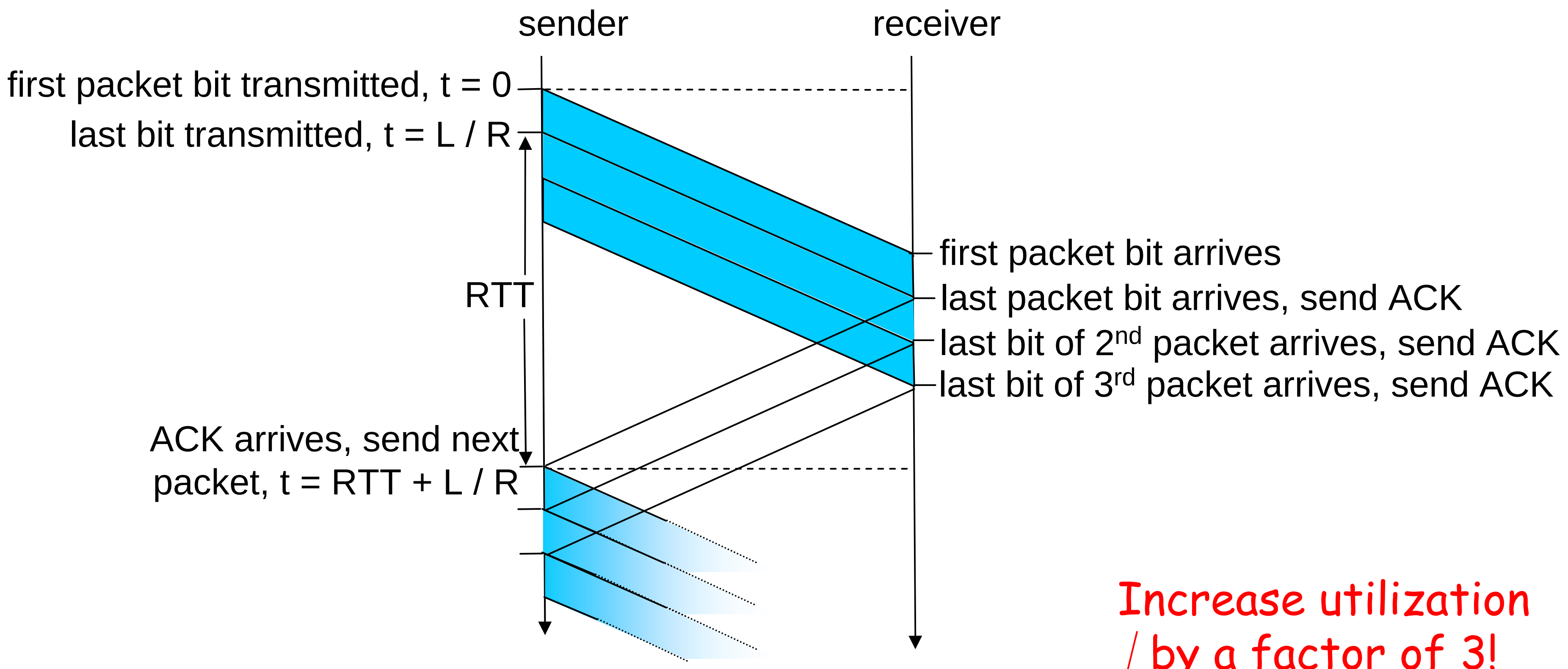
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

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

Pipelining: increased utilization

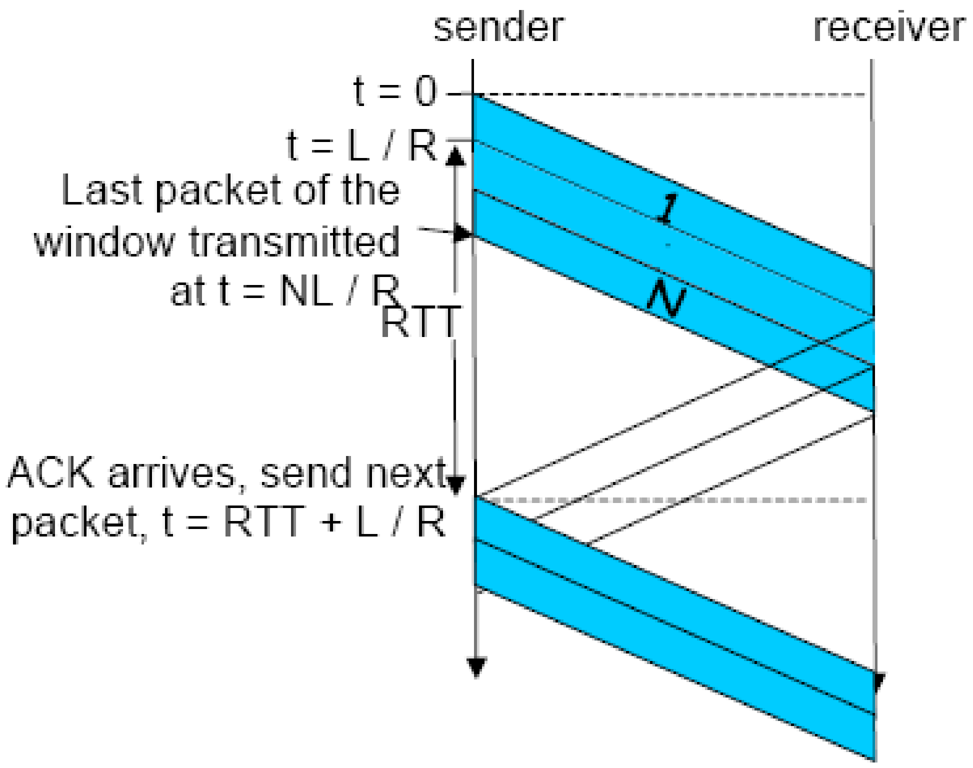

sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
ACK arrives, send next packet, t = RTT + L / R
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK

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

Increase utilization by a factor of 3!

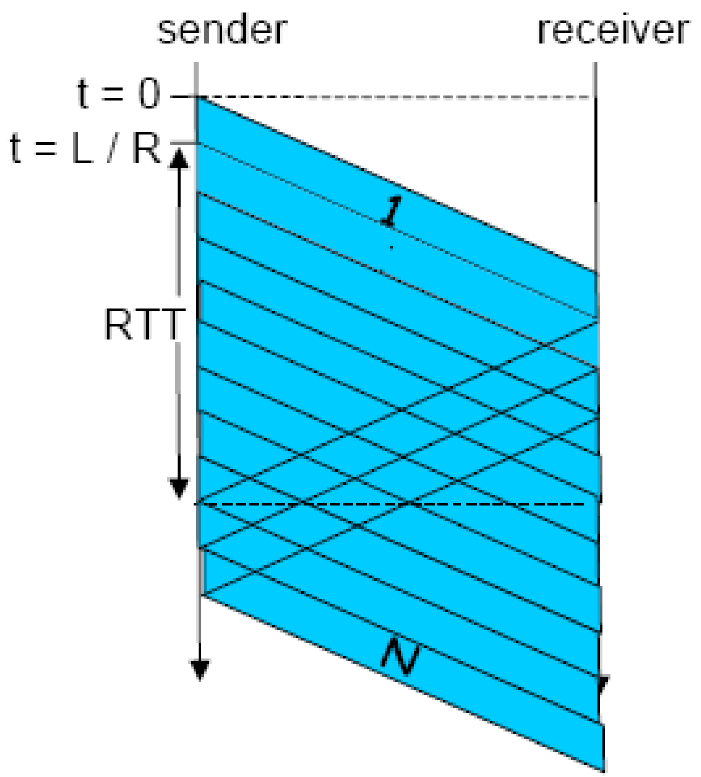
Window Size (N) vs. Utilization

Case 1.


sender
receiver
t = 0
t = L / R
Last packet of the window transmitted at t = NL / R
RTT
1
N
ACK arrives, send next packet, t = RTT + L / R

Utilization=N(L/R)/(RTT+L/R) if NL/R<RTT+L/R and the sender pauses after it transmits a window of packets until it receives first ACK

Case 2.

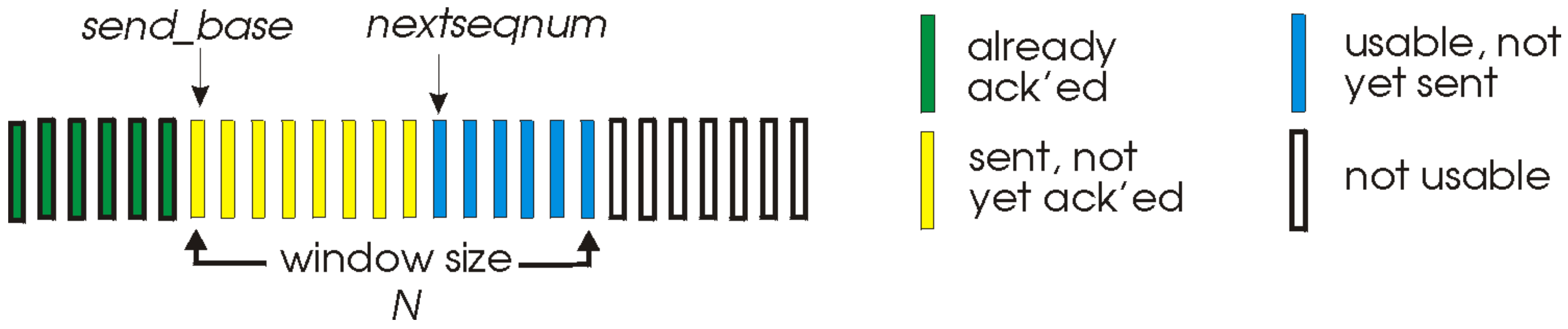


Utilization=1 if NL/R>RTT+L/R and the sender does not pause

Go-Back-N

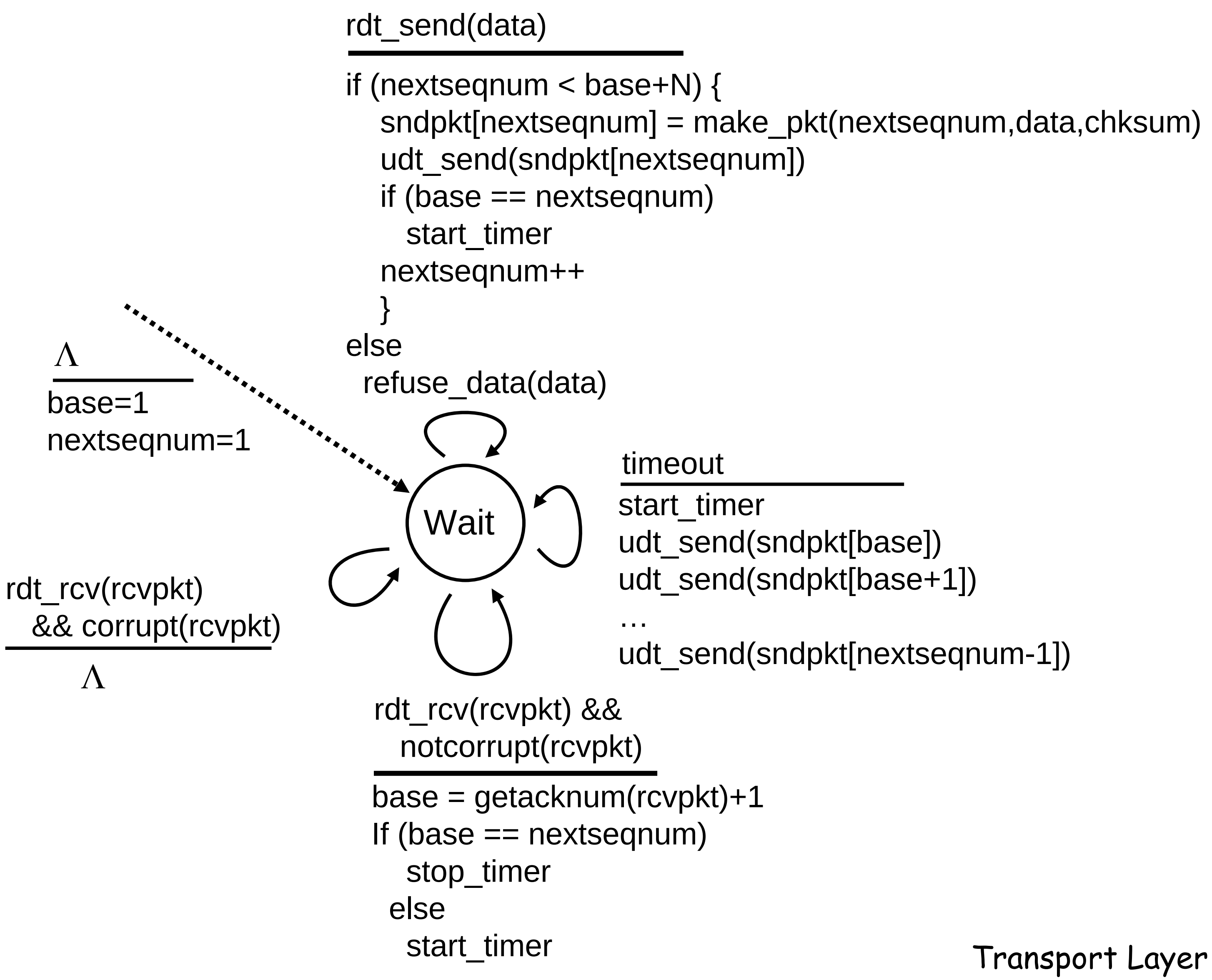
Sender:

- 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 the entire window
- *timeout(n):* retransmit pkt 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)
```

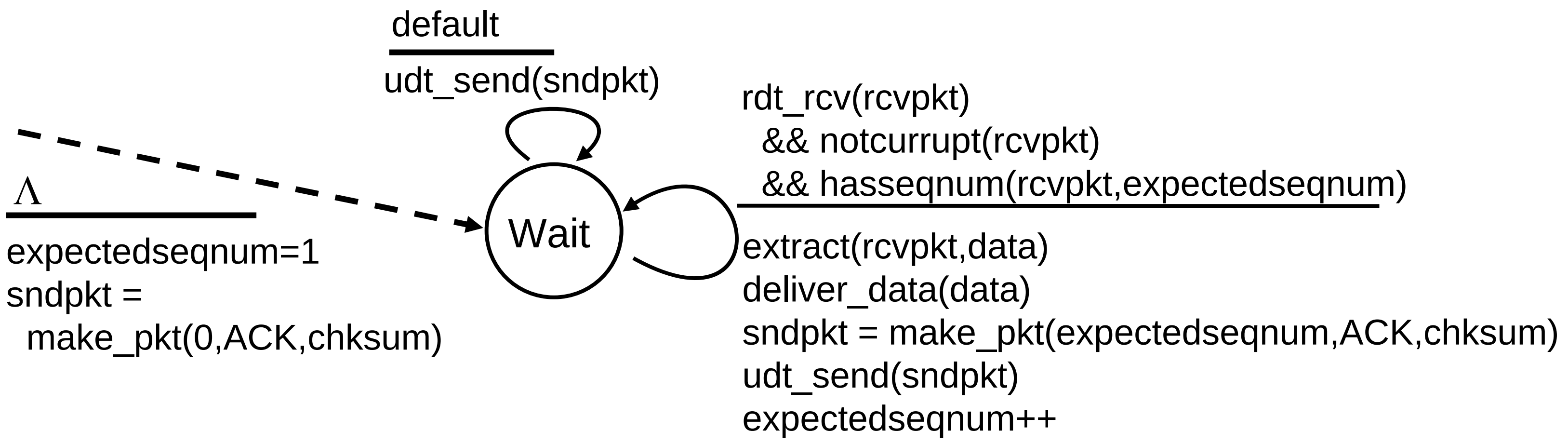
```
Λ
______________
base=1
nextseqnum=1
```

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

```
rdt_rcv(rcvpkt)
   && corrupt(rcvpkt)
______________________________
   Λ
```

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

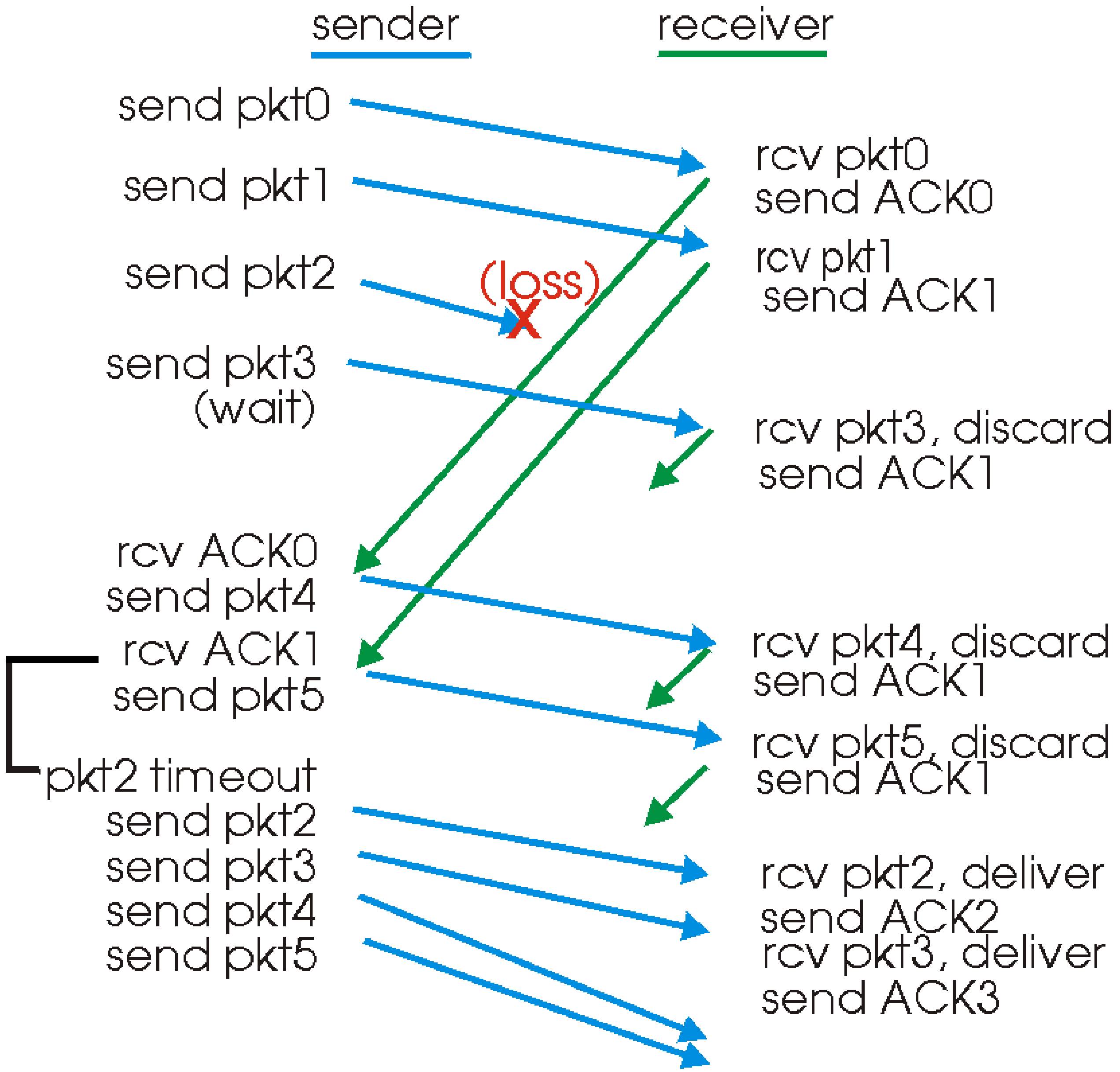
GBN: receiver extended FSM



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 #

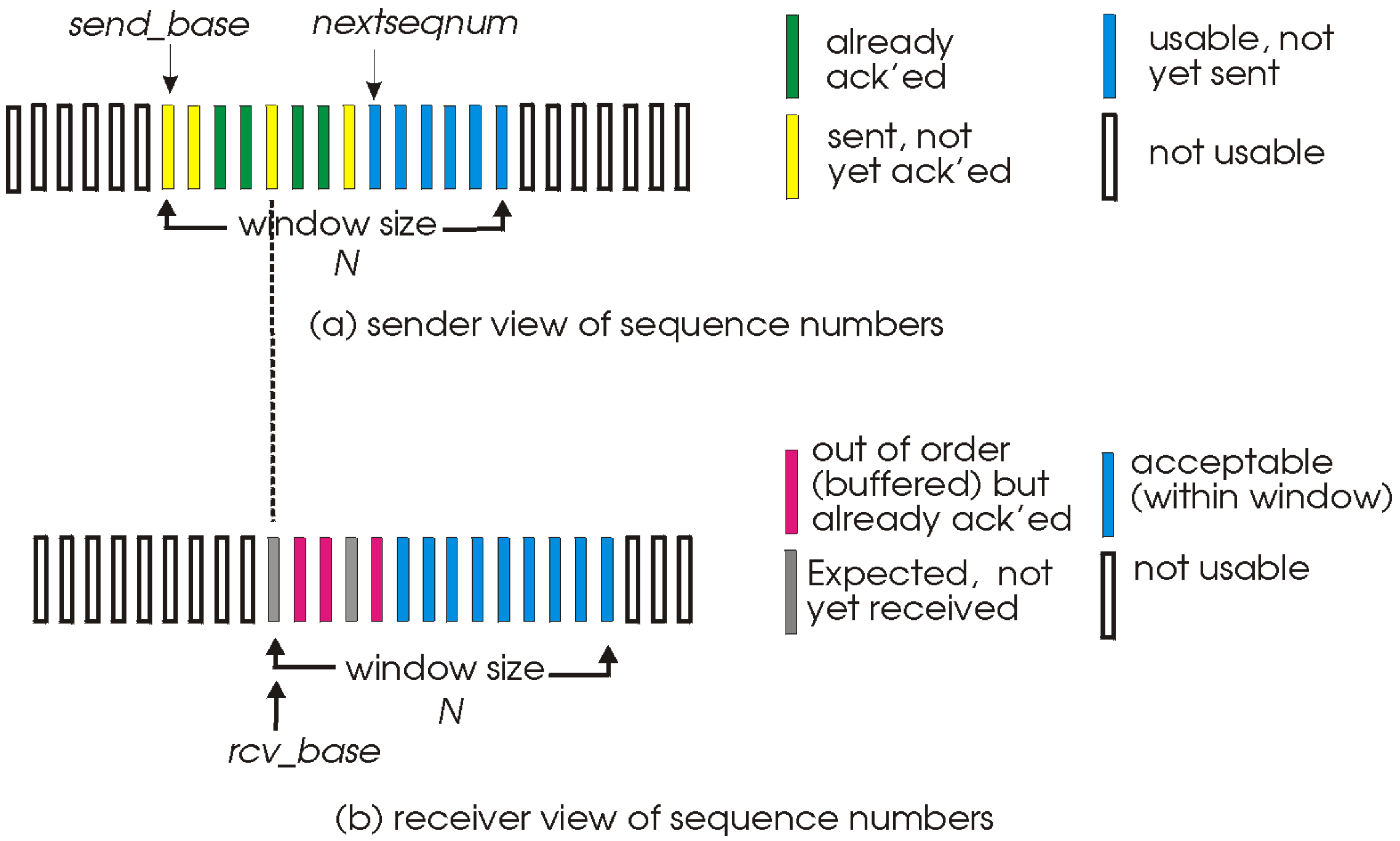
GBN in action



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
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



send_base
nextseqnum
already ack'ed
usable, not yet sent
sent, not yet ack'ed
not usable
window size N
(a) sender view of sequence numbers
out of order (buffered) but already ack'ed
acceptable (within window)
Expected, not yet received
not usable
window size N
rcv_base
(b) receiver view of sequence numbers

Selective repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N-1]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

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

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

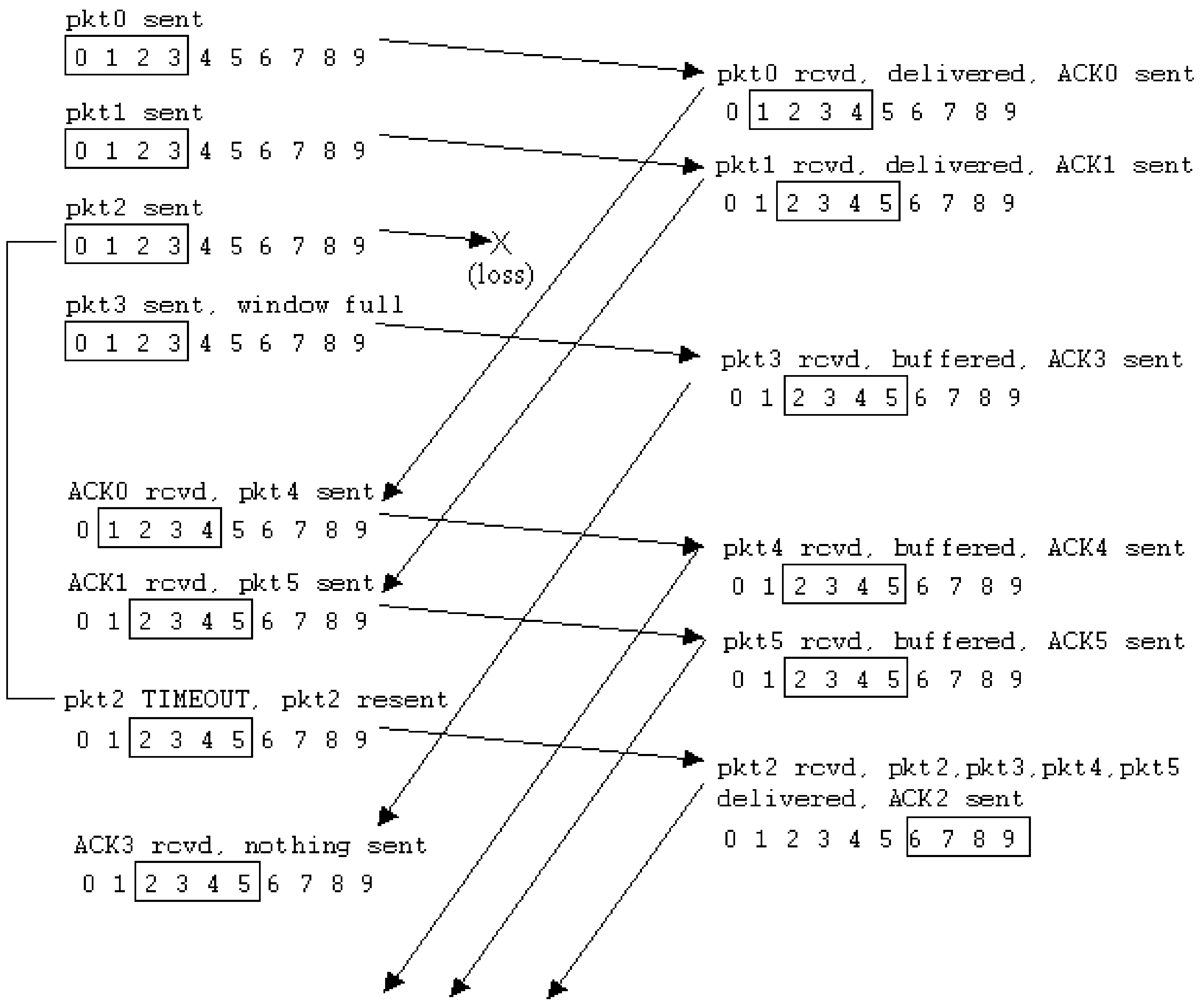
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



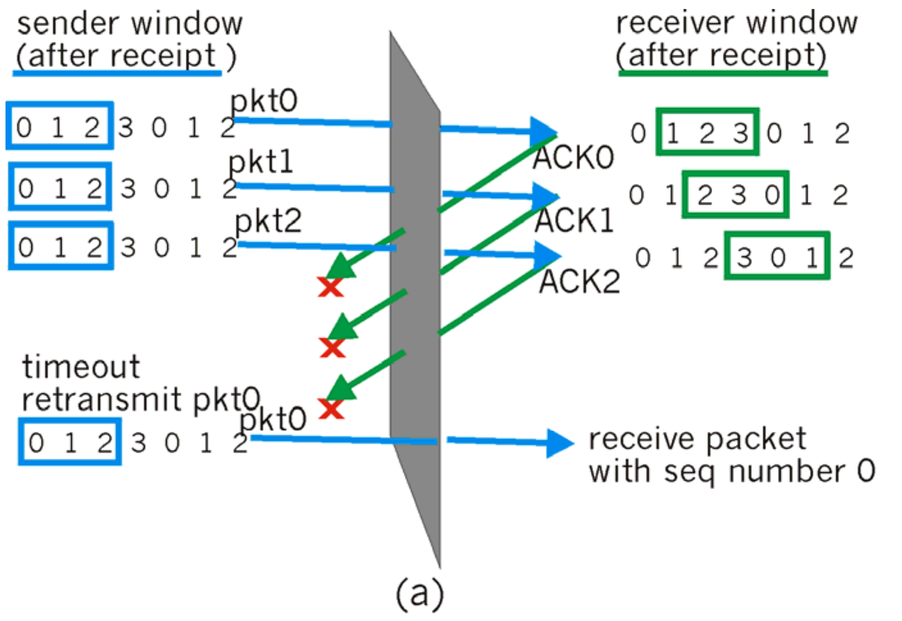
Selective repeat: dilemma

Example:

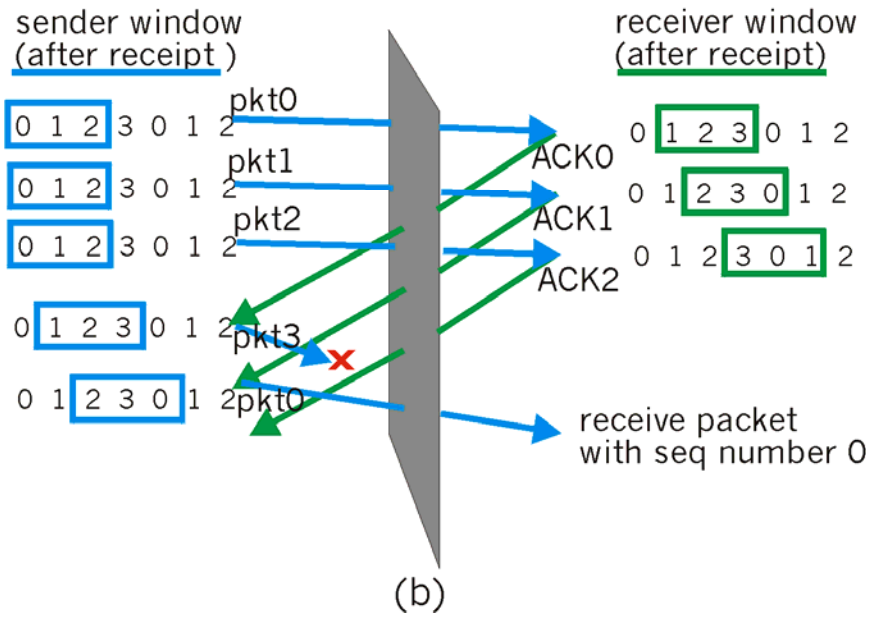
- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?



sender window (after receipt)
receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
timeout retransmit pkt0
pkt0
receive packet with seq number 0
(a)



sender window (after receipt)
receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
pkt3
pkt0
receive packet with seq number 0
(b)

Sequence Number vs. Window Size

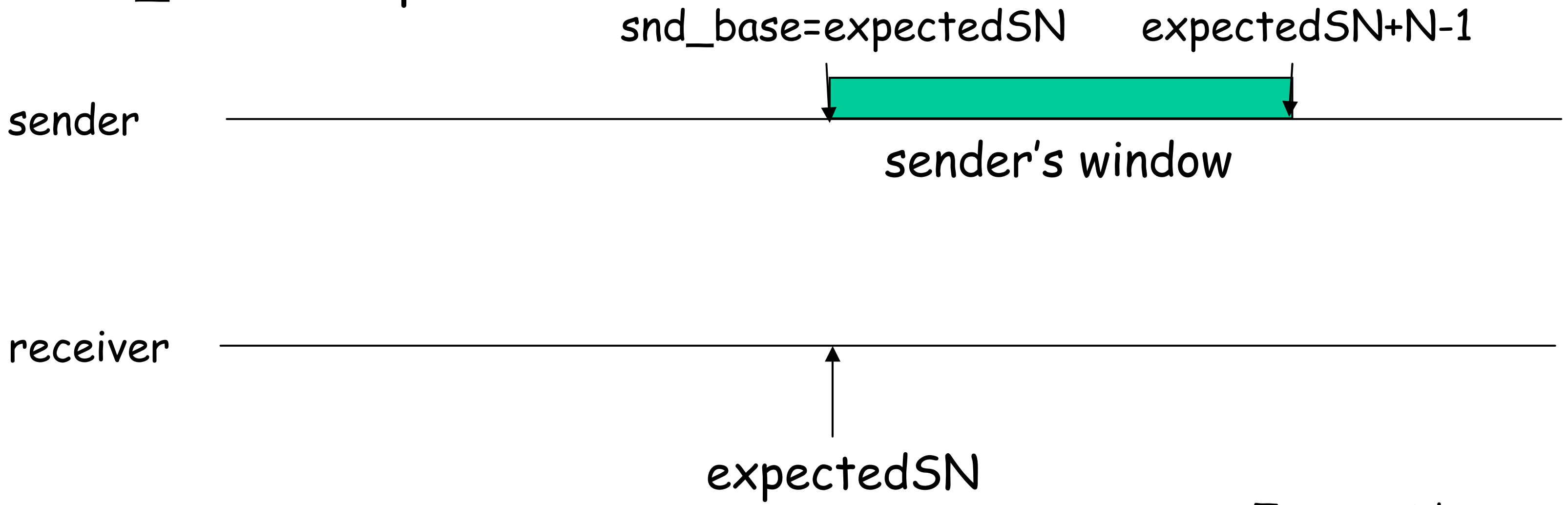
Suppose we use k bits to represent SN

Question: What's the minimum number of bits k necessary for a window size of N?

Go-Back-N

Q: For a given expectedSN, what's the *largest possible value* for snd_base?

A: If all the last N ACKs sent by the receiver **are** received,
snd_base = expectedSN



Sequence Number vs. Window Size

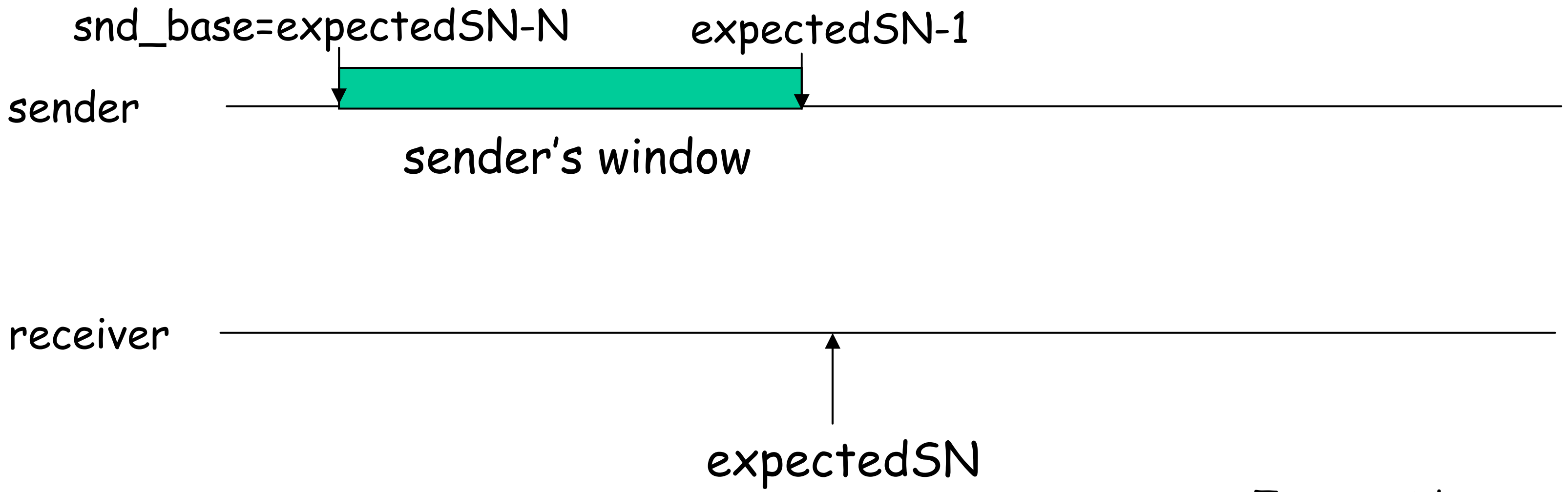
Suppose we use k bits to represent SN

Question: What's the minimum number of bits k necessary for a window size of N?

Go-Back-N

Q: For a given expectedSN, what's the *smallest possible value* for snd_base?

A: If all the last N ACKs sent by the receiver **are not** received,
snd_base = expectedSN-N

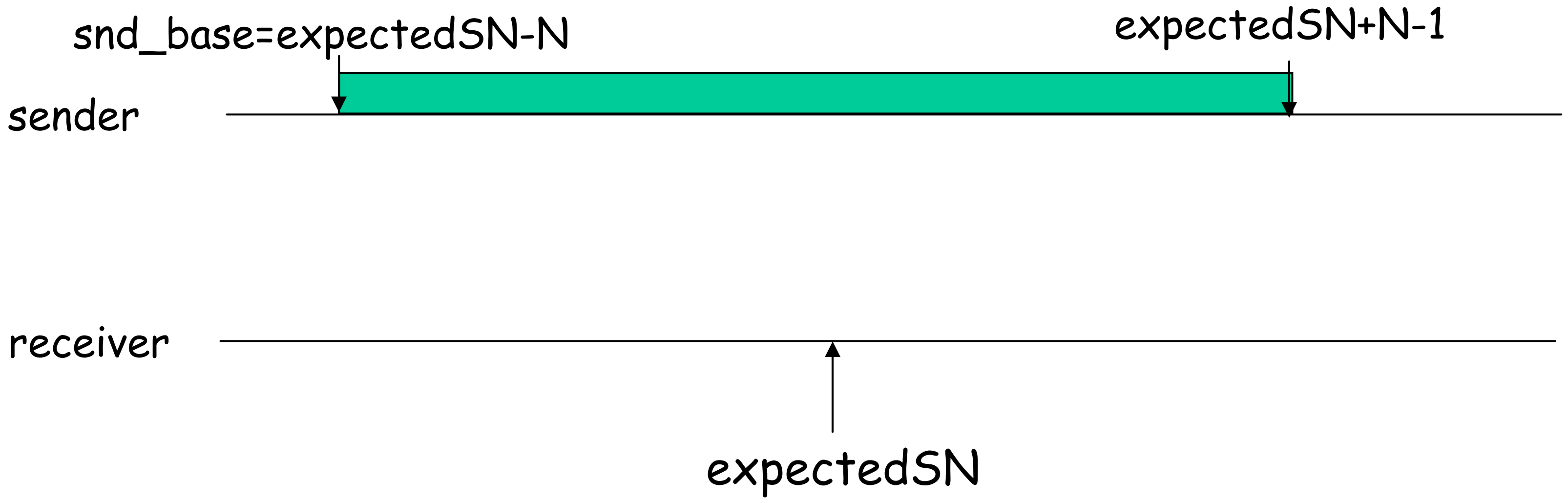


Sequence Number vs. Window Size

Go-Back-N

All SNs in the interval [expectedSN-N,expectedSN+N-1] (an interval of size 2N) can be received by the receiver. Since the receiver accepts on the packet with SN=expectedSN, there should be no other packet within this interval with SN=expectedSN. Therefore,

$$\boxed{2^k \geq N+1}$$



Sequence Number vs. Window Size

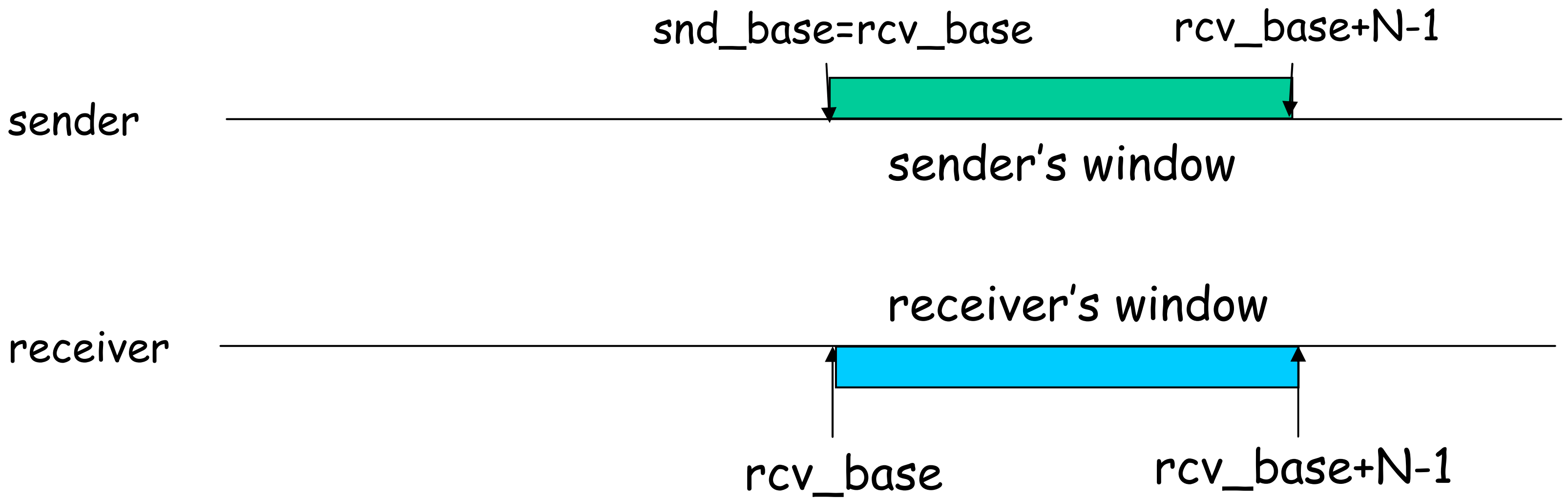
Suppose we use k bits to represent SN

Question: What's the minimum number of bits k necessary for a window size of N?

Selective Repeat

Q: For a given rcv_base, what's the *largest possible value* for snd_base?

A: If all the last N ACKs sent by the receiver **are** received, snd_base = rcv_base (same as go_back-N)



Sequence Number vs. Window Size

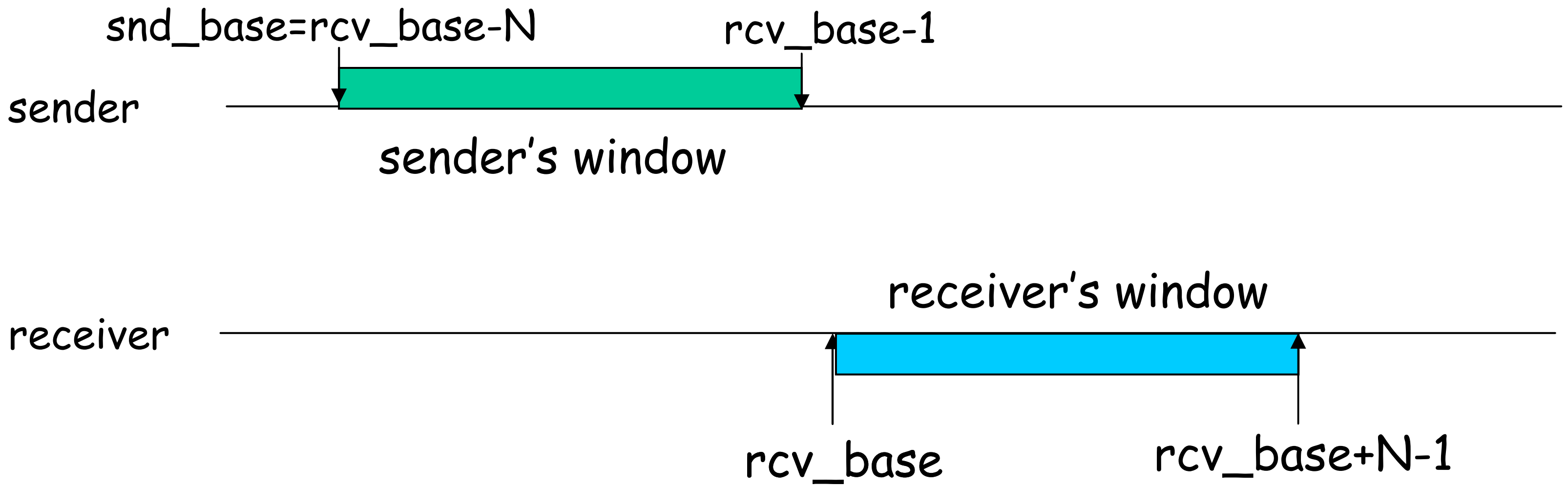
Suppose we use k bits to represent SN

Question: What's the minimum number of bits k necessary for a window size of N?

Selective Repeat

Q: For a given rcv_base, what's the *smallest possible value* for snd_base?

A: If all the last N ACKs sent by the receiver **are not** received, snd_base = rcv_base-N (same as Go-Back-N)

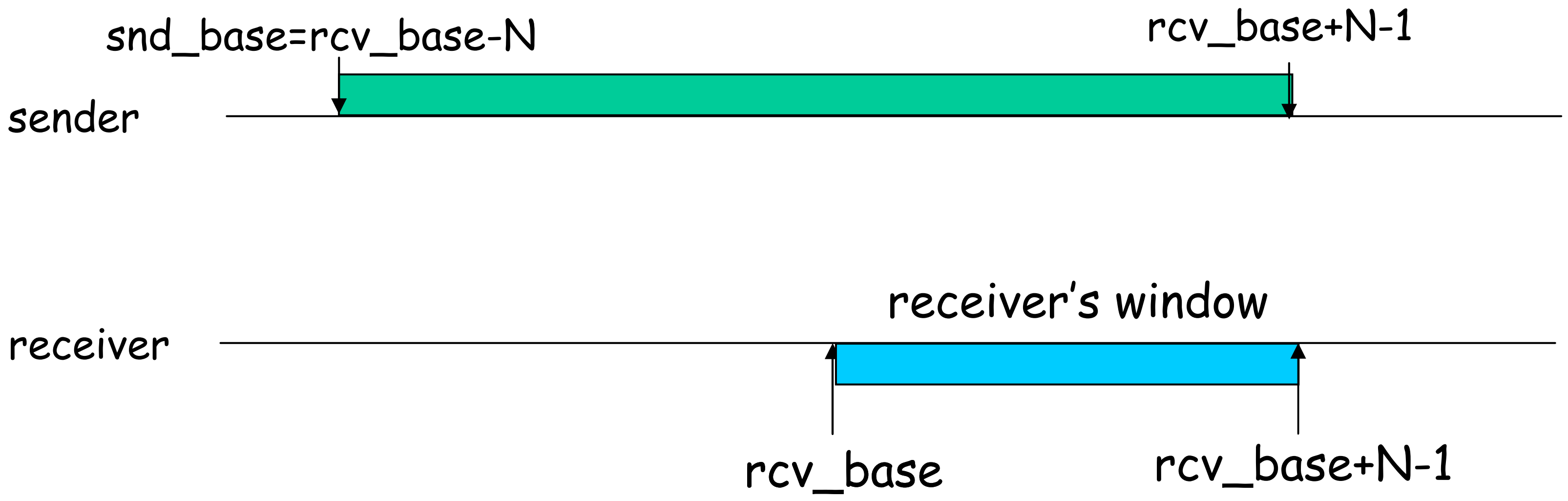


Sequence Number vs. Window Size

Selective Repeat

All SNs in the interval [rcv_base-N,rcv_base+N-1] (an interval of size 2N) can be received by the receiver. Since the receiver should be able to distinguish between all packets in this interval and take corresponding action, there should be no two packets within this interval having the same SN. Therefore,

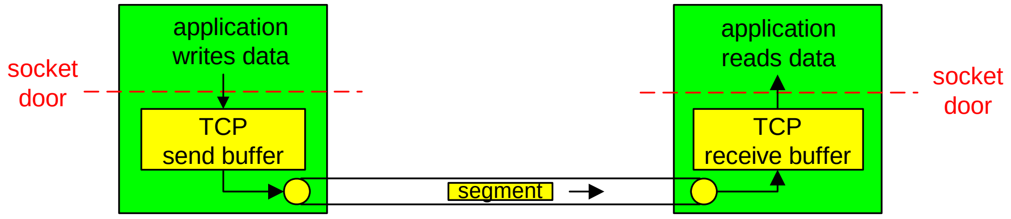
$$\boxed{2^k \geq 2N}$$



TCP: Overview

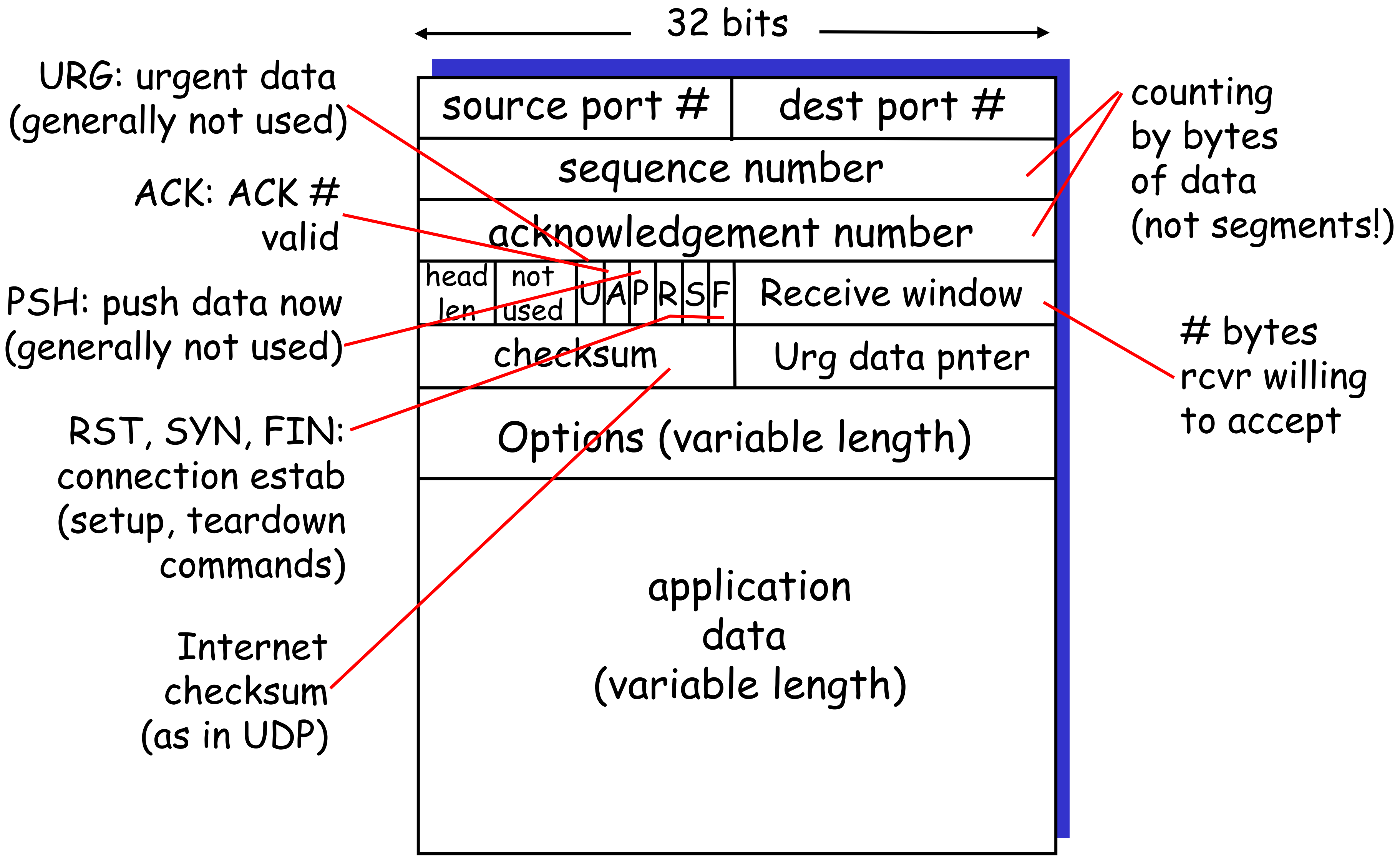
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



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

TCP segment structure


32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
Receive window
checksum
Urg data pnter
Options (variable length)
application data (variable length)
URG: urgent data (generally not used)
ACK: ACK # valid
PSH: push data now (generally not used)
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
counting by bytes of data (not segments!)
# bytes rcvr willing to accept

TCP seq. #'s and ACKs

Seq. #'s:

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

ACKs:

- 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 implementation
- Widely used implementations of TCP buffer out-of-order segments

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

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

Q: how to estimate RTT?

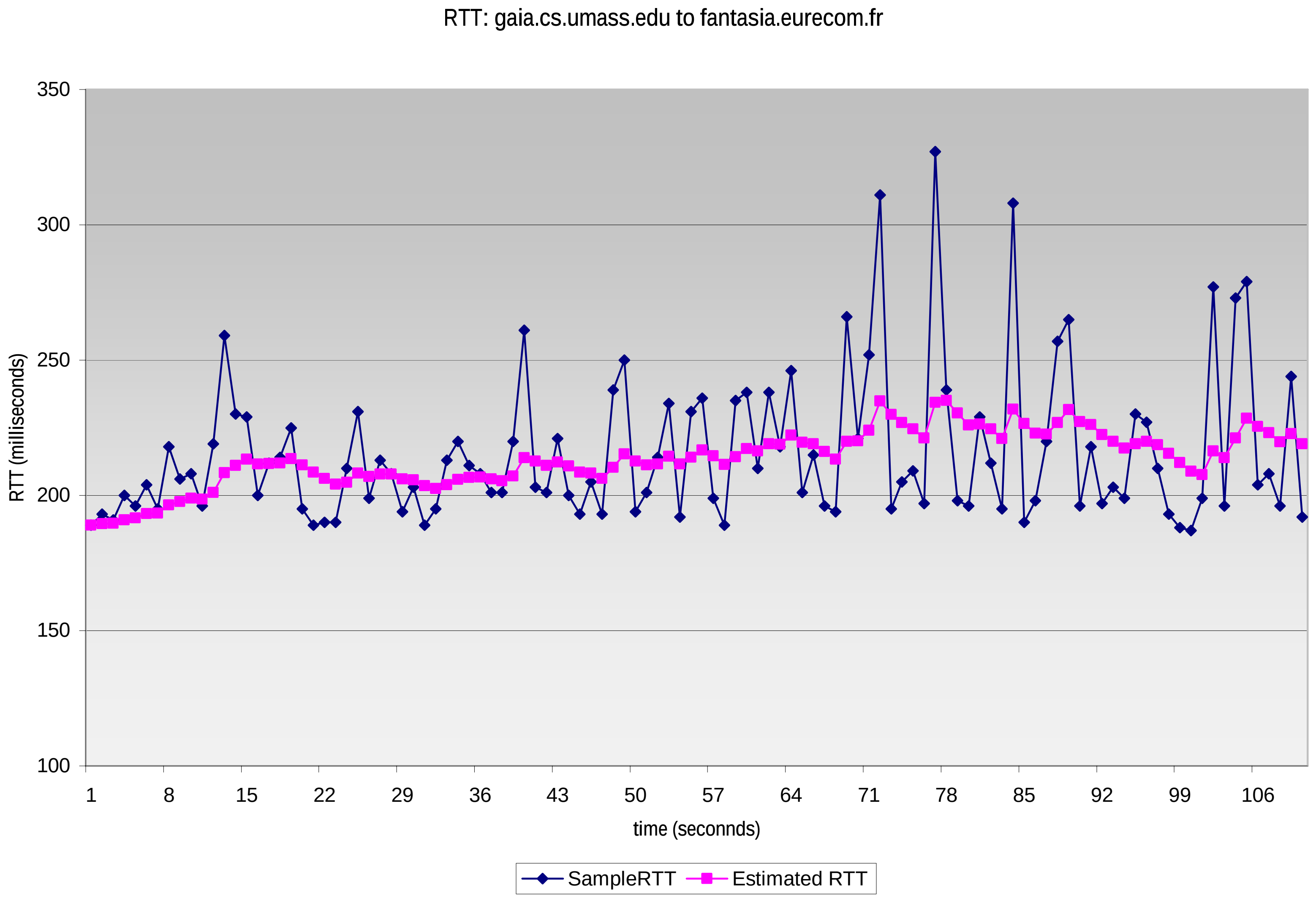
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

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

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

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** **->** larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer; however it just retransmits the first segment in the window
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

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

timeout:

- retransmit segment that caused timeout (first segment in the window)
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

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

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

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

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

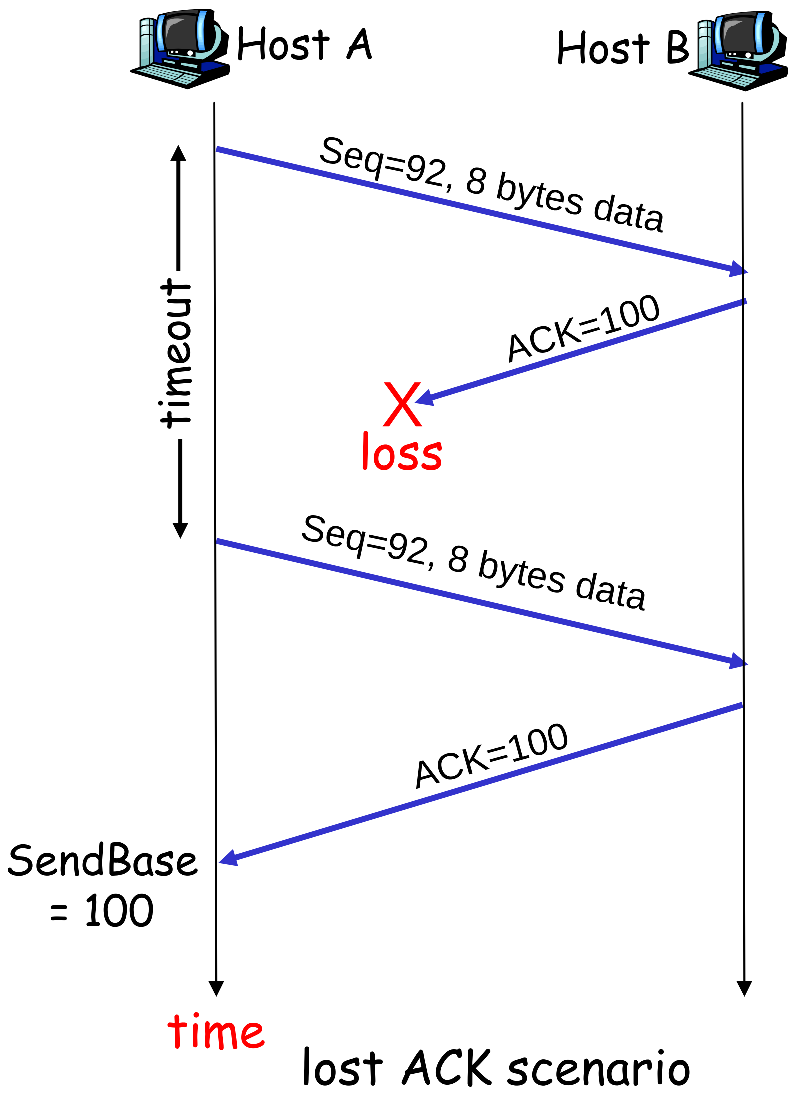
Comment:

- SendBase-1: last cumulatively ack'ed byte

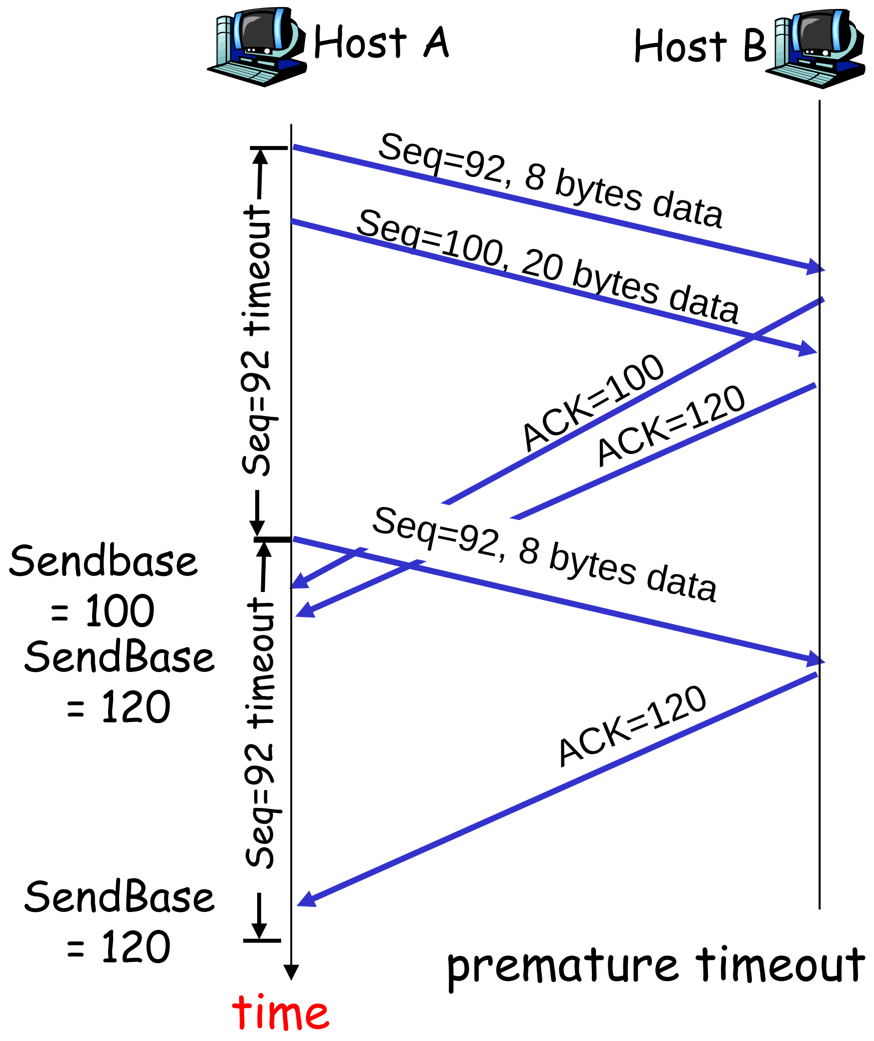
Example:

- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: retransmission scenarios

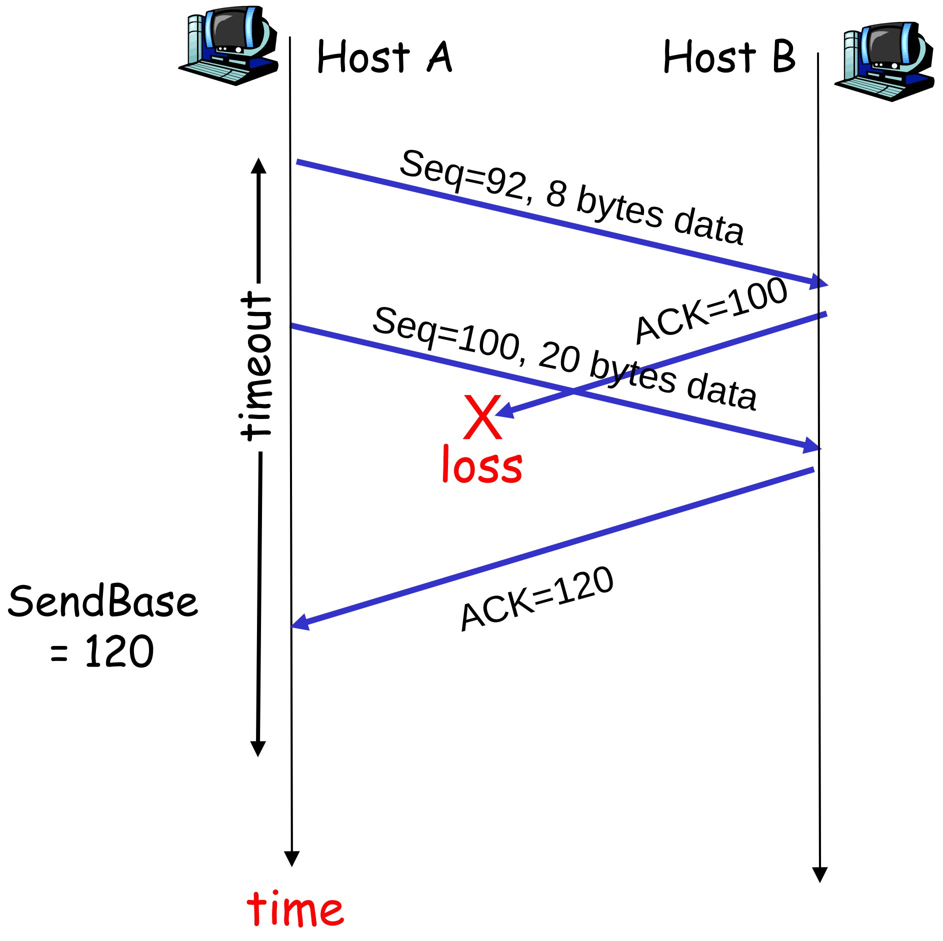


lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



Cumulative ACK scenario

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 startsat lower end of gap

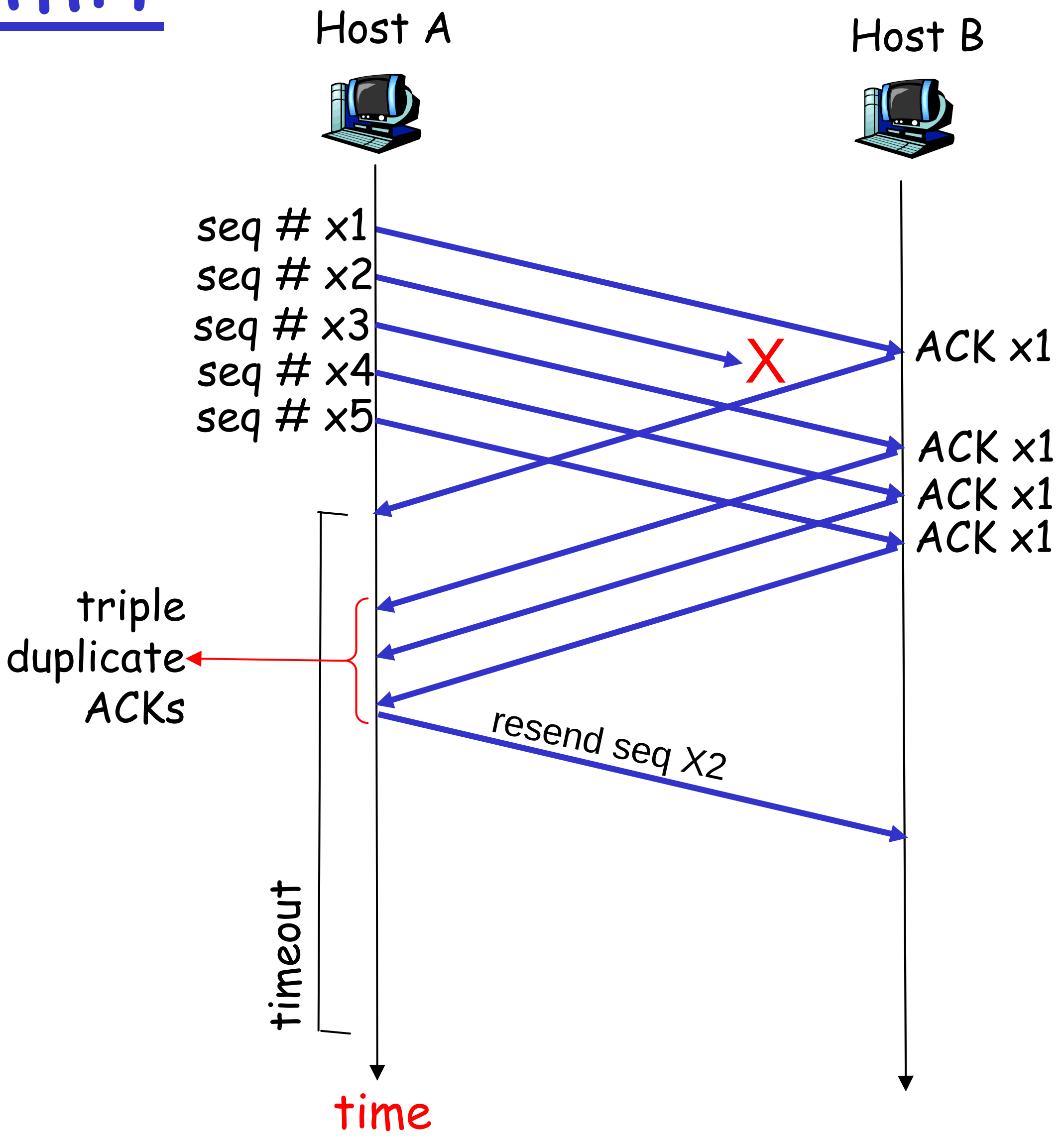
Fast Retransmit

- 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.

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast Retransmit

- Resend a segment after 3 duplicate ACKs since a duplicate ACK means that an out-of sequence segment was received
- duplicate ACKs due to packet reordering!
- if window is small don't get duplicate ACKs!



Fast retransmit algorithm:

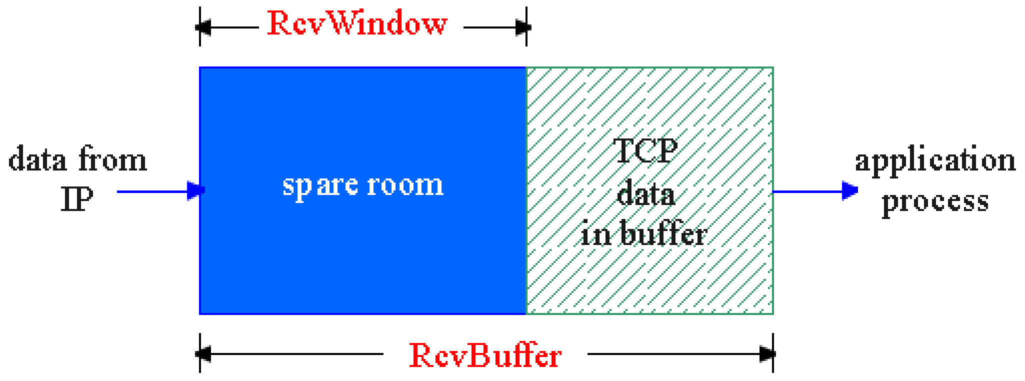
```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                 SendBase = y
                 if (there are currently not-yet-acknowledged segments)
                      start timer
               }
            else {
                   increment count of dup ACKs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

a duplicate ACK for already ACKed segment

fast retransmit

TCP Flow Control

- receive side of TCP connection has a receive buffer:



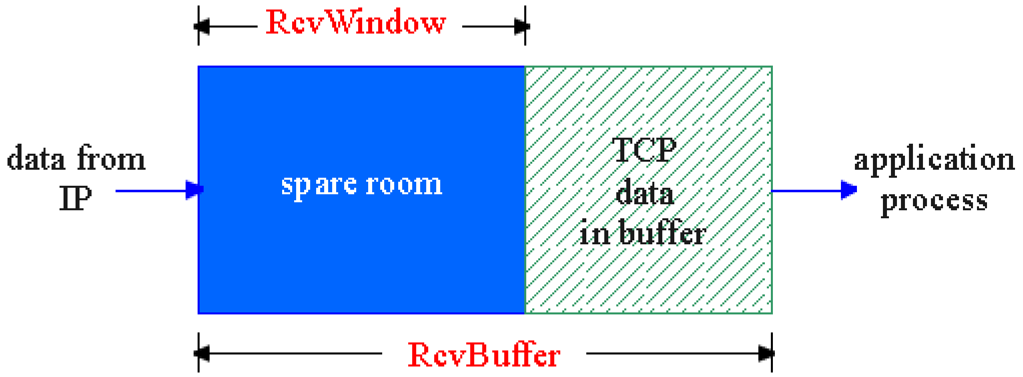
- app process may be slow at reading from buffer

flow control

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

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

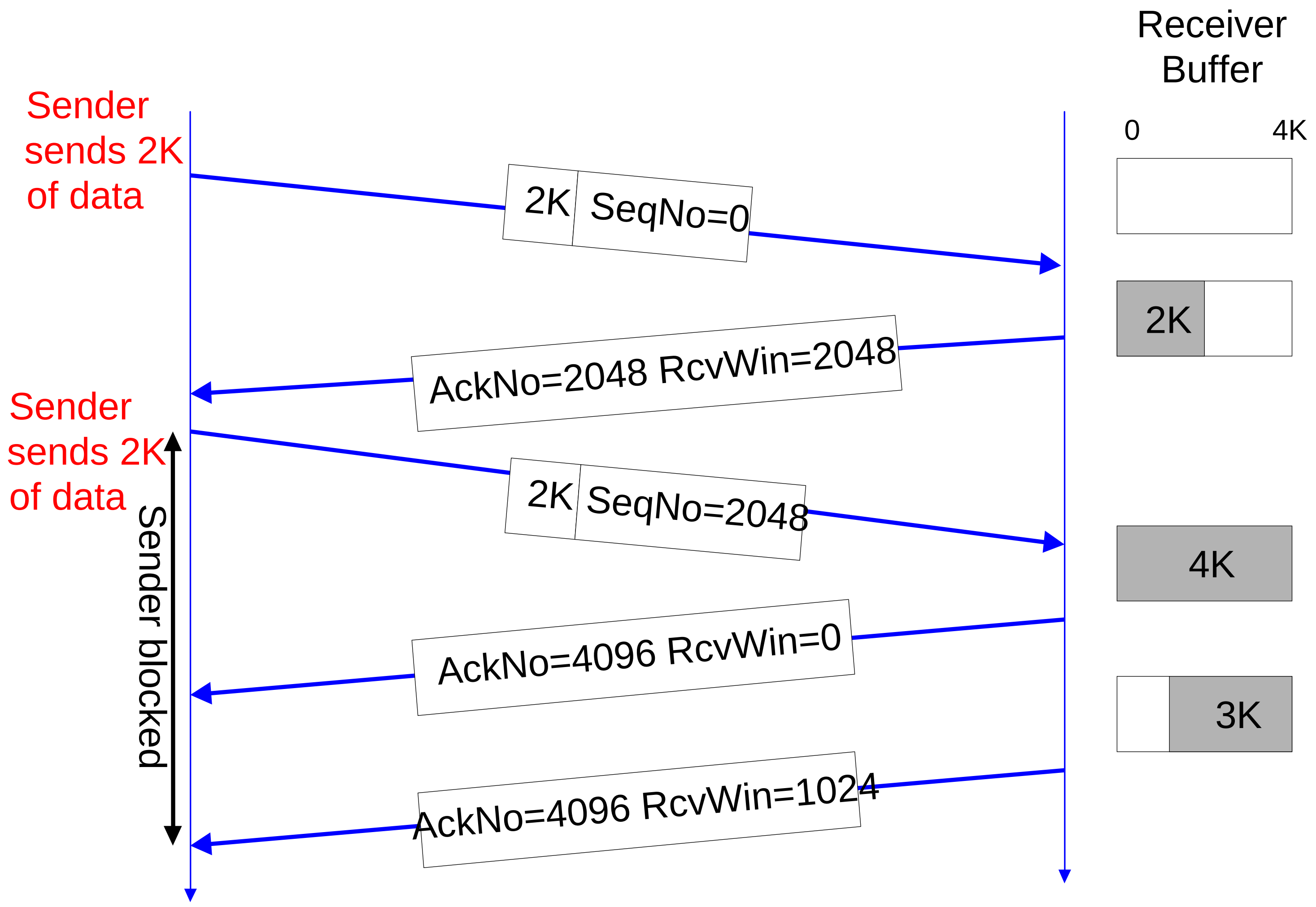
- spare room in buffer

`= RcvWin`

`= RcvBuffer-[LastByteRcvd - LastByteRead]`

- Rcvr advertises spare room by including value of **`RcvWin`** in segments
- Sender limits unACKed data to **`RcvWin`**
  - guarantees receive buffer doesn't overflow

Sliding Window Flow Control Example



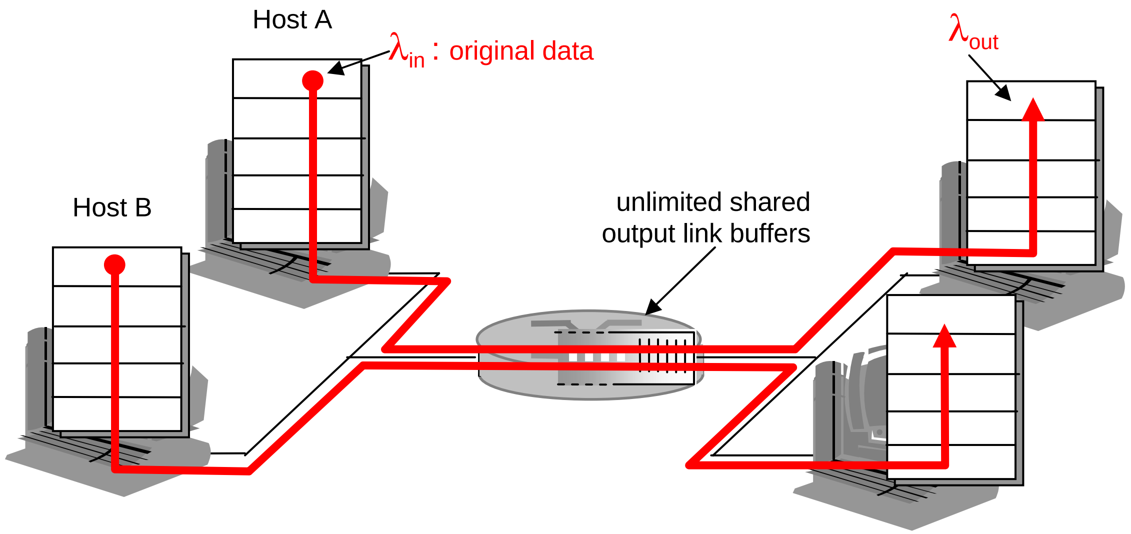
Principles of Congestion Control

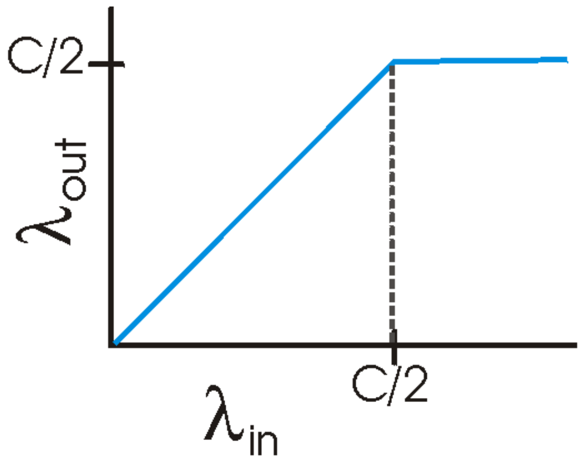
Congestion:

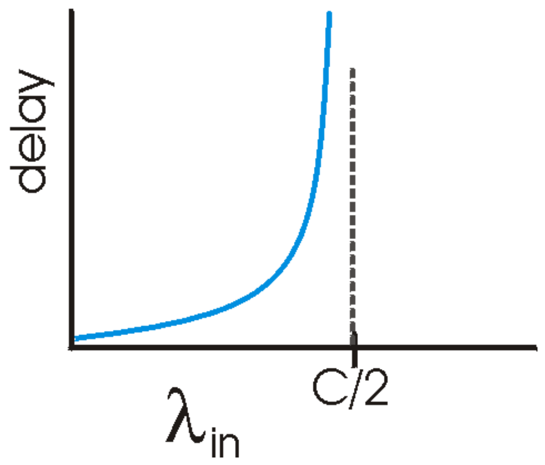
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission


Host A
λ_{in} : original data
λ_{out}
Host B
unlimited shared output link buffers

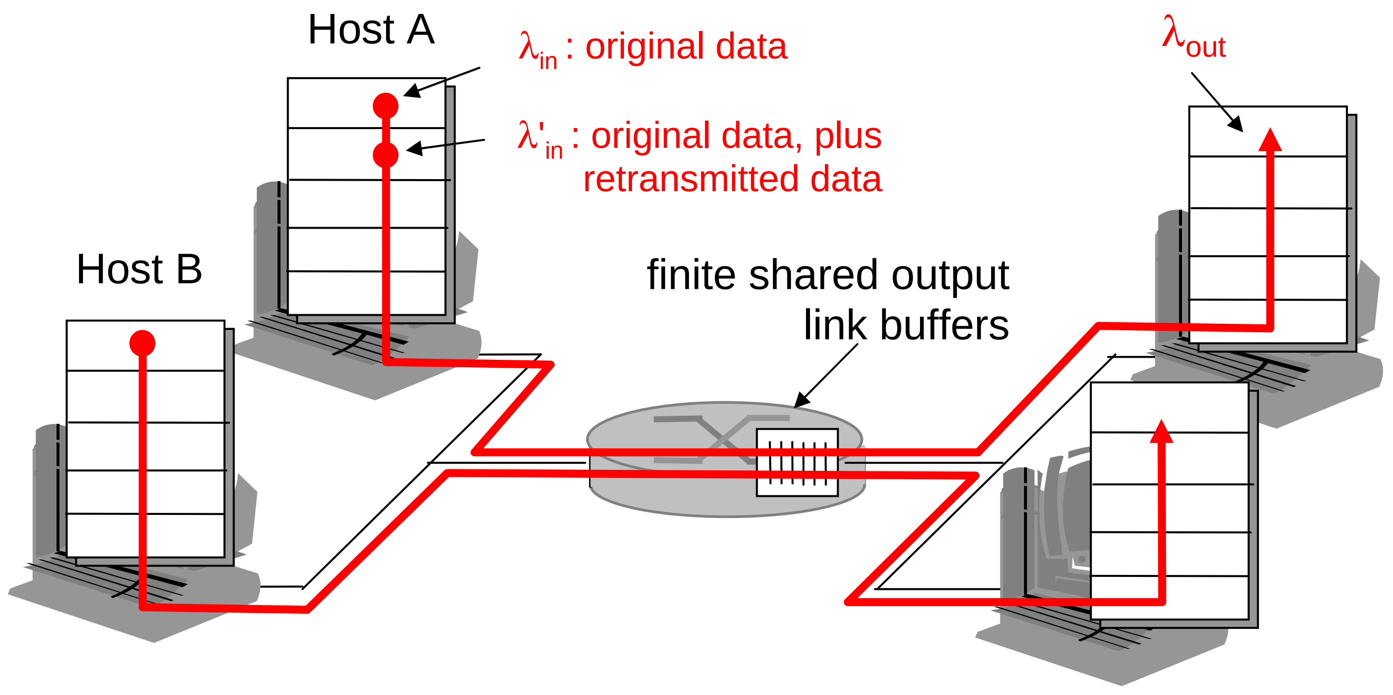




- large delays when congested
- maximum achievable throughput

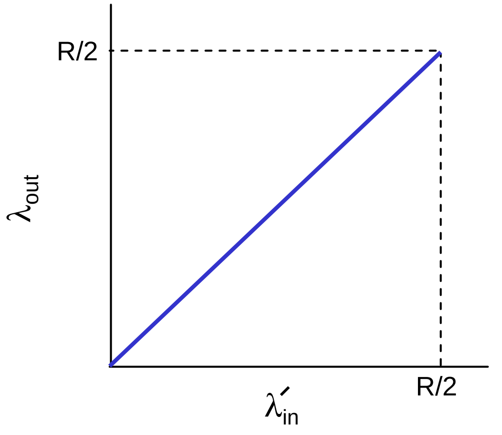
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet

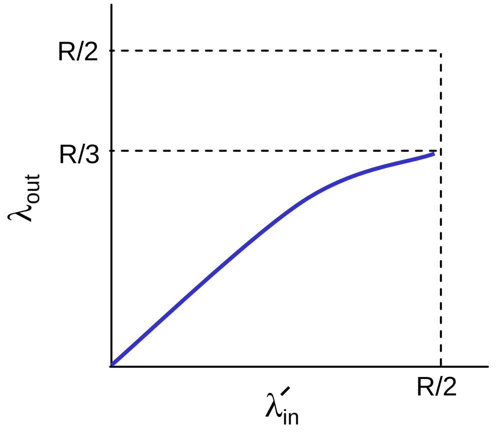

Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
Host B
finite shared output link buffers
λ_{out}

Causes/costs of congestion: scenario 2

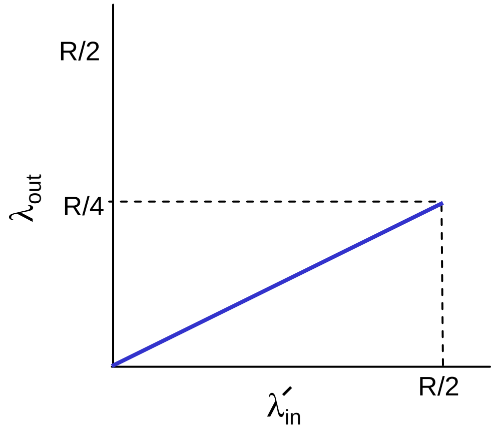
- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a.



b.



c.

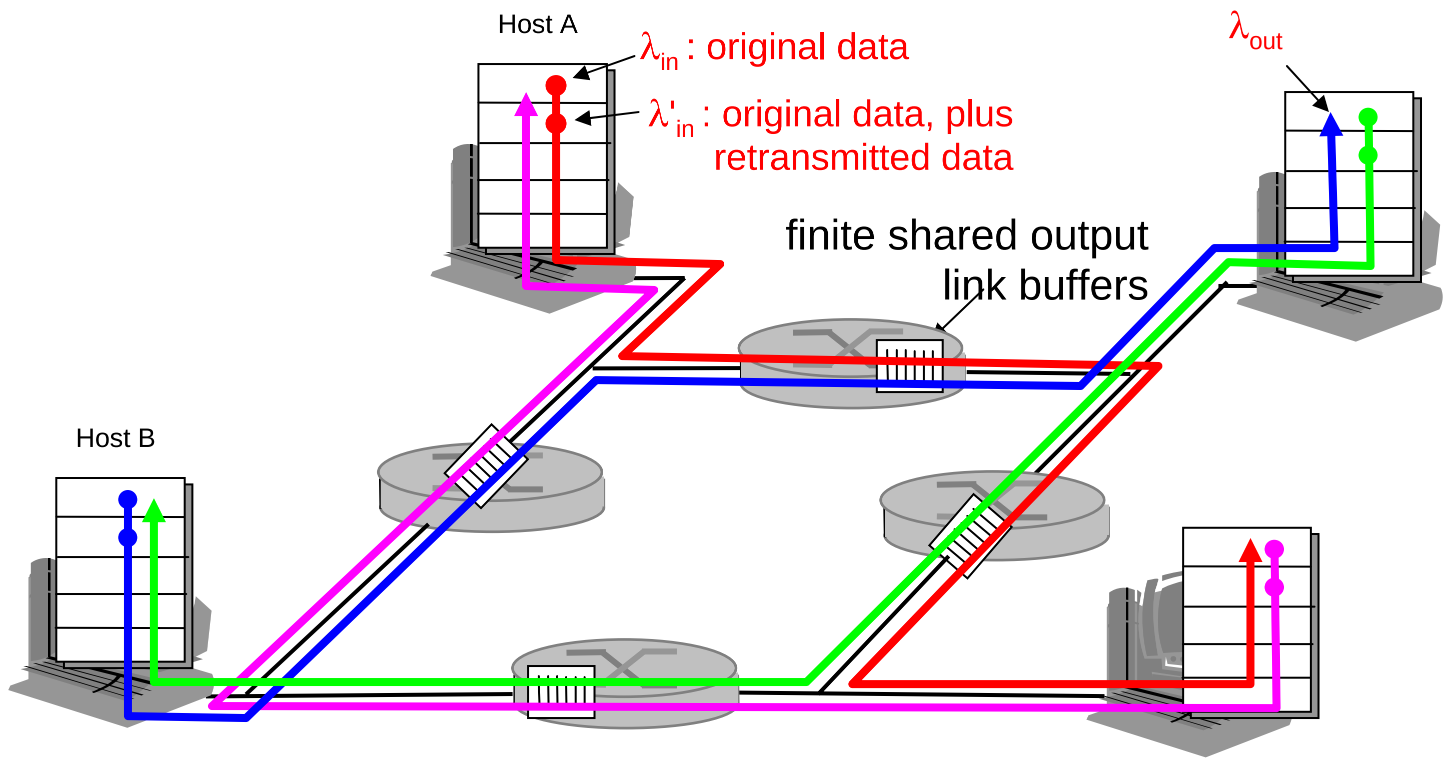
"costs" of congestion:

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

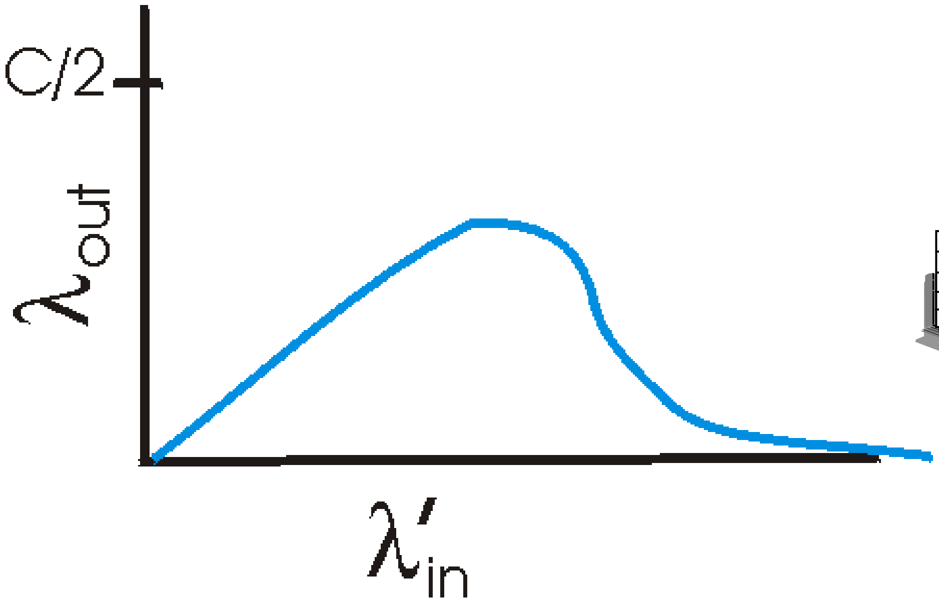
Causes/costs of congestion: scenario 3

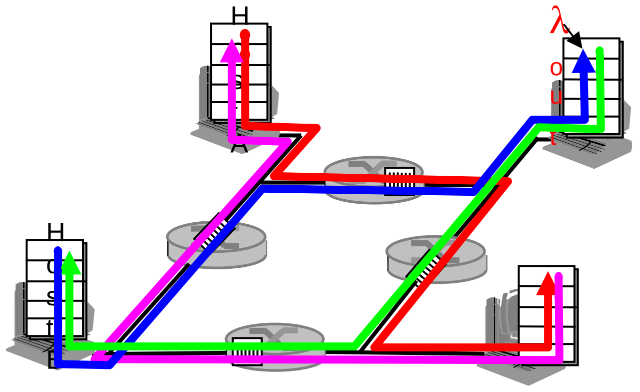
- four senders
- multihop paths
- timeout/retransmit

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


Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
finite shared output link buffers
λ_{out}
Host B

Causes/costs of congestion: scenario 3





another "cost" of congestion:

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

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

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

Network-assisted congestion control:

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

TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:
  `LastByteSent-LastByteAcked ≤ CongWin`
- `CongWin` is dynamic, function of perceived network congestion

How does sender perceive congestion?

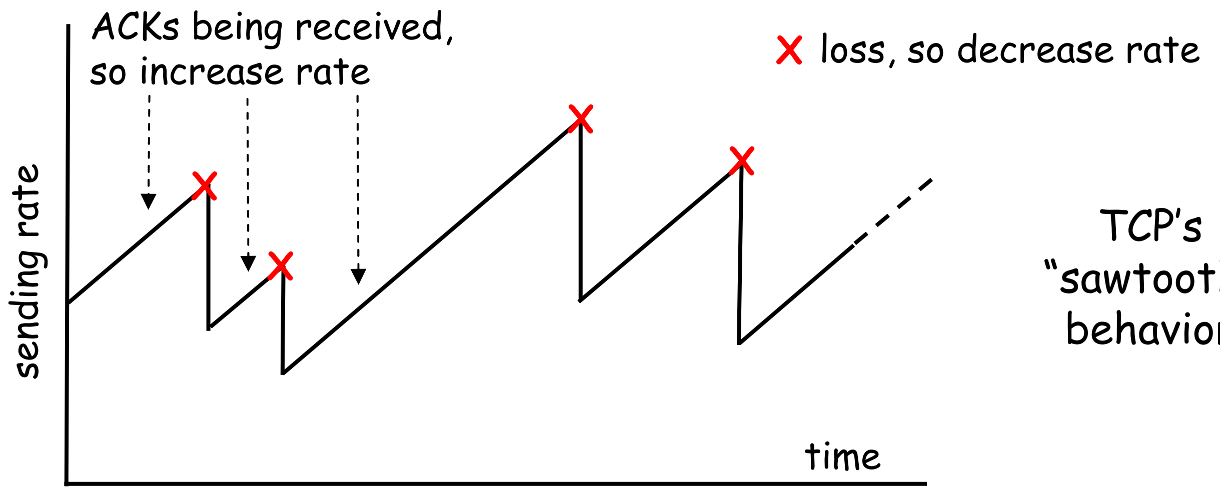
- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (`CongWin`) after loss event

two modes of operation:

  - Slow Start (SS)
  - Congestion avoidance (CA) or Additive Increase Multiplicative Decrease (AIMD)

TCP congestion control: bandwidth probing

- "probing for bandwidth": increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate
  - continue to increase on ACK, decrease on loss (since available bandwidth is changing, depending on other connections in network)



TCP's "sawtooth" behavior

- Q: how fast to increase/decrease?
  - details to follow

TCP Congestion Control: details

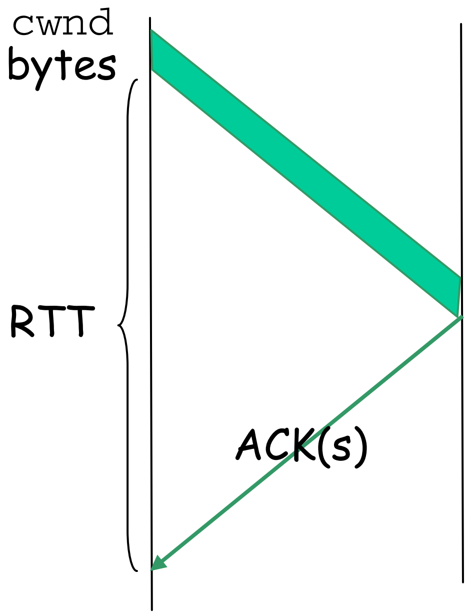
- sender limits rate by limiting number of unACKed bytes "in pipeline":

  LastByteSent-LastByteAcked ≤ cwnd

  - **cwnd:** differs from **rwnd** (how, why?)
  - sender limited by **min(cwnd,rwnd)**
- roughly,

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

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



TCP Congestion Control: more details

segment loss event: reducing `cwnd`

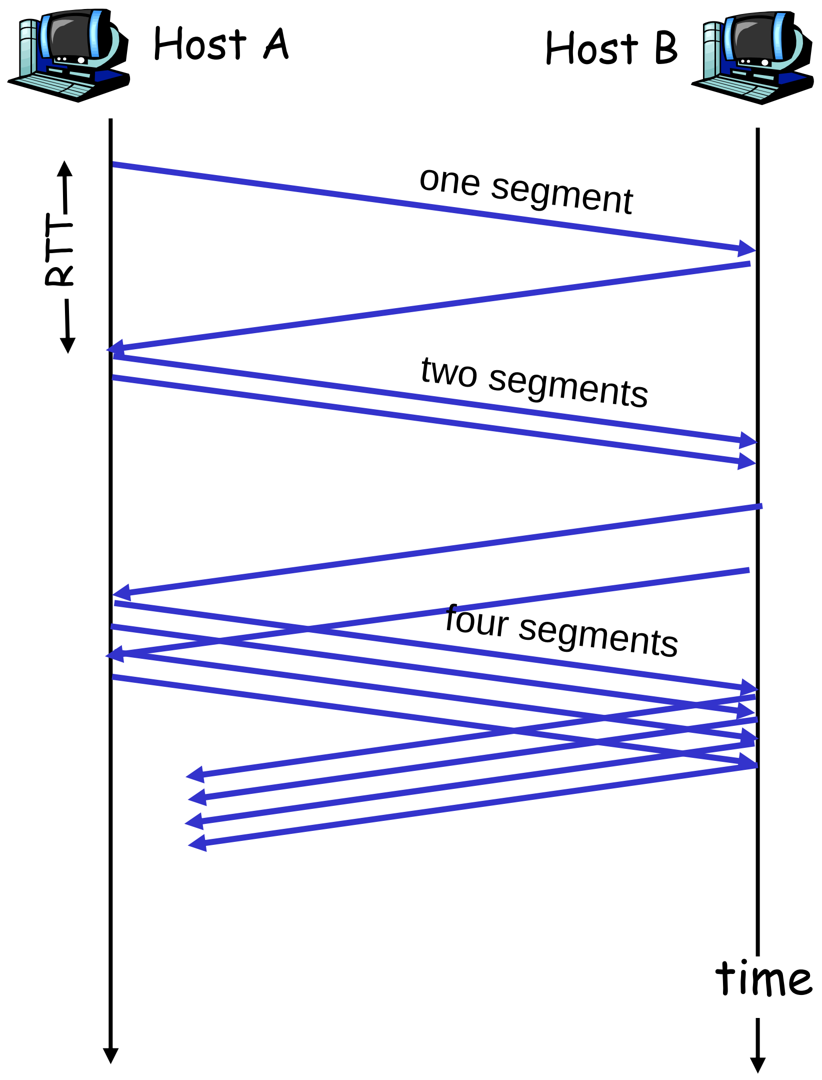
- timeout: no response from receiver
  - cut `cwnd` to 1
- 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
  - cut `cwnd` in half, less aggressively than on timeout

ACK received: increase `cwnd`

- Two modes of operation:
  - slowstart phase:
    - increase exponentially fast (despite name) at connection start, or following timeout
  - congestion avoidance:
    - increase linearly

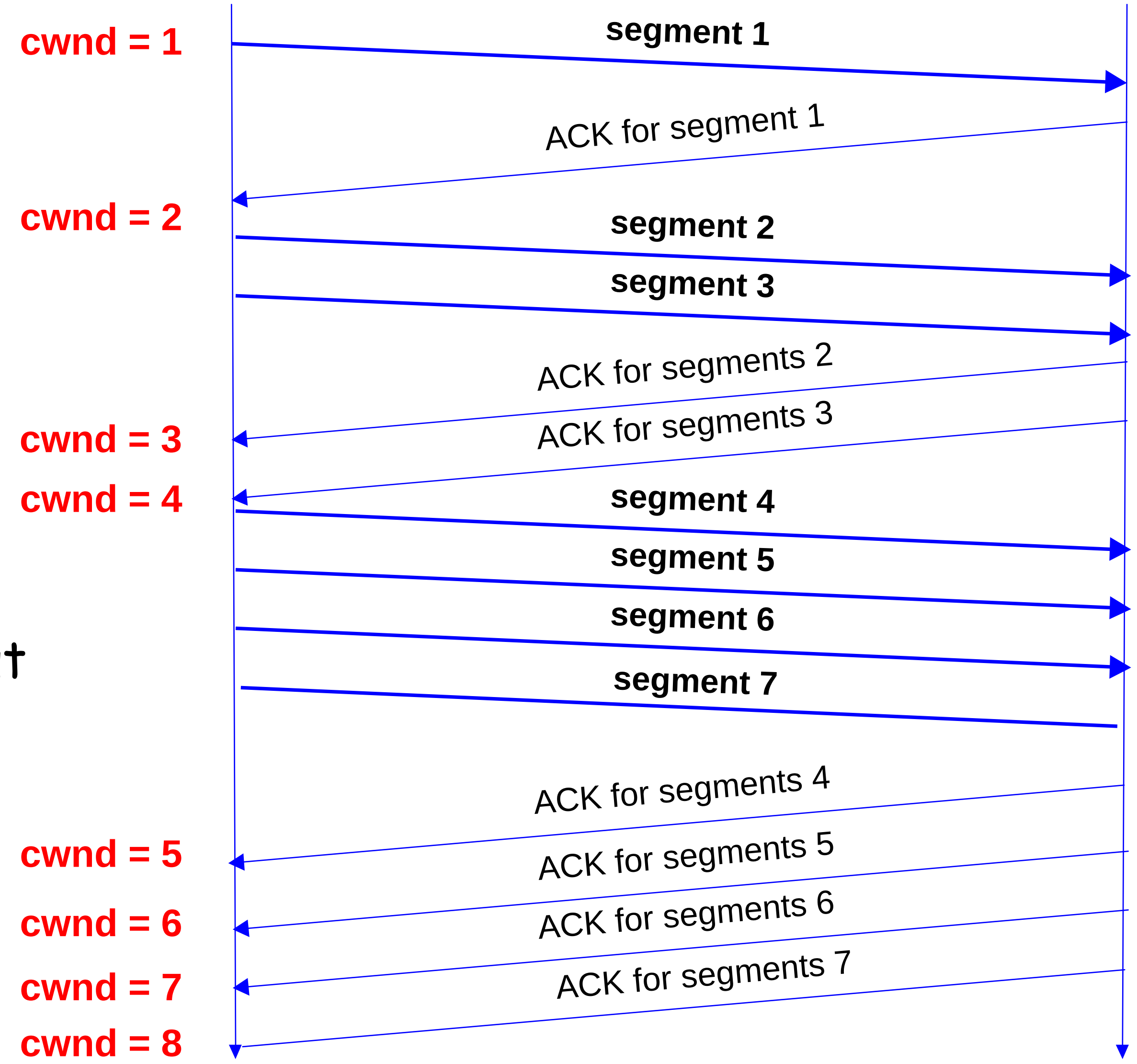
TCP Slow Start Phase

- when connection begins, **cwnd** = 1 MSS
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double **cwnd** every RTT
  - done by incrementing **cwnd** by 1 for every ACK received



Slow Start Example

- The congestion window size grows very rapidly
  - For every ACK, we increase CongWin by 1 irrespective of the number of segments ACK'ed
  - double **CongWin** every RTT
  - initial rate is slow but ramps up exponentially fast
- TCP slows down the increase of CongWin when *CongWin > ssthresh*



TCP Congestion Avoidance Phase

- when **`cwnd > ssthresh`** grow **`cwnd`** linearly
  - increase **`cwnd`** by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: **`cwnd = cwnd + MSS/cwnd`** for each ACK received

AIMD

- ACKs: increase **`cwnd`** by 1 MSS per RTT: additive increase
- loss: cut **`cwnd`** in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

Congestion Avoidance

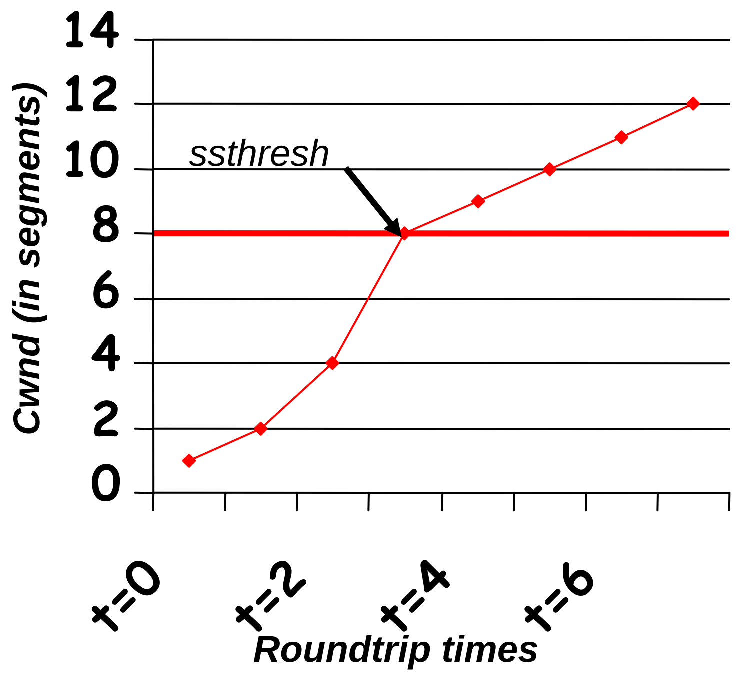
- Congestion avoidance phase is started if CongWin has reached the slow-start threshold value

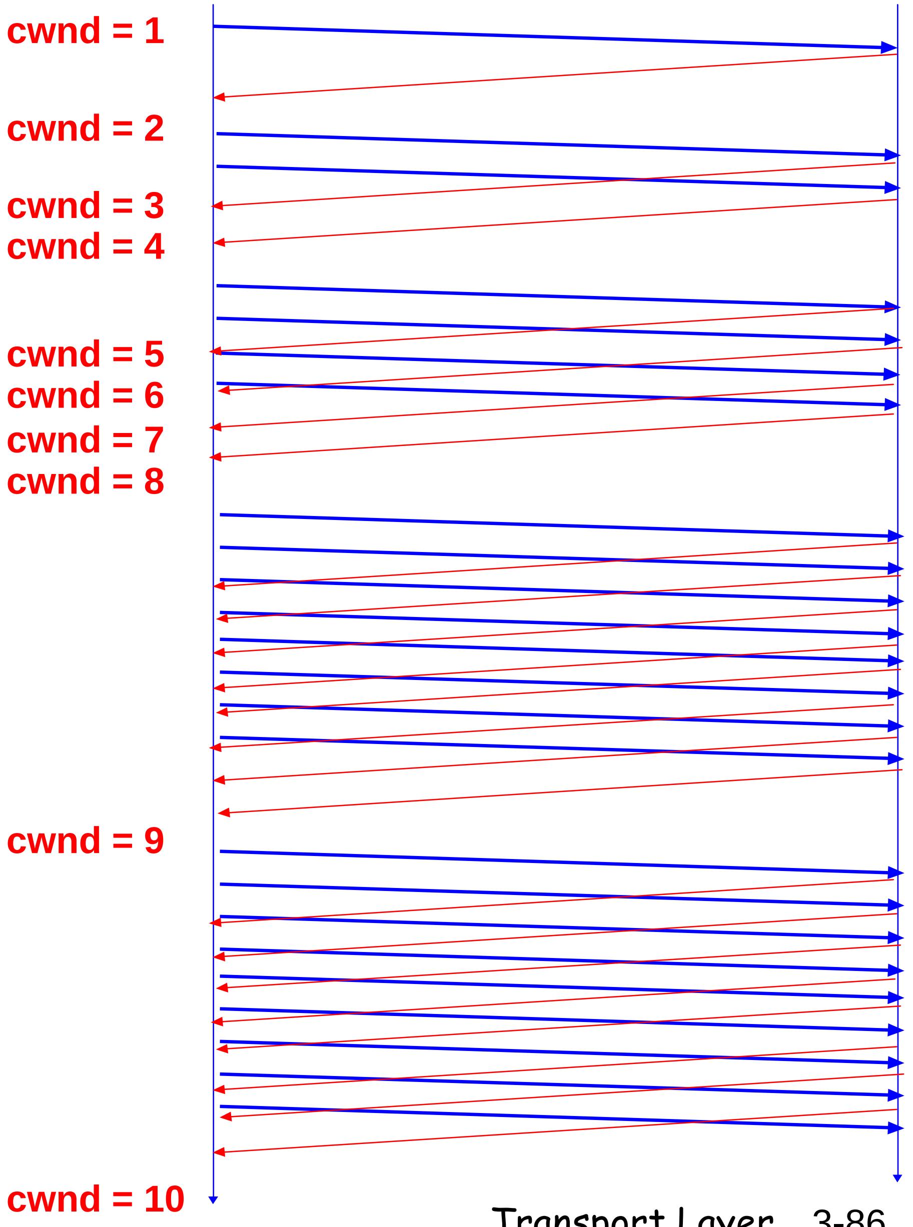
- If CongWin >= ssthresh then each time an ACK is received, increment CongWin as follows:
  - CongWin = CongWin + 1/CongWin (CongWin in segments)
  - In actual TCP implementation CongWin is in Bytes

    CongWin = CongWin + MSS * (MSS/CongWin)
- So CongWin is increased by one only if all CongWin segments have been acknowledged.

Example Slow Start/ Congestion Avoidance

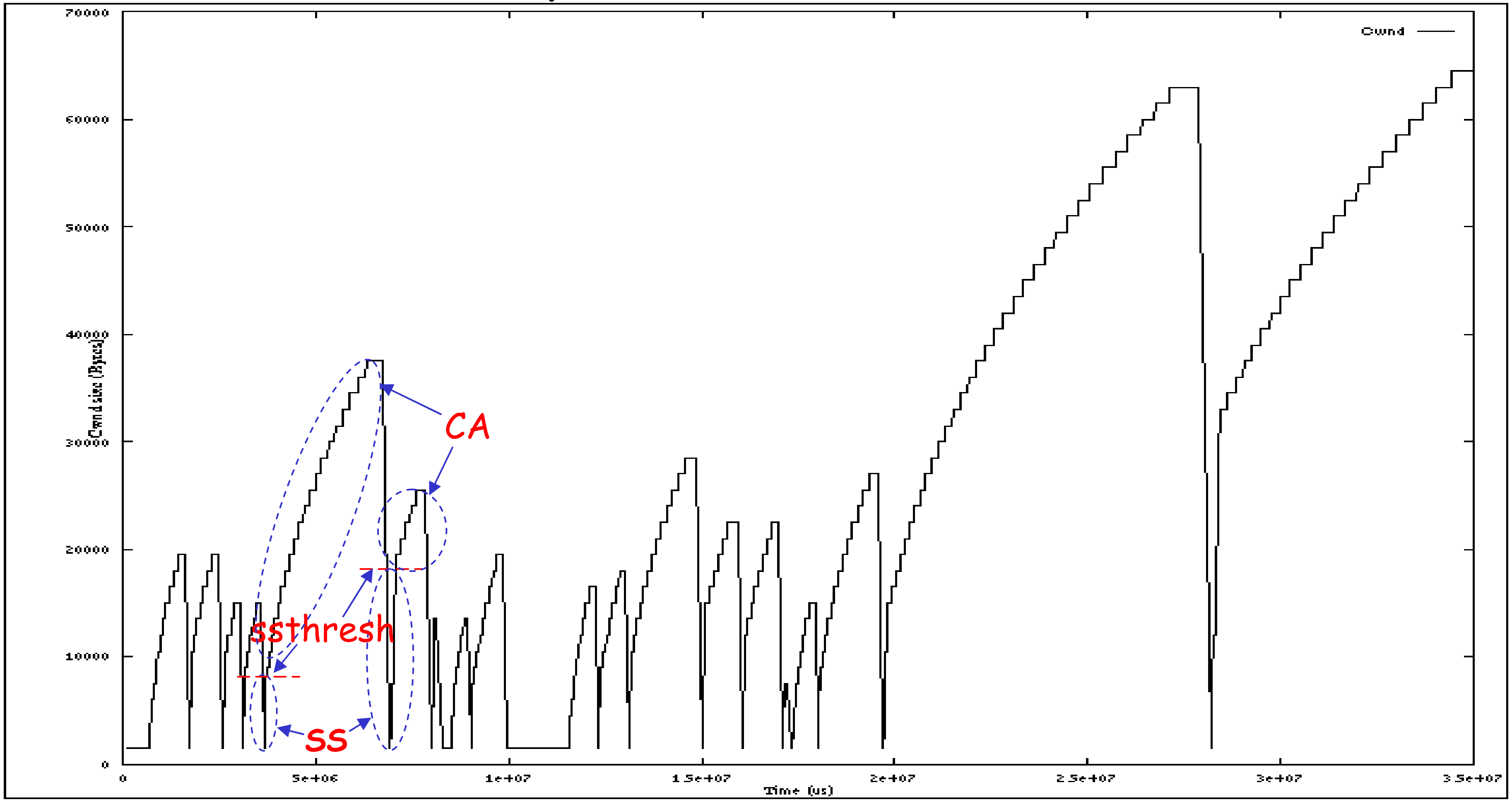
Assume that
ssthresh = 8





Slow Start / Congestion Avoidance

- A typical plot of CongWin for a TCP connection (MSS = 1500 bytes) with TCP Tahoe:



Responses to Congestion

- TCP assumes there is congestion if it detects a packet loss
- A TCP sender can detect lost packets via loss events:
    - Timeout of a retransmission timer
    - Receipt of 3 duplicate ACKs (fast retransmit)
- TCP interprets a Timeout as a binary congestion signal. When a timeout occurs, the sender performs:
    - ssthresh is set to half the current size of the congestion window:

      ssthresh = CongWin / 2
    - CongWin is reset to one:

      CongWin = 1
    - and slow-start is entered

Fast Recovery (differentiation btwn two loss events)

- After 3 dup ACKs (fast Retransmit):
  - ssthresh = CongWin/2
  - CongWin = CongWin/2
  - window then grows linearly
- But after timeout event:
  - CongWin = 1 MSS;
  - window then grows exponentially
  - to the threshold, then grows linearly

Philosophy:

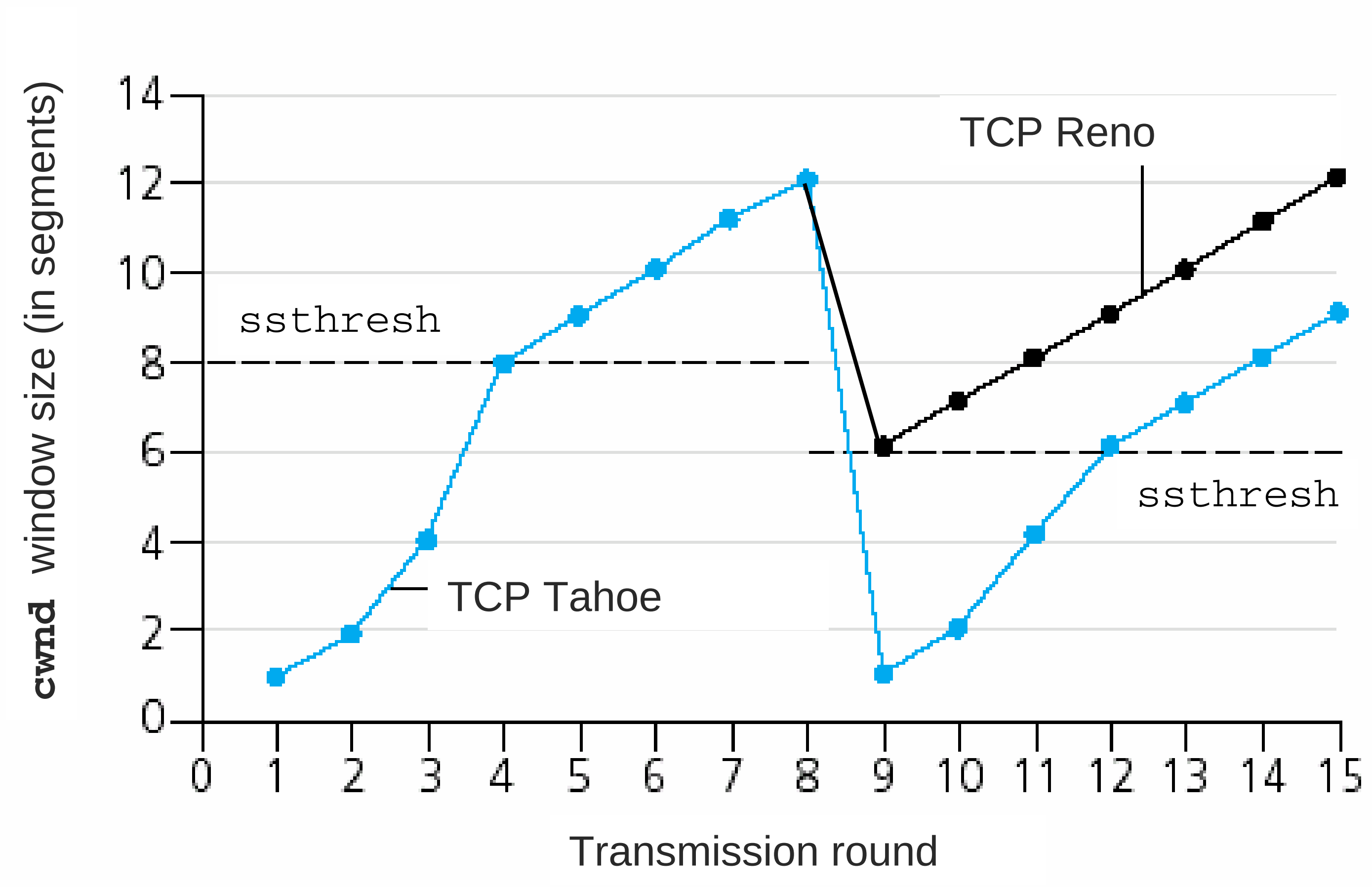
- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is "more alarming"

TCP Congestion Control

```
Initially:
    CongWin = 1;
    ssthresh = advertised window size;
New Ack received:
    if (CongWin < ssthresh) /* Slow Start*/
        CongWin = CongWin + 1;
    else /* Congestion Avoidance */
        CongWin = CongWin + 1/CongWin;
Timeout:
    ssthresh = CongWin/2;
    CongWin = 1;
Fast Retransmission:
    ssthresh = CongWin/2;
    CongWin = CongWin/2;
```

Slow Start (exponential increase phase) is continued until CongWin reaches half of the level where the loss event occurred last time. CongWin is increased slowly after (linear increase in Congestion Avoidance phase).

Popular “flavors” of TCP



Summary: TCP Congestion Control

- When CongWin is below Threshold, sender in slow-start phase, window grows exponentially.
- When CongWin is above Threshold, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.
- The actual sender window size is determined based on the congestion and flow control algorithms

SenderWin=min(RcvWin,CongWin)

TCP Congestion Control Summary

Event	State	TCP Sender Action	Commentary
ACK receipt for previously unacked data	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of CongWin every RTT
ACK receipt for previously unacked data	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
Timeout	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP throughput

- *Q:* what's average throughout of TCP as function of window size, RTT?
  - ignoring slow start
- let W be window size when loss occurs.
  - when window is W, throughput is W/RTT
  - just after loss, window drops to W/2, throughput to W/2RTT.
  - average throughout: .75 W/RTT

TCP Futures: TCP over "long, fat pipes"

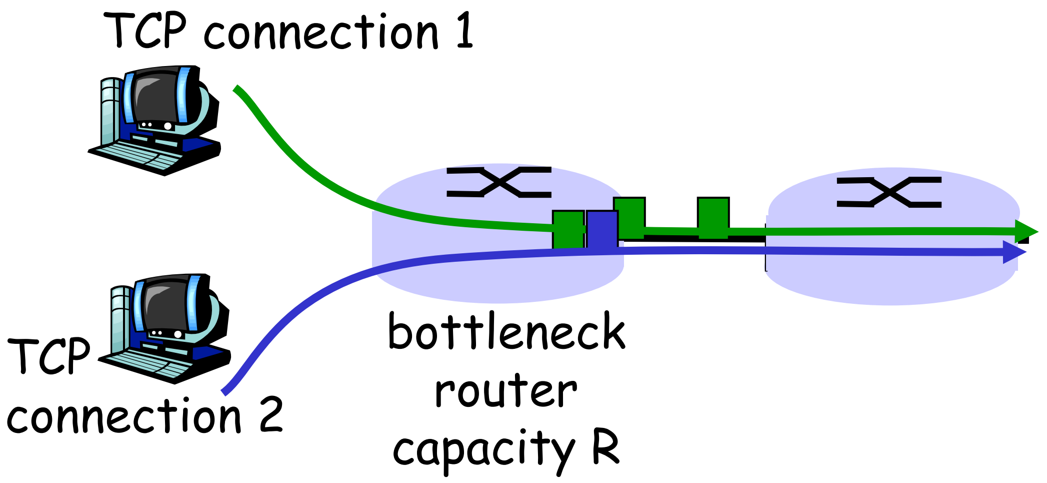
- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires window size W = 83,333 in-flight segments
- throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- $L = 2 \cdot 10^{-10}$ *Wow*
- new versions of TCP for high-speed

TCP Fairness

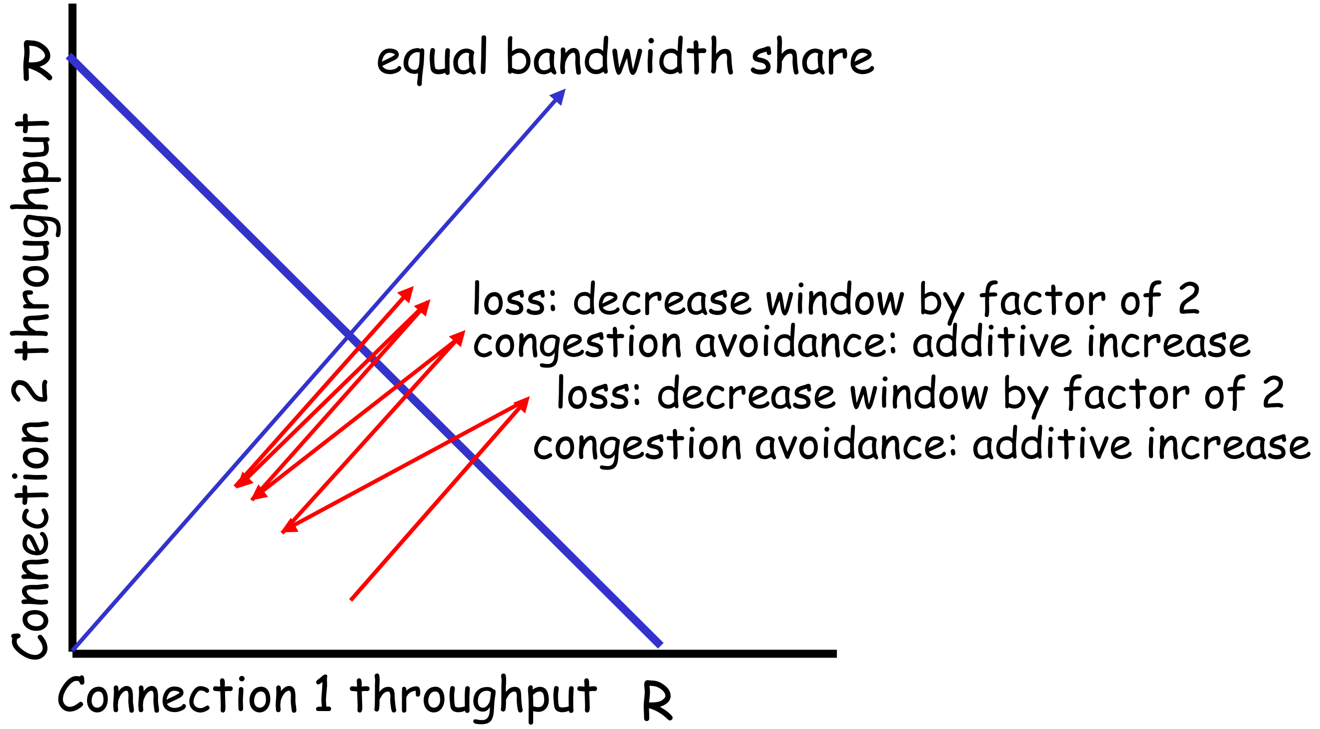
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

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


R
equal bandwidth share
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 2 throughput
Connection 1 throughput
R

Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel cnctions between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 cnctions;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !

TCP Connection Management

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

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *client:* connection initiator

```
Socket clientSocket = new
 Socket("hostname","port

 number");
```

- *server:* contacted by client

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

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

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

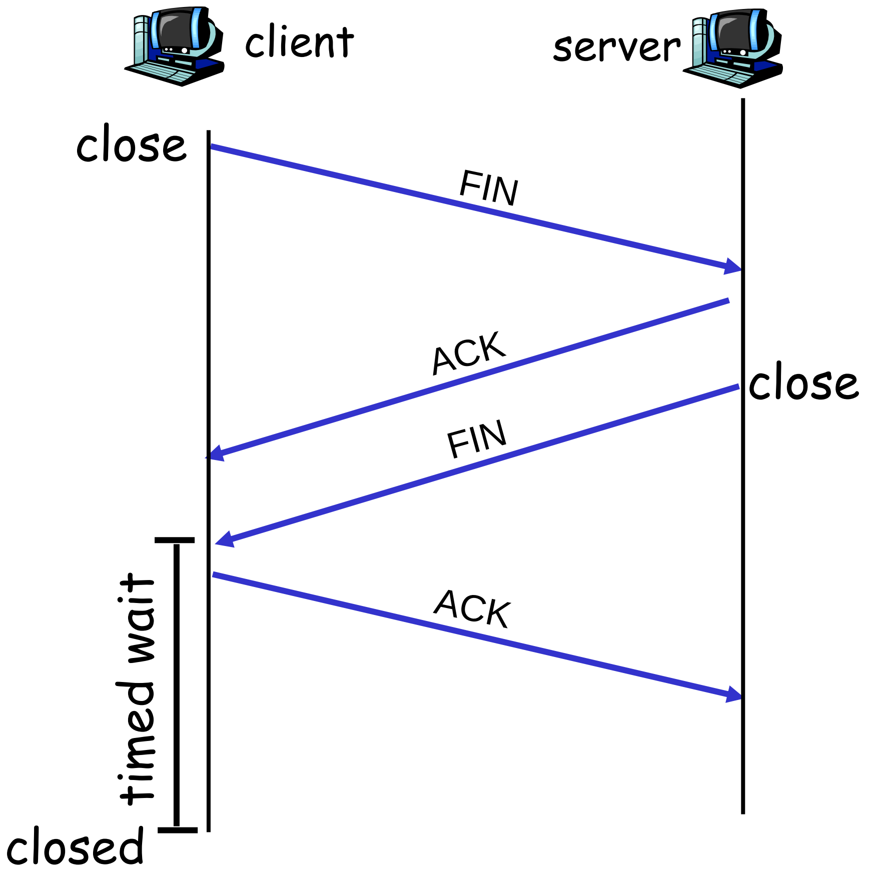
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`clientSocket.close();`

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

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

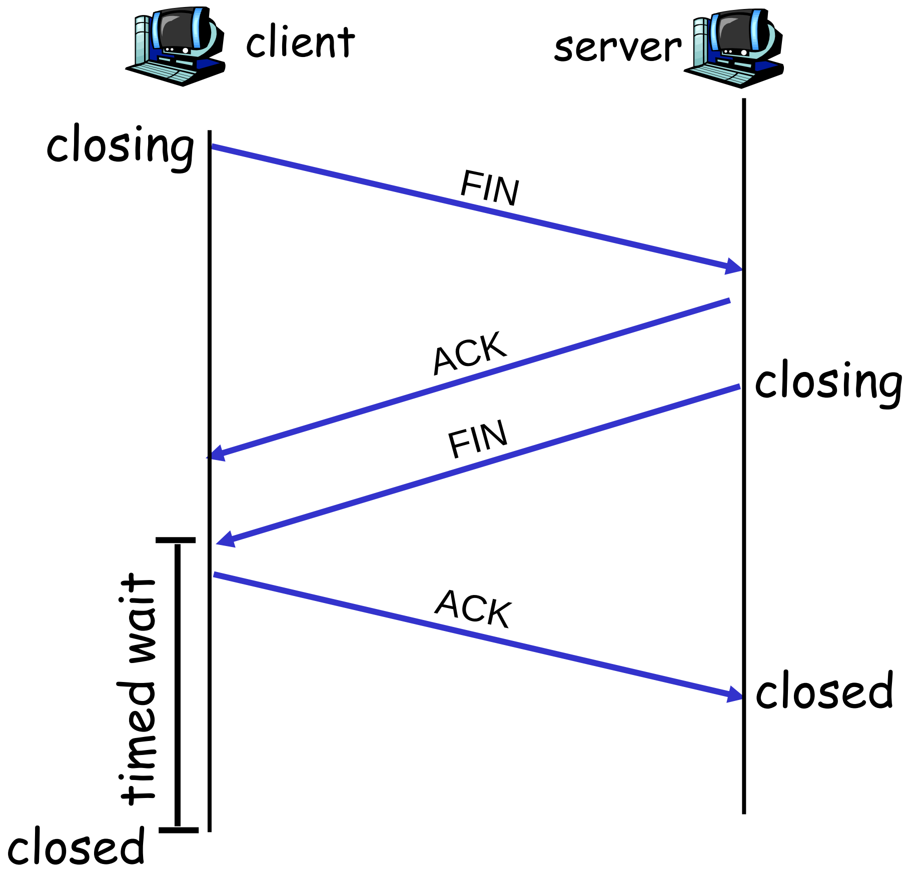


TCP Connection Management (cont.)

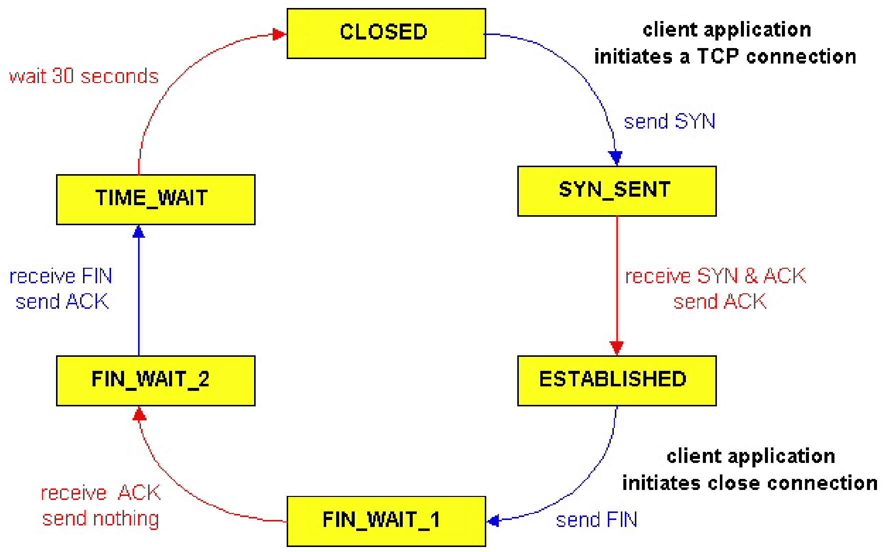
Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

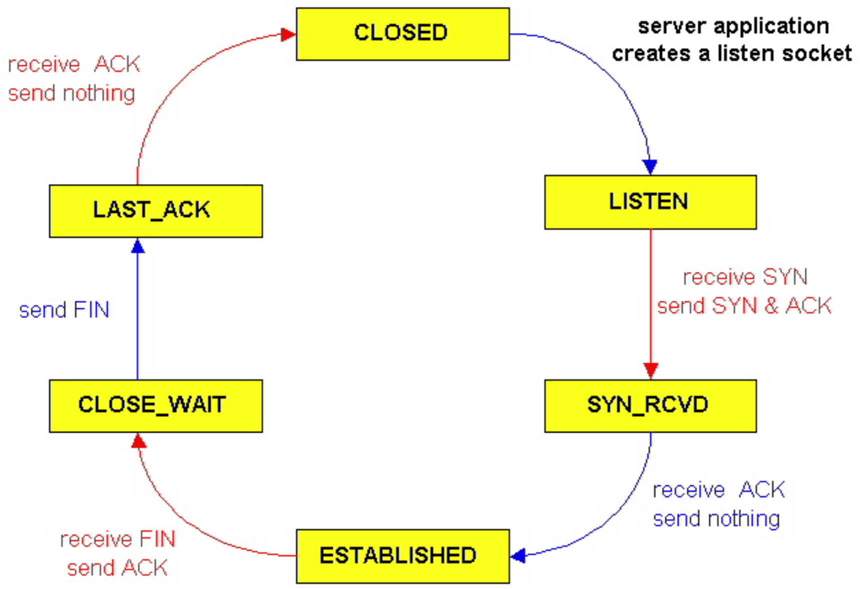


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



Tuning TCP/IP Parameters

- TCP/IP parameters
  - A set of default values may not be optimal for all applications.
  - The network administrator may wish to turn on or off some TCP/IP functions for performance or security considerations.
- Many Unix and Linux systems provide some flexibility in tuning the TCP/IP kernel.
- /sbin/sysctl is used to configure the Linux kernel parameters at runtime.
  - Default kernel configuration file is /sbin/sysctl.conf.
  - Frequently used sysctl options:
    - sysctl –a or sysctl –A: list all current values.
    - sysctl –p *file_name*: load the sysctl setting from a configuration file.
    - sysctl –w *variable=value*: change the value of the parameter

SomeTCP Parameters in Linux Kernel

- **tcp_syn_retries**
  - Number of SYN packets the kernel will send before giving up on the new connection.
- **tcp_synack_retries**
  - number of SYN+ACK packets sent before the kernel gives up on the connection.
- **tcp_window_scaling**
  - Maximum window size of 65535 bytes not enough for for really fast networks. The window scaling options allows for almost gigabyte windows, which is good for connections with large delay-bandwidth product.
- **tcp_max_syn_backlog**
  - Maximal number of remembered connection requests, which still did not receive an acknowledgment from connecting client.
- **tcp_fin_timeout**
  - How many seconds to wait for a final FIN packet before the socket is closed; required to prevent denial-of-service (DoS) attacks. Default value is 60 seconds.

SomeTCP Parameters in Linux Kernel

- **tcp_rmem**
  - This is a vector of 3 integers: [min, default, max]. These parameters are used by TCP to dynamically adjust receive buffer sizes.
  - min - minimum size of the receive buffer used by each TCP socket. The default value is 4K.
  - default - the default size of the receive buffer for a TCP socket. The default value is 87380 bytes, and is lowered to 43689 in low memory systems. If larger receive buffer sizes are desired, this value should be increased.
  - max - the maximum size of the receive buffer used by each TCP socket. The default value of 87380*2 bytes is lowered to 87380 in low memory systems.
- **tcp_smem**
  - Send buffer parameters [min, default, max] similar to tcp_rmem.