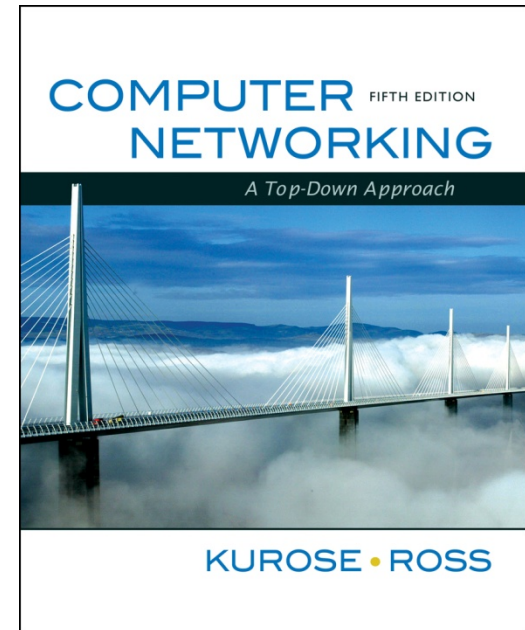


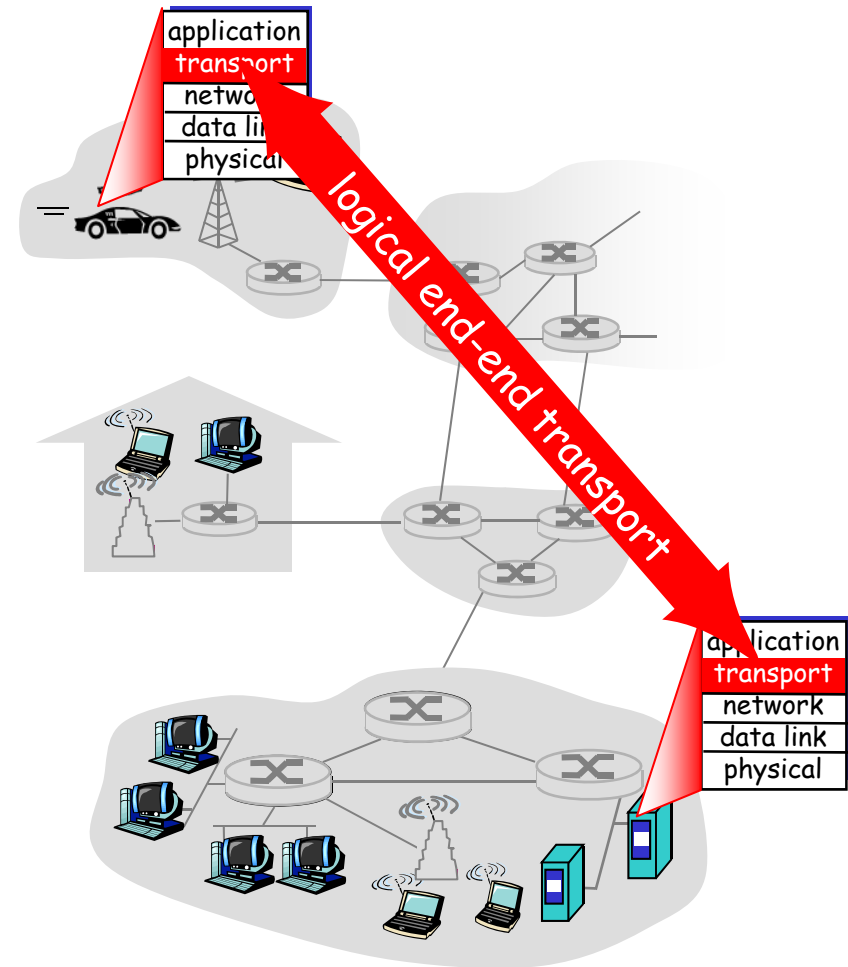
Chapter 3

Transport Layer



Transport services and protocols

- ❑ provide *logical communication* between application processes running on different hosts
- ❑ transport protocols run in end systems
 - send side: breaks application messages into *segments*, passes to network layer
 - Receiving side: reassembles segments into messages, passes to application layer
- ❑ more than one transport protocol available to applications
 - 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

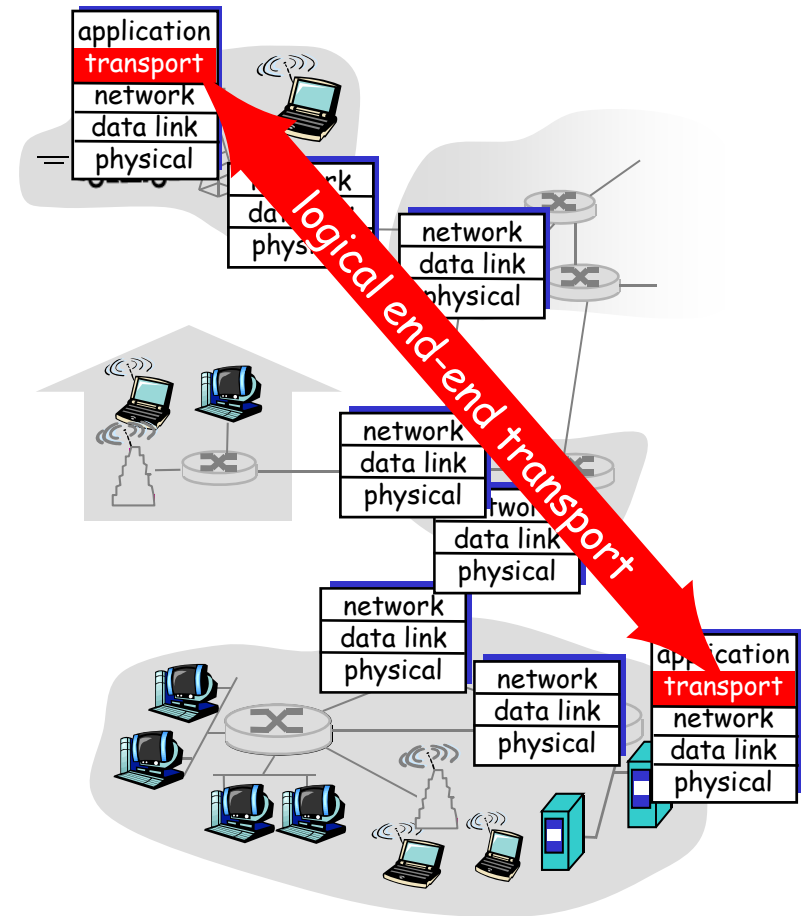
Household analogy:

12 kids sending letters to 12 kids

- ❑ processes = kids
- ❑ app messages = letters in envelopes
- ❑ hosts = houses
- ❑ transport protocol = Ann and Bill
- ❑ 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

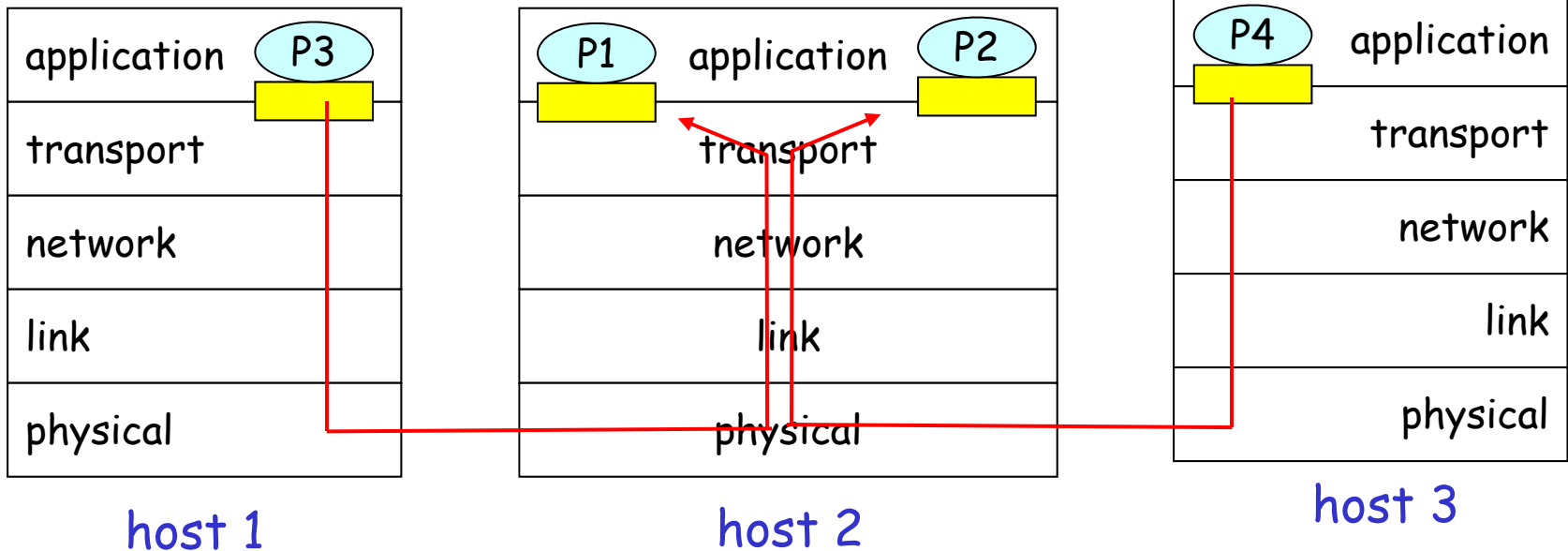
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

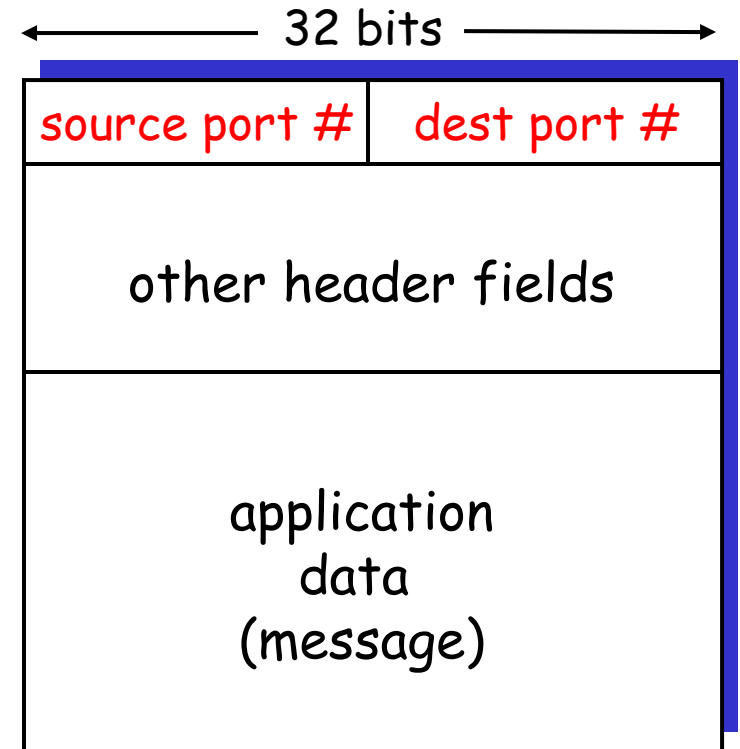
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

■ = 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
- **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( );
```

```
DatagramSocket mySocket2 = new  
    DatagramSocket( );
```

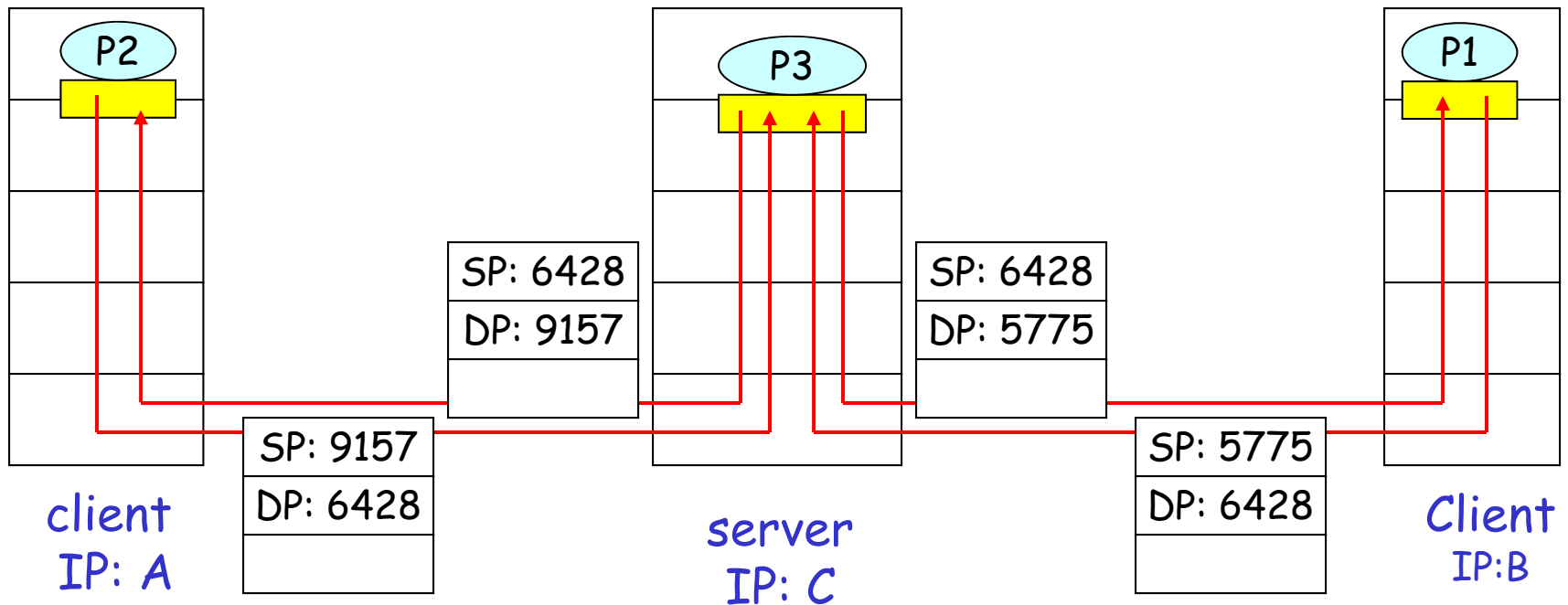
- ❑ UDP socket identified by two-tuple:

(dest IP address, dest port number)

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

Connectionless demultiplexing

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

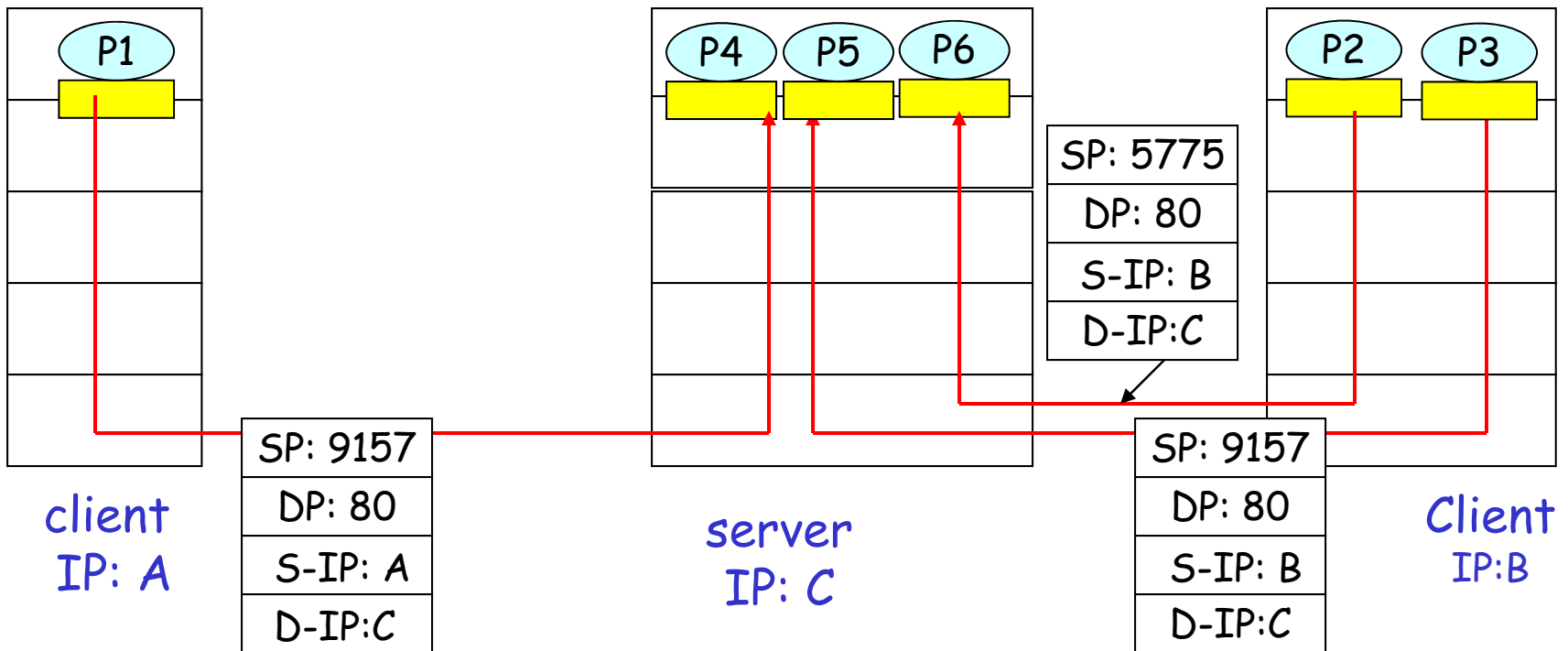


SP provides "return address"

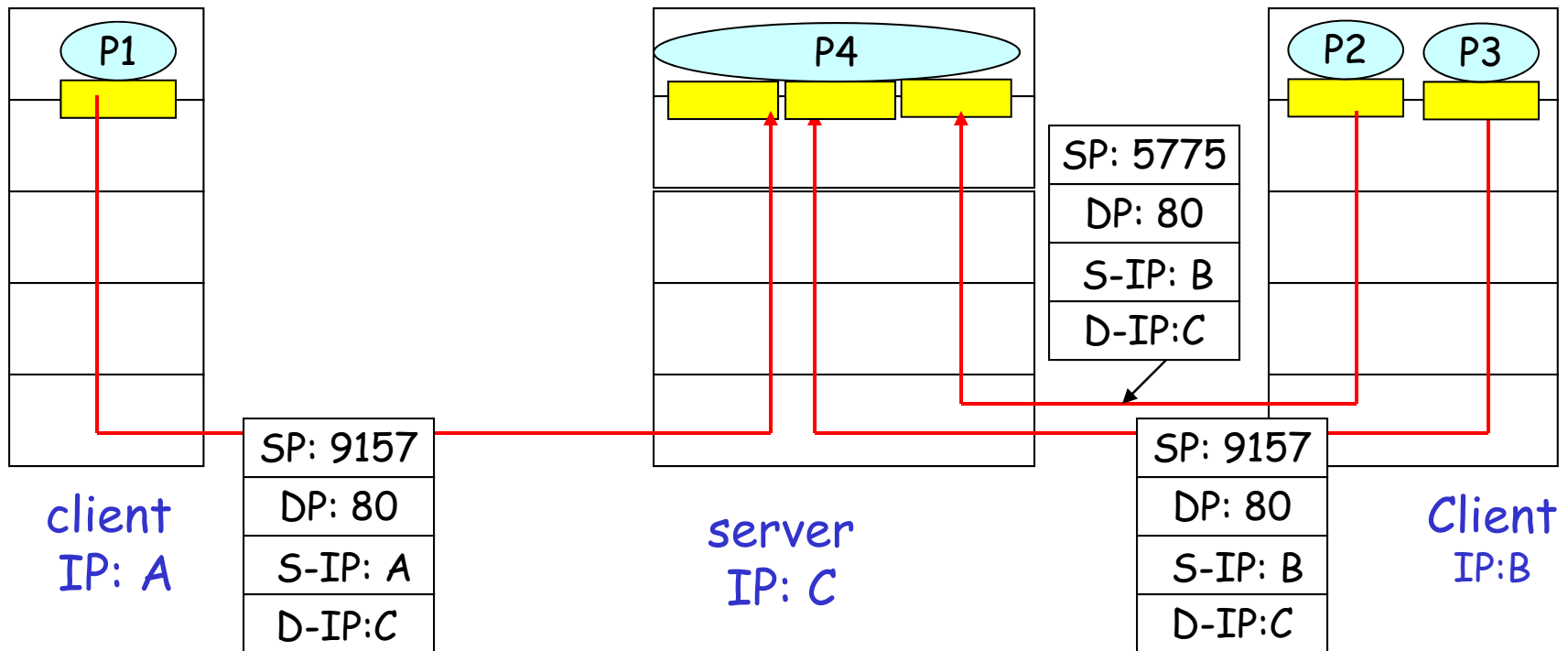
Connection-oriented demultiplexing

- ❑ 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 demultiplexing



Connection-oriented demultiplexing: Threaded Web Server



UDP: User Datagram Protocol [RFC 768]

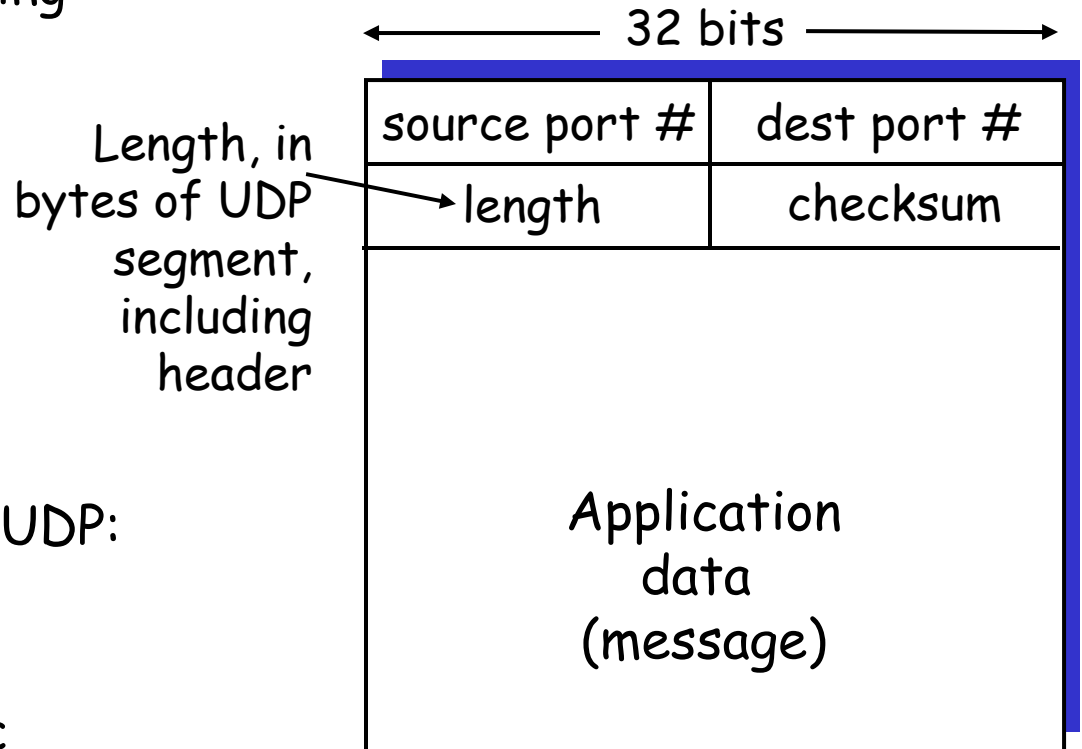
- ❑ “no frills,” “bare bones” Internet transport protocol
- ❑ “best effort” service, UDP segments may be:
 - lost
 - delivered out of order to app
- ❑ *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

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!



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
- ❑ sender puts checksum value into UDP checksum field

Receiver:

- ❑ compute checksum of received segment
 - ❑ check if computed checksum equals checksum field value:
 - NO - error detected
 - YES - no error detected.
But maybe errors nonetheless? More later
-

Internet Checksum Example

□ Note

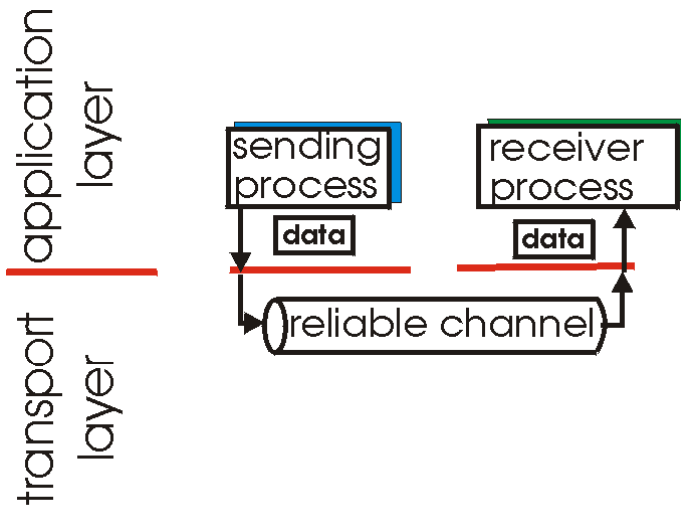
- When adding numbers, a carryout from the most significant bit needs to be added to the result

□ 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	
<hr/>																	
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
<hr/>																	
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	

Principles of Reliable data transfer

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

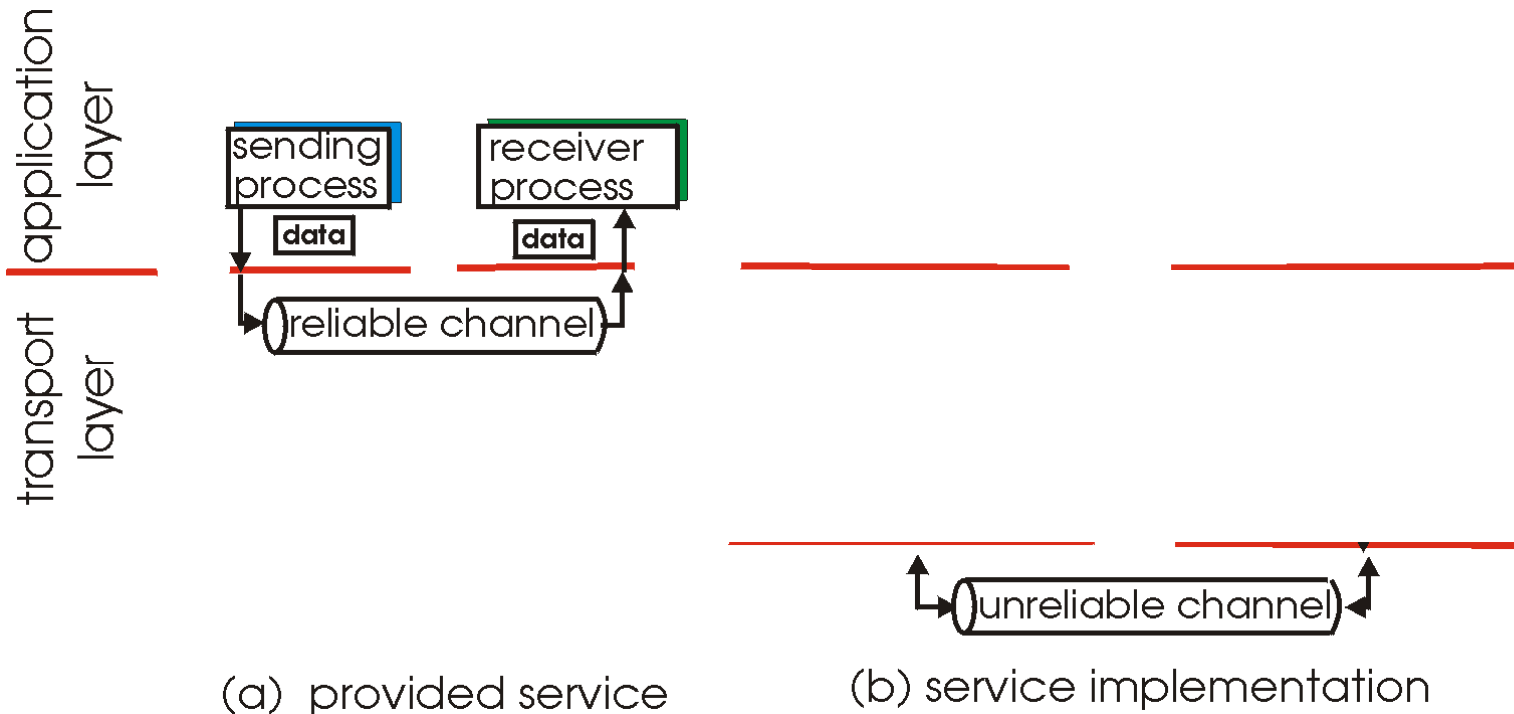


(a) provided service

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

Principles of Reliable data transfer

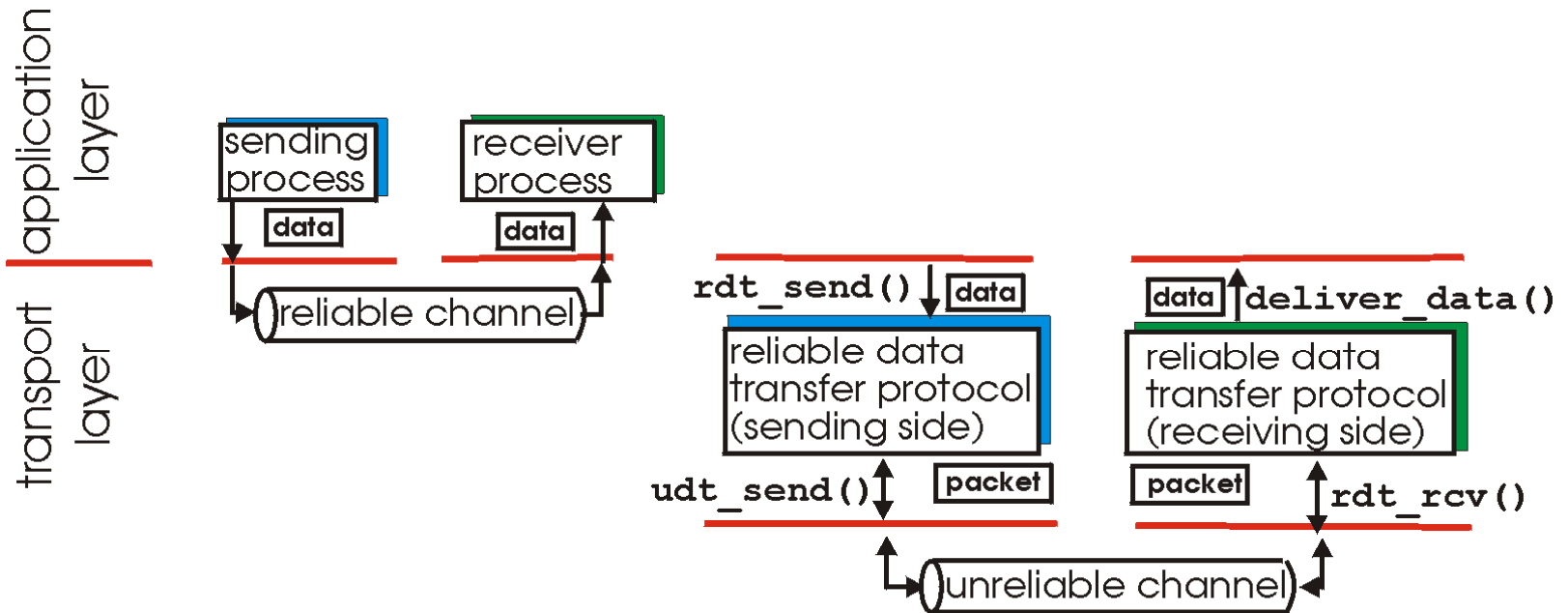
- important in app., transport, link layers
- top-10 list of important networking topics!



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

Principles of Reliable data transfer

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



(a) provided service

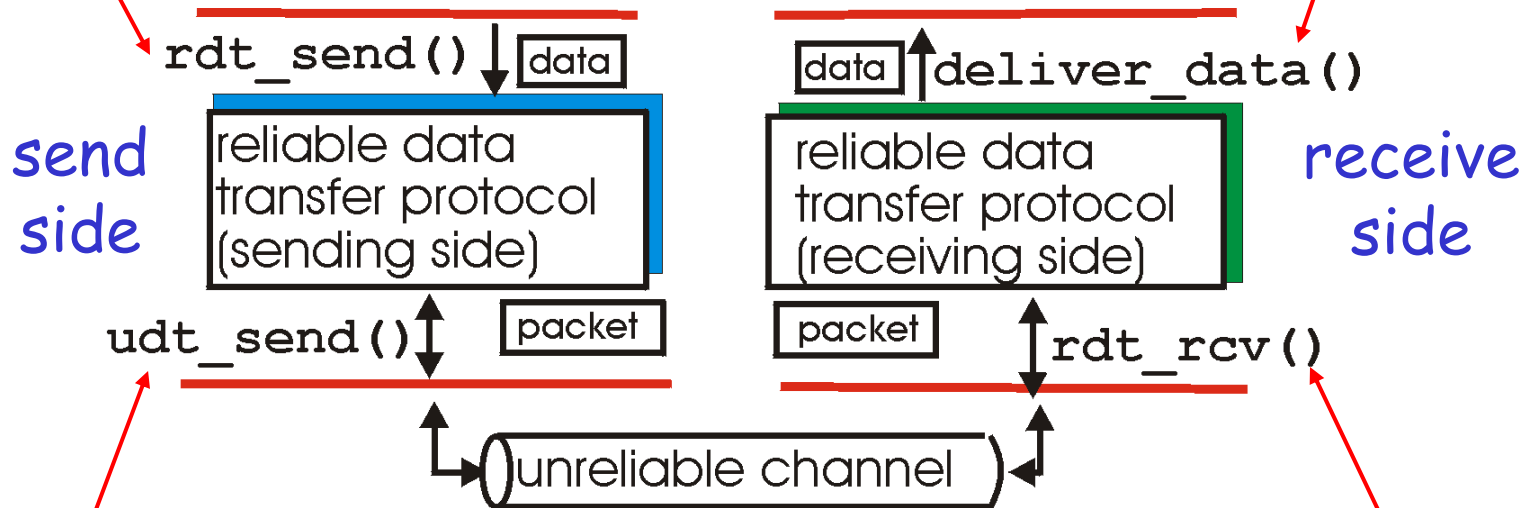
(b) service implementation

- 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



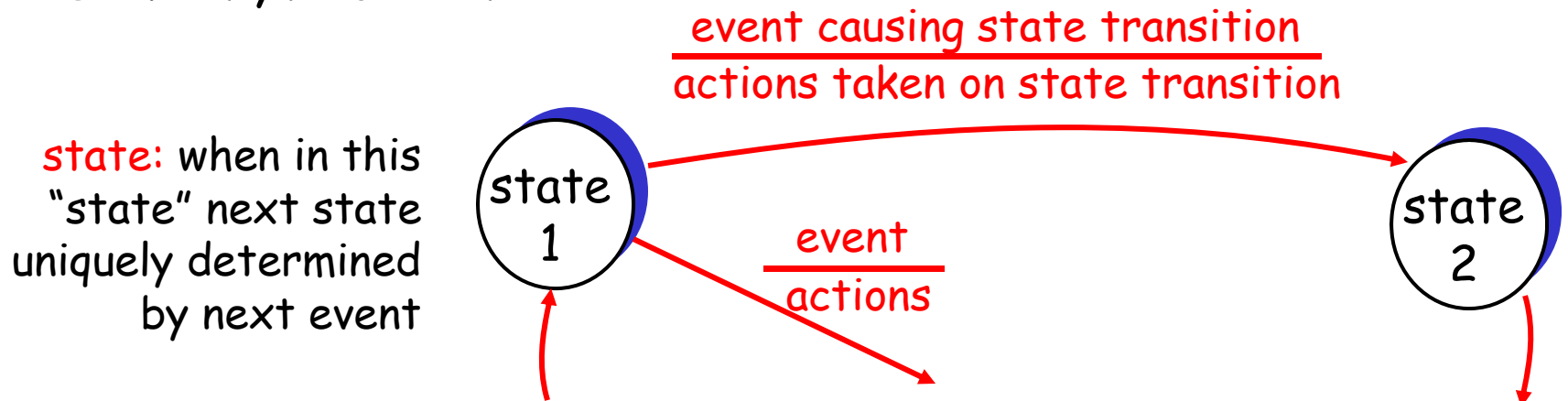
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

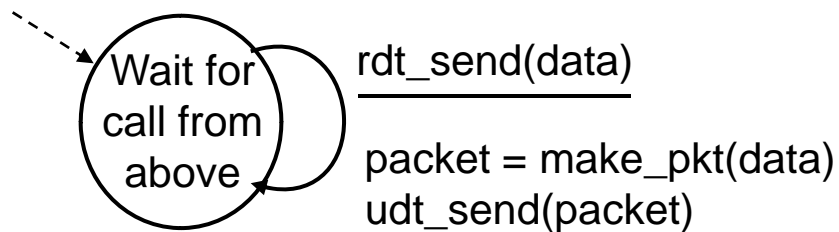
We'll:

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

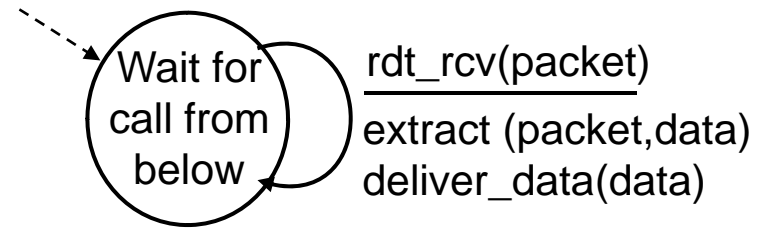


Rdt1.0: reliable transfer over a reliable channel

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



sender



receiver

Rdt2.0: channel with bit errors

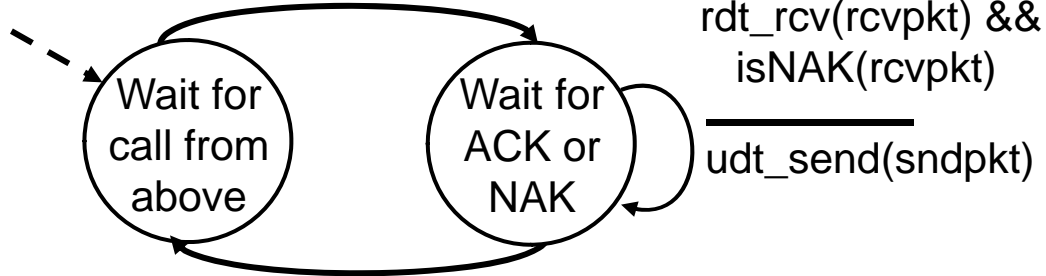
- underlying channel may flip bits in packet
 - checksum to detect bit errors
- *the question: how to recover from errors:*
 - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
 - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
 - 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)

snpkt = make_pkt(data, checksum)

udt_send(sndpkt)



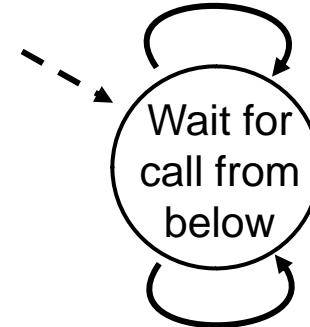
rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

sender

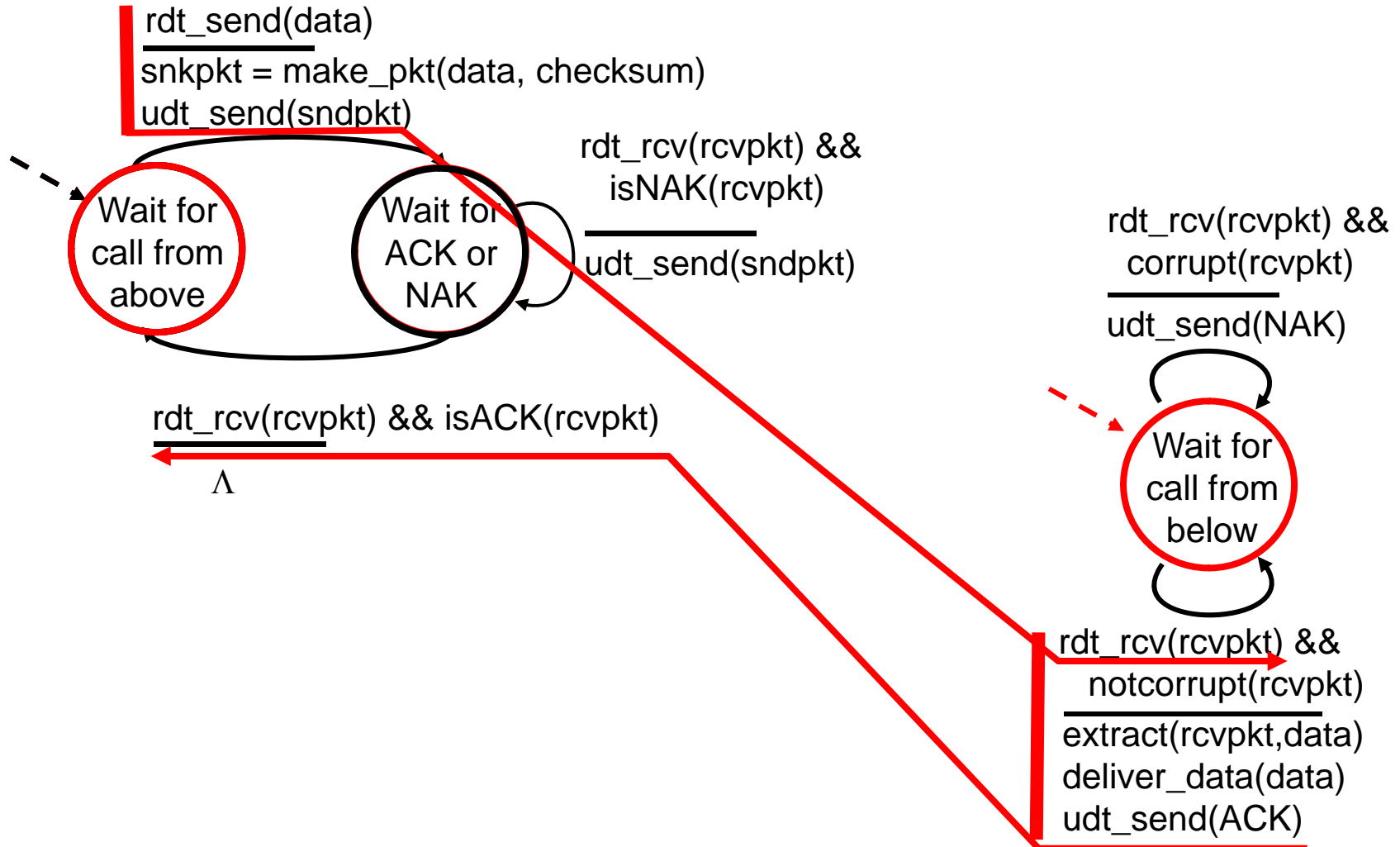
receiver

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)

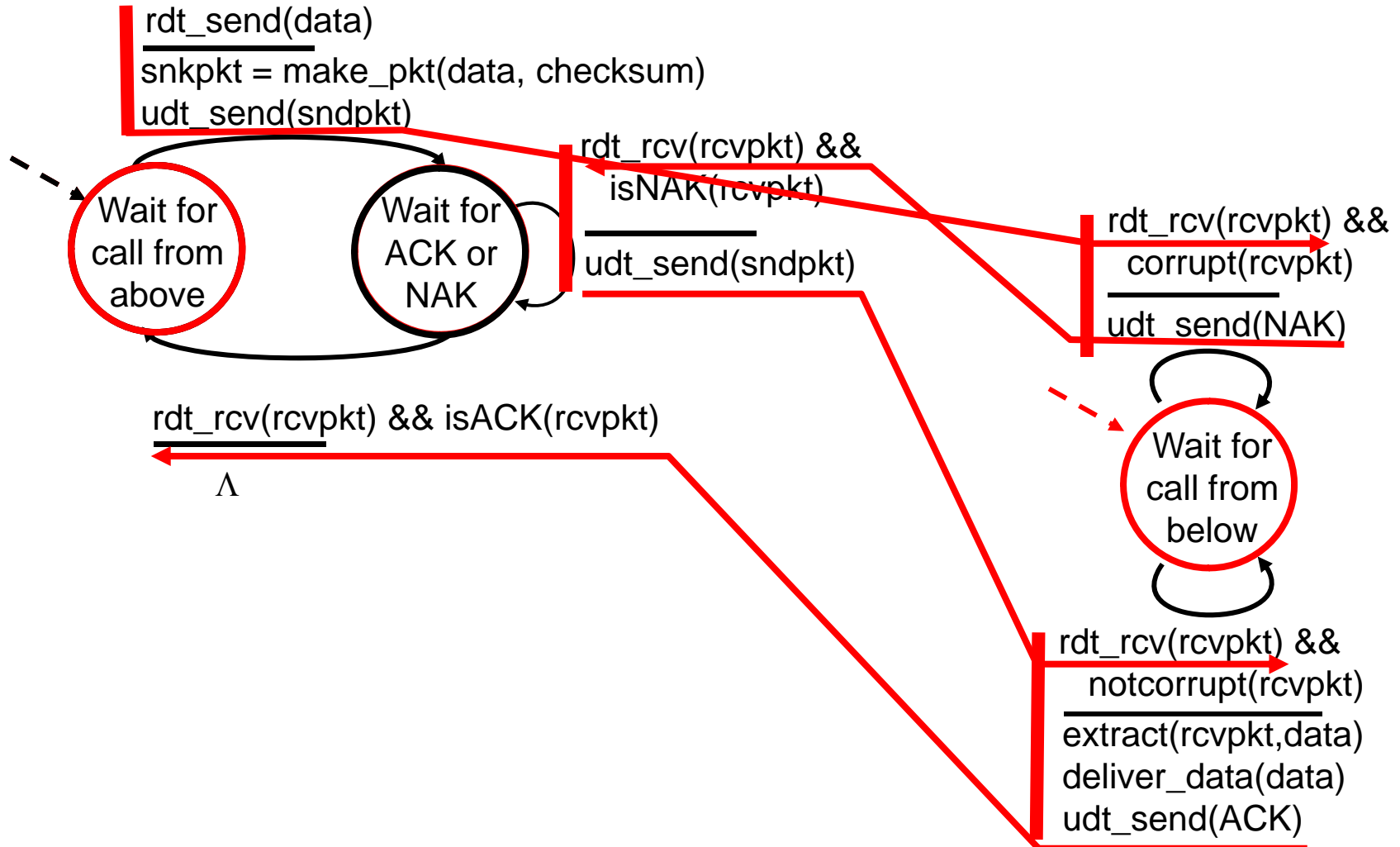


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

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

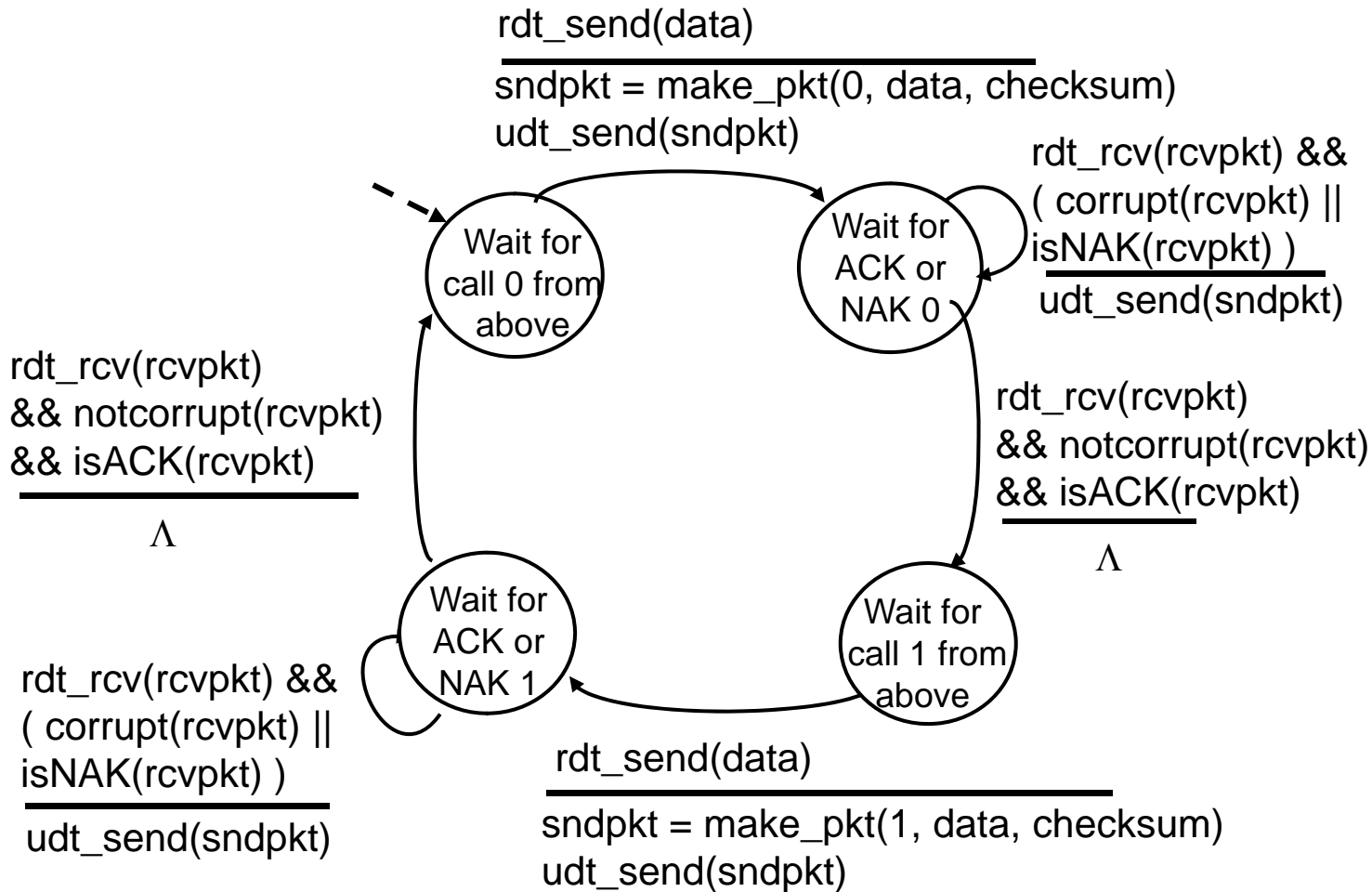
Handling duplicates:

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

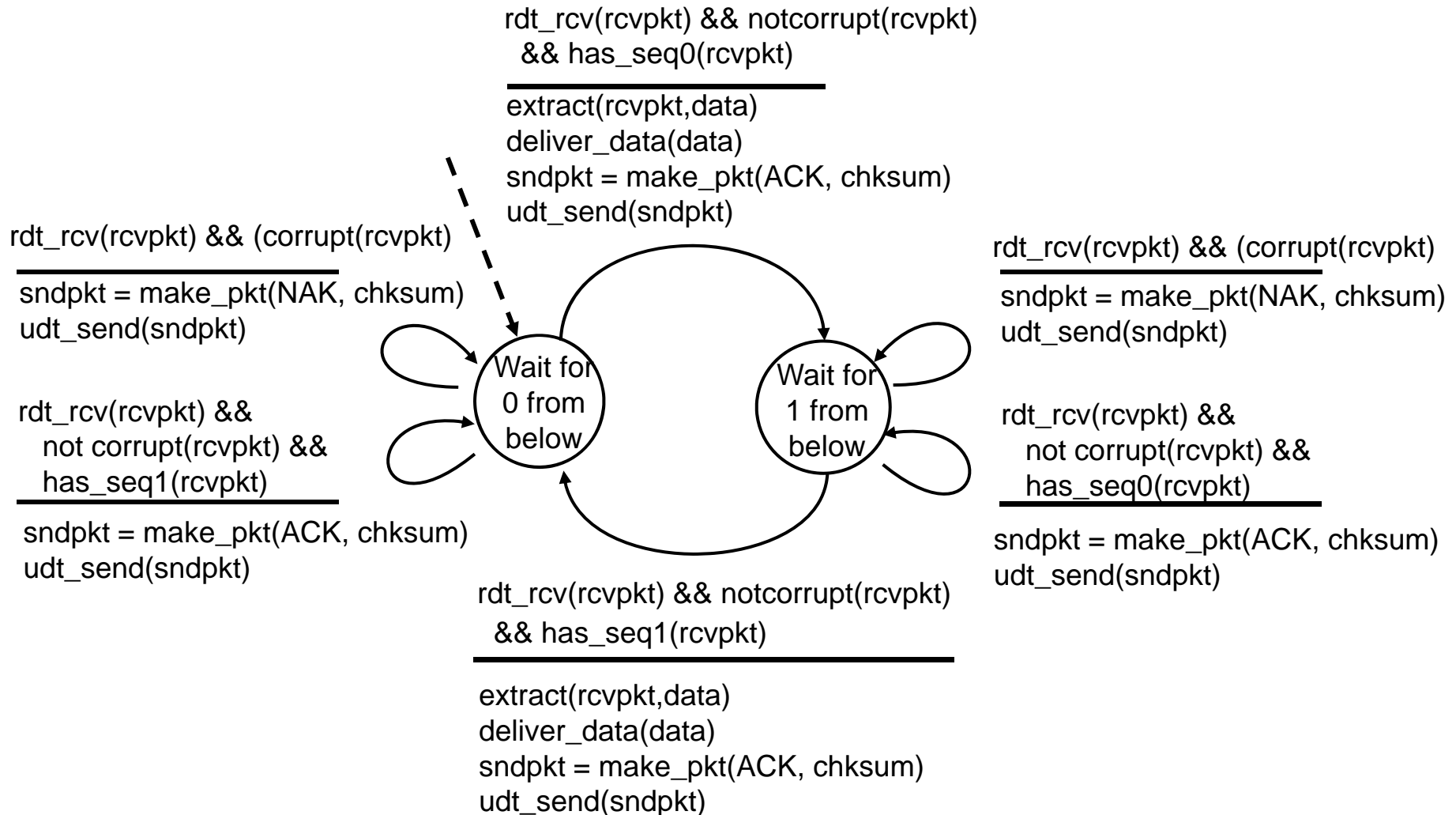
stop and wait

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

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. #

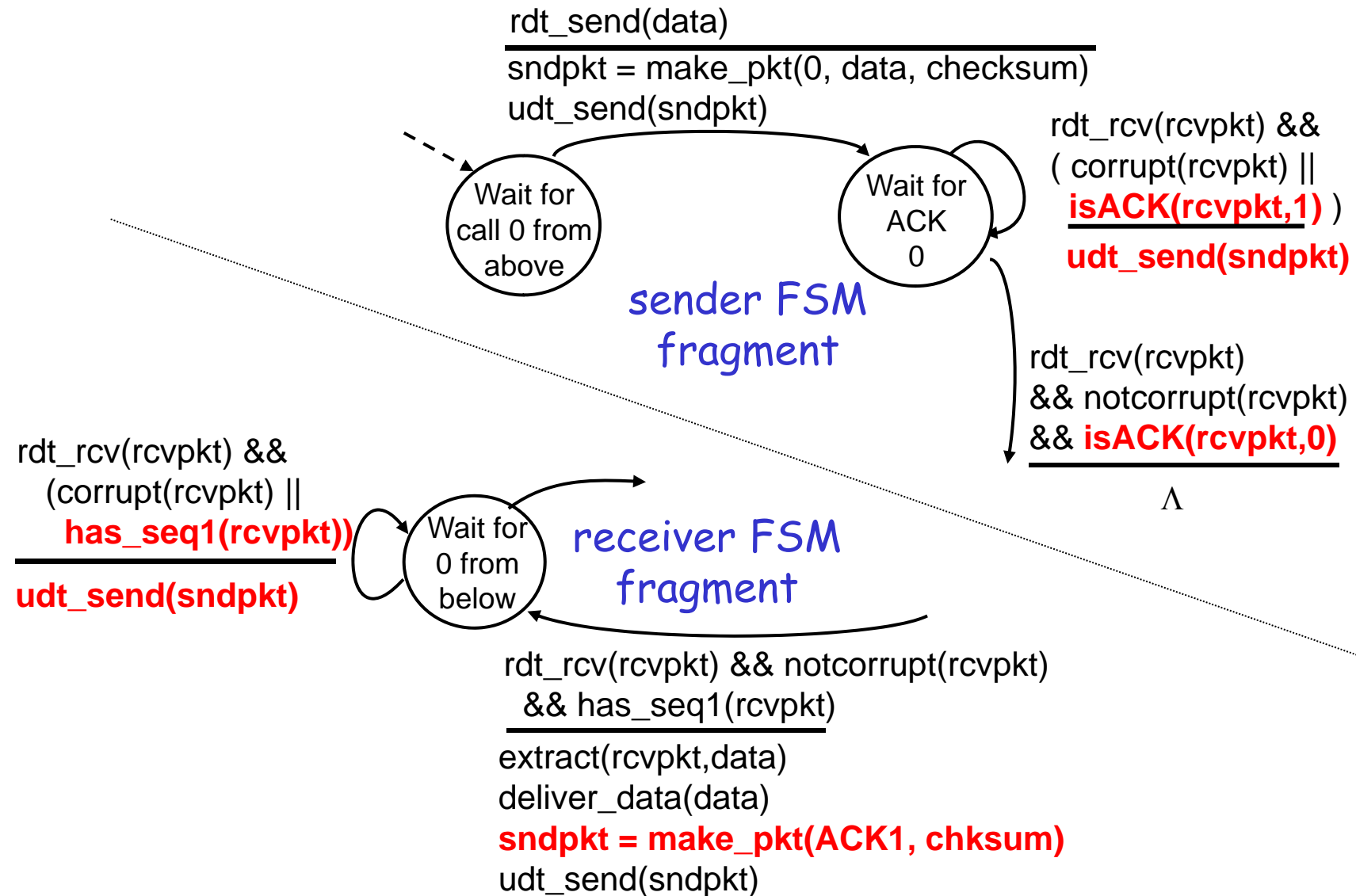
Receiver:

- ❑ 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



rdt3.0: channels with errors *and* loss

New assumption:

underlying channel can also lose packets (data or ACKs)

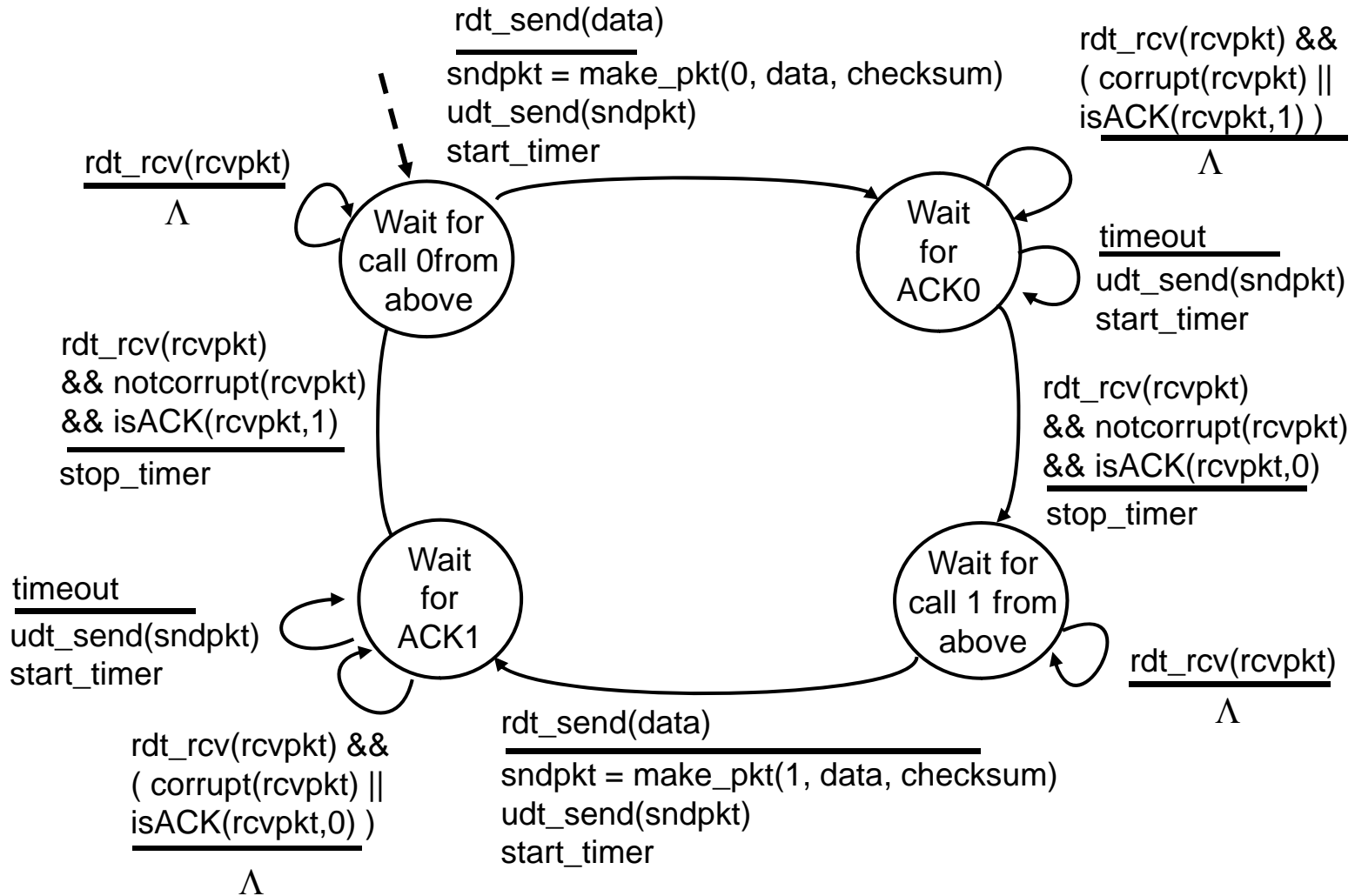
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits

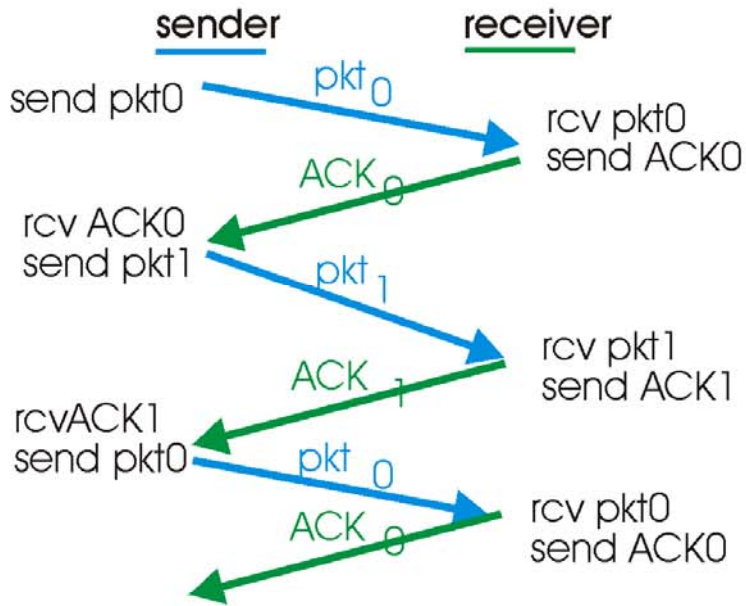
“reasonable” amount of time for ACK

- 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

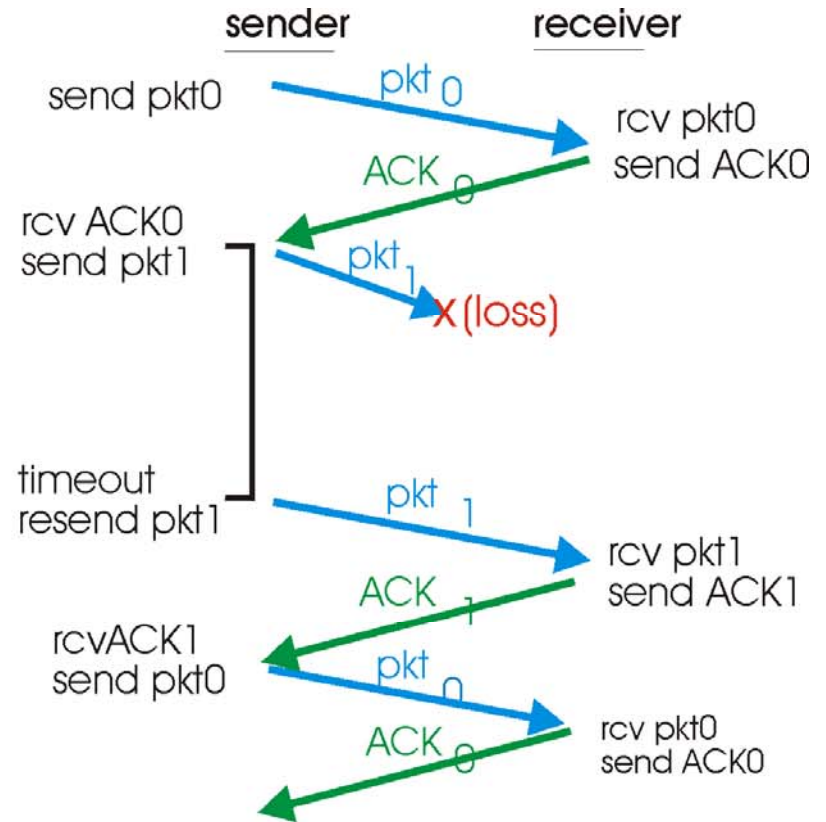
rdt3.0 sender



rdt3.0 in action

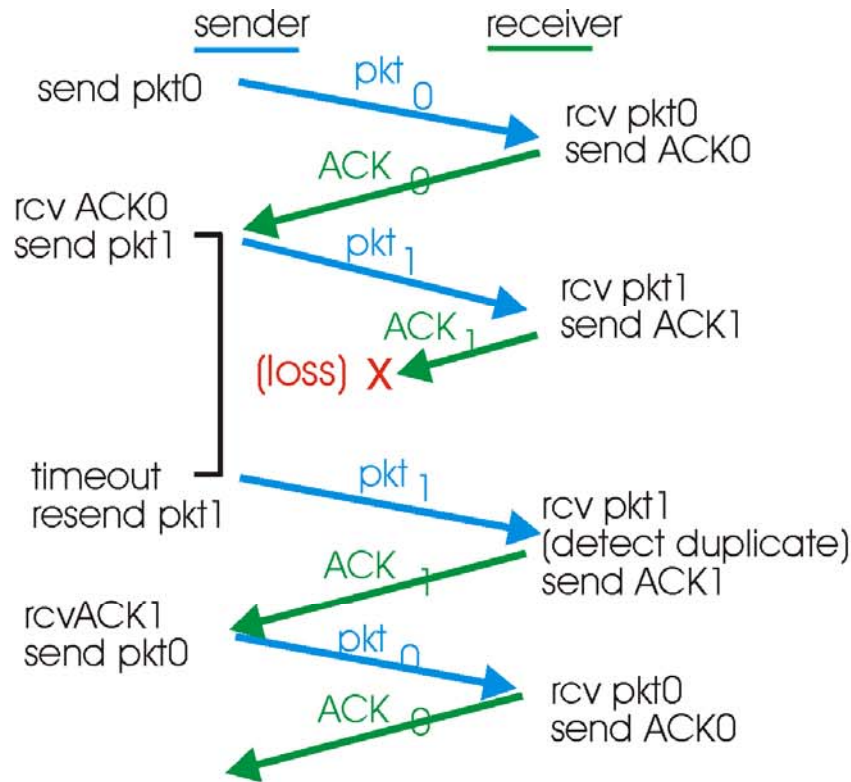


(a) operation with no loss

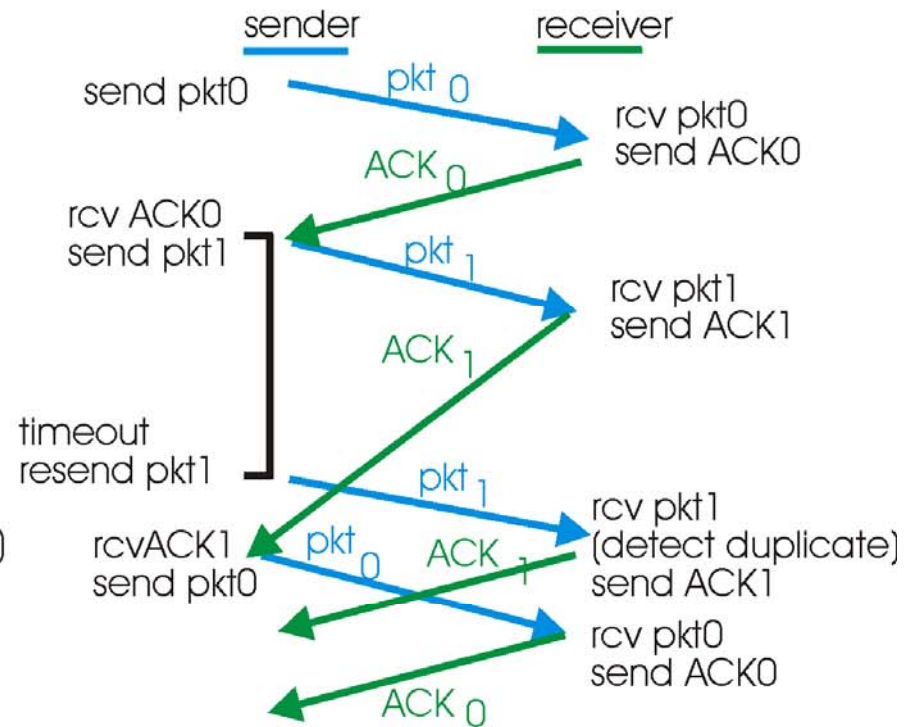


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

- ❑ rdt3.0 works, but performance stinks
- ❑ ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

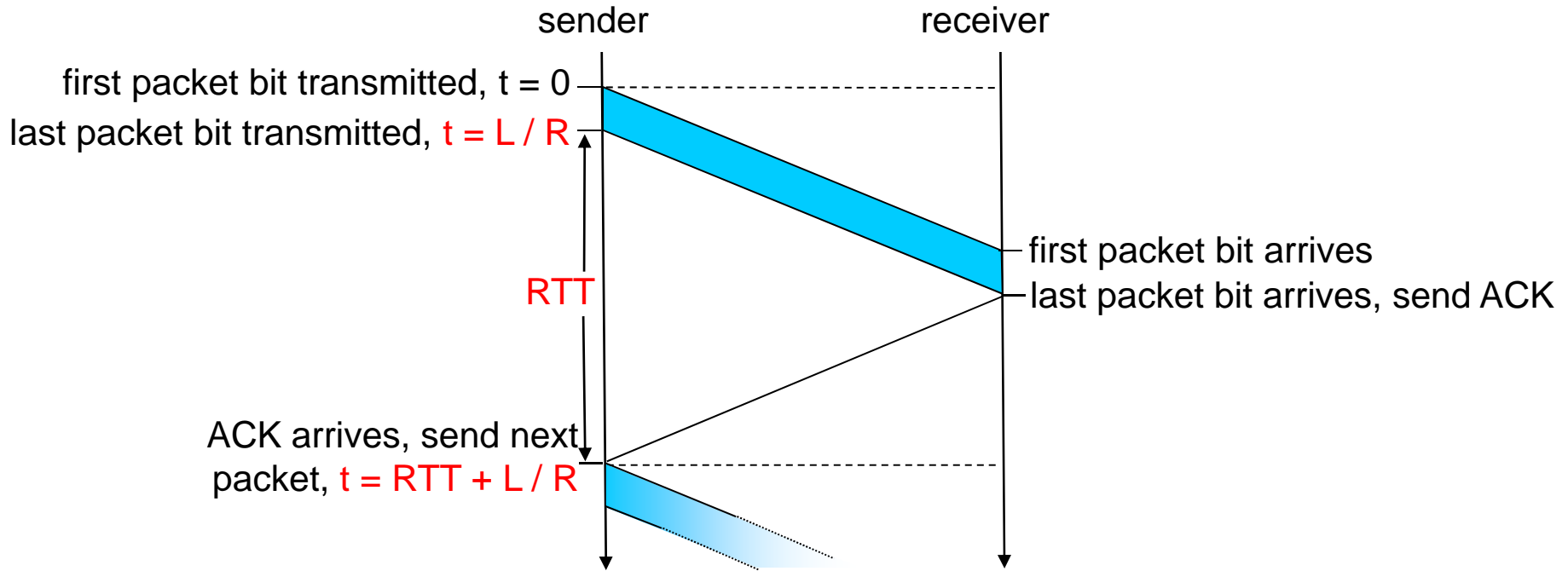
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

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

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

- 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

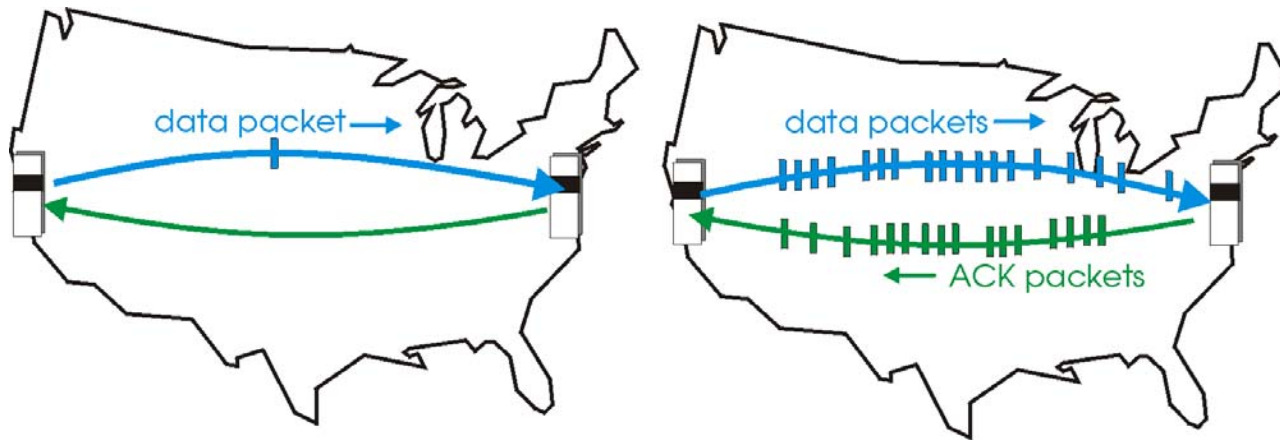


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

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

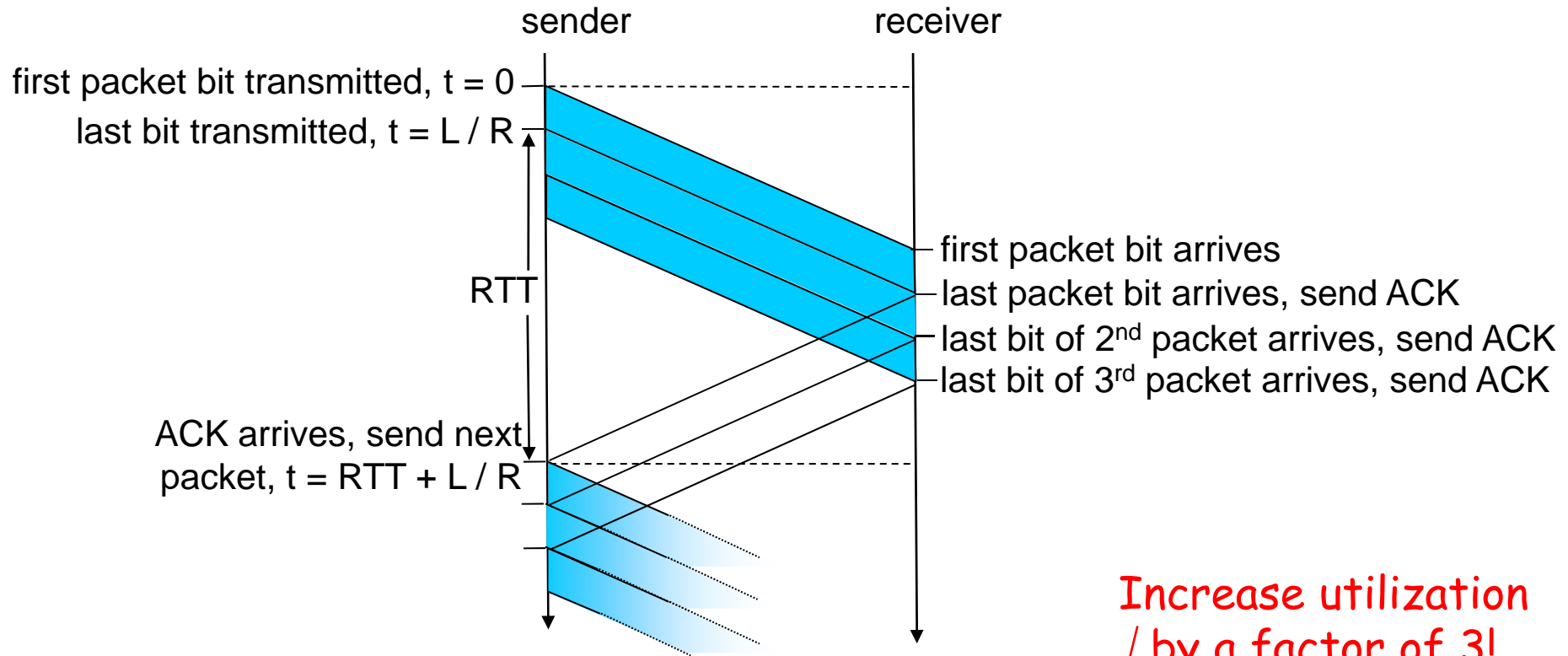


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization



Increase utilization
/ by a factor of 3!

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

Pipelining Protocols

Go-back-N: big picture:

- ❑ Sender can have up to N unacked packets in pipeline
- ❑ Rcvr only sends cumulative acks
 - Doesn't ack packet if there's a gap
- ❑ Sender has timer for oldest unacked packet
 - If timer expires, retransmit all unacked packets

Selective Repeat: big pic

- ❑ Sender can have up to N unacked packets in pipeline
- ❑ Rcvr acks individual packets
- ❑ Sender maintains timer for each unacked packet
 - When timer expires, retransmit only unack packet

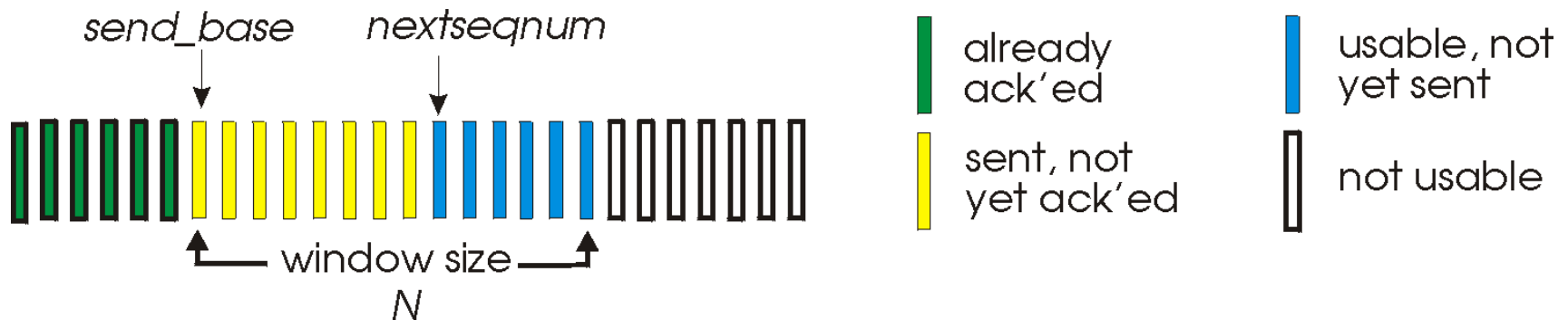
Selective repeat: big picture

- ❑ Sender can have up to N unacked packets in pipeline
- ❑ Rcvr acks individual packets
- ❑ Sender maintains timer for each unacked packet
 - When timer expires, retransmit only unack packet

Go-Back-N

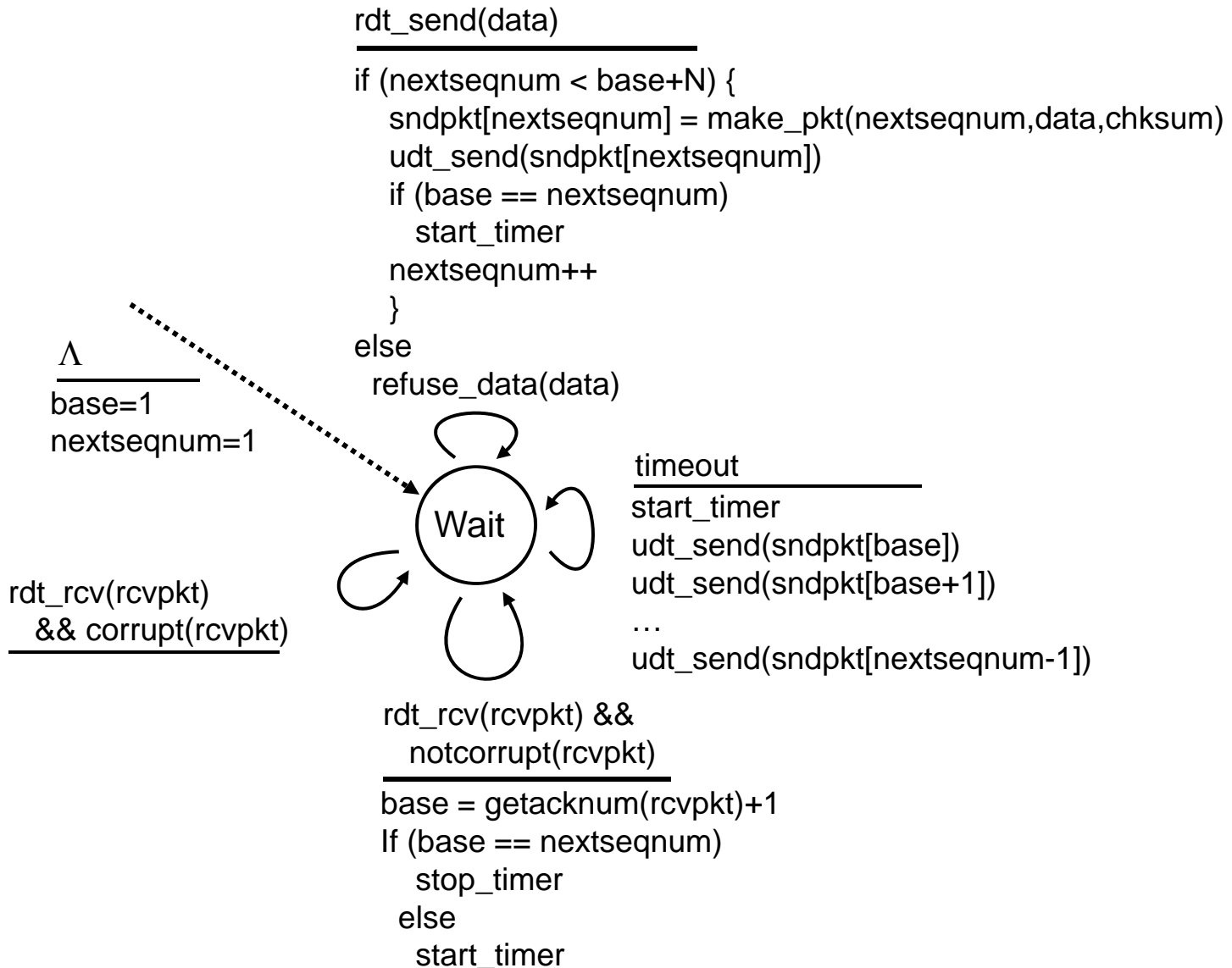
Sender:

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

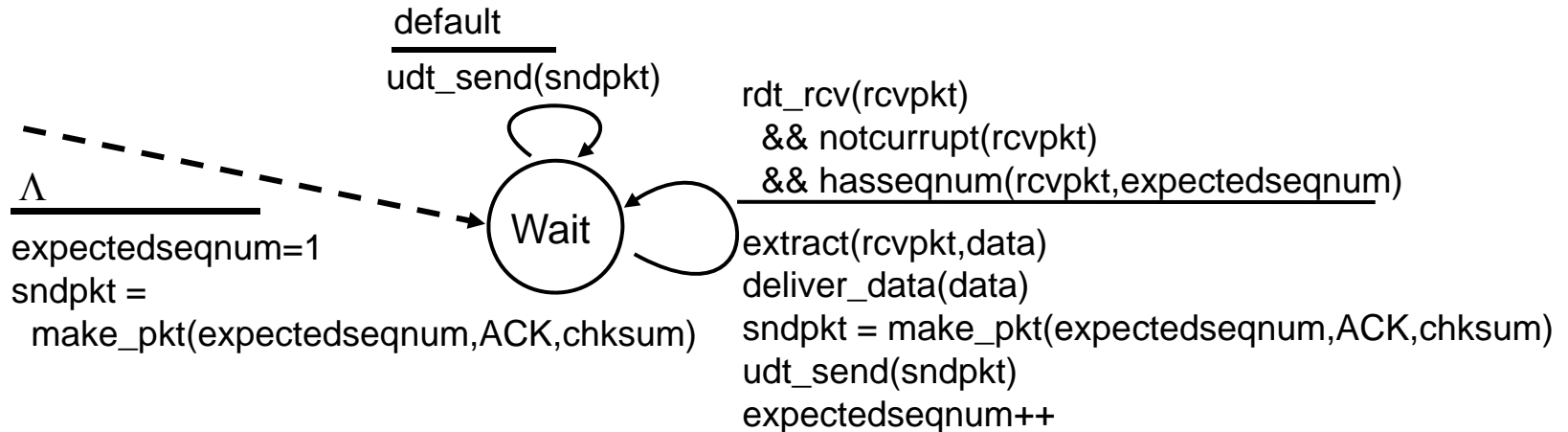


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
 - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout*(n): retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM



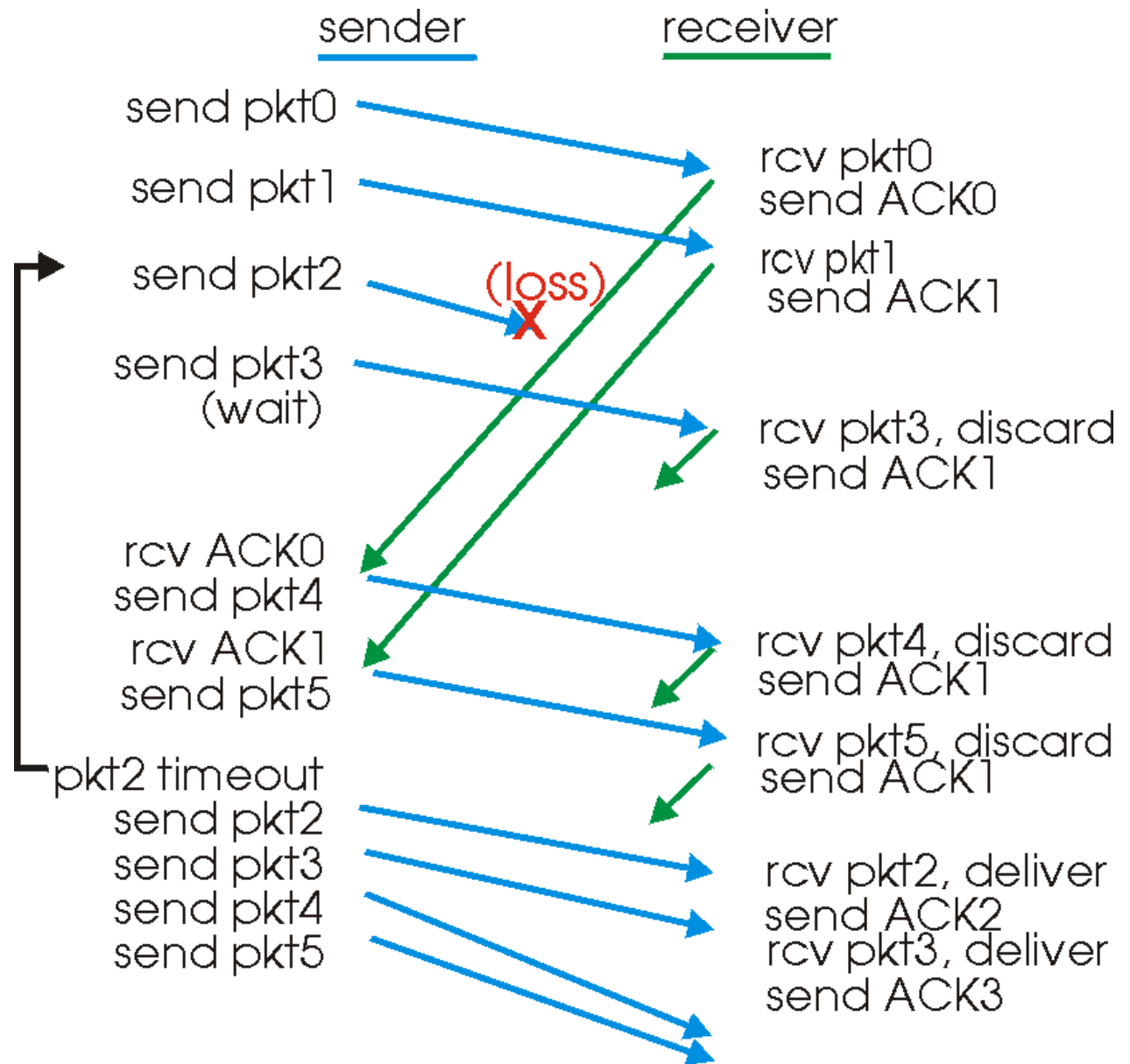
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember `expectedseqnum`

□ out-of-order pkt:

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

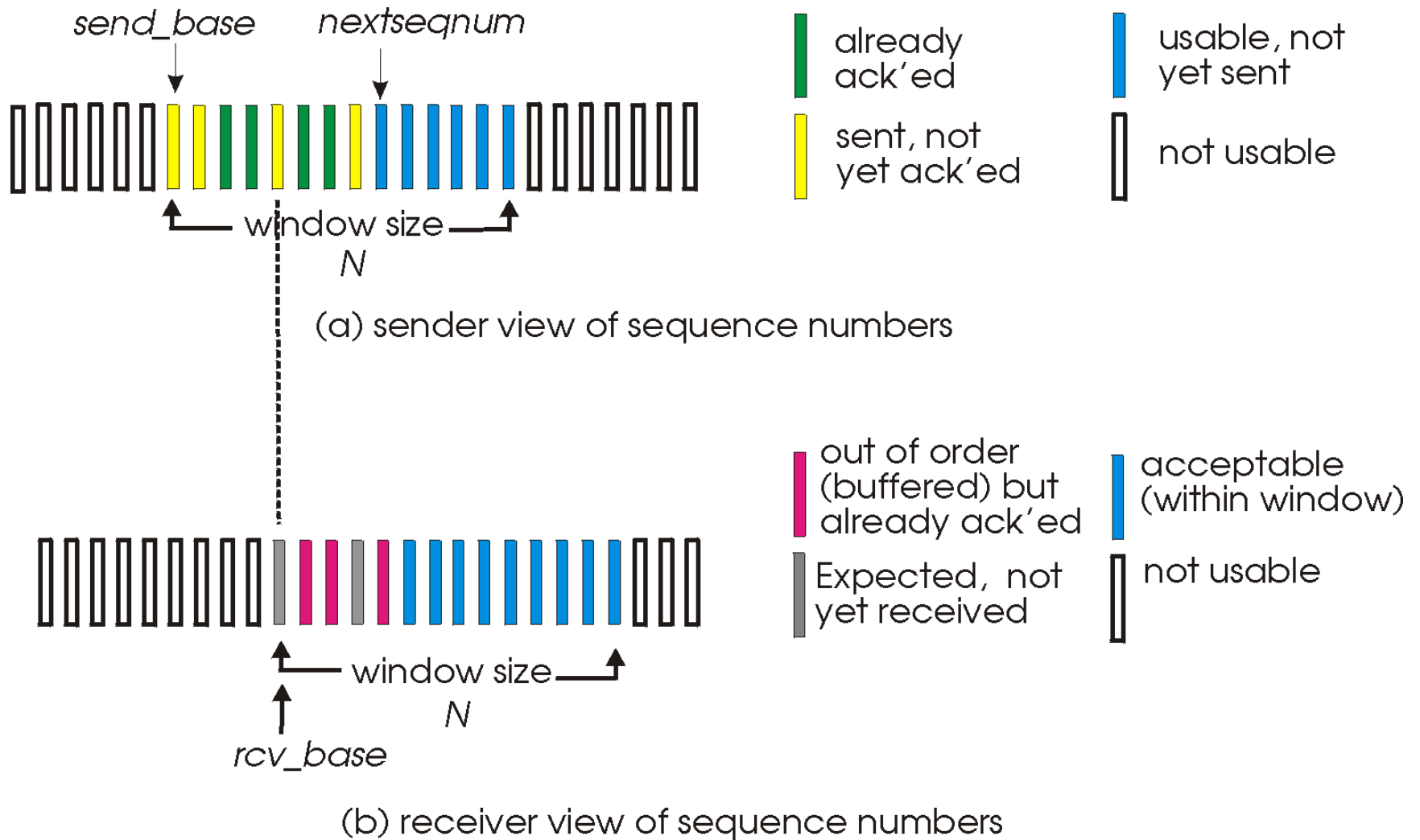
GBN in action



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



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

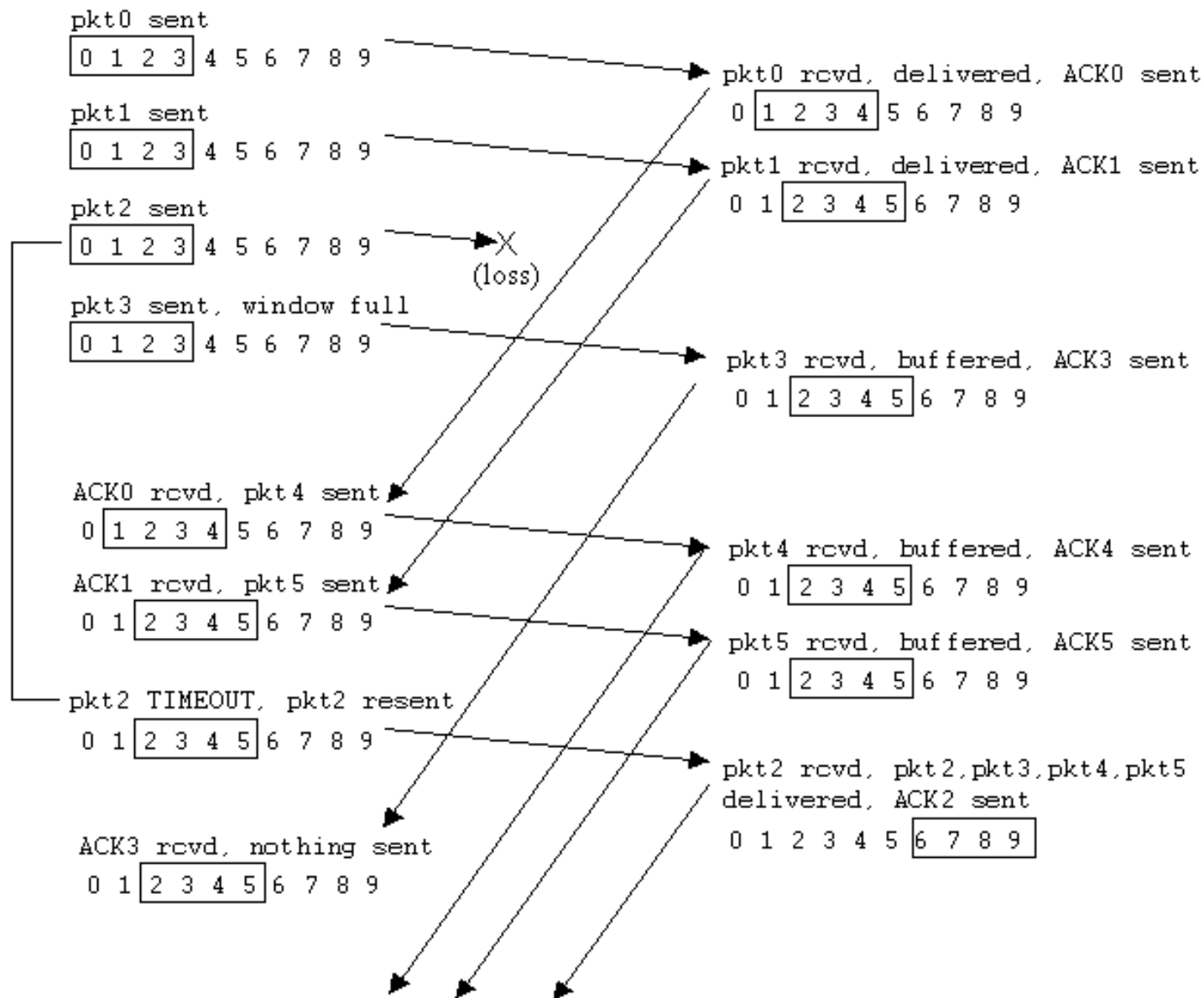
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



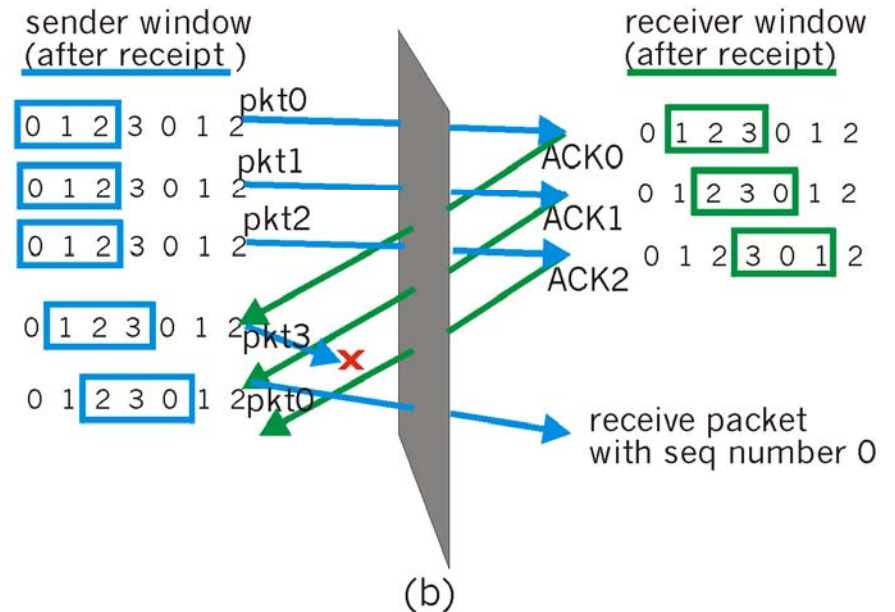
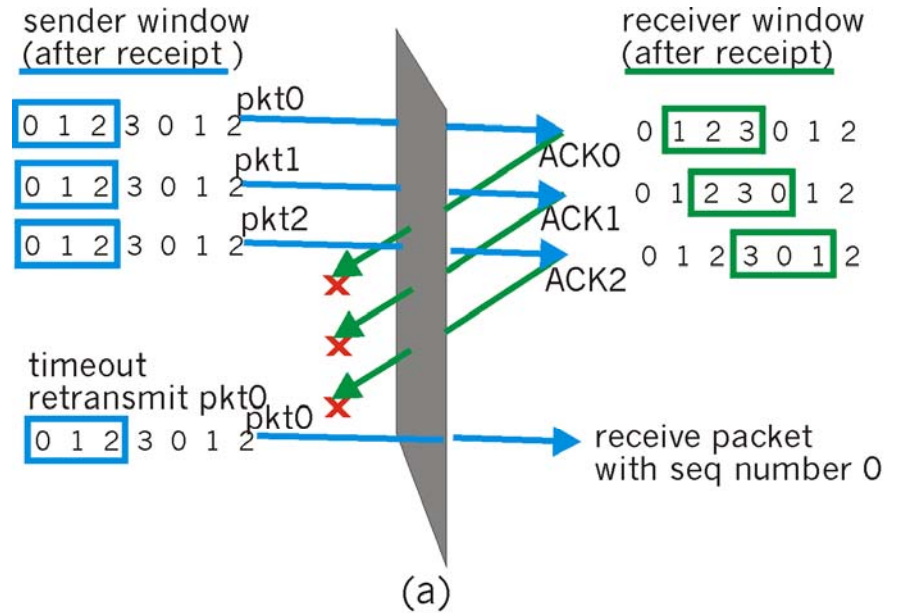
Selective repeat: dilemma

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?

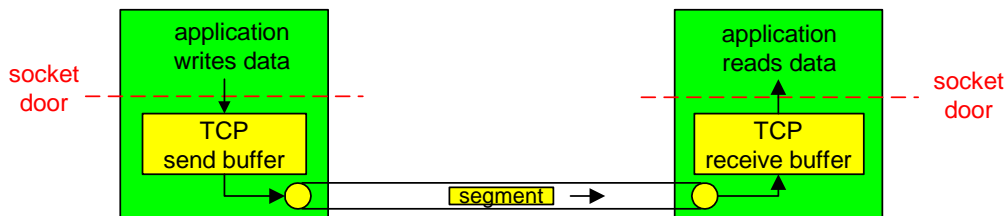


TCP: Overview

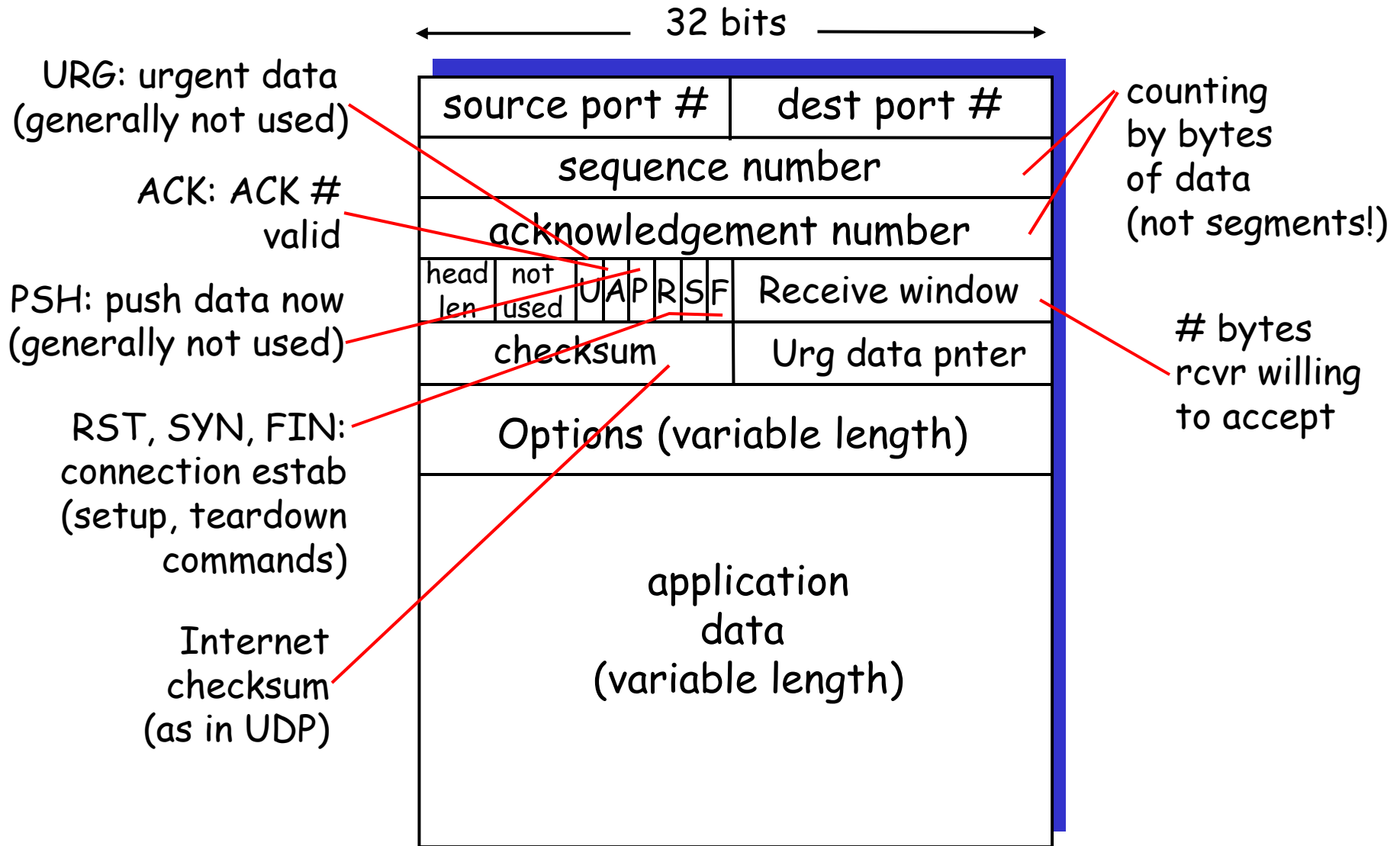
RFCs: 793, 1122, 1323, 2018, 2581

- ❑ **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



TCP seq. #'s and ACKs

Seq. #'s:

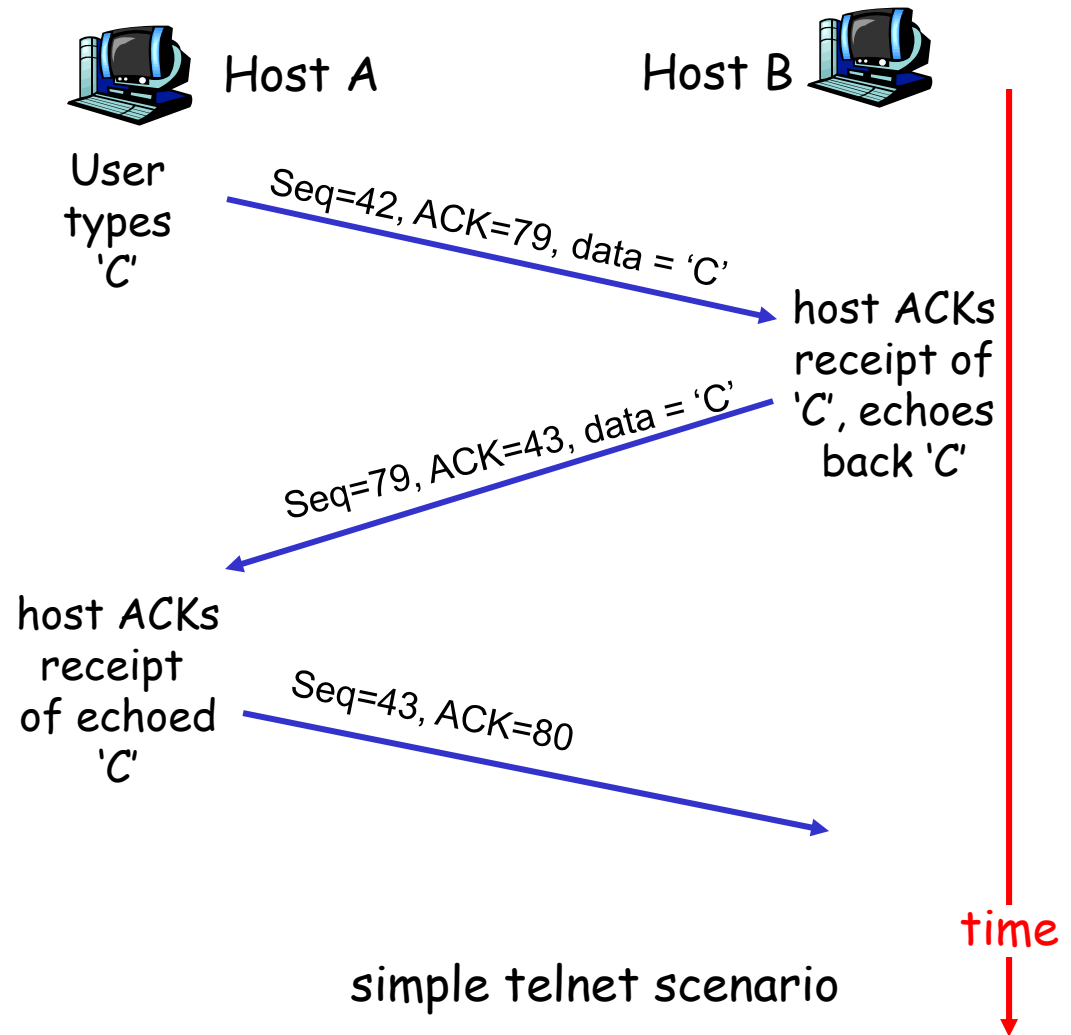
- byte stream
"number" of first byte in segment's data

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 implementor



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`

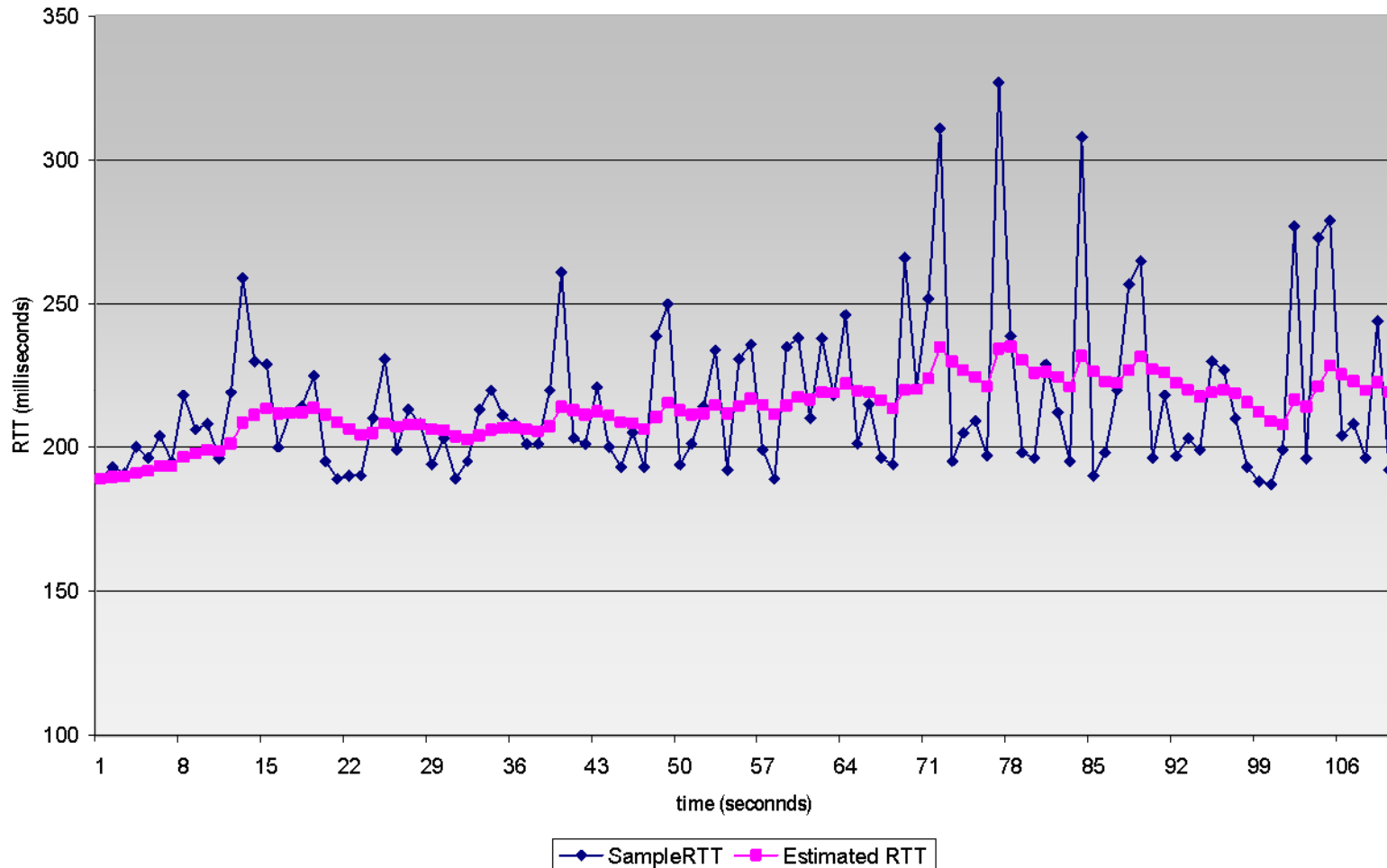
TCP Round Trip Time and Timeout

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

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

Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr



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:

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

(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

TCP reliable data transfer

- ❑ TCP creates rdt service on top of IP's unreliable service
- ❑ Pipelined segments
- ❑ Cumulative acks
- ❑ TCP uses single retransmission timer
- ❑ 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
- ❑ restart timer

Ack rcvd:

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

```
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 */
```

TCP sender (simplified)

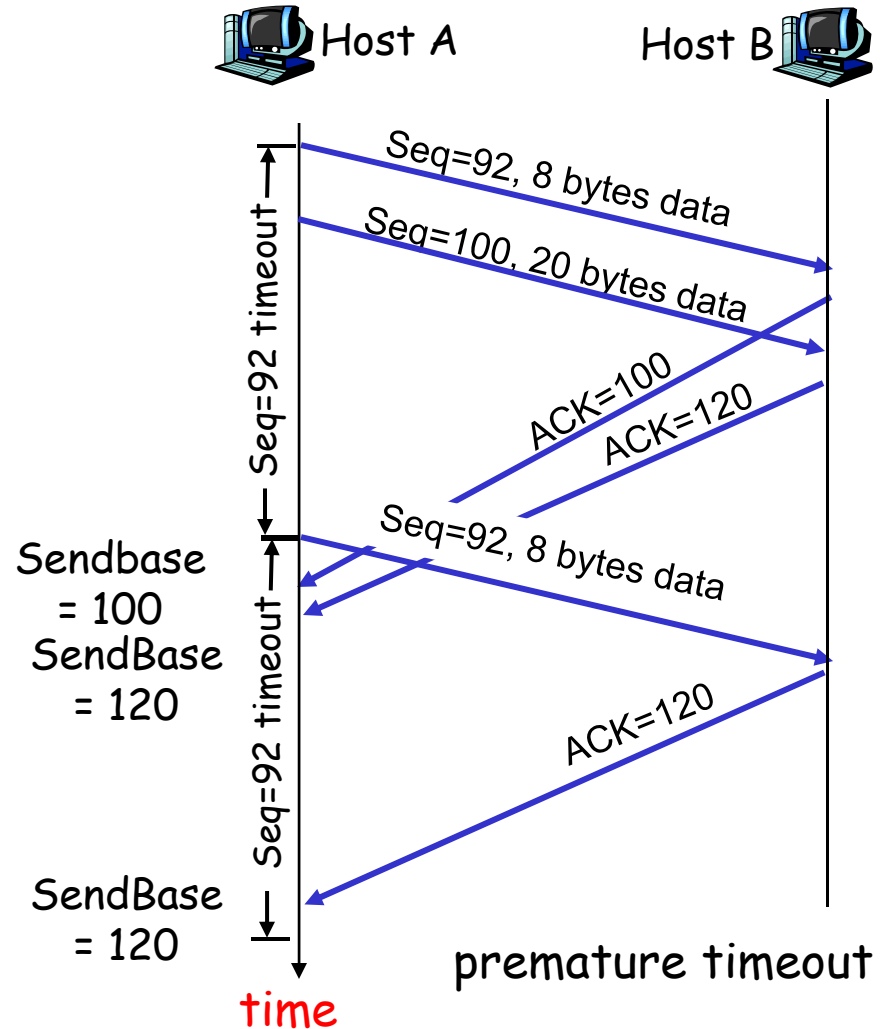
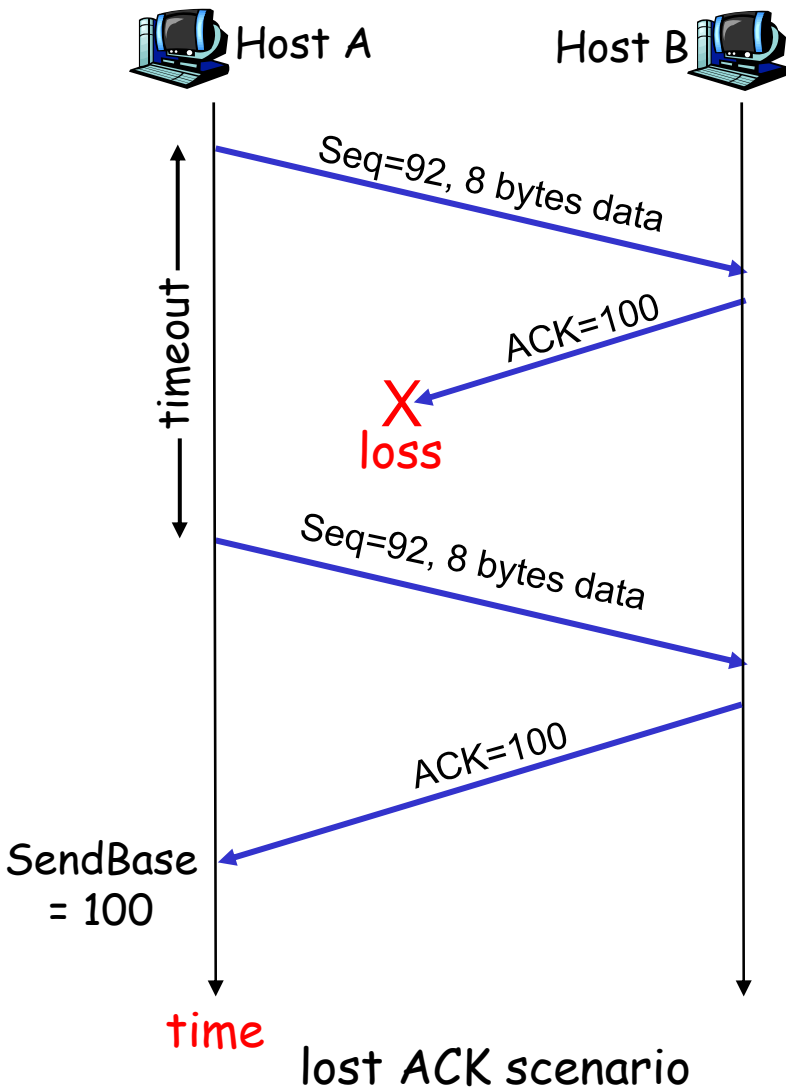
Comment:

- SendBase-1: last cumulatively ack'ed byte

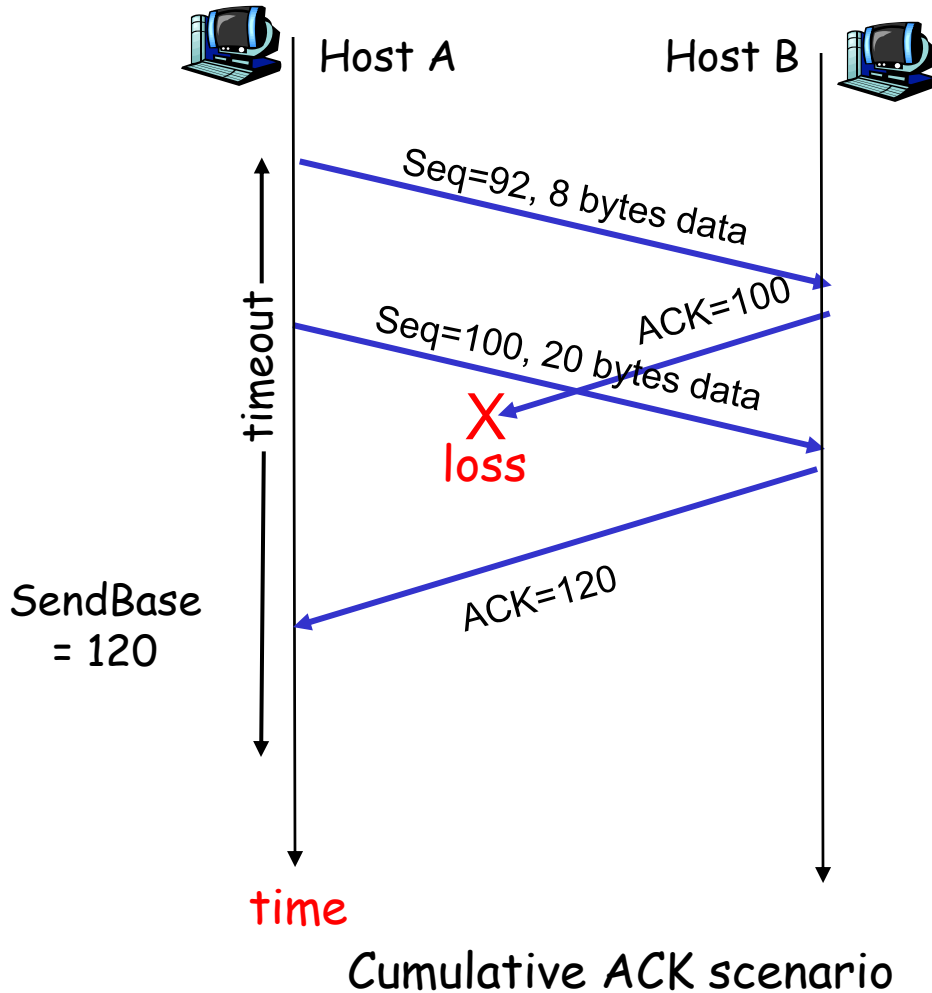
Example:

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

TCP: retransmission scenarios



TCP retransmission scenarios (more)



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-expected 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

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

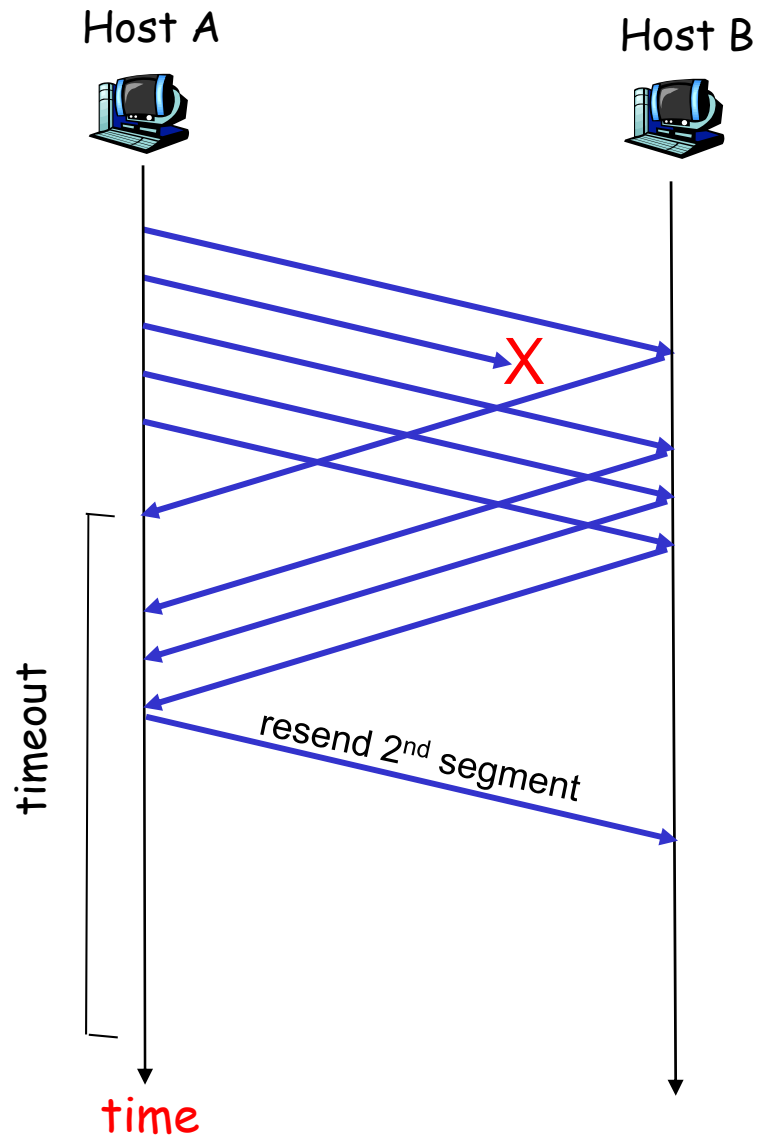


Figure 3.37 Resending a segment after triple duplicate ACK

Fast retransmit algorithm:

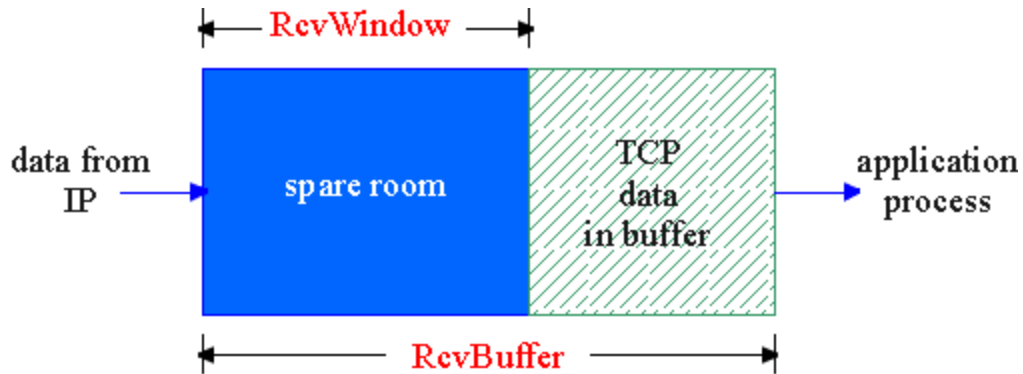
```
event: ACK received, with ACK field value of y
    if (y > SendBase) {
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
            start timer
    }
    else {
        increment count of dup ACKs received for y
        if (count of dup ACKs received for y = 3)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



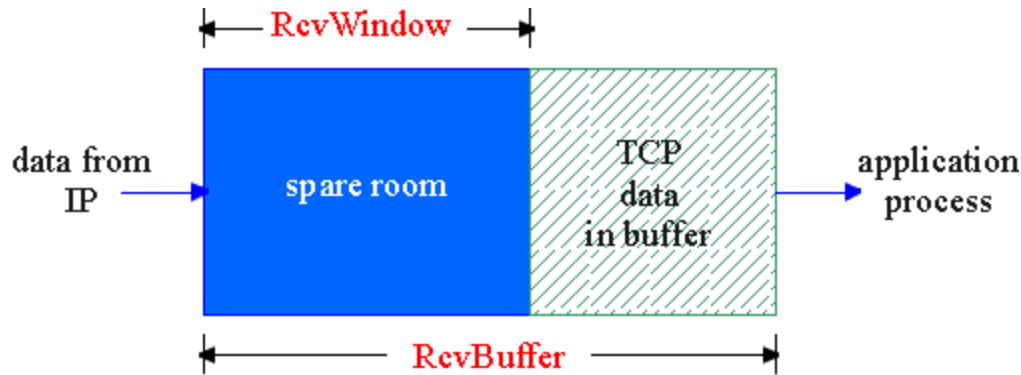
- app process may be slow at reading from buffer

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
- = RcvWindow
- = $\text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$

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

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.)

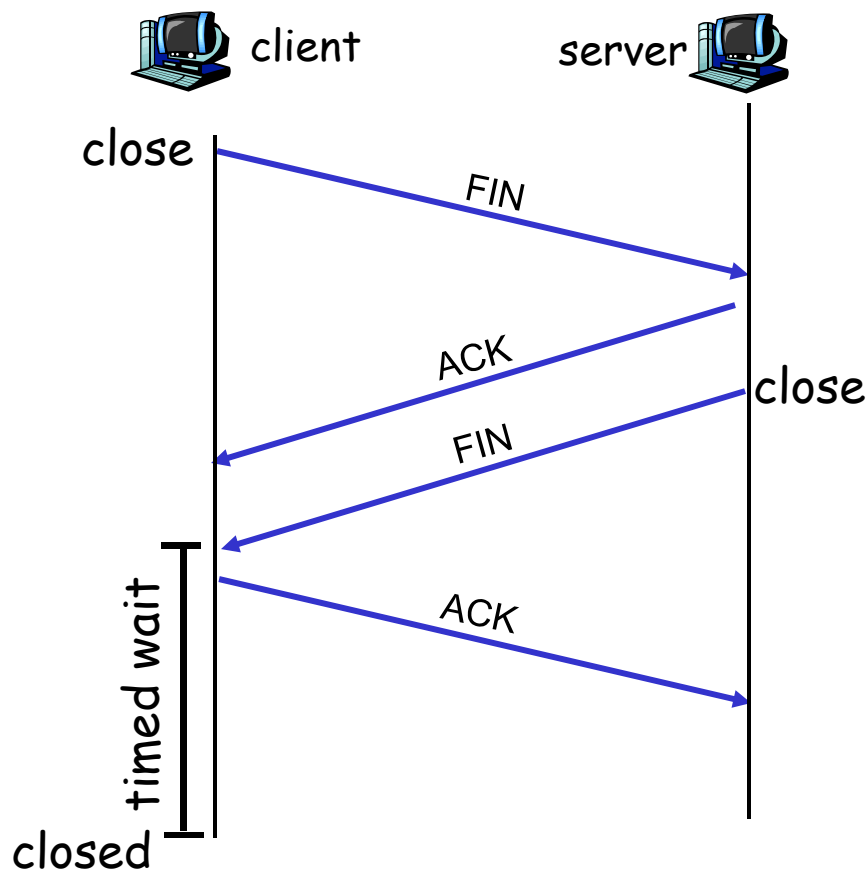
Closing a connection:

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.



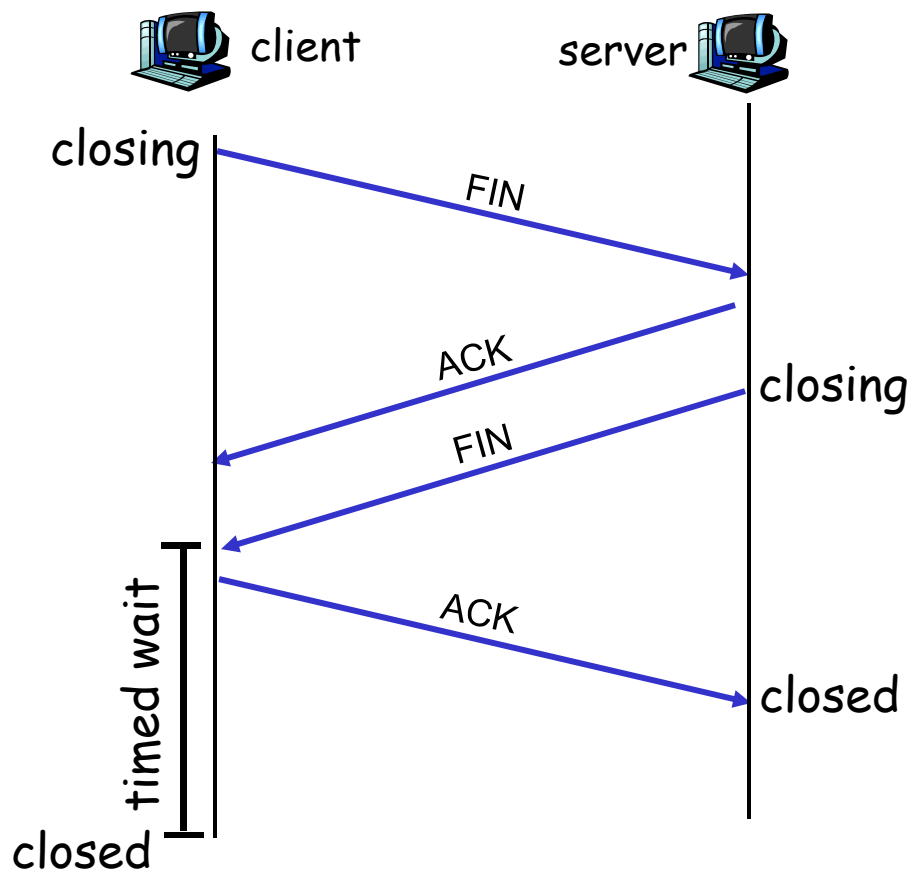
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

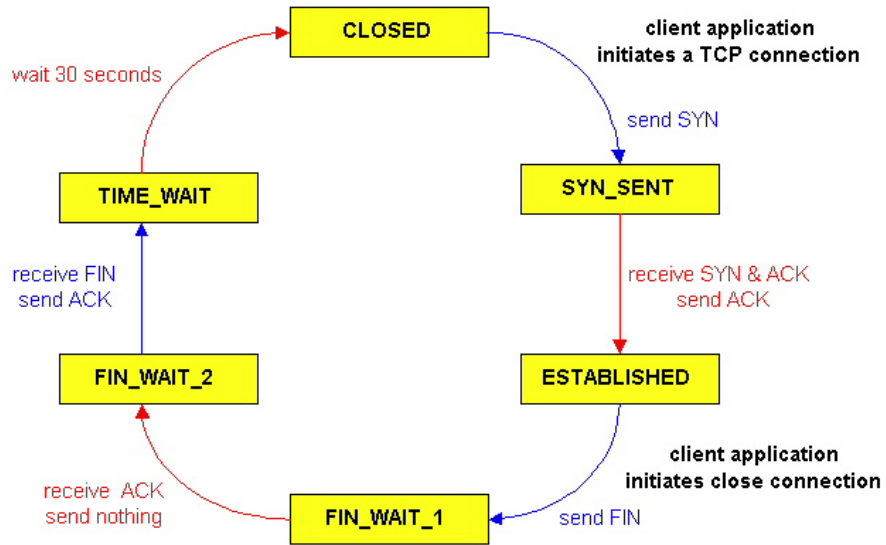
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

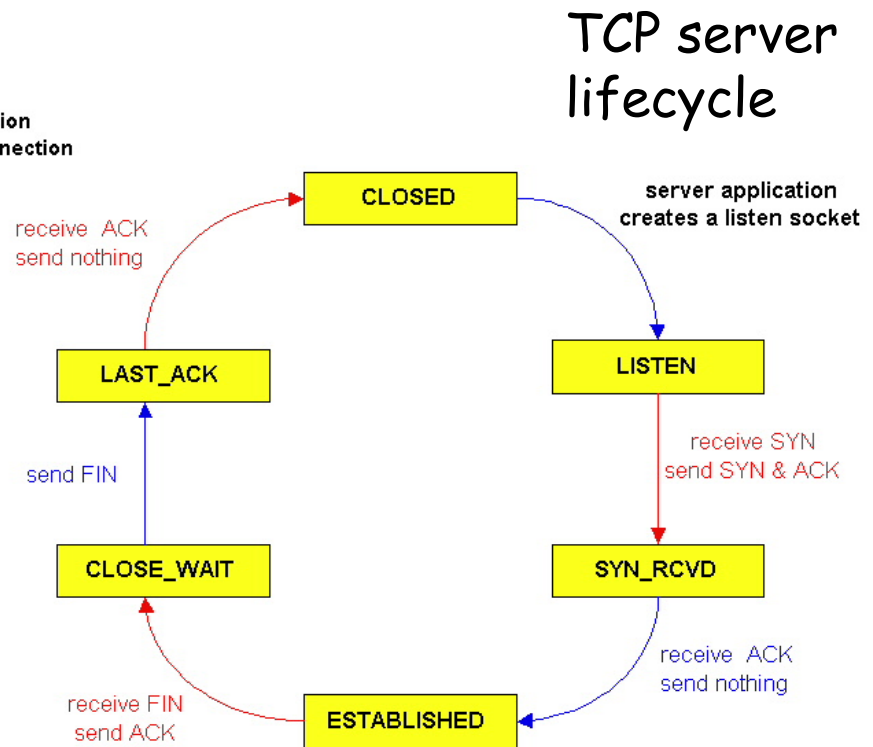
Note: with small modification, can handle simultaneous FINs.



TCP Connection Management (cont)



TCP client lifecycle



TCP server lifecycle

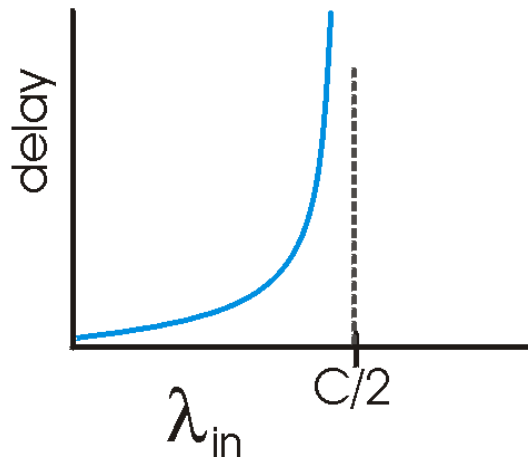
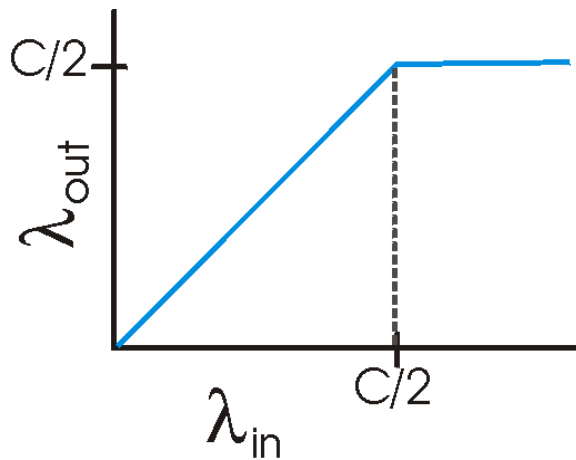
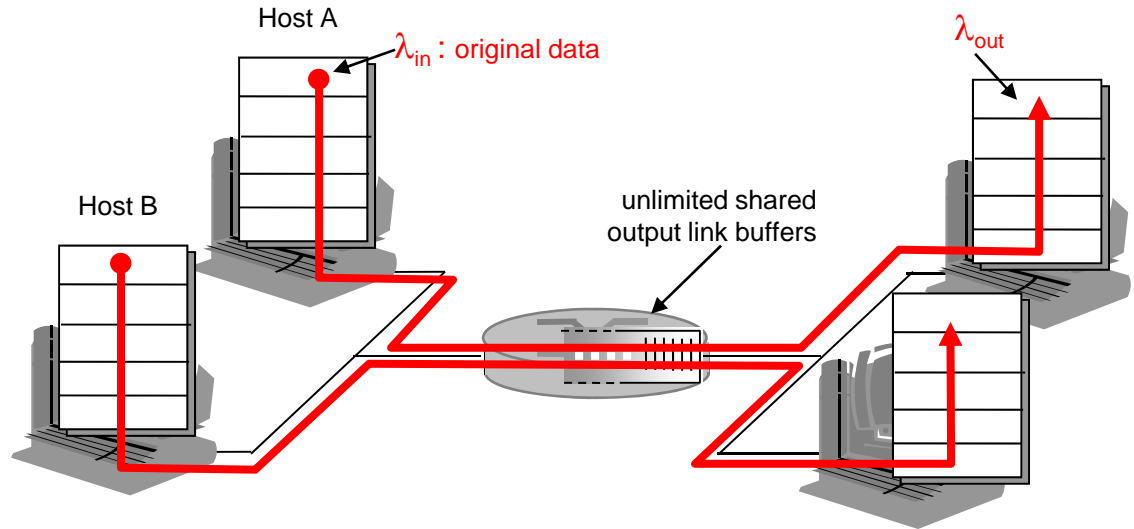
Principles of Congestion Control

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

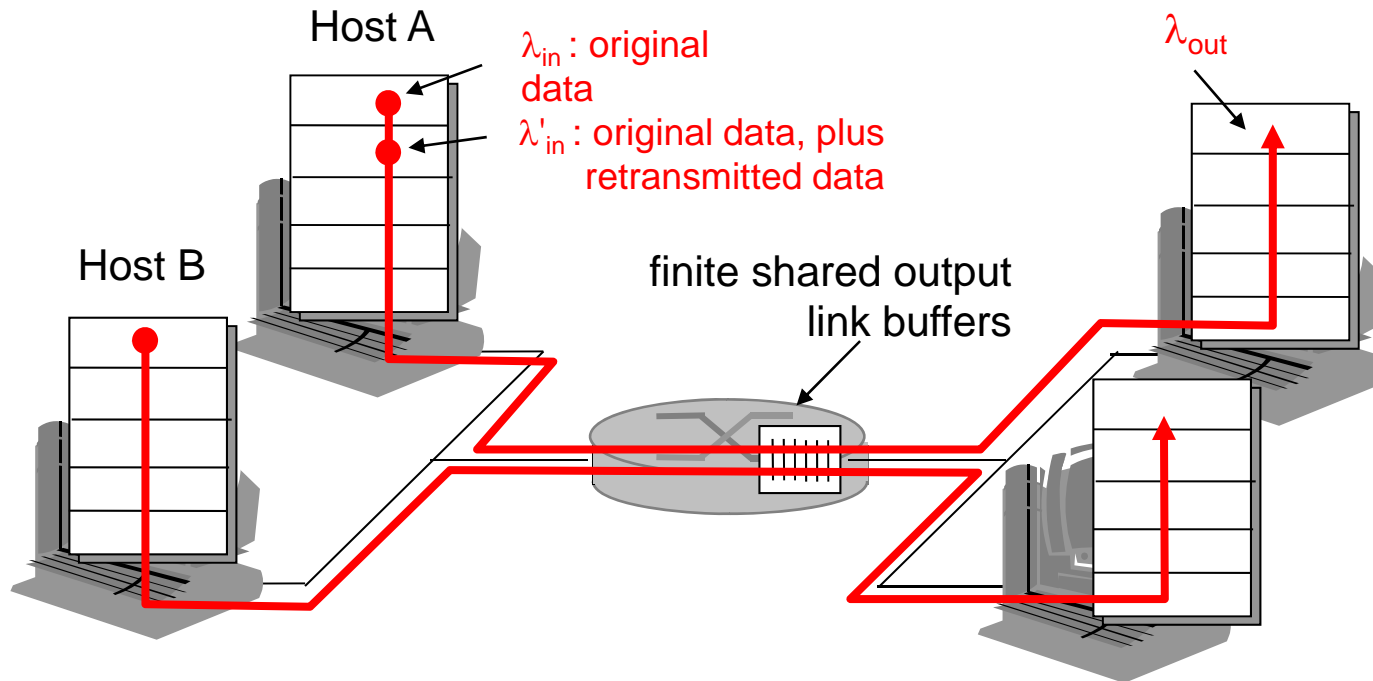
- two senders, two receivers
- one router, infinite buffers
- no retransmission



- large delays when congested
- maximum achievable throughput

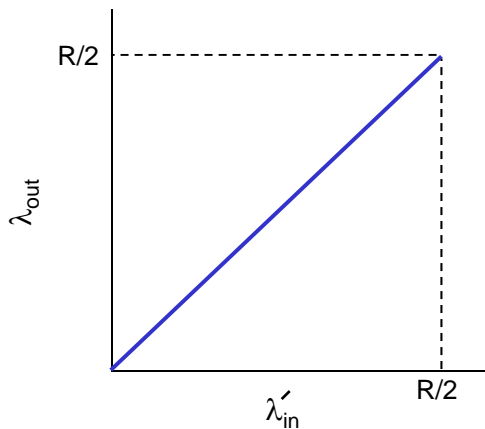
Causes/costs of congestion: scenario 2

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

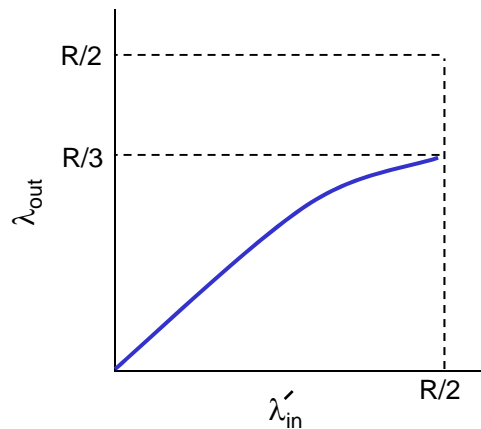


Causes/costs of congestion: scenario 2

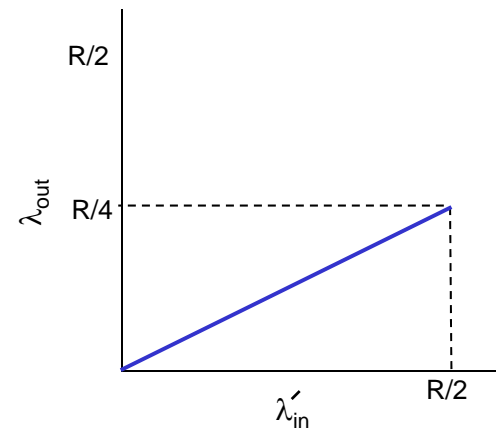
- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a.



b.



c.

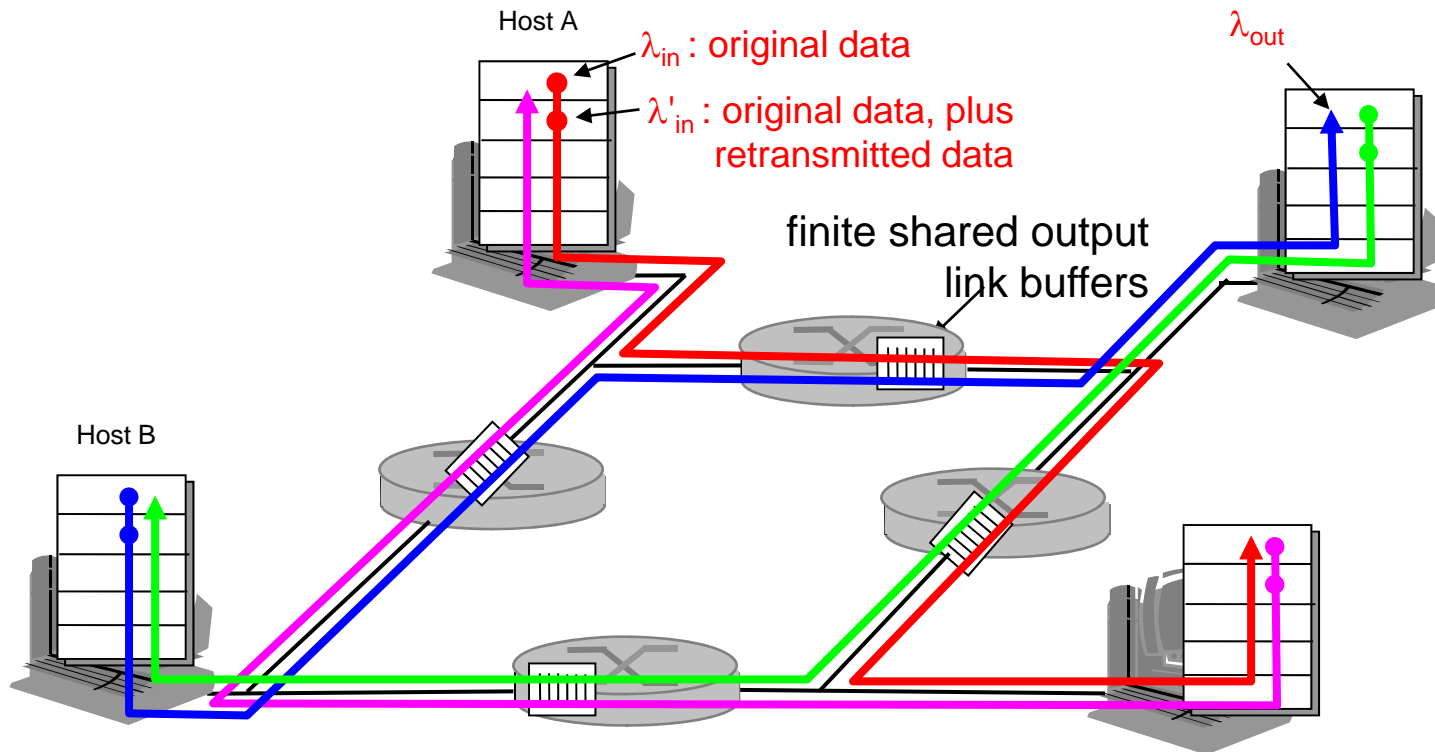
"costs" of congestion:

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

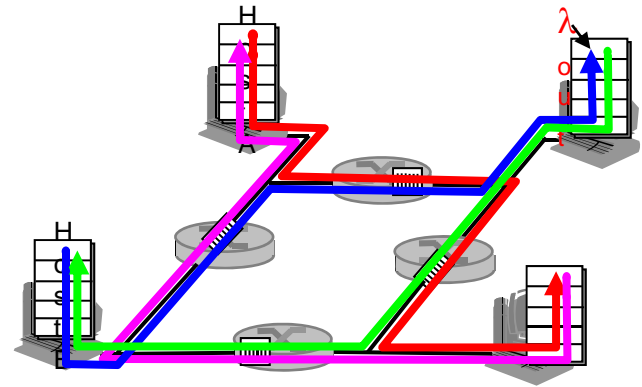
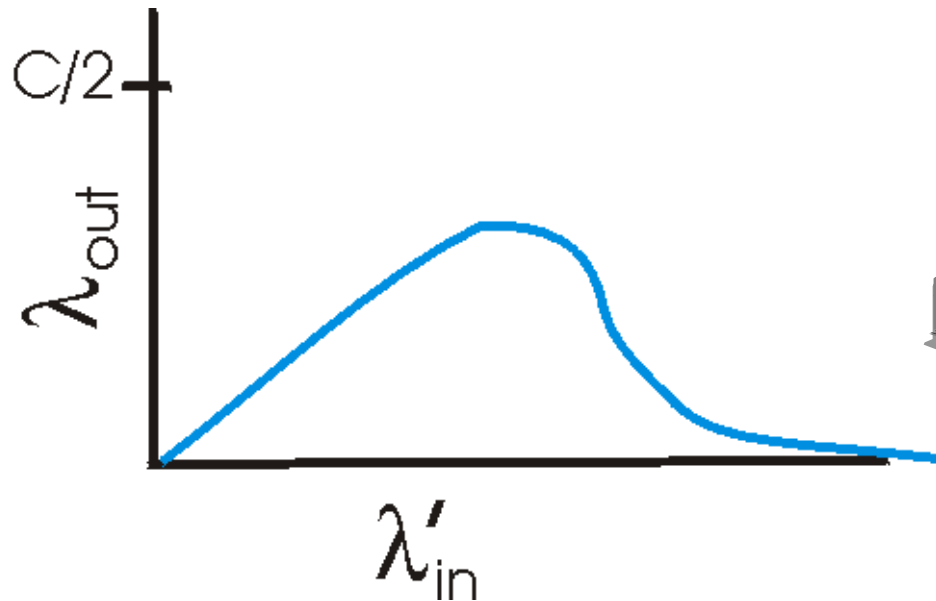
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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



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

Case study: ATM ABR congestion control

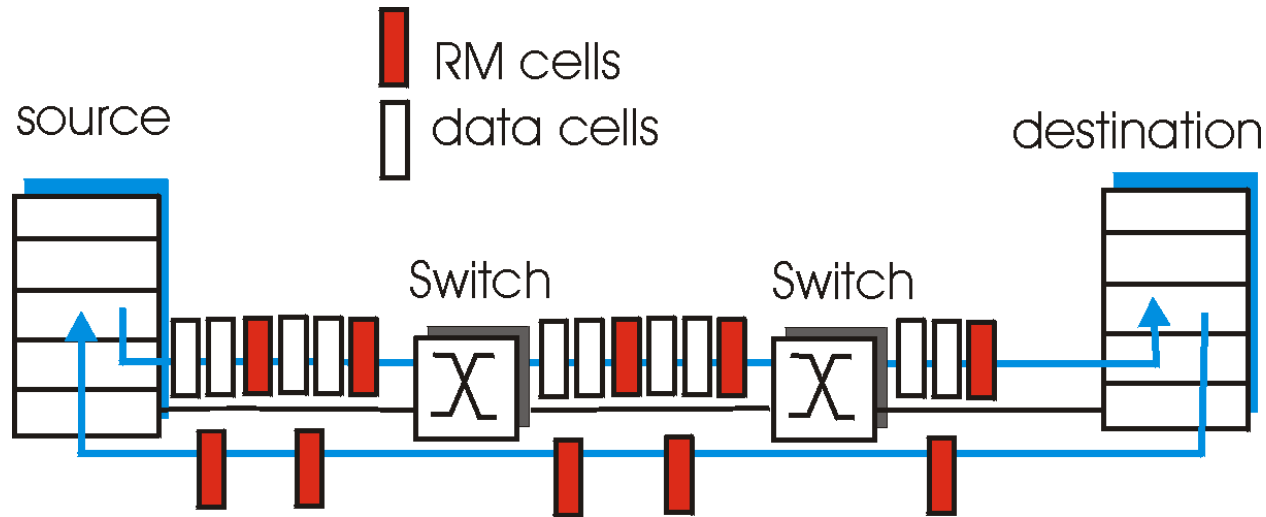
ABR: available bit rate:

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

RM (resource management) cells:

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

Case study: ATM ABR congestion control

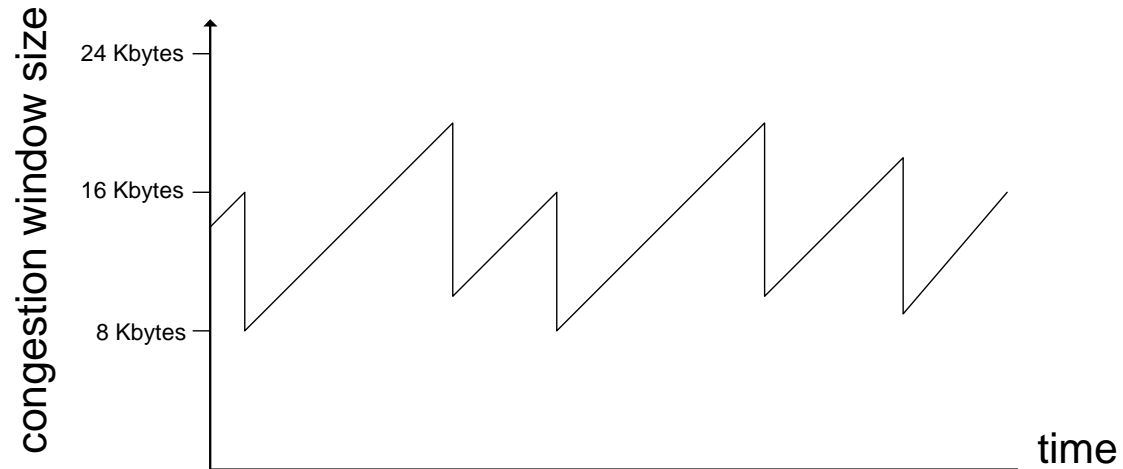


- two-byte ER (explicit rate) field in RM cell
 - congested switch may lower ER value in cell
 - sender' send rate thus maximum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
 - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

TCP congestion control: additive increase, multiplicative decrease

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

Saw tooth behavior: probing for bandwidth



TCP Congestion Control: details

- sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAked} \leq \text{CongWin}$$

- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- 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

three mechanisms:

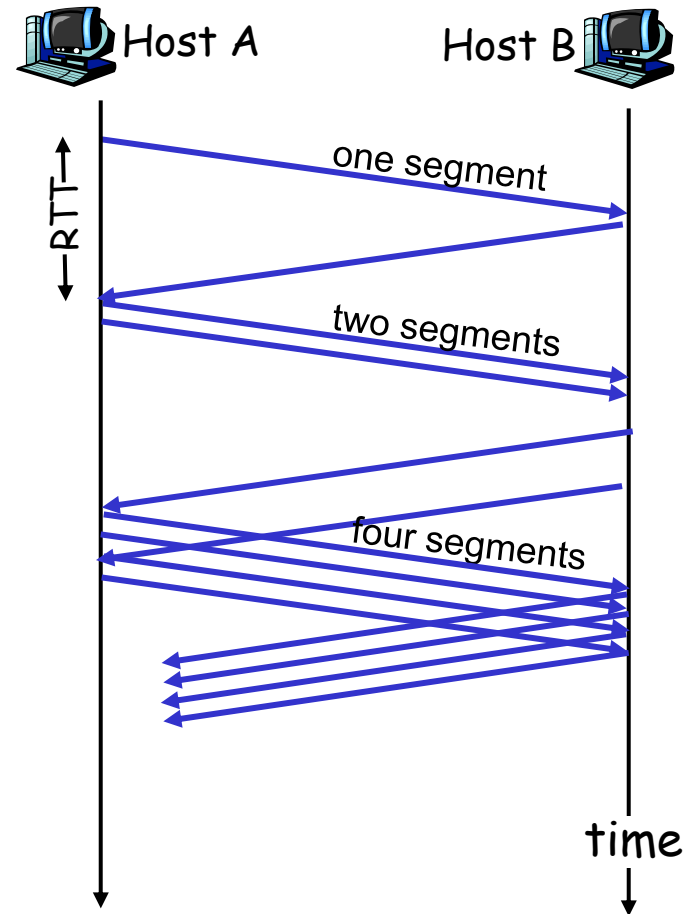
- AIMD
- slow start
- conservative after timeout events

TCP Slow Start

- ❑ When connection begins, $\text{CongWin} = 1 \text{ MSS}$
 - Example: $\text{MSS} = 500$ bytes & $\text{RTT} = 200 \text{ msec}$
 - initial rate = 20 kbps
- ❑ available bandwidth may be $\gg \text{MSS}/\text{RTT}$
 - desirable to quickly ramp up to respectable rate
- ❑ When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
 - double CongWin every RTT
 - done by incrementing CongWin for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



Refinement: inferring loss

- After 3 dup ACKs:
 - CongWin is cut in half
 - window then grows linearly
- But after timeout event:
 - CongWin instead set to 1 MSS;
 - window then grows exponentially
 - to a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

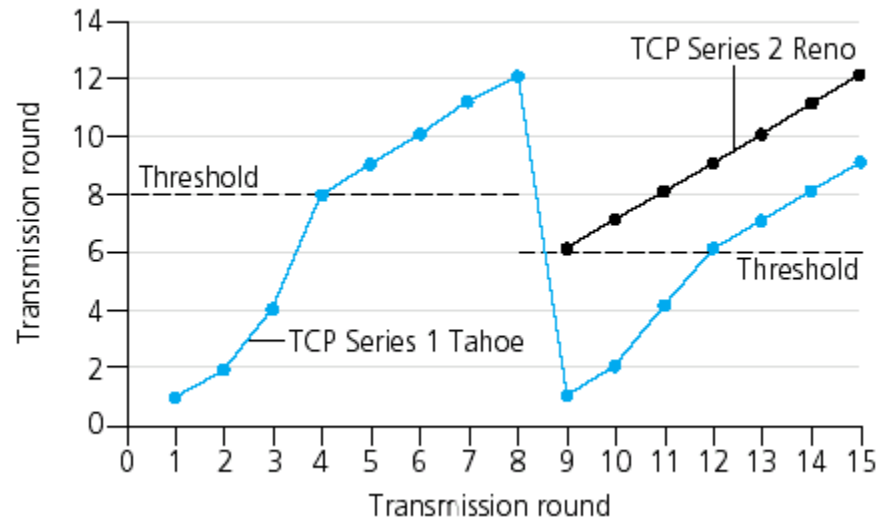
Refinement

Q: When should the exponential increase switch to linear?

A: When `CongWin` gets to 1/2 of its value before timeout.

Implementation:

- ❑ Variable Threshold
- ❑ At loss event, Threshold is set to 1/2 of `CongWin` just before loss event



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 $\text{CongWin}/2$ and CongWin set to Threshold.
- ❑ When **timeout** occurs, Threshold set to $\text{CongWin}/2$ and CongWin is set to 1 MSS.

TCP sender congestion control

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

TCP throughput

- ❑ What's the average throughput of TCP as a function of window size and RTT?
 - Ignore slow start
- ❑ Let W be the 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 throughput: $.75 W/RTT$

TCP Futures: TCP over "long, fat pipes"

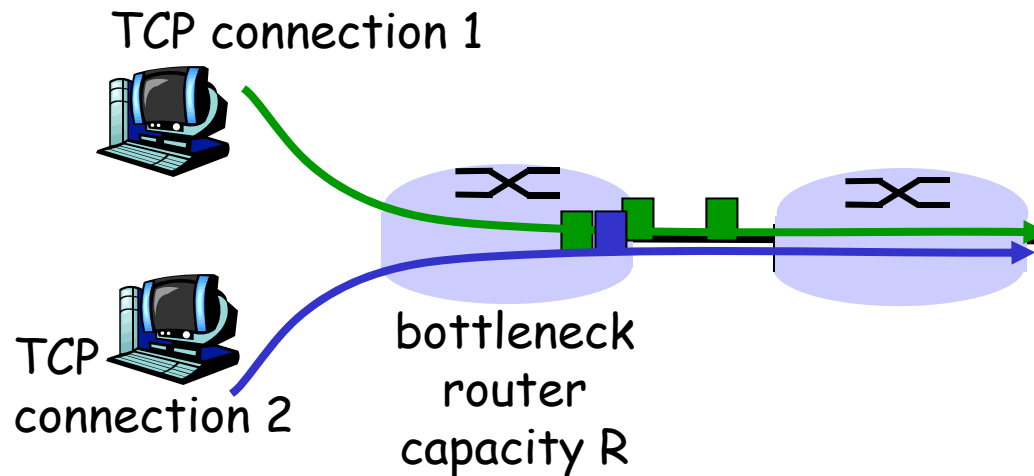
- ❑ Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- ❑ Requires window size $W = 83,333$ in-flight segments
- ❑ Throughput in terms of loss rate:

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

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

TCP Fairness

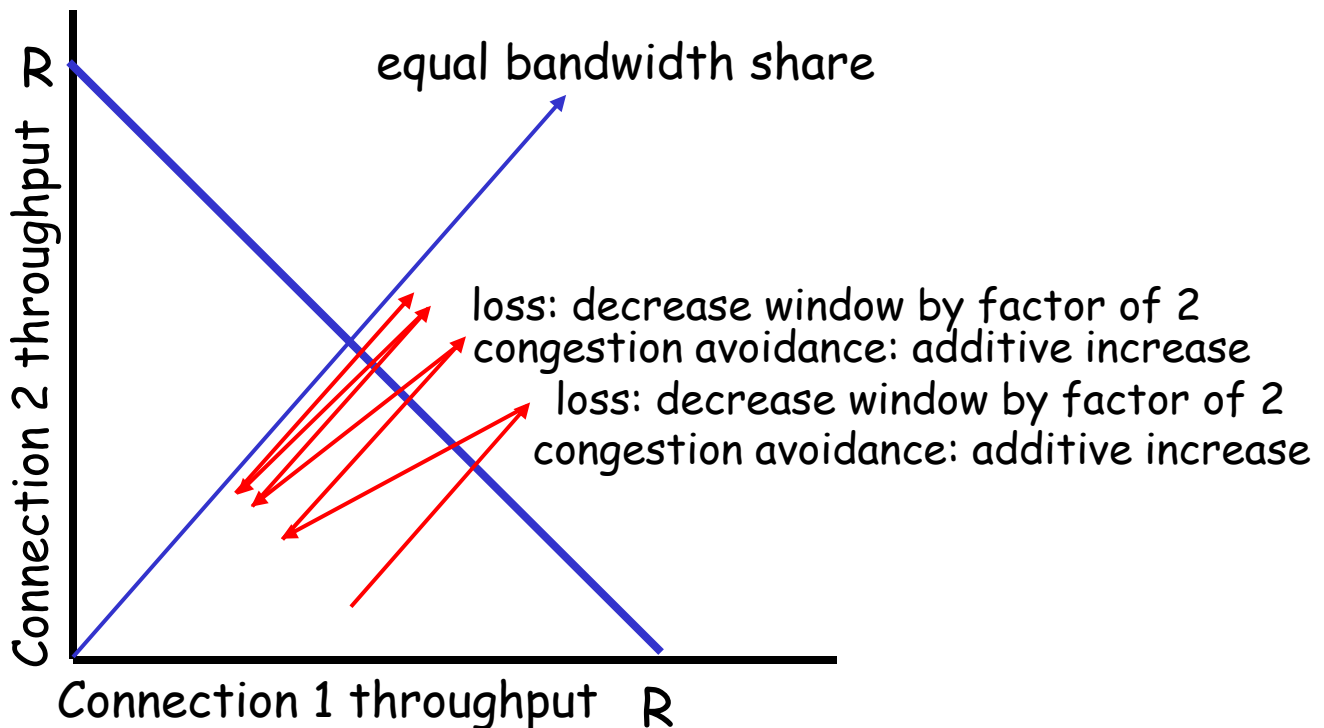
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput 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:
 - pump audio/video at constant rate, tolerate packet loss
- ❑ Research area: TCP friendly

Fairness and parallel TCP connections

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