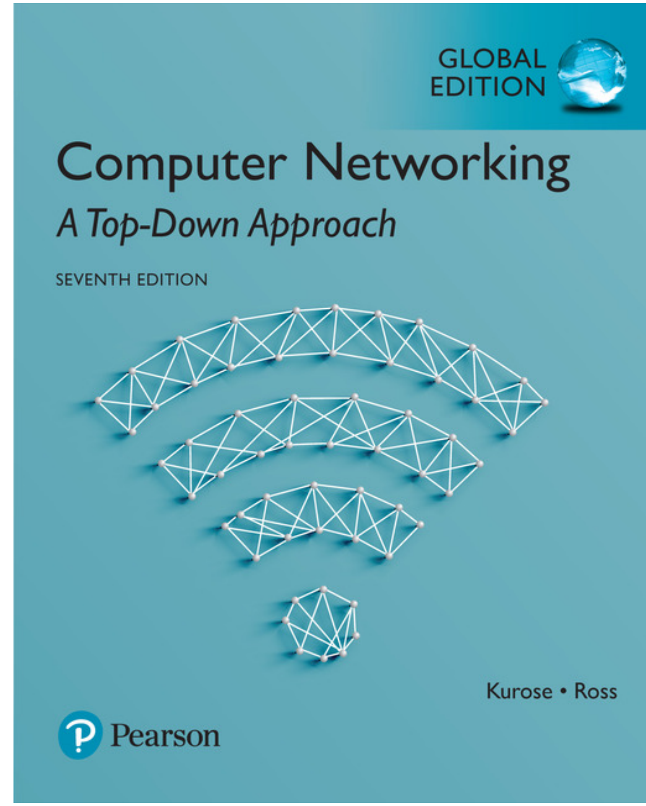
Chapter 3 Transport Layer



Computer Networking: A Top-Down Approach

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

A note on the use of these Powerpoint slides:

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

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

Thanks and enjoy! JFK/KWR

Chapter 3: Transport Layer

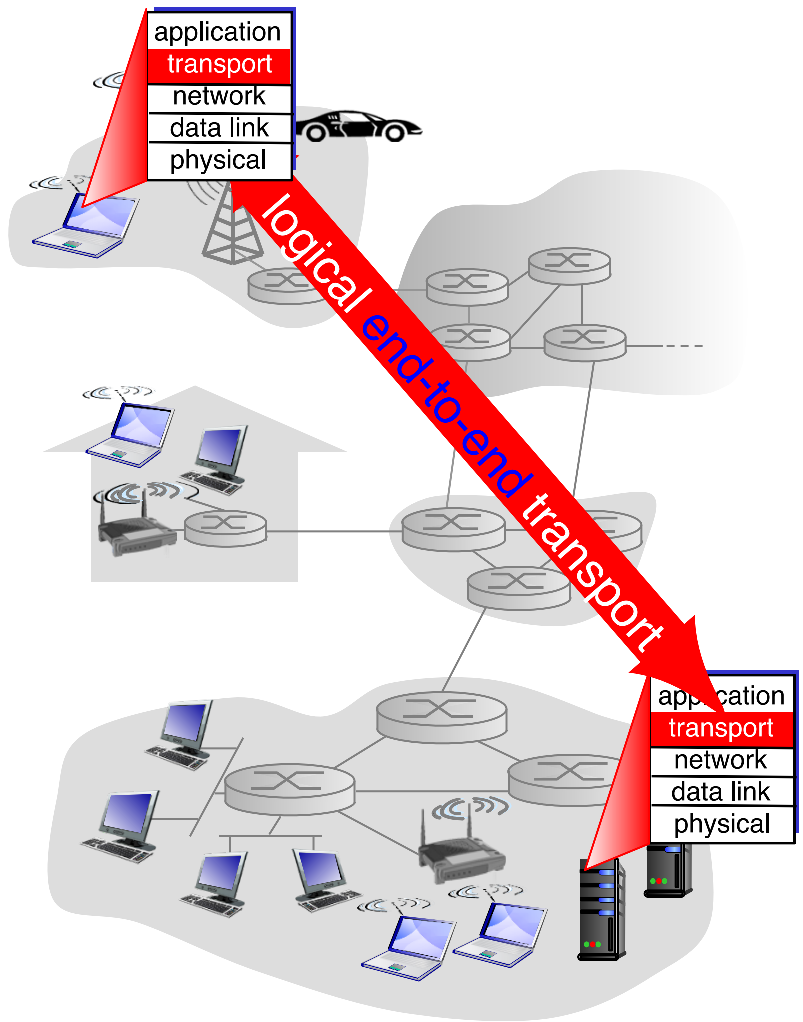
our goals:

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

Chapter 3 outline

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



• logical connection: as if the hosts are directly connected

Transport vs. network layer

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

• Ann and Bill do all their work within their respective homes

• transport-layer protocols live in the end systems

household analogy:

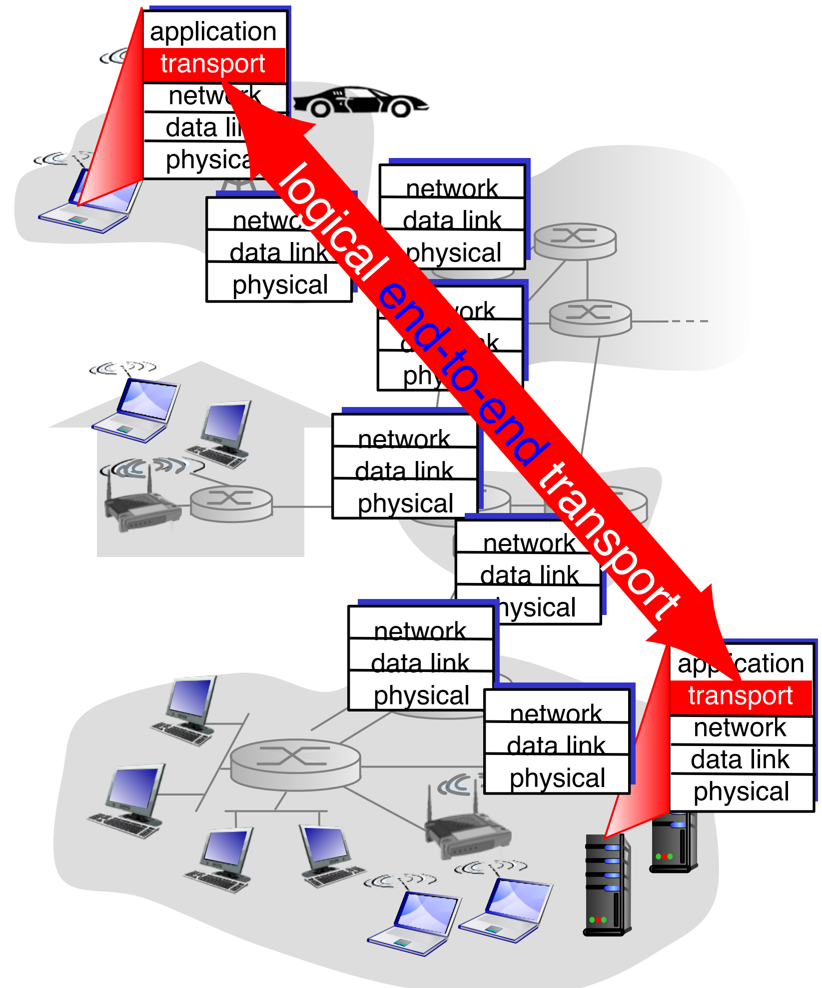
12 kids in Ann's house sending letters to 12 kids in Bill's house:

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

• Susan and Harvey may substitute for Ann and Bill, they may drop mails sometimes

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



- TCP and UDP packets → segments
- the network-layer packets → datagrams

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

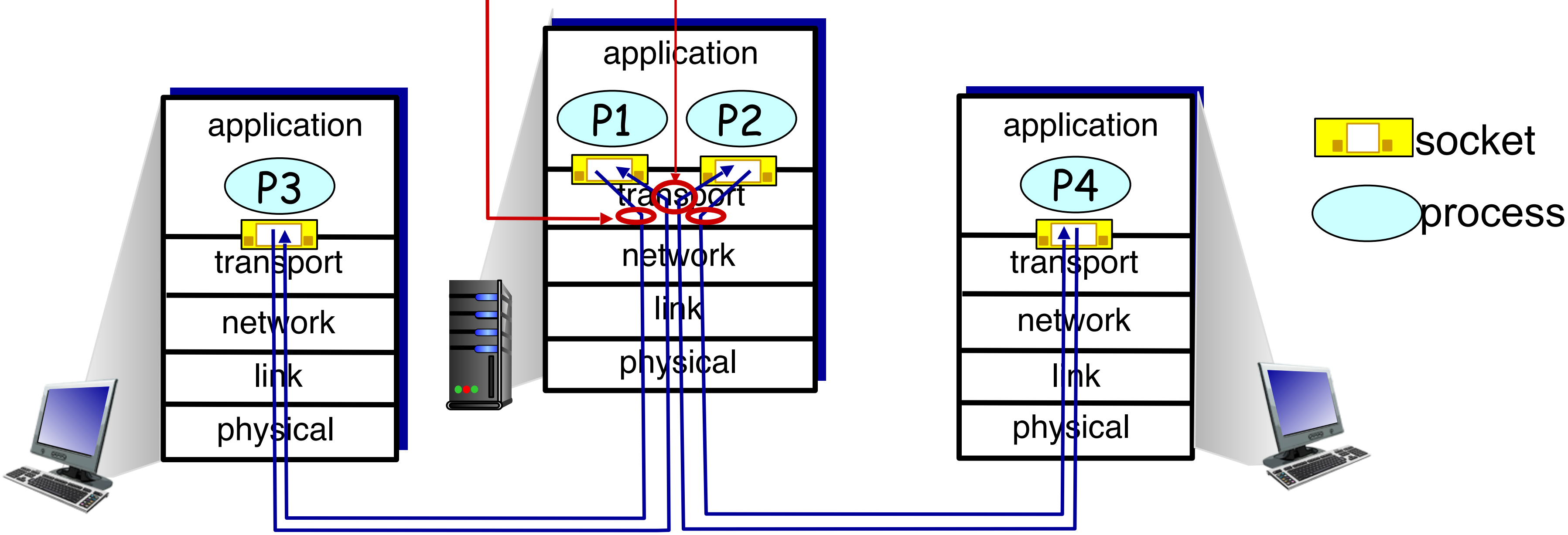
Multiplexing/demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

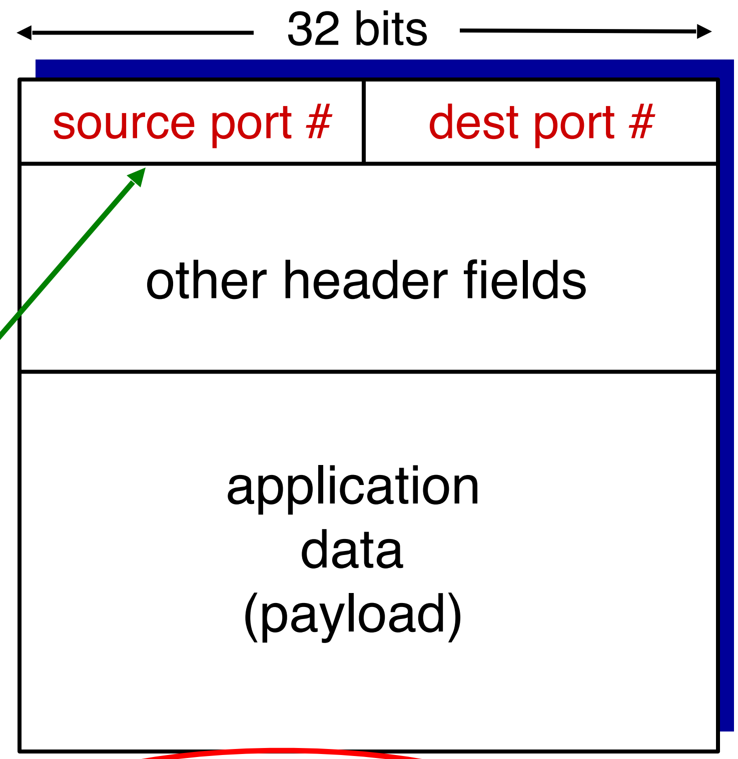
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

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



TCP/UDP segment format

- 16 bits: 0 ~ 65535
- 0 ~ 1023 are well-known port numbers

Connectionless demultiplexing

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

```
clientSocket = socket(AF_INET,
  SOCK_DGRAM)
```

• the transport layer automatically assigns a port number in the range 1024 to 65535

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

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

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

• Why need source port number?

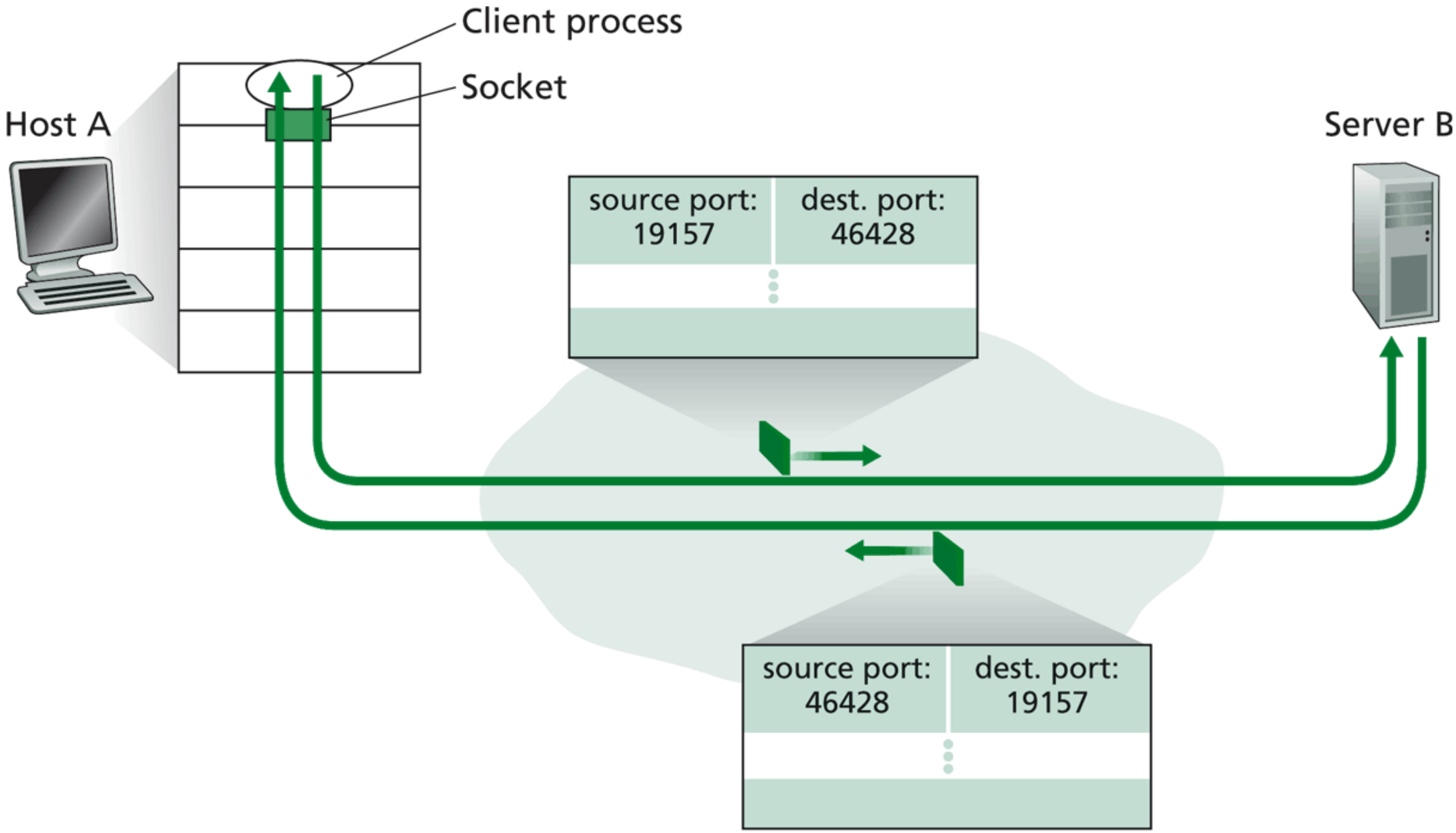


Figure 3.4 ♦ The inversion of source and destination port numbers

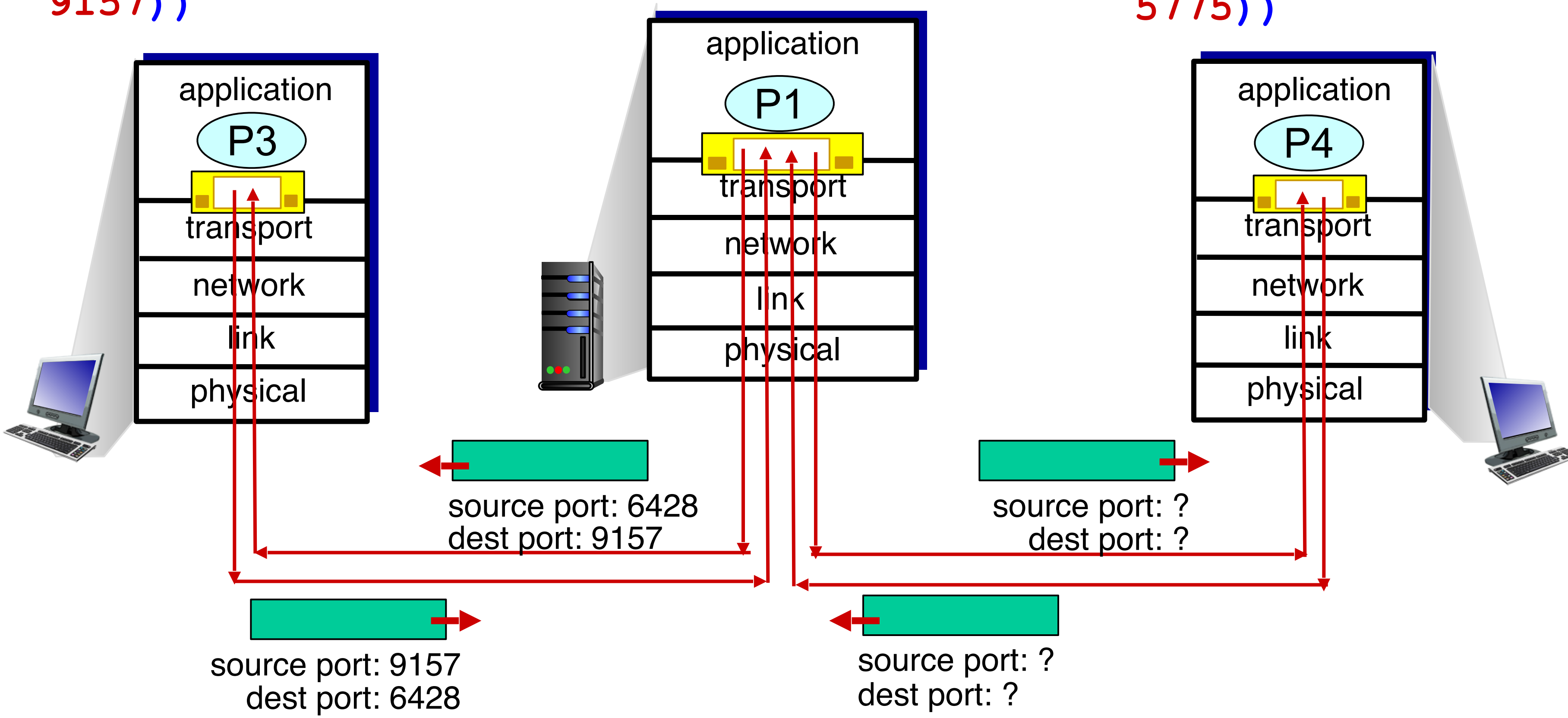
Source port (SP) provides "return address"

Connectionless demux: example

```
mySocket1 =
 socket(AF_INET,
 SOCK_DGRAM)
mySocket1.bind(('',
 9157))
```

```
serverSocket =
 socket(AF_INET,
 SOCK_DGRAM)
serverSocket.bind(('
 ', 6428))
```

```
mySocket2 =
 socket(AF_INET,
 SOCK_DGRAM)
mySocket2.bind(('',
 5775))
```



Connection-oriented demux

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

- UDP socket identified by two-tuple:
  - dest. IP address
  - dest. port number

- 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

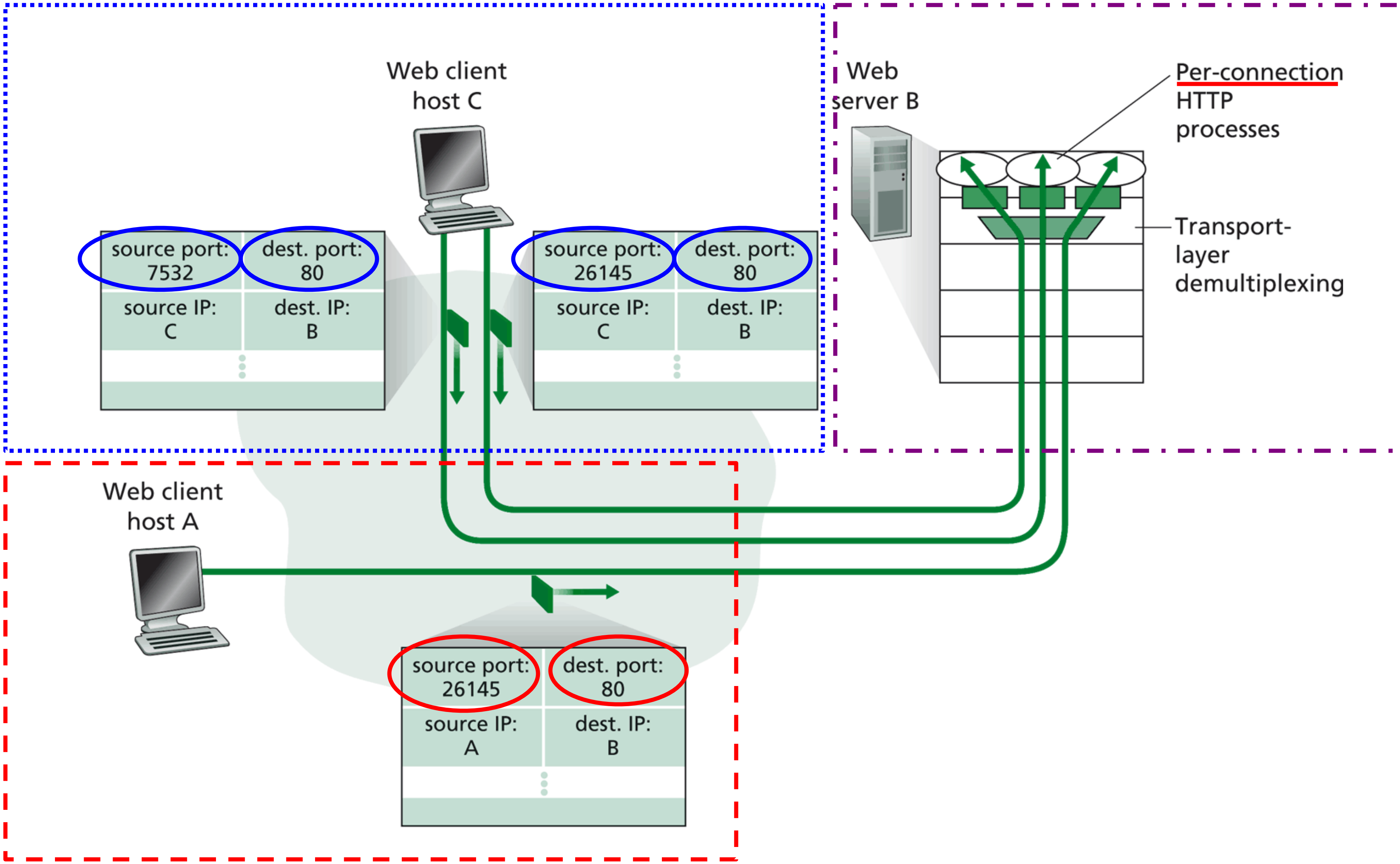
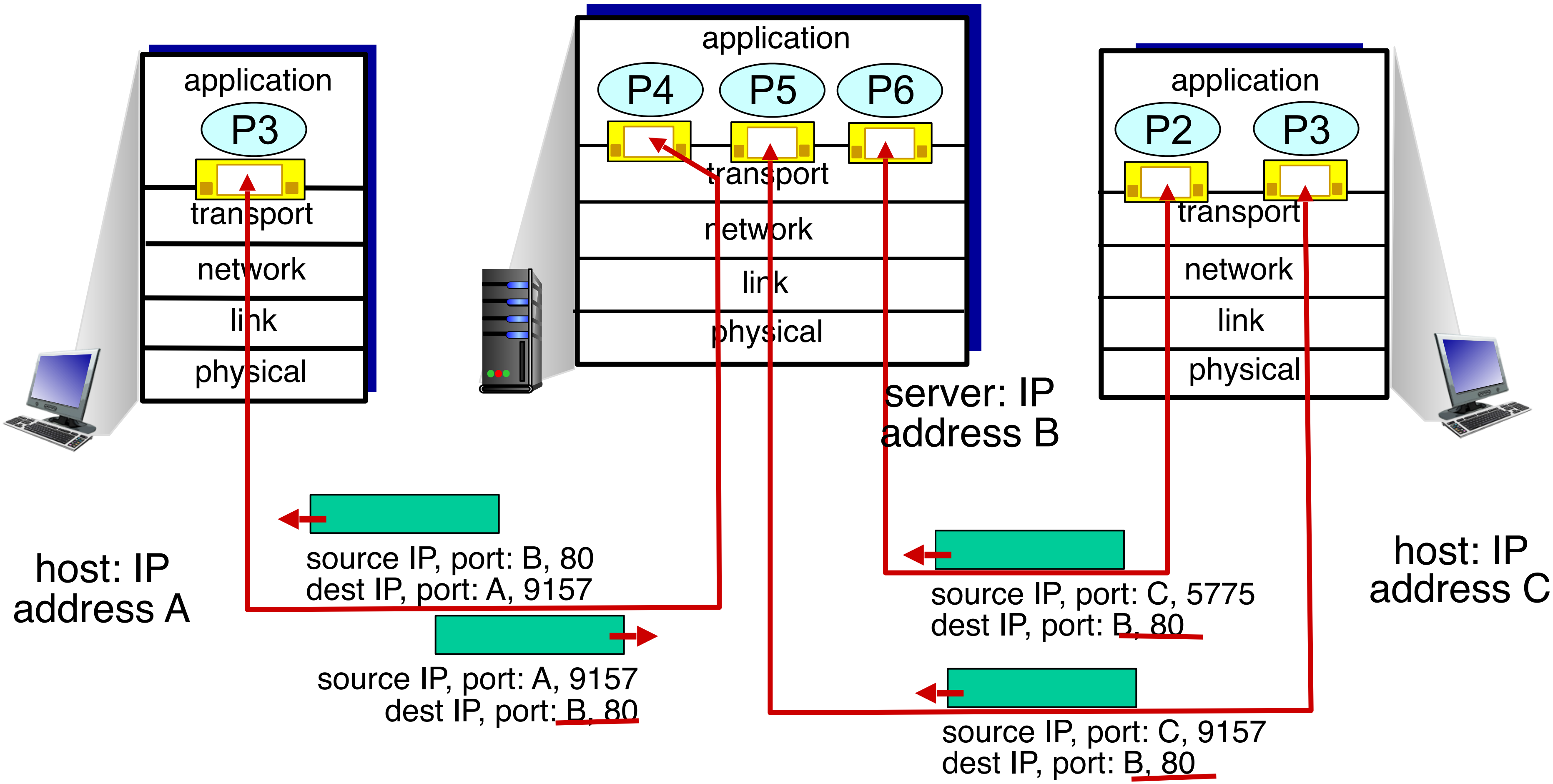


Figure 3.5 ♦ Two clients, using the same destination port number (80) to communicate with the same Web server application

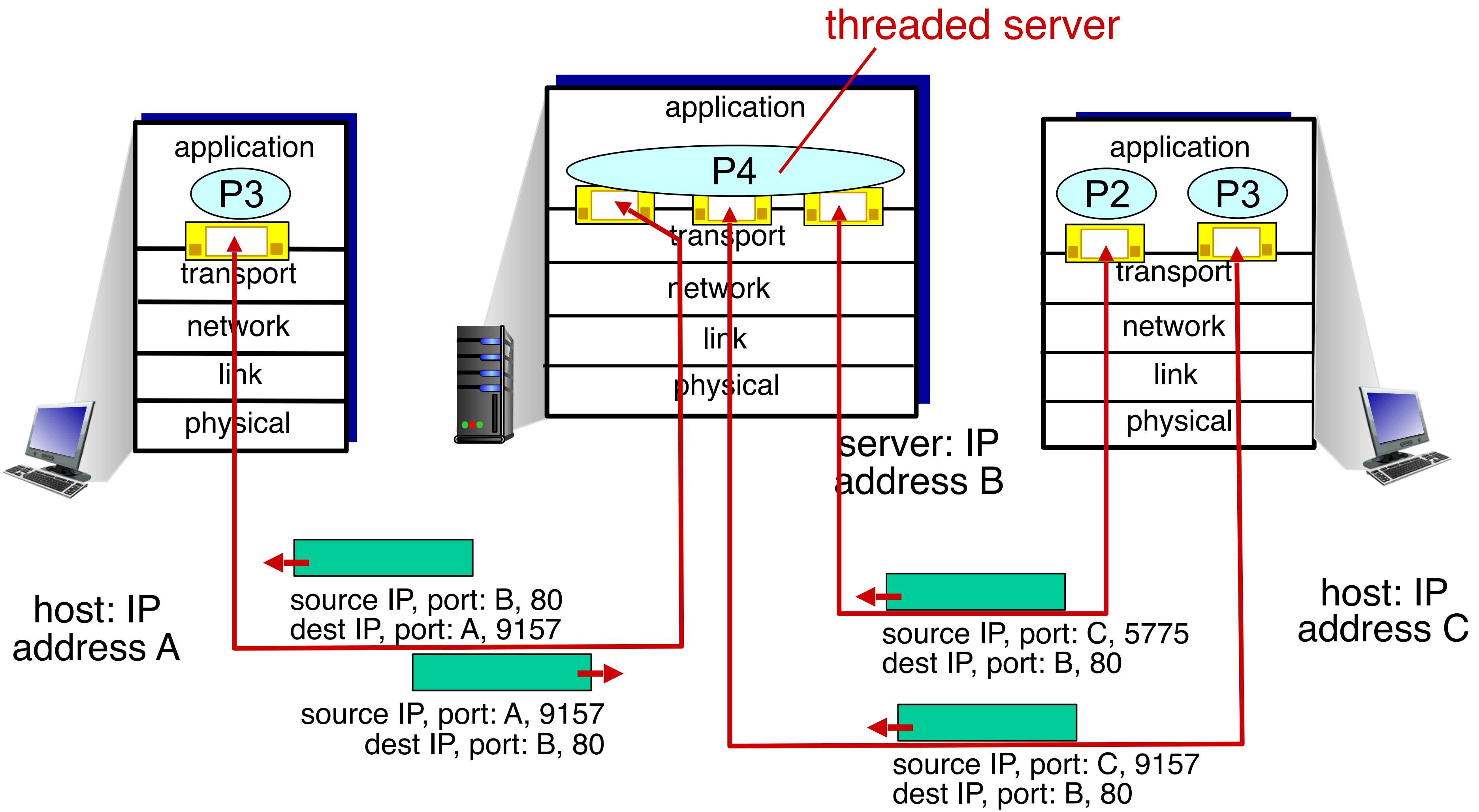
Connection-oriented demux: example

- Web server spawns a new process for each new client connection



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

Connection-oriented demux: example



- Web server creates a new thread with a new connection socket for each new client connection

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

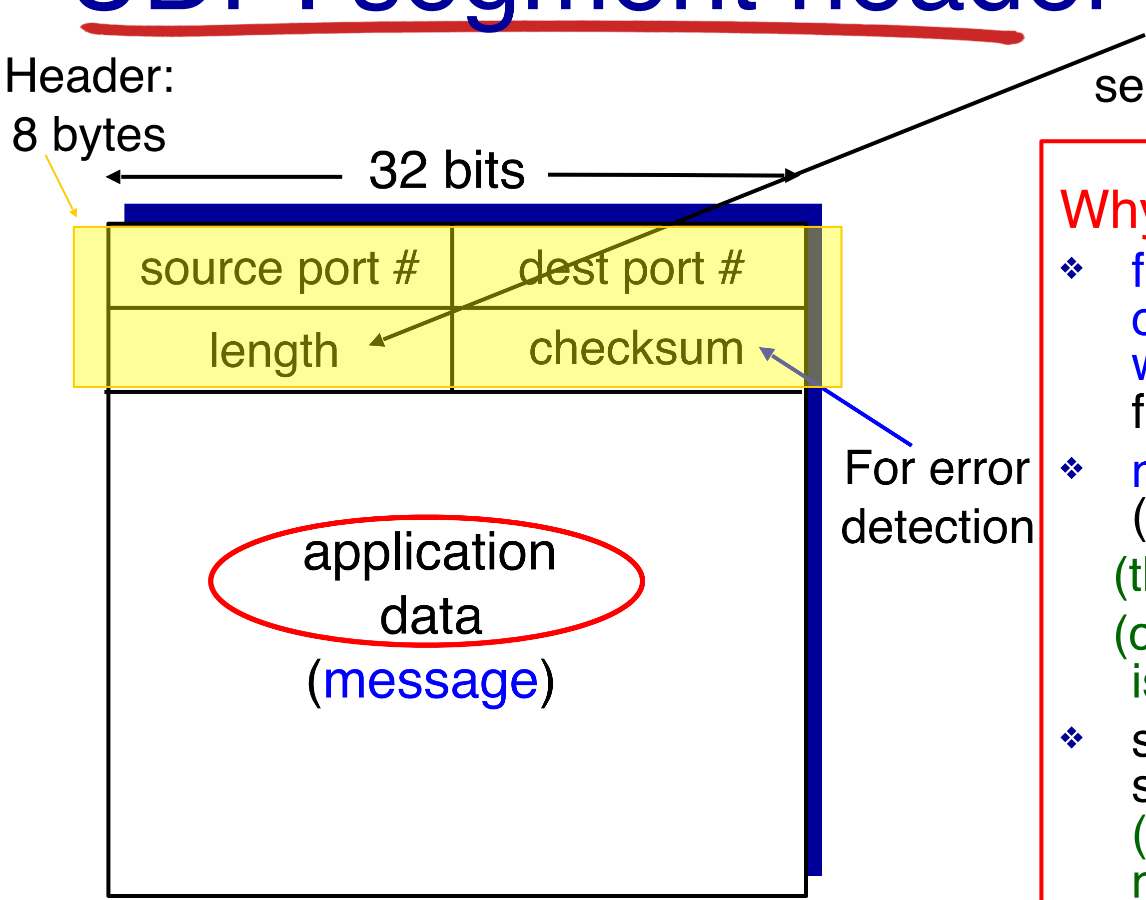
- UDP provides more than IP
  - Multiplexing/demultiplexing
  - Error checking

UDP: User Datagram Protocol [RFC 768]

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

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

UDP: segment header



UDP segment structure

Why is there a UDP?

- finer application-level control over what data is sent, and when: UDP can blast away as fast as desired
- no connection establishment (which can add delay)
  (this is why DNS use UDP)
  (connection-establishment delay is a major contribution to delay)
- simple: no connection state at sender, receiver
  (buffer, sequence and ack number.....)
  (an AP running on UDP can support more connections)
- small packet header overhead
  (TCP: 20 bytes of overhead)
  (UDP: 8 bytes of overhead)

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

Figure 3.6 ♦ Popular Internet applications and their underlying transport protocols

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment on an end-to-end basis

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement of the sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- Add all 16-bit words, including the checksum
- If the sum is 1111111111111111 → no error
  - *But maybe errors nonetheless?* More later ....
- Only error detection, no error correction, may
  - discard the damaged segment
  - give a warn to the app

Internet checksum: example

example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

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

UDP checksum

- Why UDP provides a checksum?
  - Although some link protocols (ex. Ethernet) provide error checking, it's no guarantee that all links between source and destination provide error checking
  - Some error may happen in storing in a router's memory
- UDP provides error detection on an end-to-end basis
  - Only error detection, no error correction, may
    - discard the damaged segment
    - give a warn to the app

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

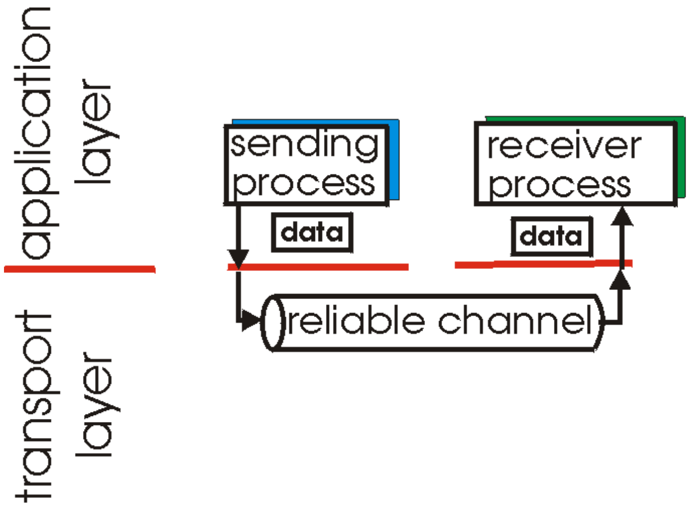
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

Principles of reliable data transfer

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

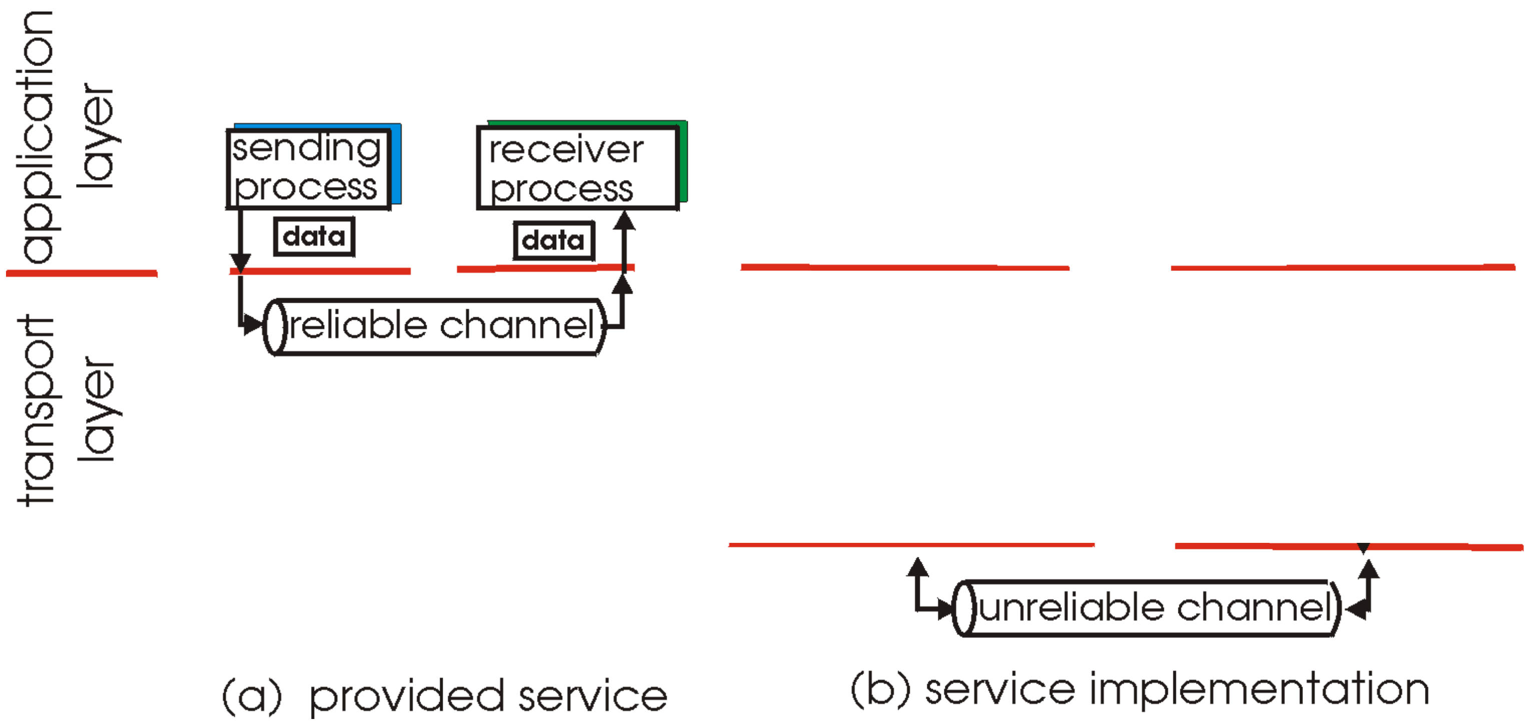


(a) provided service

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

Principles of reliable data transfer

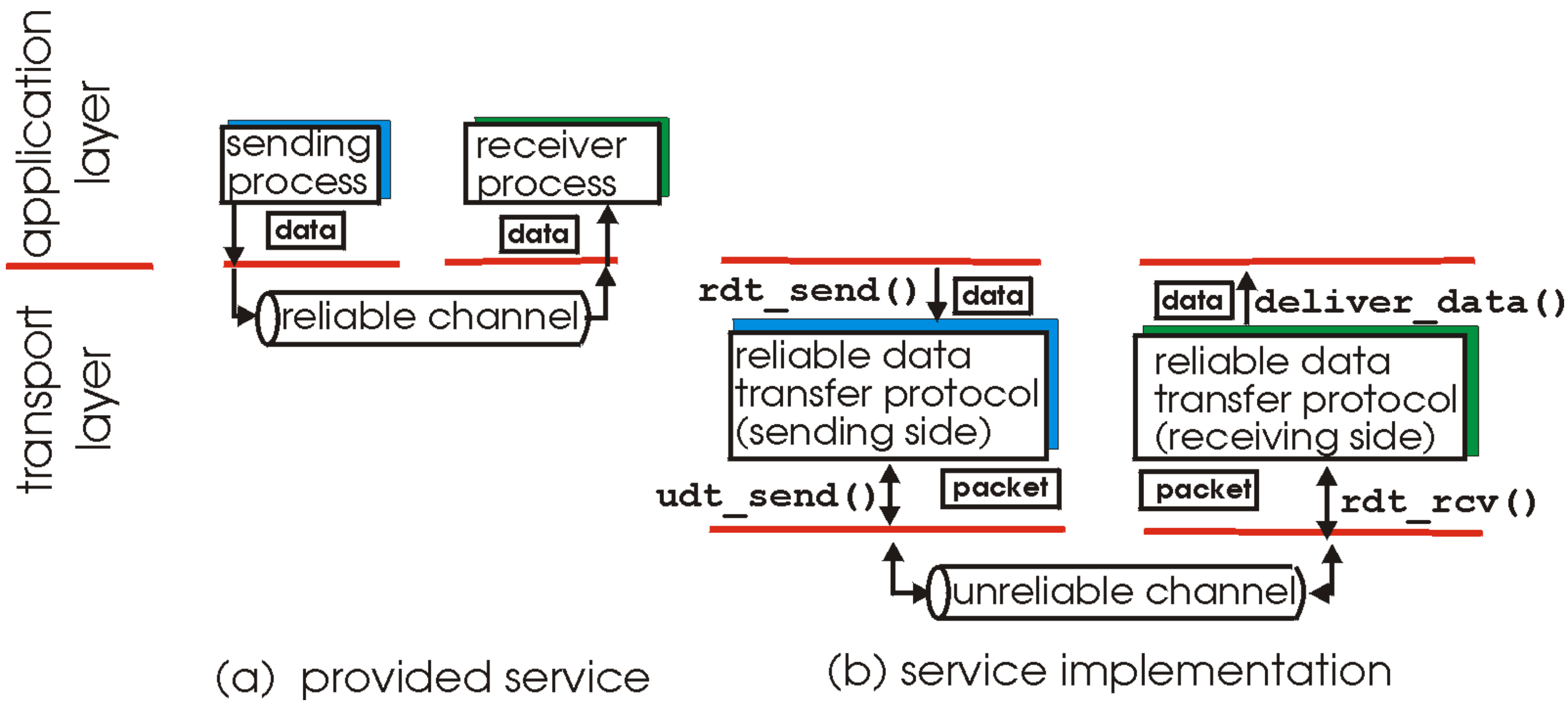
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

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

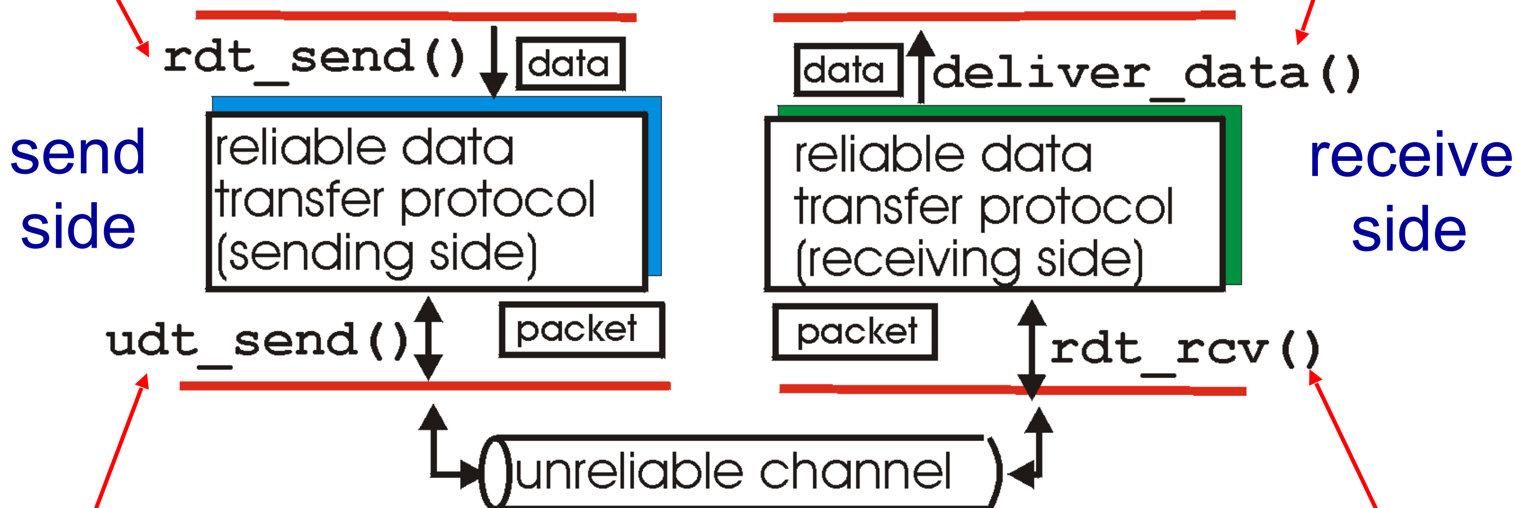


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

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

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



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

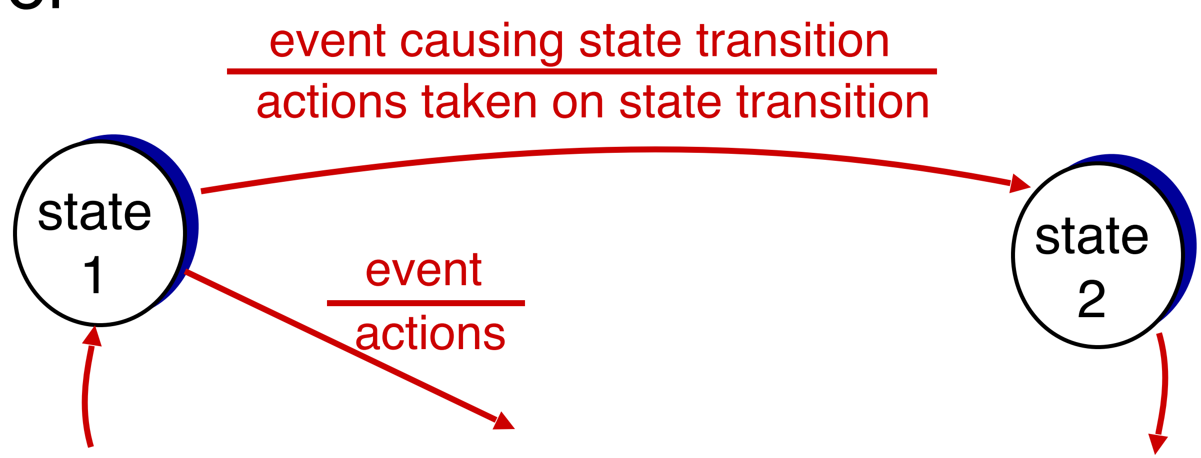
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

we'll:

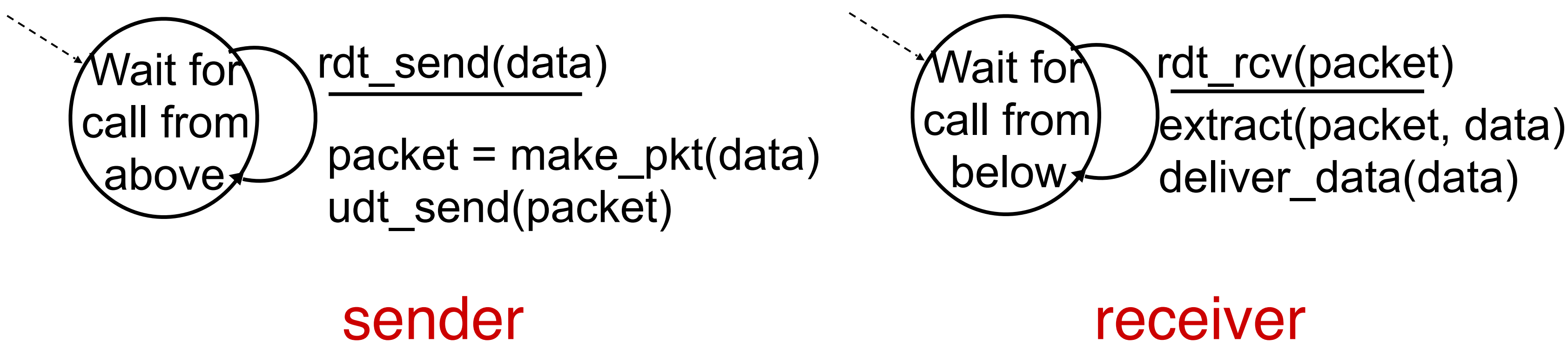
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite-state machine (FSM) to specify sender, receiver

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 reads data from underlying channel



rdt2.0: channel with bit errors

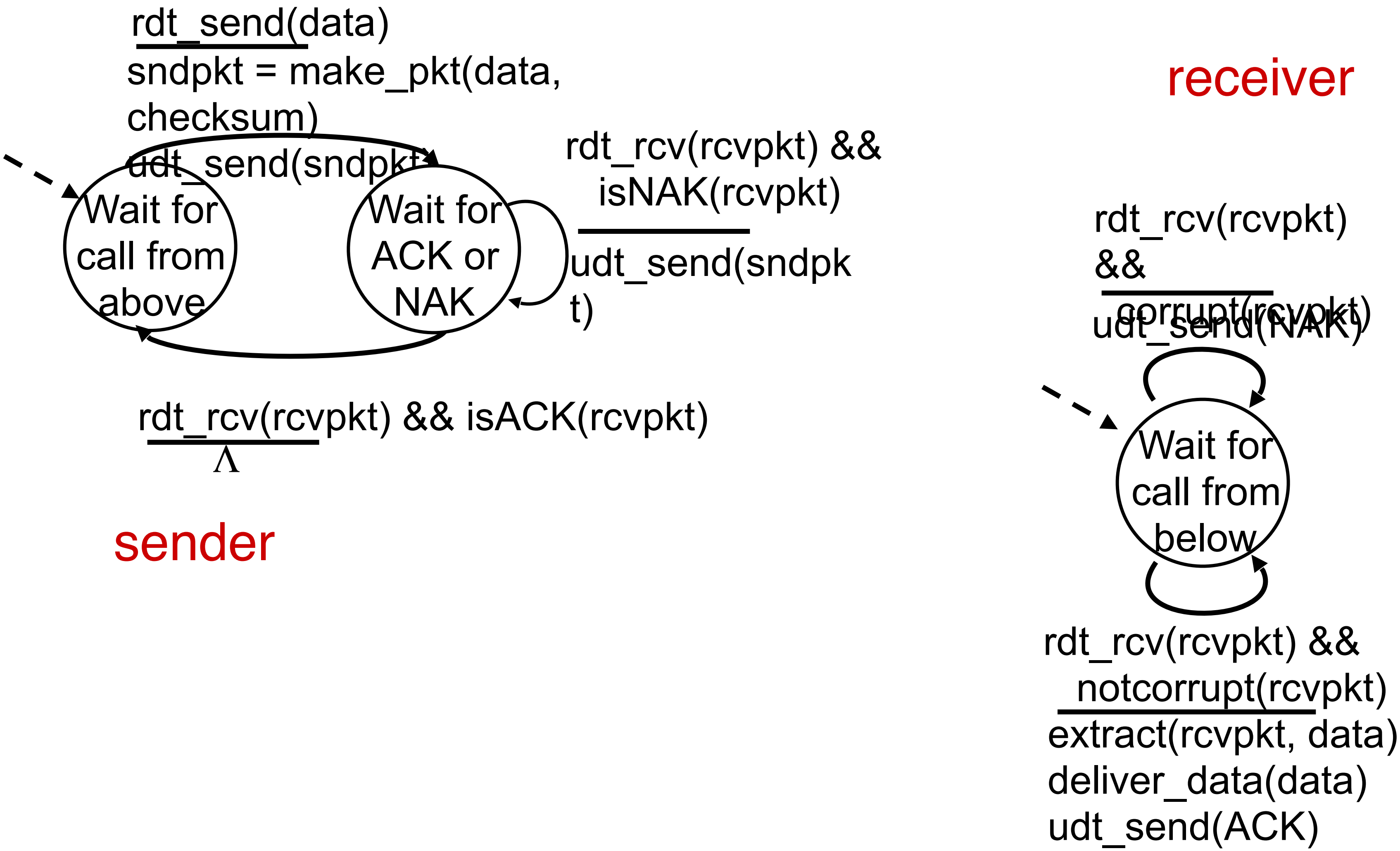
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

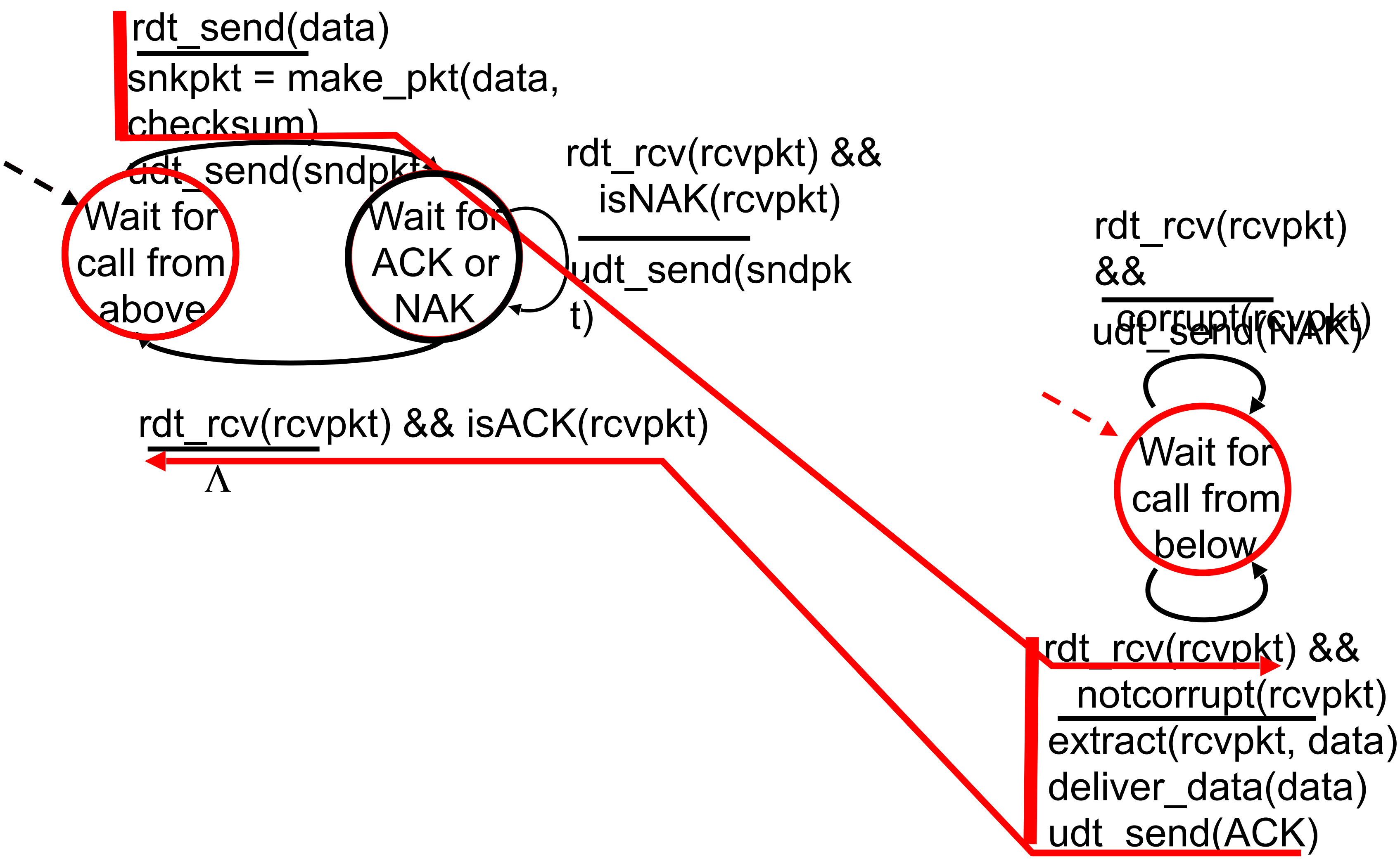
rdt2.0: channel with bit errors

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

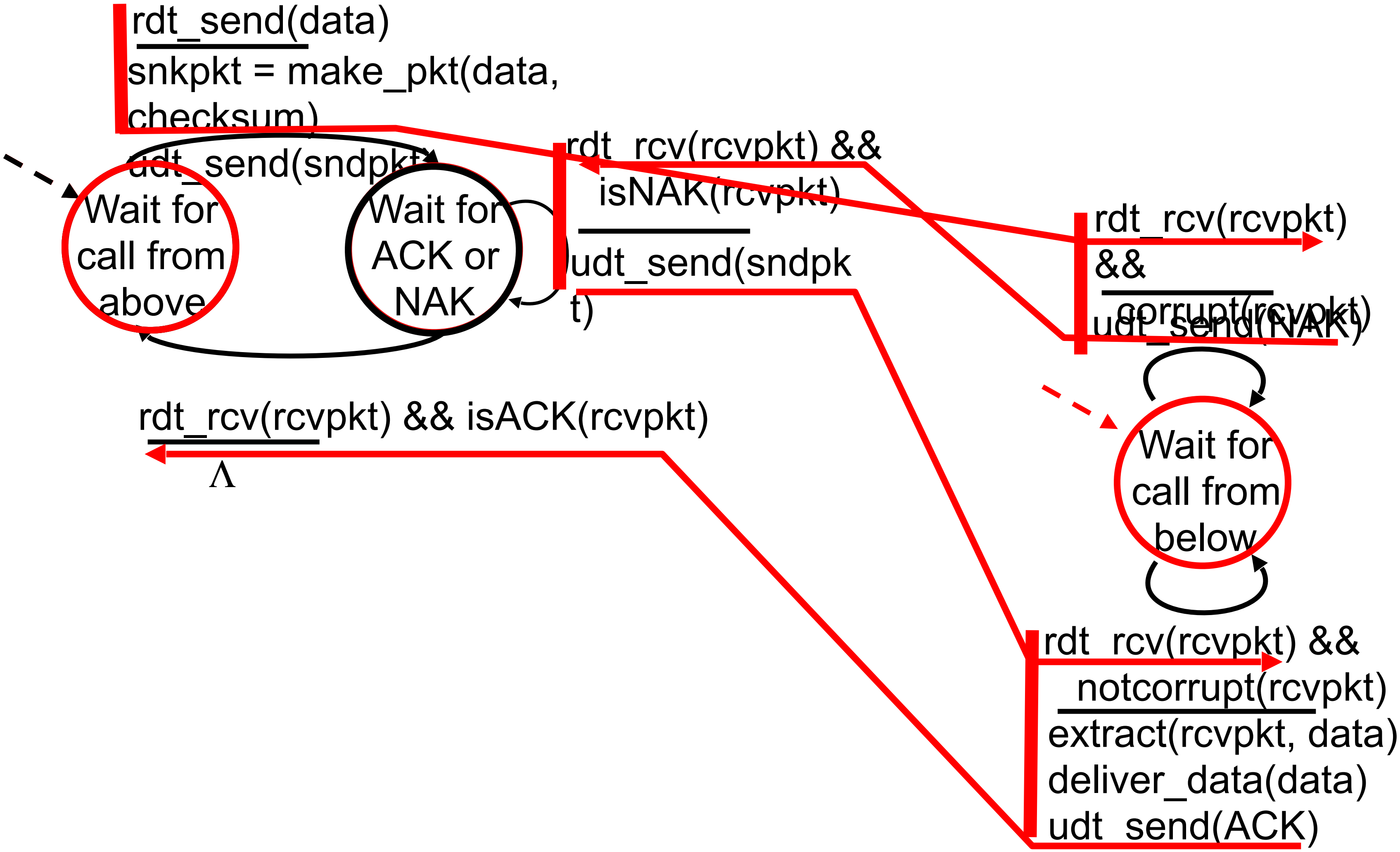
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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

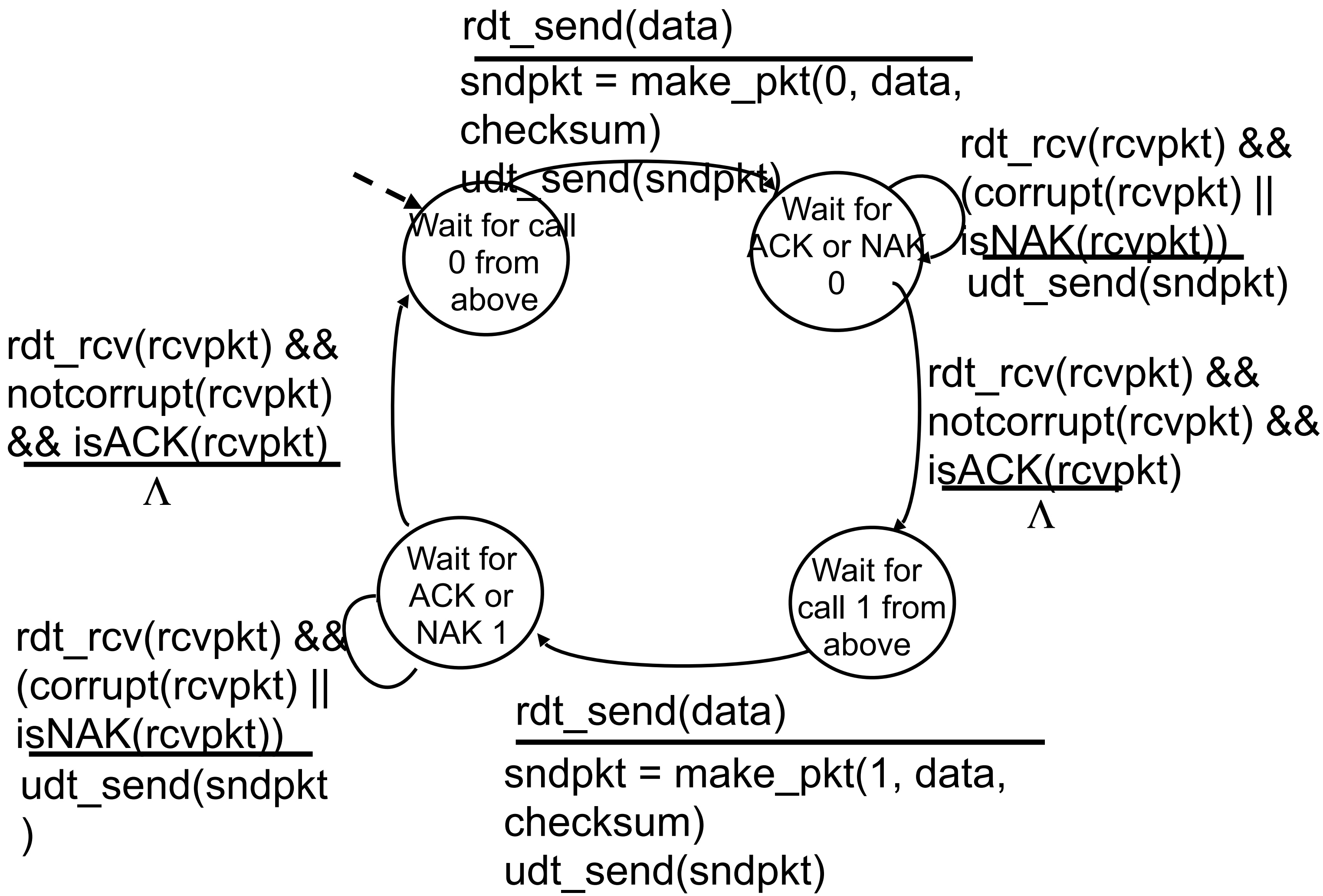
handling duplicates:

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

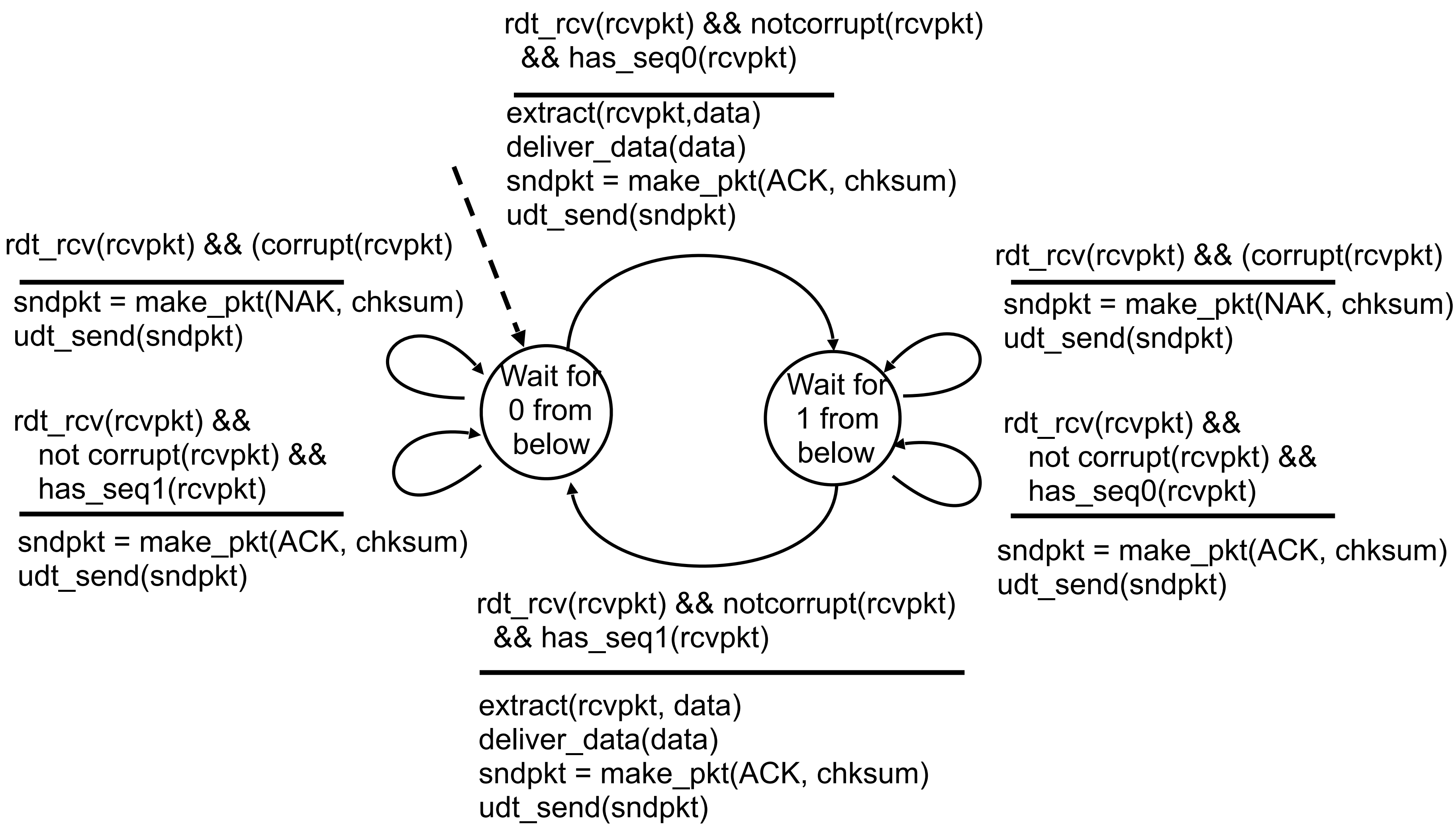
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: 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 "expected" pkt should have seq. # of 0 or 1

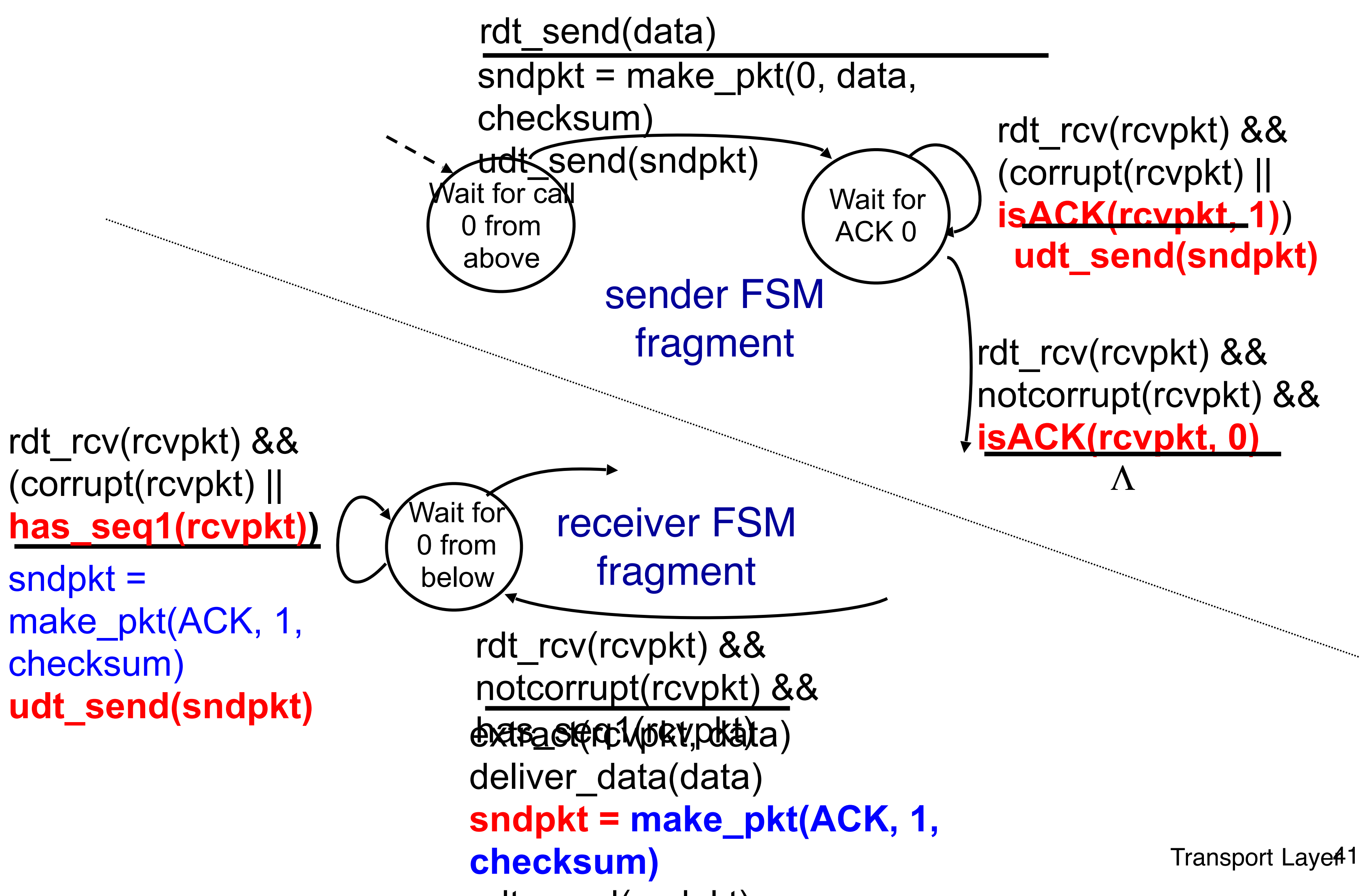
receiver:

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

rdt2.2: a NAK-free protocol

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

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

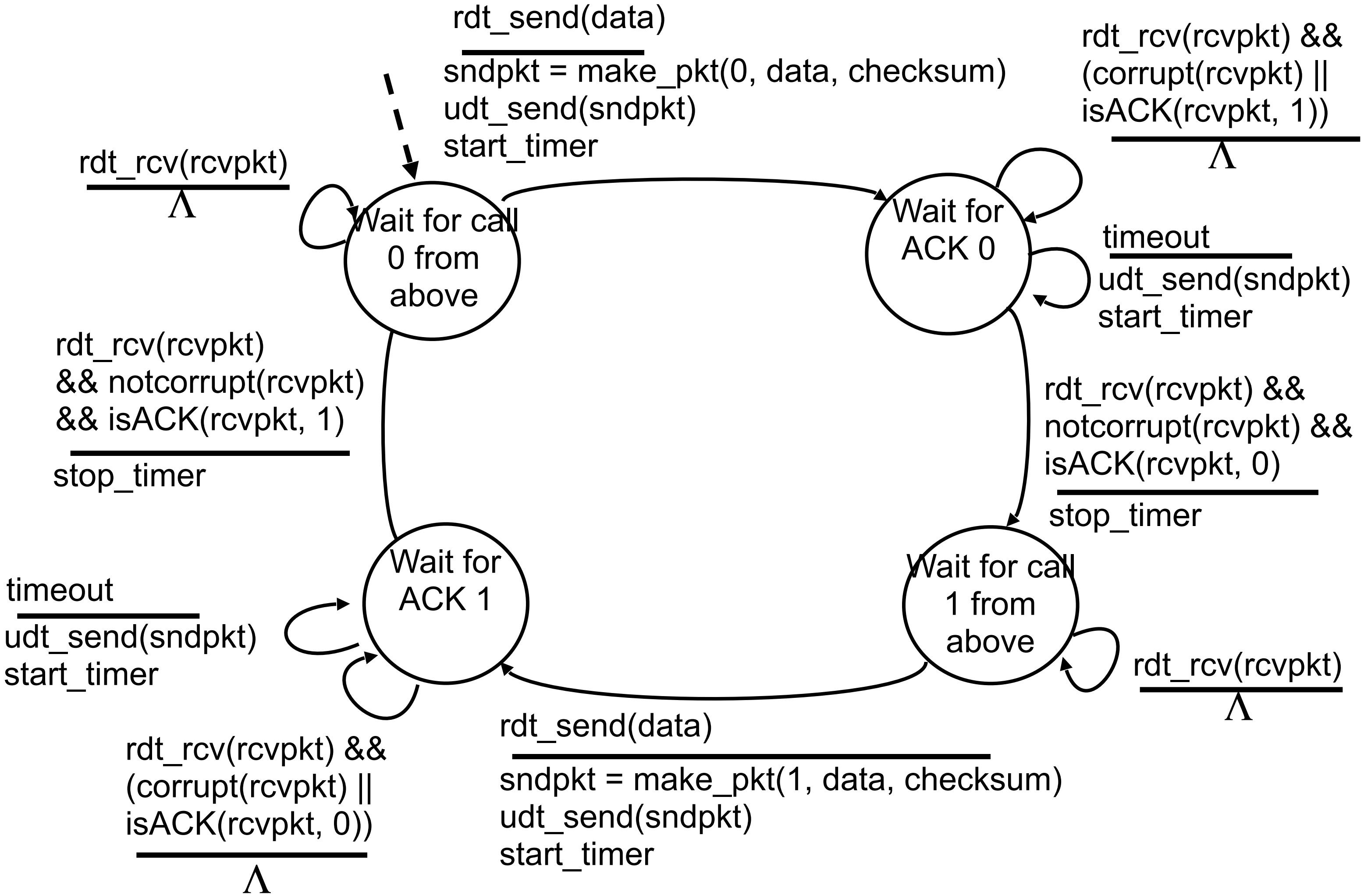
new assumption: underlying channel can also lose packets (data, ACKs)

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

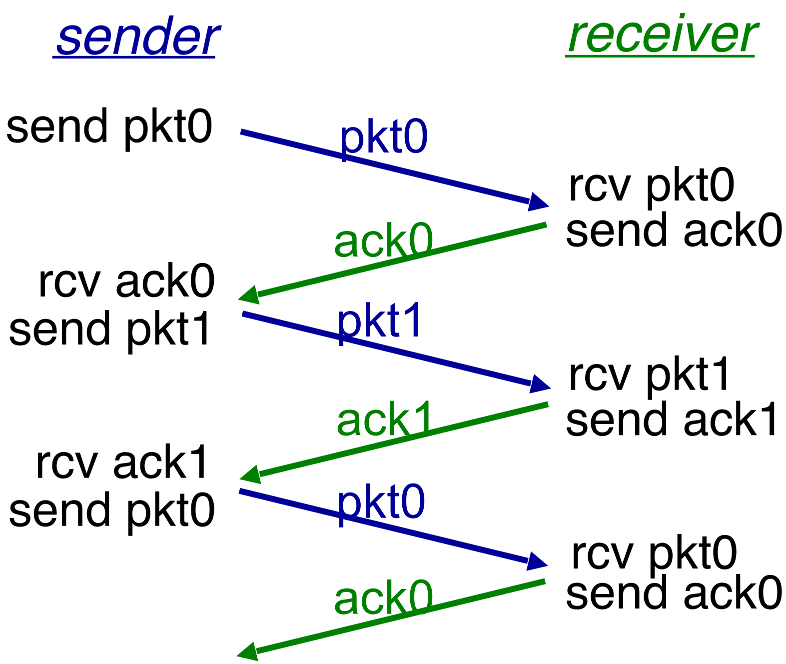
approach: sender waits "reasonable" amount of time for ACK

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

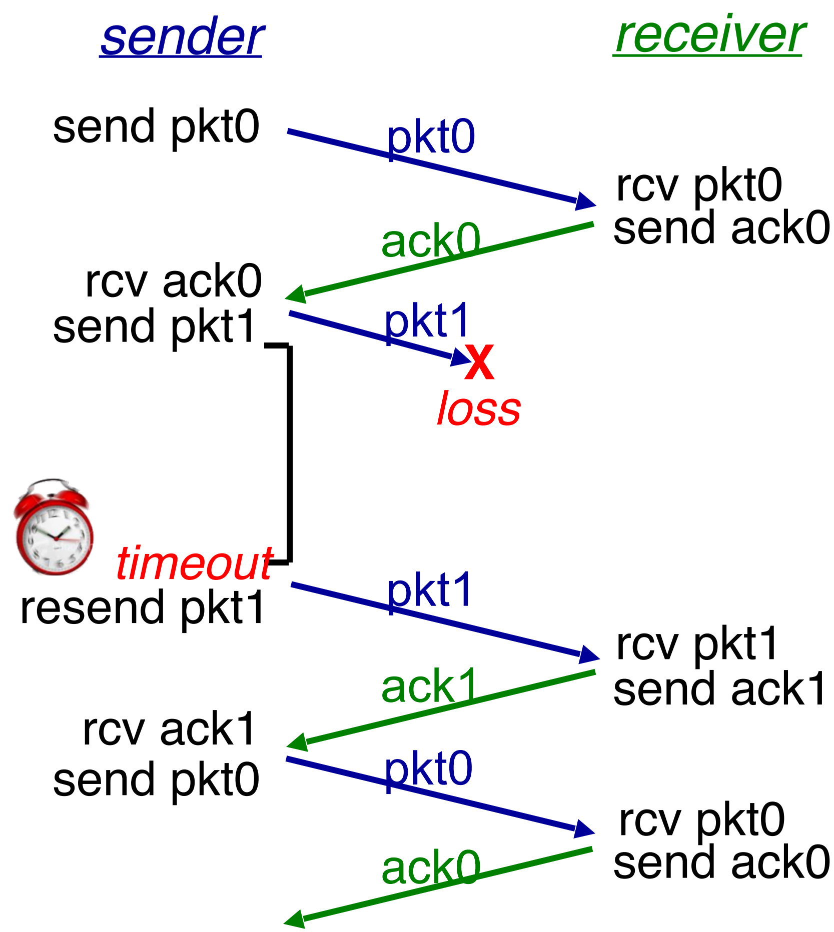
rdt3.0 sender



rdt3.0 in action

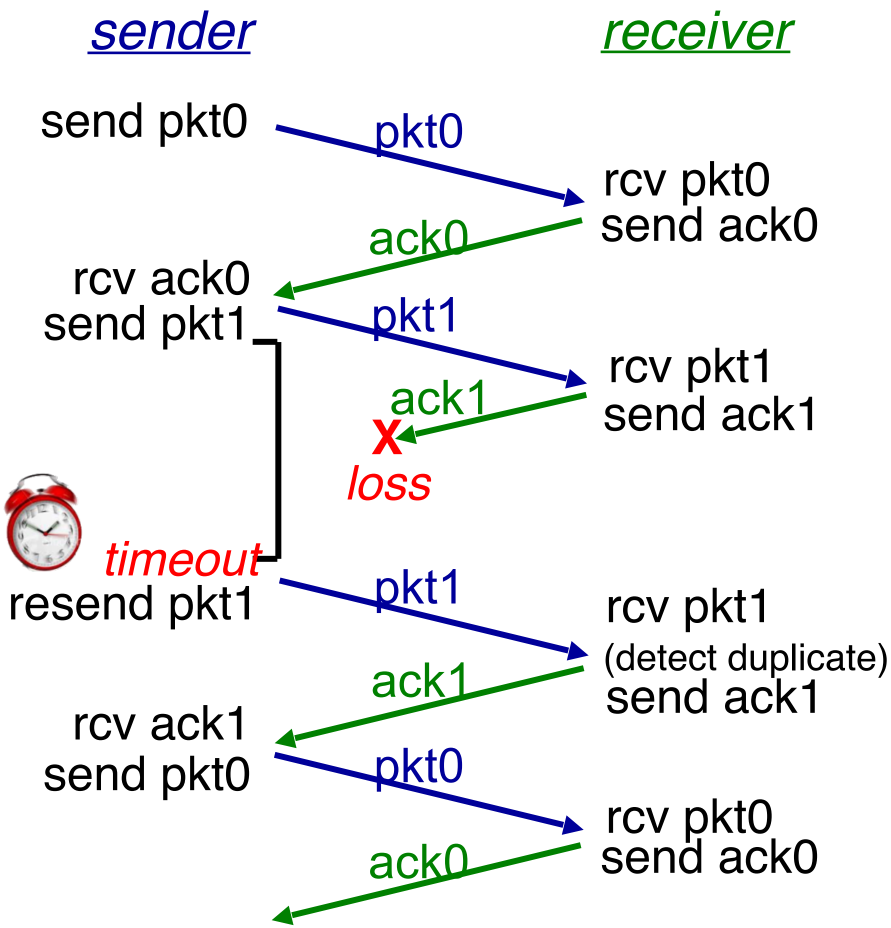


(a) Operation with no loss

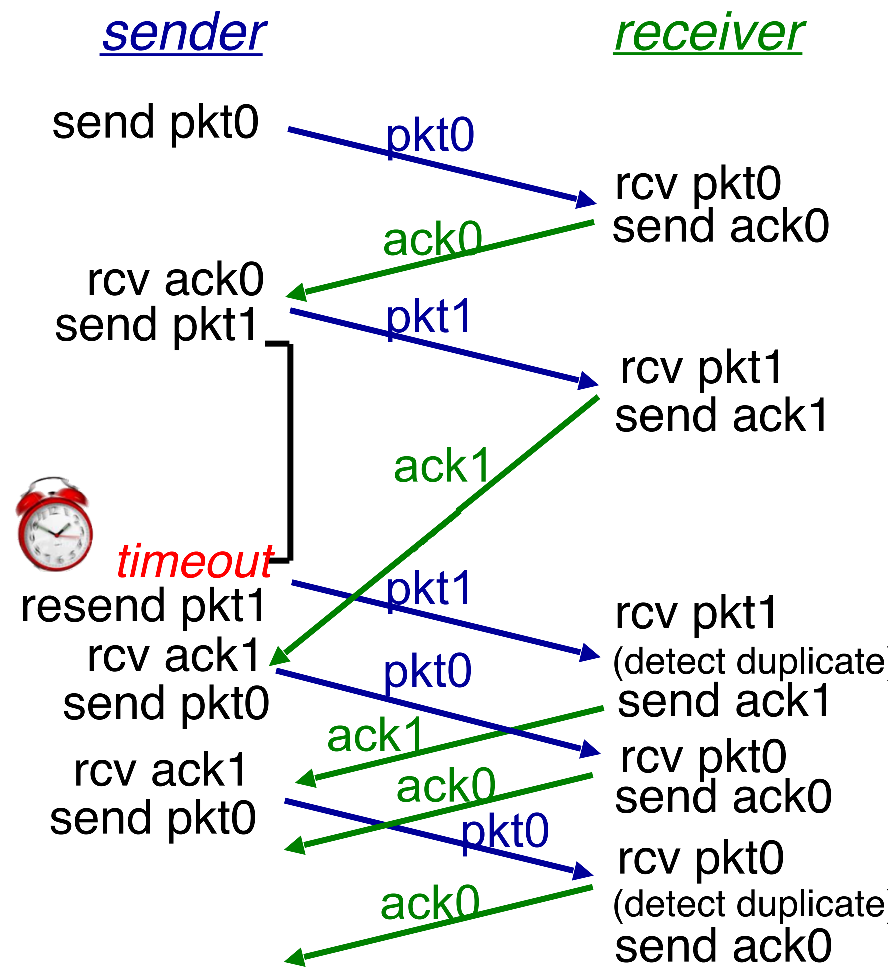


(b) Lost packet

rdt3.0 in action



(c) Lost ACK



(d) Premature timeout / delayed ACK

Performance of rdt3.0

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

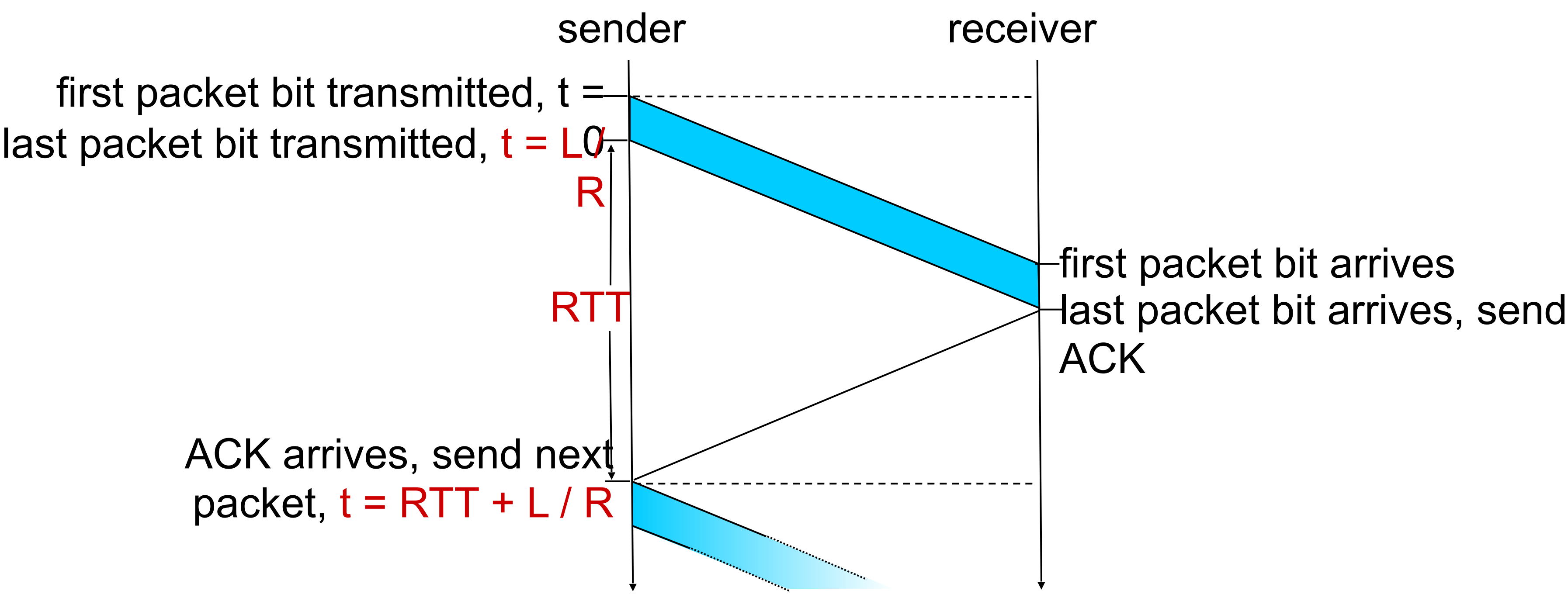
$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

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

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

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

rdt3.0: stop-and-wait operation

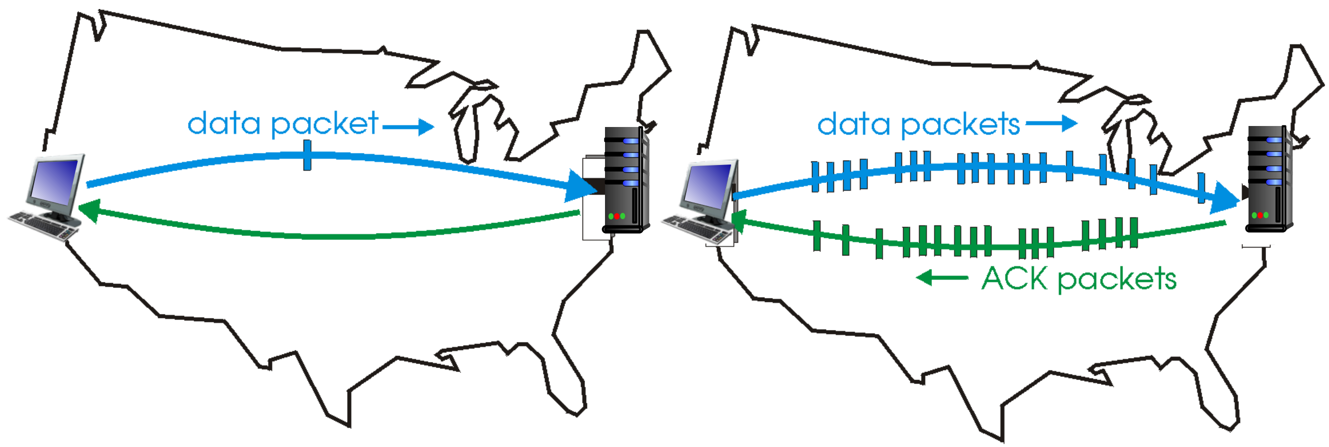


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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

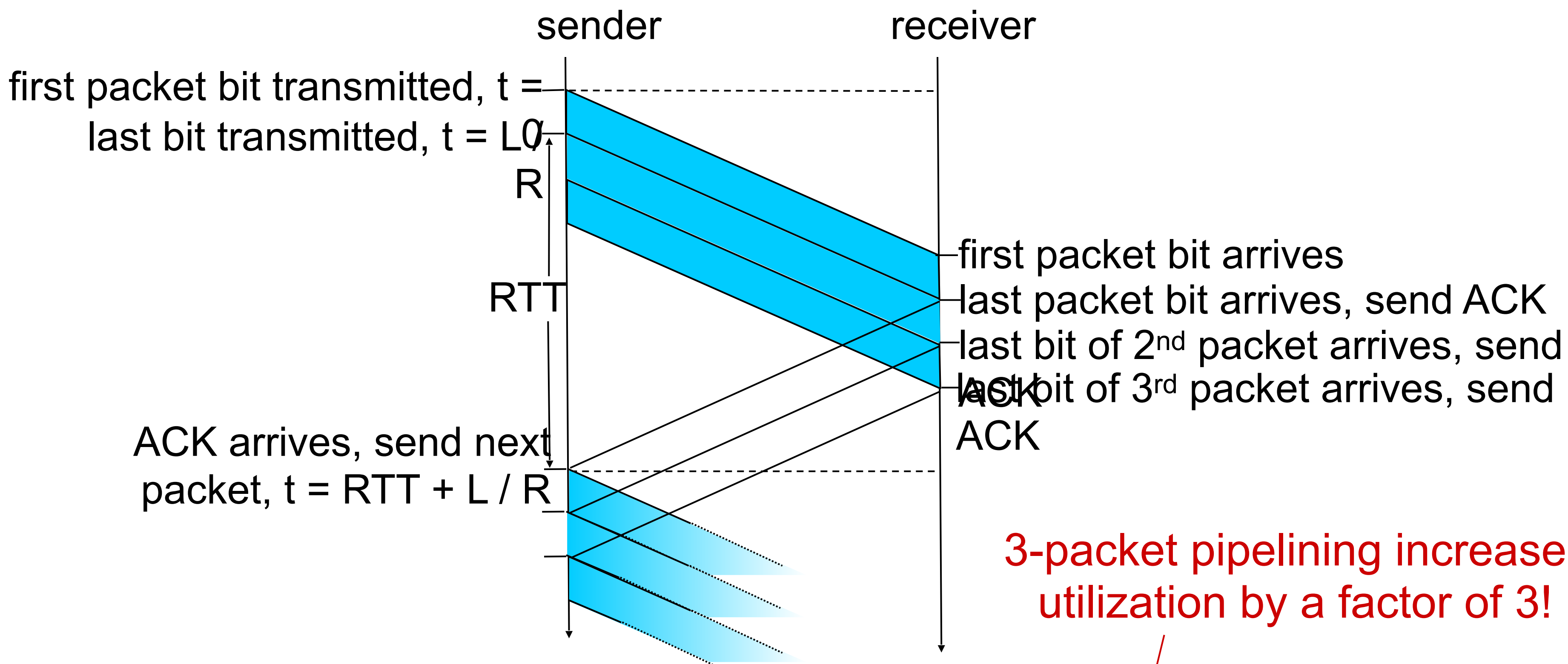


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization



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

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

Pipelined protocols: overview

Go-back-N (GBN):

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

Selective Repeat (SR):

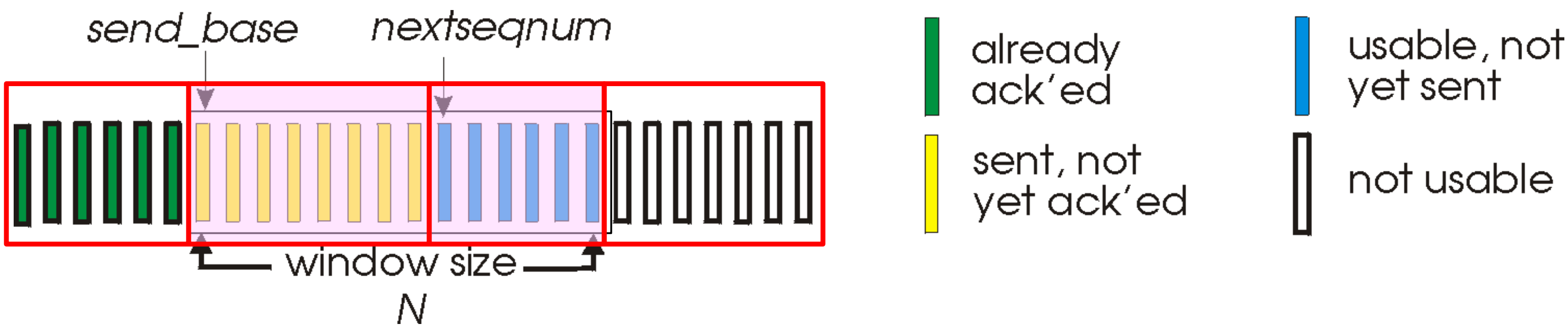
- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N: sender

•Sliding-window protocol

•Why need limit the window size?
→Flow control and congestion control

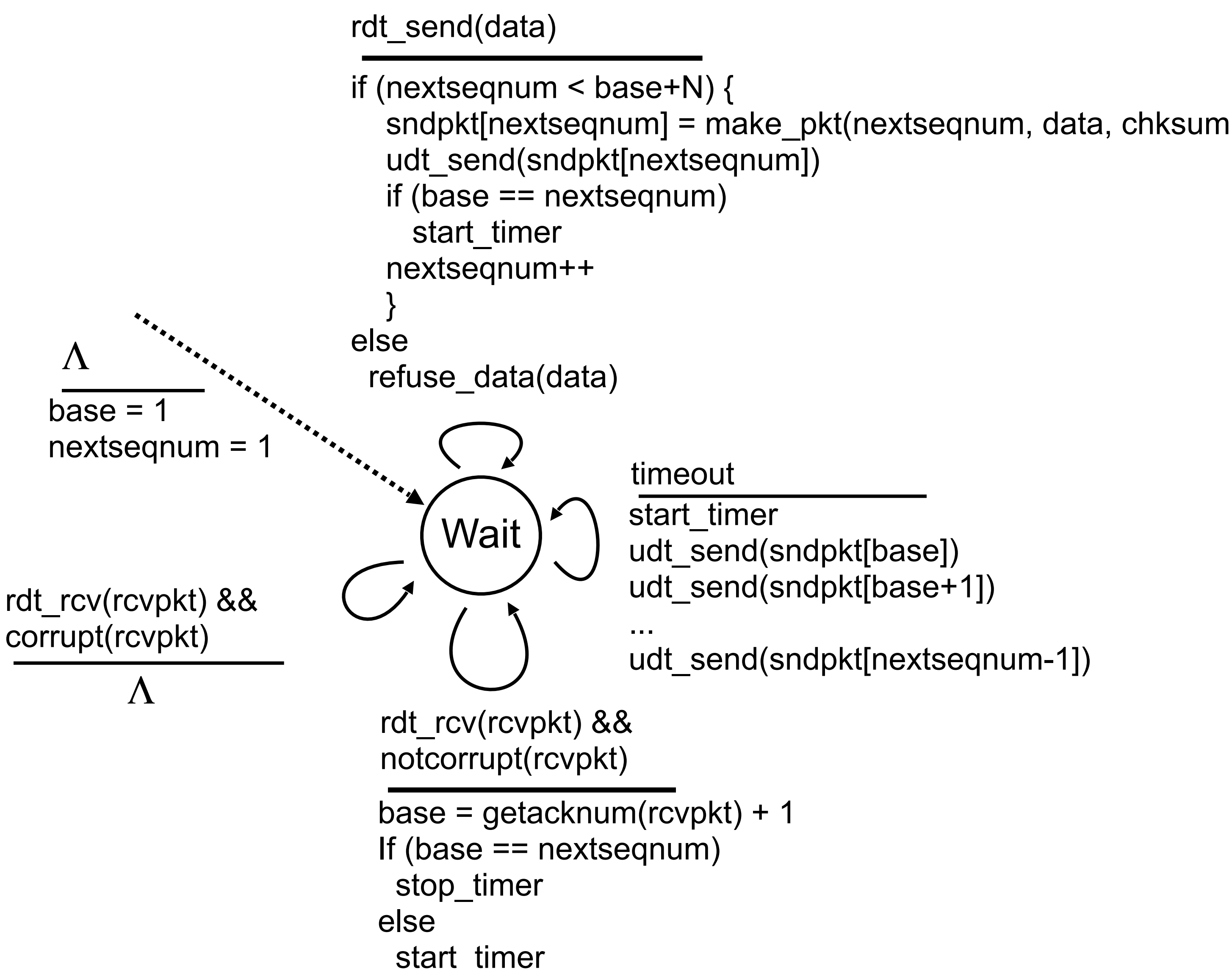
- k-bit seq. # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed



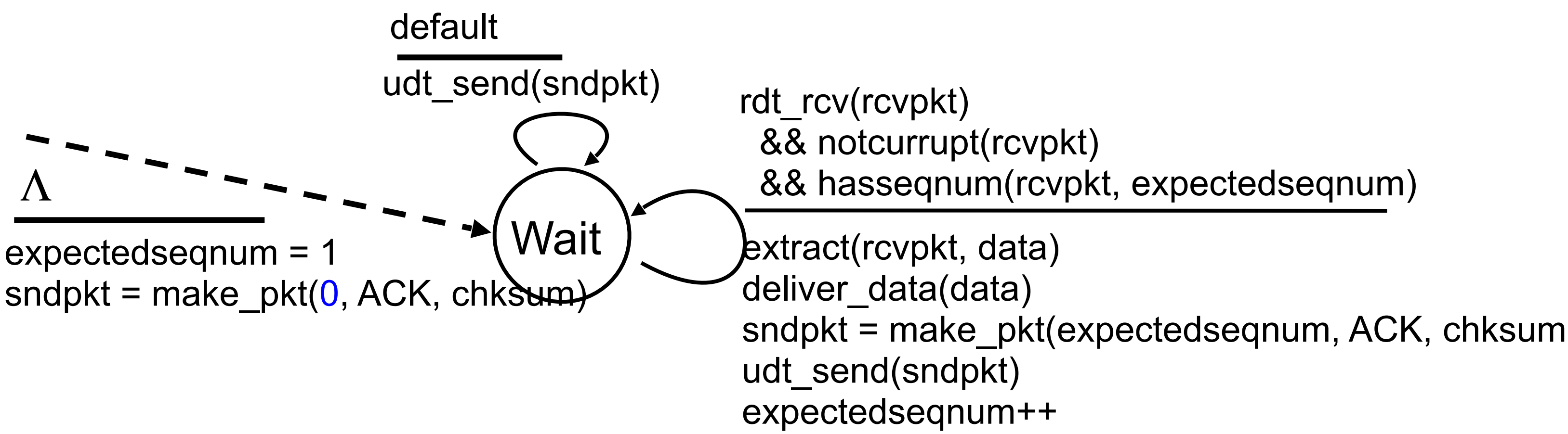
- ACK(n): ACKs all pkts up to, including seq. # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- use only a single timer (for oldest in-flight pkt)
- *timeout(n):* retransmit packet n and all higher seq. # pkts in window

•Sequence number is carried in a fixed-length field in the packet header; ex. 32 bits in TCP, the range of sequence number is $[0, 2^{32}-1]$

GBN: sender extended FSM



GBN: receiver extended FSM

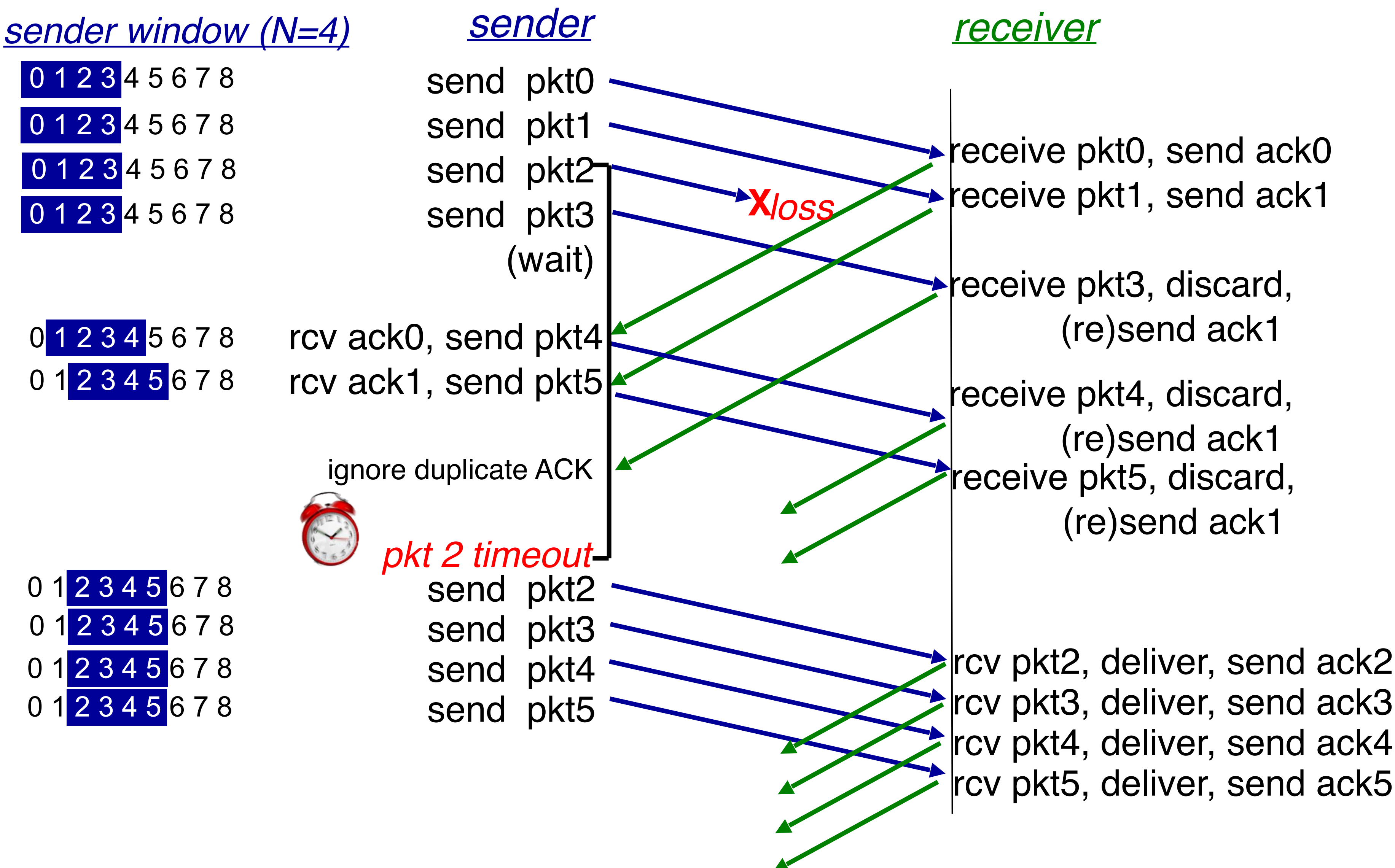


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq. #

- 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



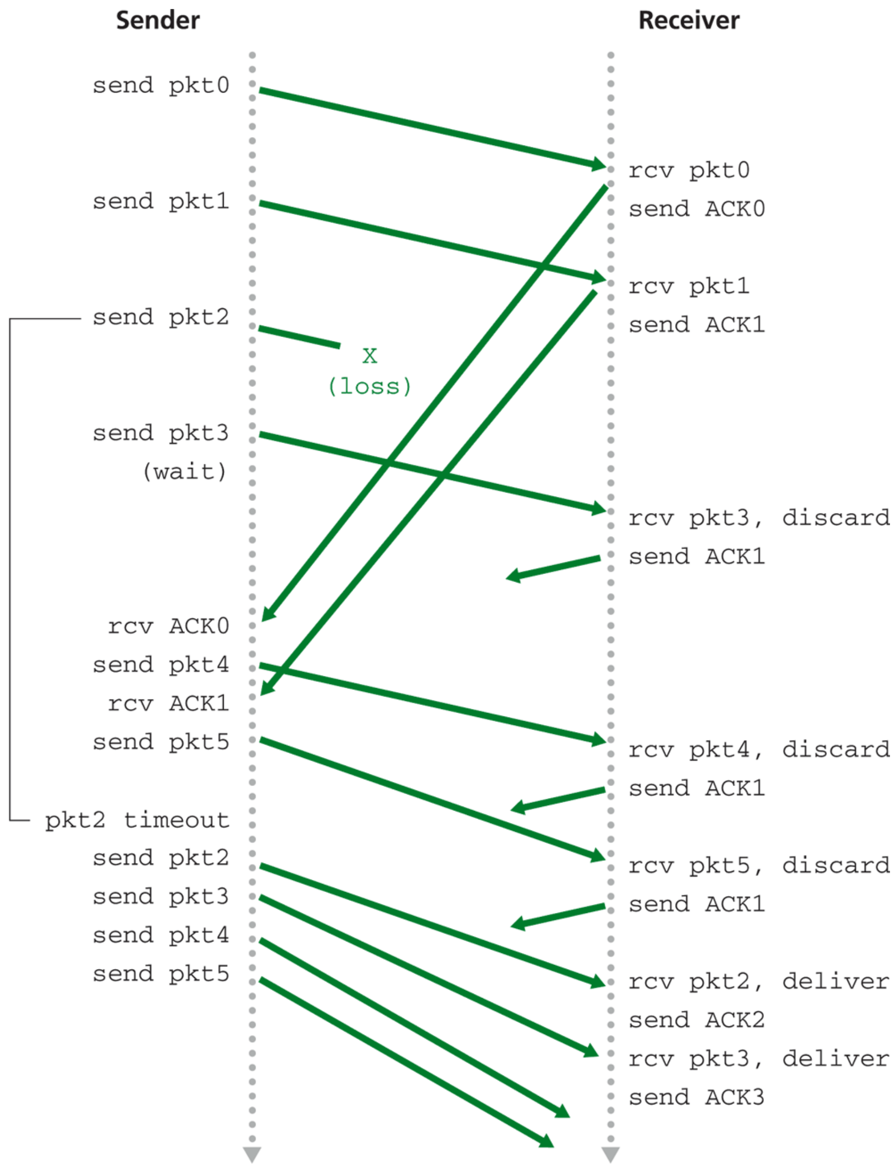
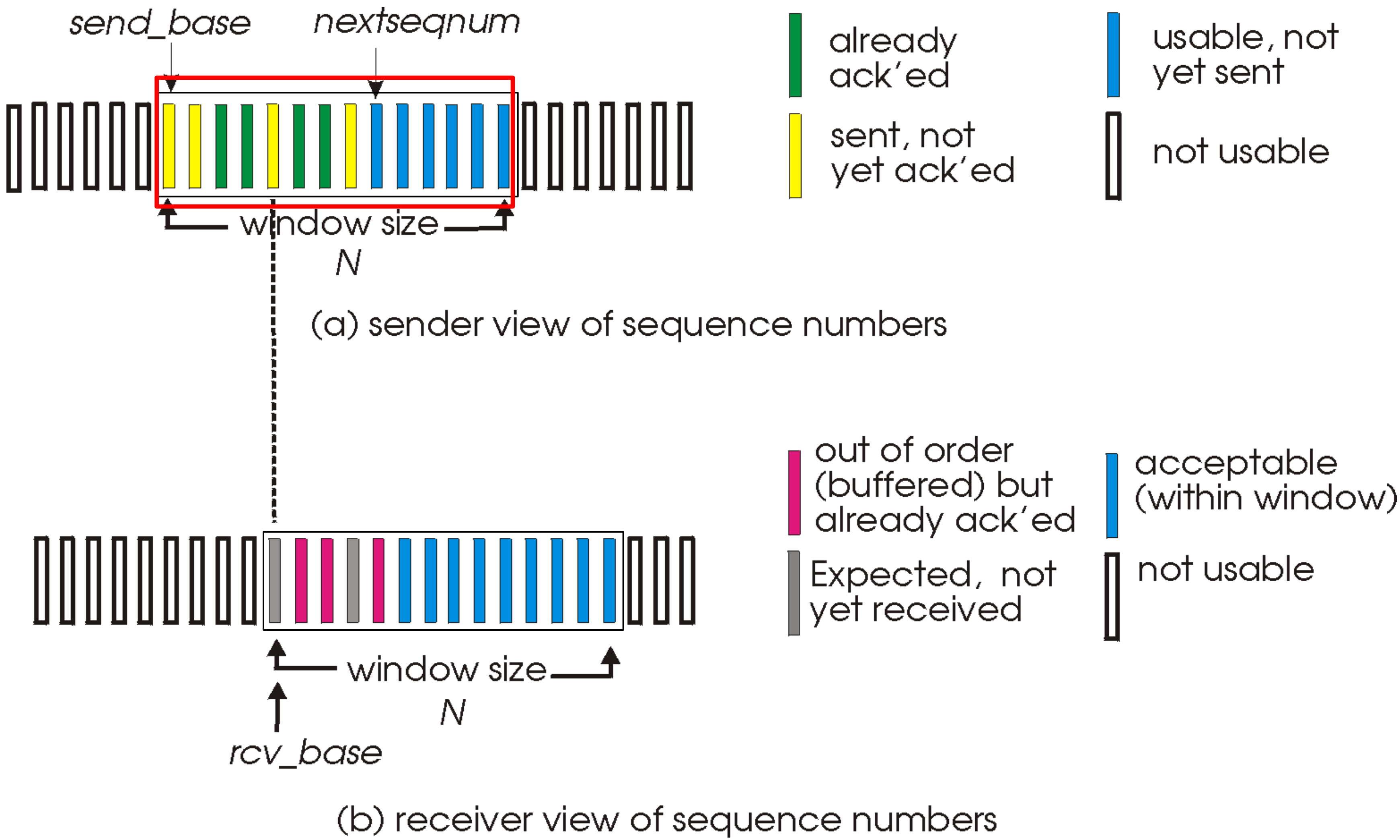


Figure 3.22 ♦ Go-Back-N in operation

Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - *N* consecutive seq. #s
  - limits seq. #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



•The sender and the receiver will not always have an identical view of sliding window

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 is equal to 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

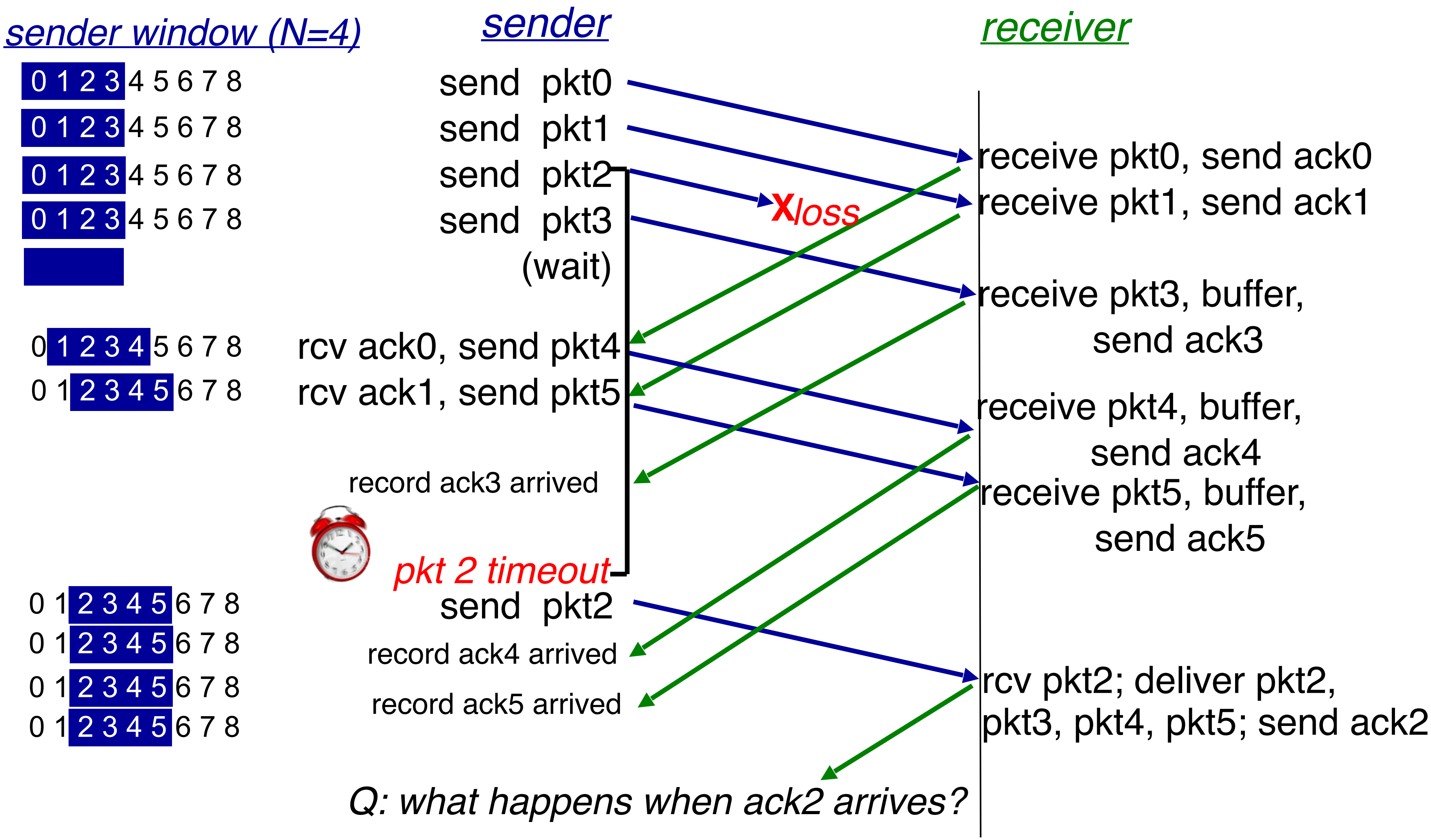
pkt n in [rcvbase-N, rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



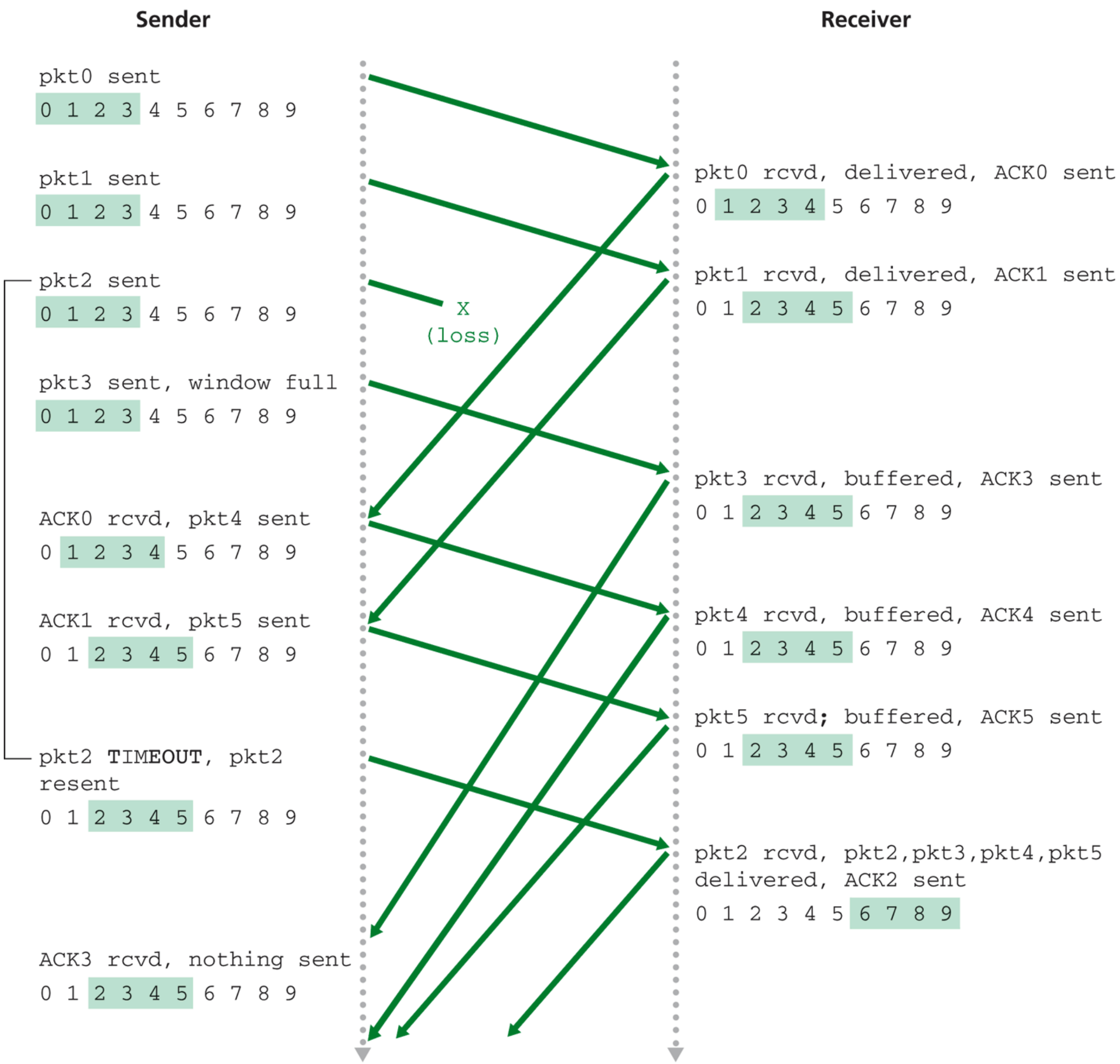


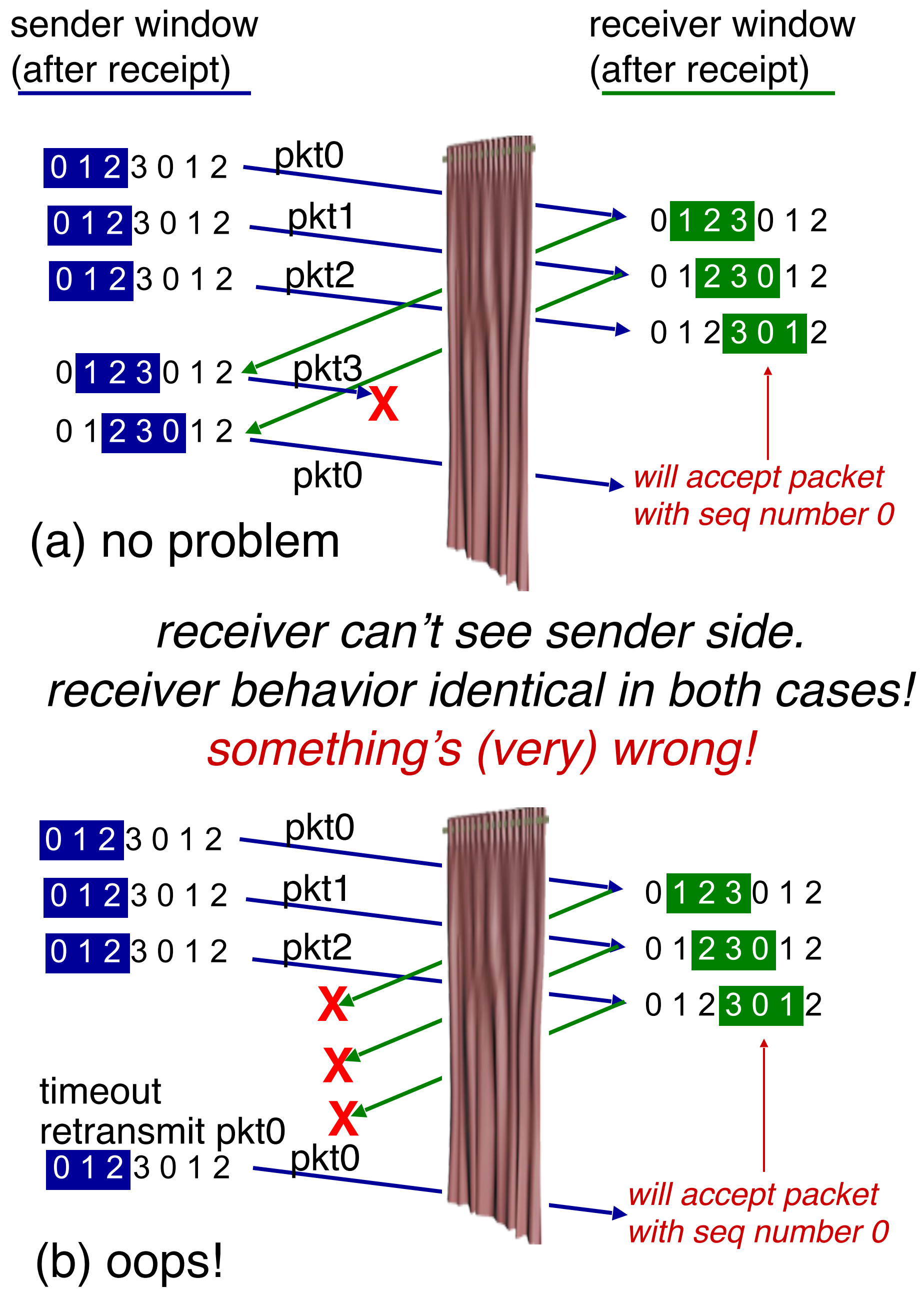
Figure 3.26 ♦ SR operation

Selective repeat: dilemma

example:

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

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



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

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

•Point-to-point: one to one
•Multicast: one to many

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no “message boundaries”
- pipelined:
  - TCP congestion and

•MSS depends on MTU (Maximum Transmission Unit)
•MTU: the length of the link-layer frame
•Path MTU

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size (only app-layer data, not including header)
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

•Both Ethernet and PPP link-layer protocols have an MTU of 1,500 bytes; Thus a typical value of MSS is 1,460 bytes
•TCP segment plus the TCP/IP header length (typically 40 bytes)

傳送方 (Sender)

- 封包1：A B C
  - Seq. #: 1
- 封包2：D E F G
  - Seq. #: 4
- 封包3：H I J
  - Seq. #: 8

接收方 (Receiver)

UDP (Datagram):

- 封包1：A B C
- 封包2：D E F G
- 封包3：H I J

TCP (Byte Stream):

- A B C D E F G H I J
- A B C D E F G H I J
- A B C D E F G H I J
- A B C D E F G H I J

TCP segment structure

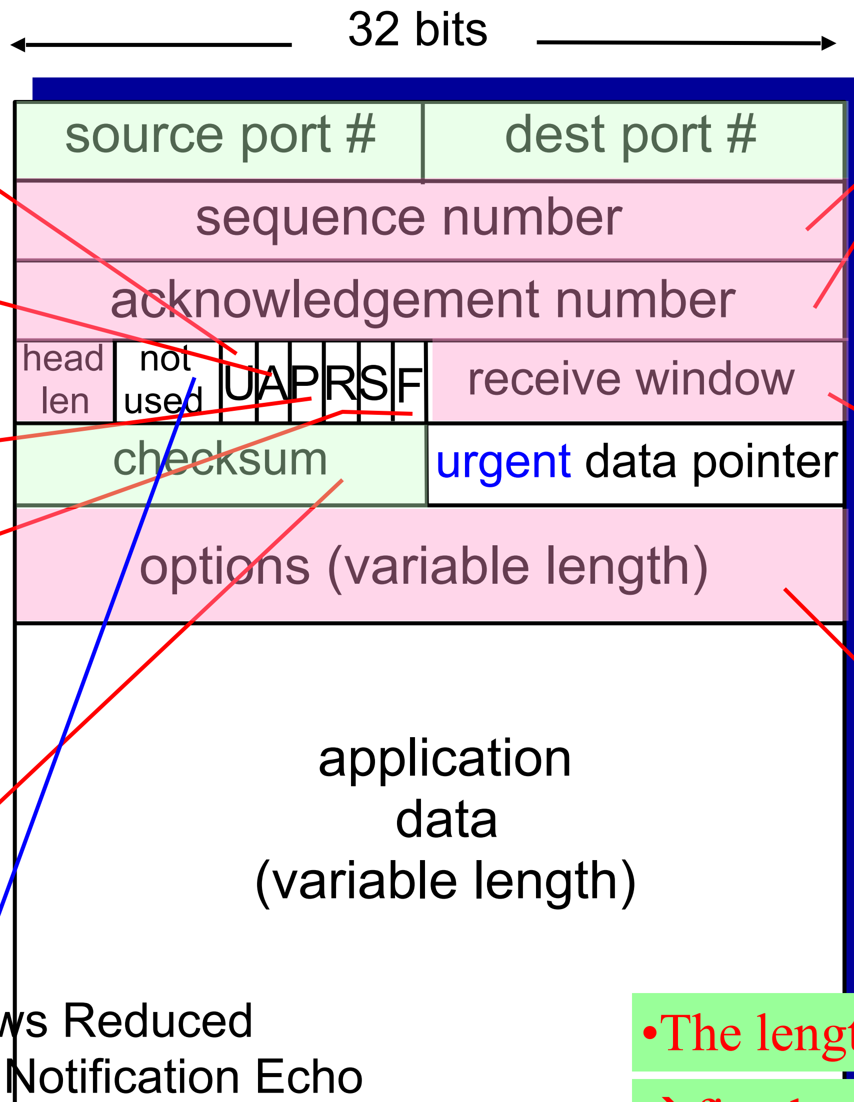
URG: urgent data
(generally not used)

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 (for
flow control)

usually empty
Sometimes used for
negotiating the MSS

CWR: Congestion Windows Reduced
ECE: Explicit Congestion Notification Echo
Explicit Congestion Notification (RFC 3168)

•The length of the TCP header?

→fixed or variable?
→typically, how long?

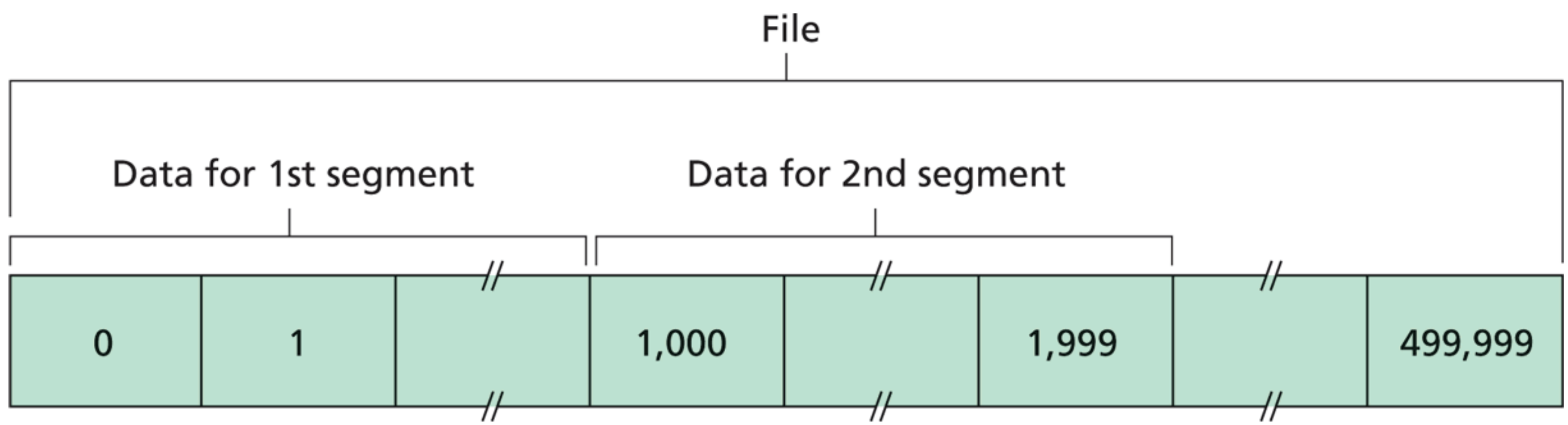


Figure 3.30 ♦ Dividing file data into TCP segments

TCP seq. numbers, ACKs

sequence numbers:

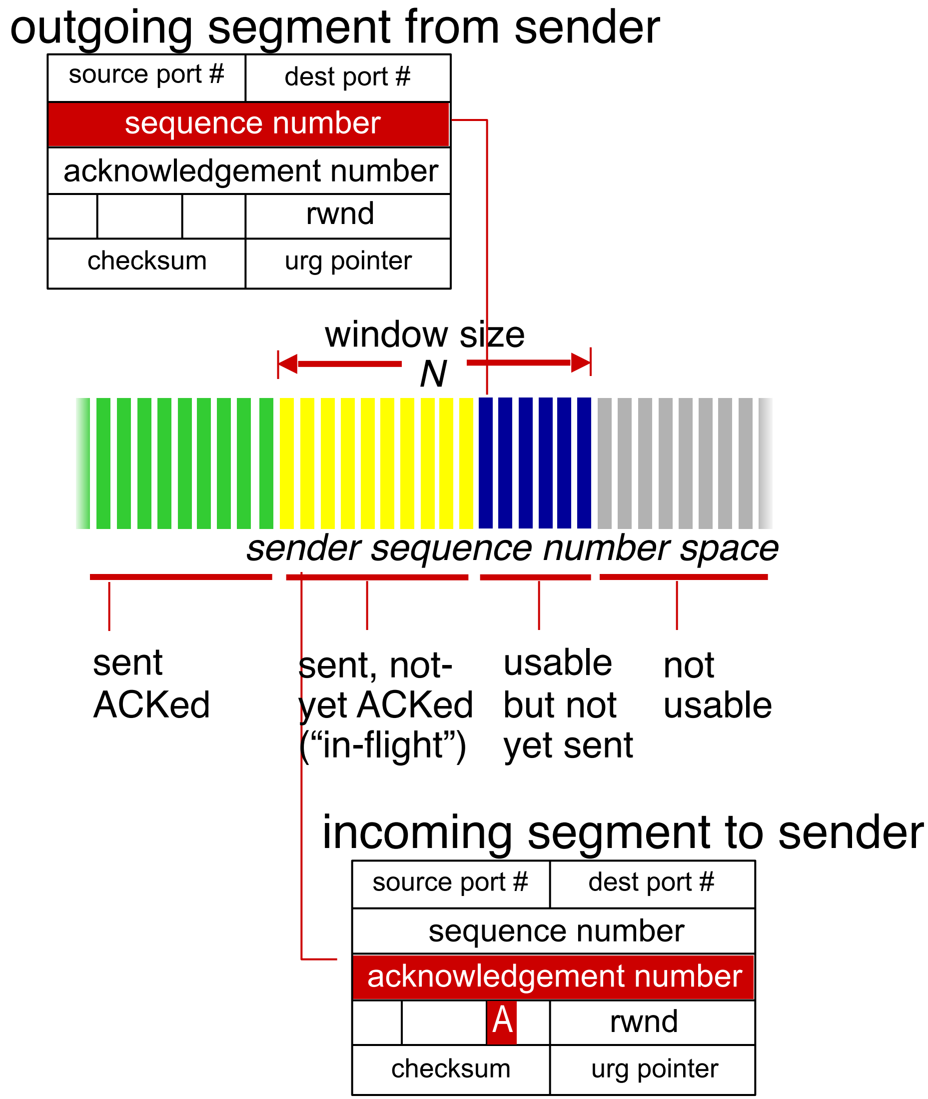
- byte stream "number" of first byte in segment's data

acknowledgements:

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

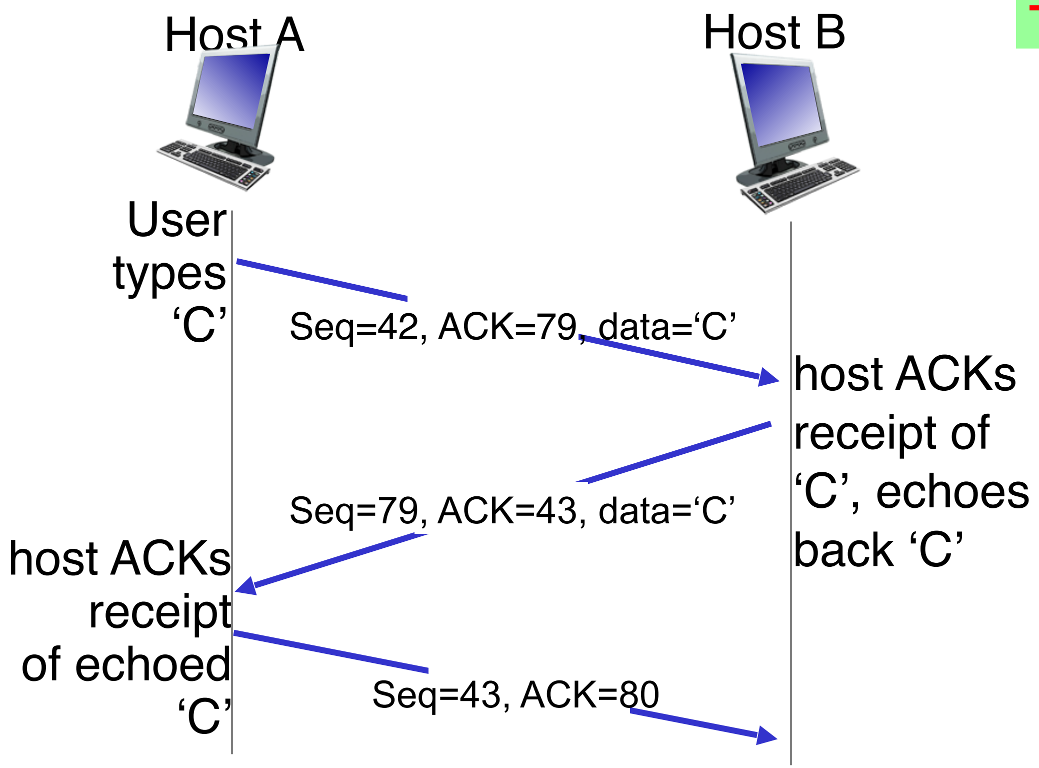
Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACKs

•Example: telnet
→'echo back'



simple telnet scenario

•Piggyback
→the ack for client-to-server data is carried in a segment carrying server-to-client data

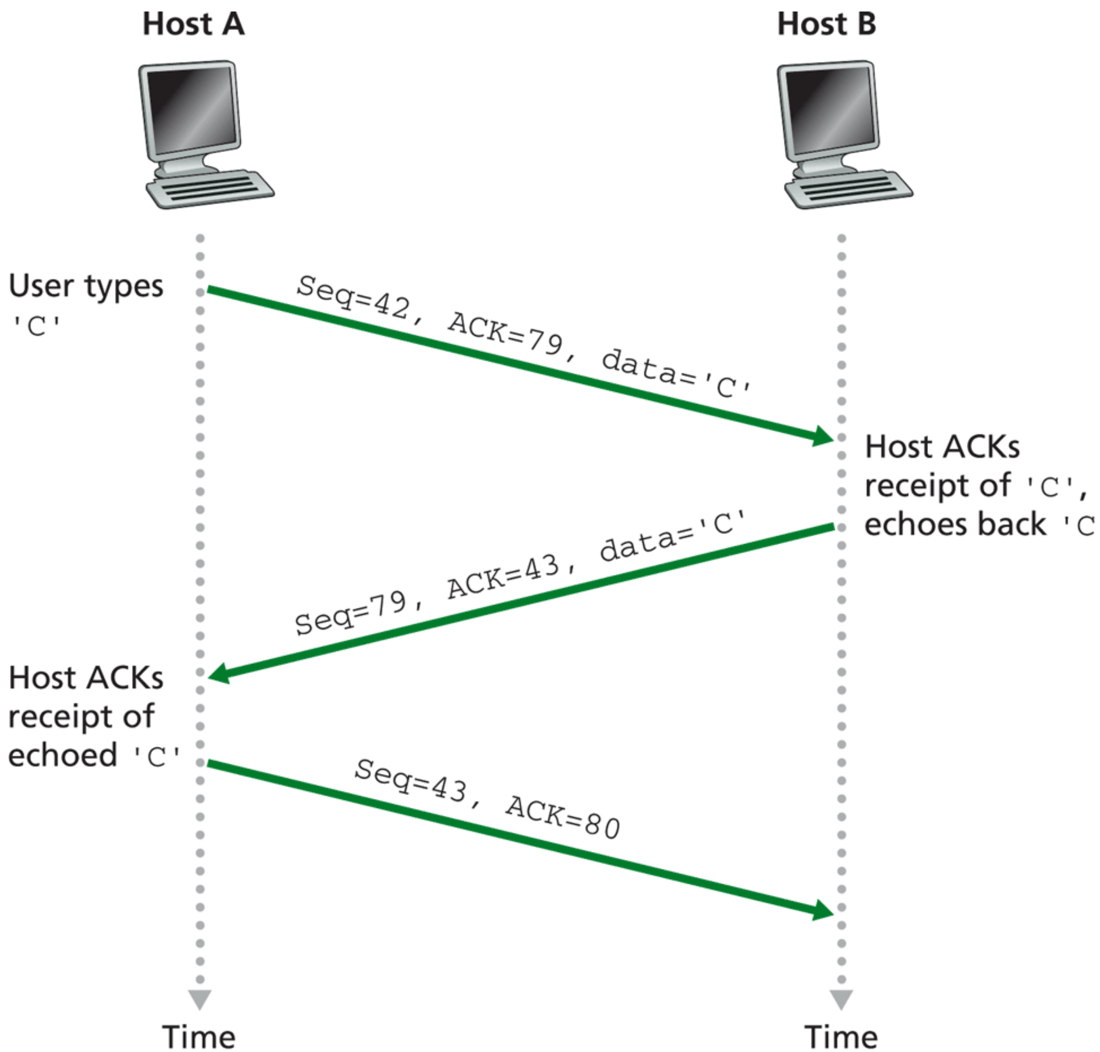


Figure 3.31 ♦ Sequence and acknowledgement numbers for a simple Telnet application over TCP

TCP round trip time, timeout

Q: how to set TCP timeout value?

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

Q: how to estimate RTT?

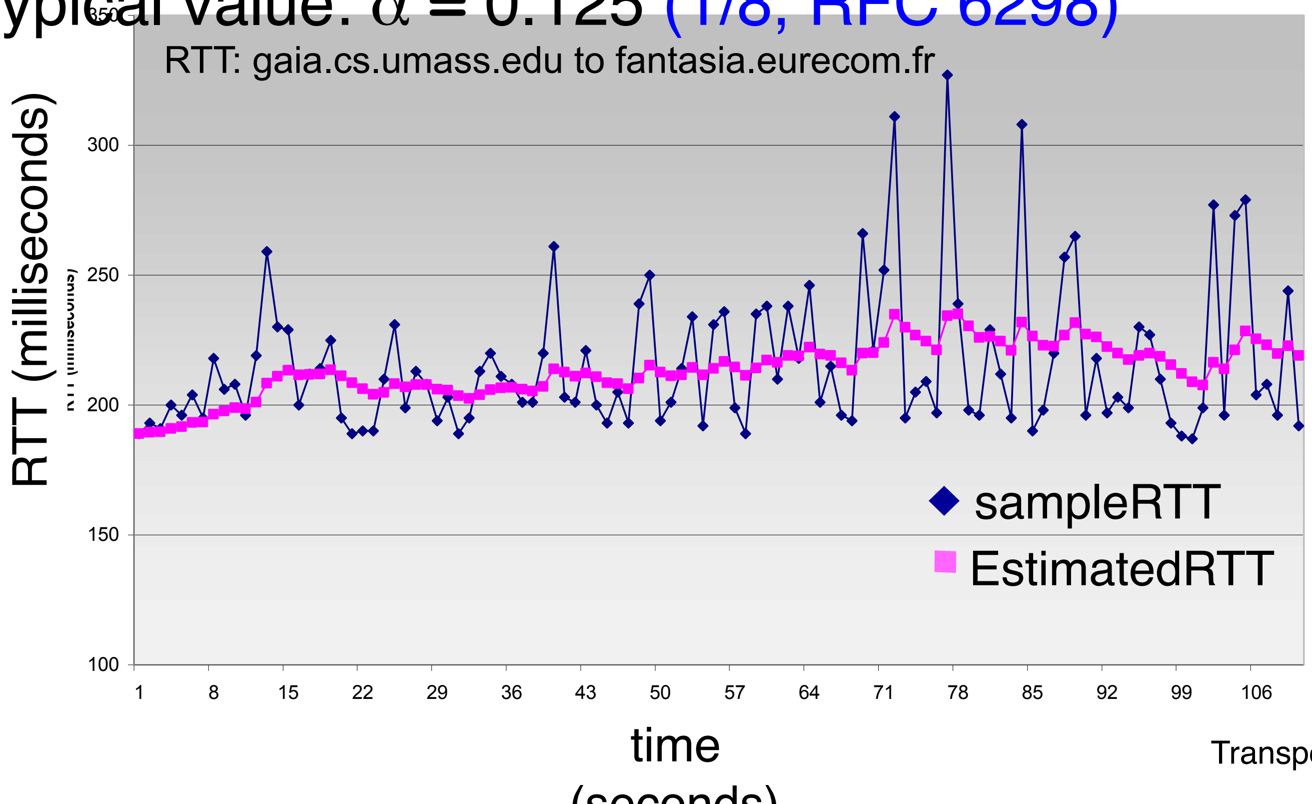
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

•TCP uses a timeout/retransmit mechanism to recover from lost segments

TCP round trip time, timeout

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125 (1/8, RFC 6298)



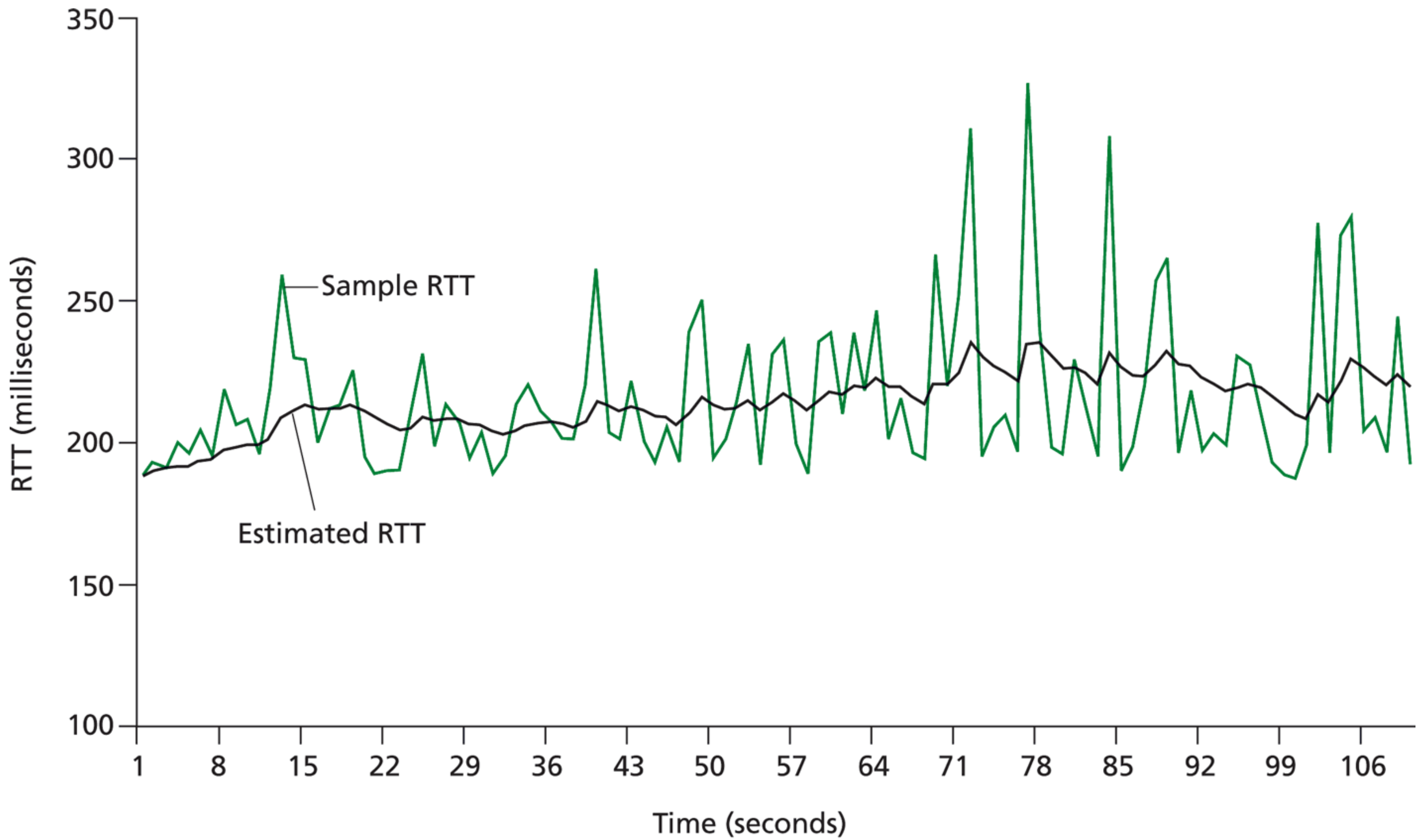


Figure 3.32 ♦ RTT samples and RTT estimates

TCP round trip time, timeout

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

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

(typically, $\beta = 0.25$)

$$\texttt{TimeoutInterval} = \underbrace{\texttt{EstimatedRTT}}_{\text{estimated RTT}} + 4 * \underbrace{\texttt{DevRTT}}_{\text{"safety margin"}}$$

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

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

TCP reliable data transfer

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

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

TCP sender events:

data rcvd from app:

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

timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

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

```
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)
      break;
  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer
      break;
  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
              start timer
      }
      break;
```

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

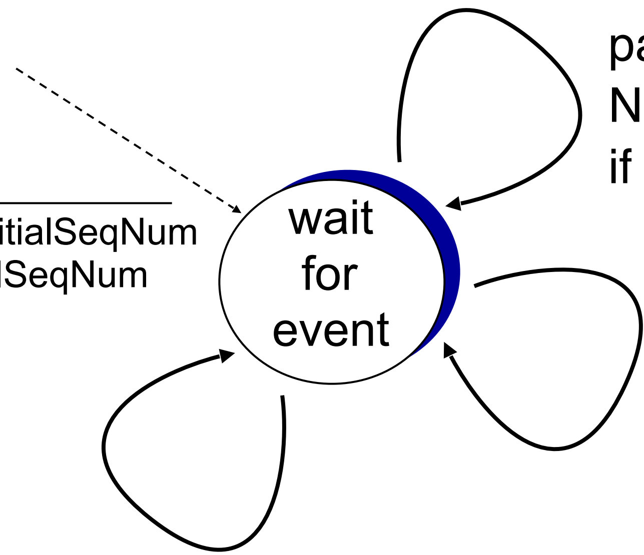
TCP sender (simplified)

data received from application above

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

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



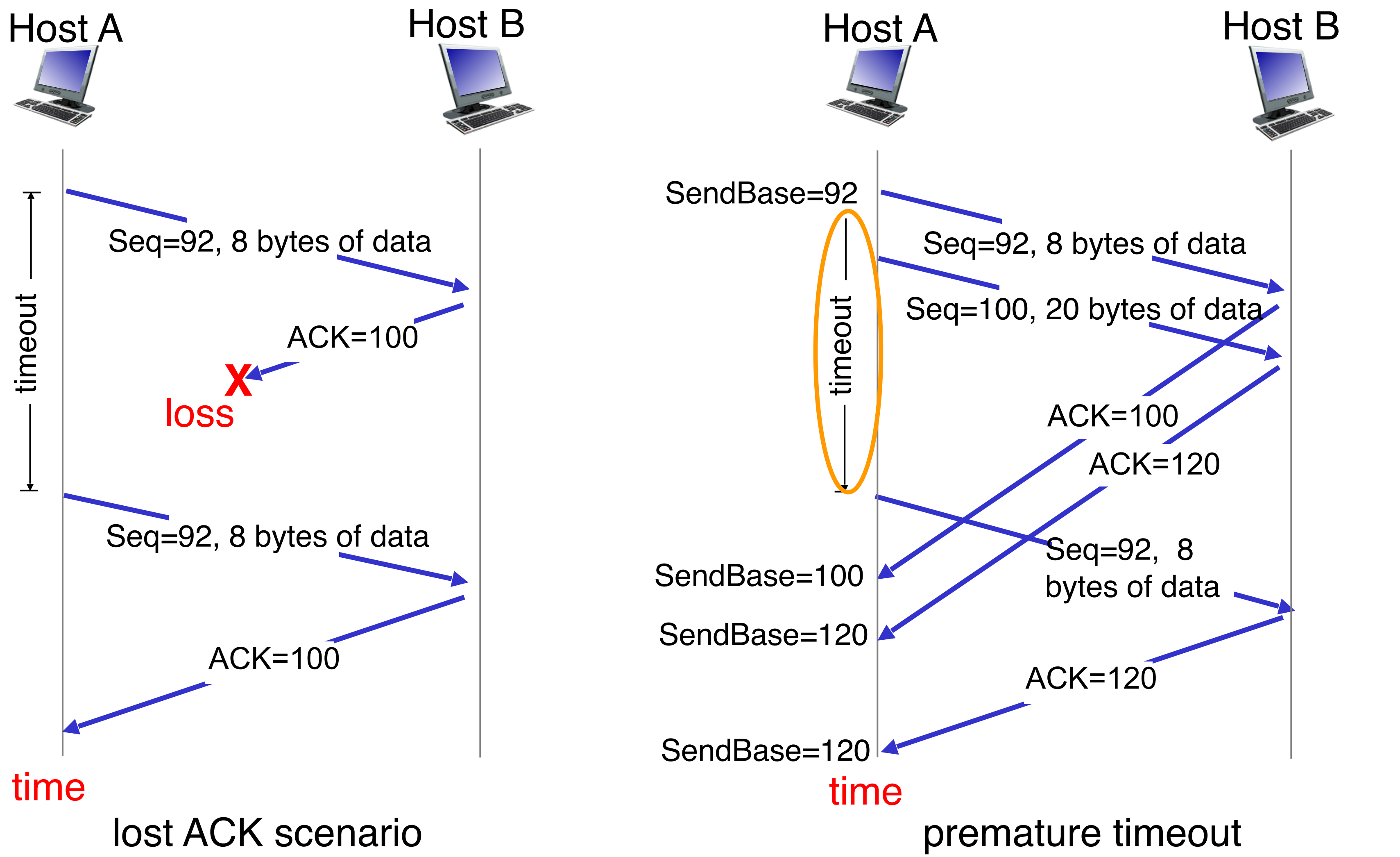
timeout

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

ACK received, with ACK field value y

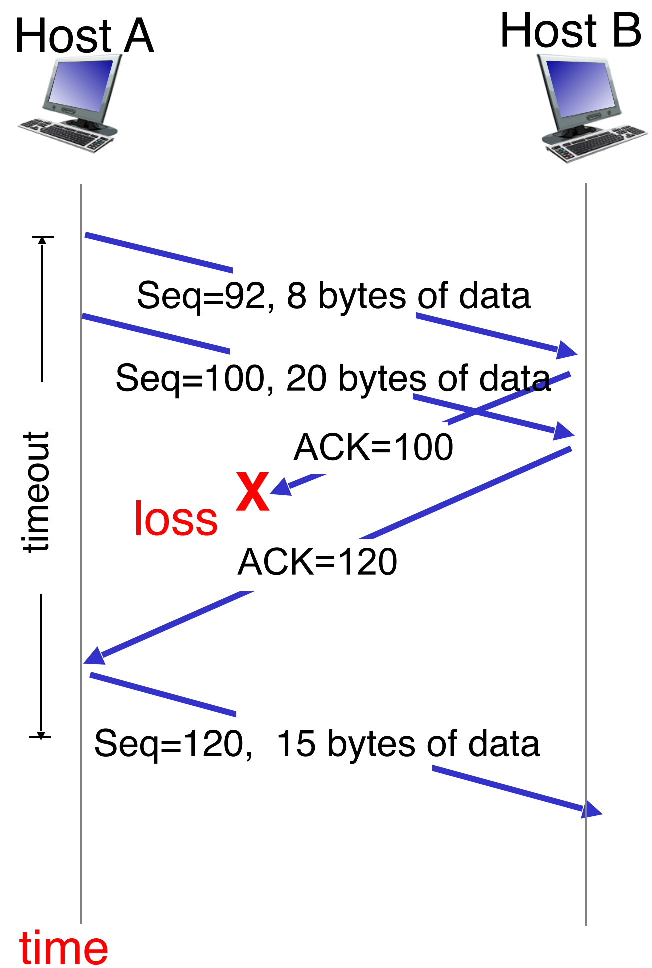
```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
 }
```

TCP: retransmission scenarios



•TCP has only one retransmission timer

TCP: retransmission scenarios



cumulative ACK

•Doubling the timeout interval
→ ex. TimeoutInterval associated with the oldest not-yet-acknowledged segment: 0.75 sec, 1.5 sec, 3.0 sec ......

TCP ACK generation [RFC 1122, RFC 2581, RFC 5681]

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq. #. All data up to expected seq. # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq. #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. #. Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

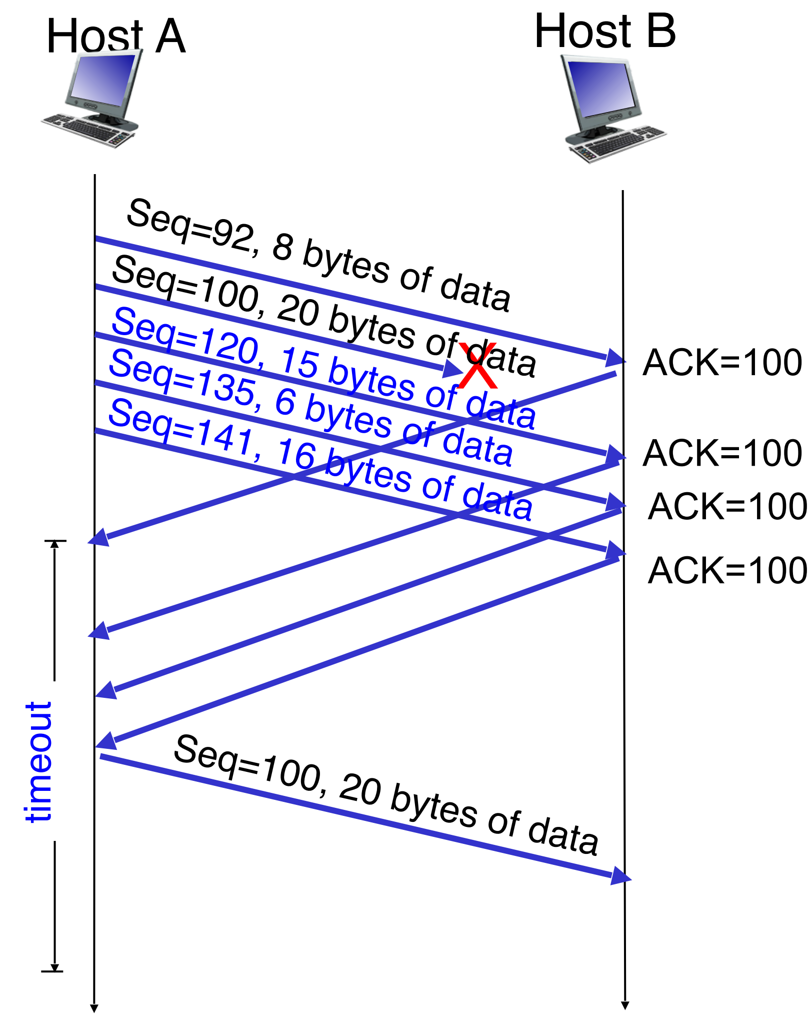
- timeout period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs
  - sender often sends many segments back to back
  - if segment is lost, there will likely be many back-to-back duplicate ACKs

TCP fast retransmit

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

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

TCP fast retransmit



fast retransmit after sender
receipt of triple duplicate ACK

Fast retransmit algorithm:

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

a duplicate ACK for
already ACKed segment

TCP fast retransmit

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

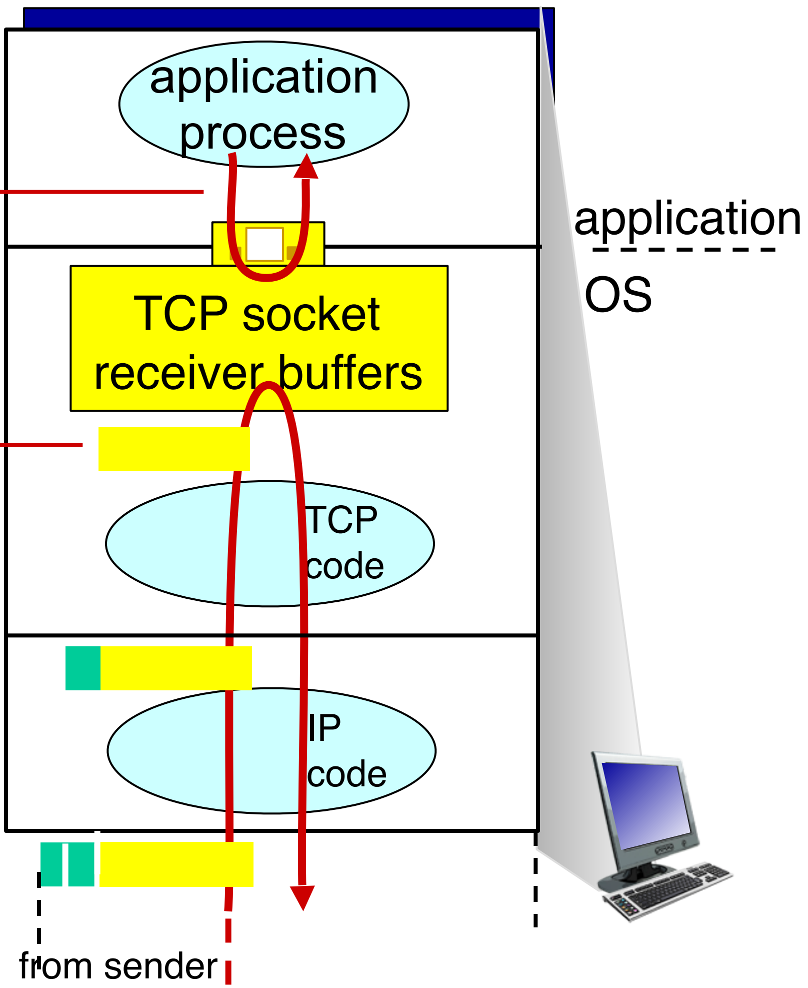
3.6 principles of congestion control

3.7 TCP congestion control

TCP flow control

application may remove data from TCP socket buffers ....

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



receiver protocol stack

flow control

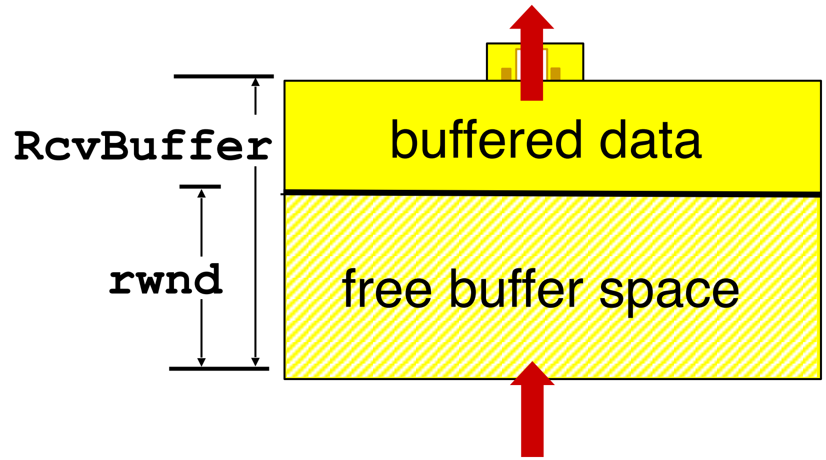
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

•After advertising rwnd = 0 to Host A, Host B has nothing to send to A
→ What happen?

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

to application process



TCP segment payloads

receiver-side buffering

LastByteSent - LastByteAcked <= rwnd

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

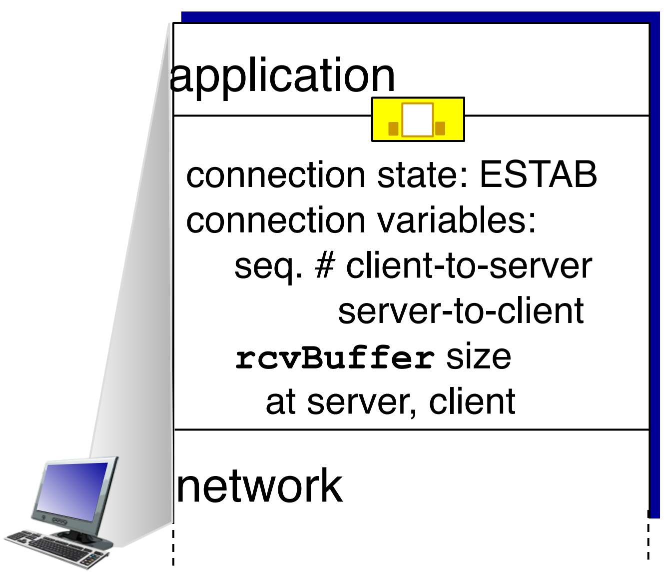
3.6 principles of congestion control

3.7 TCP congestion control

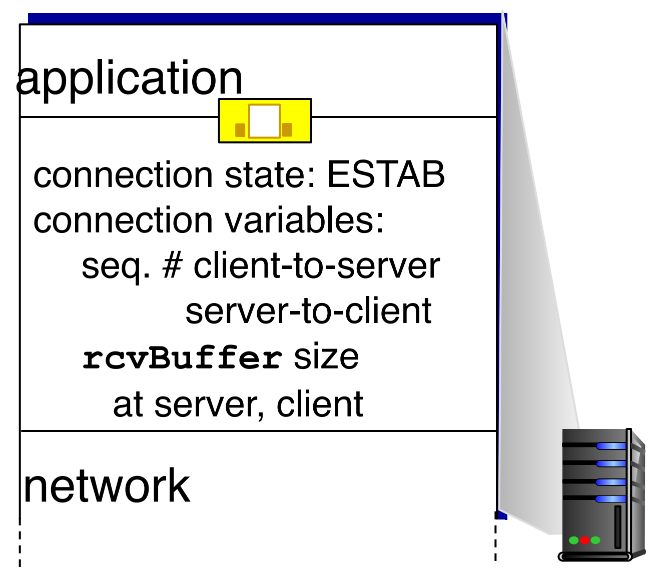
Connection Management

before exchanging data, sender/receiver "handshake":

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

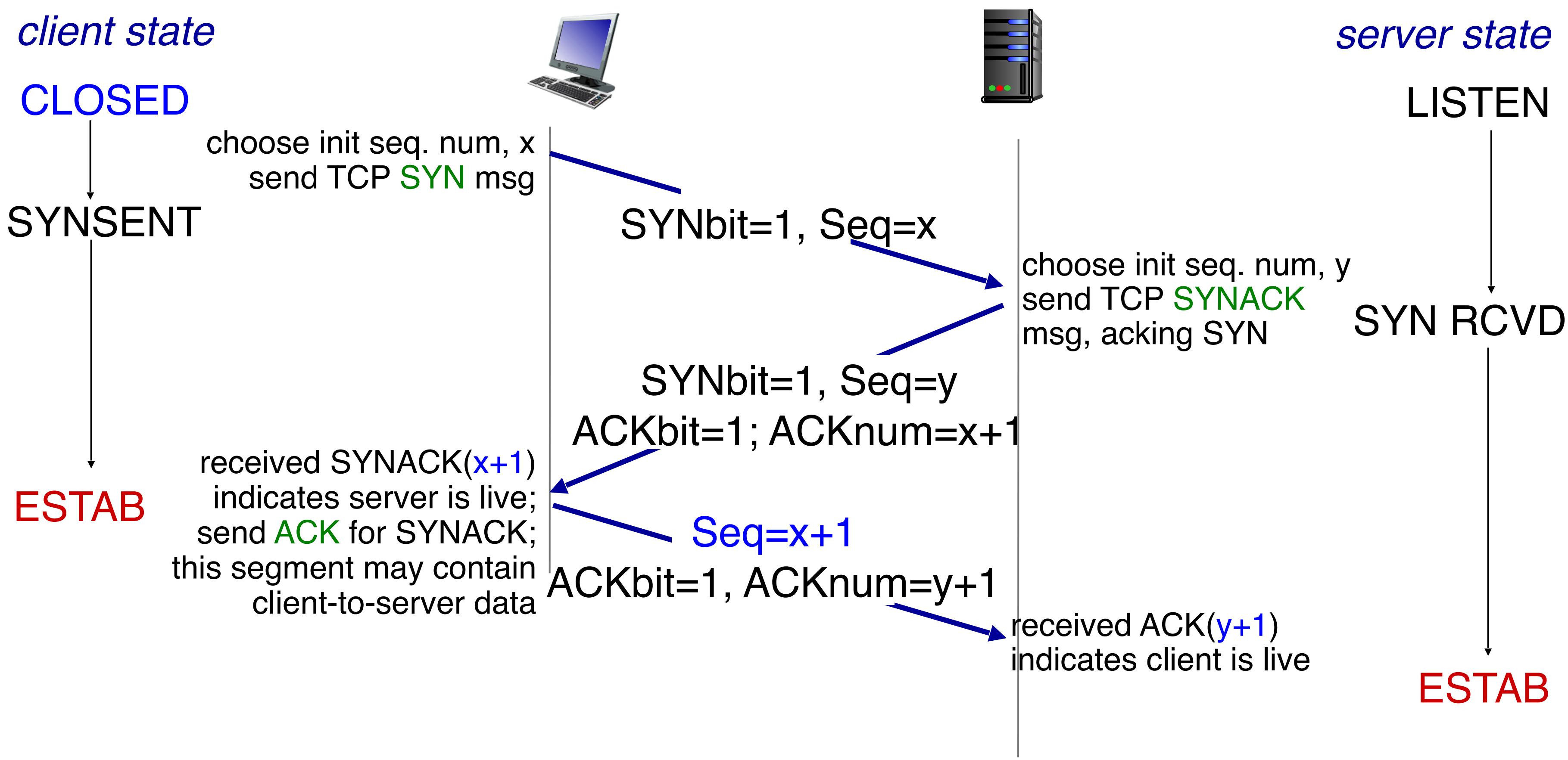



```
Socket clientSocket = new
  Socket("hostname", "port
  number");
```



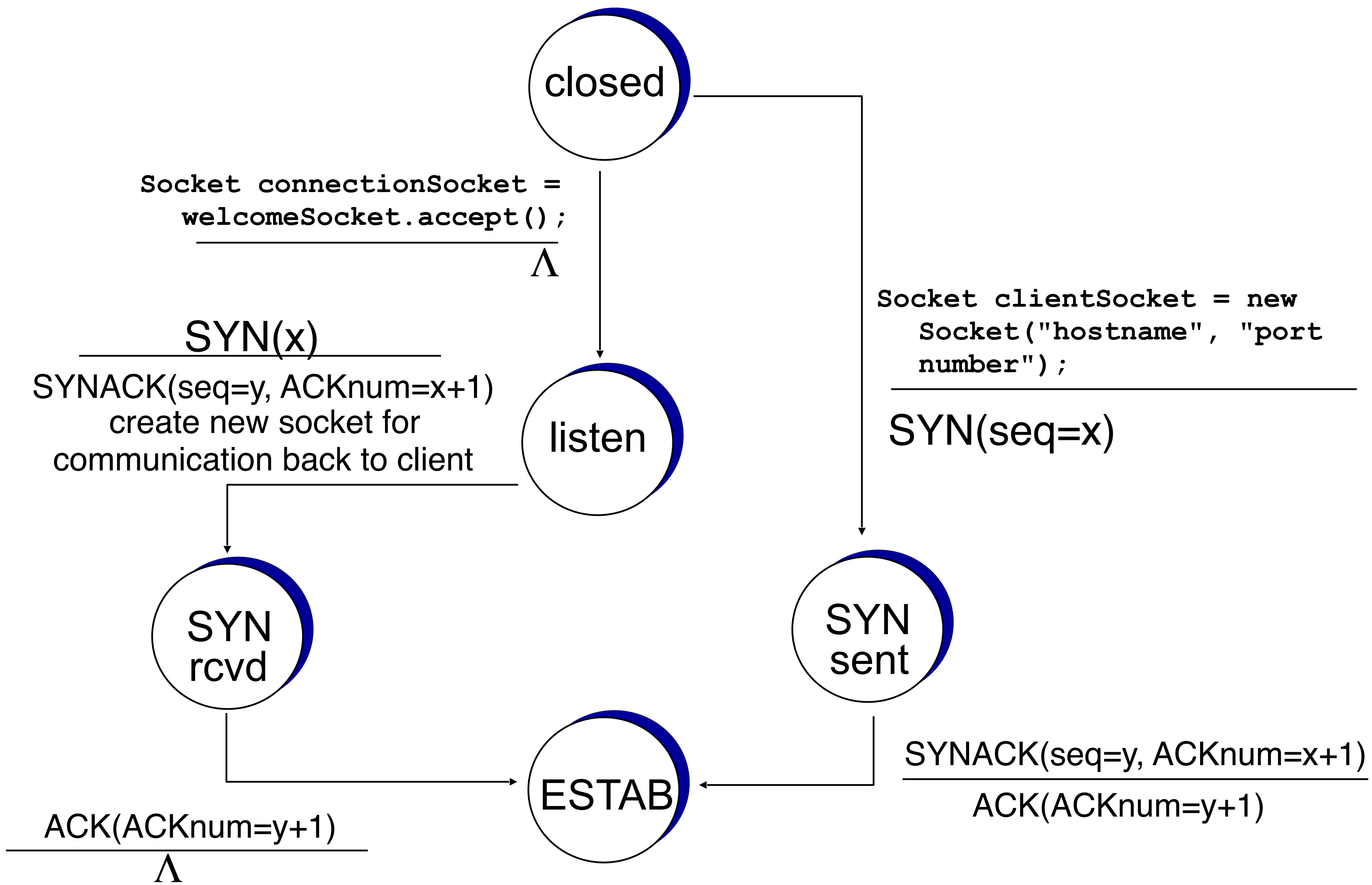

```
Socket connectionSocket =
  welcomeSocket.accept();
```

TCP 3-way handshake



- Security attack "SYN flood attack"
- An effective defense known as SYN cookies [RFC 4987]

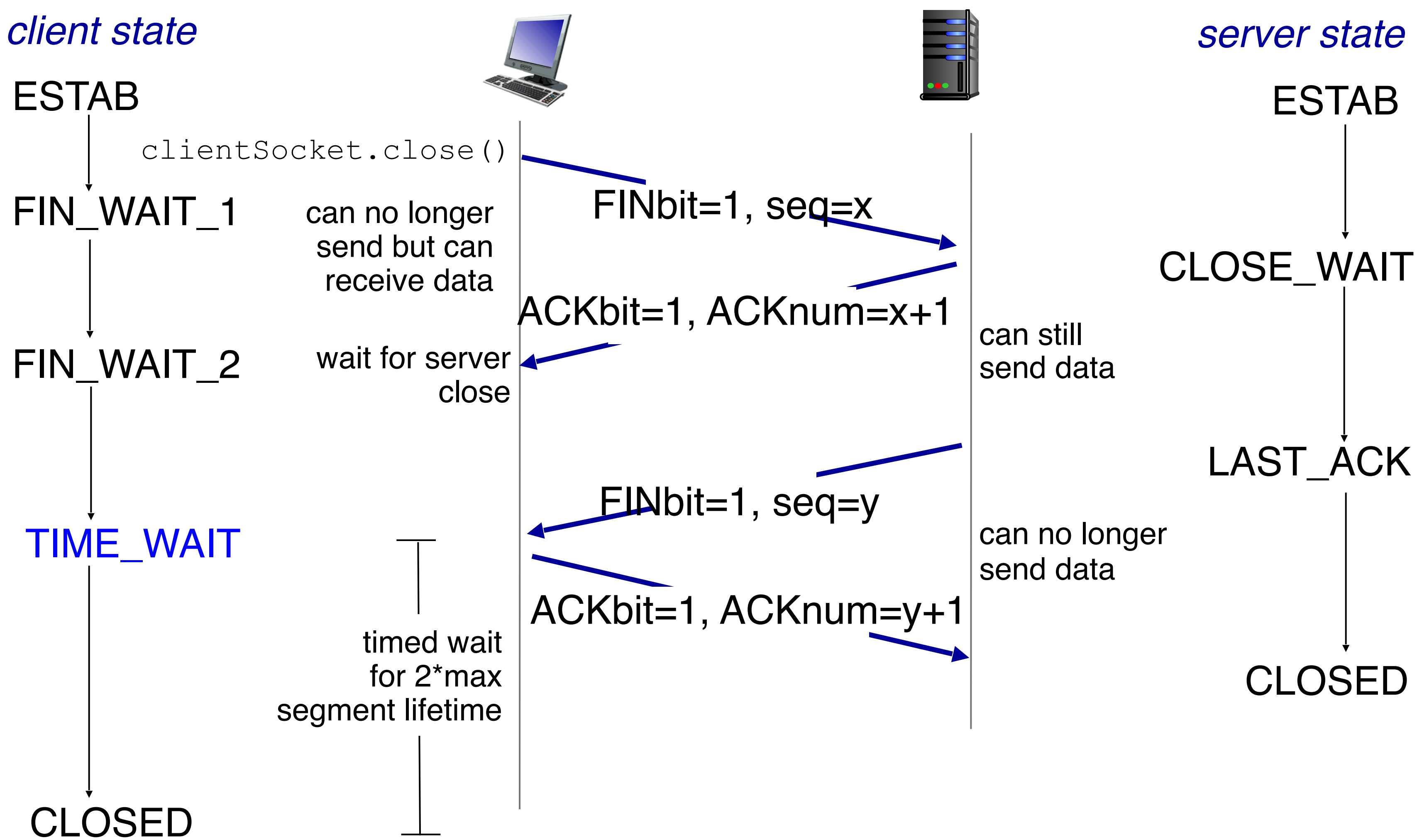
TCP 3-way handshake: FSM



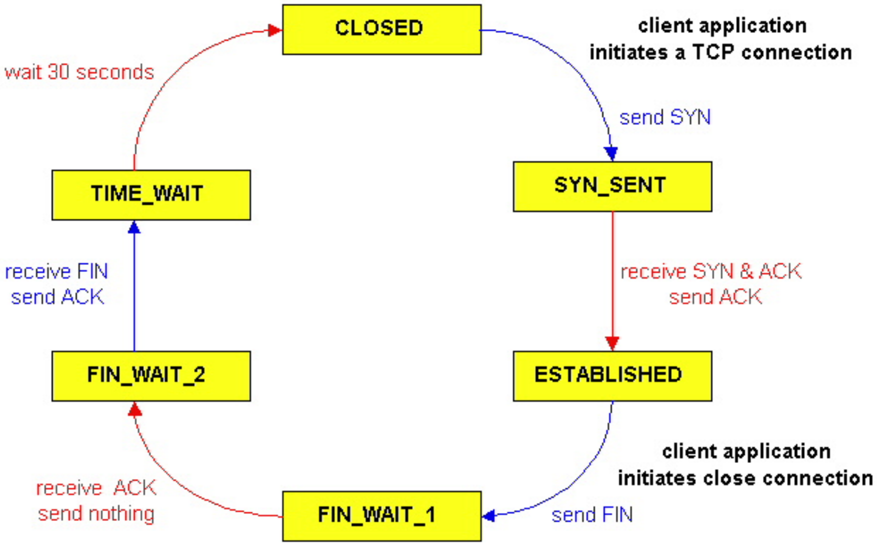
TCP: closing a connection

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

TCP: closing a connection

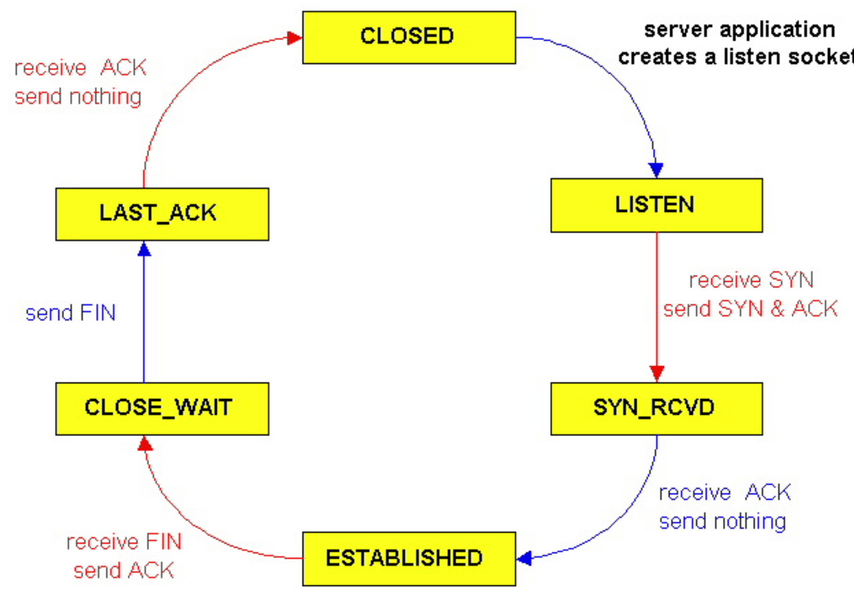


TCP States



TCP client lifecycle

TCP server lifecycle



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

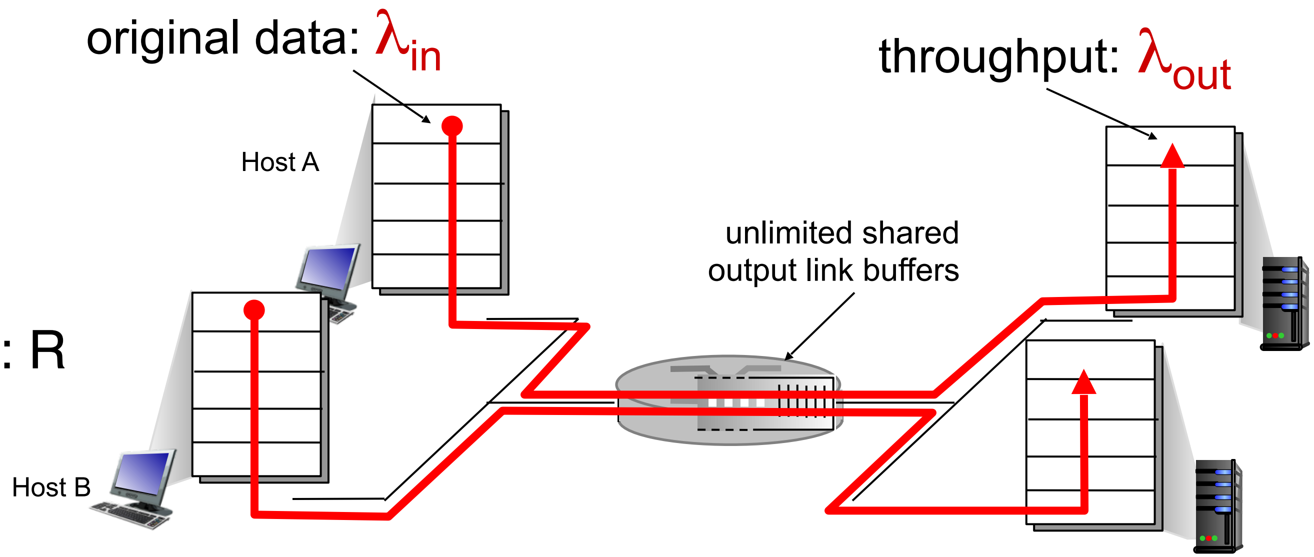
Principles of congestion control

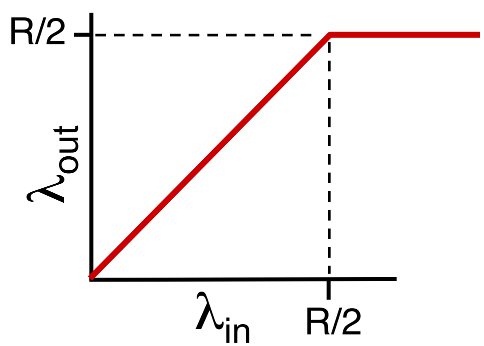
congestion:

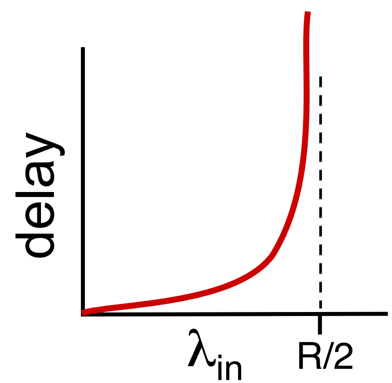
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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





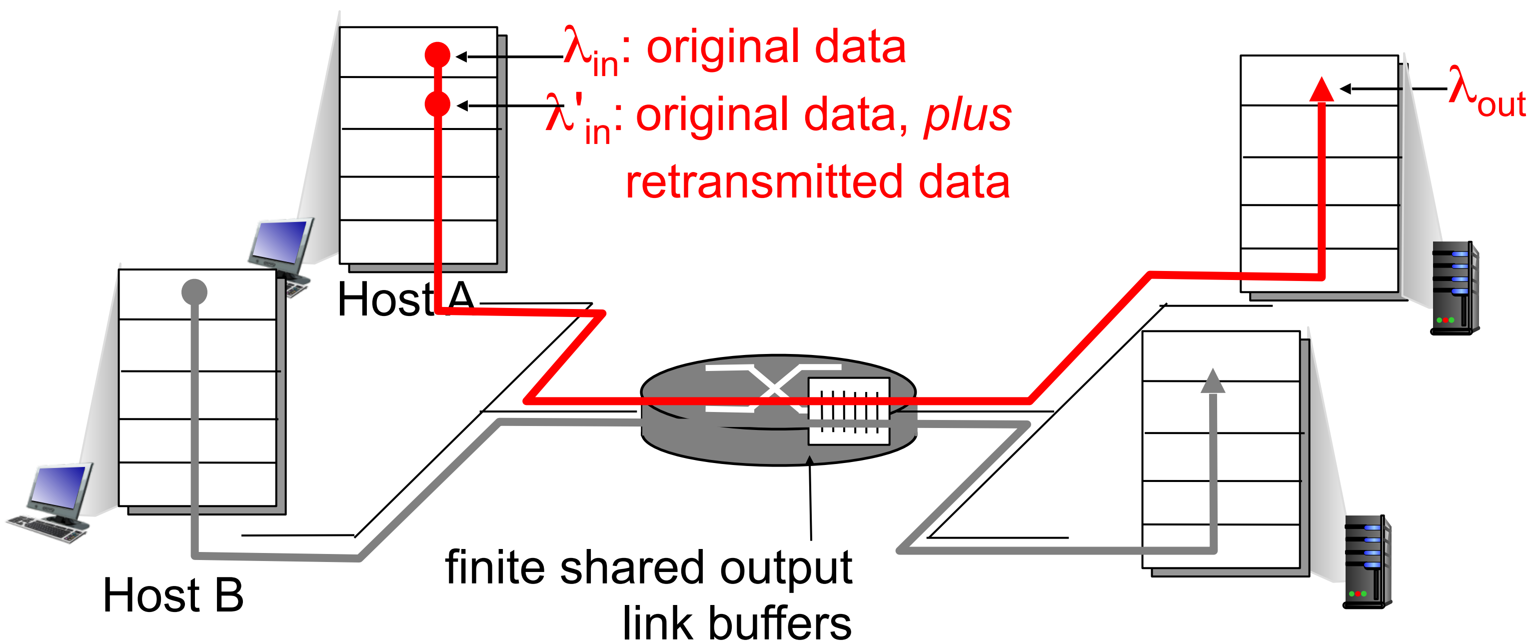


- maximum per-connection throughput: R/2

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

Causes/costs of congestion: scenario 2

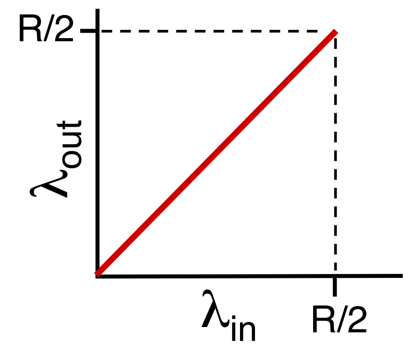
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions*: $\lambda'_{in} \geq \lambda_{in}$

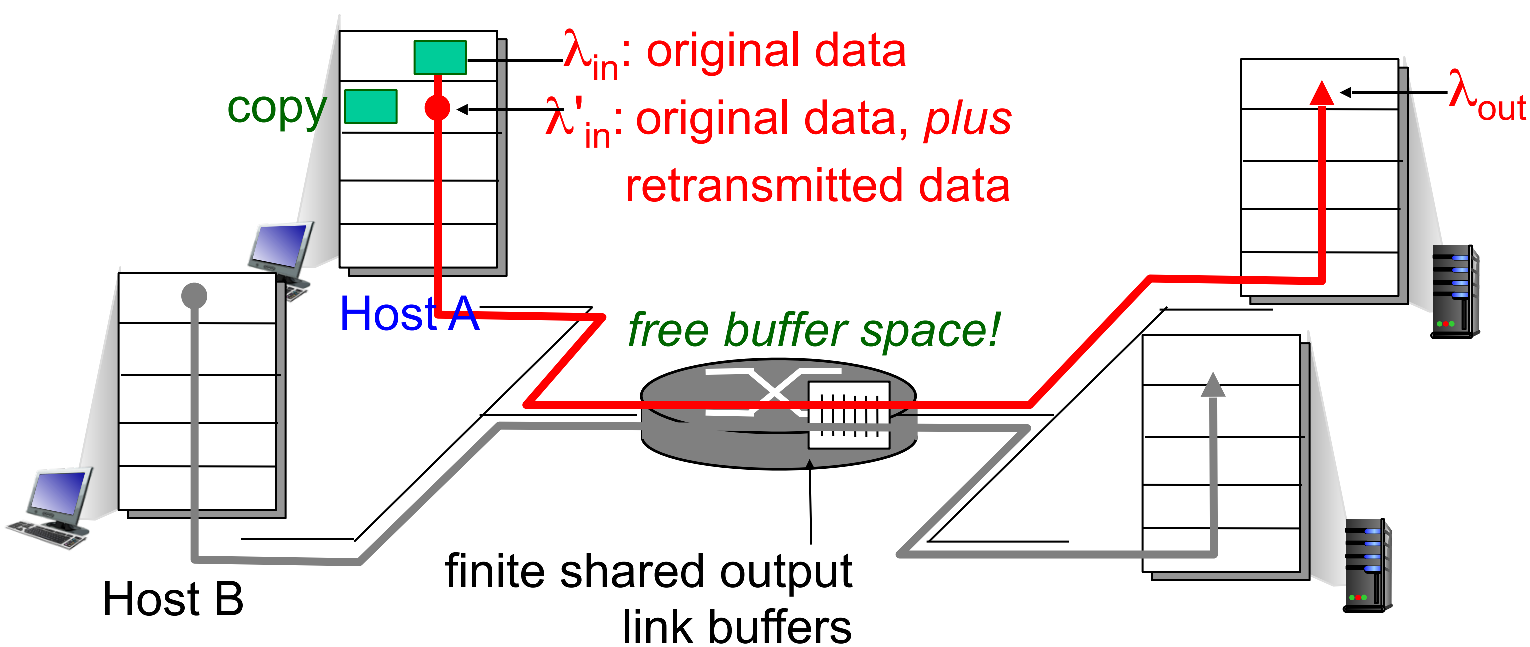


Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

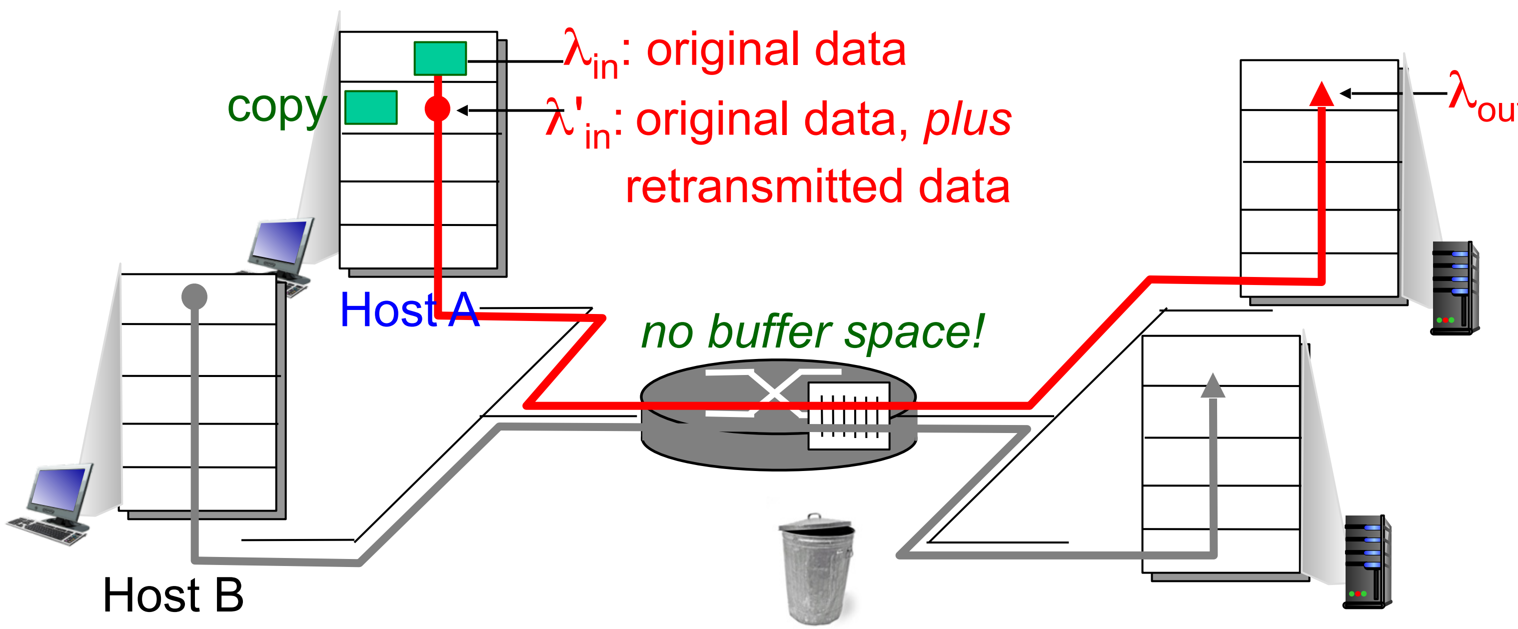




Causes/costs of congestion: scenario 2

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

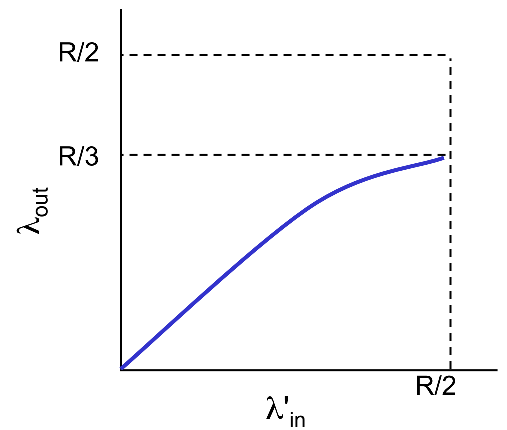
- sender only resends if packet *known* to be lost

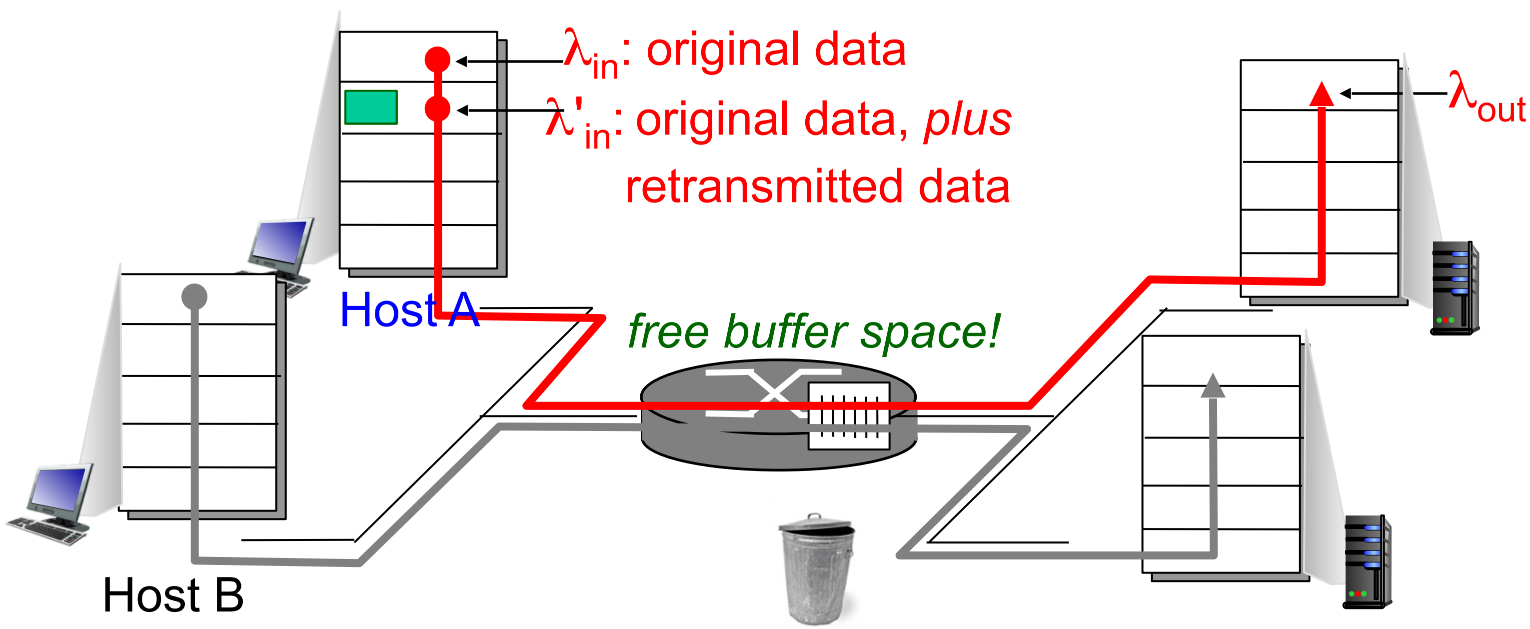


Causes/costs of congestion: scenario 2

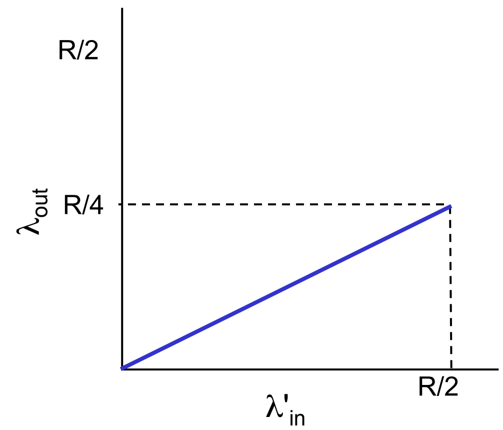
Idealization: known loss
packets can be lost, dropped at router due to full buffers

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



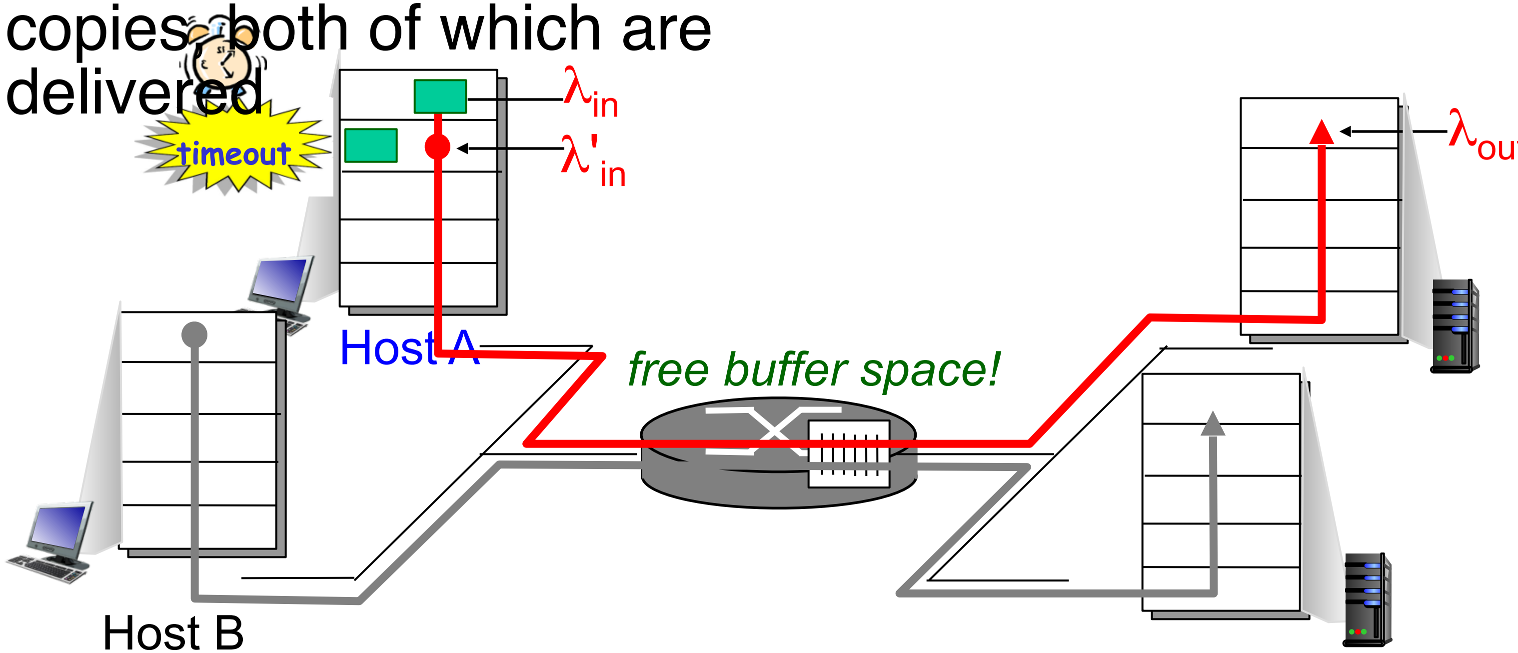


Causes/costs of congestion: scenario 2



Realistic: duplicates

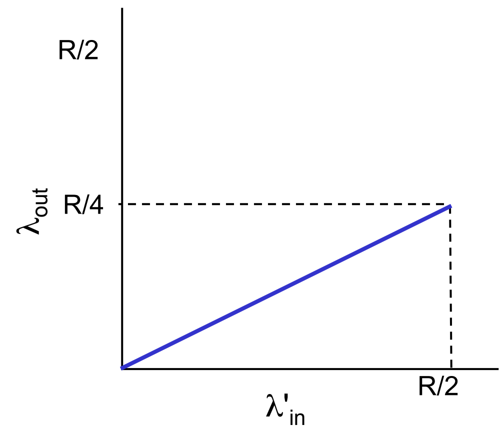
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



Causes/costs of congestion: scenario 2

Realistic: duplicates

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



"costs" of congestion:

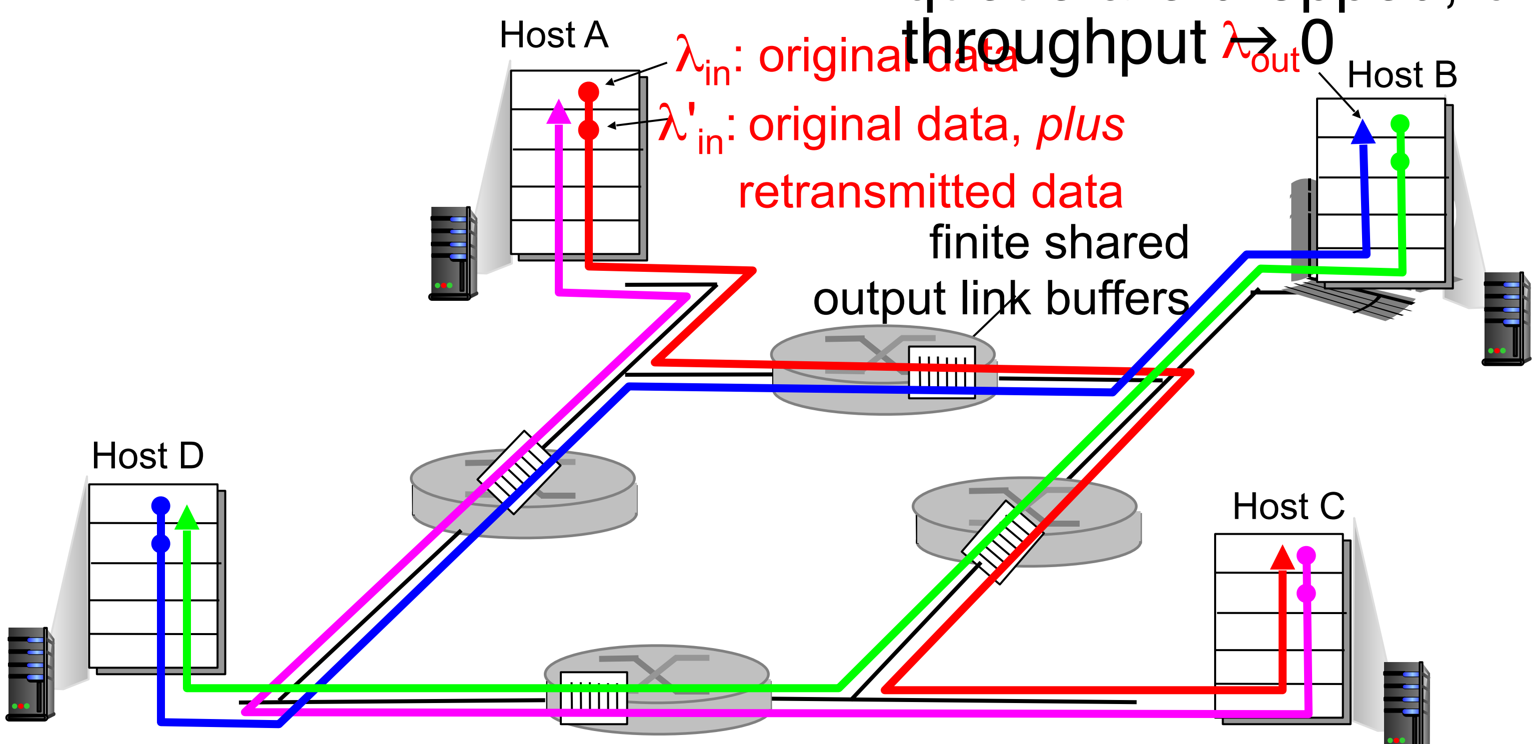
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

Causes/costs of congestion: scenario 3

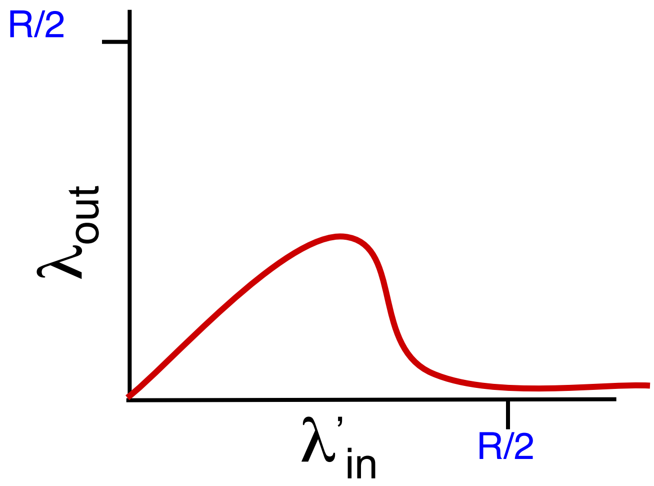
- four senders
- multihop paths
- timeout/retransmit

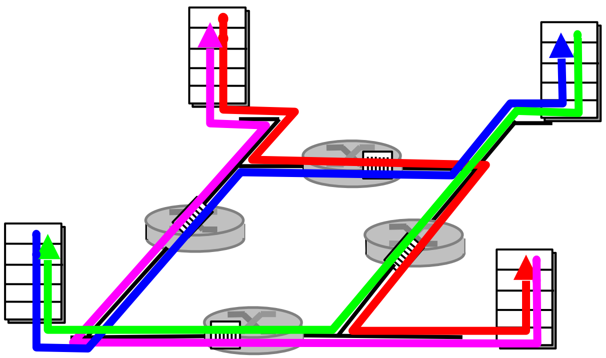
Q: what happens as λ_{in} and λ'_{in} increase?

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



Causes/costs of congestion: scenario 3





another "cost" of congestion:

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

•The router should select packets that have traversed some number of upstream routers

Approaches towards congestion control

two broad approaches towards congestion control:

end-to-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 (IBM SNA, DEC DECnet, TCP/IP ECN, ATM Available Bit Rate (ABR))
  - explicit rate for sender to send at

- Network-assisted congestion control
  1. Direct feedback: Router sends “choke packet” to the sender
  2. Router marks/updates a field in a packet flowing from sender to receiver, then receiver notifies the sender → takes a full round trip time (more common form

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

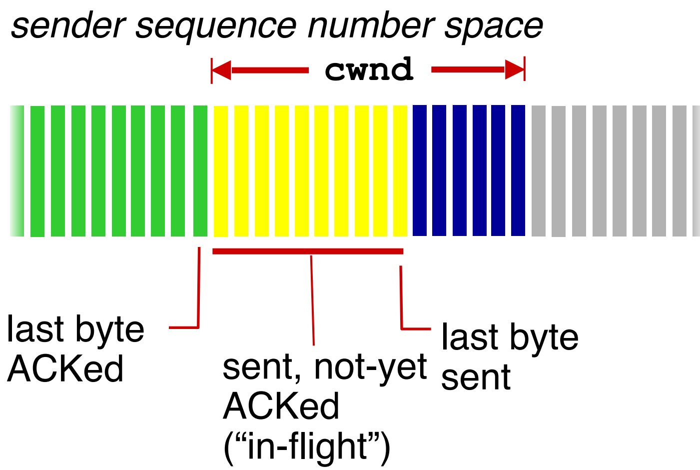
3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

TCP Congestion Control: details



- end-to-end congestion control (no network assistance)
- sender limits transmission:

$$\texttt{LastByteSent - LastByteAcked} \le \texttt{min(cwnd, rwnd)}$$

- **`cwnd`** (congestion window) is dynamic, function of perceived network congestion

TCP sending rate:

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

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

TCP Congestion Control: details

How does sender perceive congestion?

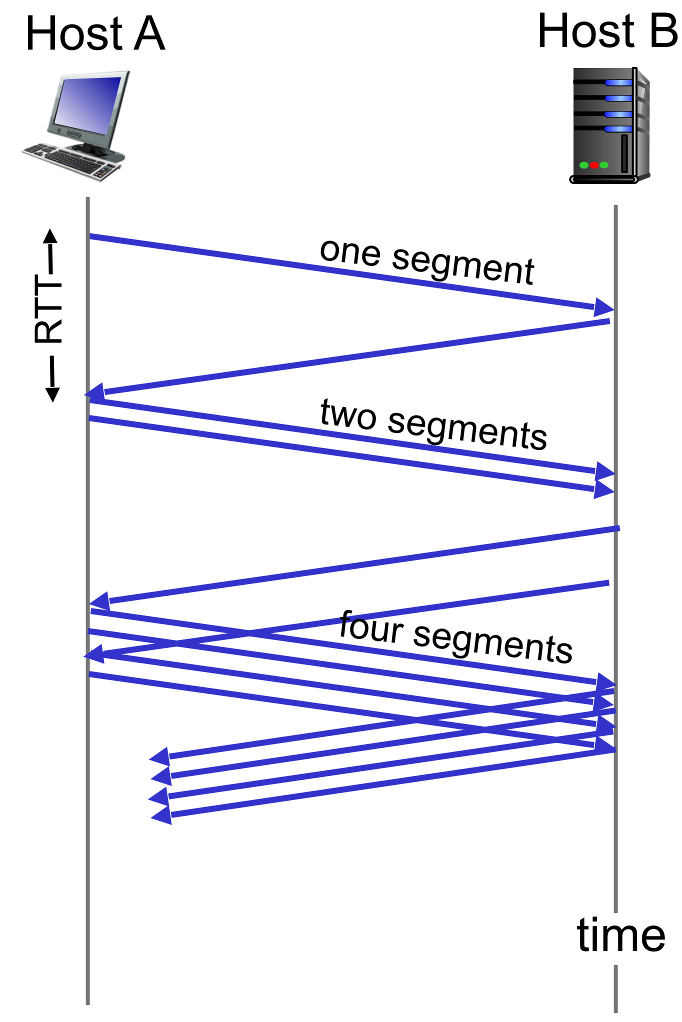
- loss event = timeout *or* 3 duplicate ACKs
- TCP sender reduces rate (**`cwnd`**) after loss event

three mechanisms:

- Slow Start
- Congestion Avoidance: Additive-Increase, Multiplicative-Decrease (AIMD)
- Fast Recovery: Reaction to Timeout Events

TCP Slow Start

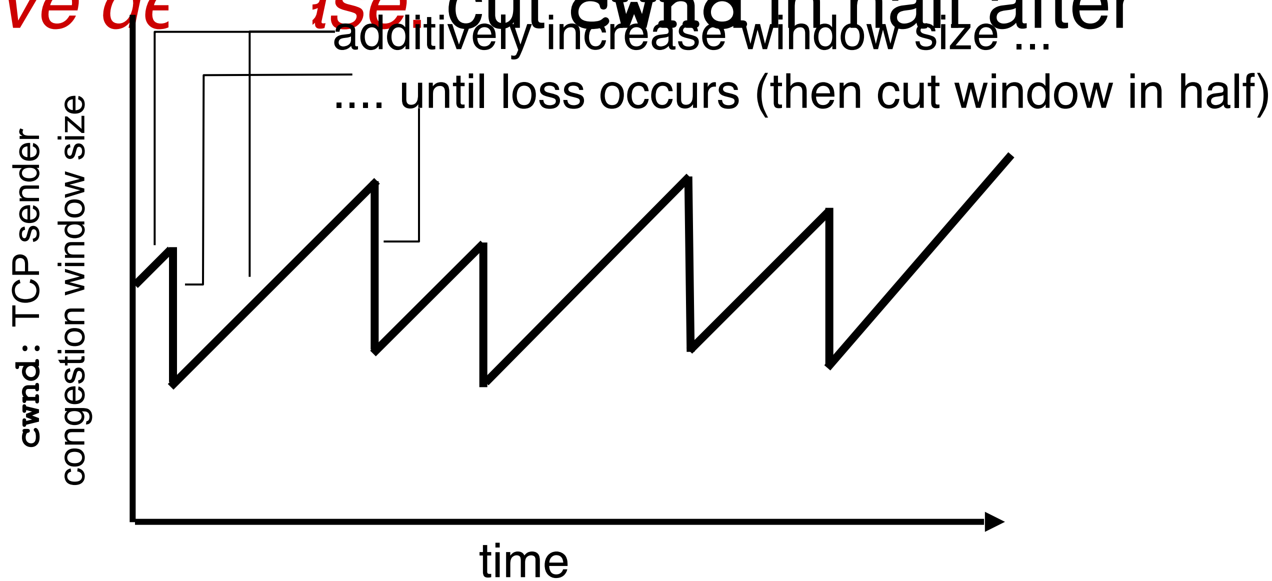
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



TCP congestion control: additive-increase, multiplicative-decrease (AIMD)

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

AIMD saw tooth behavior: probing for bandwidth



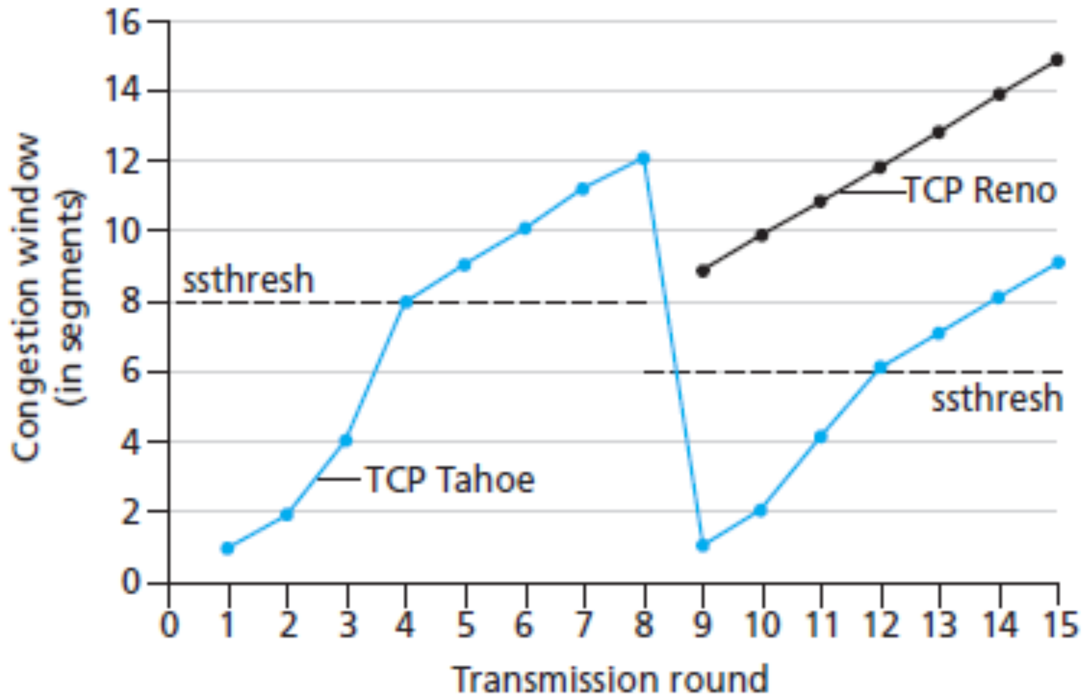
TCP: detecting, reacting to loss

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

TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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

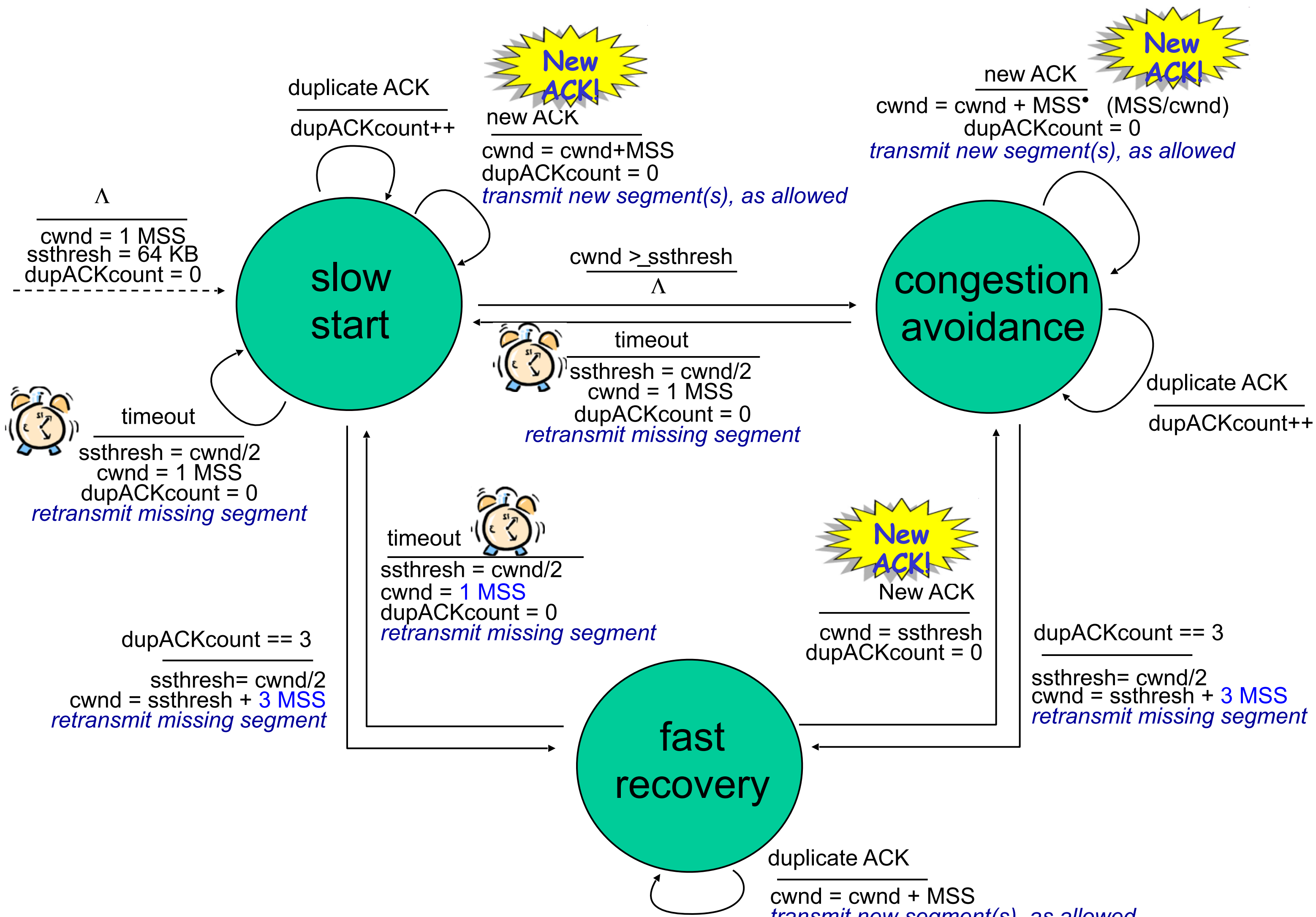


Implementation:

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

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

Summary: TCP Congestion Control



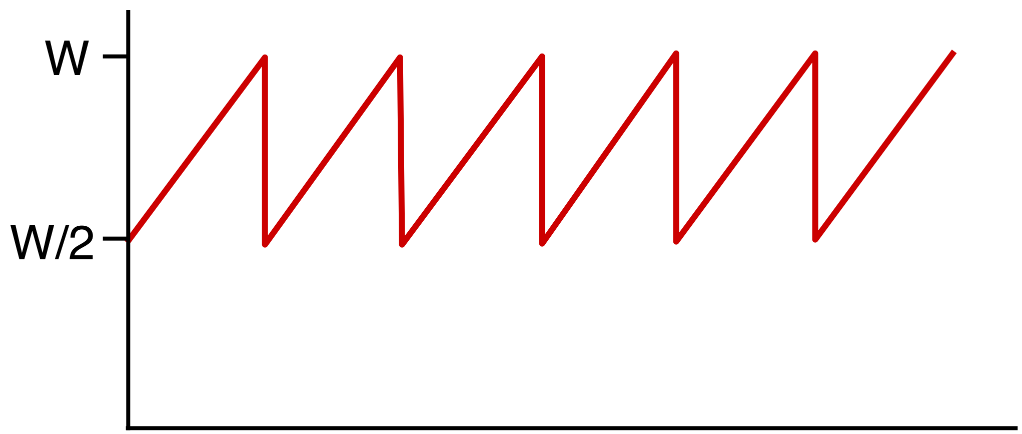
TCP sender congestion control

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 + 3*MSS, 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

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 0.75W per RTT

$$\text{avg. TCP thruput} = \frac{0.75\,W}{RTT} \text{ bytes/sec}$$



TCP Futures: TCP over "long, fat pipes"

- example: 1,500-byte segments, 100 ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

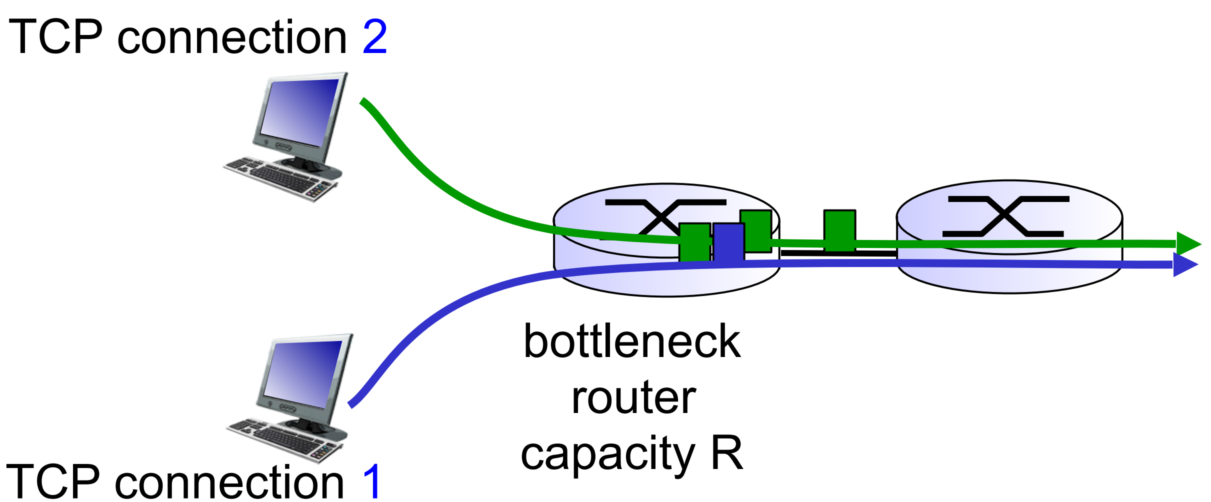
$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

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

- new versions of TCP for high-speed

TCP Fairness

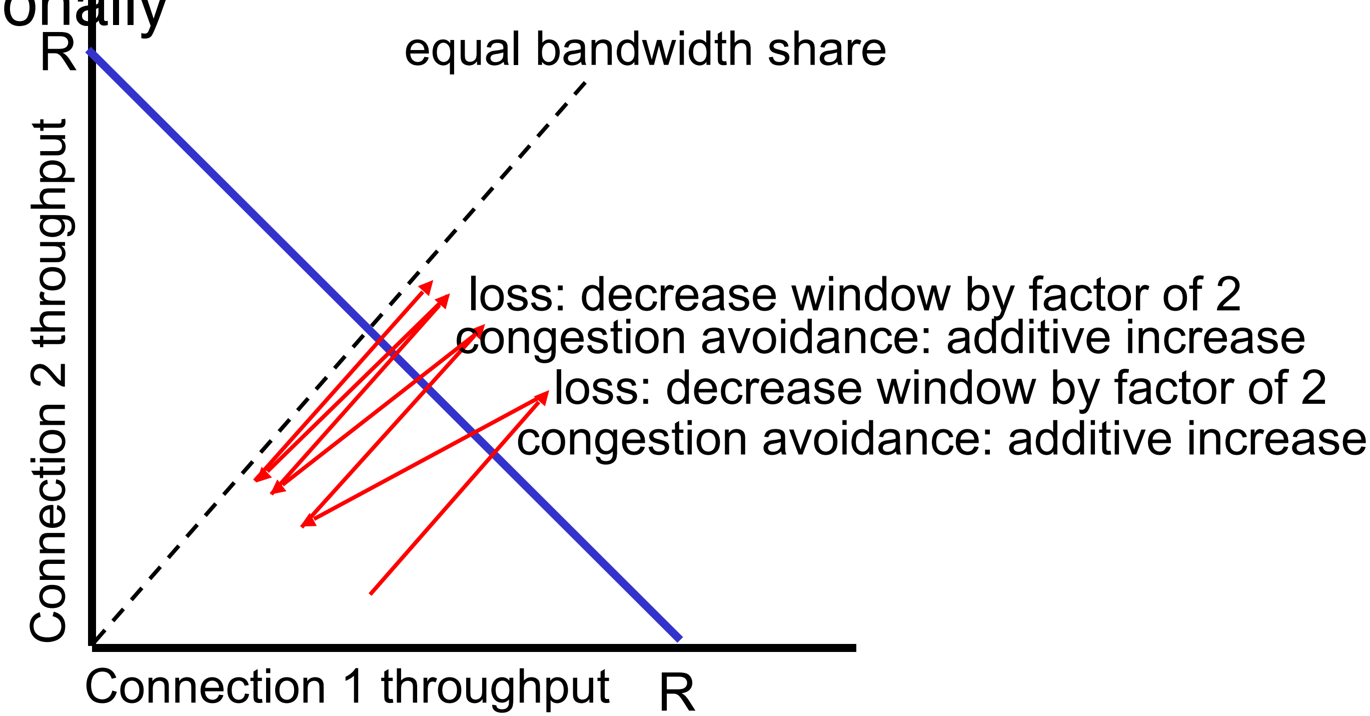
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

two competing sessions:

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



Fairness (more)

Fairness and UDP

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

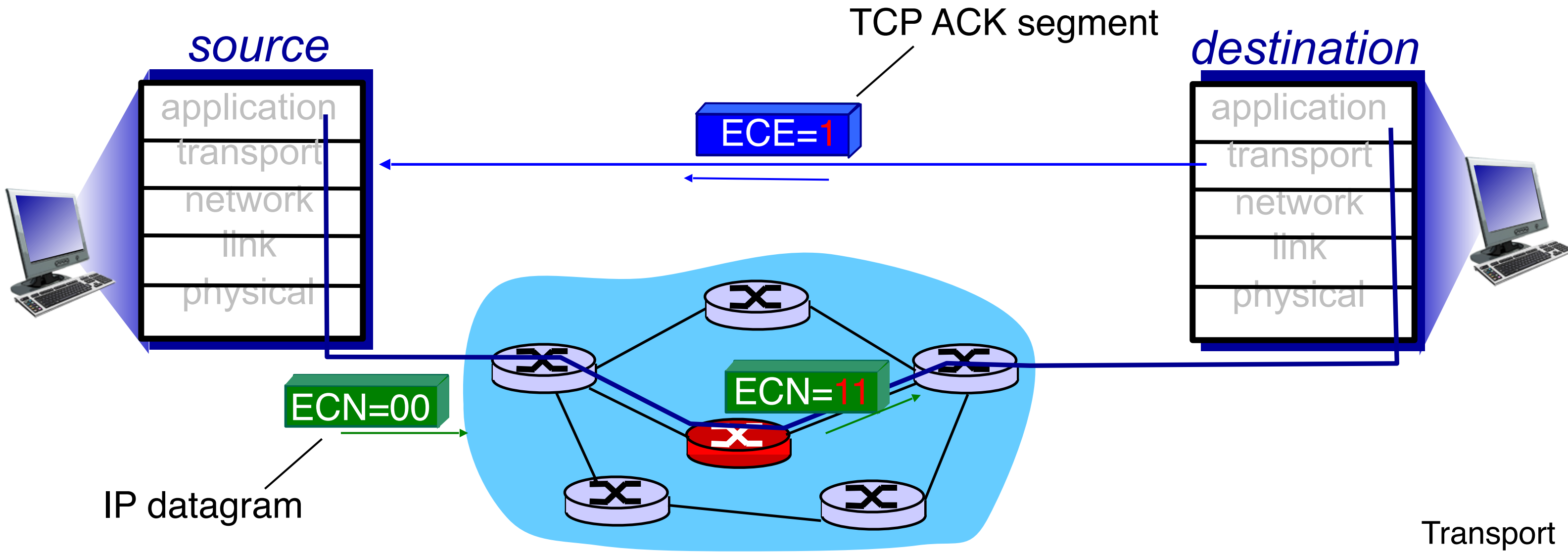
Fairness, parallel TCP connections

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

Explicit Congestion Notification ~~(ECN)~~

network-assisted congestion control:

- two bits in IP header (Type of Service field) marked *by network router* to indicate congestion (ECN=11)
- congestion indication carried to receiving host
- receiver (seeing congestion indication in IP datagram) sets ECE (Explicit Congestion Notification Echo) bit in receiver-to-sender TCP ACK segment to notify sender of congestion



Chapter 3: summary

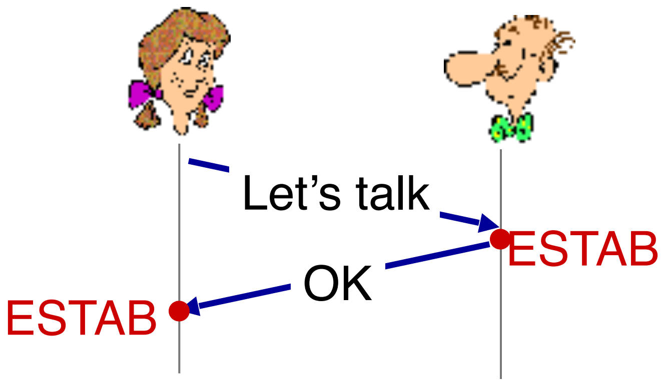
- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

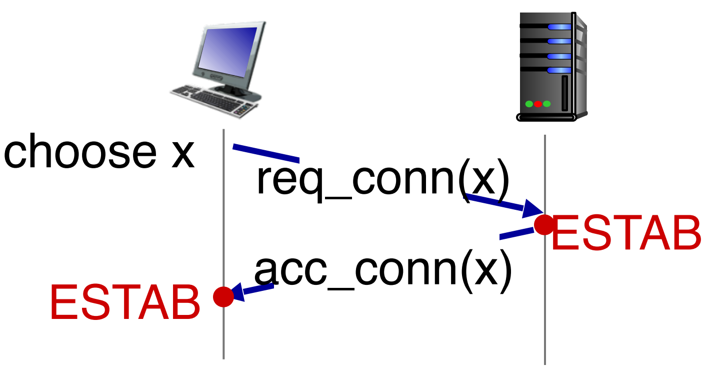
next:

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

Agreeing to establish a connection

2-way handshake:



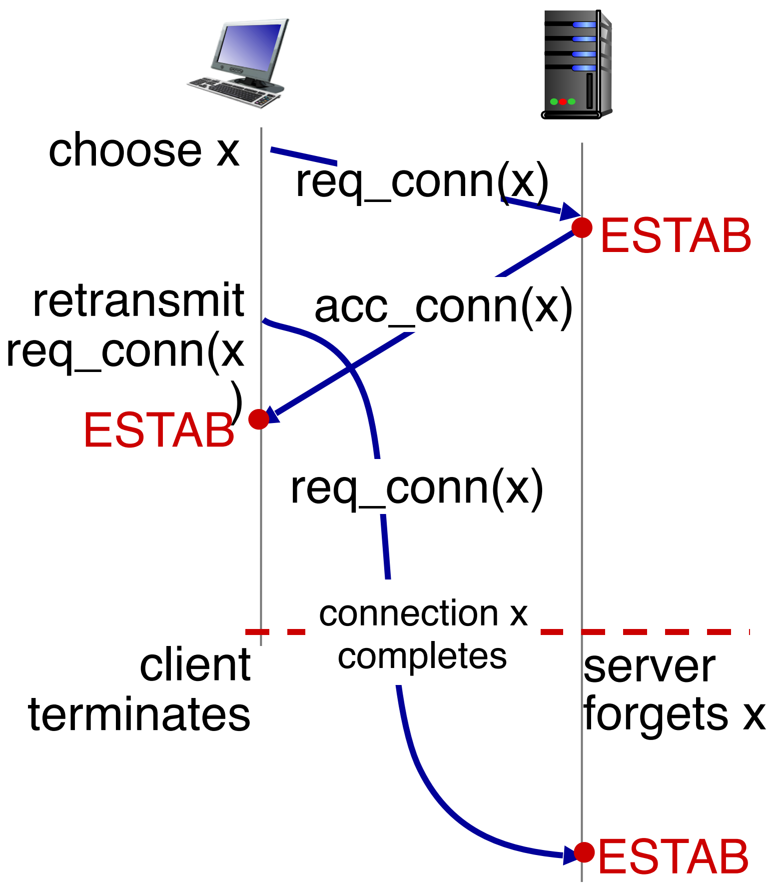


Q: will 2-way handshake always work in network?

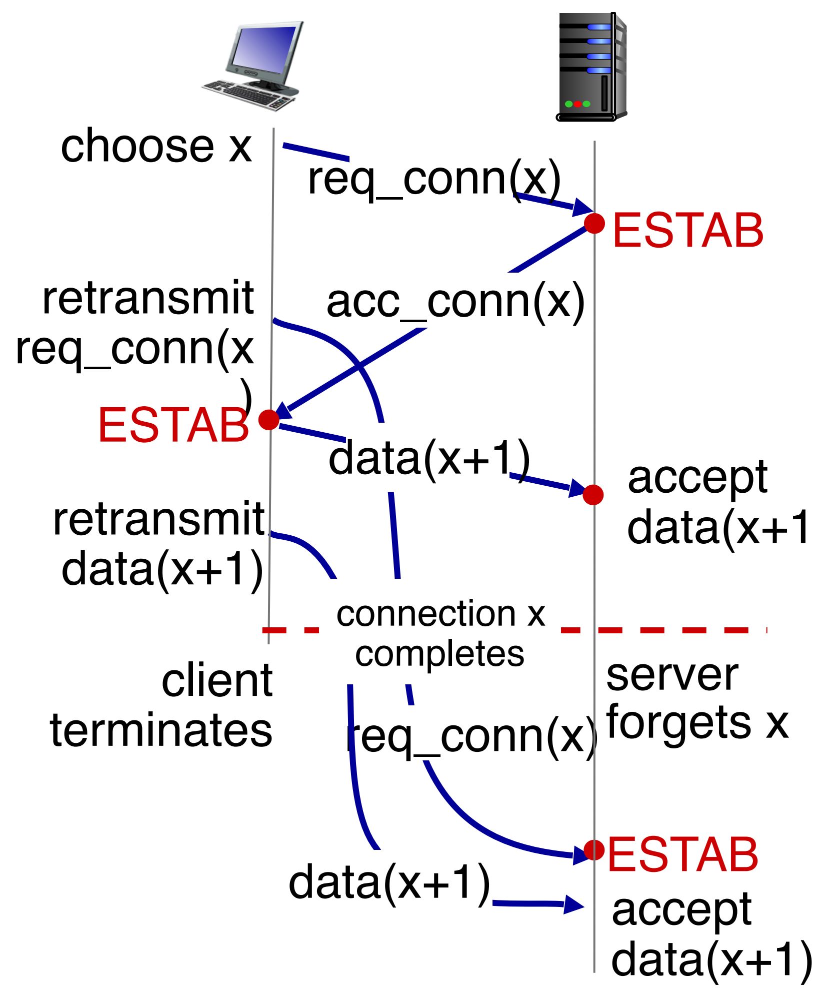
- variable delays
- retransmitted messages (e.g., req_conn(x)) due to message loss
- message reordering
- can't "see" other side

Agreeing to establish a connection

2-way handshake failure scenarios:



half open connection!
(no client!)



Case study: ATM ABR congestion control

•Network-assisted congestion control Example: ATM ABR congestion control

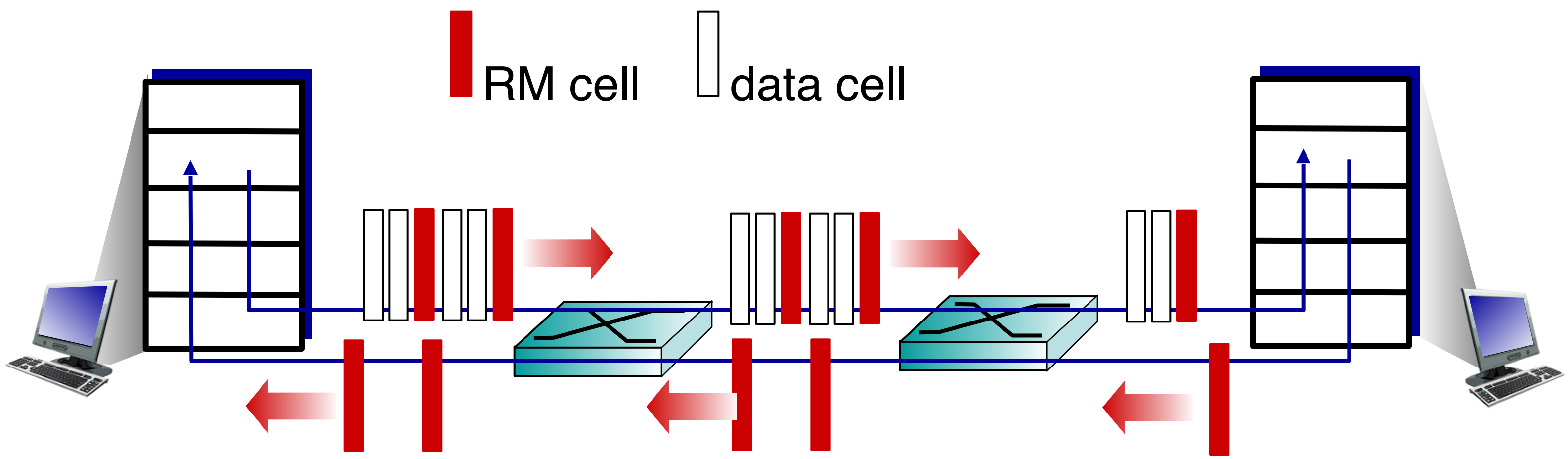
ABR: available bit rate:

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

RM (resource management) cells:

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

Case study: ATM ABR congestion control



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