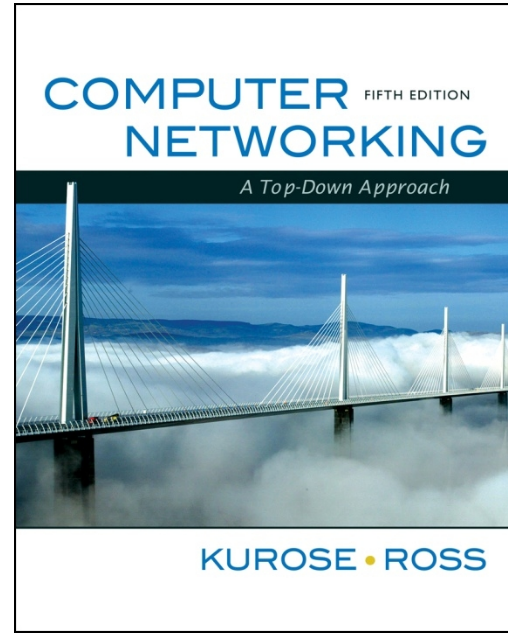
Chapter 3
Transport Layer



A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you 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) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form 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

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Chapter 3: Transport Layer

Our goals:

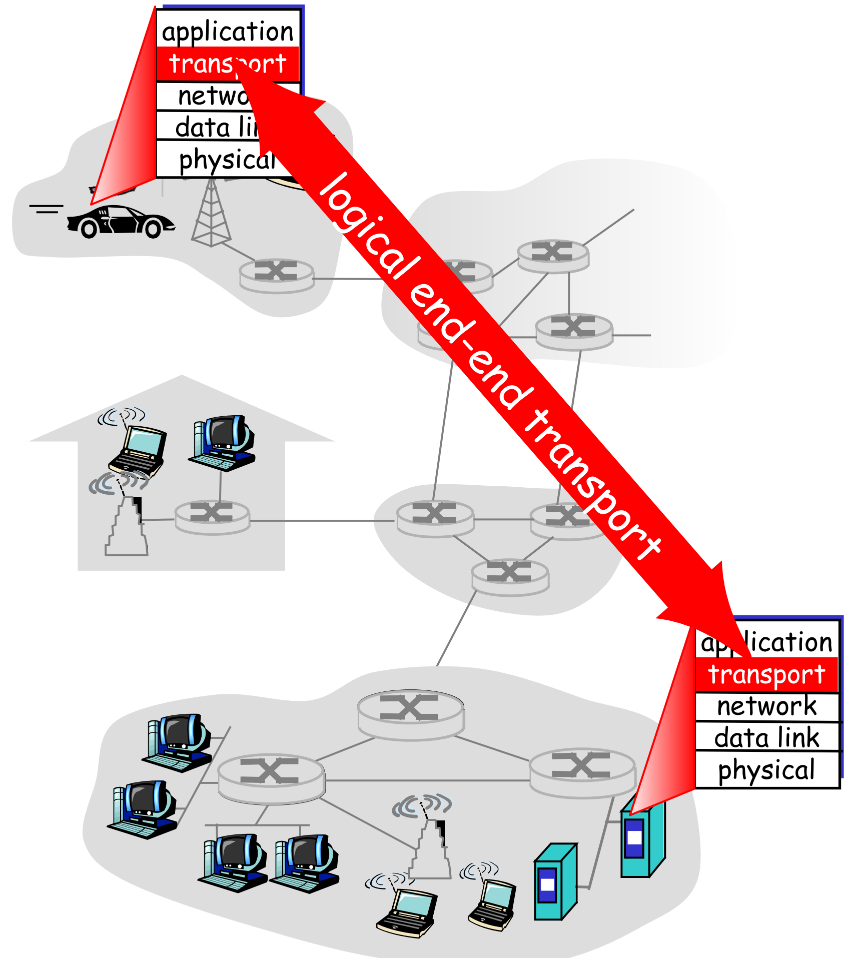
- understand principles behind transport layer services:
  - multiplexing/ demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion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

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

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

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

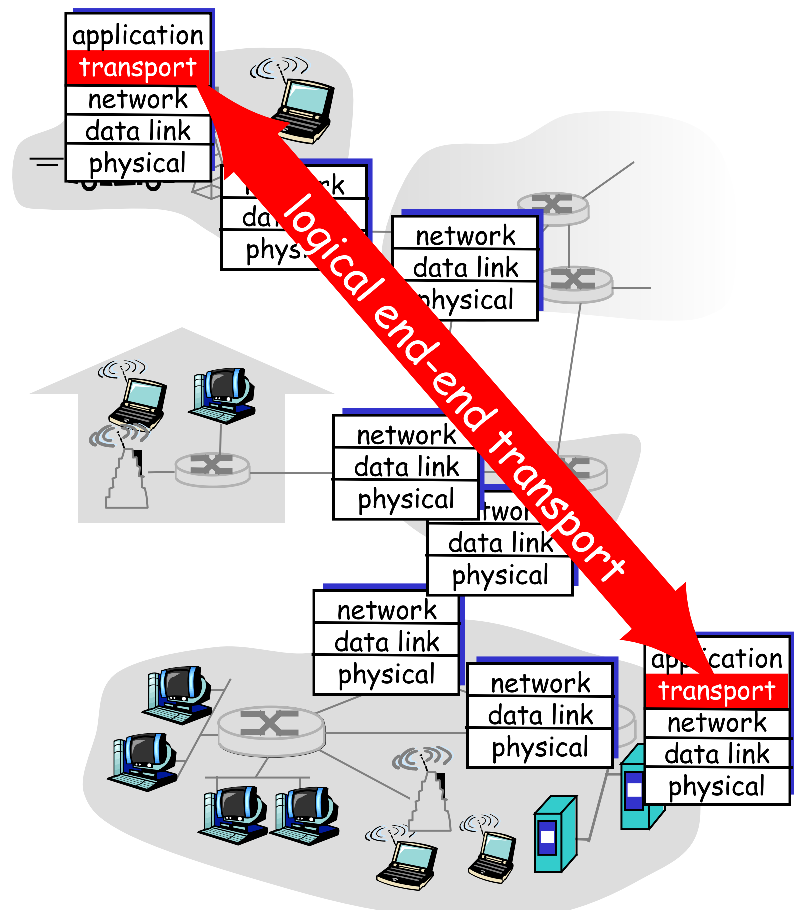
Household analogy:

12 kids sending letters to 12 kids

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



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

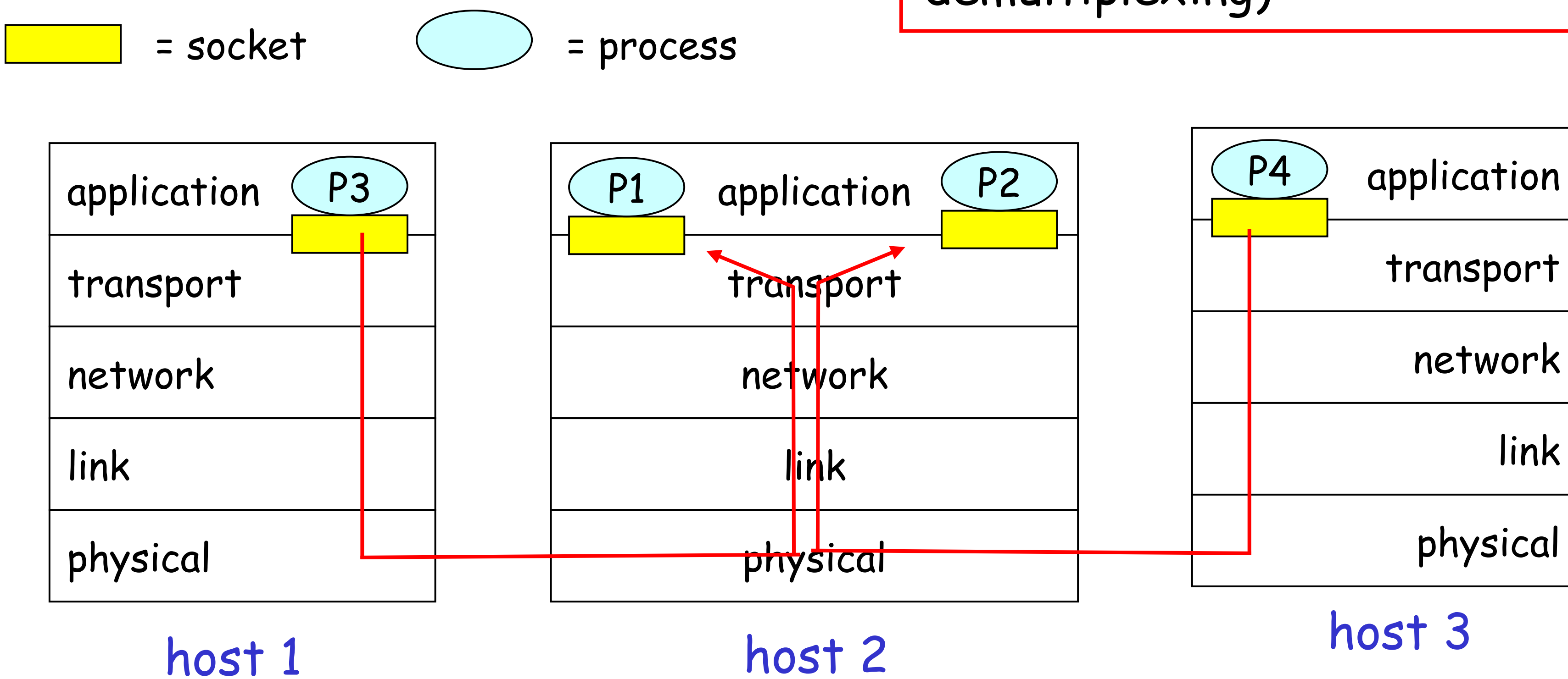
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

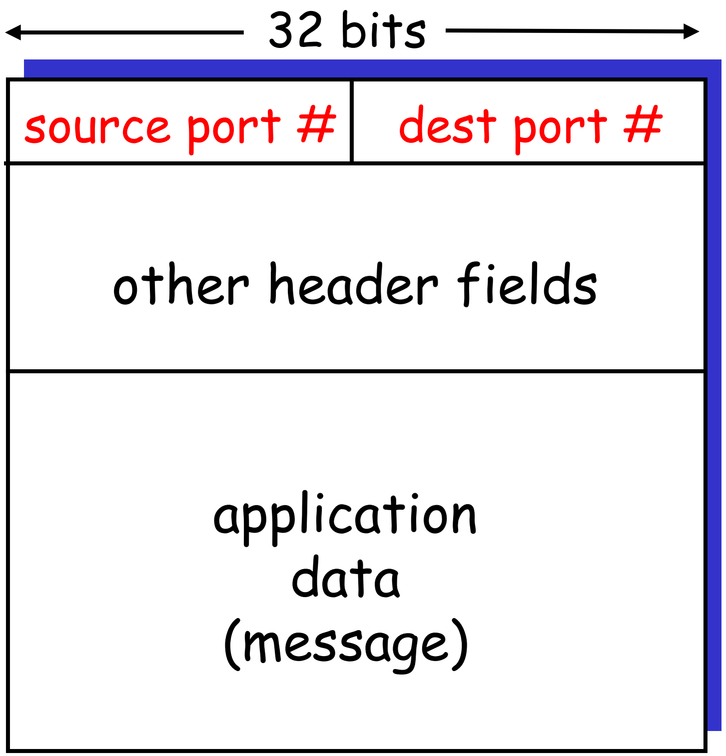
Multiplexing at send host:

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



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(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

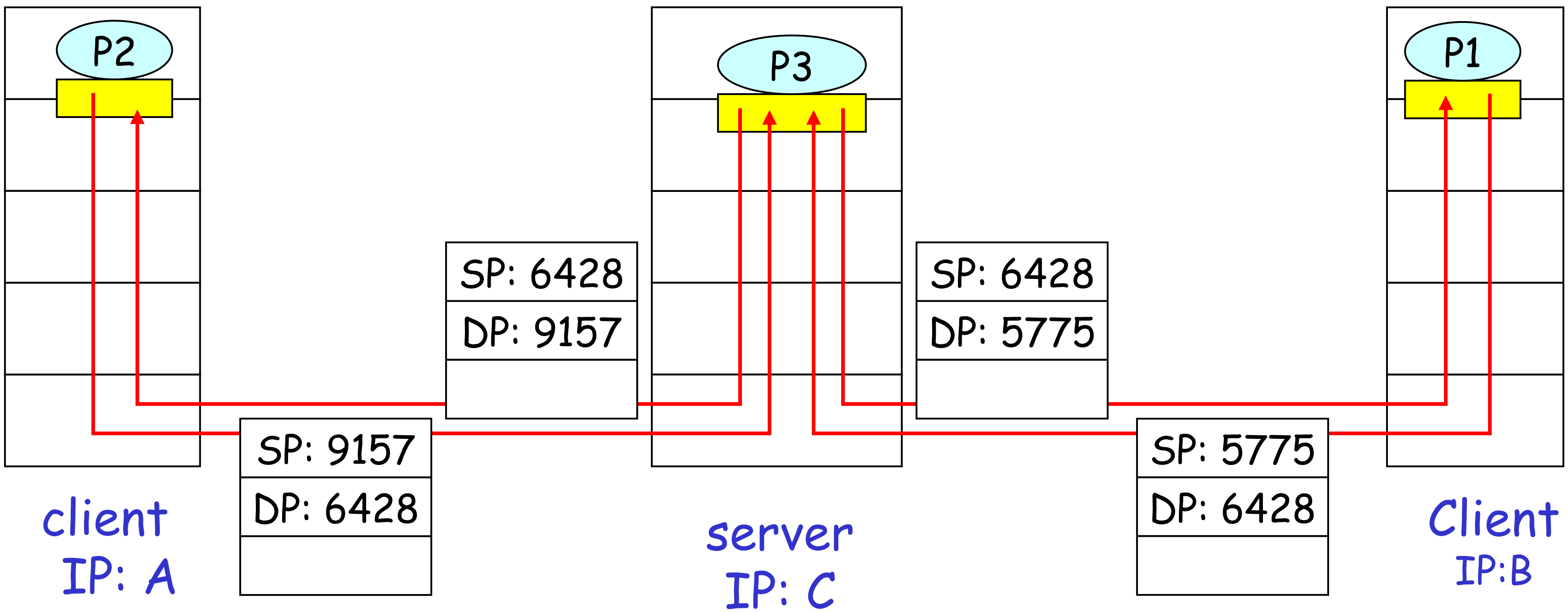
- UDP socket identified by two-tuple:

(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 demux (cont)

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



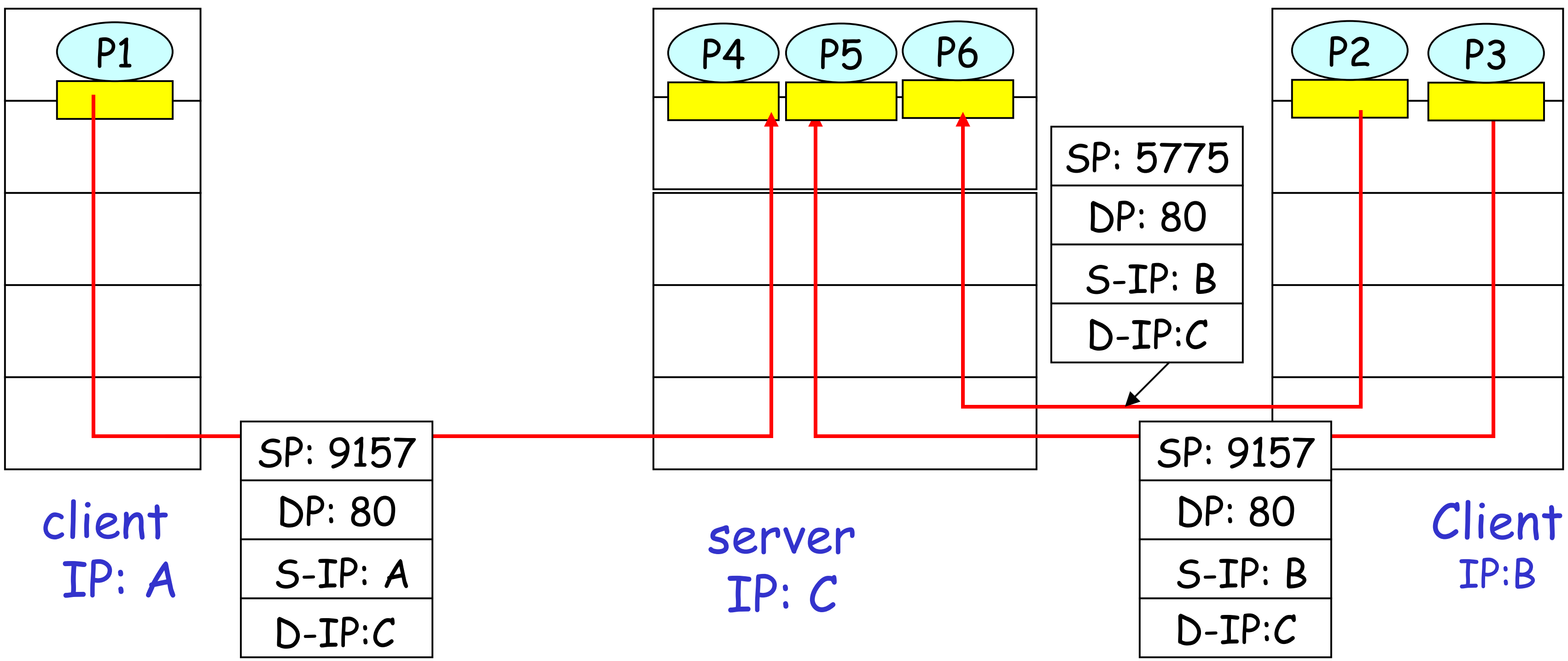
SP provides "return address"

Connection-oriented demux

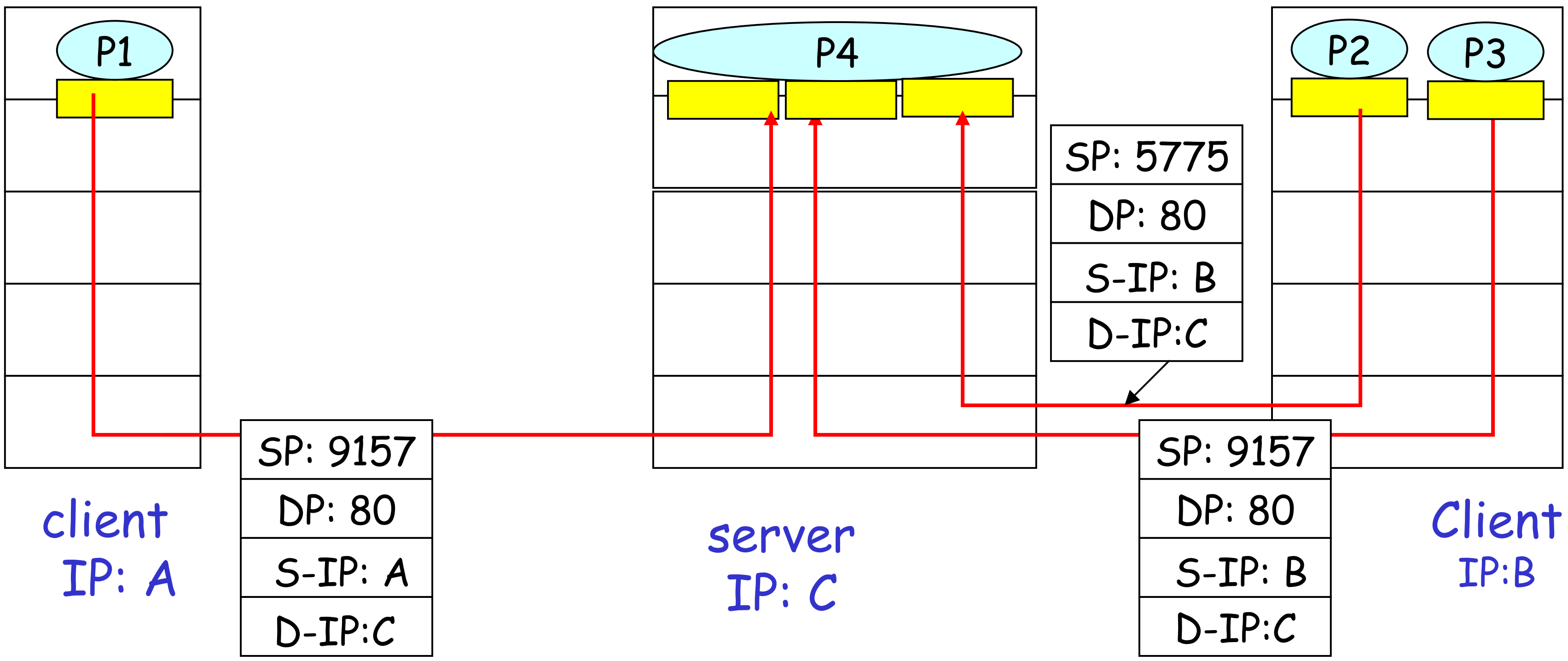
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- receiving host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



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: User Datagram Protocol [RFC 768]

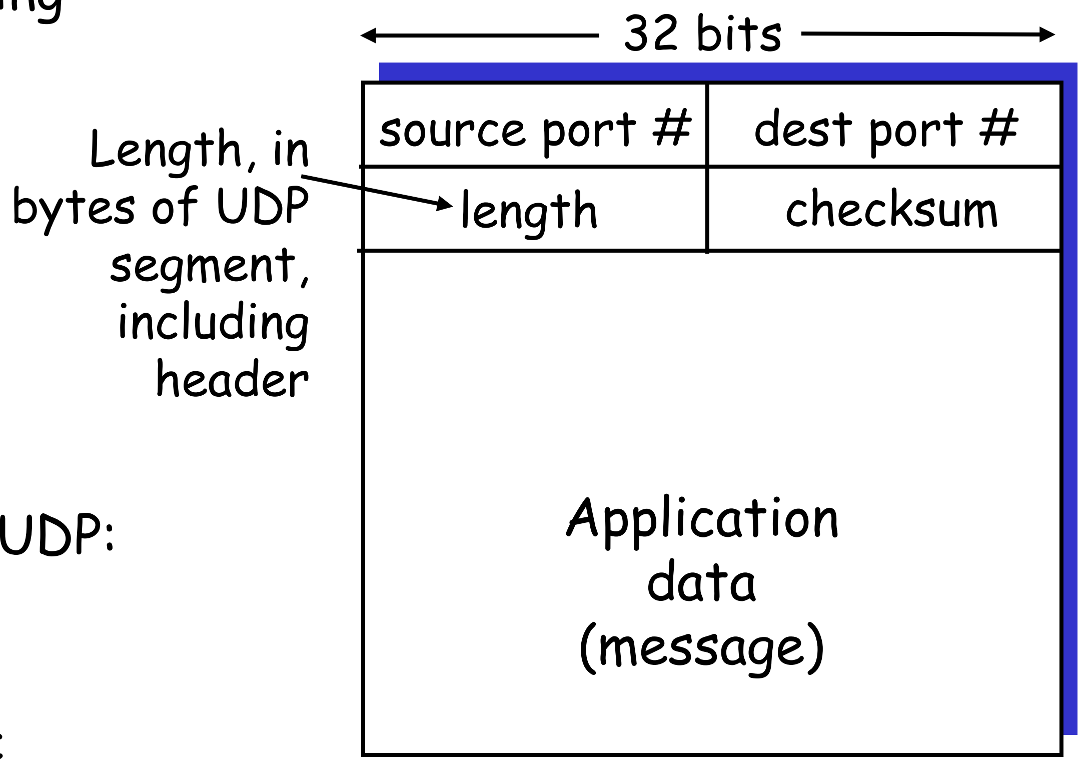
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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

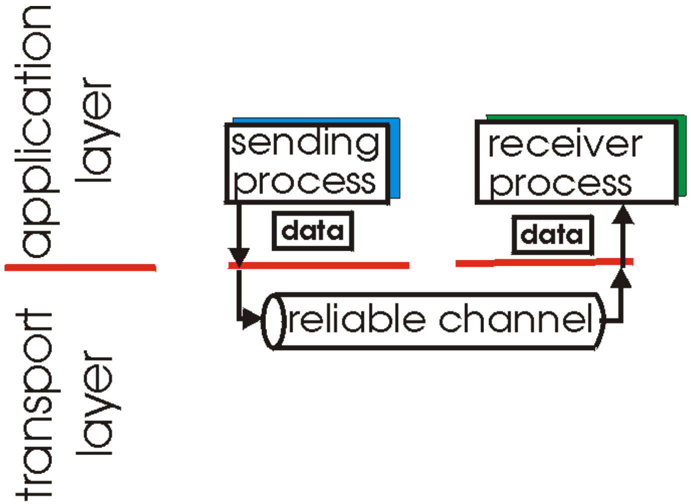
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

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

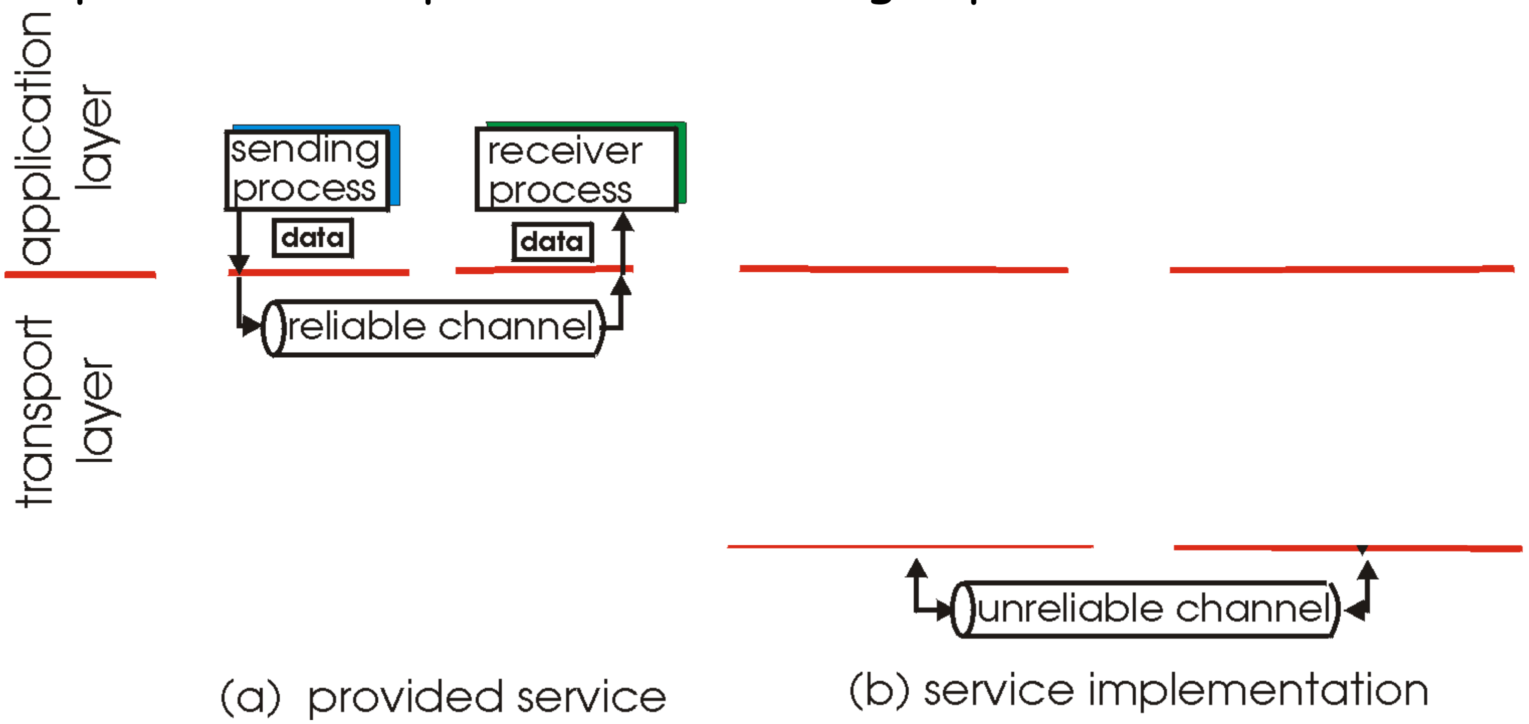


(a) provided service

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

Principles of Reliable data transfer

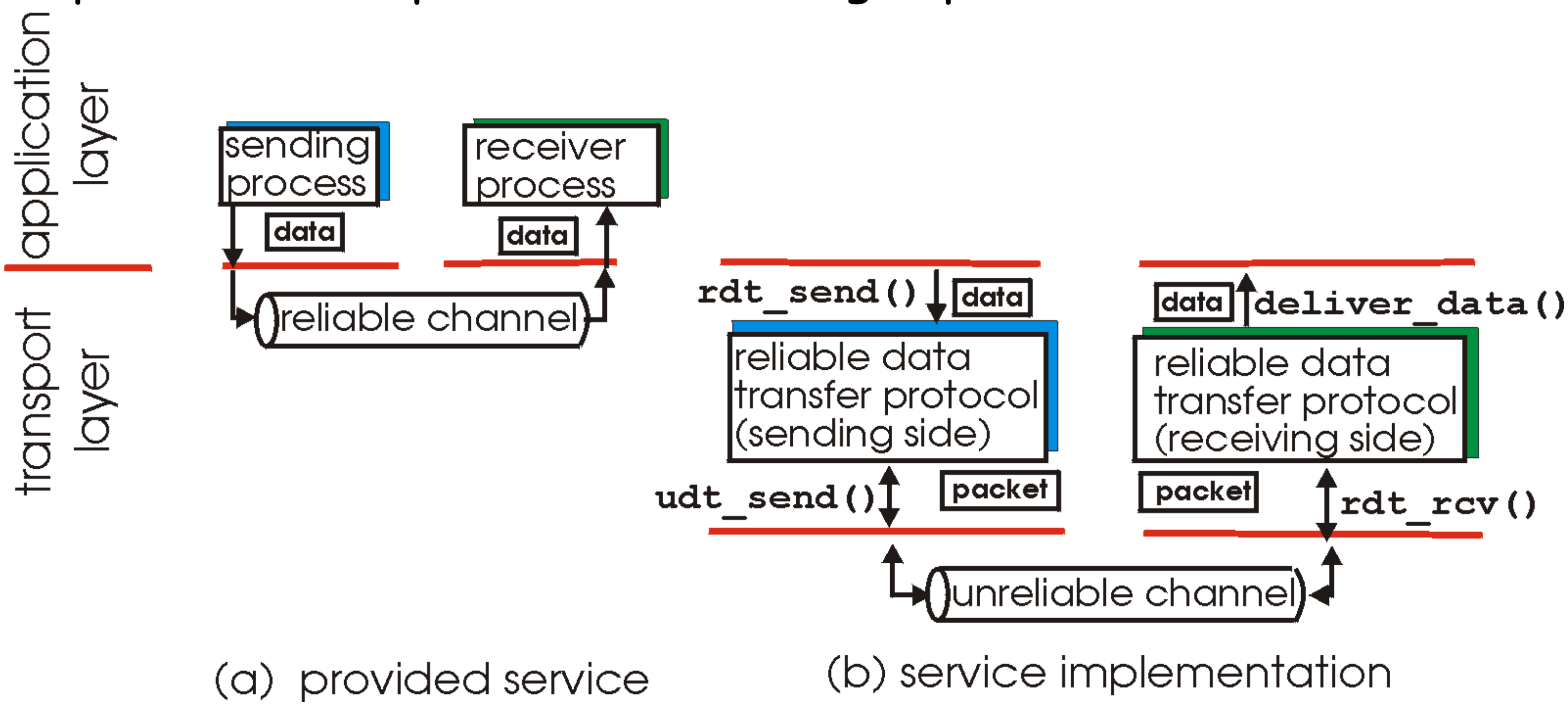
- important in app., transport, link layers
- top-10 list of important networking topics!



- 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!

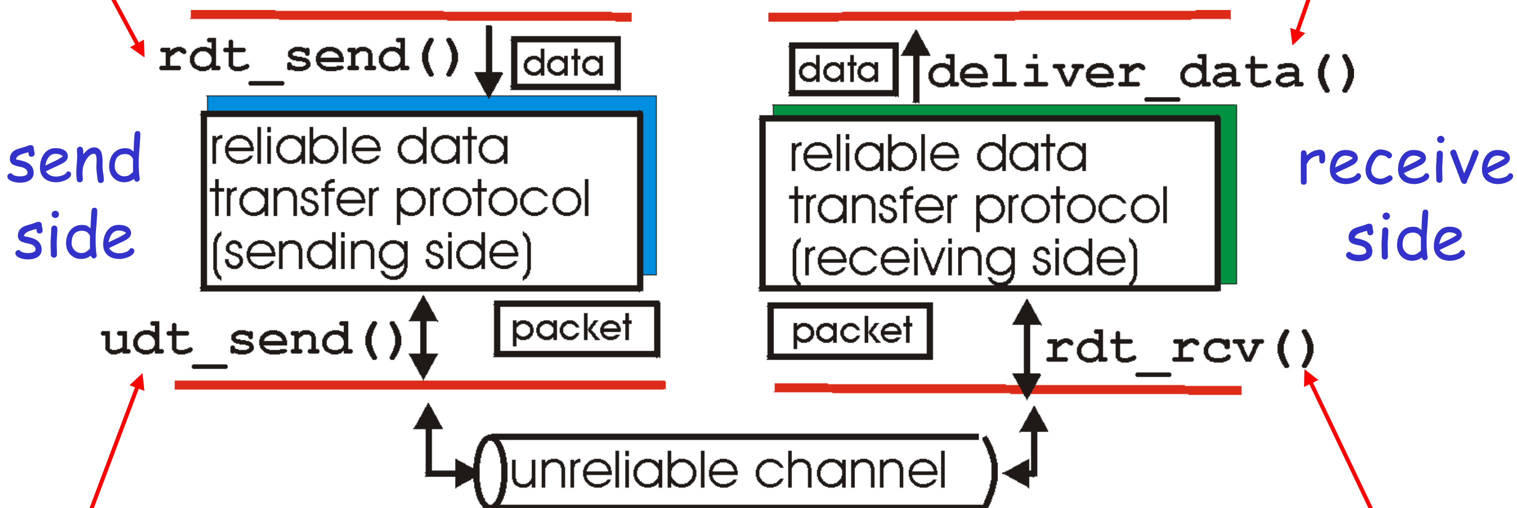


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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



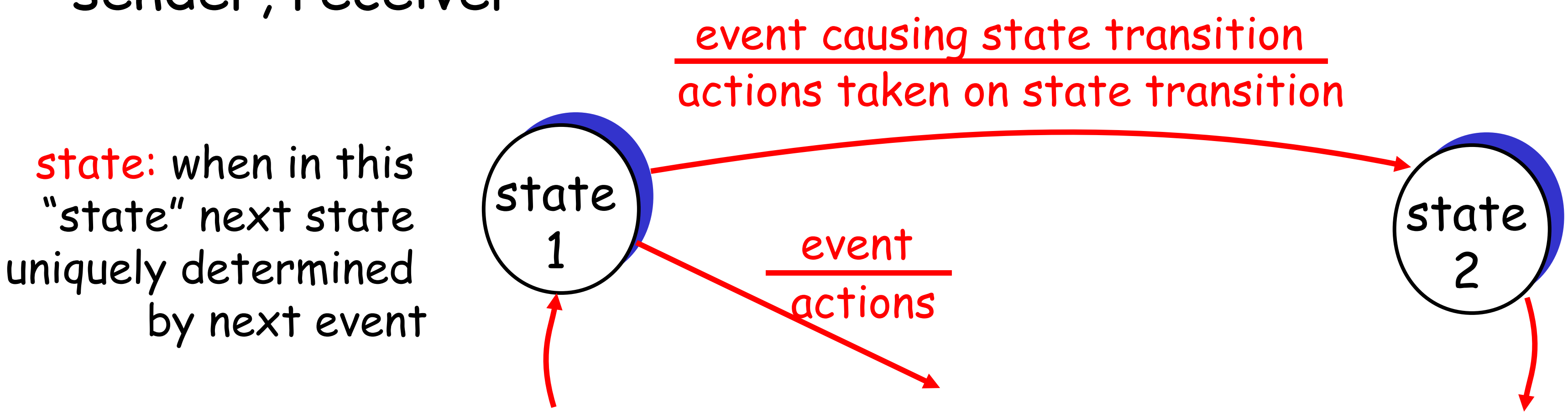
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

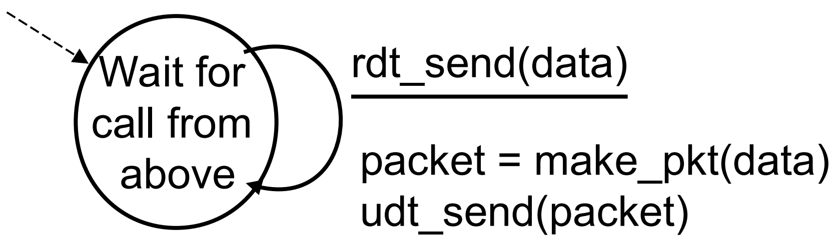
We'll:

- 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

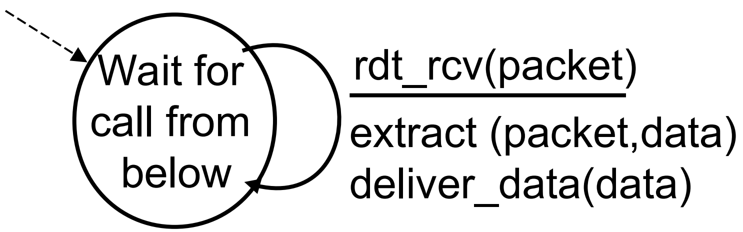


Rdt1.0: reliable transfer over a reliable channel

- 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

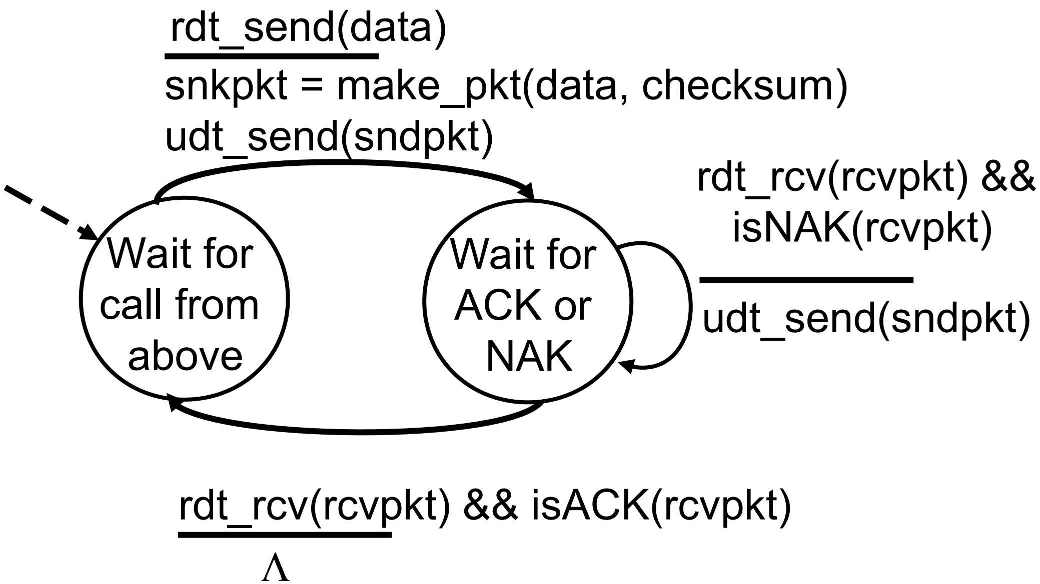


receiver

Rdt2.0: channel with bit errors

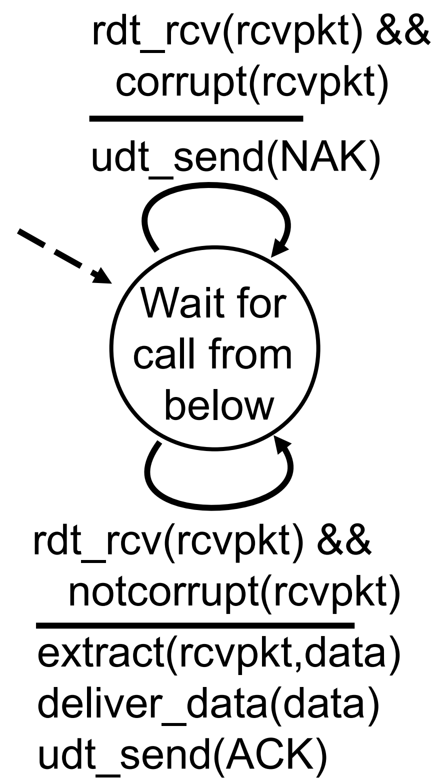
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors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

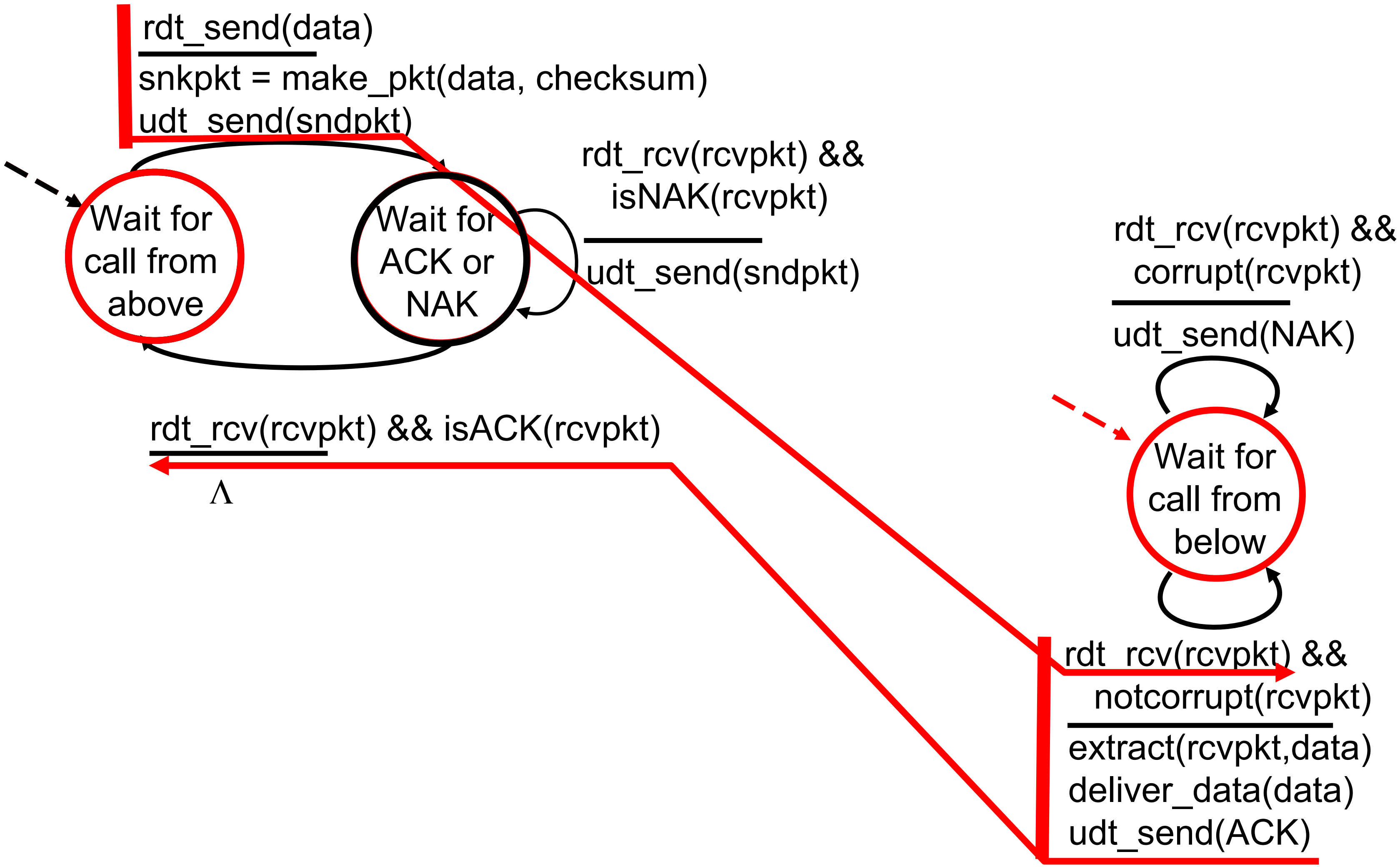


sender

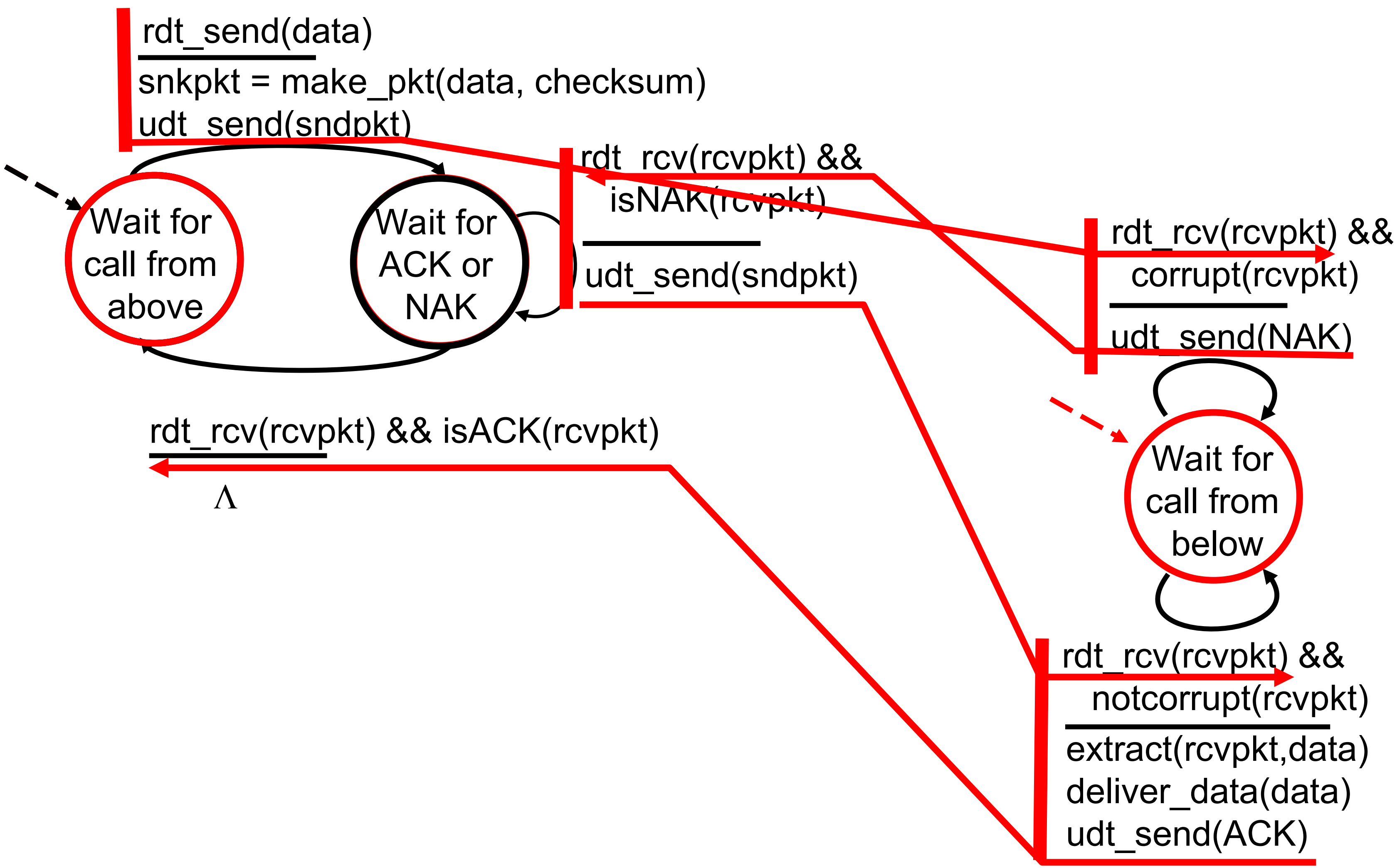
receiver



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

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

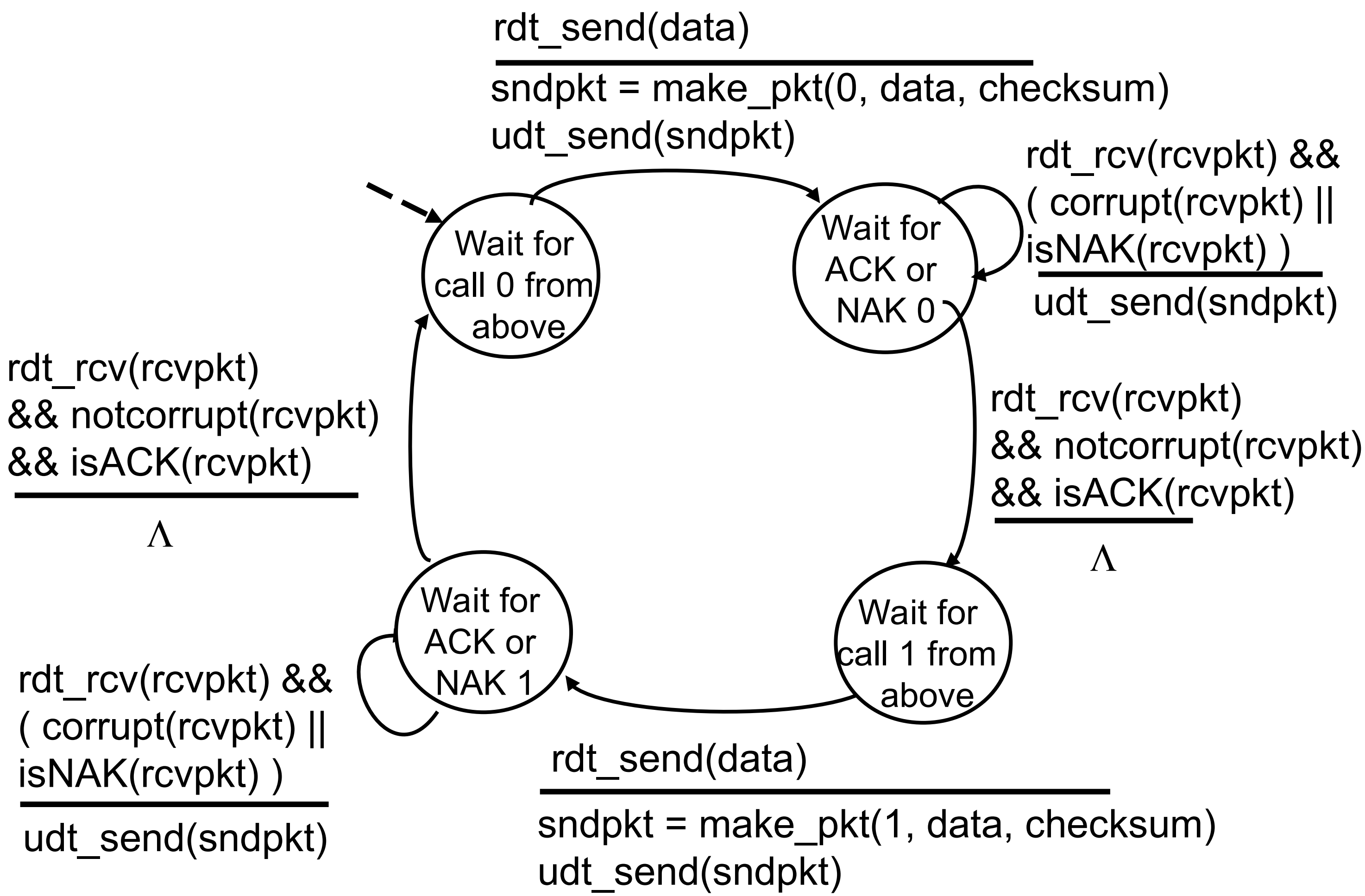
Handling duplicates:

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

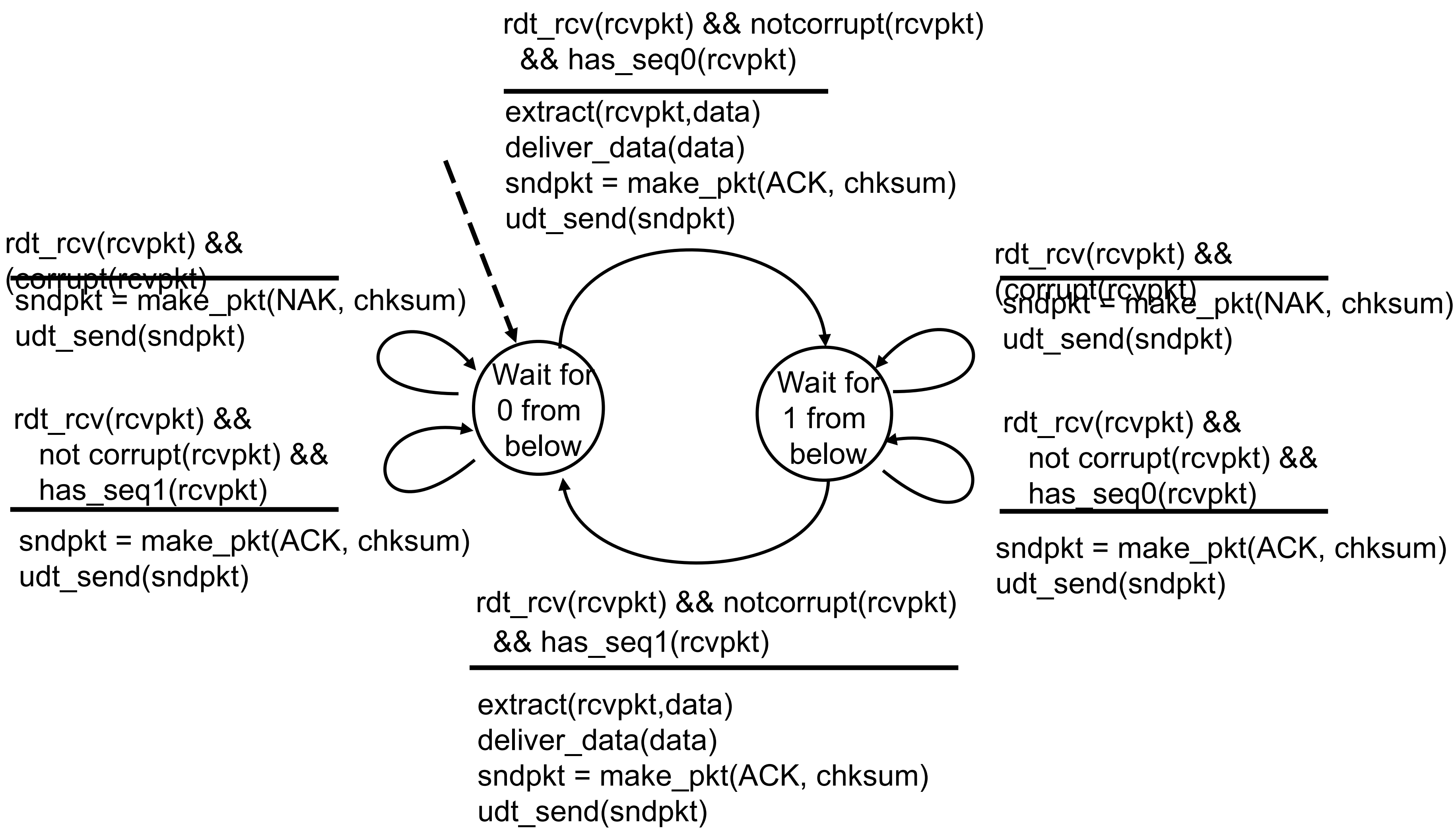
stop and wait

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. #

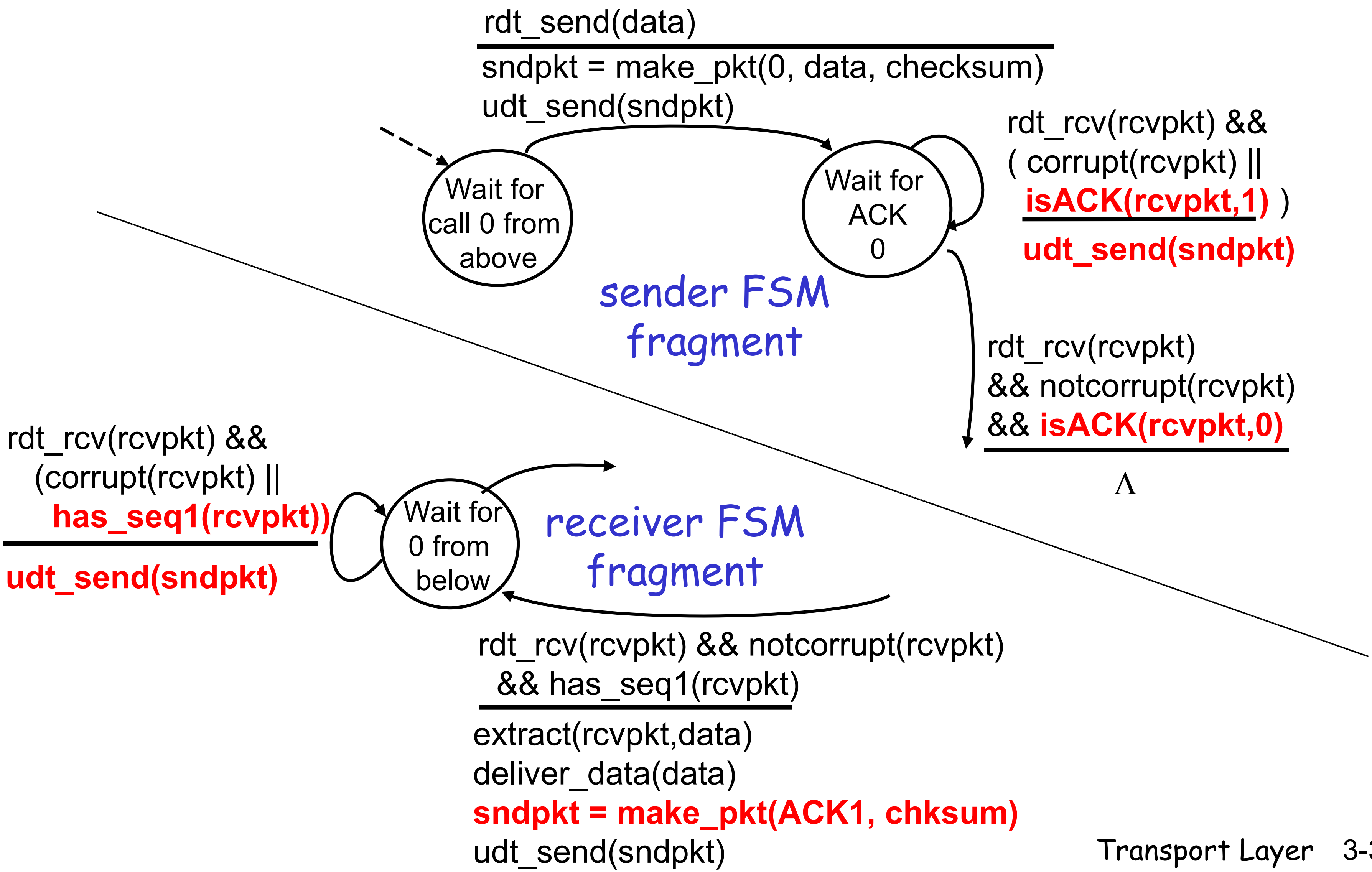
Receiver:

- 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

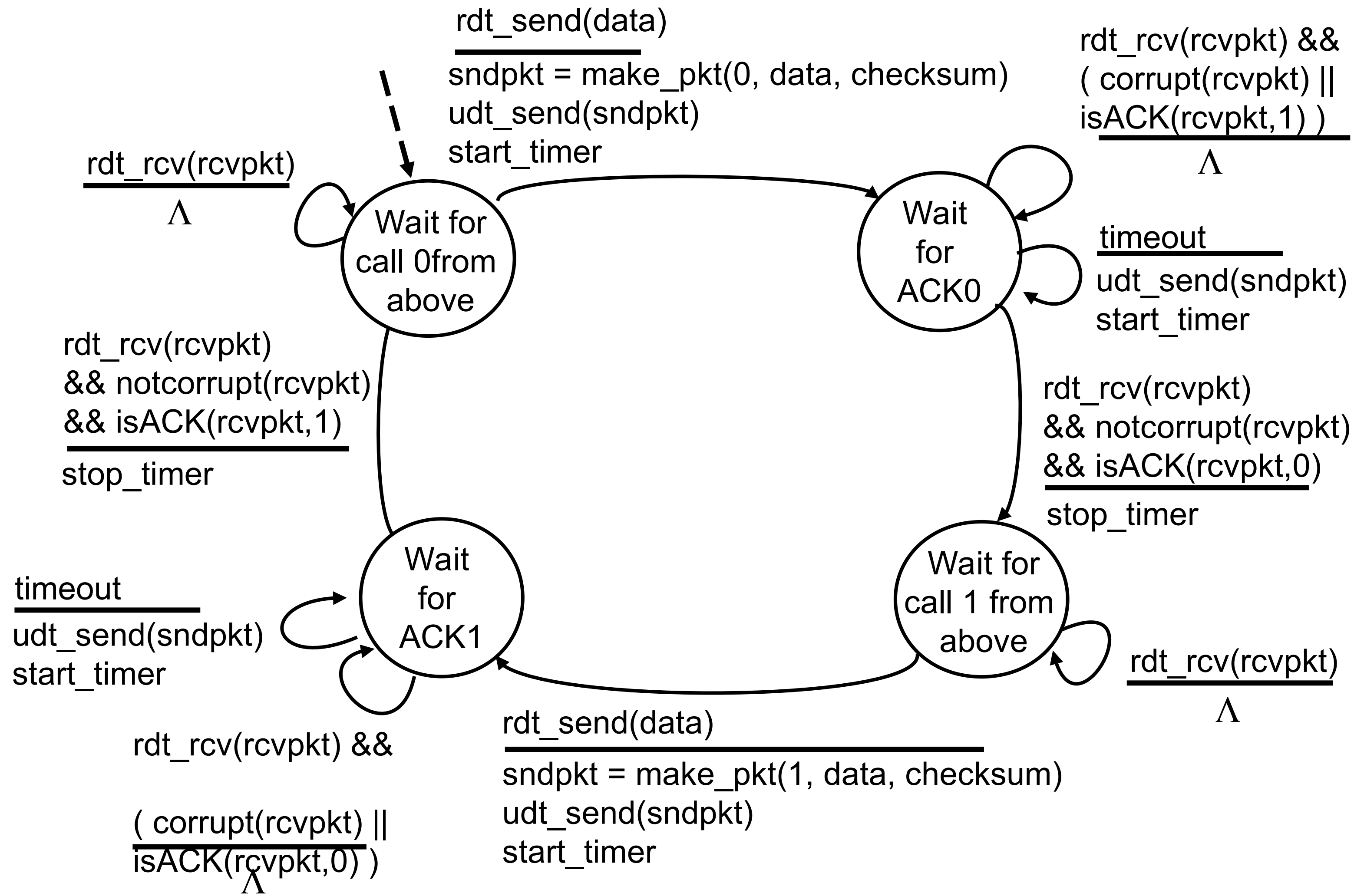
New assumption: underlying channel can also lose packets (data or ACKs)

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

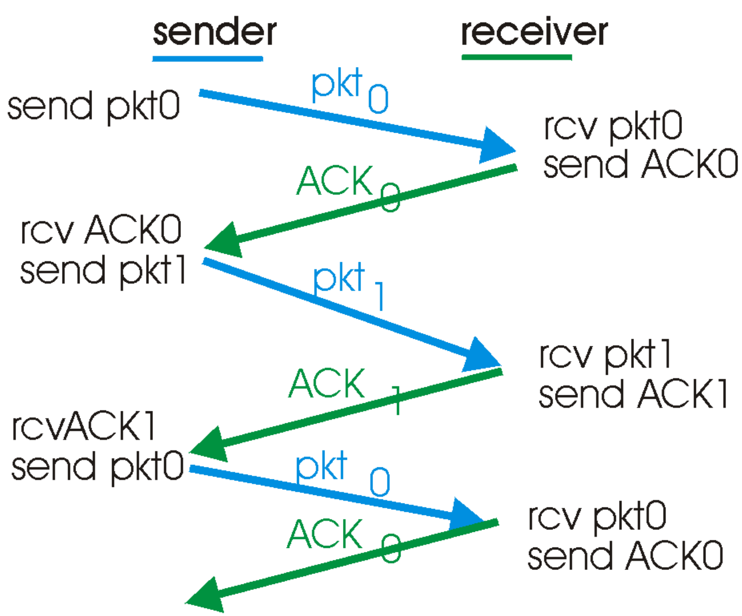
Approach: sender waits "reasonable" amount of time for ACK

- 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

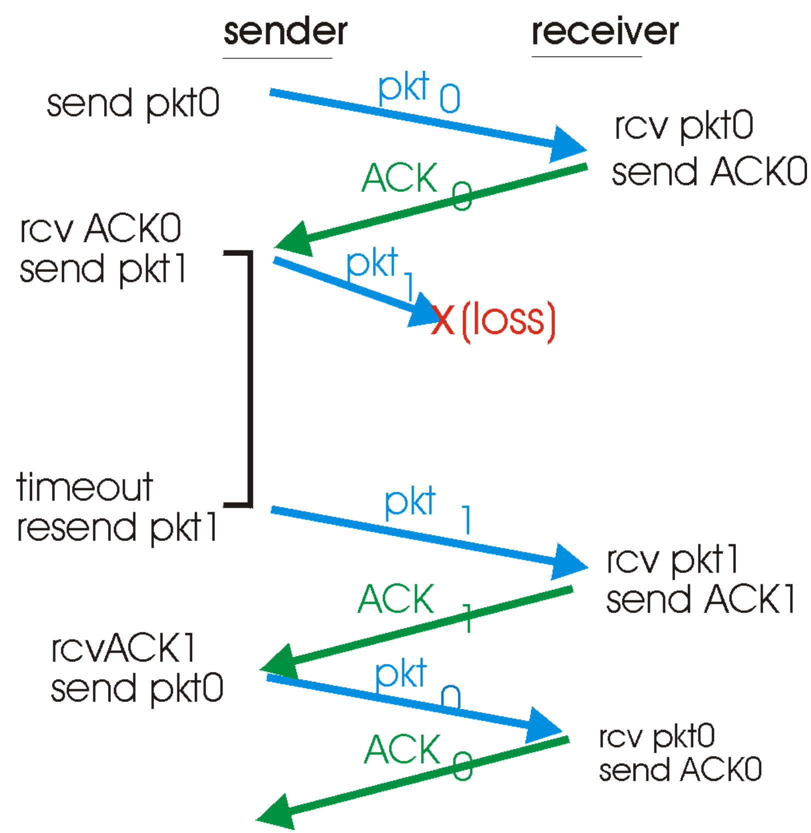
rdt3.0 sender



rdt3.0 in action

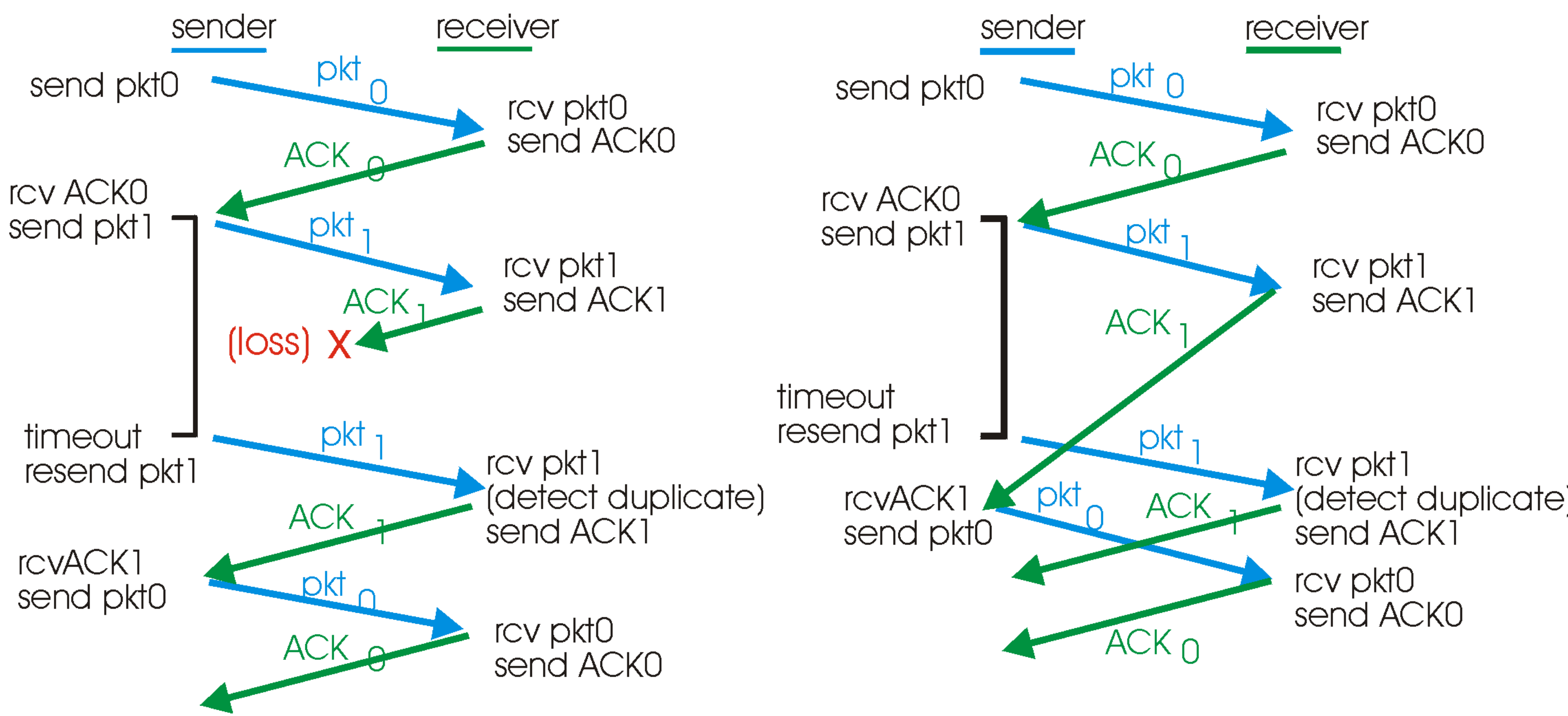


(a) operation with no loss



(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:

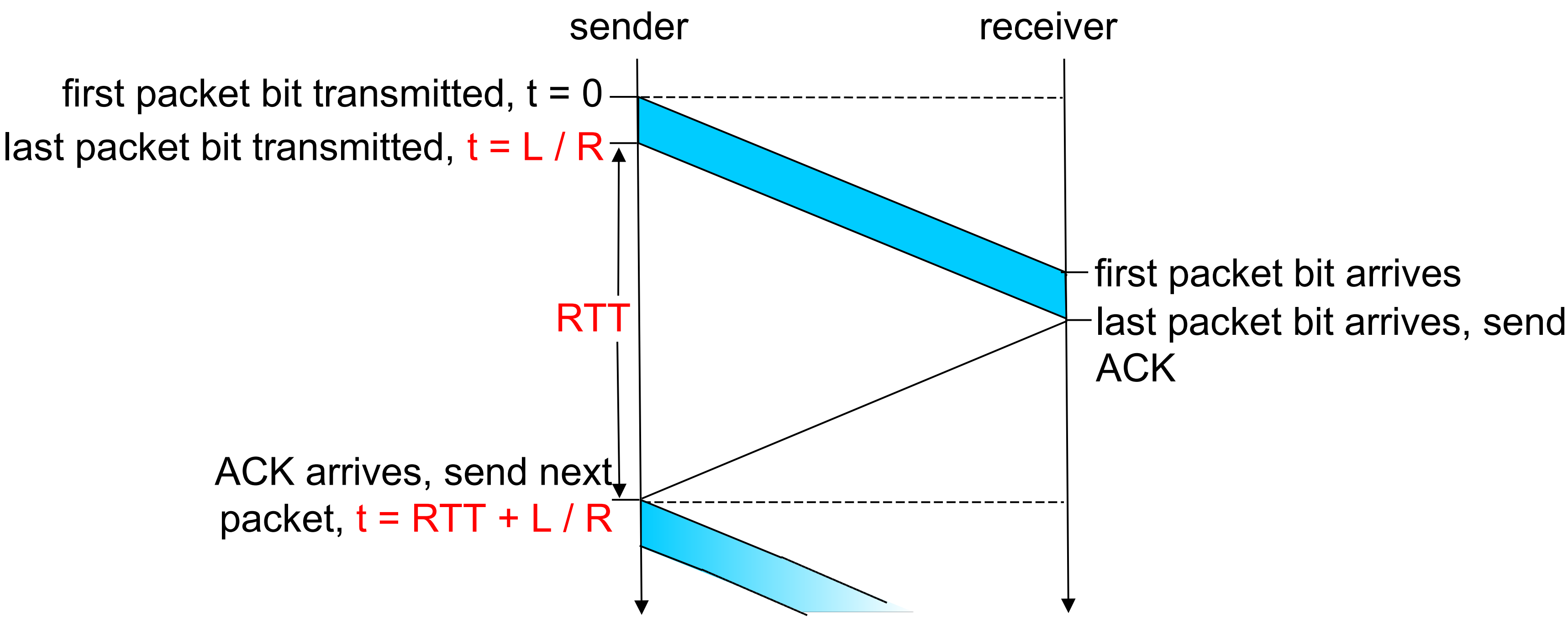
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{ bps}} = 8\text{ microseconds}$$

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

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

  - 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

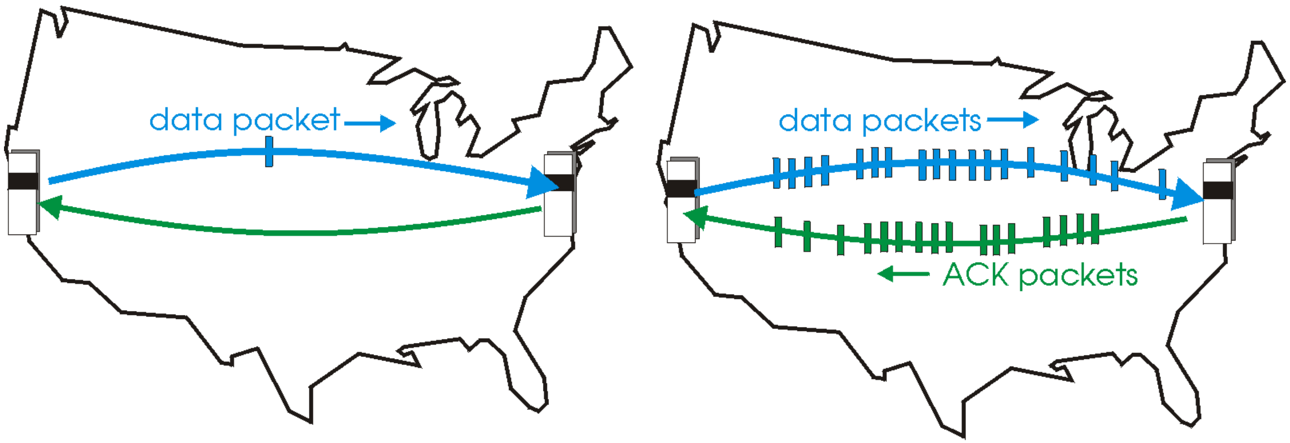


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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

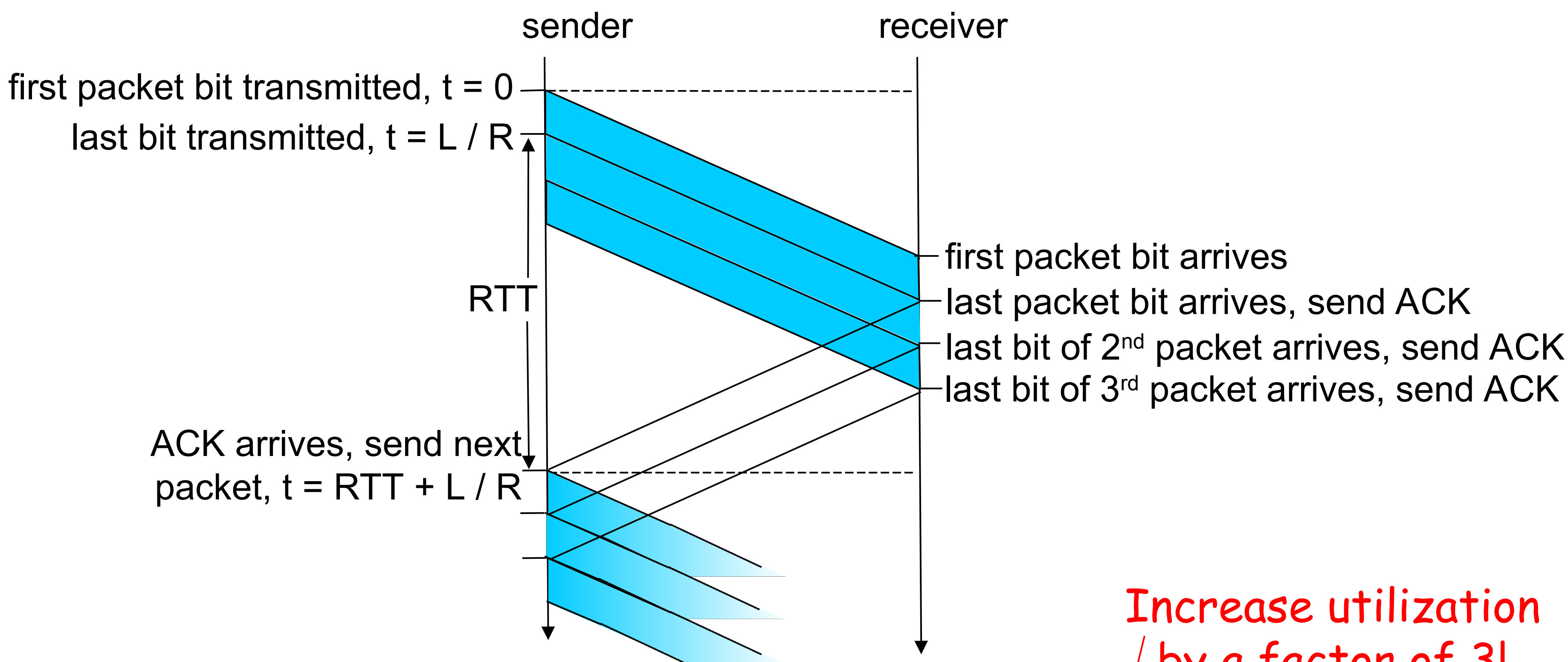


(a) a stop-and-wait protocol in operation

(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!

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

Pipelining Protocols

Go-back-N: overview

- *sender:* up to N unACKed pkts in pipeline
- *receiver:* only sends cumulative ACKs
  - doesn't ACK pkt if there's a gap
- *sender:* has timer for oldest unACKed pkt
  - if timer expires: retransmit all unACKed packets

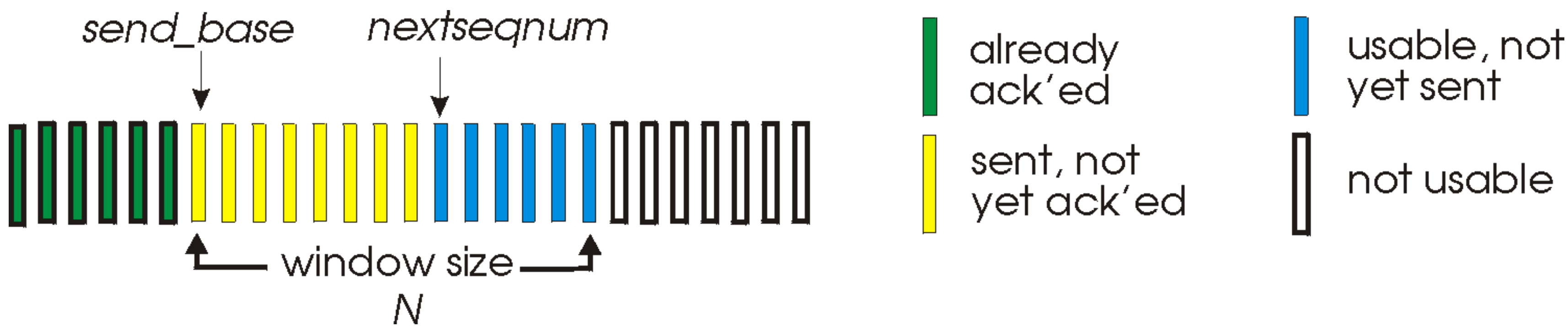
Selective Repeat: overview

- *sender:* up to N unACKed packets in pipeline
- *receiver:* ACKs individual pkts
- *sender:* maintains timer for each unACKed pkt
  - if timer expires: retransmit only unACKed packet

Go-Back-N

Sender:

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

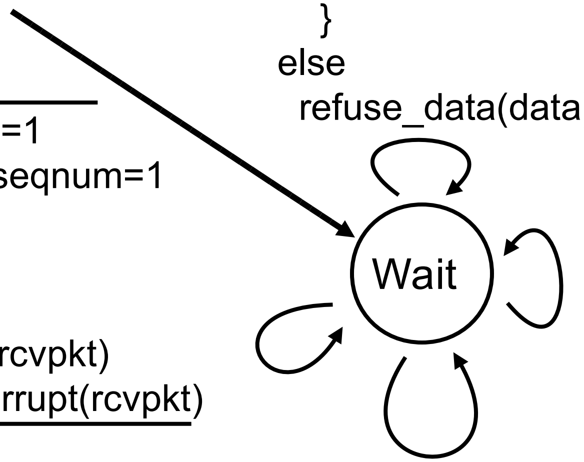


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM

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

```
Λ
____________
base=1
nextseqnum=1
```

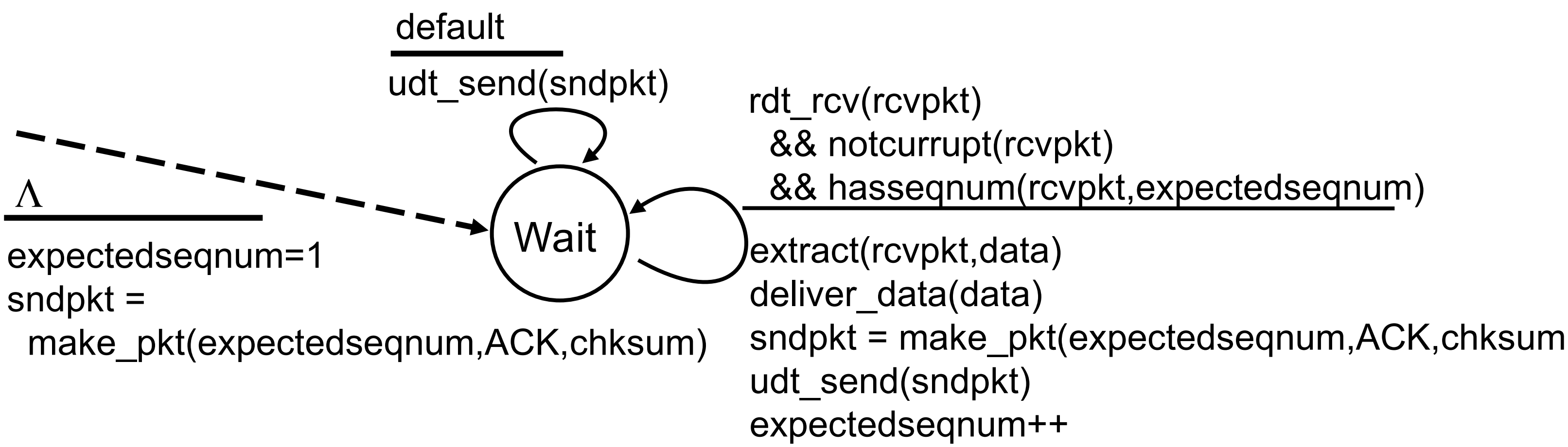



```
timeout
________________________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

```
rdt_rcv(rcvpkt)
   && corrupt(rcvpkt)
______________________
```

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

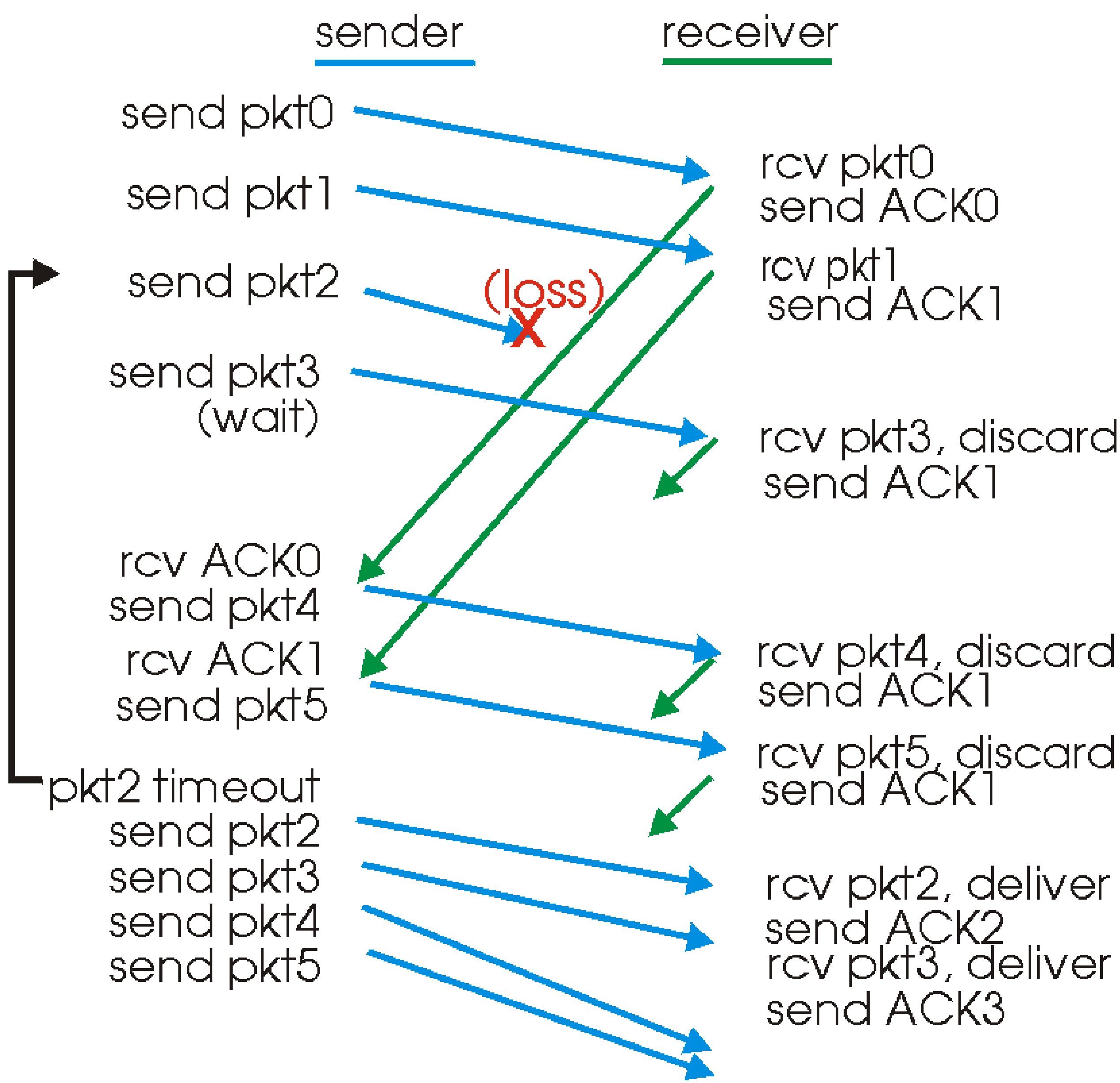
GBN: receiver extended FSM



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

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

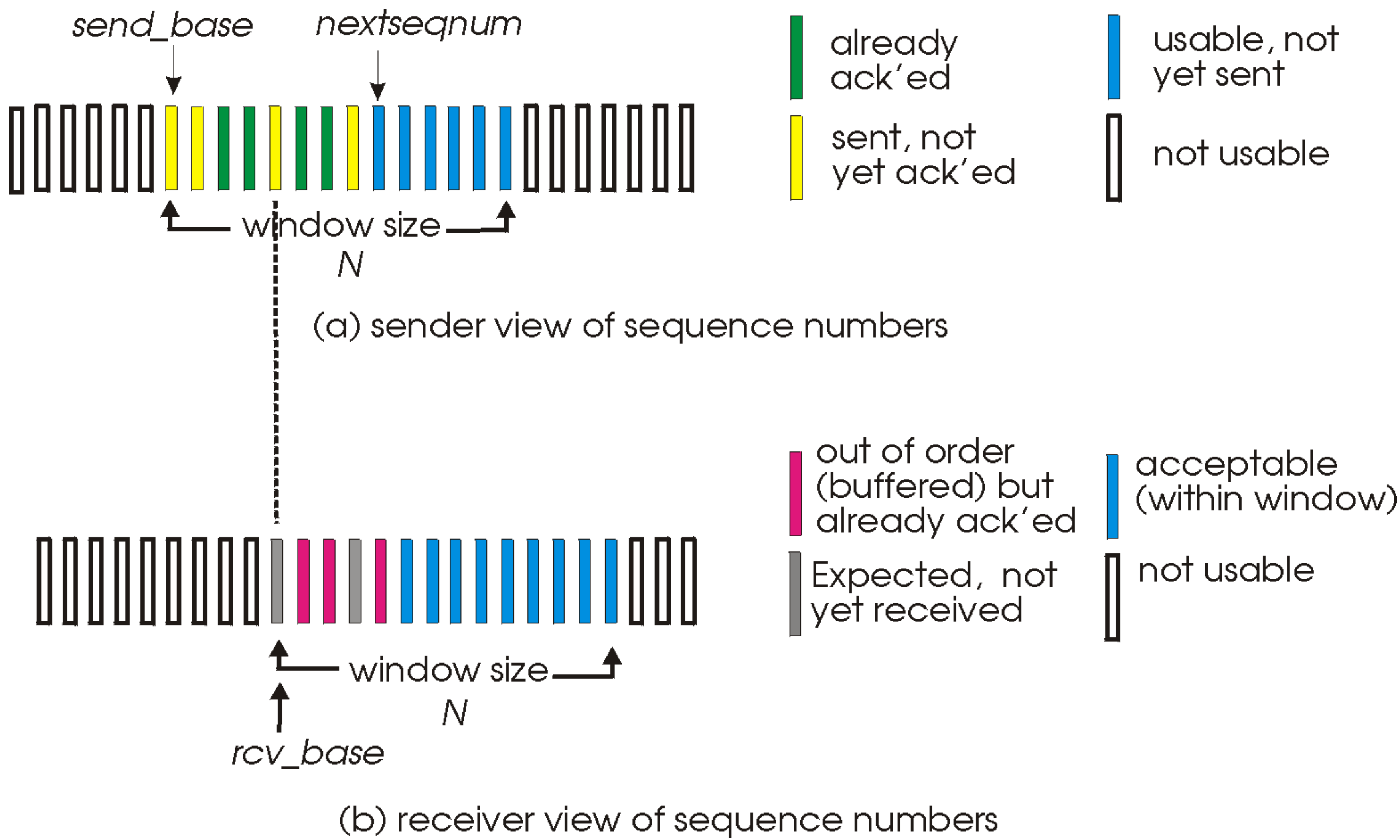
GBN in action



Selective Repeat

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

Selective repeat: sender, receiver windows



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

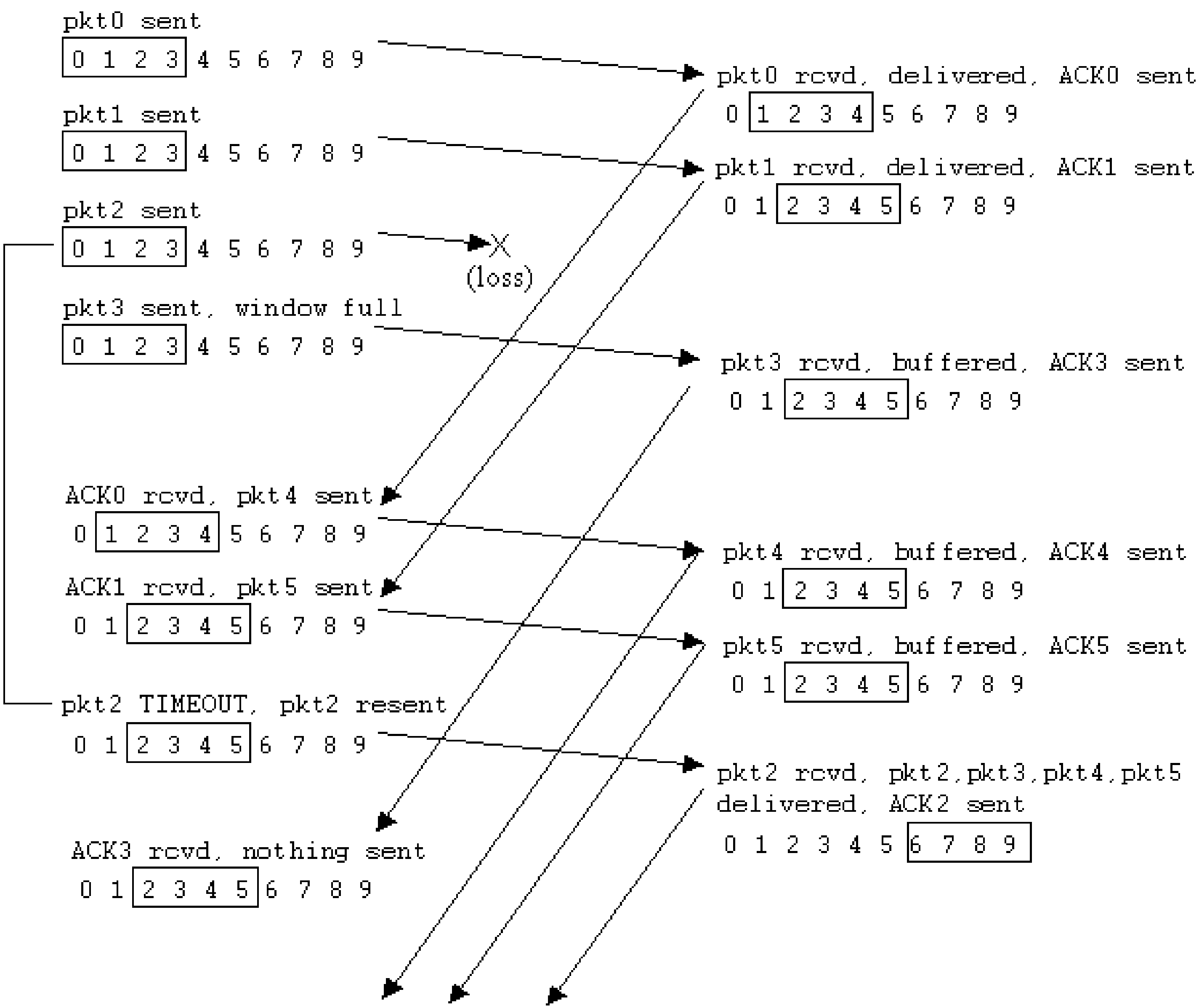
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

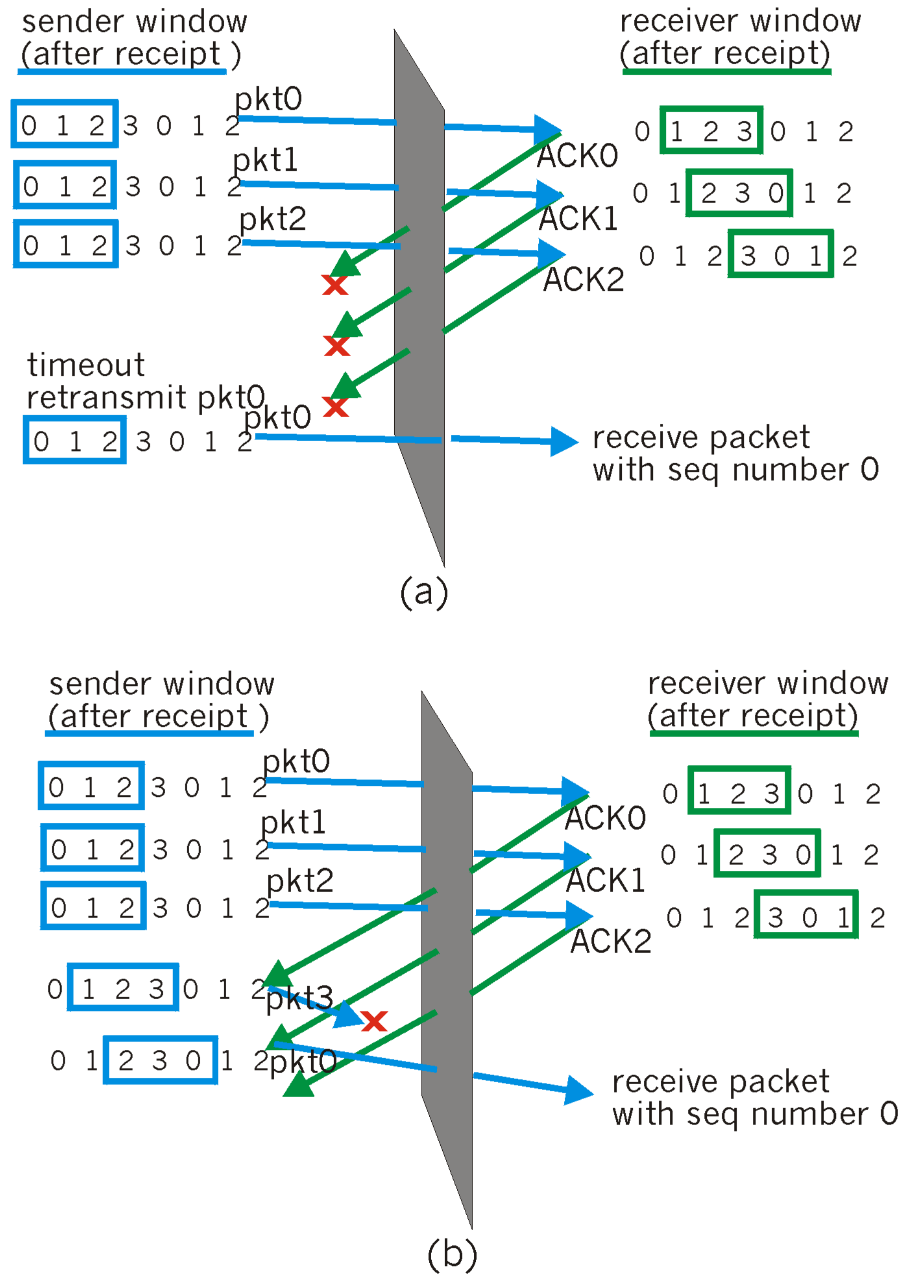
Selective repeat in action



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?

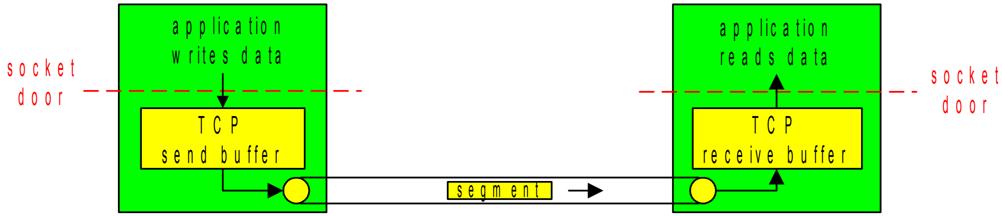
Chapter 3 outline

- 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

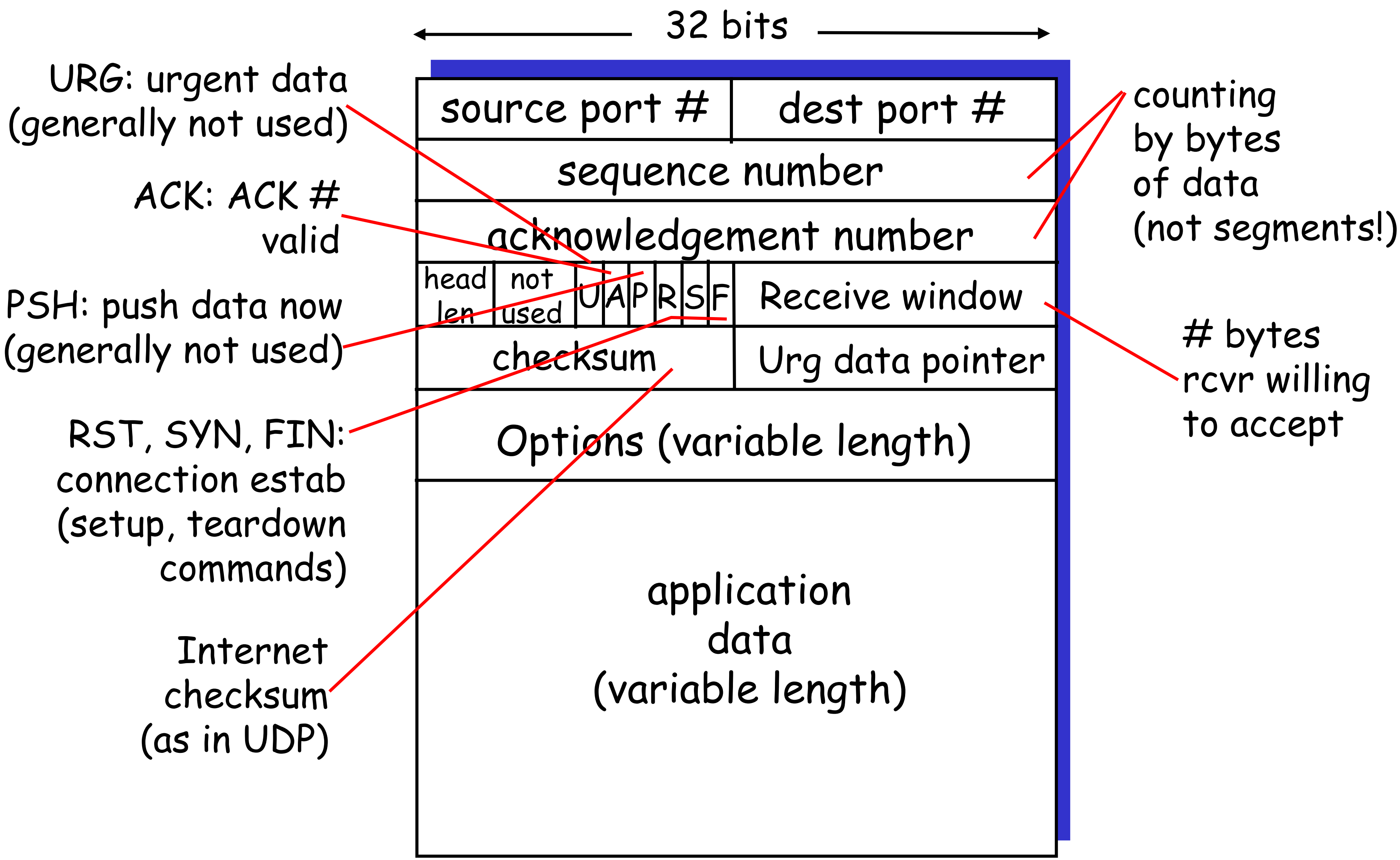
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - 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:

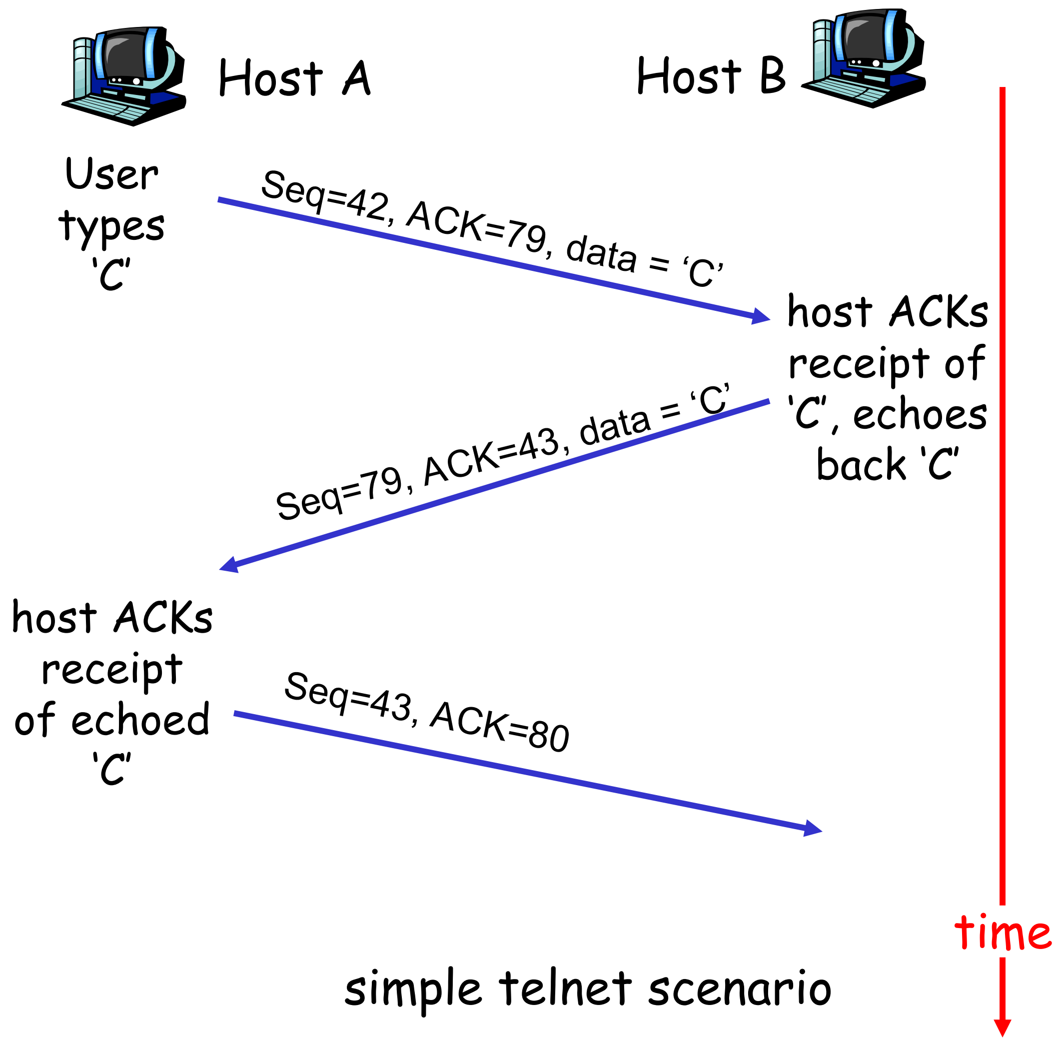
- byte stream "number" of first byte in segment's data

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 implementer



simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

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

Q: how to estimate RTT?

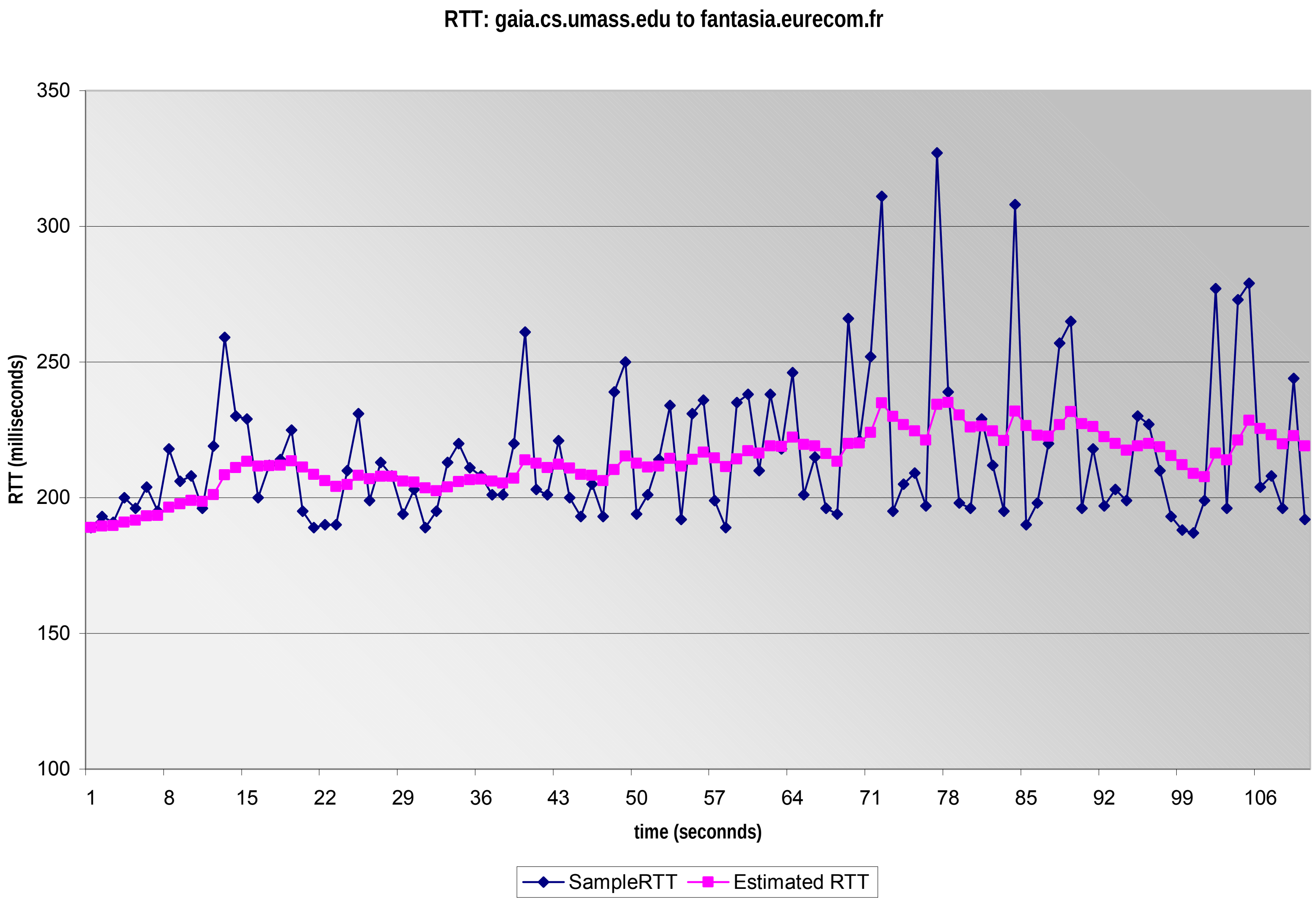
- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current `SampleRTT`

TCP Round Trip Time and Timeout

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

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

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

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

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

(typically, β = 0.25)

Then set timeout interval:

TimeoutInterval = EstimatedRTT + 4*DevRTT

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
- 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

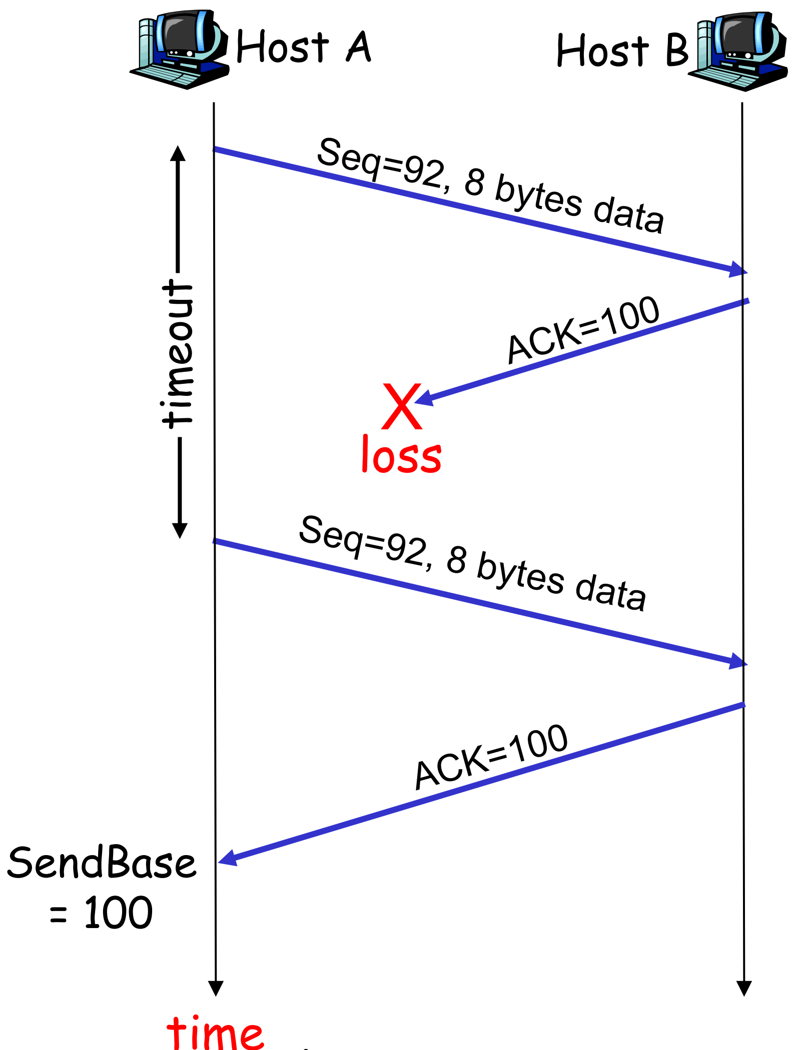
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

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

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

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

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

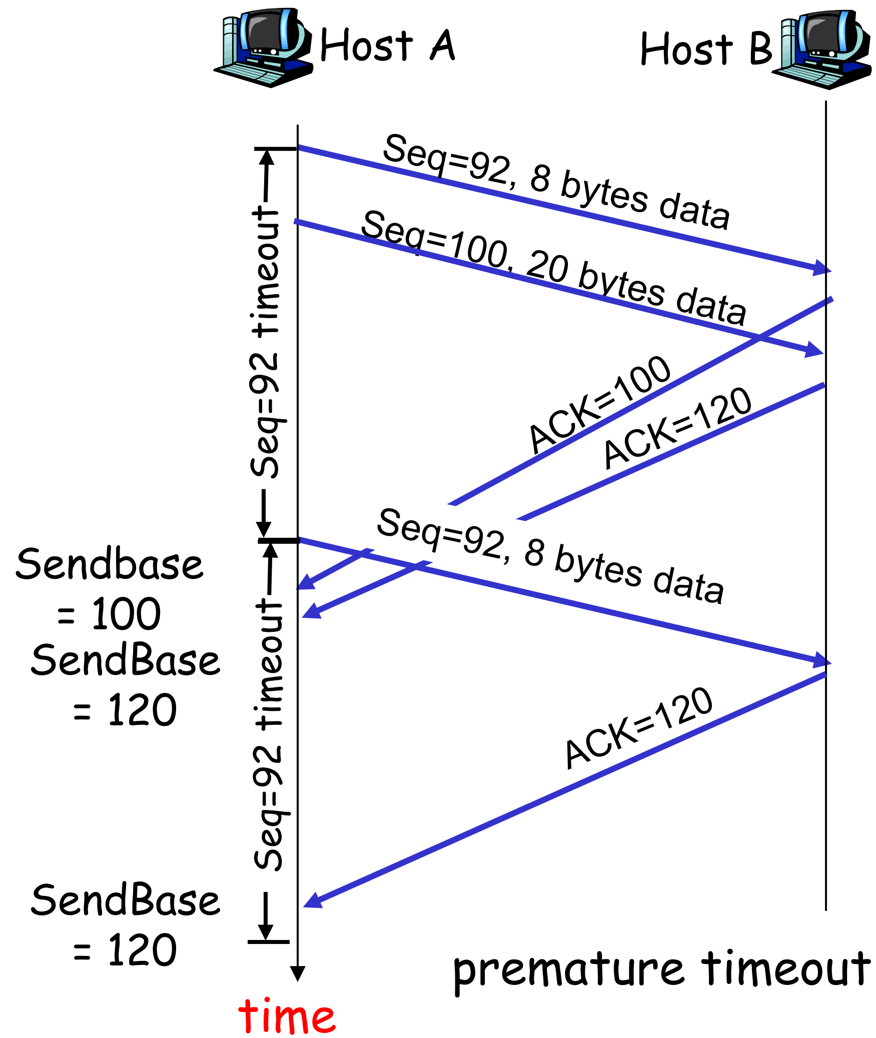
Comment:
- SendBase-1: last cumulatively ACKed byte

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

TCP: retransmission scenarios

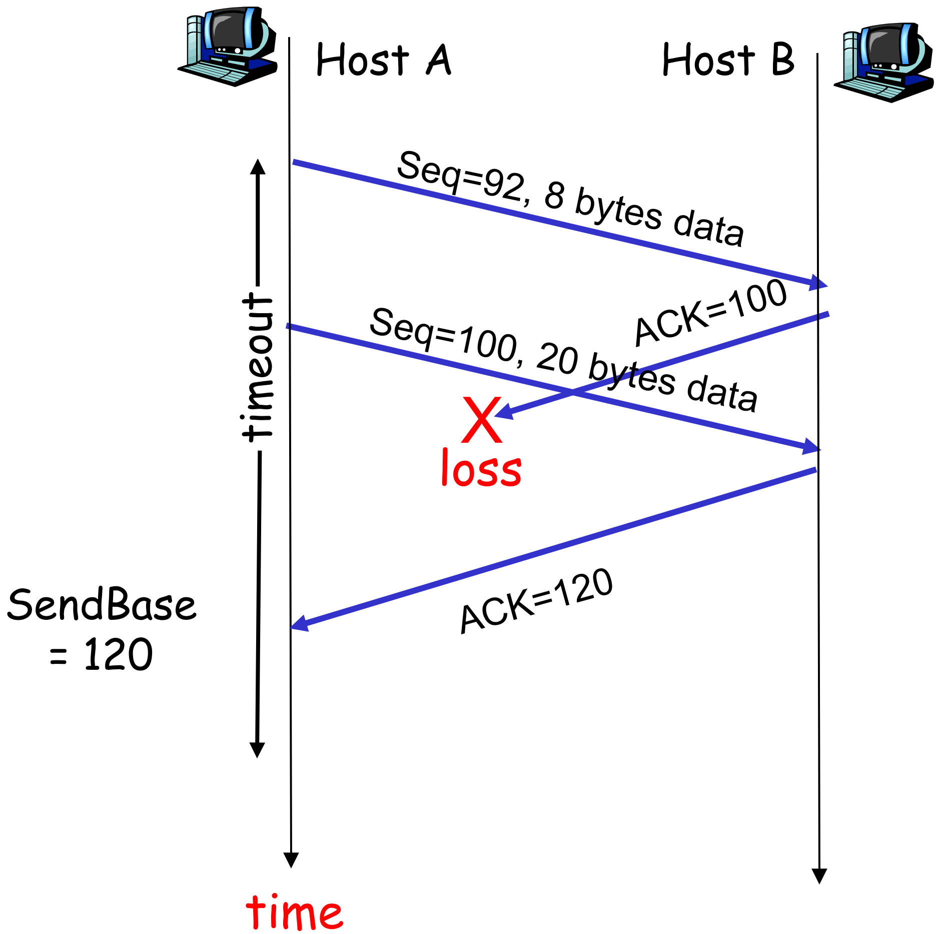


lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



Cumulative ACK scenario

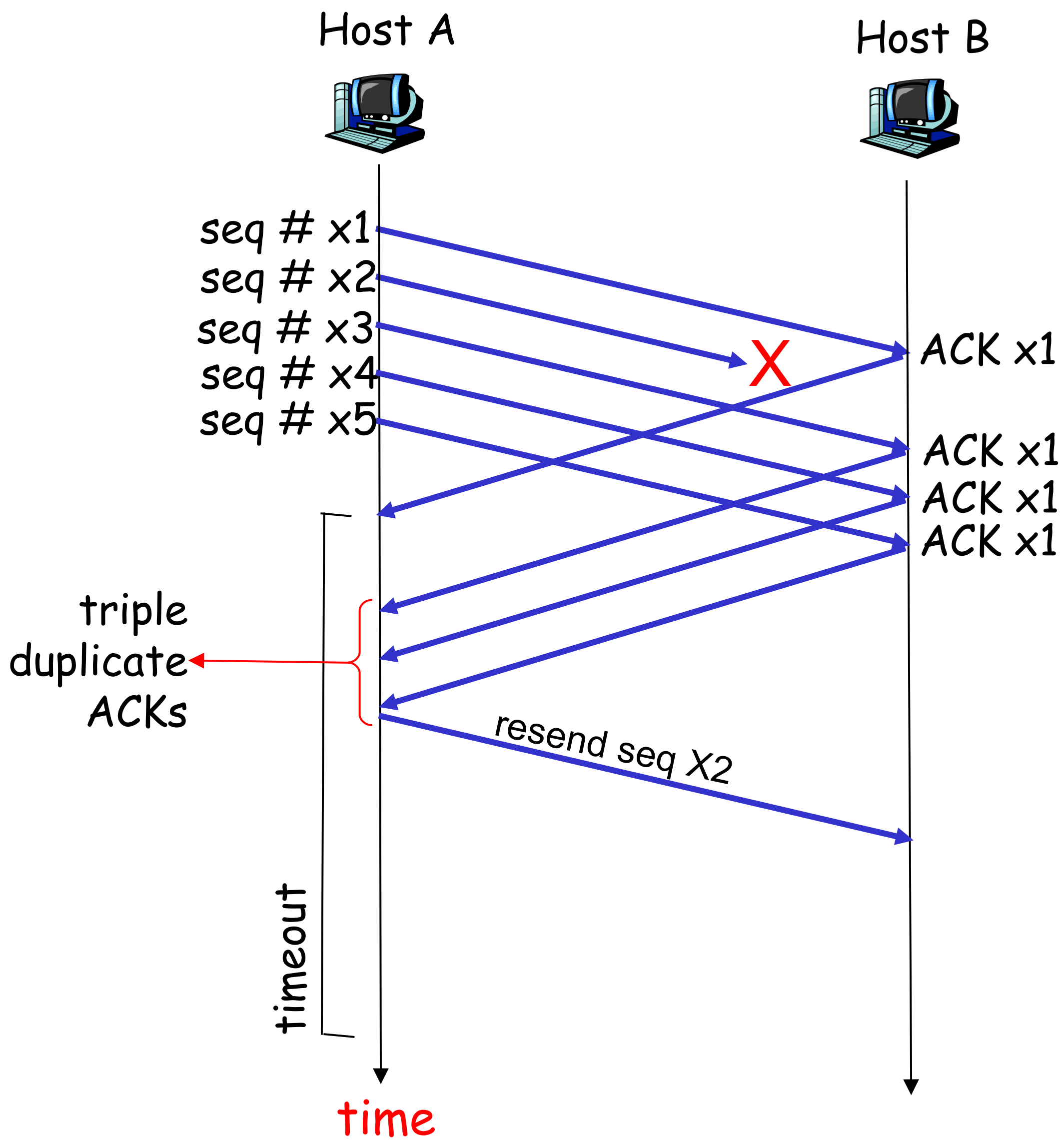
TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment 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 for that segment

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


Host A
Host B
seq # x1
seq # x2
seq # x3
seq # x4
seq # x5
X
ACK x1
ACK x1
ACK x1
ACK x1
triple
duplicate
ACKs
resend seq X2
timeout
time

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

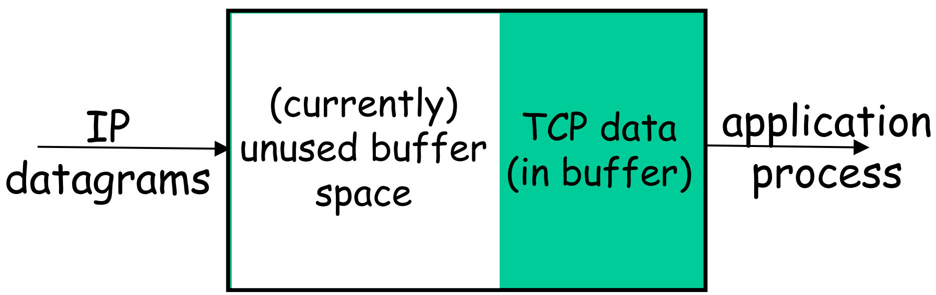
fast retransmit

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

- receive side of TCP connection has a receive buffer:



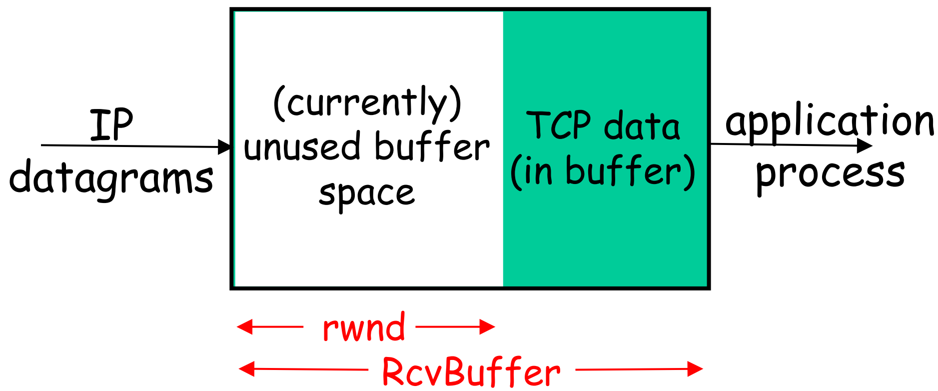
- app process may be slow at reading from buffer

flow control

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

- *speed-matching service:* matching send rate to receiving application's drain rate

TCP Flow control: how it works



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

- unused buffer space:

= **rwnd**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

- receiver: advertises unused buffer space by including `rwnd` value in segment header
- sender: limits # of unACKed bytes to `rwnd`
  - guarantees receiver's buffer doesn't overflow

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 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

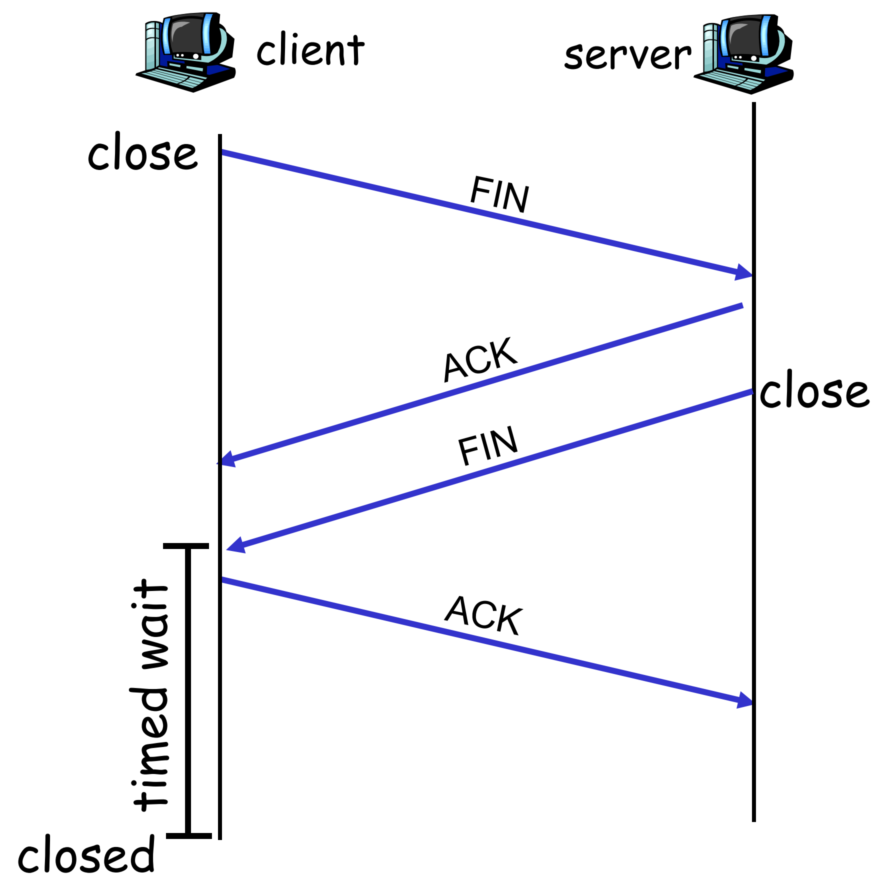
TCP Connection Management (cont.)

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.



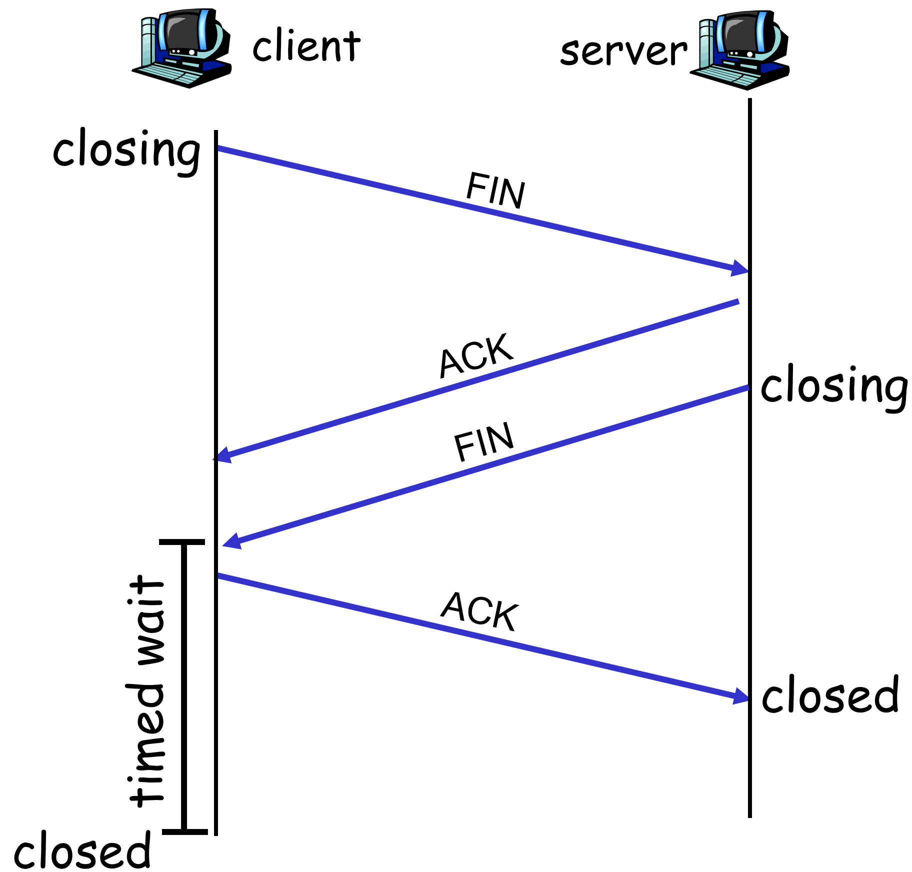
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

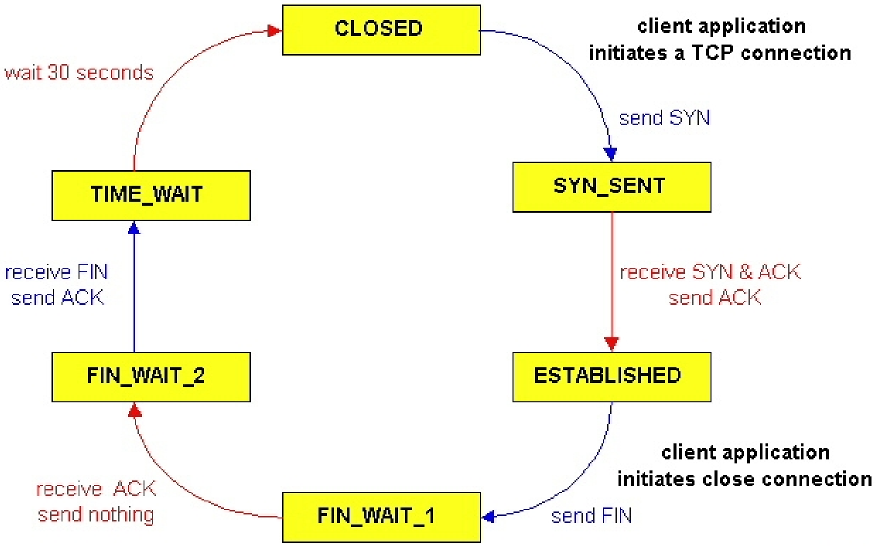
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

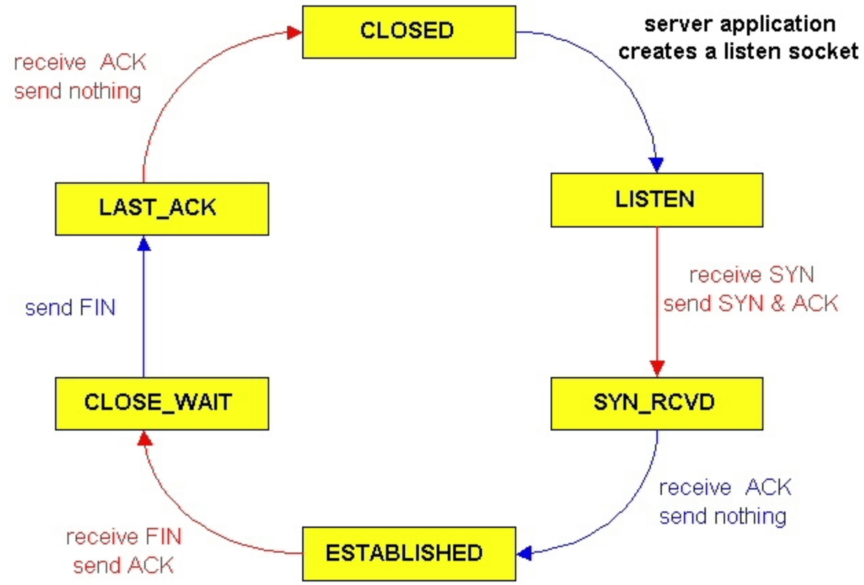


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



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

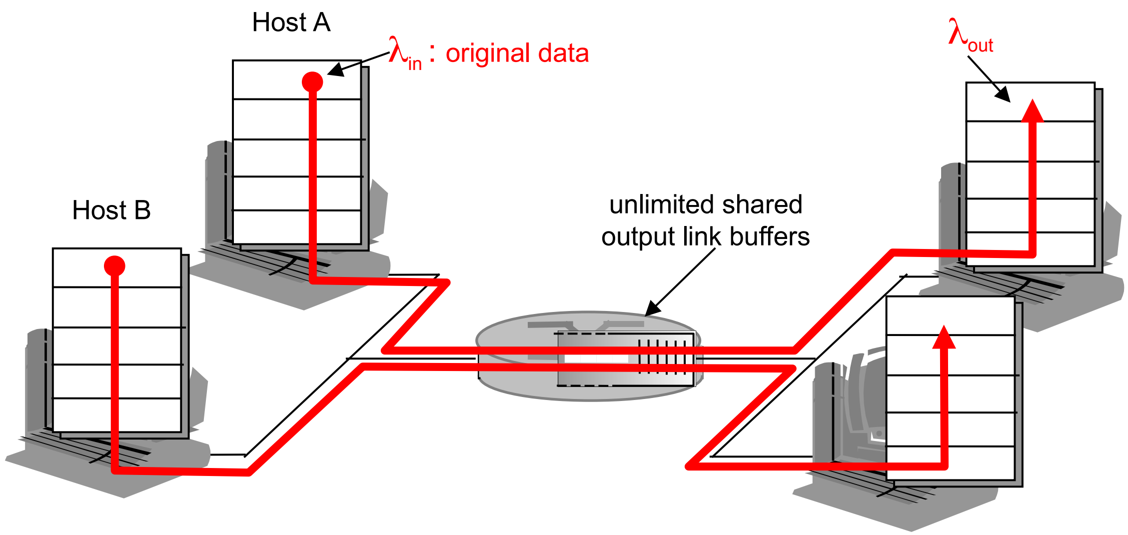
Principles of Congestion Control

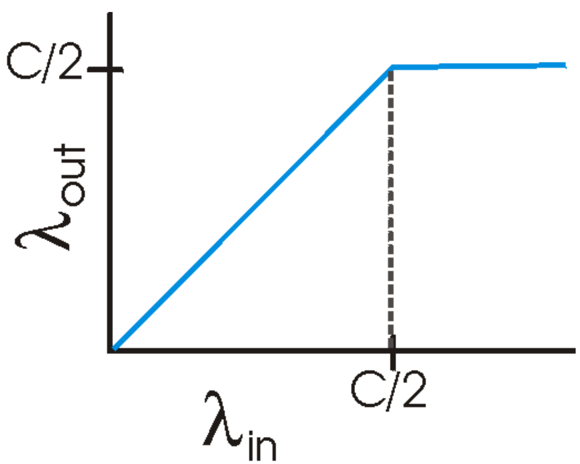
Congestion:

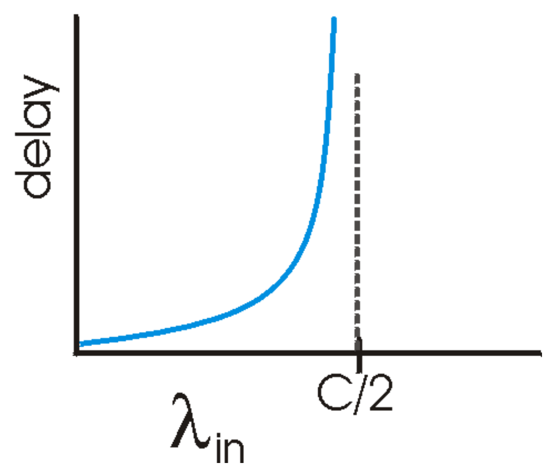
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

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



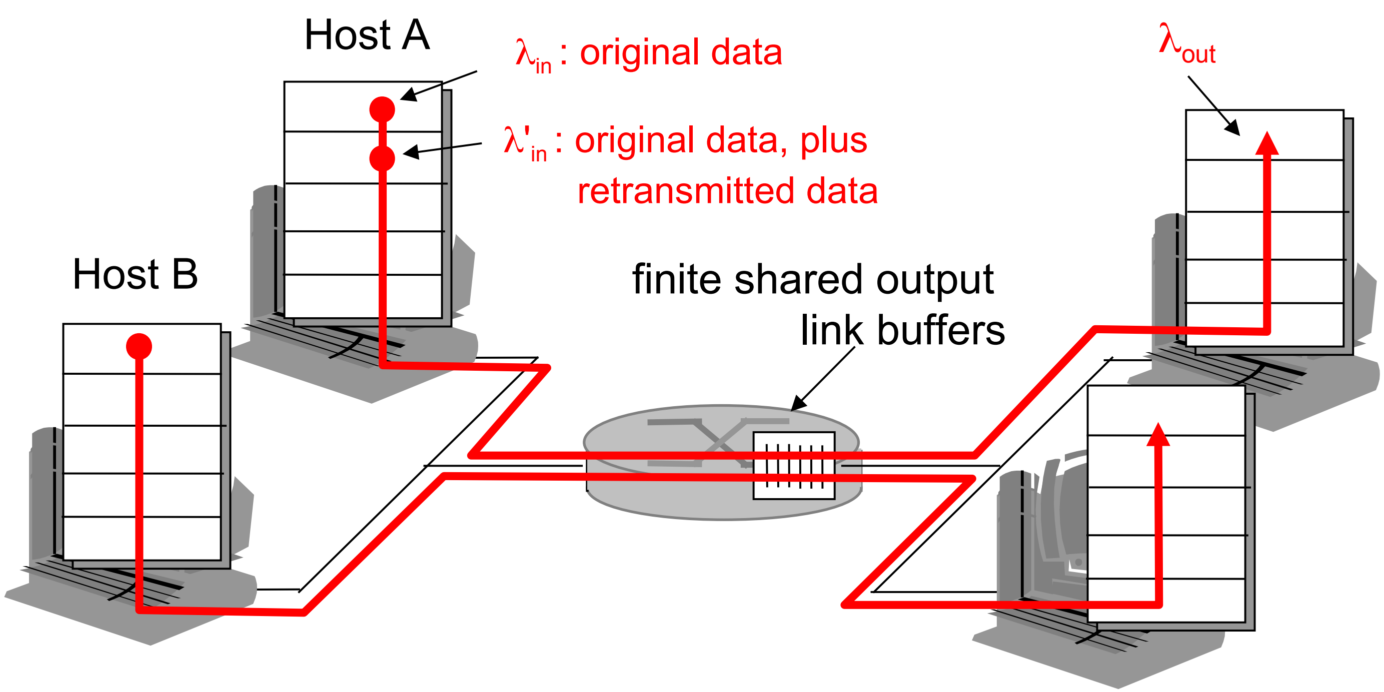




- large delays when congested
- maximum achievable throughput

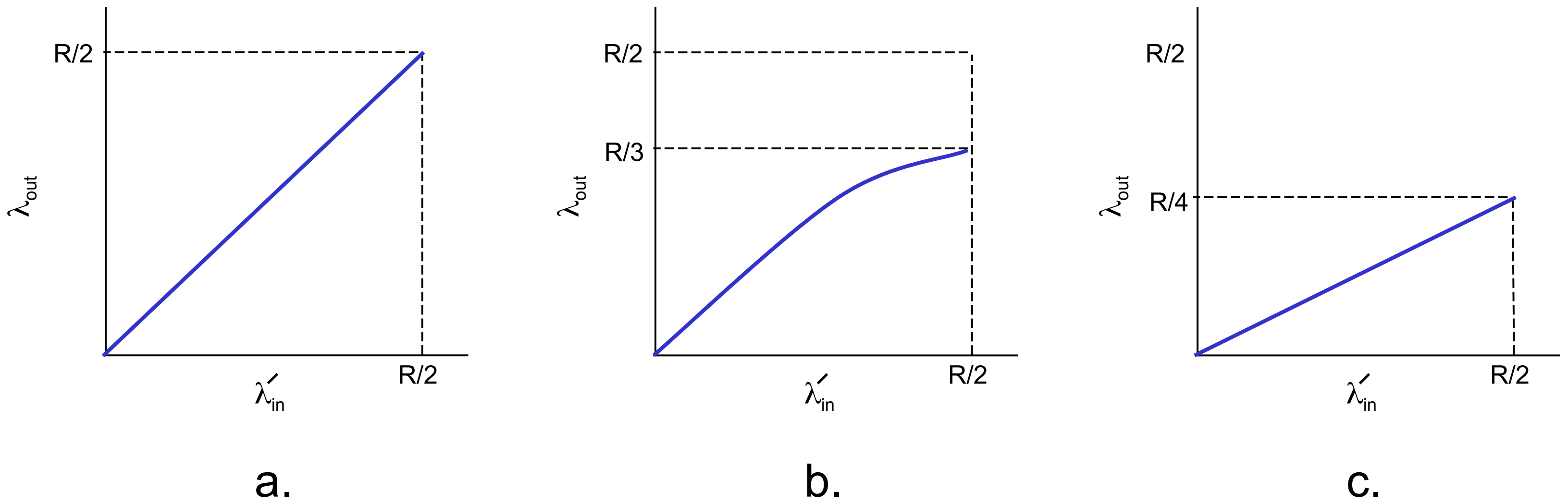
Causes/costs of congestion: scenario 2

- 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}



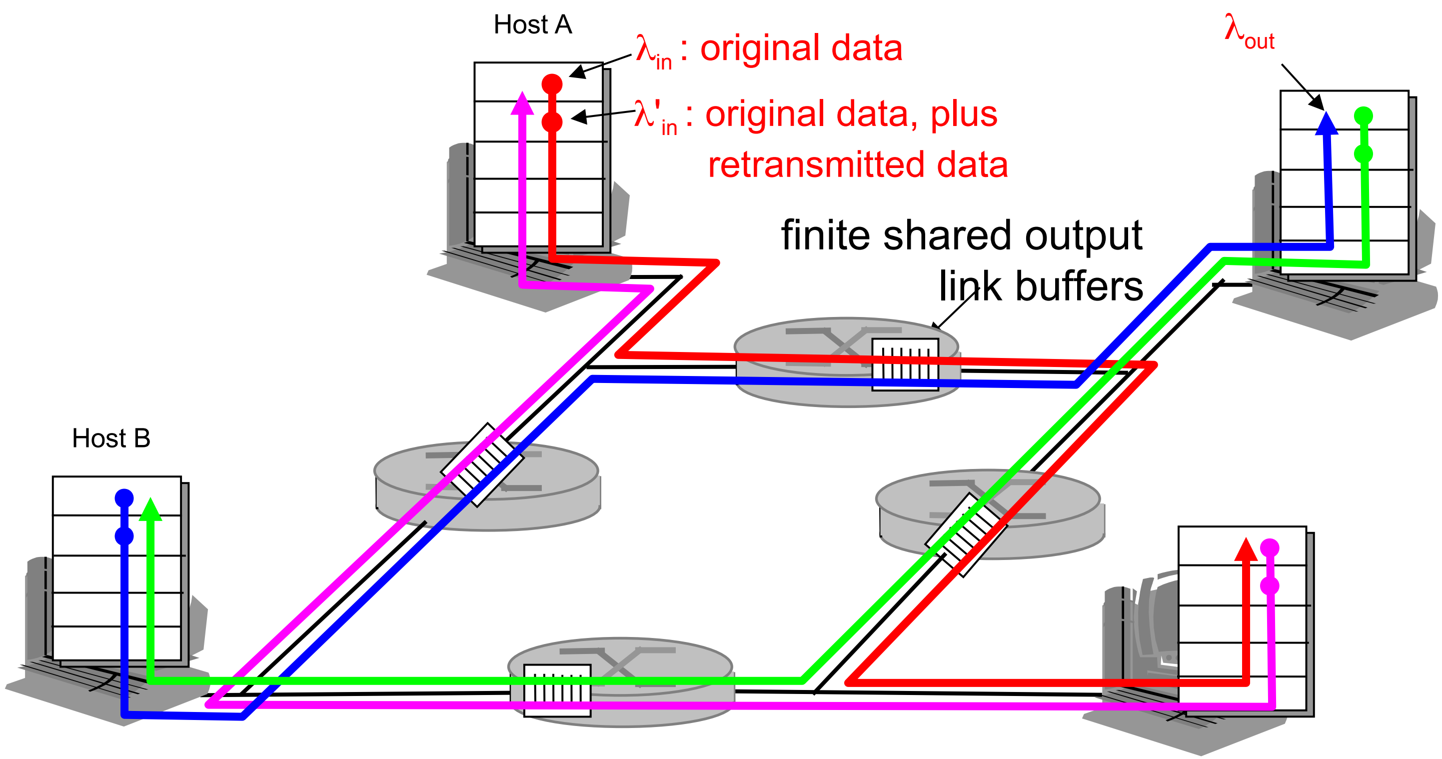
"costs" of congestion:

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

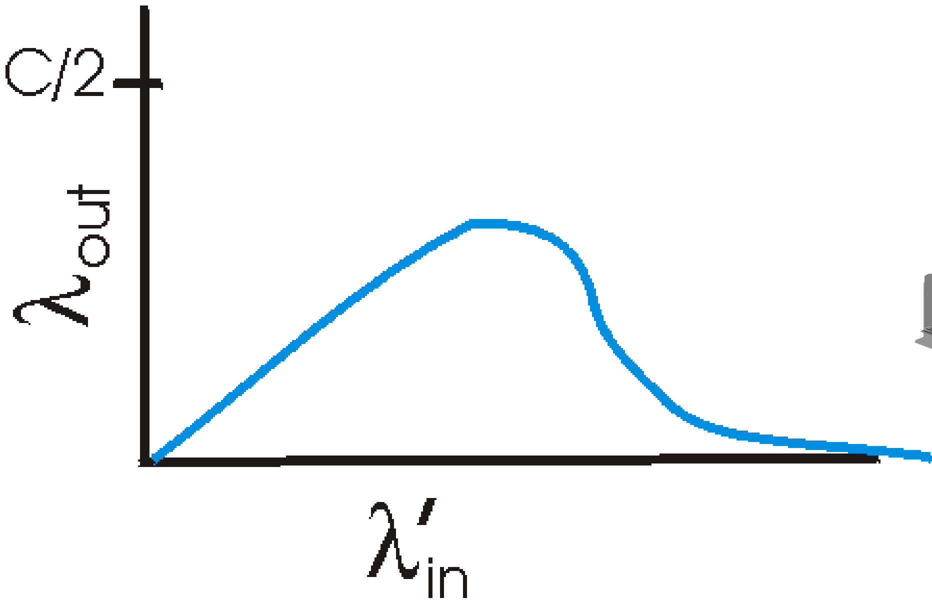
Causes/costs of congestion: scenario 3

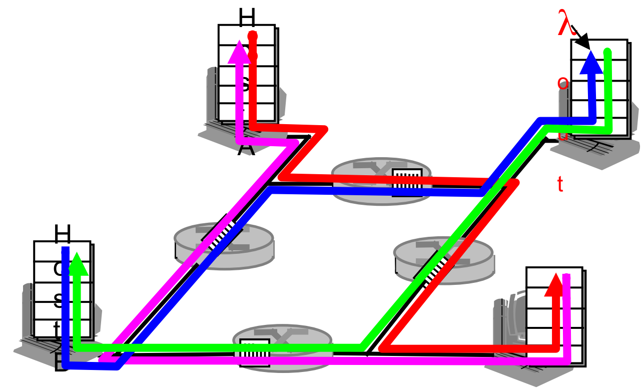
- four senders
- multihop paths
- timeout/retransmit

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

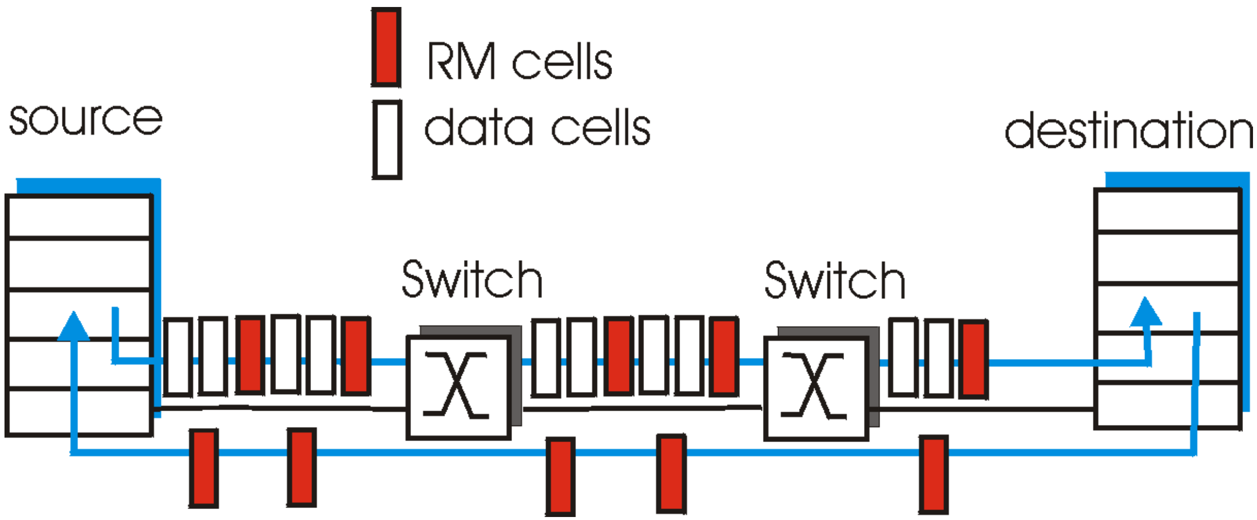
ABR: available bit rate:

- "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

Chapter 3 outline

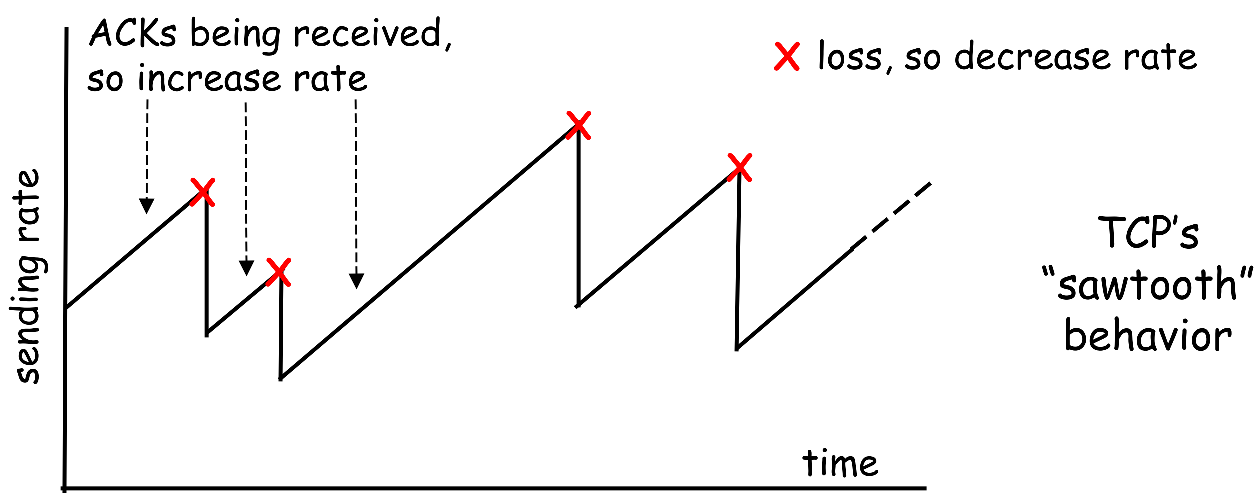
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP congestion control:

- *goal:* TCP sender should transmit as fast as possible, but without congesting network
  - Q: how to find rate *just* below congestion level
- decentralized: each TCP sender sets its own rate, based on *implicit* feedback:
  - *ACK:* segment received (a good thing!), network not congested, so increase sending rate
  - *lost segment:* assume loss due to congested network, so decrease sending rate

TCP congestion control: bandwidth probing

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



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

TCP Congestion Control: details

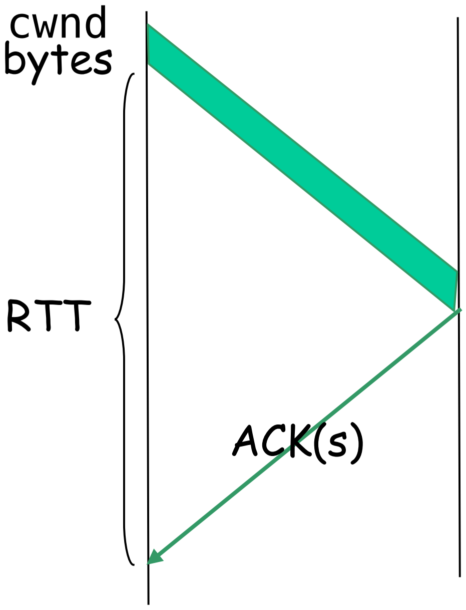
- sender limits rate by limiting number of unACKed bytes "in pipeline":

  LastByteSent-LastByteAcked ≤ cwnd

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

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

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



TCP Congestion Control: more details

segment loss event: reducing `cwnd`

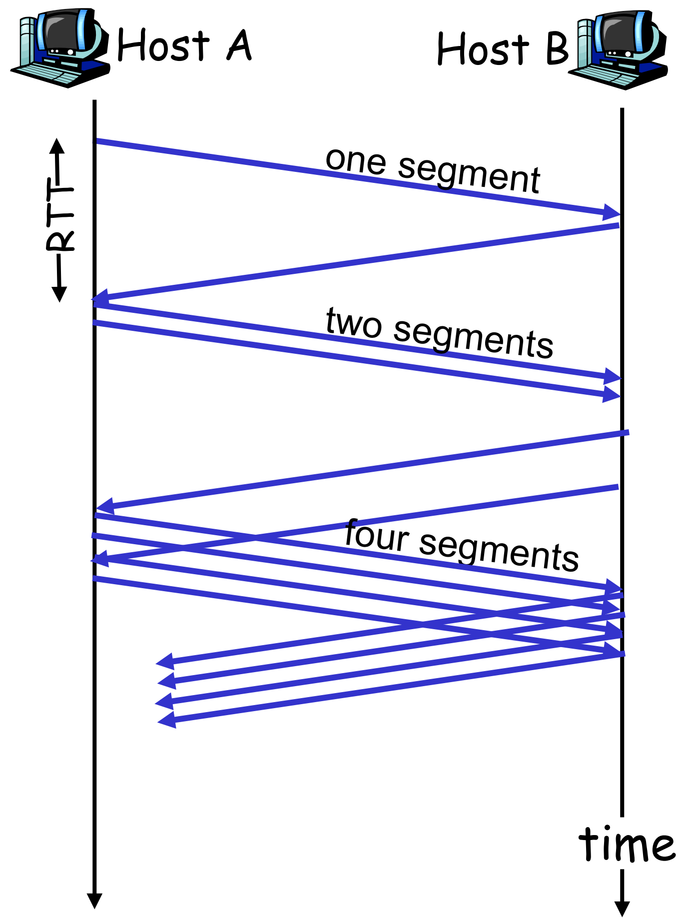
- timeout: no response from receiver
  - cut `cwnd` to 1
- 3 duplicate ACKs: at least some segments getting through (recall fast retransmit)
  - cut **cwnd** in half, less aggressively than on timeout

ACK received: increase `cwnd`

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

TCP Slow Start

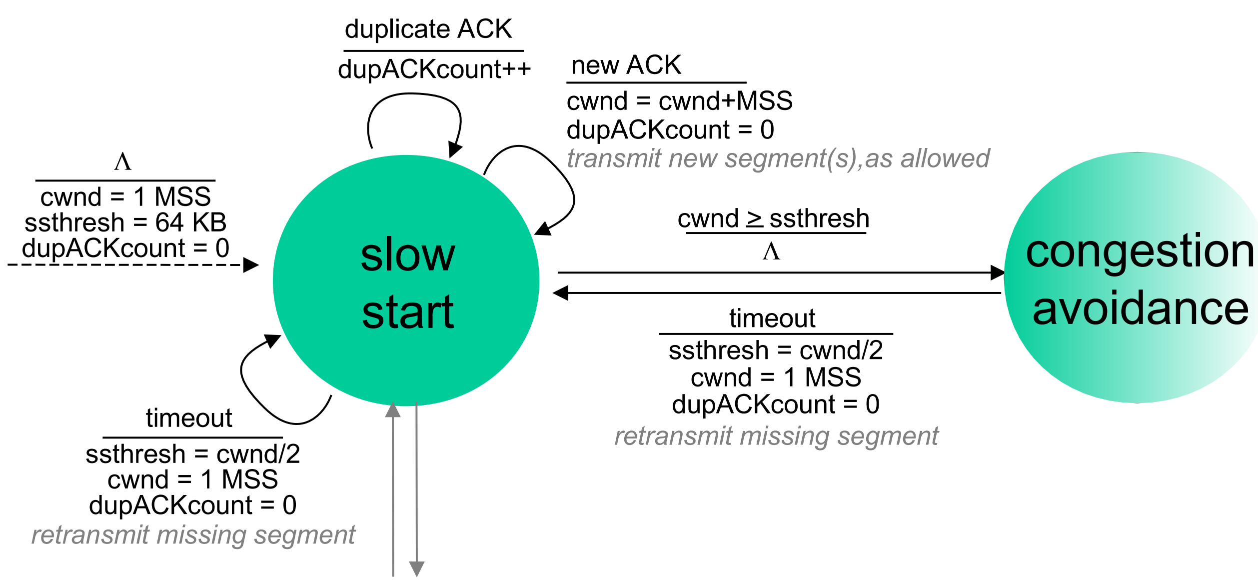
- when connection begins, **cwnd** = 1 MSS
  - example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- increase rate exponentially until first loss event or when threshold reached
  - double **cwnd**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



Transitioning into/out of slowstart

ssthresh: cwnd threshold maintained by TCP

- on loss event: set **ssthresh** to **cwnd/2**
  - remember (half of) TCP rate when congestion last occurred
- when **cwnd >= ssthresh**: transition from slowstart to congestion avoidance phase



TCP: congestion avoidance

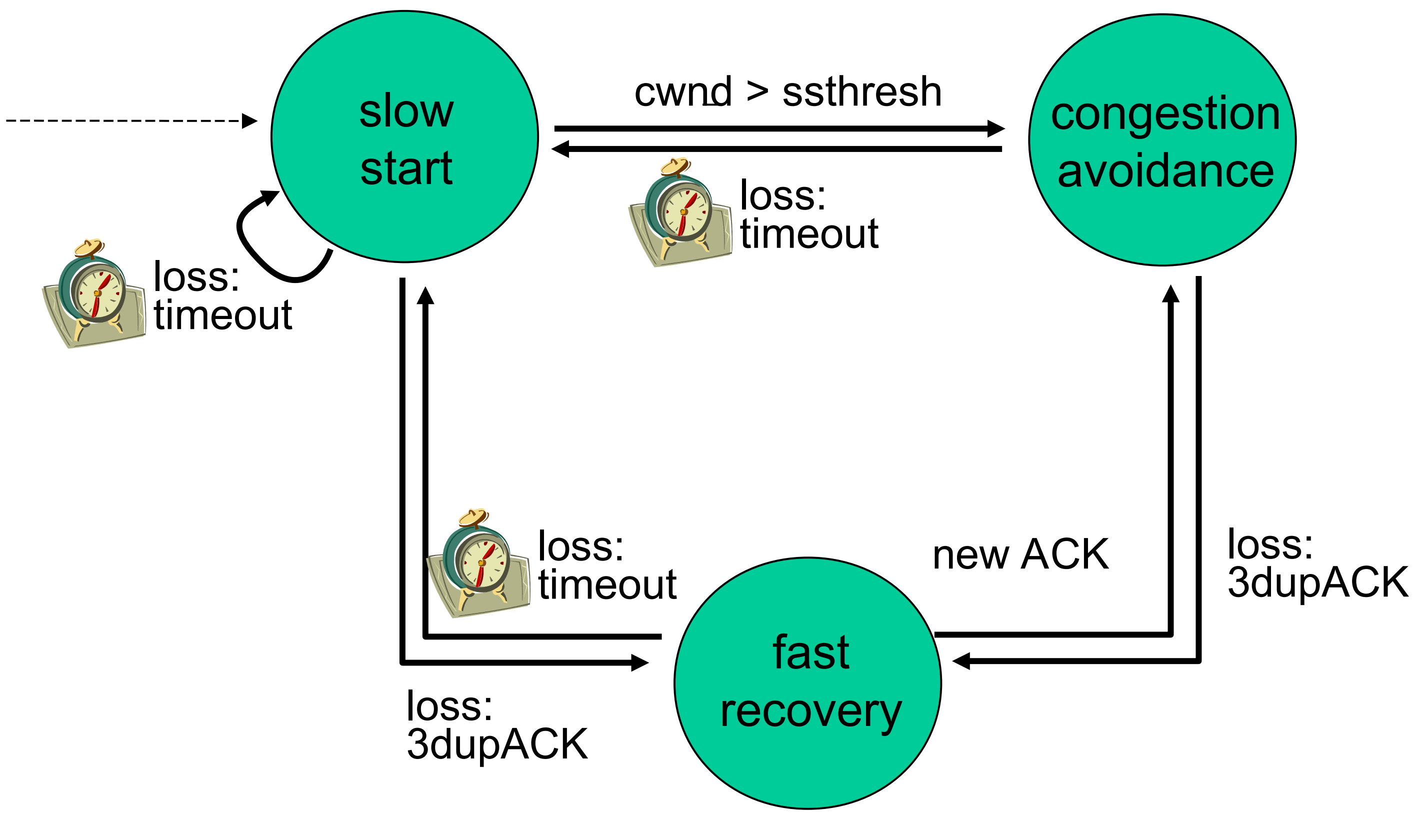
- when `cwnd > ssthresh` grow `cwnd` linearly
  - increase `cwnd` by 1 MSS per RTT
  - approach possible congestion slower than in slowstart
  - implementation: `cwnd = cwnd + MSS/cwnd` for each ACK received

AIMD

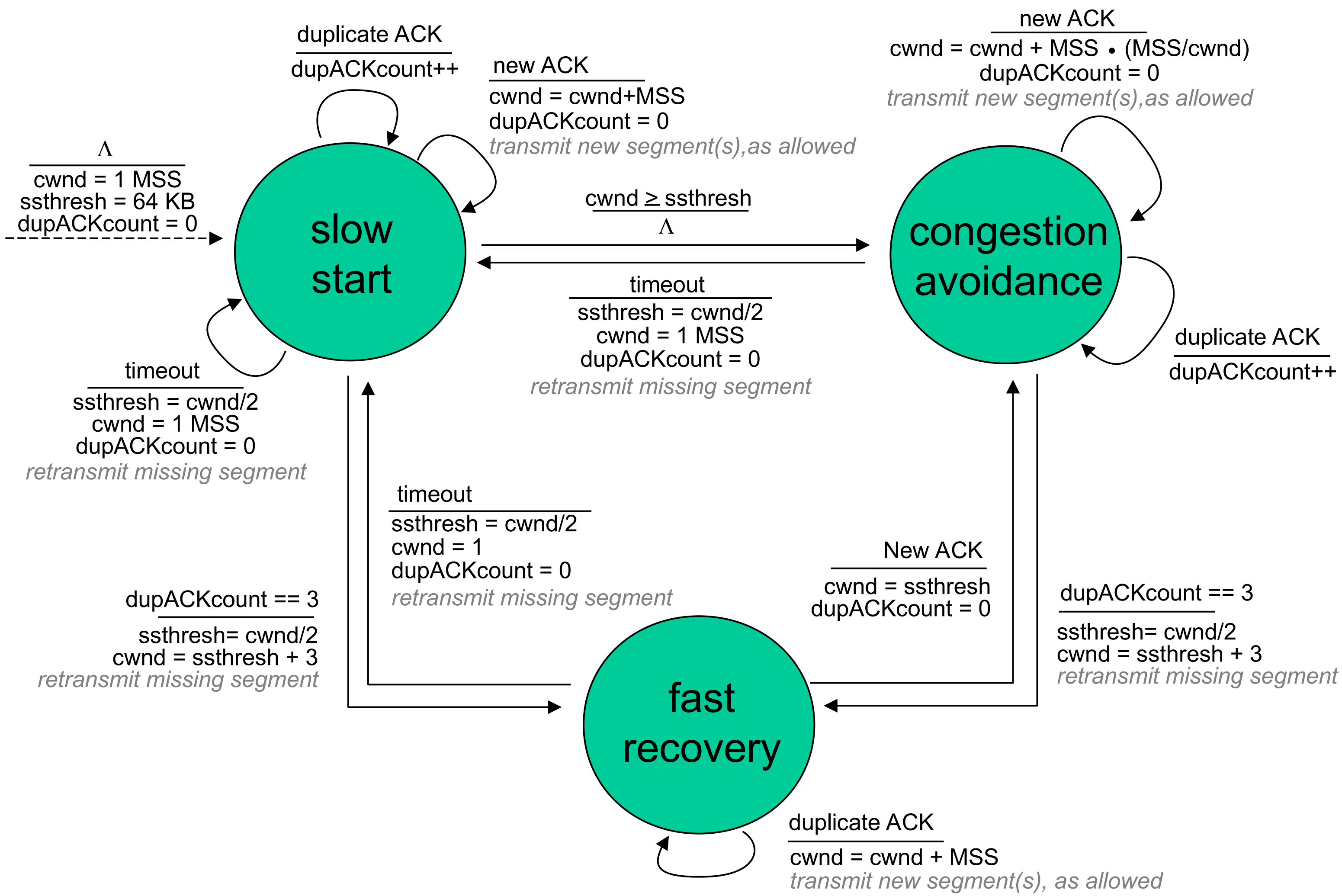
- ACKs: increase `cwnd` by 1 MSS per RTT: additive increase
- loss: cut `cwnd` in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

TCP congestion control FSM: overview

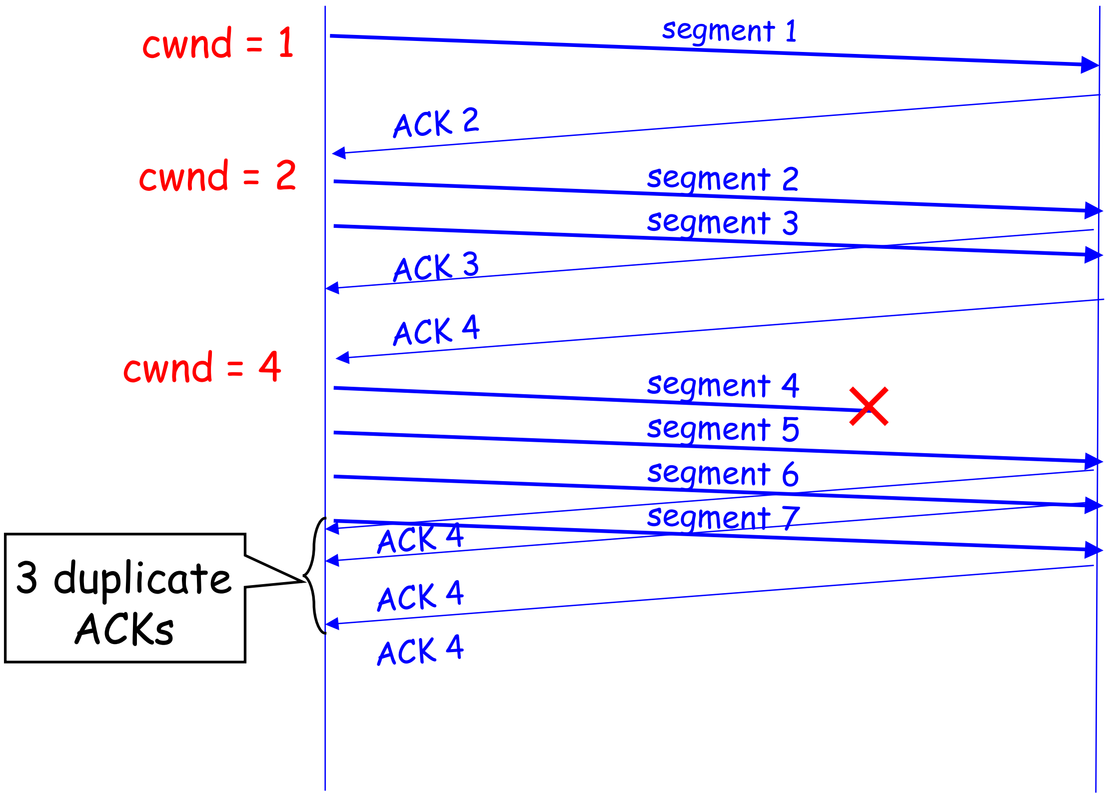


TCP congestion control FSM: details



Fast Retransmits

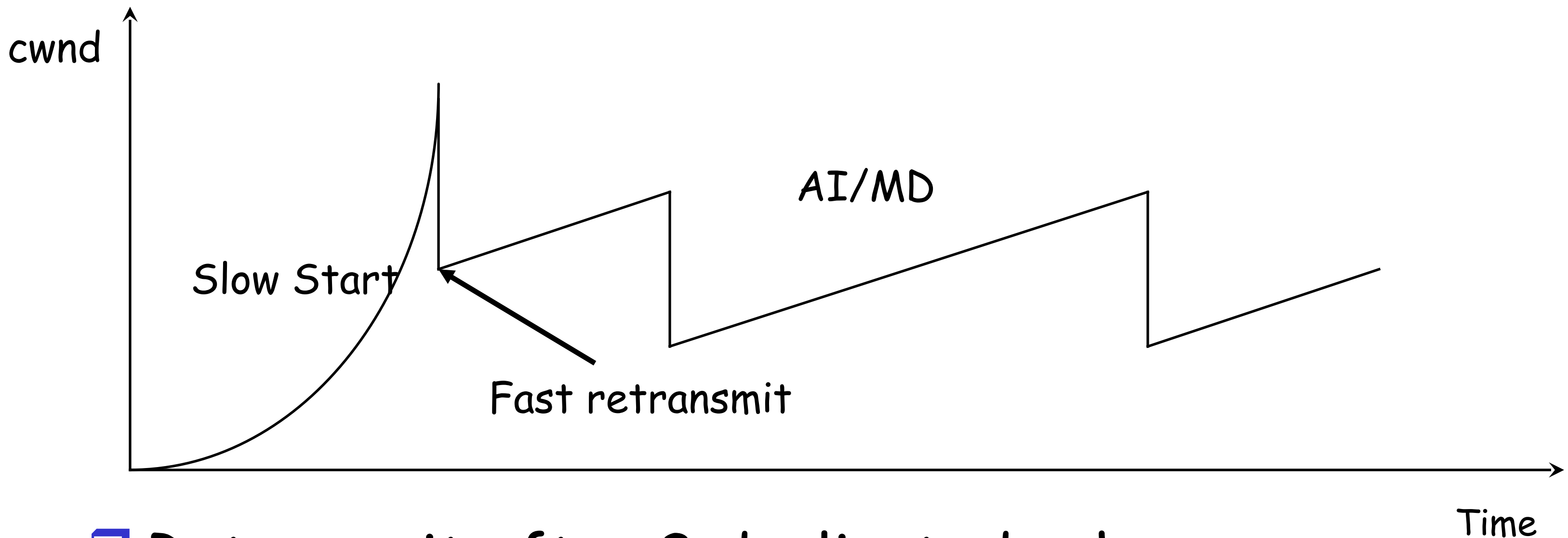
- Resend a segment after 3 duplicate ACKs
  - Duplicate ACK means that an out-of sequence segment was received
- Notes:
  - ACKs are for next expected packet
  - Packet reordering can cause duplicate ACKs
  - Window may be too small to get enough duplicate ACKs



Fast Recovery: After a Fast Retransmit

- *ssthresh* = *cwnd* / 2
- *cwnd* = *ssthresh*
  - Instead of setting *cwnd* to 1, cut cwnd in half (multiplicative decrease)
- For each dup ack arrival
  - *dupack*++
  - Indicates packet left network, so we may be able to send more
  - MaxWindow = min(*cwnd* + *dupack*, AdvWin)
- Receive ack for new data (beyond initial dup ack)
  - *dupack* = 0
  - Exit fast recovery
- But when RTO expires still do *cwnd* = 1

Fast Retransmit and Fast Recovery



- Retransmit after 3 duplicated acks
  - Prevent expensive timeouts
- Reduce slow starts
- At steady state, *cwnd* oscillates around the optimal window size

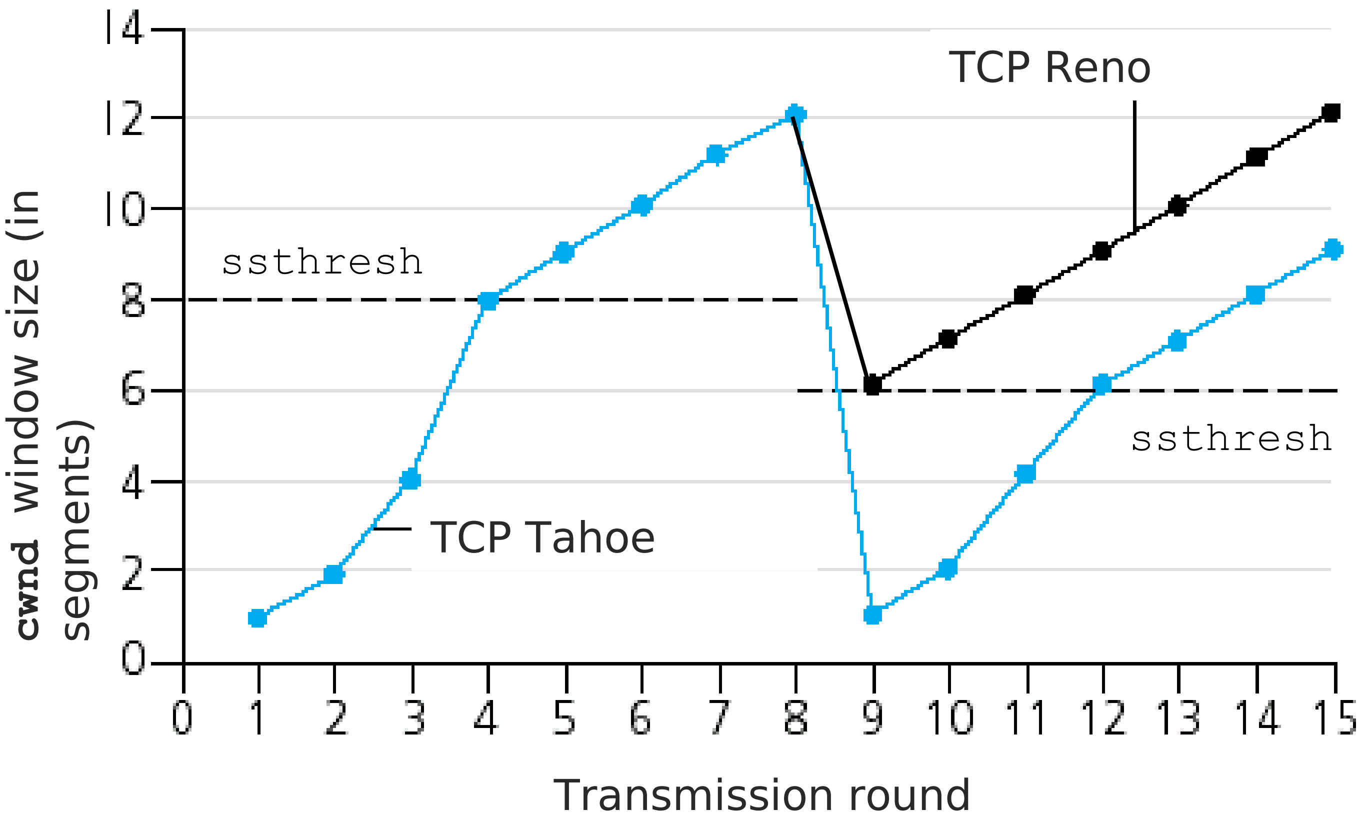
TCP Congestion Control Summary

- Measure available bandwidth
  - Slow start: fast, hard on network
  - AIMD: slow, gentle on network
- Detecting congestion
  - Timeout based on RTT
    - Robust, causes low throughput
  - Fast Retransmit: avoids timeouts when few packets lost
    - Can be fooled, maintains high throughput
- Recovering from loss
  - Fast recovery: don't set cwnd=1 with fast retransmits

TCP Flavors

- TCP-Tahoe
  - cwnd =1 whenever drop is detected
- TCP-Reno
  - cwnd =1 on timeout
  - cwnd = cwnd/2 on dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK

Popular "flavors" of TCP

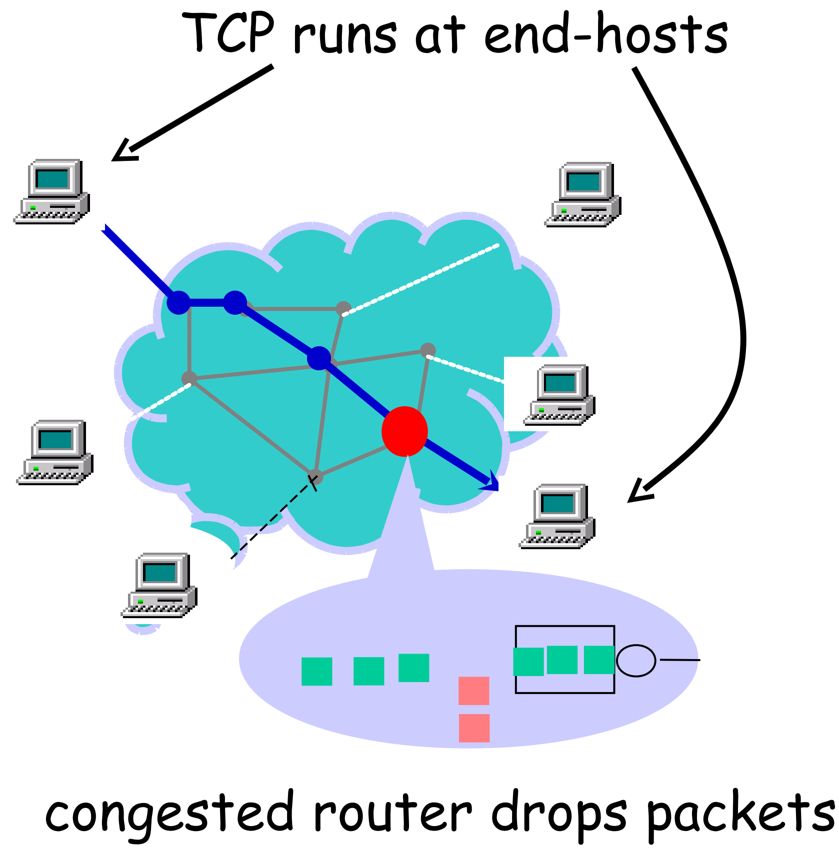


TCP-SACK

- SACK = Selective Acknowledgements
- ACK packets identify exactly which packets have arrived
- Makes recovery from multiple losses much easier

TCP behavior

- congestion control:
  - decrease sending rate when loss detected, increase when no loss
- routers
  - discard, mark packets when congestion occurs
- interaction between end systems (TCP) and routers?
  - want to understand (quantify) this interaction



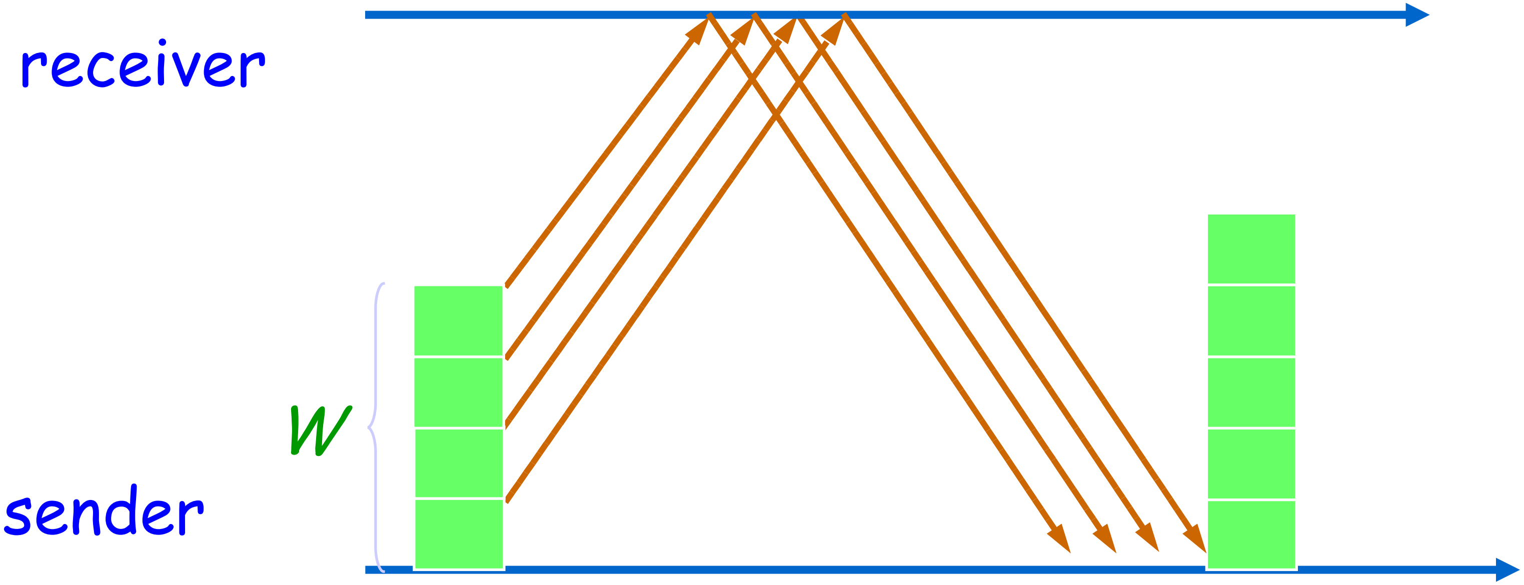
Generic TCP behavior

- window algorithm (window W)
  - up to W packets in network
  - return of ACK allows sender to send another packet
  - cumulative ACKS
- increase window by one per RTT

$$W \leftarrow W + 1/W \text{ per ACK}$$

$$\Rightarrow W \leftarrow W + 1 \text{ per RTT}$$

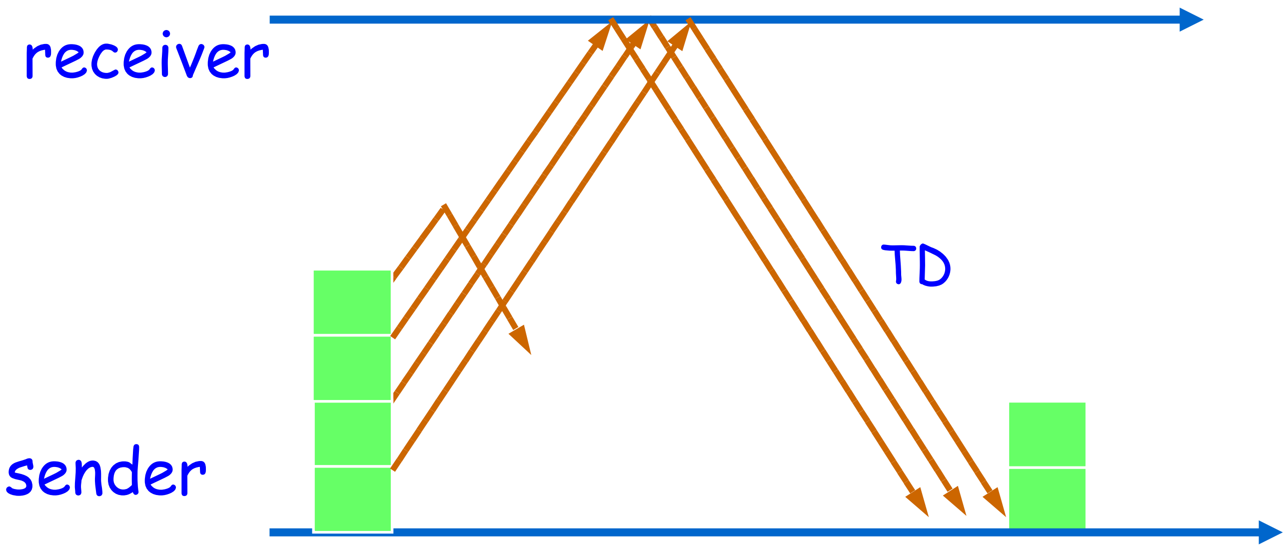
- seeks available network bandwidth


receiver
W
sender

Generic TCP behavior

- window algorithm (window W)
- increase window by one per RTT

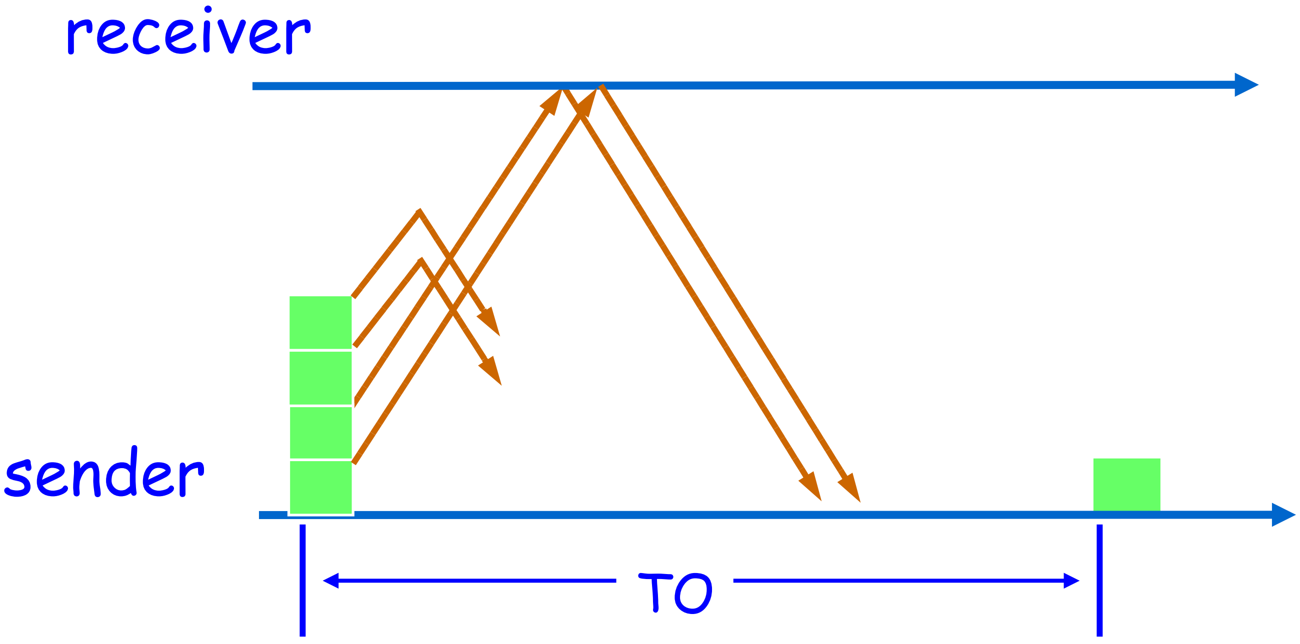
  W <- $W + 1/W$ *per ACK*
- loss indication of congestion
- decrease window by half on detection of loss, (triple duplicate ACKs), W <- $W/2$


receiver
TD
sender

Generic TCP Behavior

- window algorithm (window W)
- increase window by one per RTT

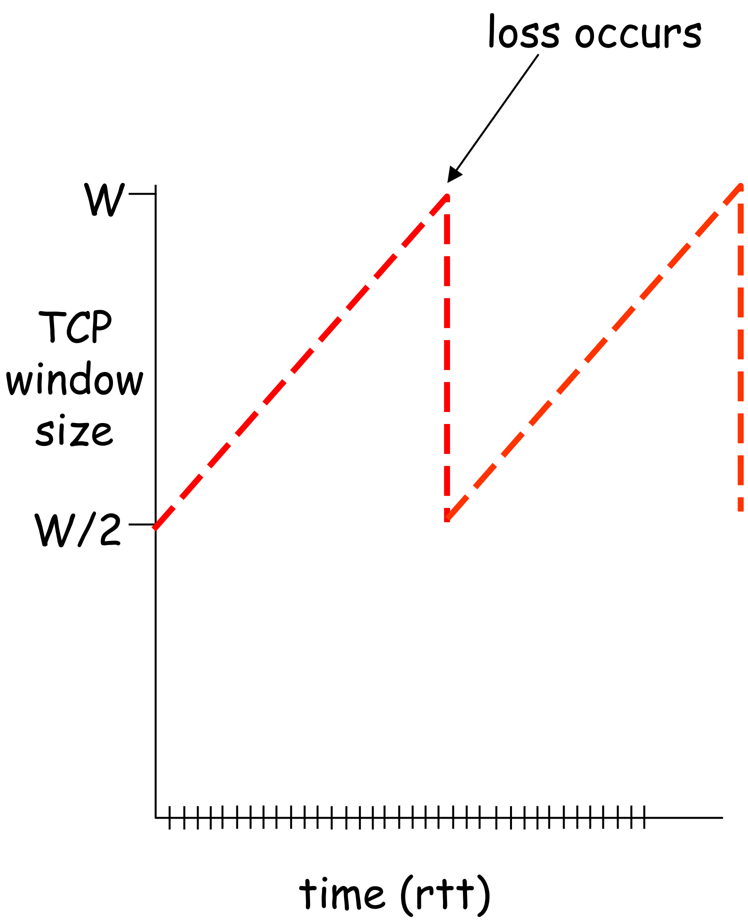
  W <- W +1/ W *per* ACK
- halve window on detection of loss, W <- W/2
- timeouts due to lack of ACKs -> window reduced to one, W <- 1


receiver
sender
TO

Generic TCP Behavior

- window algorithm (window W)
- increase window by one per RTT (or one over window per ACK, W <- W+1/W)
- halve window on detection of loss, W <- W/2
- timeouts due to lack of ACKs, W <- 1
- successive timeout intervals grow exponentially long up to six times

TCP throughput/loss relationship

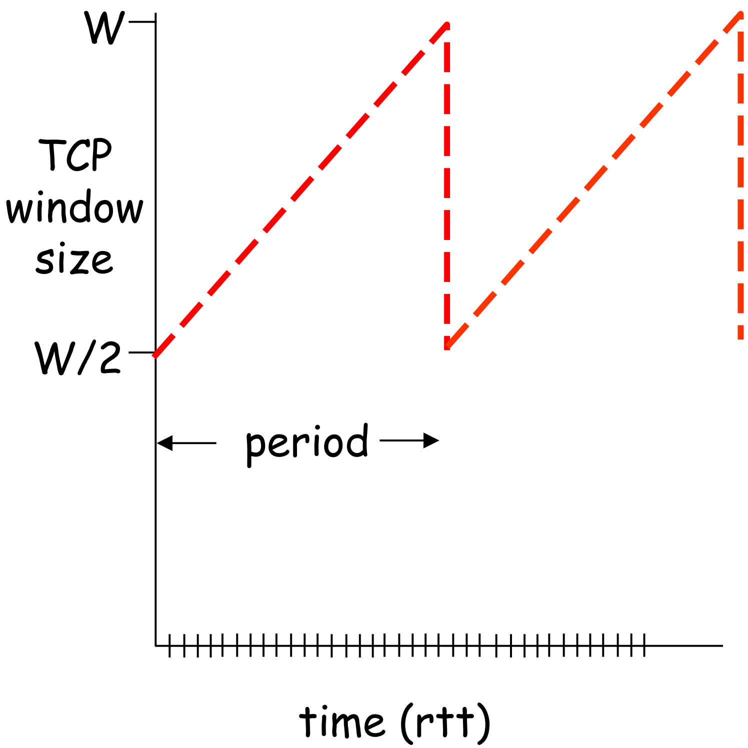


Idealized model:

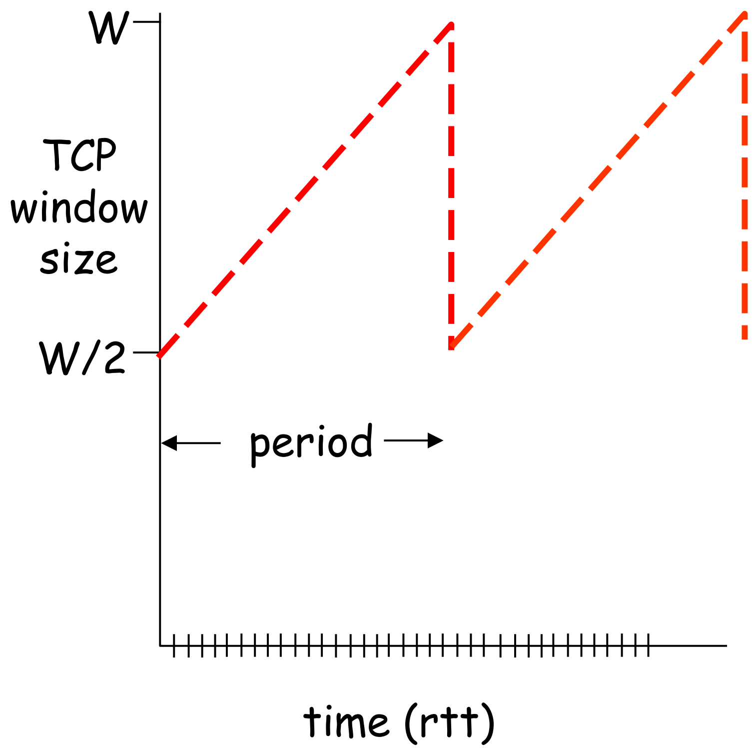
- W is maximum supportable window size (then loss occurs)
- TCP window starts at W/2 grows to W, then halves, then grows to W, then halves…
- one window worth of packets each RTT
- *to find:* throughput as function of loss, RTT

TCP throughput/loss relationship

packets sent per “period” =



TCP throughput/loss relationship



\# packets sent per "period" =

$$\frac{W}{2}+\left(\frac{W}{2}+1\right)+\ldots+W=\sum_{n=0}^{W/2}\left(\frac{W}{2}+n\right)$$

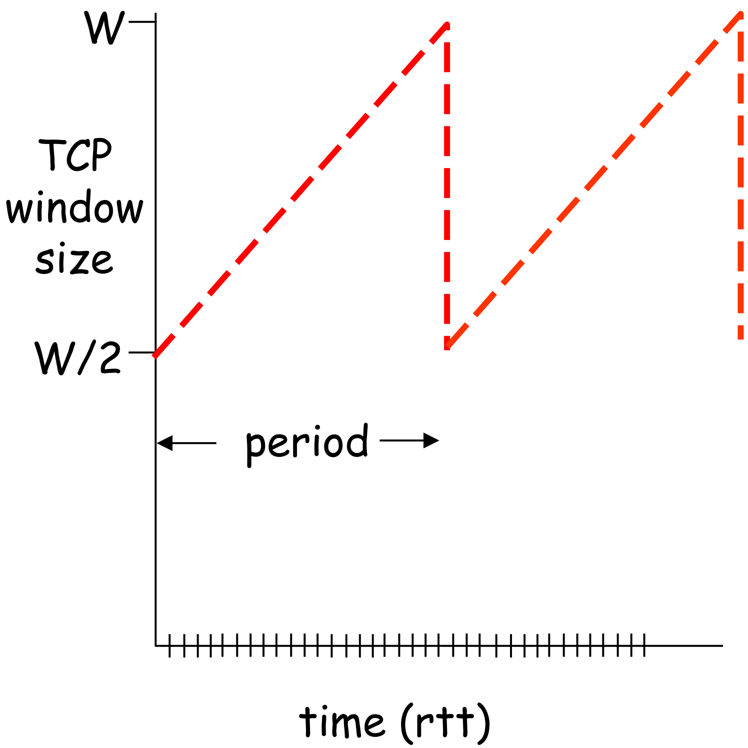
$$=\left(\frac{W}{2}+1\right)\frac{W}{2}+\sum_{n=0}^{W/2}n$$

$$=\left(\frac{W}{2}+1\right)\frac{W}{2}+\frac{W/2(W/2+1)}{2}$$

$$=\frac{3}{8}W^2+\frac{3}{4}W$$

$$\approx\frac{3}{8}W^2$$

TCP throughput/loss relationship



packets sent per "period" $\approx \frac{3}{8}W^2$

1 packet lost per "period" implies:

$p_{loss} \approx \frac{8}{3W^2}$ or: $W = \sqrt{\frac{8}{3p_{loss}}}$

$$B = \text{avg.\_thruput} = \frac{3}{4}W \frac{\text{packets}}{\text{rtt}}$$

$$\boxed{B = \text{avg.\_thruput} = \frac{1.22}{\sqrt{p_{loss}}} \frac{\text{packets}}{\text{rtt}}}$$

B throughput formula can be extended to model timeouts and slow start (PFTK)

TCP Futures: TCP over "long, fat pipes"

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

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

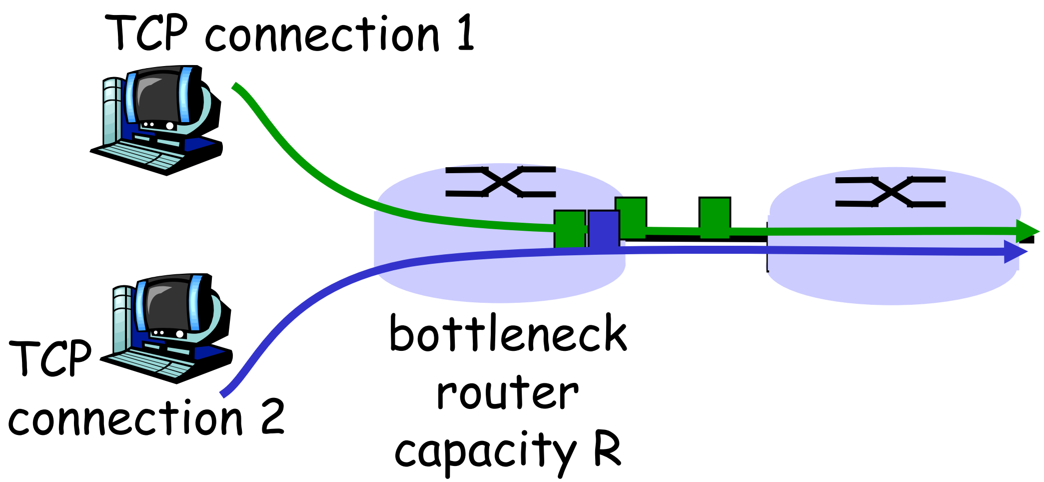
- → $L = 2 \cdot 10^{-10}$ *Wow*
- new versions of TCP for high-speed

Summary: TCP Congestion Control

- when **cwnd < ssthresh**, sender in slow-start phase, window grows exponentially.
- when **cwnd >= ssthresh**, sender is in congestion-avoidance phase, window grows linearly.
- when triple duplicate ACK occurs, **ssthresh** set to **cwnd/2**, **cwnd** set to ~ **ssthresh**
- when timeout occurs, **ssthresh** set to **cwnd/2**, **cwnd** set to 1 MSS.

TCP Fairness

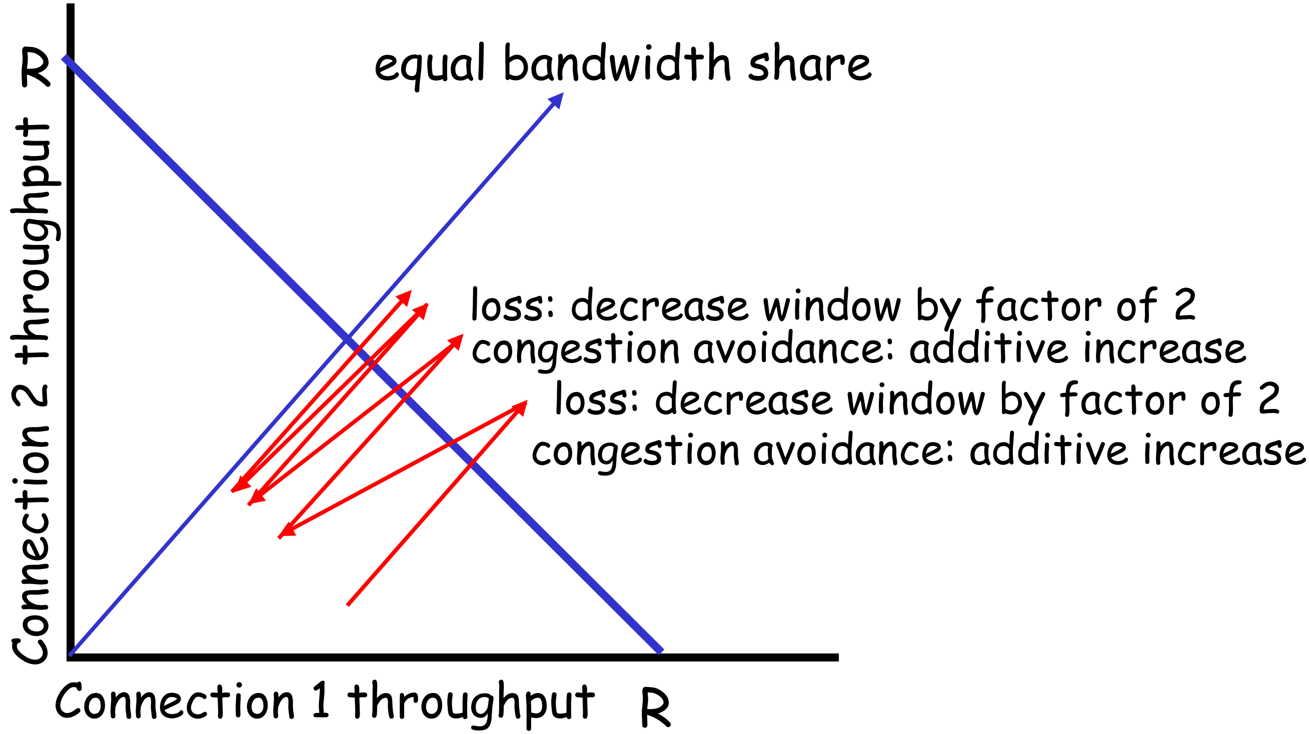
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



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:
  - pump audio/video at constant rate, tolerate packet loss

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 !

Recap : TCP/ Reno

- **Two cases**
  - 3 duplicate ACKs (network capable of delivering some packets)
  - Timeout (more alarming)

- **Two phases**

  1. Slow start (SS) - MI

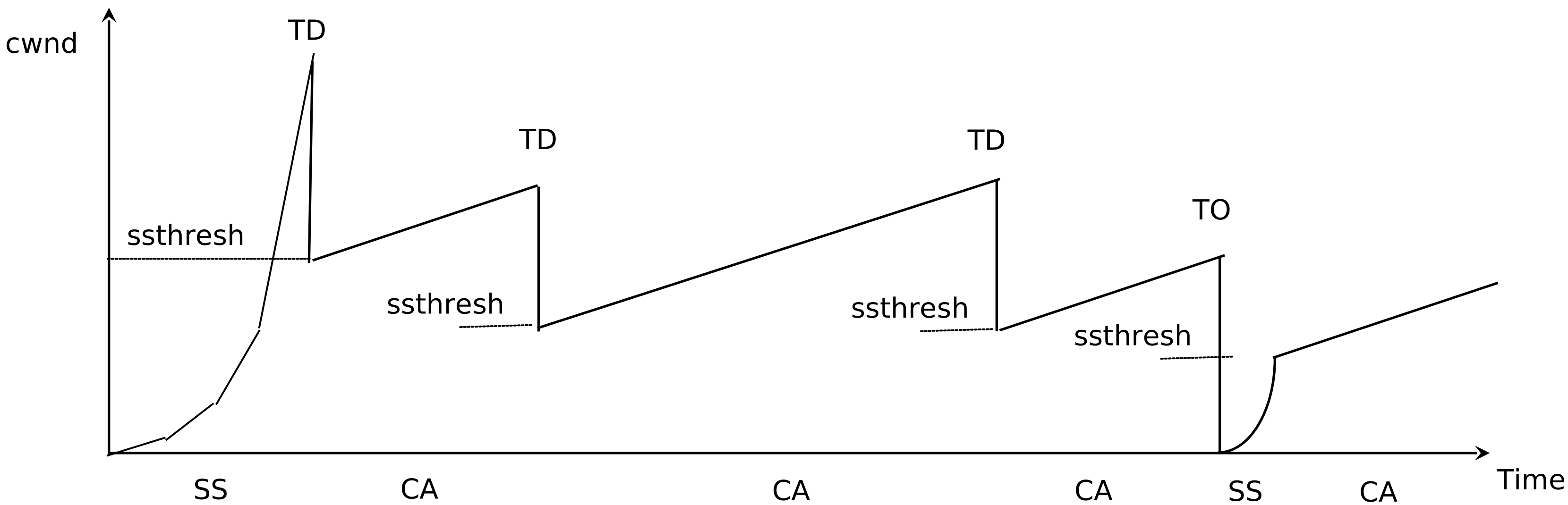
     2*cwnd per RTT till congestion

  2. Congestion avoidance(CA) – AIMD

     cwnd increase by 1 per RTT

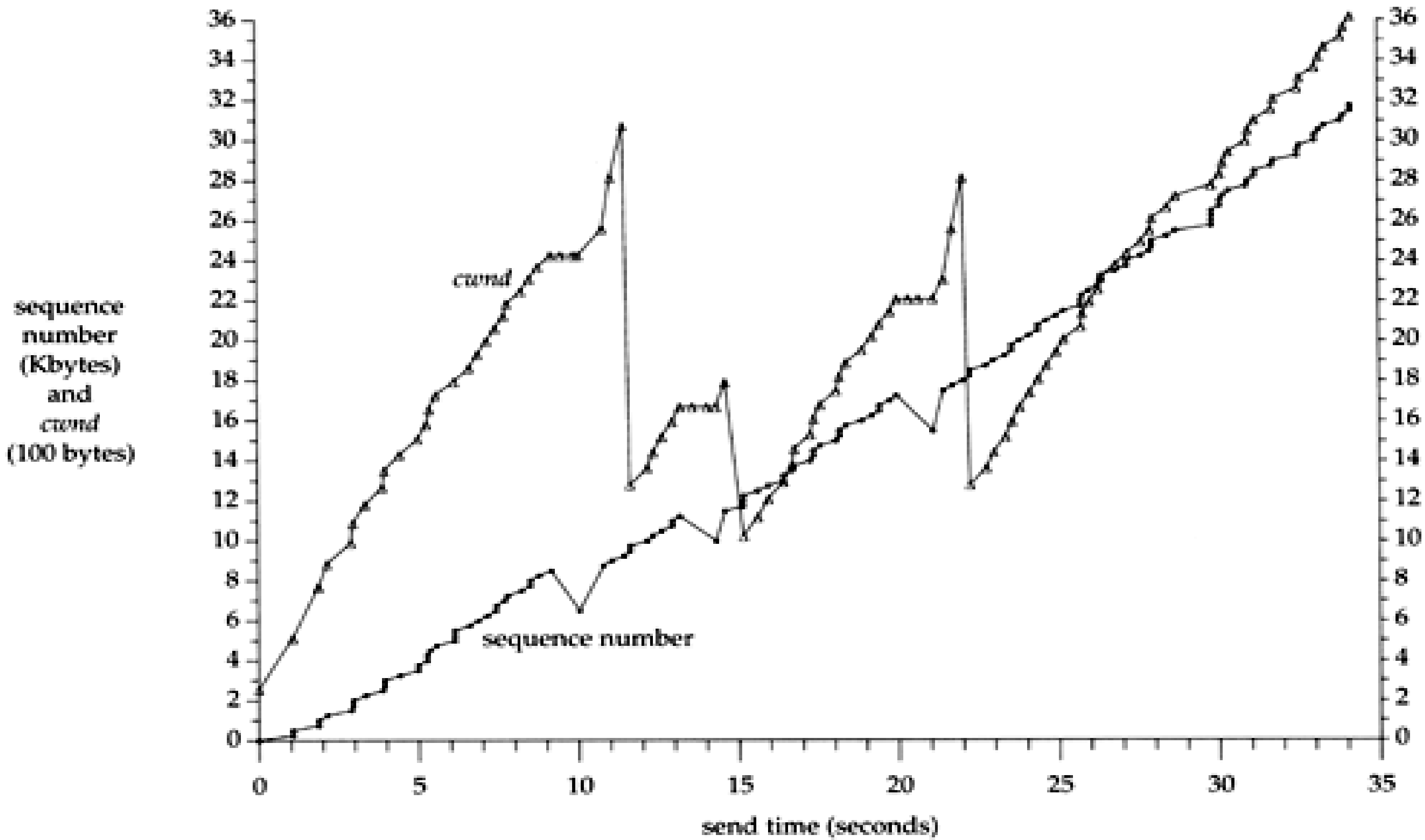
     - **3 duplicate ACKs ➔ cwnd = cwnd/2**
     - Timeout ➔ cwnd =1
     - In timeout ➔ timeout = 2*timeout

Recap : TCP/ Reno



SS – Slow Start
CA – Congestion Avoidance
TD – Three Duplicate ACKs
TO - Timeout

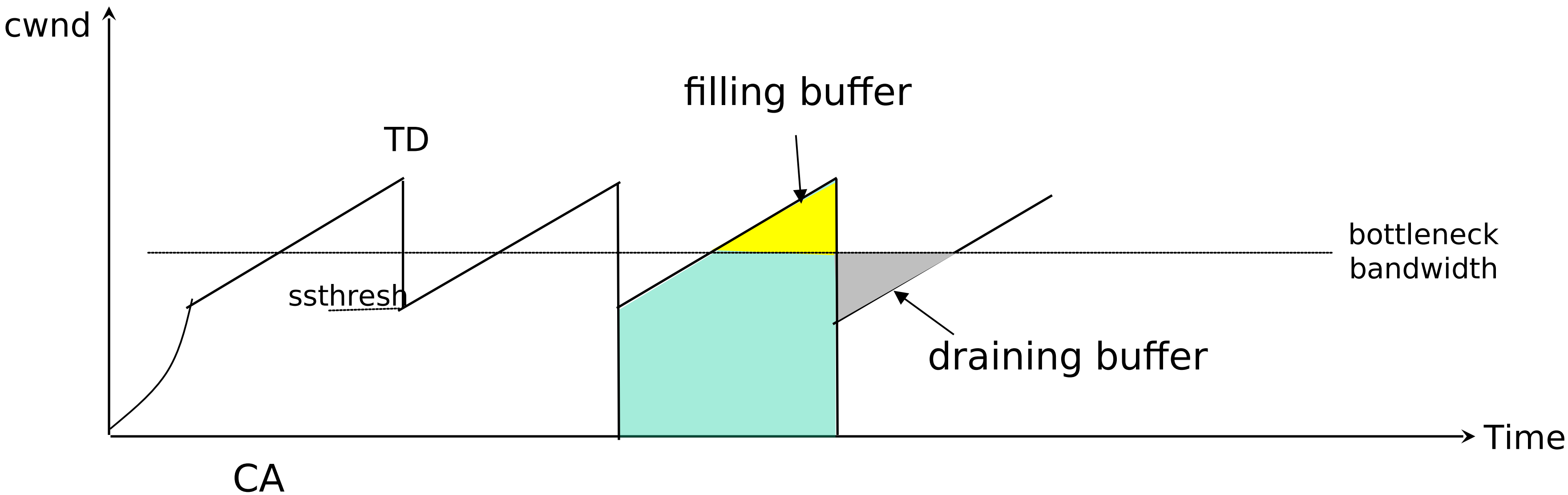
Recap : TCP/ Reno

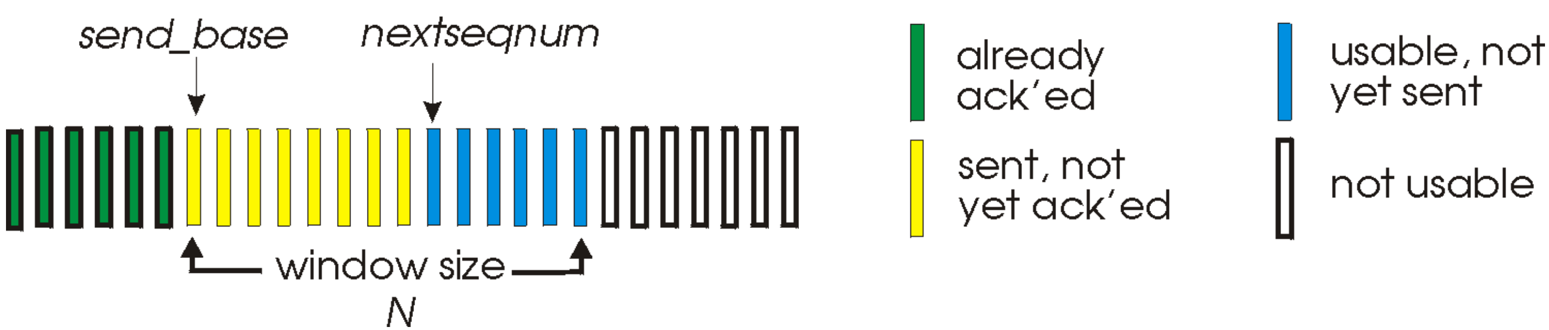


When cwnd is cut to half, why does sending rate not get cut?

Recap : TCP/ Reno

There is a filling and draining of buffers for each TCP flow.





TCP/ Reno Analysis

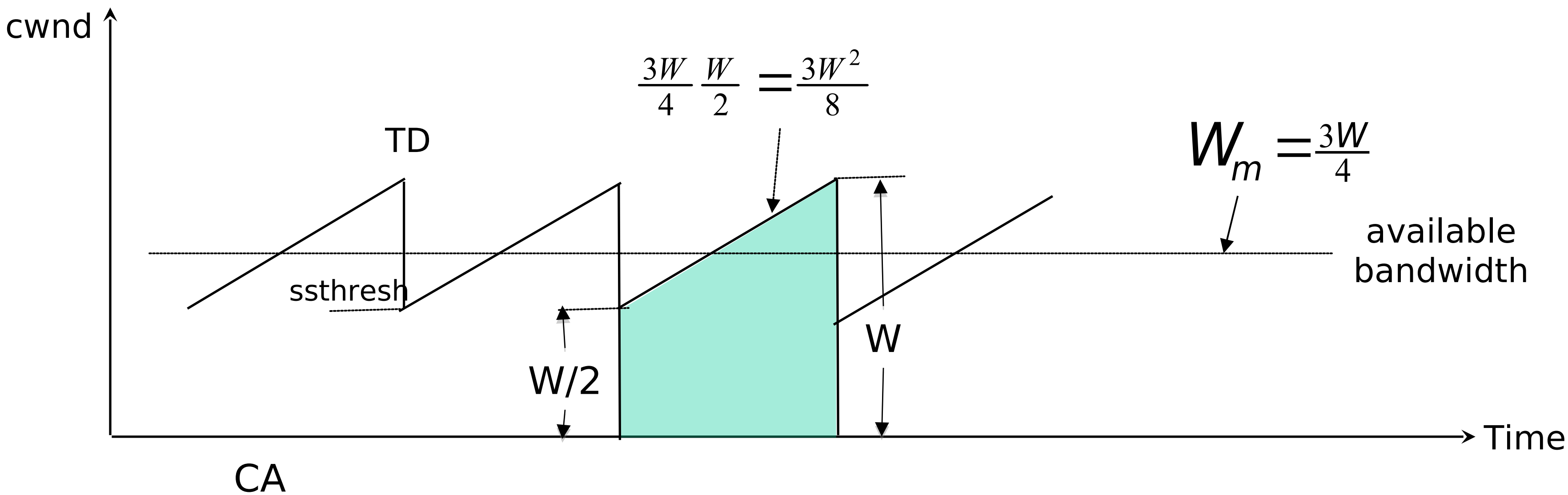
TCP/ Reno Throughput Analysis

- **Understand throughput in terms of**
  - RTT
  - Packet loss rate (p)
  - Packet size (S)

- **Throughput calculations**
  - Assume congestion avoidance and no timeouts occur
  - Mean window size W_m segments, round trip time RTT & pack size S
  - Throughput ≈ $\frac{W_m * S}{RTT}$ bytes/sec

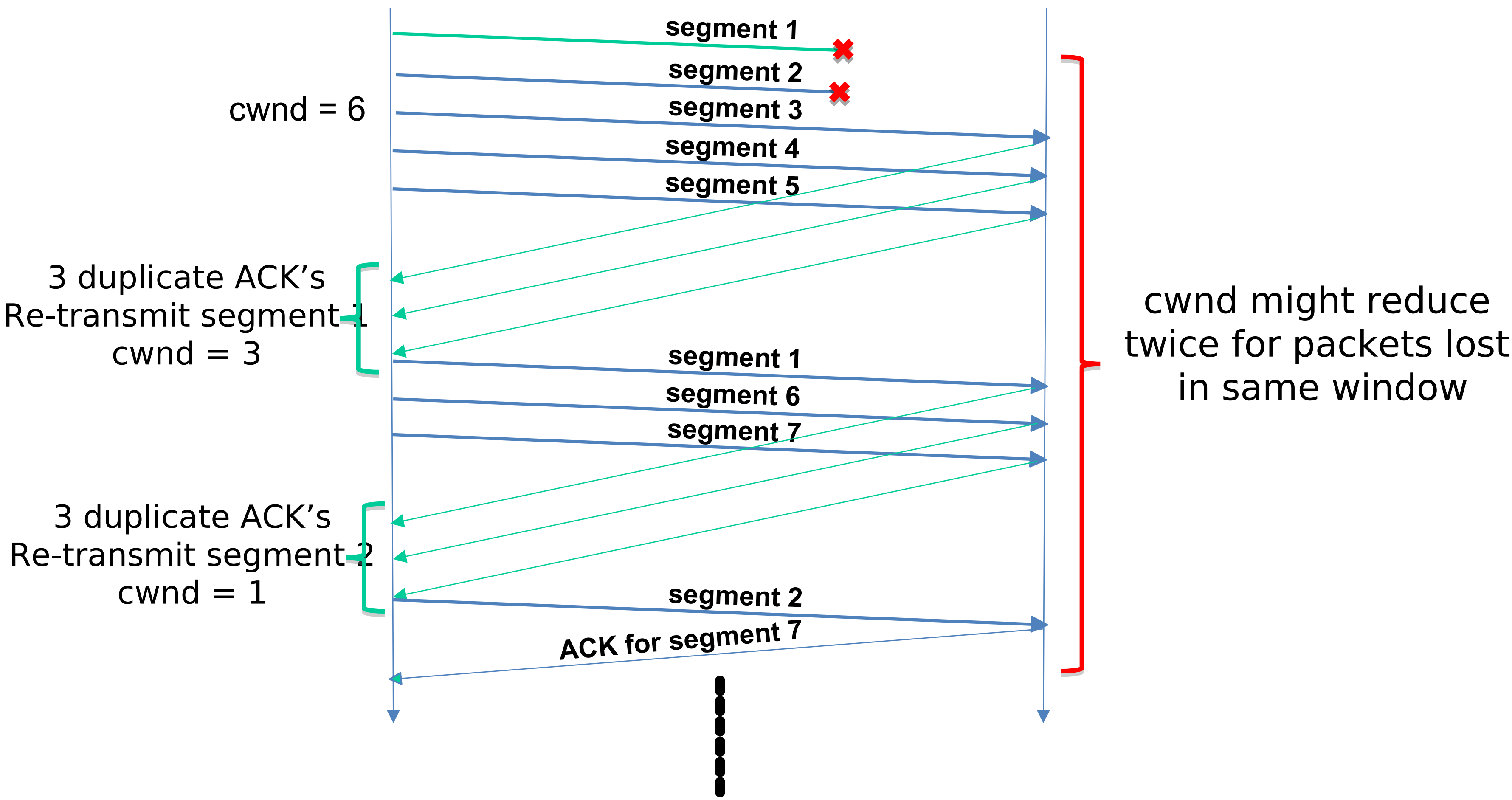
Deterministic Analysis

- Consider congestion avoidance
  - Assume one packet is lost per cycle
  - Total packets sent per cycle $= \frac{1}{2}*(W + W/2) * W/2 = 3W^2/8$
  - Packet loss (p) $= 1/(3W^2/8) = 8/(3W^2) \rightarrow W = \frac{\sqrt{8/3}}{\sqrt{p}} = \frac{1.6}{\sqrt{p}}$
- Throughput $= \frac{S*W_m}{RTT} = \frac{S}{RTT}\left(\frac{3}{4}\frac{1.6}{\sqrt{p}}\right) = \frac{1.2S}{RTT\sqrt{p}}$



TCP/ Reno Drawbacks

- Multiple packets lost simultaneously cannot be accounted f(



TCP/ Reno Drawbacks

- RTT unfairness
  - Flows with different RTT's grow their congestion windows differently
  - Users with shorter RTT ramp up faster!
  - On long distance links, RTT is high and cwnd takes longer to increase leading to underutilization of link.

- Synchronized losses
  - Simultaneous packet loss events for multiple competing flows.

New Protocol Necessary!!

Desired Characteristics in TCP

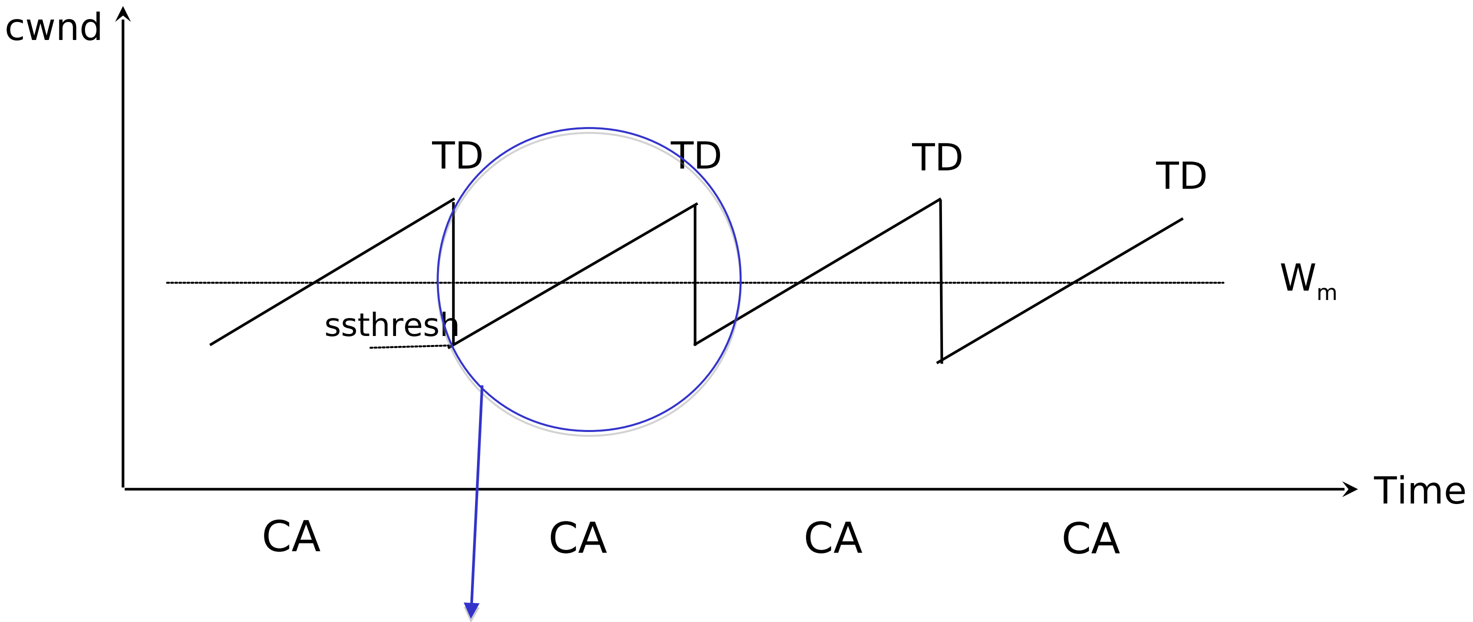
- Adaptive schemes that grow the congestion window dep
network conditions
  - Scalable
  - RTT Fairness
  - Faster convergence to better utilize full bandwidth

TCP BIC

http://www.land.ufrj.br/~classes/coppe-redes-2007/projeto/BIC-TCP-infocom-04.pdf

Growth functions

- Consider TCP/Reno growth function



Grows linearly throughout

TCP BIC

Binary Increase Congestion Control (BIC) algorithm

PHASE 1

- cwnd < low_wind, follows TCP
  - ACK received : cwnd = cwnd + 1
  - Loss event: cwnd = cwnd/2

PHASE 2

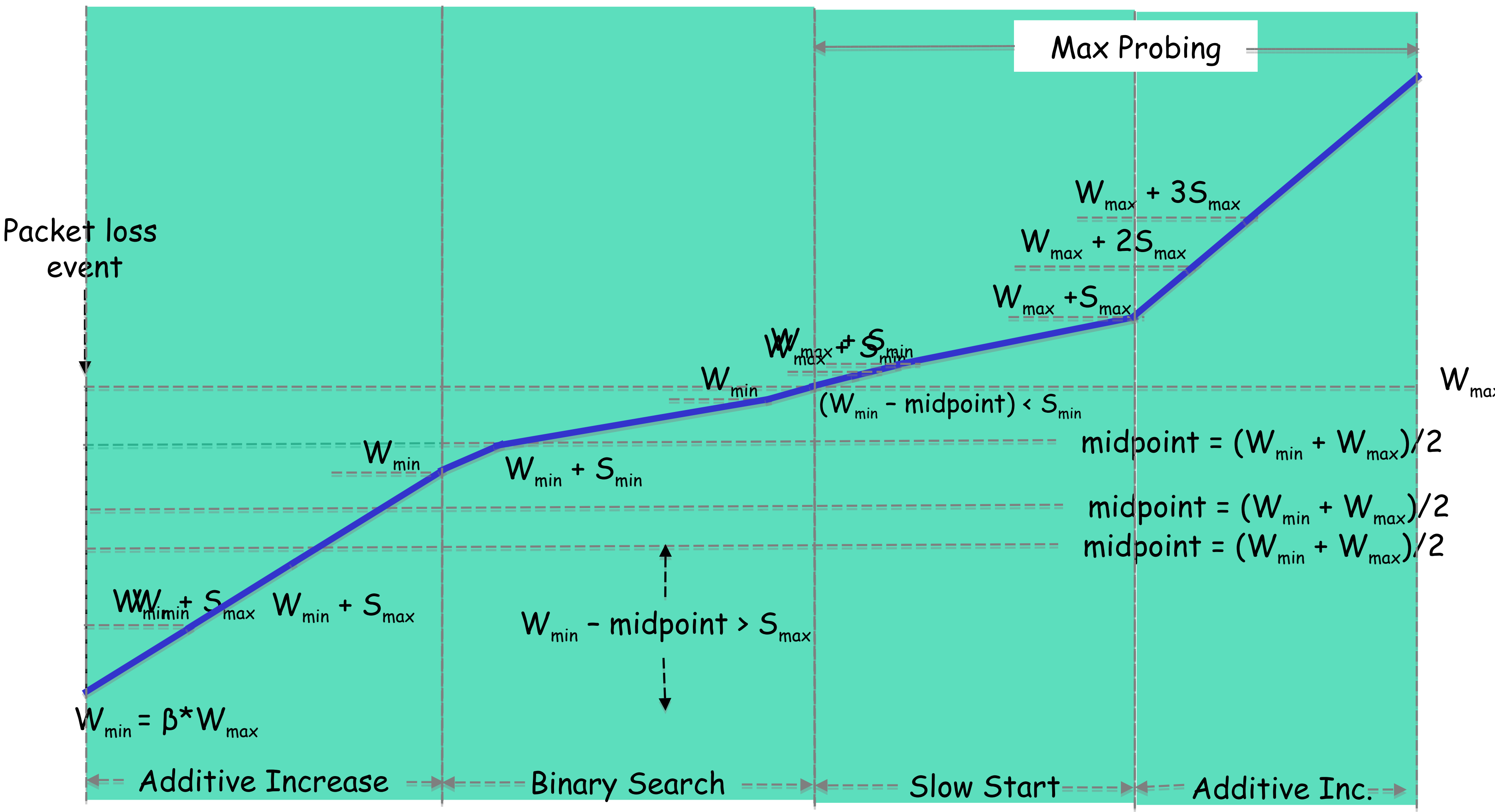
- cwnd > low_wind, follows BIC

BIC Algorithm

- Some preliminaries
  - βmultiplicative decrease factor
  - W_{max} = cwnd size before the reduction
  - $W_{min} = \beta * W_{max}$ – just after reduction
  - midpoint = (Wmax + Wmin)/2

BIC performs binary search between W_{max} and W_{min} looking for the midpoint.

BIC Algorithm



BIC Algorithm

```
while (cwnd != W_max){
If ((W_min - midpoint) > S_max)
      cwnd = cwnd + S_max          → Additive Increase
   else
      If ((W_min – midpoint) < S_min)
            cwnd = W_max
      else
            cwnd = midpoint        → Binary Search
   If (no packet loss)
      W_min = cwnd
   else
      W_min = β*cwnd
      W_max = cwnd
   midpoint = (W_max + W_min)/2
   }
```

BIC Algorithm

while ($cwnd >= W_{max}$){

If ($cwnd < W_{max} + S_{max}$)

$cwnd = cwnd + S_{min}$ → Slow Start

else

$cwnd = cwnd + S_{max}$ → Additive Increase

If (packet loss)

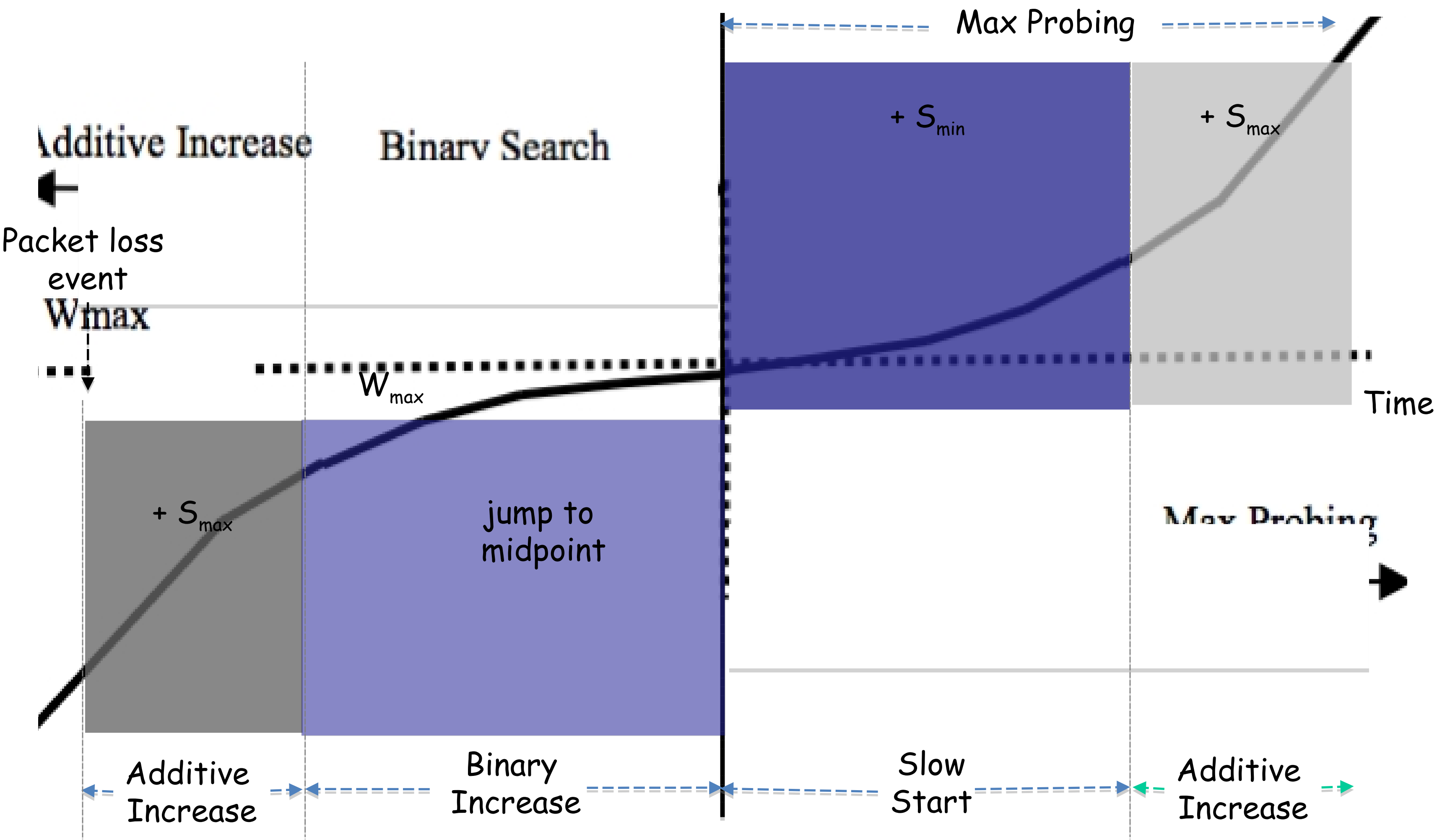
$W_{min} = \beta * cwnd$

$W_{max} = cwnd$

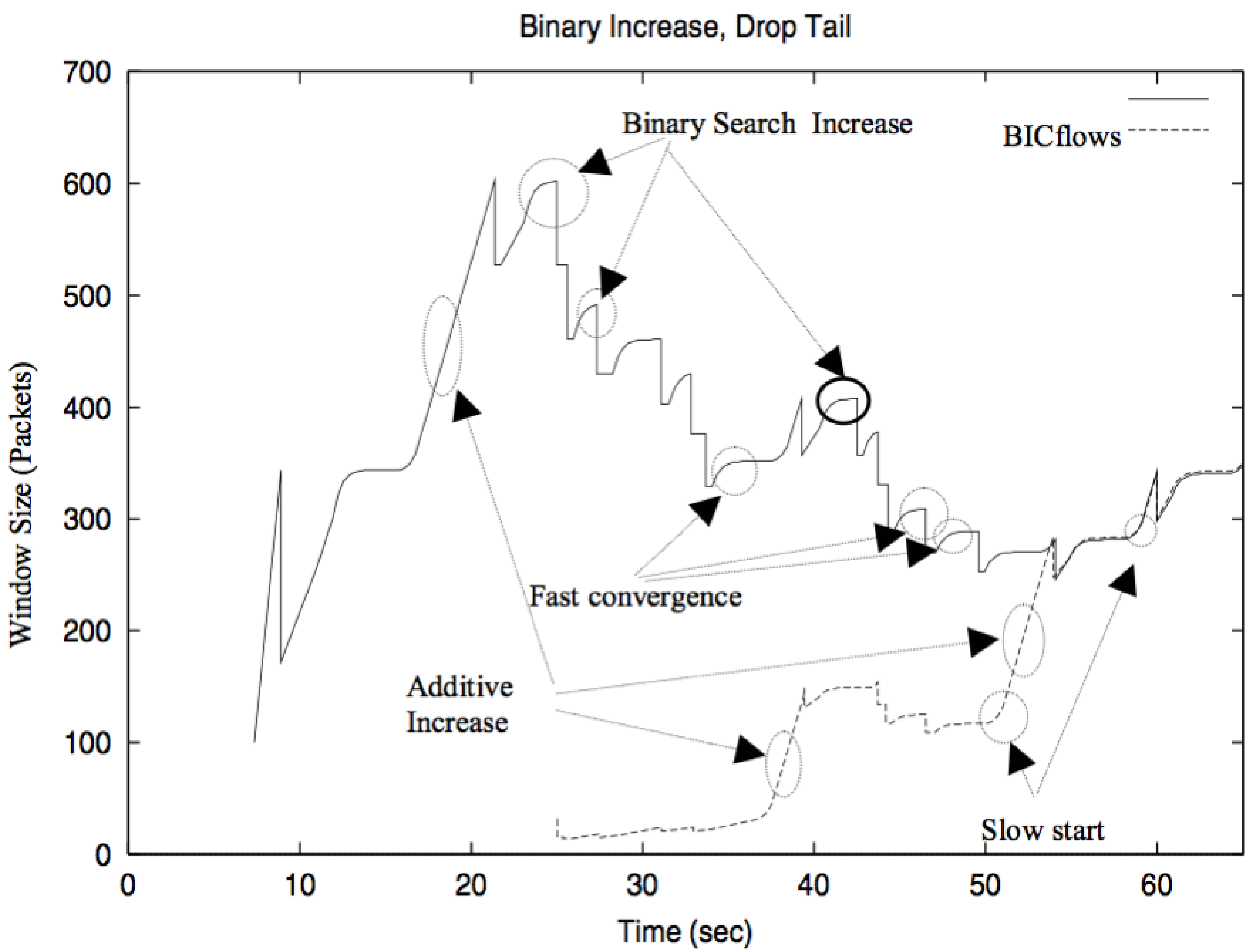
}

Max Probing

TCP BIC - Summary



TCP BIC in Action



TCP BIC Advantages

- Scalability: quickly scales to fair BW share
- Fairness and convergence: Achieves better fairness and faster convergence
- Slow Growth around W_{max} ensures that unnecessary timeouts do not occur.

TCP BIC Drawbacks

- cwnd growth is aggressive for TCP with short RTT or low speed
  - Short RTT makes cwnd ramp up soon
- Still dependent on RTT
  - Proportional to inverse square of the RTT like TCP/ Reno
- Complex window growth function
  - Difficult for analysis and actual implementation

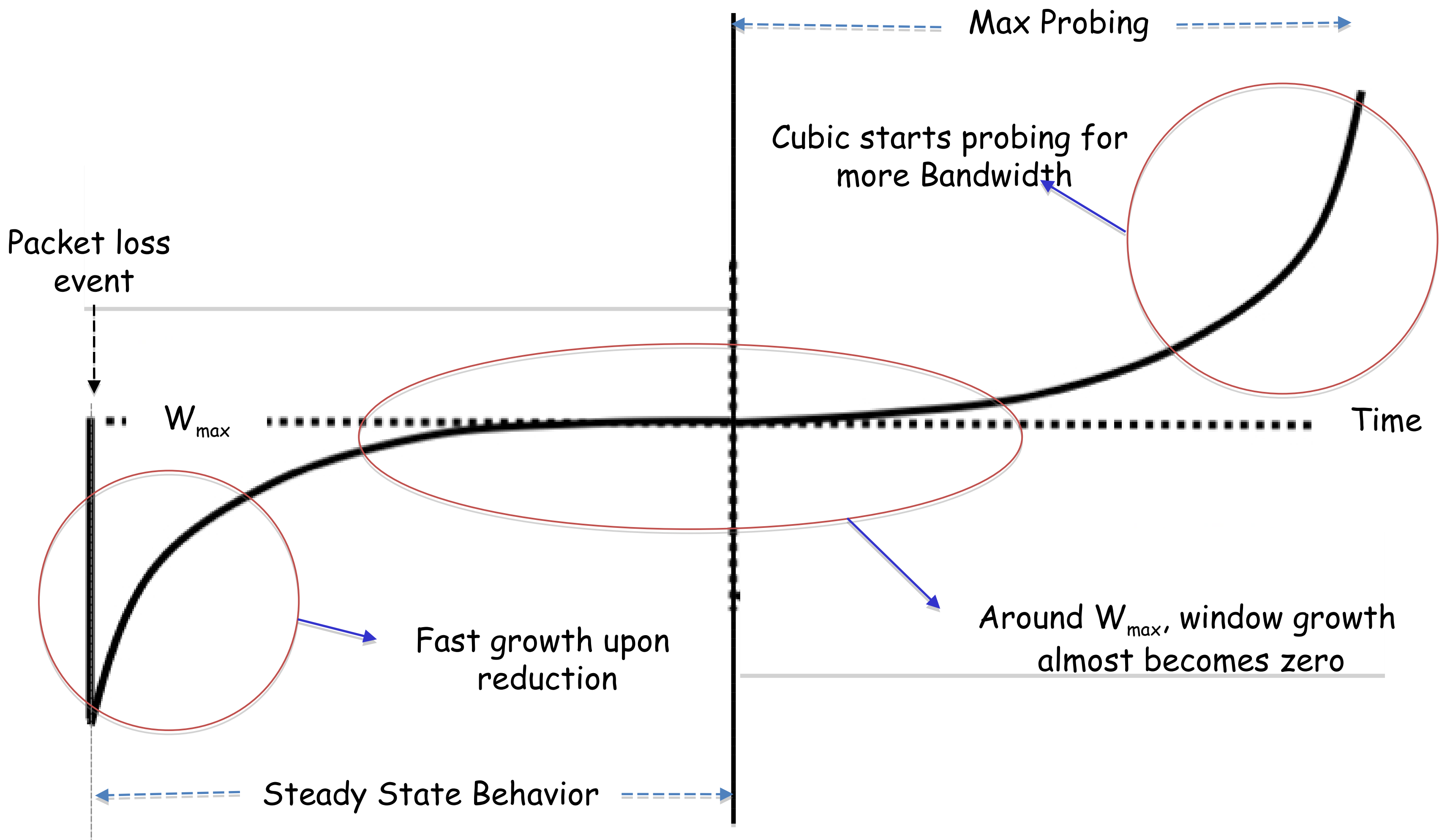
TCP Cubic

http://www4.ncsu.edu/~rhee/export/bitcp/cubic-paper.pdf

TCP Cubic

- cwnd = $C(t - K)^3 + W_{max}$
  - W_{max} = cwnd before last reduction
  - βmultiplicative decrease factor
  - C scaling factor
  - $$K = \sqrt[3]{W\beta / C}$$
  - t is the time elapsed since last window reduction

TCP CUBIC


Max Probing
Cubic starts probing for more Bandwidth
Packet loss event
W_{max}
Time
Fast growth upon reduction
Around W_{max}, window growth almost becomes zero
Steady State Behavior

TCP Cubic Advantages

- Good RTT fairness
  - Growth dominated by t, competing flows have same t after synchronized packet loss

- Real-time dependent
  - Similar to BIC but linear increases are time dependent
  - Does not depend on ACK's like TCP/ Reno

- Scalability
  - Cubic increases window to W_{max} (or its vicinity) quickly and keeps it there longer

TCP Cubic Drawbacks

- Slow Convergence
  - Flows with higher cwnd are more aggressive initially
  - Prolonged unfairness between flows

- Bandwidth Delay Products
  - Linear increase artefacts

Chapter 3: Summary

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

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"