Chapter 3
Transport Layer

Transport layer: overview

Our goal:

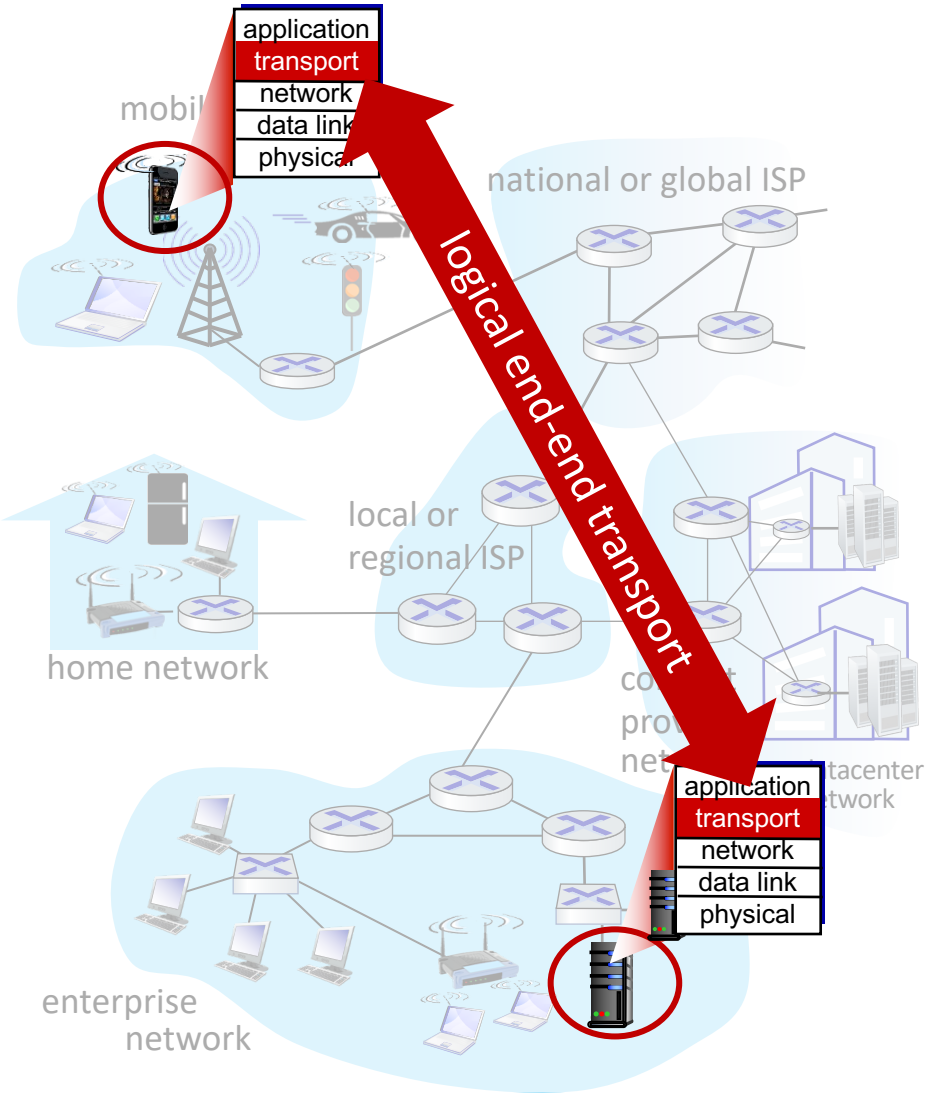
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion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

Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Transport services and protocols

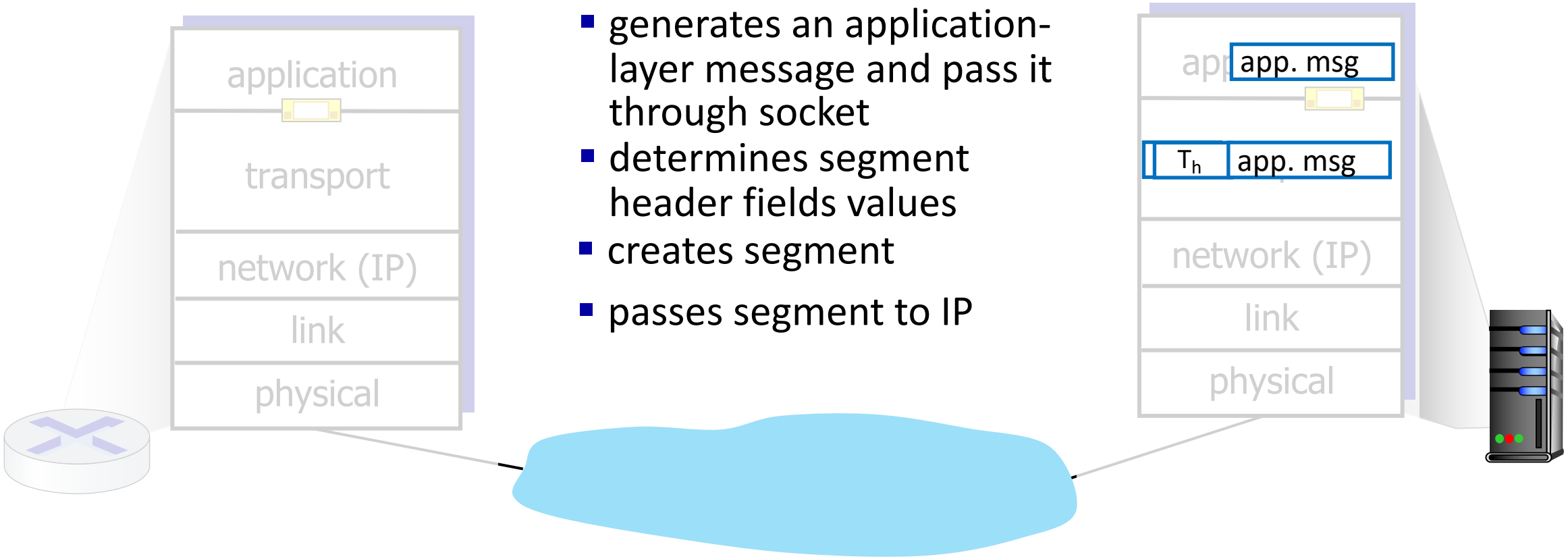
- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



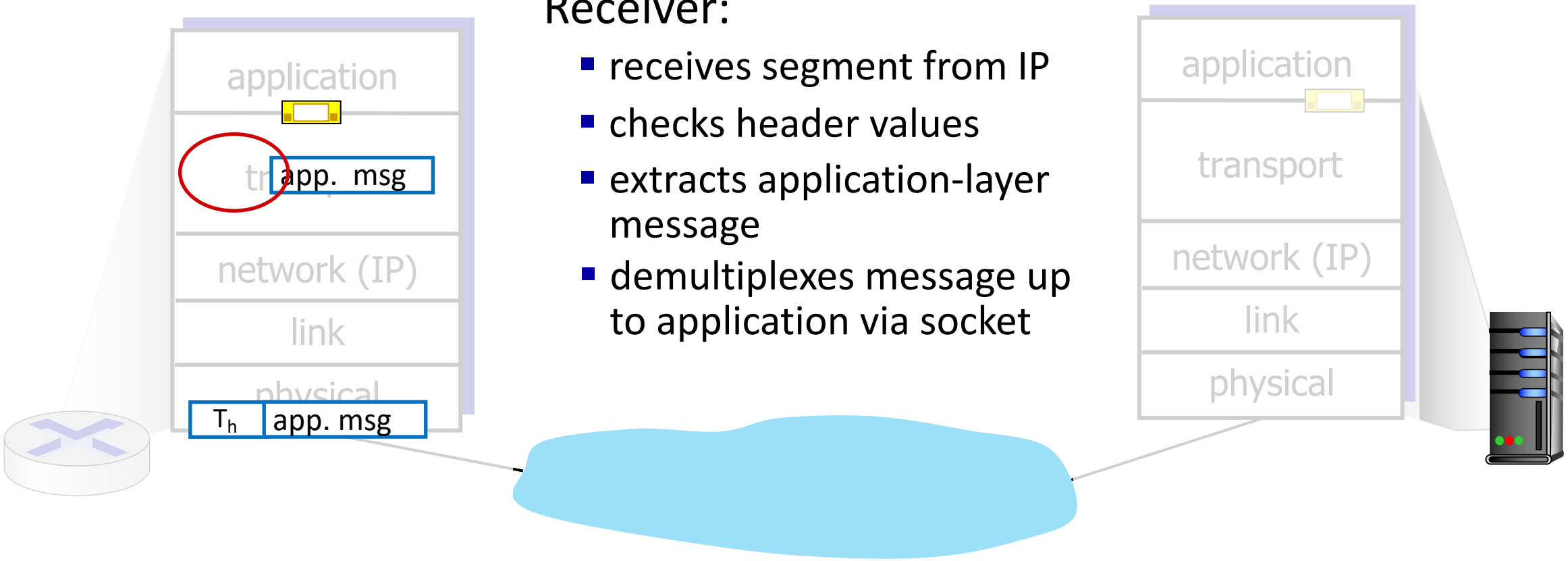
Transport Layer Actions

Sender:

- generates an application-layer message and pass it through socket
- determines segment header fields values
- creates segment
- passes segment to IP



Transport Layer Actions

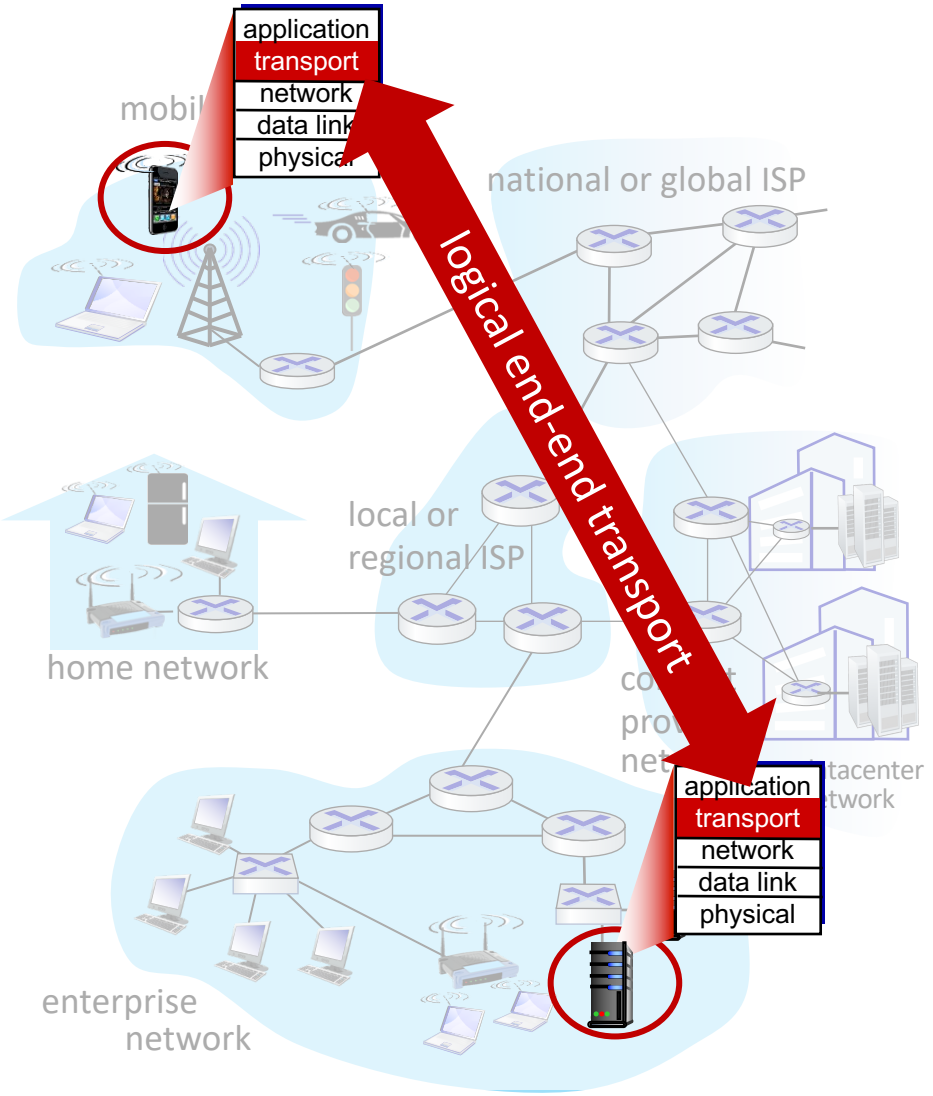


Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

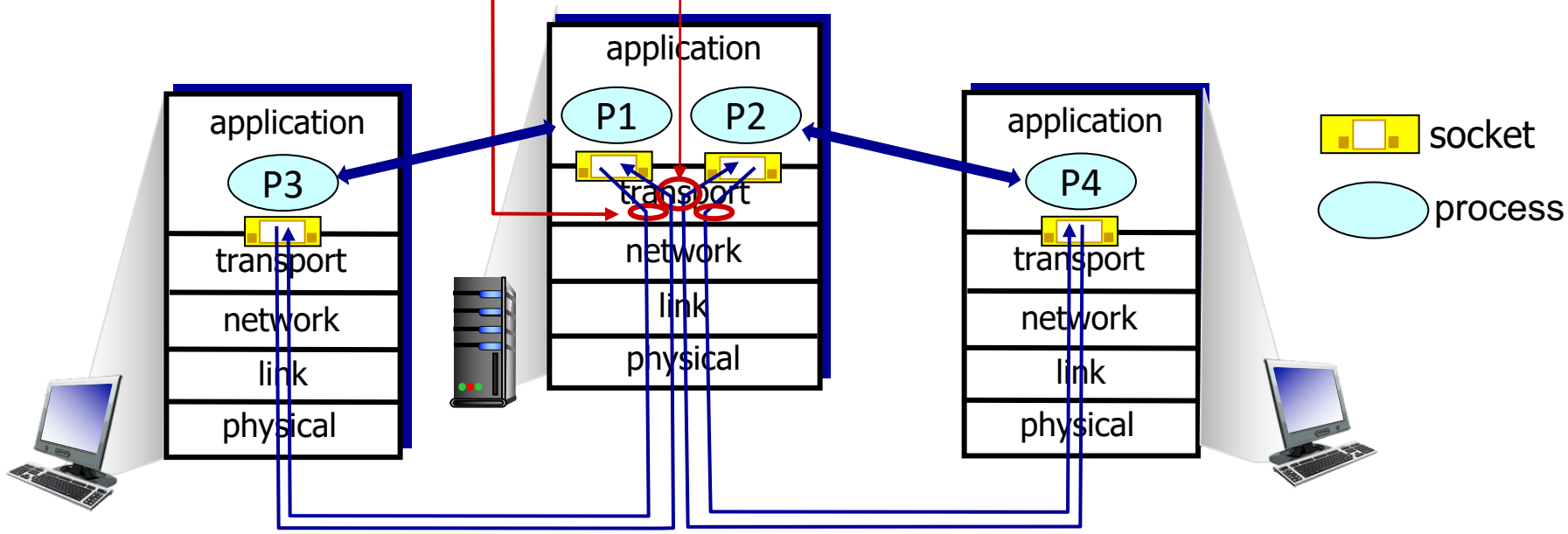
Multiplexing/demultiplexing

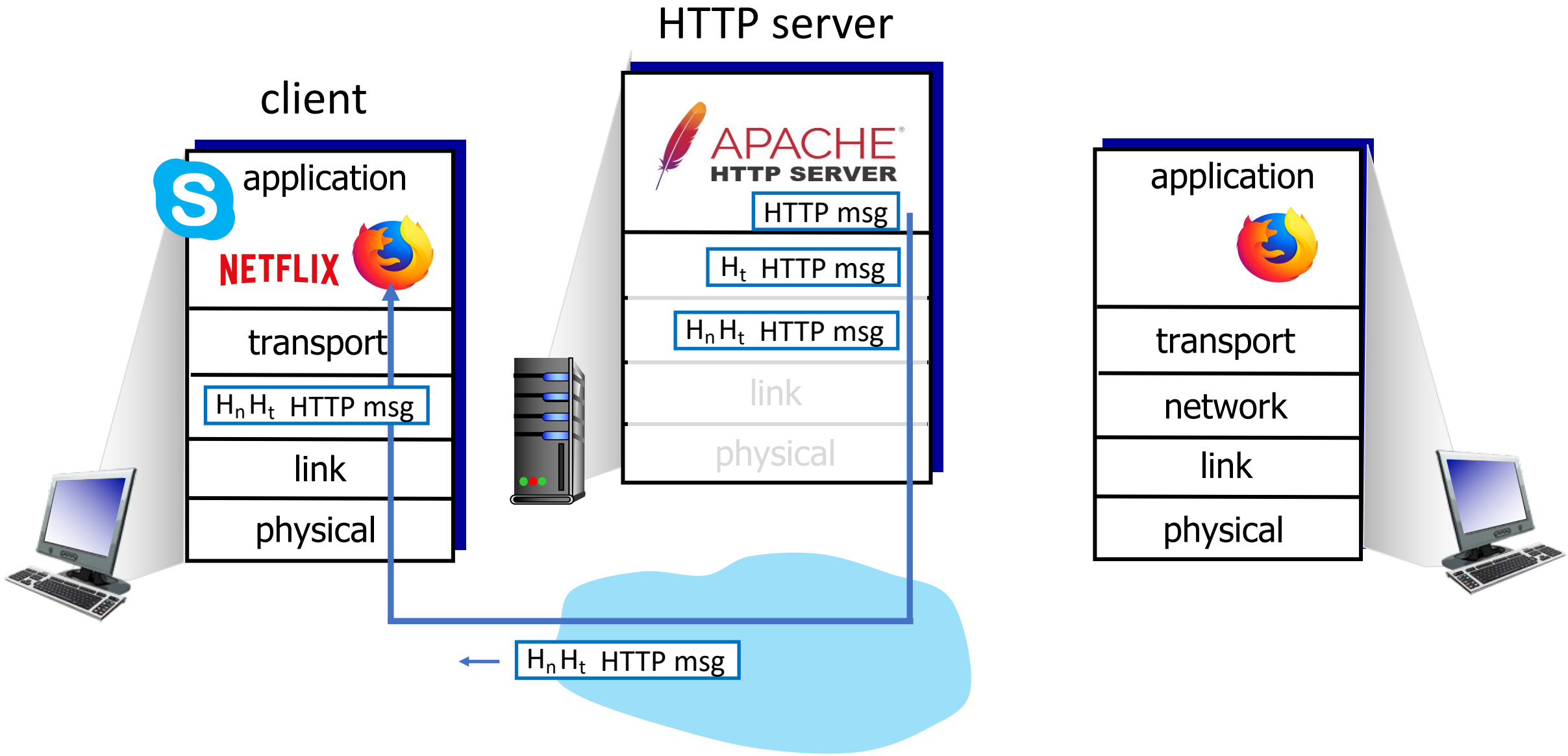
multiplexing as sender:

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

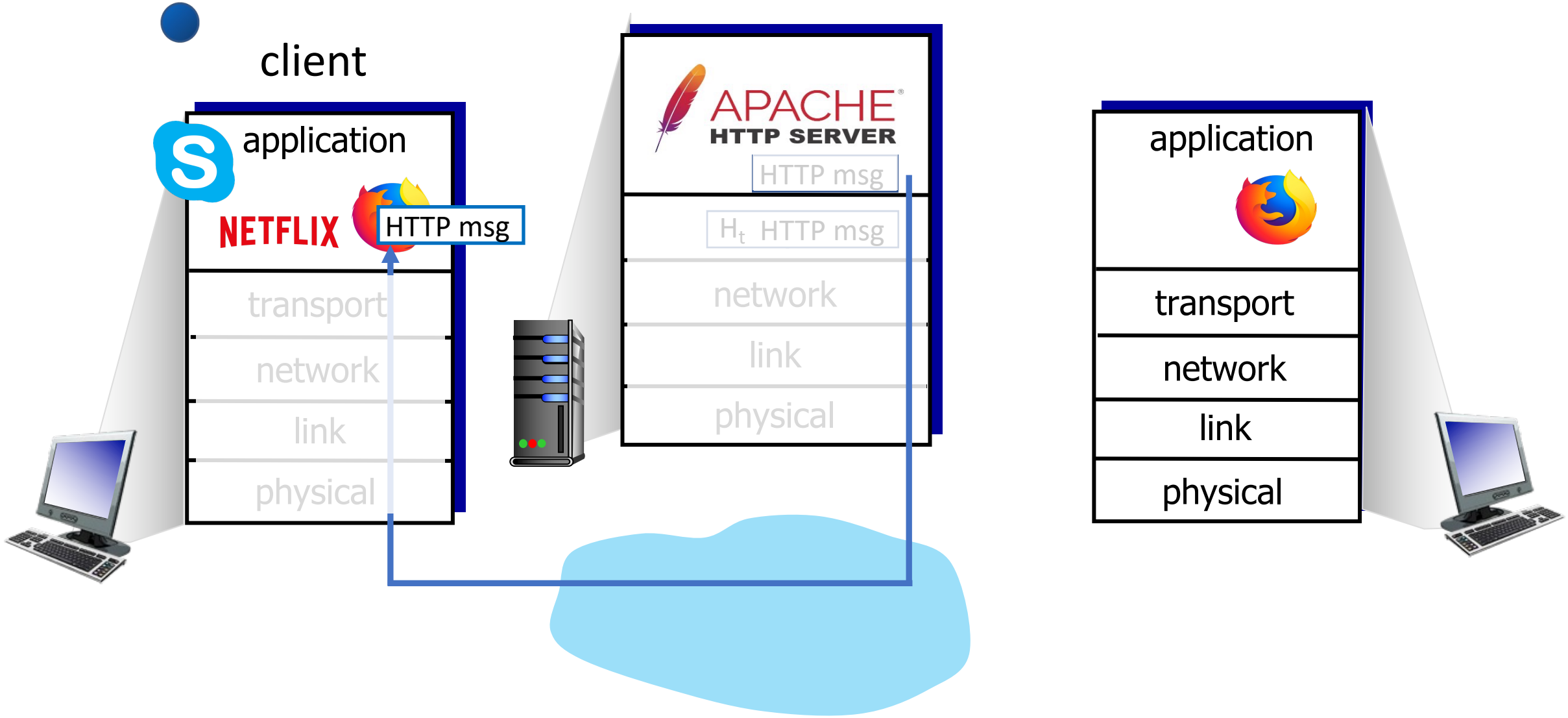
demultiplexing as receiver:

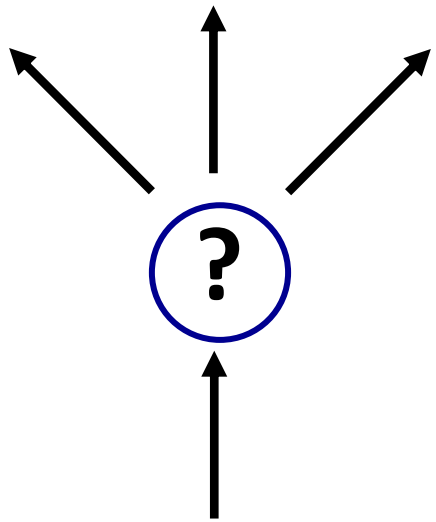
use header info to deliver received segments to correct socket



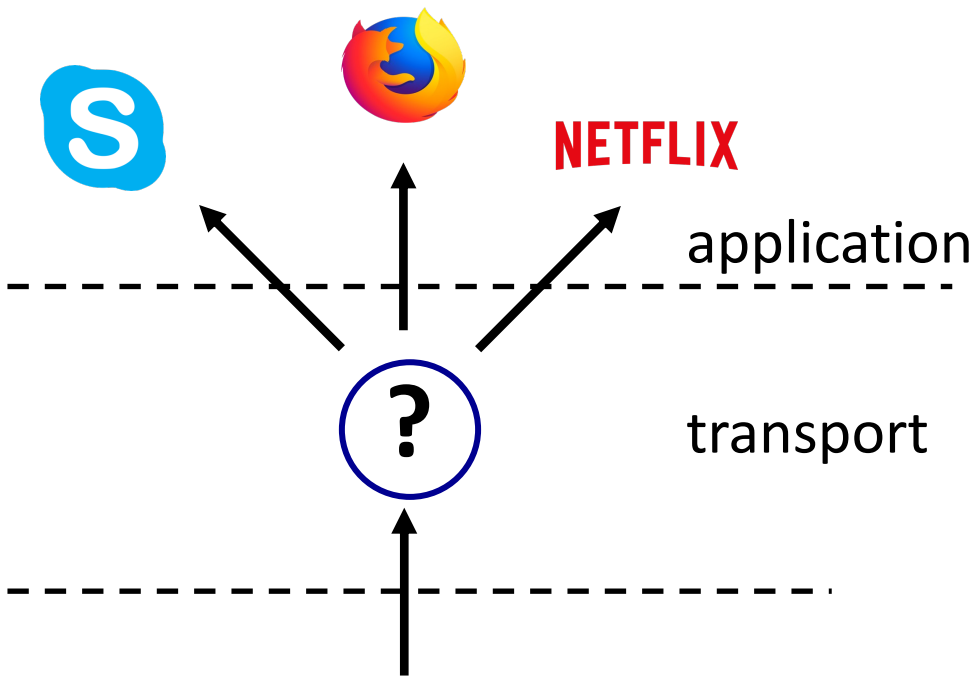

client
application
NETFLIX
transport
$H_n H_t$ HTTP msg
link
physical
HTTP server
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
$H_n H_t$ HTTP msg
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg

Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?





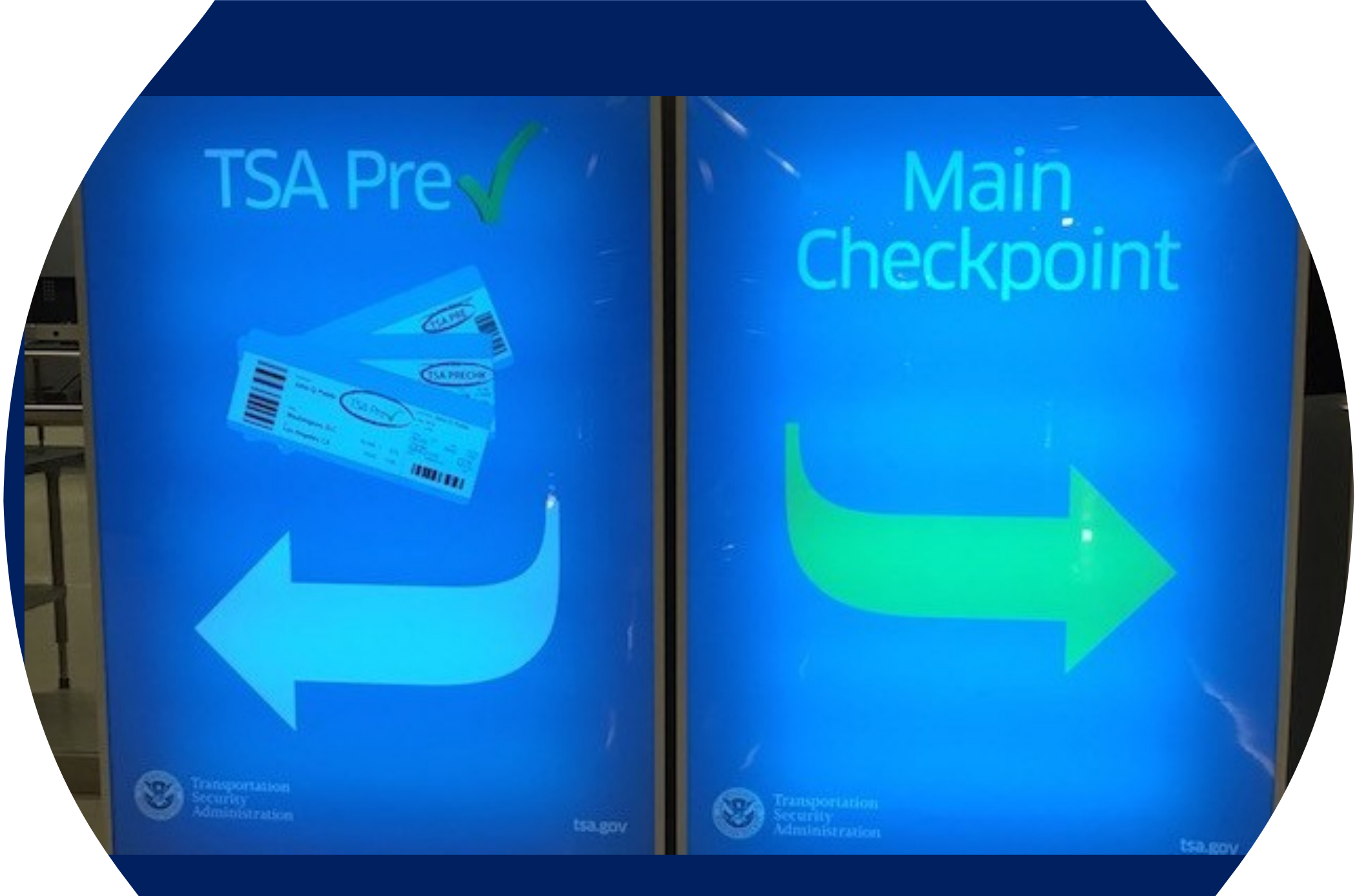
de-multiplexing

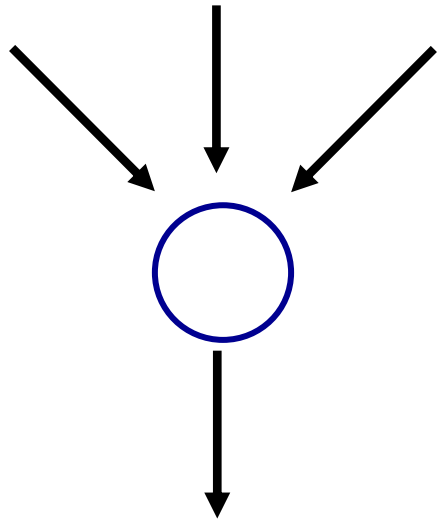


de-multiplexing

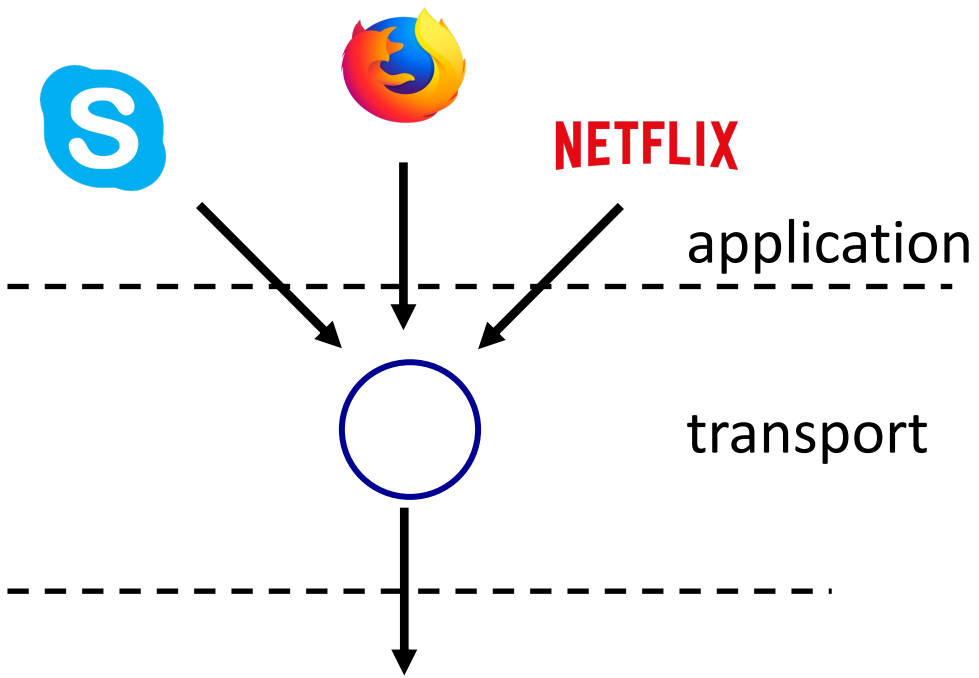
Demultiplexing


AIRFRANCE
ECONOMY
SKYTEAM
AIRFRANCE
SKY
PRIORITY™
SKYTEAM


TSA Pre✓
TSA PRE
TSA PRECHK
Transportation Security Administration
tsa.gov
Main Checkpoint
Transportation Security Administration
tsa.gov



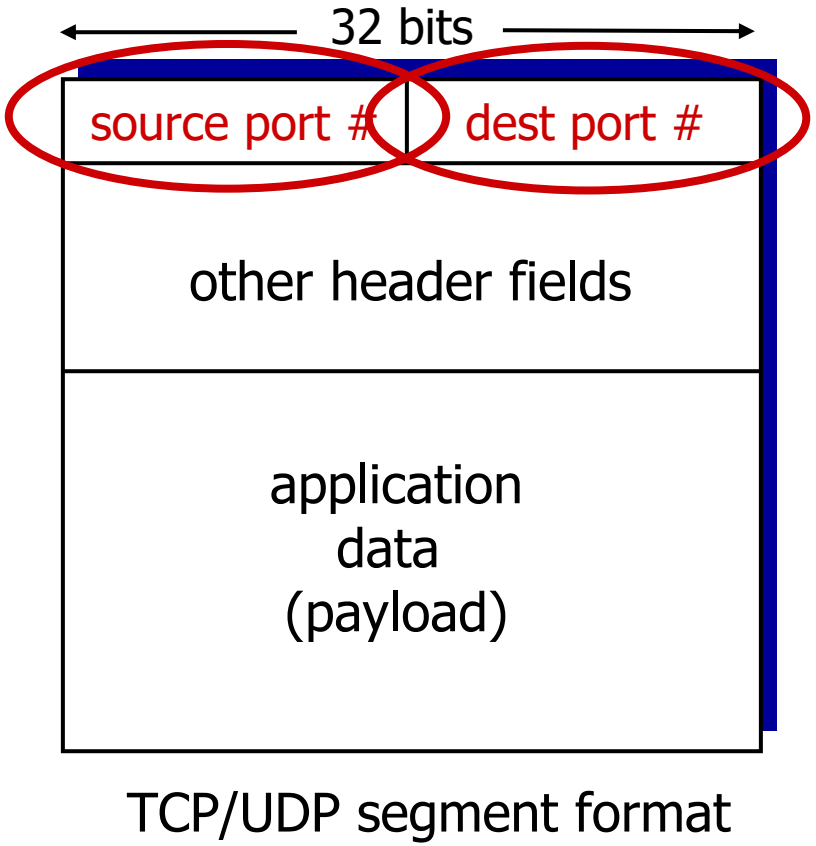
multiplexing


NETFLIX
application
transport
multiplexing


Multiplexing

How demultiplexing works

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



TCP/UDP segment format

Connectionless demultiplexing

Recall:

- when creating socket, must specify *host-local* port #:

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

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

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

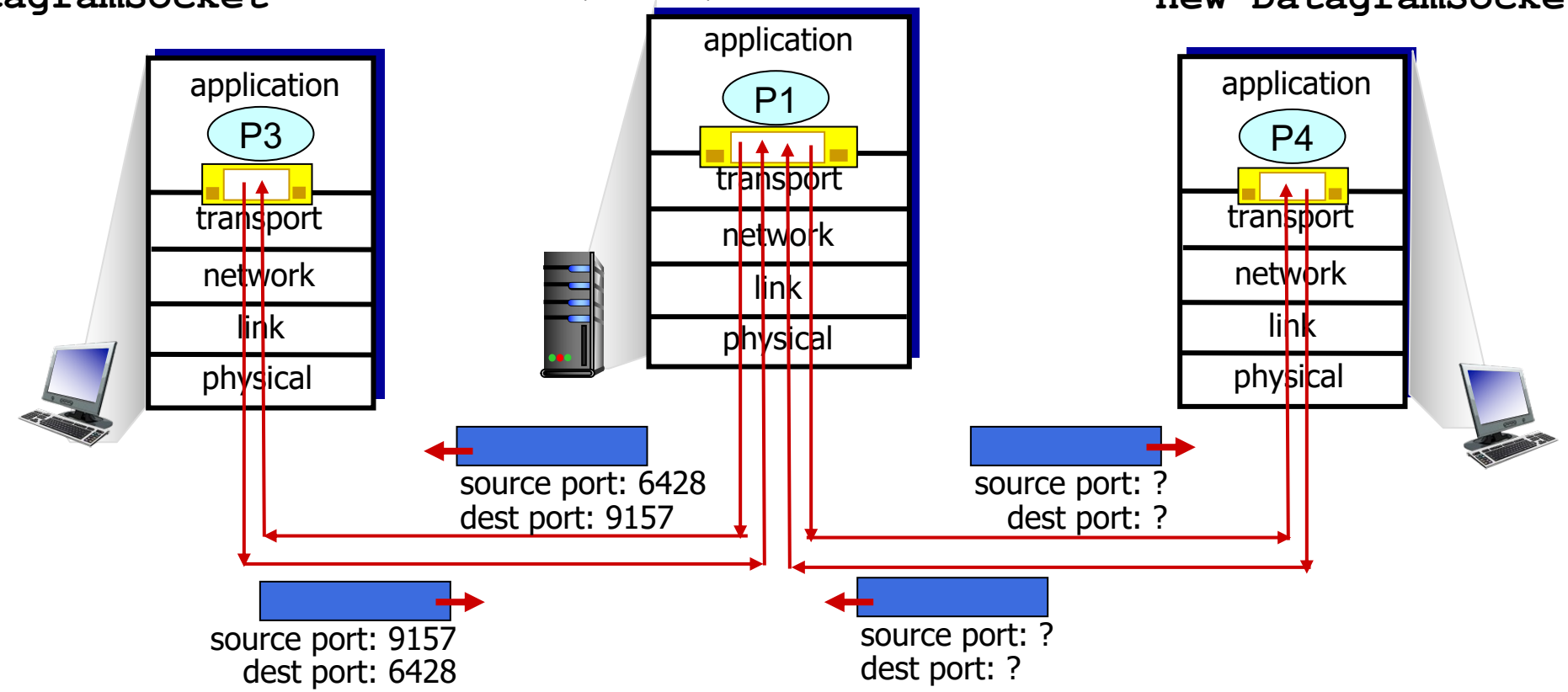
IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

```
DatagramSocket mySocket2 =
 new DatagramSocket
 (9157);
```

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

```
DatagramSocket mySocket1 =
 new DatagramSocket (5775);
```

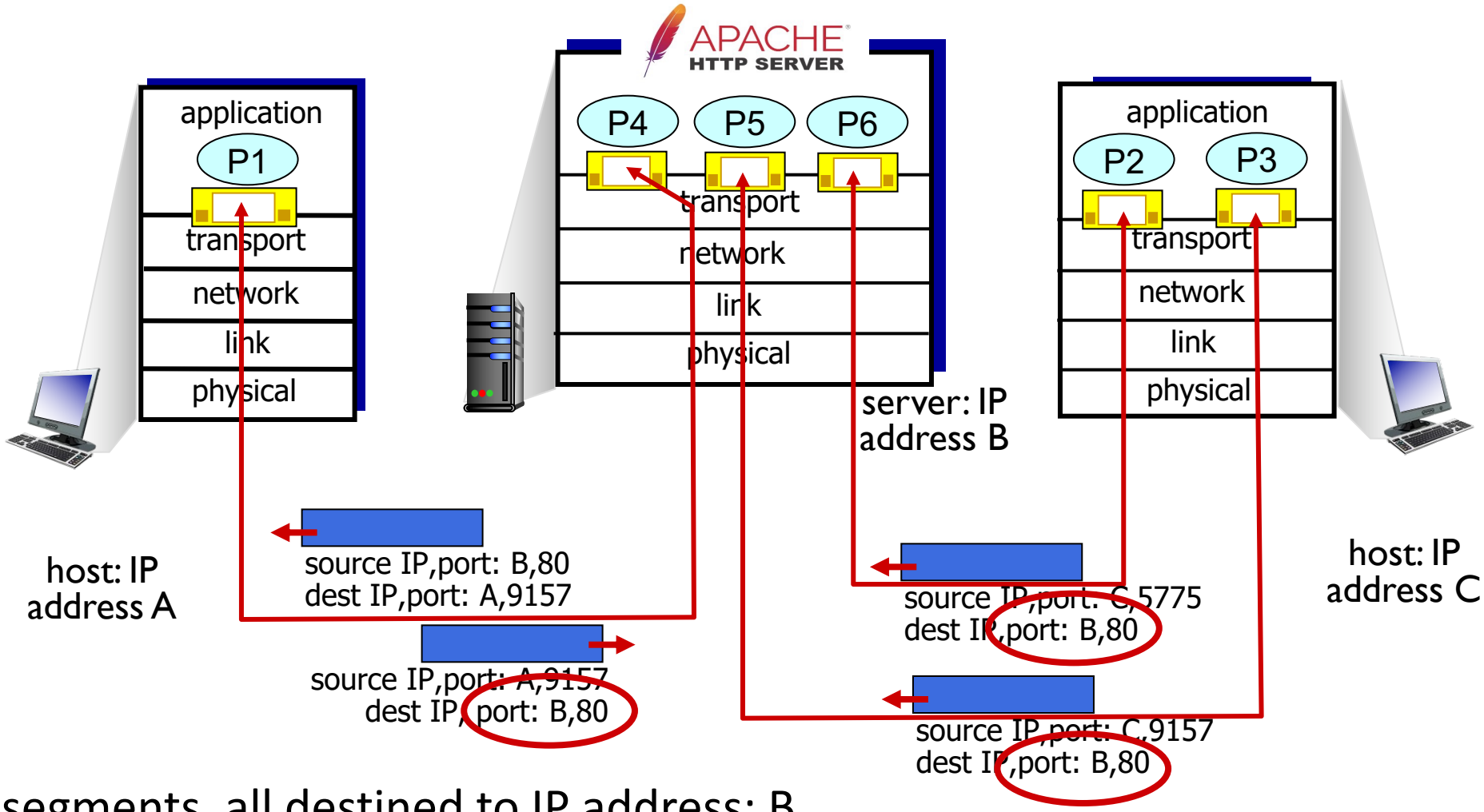


Connection-oriented demultiplexing

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

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

Connection-oriented demultiplexing: example



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

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

UDP: User Datagram Protocol

- "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 RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

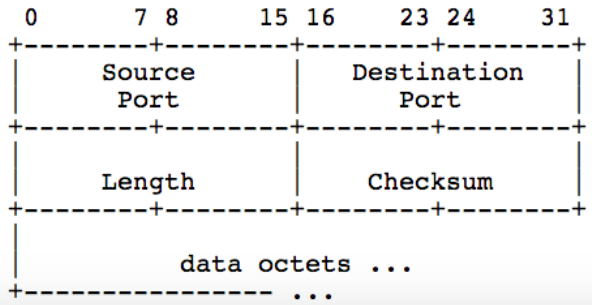
User Datagram Protocol

Introduction

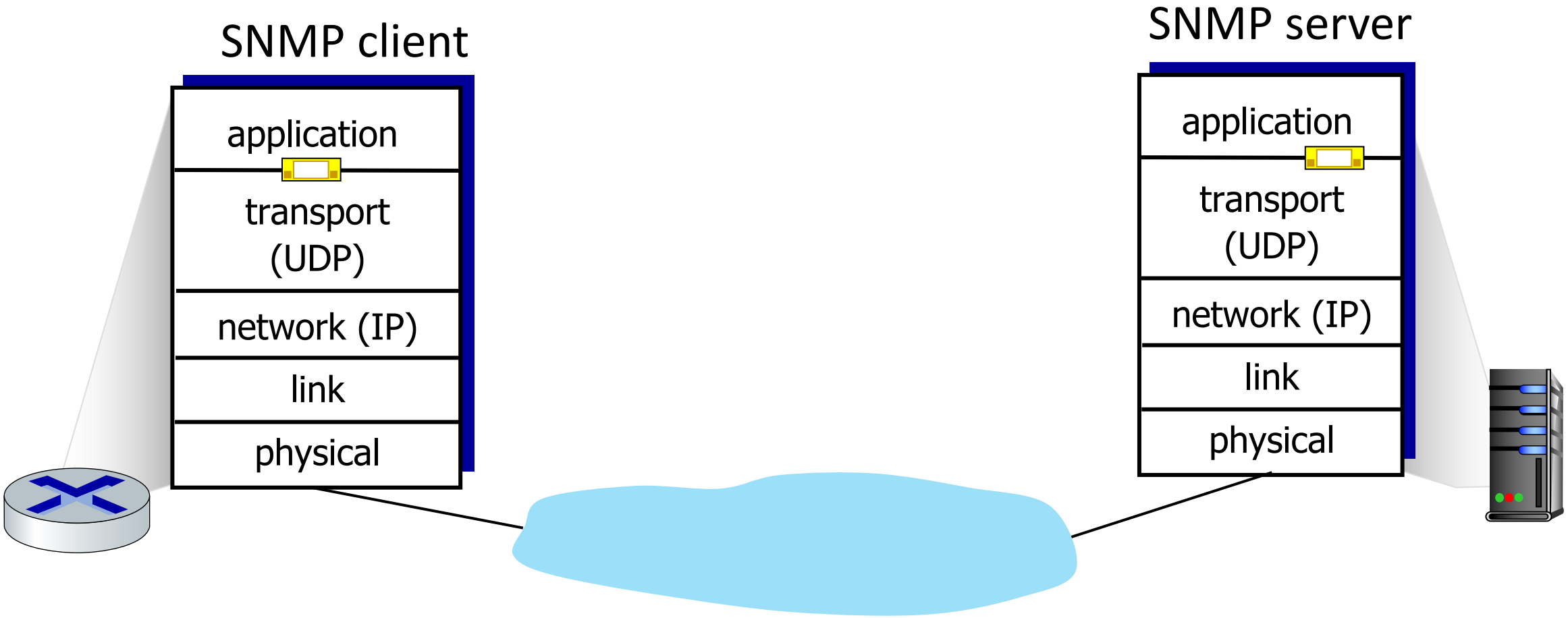
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

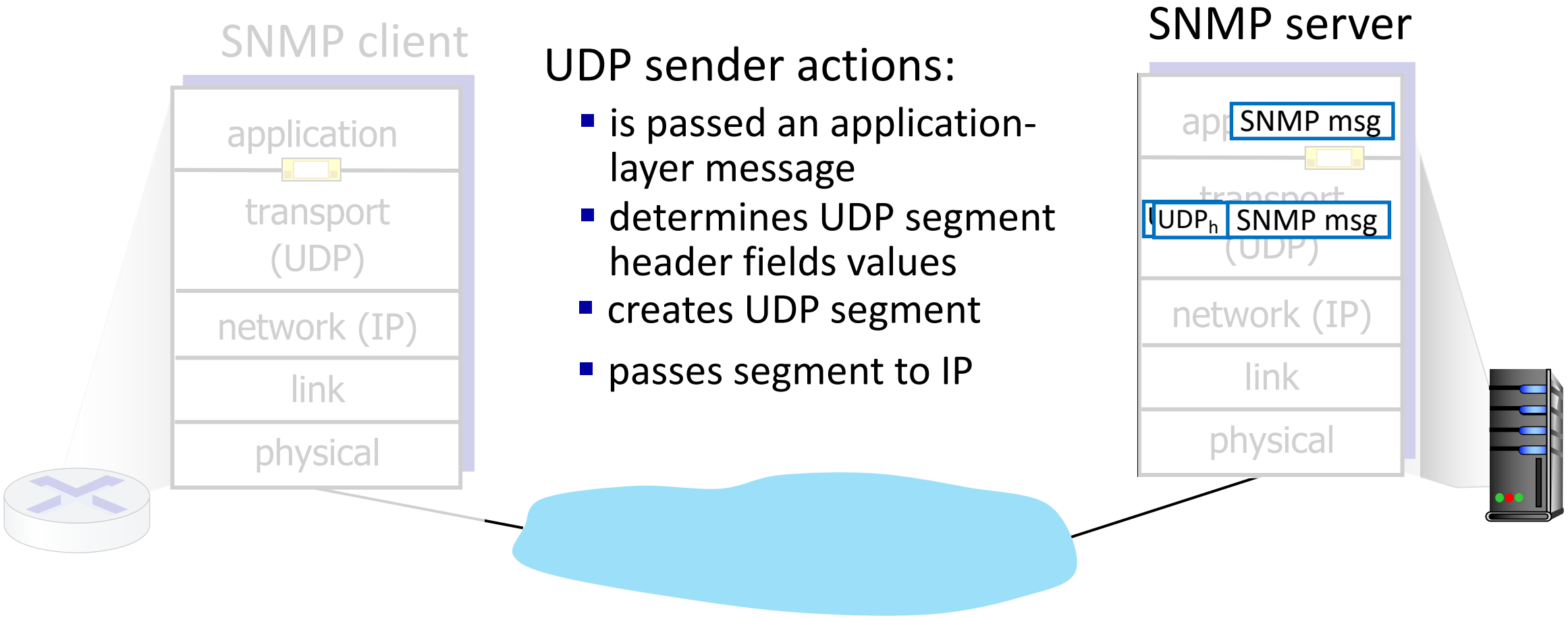
Format



UDP: Transport Layer Actions



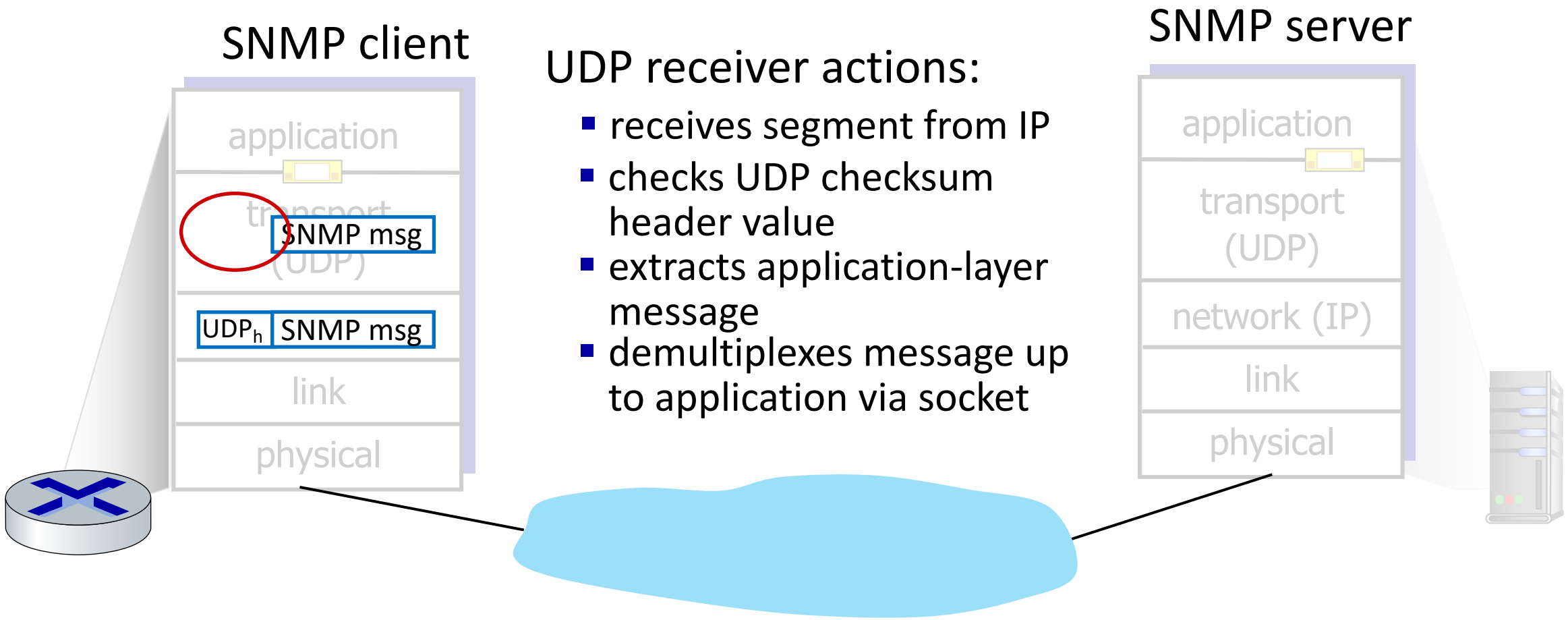
UDP: Transport Layer Actions



UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

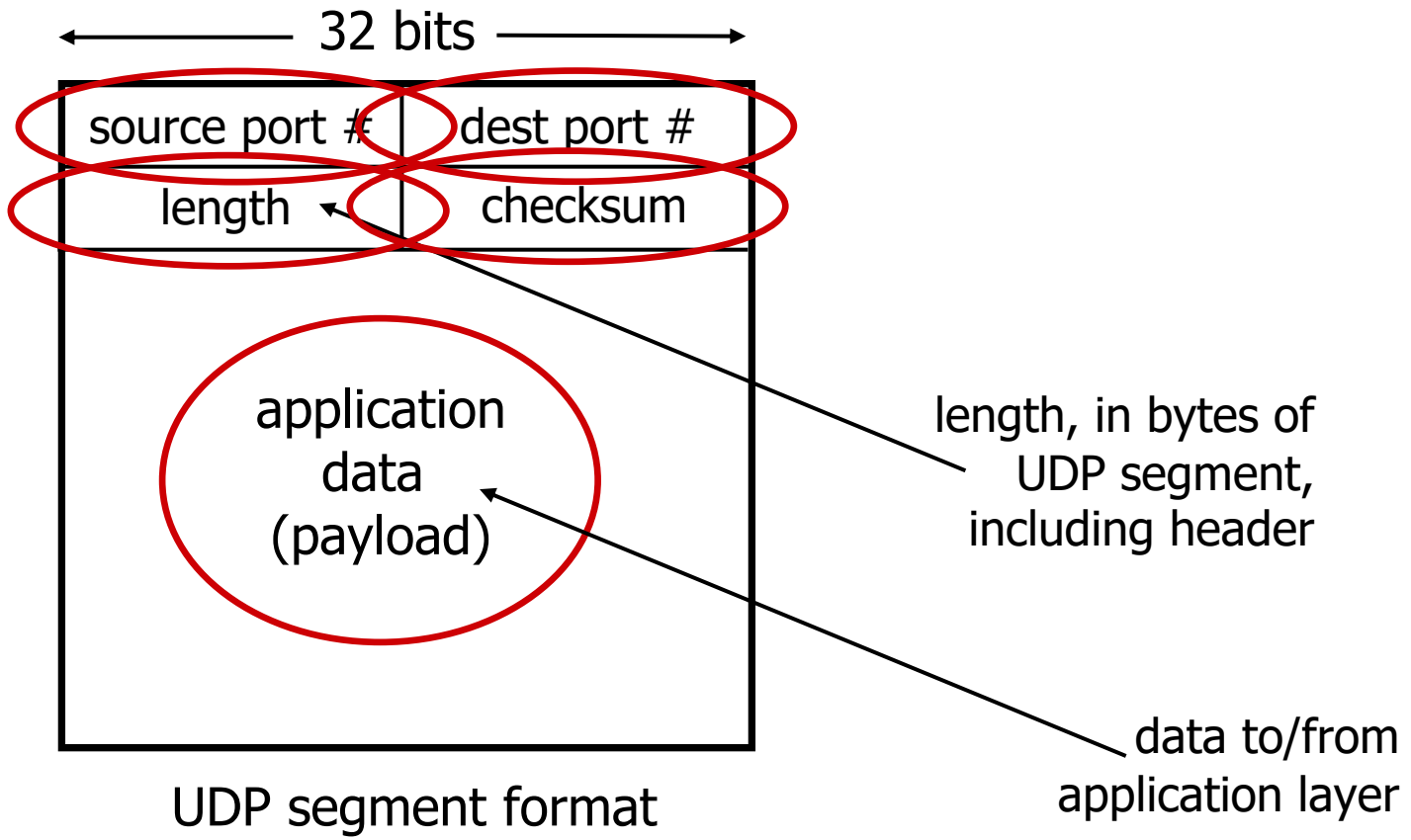
UDP: Transport Layer Actions



UDP receiver actions:

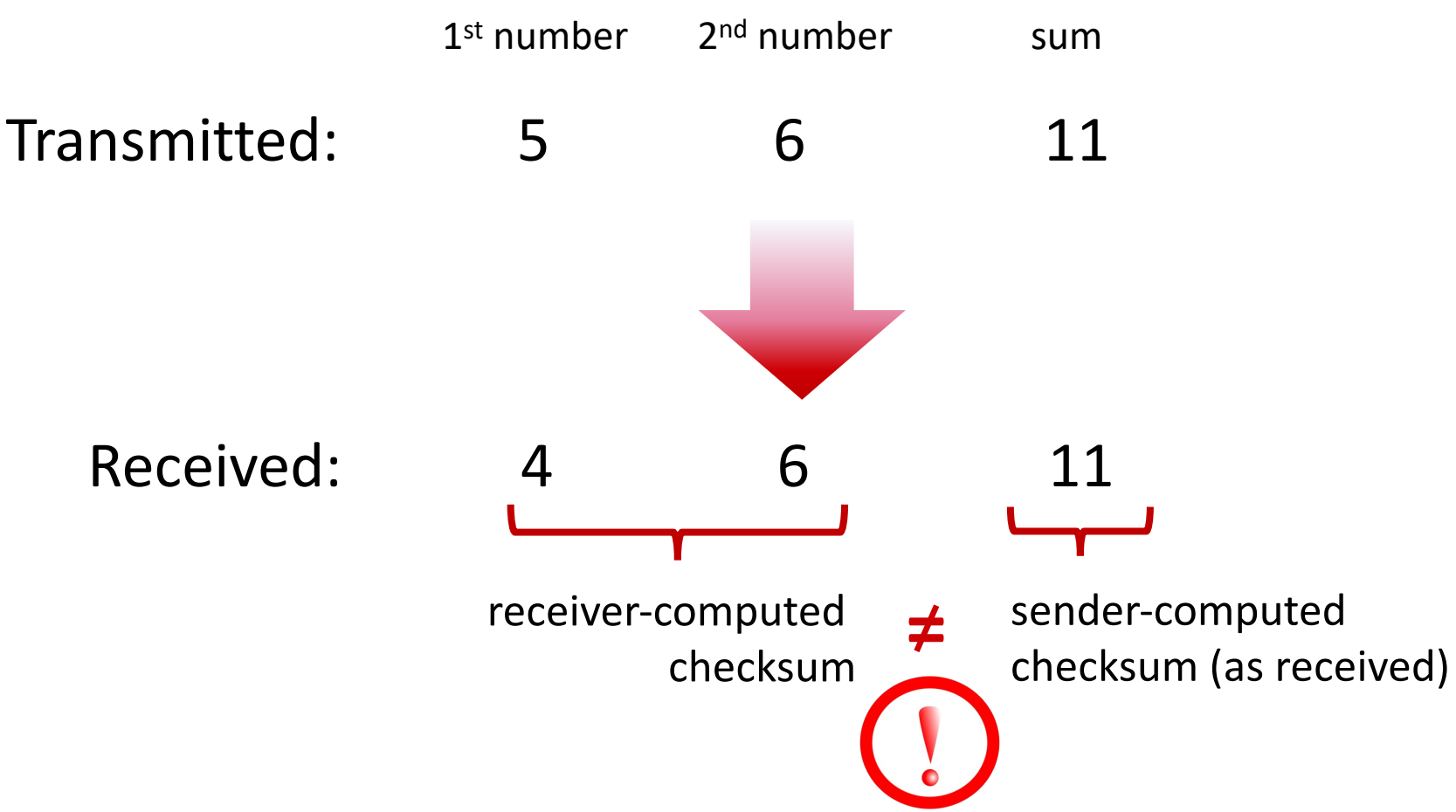
- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

UDP segment header



UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment



UDP checksum

Goal: detect errors (*i.e.,* flipped bits) in transmitted segment

sender:

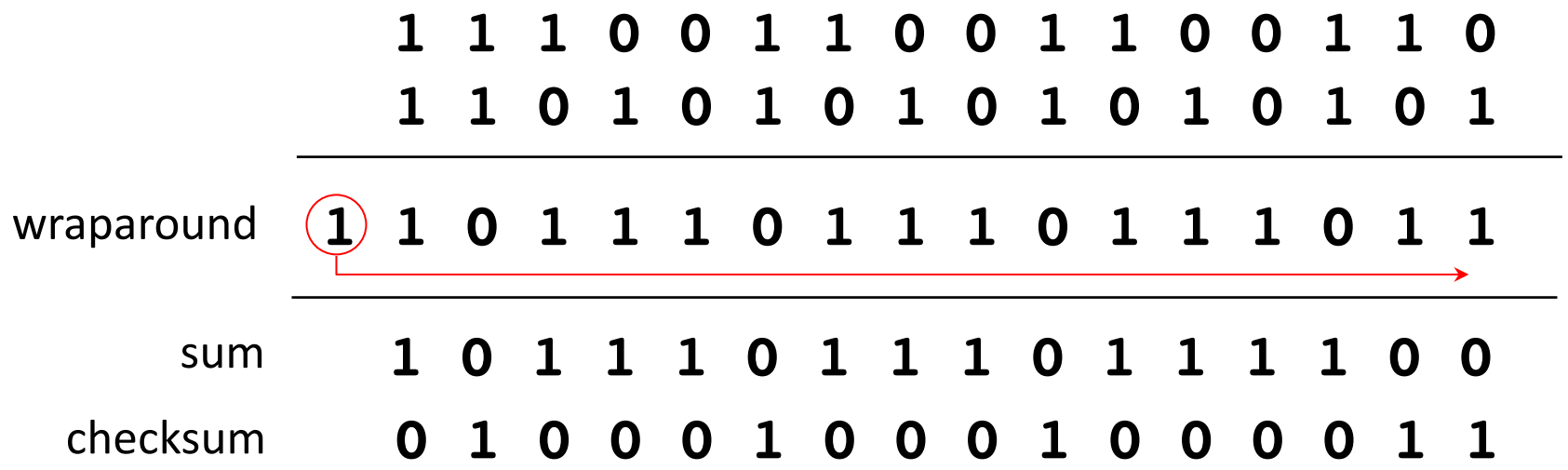
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an example

example: add two 16-bit integers

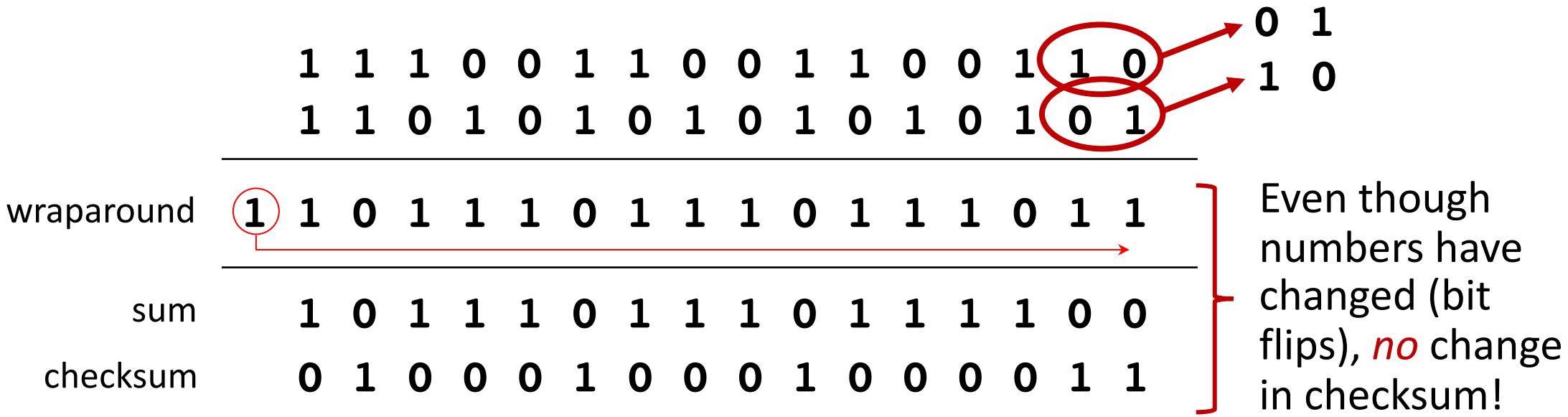


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

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

Internet checksum: weak protection!

example: add two 16-bit integers



Summary: UDP

- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Chapter 3: roadmap

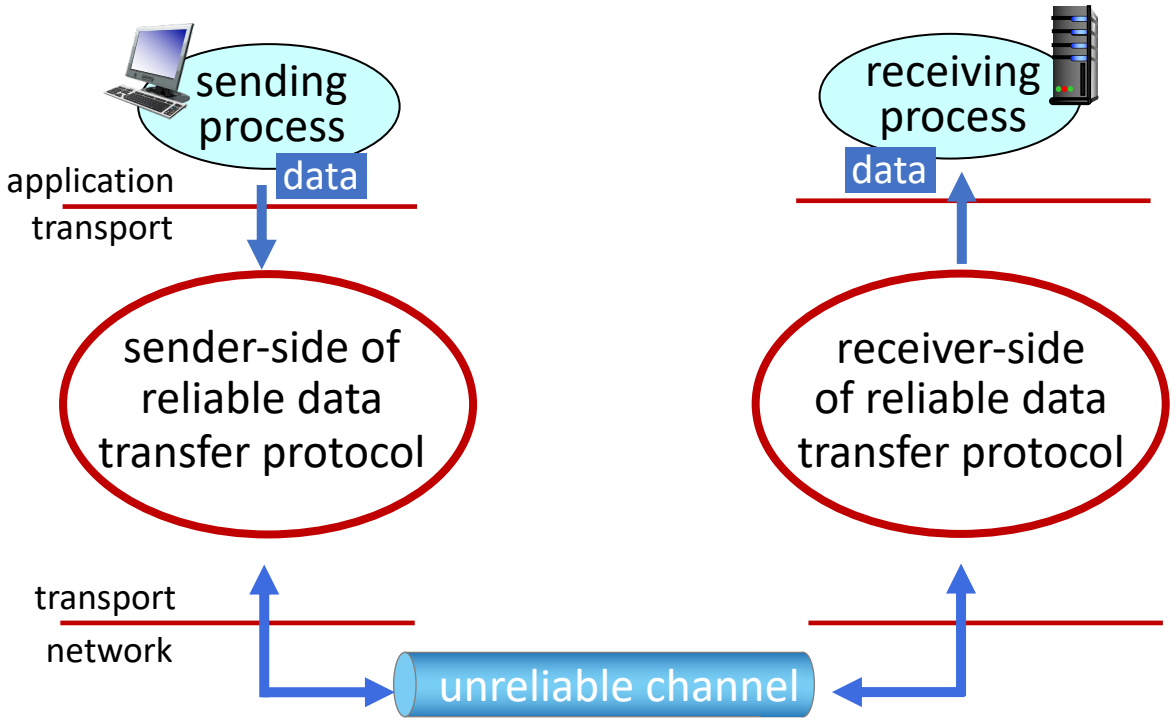
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

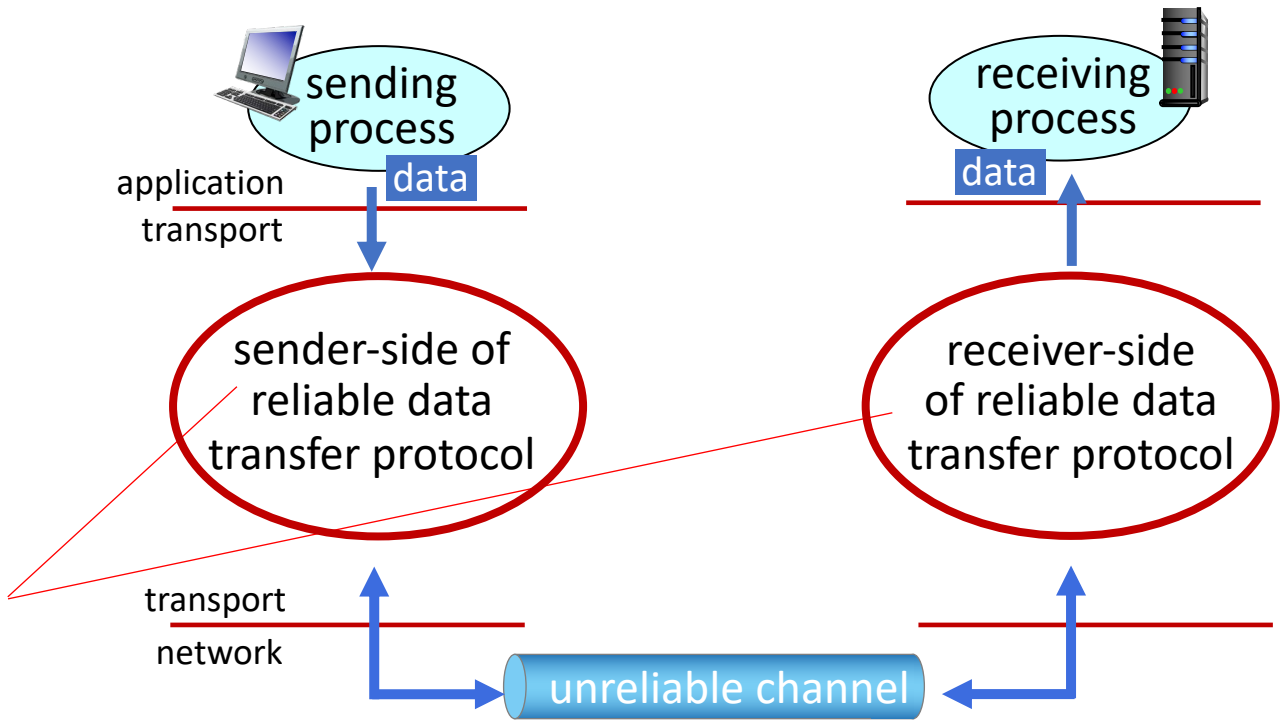
reliable service *abstraction*



reliable service *implementation*

Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

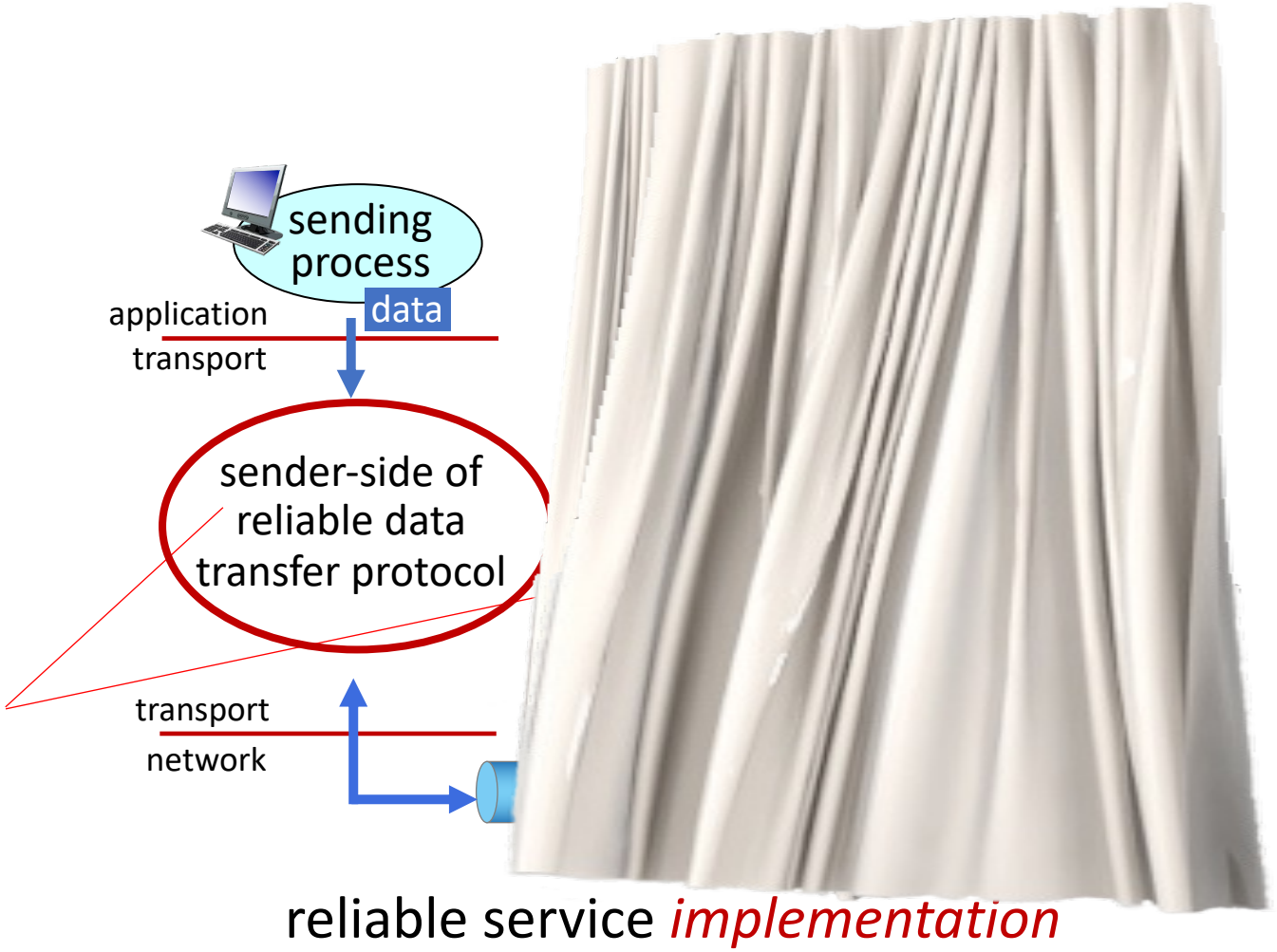


reliable service *implementation*

Principles of reliable data transfer

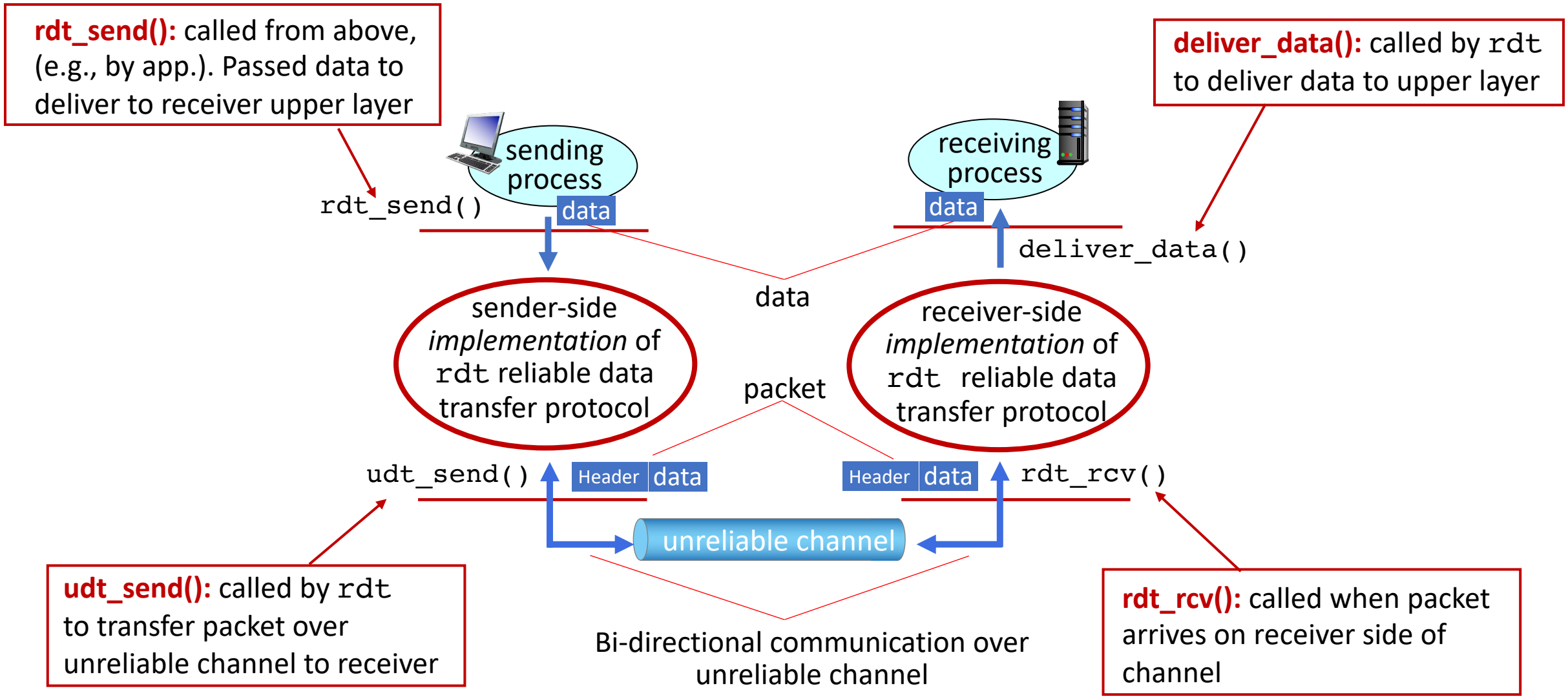
Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message



reliable service *implementation*

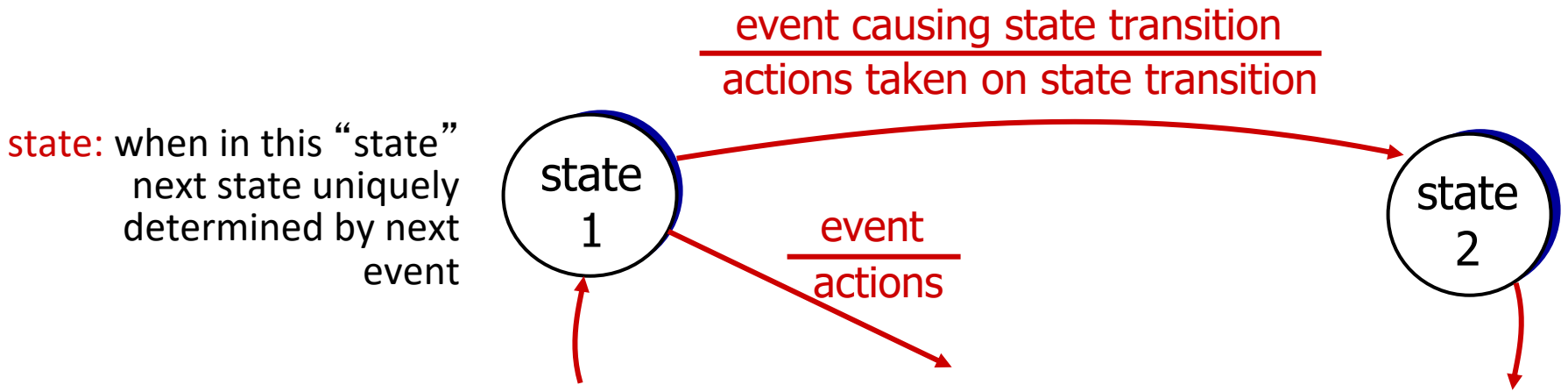
Reliable data transfer protocol (rdt): interfaces



Reliable data transfer: getting started

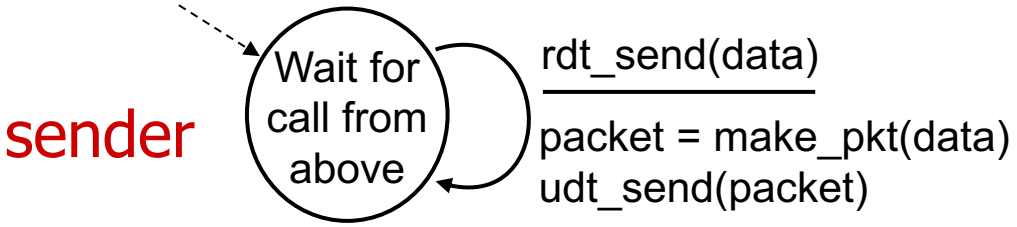
We will:

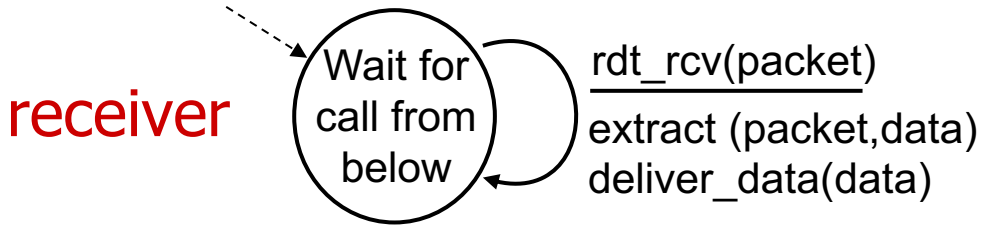
- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



rdt1.0: reliable transfer over a reliable channel

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





rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

How do humans recover from "errors" during conversation?

rtd2.0 – three additional capabilities

1. *Error detection*
2. *Receiver feedback*
3. *Retransmission*

In computer networks, reliable data transfer protocols based on such retransmission are known as

ARQ (Automatic Repeat reQuest) protocols

rdt2.0: channel with bit errors

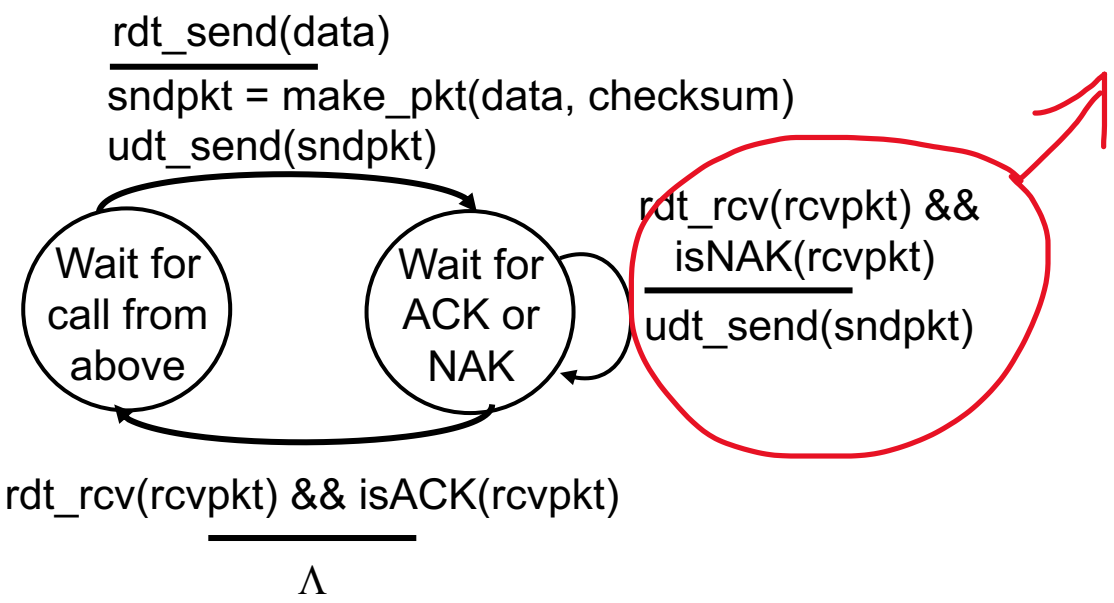
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors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

stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications

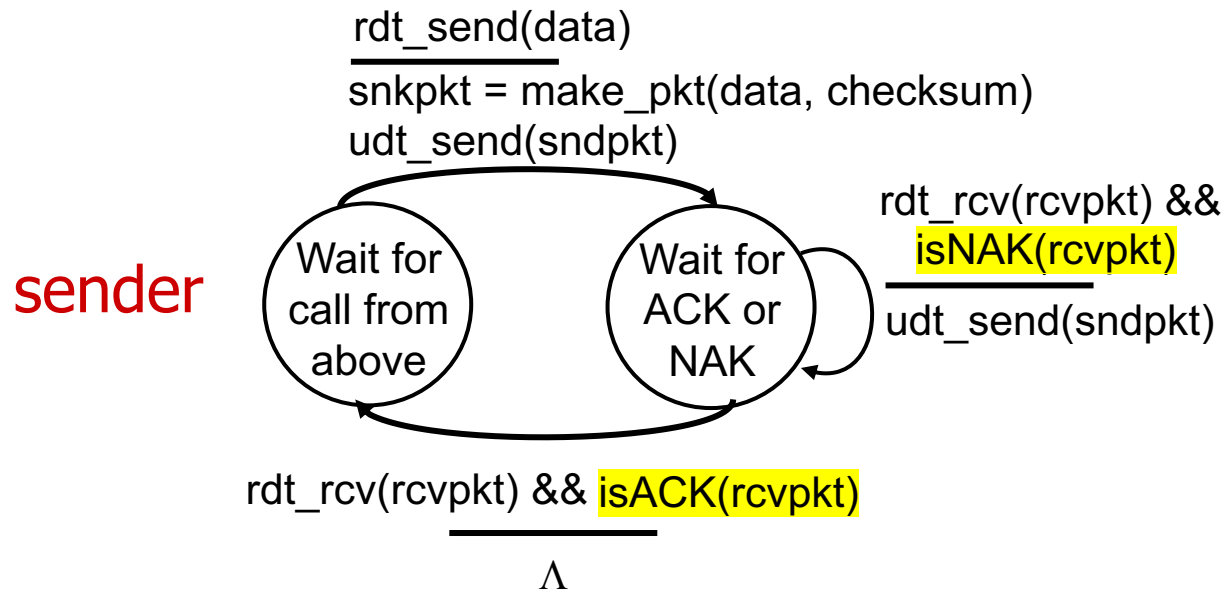
sender



While in "Wait for ACK or NACK" the sender cannot receive any data from the upper layer

→ STOP and WAIT behavior

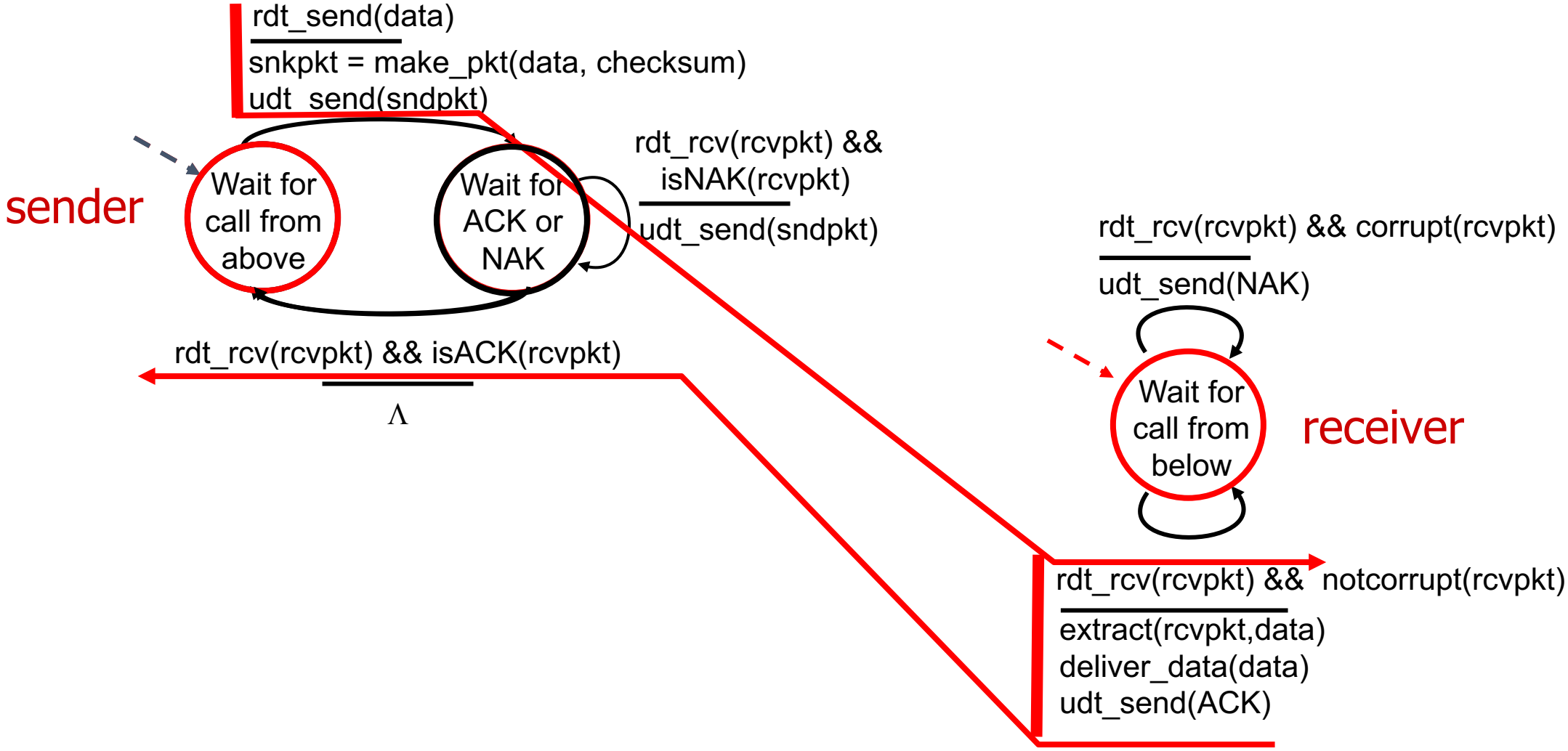
rdt2.0: FSM specification



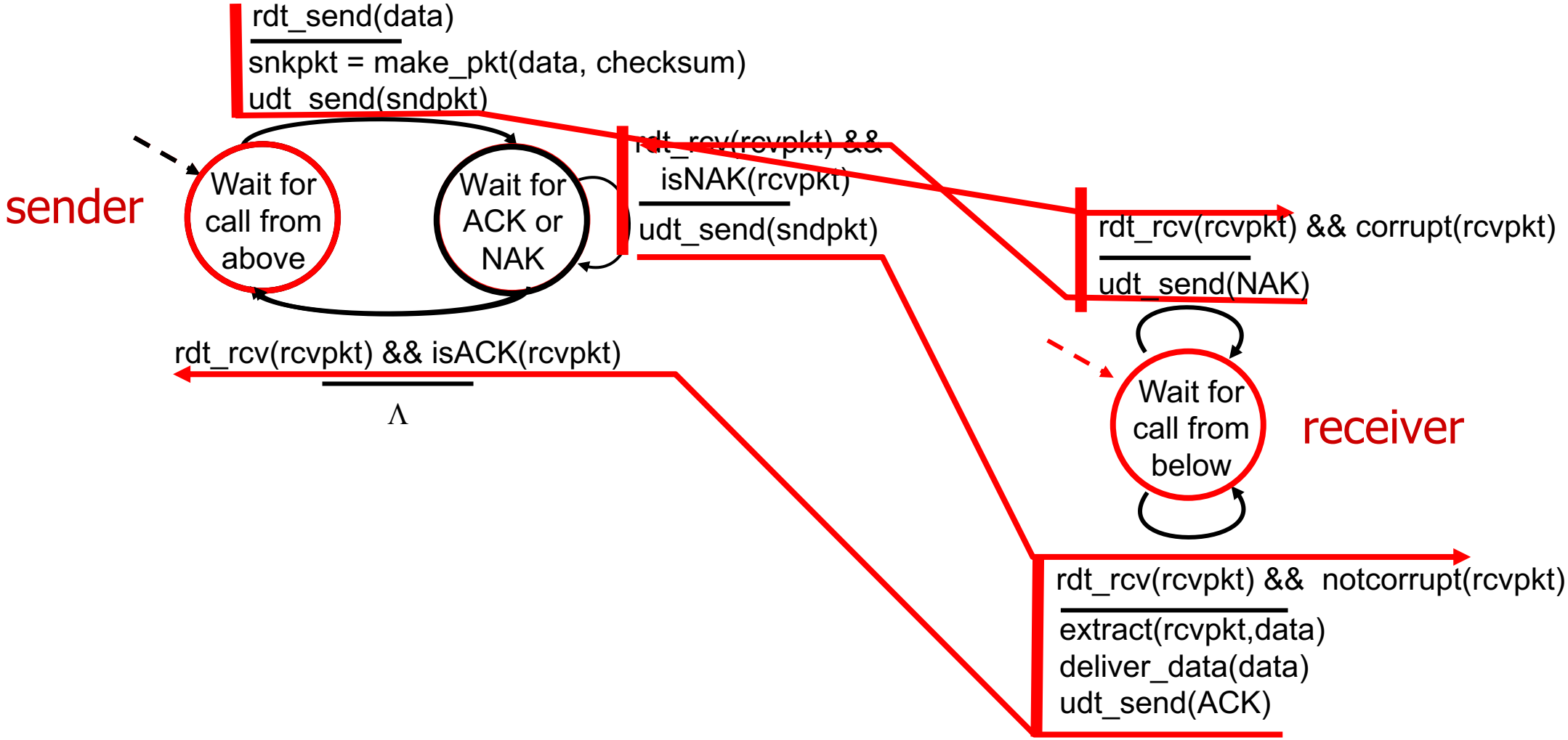
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



rdt2.0 has a fatal flaw!

ACKs and NACKs can be corrupted too!

what happens if ACK/NAK corrupted?

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

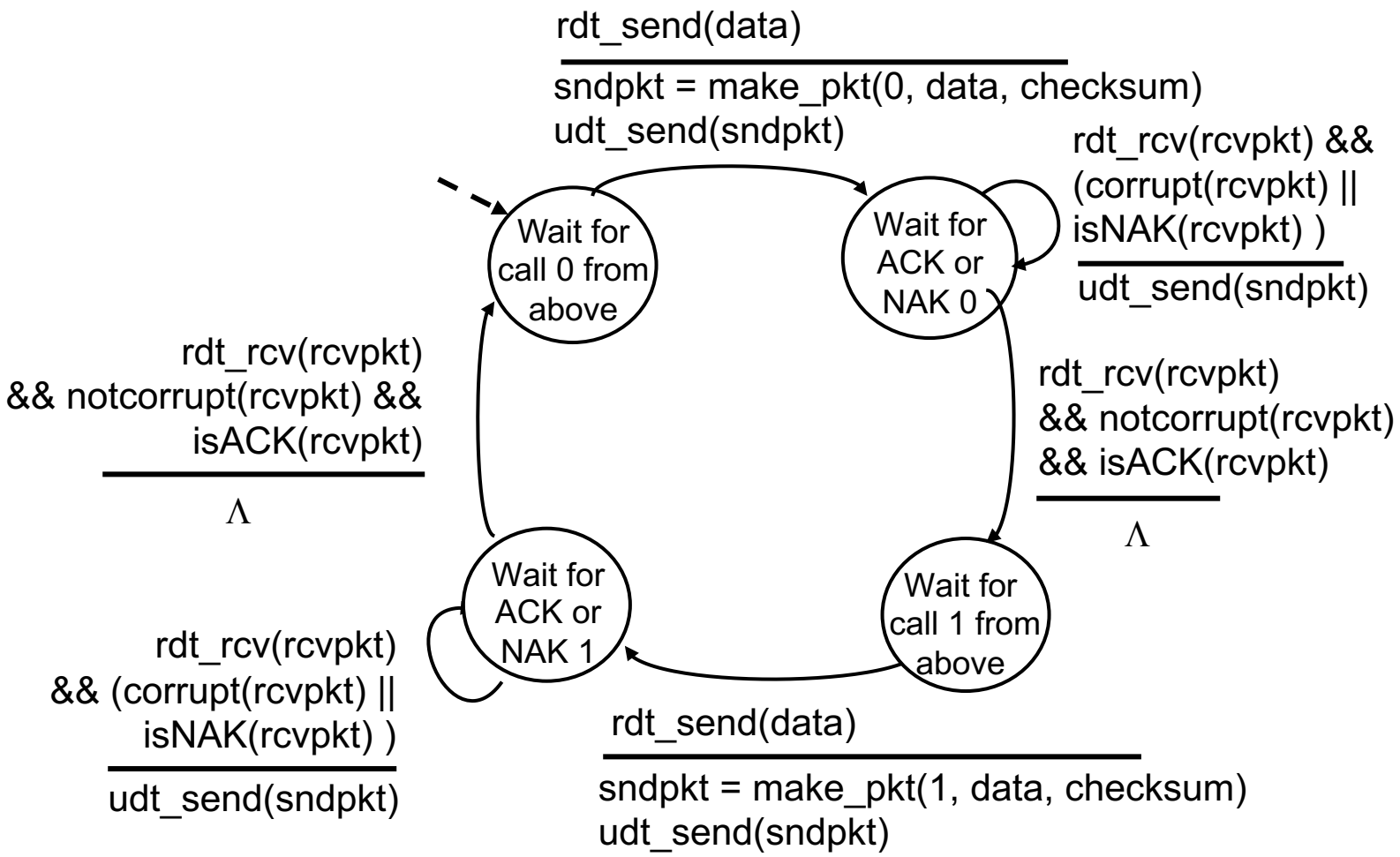
handling duplicates:

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

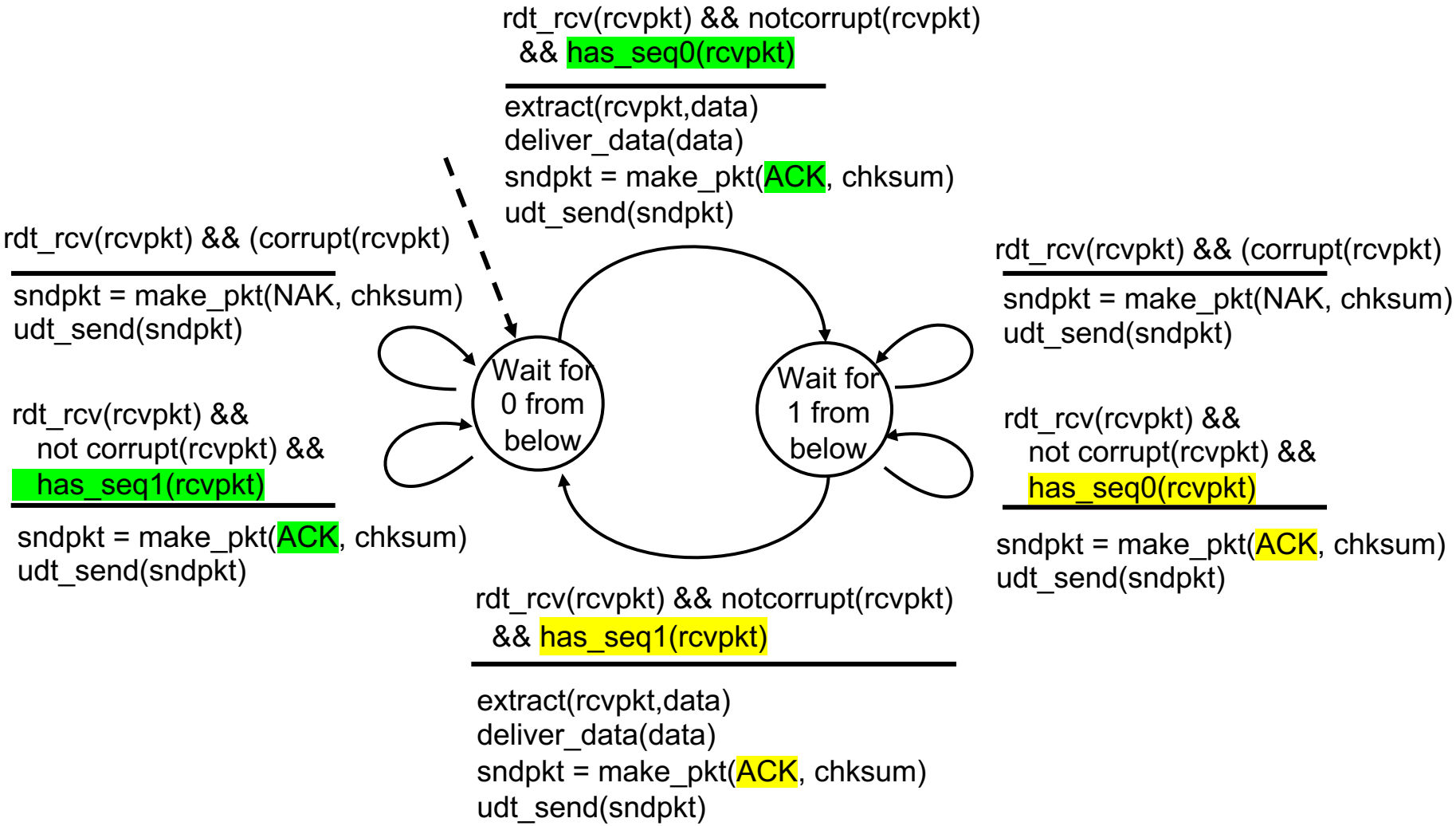
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling 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 was received correctly by the sender

rtd2.1: discussion

- Do we need to associate a sequence number to ACK and NACK packets?

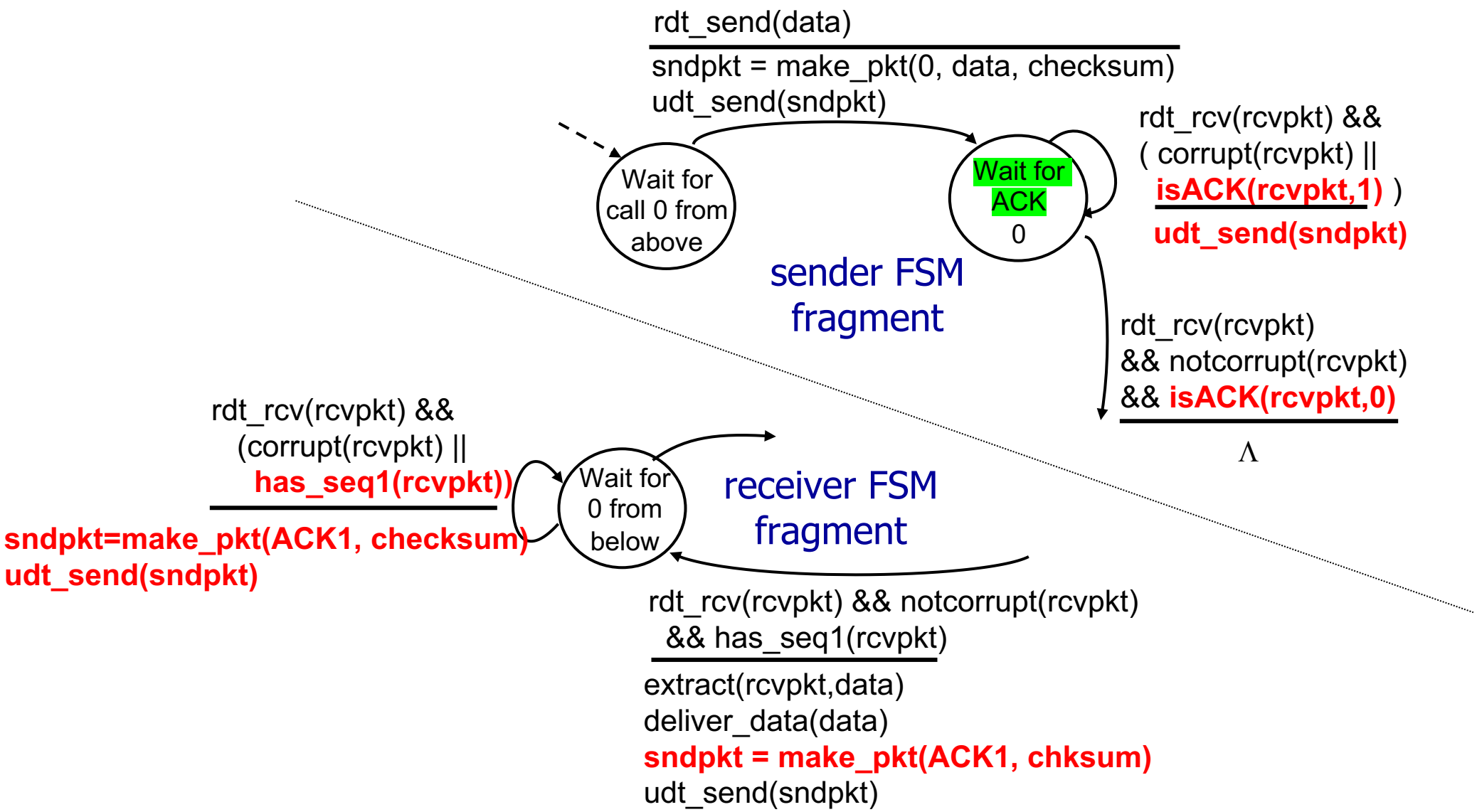
NO! In this STOP & WAIT implementation, they always refer to the most recently transmitted packet

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*

As we will see, TCP uses this approach to be NAK-free

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

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

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?

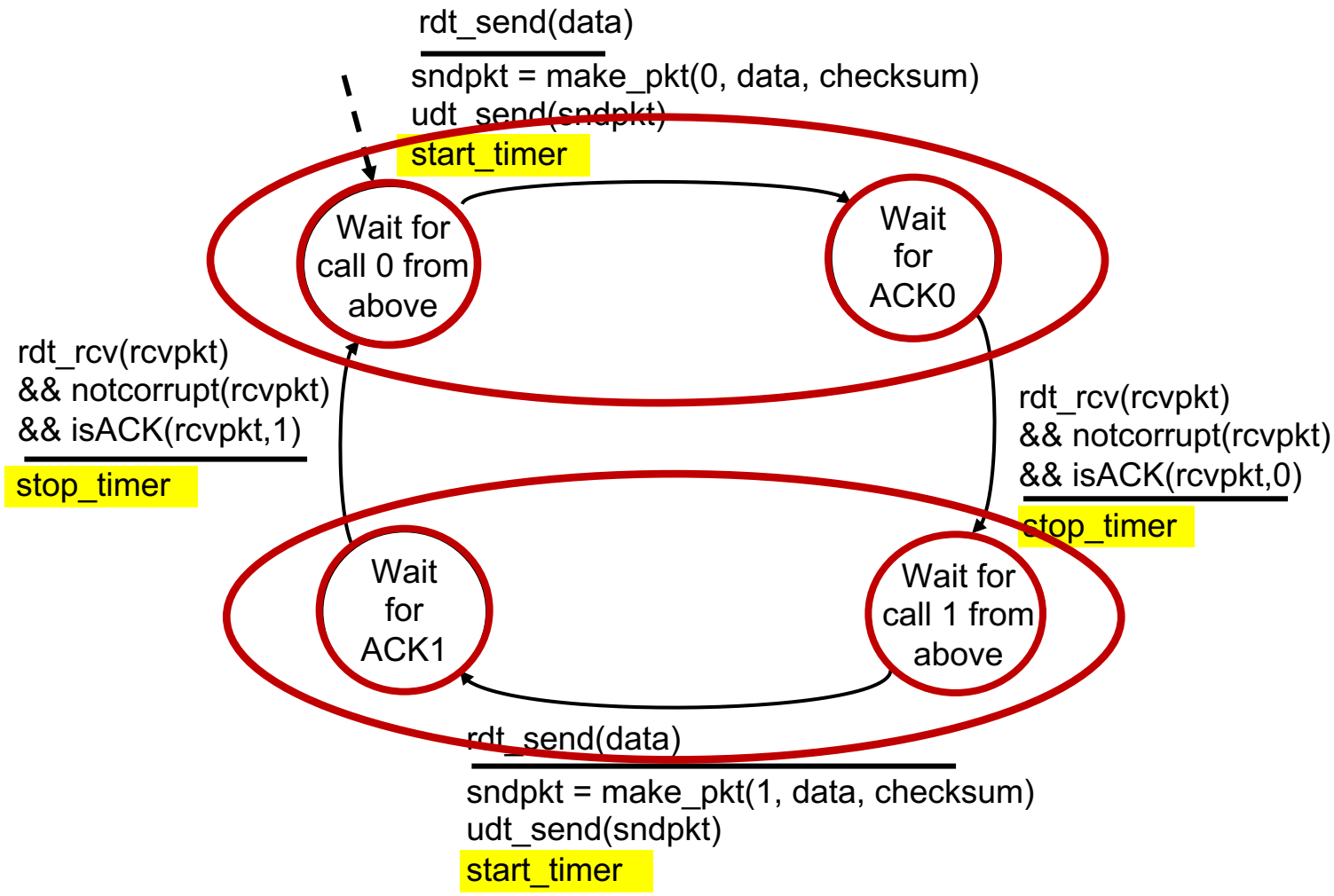
rdt3.0: channels with errors *and* loss

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

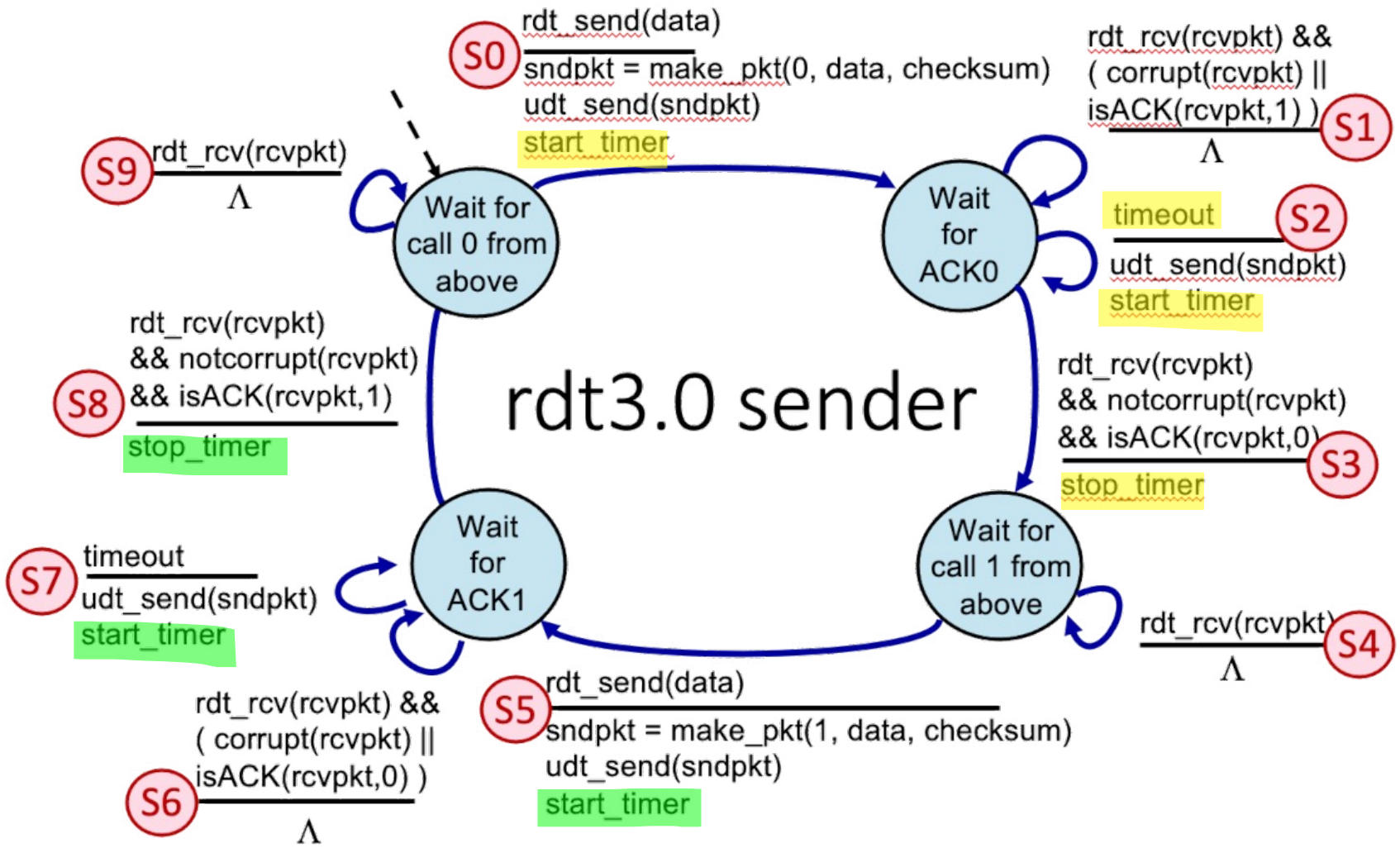
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

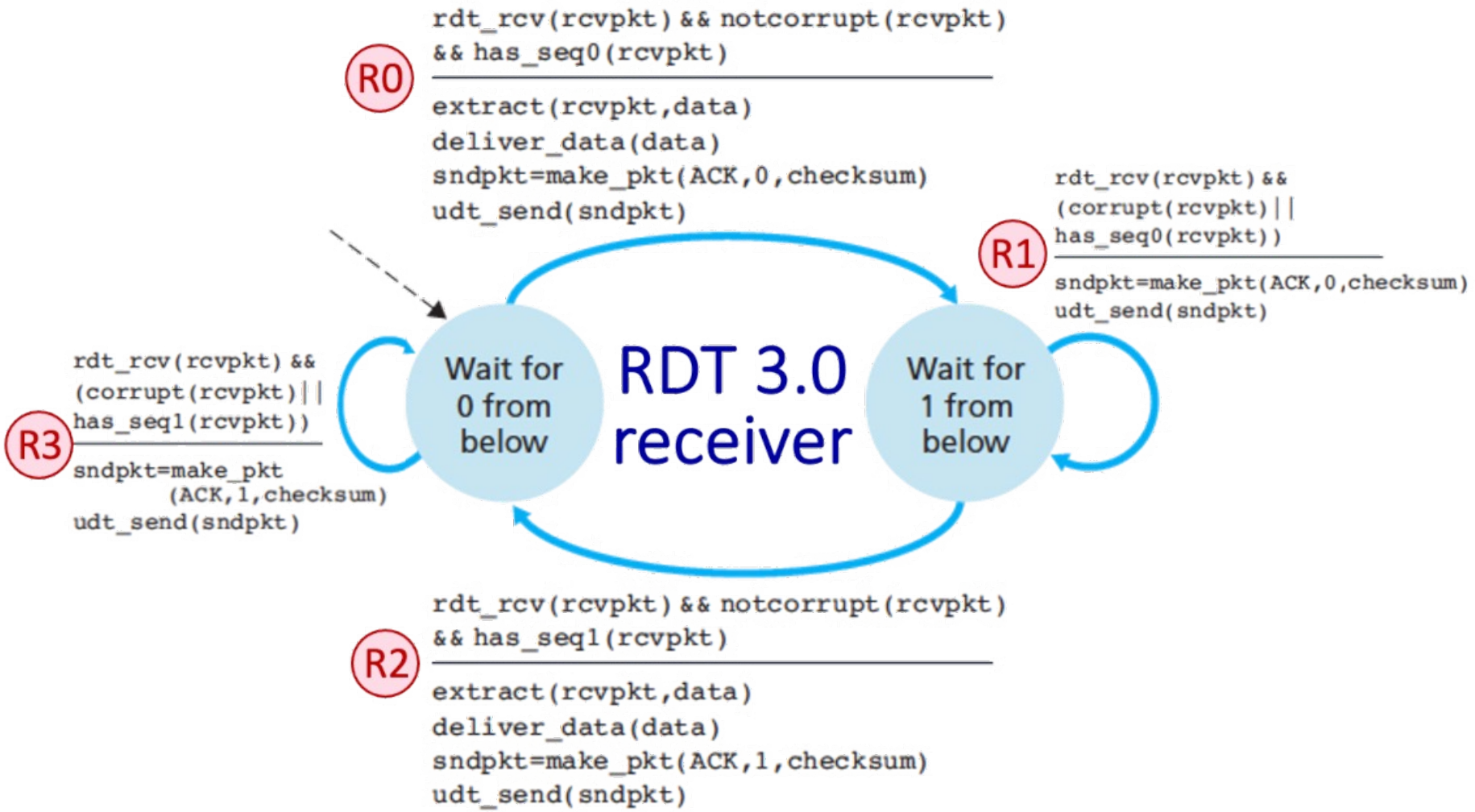
rdt3.0 sender



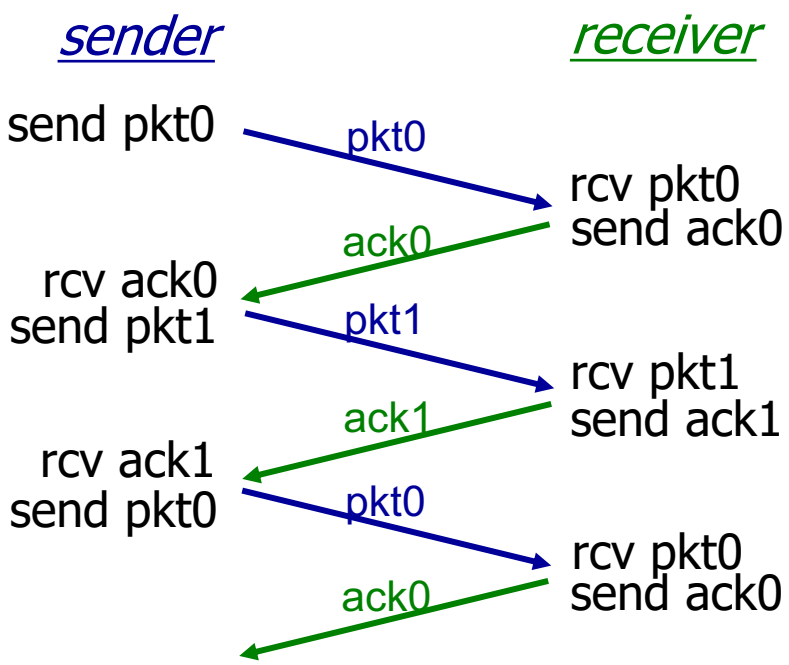
rtd3.0 sender



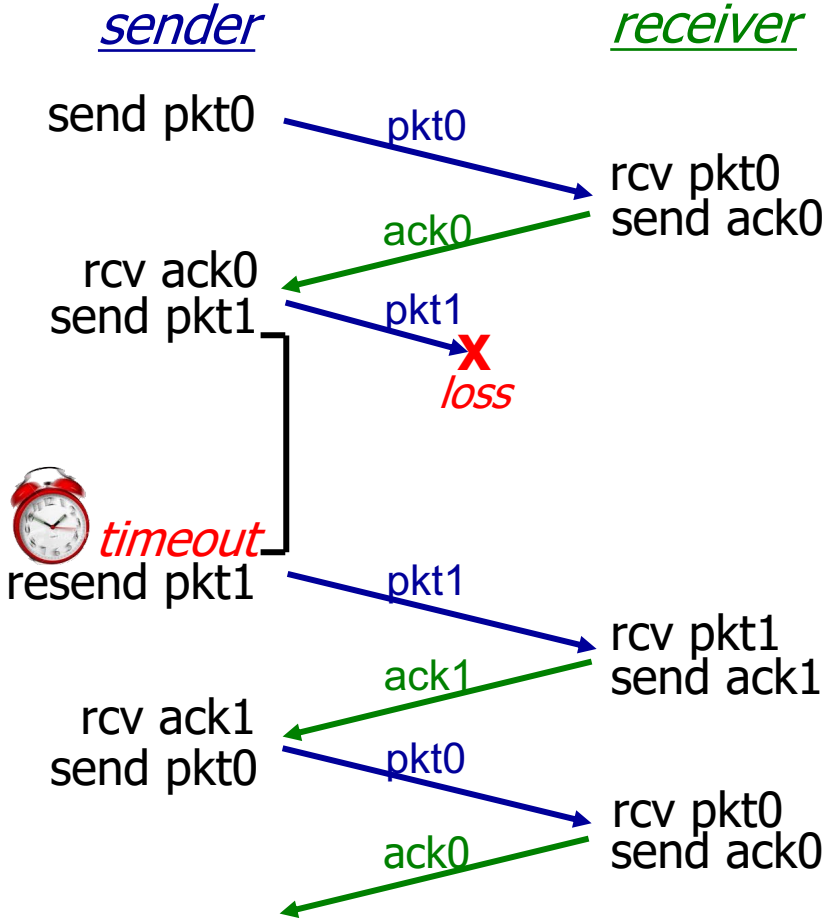
rtd3.0 receiver



rdt3.0 in action

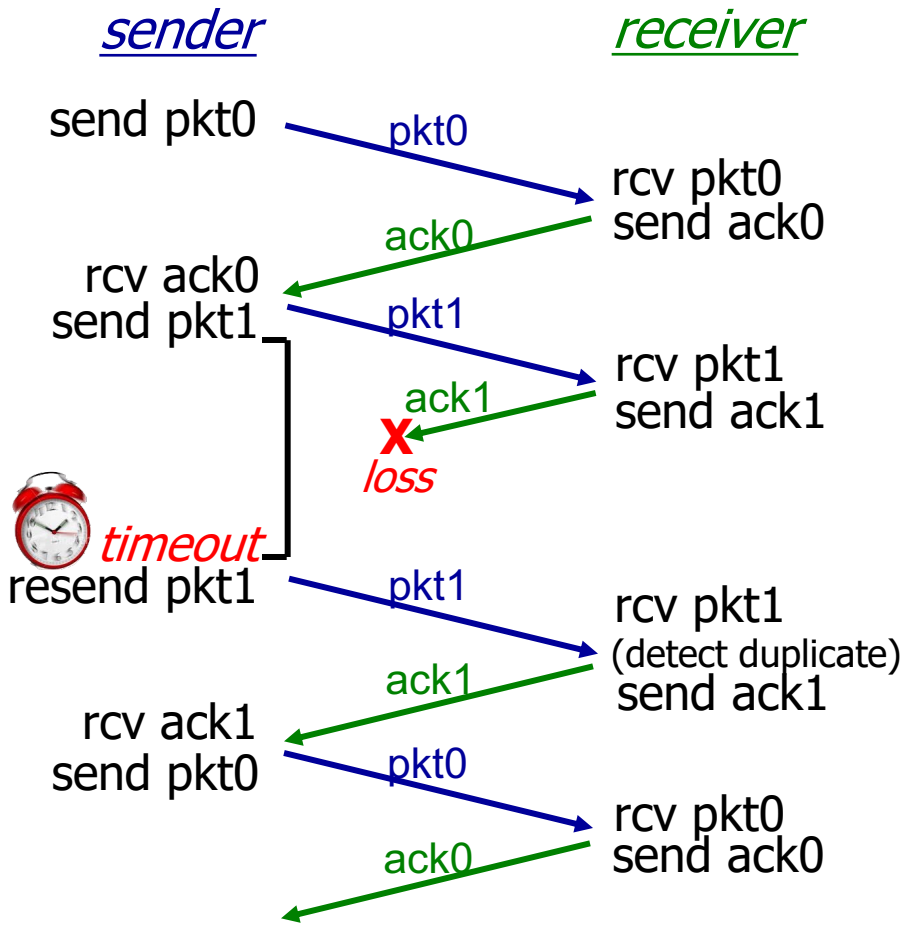


(a) no loss

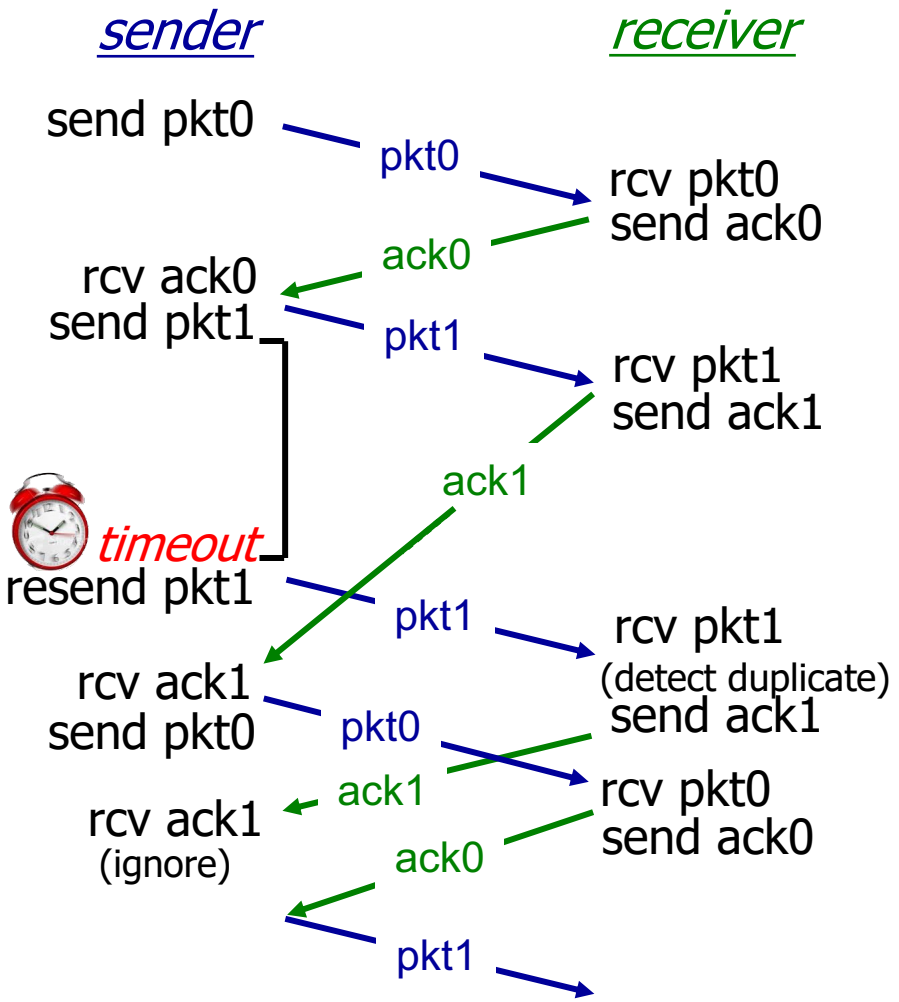


(b) packet loss

rdt3.0 in action

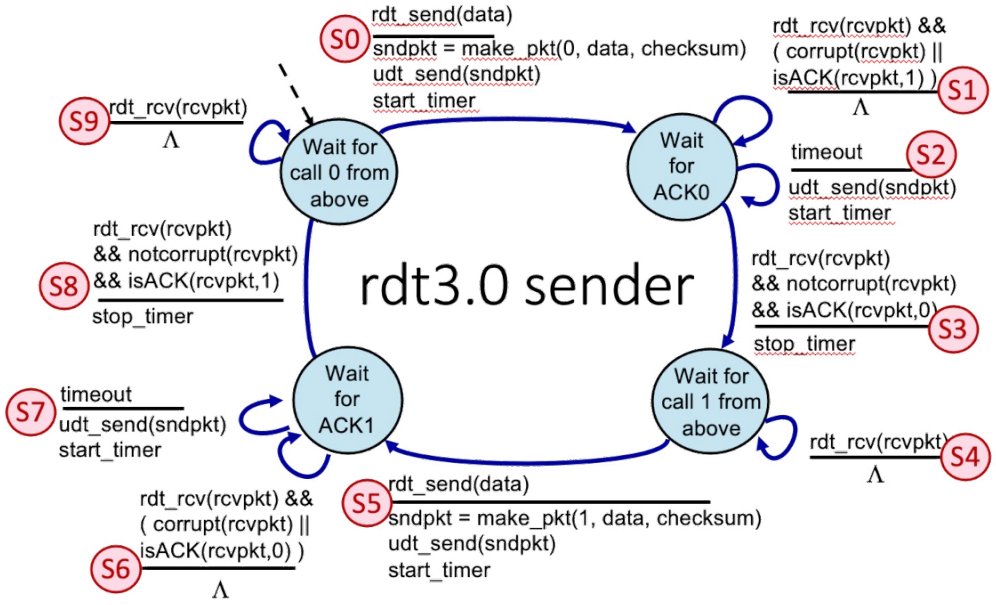


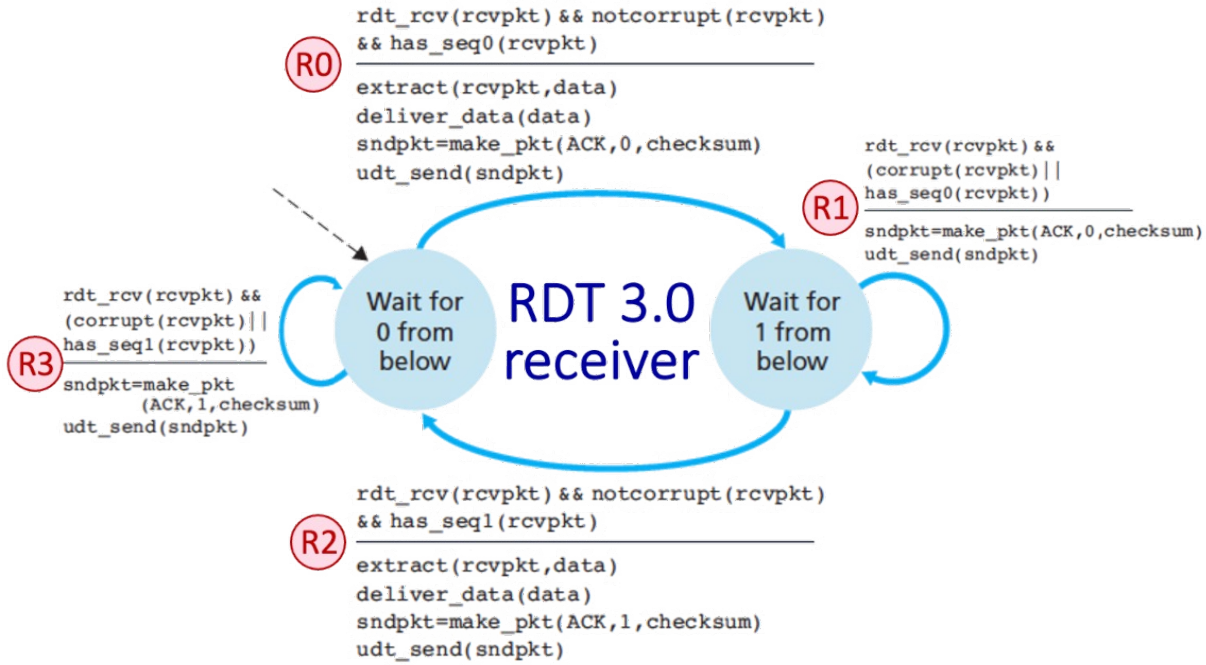
(c) ACK loss



(d) premature timeout/ delayed ACK

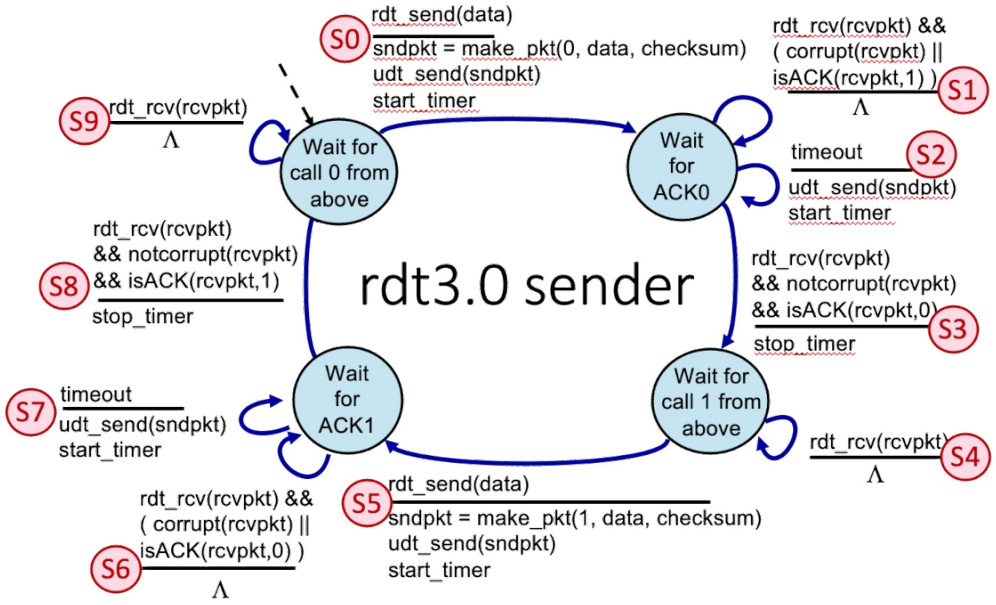
Exercise

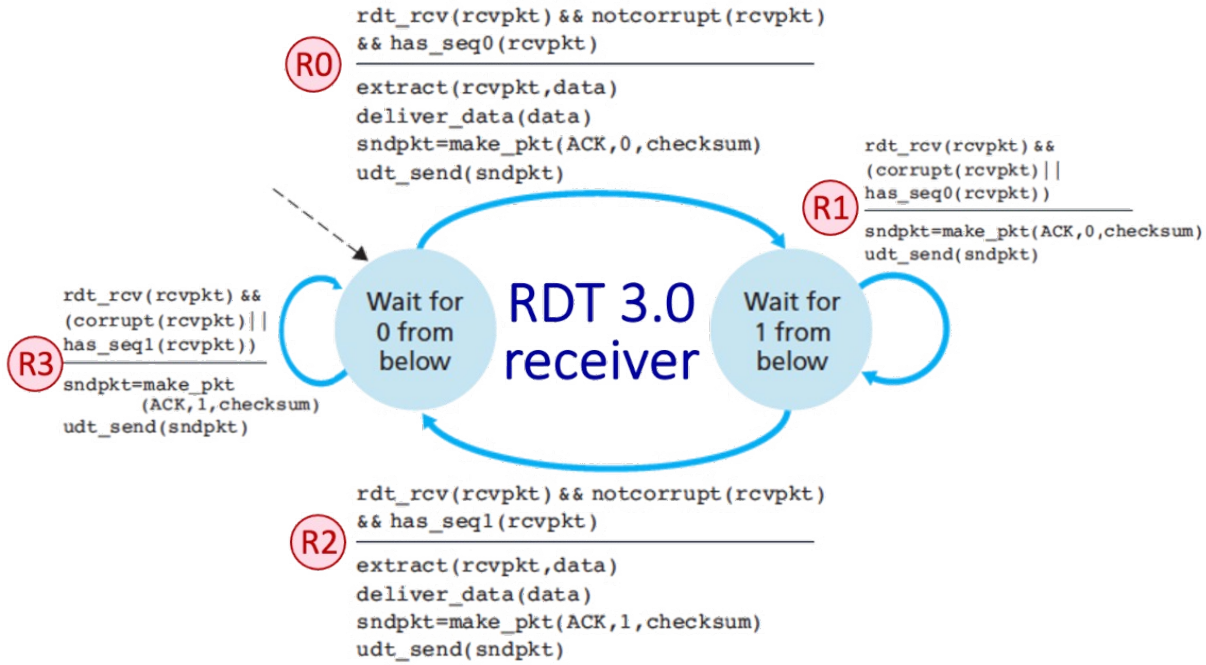




Given the transition sequence: S0, *, S3, S5, R2
What is the missing transition?

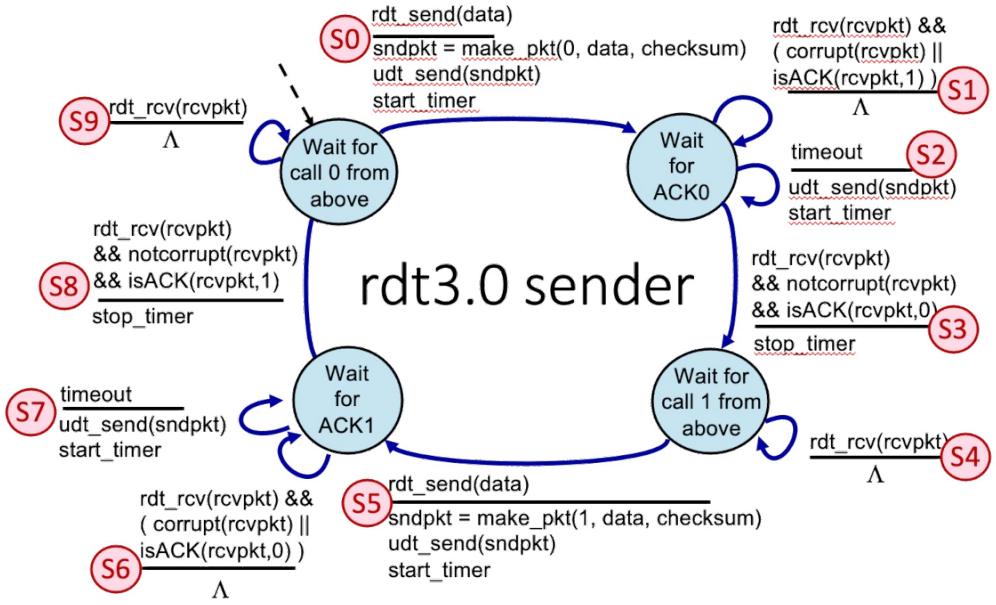
Exercise

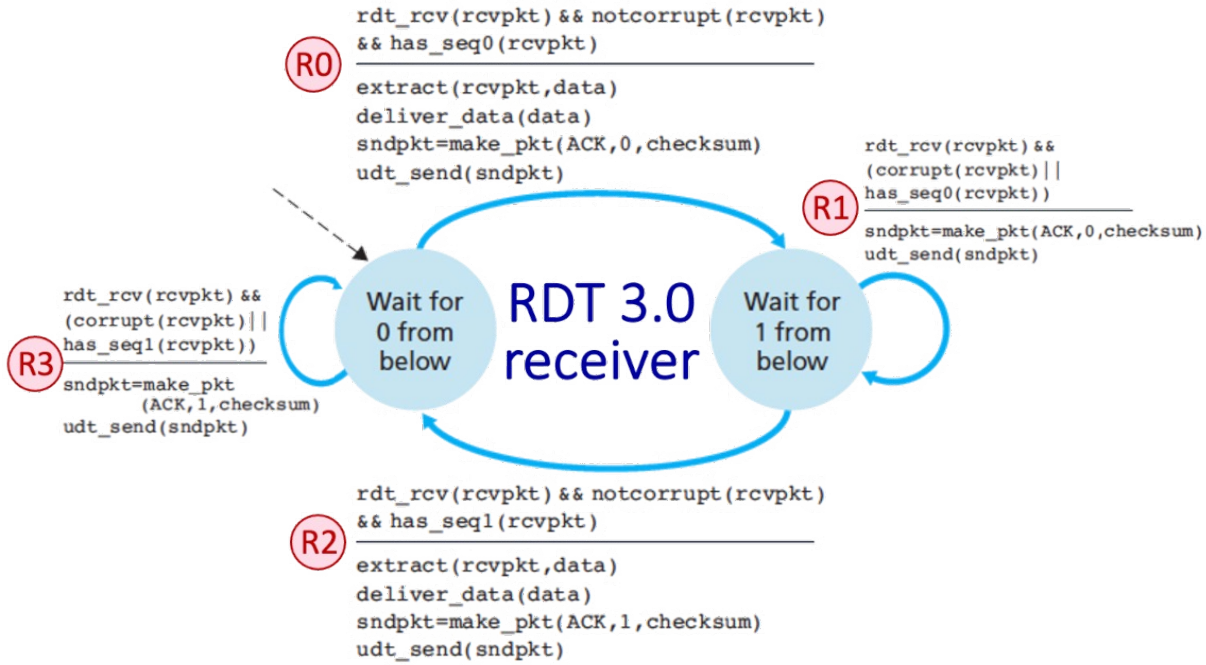




Given the transition sequence: S0, *, S3, S5, R2
What is the missing transition?
R0

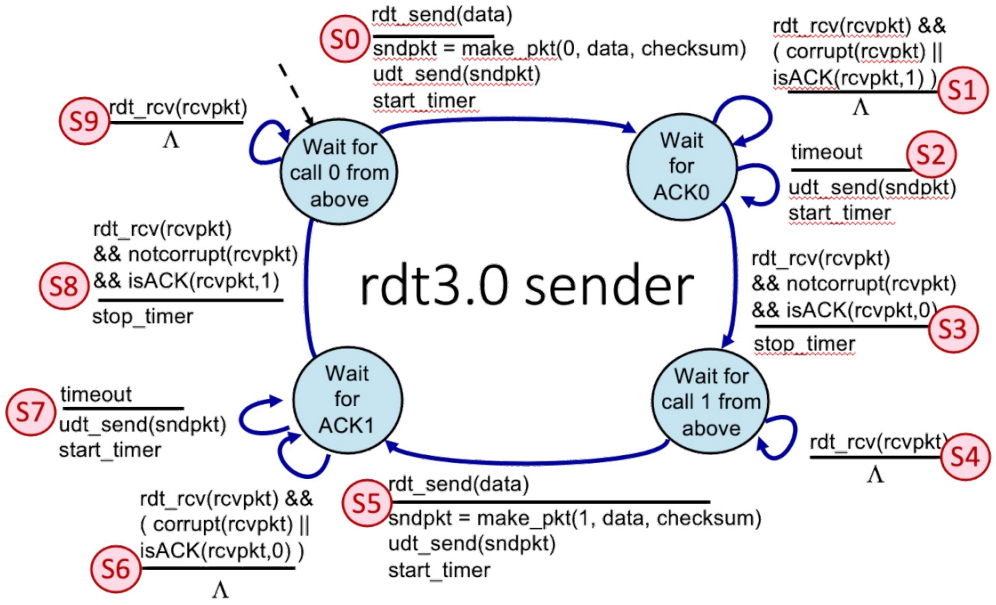
Exercise

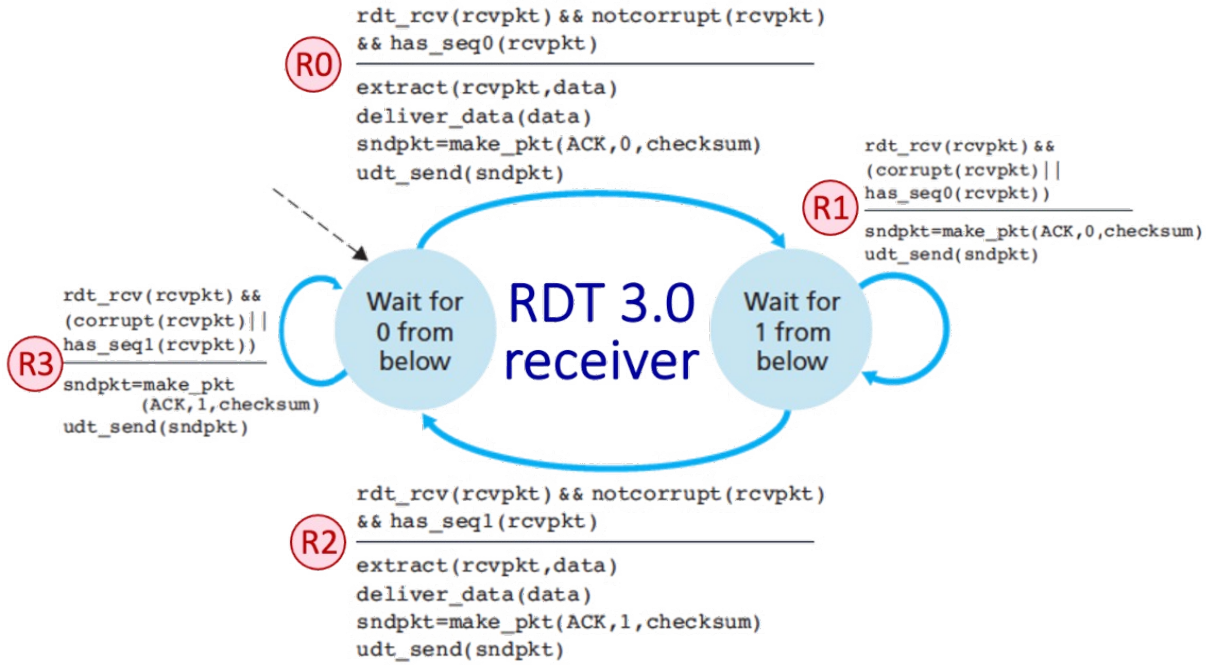




Given the transition sequence: S0, R0, S1, S2, R1, S1, *, R1, S1, S2, R1, S1, S2, R1, S3, S5, R2, S8
What is the missing transition?

Exercise



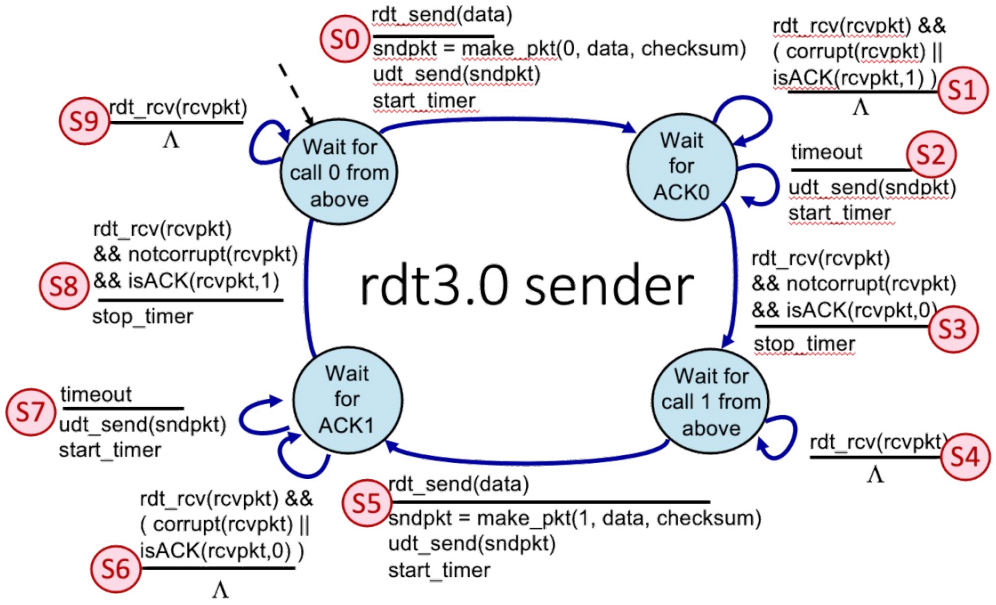


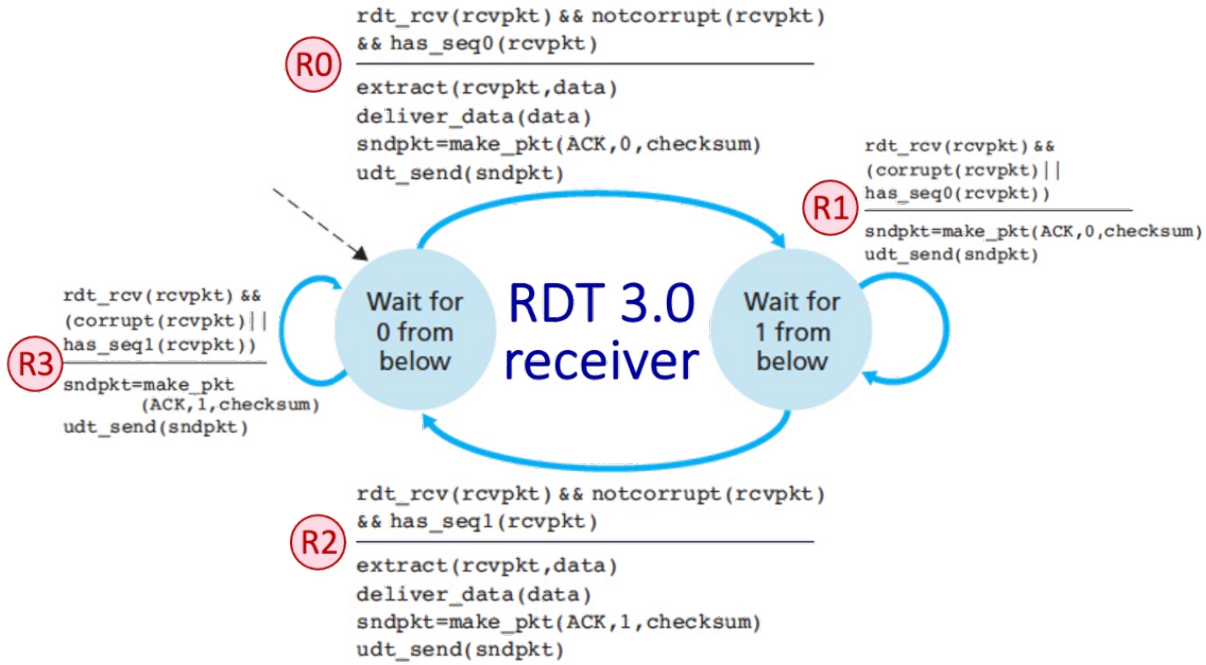
Given the transition sequence: S0, R0, S1, S2, R1, S1, *, R1, S1, S2, R1, S1, S2, R1, S3, S5, R2, S8

What is the missing transition?

S2

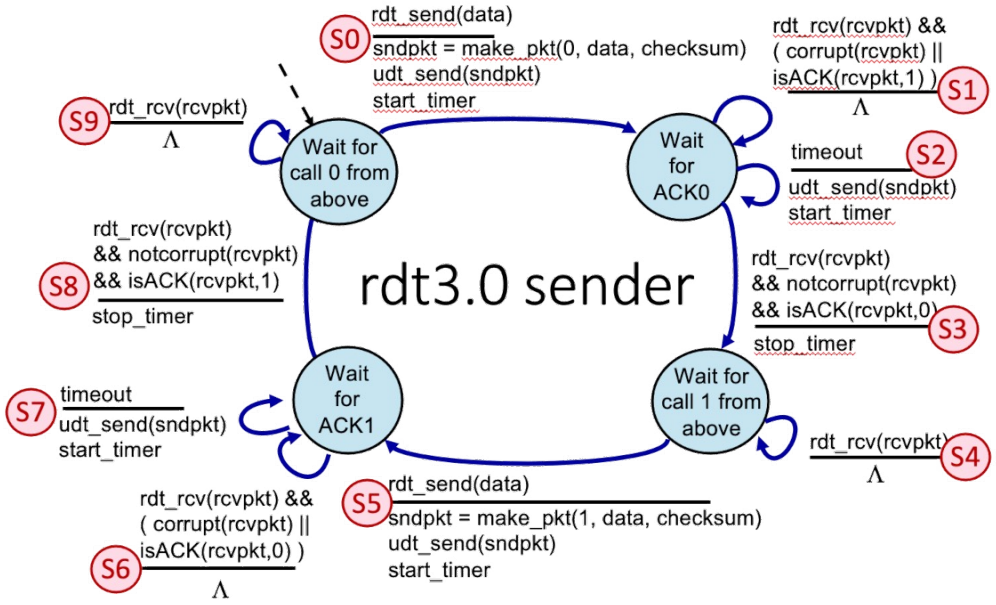
Exercise

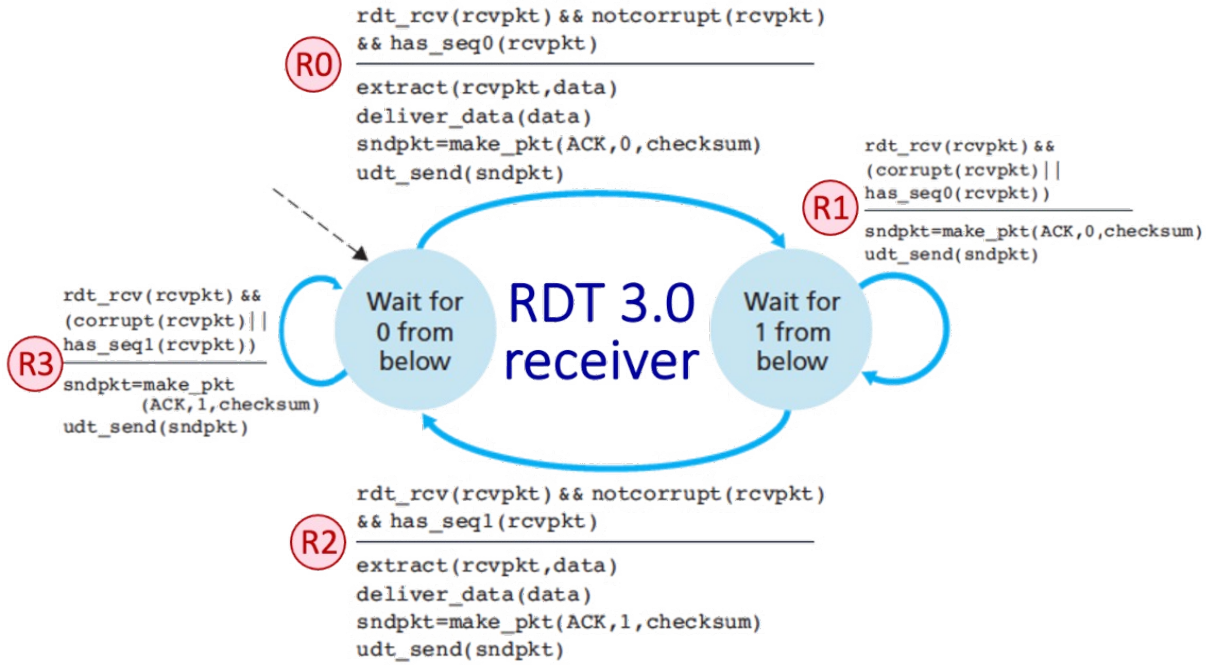




Given the transition sequence: S0, R0, S3, S5, R2, S7, *, S8,
What is the missing transition?

Exercise

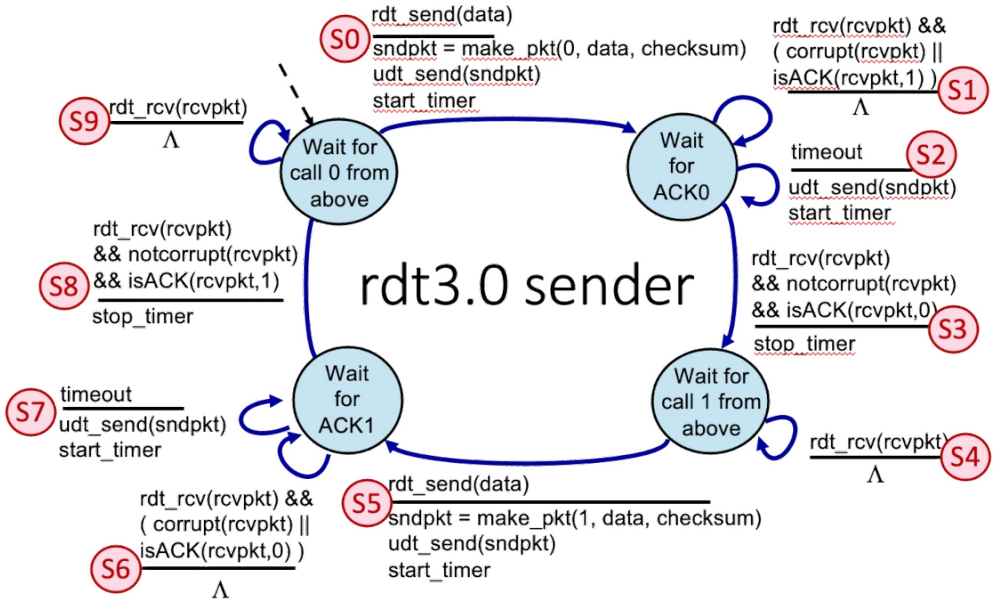


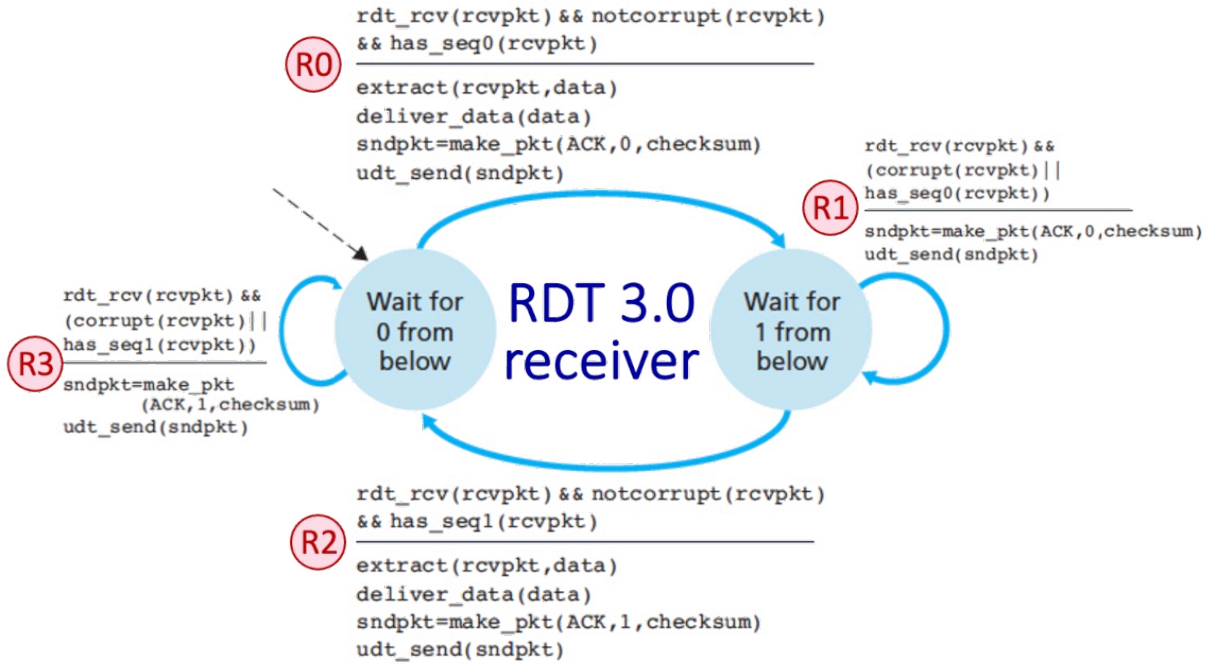


Given the transition sequence: S0, R0, S3, S5, R2, S7, *, S8,
What is the missing transition?

R3

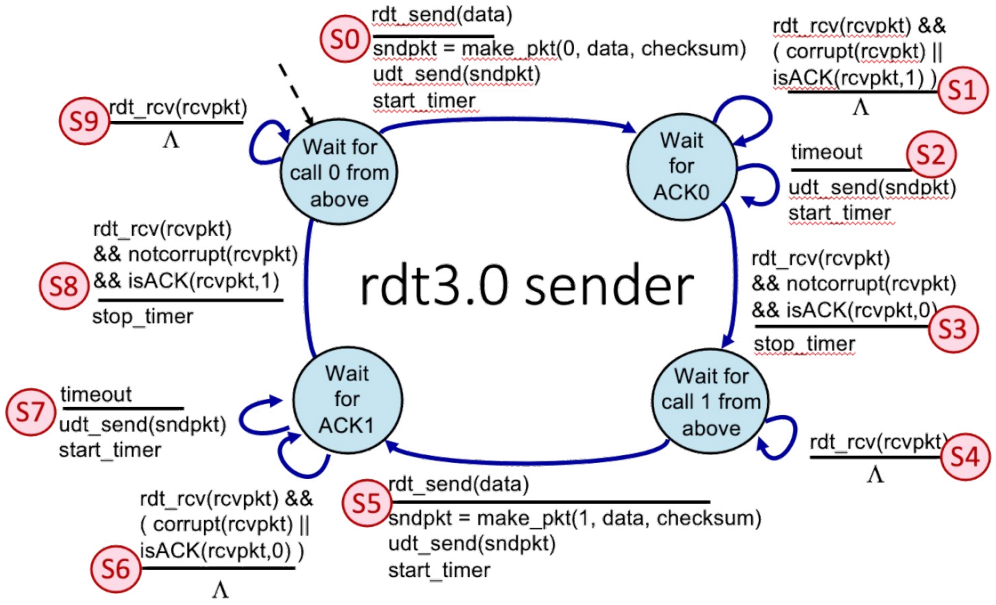
Exercise

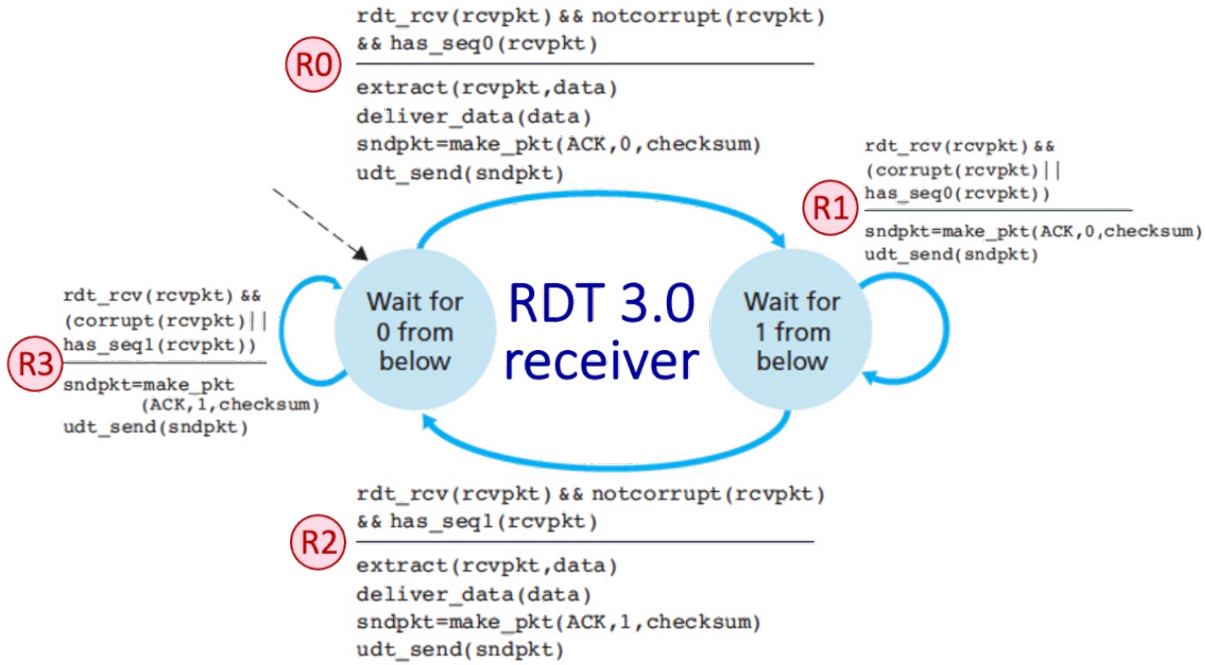




Given the transition sequence: S0, R0, S2, R1, *, S5, S7, R2, S8
What is the missing transition?

Exercise





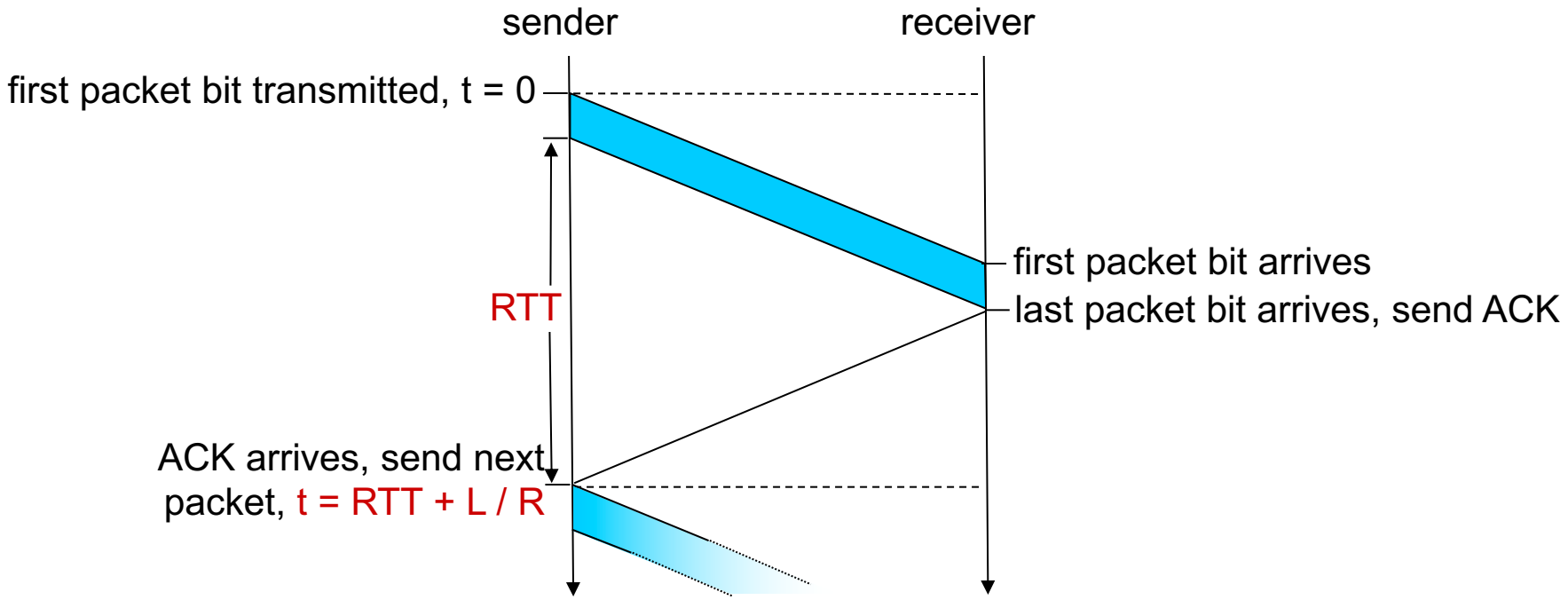
Given the transition sequence: S0, R0, S2, R1, *, S5, S7, R2, S8
What is the missing transition?
S3

Performance of rdt3.0 (stop-and-wait)

- U_{sender}: *utilization* – fraction of time the sender is busy sending
- RTT of the link is 30 msec
- 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

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

rdt3.0: stop-and-wait operation

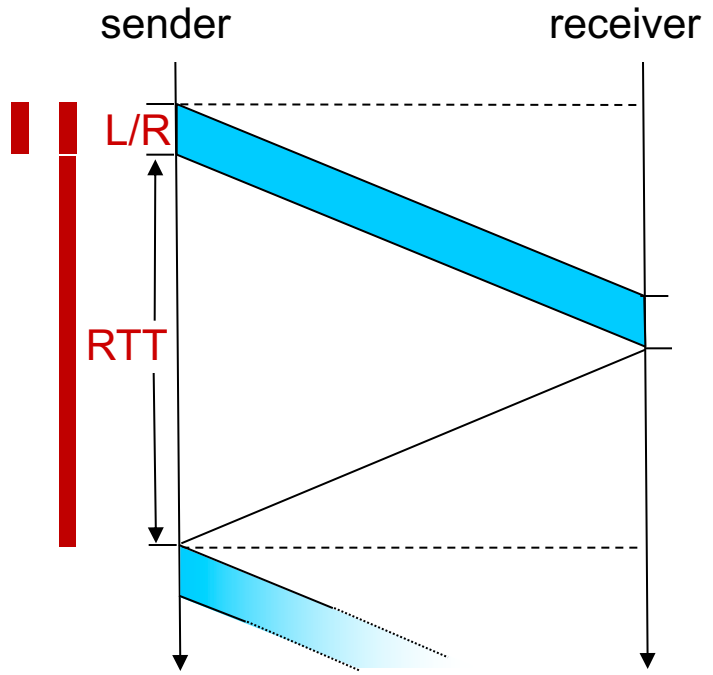


rdt3.0: stop-and-wait operation

$$U_{sender} = \frac{L / R}{RTT + L / R}$$

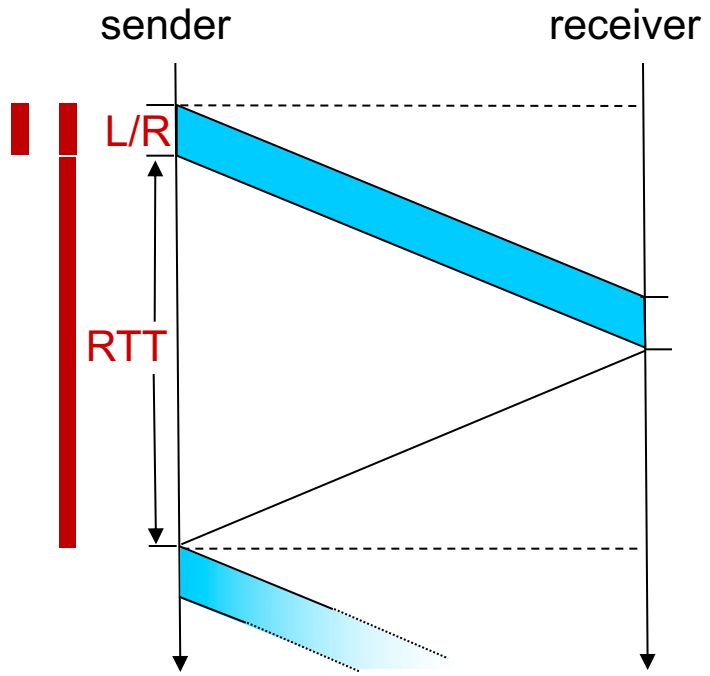
$$= \frac{.008}{30.008}$$

$$= 0.00027$$



- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)
- **How about the throughput?**

rdt3.0: stop-and-wait operation

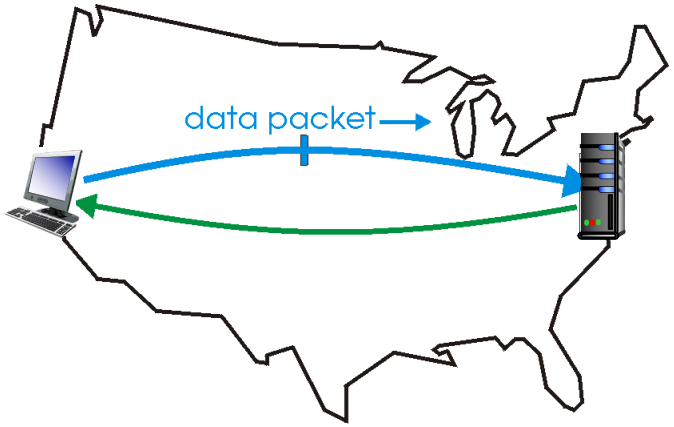


- The effective throughput is only 1000 bytes/30008 msec=267kbps even if we have a 1 Gbps link!

rdt3.0: pipelined protocols operation

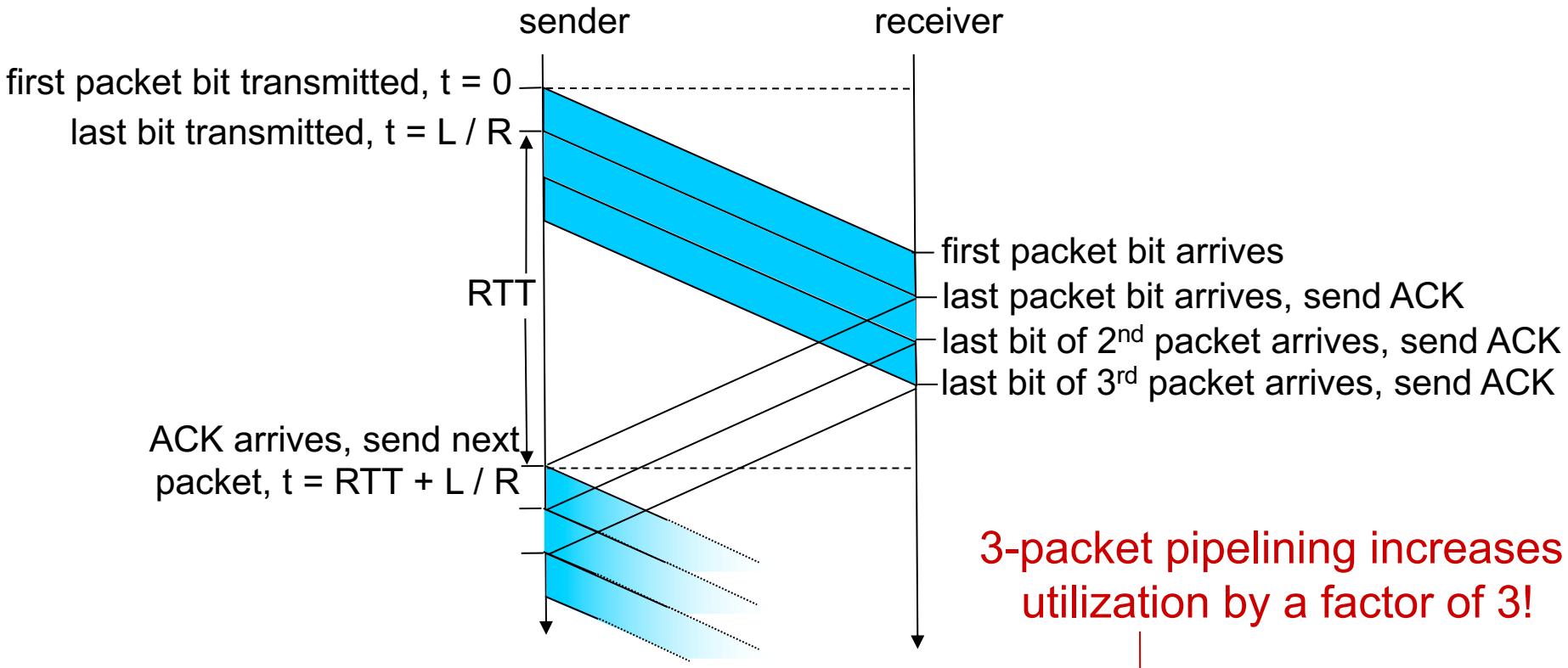
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



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

Pipelining: increased utilization



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

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

Pipelining: increased utilization

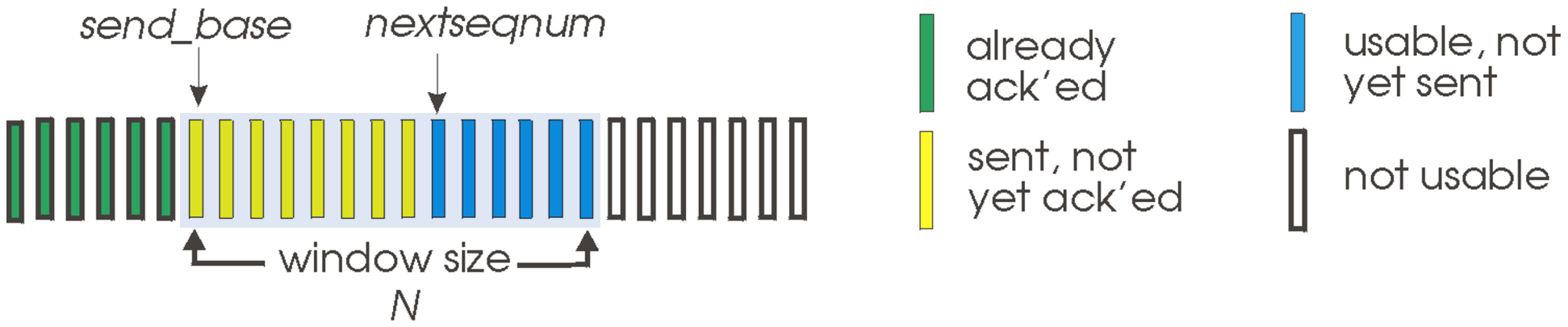
The range of sequence numbers must be increased as there may be multiple in-transit unacknowledged packets

Sender and receiver may have to buffer more than one packet

Two approaches toward pipelined error recovery:
Go-Back-N and Selective Repeat

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

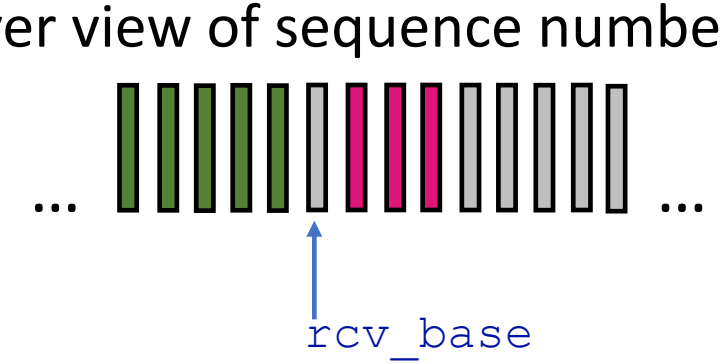


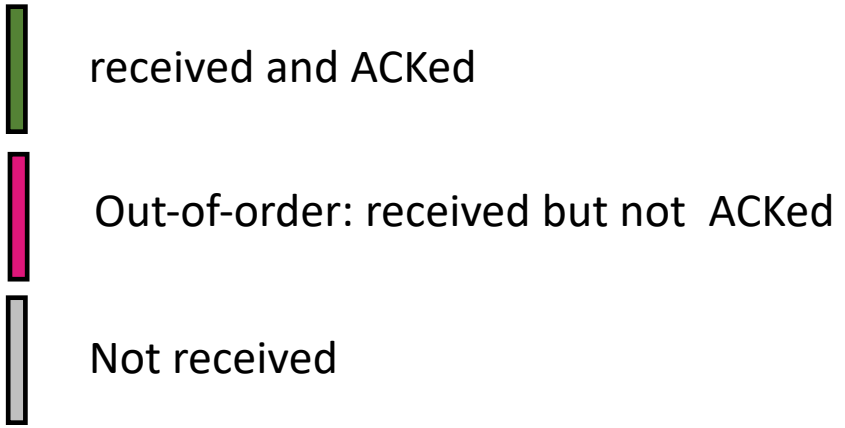
- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

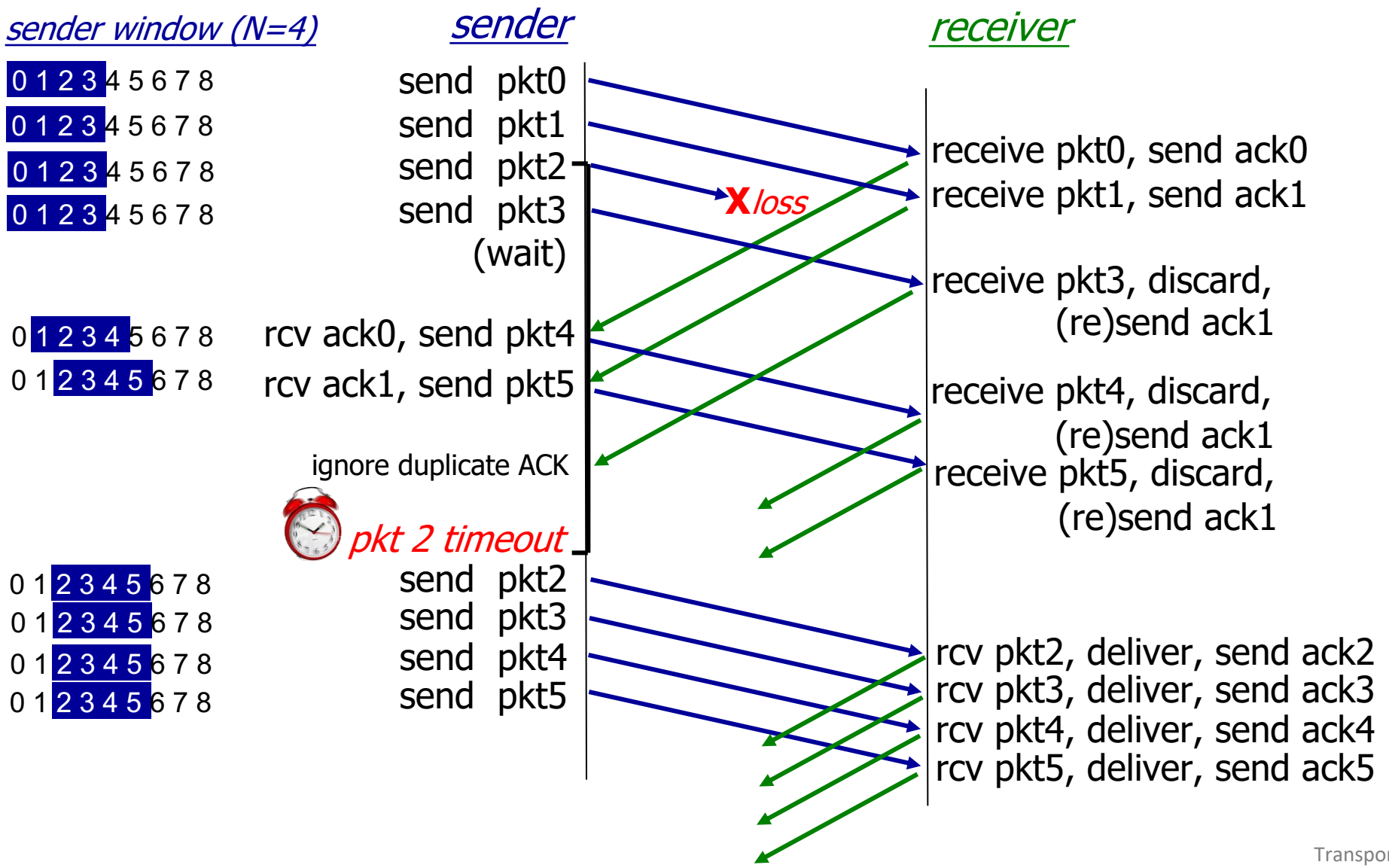
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:





Go-Back-N in action



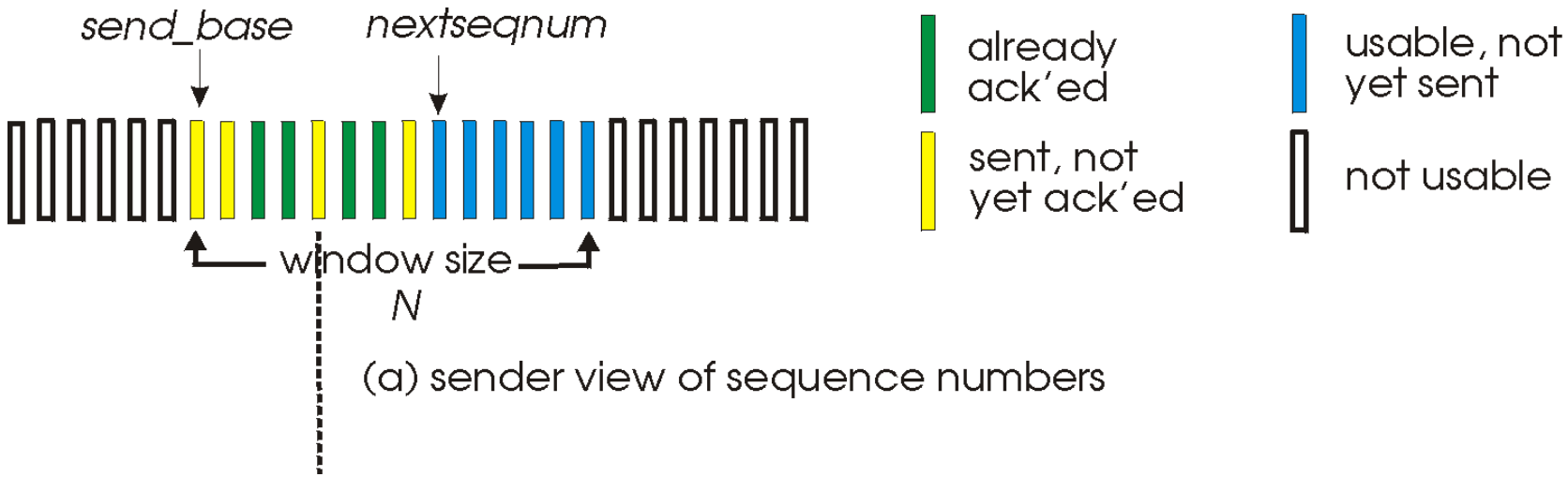
Animations GBN

https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html

Selective repeat: the approach

- *pipelining*: *multiple* packets in flight
- *receiver individually ACKs* all correctly received packets
  - buffers packets, as needed, for in-order delivery to upper layer
- sender:
  - maintains (conceptually) a timer for each unACKed pkt
    - timeout: retransmits single unACKed packet associated with timeout
  - maintains (conceptually) "window" over *N* consecutive seq #s
    - limits pipelined, "in flight" packets to be within this window

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

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

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

receiver

packet *n* in [rcvbase, rcvbase+N-1]

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

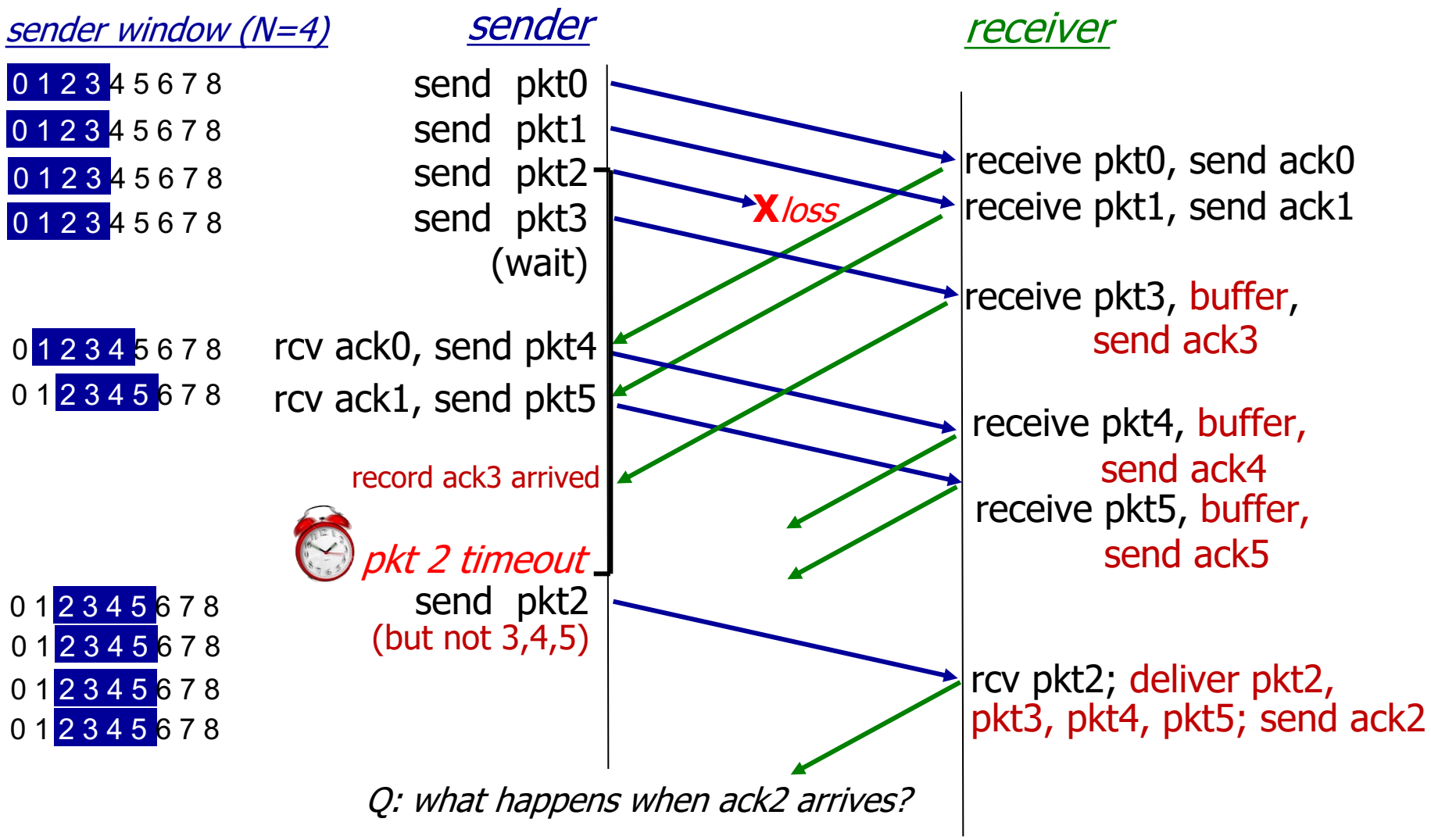
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

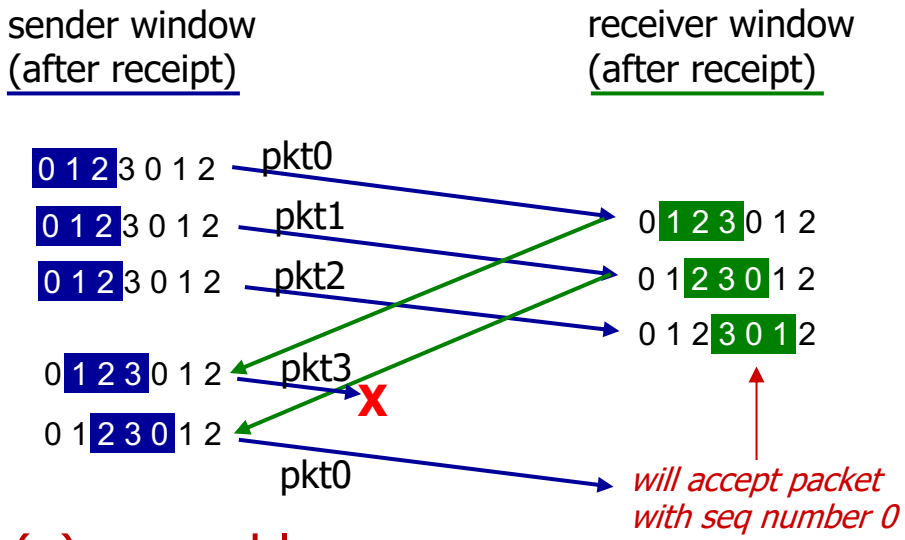
Selective Repeat in action



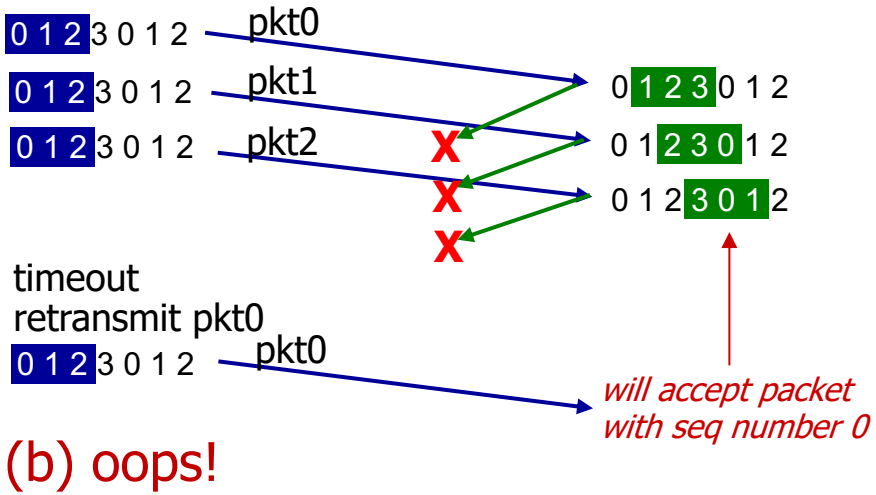
Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



(a) no problem

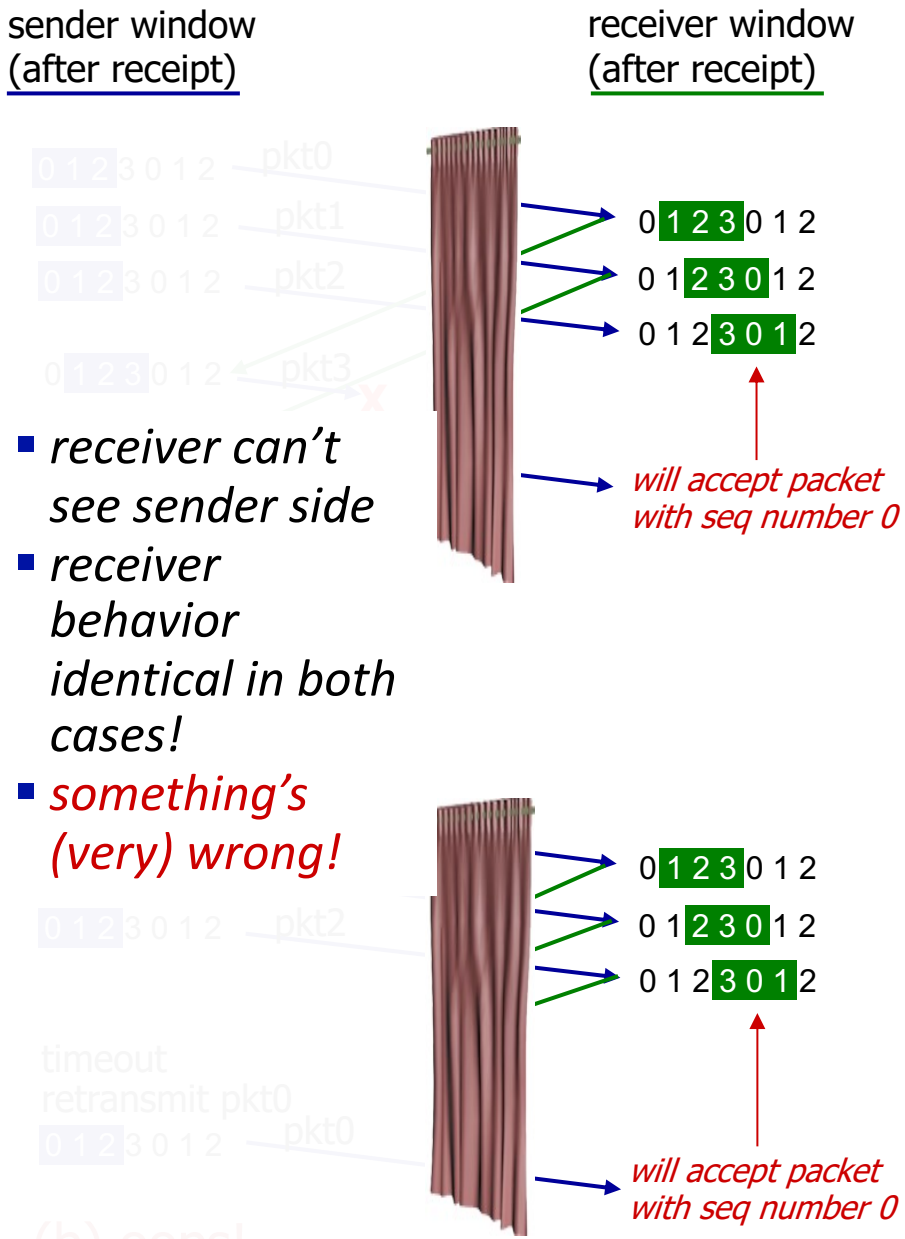


(b) oops!

Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



- *receiver can't see sender side*
- *receiver behavior identical in both cases!*
- *something's (very) wrong!*

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

window size <= max_seq_number /2

Selective repeat: another issue with packet reordering

- With packet reordering and mutiple paths between sender and receiver, the channel can be thought as buffering packets and spontaneously emitting them at any point in the future.
- We need to be sure that packets with the same sequence number do not show up causing troubles…
- TCP packets cannot live in the network for more than three minutes in high speed networks.

Animations SR

https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html

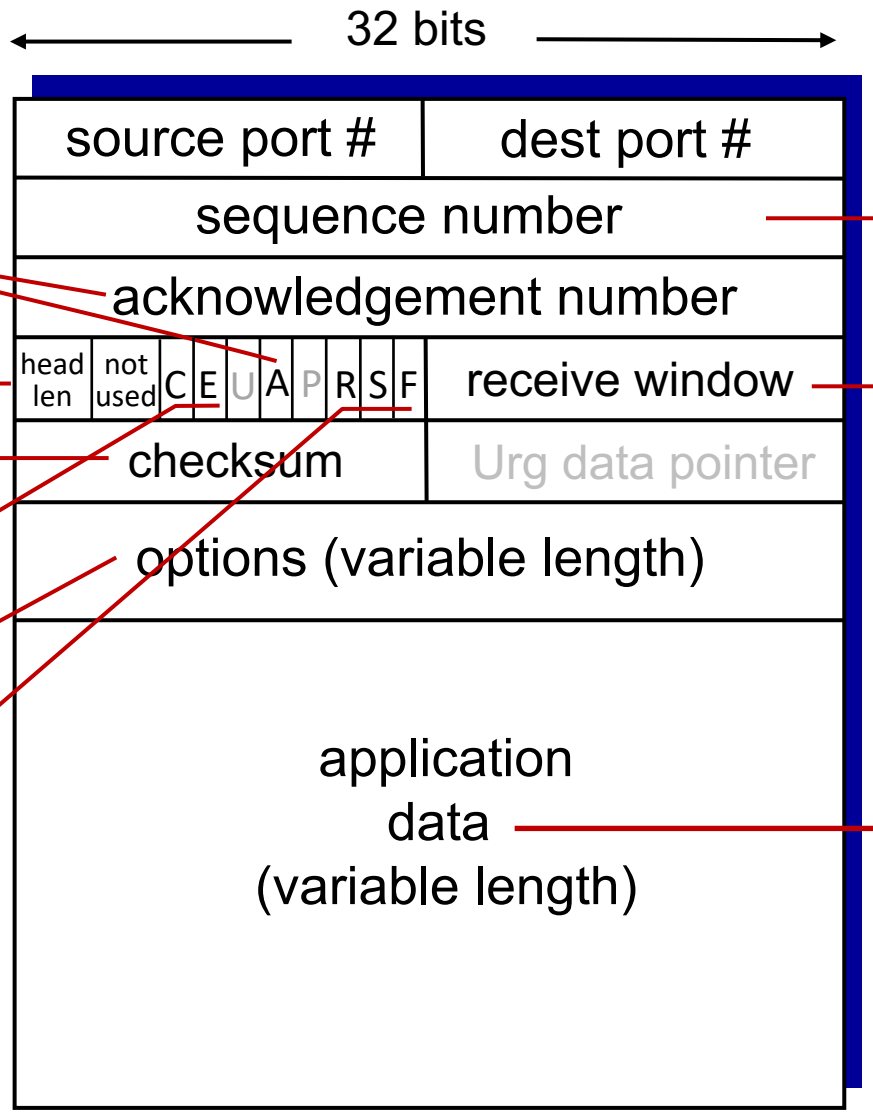
Chapter 3: roadmap

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

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
  - No multicast!
- reliable, in-order *byte stream:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

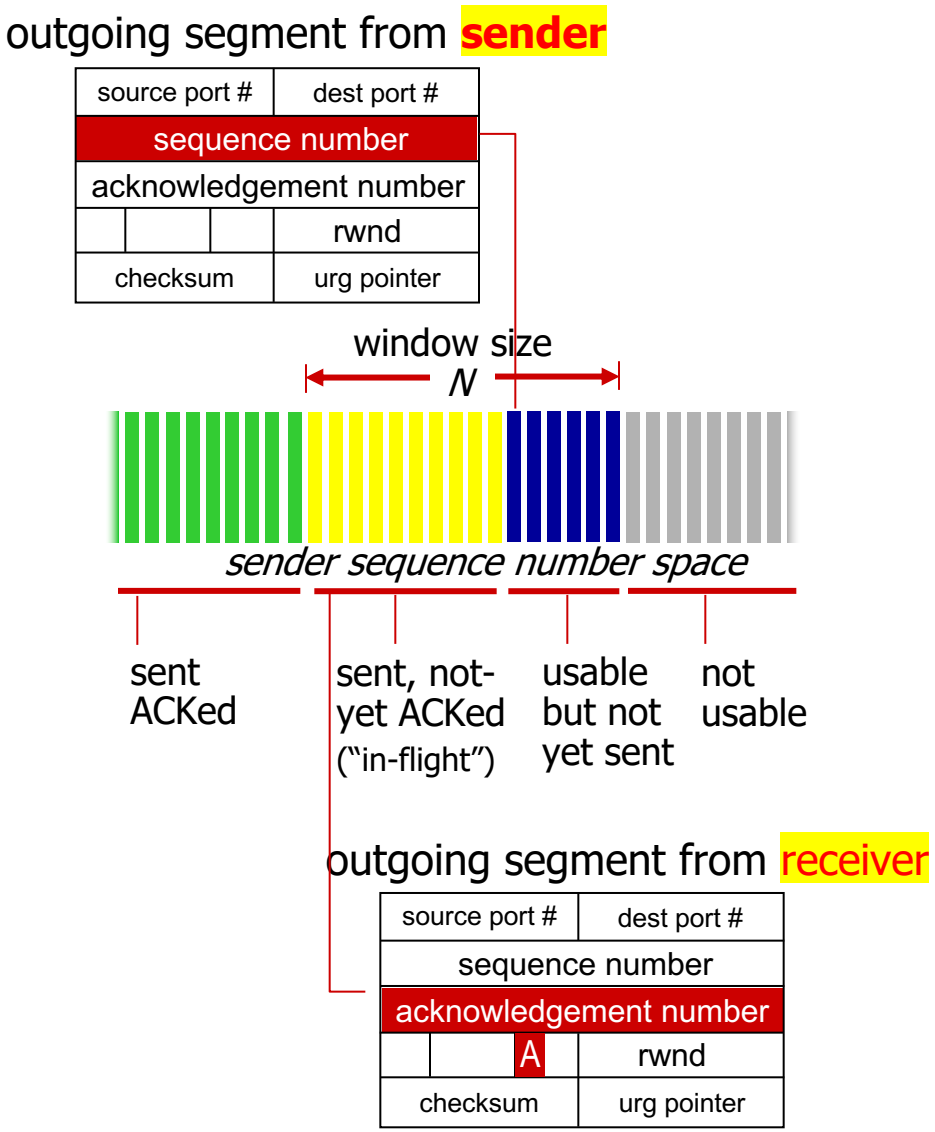
- byte stream "number" of first byte in segment's data

Acknowledgements:

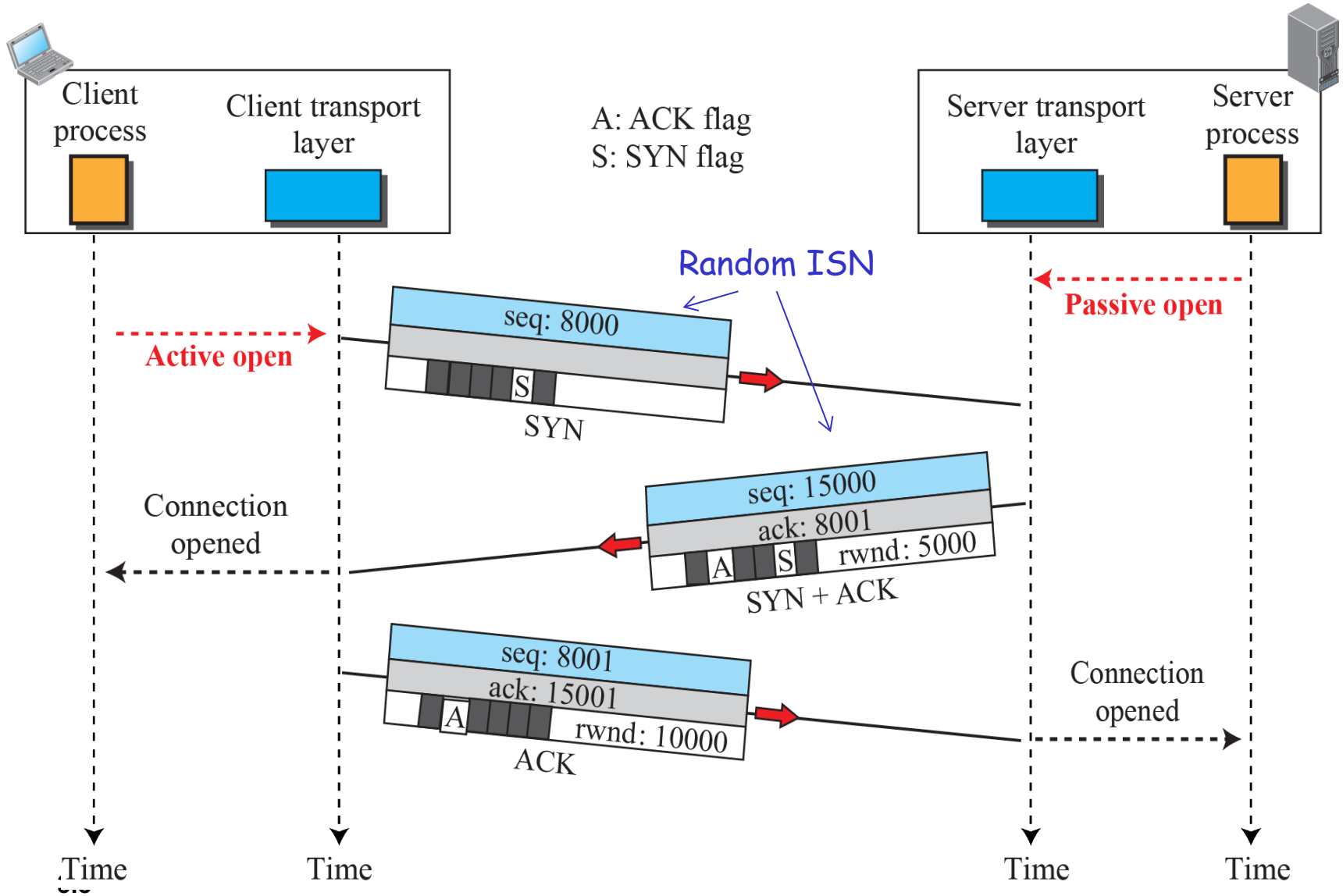
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



Connection establishment: three-way handshake

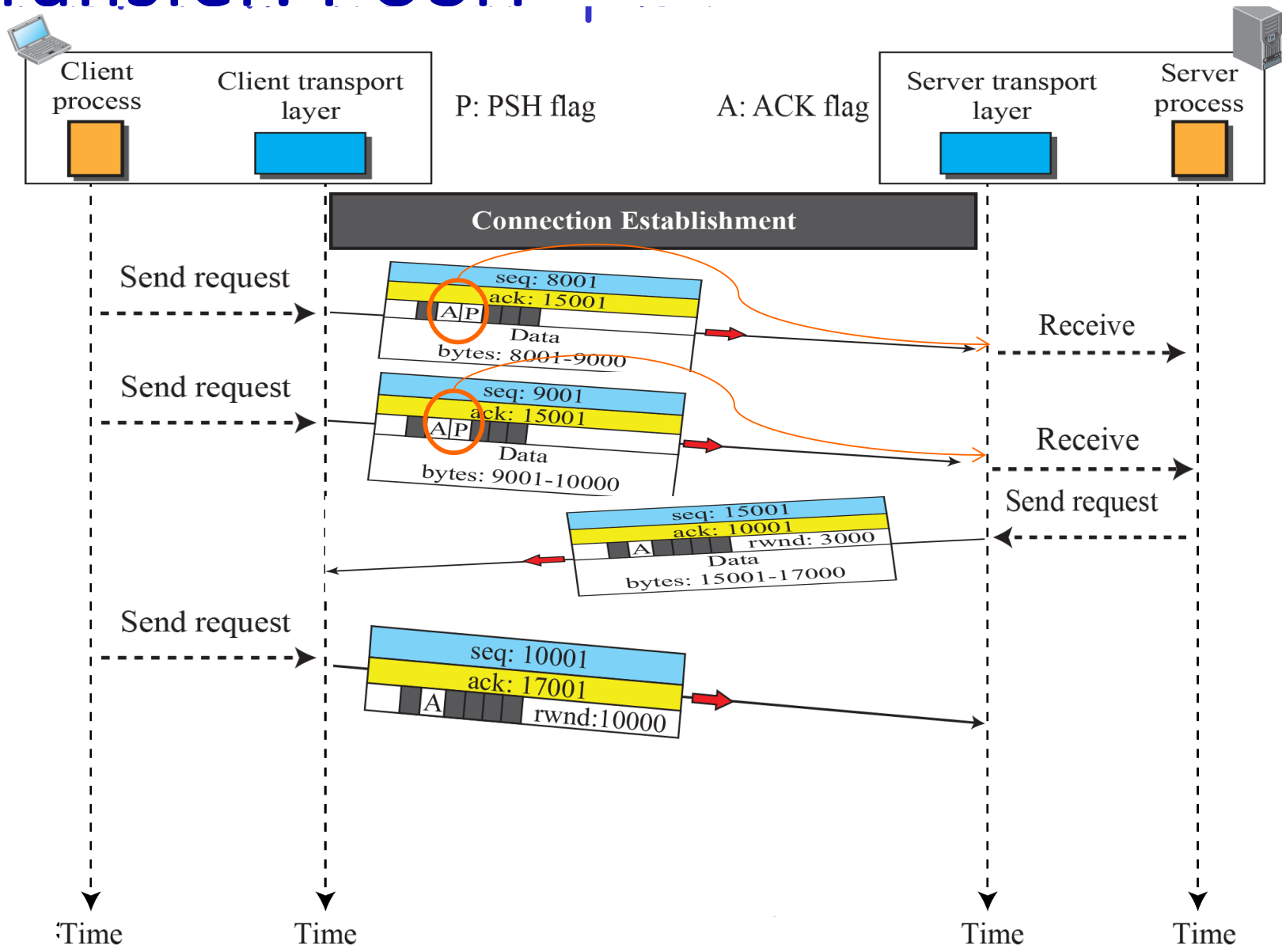


Data transfer: PUSH

send buffer on the sender side

MSS: Maximum Segment Size

CONNECTION STATE



receiver buffer on the receiver side

CONNECTION STATE

PIGGYBACK OF ACKs

MSS: maximum segment size

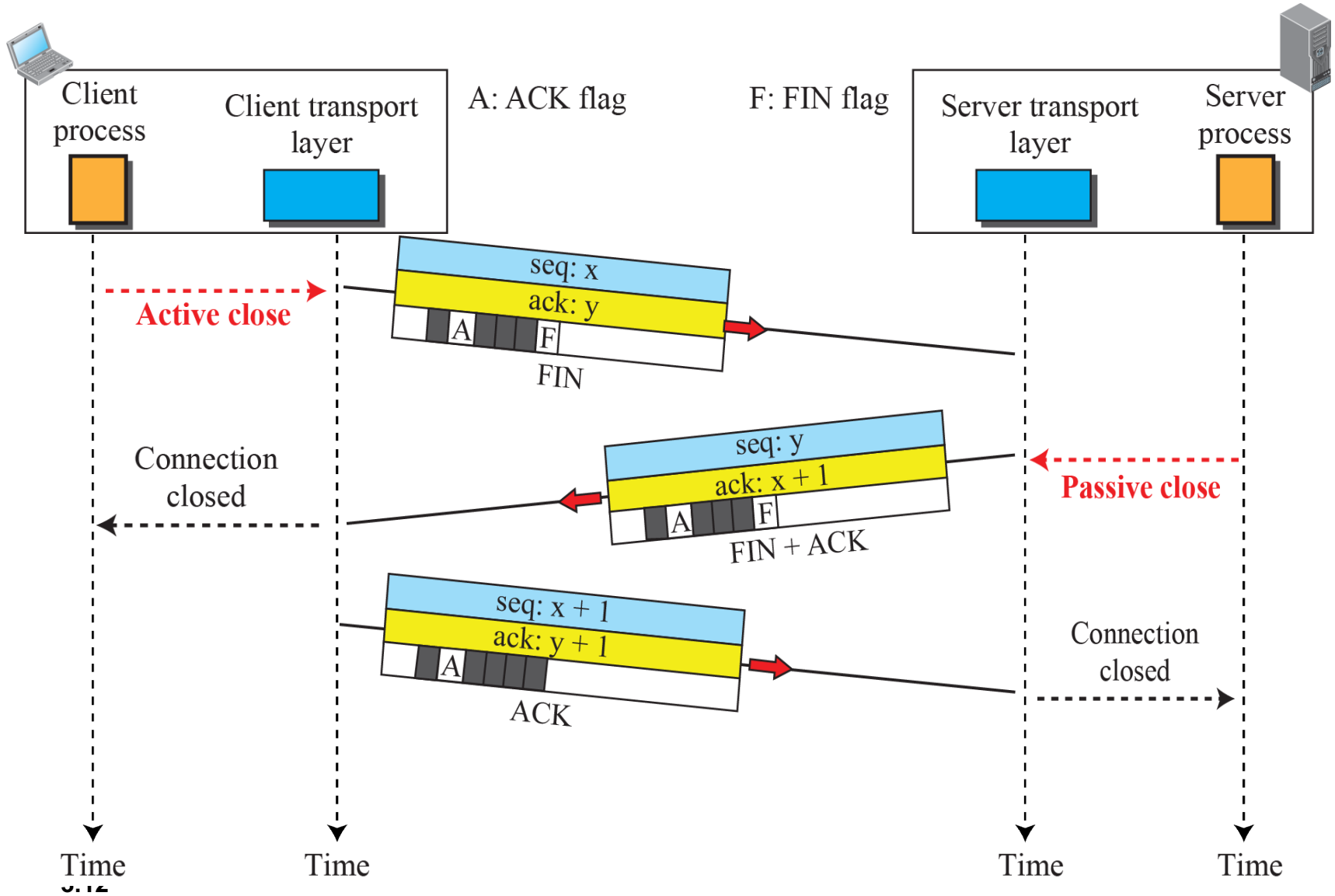
MSS is the max amount of data in a segment

Consider the MTU (Maximum Transfer Unit),
i.e., the length of the largest data link frame

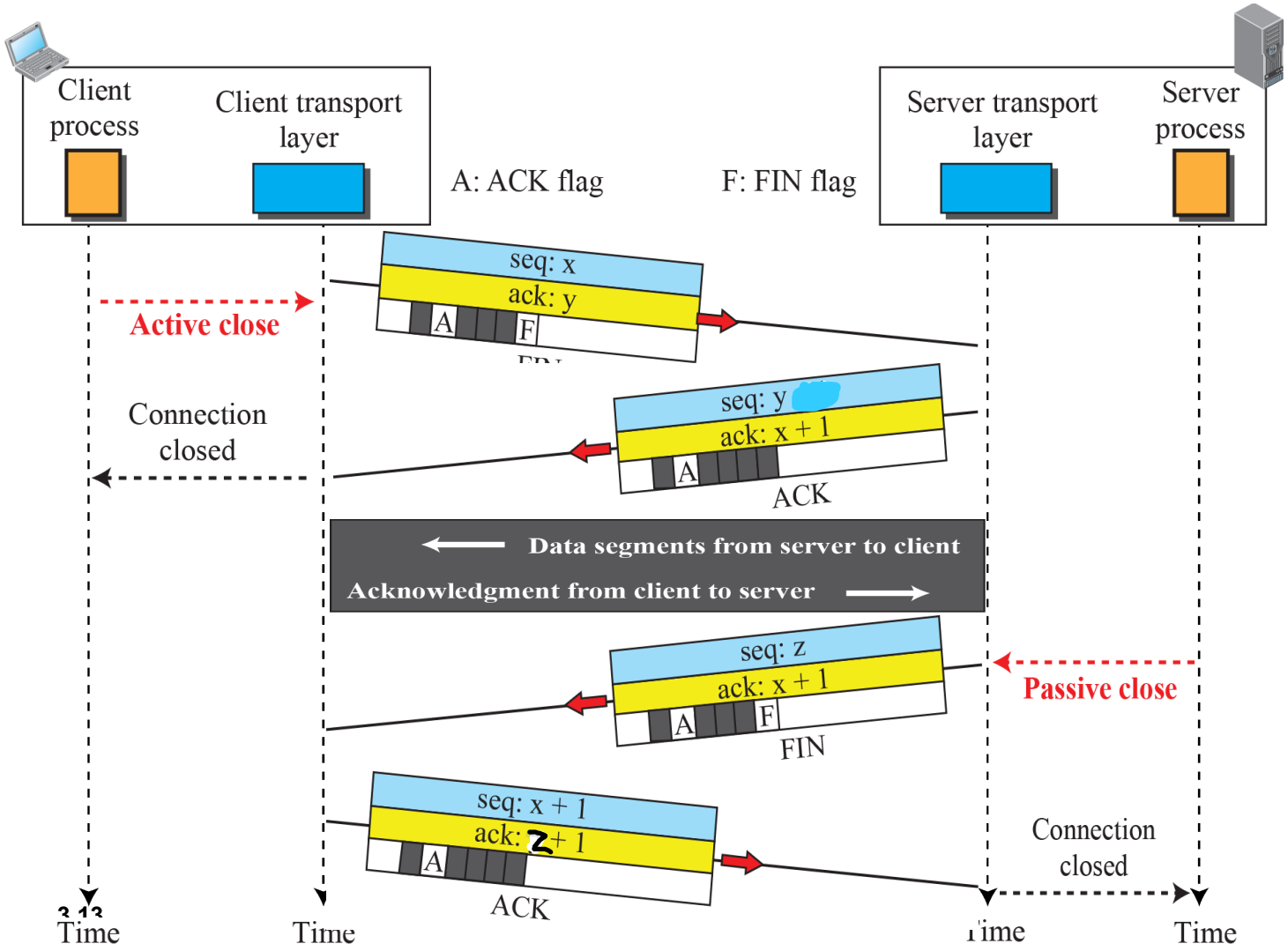
TCP segment + TCP header (typically 40 bytes) must fit into the MTU

For ethernet it is MTU=1,500 bytes
Therefore MSS=1,460 bytes

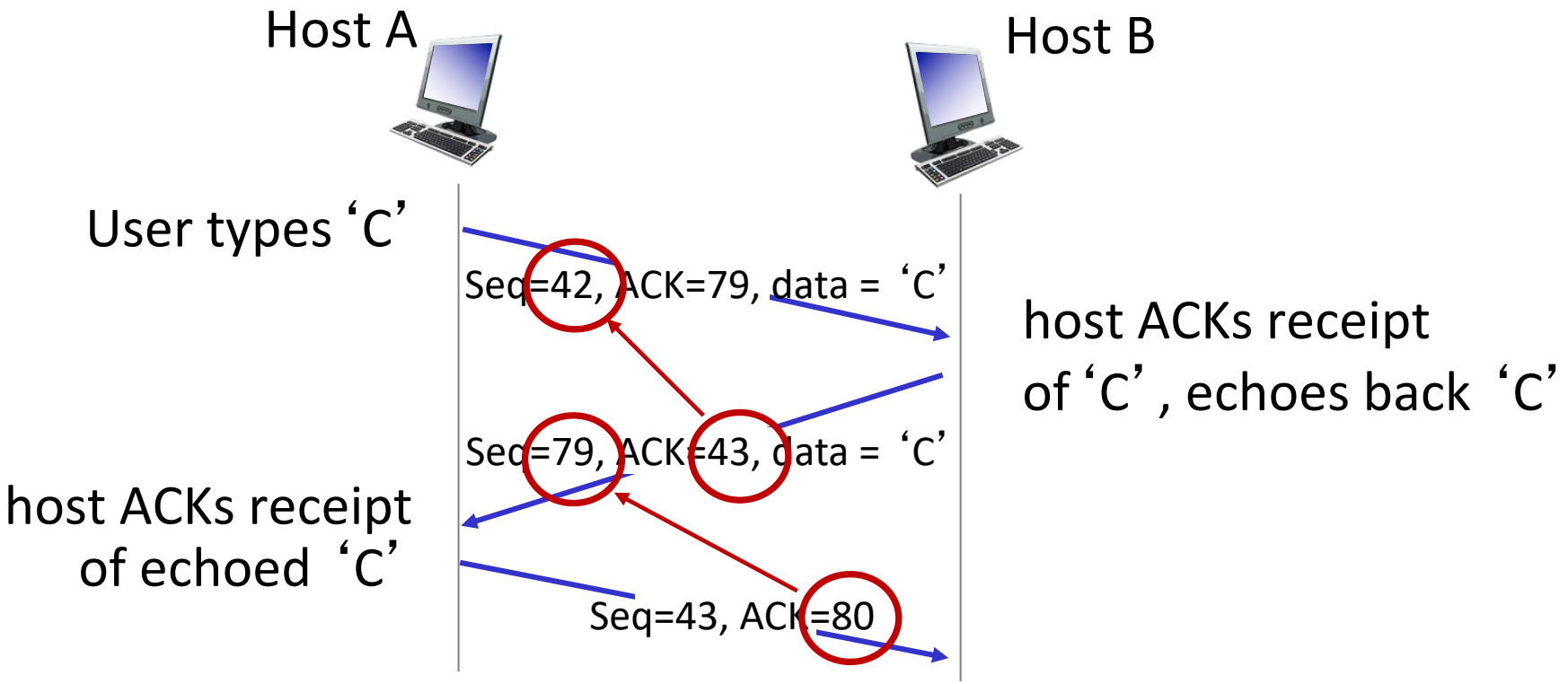
Connection close



Connection half-close



TCP sequence numbers, ACKs



simple telnet scenario – ACKs' piggyback

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`

Estimating RTT

Estimating RTT

- Notation
  - $\mathtt{rtt}_i$ = time between transmission of *i*th packet until receive ACK of *i*th packet
  - $\mathtt{RTT}_i$ = estimate of average round trip time after *i*th packet
- Exponentially weighted moving average (EWMA):

$$\mathtt{RTT}_i = \alpha * \mathtt{RTT}_{i-1} + (1-\alpha) * \mathtt{rtt}_i$$

  - (assume $\mathtt{RTT}_0 = 0$)
  - Jacobson's algorithm: $\alpha = .875$

Called "exponential" because the weigth of a given sample decays exponentially fast

Example

Example of RTT Calculation

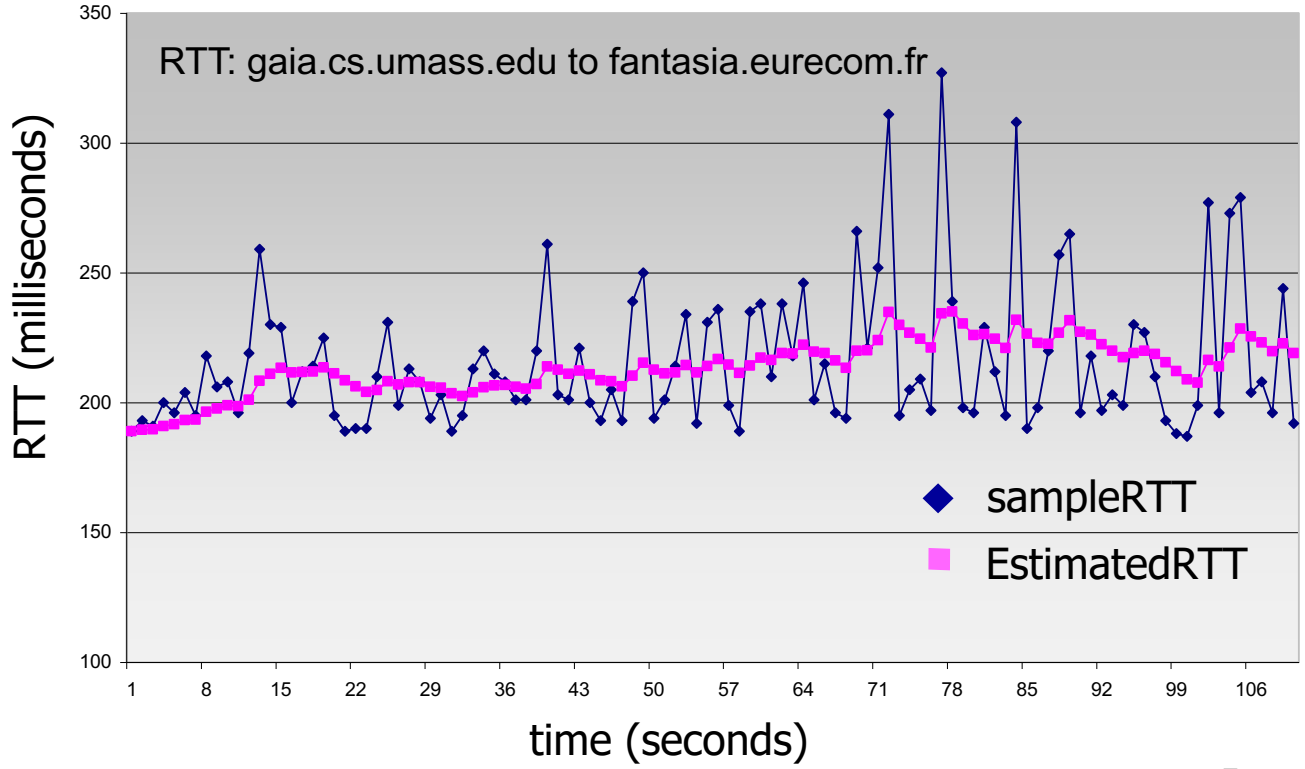
- Notes: calculations in full precision, results shown only to nearest integer, time before packet #20 not shown

Pkt #	rtt_i (ms)	RTT_i (ms) $\alpha = .875$
20		40
21	30	39
22	80	44
23	40	43
24	130	54
25	40	?

TCP round trip time, timeout

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.875



Small Correction

- Problem: if a segment is retransmitted, is an ACK for the first, second, … or n^{th} transmission of that segment?
  - can't tell how to compute $\mathtt{rtt}_i$ in this case
- Solution ("Karn's Algorithm"): don't use (i.e., ignore) $\mathtt{rtt}_i$ from any retransmitted segments to update $\mathtt{RTT}_i$

ACKs and Retransmissions

- When a segment is sent, a retransmission timer is started
- If the segment is **ACK**'ed before the timer expires, the timer is turned off
- Otherwise, the segment is retransmitted and timer is started again
- Notation
  - $\mathbf{RTO}_i$ = retransmission timeout interval for the i^{th} packet

Determining RTO

- RFC 793 originally recommended:

  $RTO_i = 2 * RTT_{i-1}$

  – (i.e. allow two roundtrip times before timing out)

- Problem

  – when RTT_i has a high standard deviation, this method frequently times out too quickly (doesn't wait long enough)

Example of RTO Calculation (Original)

Pkt #	rtt_i (ms)	RTO_i	RTT_i (ms) α = .875
20			40
21	30	80	39
22	80	78	44
23	40	88	43
24	130	86	54
25	50	108	54

Retransmission timeouts would occur! (retransmissions not shown, Karn's algorithm not used)

Improved Method of Computing RTO

- Solution
  - set **RTO** = **RTT** + a multiple of the mean deviation
- Jacobson's algorithm:

$$RTO_i = RTT_{i-1} + 4 * MDEV_{i-1}$$

  - calculations can be implemented in a very efficient way

Variation of RTT

Calculating the Variation in the RTT

- Motivation: ???
- True standard deviation (expensive to compute):

$$STDDEV = ((rtt_1 - avg(rtt))^2 + (rtt_2 - avg(rtt))^2 + \ldots)^{1/2}$$

- The "*mean deviation*" (cheap to compute):

$$MDEV_i = (1-\rho) * MDEV_{i-1} + \rho * |rtt_i - RTT_{i-1}|$$

  - another EWMA!
  - recommended value for $\rho = .25$

Example of MDEV calculation

Example of MDEV Calculation

Pkt #	$\mathtt{rtt}_i$ (ms)	$\mathtt{RTT}_i$ (ms) $\alpha = .875$	$\mathtt{MDEV}_i$ (ms) $\rho = .25$
20		40	10
21	30	39	10
22	80	44	18
23	40	43	14
24	130	54	32
25	50	54	25

Example of RTO Calculation (Improved)

Pkt #	rtt_i (ms)	RTO_i	RTT_i (ms) $\alpha = .875$	$MDEV_i$ (ms) $\rho = .25$
20			40	20
21	30	120	39	18
22	80	109	44	23
23	40	138	43	19
24	130	118	54	36
25	50	196	54	28

Retransmission still occurs!
(no retransmissions shown, karn's algorithm not used)

Initial value of RTO is 1 second (recommended in RFC 6298)

TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT:** want a larger safety margin

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

estimated RTT

"safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

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

(typically, `rho` = 0.25)

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

TCP Sender (simplified)

event: data received from application

- 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`**

event: timeout

- retransmit segment that caused timeout
- restart timer

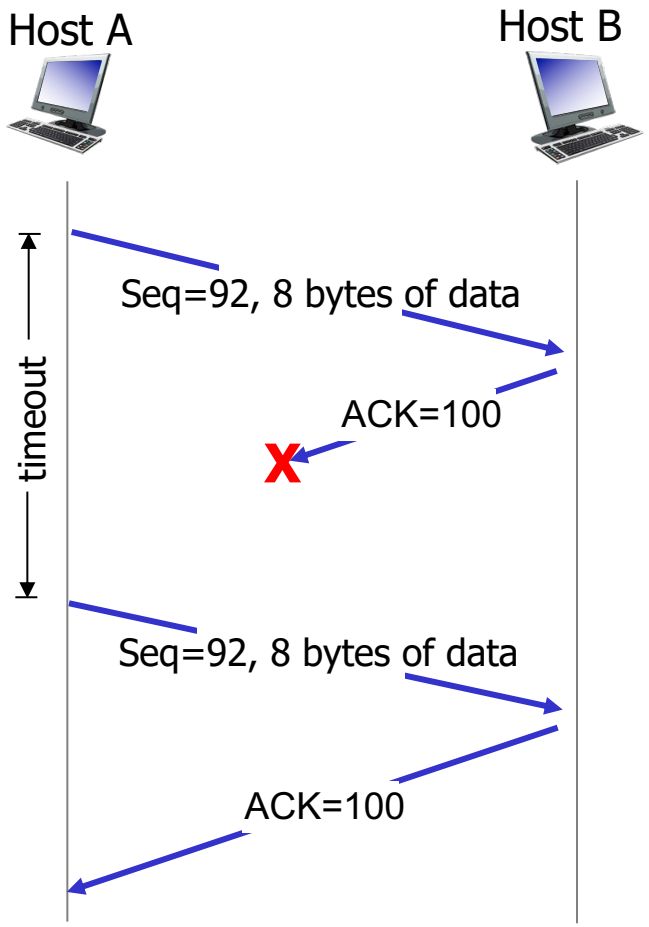
event: ACK received

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

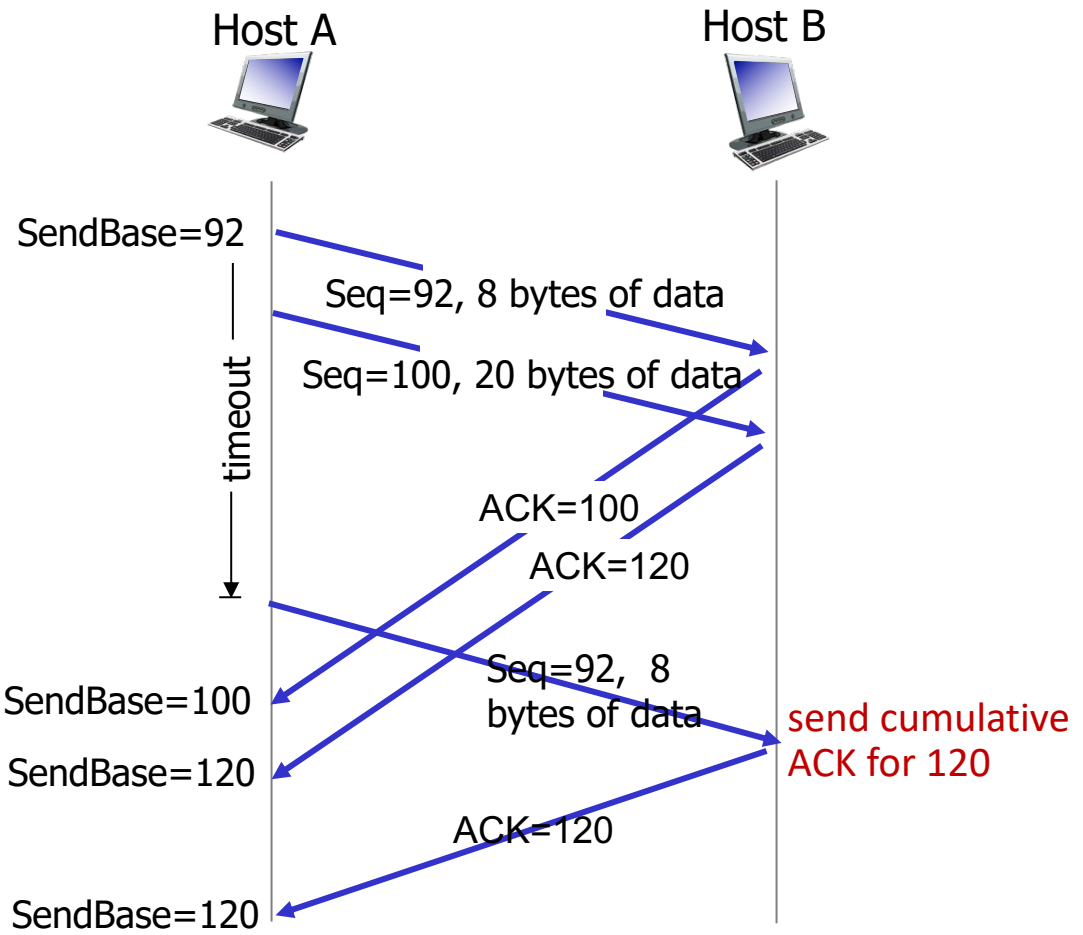
TCP Receiver: ACK generation [RFC 5681]

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: retransmission scenarios

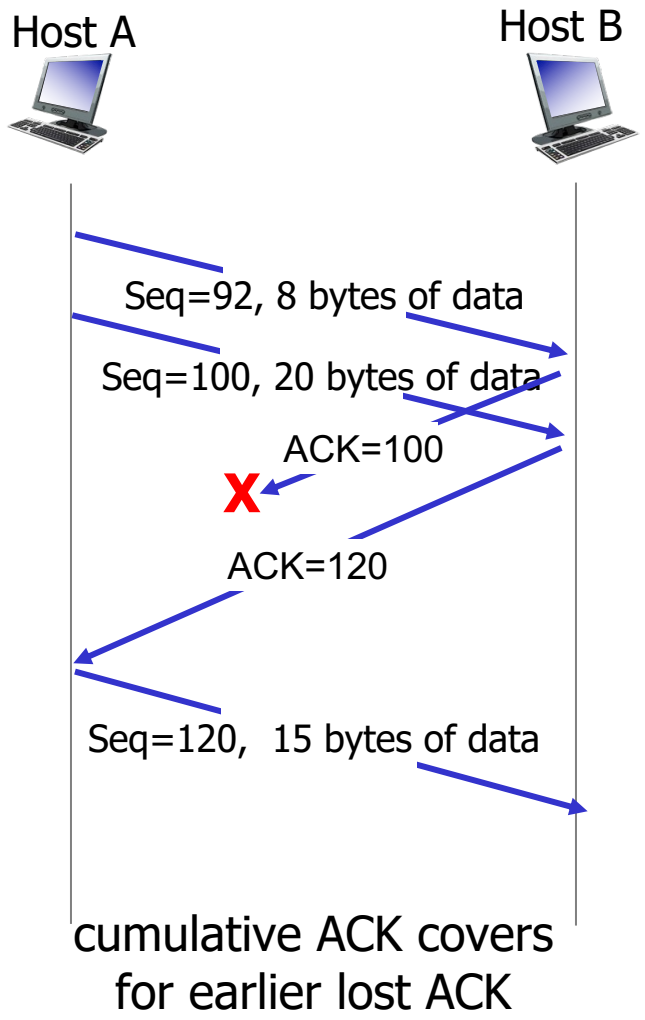


lost ACK scenario



premature timeout

TCP: retransmission scenarios



Retransmission Timer Backoff

- Retransmission may occur because **RTO** is too short
  - e.g., if the network congestion increases, RTO should be increased
  - problem: Karn's algorithm says don't modify RTT / MDEV for retransmitted segments!
- Solution: *exponential backoff*
  - upon timeout, retransmit and compute:
    $RTO_i = 2 * RTO_{i-1}$
    and no change to RTT

Retransmission Timer Backoff (…)

Example: RTO=2s
After first timeout = 4s
After second timeout = 8s, etc.

The interval grows exponentially after each retransmission.
Set back as functions of the previous estimation in case of

- data sent from the application or
- ACK received

-> limited form of congestion control

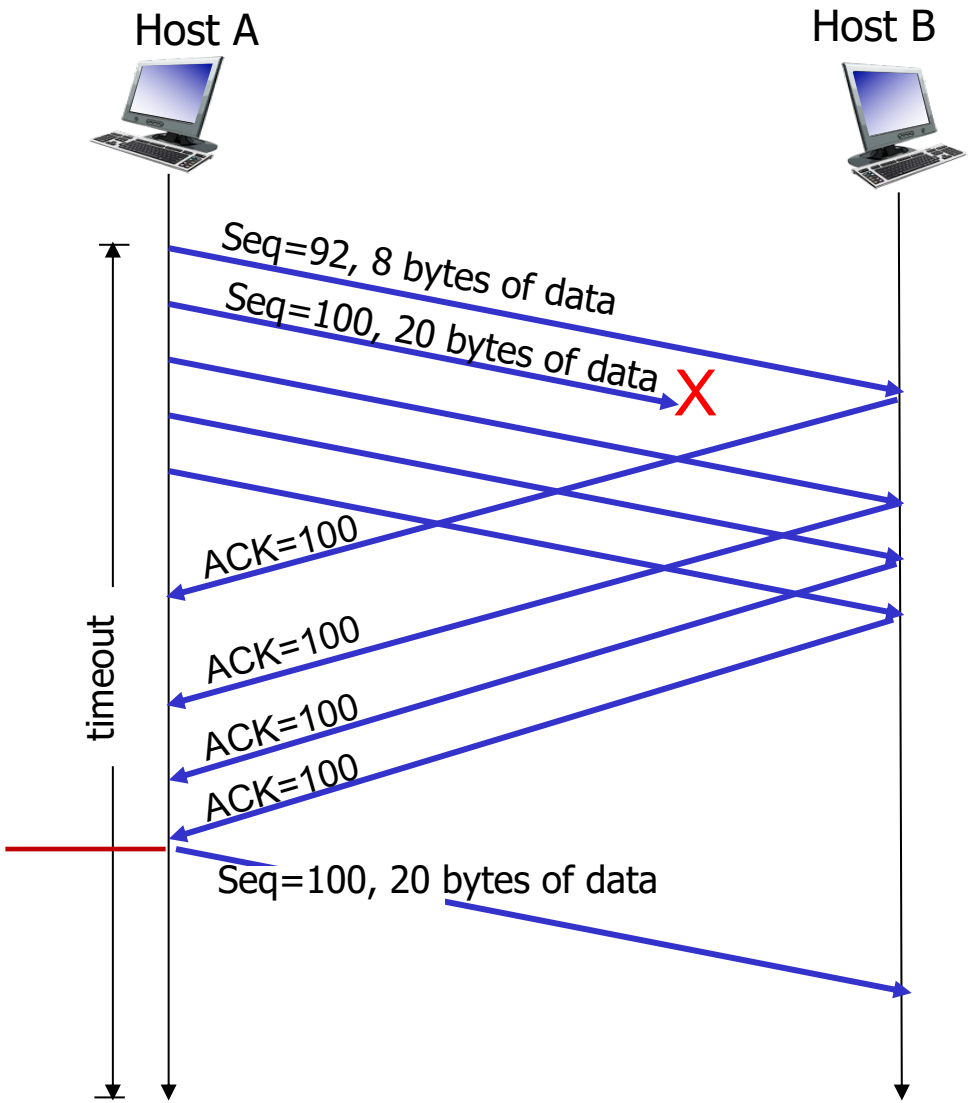
TCP fast retransmit

TCP fast retransmit

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

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

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



This retransmission is NOT motivated by the occurrence of a time-out! Same as a NACK (not necessarily congestion related, simply an error, or an occasional loss)

TCP adopted mechanisms

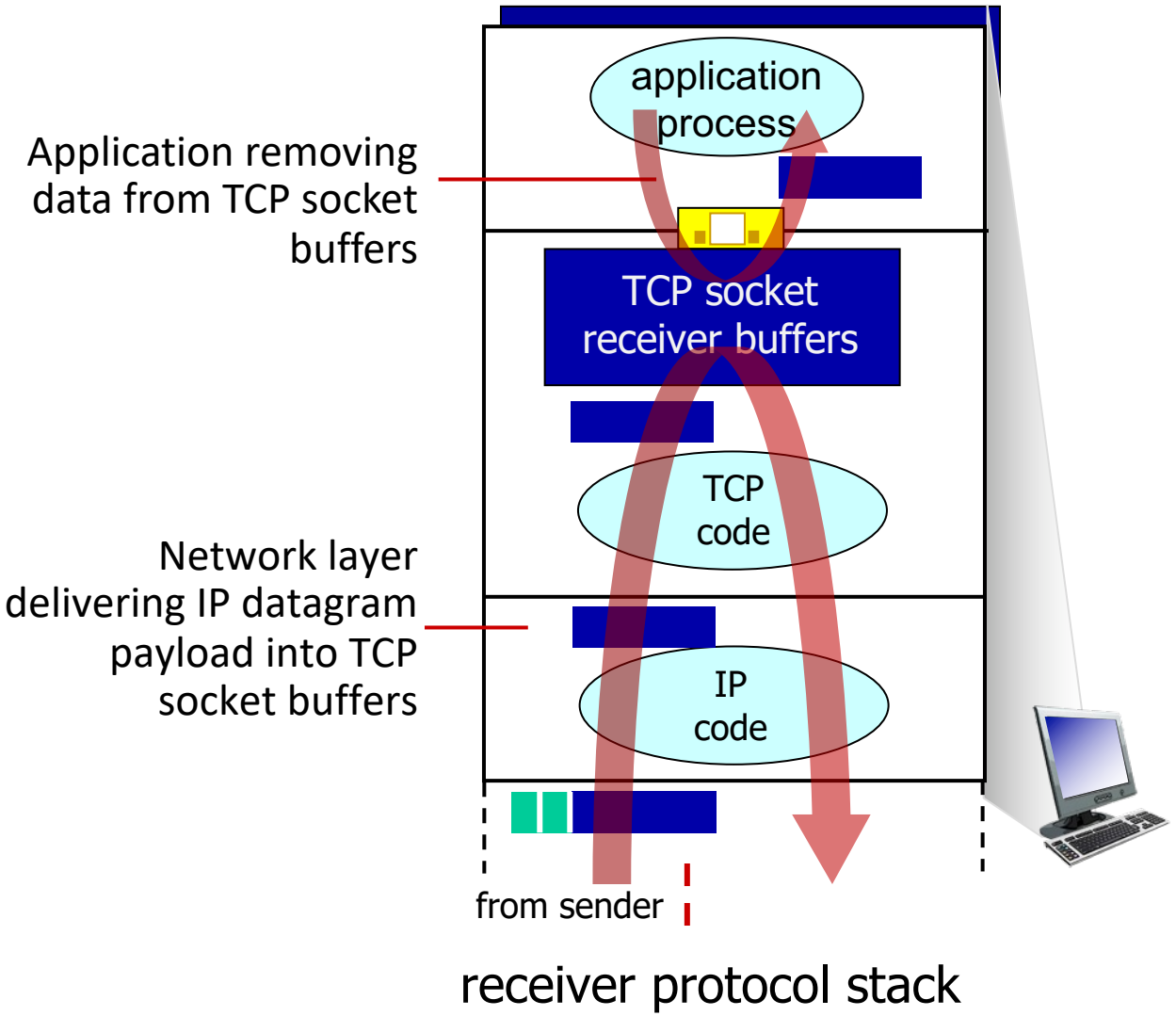
- **Pipeline** (hybrid between Go-Back-N and Selective-Repeat)
- **Full-duplex** with **piggybakcing**
- Use of **Sequence Number** (first byte being transmitted)
- **Cumulative ACK** (it confirms all previous bytes) and delayed ACK (if all previous are ok)
- **RTT-based timeout**: unique retransmission timer (associated to the last unacknowledged segment). When an ack is received, the timer is restarted on the basis of the oldest unacknowledged segment.
- **Retransmission**:
  - *Single-*, only of the unacknowledged segment
  - *Fast-*, when three acks are received before timeout

Chapter 3: roadmap

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

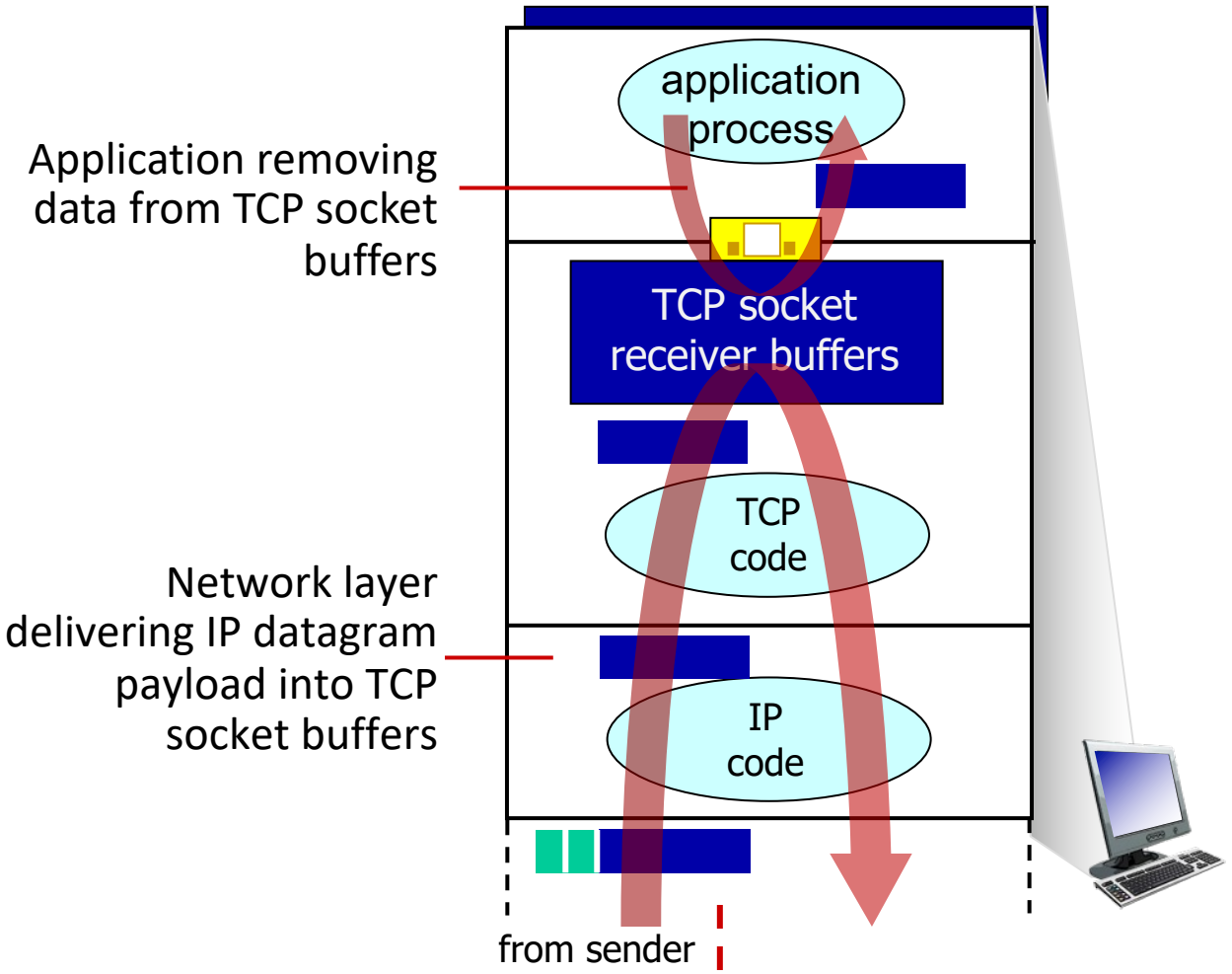
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP flow control

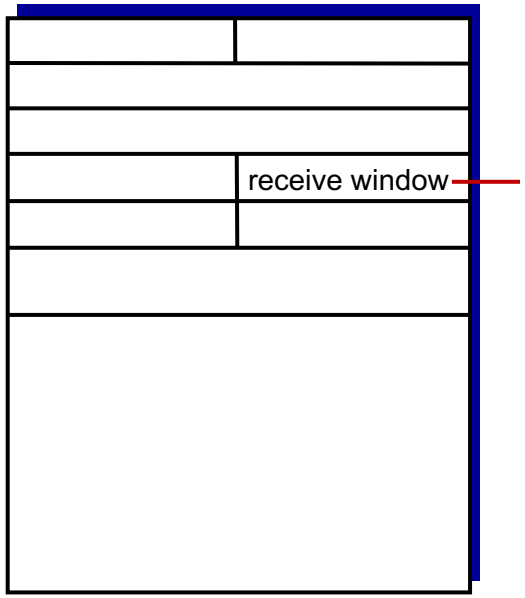
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

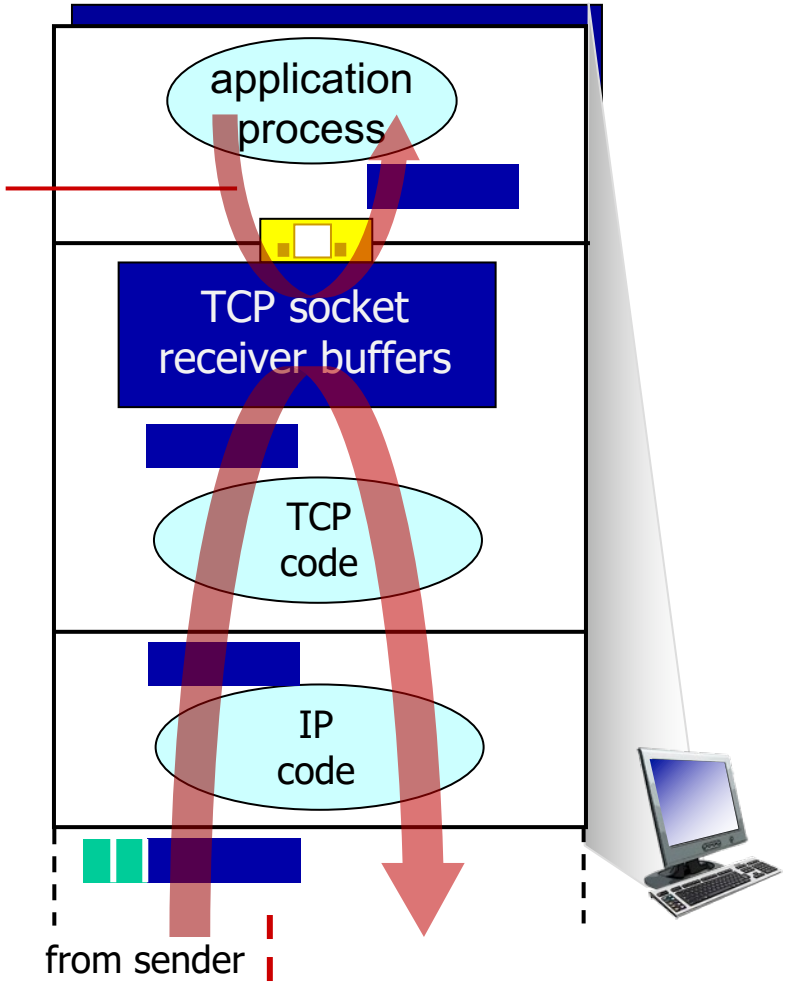
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



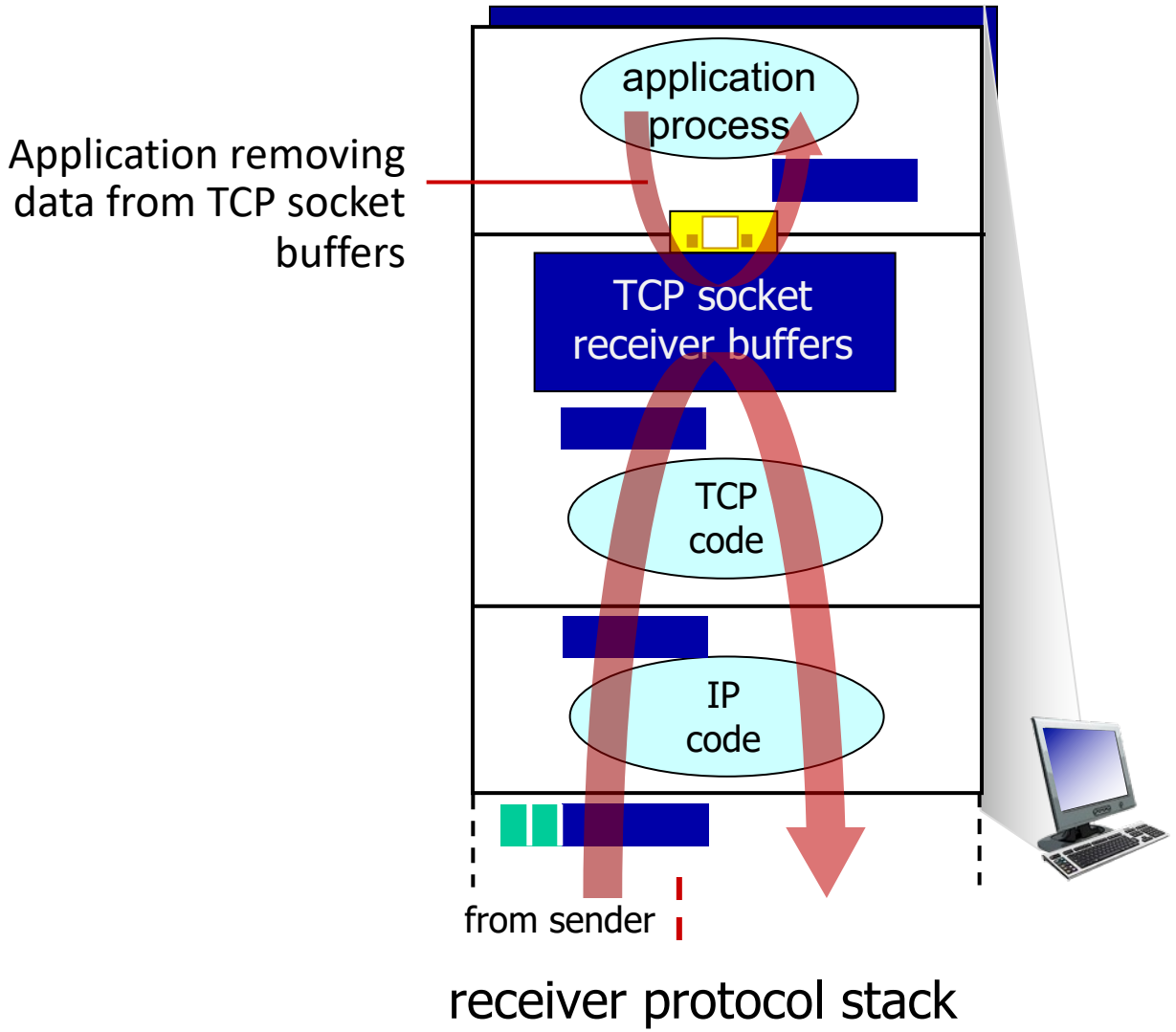
receiver protocol stack

TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

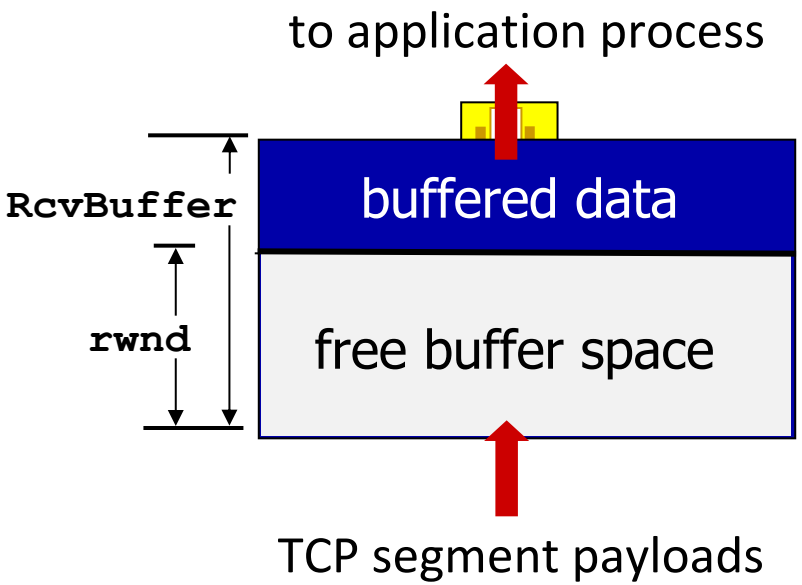
flow control

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



TCP flow control

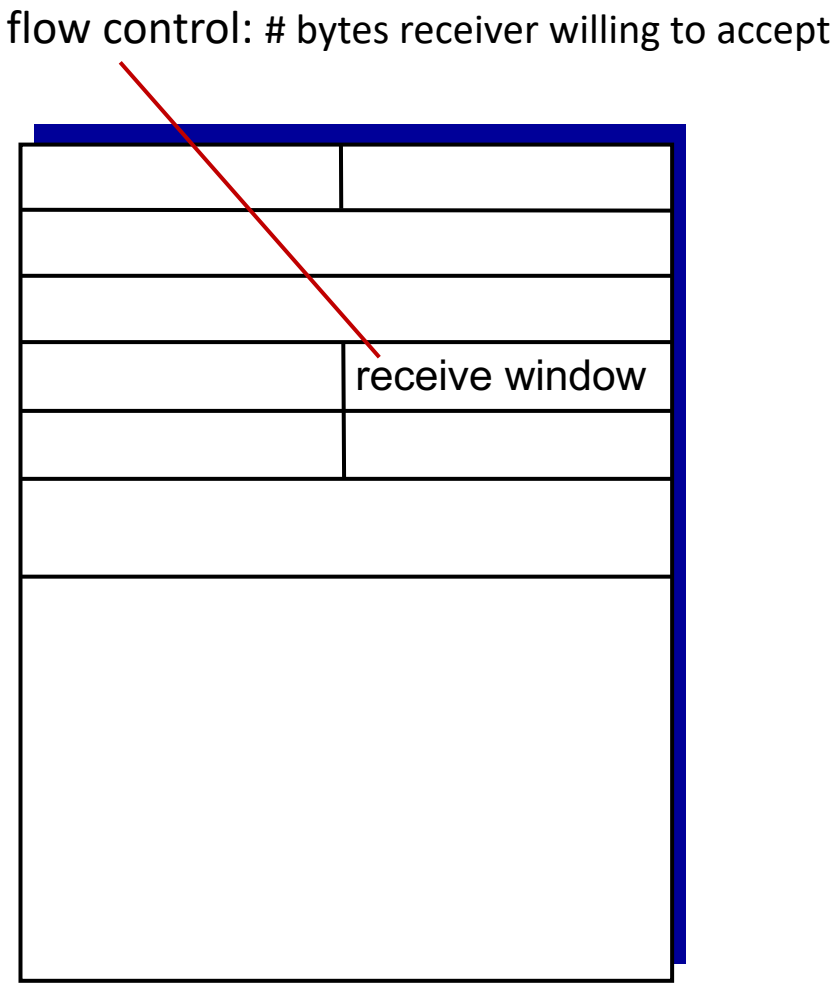
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

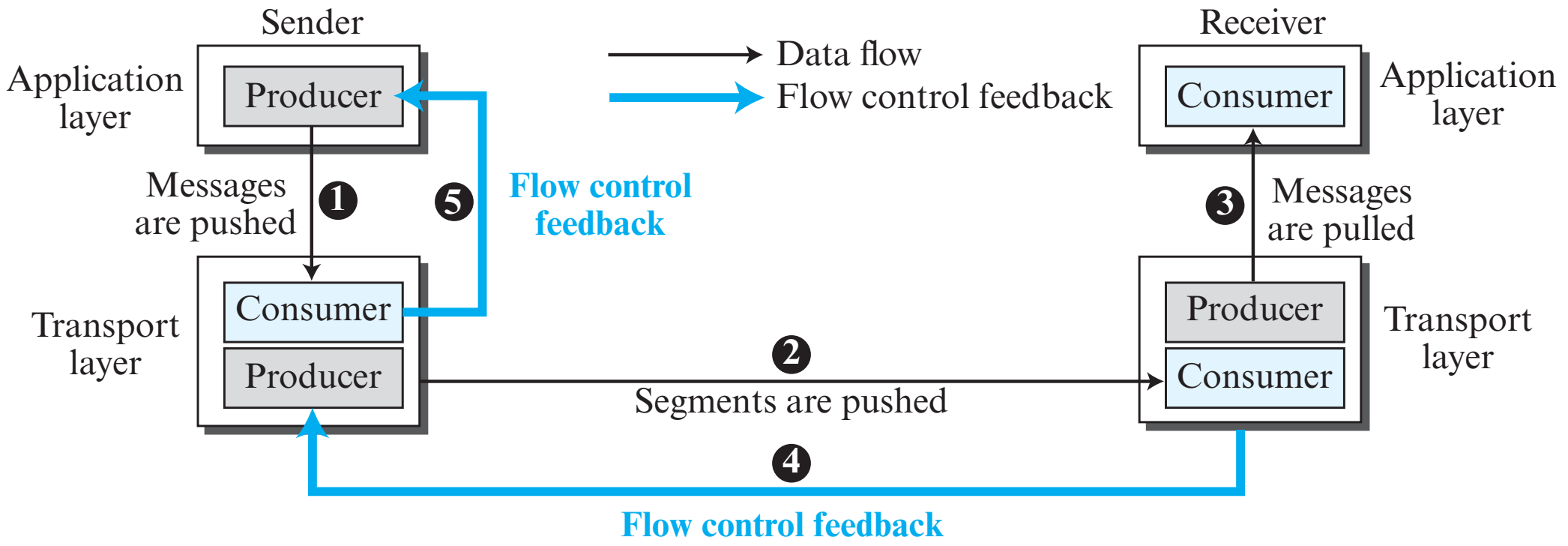
TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`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 received **`rwnd`**
- guarantees receive buffer will not overflow



TCP segment format

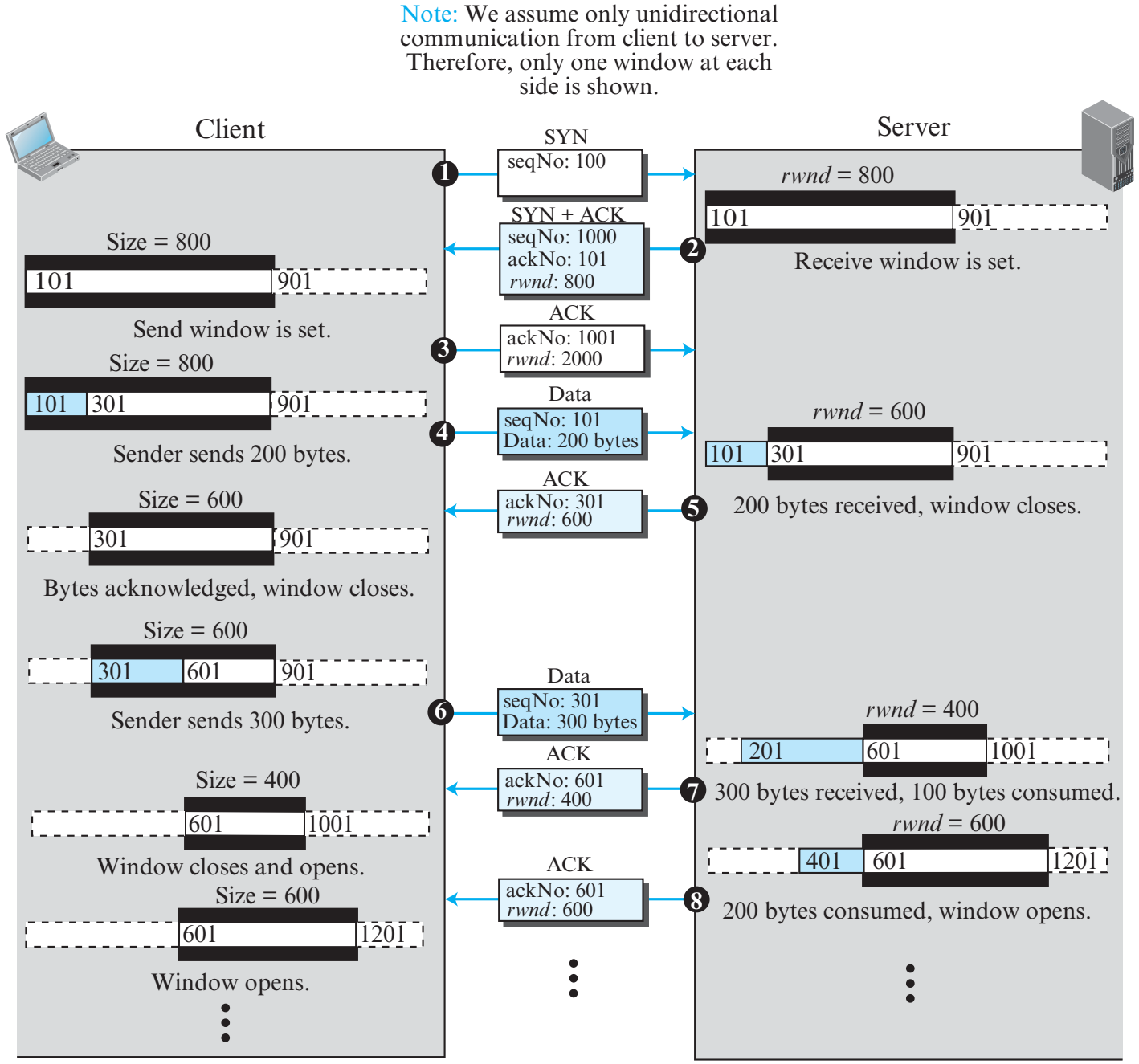
Data flow and flow control feedback



An example of flow control

At the receiver it must always be:
LastByteRcvd-LastByteRead <= RcvBuffer

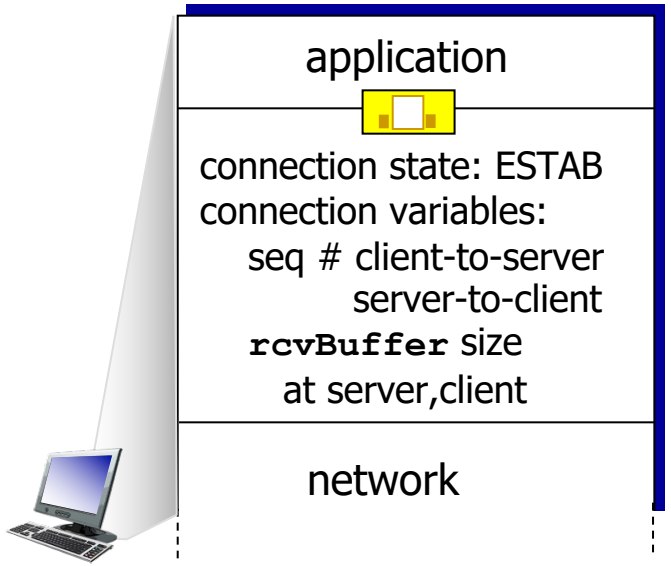
Rwnd=RCVBuffer- (LastByteRcvd-LastByteRead)

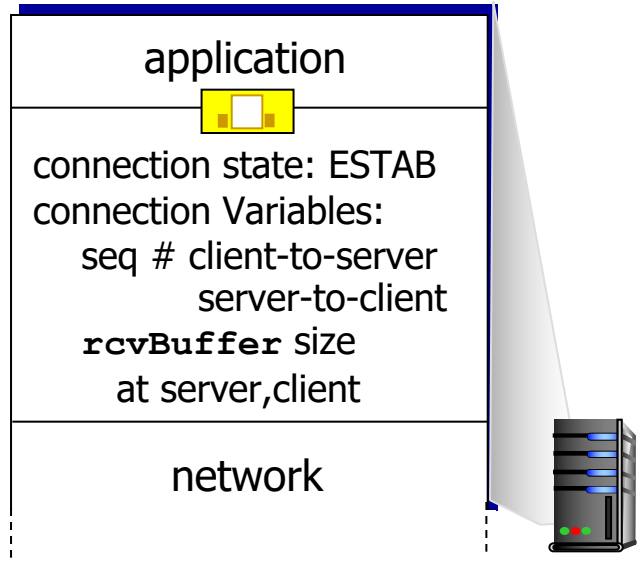


TCP connection management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)



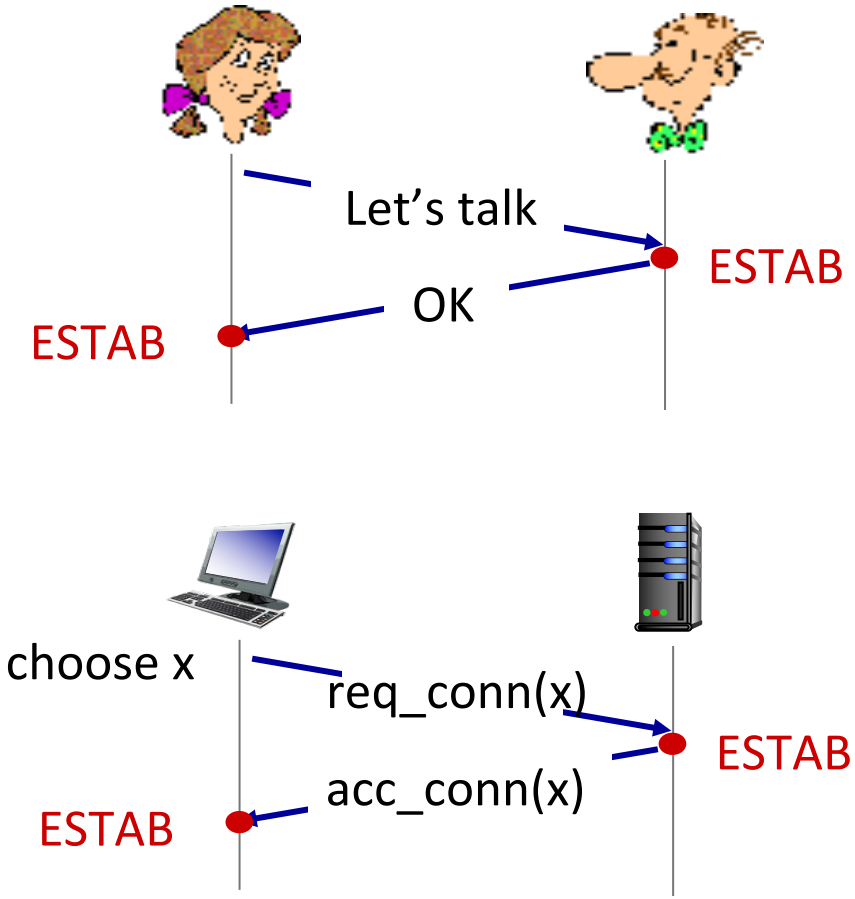



```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

Agreeing to establish a connection

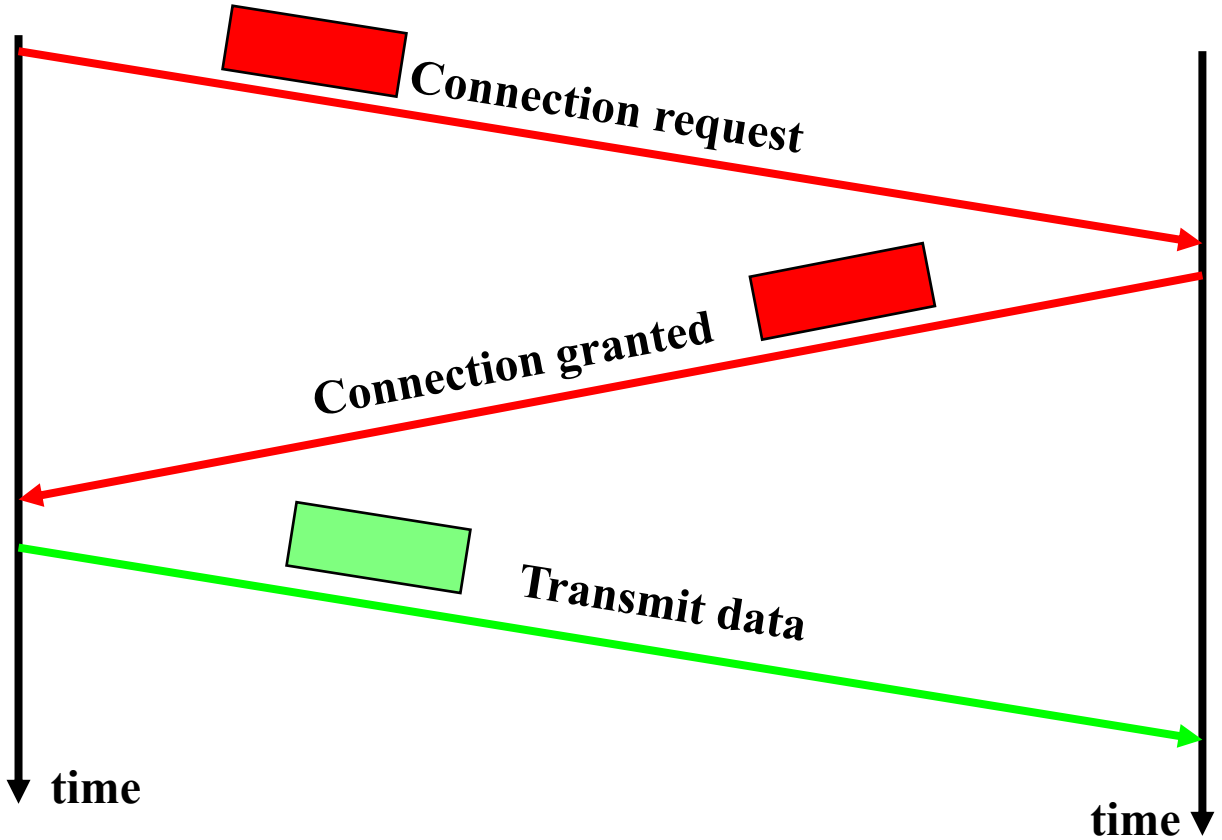
2-way handshake:



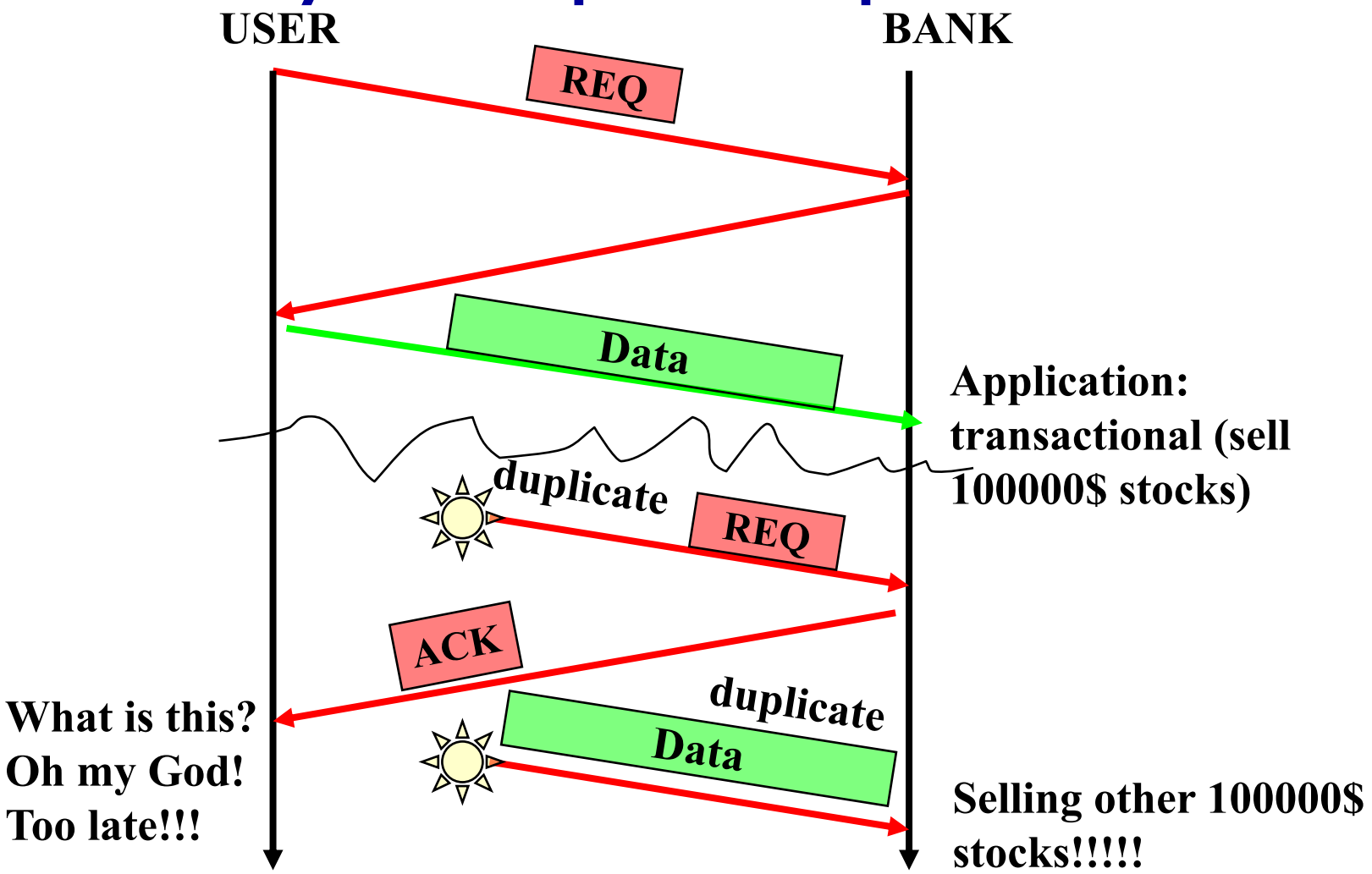
Q: will 2-way handshake always work in network?

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

Connection establishment: simplest approach (not TCP)

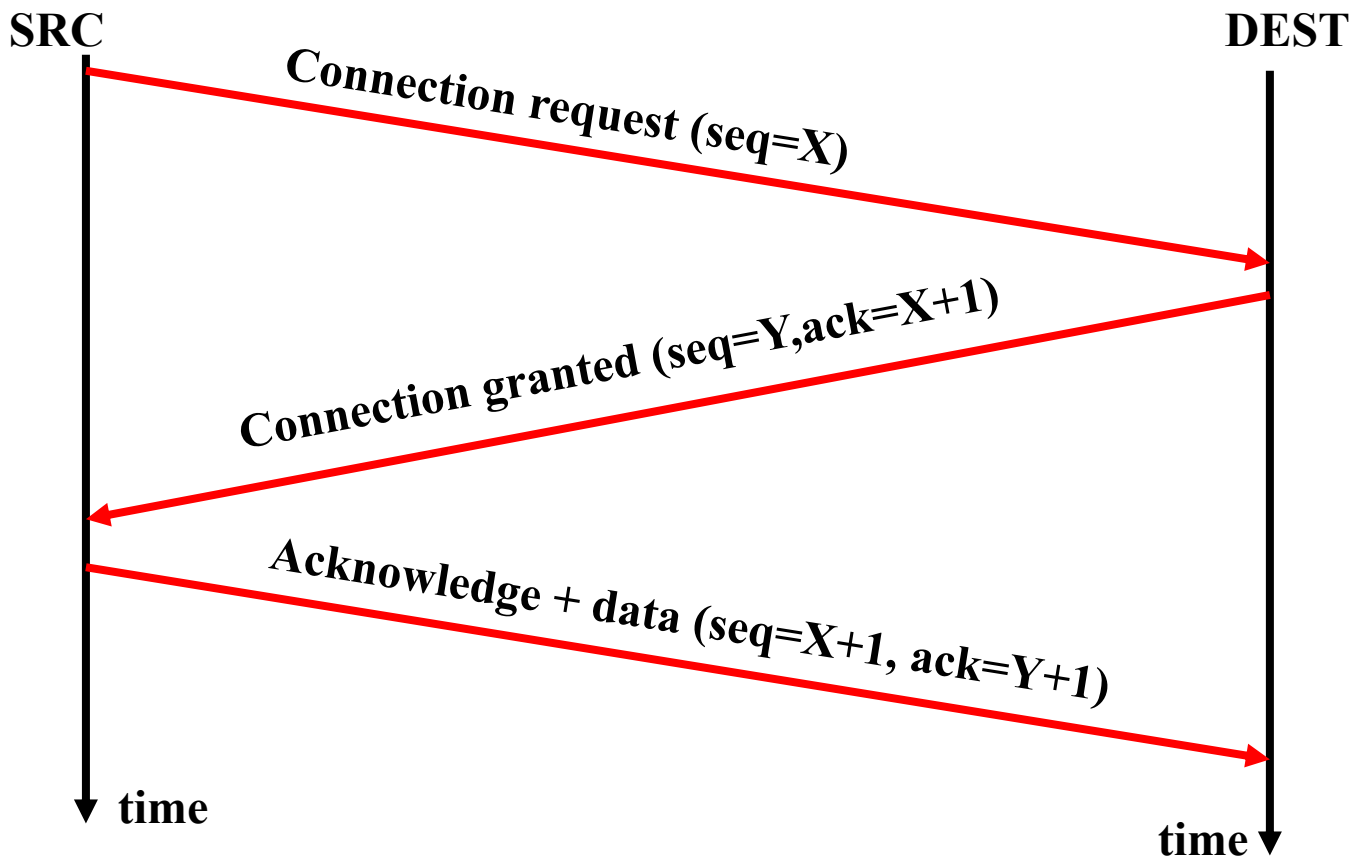


Delayed duplicate problem

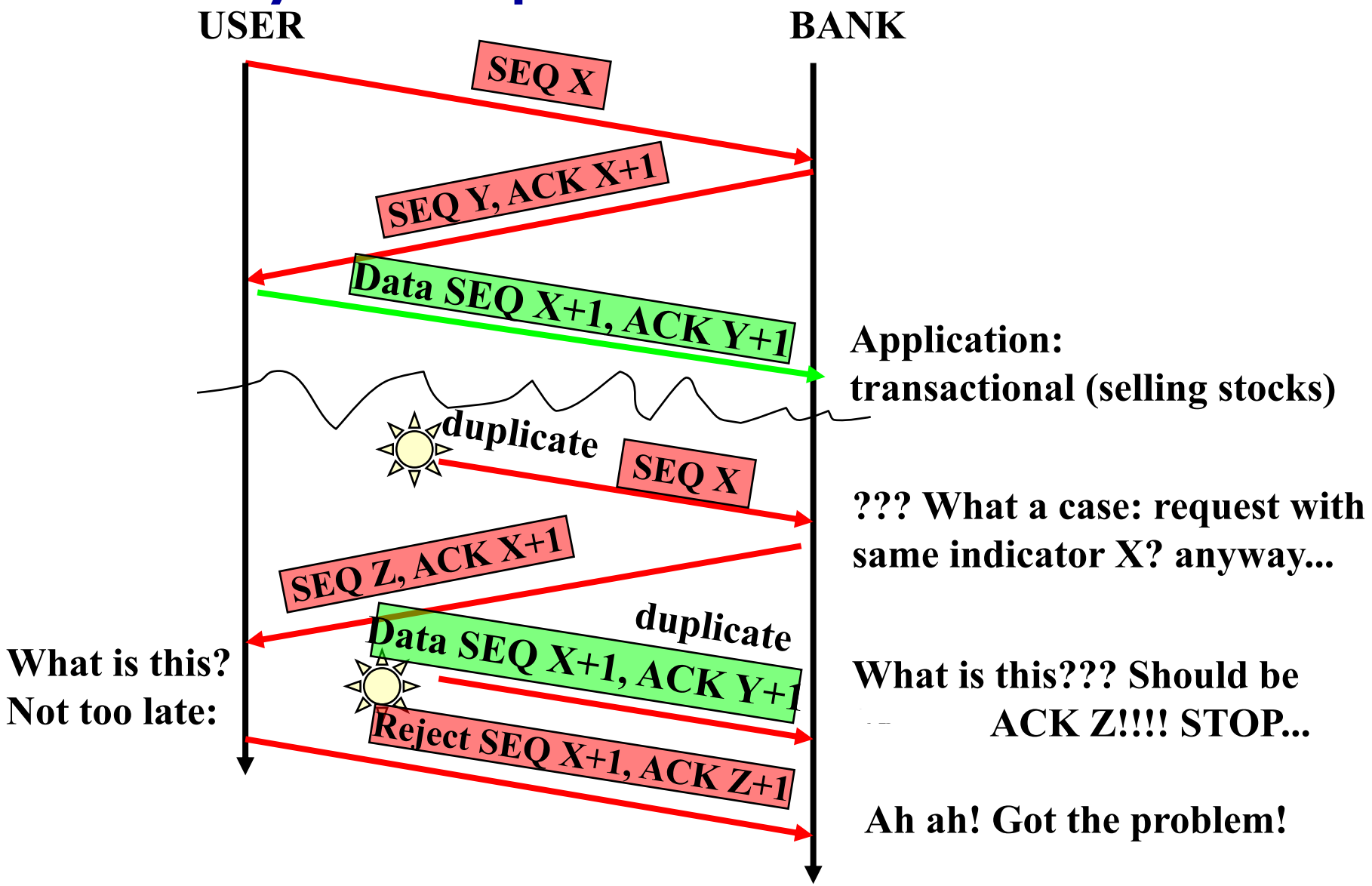


Solution: three way handshake

Tomlinson 1975

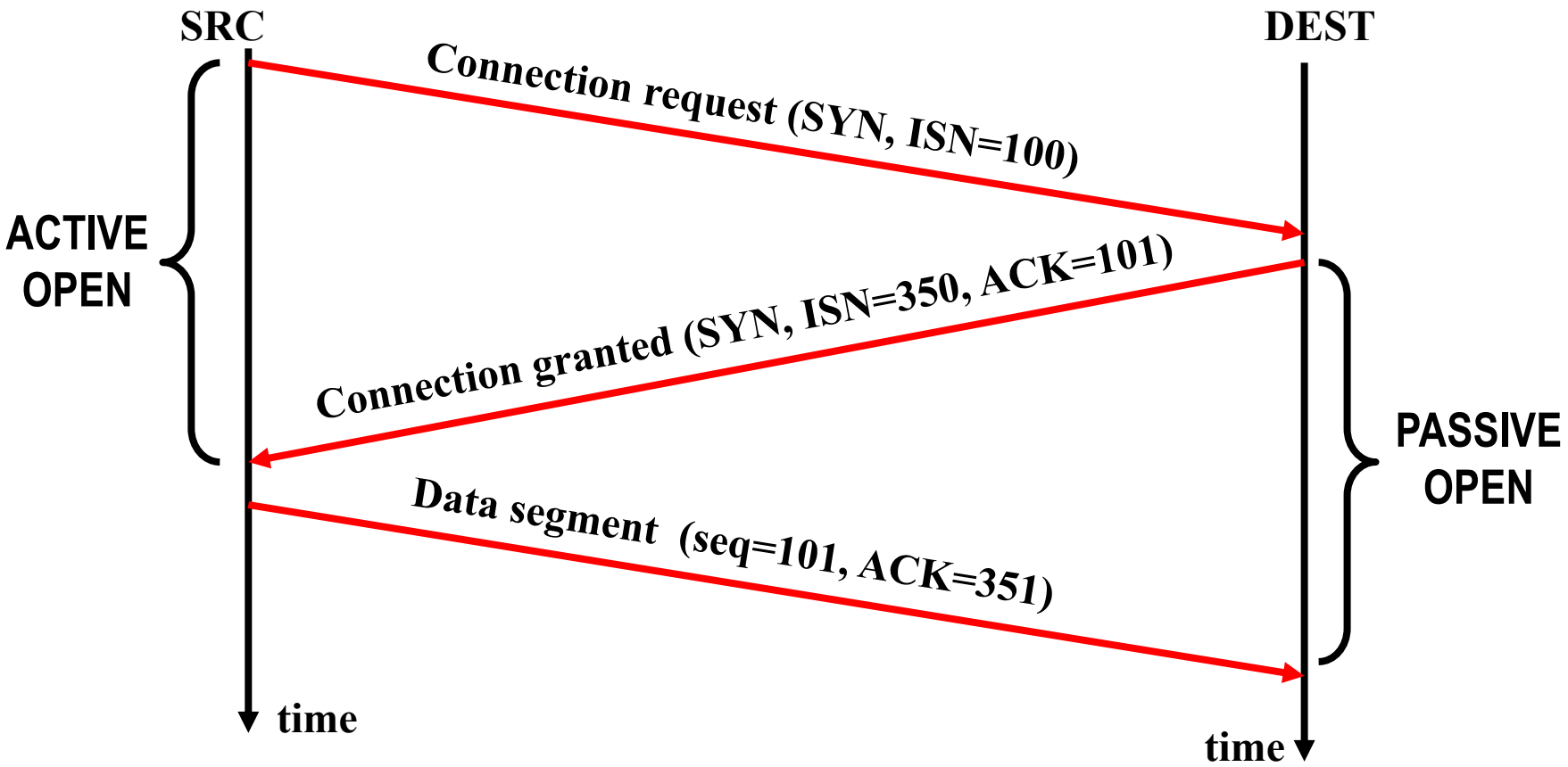


Delayed duplicate detection



Disaster could not be avoided with a two-way handshake

Three way handshake in TCP



Full *duplex connection: opened in both ways*

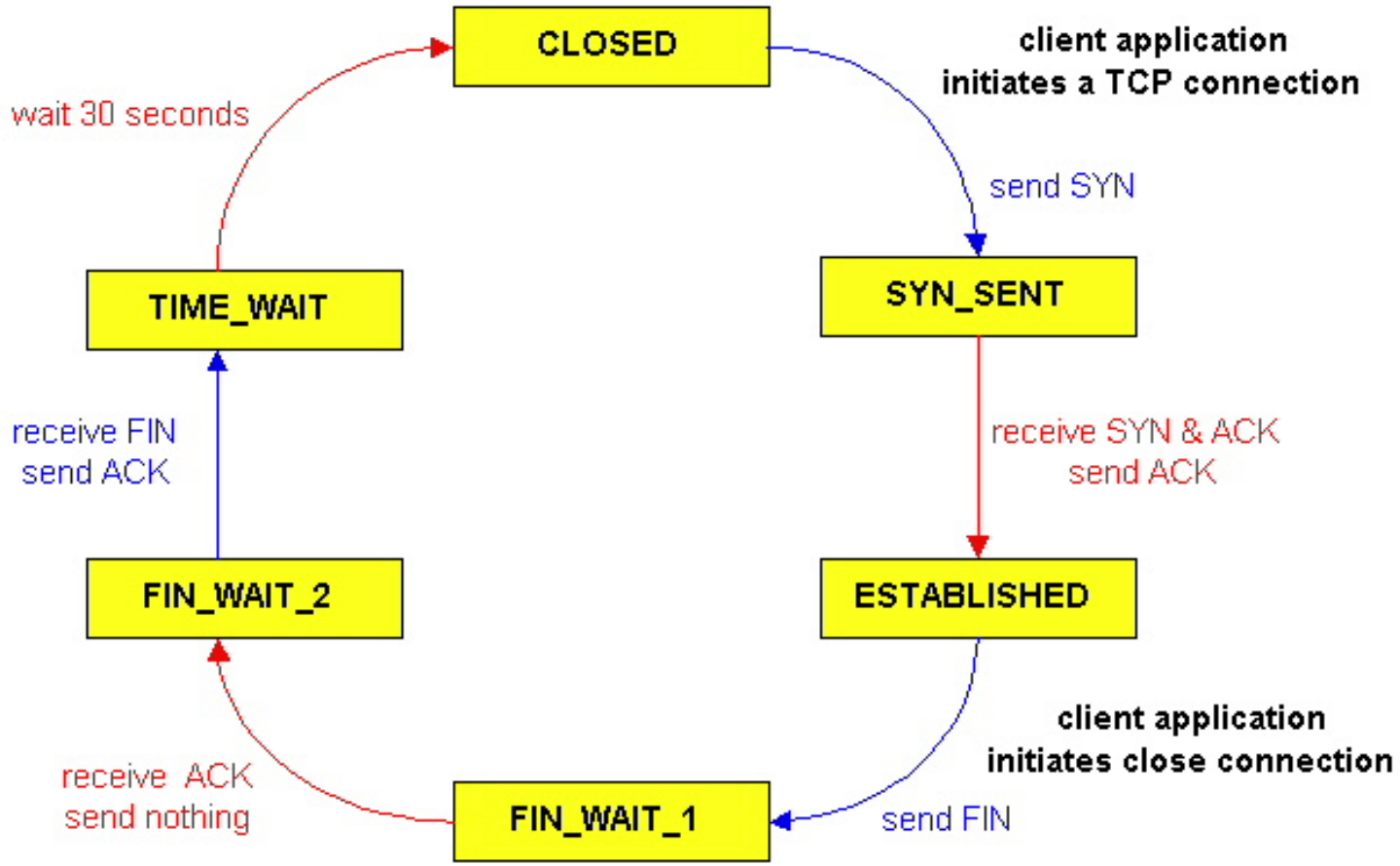
SRC: performs ACTIVE OPEN

DEST: Performs PASSIVE OPEN

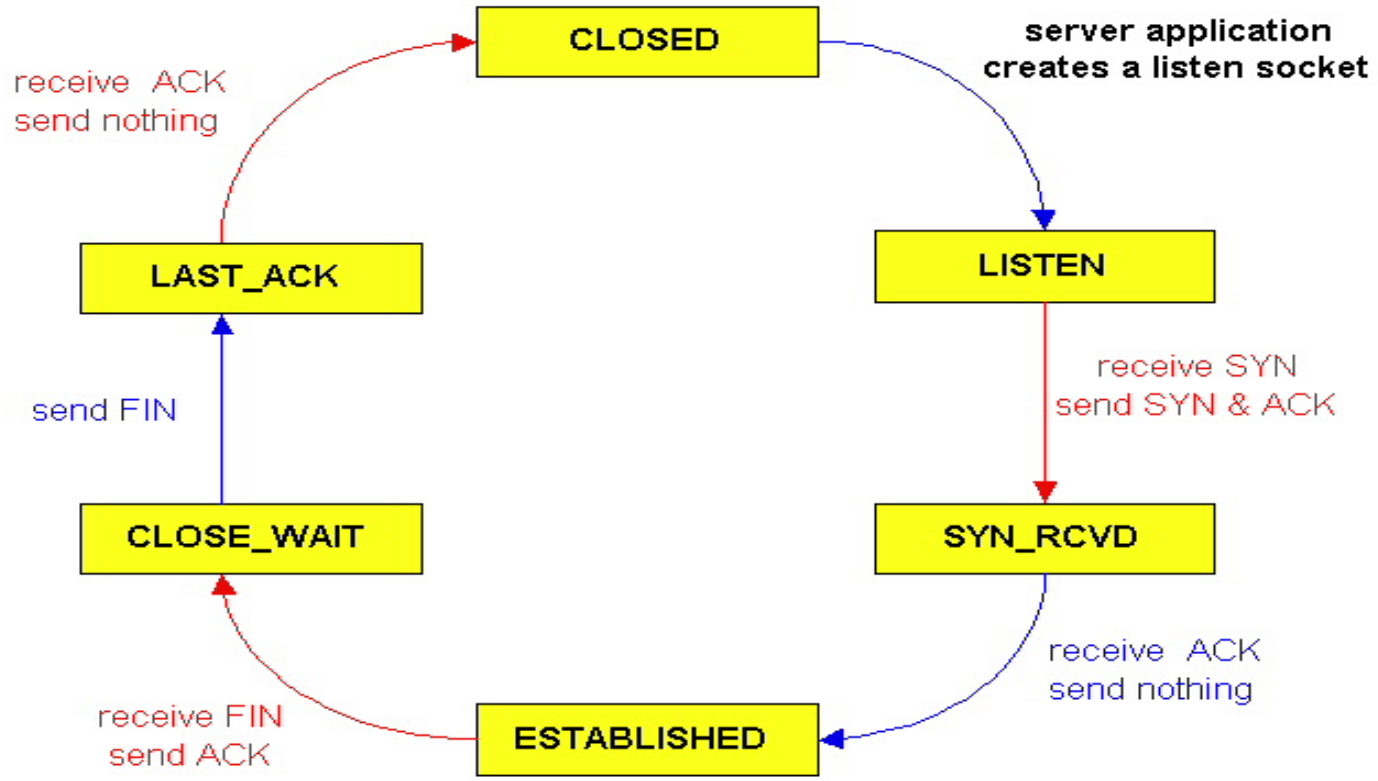
Initial Sequence Number

- Should change in time
  - RFC 793 (but not all implementations are conforming) suggests to generate ISN as a sample of a 32 bit counter incrementing at 4μs rate (4.55 hour to wrap around—Maximum Segment Lifetime much shorter)
- transmitted whenever SYN (Synchronize sequence numbers) flag active
  - note that both src and dest transmit THEIR initial sequence number (remember: full duplex)
- Data Bytes numbered from ISN+1
  - necessary to allow SYN segment ack

Connection states - Client



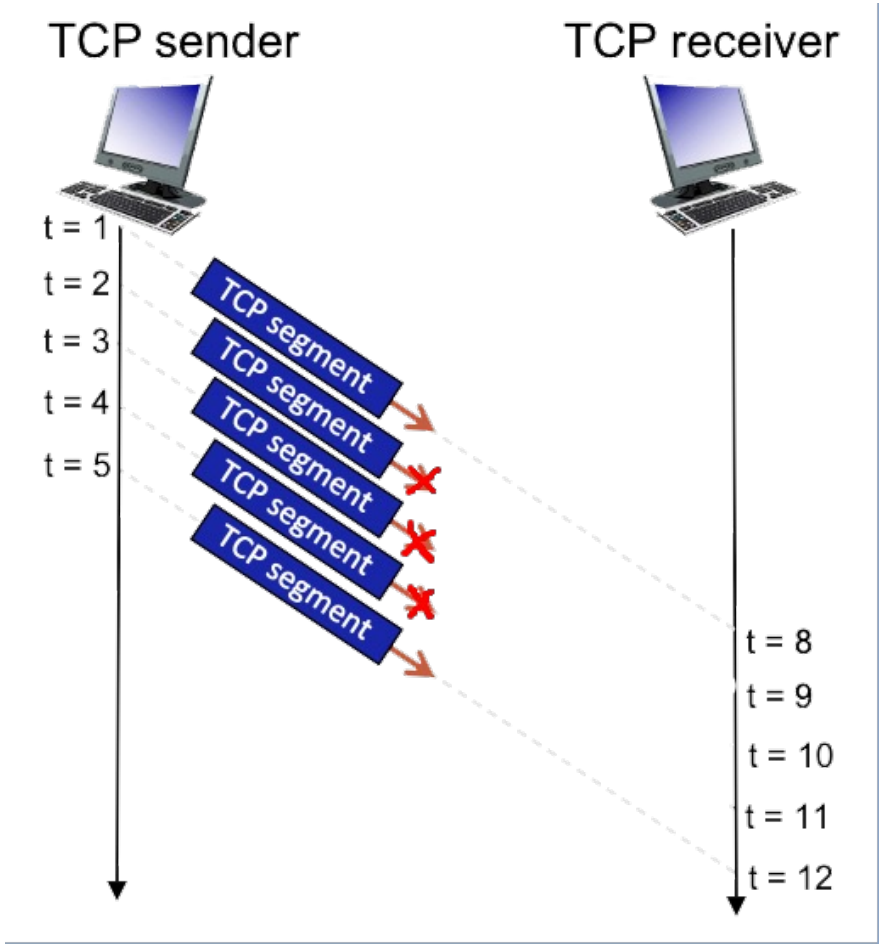
Connection States - Server



Closing a TCP 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

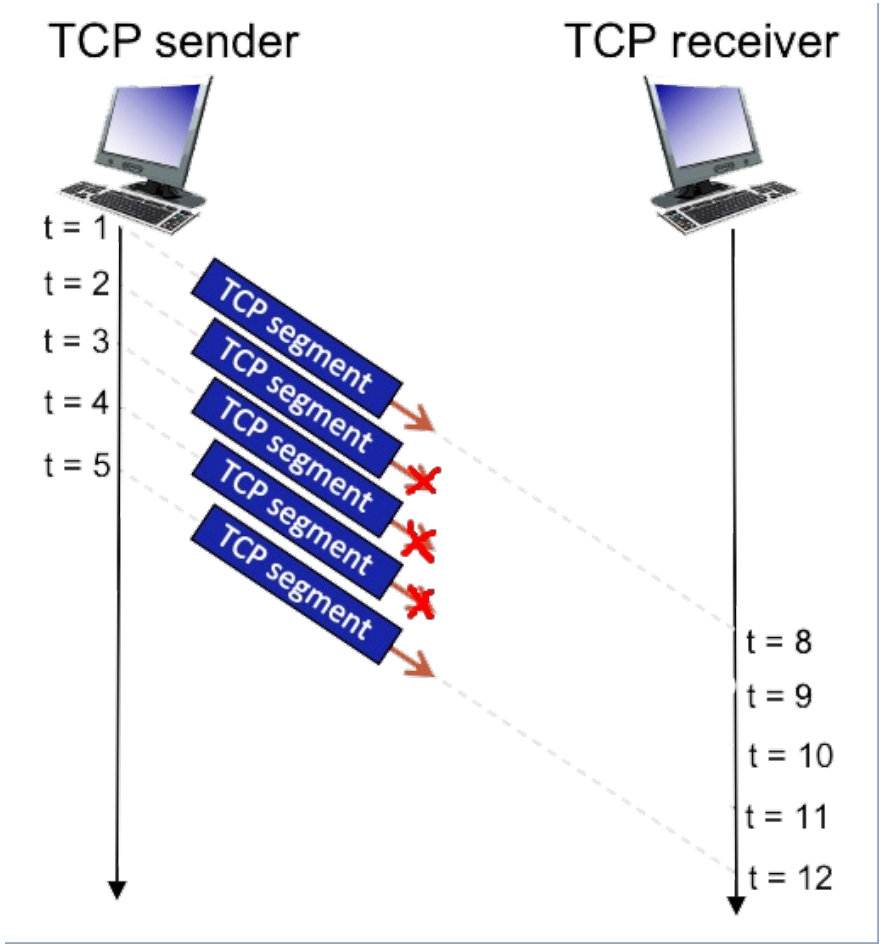
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 461 and the first 5 segments *each* contain 730 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

1) Give the sequence numbers provided by the sender in the five packets:

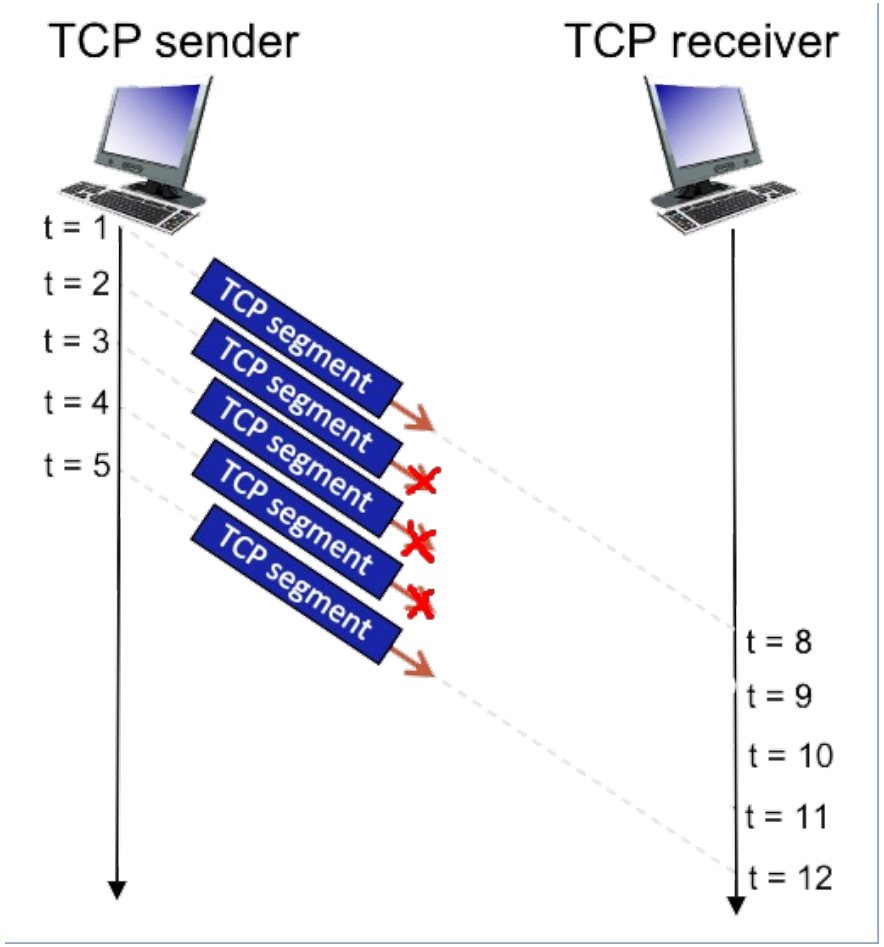
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 461 and the first 5 segments *each* contain 730 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

1) Give the sequence numbers provided by the sender in the five packets:
461, 1191, 1921, 2651, 3381

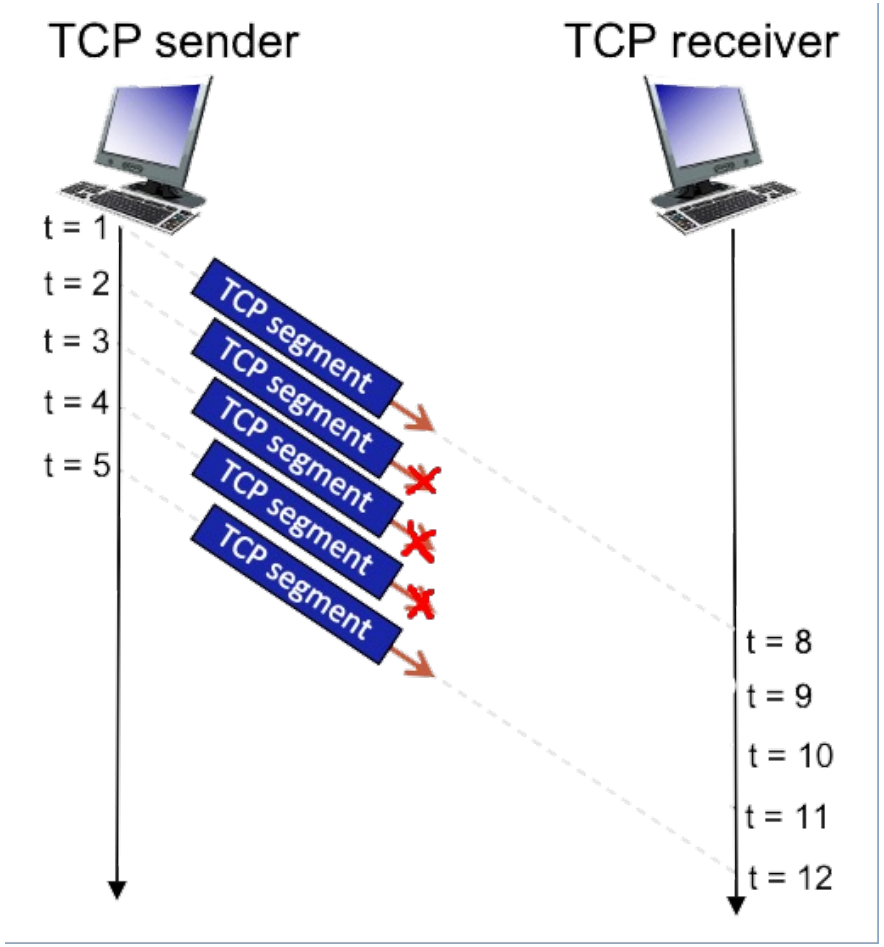
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 461 and the first 5 segments *each* contain 730 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

2) Give the ACK numbers the receiver sends in response to each of the segments.
If a segment never arrives use 'x' to denote it,
and format your answer as: a,b,c,...

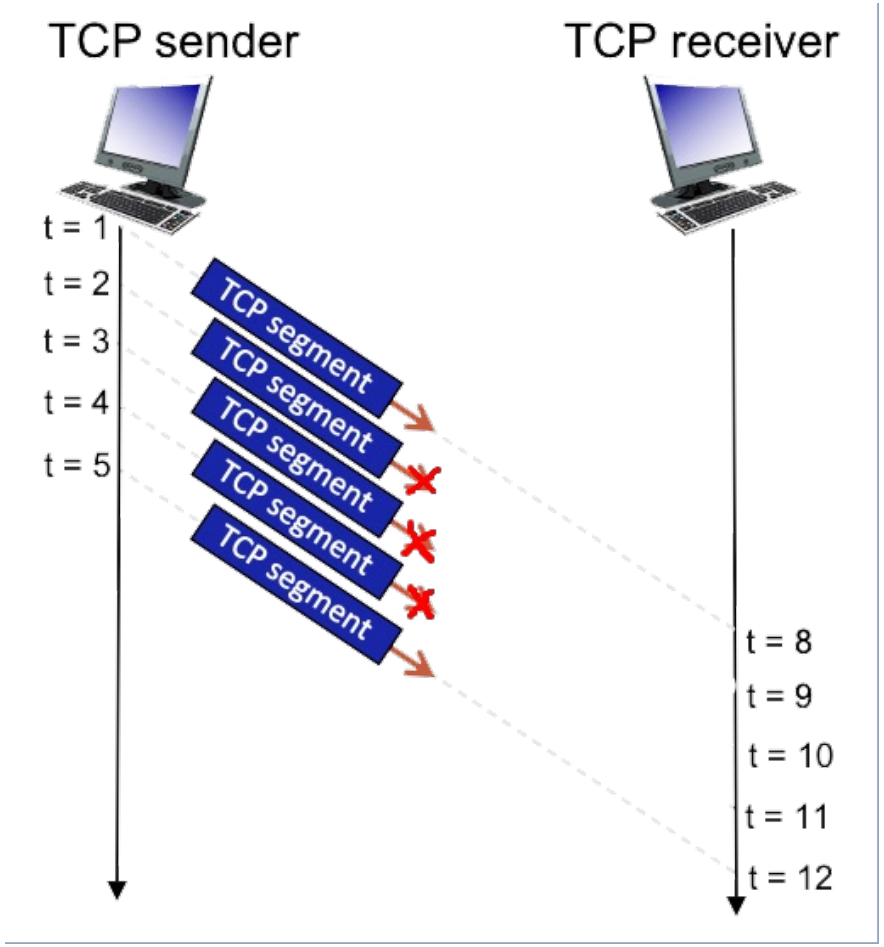
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 461 and the first 5 segments *each* contain 730 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

2) Give the ACK numbers the receiver sends in response to each of the segments.
If a segment never arrives use 'x' to denote it,
and format your answer as: a,b,c,...
1191,x,x,x,1191

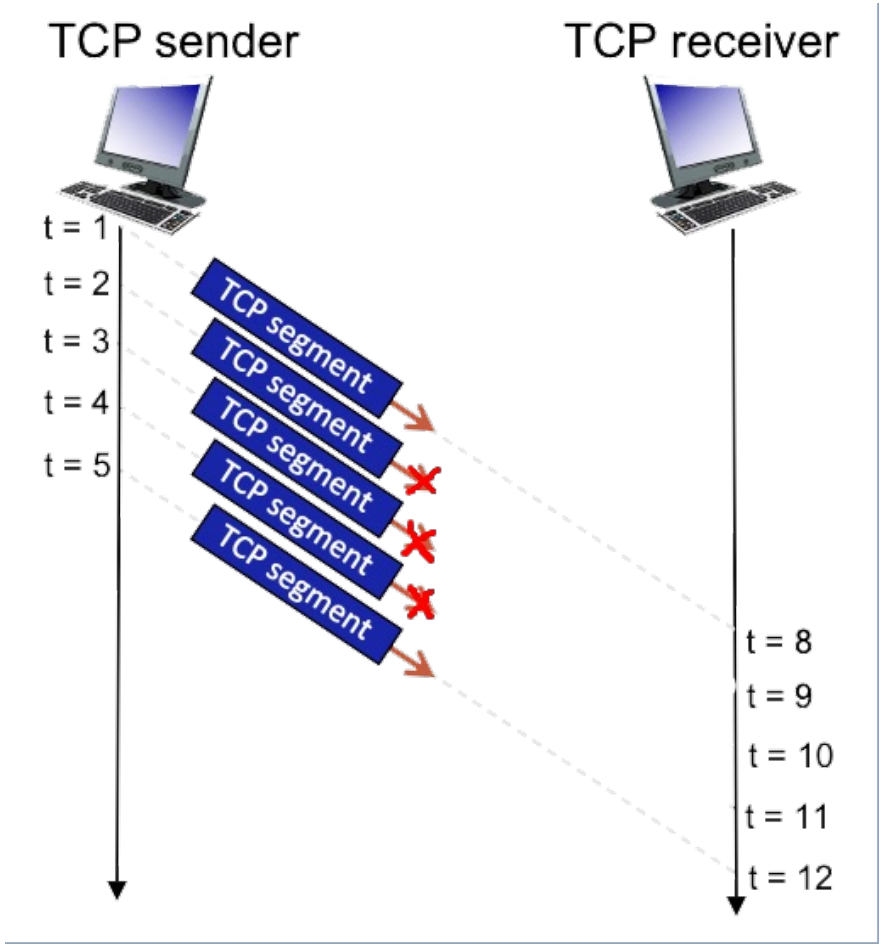
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 309 and the first 5 segments *each* contain 953 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

1) Give the sequence numbers provided by the sender in the five packets:

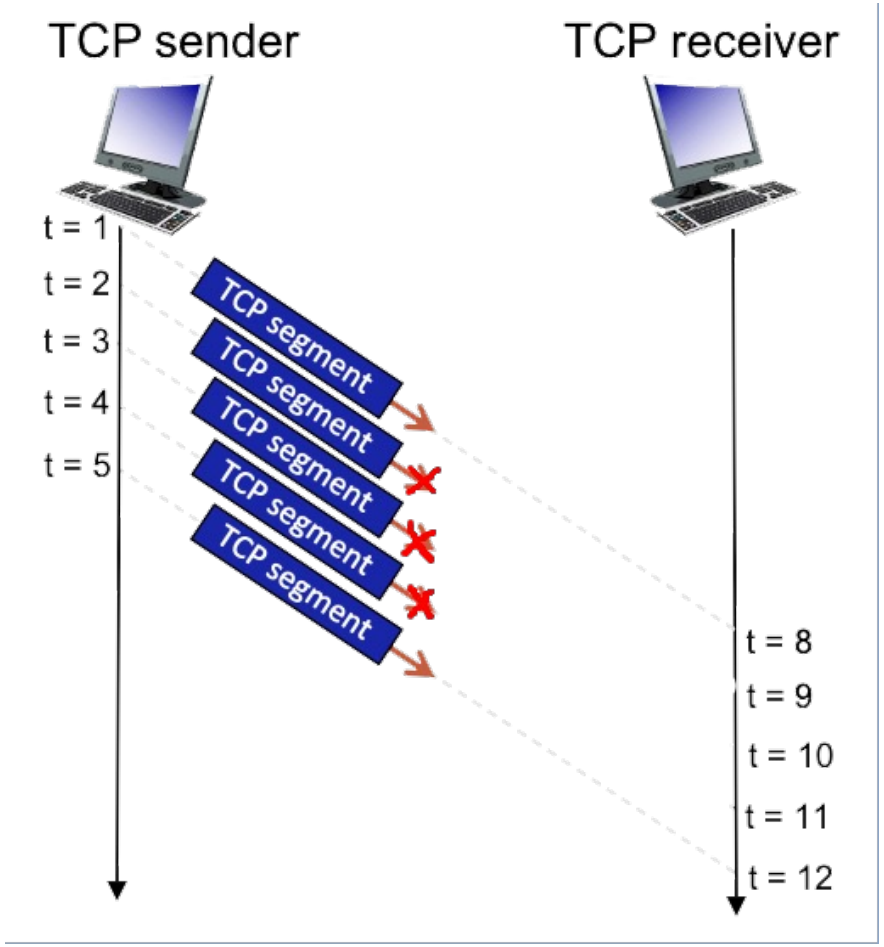
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 309 and the first 5 segments *each* contain 953 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

1) Give the sequence numbers provided by the sender in the five packets:
309,1262,2215,3168,4121

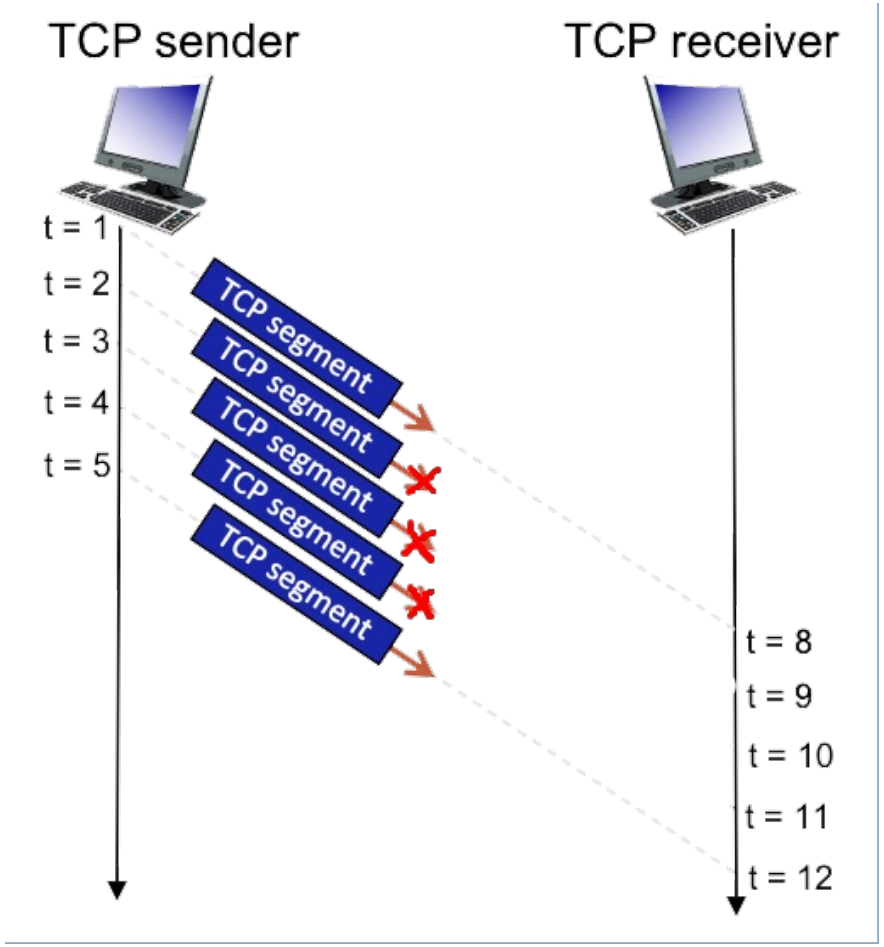
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 309 and the first 5 segments *each* contain 953 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

2) Give the ACK numbers the receiver sends in response to each of the segments.
If a segment never arrives use 'x' to denote it,
and format your answer as: a,b,c,...

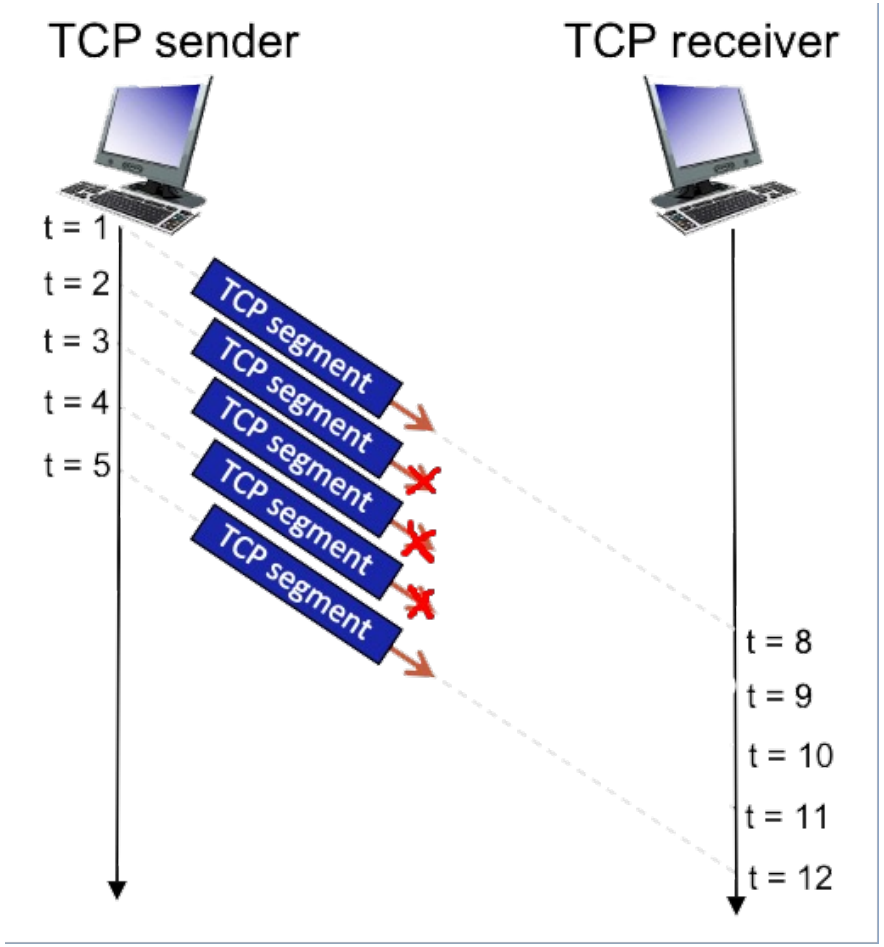
TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



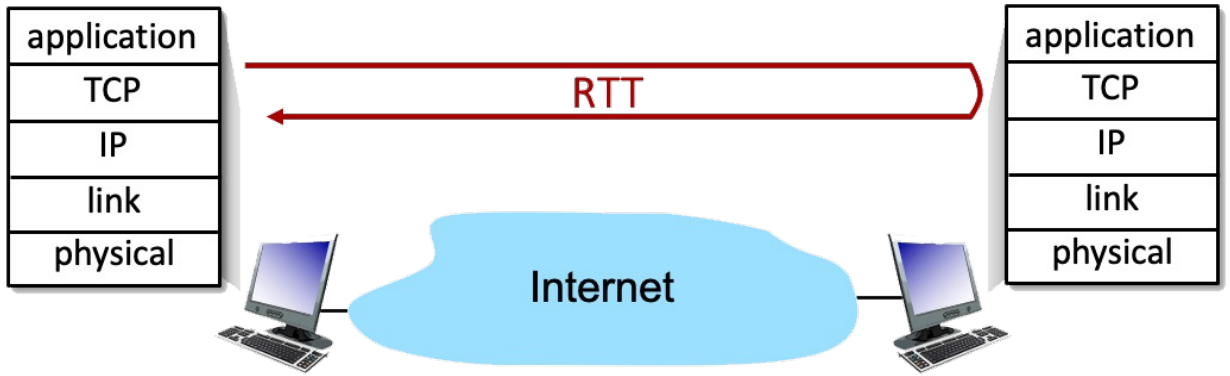
The TCP sender sends an initial window of 5 segments. Suppose the initial value of the sender->receiver sequence number is 309 and the first 5 segments *each* contain 953 bytes. The delay between the sender and receiver is 7 time units, and so the first segment arrives at the receiver at t=8.
As shown in the figure below, 3 of the 5 segment(s) are lost between the sender and receiver.

2) Give the ACK numbers the receiver sends in response to each of the segments.
If a segment never arrives use 'x' to denote it,
and format your answer as: a,b,c,...
1262,x,x,x,1262

TCP SEQUENCE AND ACK NUMBERS WITH SEGMENT LOSS



COMPUTING TCP'S RTT AND TIMEOUT VALUES



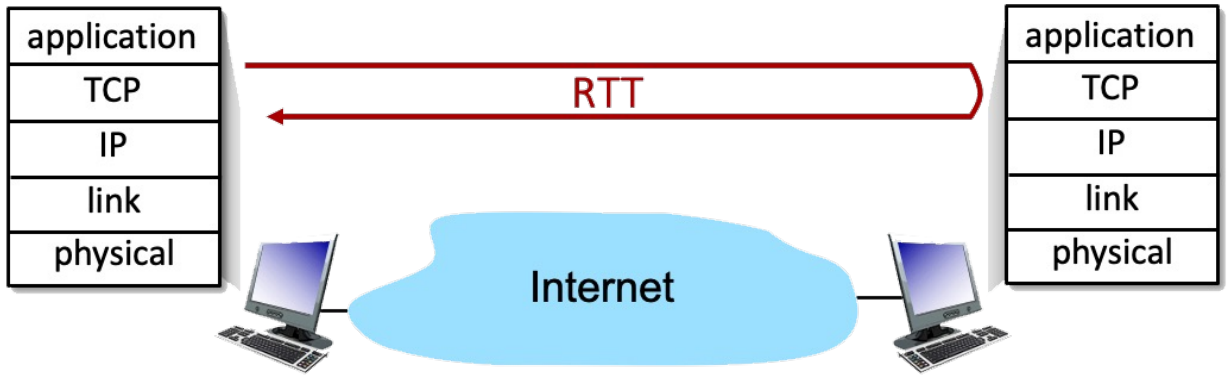
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

1. What is the estimatedRTT after the first RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	?					
350,000						
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

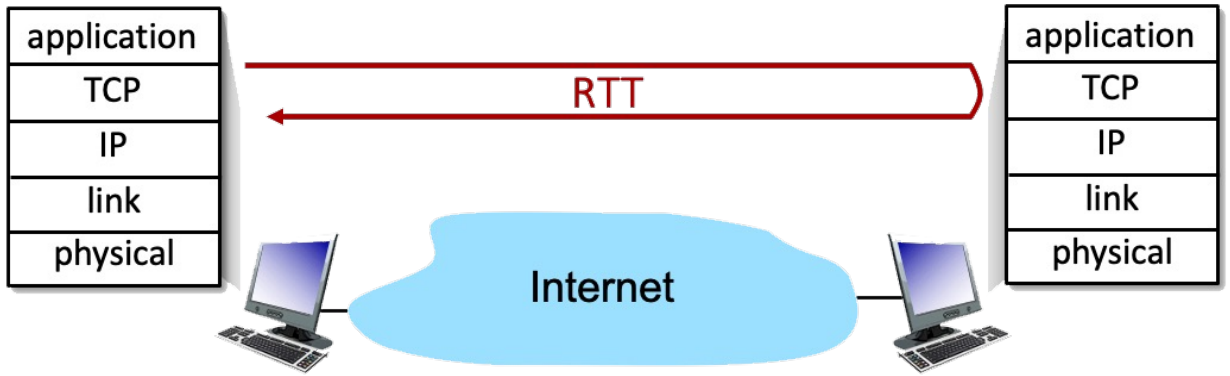
Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

1. What is the estimatedRTT after the first RTT?

estimatedRTT =0.125*310+0.875*400 =388.75

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	**338,750**					
350,000						
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



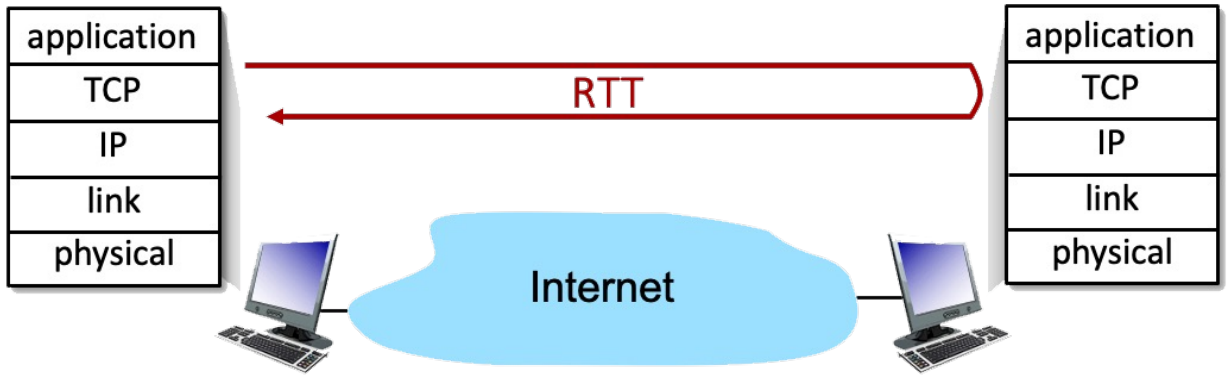
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of $\alpha = 0.125$, and $\beta = 0.25$. Round your answers to two decimal places after leading zeros.

2. What is the DevRTT after the first RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	?				
350,000						
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

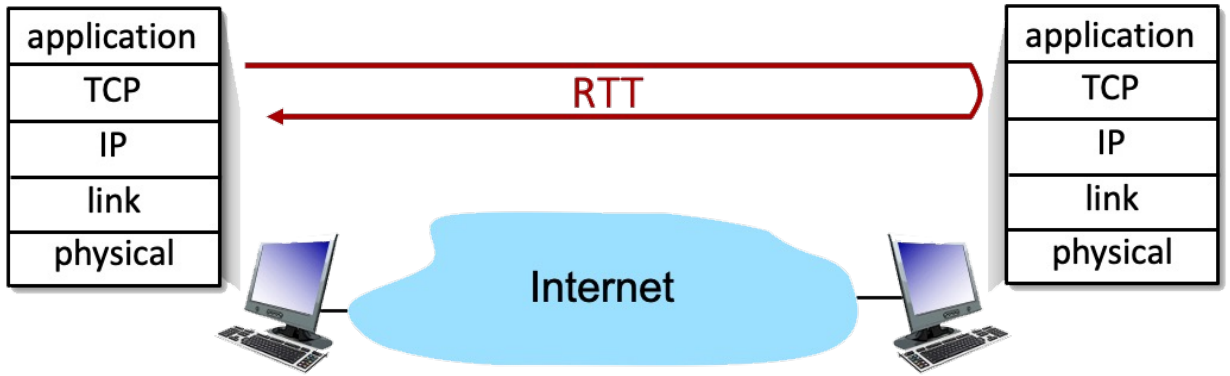
Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

2. What is the DevRTT after the first RTT?

DevRTT = 0.25*90 + 0.75* 15 = 33.75

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	**33,750**				
350,000						
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



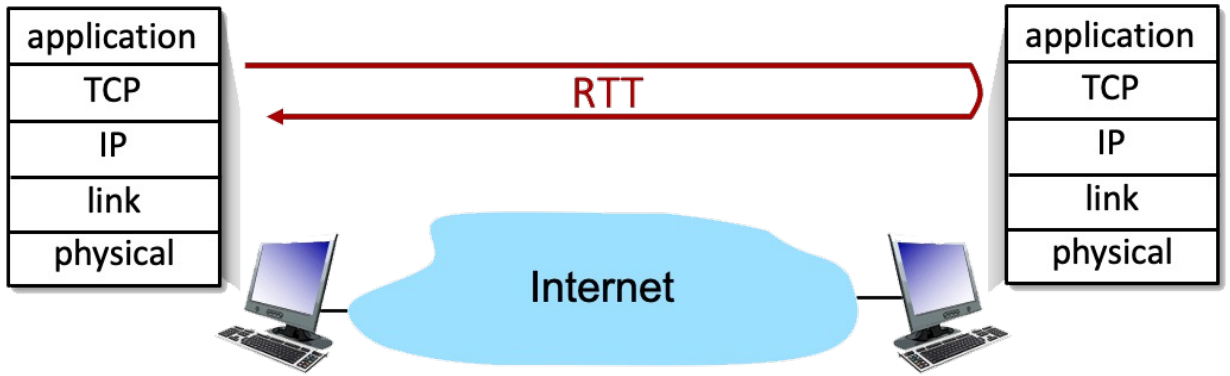
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of $\alpha = 0.125$, and $\beta = 0.25$. Round your answers to two decimal places after leading zeros.

3. What is the TCP timeout for the first RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	?			
350,000						
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

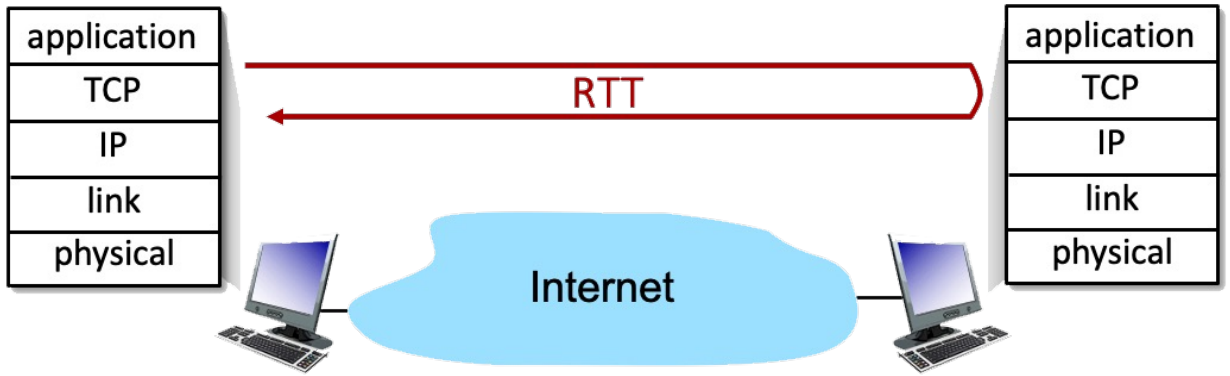
Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

3. What is the TCP timeout for the first RTT?

RTO = 388.75+4*33.75=523.75

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	**523,750**			
350,000						
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



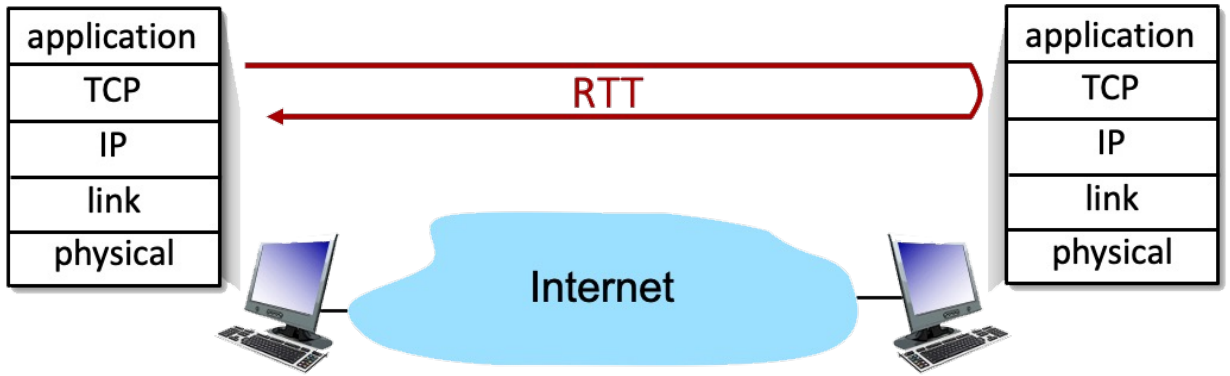
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

4. What is the estimatedRTT after the second RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	?					
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



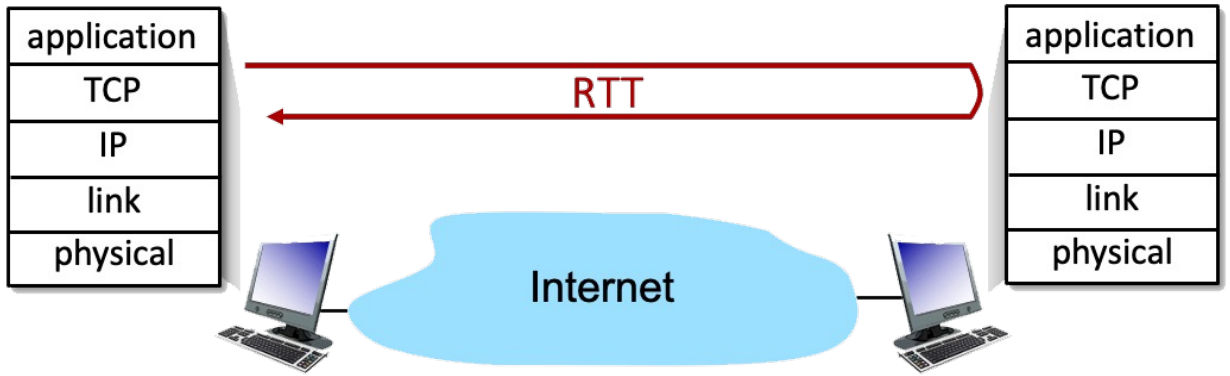
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

4. What is the estimatedRTT after the second RTT?
estimatedRTT = 388.75*0.875+ 350*0.125=383.91

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	**383,906**					
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



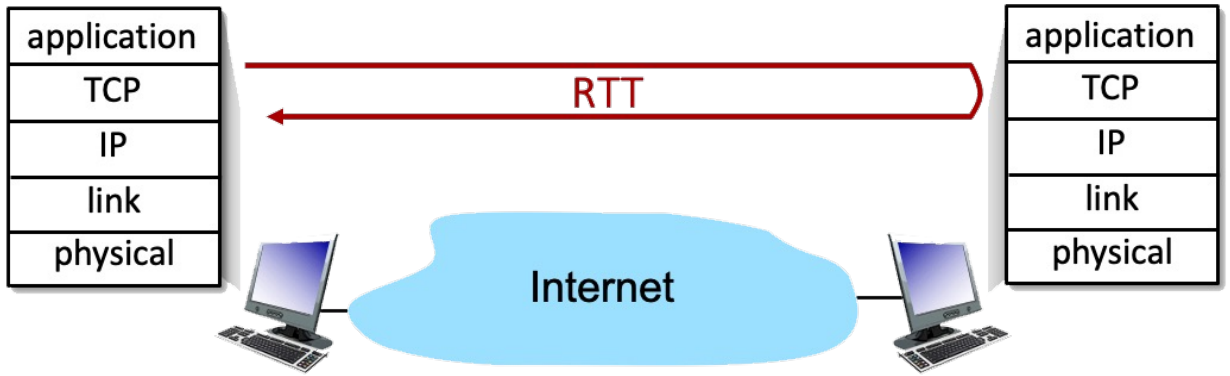
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of $\alpha = 0.125$, and $\beta = 0.25$. Round your answers to two decimal places after leading zeros.

5. What is the DevRTT after the second RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	?				
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



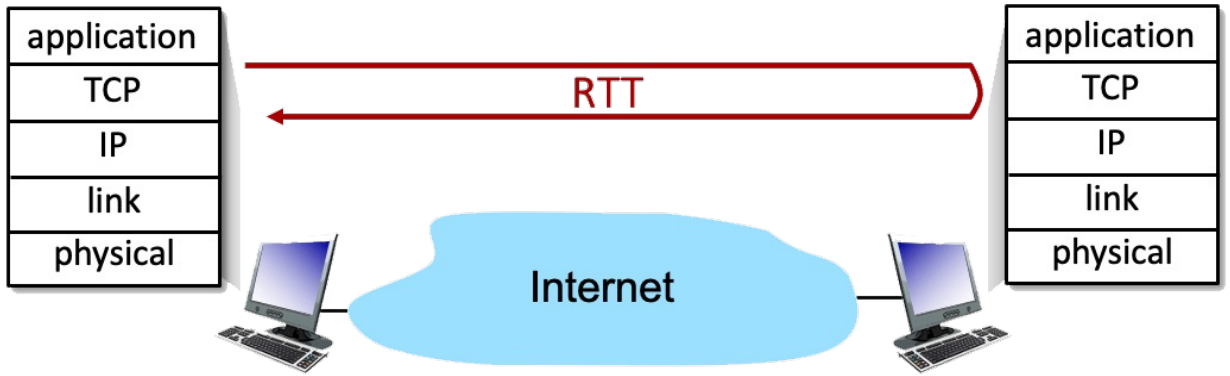
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

5. What is the DevRTT after the second RTT?
DevRTT = 0.75 * 33.75 + 0.25 * 38.75 =35

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	**35**				
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



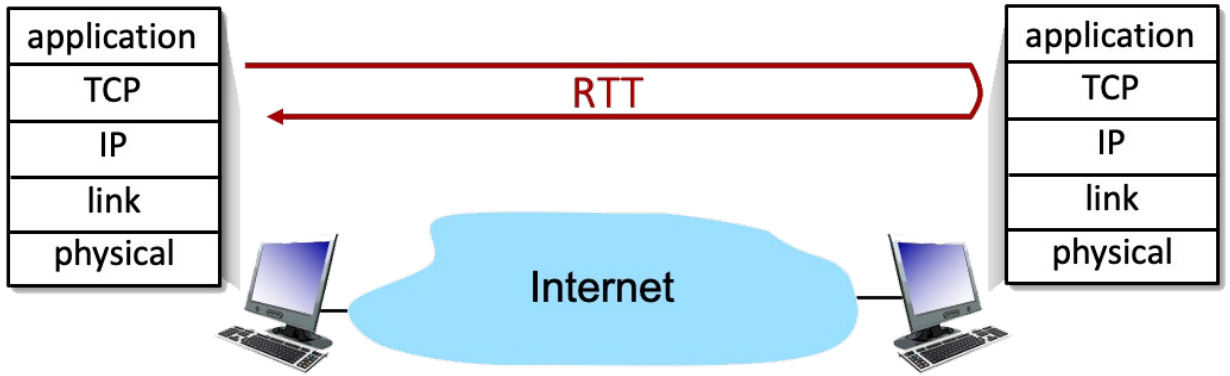
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of $\alpha = 0.125$, and $\beta = 0.25$. Round your answers to two decimal places after leading zeros.

6. What is the TCP timeout for the second RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	?			
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

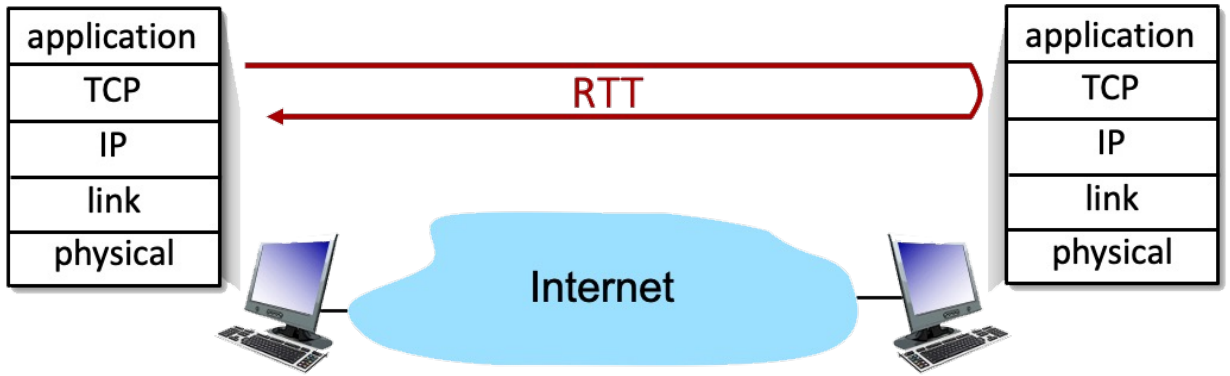
Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

6. What is the TCP timeout for the second RTT?

RTO = 383.91+4*35=523.91

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	**523,906**			
270,000						

COMPUTING TCP'S RTT AND TIMEOUT VALUES



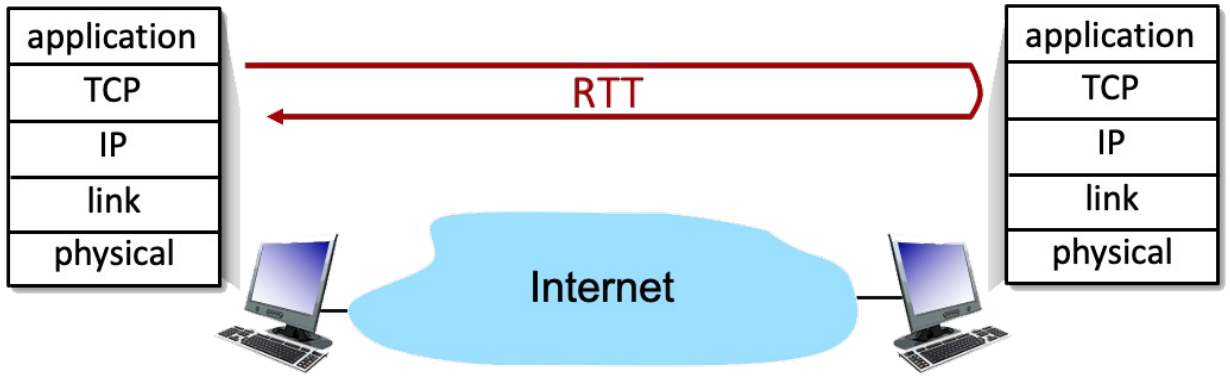
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of $\alpha = 0.125$, and $\beta = 0.25$. Round your answers to two decimal places after leading zeros.

7. What is the estimatedRTT after the third RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	523,906			
270,000	?					

COMPUTING TCP'S RTT AND TIMEOUT VALUES



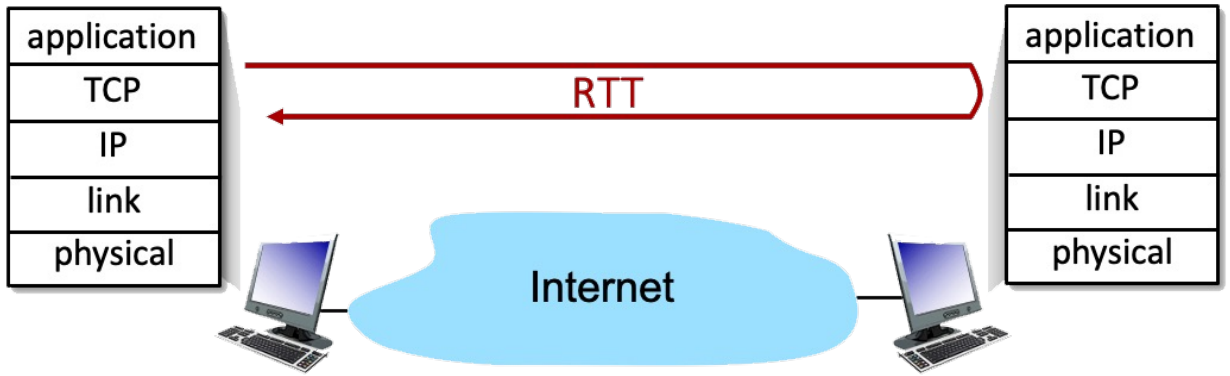
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

7. What is the estimatedRTT after the third RTT?
estimatedRTT = 383.91*0.875+ 270*0.125=369.67

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	523,906			
270,000	**369,668**					

COMPUTING TCP'S RTT AND TIMEOUT VALUES



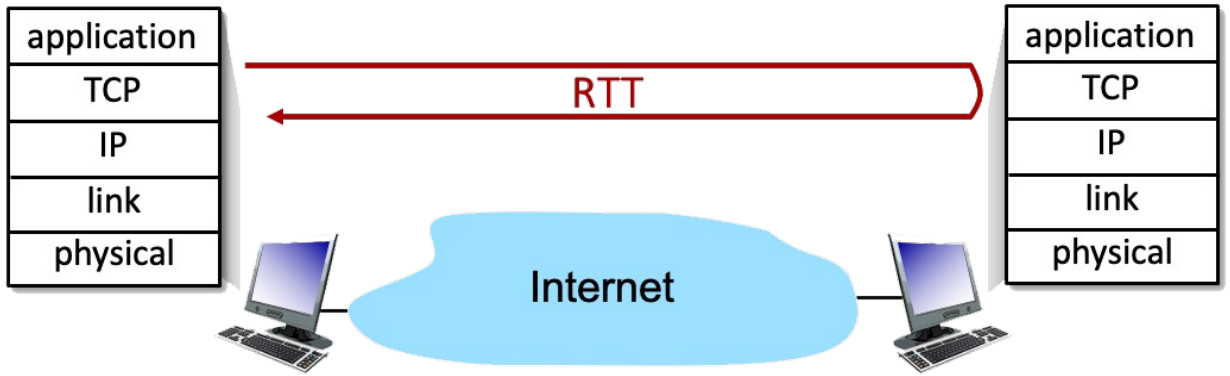
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

8. What is the DevRTT after the third RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	523,906			
270,000	369,668	?				

COMPUTING TCP'S RTT AND TIMEOUT VALUES



Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

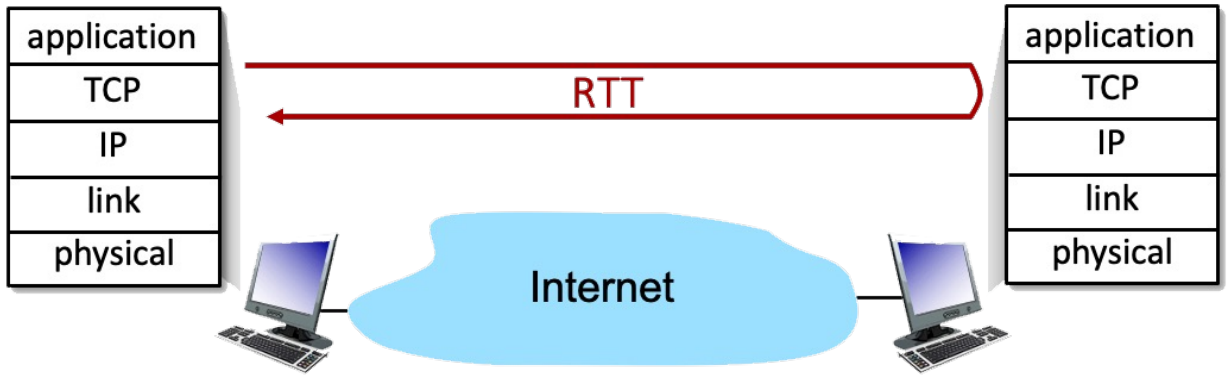
Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

8. What is the DevRTT after the third RTT?

DevRTT = 0.75 * 35 + 0.25 * |270-383.91| = 54,73

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	523,906			
270,000	369,668	**54,727**				

COMPUTING TCP'S RTT AND TIMEOUT VALUES



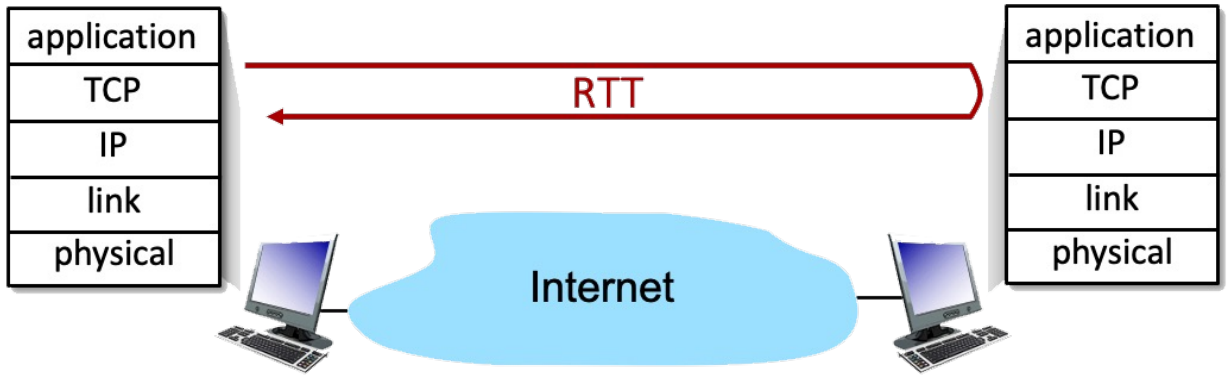
Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of $\alpha = 0.125$, and $\beta = 0.25$. Round your answers to two decimal places after leading zeros.

9. What is the TCP timeout for the third RTT?

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	523,906			
270,000	369,668	54,727	?			

COMPUTING TCP'S RTT AND TIMEOUT VALUES



Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 400 msec and 15 msec, respectively. Suppose that the next three measured values of the RTT are 310 msec, 350 msec, and 270 msec respectively.

Compute TCP's new value of *DevRTT, estimatedRTT,* and the TCP timeout value after each of these three measured RTT values is obtained. Use the values of α = 0.125, and β = 0.25. Round your answers to two decimal places after leading zeros.

9. What is the TCP timeout for the third RTT?

RTO = 369.67+4*54.73 = 588.57

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	400,000	15,000			0,125	0,250
310,000	338,750	33,750	523,750			
350,000	383,906	35	523,906			
270,000	369,668	54,727	**588,574**			

COMPUTING TCP'S RTT AND TIMEOUT VALUES

Suppose that TCP's current estimated values for the round trip time (*estimatedRTT*) and deviation in the RTT (*DevRTT*) are 220 msec and 26 msec, respectively. Suppose that the next three measured values of the RTT are 370 msec, 330 msec, and 200 msec respectively.

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

rtt(i)	RTT(i)	DevRTT(i)	RTO(i)		alpha	beta
	220,000	26,000			0,125	0,250
370,000	238,750	57,000	466,750			
330,000	250,156	65,563	512,406			
200,000	243,887	61,711	490,730			

P26. Consider transferring an enormous file of L bytes from Host A to Host B. Assume an MSS of 536 bytes.

a. What is the maximum value of L such that TCP sequence numbers are not exhausted? Recall that the TCP sequence number field has 4 bytes.

b. For the L you obtain in (a), find how long it takes to transmit the file. Assume that a total of 66 bytes of transport, network, and data-link header are added to each segment before the resulting packet is sent out over a 155 Mbps link. Ignore flow control and congestion control so A can pump out the segments back to back and continuously.

a. Recall that we have 4 bytes to represent a sequence number in TCP, hence there are 2^{32} possible seq. numbers associated to the transferred bytes. Then $L=2^{32}$ bytes of data.

b. Given the MSS of 536 bytes, the number of segments will be $n_{segments} = \left\lceil \frac{2^{32}}{536} \right\rceil$ = 8012999 segments.
Each segment will have 66 bytes of added header.
The number of bytes transferred due to headers will be $s_{headers}=66 \times n_{segments}$ = 528857934 bytes.

The total number of transmitted bytes will therefore be $L + s_{headers} = (2^{32} + 528857934)$ bytes = 4.824×10^9 bytes

[… P26 follows]

The total number of transmitted bytes will therefore be
$L + s_{headers} = (2^{32} + 528857934)$ bytes $= 4.824 \times 10^9$ bytes

Given the bandwidth R = 155 Mbps the time taken to transfer the given amount of bytes is

$T = (L + s_{headers})/R = 4.824 \times 10^9 \times 8$ bits / $(155 \times 10^6$ bit/sec) = 249 secs

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- **Principles of congestion control**
- TCP congestion control
- Evolution of transport-layer functionality

Principles of congestion control

Congestion:

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

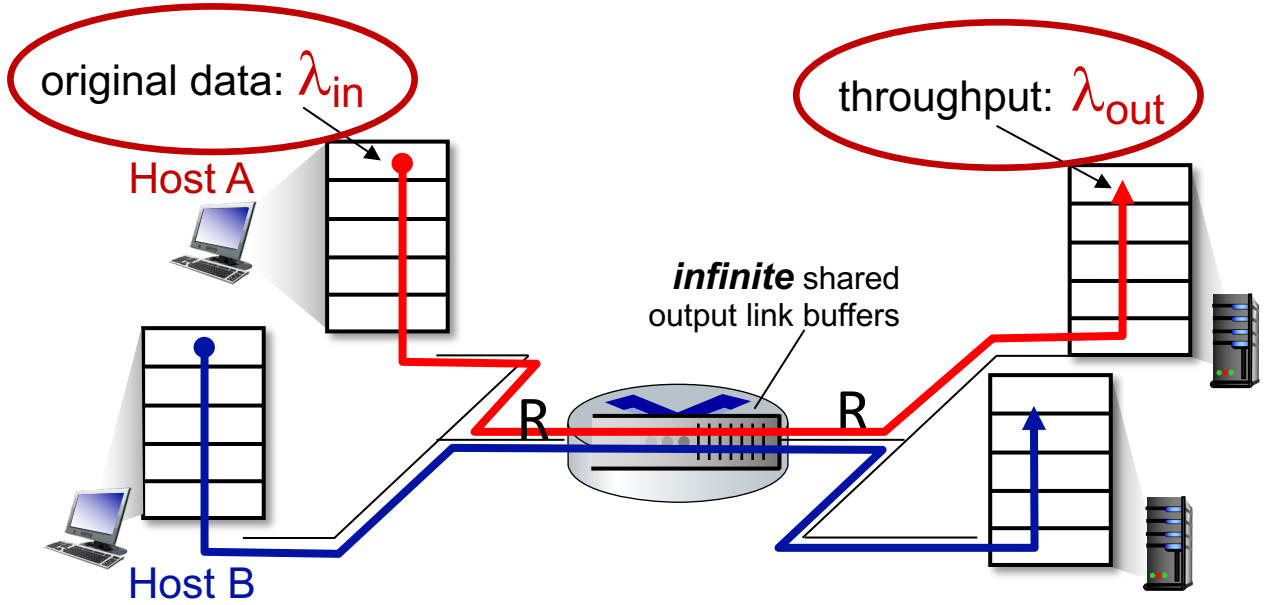
congestion control:
too many senders,
sending too fast

flow control: one sender
too fast for one receiver

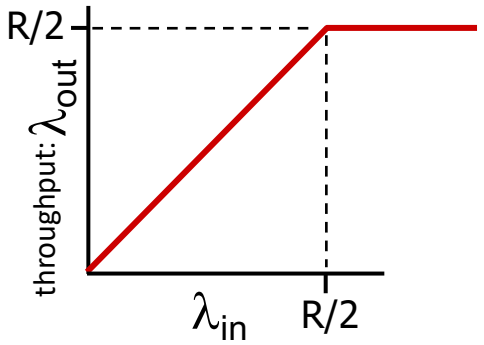
Causes/costs of congestion: scenario 1

Simplest scenario:

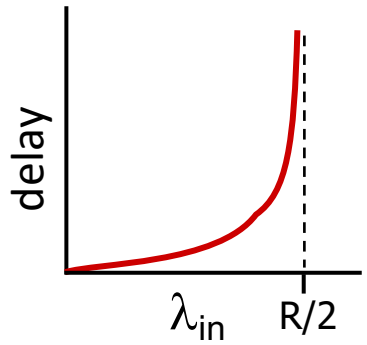
- one router, infinite buffers
- input, output link capacity: R
- two flows
- no retransmissions needed



Q: What happens as arrival rate λ_{in} approaches R/2?



maximum per-connection throughput: R/2

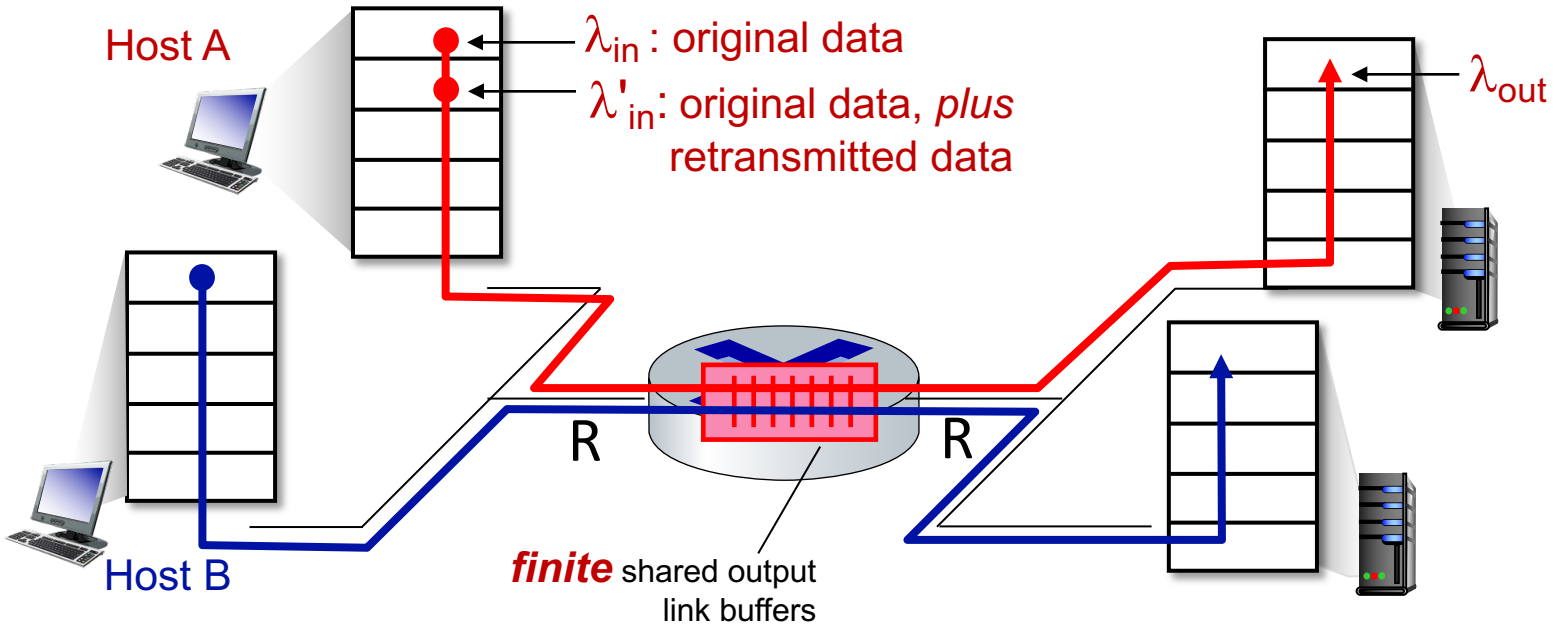


large delays as arrival rate λ_{in} approaches capacity

Causes/costs of congestion: scenario 2

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

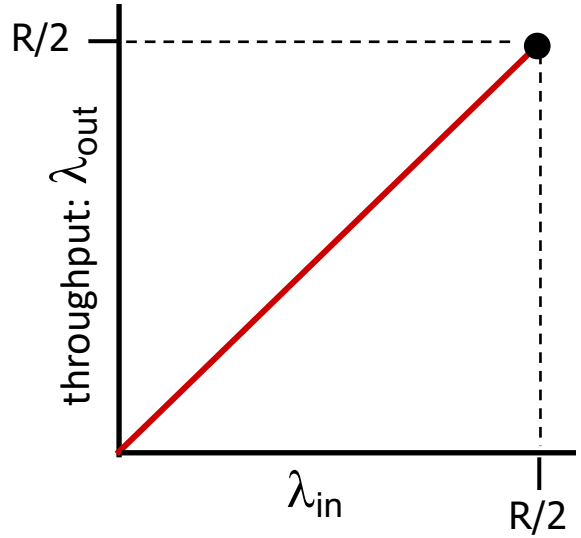
Two different sending rates at application and transport layer

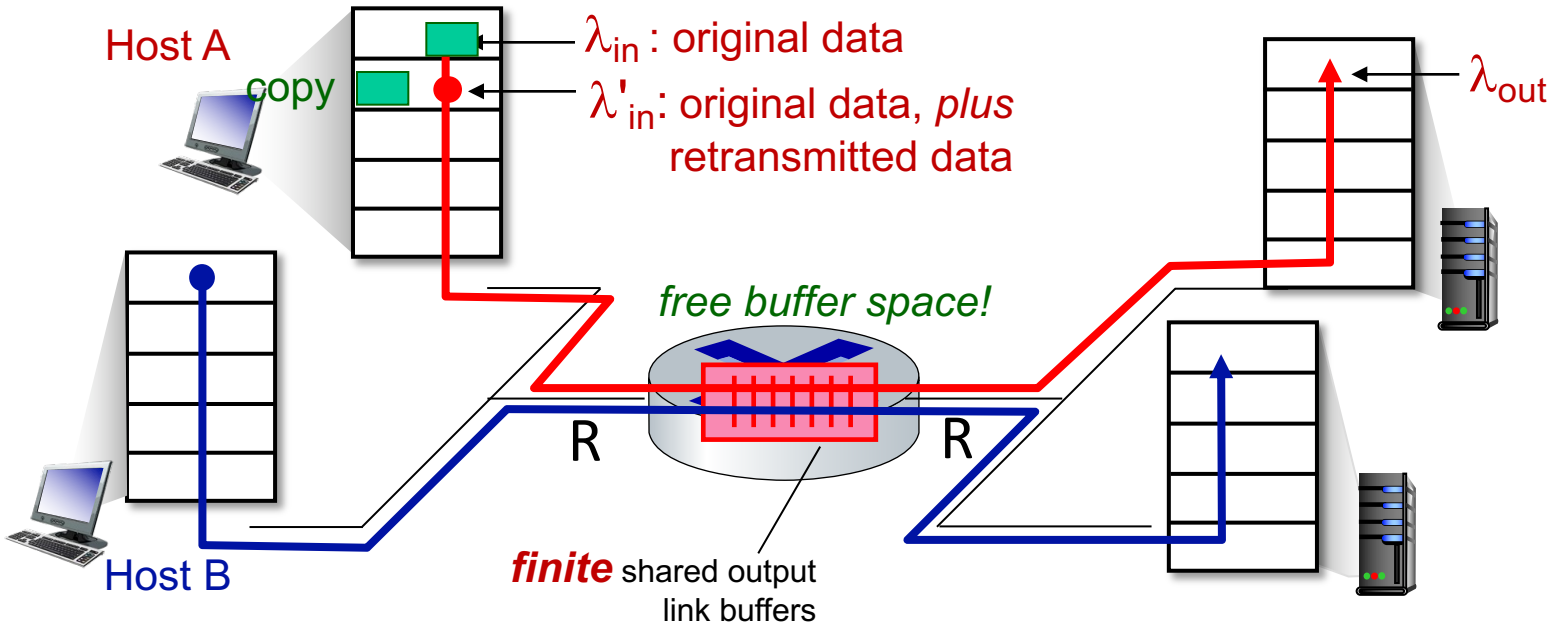


Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

- sender sends only when router buffers available (no loss)

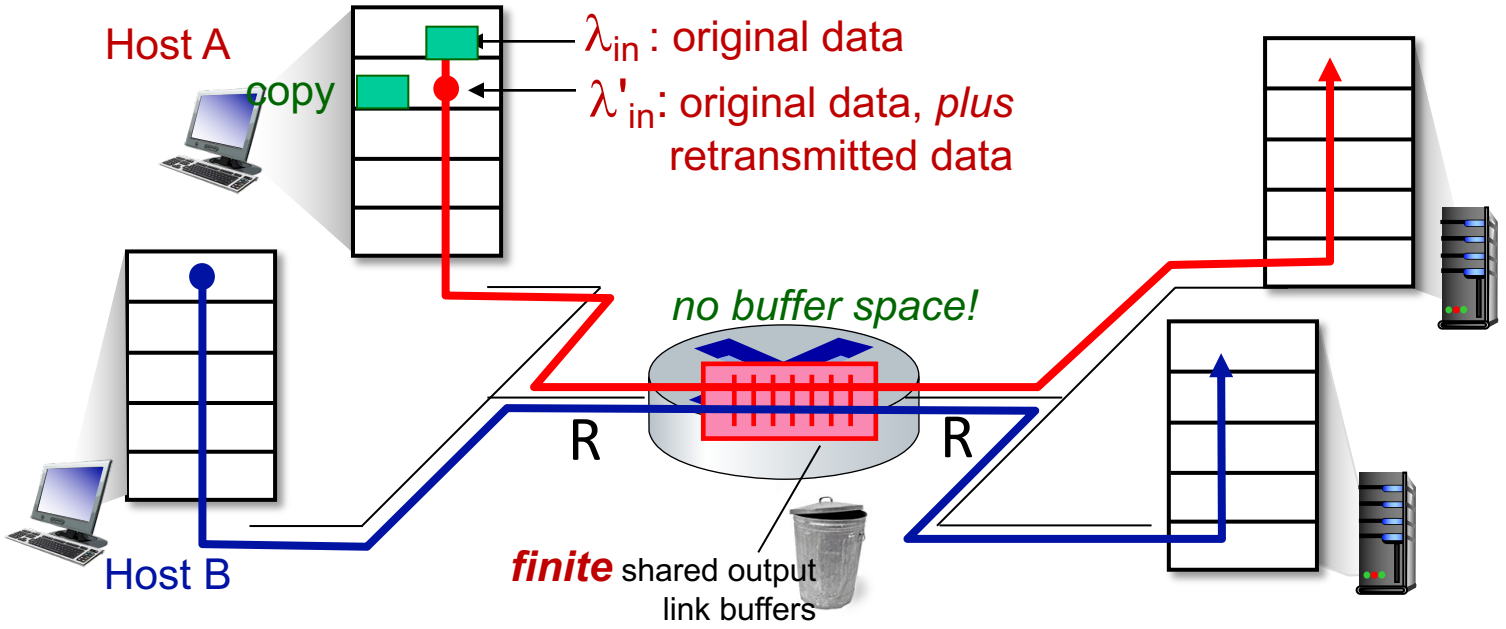




Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

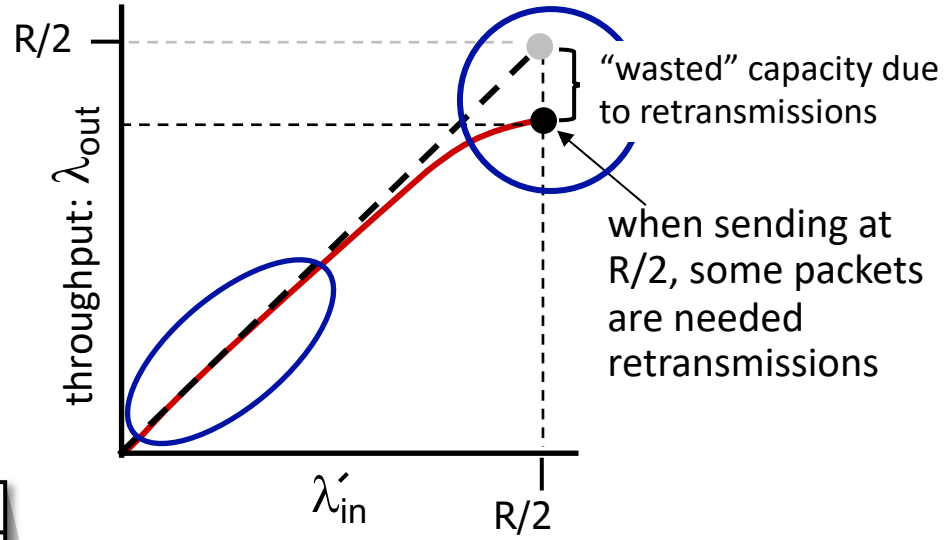
- packets can be lost (dropped at router) due to full buffers
- sender **knows** when packet has been dropped: only resends if packet *known* to be lost

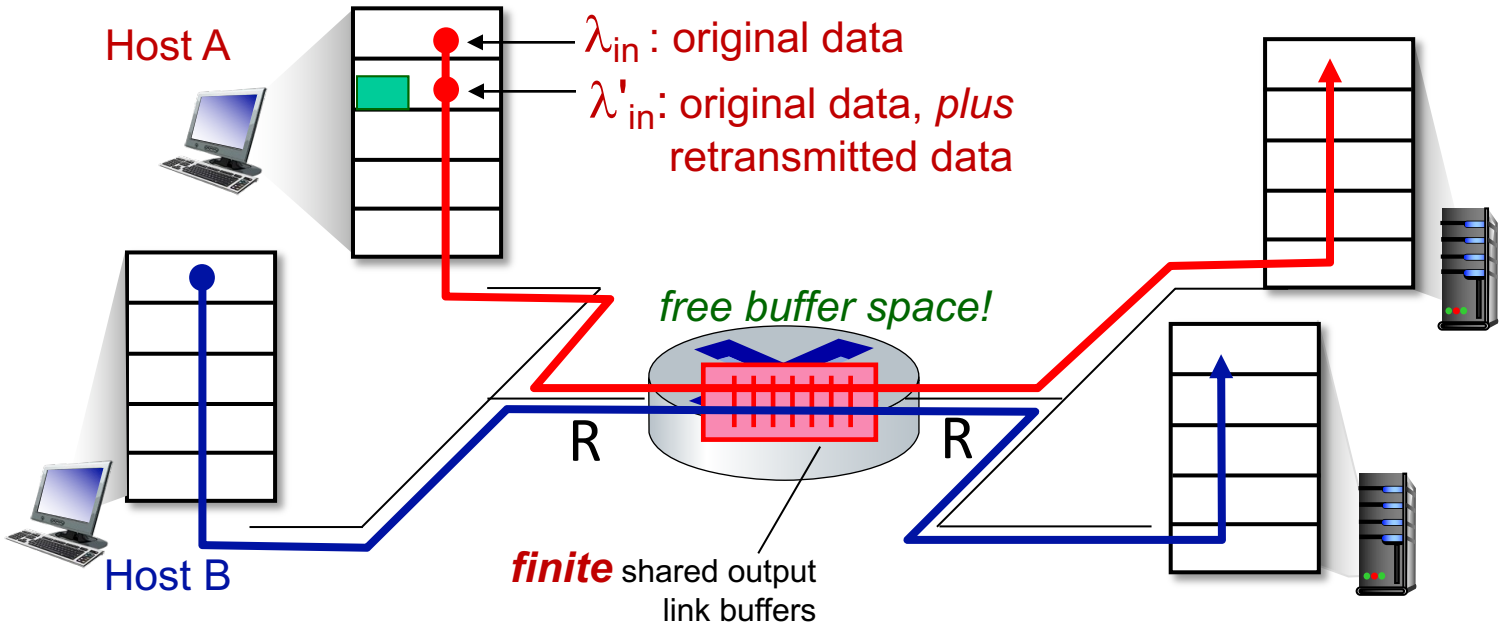


Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

- packets can be lost (dropped at router) due to full buffers
- sender **knows** when packet has been dropped: only resends if packet *known* to be lost

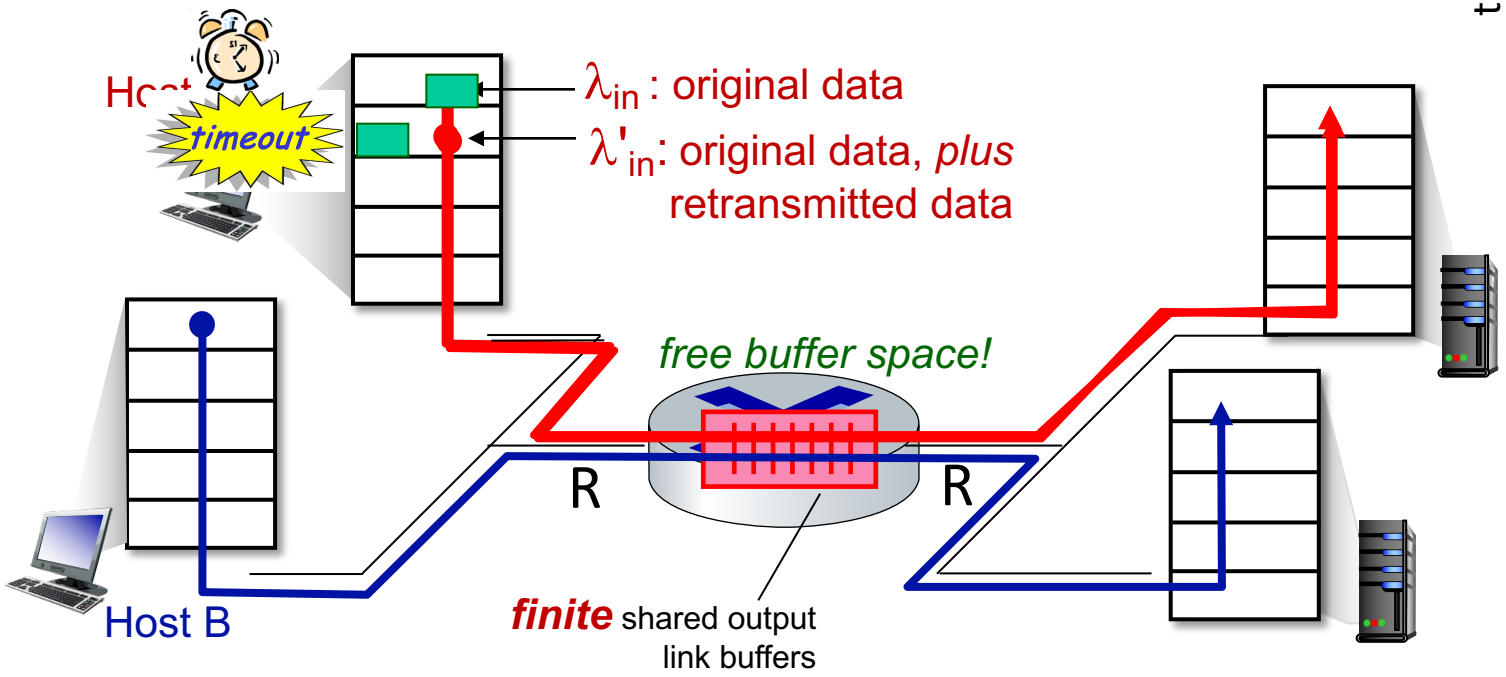


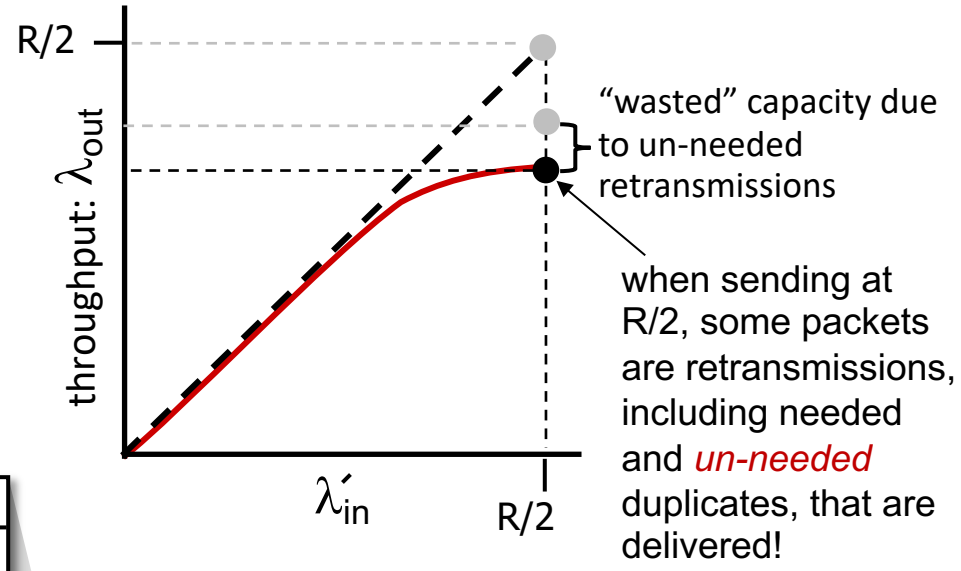


Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

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

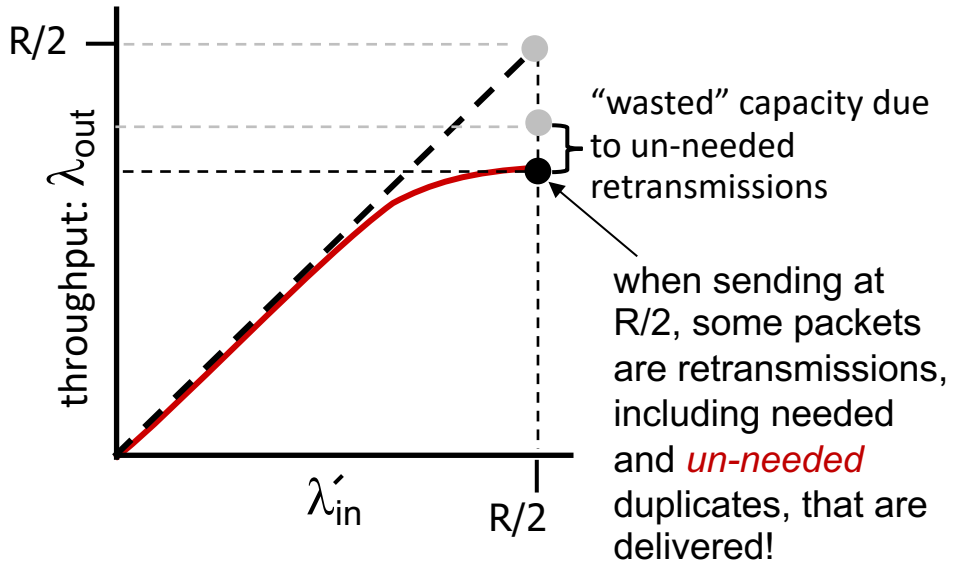




Causes/costs of congestion: scenario 2

Realistic scenario: *un-needed duplicates*

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



"costs" of congestion:

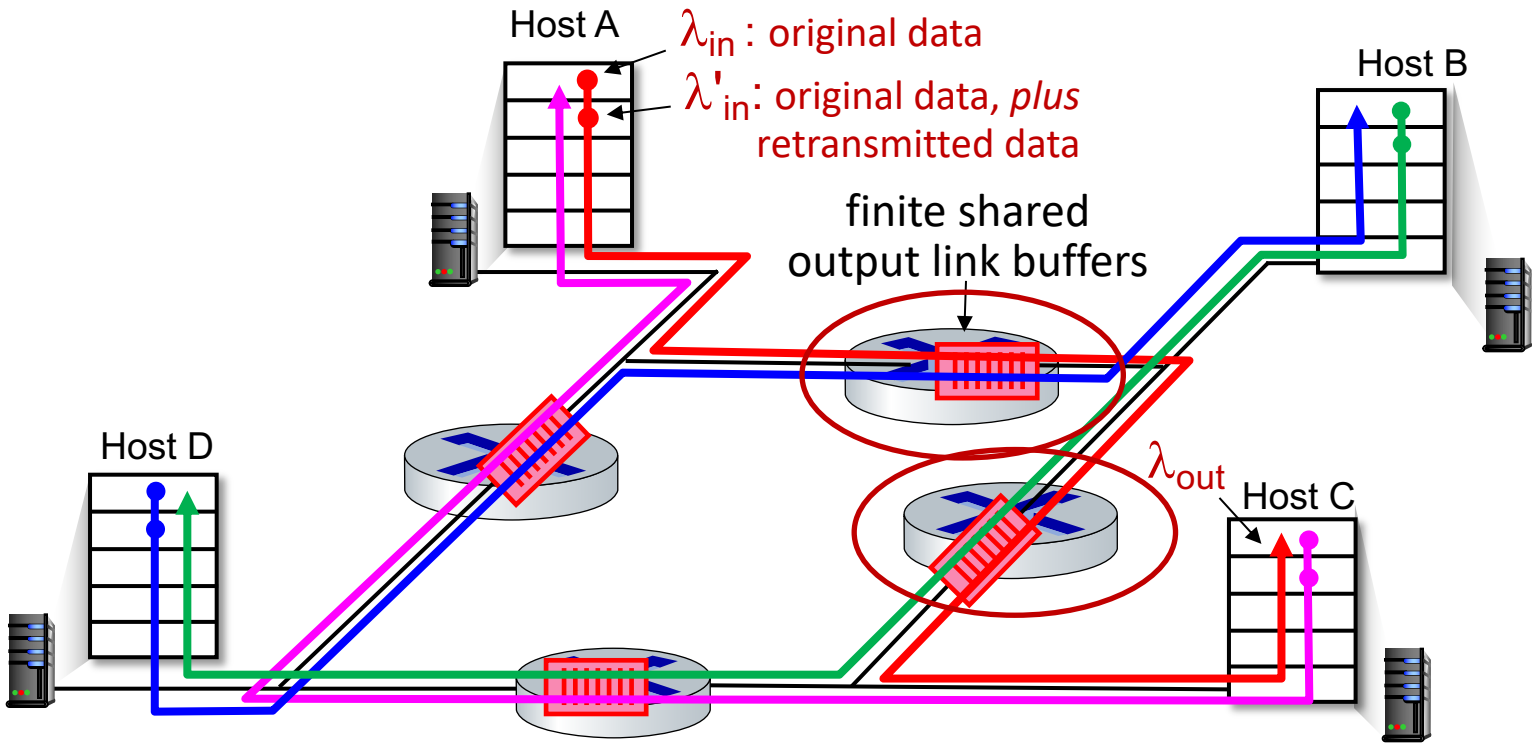
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

Causes/costs of congestion: scenario 3

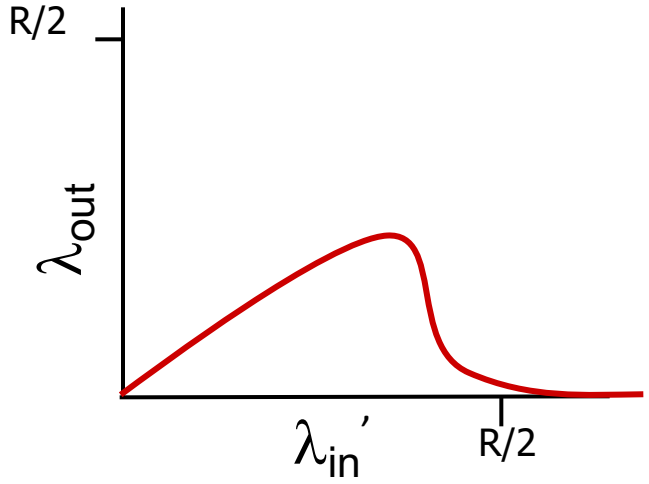
- *four* senders
- *multi-hop* paths
- timeout/retransmit

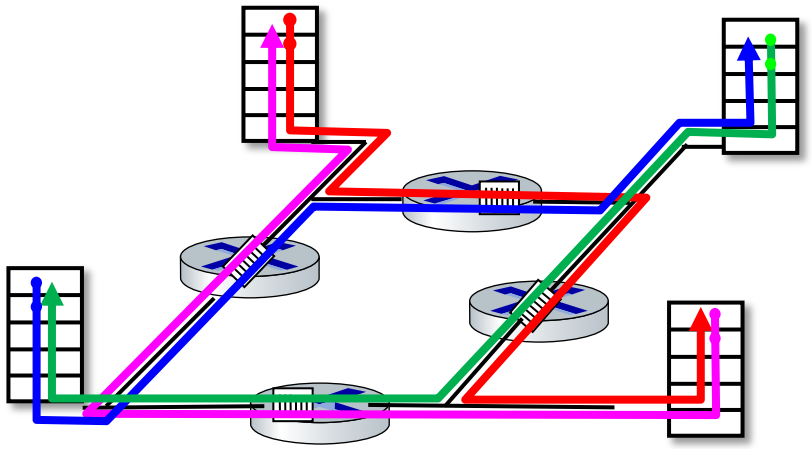
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



Causes/costs of congestion: scenario 3



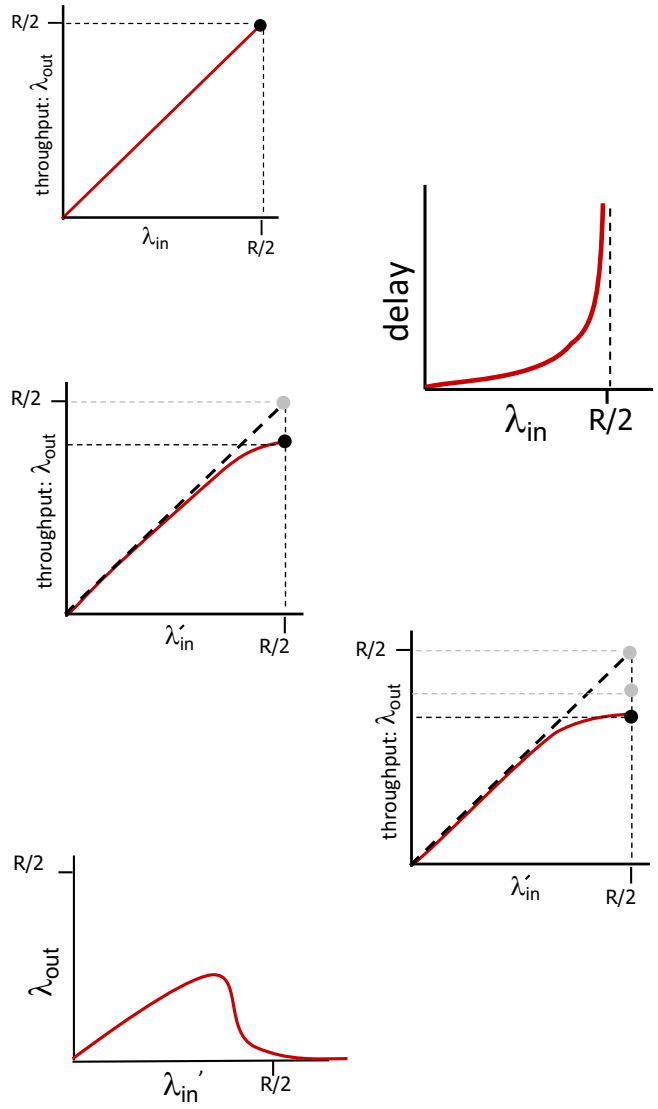


another "cost" of congestion:

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

Causes/costs of congestion: insights

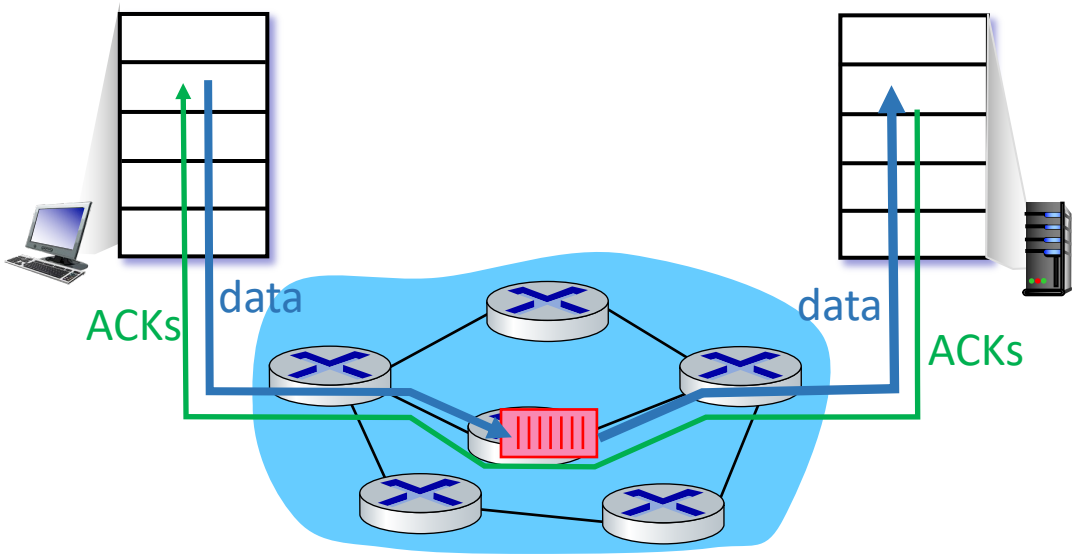
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



Approaches towards congestion control

End-end congestion control:

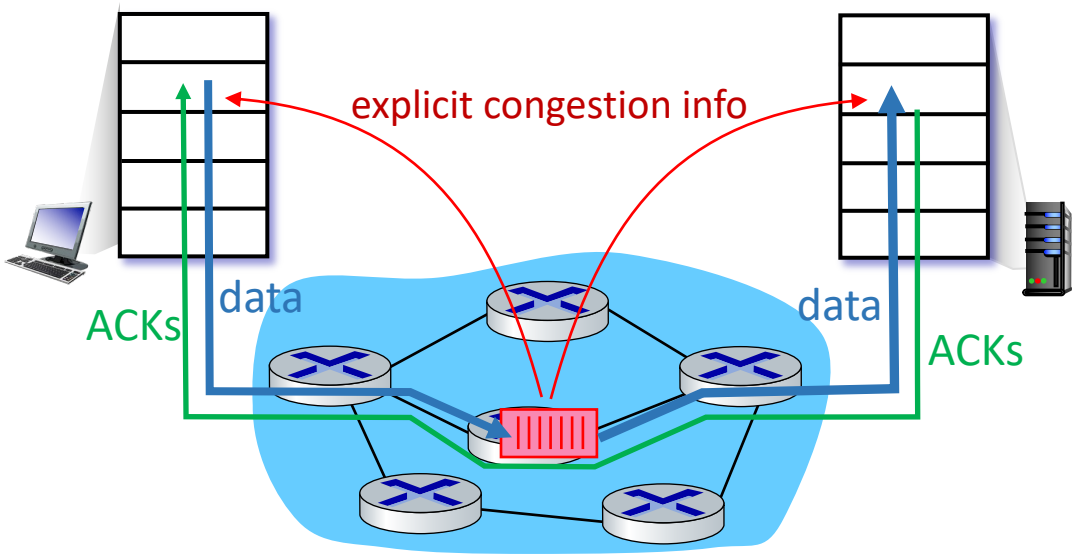
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate



- TCP ECN (Explicit Congestion Notification), ATM, DECbit protocols

Problems to be addressed for e2e congestion control

1. How can the sender limit the rate?

2. How can the sender detect congestion?

3. Which algorithm should be used to set the sender rate (how much to decrease or increase the rate and when)

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

1. How can the sender limit the rate?

- To limit the sending rate, the sender uses the variable CWND (congestion window) which, together with RWND, defines the size of the sender window
  - CWND: congestion window (network congestion control)
  - RWND: receiver window (end to end flow control)

The sender window size is set to = min(rwnd, cwnd)

The sender rate is bounded from above by $\frac{\text{window size}}{\text{RTT}}$

as the sender can send up to the window size and must wait an ACK before updating it (open or close) → rate control

2. How to detect congestion?

When the sender detects a loss:

- Duplicate ACKs and timeouts are weak / strong congestion indicators

The sender can react to the perceived network status:

- If ACKs arrive in a sequence and with reasonable frequency, the rate can be increased.
  To increase the sender rate we must increase the sender window size.

- If there are duplicate ACKs or timeouts, the rate must be decreased.
  To decrease the sender rate we must decrease the sender window size.

TCP is said to be SELF-CLOCKING as it reacts to ACKs to determine the congestion window size

3. Which algorithm should be used to set the rate?

The idea of TCP is to increment the rate when there is no congestion and decrease it when there is.

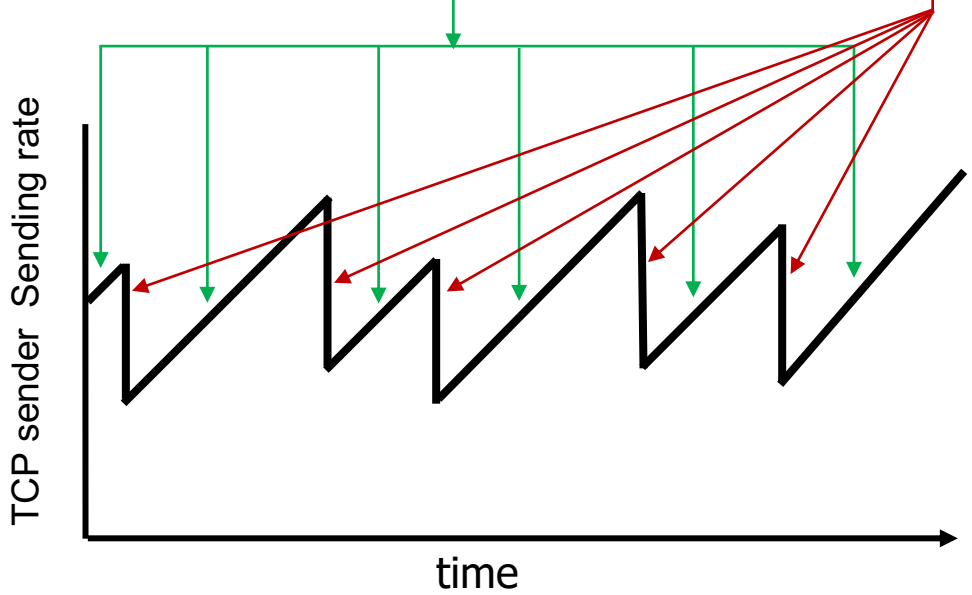
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size (MSS) every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe and Reno)

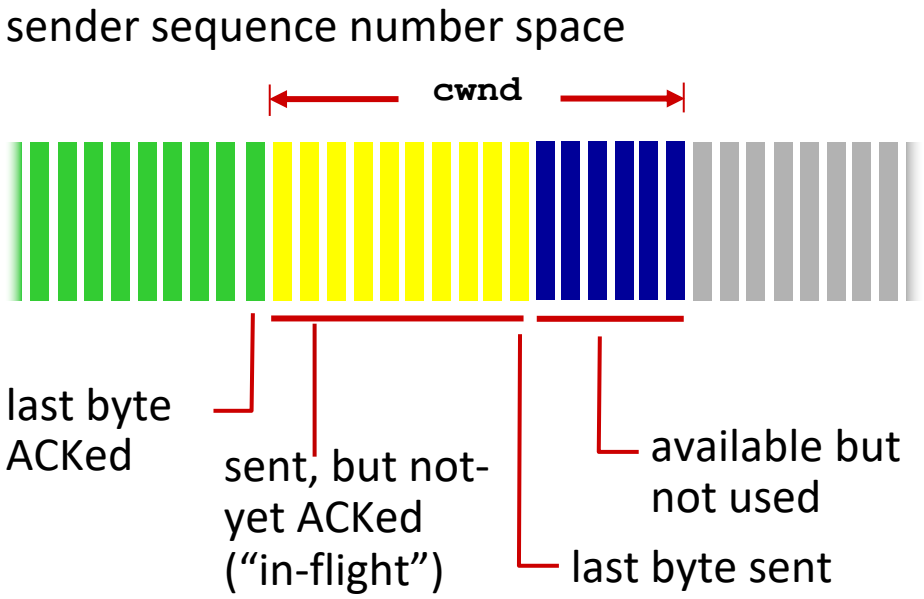
Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

Goal: to find a tradeoff between network utilization and delivery performance

TCP congestion control: details

We neglect rwnd here



TCP sending behavior:

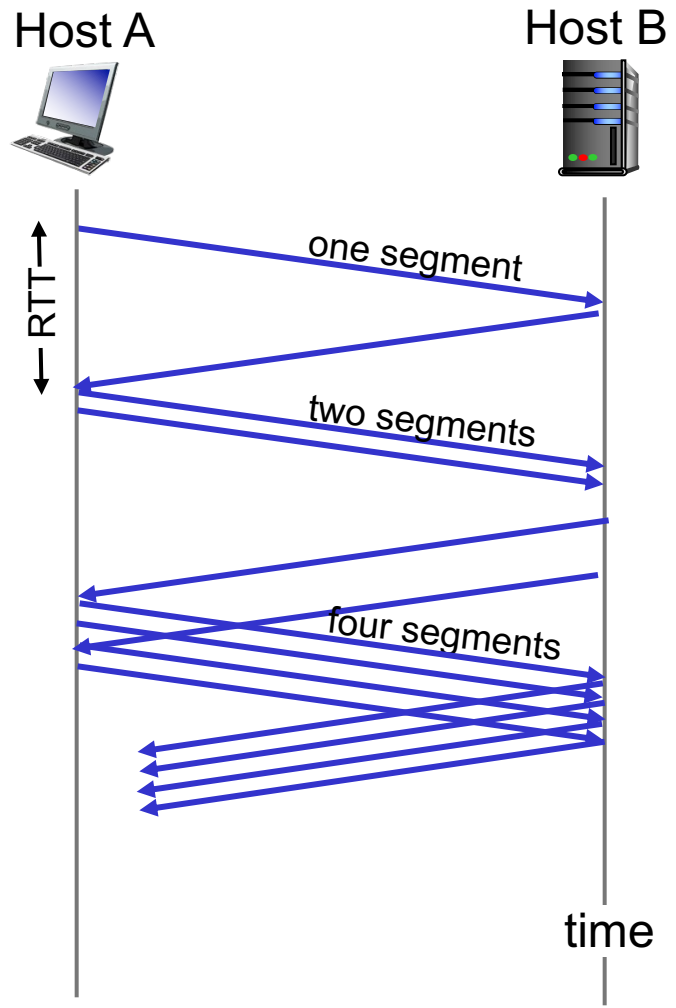
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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

- TCP sender limits transmission: `LastByteSent- LastByteAcked ≤ cwnd`
- cwnd is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

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



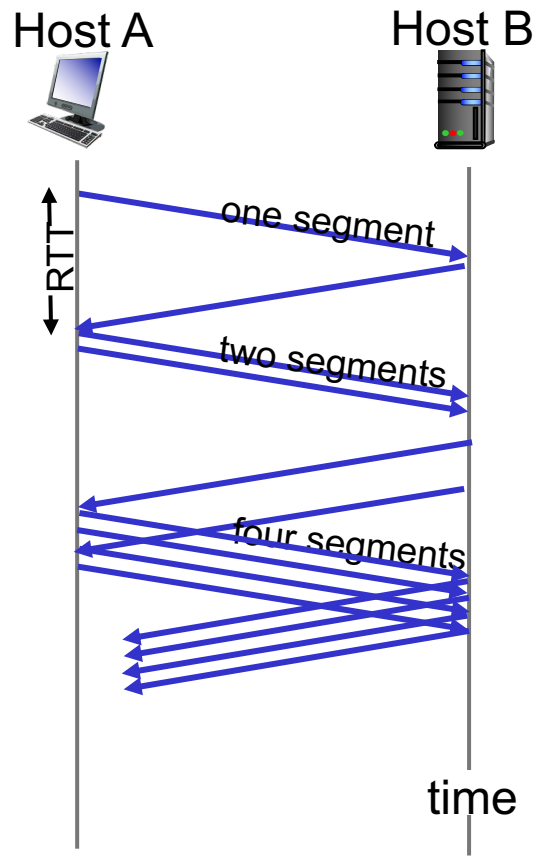
TCP slow start in detail

increase rate exponentially at each RTT

done by incrementing **`cwnd`** for every ACK received

If an ACK arrives, $cwnd = cwnd + 1$.

Start	$\rightarrow$	$cwnd = 1 \rightarrow 2^0$
After 1 RTT	$\rightarrow$	$cwnd = cwnd + 1 = 1 + 1 = 2 \rightarrow 2^1$
After 2 RTT	$\rightarrow$	$cwnd = cwnd + 2 = 2 + 2 = 4 \rightarrow 2^2$
After 3 RTT	$\rightarrow$	$cwnd = cwnd + 4 = 4 + 4 = 8 \rightarrow 2^3$



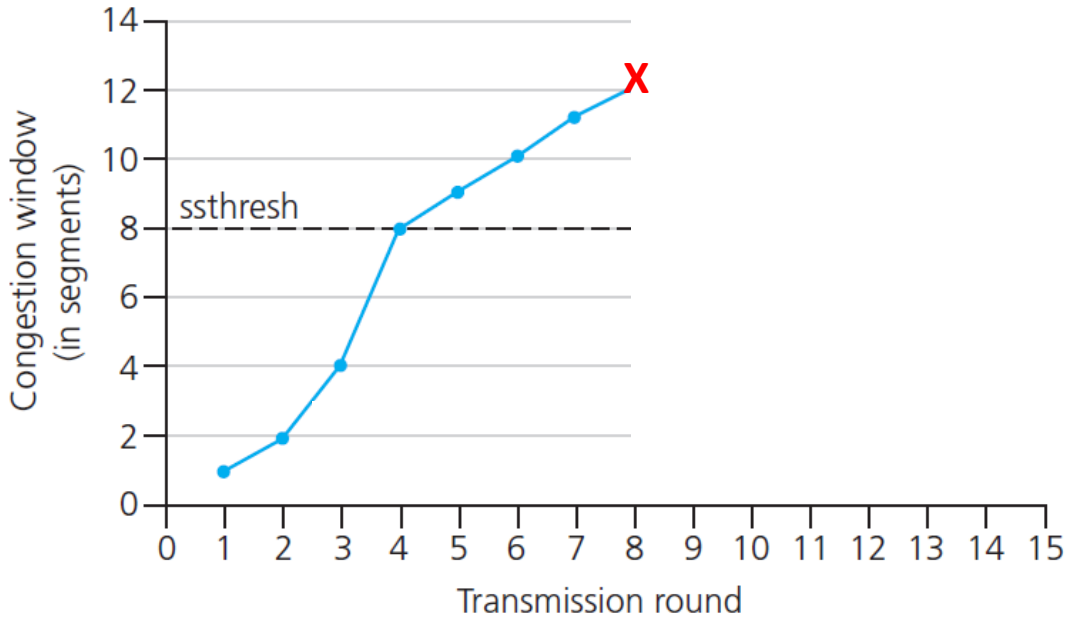
In the slow-start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.

TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to a more moderate – linear - increase?

A: when **`cwnd`** gets to a threshold set to 1/2 of its value before a loss detection.

Because by doubling it we would likely be in trouble again!



Threshold update:

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

(i.e., upon reception of 3 duplicate ACKs or at the expiration of a timeout)

End of the slow start phase

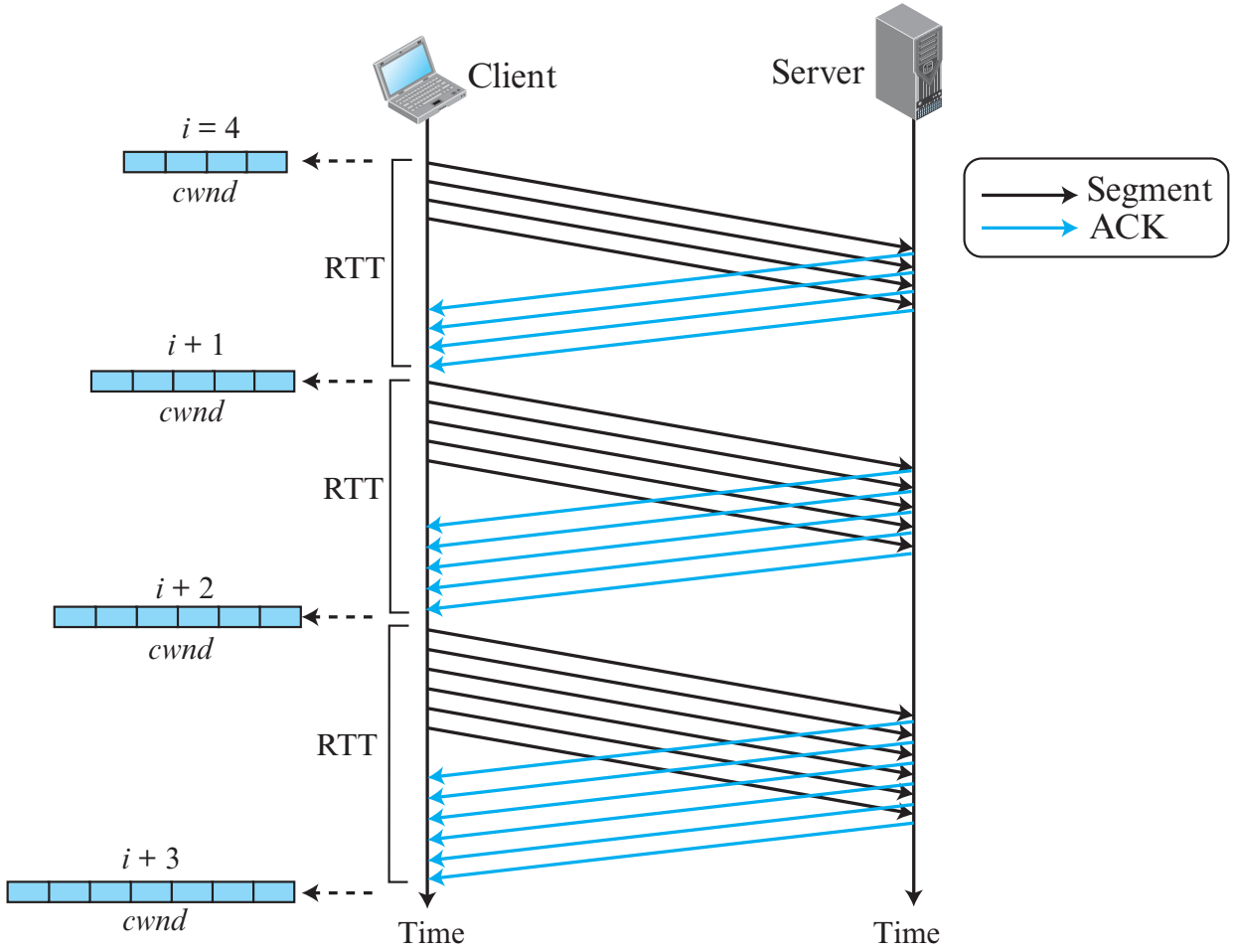
- cwnd **grows exponentially** until
  (a) the sender experiences **a loss or**
  (b) **the cwnd size reaches the ssthreshold**

  - In case (a) of a timeout we set ssthreshold = cwnd/2 and cwnd=1, and mantain exp growth
  - In case (b) of threshold we enter a phase of LINEAR INCREASE (congestion avoidance)

The CONGESTION AVOIDANCE phase provides

- Linear increase of the cwnd (increment of 1 MSS every time the entire window is acked) ← a fraction of MSS at each ACK
- Stop linear increase upon congestion detection (timeout or 3 duplicate acks)
- At a timeout event, ssthreshold=cwnd/2 and cwnd=1
- Both in the slow start and in the congestion avoidance phase, what happens at the 3 duplicate ack event varies depending on the TCP implementation (Taho or Reno)

Congestion avoidance: additive increase



Congestion avoidance: additive increase

If an ACK arrives, $cwnd = cwnd + (1/cwnd)$.

Start	$\rightarrow$	$cwnd = i$
After 1 RTT	$\rightarrow$	$cwnd = i + 1$
After 2 RTT	$\rightarrow$	$cwnd = i + 2$
After 3 RTT	$\rightarrow$	$cwnd = i + 3$

In the congestion-avoidance algorithm, the size of the congestion window increases additively until congestion is detected.

The 3-duplicate-ACKs case

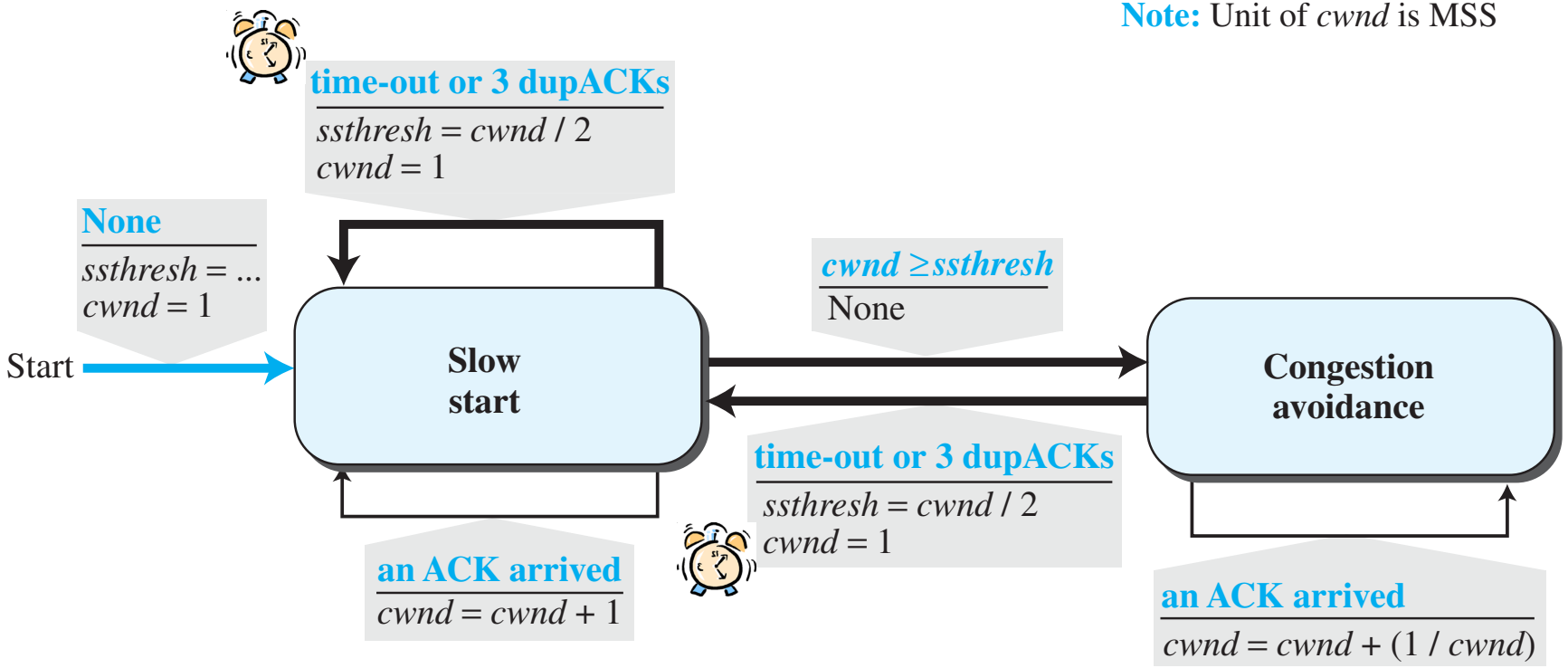
TCP Tahoe solution

The older version of TCP (Taho) considers a 3-Duplicate ACK in the same manner as it does for a timeout.
The only two modes are Slow Start and Congestion Avoidance.

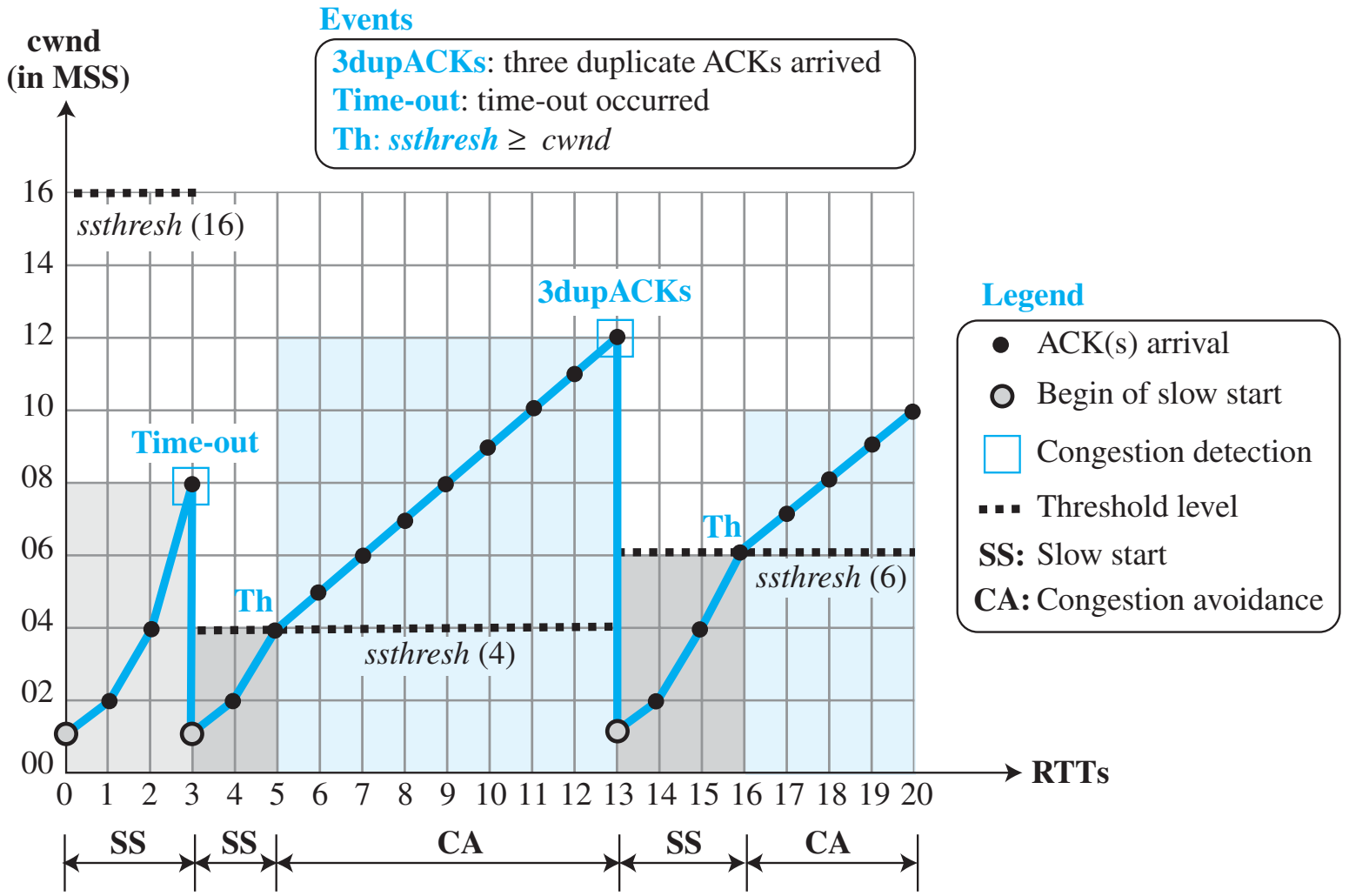
TCP Reno solution

The philosophy of TCP Reno is that 3 duplicate ACKs is a lighter indicator of congestion than a timeout .
It then brings to the Fast Recovery mode.

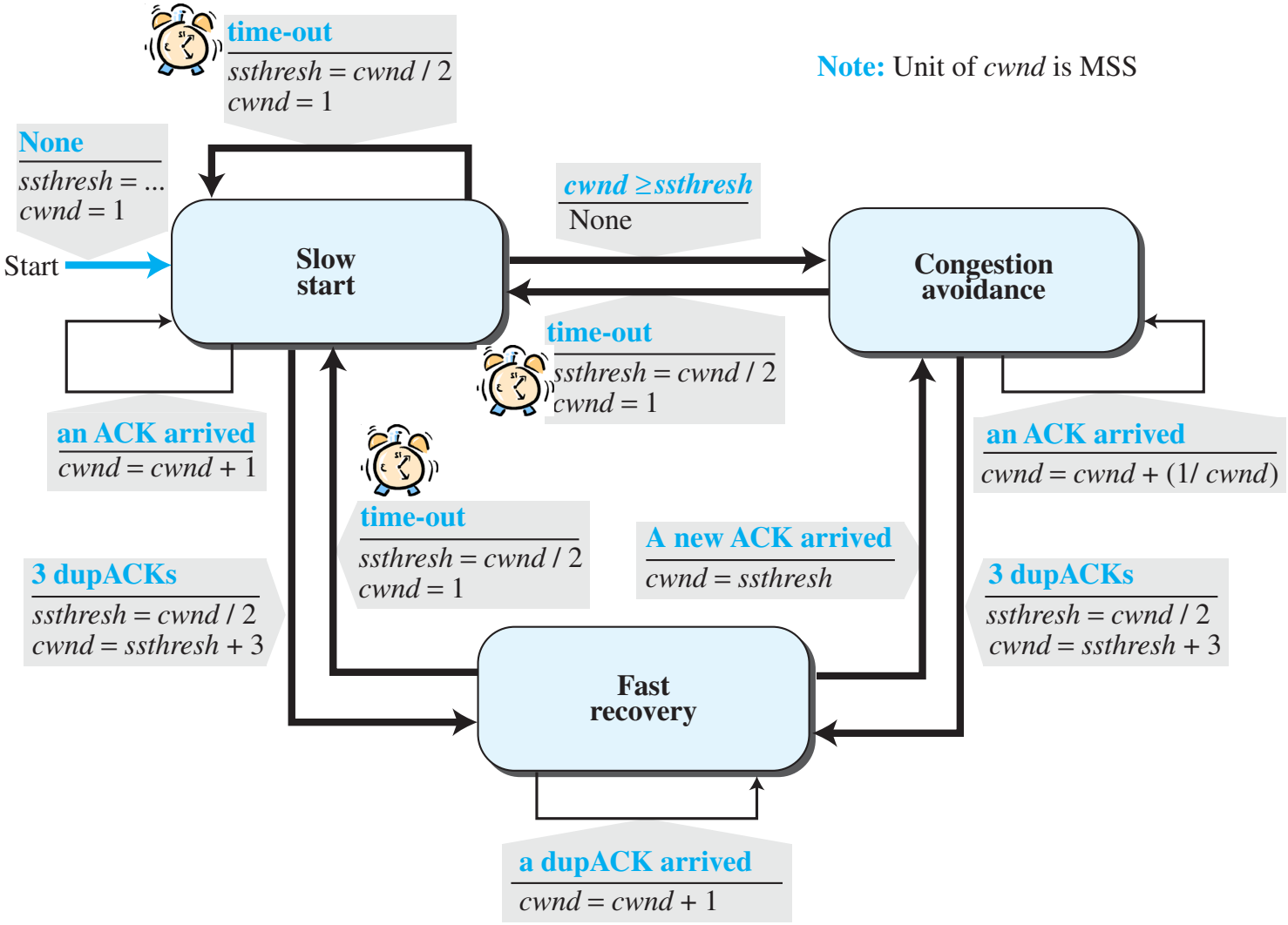
Tahoe TCP


Note: Unit of cwnd is MSS
time-out or 3 dupACKs
ssthresh = cwnd / 2
cwnd = 1
None
ssthresh = ...
cwnd = 1
Start
Slow start
cwnd ≥ssthresh
None
Congestion avoidance
time-out or 3 dupACKs
ssthresh = cwnd / 2
cwnd = 1
an ACK arrived
cwnd = cwnd + 1
an ACK arrived
cwnd = cwnd + (1 / cwnd)

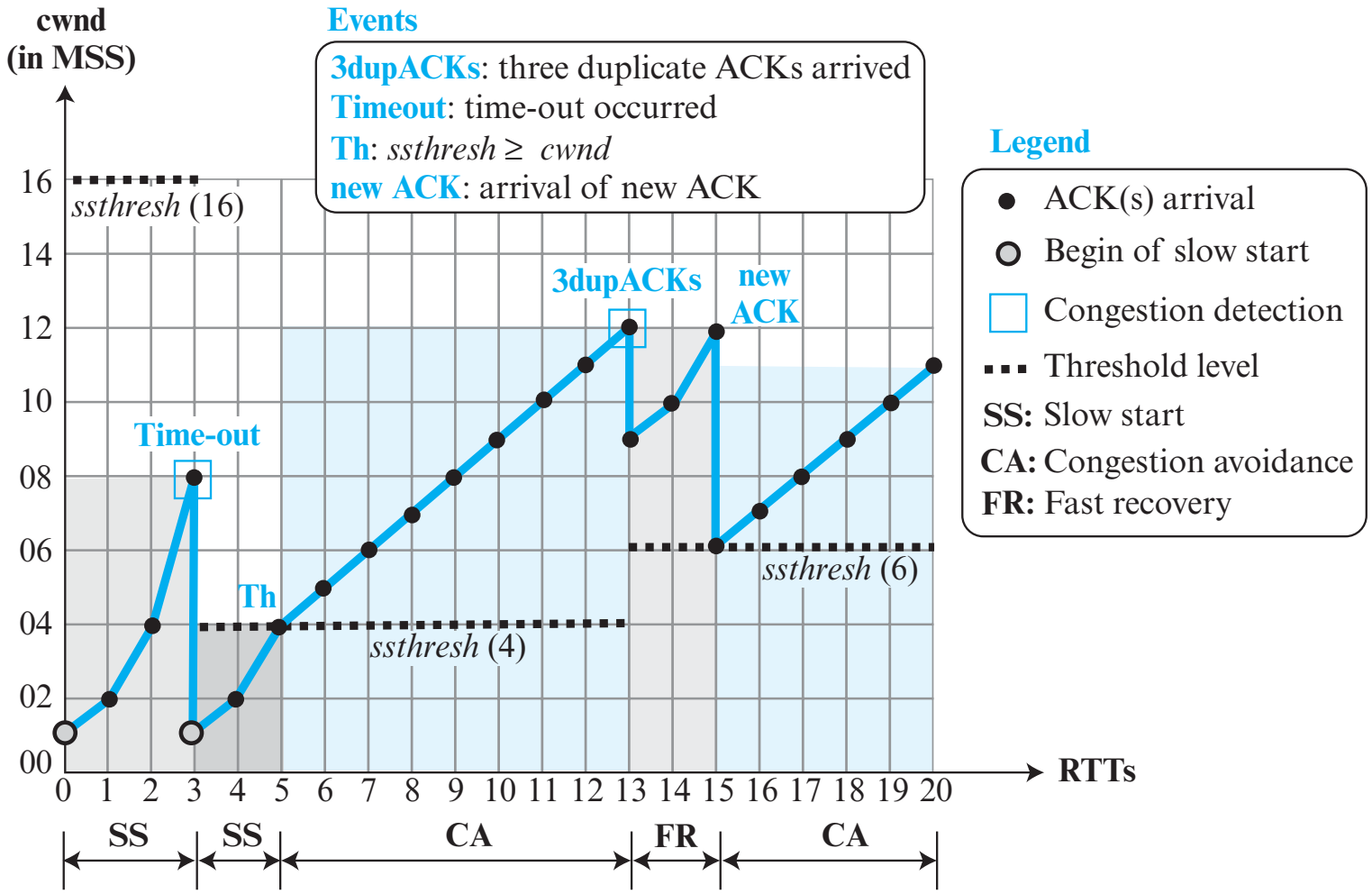
Example of Tahoe TCP



Reno TCP


time-out
ssthresh = cwnd / 2
cwnd = 1
Note: Unit of cwnd is MSS
None
ssthresh = ...
cwnd = 1
cwnd ≥ssthresh
None
Start
Slow start
Congestion avoidance
time-out
ssthresh = cwnd / 2
cwnd = 1
an ACK arrived
cwnd = cwnd + 1
time-out
ssthresh = cwnd / 2
cwnd = 1
an ACK arrived
cwnd = cwnd + (1/ cwnd)
A new ACK arrived
cwnd = ssthresh
3 dupACKs
ssthresh = cwnd / 2
cwnd = ssthresh + 3
3 dupACKs
ssthresh = cwnd / 2
cwnd = ssthresh + 3
Fast recovery
a dupACK arrived
cwnd = cwnd + 1

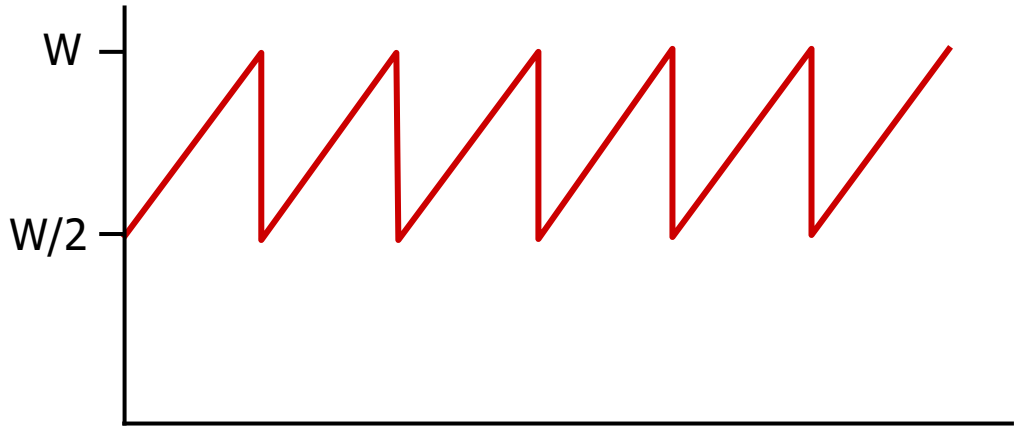
Example of Reno TCP



TCP throughput

- avg. TCP throughput 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. throughput is 3/4W per RTT

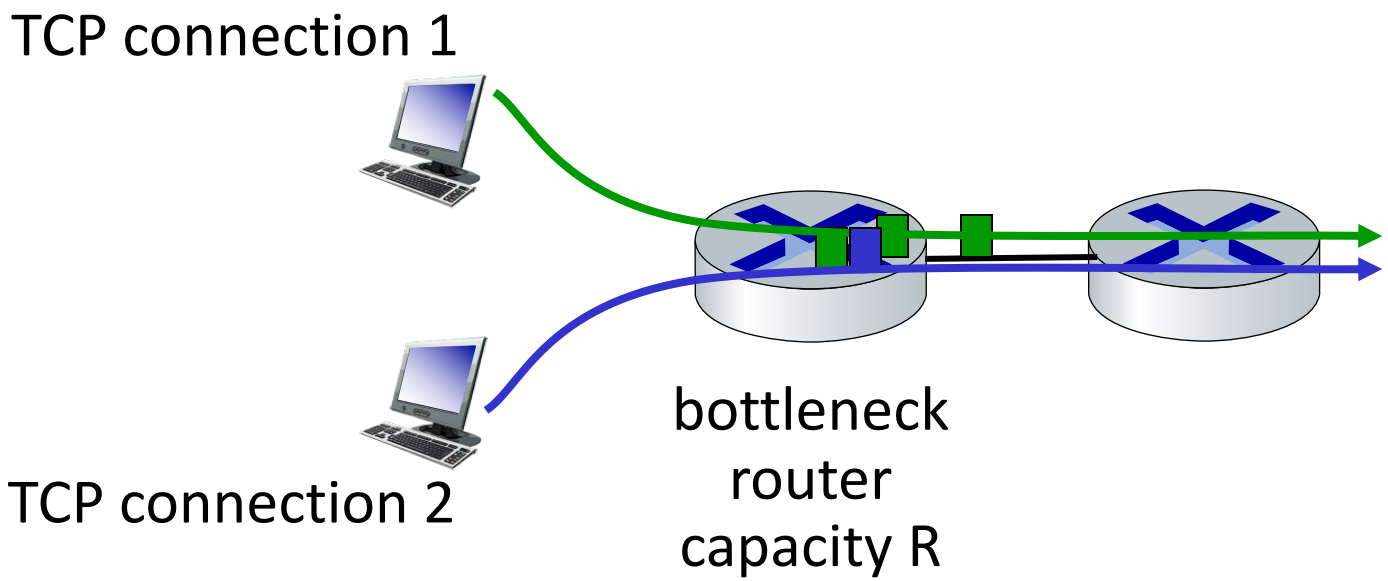
$$\text{avg TCP throughput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



As we ignore the slow start we consider half the window size at losses (it would be 1 MSS but exponentially grows up to half of W due to slow start)

TCP fairness

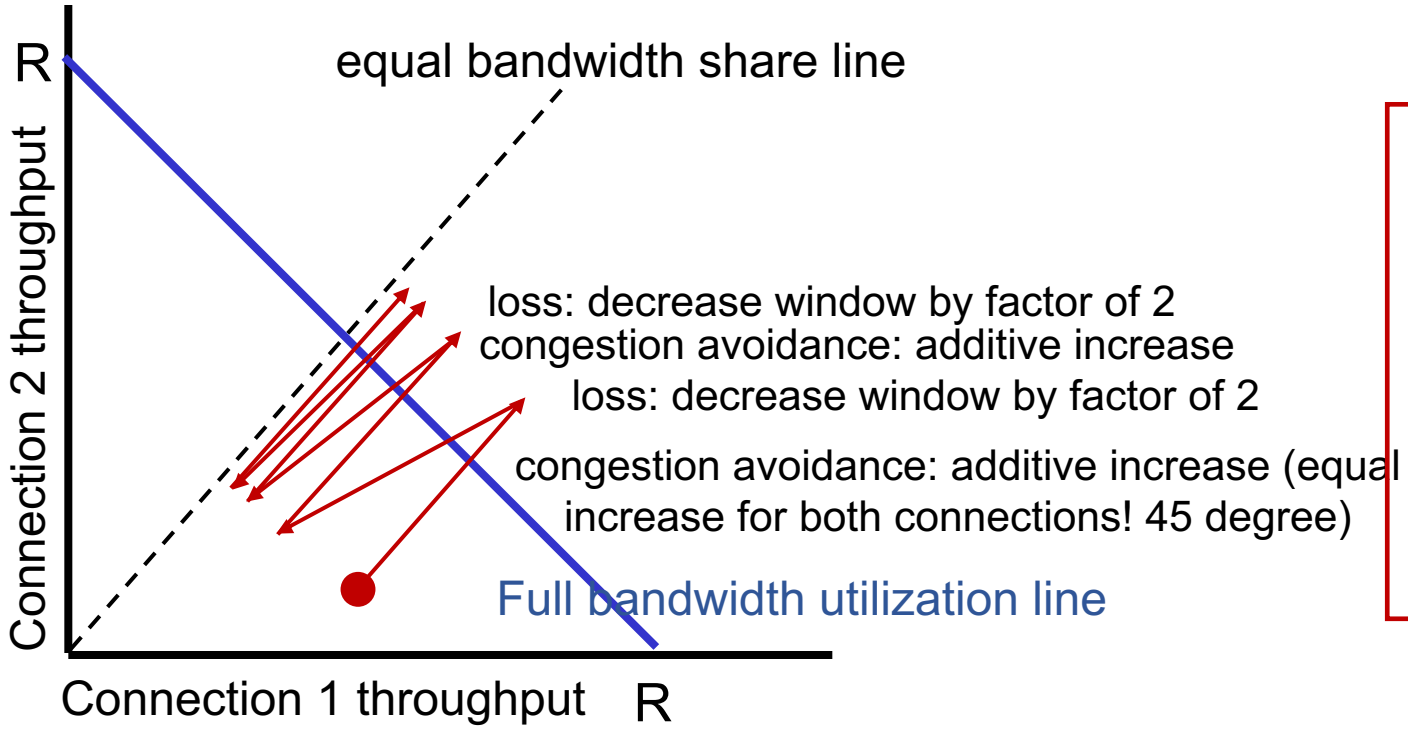
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

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



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Ideally the two rates should sum to R - plus - they should be equal in case of fairness

Fairness: must all network apps be “fair”?

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
- there is no “Internet police” policing use of congestion control

Fairness, parallel TCP connections

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

Exercise P40 (following slides): before addressing the complete exercise note that
the proposed plot ignores some important details of fast recovery. Consider the new plot, what do you think?

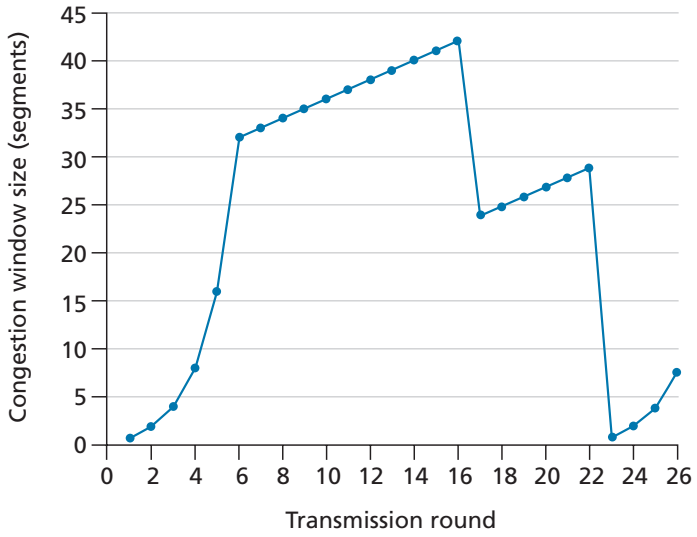
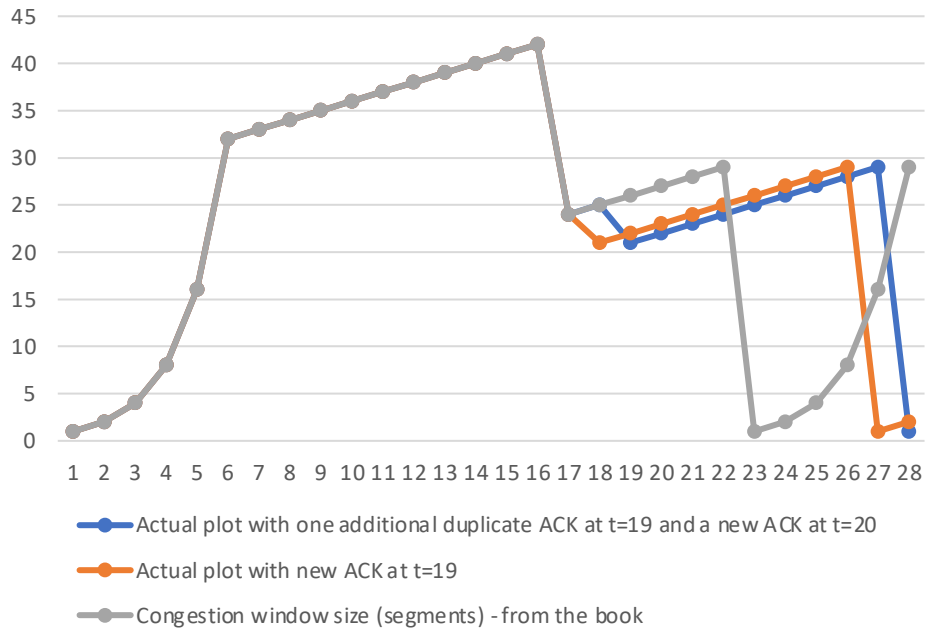


Figure 3.61 ♦ TCP window size as a function of time



P40. Consider Figure 3.61. Assuming TCP Reno is the protocol experiencing the behavior shown above, answer the following questions. In all cases, you should provide a short discussion justifying your answer.

a. Identify the intervals of time when TCP slow start is operating.

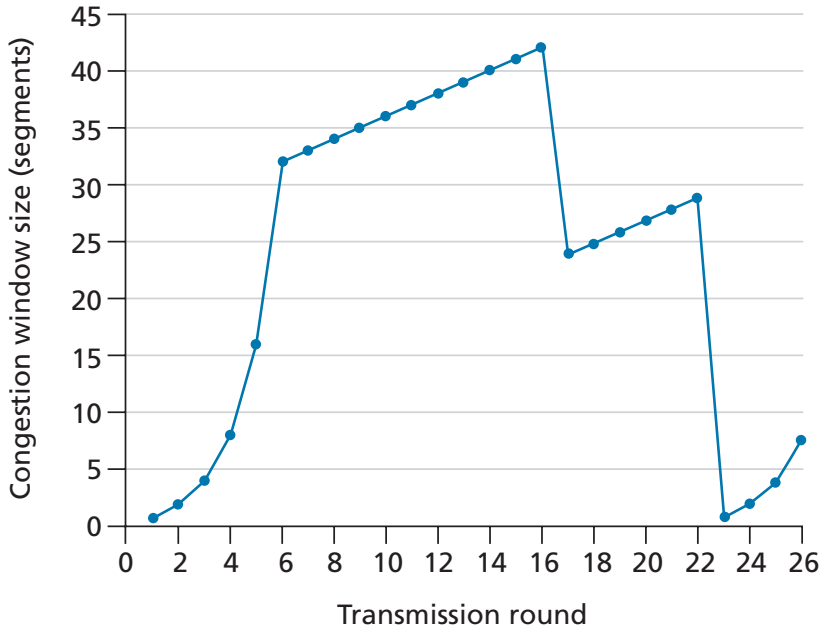


Figure 3.61 ♦ TCP window size as a function of time

P40. Consider Figure 3.61. Assuming TCP Reno is the protocol experiencing the behavior shown above, answer the following questions. In all cases, you should provide a short discussion justifying your answer.

a. Identify the intervals of time when TCP slow start is operating.

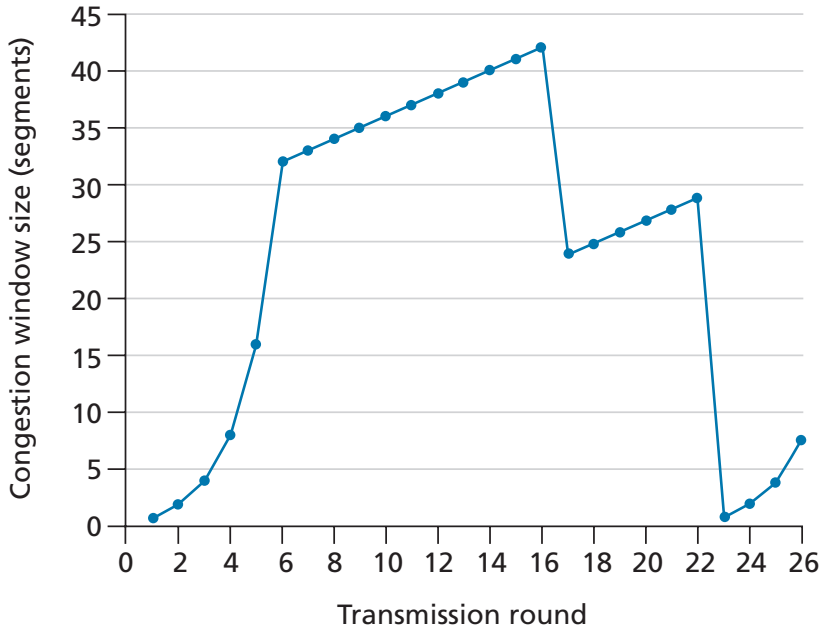


Figure 3.61 ♦ TCP window size as a function of time

Answer: [1,6] and [23,26].

b. Identify the intervals of time when TCP congestion avoidance is operating.

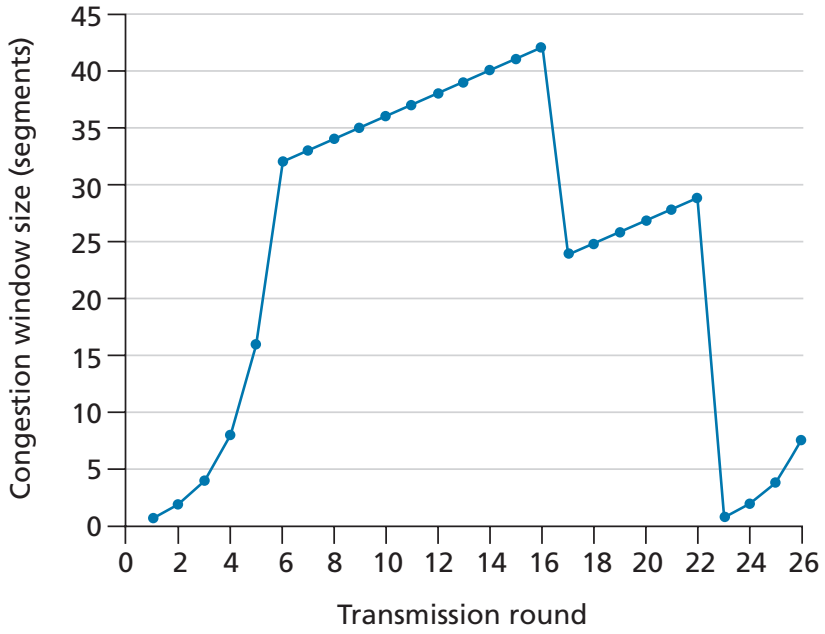


Figure 3.61 ♦ TCP window size as a function of time

b. Identify the intervals of time when TCP congestion avoidance is operating.

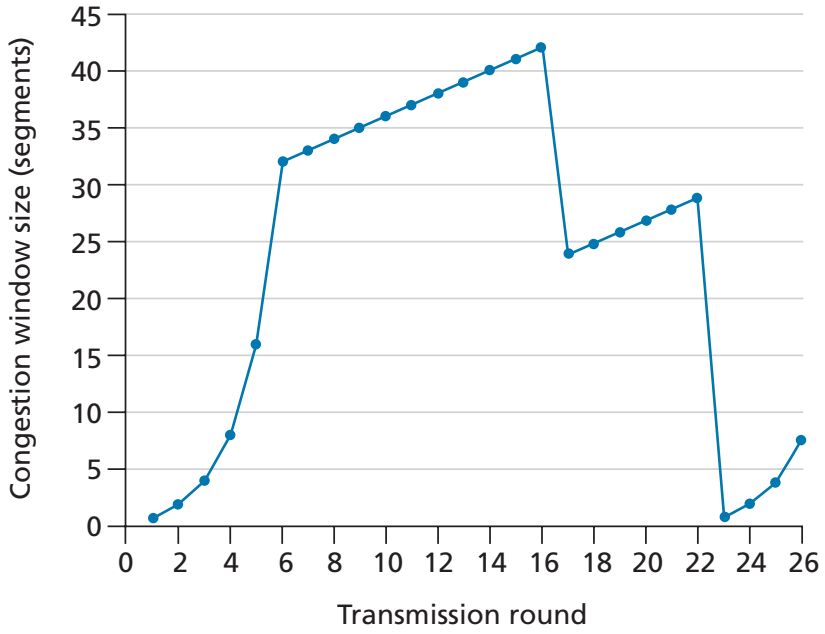


Figure 3.61 ♦ TCP window size as a function of time

Answer: [6,16] and [17,22].

c. After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

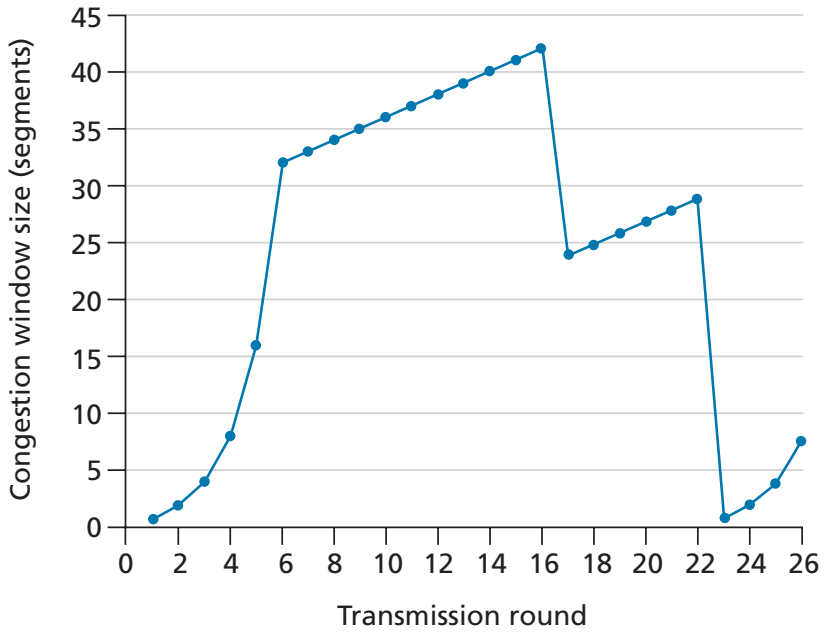


Figure 3.61 ♦ TCP window size as a function of time

c. After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

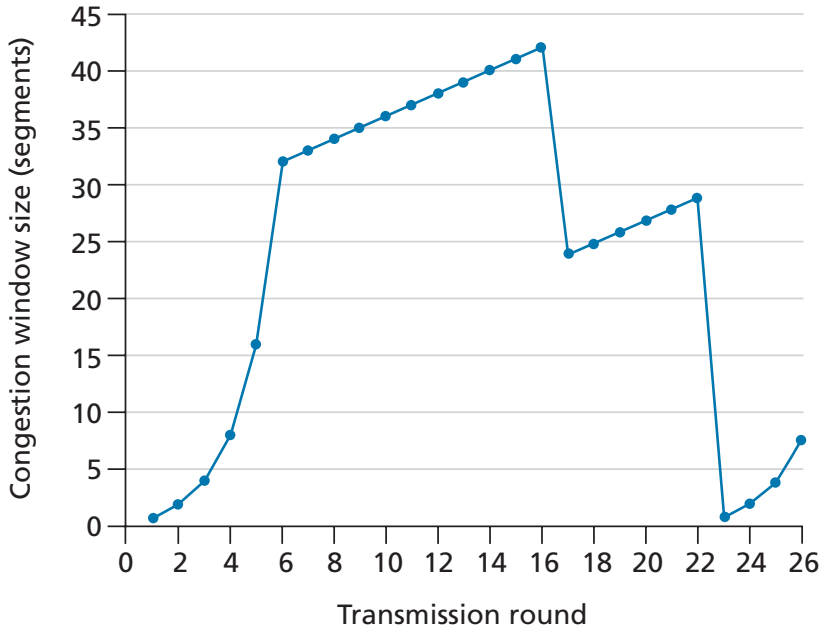


Figure 3.61 ♦ TCP window size as a function of time

Answer: The cwnd' is set to cwnd/2 + 3 = 42/2 + 3 = 24, hence it is a triple duplicate ACK

d. After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

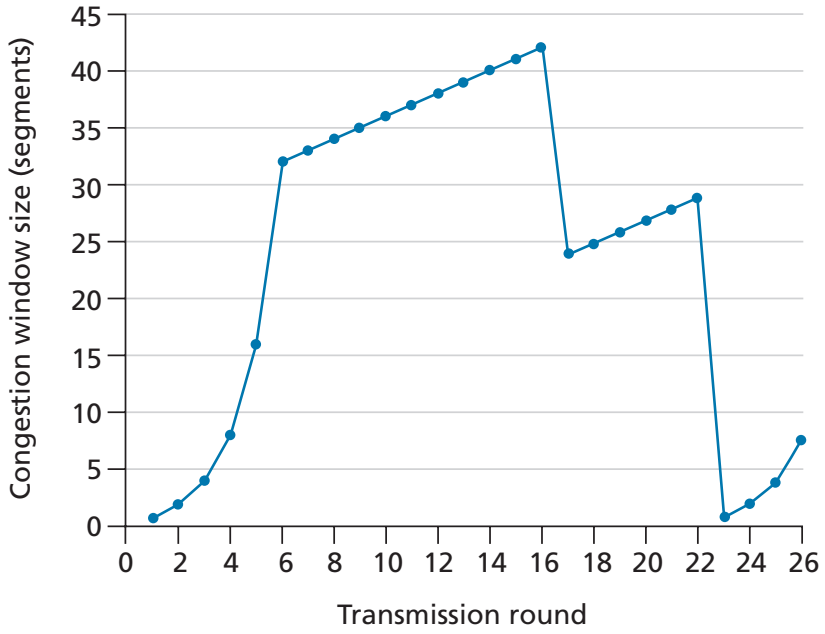


Figure 3.61 ♦ TCP window size as a function of time

d. After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

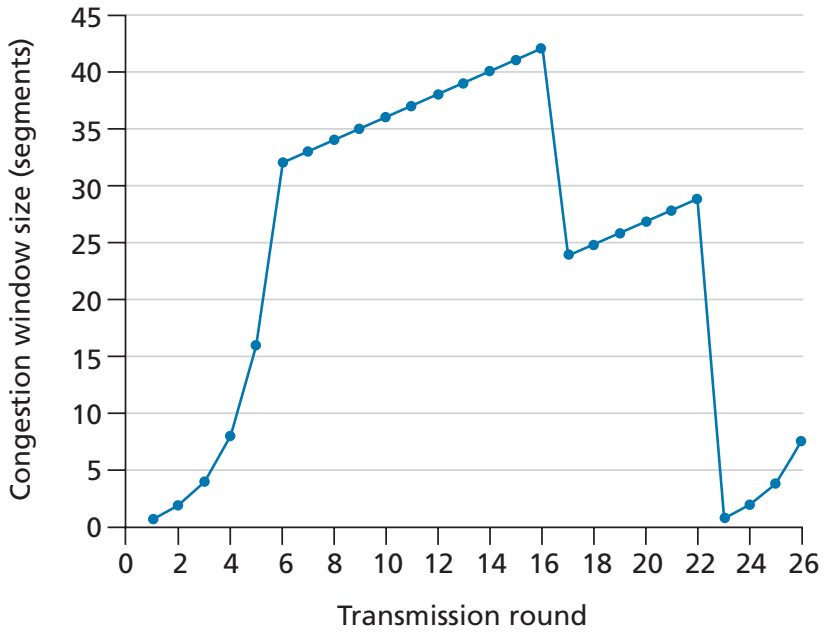


Figure 3.61 ♦ TCP window size as a function of time

Answer: The cwnd' is set to 1, hence it is a timeout

e. What is the initial value of `ssthresh` at the first transmission round?

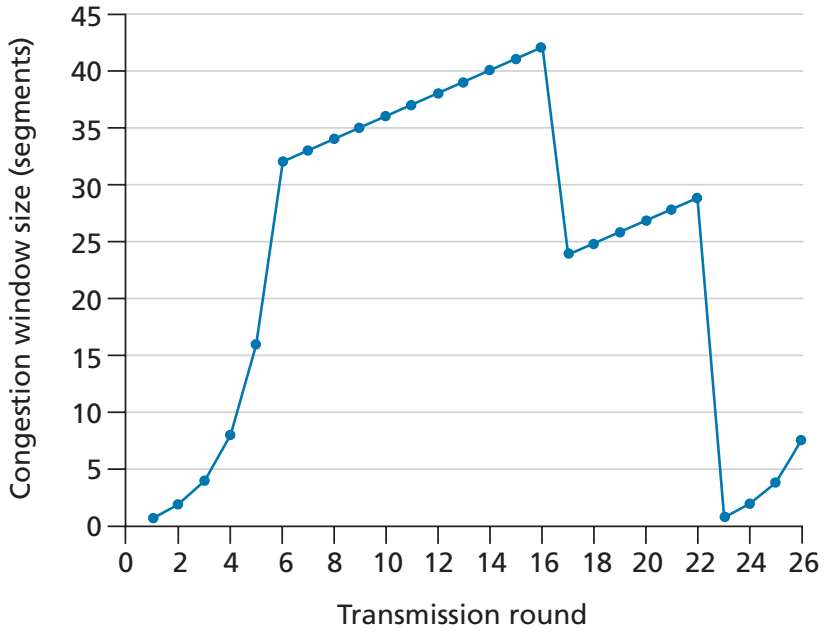


Figure 3.61 ♦ TCP window size as a function of time

e. What is the initial value of `ssthresh` at the first transmission round?

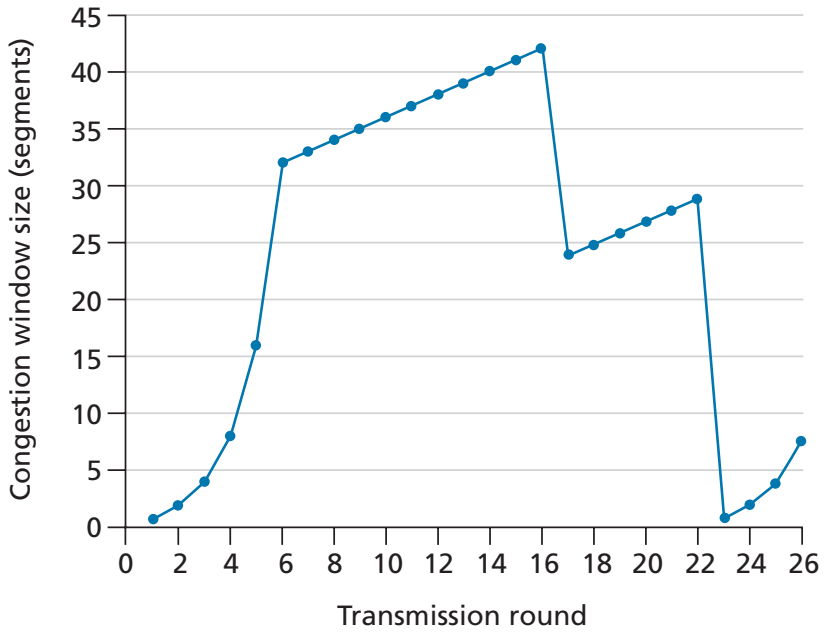


Figure 3.61 ♦ TCP window size as a function of time

Answer: sshthresh is initially set to 32.

f. What is the value of `ssthresh` at the 18th transmission round?

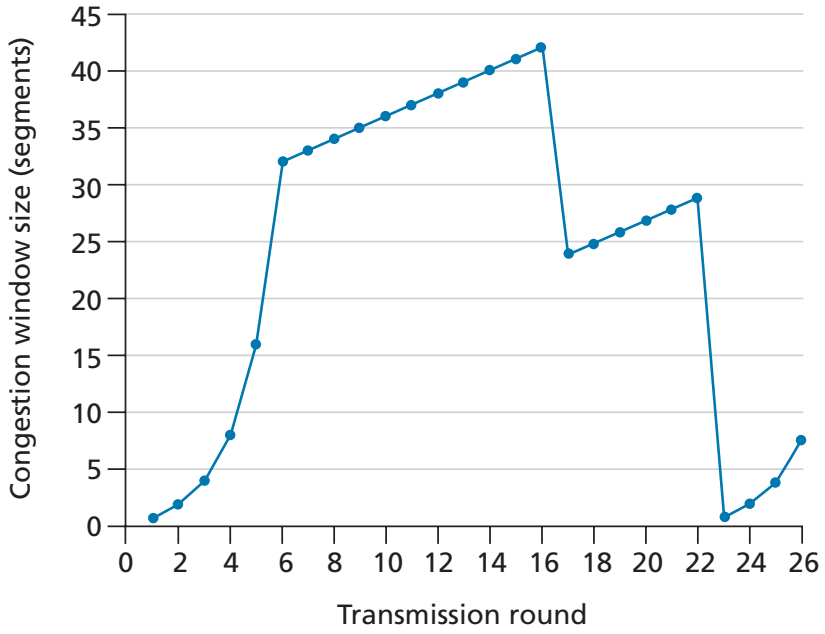


Figure 3.61 ♦ TCP window size as a function of time

f. What is the value of `ssthresh` at the 18th transmission round?

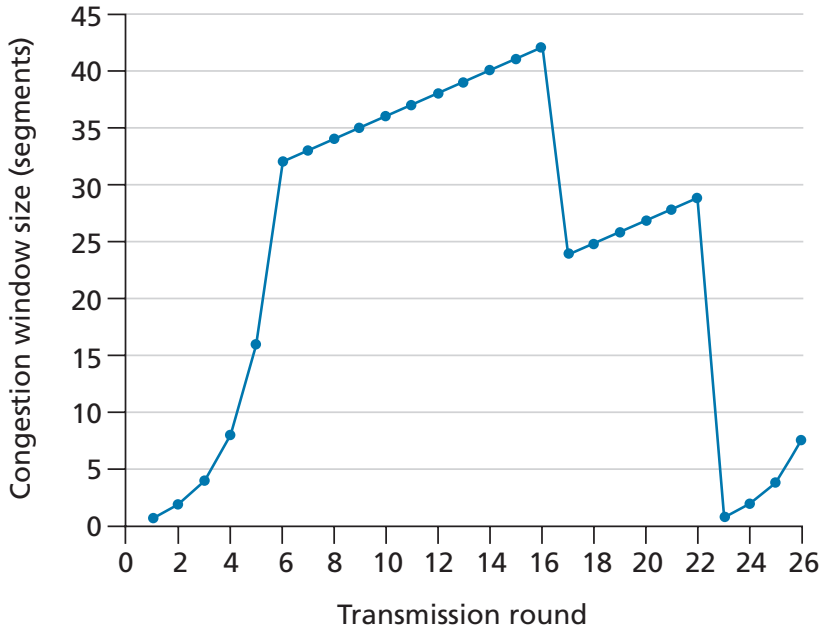


Figure 3.61 ♦ TCP window size as a function of time

Answer: sshthresh at the 18th round is set to cwnd/2 considering the cwnd observed before the detected loss, then sshthresh is 21.

g. What is the value of `ssthresh` at the 24th transmission round?

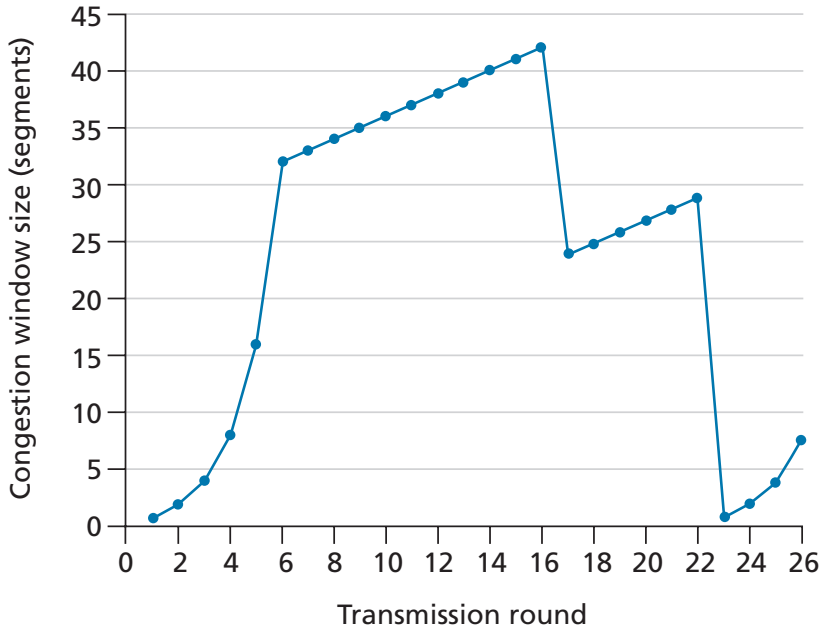


Figure 3.61 ♦ TCP window size as a function of time

g. What is the value of `ssthresh` at the 24th transmission round?

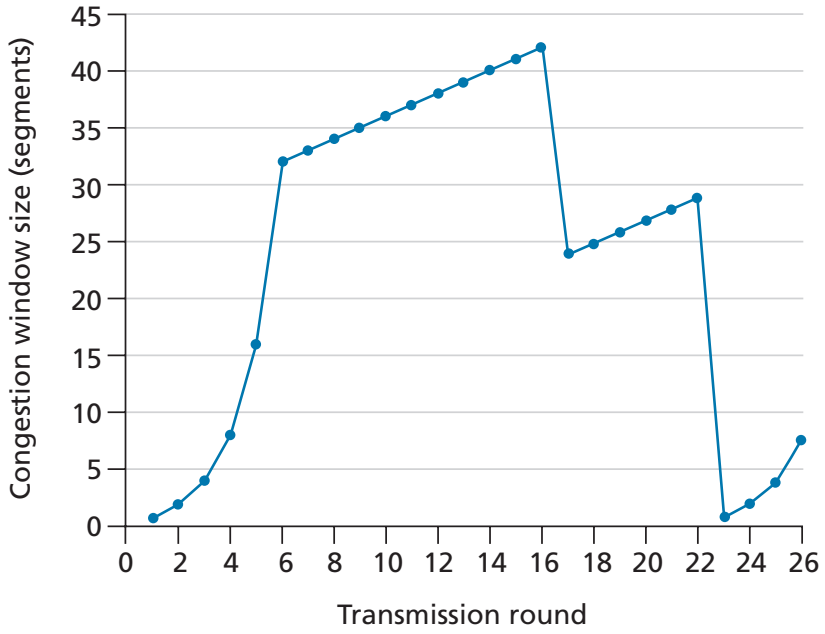


Figure 3.61 ♦ TCP window size as a function of time

Answer: sshthresh is set to half the value of cwnd observed when the loss was detected.
It was detected at round 22, when cwnd was 28. Hence sshthresh is set to 28/2=14.

h. During what transmission round is the 70th segment sent?

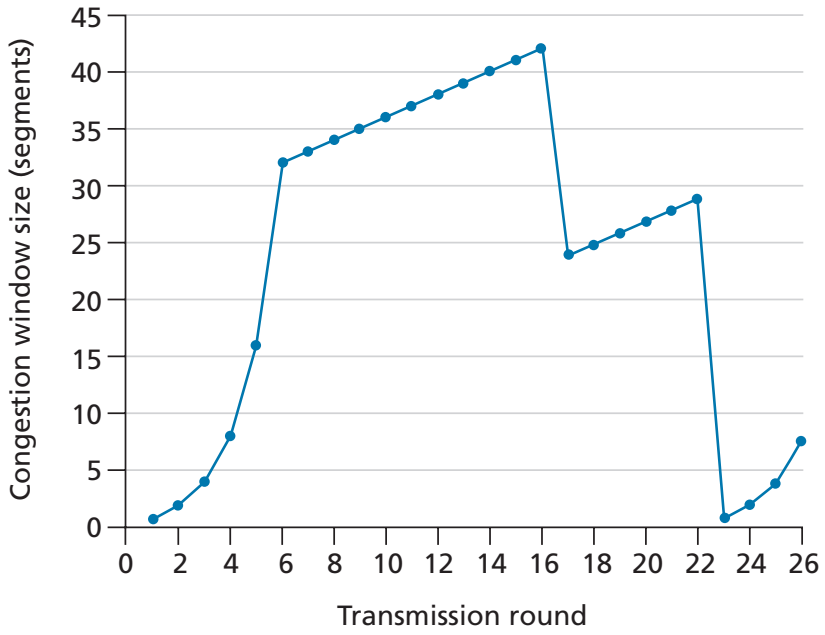


Figure 3.61 ♦ TCP window size as a function of time

h. During what transmission round is the 70th segment sent?

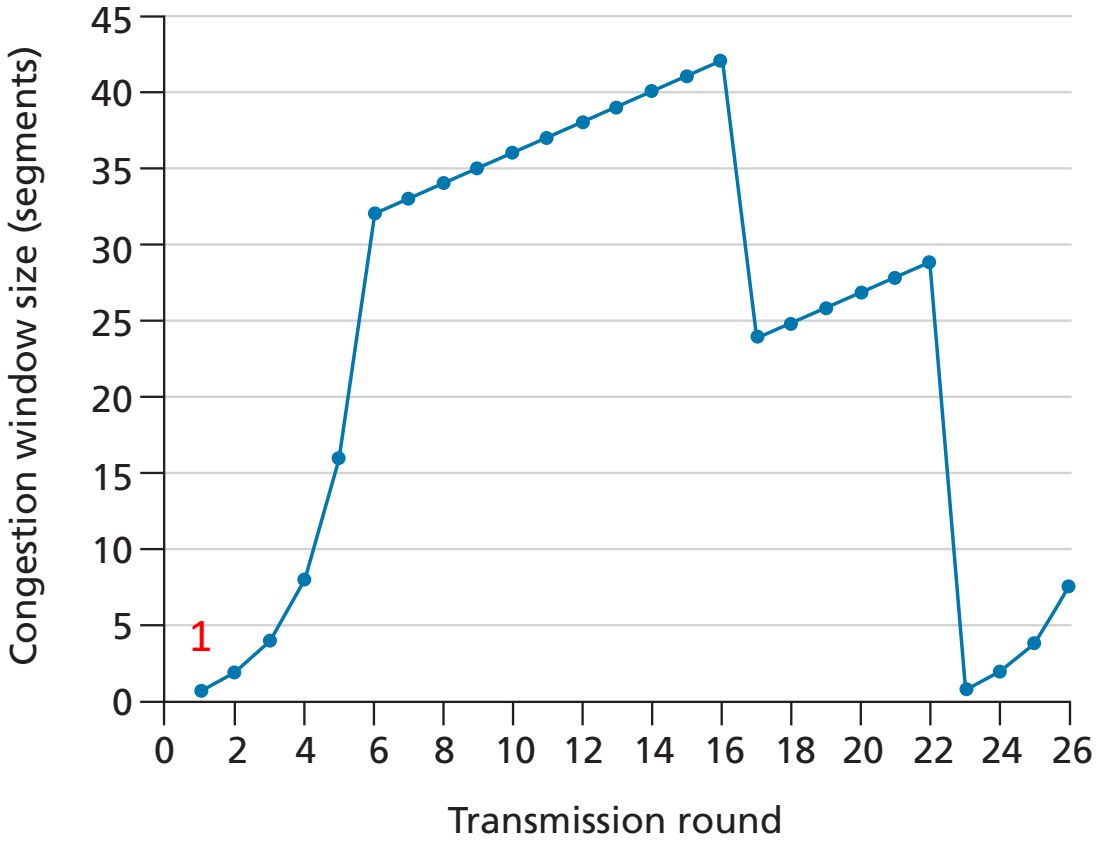


Figure 3.61 ♦ TCP window size as a function of time

h. During what transmission round is the 70th segment sent?

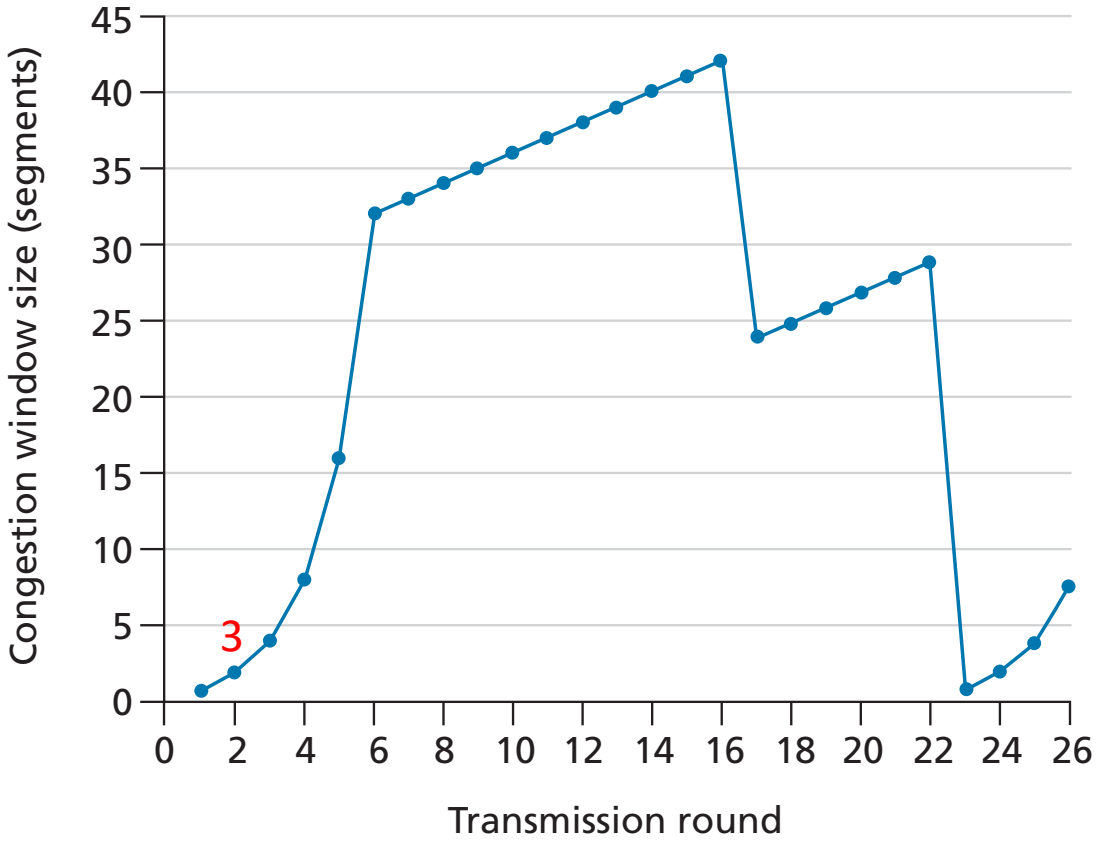


Figure 3.61 ♦ TCP window size as a function of time

h. During what transmission round is the 70th segment sent?

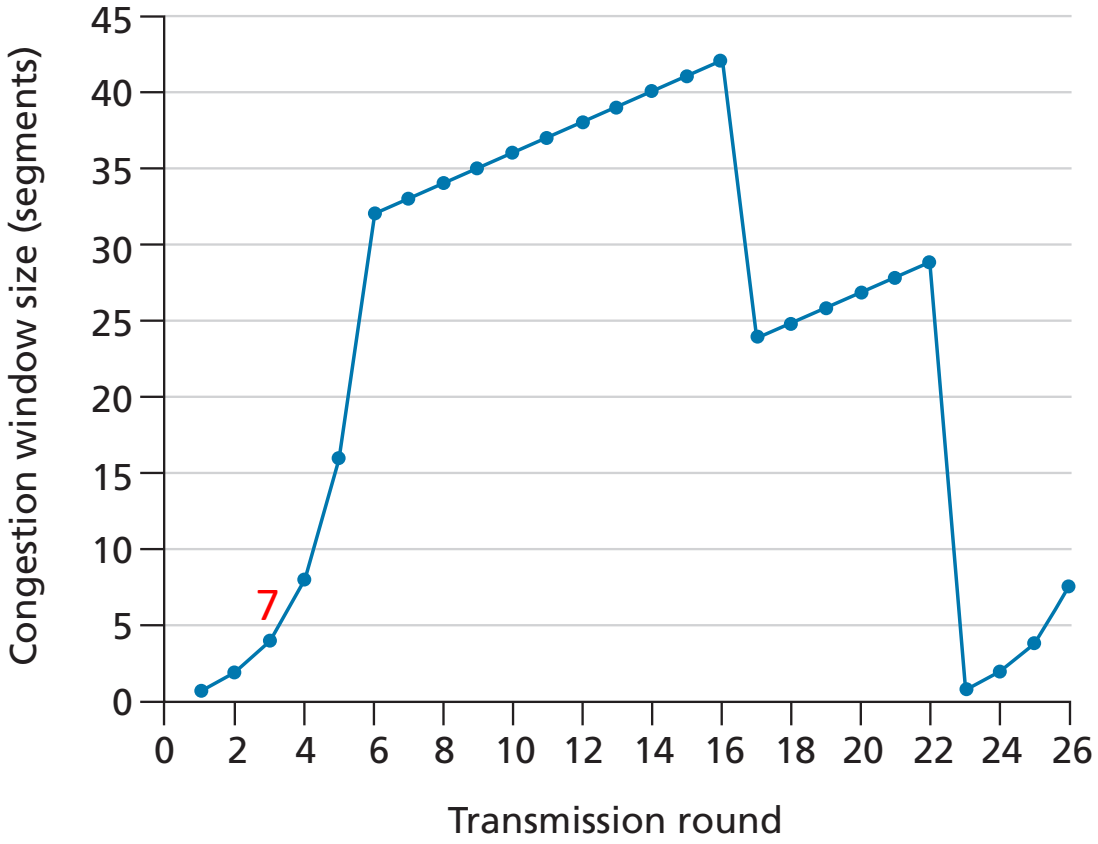


Figure 3.61 ♦ TCP window size as a function of time

h. During what transmission round is the 70th segment sent?

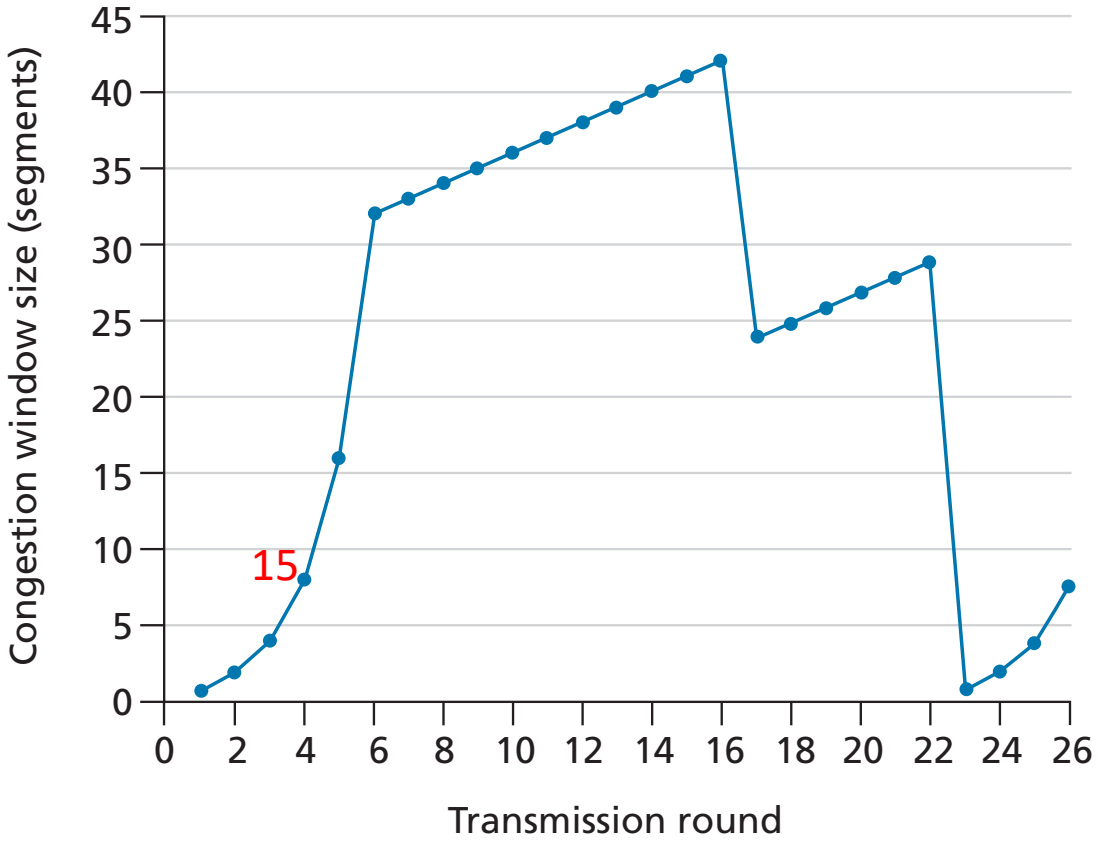


Figure 3.61 ♦ TCP window size as a function of time

h. During what transmission round is the 70th segment sent?

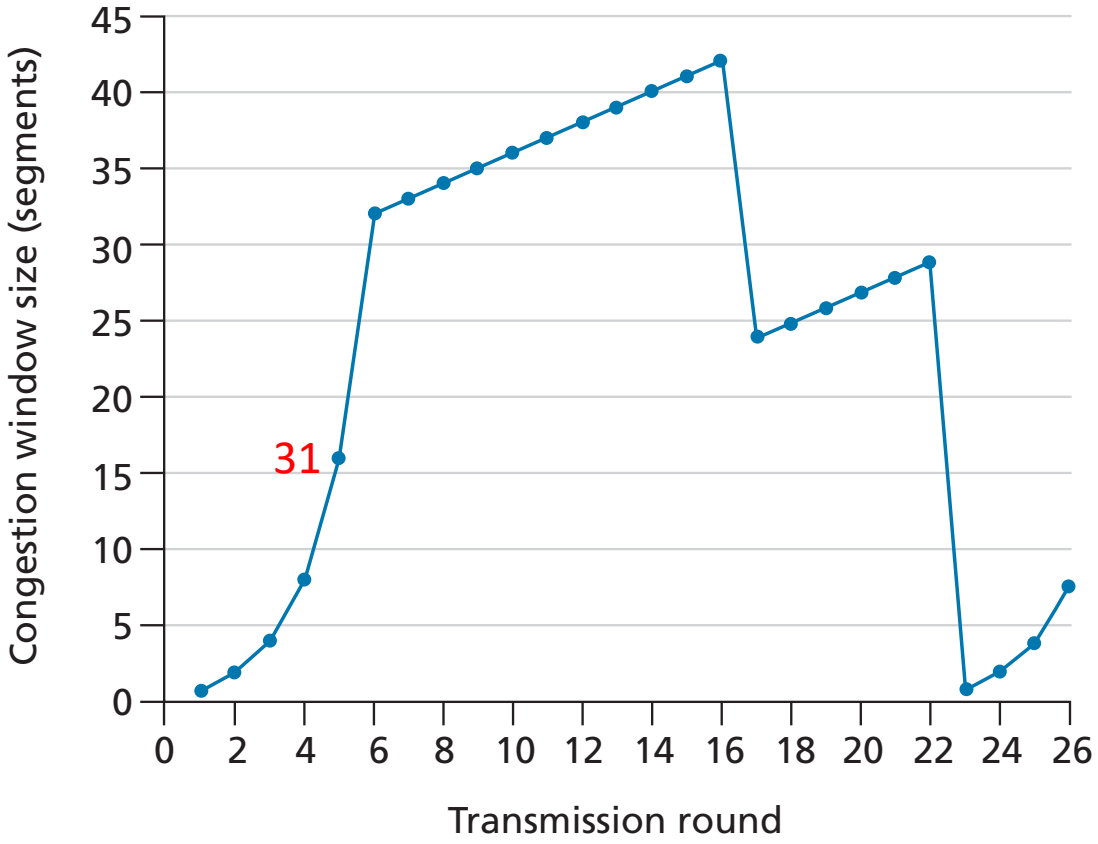


Figure 3.61 ♦ TCP window size as a function of time

h. During what transmission round is the 70th segment sent?

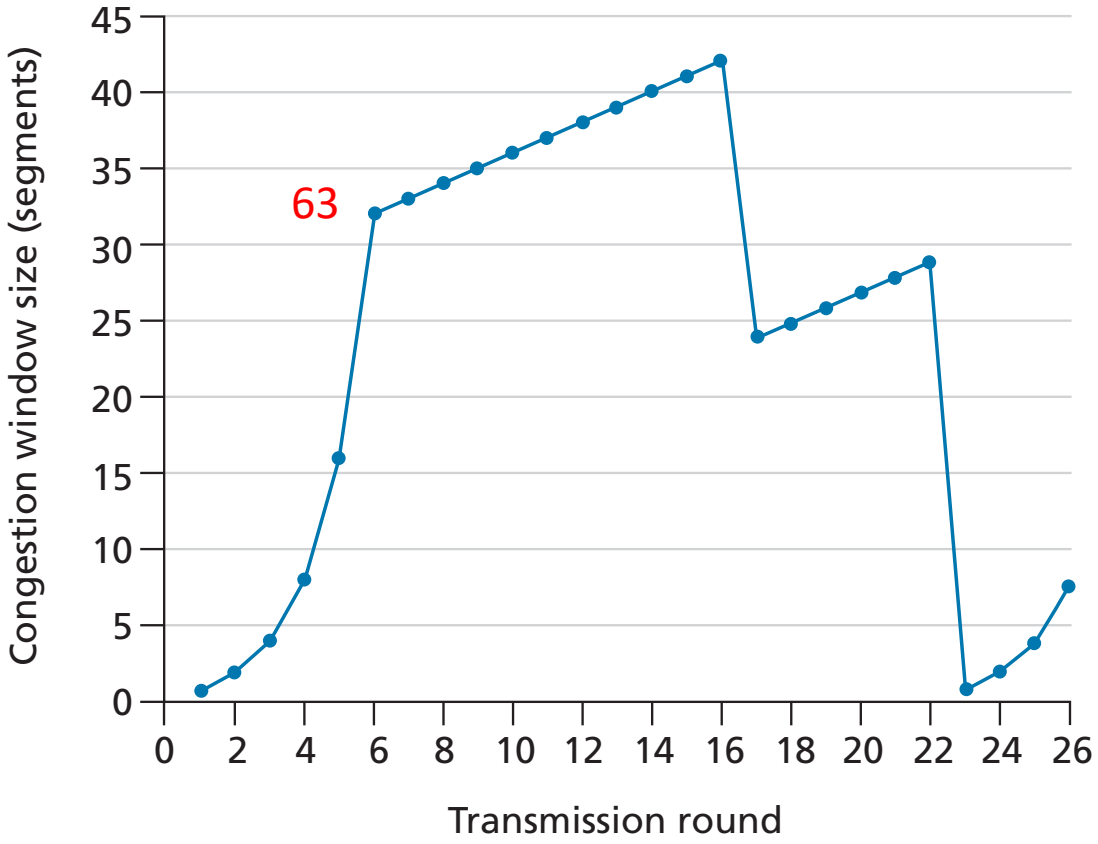


Figure 3.61 ♦ TCP window size as a function of time

h. During what transmission round is the 70th segment sent?

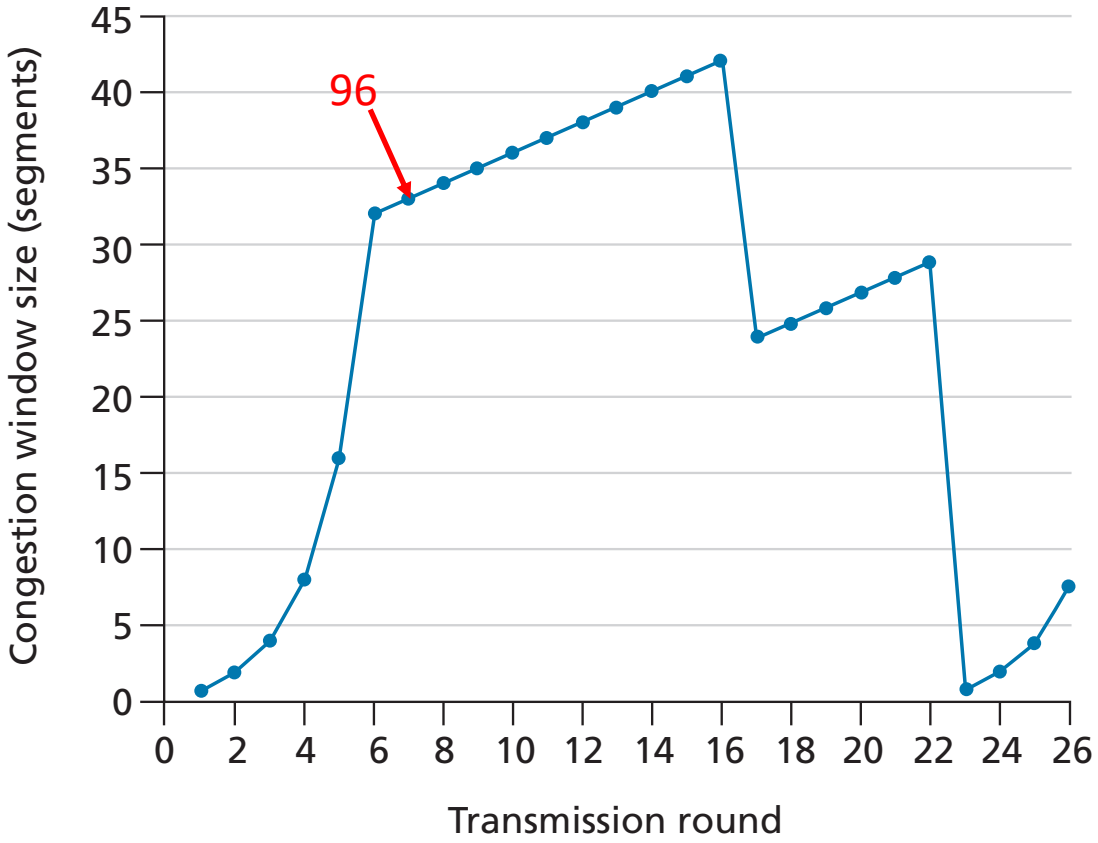


Figure 3.61 ♦ TCP window size as a function of time

Answer: The 70th segment is transferred at the 7th round

i. Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the congestion window size and of `ssthresh`?

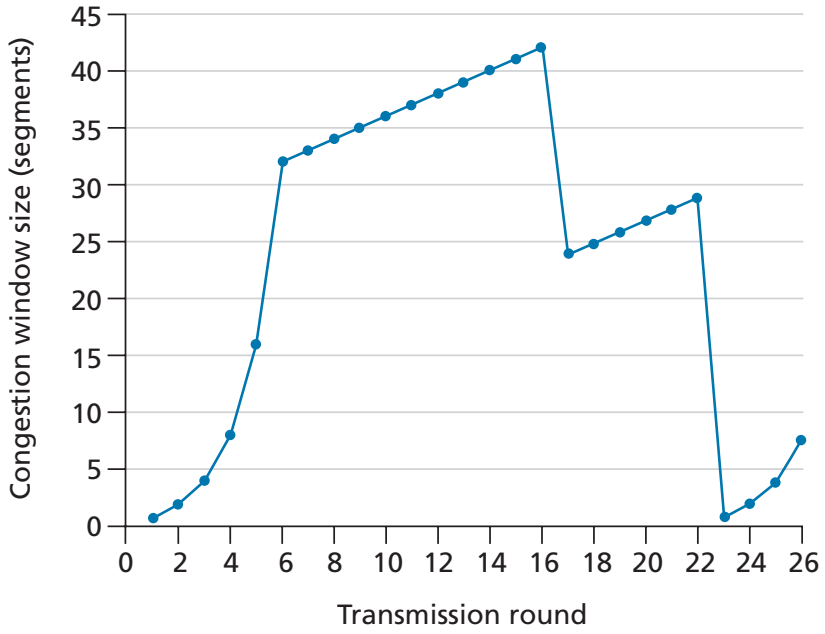


Figure 3.61 ♦ TCP window size as a function of time

i. Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the congestion window size and of `ssthresh`?

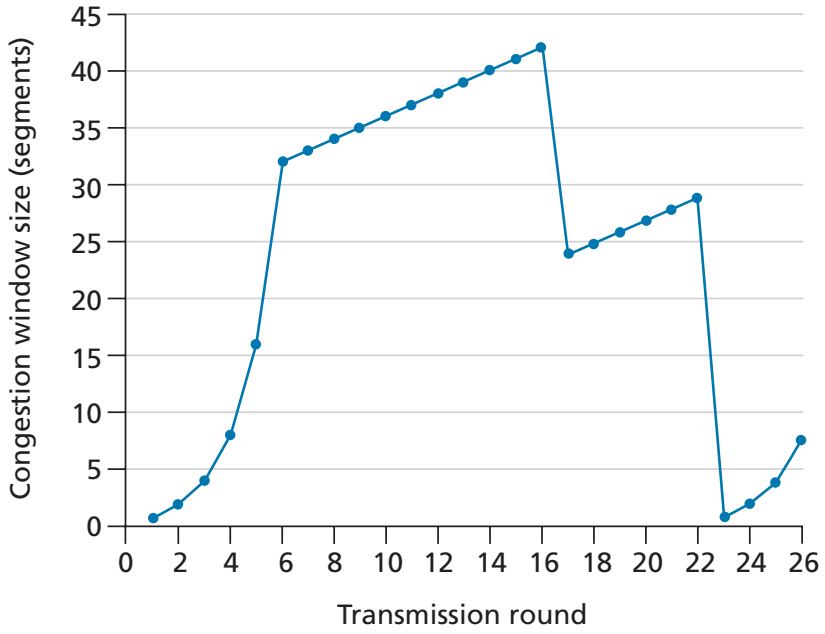


Figure 3.61 ♦ TCP window size as a function of time

Answer: ssthresh = cwnd/2 = 4, while cwnd'=ssthrsh + 3 =7.

j. Suppose TCP Tahoe is used (instead of TCP Reno), and assume that triple duplicate ACKs are received at the 16th round. What are the `ssthresh` and the congestion window size at the 19th round?

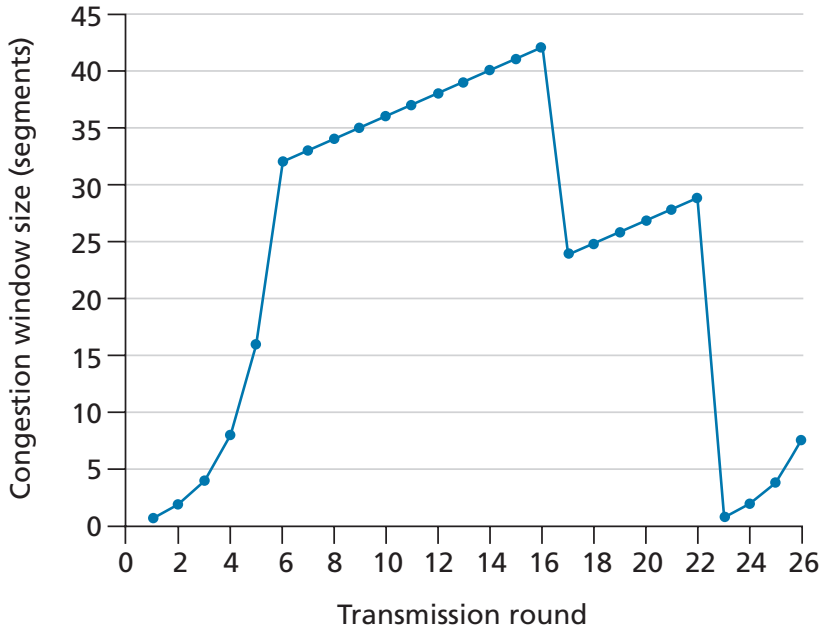


Figure 3.61 ♦ TCP window size as a function of time

j. Suppose TCP Tahoe is used (instead of TCP Reno), and assume that triple duplicate ACKs are received at the 16th round. What are the `ssthresh` and the congestion window size at the 19th round?

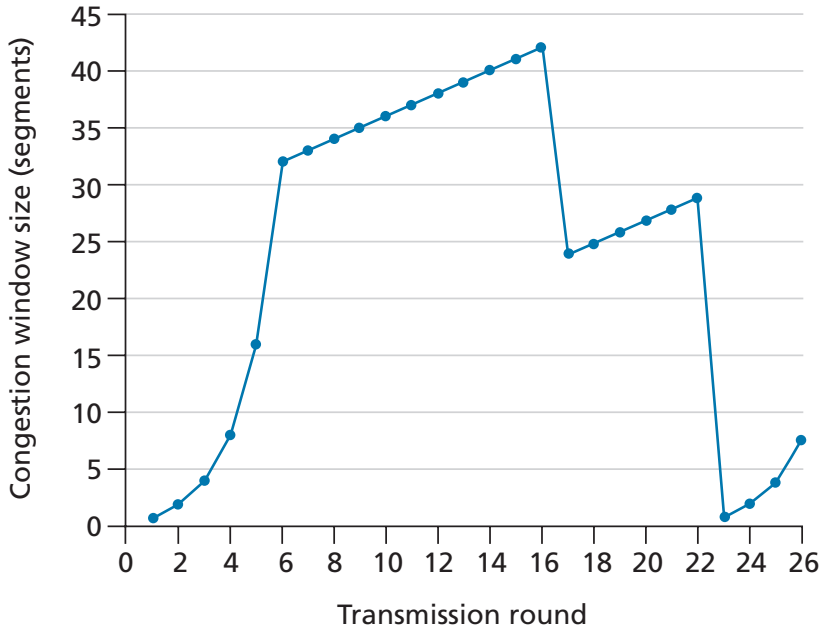


Figure 3.61 ♦ TCP window size as a function of time

Answer:
There is no fast recovery with TCP Tahoe, hence
ssthresh= cwnd/2 = 42/2=21, while the new cwnd'=1.

k. Again suppose TCP Tahoe is used, and there is a timeout event at 22nd round. How many packets have been sent out from 17th round till 22nd round, inclusive?

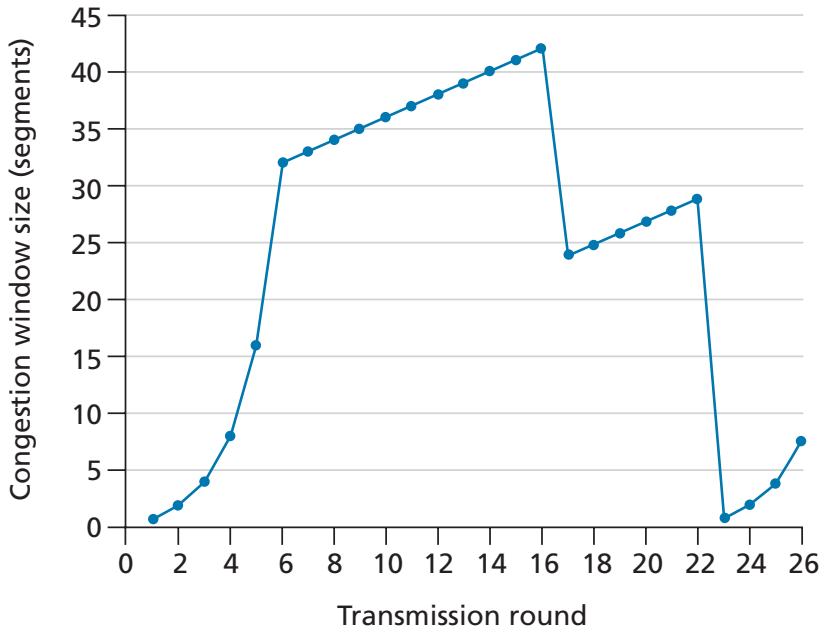


Figure 3.61 ♦ TCP window size as a function of time

k. Again suppose TCP Tahoe is used, and there is a timeout event at 22nd round. How many packets have been sent out from 17th round till 22nd round, inclusive?

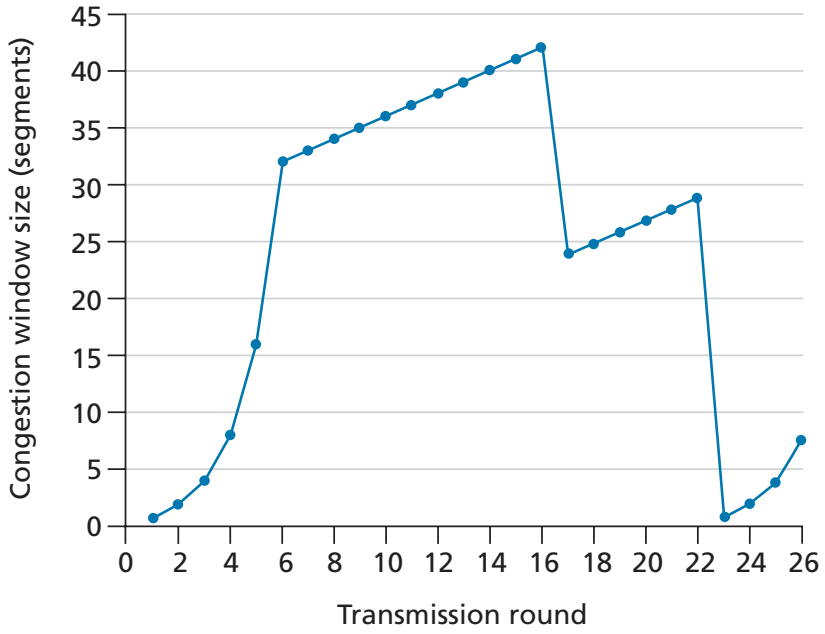


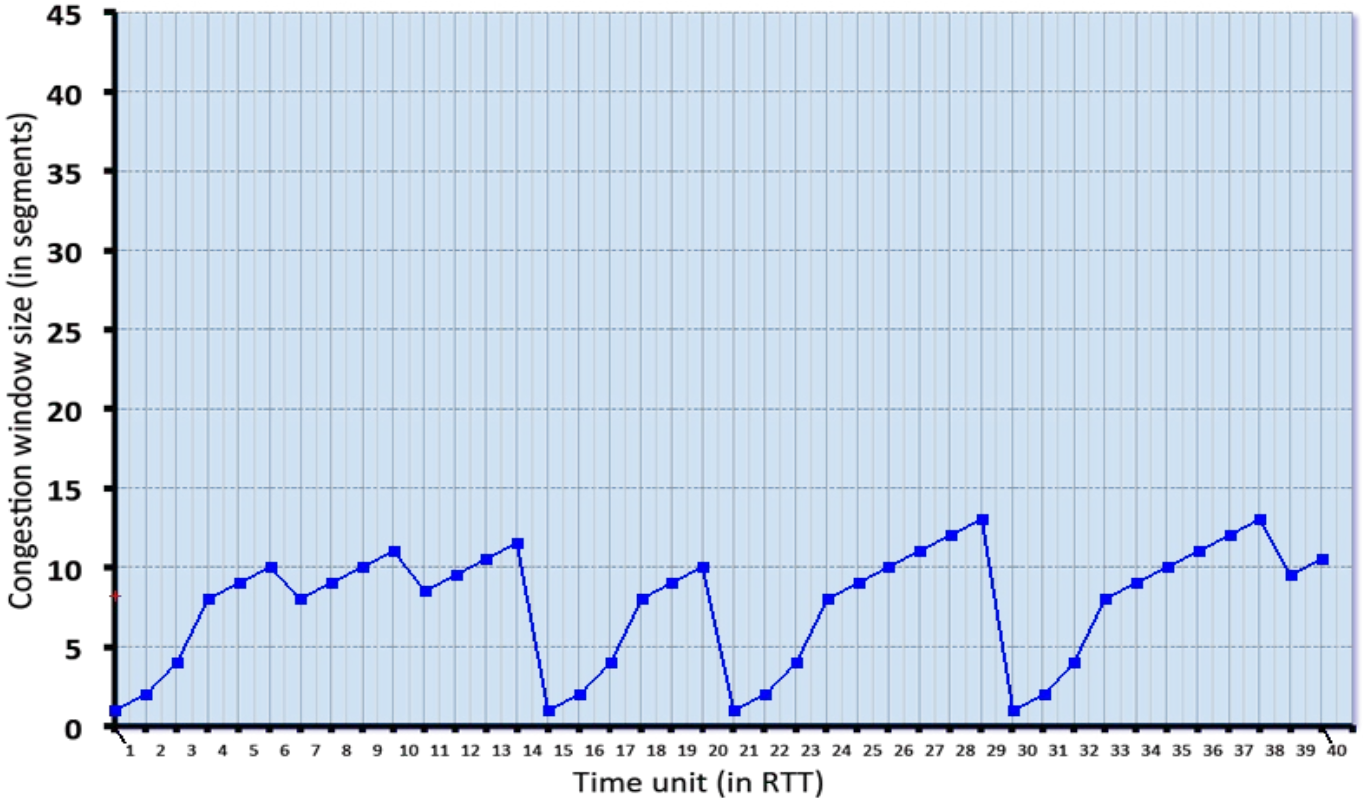
Figure 3.61 ♦ TCP window size as a function of time

Answer:

With TCP Tahoe at the 17th round we would be in slow start mode with ssthresh=21.

Hence In the 6 rounds from the 17th to the 22nd we will have 1+2+4+8+16+**21** = 52 packets.

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

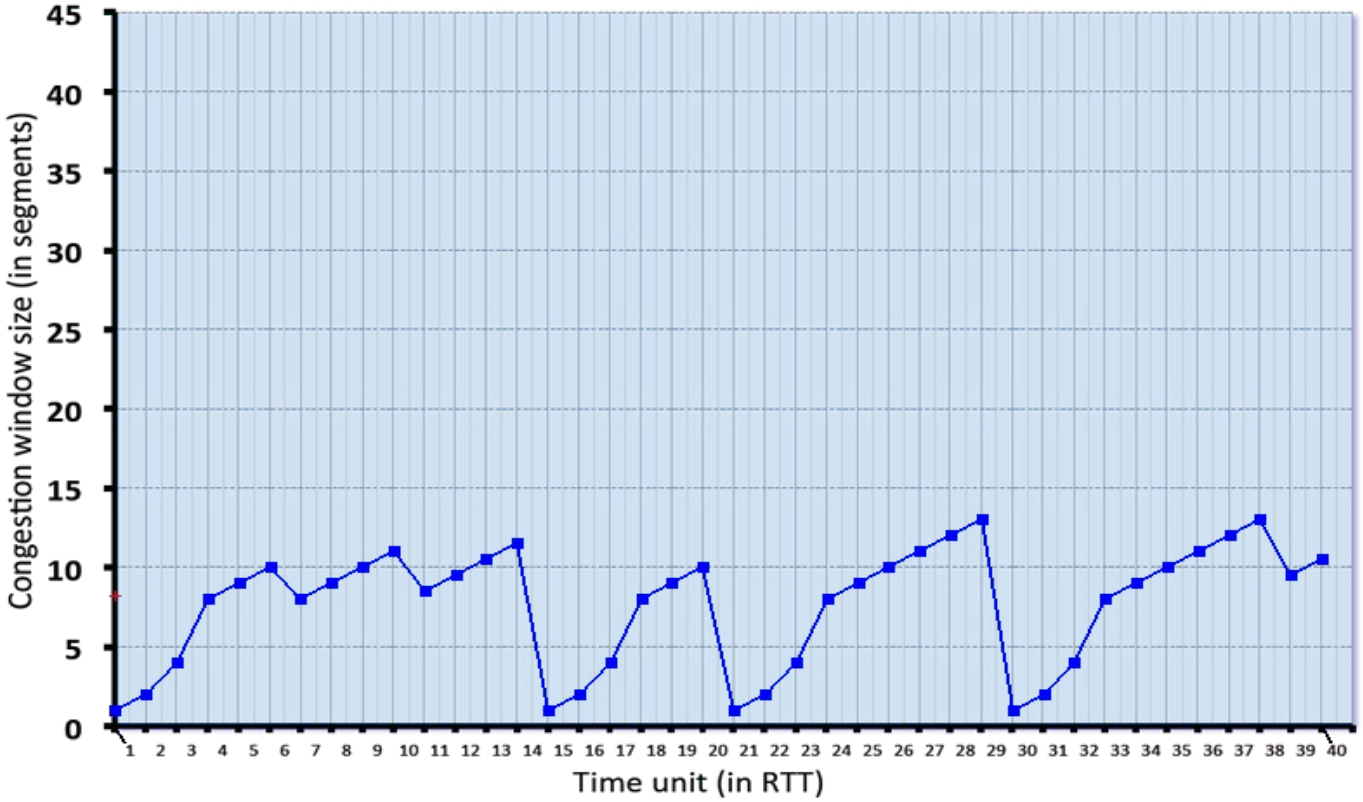
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

1. Give the times at which TCP is in slow start. Format your answer like: 1,3,5,9

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

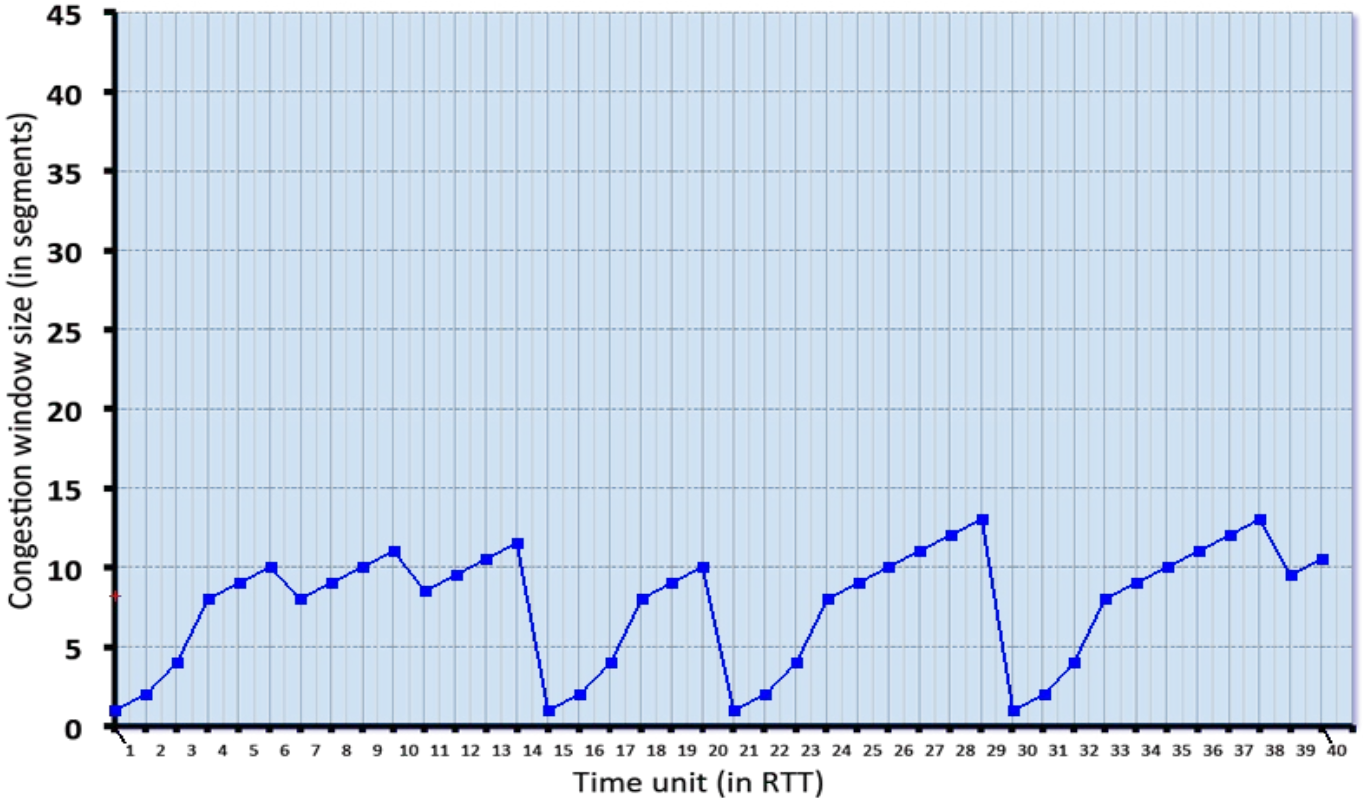
The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

1. Give the times at which TCP is in slow start. Format your answer like: 1,3,5,9

TCP is in slow start when starting or after a timeout, and below ssthresh

1,2,3,15,16,17,21,22,23,30,31,32

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

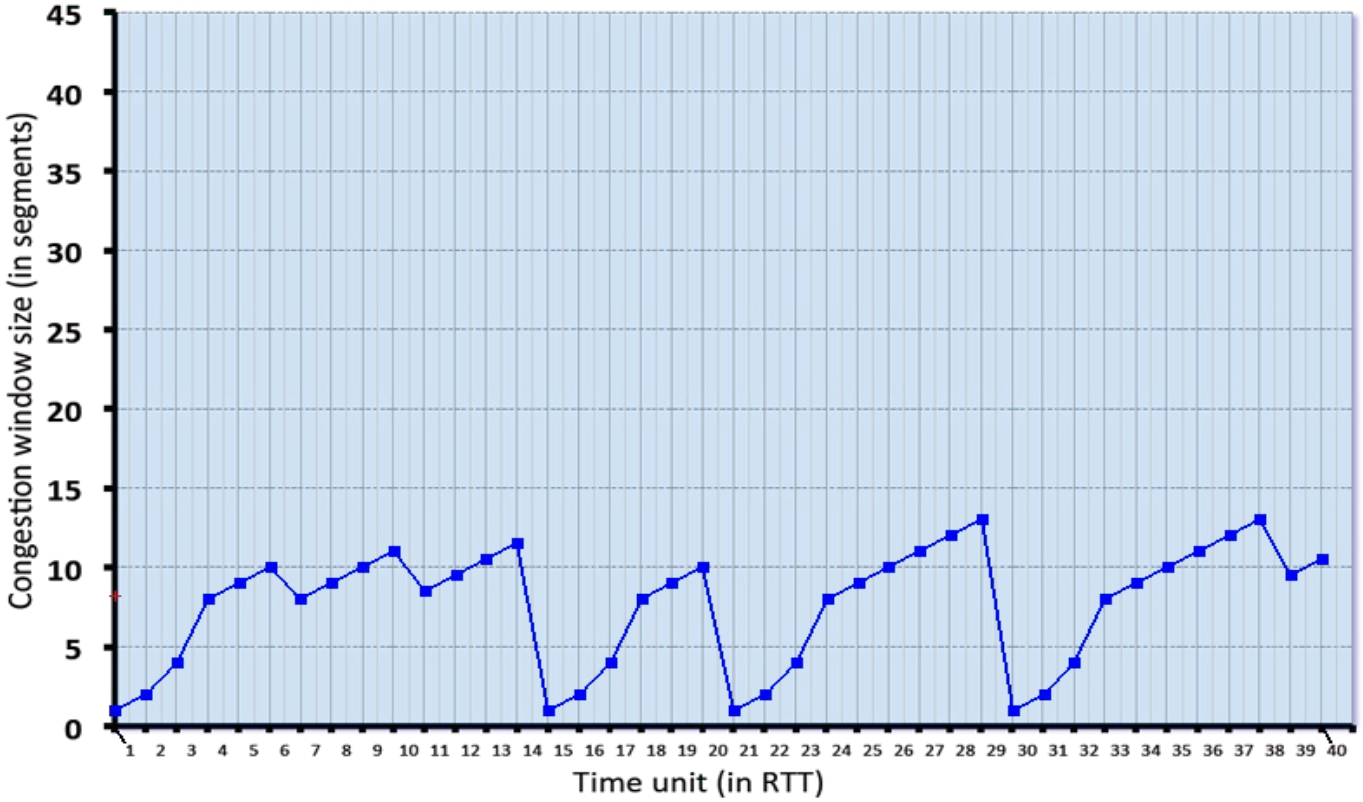
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

2. Give the times at which TCP is in congestion avoidance.

TCP in action

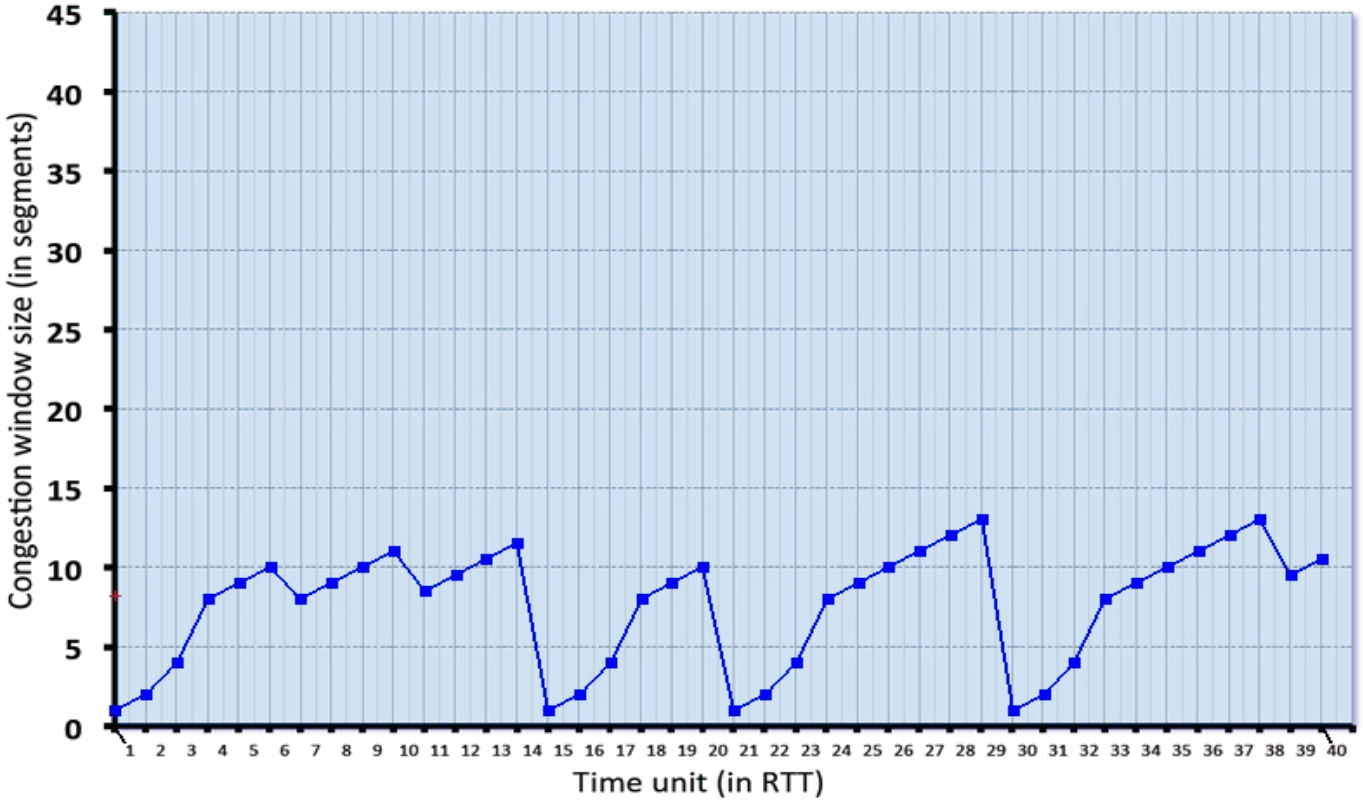


Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.
The result of sending that flight of packets is that either
***(i)* all packets are ACKed at the end of the interval, or**
***(ii)* there is a timeout for the first packet, or**
***(iii)* there is a triple duplicate ACK for the first packet**.
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.
The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

2. Give the times at which TCP is in congestion avoidance.

TCP is in congestion avoidance when it is above sshthresh. Therefore the answer is:
4,5,6,7,8,9,10,11,12,13,18,19,24,25,26,27,28,33,34,35,36,37,38,39

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

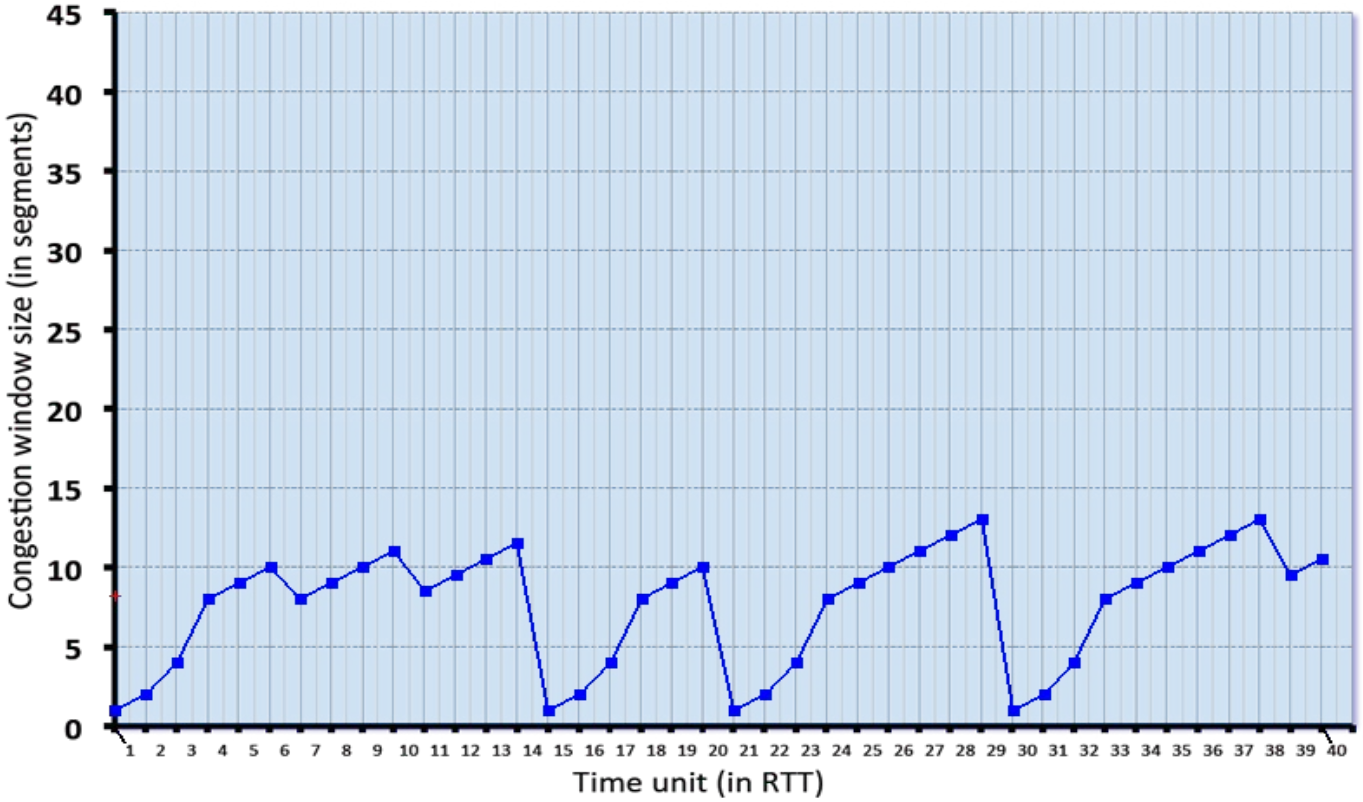
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

3. Give the times at which TCP is in fast recovery.

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

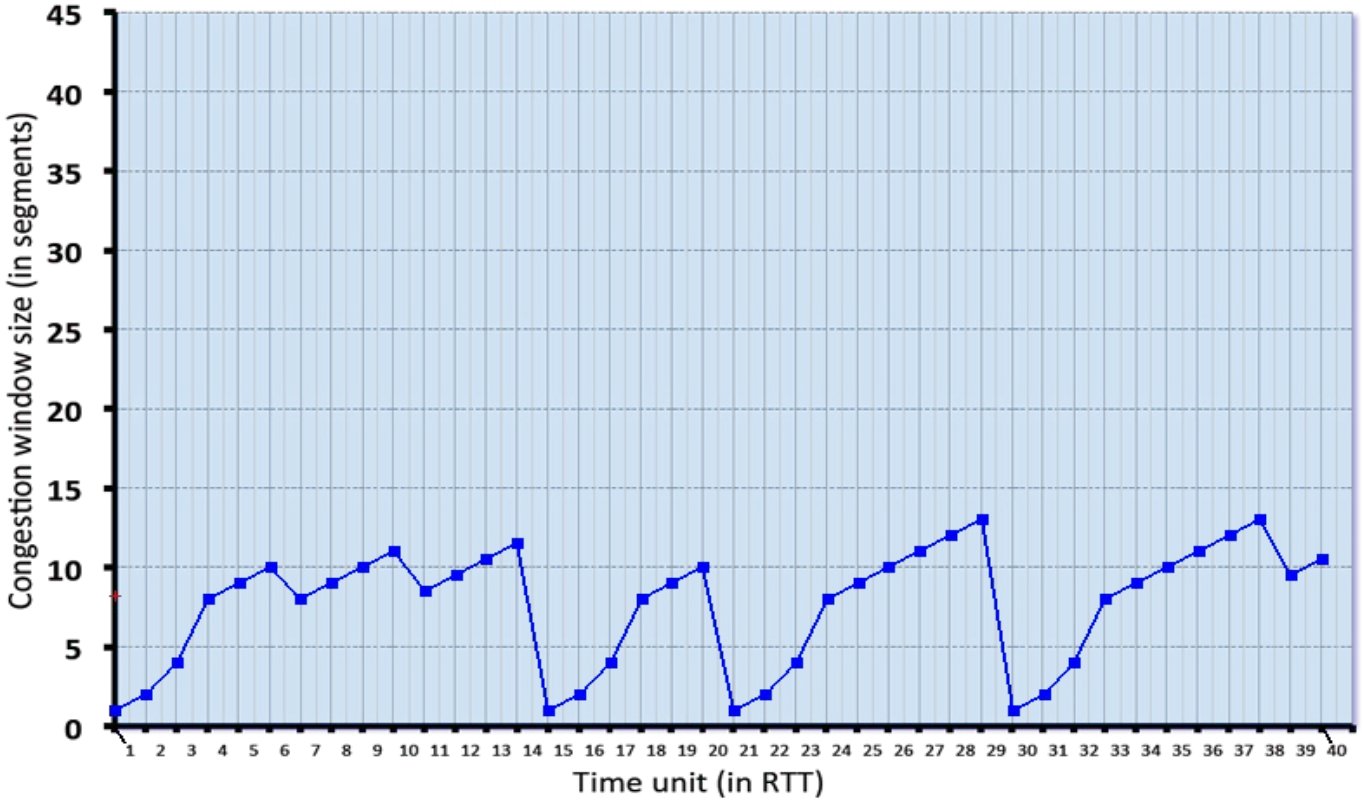
The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

3. Give the times at which TCP is in fast recovery.

TCP is in fast recovery after a loss due to triple duplicate ACK and after the first time when below ssthresh

7,11,39

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

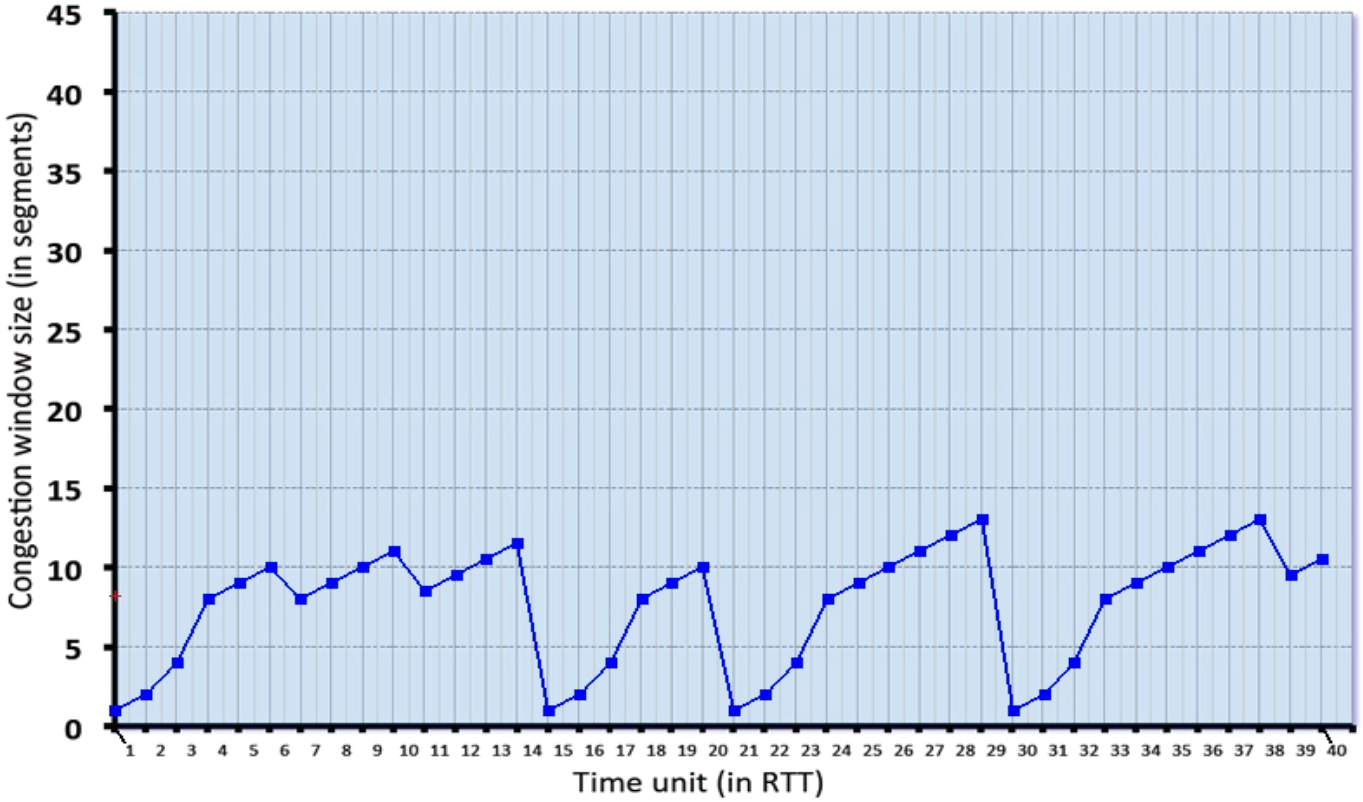
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

4. Give the times at which packets are lost via timeout.

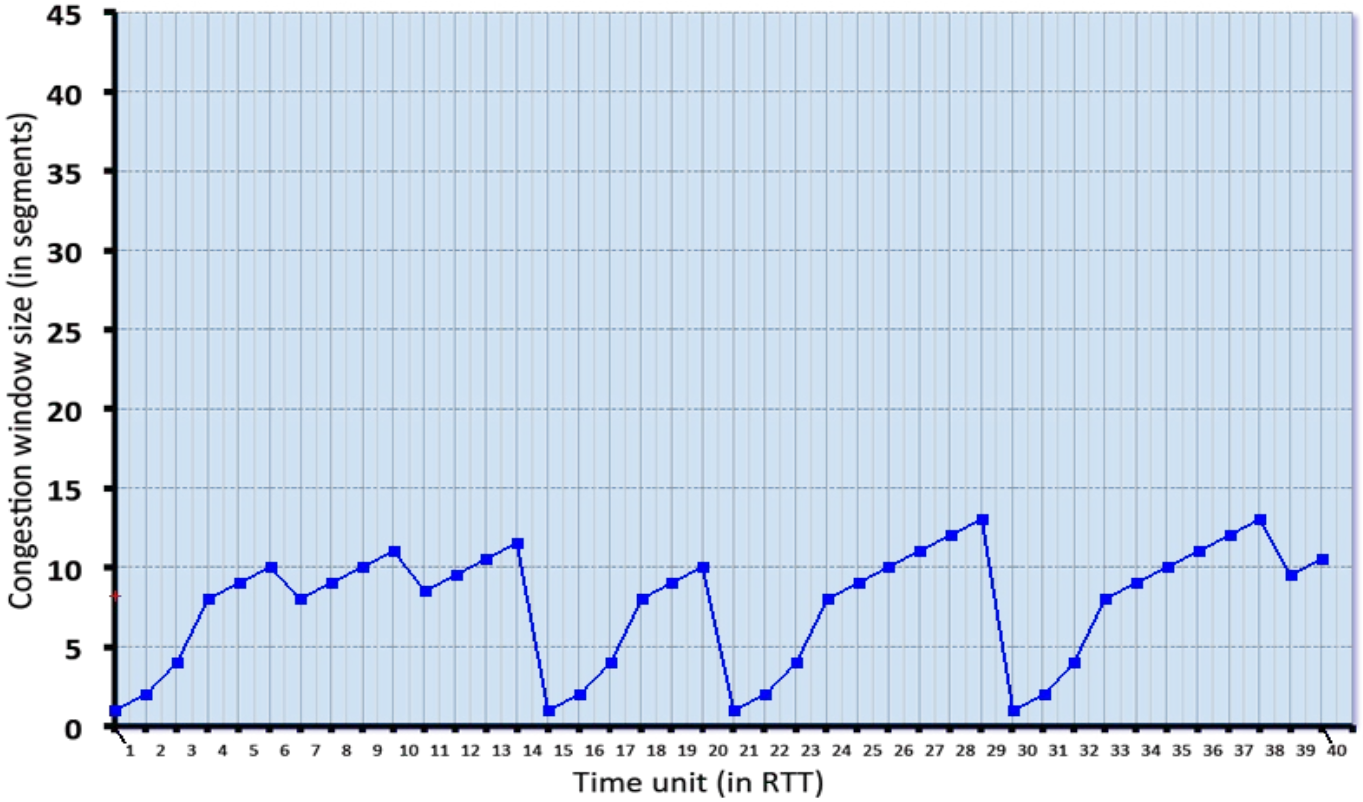
TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.
The result of sending that flight of packets is that either
***(i)* all packets are ACKed at the end of the interval, or**
***(ii)* there is a timeout for the first packet, or**
***(iii)* there is a triple duplicate ACK for the first packet**.
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.
The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

4. Give the times at which packets are lost via timeout.
A timeout loss is characterized by a drop to cwnd=1, hence the answer is:
14,20,29

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

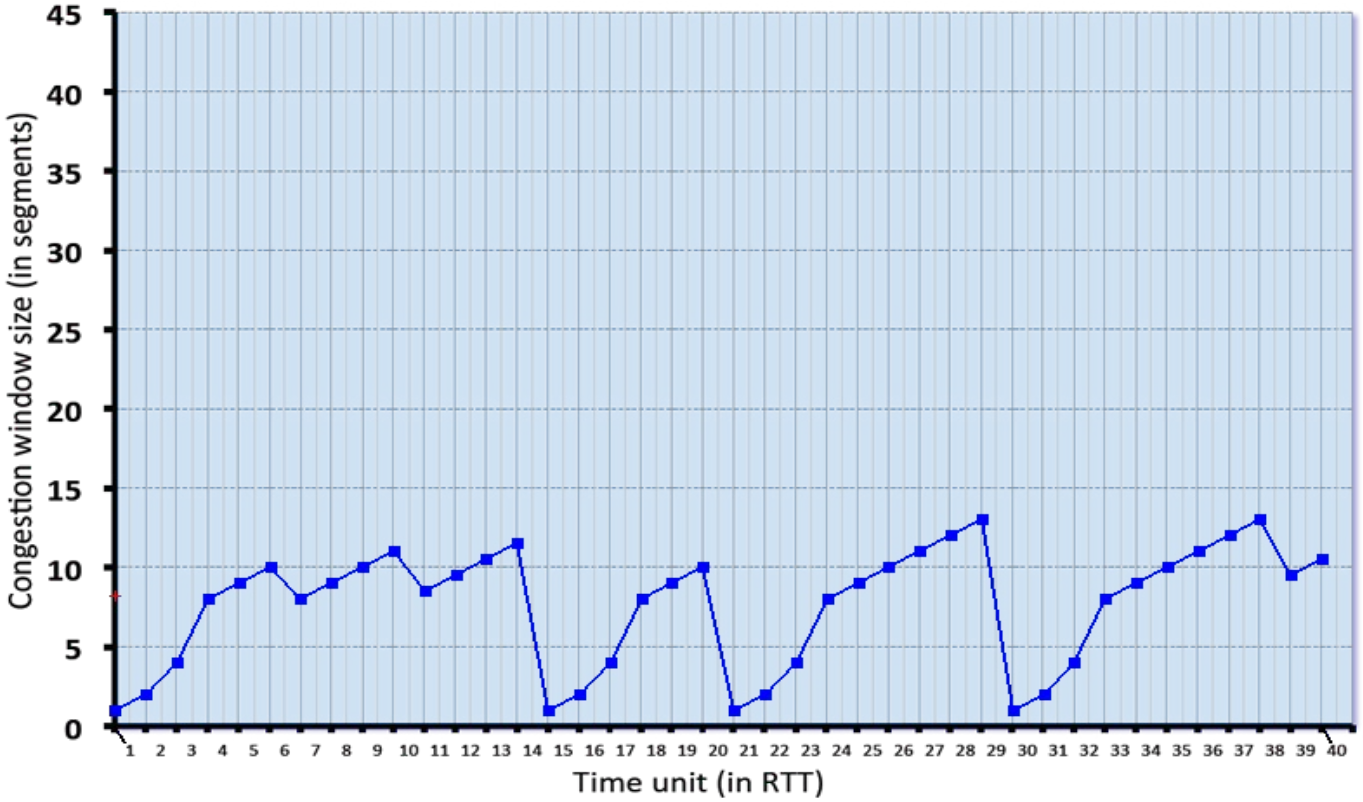
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

5. Give the times at which packets are lost via *triple ACK.*

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

***(iii)* there is a triple duplicate ACK for the first packet**.

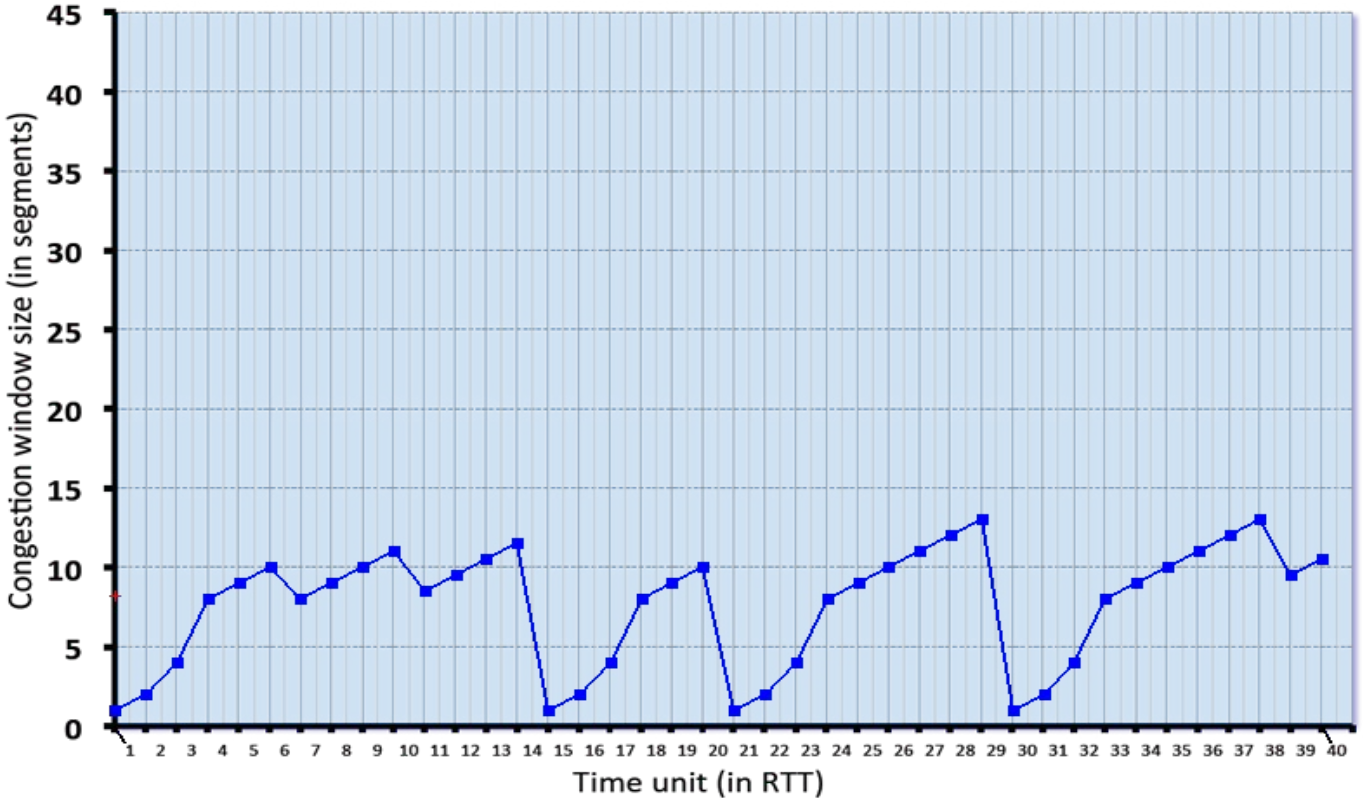
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

5. Give the times at which packets are lost via *triple ACK.*

6,10,38

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

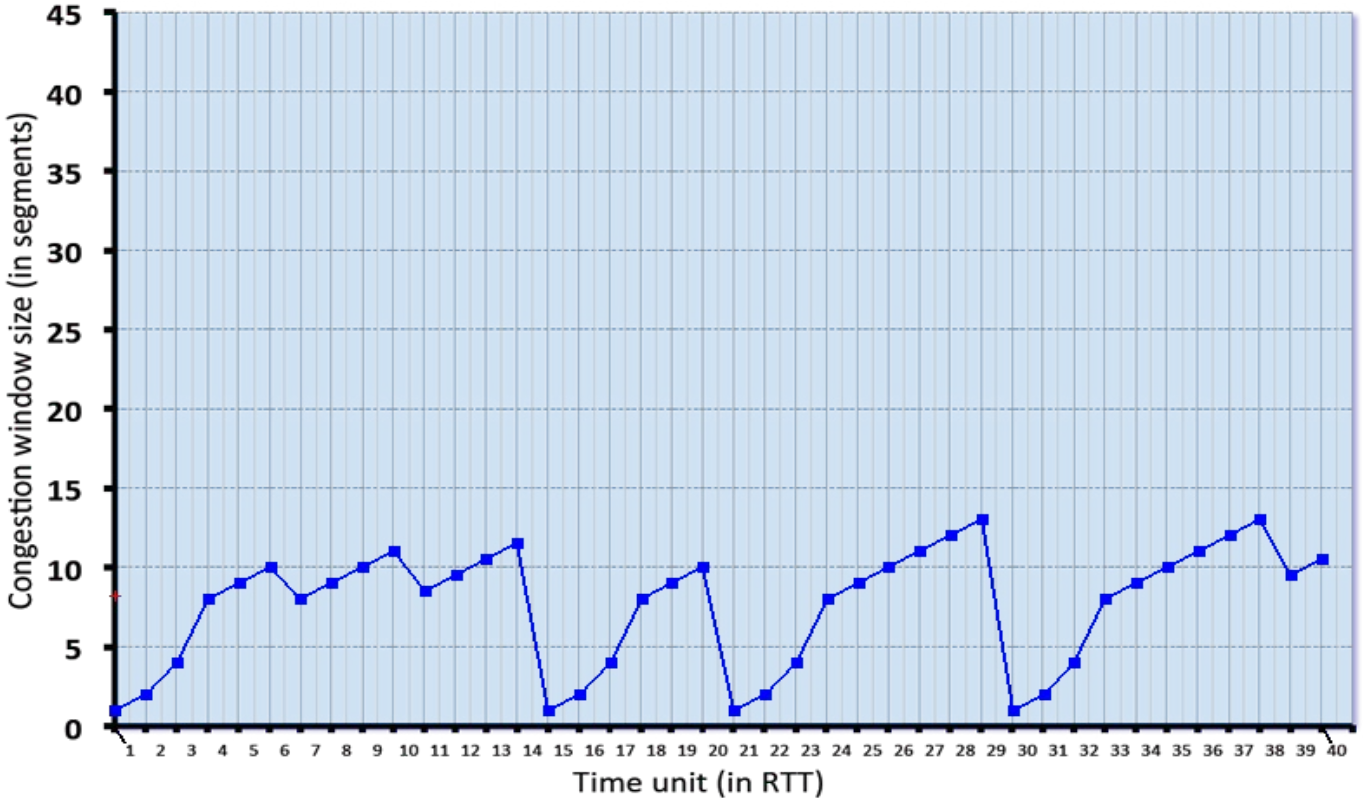
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

6. Give the times at which the value of *ssthresh* changes (if it changes between t=3 and t=4, use t=4 in your answer)

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

***(iii)* there is a triple duplicate ACK for the first packet**.

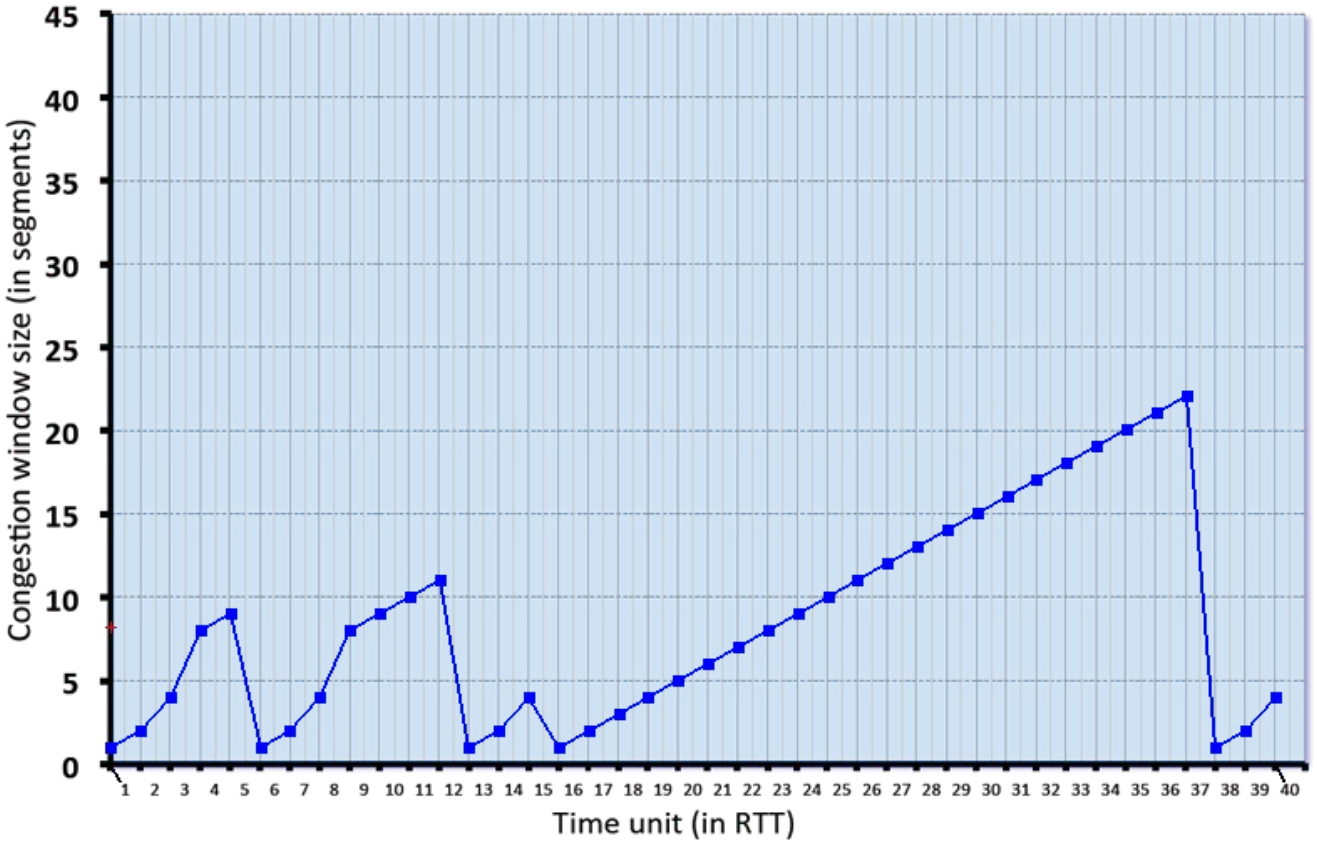
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

6. Give the times at which the value of *ssthresh* changes (if it changes between t=3 and t=4, use t=4 in your answer)

The ssthresh can change when there is a loss or triple ack, hence the answer is 7,11,15,21,30,39

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

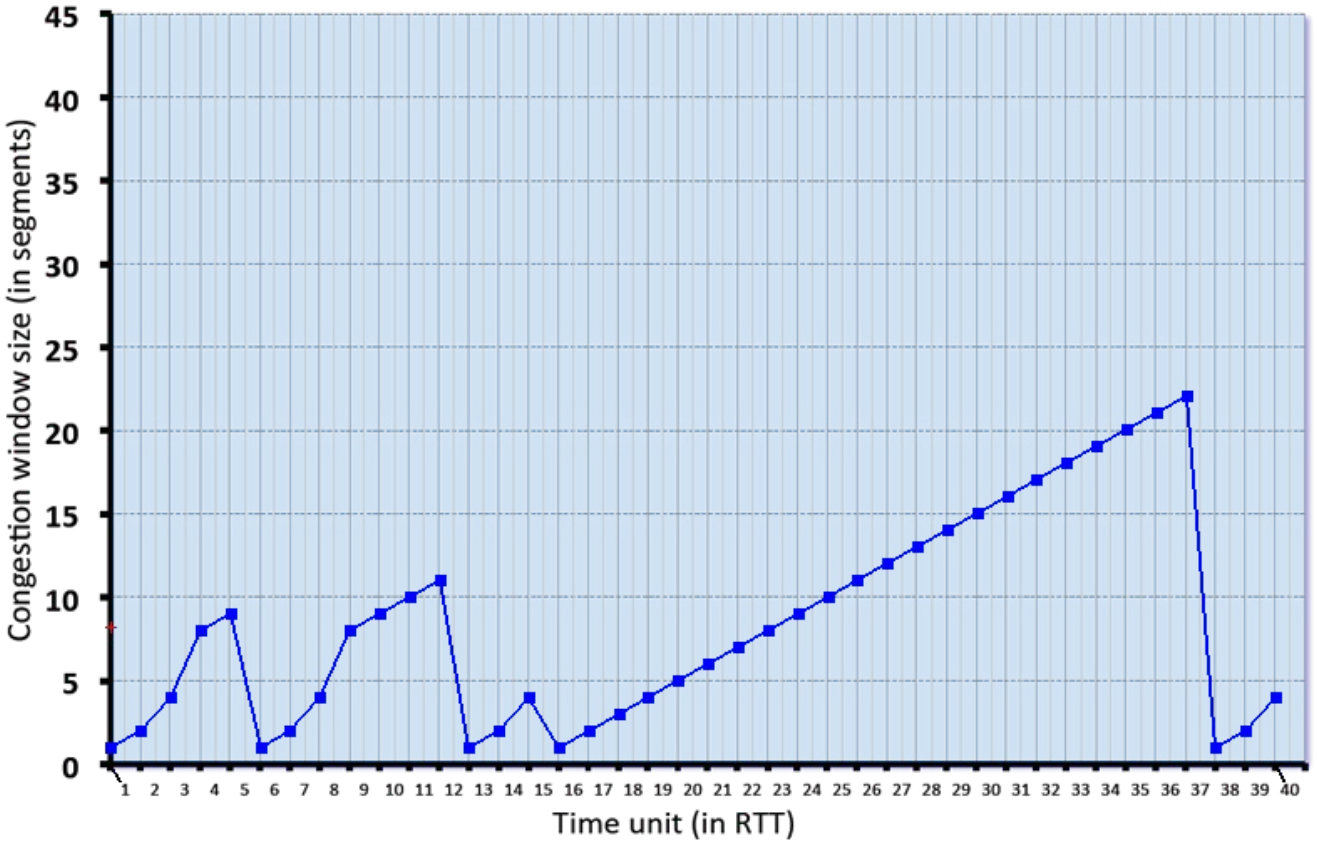
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

1. Give the times at which TCP is in slow start. Format your answer like: 1,3,5,9

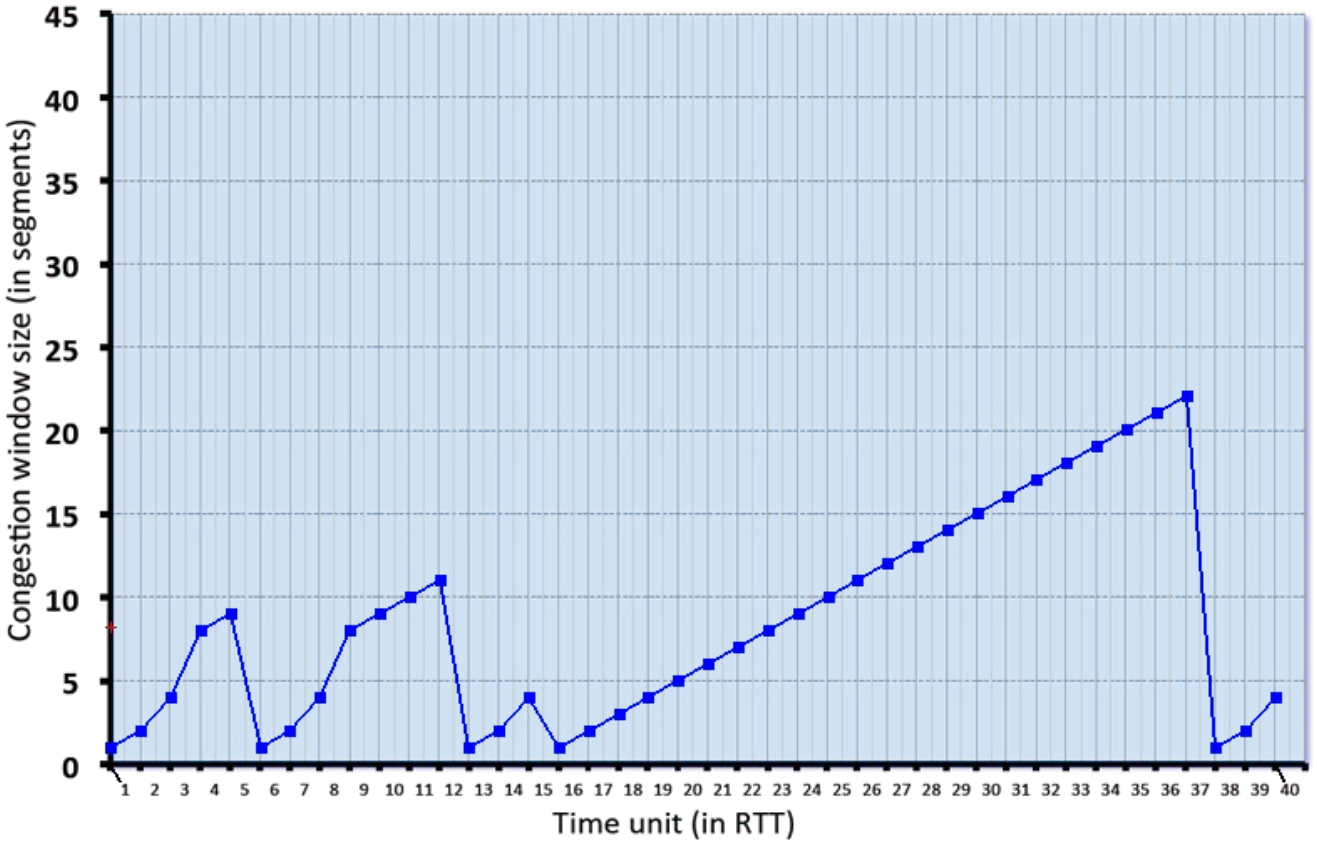
TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.
The result of sending that flight of packets is that either
***(i)* all packets are ACKed at the end of the interval, or**
***(ii)* there is a timeout for the first packet, or**
***(iii)* there is a triple duplicate ACK for the first packet**.
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.
The initial value of *cwnd* is 1, and the initial value of *ssthresh* is 8.

1. Give the times at which TCP is in slow start. Format your answer like: 1,3,5,9
TCP is in slow start when starting or after a timeout, and below ssthresh
1,2,3,6,7,8,13,14,15,16,38,39,40

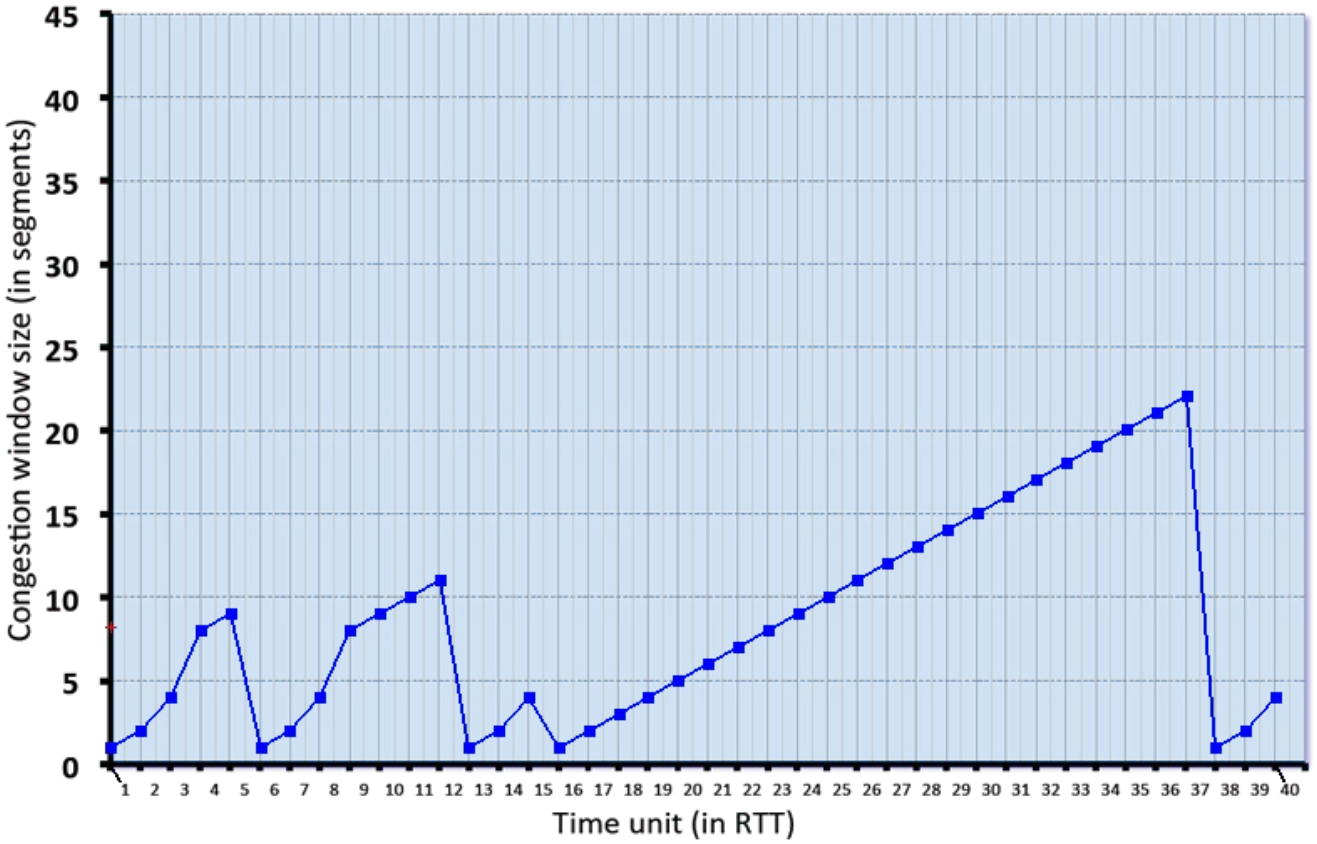
TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.
The result of sending that flight of packets is that either
***(i)* all packets are ACKed at the end of the interval, or**
***(ii)* there is a timeout for the first packet, or**
***(iii)* there is a triple duplicate ACK for the first packet**.
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.
The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

2. Give the times at which TCP is in congestion avoidance.

TCP in action

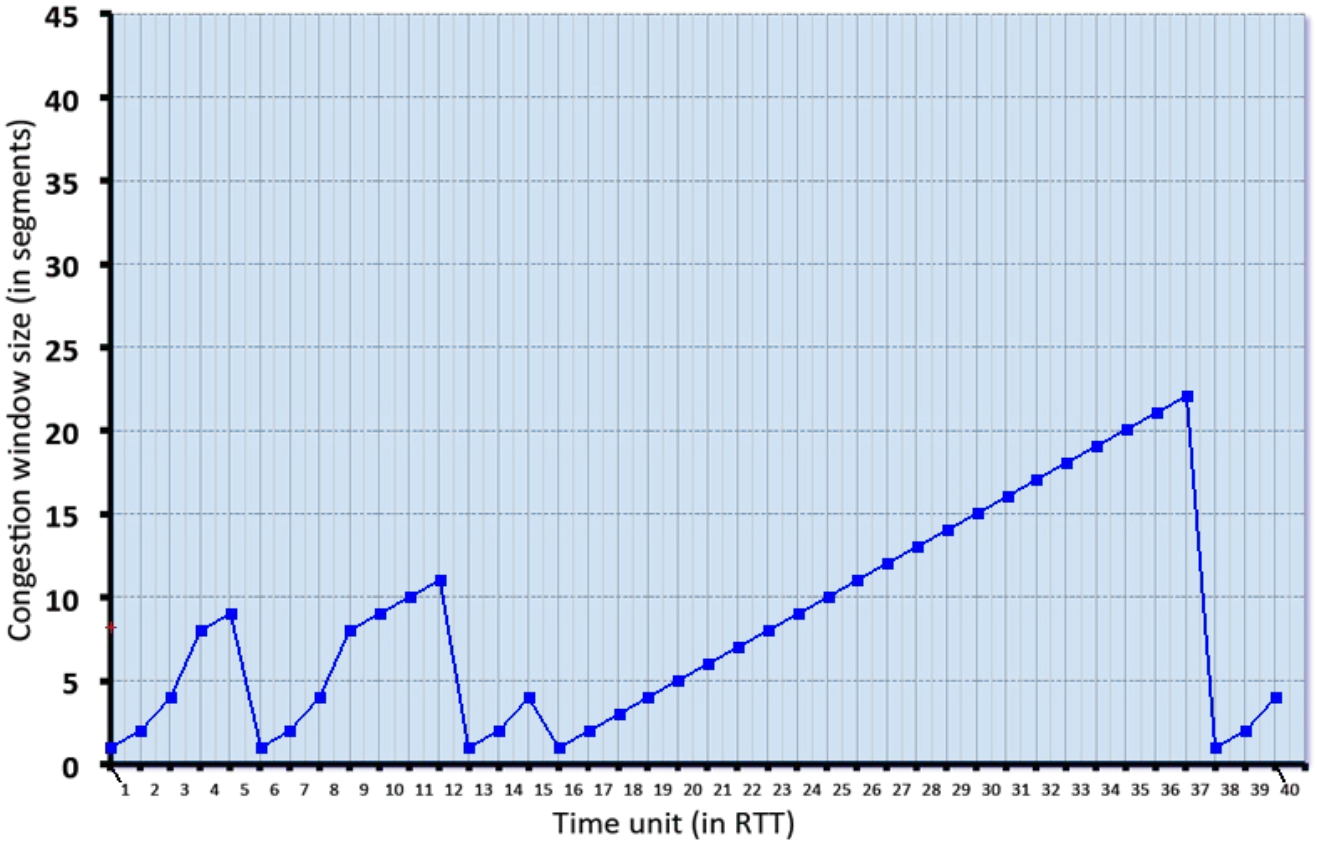


Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.
The result of sending that flight of packets is that either
***(i)* all packets are ACKed at the end of the interval, or**
***(ii)* there is a timeout for the first packet, or**
***(iii)* there is a triple duplicate ACK for the first packet**.
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.
The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

2. Give the times at which TCP is in congestion avoidance.

TCP is in congestion avoidance when it is above sshthresh. Therefore the answer is:
4,5,9,10,11,12,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

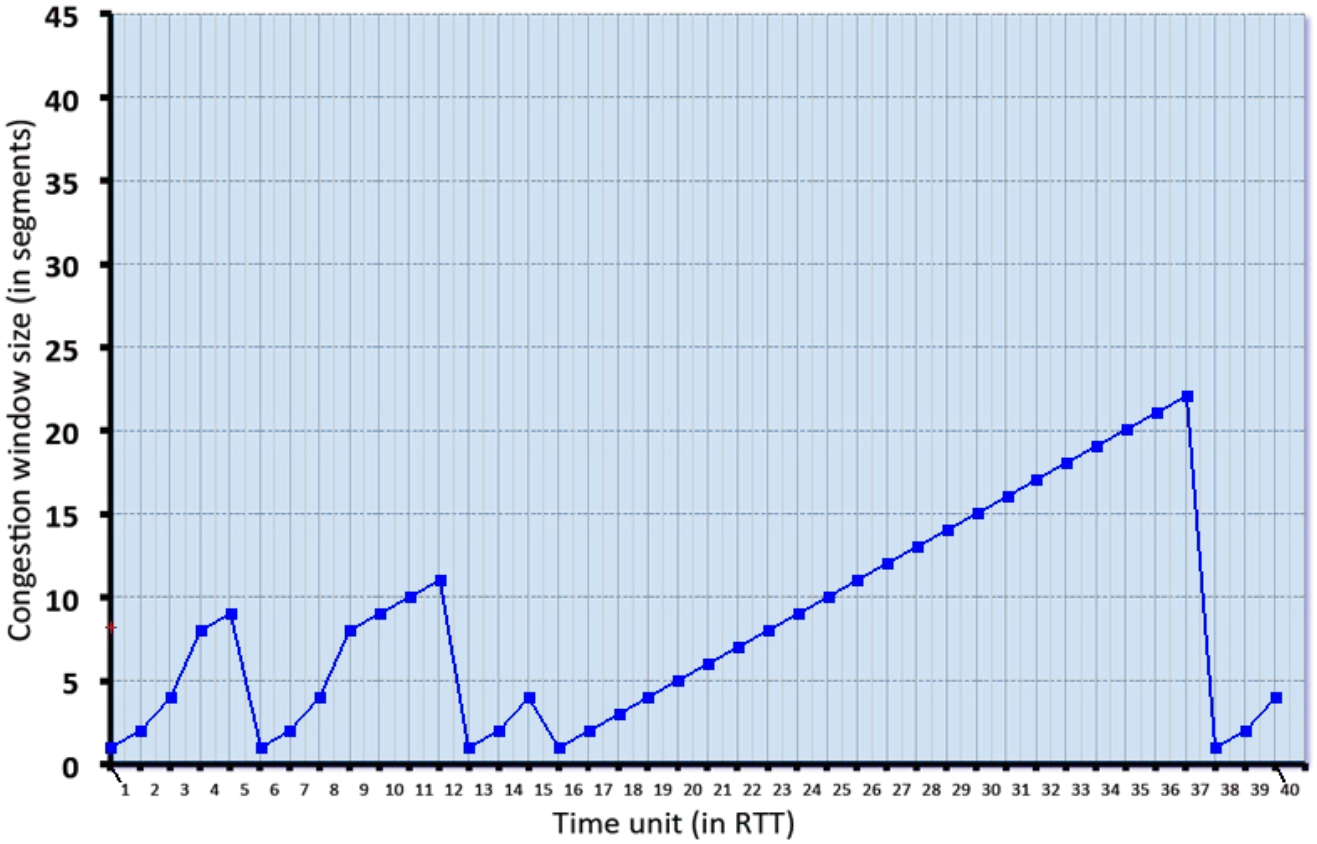
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

3. Give the times at which TCP is in fast recovery.

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

***(iii)* there is a triple duplicate ACK for the first packet**.

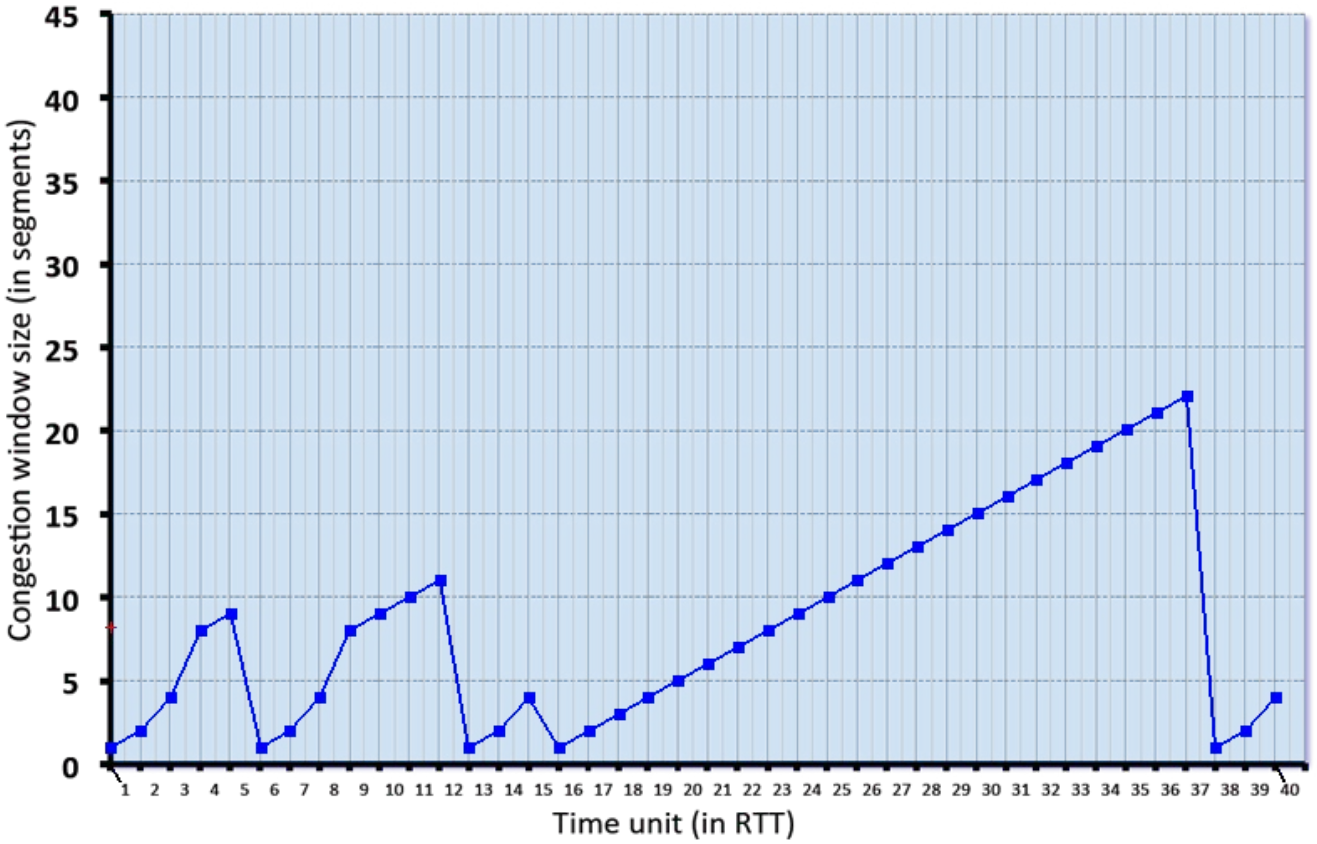
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

3. Give the times at which TCP is in fast recovery.

TCP is in fast recovery after a loss due to triple duplicate ACK and after the first time when below ssthresh:
None here

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

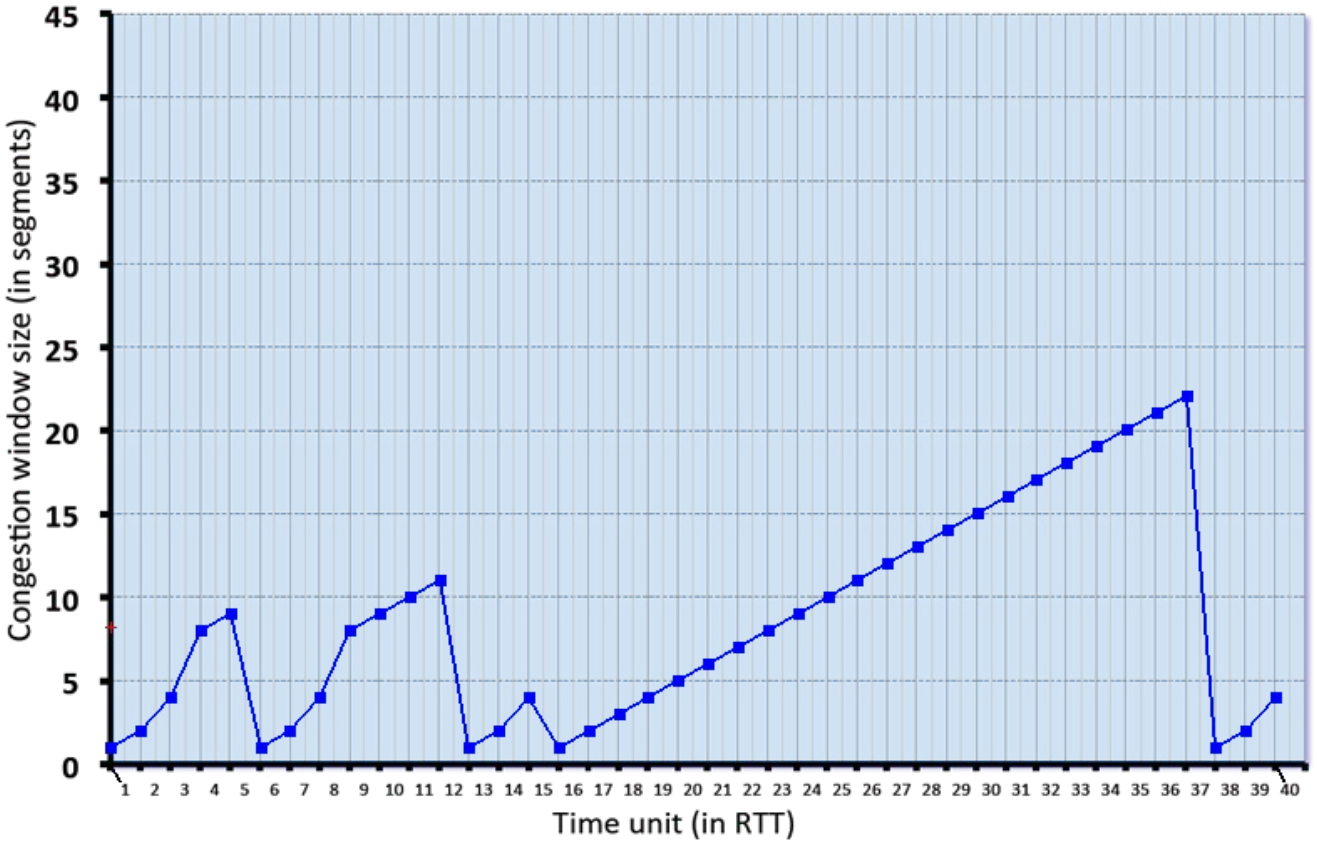
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

4. Give the times at which packets are lost via timeout.

TCP in action

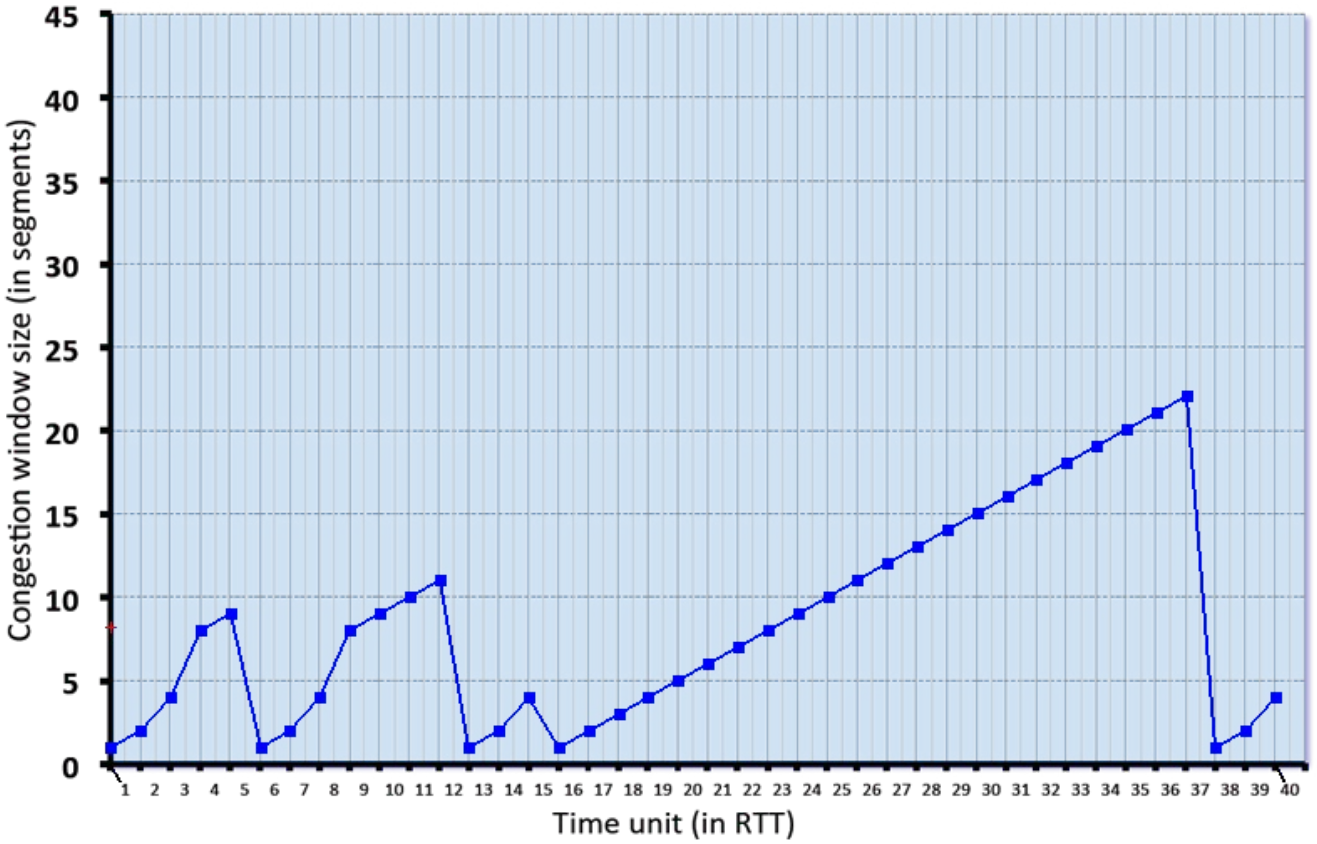


Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.
The result of sending that flight of packets is that either
***(i)* all packets are ACKed at the end of the interval, or**
***(ii)* there is a timeout for the first packet, or**
***(iii)* there is a triple duplicate ACK for the first packet**.
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.
The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

4. Give the times at which packets are lost via timeout.

A timeout loss is characterized by a drop to cwnd=1, hence the answer is:
5,12,15,37

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

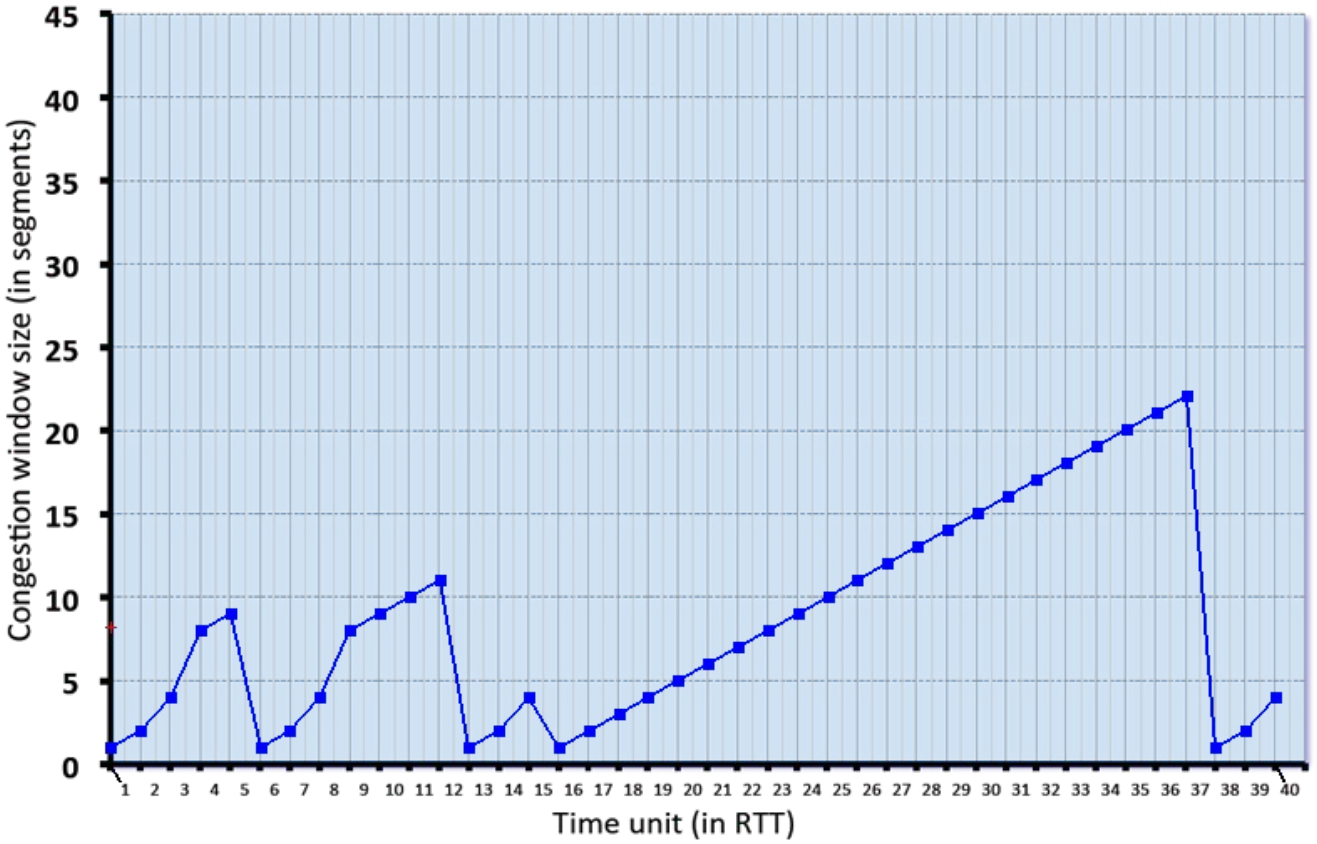
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

5. Give the times at which packets are lost via *triple ACK.*

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

***(iii)* there is a triple duplicate ACK for the first packet**.

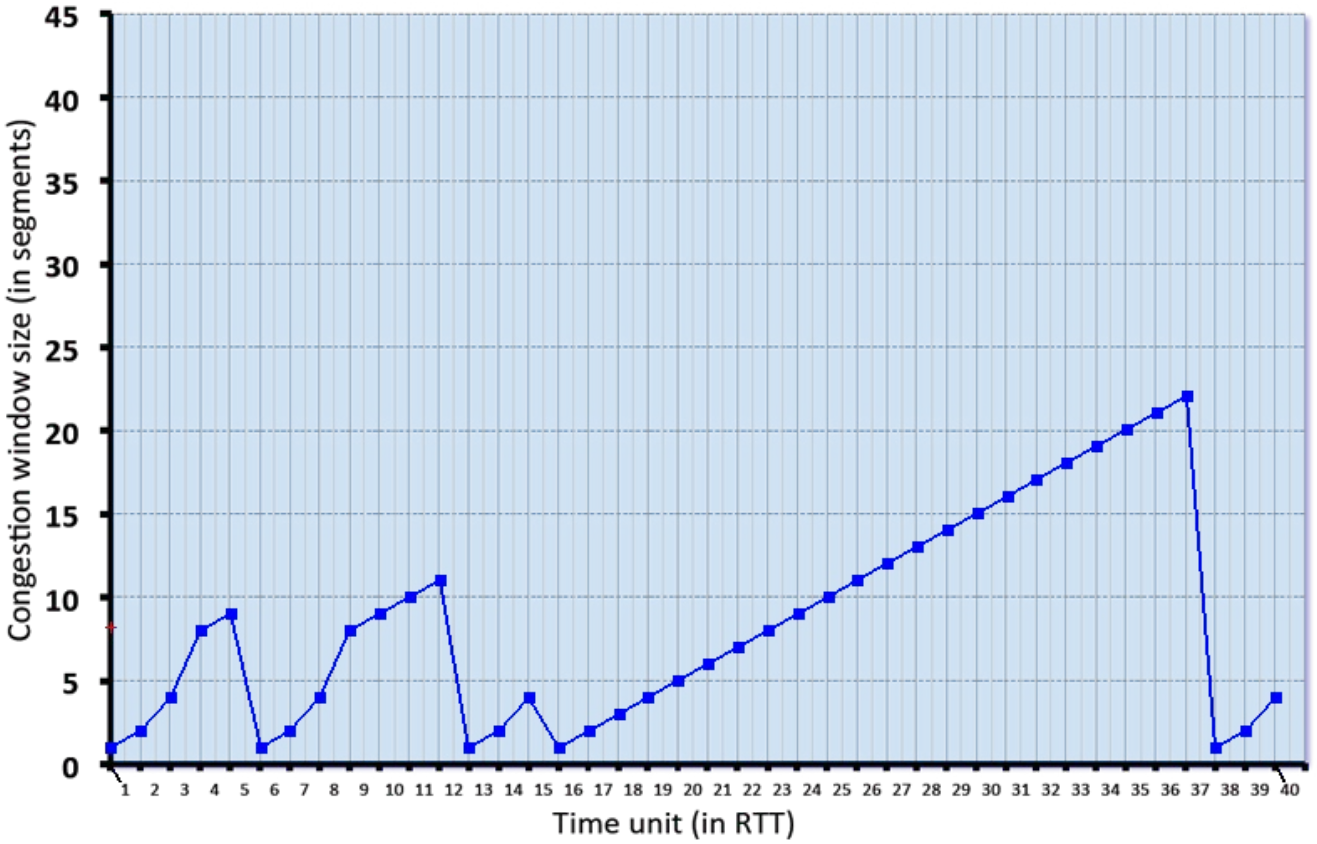
In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *sshthresh* is 8.

5. Give the times at which packets are lost via *triple ACK.*

None here

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

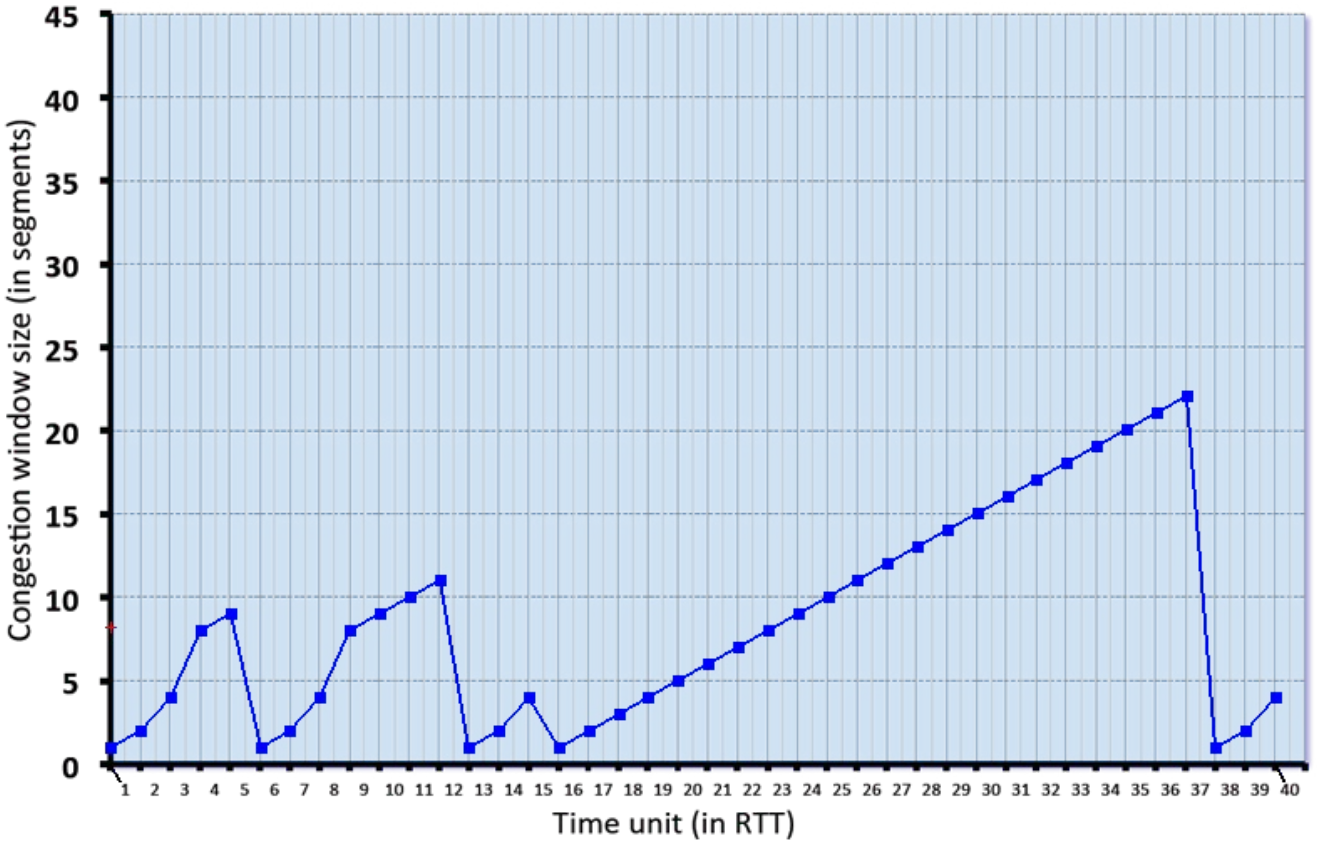
***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *ssthresh* is 8.

6. Give the times at which the value of *ssthresh* changes (if it changes between t=3 and t=4, use t=4 in your answer)

TCP in action



Consider the figure, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit.

The result of sending that flight of packets is that either

***(i)* all packets are ACKed at the end of the interval, or**

***(ii)* there is a timeout for the first packet, or**

***(iii)* there is a triple duplicate ACK for the first packet**.

In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that resulted in the given evolution of TCP's *cwnd*.

The initial value of *cwnd* is 1, and the initial value of *ssthresh* is 8.

6. Give the times at which the value of *ssthresh* changes (if it changes between t=3 and t=4, use t=4 in your answer)

The ssthresh can change when there is a loss, hence the answer is 6,13,16,38