Chapter 3
Transport Layer

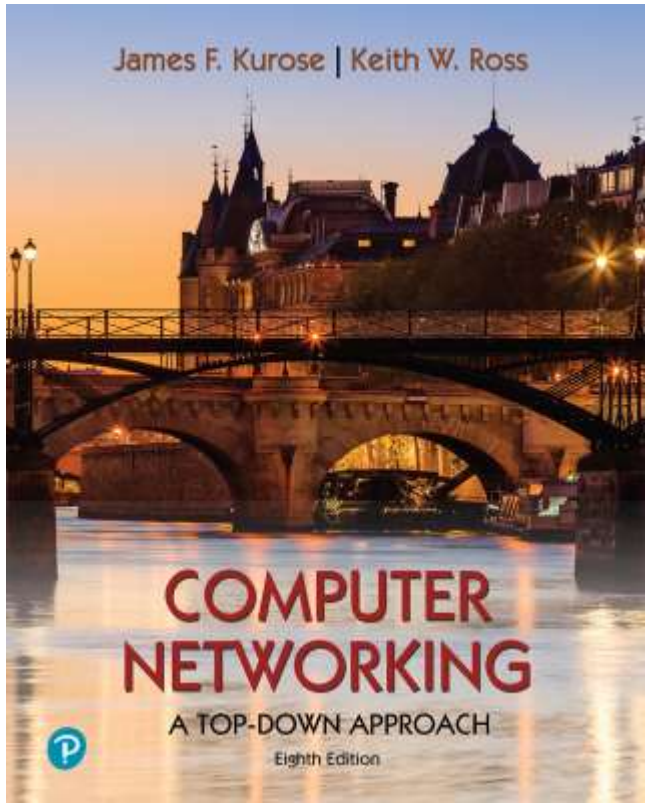
A note on the use of these PowerPoint slides:

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

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

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top-Down Approach

8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport layer: overview

Our goal:

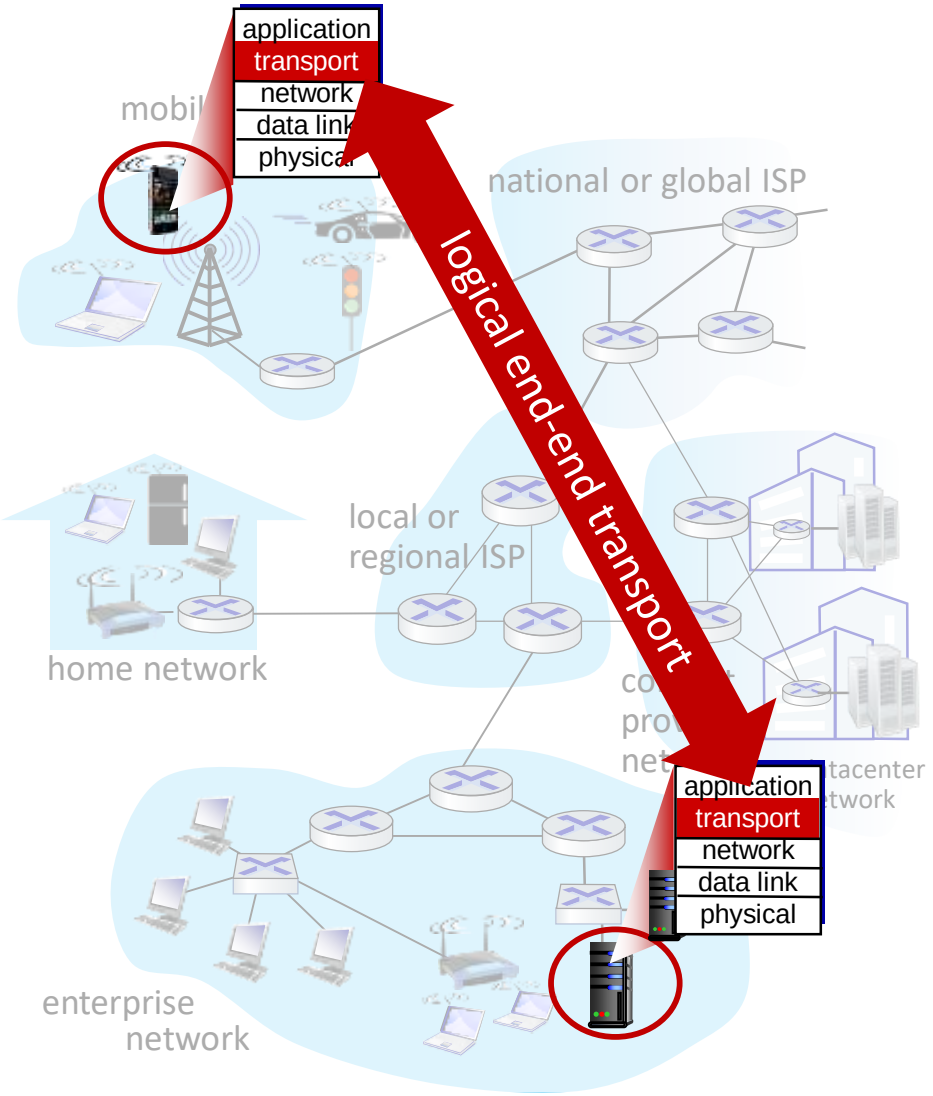
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion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

household analogy:

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

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

Transport vs. network layer services and protocols

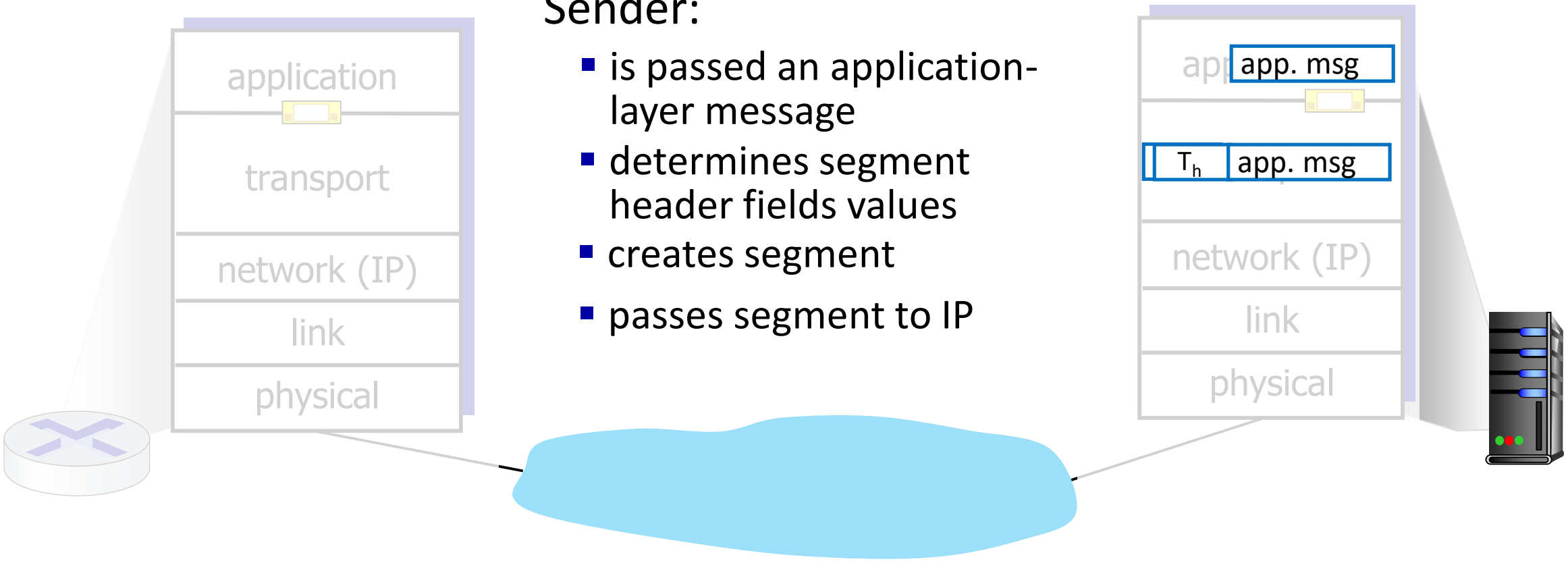
- network layer: logical communication between *hosts*
- transport layer: logical communication between *processes*
  - relies on, enhances, network layer services

household analogy:

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

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

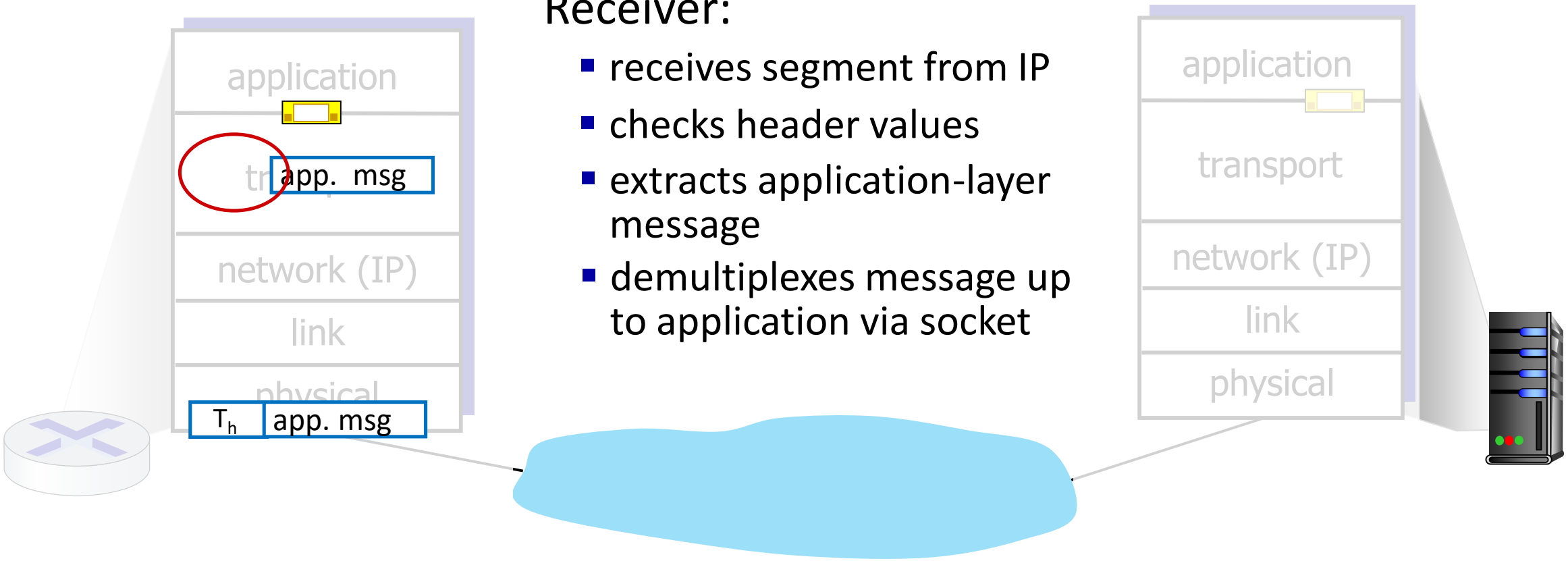
Transport Layer Actions



Sender:

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

Transport Layer Actions

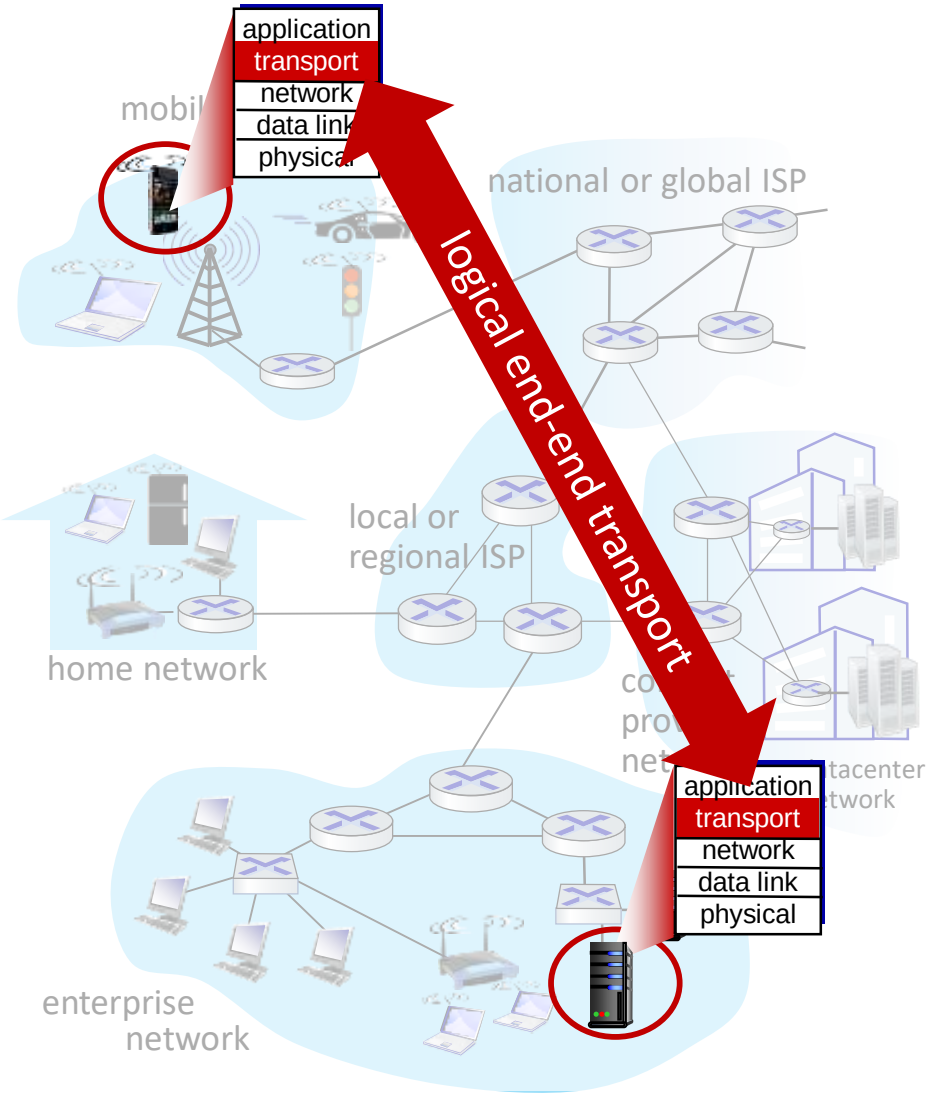


Receiver:

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

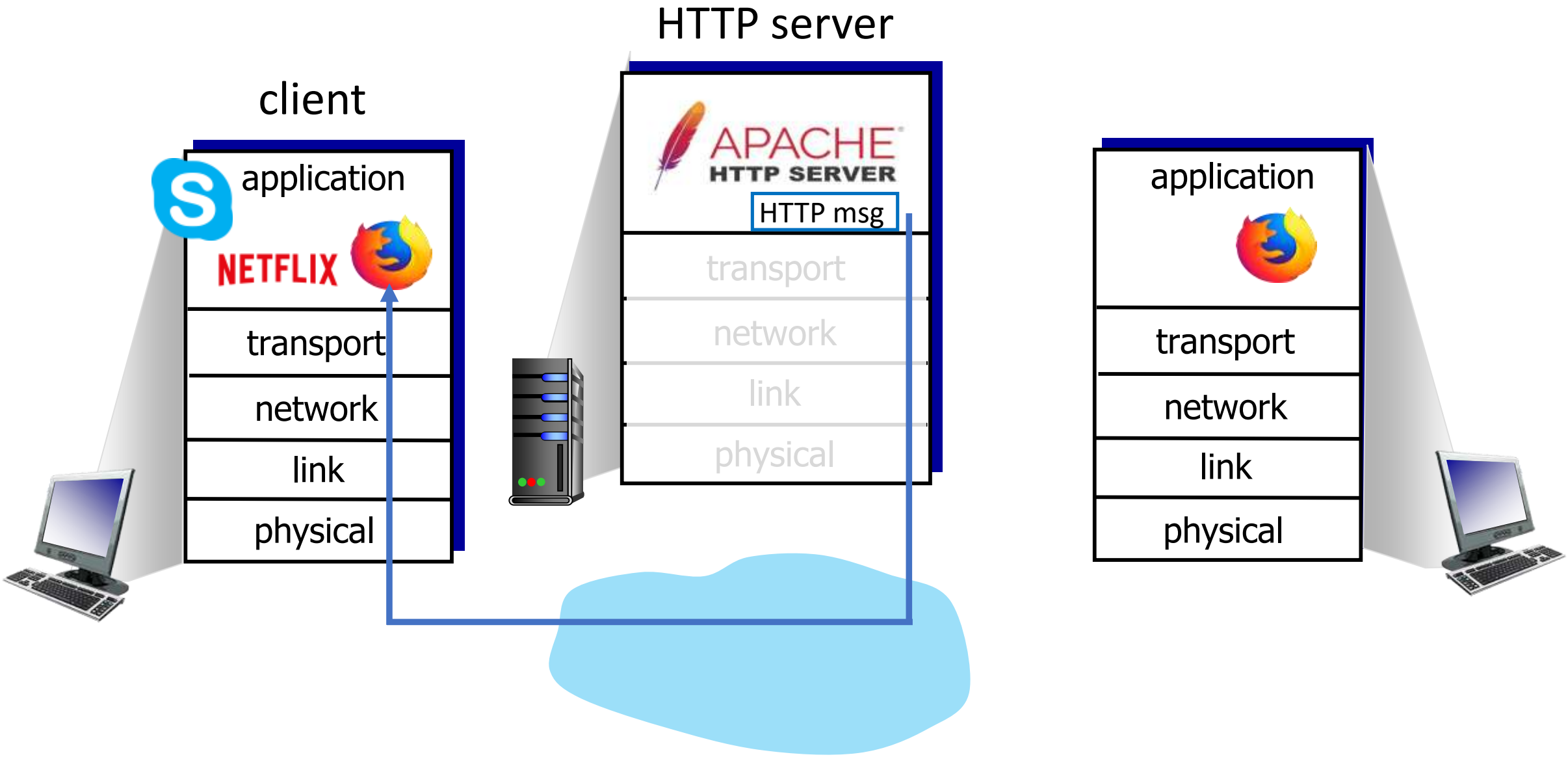
Two principal Internet transport protocols

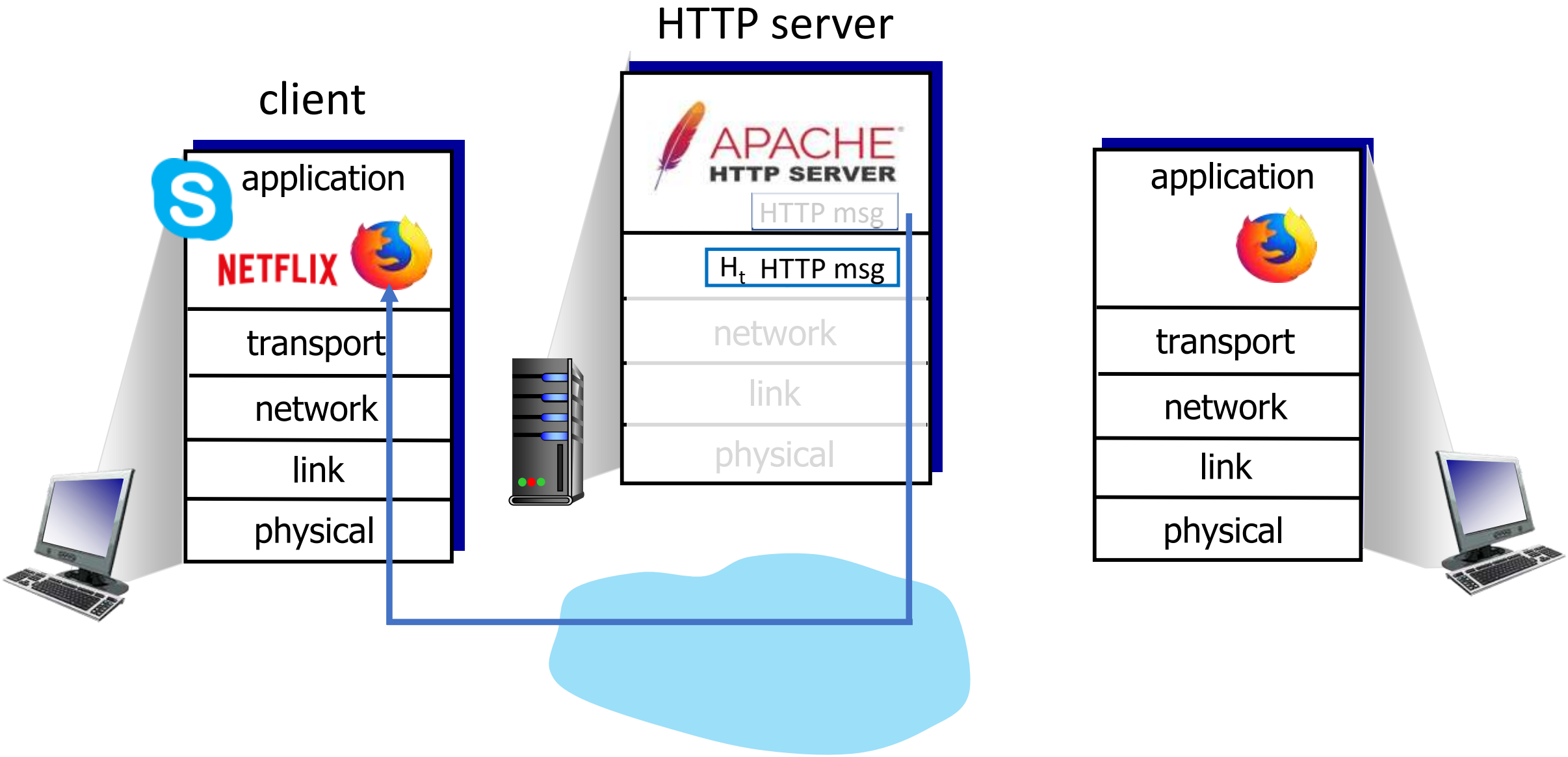
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth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

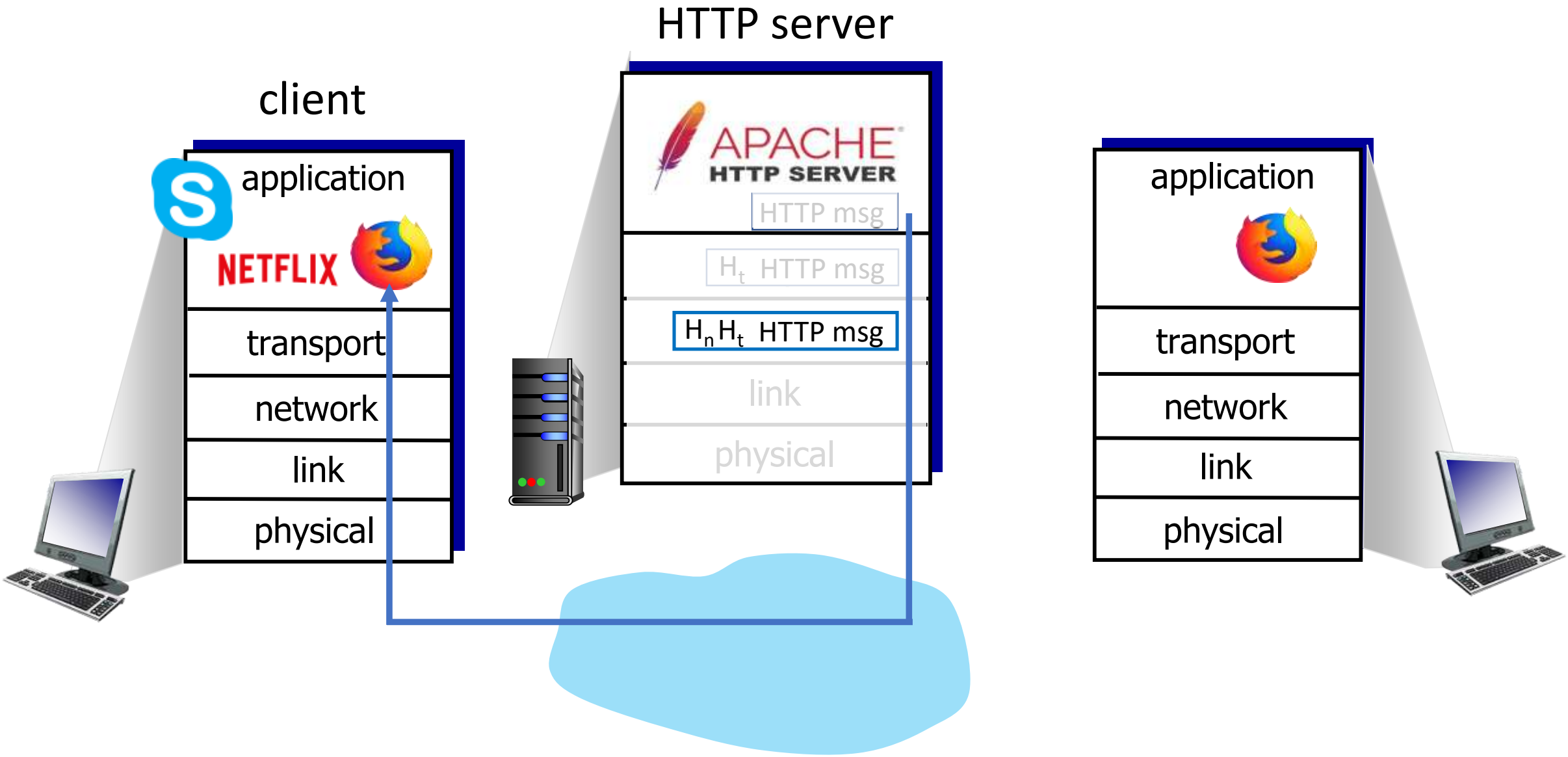


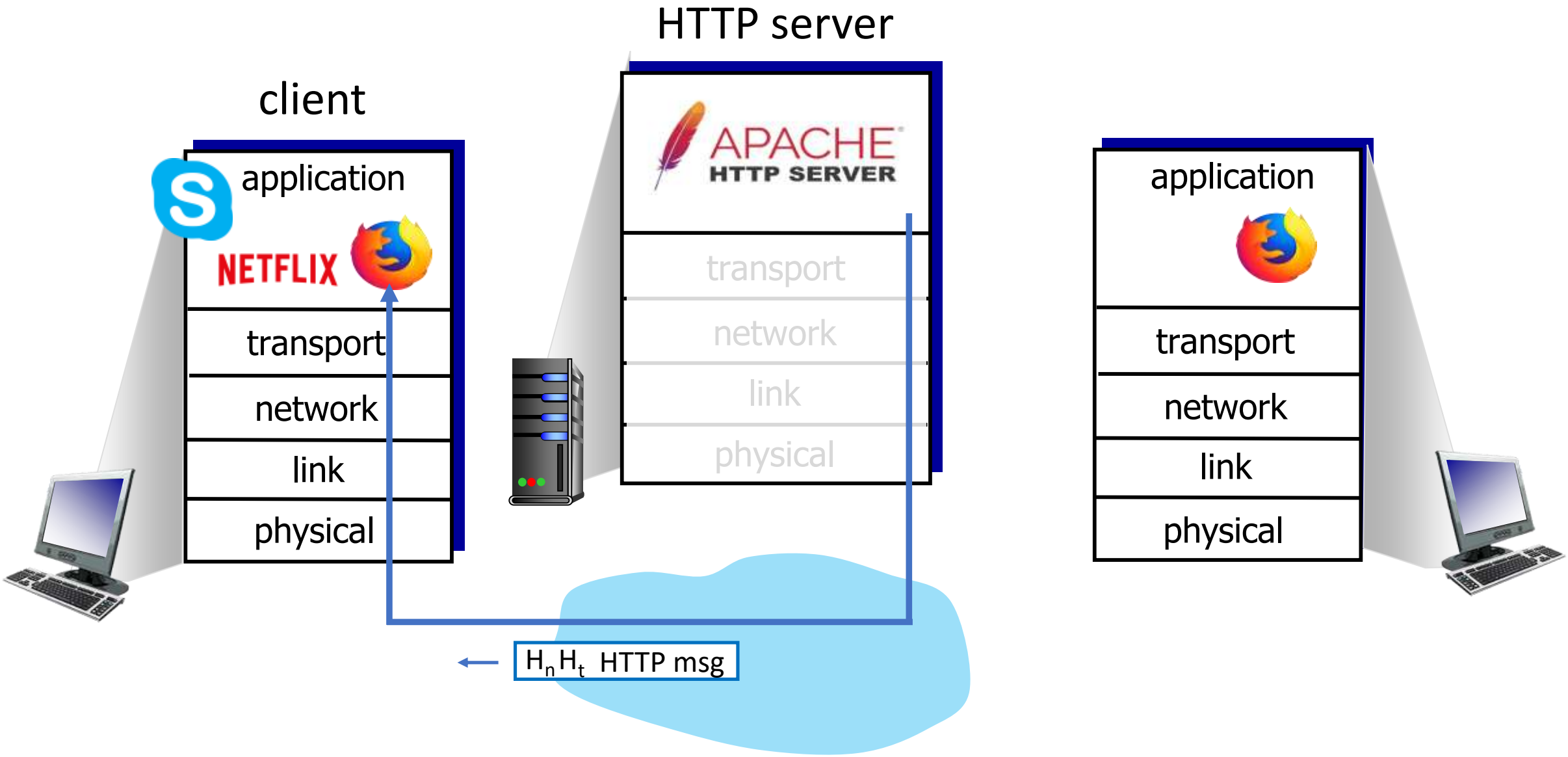
Chapter 3: roadmap

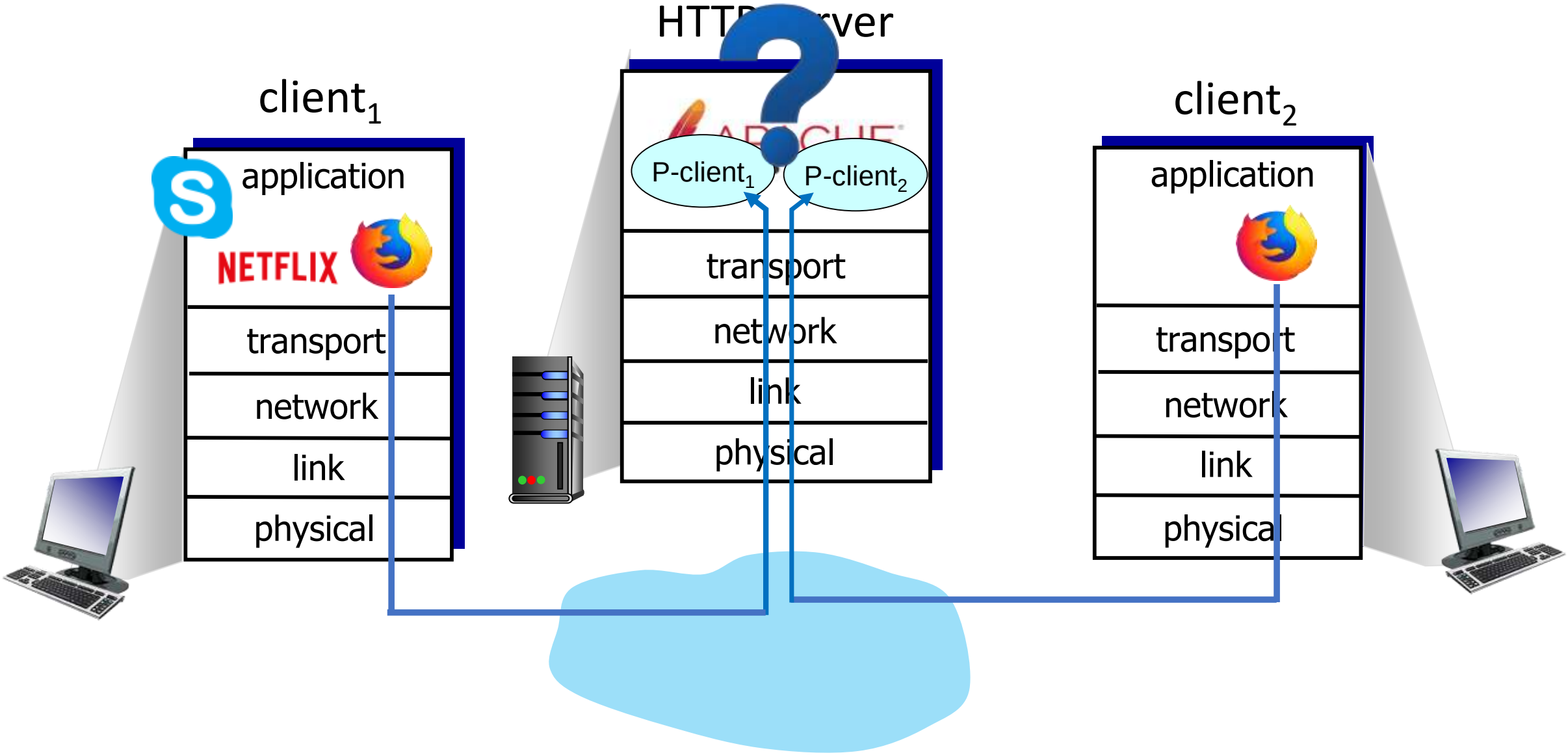
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality


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


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


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


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


client1
application
NETFLIX
transport
network
link
physical
HTTP server
P-client1
P-client2
transport
network
link
physical
client2
application
transport
network
link
physical

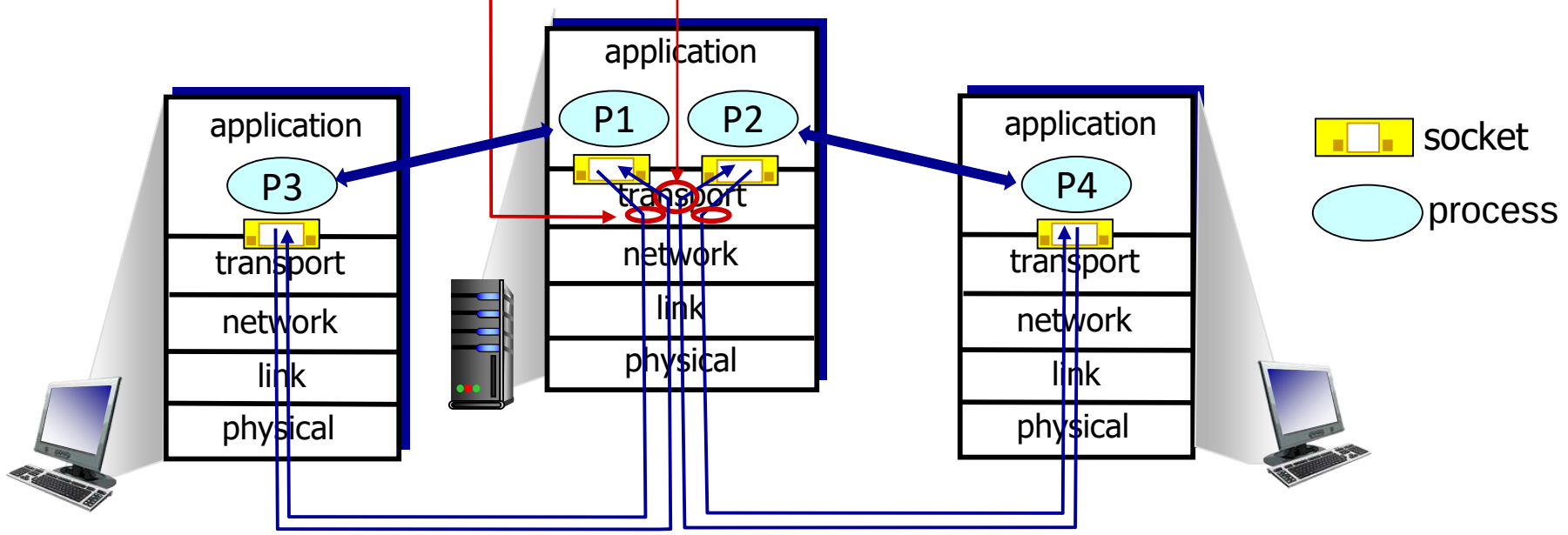
Multiplexing/demultiplexing

multiplexing at sender:

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

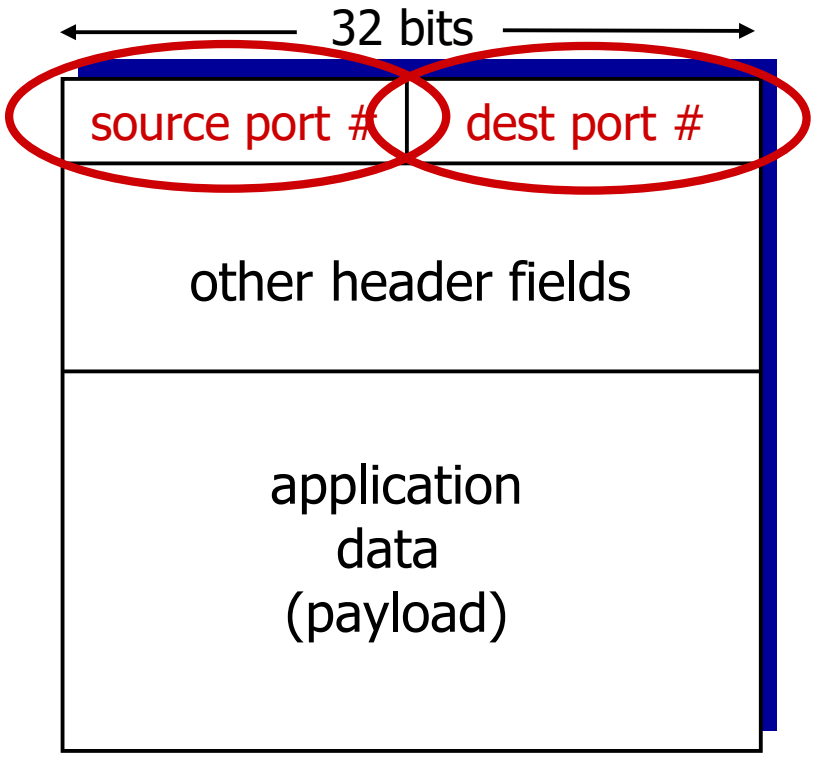
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

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



TCP/UDP segment format

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 2-tuple
  - destination IP address
  - destination port #

when receiving host receives *UDP* segment:

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

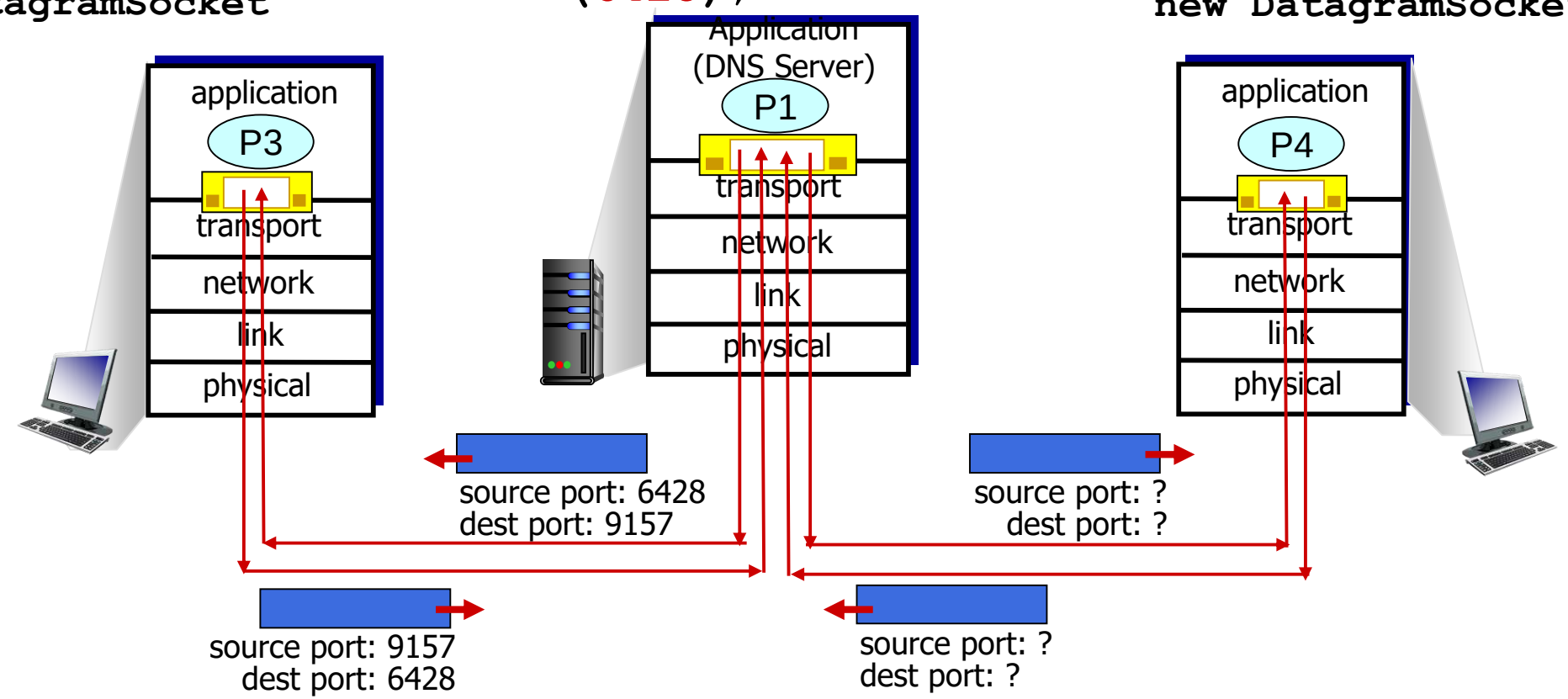
IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

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

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

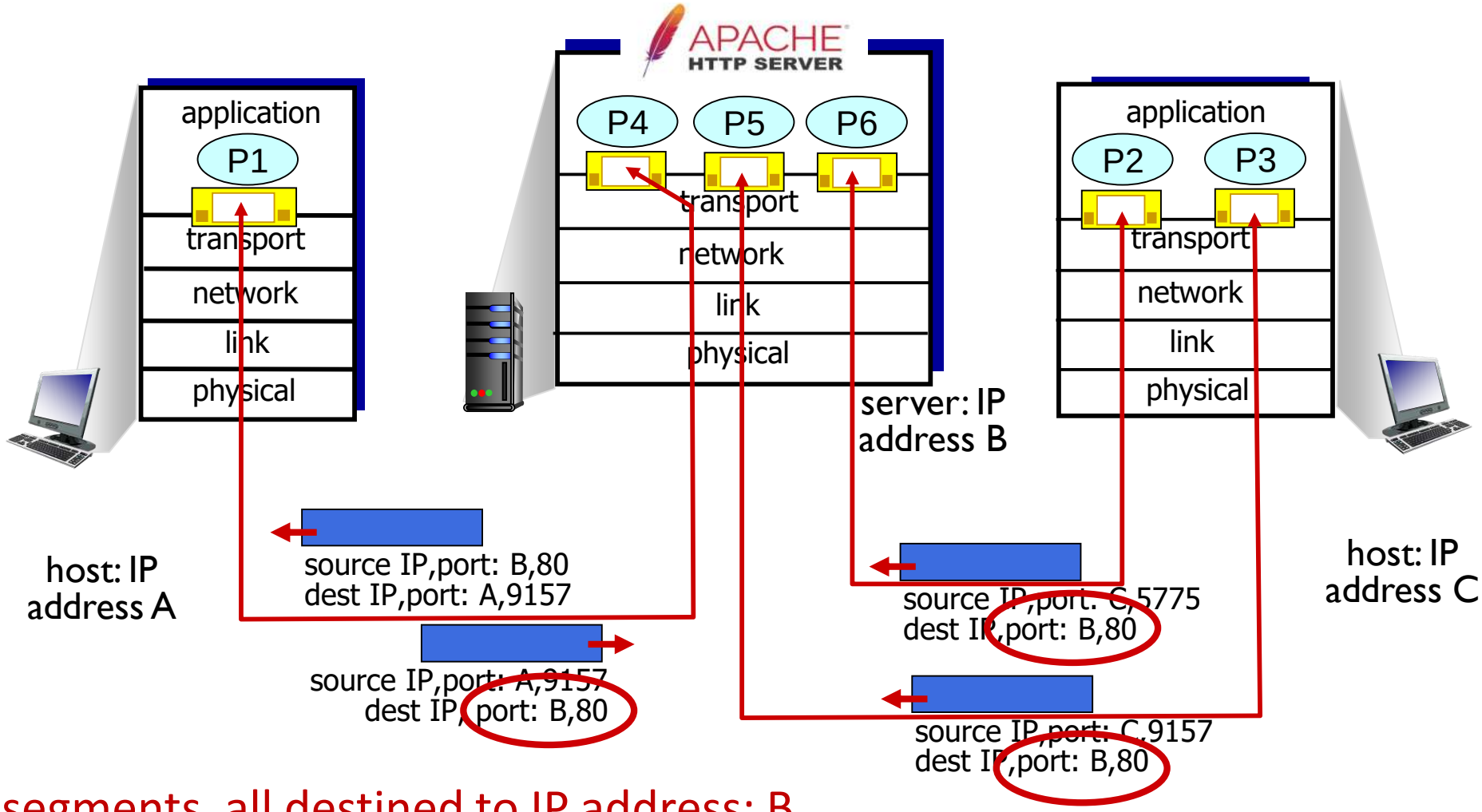
```
DatagramSocket mySocket1 =
 new DatagramSocket (5775);
```



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
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demultiplexing: example



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

Connection-oriented demultiplexing: example

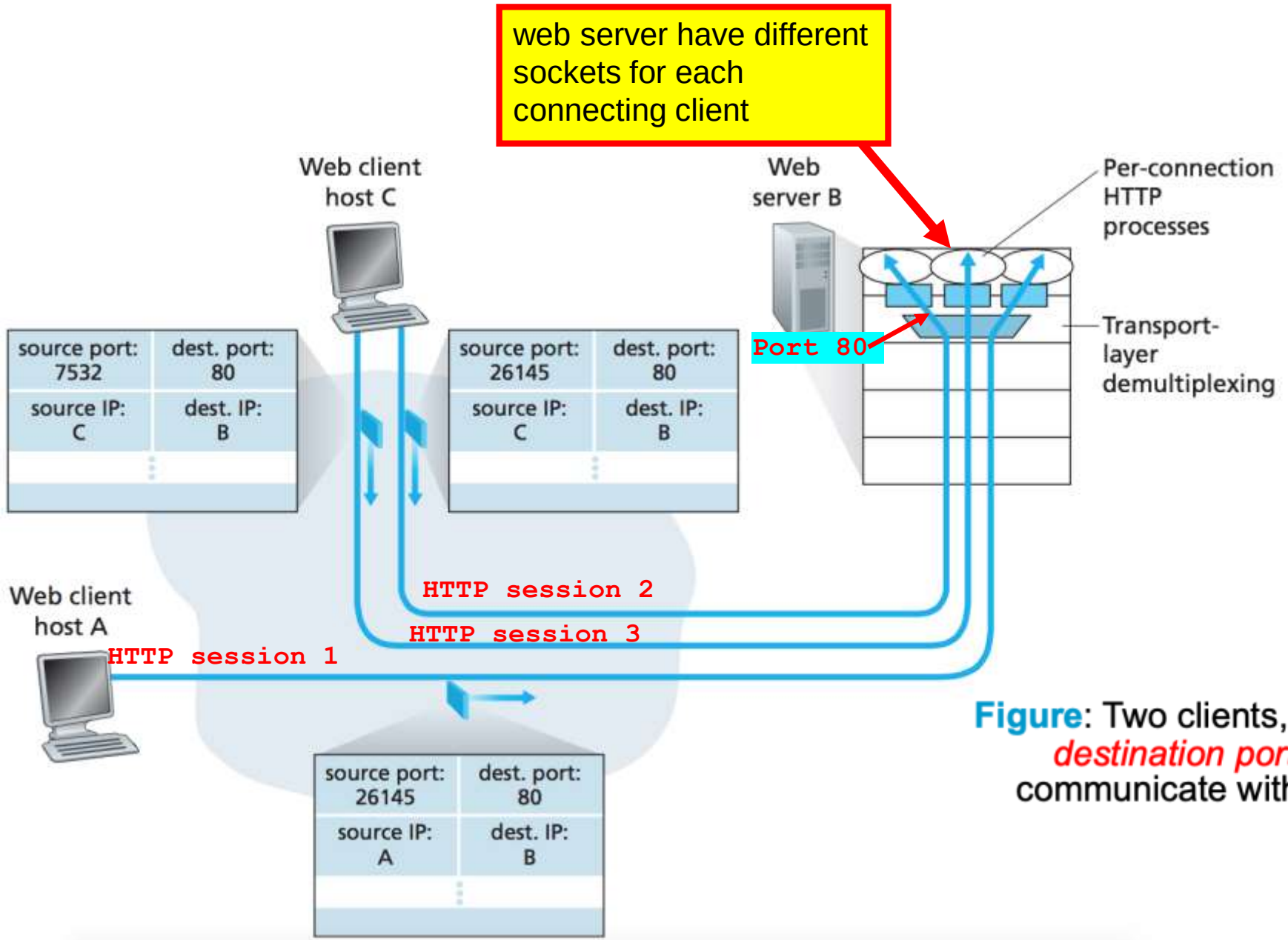


Figure: Two clients, using the *same destination port number* (**80**) to communicate with the same Web server B

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

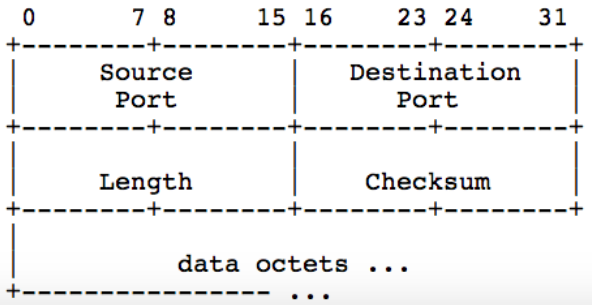
User Datagram Protocol

Introduction

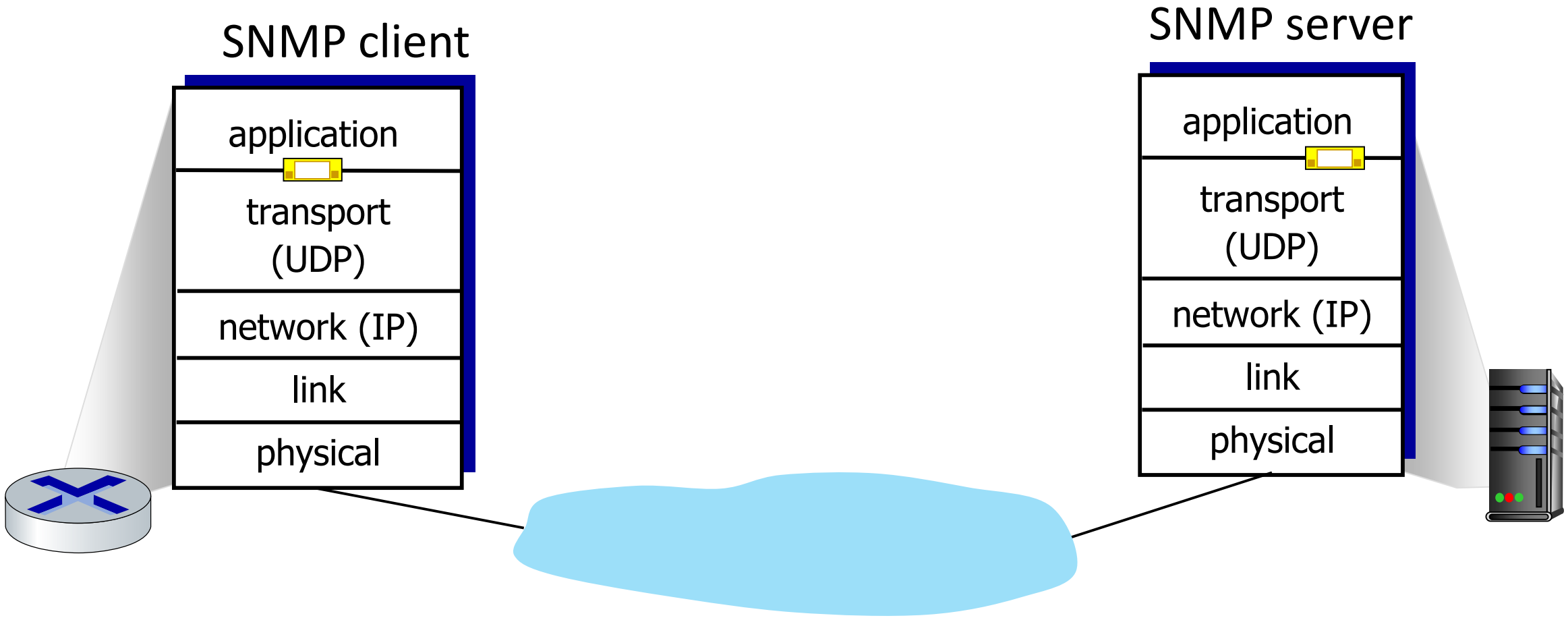
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

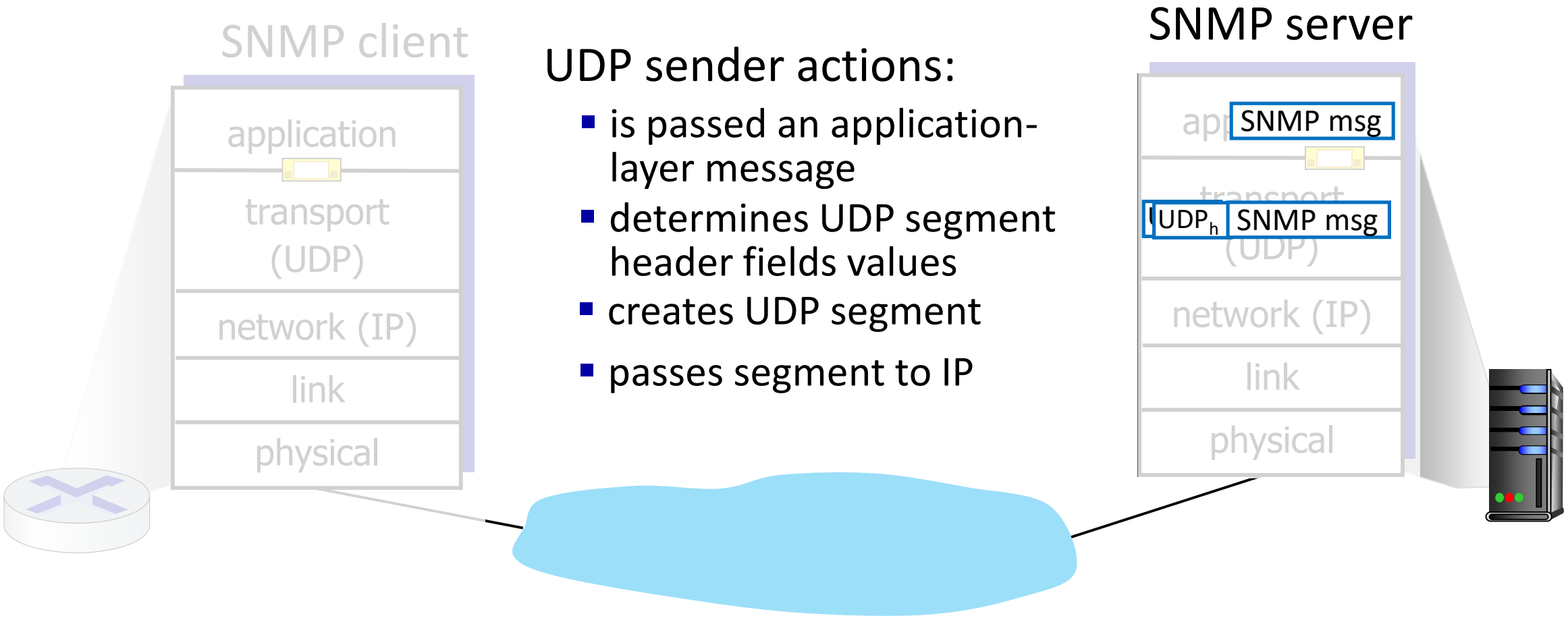
Format



UDP: Transport Layer Actions



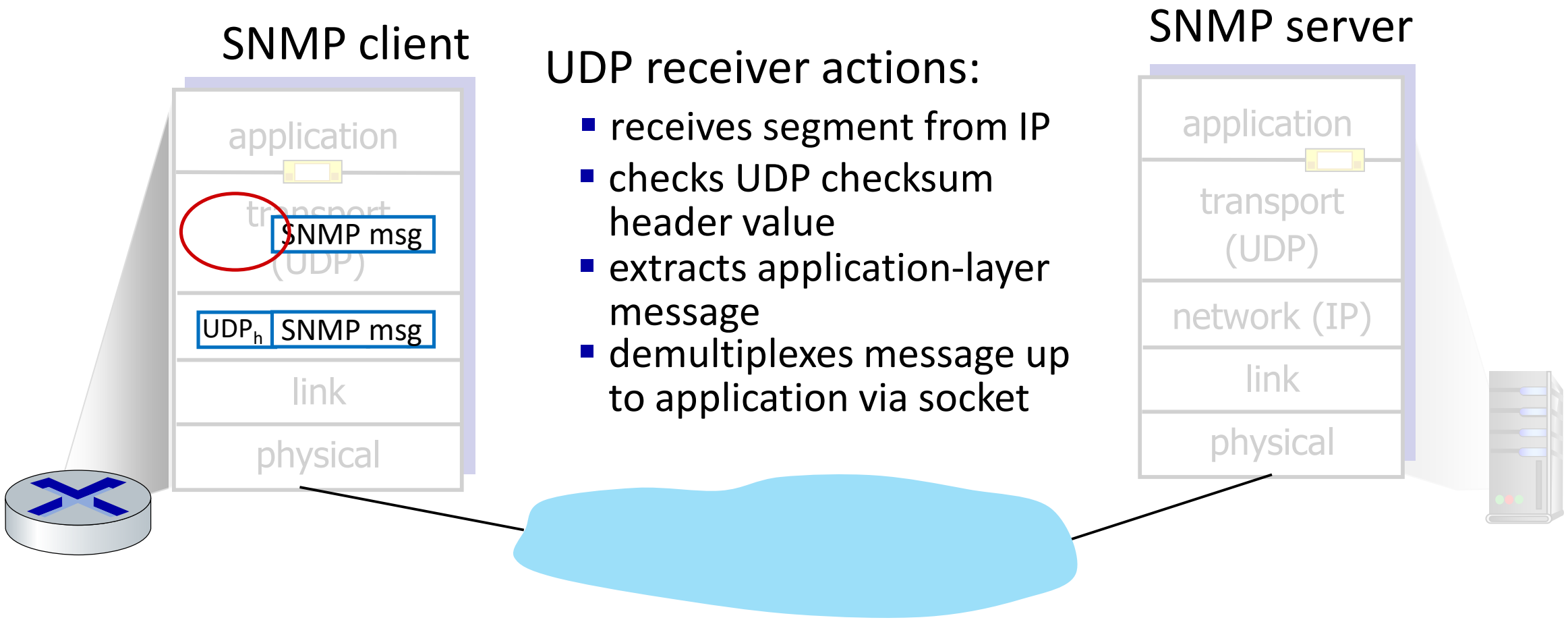
UDP: Transport Layer Actions



UDP sender actions:

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

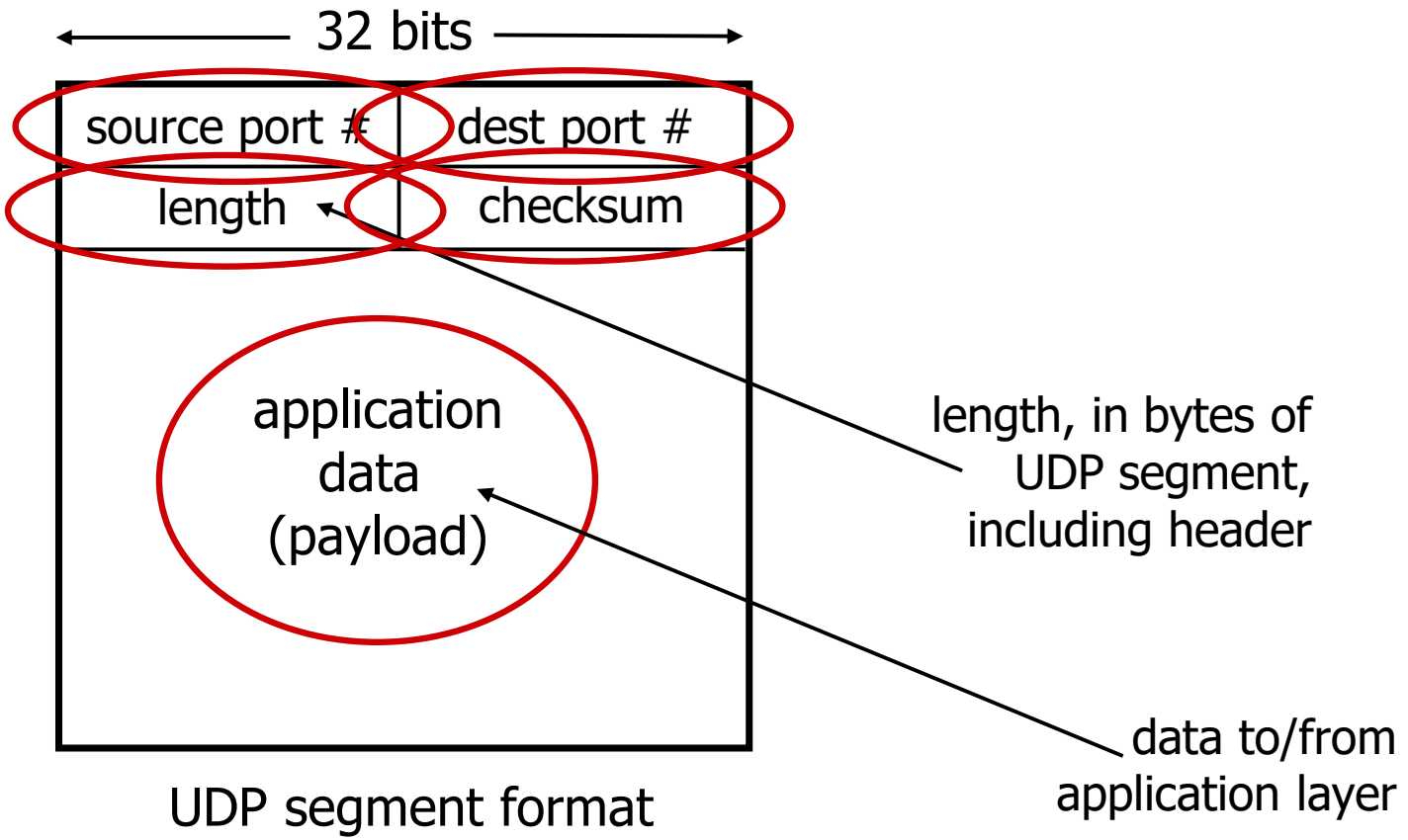
UDP: Transport Layer Actions



UDP receiver actions:

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

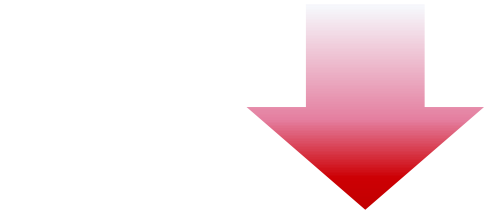
UDP segment header



UDP checksum

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

	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11



receiver-computed checksum ≠ sender-computed checksum (as received)

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
- sender puts 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 ….

Checksum Calculation

At the sender:

1087	13
15	FBA4H
Application Data (Payload)	

00000100 00111111 ⟶ 1087
00000000 00001101 ⟶ 13
00000000 00001111 ⟶ 15

0000 0100 0101 1011 → SUM

1st compliment:
1111 1011 1010 0100 → CHECKSUM
= FBA4H

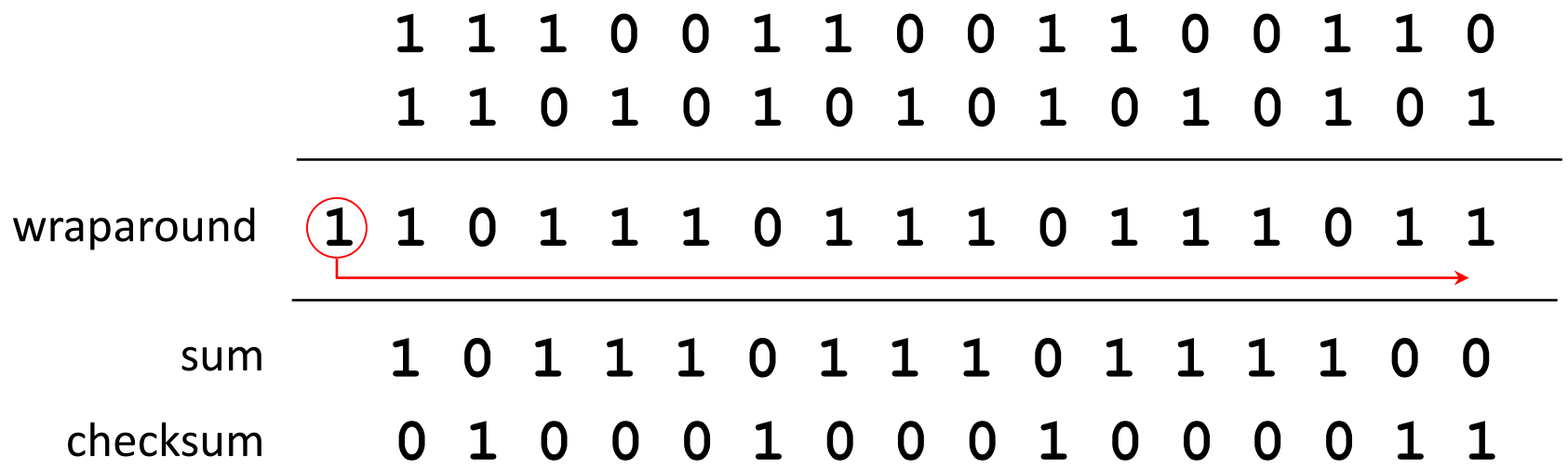
At the receiver:

1087	13
15	FBA4H
Application Data (Payload)	

0000 0100 0011 1111 → Source Port
+ 0000 0000 0000 1101 → Dest Port
+ 0000 0000 0000 1111 → Length
+ 1111 1011 1010 0100 → Checksum
1111 1111 1111 1111 **All 1's**
No Error

Internet checksum: an example

example: add two 16-bit integers

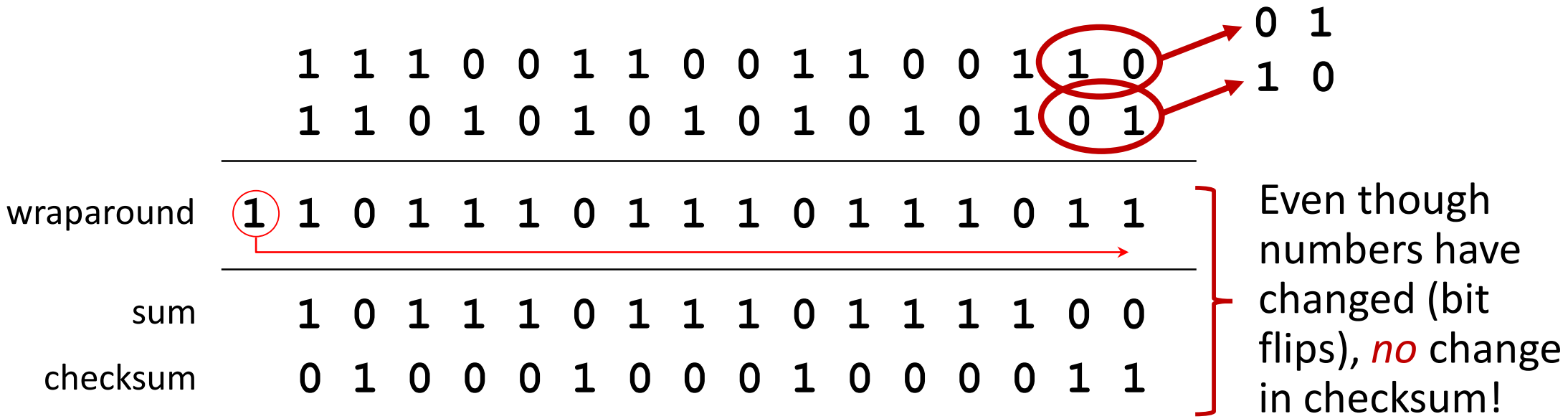


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

* 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



Example 1

Consider a receiver host received the segment with content given.
Generate the checksum value.

50439	16397
15	Checksum
Application Data (Payload)	

Solution:

```
50439 → 11000101 00000111
16397 → 01000000 00001101
   15 → 00000000 00001111
        ---------------------
        100000101 00100011   (Sum)
                         1
        ---------------------
        00000101 00100100    (Sum)
        11111010 11011011    (Checksum: 1's Complement)
```

$FADB_{16}$ (Checksum *in hexadecimal*)

Example 2

Consider a receiver host received the segment with content given, check if any error occurred.

1287	13
15	$\mathbf{7ADC_{16}}$
Application Data (Payload)	

Solution:

```
1287 → 00000101 00000111
  13 → 00000000 00001101
  15 → 00000000 00001111
7ADC → 01111010 11011100
       ---------------------
       01111111 11111111   (Sum)
```

Why error ???

summation result not equal to all 1's

Error !

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

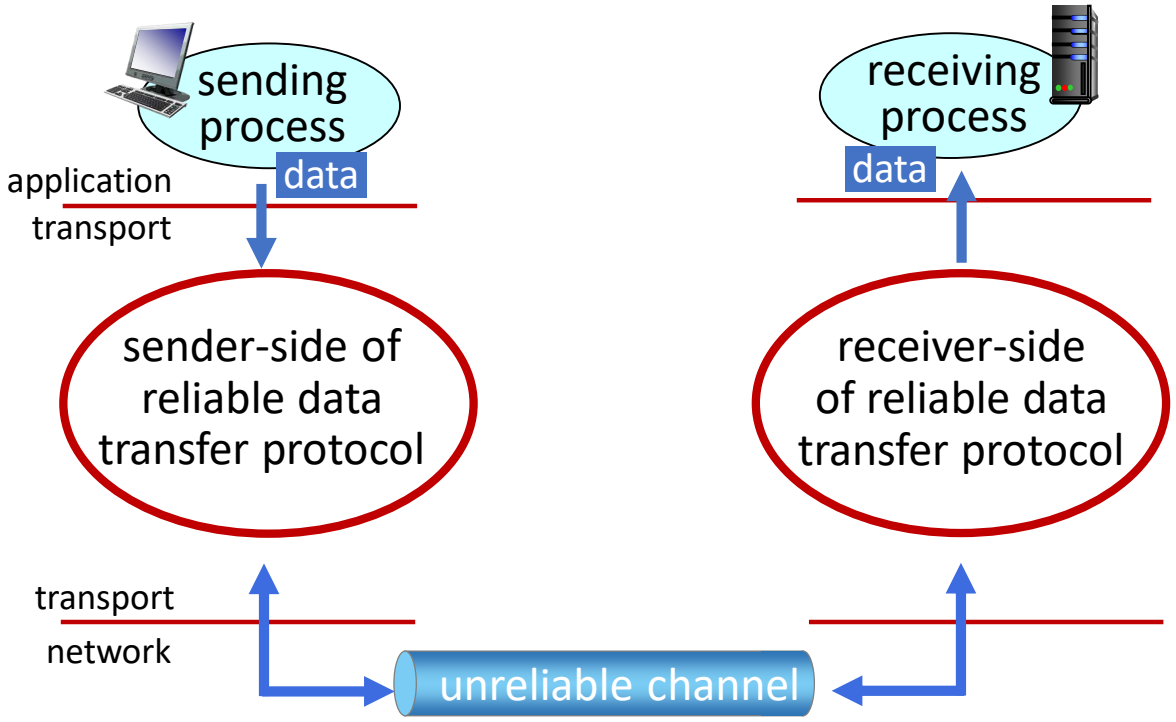
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Principles of reliable data transfer (rdt)

reliable service *abstraction*

Principles of reliable data transfer (rdt)

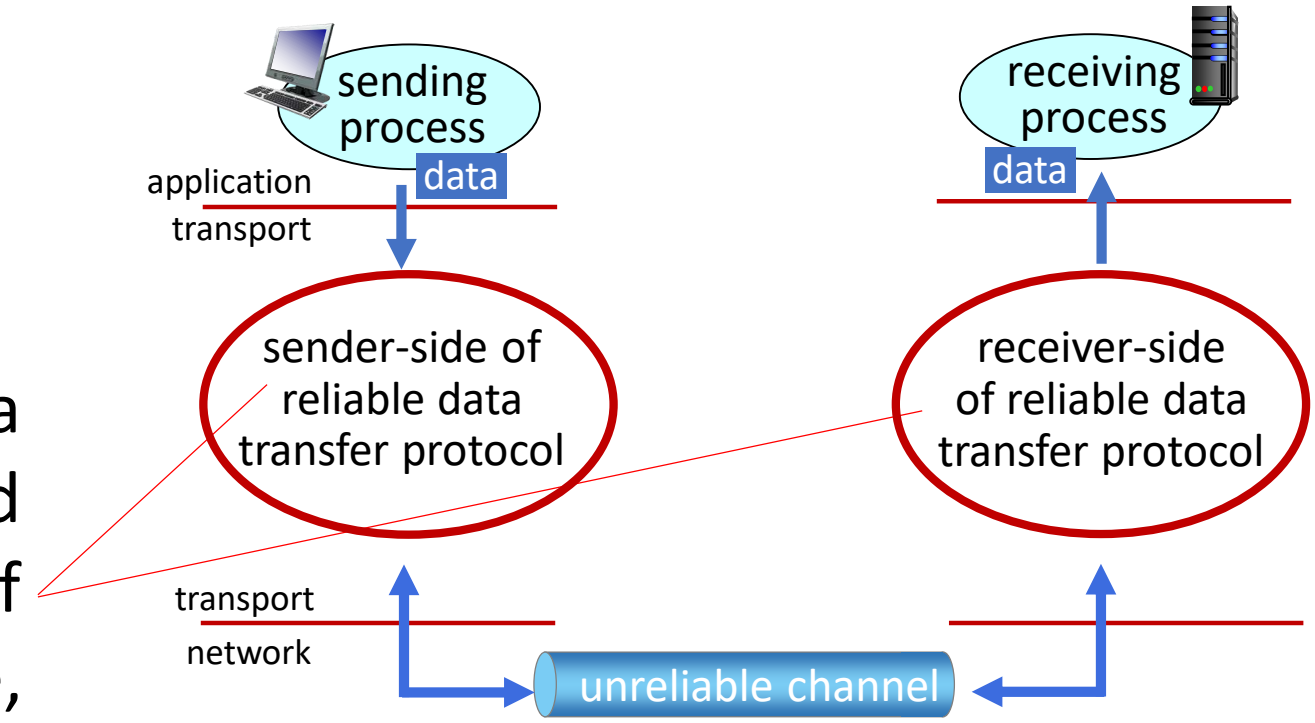
reliable service *abstraction*



reliable service *implementation*

Principles of reliable data transfer (rdt)

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

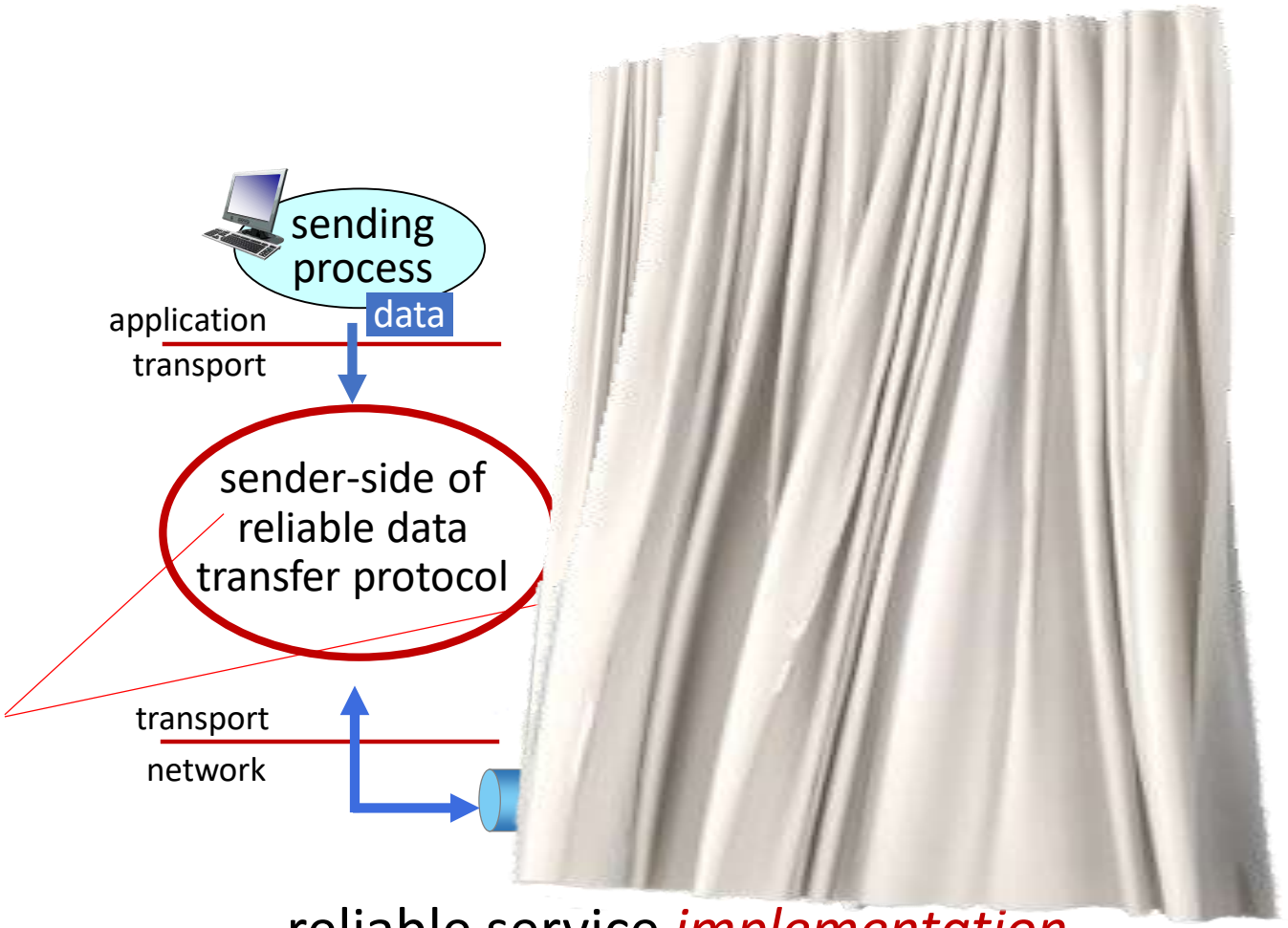


reliable service *implementation*

Principles of reliable data transfer (rdt)

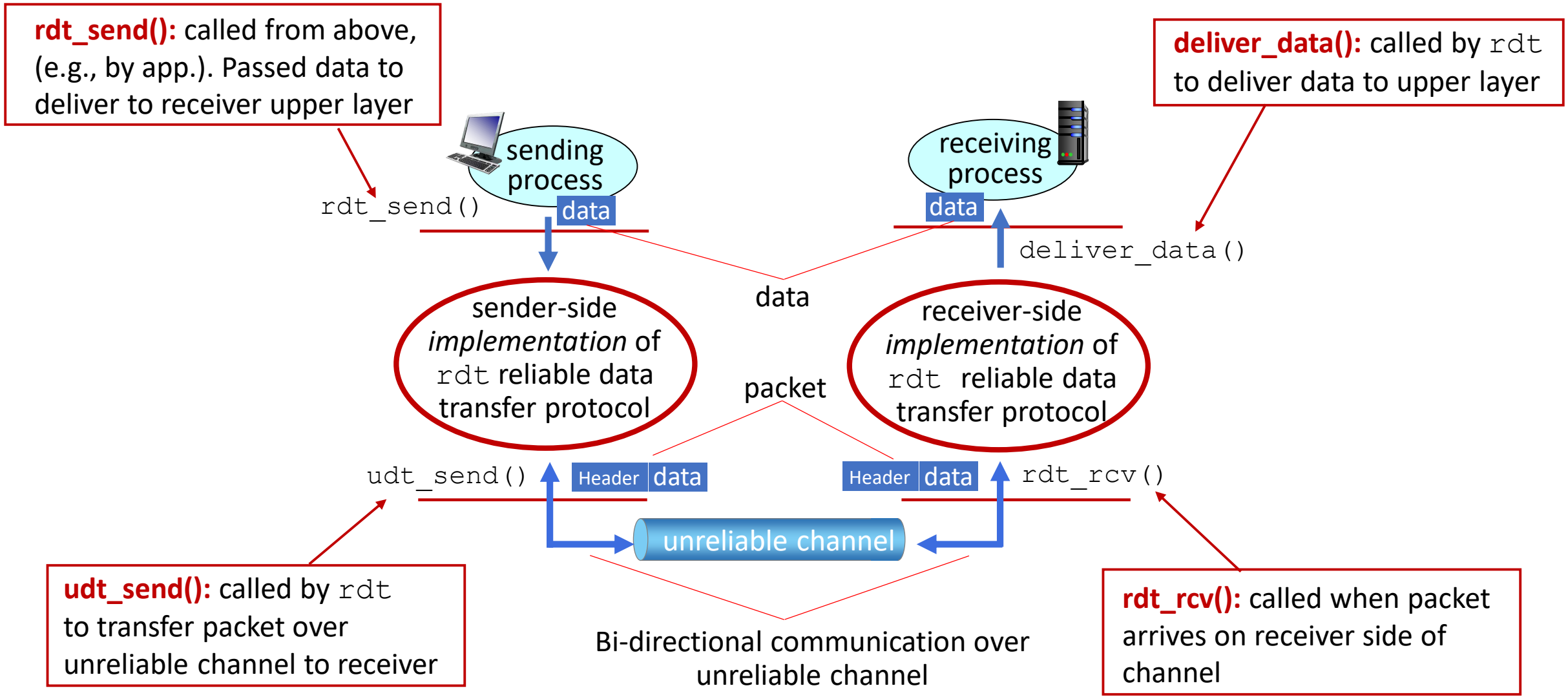
Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message



reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces



Reliable data transfer (rdt): 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!

rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- no need to provide feedback to sender
- no need for the rcv to ask sender to slow down sending rate

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?

rdt2.0: channel with bit errors

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

stop and wait

sender sends one packet, then waits for receiver response

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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

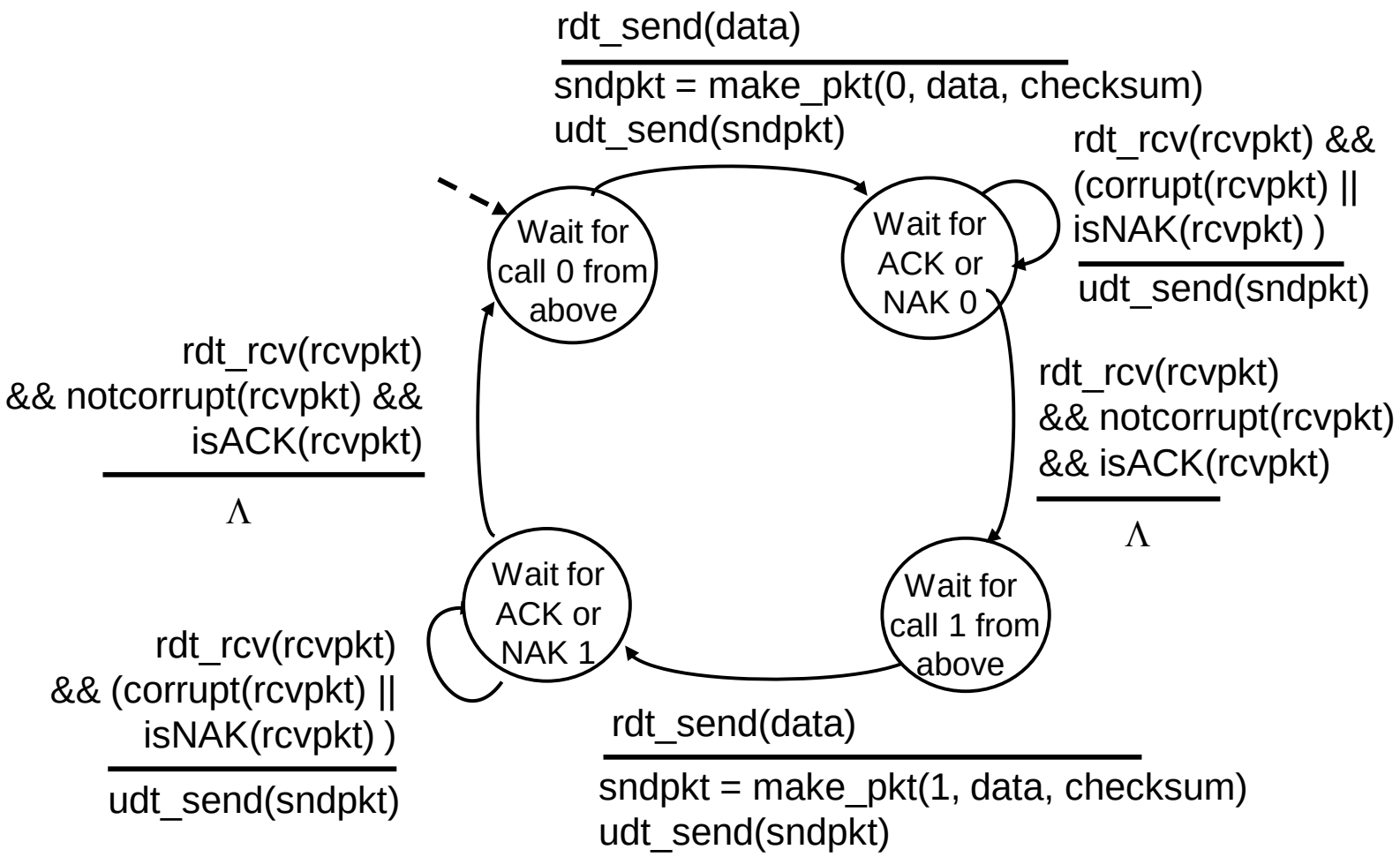
handling duplicates:

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

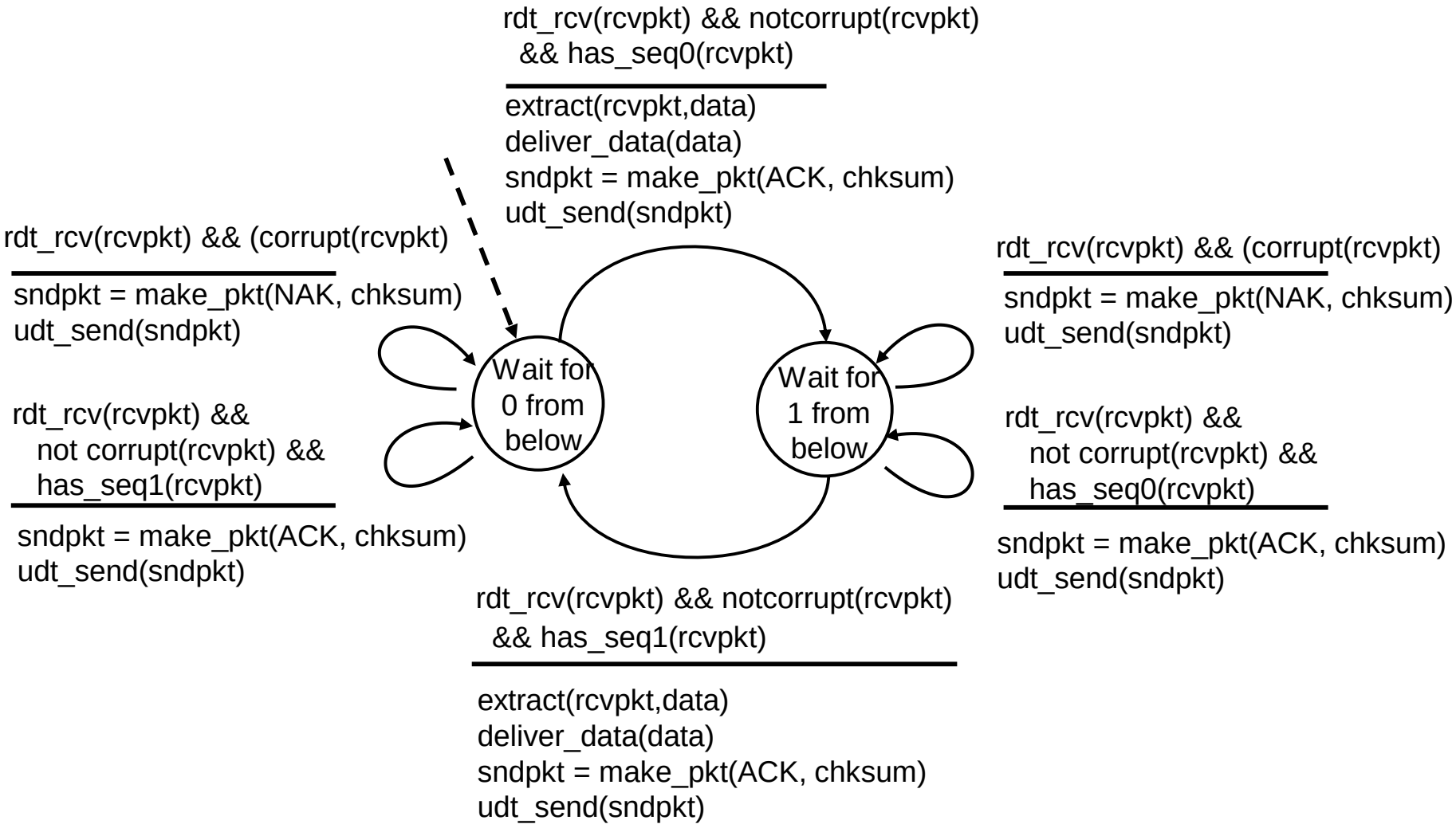
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, 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 received OK at sender

rdt2.2: a NAK-free protocol

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

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

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?

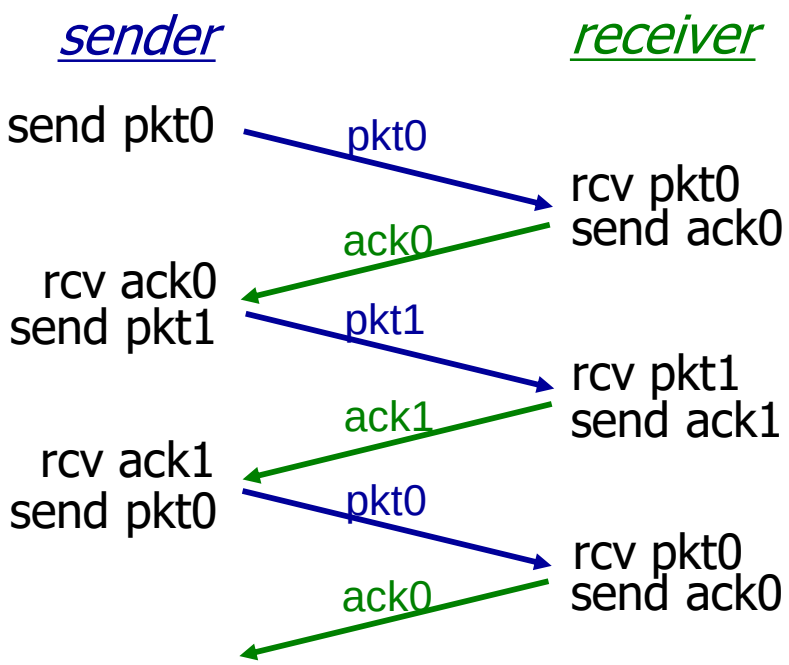
rdt3.0: channels with errors *and* loss

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

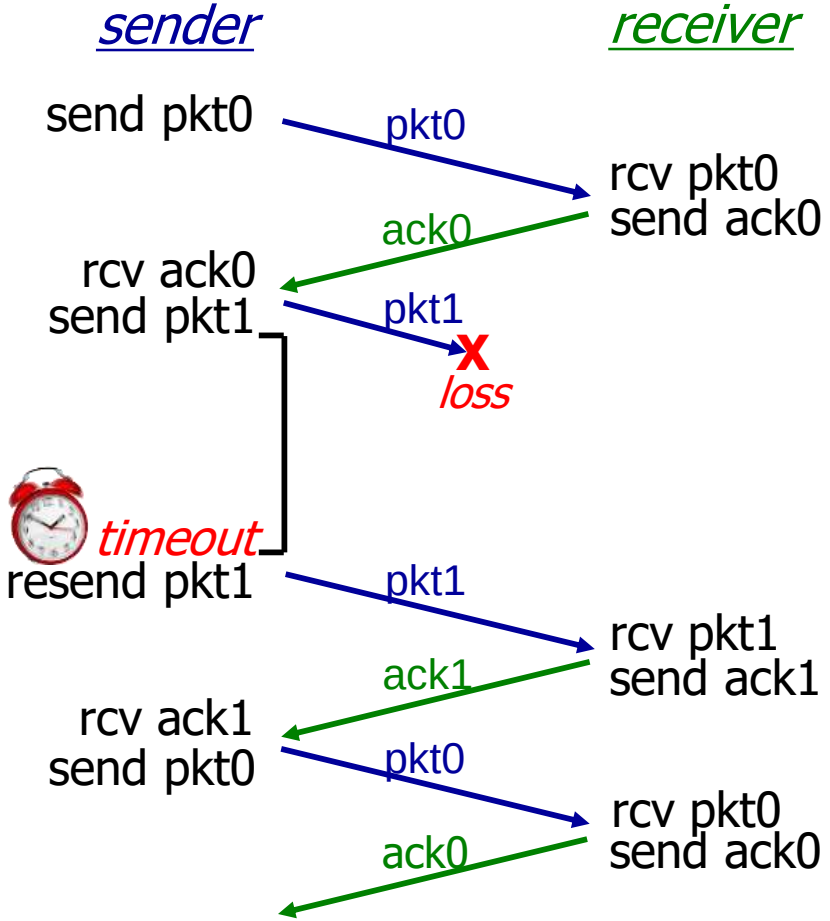
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

rdt3.0 in action (stop and wait)

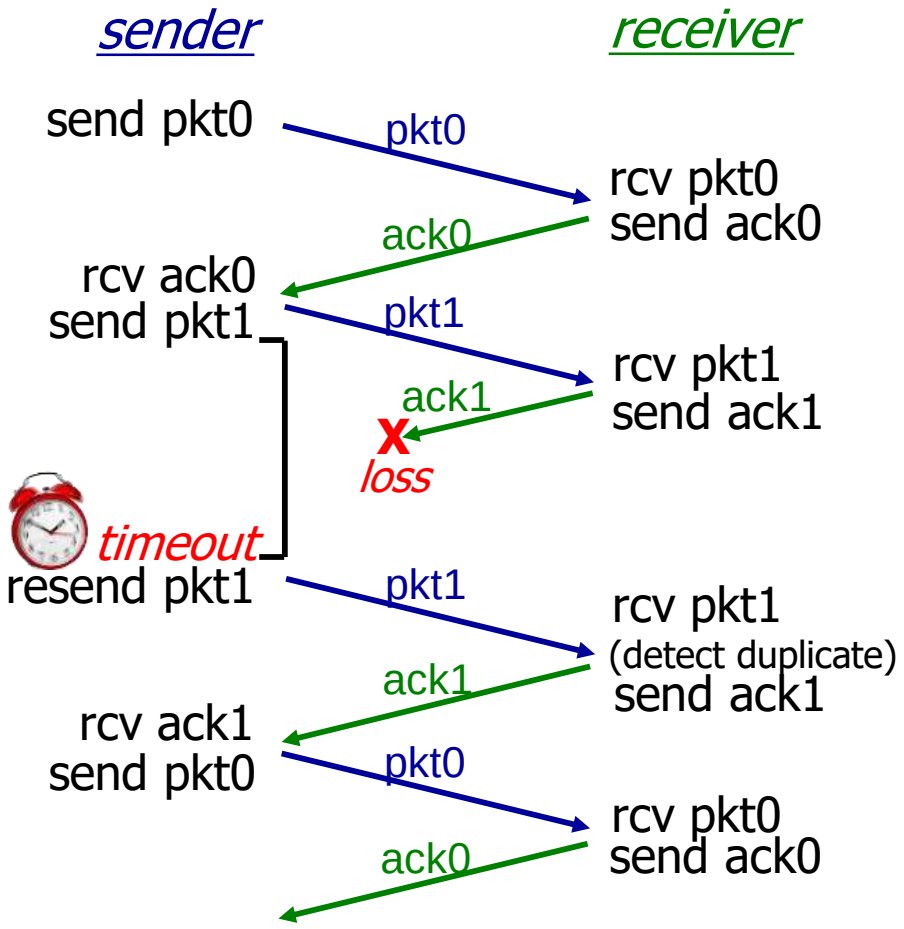


(a) no loss

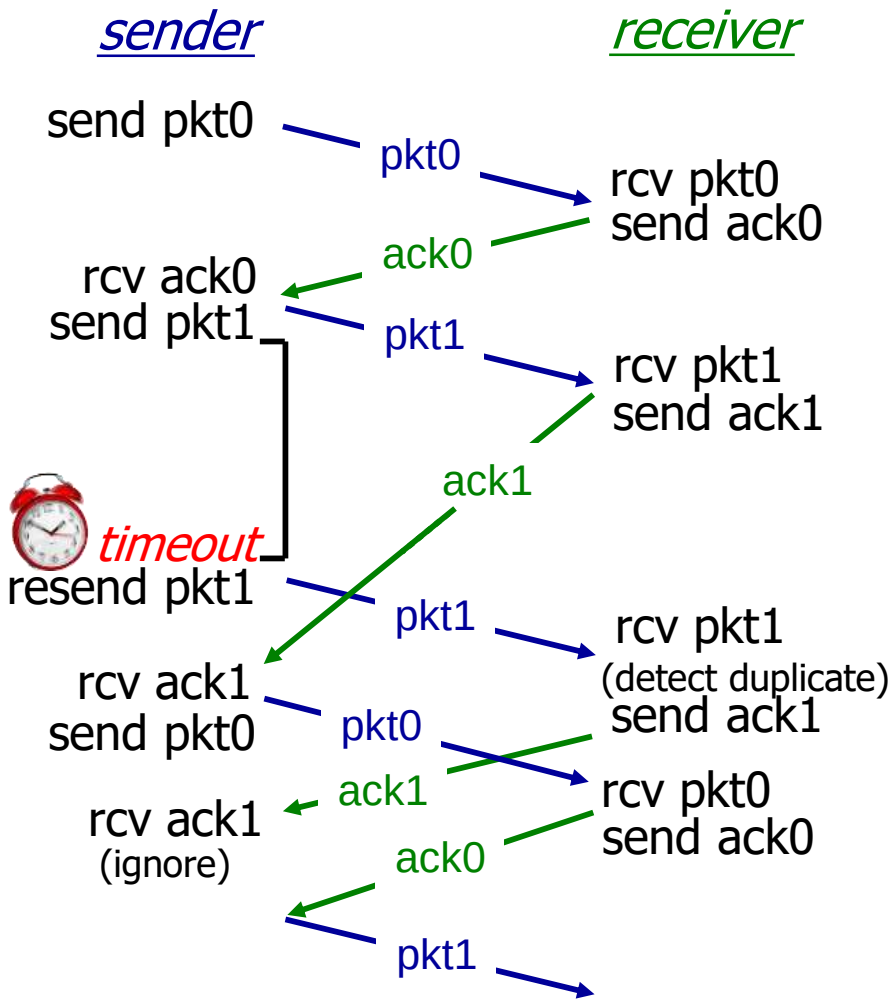


(b) packet loss

rdt3.0 in action (stop and wait)



(c) ACK loss



(d) premature timeout/ delayed ACK

Self-Study

Question:

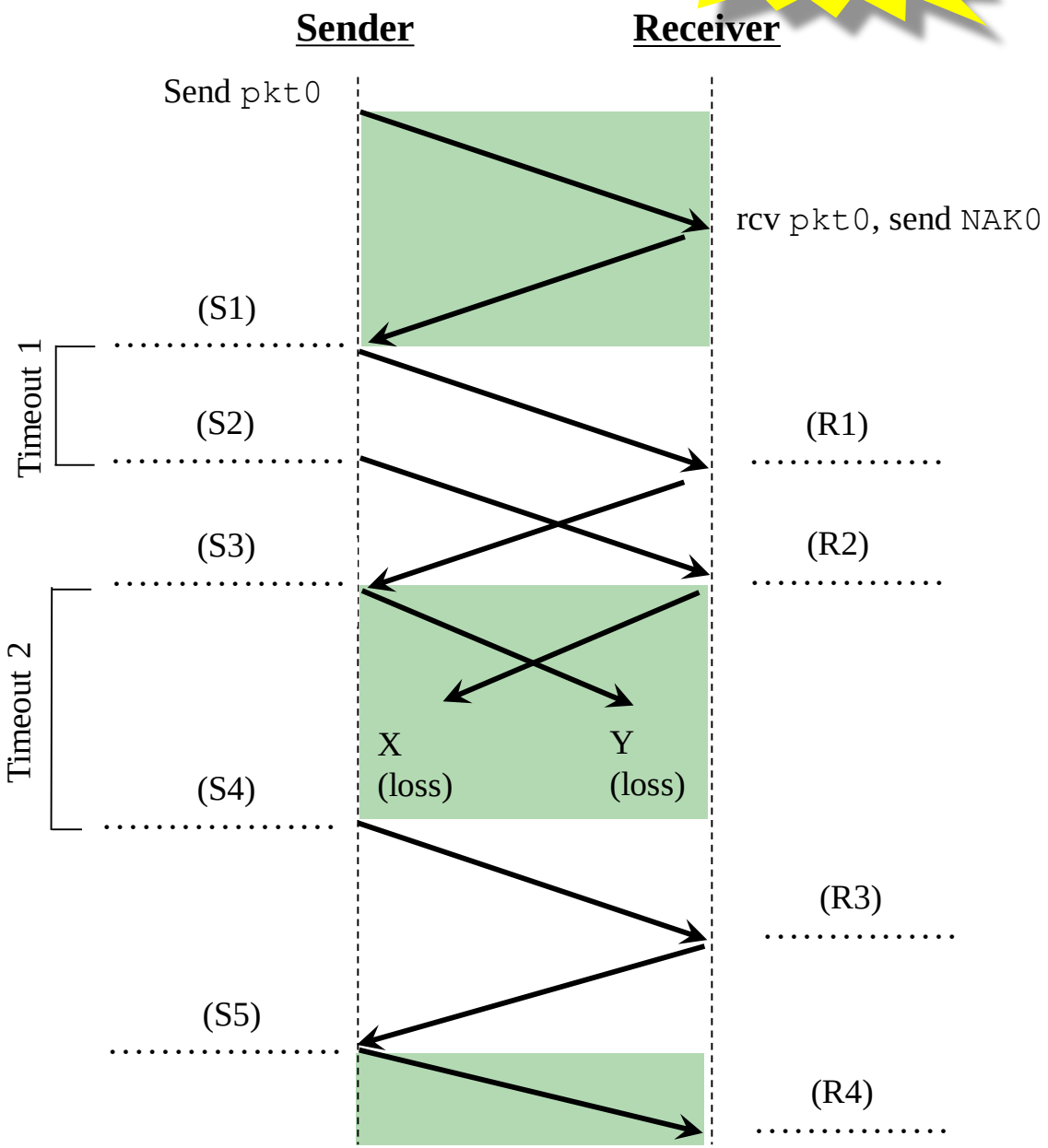
Supposed a sender has 3 packets to be sent to a receiver.

a) Complete the following figure by writing down the best answer for all S# and R#. Assume that no error after S1.

b) What are X and Y?

c) What is the problem between S1 and S2?

d) Is the any discard packet at receiver? Why?



Self-Study

Question:

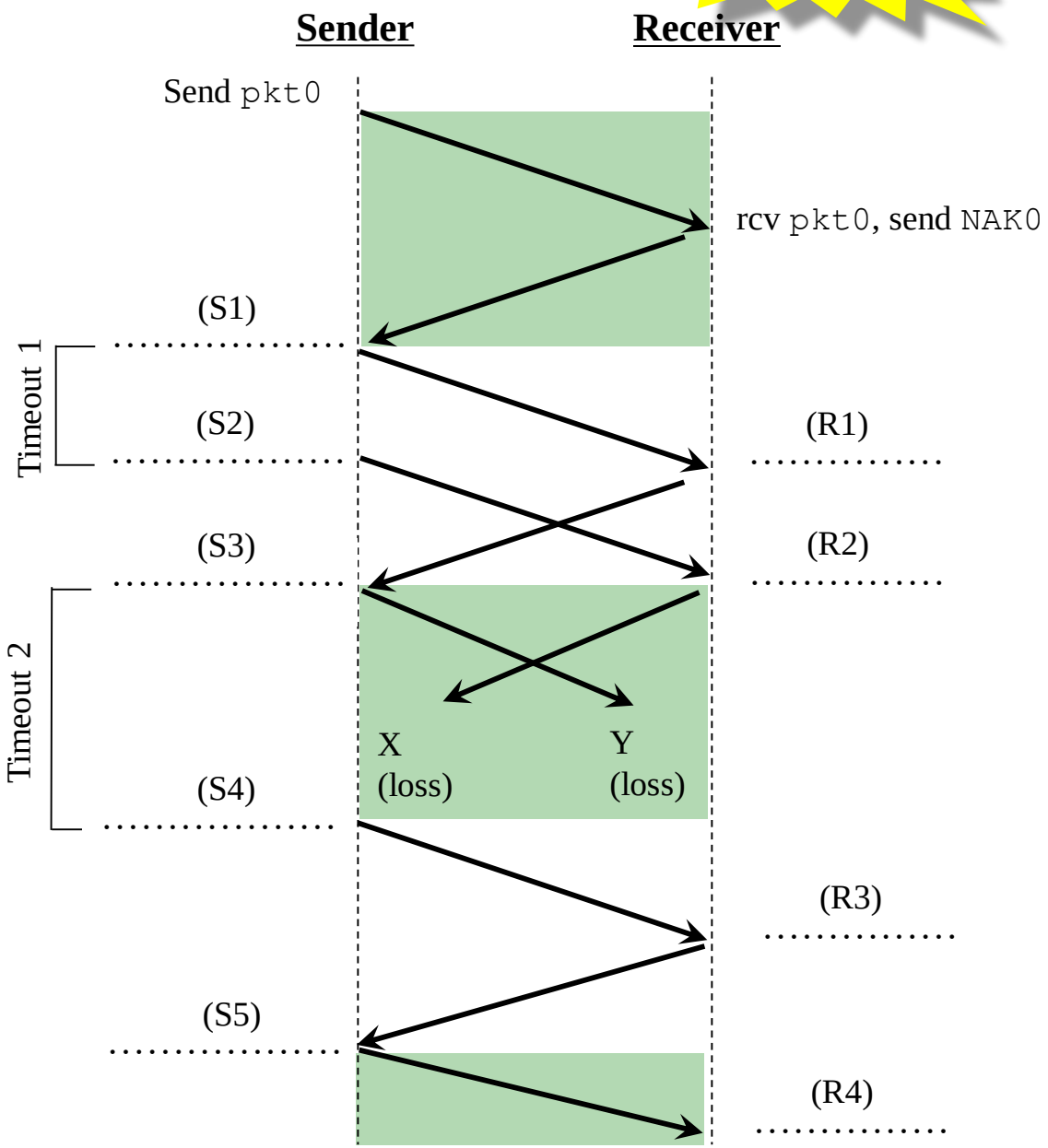
Supposed a sender has 3 packets to be sent to a receiver.

a) Complete the following figure by writing down the best answer for all S# and R#. Assume that no error after S1.

b) What are X and Y?

c) What is the problem between S1 and S2?

d) Is the any discard packet at receiver? Why?



Self-Study

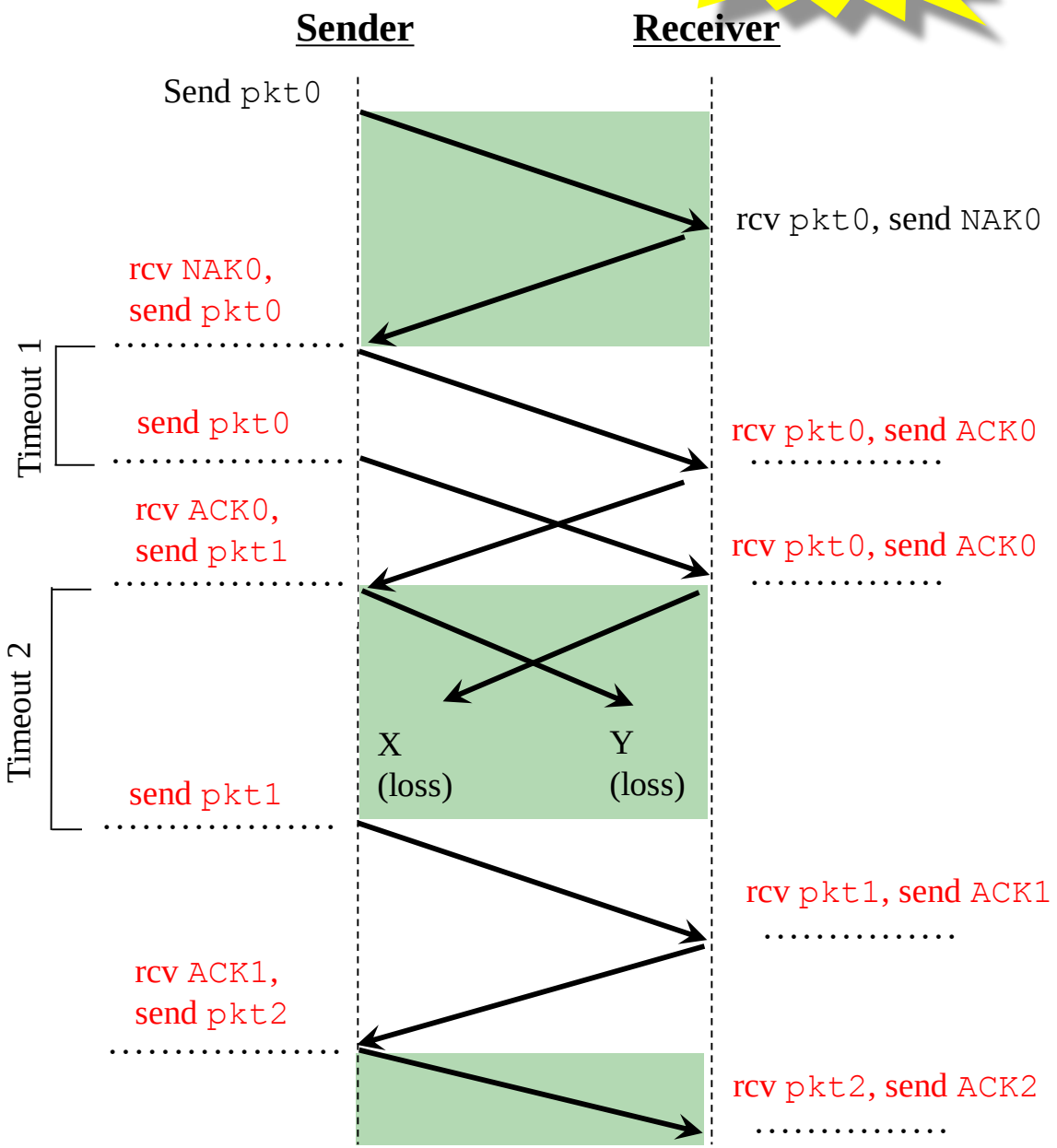
Solutions:

a) (as in figure)

b)

c)

d)



Self-Study

Question:

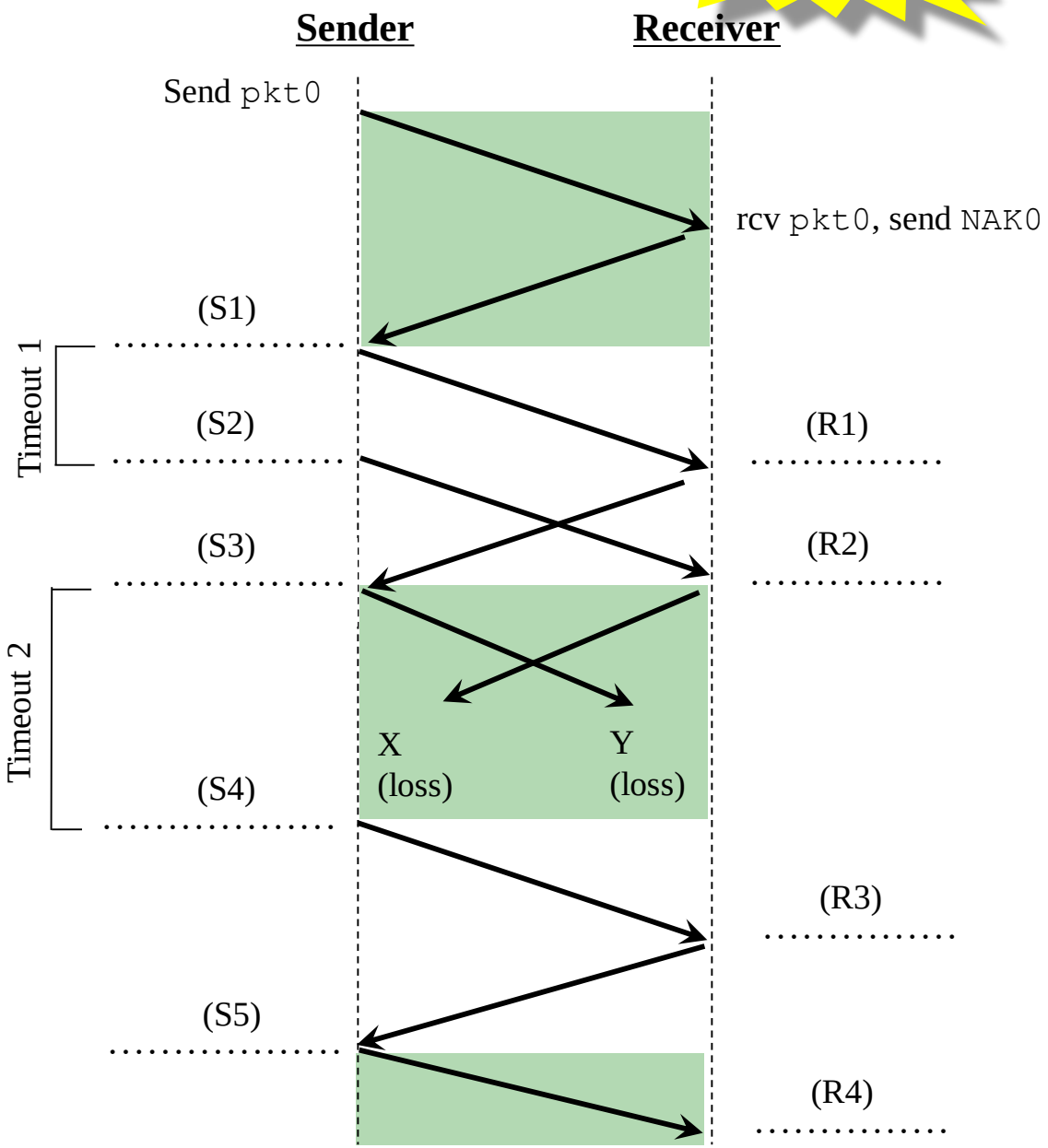
Supposed a sender has 3 packets to be sent to a receiver.

a) Complete the following figure by writing down the best answer for all S# and R#. Assume that no error after S1.

b) What are X and Y?

c) What is the problem between S1 and S2?

d) Is the any discard packet at receiver? Why?

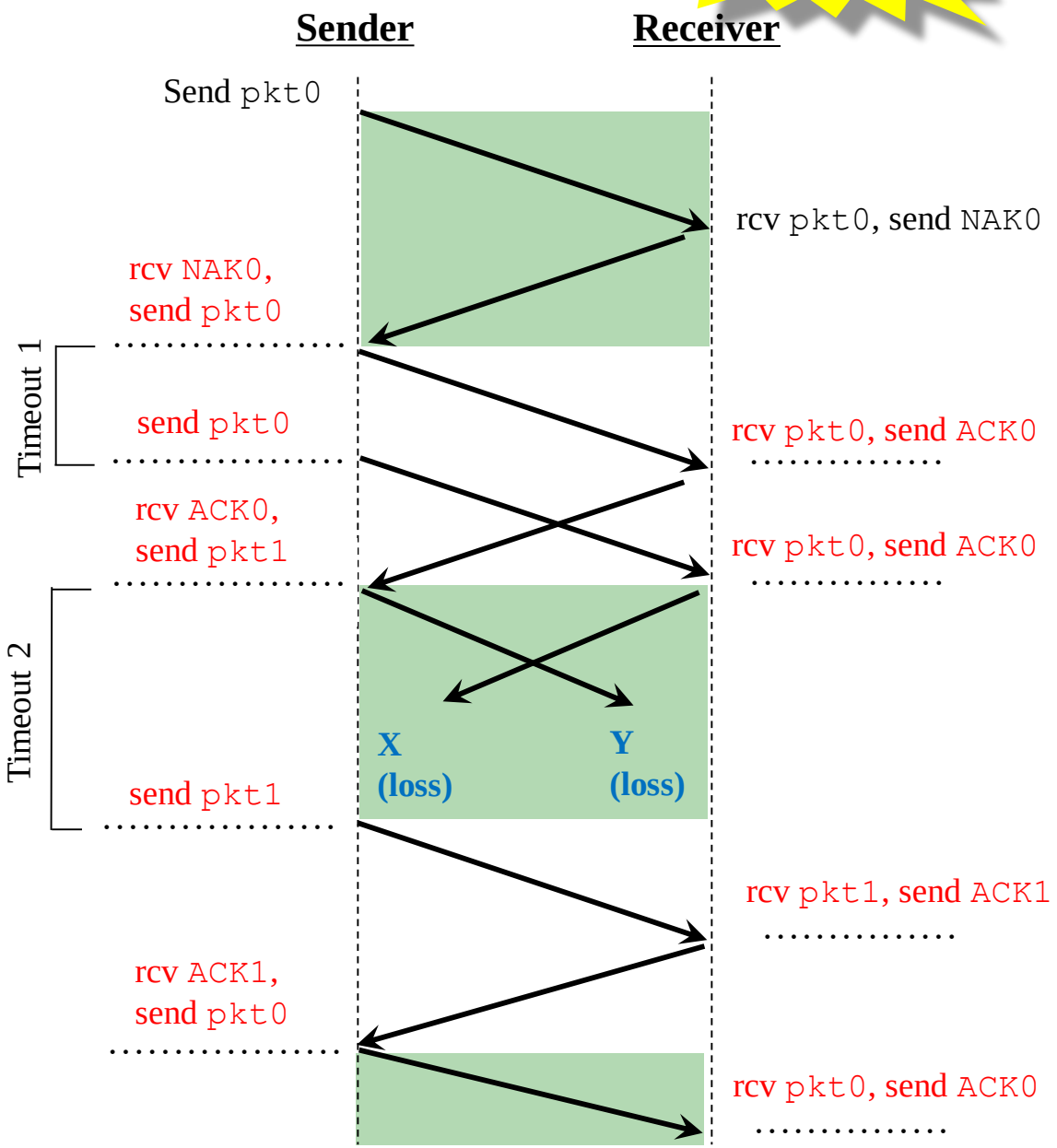


Self-Study

Solutions:

a) (in figure)

b) X – ACK loss,
Y – packet loss



Self-Study

Question:

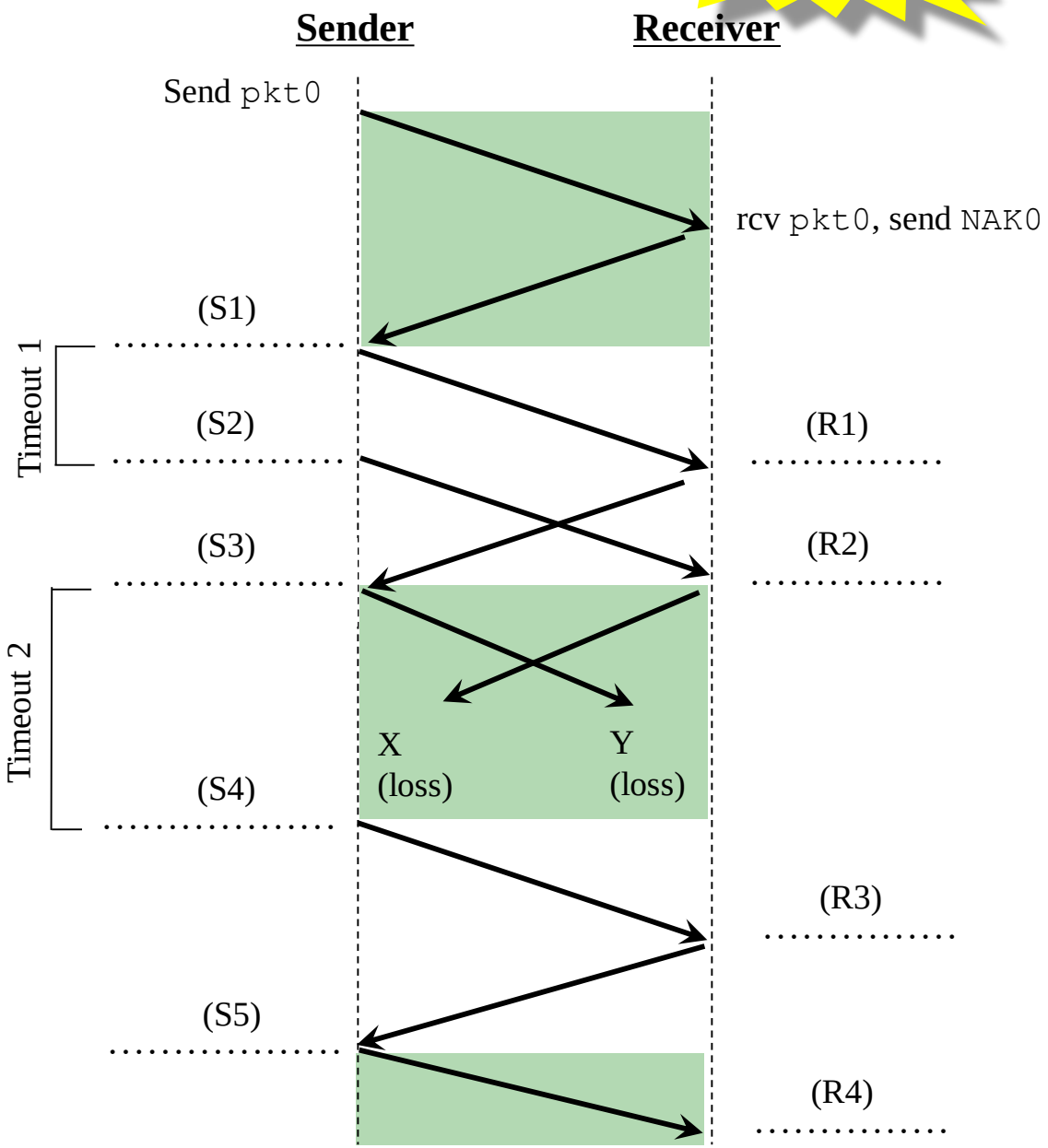
Supposed a sender has 3 packets to be sent to a receiver.

a) Complete the following figure by writing down the best answer for all S# and R#. Assume that no error after S1.

b) What are X and Y?

c) What is the problem between S1 and S2?

d) Is the any discard packet at receiver? Why?



Self-Test

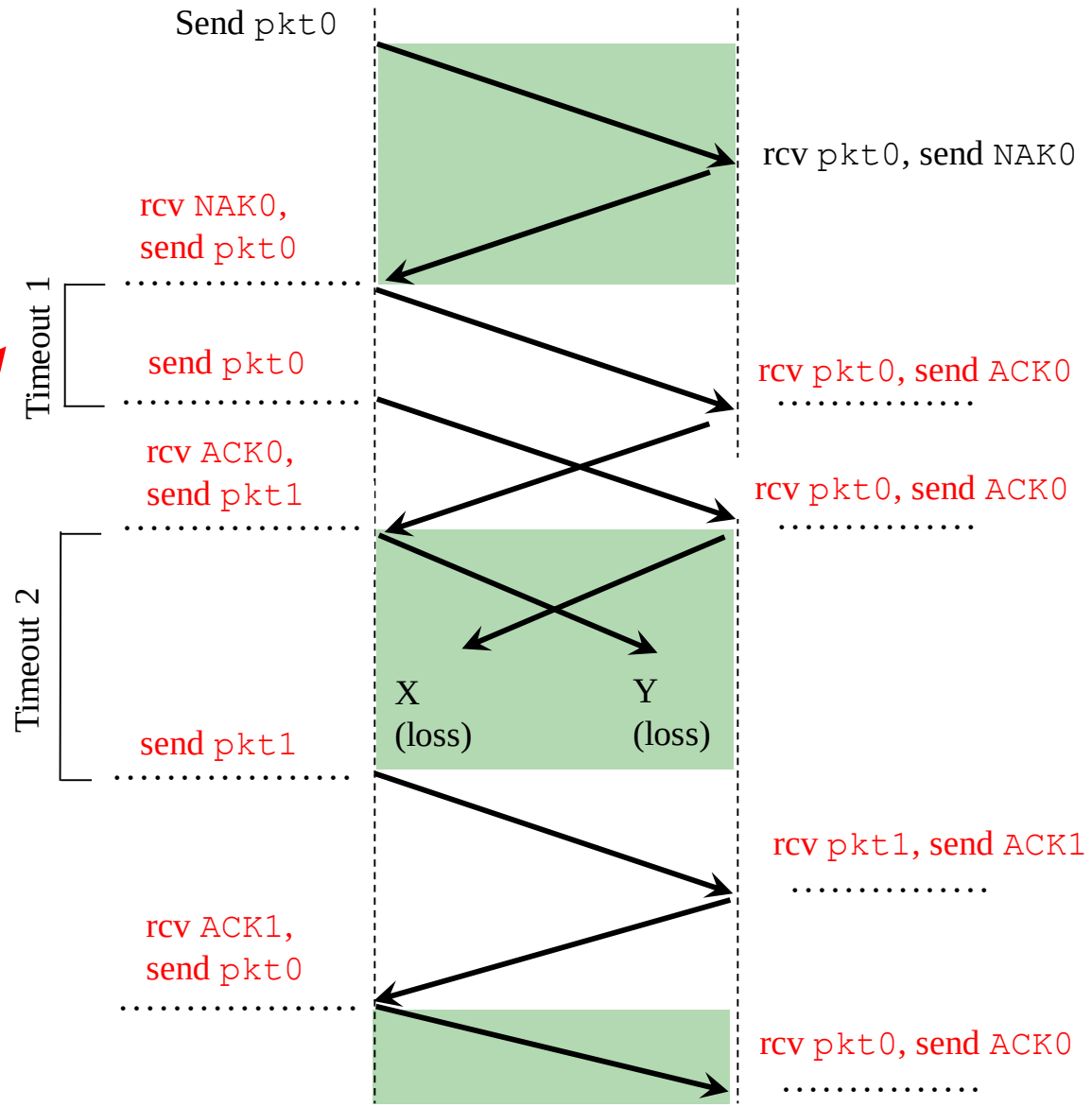
Solutions:

a) (in figure)

b) X – ACK loss,
Y – packet loss

c) Premature timeout

d)



Self-Study

Question:

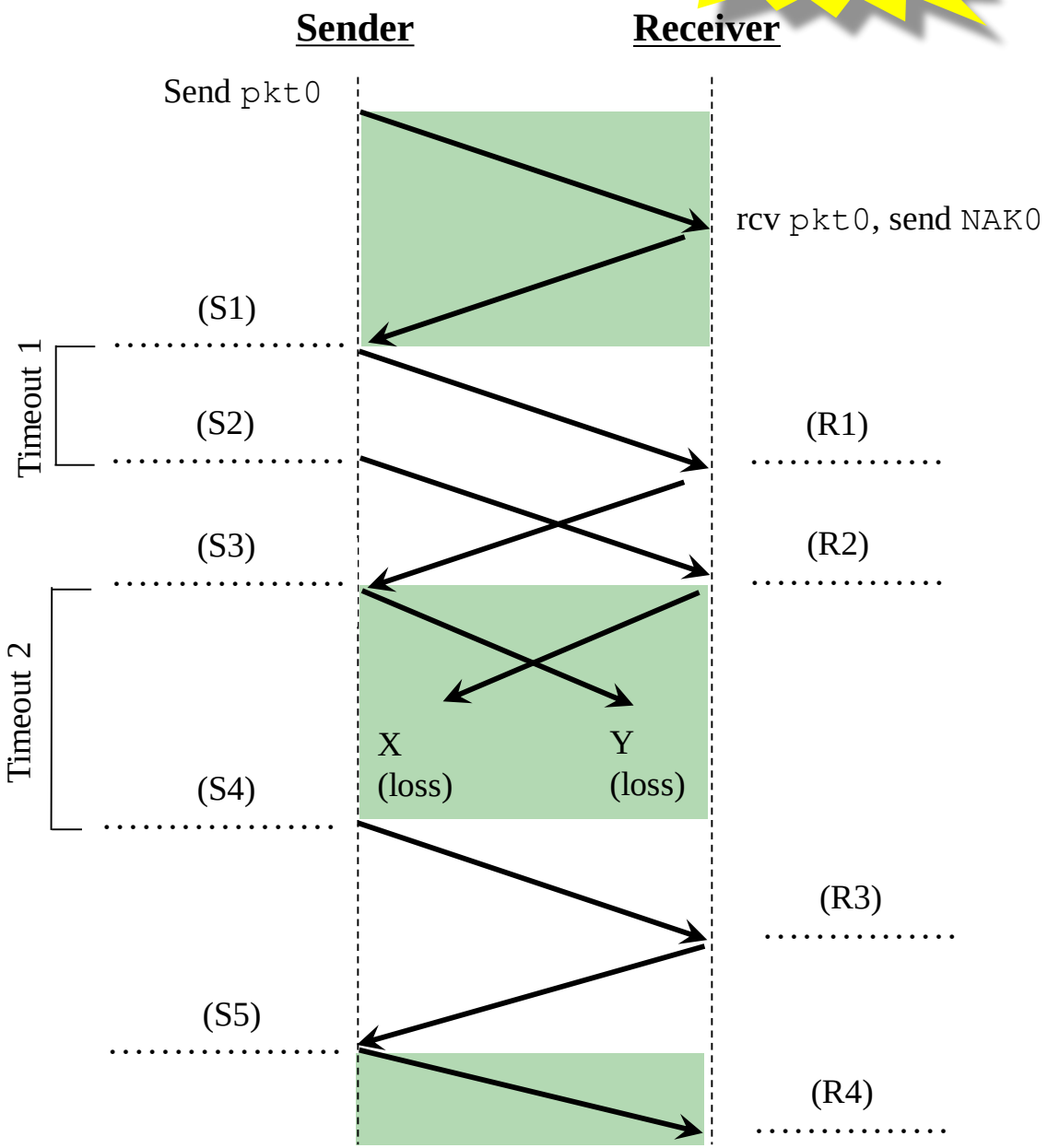
Supposed a sender has 3 packets to be sent to a receiver.

a) Complete the following figure by writing down the best answer for all S# and R#. Assume that no error after S1.

b) What are X and Y?

c) What is the problem between S1 and S2?

d) Is the any discard packet at receiver? Why?



Self-Study

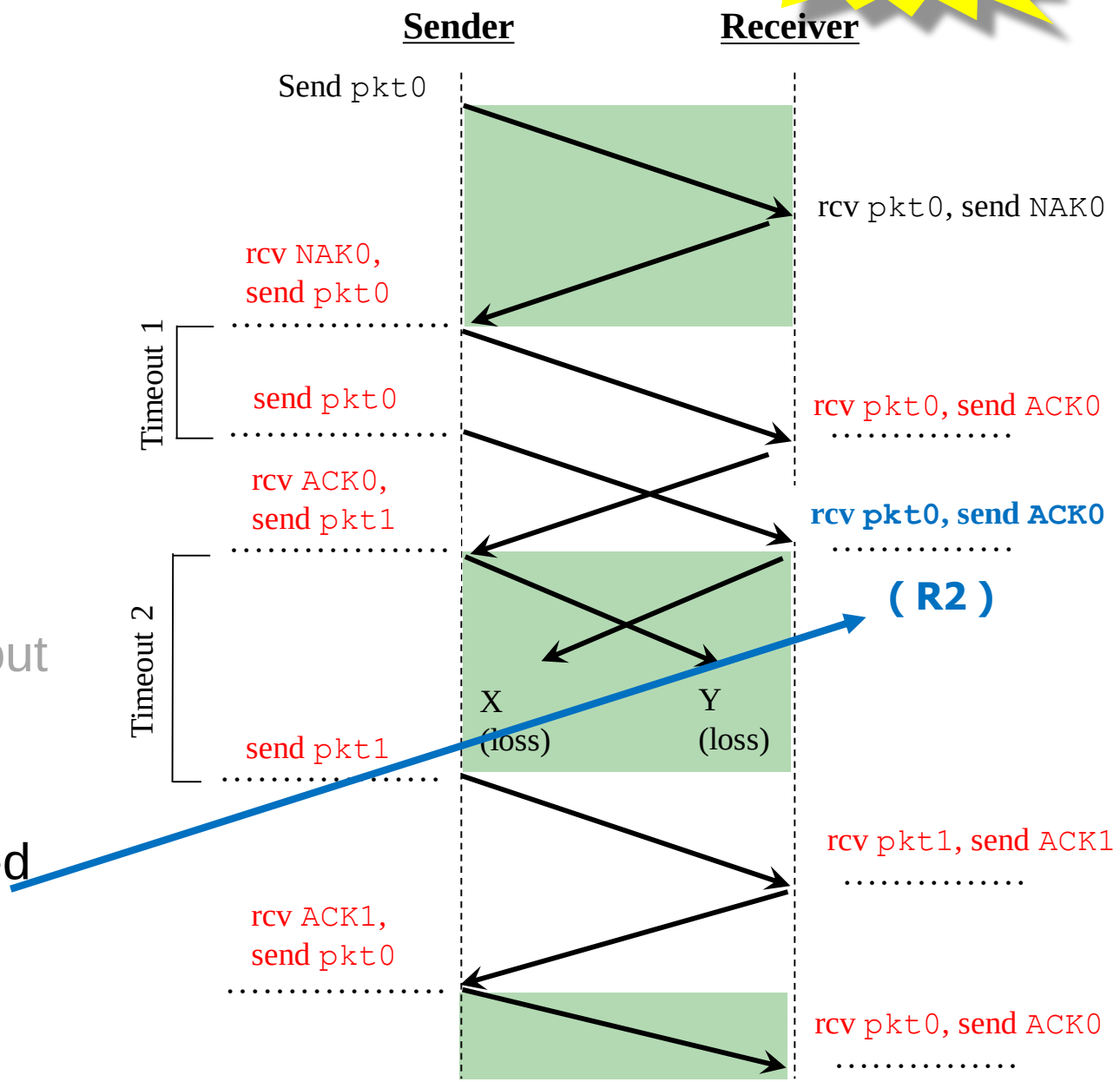
Solutions:

a) (in figure)

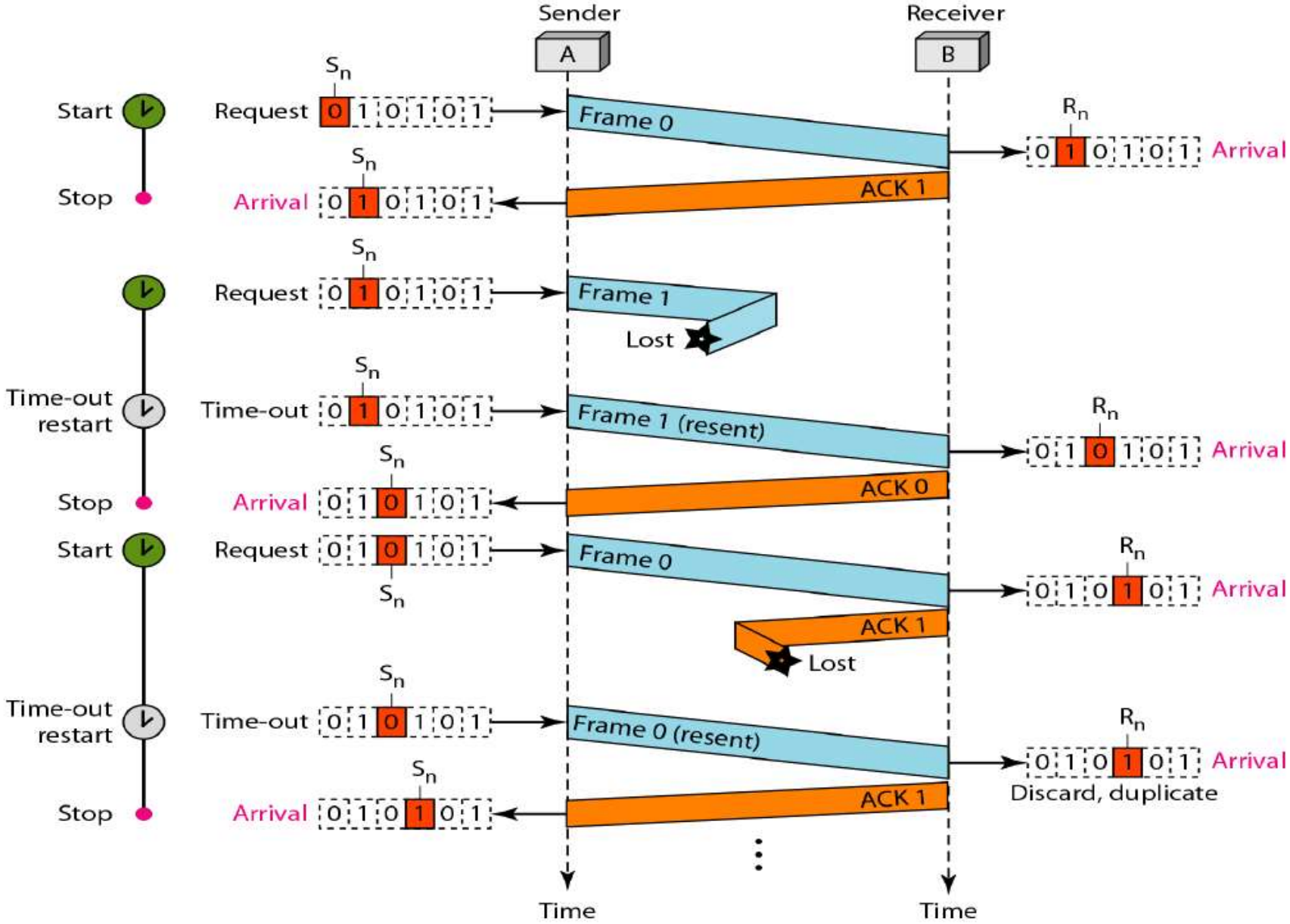
b) X – ACK loss,
Y – packet loss

c) Premature timeout

d) Yes. Duplicate packet 0 detected at **R2**.



Flow Diagram (stop and wait)

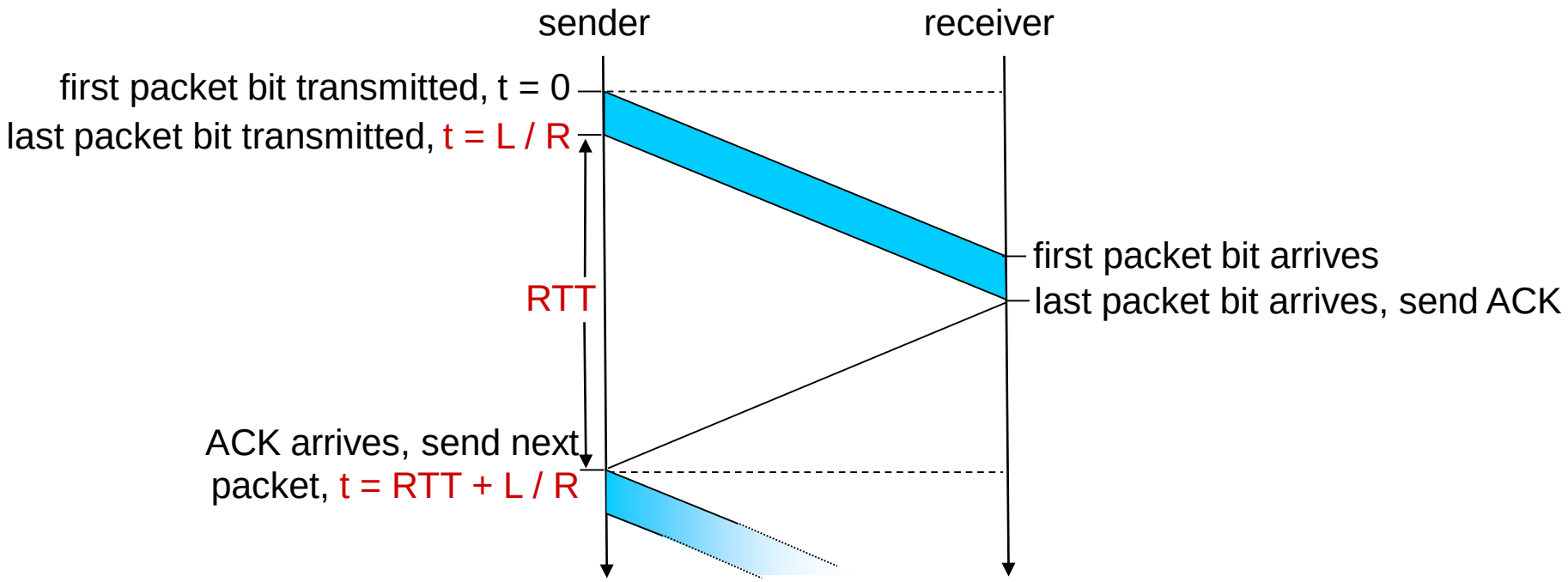


Performance of rdt3.0 (stop-and-wait)

- rdt3.0 is functionally correct, but performance stinks (low)
- U_{sender}: *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs = 0.008\text{ms}$$

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



RTT= 30ms

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027 = 0.027\%$$

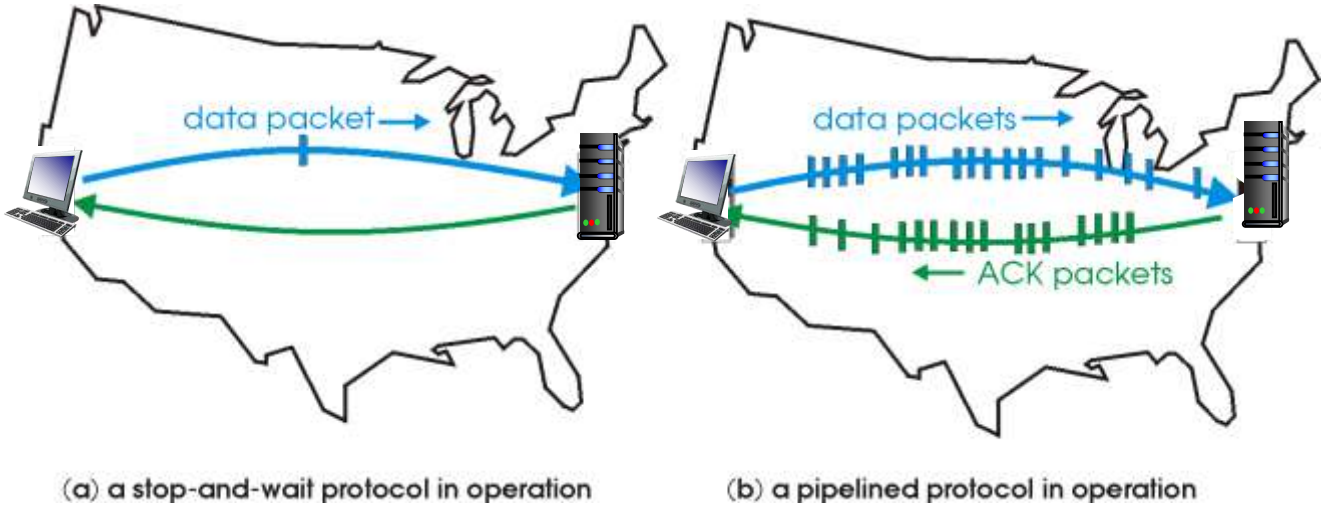
Throughtput = 1Gbps x 0.027% = 270 Kbps

- Problem: network protocol limits use of physical resources/underlying infrastructure (channel)!

rdt3.0: pipelined protocols operation

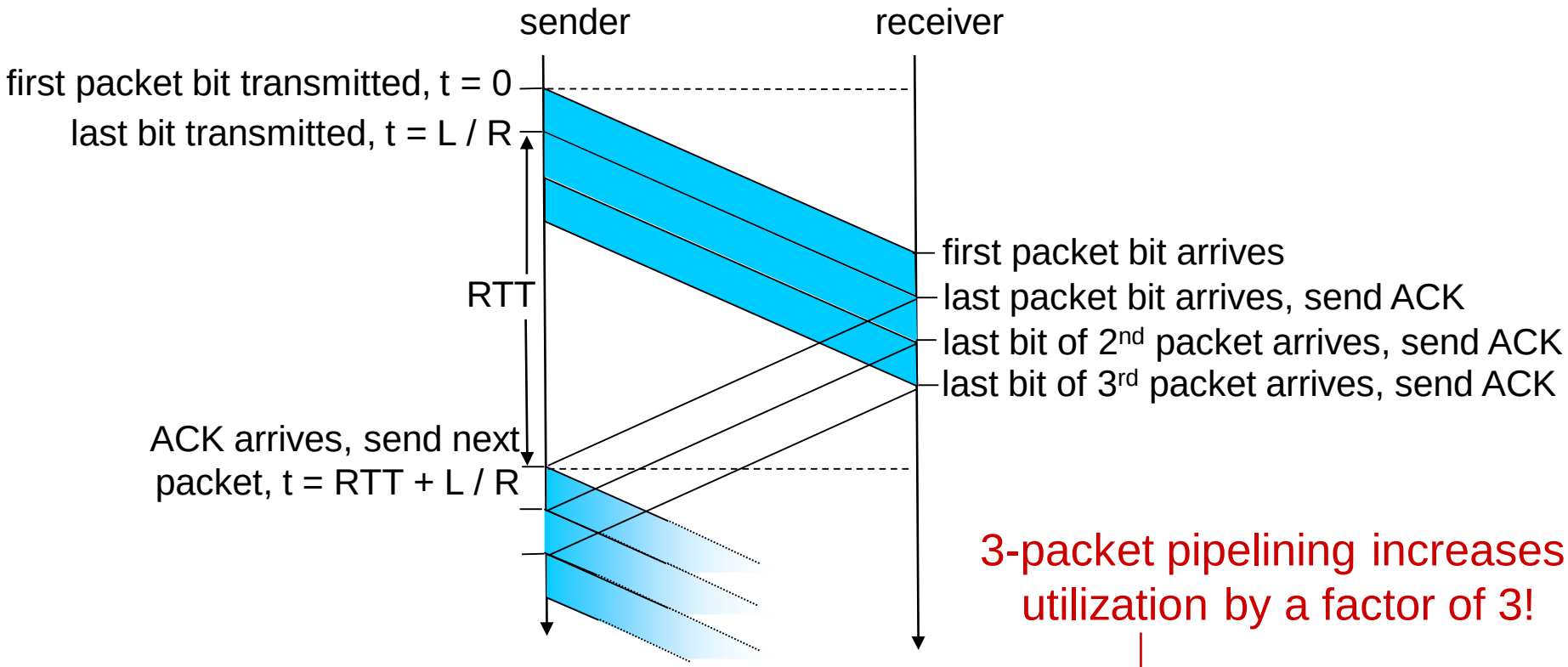
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



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

Pipelining: increased utilization



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

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.00080 = 0.080\%$$

Pipelined protocols: overview

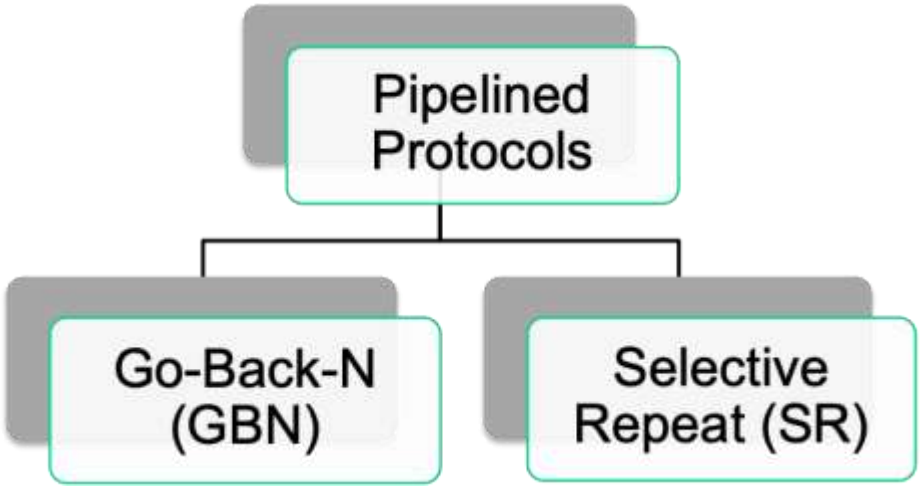


Figure: Two basic approaches of pipelined toward error recovery

- The range of *sequence #* needed and the *buffering* requirements depend on the manner in which a data transfer protocol responds to:
  - lost, corrupted, and overly delayed packets.

Pipelined protocols: overview

Go-back-N:

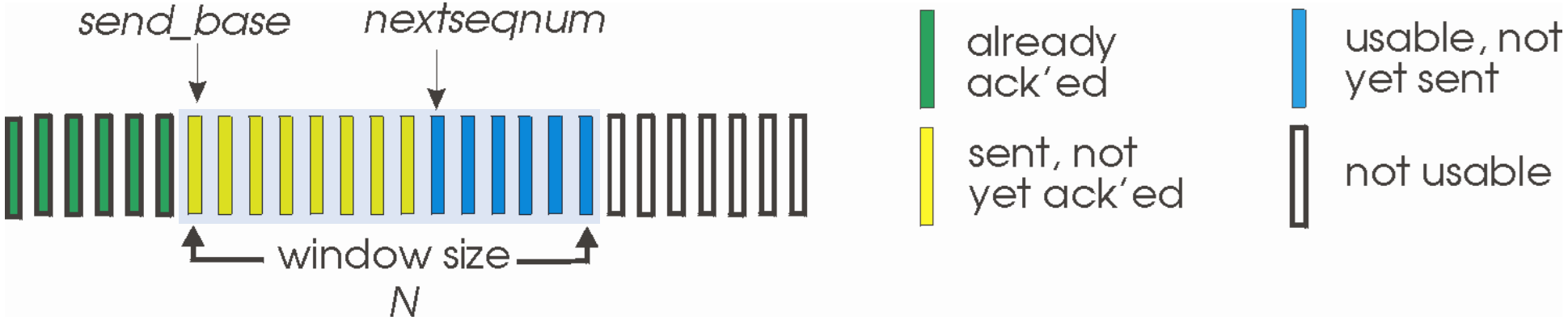
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

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

Go-Back-N (GBN): sender

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

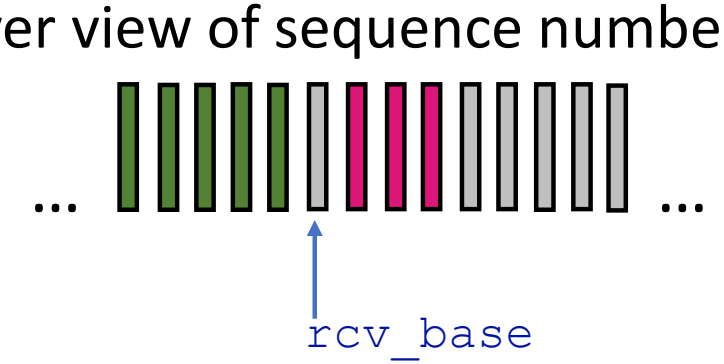


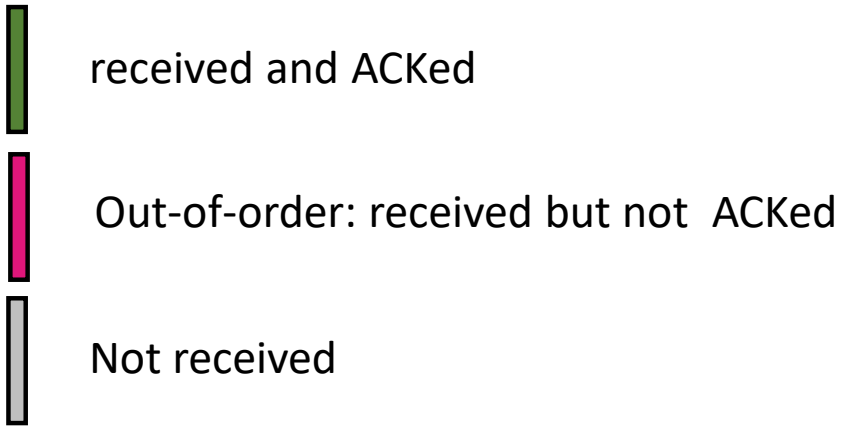
- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

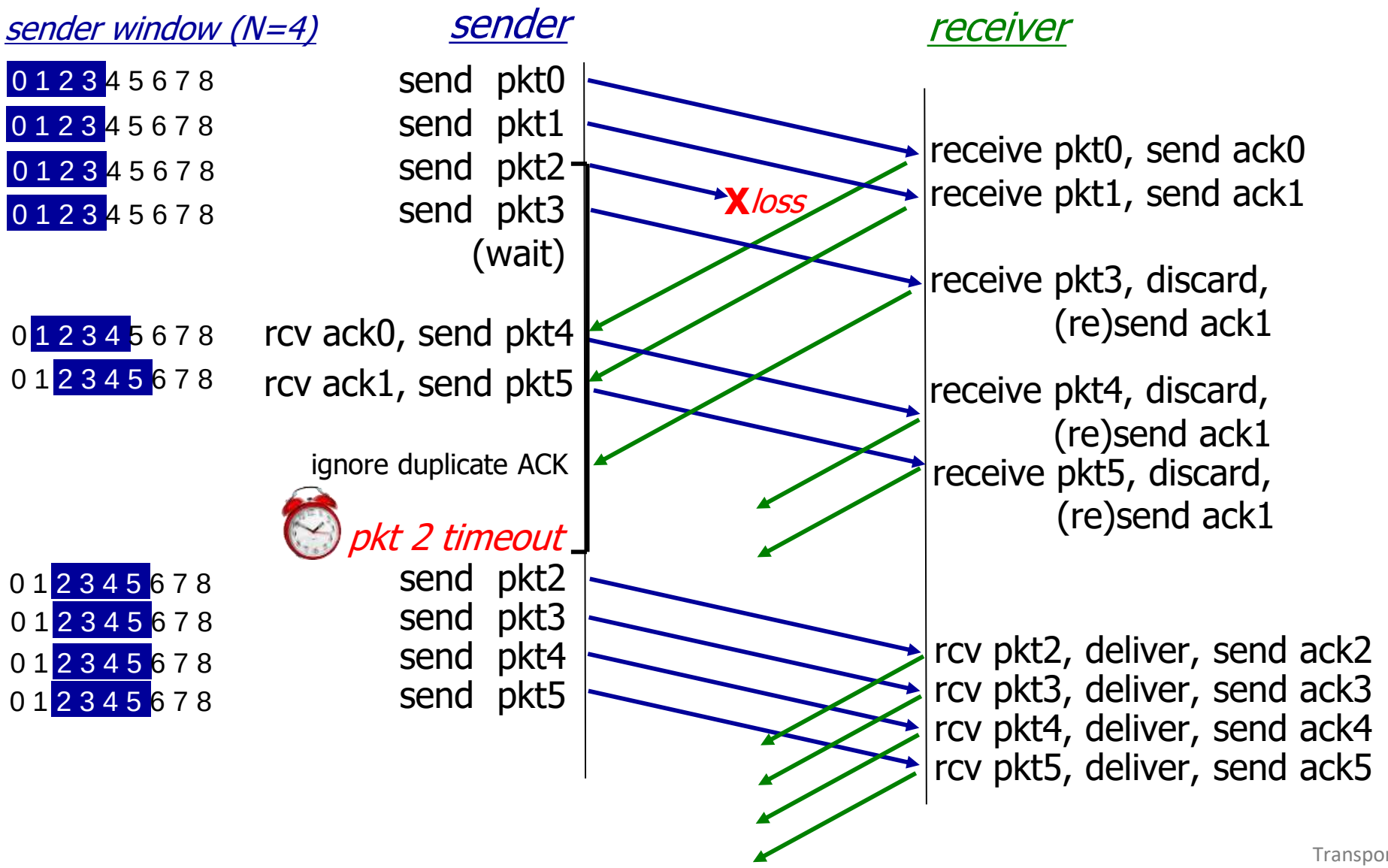
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order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

Receiver view of sequence number space:

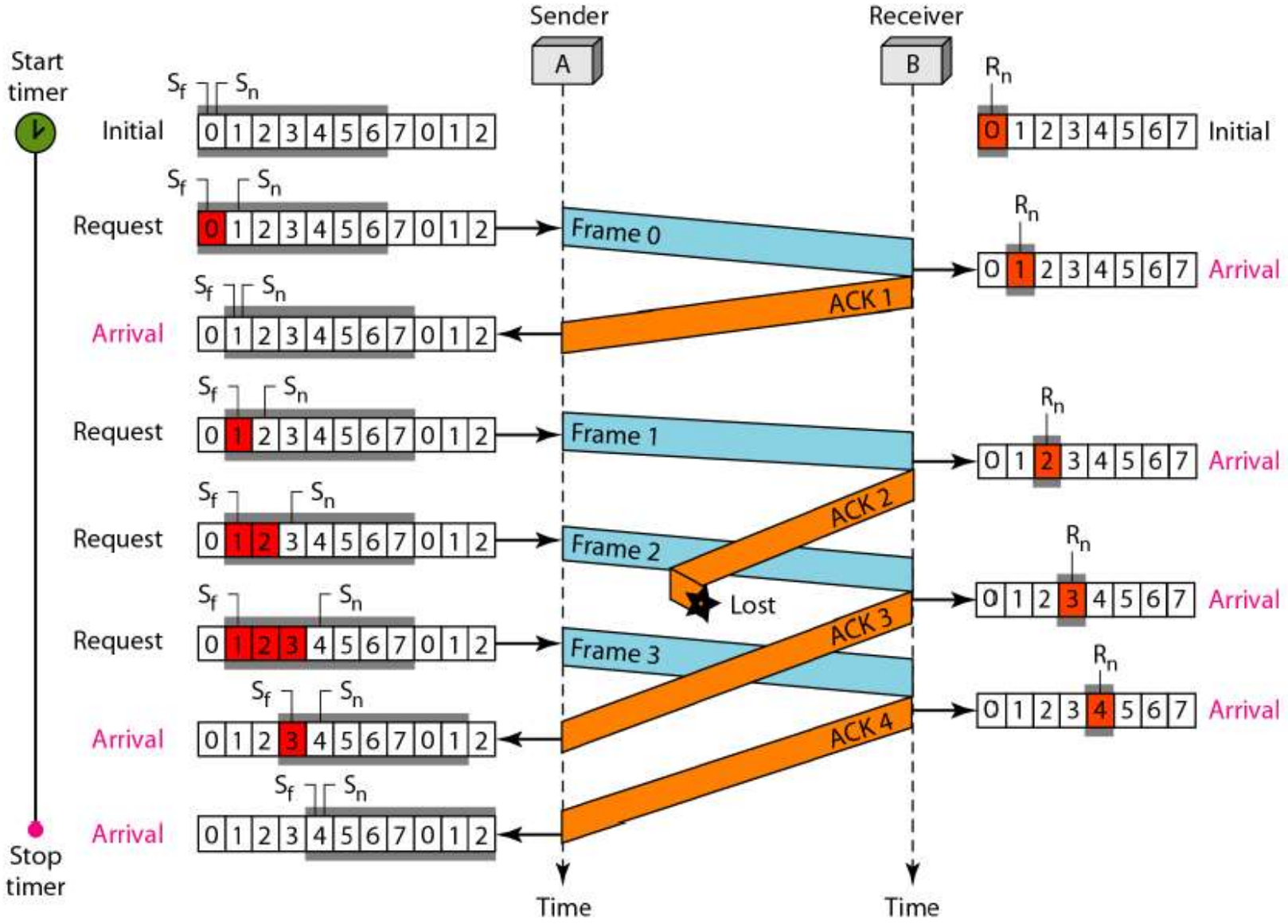




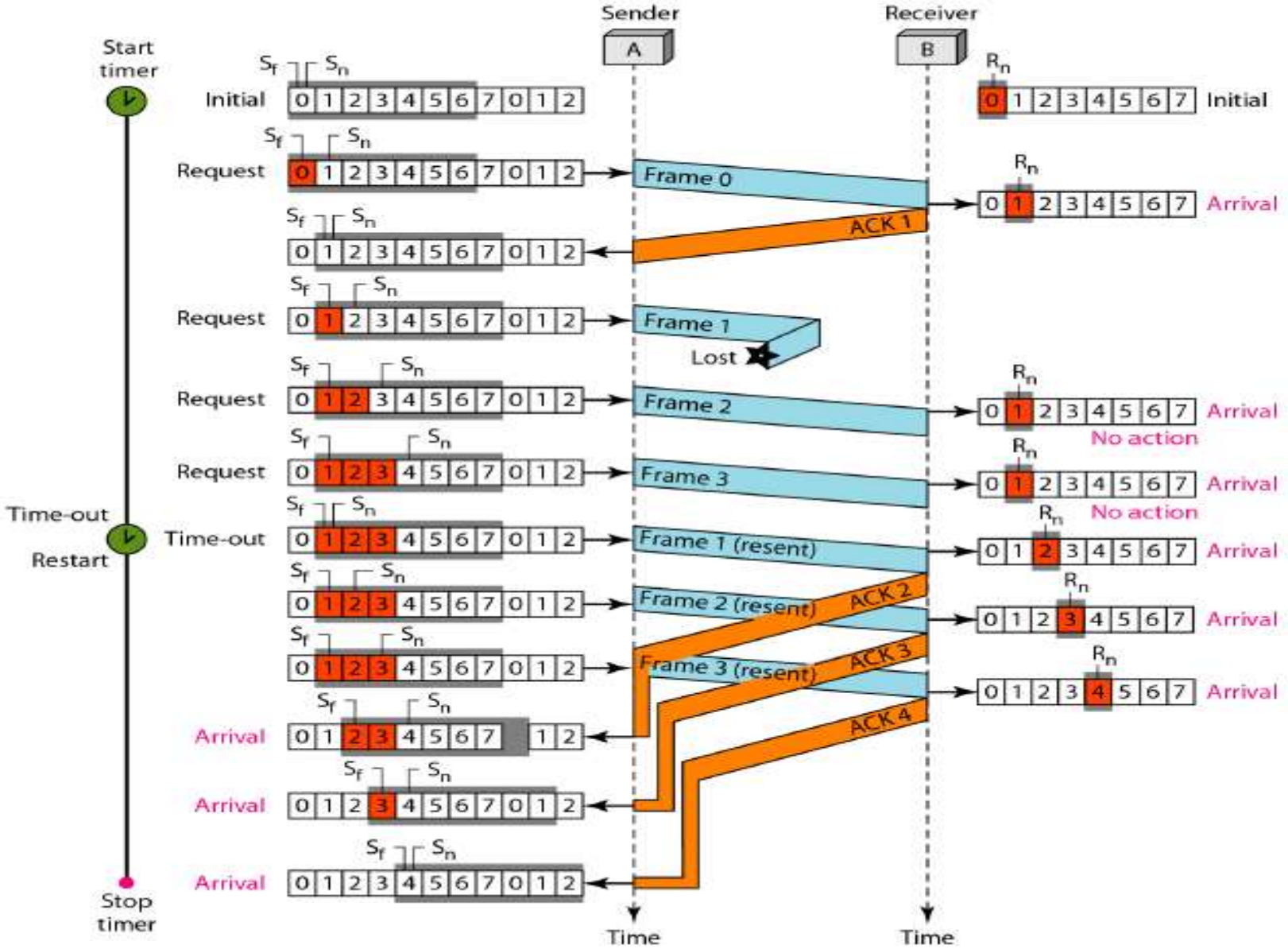
Go-Back-N in action



Flow Diagram (Go-Back-N)



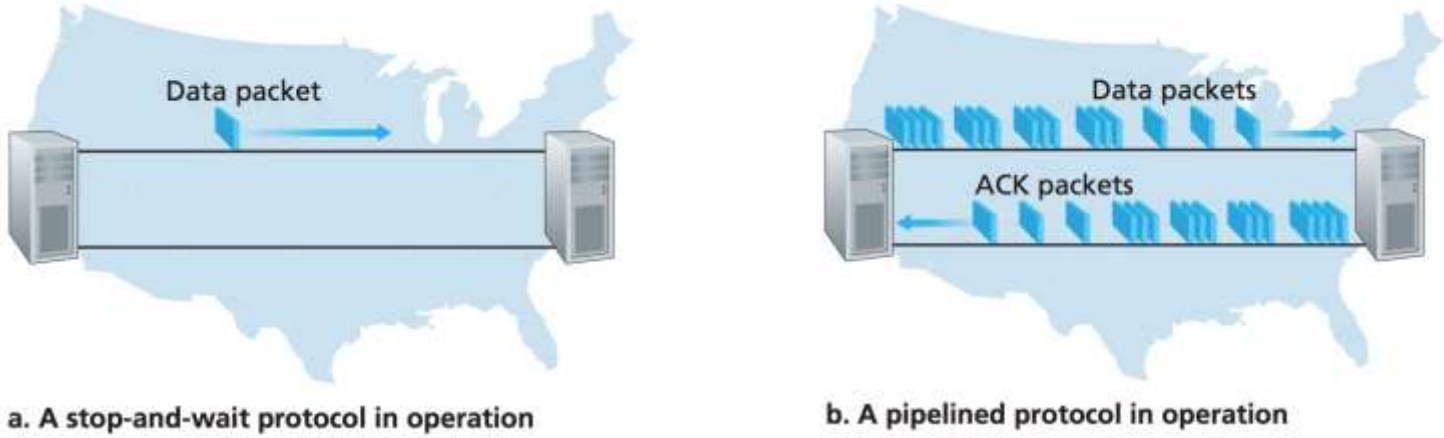
Flow Diagram (Go-Back-N)



Transport Layer: 3-79

Go-Back-N (GBN): Weakness

- The GBN protocol allows the sender to potentially “fill the pipeline” with packets
- thus avoiding the channel utilization problems we noted with stop- and-wait protocols.



a. A stop-and-wait protocol in operation

b. A pipelined protocol in operation

Go-Back-N (GBN): Weakness

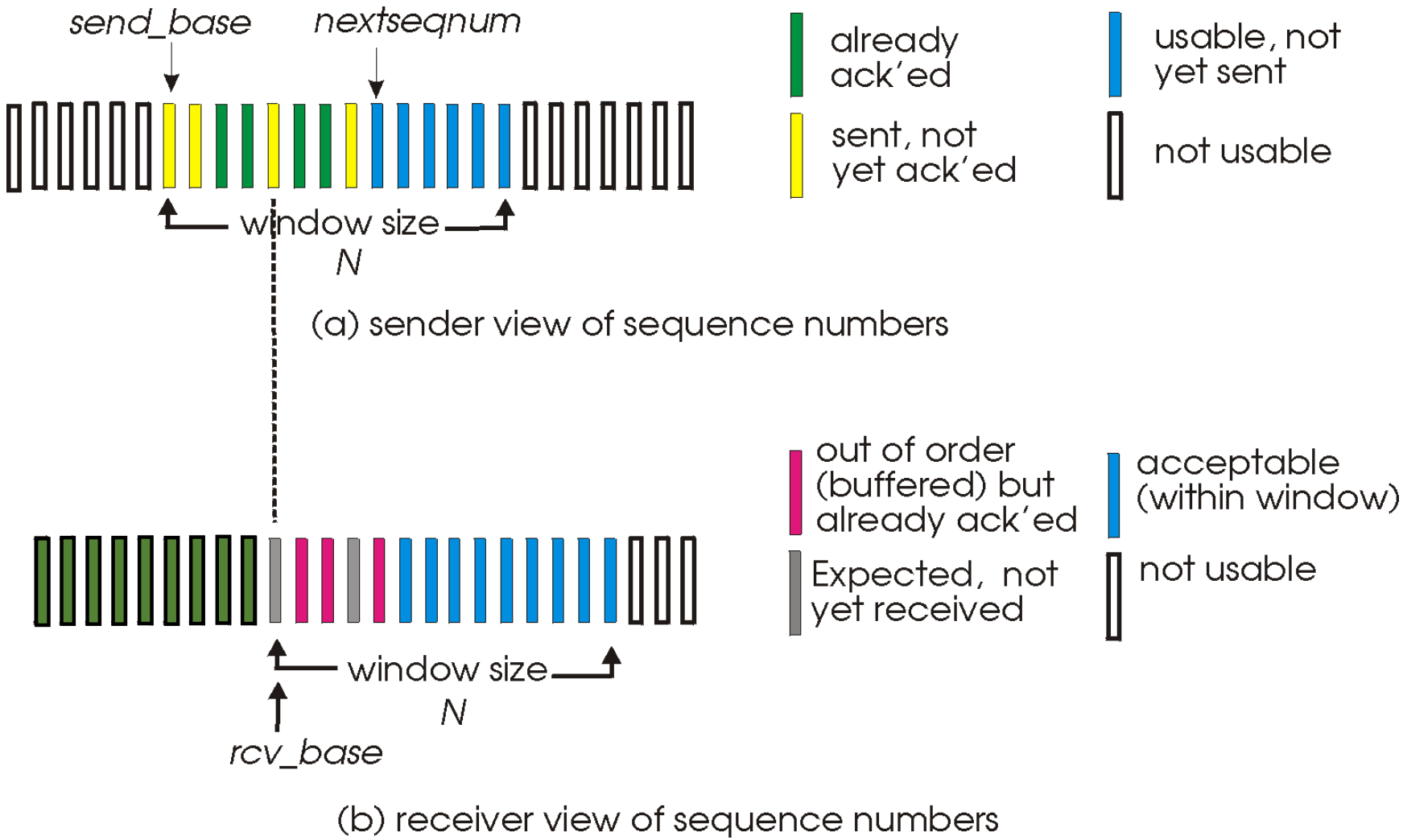
- BUT … GBN itself can suffers from performance problems;
- When the *window size* and *bandwidth-delay* are *both large*, many packets can be in the pipeline.
  - A single packet error can thus cause GBN to *retransmit a large number of packets*, *many unnecessarily*.
  - As the probability of *channel errors increases*, the *pipeline* can be *filled with* these *unnecessary retransmissions*.

- Thus, "**selective-repeat**" protocols *avoid unnecessary retransmissions* by having the sender retransmit only those packets that it suspects were received in error
  - (that is, were lost or corrupted) at the receiver.

Selective repeat (SR)

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

Selective repeat (SR): sender, receiver windows



Selective repeat (SR): 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

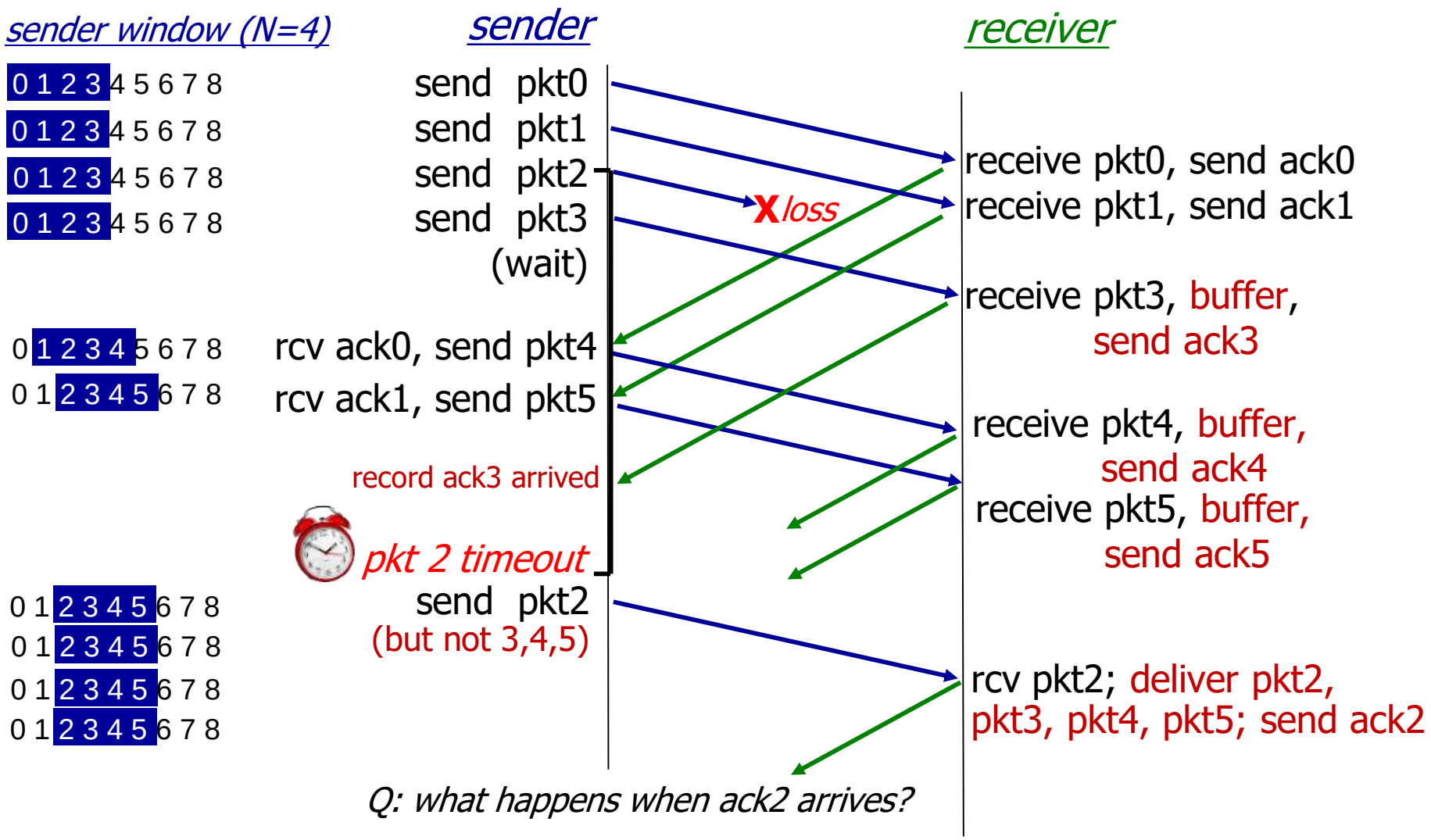
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

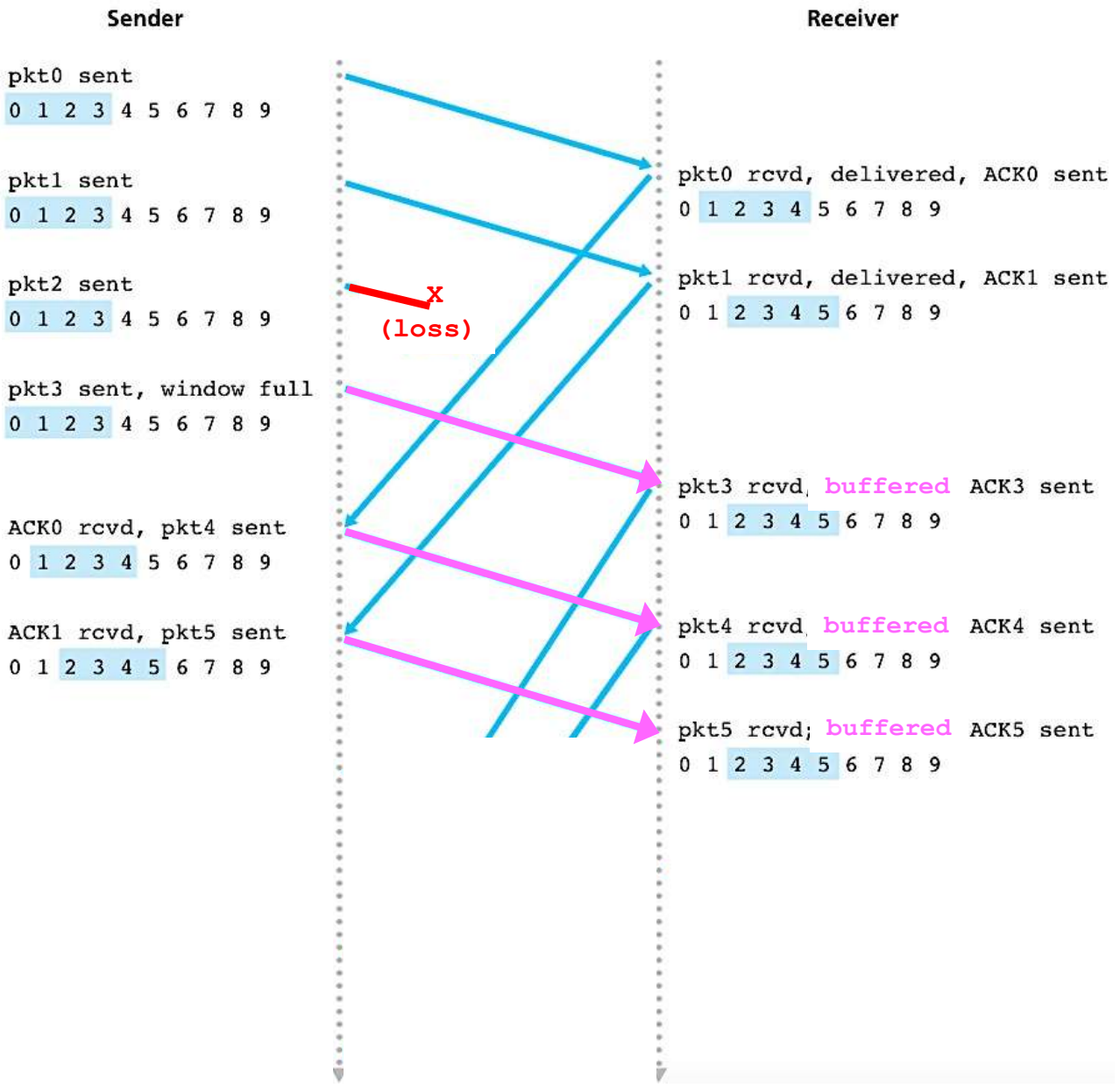
otherwise:

- ignore

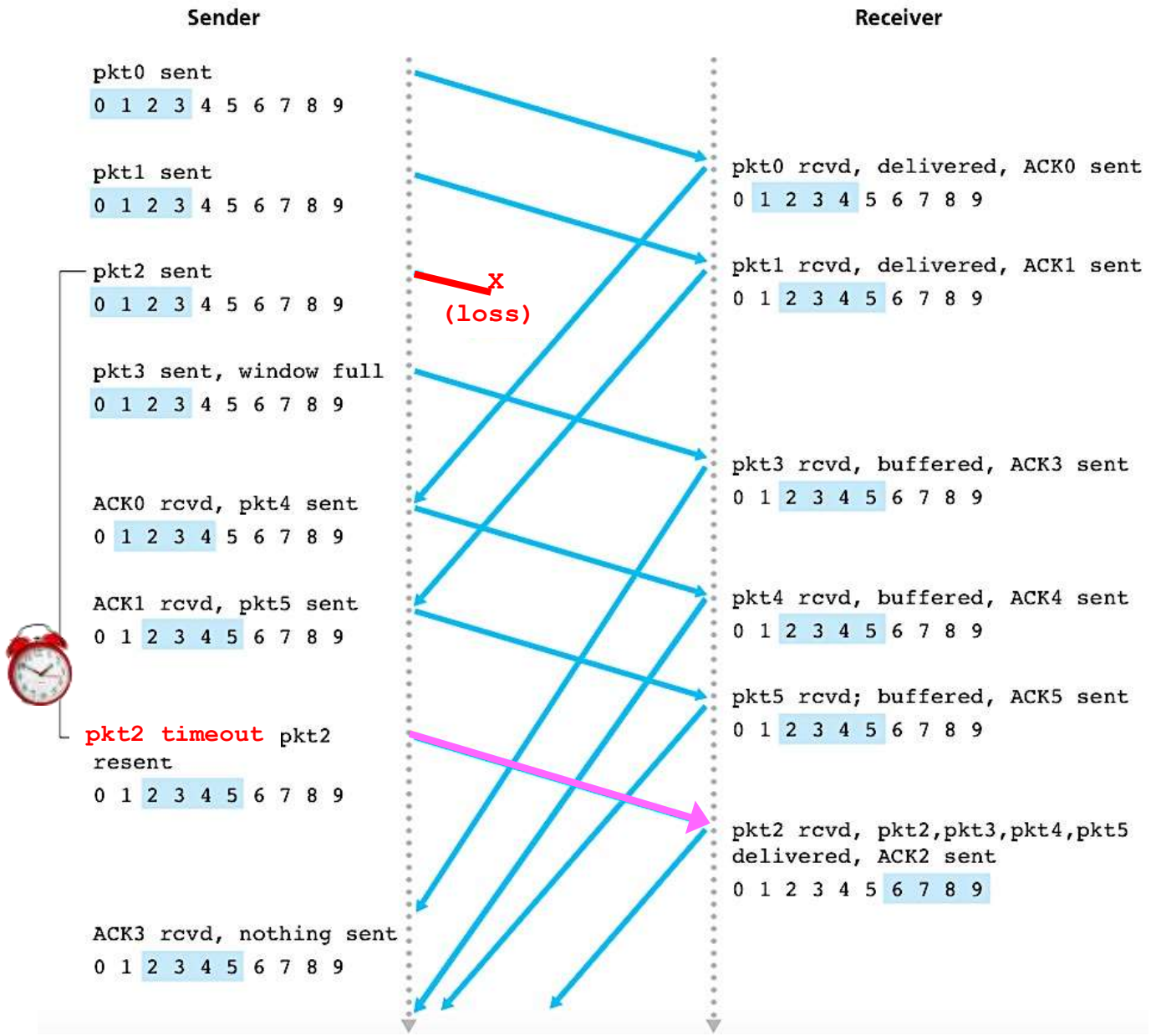
Selective Repeat (SR) in action



Selective Repeat (SR): Operation



Selective Repeat (SR): Operation



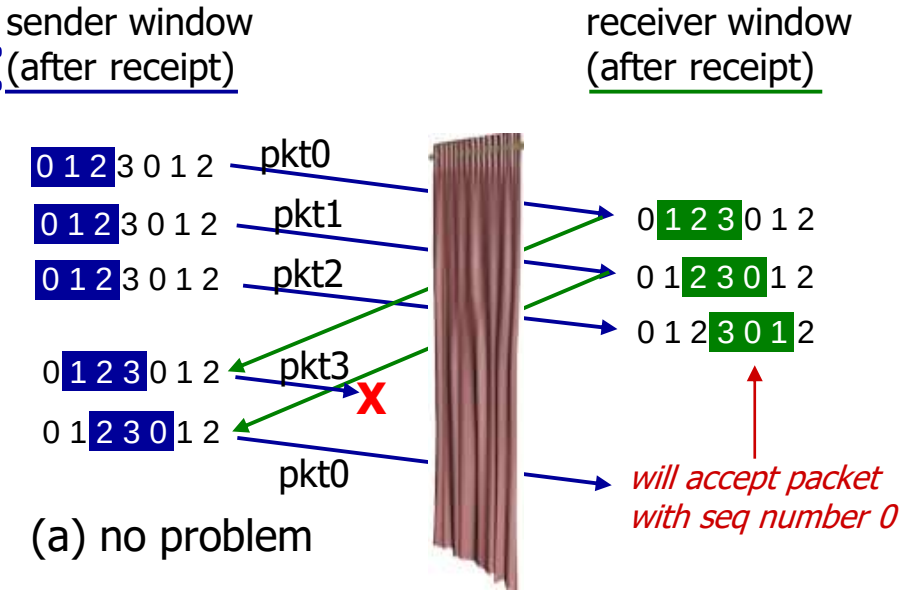
Selective repeat (SR): dilemma

example:

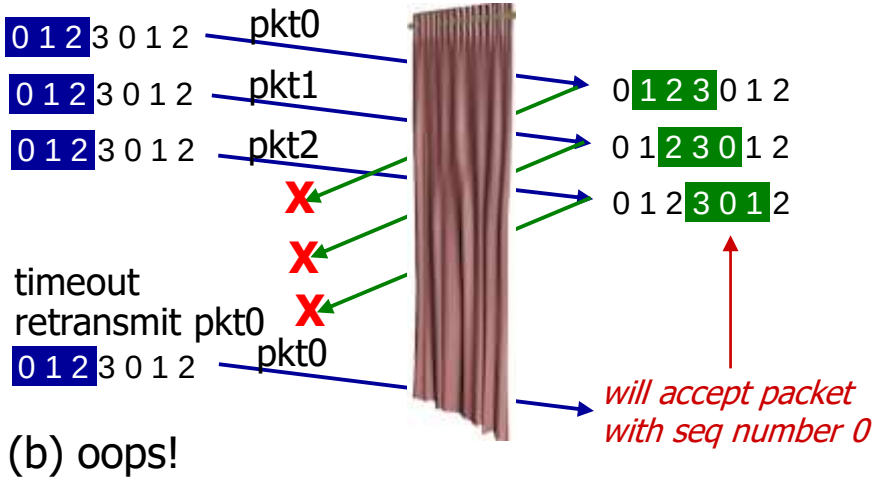
- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

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



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

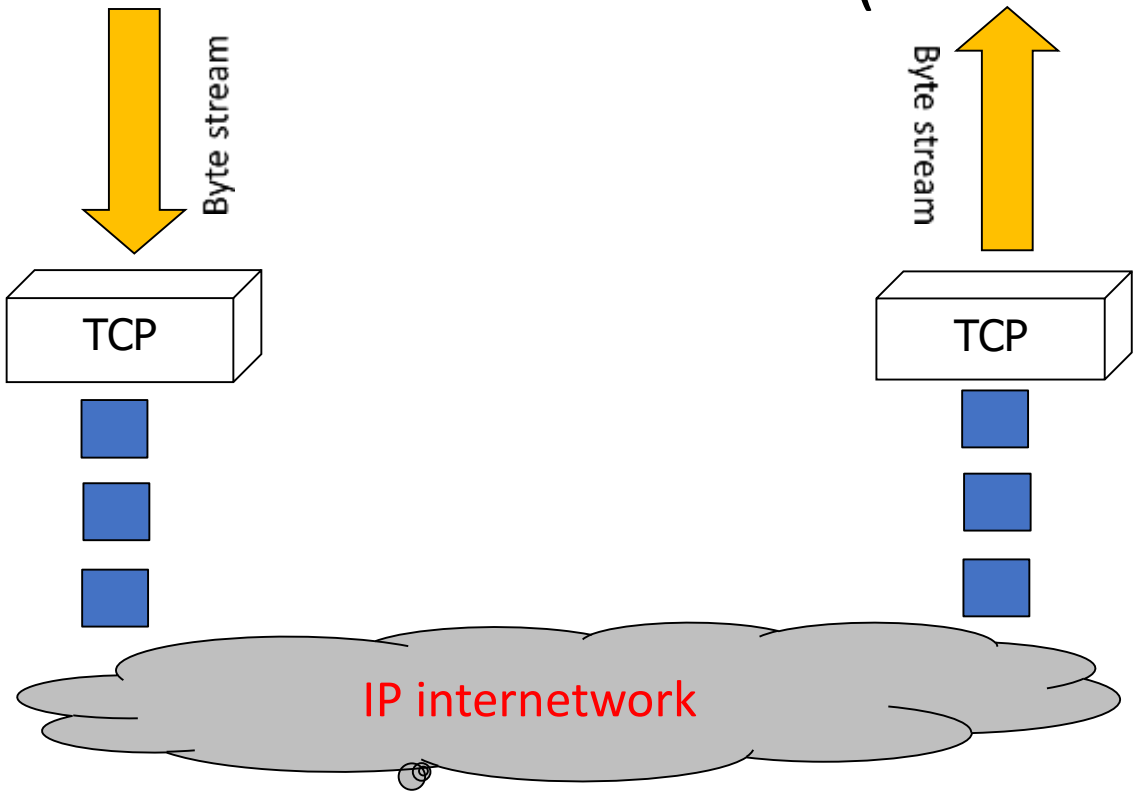


Chapter 3: roadmap

- 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

- TCP = Transmission Control Protocol
- Connection-oriented protocol
- Provides a reliable unicast end-to-end byte stream over an unreliable internetwork (i.e. IP internetwork)



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

- point-to-point:
  - one sender, one receiver

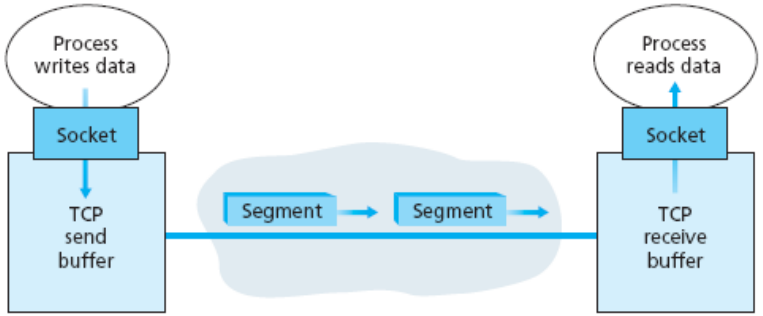
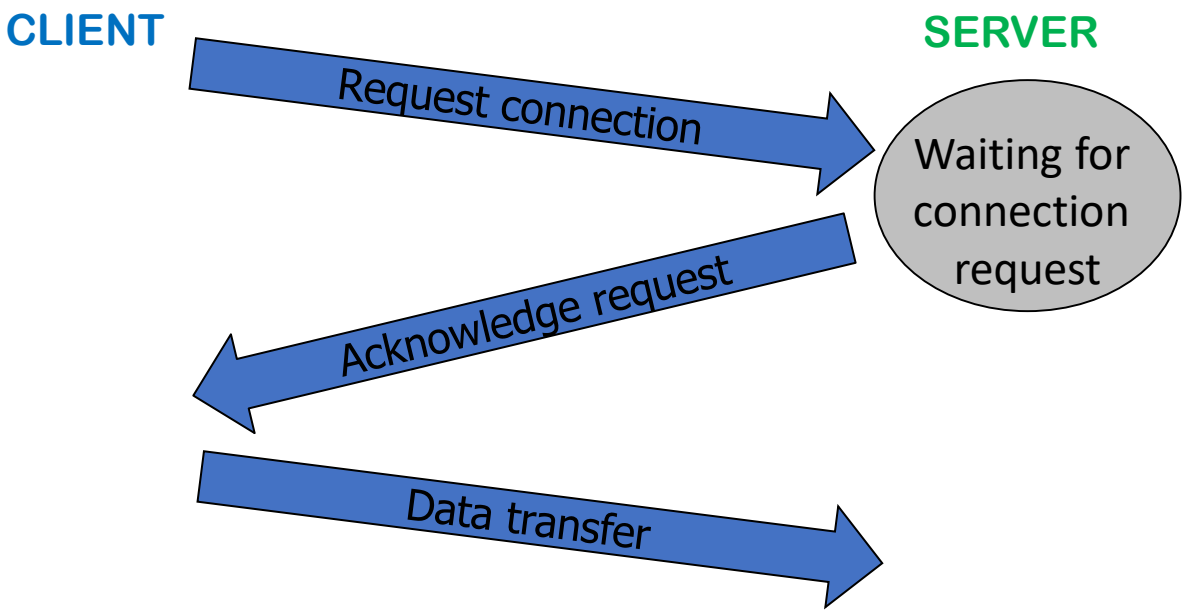


Figure 3.28 ♦ TCP send and receive buffers

  - no "message boundaries"
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control messages) initializers sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP: Overview

- Connection oriented
  - Before any data transfer, TCP establishes a connection:
    - One TCP entity is waiting for a connection (*"server"*)
    - The other TCP entity (*"client"*) contacts the server



TCP: Overview

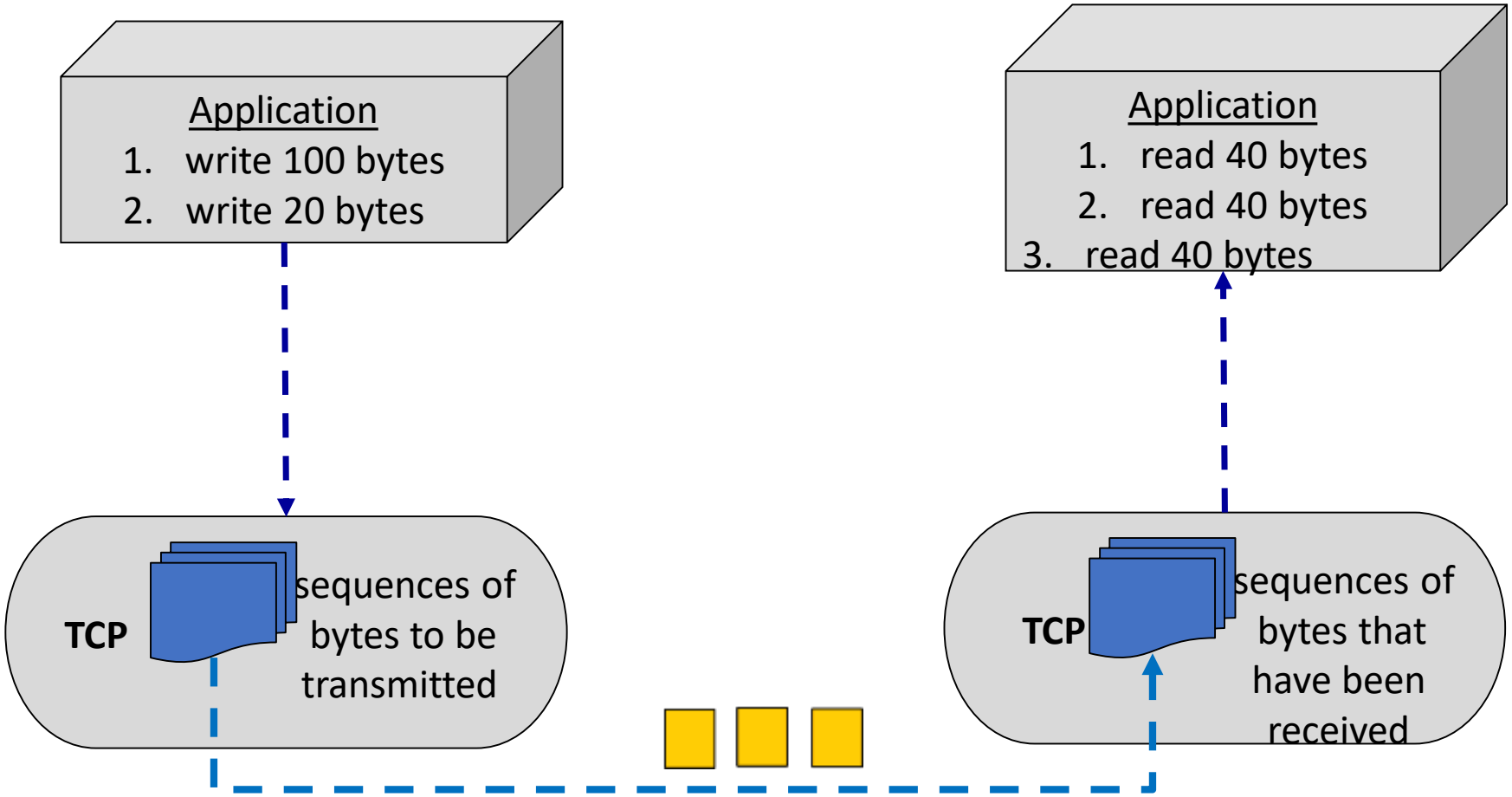
- Reliable
  - Byte stream is broken up into chunks which are called *segments*
    - Receiver sends acknowledgements (ACKs) for segments
    - TCP maintains a *timer*. If an ACK is not received in time, the segment is retransmitted
- Detecting errors
  - TCP has checksums for header and data.
    - Segments with invalid checksums are discarded
  - Each byte that is transmitted has a sequence number
    - If some intermediate sequence number is missing the receiver knows that something has been lost.

TCP: Overview

- Byte stream service
  - To the lower layers TCP handles data in blocks (i.e. the segments)
  - To the higher layers, TCP handles data as a sequence of bytes and does not identify boundaries between bytes
  - So; higher layers do not know about the beginning and end of segments

TCP: Overview

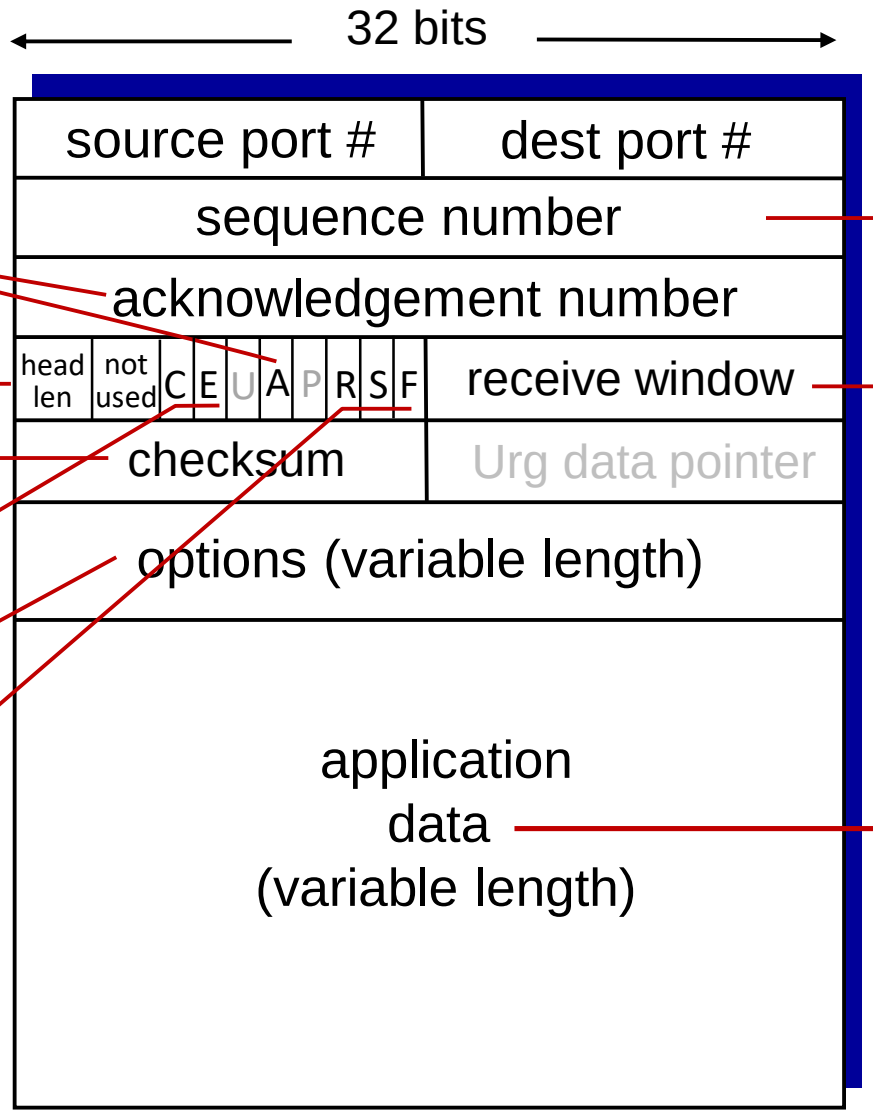
- Byte stream service



TCP segment structure

- The unit of data transfer between devices using TCP is a *segment*
- 20 to 60 byte of *header*, followed by *data*
  - 20 byte header if there are no options
  - and up to 60 bytes if contains some options

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

- Source port address
  - 16 bit
  - Defines the port number of the application program on the host that is sending the segment

- Destination port address
  - 16 bit
  - Defines the port number of the application program on the host that is receiving the segment

- Sequence number
  - 32 bit
  - Defines the number assigned to the first byte of data contained in the segment
  - During connection establishment, a random number generator is used to create an *initial sequence number* (ISN)
- Acknowledgement number
  - 32 bit
  - Defines the byte number that the receiver of the segment is expecting to receive from the other party

TCP sequence numbers, ACKs

Sequence numbers:

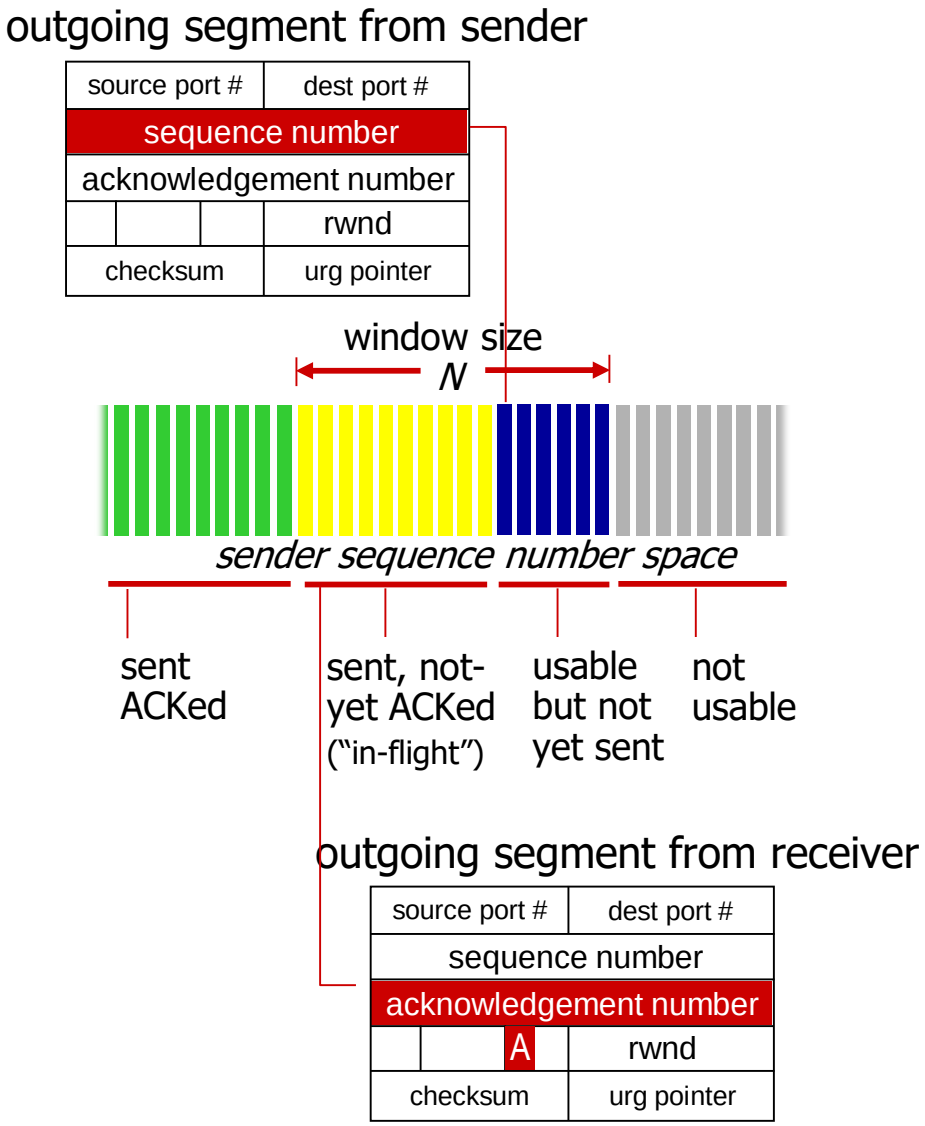
- byte stream "number" of first byte in segment's data

Acknowledgements:

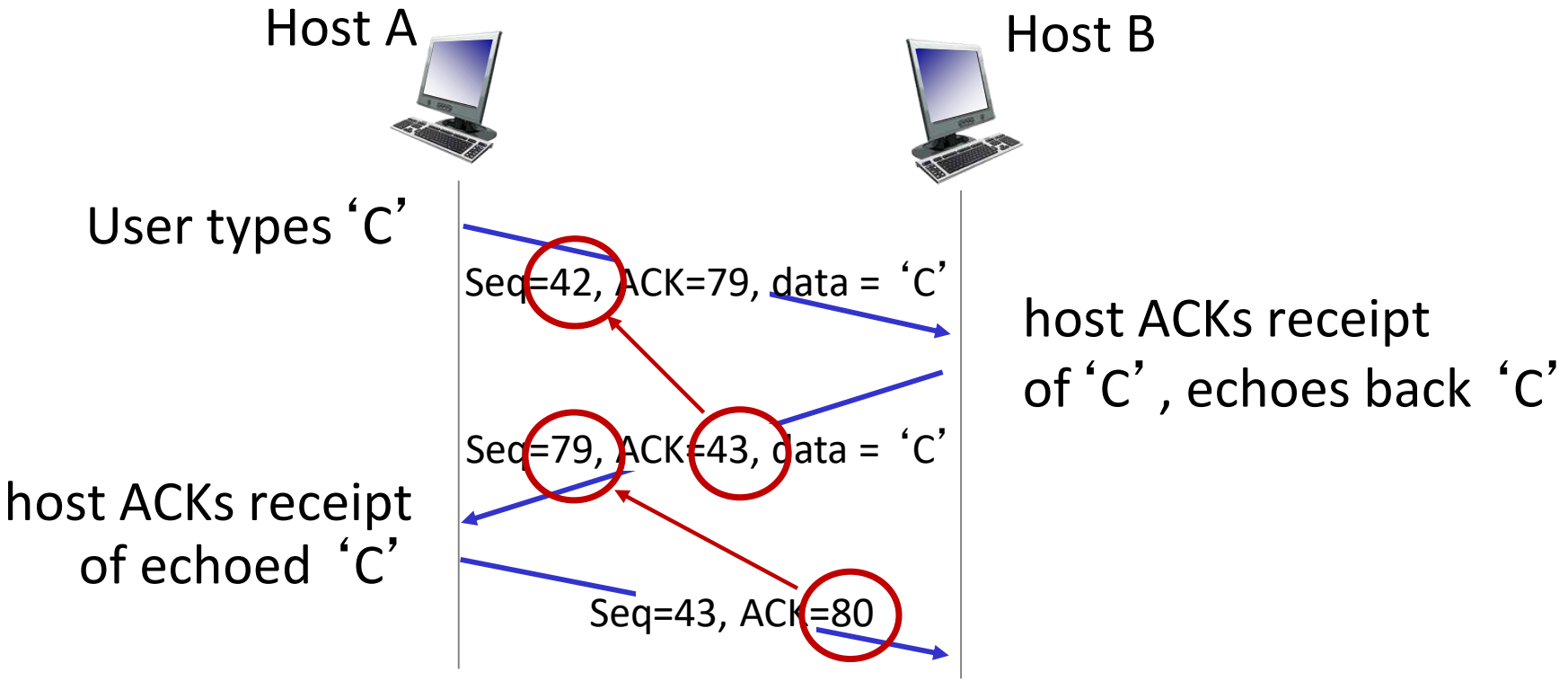
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs

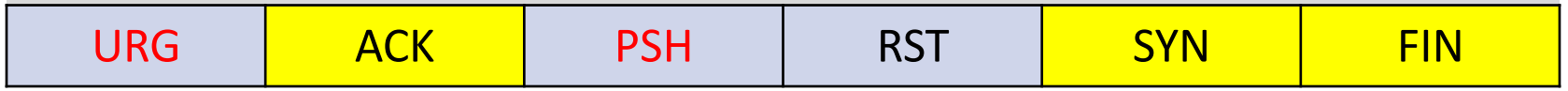


simple telnet scenario

- Control
  - 6 bit
  - Defines 6 control flags

URG: Urgent pointer is valid
(*generally not used*)
ACK: Acknowledgement is valid
PSH: Request for push
(*generally not used*)

RST: Reset the connection
SYN: Synchronize sequence number
FIN: Terminate the connection



- Header length
  - 4 bit
  - Indicates the length of this TCP header
- Reserved
  - 6 bit
  - Reserved for future use

- Window size
  - 16 bit
  - Defines the size of the sliding window, in bytes, that the other party must maintain
- Checksum
  - 16 bit
  - Contains checksum value for error checking
  - Same calculation as UDP
  - Inclusion is mandatory for TCP

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 reliable data transfer

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

let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

duplicate ACK, indicating seq. # of next expected byte
Due to some reason expected seq. # is not received at receiver)

TCP round trip time, timeout

Q: how to set TCP timeout value?

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

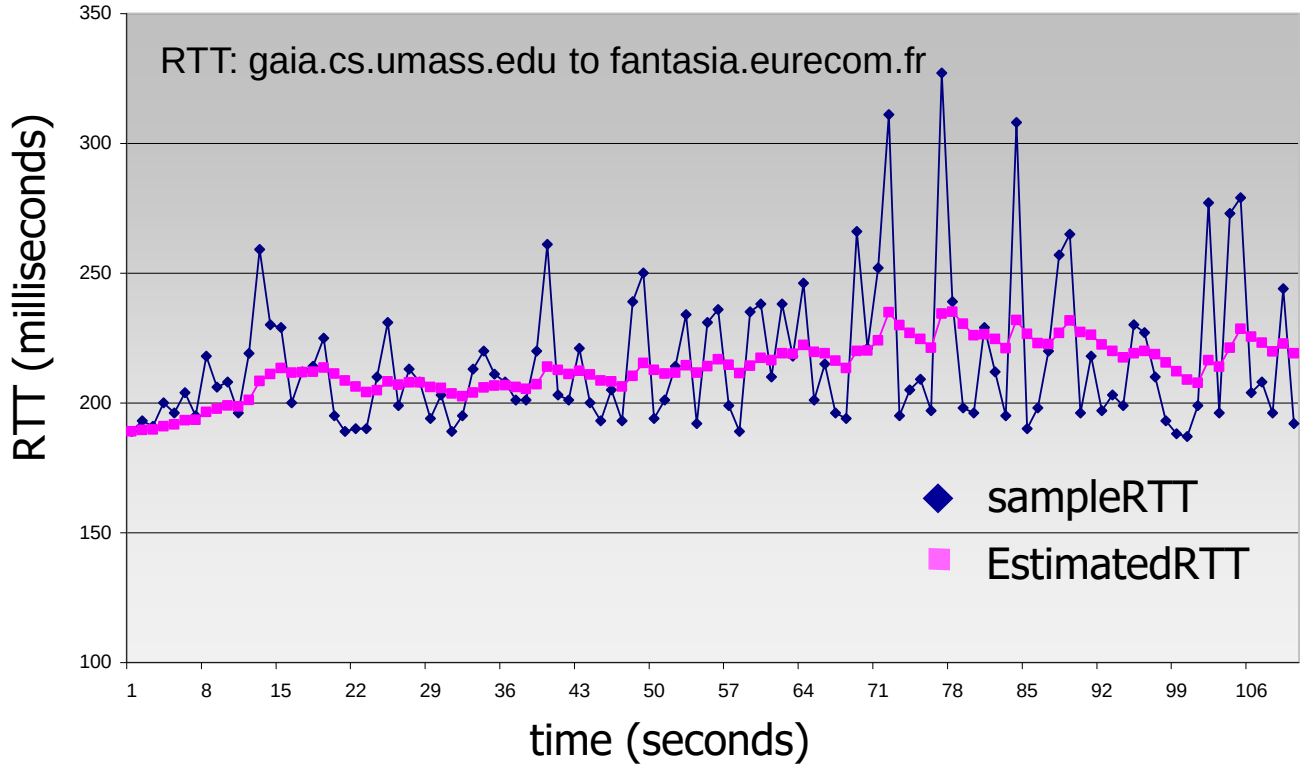
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



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-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 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

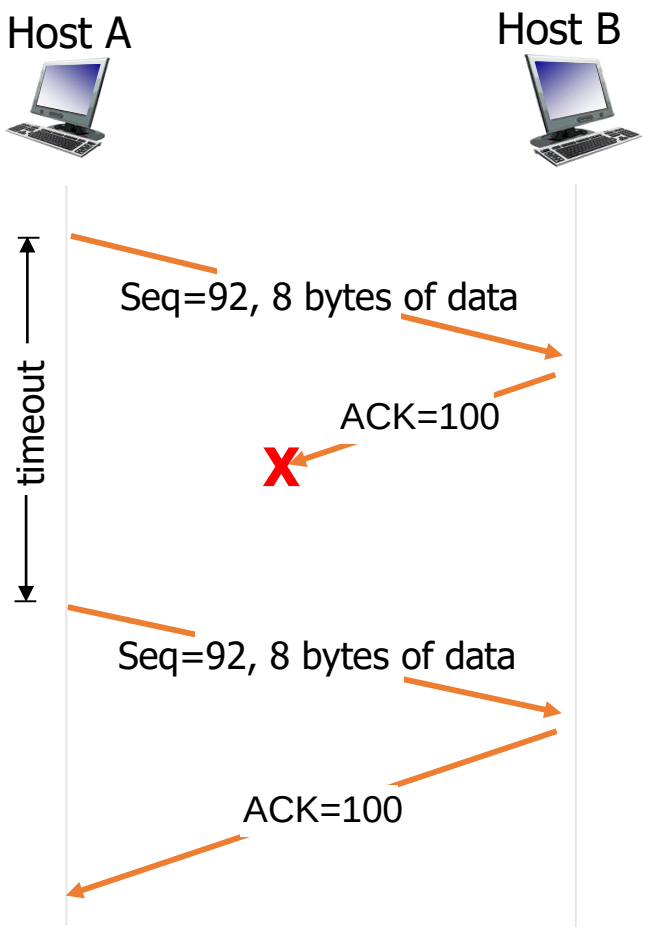
event: ACK received

- 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

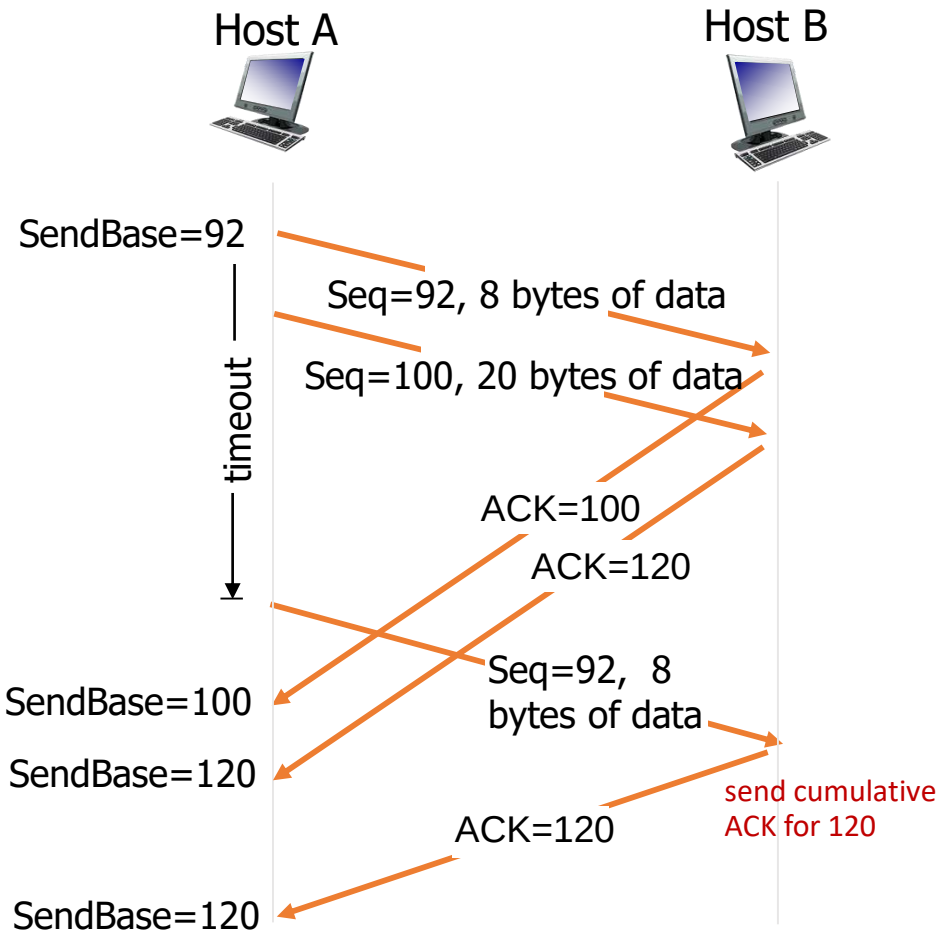


a. lost ACK scenario

- Host A sends a segment to Host B
  - segment has sequence number 92 and contains 8 bytes of data.
- After sending this segment, Host A waits for a segment from B with acknowledgment number 100.
- BUT, the acknowledgment from B to A gets lost.
  - … the timeout event occurs, and Host A retransmits the same segment.
- Host B will discard the bytes in the retransmitted segment.
  - it observes from the sequence number that the segment contains data that has already been received.

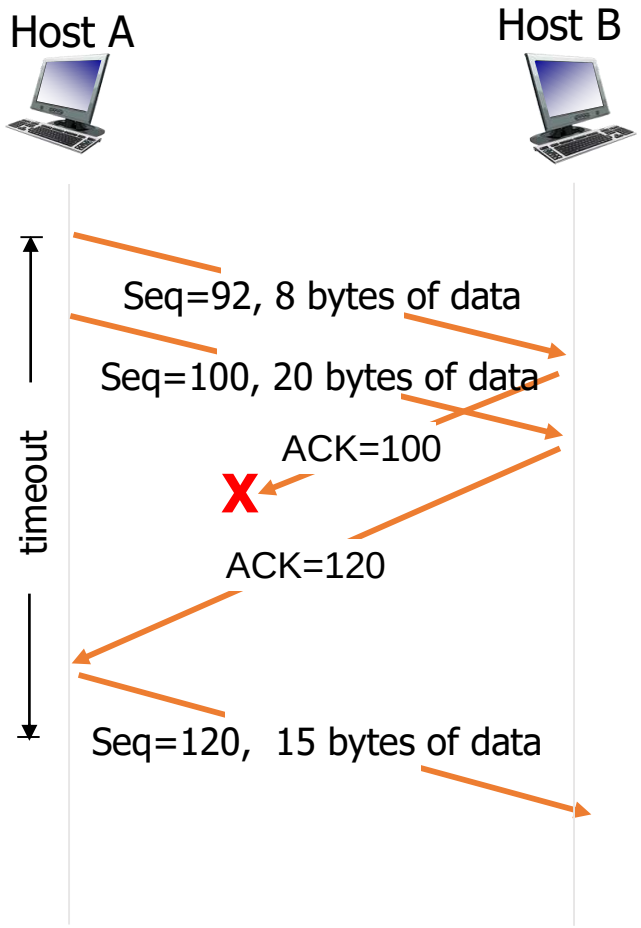
TCP: retransmission scenarios

- Host A sends two segments back to back.
  - The first segment has sequence number 92 and 8 bytes of data, and the second segment has sequence number 100 and 20 bytes of data.
- Suppose that both segments arrive intact at B, and B sends two separate acknowledgments for each of these segments.
  - The first of these acknowledgments has acknowledgment number 100; the second has acknowledgment number 120.
- Suppose now that neither of the acknowledgments arrives at Host A before the timeout.
- When the timeout event occurs, Host A resends the first segment with sequence number 92 and restarts the timer.
  - As long as the ACK for the second segment arrives before the new timeout, the second segment will not be retransmitted.



b. premature timeout

TCP: retransmission scenarios



c. cumulative ACK covers for earlier lost ACK

- suppose Host A sends the two segments, exactly as in the second example.
- The acknowledgment of the first segment is lost in the network, but just before the timeout event,
- Host A receives an acknowledgment with acknowledgment number 120.
- Host A therefore knows that Host B has received *everything* up through byte 119; so Host A does not resend both of the two segments.

TCP fast retransmit

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

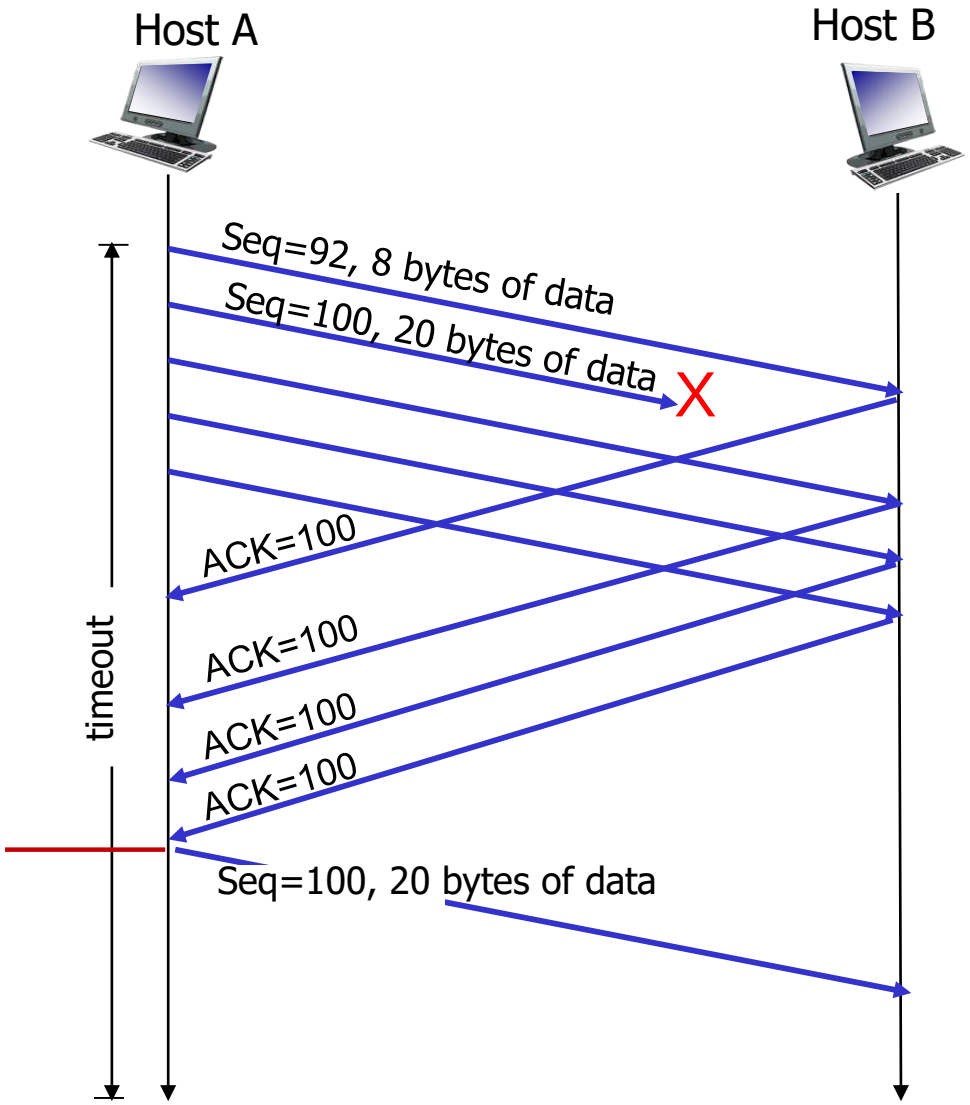
TCP fast retransmit

TCP fast retransmit

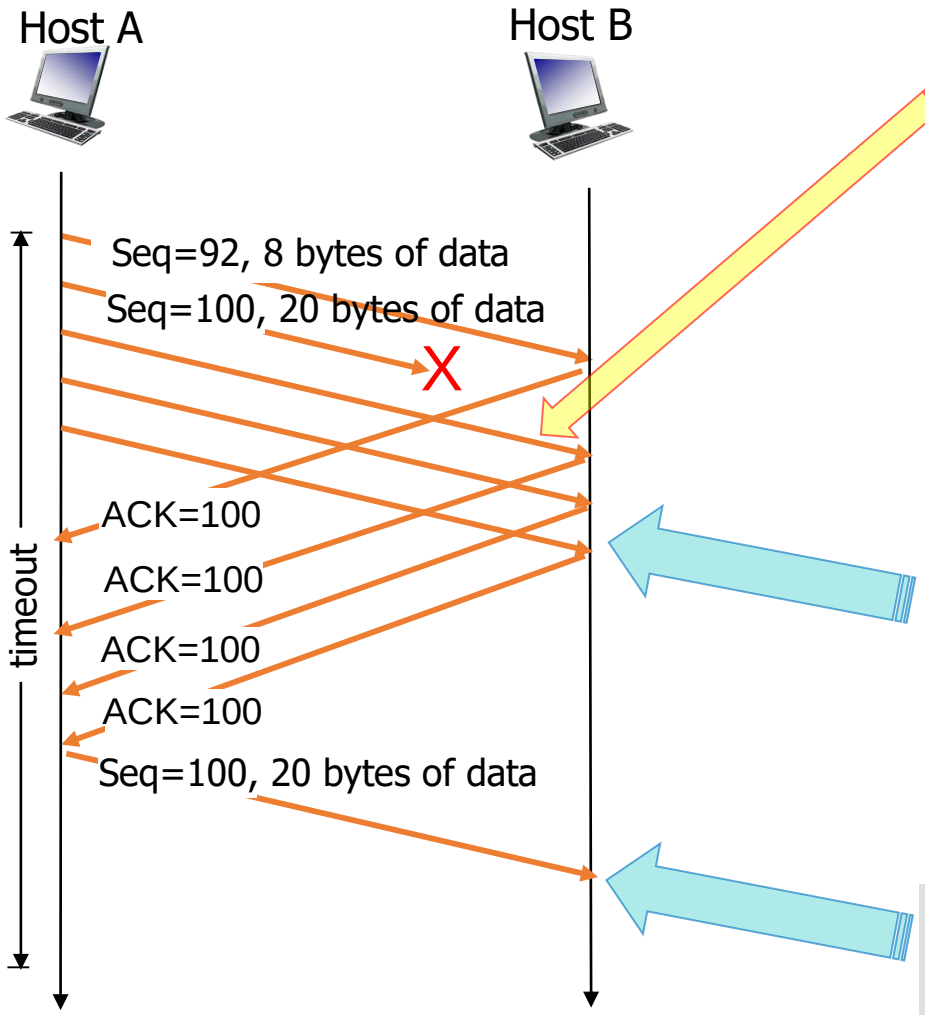
if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- 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!



TCP fast retransmit



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

Event @receiver:

- Arrival of out-of-order segment
- higher-than-expected `seq#`.
- Gap is detected

Action:

- Immediately send *triple duplicate* `ACK`,
- Indicating `seq#` of next expected byte

Sender:

- detect triple duplicate ACKs
- without waiting for timeout, resend the lost segment

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

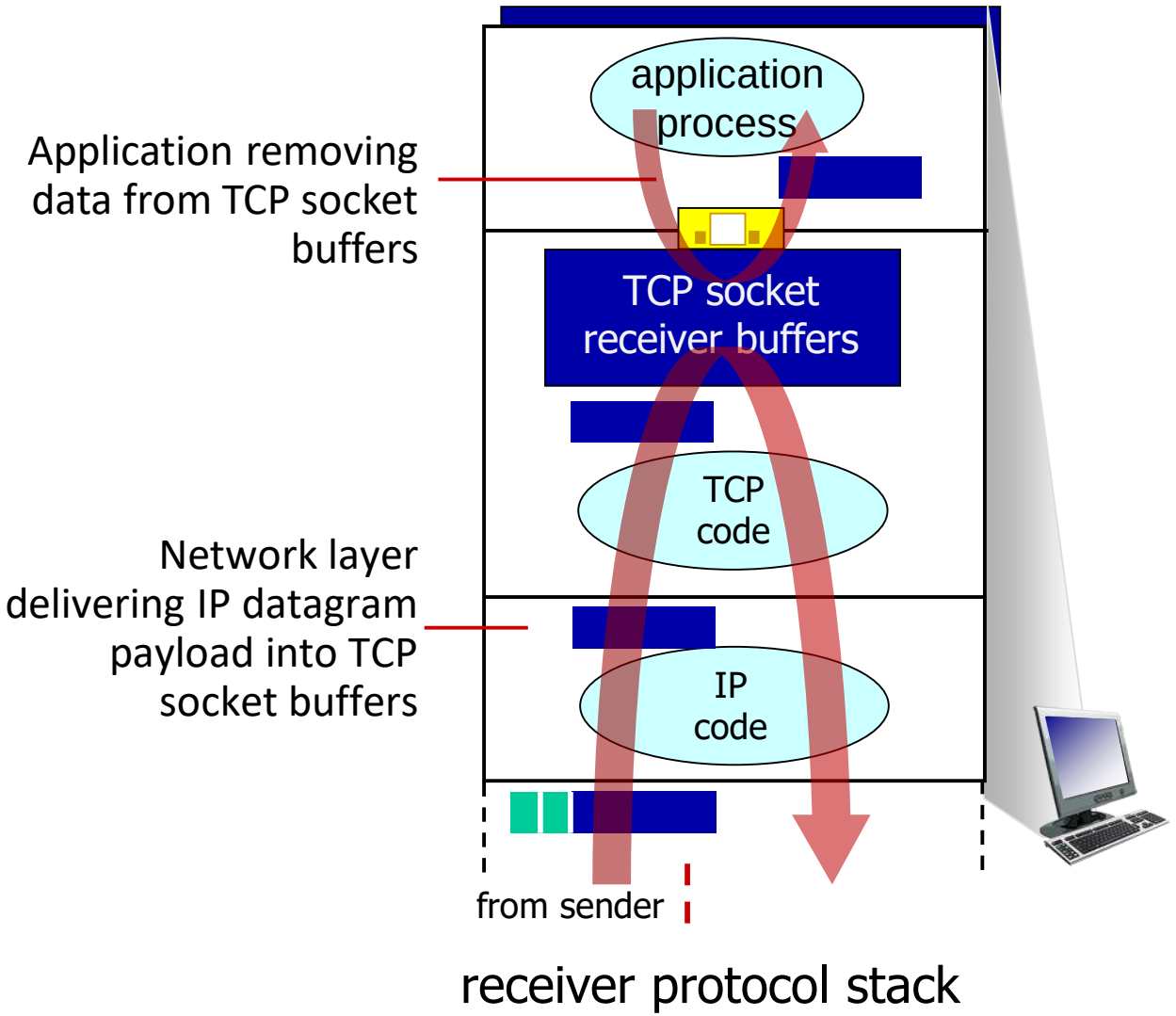
- When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer.
- The associated *application* process *will read data* from this buffer, but NOT necessarily at the instant the data arrives.
- Indeed, the *receiving application may be busy with some other task* and may not even attempt to read the data until long after it has arrived.

TCP flow control

- If the application is relatively slow at reading the data,
  - the sender can very easily *overflow the receiver buffer* by sending too much data too quickly.
- TCP provides a **flow-control service**
  - to *eliminate* the possibility of the *sender overflowing* the *receiver's buffer*.
- Flow control is thus *a speed-matching service*
  - matching the rate at which the sender is sending against the rate at which the receiving application is reading.

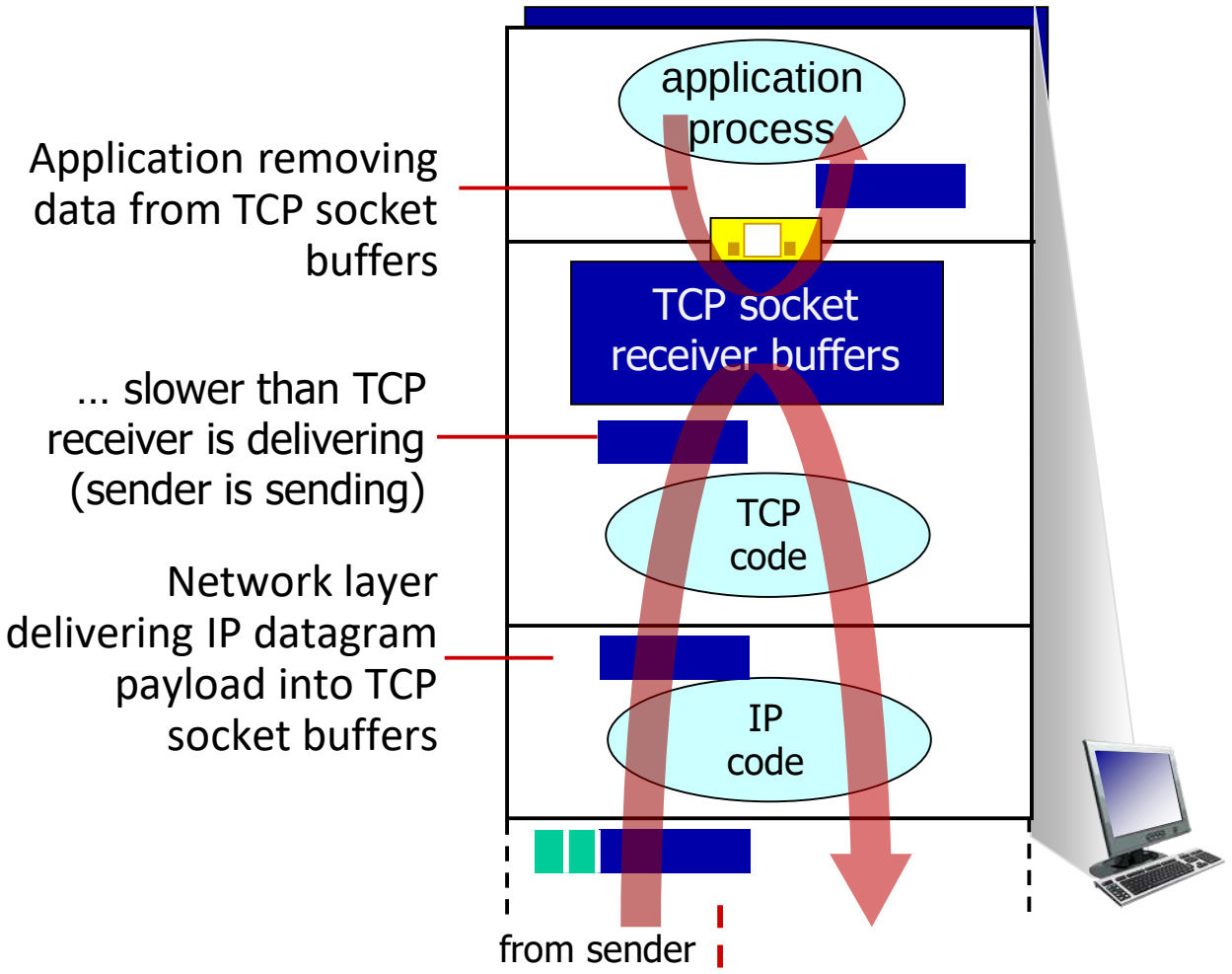
TCP flow control

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



TCP flow control

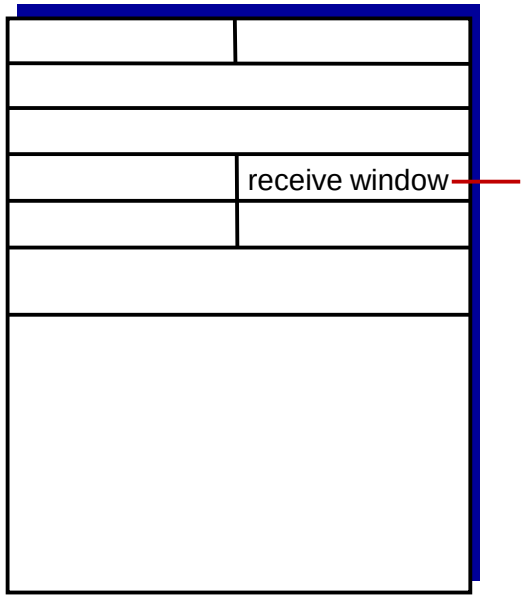
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

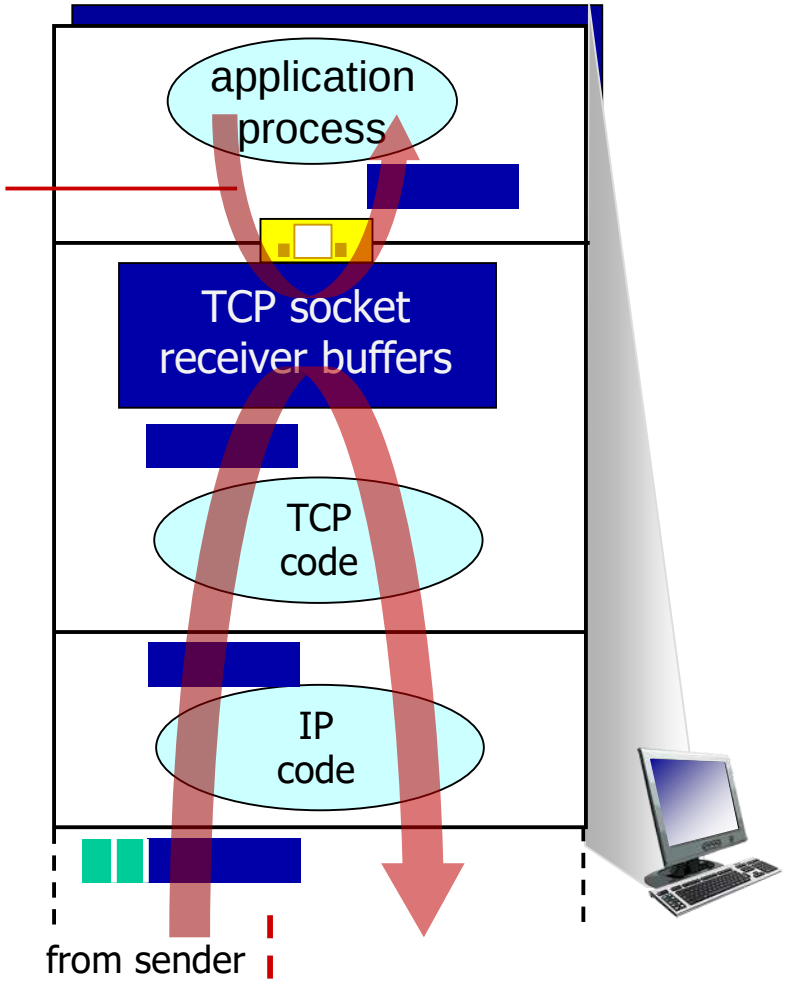
TCP flow control

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



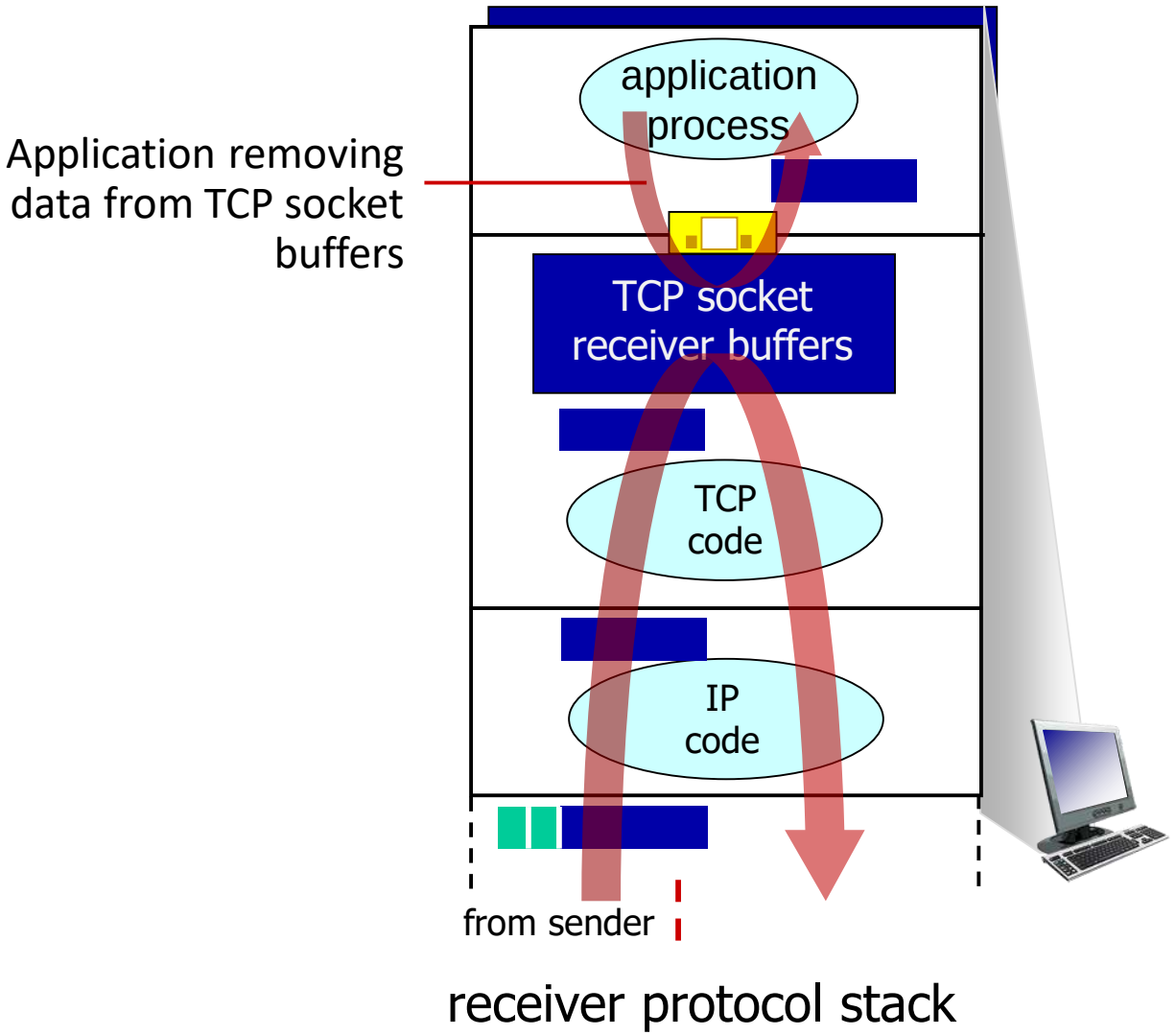
receiver protocol stack

TCP flow control

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

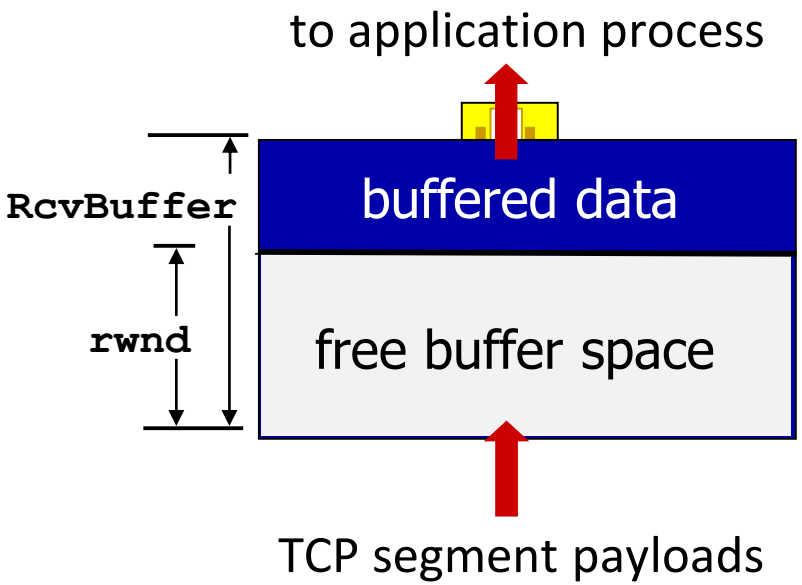
flow control

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



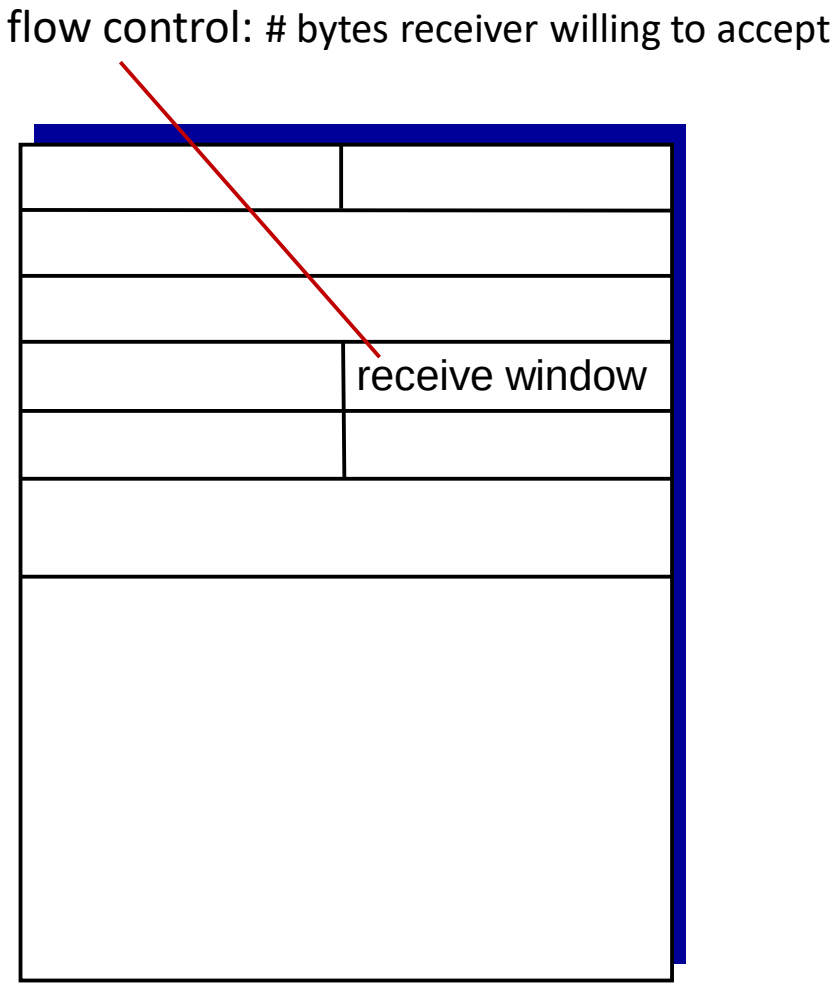
TCP receiver-side buffering

`RcvBuffer` → received buffer data

`rwnd` → received window size

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

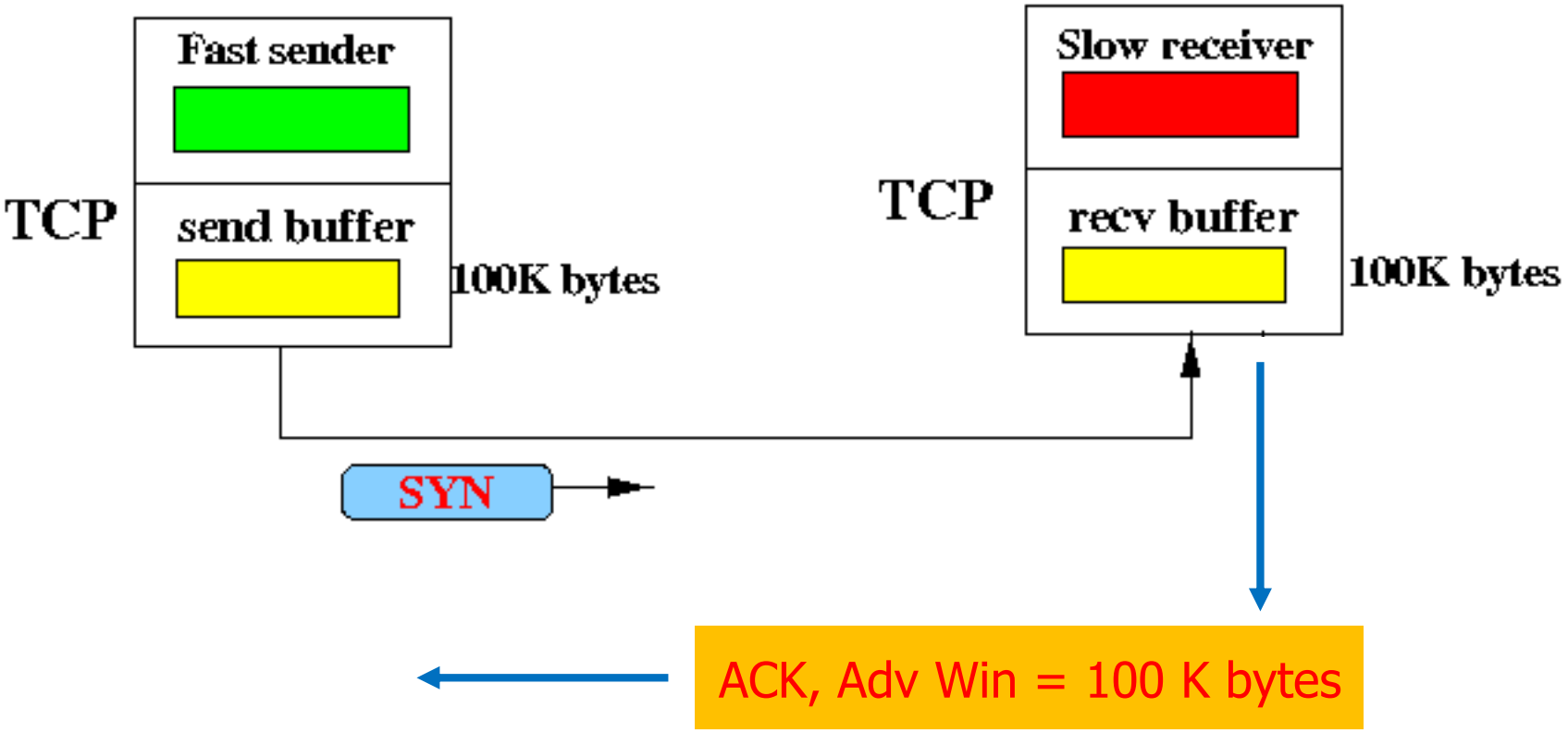


TCP segment format

Example: TCP flow control

1

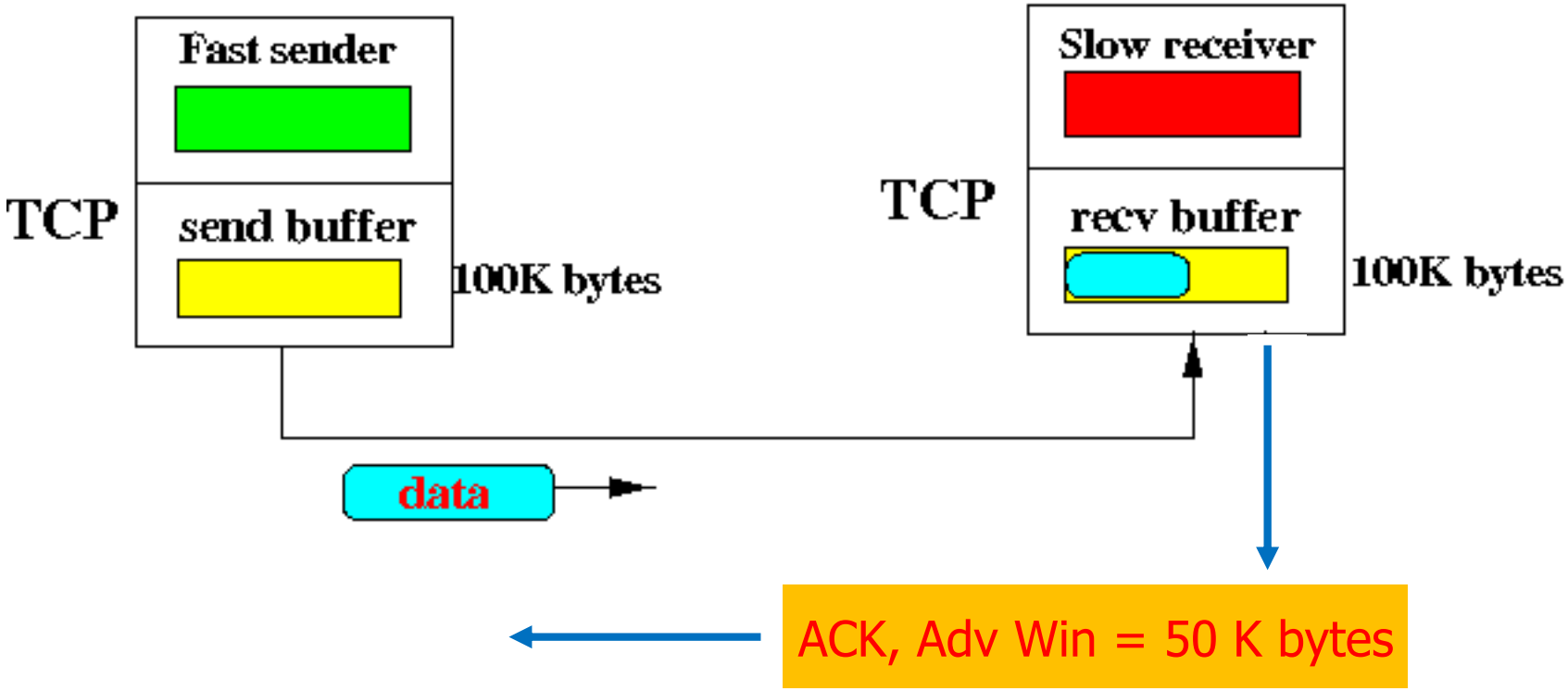
- Initially, the send and receive buffer are empty.
- Receiver advertises a window of **100 K bytes**.



Example: TCP flow control

2

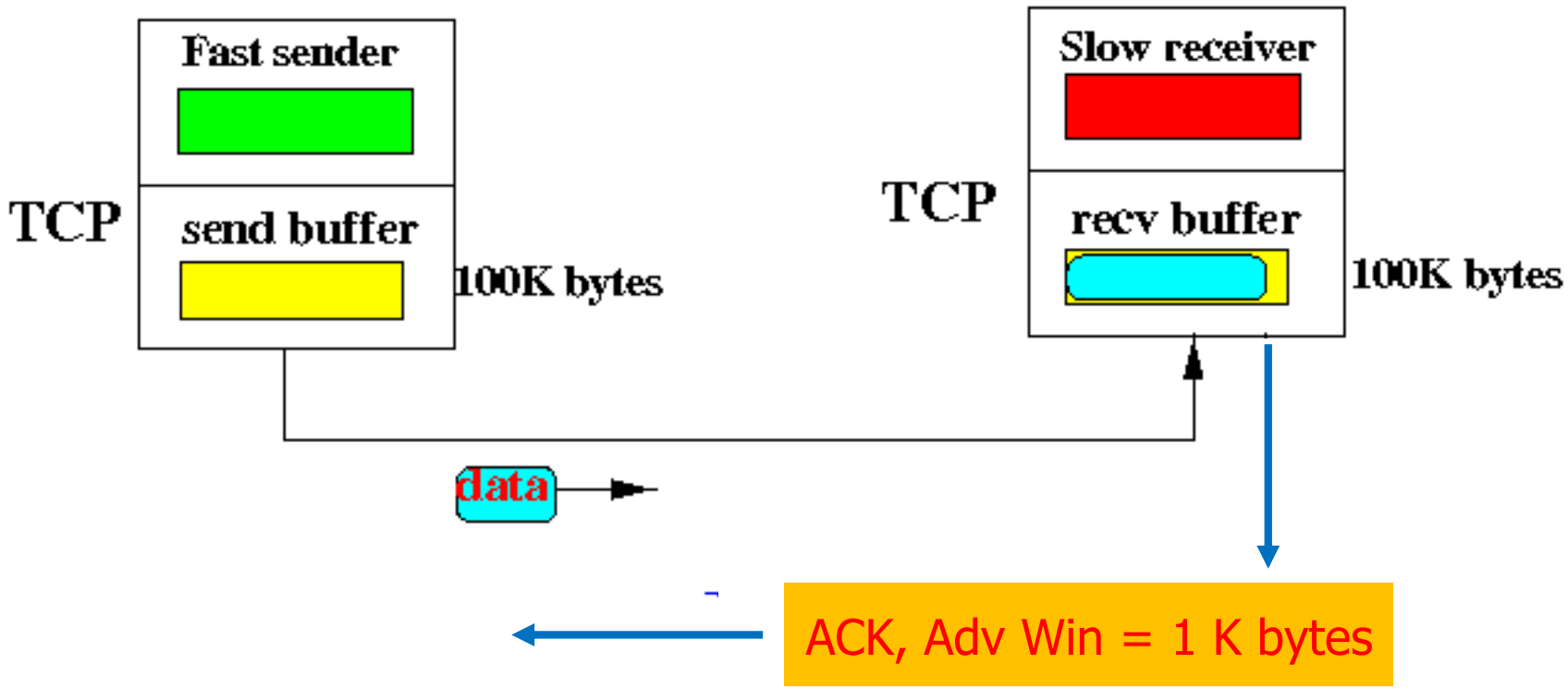
- Sender sends many packets quickly, faster than the receiver can process...
- the receive buffer starts to fill and soon the advertised window drops to 50 K bytes:



Example: TCP flow control

3

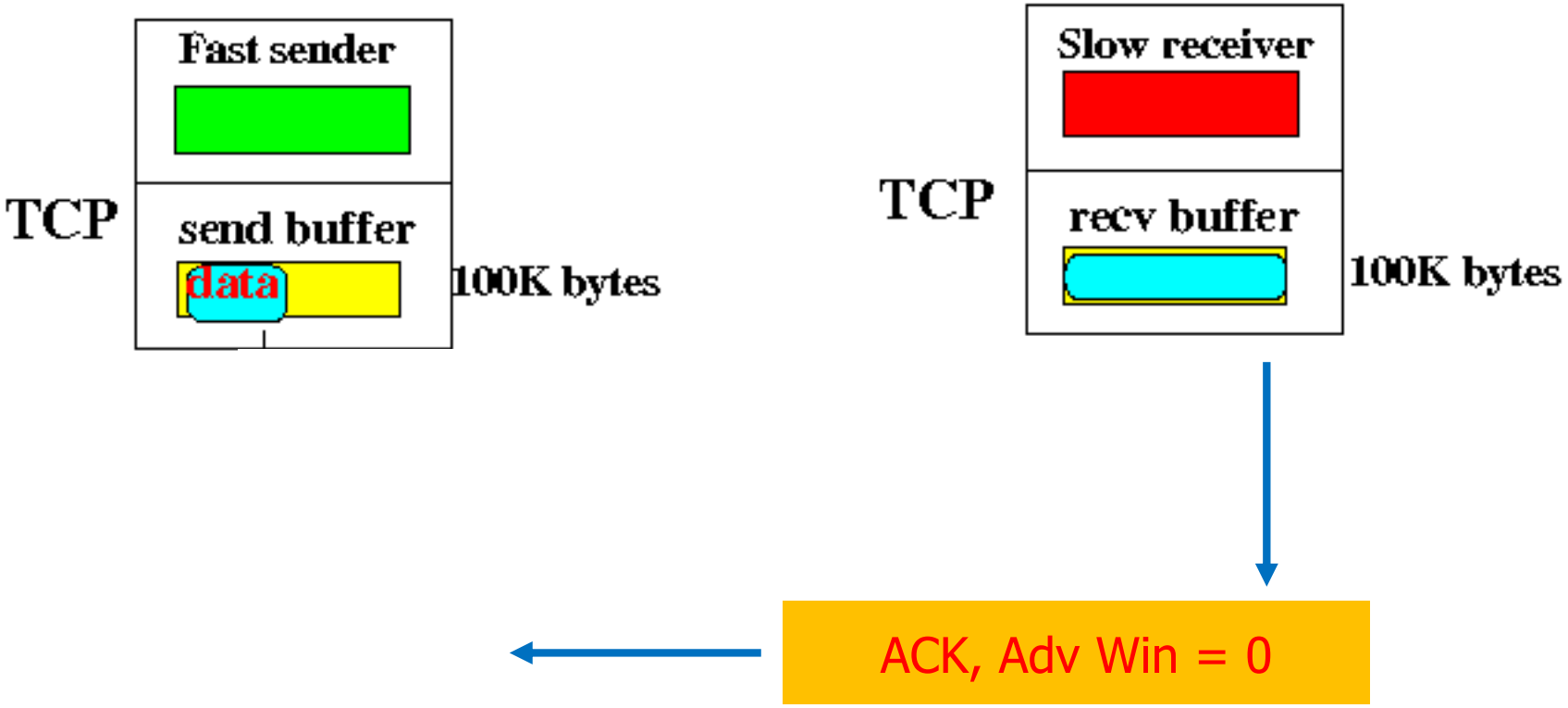
- The situation continues and the receive buffer fills further
- The ACK packets from the receiver will now have a lower advertised window, e.g., 1 K bytes:



Example: TCP flow control

4

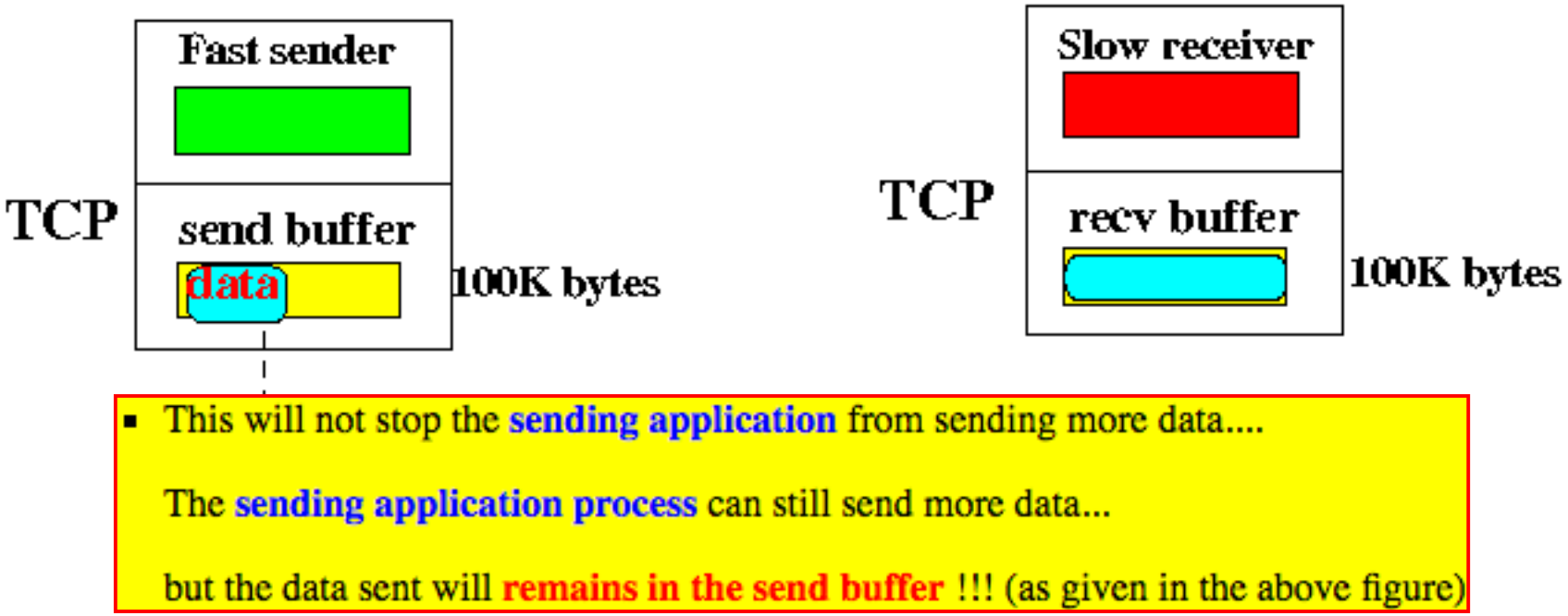
- Sooner or later, the receive buffer fills up
- The receiver returns an ACK packets with the advertised window = 0 bytes !!!



Example: TCP flow control

4

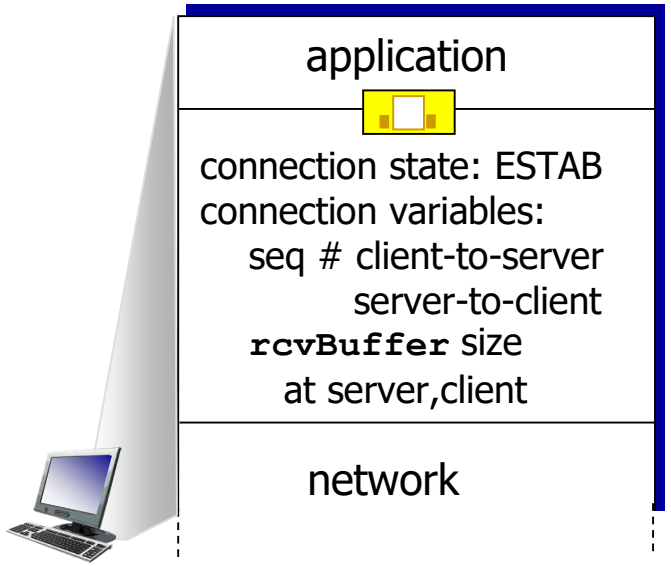
- … cont.; This causes the sending TCP to stop transmitting more data.....
- and prevent the sending TCP to overflow the receive buffer of the receiving TCP.

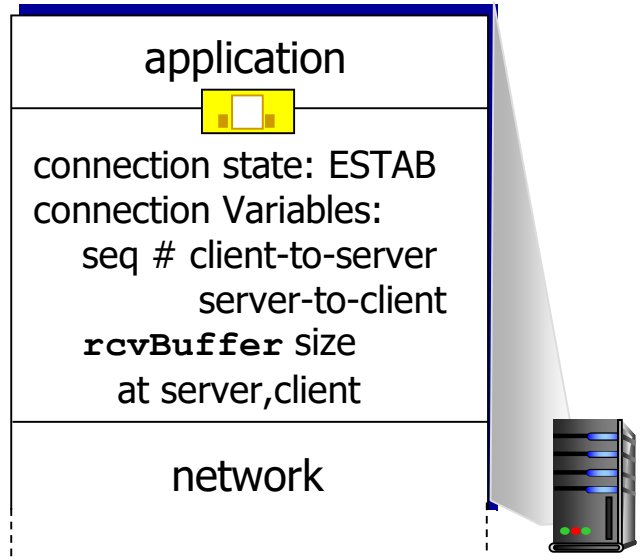


TCP connection management

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)



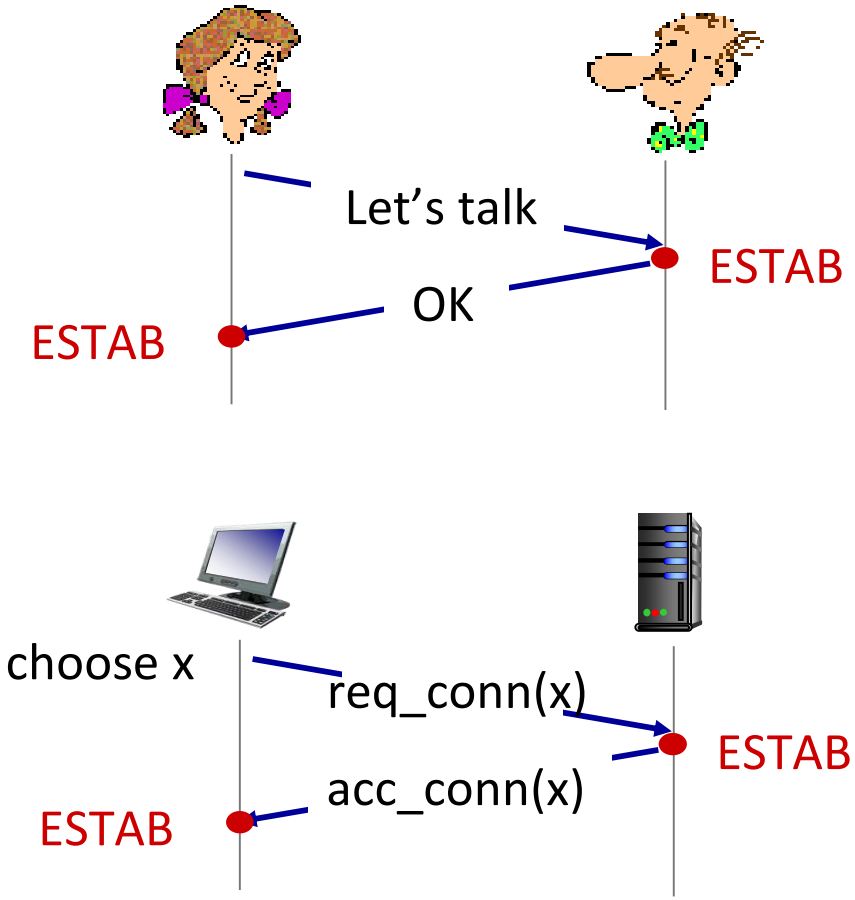



```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

Agreeing to establish a connection

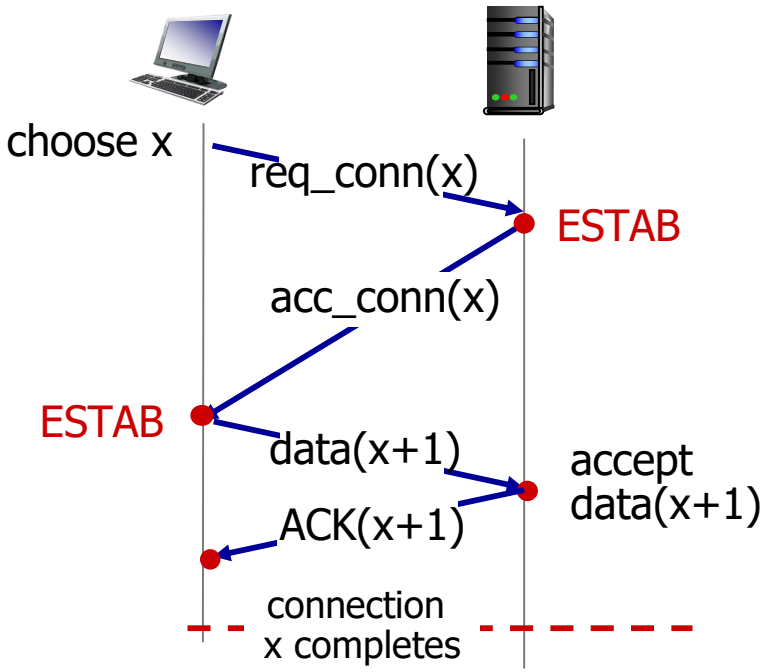
2-way handshake:



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

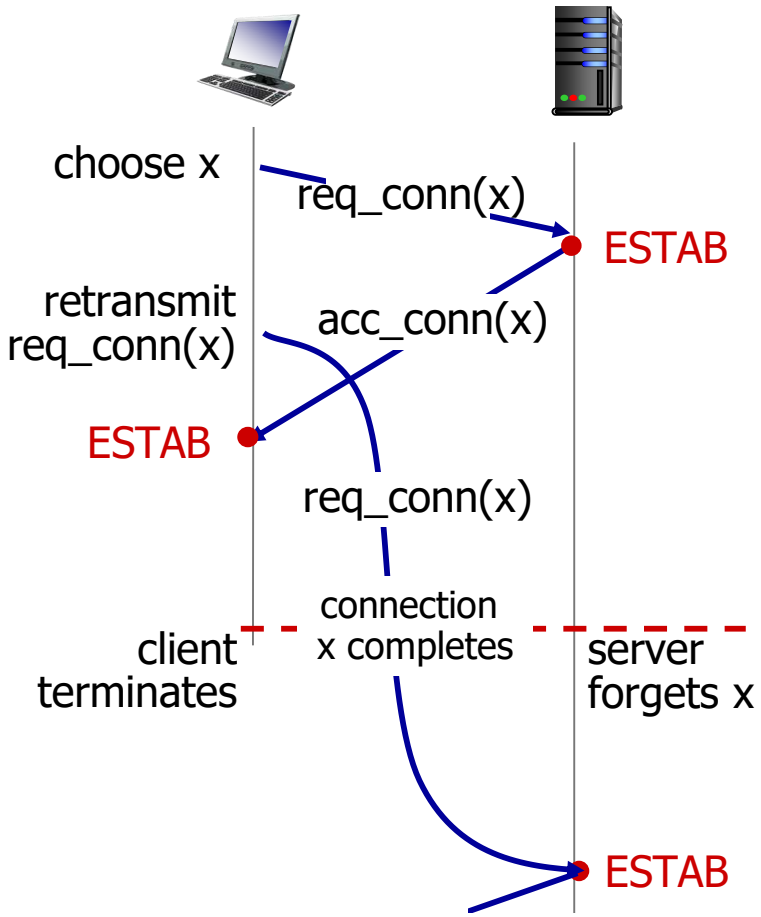
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

2-way handshake scenarios



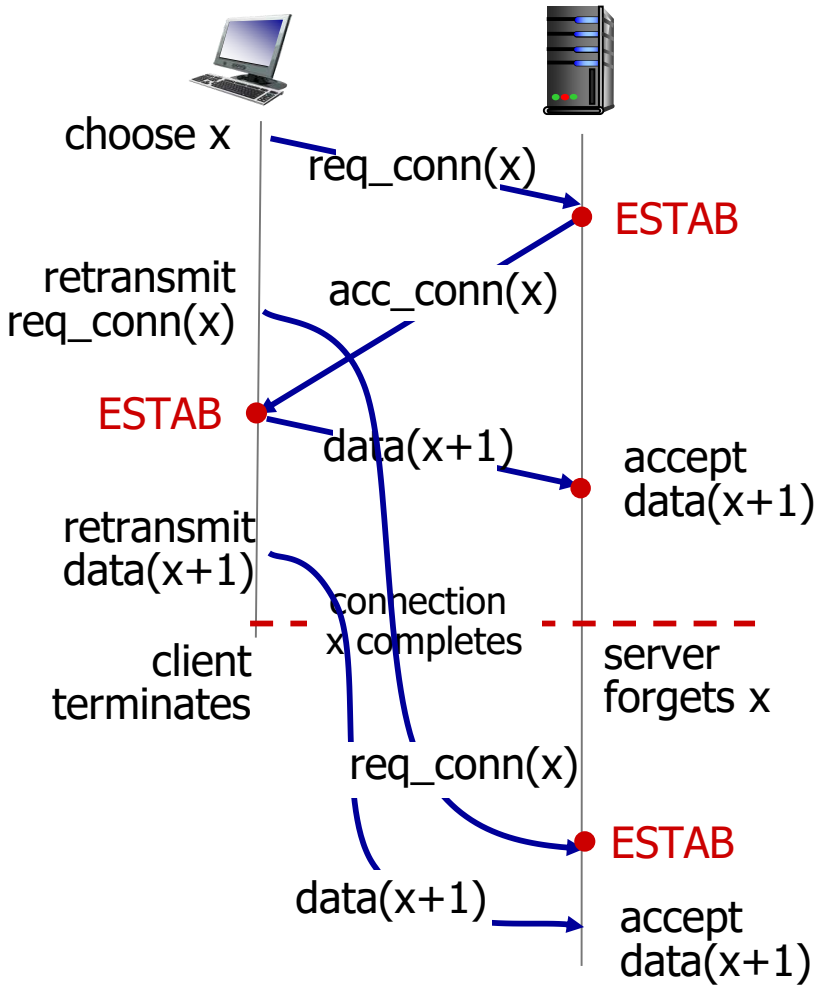
No problem!

2-way handshake scenarios



Problem: half open connection! (no client)

2-way handshake scenarios

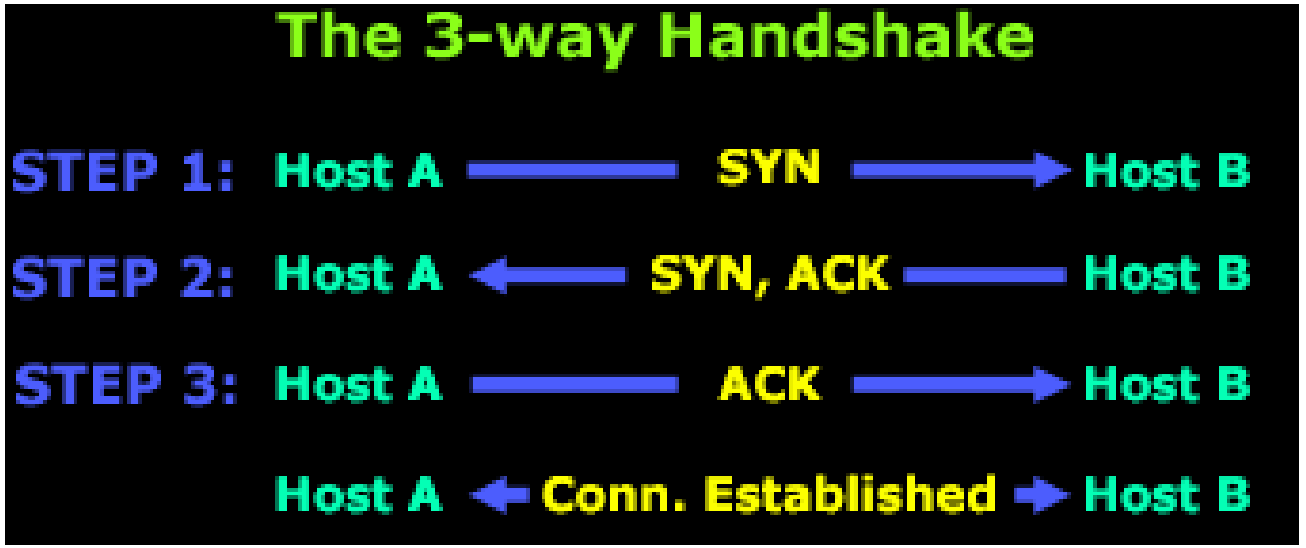


Problem: dup data accepted!

TCP 3-way handshake

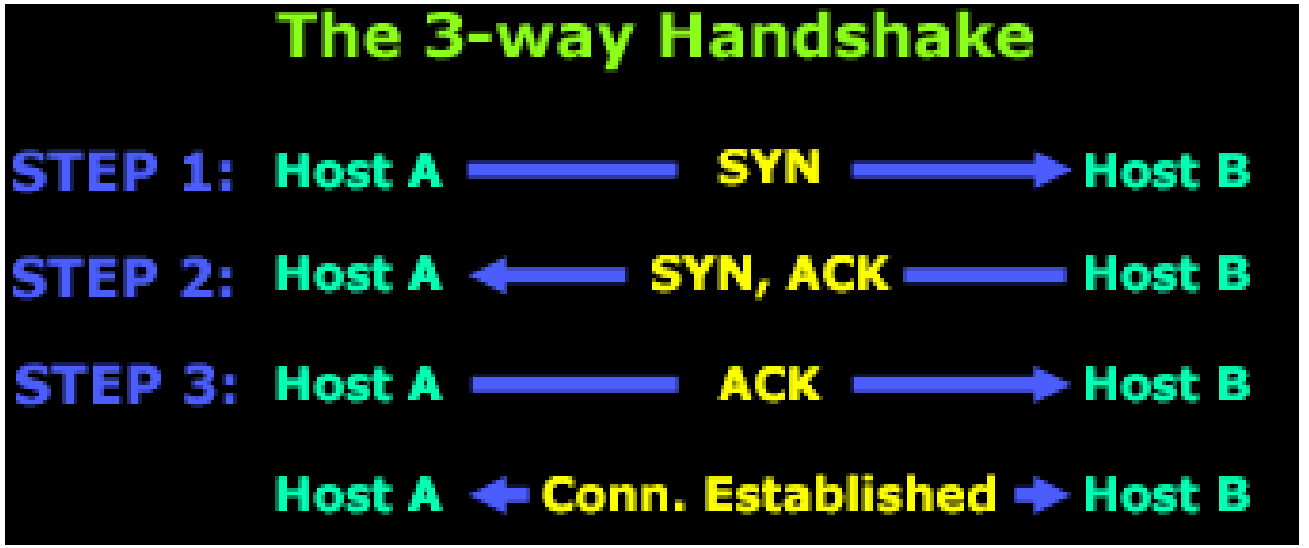
- Four actions are required before sending data
  - Host A sends a segment to announce its wish for connection and includes its initialization information
  - Host B sends a segment to acknowledge the request of A
  - Host B sends a segment that includes initialization information
  - Host A sends a segment to acknowledge the request from B
- 2nd and 3rd can be combined
- Called *Three-Way Handshaking*

TCP 3-way handshake



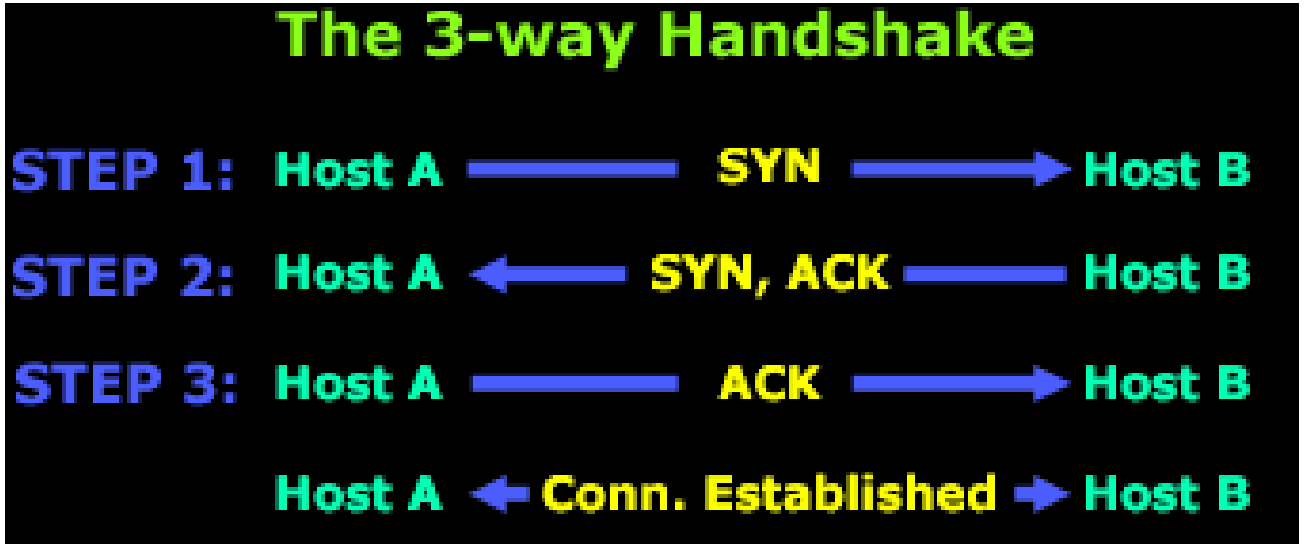
- **STEP 1:**
  - Host A sends the initial packet to Host B. This packet has the "SYN" bit enabled.
  - Host B receives the packet and sees the "SYN" bit enabled, so it knows that Host A is trying to establish a connection with it.

TCP 3-way handshake



- **STEP 2:**
  - Assuming Host B has enough resources, it sends a packet back to Host A and with the "SYN and ACK" bits enabled (set to 1).
  - The SYN that Host B sends, at this step, means '*I want to synchronize with you*' and the ACK means '*I acknowledge your previous SYN request*'.

TCP 3-way handshake



- **STEP 3:**
  - Host A sends another packet to Host B and with the "ACK" bit set (1),
  - … it effectively tells Host B '*Yes, I acknowledge your previous request*'.

TCP 3-way handshake

Client state

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

LISTEN

```
clientSocket.connect((serverName,serverPort))
```

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

ESTAB

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

Server state

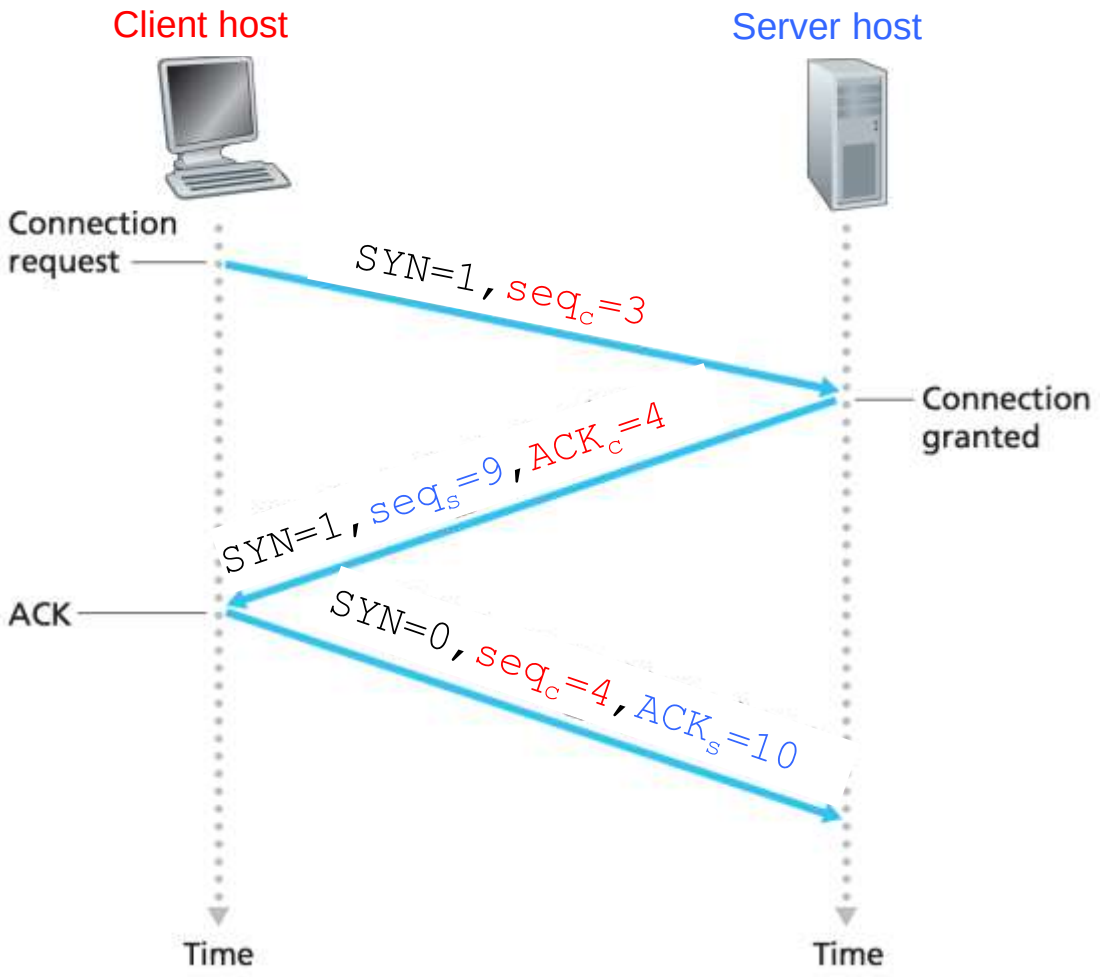
```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

ESTAB

EXAMPLE 1 : TCP 3-way handshake



TCP 3-way handshake: segment exchange

TCP State:

SYN_SENT

ESTABLISHED

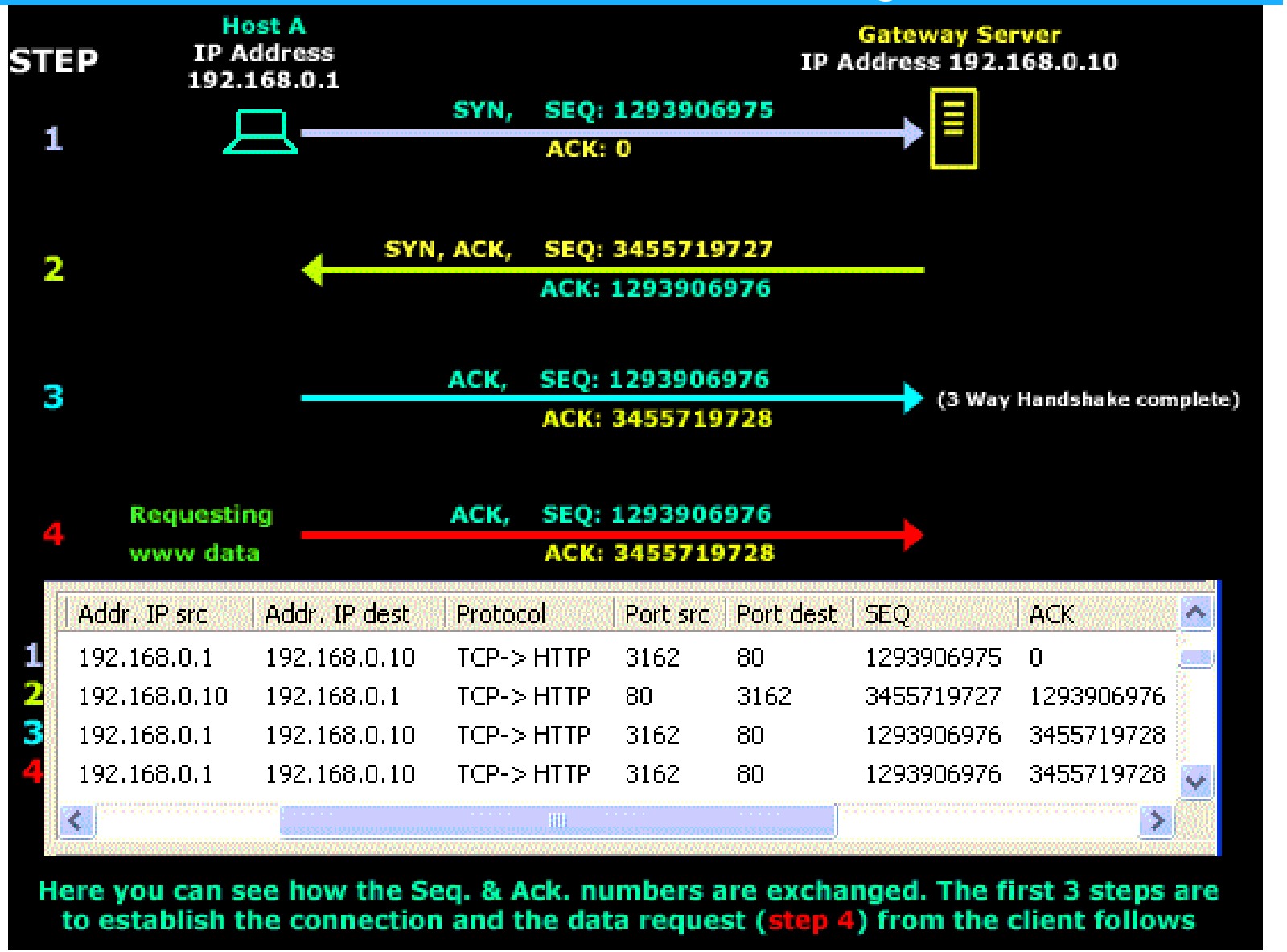
TCP State:

LISTEN

SYN_RCVD

ESTABLISHED

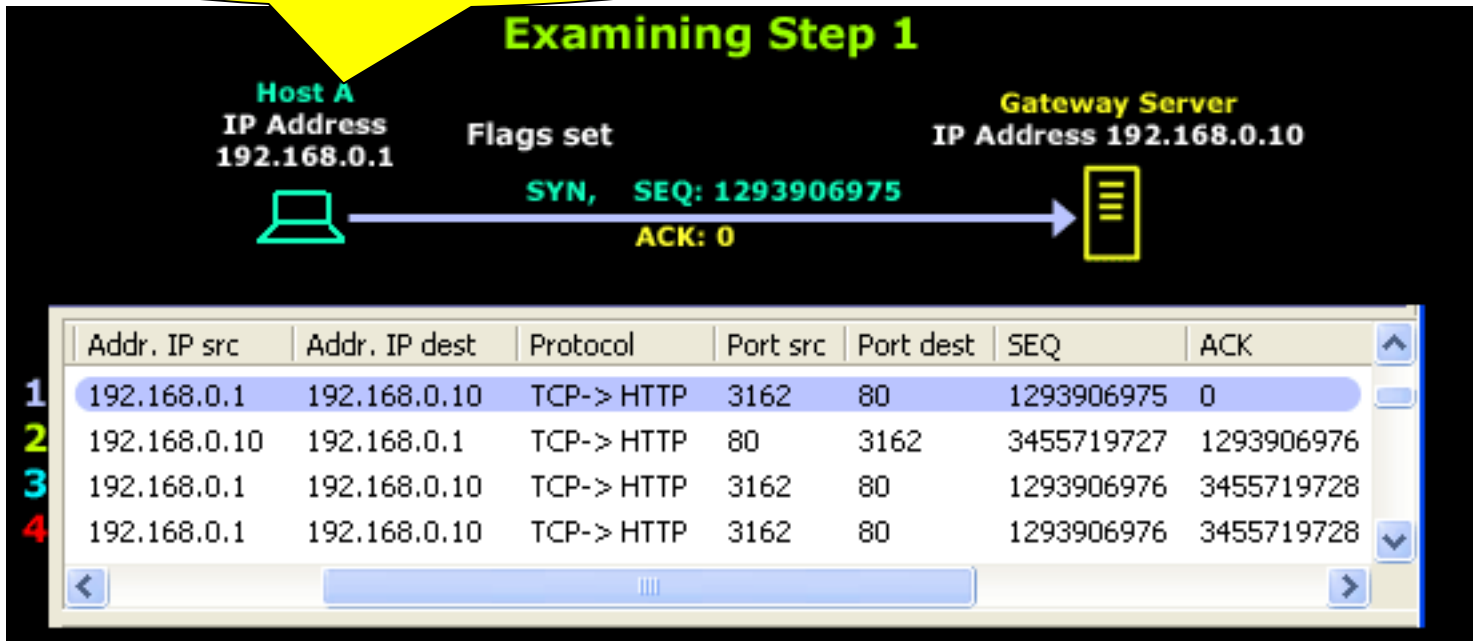
EXAMPLE 2 : Examining Sequence and Acknowledgement Numbers



"I'd like to initiate a new connection with you.
My **Sequence** number is **1293906975**".

EXAMPLE 2

STEP 1

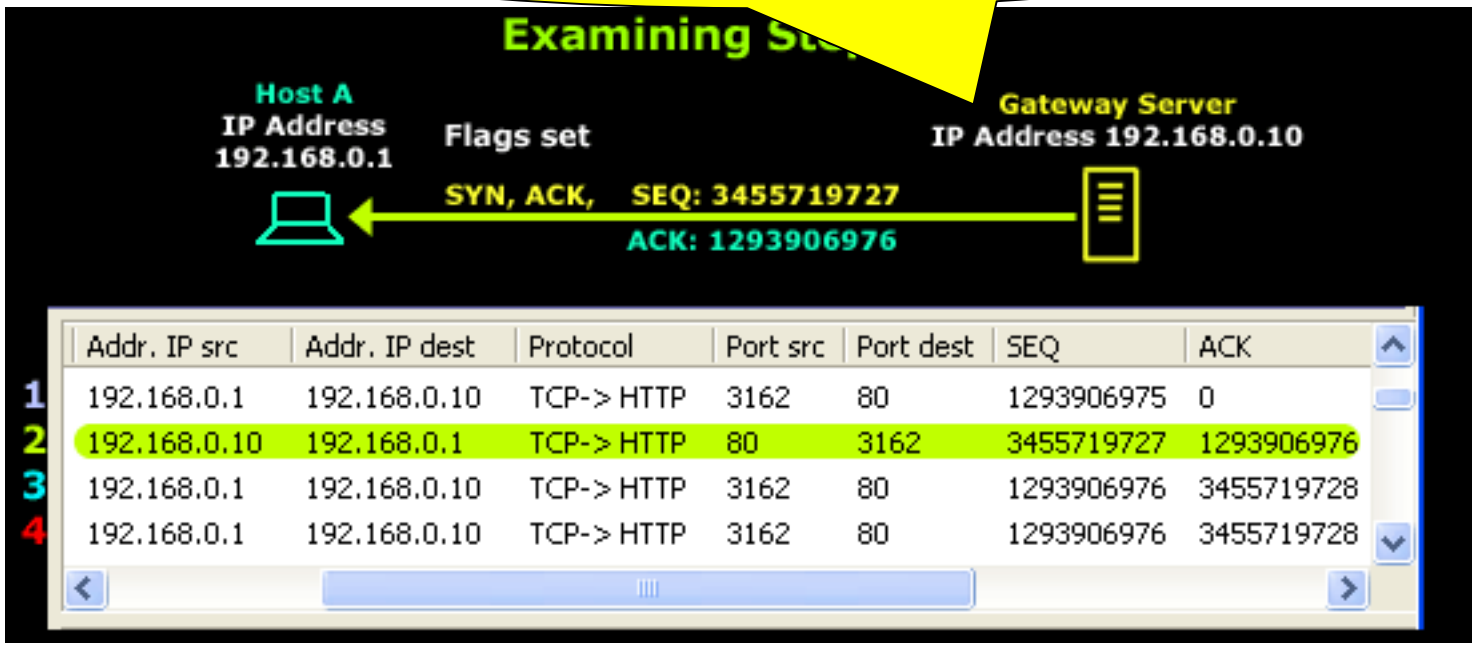


- **Host A** wishes to download a webpage from the **Gateway Server**.
- To establish a new connection, **Host A** sends a packet to the **Gateway Server**;
  - This packet has the **SYN** flag set (i.e. SYN bit = 1) and
  - also contains the ISN generated by **Host A's** operating system, that is **1293906975**.
  - the **Acknowledgment** number is set to zero (0),
    - … since **Host A** is initiating the connection and hasn't received a reply from the **Gateway Server**

EXAMPLE 2

"I acknowledge your sequence number and expecting your next packet with sequence number **1293906976**. My sequence number is **3455719727**".

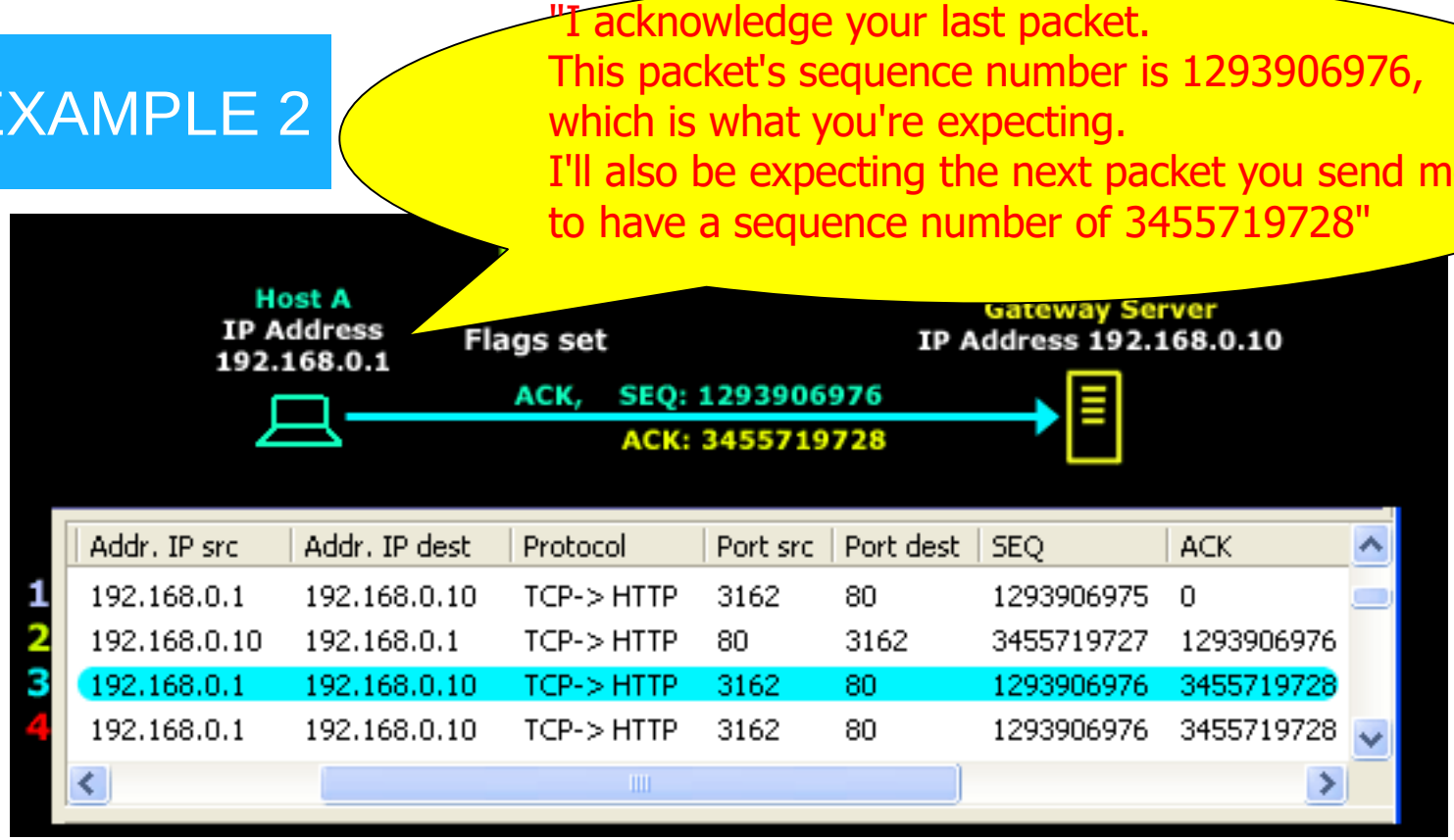
STEP 2



- The **Gateway Server** receives **Host A's** request and generates a reply containing
  - its own generated ISN, that is **3455719727**, and
  - the next **Sequence** number it is expecting from **Host A** which is **1293906976**.
  - also has the **SYN** & **ACK** flags set, acknowledging the previous packet it received and informing **Host A** of its own **Sequence** number.

EXAMPLE 2

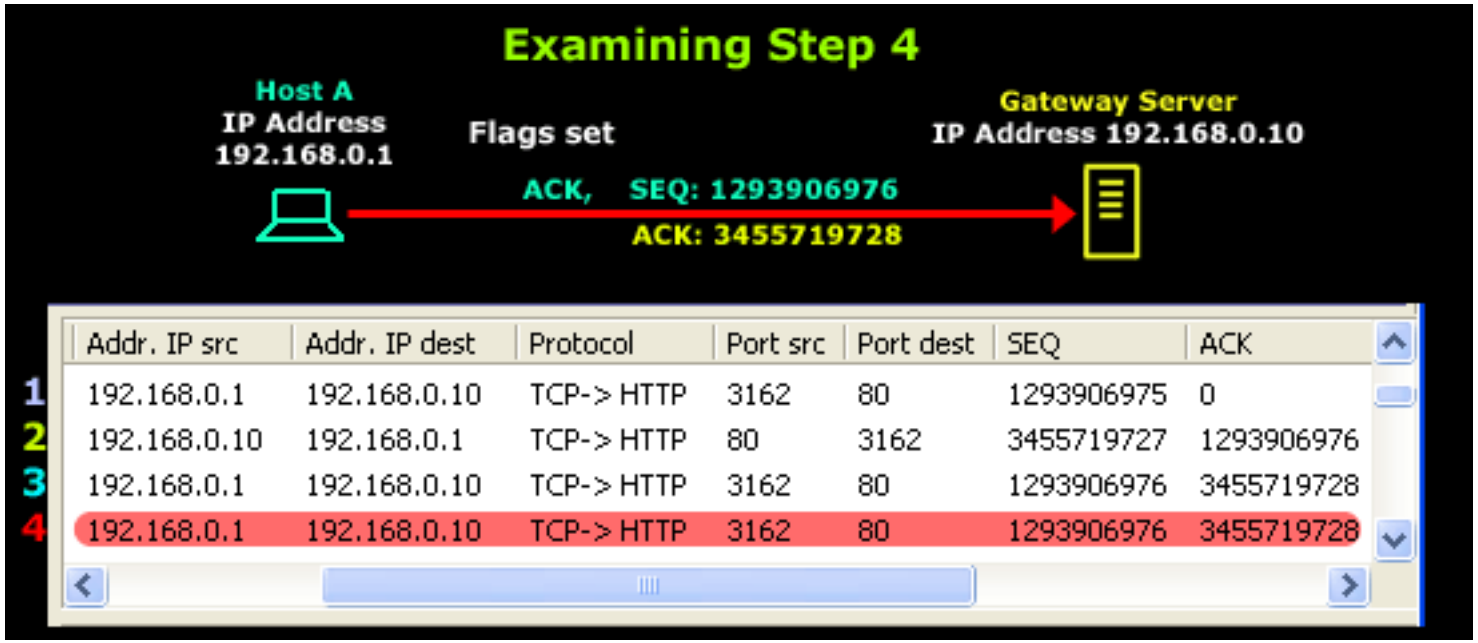
STEP 3



- **Host A** receives the reply and now knows Gateway's sequence number.
- It generates another packet to complete the connection.
  - This packet has the **ACK** flag set and also contains
  - the sequence number that it expects the Gateway Server to use next, that is **3455719728**.
- Since the 3-way TCP handshake has been completed, a virtual connection between the two now exists and the **Gateway Server** is ready to listen to **Host A's** request.

EXAMPLE 2

STEP 4



- In this step, Host A generates a packet with some data and sends it to the Gateway Server.
- The data tells the Gateway Server which webpage it would like sent.
- … remember in Module 2

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
```

A human 3-way handshake protocol

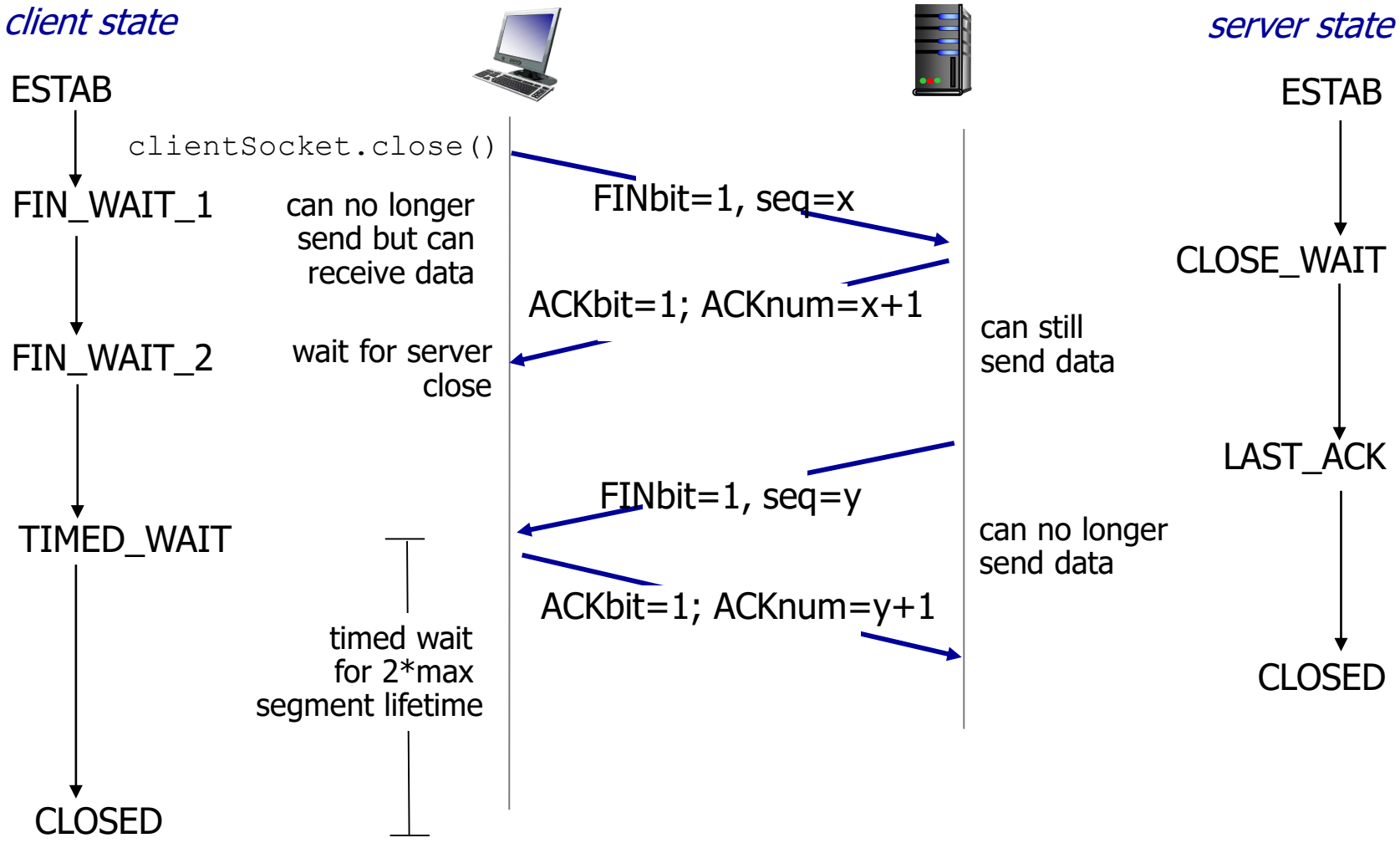
TCP: closing a connection

- Closing connection, four actions are required:
  - Host A sends a segment announcing connection termination
  - Host B sends a segment acknowledging the request from A. After this the connection is closed in one direction.
  - When host B has finished sending data, it sends a segment indicating connection closure
  - Host A acknowledge the request from B
- 2nd and 3rd cannot be combined
- It is called *Four-Way Handshaking*

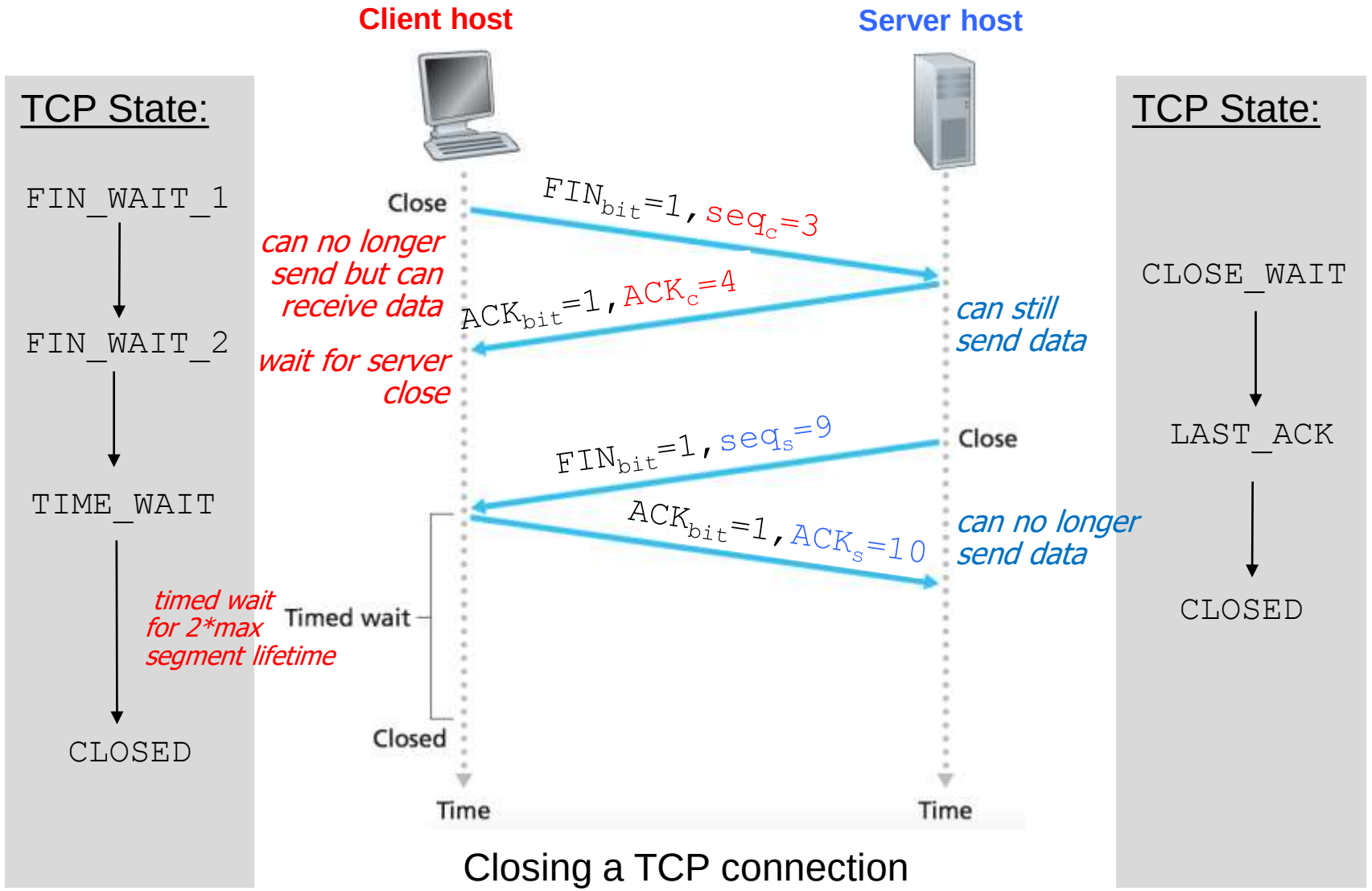
Closing a TCP connection

- 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: closing a connection



TCP: Example closing a connection



Closing a TCP connection

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!

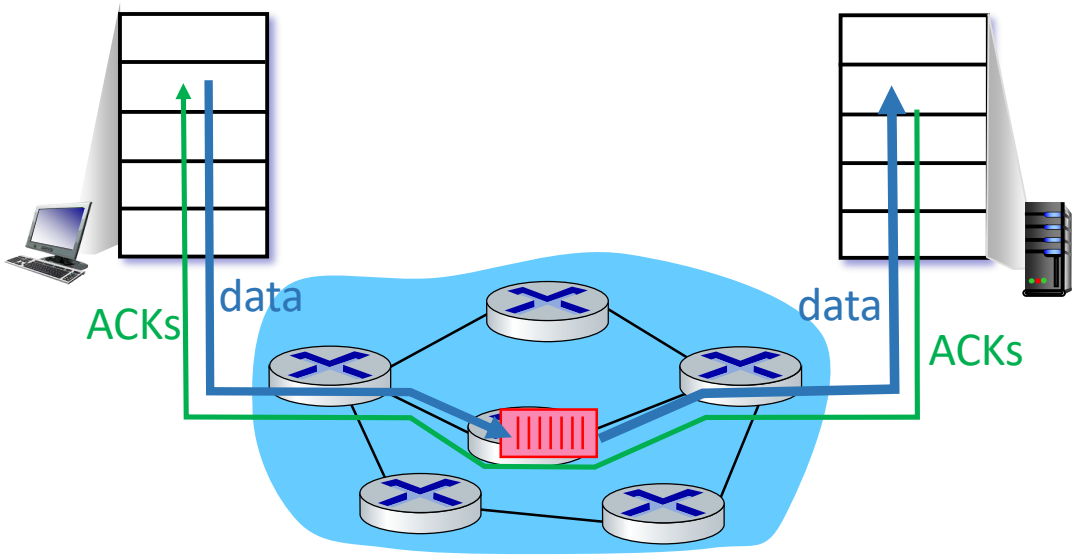
congestion control:
too many senders,
sending too fast

flow control: one sender
too fast for one receiver

Approaches towards congestion control

End-end congestion control:

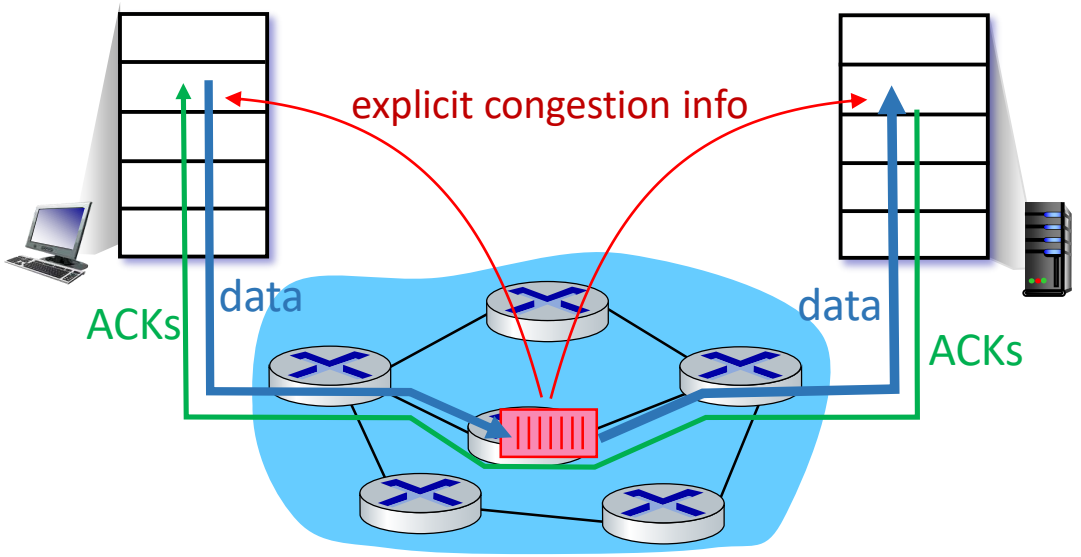
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



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, ATM, DECbit protocols



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

TCP congestion control: Additive Increase Multiplicative Decrease (AIMD)

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

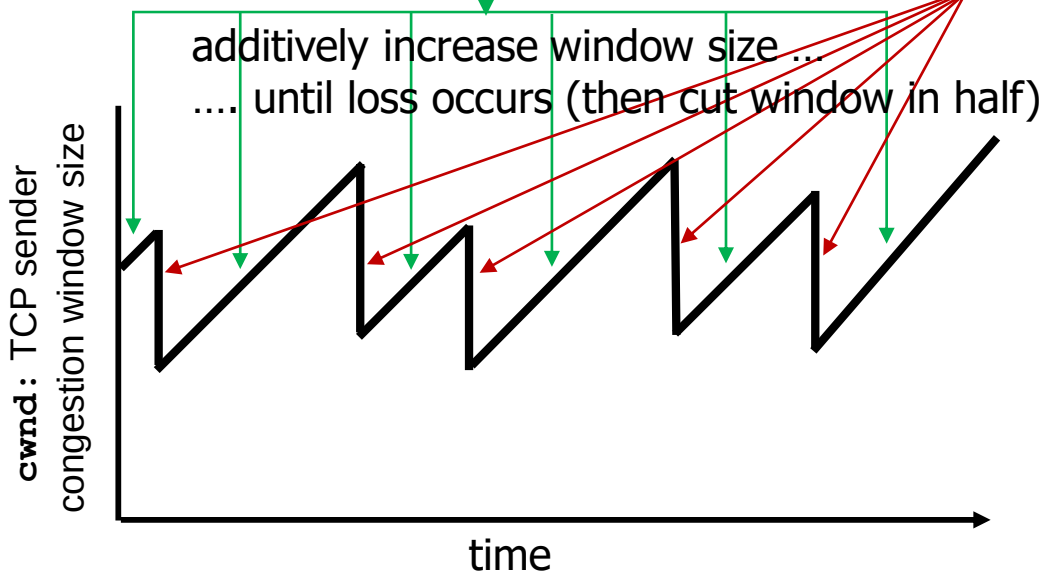
Additive Increase

increase cwnd sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut cwnd sending rate in half at each loss event

AIMD sawtooth behavior: *probing* for bandwidth



TCP AIMD: more

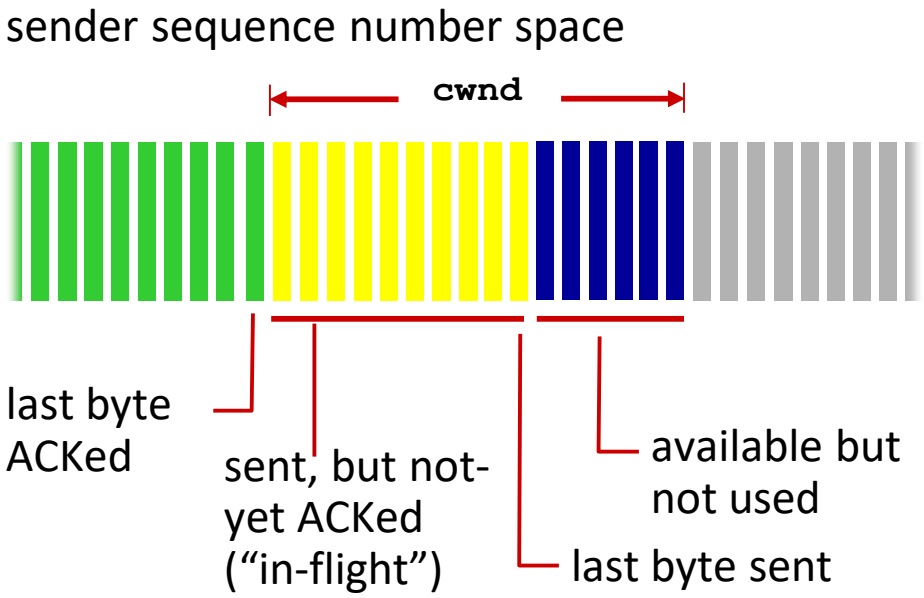
Multiplicative decrease detail: sending rate is

- 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)

Why AIMD?

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

TCP congestion control: details



TCP sending behavior:

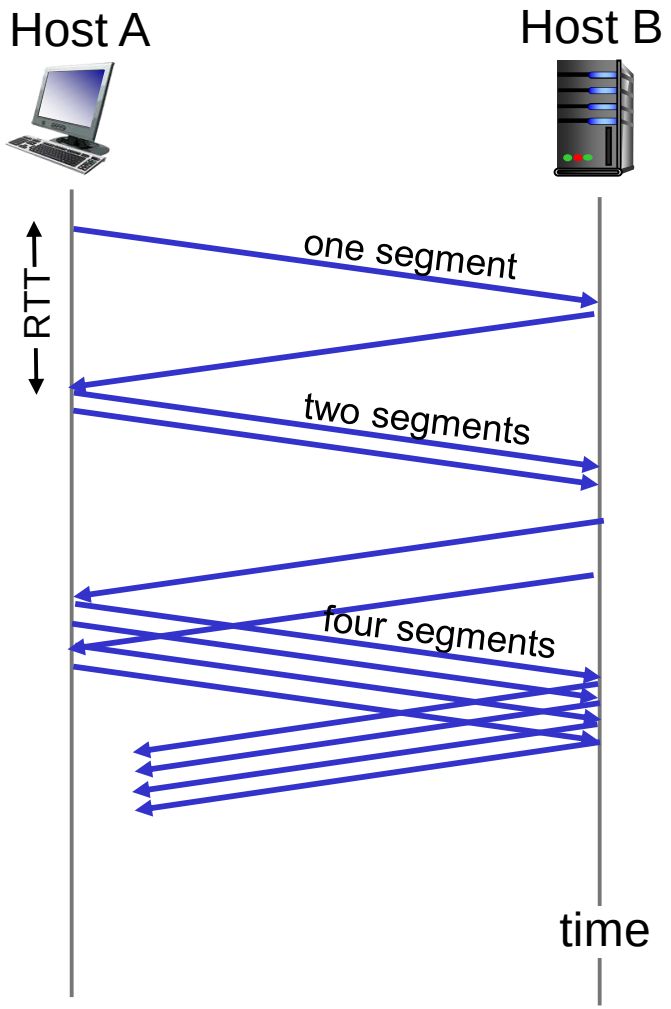
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

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

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

TCP slow start

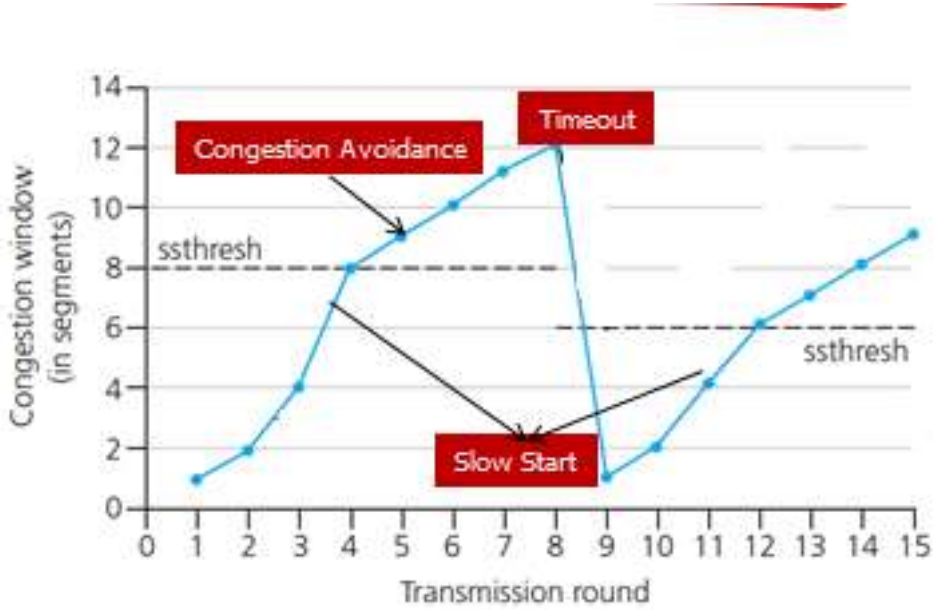
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



TCP: detecting, reacting to loss

- loss indicated by timeout:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- TCP Tahoe always sets **`cwnd`** to 1 (timeout or 3 duplicate ACKs)

TCP: Switching From Slow Start To Congestion Avoidance (CA)

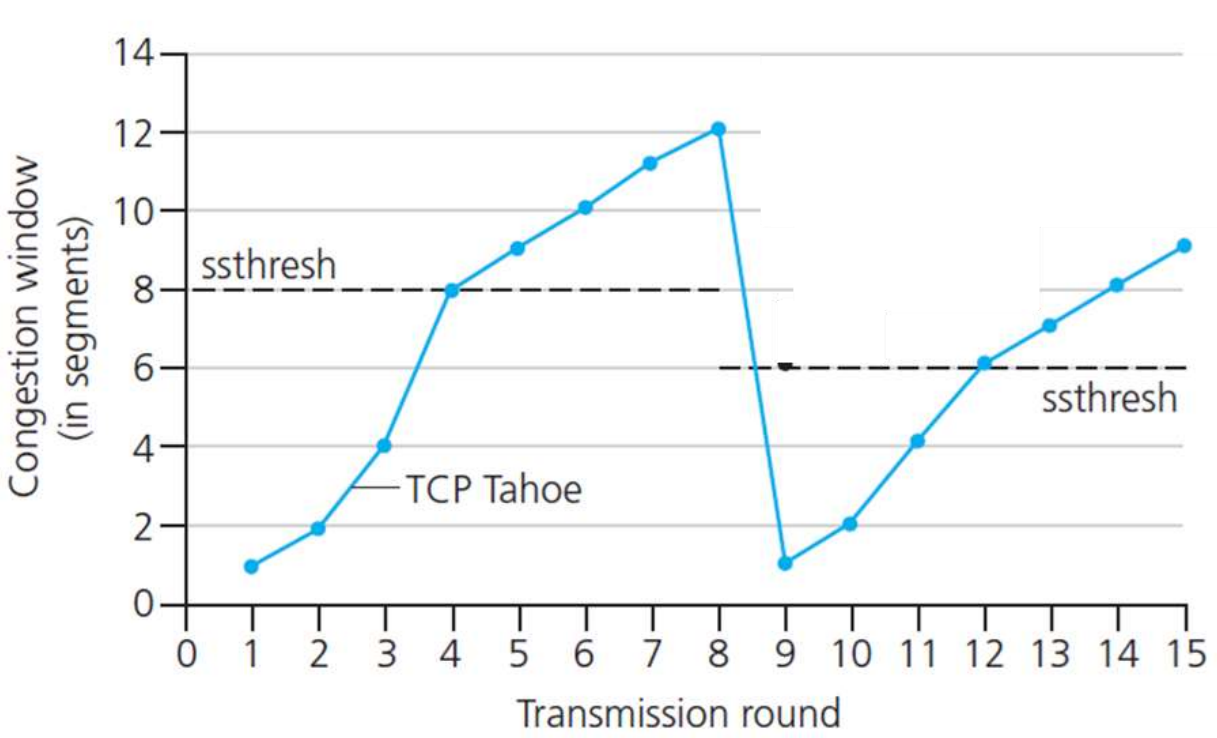


TCP TAHOE Implementation:

- variable **`ssthresh (slow-start threshold)`**
- on loss event:
  - **`ssthresh`** is set to ½ (half) of **`cwnd (cwnd/2)`** just before loss event
  - Value of cwnd is set to 1 MSS (slow start)

TCP: Slow Start & Congestion Avoidance (CA) (Loss Because of Timeout)

Phase	TR	C W	SS	ssthresh
Slow Start (1)	1	1	1	8
	2	2	3	8
	3	4	7	8
	4	8	15	8
CA	5	9	24	8
	6	10	34	8
	7	11	45	8
	8	12	57	8



TR 1 to 4
- Slow Start, Exponential growth, `ssthresh=8`

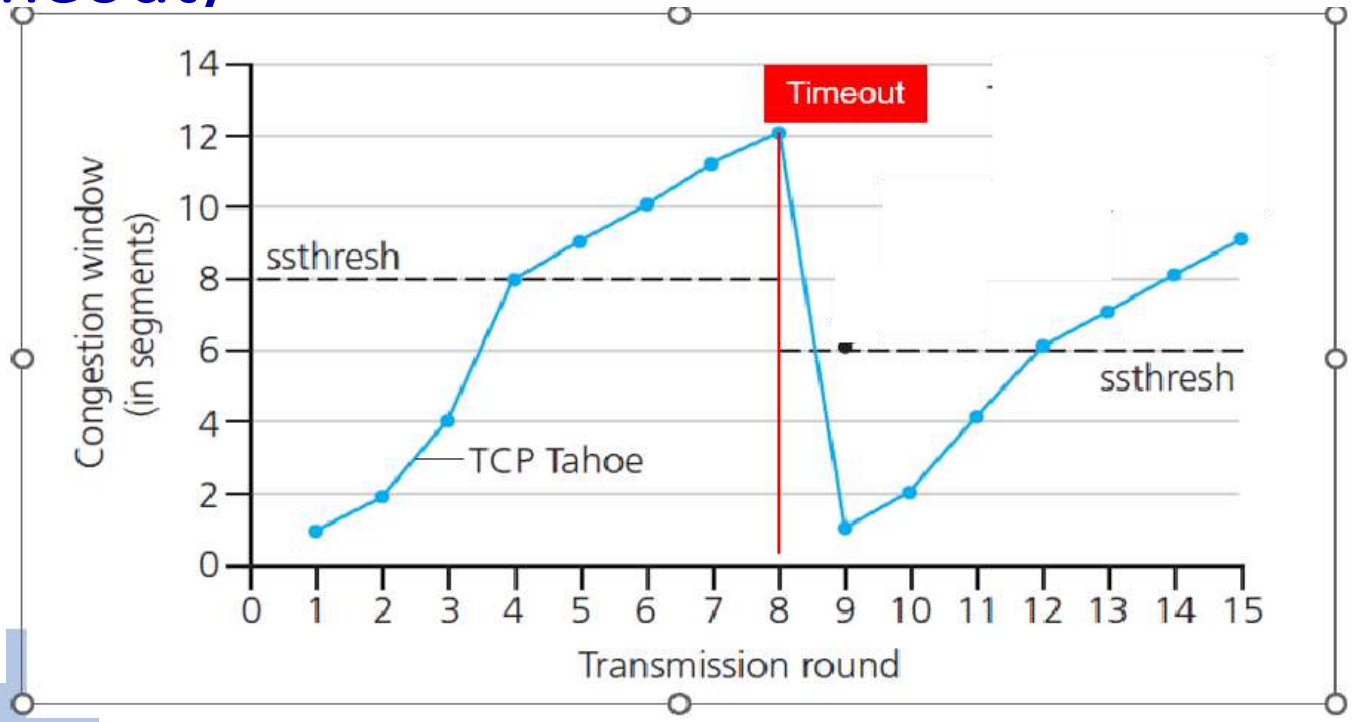
TR 4, CW = `ssthresh` is detected and Congestion Avoidance (CA) starts

TR 5 to 8
- Operate at CA, Linear growth

TR = Transmission Round
CW = Congestion Window
SS = Segment Send
`ssthresh` = Slow Start Threshold

TCP: Slow Start & Congestion Avoidance (CA) (Loss because of Timeout)

Phase	TR	C W	SS	ssthresh
Slow Start (1)	1	1	1	8
	2	2	3	8
	3	4	7	8
	4	8	15	8
CA	5	9	24	8
	6	10	34	8
	7	11	45	8
	8	12	57	12 / 2 = 6



After TR 8
- Timeout is detected
- New ssthresh value will be calculated;
  = $cwnd_{timeout}$ / 2

  = 12 / 2 = 6

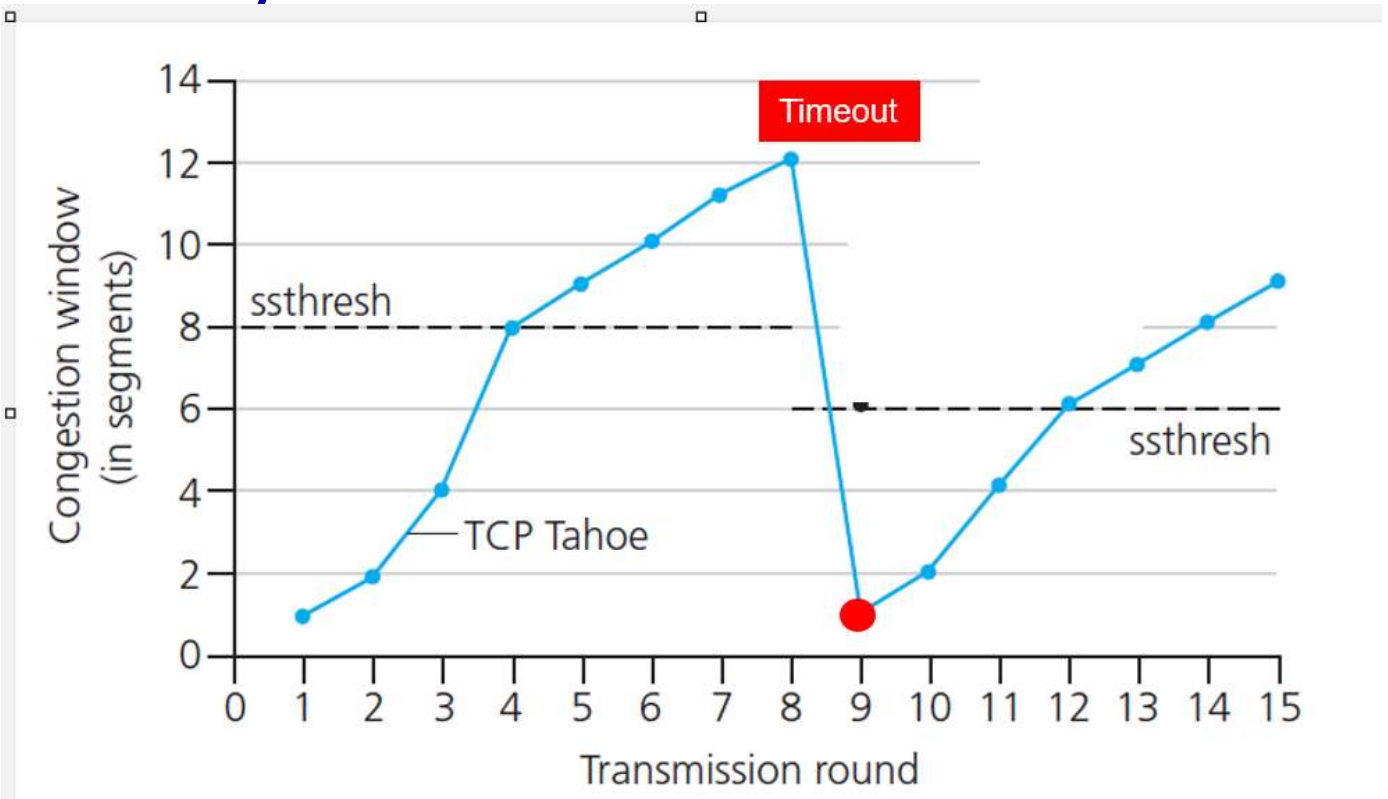
TR = Transmission Round
CW = Congestion Window
SS = Segment Send
ssthresh = Slow Start Threshold

TCP: Slow Start & Congestion Avoidance (CA) (Loss because of Timeout)

Phase	TR	CW	SS	ssthresh
Slow Start (2)	9	1	58	6
	10	2	60	6
	11	4	64	6
	12	6 (8)	70	6
CA	13	7	77	6
	14	8	85	6
	15	9	94	6

TR 9 to 12 (refer table)
- CW=1, and **ssthresh=6**
- Start Slow, Exponential Growth

TR 13 to 15
- Operate at CA, Linear Growth

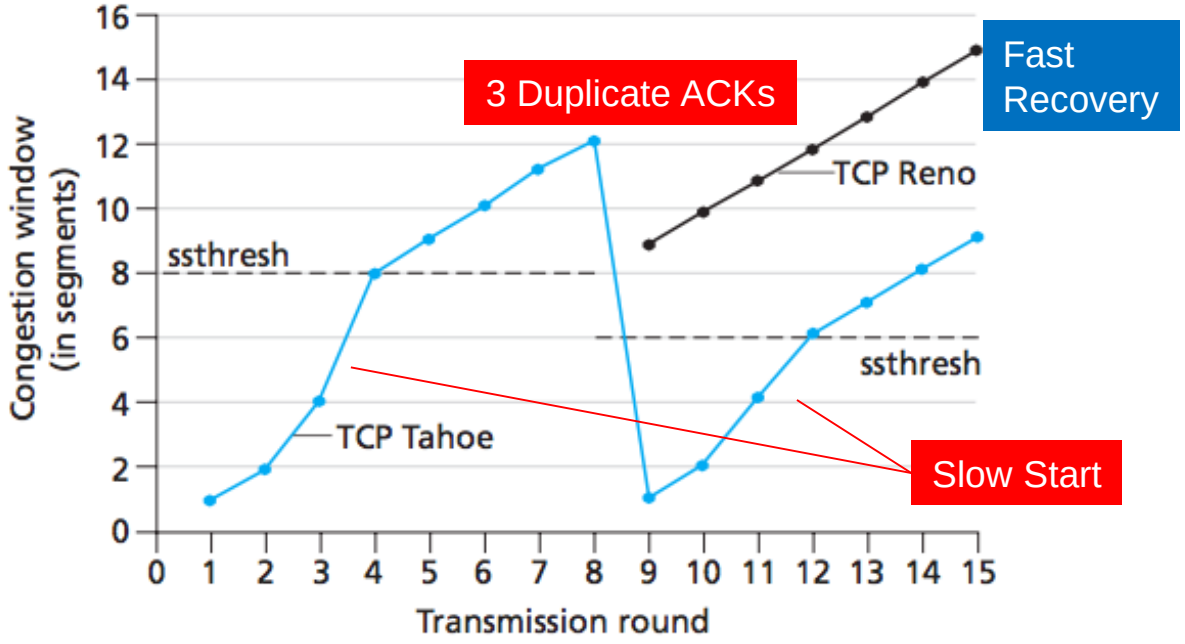


TR = Transmission Round
CW = Congestion Window
SS = Segment Send
ssthresh = Slow Start Threshold

TCP: Fast Recovery
(Loss because of 3 Duplicate ACK)

Earlier version of TCP (TCP TAHOE) entered Slow start

Newer version of TCP (TCP RENO) incorporated fast recovery



TCP RENO Implementation

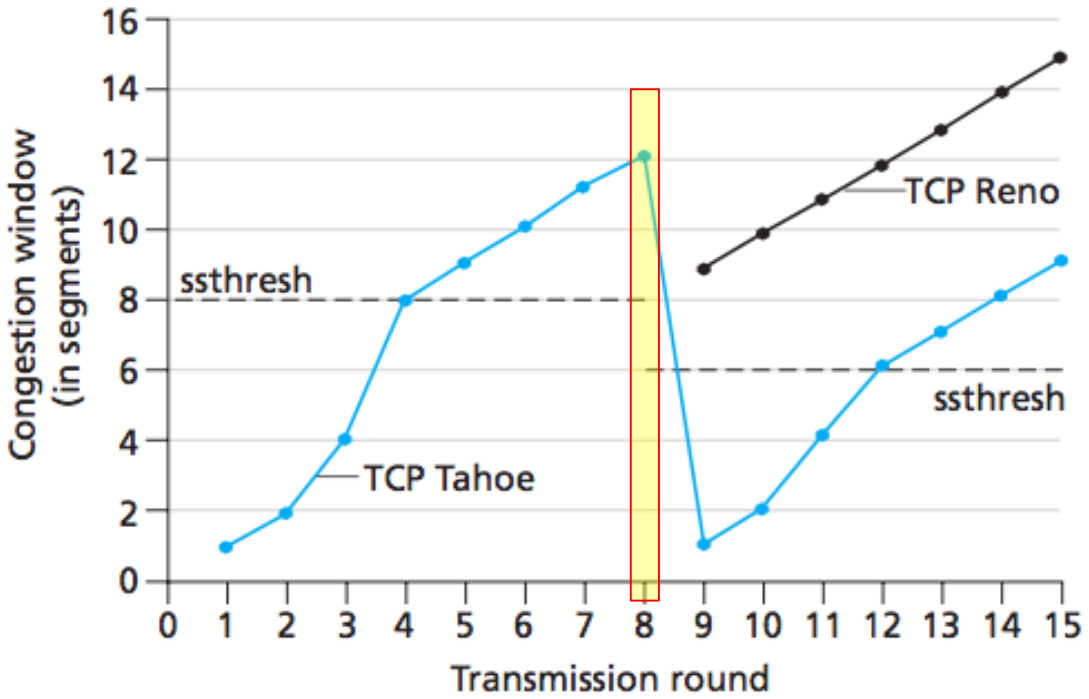
- variable **`ssthresh (slow-start threshold)`**
- on loss event:
  - **`ssthresh`** is set to ½ (half) of **`cwnd(cwnd/2)`** just before loss event
  - Set **cwnd value** to ssthresh plus 3 (ssthresh+3) the segment size
    - **The equation → cwnd=ssthresh + 3MSS**

TCP: Fast Recovery
(Loss because of 3 Duplicate ACK)

TR	CW	SS	ssthresh
8	12	57	12 / 2 = 6

TR	CW	SS	ssthresh
9			
10			
11			
12			
13			
14			
15			

After TR 8, Triple Duplicate ACKs is detected



TR = Transmission Round
CW = Congestion Window
SS = Segment Send
ssthreshold = Slow Start Threshold

TCP: Fast Recovery
(Loss because of 3 Duplicate ACK)

TR	CW	SS	ssthresh
8	12	57	12 / 2 = 6

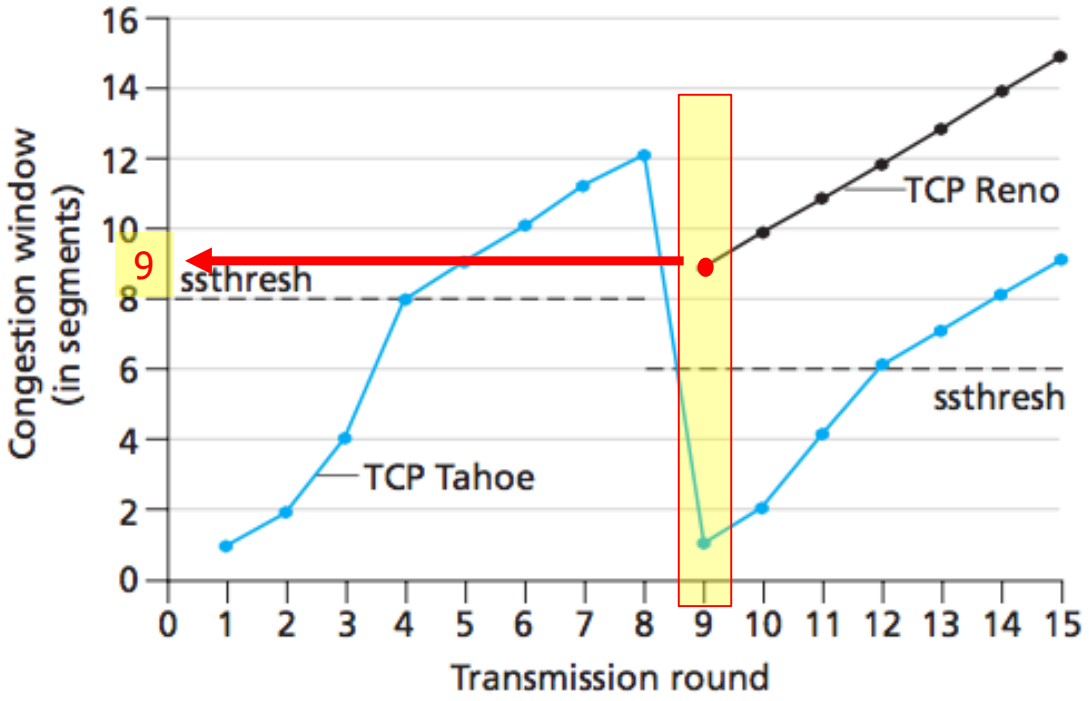
TR	CW	SS	ssthresh
9	9	66	6
10	10	76	6
11	11	87	6
12	12	99	6
13	13	112	6
14	14	126	6
15	15	141	6

After TR 8, 3DUP ACKs is detected

At TR 9
→ CW = 6+3 = 9

TR 9,10 to 15

- Enters Fast Recovery
- Operate at CA
- Linear growth



TR = Transmission Round
CW = Congestion Window
SS = Segment Send
ssthreshold = Slow Start Threshold

TCP: Fast Recovery
(Loss because of 3 Duplicate ACK)

TR	CW	SS	ssthresh
8	12	56	12 / 2 = 6

TR	CW	SS	ssthresh
9	9	65	6
10	10	75	6
11	11	86	6
12	12	98	6
13	13	111	6
14	14	125	6
15	15	140	6

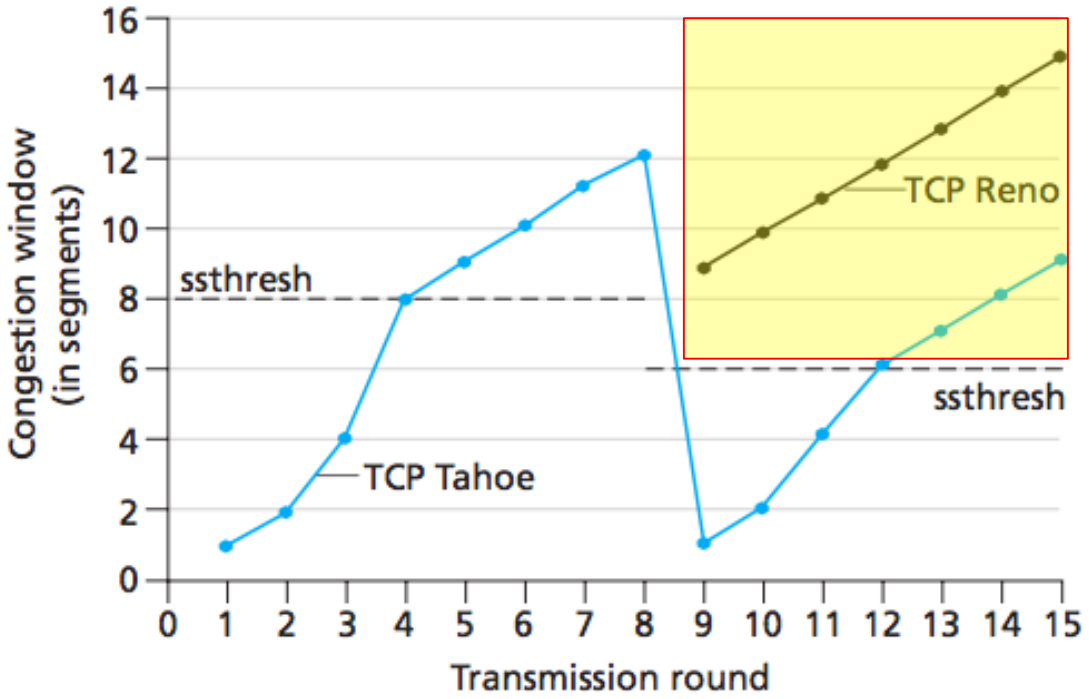
After TR 8 3DUP ACKs is detected

TR 9
→ CW=6

TR 9,10 to 15

TCP RENO

- Enters Fast Recovery
- Operate at CA
- Linear growth



TR = Transmission Round
CW = Congestion Window
SS = Segment Send
`ssthreshold` = Slow Start Threshold

TCP: detecting, reacting to loss

TCP RENO

- loss indicated by timeout: (Slow Start)
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs (Fast Recovery)
  - dup ACKs indicate network capable of delivering some segments
  - **`cwnd`** is cut in half window plus 3 times the segment then grows linearly

TCP TAHOE

- loss indicated by timeout or 3 duplicate ACKs : (Slow Start)
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

Initial threshold = 6
Timeout @ TR 8

Self-Test

Question:

Supposed host A connected to host B for transmitting segments over TCP with congestion control.

Assume that the initial threshold is 6 MSS.

If the timeout event occurred at transmission round (TR=8), answer the following questions:

TR	CW	SS
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		

Note: CW (Congestion Window), SS (Segment Send)

Initial threshold = 6
Timeout @ TR 8

Self-Test

...

a) Complete the table.

b) What is the new threshold after timeout?

c) What is the range of TRs involved in the congestion avoidance.

d) What is the range of TRs involved in the fast recovery?

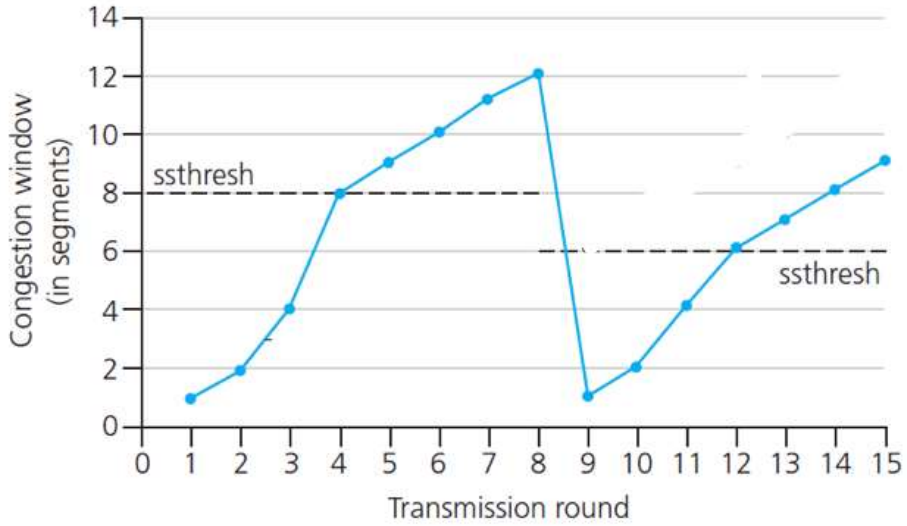
e) At which TR the new threshold applied and how many segments sent at that TR?

TR	CW	SS
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		

Note: CW (Congestion Window), SS (Segment Send)

Initial threshold = 6
Timeout @ TR 8

Self-Test



TR	CW	SS
1	1	1
2	2	3
3	4	7
4	6 threshold1	13
5	7	20
6	8	28
7	9	37
8	10	47
9	1	48
10	2	50
11	4	54
12	5 threshold2	59
13	6	65
14	7	72

c)What is the range of TRs involved in the congestion avoidance.

= 5 to 8, 13 to 14

Note: CW (Congestion Window), SS (Segment Send)

Initial threshold = 6
Timeout @ TR 8

Self-Test

Solution:

d)What is the range of TRs involved in the fast recovery?

= none. Loss because of timeout only

e)At which TR the new threshold applied and how many segments sent at that TR?

TR = 12;

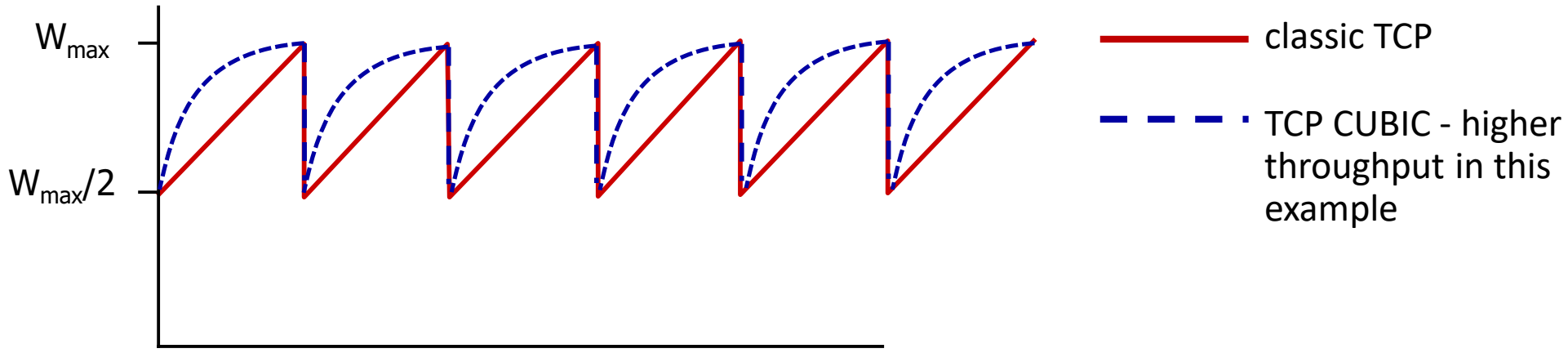
Segment sent = 59

TR	CW	SS
1	1	1
2	2	3
3	4	7
4	6 threshold1	13
5	7	20
6	8	28
7	9	37
8	10	47
9	1	48
10	2	50
11	4	54
12	5 threshold2	59
13	6	65
14	7	72

Note: CW (Congestion Window), SS (Segment Send)

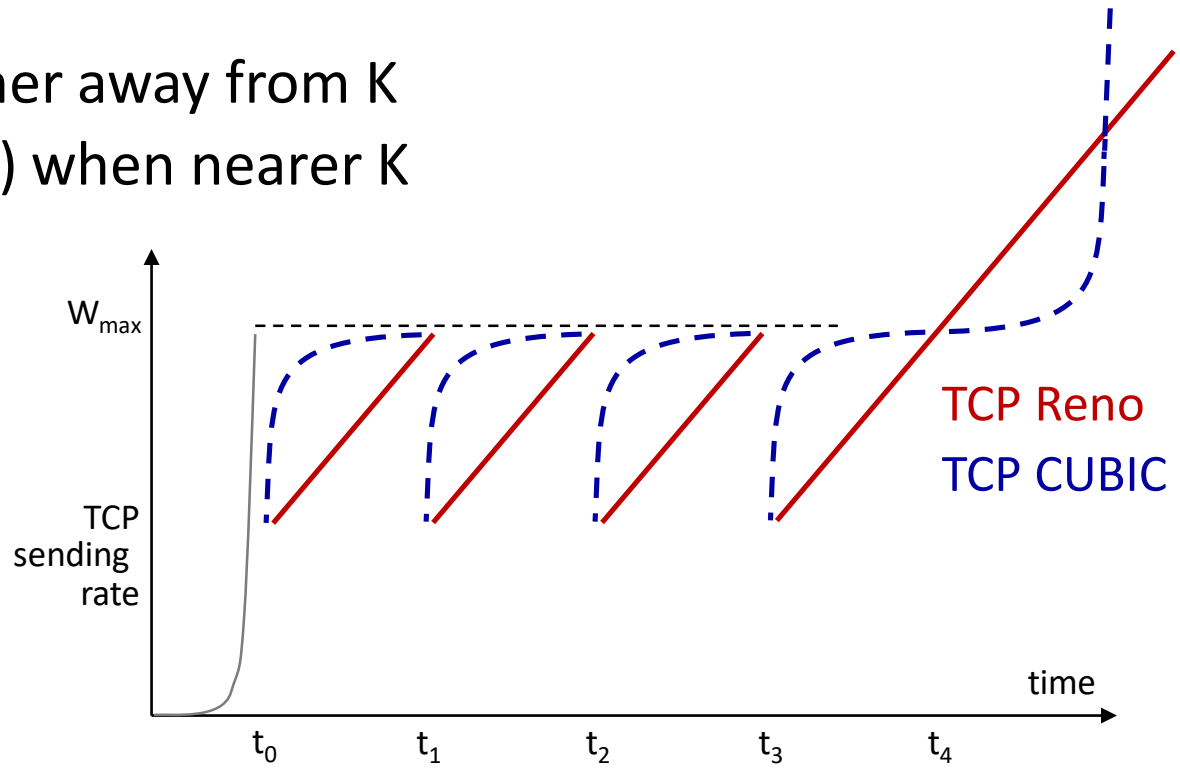
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



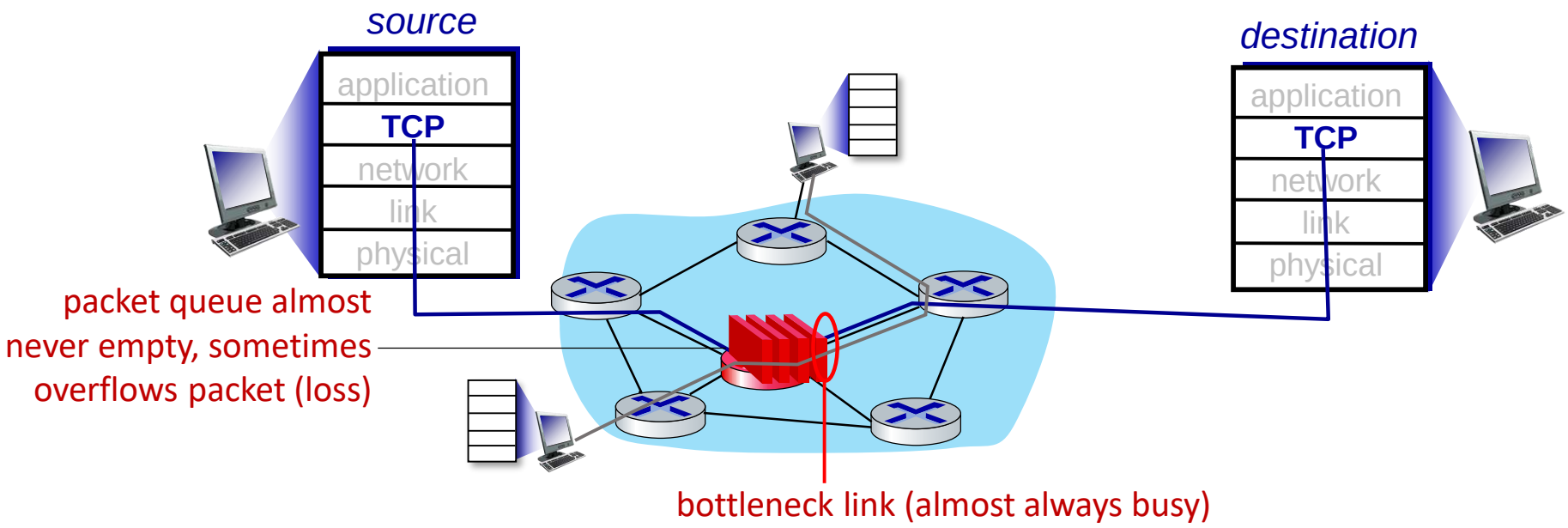
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



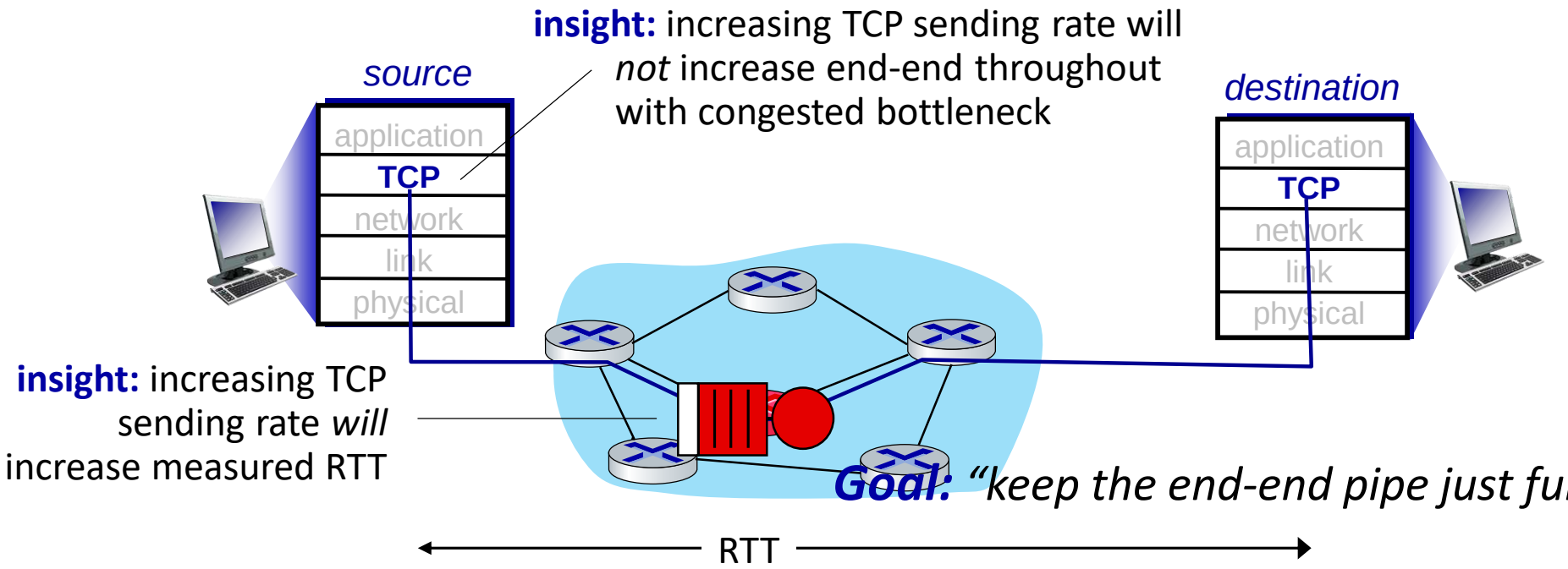
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



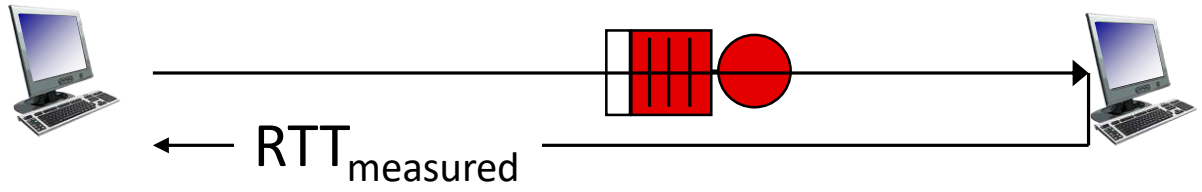
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



Delay-based TCP congestion control

Keeping sender-to-receiver pipe "just full enough, but not fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}

if measured throughput "very close" to uncongested throughput
    increase `cwnd` linearly /* since path not congested */
else if measured throughput "far below" uncongested throughout
    decrease `cwnd` linearly /* since path is congested */

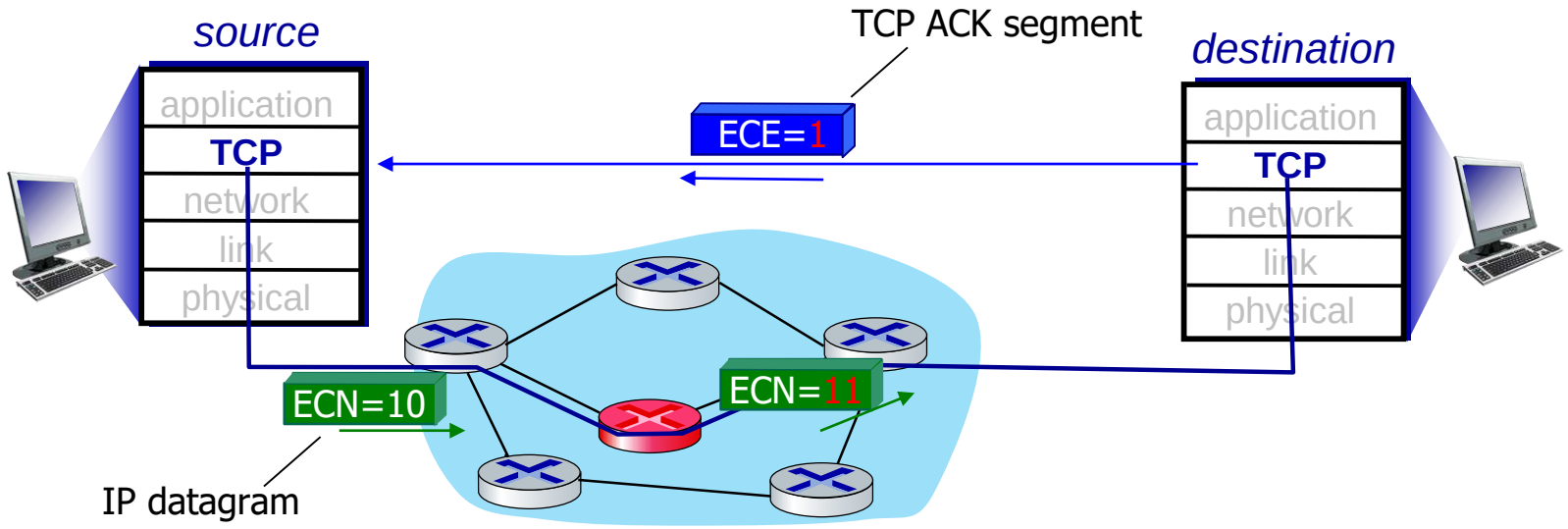
Delay-based TCP congestion control

- congestion control without inducing/forcing loss
- maximizing throughout ("keeping the just pipe full… ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - TCP BBR deployed on Google's (internal) backbone network

Explicit congestion notification (ECN)

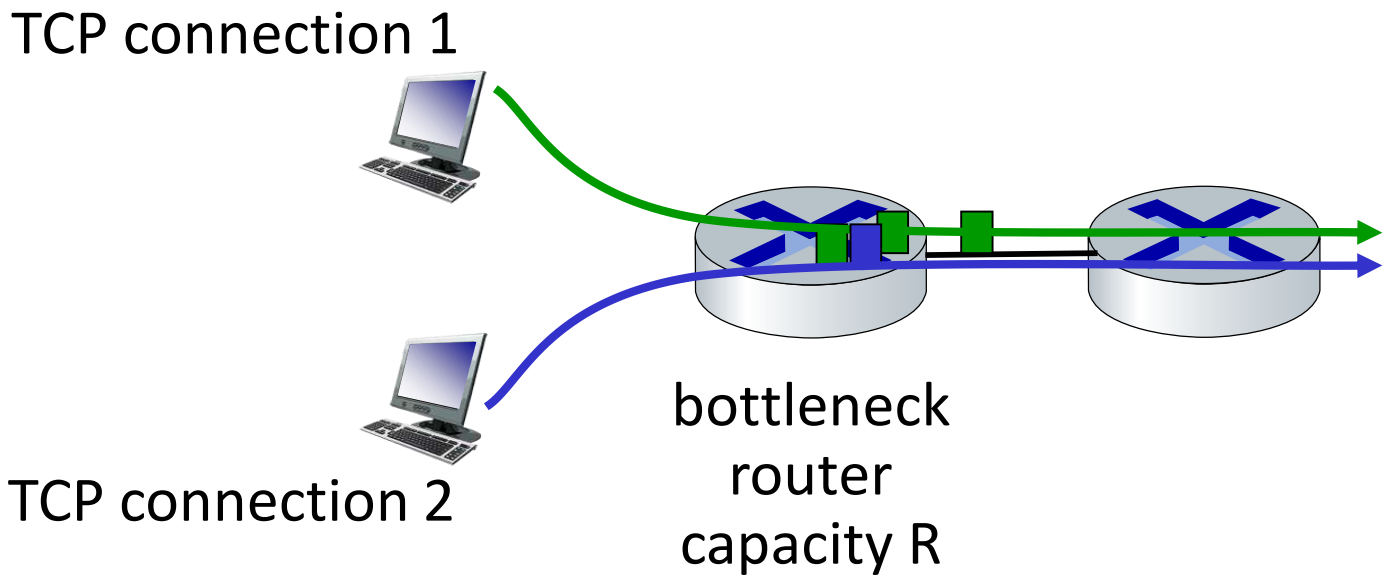
TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



TCP fairness

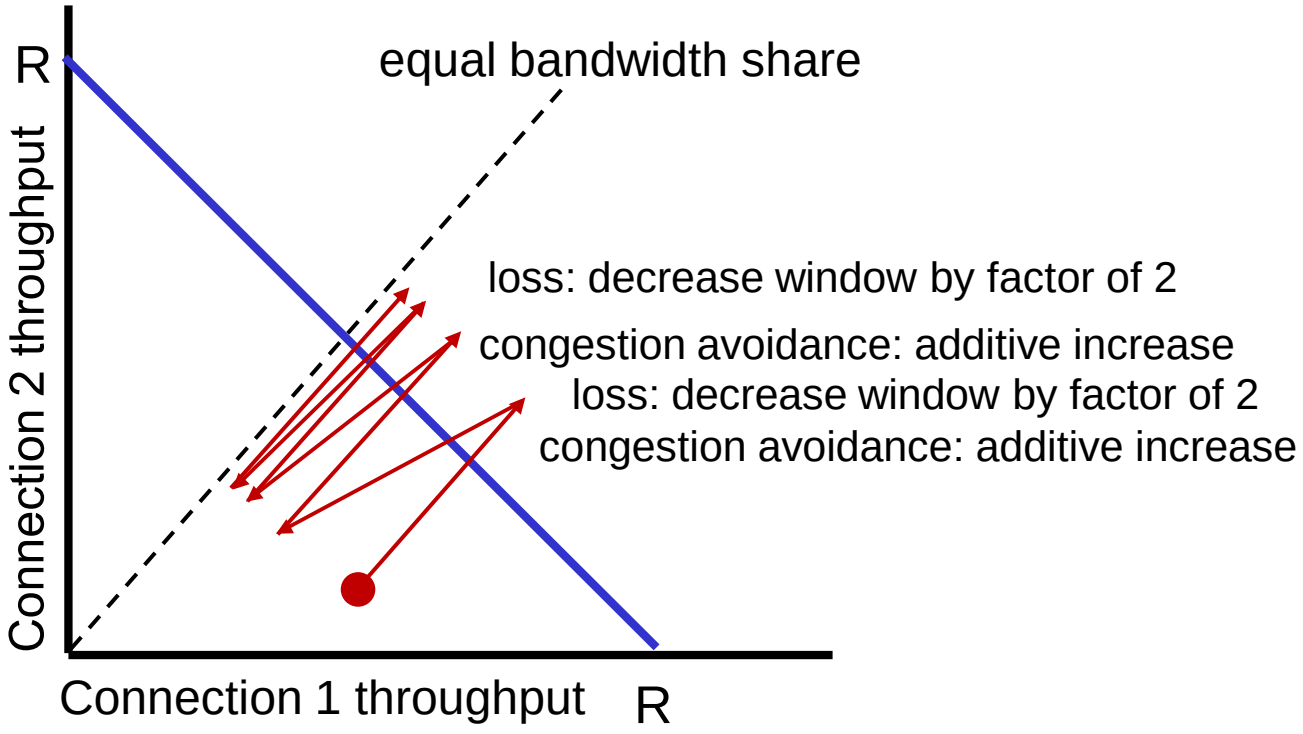
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

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



Is TCP fair?

A: Yes, under idealized assumptions:

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

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

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

Evolving transport-layer functionality

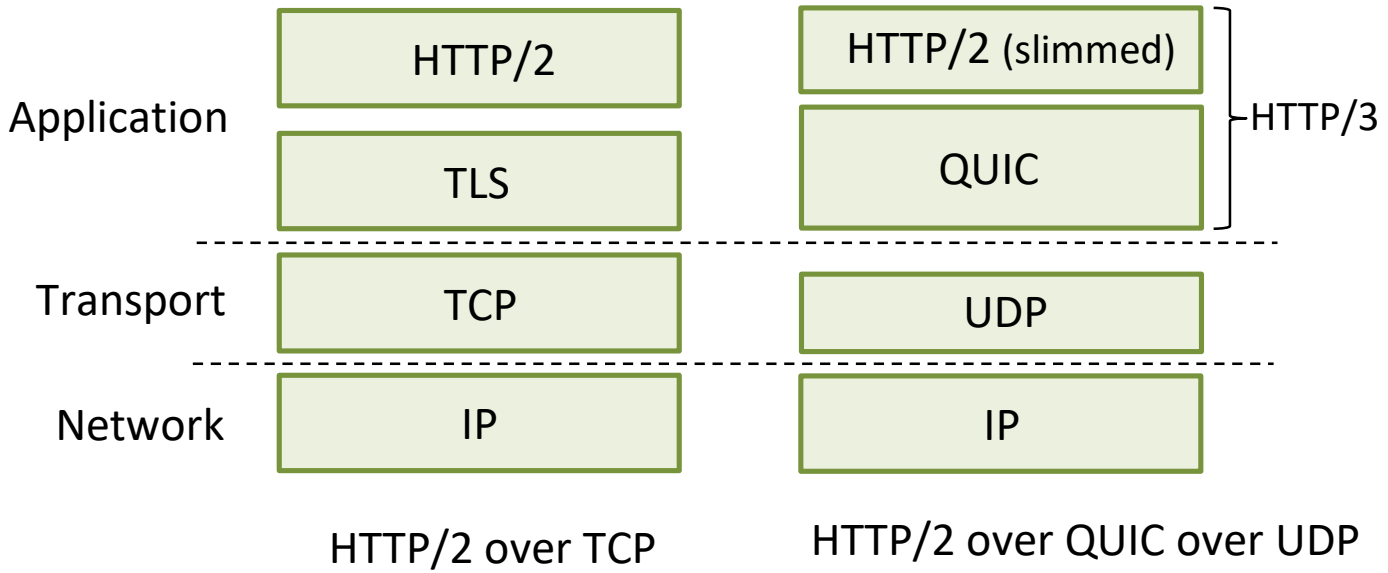
- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets “in flight”; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, “background” TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



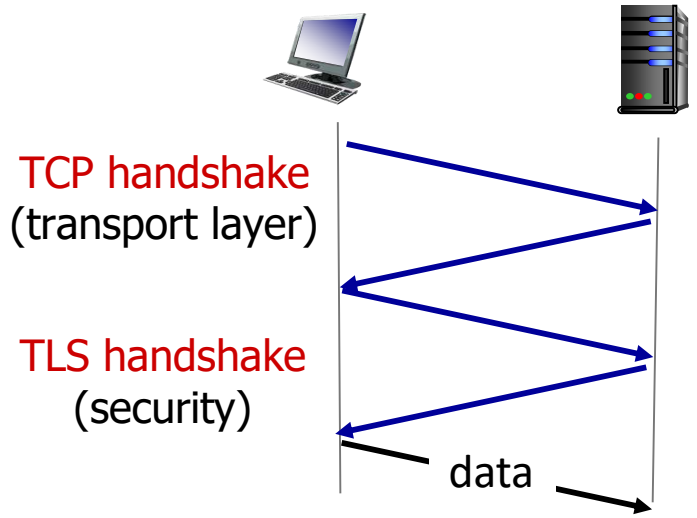
QUIC: Quick UDP Internet Connections

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

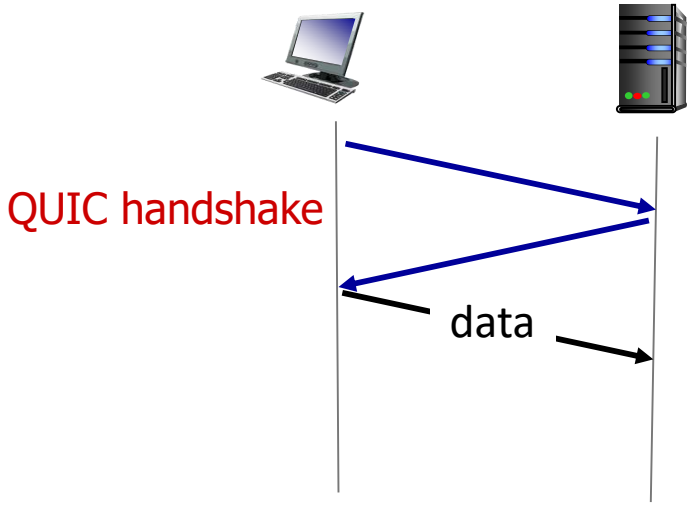
- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment



TCP (reliability, congestion control state) + TLS (authentication, crypto state)

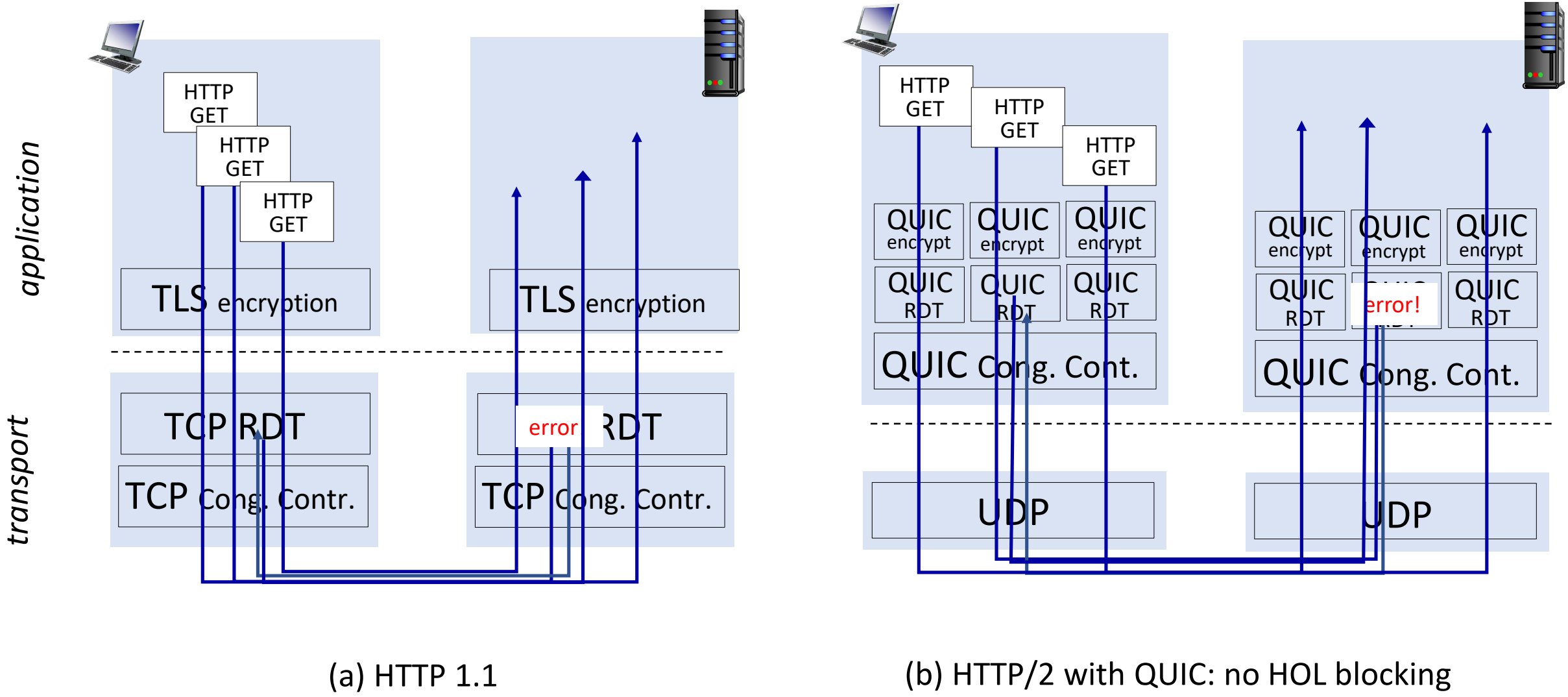
- 2 serial handshakes



QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

(b) HTTP/2 with QUIC: no HOL blocking

Chapter 3: summary

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

Up next:

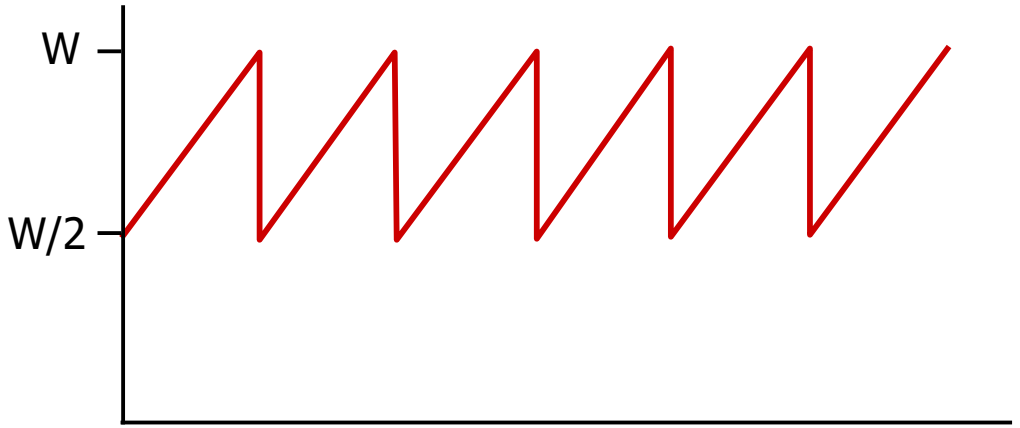
- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane

Additional Chapter 3 slides

TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

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



TCP Futures: TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

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

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

- new versions of TCP for high-speed