Applied!

Computer Networks

Behnam Amiri

acn.dailysec.ir

aComputerNetworks.github.io

Spring 2025

Fast Recap

Application Layer

- Web and HTTP
- HTTP Request
  - GET, POST, ...
- HTTP Response
  - **`HTTP/1.1 200 OK`**
- HTTP is over TCP

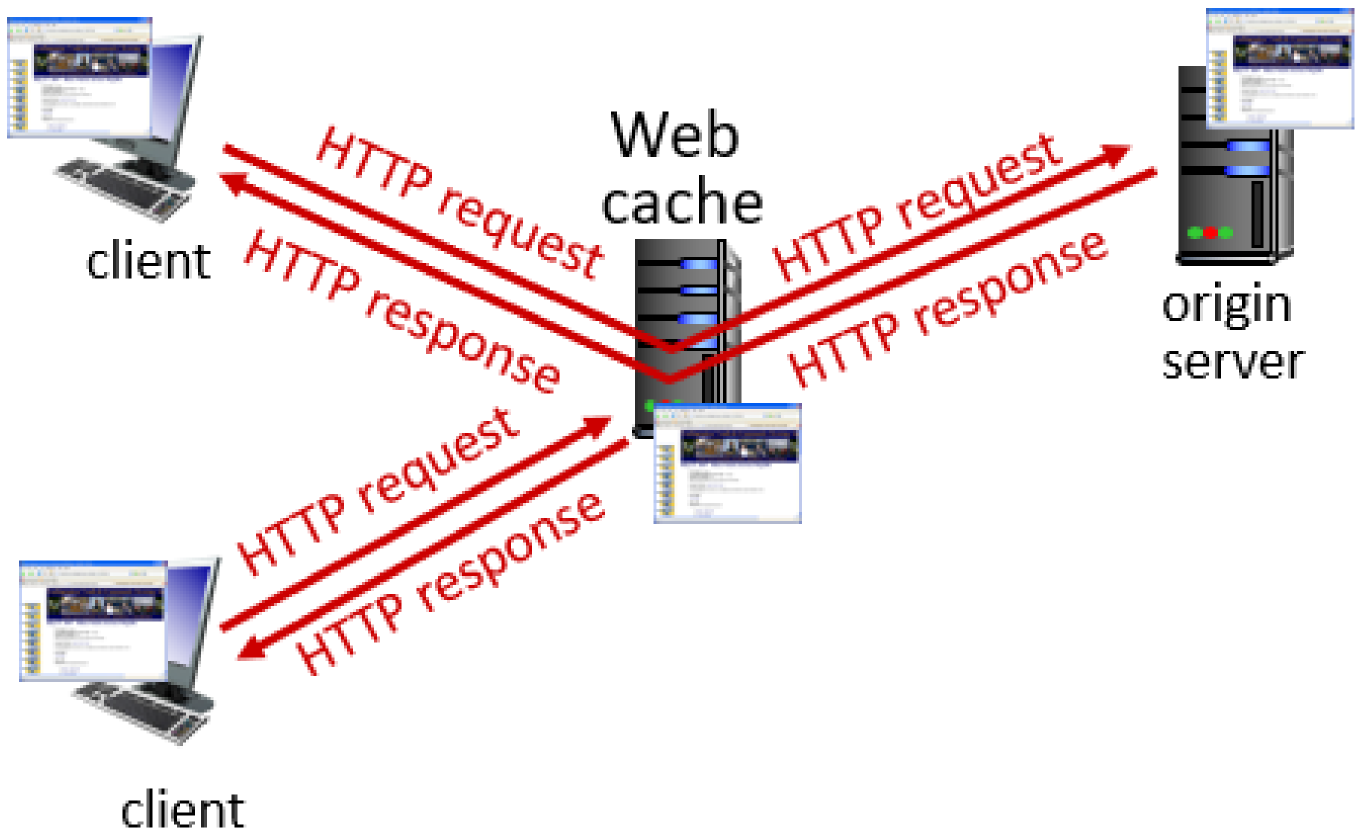
$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

Maintaining user/server state

- HTTP is stateless protocol.
- HTTP can’t remember previous actions.
- Web sites and client browser use cookies to maintain some state between transactions.
- Cookies can be used to:
  - track user behavior on a given website (first party cookies)
  - track user behavior across multiple websites (third party cookies) without user ever choosing to visit tracker site (!)
  - tracking may be *invisible* to user.
- GDPR (EU General Data Protection Regulation) and cookies

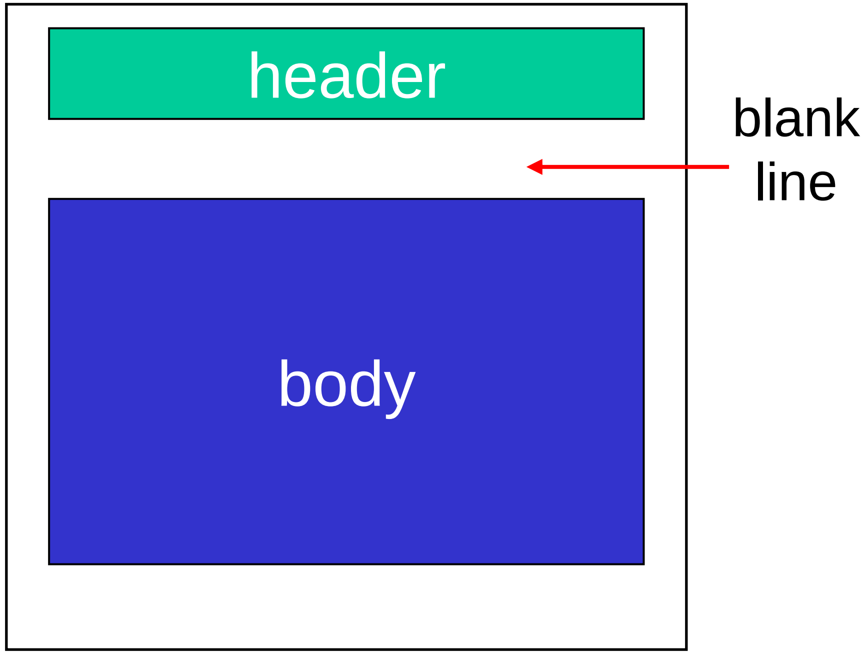
Web Cache

- Goal: satisfy client requests without involving origin server



EMAIL

- SMTP protocol between mail servers to send email messages
  - client: sending mail server
  - server: receiving mail server
- Uses TCP to reliably transfer email message
- Use port 25 TCP
- Mail message format

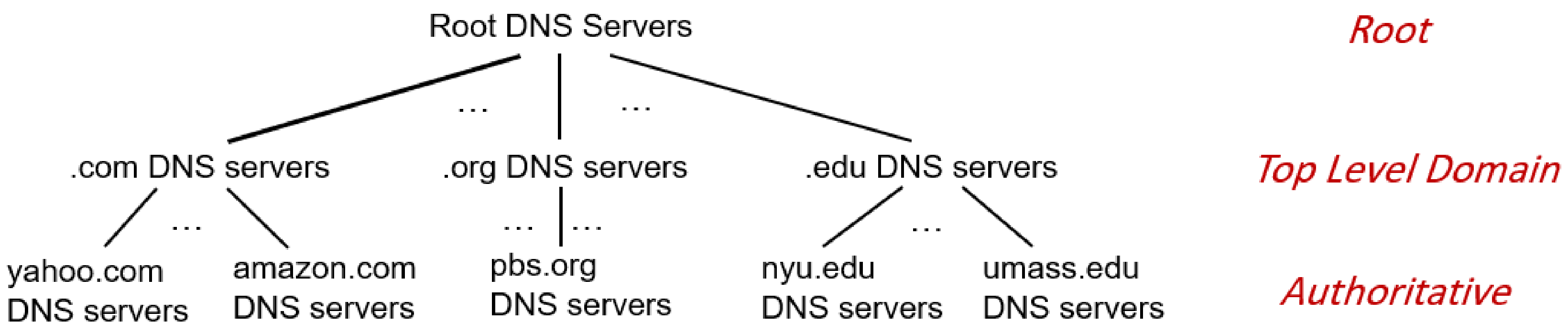


Receive Email

- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501].
  - Messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server.
- HTTP: Gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages.

DNS: Domain Name System

- Translate *Domain names* to *IP.*
- *distributed database* implemented in hierarchy of many *name servers.*



DNS records

DNS: distributed database storing resource records (RR)

RR format: (`name, value, type, ttl`)

type=A

- `name` is hostname
- `value` is IP address

type=NS

- `name` is domain (e.g., foo.com)
- `value` is hostname of authoritative name server for this domain

type=CNAME

- `name` is alias name for some "canonical" (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- `value` is canonical name

type=MX

- `value` is name of SMTP mail server associated with `name`

DNS Security

- DNS request and response are in clear text (*Sniff*)
- Change request and response (Spoofing)
- DNSec is a good solution!

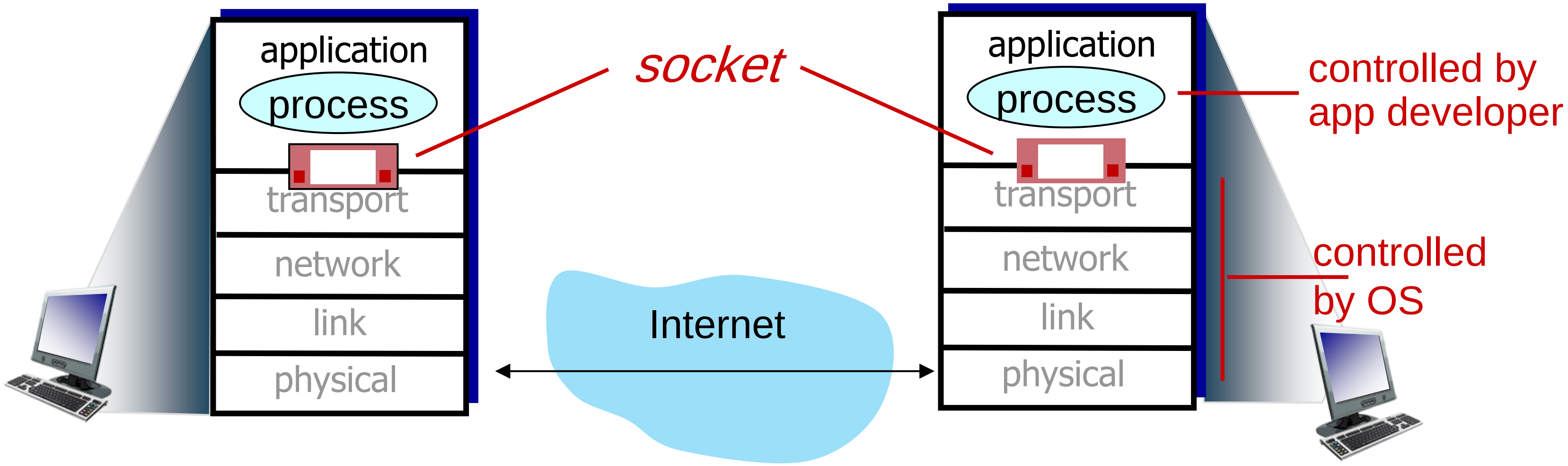
Other topics

- P2P file sharing
  - Like torrent
- CDN
  - Distribute web contents over servers.

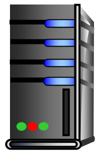
Socket Programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Client/server socket interaction: UDP



server (running on serverIP)

client

Server:

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

Client:

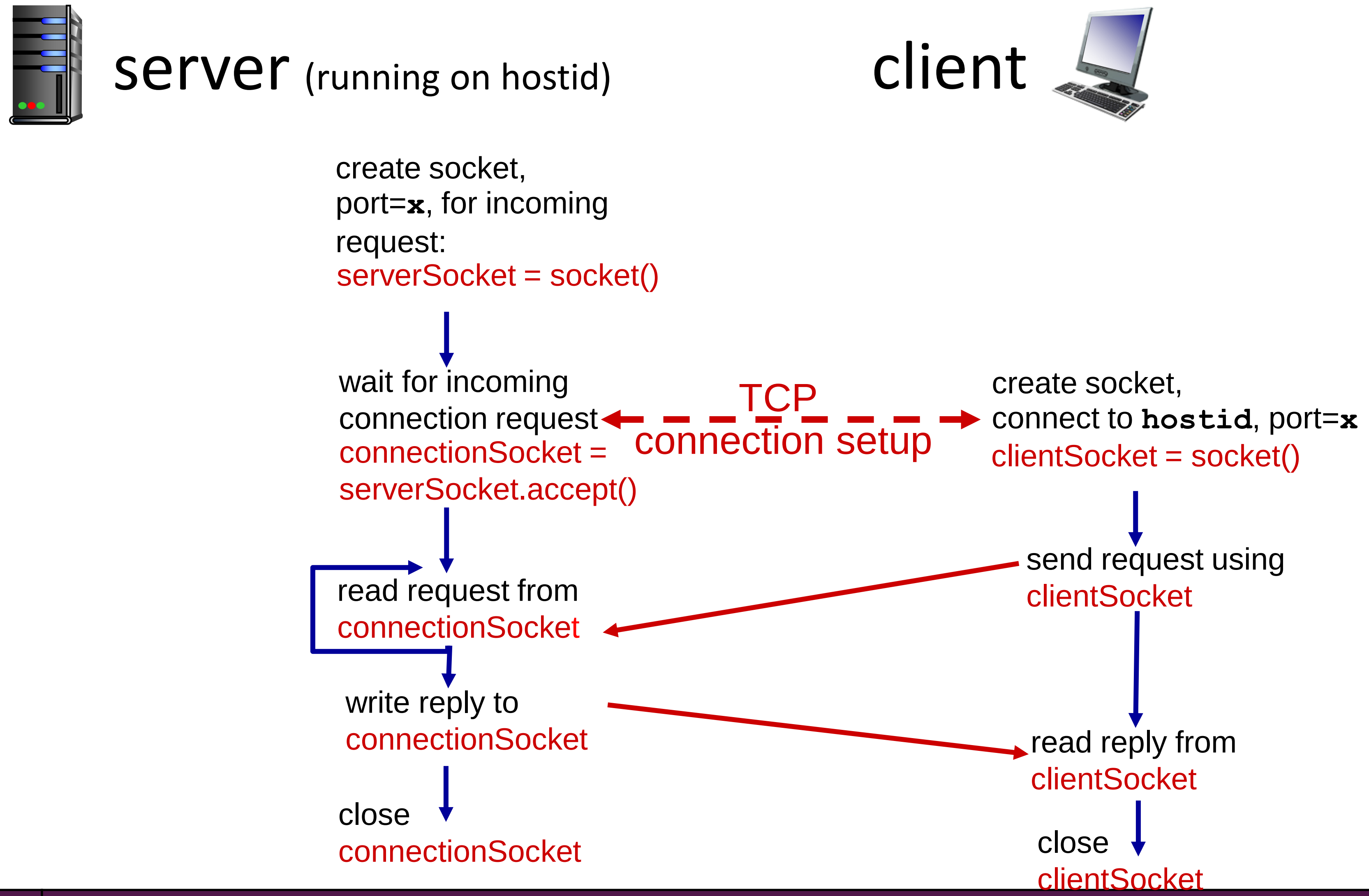
create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Client/server socket interaction: TCP

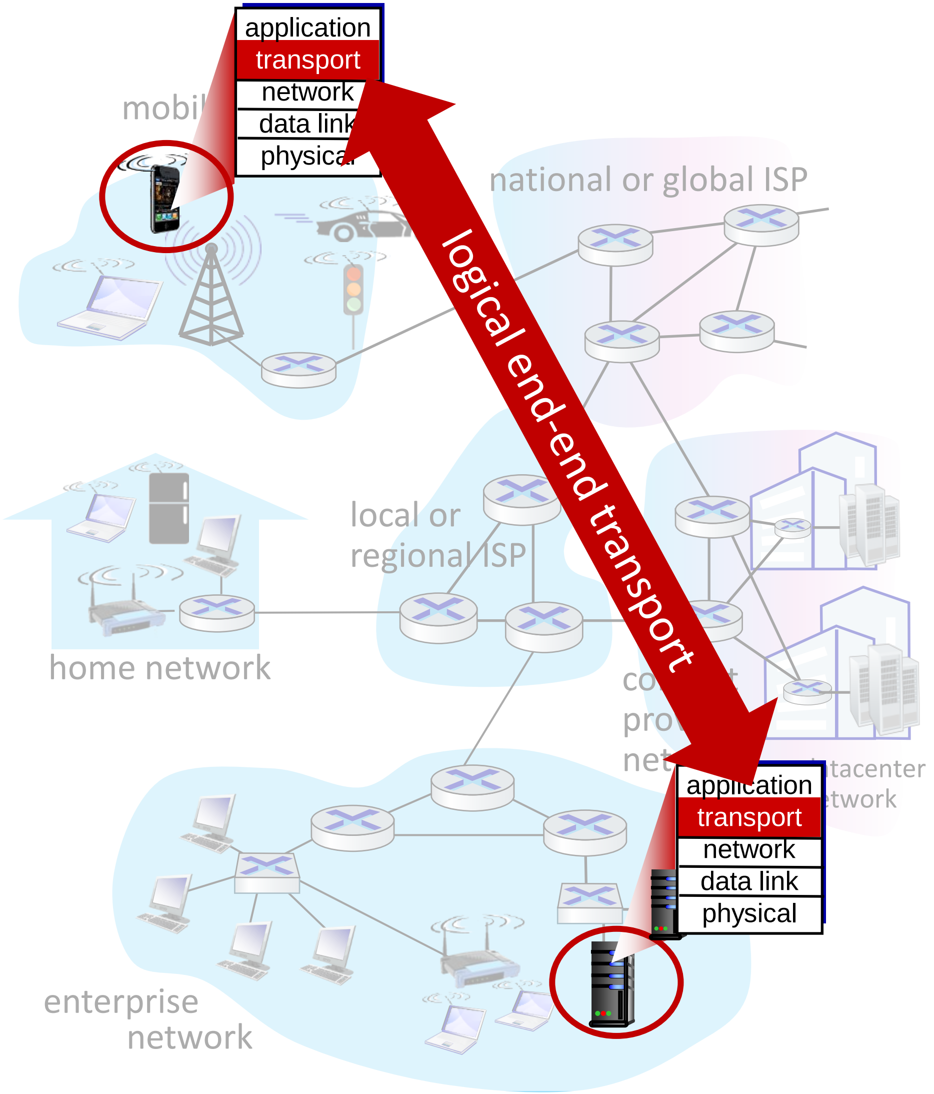


Transport Layer

Based on https://gaia.cs.umass.edu/kurose_ross/index.php slides.

Transport services and protocols

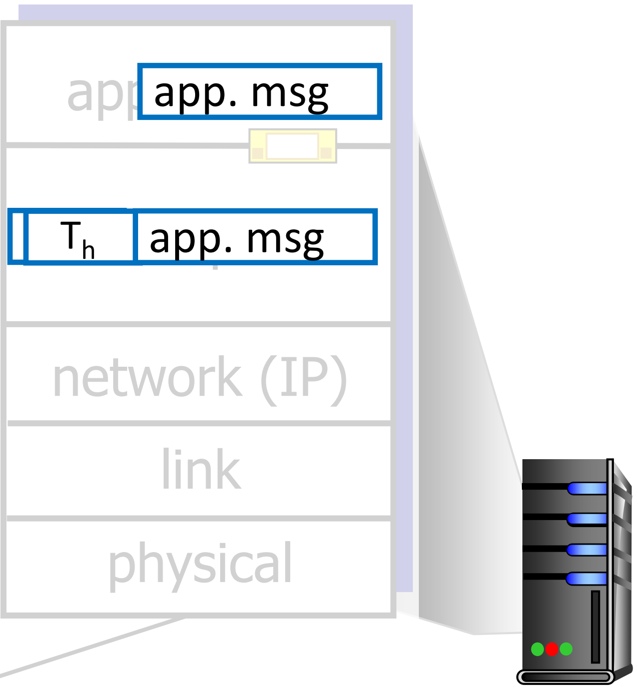
- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two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



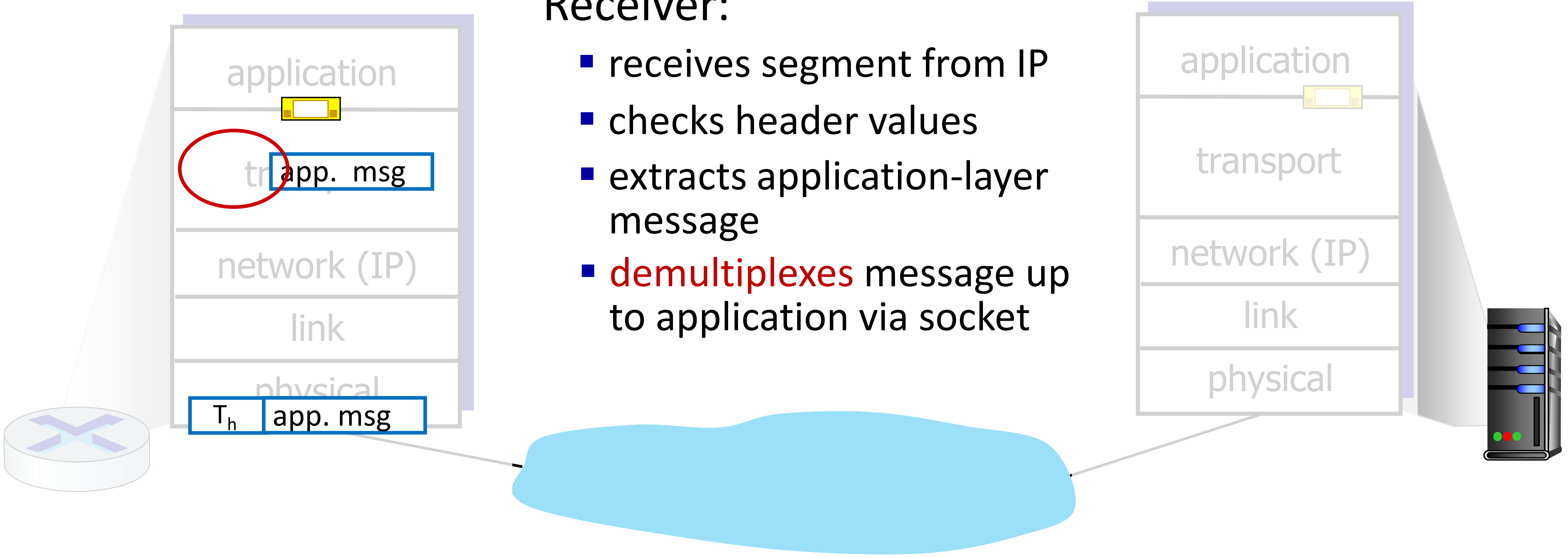
Transport Layer Actions

Sender:

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



Transport Layer Actions



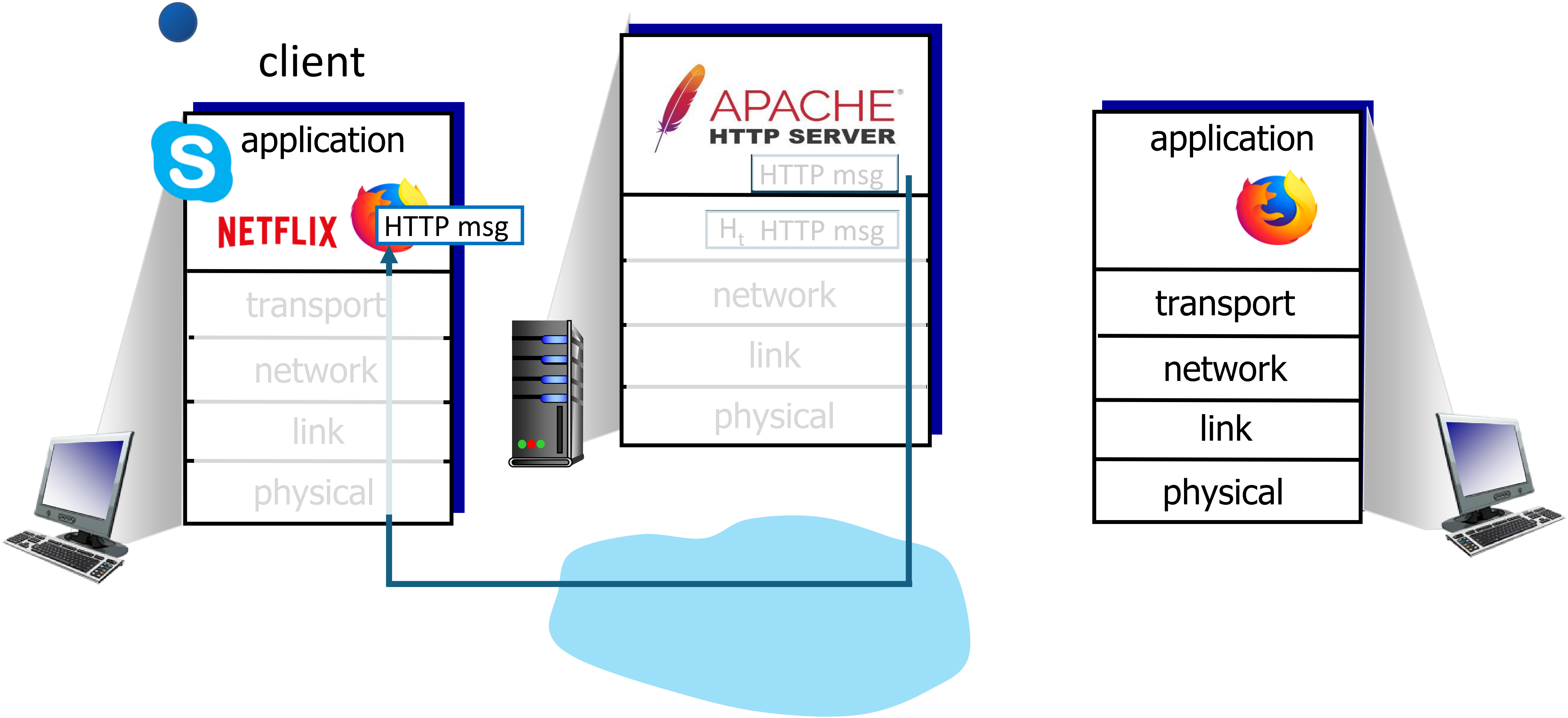
Receiver:

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

Internet transport protocols

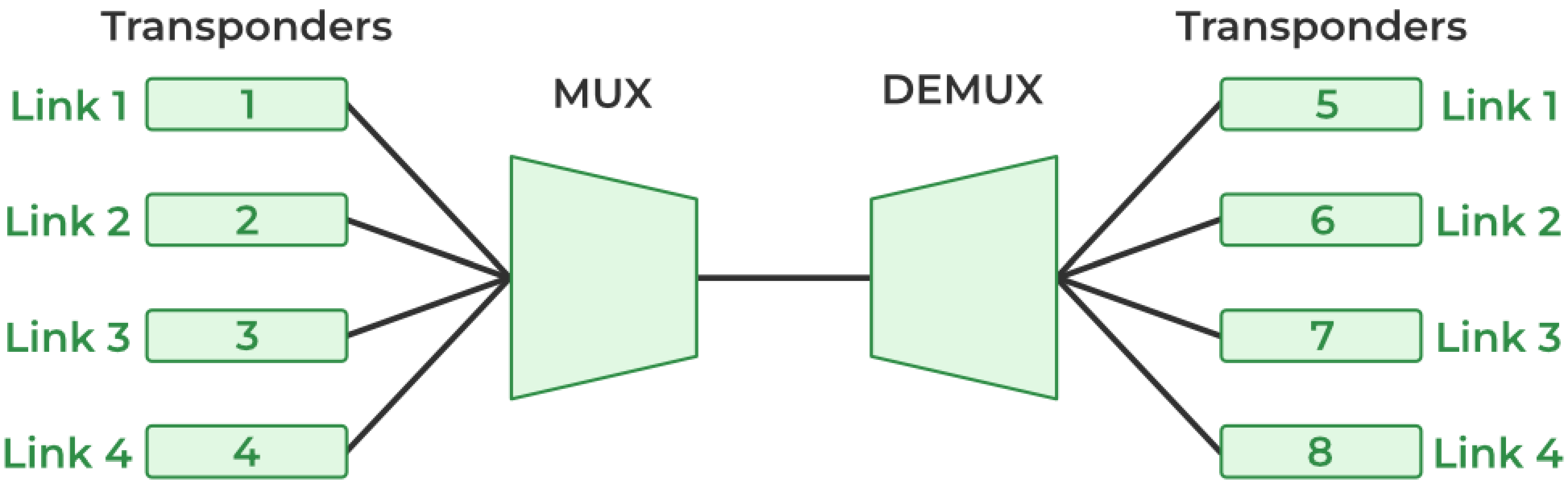
- **TCP:** Transmission Control Protocol
  - Reliable
  - in-order delivery
  - Congestion control
  - Flow control
  - Connection setup
- **UDP:** User Datagram Protocol
  - Unreliable
  - Unordered delivery
  - Faster than TCP

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



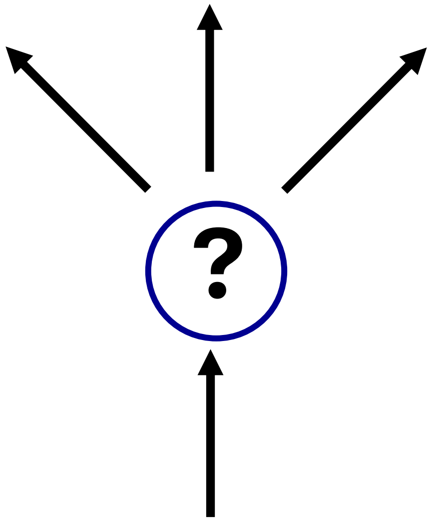
Multiplexing

Multiplexing

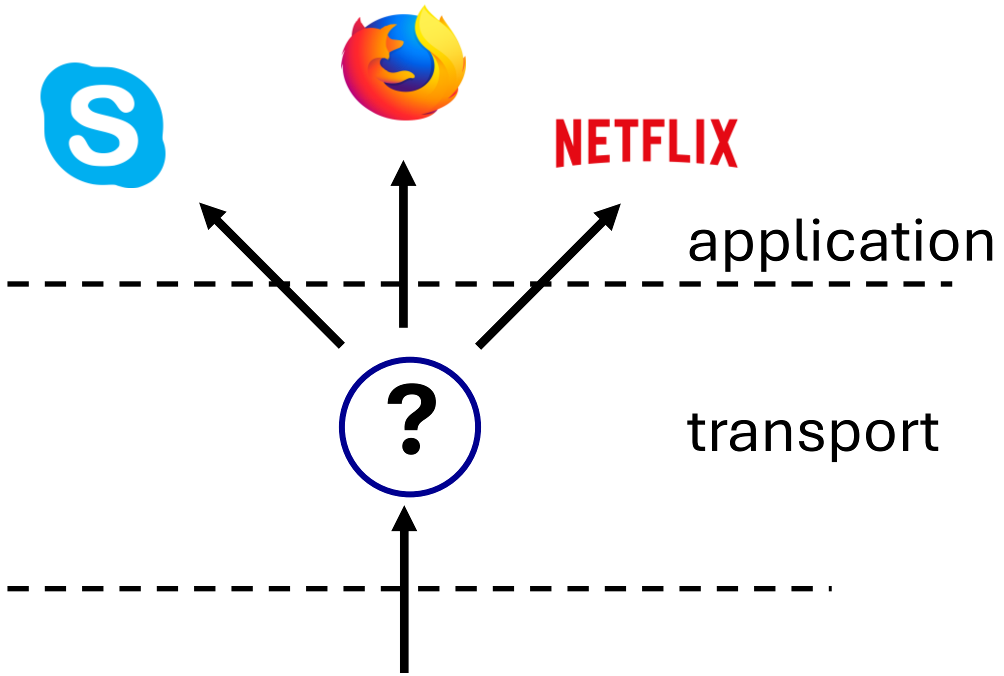


Multiplexing Example

- Speak in different languages.

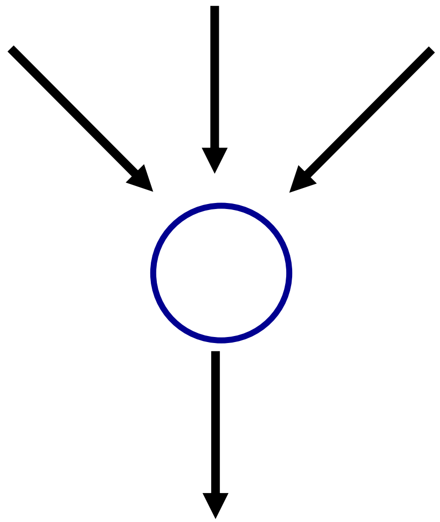


de-multiplexing

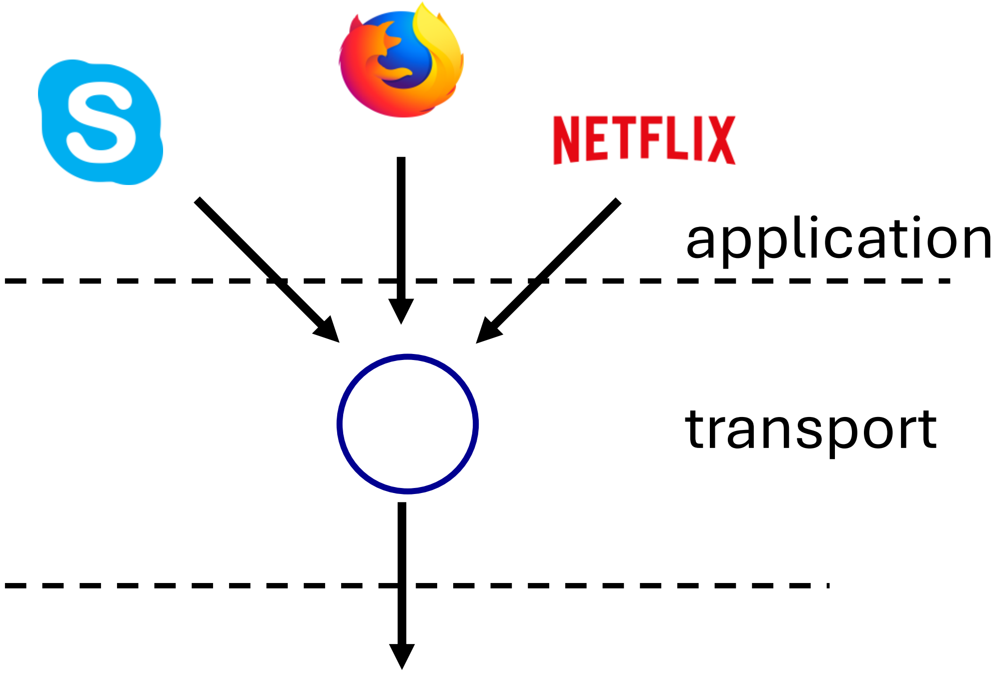


de-multiplexing

Demultiplexing



multiplexing

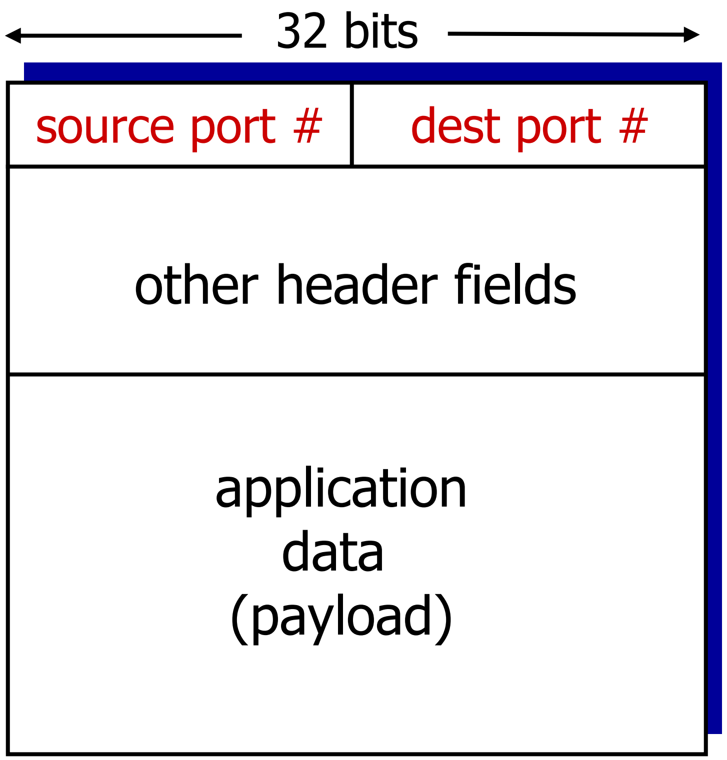


multiplexing


Multiplexing

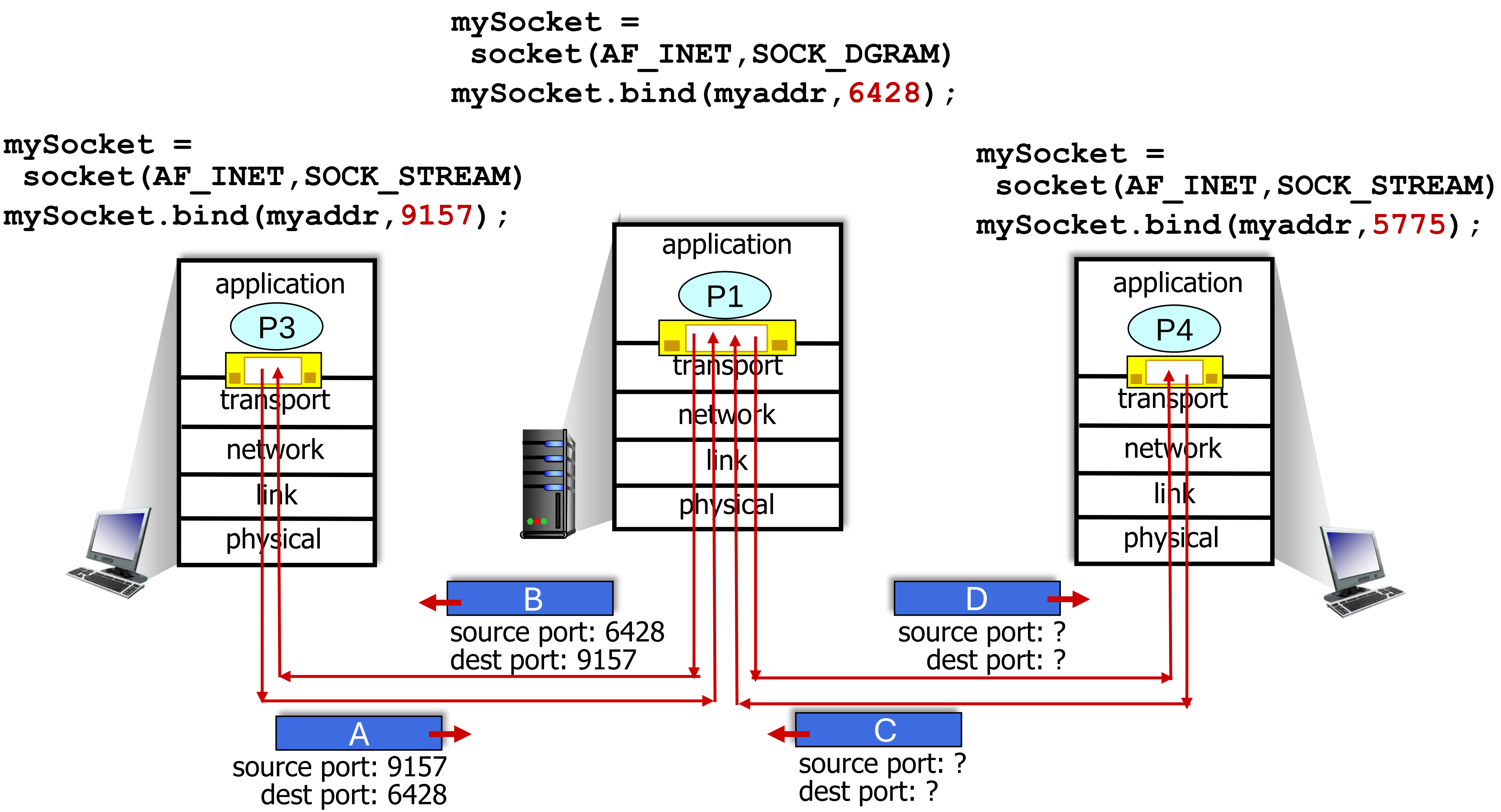
Multiplexing in Computer Networks

- Use IP & Port number.

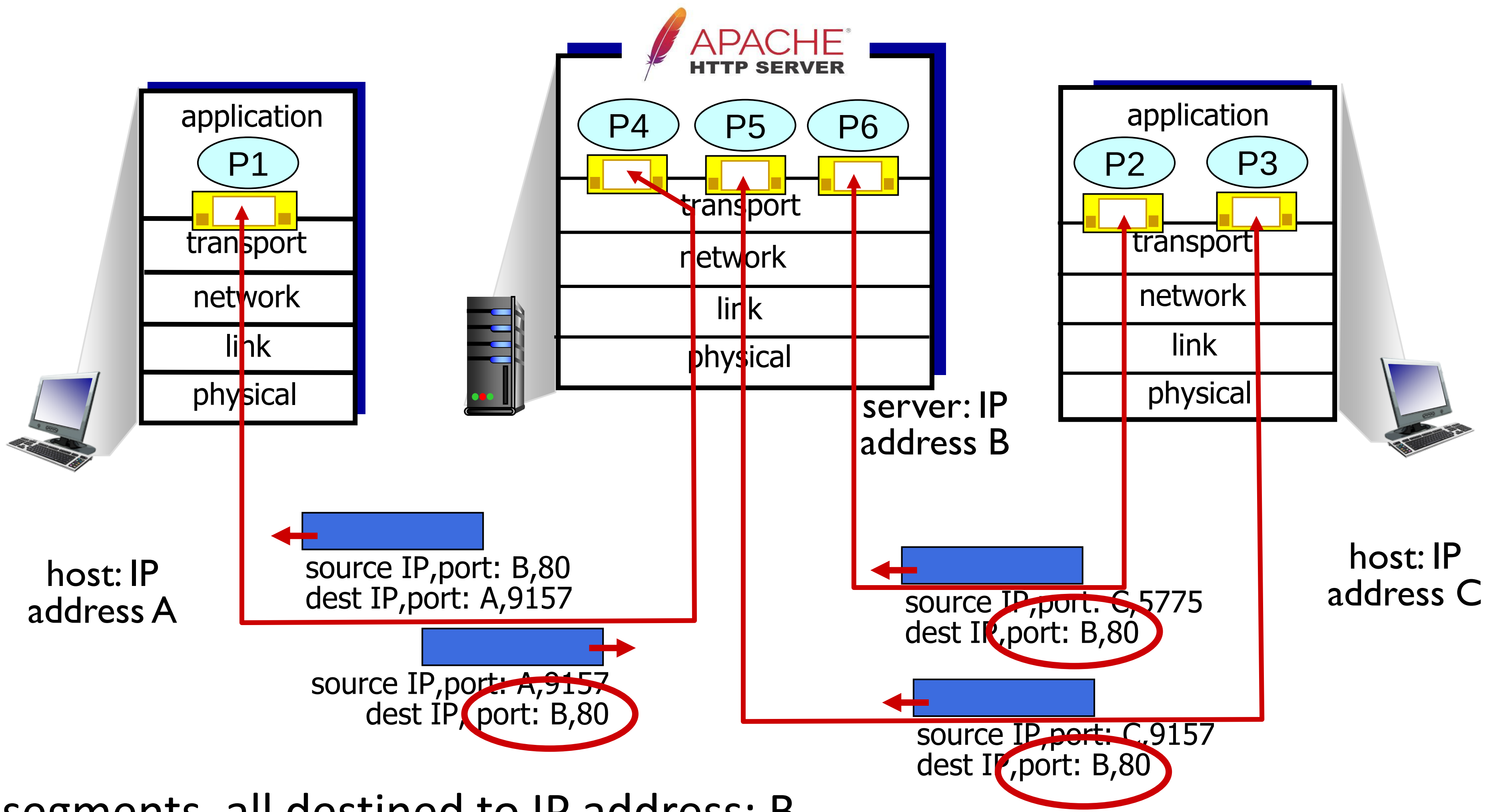


TCP/UDP segment format

Connectionless demultiplexing: an example



Connection-oriented demultiplexing: example



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

Summary

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

UDP: User Datagram Protocol

UDP

- *connectionless:*
  - no handshaking between UDP sender, receiver.
  - each UDP segment handled independently of others.
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive).
  - DNS
  - SNMP
  - HTTP/3

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768 J. Postel
ISI
28 August 1980

User Datagram Protocol

Introduction

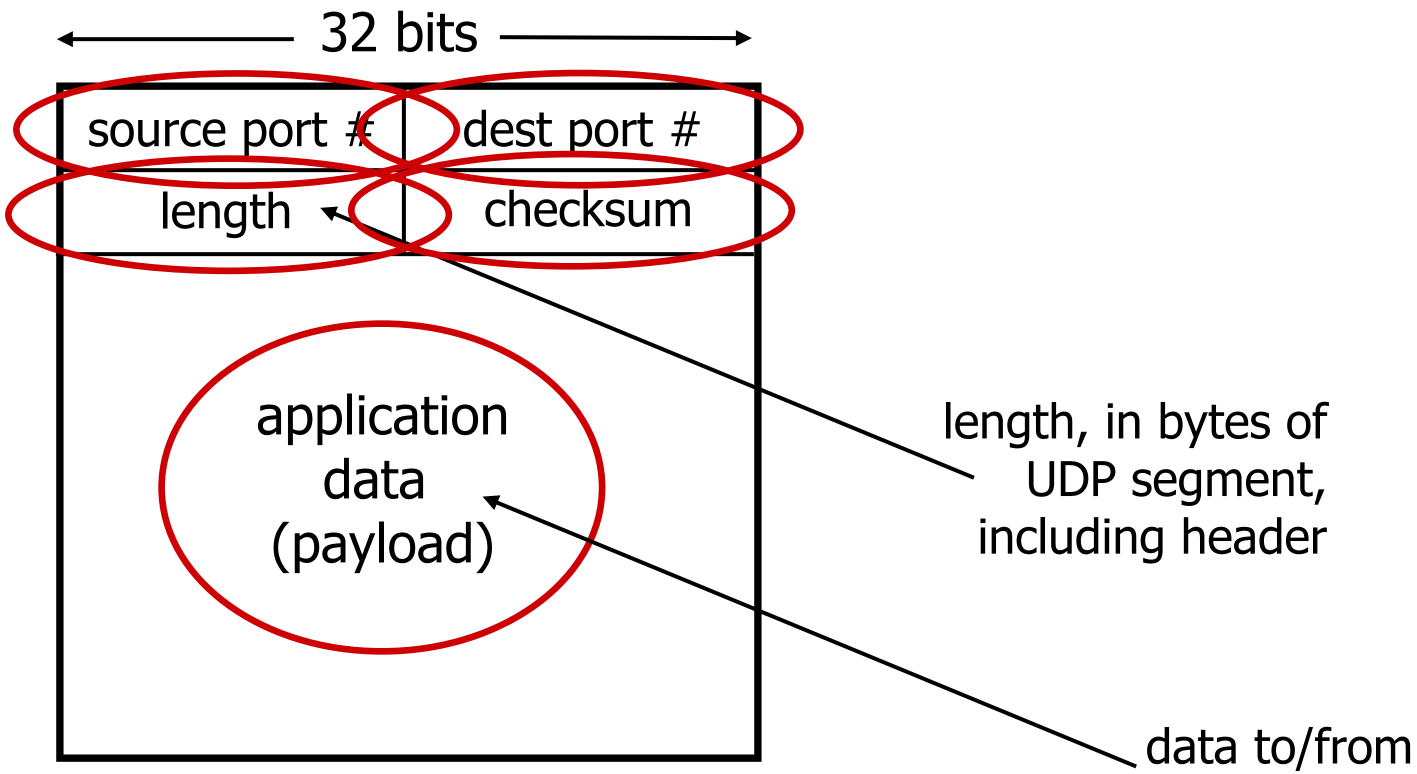
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

UDP segment header



UDP segment format

UDP checksum

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

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

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

Internet checksum: an example

example: add two 16-bit integers

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

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

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

Internet checksum: weak protection!

example: add two 16-bit integers

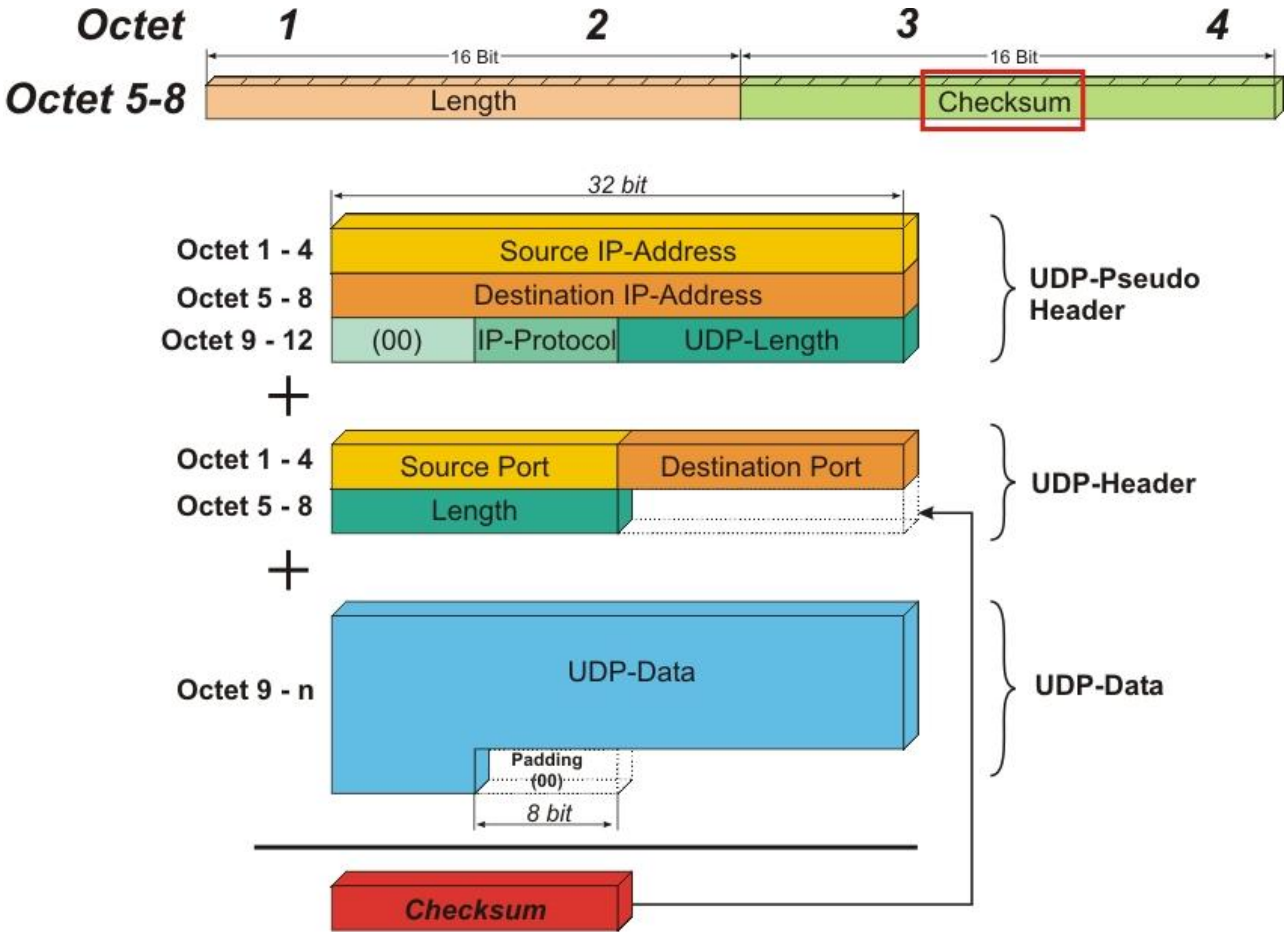
```
              1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0      0 1
              1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1      1 0

wraparound  1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

sum           1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum      0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

Even though numbers have changed (bit flips), *no* change in checksum!

UDP Checksum



UDP Checksum Error detection

- All 1 bit error detected.
- Not All 2 bit errors.
- Odd Number of Bit Errors detected (e.g., 1, 3, 5, etc.)
- Some combinations of bit flips may also go undetected.

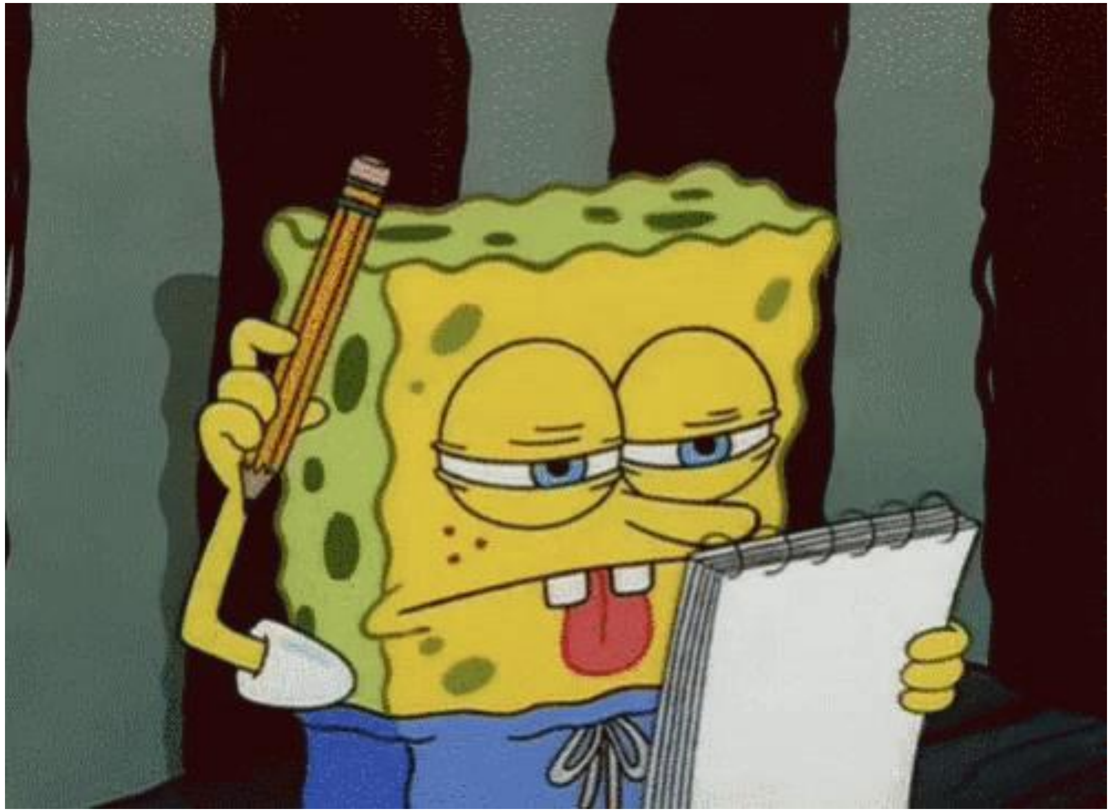
Reliable Data Transfer

Reliable Data Transfer

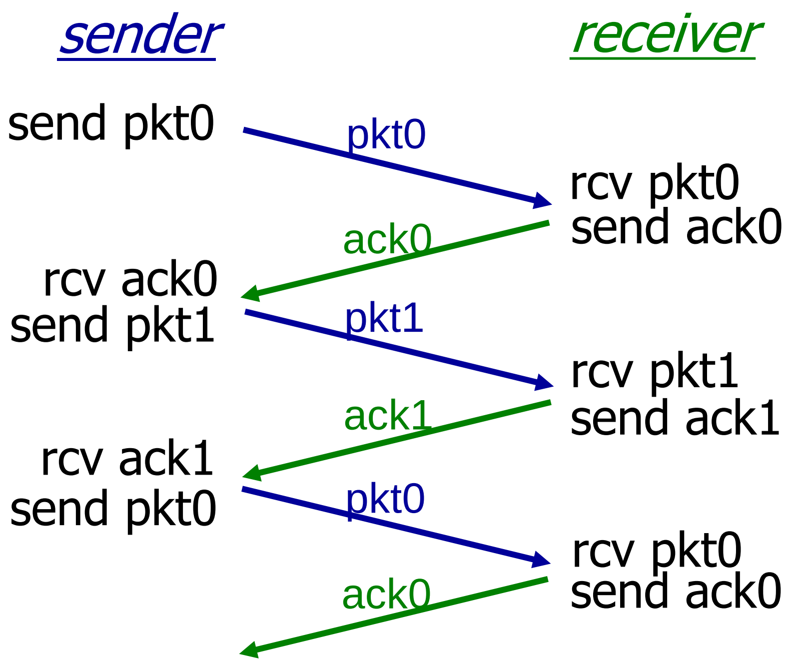
- There is Packet Loss and Bit Error in packet transfer.
- How can guarantee receive data?
- Let’s Think about it!

Acknowledge - ACK

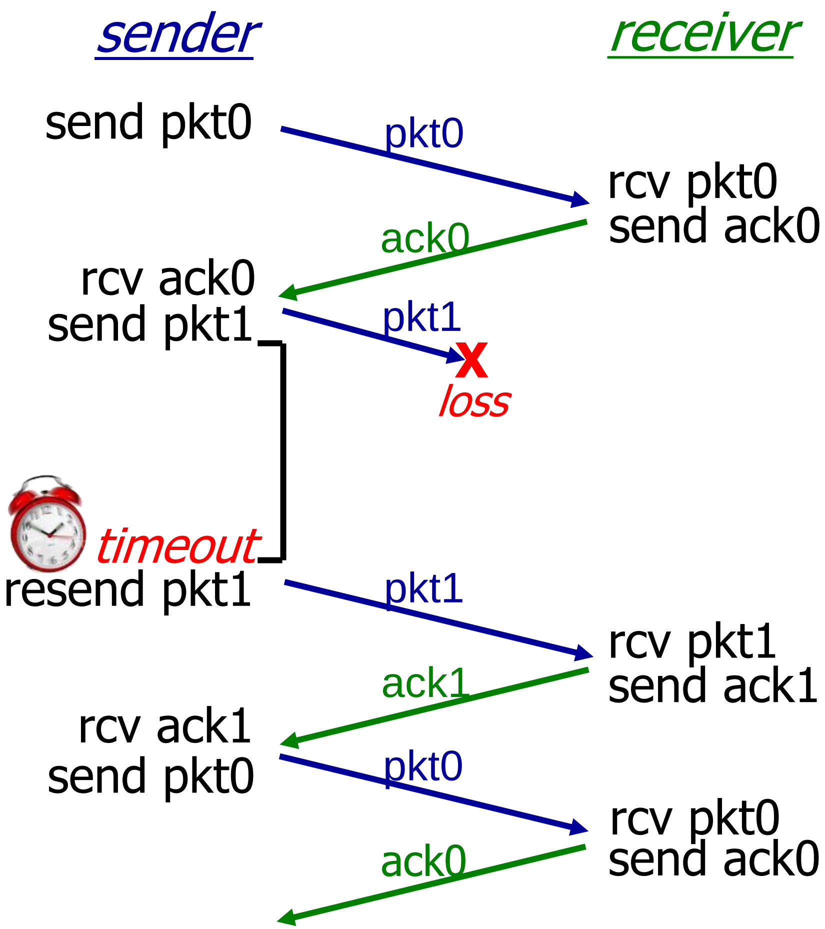
- Real Examples
  - Text Message delivery report.
  - Text me whenever you get home.
- This action called Acknowledge – ACK.
- What if loss or error in ACK?
- Send an Ack2 for Ack?
- What if loss or error in ACK2?



In action

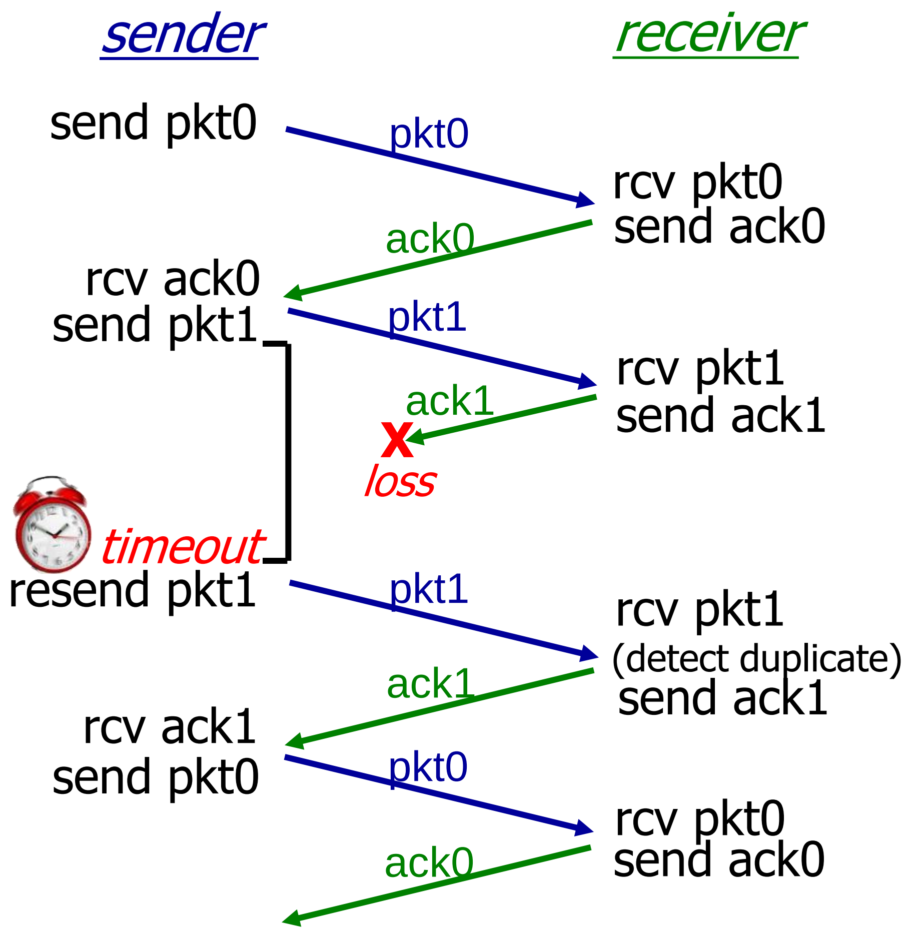


(a) no loss

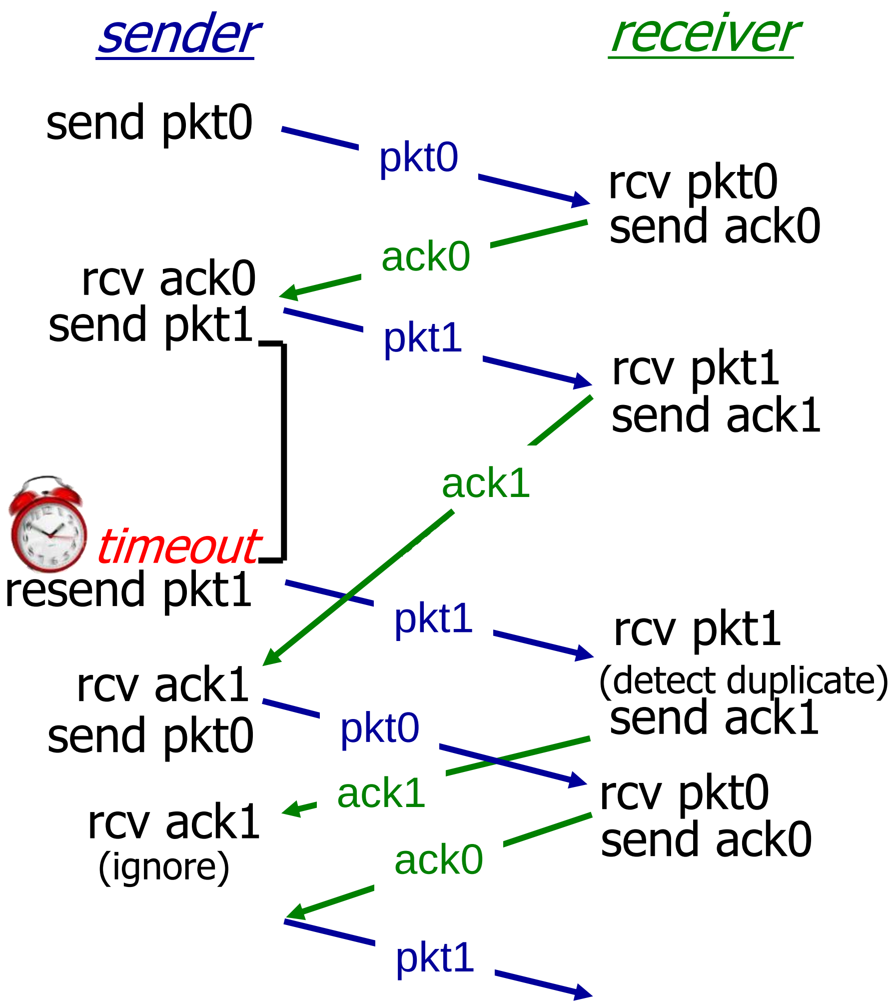


(b) packet loss

rdt3.0 in action

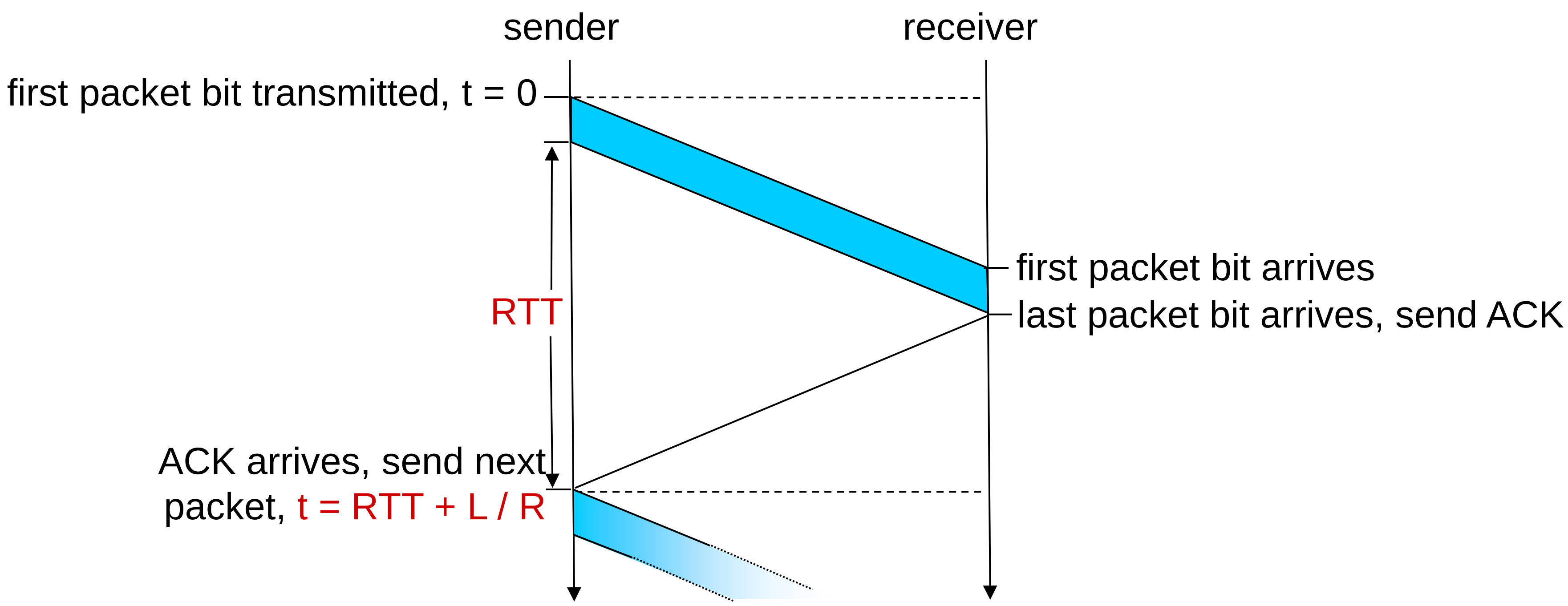


(c) ACK loss

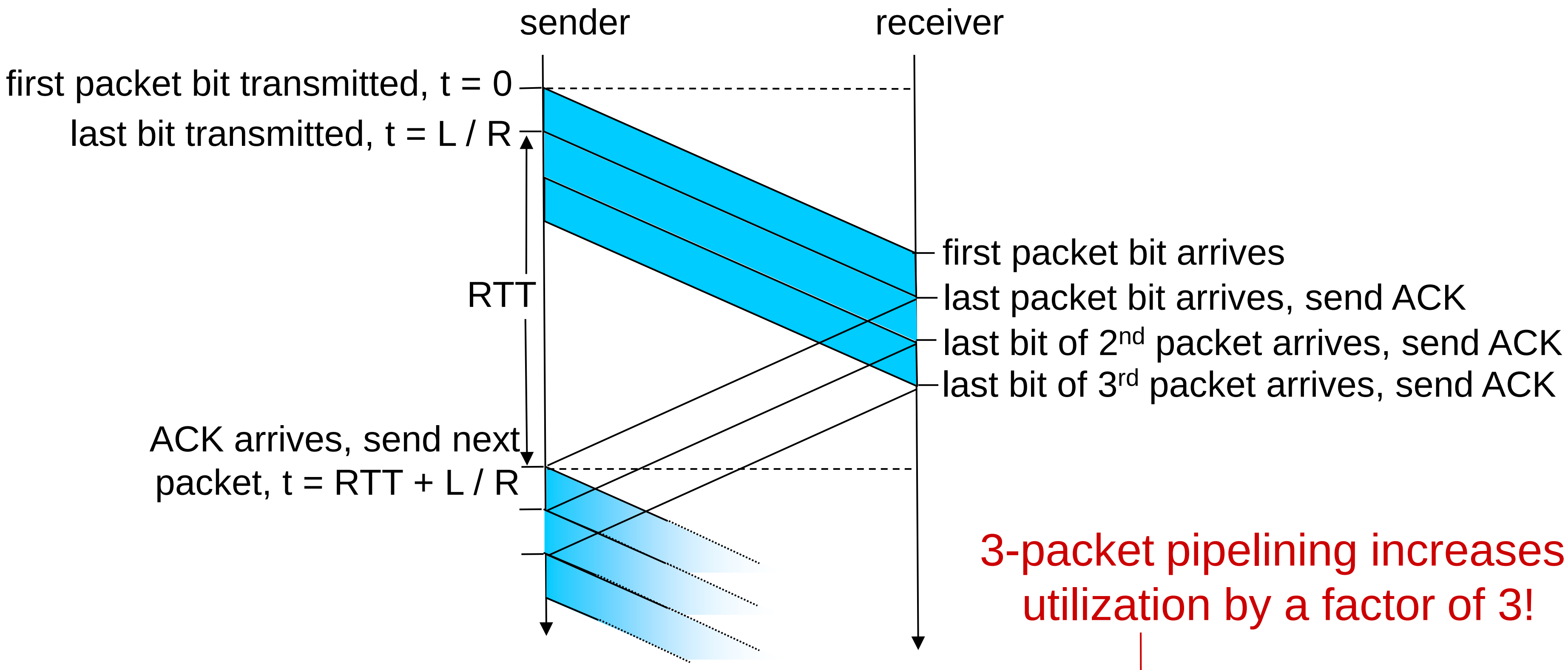


(d) premature timeout/ delayed ACK

Stop-and-wait operation



Pipelining: increased utilization

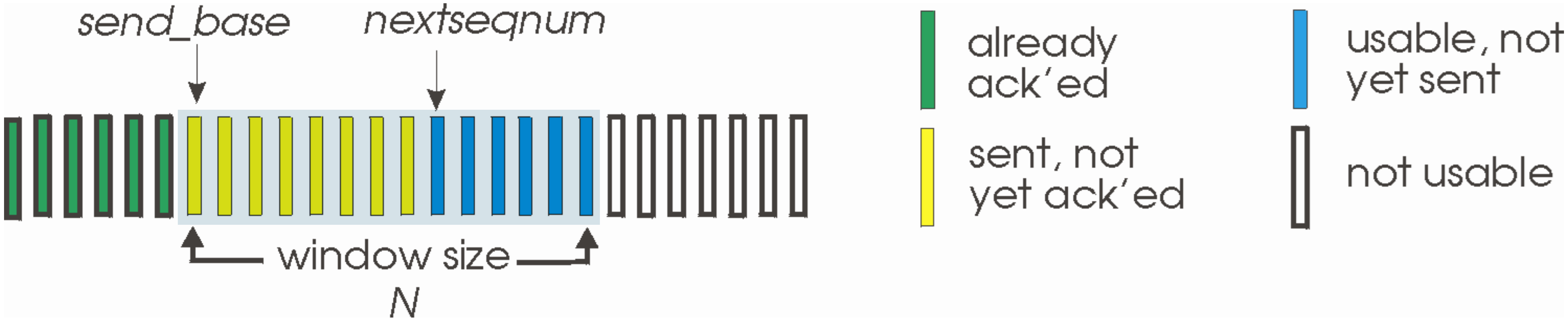


3-packet pipelining increases utilization by a factor of 3!

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

Go-Back-N: sender

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

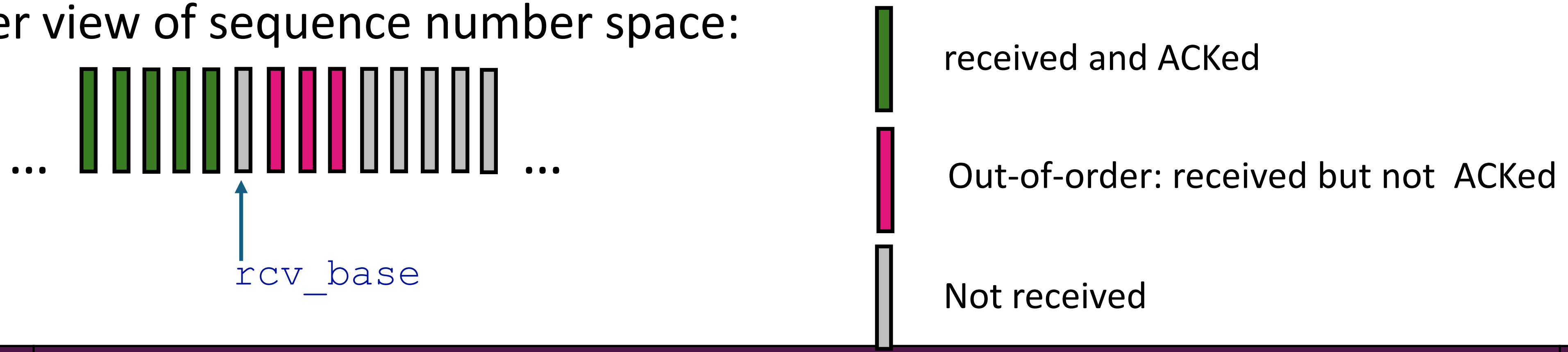


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

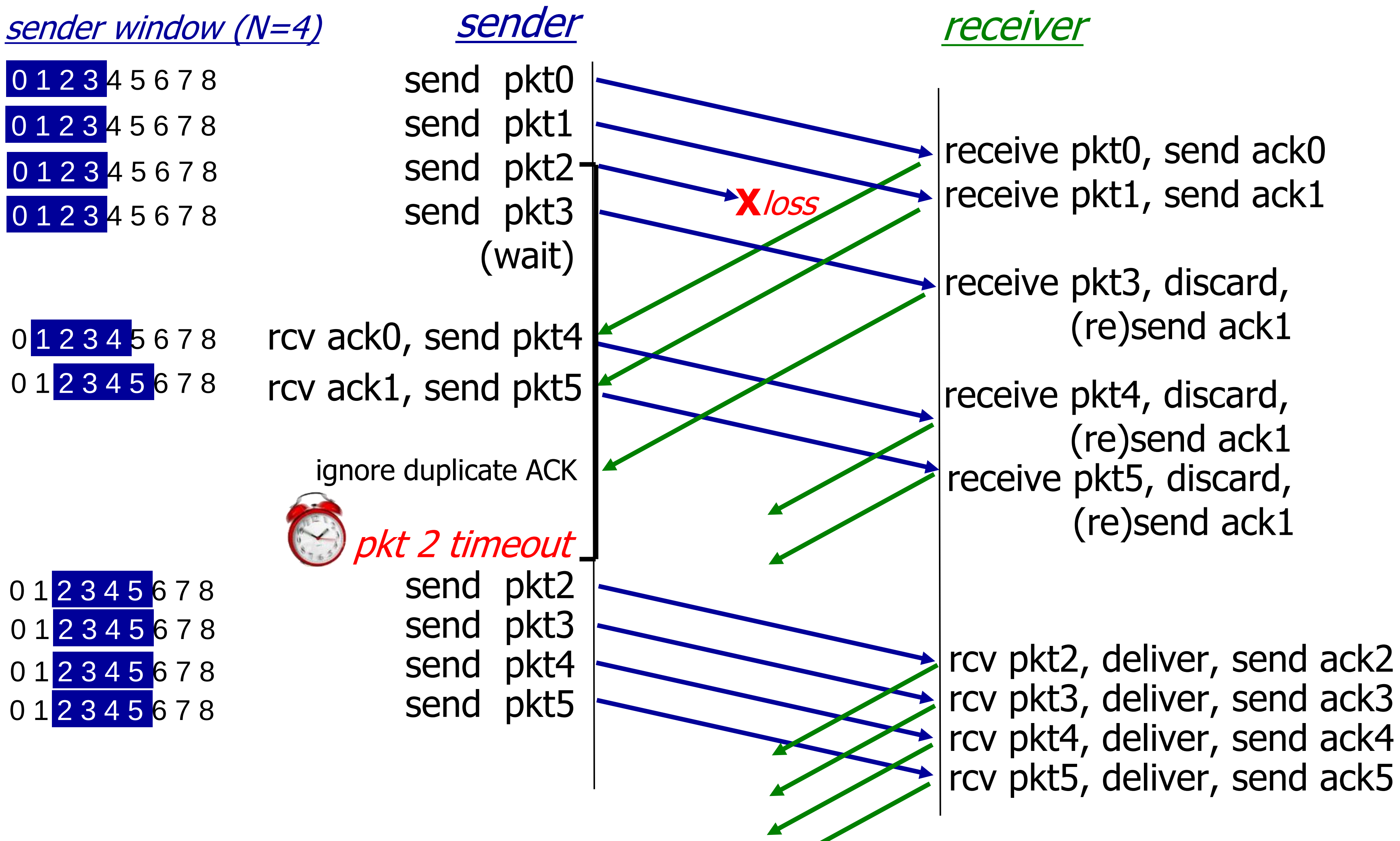
Go-Back-N: receiver

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

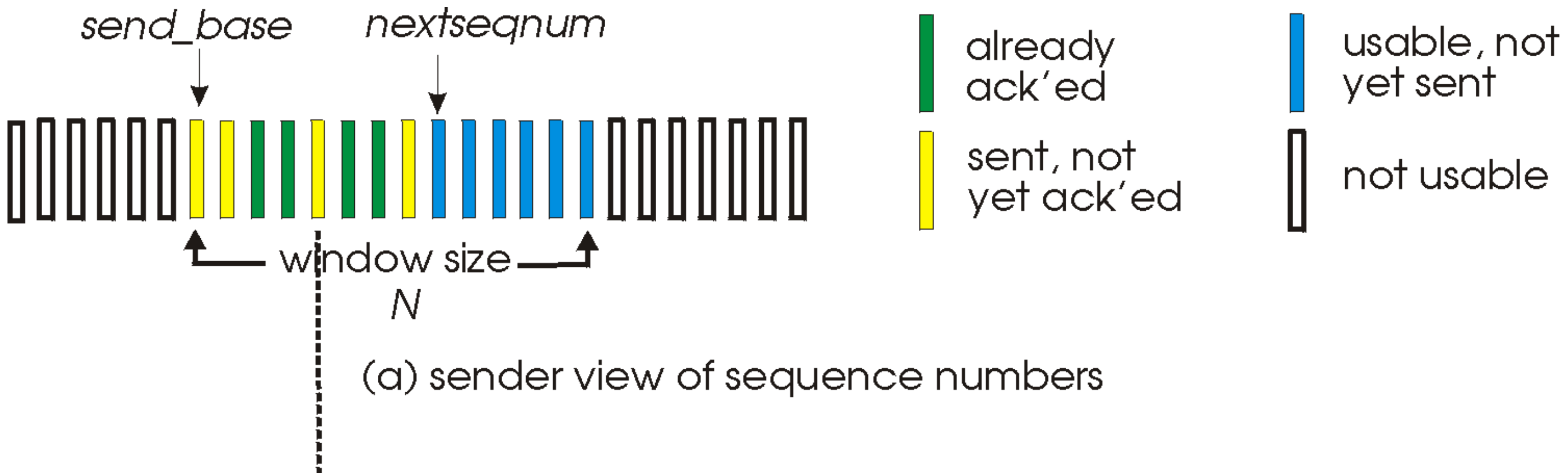
Receiver view of sequence number space:



Go-Back-N in action



Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N-1]:

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

receiver

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

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

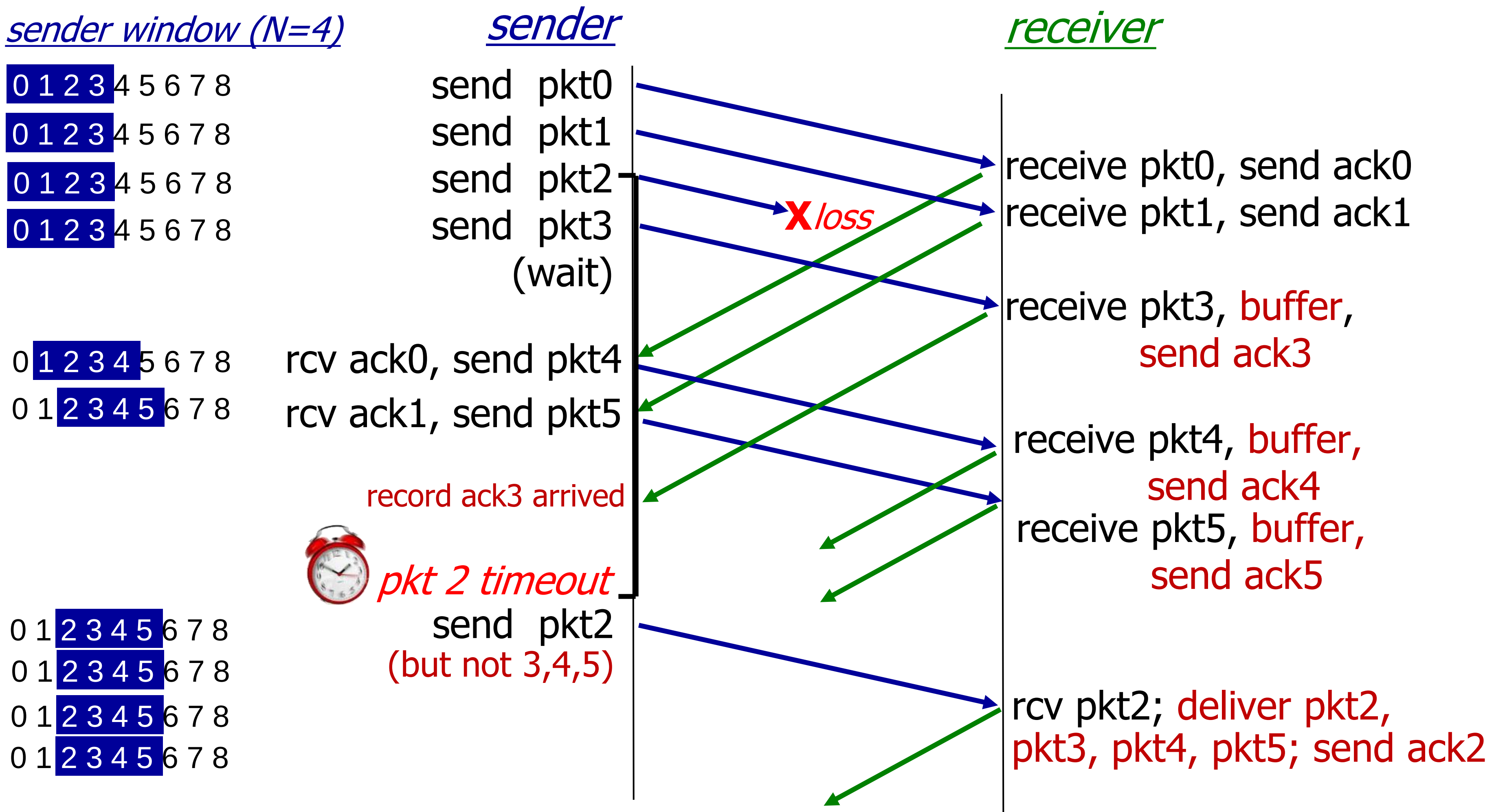
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action

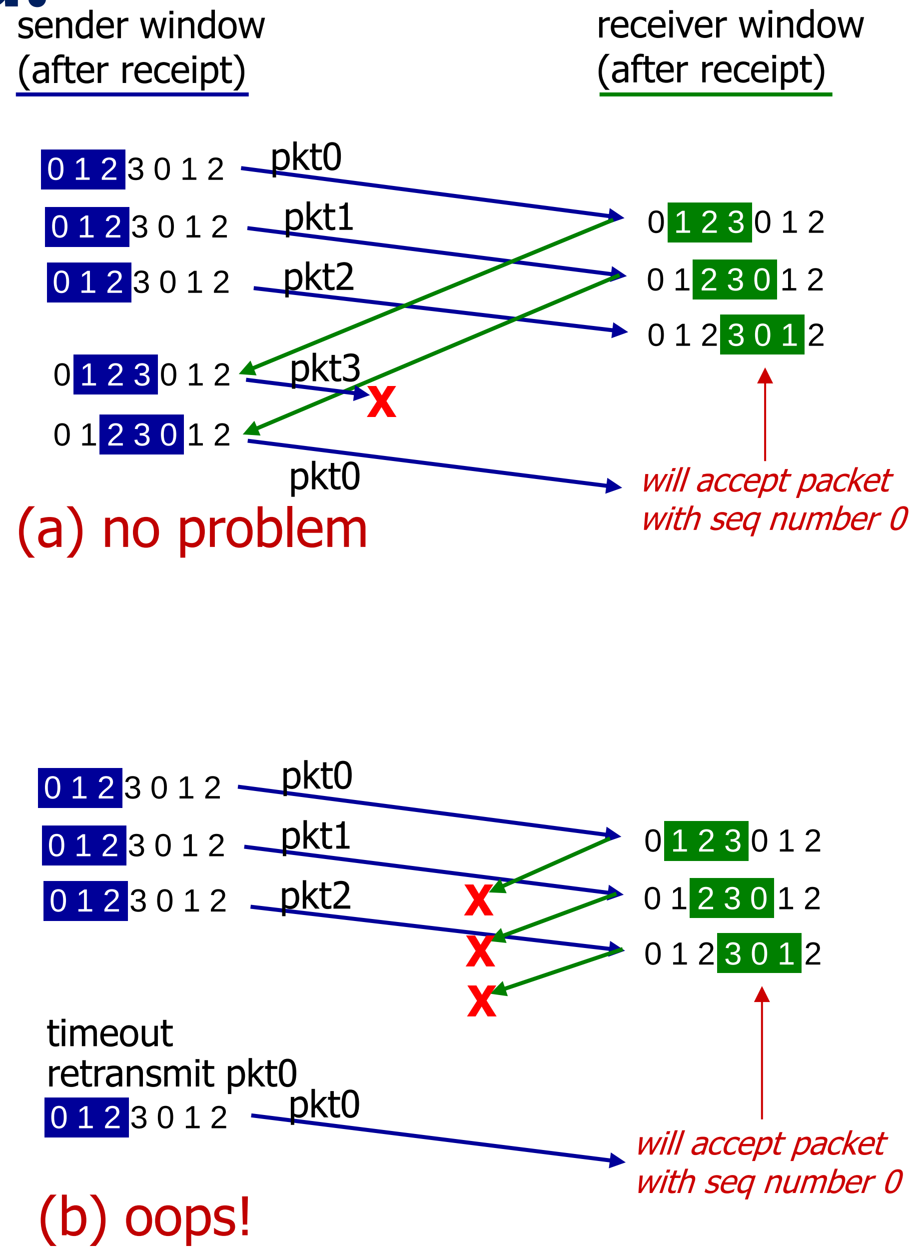


Q: what happens when ack2 arrives?

Selective repeat: a dilemma!

example:

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



Selective repeat: a dilemma!

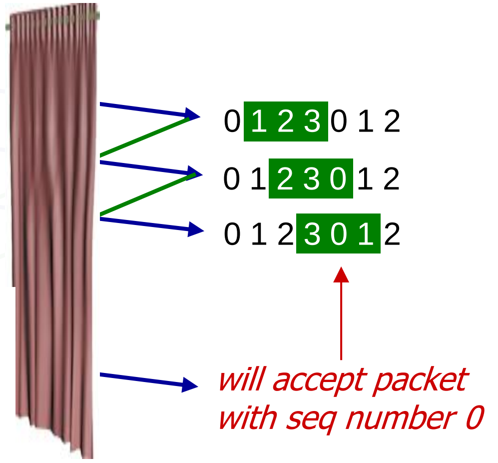
example:

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

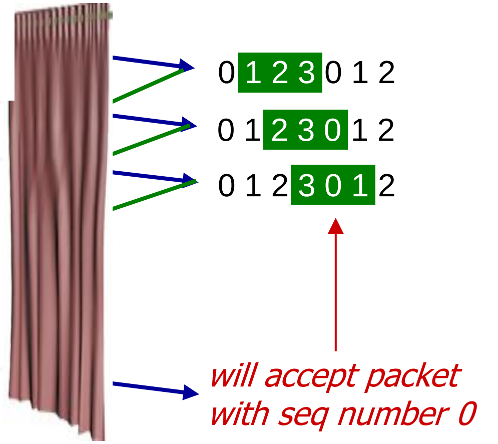
Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

sender window (after receipt)

receiver window (after receipt)



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

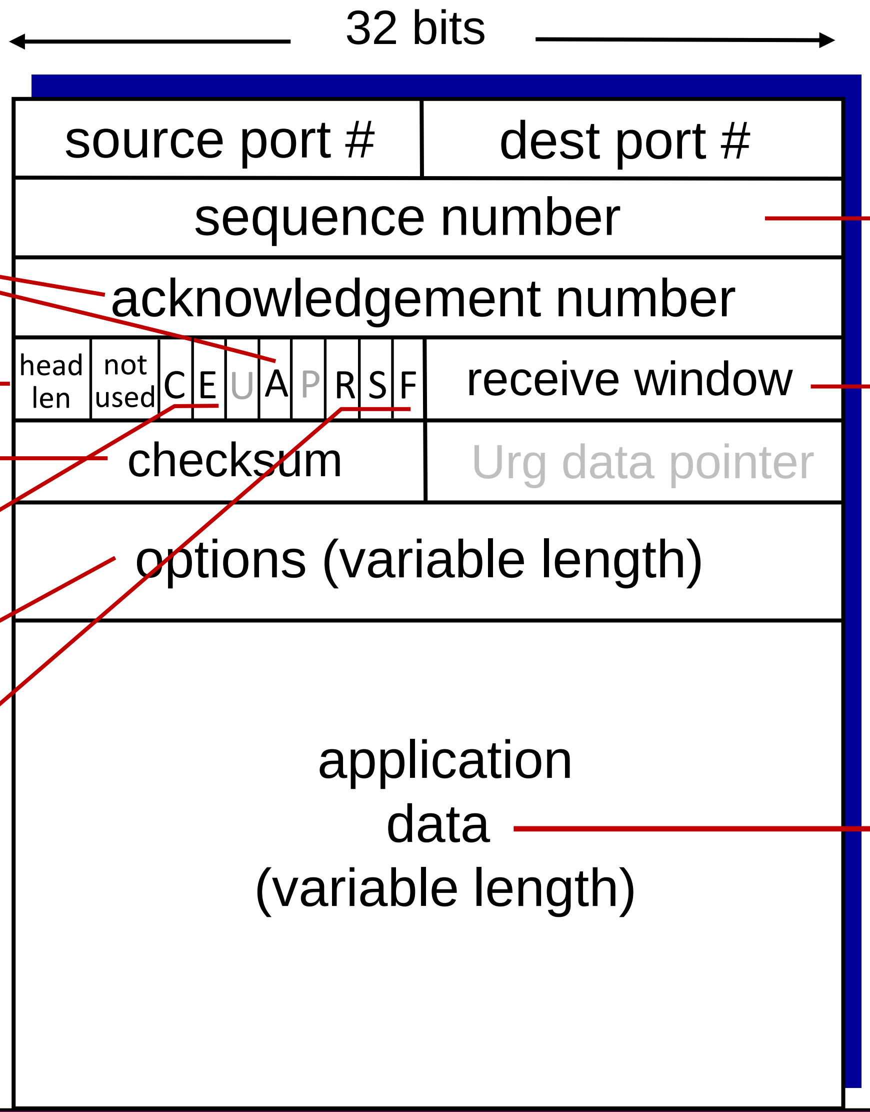


TCP: Transmission Control Protocol

TCP

- RFCs: 793,1122, 2018, 5681, 7323
  - connection-oriented
  - flow controlled: sender will not overwhelm receiver
  - point-to-point: one sender, one receiver
  - reliable, in-order *byte steam:* no “message boundaries"
  - full duplex data:
    - bi-directional data flow in same connection
    - MSS: maximum segment size

TCP segment structure



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

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

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

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

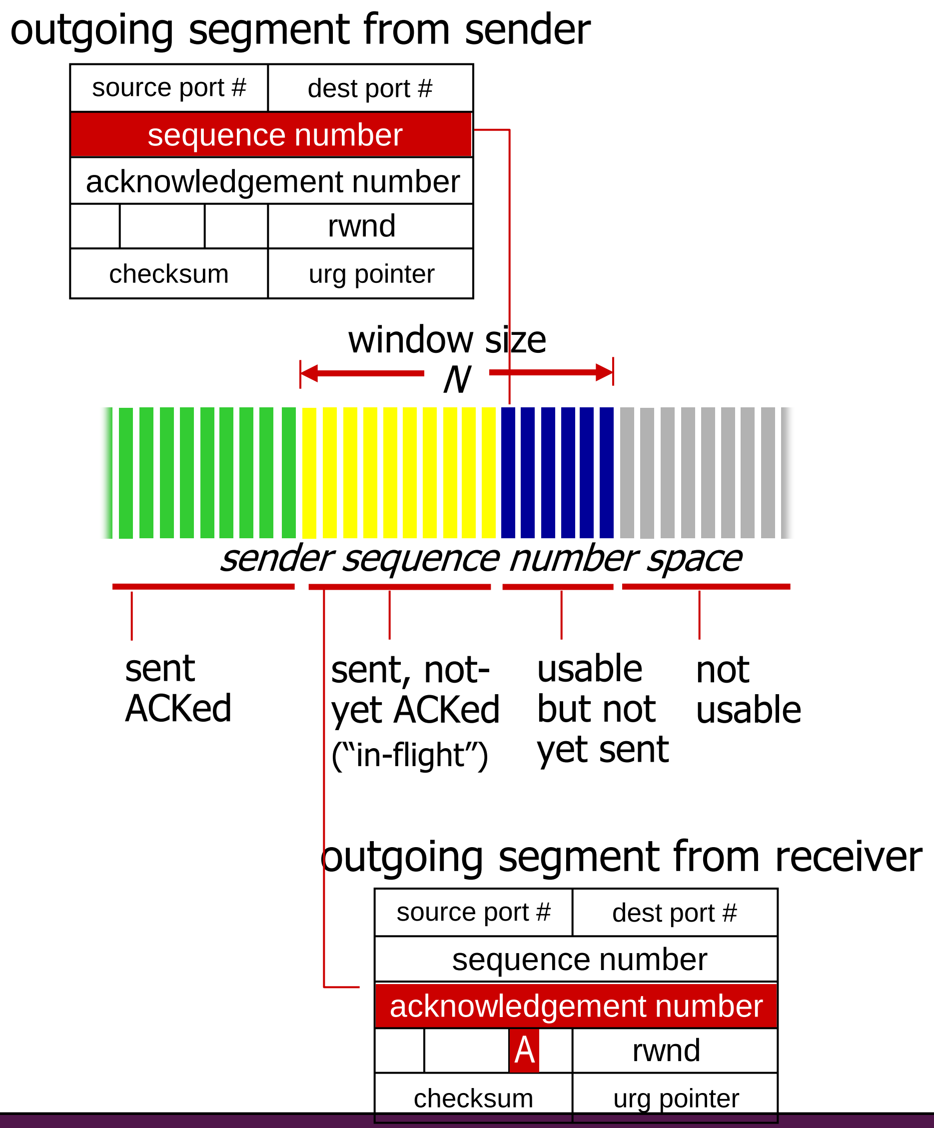
- byte stream "number" of first byte in segment's data

Acknowledgements:

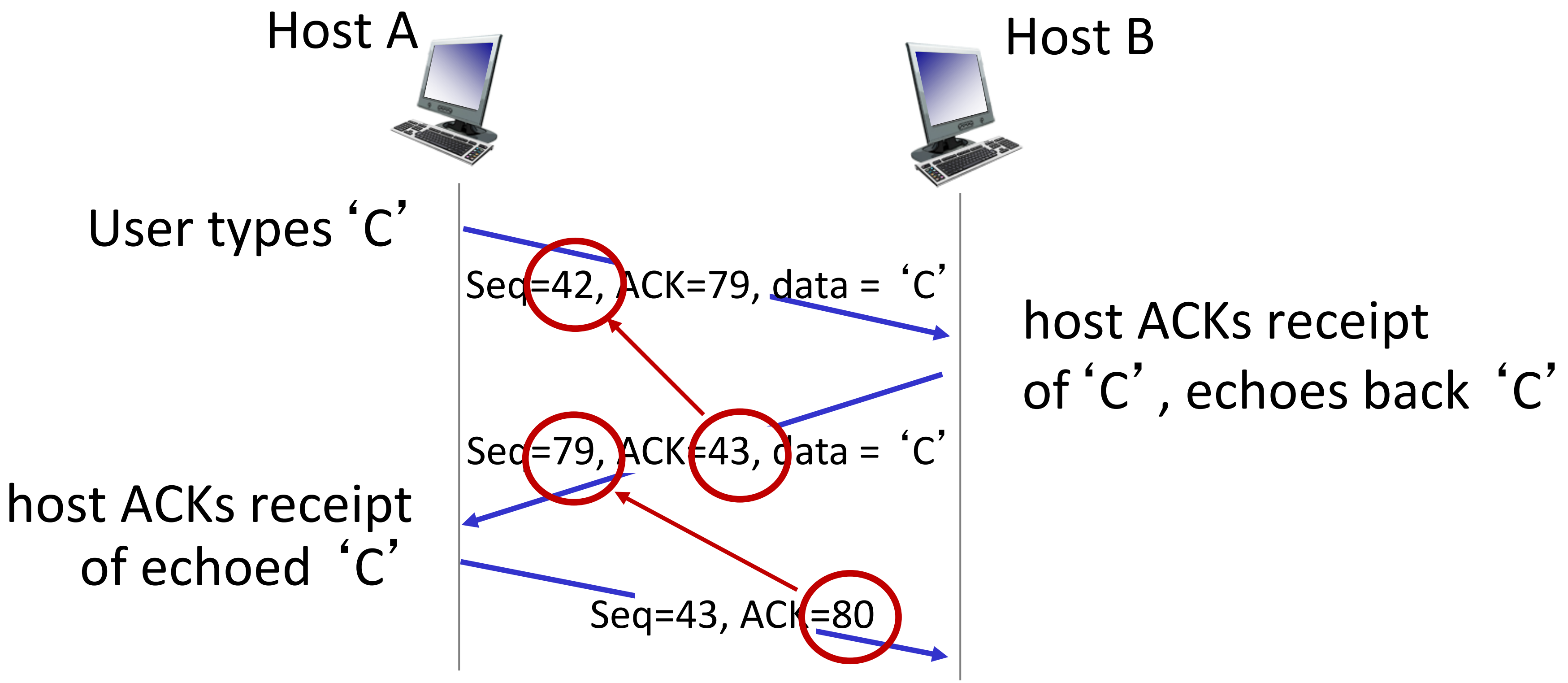
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs



simple telnet scenario

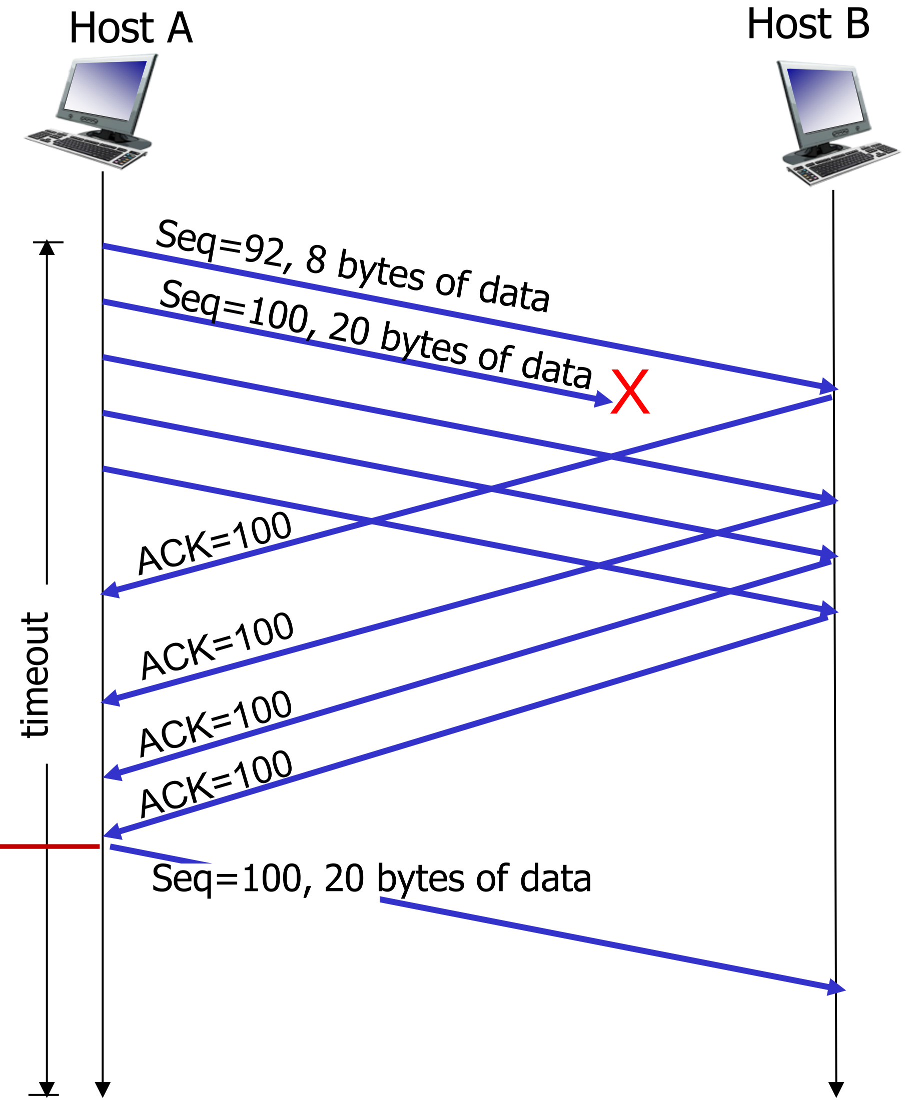
TCP fast retransmit

TCP fast retransmit

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

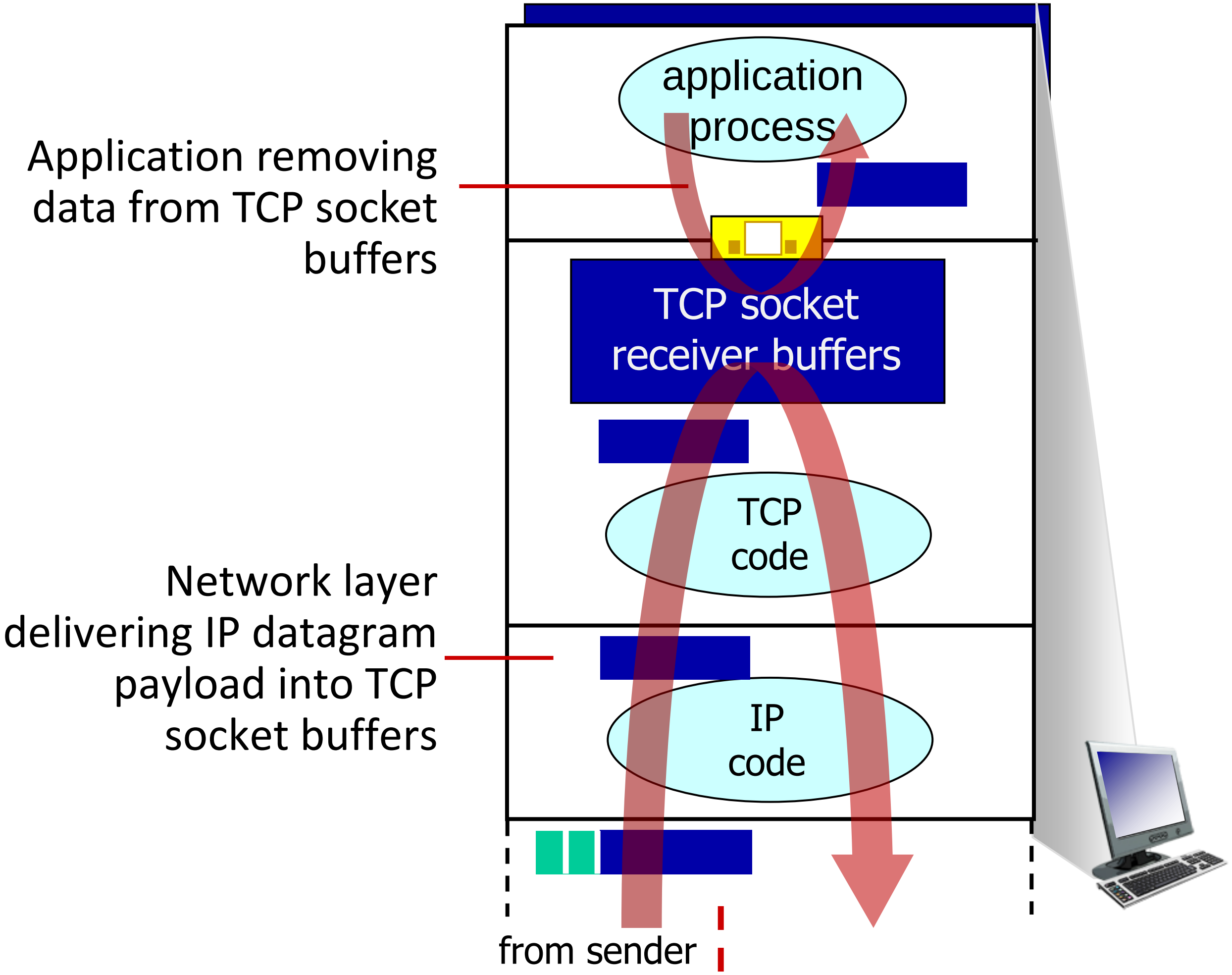
- likely that unACKed segment lost, so don’t wait for timeout

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



TCP flow control

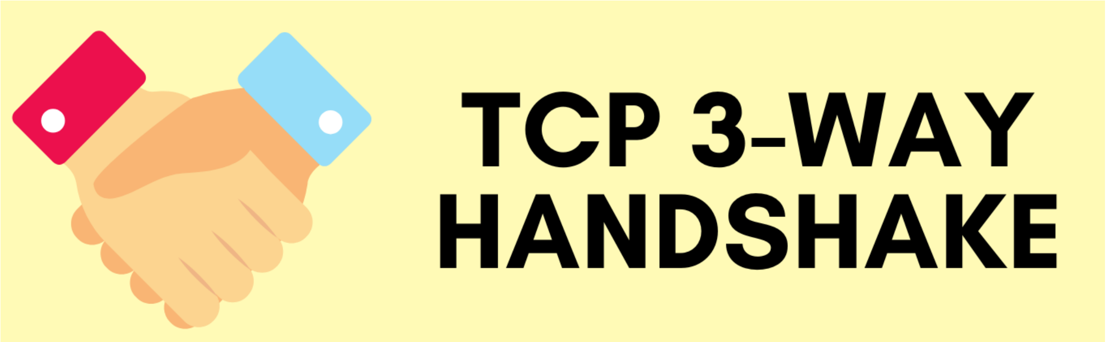
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

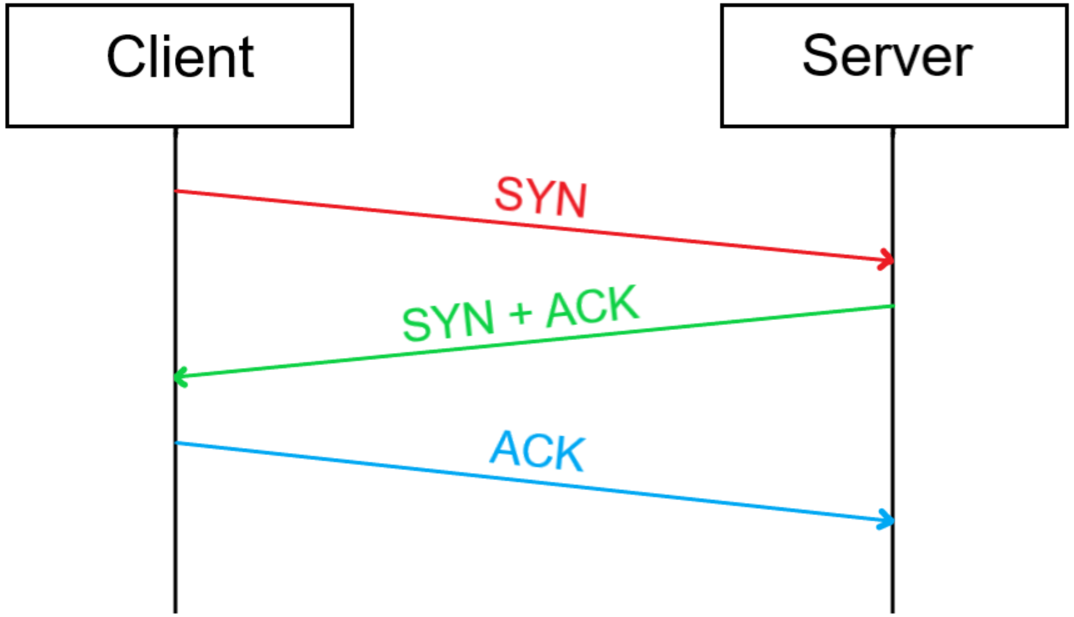


receiver protocol stack

TCP Connection establishment

- Why 3 way hand shake not 2 way?

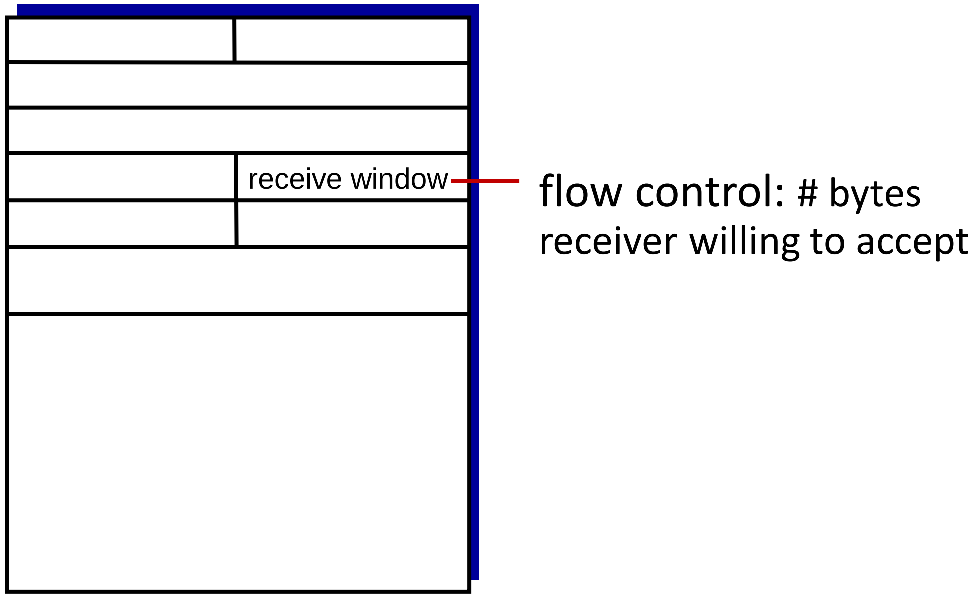


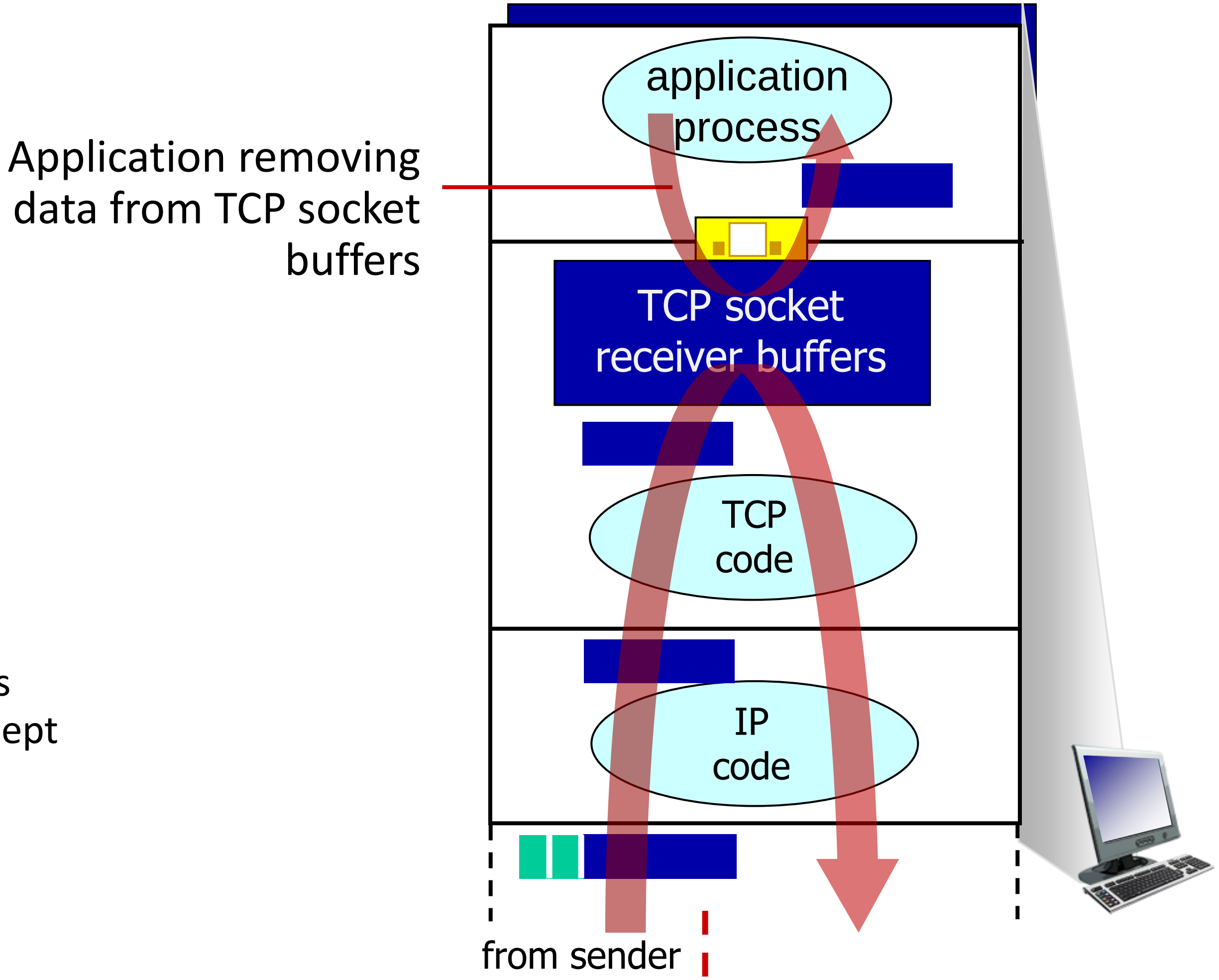


3-Way TCP Handshake

TCP flow control

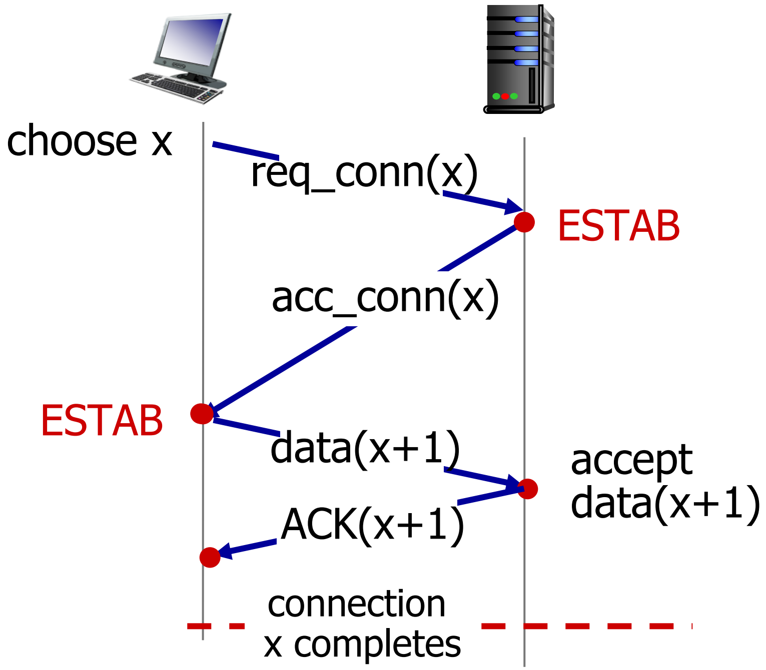
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?





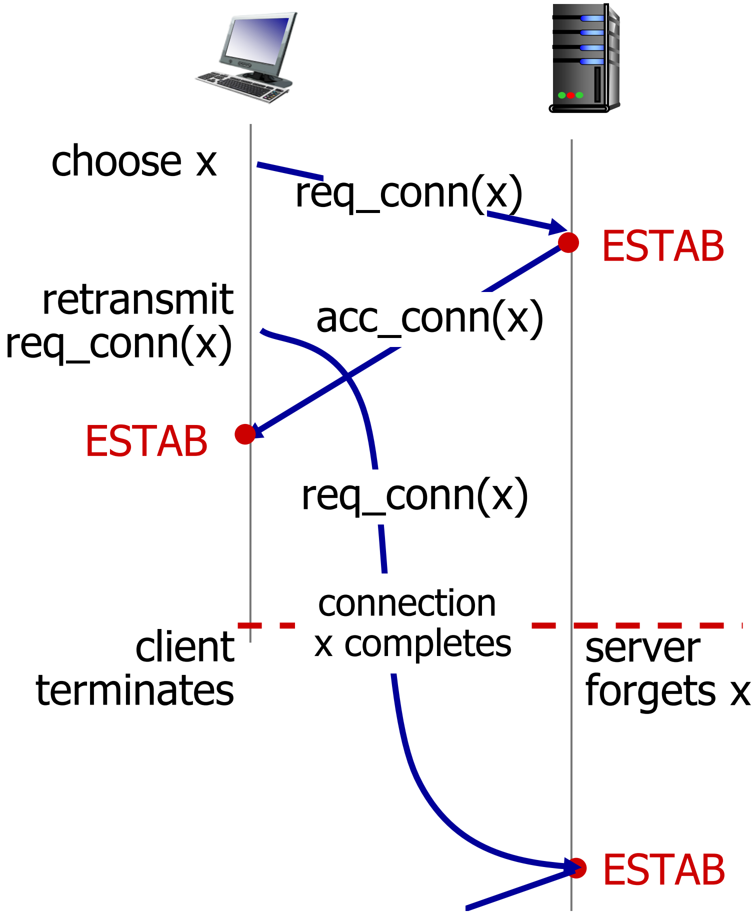
receiver protocol stack

2-way handshake scenarios



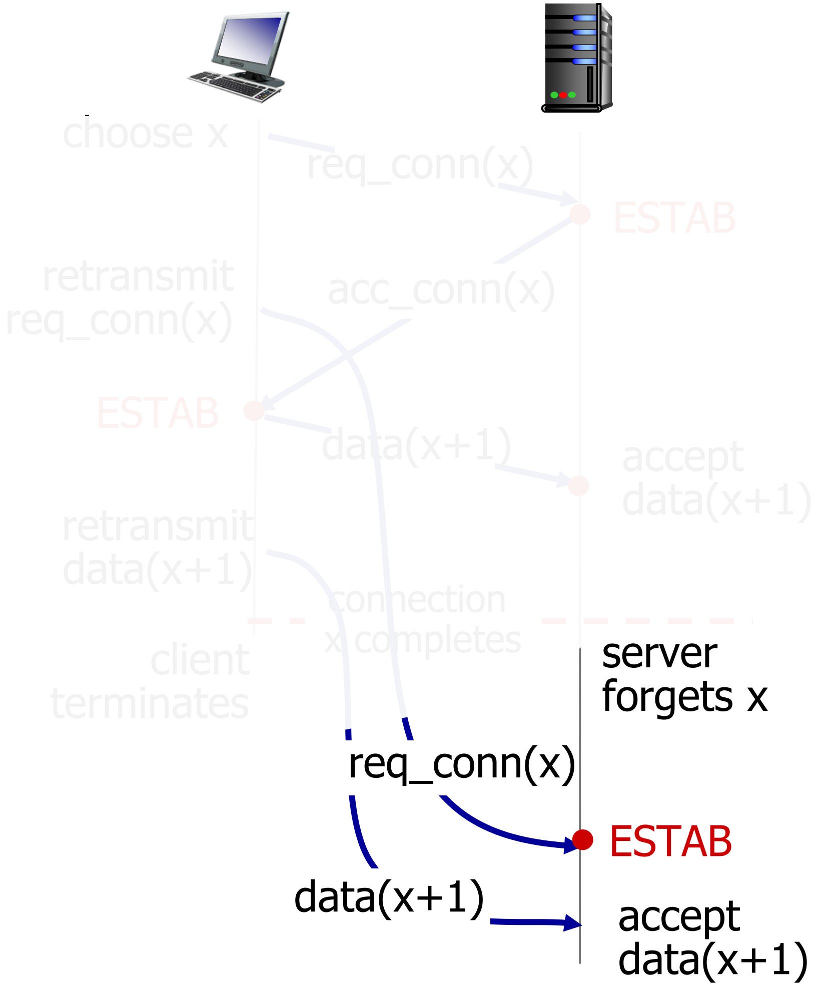
No problem!

2-way handshake scenarios



Problem: half open connection! (no client)

2-way handshake scenarios



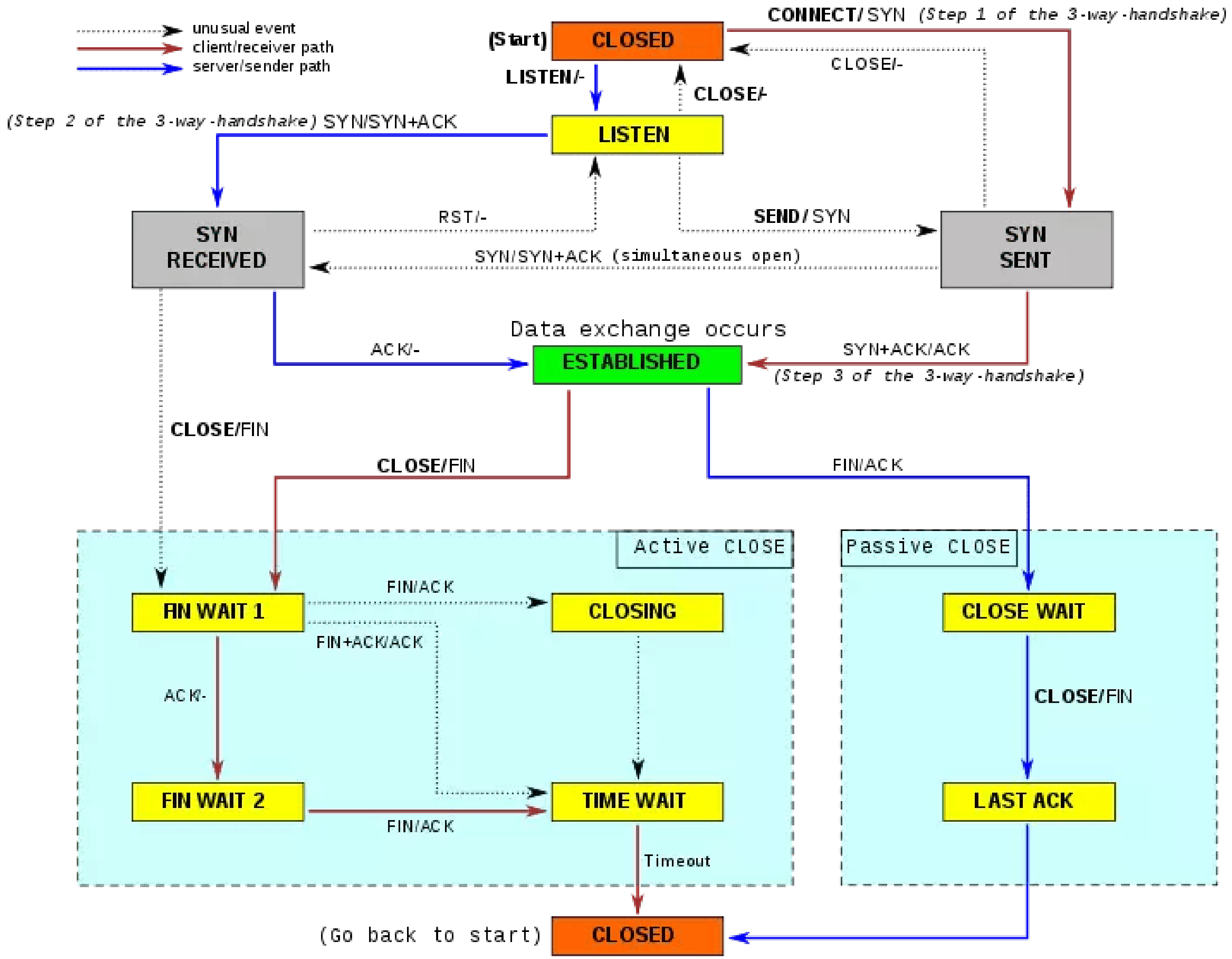
Problem: dup data accepted!

A human 3-way handshake protocol

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

Full TCP state diagram



Congestion Control

TCP congestion control: AIMD

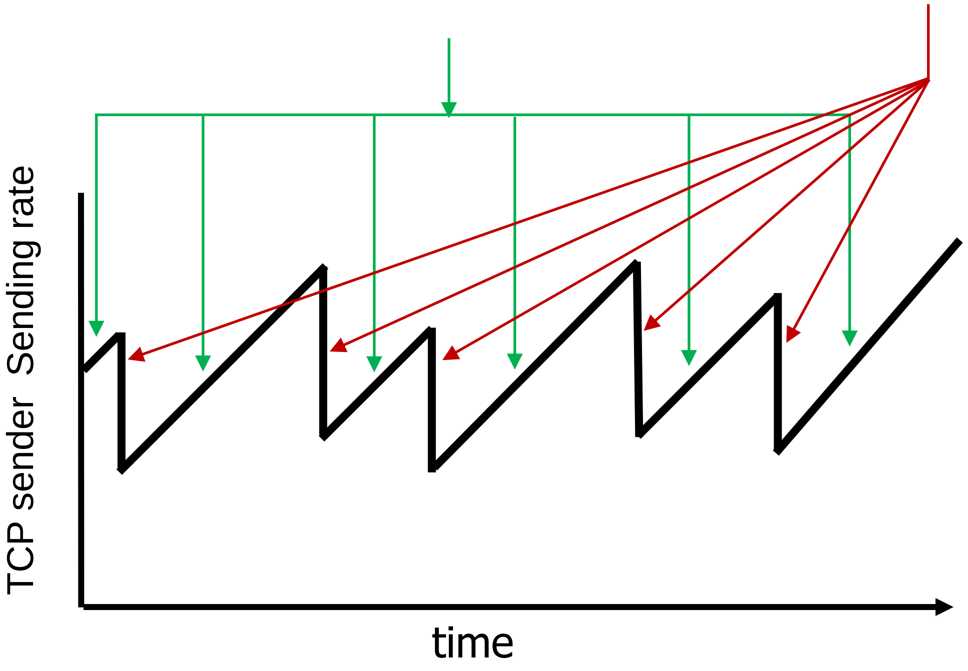
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

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

Multiplicative Decrease

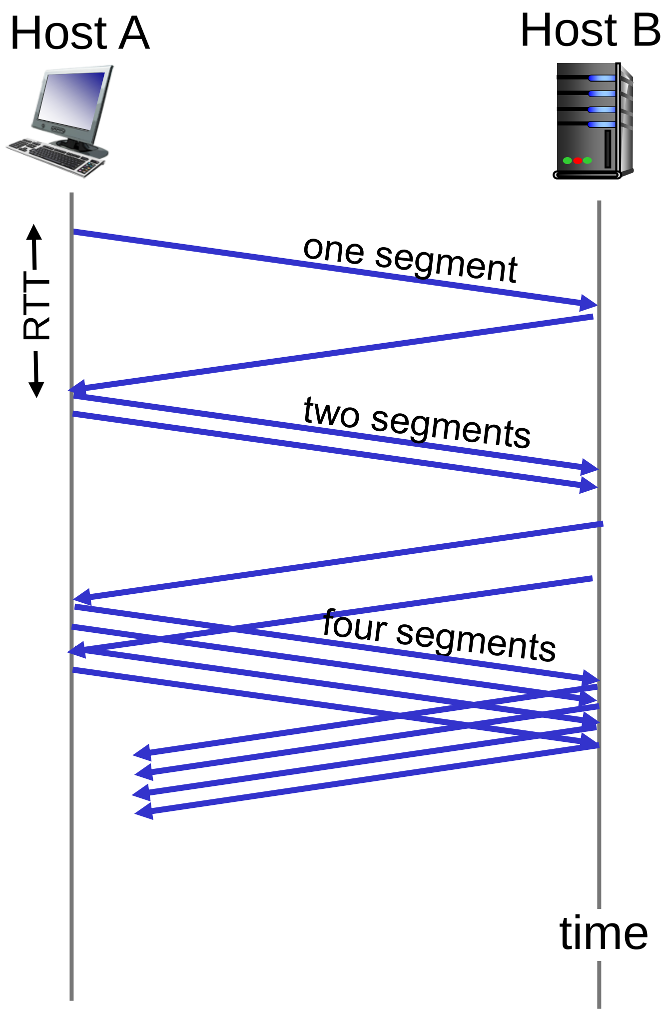
cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP slow start

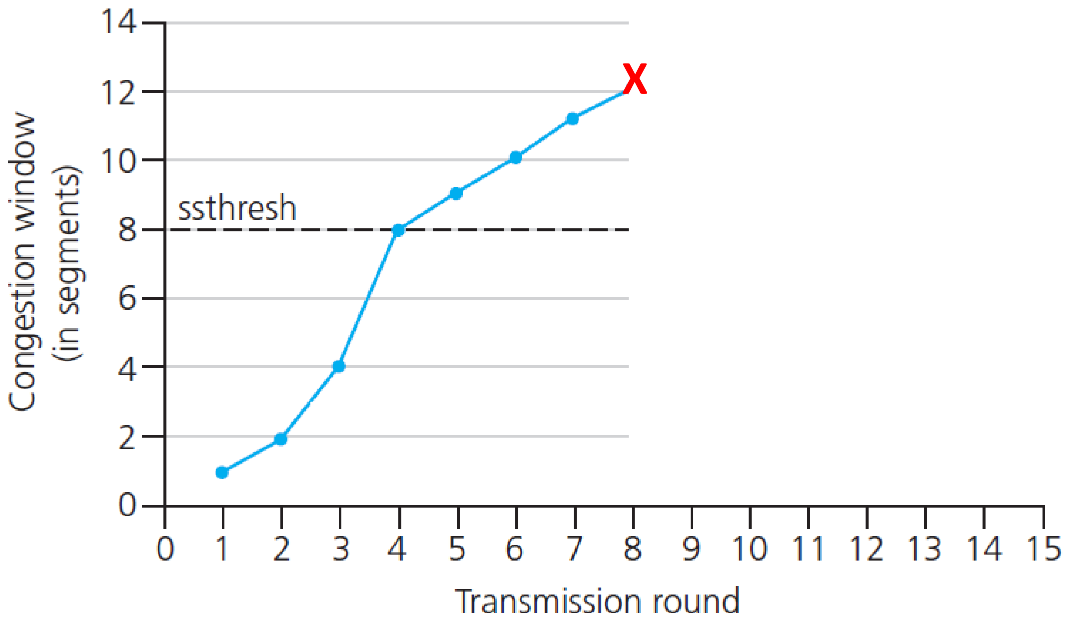
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

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

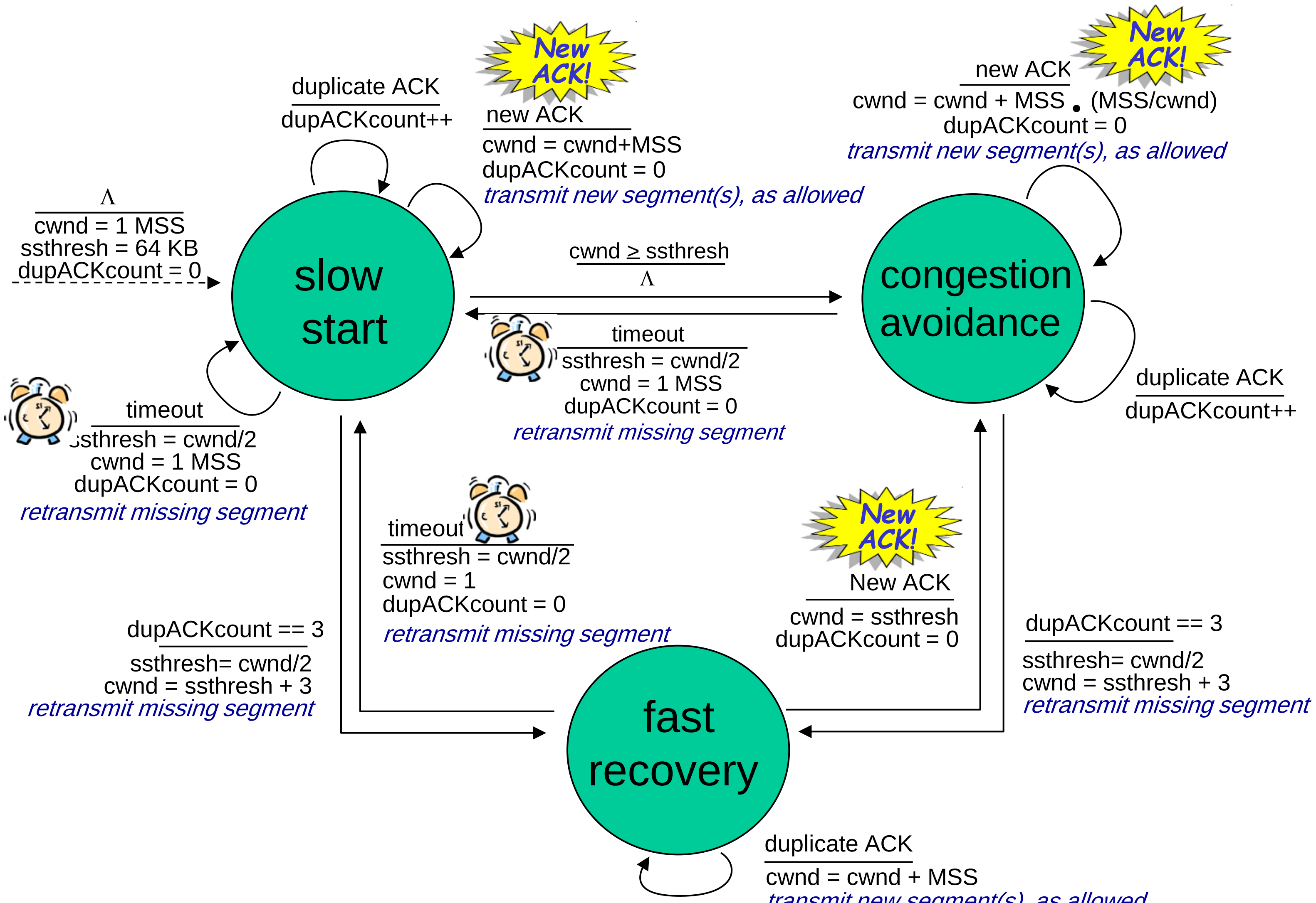


Implementation:

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

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

Summary: TCP congestion control



TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tunable
- 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