CSC/CPE 138

COMPUTER NETWORK FUNDAMENTALS

Lecture 3_1: Transport Layer

California State University, Sacramento
Fall 2024

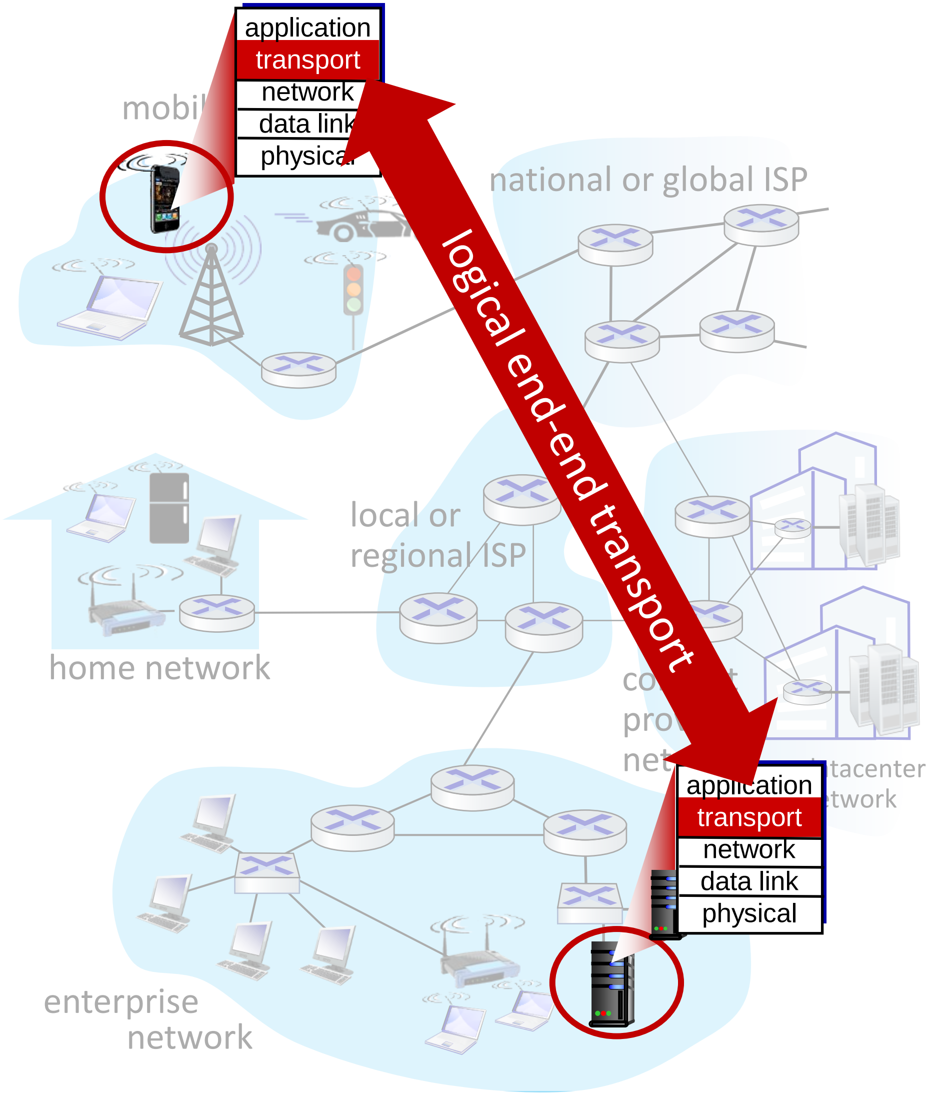
Slide Courtesy: Computer Networking: A Top-Down Approach, Kurose Ross, 8th Edition

Overview

- Understand the principles behind transport layer services:
  - Multiplexing, demultiplexing
  - UDP transport protocol
  - Reliable data transfer
  - Pipelining

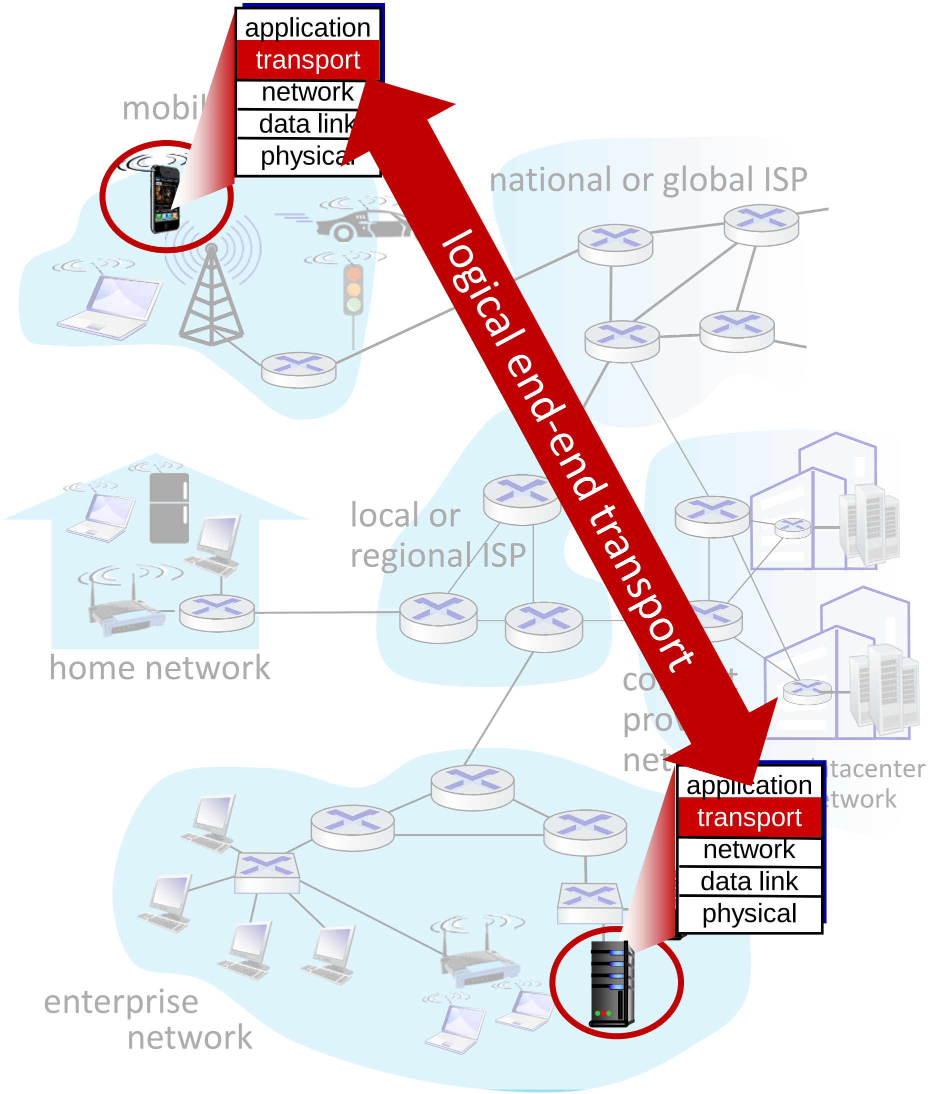
Transport services and protocols

- 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



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



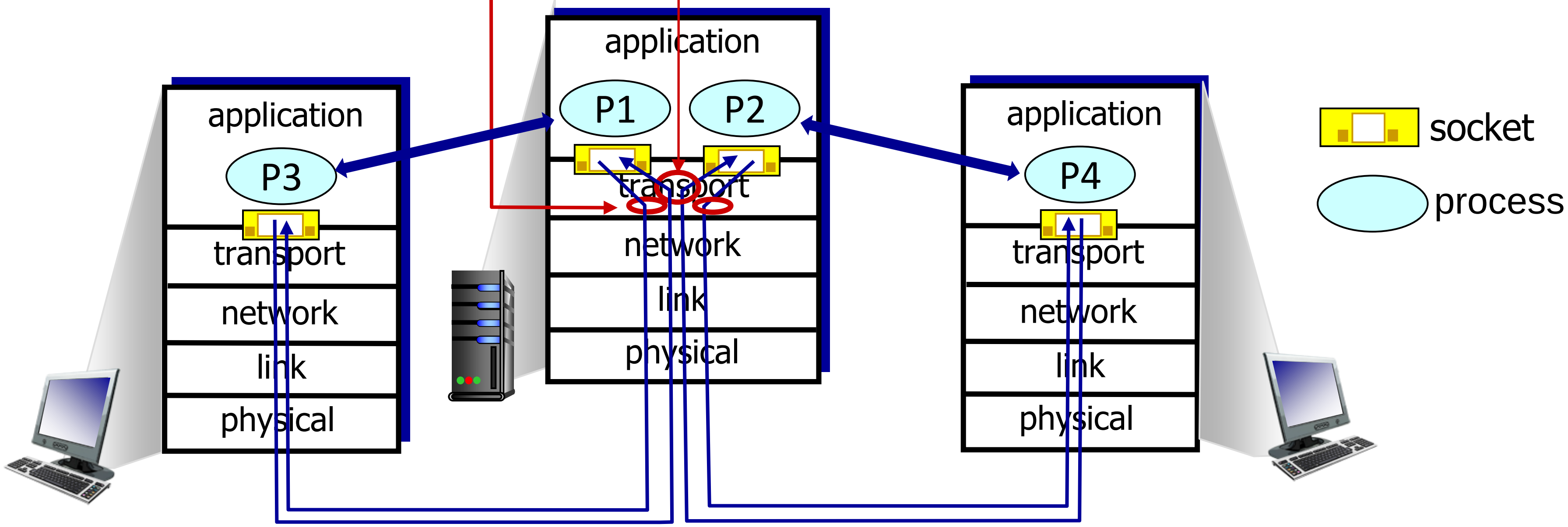
Multiplexing/demultiplexing

multiplexing as sender:

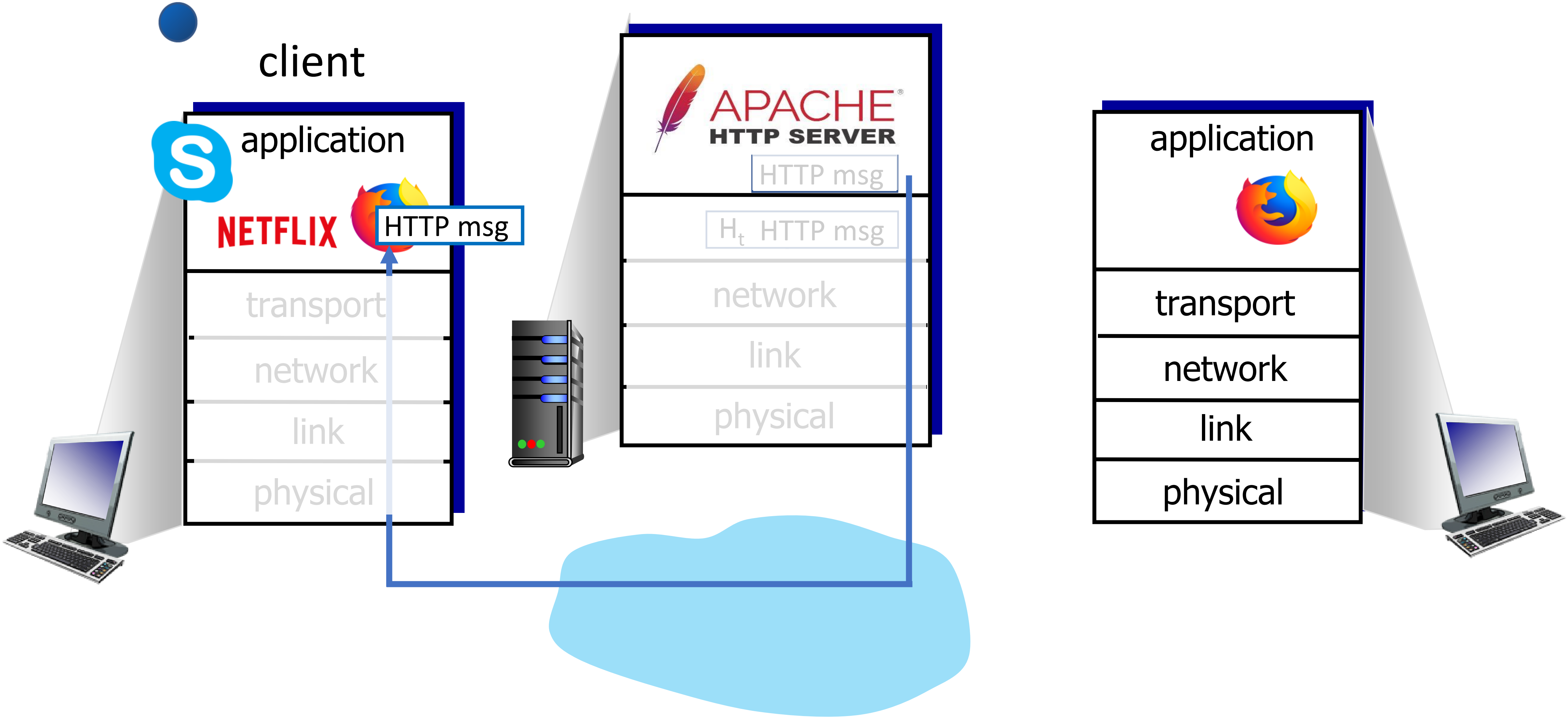
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing as receiver:

use header info to deliver received segments to correct socket



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

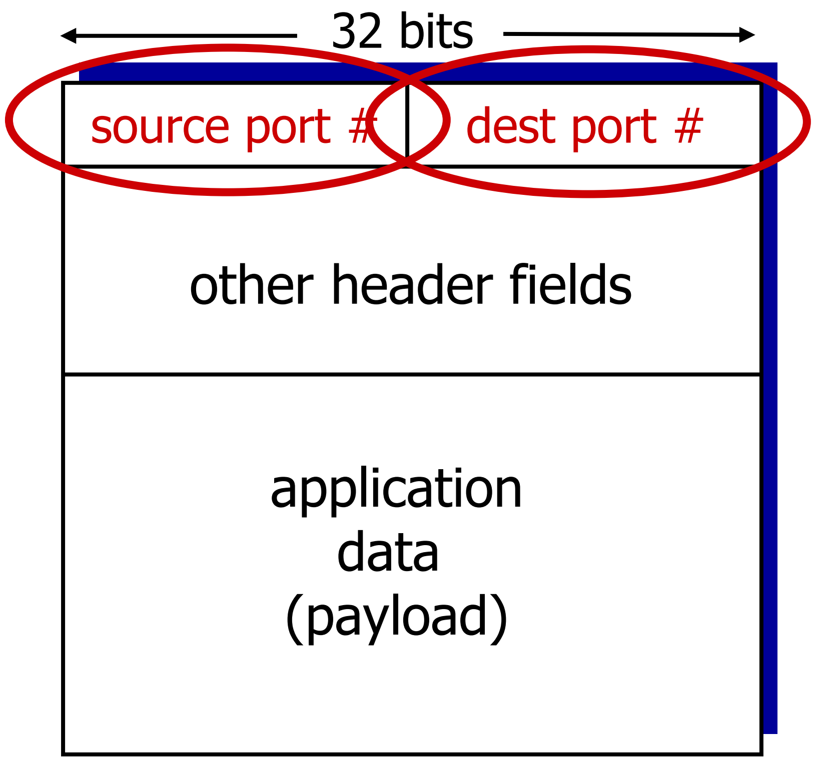


Demultiplexing

Multiplexing

How demultiplexing works

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



TCP/UDP segment format

Connectionless demultiplexing

Recall:

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

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

When receiving host receives *UDP* segment:

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

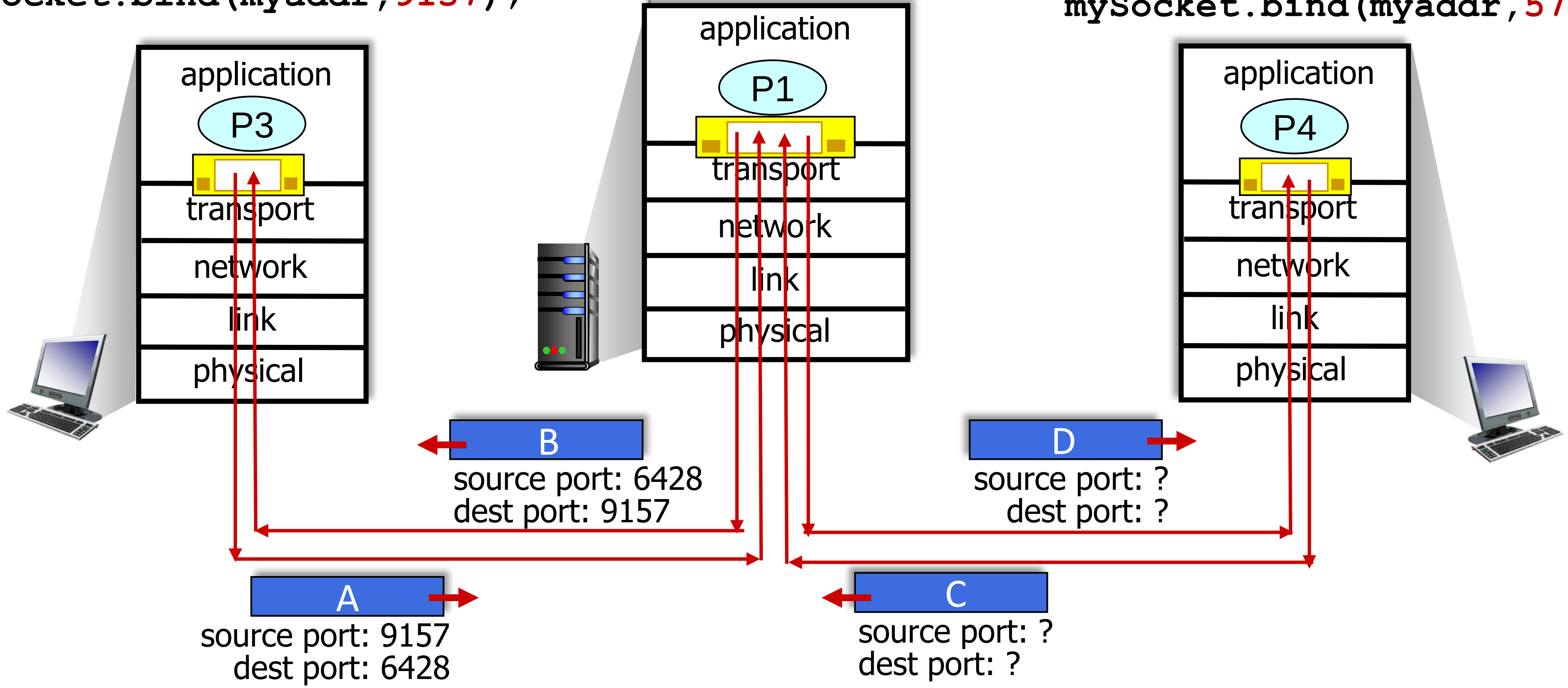
IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,6428);
```

```
mySocket =
 socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr,9157);
```

```
mySocket =
 socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr,5775);
```

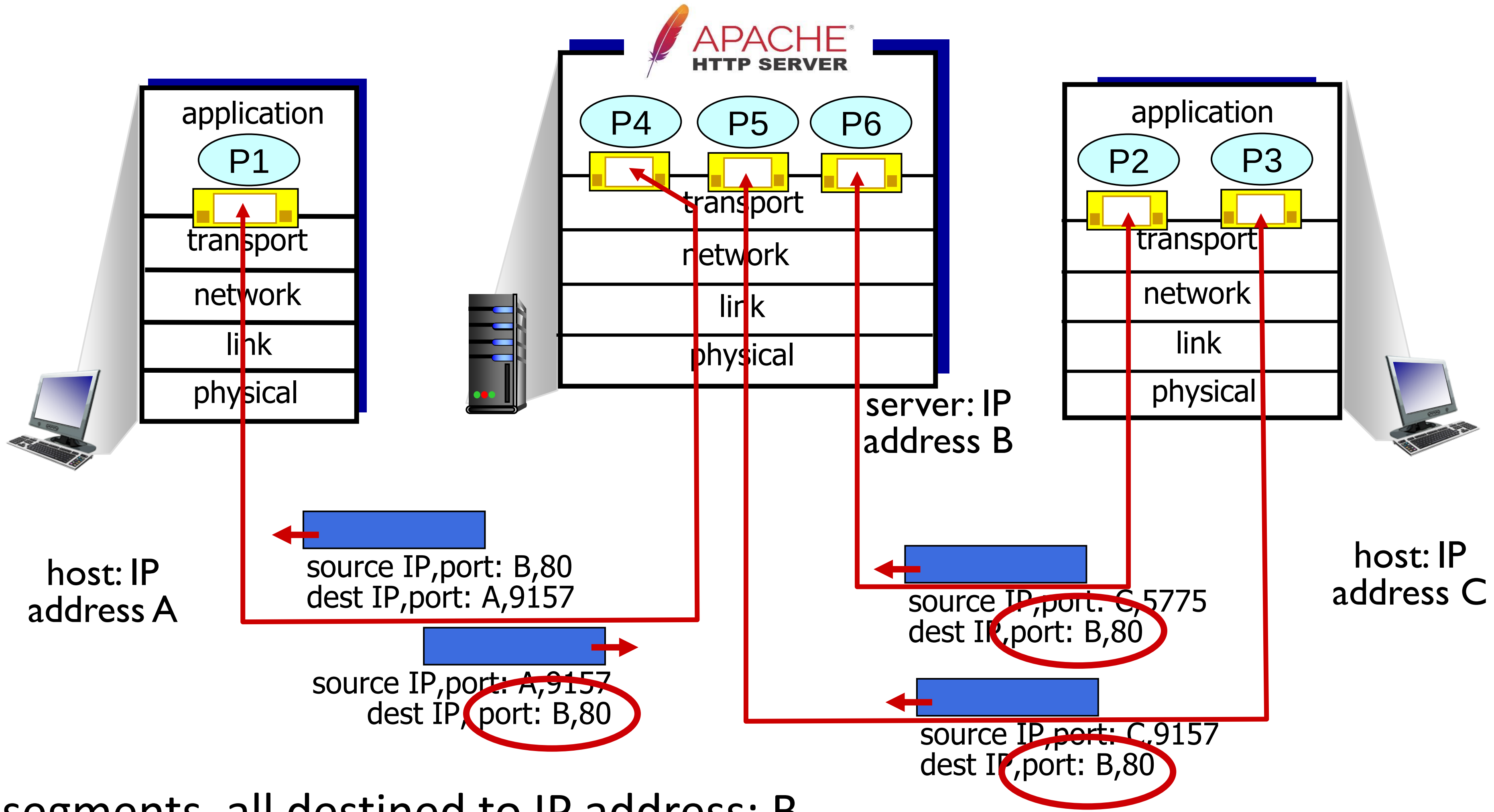


Connection-oriented demultiplexing

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

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

Connection-oriented demultiplexing: example



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

Review Question

- Why do you use destination port and IP address in connectionless demultiplexing as compared to 4 tuples (Source IP, Destination IP, Source Port, and Destination port) in connection-oriented demultiplexing?

- Answer: TCP creates connection with the help of IP address and port numbers of source and destination. UDP does not require connection establishment

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

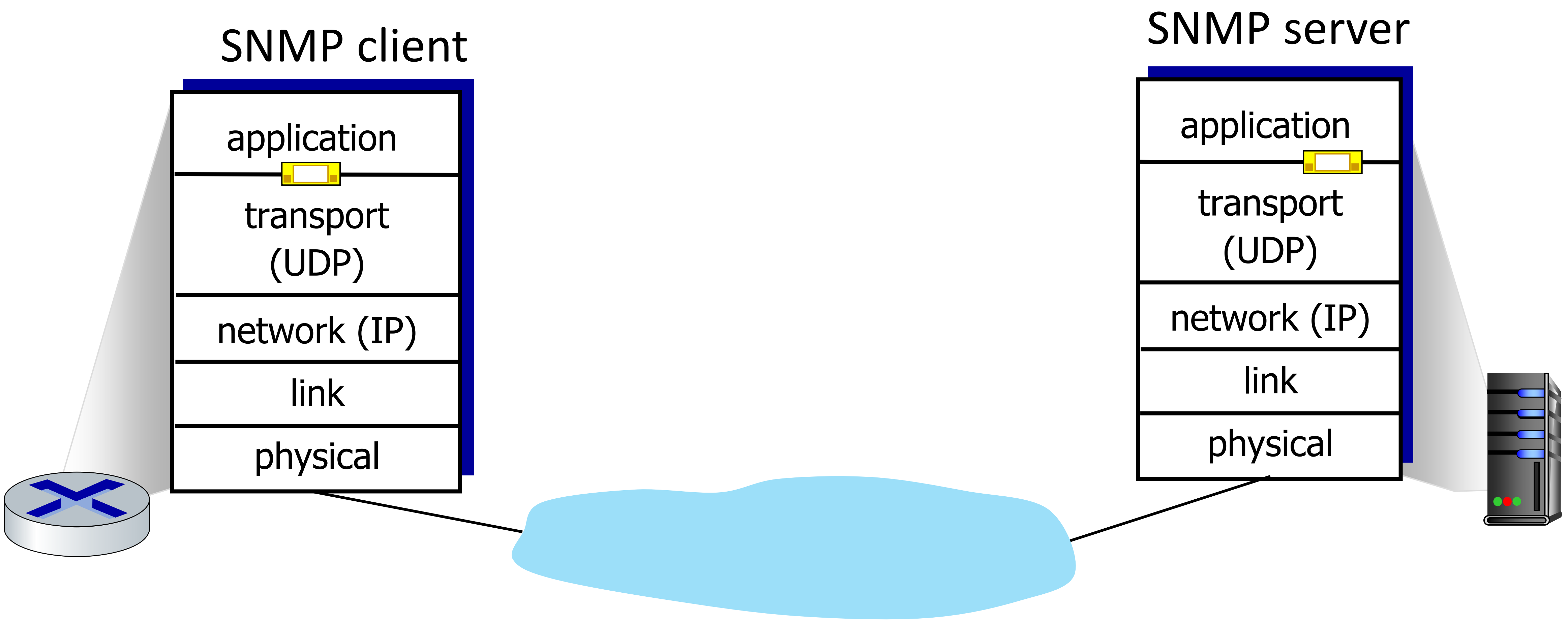
Why is there a UDP?

- No connection establishment (which can add RTT delay)
- Simple: no connection state at sender, receiver
- Small header size
- No congestion control

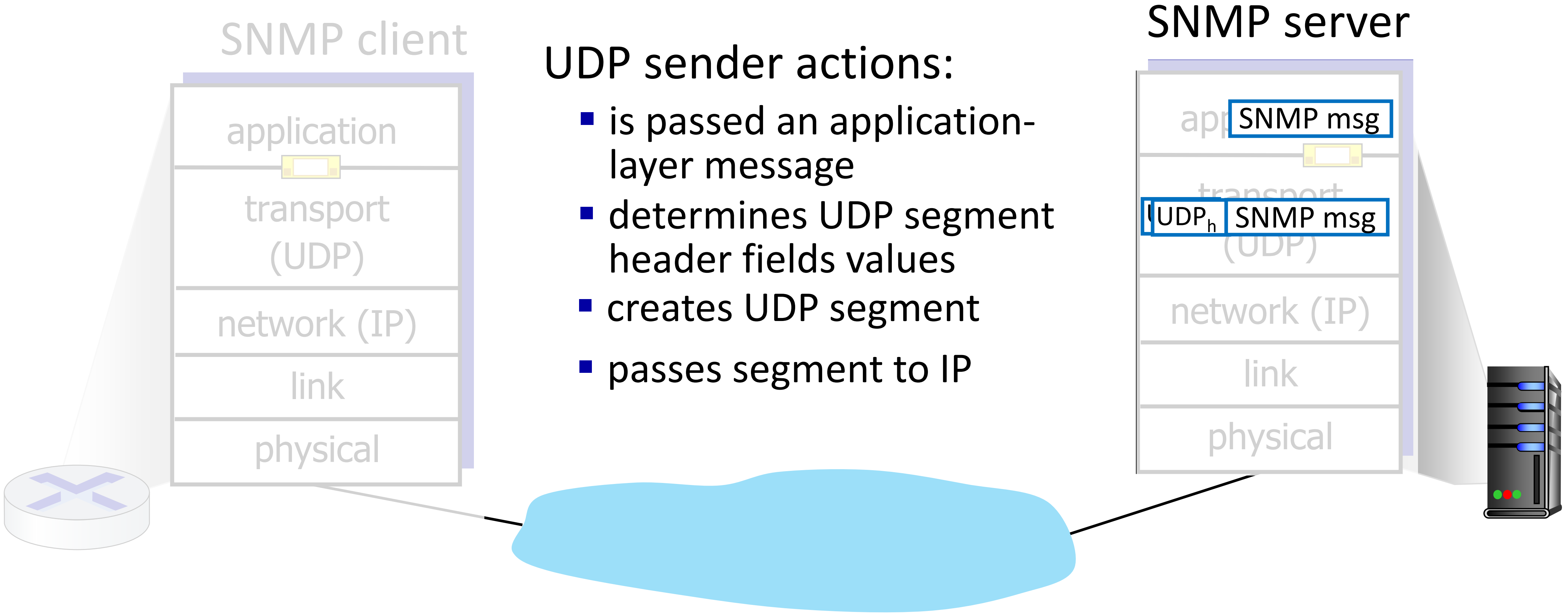
UDP: User Datagram Protocol

- 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: Transport Layer Actions



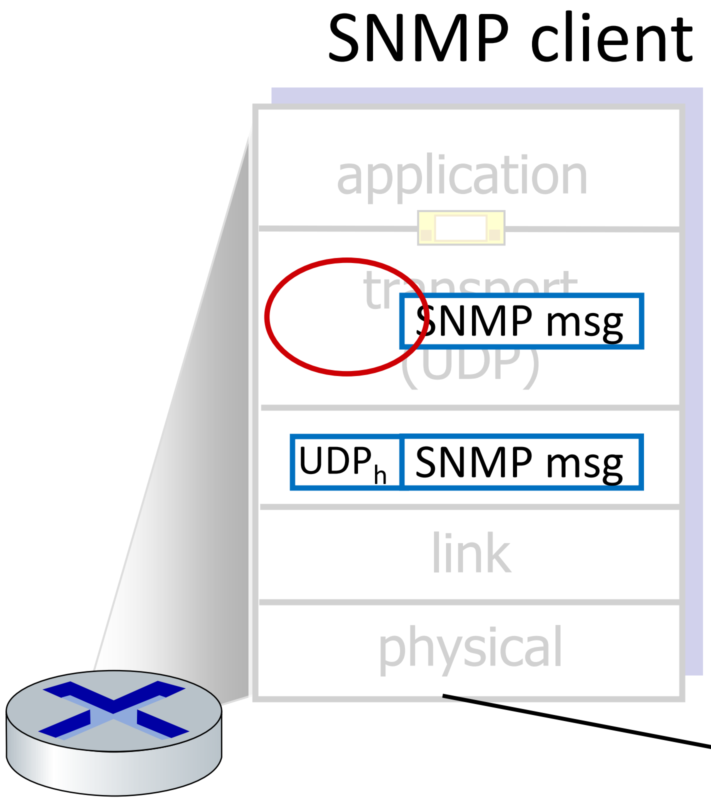
UDP: Transport Layer Actions



UDP sender actions:

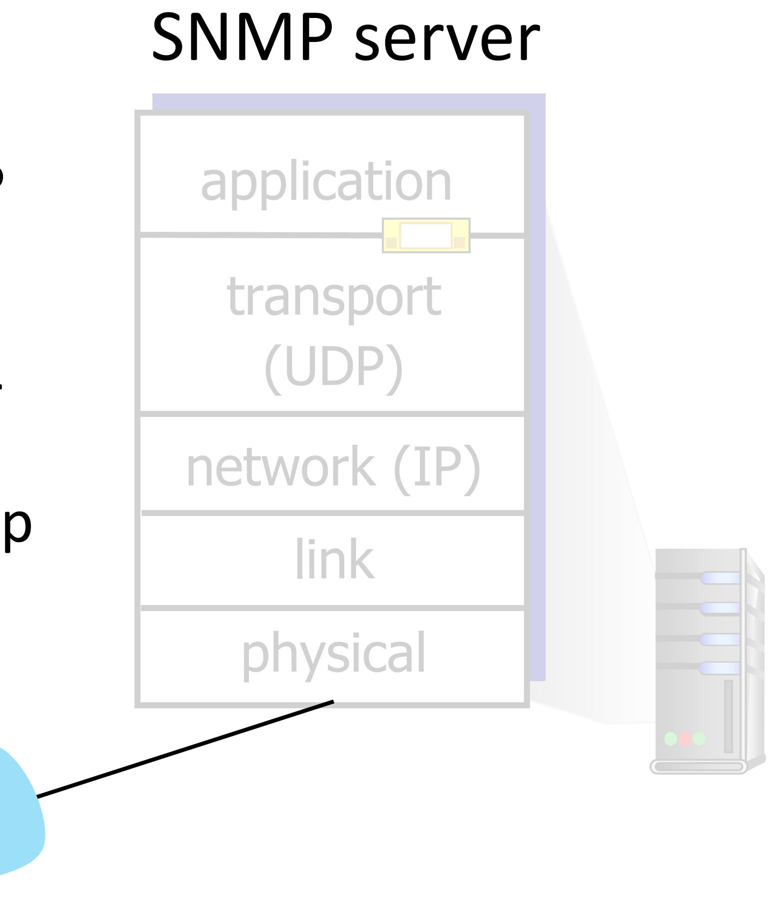
- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

UDP: Transport Layer Actions

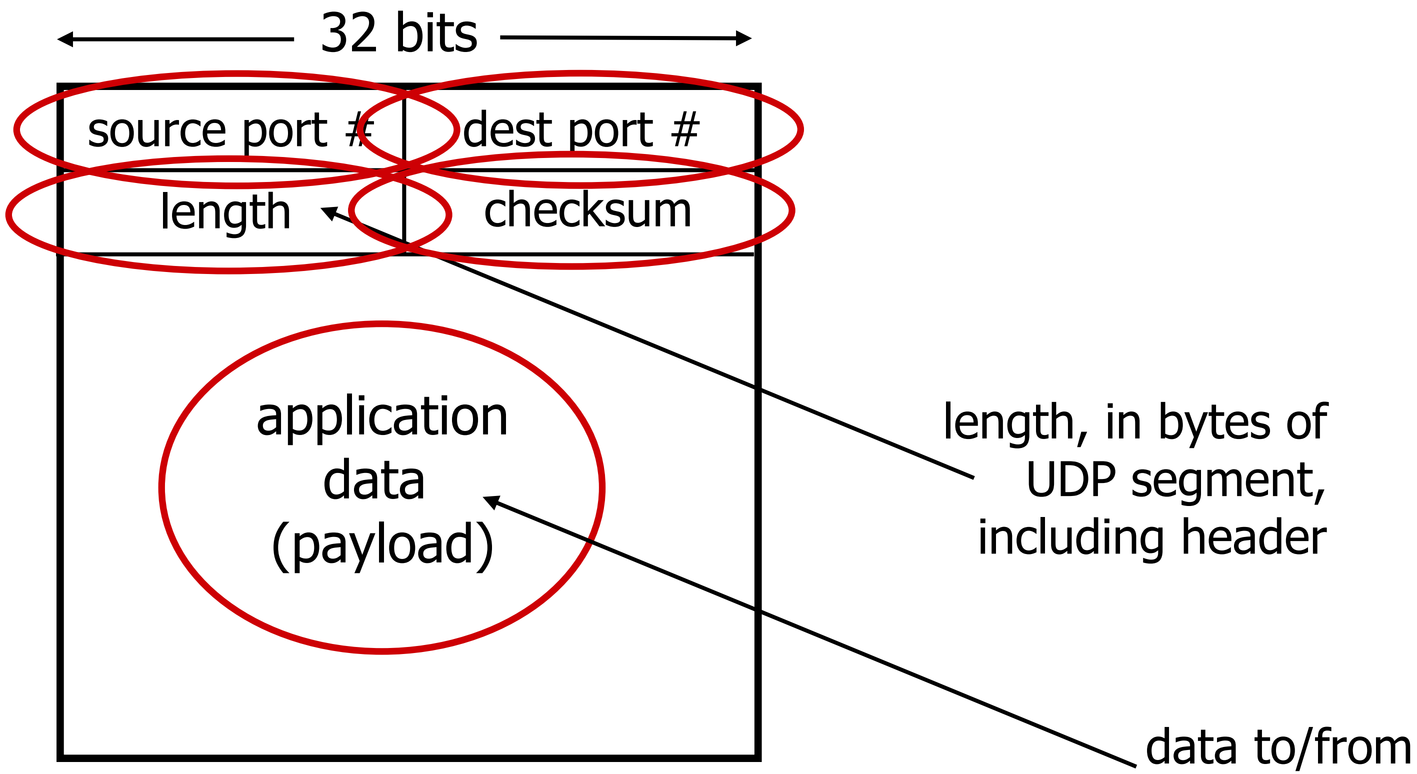


UDP receiver actions:

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



UDP segment header



UDP 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

SACRAMENTO STATE

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

sender:

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

receiver:

- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - not equal - error detected
  - equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an example

example: add two 16-bit integers

		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

Internet checksum: weak protection!

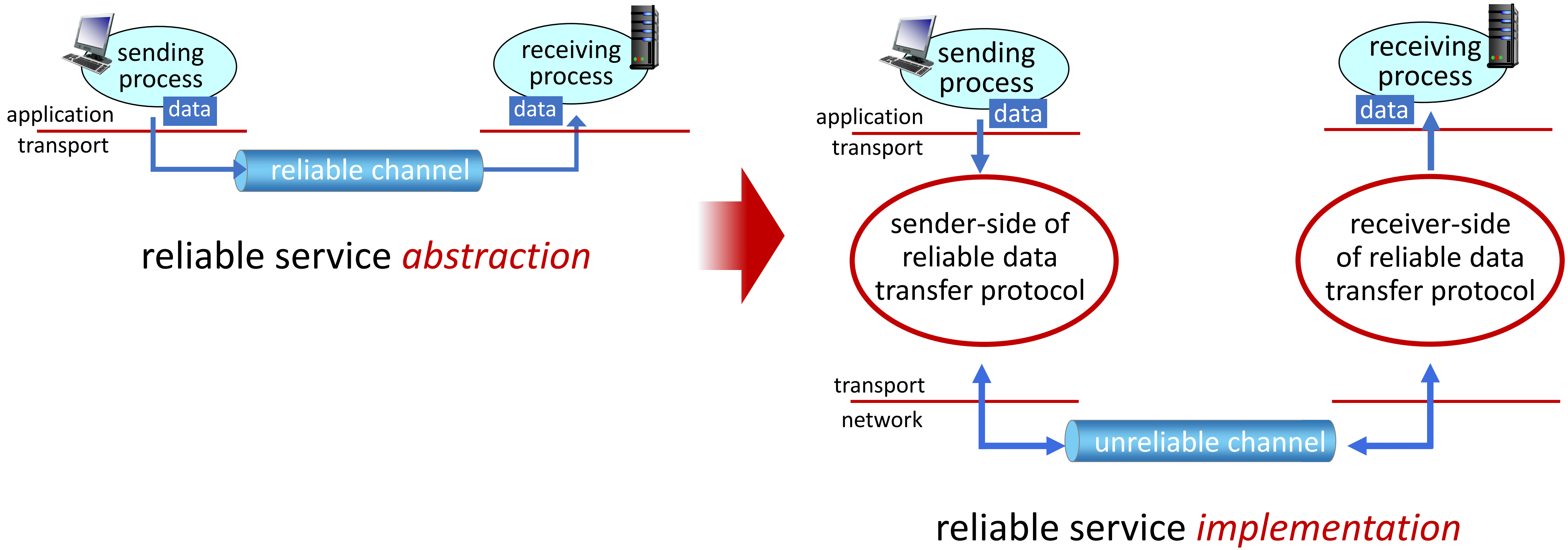
example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		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

0 1
1 0

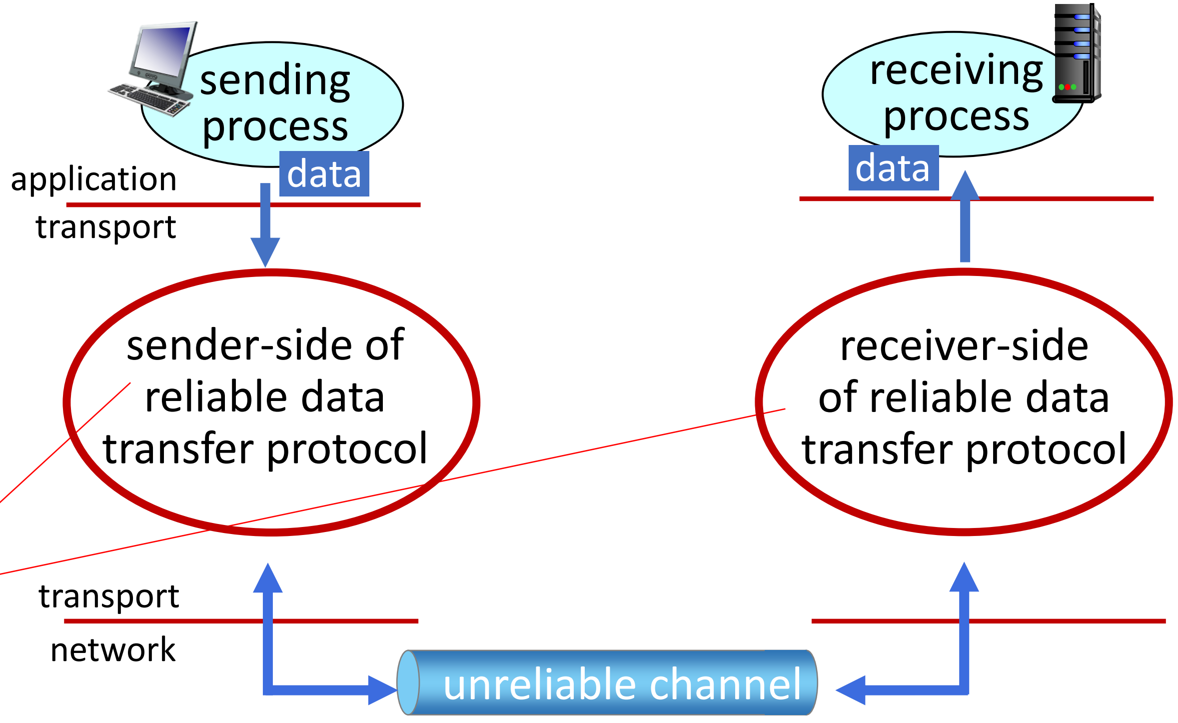
Even though numbers have changed (bit flips), *no* change in checksum!

Principles of reliable data transfer



Principles of reliable data transfer

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

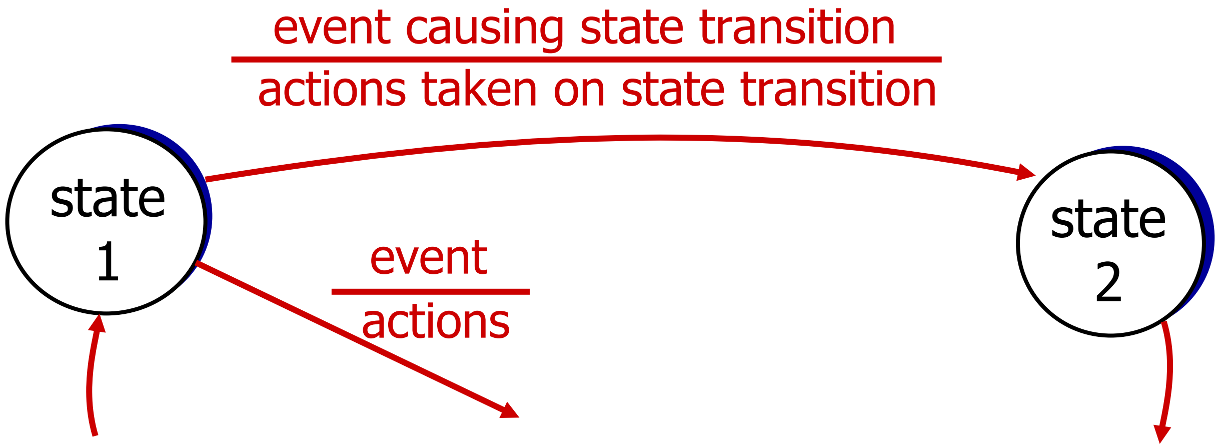


reliable service *implementation*

Reliable data transfer: Evolution

- We will see rdt 1.0, rdt 2.0 and rdt 3.0
  - Each of the varies with respect to their different mechanisms to ensure reliability of data transfer
  - Use finite state machines (FSM) to specify sender, receiver

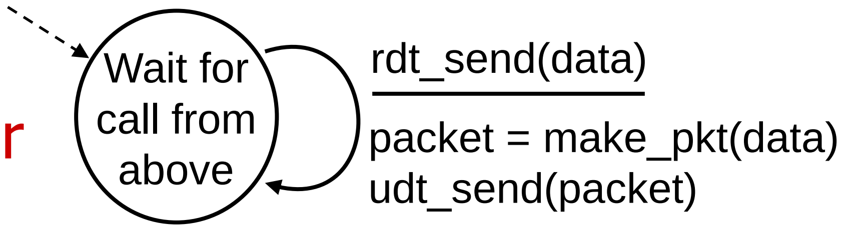
state: when in this "state" next state uniquely determined by next event



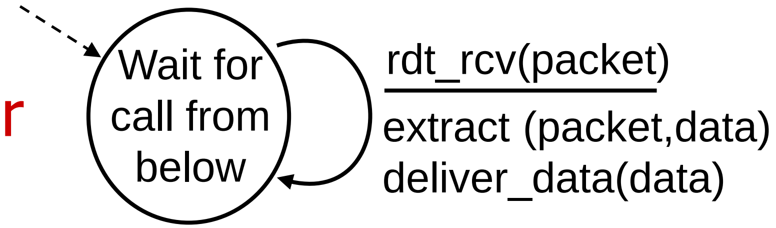
rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- *Separate* FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel
  - rdt = Reliable Data Transfer
  - udt = Unreliable Data Transfer

sender



receiver



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

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

rdt 2.1 addresses the problem of rdt 2.0 by including sequences

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?

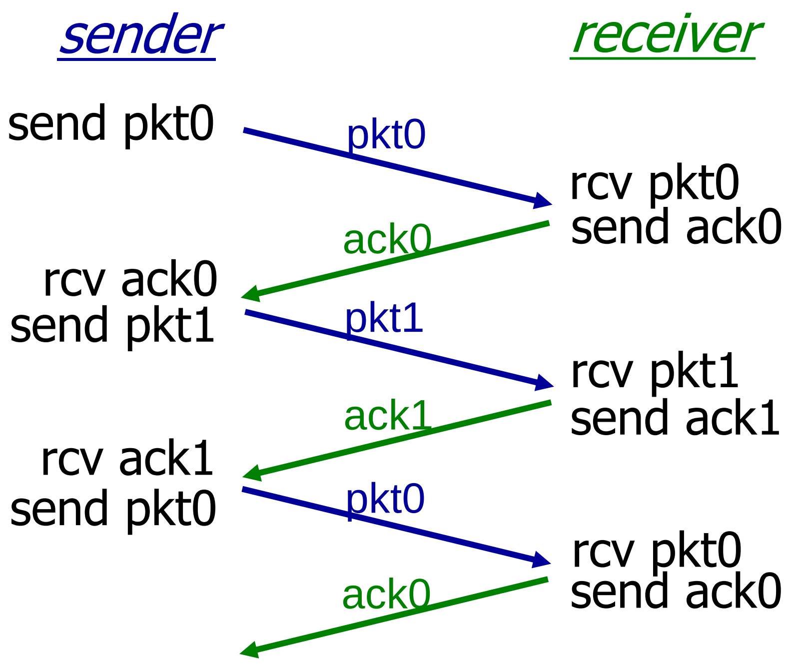
rdt3.0: channels with errors *and* loss

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

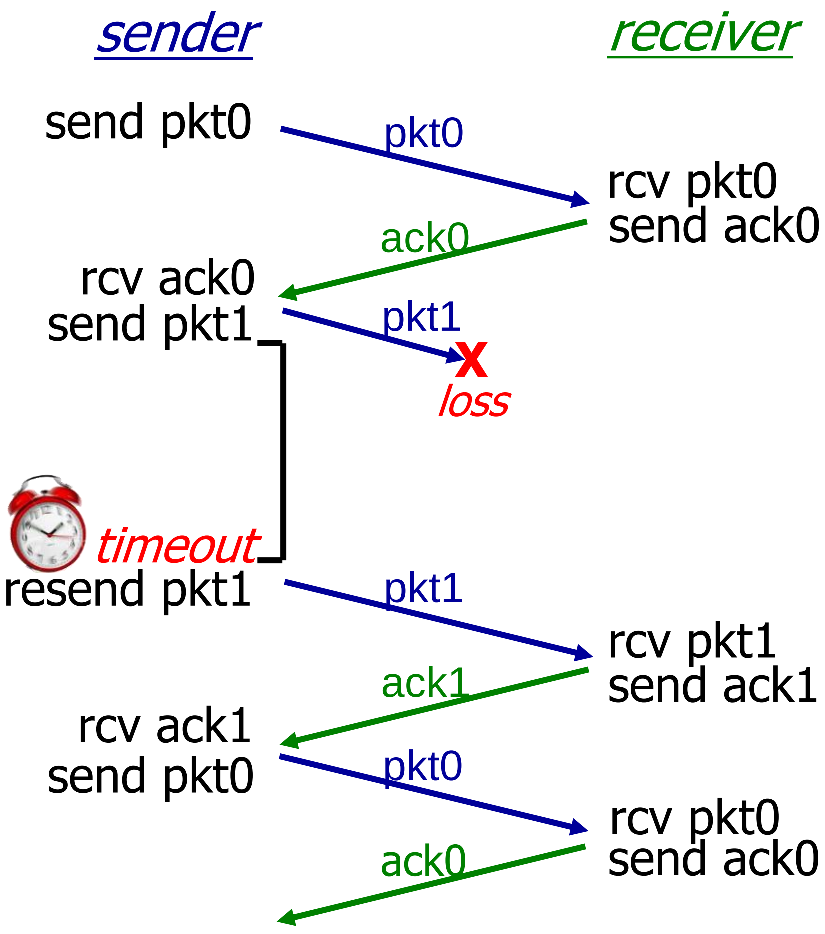
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

rdt3.0 in action

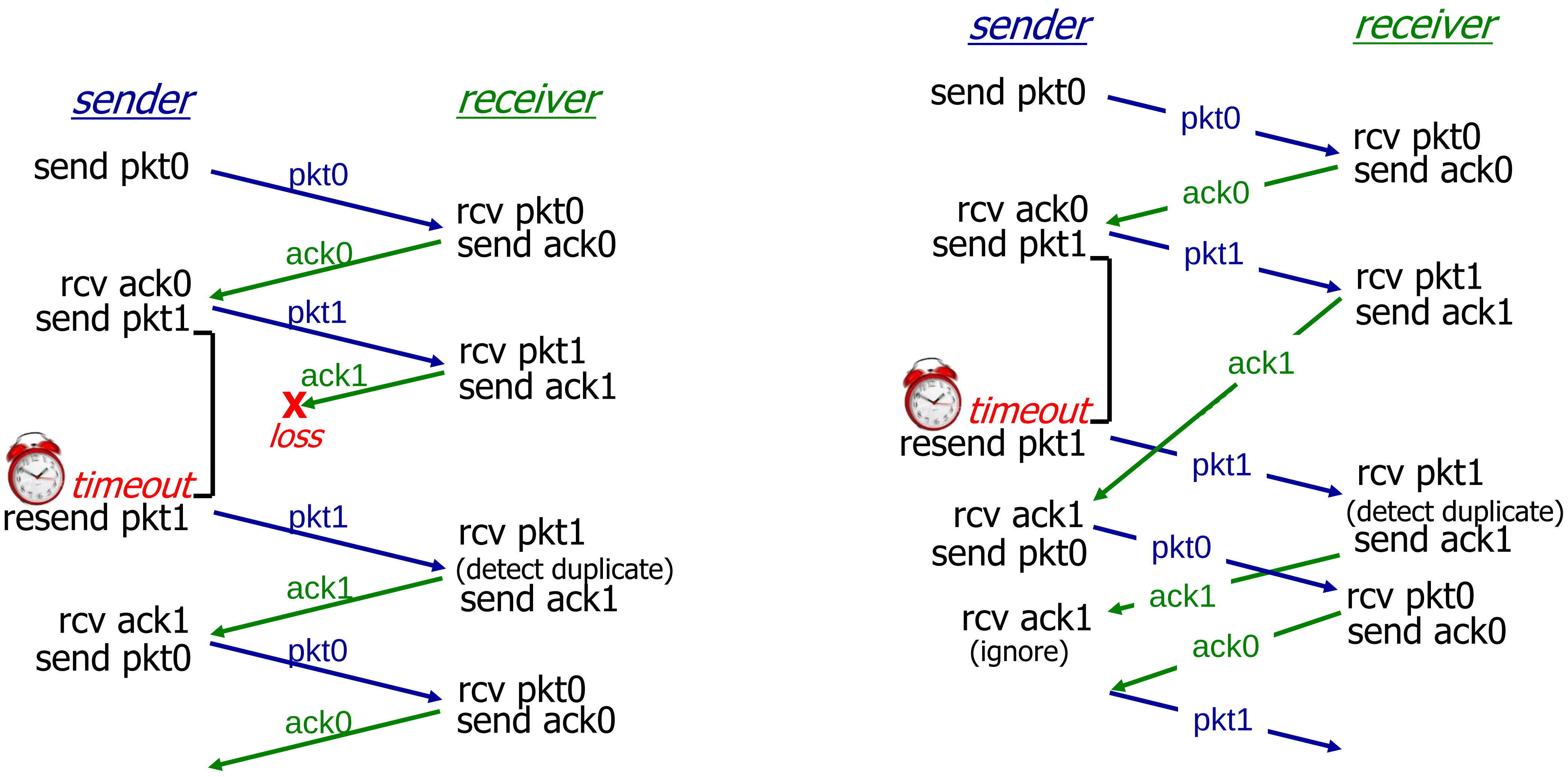


(a) no loss



(b) packet loss

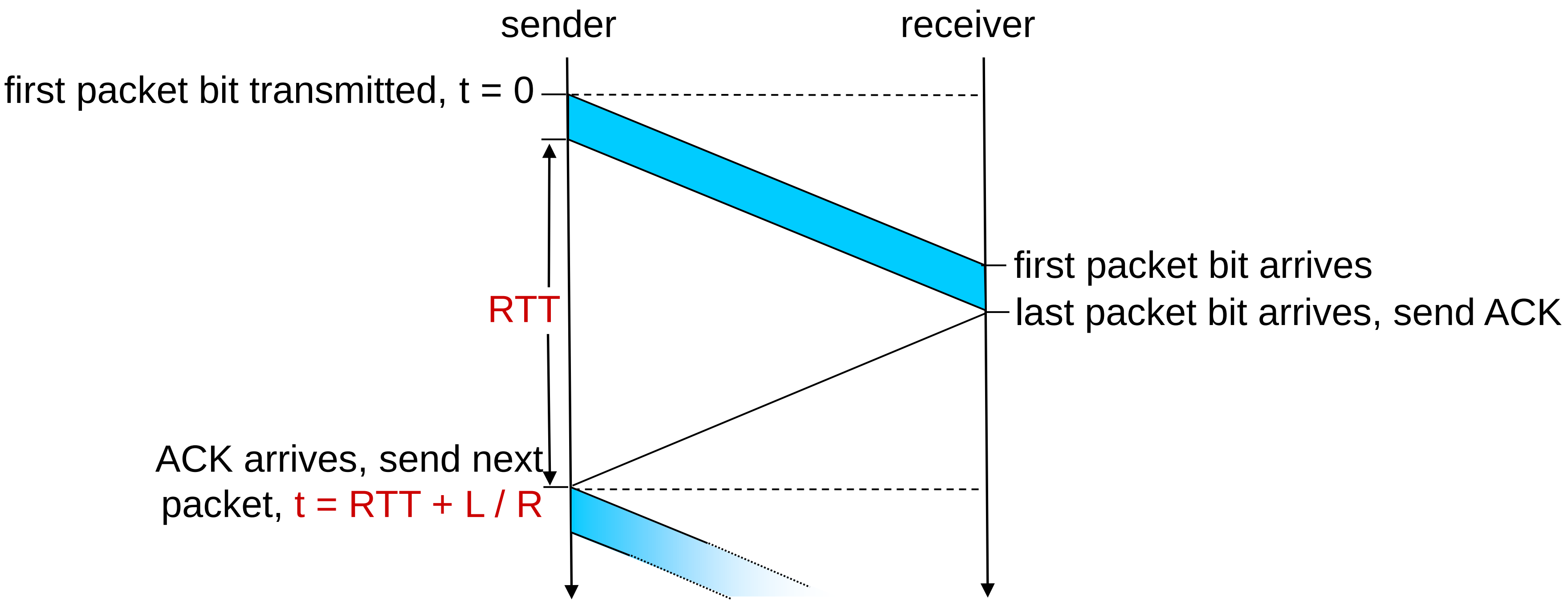
rdt3.0 in action



(c) ACK loss

(d) premature timeout/ delayed ACK

rdt3.0: stop-and-wait operation



Assume L = 8000 bits and R = 10^9 bits/sec and RTT is 30

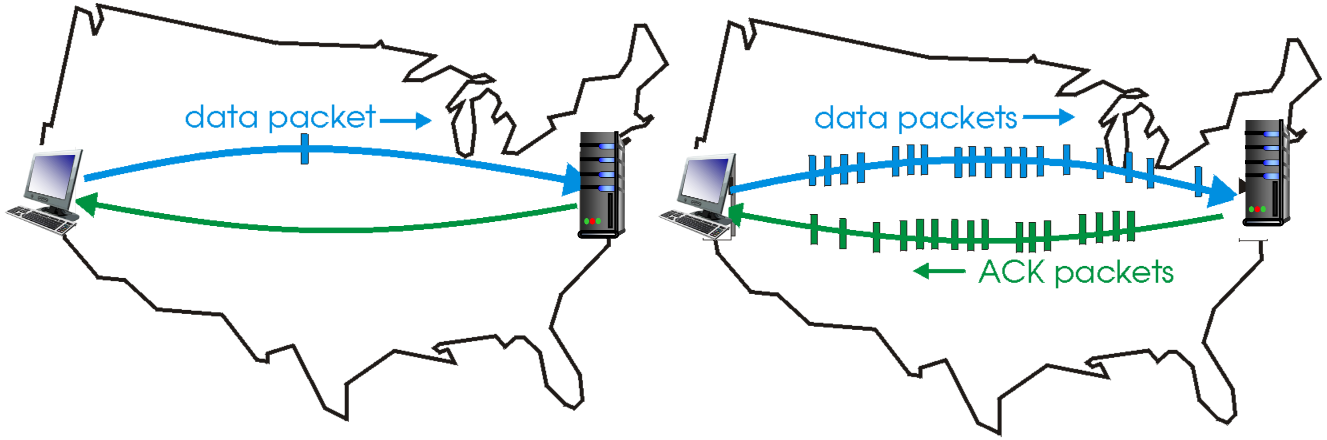
$$D_{trans} = \frac{8000\ bits}{10^9\ bits/sec} = 8 \text{ microseconds}$$

$$U_{sender} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30.008} = 0.00027$$ (Utilization of sender)

rdt3.0: pipelined protocols operation

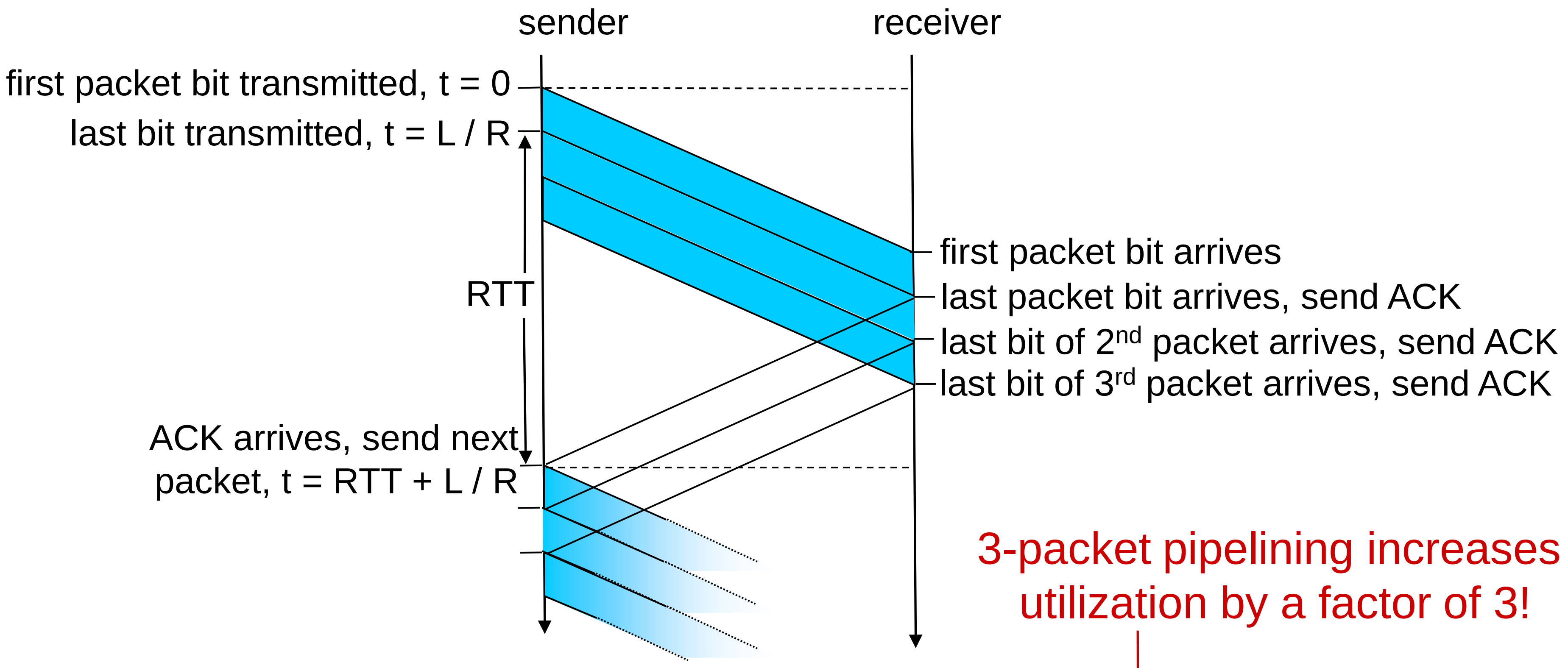
Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

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



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

Pipelining: increased utilization

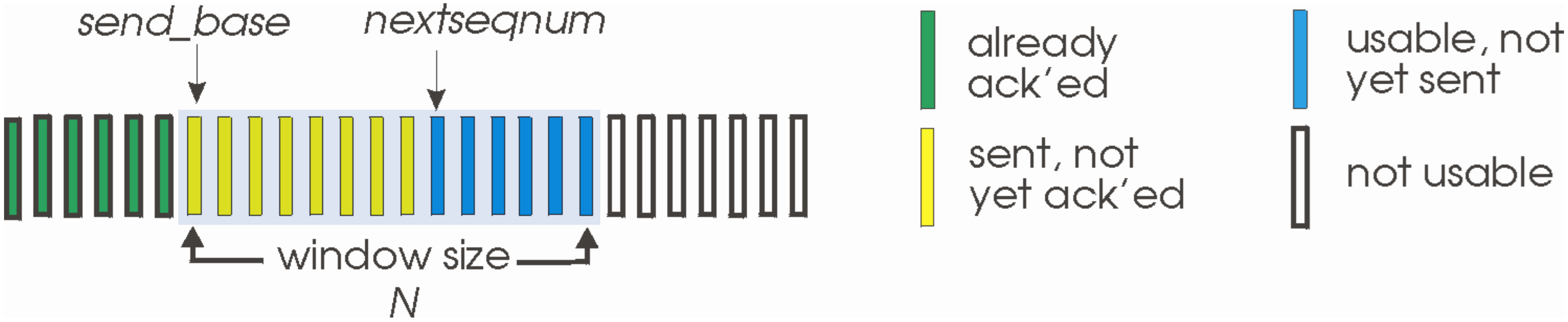


3-packet pipelining increases utilization by a factor of 3!

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

Go-Back-N: sender

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

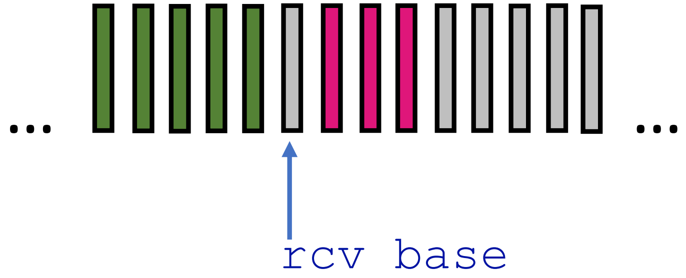


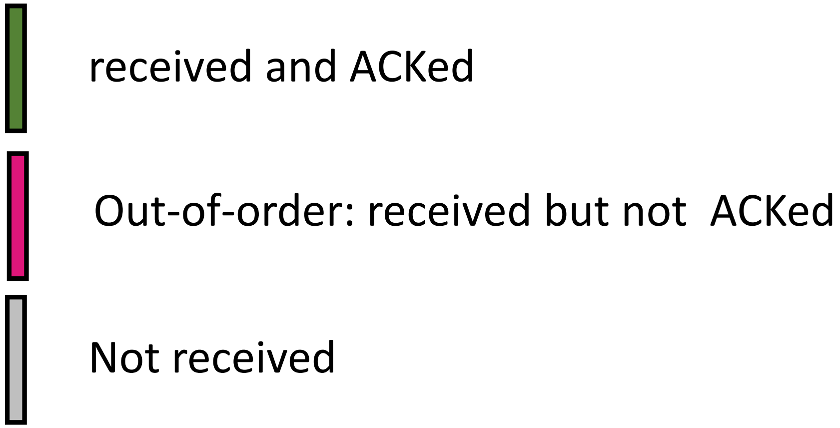
- *Cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- Timer for oldest in-flight packet
- *Timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

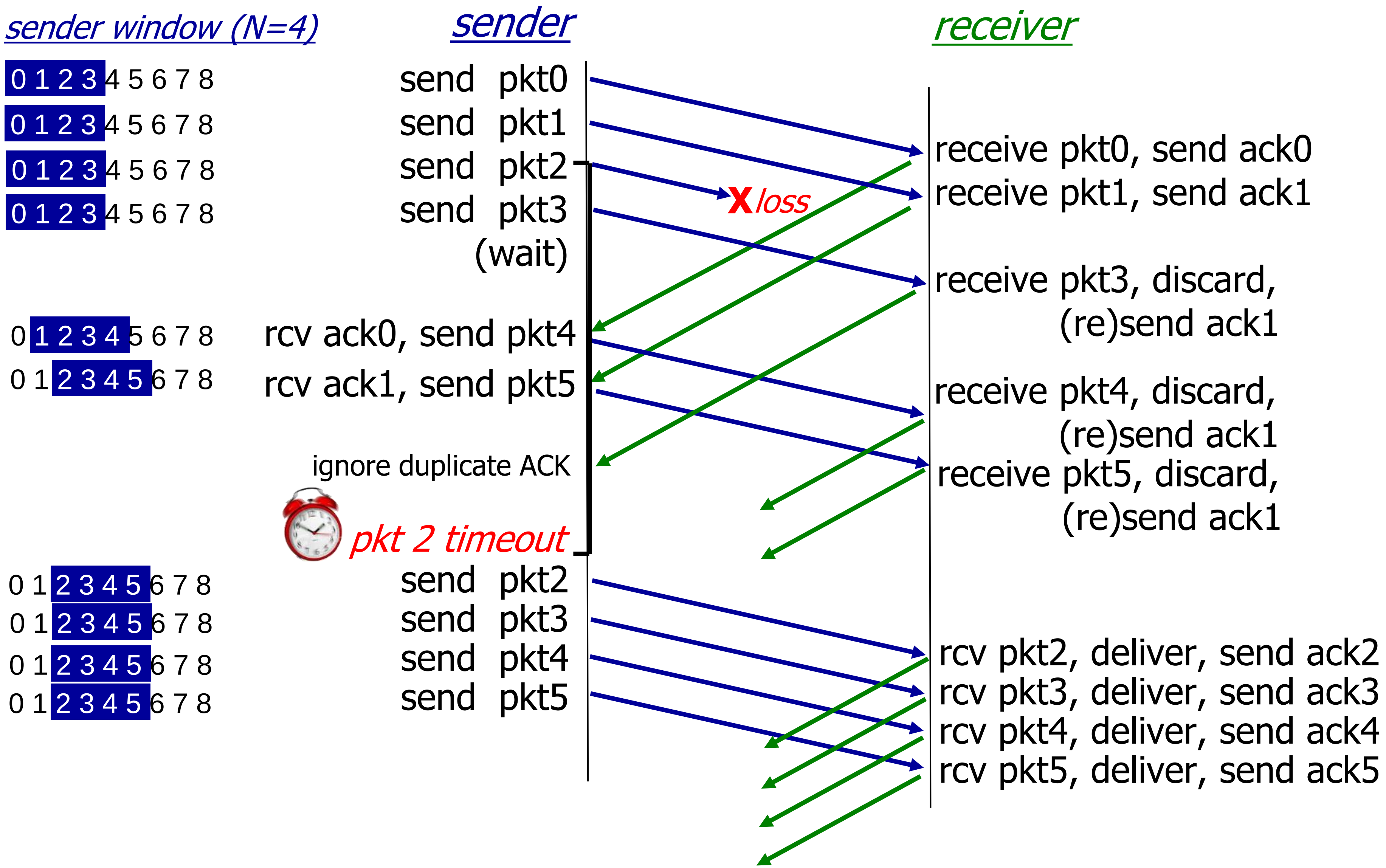
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- On receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order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

Receiver view of sequence number space:





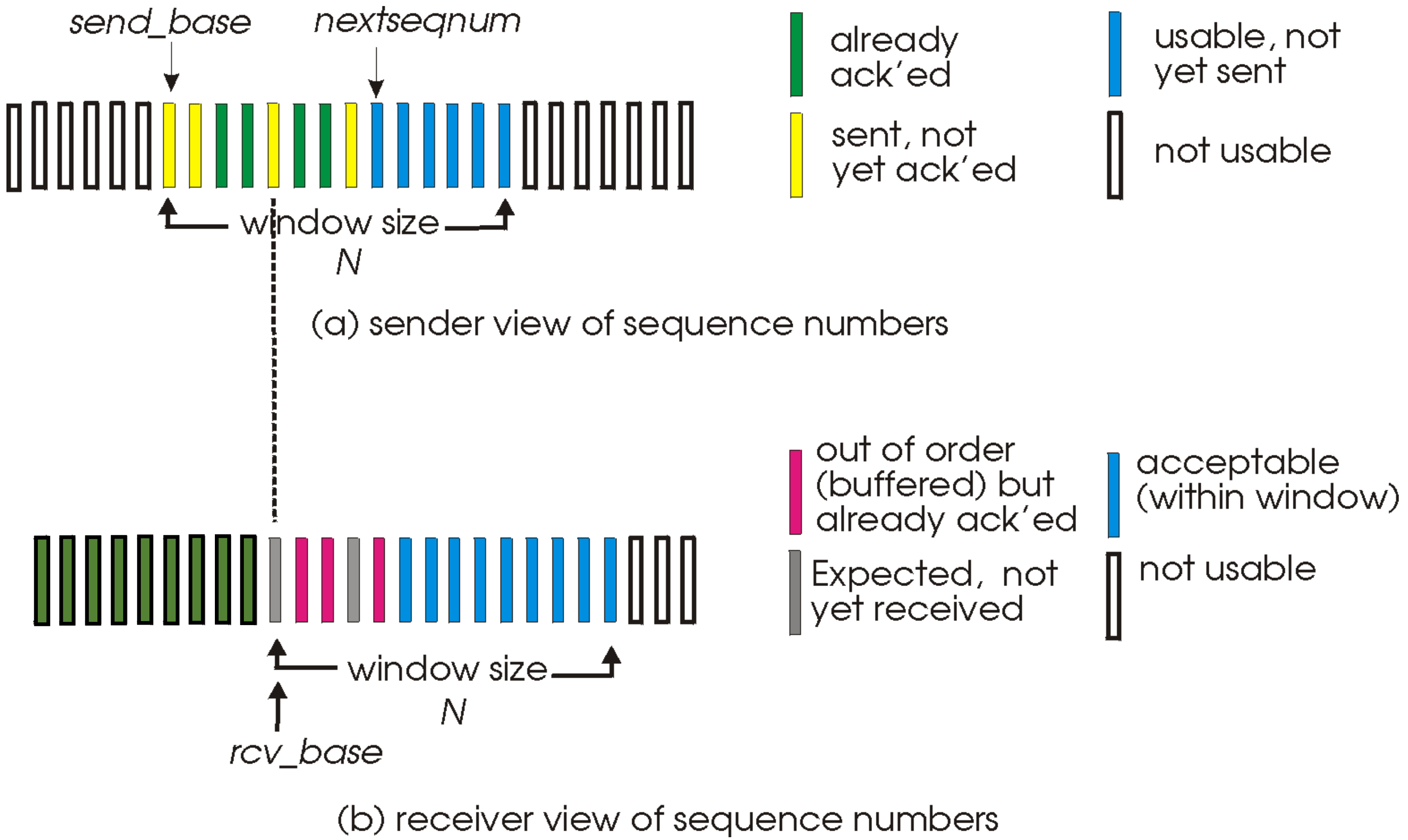
Go-Back-N in action



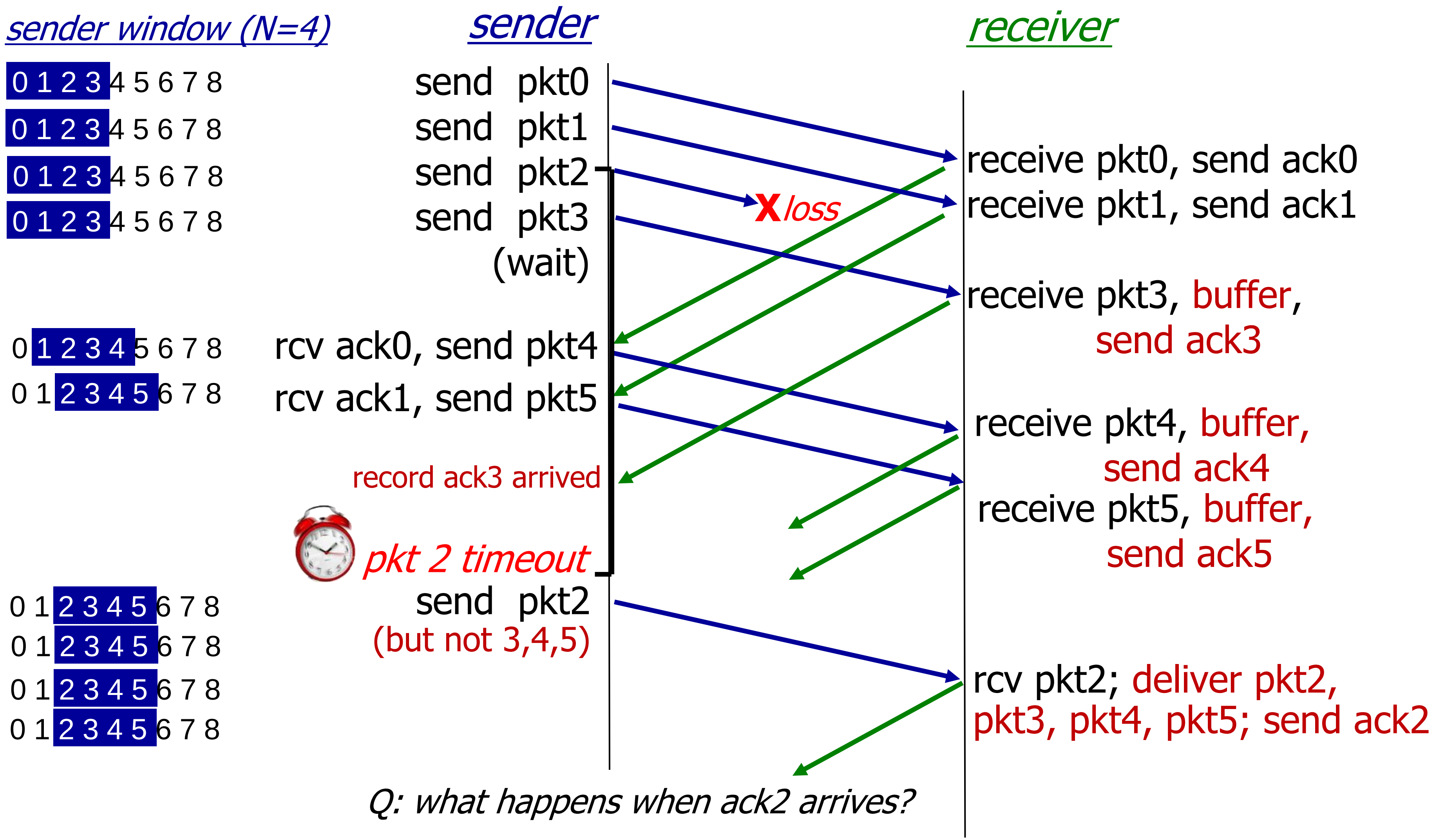
Selective repeat: the approach

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

Selective repeat: sender, receiver windows



Selective Repeat in action



Lecture 3_1 Summary

- Multiplexing, demultiplexing
- UDP transport protocol
- Reliable data transfer
- Pipelining

End of Lecture 3_1