COMS 2014A / 2020A

Computer Networks

Mr. Gift Khangamwa

Office: TWK MSB UG05

Lecture 5: Transport layer

Our goal:

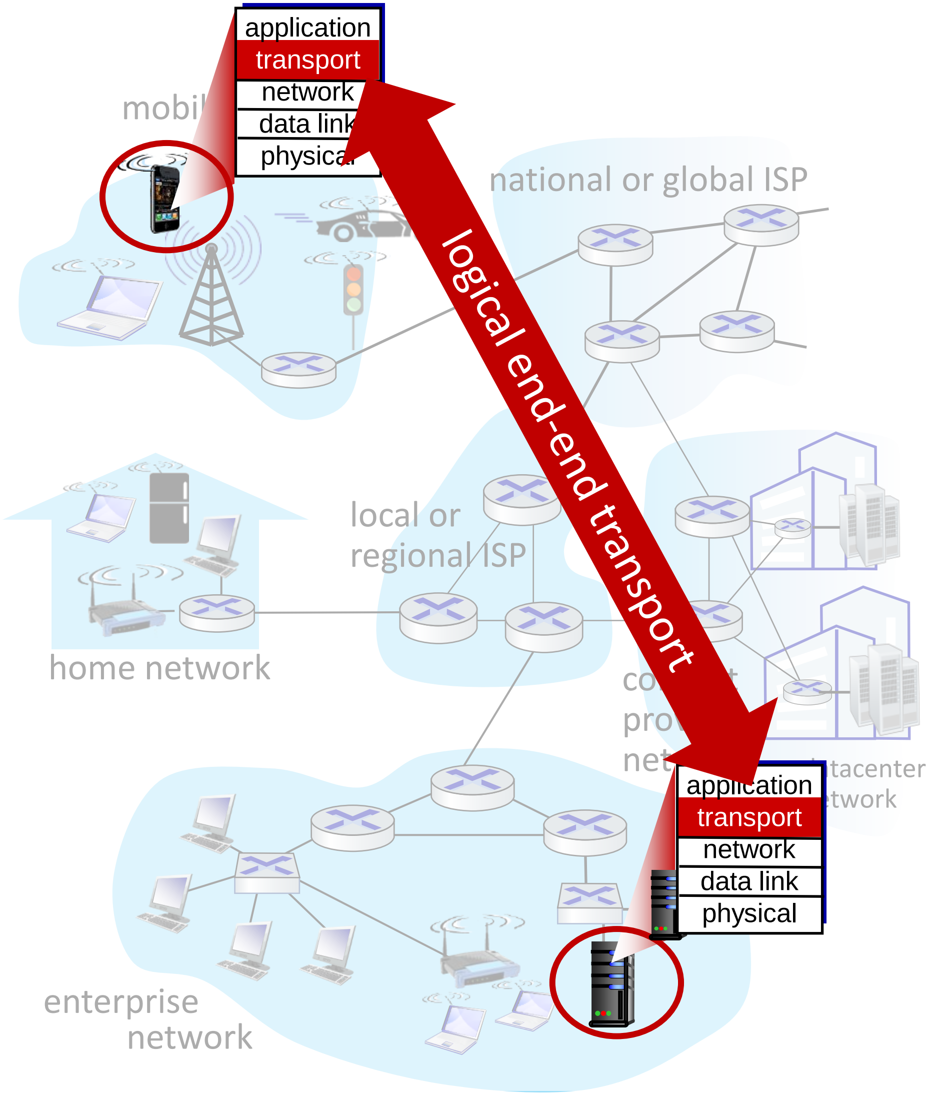
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - We will briefly outline reliable transfer protocol requirements
  - 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:

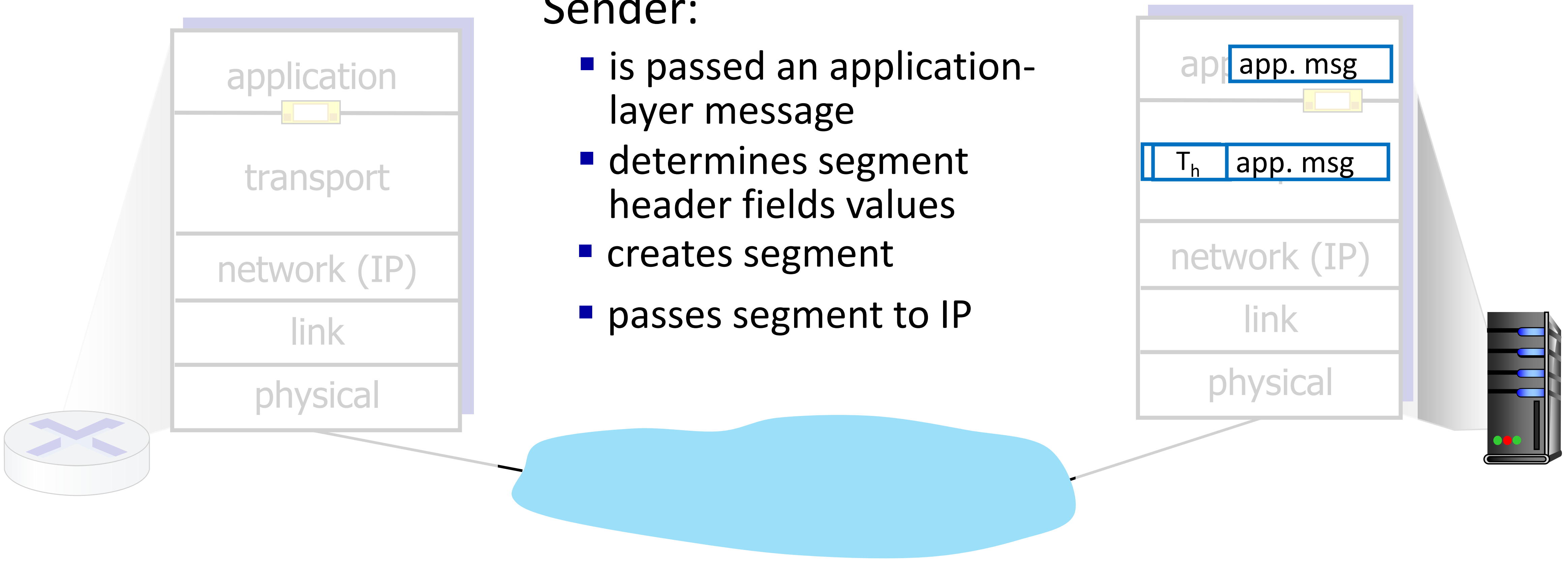
12 kids in Ann's house sending letters to 12 kids in Bill's house:

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

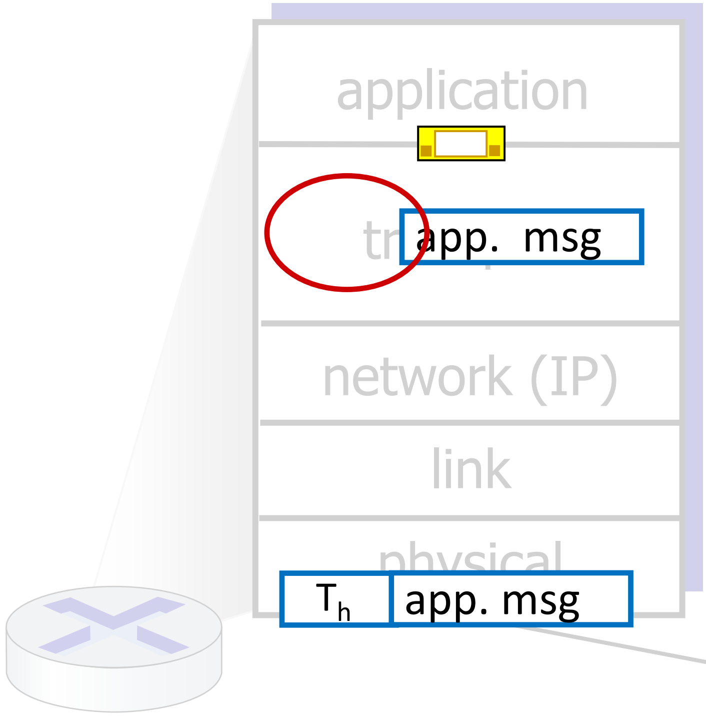
Transport Layer Actions

Sender:

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

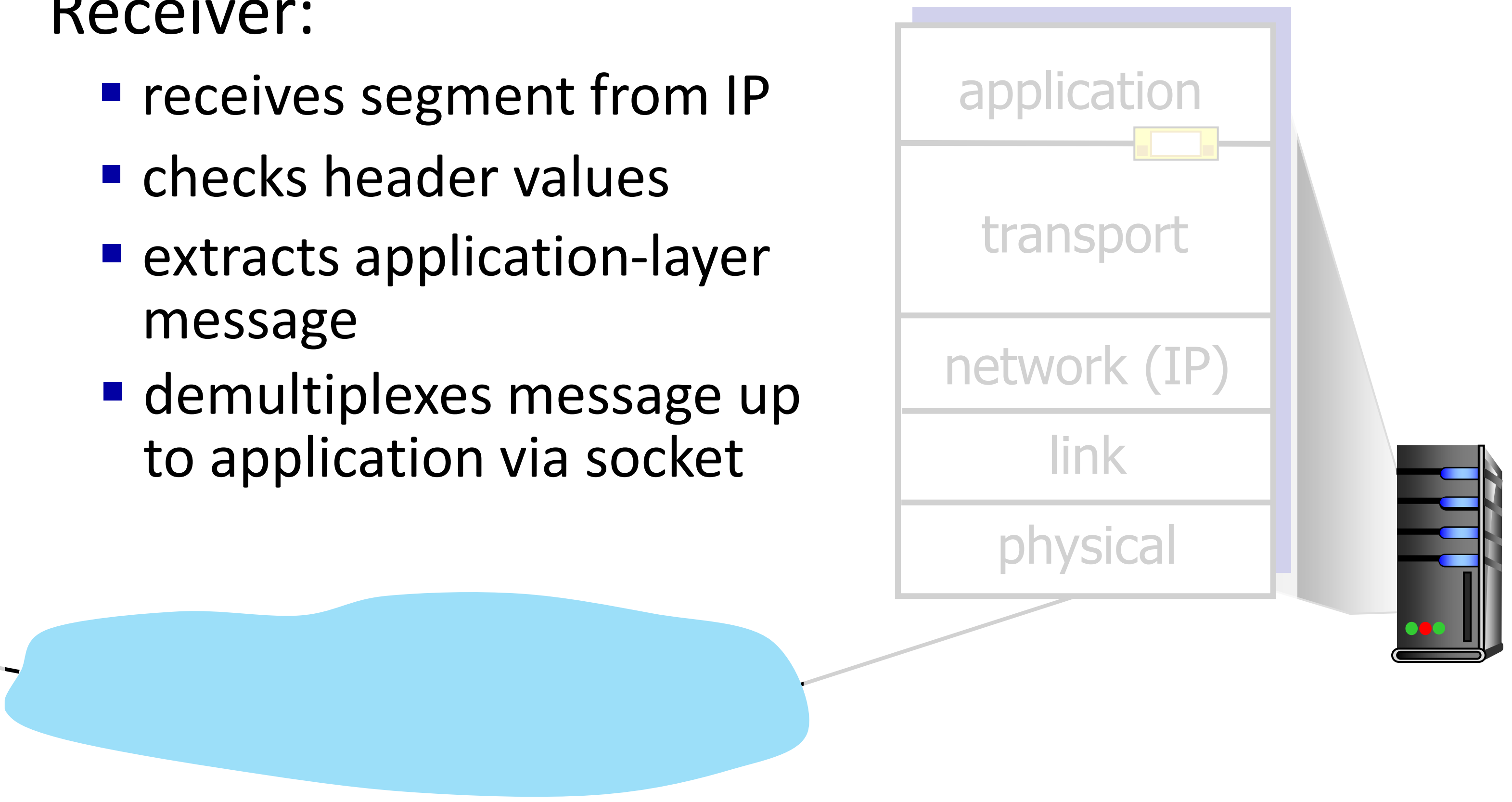


Transport Layer Actions



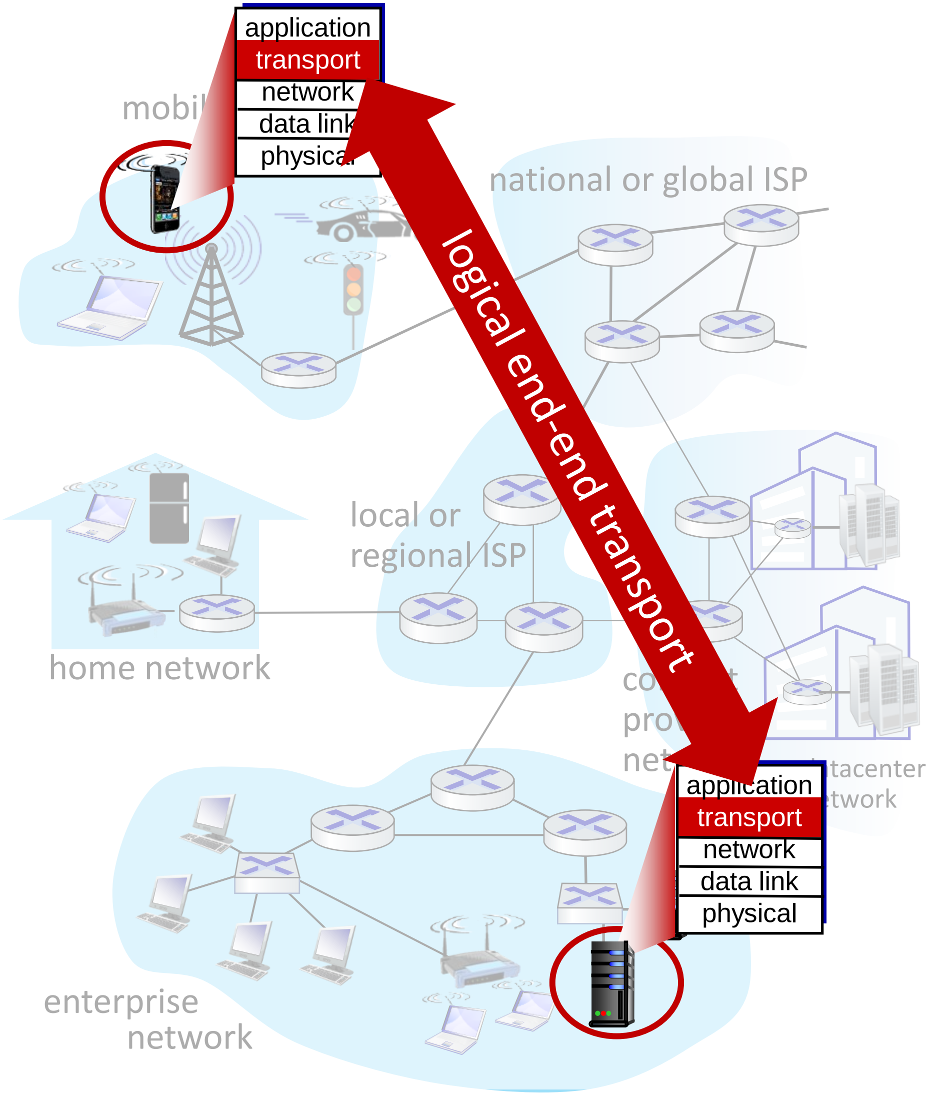
Receiver:

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

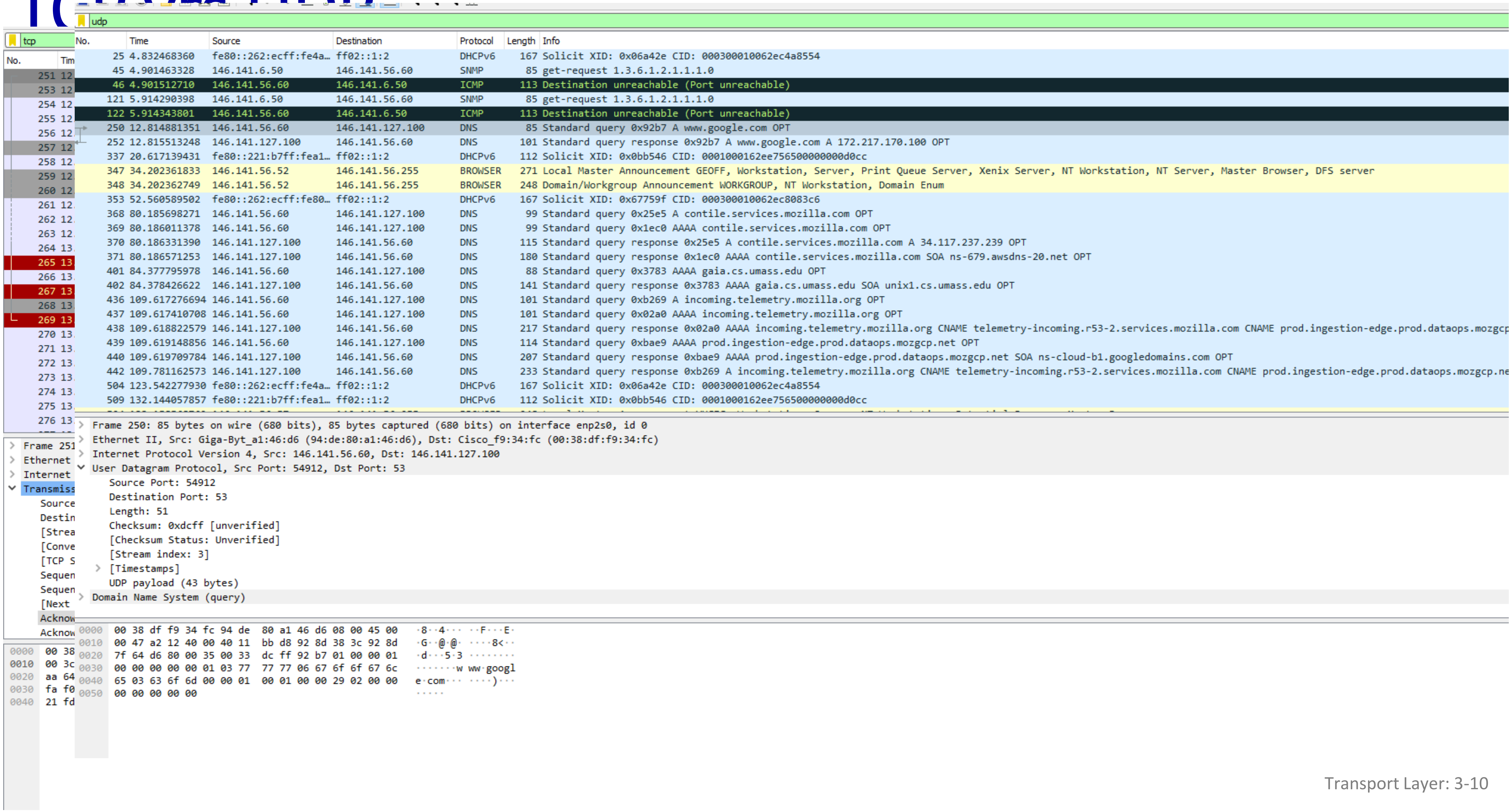


Two principal Internet transport protocols

- **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



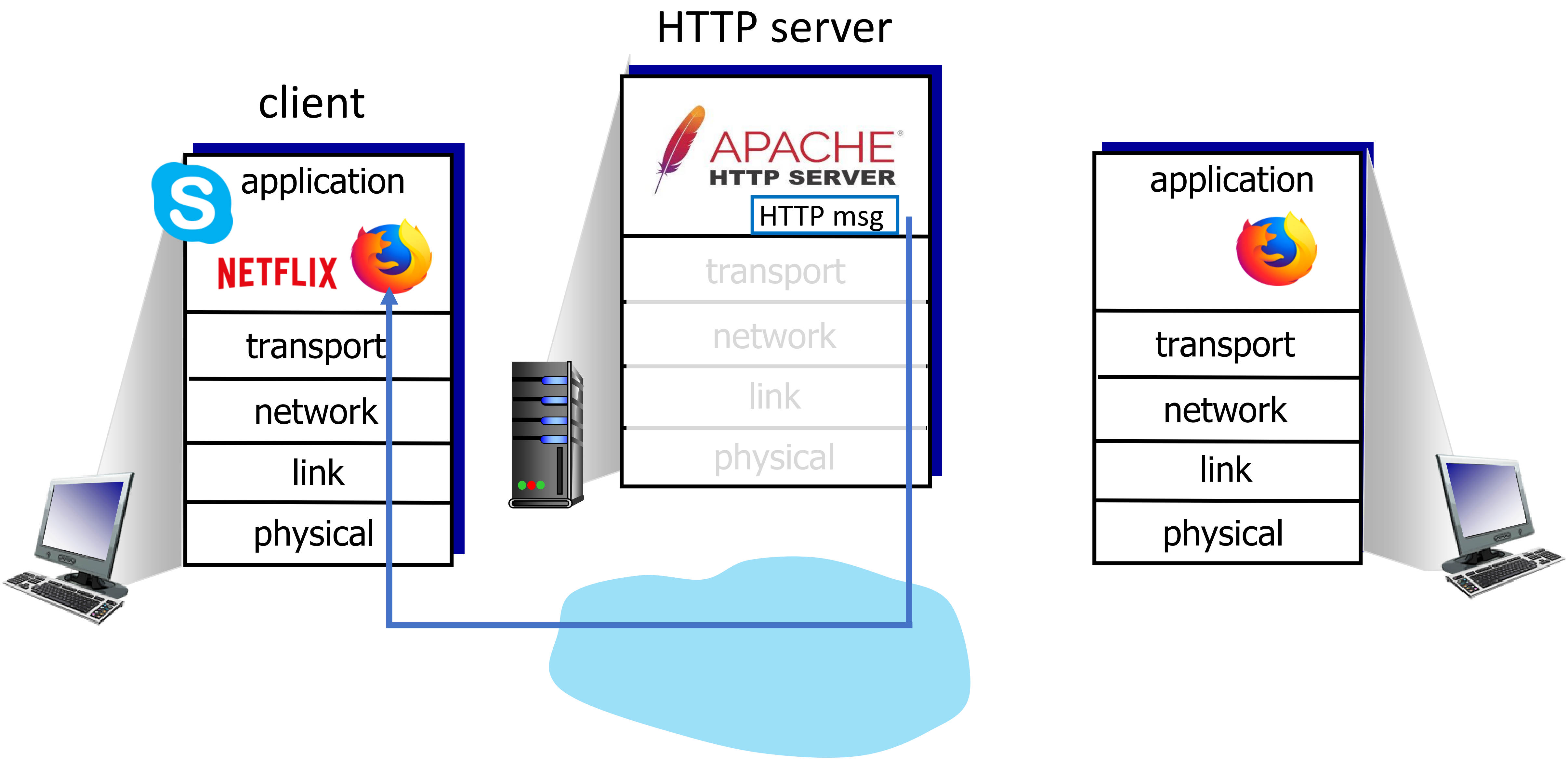
TCP vrs UDP

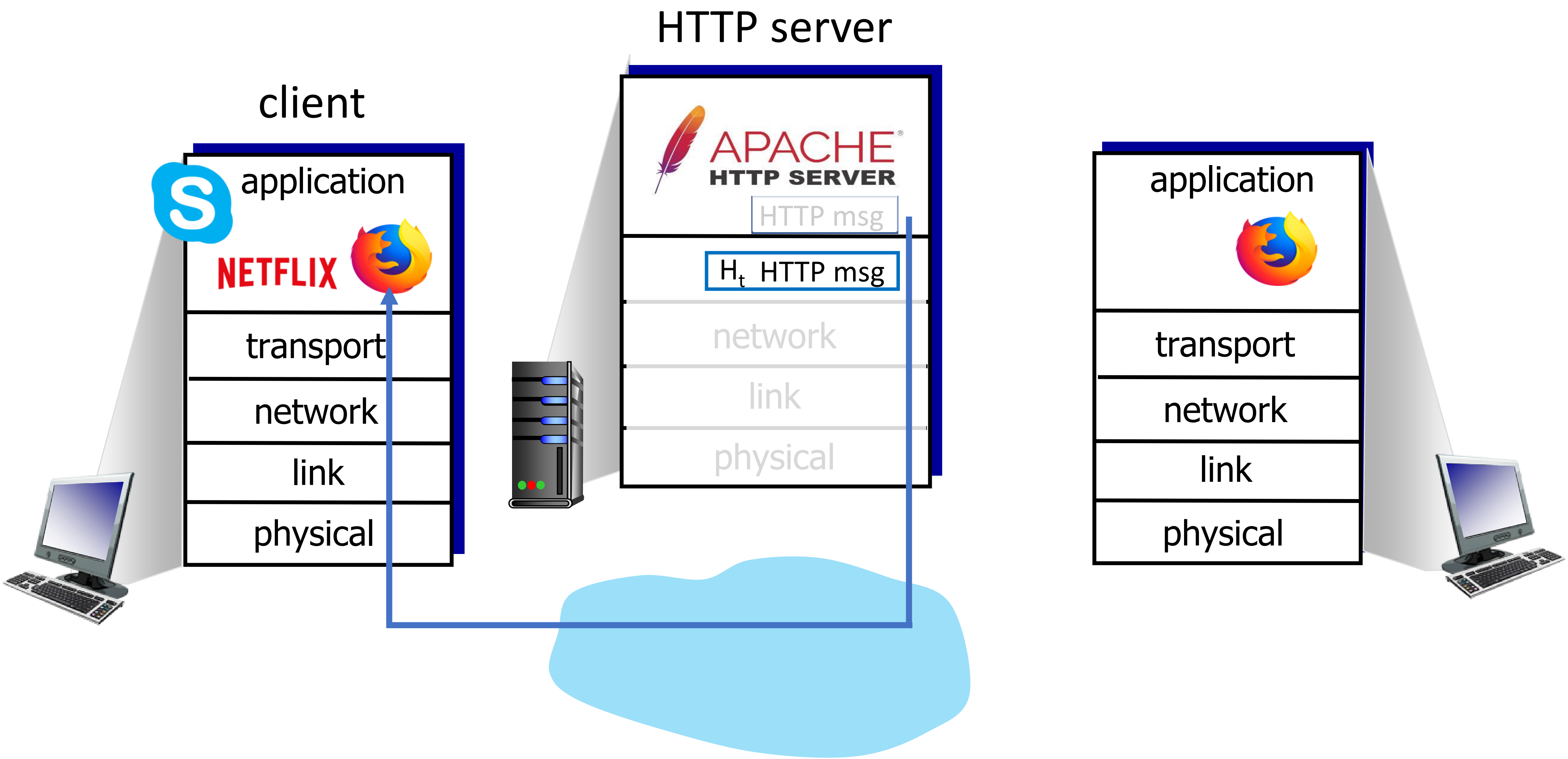


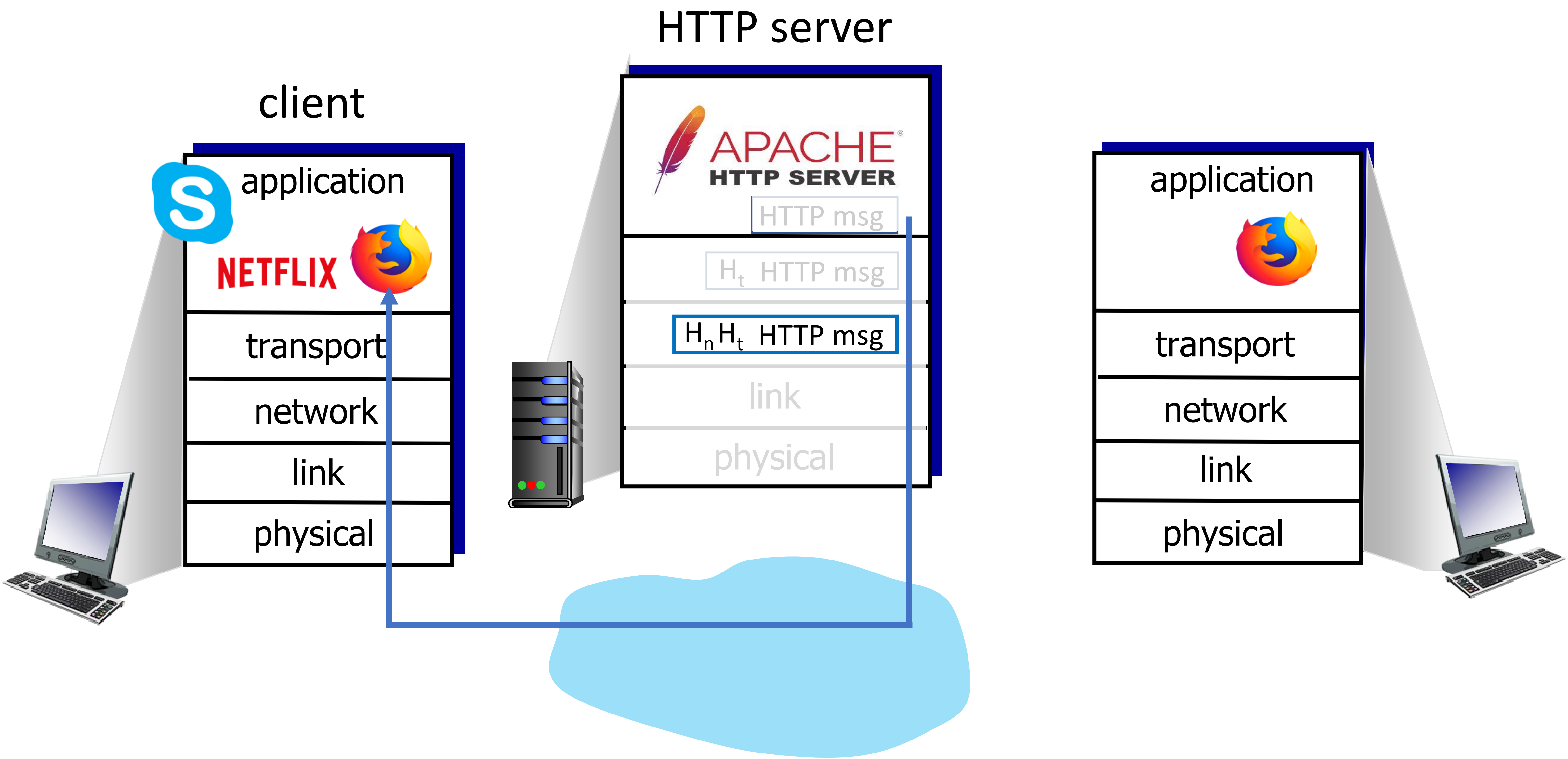
tcp
udp
No. Time Source Destination Protocol Length Info
25 4.832468360 fe80::262:ecff:fe4a… ff02::1:2 DHCPv6 167 Solicit XID: 0x06a42e CID: 000300010062ec4a8554
45 4.901463328 146.141.6.50 146.141.56.60 SNMP 85 get-request 1.3.6.1.2.1.1.1.0
46 4.901512710 146.141.56.60 146.141.6.50 ICMP 113 Destination unreachable (Port unreachable)
121 5.914290398 146.141.6.50 146.141.56.60 SNMP 85 get-request 1.3.6.1.2.1.1.1.0
122 5.914343801 146.141.56.60 146.141.6.50 ICMP 113 Destination unreachable (Port unreachable)
250 12.814881351 146.141.56.60 146.141.127.100 DNS 85 Standard query 0x92b7 A www.google.com OPT
252 12.815513248 146.141.127.100 146.141.56.60 DNS 101 Standard query response 0x92b7 A www.google.com A 172.217.170.100 OPT
337 20.617139431 fe80::221:b7ff:fea1… ff02::1:2 DHCPv6 112 Solicit XID: 0x0bb546 CID: 0001000162ee756500000000d0cc
347 34.202361833 146.141.56.52 146.141.56.255 BROWSER 271 Local Master Announcement GEOFF, Workstation, Server, Print Queue Server, Xenix Server, NT Workstation, NT Server, Master Browser, DFS server
348 34.202362749 146.141.56.52 146.141.56.255 BROWSER 248 Domain/Workgroup Announcement WORKGROUP, NT Workstation, Domain Enum
353 52.560589502 fe80::262:ecff:fe80… ff02::1:2 DHCPv6 167 Solicit XID: 0x67759f CID: 000300010062ec8083c6
368 80.185698271 146.141.56.60 146.141.127.100 DNS 99 Standard query 0x25e5 A contile.services.mozilla.com OPT
369 80.186011378 146.141.56.60 146.141.127.100 DNS 99 Standard query 0x1ec0 AAAA contile.services.mozilla.com OPT
370 80.186331390 146.141.127.100 146.141.56.60 DNS 115 Standard query response 0x25e5 A contile.services.mozilla.com A 34.117.237.239 OPT
371 80.186571253 146.141.127.100 146.141.56.60 DNS 180 Standard query response 0x1ec0 AAAA contile.services.mozilla.com SOA ns-679.awsdns-20.net OPT
401 84.377795978 146.141.56.60 146.141.127.100 DNS 88 Standard query 0x3783 AAAA gaia.cs.umass.edu OPT
402 84.378426622 146.141.127.100 146.141.56.60 DNS 141 Standard query response 0x3783 AAAA gaia.cs.umass.edu SOA unix1.cs.umass.edu OPT
436 109.617276694 146.141.56.60 146.141.127.100 DNS 101 Standard query 0xb269 A incoming.telemetry.mozilla.org OPT
437 109.617410708 146.141.56.60 146.141.127.100 DNS 101 Standard query 0x02a0 AAAA incoming.telemetry.mozilla.org OPT
438 109.618822579 146.141.127.100 146.141.56.60 DNS 217 Standard query response 0x02a0 AAAA incoming.telemetry.mozilla.org CNAME telemetry-incoming.r53-2.services.mozilla.com CNAME prod.ingestion-edge.prod.dataops.mozgcp
439 109.619148856 146.141.56.60 146.141.127.100 DNS 114 Standard query 0xbae9 AAAA prod.ingestion-edge.prod.dataops.mozgcp.net OPT
440 109.619709784 146.141.127.100 146.141.56.60 DNS 207 Standard query response 0xbae9 AAAA prod.ingestion-edge.prod.dataops.mozgcp.net SOA ns-cloud-b1.googledomains.com OPT
442 109.781162573 146.141.127.100 146.141.56.60 DNS 233 Standard query response 0xb269 A incoming.telemetry.mozilla.org CNAME telemetry-incoming.r53-2.services.mozilla.com CNAME prod.ingestion-edge.prod.dataops.mozgcp.ne
504 123.542277930 fe80::262:ecff:fe4a… ff02::1:2 DHCPv6 167 Solicit XID: 0x06a42e CID: 000300010062ec4a8554
509 132.144057857 fe80::221:b7ff:fea1… ff02::1:2 DHCPv6 112 Solicit XID: 0x0bb546 CID: 0001000162ee756500000000d0cc
Frame 250: 85 bytes on wire (680 bits), 85 bytes captured (680 bits) on interface enp2s0, id 0
Ethernet II, Src: Giga-Byt_a1:46:d6 (94:de:80:a1:46:d6), Dst: Cisco_f9:34:fc (00:38:df:f9:34:fc)
Internet Protocol Version 4, Src: 146.141.56.60, Dst: 146.141.127.100
User Datagram Protocol, Src Port: 54912, Dst Port: 53
Source Port: 54912
Destination Port: 53
Length: 51
Checksum: 0xdcff [unverified]
[Checksum Status: Unverified]
[Stream index: 3]
[Timestamps]
UDP payload (43 bytes)
Domain Name System (query)
0000 00 38 df f9 34 fc 94 de 80 a1 46 d6 08 00 45 00
0010 00 47 a2 12 40 00 40 11 bb d8 92 8d 38 3c 92 8d
0020 7f 64 d6 80 00 35 00 33 dc ff 92 b7 01 00 00 01
0030 00 00 00 00 00 01 03 77 77 77 06 67 6f 6f 67 6c
0040 65 03 63 6f 6d 00 00 01 00 01 00 00 29 02 00 00
0050 00 00 00 00 00
No. Tim
251 12
253 12
254 12
255 12
256 12
257 12
258 12
259 12
260 12
261 12
262 12
263 12
264 13
265 13
266 13
267 13
268 13
269 13
270 13
271 13
272 13
273 13
274 13
275 13
276 13
Frame 251
Ethernet
Internet
Transmiss
Source
Destin
[Strea
[Conve
[TCP S
Sequen
Sequen
[Next
Acknow
Acknow

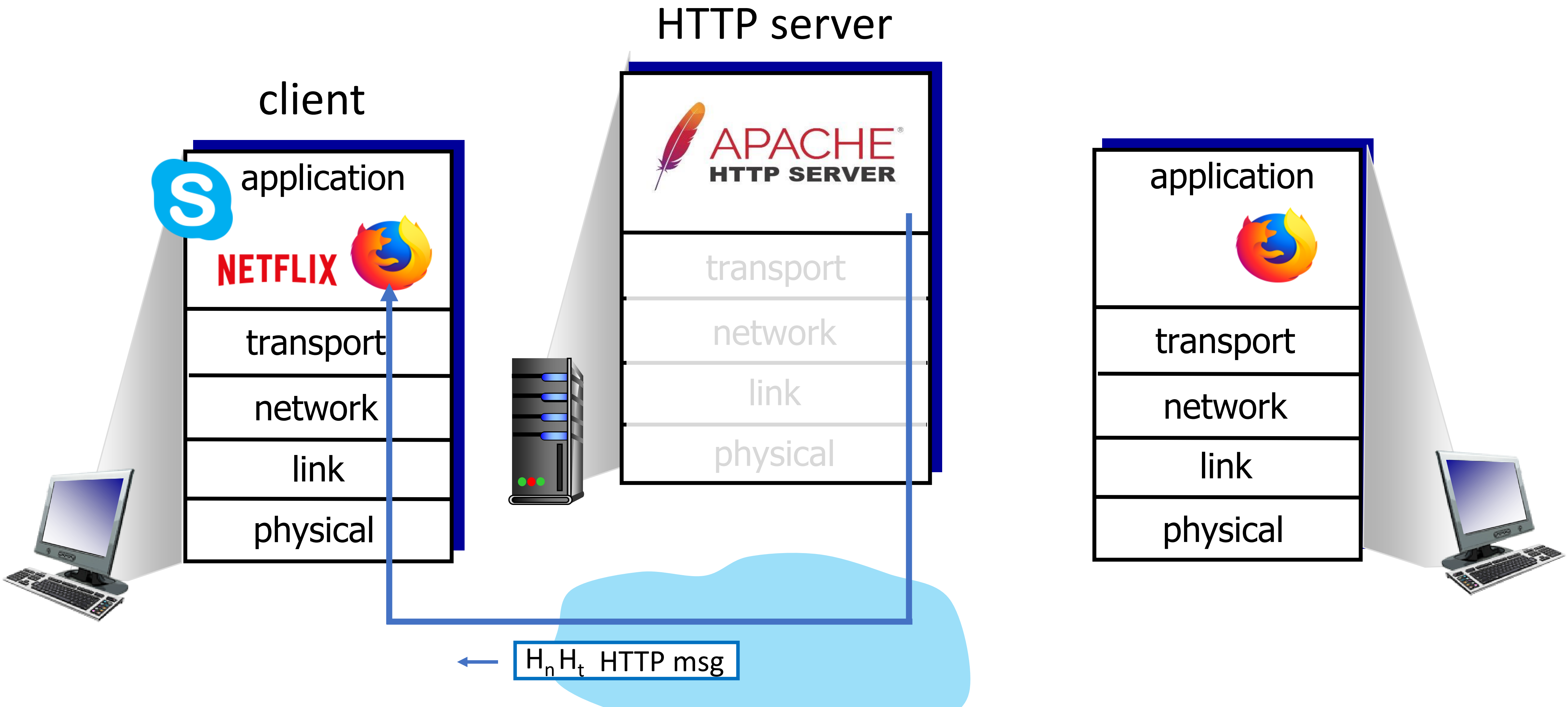
Chapter 3: roadmap

- 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

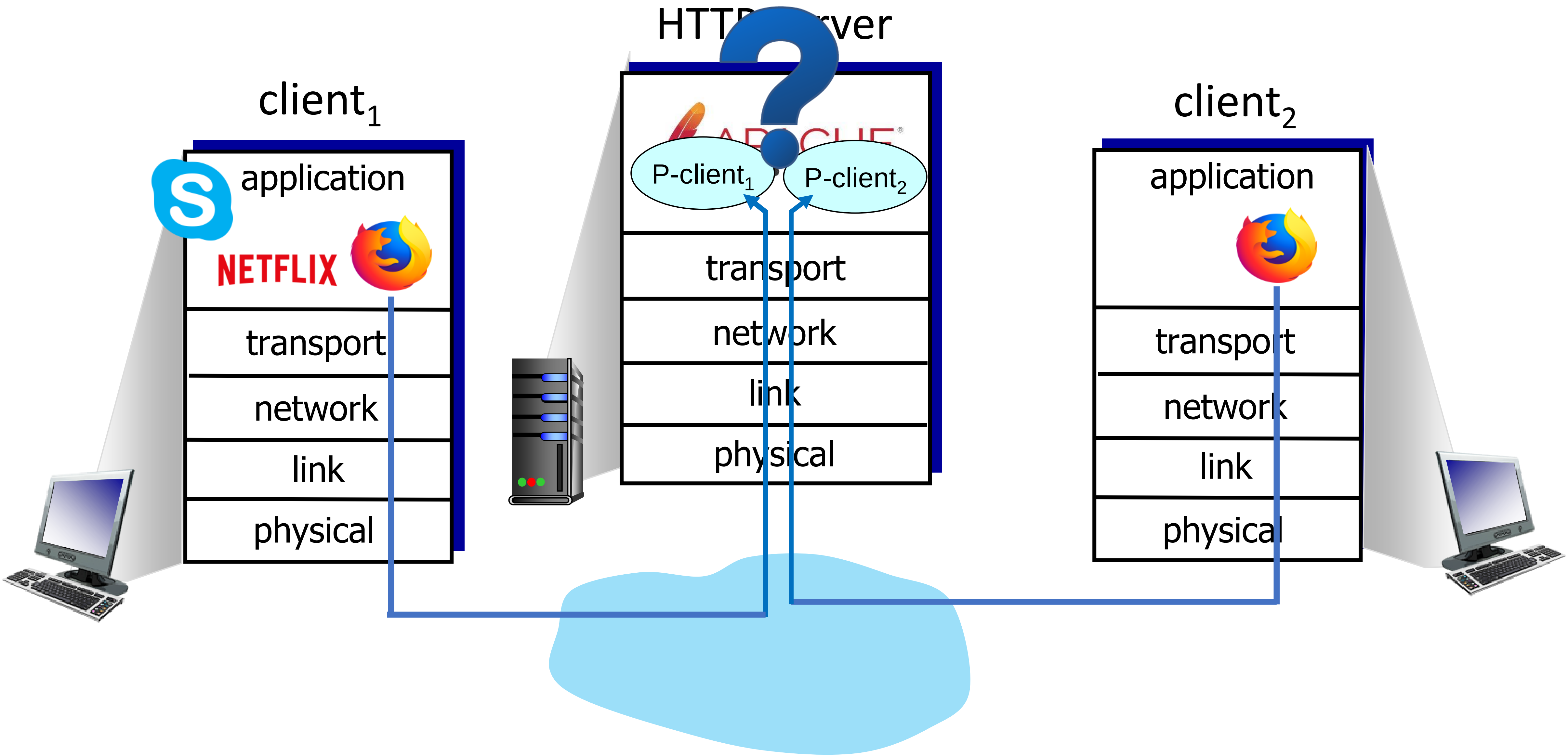

HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
transport
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
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



transport-layer multiplexing requires (1) that sockets have unique identifiers, and (2) that each segment have special fields that indicate the socket to which the segment is to be delivered



HTTP
rver
client$_1$
application
NETFLIX
transport
network
link
physical
P-client$_1$
P-client$_2$
transport
network
link
physical
client$_2$
application
transport
network
link
physical

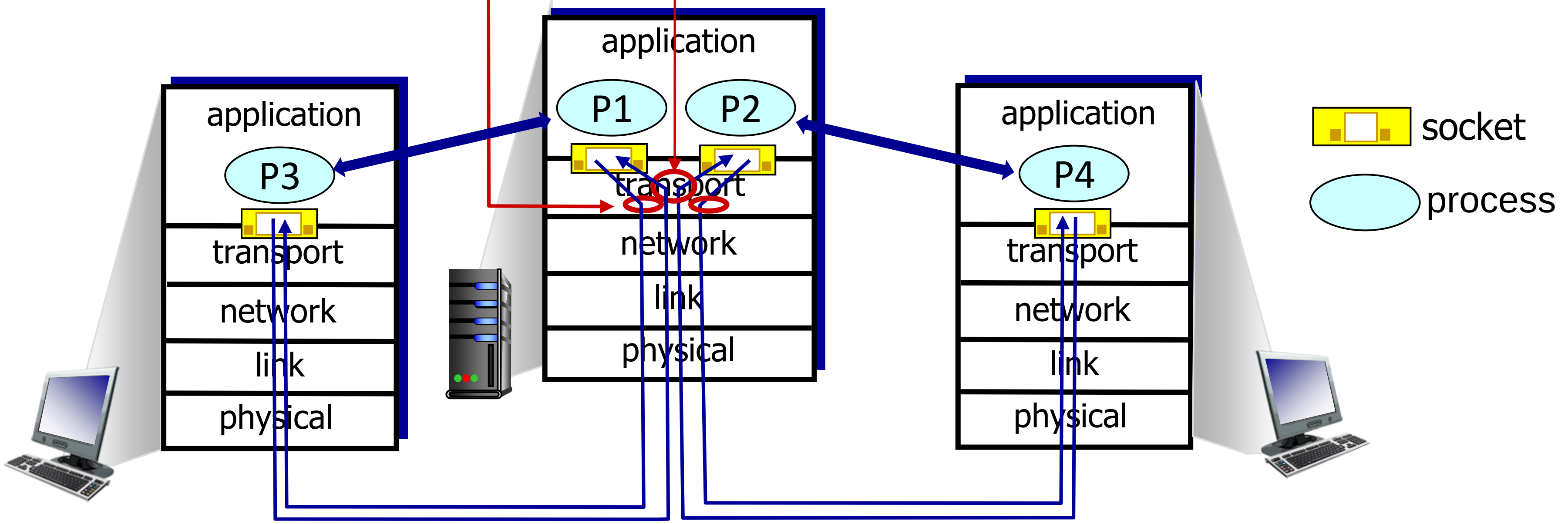
Multiplexing/demultiplexing

multiplexing at sender:

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

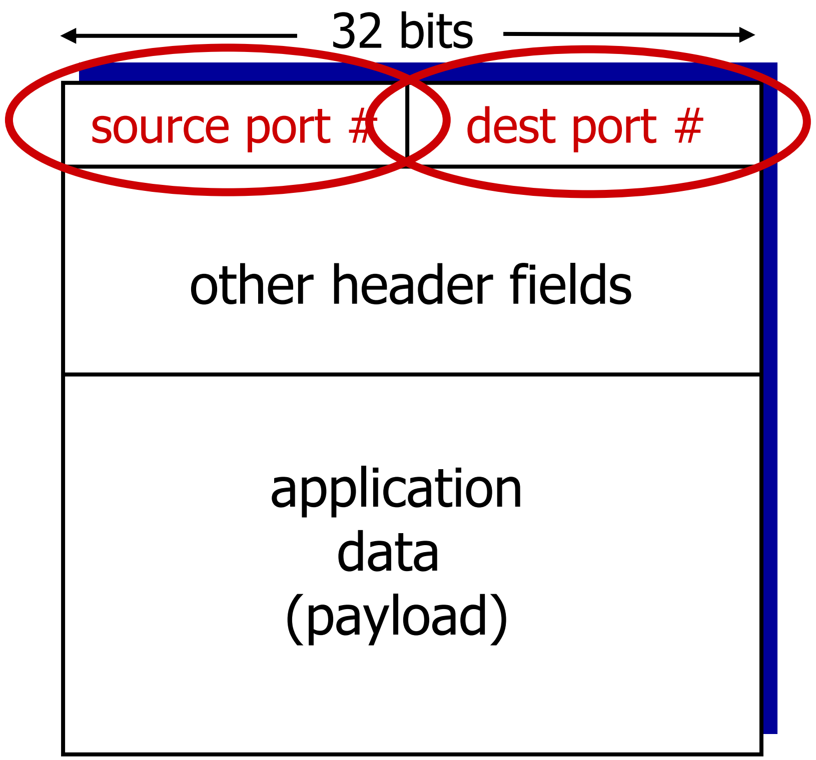
demultiplexing at receiver:

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);
```

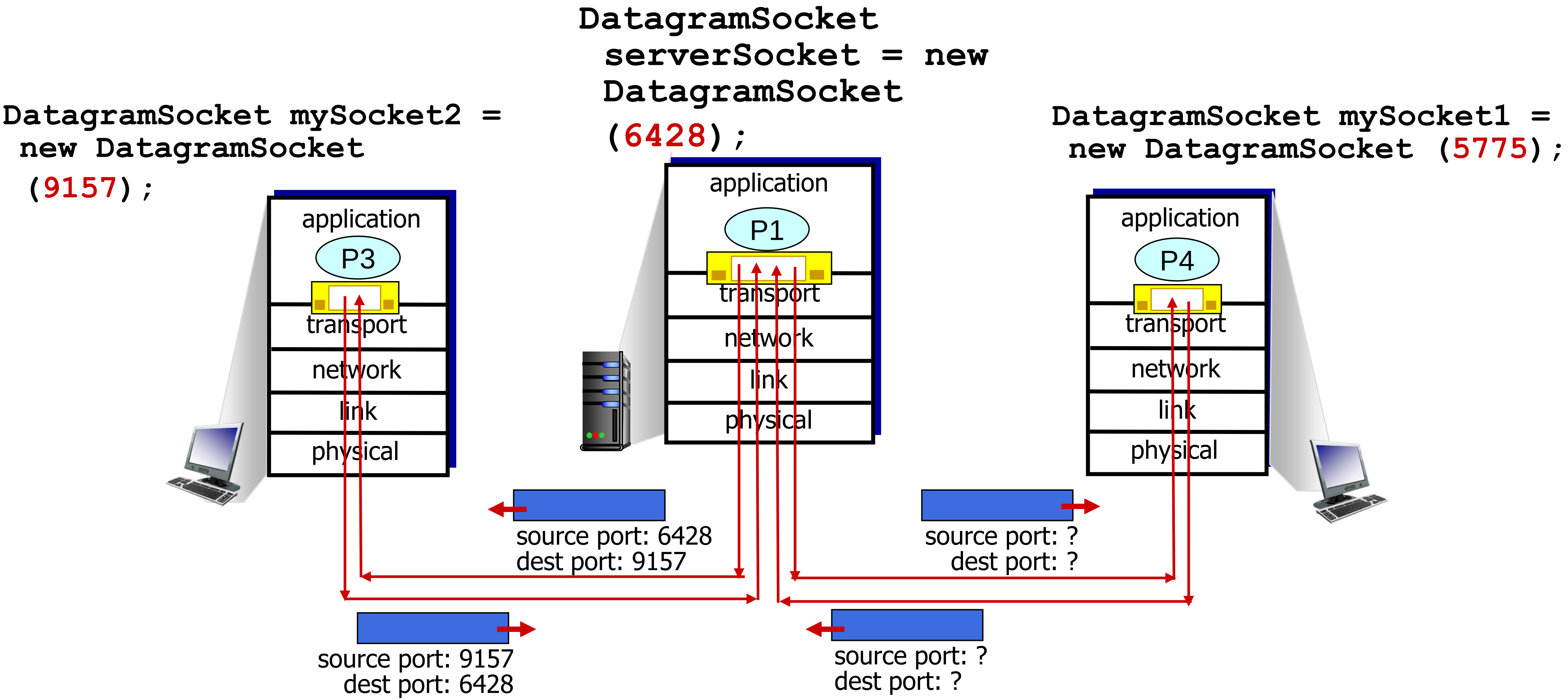
- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #
  - Range **1024 - 65535**

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



Quick overview UDP Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example Tutorial 1:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming with UDP

UDP: no "connection" between client and server:

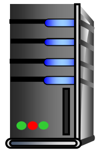
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server processes

Client/server socket interaction: UDP



server (running on serverIP)

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

Example app: UDP server

Python UDPServer

create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)

bind socket to local port number 12000 → serverSocket.bind(('', serverPort))

loop forever → while True:

Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)

send upper case string back to this client → serverSocket.sendto(modifiedMessage.encode(), clientAddress)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

Example app: UDP client

Python UDPClient

include Python's socket library → `from socket import *`

```
serverName = 'hostname'
serverPort = 12000
```

create UDP socket for server → `clientSocket = socket(AF_INET, SOCK_DGRAM)`

get user keyboard input → `message = raw_input('Input lowercase sentence:')`

attach server name, port to message; send into socket → `clientSocket.sendto(message.encode(), (serverName, serverPort))`

read reply characters from socket into string → `modifiedMessage, serverAddress = clientSocket.recvfrom(2048)`

print out received string and close socket → `print modifiedMessage.decode()`

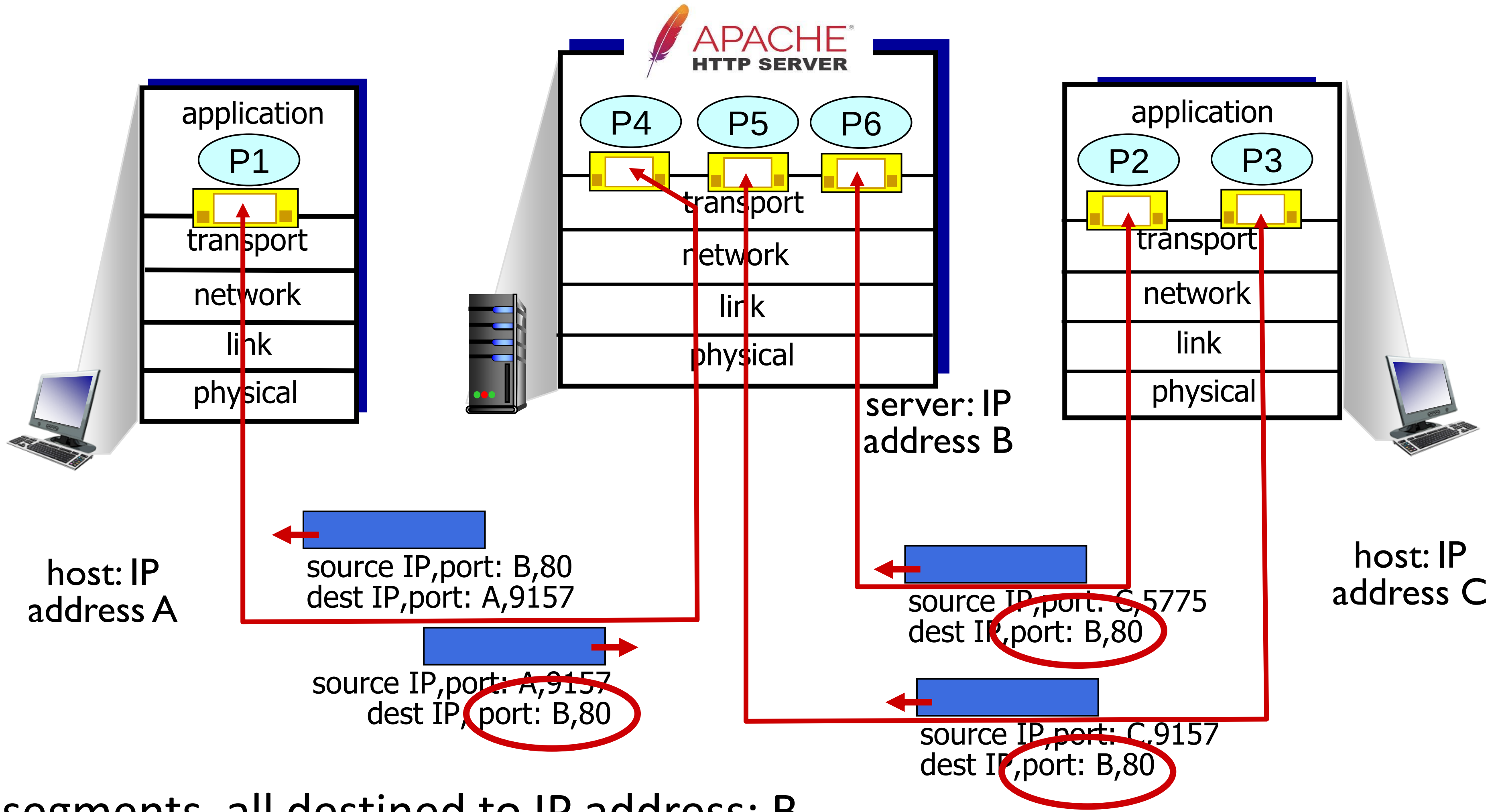
```
clientSocket.close()
```

Connection-oriented demultiplexing

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

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

Connection-oriented demultiplexing: example



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

Summary

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

Questions ?

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

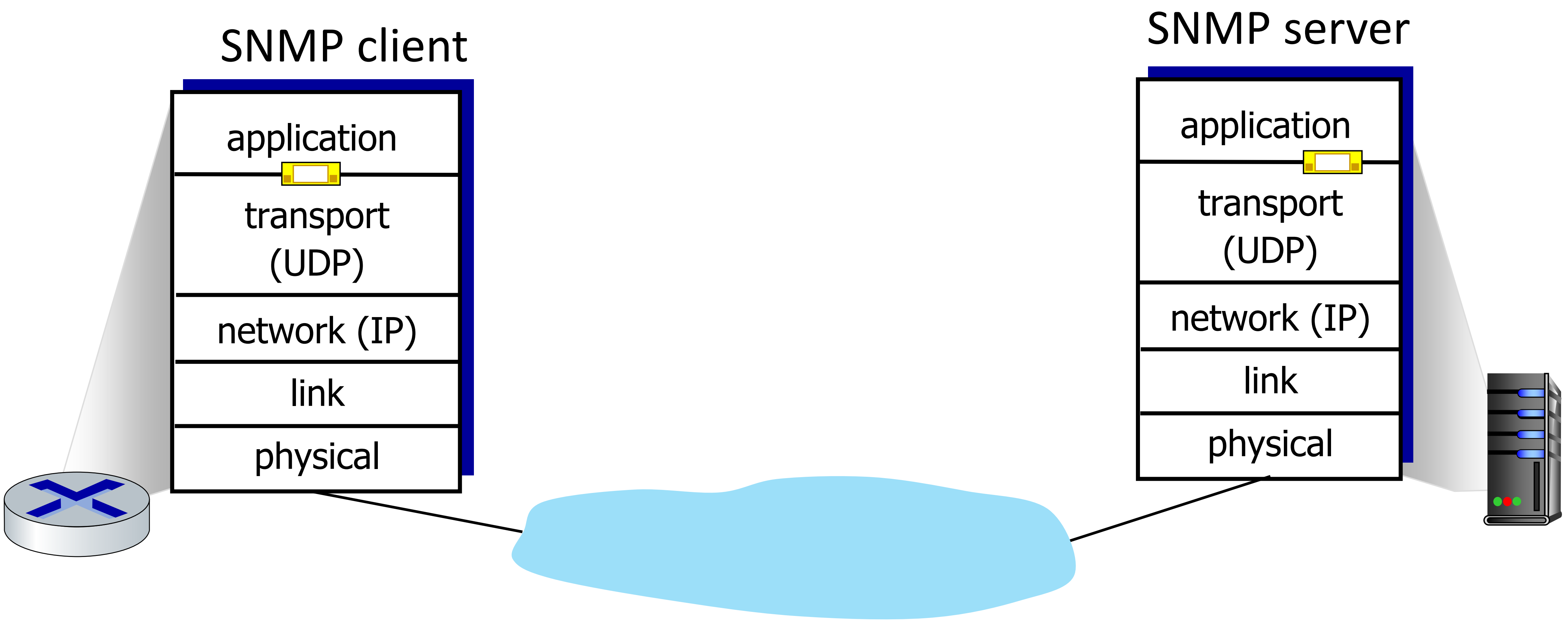
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

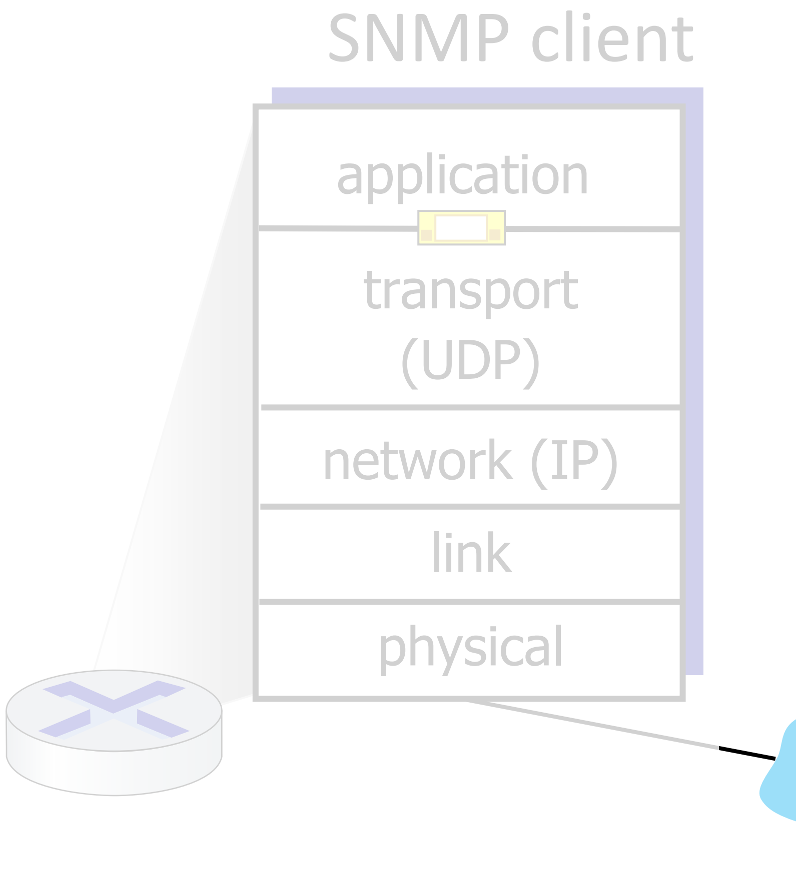
Format

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

UDP: Transport Layer Actions

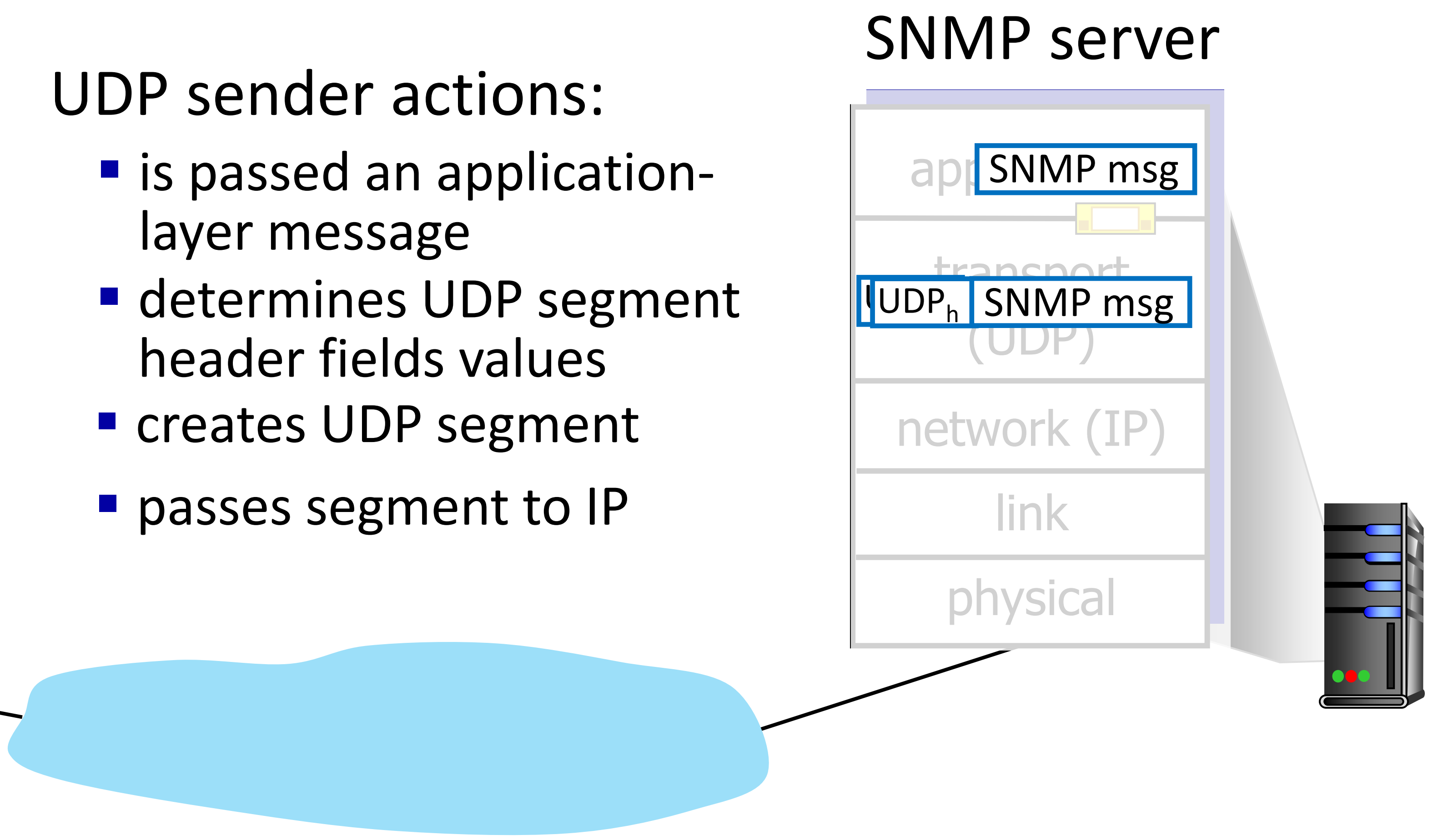


UDP: Transport Layer Actions

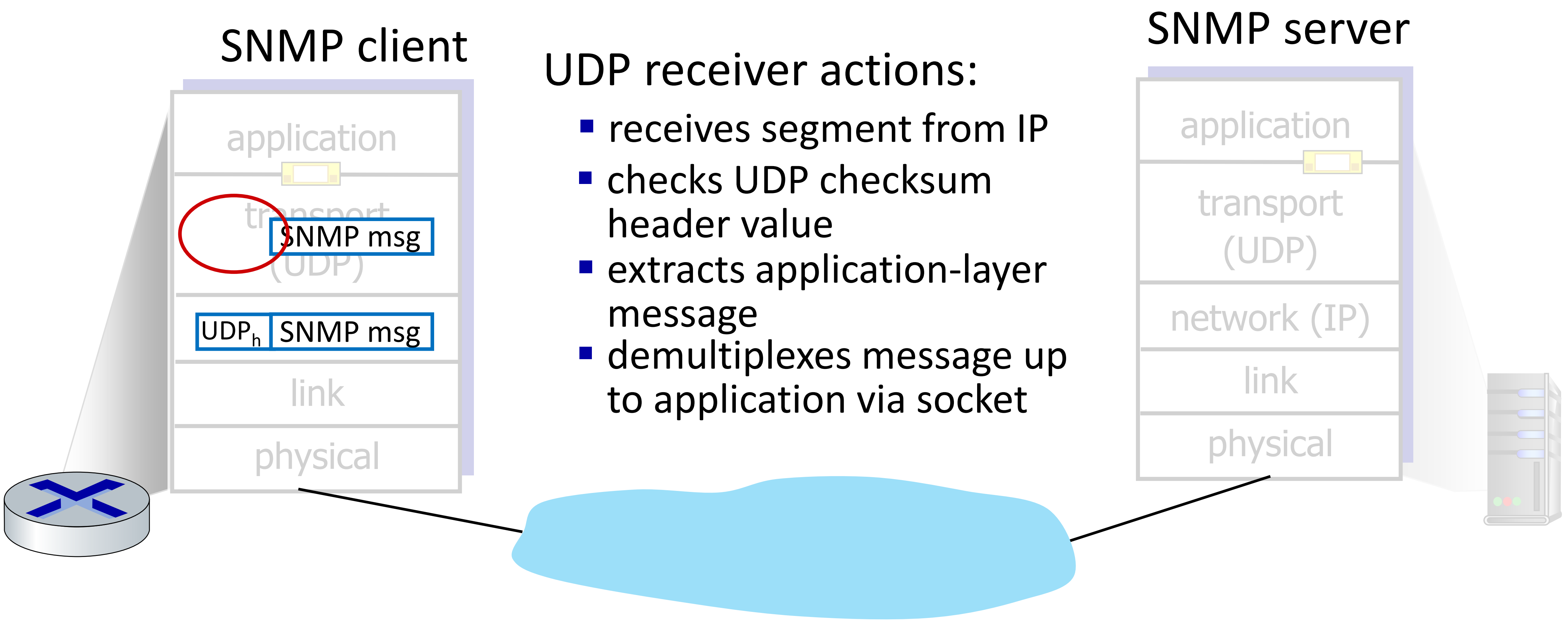


UDP sender actions:

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



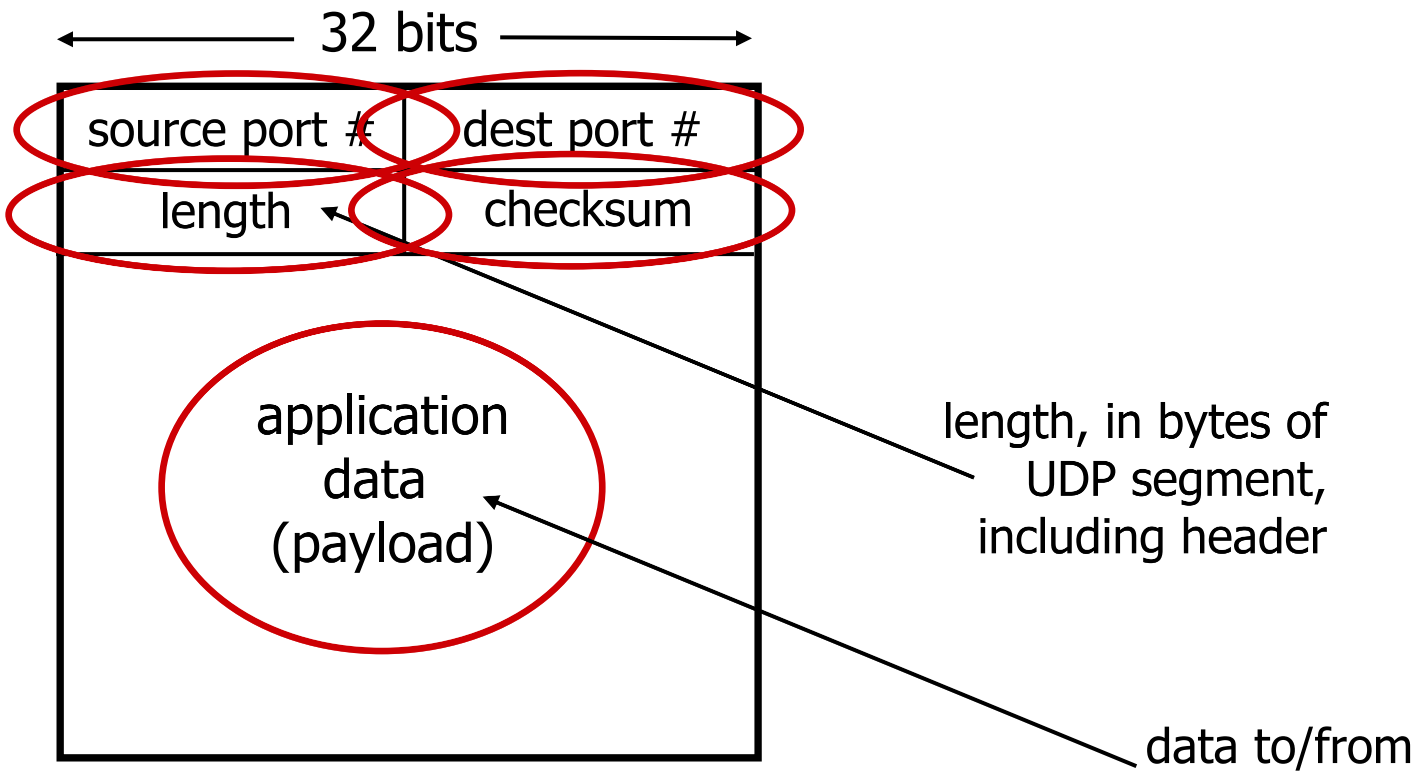
UDP: Transport Layer Actions



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

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

receiver-computed checksum $\neq$ 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
- 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

* 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!

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)

Questions ?

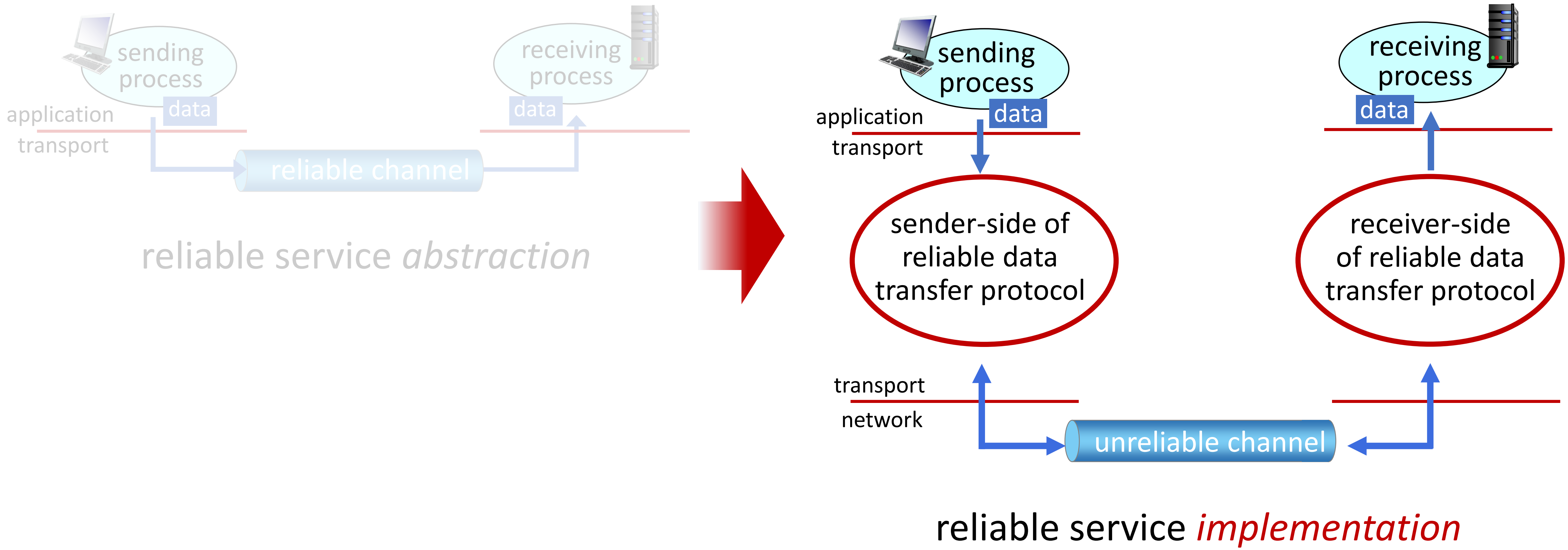
Chapter 3: roadmap

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

Principles of reliable data transfer

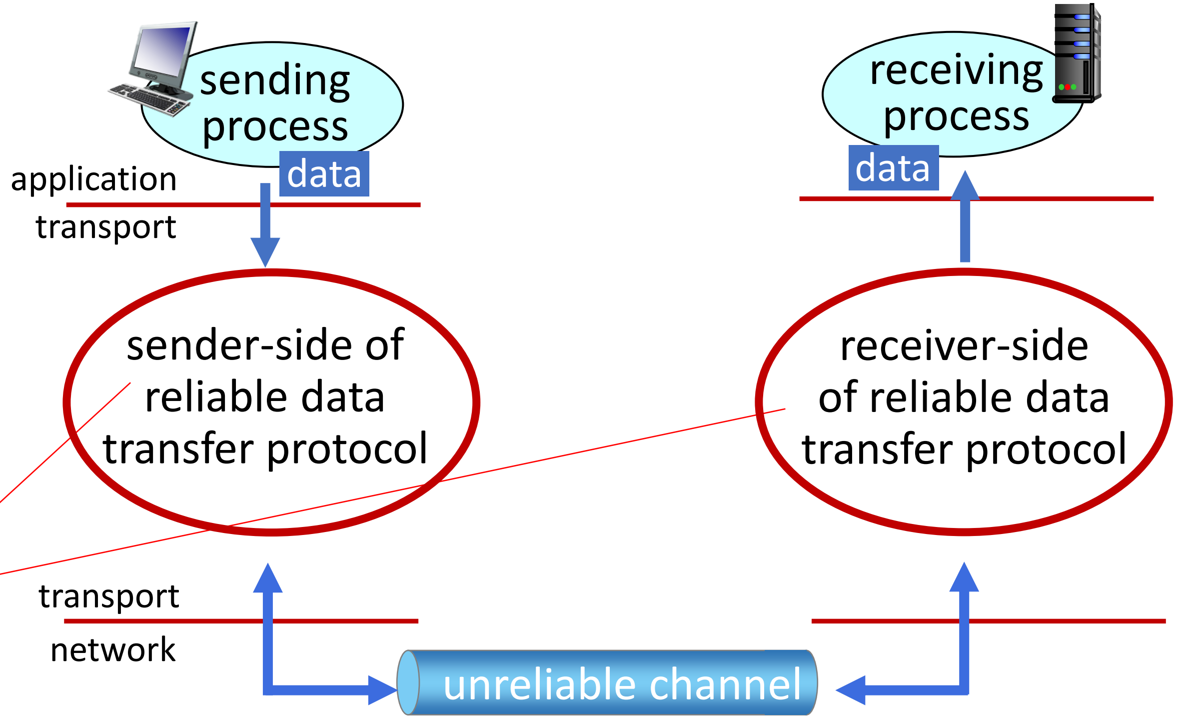
reliable service *abstraction*

Principles of reliable data transfer



Principles of reliable data transfer

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

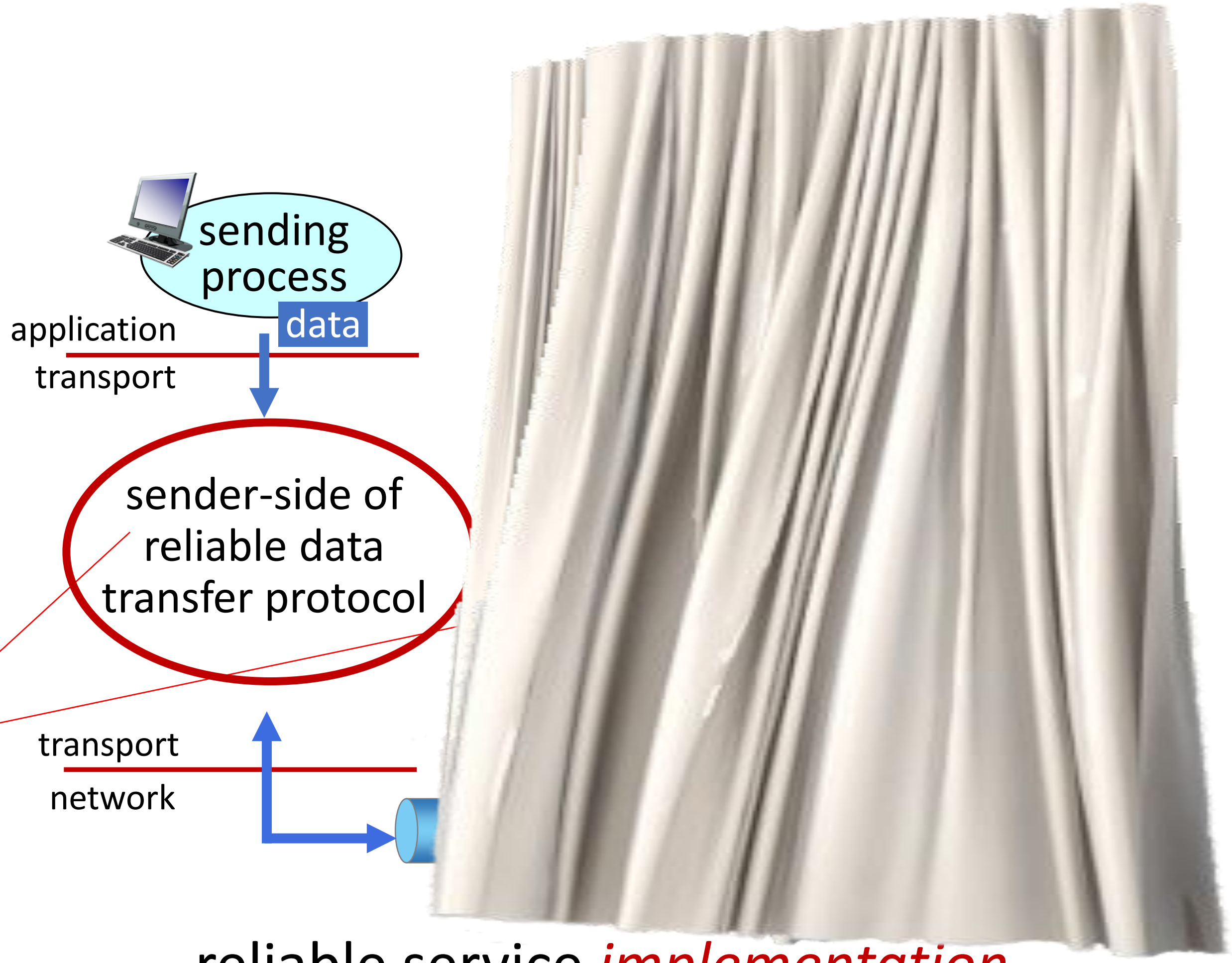


reliable service *implementation*

Principles of reliable data transfer

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

- unless communicated via a message

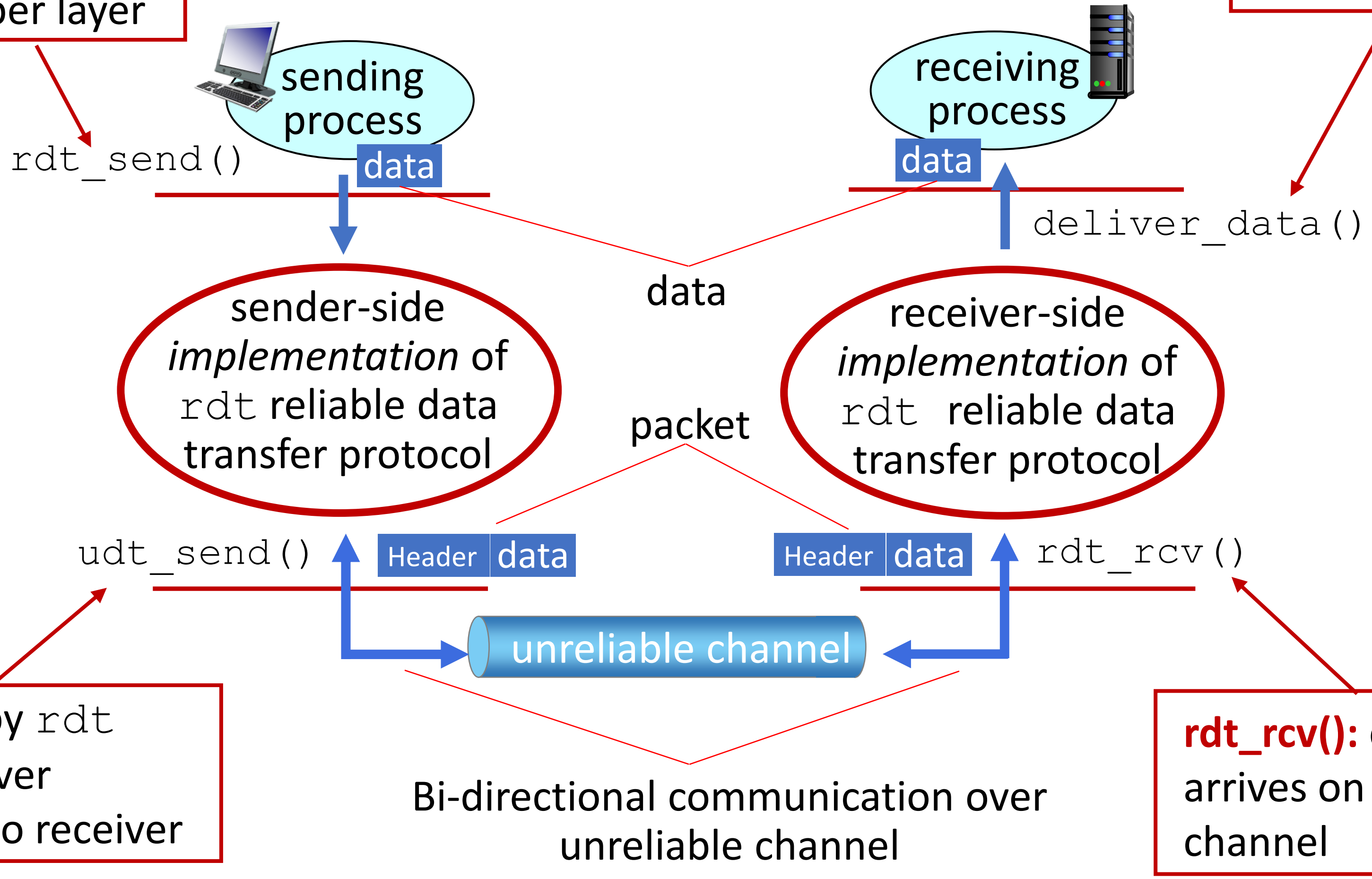


reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by `rdt` to deliver data to upper layer



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

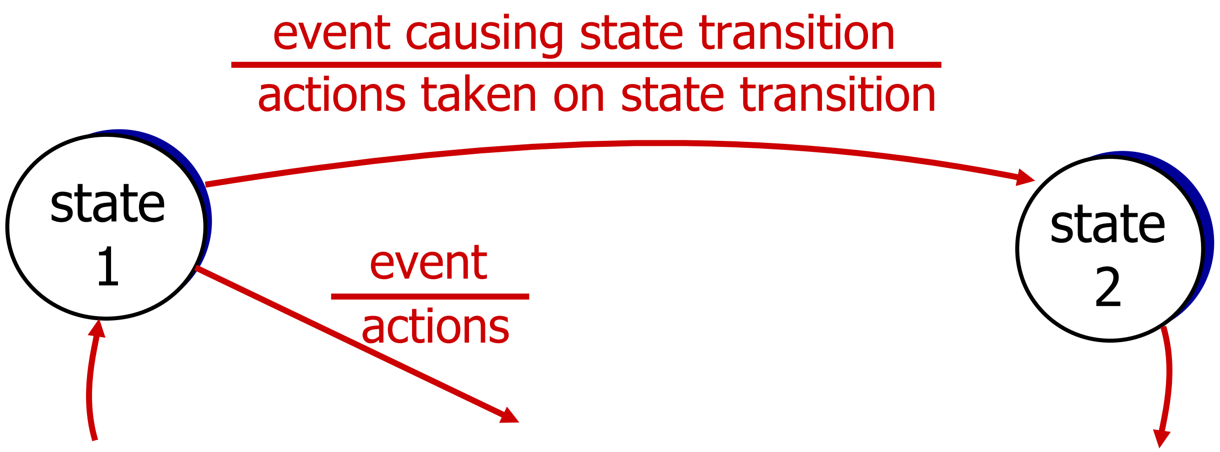
rdt_rcv(): called when packet arrives on receiver side of channel

Reliable data transfer: getting started

We will:

- incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver

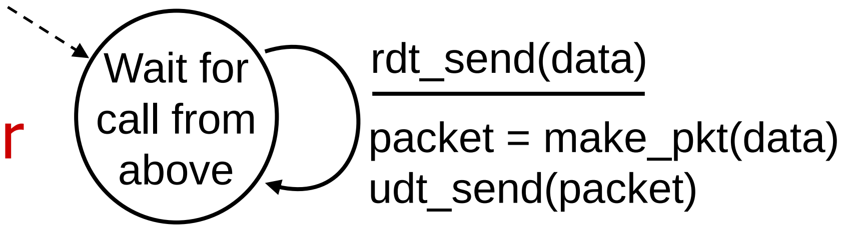
state: when in this "state" next state uniquely determined by next event



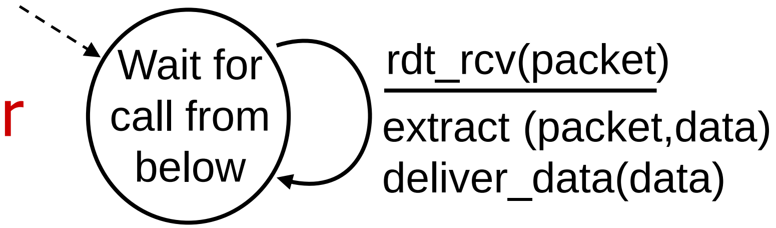
rdt1.0: reliable transfer over a reliable channel

- 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

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

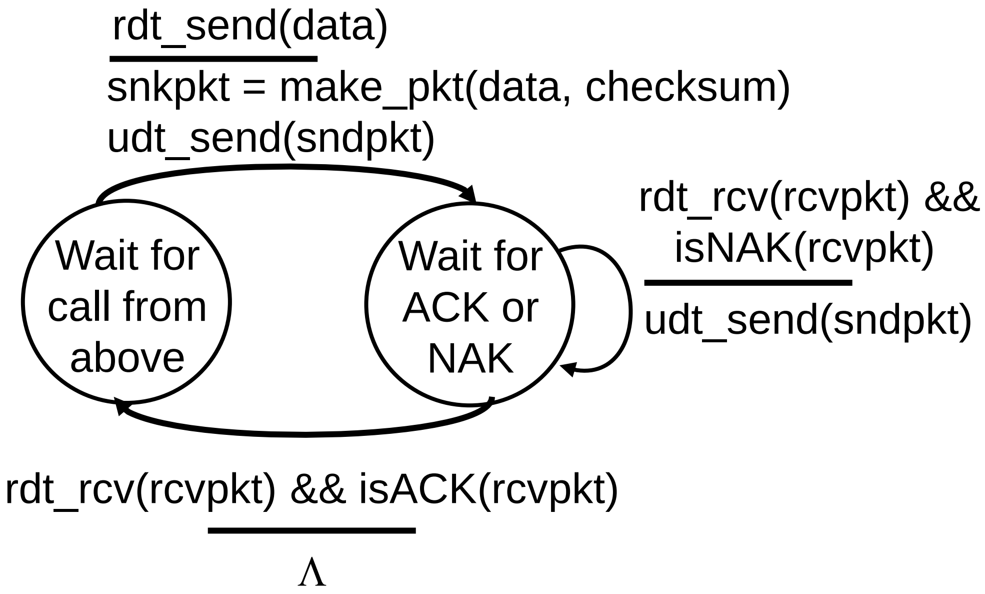
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors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

stop and wait

sender sends one packet, then waits for receiver response

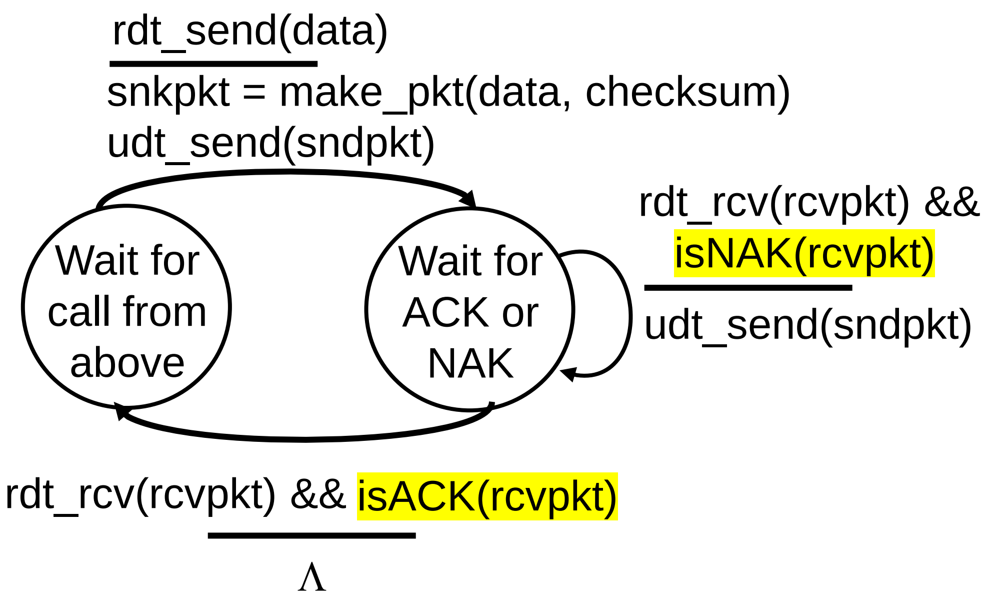
rdt2.0: FSM specifications

sender



rdt2.0: FSM specification

sender



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

- that’s why we need a protocol!

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

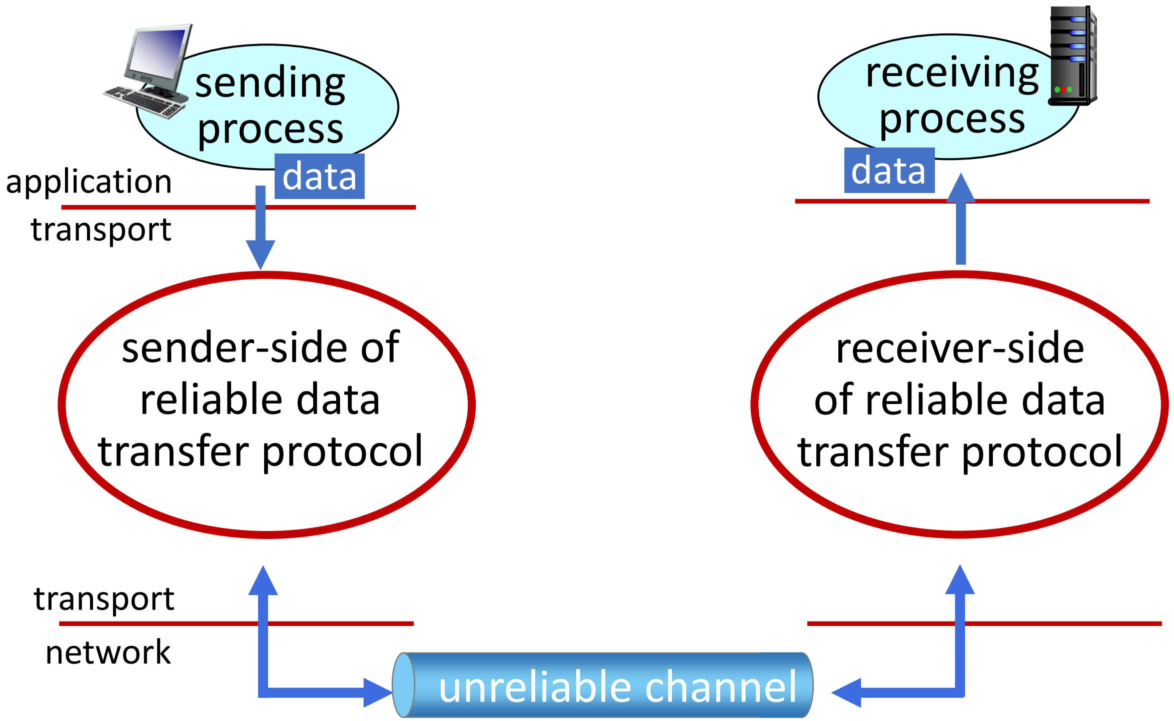
Questions ?

Principles of reliable data transfer

reliable service *abstraction*

Principles of reliable data transfer

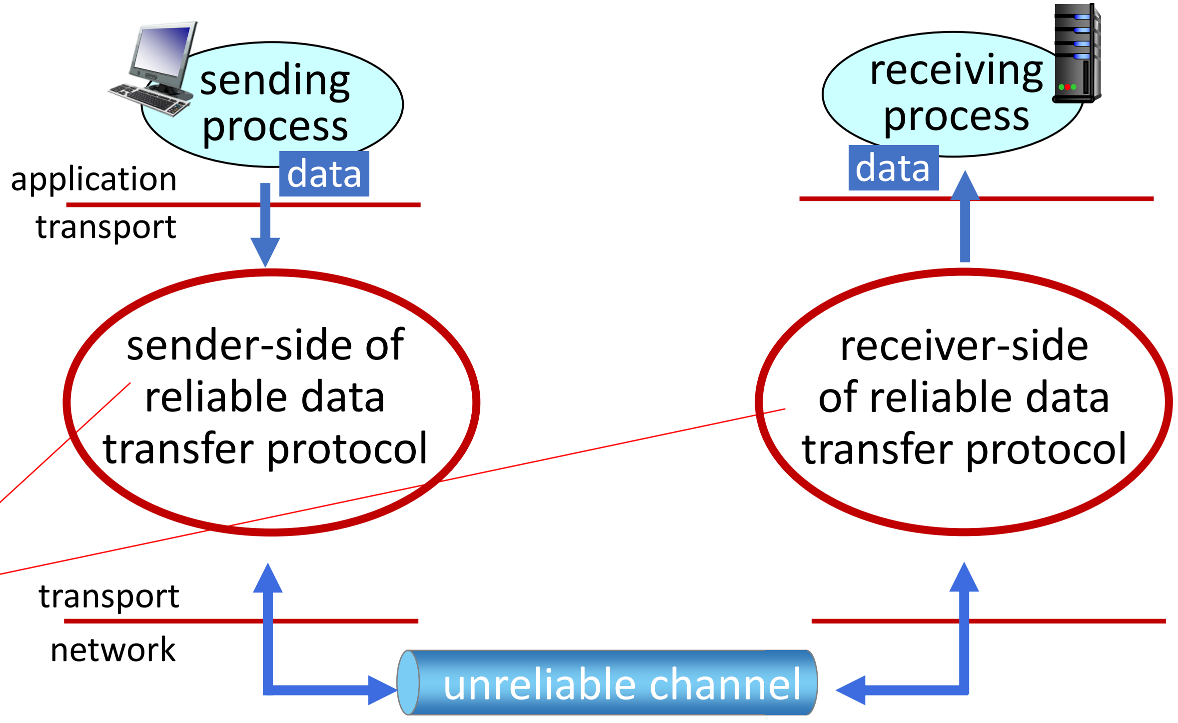
reliable service *abstraction*



reliable service *implementation*

Principles of reliable data transfer

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

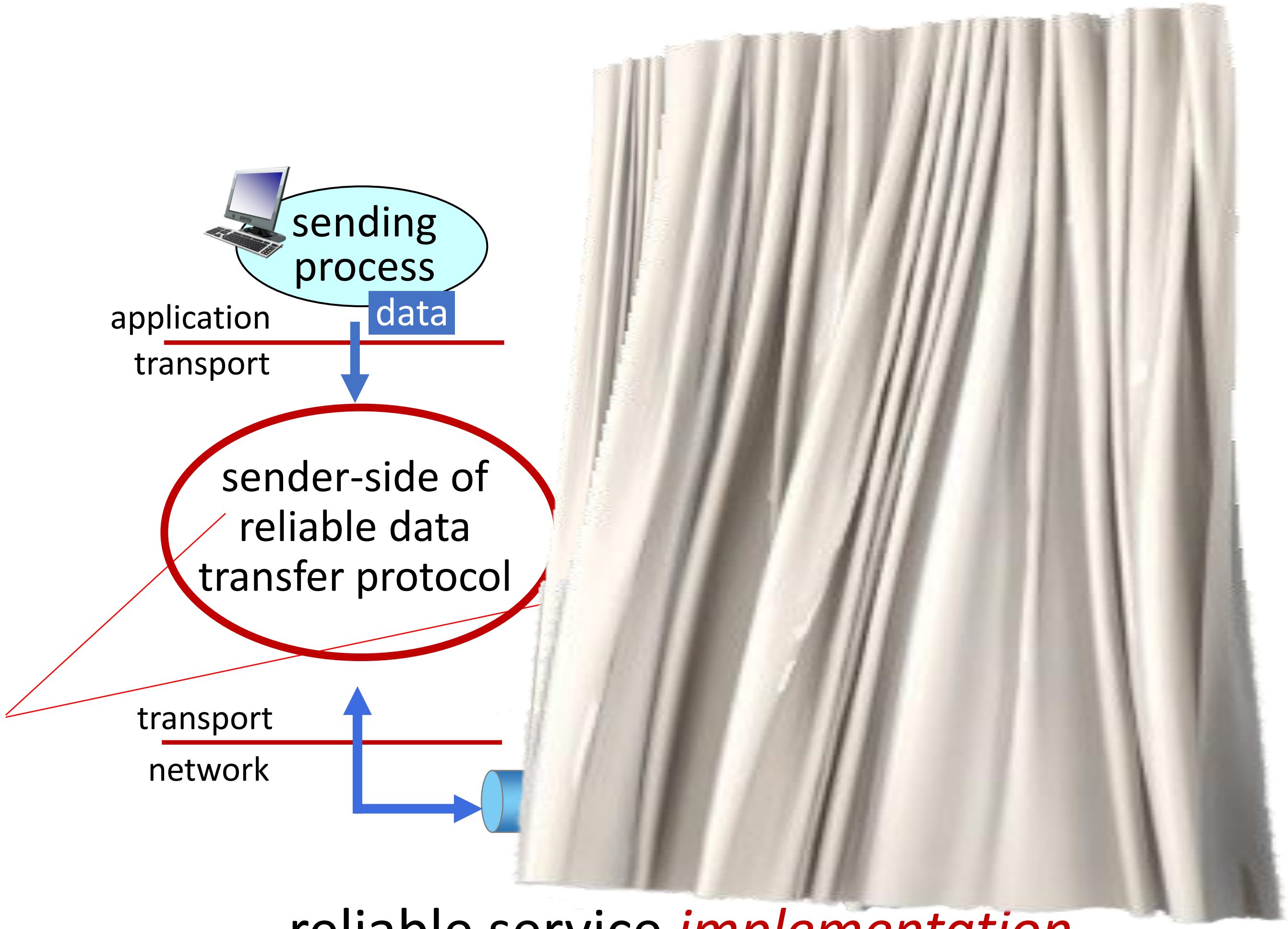


reliable service *implementation*

Principles of reliable data transfer

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

- unless communicated via a message

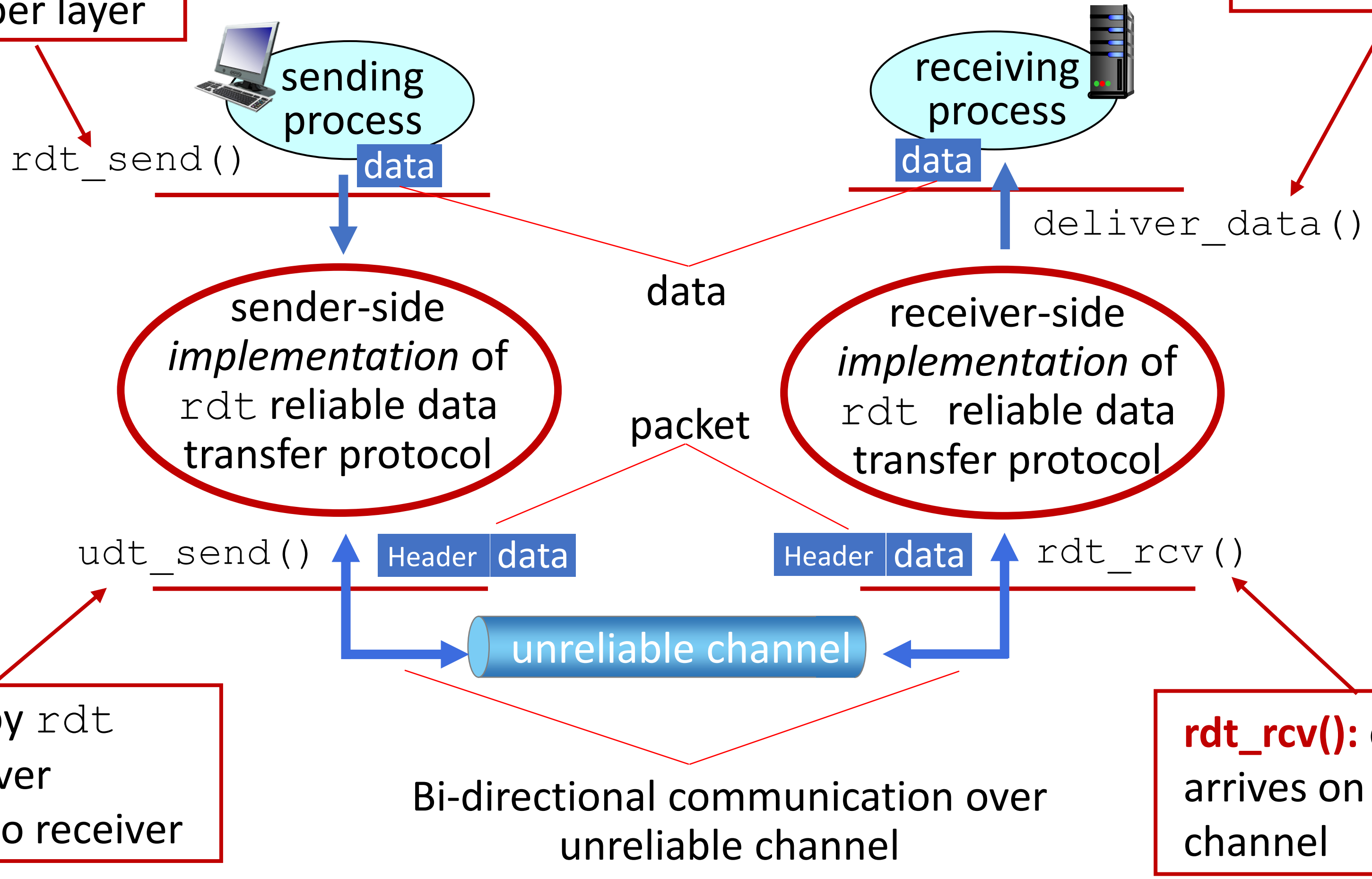


reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by `rdt` to deliver data to upper layer



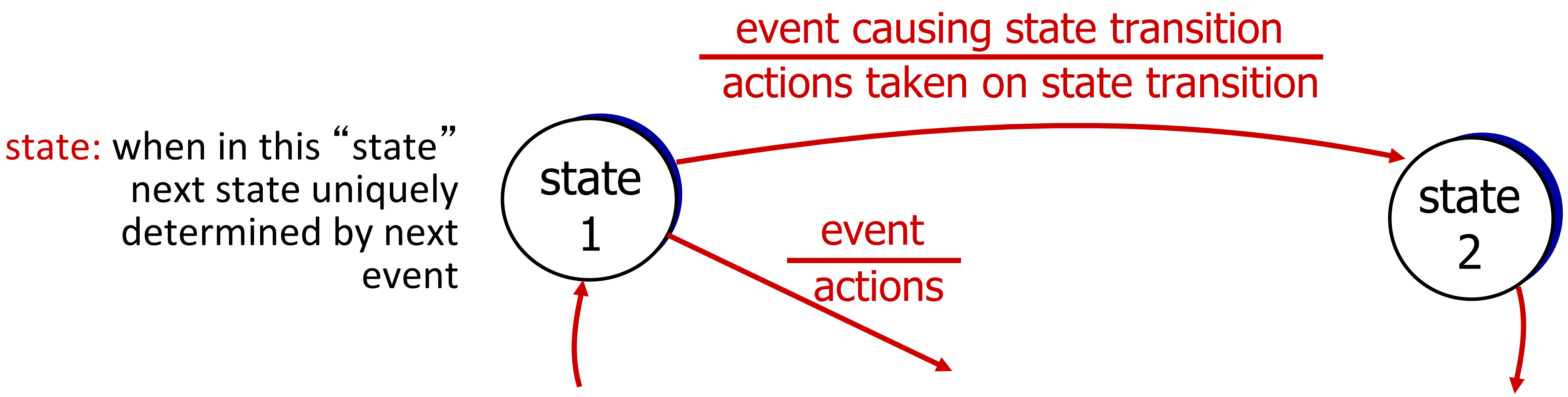
udt_send(): called by `rdt` to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

We will:

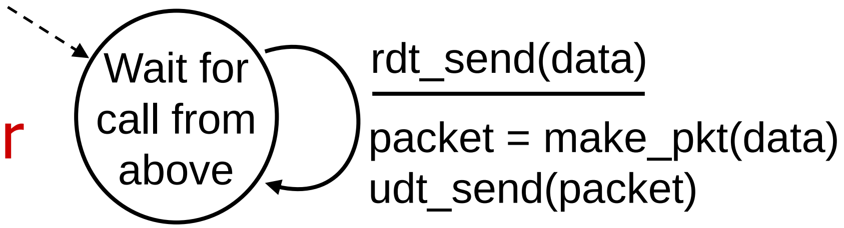
- incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer protocol (`rdt`)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



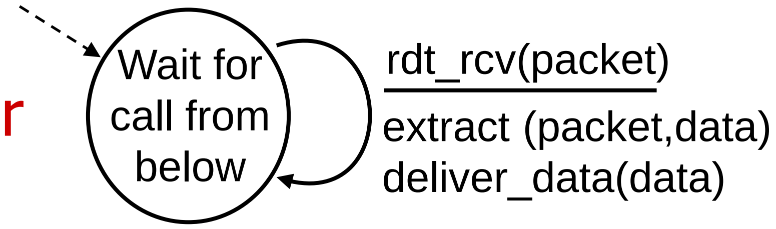
rdt1.0: reliable transfer over a reliable channel

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

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

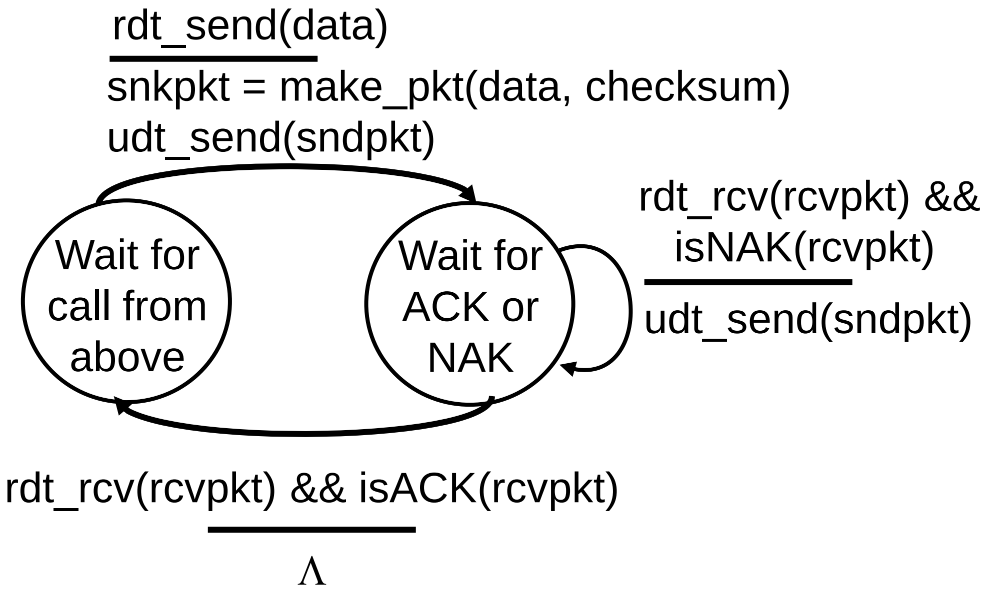
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors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

stop and wait

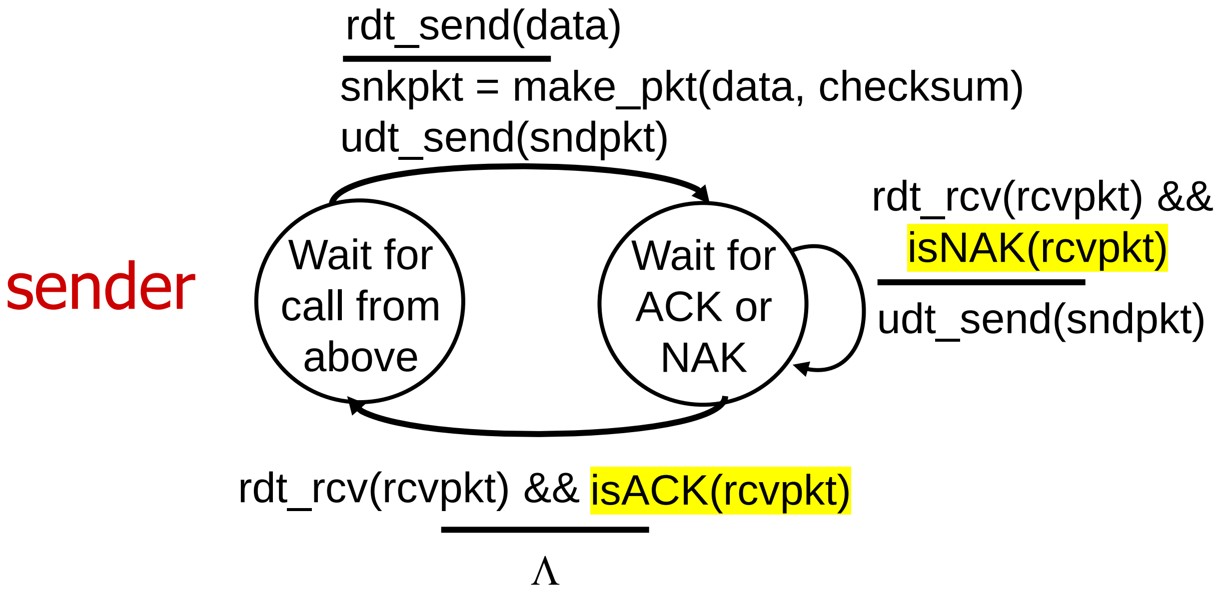
sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications

sender



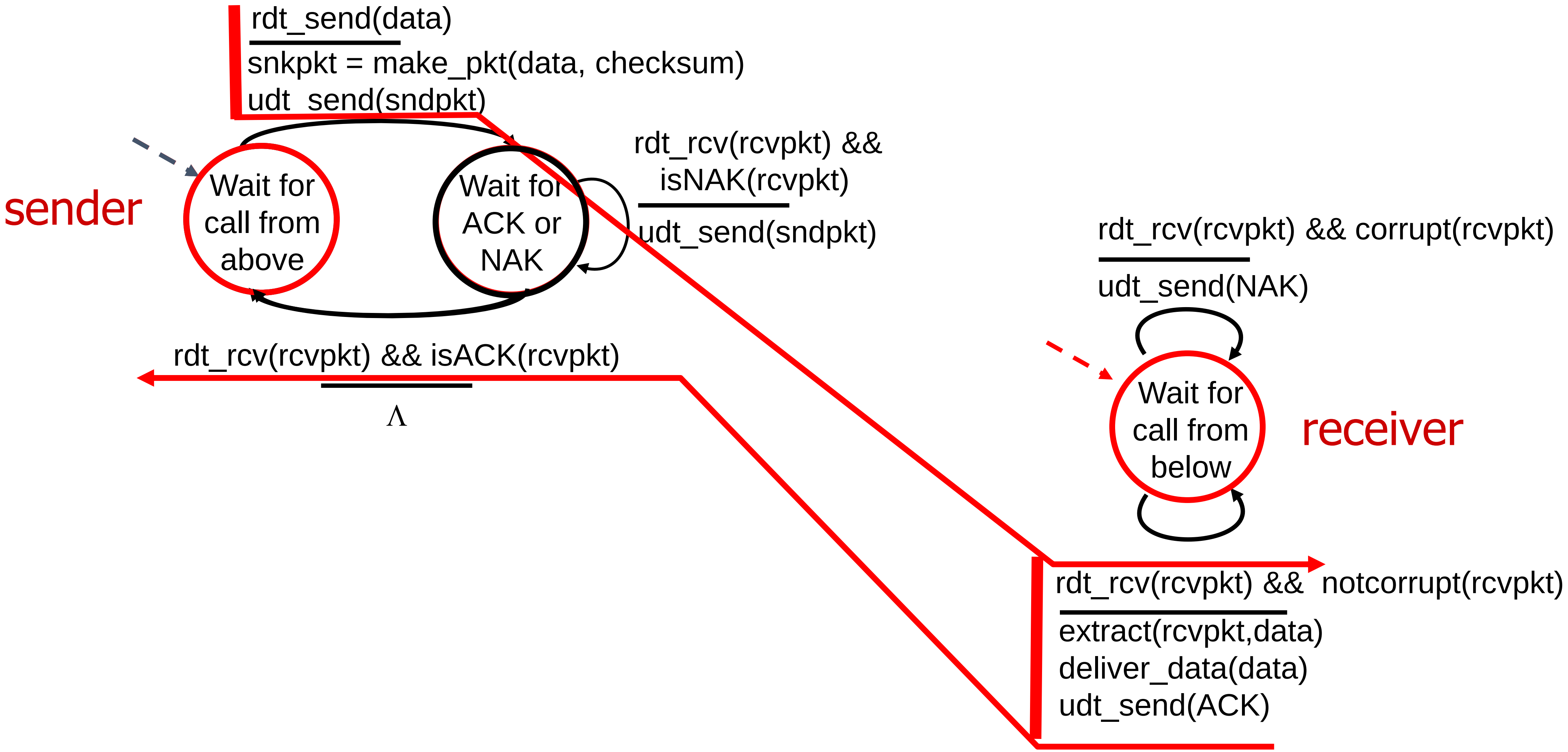
rdt2.0: FSM specification



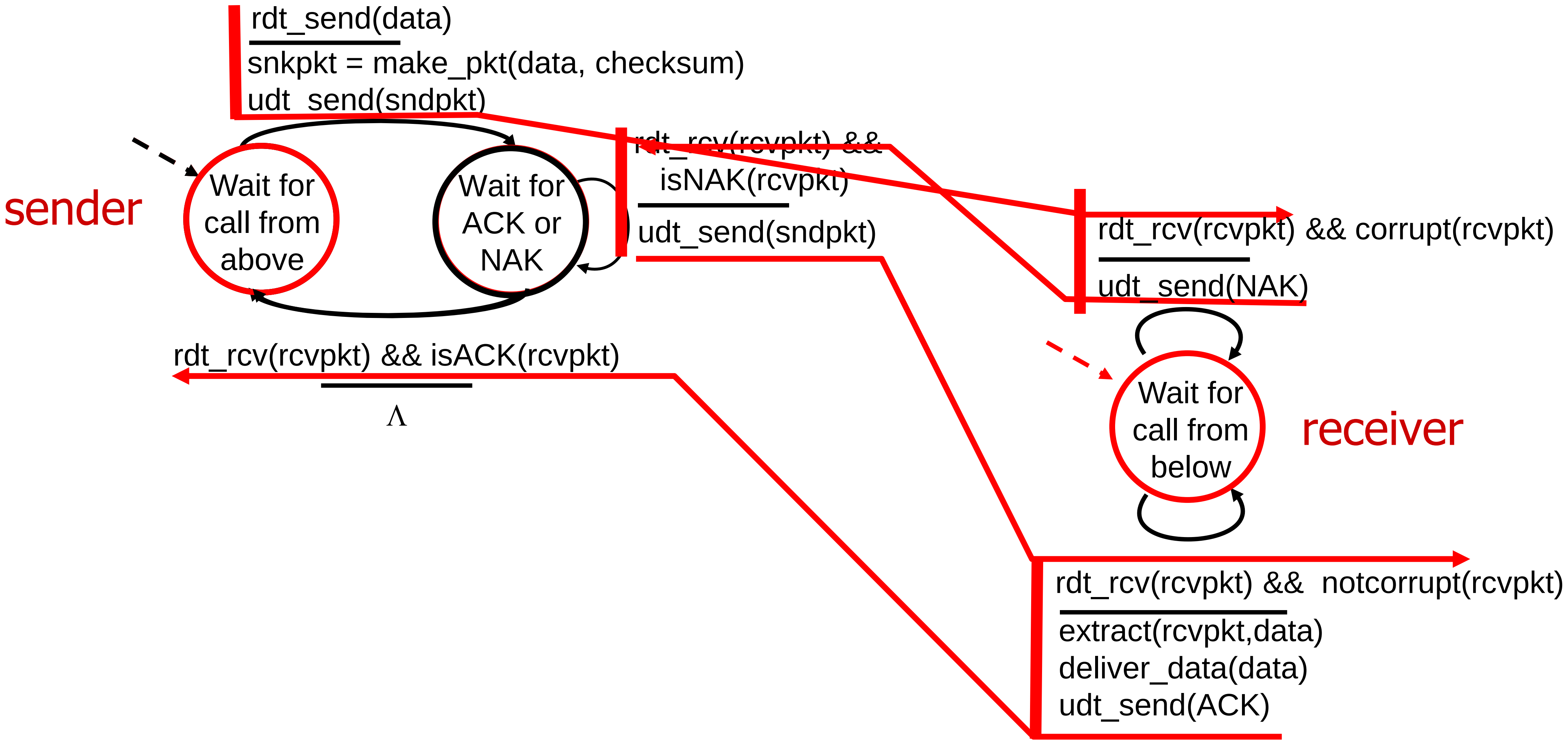
Note: “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender

- that’s why we need a protocol!

rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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

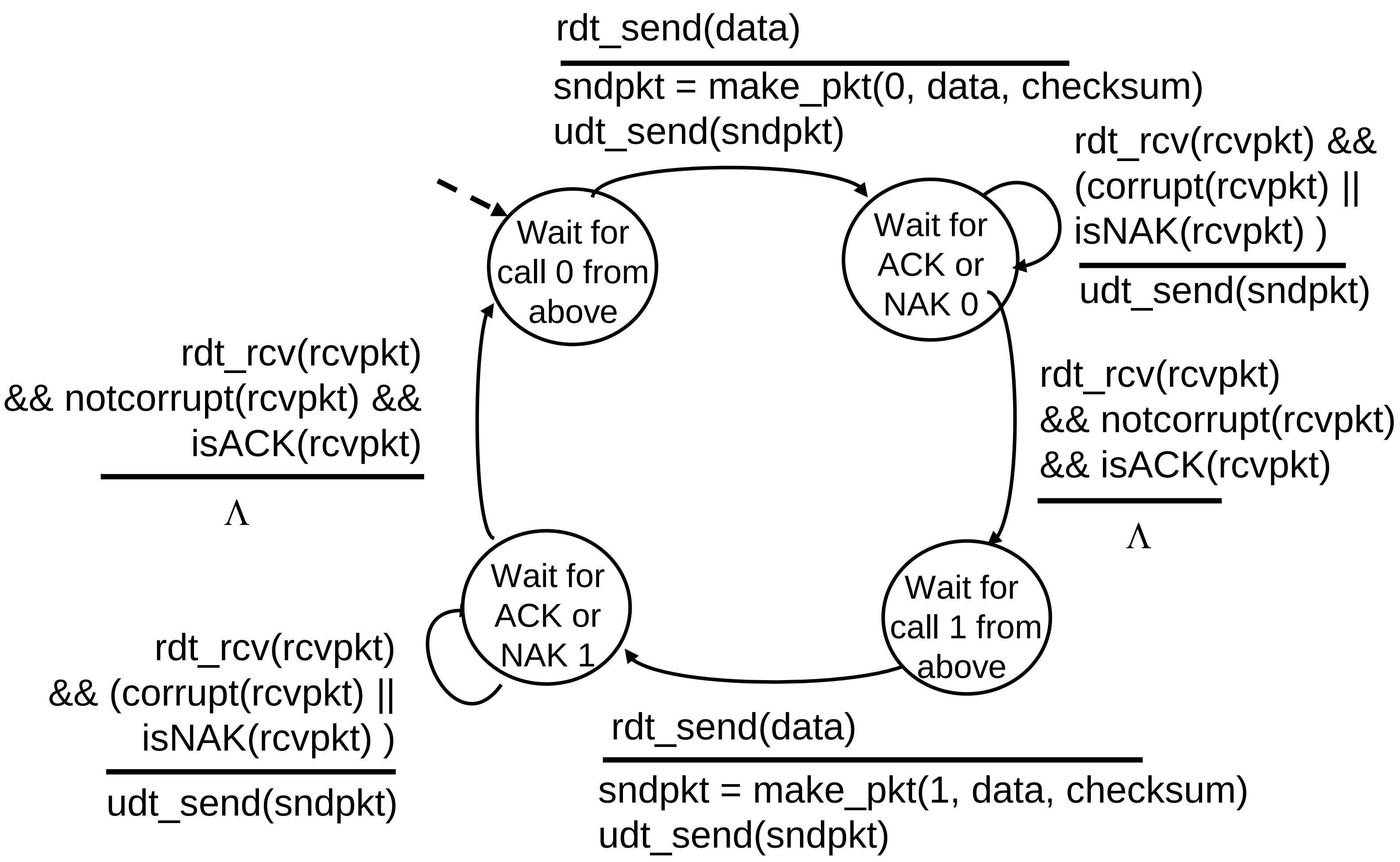
handling duplicates:

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

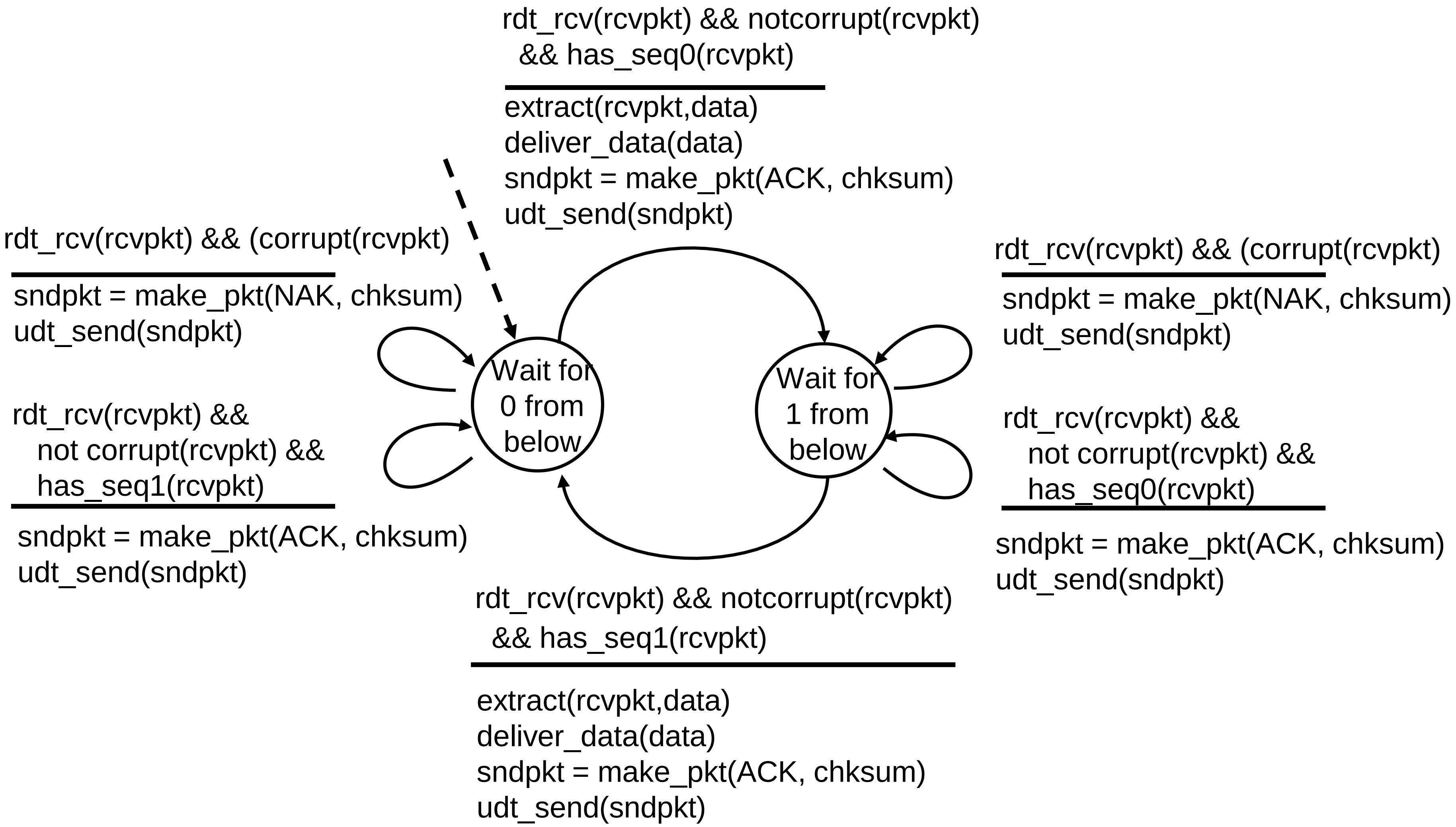
stop and wait

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:

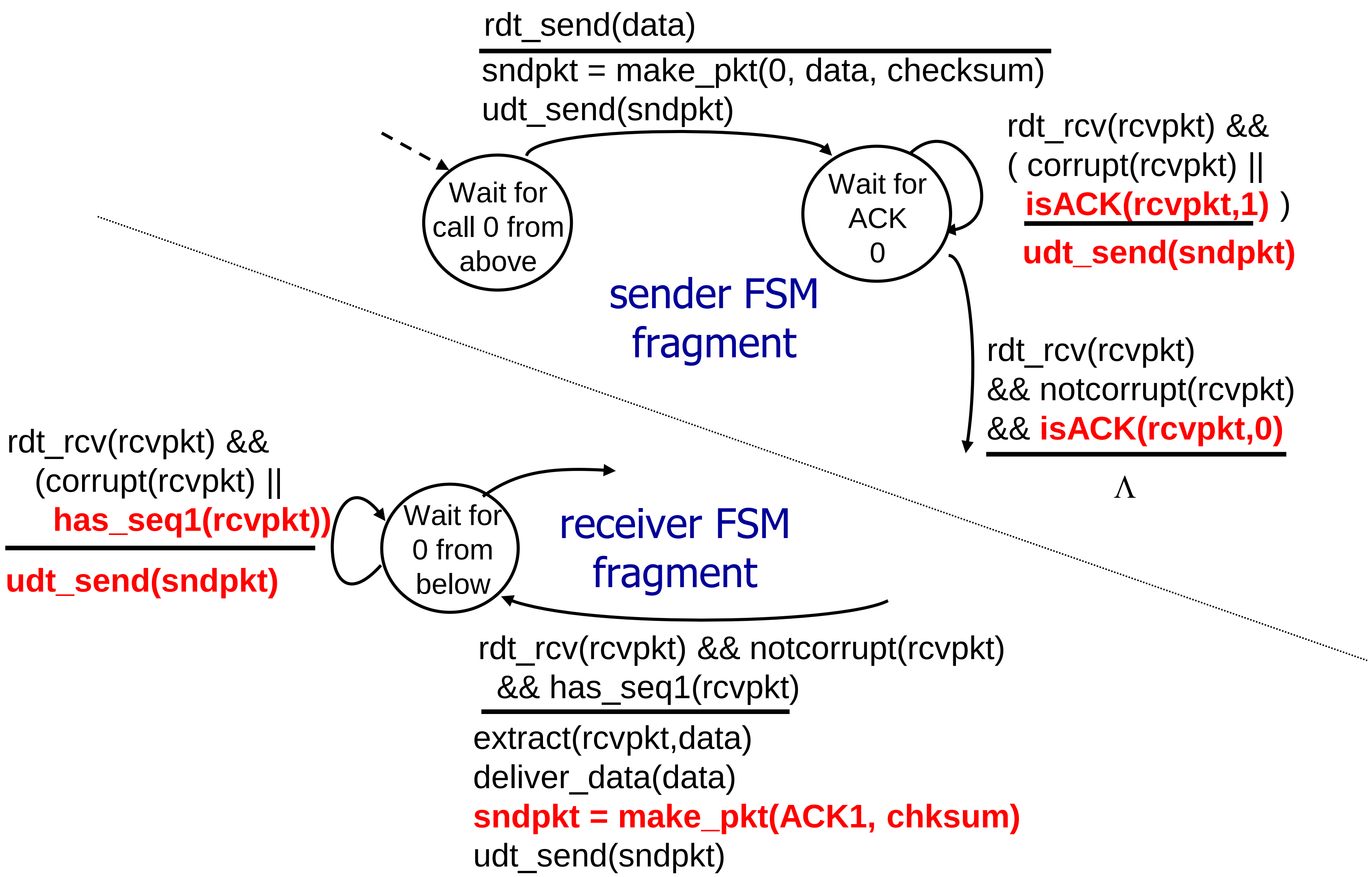
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

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

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

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

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

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

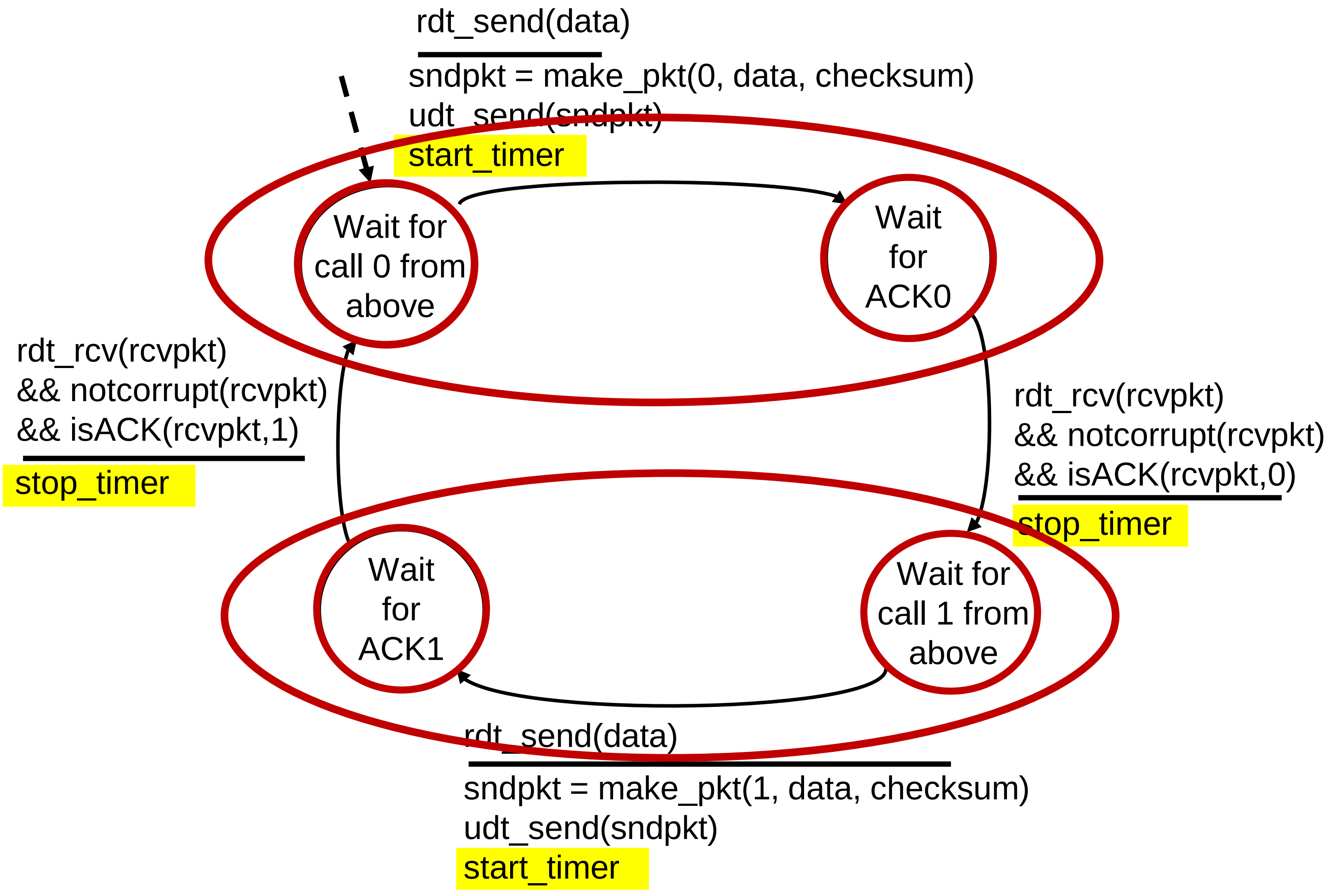
rdt3.0: channels with errors *and* loss

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

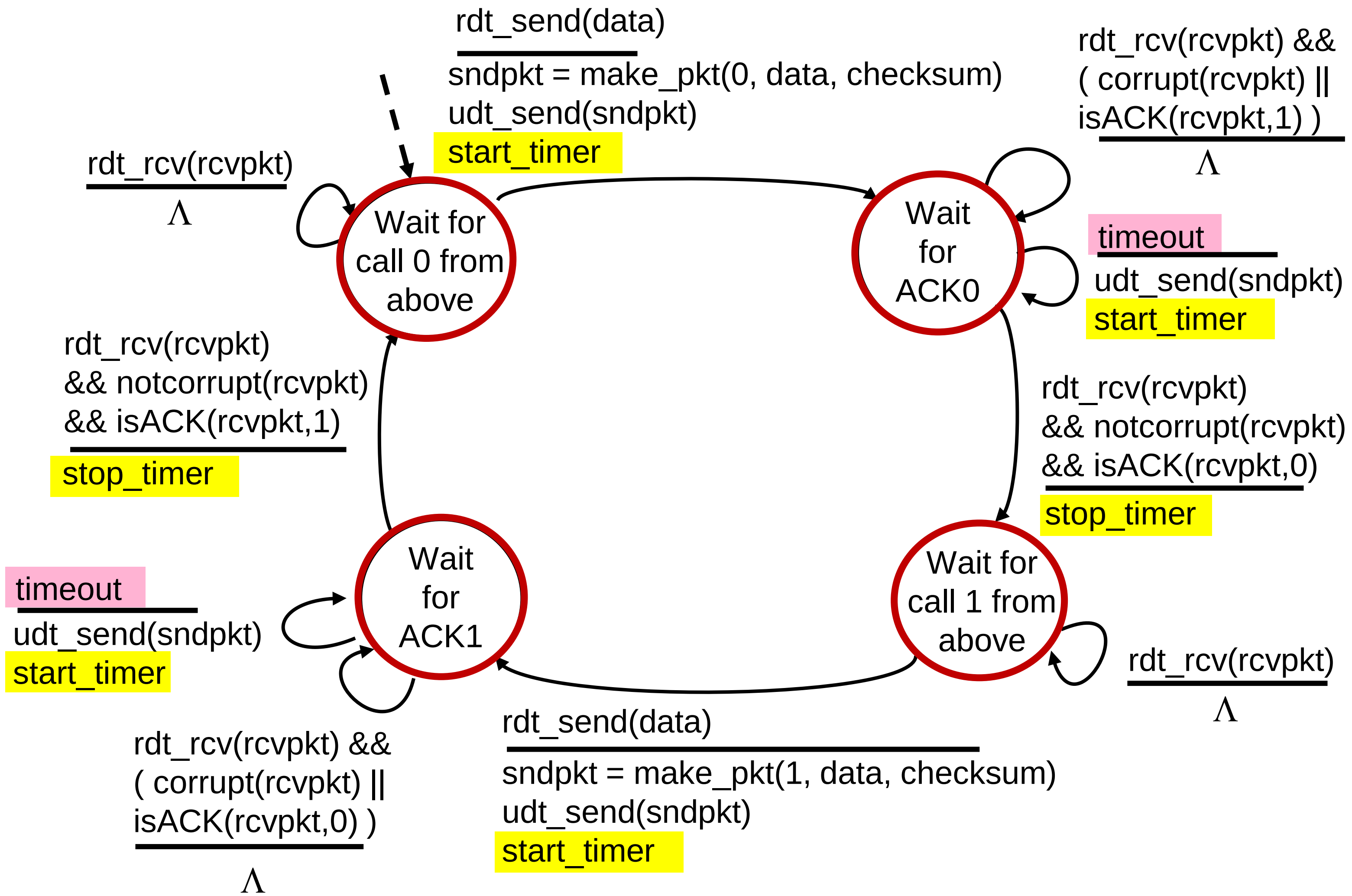
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

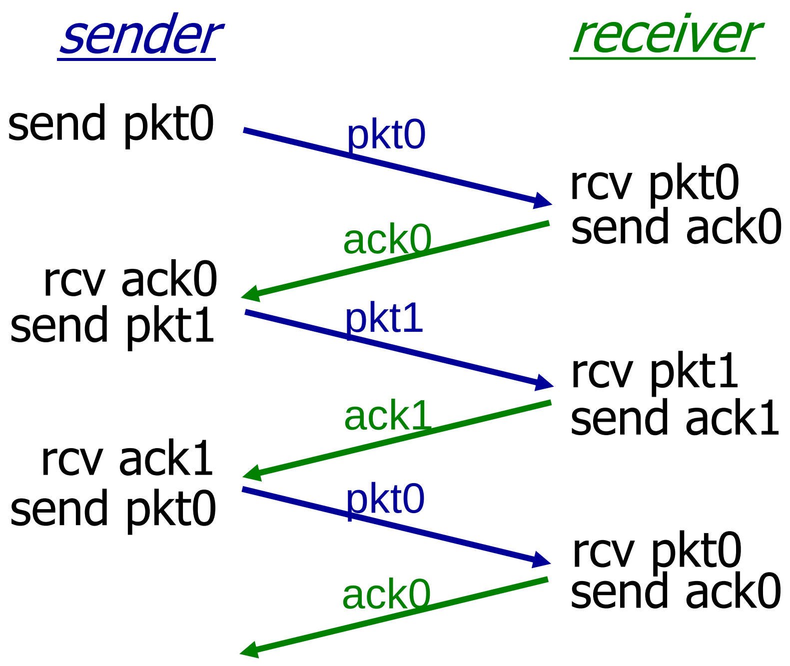
rdt3.0 sender



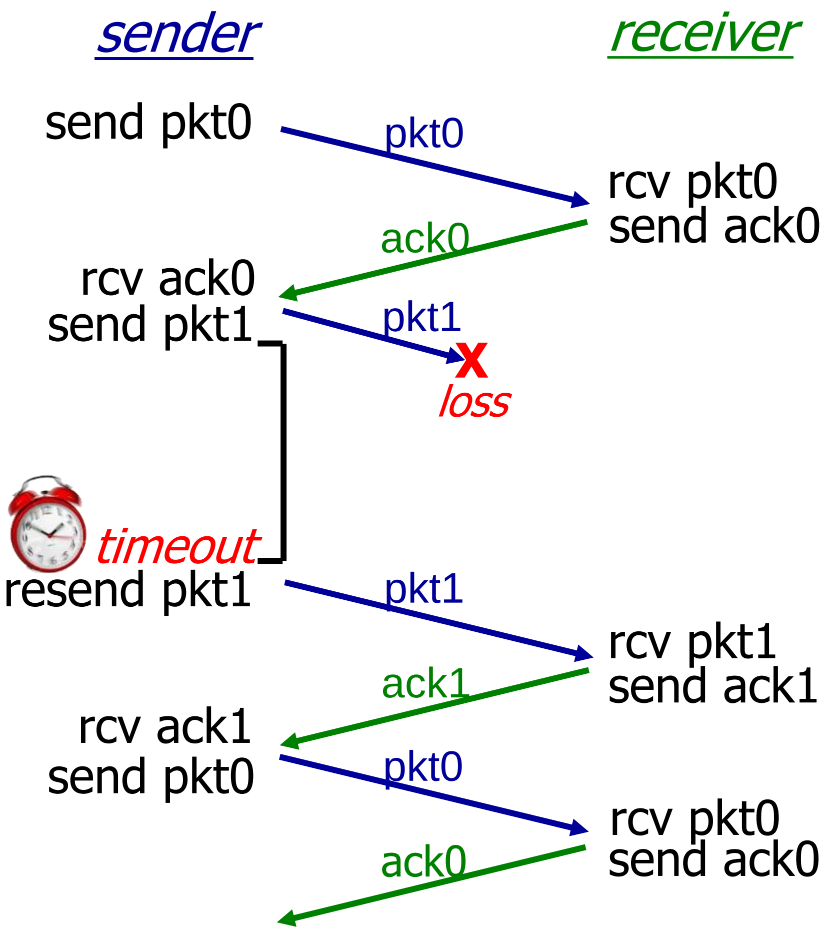
rdt3.0 sender



rdt3.0 in action

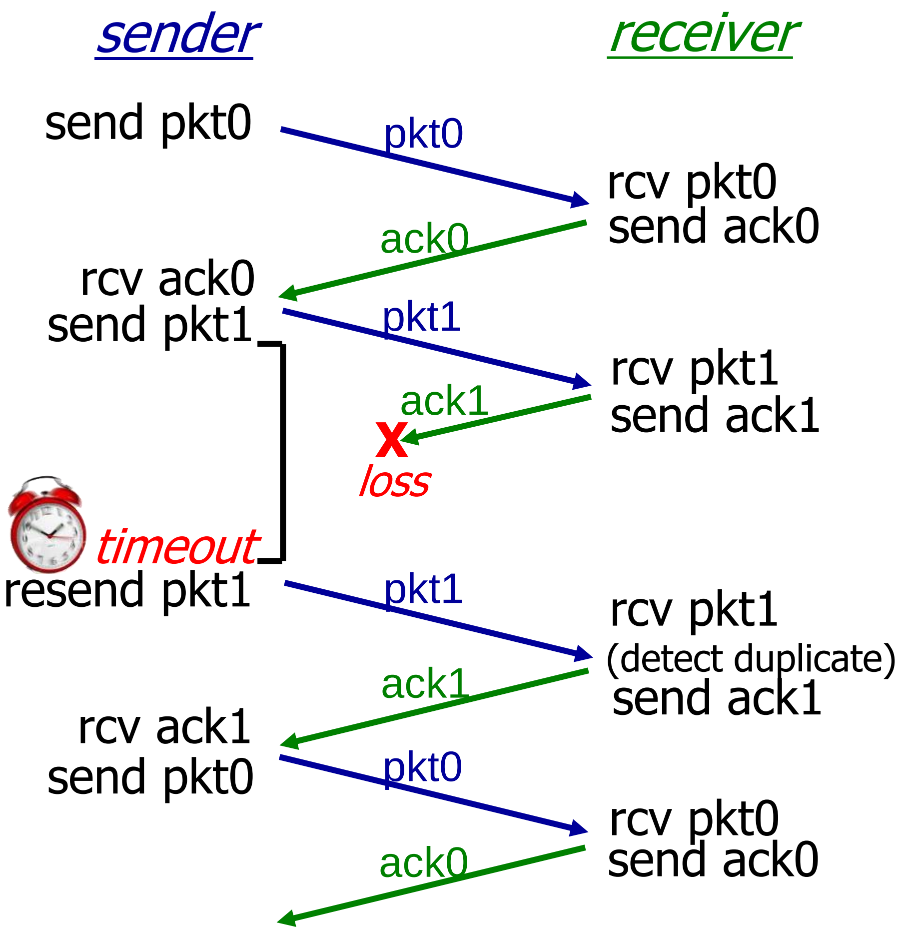


(a) no loss

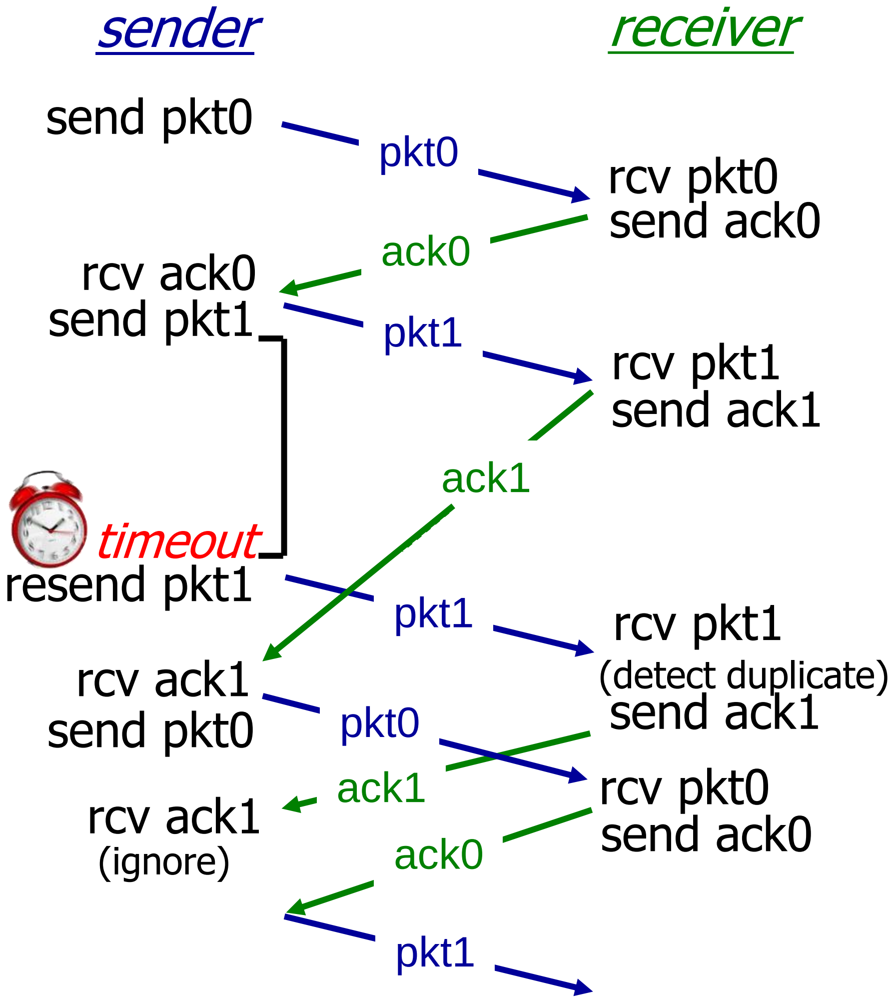


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

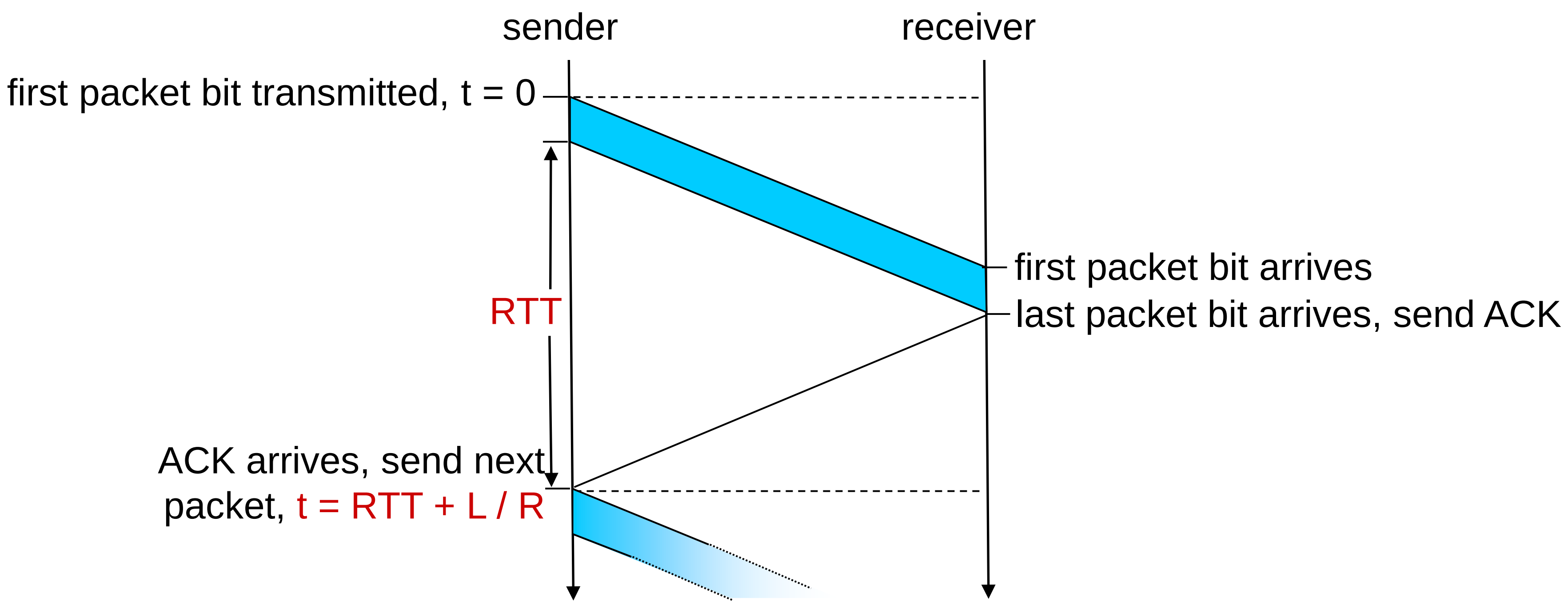
Performance of rdt3.0 (stop-and-wait)

- 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:

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

rdt3.0: stop-and-wait operation

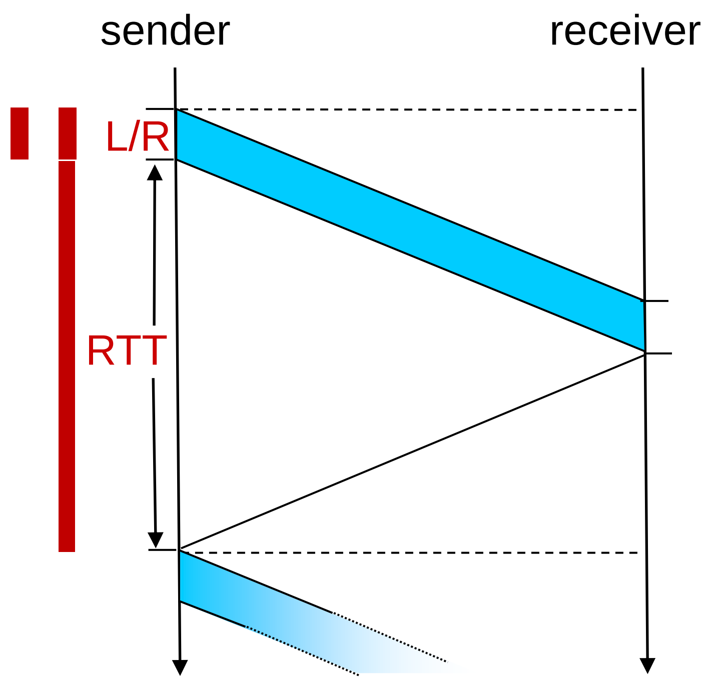


rdt3.0: stop-and-wait operation

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

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

$$= 0.00027$$

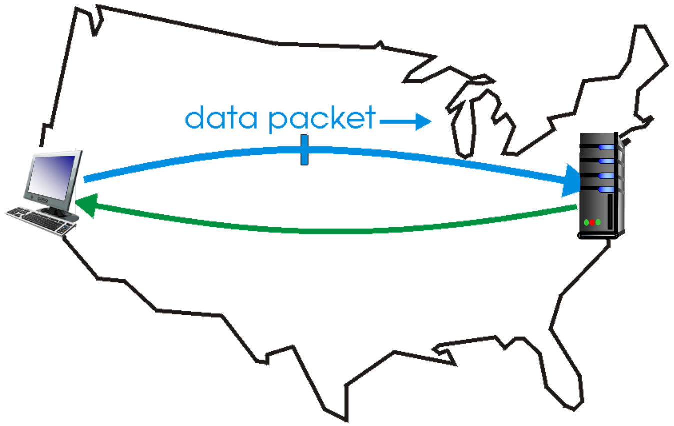


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

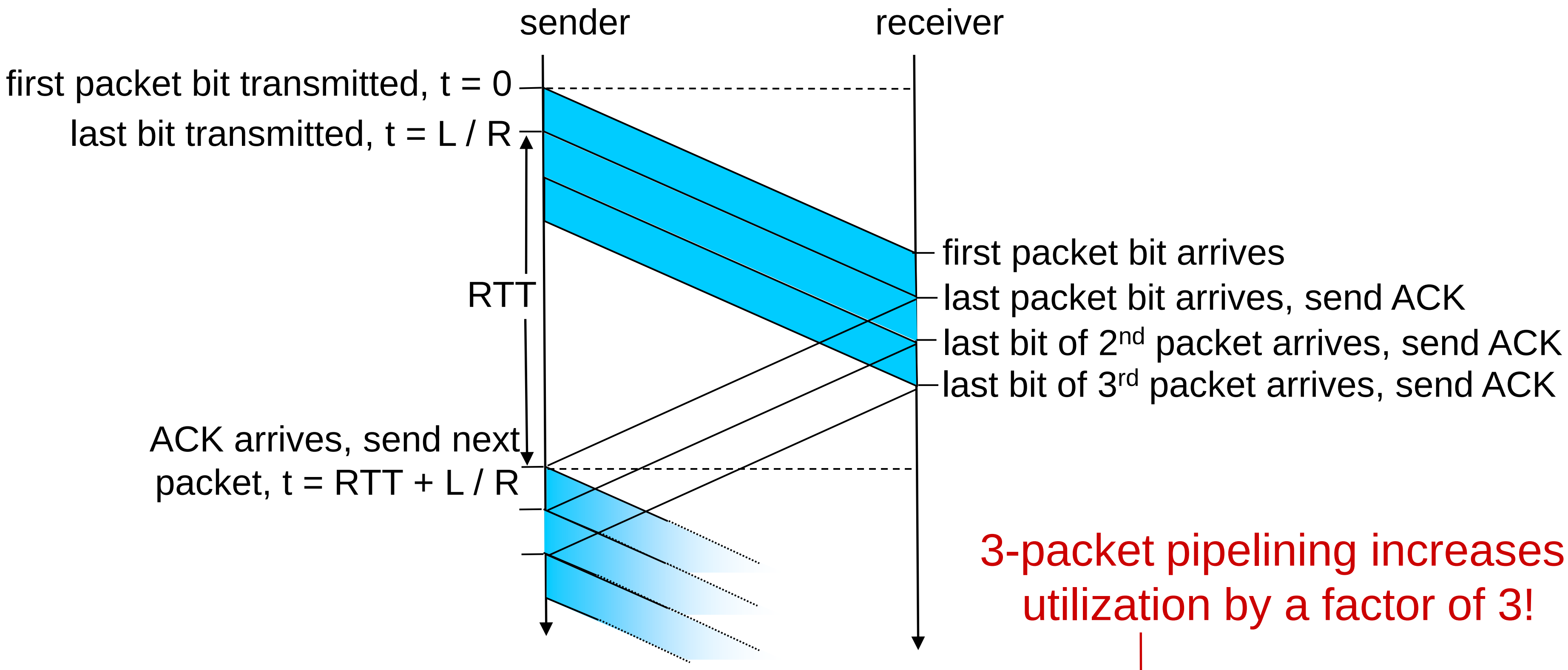
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



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

Pipelining: increased utilization

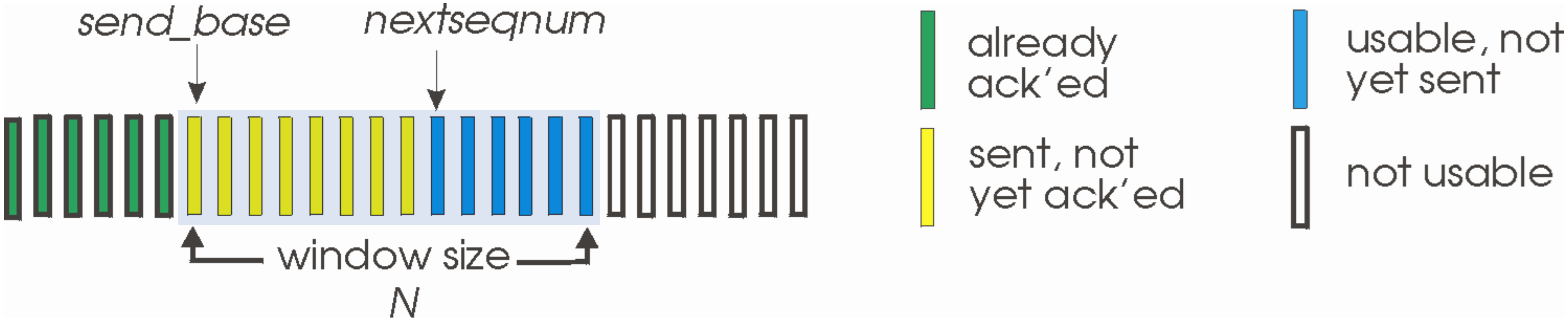


3-packet pipelining increases utilization by a factor of 3!

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

Go-Back-N: sender

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

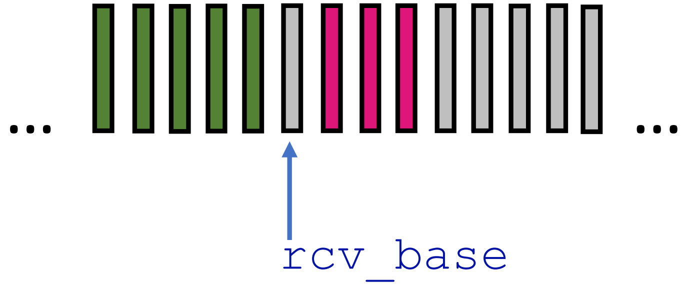


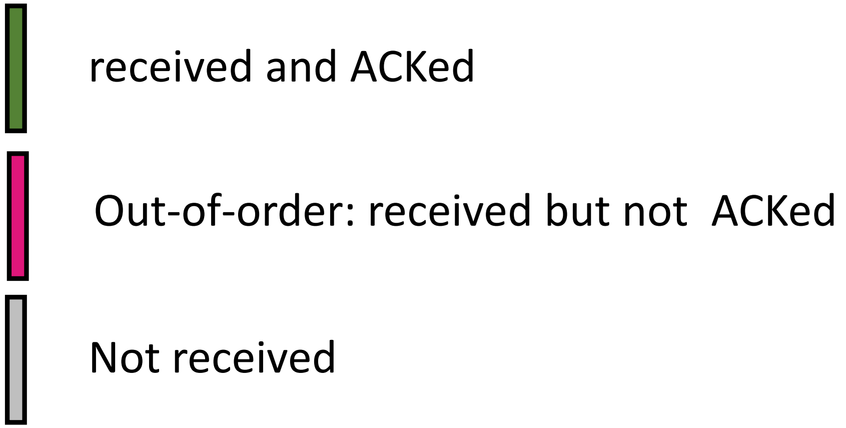
- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

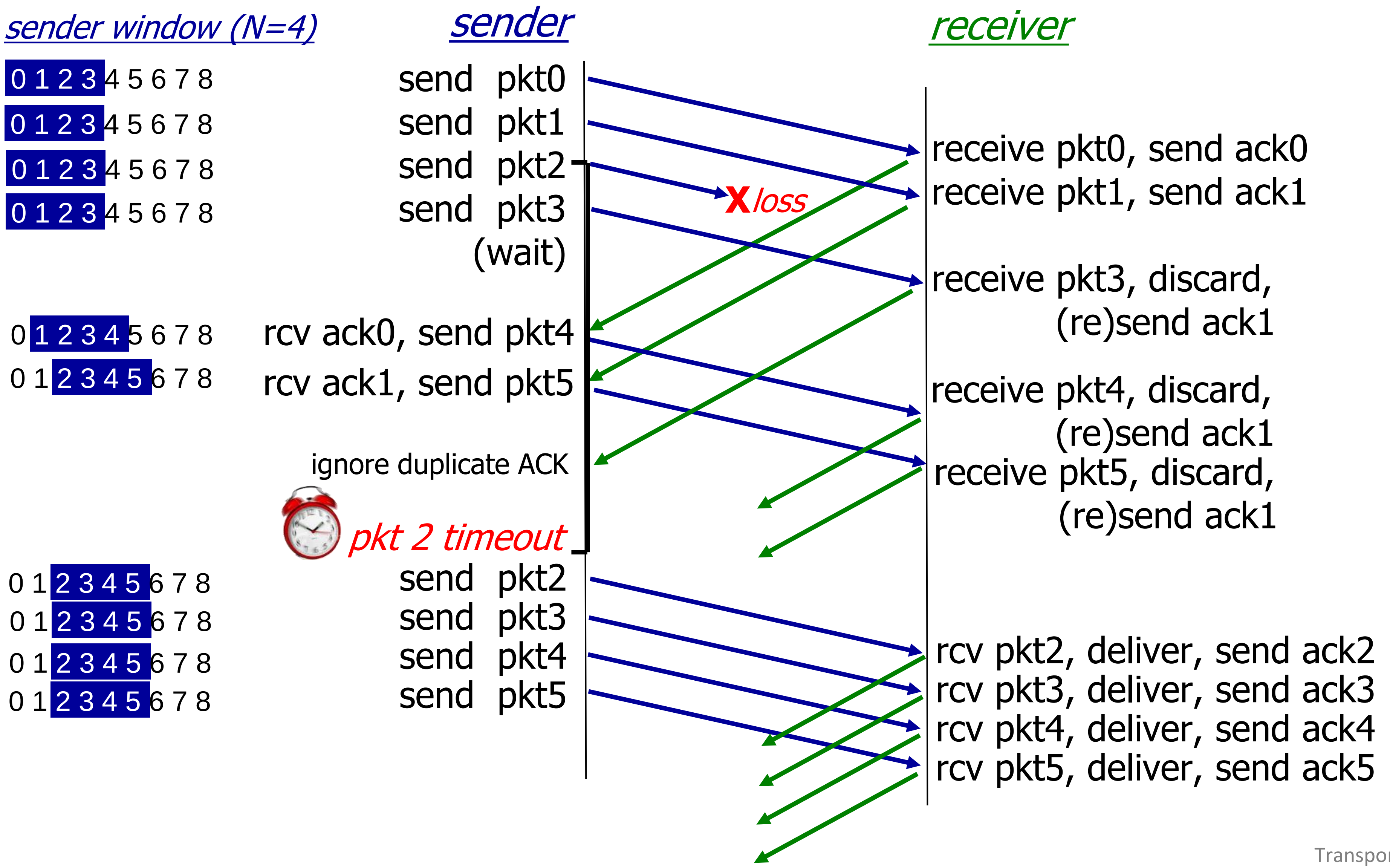
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don’t buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order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

Receiver view of sequence number space:





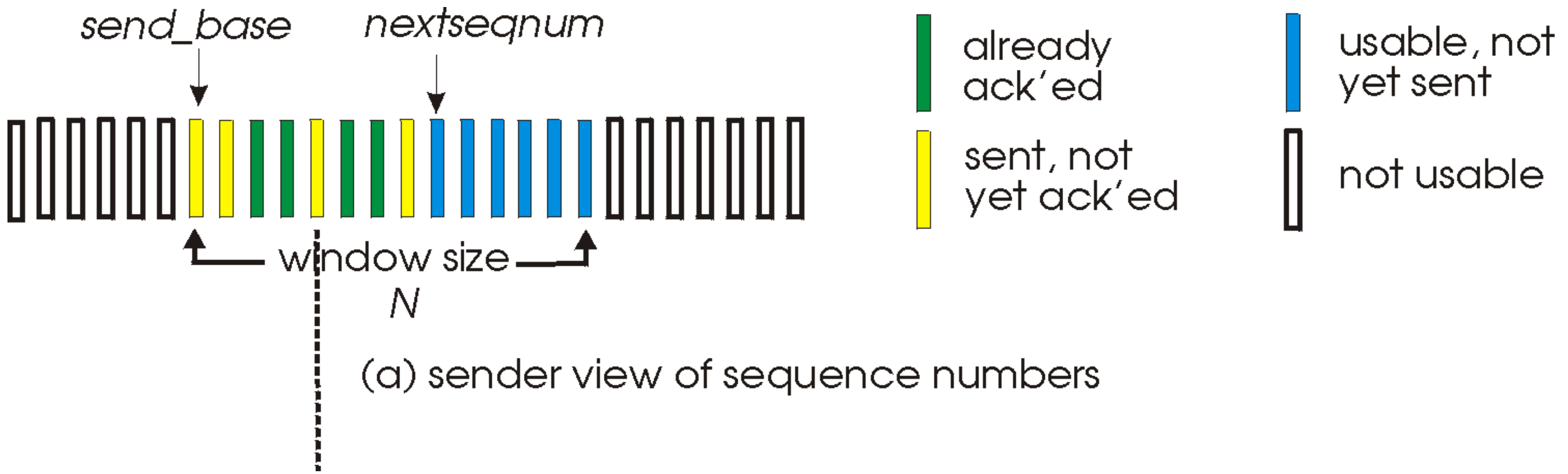
Go-Back-N in action



Selective repeat

- 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: sender, receiver windows



Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

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

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

receiver

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

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

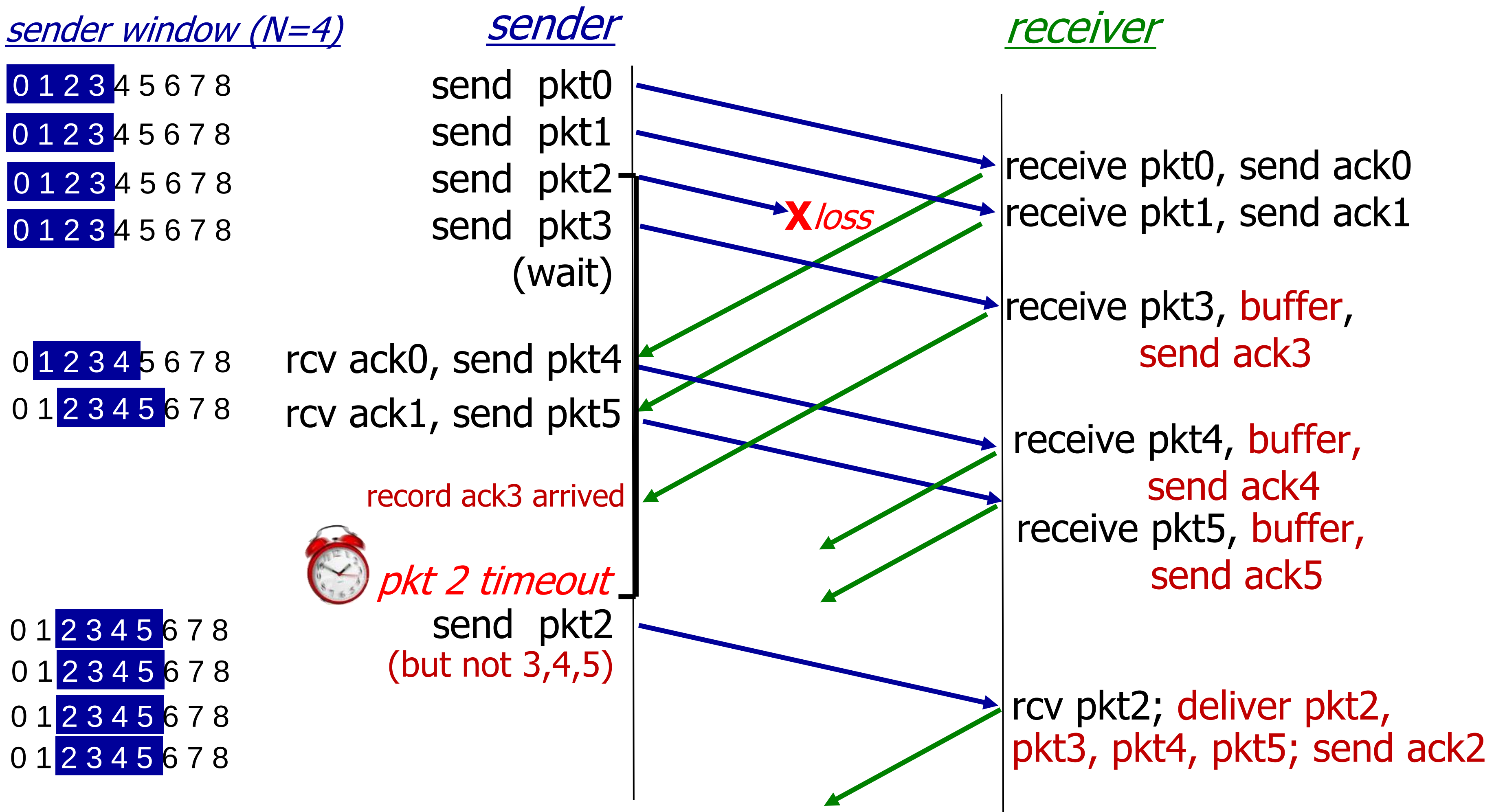
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action

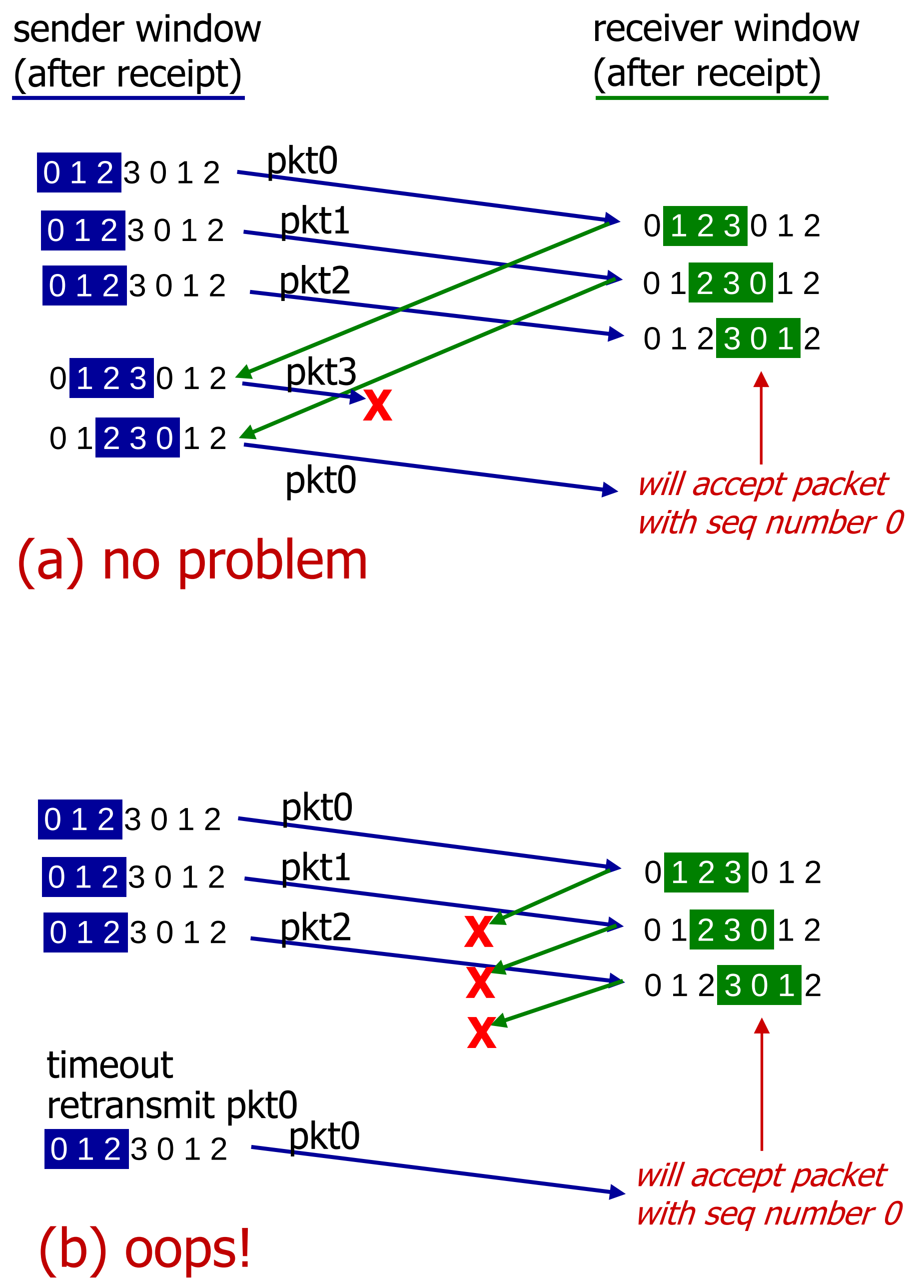


Q: what happens when ack2 arrives?

Selective repeat: a dilemma!

example:

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



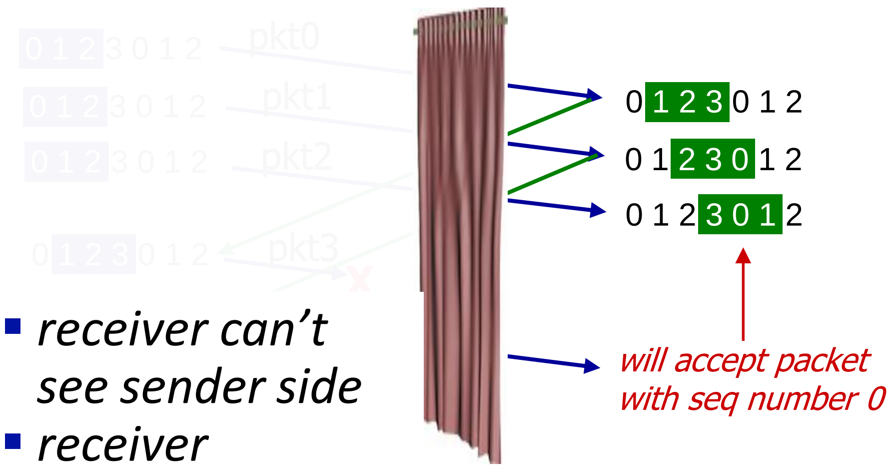
Selective repeat: a dilemma!

example:

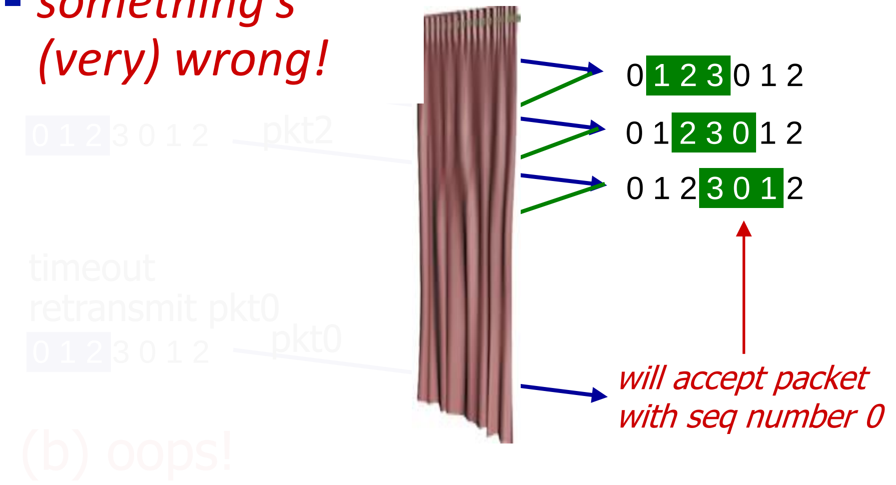
- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

sender window (after receipt)

receiver window (after receipt)



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



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

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