CCCN 312 Computer Networks

Instructor: Instructor name

1st Trimester 2022/23

University of Jeddah

Outline

1. Introduction
2. Application layer
3. **Transport layer**
4. Network layer: Data Plane – Control Plane
5. Link layer

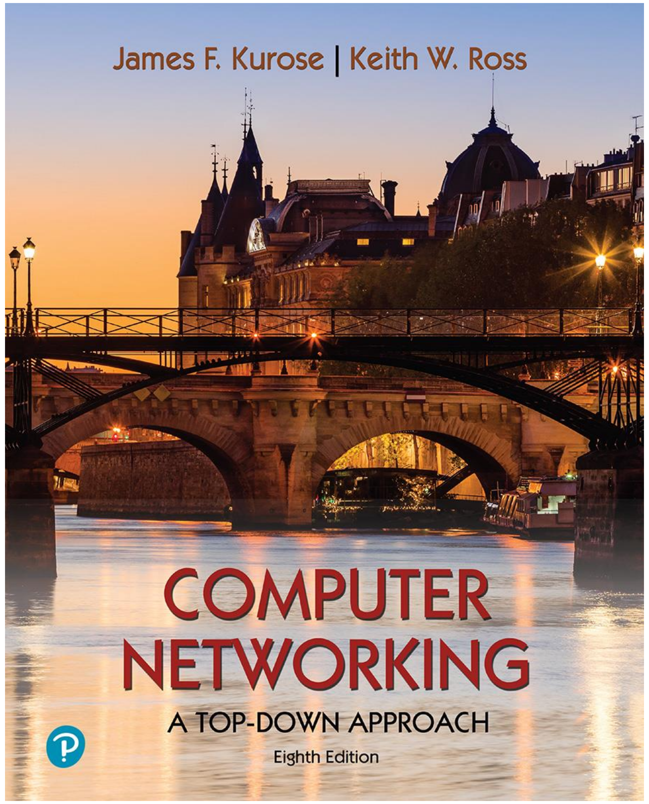
Chapter 3
Transport Layer

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:

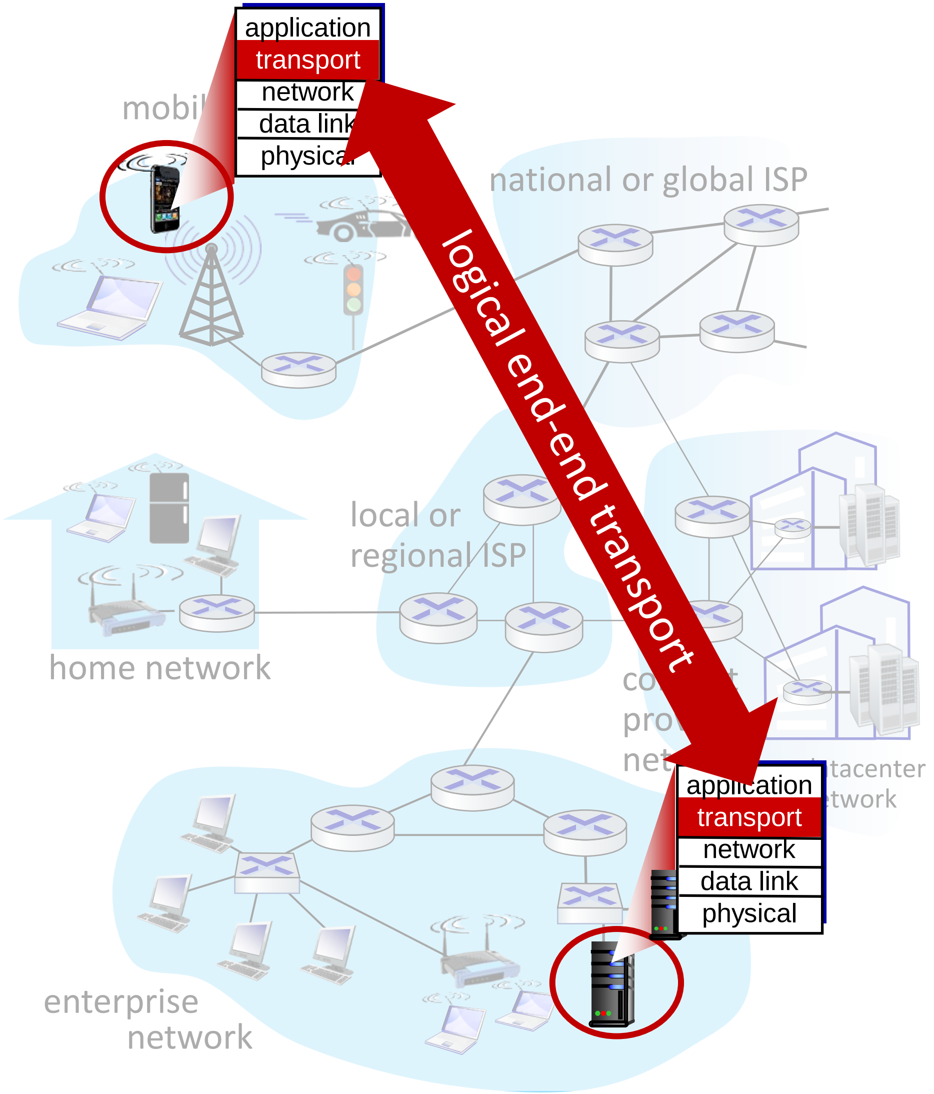
- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion 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:
  1 • sender: host breaks application messages into *segments*, passes to network layer
  2 • receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP
    1 2



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
- Port# = name of the kid receiving the letter

- transport protocol = Ann and Bill who *demux* to in-house siblings
- network-layer protocol = postal service

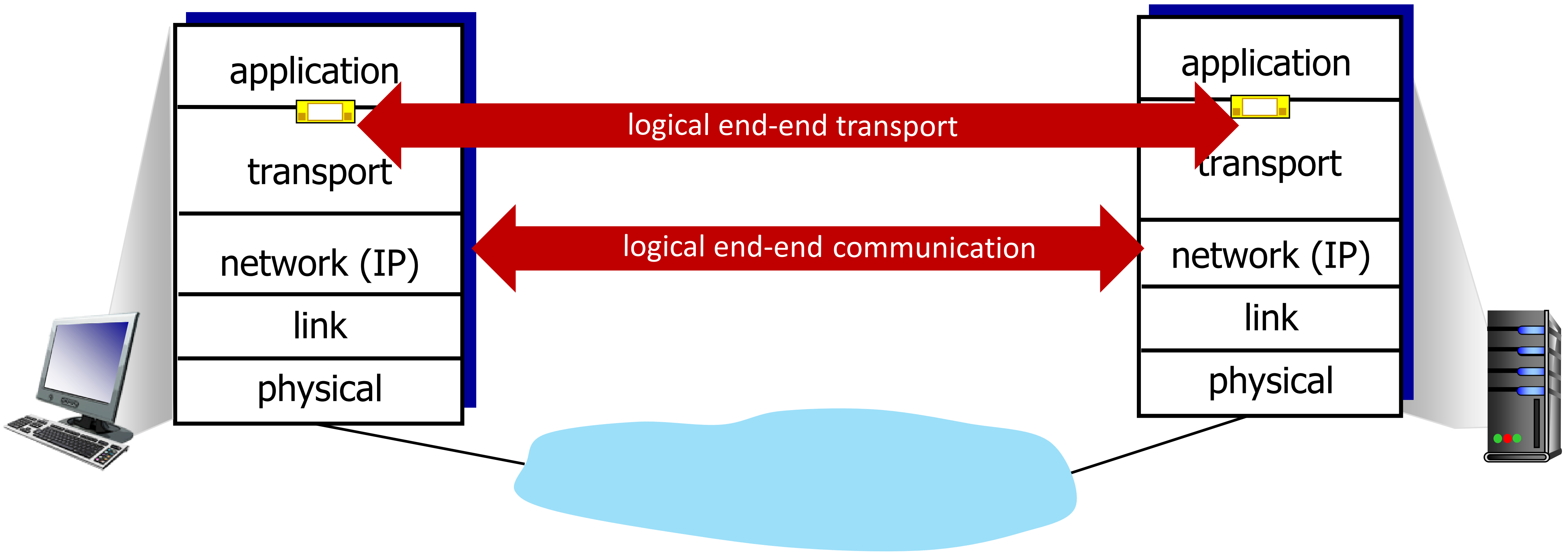
Transport vs. network layer services and protocols

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

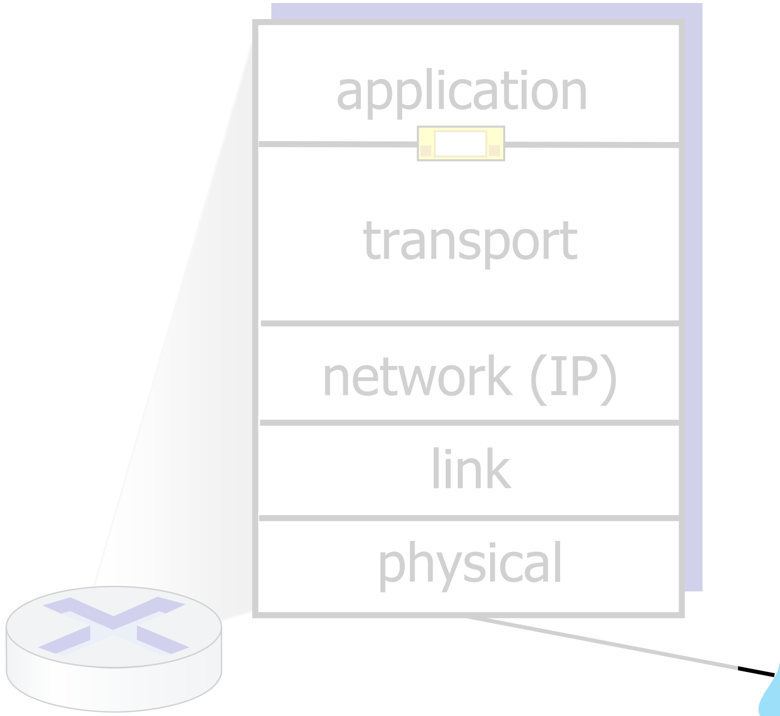
- network layer: logical communication between *hosts*
- Provides end-to-end delivery to transport segments between hosts.
- IP is "best-effort" توصيل بدون ضمان
  - No guarantees on delivery of segments
  - No guarantees on the order of delivery
  - No guarantees on the integrity of their data.

→ Unreliable service

Transport vs. network layer services

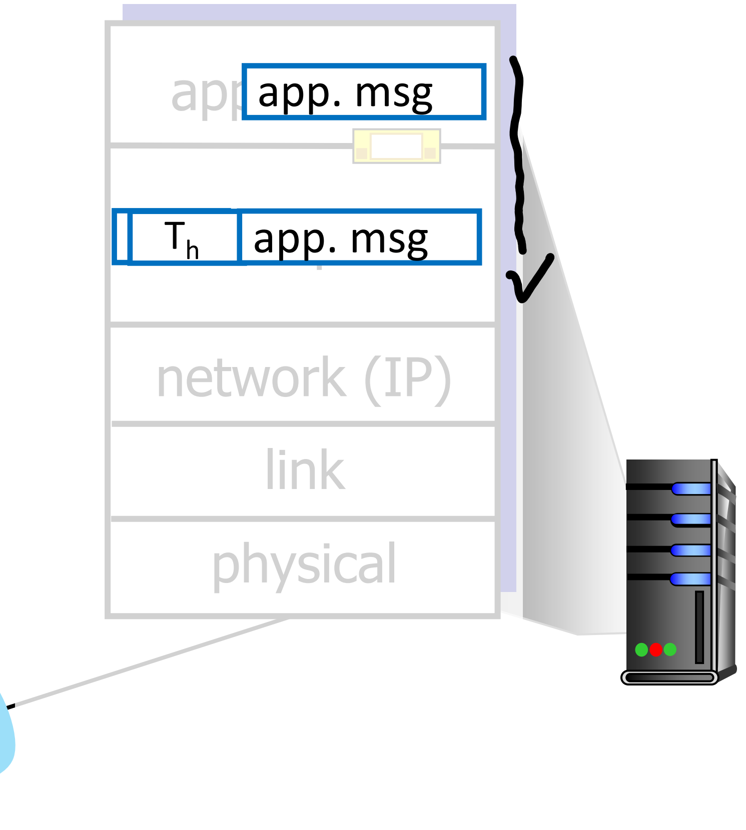


Transport Layer Actions

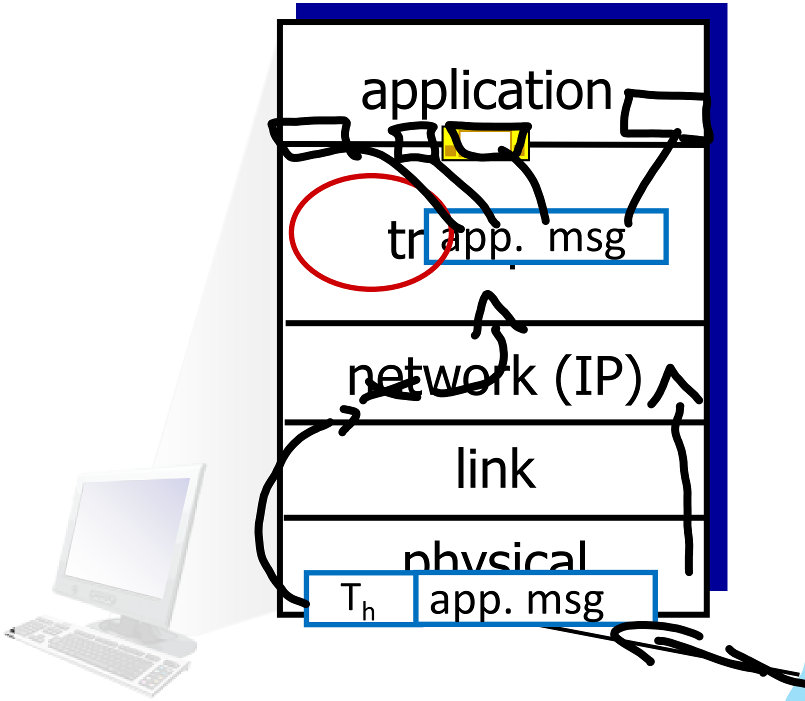


Sender:

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



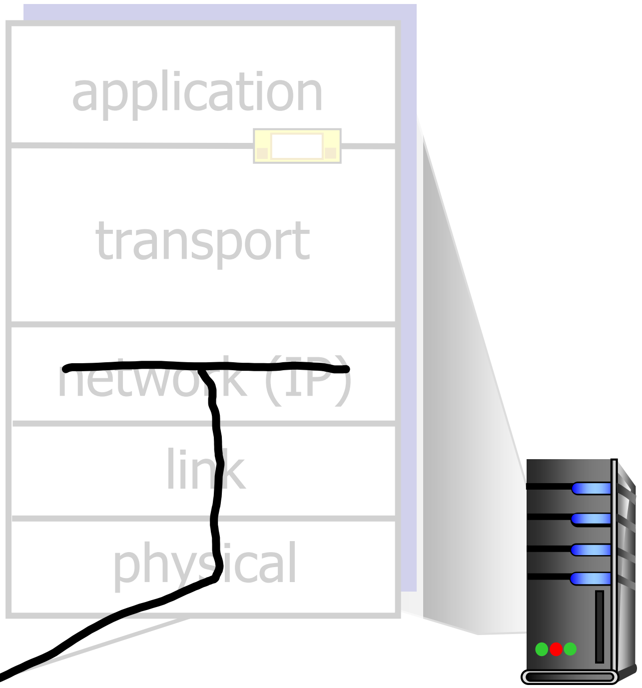
Transport Layer Actions



Receiver:

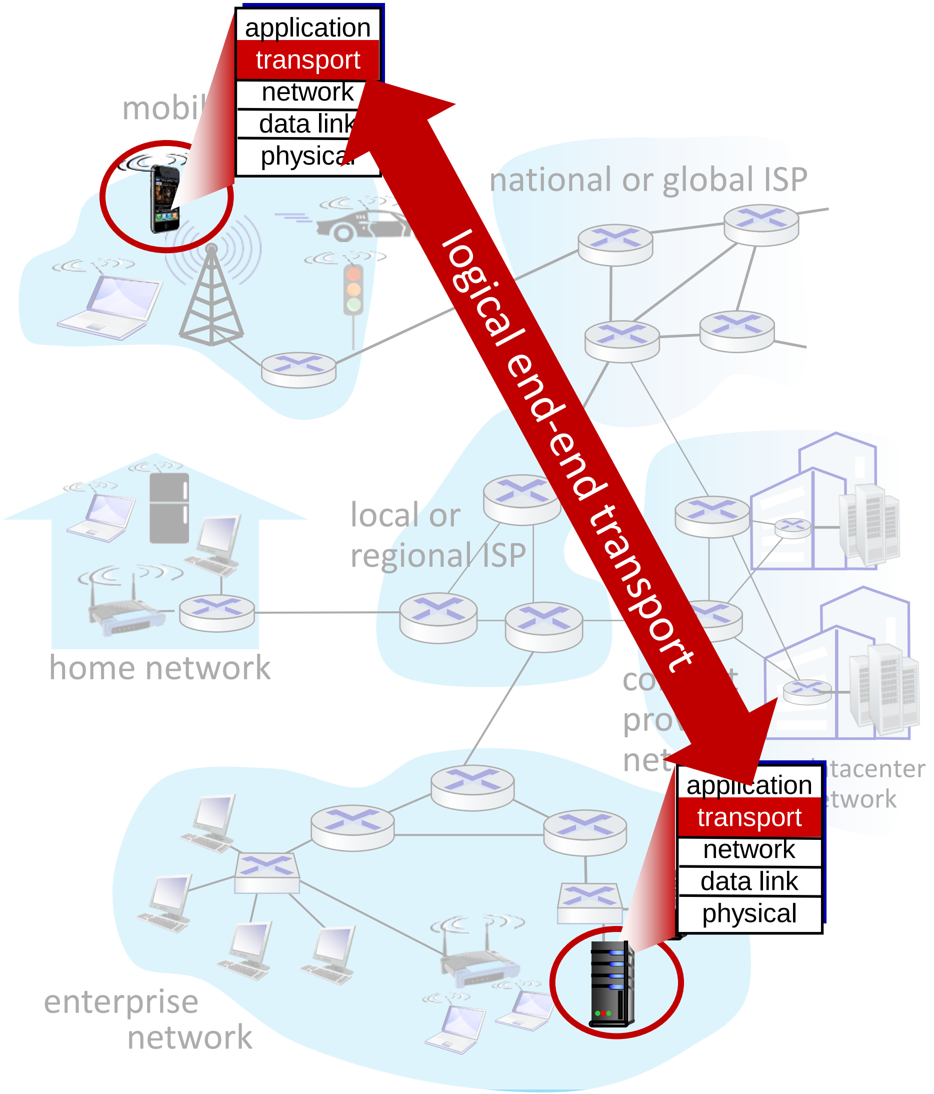
- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

هو اللي يوصله لـ socket
المناسب



Two principal Internet transport protocols

- **TCP**: Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control ضبط الازدحام
  - التحكم بسرعة الإرسال 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



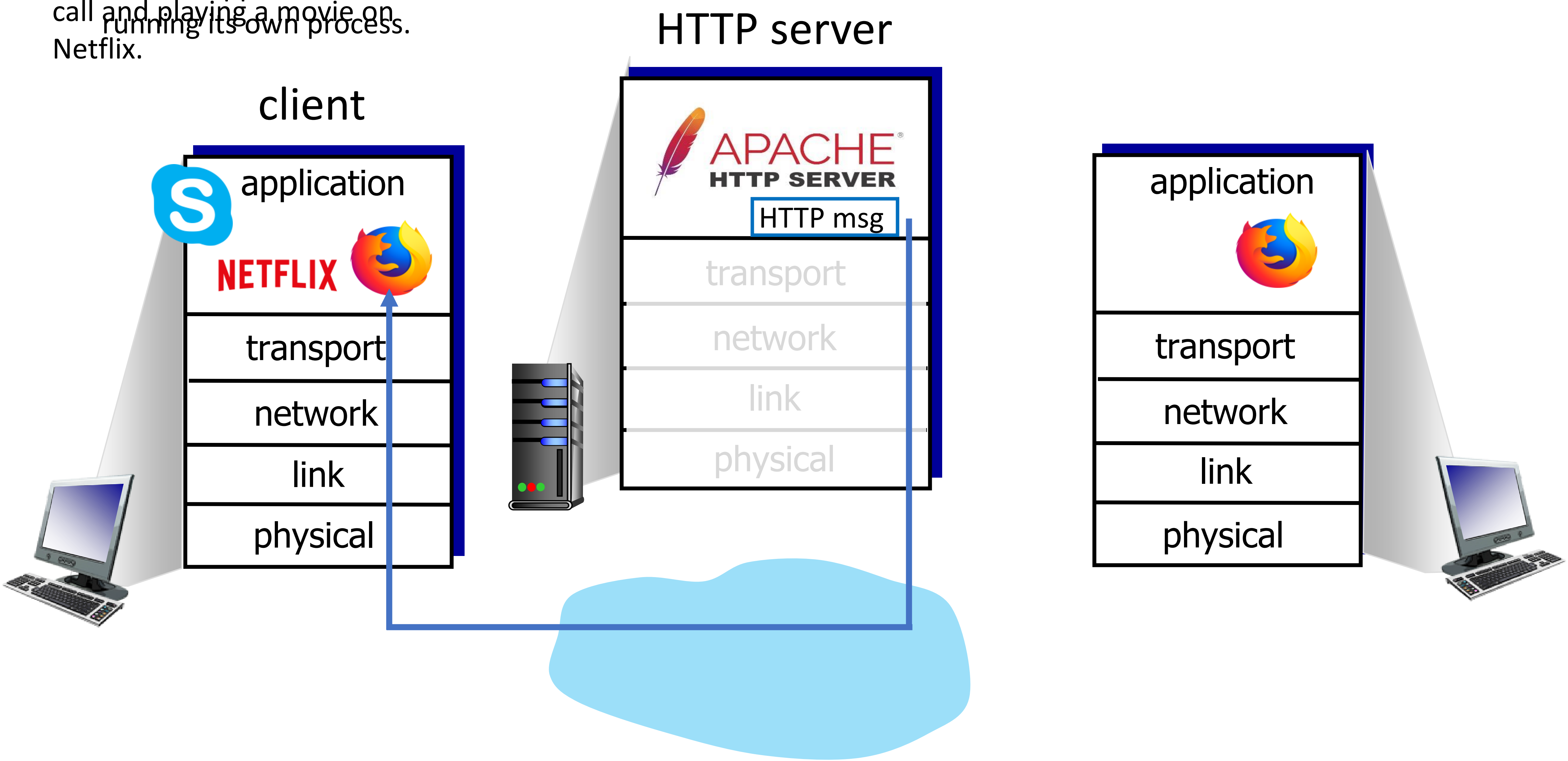
Chapter 3: roadmap

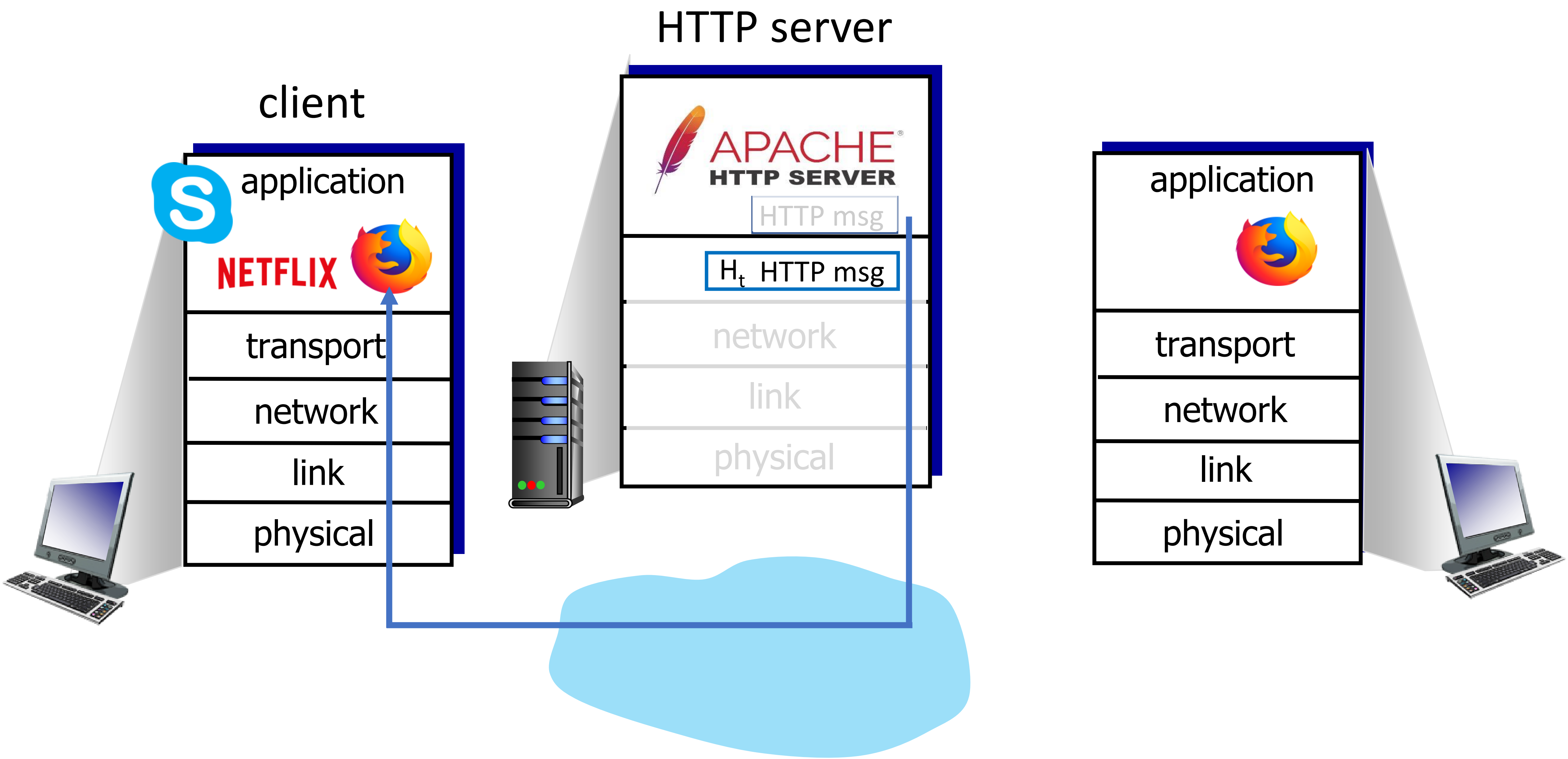
- Transport-layer services
- **Multiplexing and demultiplexing**
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Scenario - 1

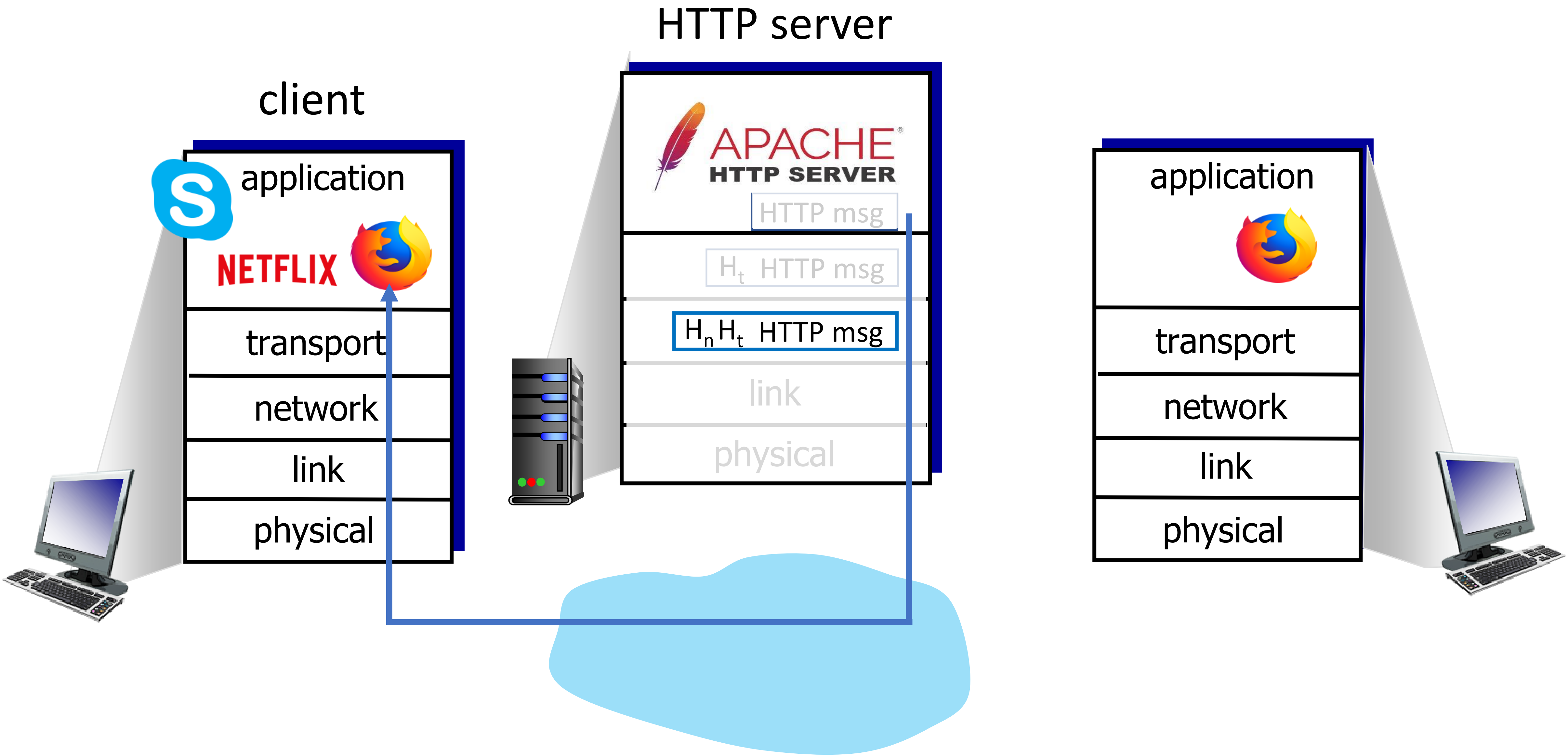
Suppose you are downloading Web pages while running Skype call and playing a movie on Netflix.

Each Application is running its own process.



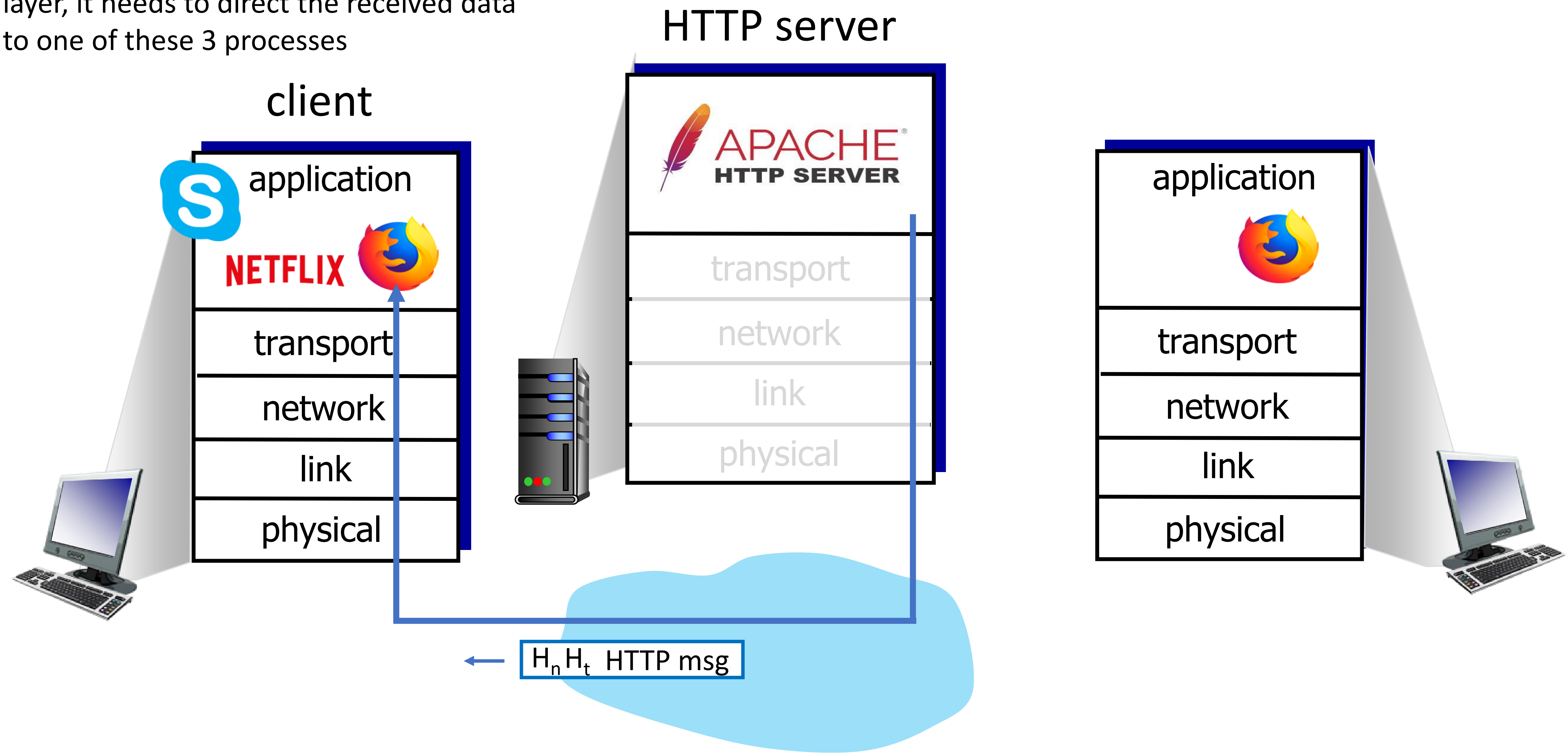

HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
Ht HTTP msg
network
link
physical
application
transport
network
link
physical

Scenario - 1


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

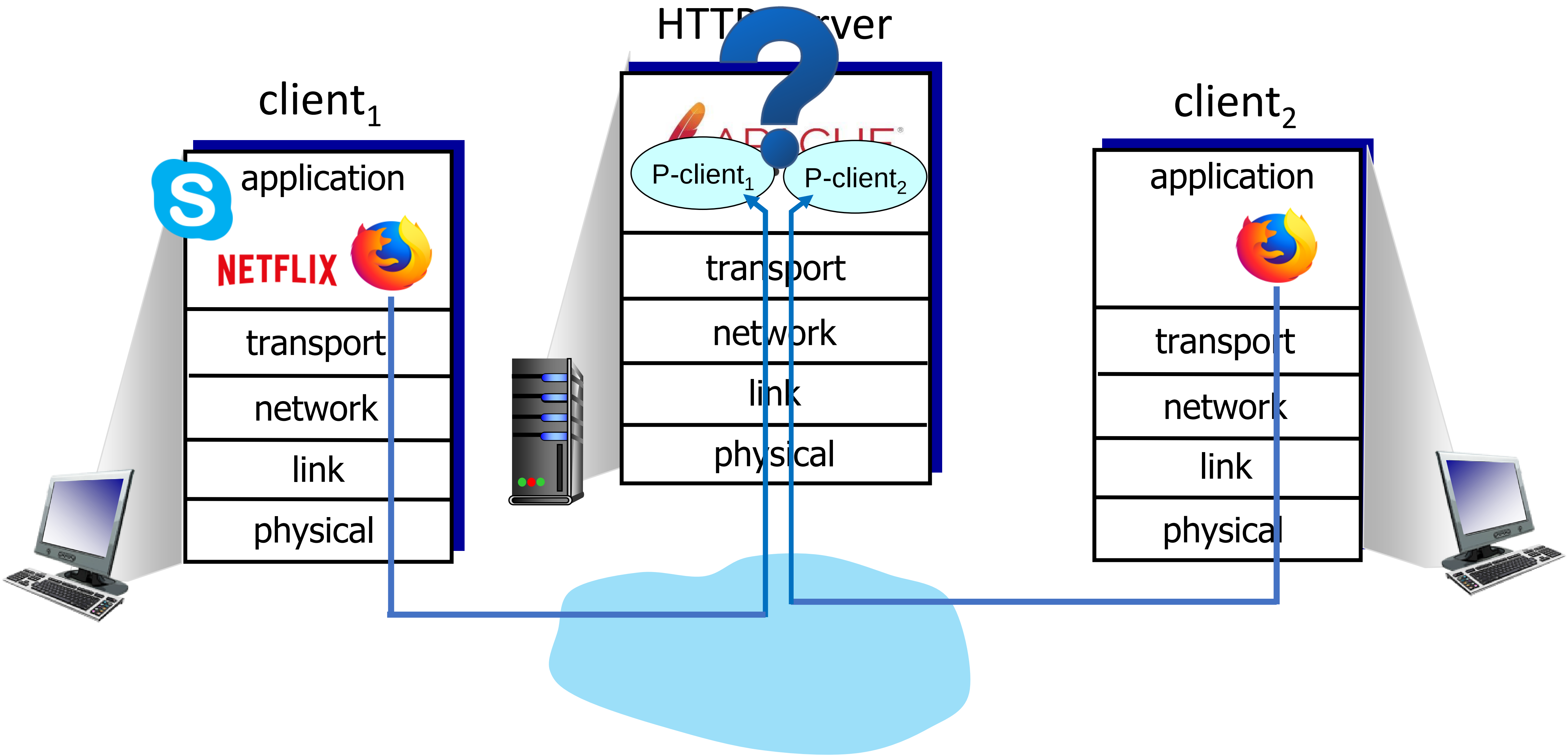
Scenario - 1

When the transport layer in your computer receives data from the network layer, it needs to direct the received data to one of these 3 processes



Two HTTP clients send request messages to server

Scenario - 2



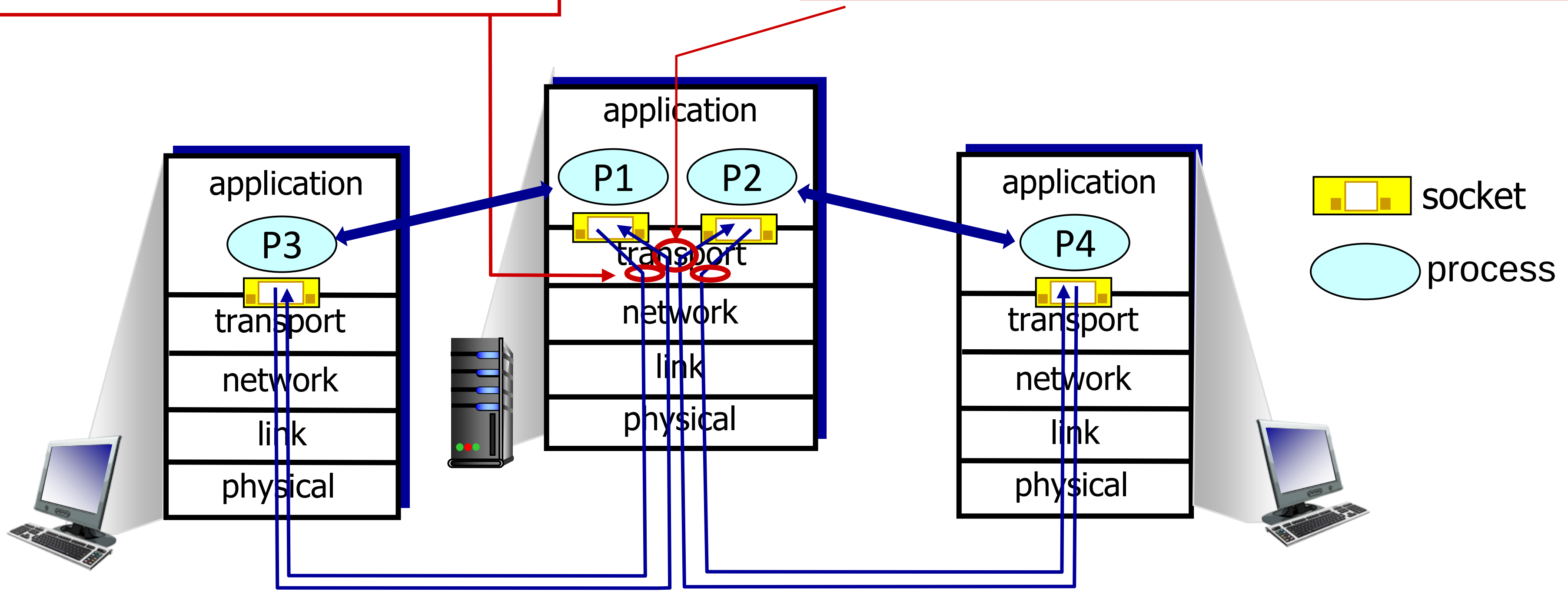
Multiplexing/demultiplexing

multiplexing at sender:

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

demultiplexing at receiver:

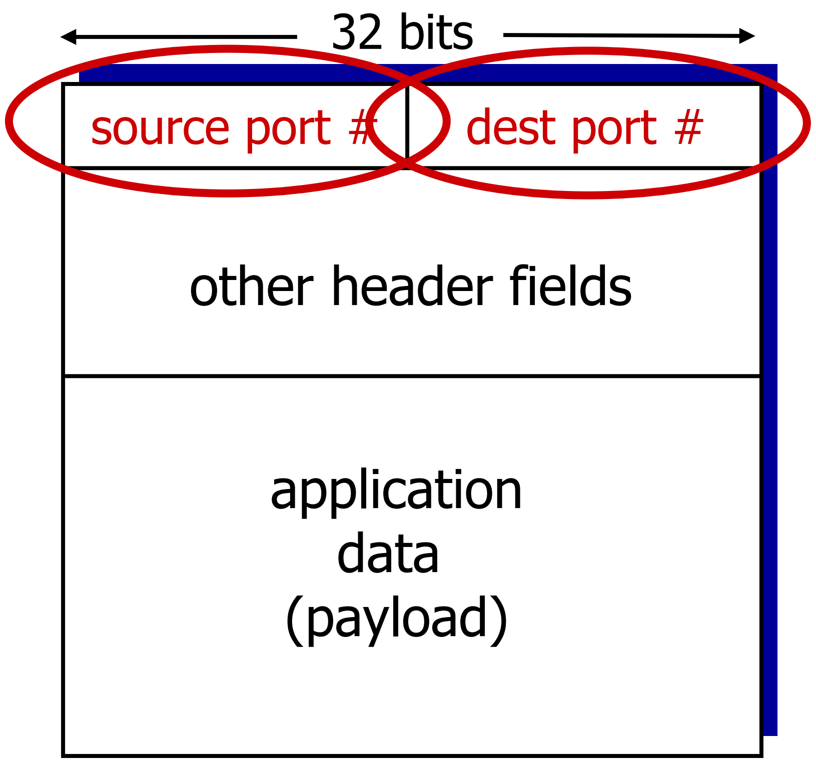
use header info to deliver received segments to correct socket



How demultiplexing works

- Each port number is a 16-bit number (2^{16}), ranging from 0 to 65535.
- Port numbers from 0 to 1023 are called well-known port numbers and are restricted.

- host receives IP datagrams
  - each datagram has *source & 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 UDP

Recall:

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

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

Sender

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

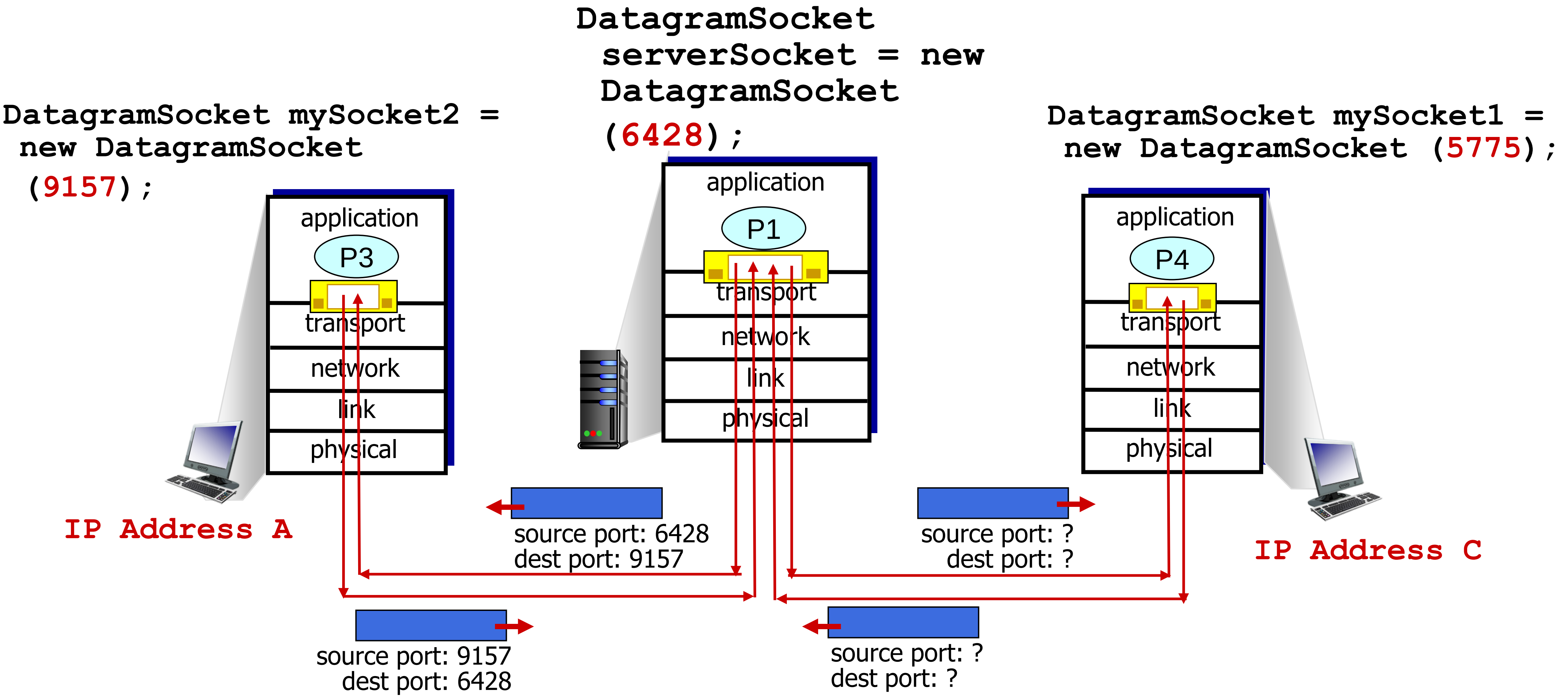
Reciever

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

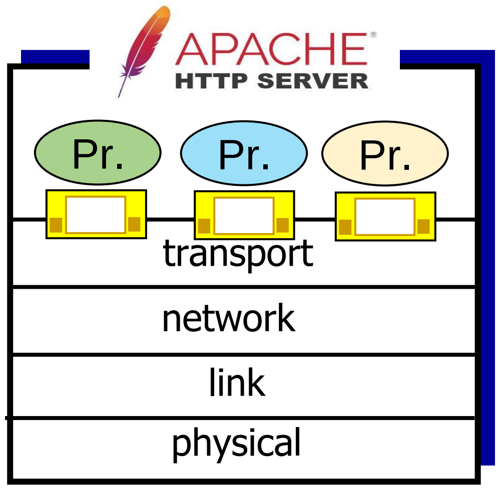
Connectionless demultiplexing: an example



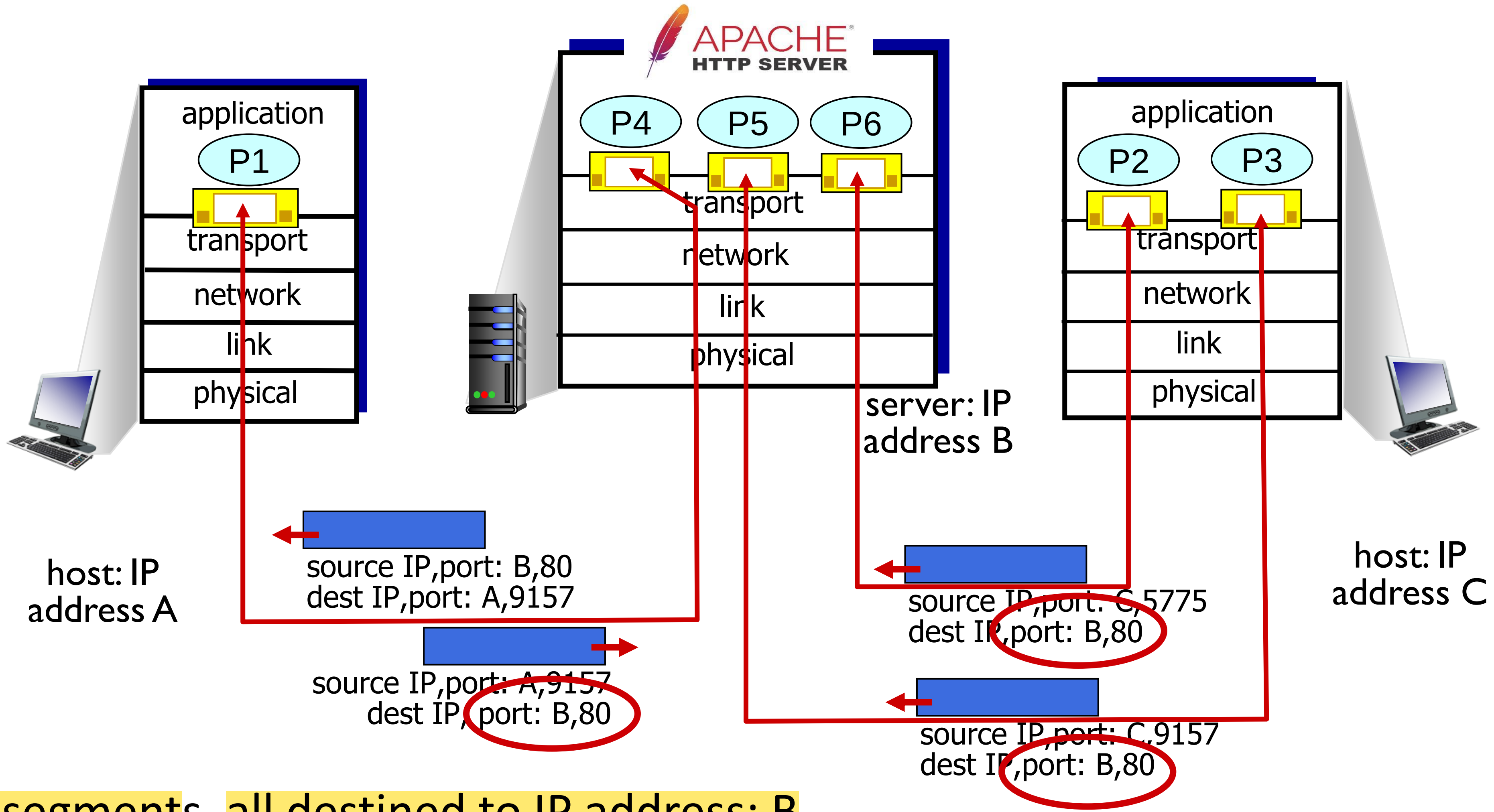
Connection-oriented demultiplexing **TCP**

- 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 and 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: حساسة
  - 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

  - The QUIC protocol (Quick UDP Internet Connection, [Iyengar 2015]), used in Google's Chrome browser, uses UDP as its underlying transport protocol and implements reliability in an application-layer protocol on top of UDP

UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive):
    - Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss.

RTT Delay

  - DNS: UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP
  - SNMP
  - HTTP/3

    - These applications can use UDP and implement, as part of the application, any additional functionality that is needed

Internet Apps & transport protocols

Application	Application-layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP (1.0, 1.1, 2)	TCP
Web	HTTP 3	UDP
Web	QUIC	UDP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	Typically proprietary	UDP or TCP
Internet telephony	Typically proprietary	UDP or TCP
Network Management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768 J. Postel
ISI
28 August 1980

User Datagram Protocol

Introduction

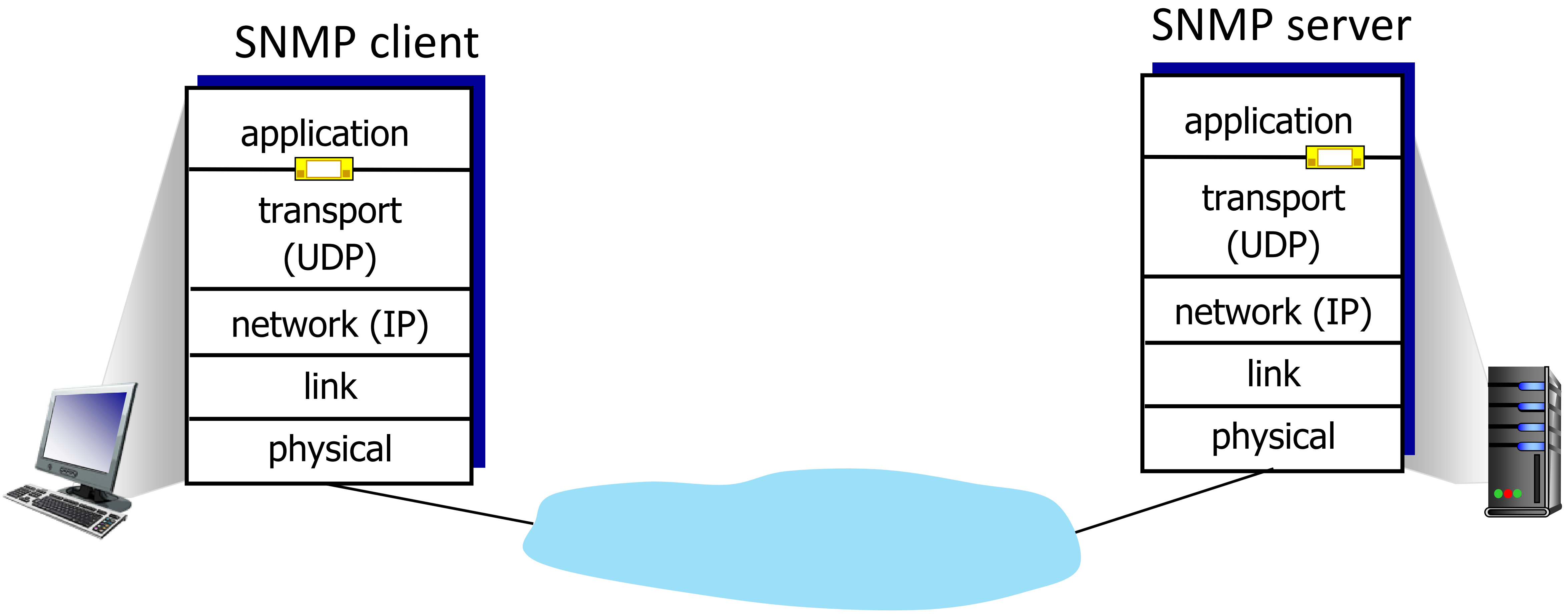
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

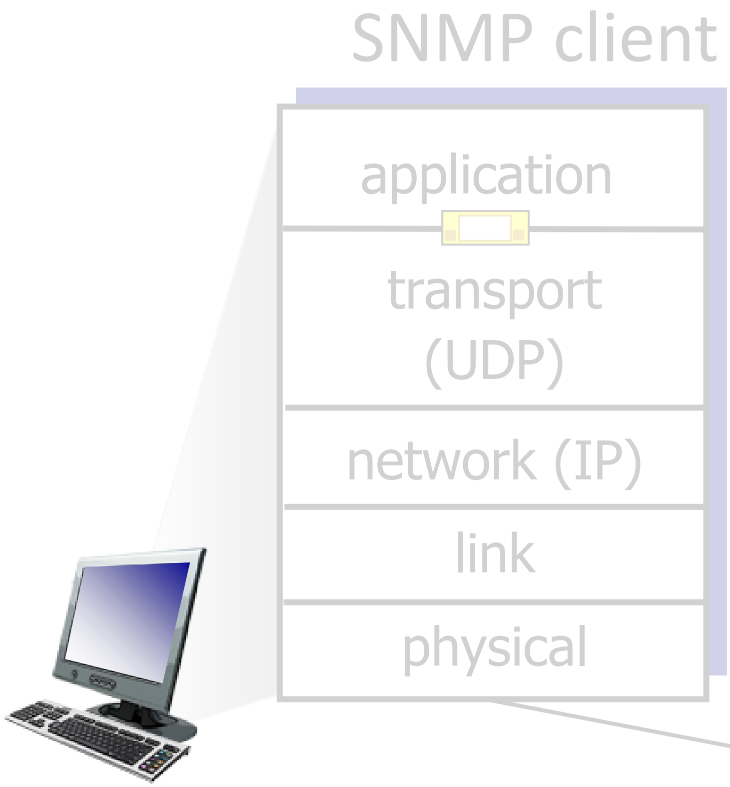
Format

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

UDP: Transport Layer Actions

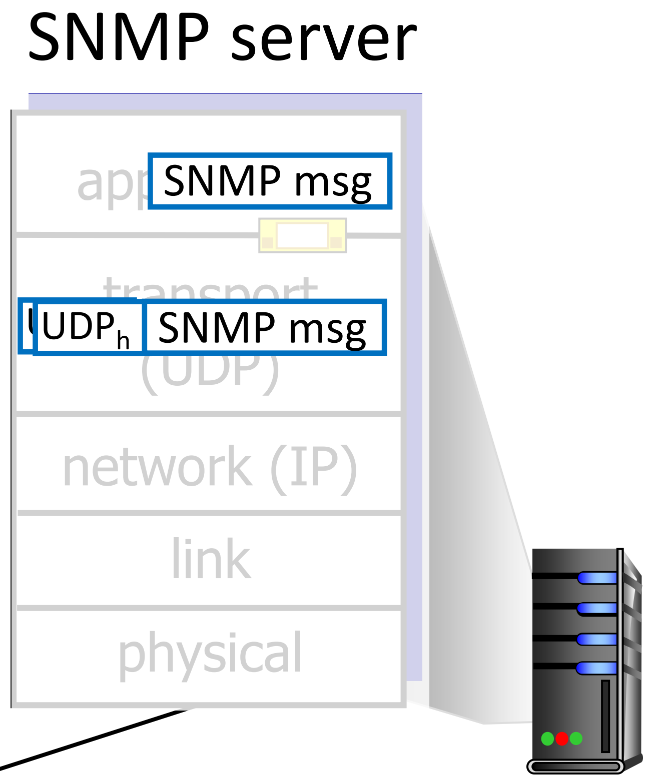


UDP: Transport Layer Actions

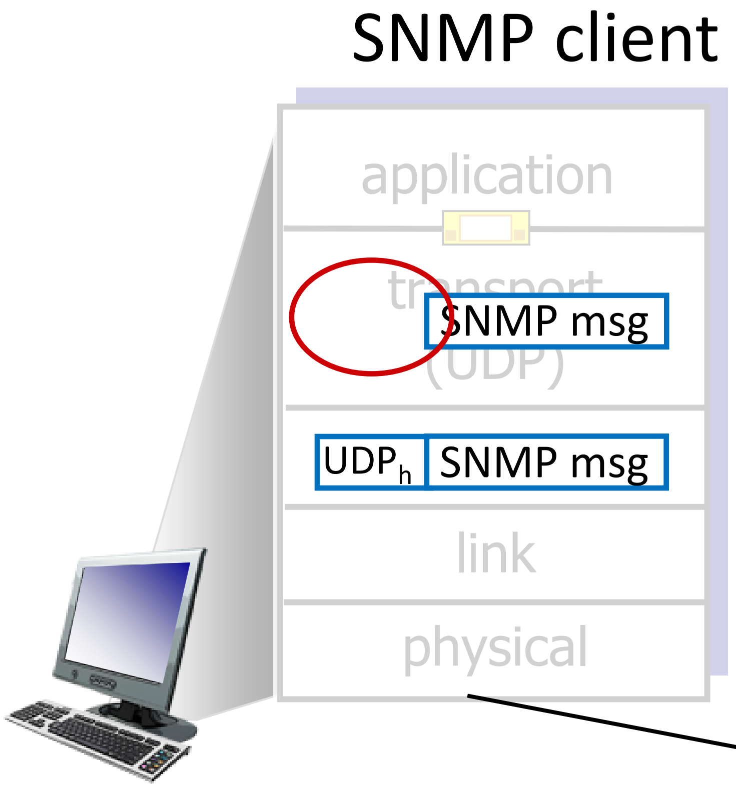


UDP sender actions:

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

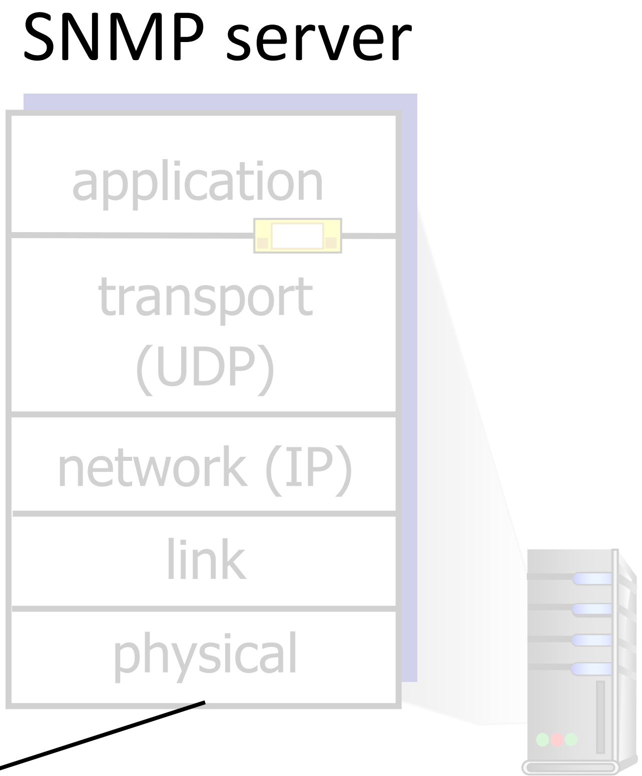


UDP: Transport Layer Actions



UDP receiver actions:

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

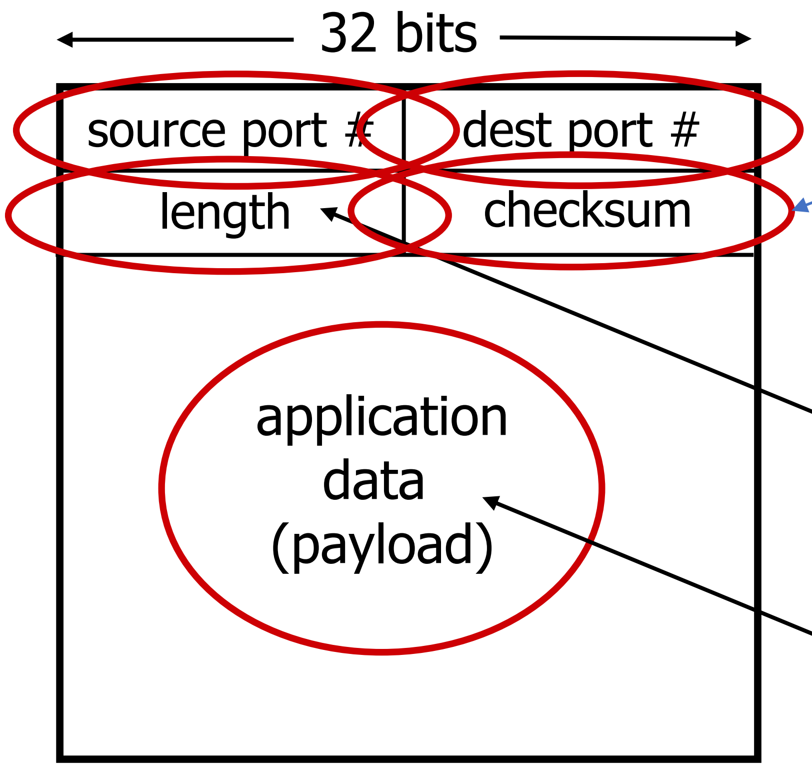


UDP segment header

8 bytes

The UDP header has only 4 fields, each consisting of 2 bytes

The **checksum** is used by the receiving host to check whether errors have been introduced into the segment



UDP segment format

length, in bytes of UDP segment, including header

- Data field may differ from segment to another

data to/from application layer

UDP overhead 8 bytes
TCP overhead 20 Bytes

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)

10

!

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
		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!

Internet Checksum: example-2

example: add two 16-bit integers

		0	0	1	0	0	0	1	0	1	1	0	0	0	0	1	1	
		0	0	0	0	0	1	0	0	1	1	1	0	1	1	0	0	
No wraparound	0	0	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1	If extra 1 appears, wraparound If 0 appeared, nothing to do.
sum		0	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1	
checksum		1	1	0	1	1	0	0	0	0	1	0	1	0	0	0	0	Complement

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/

Summary: UDP

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

Chapter 3: roadmap

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

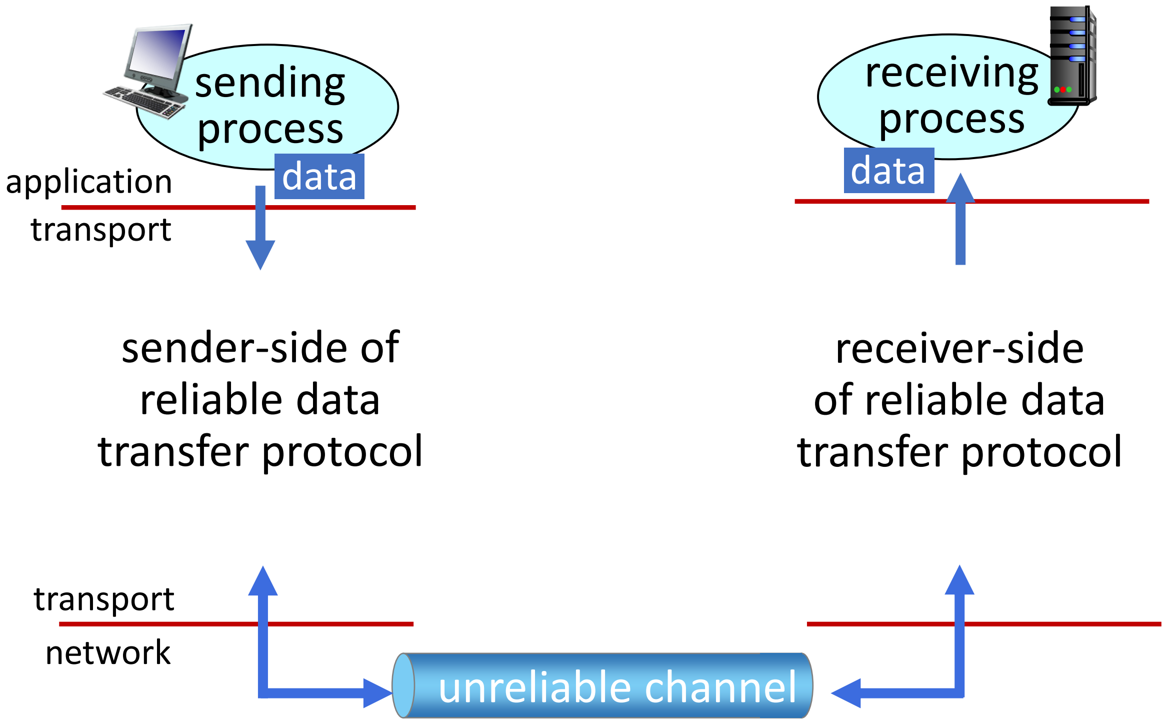
Principles of reliable data transfer

reliable service *abstraction*

reliable data transfer can be performed not only at the transport layer, but also at the link layer and the application layer as well.

Principles of reliable data transfer

- reliable data transfer (RDT) can be performed not only at the transport layer, but also at the link layer and the application layer as well.
  - We will study it here from transport layer perspective
- A protocol with RDT mechanism tries to overcome packet lose, corruption, and out-of-ordering caused by unreliable channel.

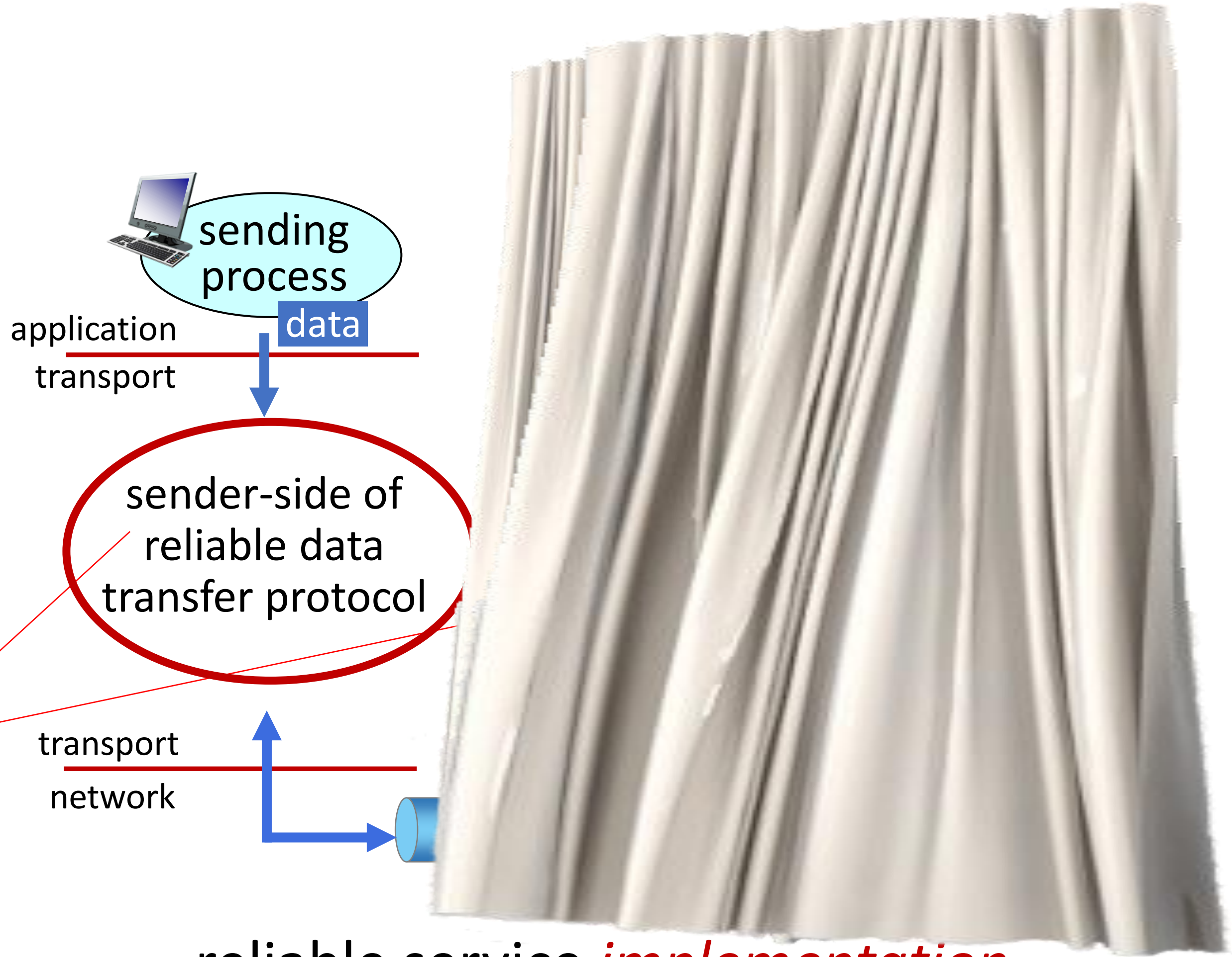


reliable service *implementation*

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: 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!

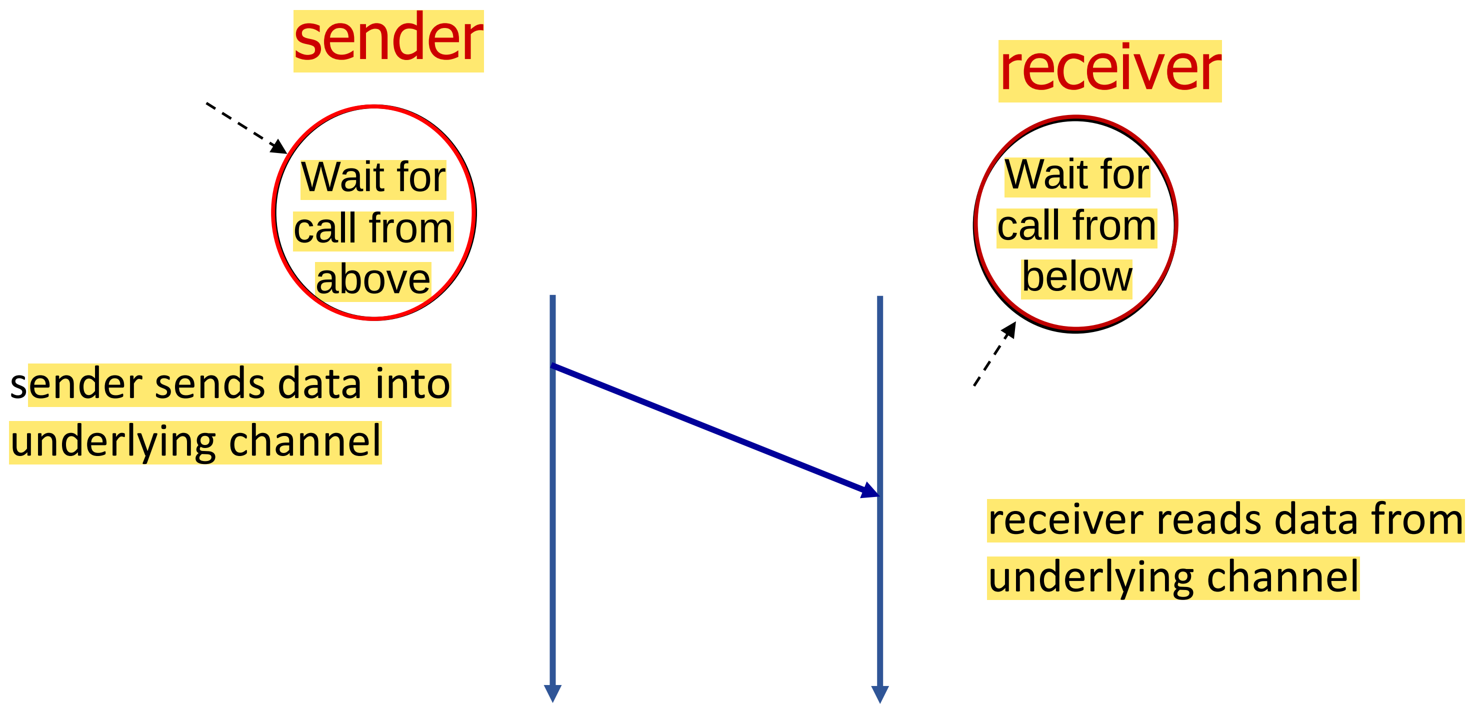
rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

- *No problems!*
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

rdt1.0: Reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
  - receiver reads data from underlying channel



rdt2.0: channel with bit errors

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

How do humans recover from "errors" during conversation?

Assume for now no losses.
Only errors

rdt2.0: channel with bit errors

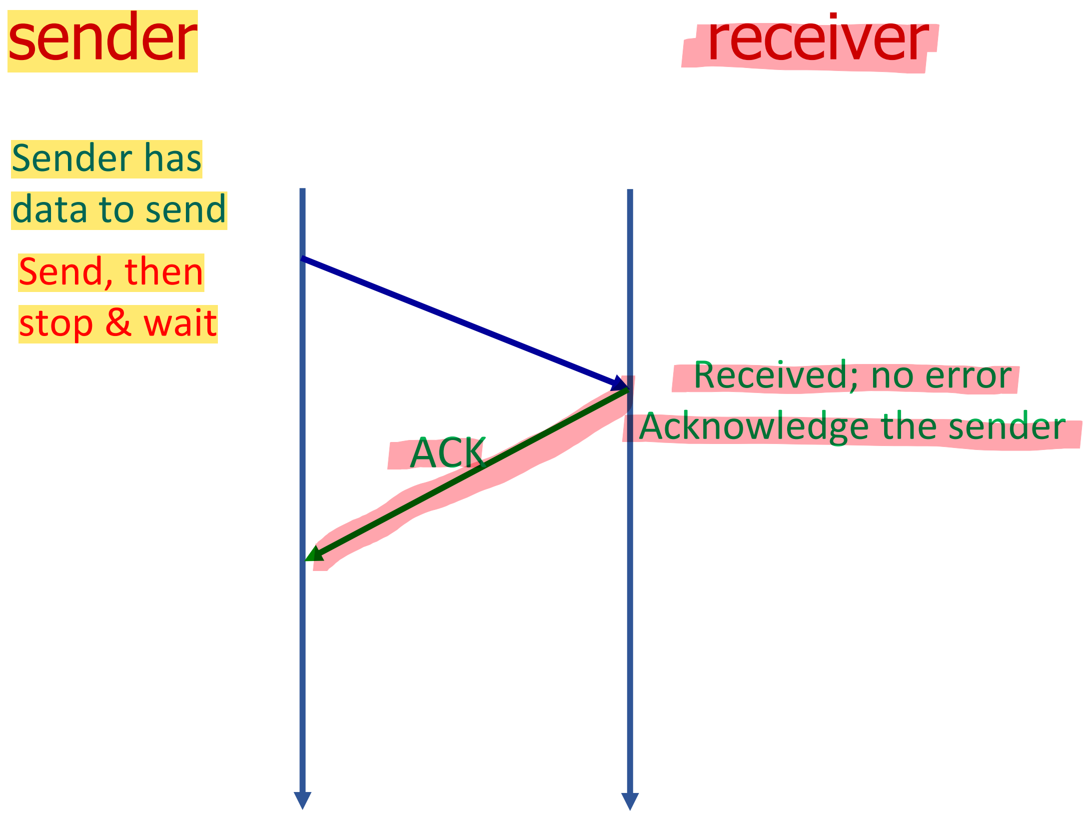
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors? إبعت وصل شكراً
  1 • *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  2 • *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors لو سمحت ممكن تعيد ...
  - sender *retransmits* pkt on receipt of NAK

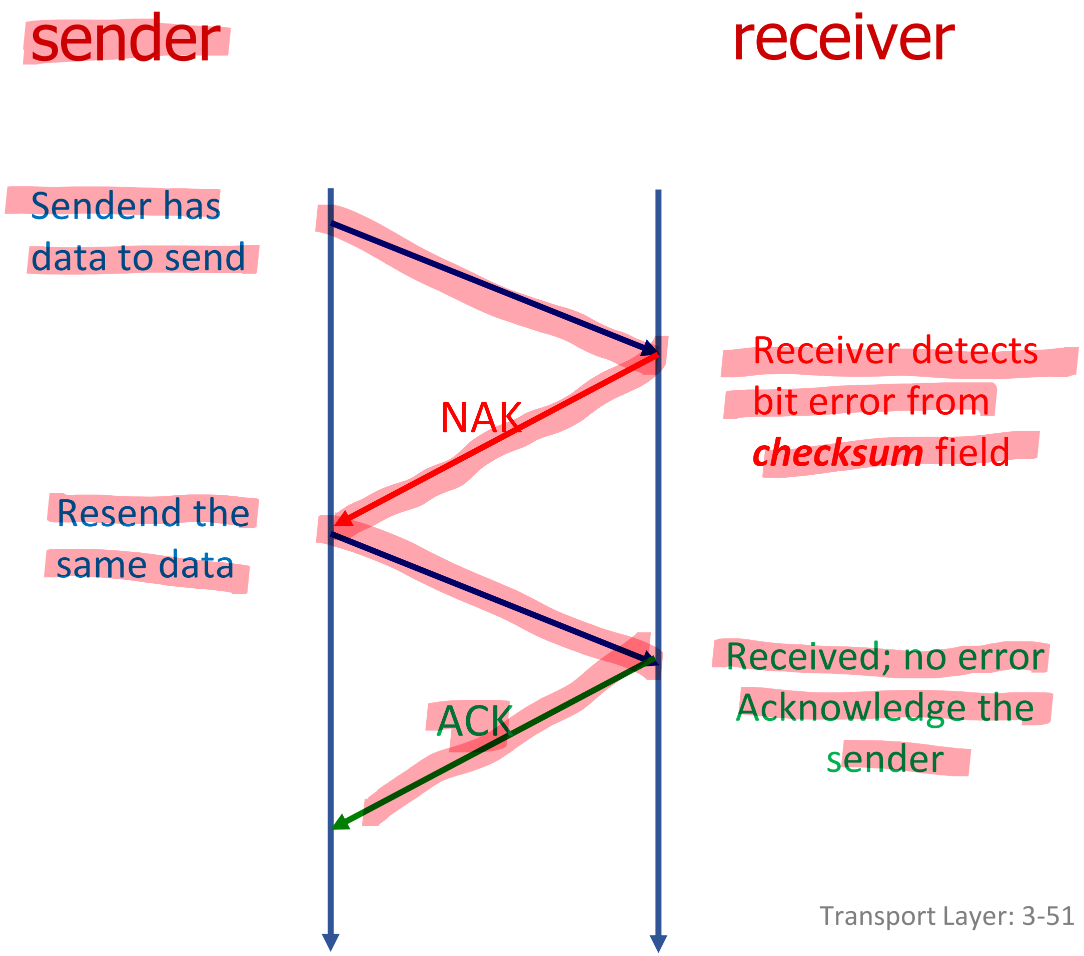
stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
- checksum detects bit errors
- *the* question: how to recover from errors?

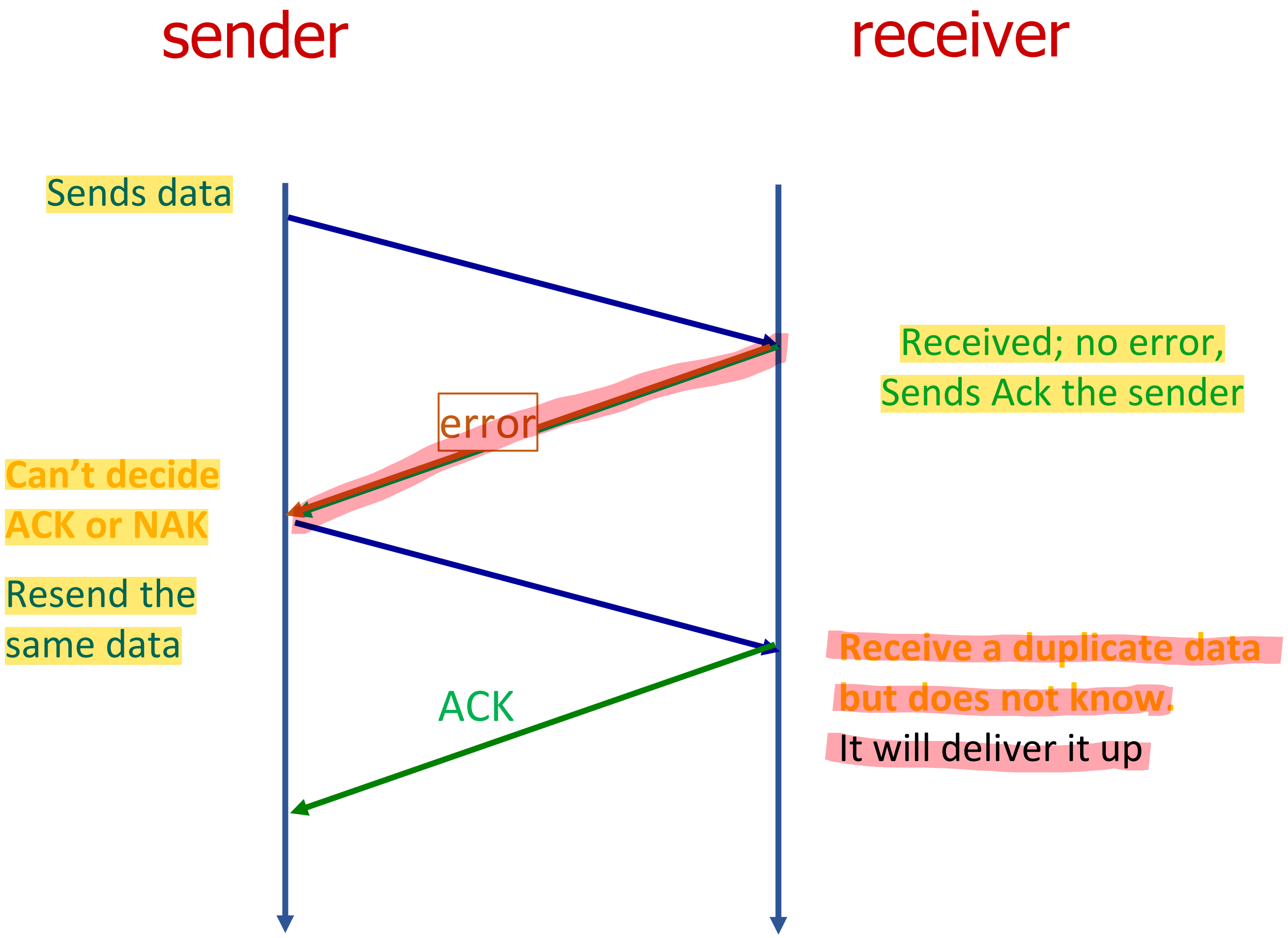




rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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



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

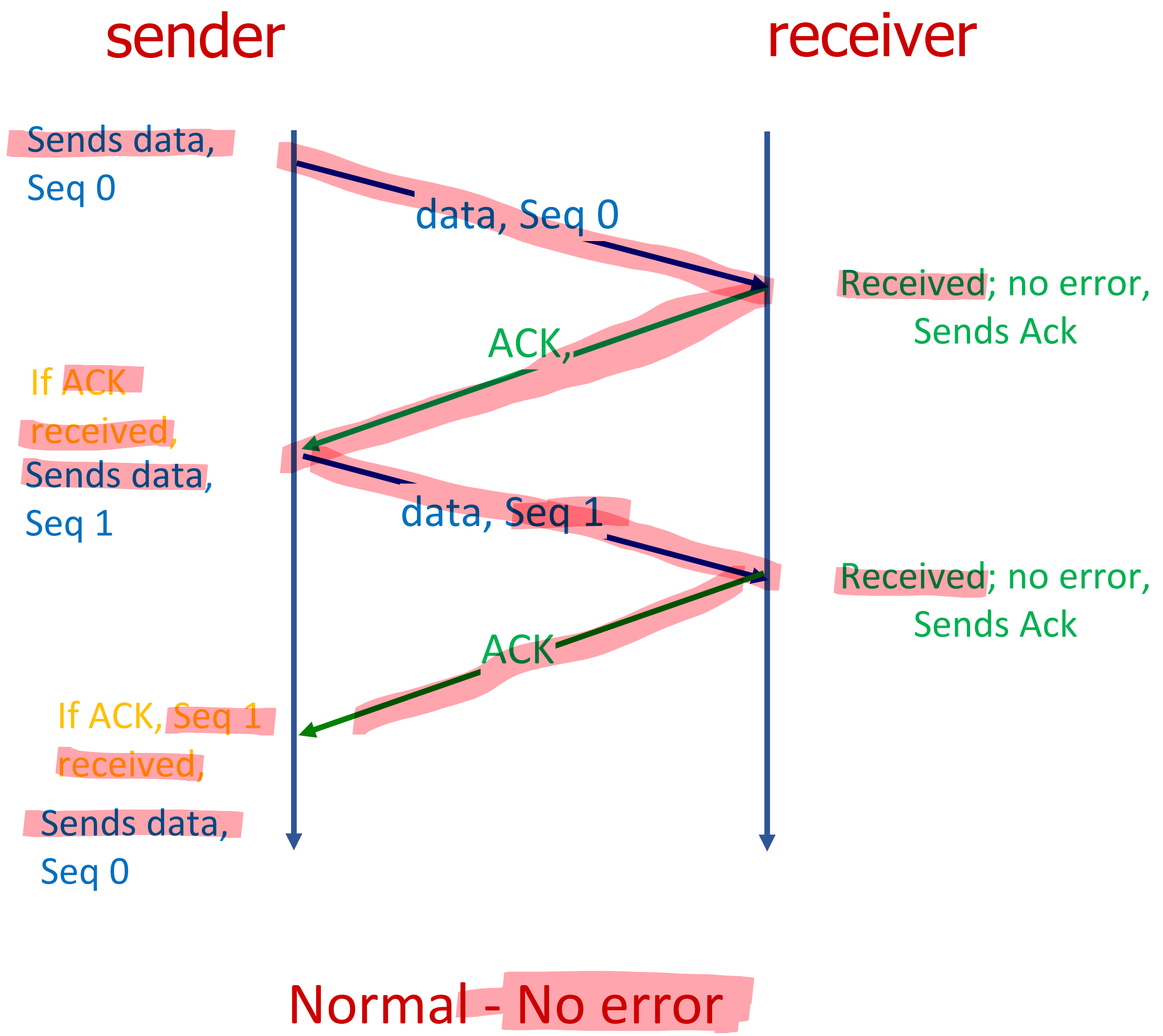
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: ACK/NAK + sequence number

handling duplicates:

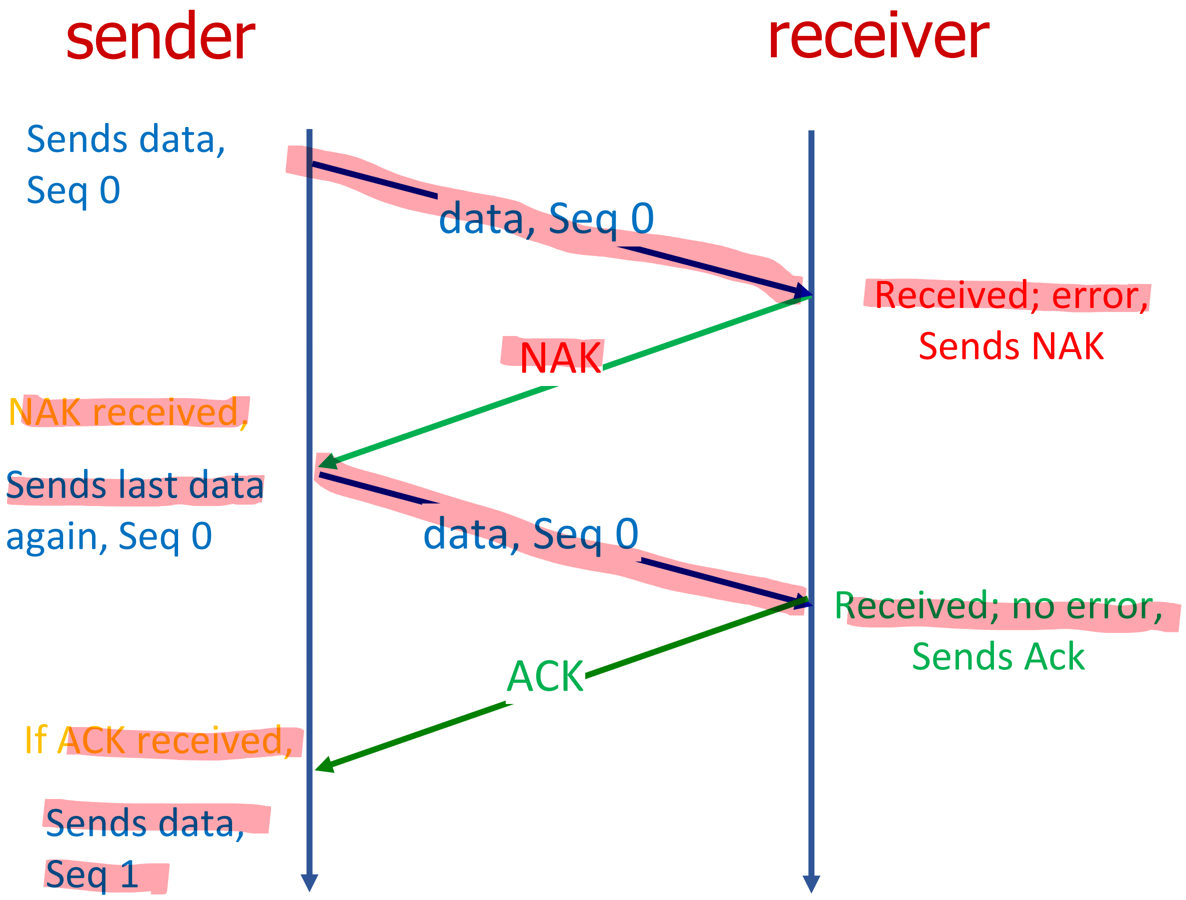
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt



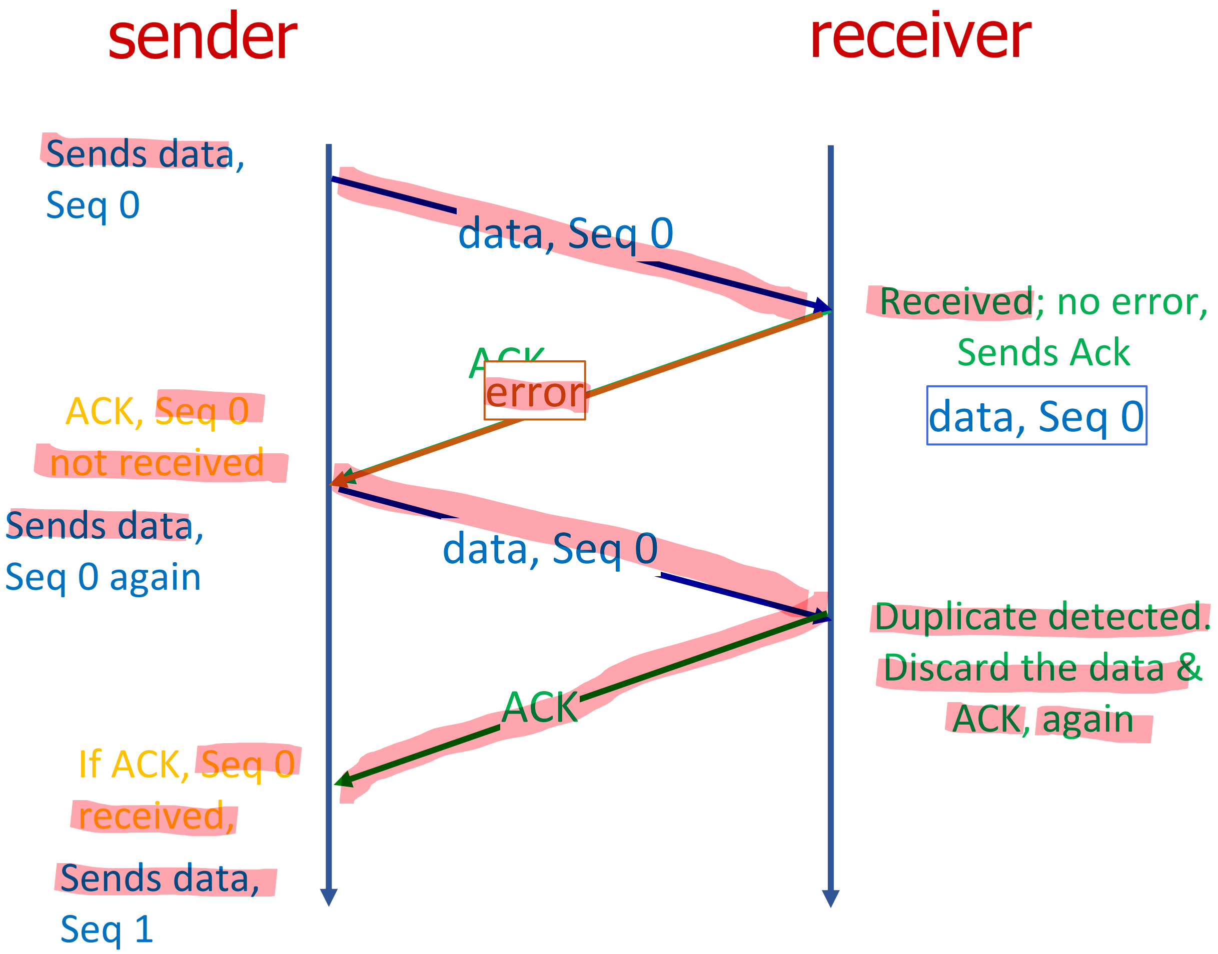
stop and wait

sender sends one packet, then stops & waits for receiver response

rdt2.1: ACK/NAK + sequence number



Corrupted data at receiver



Corrupted ACK/NAK at sender

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:

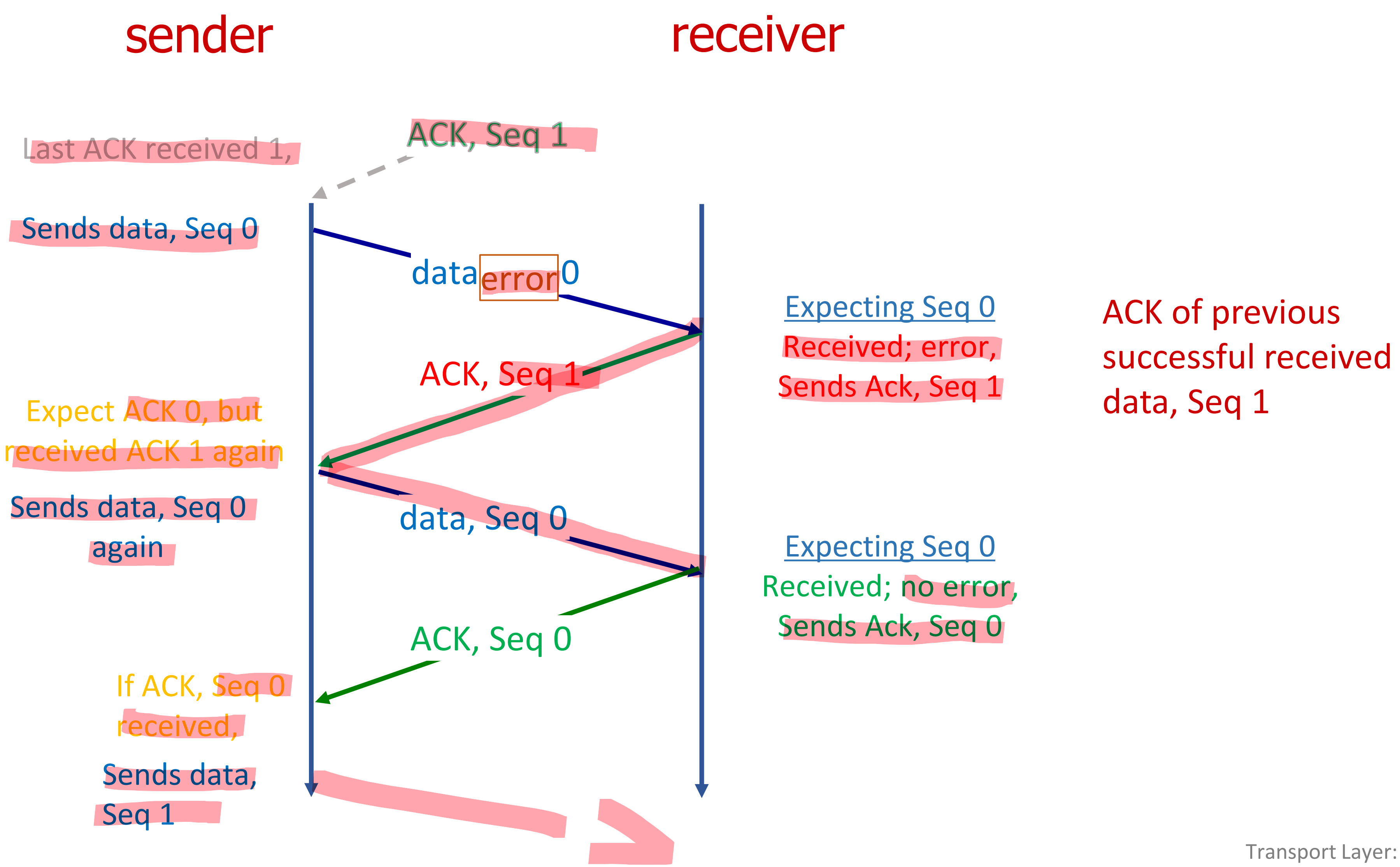
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

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 a NAK-free protocol



Dupliacte ACK 1 = NAK
i.e., re-send last sent data

rdt3.0: channels with errors *and* loss

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

- 1 checksum, 2 sequence #s, 3 ACKs, 4 retransmissions will be of help … but not quite enough

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

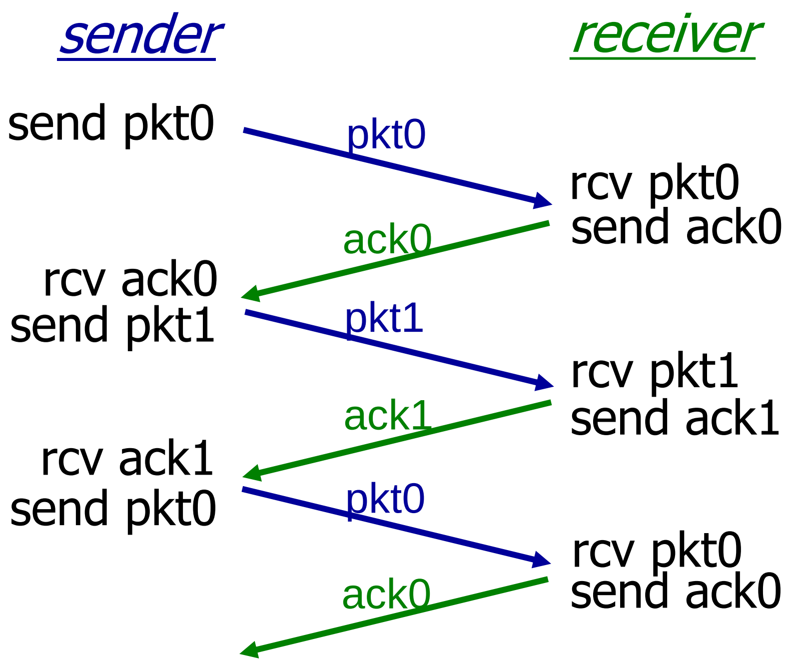
rdt3.0: channels with errors *and* loss

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

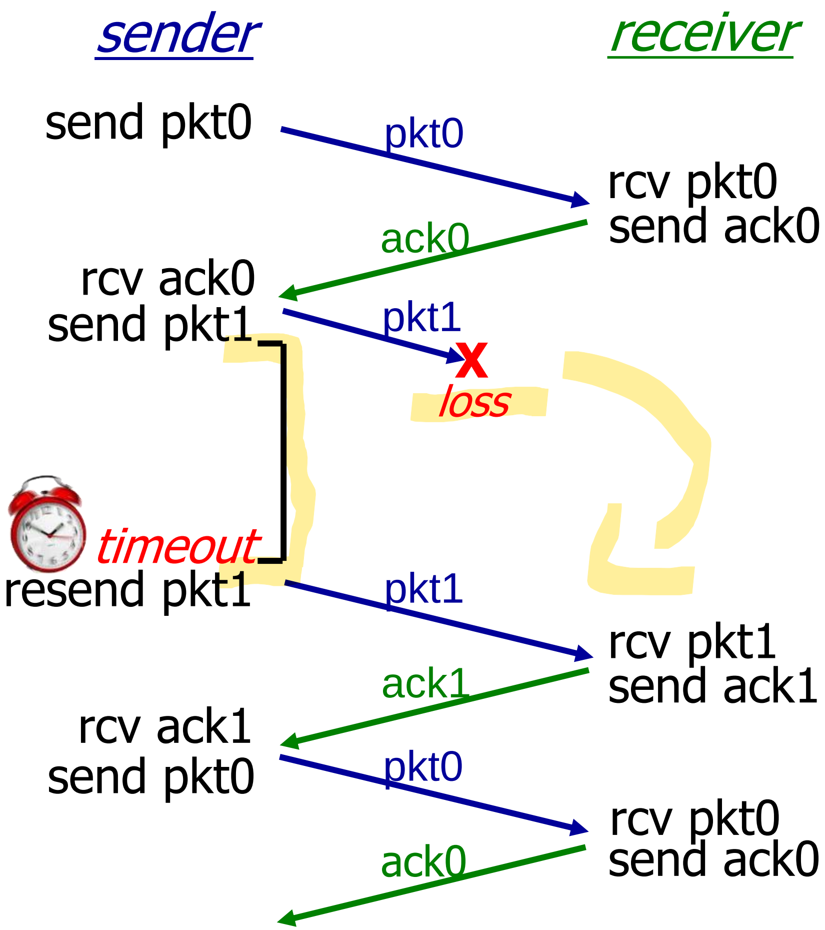
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

timeout

rdt3.0 in action

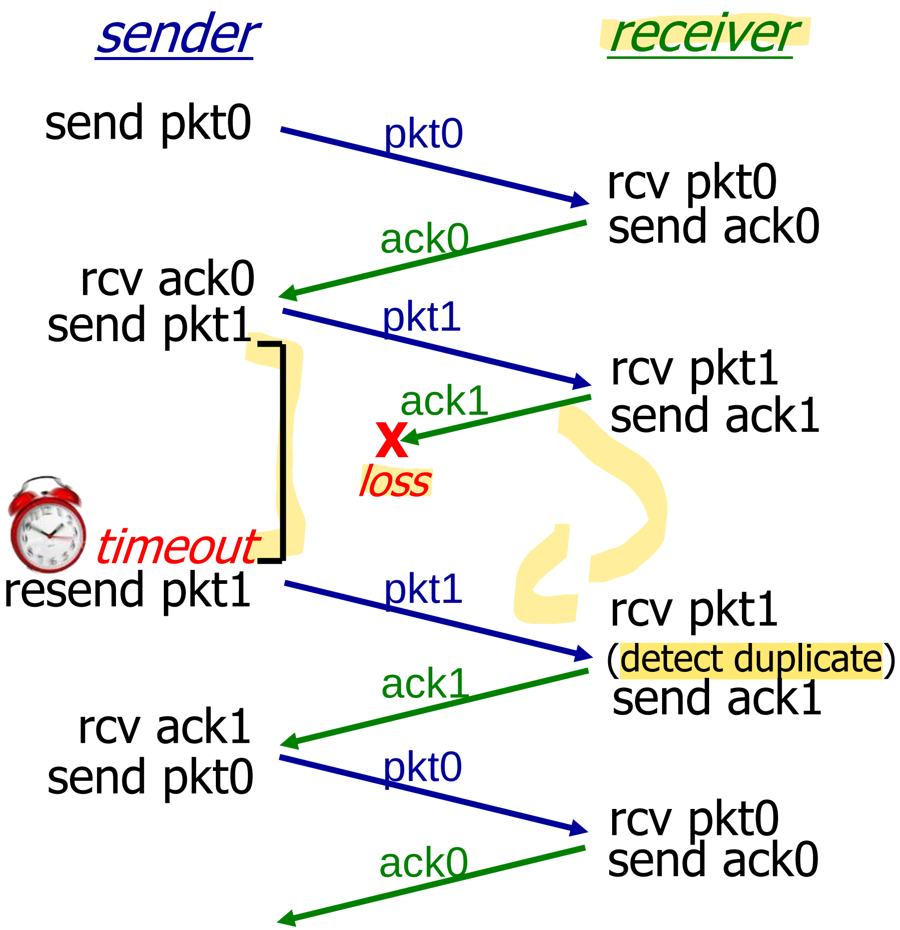


(a) no loss

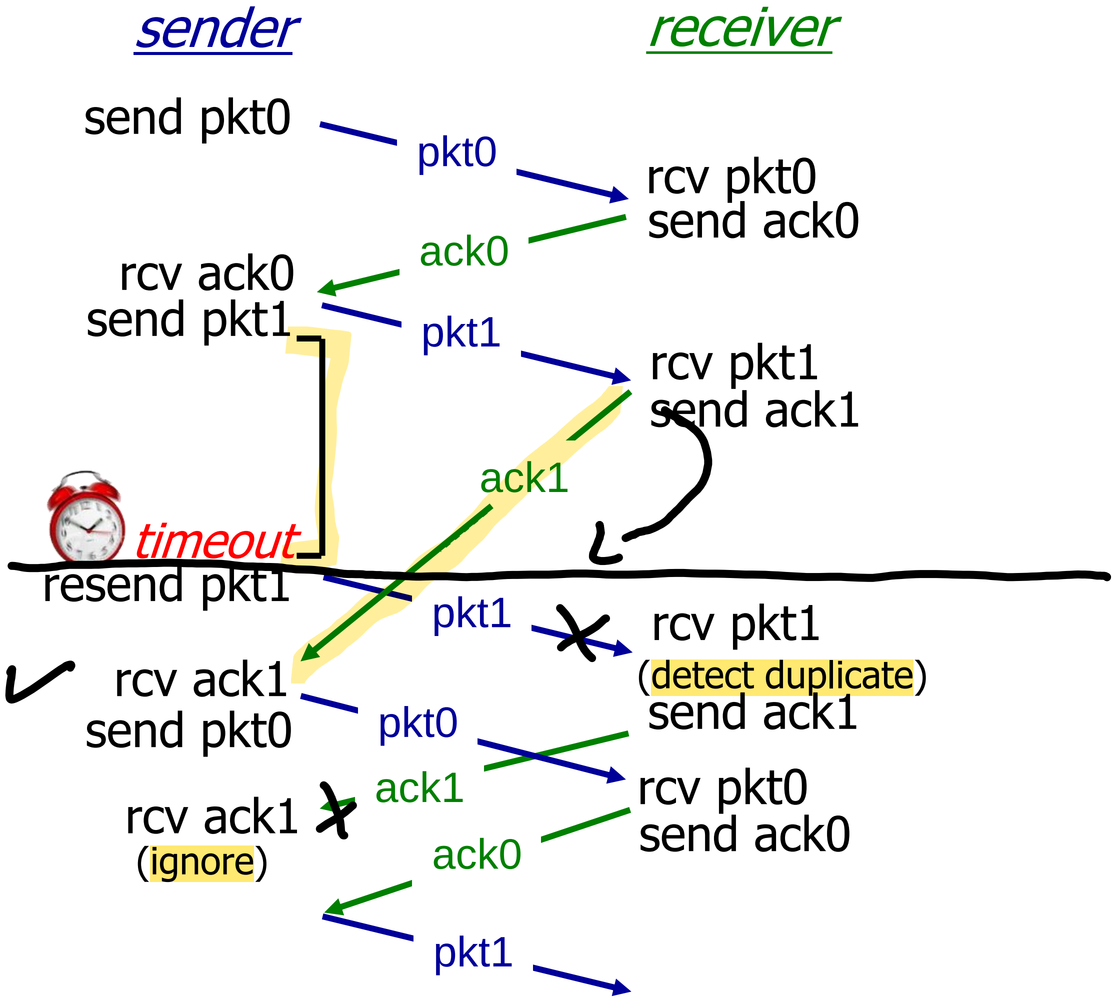


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

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:

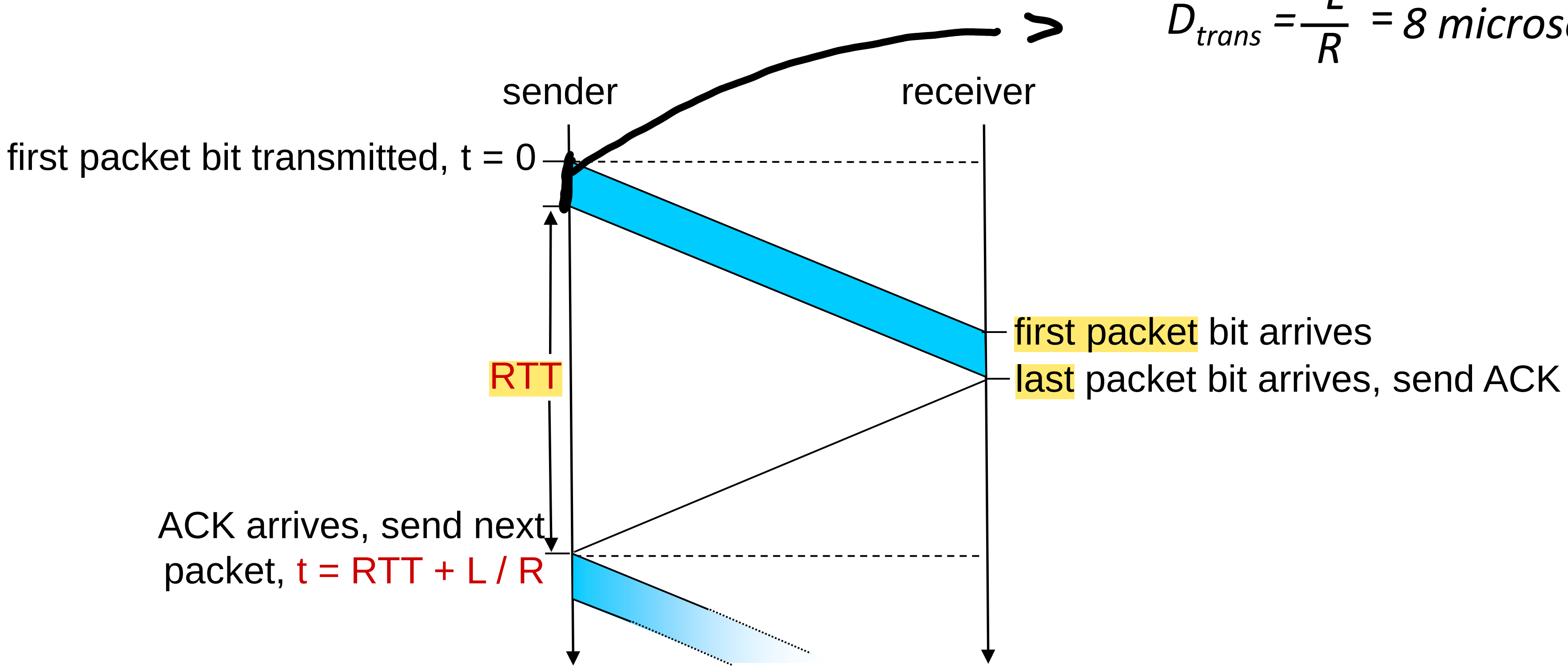
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - RTT= 2 x 15 ms one-way prop. delay = 30ms

rdt3.0: stop-and-wait operation

RTT= 2 x 15 = 30ms

$D_{trans} = \frac{L}{R} = 8\ microsecs$



rdt3.0: stop-and-wait operation

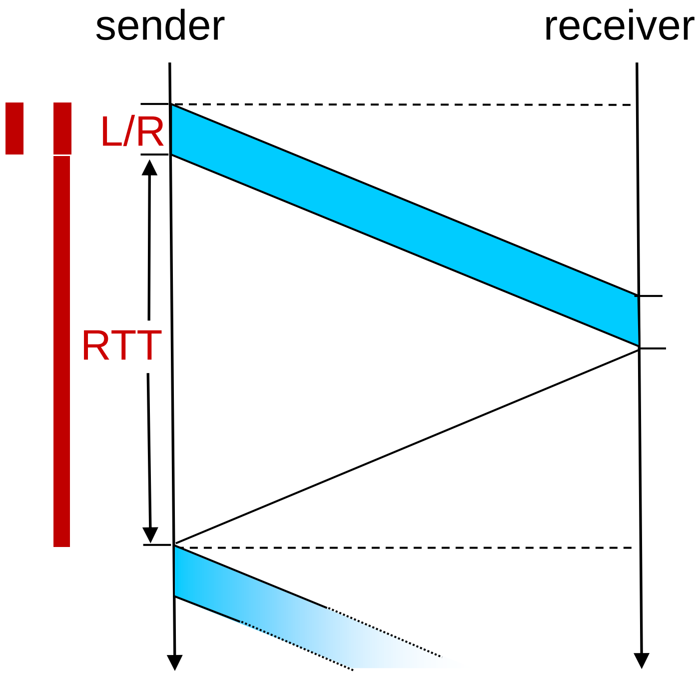
RTT= 2 x 15 = 30ms

$$D_{trans} = \frac{L}{R} = 8 \text{ microsecs}$$

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

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

$$= 0.00027$$

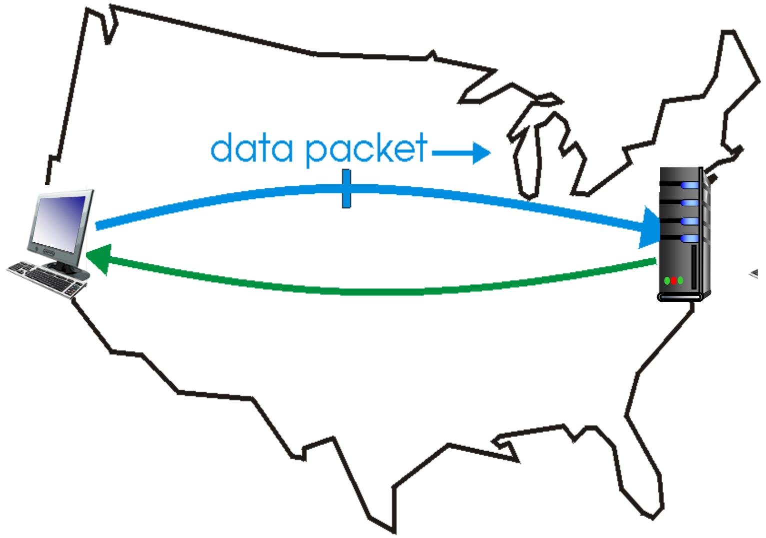


- *rdt 3.0 protocol performance stinks!*
- *Protocol limits performance of underlying infrastructure (channel)*

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

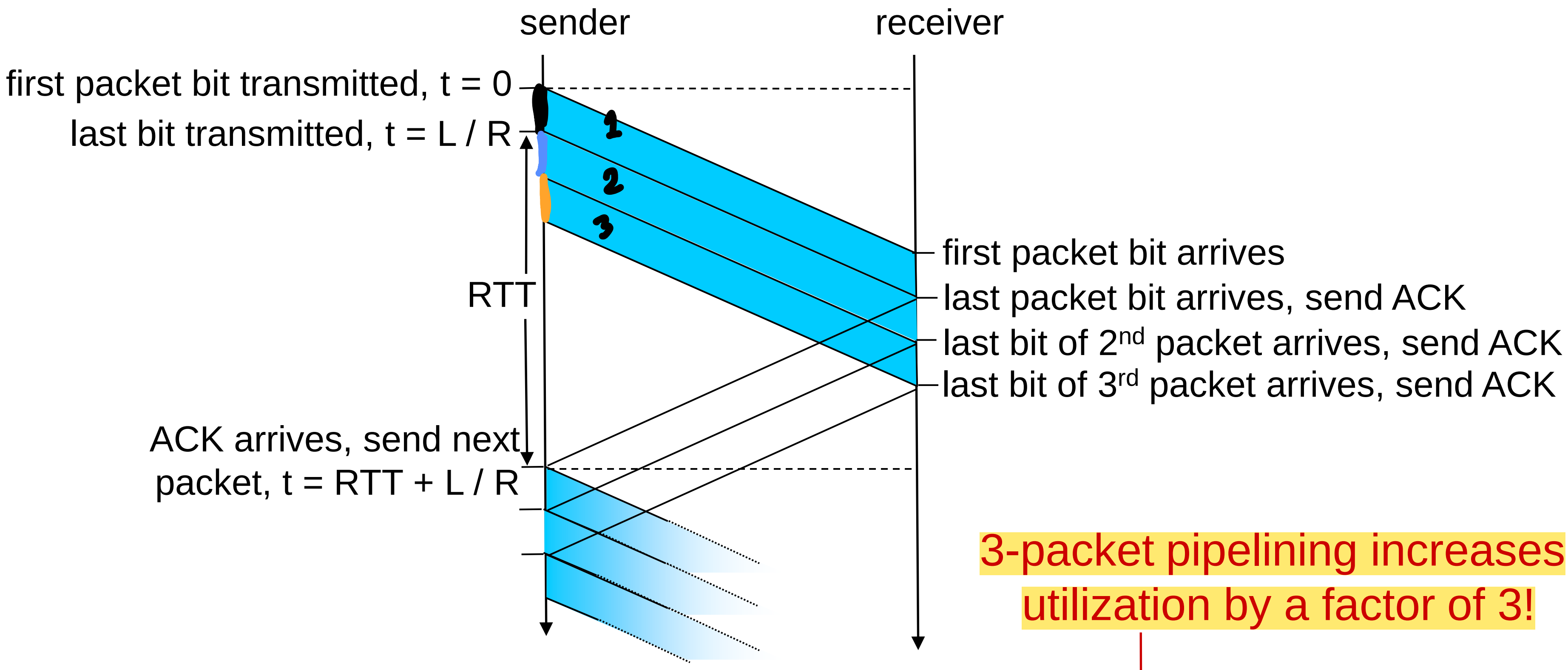


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

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

1 2

Pipelining: increased utilization



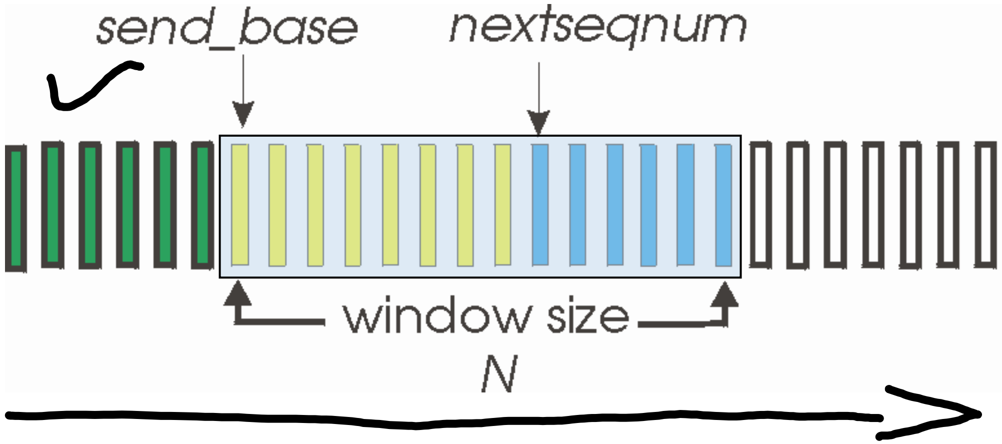
3-packet pipelining increases utilization by a factor of 3!

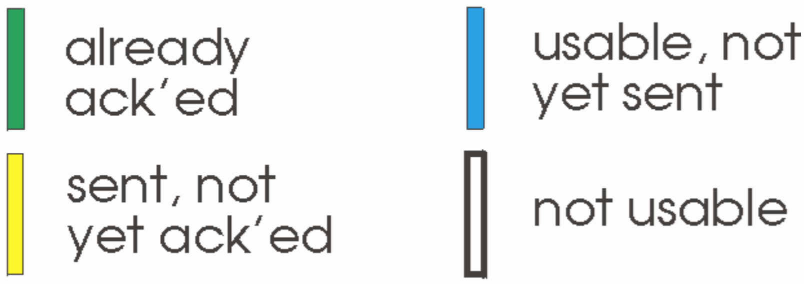
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: sender

متتابعة/متوالية

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - a k-bit sequence number in packet header





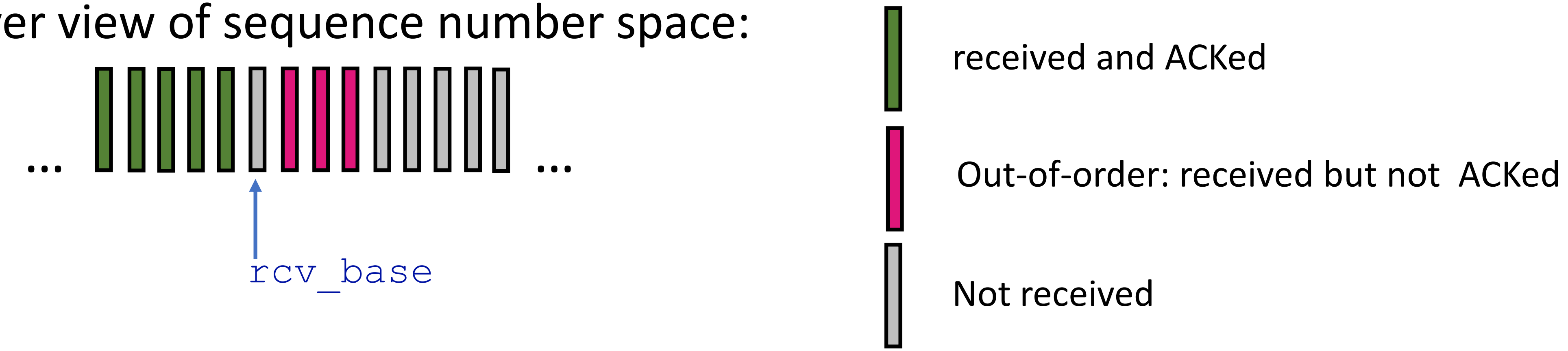
Send_based: Oldest unACKed seq#
Nextseqnum: 1st unused seq# i.e., next pkt to send

- ***cumulative ACK:*** ACK(*n*): ACKs all packets up to *n*, 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: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, and all packets w/ seq# >n restart timer

Cumulative ACK(*n*):

- mark packet *n* and all older packets in window as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

Packets

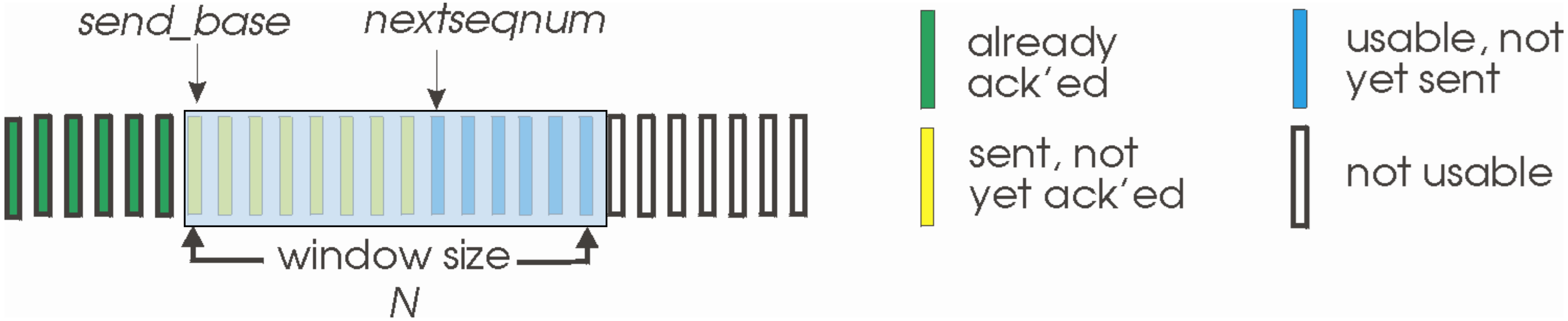
- must be received in-order.
- Always remember oldest unACKed packet.

packet *n*

- out-of-order: discard ~~or buffer~~
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet
- If all packets older than *n* are received, send cumulative ACK(n)

Go-Back-N: Sender

- The sender can transmit multiple packets (when available) without waiting for an acknowledgment but constrained to have no more than N of unacknowledged packets in the pipeline.
- *sliding-window protocol*: when packets inside window ACKed, window moves by the same amount of these packets.
- an ACK for a packet with sequence number *n* will be taken to be a cumulative acknowledgment; all packets with a sequence number less than *n* have been correctly received at the receiver.
- When ***timeout(n)****:* occurs for packet n**,** retransmit packet *n* and packets in window with seq # > n (even if they already have been sent)

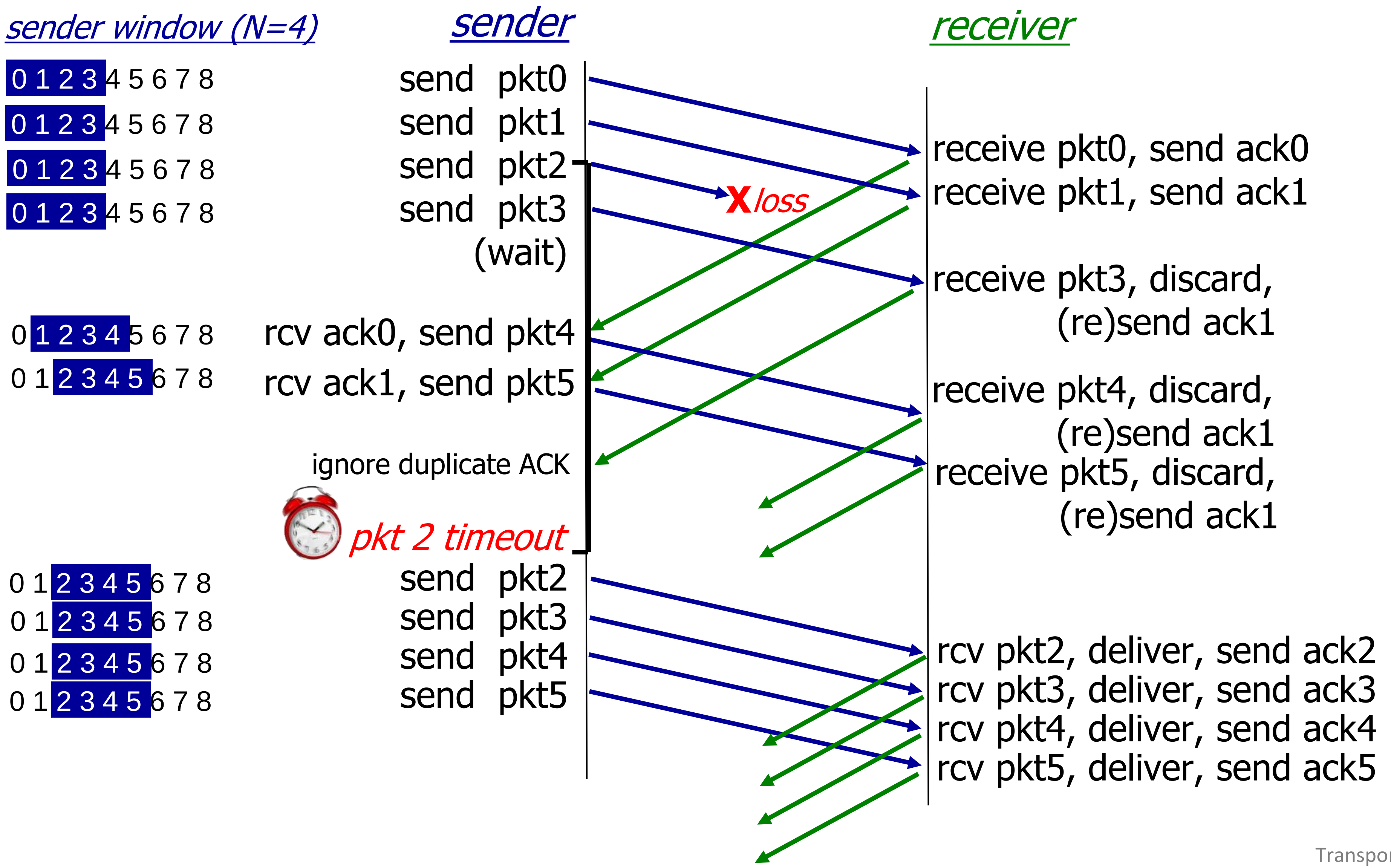


Go-Back-N: Reciever

- If a packet with sequence number *n* is received correctly and is in order, send an ACK for packet *n* and deliver the data to the upper layer.
- Hence, ACK(n) means that all packet older than *n* have been successfully received and delivered to upper layer.
- When receiving Out-of-order packets, can discard (or buffer);
  - Missing packet when retransmitted after a timeout, all newer packets will be transmitted.

- https://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198702.cw/index.html
- https://www2.tkn.tu-berlin.de/teaching/rn/animations/gbn_sr/

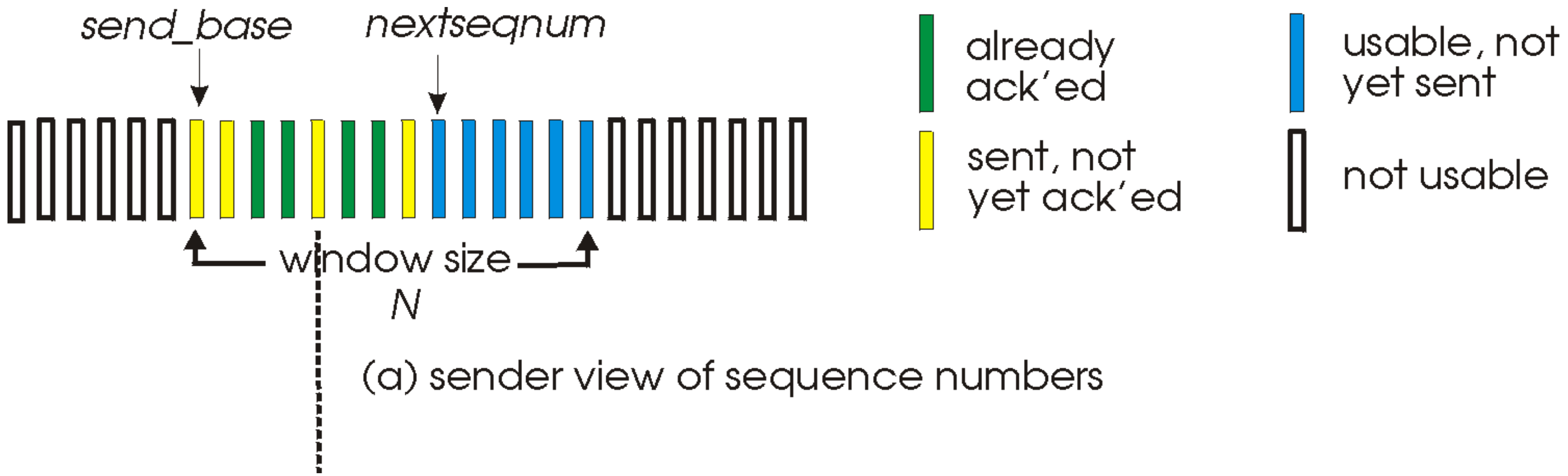
Go-Back-N in action



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



(a) sender view of sequence numbers

Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

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

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

receiver

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

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

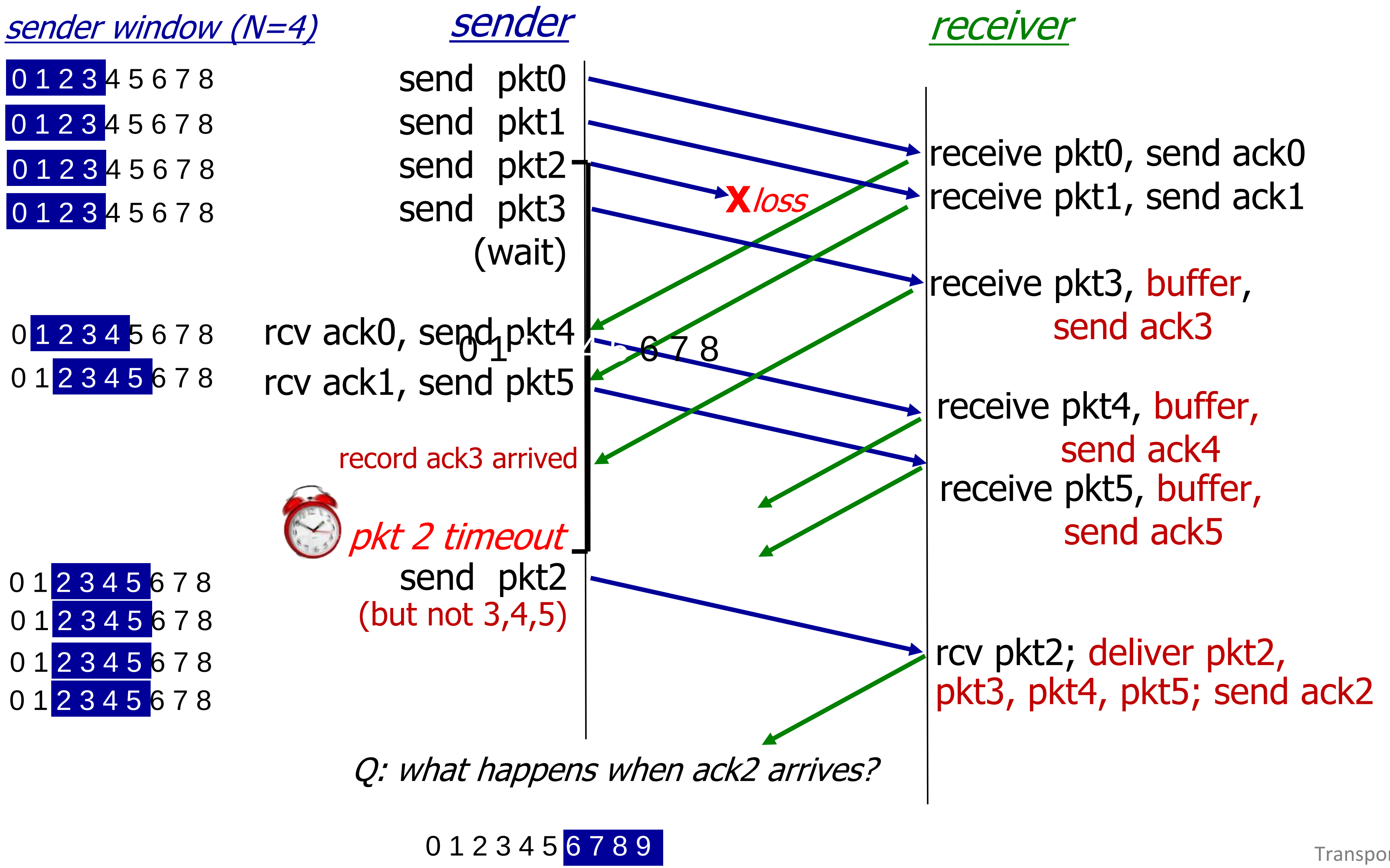
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action



Chapter 3: roadmap

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

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

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

TCP: Overview

- **3-way handshake:**
  - Sender sends a special segment; receiver replies with a special segment.
    - Segments carry *no-payload*
  - A third segment is sent from sender (may have payload)
  - TCP connection is established.

  - During handshake:
    - Buffer size is determined
    - MSS negotiated (not actually done)
    - Window size negotiation
    - Sequence numbers

- **TCP buffers**
  - Process sends application msgs (stream of data) into socket.
  - TCP directs data to send buffer.
  - From time to time, TCP grabs chunks of data from buffer and pass to network layer.

- **MSS:** maximum segment size
  - The maximum amount of application data placed in a segment.
    - not including TCP header
    - Related to largest frame & MTU. (Later!)
    - Typical MSS is 1460 bytes

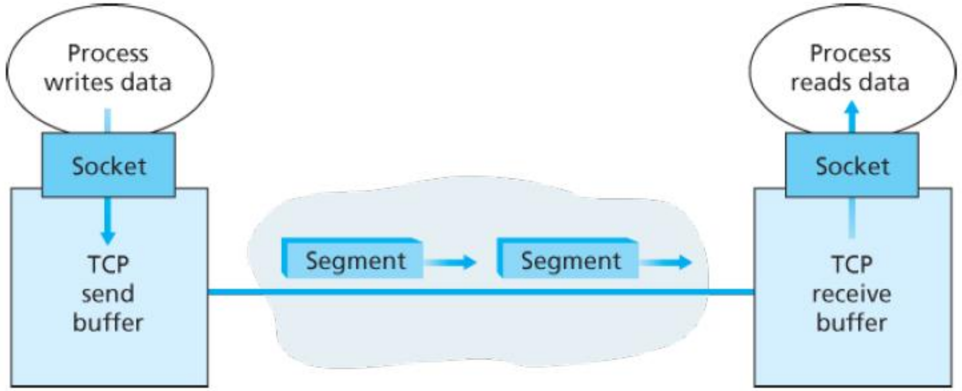
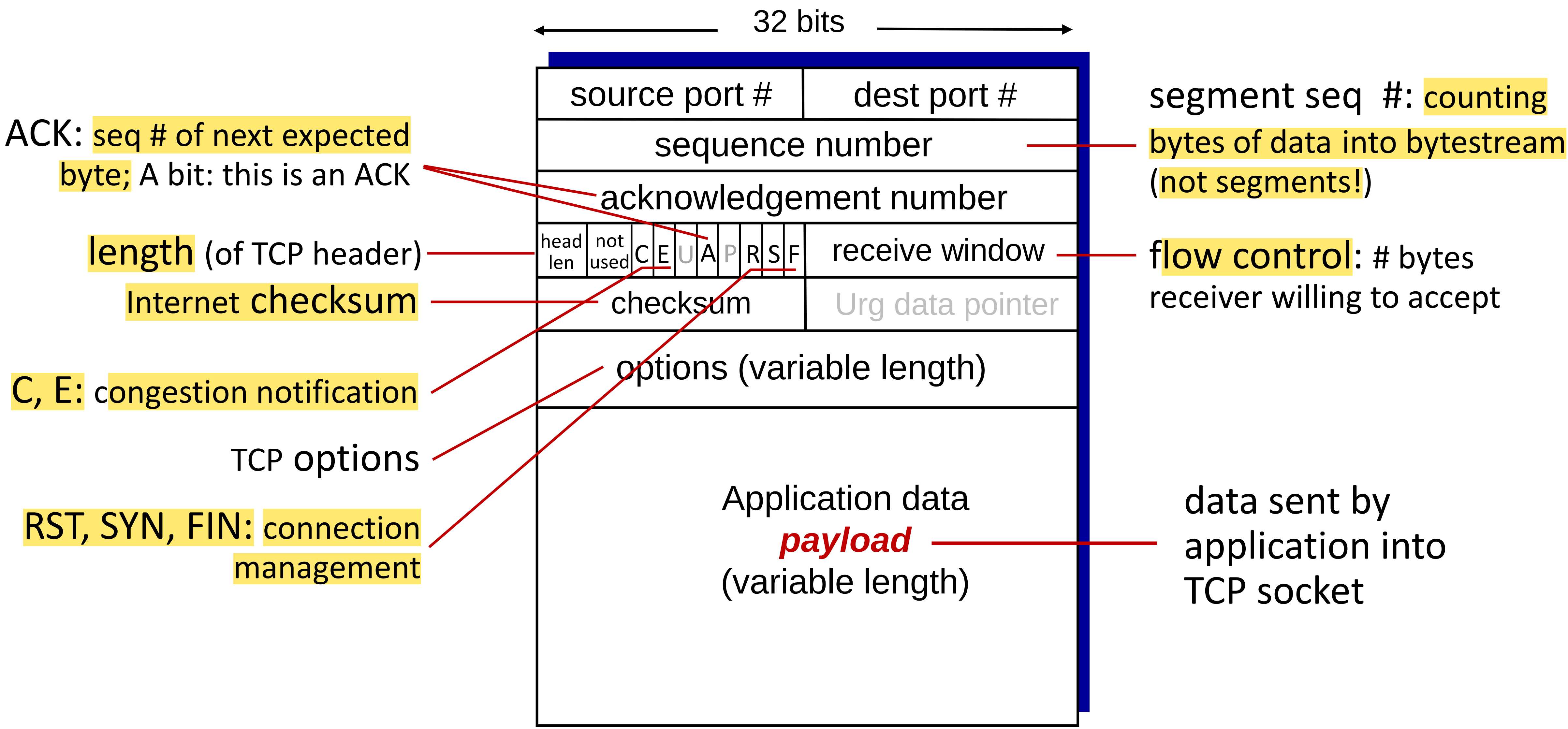


Figure 3.28 TCP send and receive buffers

TCP segment structure



TCP segment structure

Typically, the options field is empty, so that the length of the typical TCP header is 20 bytes.

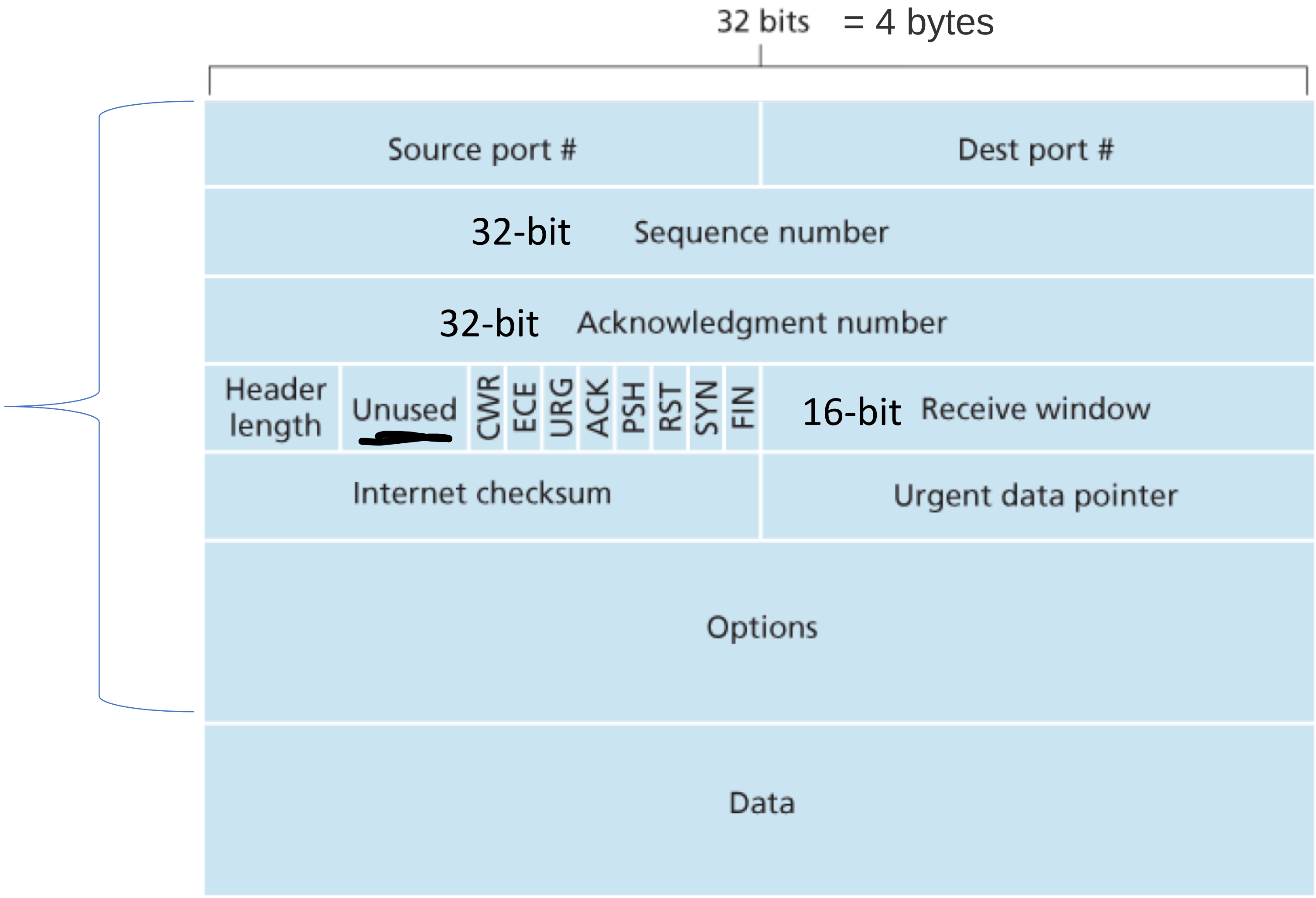


Figure 3.29 TCP segment structure

TCP sequence numbers, ACKs

- TCP views data as an unstructured, but ordered, stream of **bytes**.
- TCP's use of sequence numbers reflects this view in that
  - sequence numbers are over the stream of transmitted bytes and **not** over the series of transmitted segments.

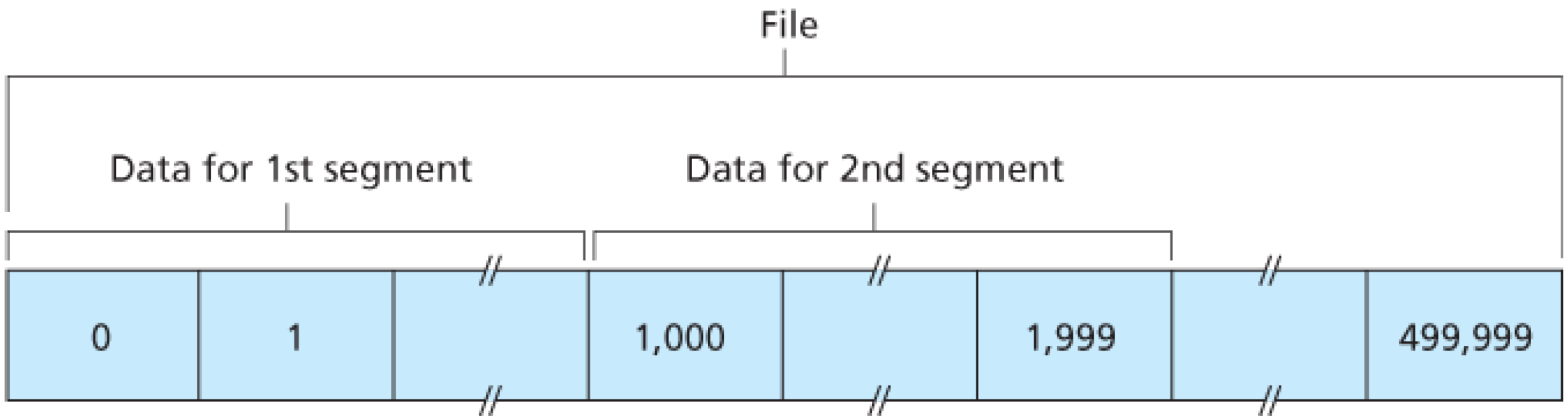


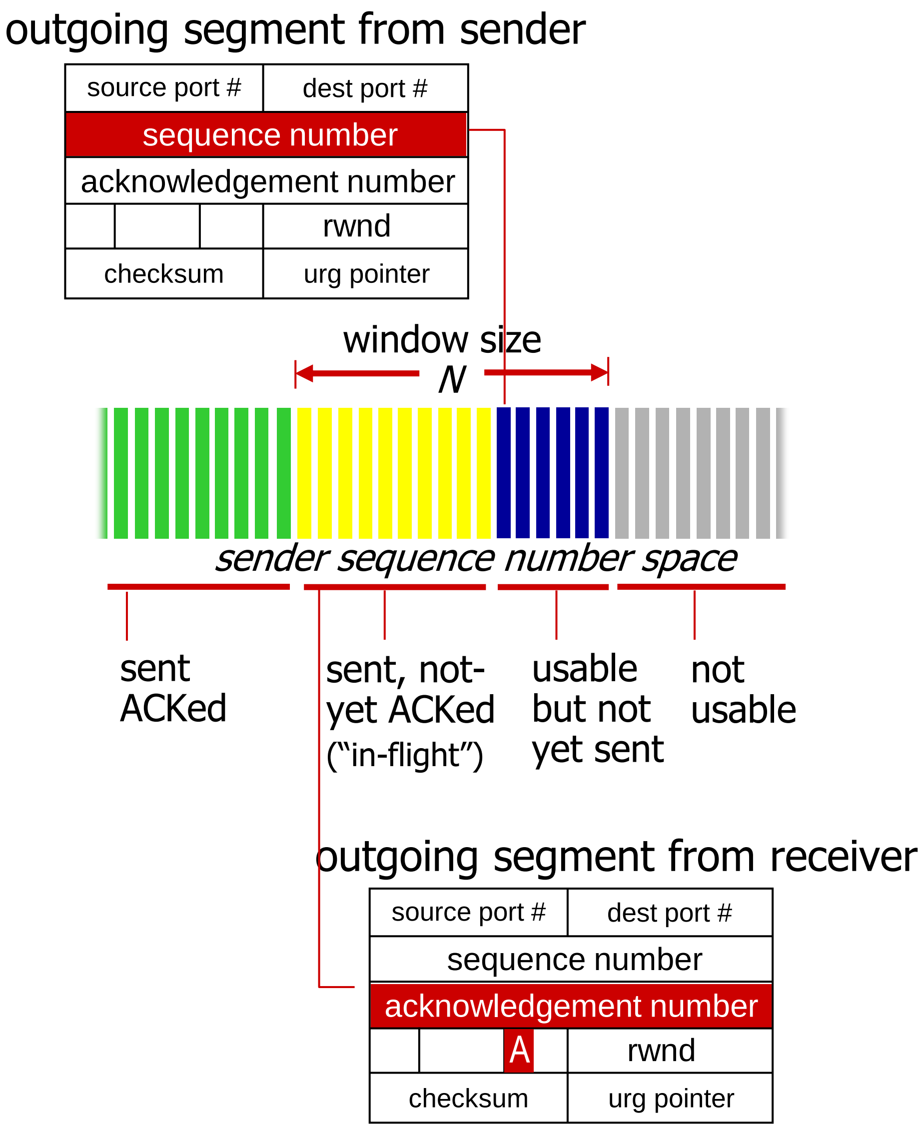
Figure 3.30 Dividing file data into TCP segments

TCP sequence numbers, ACKs

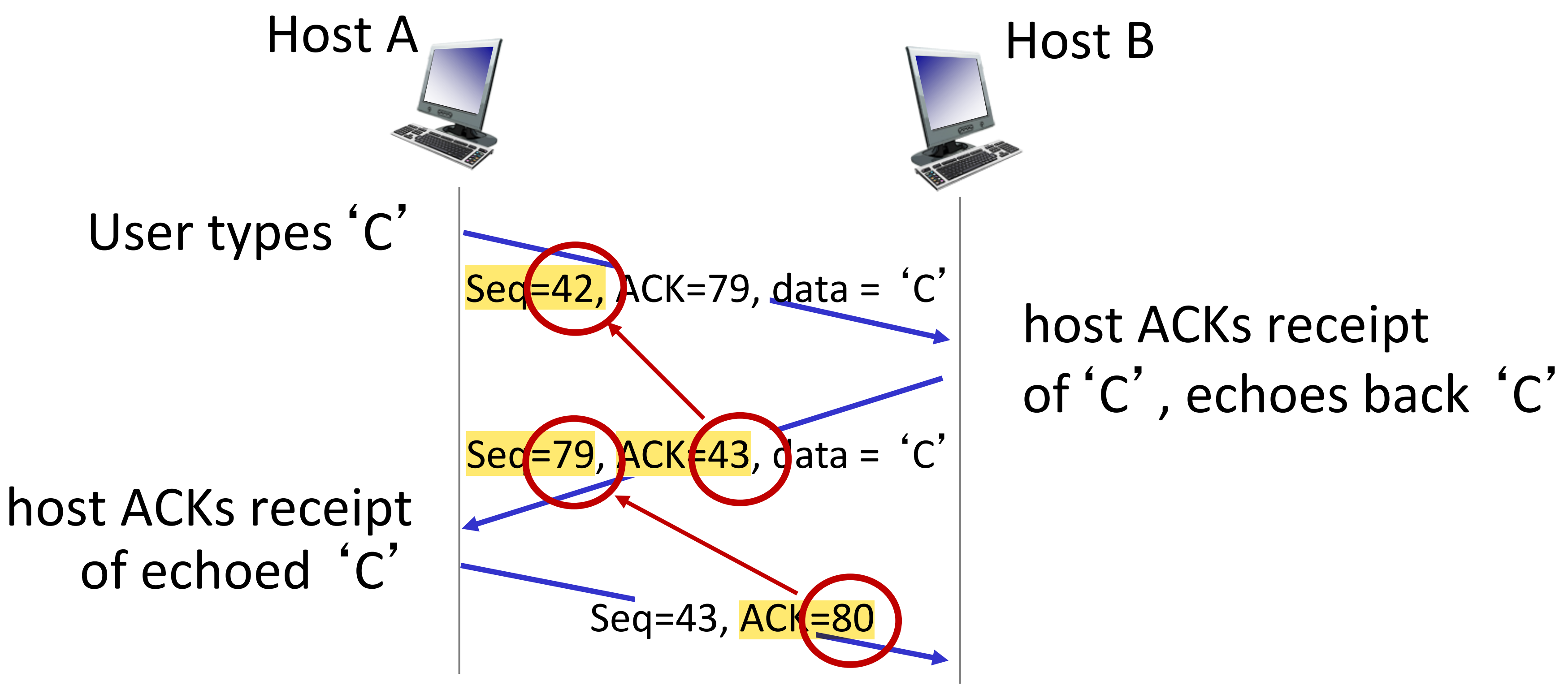
Sequence numbers:
- byte stream "number" of first byte in segment's data

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
- Combination of GBN & SR



TCP sequence numbers, ACKs



simple telnet scenario

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's reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is
  1 • uncorrupted,
  2 • without gaps,
  3 • without duplication, and
  4 • in sequence;
- that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection.

- *TCP adopts many of **RDT** principles.*

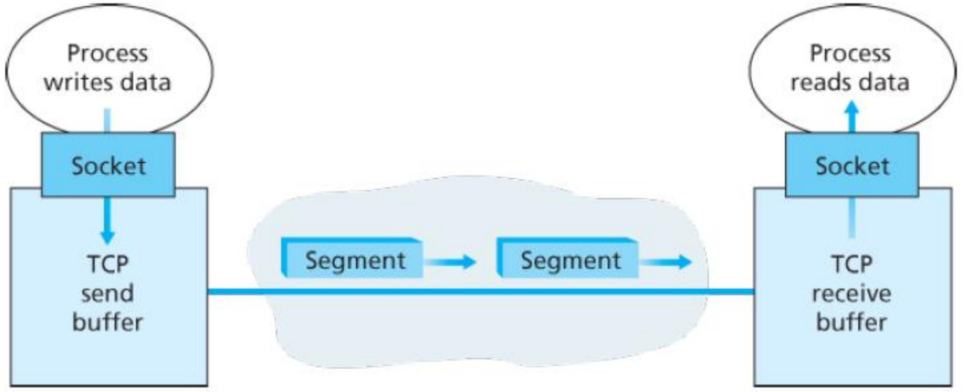


Figure 3.28 TCP send and receive buffers

TCP reliable data transfer

- TCP creates process-to-process reliable transfer of segments on top of IP's unreliable host-to-host delivery of IP datagram
  1 • *pipelining* of segments
  2 • *cumulative* ACKs
  3 • single retransmission *timer*

- retransmissions triggered by:
  1 • *timeout* events
  2 • *duplicate* acks

let's initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control, congestion control

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

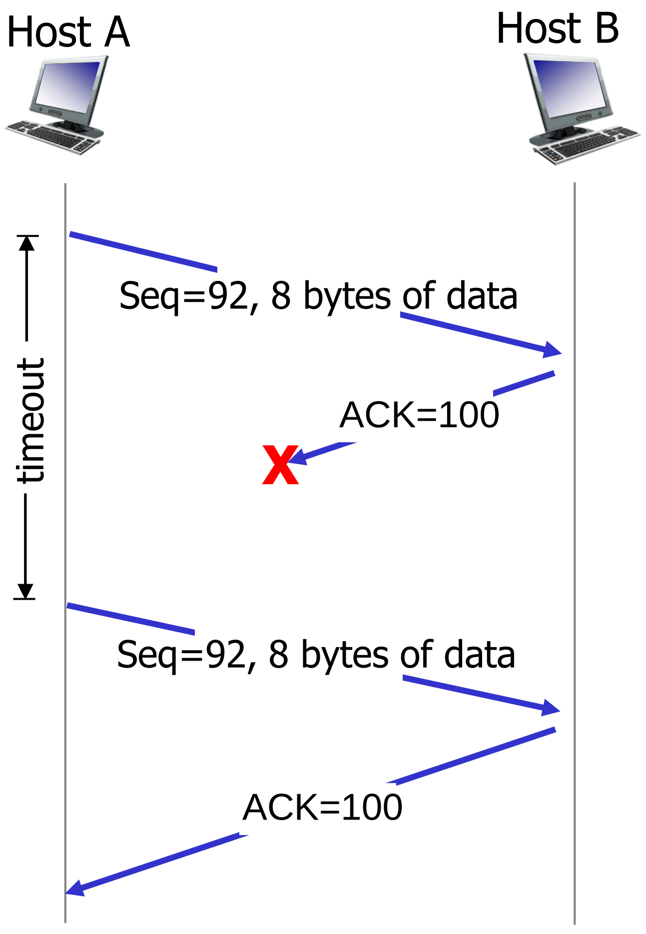
event: ACK received

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments
- TCP acks (cumulatively) correctly received
- out-of-order segments are not individually ACKed by the receiver

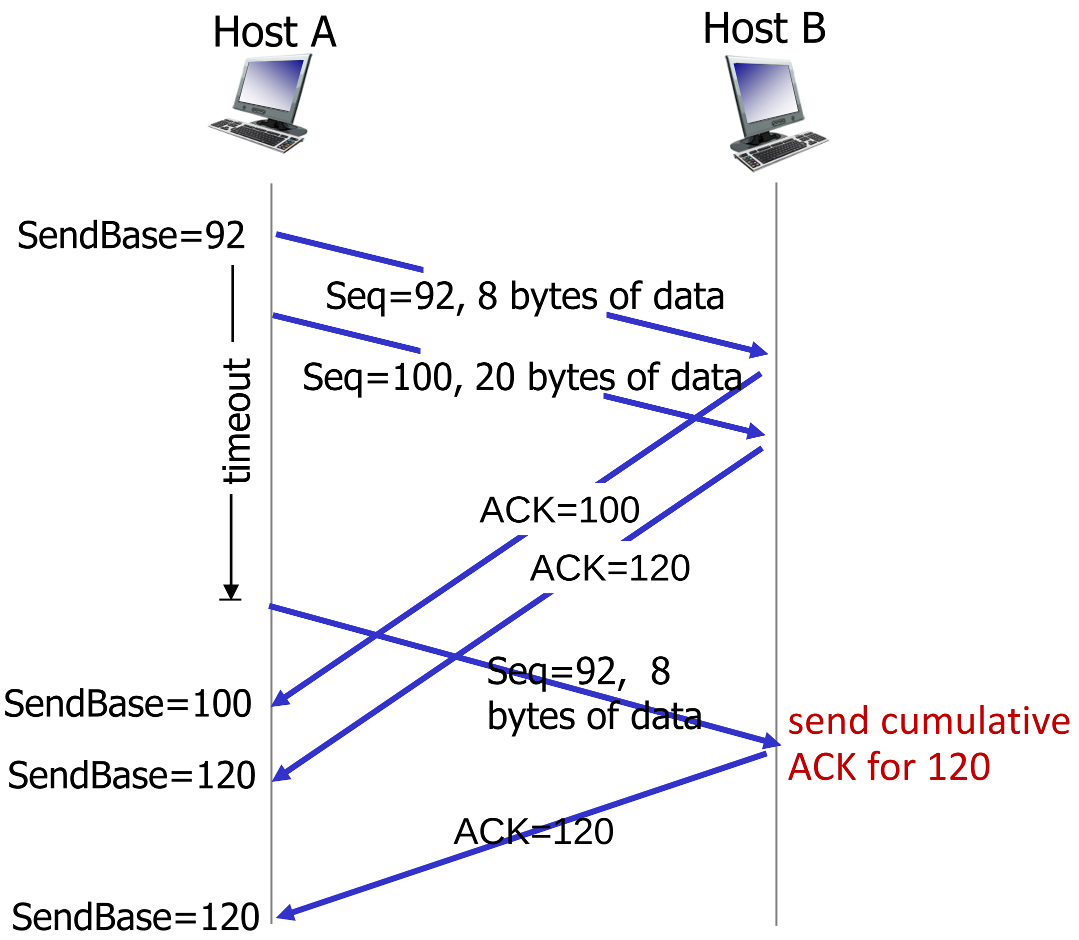
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*	
		Delayed ACK
		Cumulative ACK
		duplicate ACK

TCP: retransmission scenarios

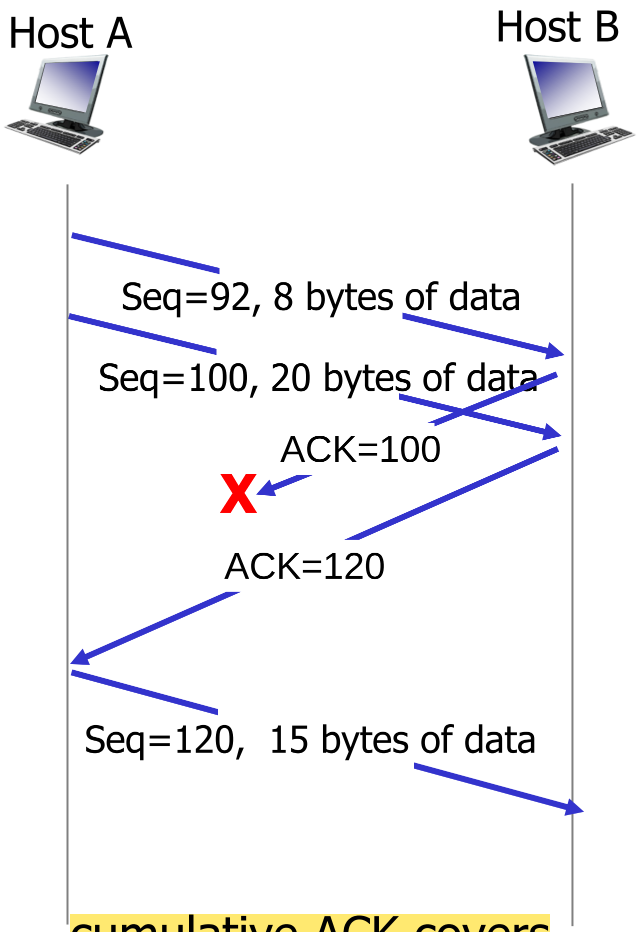


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK covers
for earlier lost ACK

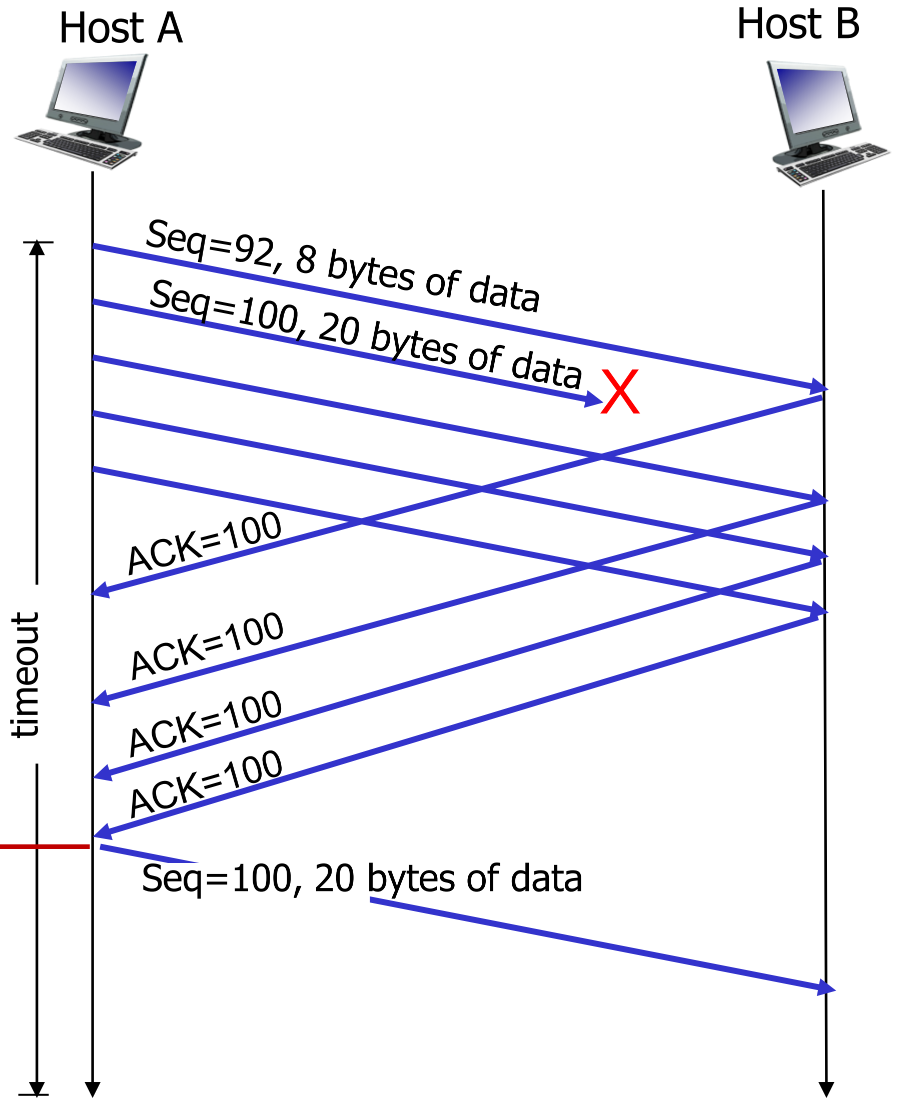
TCP fast retransmit

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!



Chapter 3: roadmap

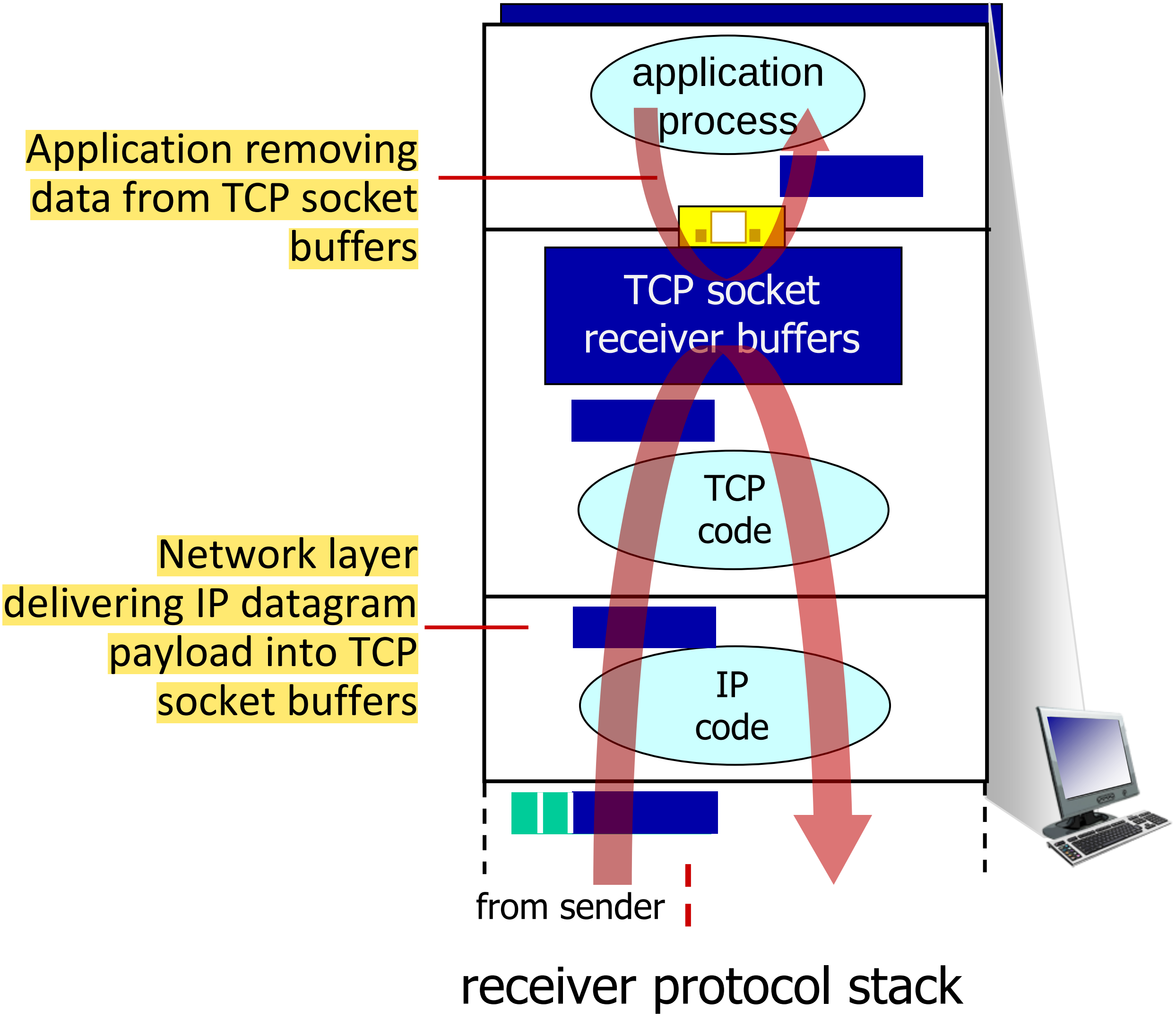
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management

Brief

- Principles of congestion control
- TCP congestion control

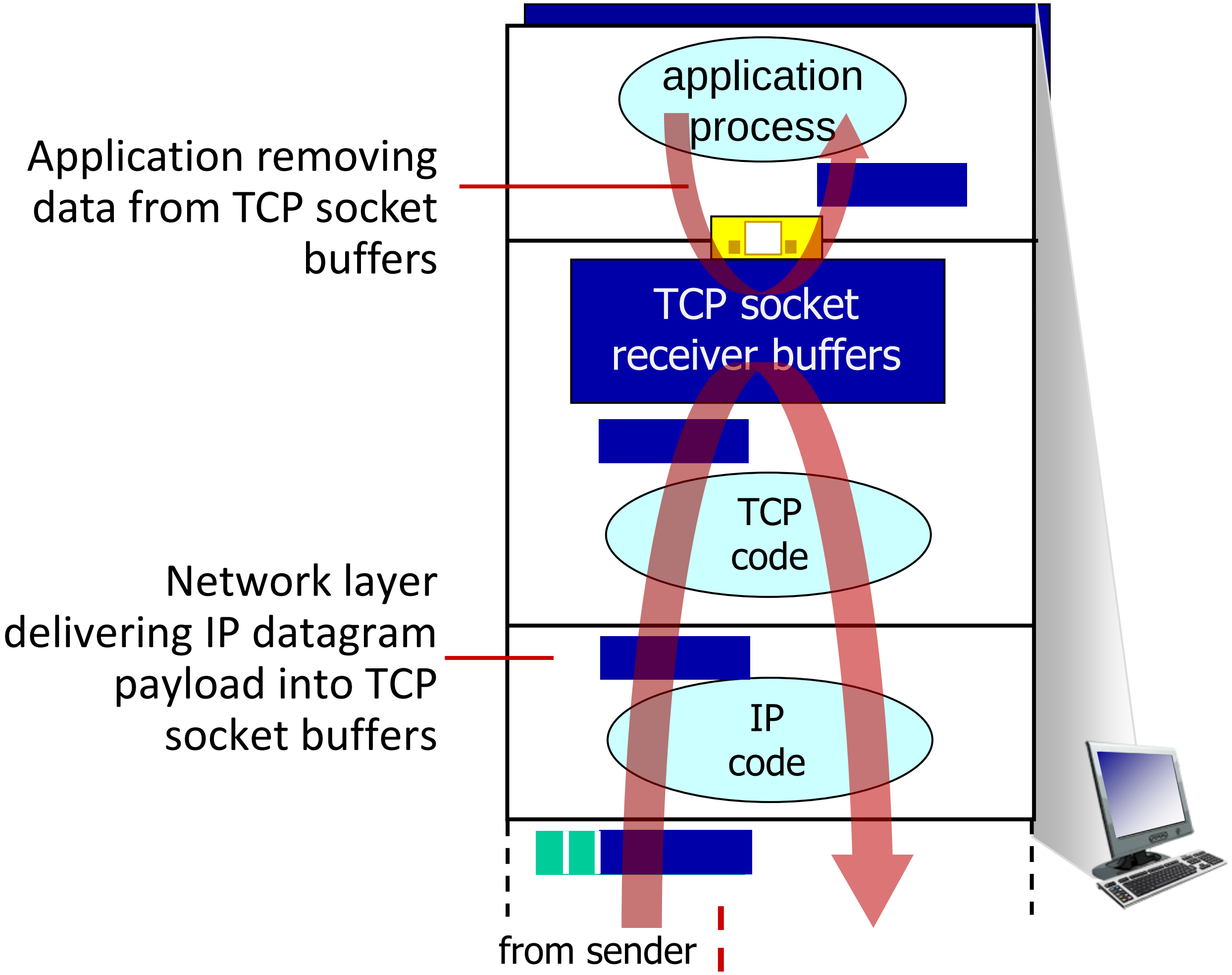
TCP flow control

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



TCP flow control

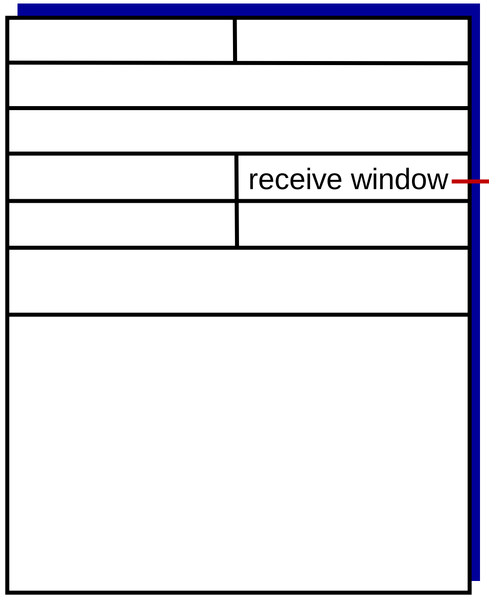
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

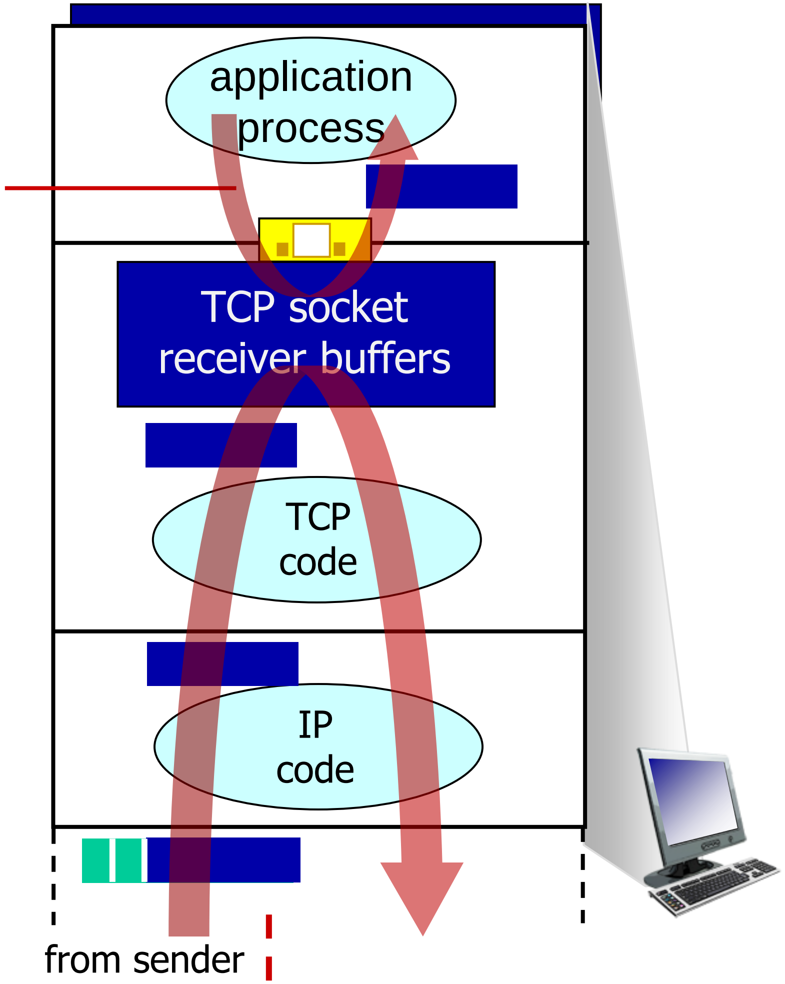
TCP flow control

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



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



receiver protocol stack

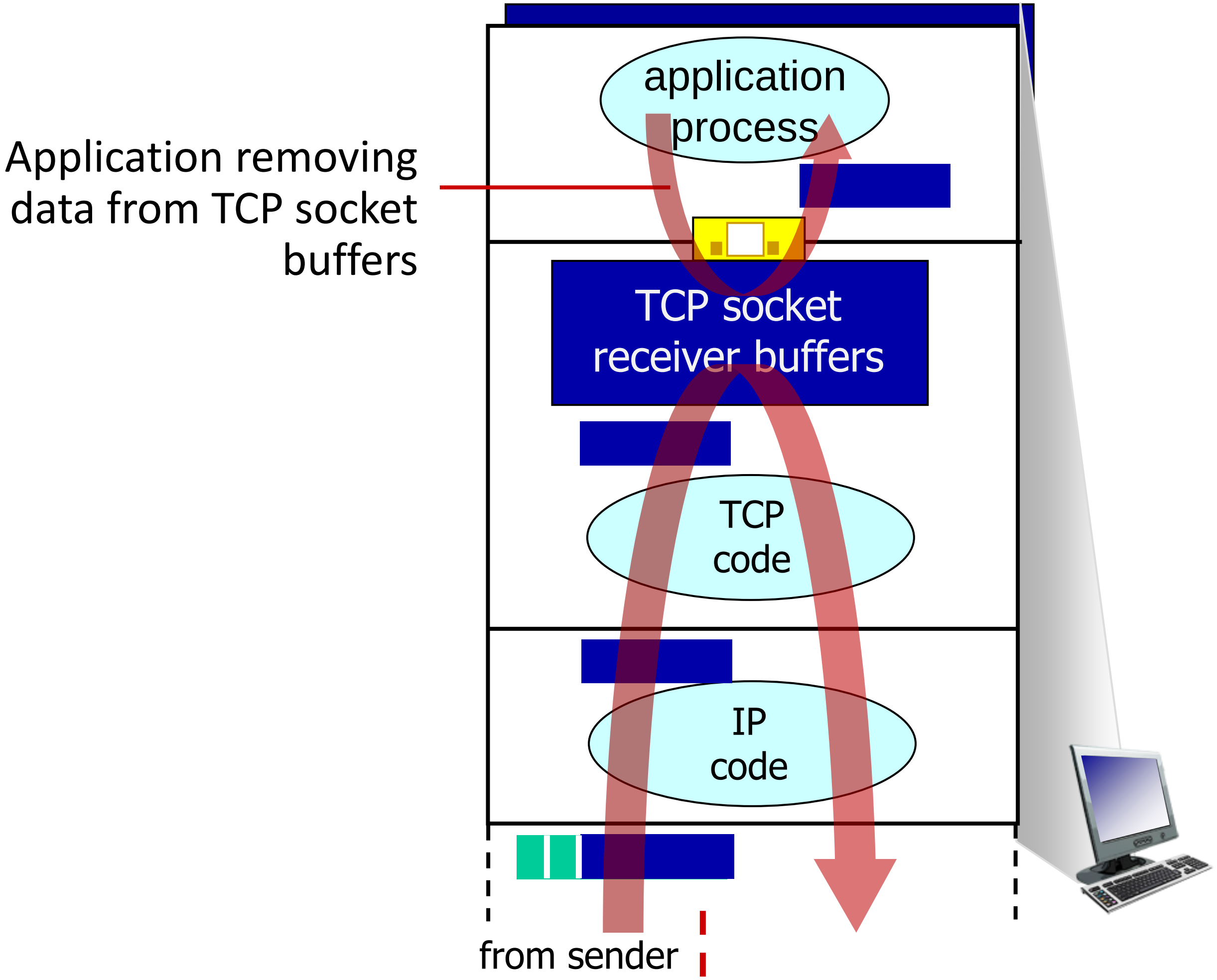
TCP flow control

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

A: If buffer fills up, new incoming data will be dropped

flow control

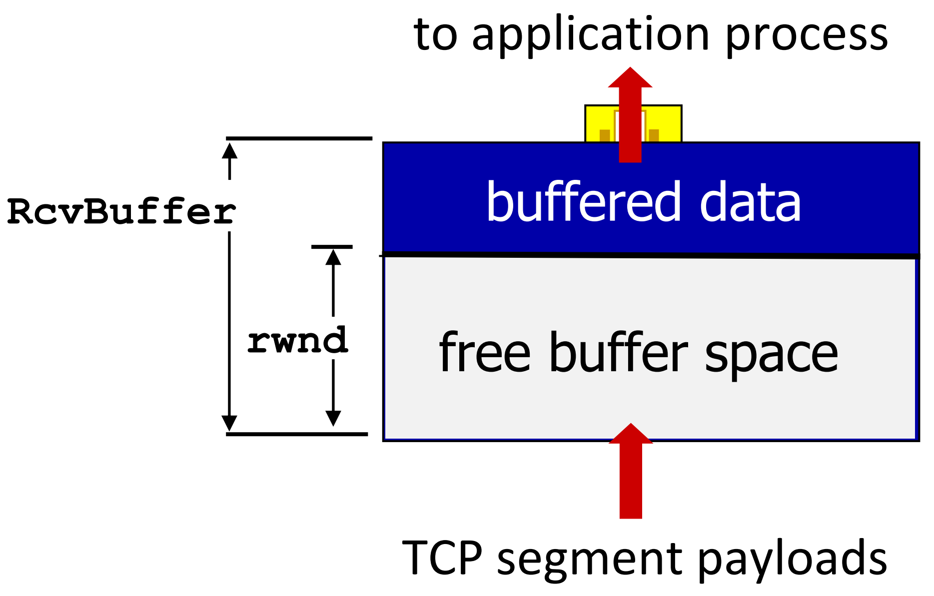
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



receiver protocol stack

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)
  - We saw this in socket programming !
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed ("in-flight") data to received **rwnd**
- guarantees receive buffer will not overflow



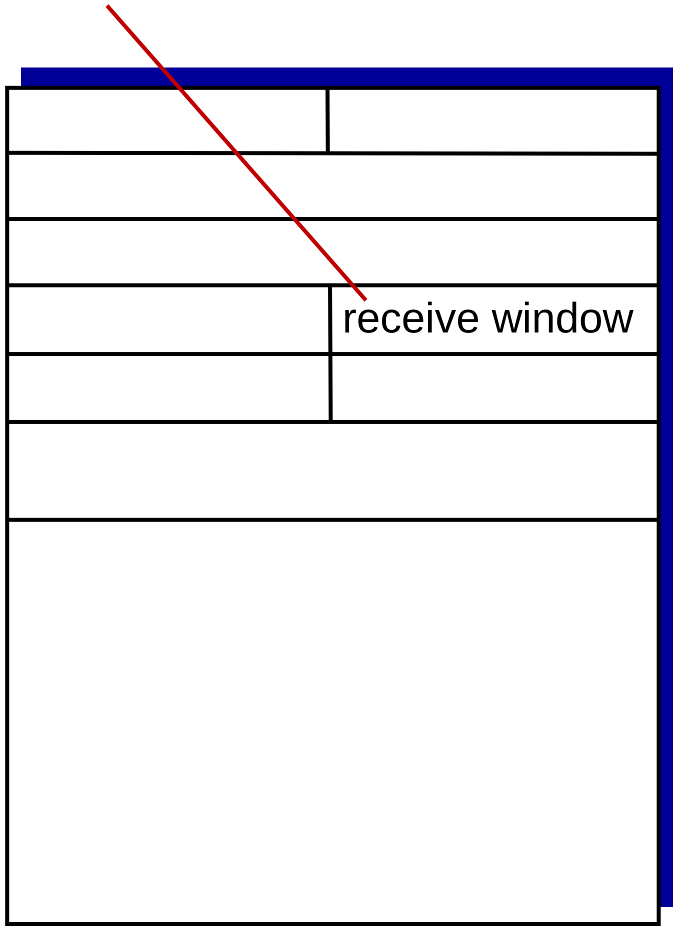
TCP receiver-side buffering

rwnd: receiver window

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)
  - We saw this in socket programming !
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow

flow control: # bytes receiver willing to accept



TCP segment format

TCP segment structure

SYN

ACK

FIN end

Flags

1. **ACK** indicates this is an ACK segment
2. **SYN** indicates this is segment for connection setup
3. **FIN** used for finishing/closing connection

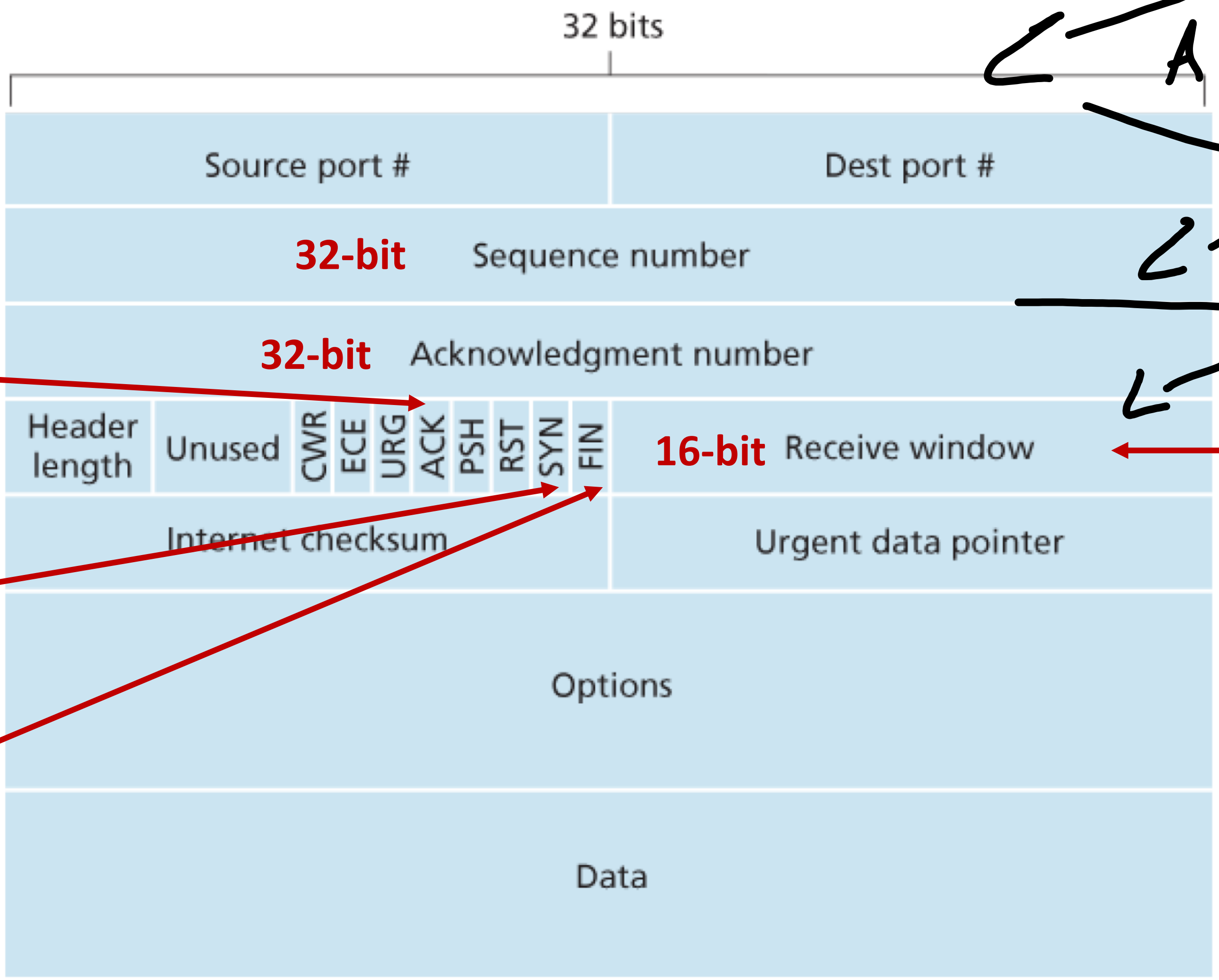


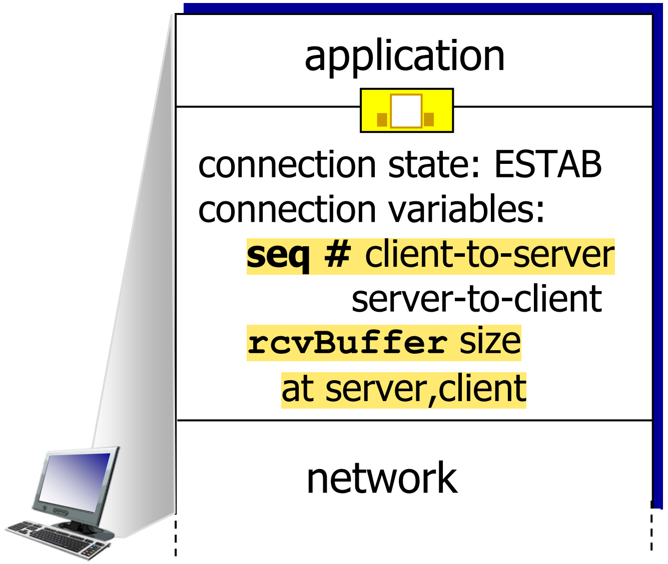
Figure 3.29 TCP segment structure

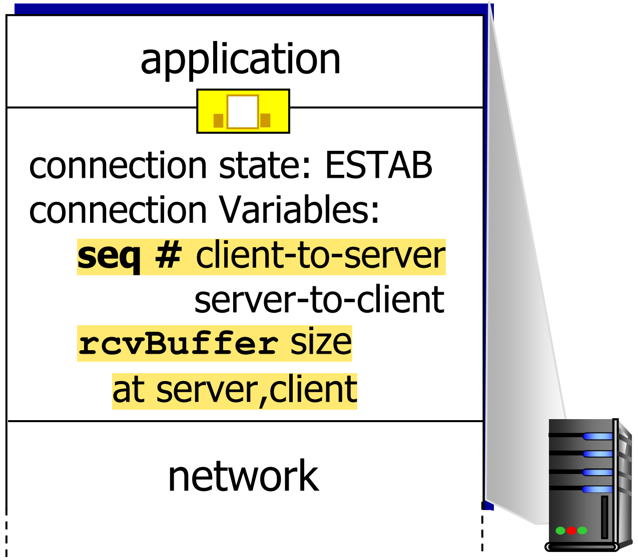
rwnd Needed for flow control

* Initial rwnd will change during connection

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();
```

TCP 3-way handshake

Server state

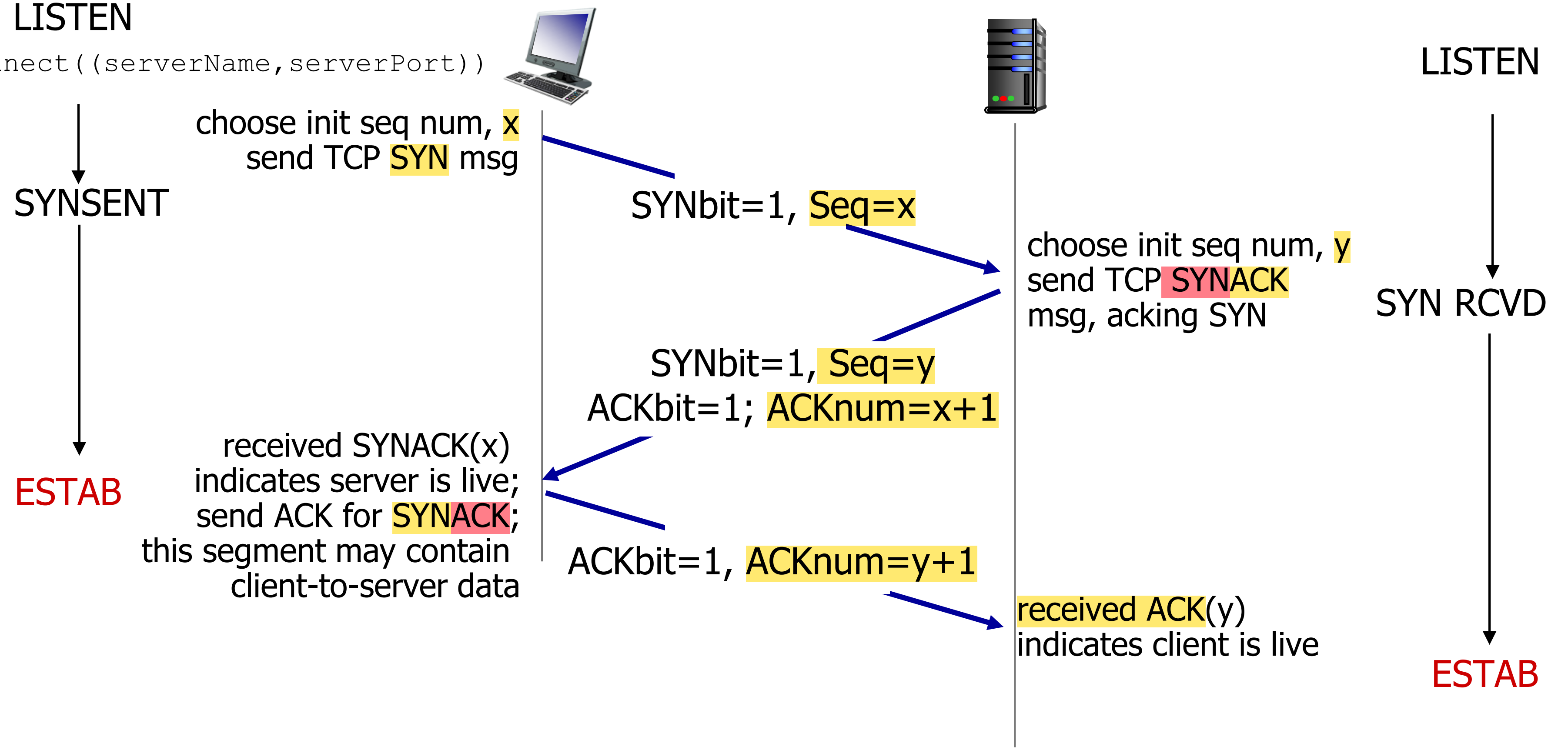
```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

Client state

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

LISTEN

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



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

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 **Brief +**
- 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

Principles of congestion control

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

- We will not cover the details in this course.

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:

- We will leave the network "edge"
  - application & transport layers
- Will dive into the network "core"
- two **network-layer** chapters:
  - data plane
  - control plane