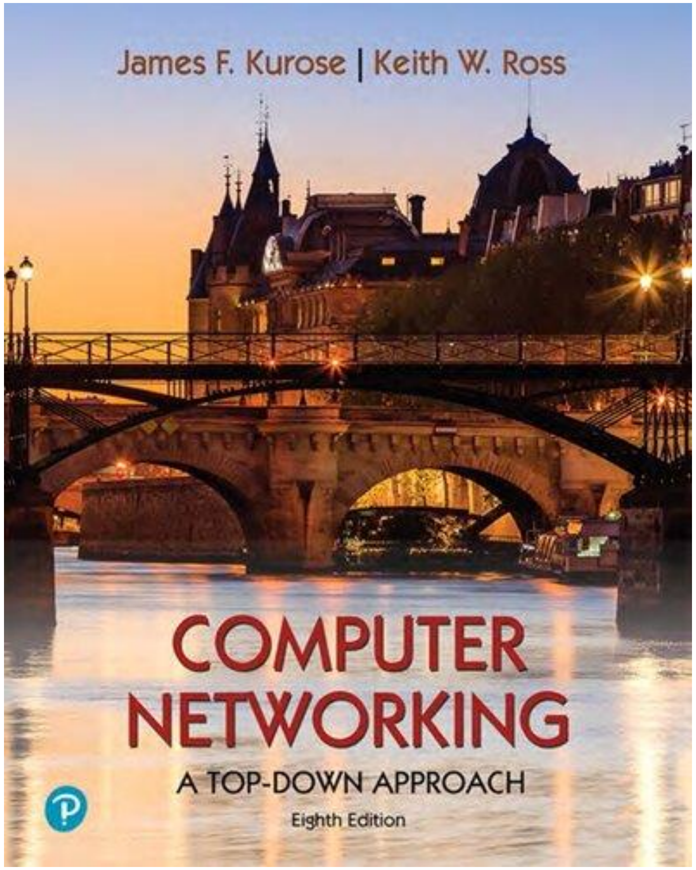
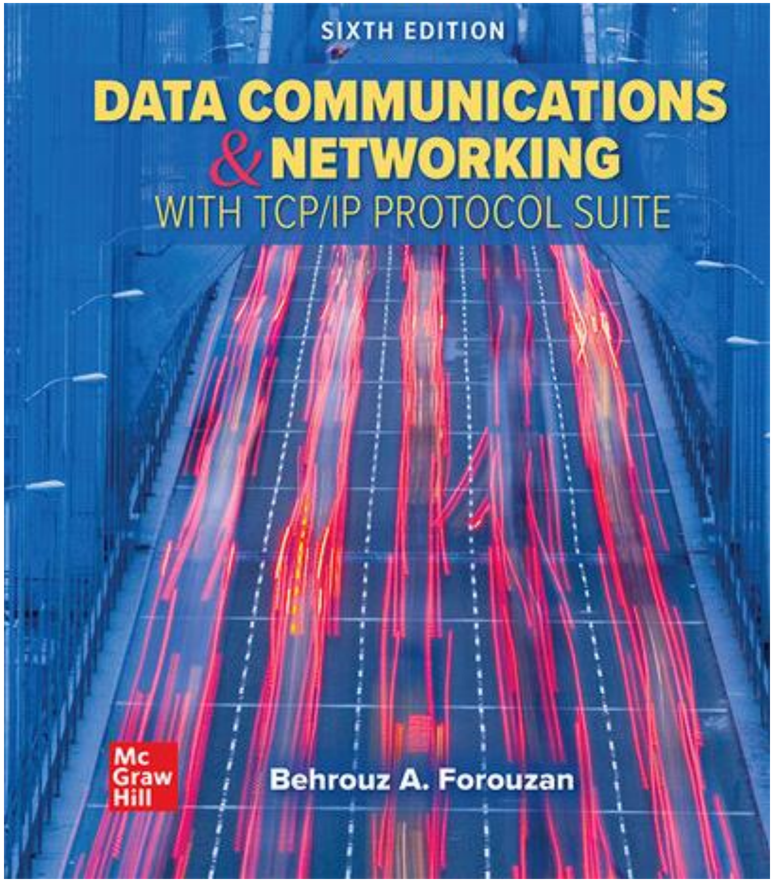
الجامعة اللبنانية
UNIVERSITE LIBANAISE

I3304
Network administration and security

Ahmad Fadlallah

Reference Textbooks





Outline

- Introduction
  - Introduction to the course
  - Recall Network Basics (I2208)
- Network Layer
  - Static Routing
  - Dynamic Routing
    - Dynamic Routing Algorithm
    - Dynamic Routing Protocols
  - NAT (Network Address Translation)
  - IPv6
- Transport Layer
  - Function of the transport layer
  - UDP Protocol
  - TCP Protocol
    - Connection management
    - Flow control
    - Congestion control
- Application Layer
  - HTTP protocol
  - FTP protocol
  - Mail protocols
  - DNS
- Introduction to Security
  - Security services
  - Cryptography
  - Digital Signature
  - Principle of network security protocols

References

- The slides are based on the:
  - Jim Kurose, Keith Ross Slides for the Computer Networking: A Top-Down Approach, 8th edition, Pearson, 2020

Learning Objectives

- Understand principles behind transport layer services:
  - Multiplexing, de-multiplexing
  - 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: Outline

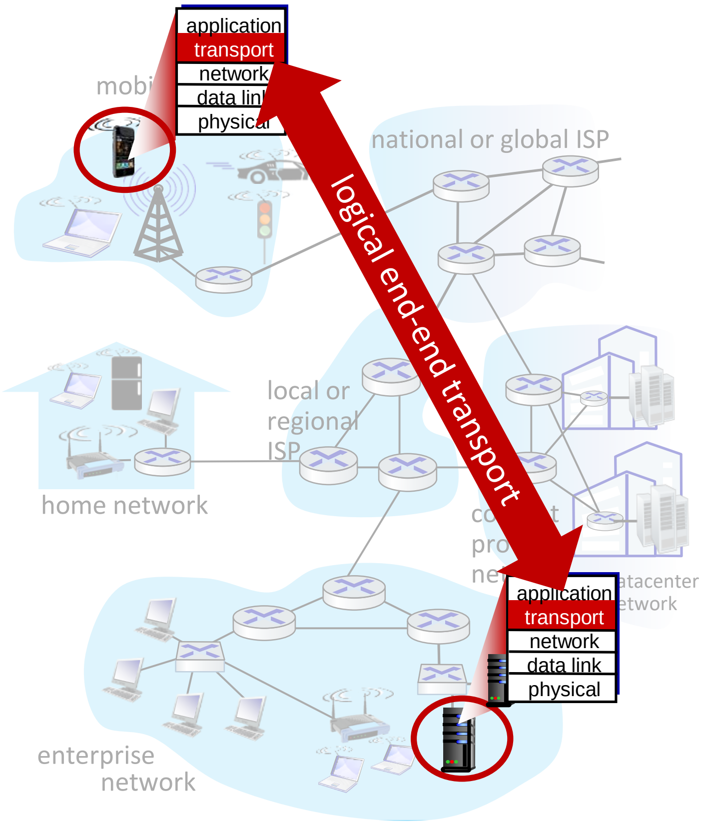
- Transport-layer services
- Multiplexing and De-Multiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- TCP congestion control
- Evolution of transport-layer functionality

Transport-layer services

Multiplexing and De-Multiplexing
Connectionless transport: UDP
Connection-oriented transport: TCP
TCP congestion control
Evolution of transport-layer functionality

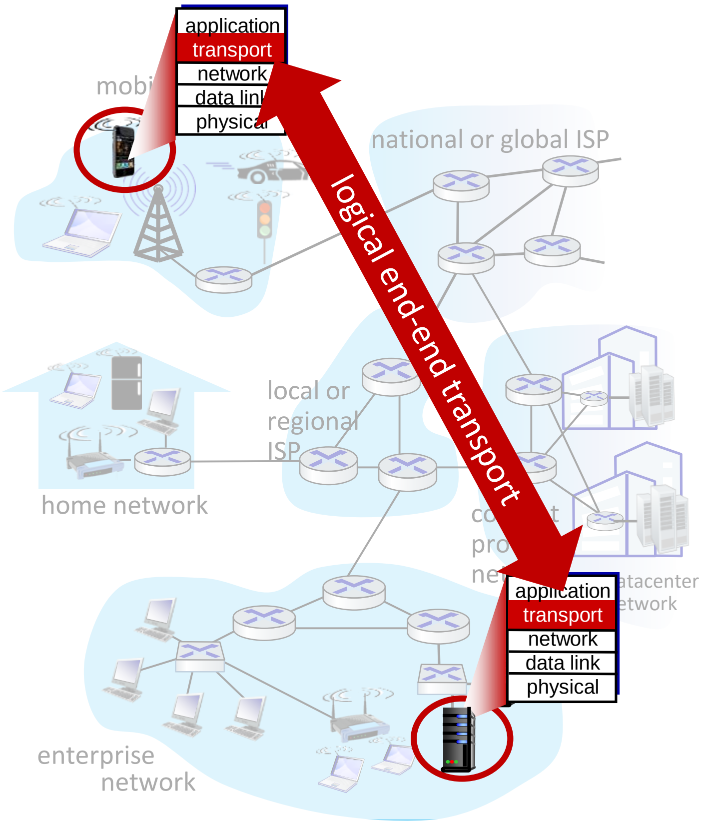
Transport services and protocols

- Provide **<u>logical</u> communication** between application processes running on different hosts
- Transport protocols actions in end systems:
  - ⊙Sender: <u>breaks</u> application messages into **segments**, passes to network layer
  - ⊙Receiver: <u>reassembles</u> segments into messages, passes to application layer
- Two transport protocols available to Internet applications
  - ⊙**TCP, UDP**



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



Transport-layer services

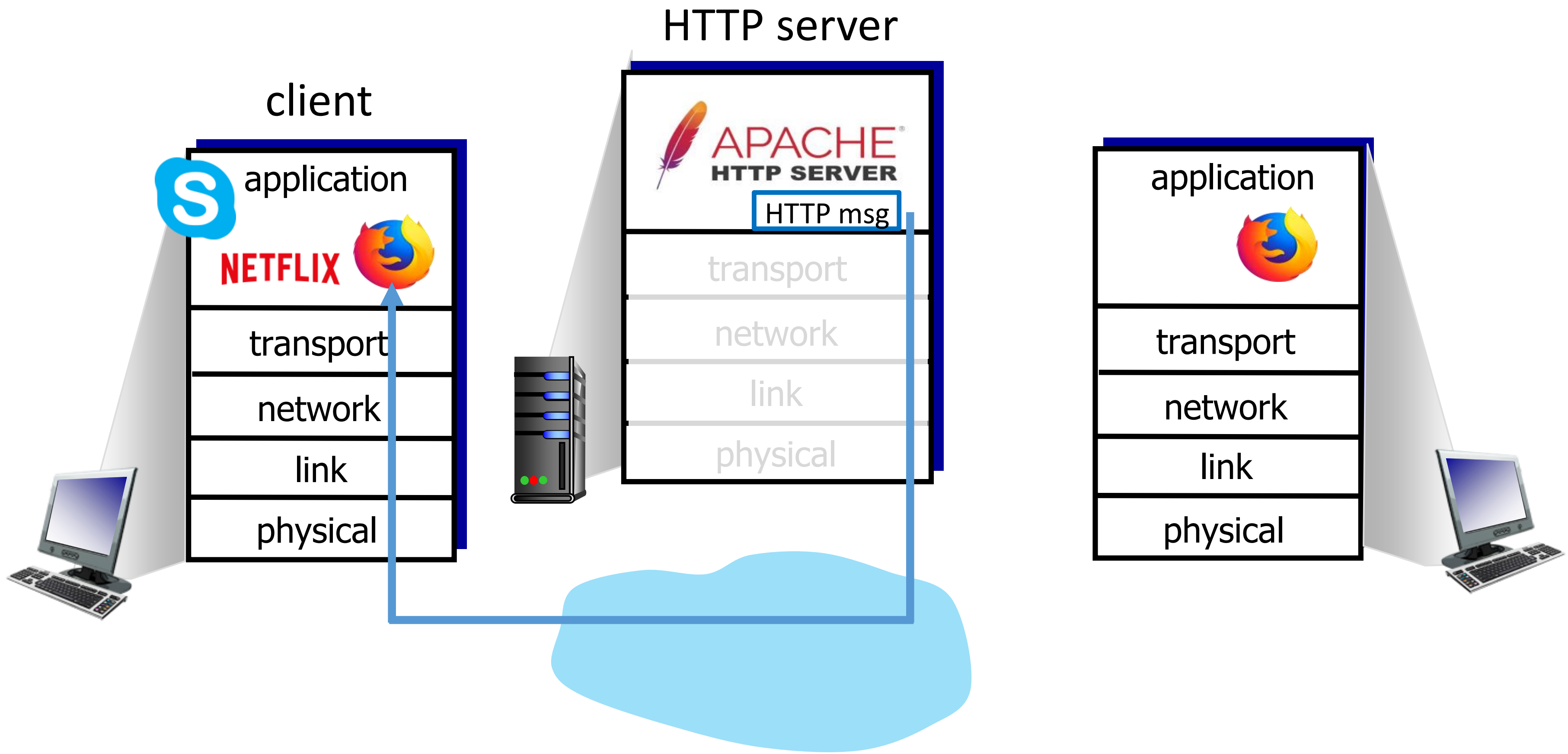
Multiplexing and De-Multiplexing

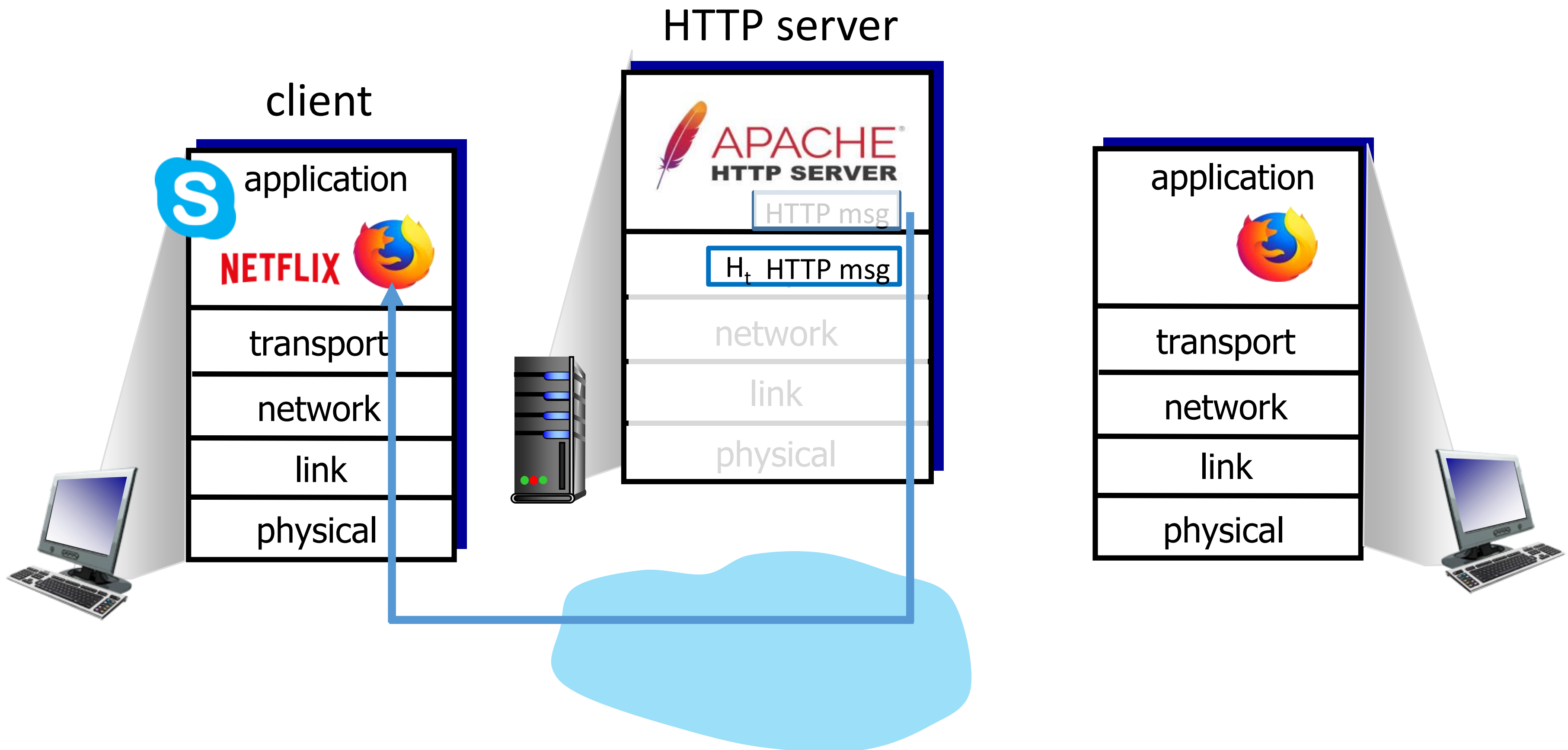
Connectionless transport: UDP

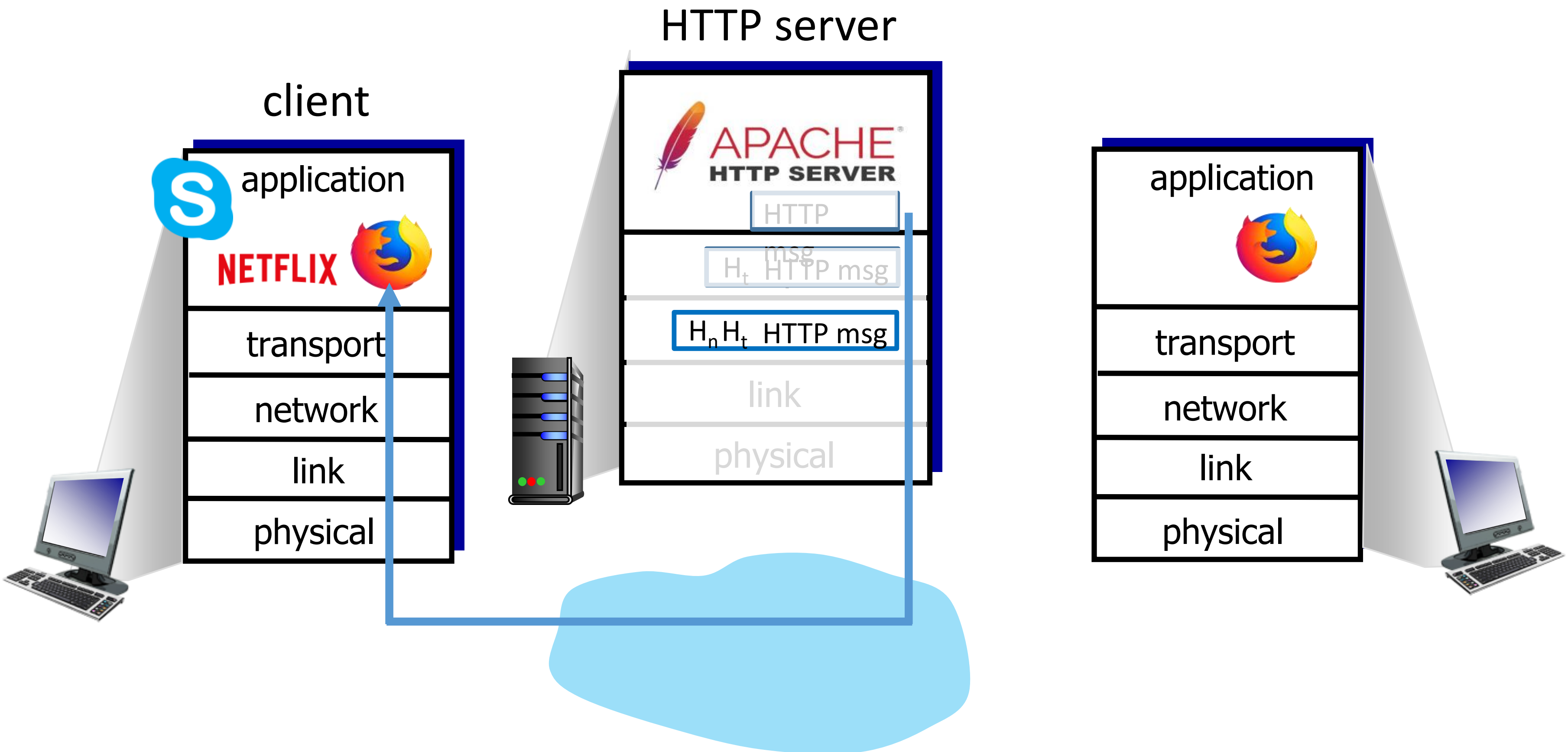
Connection-oriented transport: TCP

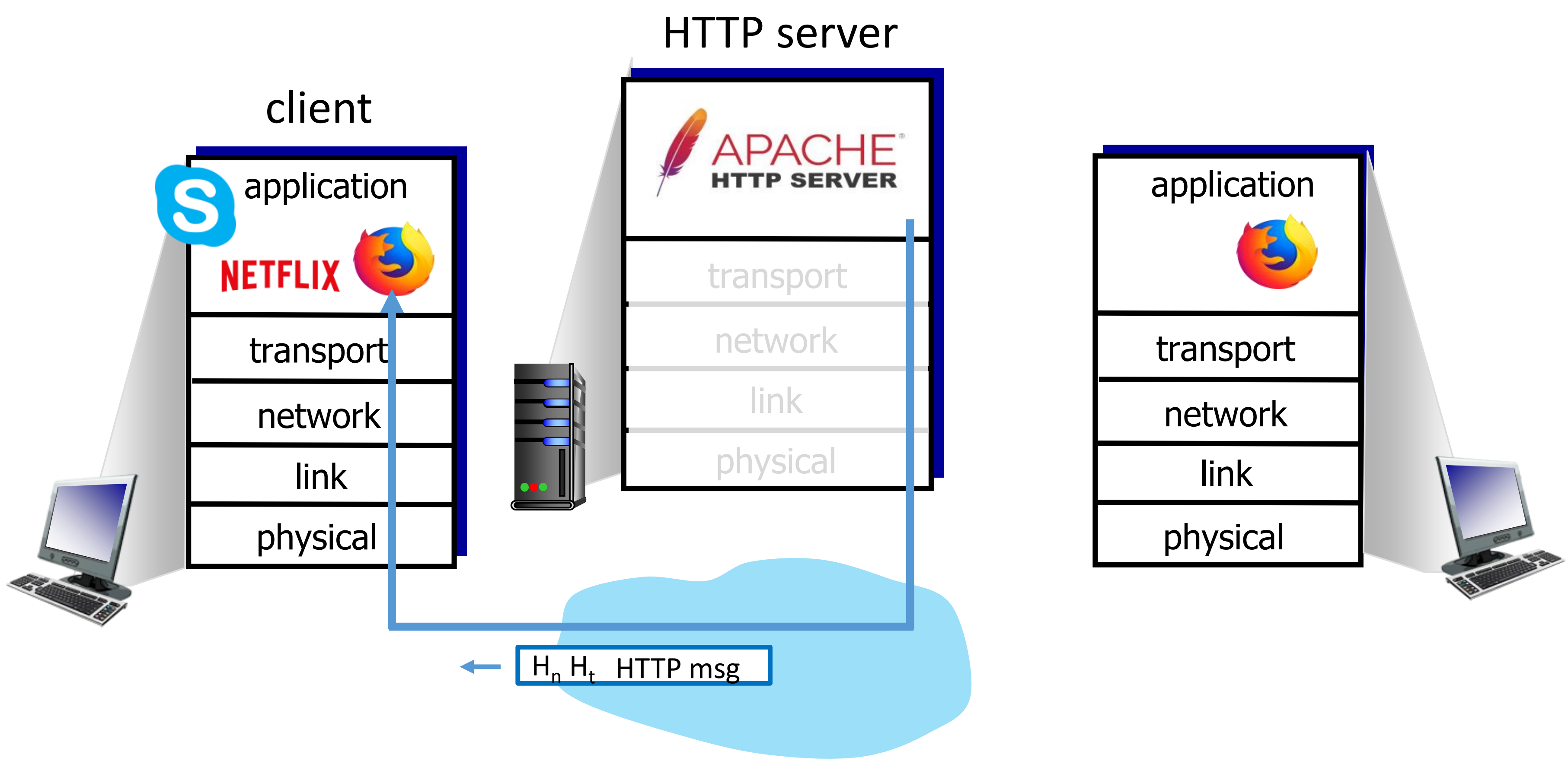
TCP congestion control

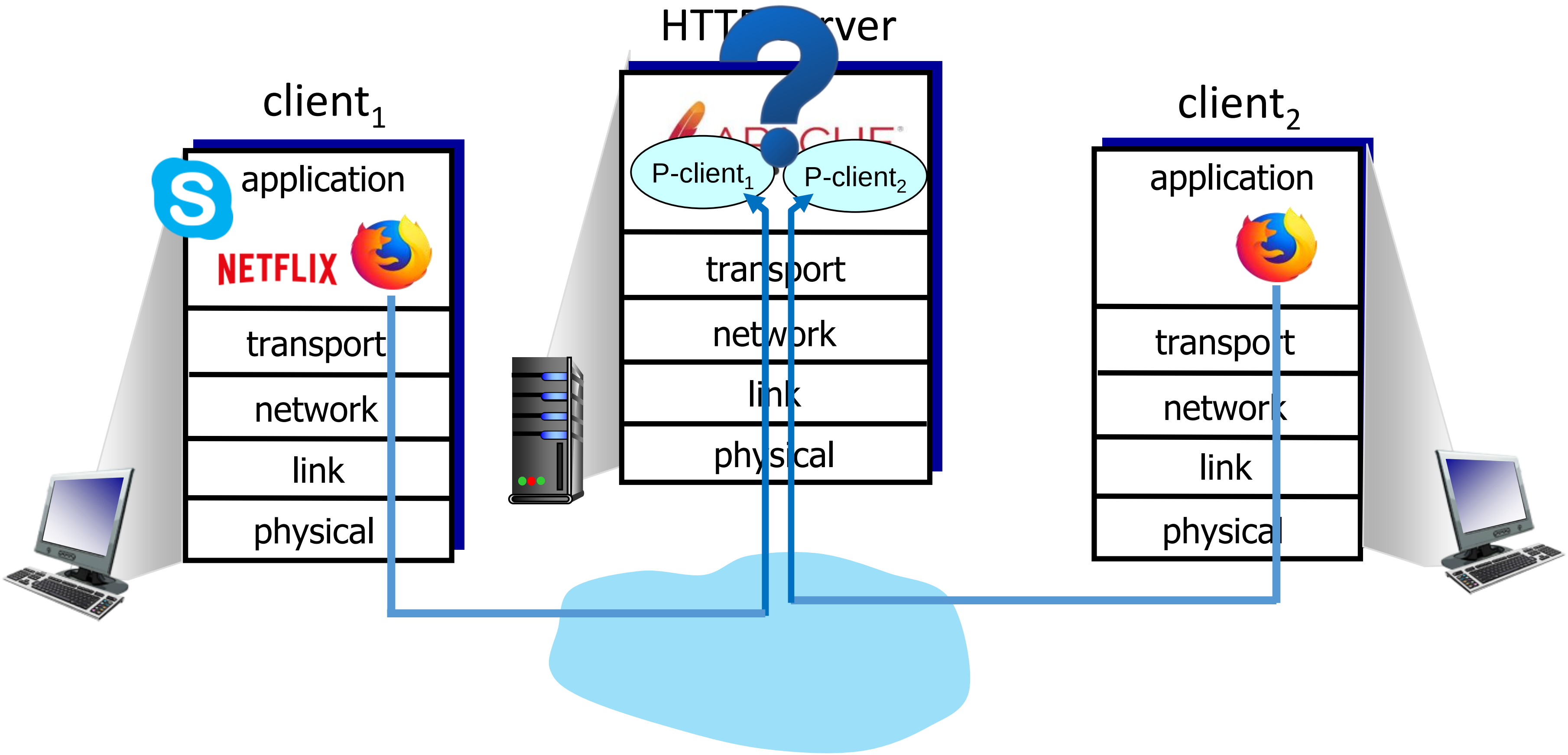
Evolution of transport-layer functionality


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


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



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

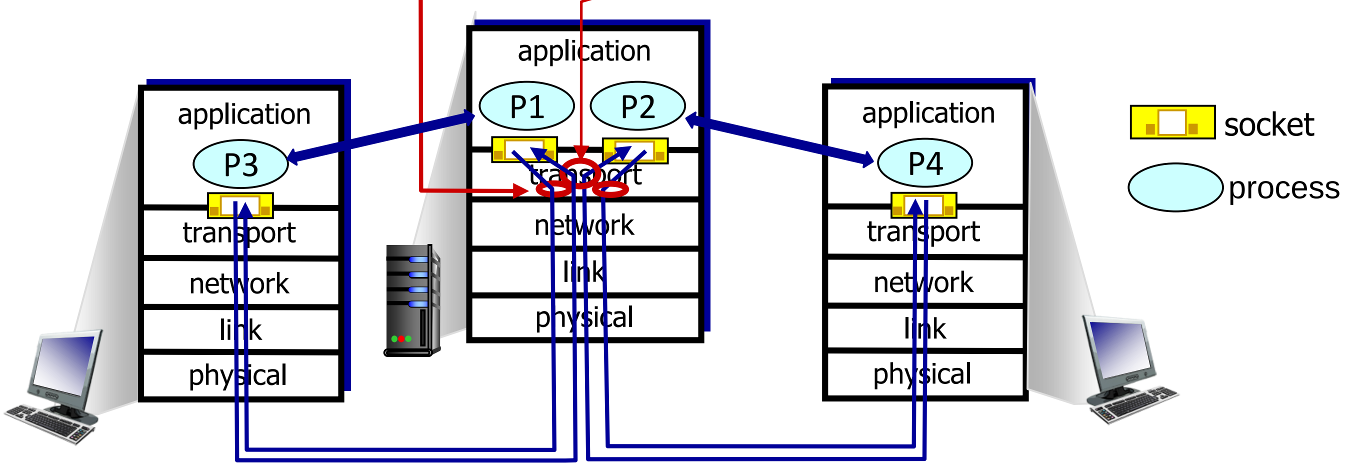
Multiplexing/demultiplexing

multiplexing at sender:

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

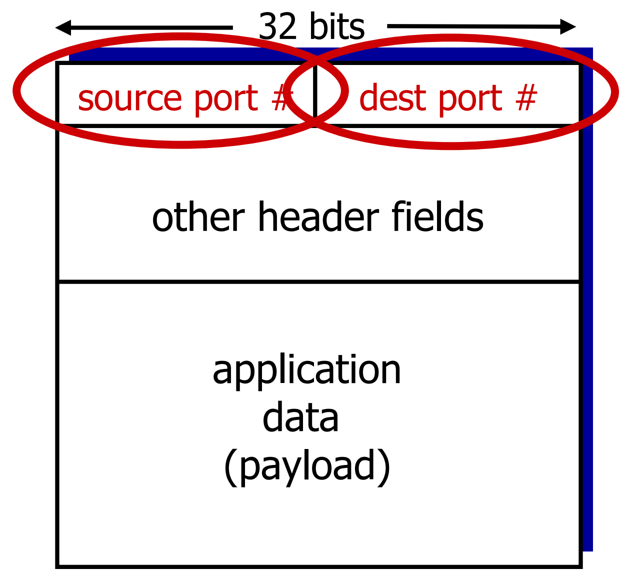
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

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



TCP/UDP segment format

Connectionless demultiplexing

Recall:

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

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

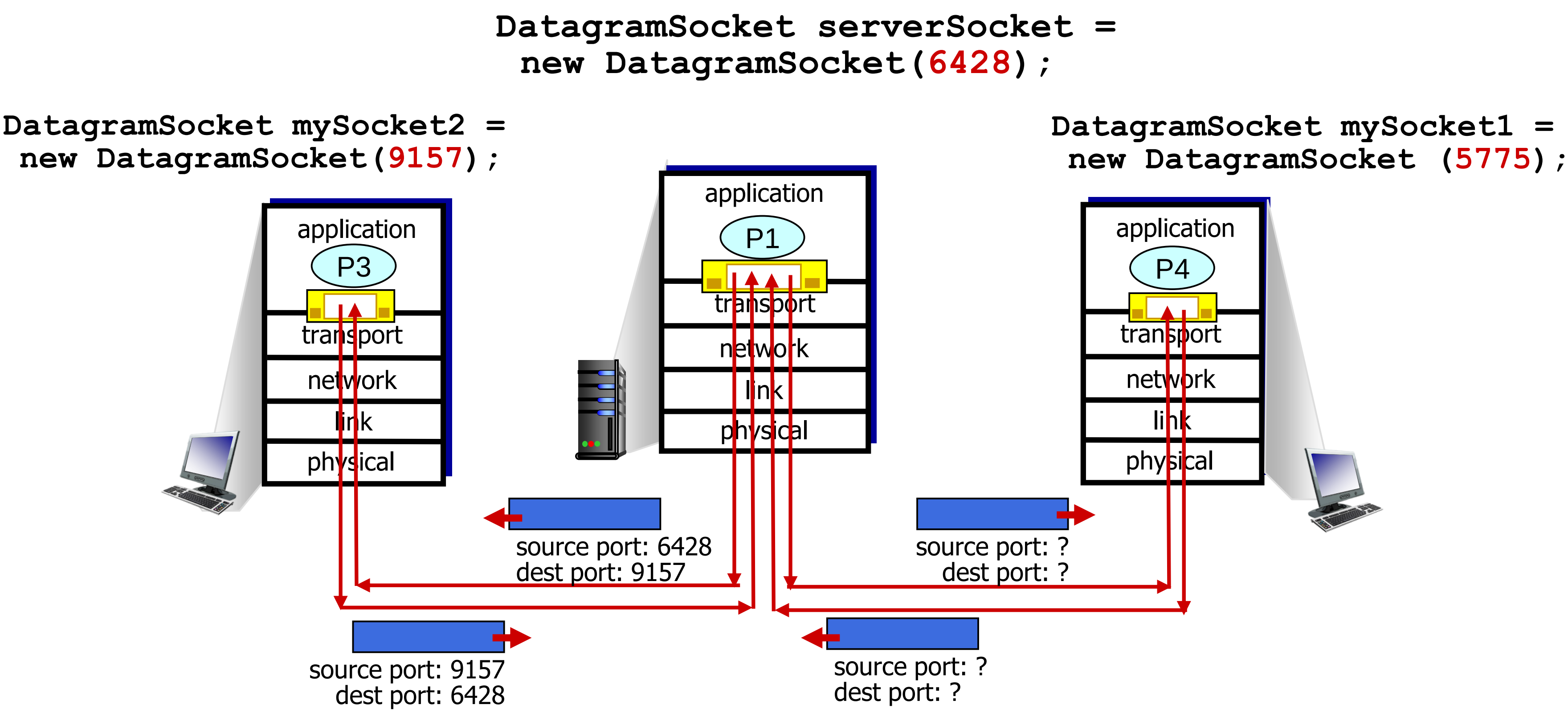
- when creating datagram to send into UDP socket, must specify
  - Destination IP address
  - Destination port #

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

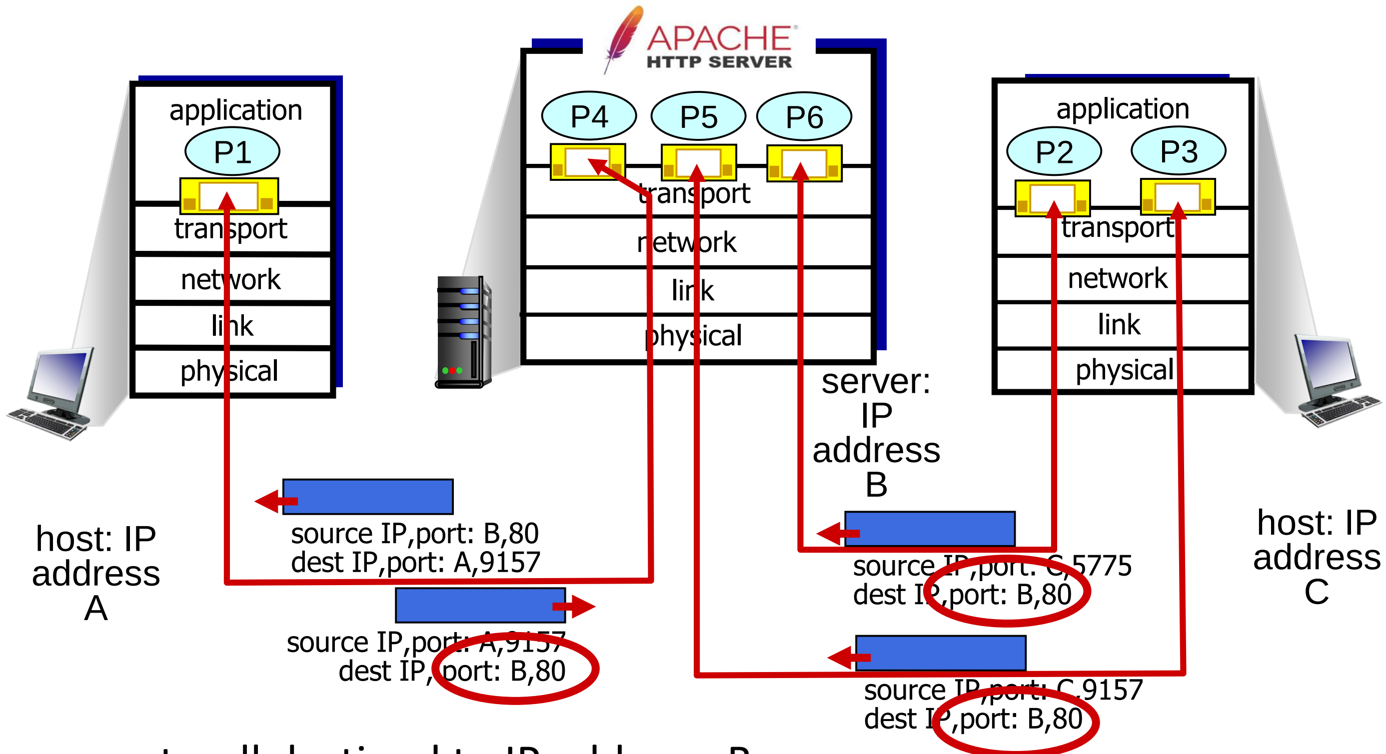


Connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination 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,
destination 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

Transport-layer services

Multiplexing and De-Multiplexing

Connectionless transport: UDP

Connection-oriented transport: TCP

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

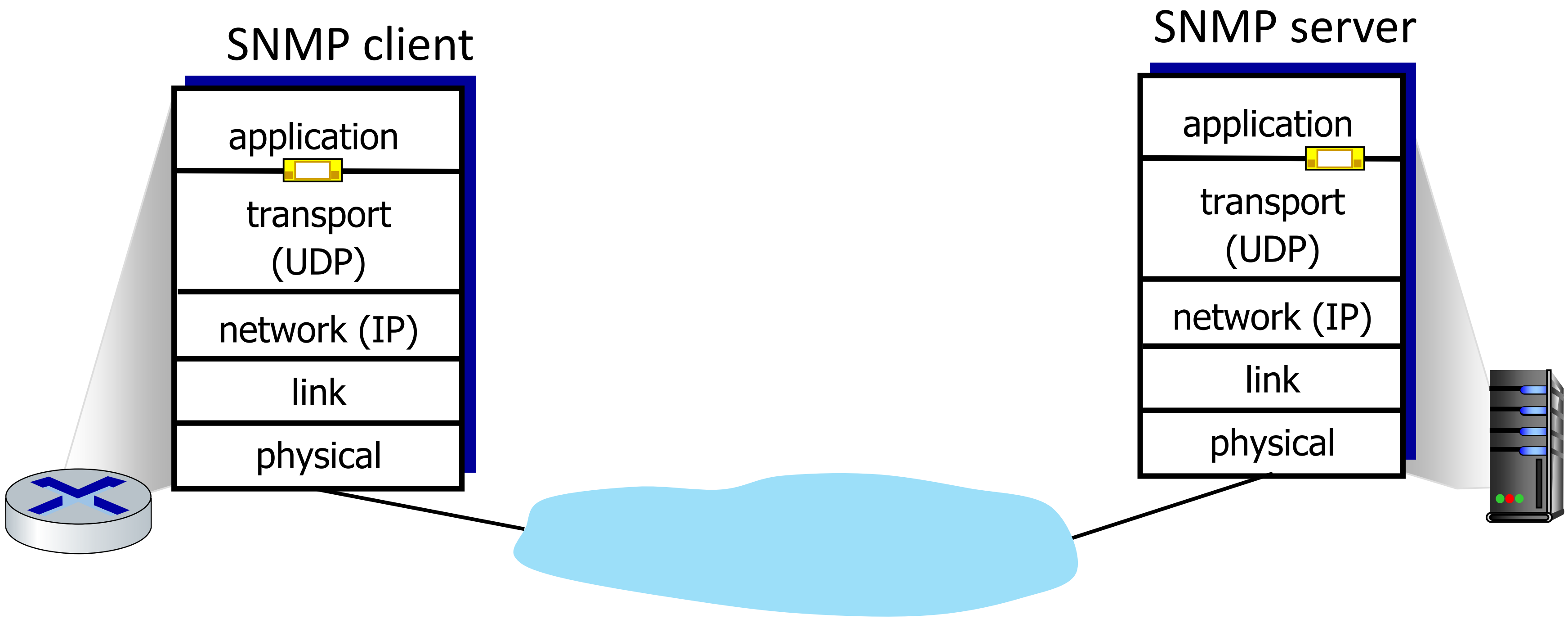
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

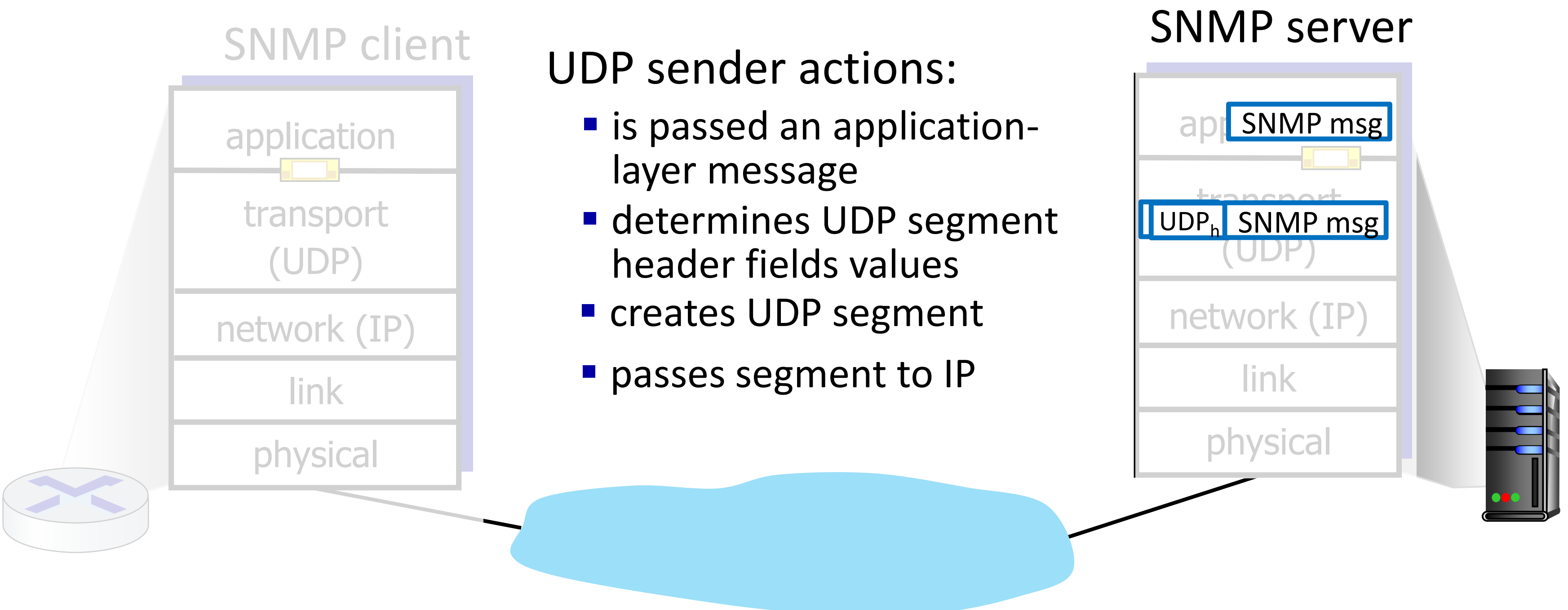
Format

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

UDP: Transport Layer Actions



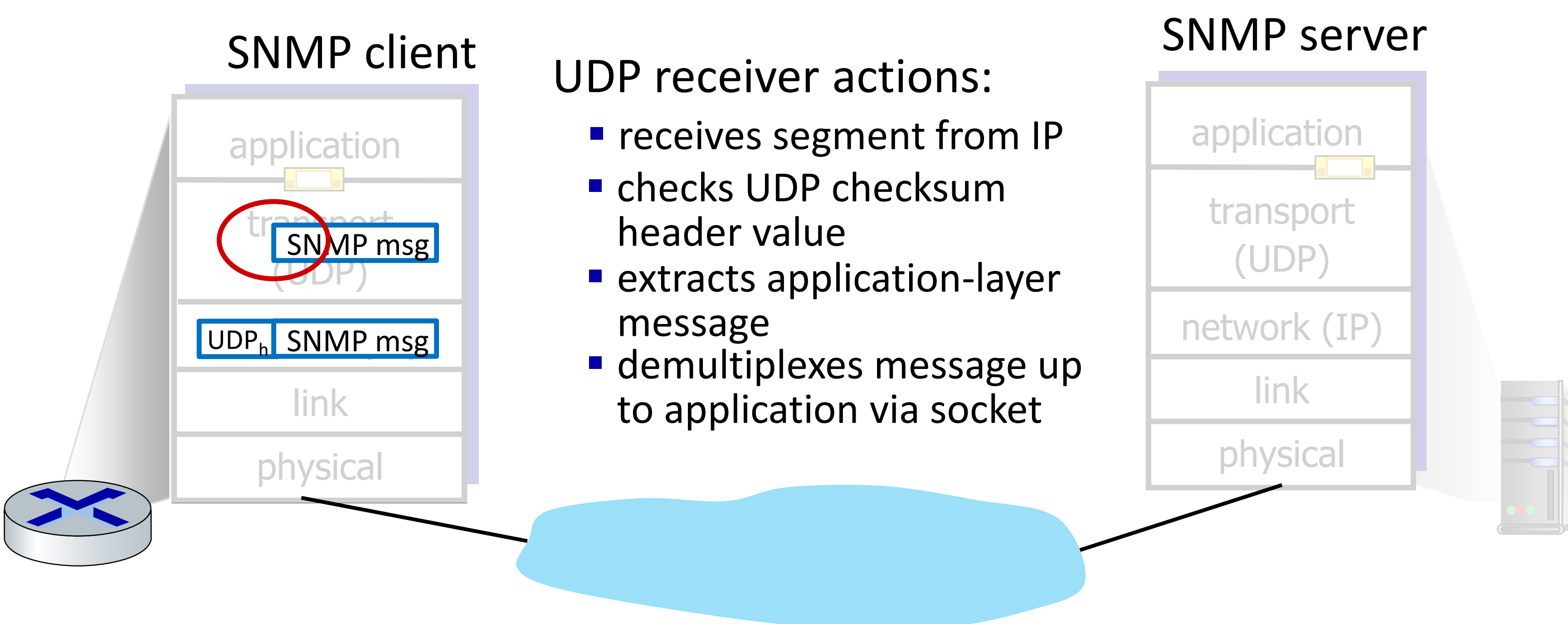
UDP: Transport Layer Actions



UDP sender actions:

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

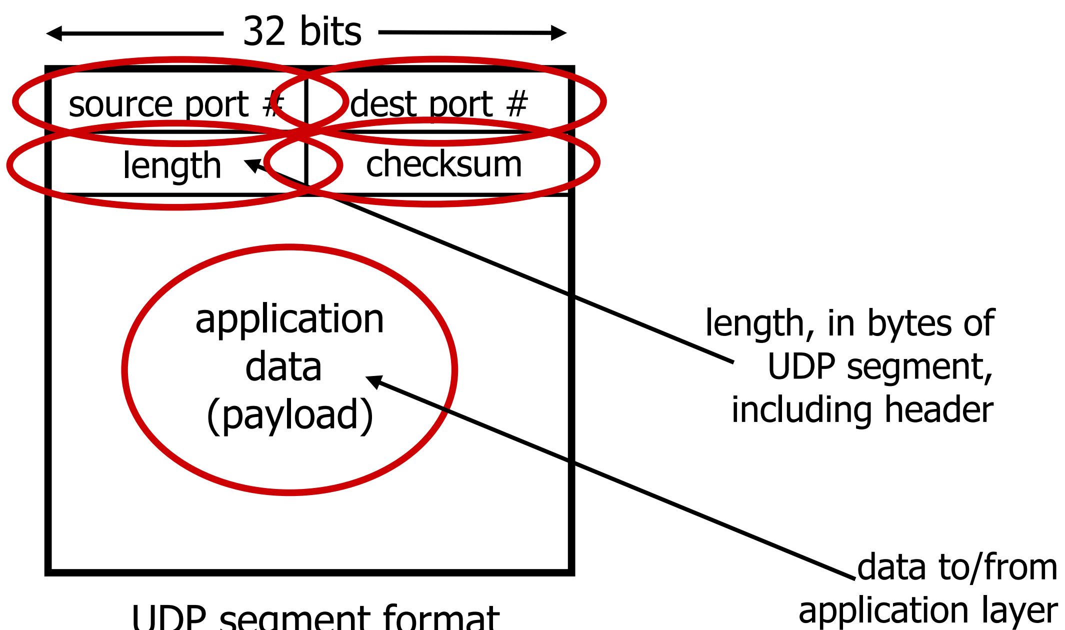
UDP: Transport Layer Actions



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 ≠ 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
		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!

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)**

Transport-layer services

Multiplexing and De-Multiplexing

Connectionless transport: UDP

Principles of reliable data transfer

Connection-oriented transport: TCP

TCP congestion control

Evolution of transport-layer functionality

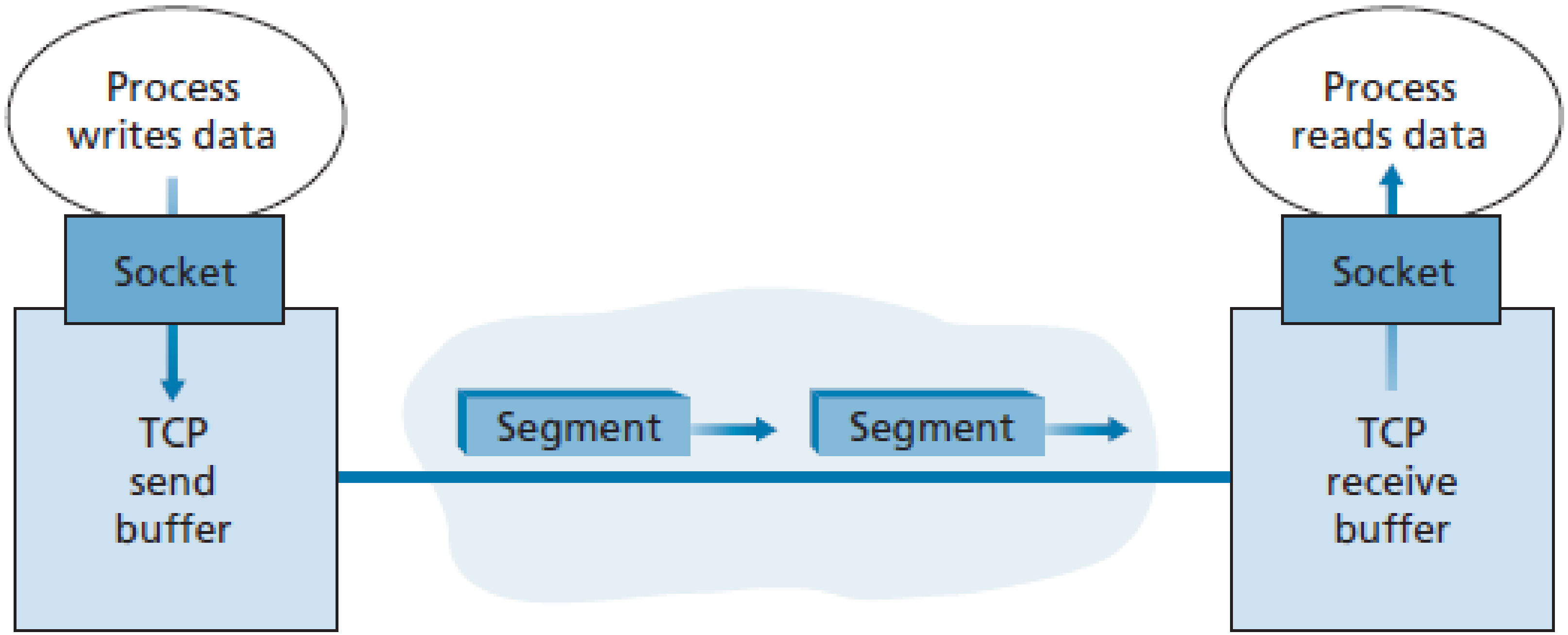
TCP: overview (RFCs: 793,1122, 2018, 5681, 7323)

- Point-to-Point:
  - one sender, one receiver
- Reliable, In-order *byte stream:*
  - 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

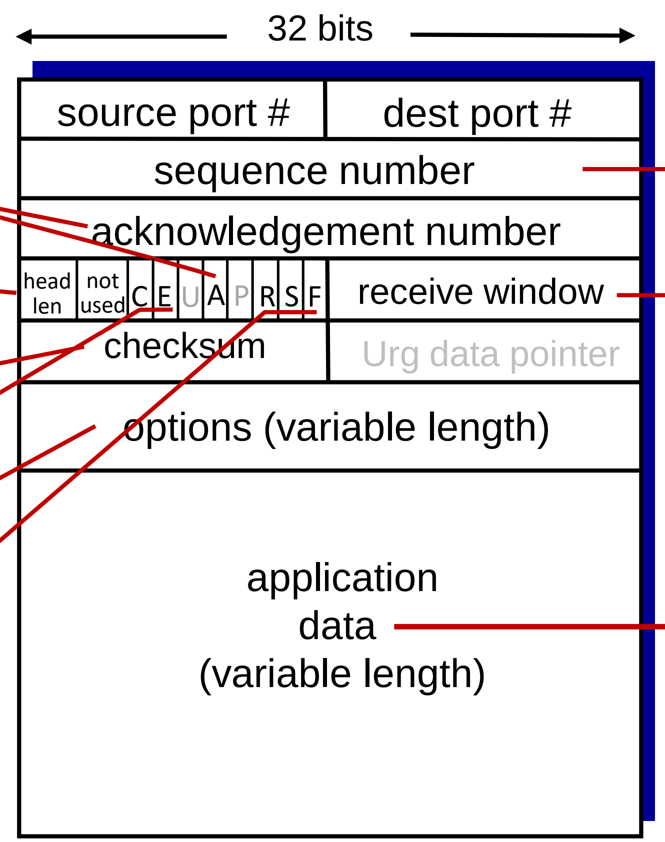
TCP: overview

- Defined in RFCs: 793,1122, 2018, 5681, 7323
- **Point-to-Point:** one sender, one receiver (no multicast)
- **Connection-oriented:** handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- **Full duplex data:** Bi-directional data flow in same connection
- **Reliable, In-order byte stream:** no “message boundaries“, Maximum Segment Size
- **Pipelining:** Sender can have multiple transmitted but yet-to-be acknowledged segments outstanding at any given time
- **Cumulative ACKs**
- **Flow controlled:** sender will not overwhelm receiver

TCP: Overview



TCP segment structure



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

length (of TCP header) in 4-bytes words

Internet checksum (as in UDP)

C, E: congestion notification

TCP options (RFC 854 & 1323)

RST, SYN, FIN: connection management

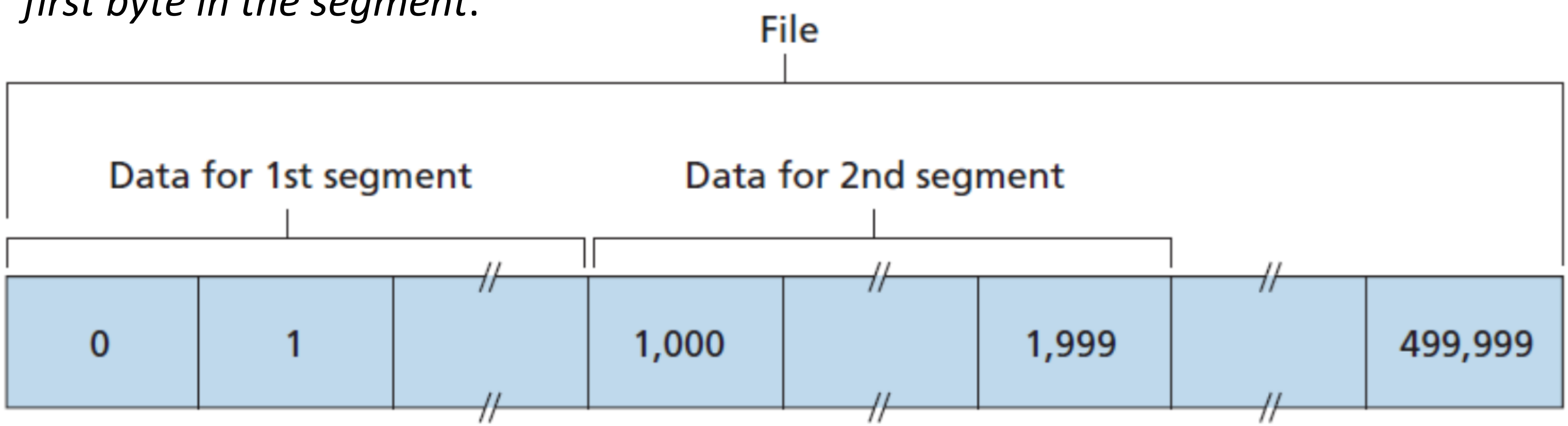
segment seq #: counting bytes of data into byte-stream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP Sequence Numbers & ACKs

- TCP views data as an unstructured, but ordered, stream of bytes
- The **sequence number** for a segment is therefore the *byte-stream number of the first byte in the segment*.



- Both sides of a TCP connection randomly choose an initial sequence number.
  - to minimize the possibility that a segment that is still present in the network from an earlier, already-terminated connection between two hosts is mistaken for a valid segment in a later connection between these same two hosts

TCP Sequence Numbers & ACKs

Sequence numbers:

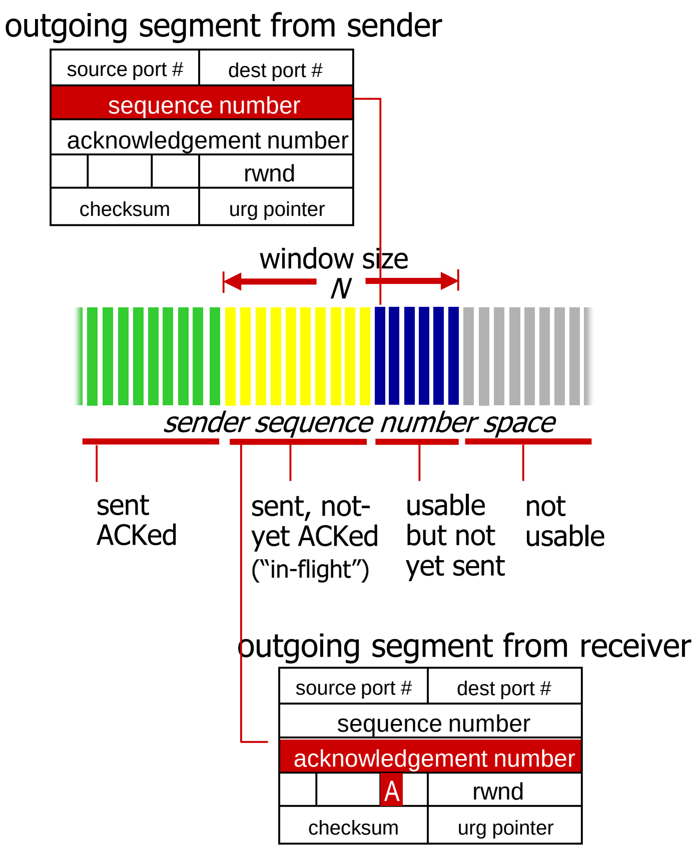
- byte stream "number" of first byte in segment's data

Acknowledgements:

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

Q: how receiver handles out-of-order segments

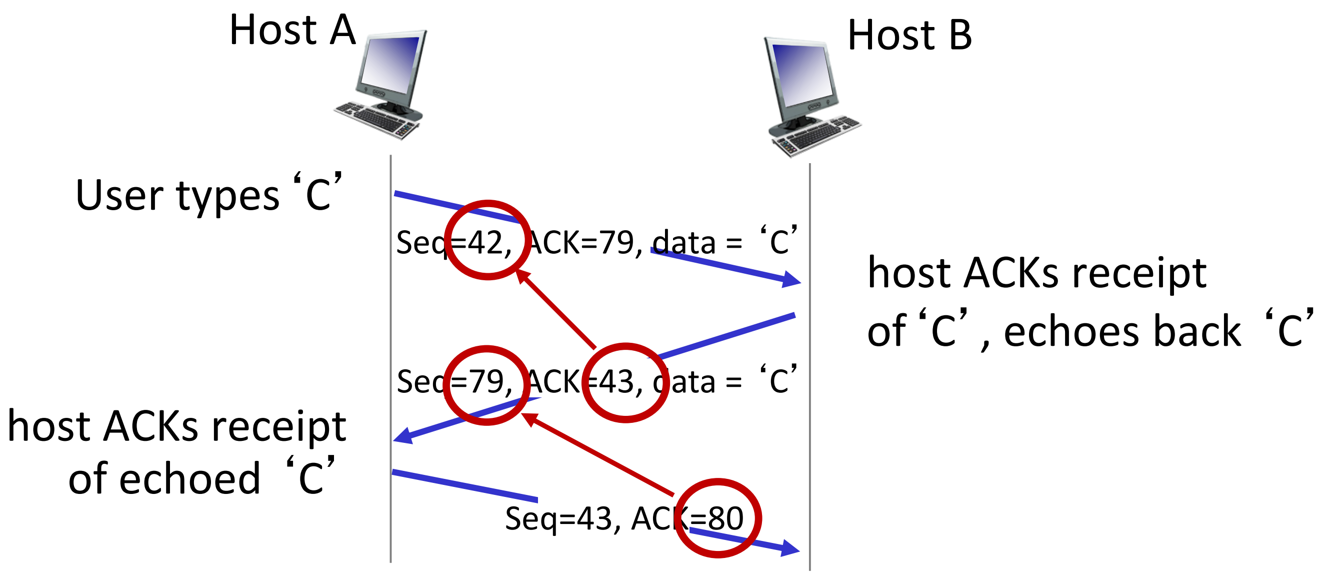
- *A:* TCP spec doesn't say, - up to implementor



TCP Sequence Numbers & ACKs

- **Example 1**
  - ⦿A has received all bytes numbered 0 through 535 from B
  - ⦿A is about to send a segment to B.
  - ⦿A is waiting for byte 536 and all the subsequent bytes in B’s data stream.
  - ⦿A puts 536 in the ACK number field of the segment it sends to B.
- **Example 2**
  - ⦿A has received one segment from B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000.
  - ⦿A has not yet received bytes 536 through 899.
  - ⦿A is still waiting for byte 536 (and beyond) in order to re-create B’s data stream.
  - ⦿A’s next segment to B will contain 536 in the acknowledgment number field.

TCP Sequence Numbers & ACKs



Simple Telnet Scenario

- The acknowledgment for client-to-server data is carried in a segment carrying server-to-client data
- This ACK is said to be **piggybacked** on the server-to-client data segment.

Round-Trip Time Estimation and Timeout

- **Why?**
  - ⊙ Uses a timeout/retransmit mechanism to recover from lost segments.
- **When to retransmit? What is the timeout value?**
  - ⊙ The timeout should be larger than the connection's Round-Trip Time (RTT)
    - The time from when a segment is sent until it is acknowledged.
    - Otherwise, unnecessary retransmissions would be sent.
  - ⊙ How much larger?
    - Too short: premature timeout, unnecessary retransmissions
    - Too long: slow reaction to segment loss
  - ⊙ Solution: RTT "Measurement-based" Estimation
    - Should a timer be associated with each and every unacknowledged segment?
    - Variable RTT depending on network conditions!
    - How to estimate RTT?

Timeout interval: Estimating the RTT

- Take sample packets for measuring the RTT
  - Denoted `SampleRTT`
- How often?
  - Most TCP implementations take only one `SampleRTT` measurement at a time.
    - At any point in time, the `SampleRTT` is being estimated for only one of the transmitted but currently unacknowledged segments
    - A new value of `SampleRTT` approximately once every RTT.
  - TCP never computes a `SampleRTT` for a segment that has been retransmitted
    - Only measures `SampleRTT` for segments that have been transmitted once
- How to deal with fluctuating `SampleRTT` ?
  - Want estimated RTT “smoother”
  - **Average several recent measurements, not just current `SampleRTT`**

Timeout interval: Estimating the RTT

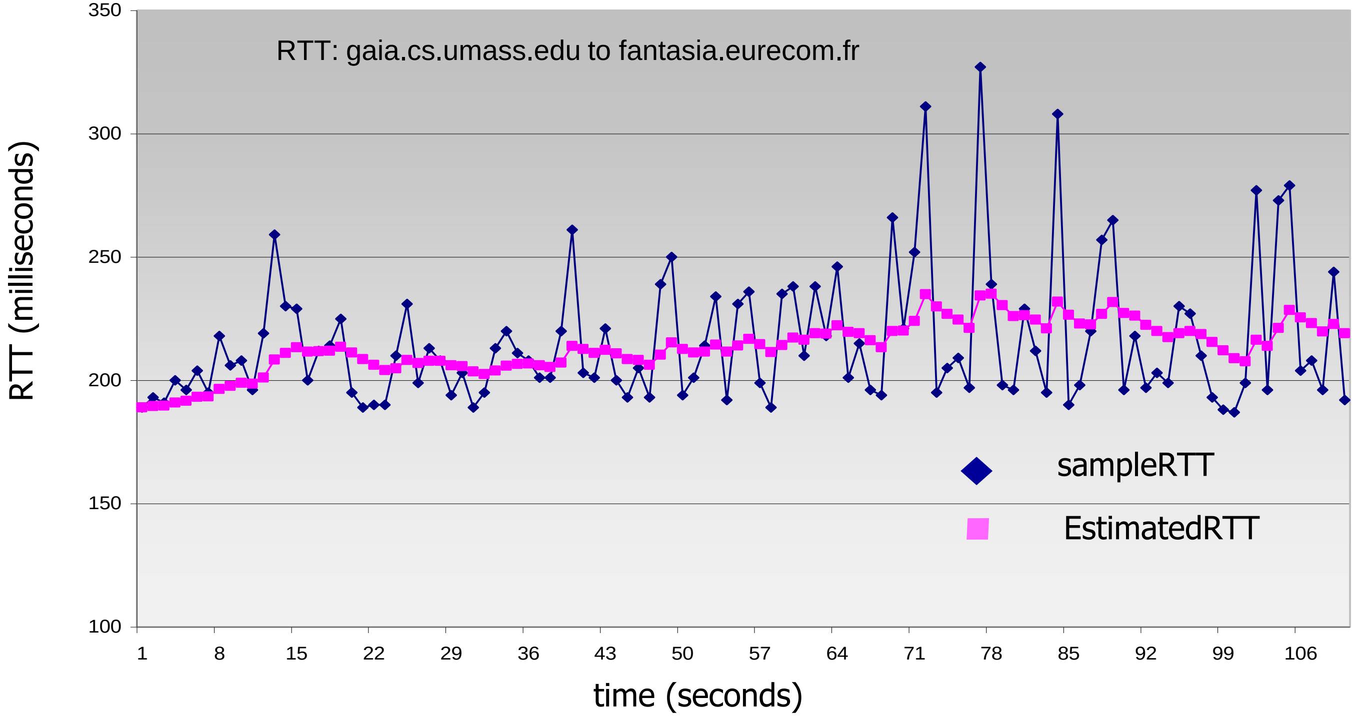
- TCP maintains an average, called `EstimatedRTT`, of the `SampleRTT` values.
- Upon obtaining a new `SampleRTT`, TCP updates `EstimatedRTT` according to the following formula:

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

  - Weighted combination of the previous value of `EstimatedRTT` and the new value for `SampleRTT`
  - Exponential Weighted Moving Average (EWMA)
  - Influence of past sample decreases exponentially fast
  - Typical value: $\alpha = 0.125$

Timeout interval: Estimating the RTT

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$



Timeout interval: Estimating the RTT

$$EstimatedRTT \; = \; (1-\alpha) * EstimatedRTT \; + \alpha * SampleRTT$$

- What about the variability of the RTT?
- DevRTT: RTT variation
  - how much SampleRTT typically deviates from EstimatedRTT

$$DevRTT \; = \; (1-\beta) * DevRTT \; + \beta * |SampleRTT - EstimatedRTT|$$

- DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT.
  - If the SampleRTT values have little fluctuation, then DevRTT will be small
  - if there is a lot of fluctuation, DevRTT will be large.
- Recommended value of β is 0.25.

Timeout Interval Calculation

- `TimeoutInterval` should be $\geq EstimatedRTT$, or unnecessary retransmissions would be sent.
- `TimeoutInterval` should not be $\gg EstimatedRTT$, otherwise interval when a segment is lost, TCP would not quickly retransmit the segment, leading to large data transfer delays.
- Desirable to set the timeout equal to the EstimatedRTT plus some margin

⊙ Margin should be

- Large when there is a lot of fluctuation in the SampleRTT values $\Rightarrow \boldsymbol{DevRTT}$
- Small when there is little fluctuation.

Timeout Interval Calculation

$$EstimatedRTT \;=\; (1-\alpha) * EstimatedRTT \;+ \alpha * SampleRTT$$

$$DevRTT \;=\; (1-\beta) * DevRTT \;+ \beta * |SampleRTT - EstimatedRTT|$$

$$TimeoutInterval \;=\; EstimatedRTT \;+ 4 * DevRTT$$ "safety margin"

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
  - expiration interval: **`TimeOutInterval`**

event: timeout

- Retransmit segment that caused timeout
- Restart timer

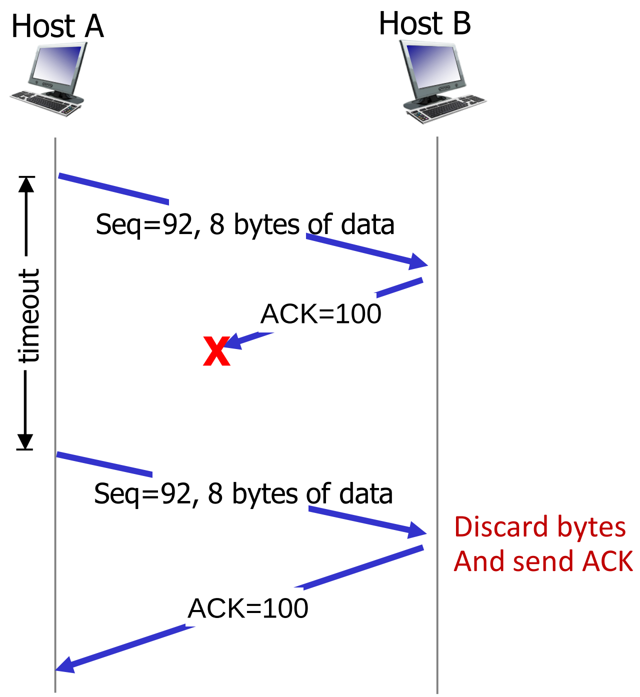
event: ACK received

- if ACK acknowledges previously not-yet-ACKed segments
  - Update what is known to be ACKed
  - Start timer if there are still not-yet-ACKed segments

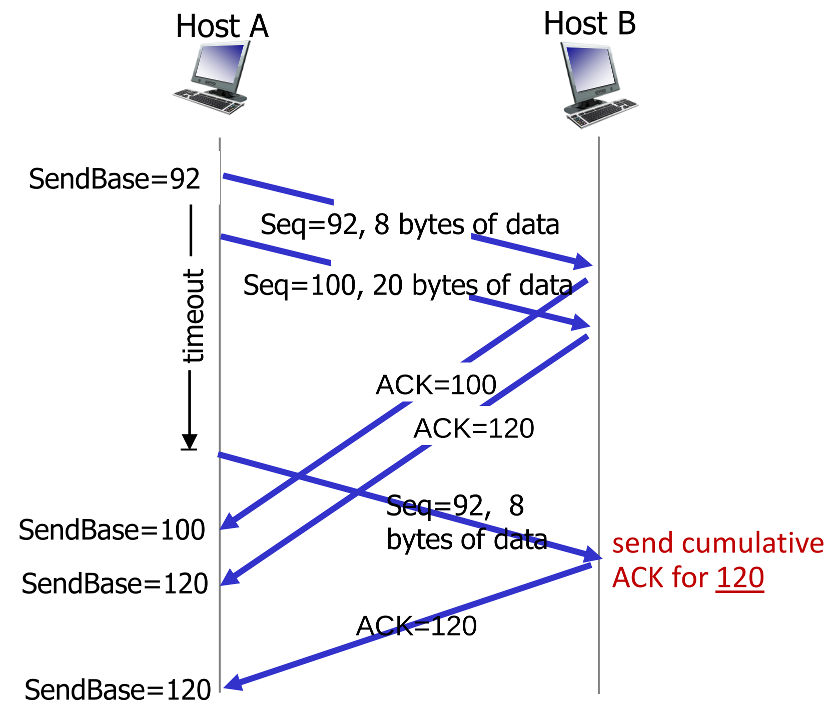
TCP Receiver: ACK Generation [RFC 5681]

Event at receiver	*TCP receiver action*
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK,* indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediately send ACK, provided that segment starts at lower end of gap

TCP: Retransmission Scenarios

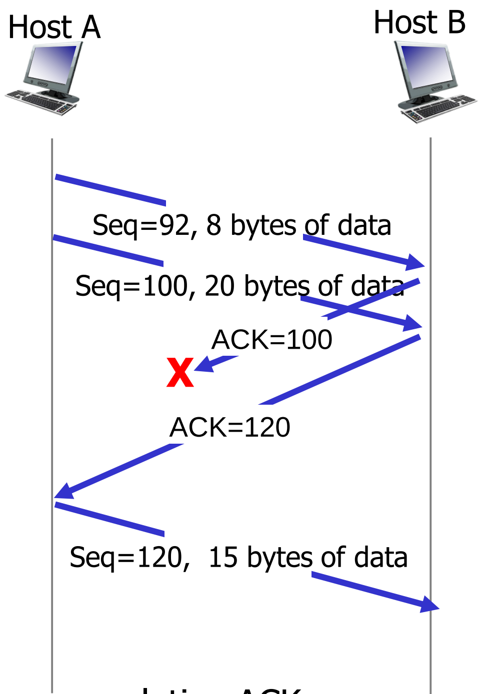


lost ACK scenario



premature timeout

TCP: Retransmission Scenarios



cumulative ACK covers
for earlier lost ACK

Doubling the Timeout Interval

- Initial `TimeoutInterval` value of 1 second is recommended [RFC6298].
- When a timeout occurs, the value of <u>`TimeoutInterval` is doubled</u> to avoid a premature timeout occurring for a subsequent segment that will soon be acknowledged.
- As soon as a segment is received and `EstimatedRTT` is updated, the `TimeoutInterval` is again computed using the formula above.
- Example:
  1. Let `TimeoutInterval` = 0.75 sec at time t, and Timer expires
  2. TCP will then retransmit this segment and set `TimeoutInterval` to 1.5 sec.
  3. If the timer expires again (1.5 sec later), TCP will again retransmit this segment, now setting `TimeoutInterval` to 3.0 sec.
     - The intervals grow exponentially after each retransmission.
  4. Whenever data received from application above, and ACK received, the `TimeoutInterval` is derived from the most recent values of `EstimatedRTT` and `DevRTT`.

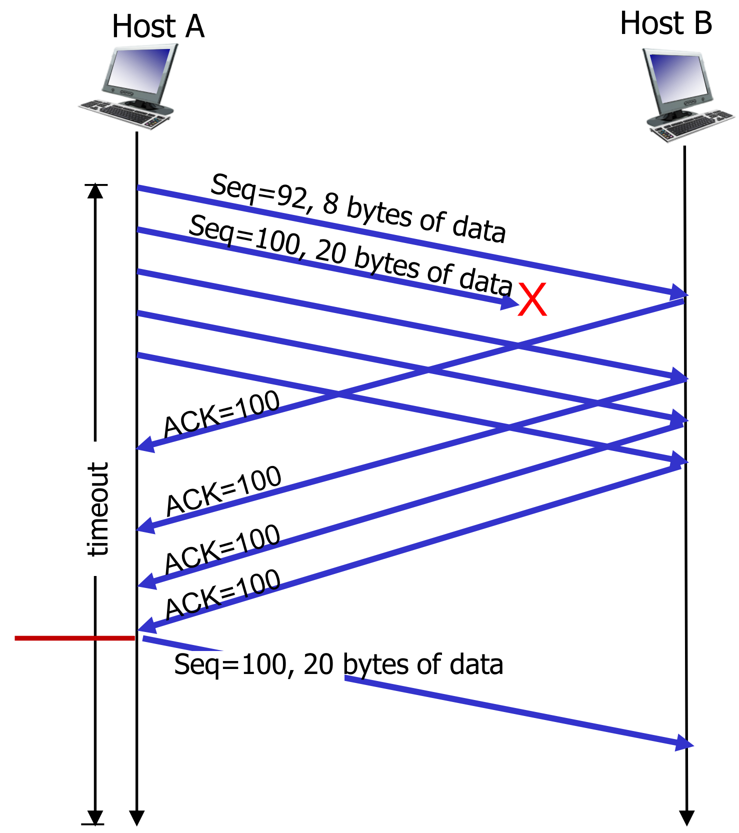
TCP Fast Retransmit

TCP fast retransmit

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

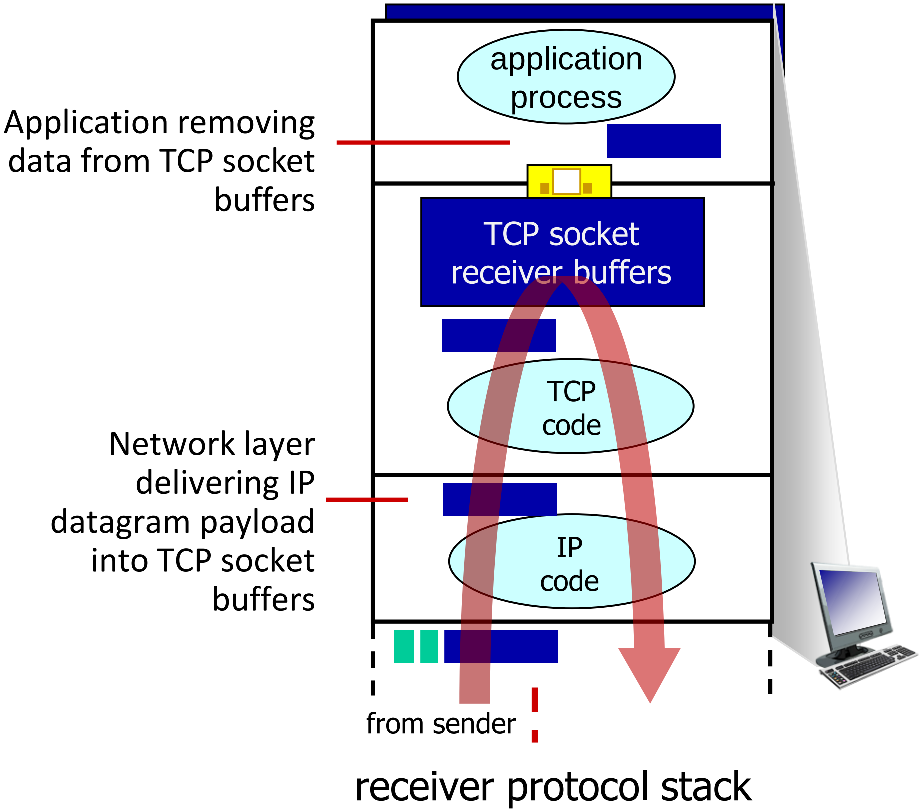
- likely that unACKed segment lost, so don't wait for timeout

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



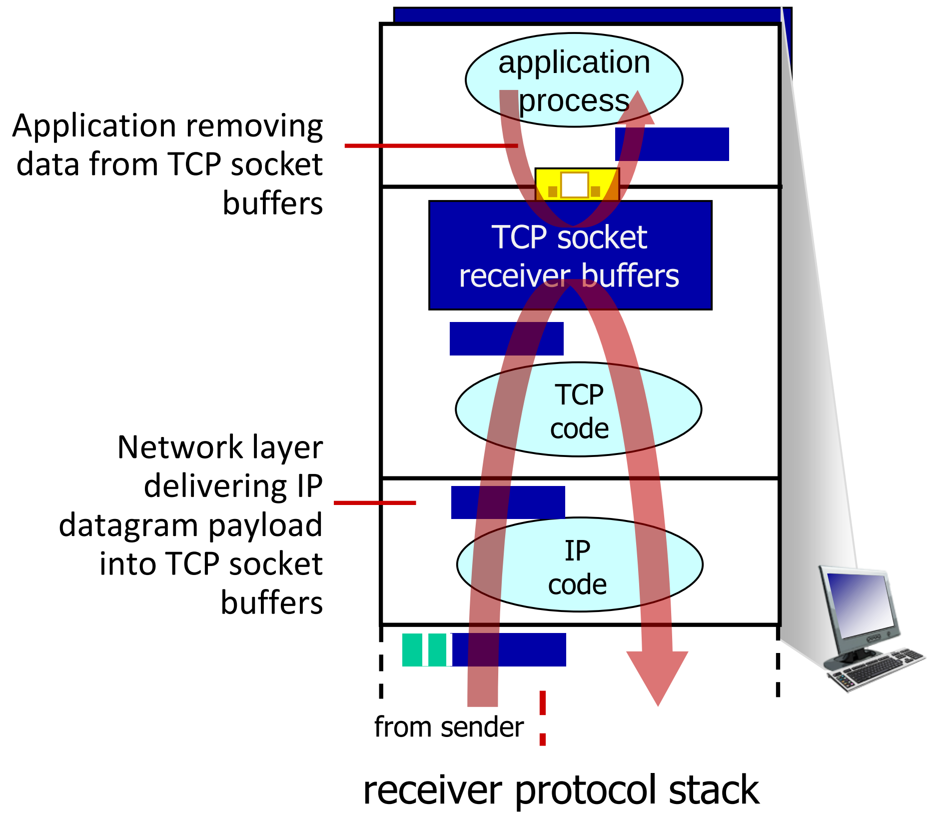
TCP Flow Control

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



TCP Flow Control

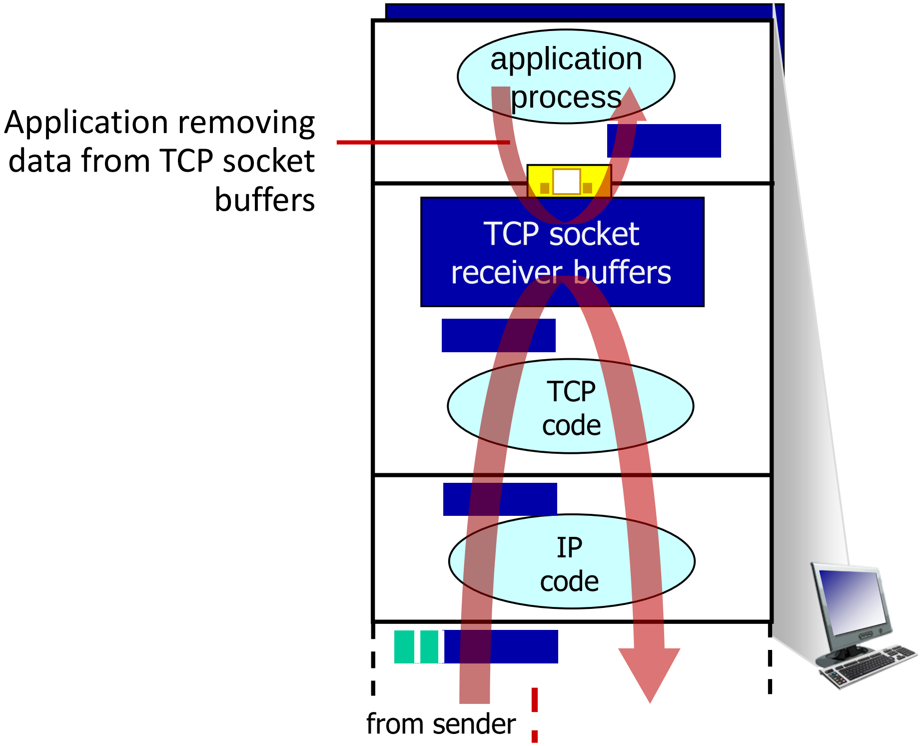
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP Flow Control

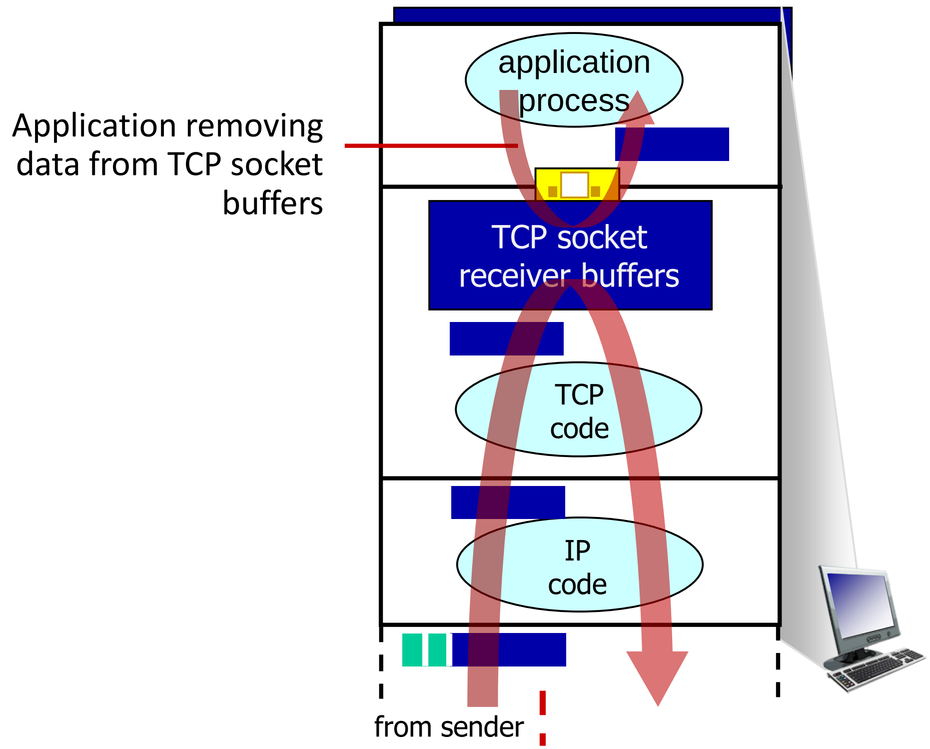
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

TCP provides a **flow-control** service to its applications to eliminate the possibility of the sender overflowing the receiver's buffer.



receiver protocol stack

TCP Flow Control



receiver protocol stack

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

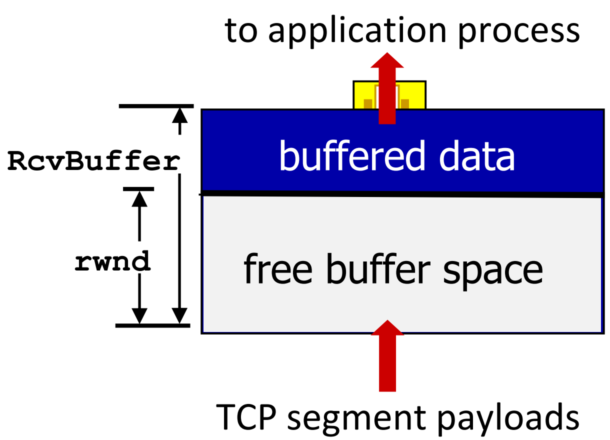
flow control

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

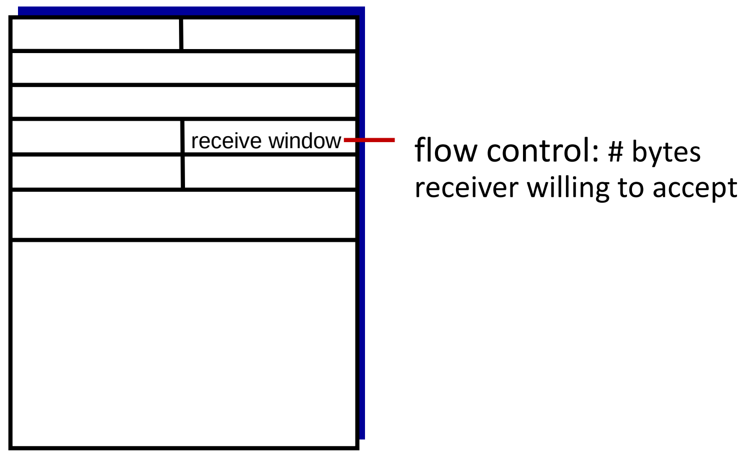
Speed- matching service:
Matching the rate at which the sender is sending against the rate at which the receiving application is reading.

TCP Flow Control

- TCP receiver "advertises" free buffer space in `rwnd` (*receive window*) field in TCP header
  - ⊙`RcvBuffer` size set via socket options (typical default is 4096 bytes)
  - ⊙Many operating systems auto-adjust `RcvBuffer`
- Sender limits amount of unACKed ("in-flight") data to received `rwnd`
- Guarantees receive buffer will not overflow
- **TCP is full-duplex** => the sender at each side of the connection maintains a distinct receive window.



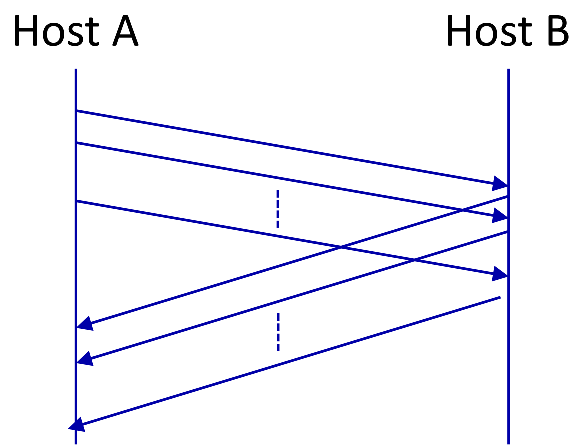
TCP receiver-side buffering



TCP Flow Control – In practice

Host A is sending a large file to Host B over a TCP connection

`LastByteSent`
`LastByteAcked`



Host B allocates a receive buffer to this connection (of size `RcvBuffer`)

From time to time, the application process in B reads from the buffer

$$LastByteSent - LastByteAcked \leq rwnd$$

`LastByteRead`: # of the last byte read from the buffer by the application process in B

`LastByteRcvd`: # of the last byte arrived from the network and placed in receive buffer at B

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

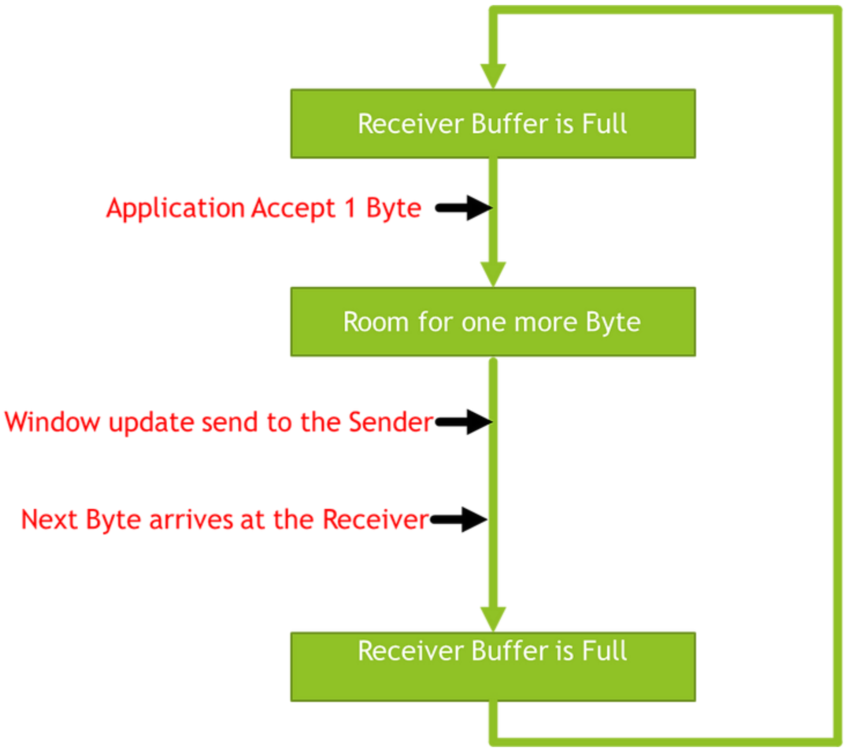
$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

$$rwnd_0 = RcvBuffer$$

TCP Flow Control – In practice

What if $rwnd = 0$?

- Host A stops sending until Receive buffer empties
- Suppose B is not sending Data to A
- How new $rwnd$ value will be communicated to A?
- **Solution**
    - the TCP specification requires Host A to continue to send segments with one data byte when B's receive window is zero.
    - These segments will be acknowledged by the receiver.
    - Eventually the buffer will begin to empty and the acknowledgments will contain a nonzero $rwnd$ value.



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 numbers)

TCP Connection Management- Three way handshake

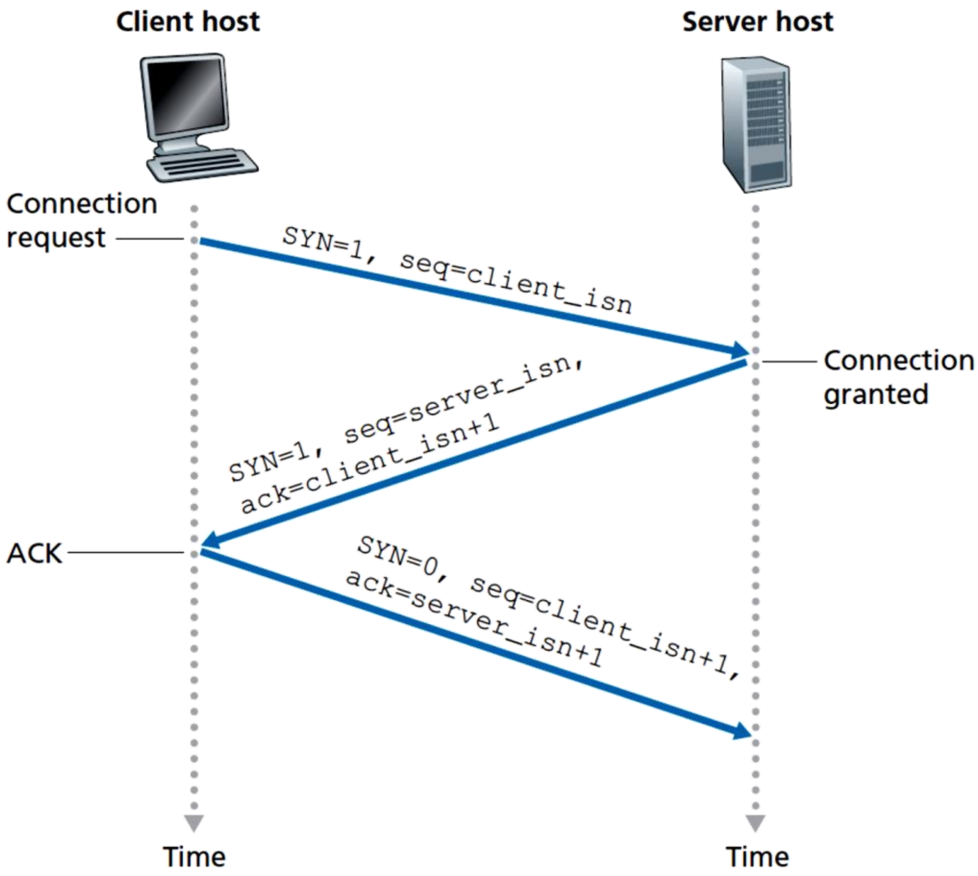
Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

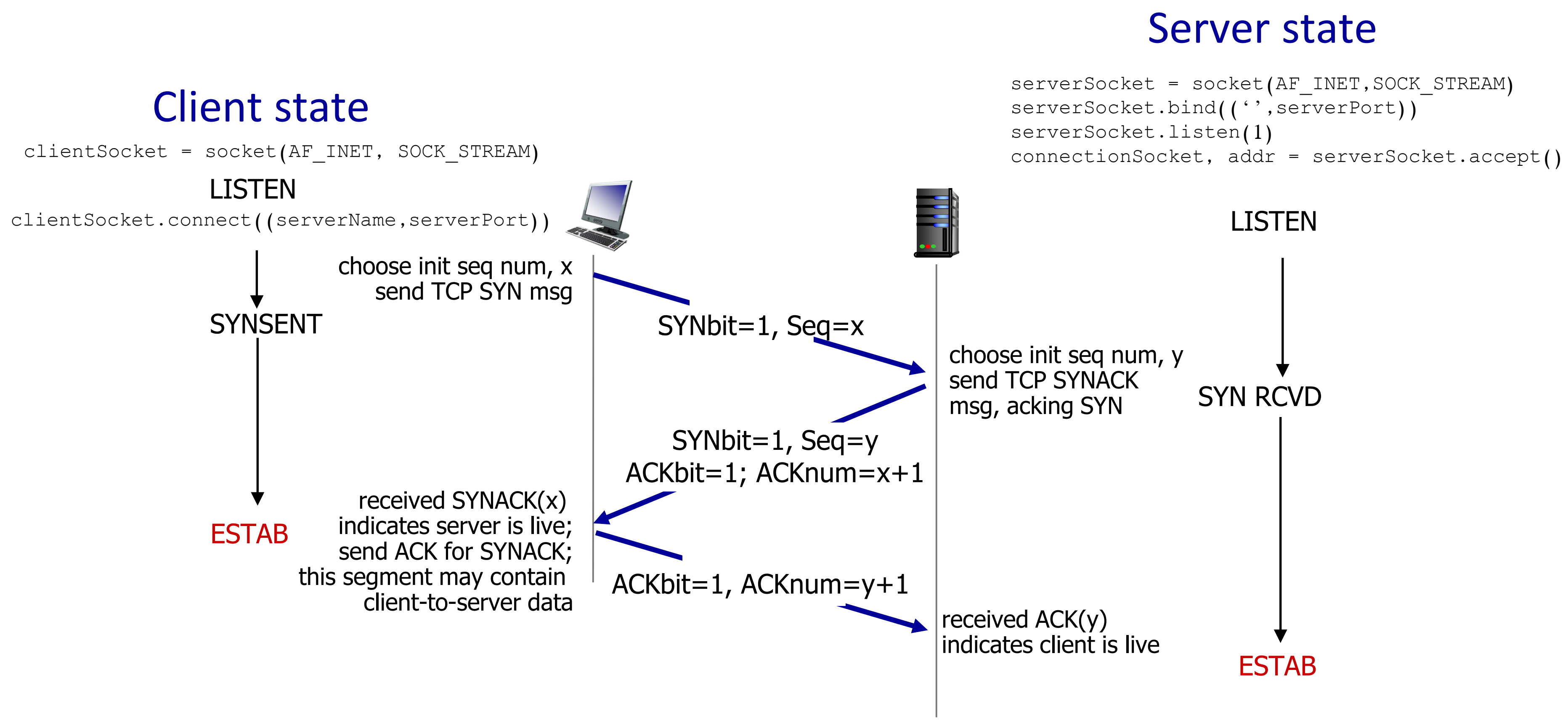
- server allocates buffers
- specifies server initial seq #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data



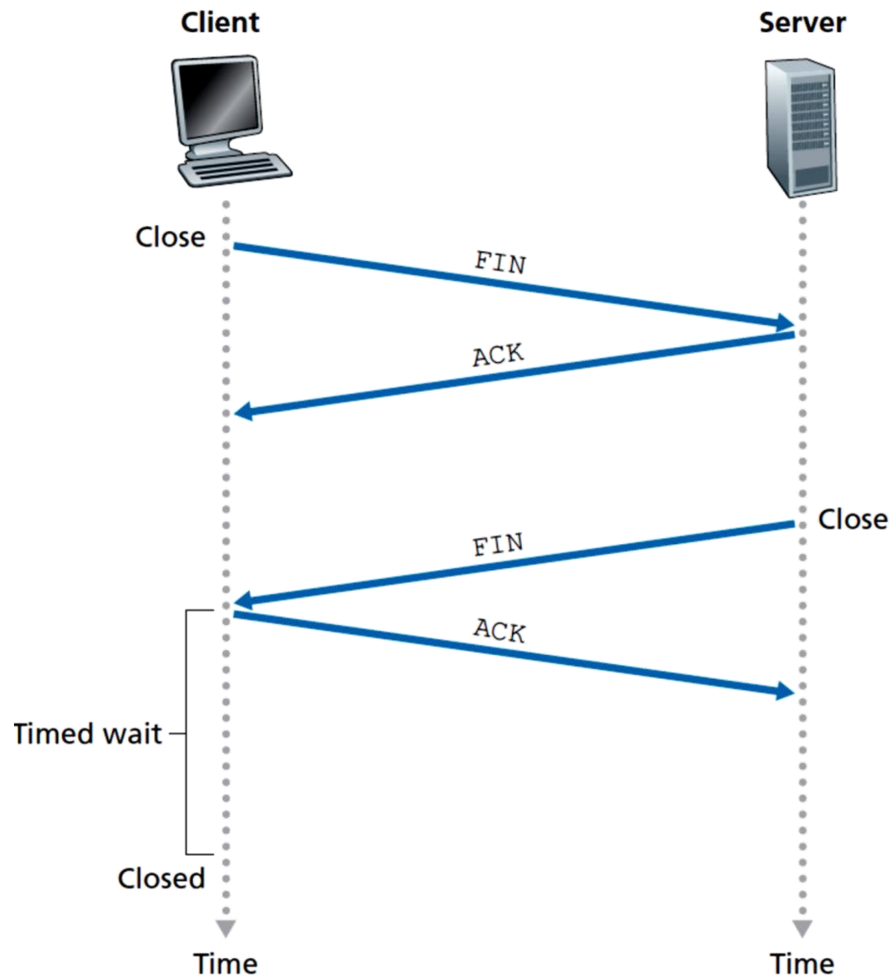
- `client_isn` and `server_isn` randomly chosen

TCP 3-way handshake



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



Closing a TCP connection

Client closes socket:

```
clientSocket.close();
```

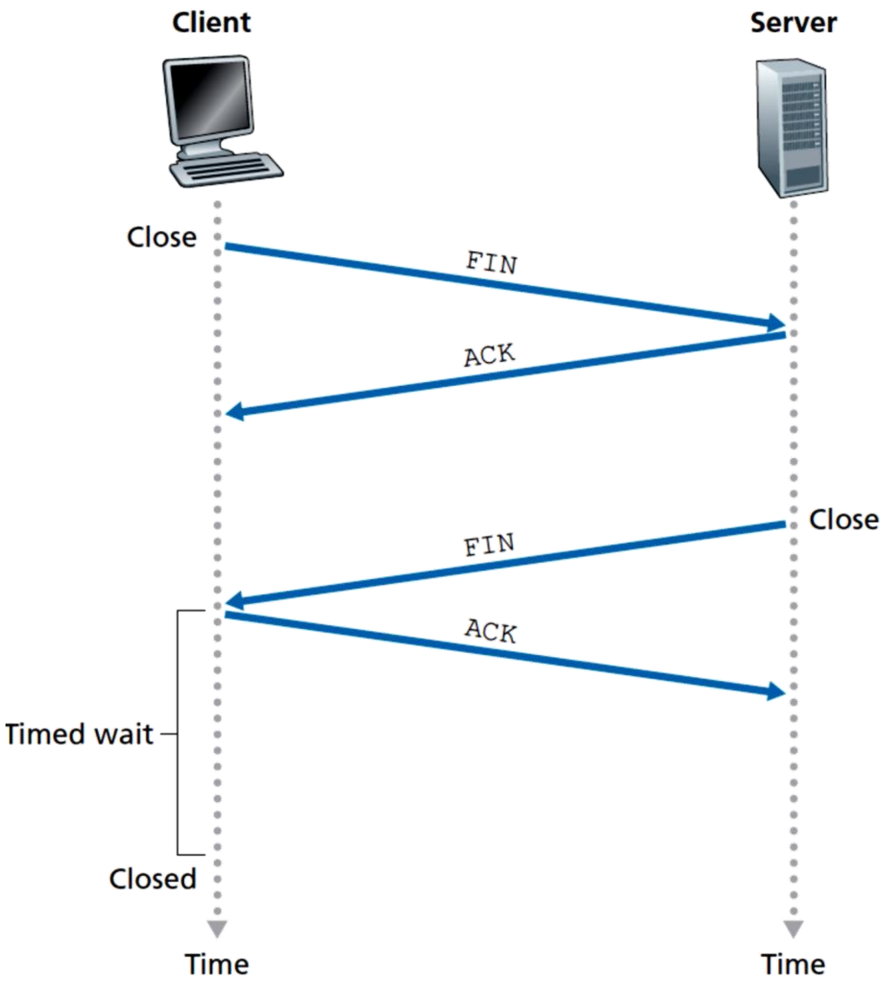
Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

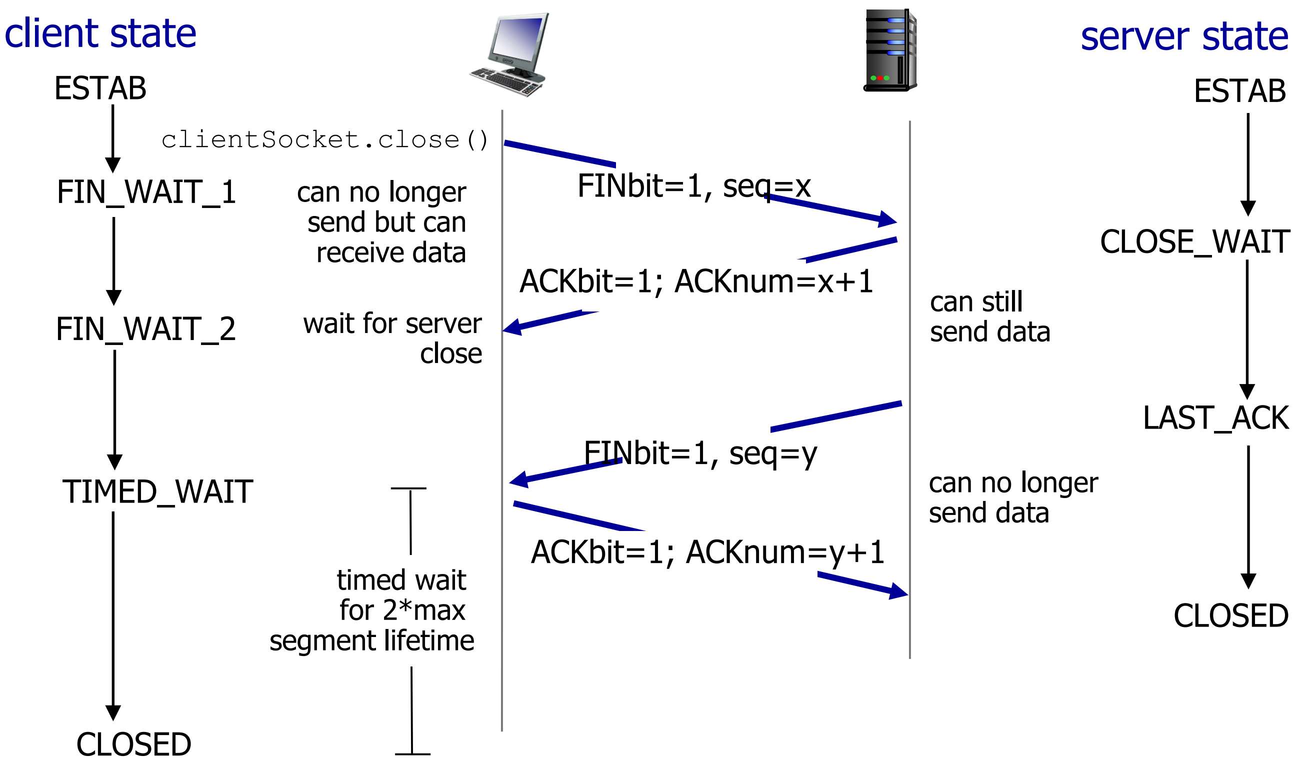
Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

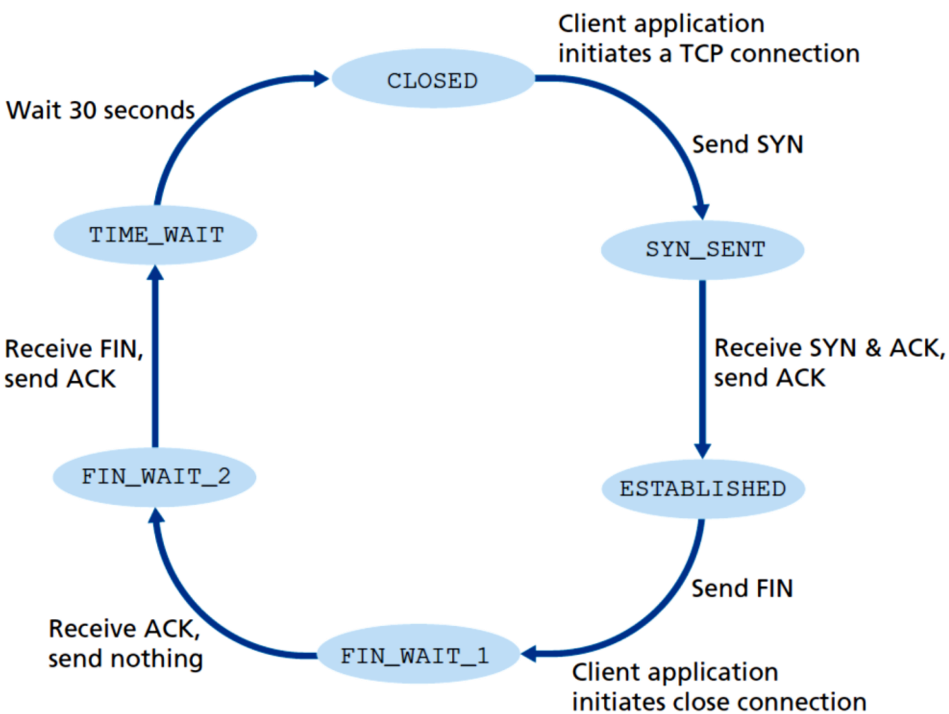
Step 4: server, receives ACK. Connection closed.



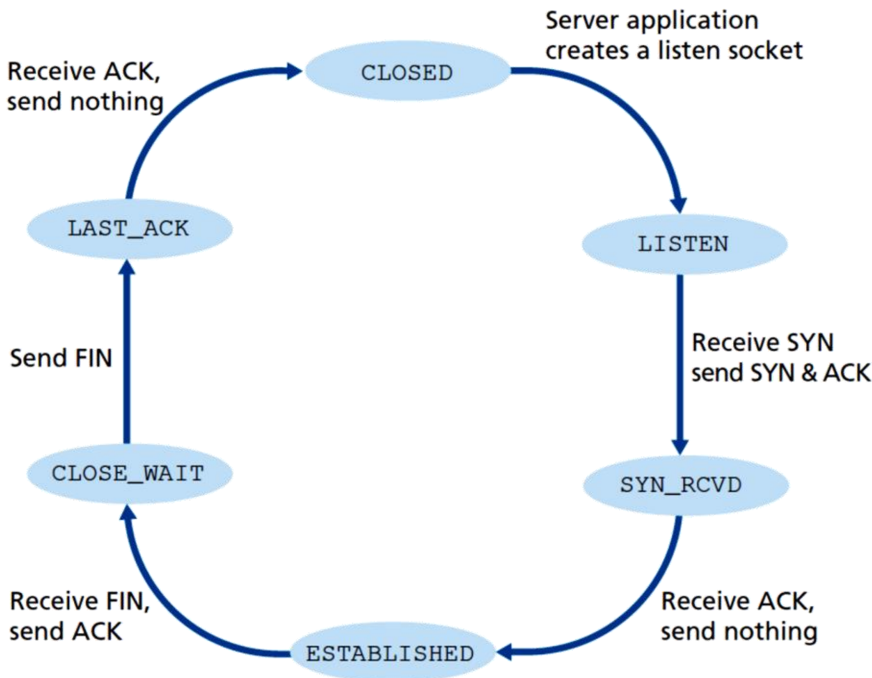
Closing a TCP connection



TCP States



TCP client lifecycle



TCP server lifecycle

Transport-layer services

Multiplexing and De-Multiplexing

Connectionless transport: UDP

Principles of reliable data transfer

Connection-oriented transport: TCP

TCP congestion control

Evolution of transport-layer functionality

Introduction

- **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 top10 problem!

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

TCP congestion control: AIMD

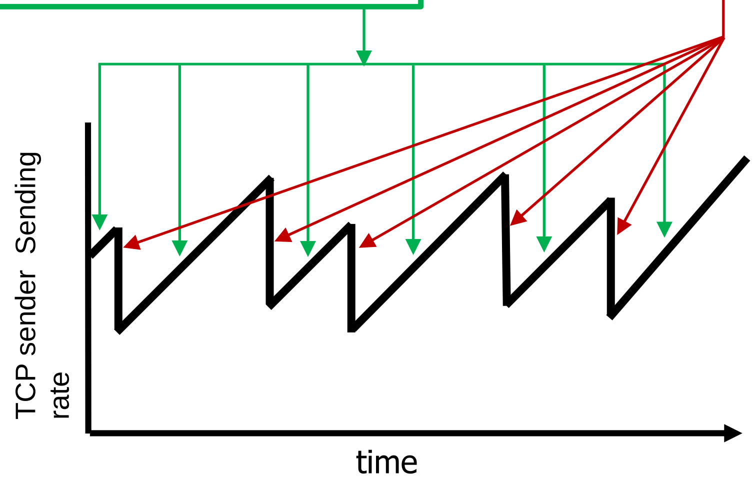
- *Approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

***A**dditive **I**ncrease*

Increase sending rate by 1 Maximum Segment Size (MSS) every RTT until loss detected

***M**ultiplicative **D**ecrease*

Cut sending rate in half at each loss event

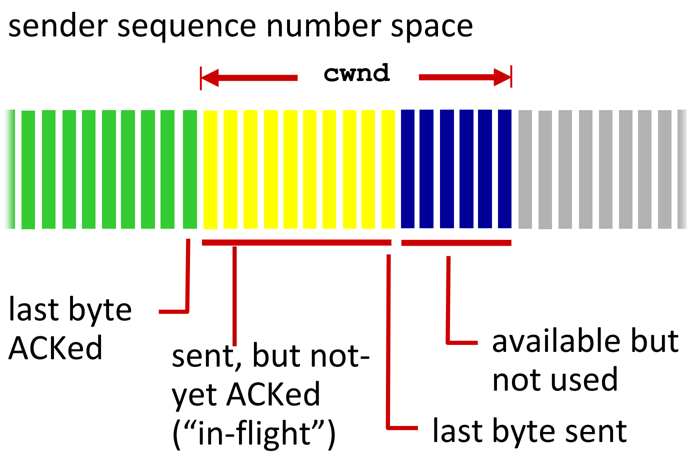


AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

- **Multiplicative decrease** detail: sending rate is
  - Cut in half on loss detected by triple duplicate ACK (**TCP Reno**)
  - Cut to 1 MSS (maximum segment size) when loss detected by timeout (**TCP Tahoe**)
- Why AIMD?
  - AIMD – a distributed, asynchronous algorithm – has been shown to:
    - Optimize congested flow rates network wide!
    - Have desirable stability properties

TCP congestion control: details



TCP sending behavior:

- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes
- $TCP\ rate \approx \frac{cwnd}{RTT}$ bytes/sec

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

TCP Congestion Control

- TCP has a mechanism for congestion control.
- The mechanism is implemented at the sender

- The window size at the sender is set as follows:

$$Send\ Window\ =\ MIN\ (flow\ control\ window, congestion\ window)$$

where

- flow control window is advertised by the receiver
- congestion window is adjusted based on feedback from the network

TCP Congestion Control

- The sender has two additional parameters:
  - ⊙**Congestion Window** (**cwnd**)
    Initial value is 1 MSS (=maximum segment size) counted as bytes
  - ⊙**Slow-start threshold Value** (**ssthresh**)

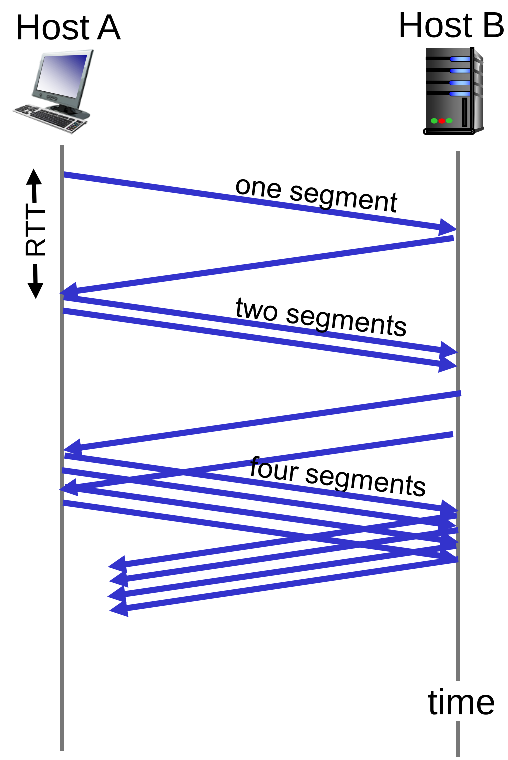
    Initial value is the advertised window size

- Congestion control works in two modes:
  - ⊙**Slow Start** ($cwnd < ssthresh$)
  - ⊙**Congestion Avoidance** ($cwnd \geq ssthresh$)

TCP Slow Start

- When connection begins, increase rate exponentially until first loss event:
  - Initially $cwnd = 1 \ MSS \ (bytes)$
  - Double $cwnd$ every RTT
    - Done by incrementing cwnd for every ACK received
- **Summary**: initial rate is slow, but ramps up exponentially fast

- TCP slows down the increase of $cwnd$ when $cwnd > ssthresh$



Congestion Avoidance

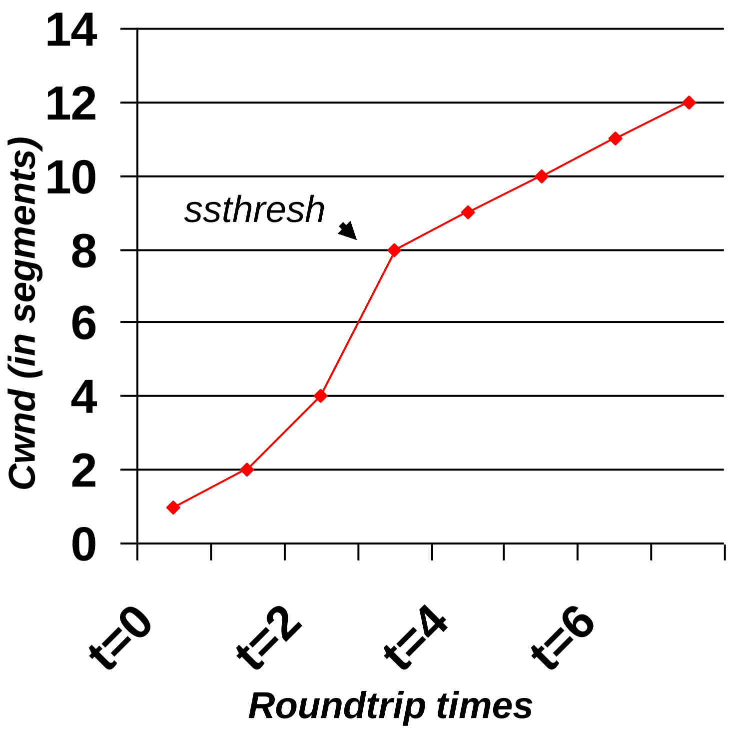
- Congestion avoidance phase is started if $cwnd$ has reached the slow-start threshold value
- If $cwnd \geq ssthresh$ then each time an ACK is received, increment $cwnd$ as follows: $cwnd = cwnd + MSS * (MSS/cwnd)$
- So *cwnd* is increased by one segment (=MSS bytes) only if all segments have been acknowledged.
- Example:
  - $MSS = 1460\ bytes\ , cwnd = 14{,}600\ bytes$
    - 10 segments are being sent within an RTT.
  - Each arriving ACK (assuming one ACK per segment) increases the congestion window size by $(MSS/10)$
  - The value of the congestion window will have increased by one MSS after ACKs when all 10 segments have been received.

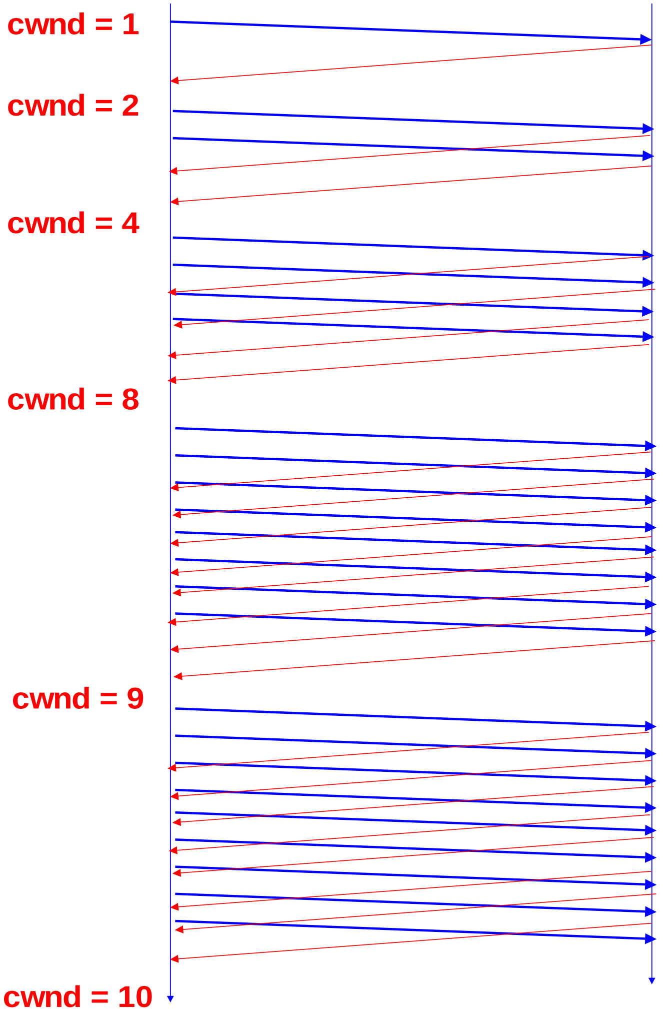
Slow Start / Congestion Avoidance

```
If    cwnd < ssthresh then /* Slow Start*/
        Each time an Ack is received:
        cwnd = cwnd + MSS
else    /*  cwnd >=  ssthresh  -- Congestion Avoidance*/
        Each time an Ack is received :
        cwnd = cwnd + MSS* MSS / cwnd
endif
```

Slow Start/Congestion Avoidance - Example

Assume that *ssthresh = 8*





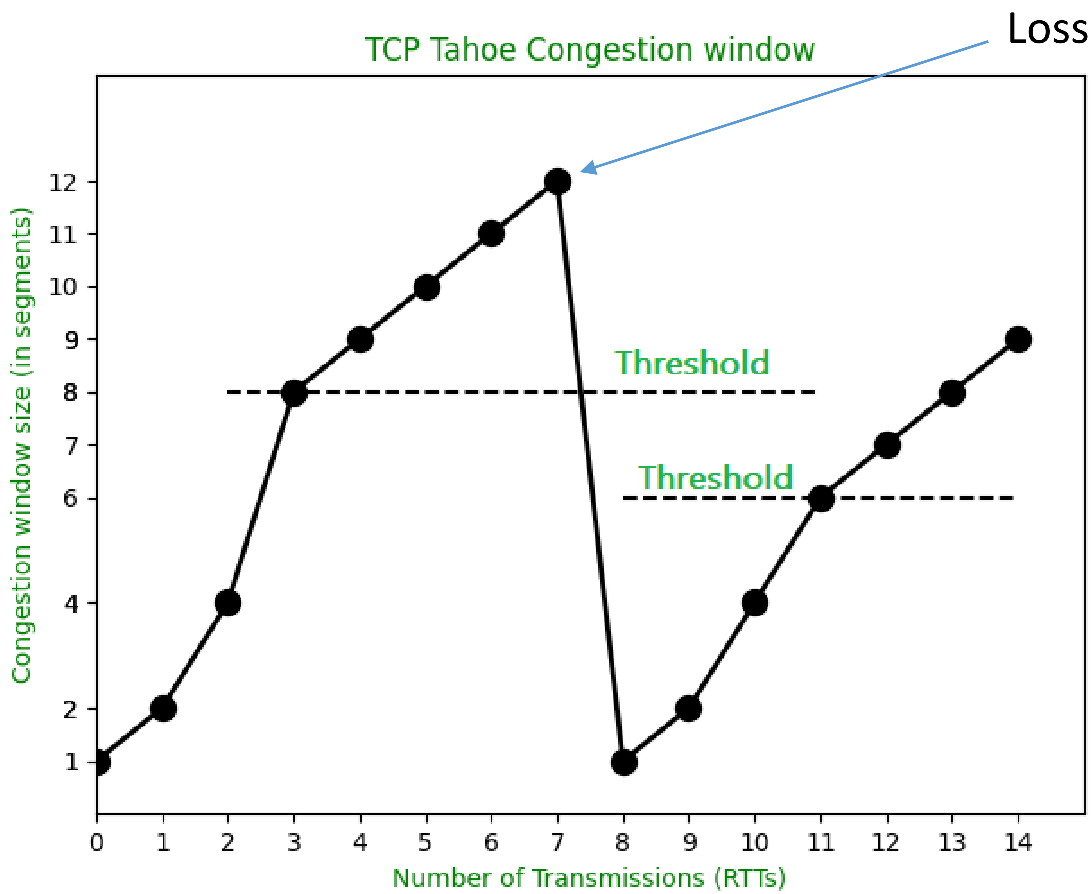
TCP: from Slow Start to Congestion Avoidance

- **Q:** when the linear increase (congestion Avoidance) ends?
- **A:** on loss event: Timeout/ Triple Duplicate ACK
  - $ssthresh = cwnd/2$ (value before loss event)
  - **Loss indicated by timeout:**
    - $cwnd$ set to 1 MSS
    - window then grows exponentially

    (as in slow start) to $ssthresh$, then grows linearly
  - **Loss indicated by 3 duplicate ACKs: TCP RENO**
    - dup ACKs indicate network capable of delivering some segments
    - $cwnd$ is cut in half window then grows linearly
  - **TCP Tahoe always sets $cwnd$ to 1** (timeout or 3 duplicate ACKs)

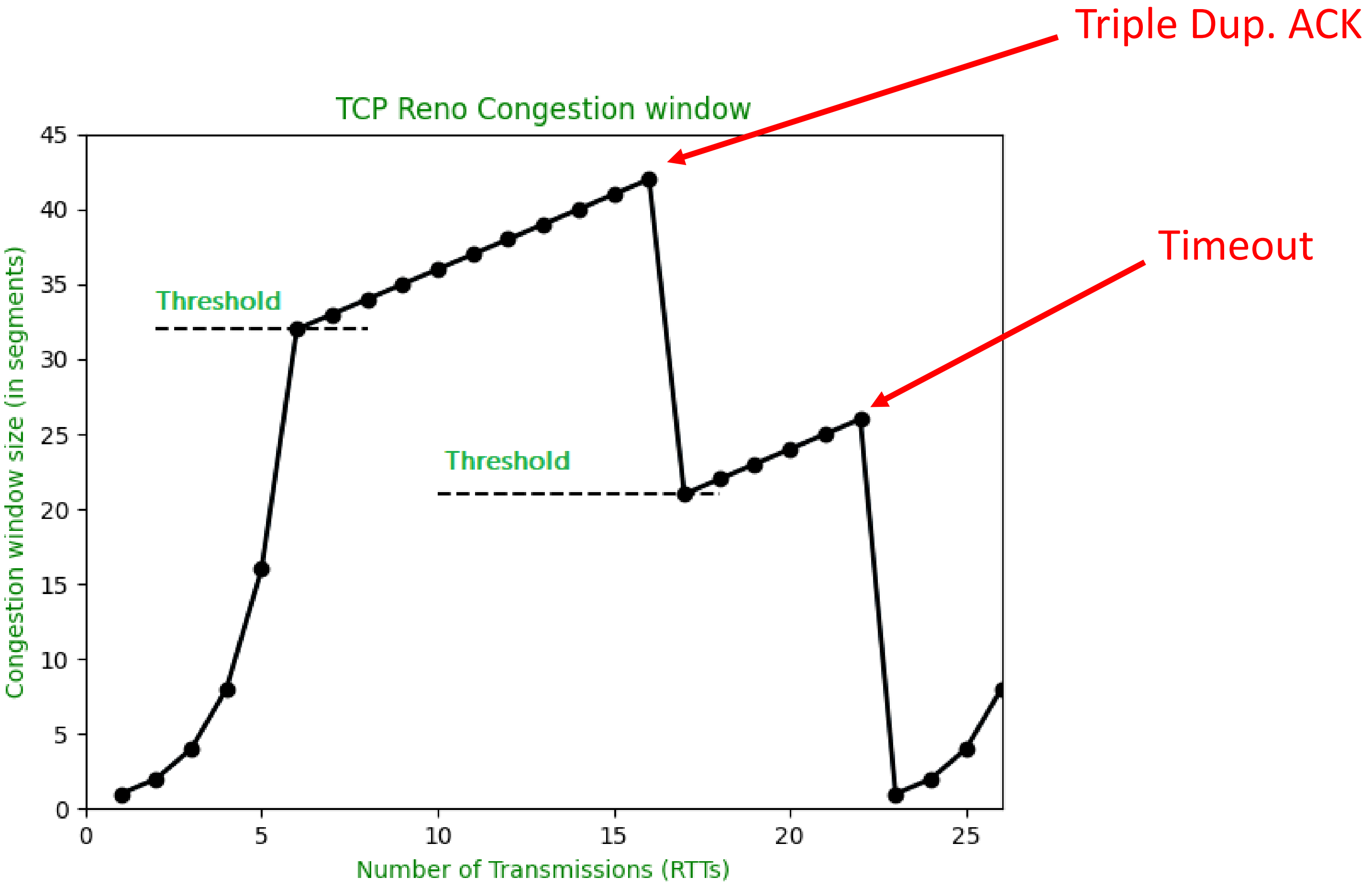
TCP Tahoe – Congestion Control

- TCP Tahoe = Slow Start + AIMD + Fast Retransmit

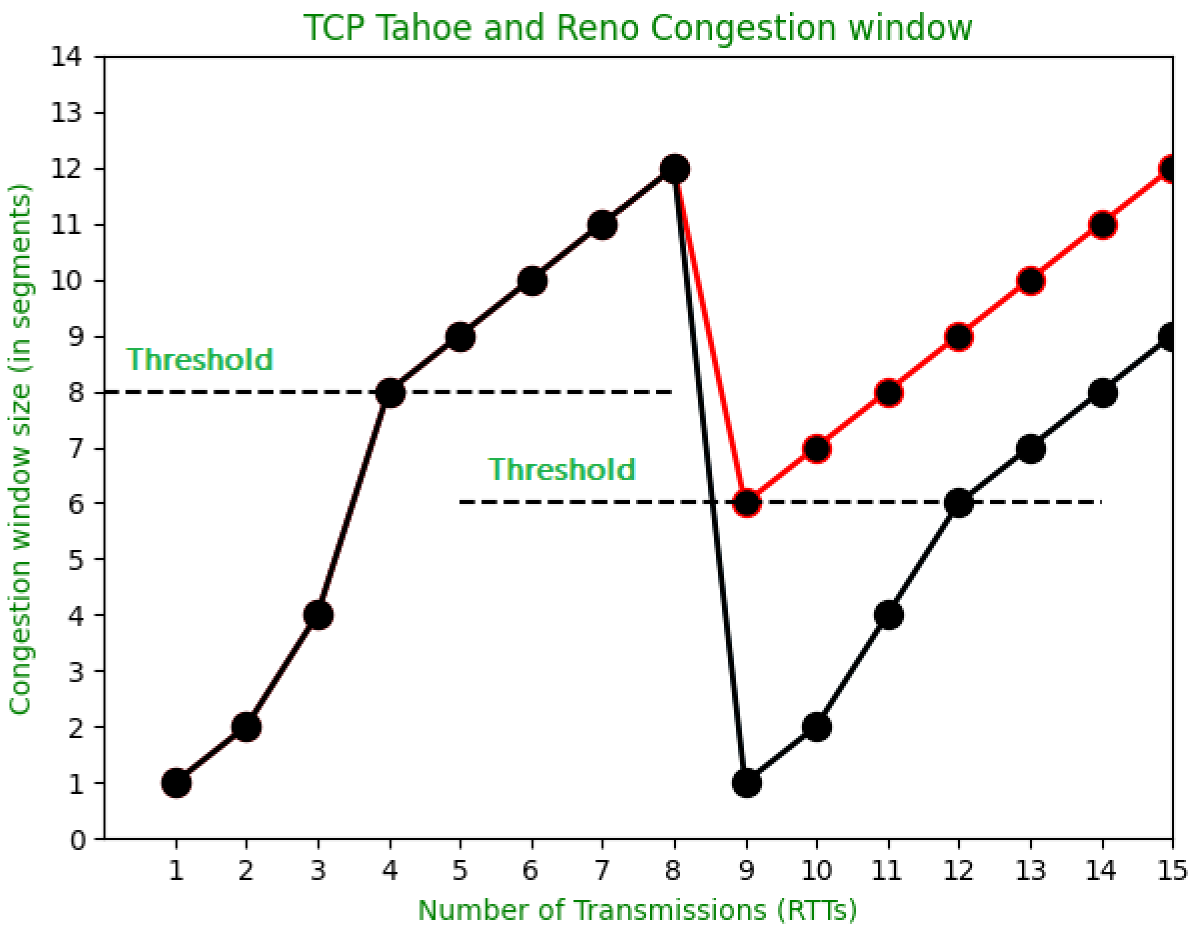


TCP Reno – Congestion Control

- TCP Reno = TCP Tahoe + Fast Recovery

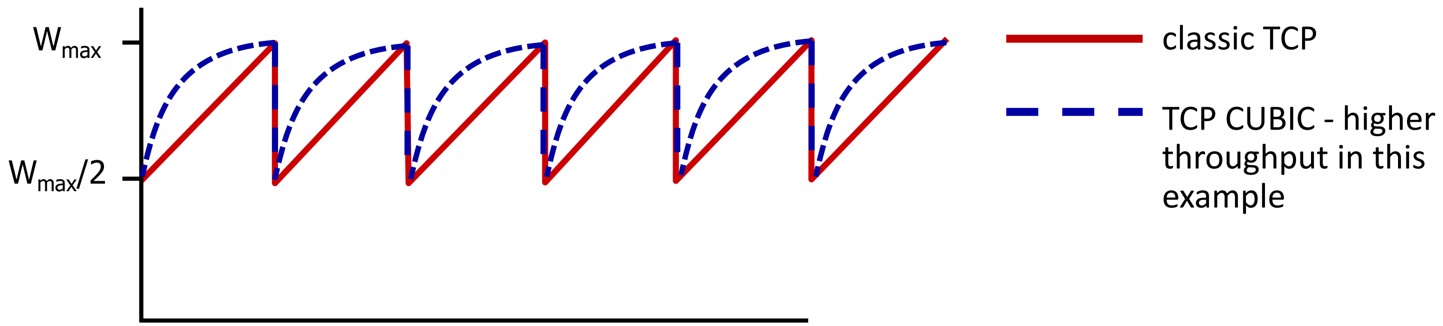


TCP Tahoe vs. Reno – Congestion Control



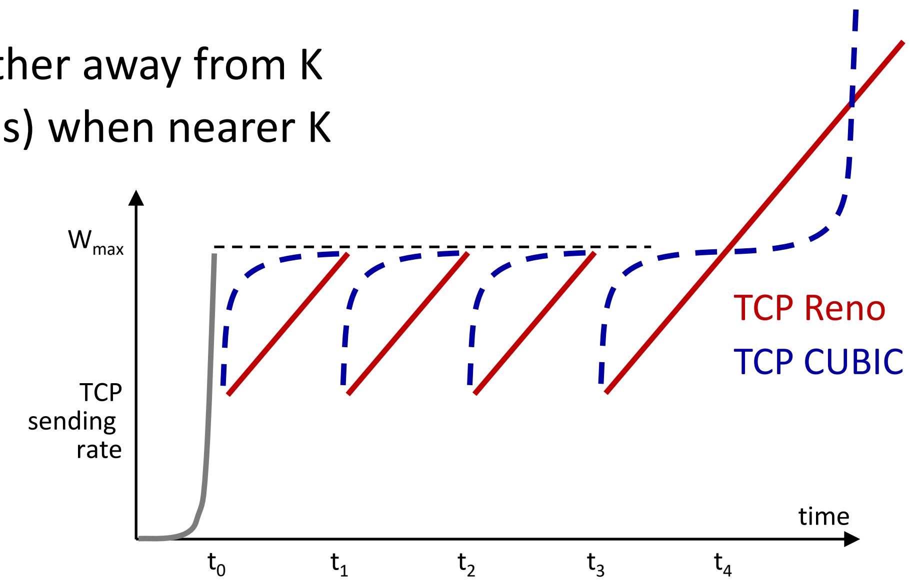
TCP CUBIC

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



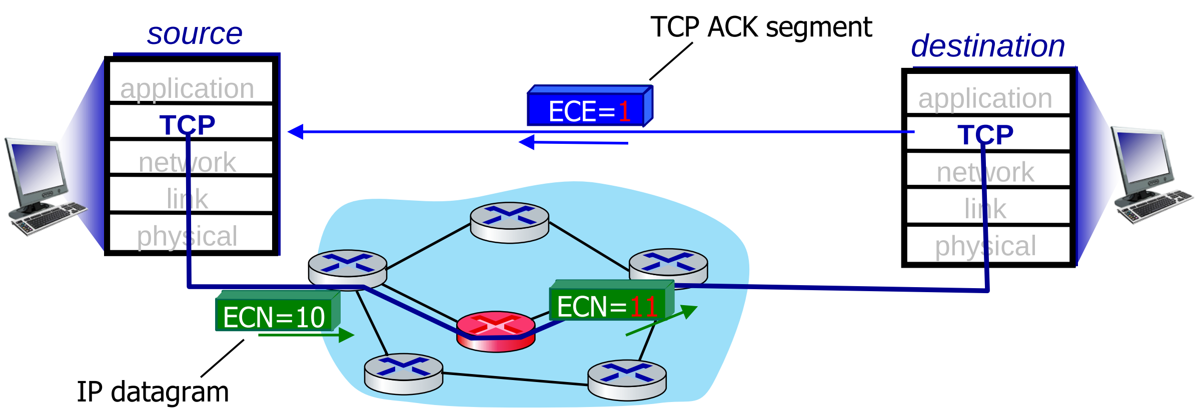
TCP CUBIC

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



Explicit congestion notification (ECN)

- TCP deployments often implement network-assisted congestion control:
- Two bits in IP header (ToS field) marked by network router to indicate congestion
    - ⦿ Policy to determine marking chosen by network operator
- Congestion indication carried to destination
- Destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



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

Evolution of transport-layer functionality

Evolving transport-layer functionality

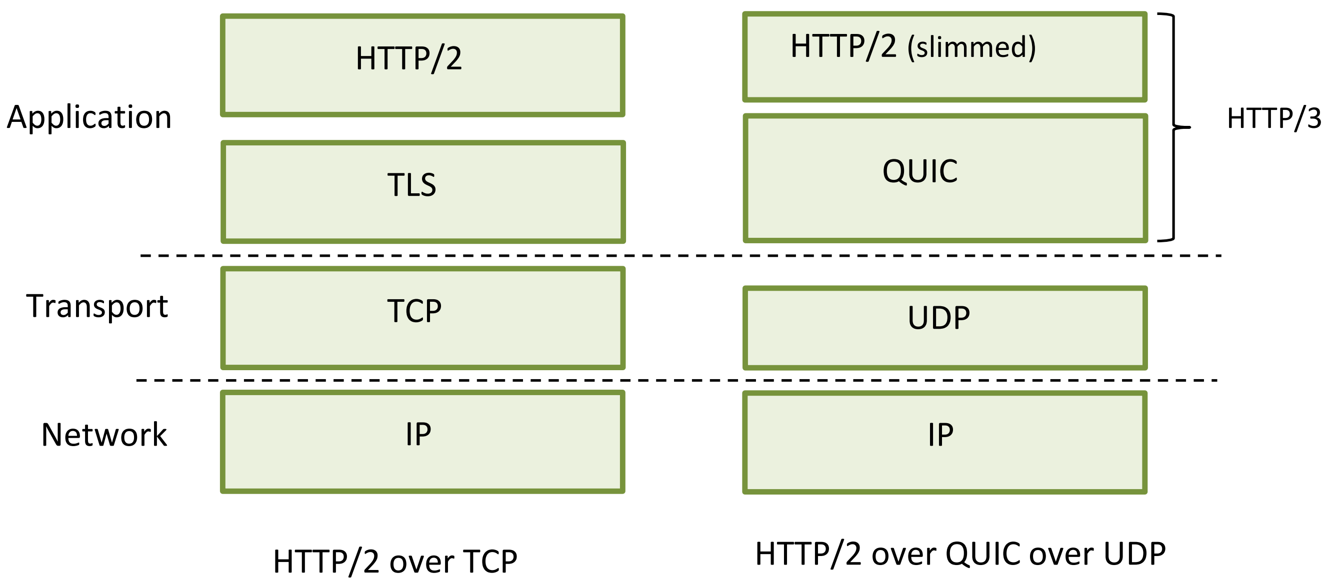
- TCP, UDP: principal transport protocols for 40 years
- Different “flavors” of TCP developed, for specific scenarios:

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

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

QUIC: Quick UDP Internet Connections

- **Application-layer protocol, on top of UDP**
  - Increase performance of HTTP
  - Deployed on many Google servers, apps (Chrome, mobile YouTube app)



QUIC: Quick UDP Internet Connections

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

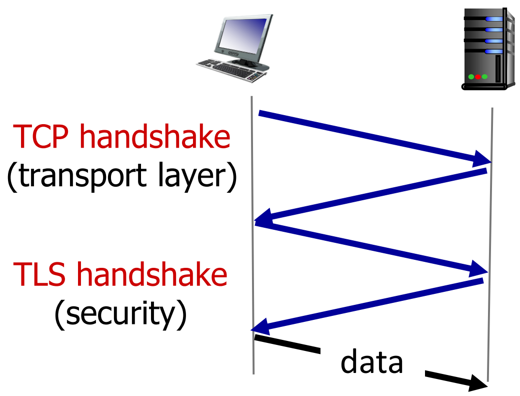
- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established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

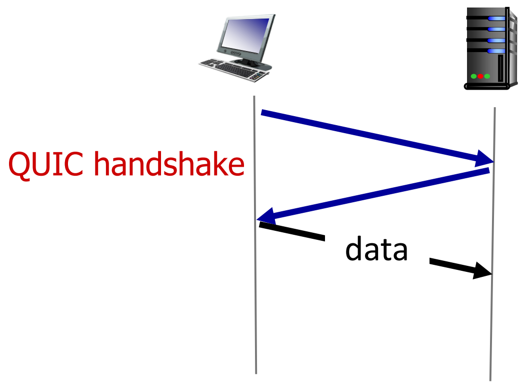
- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Quick UDP Internet Connections

- Adopts approaches already studied for *connection establishment*, *error control*, *congestion control*
  - **Error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
  - **Connection establishment:** *reliability*, *congestion control*, *authentication*, *encryption*, *state* established in one RTT
- Multiple application-level "streams" multiplexed over single QUIC connection
  - Separate reliable data transfer, security
  - Common congestion control

QUIC: Connection establishment





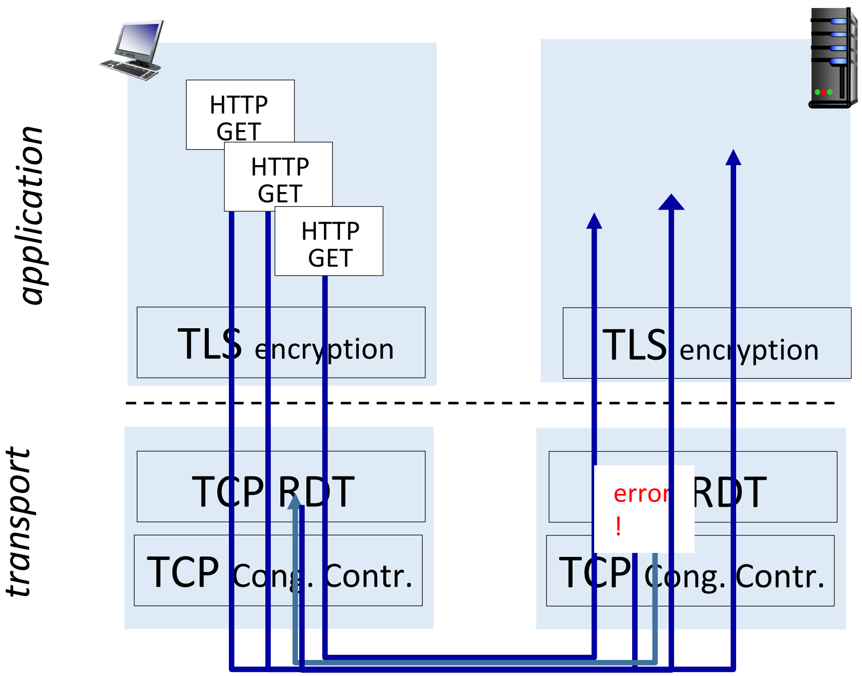
TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1