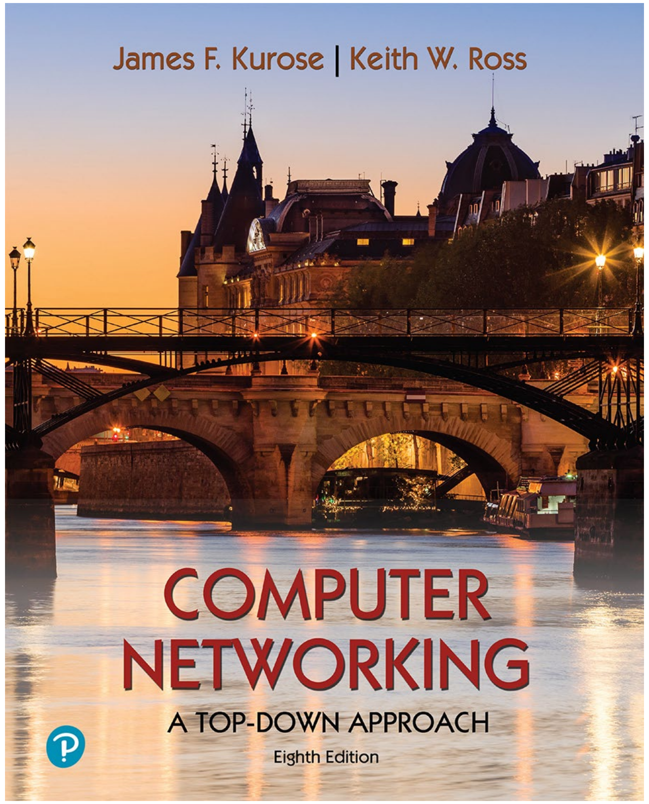
Process-to-Process Data Delivery



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Acknowledgements
These Slides have been adapted from the originals made available by J. Kurose and K. Ross

Problem Position

- GOAL*: Process-to-Process* data delivery*:*
  - logical communication between pairs of processes running on different hosts
- The network layer provides host-to-host delivery
- … but more processes typically run on the same host
- **How to fill in the gap??**
- **Transport layer**
  - relies on, and enhances, network layer services

Goals

- understand **principles** behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer **protocols**:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP flow control
  - TCP congestion control

Roadmap

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

Transport vs. Network layer services and protocols

- Network layer
  - logical communication between *hosts*
- Transport layer
  - logical communication between *processes*
  - More processes running on the same host
    - Need for multiplexing/demultiplexing

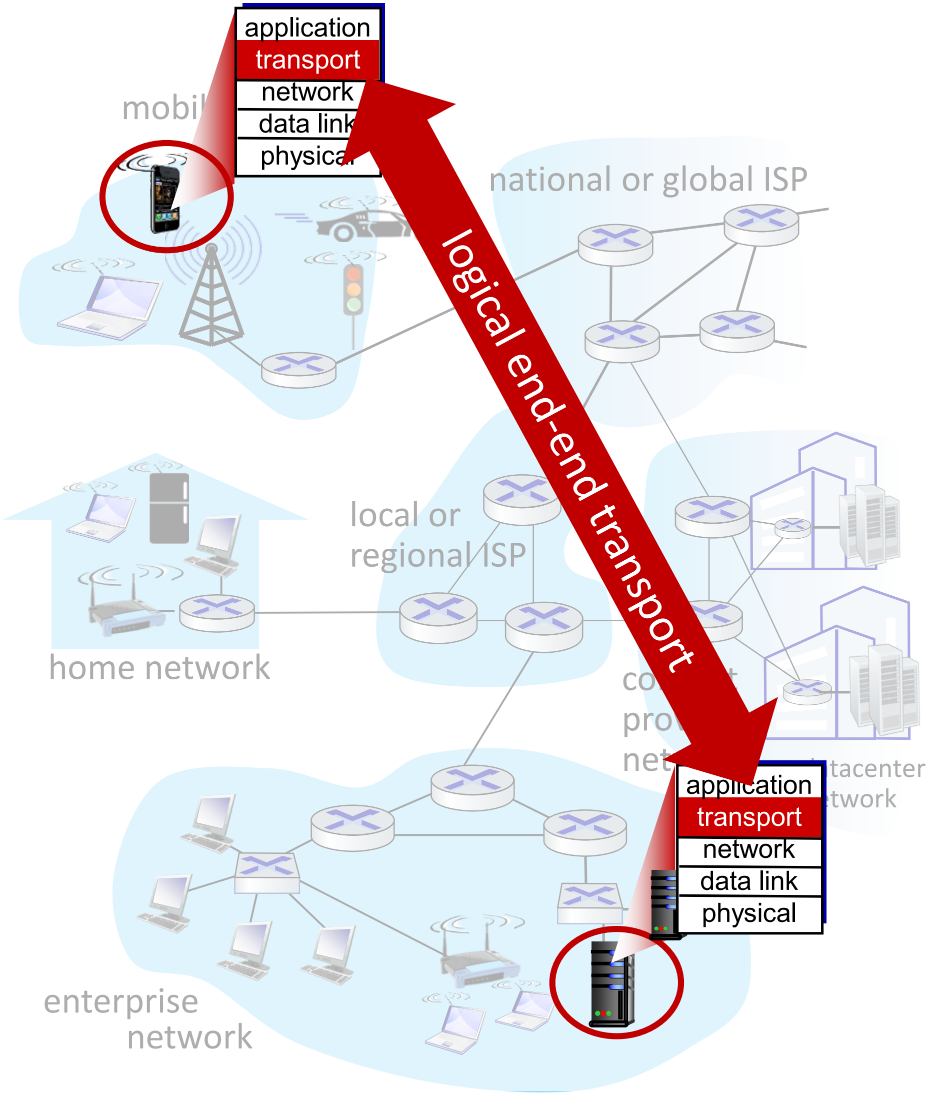
Office Analogy:

- processes = officers
- host = office
- messages = postal letters
- transport protocol = secretary who demux to persons
- network-layer protocol = postal service

Transport services and protocols

- provide *logical communication* between application processes running on different hosts

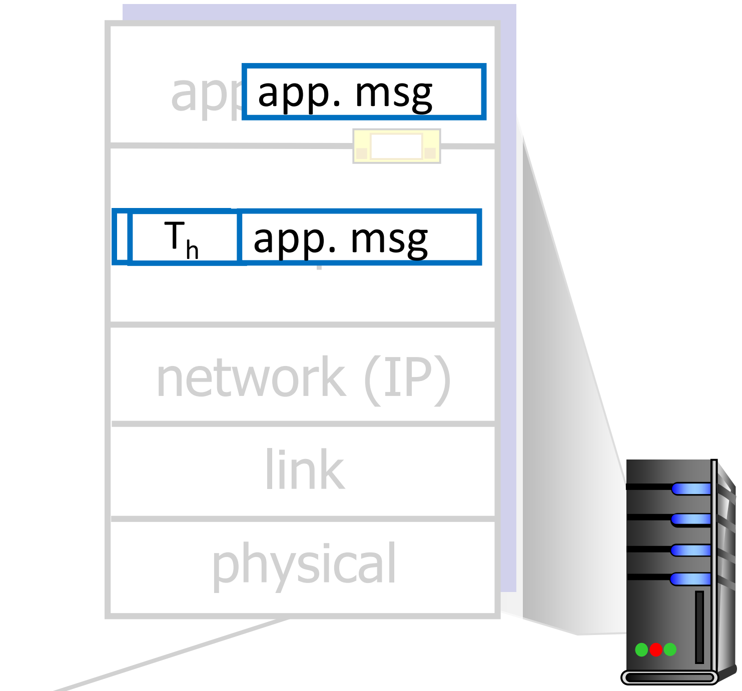
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer



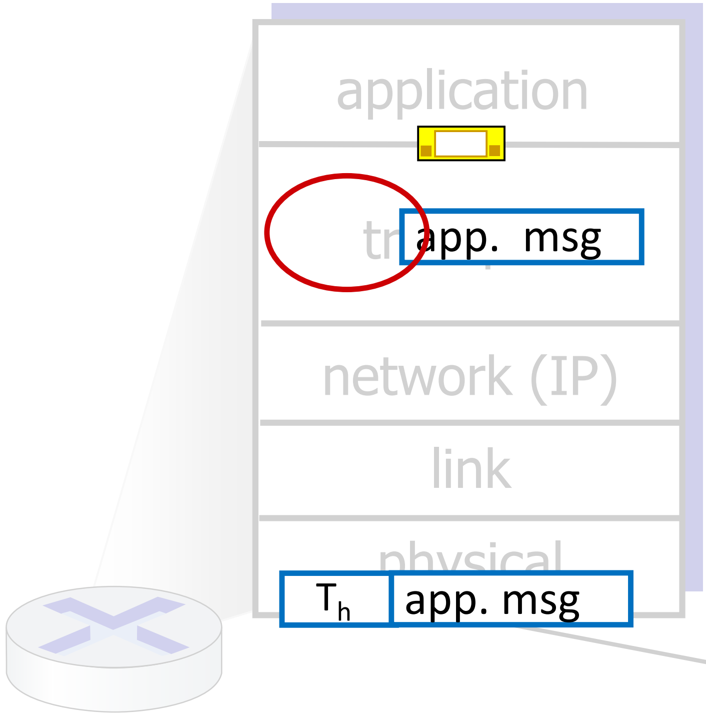
Transport Layer Actions

Sender:

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

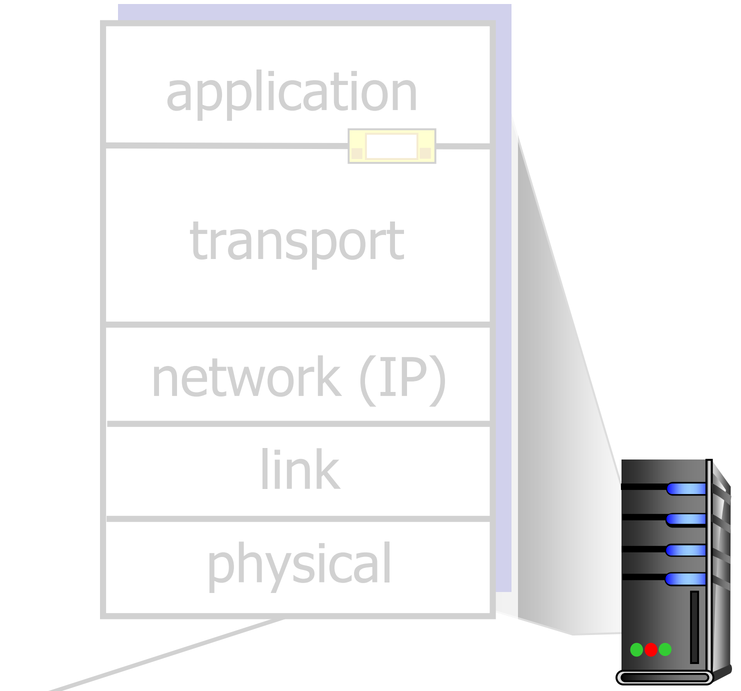


Transport Layer Actions



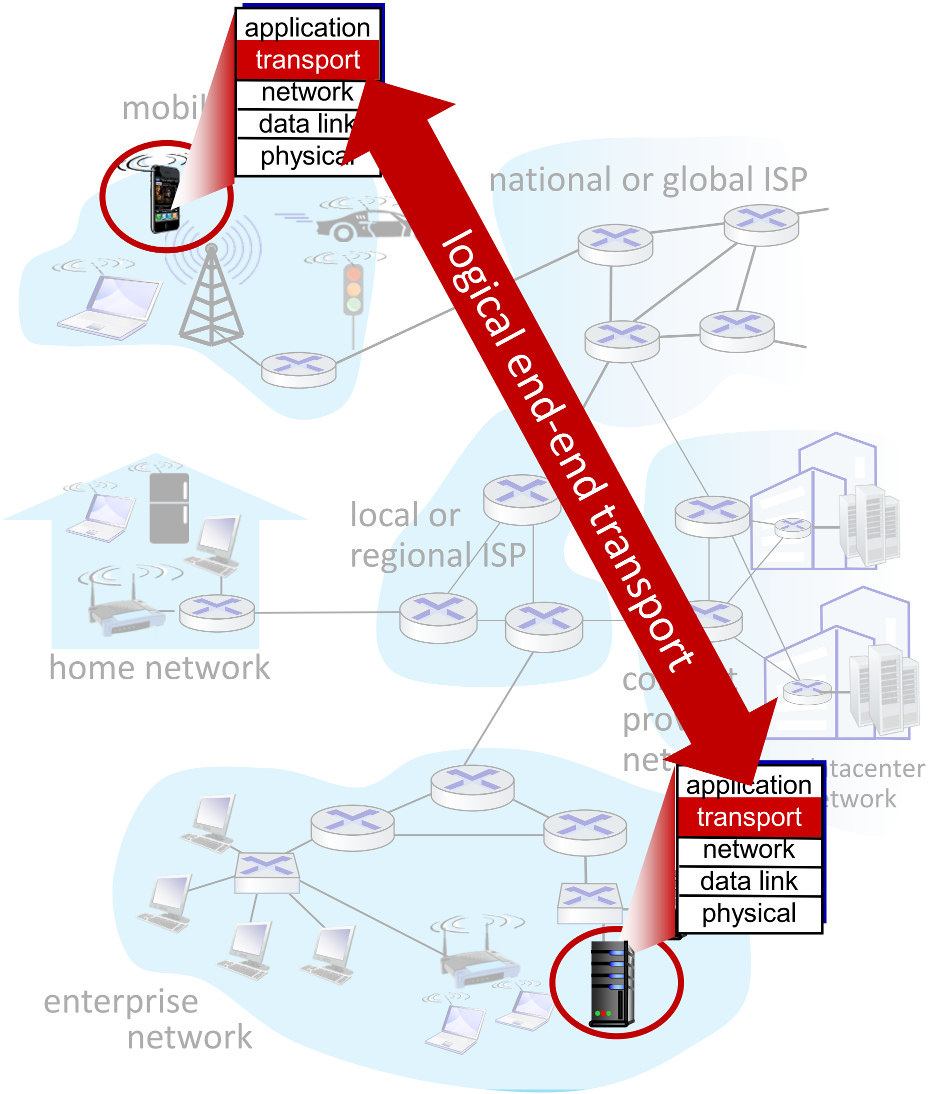
Receiver:

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



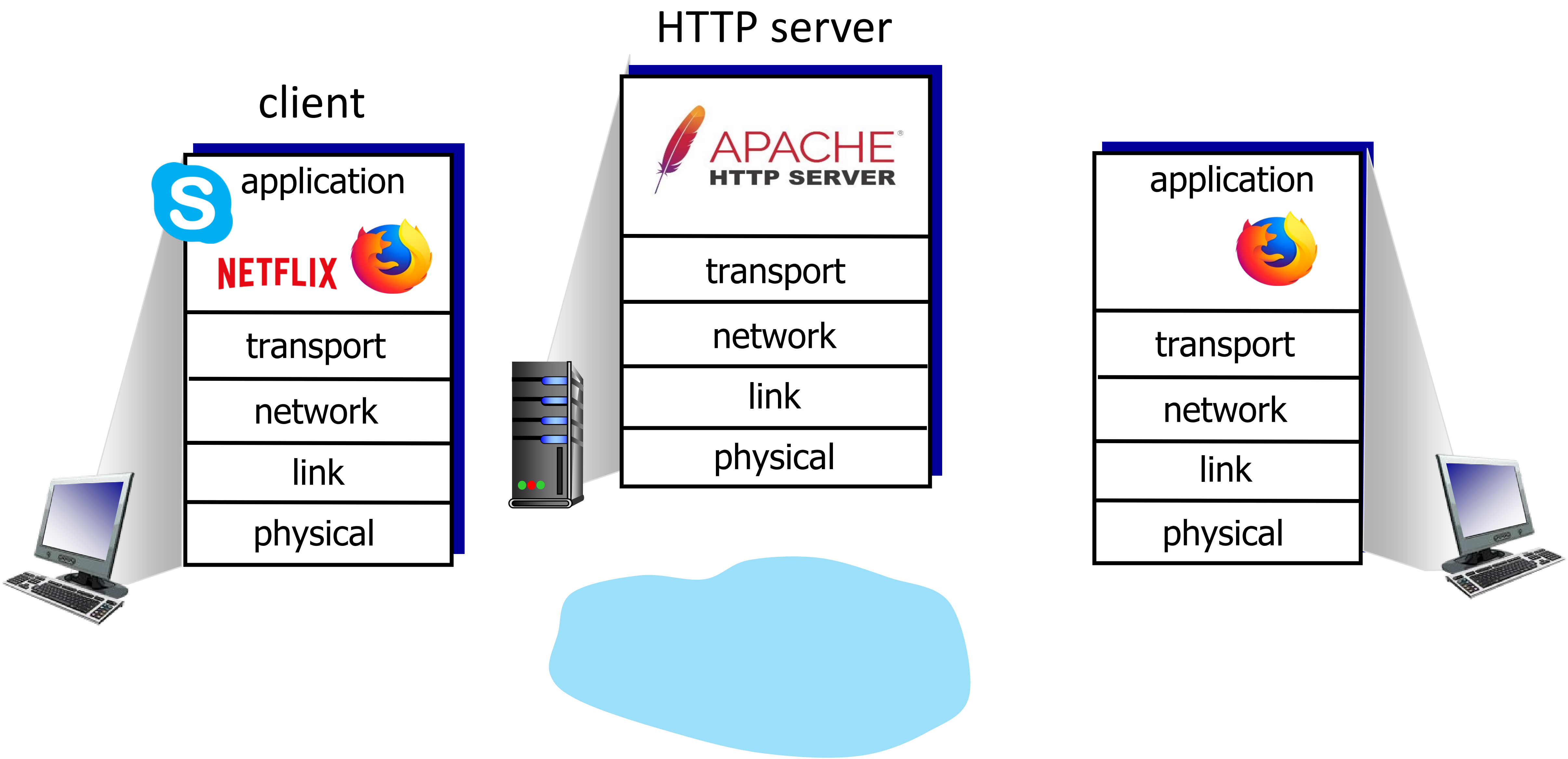
Two principal Internet transport protocols

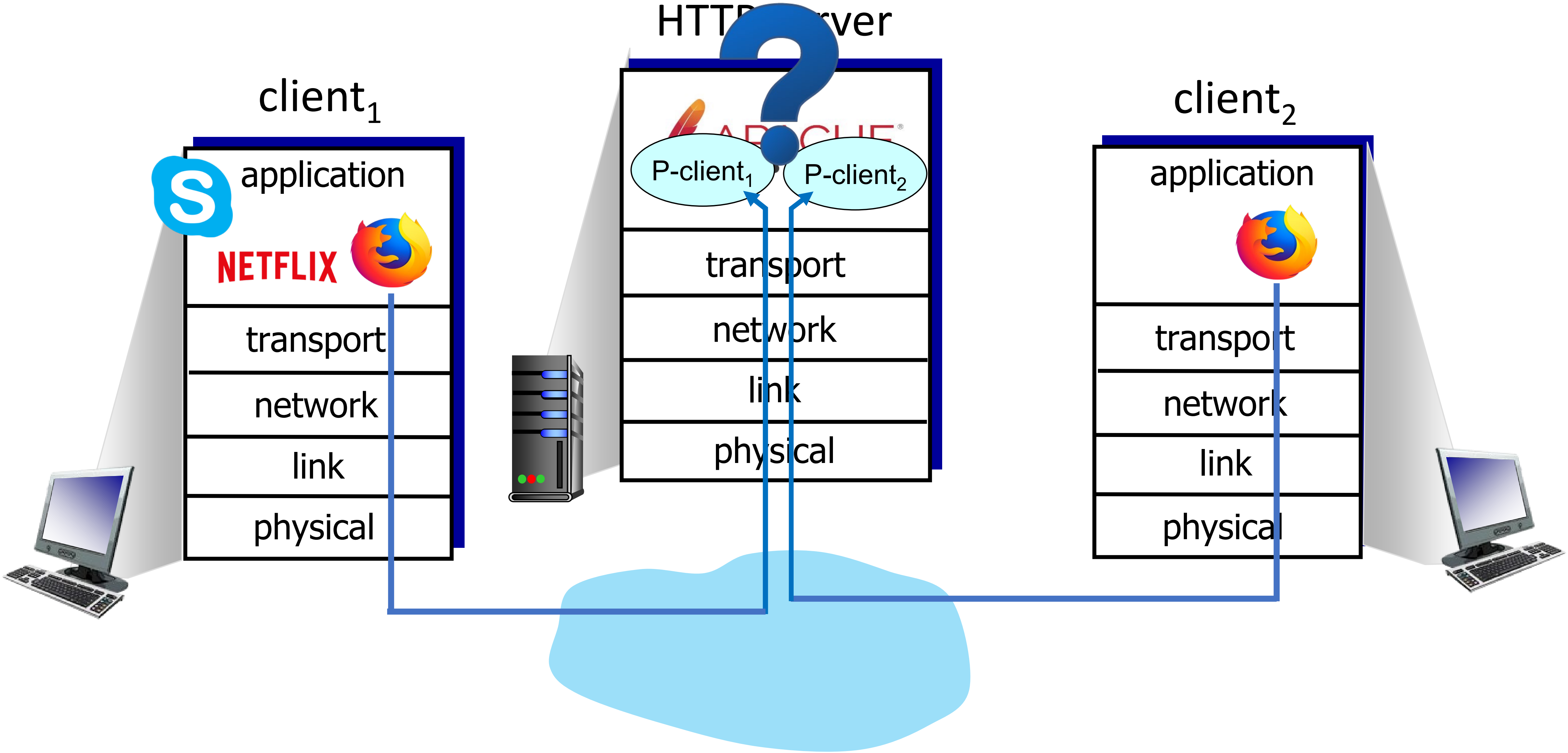
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Roadmap

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


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



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

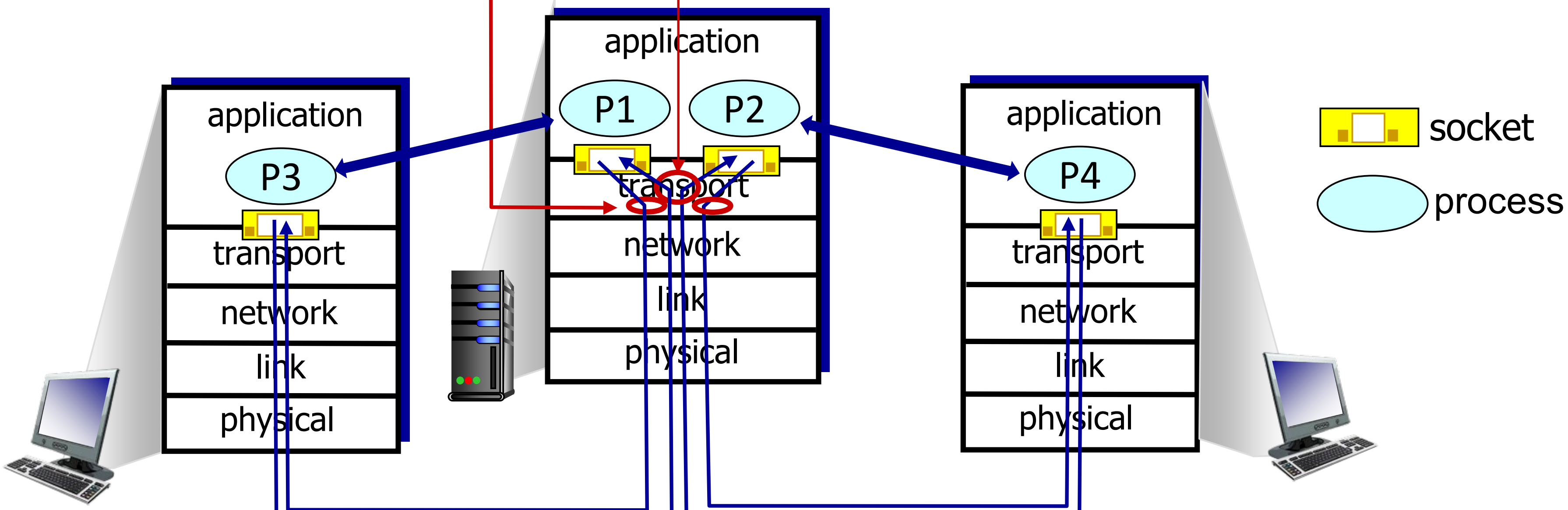
Multiplexing/demultiplexing

multiplexing at sender:

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

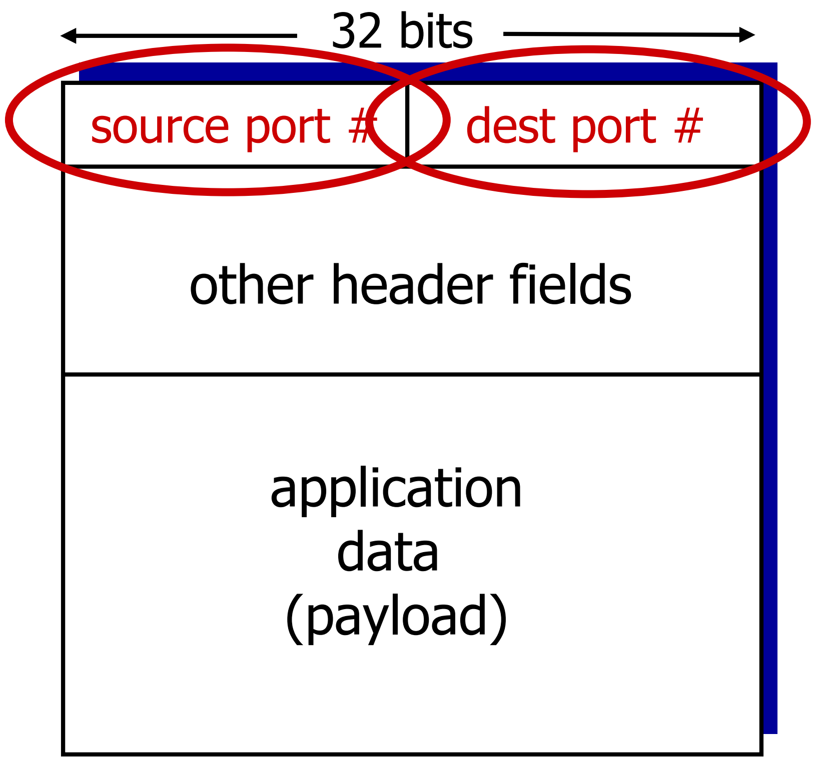
demultiplexing at receiver:

use header info to deliver received segments to correct socket



Connectionless demultiplexing

- 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



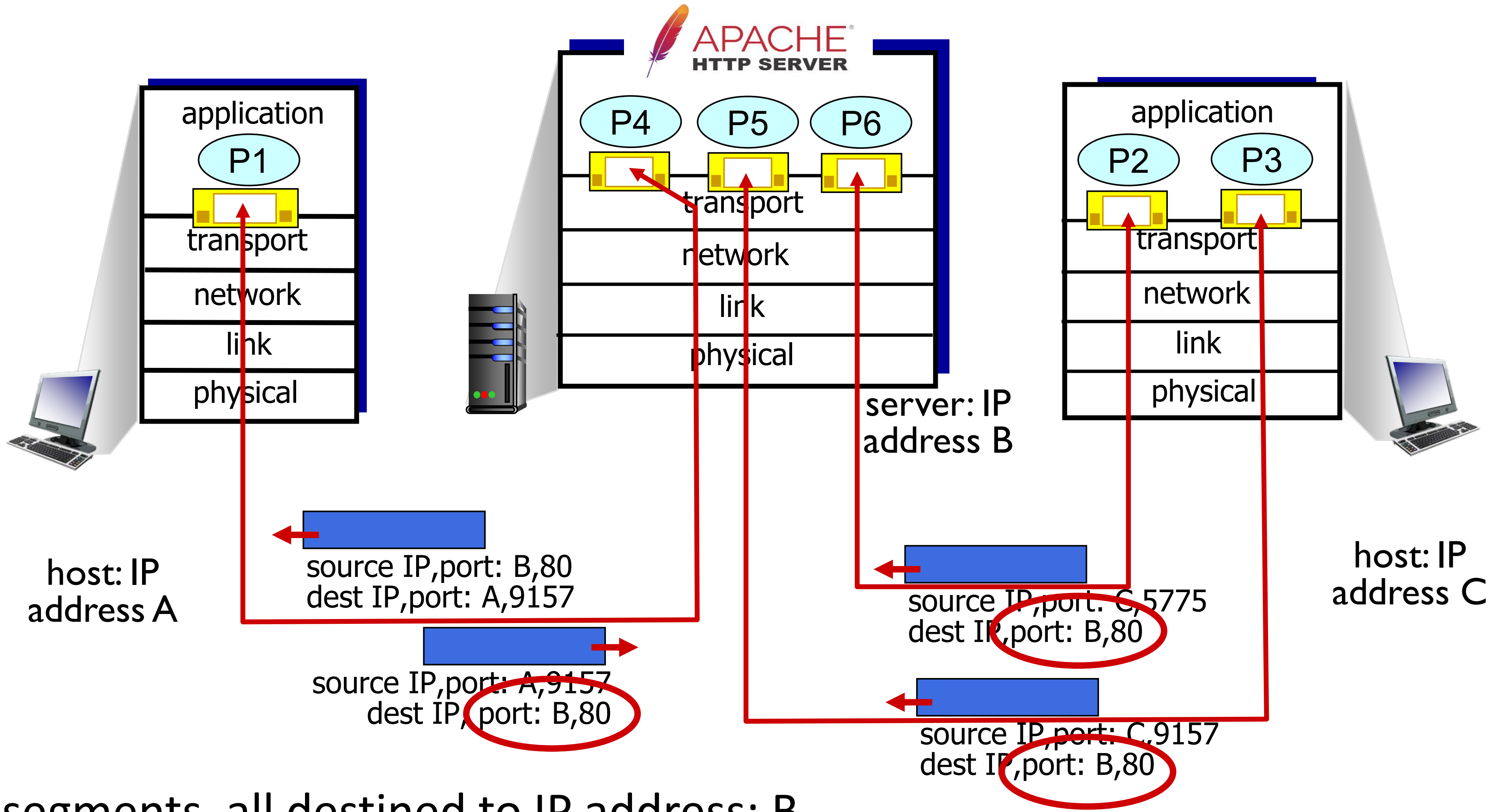
TCP/UDP segment format

Connection-oriented demultiplexing

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

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

Connection-oriented demultiplexing: example



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

Summary

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

Roadmap

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

UDP: User Datagram Protocol [RFC 768]

- "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

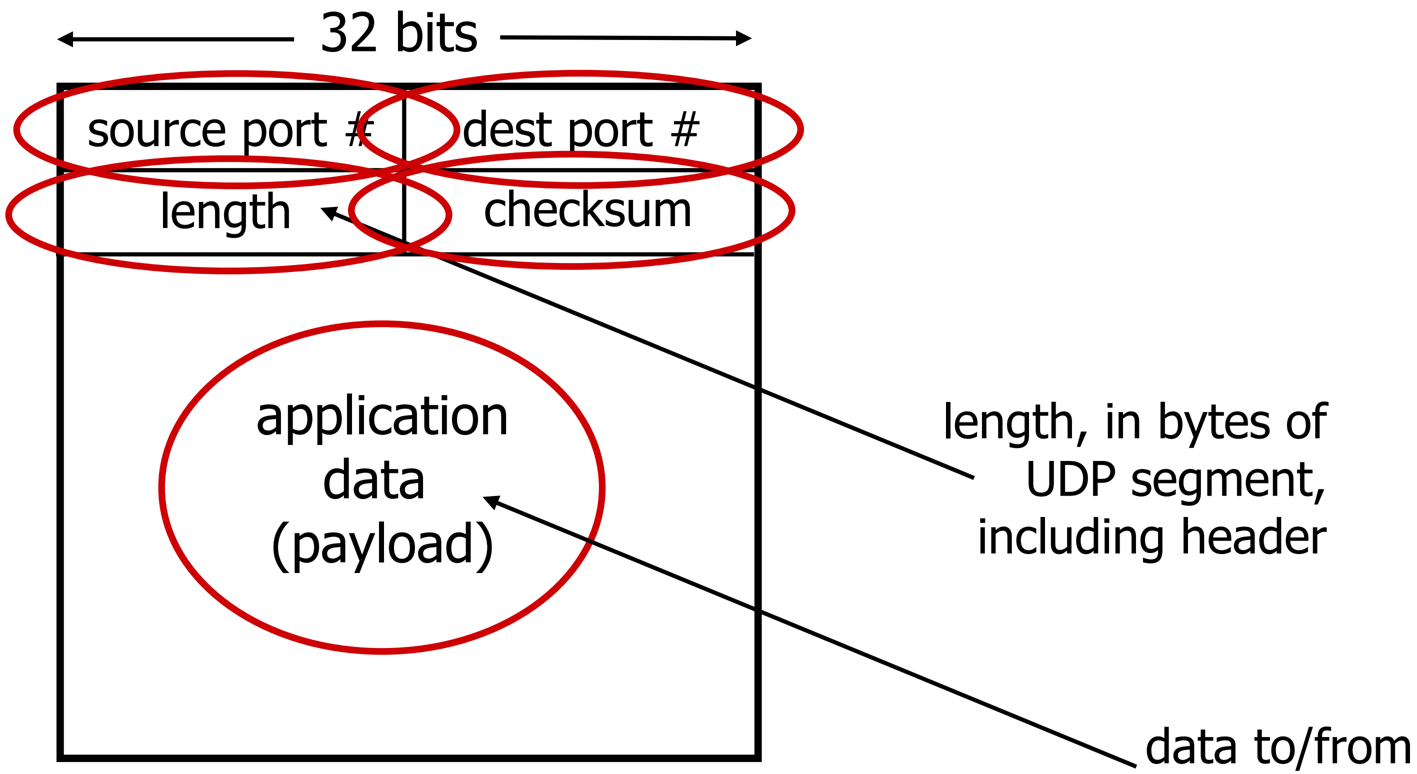
Why is there a UDP?

- No connection
  - which can add RTT delay
- Simplicity
  - no connection state at sender, receiver
- Small header size
- No flow/congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

UDP: User Datagram Protocol [RFC 768]

- 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 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)

!

Internet 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)

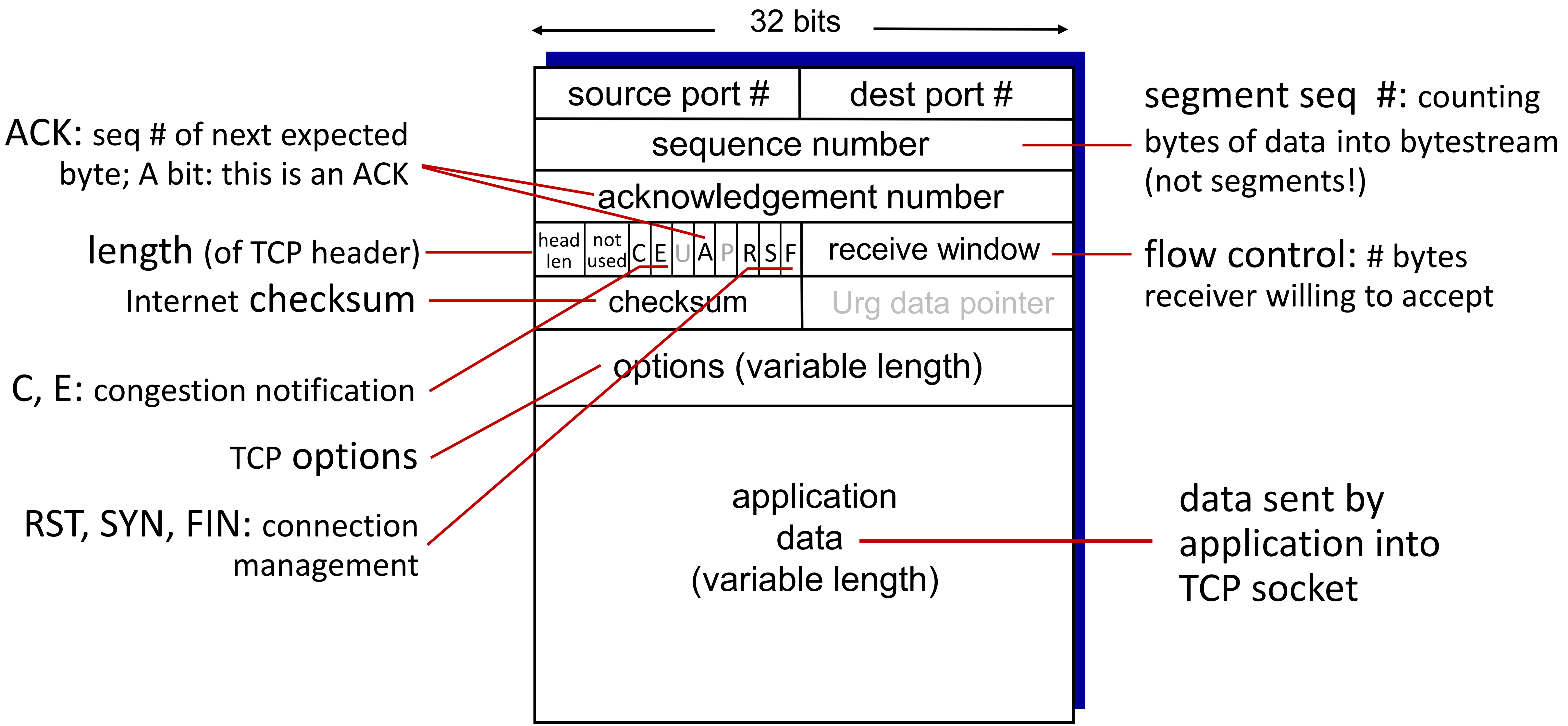
Roadmap

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



TCP sequence numbers, ACKs

Sequence numbers:

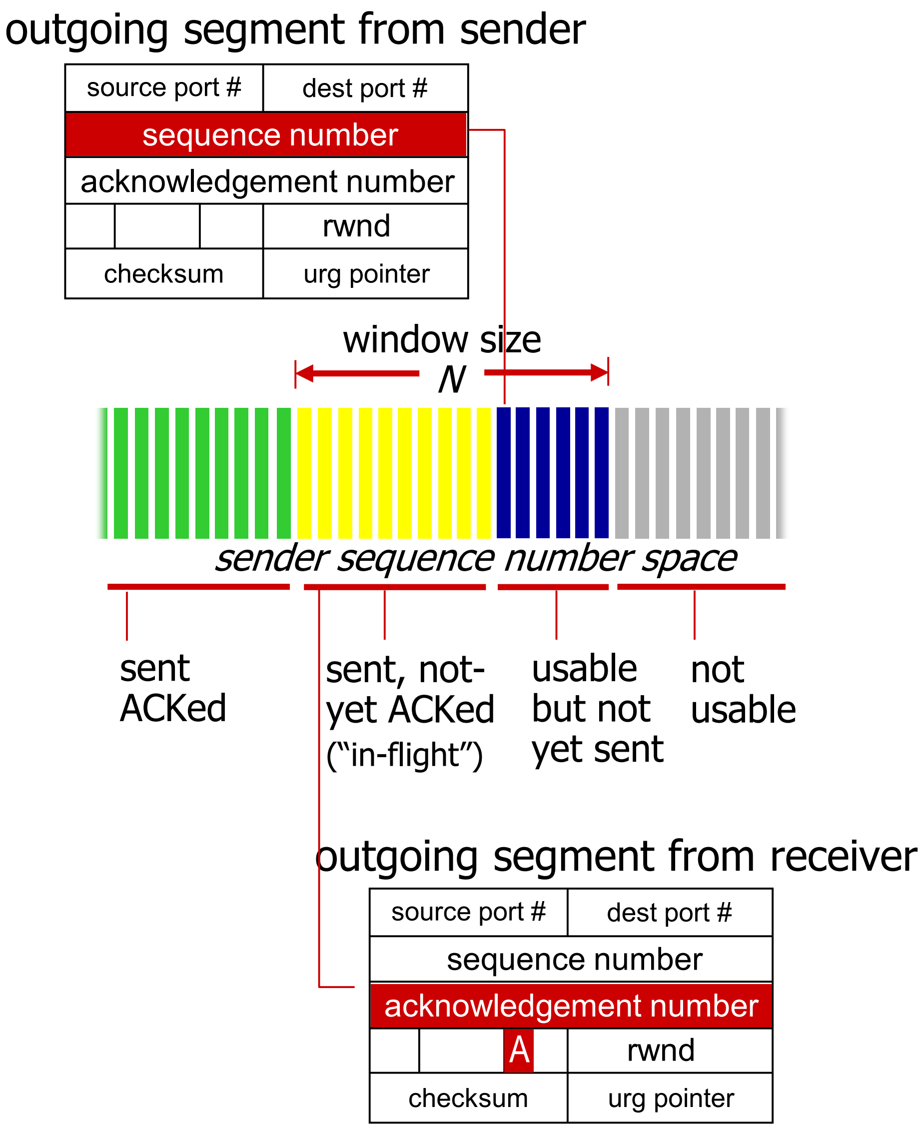
- byte stream "number" of first byte in segment's data

Acknowledgements:

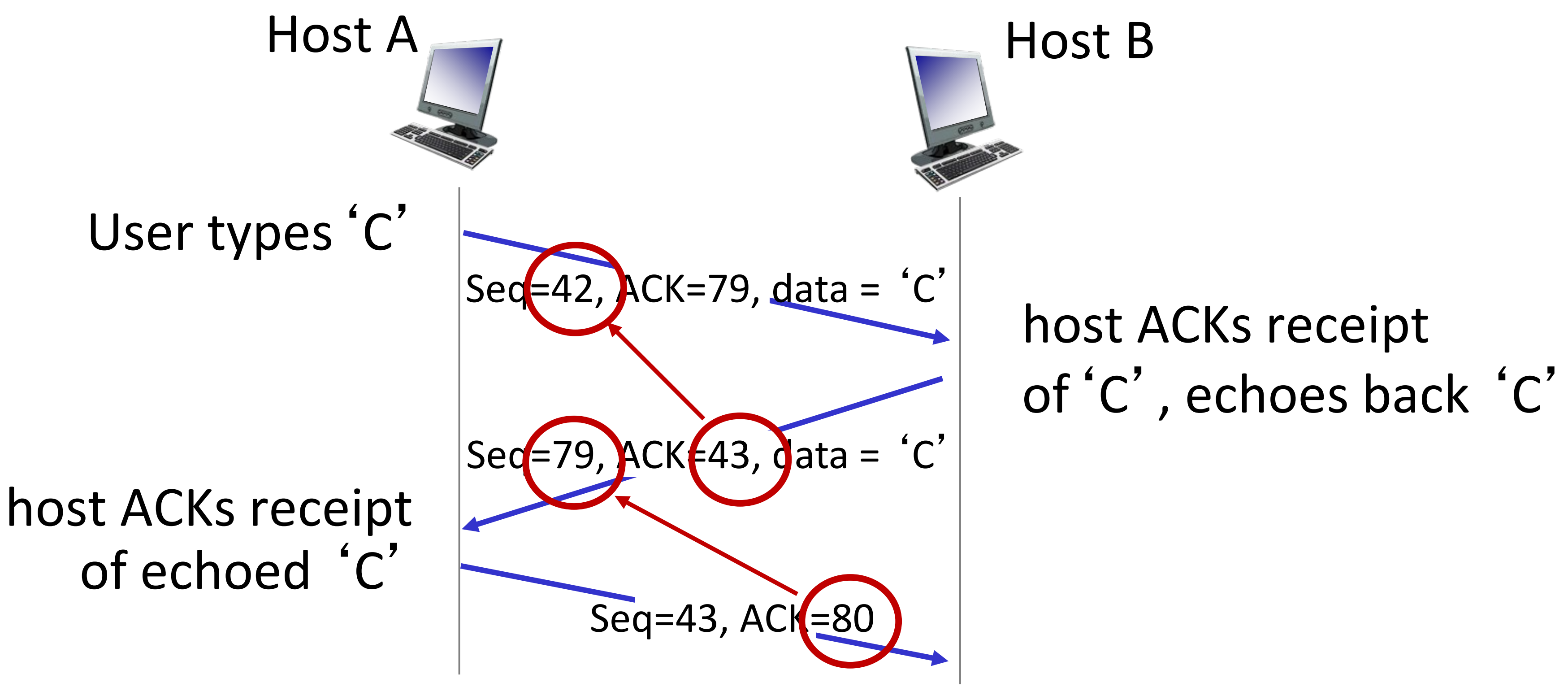
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs

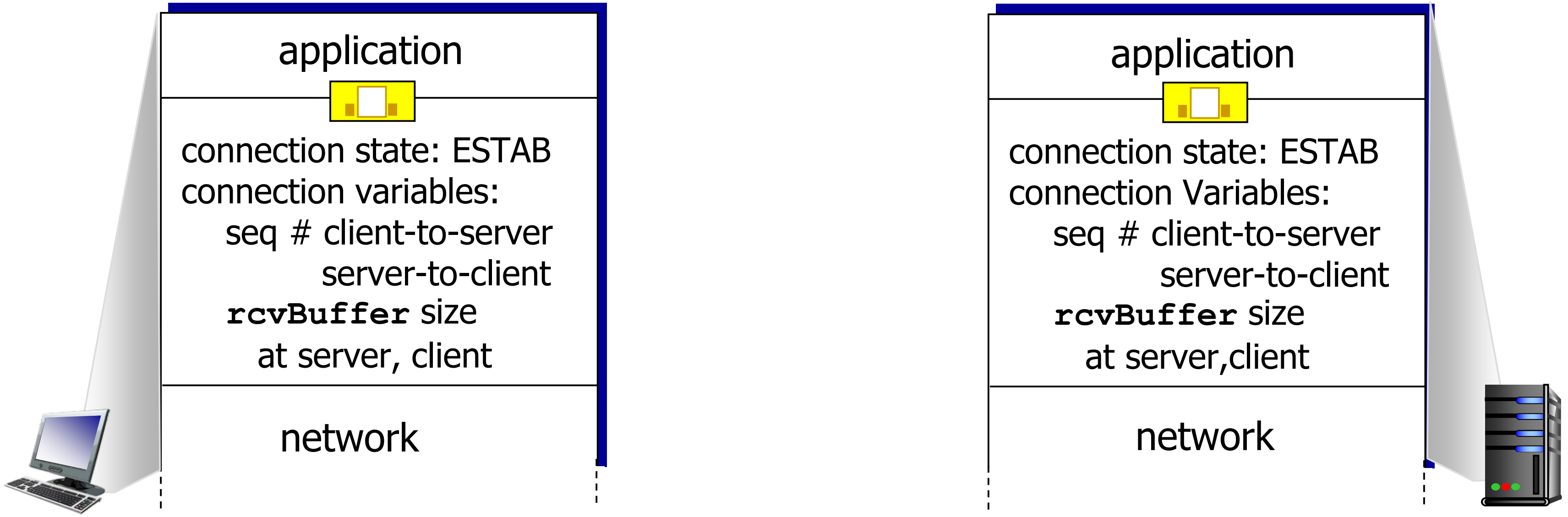


simple telnet scenario

TCP 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)




```
int connectStatus =
connect(sockD, (struct sockaddr*)&servAddr,sizeof(servAddr));
```

```
int clientSocket =
accept(servSockD, NULL, NULL);
```

TCP 3-way handshake

Client state

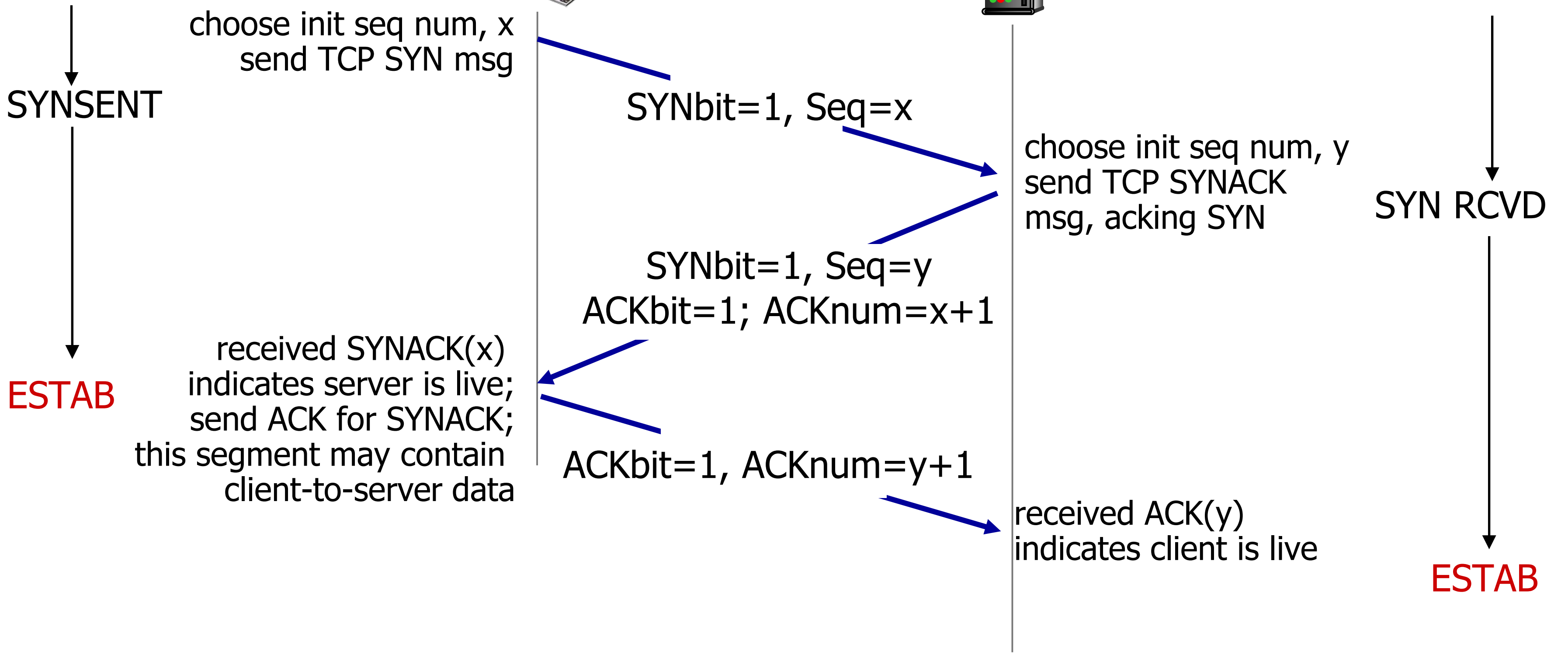
```
int sockD = socket(AF_INET, SOCK_STREAM, 0);
```

LISTEN

```
int connectStatus =
connect(sockD, (struct
sockaddr*)&servAddr, sizeof(servAddr));
```

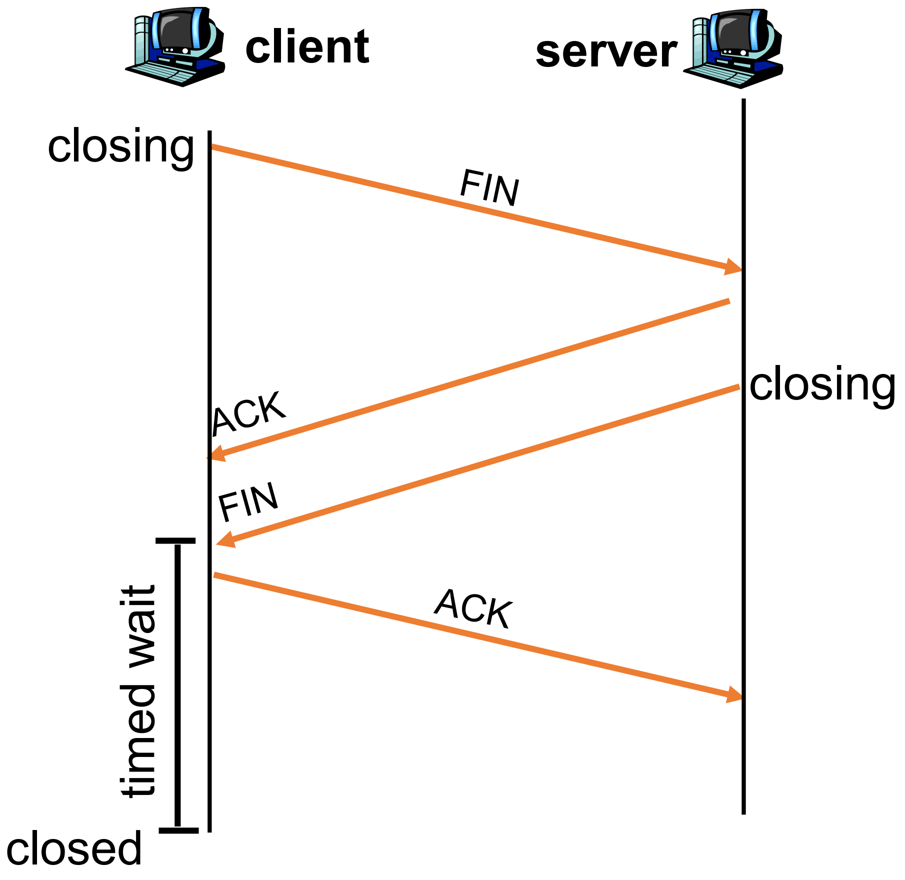
Server state

```
int servSockD = socket(AF_INET, SOCK_STREAM, 0);
bind(servSockD, (struct sockaddr*)&servAddr, sizeof(servAddr));
listen(servSockD, 1);
int clientSocket = accept(servSockD, NULL, NULL);
```



Closing a TCP connection

1. client end system sends TCP FIN control segment to server
2. server receives FIN, replies with ACK. Closes connection, sends FIN
3. client receives FIN, replies with ACK
   - Enters “timed wait” - will respond with ACK to received FINs
4. server, receives ACK. Connection closed

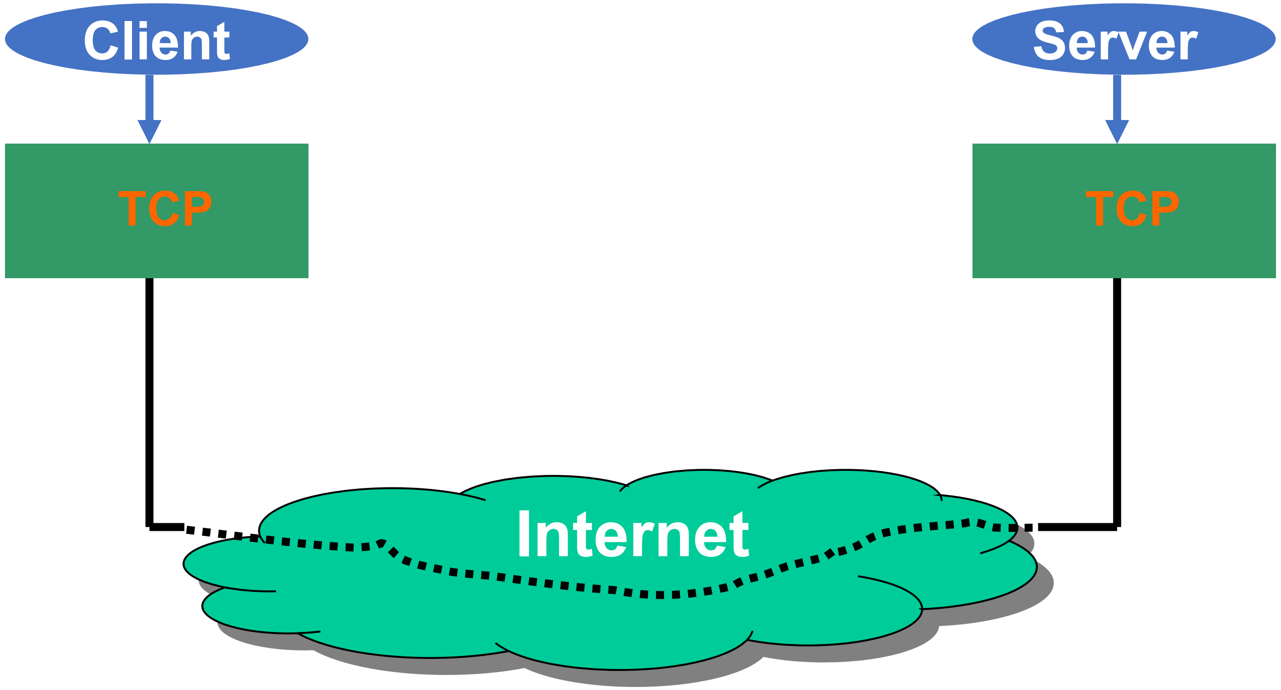


Roadmap

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

TCP Reliable Data Transfer

- TCP creates rdt service
  - on top of IP's unreliable service
- Window-based ARQ scheme
  - Acknowledgements
  - Timeouts
  - Retransmissions



How to set TCP timeout value?

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`

RTT Estimate

$SampleRTT \coloneqq RTT$
$EstimatedRTT \coloneqq ERTT$

$ERTT_1 = RTT_0$
$ERTT_2 = \alpha \cdot RTT_1 + (1 - \alpha) \cdot RTT_0$
$ERTT_3 = \alpha \cdot RTT_2 + \alpha(1 - \alpha) \cdot RTT_1 + (1 - \alpha)^2 \cdot RTT_0$
.....
$ERTT_{n+1}$
$= \alpha \cdot RTT_n + \alpha(1 - \alpha) \cdot RTT_{n-1}$
$+ \alpha(1 - \alpha)^2 \cdot RTT_{n-2} + \cdots + (1 - \alpha)^n \cdot RTT_0$

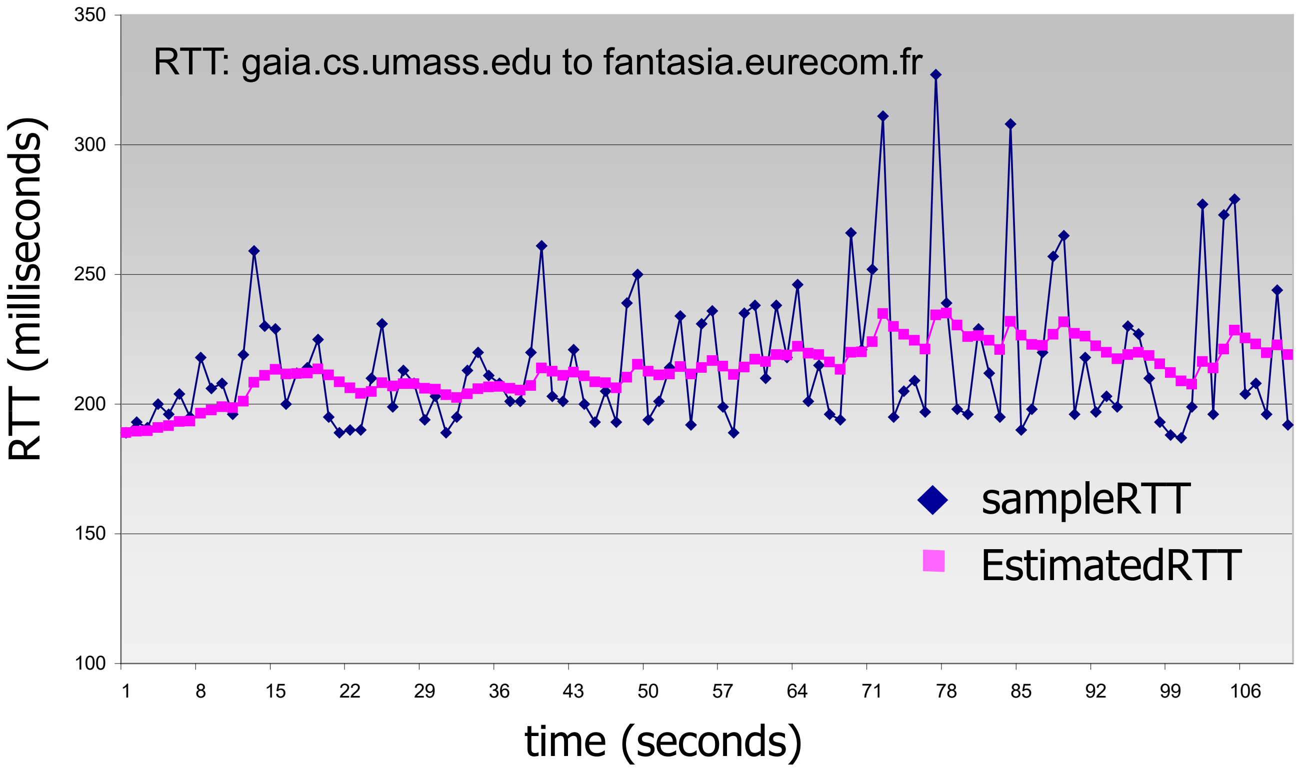
$\alpha < 1$

$$ERTT_{n+1} = \alpha \cdot RTT_n + (1 - \alpha) \cdot [\alpha \cdot RTT_{n-1} + \alpha(1 - \alpha) \cdot RTT_{n-2} + \cdots + (1 - \alpha)^{n-1} \cdot RTT_0]$$
$$ERTT_{n+1} = \alpha \cdot RTT_n + (1 - \alpha) \cdot ERTT_n$$

TCP round trip time, timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT:** want a larger safety margin

$$\texttt{TimeoutInterval = EstimatedRTT + 4*DevRTT}$$

estimated RTT "safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

$$\texttt{DevRTT} = (1-\beta)\texttt{*DevRTT} + \beta\texttt{*|SampleRTT-EstimatedRTT|}$$

(typically, $\beta = 0.25$)

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

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

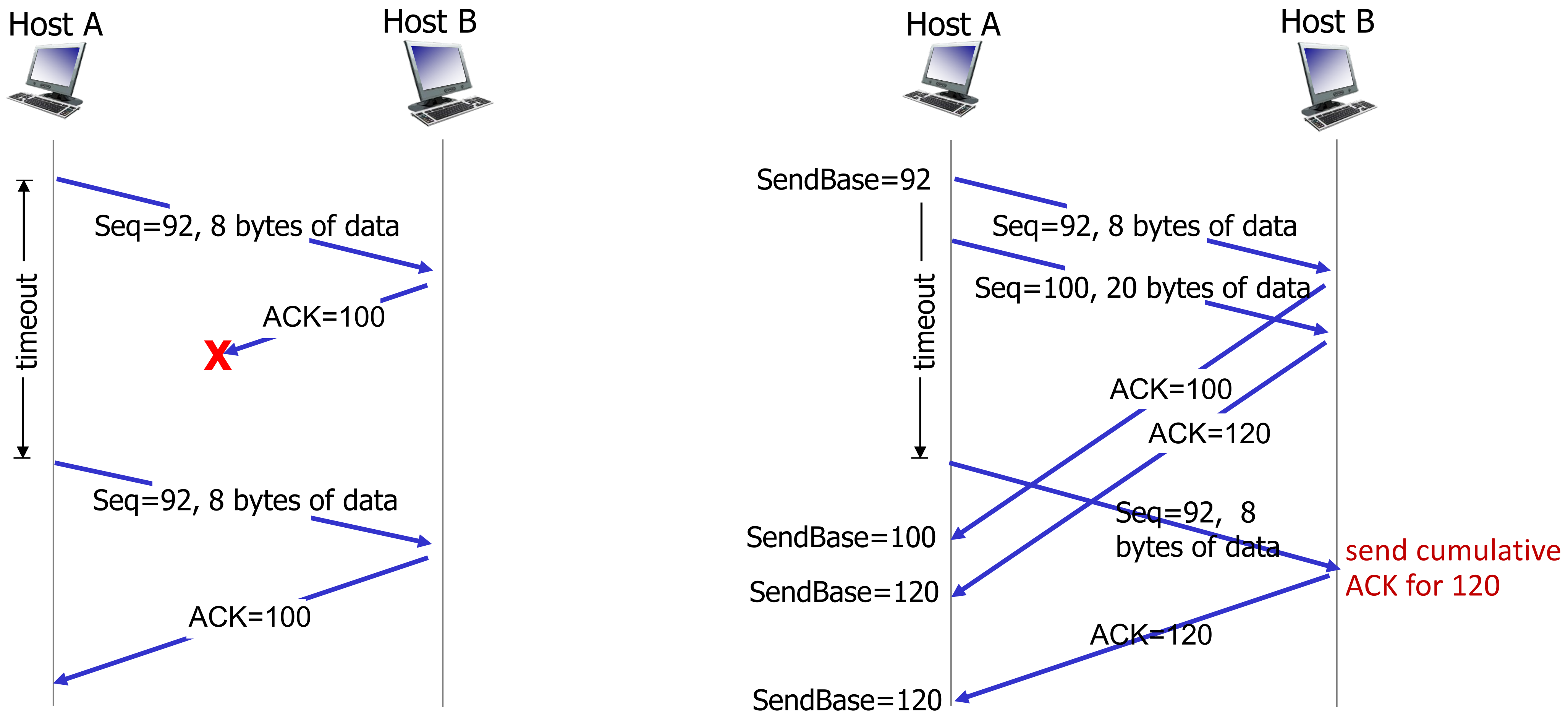
TCP Sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

   event: data received from application above
         create TCP segment with sequence number NextSeqNum
         if (timer currently not running) start timer
         pass segment to IP
         NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
         retransmit not-yet-acknowledged segment with smallest sequence number
         start timer

    event: ACK received, with ACK field value of y
         if (y > SendBase) {
              SendBase = y
             if (there are currently not-yet-acknowledged segments) start timer
          }

  }  /* end of loop forever */
```

TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*

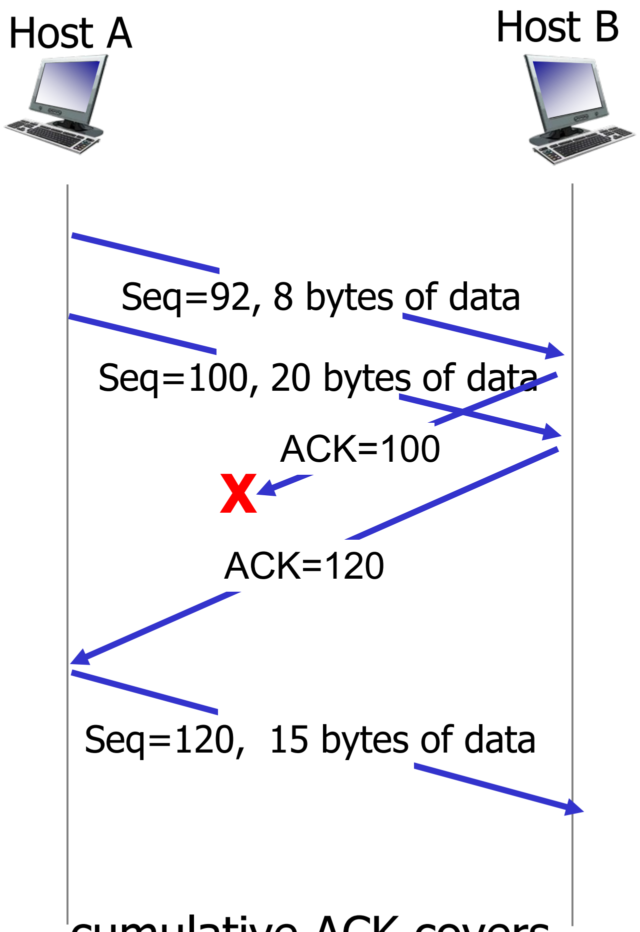
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



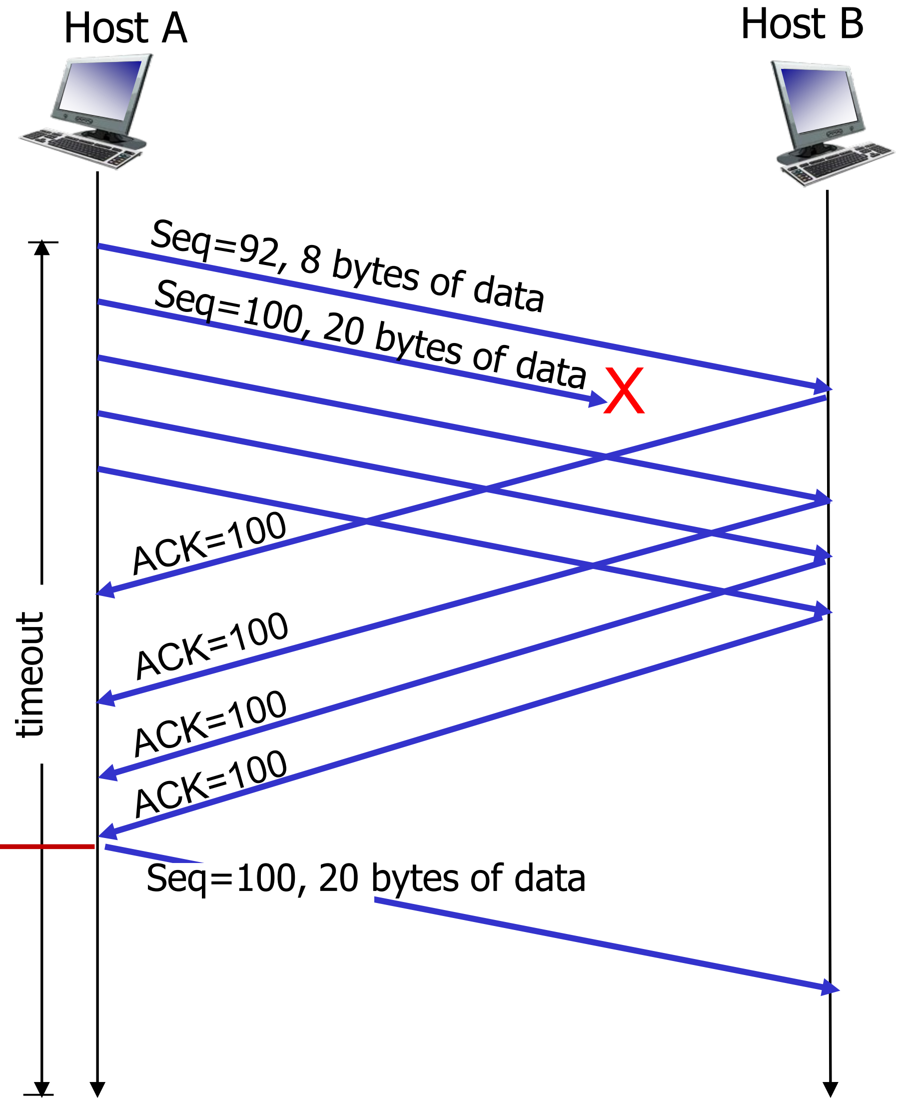
cumulative ACK covers for earlier lost ACK

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



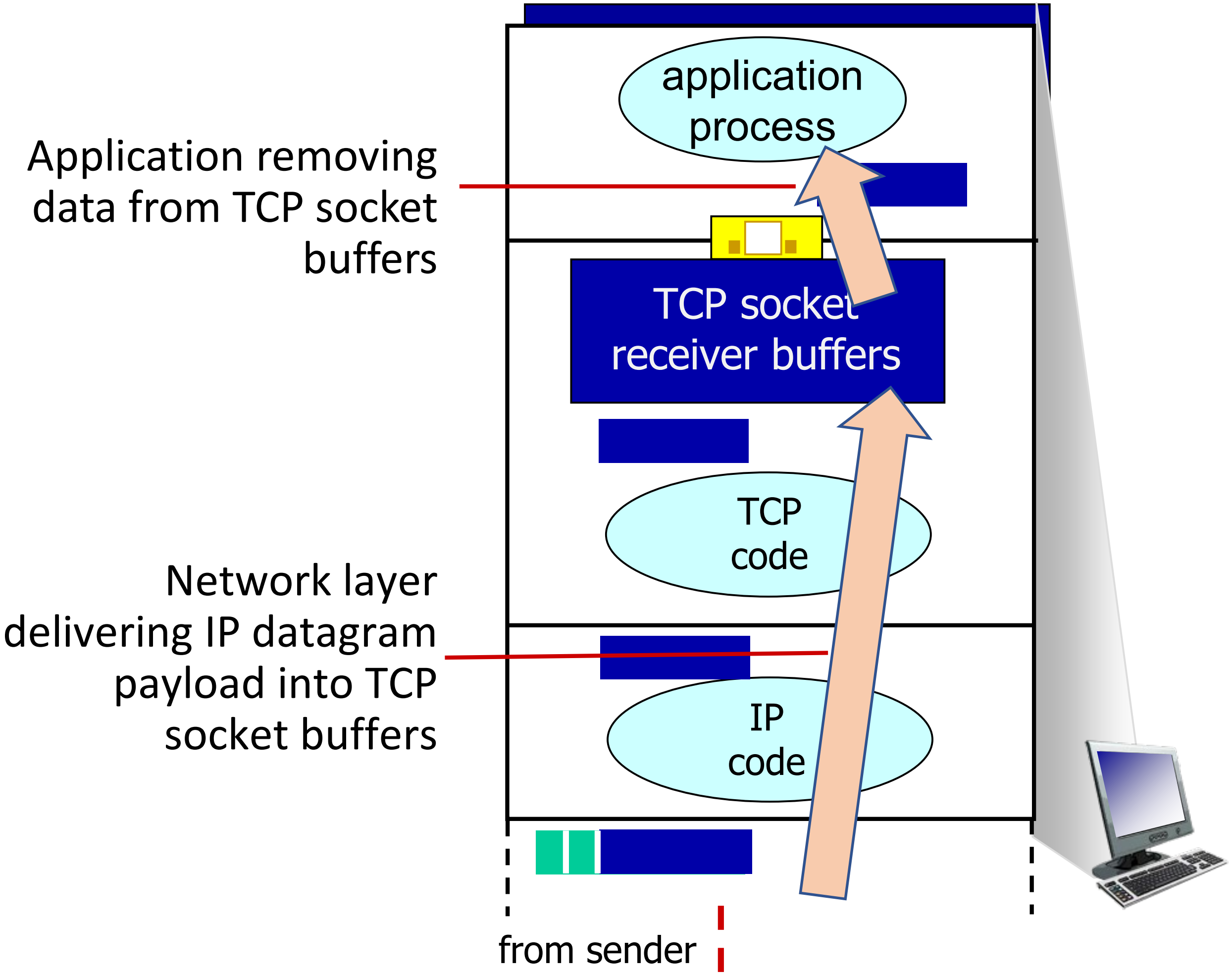
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

Roadmap

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

TCP flow control

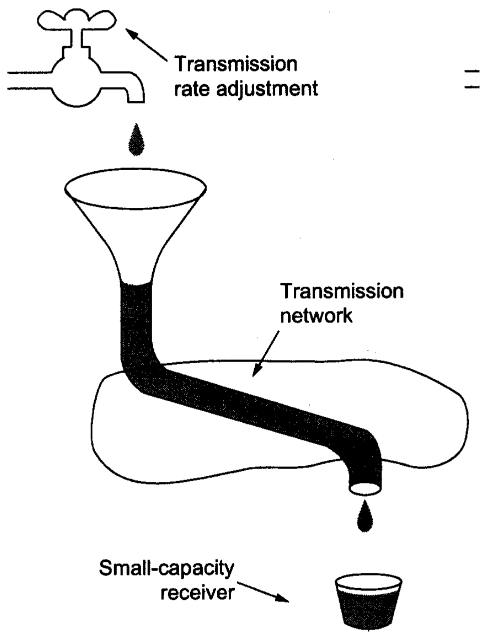
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

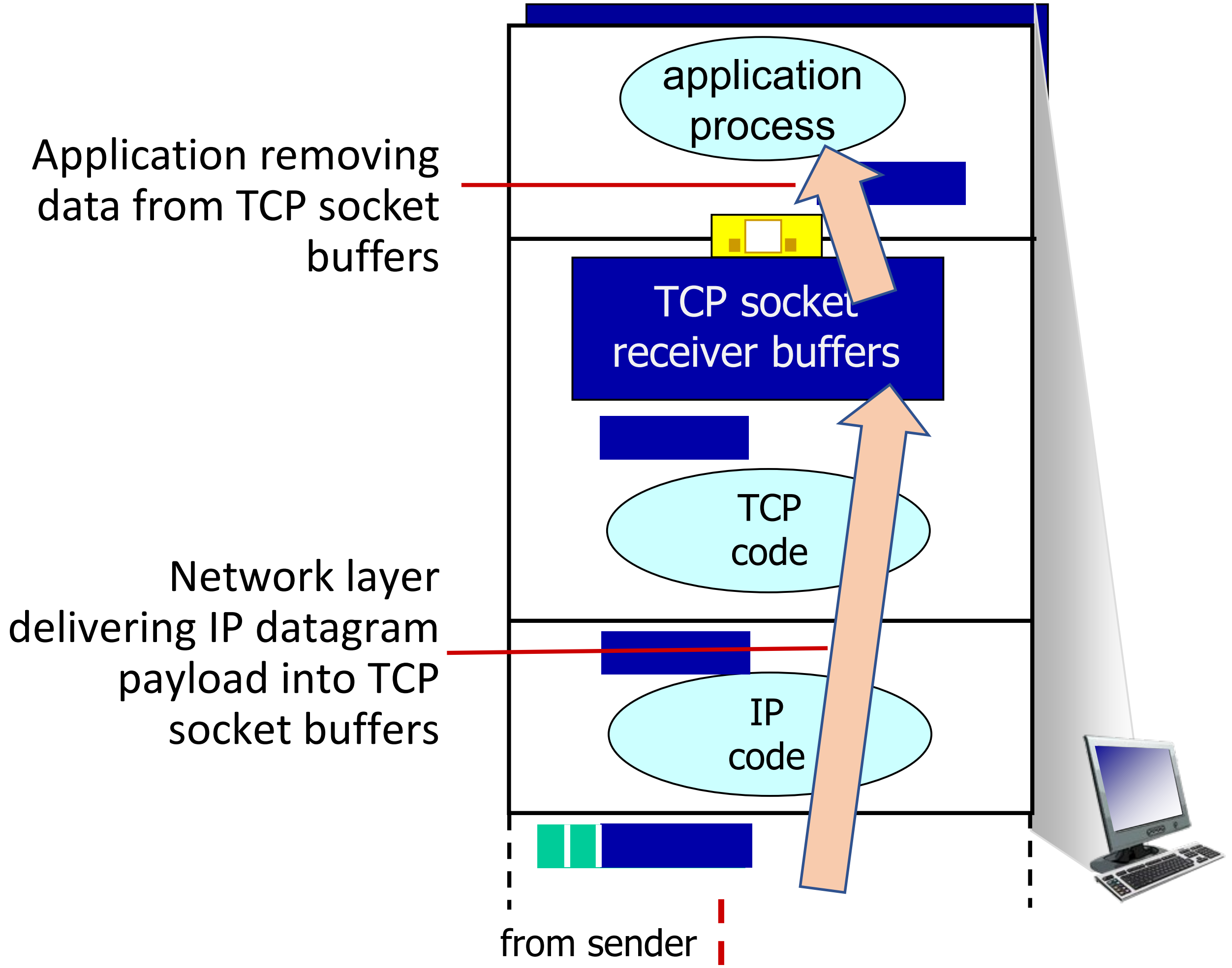


receiver protocol stack

TCP flow control

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





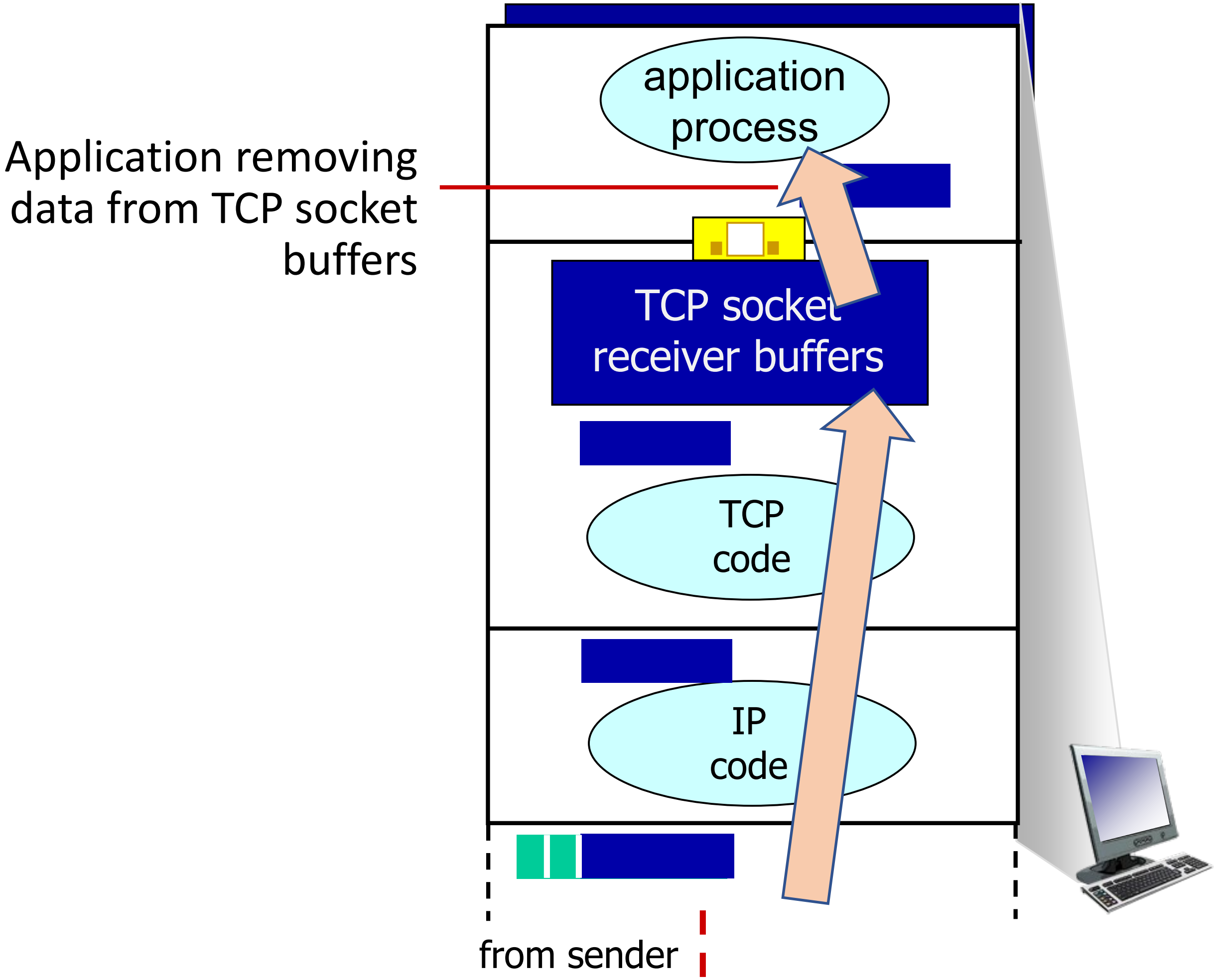
receiver protocol stack

TCP flow control

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

flow control

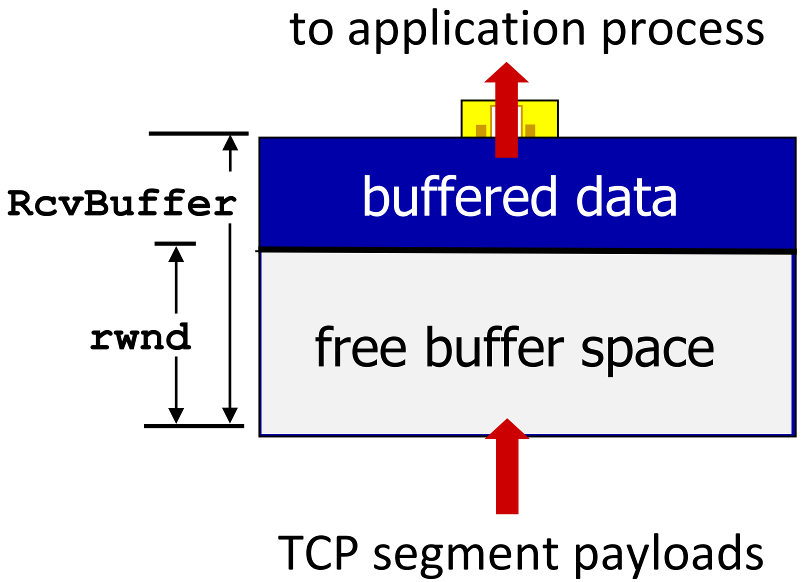
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



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

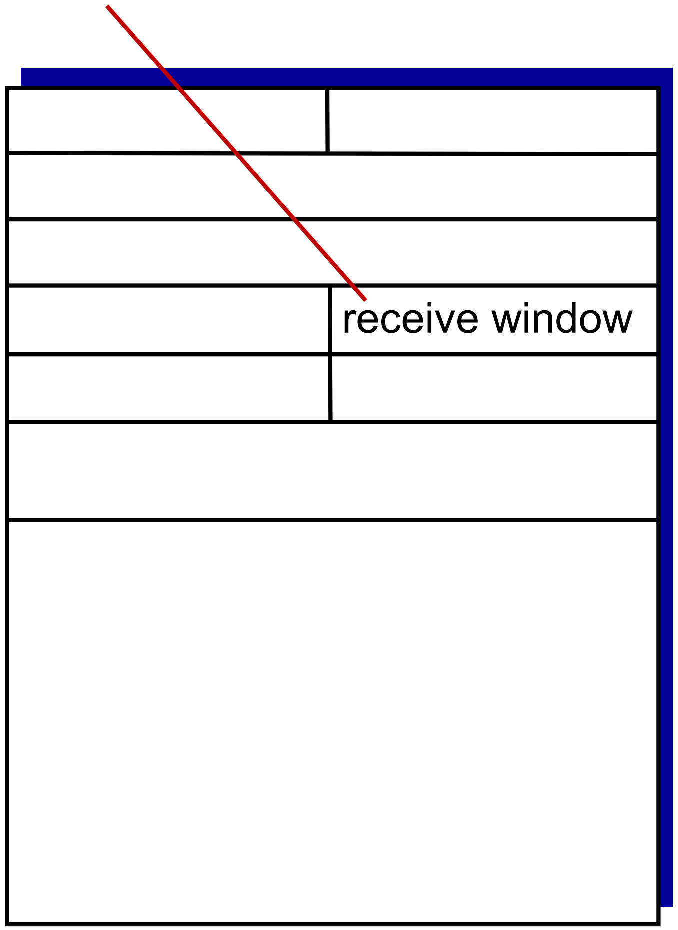


TCP receiver-side buffering

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)
  - 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

Roadmap

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

Principles of congestion control

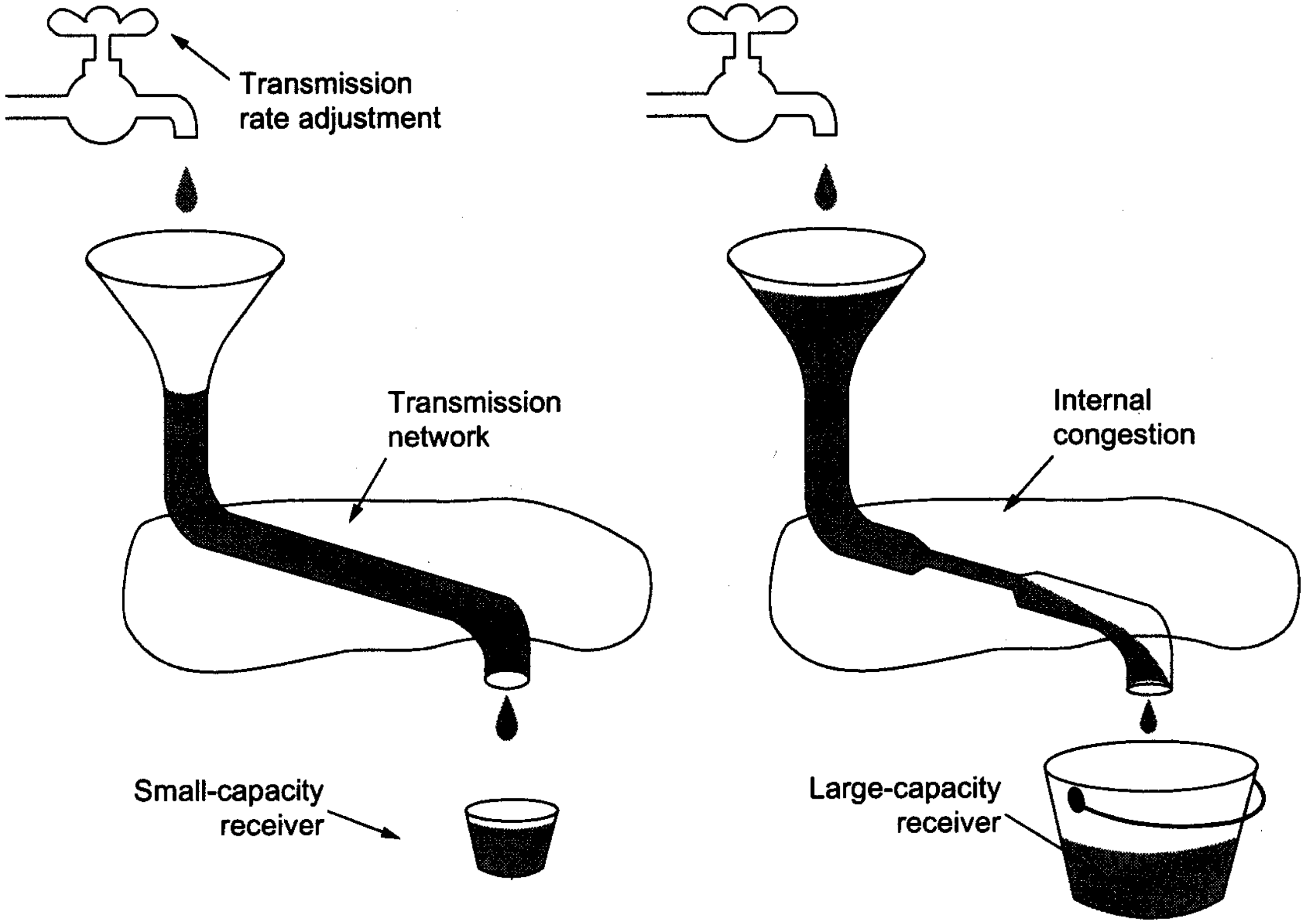
Congestion:

- 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)
- a top-10 problem!

congestion

too many senders, sending too fast

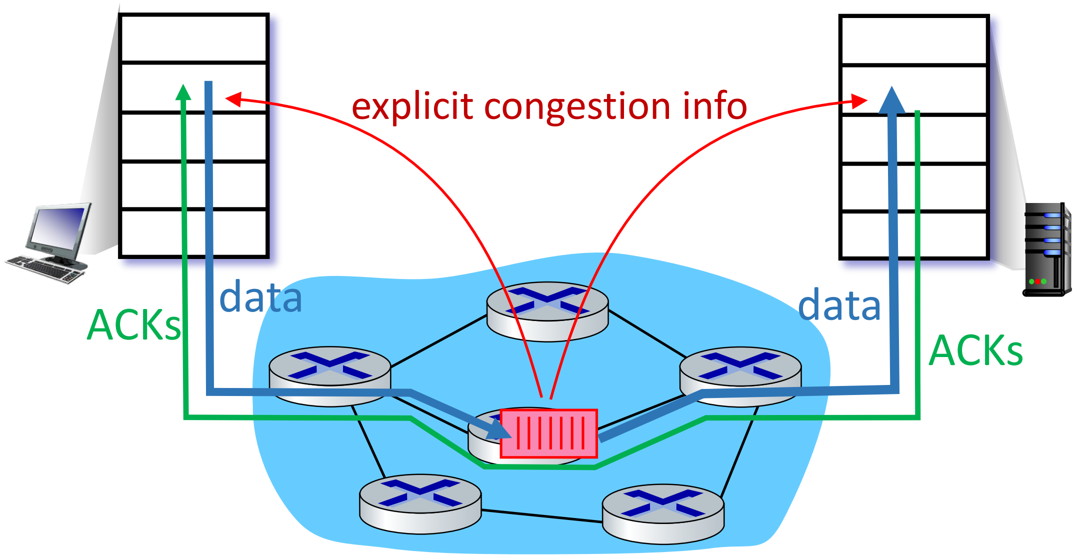
Flow Control vs. Congestion Control



Approaches towards congestion control

Network-assisted congestion control:

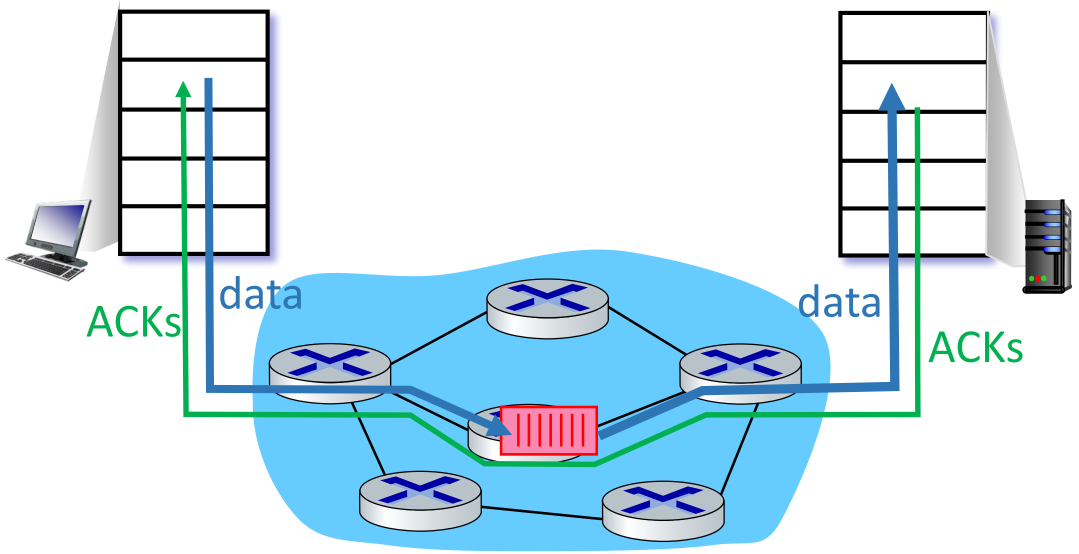
- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate
- ATM, DECbit, TCP ECN protocols



Approaches towards congestion control

End-end congestion control:

- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



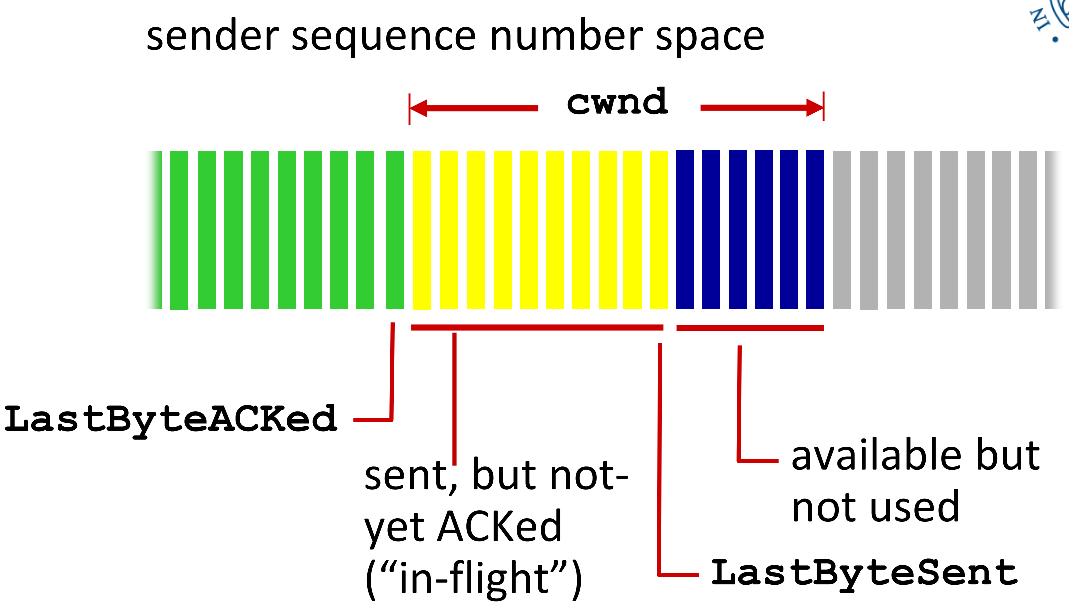
TCP congestion control

GOAL: TCP sender should transmit as fast as possible, but without congesting the network

Three Fundamental Questions

- How does the sender *limit* its rate, based on perceived congestion?
- How does the sender *perceive* congestion?
- How does the sender *adjust* the rate based on perceived congestion?

Sender Rate Adjustment



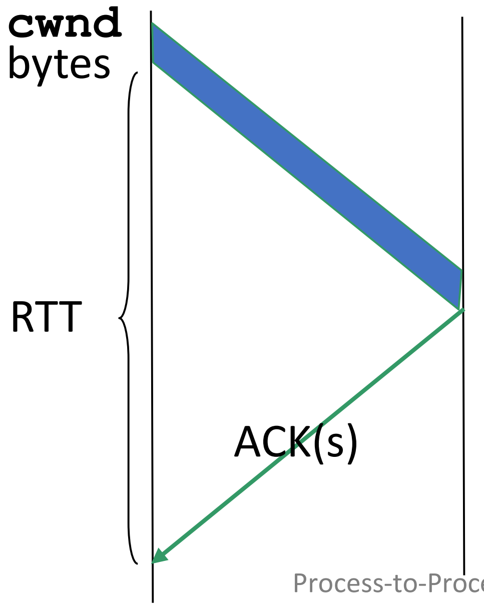
- Sender limits its rate by limiting number of unACKed bytes "in pipeline":

  LastByteSent-LastByteAcked ≤ cwnd

  - **cwnd** differs from **rwnd** (how, why?)
  - sender limited by **min(cwnd,rwnd)**

- roughly,

$$\boxed{\text{rate} = \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}}$$



- **cwnd** is dynamically adjusted, in response to perceived network congestion

How congestion is perceived?

Each TCP sender sets its own rate, based on *implicit* feedback

- *ACK:* segment received (a good thing!)
  - network not congested
  - increase sending rate
- *Lost segment*
  - Time-out
  - 3 duplicate acks
  - assume loss due to congested network, so decrease sending rate

Additive Increase, Multiplicative Decrease (AIMD)

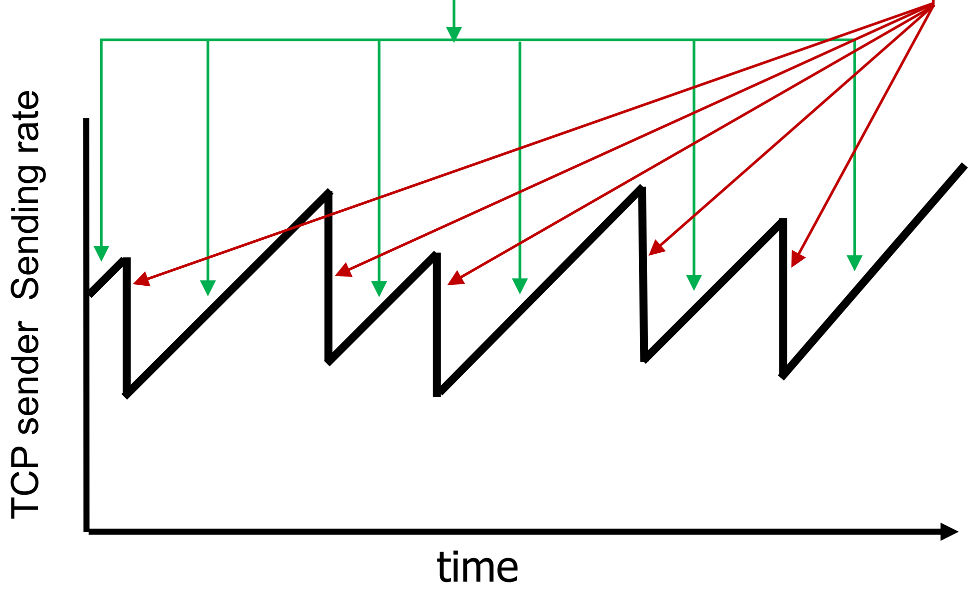
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

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

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP 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

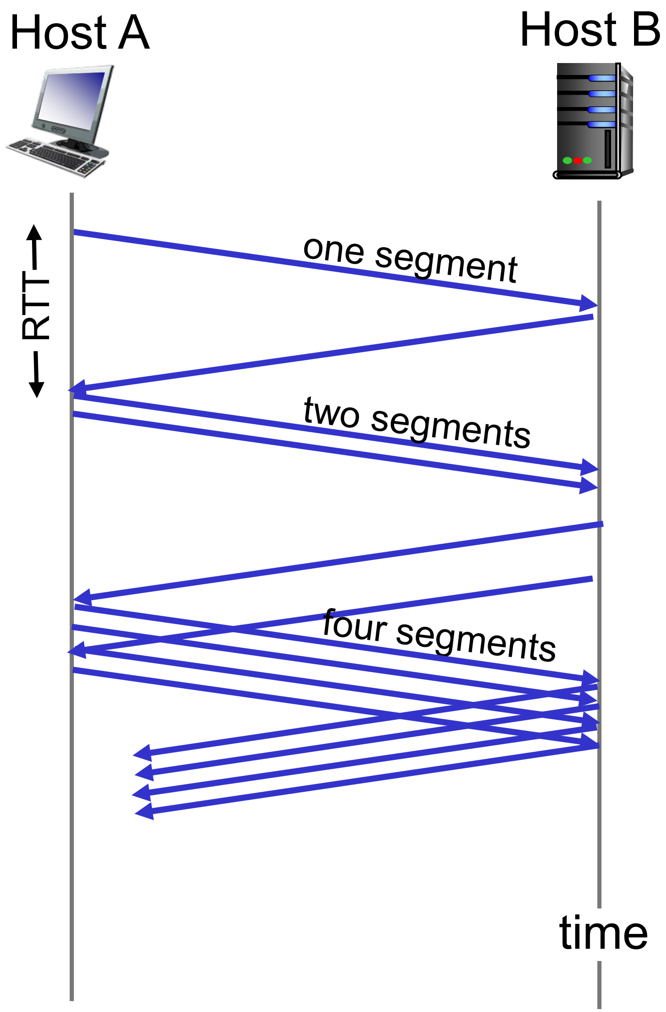
Phases

- Slow Start
- Congestion Avoidance
- Reaction to Loss Events

TCP slow start

- When connection begins, increase rate *exponentially* until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received

- *Summary:* initial rate is slow, but ramps up exponentially fast



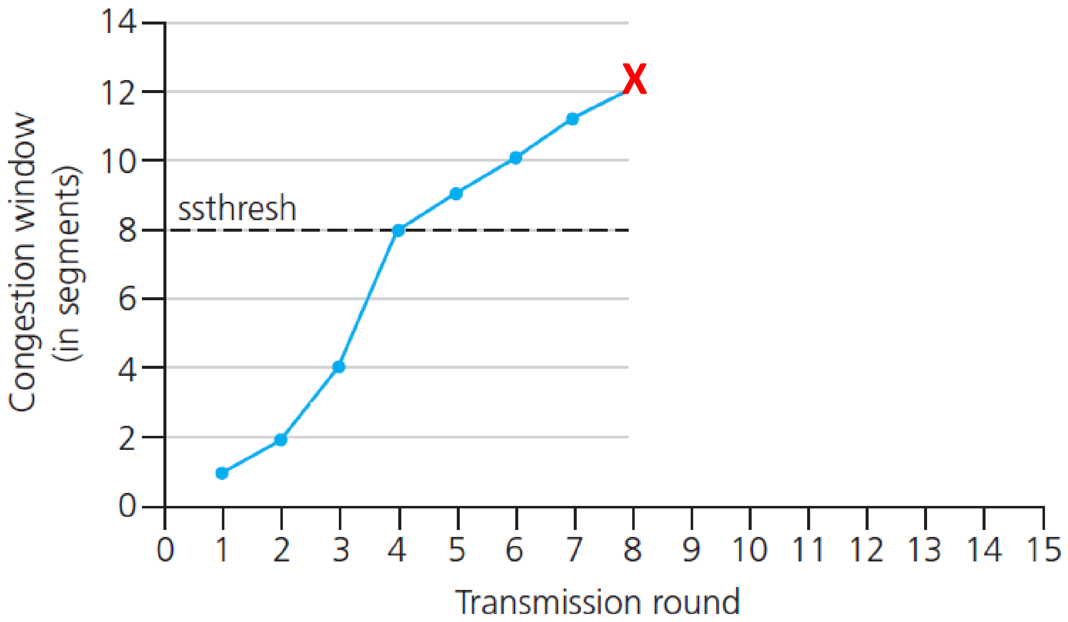
From Slow Start to Congestion Avoidance

Q: when should the exponential increase switch to linear?

A: when **`cwnd`** gets the threshold value (**`ssthresh`**)

Implementation:

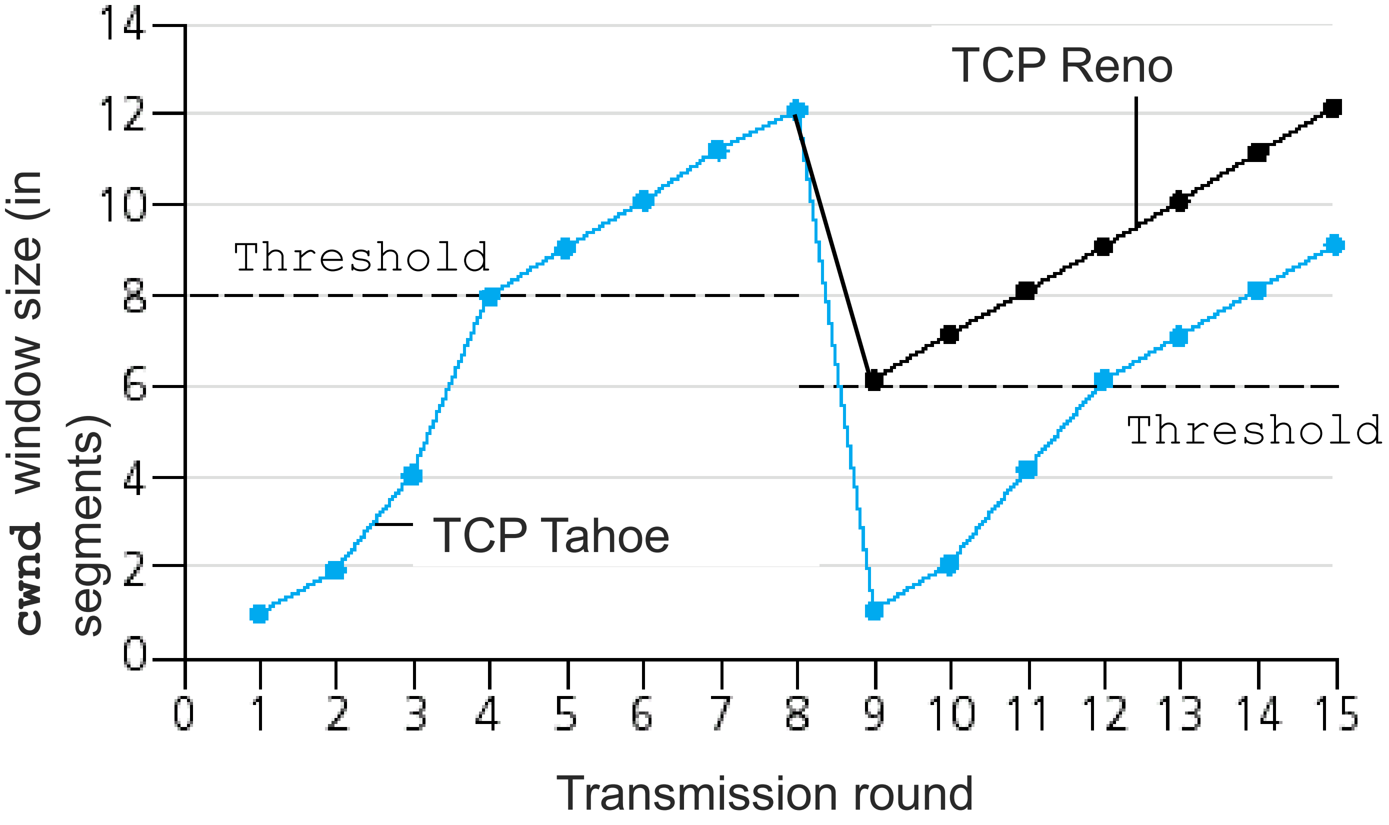
- Variable **`ssthresh`**
- Initially, **`ssthresh=64 KB`**
- Adjusted on loss events



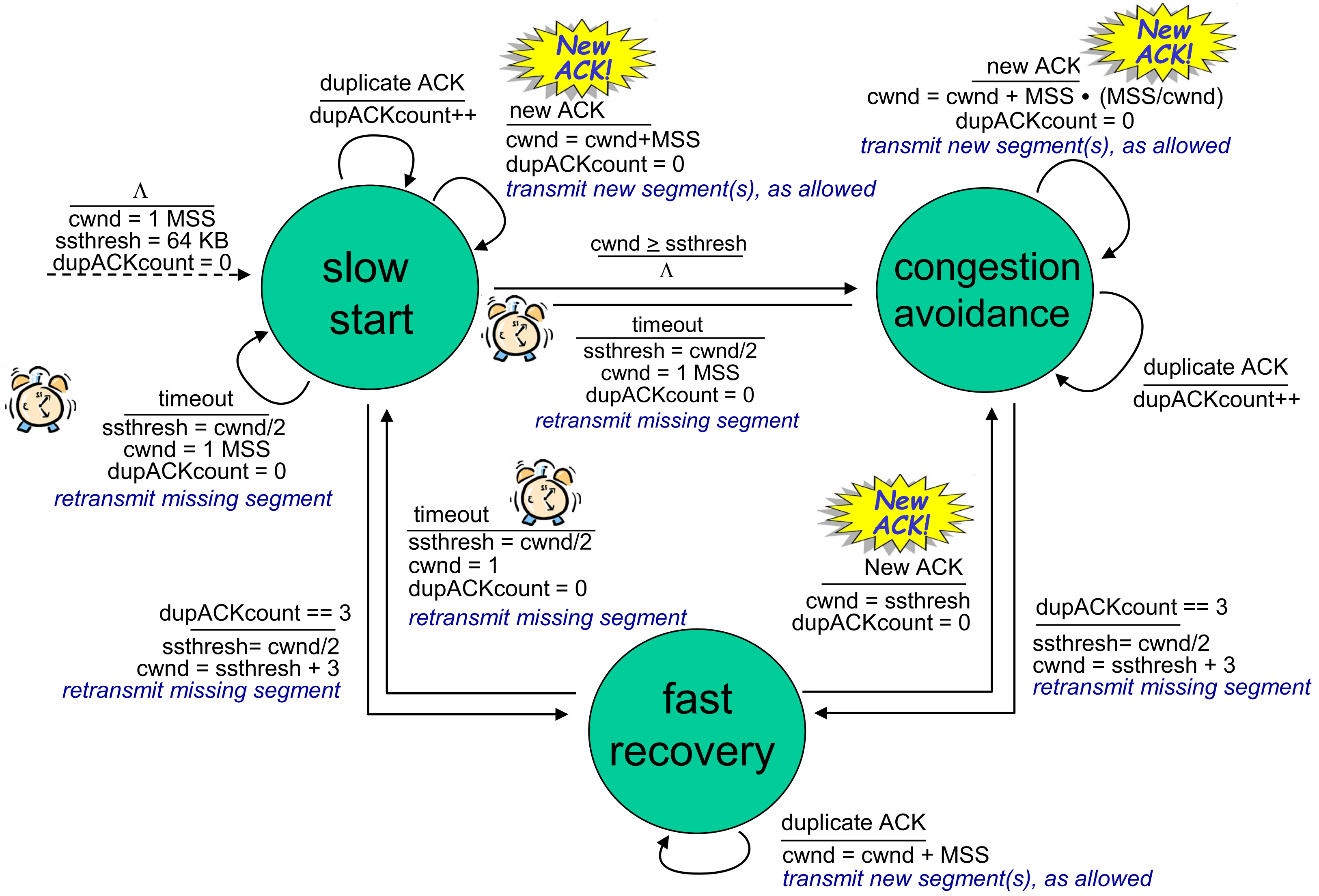
Reaction to loss events

- 3 Duplicate ACKs
  - **`ssthres=cwnd/2`**
  - **`cwnd=cwnd/2 + 3 MSS`**
  - Go to Fast Recovery (**`cwnd`** increases linearly)
- Timeout
  - **`ssthres=cwnd/2`**
  - **`cwnd=1`**
  - Go to Slow Start (**`cwind`** increases exponetially)

Reaction to loss events

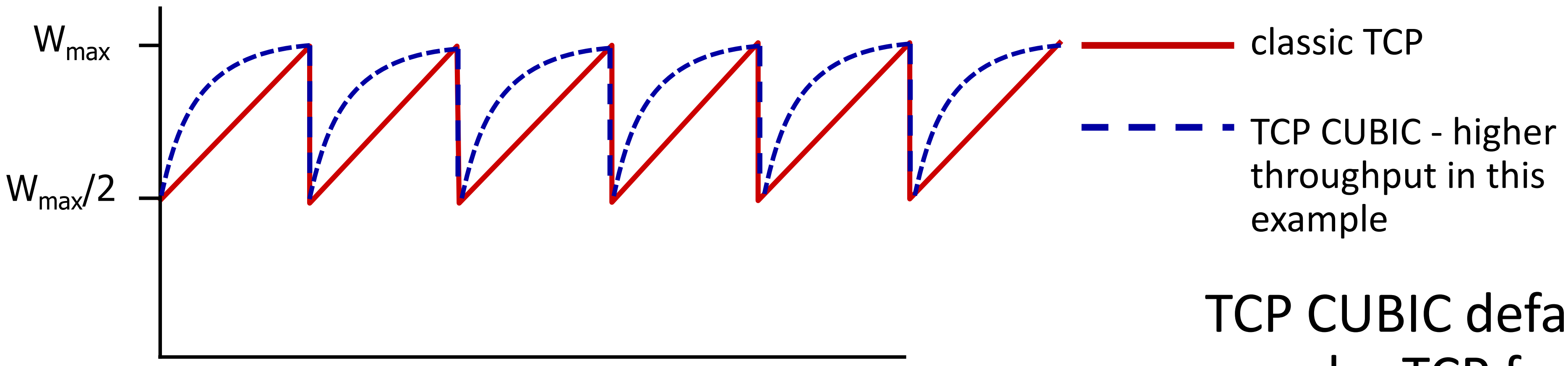


TCP Congestion Control: Summary



TCP CUBIC

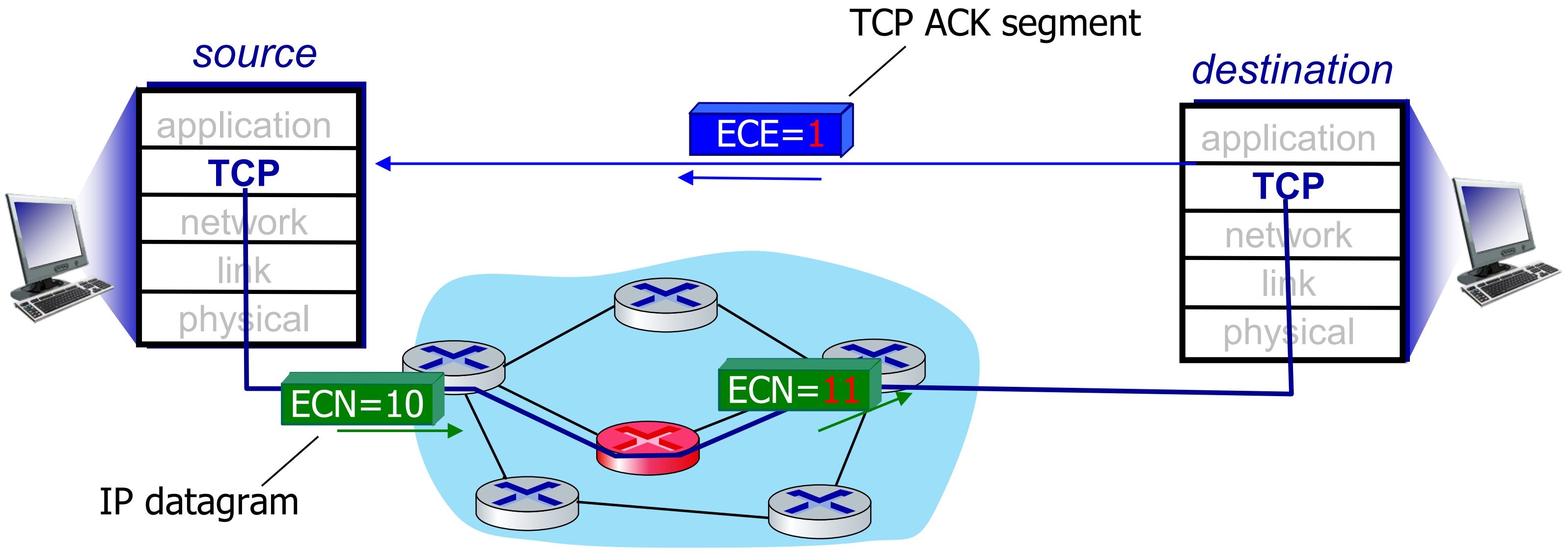
- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max}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



TCP CUBIC default in Linux, most popular TCP for popular Web servers

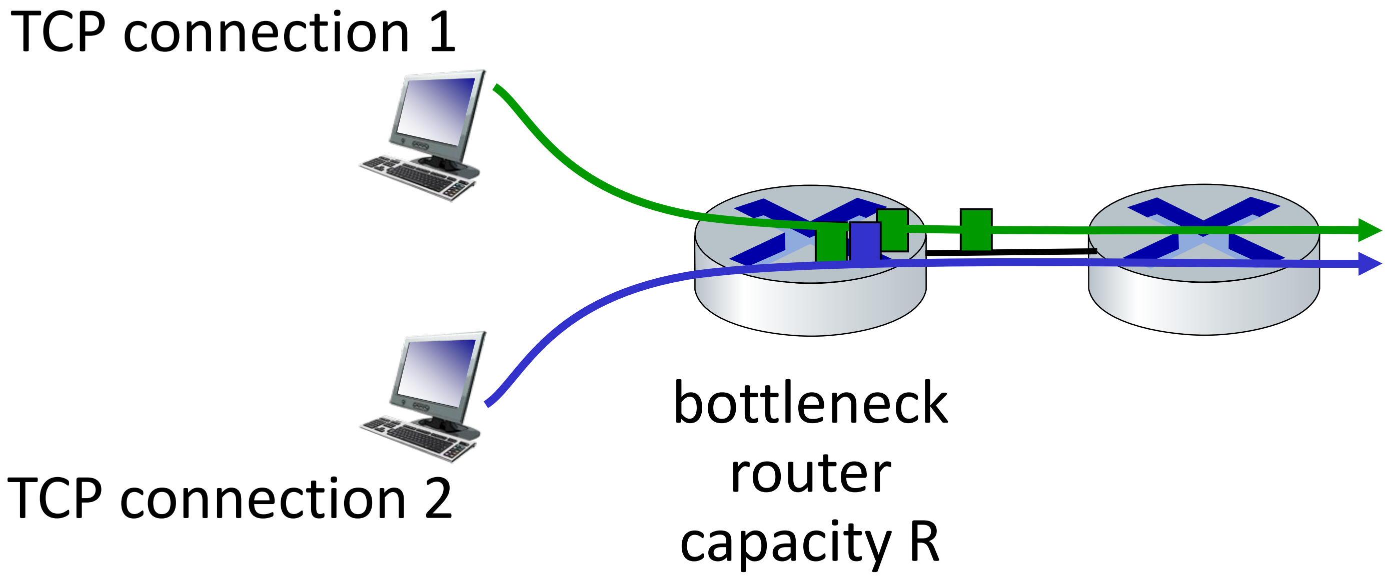
Explicit congestion notification (ECN)

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 congestion chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- Sender halves **`cwnd`**, and sets CWR in the next TCP segment header
  - involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



TCP fairness

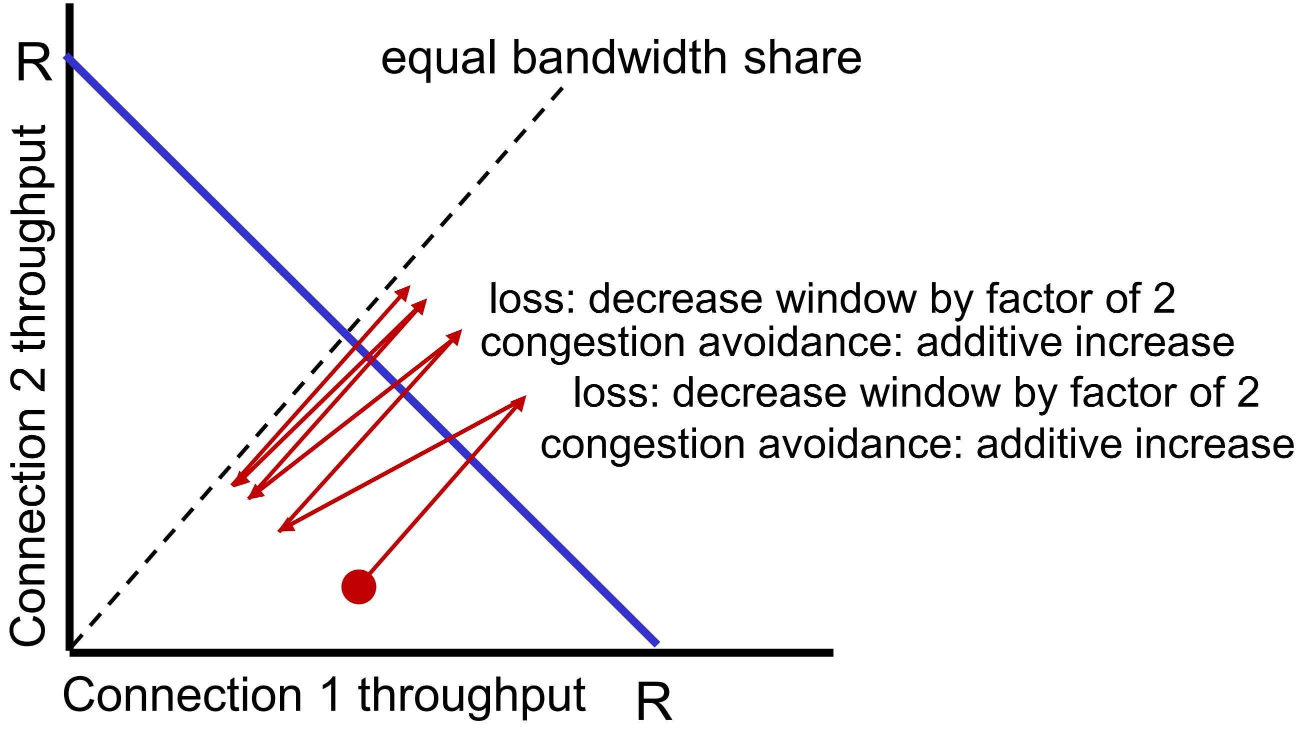
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



TCP Fairness

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be “fair”?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no “Internet police” policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Roadmap

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

Evolving transport-layer functionality

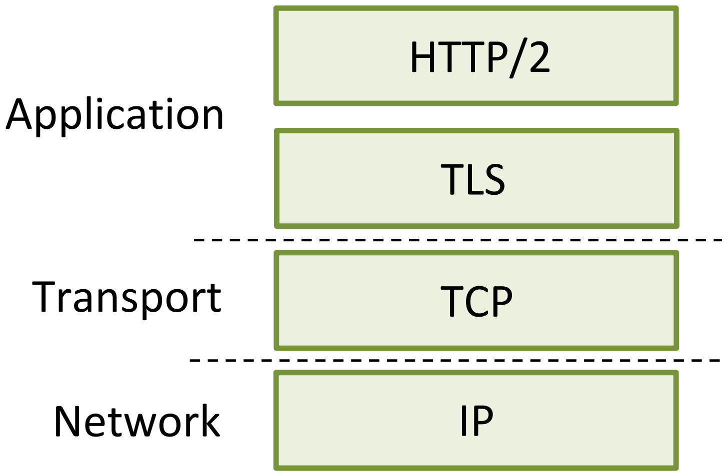
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

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)



HTTP/2 over TCP

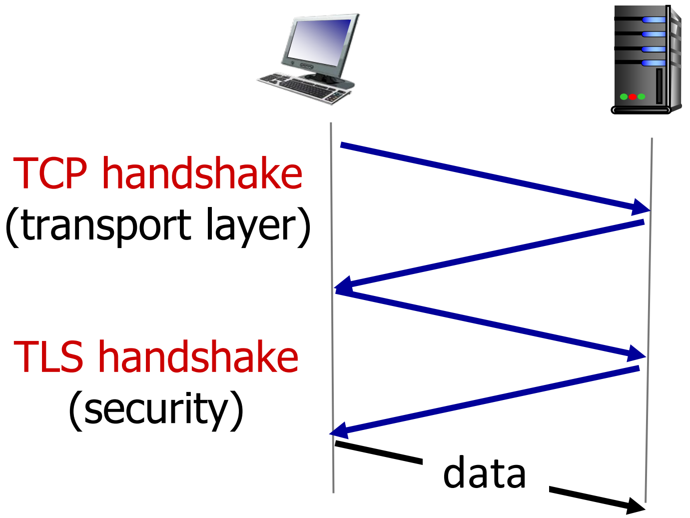
QUIC: Quick UDP Internet Connections

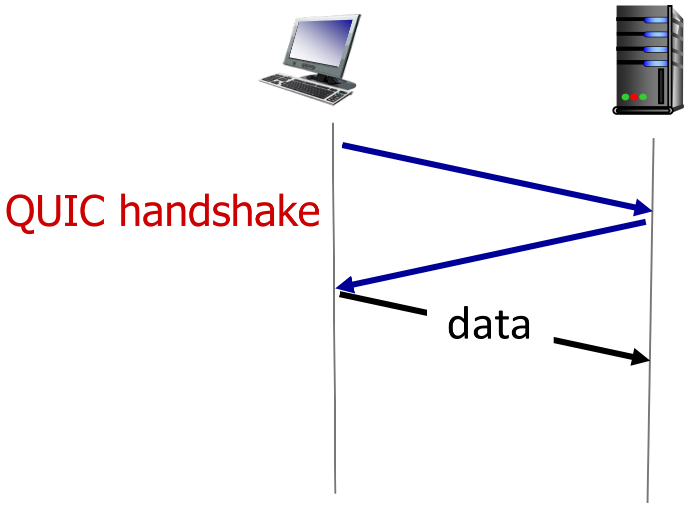
adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **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: Connection establishment





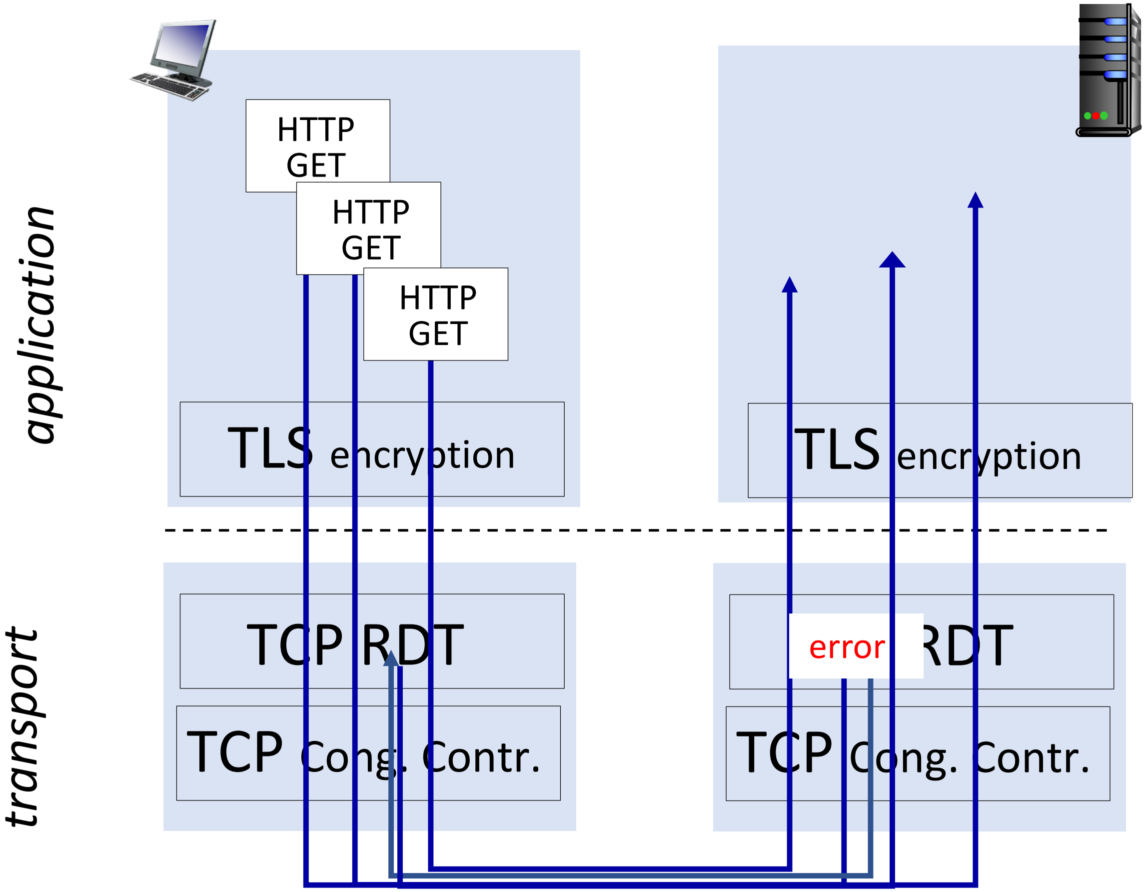
TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

Summary

- Principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Implementation in the Internet
  - UDP
  - TCP

What next?

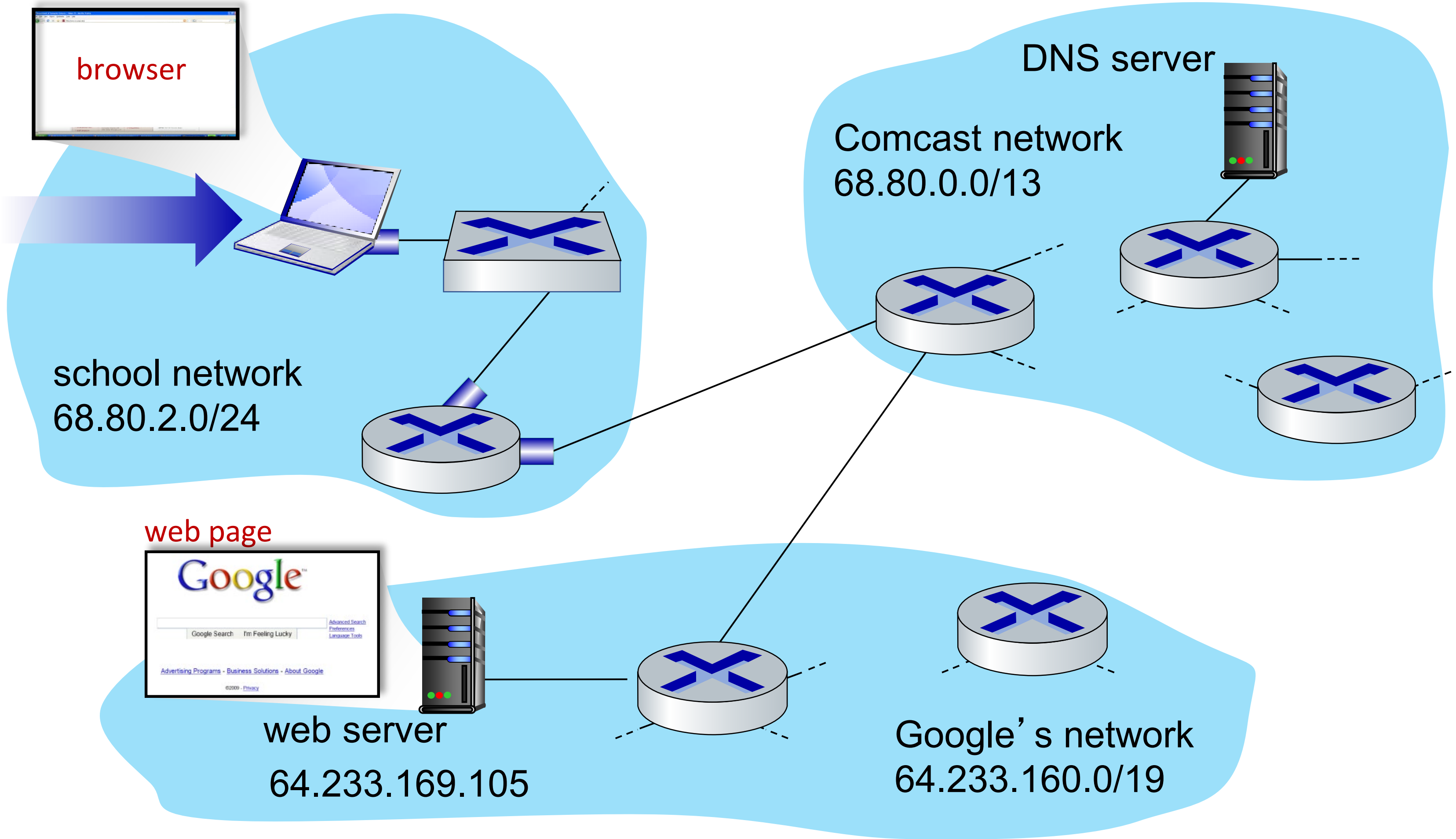
- Security
- Wireless and Mobility
- …

A day in the wonderful world of the Internet

A day in the life of a web request

- our journey down the protocol stack is now complete!
  - application, transport, network, link
- putting-it-all-together: synthesis!
  - *goal:* identify, review, understand protocols (at all layers) involved in seemingly simple scenario: requesting a Web page
  - *scenario:* student attaches laptop to campus network, requests/receives www.google.com

A day in the life: scenario

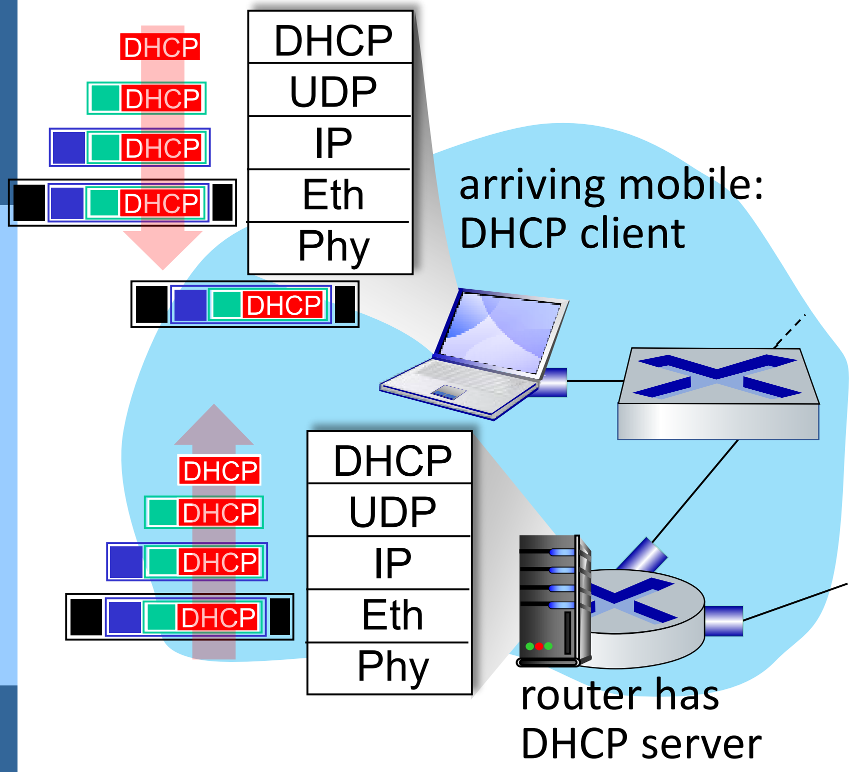


scenario:

- arriving mobile client attaches to network …
- requests web page: www.google.com

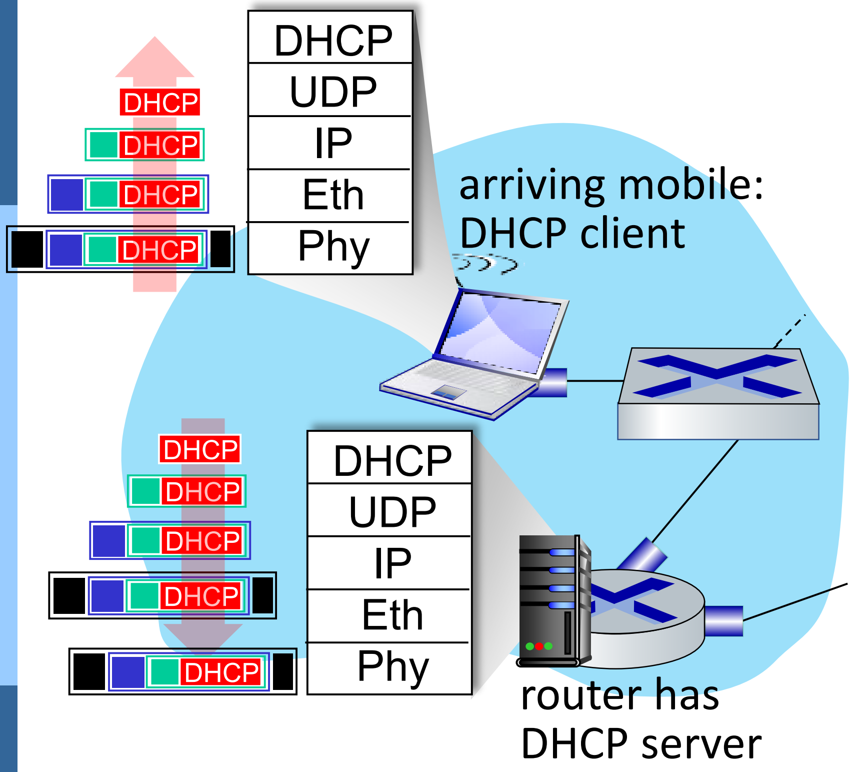
Sounds simple!

A day in the life: connecting to the Internet



- connecting laptop needs to get its own IP address, addr of first-hop router, addr of DNS server: use DHCP
- DHCP request encapsulated in UDP, encapsulated in IP, encapsulated in 802.3 Ethernet
- Ethernet frame broadcast (dest: FFFFFFFFFFFF) on LAN, received at router running DHCP server
- Ethernet demuxed to IP demuxed, UDP demuxed to DHCP

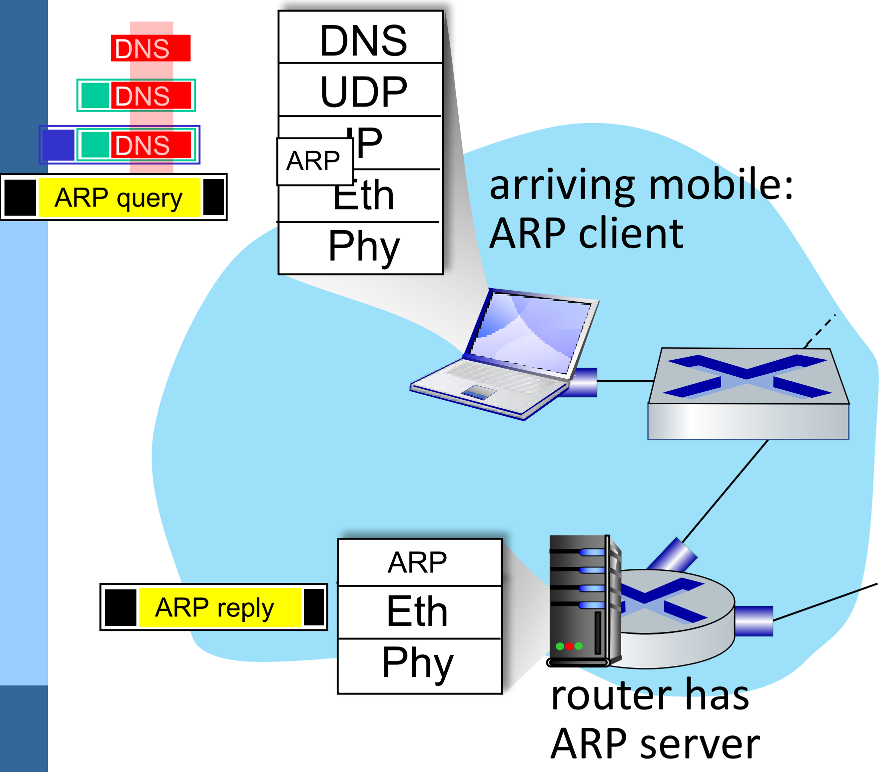
A day in the life: connecting to the Internet



- DHCP server formulates DHCP ACK containing client's IP address, IP address of first-hop router for client, name & IP address of DNS server
- encapsulation at DHCP server, frame forwarded (switch learning) through LAN, demultiplexing at client
- DHCP client receives DHCP ACK reply

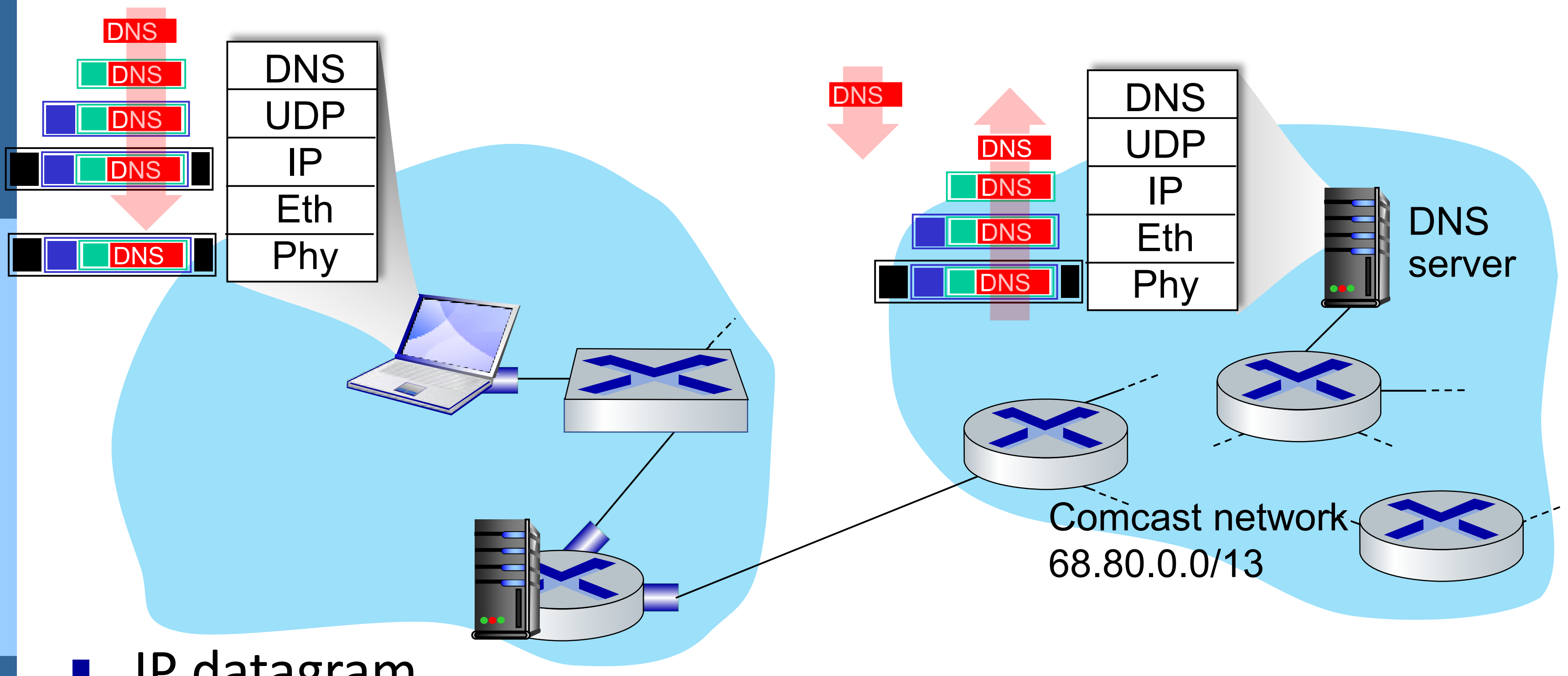
Client now has IP address, knows name & addr of DNS server, IP address of its first-hop router

A day in the life... ARP (before DNS, before HTTP)



- before sending HTTP request, need IP address of www.google.com: DNS
- DNS query created, encapsulated in UDP, encapsulated in IP, encapsulated in Eth. To send frame to router, need MAC address of router interface: ARP
- ARP query broadcast, received by router, which replies with ARP reply giving MAC address of router interface
- client now knows MAC address of first hop router, so can now send frame containing DNS query

A day in the life... using DNS

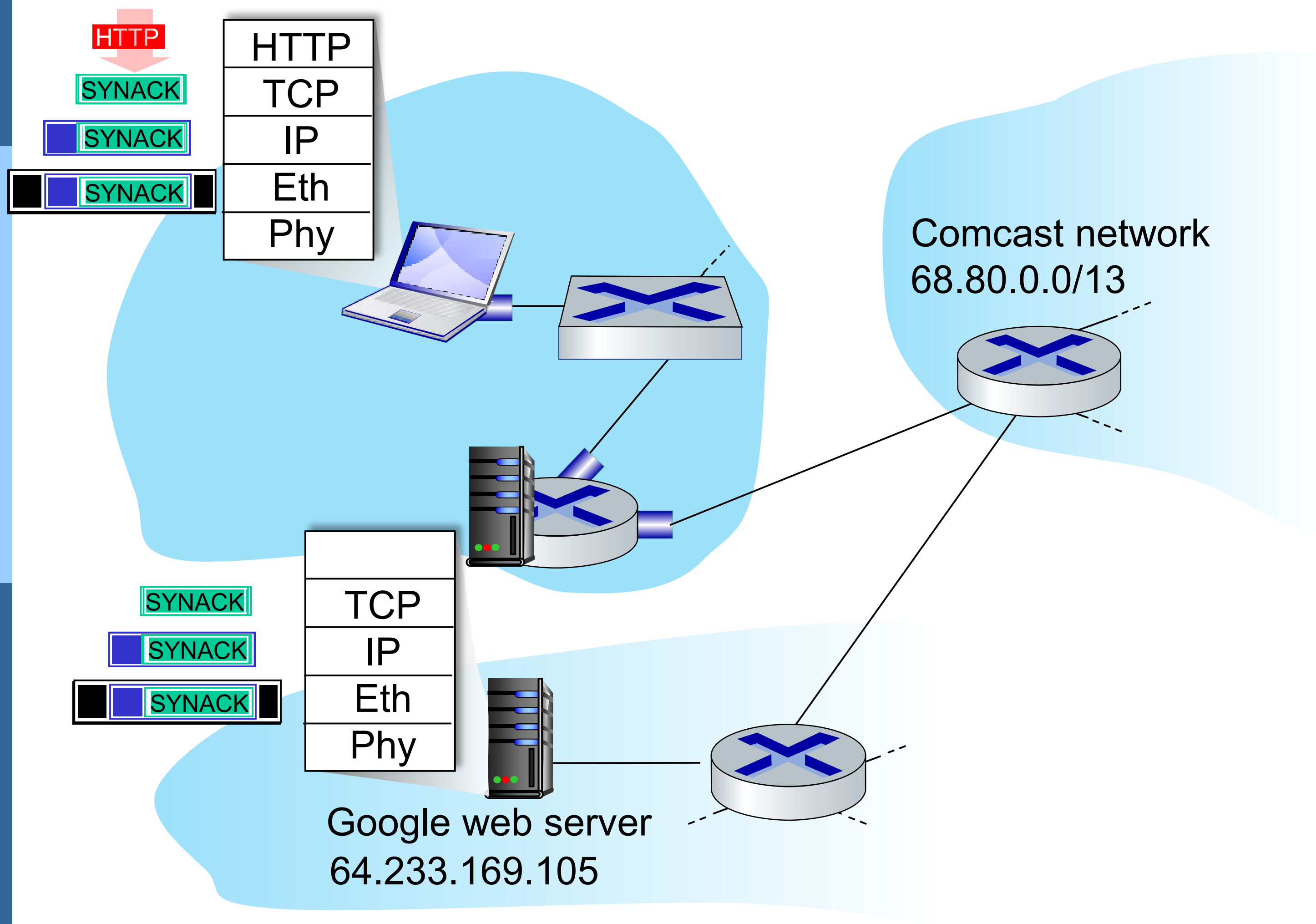


- demuxed to DNS
- DNS replies to client with IP address of www.google.com

- IP datagram containing DNS query forwarded via LAN switch from client to 1st hop router

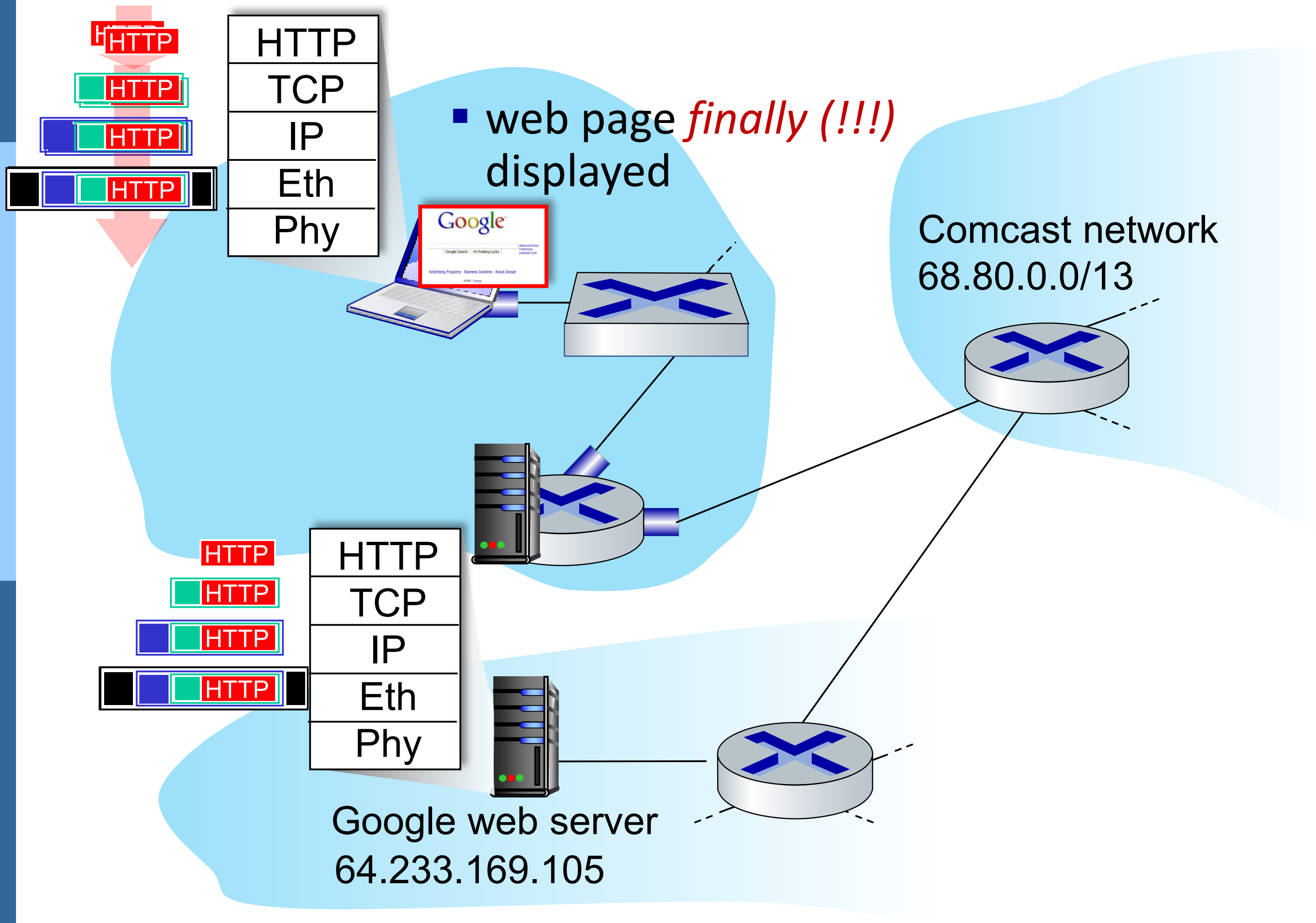
- IP datagram forwarded from campus network into Comcast network, routed (tables created by RIP, OSPF, IS-IS and/or BGP routing protocols) to DNS server

A day in the life...TCP connection carrying HTTP



- to send HTTP request, client first opens TCP socket to web server
- TCP SYN segment (step 1 in TCP 3-way handshake) inter-domain routed to web server
- web server responds with TCP SYNACK (step 2 in TCP 3-way handshake)
- TCP connection established!

A day in the life… HTTP request/reply



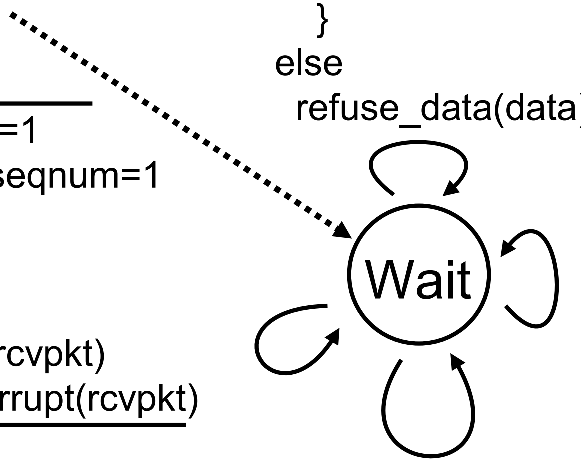
- HTTP request sent into TCP socket
- IP datagram containing HTTP request routed to www.google.com
- web server responds with HTTP reply (containing web page)
- IP datagram containing HTTP reply routed back to client

Additional Chapter 3 slides

Go-Back-N: sender extended FSM

```
rdt_send(data)
--------------------------------
if (nextseqnum < base+N) {
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum)
     start_timer
   nextseqnum++
   }
else
  refuse_data(data)
```

```
Λ
--------------
base=1
nextseqnum=1
```

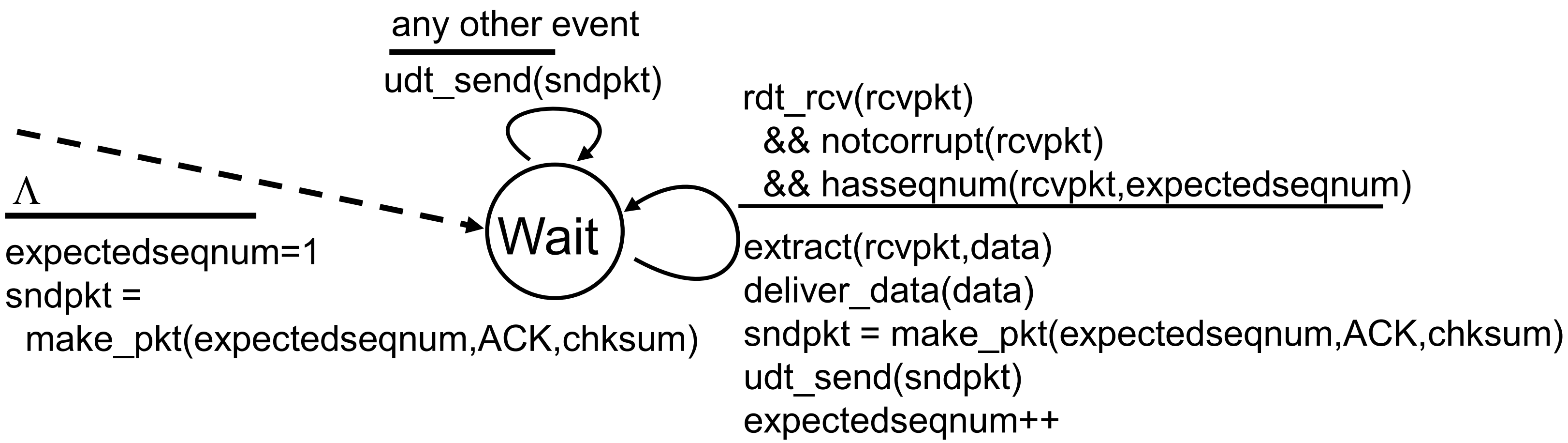



```
timeout
--------------------------------
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

```
rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)
--------------------------
```

```
rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
--------------------------
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

Go-Back-N: receiver extended FSM



ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **expectedseqnum**

- out-of-order packet:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

TCP sender (simplified)

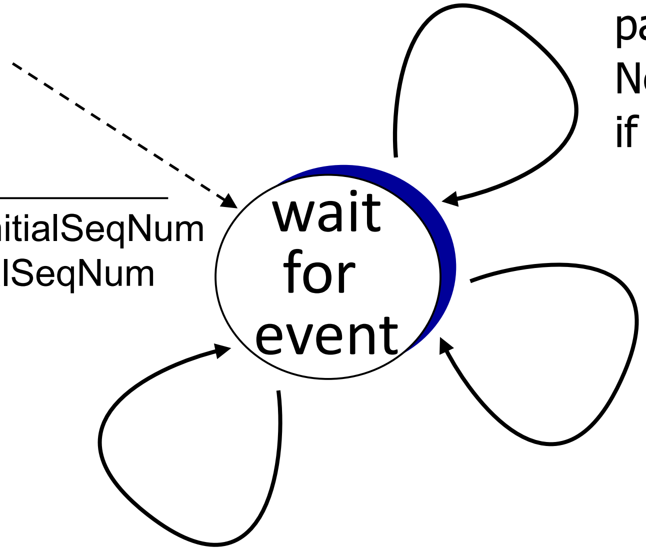
data received from application above

---

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



timeout

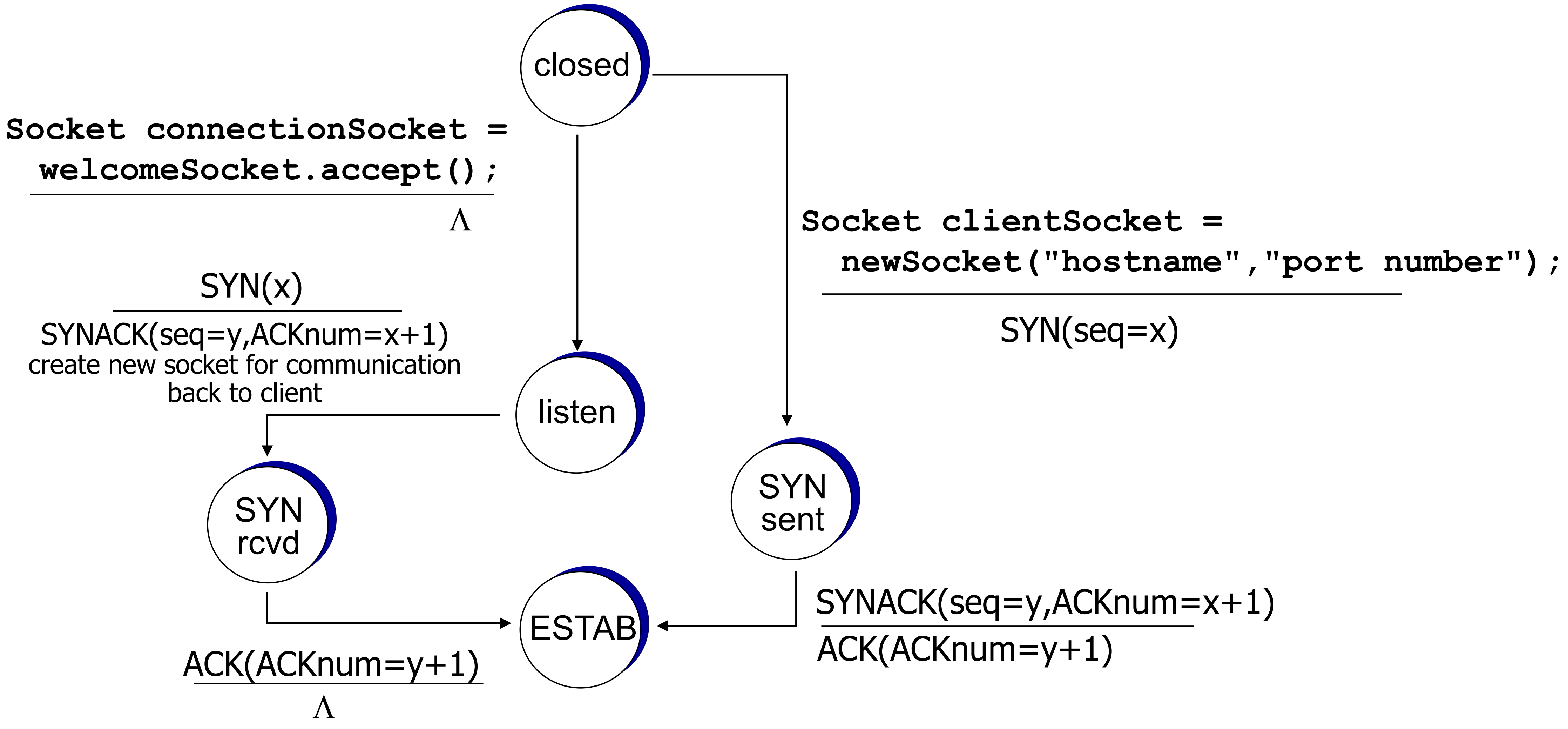
---

retransmit not-yet-acked segment
    with smallest seq. #
start timer

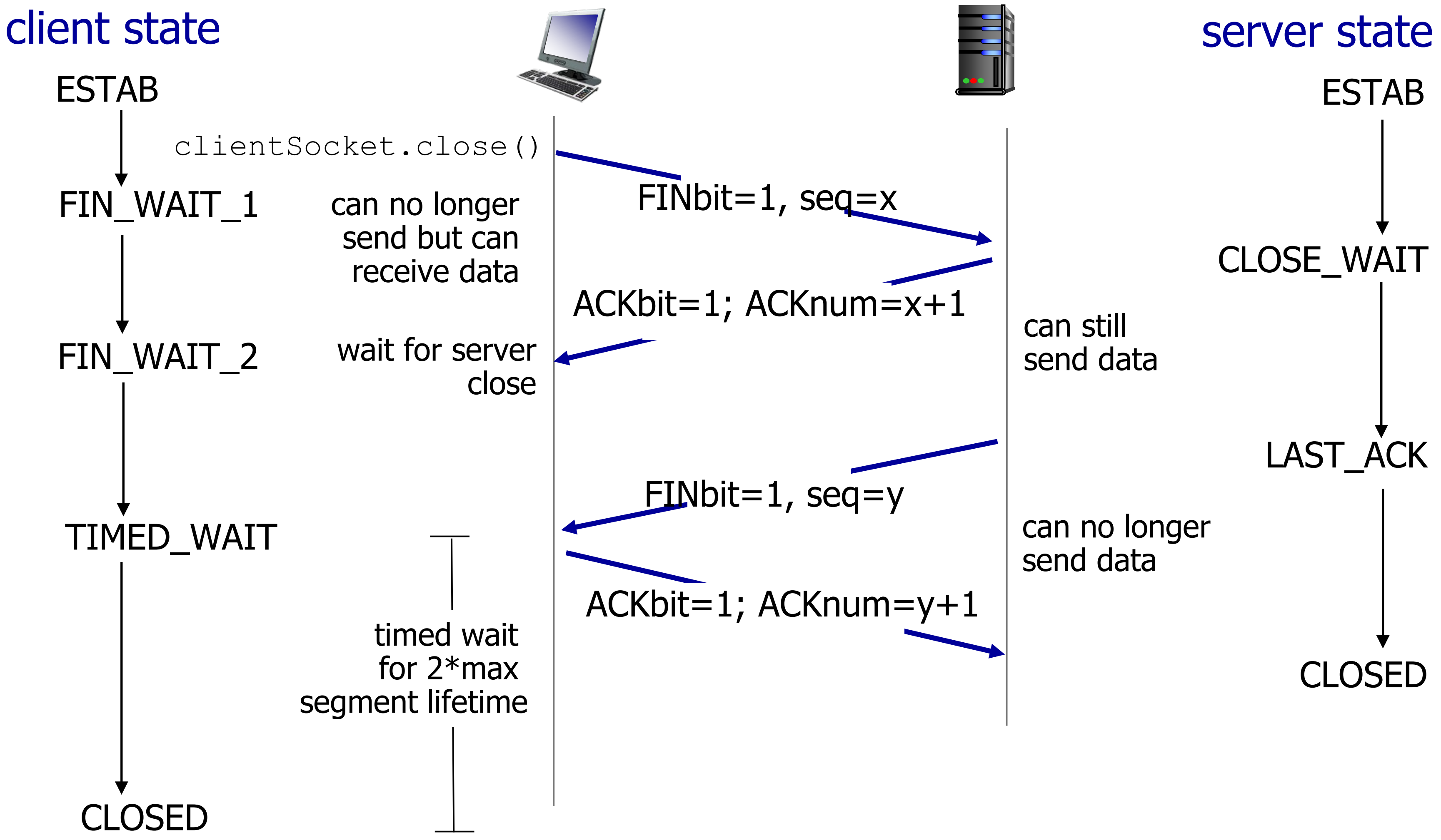
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
  }
```

TCP 3-way handshake FSM



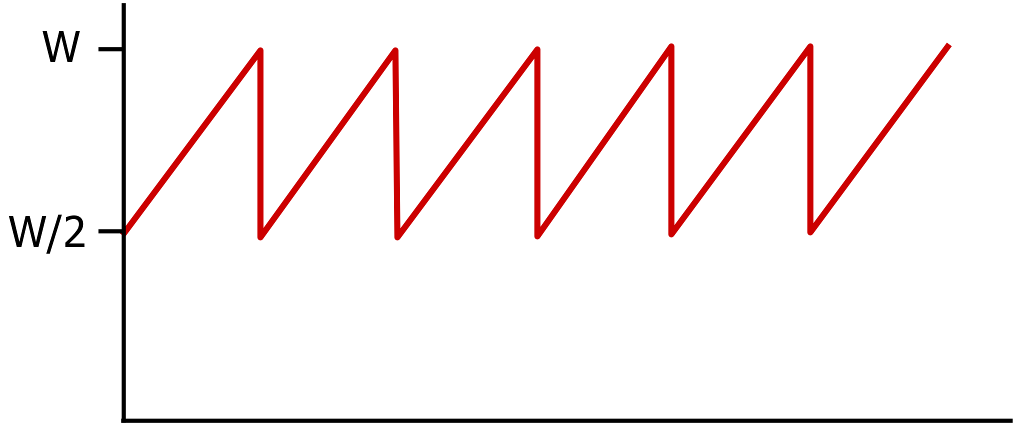
Closing a TCP connection



TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume there is always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP over "long, fat pipes"

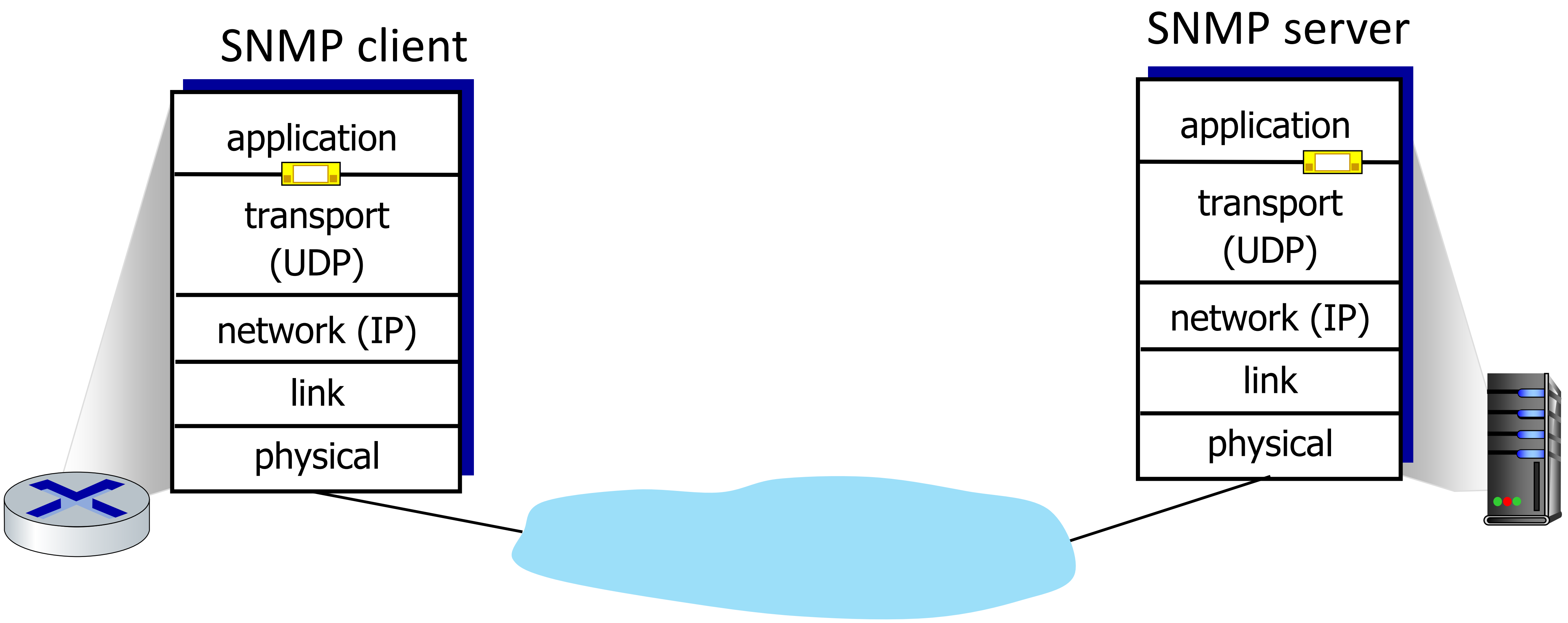
- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ *– a very small loss rate!*

- versions of TCP for long, high-speed scenarios

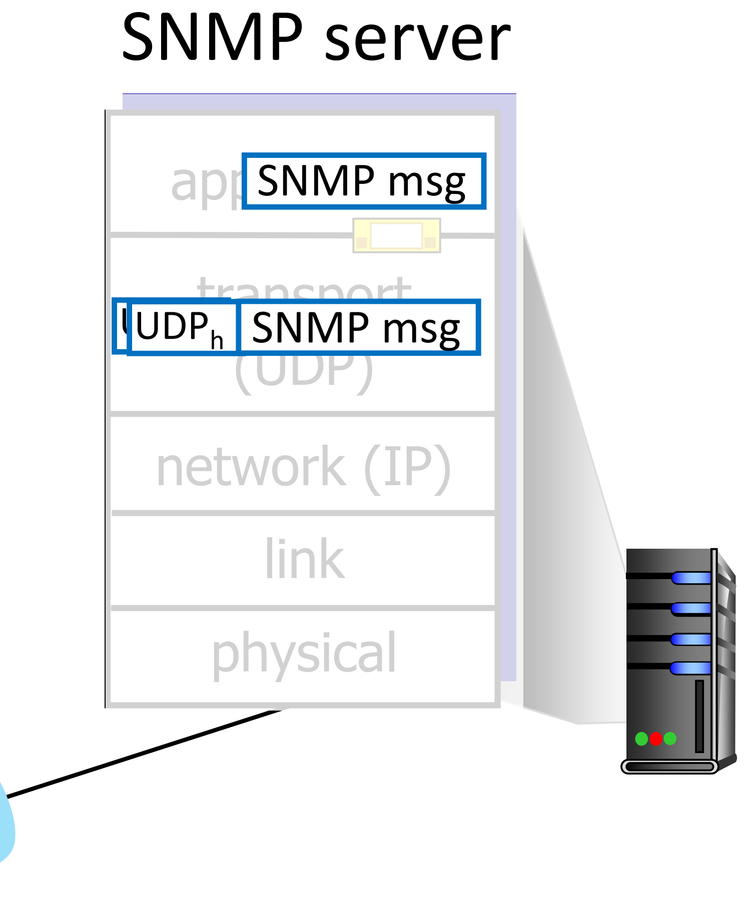
UDP: Transport Layer Actions



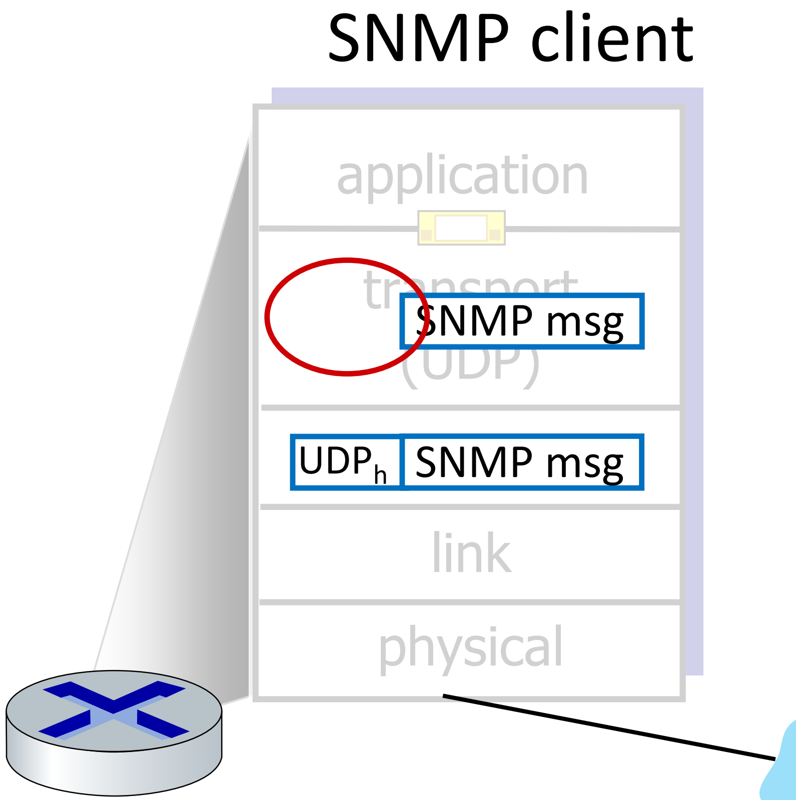
UDP: Transport Layer Actions

UDP sender actions:

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

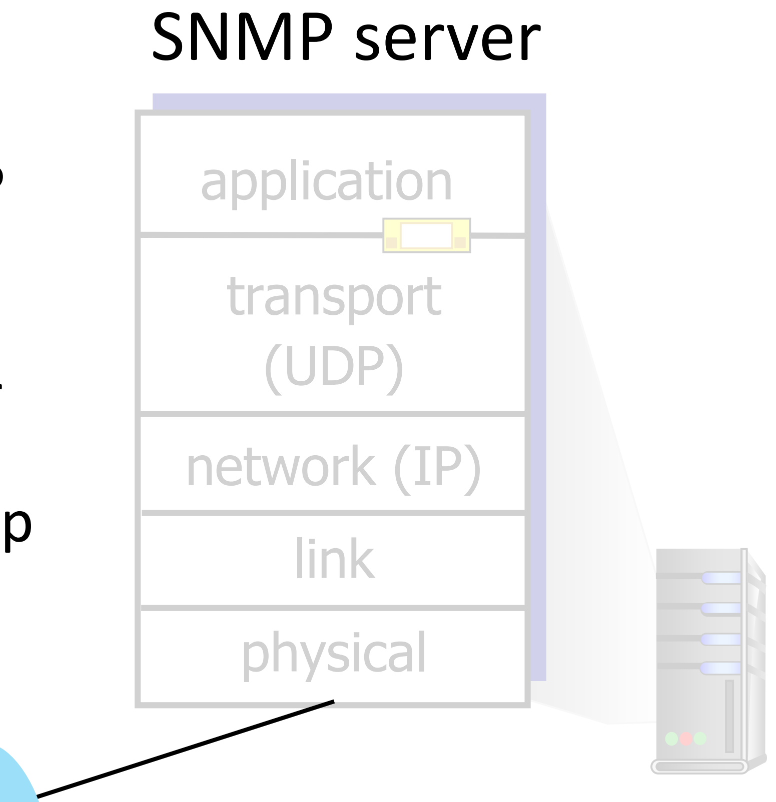


UDP: Transport Layer Actions



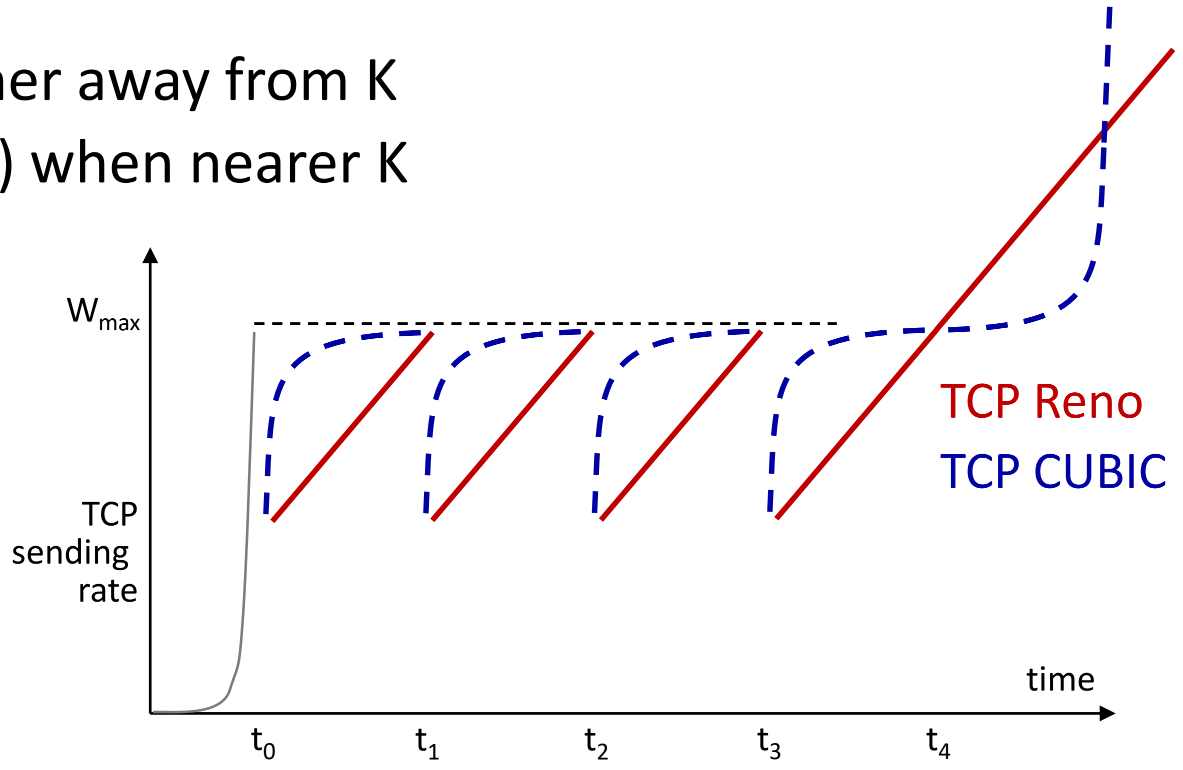
UDP receiver actions:

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



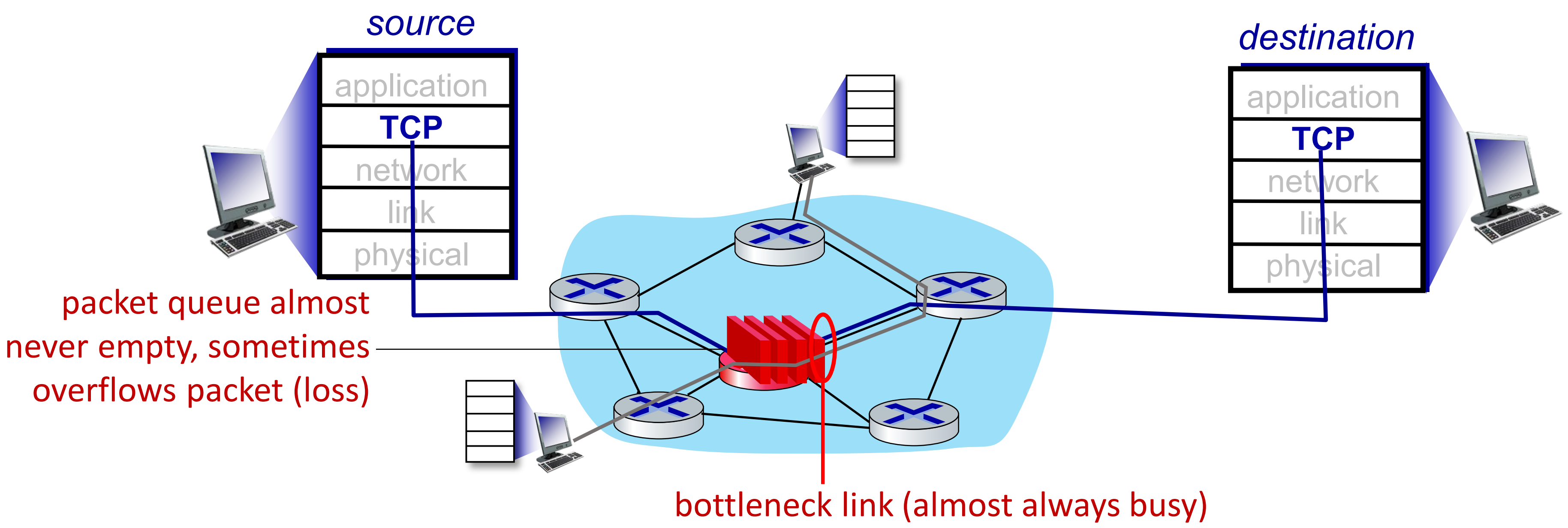
TCP CUBIC

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



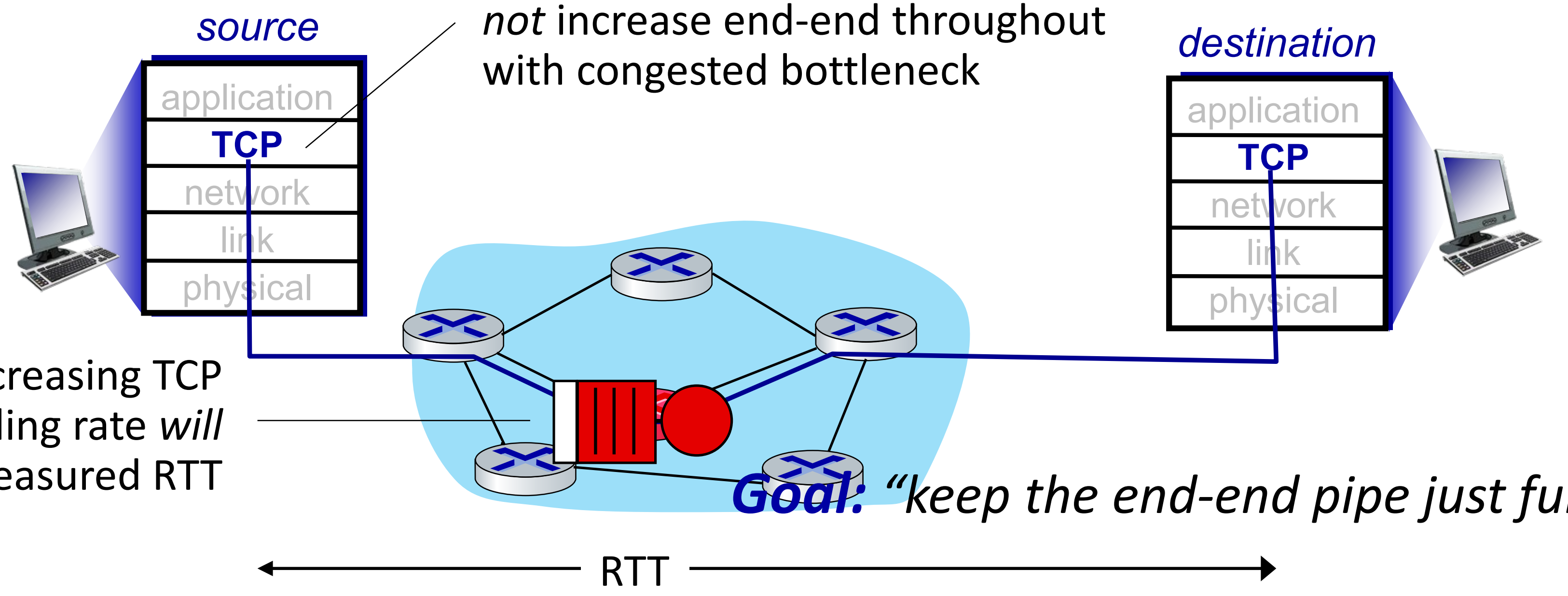
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



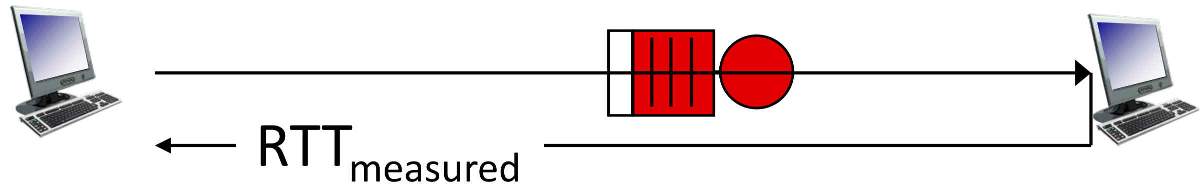
insight: increasing TCP sending rate will *not* increase end-end throughout with congested bottleneck

insight: increasing TCP sending rate *will* increase measured RTT

Goal: *"keep the end-end pipe just full, but not fuller"*

Delay-based TCP congestion control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is $cwnd/RTT_{min}$

if measured throughput "very close" to uncongested throughput
increase `cwnd` linearly /* since path not congested */
else if measured throughput "far below" uncongested throughout
decrease `cwnd` linearly /* since path is congested */

Delay-based TCP congestion control

- congestion control without inducing/forcing loss
- maximizing throughout ("keeping the just pipe full... ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network