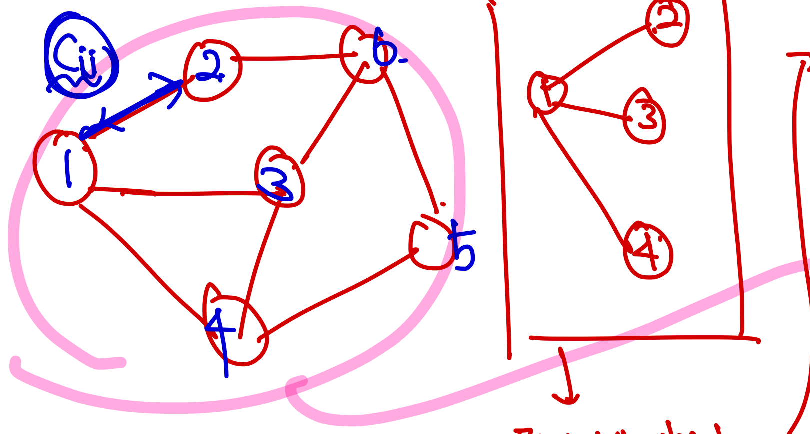


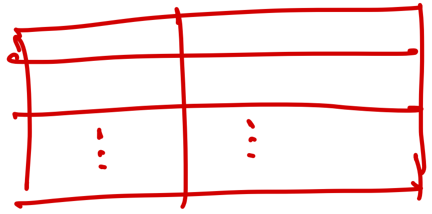
모든 노드!

그래프

여러 종류 그래프 그릴 수 있음

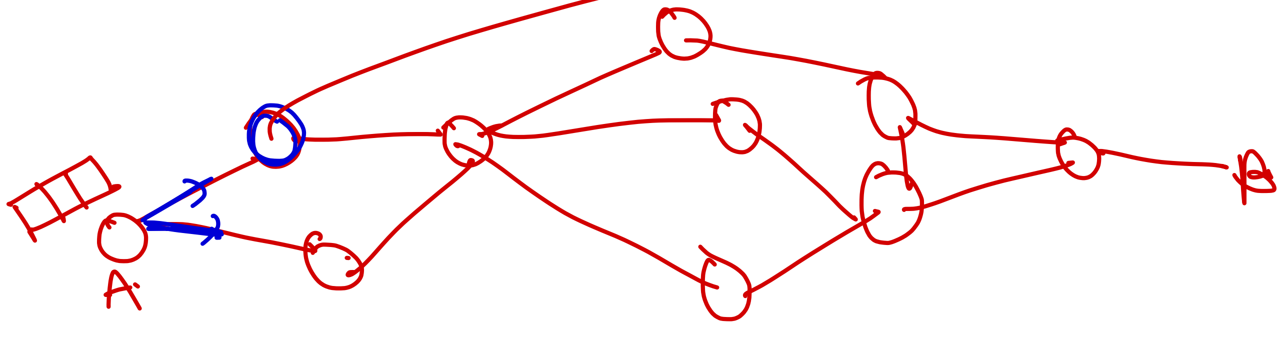
전체에게 알린다.

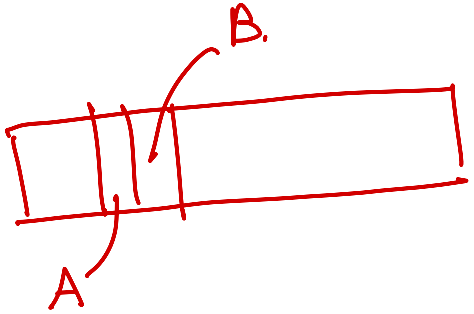
Chapter: Transport Layer Protocols & Network Layer (IP)



Routing table.

(이정표)





Transport layer: overview

Our goal:

- understand principles behind transport layer services:
  - flow control
  - congestion control

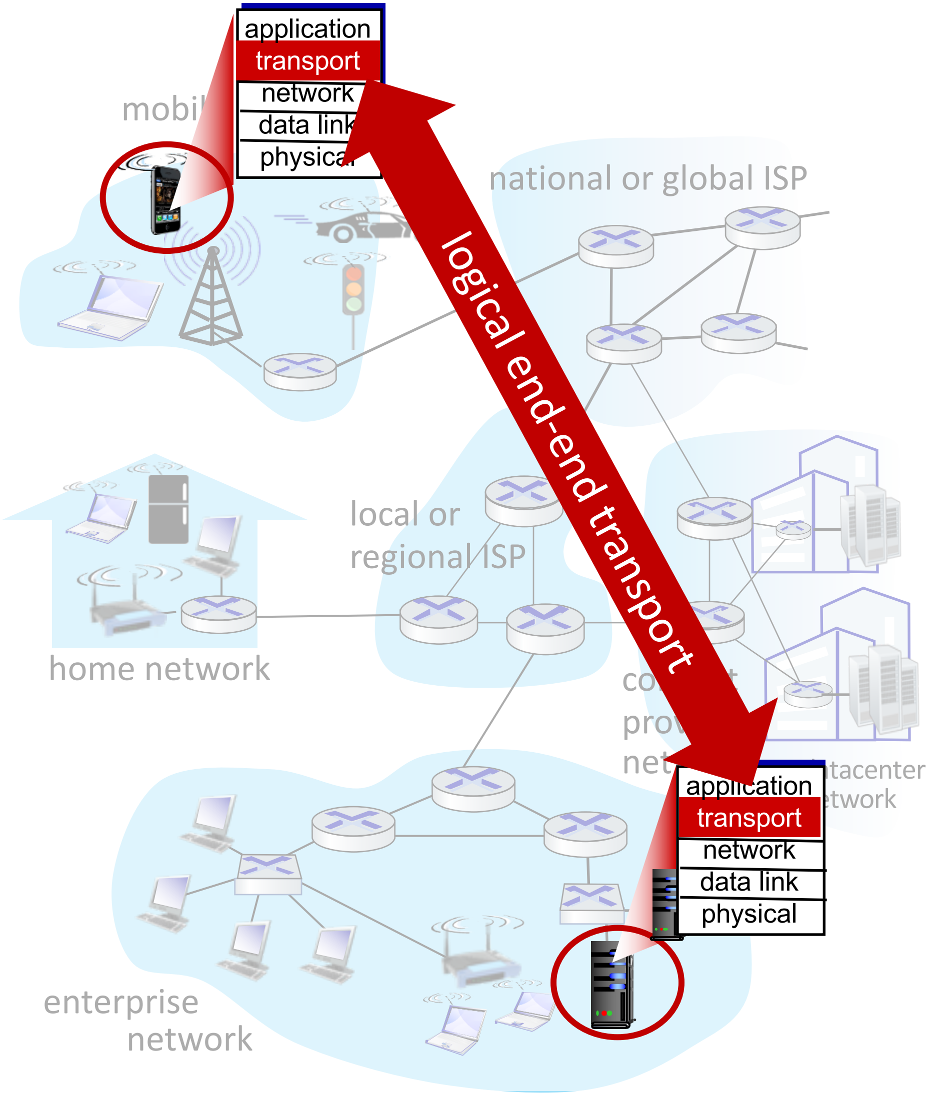
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Transport layer: roadmap

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP

Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



Transport vs. network layer services and protocols

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

household analogy:

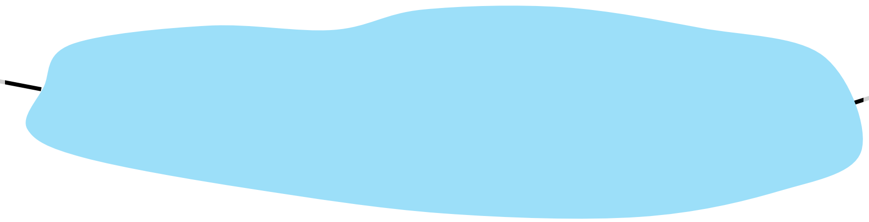
12 kids in Ann's house sending letters to 12 kids in Bill's house:

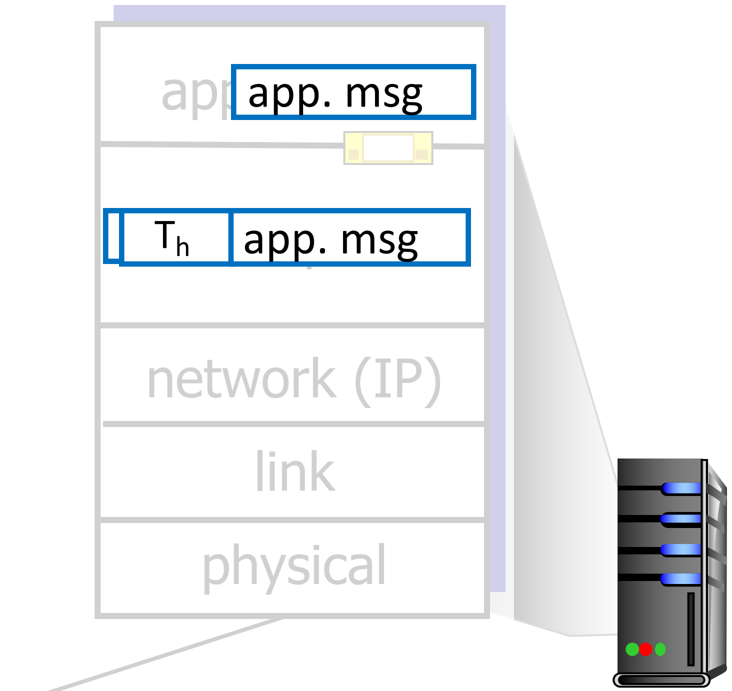
- hosts = houses
- processes = kids
- app messages = letters in envelopes

Transport Layer Actions

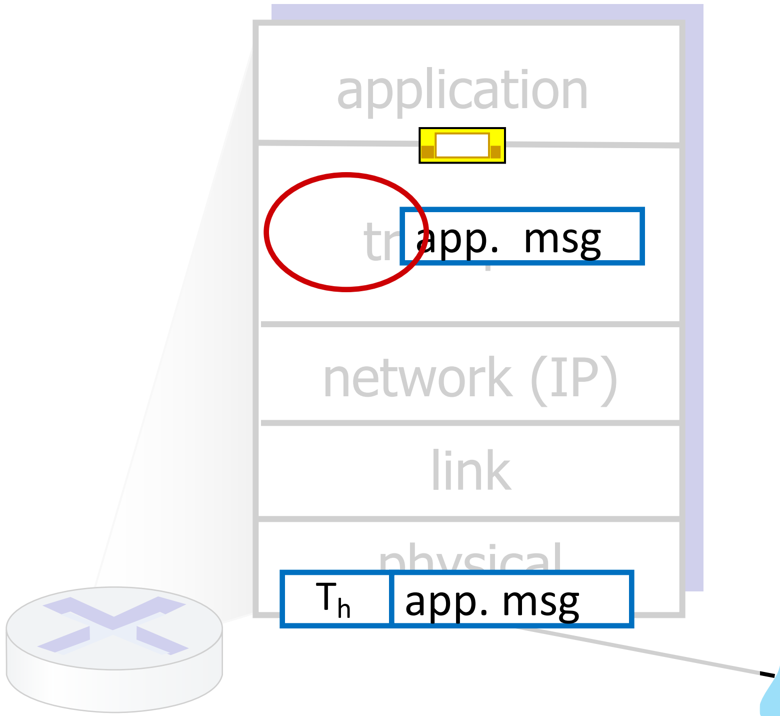
Sender:

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



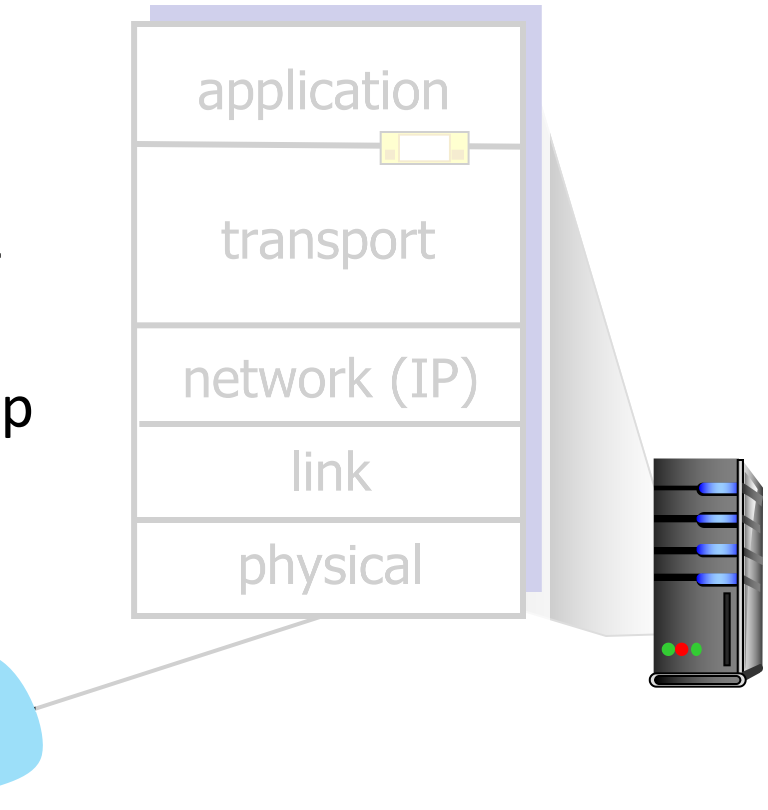


Transport Layer Actions



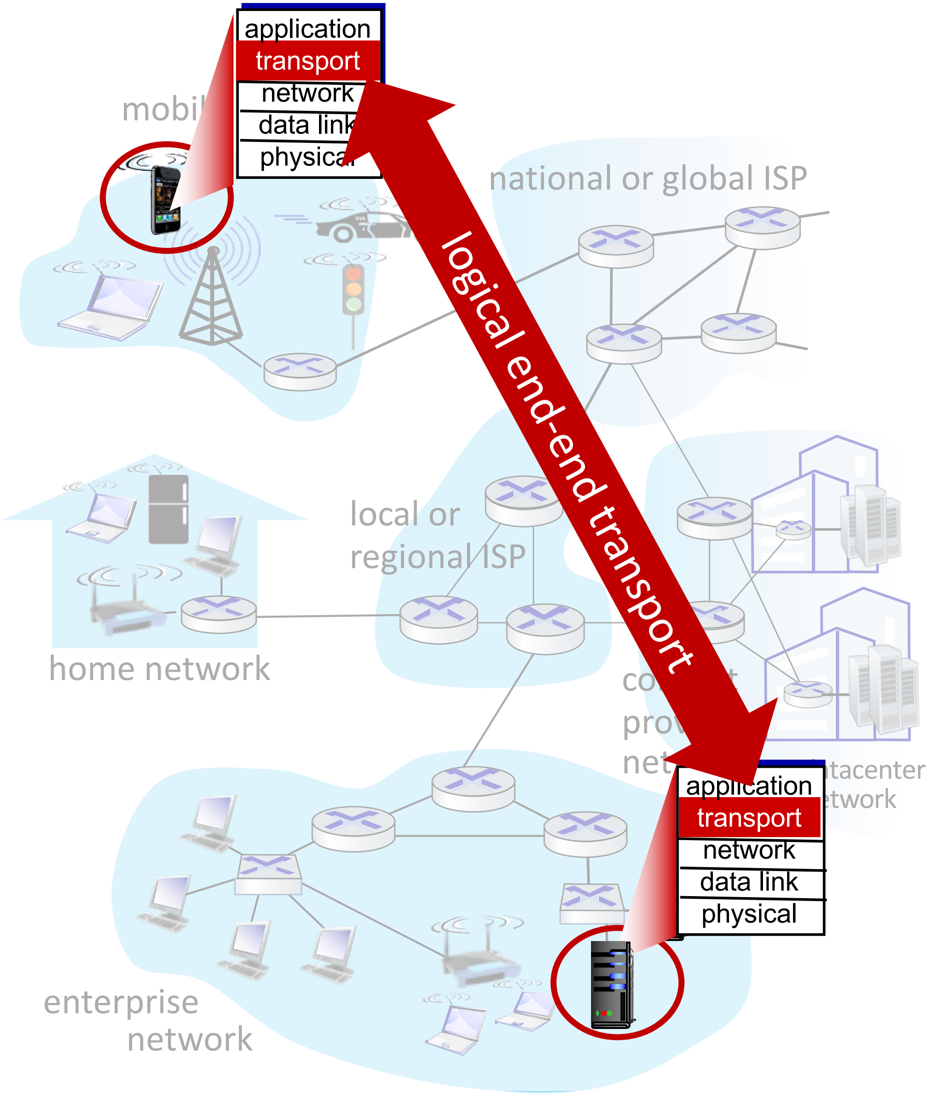
Receiver:

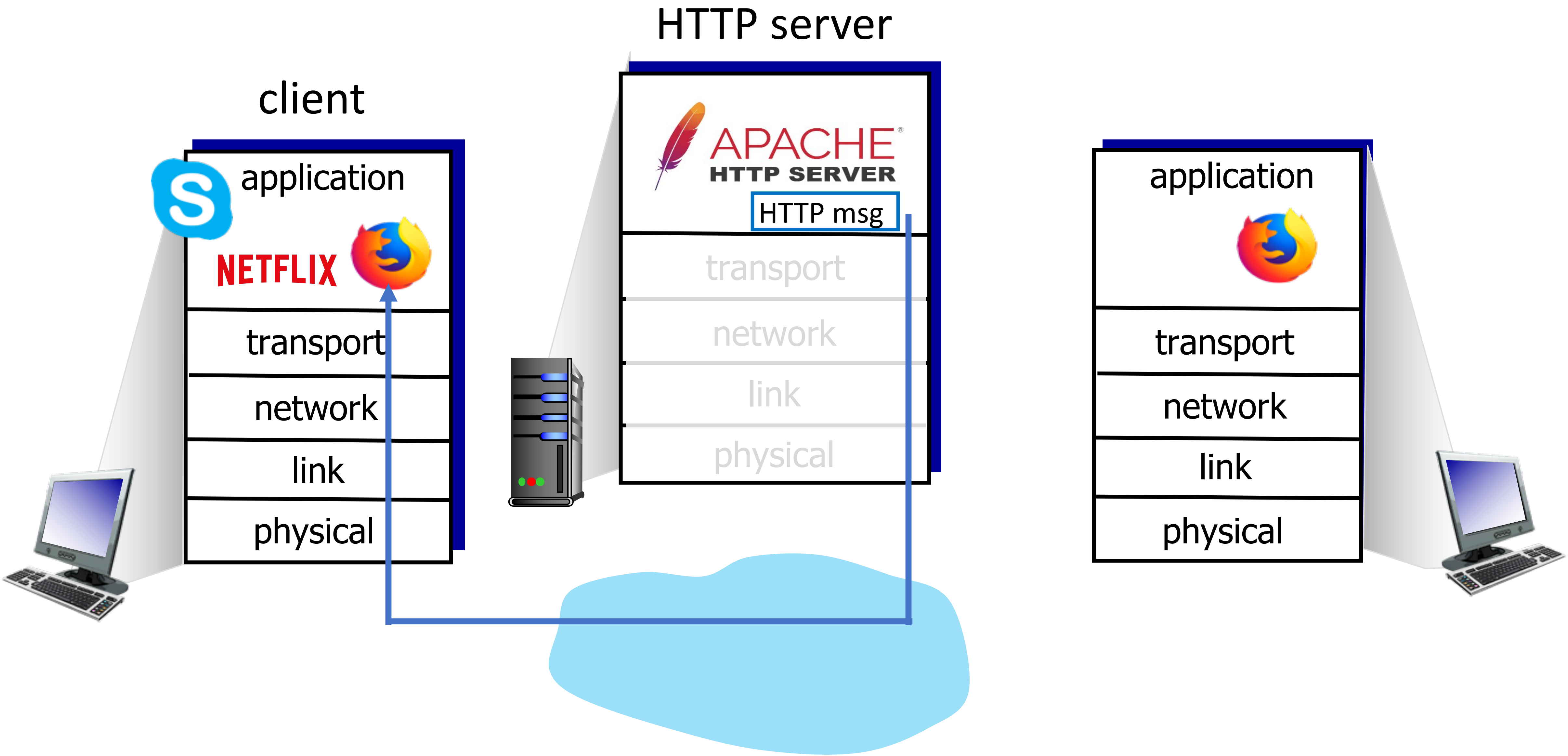
- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

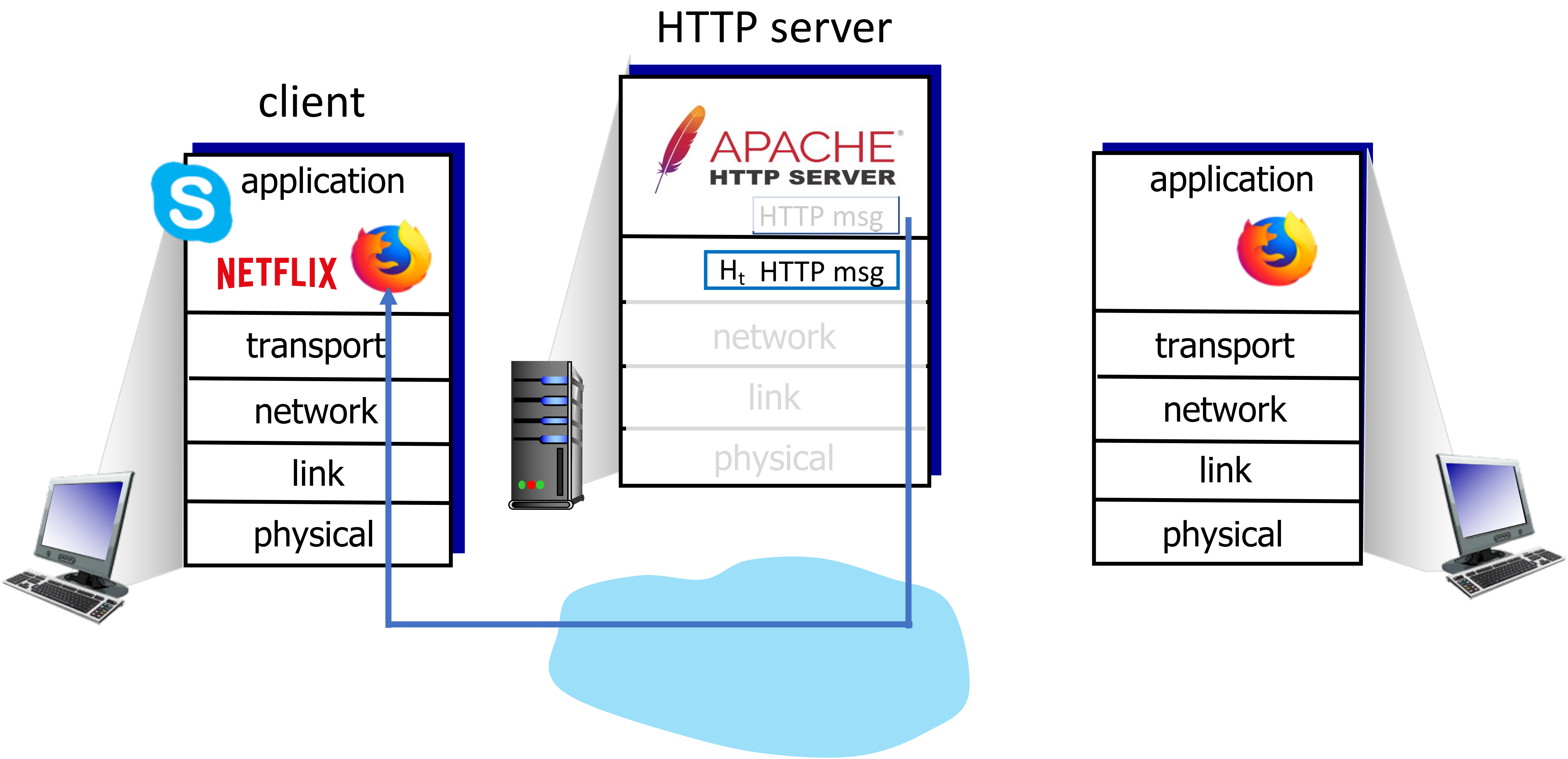


Two principal Internet transport protocols

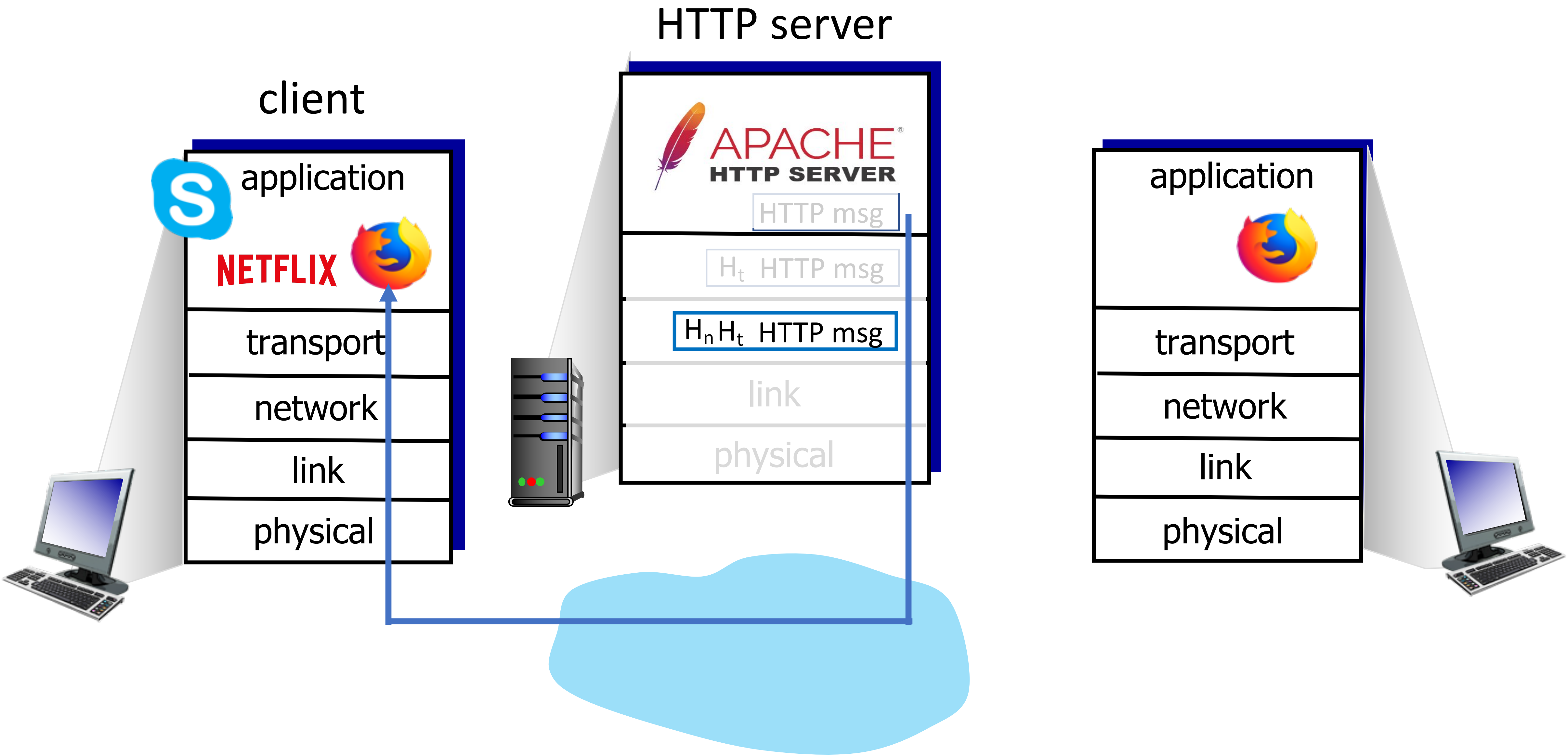
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth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

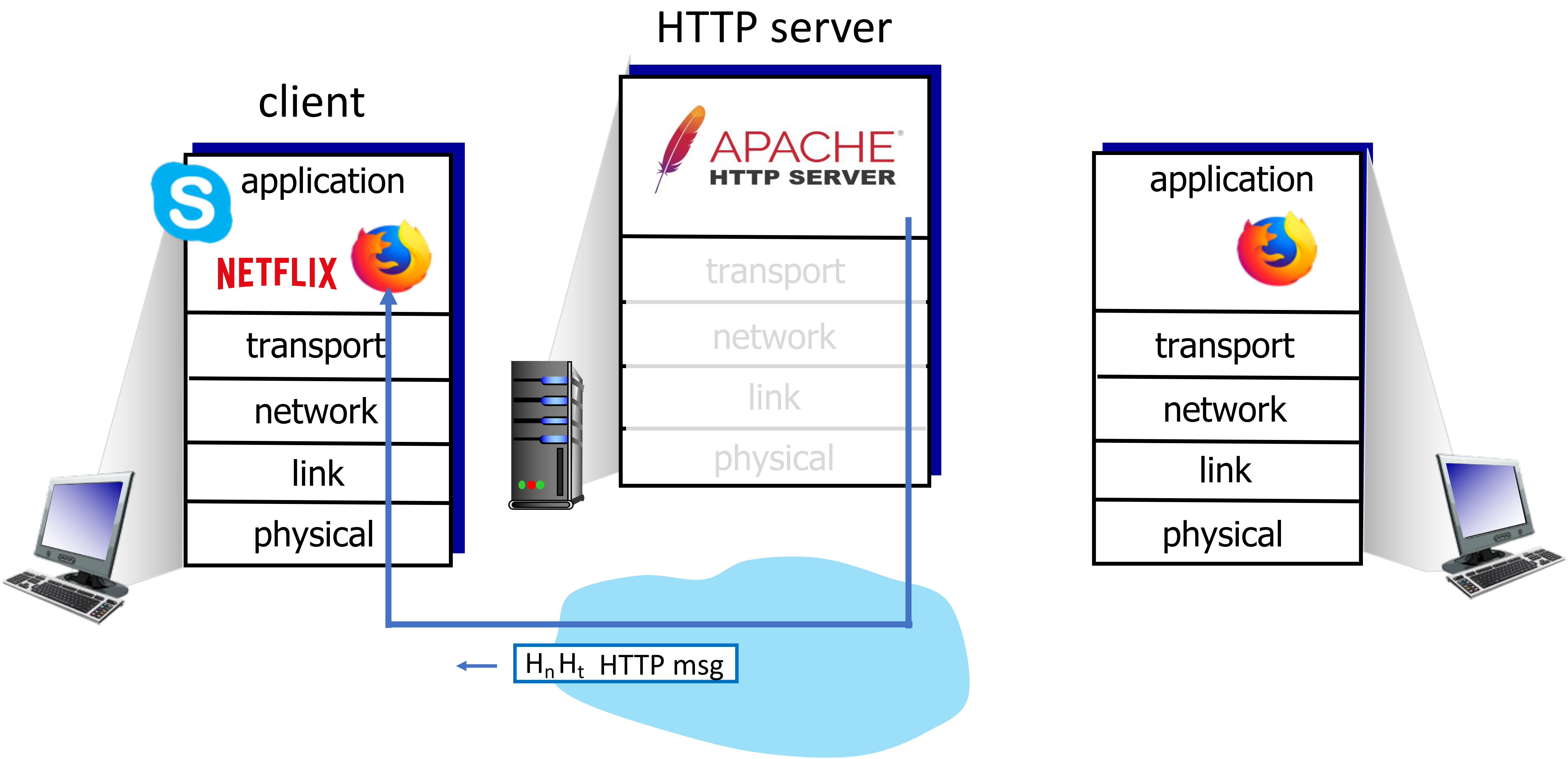



HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
transport
network
link
physical
application
transport
network
link
physical

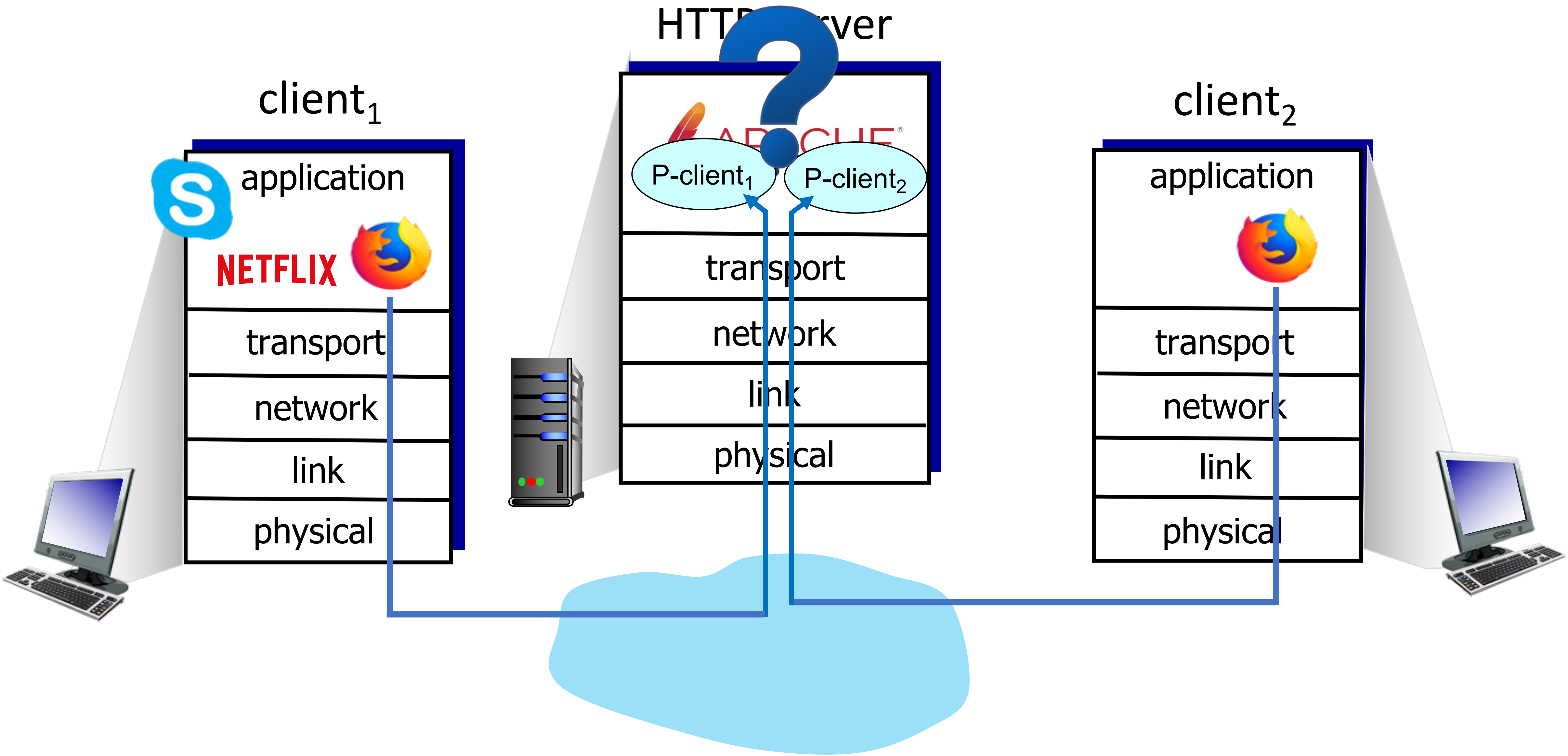


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
network
link
physical
application
transport
network
link
physical


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



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



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

Transport layer : roadmap

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP

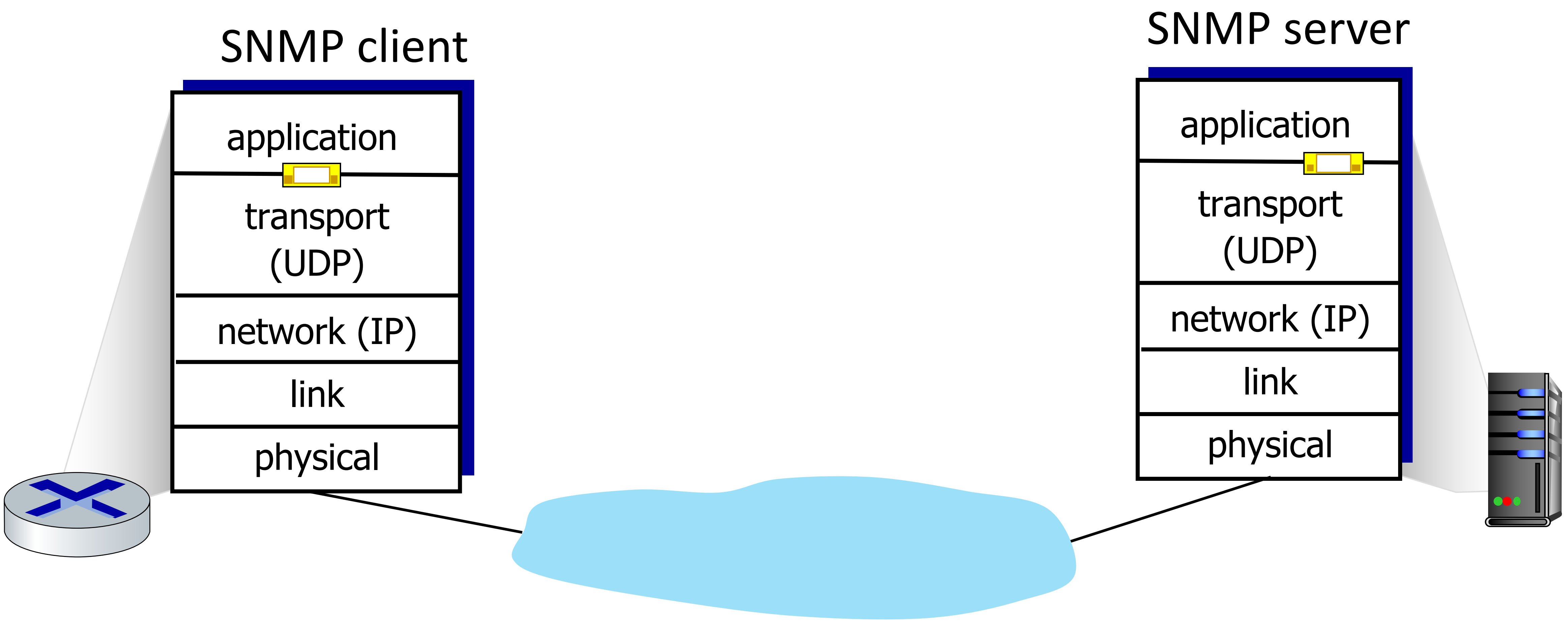
UDP: User Datagram Protocol

- "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

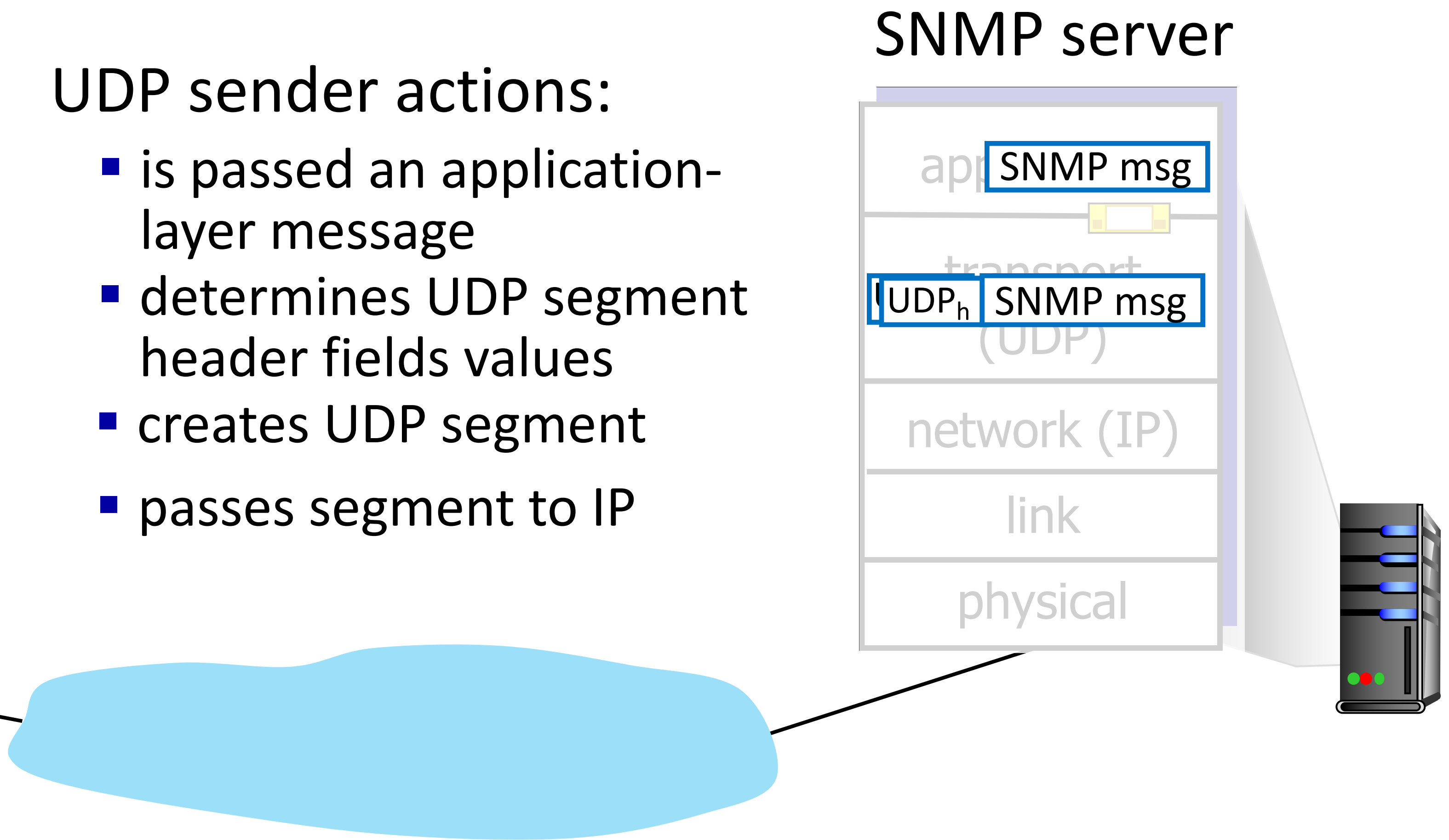
UDP: Transport Layer Actions



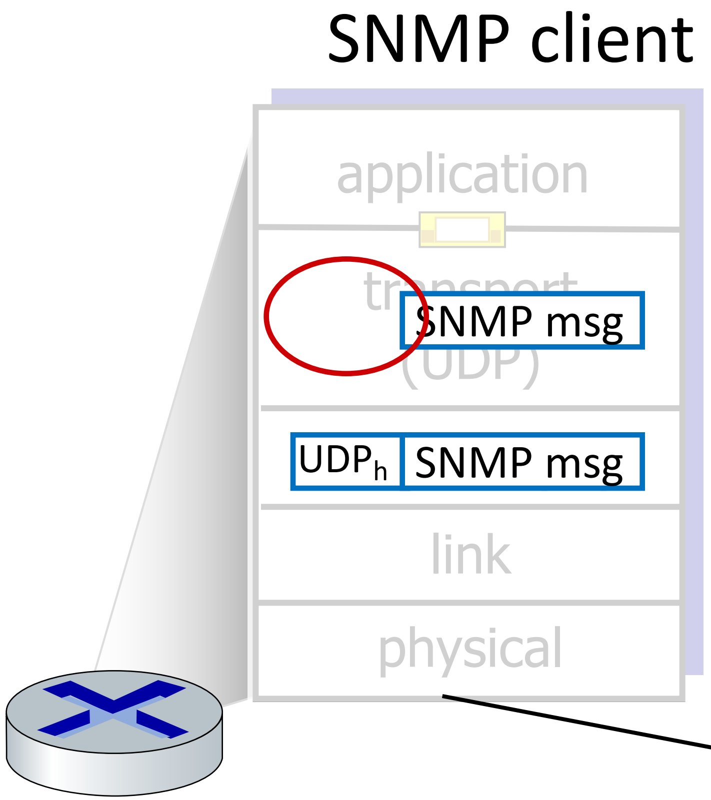
UDP: Transport Layer Actions

UDP sender actions:

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

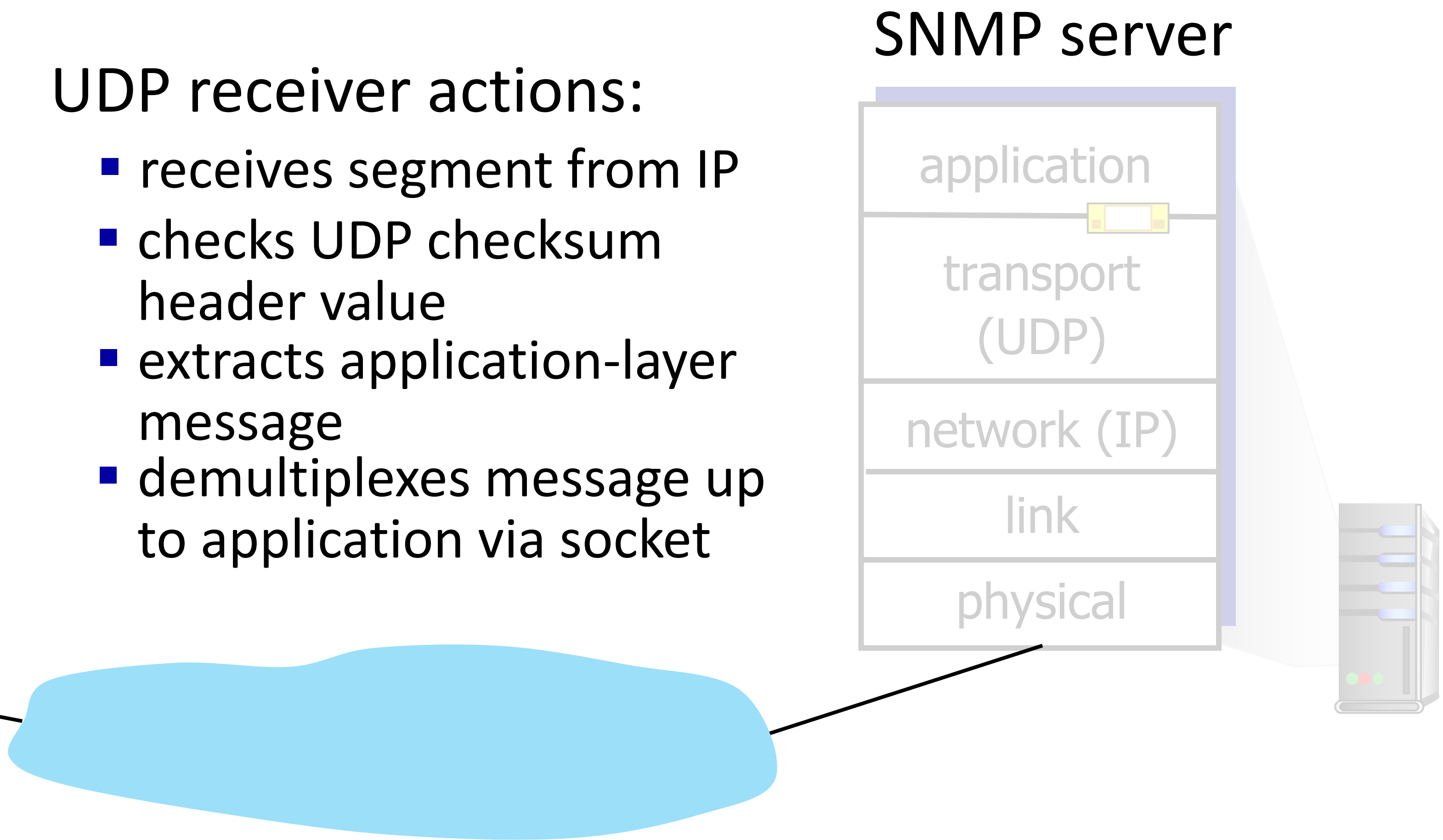


UDP: Transport Layer Actions

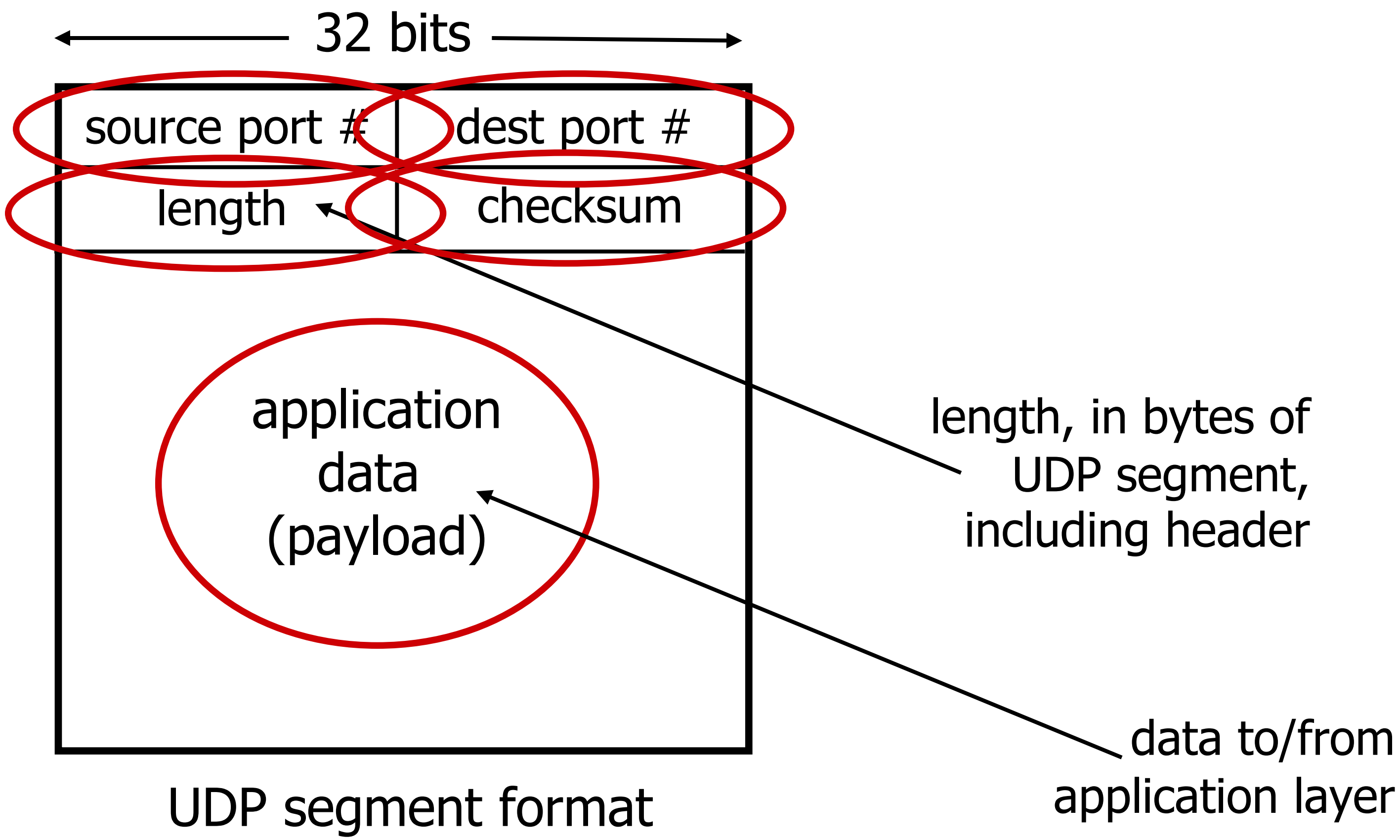


UDP receiver actions:

- 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

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)

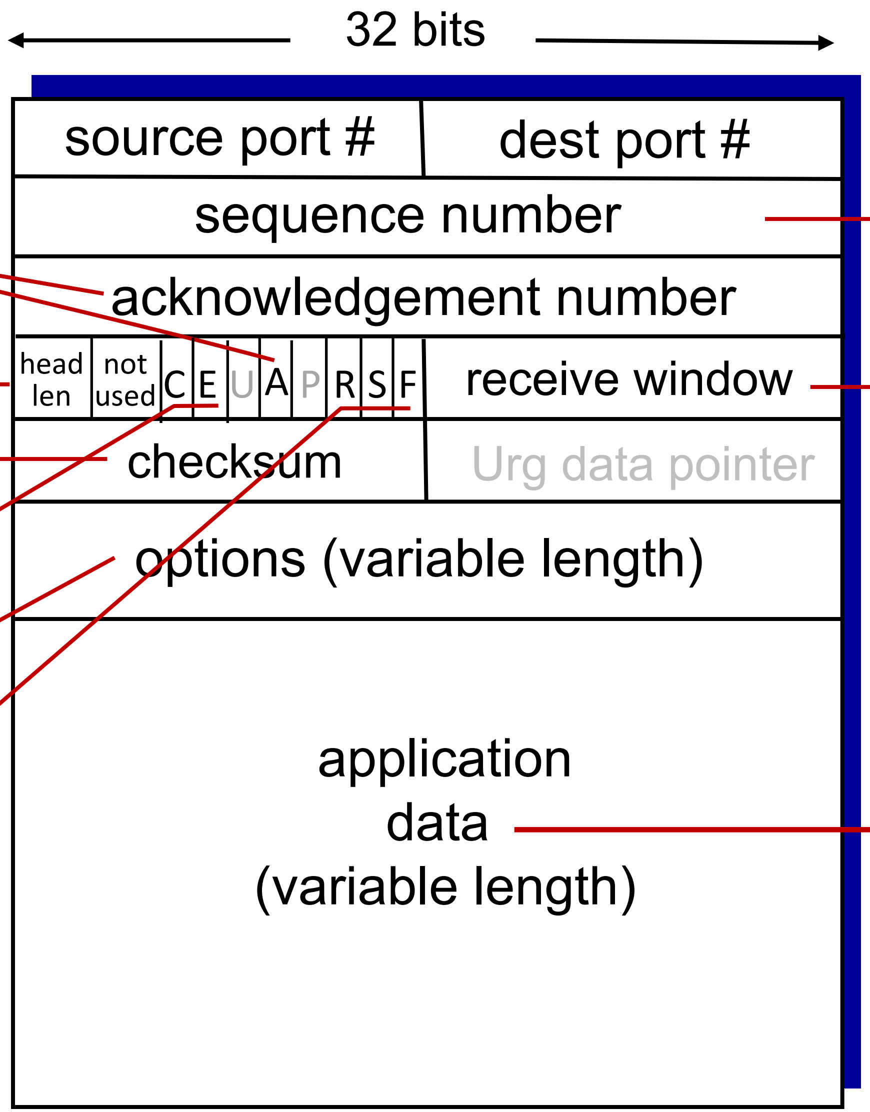
Transport layer: roadmap

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - flow control
  - connection management

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



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

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

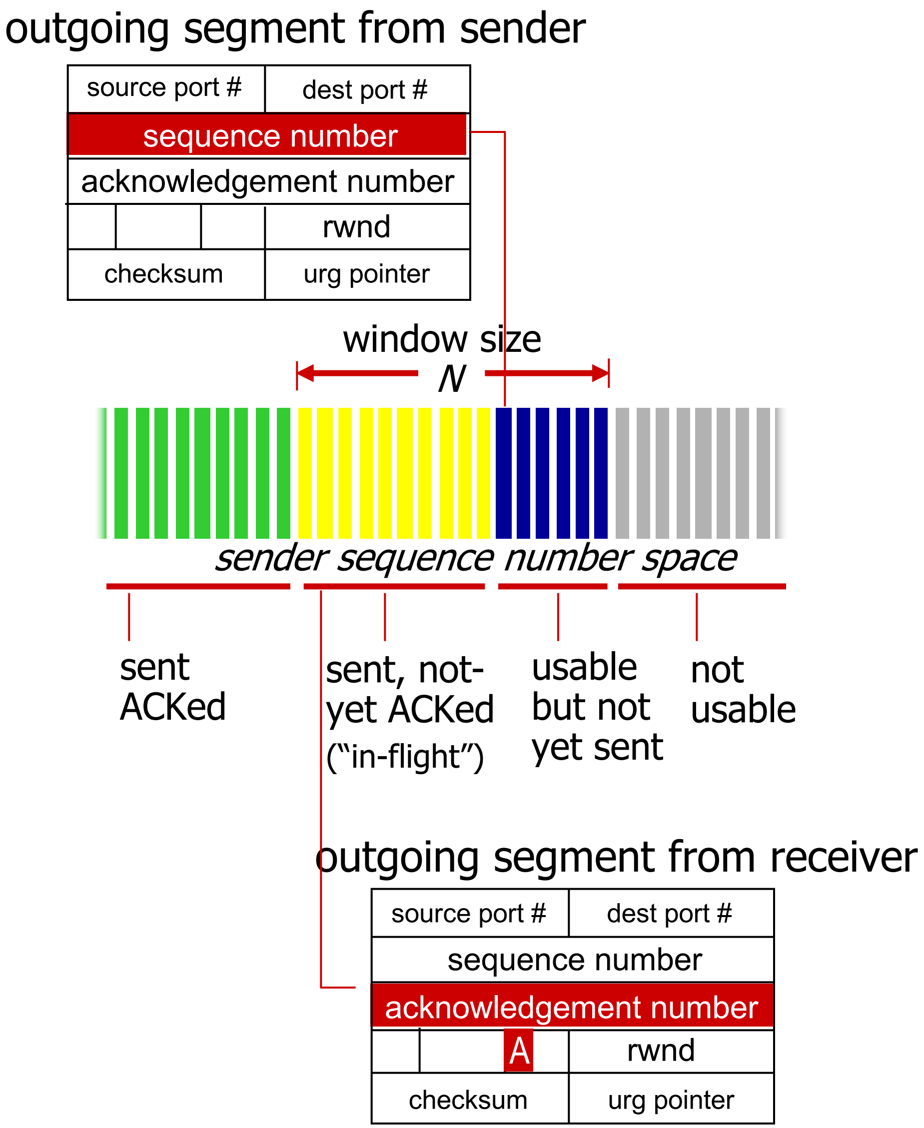
- byte stream "number" of first byte in segment's data

Acknowledgements:

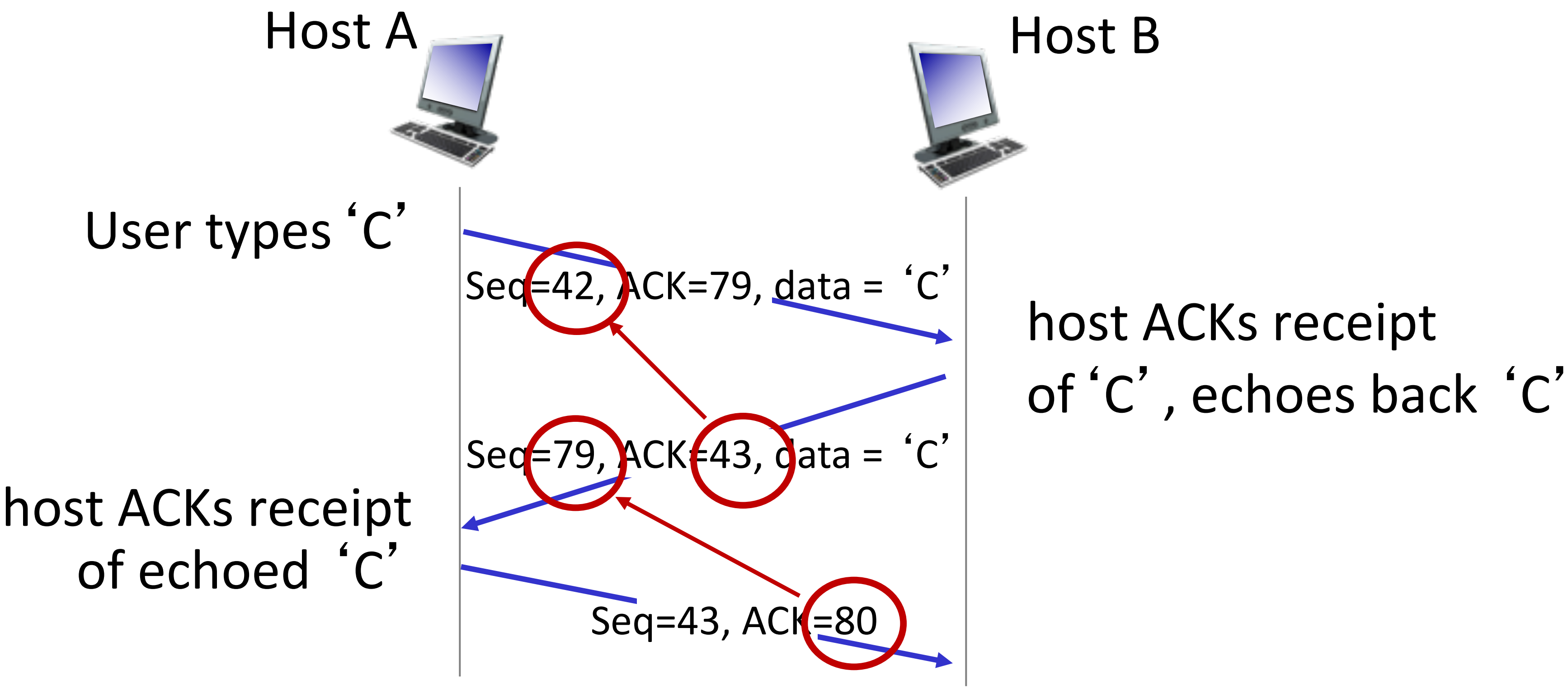
- seq # of next byte expected from other side
- cumulative ACK

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 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

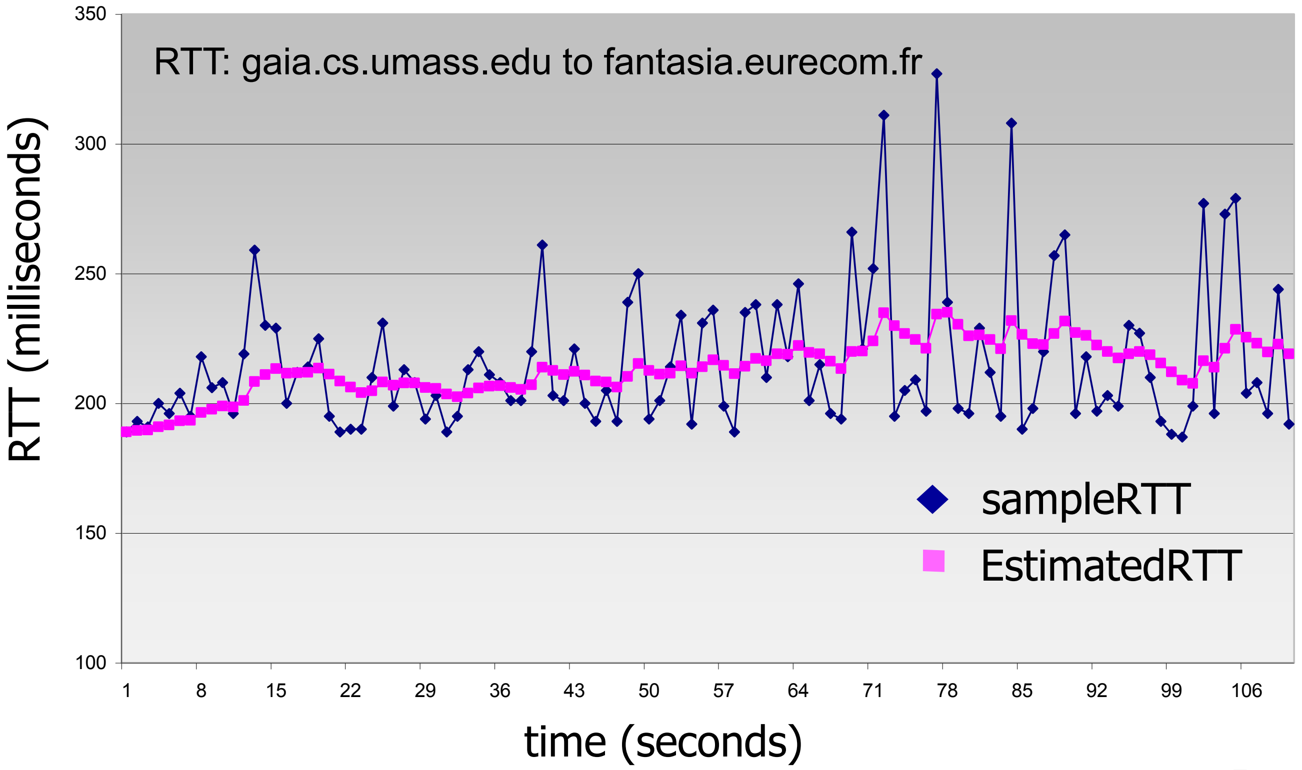
Q: how to estimate RTT?

- `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`

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

$$\texttt{TimeoutInterval} = \underbrace{\texttt{EstimatedRTT}}_{\text{estimated RTT}} + \underbrace{\texttt{4*DevRTT}}_{\text{"safety margin"}}$$

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

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

(typically, β = 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

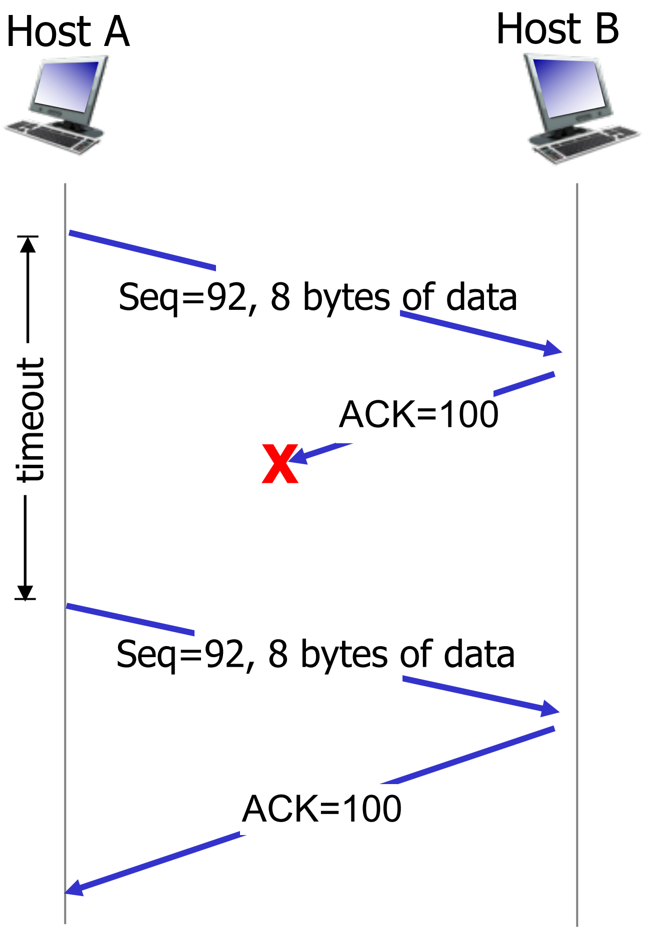
event: ACK received

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

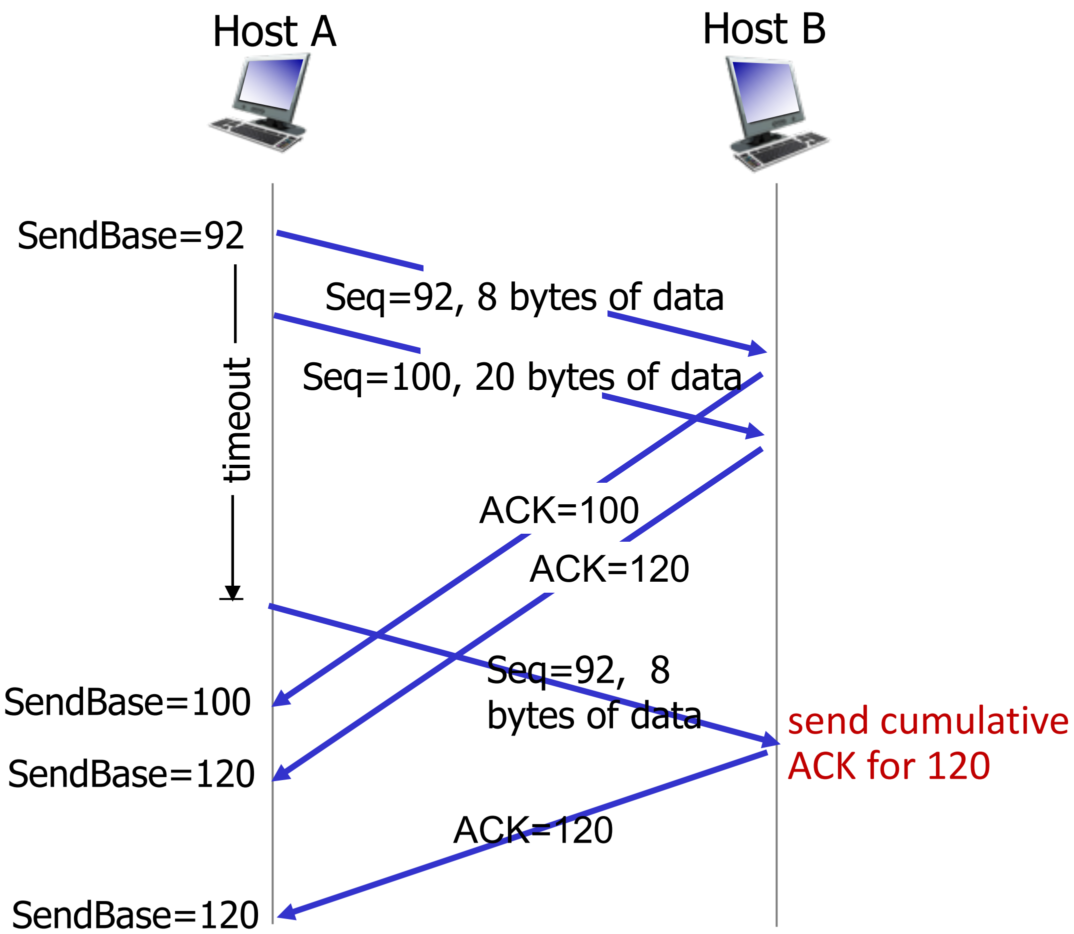
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*

TCP: retransmission scenarios

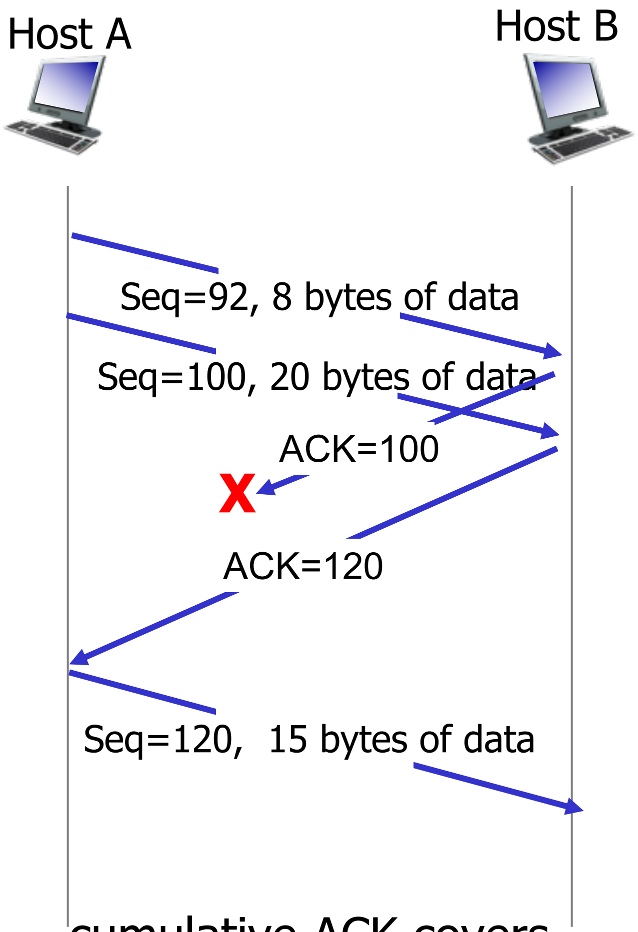


lost ACK scenario



premature timeout

TCP: retransmission scenarios



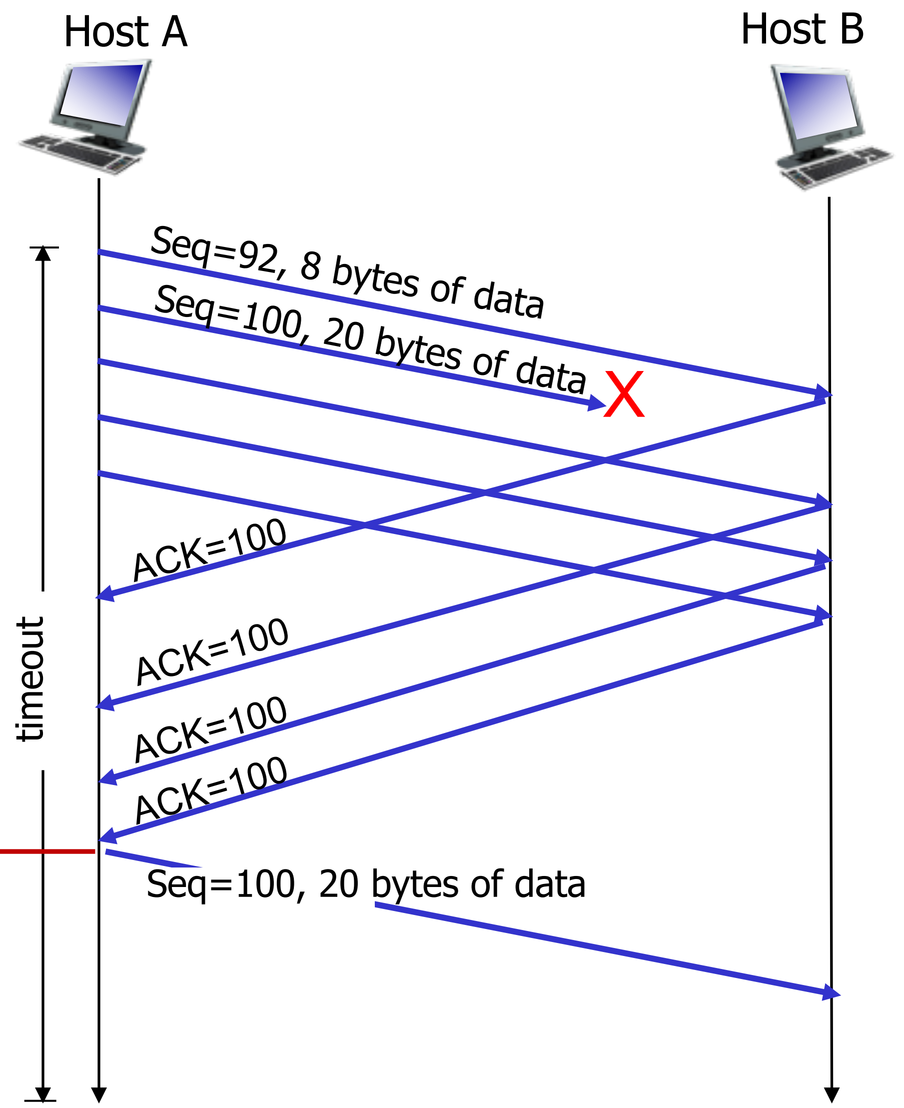
cumulative ACK covers for earlier lost ACK

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!

Transport layer : roadmap

- Transport-layer services
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - flow control
  - connection management

(congestion) control → (혼잡)

flow-control 과는 다른 개념.

↳ network에 데이터가 너무 몰려 있다.

congestion detection

" connection

데이터 못받다 → 혼잡이 생겨서 ① (congestion)

↳ 데이터 loss

↳ router에서 버려버림 (큐 buffer 초과)

(Ack이 제대로 안 옴)

처음 → 고통 발생

→ 처음 데이터 보내 봤자 제대로 못 간다~.

지수 → 일정 순간 부터는 linear 하게 증가시킴

(비트부터 점점 전송)

1개

2개

" 점차 많이 보냄 "

↓

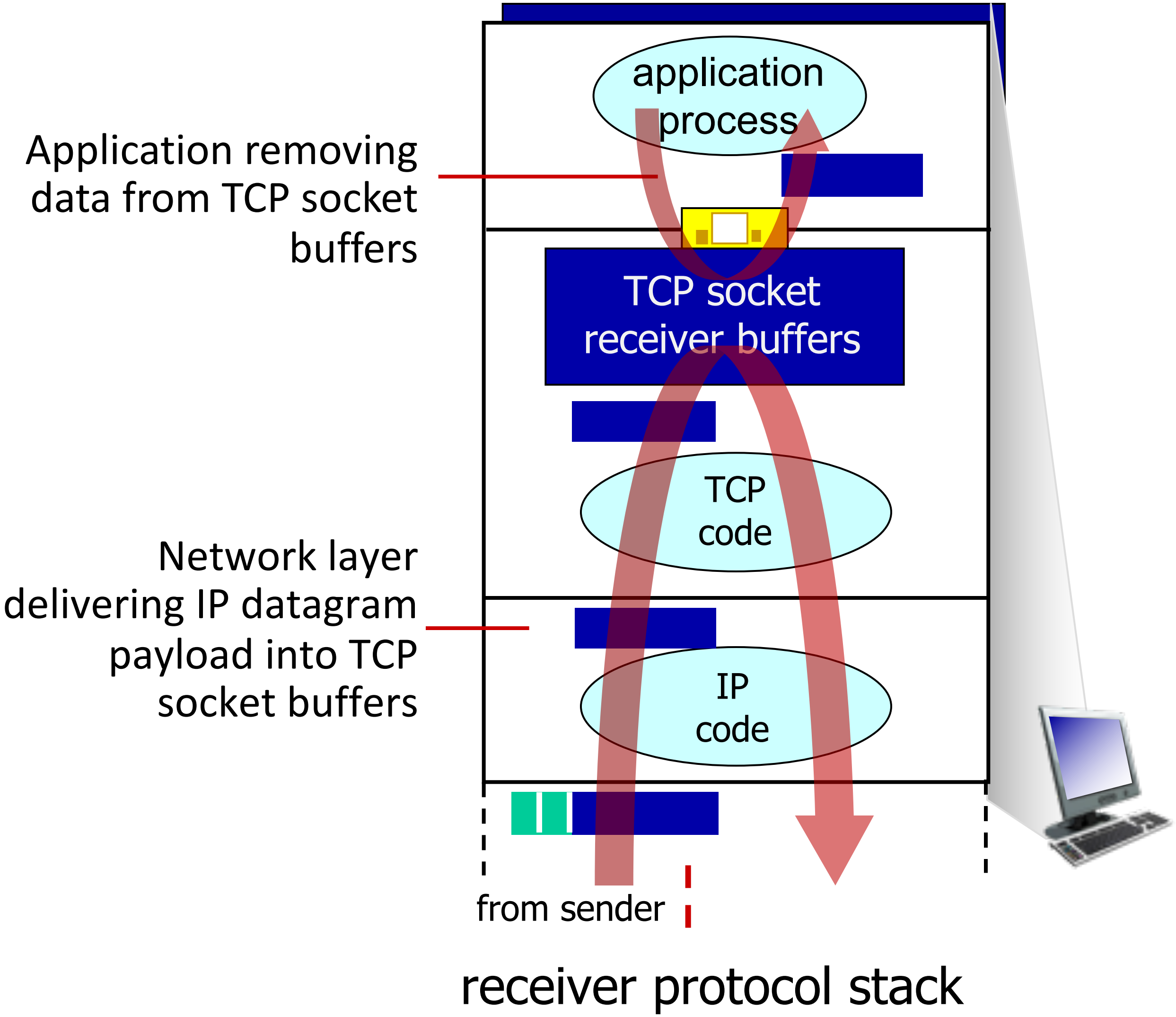
exponential 이하여 확 함.

상대측이 다시 돌려준다

(중요한 topic 인데도 너무 힘들다.)

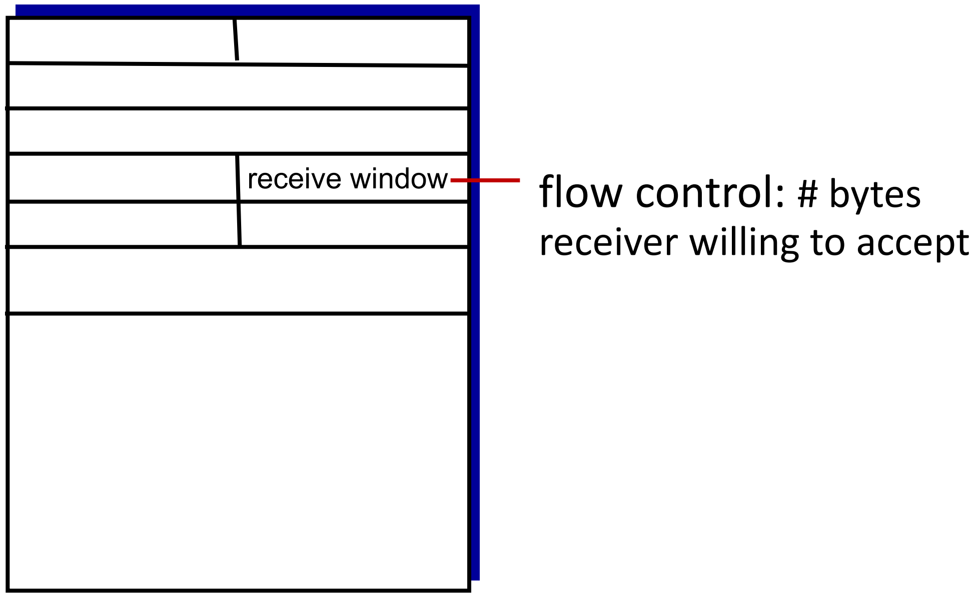
TCP flow control

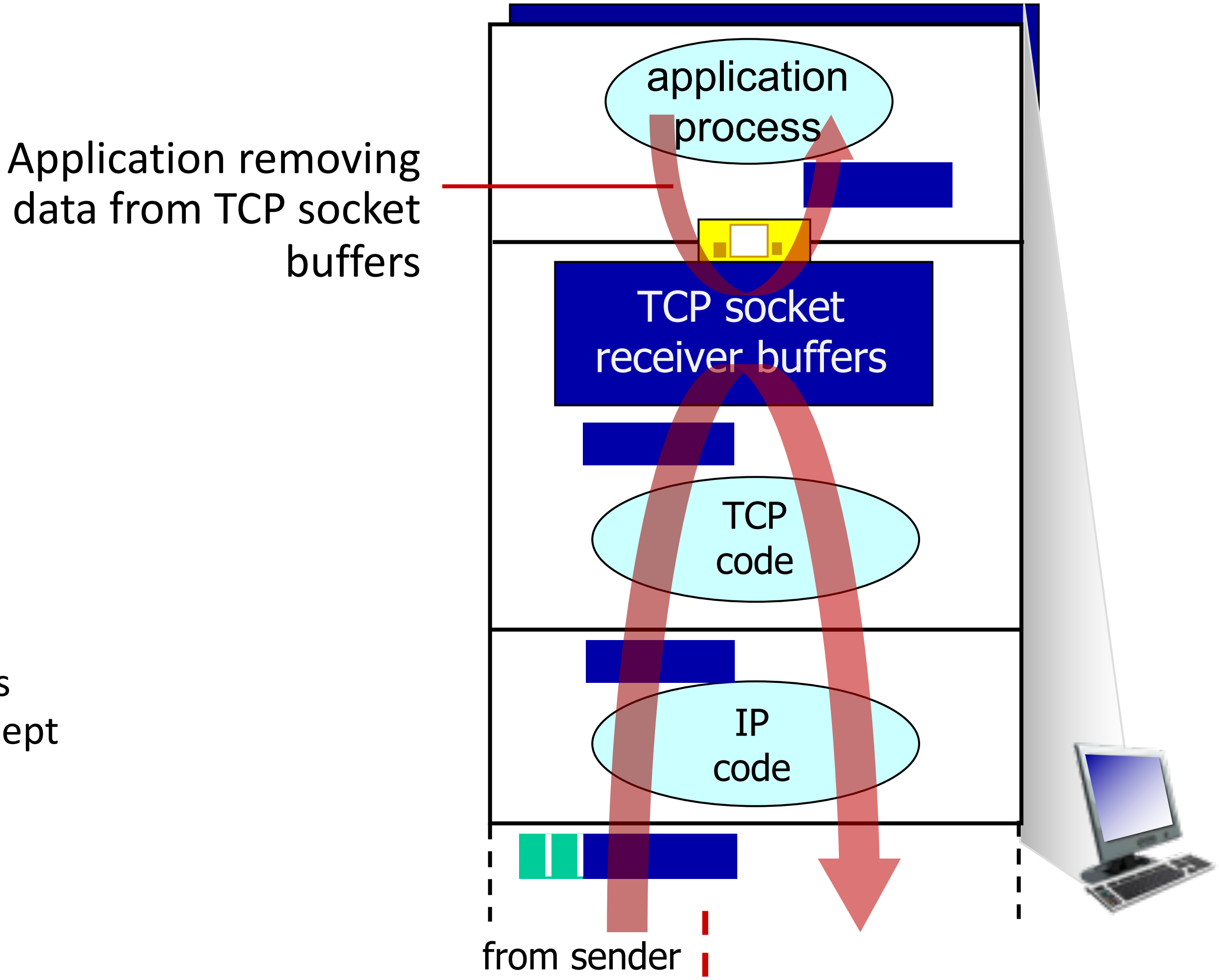
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP flow control

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

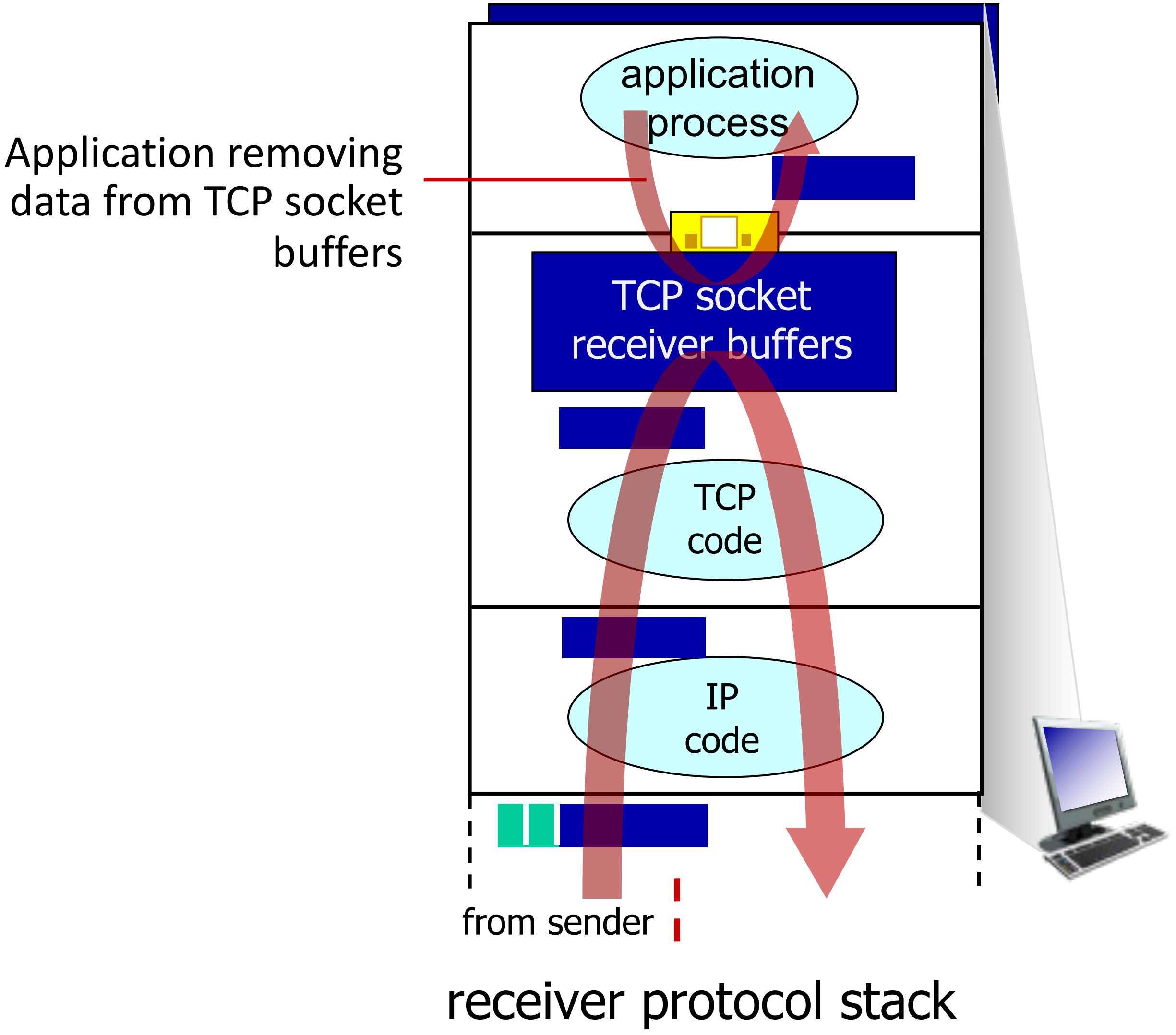




receiver protocol stack

TCP flow control

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

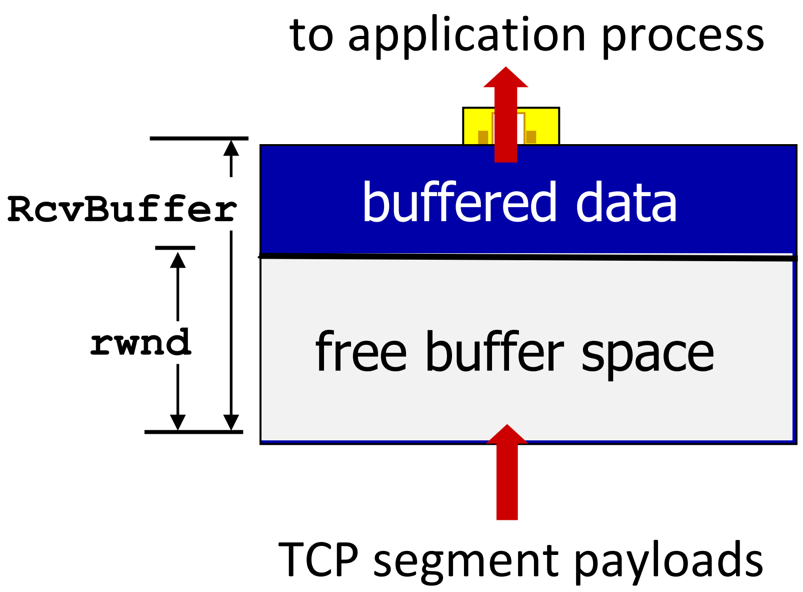


flow control

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

TCP flow control

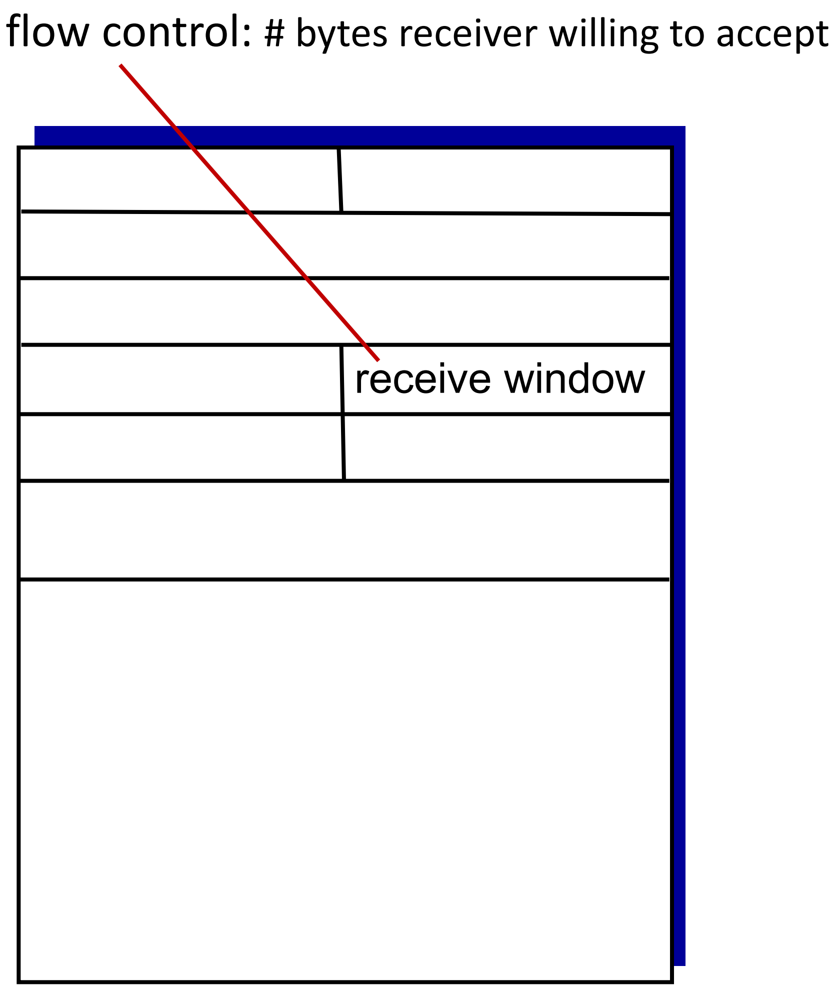
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees 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

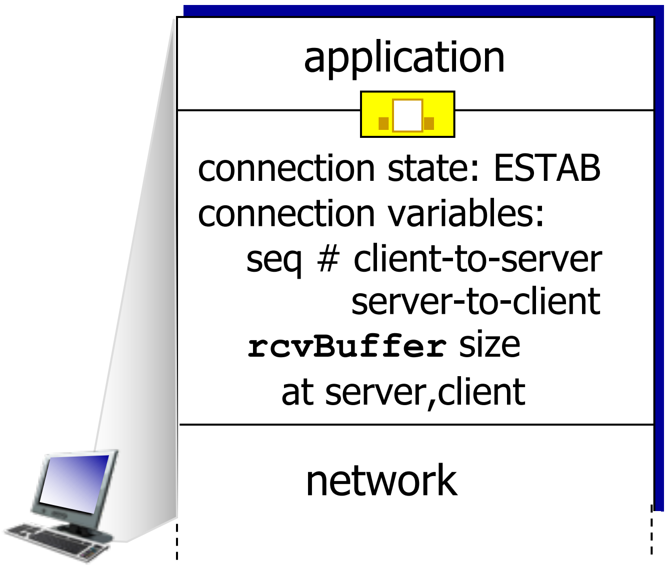


TCP segment format

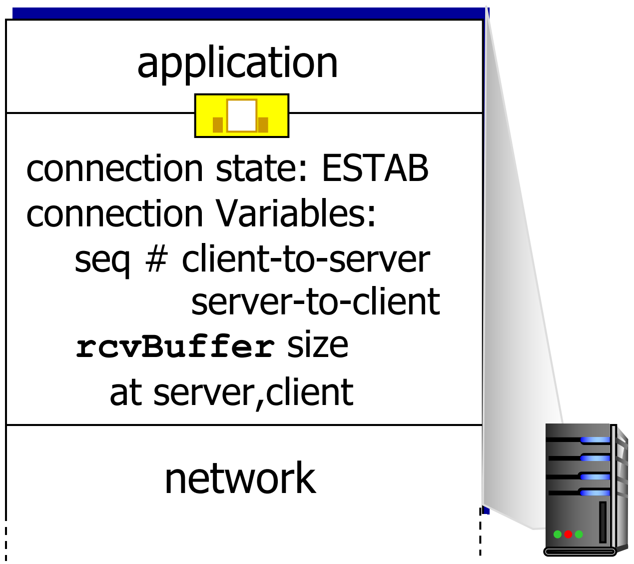
TCP connection management

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)



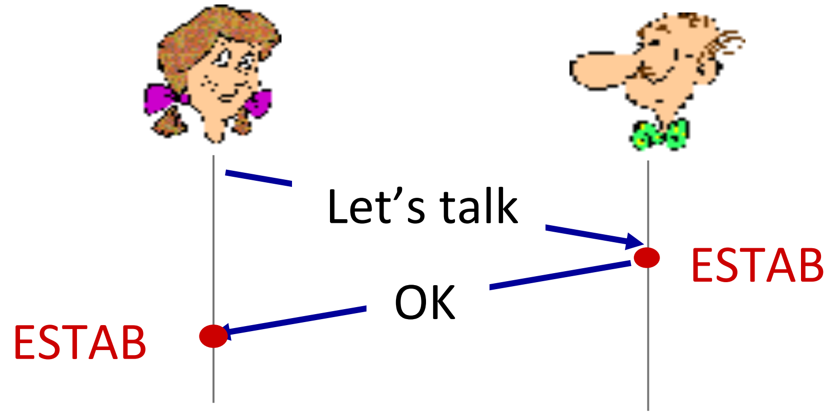

```
Socket clientSocket =
  newSocket("hostname","port number");
```

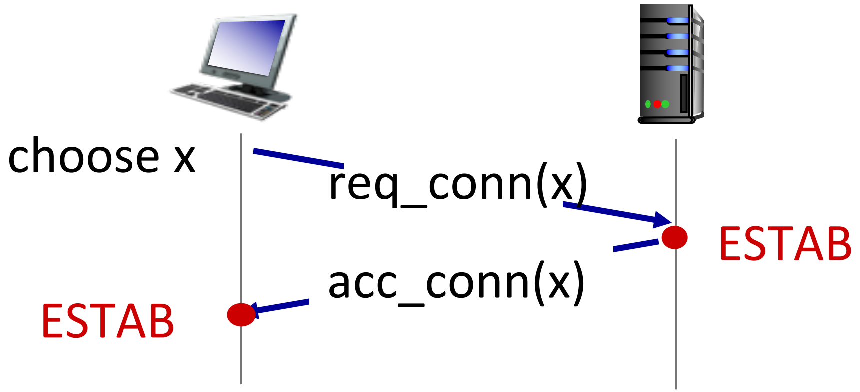



```
Socket connectionSocket =
  welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:

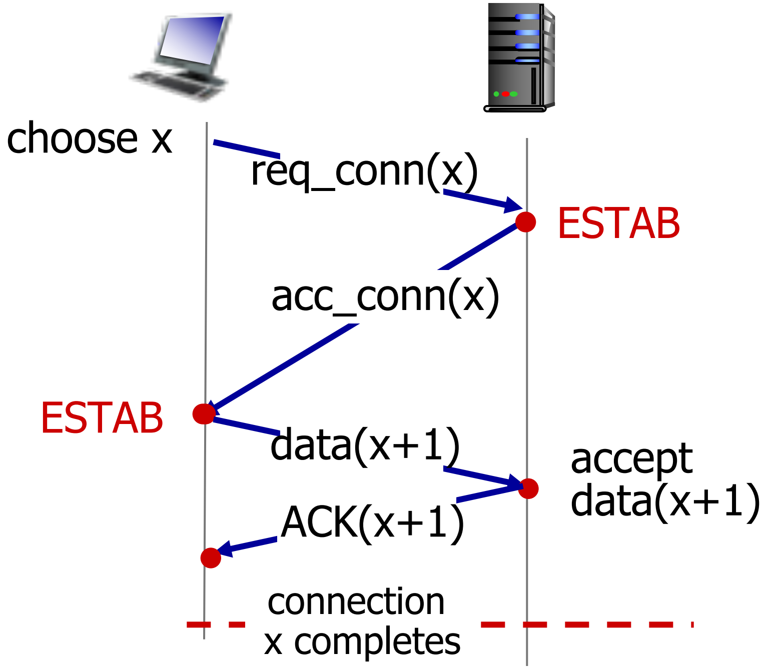




Q: will 2-way handshake always work in network?

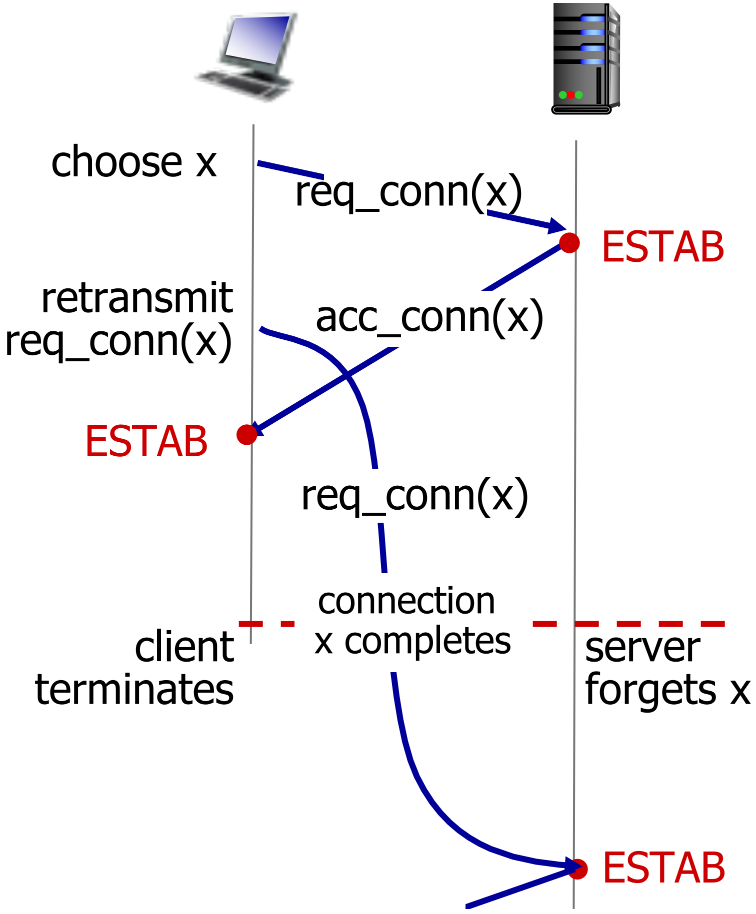
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

2-way handshake scenarios



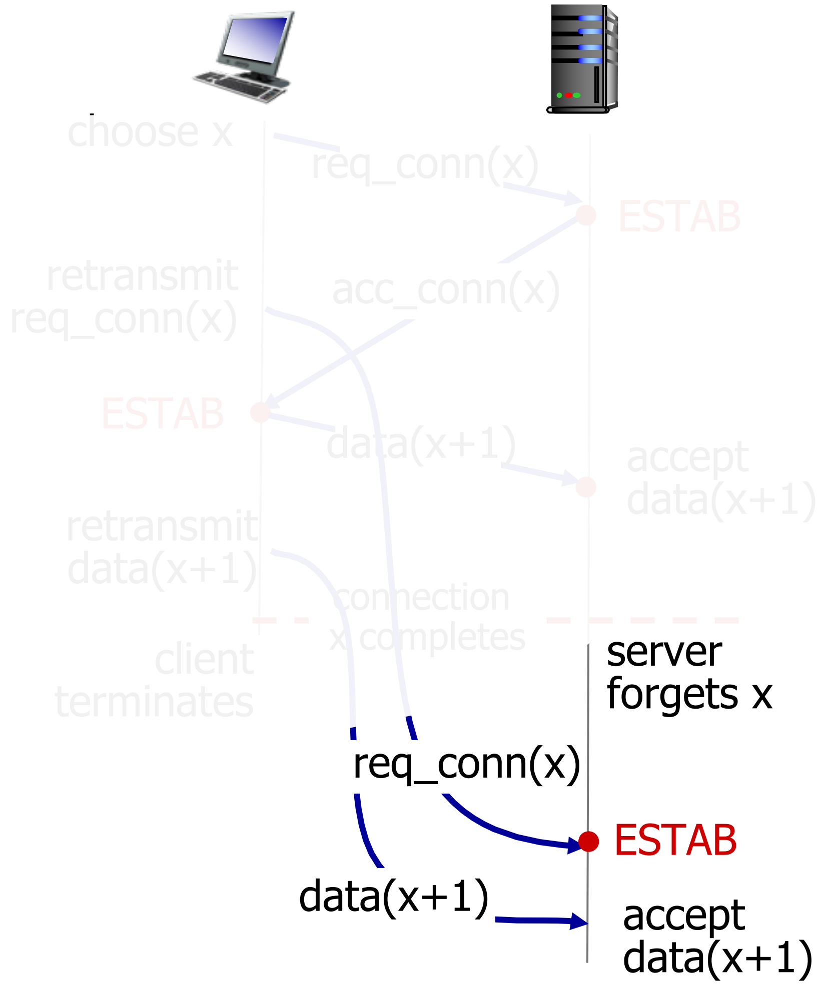
No problem!

2-way handshake scenarios



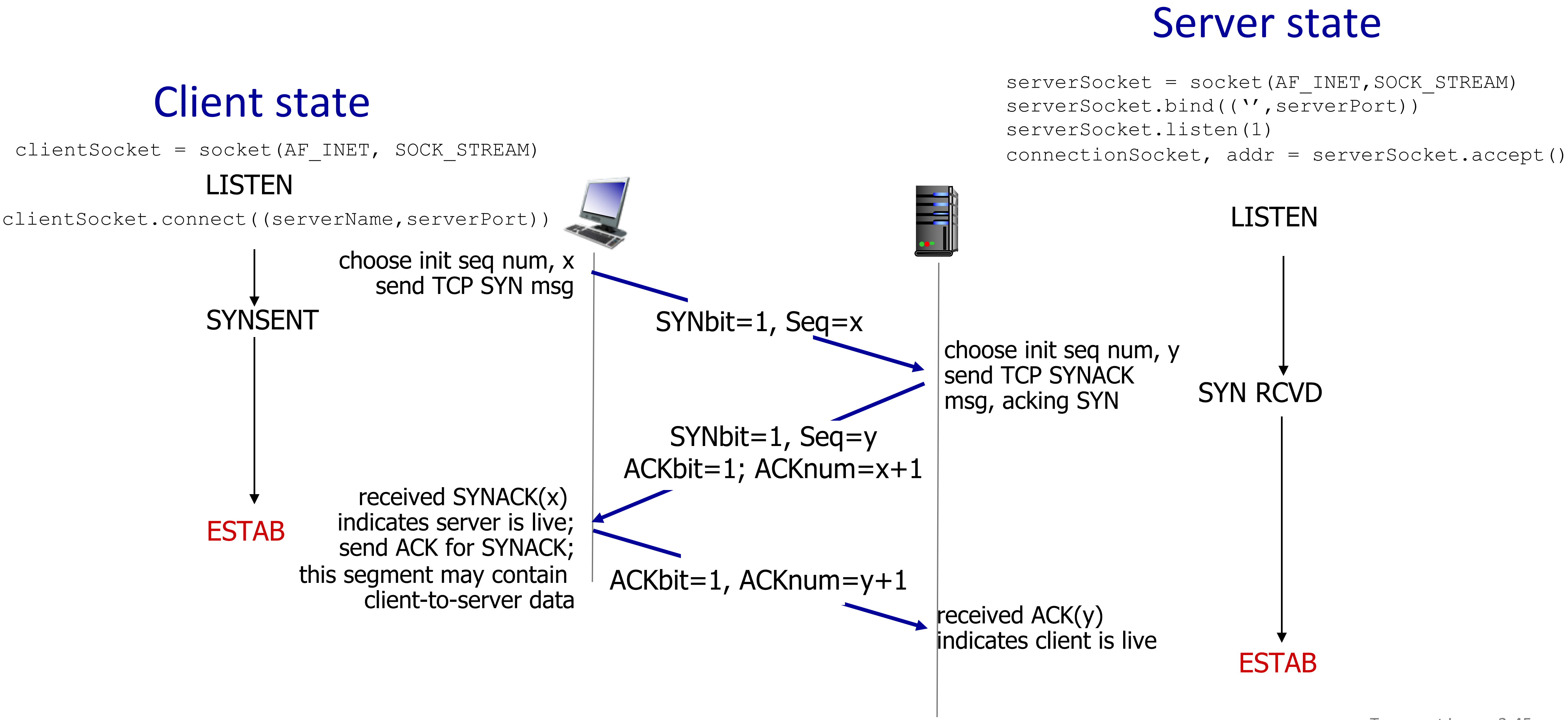
Problem: half open connection! (no client)

2-way handshake scenarios



Problem: dup data accepted!

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

Transport layer: summary

- principles behind transport layer services:
  - flow control
  - Connection management
- instantiation, implementation in the Internet
  - UDP
  - TCP

오류임. 못 전달해 주겠다.

Network Layer: Internet

IP 참고 → 어떤 구조인지

host, router network layer functions:

transport layer: TCP, UDP

network layer

Path-selection algorithms: implemented in
- routing protocols (OSPF, BGP)
- SDN controller

forwarding table

IP protocol
- datagram format
- addressing
- packet handling conventions

ICMP protocol
- error reporting
- router "signaling"

network에서 송신자에게 ~ 이유 때문에 못 보냈어!

link layer

physical layer

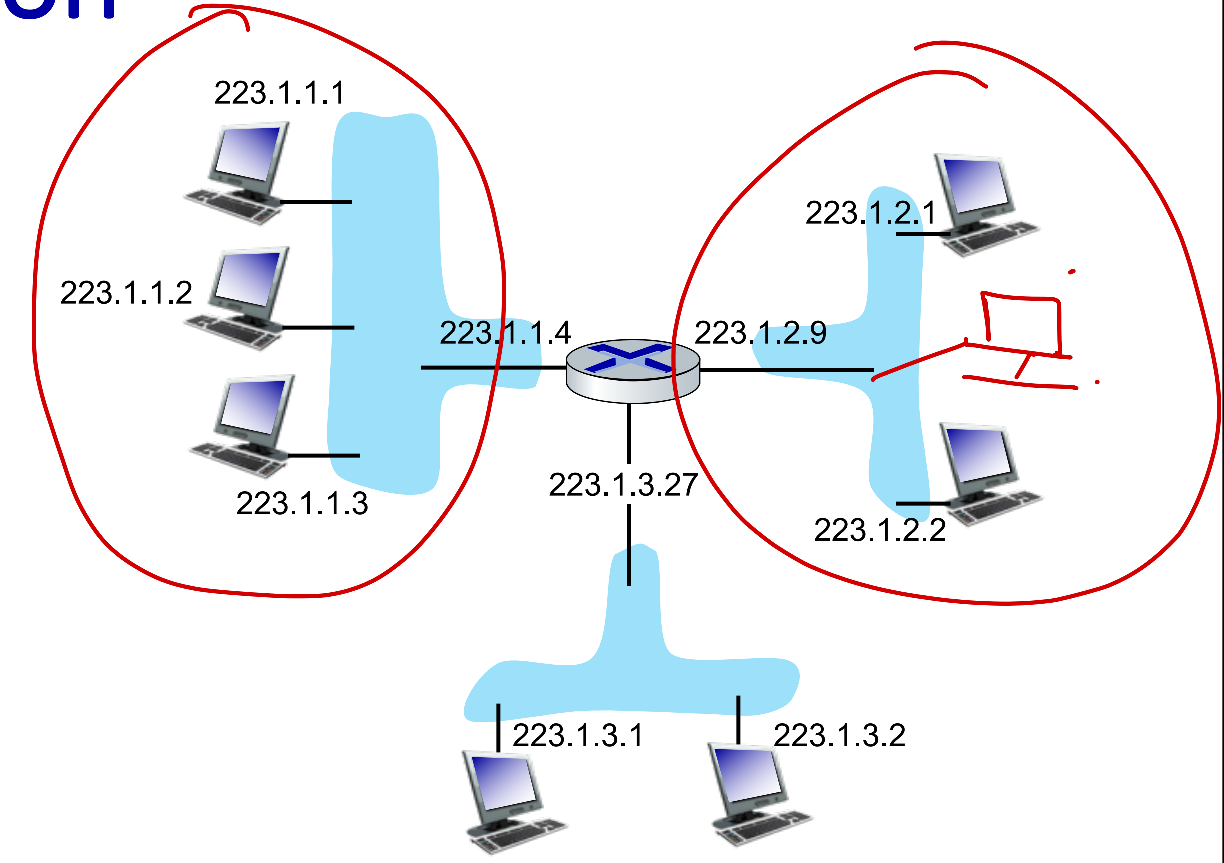
IP addressing: introduction

- IP address: 32-bit identifier associated with each host or router *interface*
- interface: connection between host/router and physical link
  - router's typically have multiple interfaces
  - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)

155.230 | 90.8 → 경북대 안의 host

경북대

network-ID host-ID



dotted-decimal IP address notation:

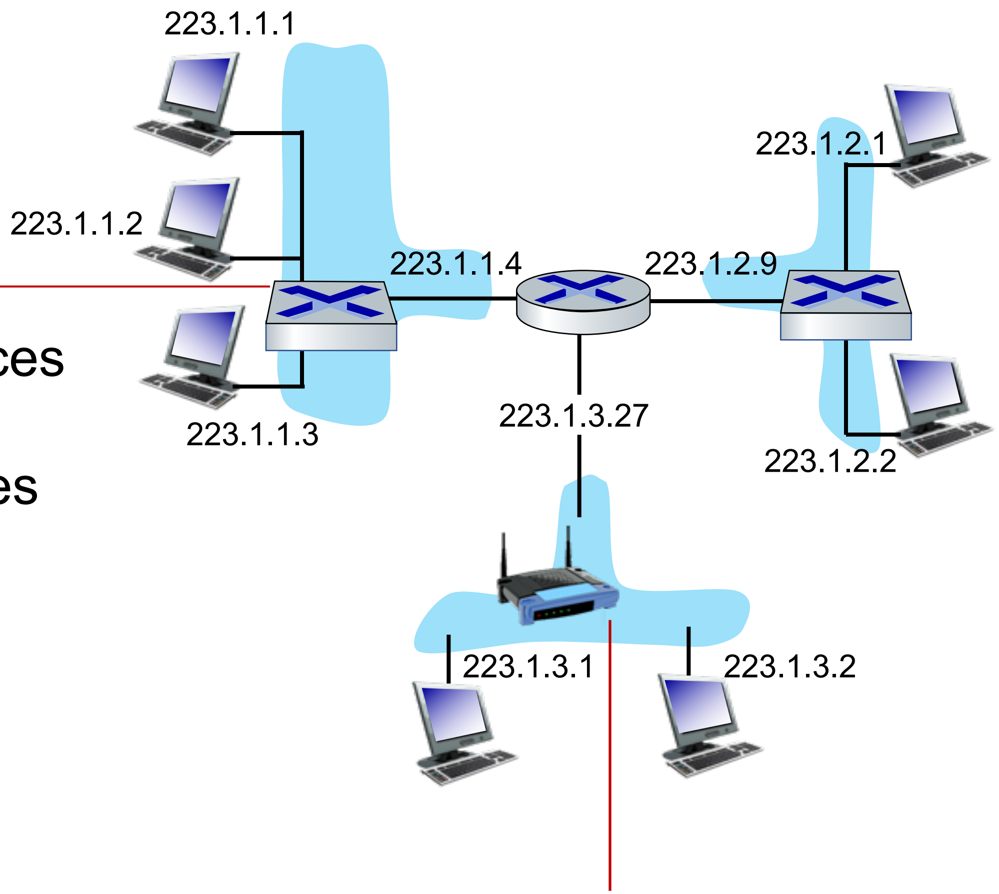
$$223.1.1.1 = \underbrace{11011111}_{223}\ \underbrace{00000001}_{1}\ \underbrace{00000001}_{1}\ \underbrace{00000001}_{1}$$

IP addressing: introduction

Q: how are interfaces actually connected?

A: we'll learn about that in chapters 6, 7

A: wired Ethernet interfaces connected by Ethernet switches

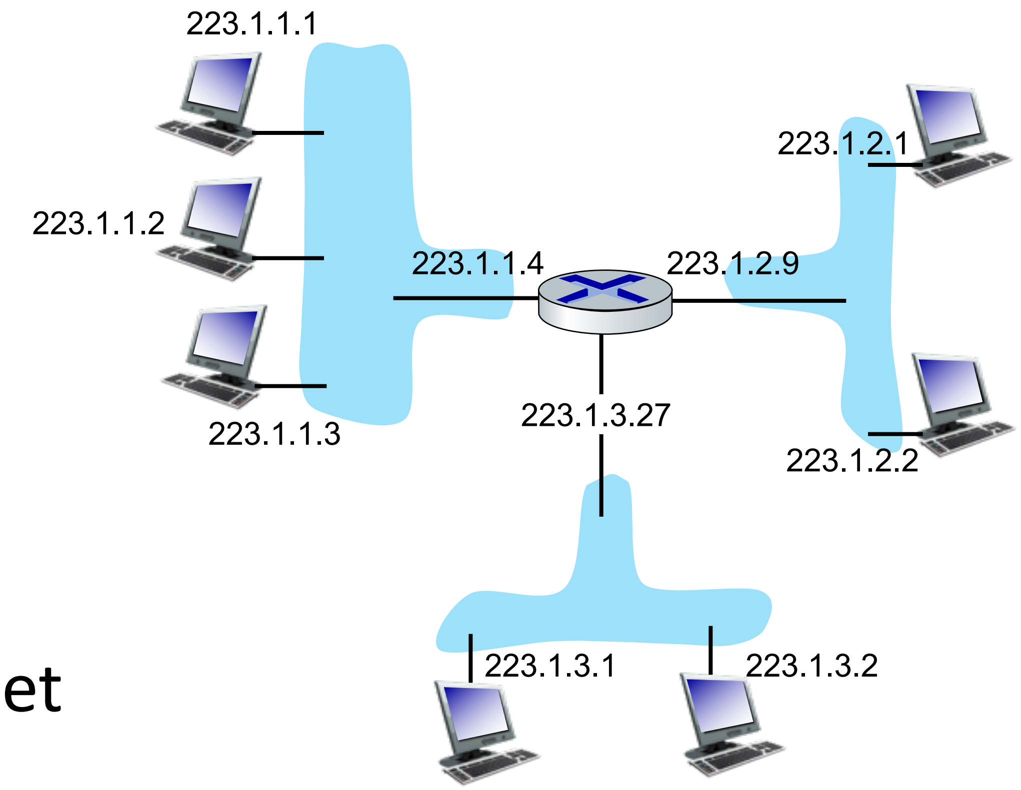


For now: don't need to worry about how one interface is connected to another (with no intervening router)

A: wireless WiFi interfaces connected by WiFi base station

Subnets

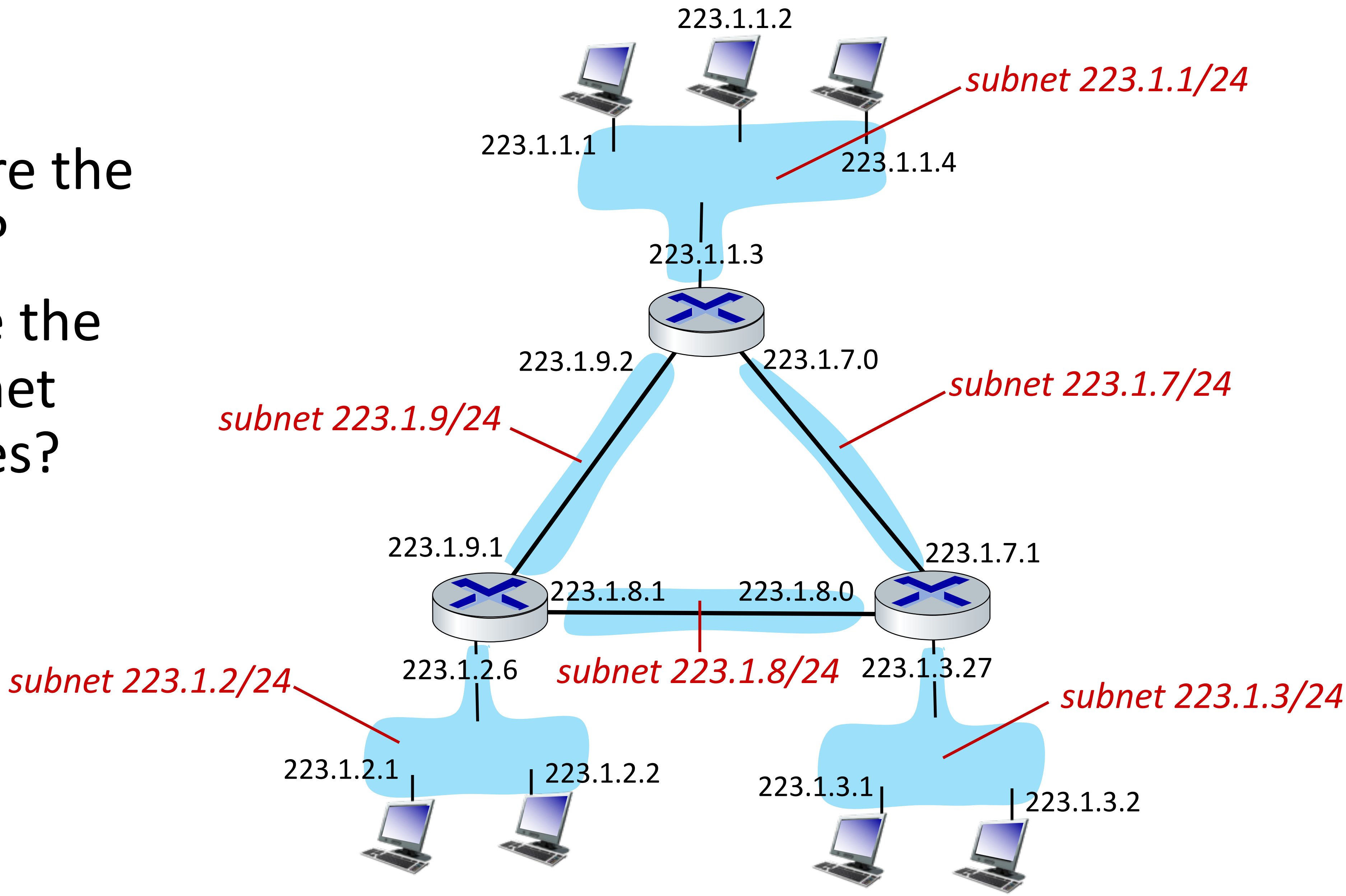
- *What's a subnet ?*
  - device interfaces that can physically reach each other without passing through an intervening router
- IP addresses have structure:
  - subnet part: devices in same subnet have common high order bits
  - host part: remaining low order bits



network consisting of 3 subnets

Subnets

- where are the subnets?
- what are the /24 subnet addresses?



IP addressing: CIDR

CIDR: Classless InterDomain Routing (pronounced "cider")

- subnet portion of address of arbitrary length
- address format: a.b.c.d/x, where x is # bits in subnet portion of address

subnet part ← → host part

11001000 00010111 00010000 00000000

200.23.16.0/23

IP addresses: how to get one?

That's actually two questions:

1. Q: How does a *host* get IP address within its network (host part of address)?
2. Q: How does a *network* get IP address for itself (network part of address)

How does *host* get IP address?

- hard-coded by sysadmin in config file (e.g., /etc/rc.config in UNIX)
- DHCP: Dynamic Host Configuration Protocol: dynamically get address from as server
  - "plug-and-play"

DHCP: Dynamic Host Configuration Protocol

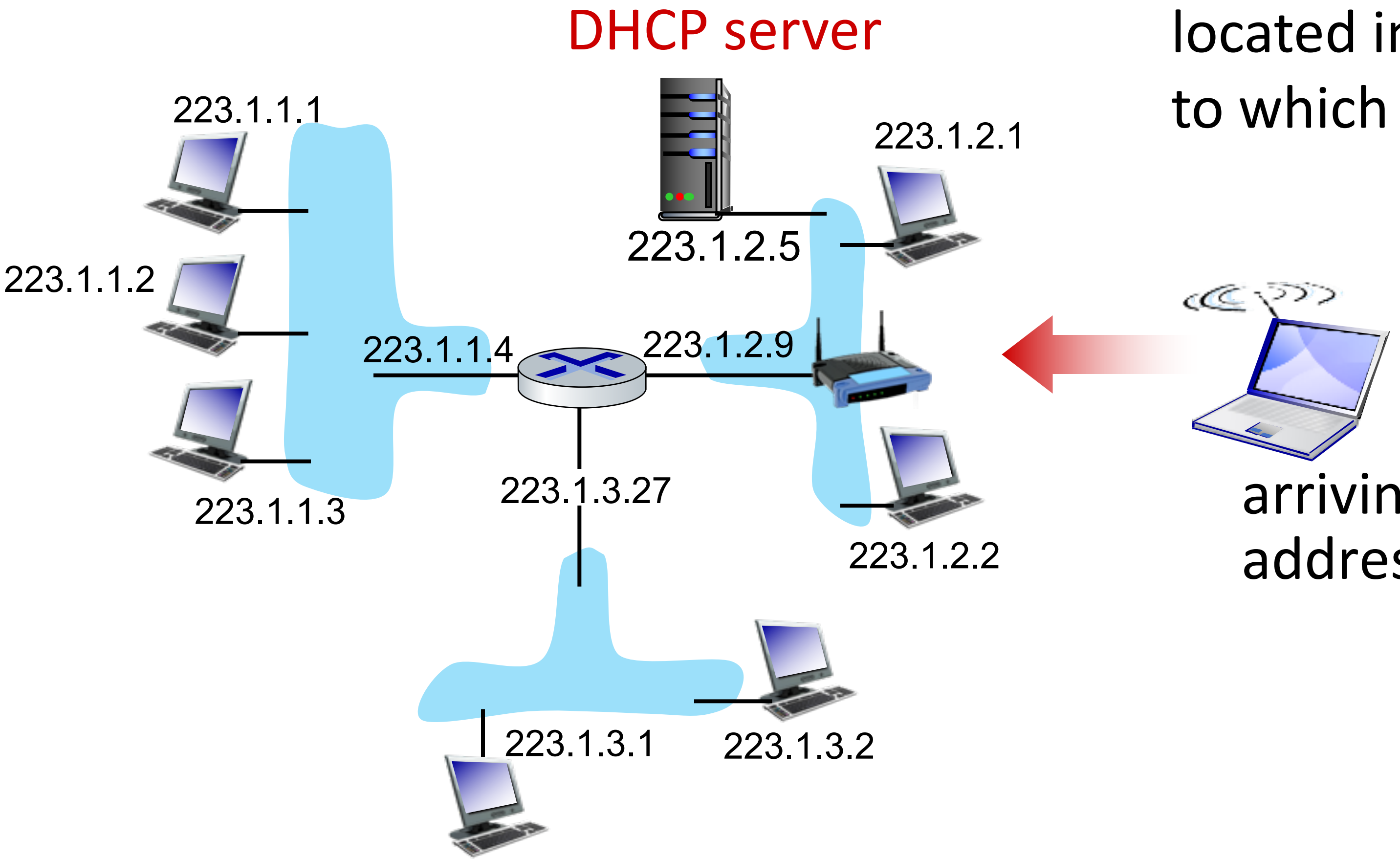
goal: host *dynamically* obtains IP address from network server when it "joins" network

- can renew its lease on address in use
- allows reuse of addresses (only hold address while connected/on)
- support for mobile users who join/leave network

DHCP overview:

- host broadcasts DHCP discover msg [optional]
- DHCP server responds with DHCP offer msg [optional]
- host requests IP address: DHCP request msg
- DHCP server sends address: DHCP ack msg

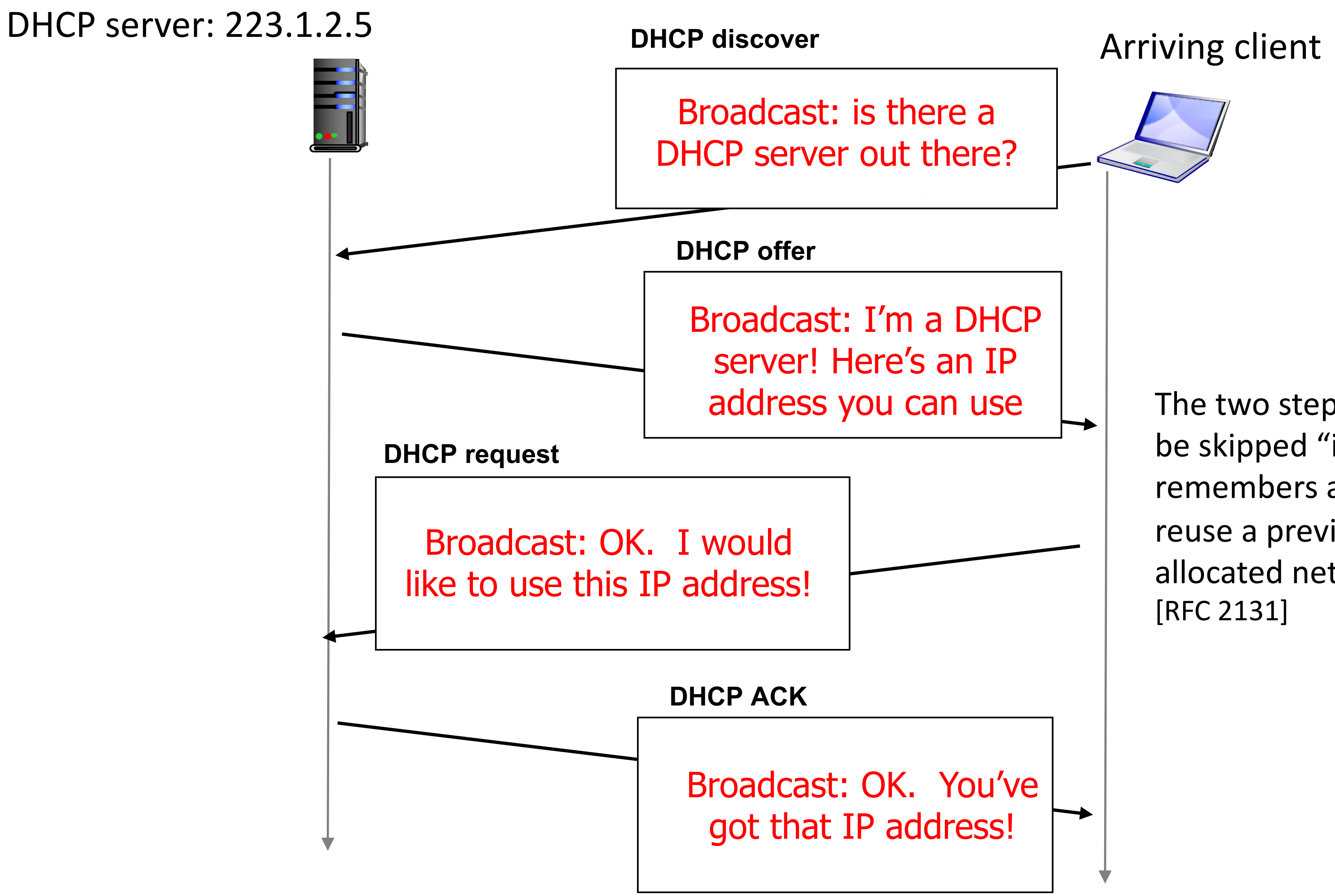
DHCP client-server scenario



Typically, DHCP server will be co-located in router, serving all subnets to which router is attached

arriving DHCP client needs address in this network

DHCP client-server scenario

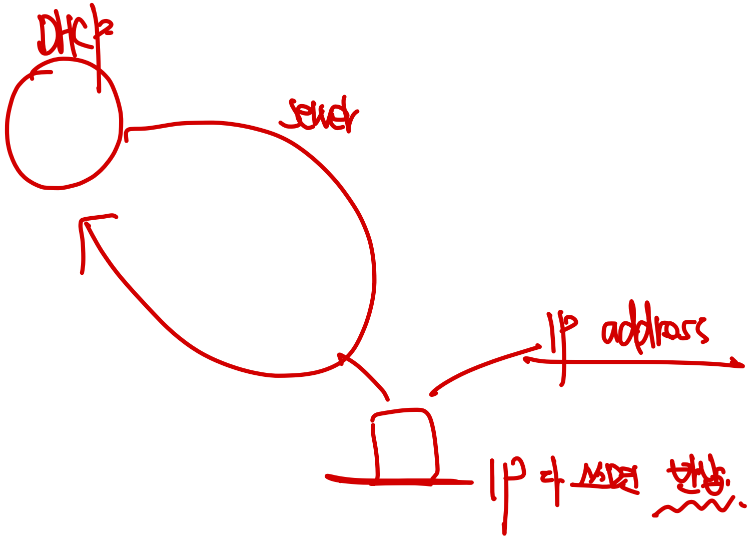


The two steps above can be skipped "if a client remembers and wishes to reuse a previously allocated network address" [RFC 2131]

DHCP: more than IP addresses

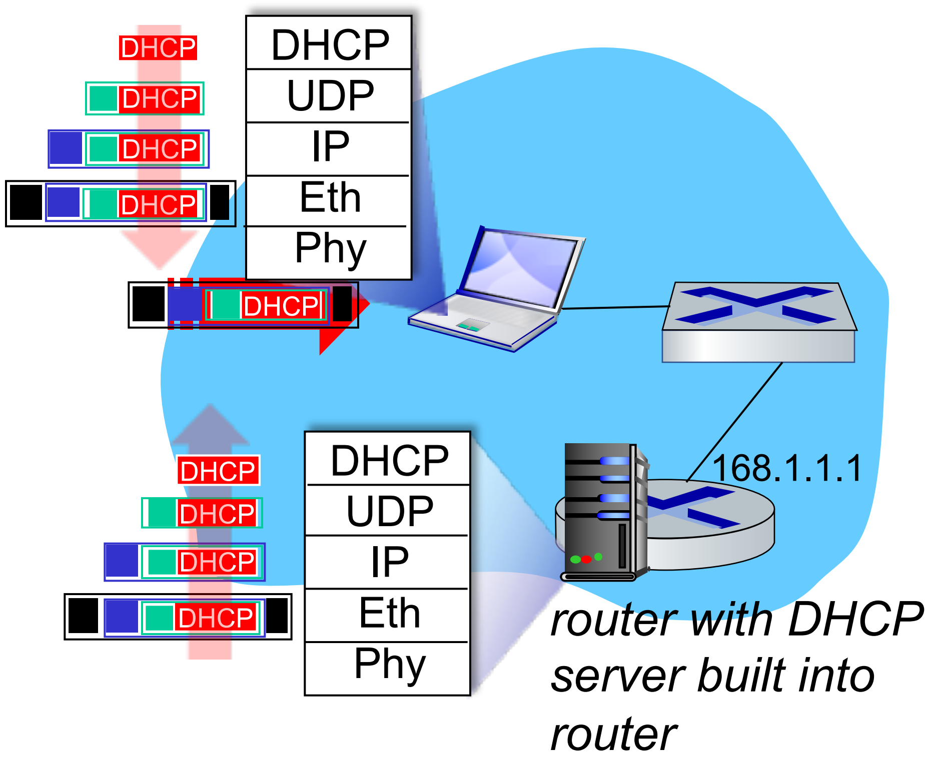
DHCP can return more than just allocated IP address on subnet:

- address of first-hop router for client
- name and IP address of DNS sever
- network mask (indicating network versus host portion of address)



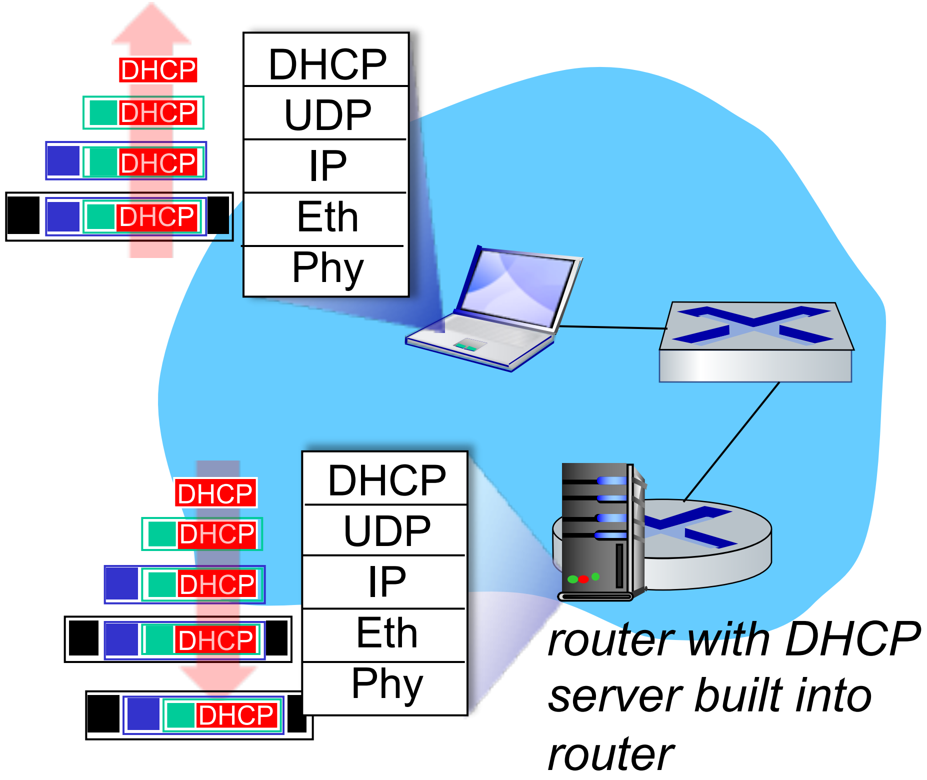
manual 하게 configuration

DHCP: example



- Connecting laptop will use DHCP to get IP address, address of first-hop router, address of DNS server.
- DHCP REQUEST message encapsulated in UDP, encapsulated in IP, encapsulated in Ethernet
- Ethernet frame broadcast (dest: FFFFFFFFFFFF) on LAN, received at router running DHCP server
- Ethernet demux'ed to IP demux'ed, UDP demux'ed to DHCP

DHCP: example



- DCP server formulates DHCP ACK containing client's IP address, IP address of first-hop router for client, name & IP address of DNS server
- encapsulated DHCP server reply forwarded to client, demuxing up to DHCP at client
- client now knows its IP address, name and IP address of DNS server, IP address of its first-hop router

IPv6: motivation

- initial motivation: 32-bit IPv4 address space would be completely allocated
- additional motivation:
  - speed processing/forwarding: 40-byte fixed length header
  - enable different network-layer treatment of “flows”

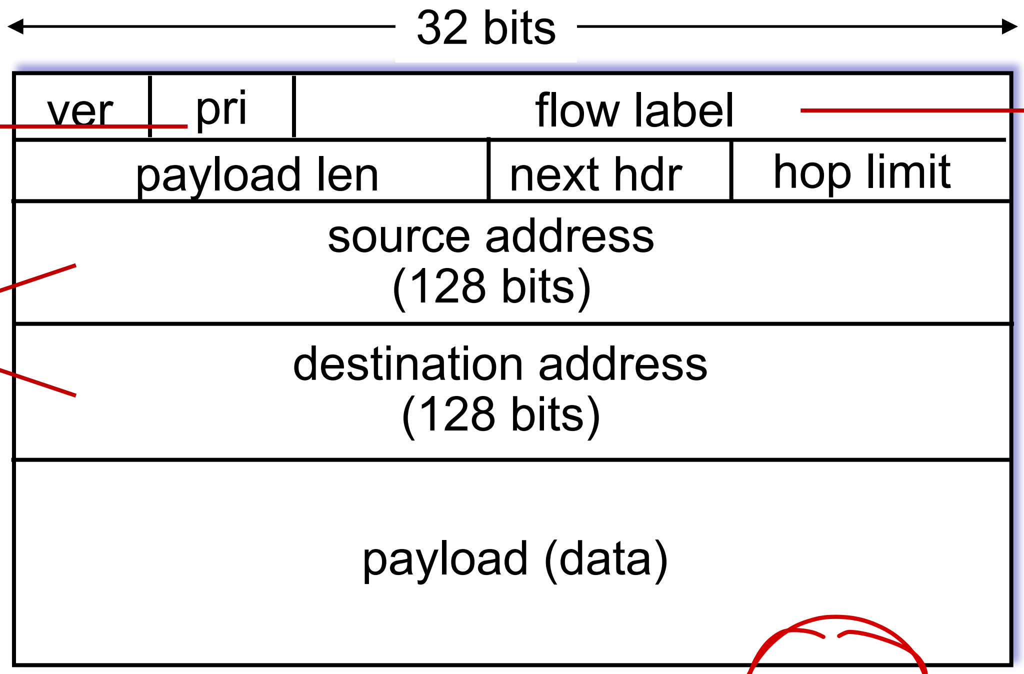
IPv6 datagram format

header format:

priority: identify priority among datagrams in flow

128-bit IPv6 addresses

DNS 끼리 물어 가면서. IP 주소를 찾음



flow label: identify datagrams in same "flow." (concept of "flow" not well defined).

IoT

온갖 사물이 인터넷과.

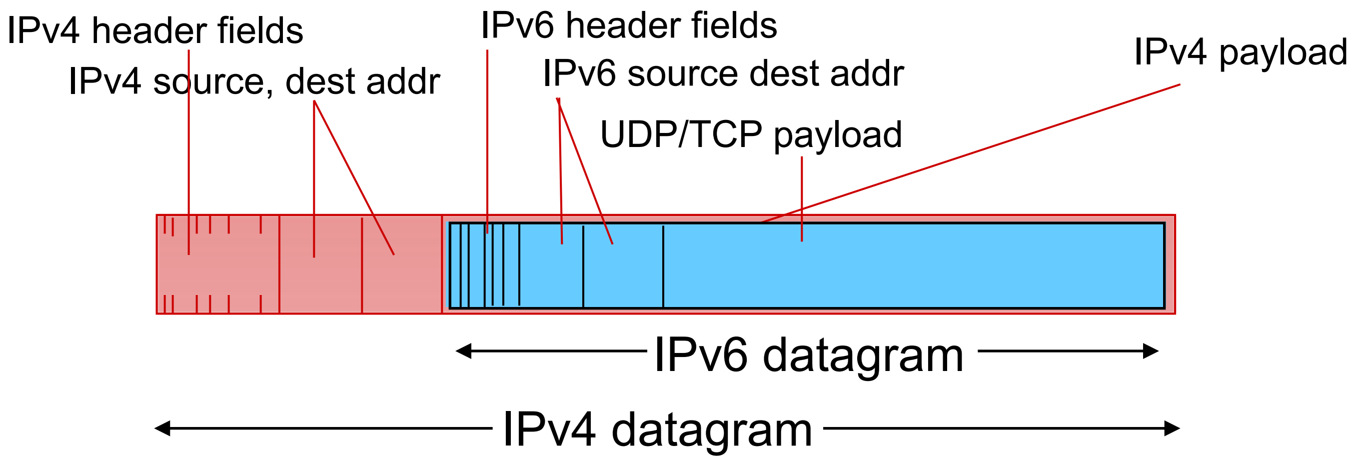
Evolution

What's missing (compared with IPv4):

- no checksum (to speed processing at routers)
- no fragmentation/reassembly
- no options (available as upper-layer, next-header protocol at router)

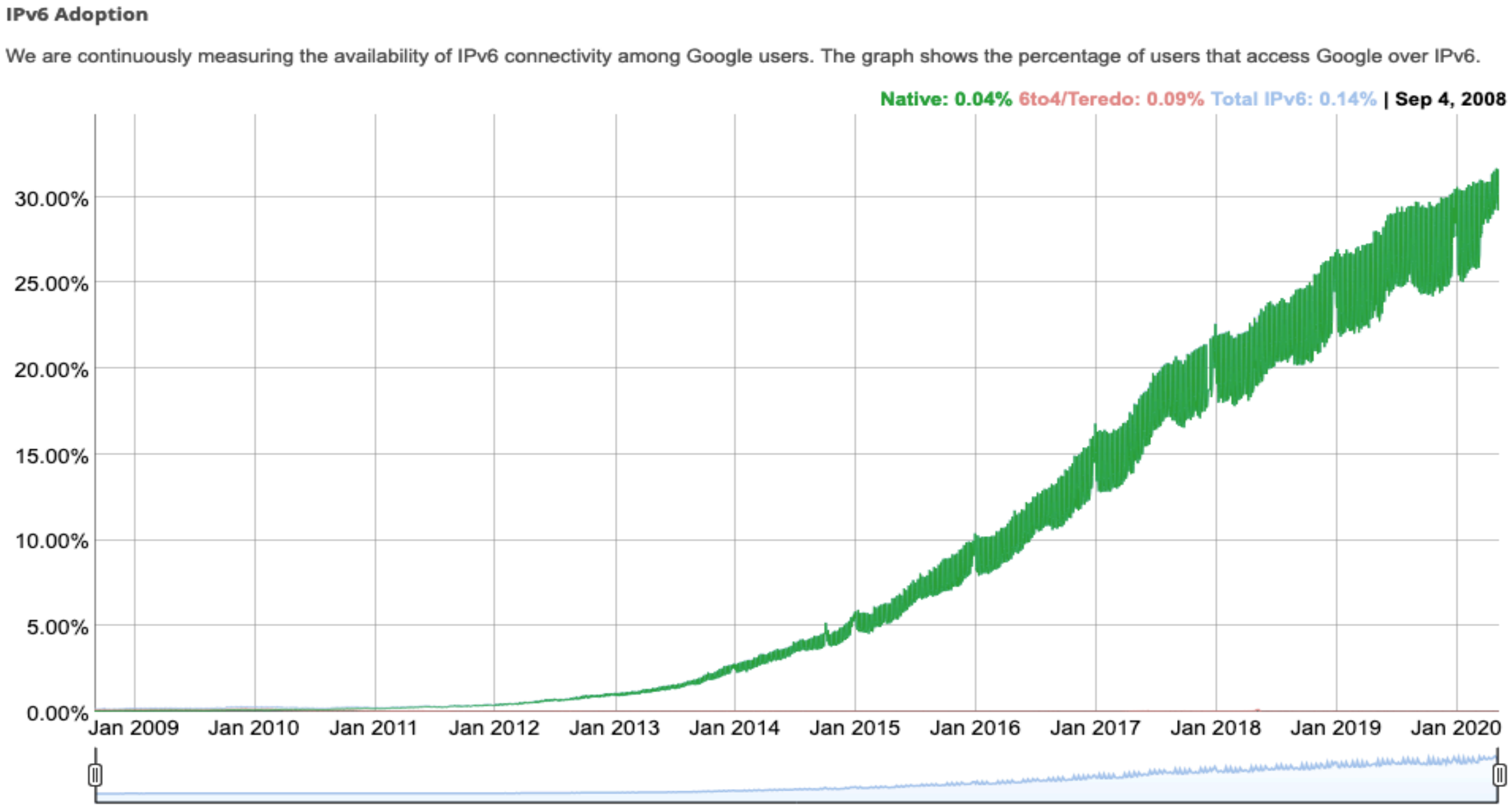
Transition from IPv4 to IPv6

- not all routers can be upgraded simultaneously
  - no "flag days"
  - how will network operate with mixed IPv4 and IPv6 routers?
- tunneling: IPv6 datagram carried as *payload* in IPv4 datagram among IPv4 routers ("packet within a packet")
  - tunneling used extensively in other contexts (4G/5G)



IPv6: adoption

- Google[1]: ~ 30% of clients access services via IPv6
- NIST: 1/3 of all US government domains are IPv6 capable



1 https://www.google.com/intl/en/ipv6/statistics.html

IPv6: adoption

- Google[1]: ~ 30% of clients access services via IPv6
- NIST: 1/3 of all US government domains are IPv6 capable
- Long (long!) time for deployment, use
  - 25 years and counting!
  - think of application-level changes in last 25 years: WWW, social media, streaming media, gaming, telepresence..

[1] https://www.google.com/intl/en/ipv6/statistics.html