

# Chapter 1: introduction

## *Chapter goal:*

- Get “feel,” “big picture,” introduction to terminology
  - more depth, detail *later* in course
- Approach:
  - use Internet as example



## *Overview/roadmap:*

- What *is* the Internet?
- What *is* a protocol?
- **Network edge:** hosts, access network, physical media
- **Network core:** packet/circuit switching, internet structure
- **Performance:** loss, delay, throughput
- Protocol layers

# The Internet: a “nuts and bolts” view



Billions of connected computing *devices*:

- *hosts* = end systems
- running *network apps* at Internet's “edge”

*Packet switches*: forward packets (chunks of data)

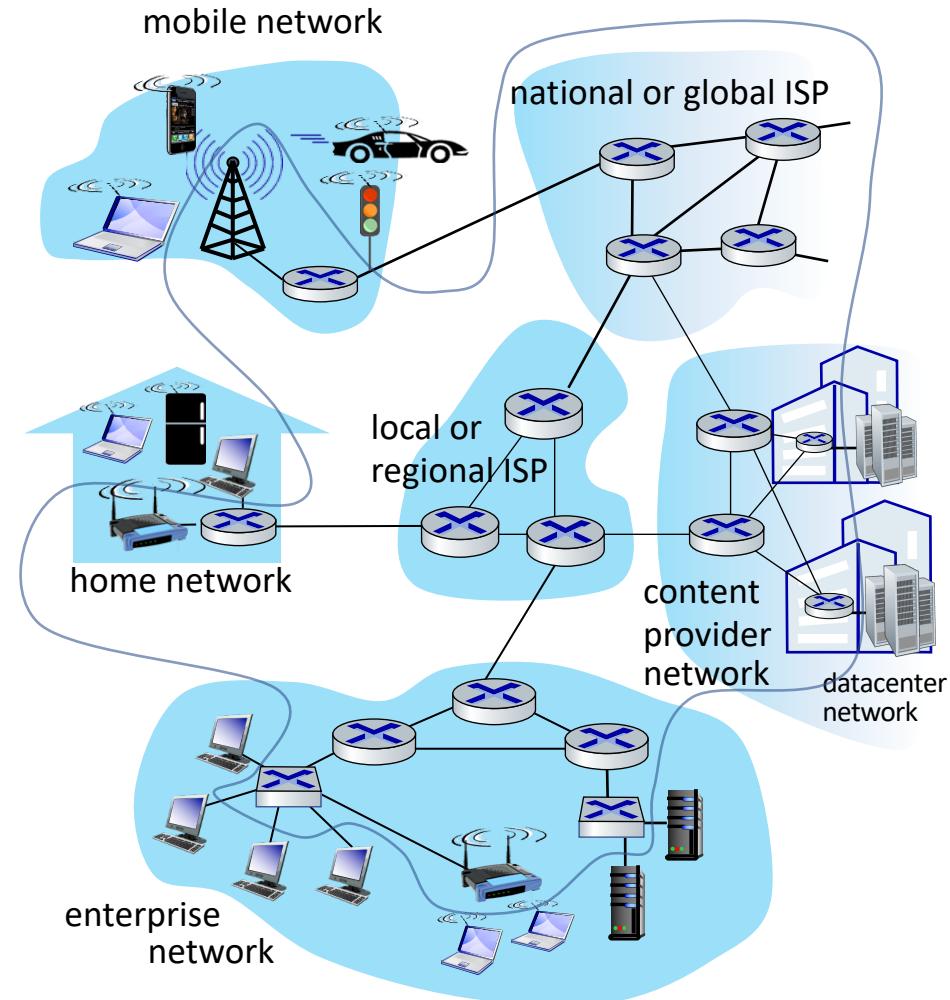
- routers, switches

*Communication links*

- fiber, copper, radio, satellite
- transmission rate: *bandwidth*

*Networks*

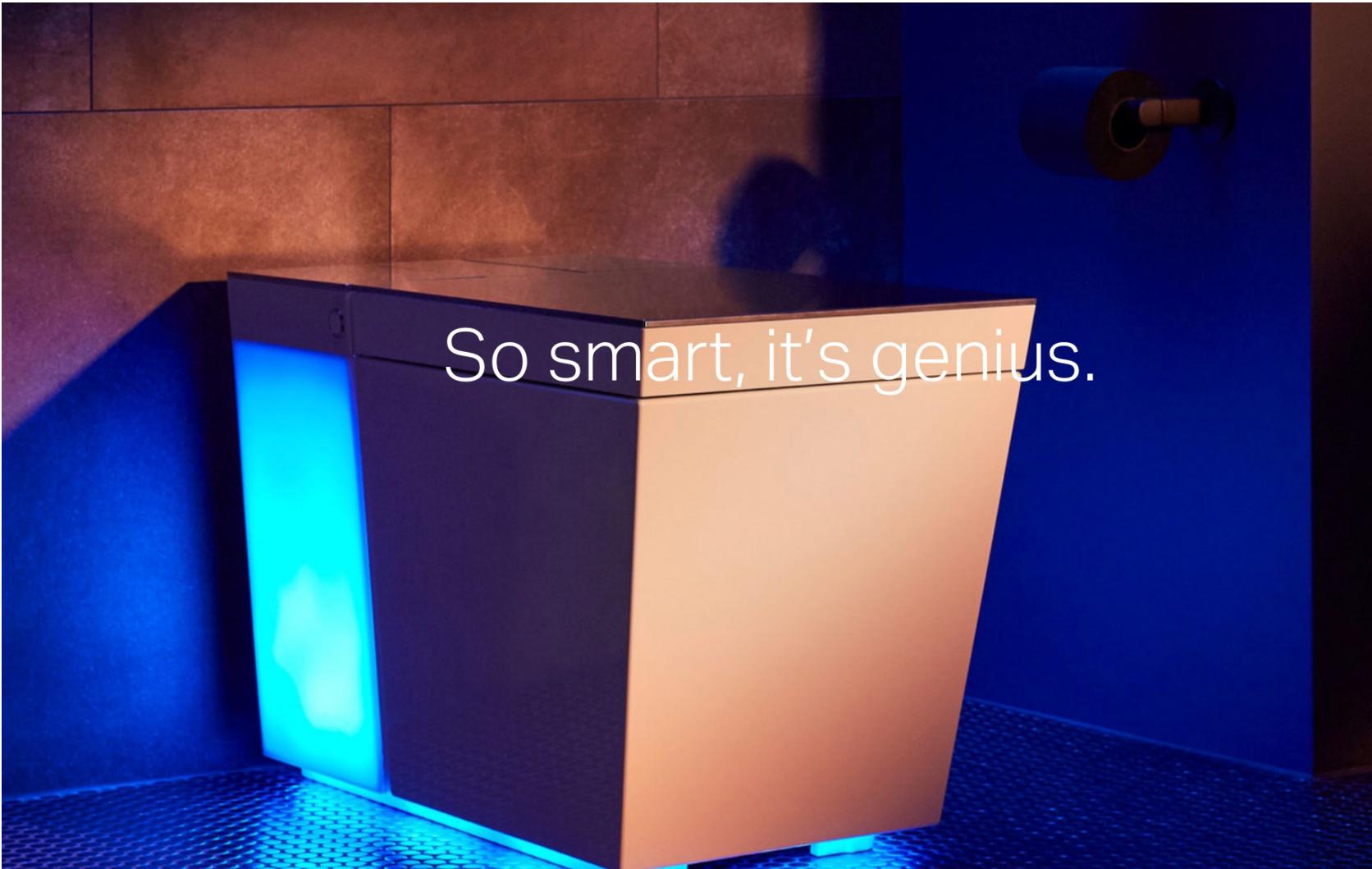
- collection of devices, routers, links: managed by an organization



# “Fun” Internet-connected devices

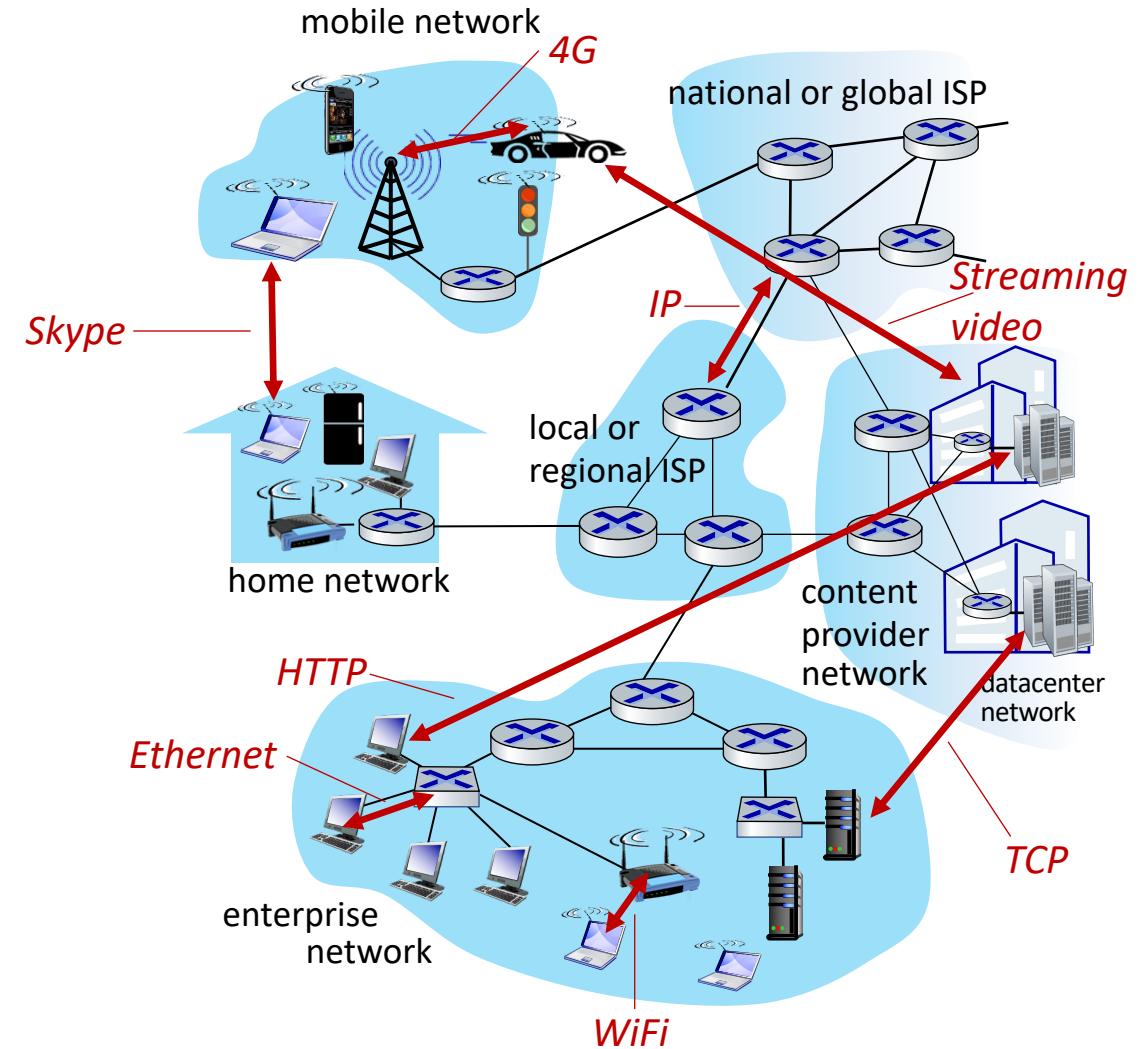


# “Fun” Internet-connected devices



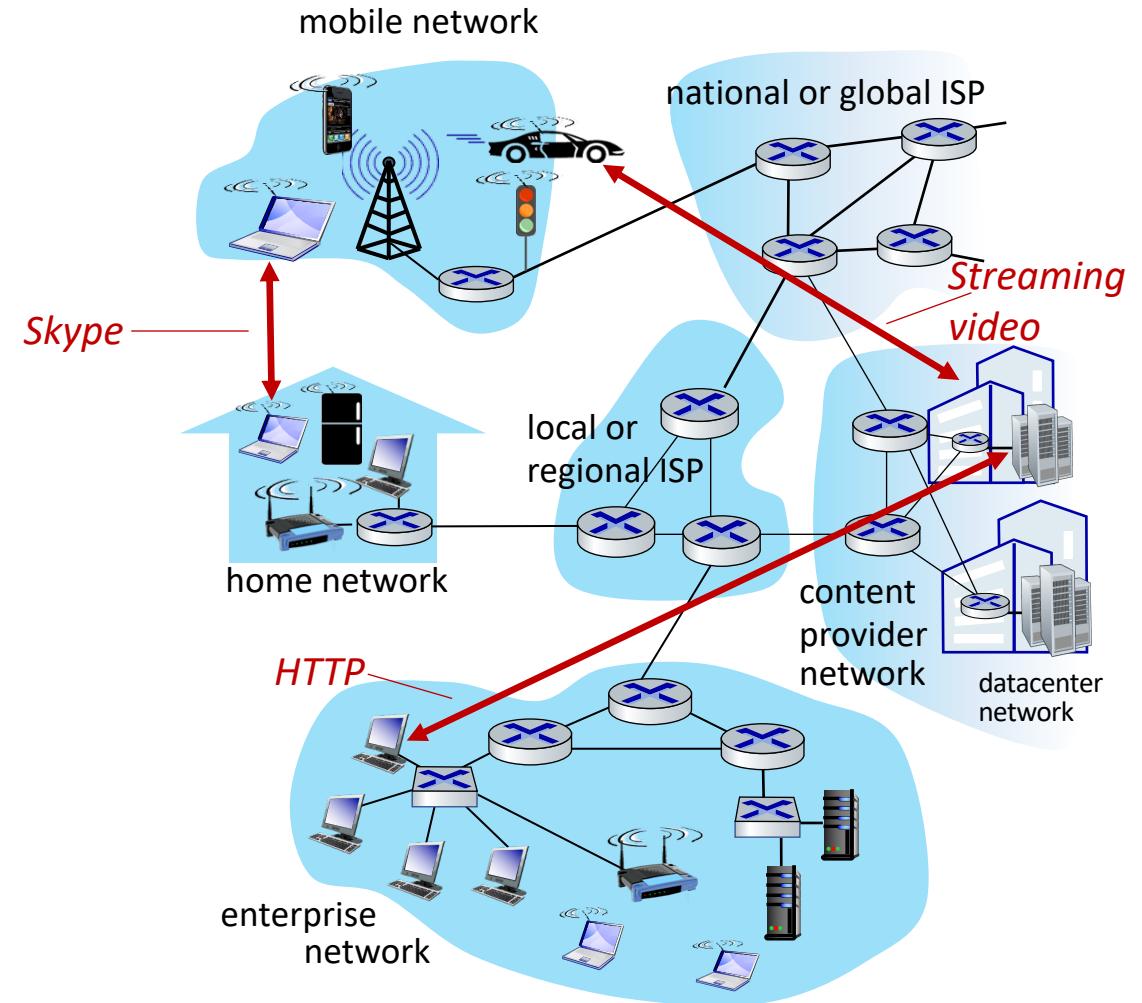
# The Internet: a “nuts and bolts” view

- *Internet: “network of networks”*
  - Interconnected ISPs
- *protocols are everywhere*
  - control sending, receiving of messages
  - e.g., HTTP (Web), streaming video, Skype, TCP, IP, WiFi, 4G, Ethernet
- *Internet standards*
  - RFC: Request for Comments
  - IETF: Internet Engineering Task Force



# The Internet: a “service” view

- *Infrastructure* that provides services to applications:
  - Web, streaming video, multimedia teleconferencing, email, games, e-commerce, social media, interconnected appliances, ...
- provides *programming interface* to distributed applications:
  - “hooks” allowing sending/receiving apps to “connect” to, use Internet transport service
  - provides service options, analogous to postal service



# What's a protocol?

## *Human protocols:*

- “what’s the time?”
- “I have a question”
- introductions

... specific messages sent  
... specific actions taken  
when message received,  
or other events

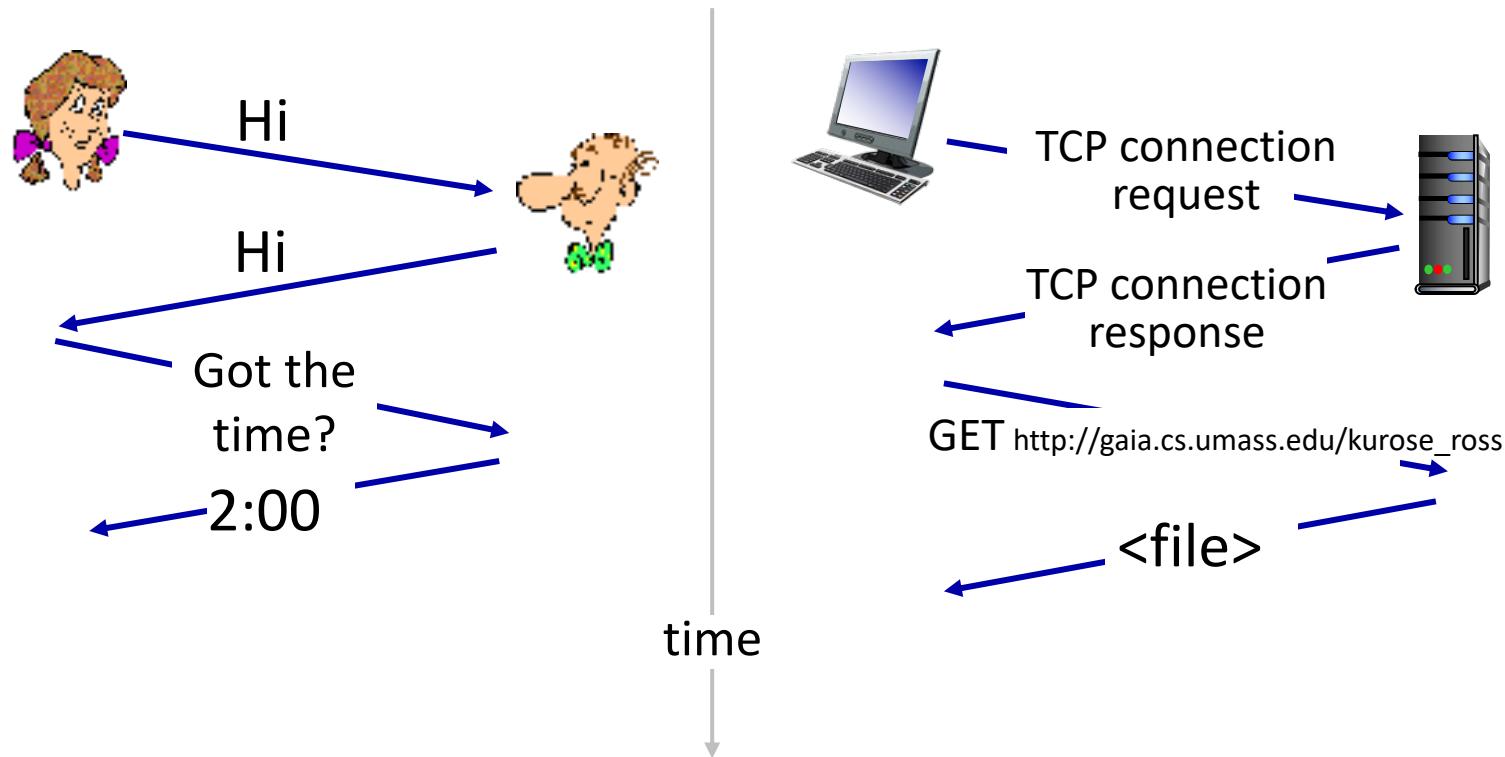
## *Network protocols:*

- computers (devices) rather than humans
- all communication activity in Internet governed by protocols

*Protocols define the **format, order** of messages sent and received among network entities, and **actions taken** on msg transmission, receipt*

# What's a protocol?

A human protocol and a computer network protocol:



*Q:* other human protocols?

# Chapter 1: roadmap

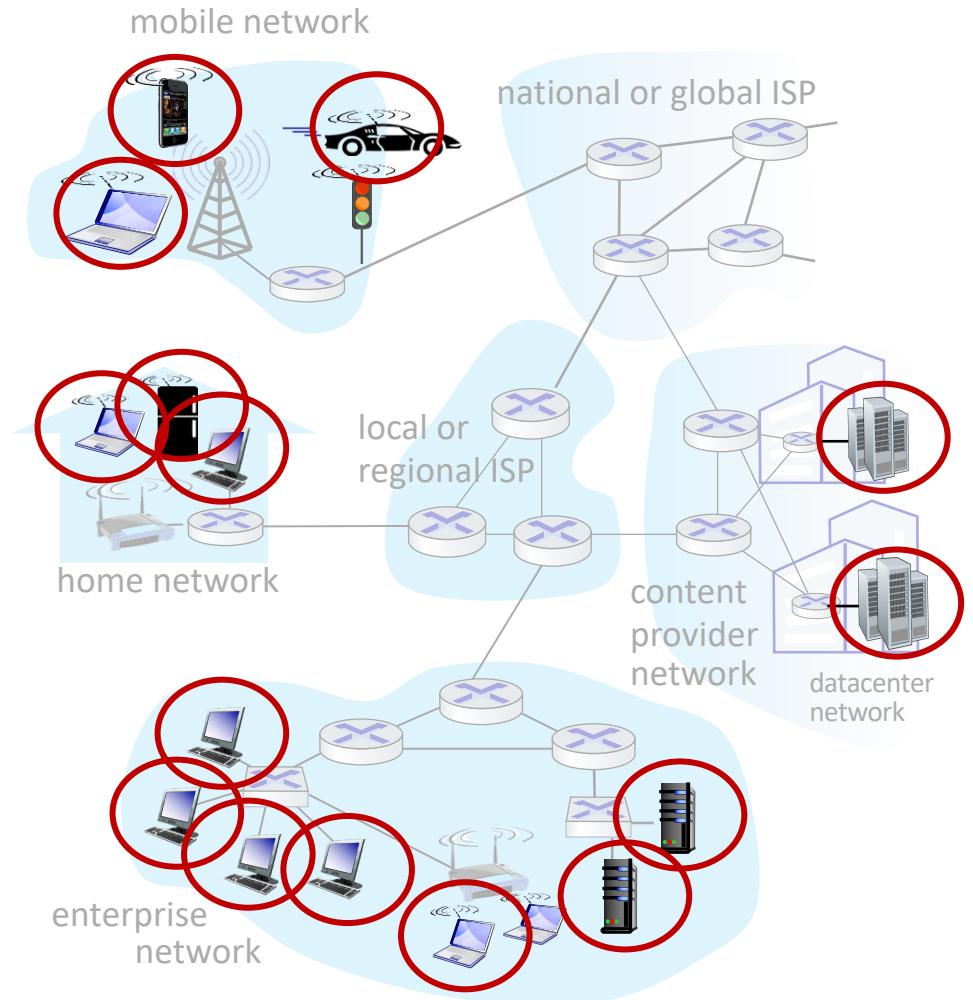
- What *is* the Internet?
- What *is* a protocol?
- **Network edge:** hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- Performance: loss, delay, throughput
- Security
- Protocol layers, service models
- History



# A closer look at Internet structure

## Network edge:

- hosts: clients and servers
- servers often in data centers



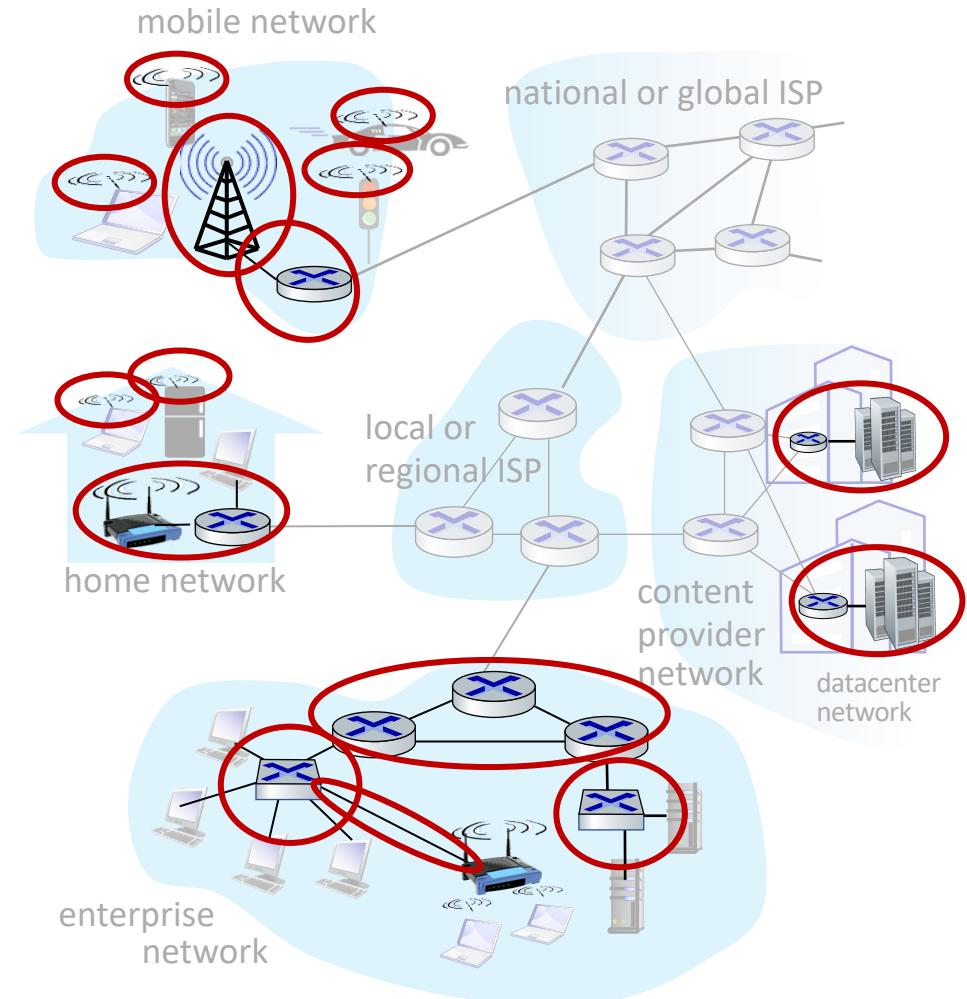
# A closer look at Internet structure

## Network edge:

- hosts: clients and servers
- servers often in data centers

## Access networks, physical media:

- wired, wireless communication links



# A closer look at Internet structure

## Network edge:

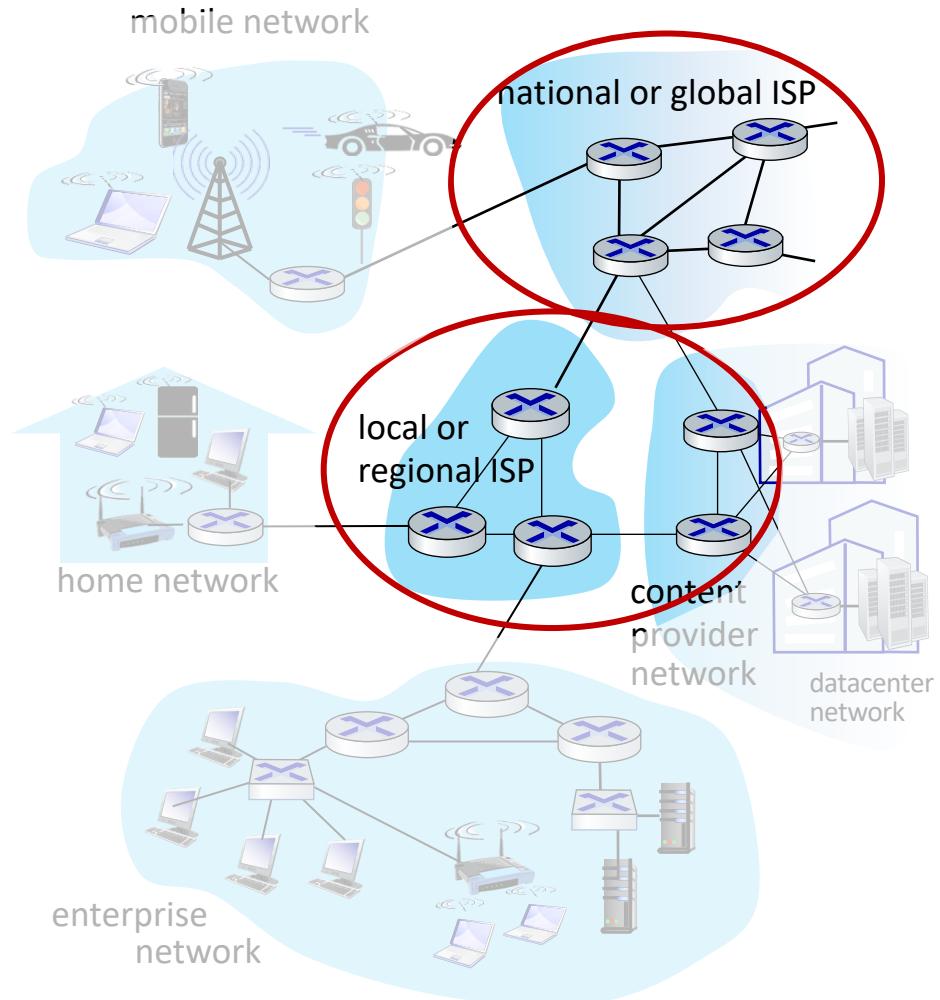
- hosts: clients and servers
- servers often in data centers

## Access networks, physical media:

- wired, wireless communication links

## Network core:

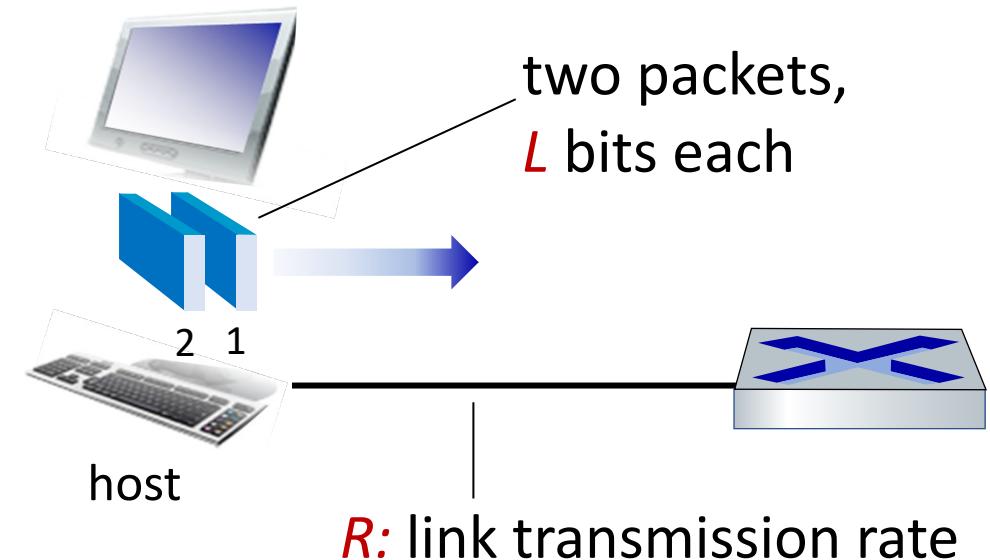
- interconnected routers
- network of networks



# Host: sends *packets* of data

host sending function:

- takes application message
- breaks into smaller chunks, known as *packets*, of length  $L$  bits
- transmits packet into access network at *transmission rate R*
  - link transmission rate, aka link *capacity, aka link bandwidth*



$$\text{packet transmission delay} = \frac{\text{time needed to transmit } L\text{-bit packet into link}}{R \text{ (bits/sec)}} = \frac{L \text{ (bits)}}{R \text{ (bits/sec)}}$$

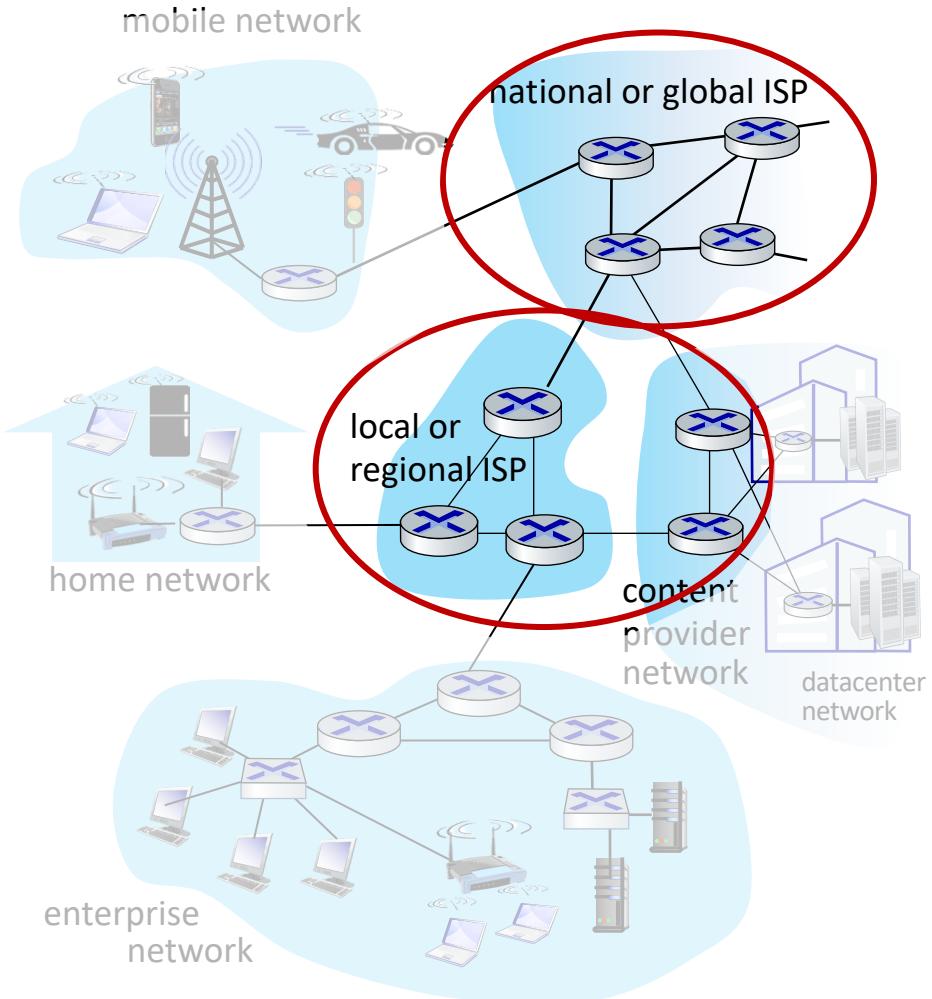
# Chapter 1: roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- **Network core:** packet switching, internet structure
- Performance: loss, delay, throughput
- Security
- Protocol layers, service models
- History

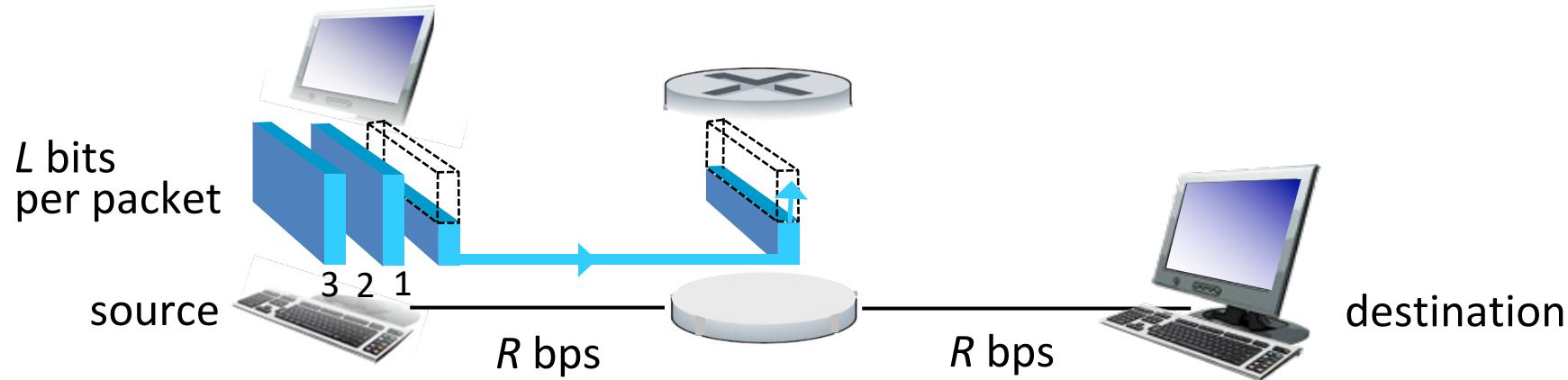


# The network core

- mesh of interconnected routers
- **packet-switching:** hosts break application-layer messages into *packets*
  - forward packets from one router to the next, across links on path from source to destination
  - each packet transmitted at full link capacity



# Packet-switching: store-and-forward



- **Transmission delay:** takes  $L/R$  seconds to transmit (push out)  $L$ -bit packet into link at  $R$  bps
- **Store and forward:** entire packet must arrive at router before it can be transmitted on next link
- **End-end delay:**  $2L/R$  (above), assuming zero propagation delay (more on delay shortly)

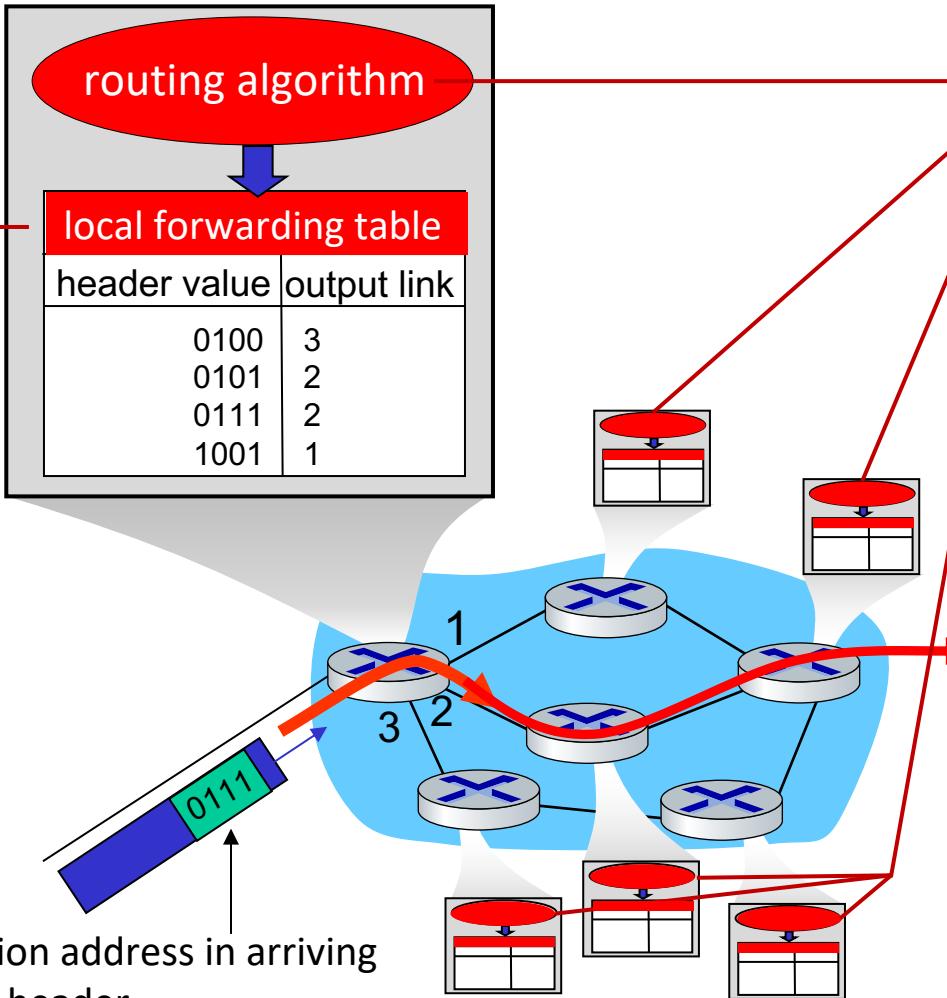
*One-hop numerical example:*

- $L = 10$  Kbits
- $R = 100$  Mbps
- one-hop transmission delay = 0.1 msec

# Two key network-core functions

*Forwarding:*

- *local* action:  
move arriving  
packets from  
router's input link  
to appropriate  
router output link



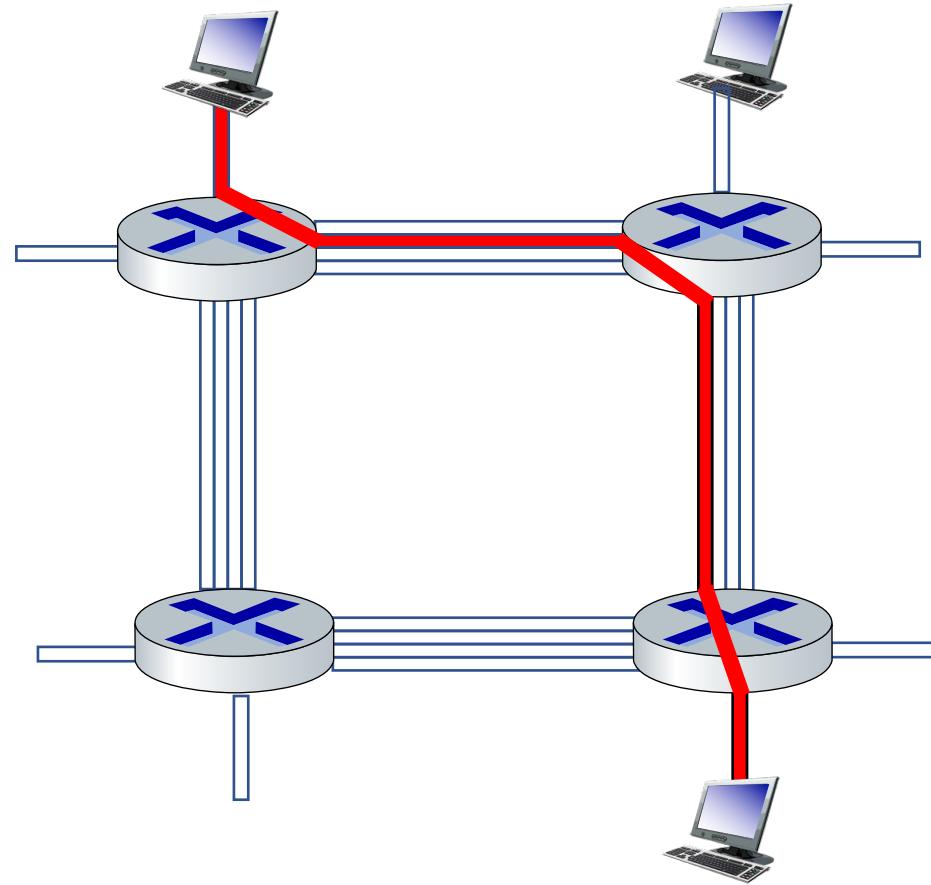
*Routing:*

- *global* action:  
determine source-  
destination paths  
taken by packets
- routing algorithms

# Alternative to packet switching: circuit switching

end-end resources allocated to,  
reserved for “call” between source  
and destination

- in diagram, each link has four circuits.
  - call gets 2<sup>nd</sup> circuit in top link and 1<sup>st</sup> circuit in right link.
- dedicated resources: no sharing
  - circuit-like (guaranteed) performance
- circuit segment idle if not used by call (no sharing)
- commonly used in traditional telephone networks

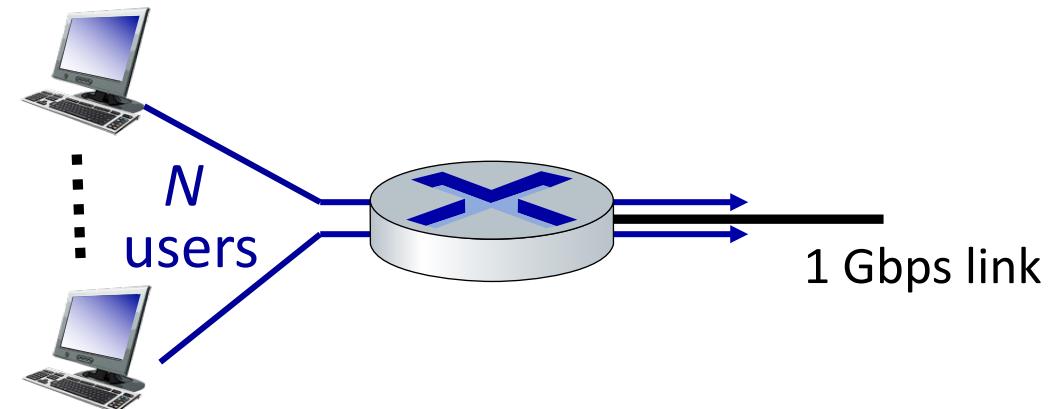


# Packet switching versus circuit switching

*packet switching allows more users to use network!*

Example:

- 1 Gb/s link
- each user:
  - 100 Mb/s when “active”
  - active 10% of time
- *circuit-switching*: 10 users
- *packet switching*: with 35 users,  
probability > 10 active at same time  
is less than .0004



# Packet switching versus circuit switching

Is packet switching a “slam dunk winner”?

- great for “bursty” data – sometimes has data to send, but at other times not
  - resource sharing
  - simpler, no call setup
- **excessive congestion possible:** packet delay and loss due to buffer overflow
  - protocols needed for reliable data transfer, congestion control
- **Q: How to provide circuit-like behavior?**
  - bandwidth guarantees traditionally used for audio/video applications

**Q:** human analogies of reserved resources (circuit switching) versus on-demand allocation (packet switching)?

# Chapter 1: roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- **Performance:** loss, delay, throughput
- Security
- Protocol layers, service models
- History

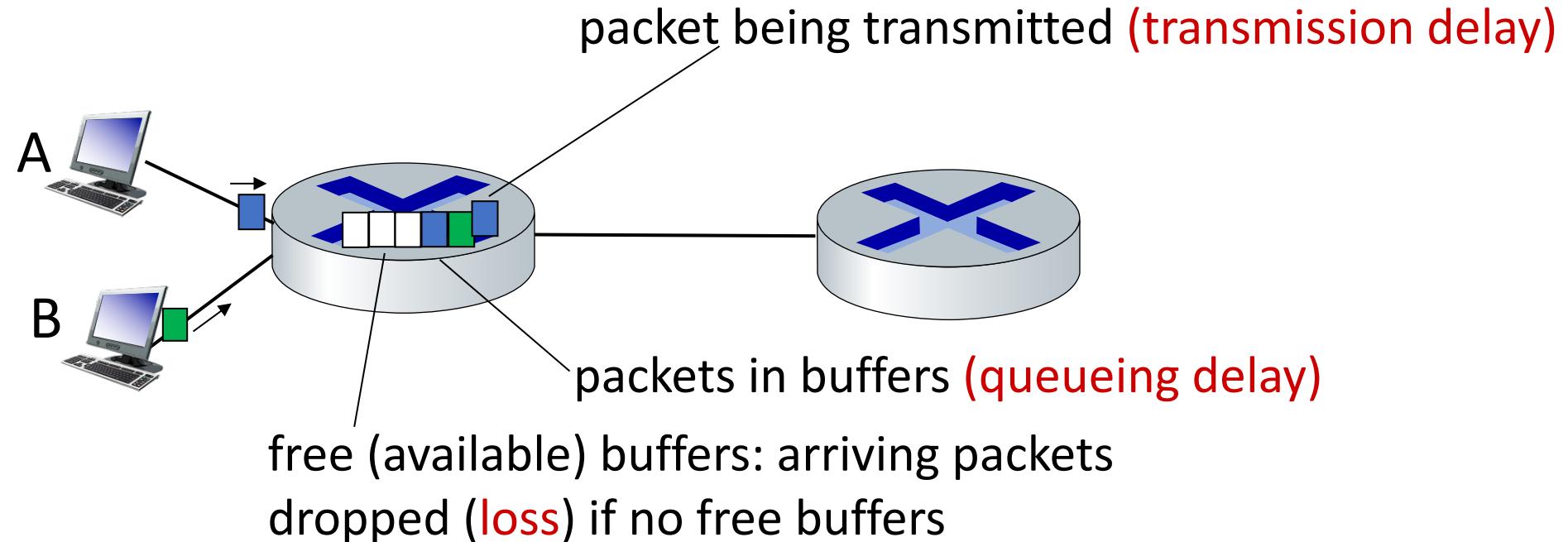


# How do packet loss and delay occur?

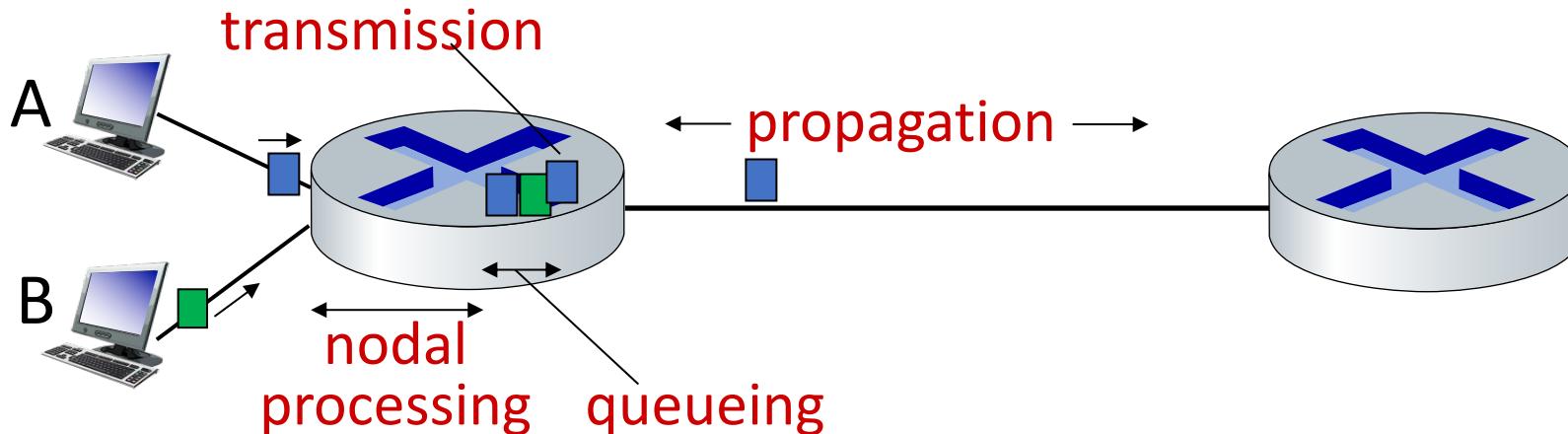
# How do packet loss and delay occur?

packets *queue* in router buffers

- packets queue, wait for turn
- arrival rate to link (temporarily) exceeds output link capacity: packet loss



# Packet delay: four sources



$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

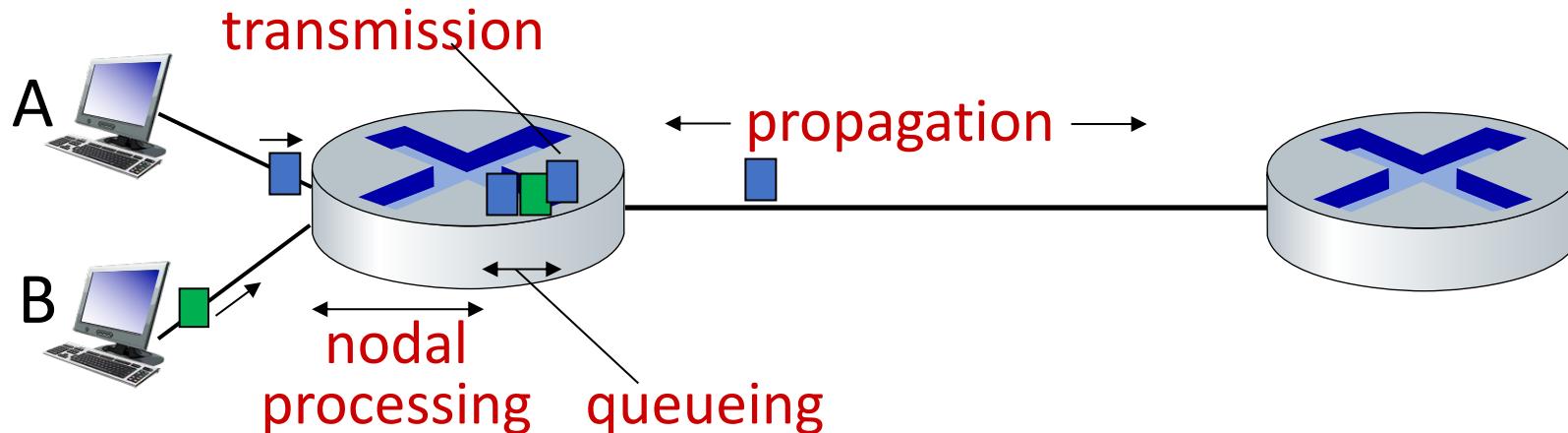
$d_{\text{proc}}$ : nodal processing

- check bit errors
- determine output link
- typically < msec

$d_{\text{queue}}$ : queueing delay

- time waiting at output link for transmission
- depends on congestion level of router

# Packet delay: four sources



$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

$d_{\text{trans}}$ : transmission delay:

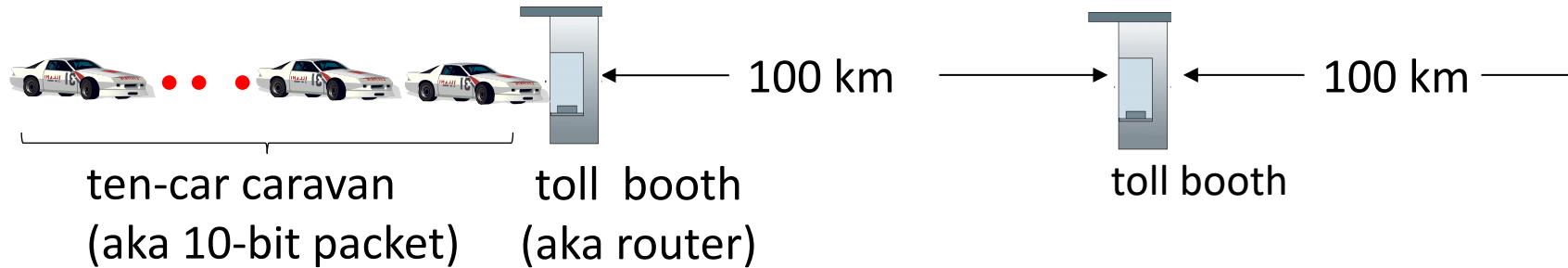
- $L$ : packet length (bits)
- $R$ : link *transmission rate (bps)*
- $d_{\text{trans}} = L/R$

$d_{\text{trans}}$  and  $d_{\text{prop}}$   
very different

$d_{\text{prop}}$ : propagation delay:

- $d$ : length of physical link
- $s$ : propagation speed ( $\sim 3 \times 10^8$  m/sec)
- $d_{\text{prop}} = d/s$

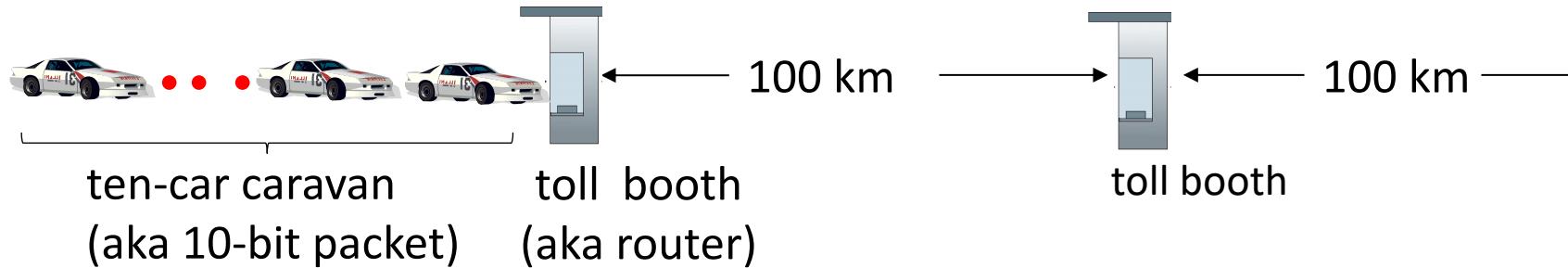
# Caravan analogy



- cars “propagate” at 100 km/hr
- toll booth takes 12 sec to service car (bit transmission time)
- car ~ bit; caravan ~ packet
- **Q: How long until caravan is lined up before 2nd toll booth?**

- time to “push” entire caravan through toll booth onto highway =  $12 * 10 = 120$  sec
- time for last car to propagate from 1st to 2nd toll both:  $100\text{km}/(100\text{km/hr}) = 1$  hr
- **A: 62 minutes**

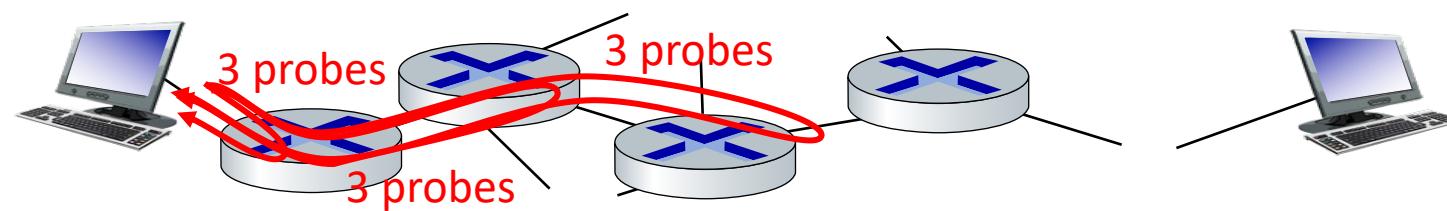
# Caravan analogy



- suppose cars now “propagate” at 1000 km/hr
  - and suppose toll booth now takes one min to service a car
  - ***Q: Will cars arrive to 2nd booth before all cars serviced at first booth?***
- A: Yes!** after 7 min, first car arrives at second booth; three cars still at first booth

# “Real” Internet delays and routes

- what do “real” Internet delay & loss look like?
- **traceroute** program: provides delay measurement from source to router along end-end Internet path towards destination. For all  $i$ :
  - sends three packets that will reach router  $i$  on path towards destination (with time-to-live field value of  $i$ )
  - router  $i$  will return packets to sender
  - sender measures time interval between transmission and reply



# Real Internet delays and routes

traceroute: gaia.cs.umass.edu to www.eurecom.fr

		3 delay measurements from gaia.cs.umass.edu to cs-gw.cs.umass.edu
1	cs-gw (128.119.240.254)	1 ms 1 ms 2 ms
2	border1-rt-fa5-1-0.gw.umass.edu (128.119.3.145)	1 ms 1 ms 2 ms
3	cht-vbns.gw.umass.edu (128.119.3.130)	6 ms 5 ms 5 ms
4	jn1-at1-0-0-19.wor.vbns.net (204.147.132.129)	16 ms 11 ms 13 ms
5	jn1-so7-0-0-0.wae.vbns.net (204.147.136.136)	21 ms 18 ms 18 ms
6	abilene-vbns.abilene.ucaid.edu (198.32.11.9)	22 ms 18 ms 22 ms
7	nycm-wash.abilene.ucaid.edu (198.32.8.46)	22 ms 22 ms 22 ms
8	62.40.103.253 (62.40.103.253)	104 ms 109 ms 106 ms
9	de2-1.de1.de.geant.net (62.40.96.129)	109 ms 102 ms 104 ms
10	de.fr1.fr.geant.net (62.40.96.50)	113 ms 121 ms 114 ms
11	renater-gw.fr1.fr.geant.net (62.40.103.54)	112 ms 114 ms 112 ms
12	nio-n2.cssi.renater.fr (193.51.206.13)	111 ms 114 ms 116 ms
13	nice.cssi.renater.fr (195.220.98.102)	123 ms 125 ms 124 ms
14	r3t2-nice.cssi.renater.fr (195.220.98.110)	126 ms 126 ms 124 ms
15	eurecom-valbonne.r3t2.ft.net (193.48.50.54)	135 ms 128 ms 133 ms
16	194.214.211.25 (194.214.211.25)	126 ms 128 ms 126 ms
17	***	
18	***	* means no response (probe lost, router not replying)
19	fantasia.eurecom.fr (193.55.113.142)	132 ms 128 ms 136 ms

3 delay measurements  
to border1-rt-fa5-1-0.gw.umass.edu

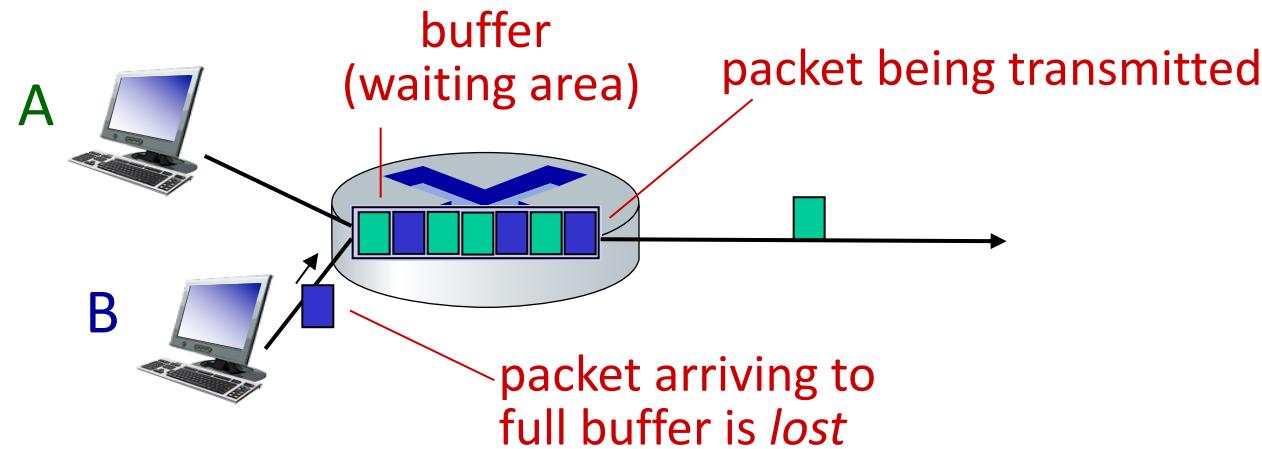
trans-oceanic link

looks like delays decrease! Why?

\* Do some traceroutes from exotic countries at [www.traceroute.org](http://www.traceroute.org)

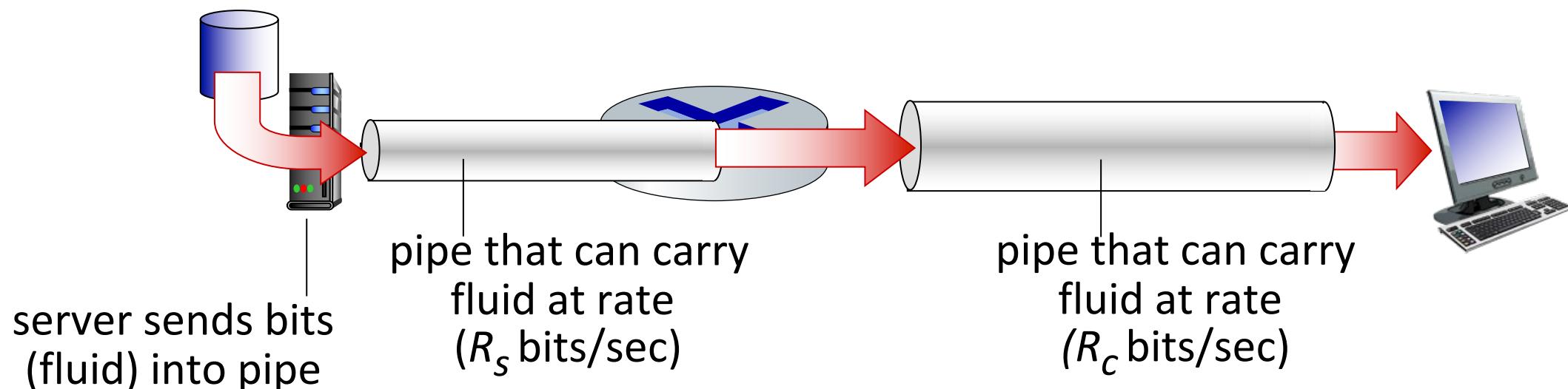
# Packet loss

- queue (aka buffer) preceding link in buffer has finite capacity
- packet arriving to full queue dropped (aka lost)
- lost packet may be retransmitted by previous node, by source end system, or not at all



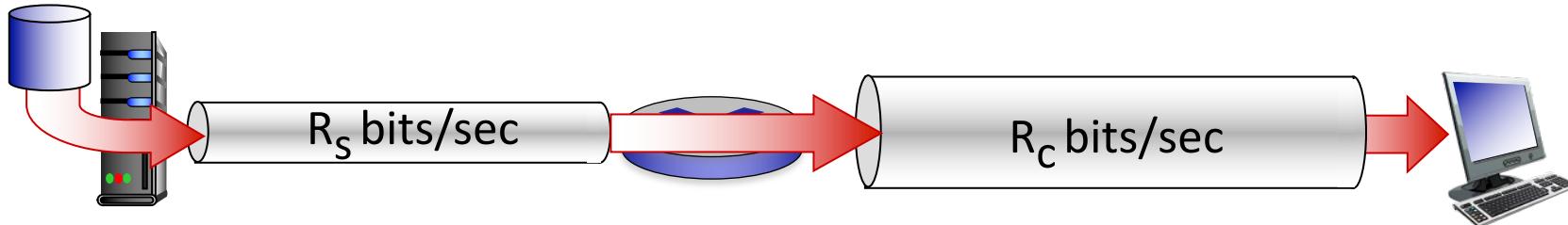
# Throughput

- *throughput*: rate (bits/time unit) at which bits are being sent from sender to receiver
  - *instantaneous*: rate at given point in time
  - *average*: rate over longer period of time

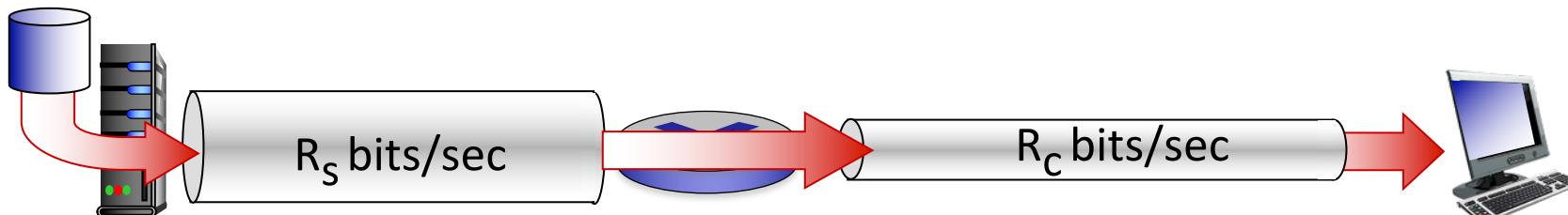


# Throughput

$R_s < R_c$  What is average end-end throughput?



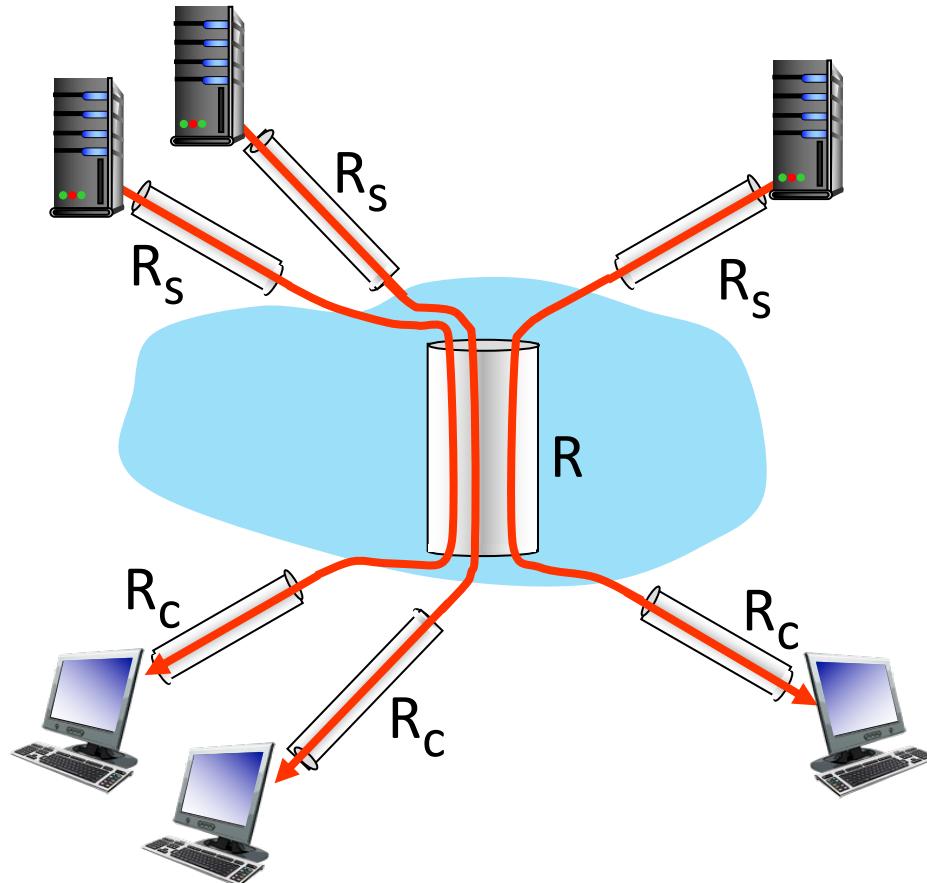
$R_s > R_c$  What is average end-end throughput?



*bottleneck link*

link on end-end path that constrains end-end throughput

# Throughput: network scenario



3 connections (fairly) share  
backbone bottleneck link  $R$  bits/sec

- per-connection end-end throughput:  $\min(R_c, R_s, R/3)$
- in practice:  $R_c$  or  $R_s$  is often bottleneck

# Chapter 1: roadmap

- What *is* the Internet?
- What *is* a protocol?
- Network edge: hosts, access network, physical media
- Network core: packet/circuit switching, internet structure
- Performance: loss, delay, throughput
- Security
- **Protocol layers, service models**
- History

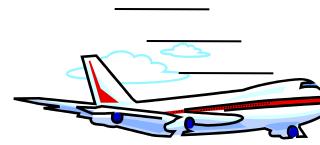


# Protocol “layers” and reference models

*Networks are complex,  
with many “pieces”:*

- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

# Example: organization of air travel



ticket (purchase)  
baggage (check)  
gates (load)  
runway takeoff  
airplane routing

ticket (complain)  
baggage (claim)  
gates (unload)  
runway landing  
airplane routing

airplane routing

airline travel: a series of steps, involving many services

# Example: organization of air travel



*layers:* each layer implements a service

- via its own internal-layer actions
- relying on services provided by layer below

*Q:* describe in words  
the service provided  
in each layer above

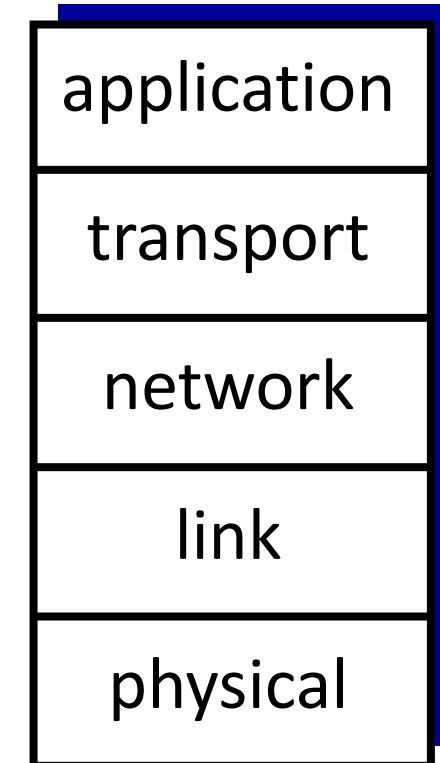
# Why layering?

dealing with complex systems:

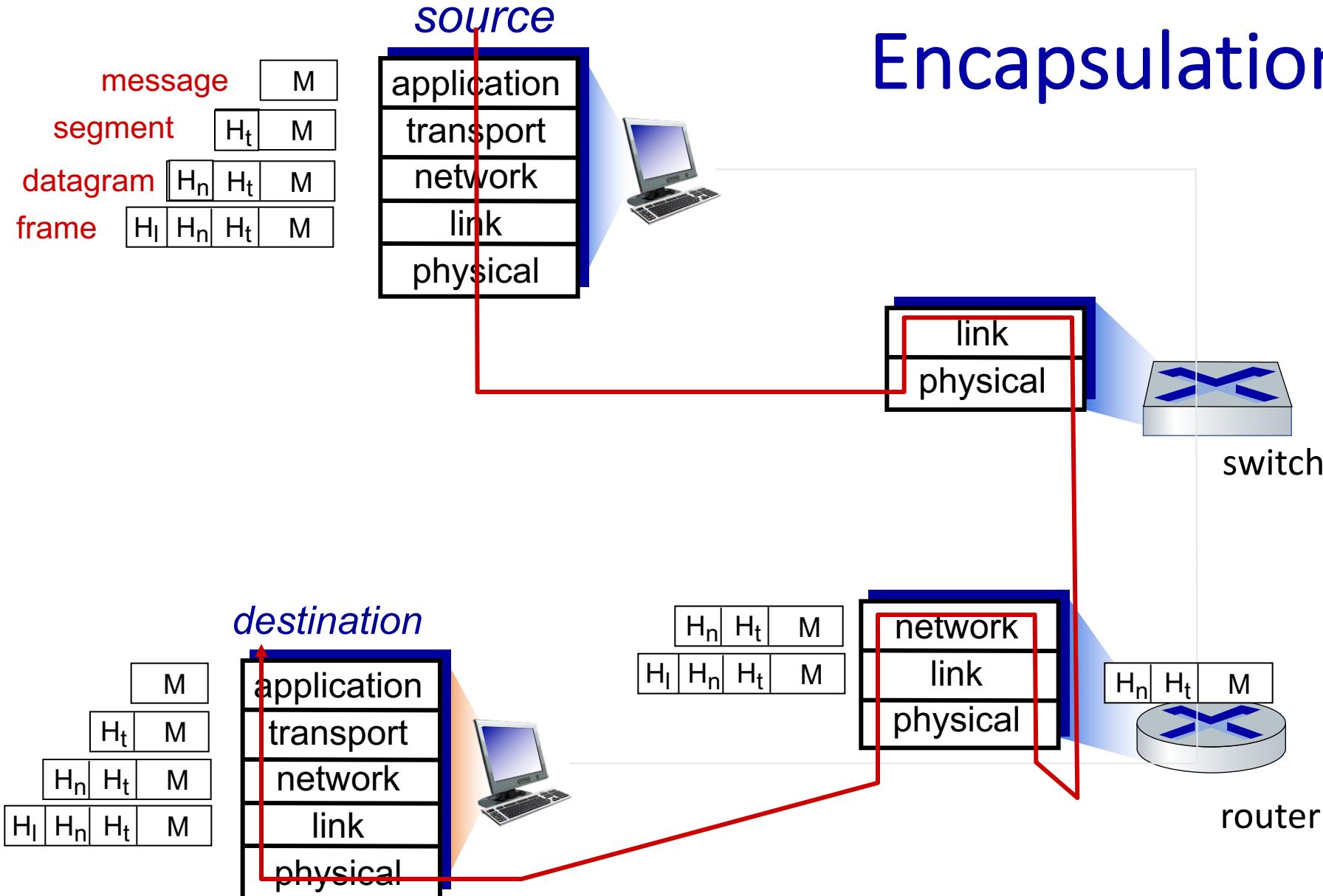
- explicit structure allows identification, relationship of complex system's pieces
  - layered *reference model* for discussion
- modularization eases maintenance, updating of system
  - change in layer's service *implementation*: transparent to rest of system
  - e.g., change in gate procedure doesn't affect rest of system
- layering considered harmful?
- layering in other complex systems?

# Internet protocol stack

- *application*: supporting network applications
  - IMAP, SMTP, HTTP
- *transport*: process-process data transfer
  - TCP, UDP
- *network*: routing of datagrams from source to destination
  - IP, routing protocols
- *link*: data transfer between neighboring network elements
  - Ethernet, 802.11 (WiFi), PPP
- *physical*: bits “on the wire”



# Encapsulation



# Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# Application layer: overview

## Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS
- programming network applications
  - socket API

# Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video  
(YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP (e.g., Zoom)
- real-time video conferencing
- Internet search
- remote login
- ...

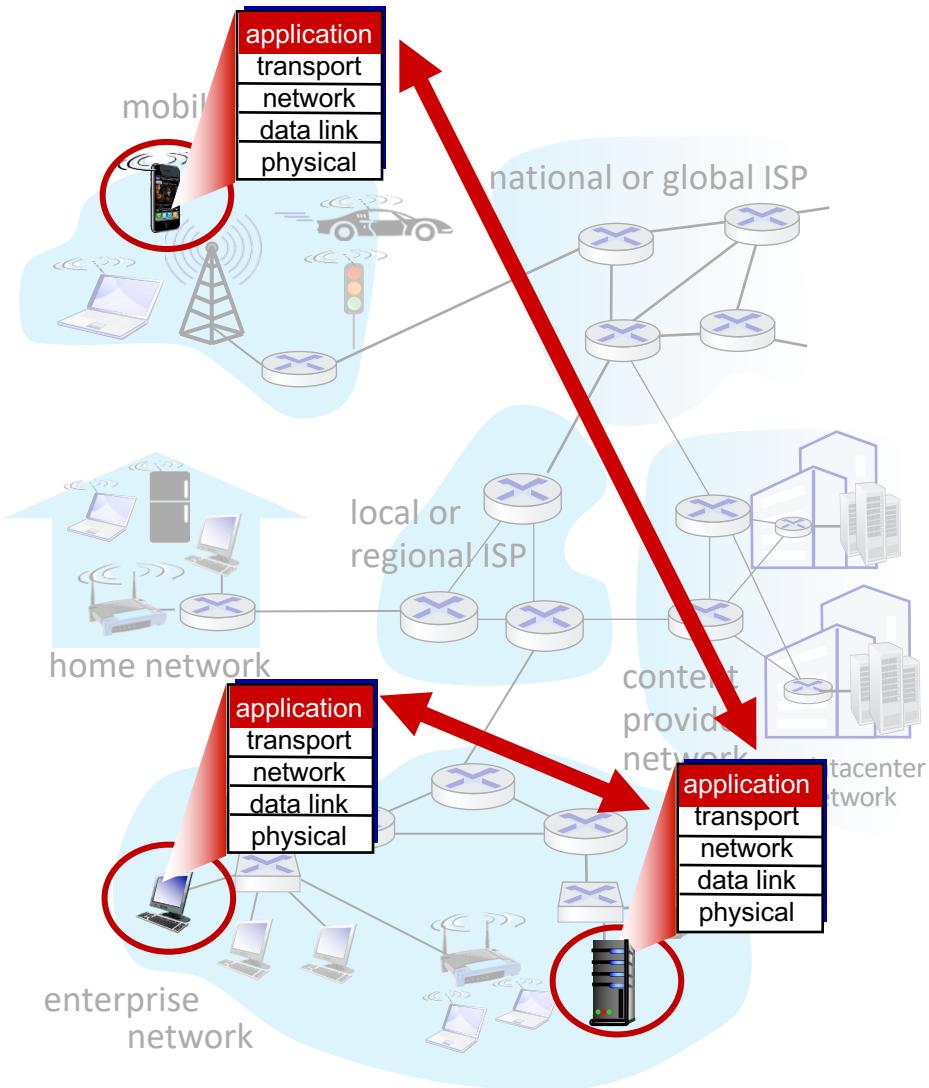
# Creating a network app

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software  
communicates with browser software

no need to write software for  
network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



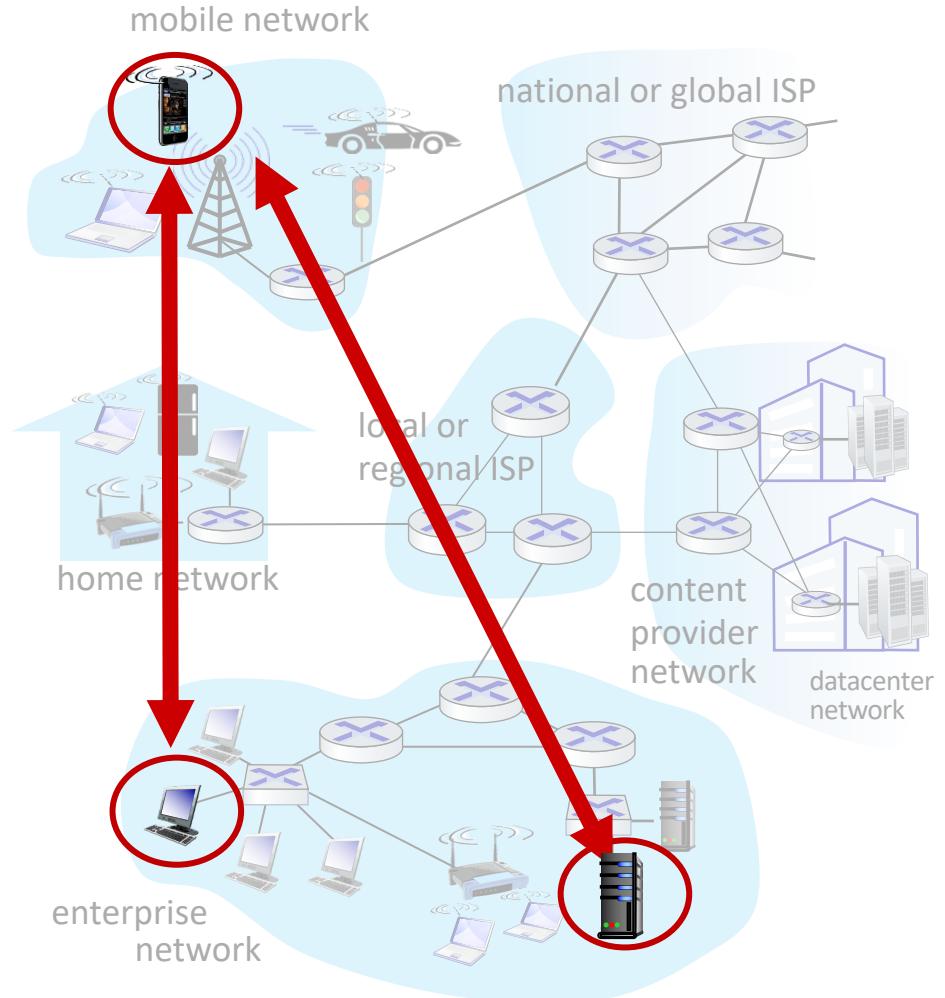
# Client-server paradigm

## server:

- always-on host
- permanent IP address
- often in data centers, for scaling

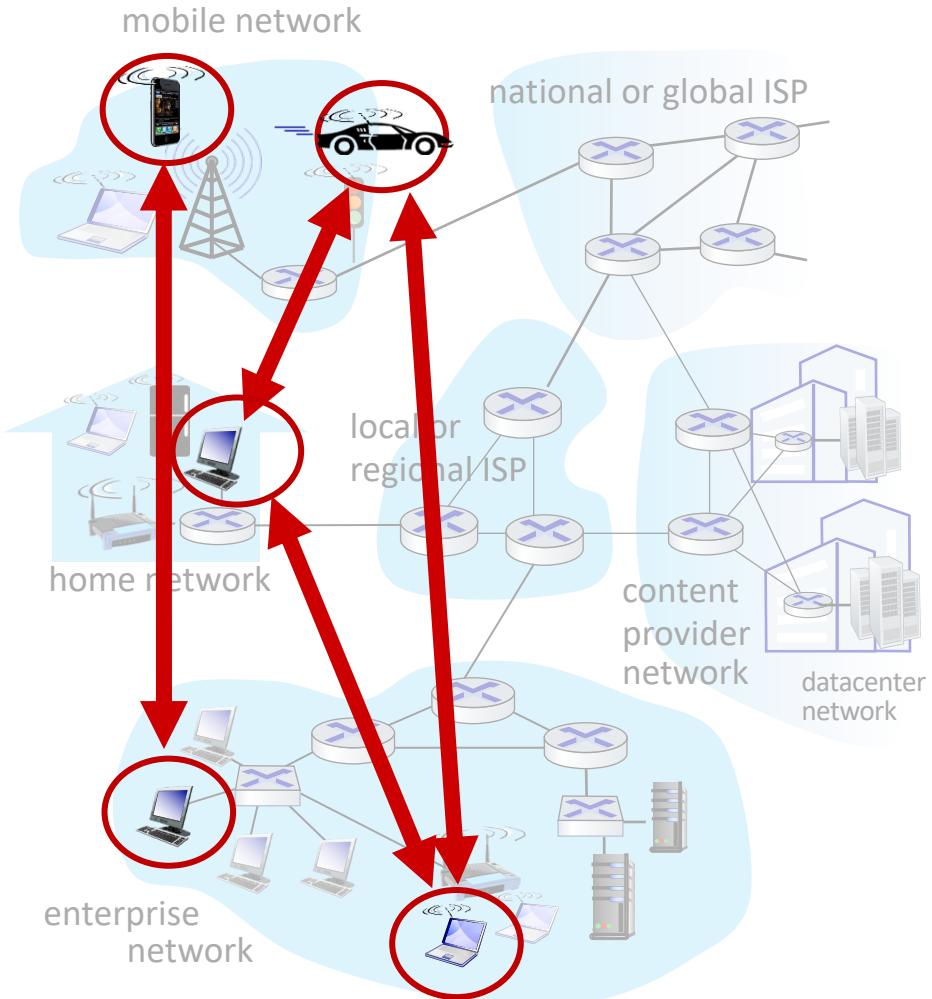
## clients:

- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other
- examples: HTTP, IMAP, FTP



# Peer-peer architecture

- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing



# Processes communicating

- process*: program running within a host
- within same host, two processes communicate using **inter-process communication** (defined by OS)
  - processes in different hosts communicate by exchanging **messages**

clients, servers

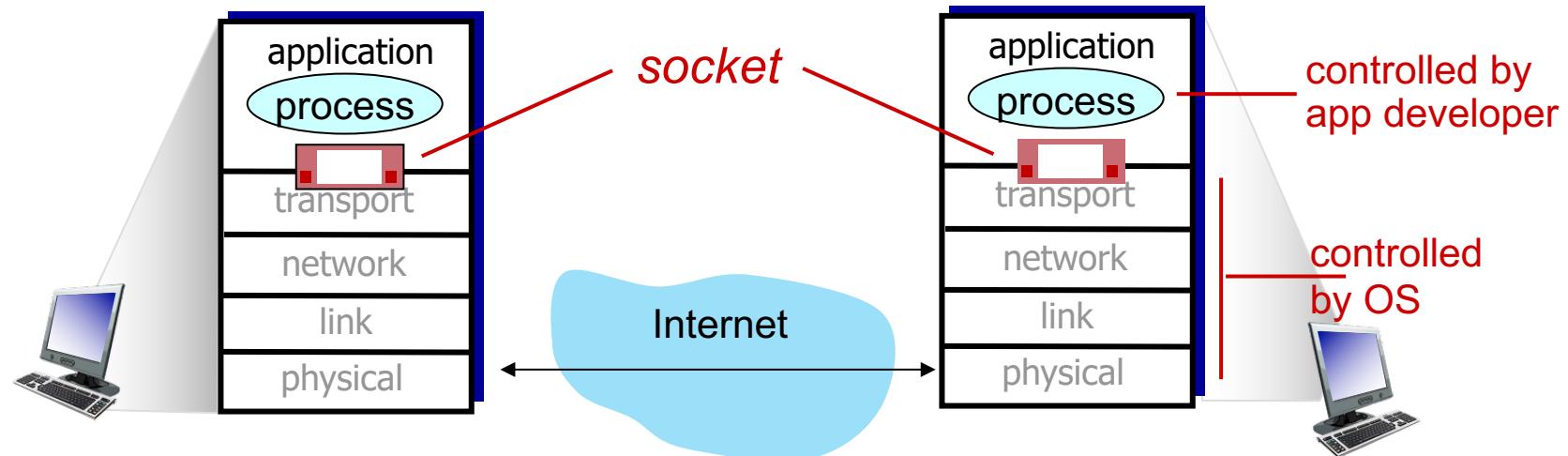
*client process*: process that initiates communication

*server process*: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

# Sockets

- process sends/receives messages to/from its **socket**
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



# Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host
- *identifier* includes both **IP address** and **port numbers** associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - **IP address:** 128.119.245.12
  - **port number:** 80

# An application-layer protocol defines:

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

## open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

## proprietary protocols:

- e.g., Skype

# What transport service does an app need?

## data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

## throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

## timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

## security

- encryption, data integrity, ...

# Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

# Internet transport protocols services

## TCP service:

- ***reliable transport*** between sending and receiving process
- ***flow control***: sender won't overwhelm receiver
- ***congestion control***: throttle sender when network overloaded
- ***does not provide***: timing, minimum throughput guarantee, security
- ***connection-oriented***: setup required between client and server processes

## UDP service:

- ***unreliable data transfer*** between sending and receiving process
- ***does not provide***: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP?

# Internet transport protocols services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

# Securing TCP

## Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

## Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

## TSL implemented in application layer

- apps use TSL libraries, that use TCP in turn

## TLS socket API

- cleartext sent into socket traverse Internet *encrypted*
- see Chapter 8

# Application layer: overview

- Principles of network applications
- **Web and HTTP**
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# Web and HTTP

- web page consists of *objects*, each of which can be stored on different Web servers
- object can be HTML file, JPEG image, audio file,...
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL*, e.g.,

www.someschool.edu/someDept/pic.gif

The URL "www.someschool.edu/someDept/pic.gif" is shown above two curly braces. The first brace, under "www.someschool.edu", is labeled "host name". The second brace, under "someDept/pic.gif", is labeled "path name".

# HTTP overview

## HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model:
  - *client*: browser that requests, receives, (using HTTP protocol) and “displays” Web objects
  - *server*: Web server sends (using HTTP protocol) objects in response to requests



# HTTP overview (continued)

## *HTTP uses TCP:*

- client initiates TCP connection (creates socket) to server, port 80 (or port 443)
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

## *HTTP is “stateless”*

- server maintains *no* information about past client requests

*aside*  
protocols that maintain “state” are complex!

- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

# HTTP connections: two types

## *Non-persistent HTTP*

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

## *Persistent HTTP*

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

# Non-persistent HTTP: example

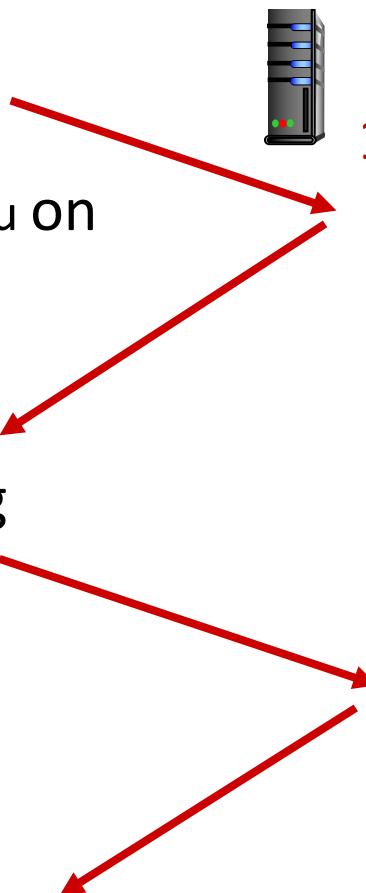
User enters URL: `www.someSchool.edu/someDepartment/home.index`  
(containing text, references to 10 jpeg images)



1a. HTTP client initiates TCP connection to HTTP server (process) at `www.someSchool.edu` on port 80

time  
↓

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `someDepartment/home.index`



1b. HTTP server at host `www.someSchool.edu` waiting for TCP connection at port 80 “accepts” connection, notifying client

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

# Non-persistent HTTP: example (cont.)

User enters URL: `www.someSchool.edu/someDepartment/home.index`  
(containing text, references to 10 jpeg images)



5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects



4. HTTP server closes TCP connection.

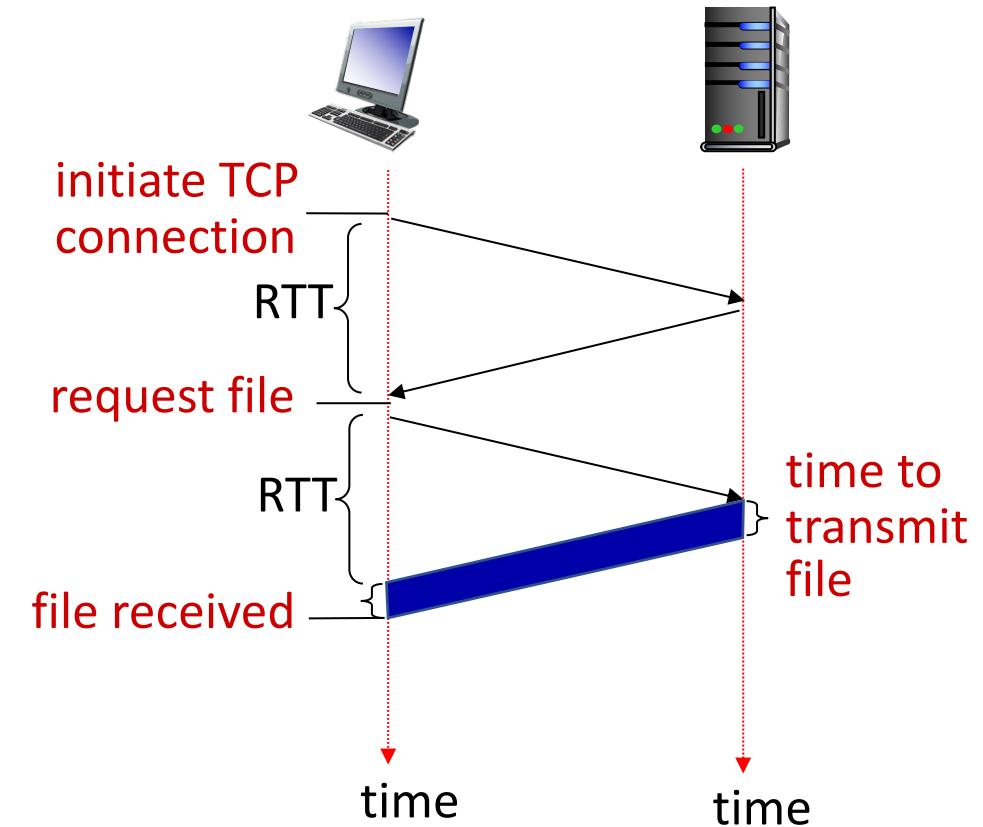
time

# Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back

HTTP response time (per object):

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



$$\text{Non-persistent HTTP response time} = 2\text{RTT} + \text{file transmission time}$$

# Persistent HTTP (HTTP 1.1)

## *Non-persistent HTTP issues:*

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

## *Persistent HTTP (HTTP1.1):*

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)

# HTTP request message

- two types of HTTP messages: *request, response*
- **HTTP request message:**

- ASCII (human-readable format)

request line (GET, POST,  
HEAD commands)

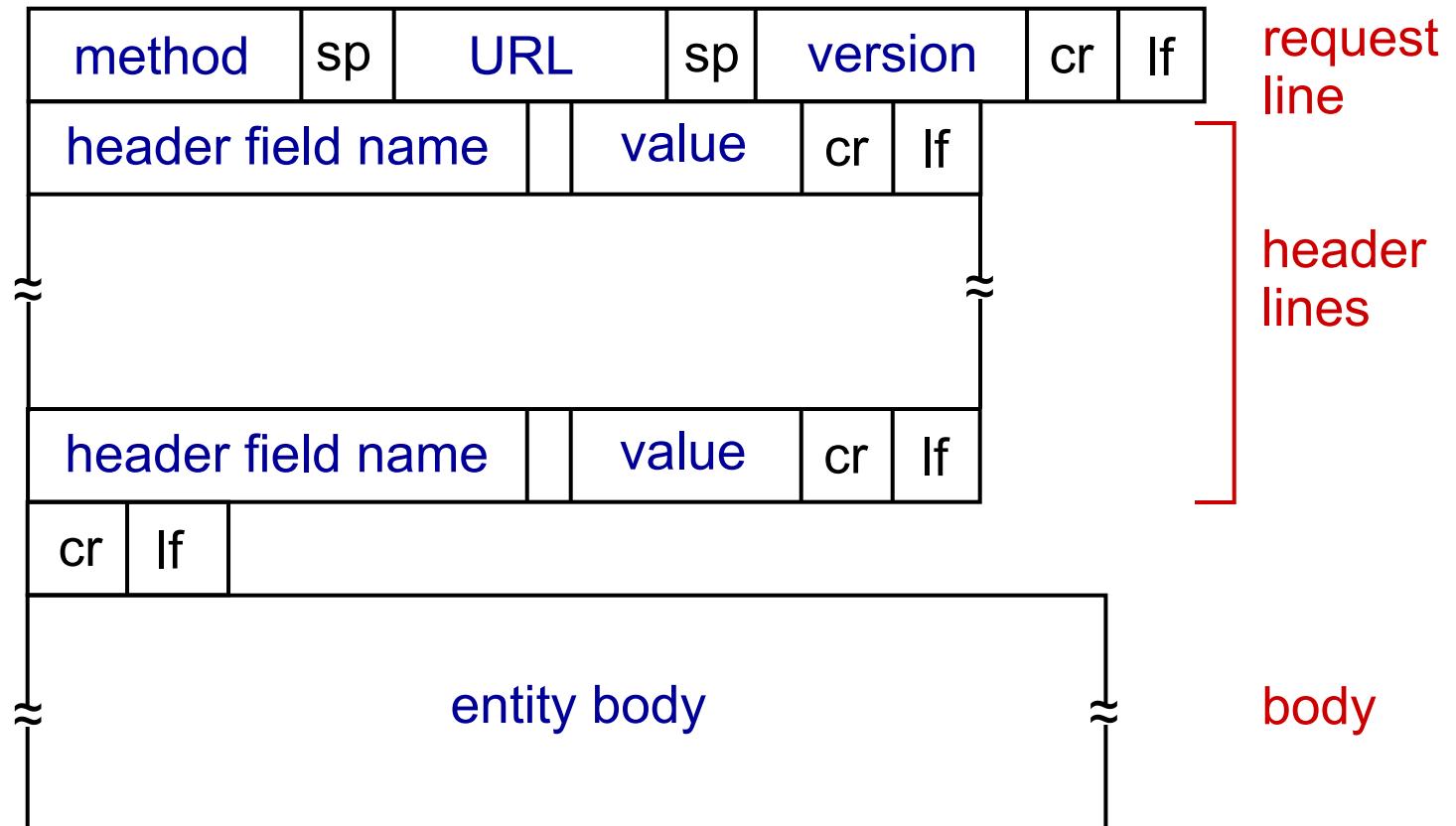
```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

carriage return character  
line-feed character

header  
lines

carriage return, line feed  
at start of line indicates  
end of header lines

# HTTP request message: general format



# Other HTTP request messages

## POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

## GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

`www.somesite.com/animalsearch?monkeys&banana`

## HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

## PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

# HTTP response message

status line (protocol  
status code status phrase)

HTTP/1.1 200 OK\r\nDate: Sun, 26 Sep 2010 20:09:20 GMT\r\nServer: Apache/2.0.52 (CentOS) \r\nLast-Modified: Tue, 30 Oct 2007 17:00:02  
GMT\r\n

header  
lines

data, e.g., requested  
HTML file

ETag: "17dc6-a5c-bf716880"\r\nAccept-Ranges: bytes\r\nContent-Length: 2652\r\nKeep-Alive: timeout=10, max=100\r\nConnection: Keep-Alive\r\nContent-Type: text/html; charset=ISO-8859-  
1\r\n\r\ndata data data data data ...

# HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

## 200 OK

- request succeeded, requested object later in this message

## 301 Moved Permanently

- requested object moved, new location specified later in this message (in Location: field)

## 400 Bad Request

- request msg not understood by server

## 404 Not Found

- requested document not found on this server

## 505 HTTP Version Not Supported

# Trying out HTTP (client side) for yourself

## 1. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

- opens TCP connection to port 80 (default HTTP server port) at gaia.cs.umass.edu.
- anything typed in will be sent to port 80 at gaia.cs.umass.edu

## 2. type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1  
Host: gaia.cs.umass.edu
```

- by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

## 3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

# Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



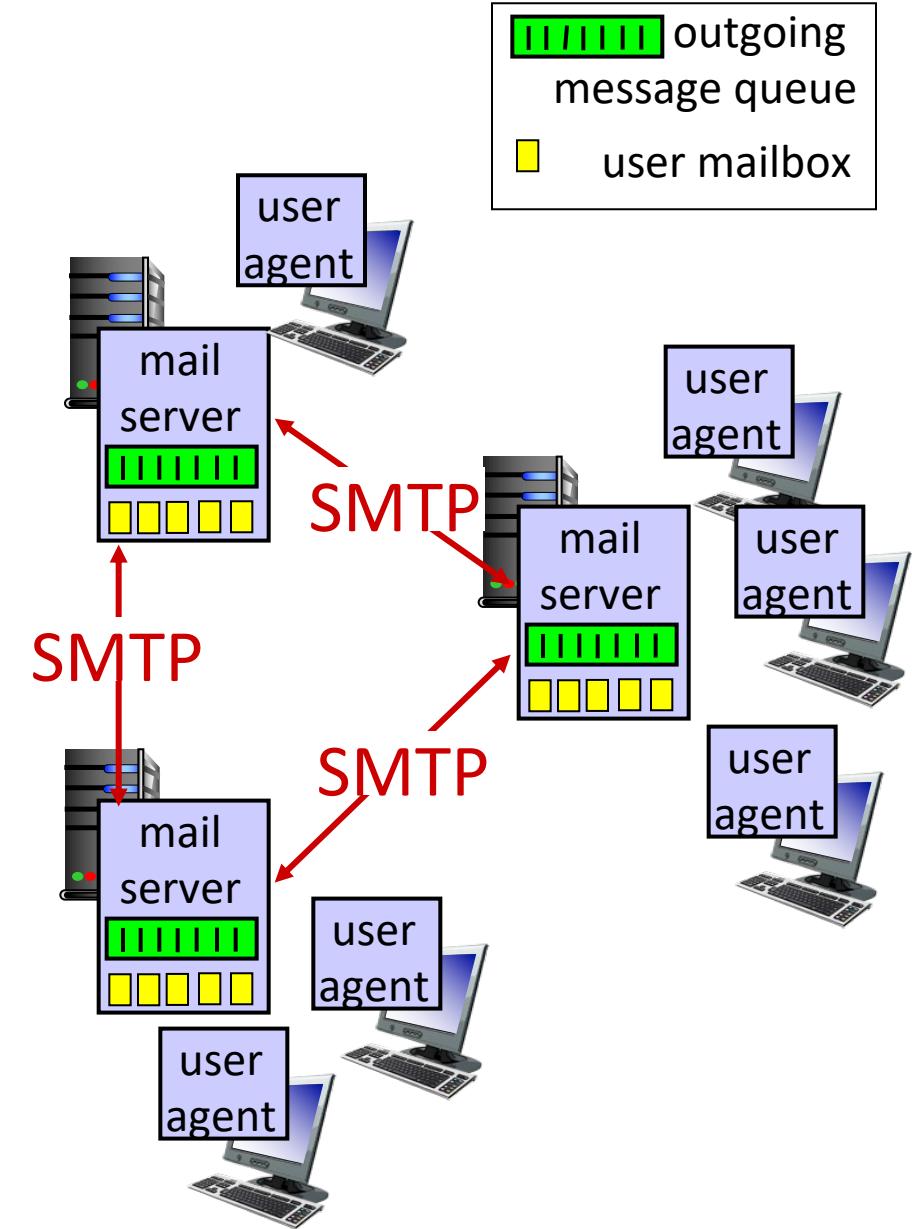
# E-mail

Three major components:

- user agents
- mail servers
- simple mail transfer protocol: SMTP

## User Agent

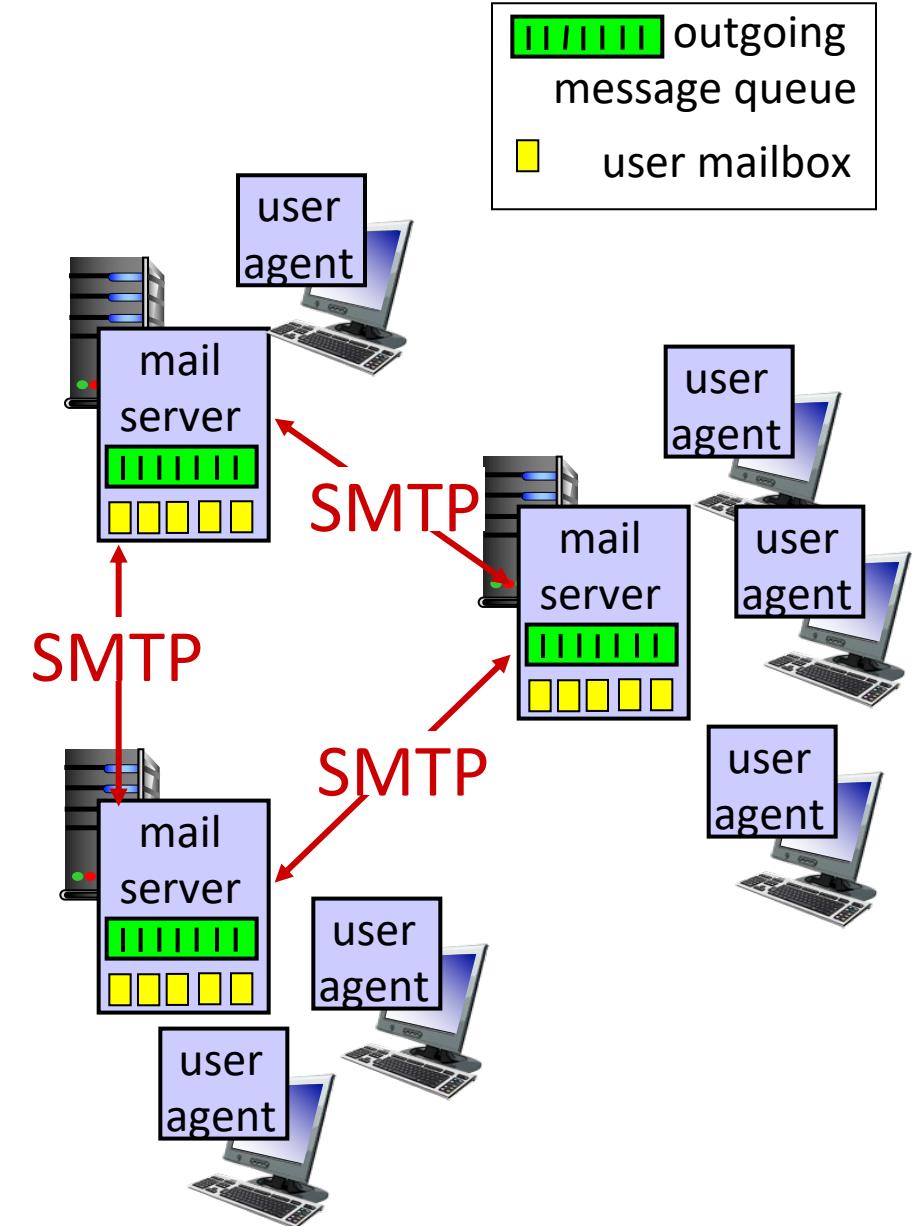
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server



# E-mail: mail servers

## mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages
- *SMTP protocol* between mail servers to send email messages
  - client: sending mail server
  - “server”: receiving mail server



# E-mail: the RFC (5321)

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
- direct transfer: sending server (acting like client) to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP)
  - **commands:** ASCII text
  - **response:** status code and phrase
- messages must be in 7-bit ASCII

# Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message “to” bob@someschool.edu

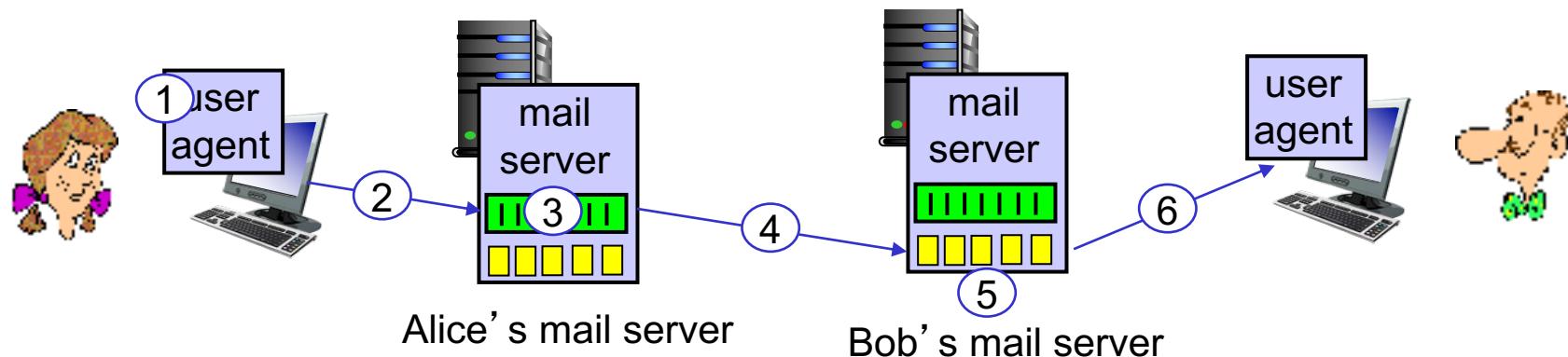
2) Alice’s UA sends message to her mail server; message placed in message queue

3) client side of SMTP opens TCP connection with Bob’s mail server

4) SMTP client sends Alice’s message over the TCP connection

5) Bob’s mail server places the message in Bob’s mailbox

6) Bob invokes his user agent to read message



# Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

# SMTP: closing observations

*comparison with HTTP:*

# SMTP: closing observations

## *comparison with HTTP:*

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message
- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

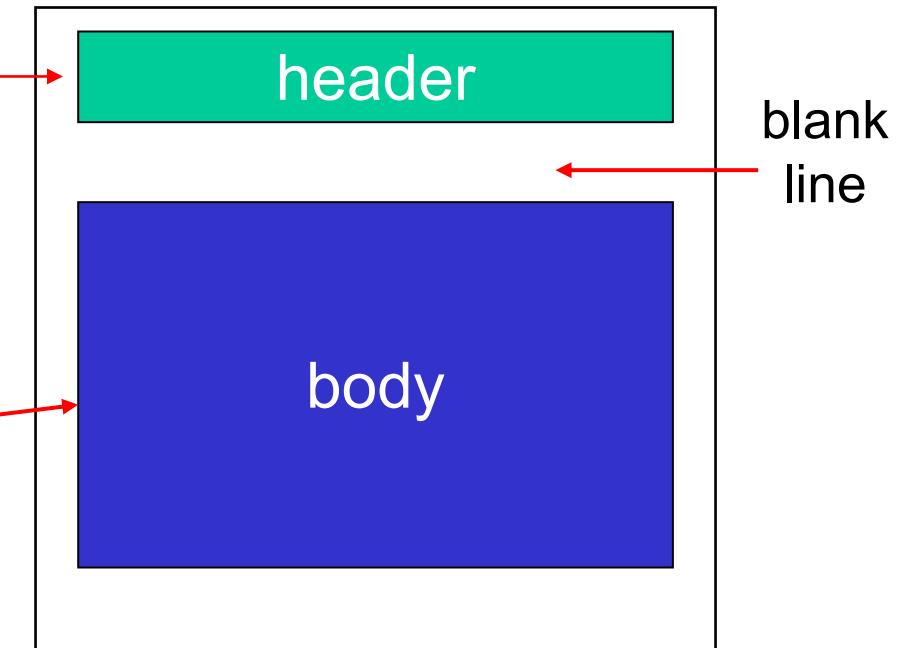
# Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

RFC 822 defines *syntax* for e-mail message itself (like HTML)

- header lines, e.g.,
  - To:
  - From:
  - Subject:

these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the “message”, ASCII characters only



# Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# DNS: Domain Name System

*people:* many identifiers:

- SSN, name, passport #

*Internet hosts, routers:*

- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., cs.umass.edu - used by humans

Q: how to map between IP address and name, and vice versa ?

*Domain Name System:*

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, *implemented as application-layer protocol*
  - complexity at network’s “edge”

# DNS: services, structure

## DNS services

- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

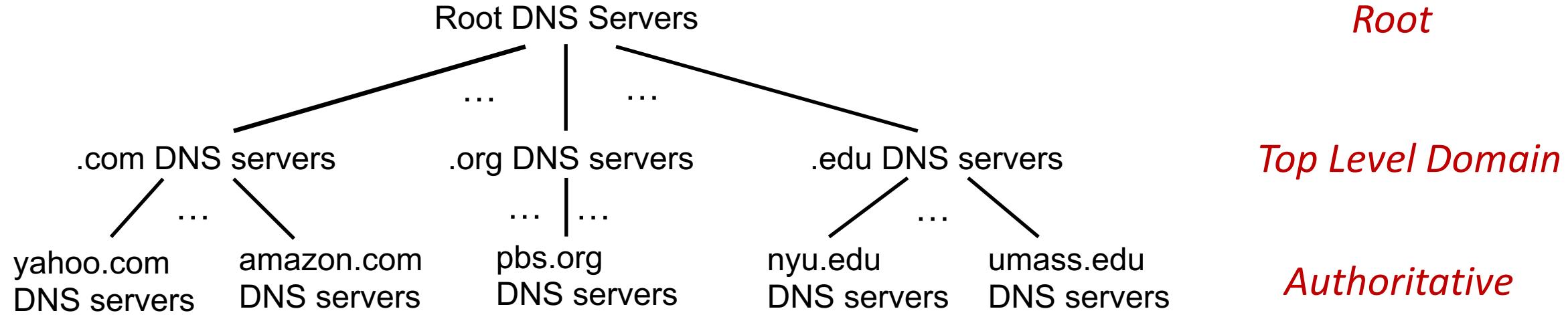
*Q: Why not centralize DNS?*

- single point of failure
- traffic volume
- distant centralized database
- maintenance

*A: doesn't scale!*

- Comcast DNS servers alone: 1 Trillion DNS queries per day

# DNS: a distributed, hierarchical database



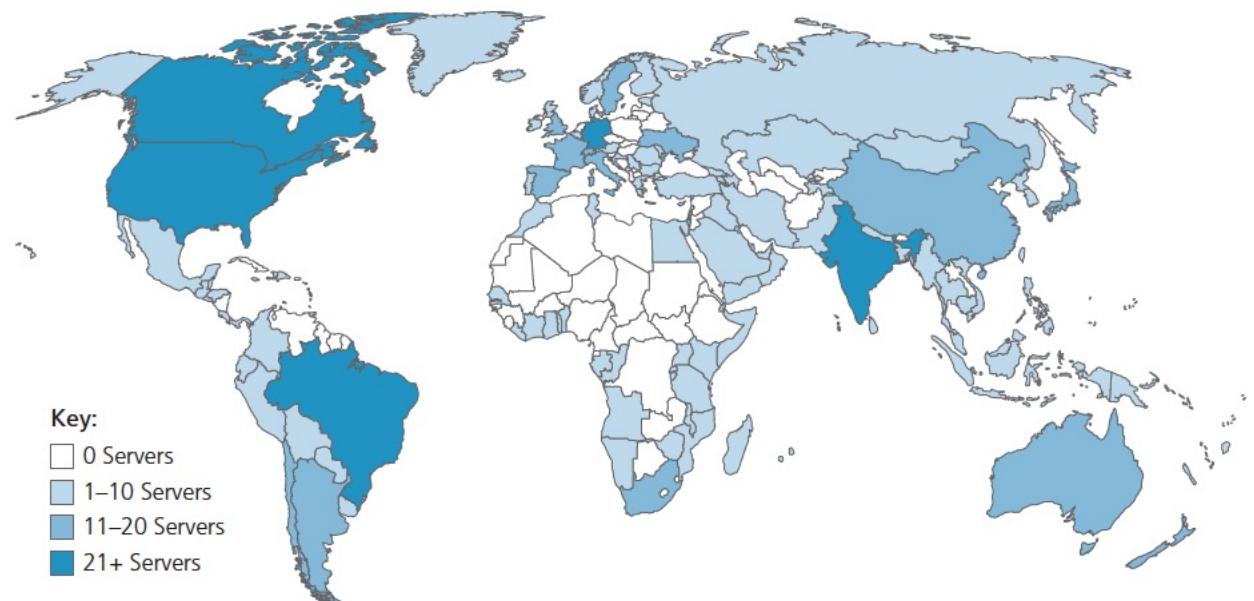
Client wants IP address for [www.amazon.com](http://www.amazon.com); 1<sup>st</sup> approximation:

- client queries root server to find .com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

# DNS: root name servers

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name “servers” worldwide each “server” replicated many times (~200 servers in US)



# TLD: authoritative servers

## Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .edu TLD

## Authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

# Local DNS name servers

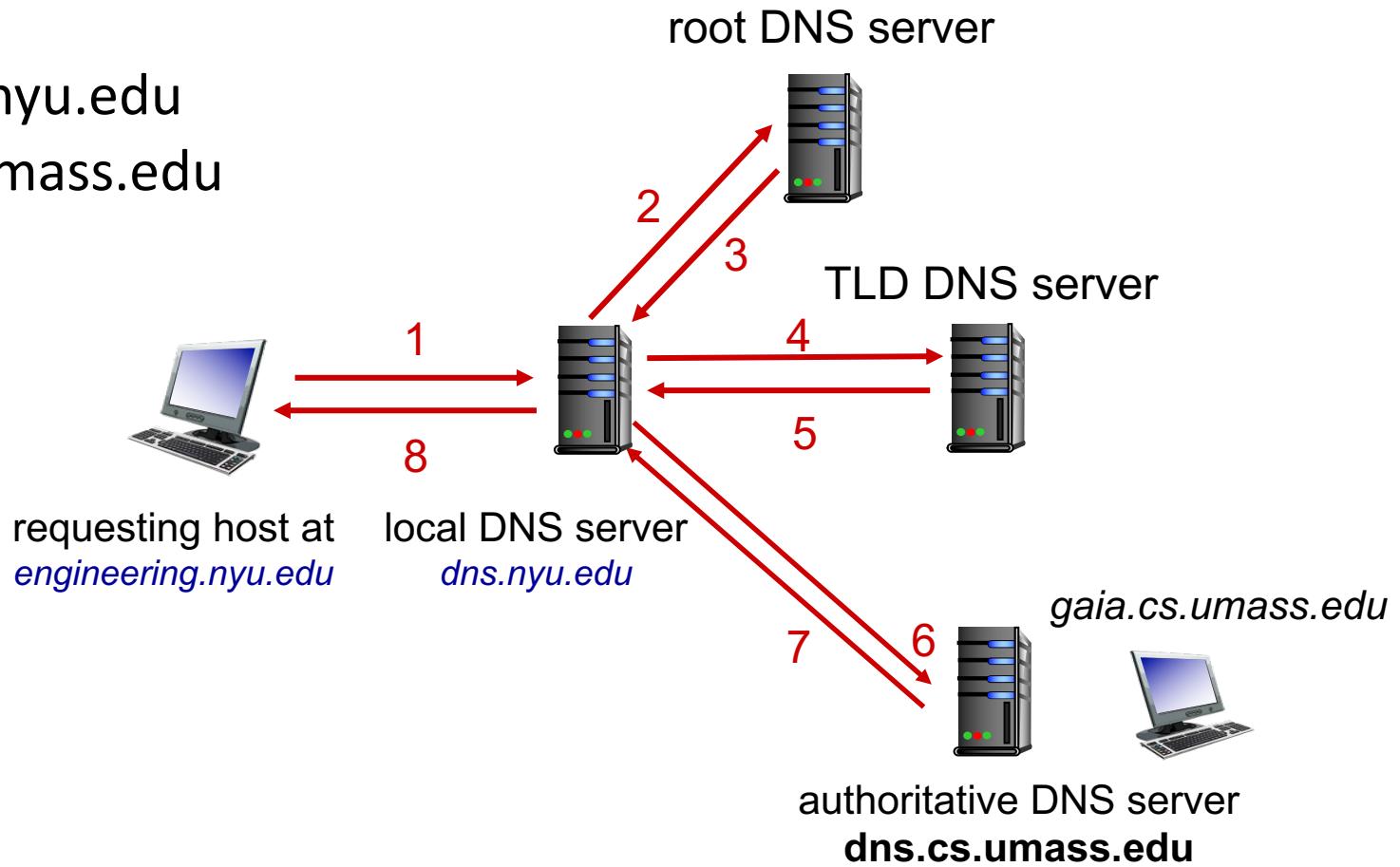
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

# DNS name resolution: iterated query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

## Iterated query:

- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”

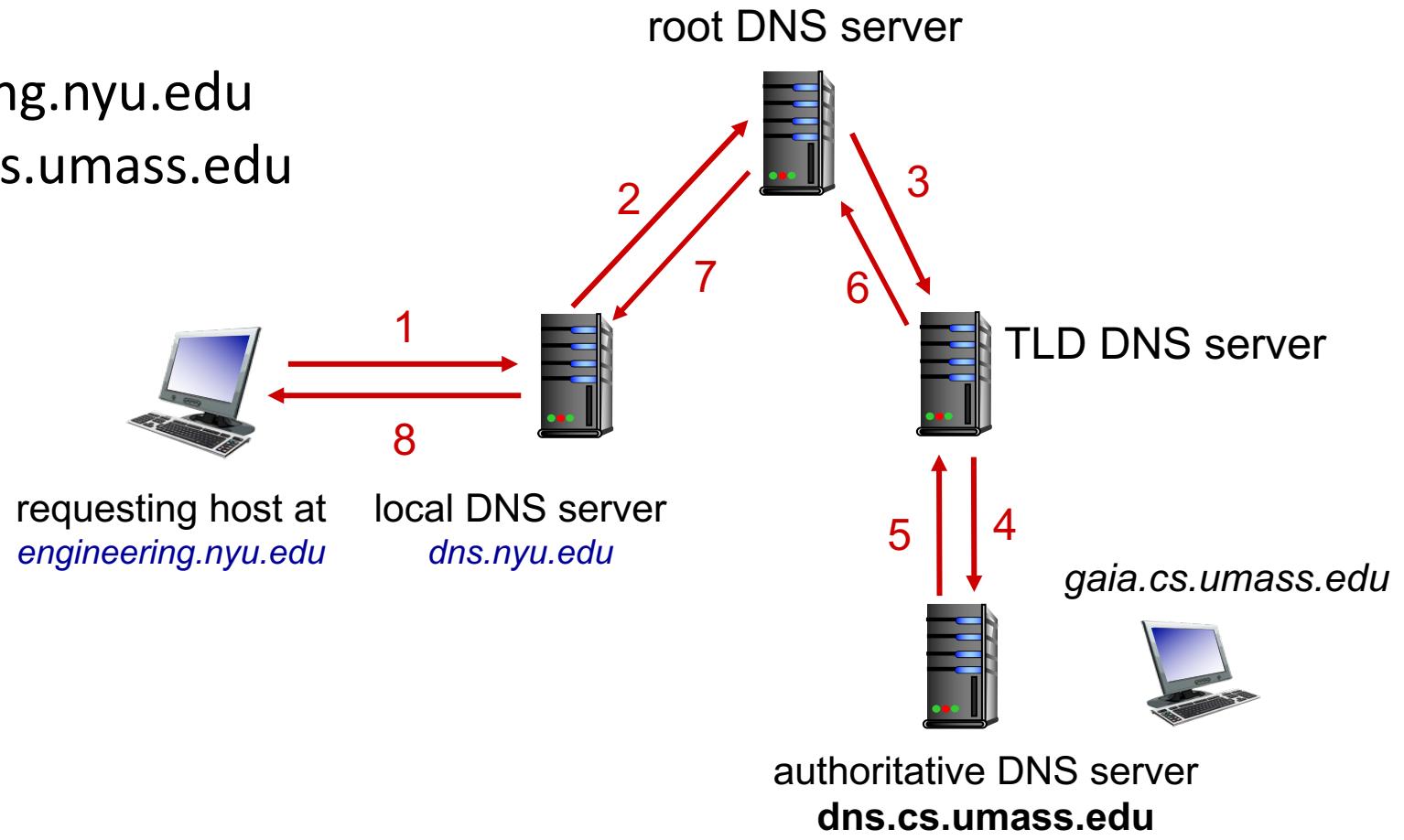


# DNS name resolution: recursive query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

## Recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



# Caching, Updating DNS Records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best-effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire!
- update/notify mechanisms proposed IETF standard
  - RFC 2136

# DNS records

**DNS:** distributed database storing resource records (**RR**)

RR format: (name, value, type, ttl)

## type=A

- name is hostname
- value is IP address

## type=NS

- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

## type=CNAME

- name is alias name for some “canonical” (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name

## type=MX

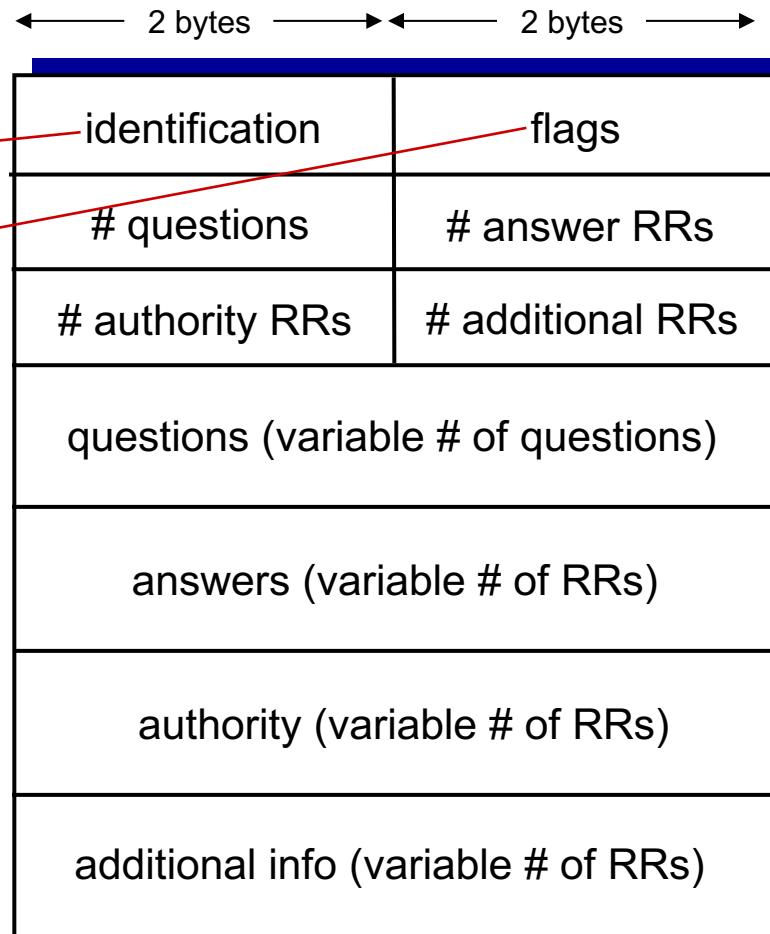
- value is name of mailserver associated with name

# DNS protocol messages

DNS *query* and *reply* messages, both have same *format*:

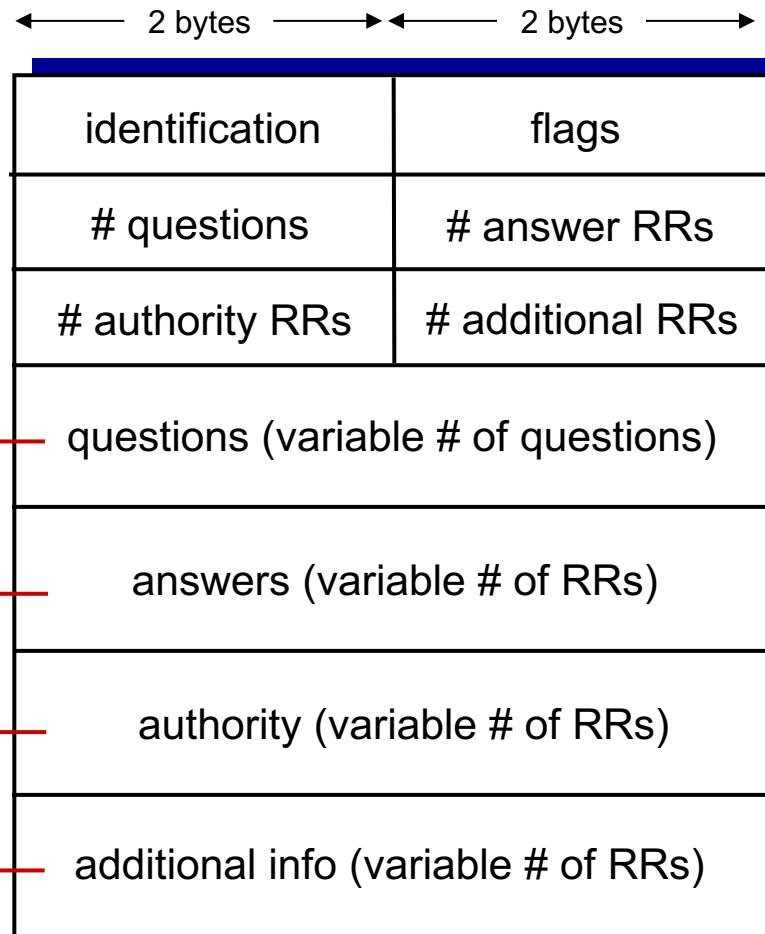
message header:

- **identification:** 16 bit # for query,  
reply to query uses same #
- **flags:**
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



# DNS protocol messages

DNS *query* and *reply* messages, both have same *format*:



# Inserting records into DNS

Example: new startup “Network Utopia”

- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts NS, A RRs into .com TLD server:  
(networkutopia.com, dns1.networkutopia.com, NS)  
(dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server locally with IP address 212.212.212.1
  - type A record for www.networkuptopia.com
  - type MX record for networkutopia.com

# DNS security

## DDoS attacks

- bombard root servers with traffic
  - not successful to date
  - traffic filtering
  - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
  - potentially more dangerous

## Redirect attacks

- man-in-middle
  - intercept DNS queries
- DNS poisoning
  - send bogus replies to DNS server, which caches

## Exploit DNS for DDoS

- send queries with spoofed source address: target IP
- requires amplification

DNSSEC  
[RFC 4033]

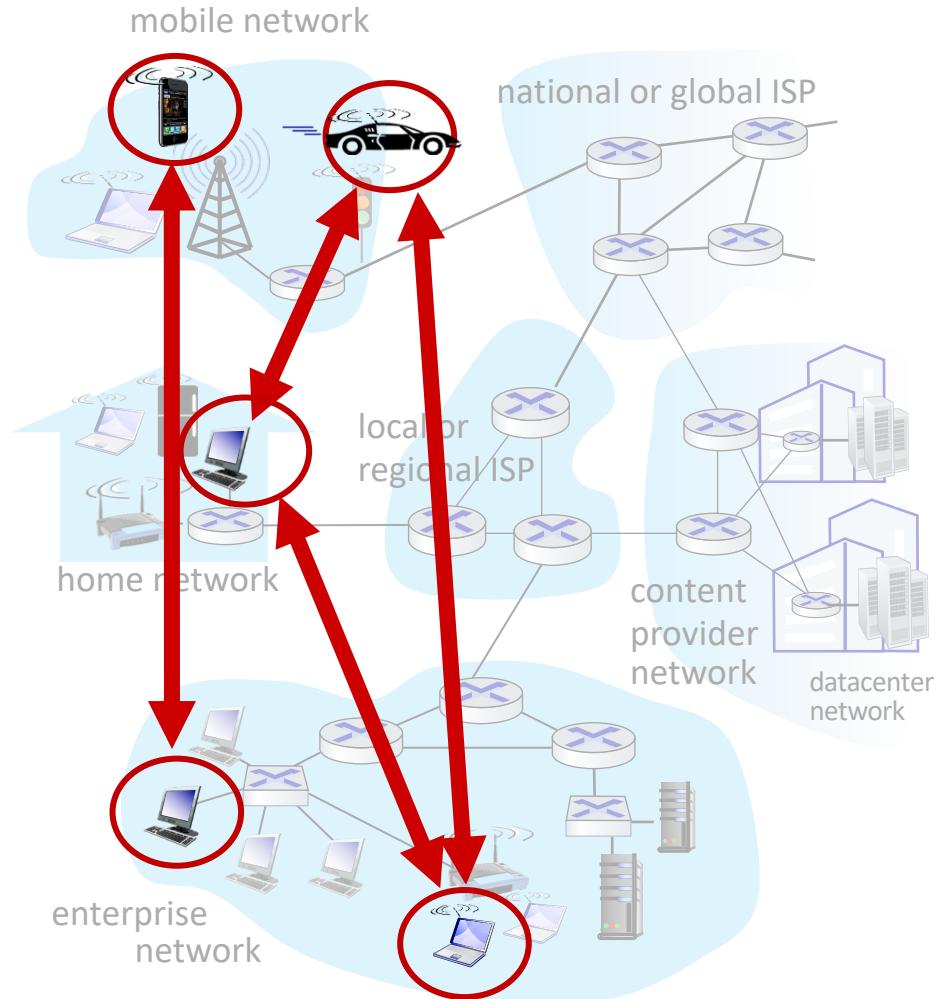
# Application Layer: Overview

- Principles of network applications
  - Web and HTTP
  - E-mail, SMTP, IMAP
  - The Domain Name System DNS
- P2P applications
  - video streaming and content distribution networks
  - socket programming with UDP and TCP



# Peer-to-peer (P2P) architecture

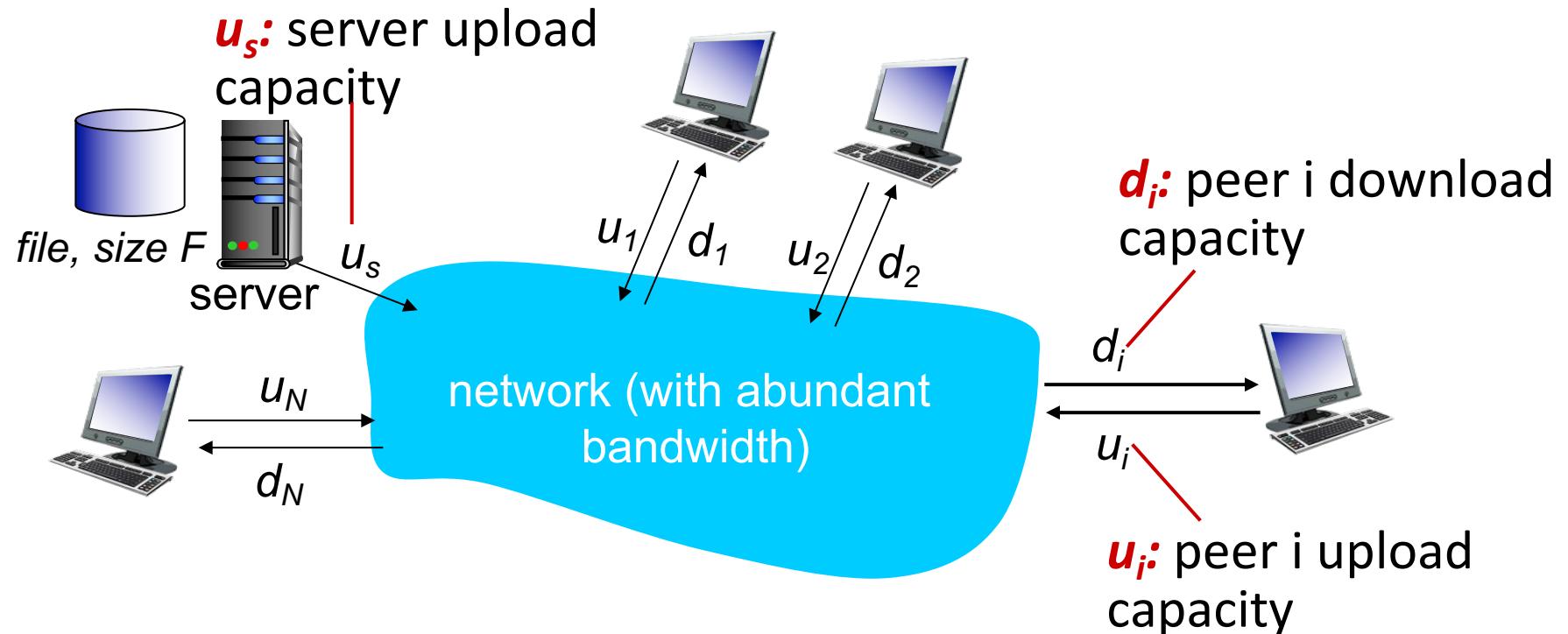
- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)



# File distribution: client-server vs P2P

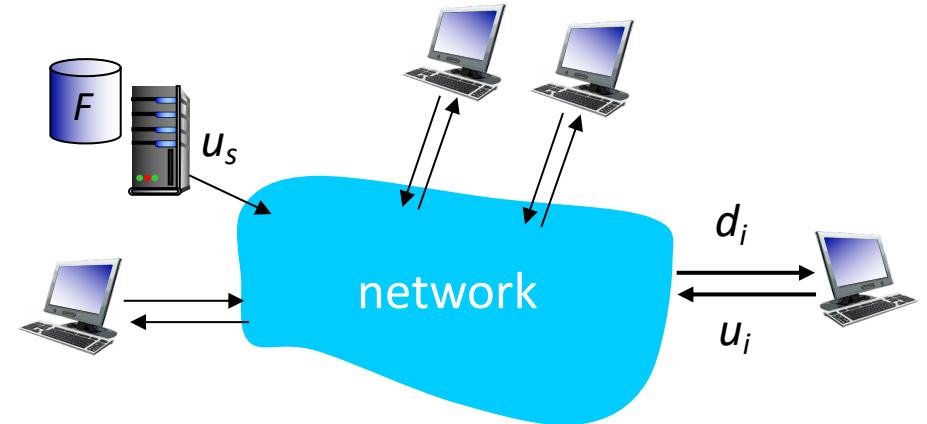
**Q:** how much time to distribute file (size  $F$ ) from one server to  $N$  peers?

- peer upload/download capacity is limited resource



# File distribution time: client-server

- *server transmission*: must sequentially send (upload)  $N$  file copies:
  - time to send one copy:  $F/u_s$
  - time to send  $N$  copies:  $NF/u_s$
- *client*: each client must download file copy
  - $d_{min}$  = min client download rate
  - min client download time:  $F/d_{min}$



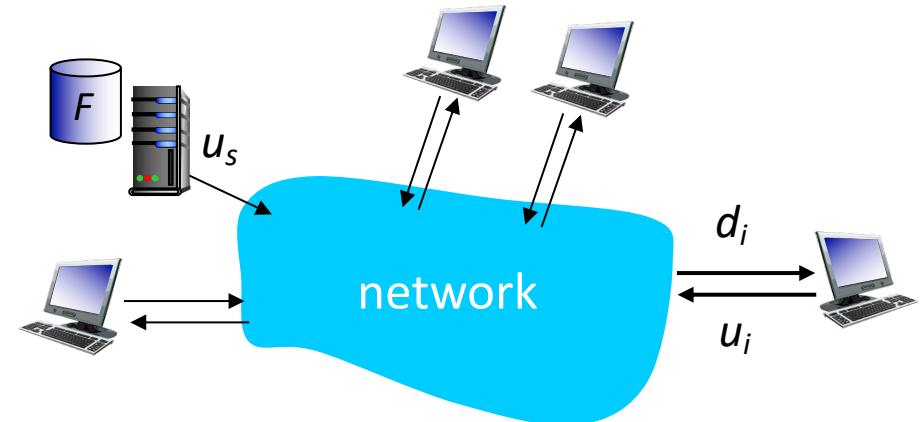
*time to distribute  $F$   
to  $N$  clients using  
client-server approach*

$$D_{c-s} \geq \max\{NF/u_s, F/d_{min}\}$$

increases linearly in  $N$

# File distribution time: P2P

- *server transmission*: must upload at least one copy:
  - time to send one copy:  $F/u_s$
- *client*: each client must download file copy
  - min client download time:  $F/d_{min}$
- *clients*: as aggregate must download  $NF$  bits
  - max upload rate (limiting max download rate) is  $u_s + \sum u_i$



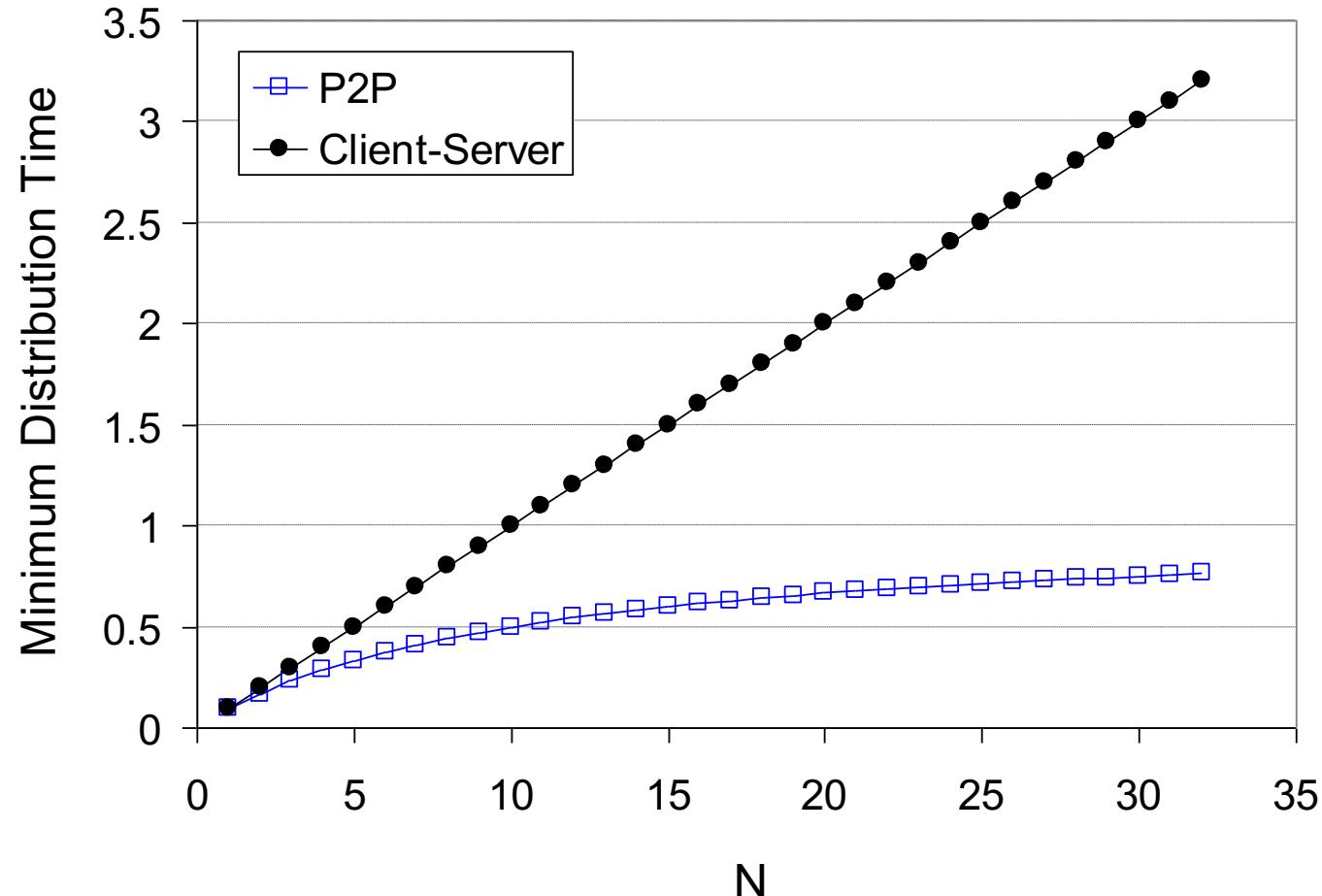
time to distribute  $F$   
to  $N$  clients using  
P2P approach

$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

increases linearly in  $N$  ...  
... but so does this, as each peer brings service capacity

# Client-server vs. P2P: example

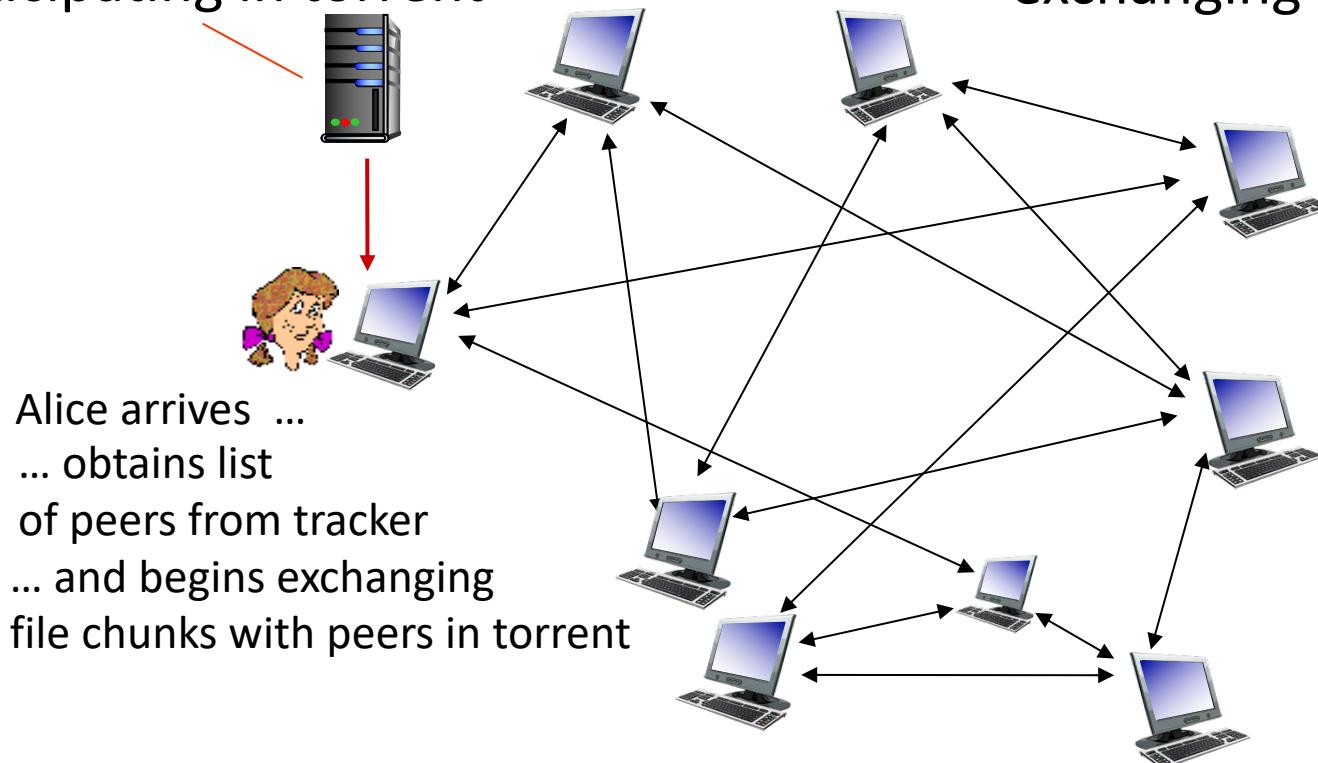
client upload rate =  $u$ ,  $F/u = 1$  hour,  $u_s = 10u$ ,  $d_{min} \geq u_s$



# P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

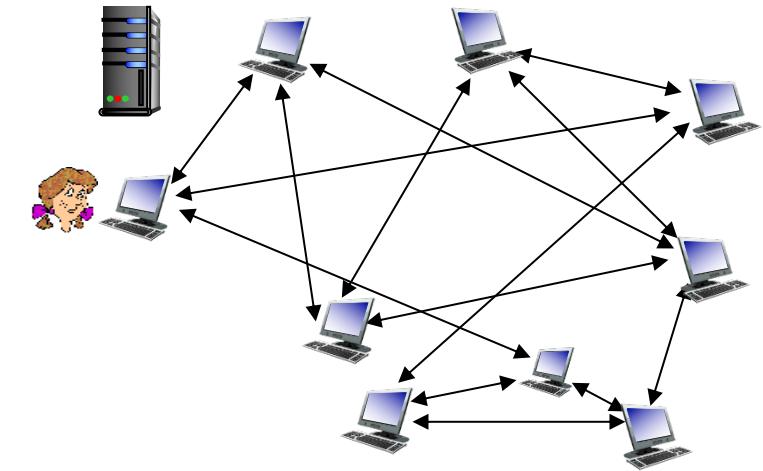
*tracker:* tracks peers  
participating in torrent



*torrent:* group of peers  
exchanging chunks of a file

# P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn*: peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent



# BitTorrent: requesting, sending file chunks

## Requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

# BitTorrent: requesting, sending file chunks

## Requesting chunks:

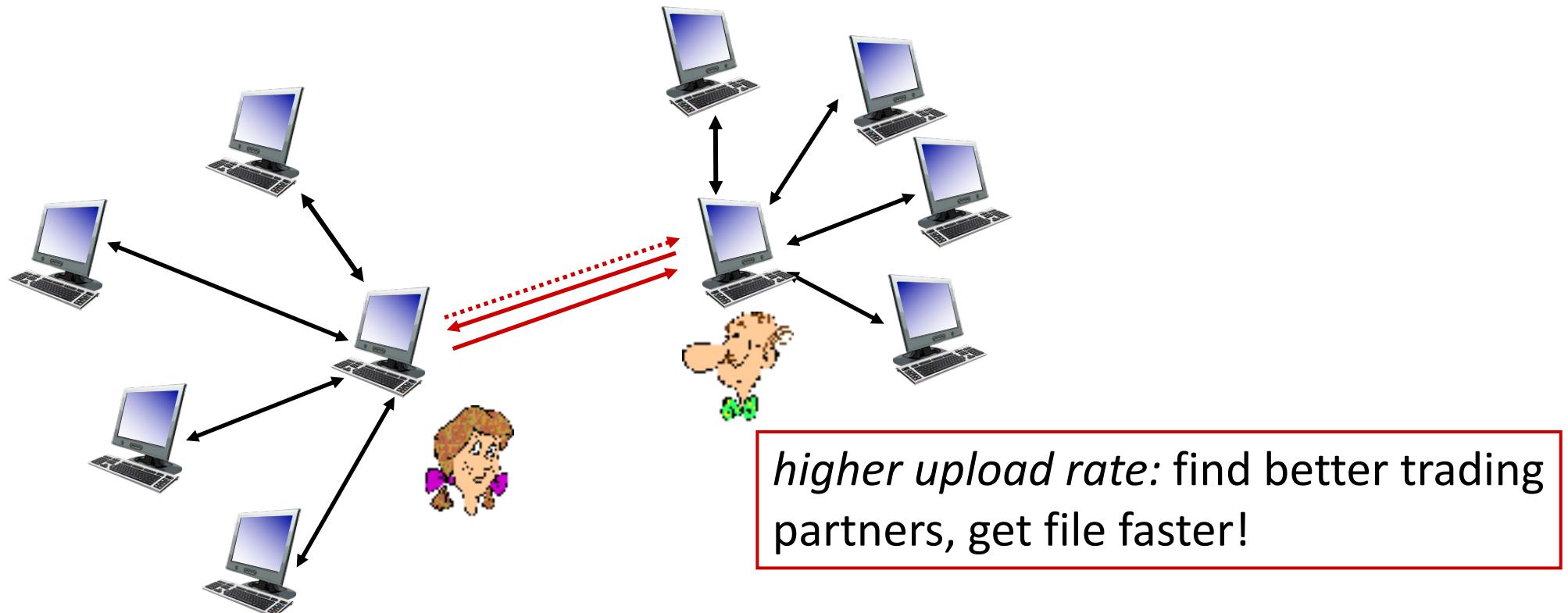
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

## Sending chunks: tit-for-tat

- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - “optimistically unchoke” this peer
  - newly chosen peer may join top 4

# BitTorrent: tit-for-tat

- (1) Alice “optimistically unchoke” Bob
- (2) Alice becomes one of Bob’s top-four providers; Bob reciprocates
- (3) Bob becomes one of Alice’s top-four providers



# Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# Video Streaming and CDNs: context

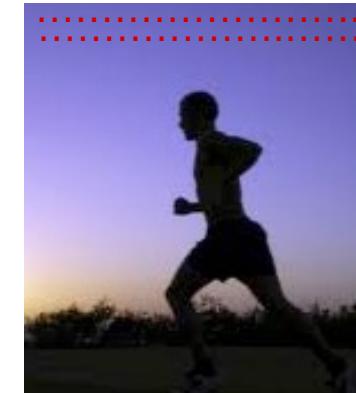
- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution: distributed, application-level infrastructure*



# Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)

*spatial coding example:* instead of sending  $N$  values of same color (all purple), send only two values: color value (*purple*) and number of repeated values ( $N$ )



frame *i*

*temporal coding example:* instead of sending complete frame at  $i+1$ , send only differences from frame  $i$

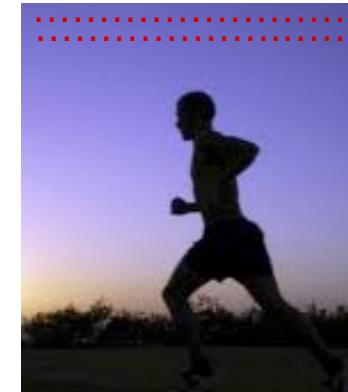


frame *i+1*

# Multimedia: video

- CBR: (constant bit rate): video encoding rate fixed
- VBR: (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples:
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, 64Kbps – 12 Mbps)

*spatial coding example:* instead of sending  $N$  values of same color (all purple), send only two values: color value (*purple*) and number of repeated values ( $N$ )



frame  $i$

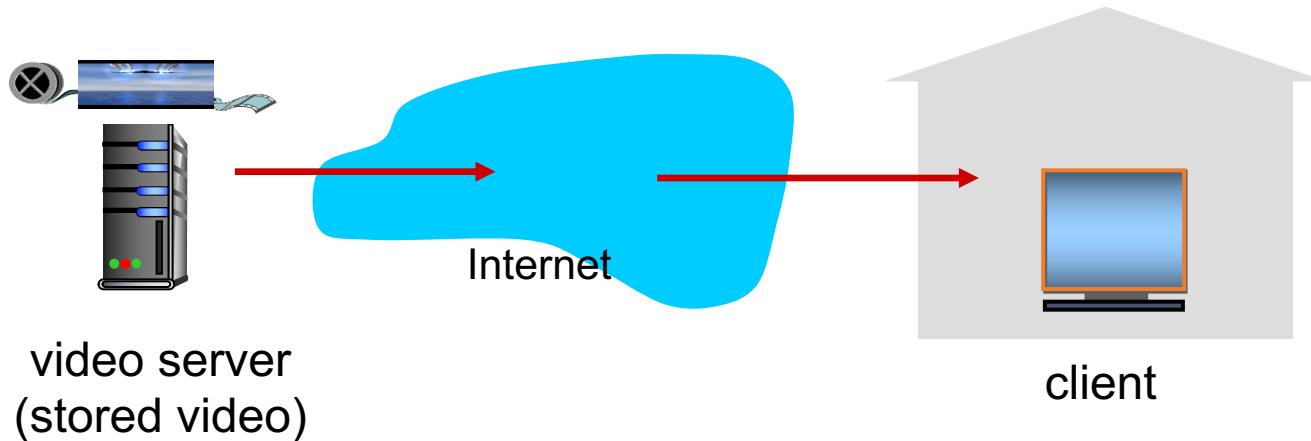
*temporal coding example:* instead of sending complete frame at  $i+1$ , send only differences from frame  $i$



frame  $i+1$

# Streaming stored video

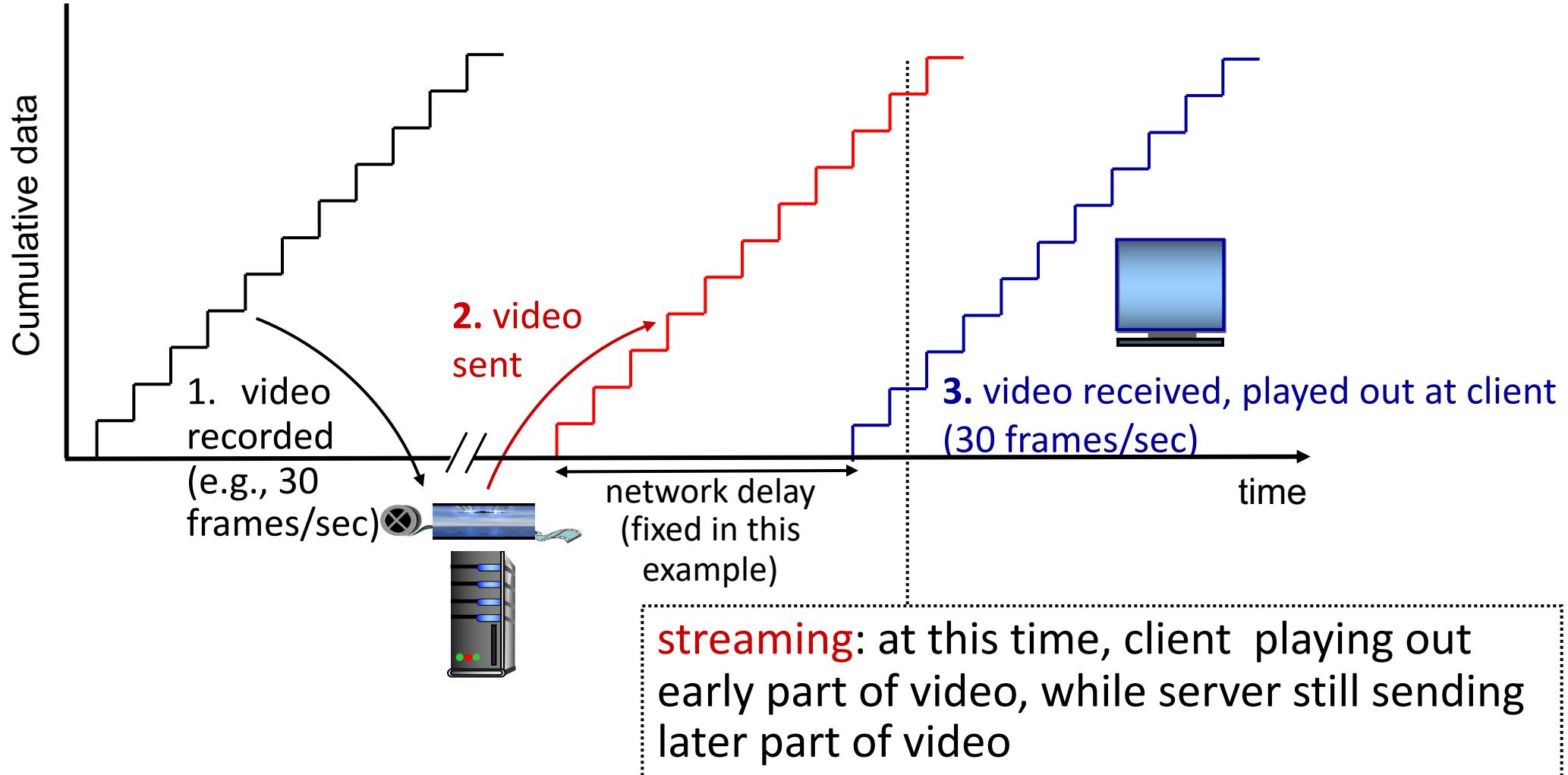
simple scenario:



Main challenges:

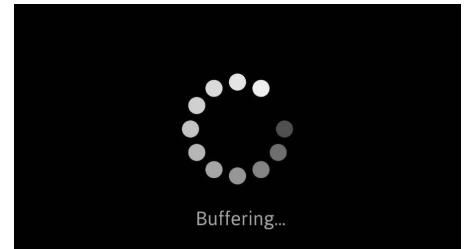
- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, in access network, in network core, at video server)
- packet loss and delay due to congestion will delay playout, or result in poor video quality

# Streaming stored video

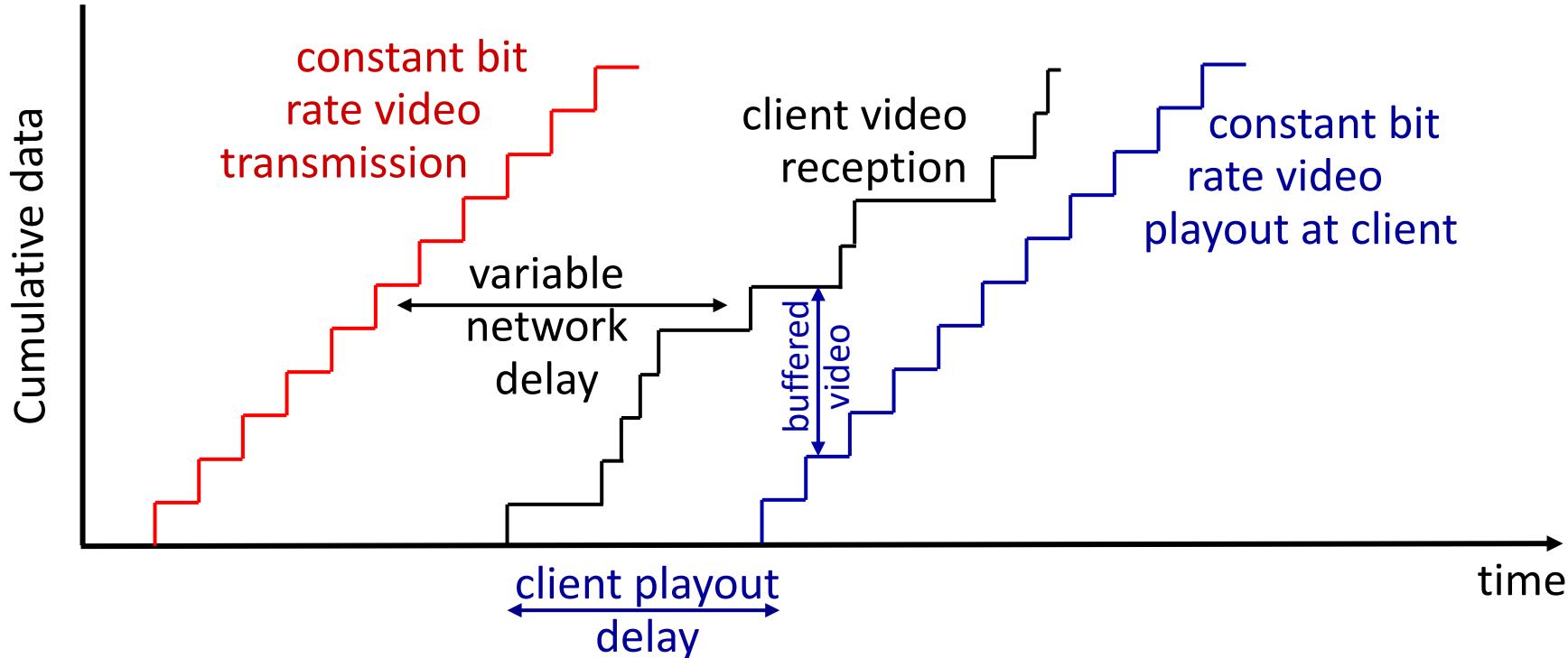


# Streaming stored video: challenges

- **continuous playout constraint:** once client playout begins, playback must match original timing
  - ... but **network delays are variable** (jitter), so will need **client-side buffer** to match playout requirements
- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted



# Streaming stored video: playout buffering



- *client-side buffering and playout delay:* compensate for network-added delay, delay jitter

# Streaming multimedia: DASH

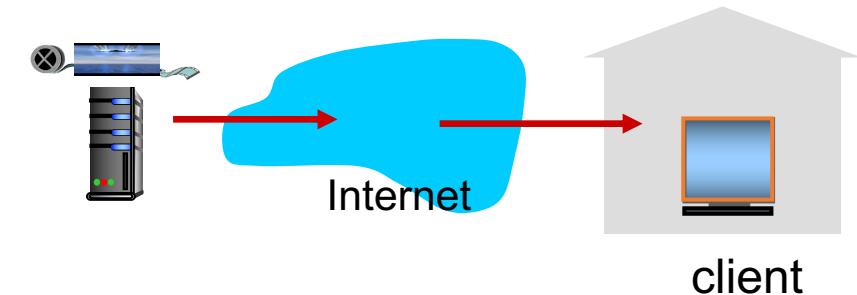
- **DASH: Dynamic, Adaptive Streaming over HTTP**

- **server:**

- divides video file into multiple chunks
- each chunk stored, encoded at different rates
- *manifest file*: provides URLs for different chunks

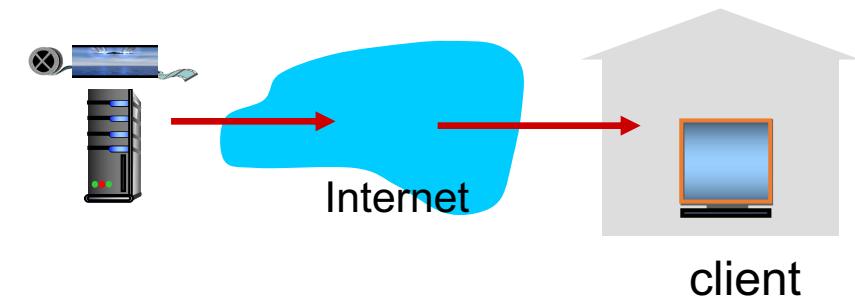
- **client:**

- periodically measures server-to-client bandwidth
- consulting manifest, requests one chunk at a time
  - chooses maximum coding rate sustainable given current bandwidth
  - can choose different coding rates at different points in time (depending on available bandwidth at time)



# Streaming multimedia: DASH

- “*intelligence*” at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is “close” to client or has high available bandwidth)



Streaming video = encoding + DASH + playout buffering

# Content distribution networks (CDNs)

- **challenge:** how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- **option 1:** single, large “mega-server”
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

....quite simply: this solution *doesn't scale*

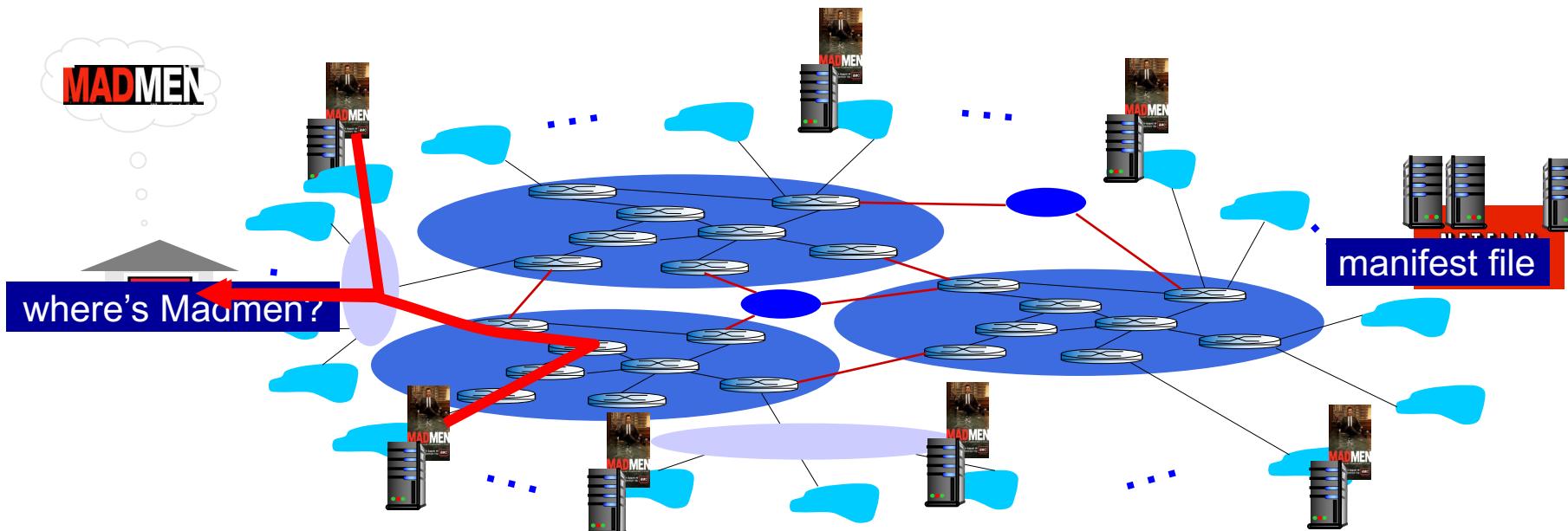
# Content distribution networks (CDNs)

- **challenge:** how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- **option 2:** store/serve multiple copies of videos at multiple geographically distributed sites (**CDN**)
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - Akamai: 240,000 servers deployed in more than 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in IXPs near (but not within) access networks
    - used by Limelight



# Content distribution networks (CDNs)

- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested



# Content distribution networks (CDNs)



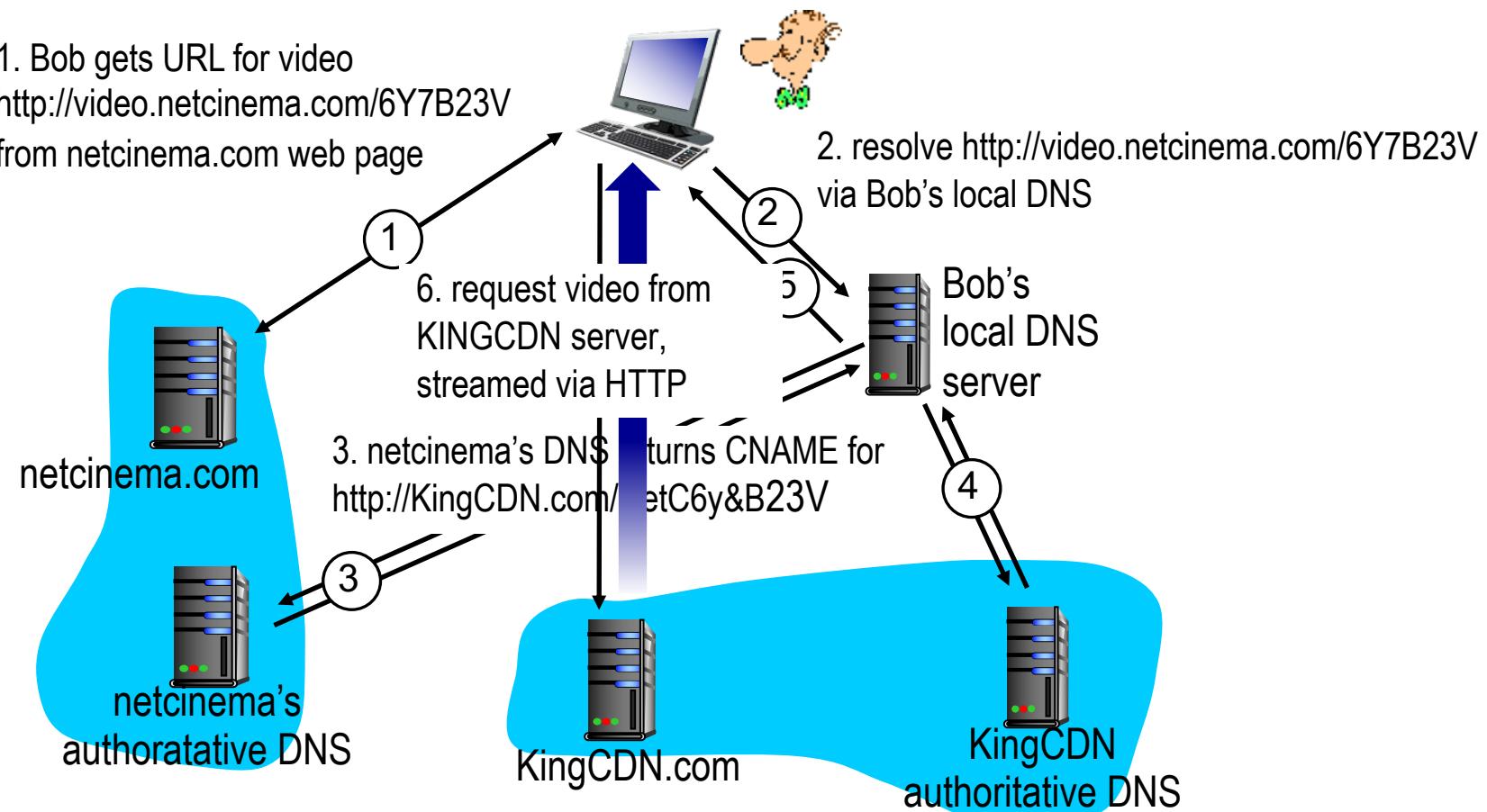
*OTT challenges:* coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

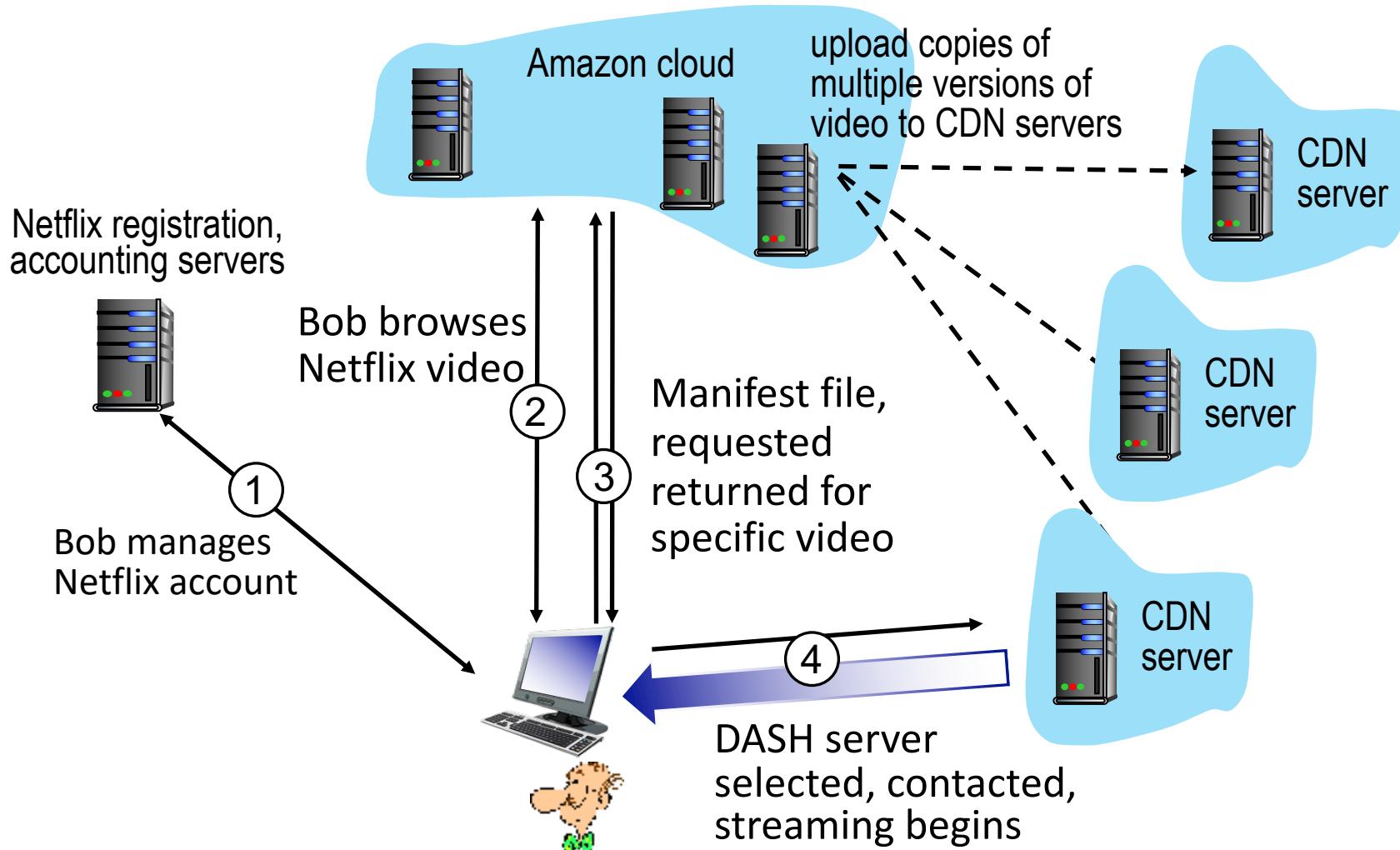
# CDN content access: a closer look

Bob (client) requests video <http://netcinema.com/6Y7B23V>

- video stored in CDN at <http://KingCDN.com/NetC6y&B23V>



# Case study: Netflix



# Application Layer: Overview

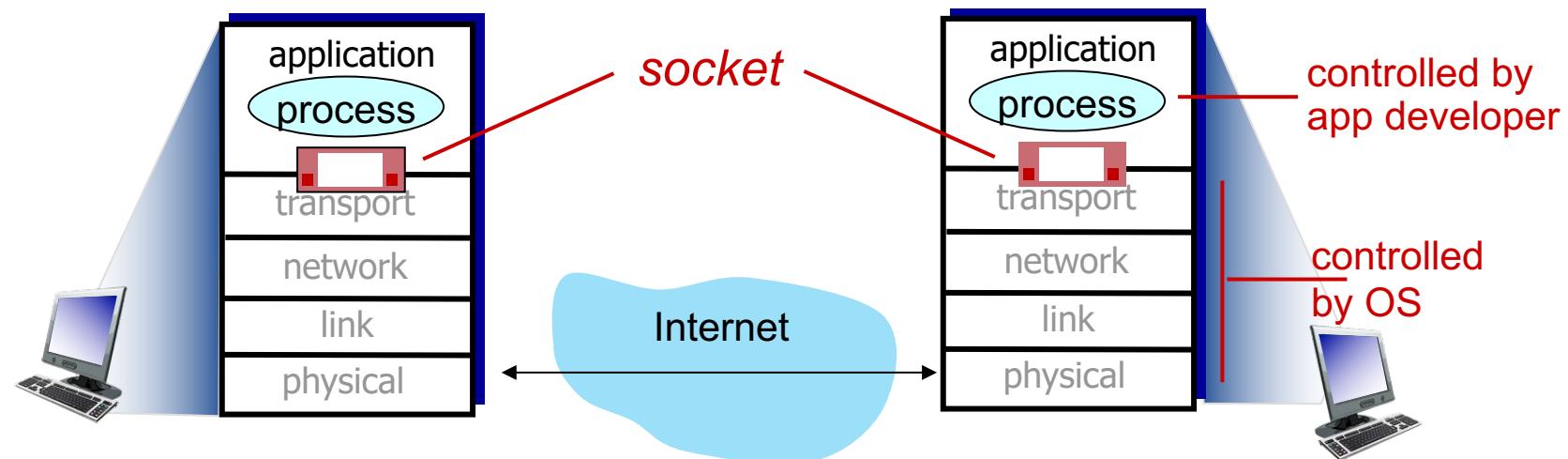
- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- **socket programming with UDP and TCP**



# Socket programming

*goal:* learn how to build client/server applications that communicate using sockets

*socket:* door between application process and end-end-transport protocol



# Socket programming

Two socket types for two transport services:

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

Application Example:

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

# Socket programming with UDP

**UDP:** no “connection” between client & server

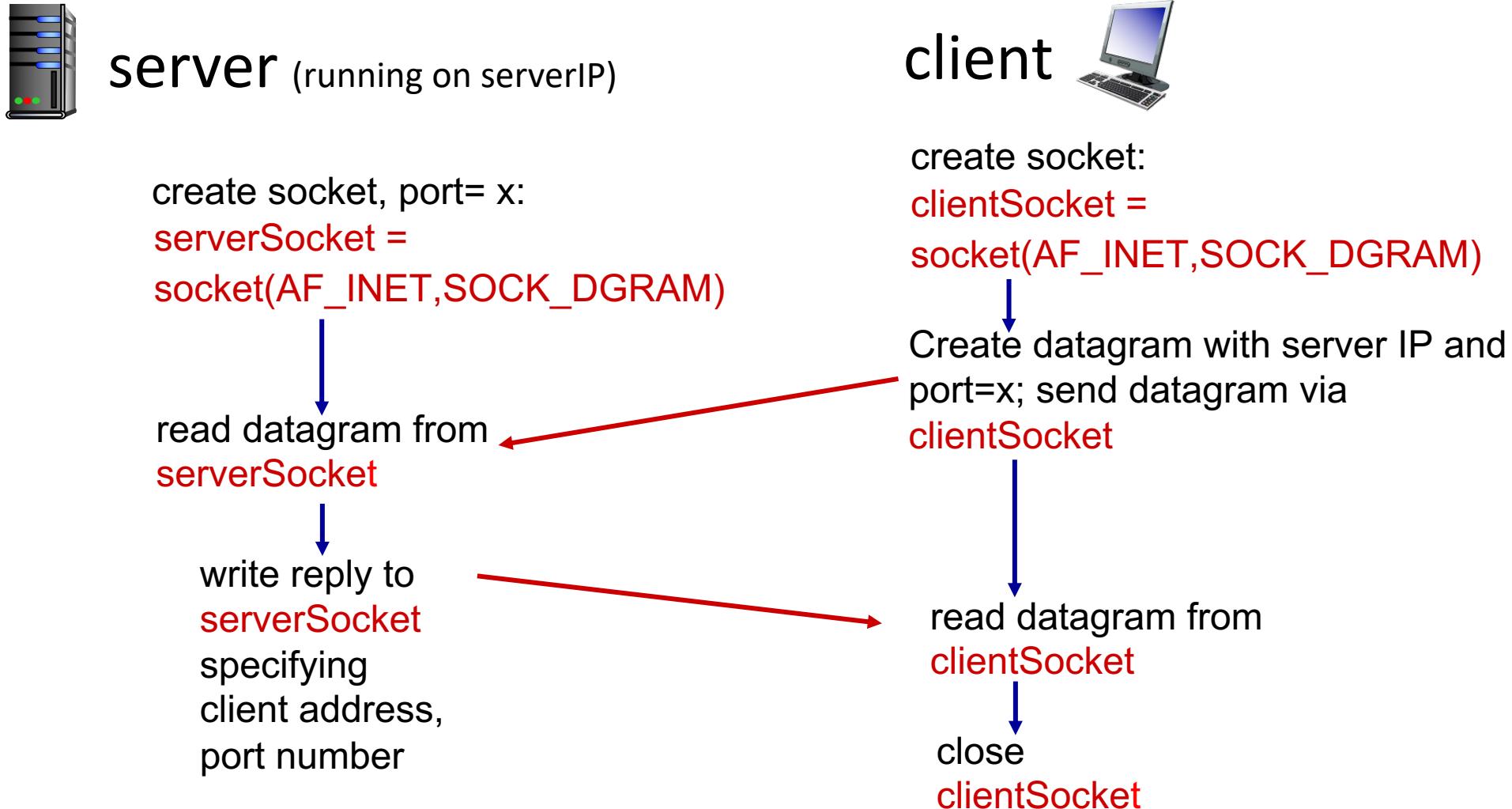
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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

**Application viewpoint:**

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

# Client/server socket interaction: UDP



# Example app: UDP client

## *Python UDPCClient*

```
include Python's socket library → from socket import *
serverName = 'hostname'
serverPort = 12000
create UDP socket for server → clientSocket = socket(AF_INET,
                                                    SOCK_DGRAM)
get user keyboard input → message = raw_input('Input lowercase sentence:')
attach server name, port to message; send into socket → clientSocket.sendto(message.encode(),
                           (serverName, serverPort))
read reply characters from socket into string → modifiedMessage, serverAddress =
                                               clientSocket.recvfrom(2048)
print out received string and close socket → print modifiedMessage.decode()
                                              clientSocket.close()
```

# Example app: UDP server

## *Python UDPServer*

```
from socket import *
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(("", serverPort))
                                         print ("The server is ready to receive")
loop forever → while True:
Read from UDP socket into message, getting →   message, clientAddress = serverSocket.recvfrom(2048)
client's address (client IP and port)           modifiedMessage = message.decode().upper()
                                               serverSocket.sendto(modifiedMessage.encode(),
send upper case string back to this client →   clientAddress)
```

# Socket programming with TCP

## Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

## Client contacts server by:

- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket*: client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

### Application viewpoint

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

# Client/server socket interaction: TCP



server (running on hostid)



client

create socket,  
port=x, for incoming  
request:  
`serverSocket = socket()`

wait for incoming  
connection request  
`connectionSocket = serverSocket.accept()`

read request from  
`connectionSocket`

write reply to  
`connectionSocket`

close  
`connectionSocket`

TCP  
connection setup

create socket,  
connect to `hostid`, port=x  
`clientSocket = socket()`

send request using  
`clientSocket`

read reply from  
`clientSocket`

close  
`clientSocket`

# Example app: TCP client

## *Python TCPClient*

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server,  
remote port 12000 → clientSocket = socket(AF\_INET, SOCK\_STREAM)

No need to attach server name, port → clientSocket.connect((serverName, serverPort))

# Example app: TCP server

## *Python TCPServer*

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                           encode())
connectionSocket.close()
```

create TCP welcoming socket → from socket import \*

server begins listening for incoming TCP requests → serverPort = 12000  
→ serverSocket = socket(AF\_INET,SOCK\_STREAM)  
→ serverSocket.bind(("","serverPort"))  
→ serverSocket.listen(1)

loop forever → print 'The server is ready to receive'  
→ while True:

server waits on accept() for incoming requests, new socket created on return → connectionSocket, addr = serverSocket.accept()

read bytes from socket (but not address as in UDP) → sentence = connectionSocket.recv(1024).decode()  
→ capitalizedSentence = sentence.upper()  
→ connectionSocket.send(capitalizedSentence.  
 encode())

close connection to this client (but *not* welcoming socket) → connectionSocket.close()

# Chapter 2: Summary

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming:
  - TCP, UDP sockets

# Chapter 2: Summary

Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data*: info(payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- “complexity at network edge”

# Additional Chapter 2 slides

# Transport layer: overview

*Our goal:*

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

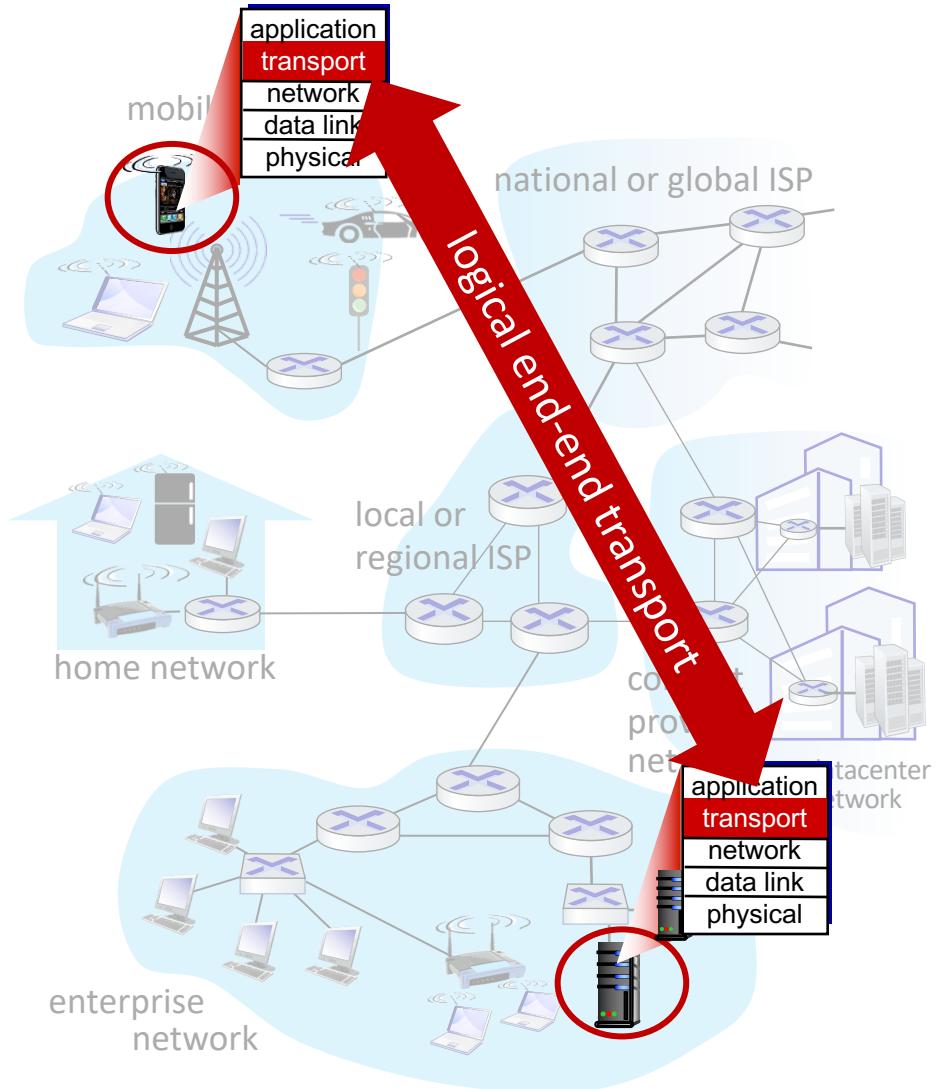
# Transport layer: roadmap

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



# Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - 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:*

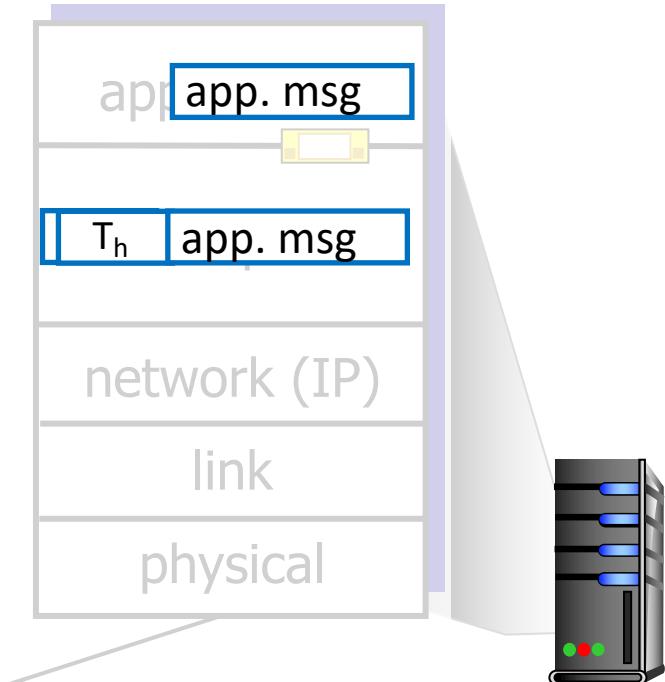
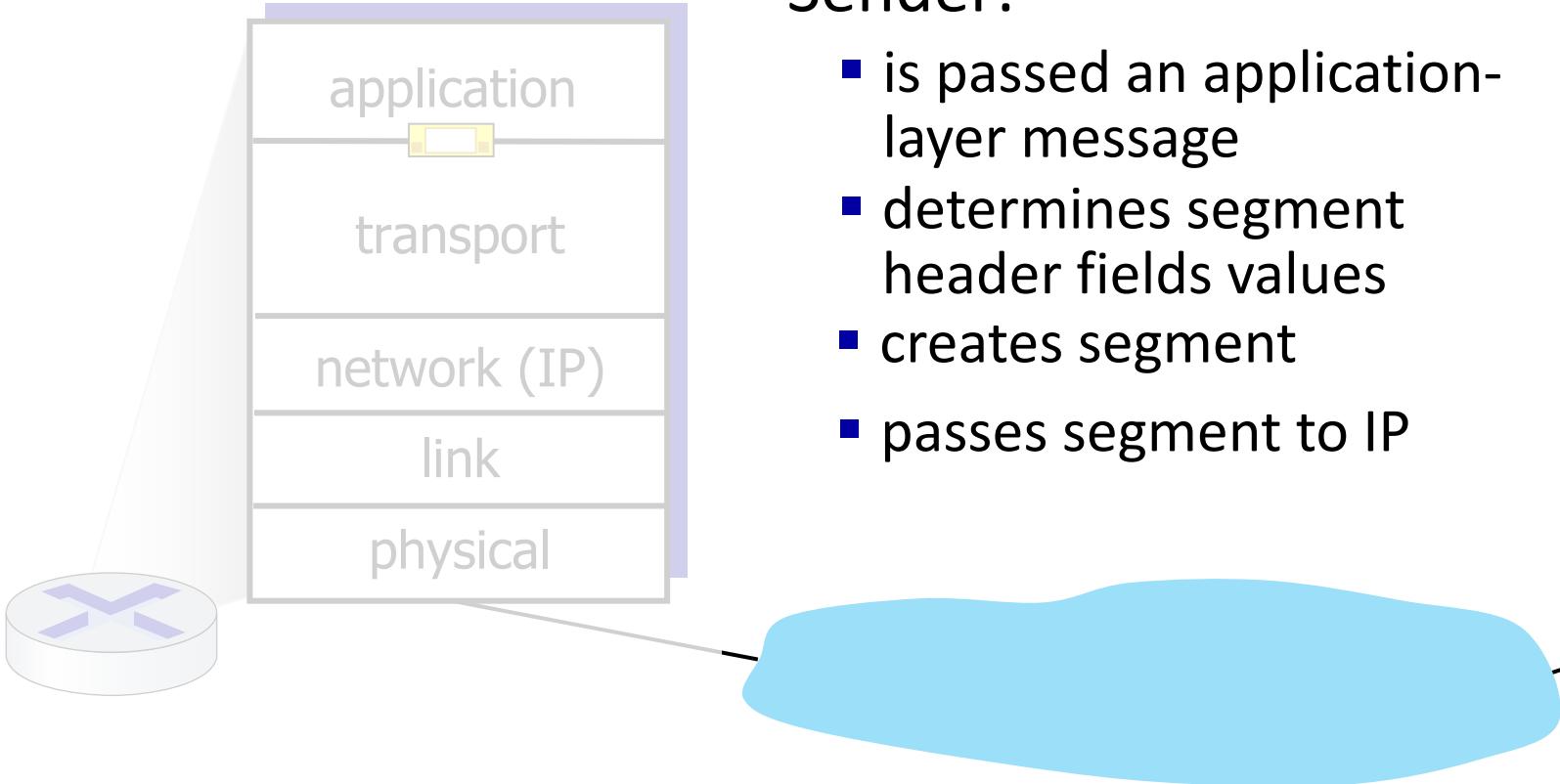
*12 kids in Ann's house sending letters to 12 kids in Bill's house:*

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

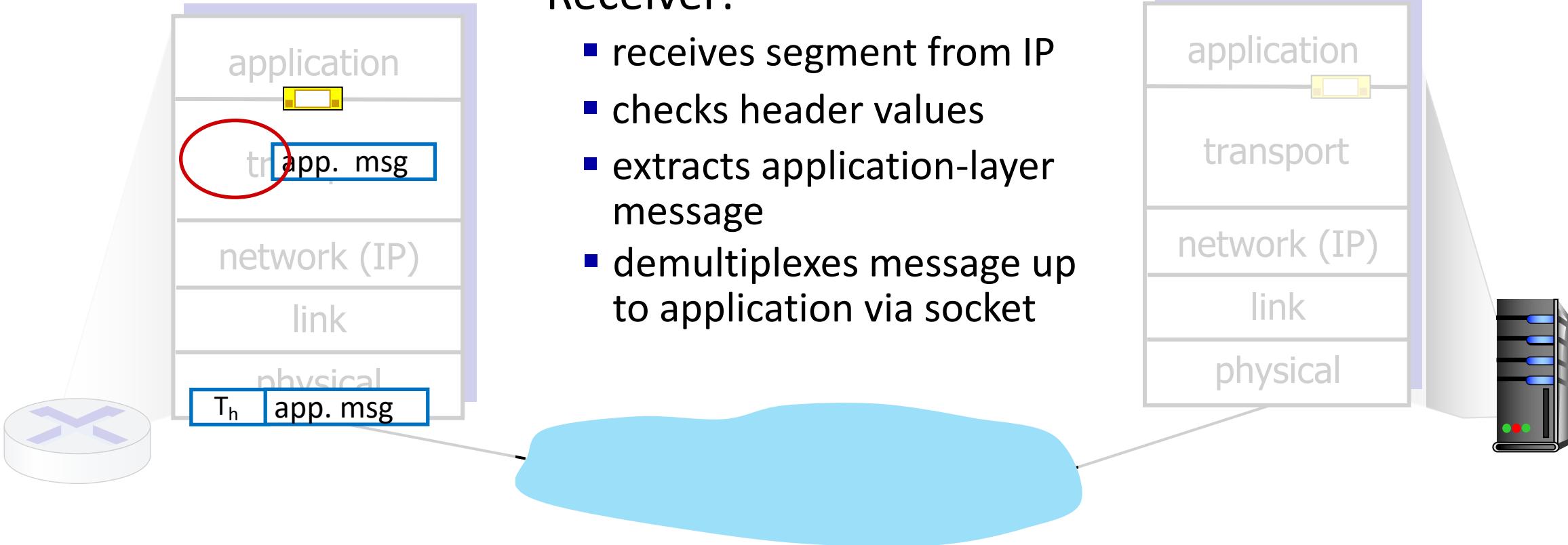
# Transport Layer Actions

Sender:

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

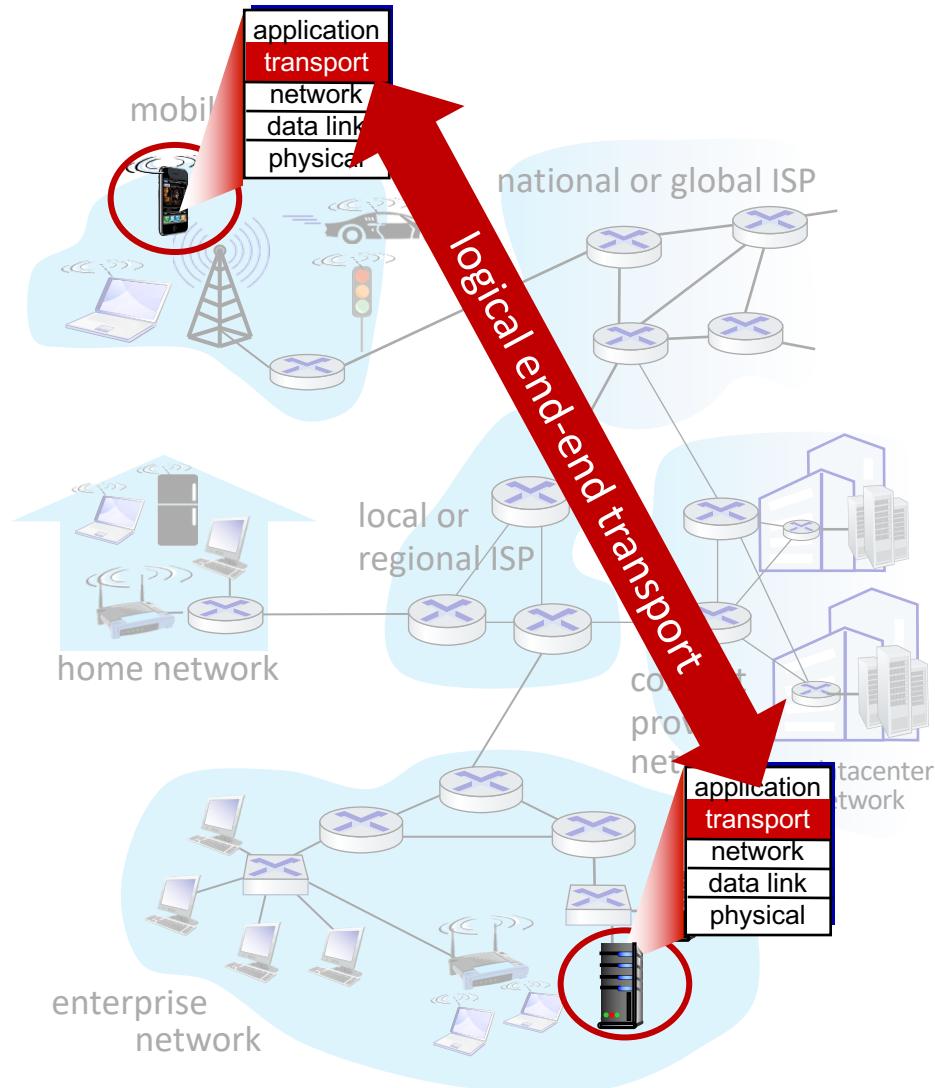


# Transport Layer Actions



# Two principal Internet transport protocols

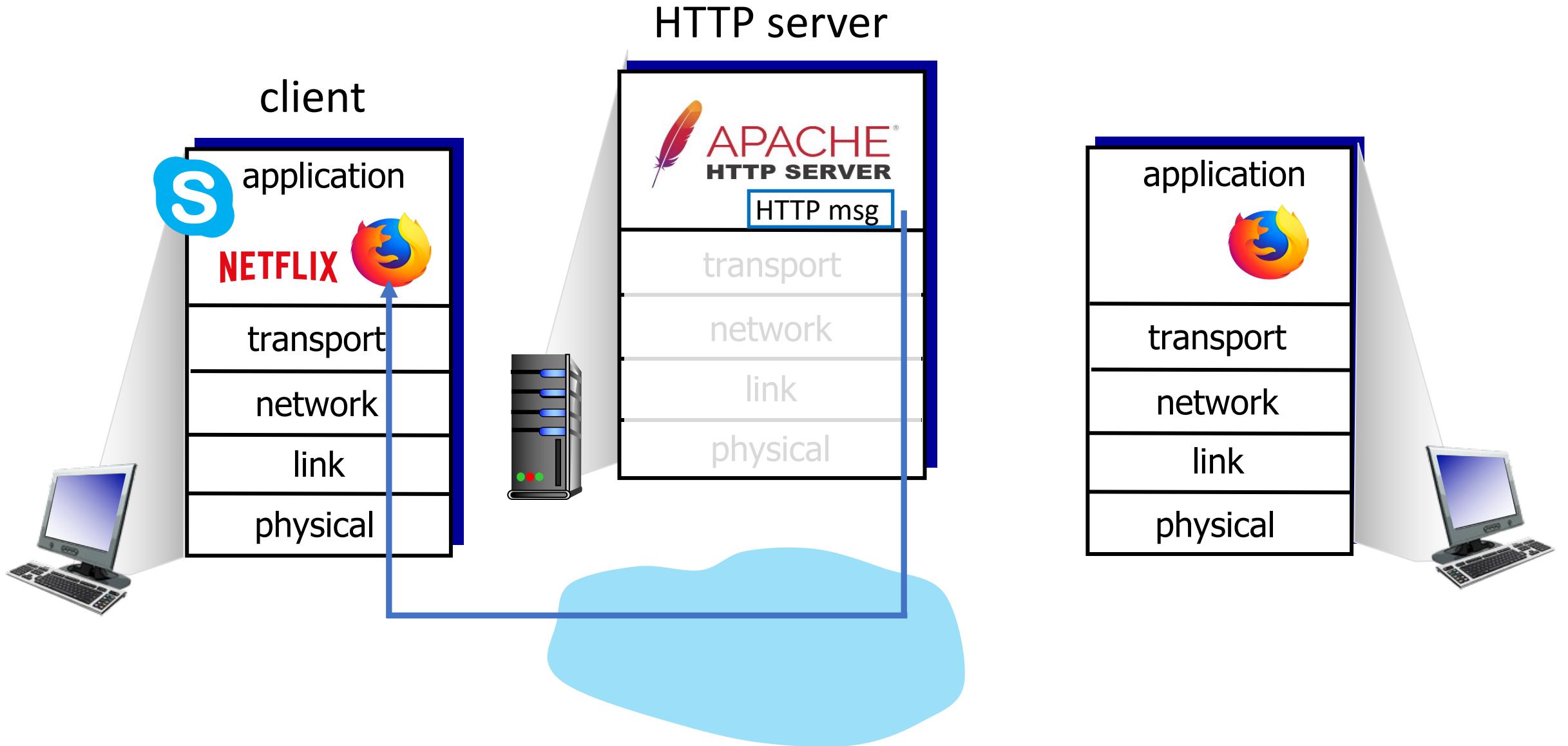
- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth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

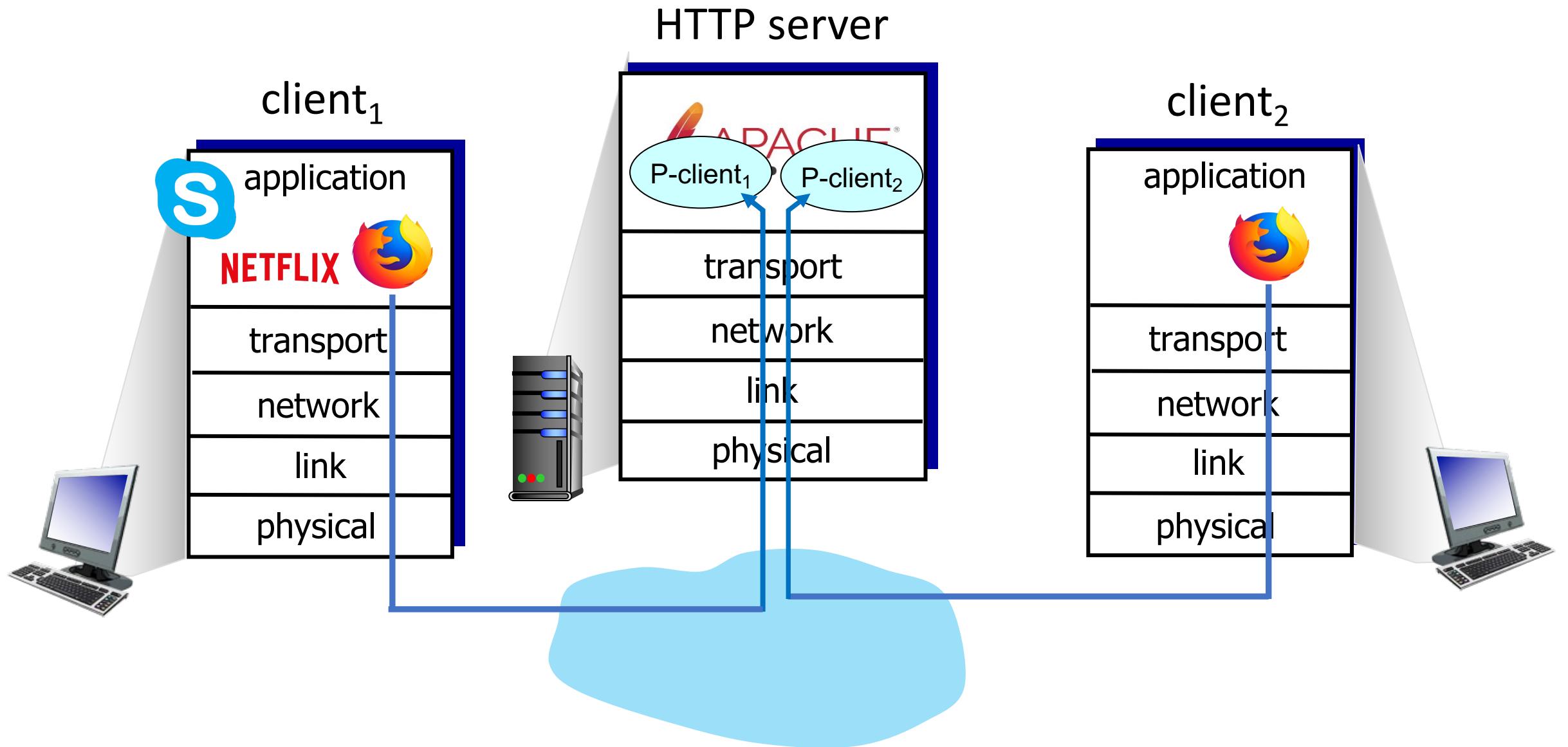


# Chapter 3: roadmap

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







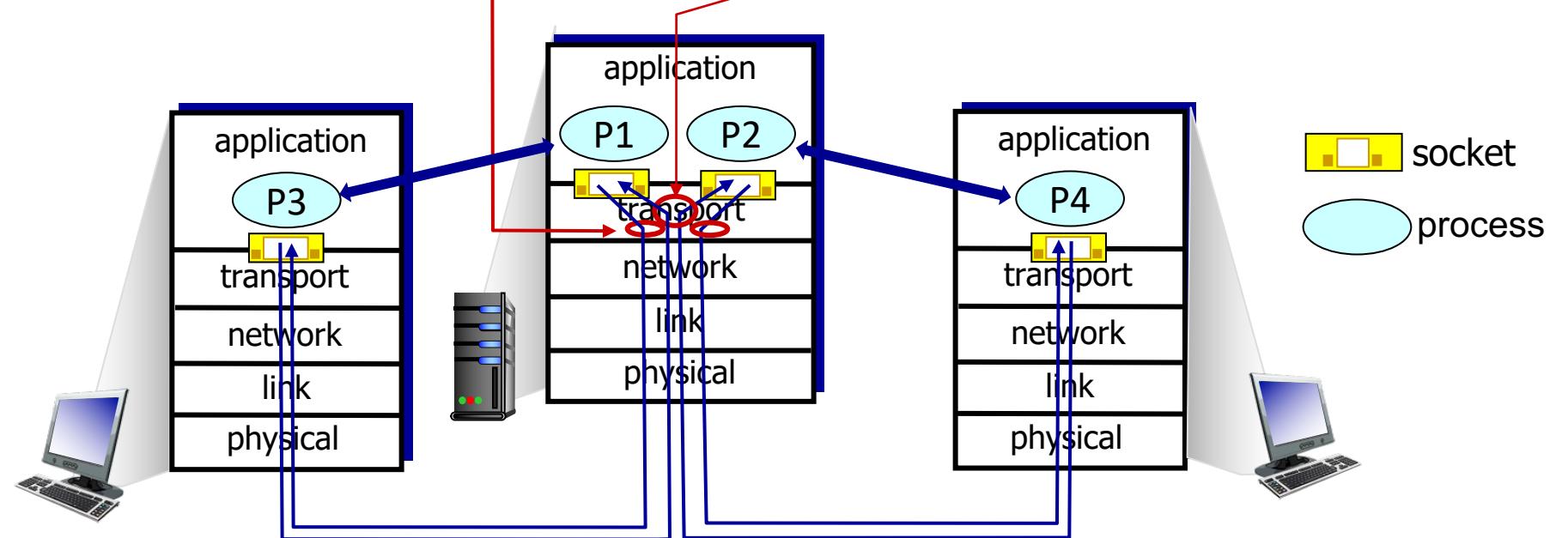
# Multiplexing/demultiplexing

*multiplexing at sender:*

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

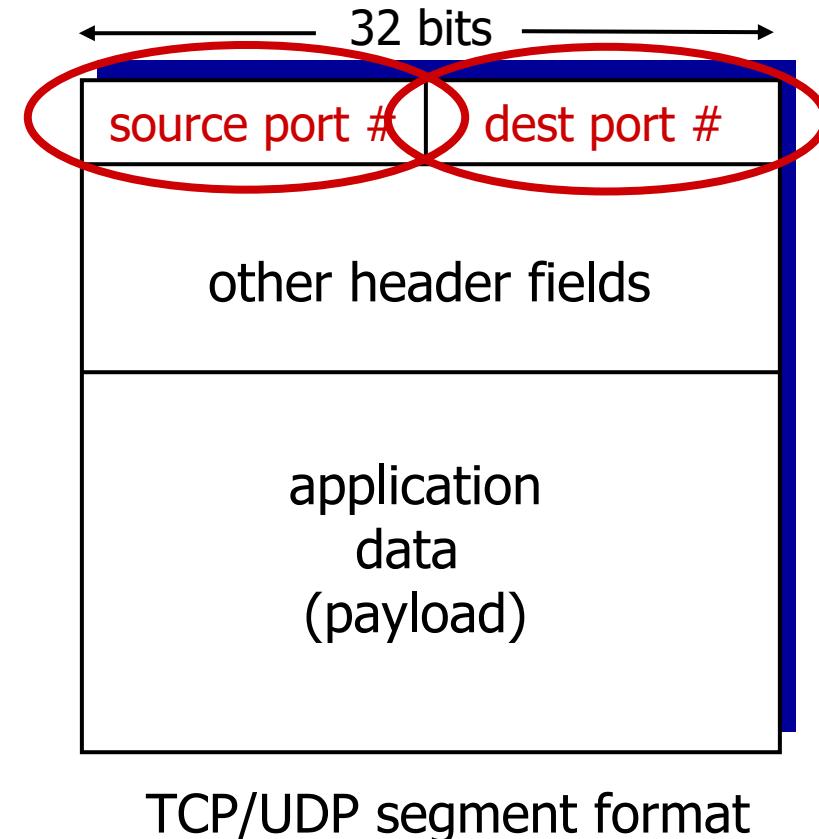
*demultiplexing at receiver:*

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



# 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 #



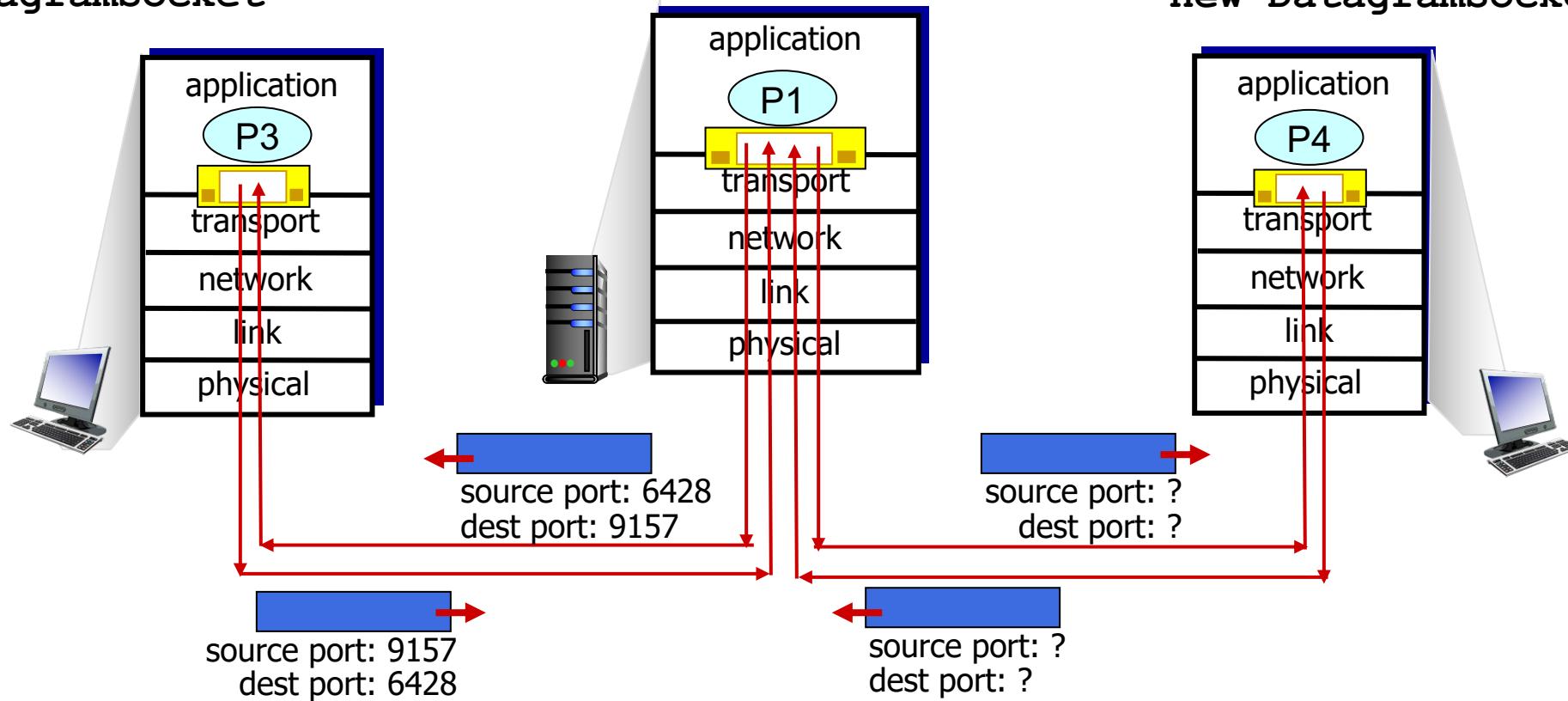
IP/UDP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

# Connectionless demultiplexing: an example

```
DatagramSocket mySocket2 =  
new DatagramSocket  
(9157);
```

```
DatagramSocket  
serverSocket = new  
DatagramSocket  
(6428);
```

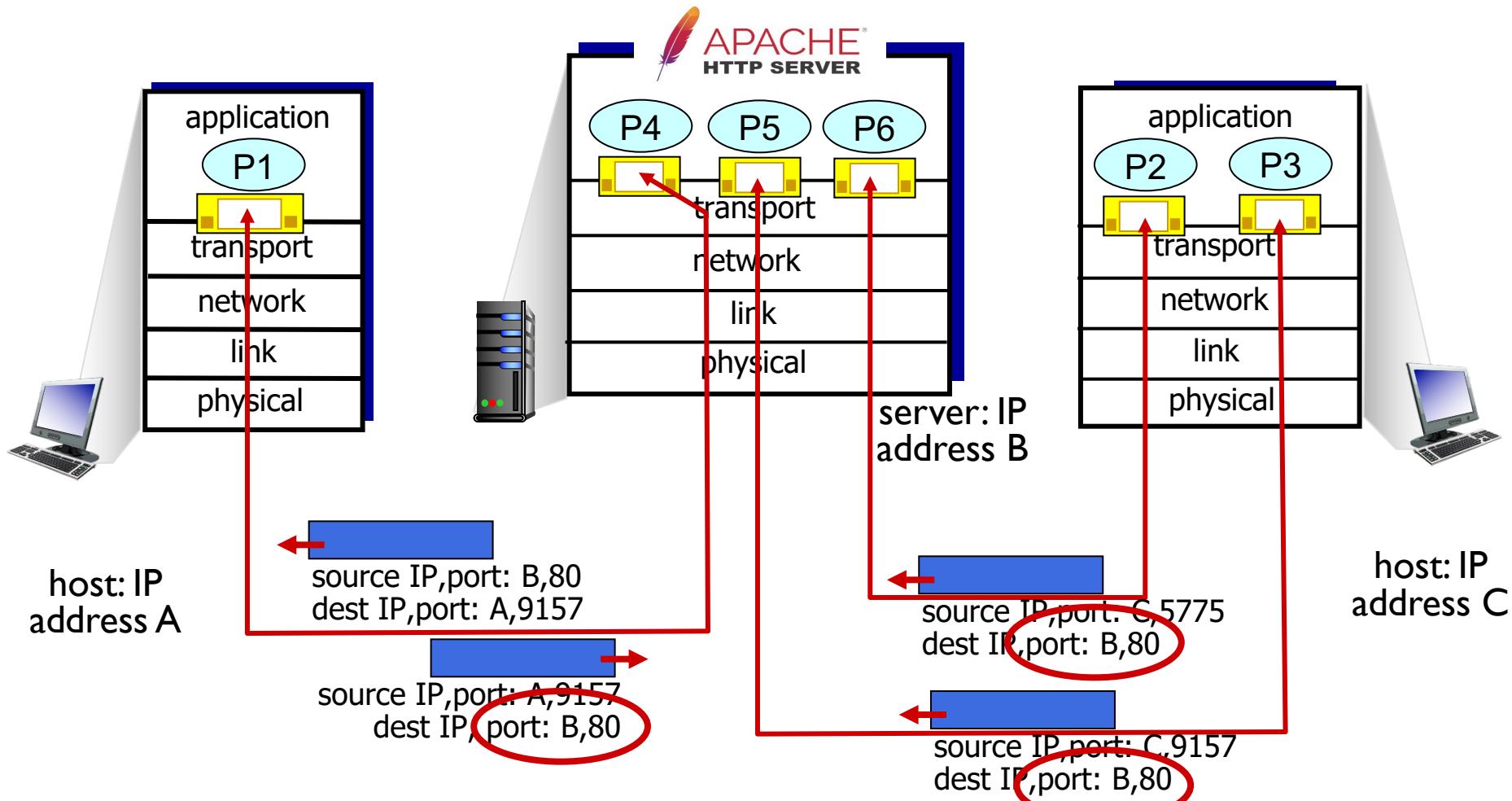
```
DatagramSocket mySocket1 =  
new DatagramSocket (5775);
```



# 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 IP and destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers

# Chapter 3: roadmap

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



# UDP: User Datagram Protocol

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless*:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

## Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

# UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

# UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD  
RFC 768 J. Postel  
ISI  
28 August 1980

## User Datagram Protocol

---

### Introduction

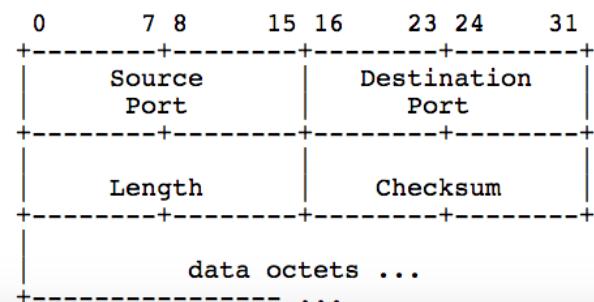
---

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

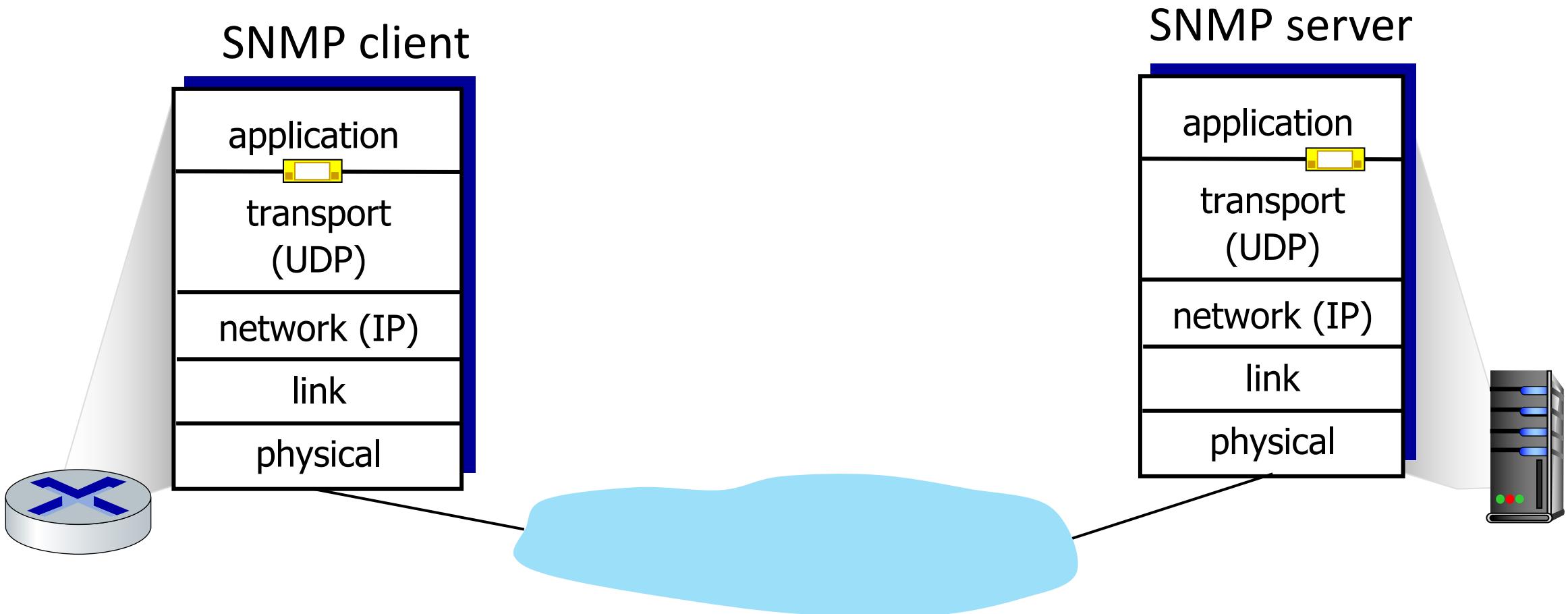
This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

### Format

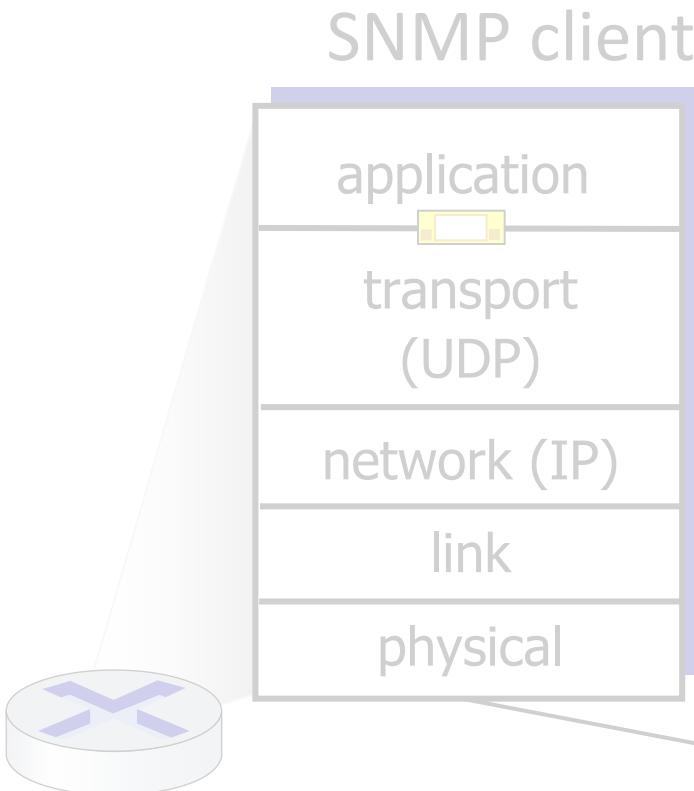
---



# UDP: Transport Layer Actions



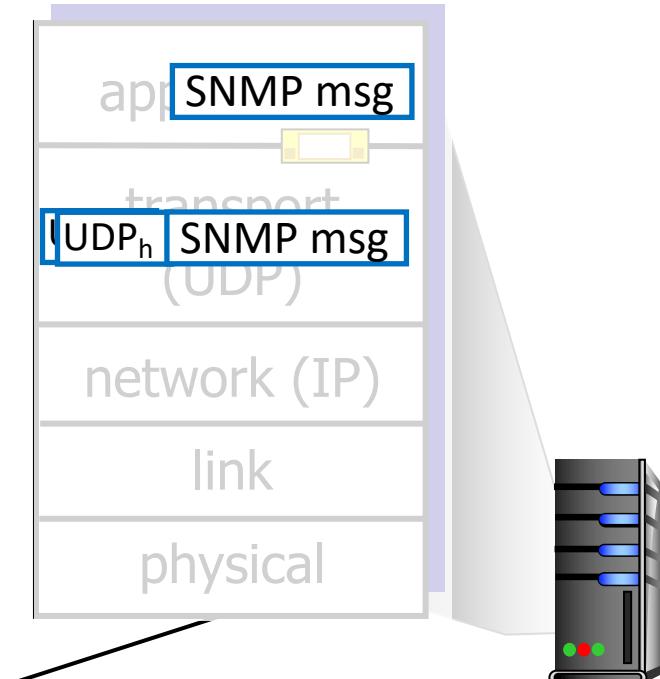
# UDP: Transport Layer Actions



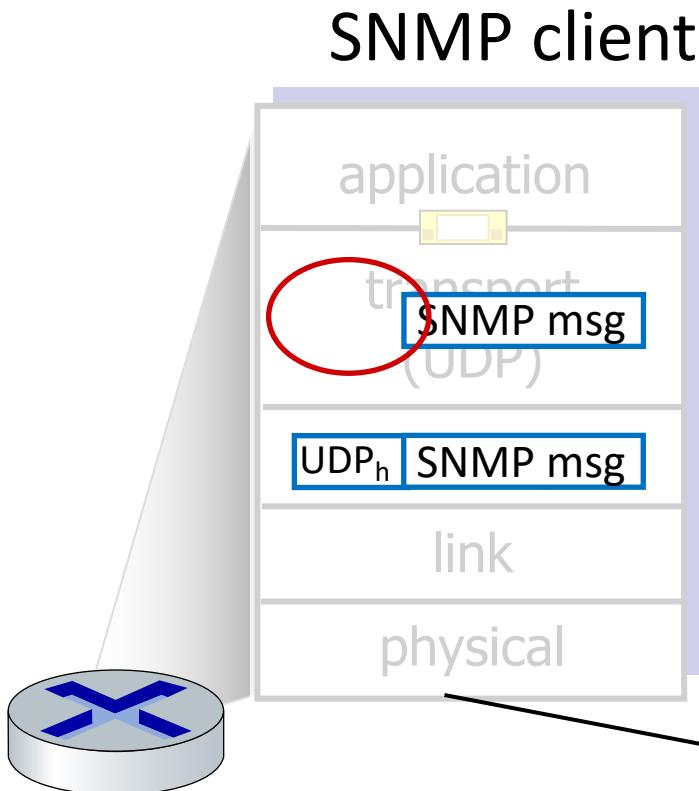
## UDP sender actions:

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

## SNMP server



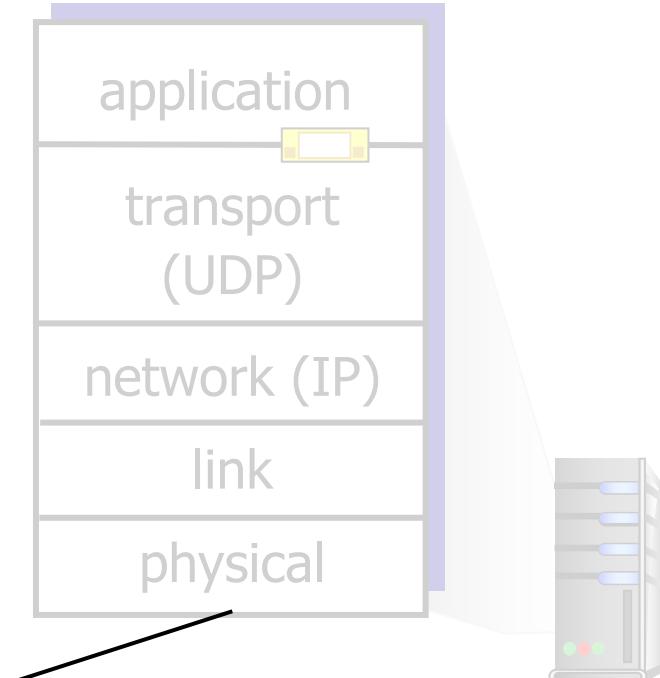
# UDP: Transport Layer Actions



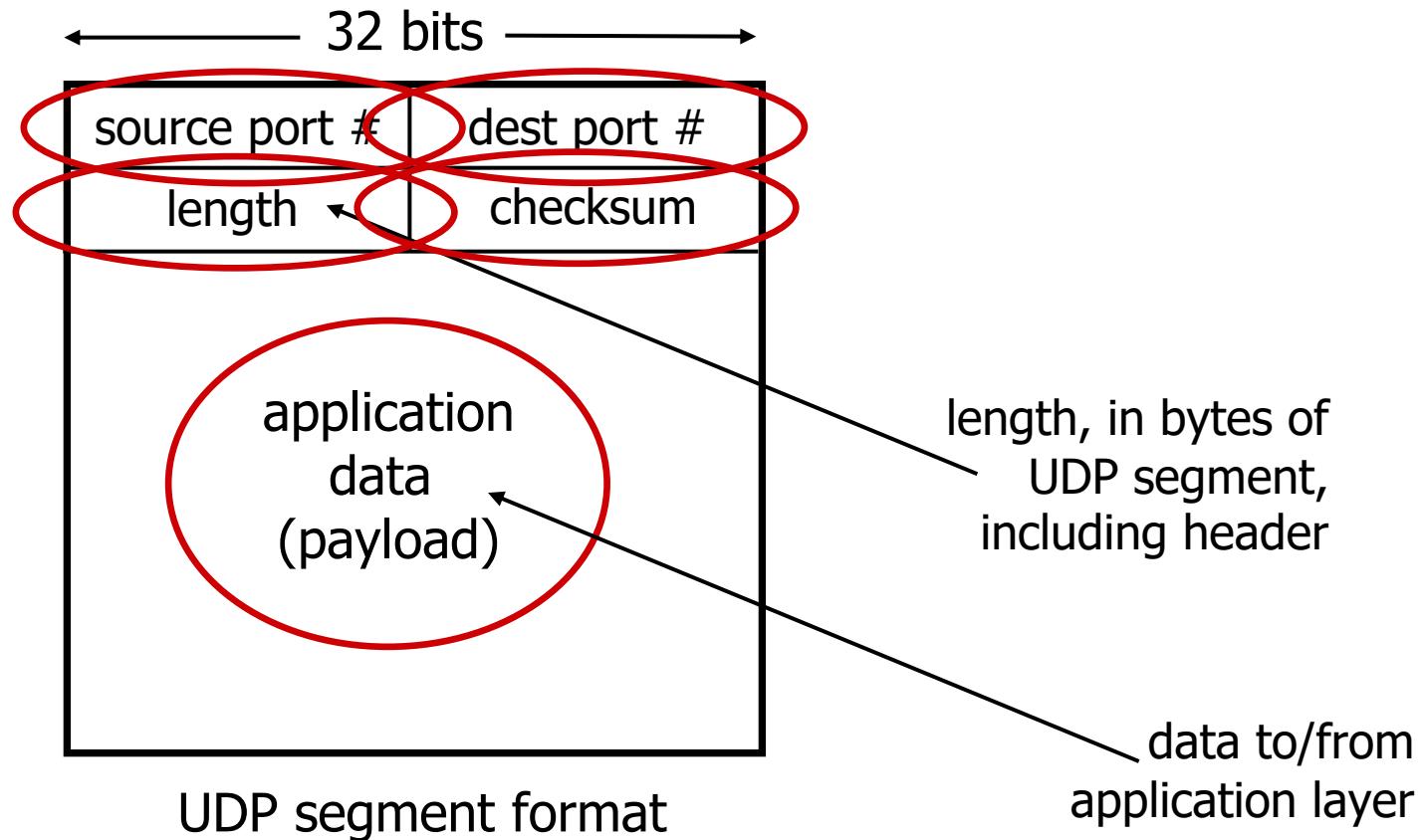
## UDP receiver actions:

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

## SNMP server

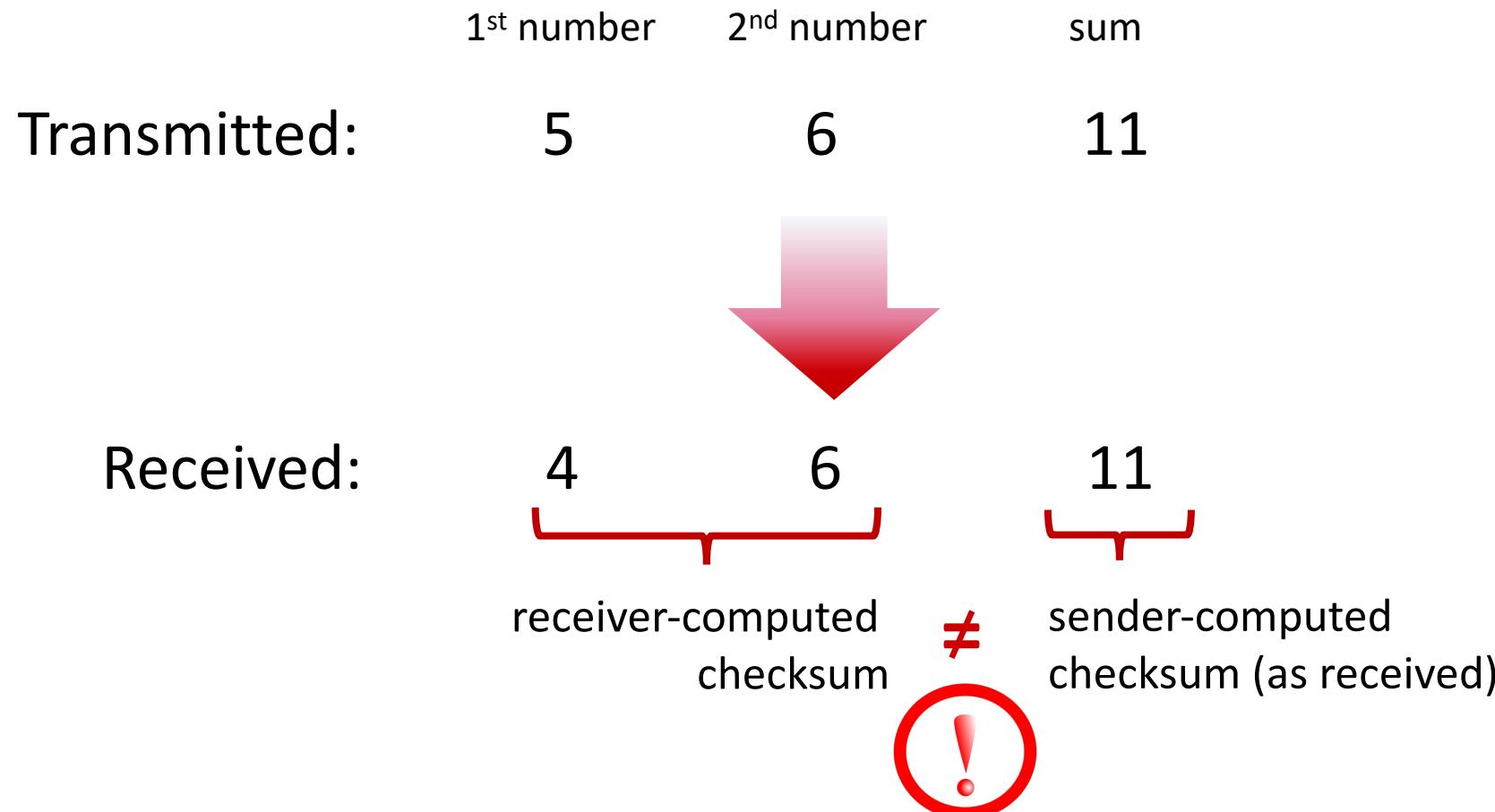


# UDP segment header



# UDP checksum

**Goal:** detect errors (*i.e.*, flipped bits) in transmitted segment



# 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
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
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

# Internet checksum: weak protection!

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
<hr/>																	
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	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0

Even though numbers have changed (bit flips), *no* change in checksum!

# Summary: UDP

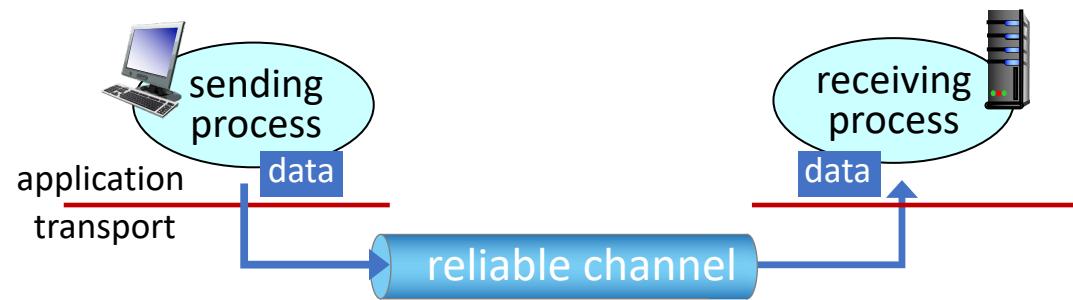
- “no frills” protocol:
  - segments may be lost, delivered out of order
  - best effort service: “send and hope for the best”
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional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

# Chapter 3: roadmap

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

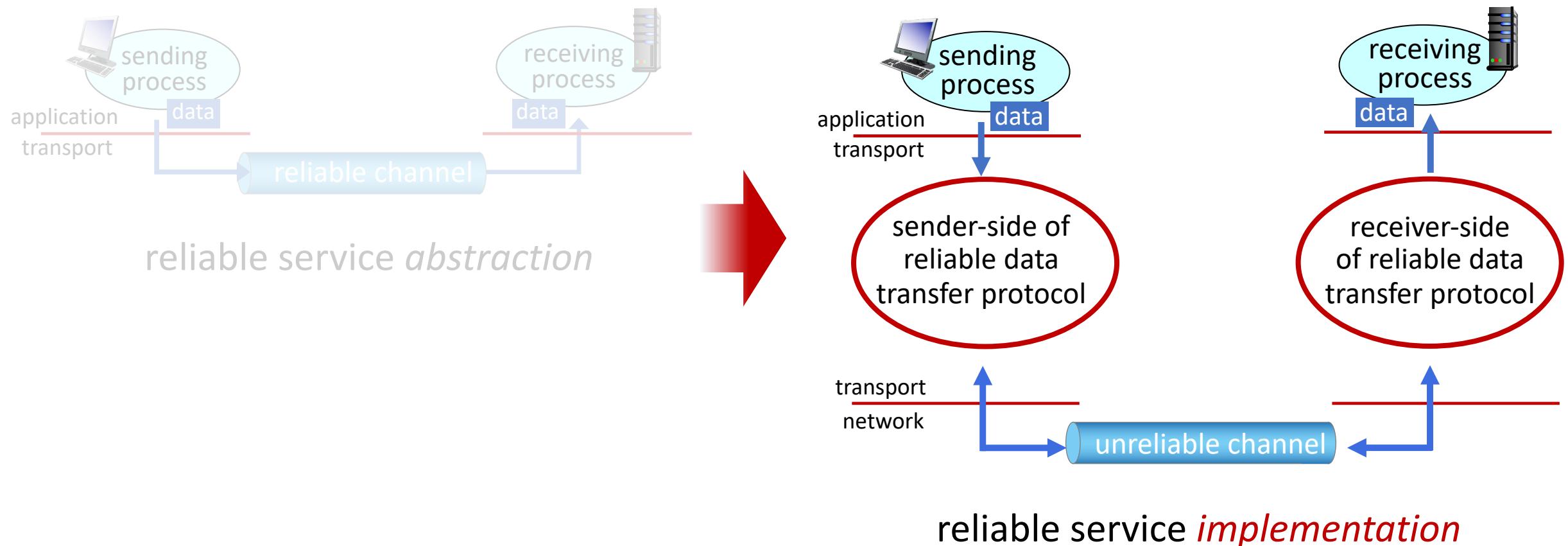


# Principles of reliable data transfer



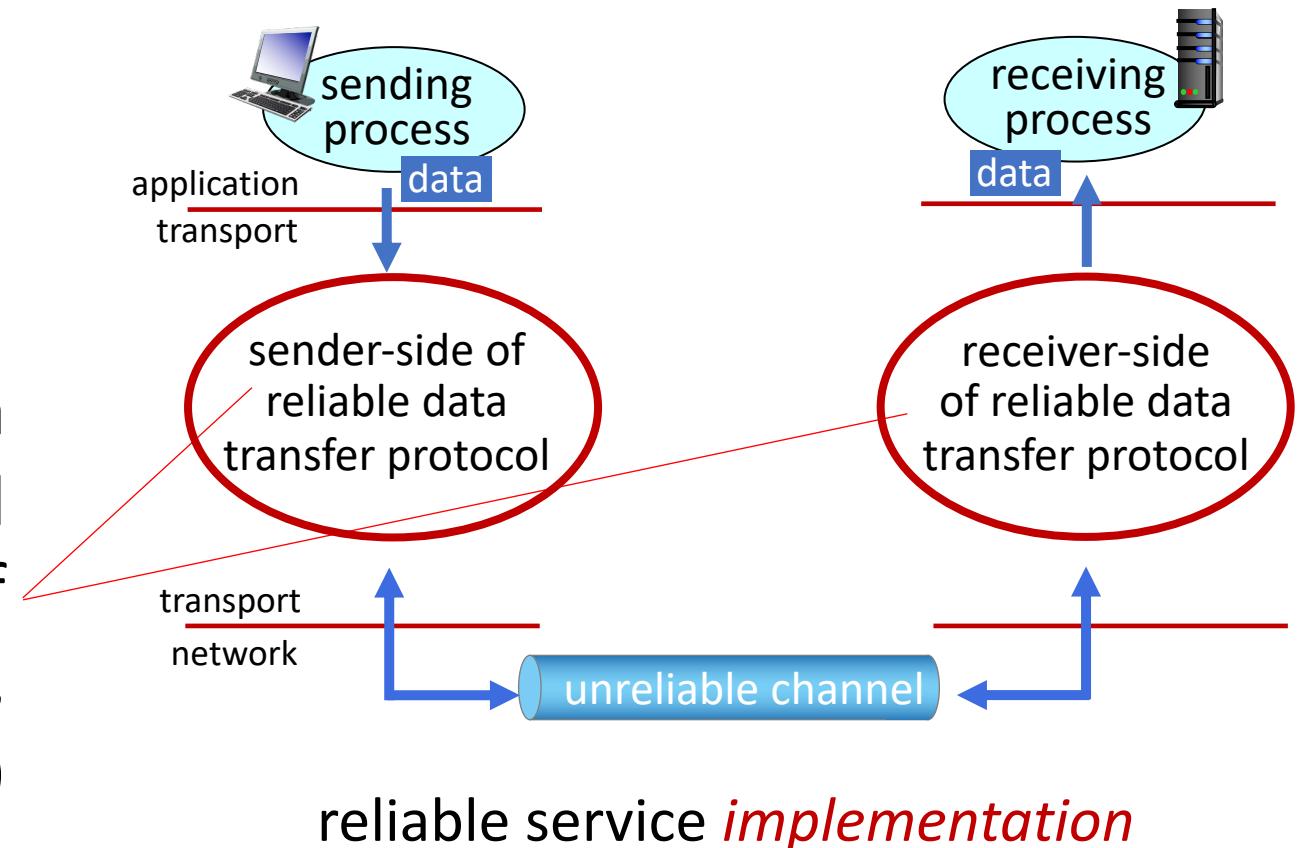
reliable service *abstraction*

# Principles of reliable data transfer



# Principles of reliable data transfer

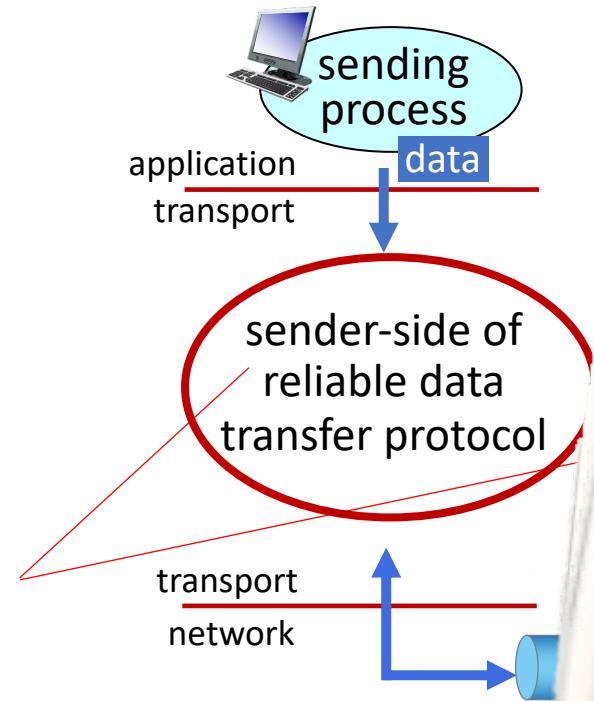
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



# Principles of reliable data transfer

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

- unless communicated via a message

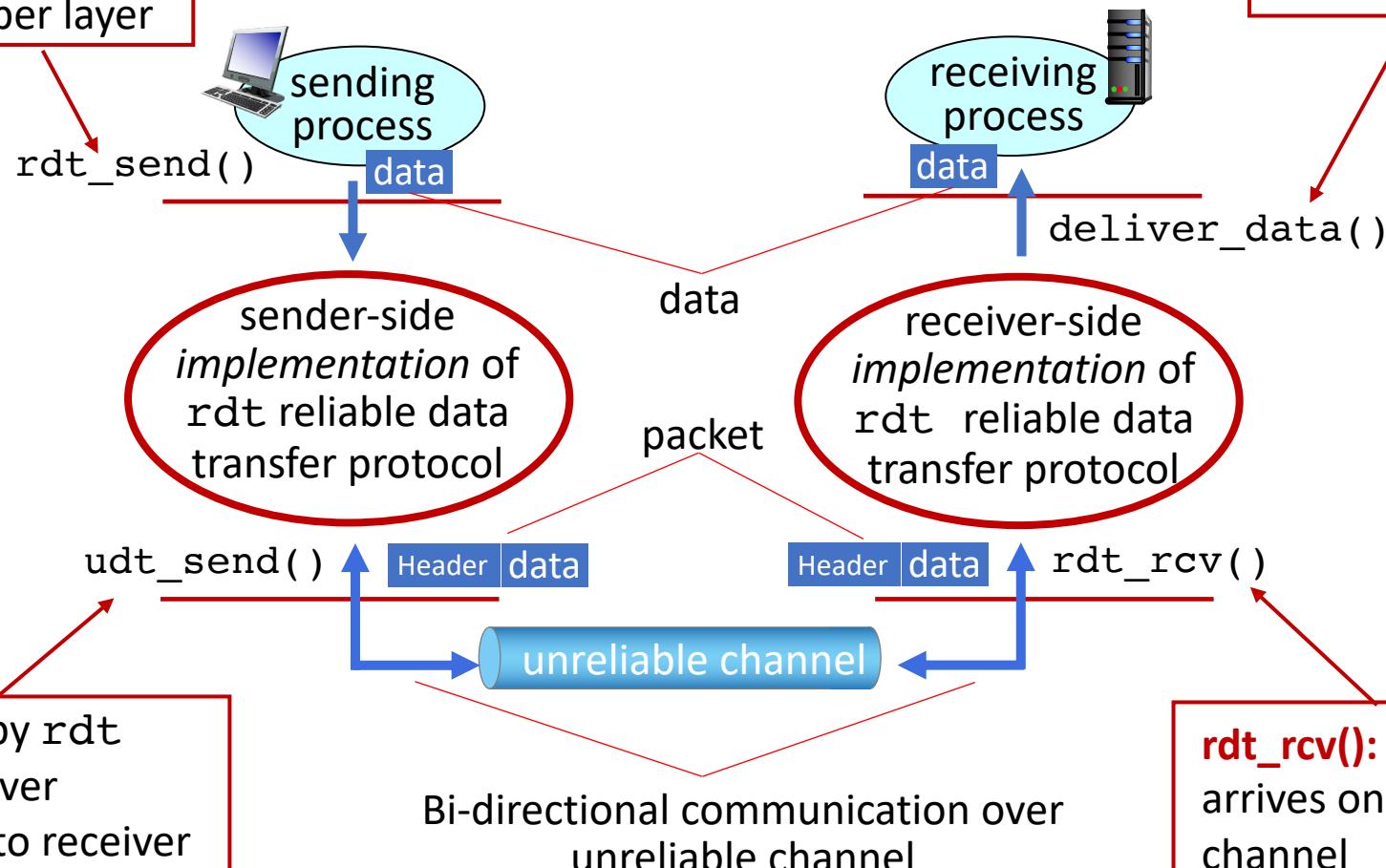


reliable service *implementation*



# Reliable data transfer protocol (rdt): interfaces

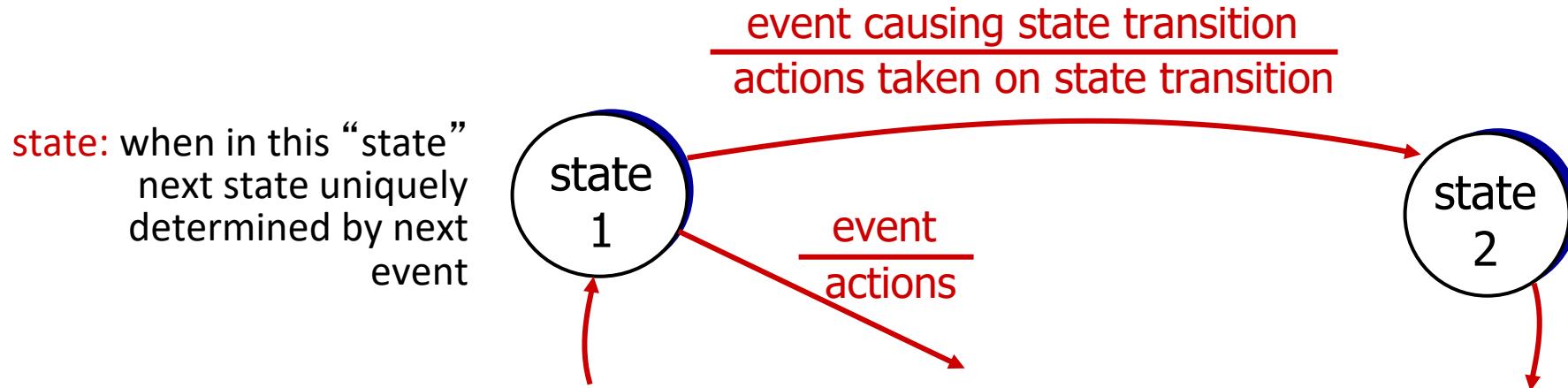
**rdt\_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



# Reliable data transfer: getting started

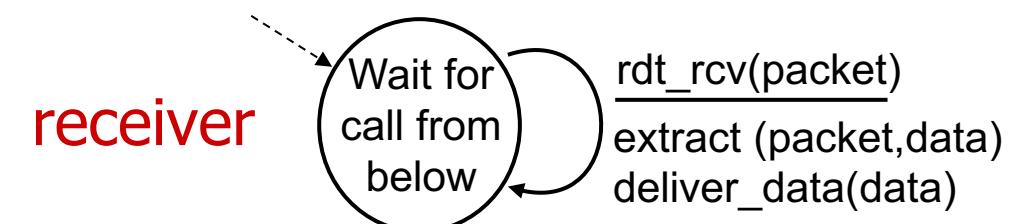
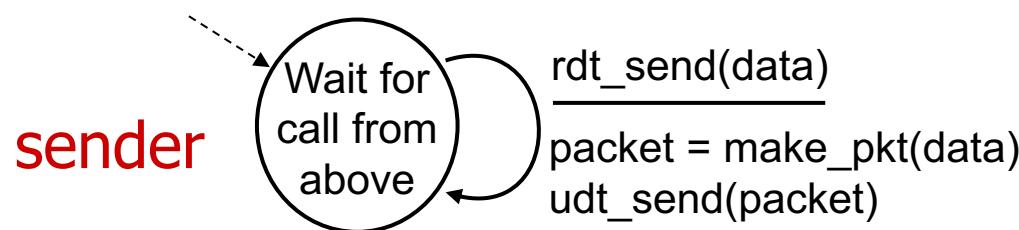
We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



# rdt1.0: reliable transfer over a reliable channel

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



# rdt2.0: channel with bit errors

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

*How do humans recover from “errors” during conversation?*

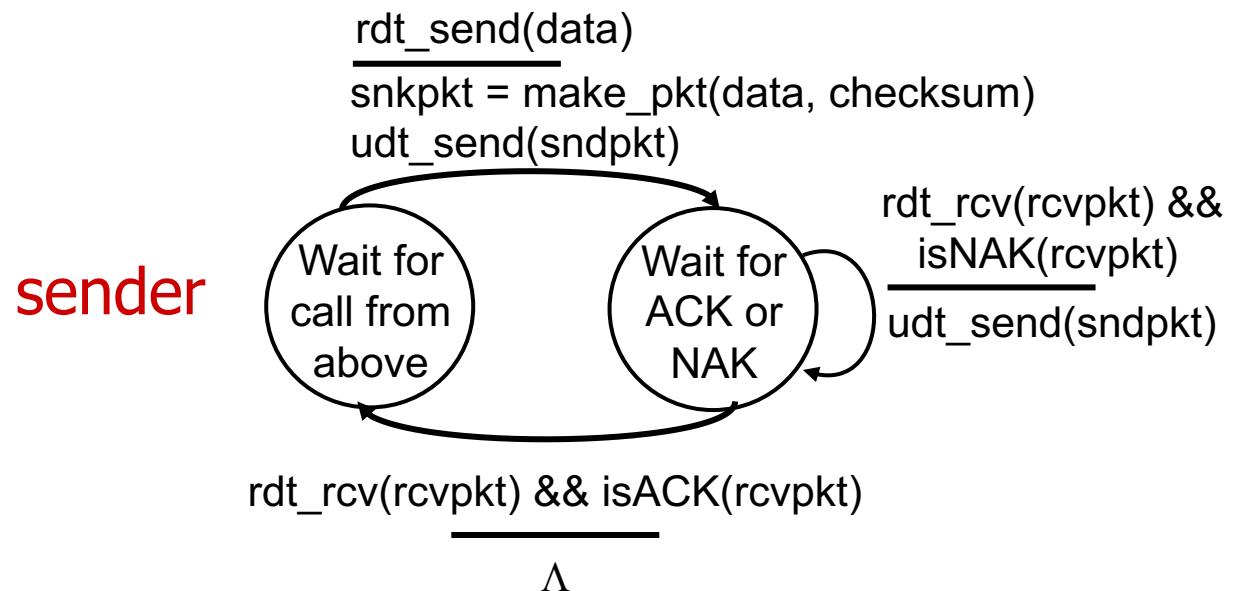
# rdt2.0: channel with bit errors

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

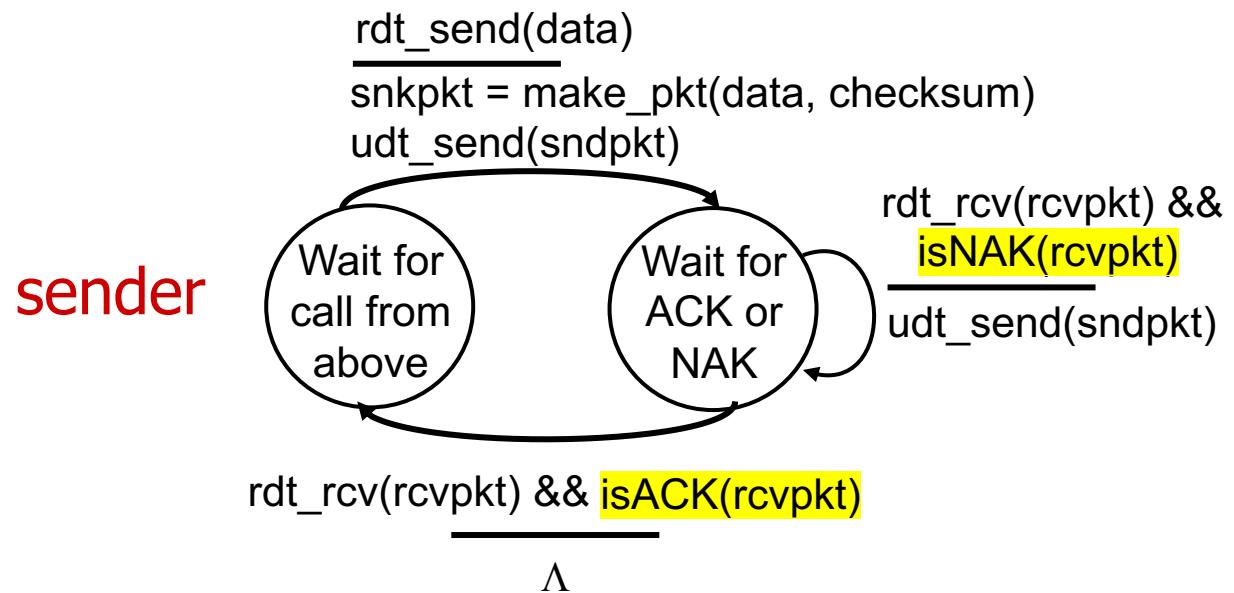
stop and wait

sender sends one packet, then waits for receiver response

# rdt2.0: FSM specifications



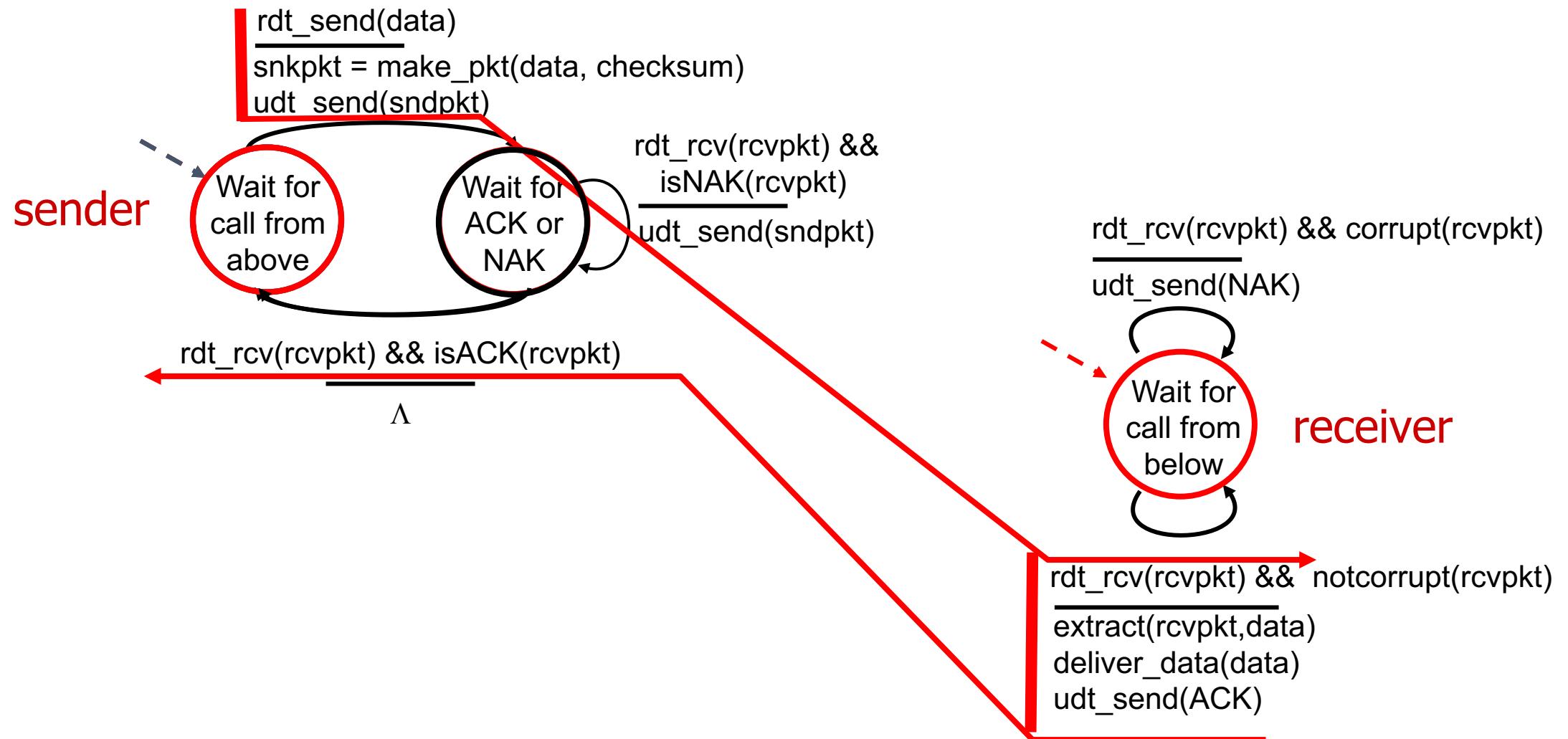
# rdt2.0: FSM specification



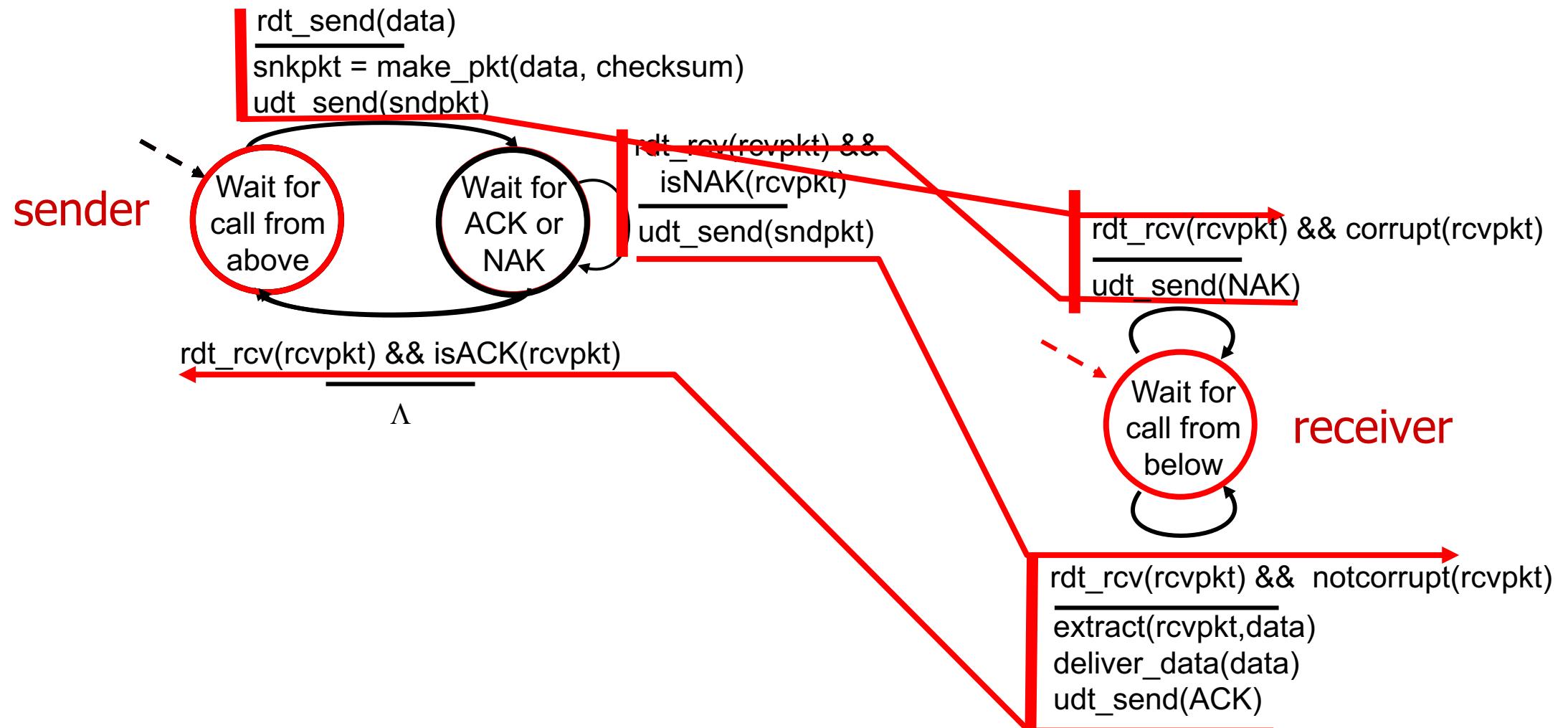
**Note:** “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender

- that’s why we need a protocol!

# rdt2.0: operation with no errors



# rdt2.0: corrupted packet scenario



# rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

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

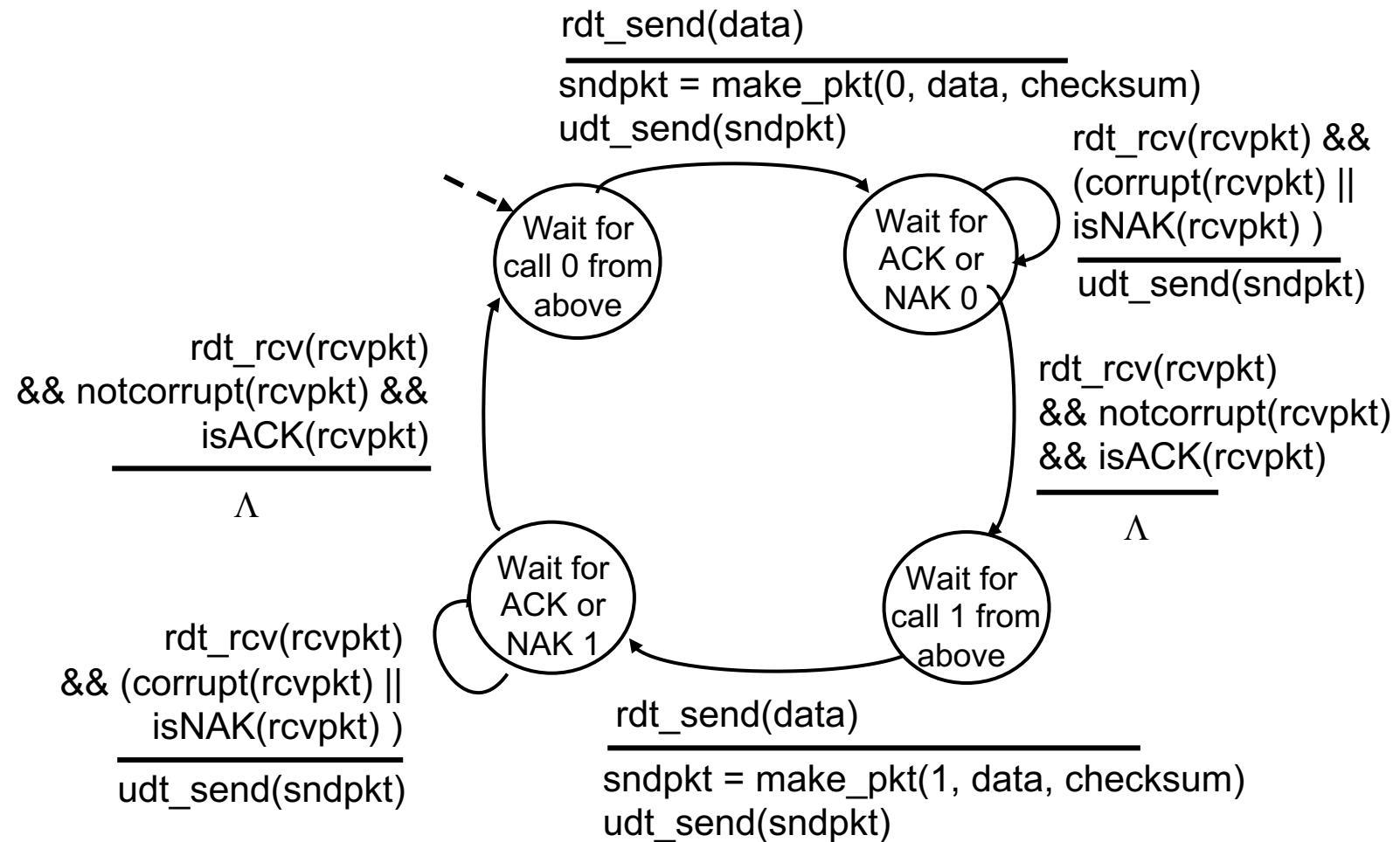
handling duplicates:

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

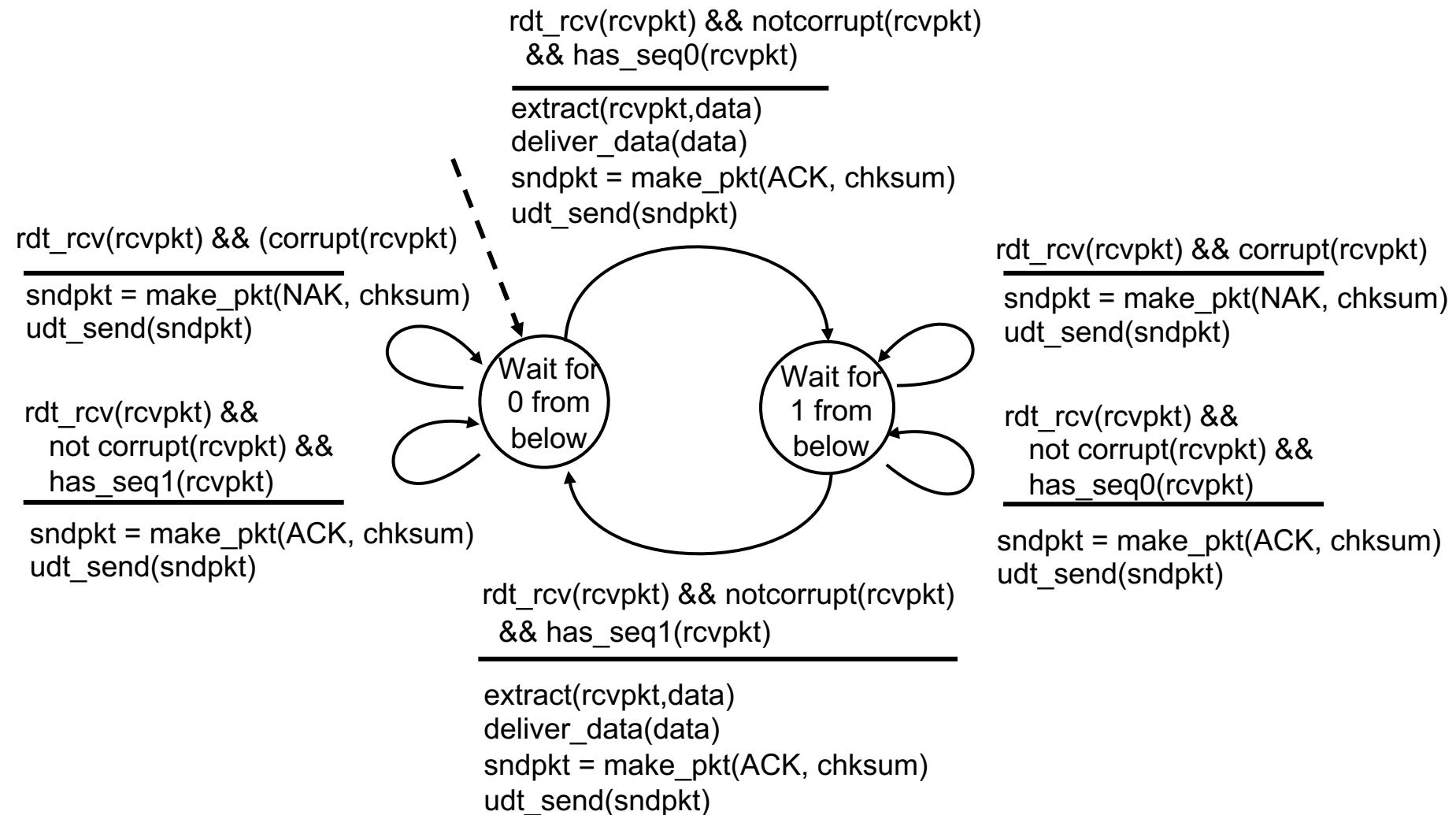
stop and wait

sender sends one packet, then waits for receiver response

# rdt2.1: sender, handling garbled ACK/NAKs



# rdt2.1: receiver, handling garbled ACK/NAKs



# rdt2.1: discussion

## sender:

- seq # added to pkt
- two seq. #s (0,1) will suffice.  
Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

## receiver:

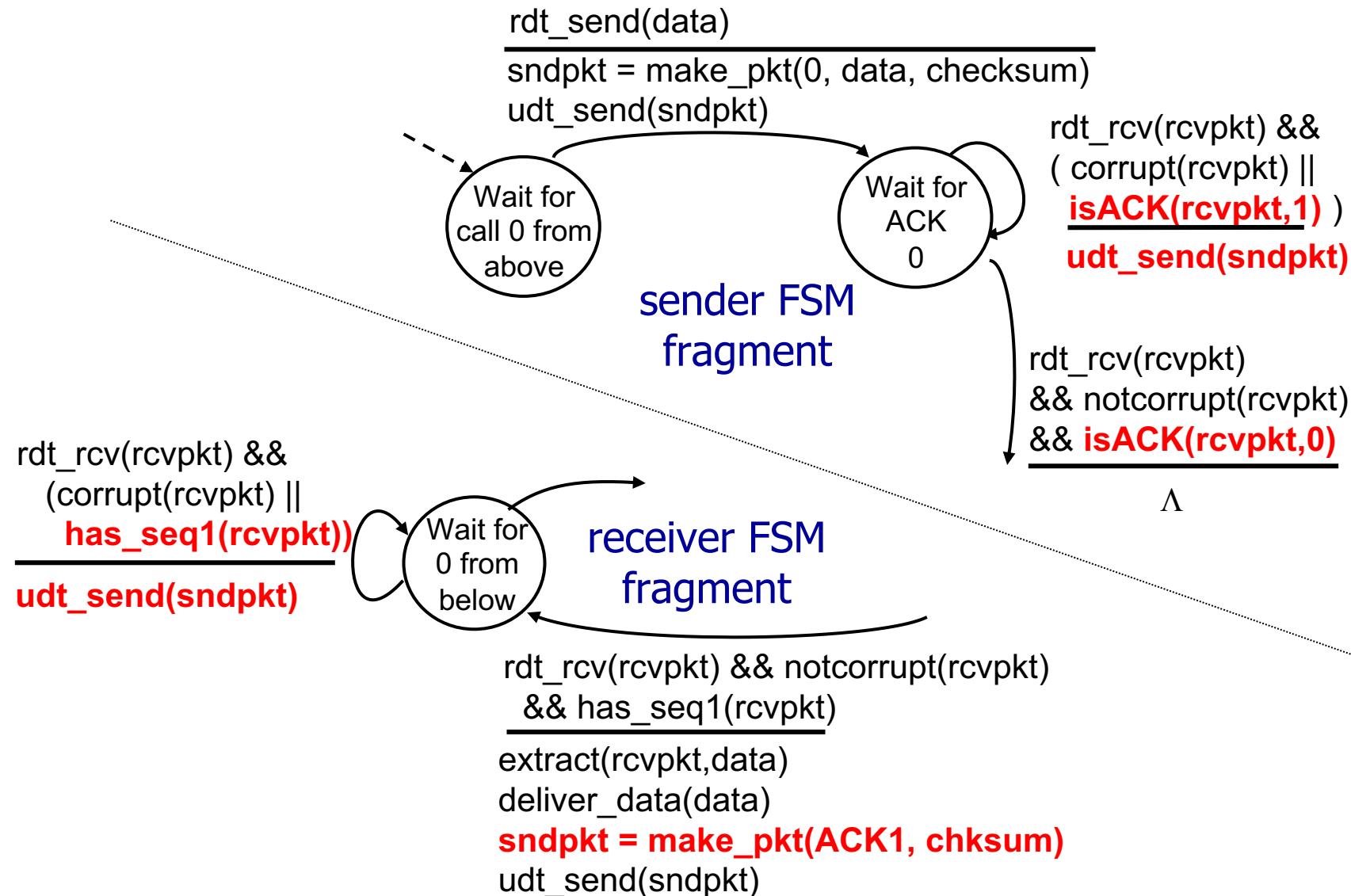
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

# rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK:  
*retransmit current pkt*

As we will see, TCP uses this approach to be NAK-free

# rdt2.2: sender, receiver fragments



# rdt3.0: channels with errors *and* loss

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

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

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

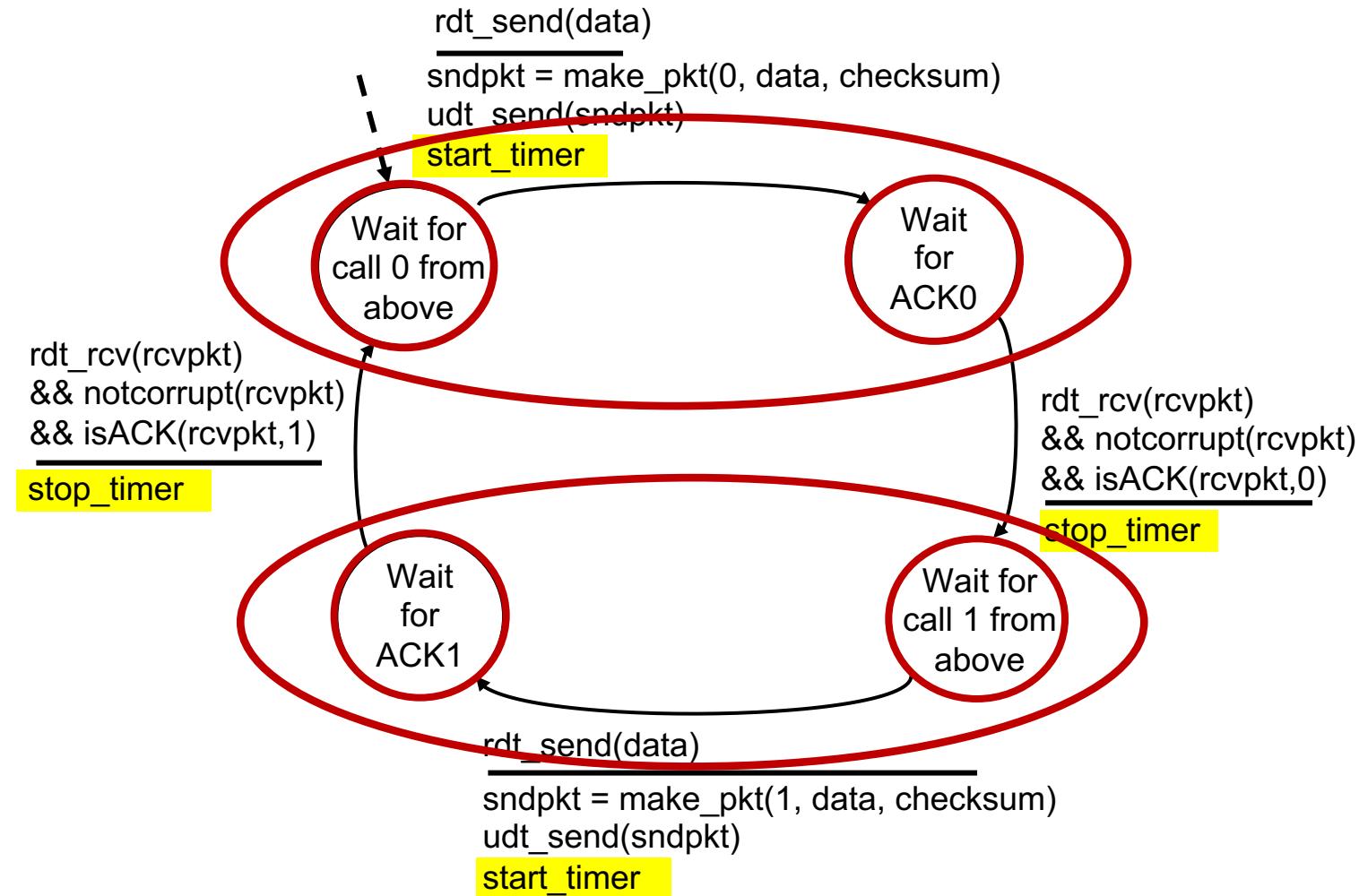
# rdt3.0: channels with errors *and* loss

**Approach:** sender waits “reasonable” amount of time for ACK

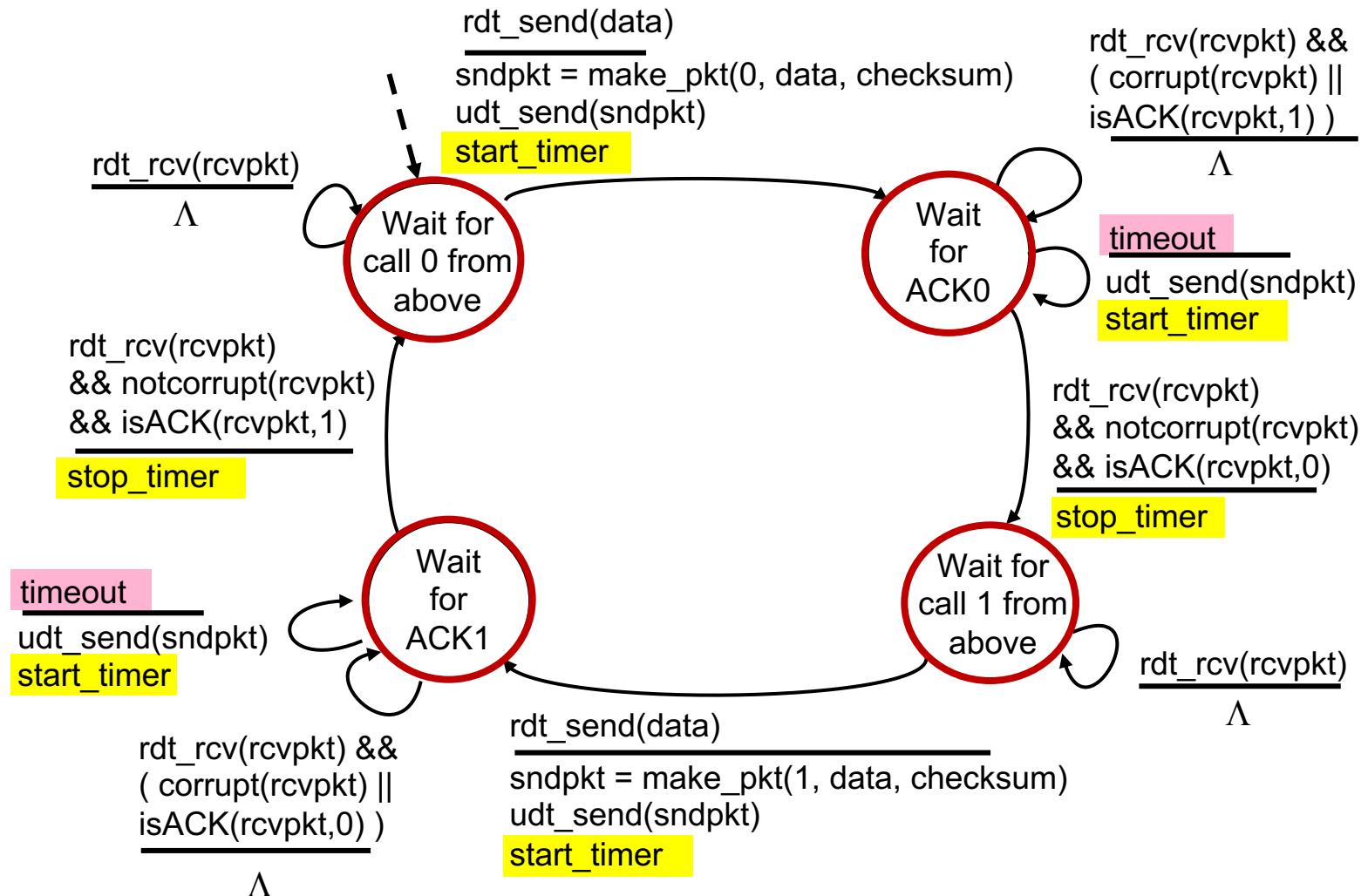
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq #s already handles this!
  - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time



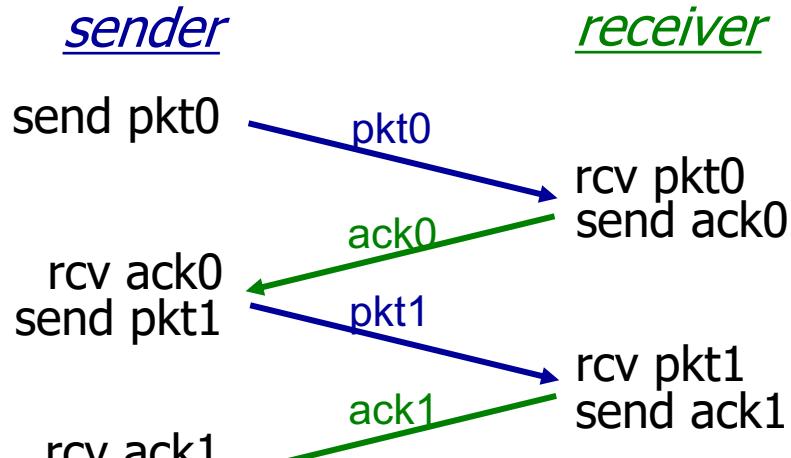
# rdt3.0 sender



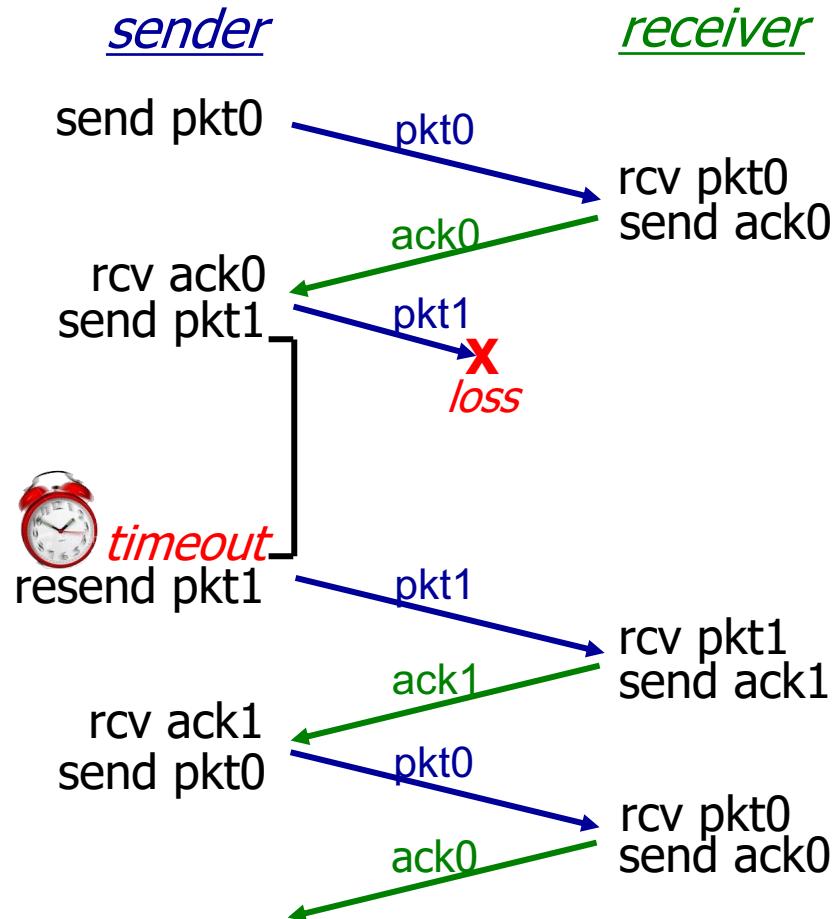
# rdt3.0 sender



# rdt3.0 in action

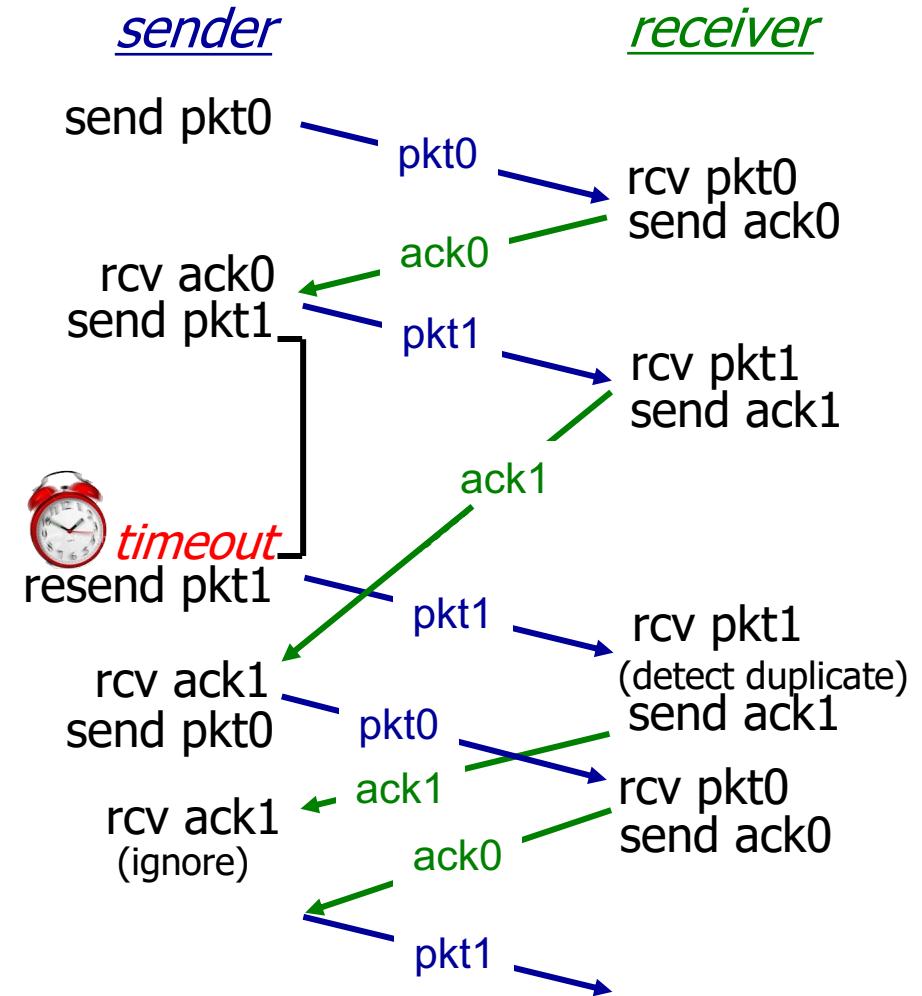
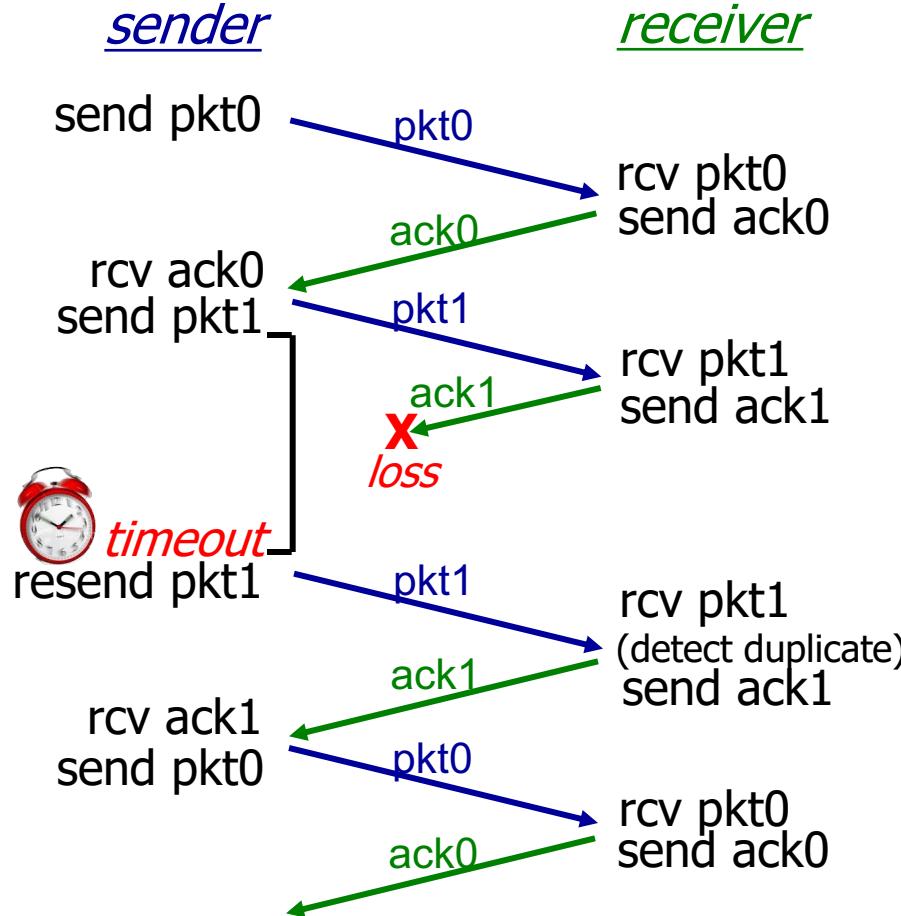


(a) no loss



(b) packet loss

# rdt3.0 in action



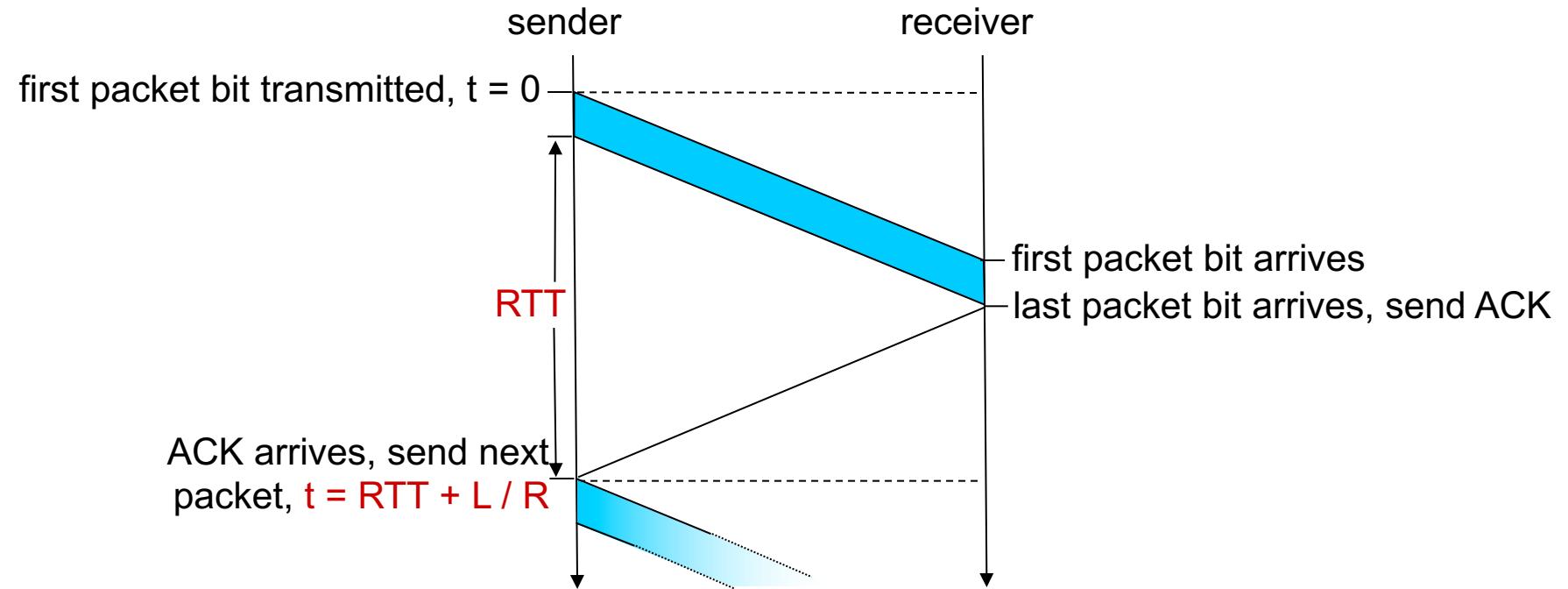
**(d) premature timeout/ delayed ACK**

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

- $U_{sender}$ : *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

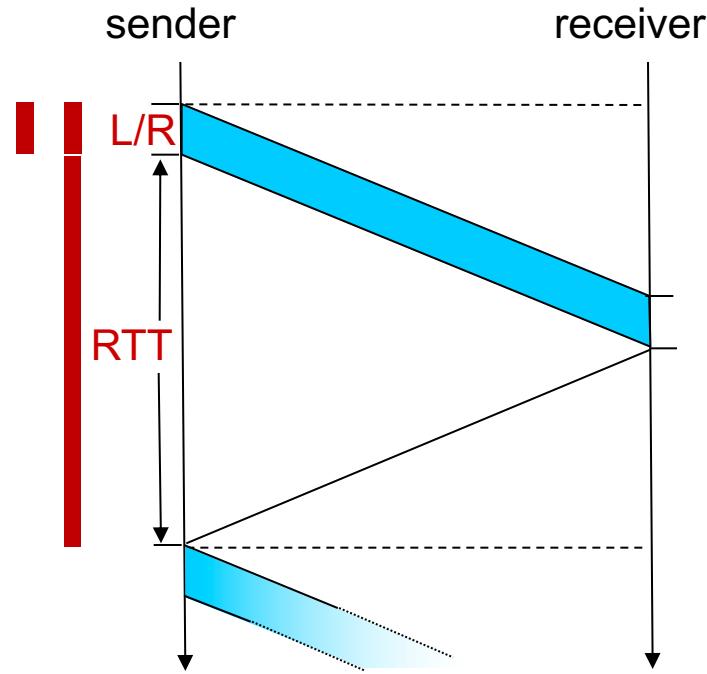
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

# rdt3.0: stop-and-wait operation



# rdt3.0: stop-and-wait operation

$$\begin{aligned} U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\ &= \frac{.008}{30.008} \\ &= 0.00027 \end{aligned}$$

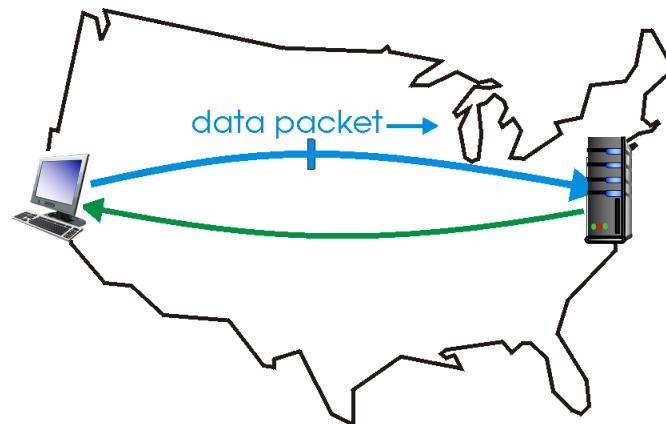


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

# rdt3.0: pipelined protocols operation

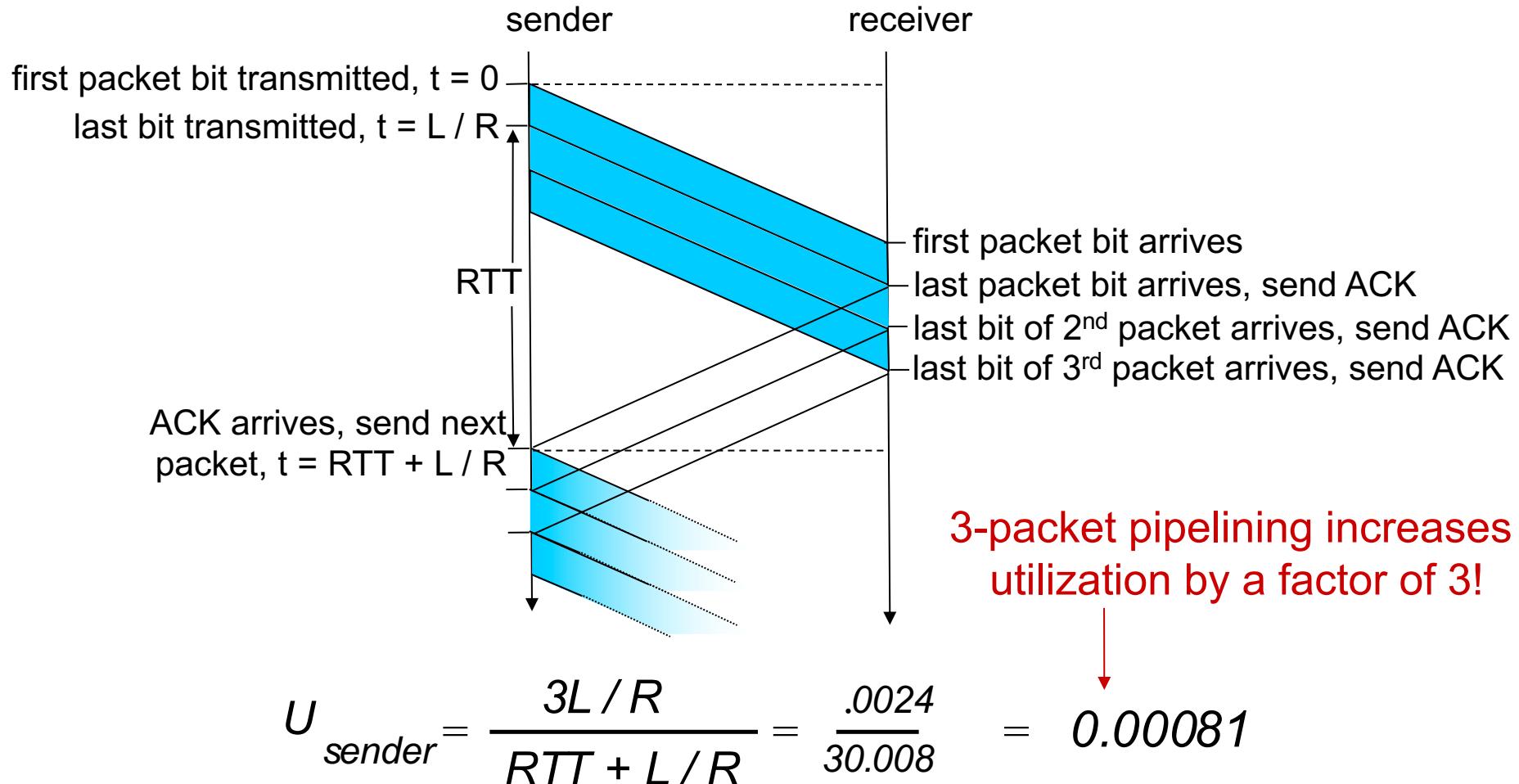
**pipelining:** sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



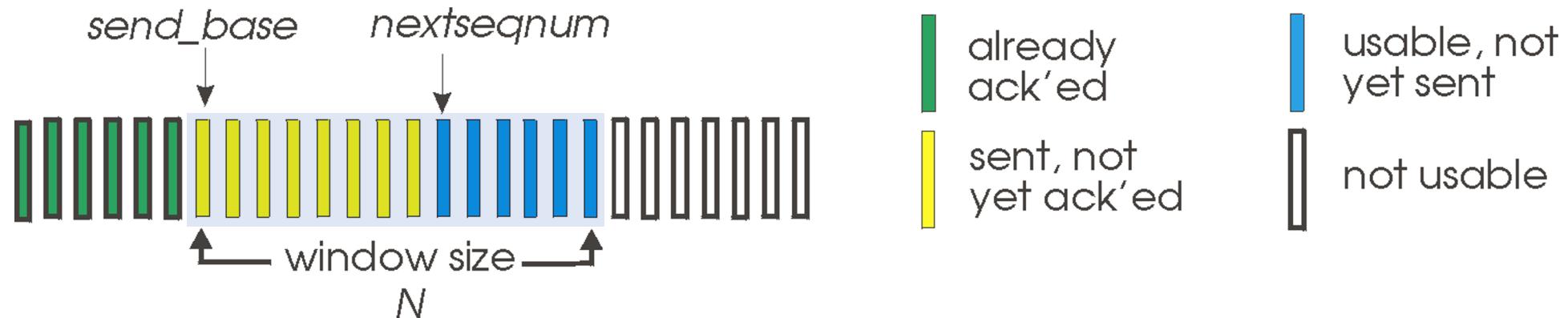
(a) a stop-and-wait protocol in operation

# Pipelining: increased utilization



# Go-Back-N: sender

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

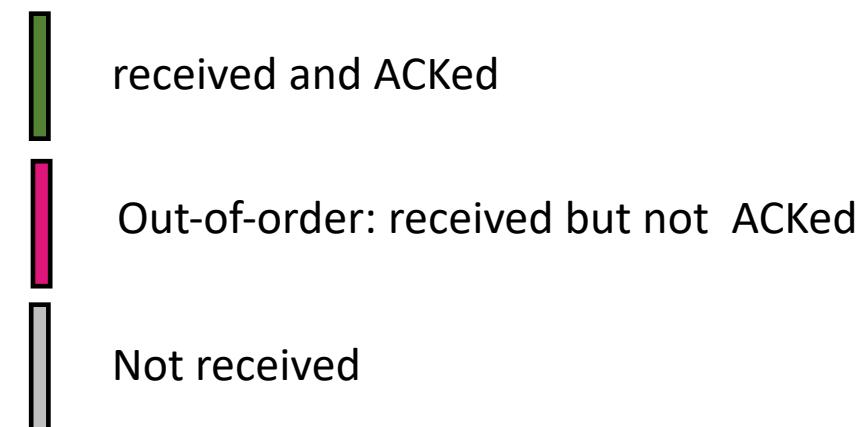
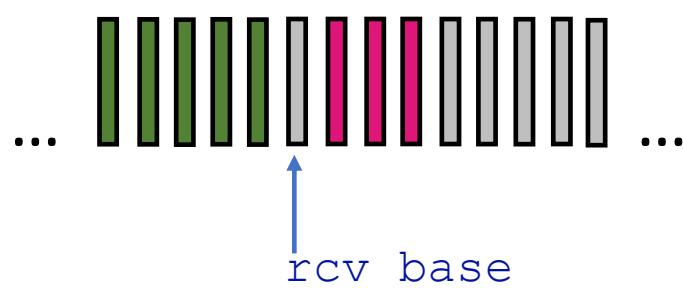


- *cumulative ACK*:  $\text{ACK}(n)$ : ACKs all packets up to, including seq #  $n$ 
  - on receiving  $\text{ACK}(n)$ : move window forward to begin at  $n+1$
- timer for oldest in-flight packet
- $\text{timeout}(n)$ : retransmit packet  $n$  and all higher seq # packets in window

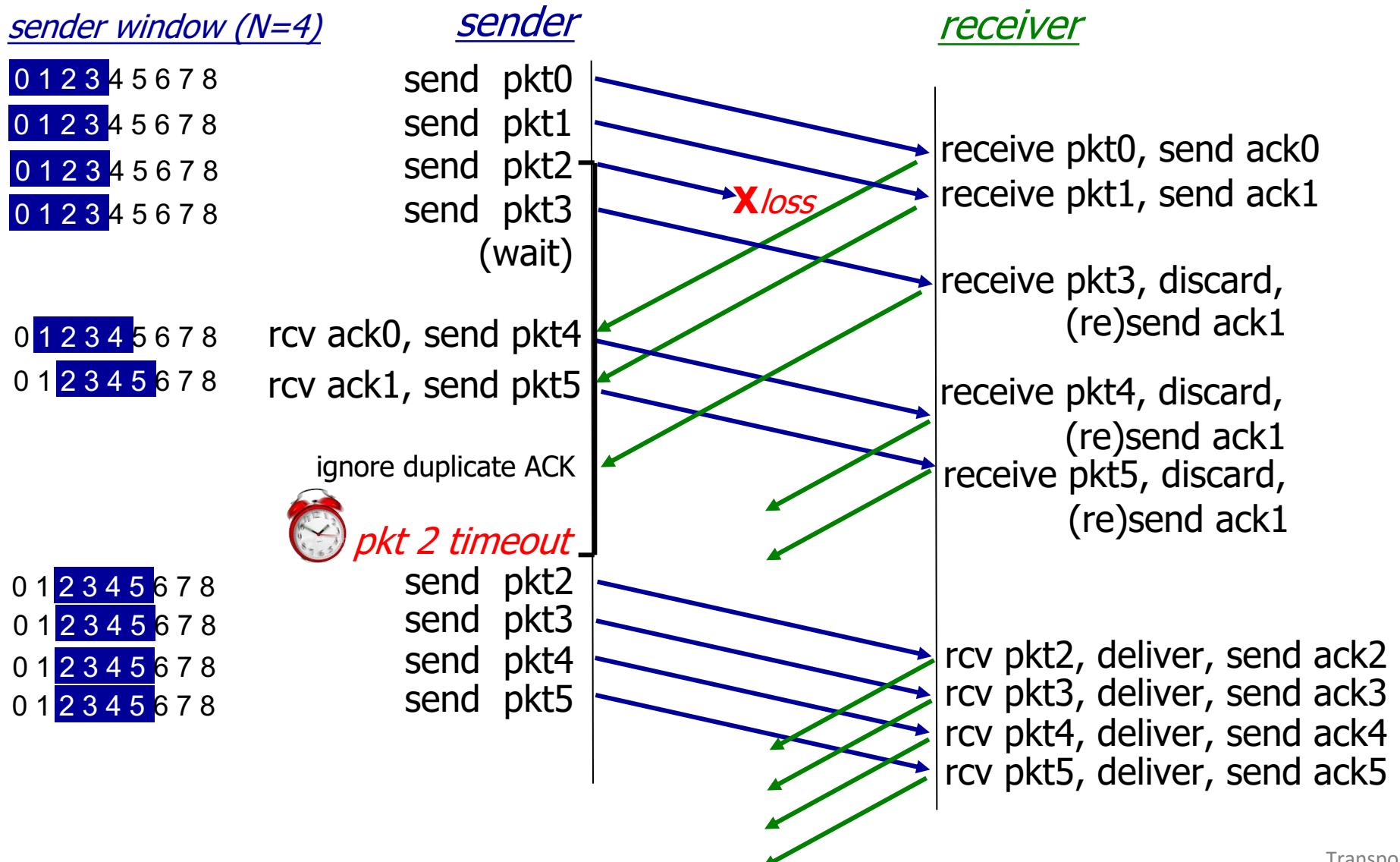
# Go-Back-N: receiver

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

Receiver view of sequence number space:



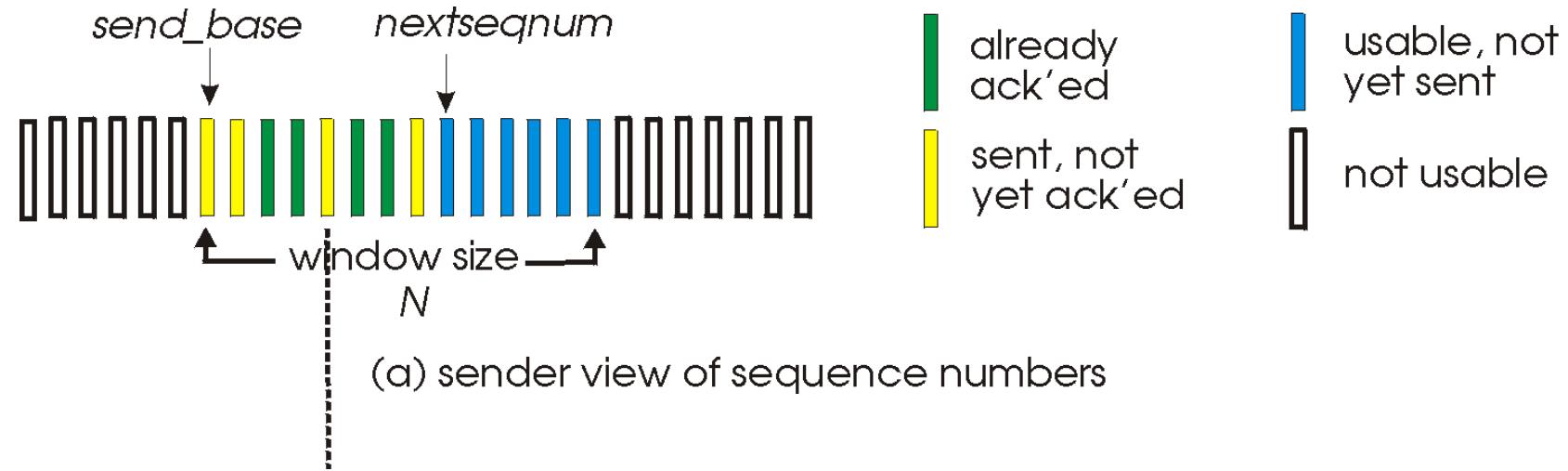
# Go-Back-N in action



# Selective repeat

- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - $N$  consecutive seq #s
  - limits seq #s of sent, unACKed packets

# Selective repeat: sender, receiver windows



# Selective repeat: sender and receiver

## sender

data from above:

- if next available seq # in window, send packet

**timeout( $n$ ):**

- resend packet  $n$ , restart timer

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

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

## receiver

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

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

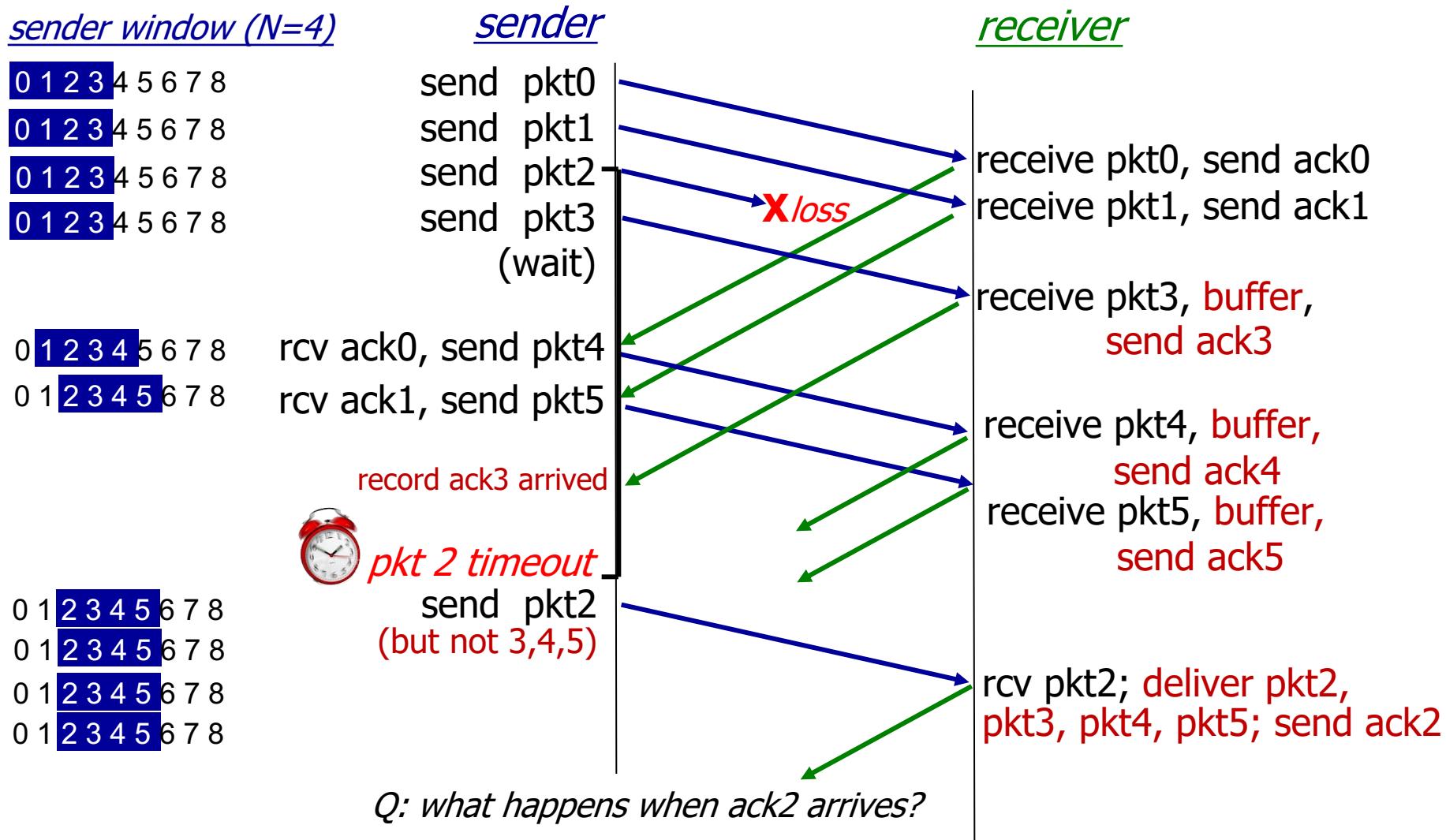
packet  $n$  in [rcvbase-N,rcvbase-1]

- ACK( $n$ )

**otherwise:**

- ignore

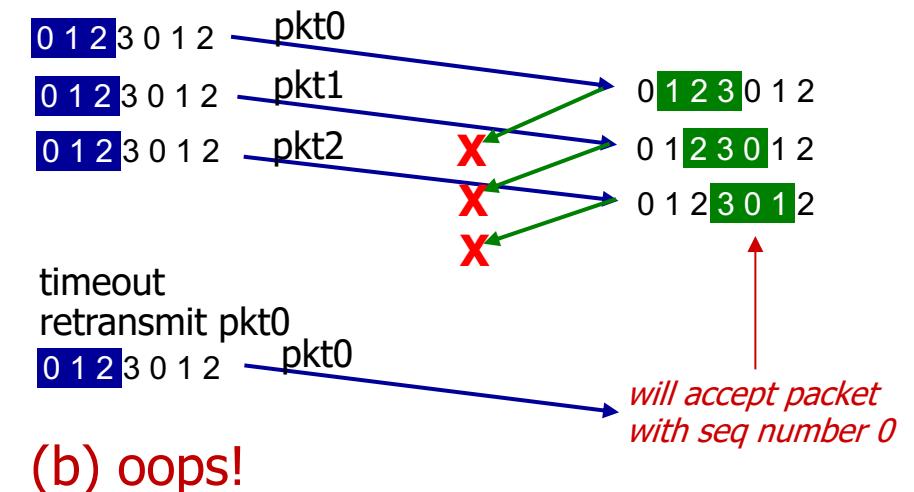
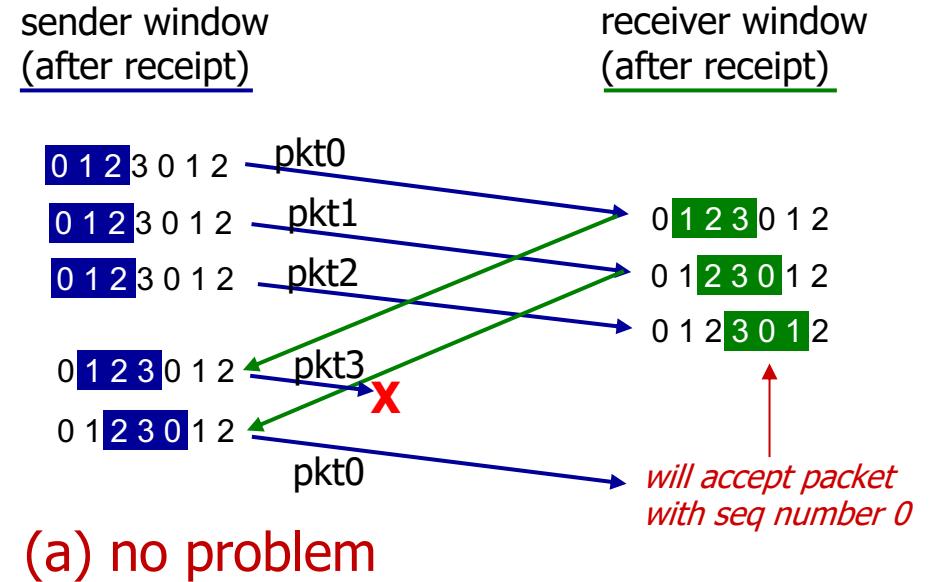
# Selective Repeat in action



# Selective repeat: a dilemma!

example:

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



# Selective repeat: a dilemma!

example:

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

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

sender window  
(after receipt)

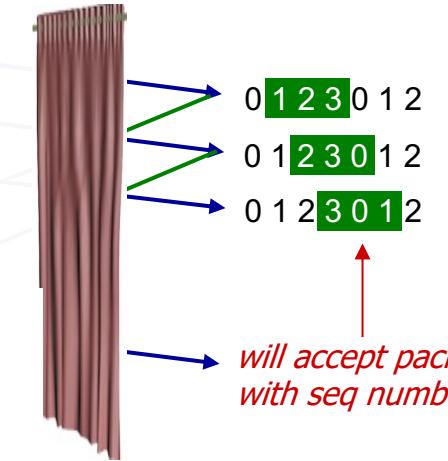
0 1 2 3 0 1 2  
0 1 2 3 0 1 2  
0 1 2 3 0 1 2  
0 1 2 3 0 1 2  
0 1 2 3 0 1 2

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

timeout  
retransmit pkt0  
0 1 2 3 0 1 2

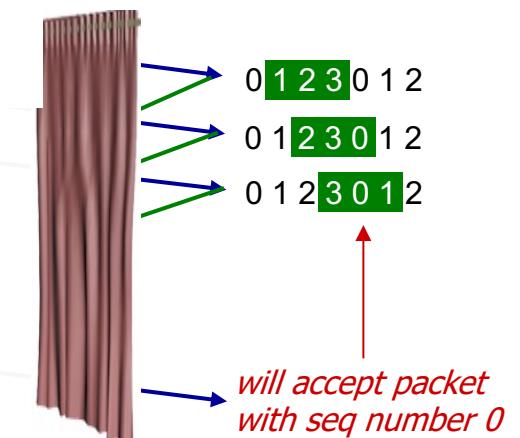
(b) oops!

receiver window  
(after receipt)



0 1 2 3 0 1 2  
0 1 2 3 0 1 2  
0 1 2 3 0 1 2

timeout  
retransmit pkt0  
0 1 2 3 0 1 2



# Chapter 3: roadmap

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



# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

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

# TCP segment structure

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

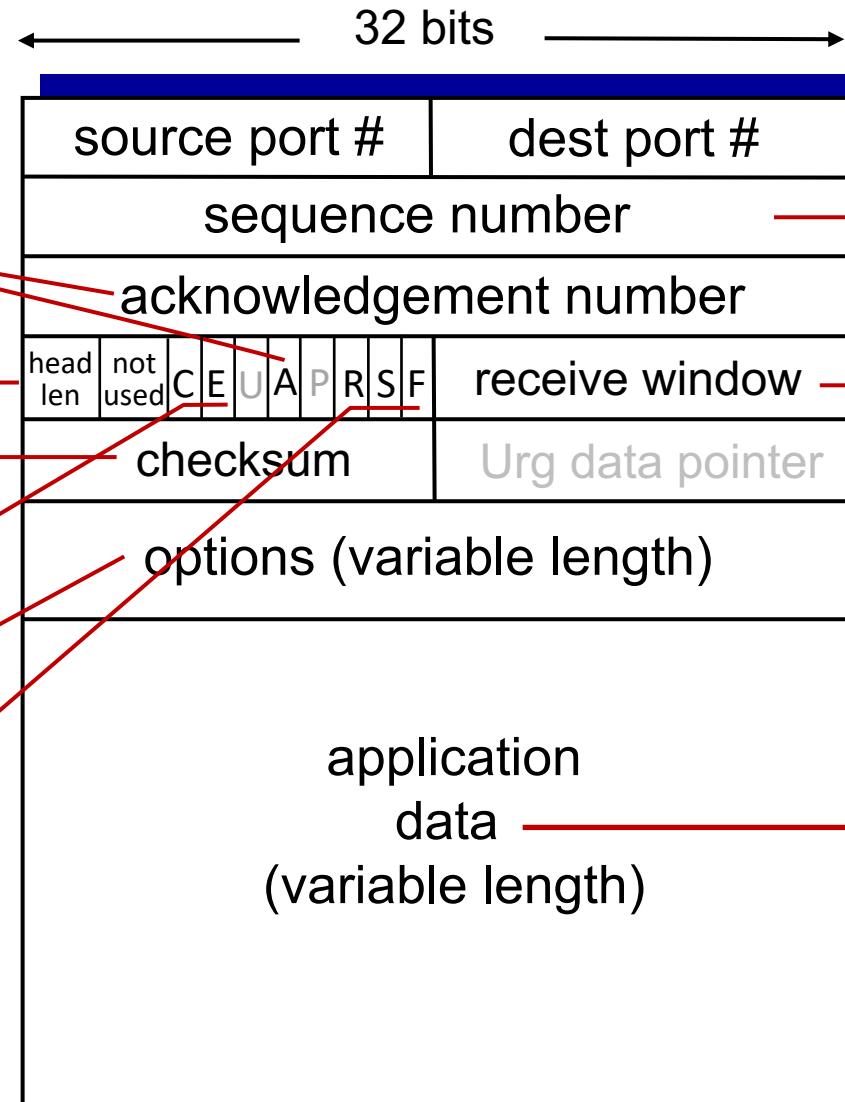
length (of TCP header)

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:*

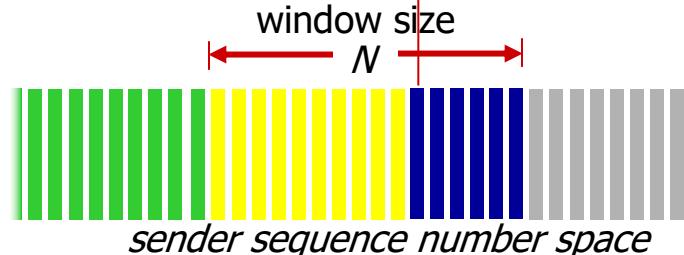
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender

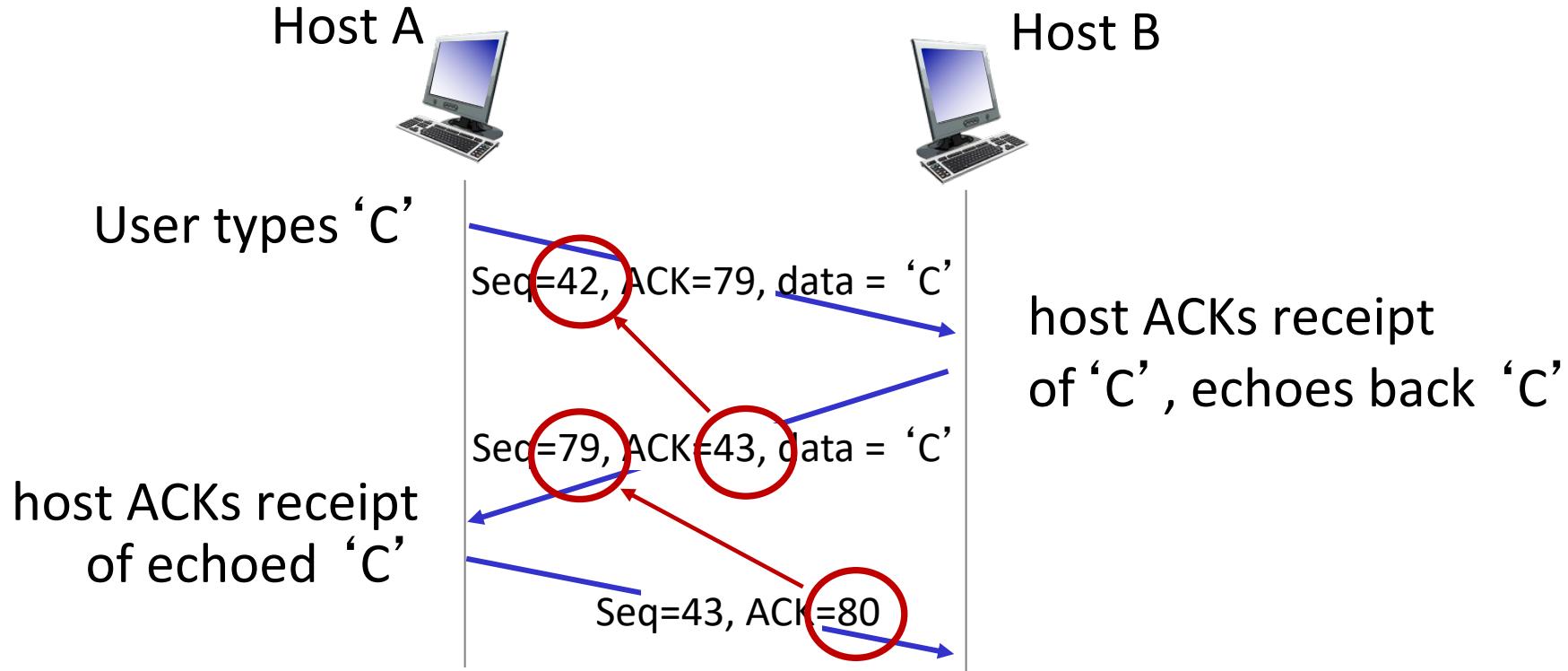
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



outgoing segment from receiver

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

# 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

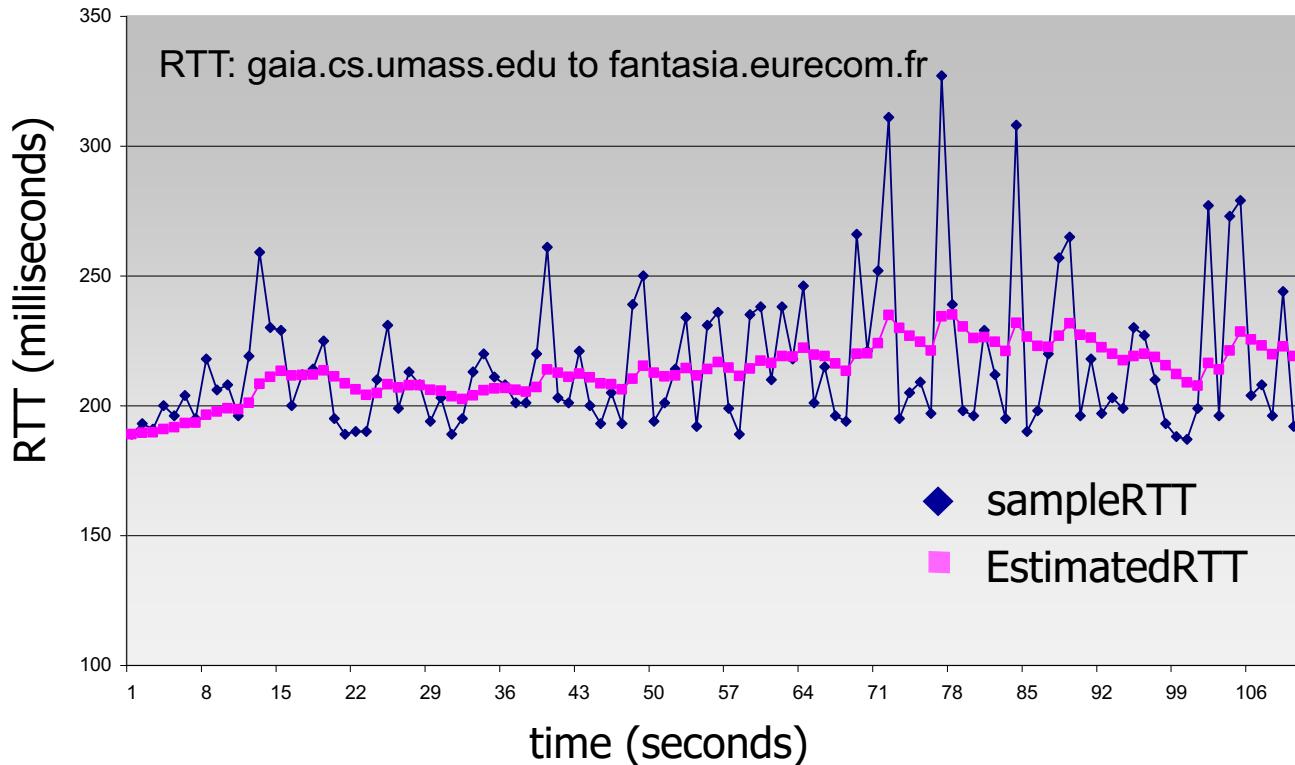
Q: how to estimate RTT?

- *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

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

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



# TCP round trip time, timeout

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

`TimeoutInterval = EstimatedRTT + 4*DevRTT`



estimated RTT

“safety margin”

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

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

(typically,  $\beta = 0.25$ )

# 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**

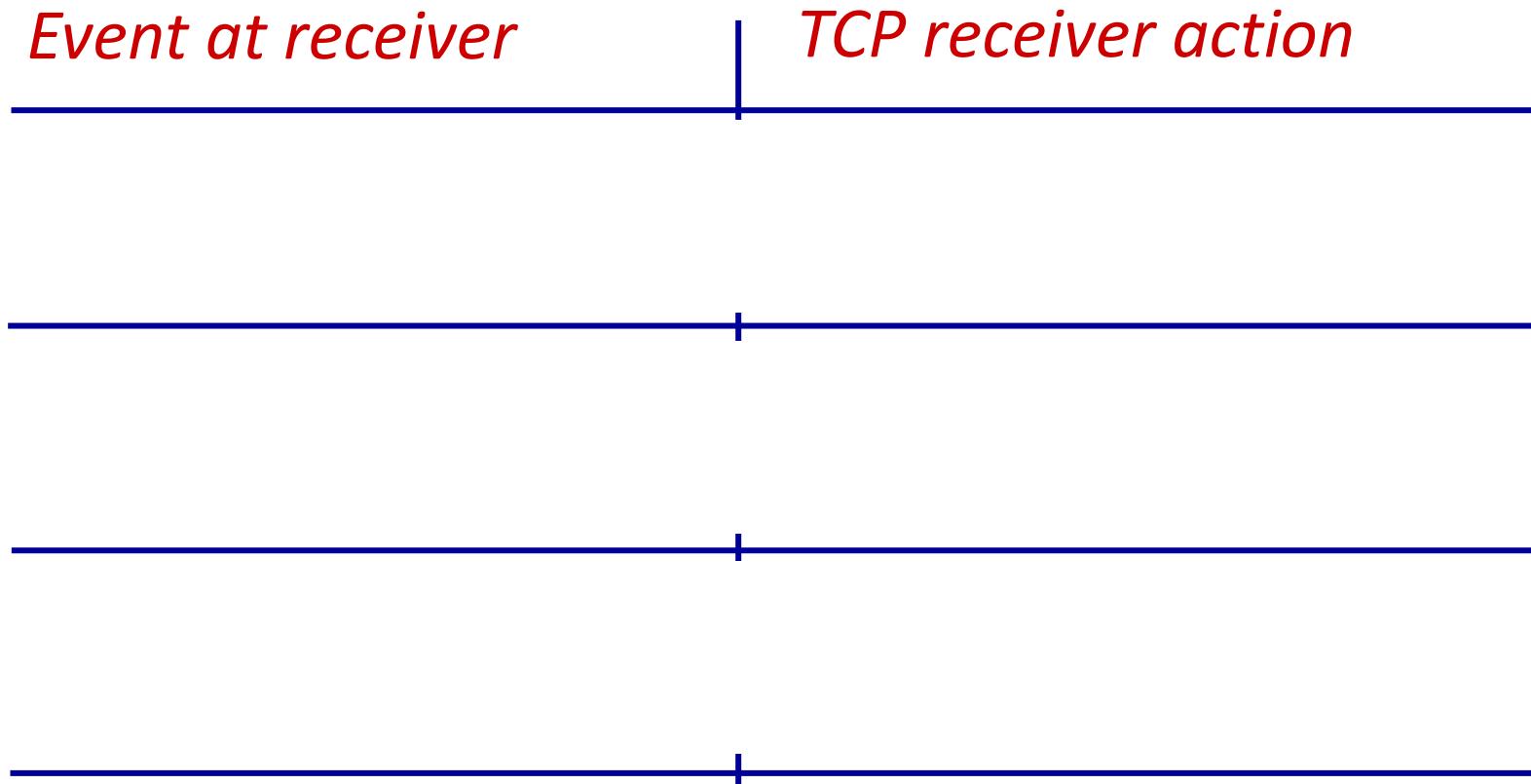
*event: timeout*

- retransmit segment that caused timeout
- restart timer

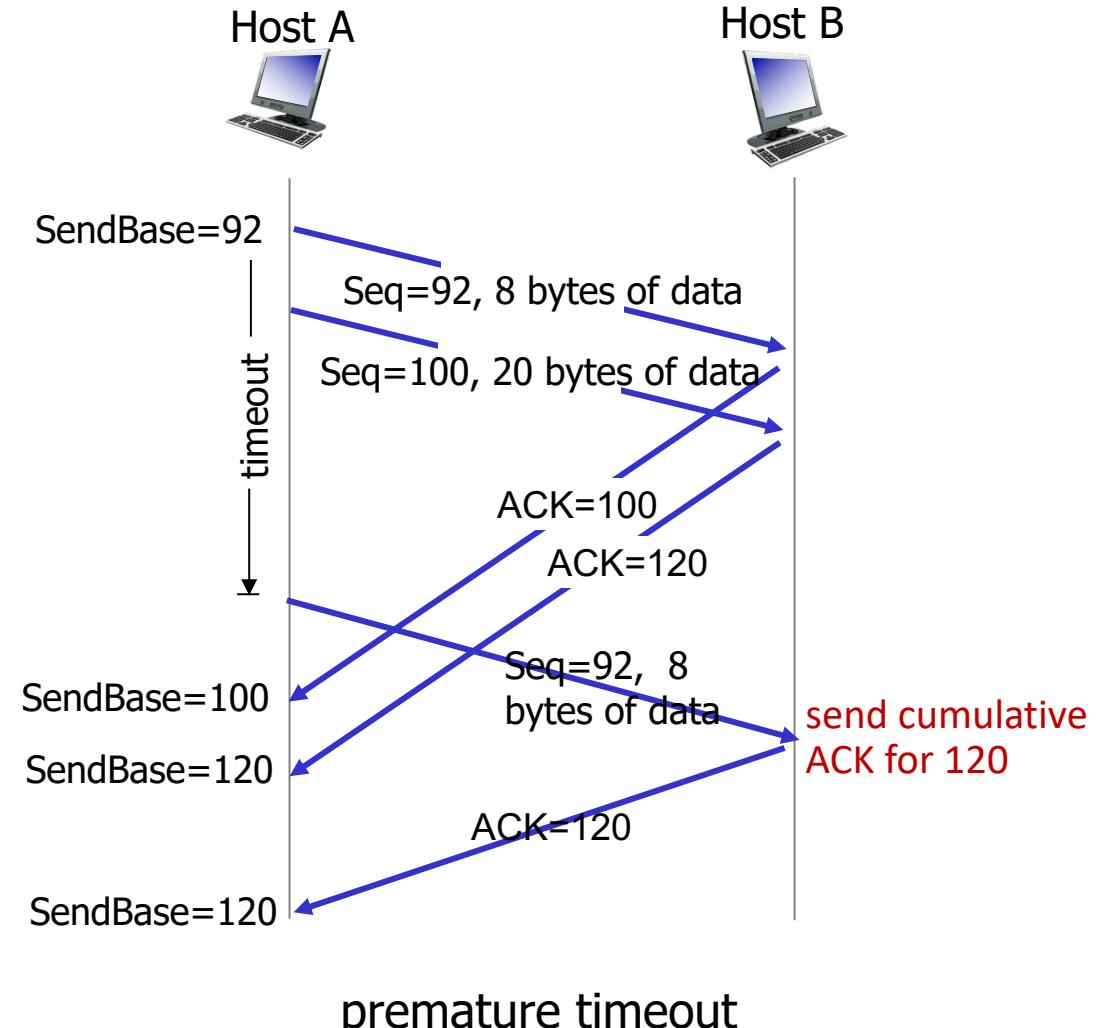
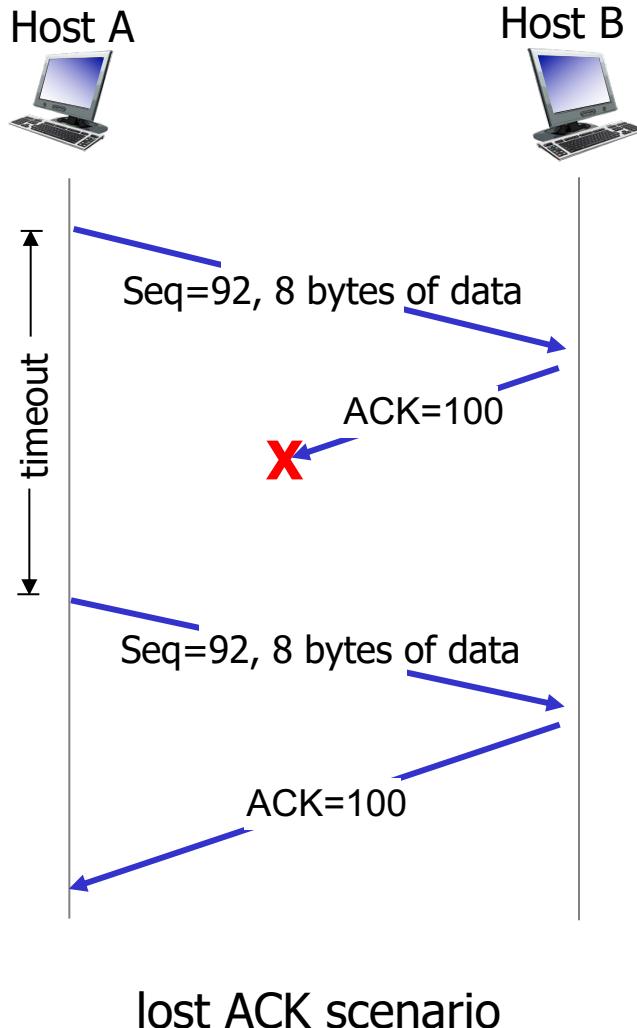
*event: ACK received*

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

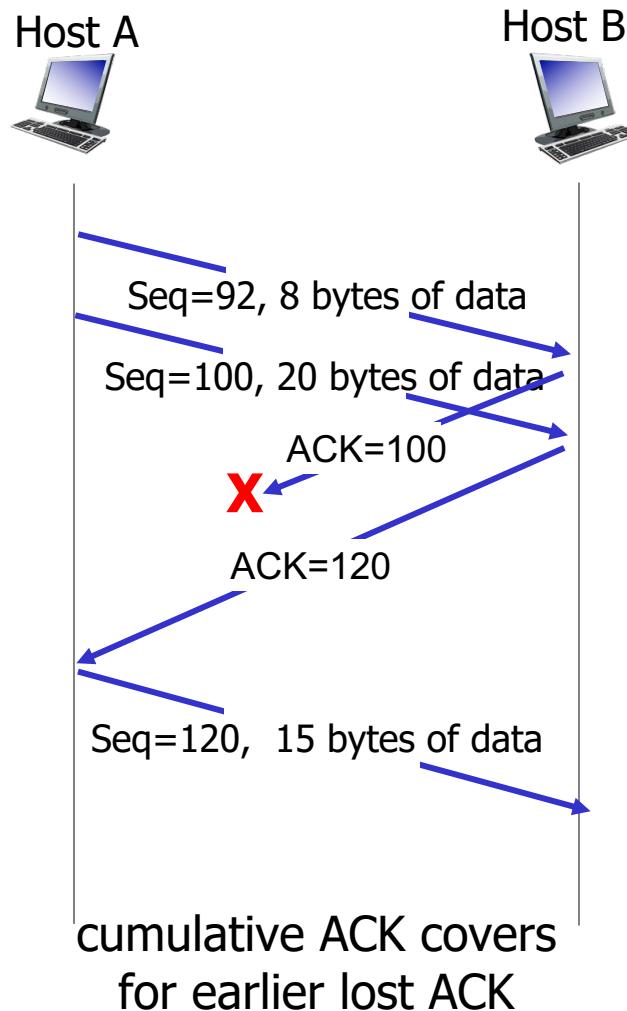
# TCP Receiver: ACK generation [RFC 5681]



# TCP: retransmission scenarios



# TCP: retransmission scenarios



# TCP fast retransmit

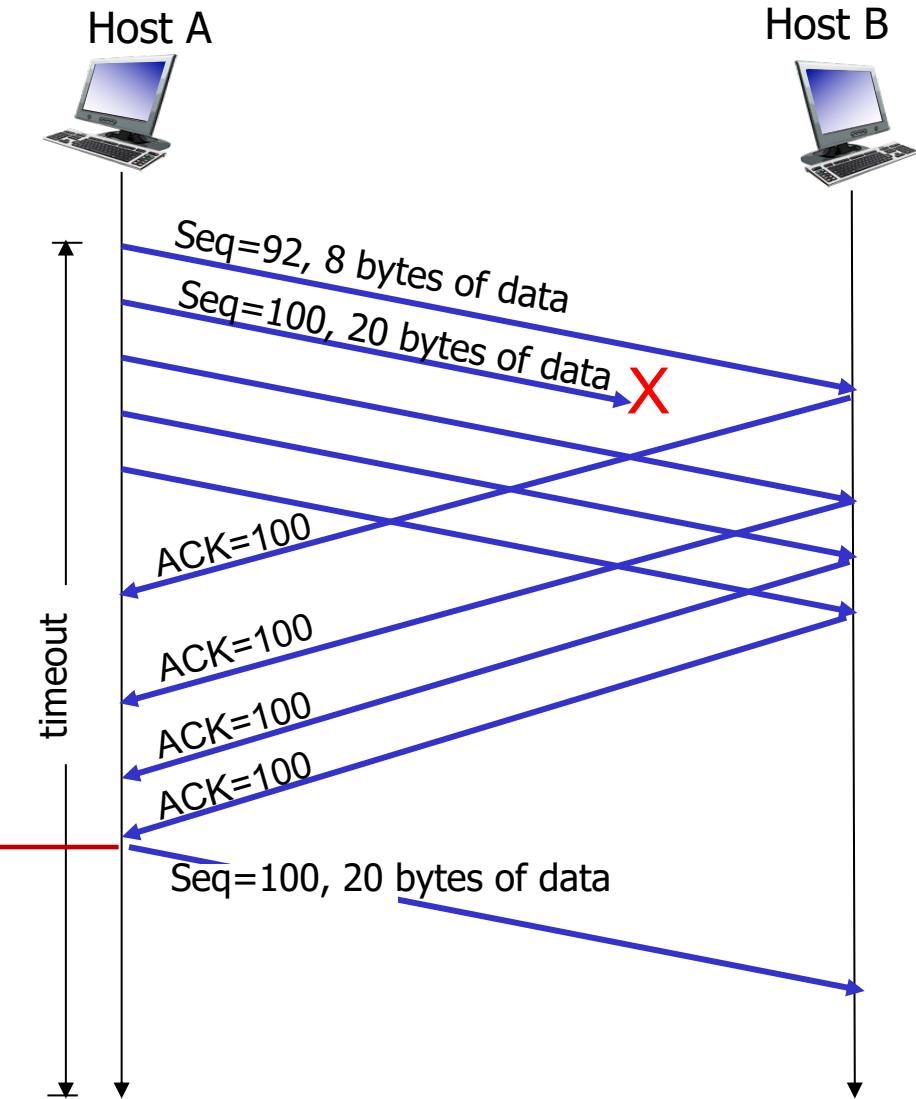
## *TCP fast retransmit*

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

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



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



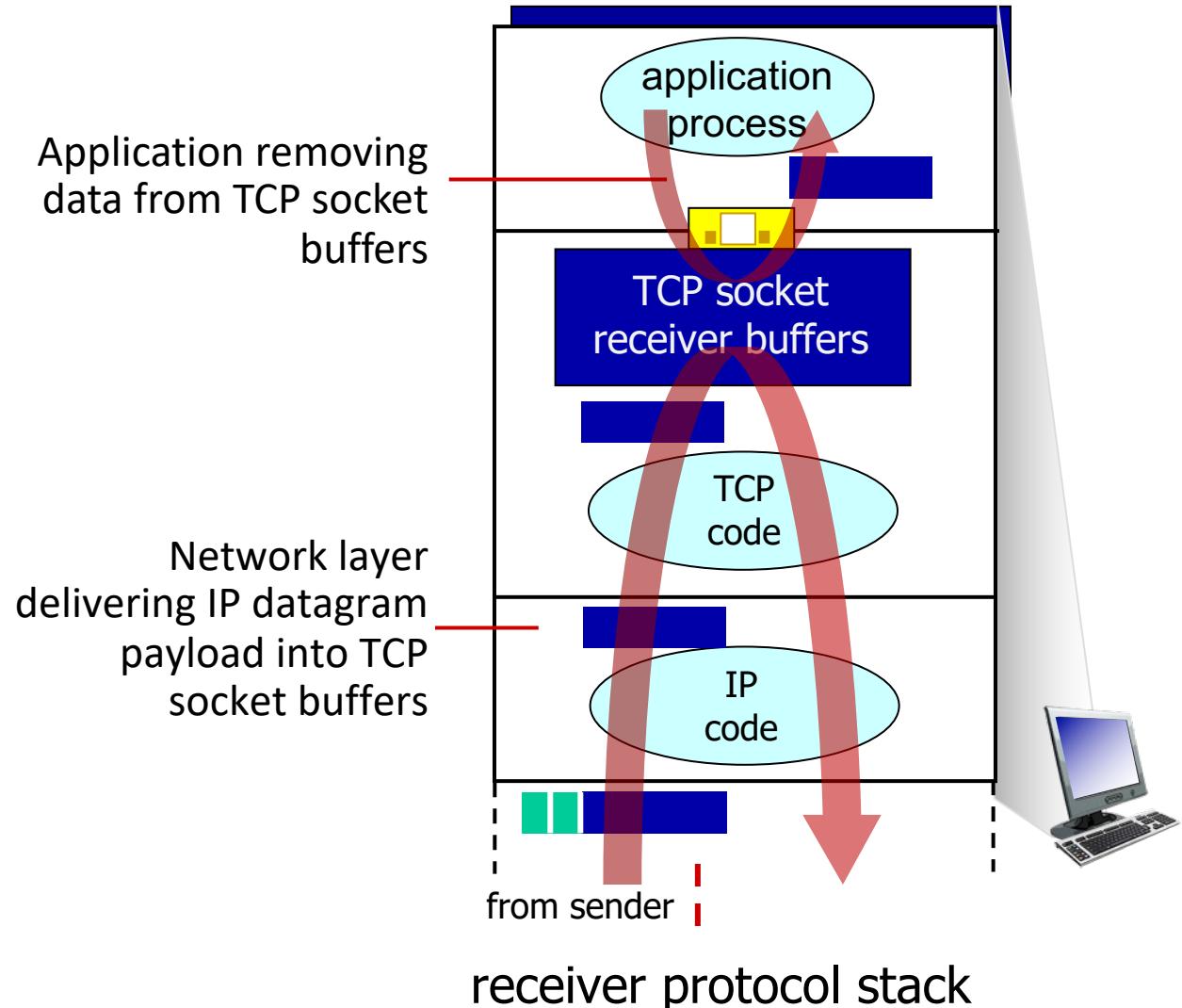
# Chapter 3: roadmap

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



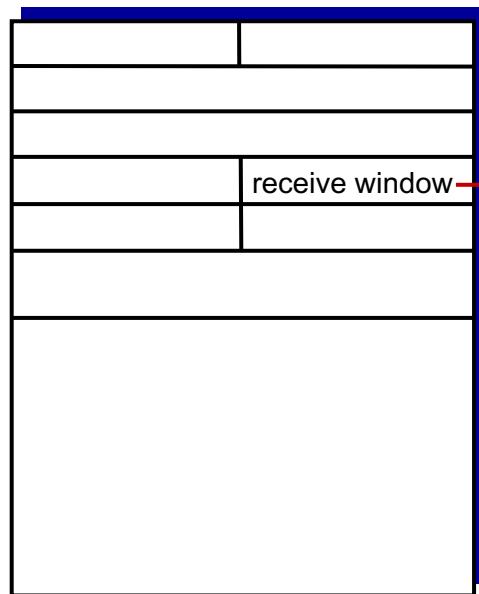
# TCP flow control

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



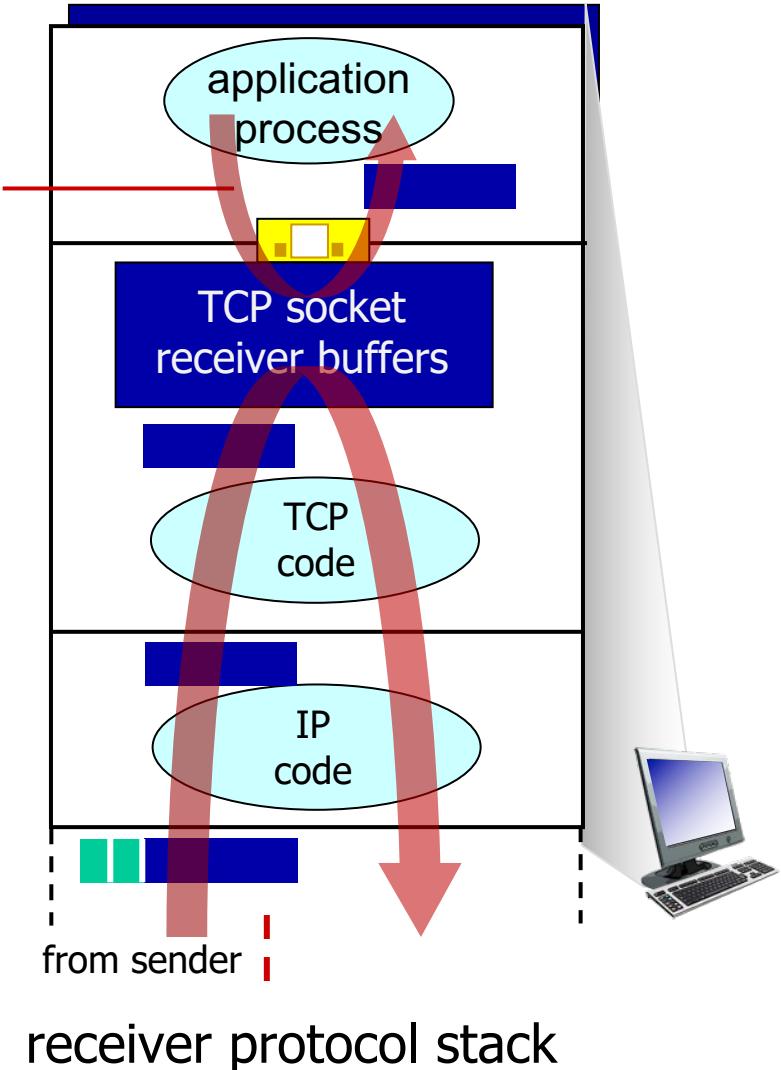
# TCP flow control

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



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



receiver protocol stack

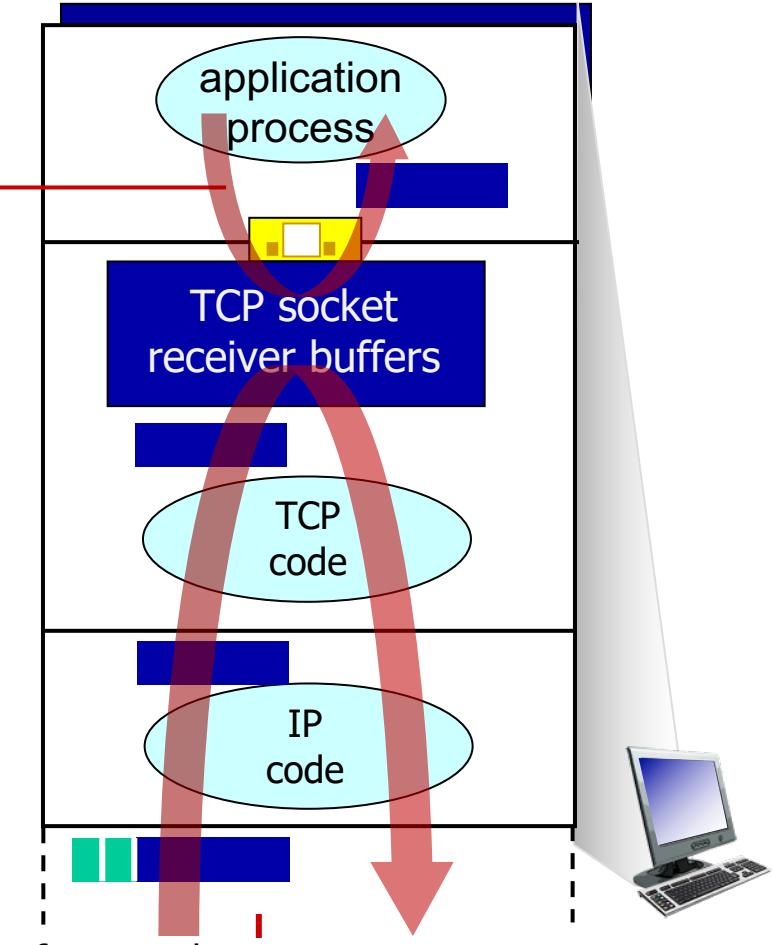
# TCP flow control

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

## flow control

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

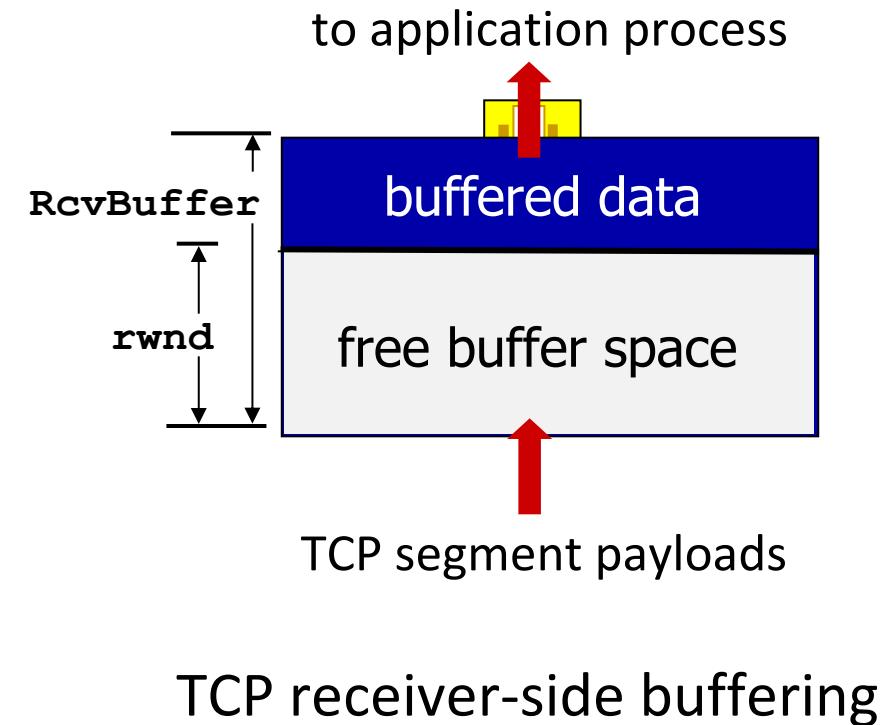
Application removing data from TCP socket buffers



receiver protocol stack

# TCP flow control

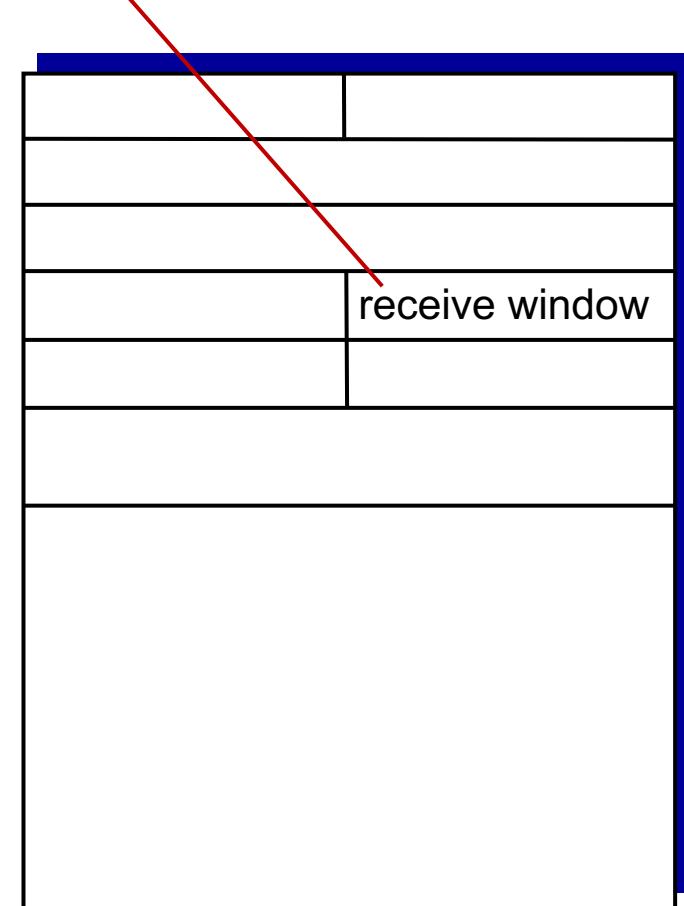
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive buffer will not overflow



# TCP 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

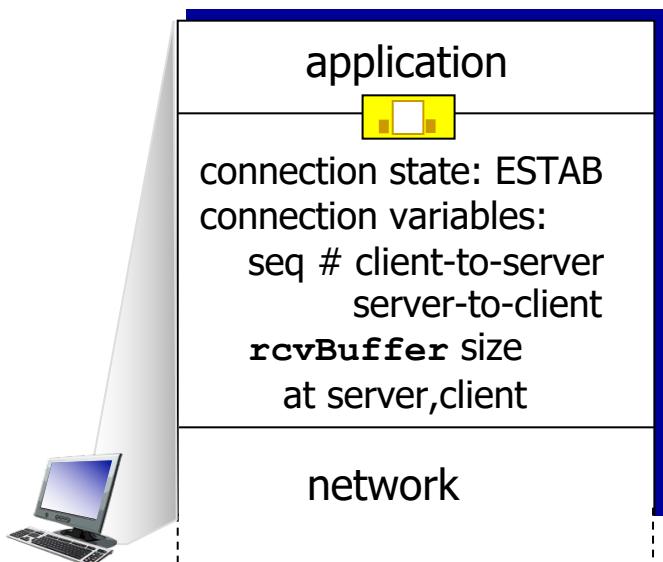


TCP segment format

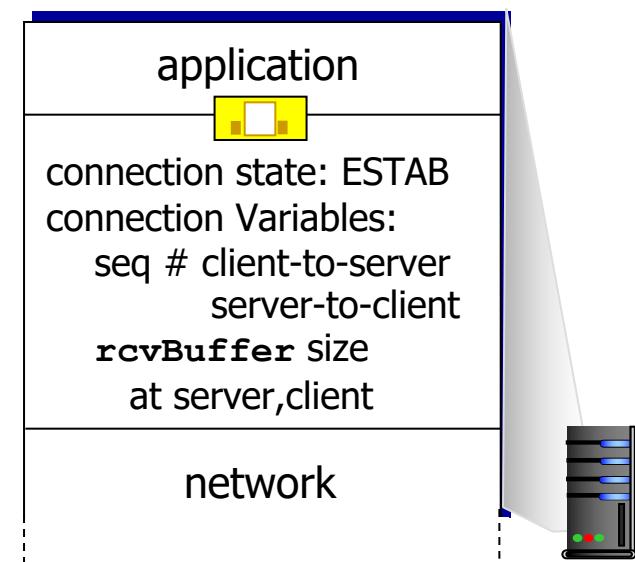
# TCP connection management

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)



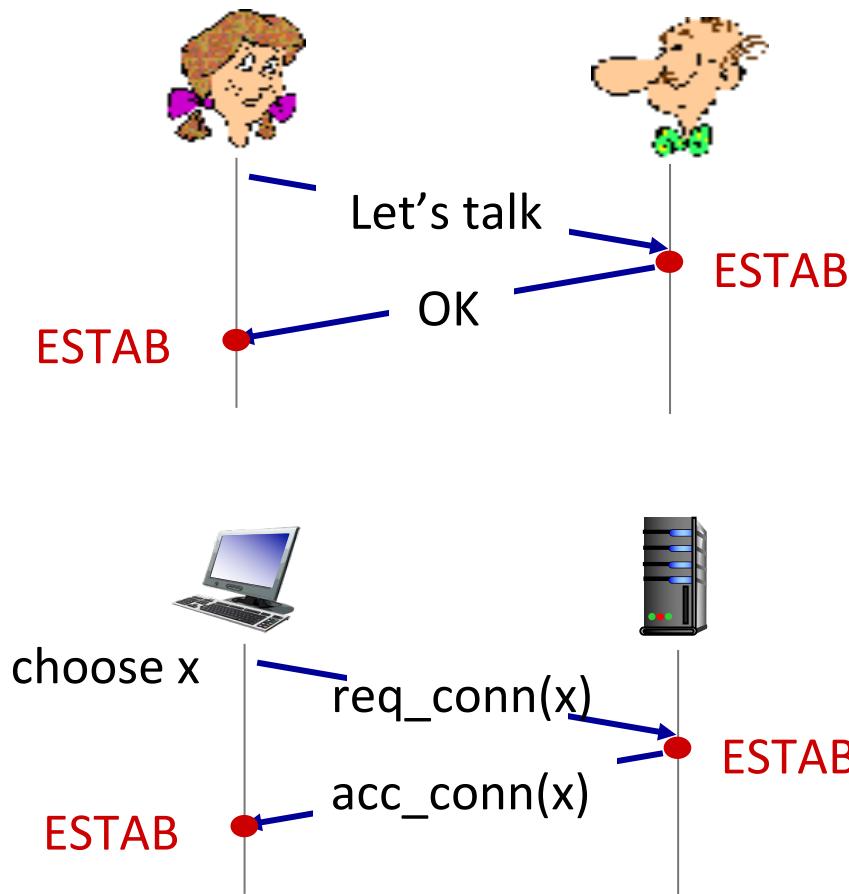
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

# Agreeing to establish a connection

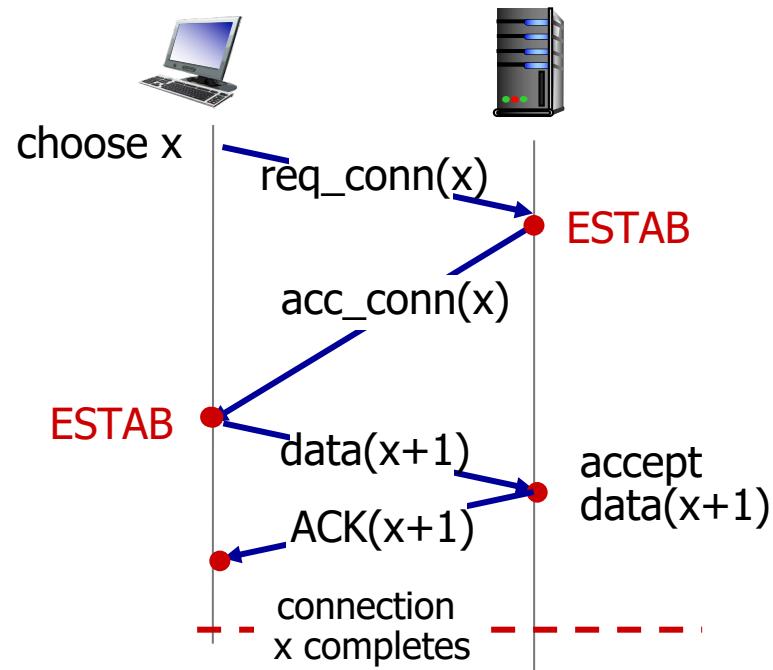
2-way handshake:



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

- variable delays
- retransmitted messages (e.g.  $\text{req\_conn}(x)$ ) due to message loss
- message reordering
- can't “see” other side

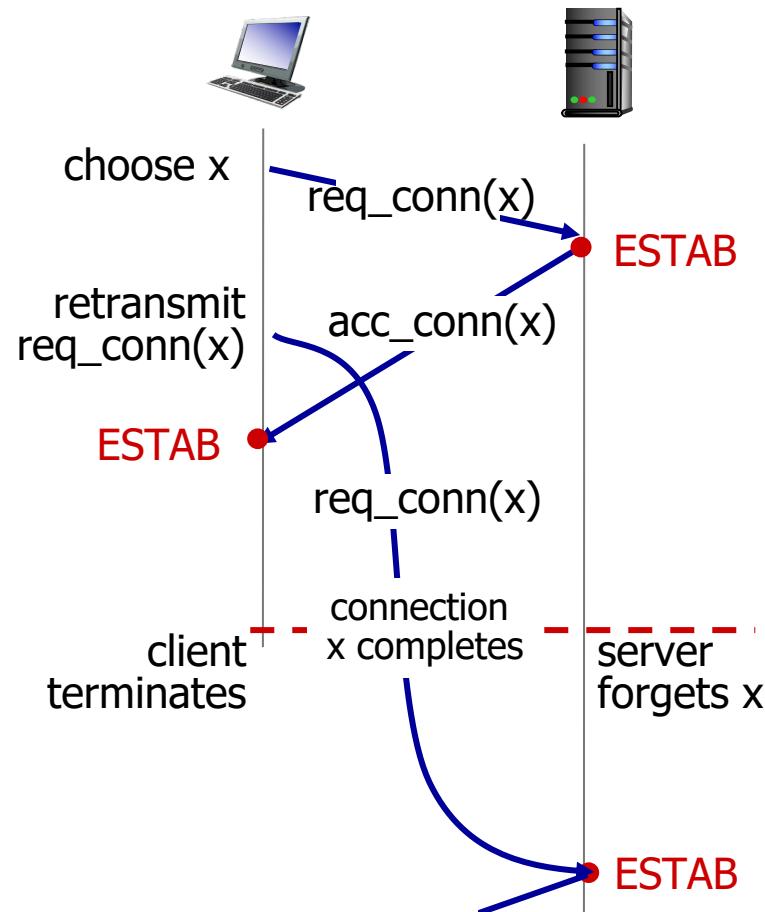
# 2-way handshake scenarios



No problem!

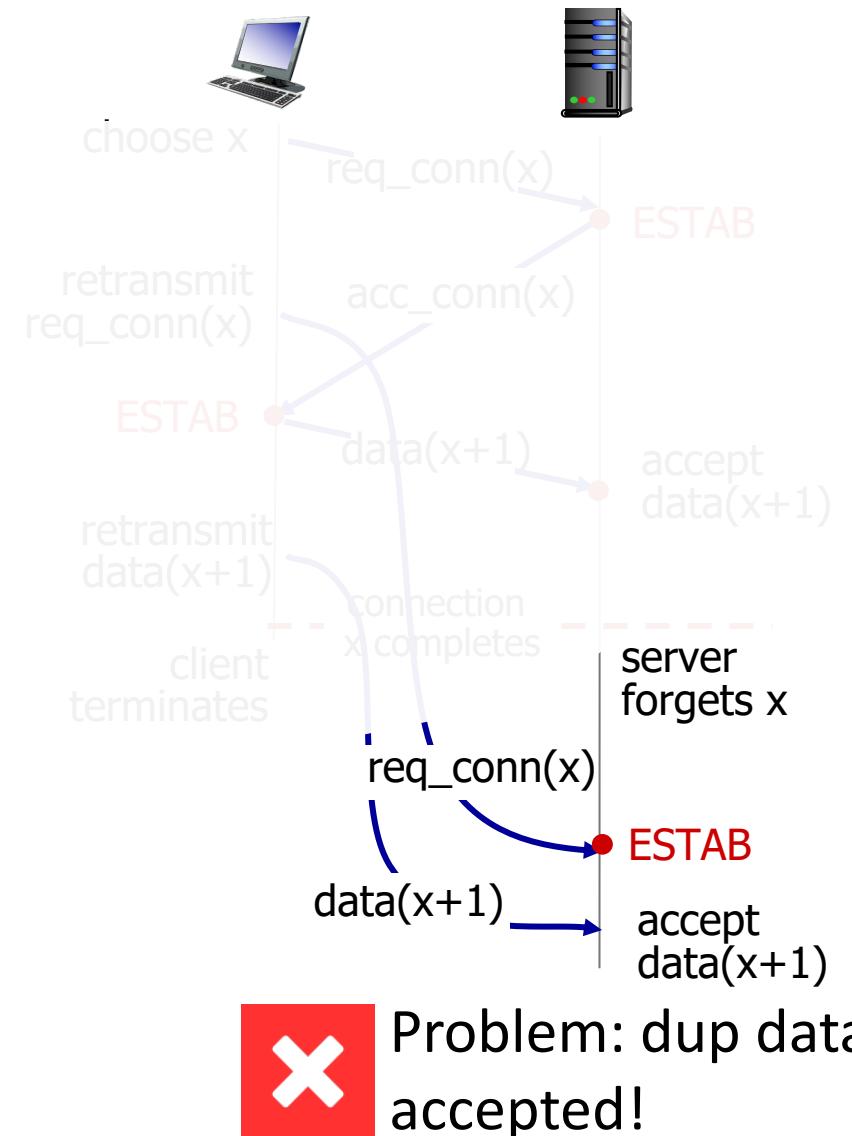


# 2-way handshake scenarios



Problem: half open connection! (no client)

# 2-way handshake scenarios



# TCP 3-way handshake

## Client state

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

LISTEN

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

SYNSENT

choose init seq num, x  
send TCP SYN msg



SYNbit=1, Seq=x

ESTAB

received SYNACK(x)  
indicates server is live;  
send ACK for SYNACK;  
this segment may contain  
client-to-server data

SYNbit=1, Seq=y  
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

## Server state

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

LISTEN

SYN RCV

choose init seq num, y  
send TCP SYNACK  
msg, acking SYN

ESTAB

received ACK(y)  
indicates client is live

# A human 3-way handshake protocol



# Closing a TCP connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

# Chapter 3: roadmap

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



# Principles of congestion control

## Congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!



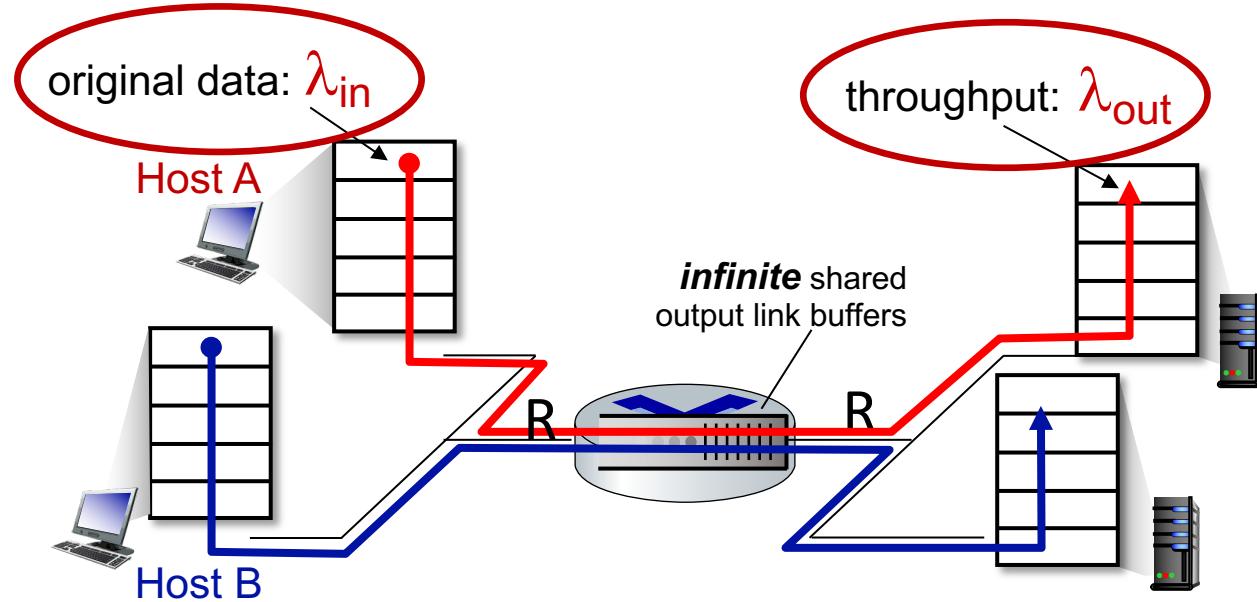
**congestion control:**  
too many senders,  
sending too fast

**flow control:** one sender  
too fast for one receiver

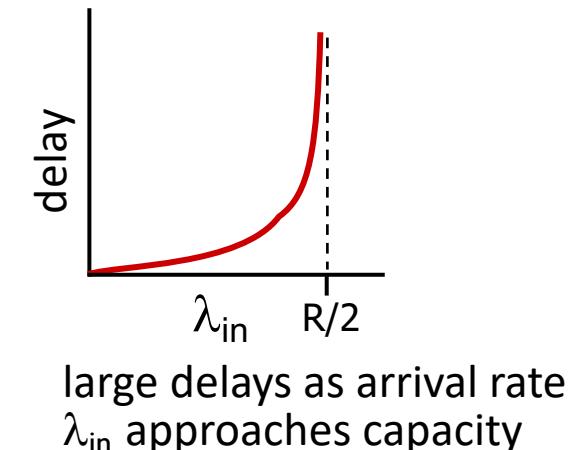
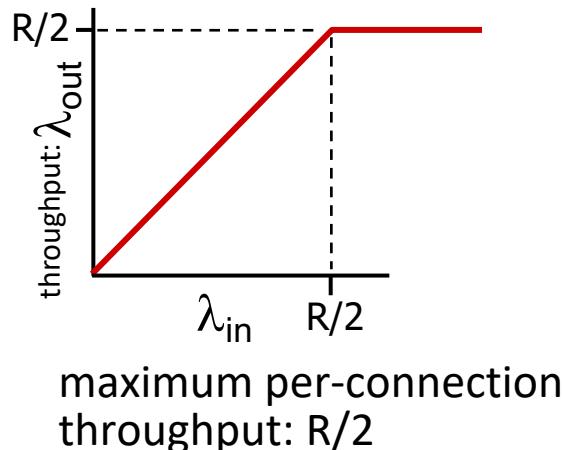
# Causes/costs of congestion: scenario 1

Simplest scenario:

- one router, infinite buffers
- input, output link capacity:  $R$
- two flows
- no retransmissions needed

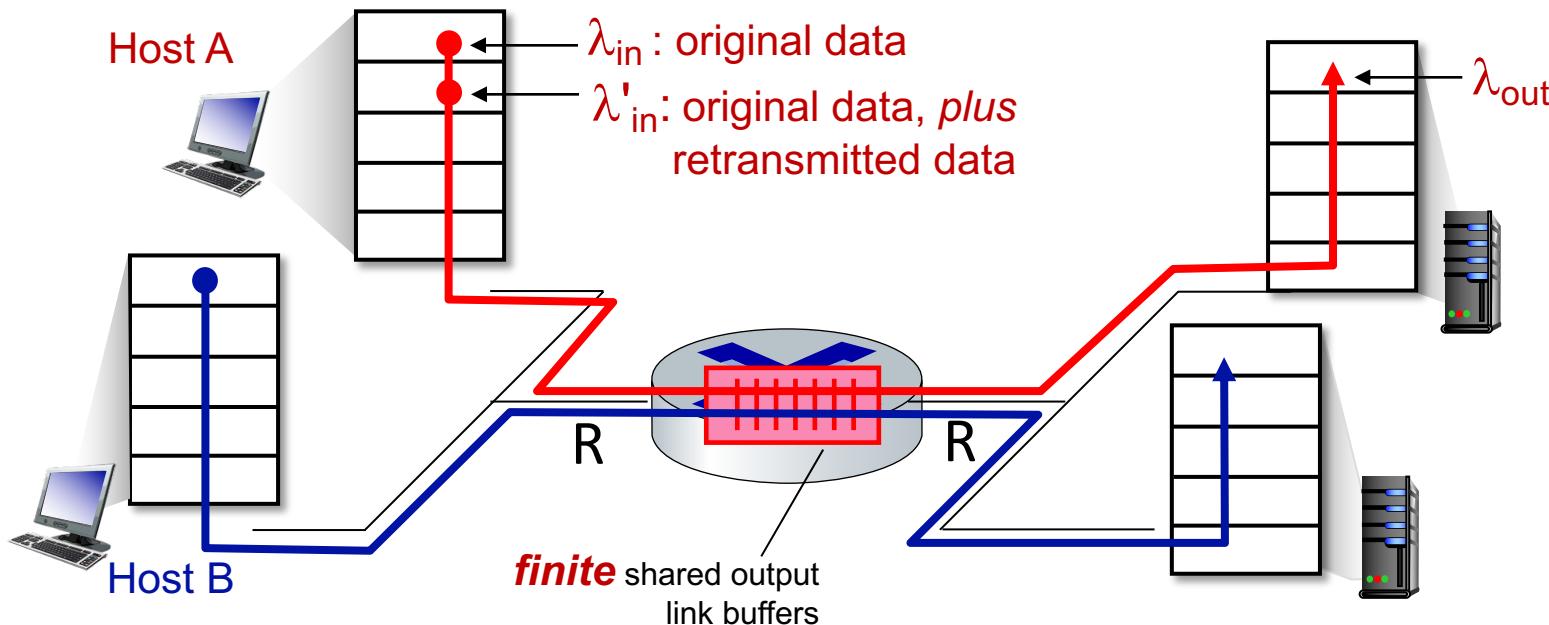


**Q:** What happens as arrival rate  $\lambda_{in}$  approaches  $R/2$ ?



# Causes/costs of congestion: scenario 2

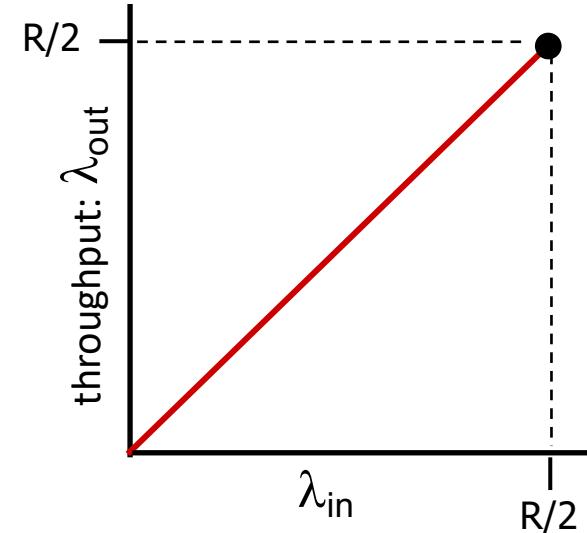
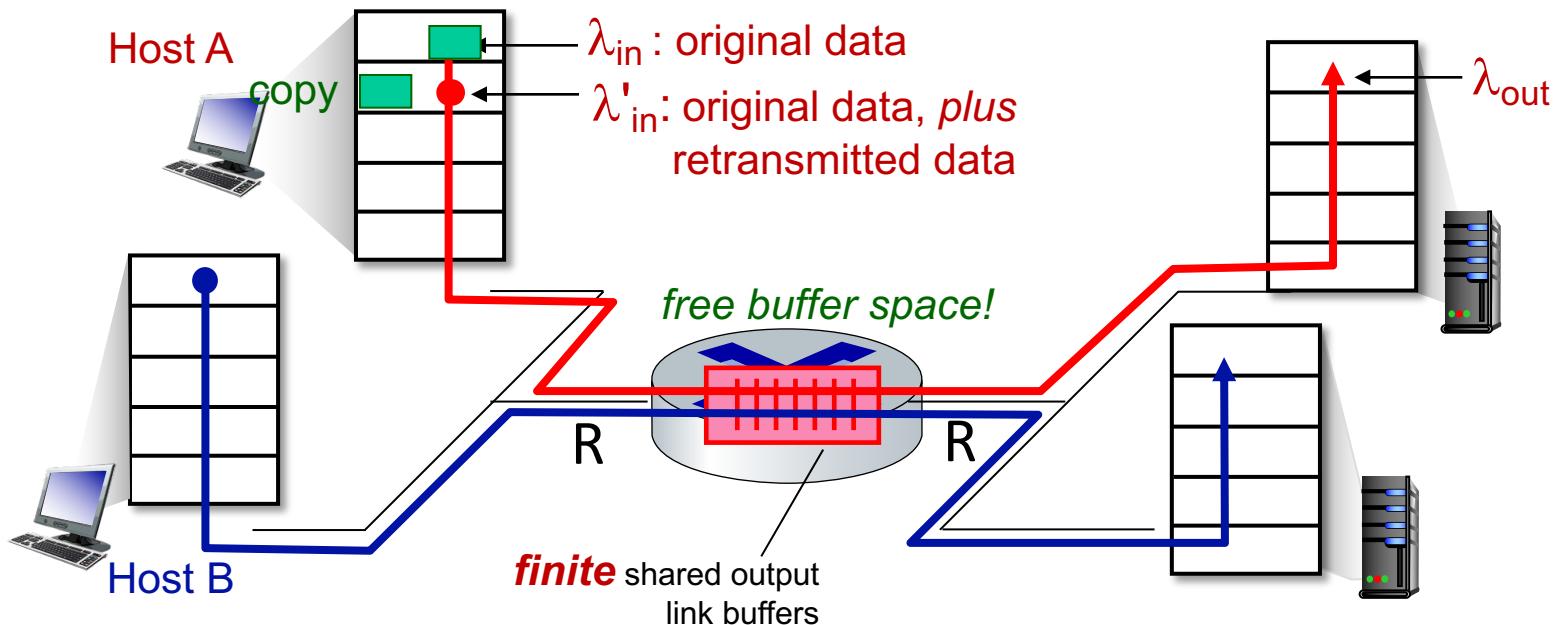
- one router, *finite* buffers
- sender retransmits lost, timed-out packet
  - application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* :  $\lambda'_{in} \geq \lambda_{in}$



# Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

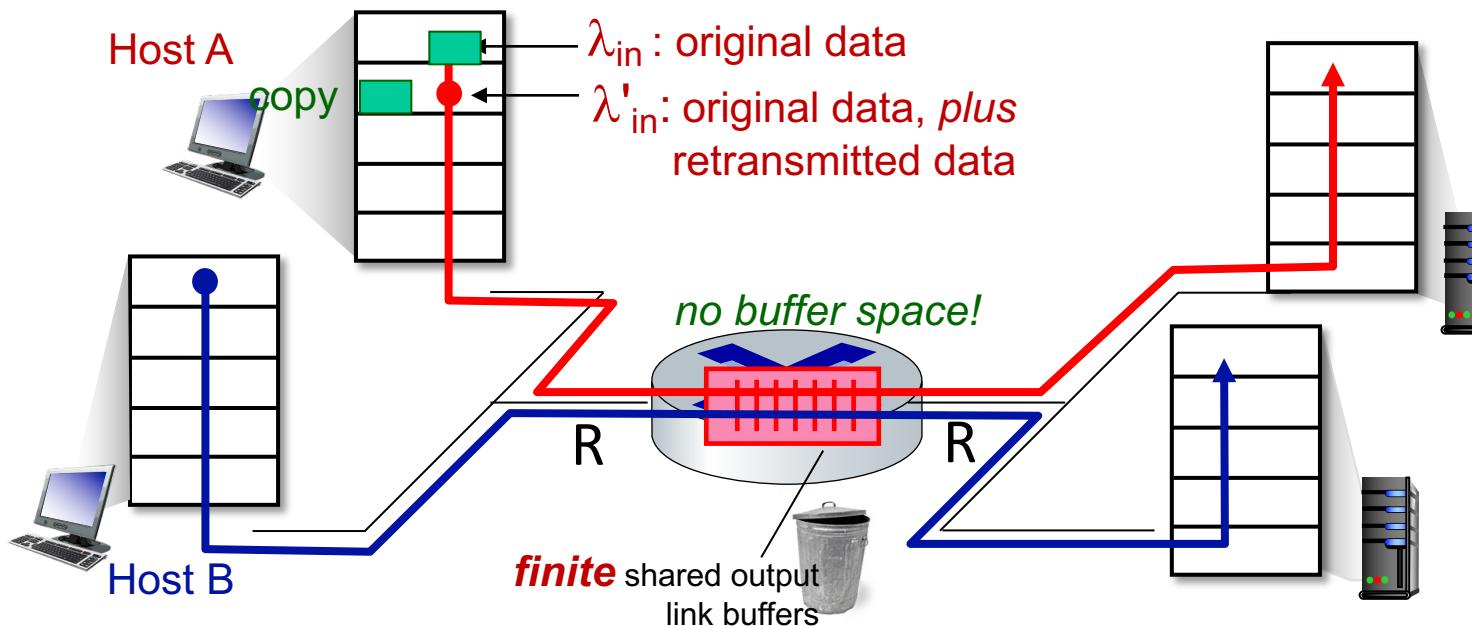
- sender sends only when router buffers available



# Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

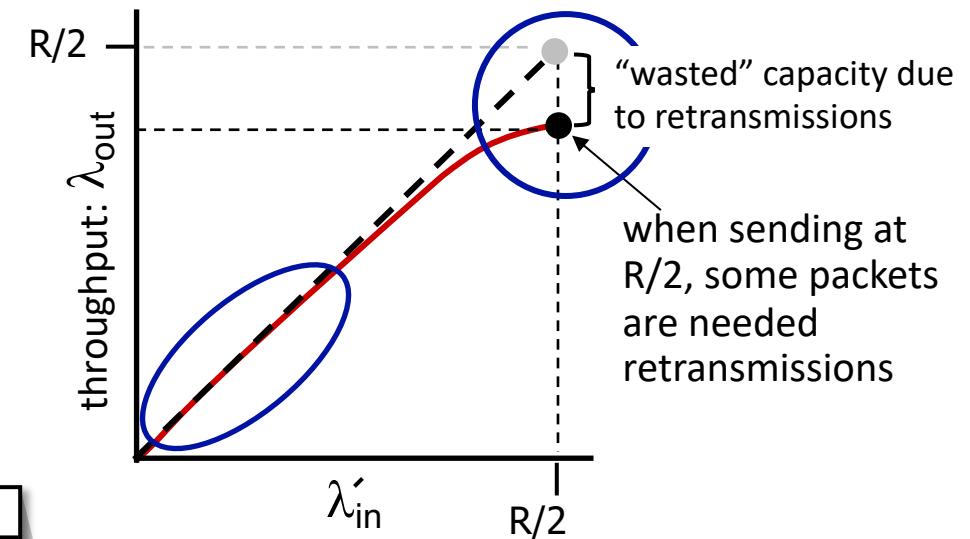
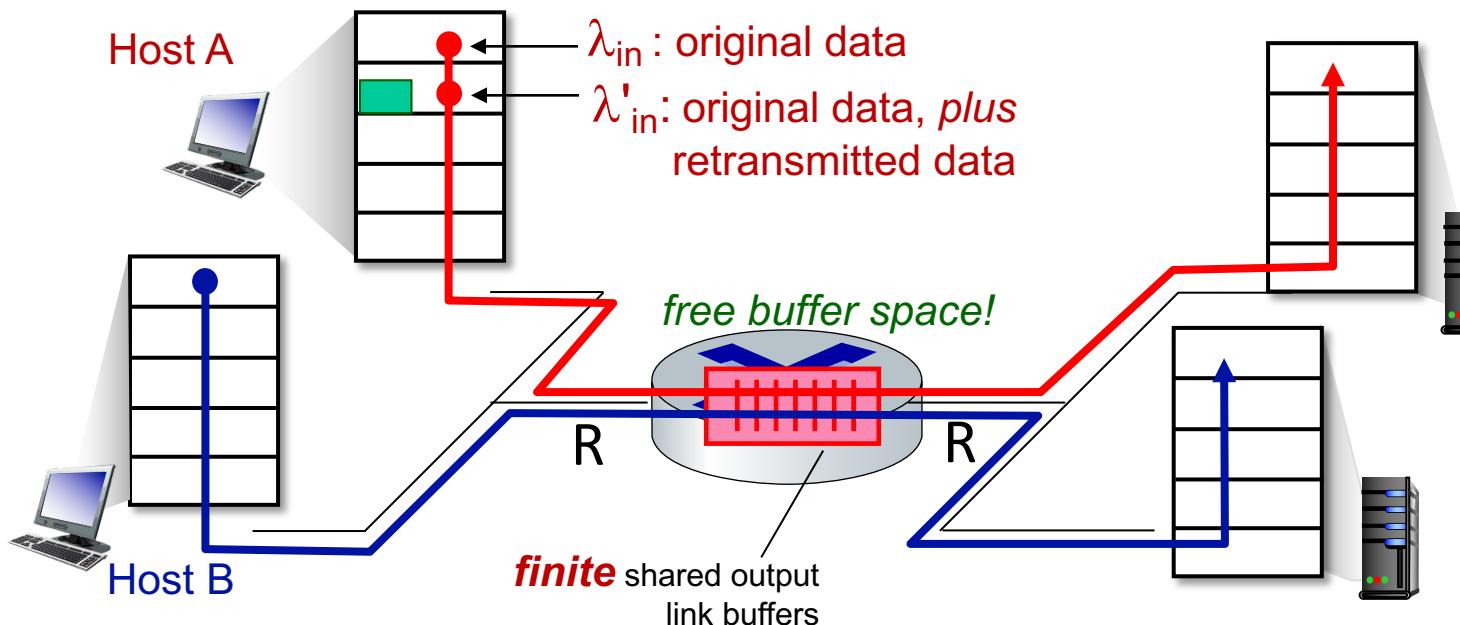
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

Idealization: *some* perfect knowledge

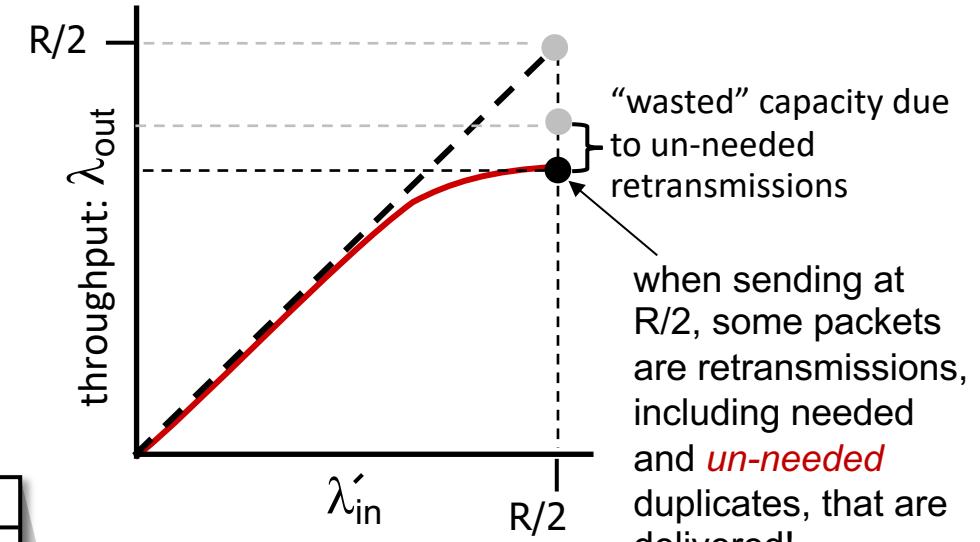
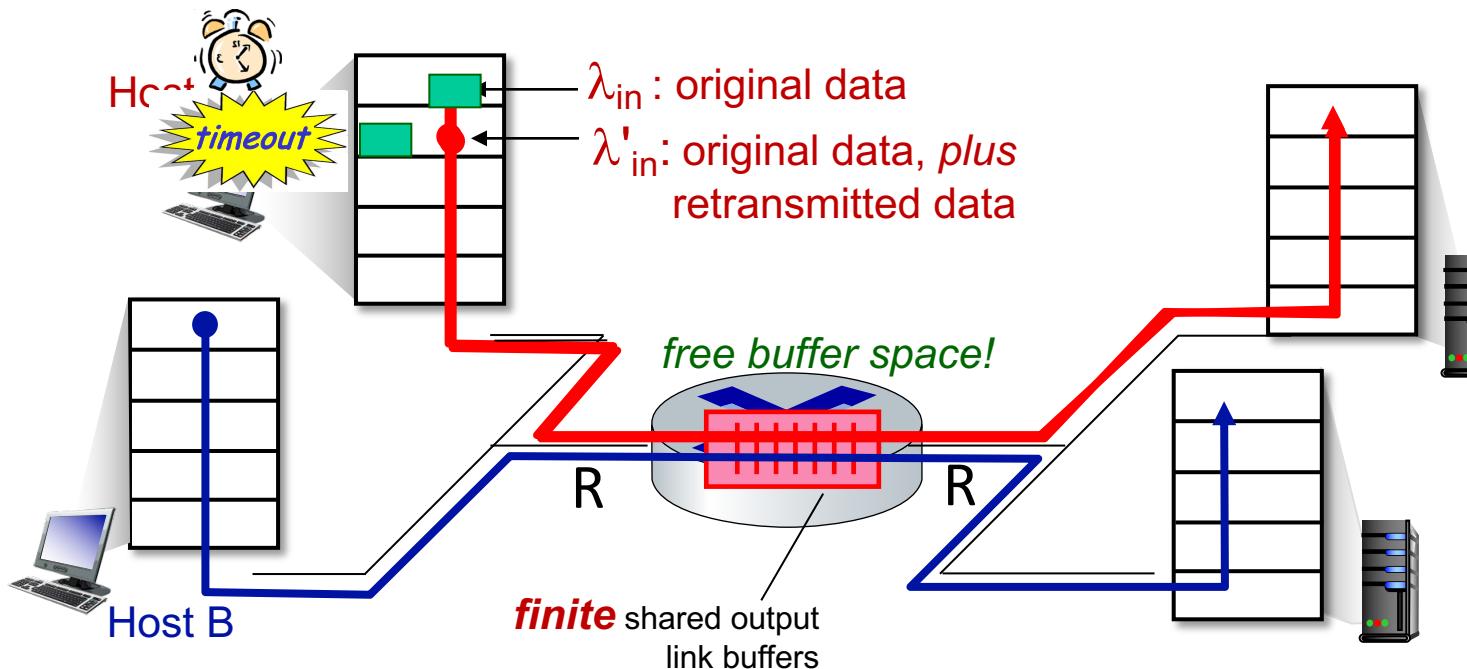
- packets can be lost (dropped at router) due to full buffers
- sender knows when packet has been dropped: only resends if packet *known* to be lost



# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

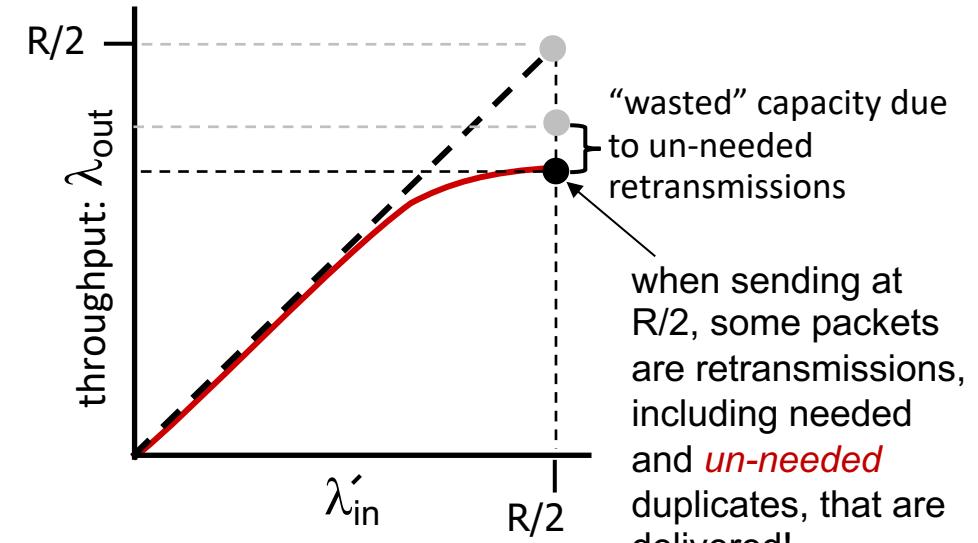
- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



# Causes/costs of congestion: scenario 2

## Realistic scenario: *un-needed duplicates*

- packets can be lost, dropped at router due to full buffers – requiring retransmissions
- but sender times can time out prematurely, sending *two* copies, *both* of which are delivered



## "costs" of congestion:

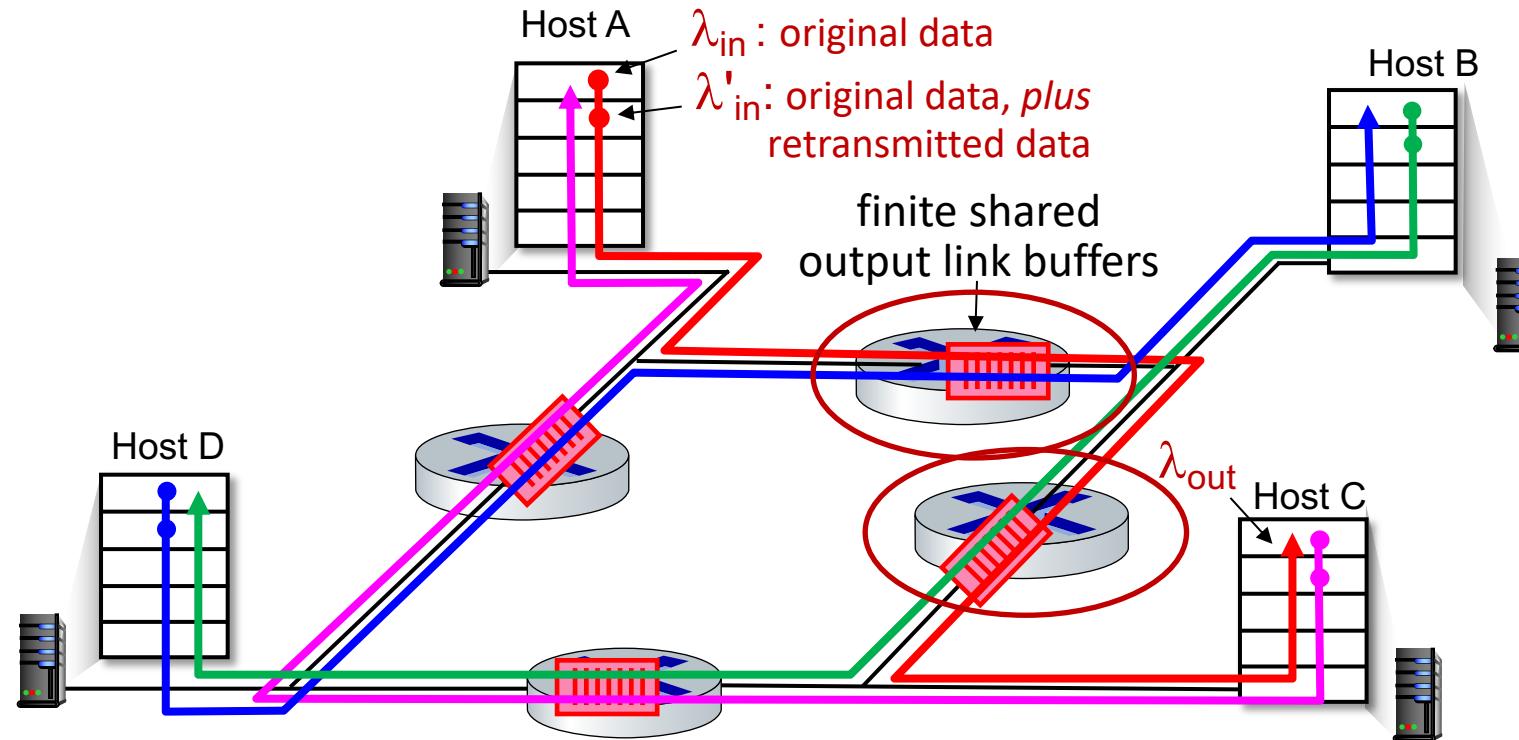
- more work (retransmission) for given receiver throughput
- unneeded retransmissions: link carries multiple copies of a packet
  - decreasing maximum achievable throughput

# Causes/costs of congestion: scenario 3

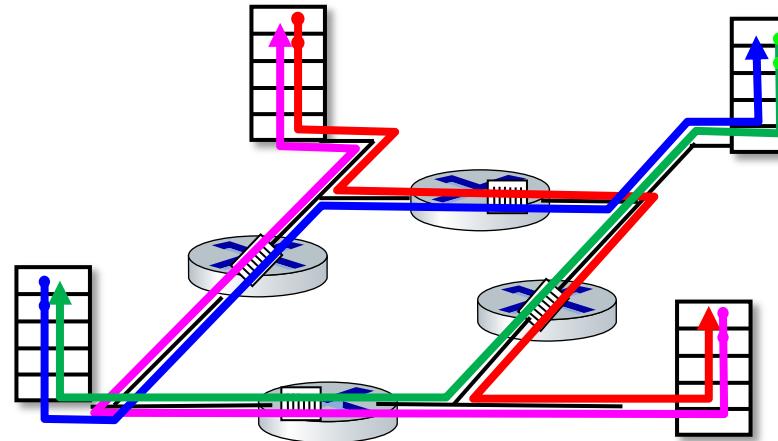
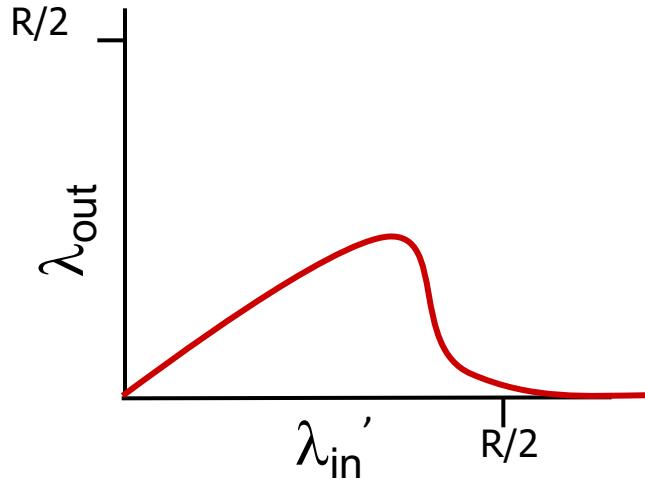
- four senders
- multi-hop paths
- timeout/retransmit

**Q:** what happens as  $\lambda_{in}$  and  $\lambda'_{in}$  increase ?

**A:** as red  $\lambda'_{in}$  increases, all arriving blue pkts at upper queue are dropped, blue throughput  $\rightarrow 0$



# Causes/costs of congestion: scenario 3

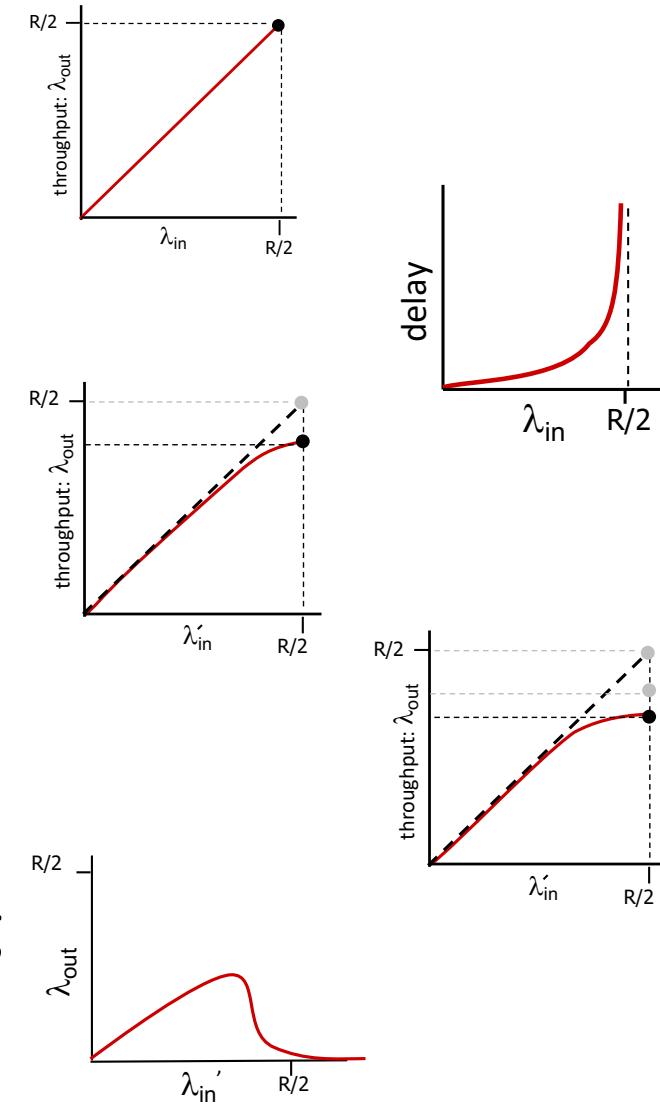


another “cost” of congestion:

- when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted!

# Causes/costs of congestion: insights

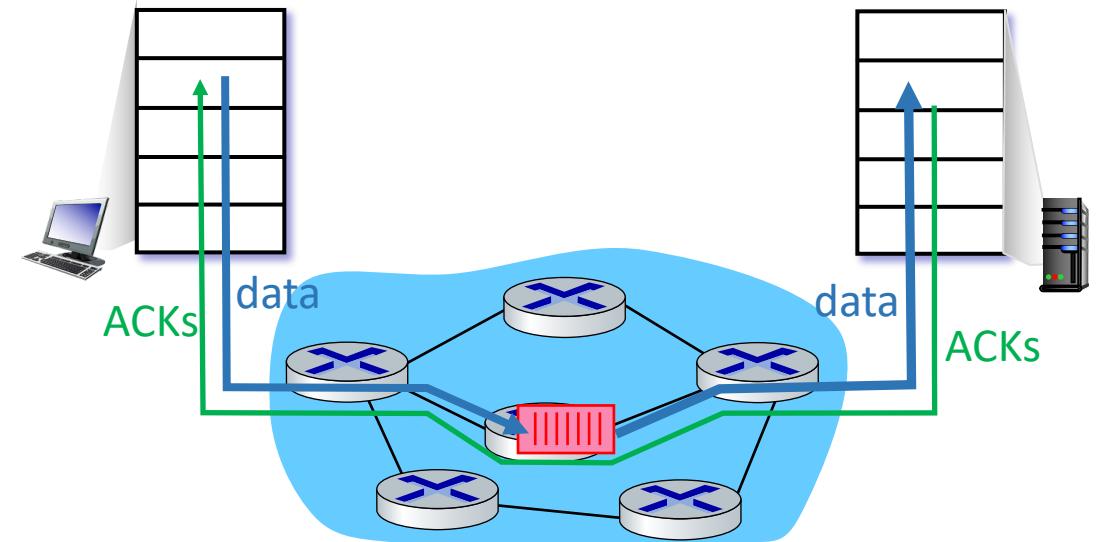
- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



# Approaches towards congestion control

## End-end congestion control:

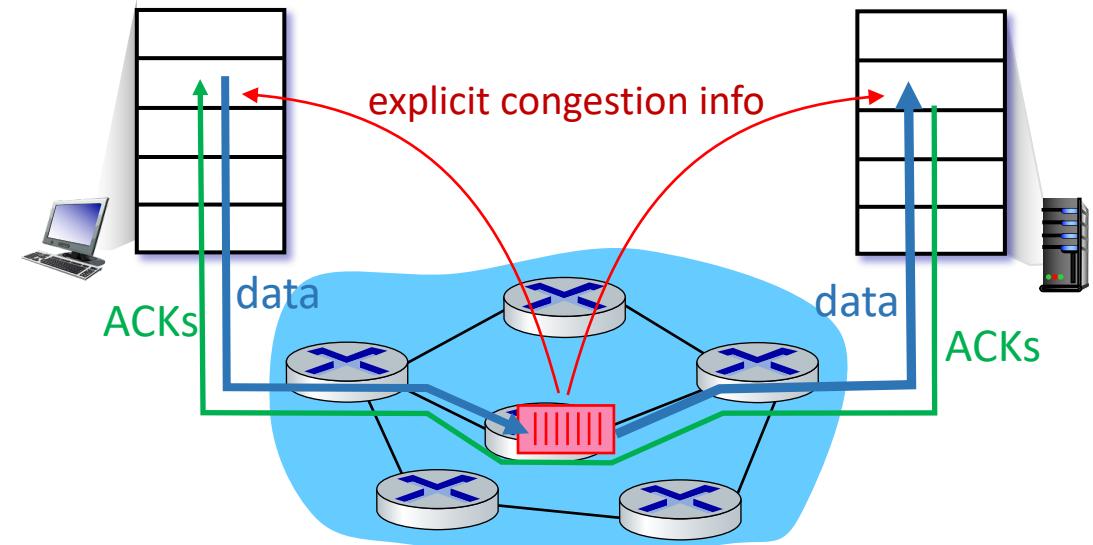
- no explicit feedback from network
- congestion *inferred* from observed loss, delay
- approach taken by TCP



# 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
- TCP ECN, ATM, DECbit protocols



# Chapter 3: roadmap

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



# TCP congestion control: AIMD

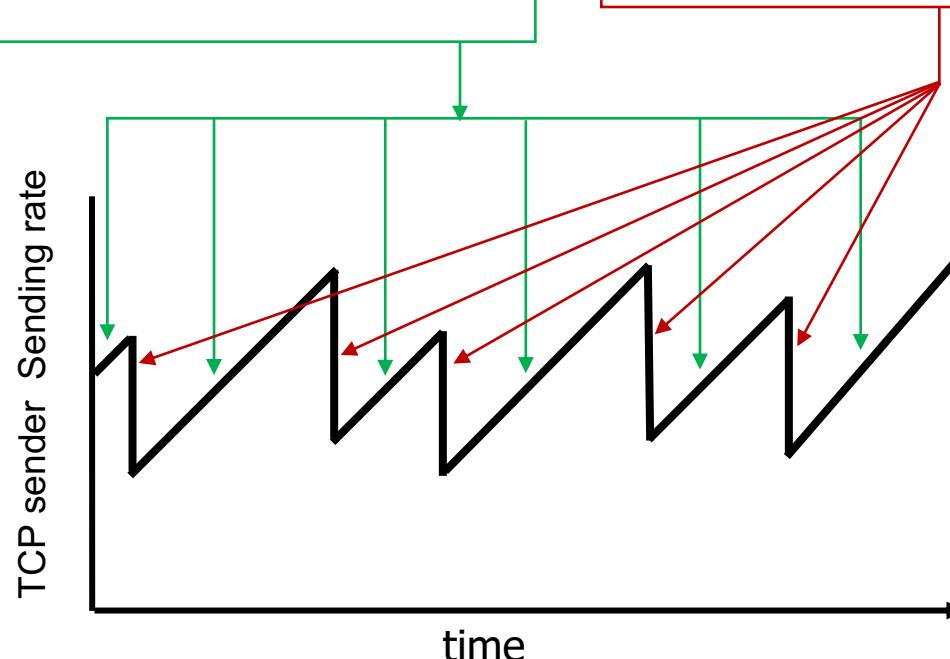
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

## 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

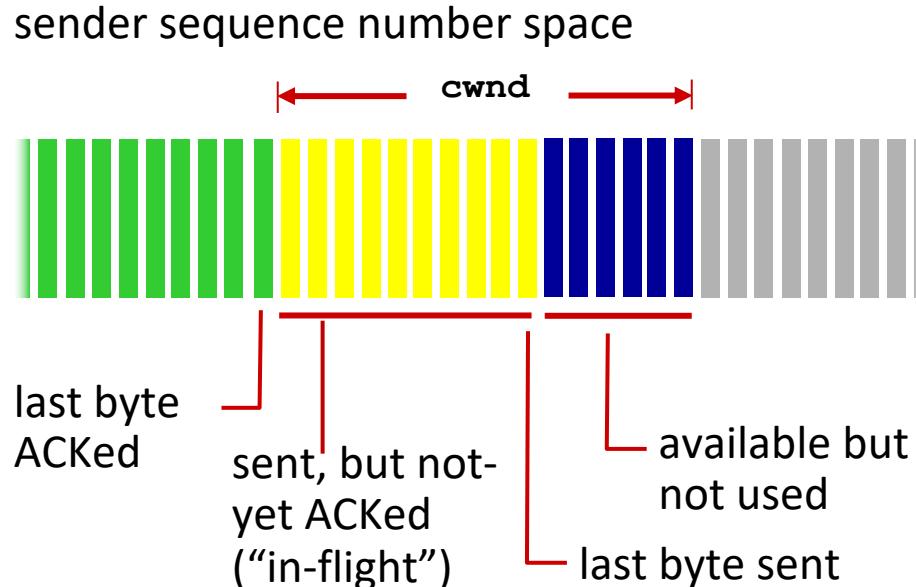
*Multiplicative decrease* detail: sending rate is

- 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: details



TCP sending behavior:

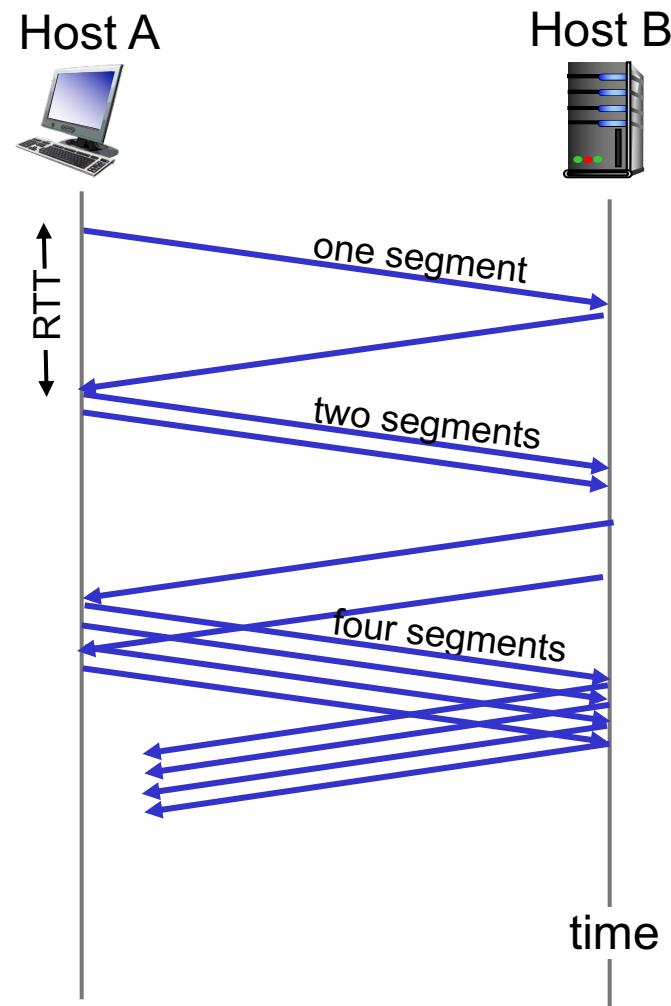
- *roughly*: send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

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

# 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



# TCP: from slow start to congestion avoidance

**Q:** when should it stop growing exponential?

**A:** Initially after loss occurs.  
Then when a threshold is passed, ssthresh.

**Q:** what should it do after it stops growing exponential?

**A:** Send data linearly.

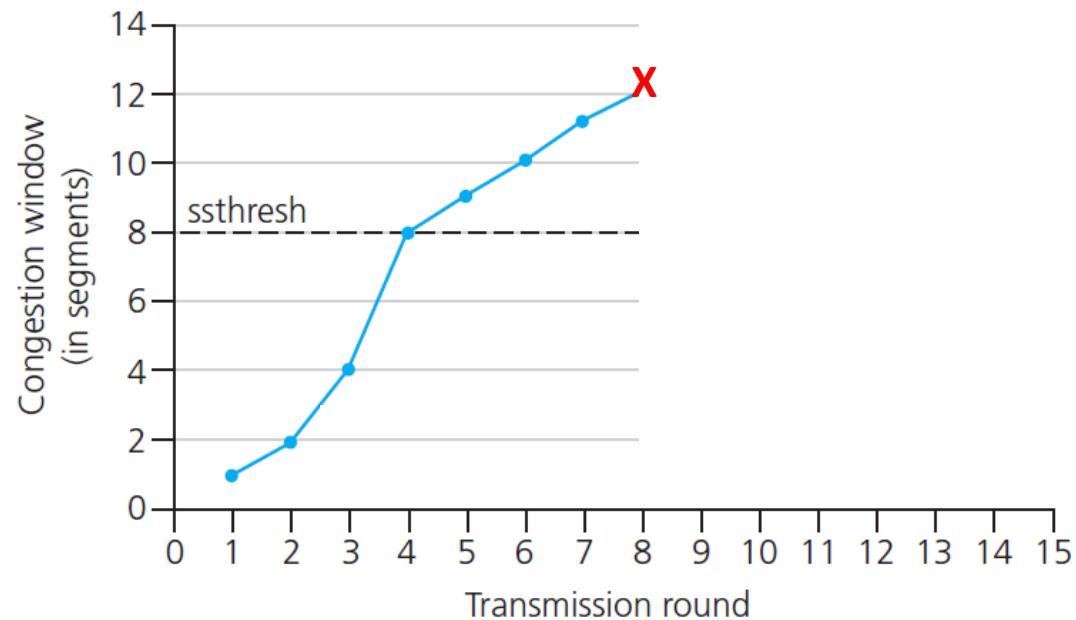
# TCP: from slow start to congestion avoidance

**Q:** when should the exponential increase switch to linear?

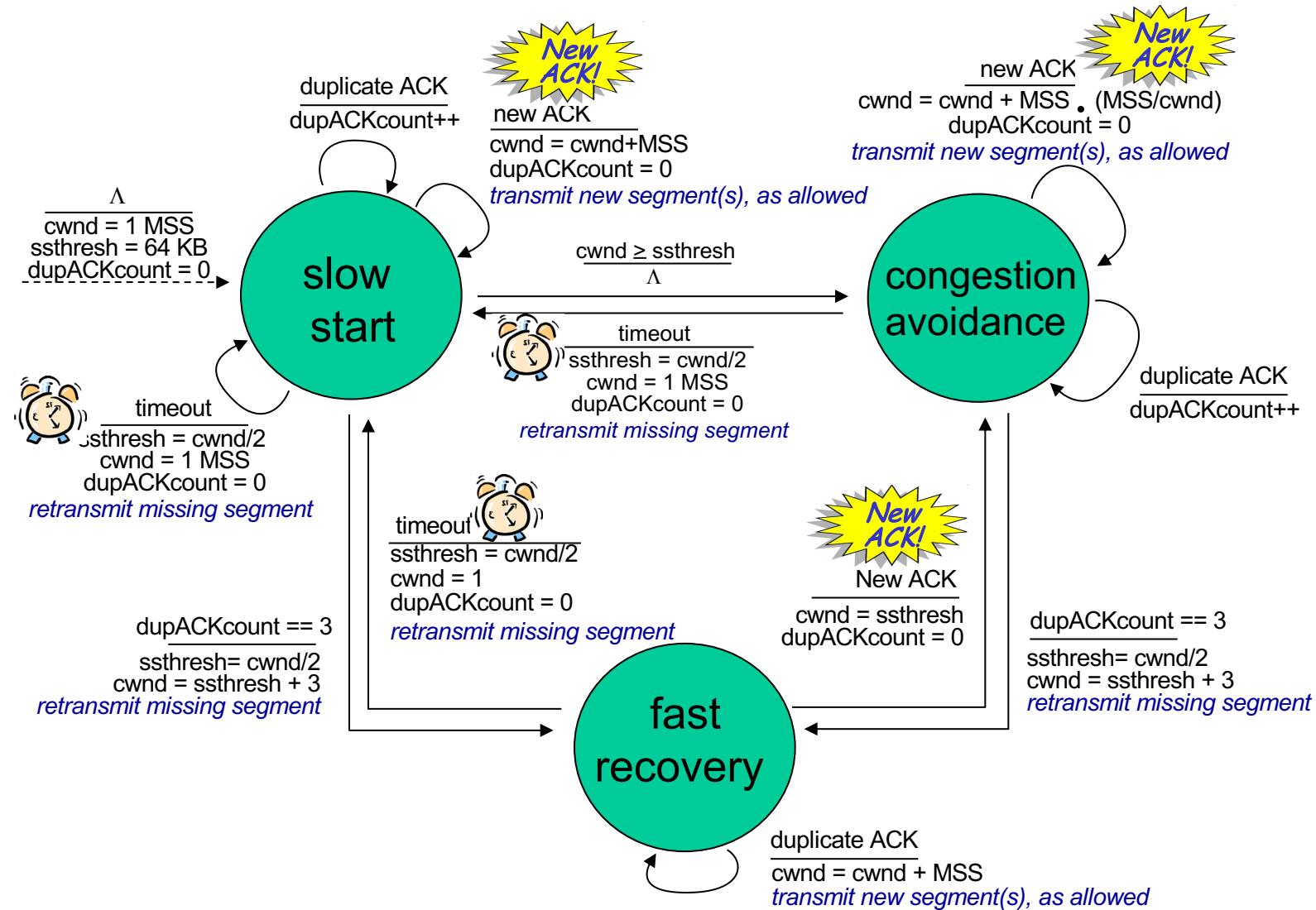
**A:** when **cwnd** gets to 1/2 of its value before timeout.

## Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

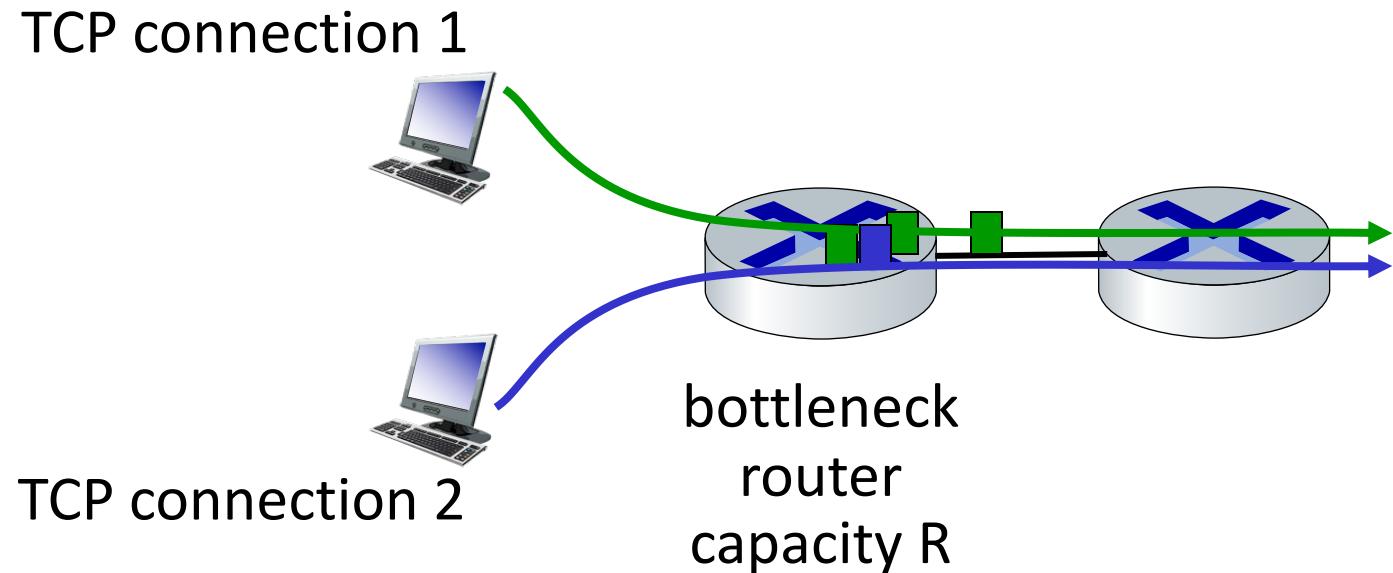


# Summary: TCP congestion control



# TCP fairness

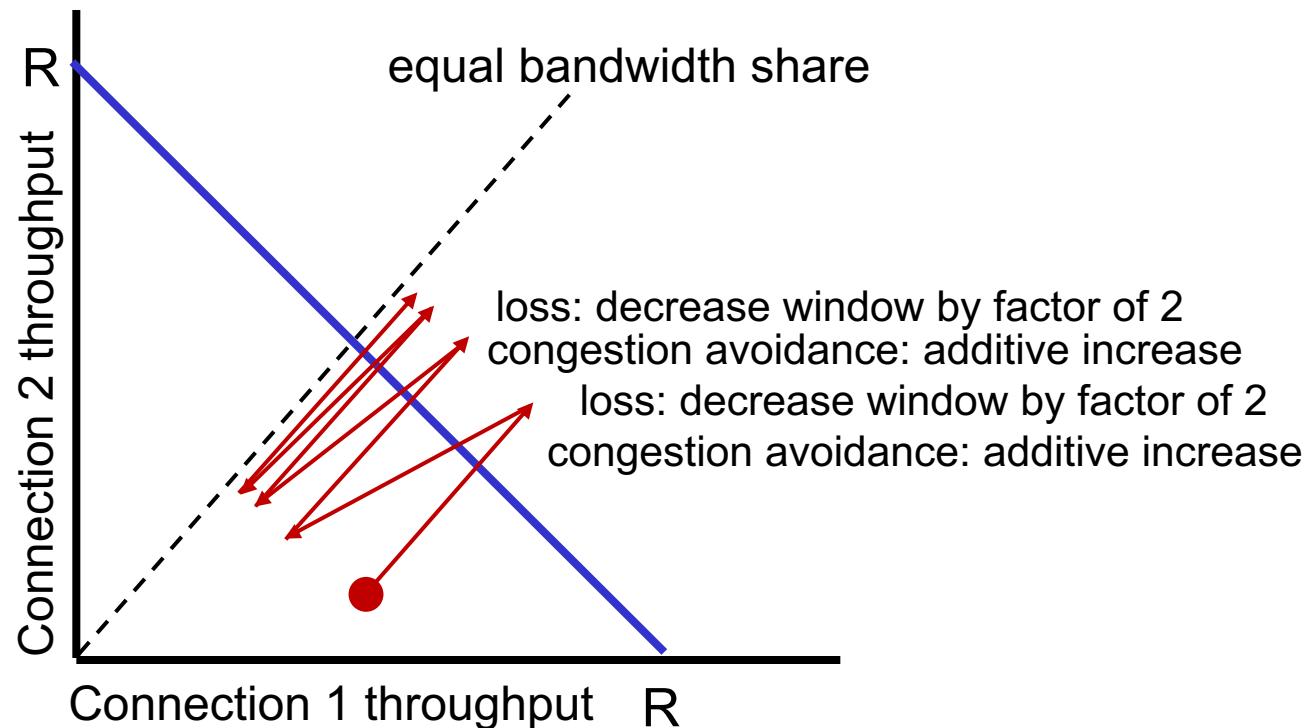
**Fairness goal:** if  $K$  TCP sessions share same bottleneck link of bandwidth  $R$ , each should have average rate of  $R/K$



# Q: is TCP Fair?

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

# Transport layer: roadmap

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



# Evolving transport-layer functionality

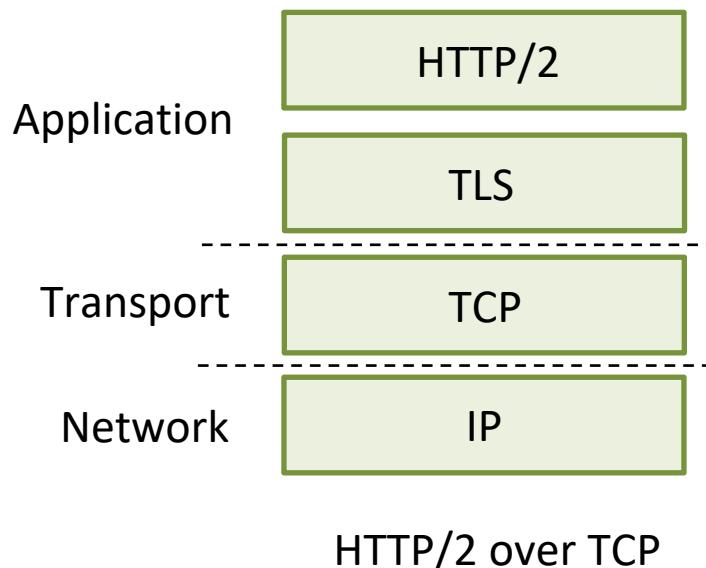
- TCP, UDP: principal transport protocols for 40 years
- different “flavors” of TCP developed, for specific scenarios:

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)

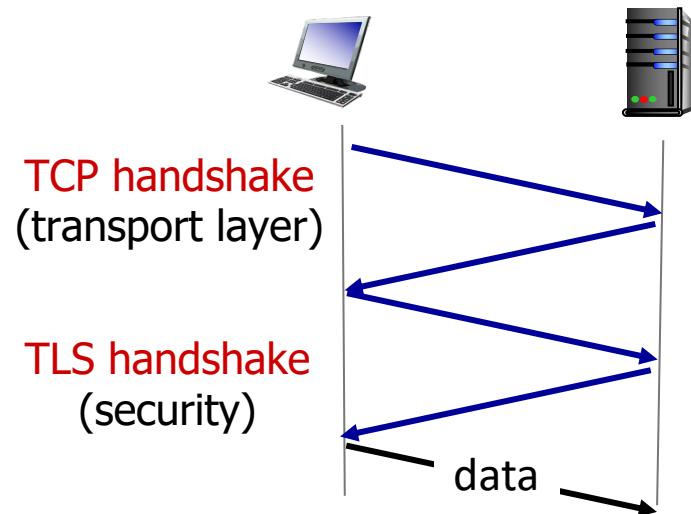


# QUIC: Quick UDP Internet Connections

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

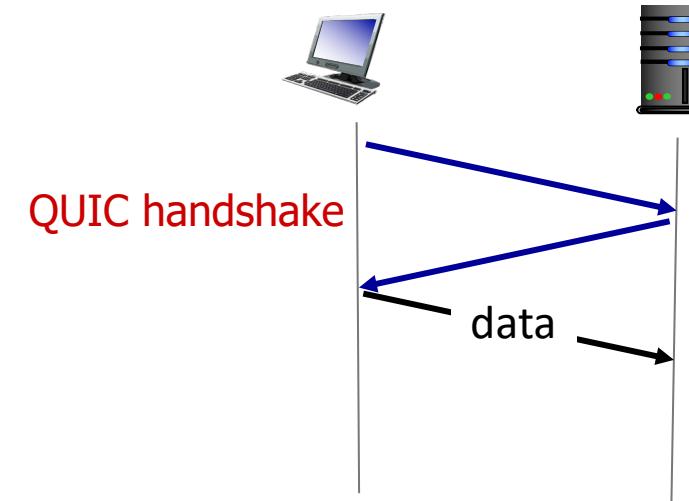
- **error and congestion control:** “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that parallel well-known TCP ones.” [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT
- multiple application-level “streams” multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

# QUIC: Connection establishment



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

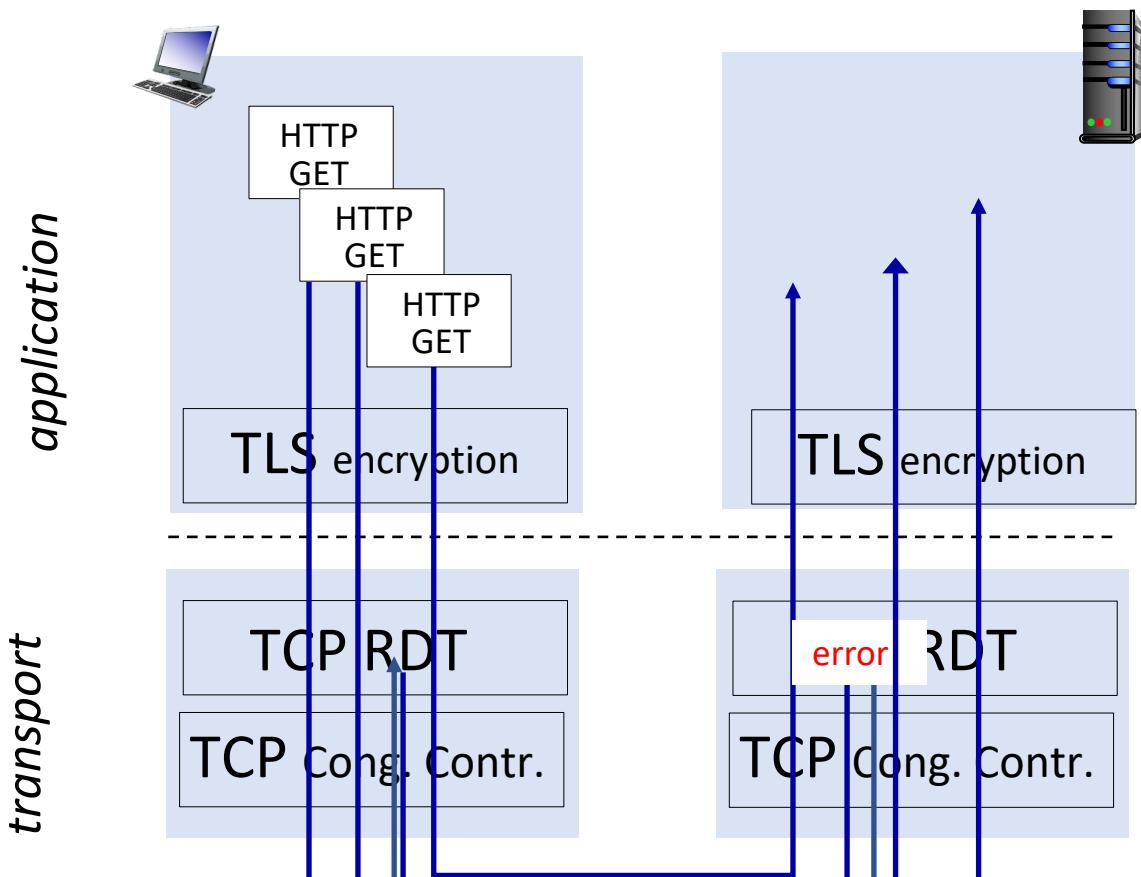
- 2 serial handshakes



QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

# QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

# Chapter 3: summary

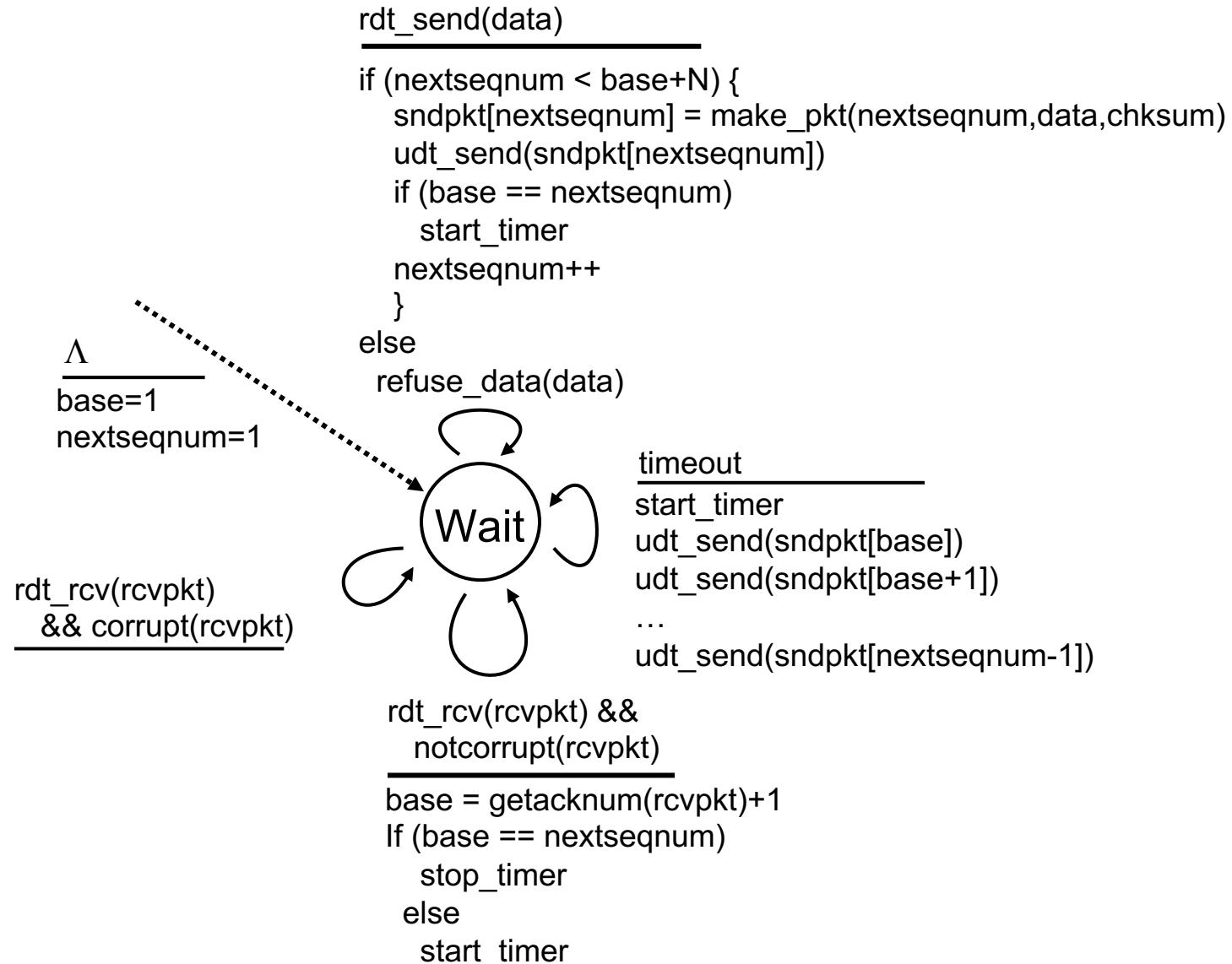
- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

## Up next:

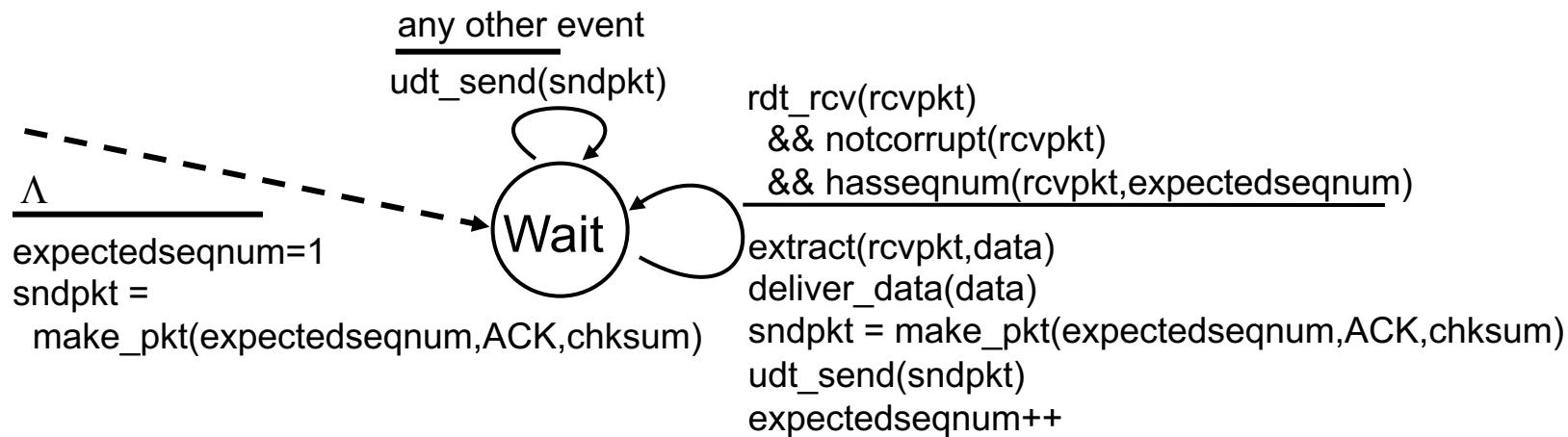
- leaving the network “edge” (application, transport layers)
- into the network “core”
- two network-layer chapters:
  - data plane
  - control plane

# Additional Chapter 3 slides

# Go-Back-N: sender extended FSM



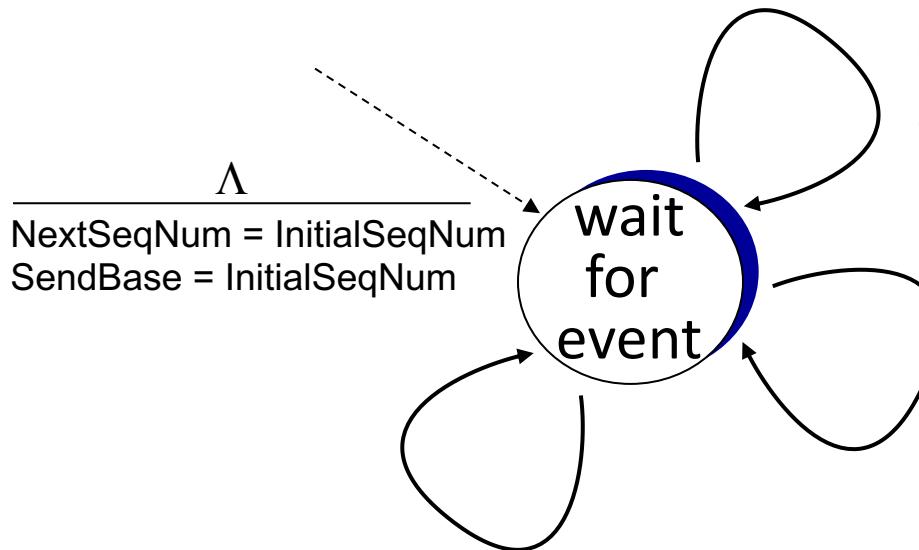
# Go-Back-N: receiver extended FSM



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)



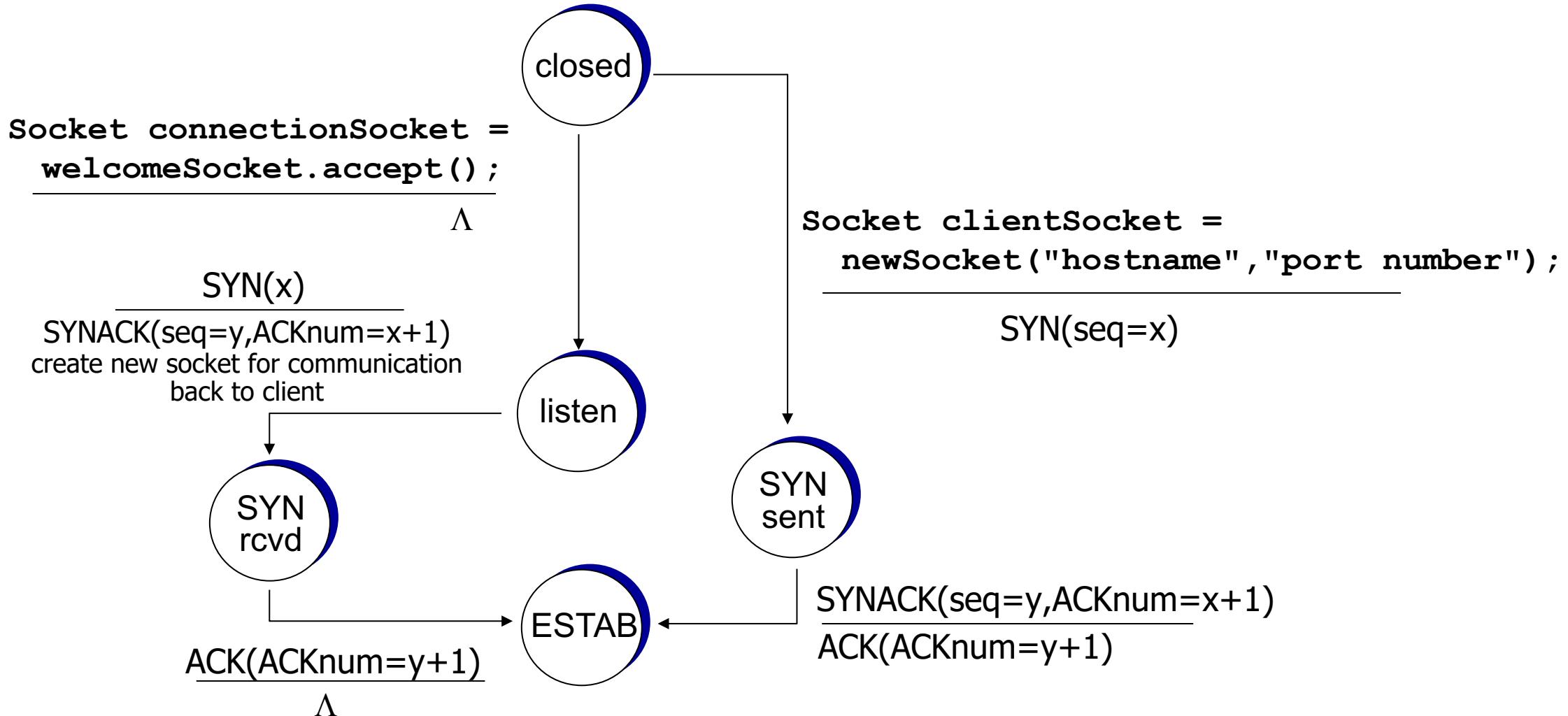
data received from application above  
create segment, seq. #: NextSeqNum  
pass segment to IP (i.e., “send”)  
NextSeqNum = NextSeqNum + length(data)  
if (timer currently not running)  
start timer

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

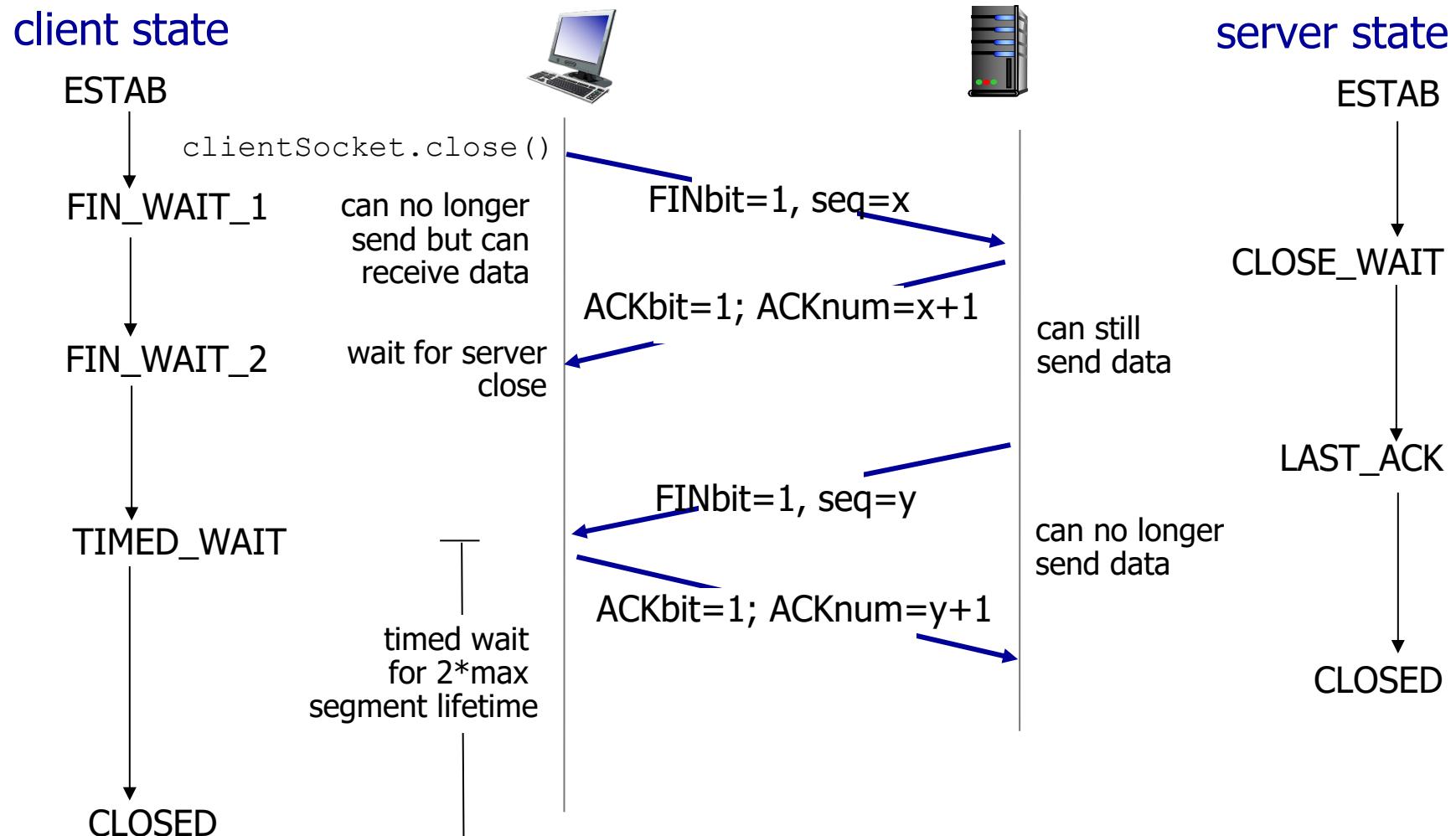
ACK received, with ACK field value y

```
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



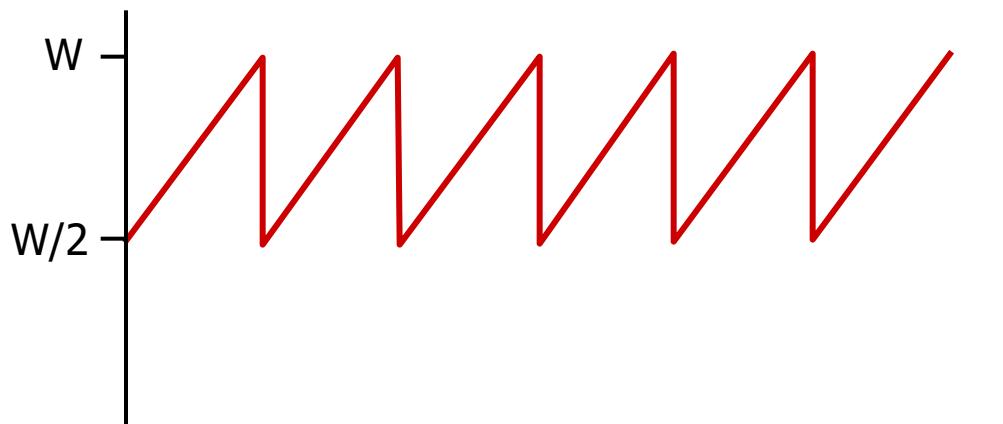
# Closing a TCP connection



# 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  $\frac{3}{4} W$
  - avg. thruput is  $\frac{3}{4}W$  per RTT

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



# 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]:

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

→ 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

# Network layer: “data plane” roadmap

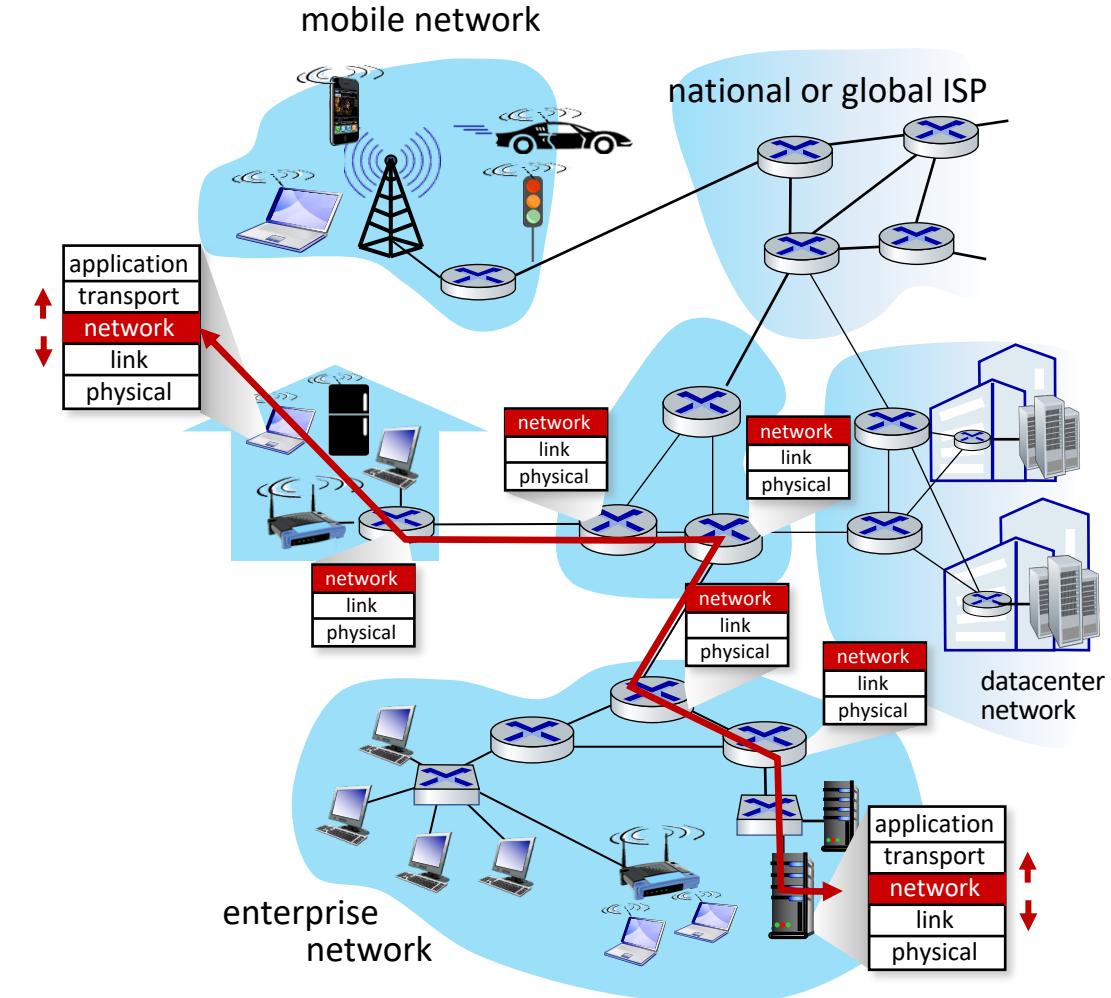
- Network layer: overview
  - data plane
  - control plane
- What's inside a router
  - input ports, switching, output ports
  - buffer management, scheduling
- IP: the Internet Protocol
  - datagram format
  - addressing
  - network address translation
  - IPv6



- Generalized Forwarding, SDN
  - Match+action
  - OpenFlow: match+action in action
- Middleboxes

# Network-layer services and protocols

- transport segment from sending to receiving host
  - **sender**: encapsulates segments into datagrams, passes to link layer
  - **receiver**: delivers segments to transport layer protocol
- network layer protocols in *every Internet device*: hosts, routers
- **routers**:
  - examines header fields in all IP datagrams passing through it
  - moves datagrams from input ports to output ports to transfer datagrams along end-end path



# Two key network-layer functions

## network-layer functions:

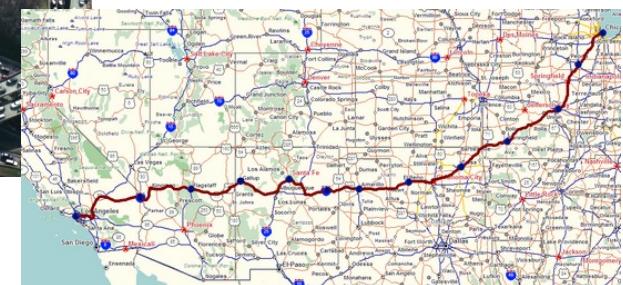
- *forwarding*: move packets from a router's input link to appropriate router output link
- *routing*: determine route taken by packets from source to destination
  - *routing algorithms*

## analogy: taking a trip

- *forwarding*: process of getting through single interchange
- *routing*: process of planning trip from source to destination



forwarding

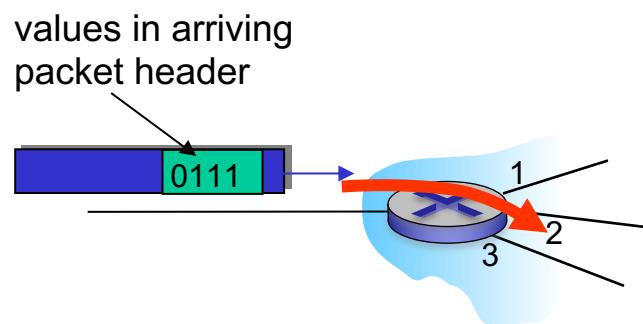


routing

# Network layer: data plane, control plane

## Data plane:

- *local*, per-router function
- determines how datagram arriving on router input port is forwarded to router output port

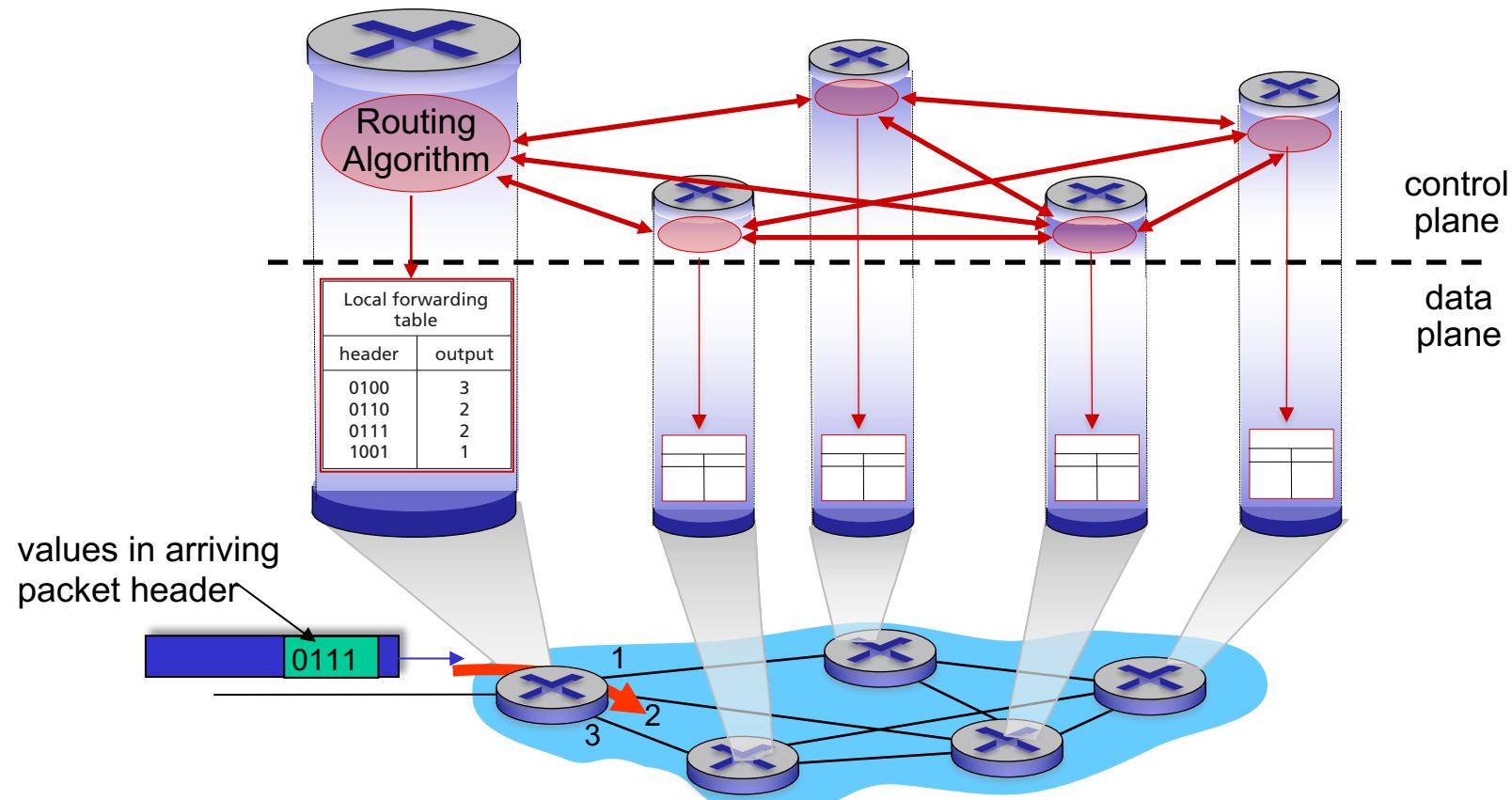


## Control plane

- *network-wide* logic
- determines how datagram is routed among routers along end-end path from source host to destination host
- two control-plane approaches:
  - *traditional routing algorithms*: implemented in routers
  - *software-defined networking (SDN)*: implemented in (remote) servers

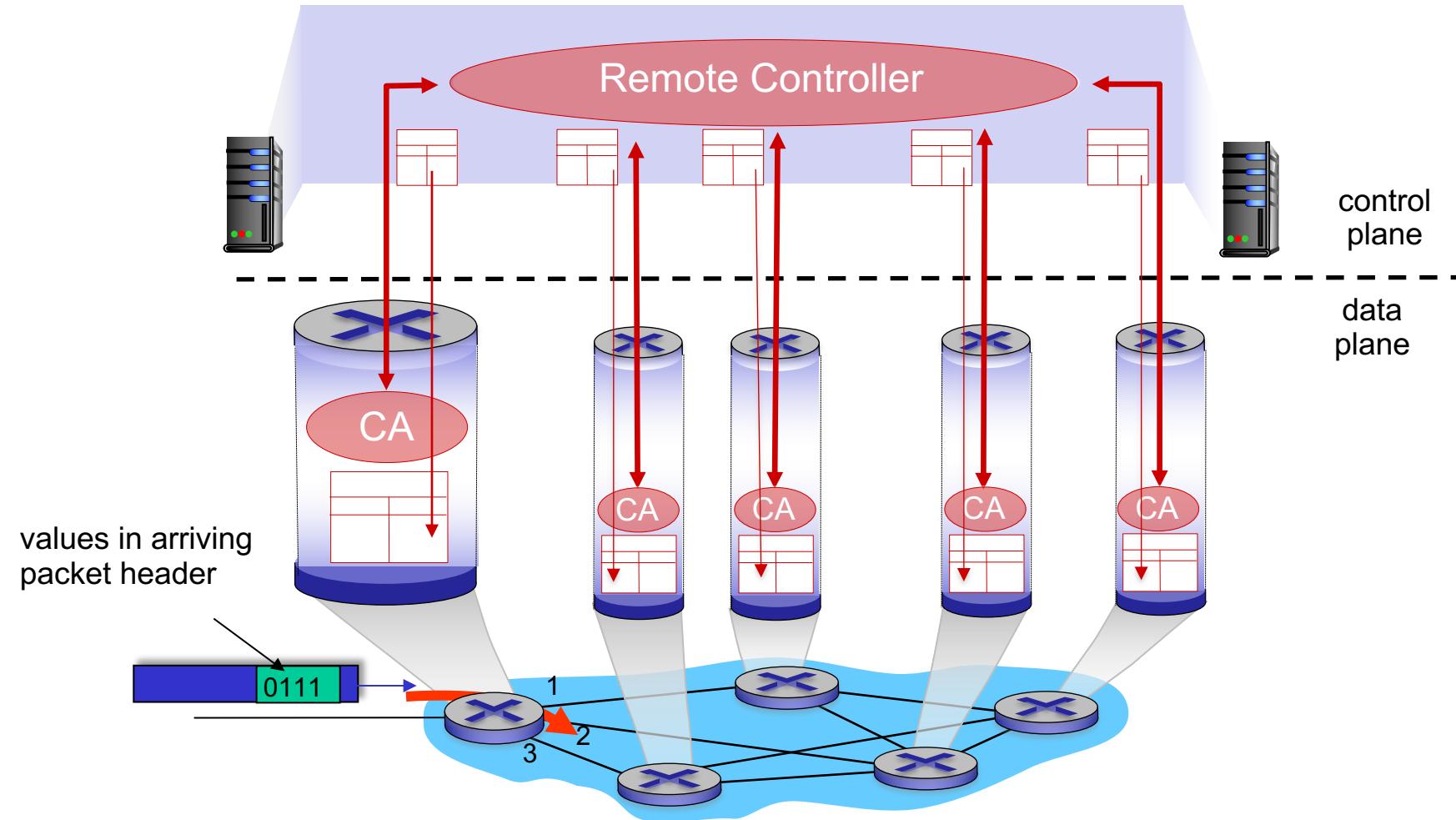
# Per-router control plane

Individual routing algorithm components *in each and every router* interact in the control plane



# Software-Defined Networking (SDN) control plane

Remote controller computes, installs forwarding tables in routers



# Network service model

*Q:* What *service model* for “channel” transporting datagrams from sender to receiver?

example services for  
*individual* datagrams:

- guaranteed delivery
- guaranteed delivery with less than 40 msec delay

example services for a *flow* of datagrams:

- in-order datagram delivery
- guaranteed minimum bandwidth to flow
- restrictions on changes in inter-packet spacing

# Network-layer service model

Network Architecture	Service Model	Quality of Service (QoS) Guarantees ?			
		Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no

Internet “best effort” service model

*No* guarantees on:

- i. successful datagram delivery to destination
- ii. timing or order of delivery
- iii. bandwidth available to end-end flow

# Network-layer service model

Network Architecture	Service Model	Quality of Service (QoS) Guarantees ?			
		Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no
ATM	Constant Bit Rate	Constant rate	yes	yes	yes
ATM	Available Bit Rate	Guaranteed min	no	yes	no
Internet	Intserv Guaranteed (RFC 1633)	yes	yes	yes	yes
Internet	Diffserv (RFC 2475)	possible	possibly	possibly	no

# Reflections on best-effort service:

- simplicity of mechanism has allowed Internet to be widely deployed adopted
- sufficient provisioning of bandwidth allows performance of real-time applications (e.g., interactive voice, video) to be “good enough” for “most of the time”
- replicated, application-layer distributed services (datacenters, content distribution networks) connecting close to clients’ networks, allow services to be provided from multiple locations
- congestion control of “elastic” services helps

*It's hard to argue with success of best-effort service model*

# Network layer: “data plane” roadmap

- Network layer: overview
  - data plane
  - control plane

- What's inside a router
  - input ports, switching, output ports
  - buffer management, scheduling

- IP: the Internet Protocol
  - datagram format
  - addressing
  - network address translation
  - IPv6

- Generalized Forwarding, SDN
  - Match+action
  - OpenFlow: match+action in action
- Middleboxes



# Longest prefix matching

## longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*** *****	0
11001000 00010111 00011000 *****	1
11001000 00010111 00011*** *****	2
otherwise	3

examples:

11001000 00010111 00010110 10100001 which interface?  
11001000 00010111 00011000 10101010 which interface?

# Longest prefix matching

longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range	Link interface
11001000 00010111 00010*****	0
11001000 00010111 00011000 *****	1
11001000 1 00011*** *****	2
otherwise	3

examples:

11001000 00010111 00010110 10100001 which interface?

11001000 00010111 00011000 10101010 which interface?

# Longest prefix matching

## longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range					Link interface
11001000	00010111	00010***	*****	*	0
11001000	00010111	00011000	*****	*	1
11001000	00010111	00011***	*****	*	2
otherwise					3

match!

examples:

11001000	00010111	00010110	10100001	which interface?
11001000	00010111	00011000	10101010	which interface?

# Longest prefix matching

## longest prefix match

when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

Destination Address Range					Link interface
11001000	00010111	00010***	*****	*	0
11001000	00010111	00011000	*****	*	1
11001000	00010111	00011***	*****	*	2
otherwise					3

match!

examples:

11001000	00010111	00010110	10100001	which interface?
11001000	00010111	00011000	10101010	which interface?

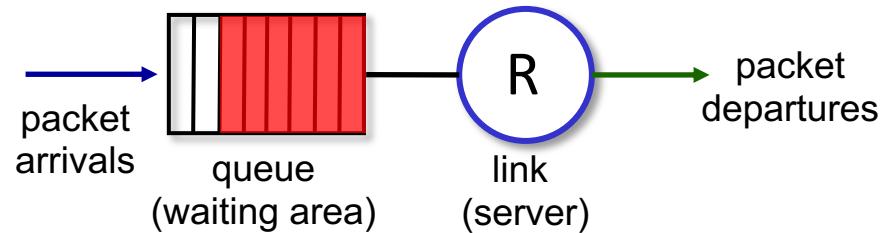
# How much buffering?

- RFC 3439 rule of thumb: average buffering equal to “typical” RTT (say 250 msec) times link capacity C
  - e.g.,  $C = 10 \text{ Gbps}$  link: 2.5 Gbit buffer
- more recent recommendation: with  $N$  flows, buffering equal to

$$\frac{\text{RTT} \cdot C}{\sqrt{N}}$$

- but *too* much buffering can increase delays (particularly in home routers)
  - long RTTs: poor performance for realtime apps, sluggish TCP response
  - recall delay-based congestion control: “keep bottleneck link just full enough (busy) but no fuller”

# Buffer Management



## buffer management:

- **drop:** which packet to add, drop when buffers are full
  - **tail drop:** drop arriving packet
  - **priority:** drop/remove on priority basis
- **marking:** which packets to mark to signal congestion (ECN, RED)

# Packet Scheduling: FCFS

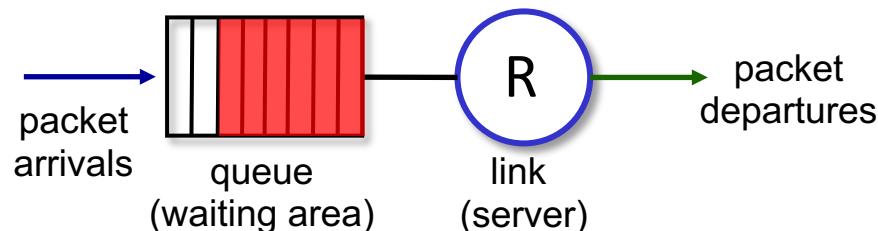
**packet scheduling:** deciding which packet to send next on link

- first come, first served
- priority
- round robin
- weighted fair queueing

**FCFS:** packets transmitted in order of arrival to output port

- also known as: First-in-first-out (FIFO)
- real world examples?

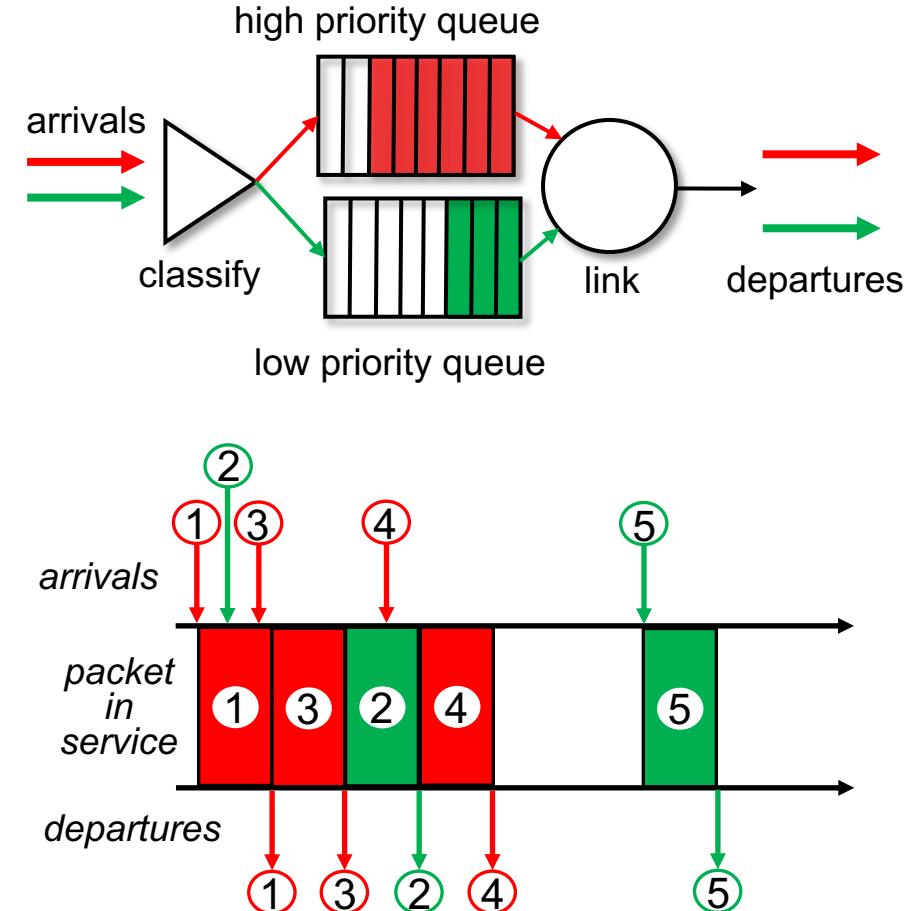
Abstraction: queue



# Scheduling policies: priority

## *Priority scheduling:*

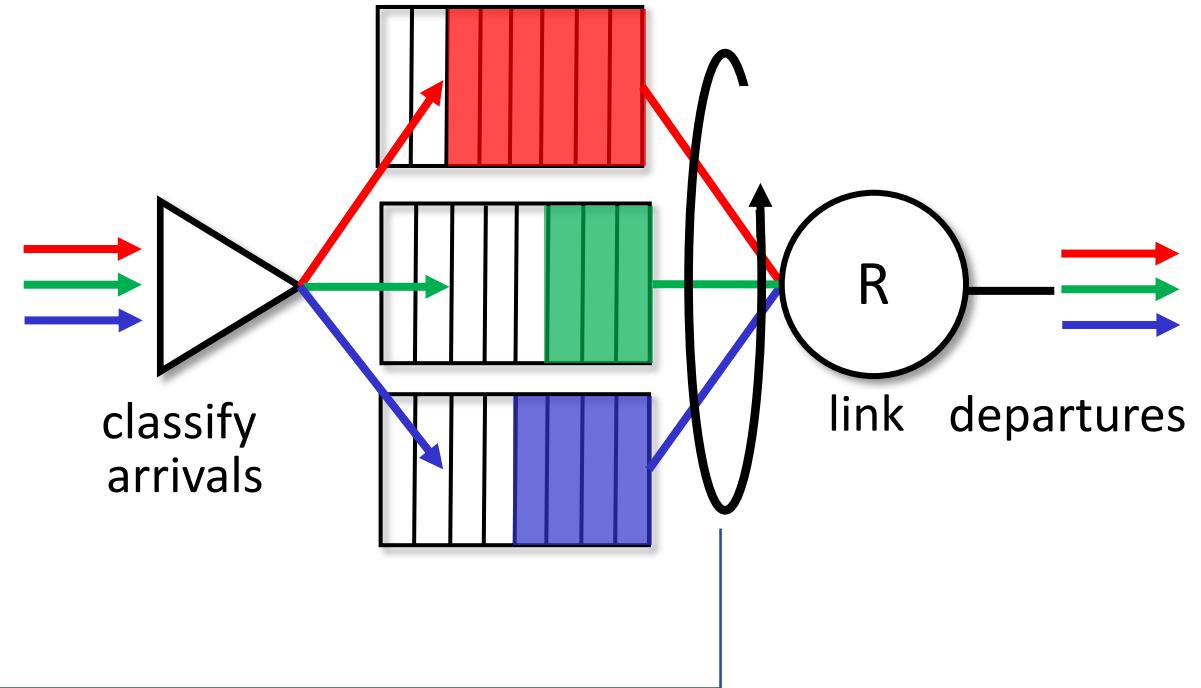
- arriving traffic classified, queued by class
  - any header fields can be used for classification
- send packet from highest priority queue that has buffered packets
  - FCFS within priority class



# Scheduling policies: round robin

## *Round Robin (RR) scheduling:*

- arriving traffic classified, queued by class
  - any header fields can be used for classification
- server cyclically, repeatedly scans class queues, sending one complete packet from each class (if available) in turn



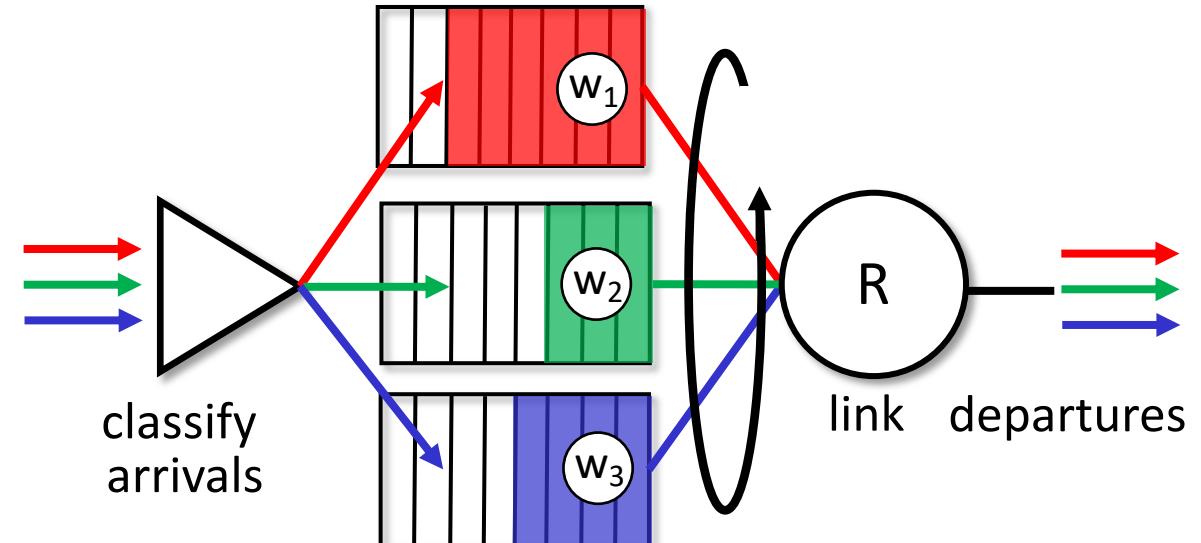
# Scheduling policies: weighted fair queueing

## *Weighted Fair Queueing (WFQ):*

- generalized Round Robin
- each class,  $i$ , has weight,  $w_i$ , and gets weighted amount of service in each cycle:

$$\frac{w_i}{\sum_j w_j}$$

- minimum bandwidth guarantee (per-traffic-class)



# Sidebar: Network Neutrality

What is network neutrality?

- *technical*: how an ISP should share/allocation its resources
  - packet scheduling, buffer management are the *mechanisms*
- *social, economic* principles
  - protecting free speech
  - encouraging innovation, competition
- enforced *legal* rules and policies

*Different countries have different “takes” on network neutrality*

# Network layer: “data plane” roadmap

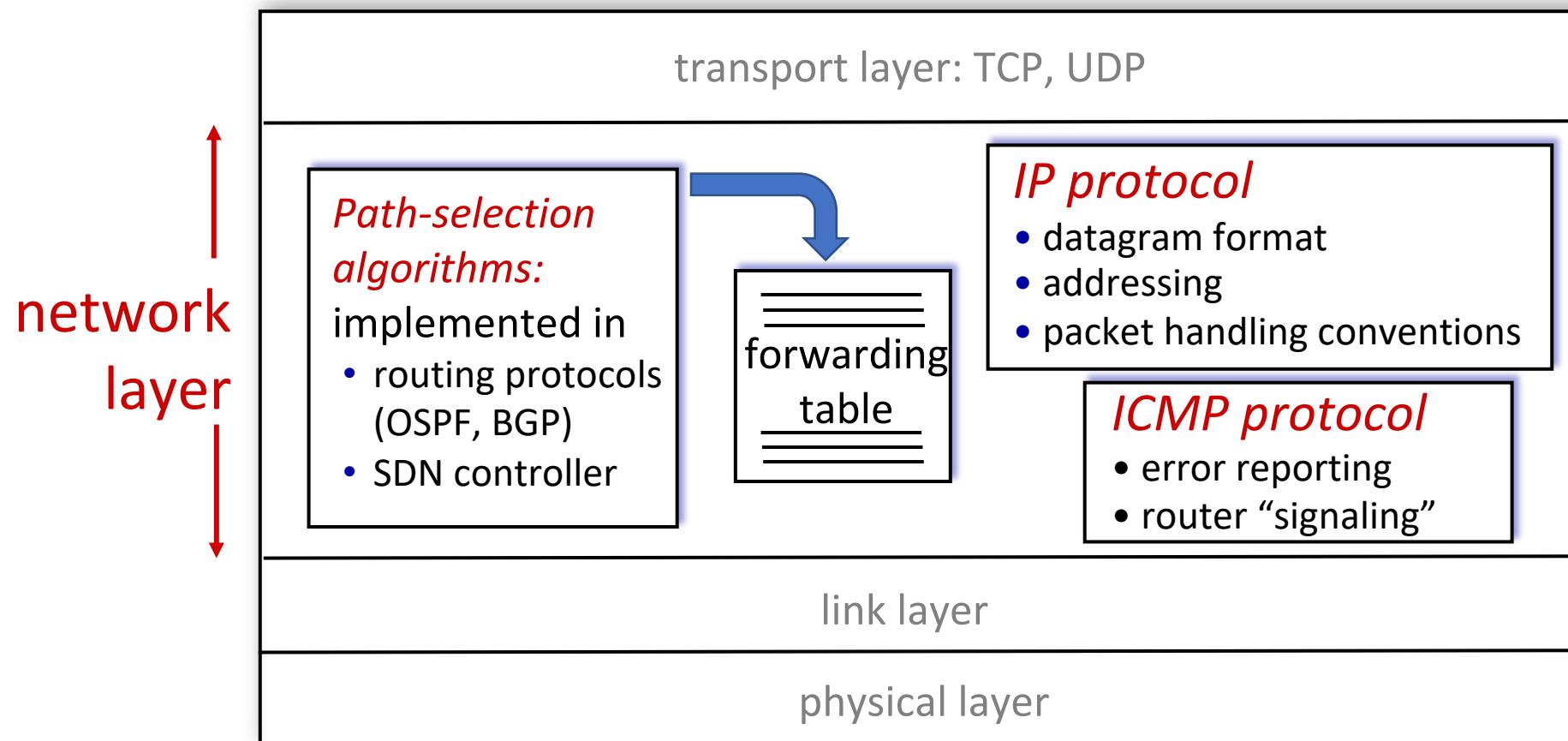
- Network layer: overview
  - data plane
  - control plane
- What's inside a router
  - input ports, switching, output ports
  - buffer management, scheduling
- IP: the Internet Protocol
  - datagram format
  - addressing
  - network address translation
  - IPv6



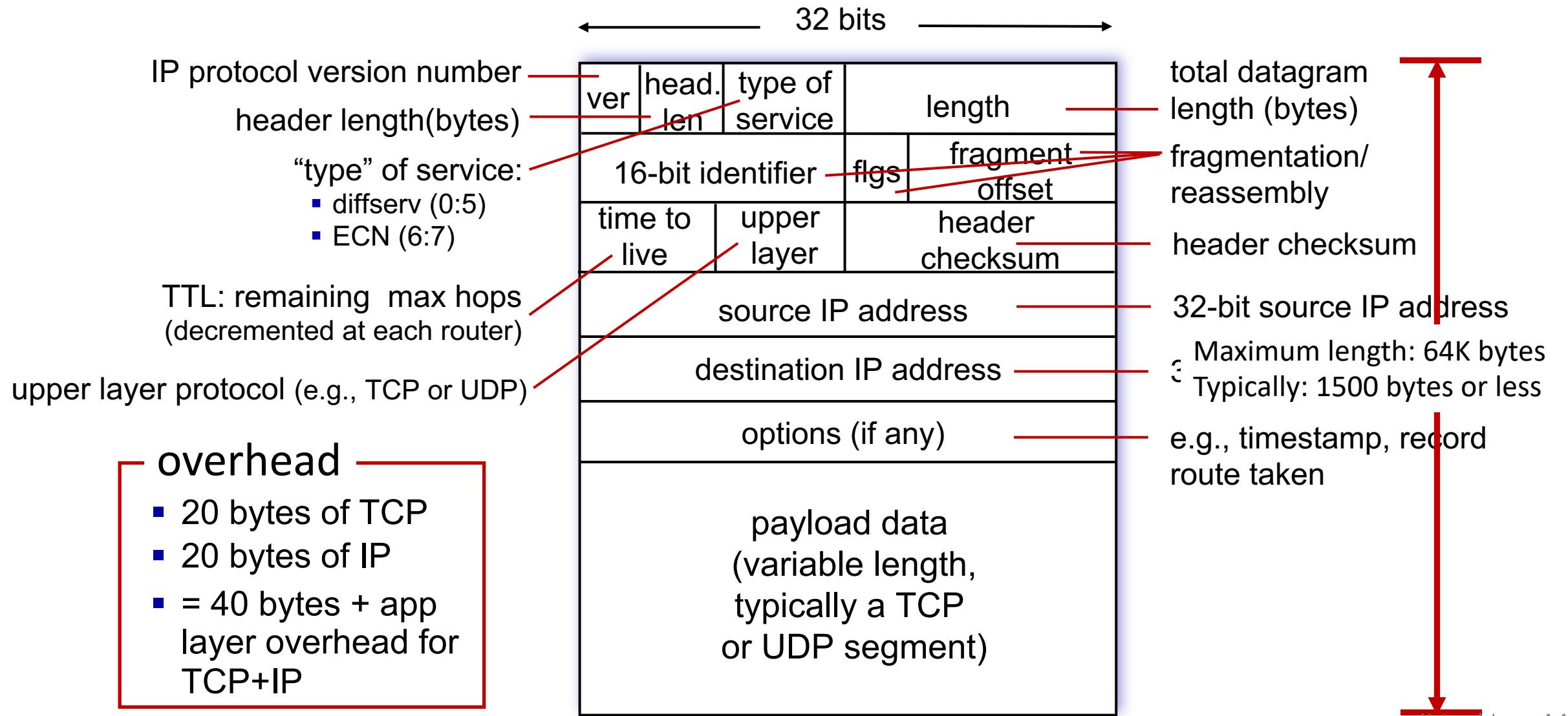
- Generalized Forwarding, SDN
  - match+action
  - OpenFlow: match+action in action
- Middleboxes

# Network Layer: Internet

host, router network layer functions:

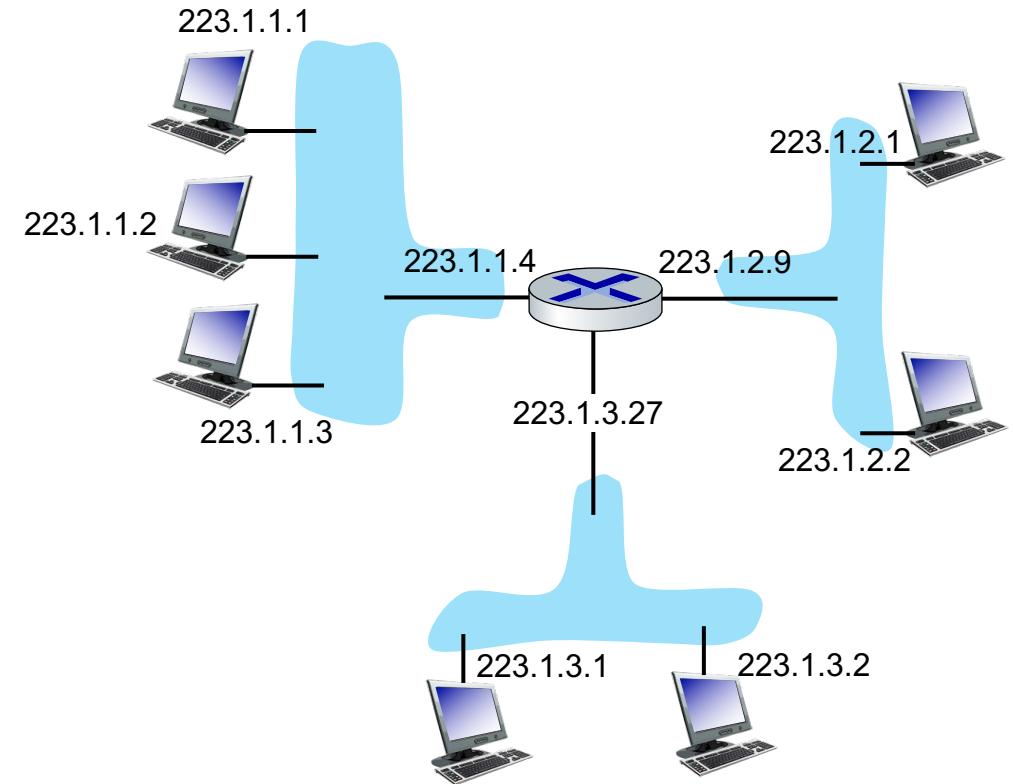


# IP Datagram format



# 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)



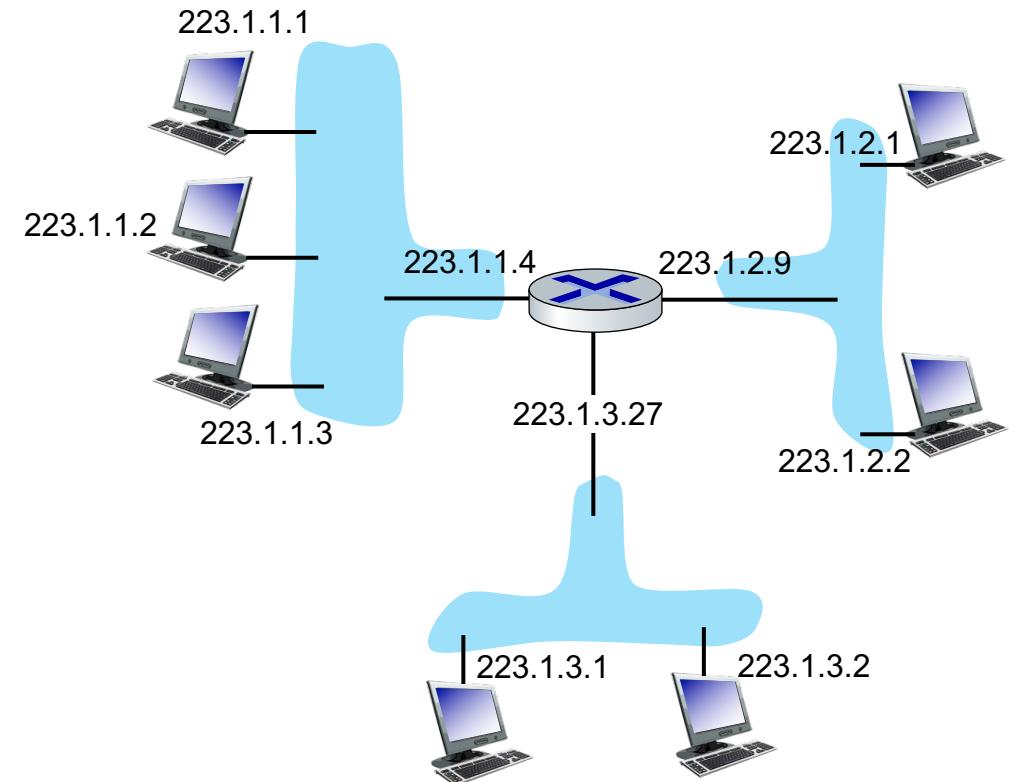
dotted-decimal IP address notation:

223.1.1.1 = 

11011111	00000001	00000001	00000001
----------	----------	----------	----------

# 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)



dotted-decimal IP address notation:

223.1.1.1 = 11011111 00000001 00000001 00000001

                |                |                |                |  
                223            1            1            1

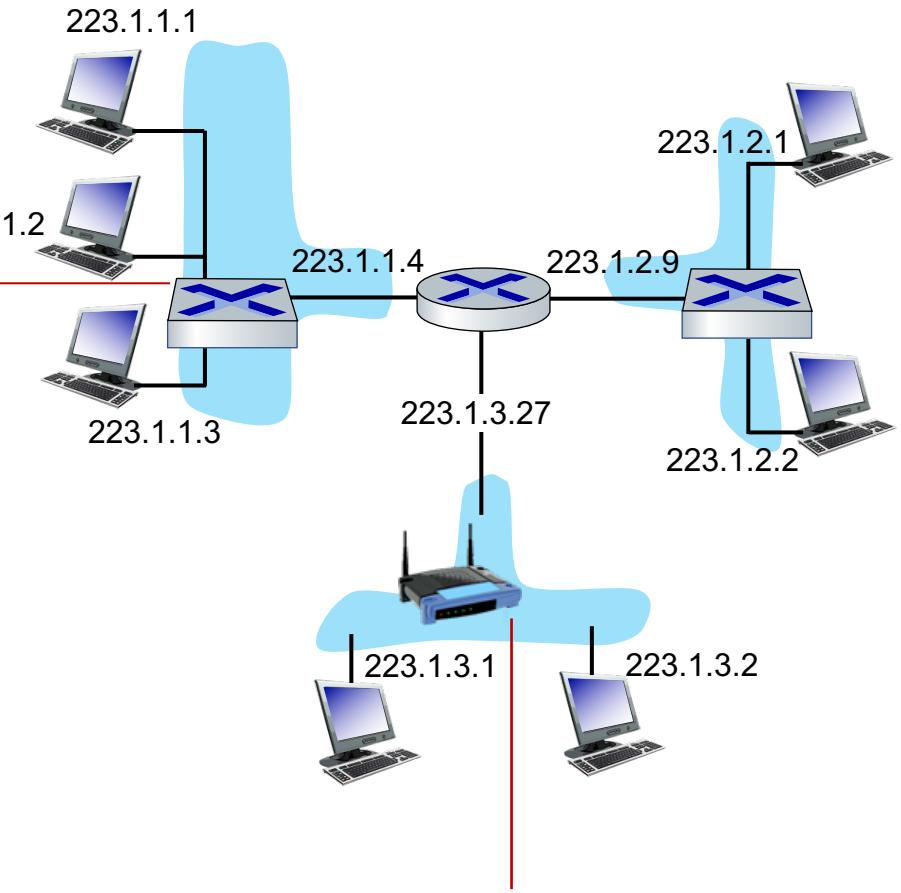
# IP addressing: introduction

**Q:** how are interfaces actually connected?

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

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

**A:** wired Ethernet interfaces connected by Ethernet switches



**A:** wireless WiFi interfaces connected by WiFi base station

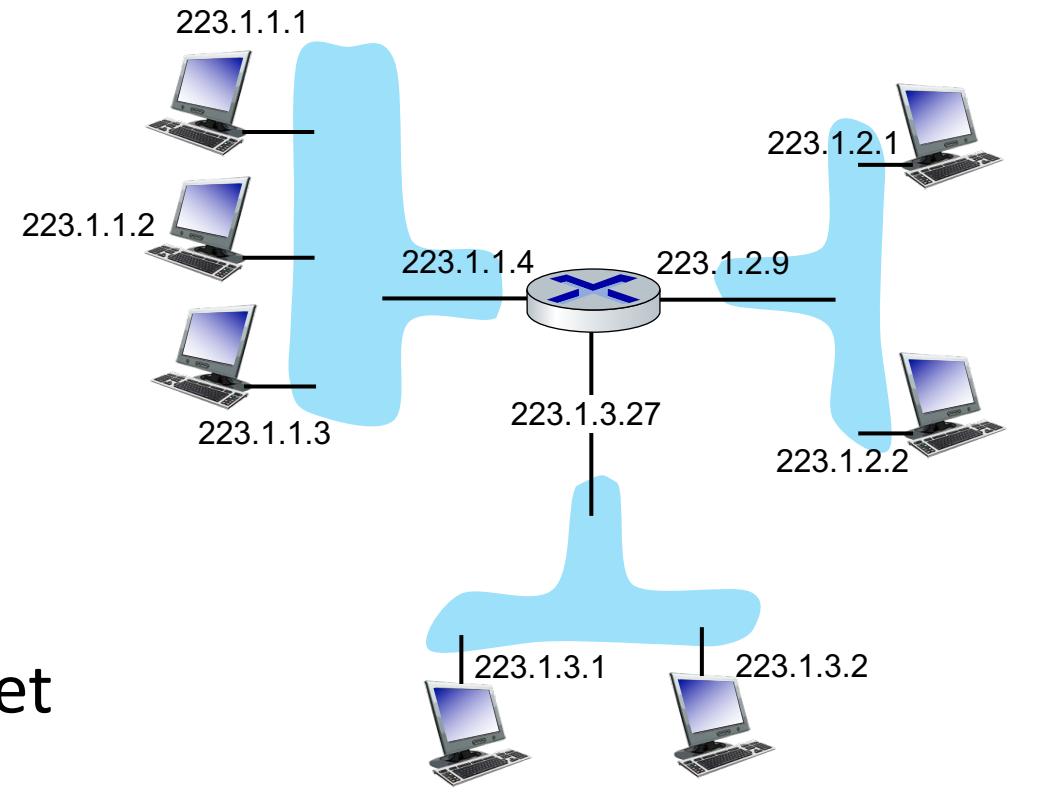
# Subnets

- *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

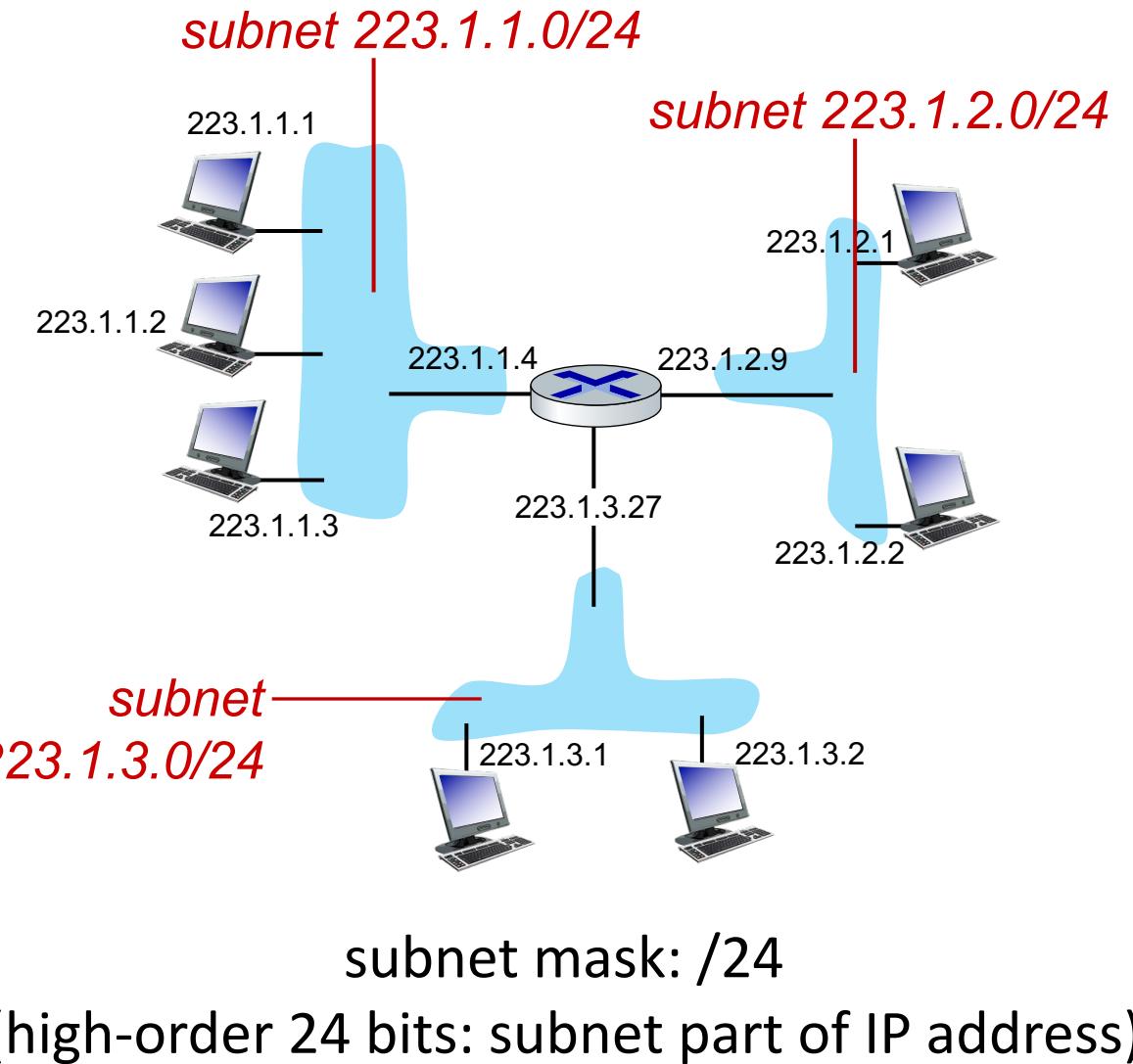


network consisting of 3 subnets

# Subnets

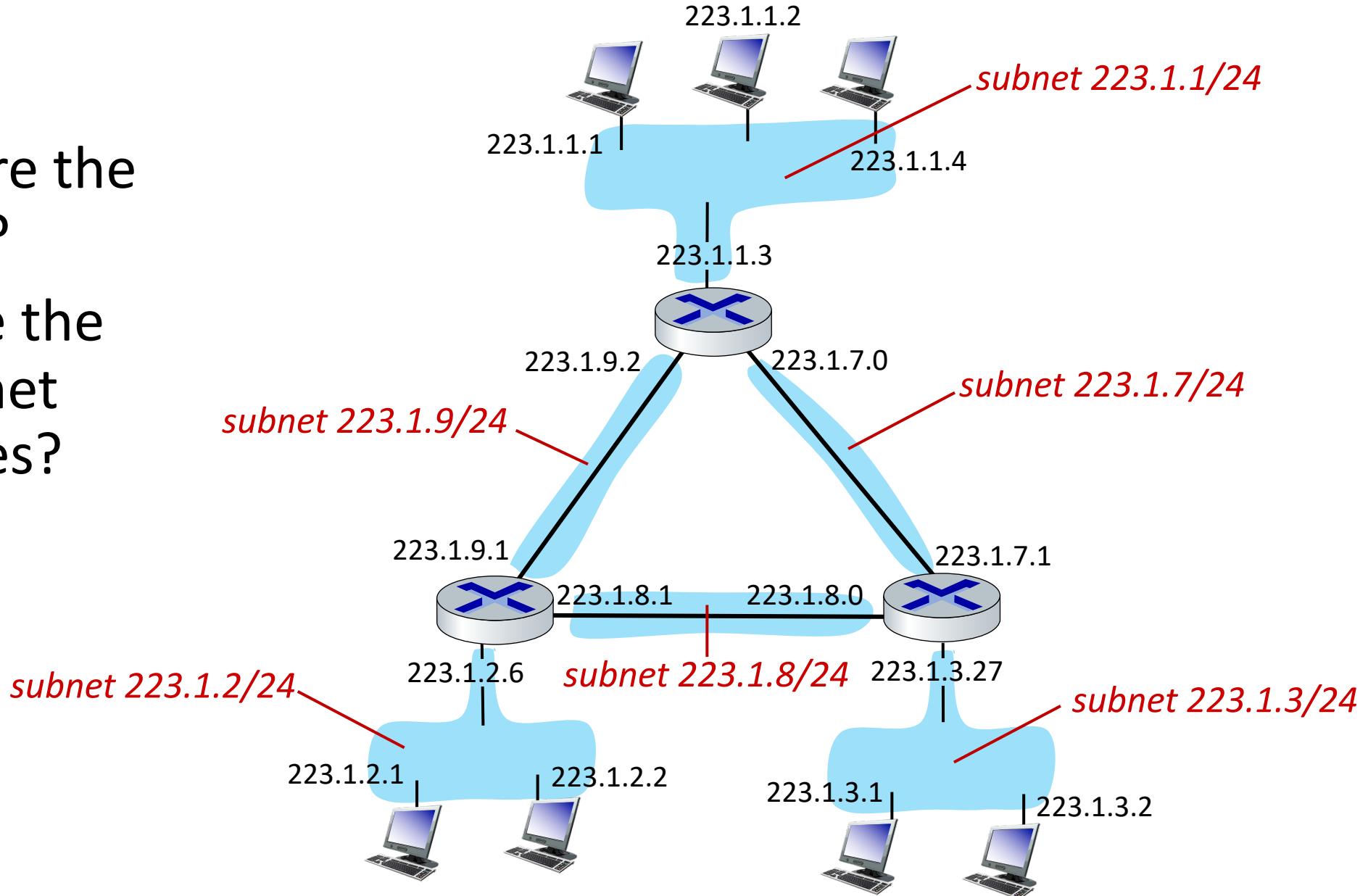
*Recipe for defining subnets:*

- detach each interface from its host or router, creating “islands” of isolated networks
- each isolated network is called a *subnet*



# 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



- <https://en.wikipedia.org/wiki/IPv4#Addressing>

# 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 server
  - “plug-and-play”

# DHCP: Dynamic Host Configuration Protocol

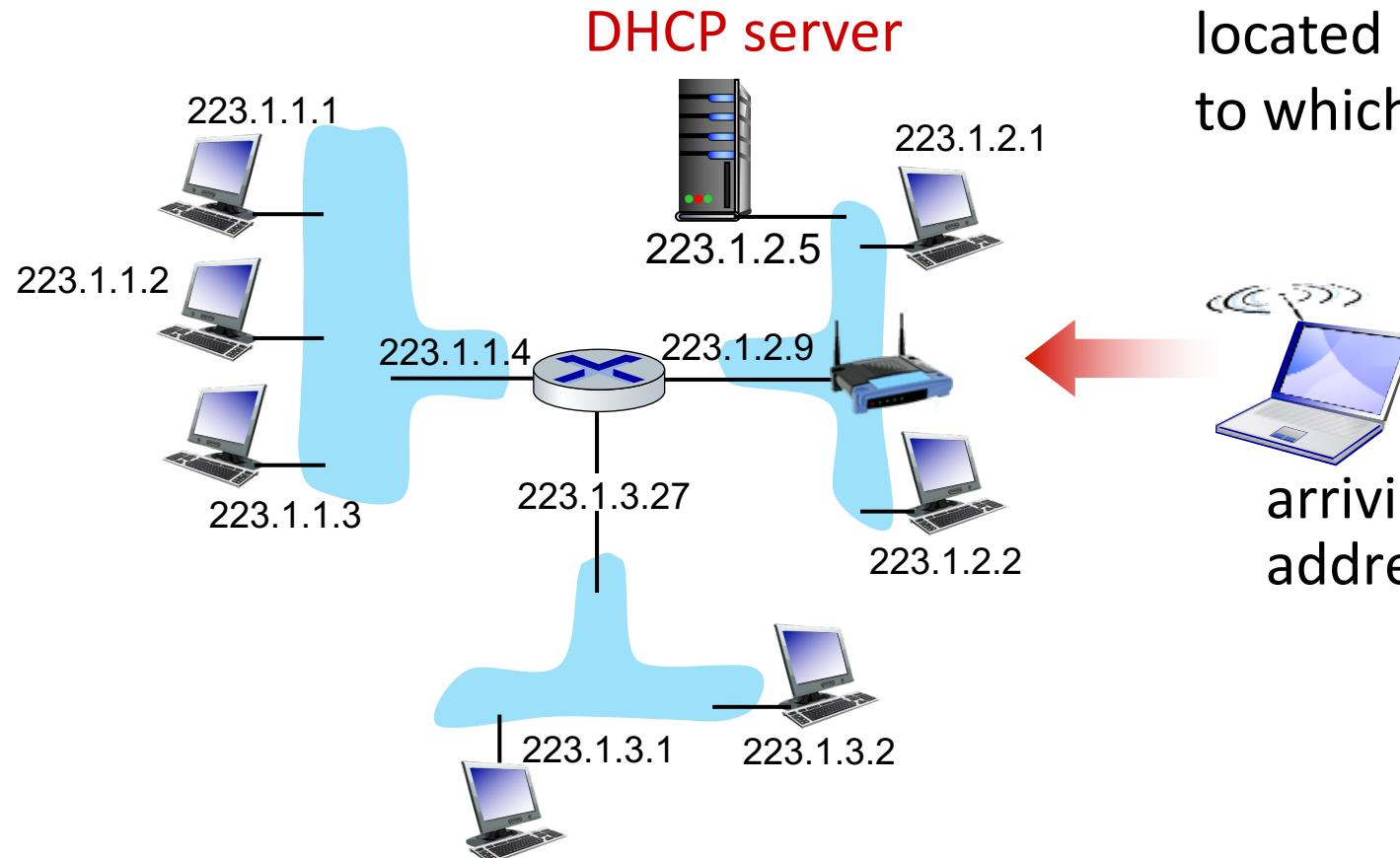
**goal:** host *dynamically* obtains IP address from network server when it “joins” network

- 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



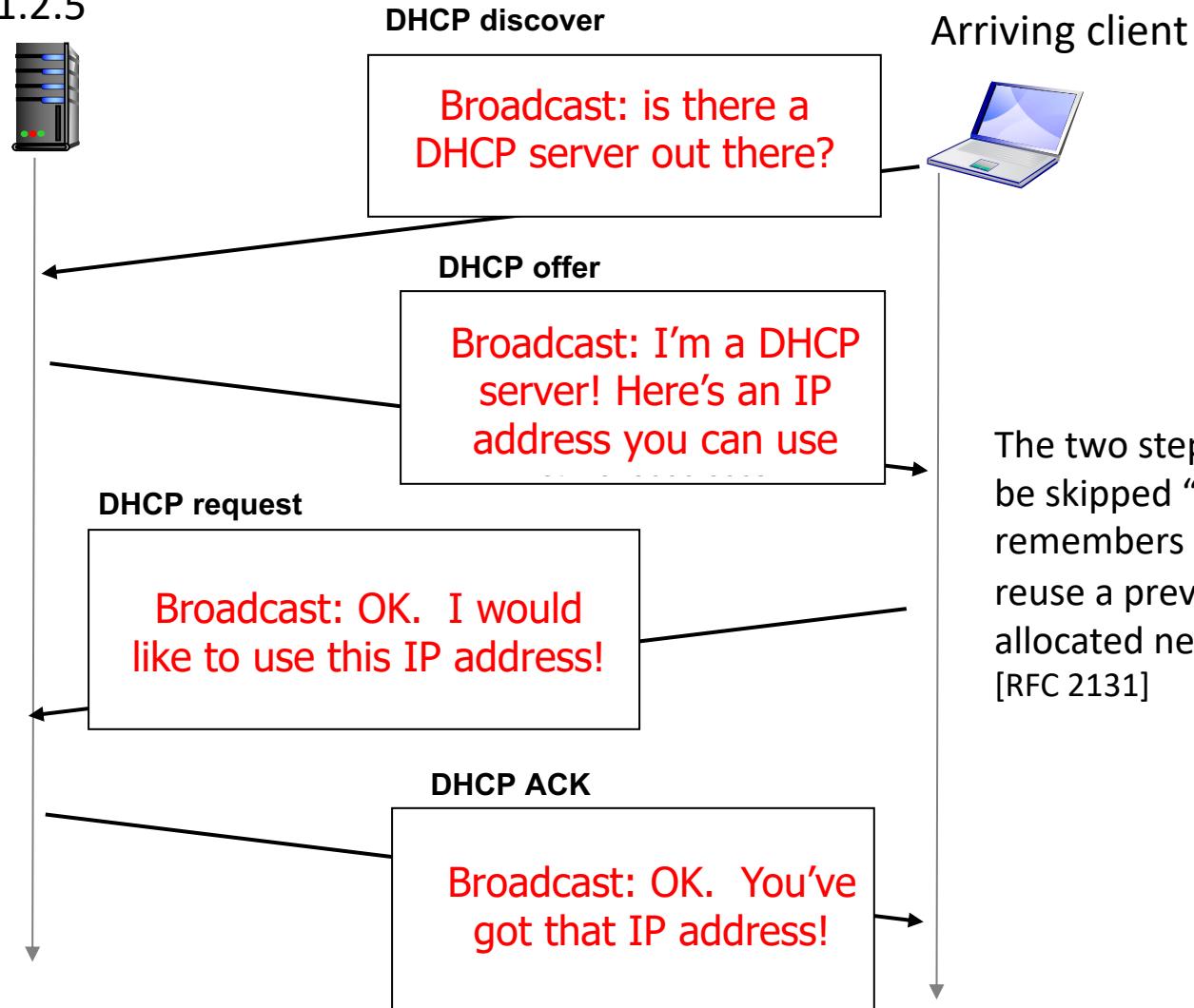
Typically, DHCP server will be co-located in router, serving all subnets to which router is attached



arriving **DHCP client** needs address in this network

# DHCP client-server scenario

DHCP server: 223.1.2.5

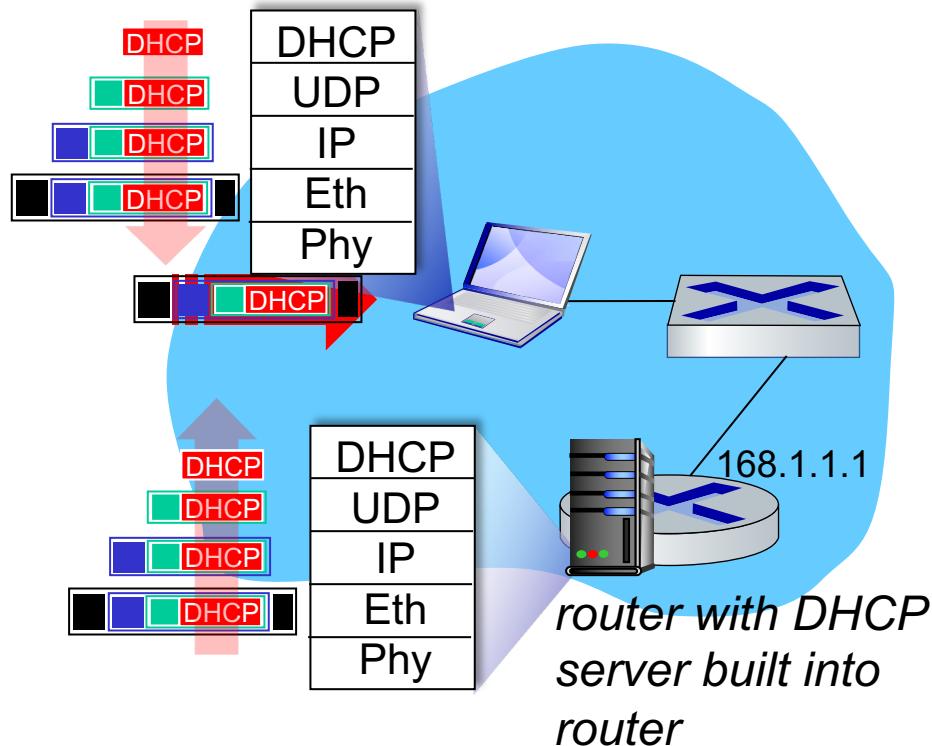


# DHCP: more than IP addresses

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

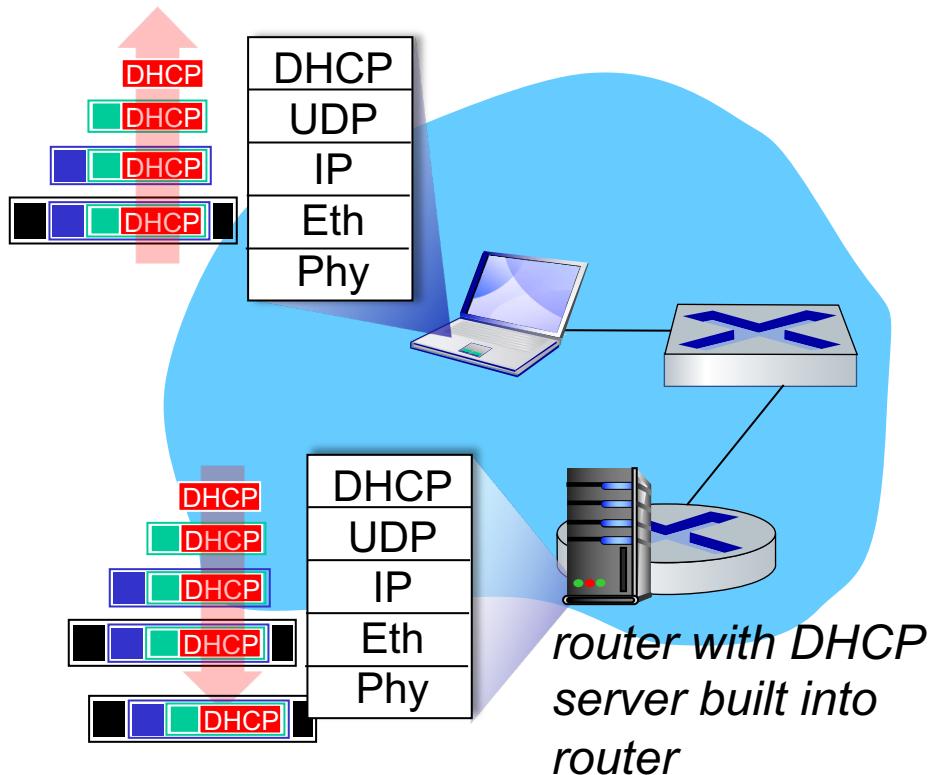
- address of first-hop router for client
- name and IP address of DNS sever
- network mask (indicating network versus host portion of address)

# 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



- DHCP 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

# IP addresses: how to get one?

**Q:** how does *network* get subnet part of IP address?

**A:** gets allocated portion of its provider ISP's address space

ISP's block	<u>11001000</u>	<u>00010111</u>	<u>00010000</u>	<u>00000000</u>	200.23.16.0/20
-------------	-----------------	-----------------	-----------------	-----------------	----------------

ISP can then allocate out its address space in 8 blocks:

Organization 0	<u>11001000</u>	<u>00010111</u>	<u>00010000</u>	<u>00000000</u>	200.23.16.0/23
----------------	-----------------	-----------------	-----------------	-----------------	----------------

Organization 1	<u>11001000</u>	<u>00010111</u>	<u>00010010</u>	<u>00000000</u>	200.23.18.0/23
----------------	-----------------	-----------------	-----------------	-----------------	----------------

Organization 2	<u>11001000</u>	<u>00010111</u>	<u>00010100</u>	<u>00000000</u>	200.23.20.0/23
----------------	-----------------	-----------------	-----------------	-----------------	----------------

...

.....

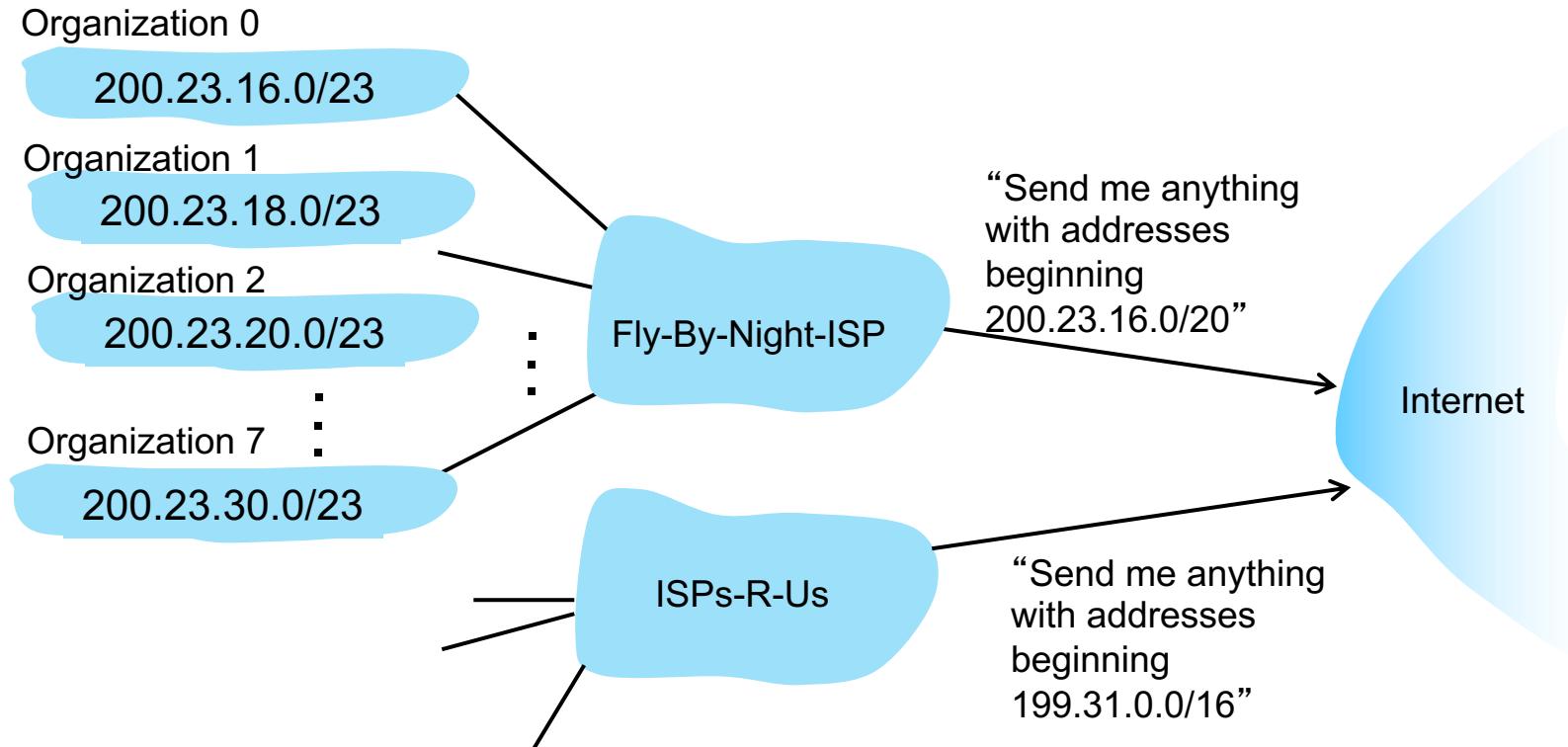
.....

.....

Organization 7	<u>11001000</u>	<u>00010111</u>	<u>00011110</u>	<u>00000000</u>	200.23.30.0/23
----------------	-----------------	-----------------	-----------------	-----------------	----------------

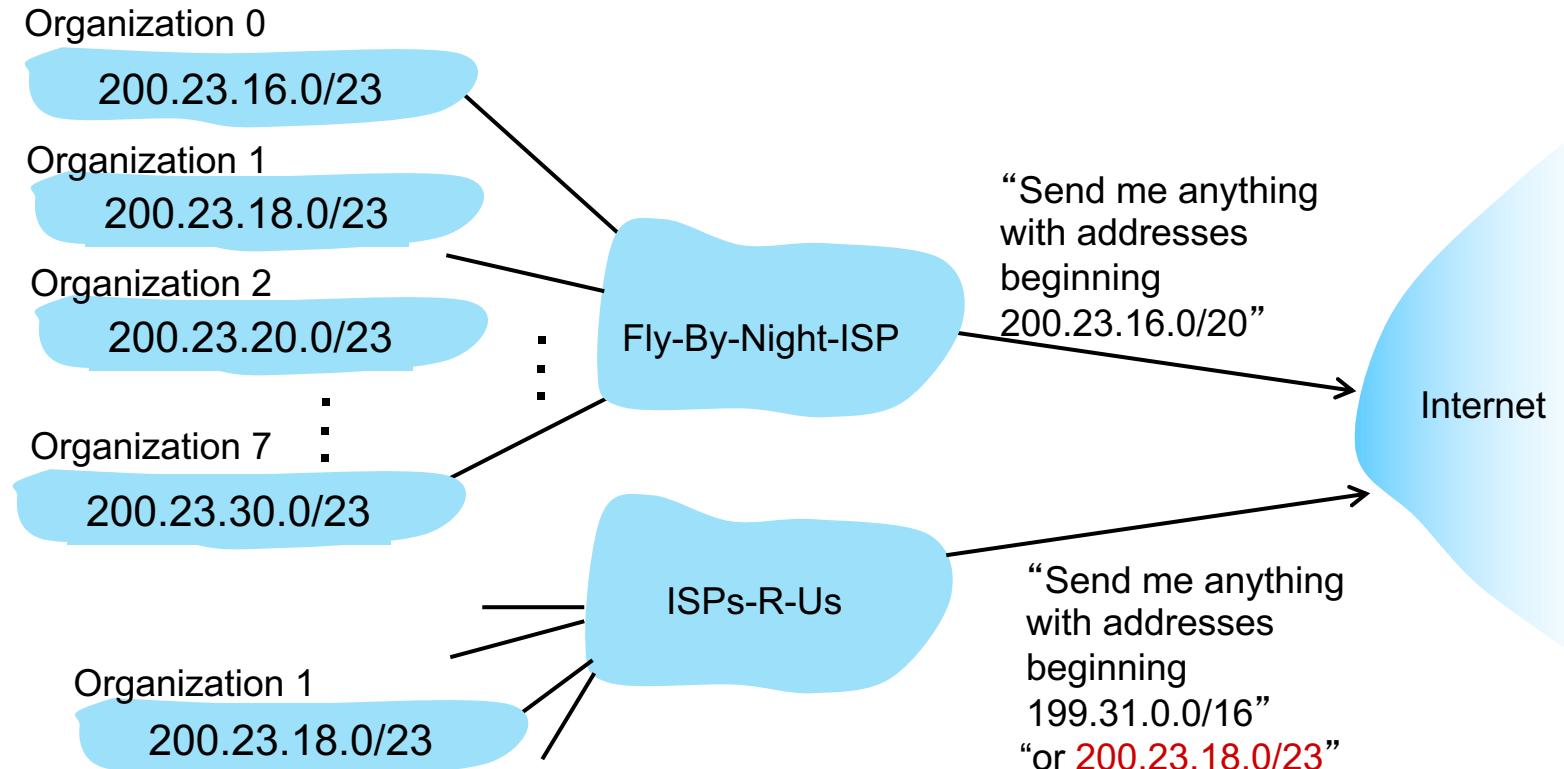
# Hierarchical addressing: route aggregation

hierarchical addressing allows efficient advertisement of routing information:



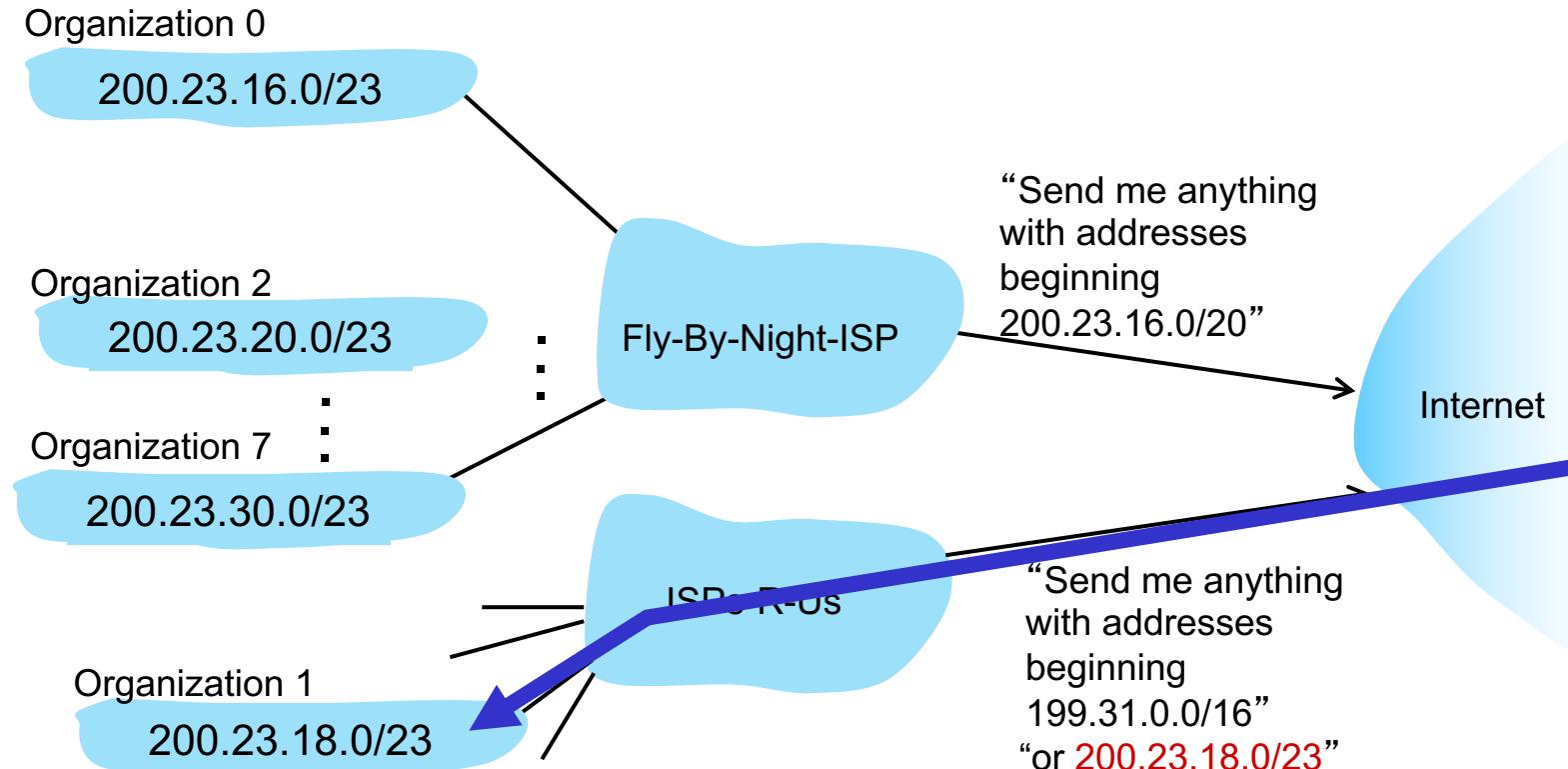
# Hierarchical addressing: more specific routes

- Organization 1 moves from Fly-By-Night-ISP to ISPs-R-Us
- ISPs-R-Us now advertises a more specific route to Organization 1



# Hierarchical addressing: more specific routes

- Organization 1 moves from Fly-By-Night-ISP to ISPs-R-Us
- ISPs-R-Us now advertises a more specific route to Organization 1



# IP addressing: last words ...

**Q:** how does an ISP get block of addresses?

**A:** ICANN: Internet Corporation for Assigned Names and Numbers

<http://www.icann.org/>

- allocates IP addresses, through 5 regional registries (RRs) (who may then allocate to local registries)
- manages DNS root zone, including delegation of individual TLD (.com, .edu , ...) management

**Q:** are there enough 32-bit IP addresses?

- ICANN allocated last chunk of IPv4 addresses to RRs in 2011
- NAT (next) helps IPv4 address space exhaustion
- IPv6 has 128-bit address space

"Who the [heck] knew how much address space we needed?" Vint Cerf (reflecting on decision to make IPv4 address 32 bits long)

# Network layer: “data plane” roadmap

- Network layer: overview
  - data plane
  - control plane

- What's inside a router
  - input ports, switching, output ports
  - buffer management, scheduling

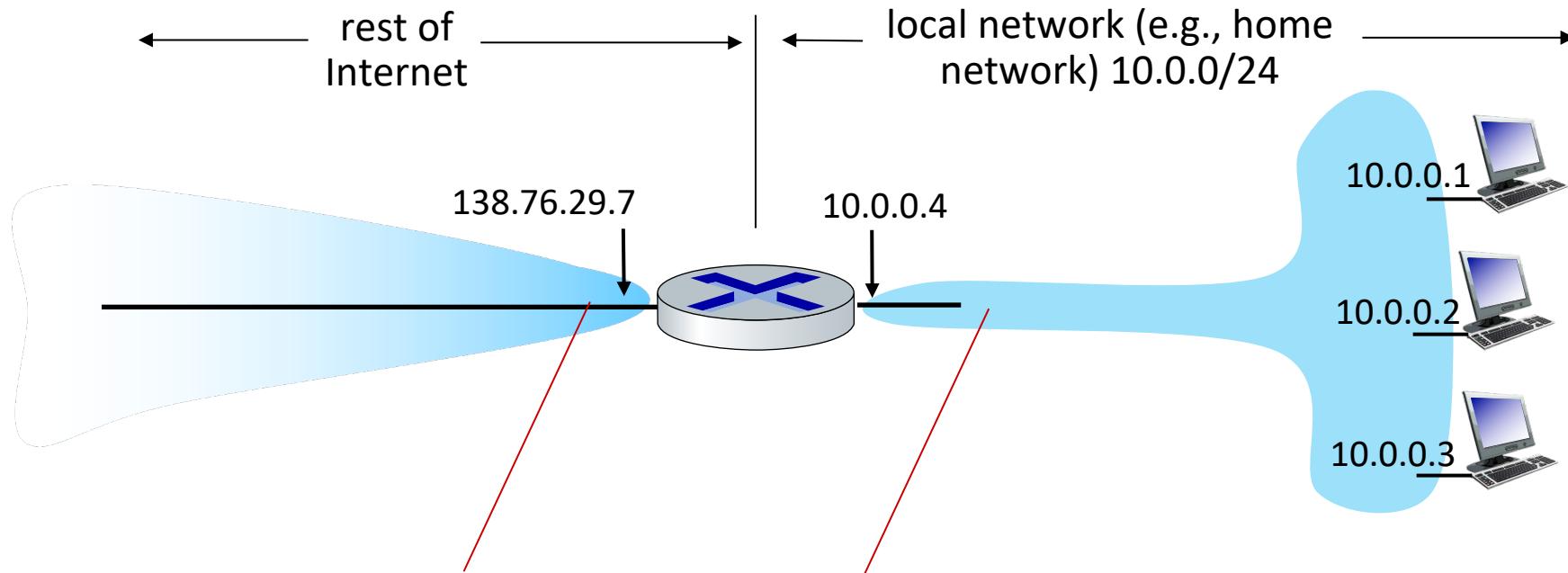
- IP: the Internet Protocol
  - datagram format
  - addressing
  - network address translation
  - IPv6



- Generalized Forwarding, SDN
  - match+action
  - OpenFlow: match+action in action
- Middleboxes

# NAT: network address translation

**NAT:** all devices in local network share just **one** IPv4 address as far as outside world is concerned



*all* datagrams *leaving* local network have *same* source NAT IP address: 138.76.29.7, but *different* source port numbers

datagrams with source or destination in this network have 10.0.0/24 address for source, destination (as usual)

# NAT: network address translation

- all devices in local network have 32-bit addresses in a “private” IP address space (10/8, 172.16/12, 192.168/16 prefixes) that can only be used in local network
- advantages:
  - just **one** IP address needed from provider ISP for ***all*** devices
  - can change addresses of host in local network without notifying outside world
  - can change ISP without changing addresses of devices in local network
  - security: devices inside local net not directly addressable, visible by outside world

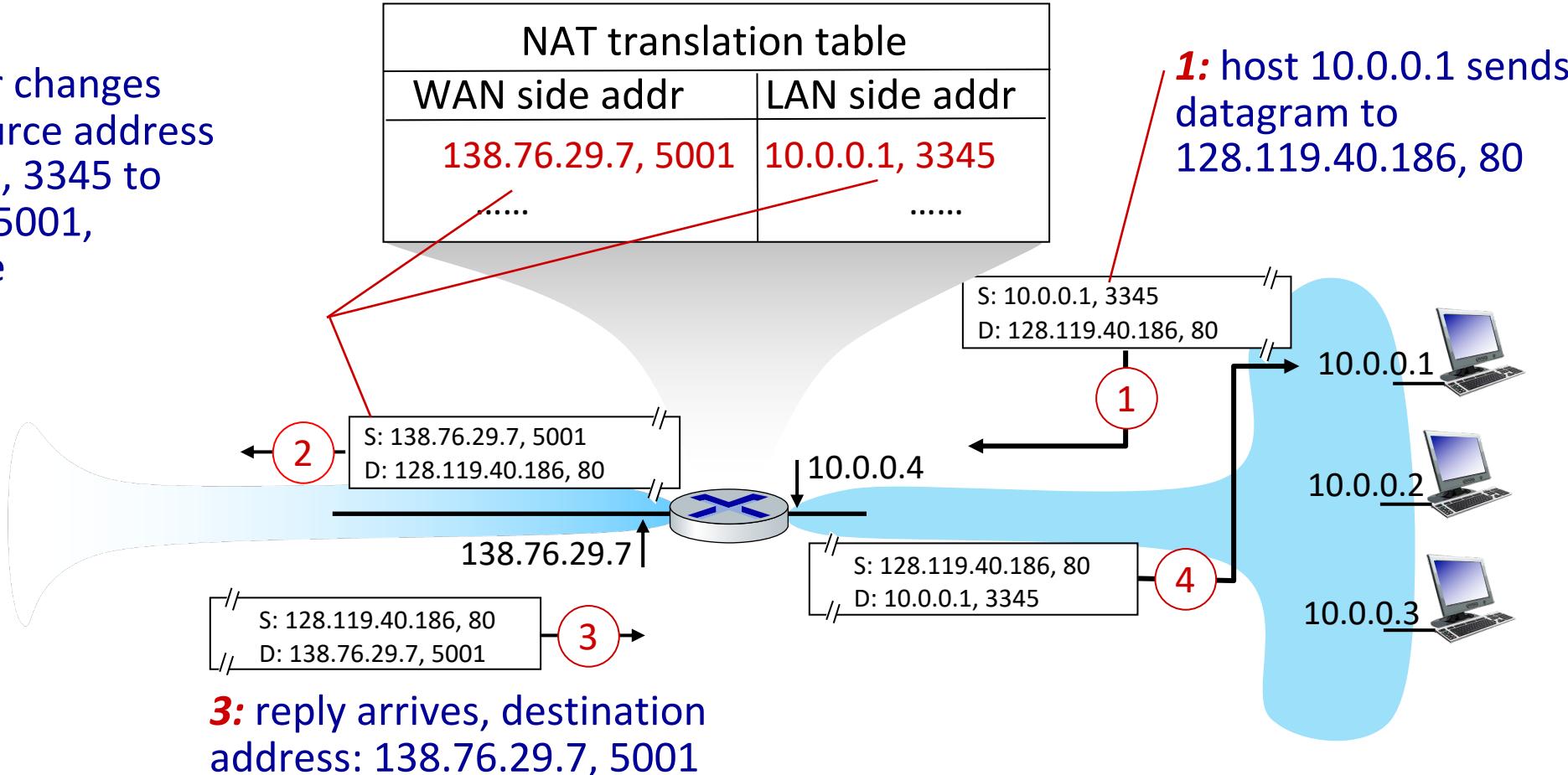
# NAT: network address translation

implementation: NAT router must (transparently):

- outgoing datagrams: replace (source IP address, port #) of every outgoing datagram to (NAT IP address, new port #)
  - remote clients/servers will respond using (NAT IP address, new port #) as destination address
- remember (in NAT translation table) every (source IP address, port #) to (NAT IP address, new port #) translation pair
- incoming datagrams: replace (NAT IP address, new port #) in destination fields of every incoming datagram with corresponding (source IP address, port #) stored in NAT table

# NAT: network address translation

**2:** NAT router changes datagram source address from 10.0.0.1, 3345 to 138.76.29.7, 5001, updates table



# NAT: network address translation

- NAT has been controversial:
  - routers “should” only process up to layer 3
  - address “shortage” should be solved by IPv6
  - violates end-to-end argument (port # manipulation by network-layer device)
  - NAT traversal: what if client wants to connect to server behind NAT?
- but NAT is here to stay:
  - extensively used in home and institutional nets, 4G/5G cellular nets

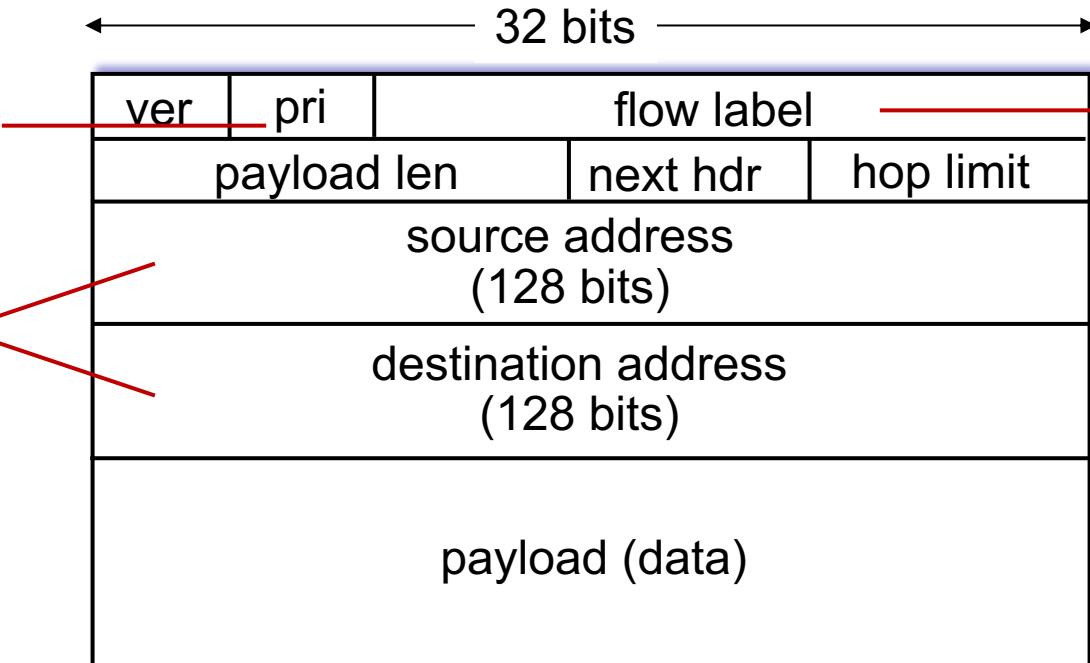
# IPv6: motivation

- **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

**priority:** identify priority among datagrams in flow

**128-bit**  
IPv6 addresses



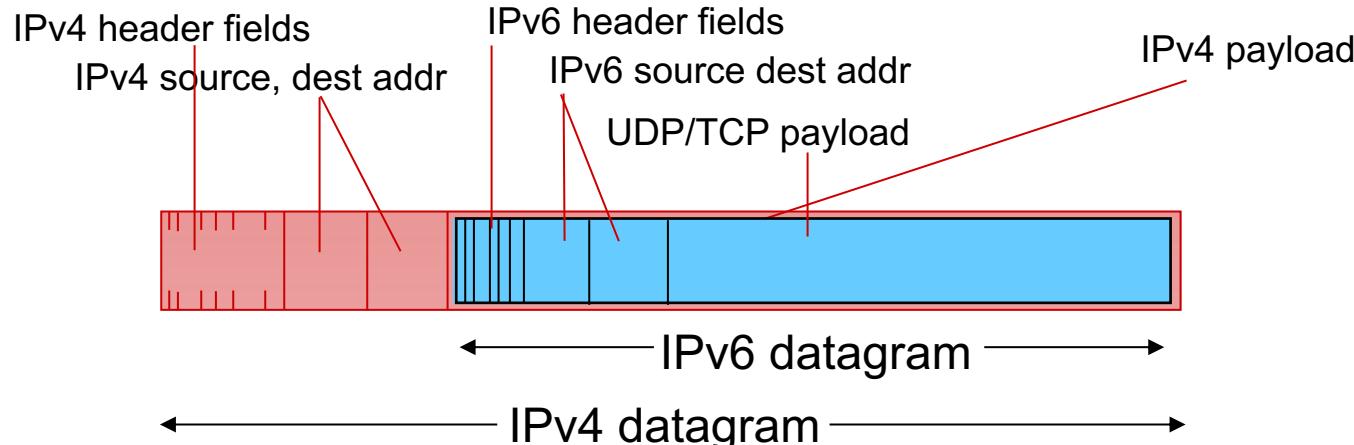
**flow label:** identify datagrams in same "flow." (concept of "flow" not well defined).

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)

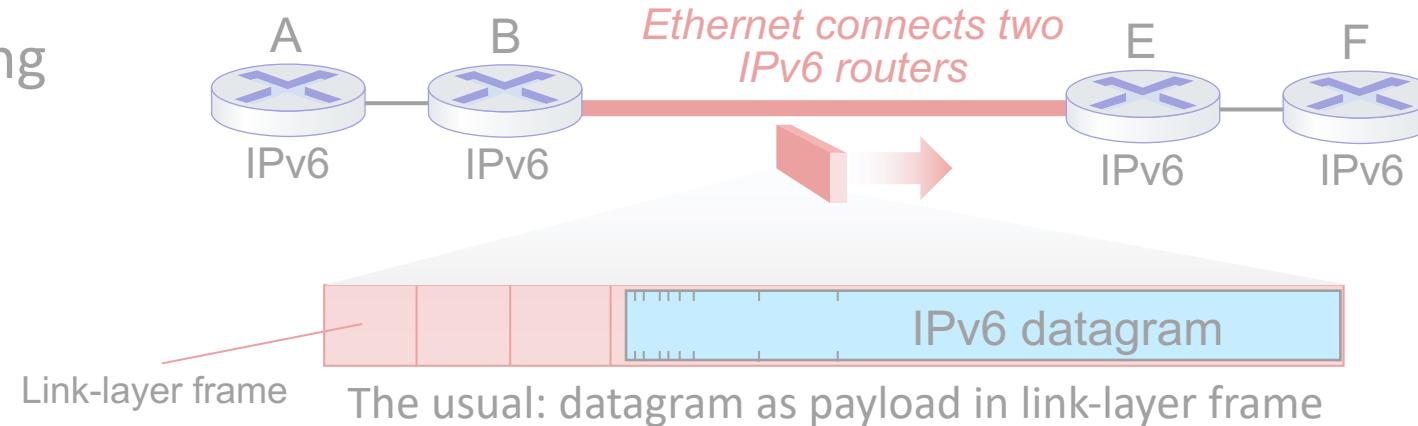
# Transition from IPv4 to IPv6

- 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)

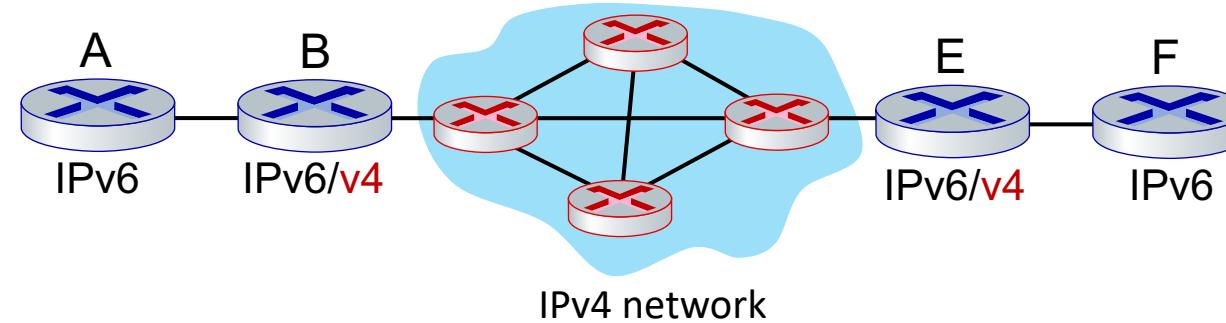


# Tunneling and encapsulation

Ethernet connecting  
two IPv6 routers:

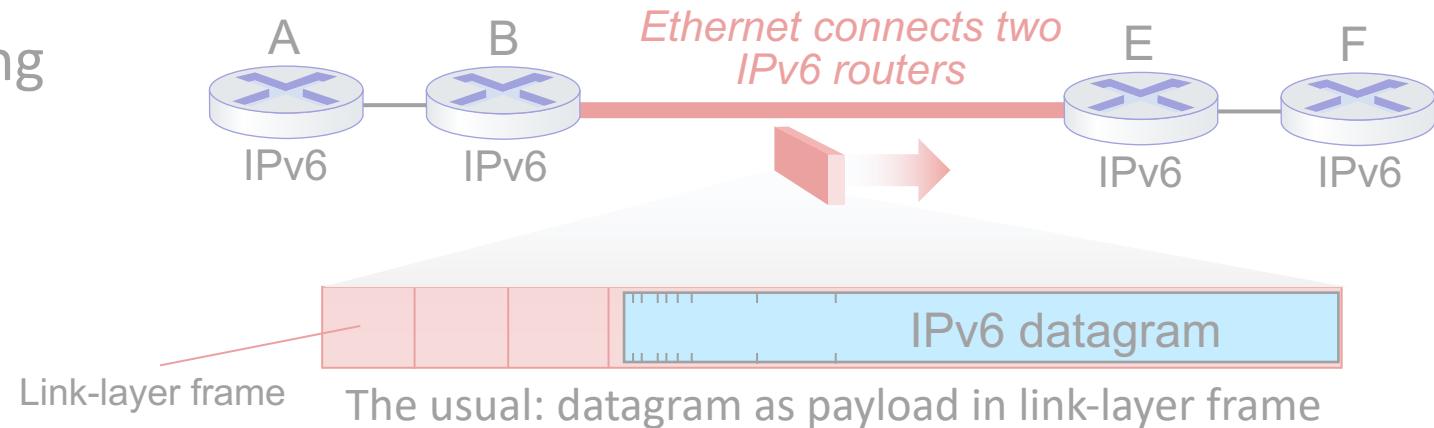


IPv4 network  
connecting two  
IPv6 routers

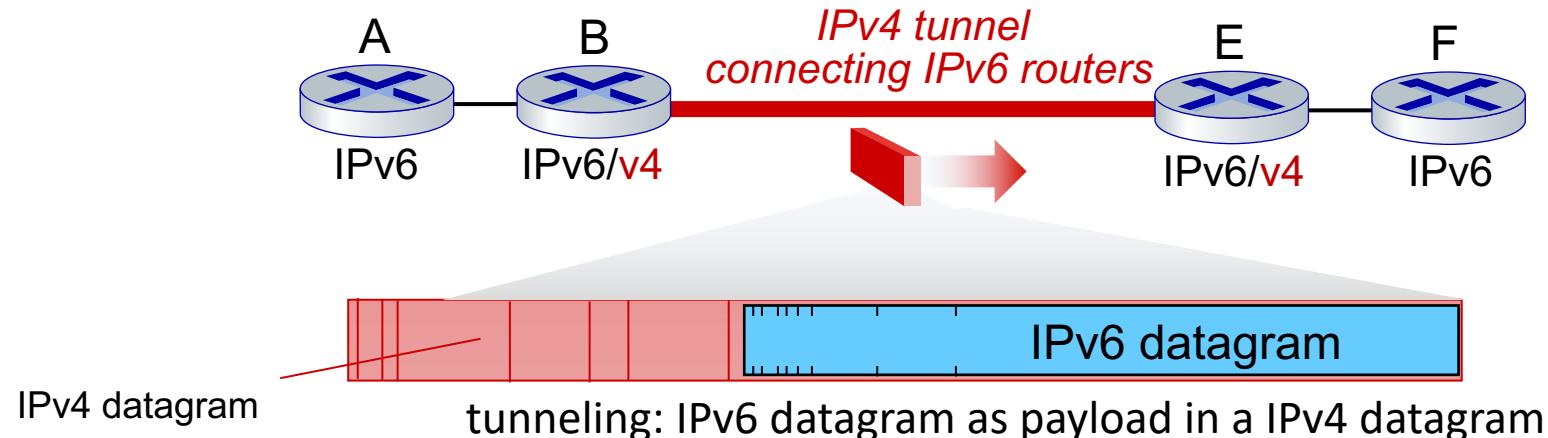


# Tunneling and encapsulation

Ethernet connecting  
two IPv6 routers:



IPv4 tunnel  
connecting two  
IPv6 routers



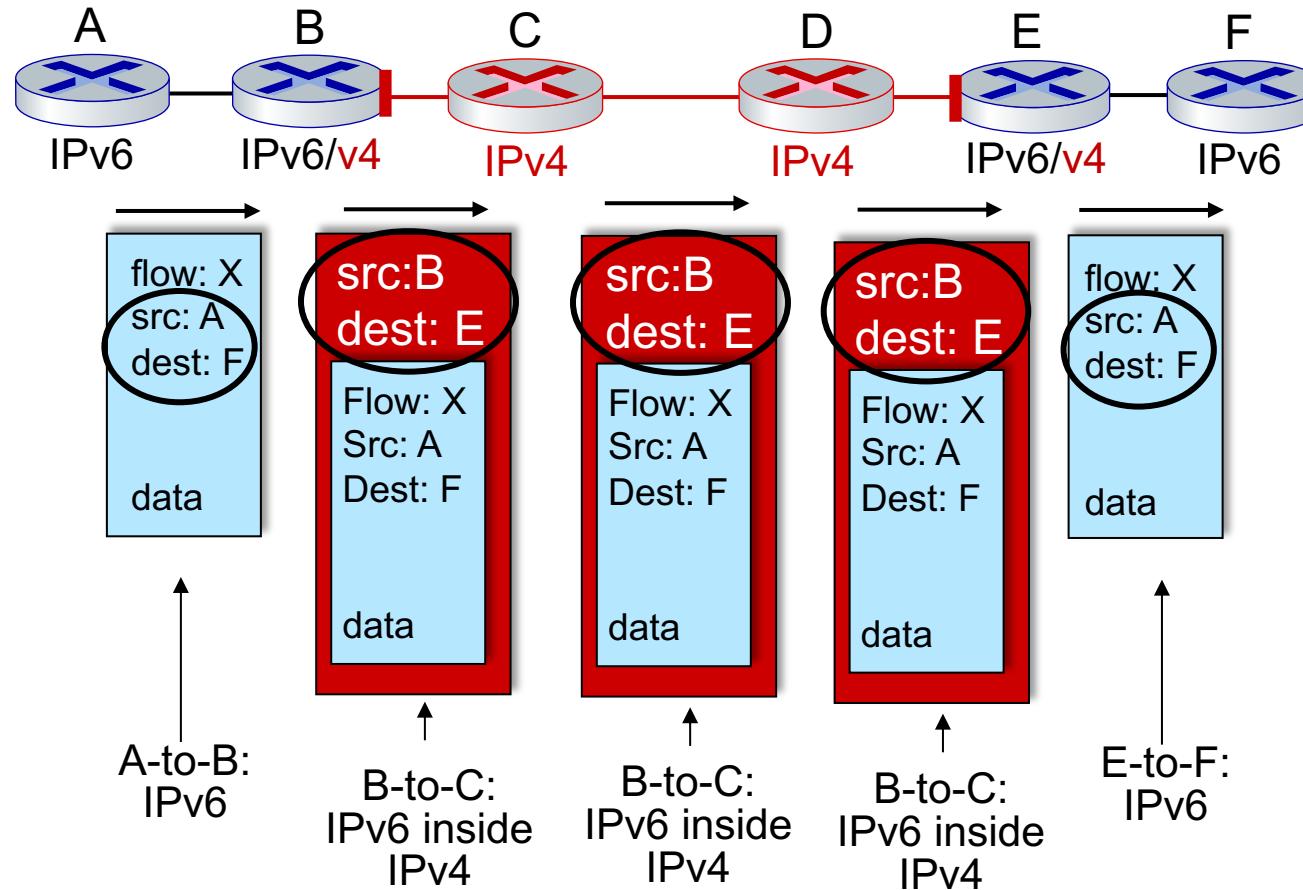
# Tunneling

logical view:



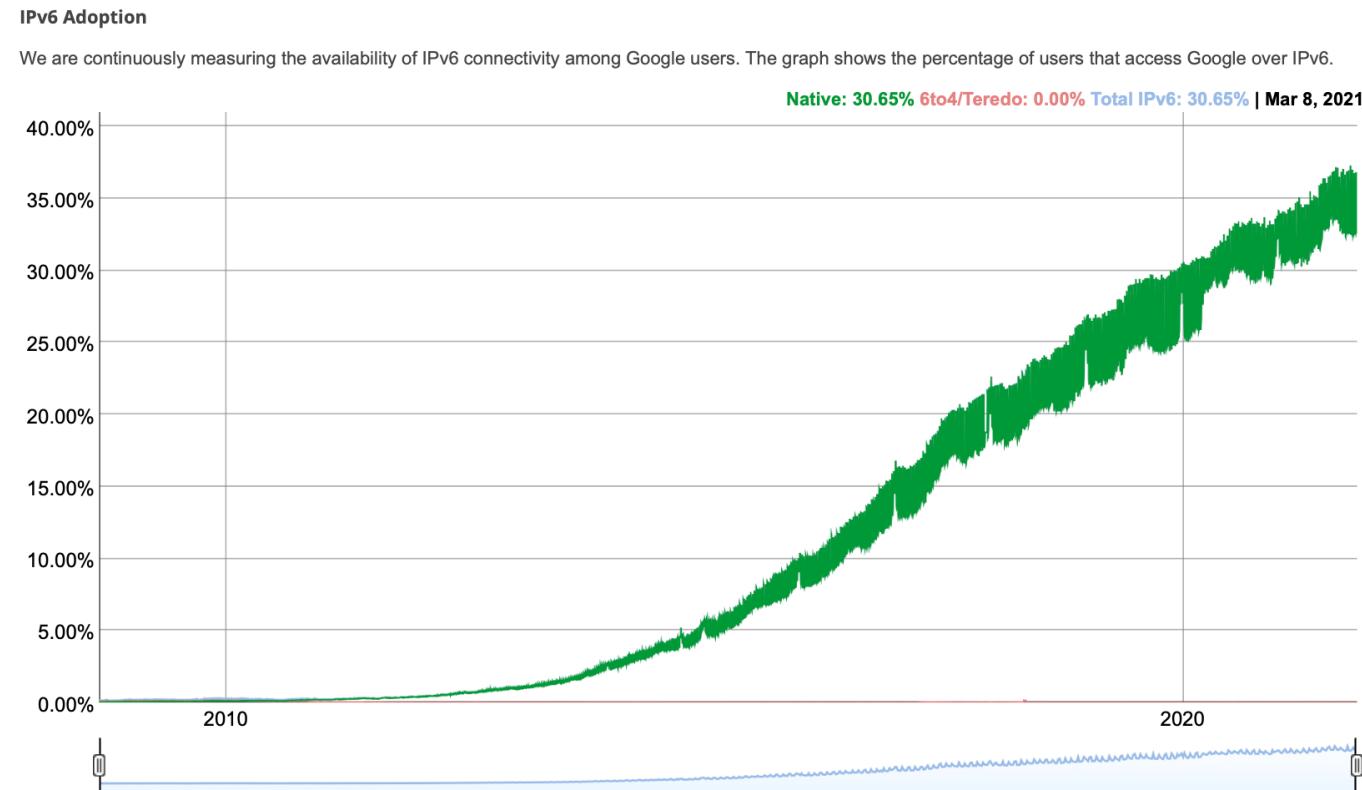
physical view:

Note source and destination addresses!



# IPv6: adoption

- Google<sup>1</sup>: ≈35% of clients access services via IPv6
- NIST: 1/3 of all US government domains are IPv6 capable



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

# IPv6: adoption

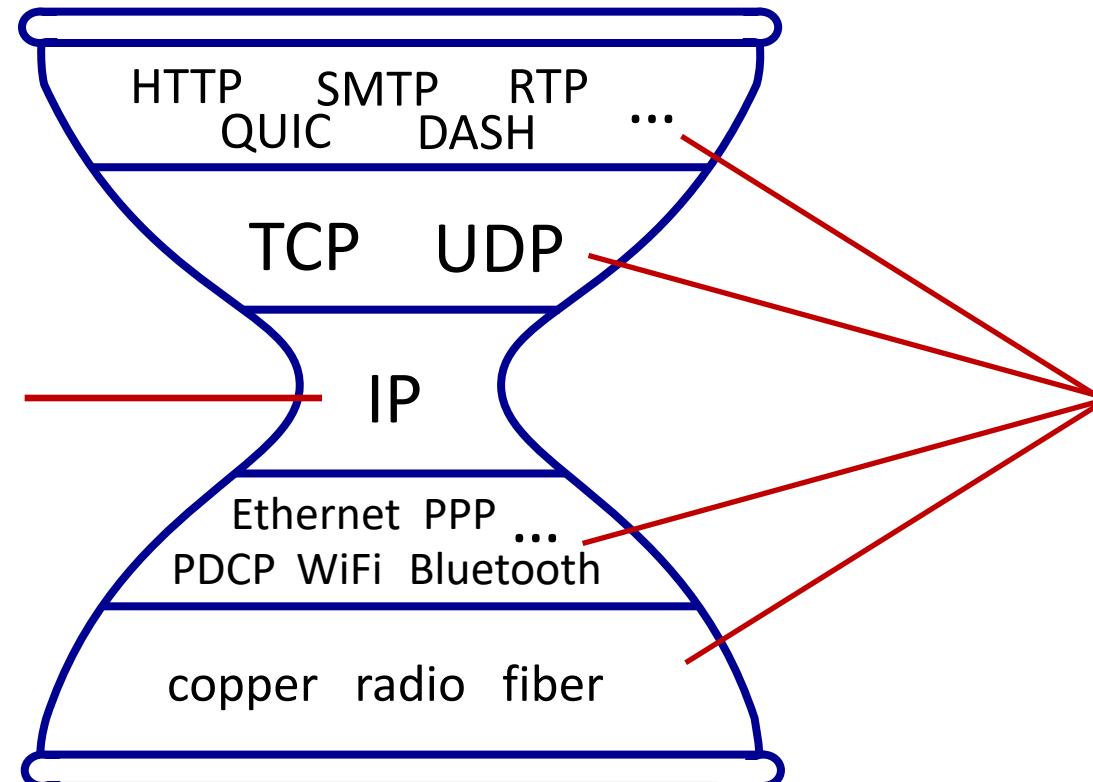
- Google<sup>1</sup>: ≈35% of clients access services via IPv6
- NIST: 1/3 of all US government domains are IPv6 capable
- Long (long!) time for deployment, use
  - 30 years and counting!
  - think of application-level changes in last 25 years: WWW, social media, streaming media, gaming, telepresence, ...
  - *Why?*

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

# The IP hourglass

Internet's "thin waist":

- *one* network layer protocol: IP
- *must* be implemented by every (billions) of Internet-connected devices

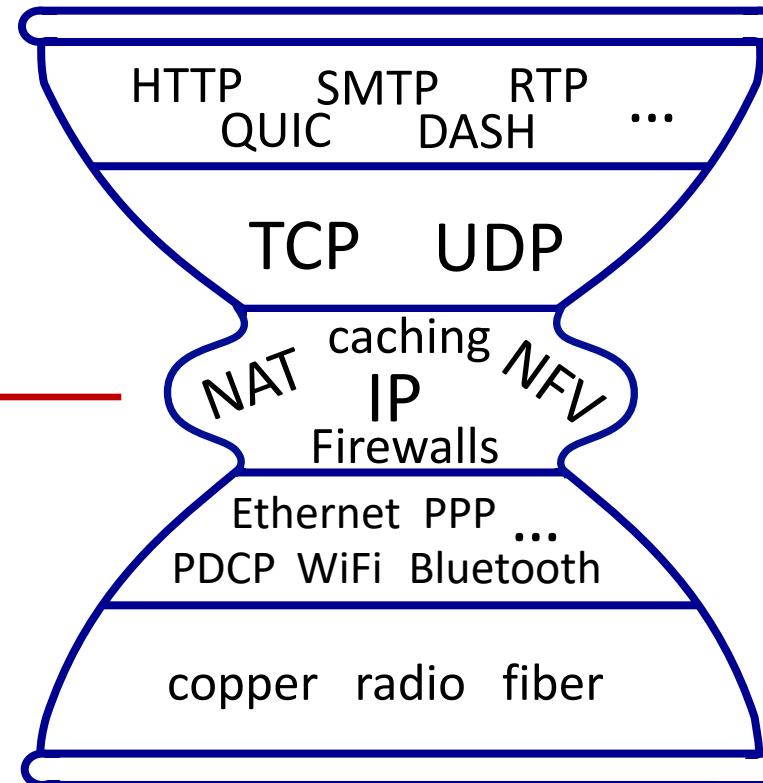


*many* protocols in physical, link, transport, and application layers

# The IP hourglass, at middle age

Internet's middle age  
“love handles”?

- middleboxes,  
operating inside the  
network



# Architectural Principles of the Internet

RFC 1958

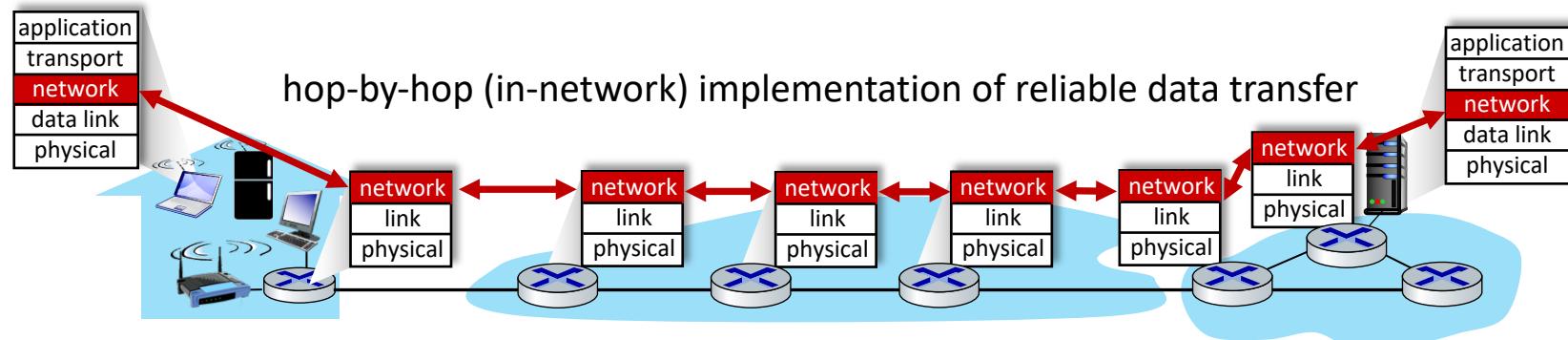
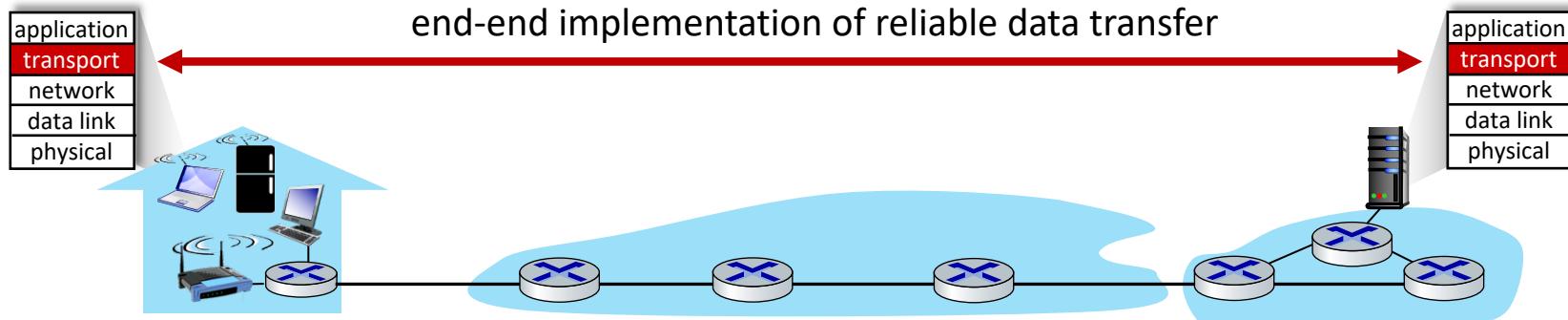
“Many members of the Internet community would argue that there is no architecture, but only a tradition, which was not written down for the first 25 years (or at least not by the IAB). However, in very general terms, the community believes that **the goal is connectivity, the tool is the Internet Protocol, and the intelligence is end to end rather than hidden in the network.**”

Three cornerstone beliefs:

- simple connectivity
- IP protocol: that narrow waist
- intelligence, complexity at network edge

# The end-end argument

- some network functionality (e.g., reliable data transfer, congestion) can be implemented in **network**, or at **network edge**



# The end-end argument

- some network functionality (e.g., reliable data transfer, congestion) can be implemented in network, or at network edge

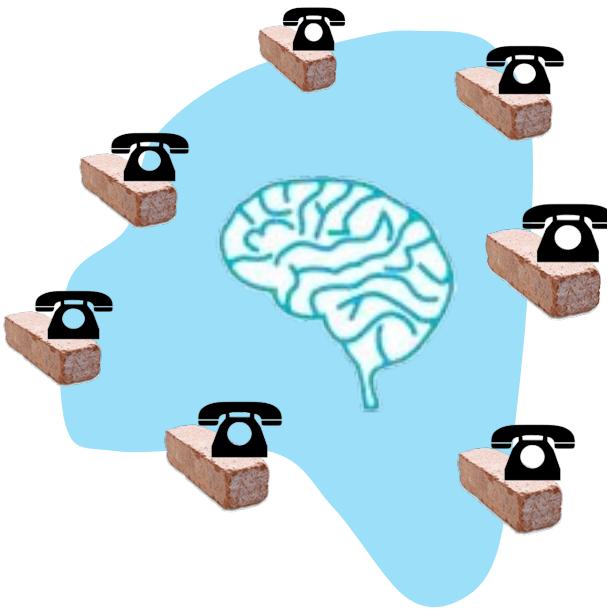
“The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

We call this line of reasoning against low-level function implementation the “end-to-end argument.”

---

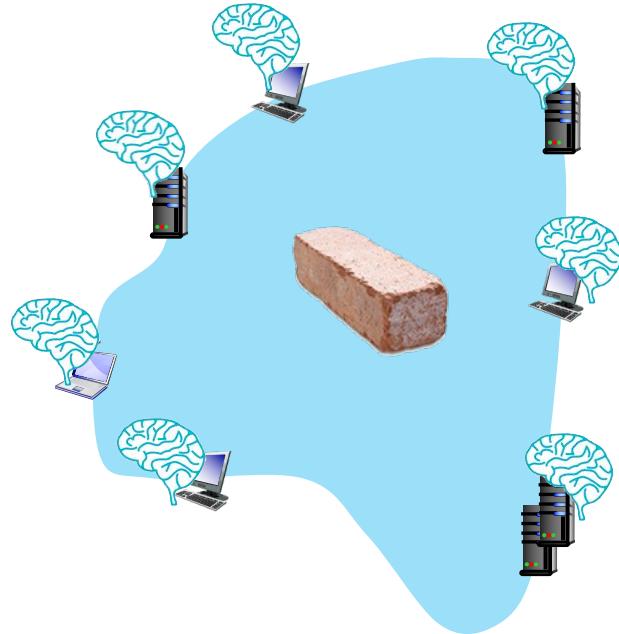
Saltzer, Reed, Clark 1981

# Where's the intelligence?



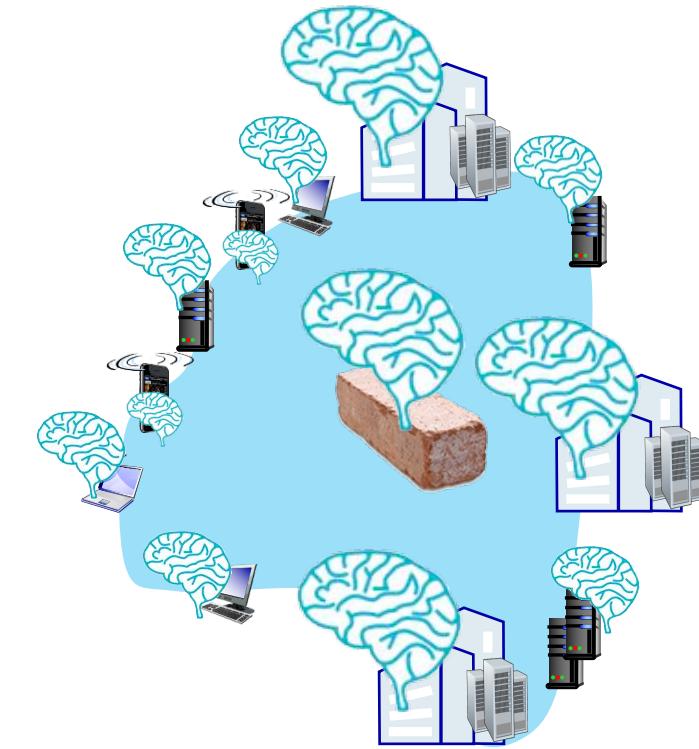
**20<sup>th</sup> century phone net:**

- intelligence/computing at network switches



**Internet (pre-2005)**

- intelligence, computing at edge



**Internet (post-2005)**

- programmable network devices
- intelligence, computing, massive application-level infrastructure at edge

# Chapter 4: done!

- Network layer: overview
- What's inside a router
- IP: the Internet Protocol
- Generalized Forwarding, SDN
- Middleboxes



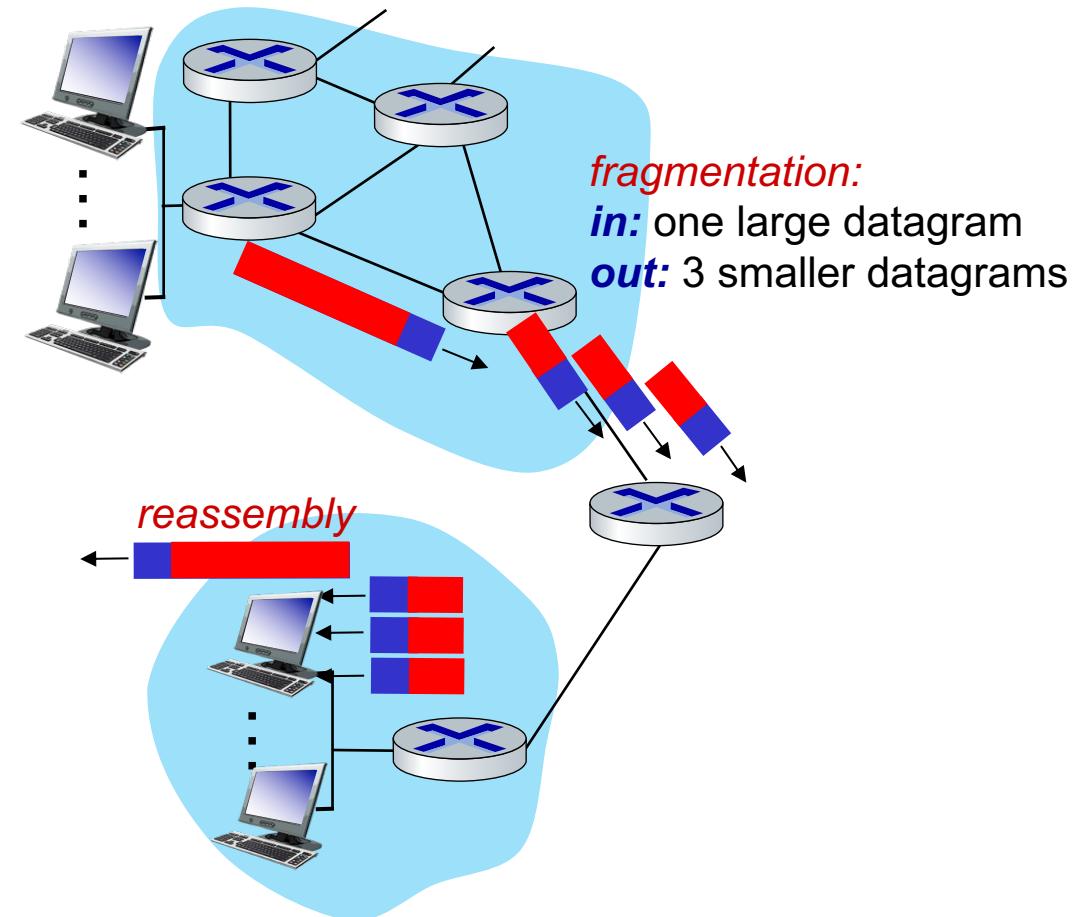
*Question:* how are forwarding tables (destination-based forwarding) or flow tables (generalized forwarding) computed?

*Answer:* by the control plane (next chapter)

# Additional Chapter 4 slides

# IP fragmentation/reassembly

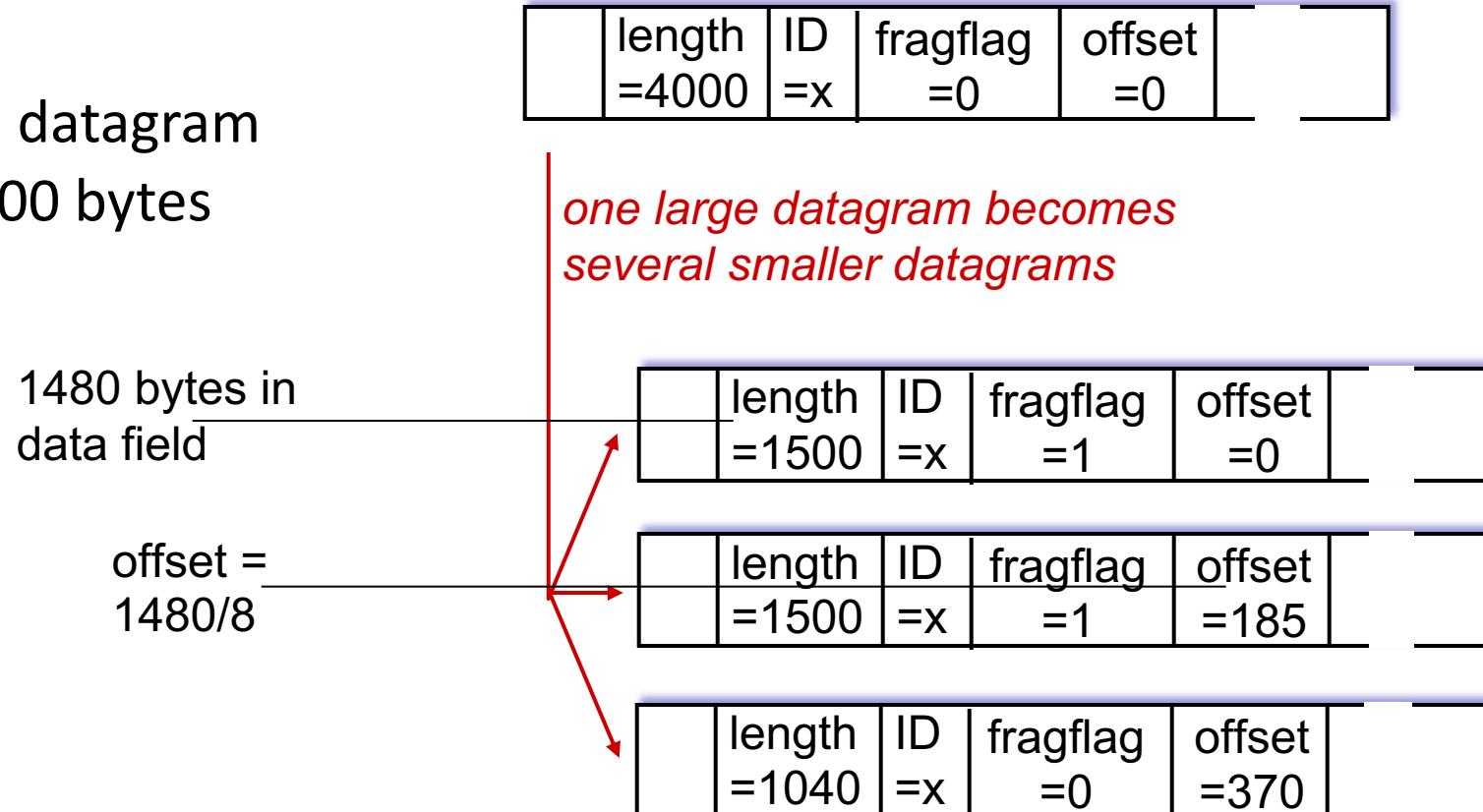
- network links have MTU (max. transfer size) - largest possible link-level frame
  - different link types, different MTUs
- large IP datagram divided (“fragmented”) within net
  - one datagram becomes several datagrams
  - “reassembled” only at *destination*
  - IP header bits used to identify, order related fragments



# IP fragmentation/reassembly

## example:

- 4000 byte datagram
- MTU = 1500 bytes



# DHCP: Wireshark output (home LAN)

Message type: **Boot Request (1)**

Hardware type: Ethernet

Hardware address length: 6

Hops: 0

**Transaction ID: 0x6b3a11b7**

Seconds elapsed: 0

Bootp flags: 0x0000 (Unicast)

Client IP address: 0.0.0.0 (0.0.0.0)

Your (client) IP address: 0.0.0.0 (0.0.0.0)

Next server IP address: 0.0.0.0 (0.0.0.0)

Relay agent IP address: 0.0.0.0 (0.0.0.0)

**Client MAC address: Wistron\_23:68:8a (00:16:d3:23:68:8a)**

Server host name not given

Boot file name not given

Magic cookie: (OK)

Option: (t=53,l=1) **DHCP Message Type = DHCP Request**

Option: (61) Client identifier

Length: 7; Value: 010016D323688A;

Hardware type: Ethernet

Client MAC address: Wistron\_23:68:8a (00:16:d3:23:68:8a)

Option: (t=50,l=4) Requested IP Address = 192.168.1.101

Option: (t=12,l=5) Host Name = "nomad"

**Option: (55) Parameter Request List**

Length: 11; Value: 010F03062C2E2F1F21F92B

**1 = Subnet Mask; 15 = Domain Name**

**3 = Router; 6 = Domain Name Server**

44 = NetBIOS over TCP/IP Name Server

.....

request

reply

Message type: **Boot Reply (2)**

Hardware type: Ethernet

Hardware address length: 6

Hops: 0

**Transaction ID: 0x6b3a11b7**

Seconds elapsed: 0

Bootp flags: 0x0000 (Unicast)

**Client IP address: 192.168.1.101 (192.168.1.101)**

Your (client) IP address: 0.0.0.0 (0.0.0.0)

**Next server IP address: 192.168.1.1 (192.168.1.1)**

Relay agent IP address: 0.0.0.0 (0.0.0.0)

Client MAC address: Wistron\_23:68:8a (00:16:d3:23:68:8a)

Server host name not given

Boot file name not given

Magic cookie: (OK)

**Option: (t=53,l=1) DHCP Message Type = DHCP ACK**

**Option: (t=54,l=4) Server Identifier = 192.168.1.1**

**Option: (t=1,l=4) Subnet Mask = 255.255.255.0**

**Option: (t=3,l=4) Router = 192.168.1.1**

**Option: (6) Domain Name Server**

Length: 12; Value: 445747E2445749F244574092;

IP Address: 68.87.71.226;

IP Address: 68.87.73.242;

IP Address: 68.87.64.146

**Option: (t=15,l=20) Domain Name = "hsd1.ma.comcast.net."**

# Link layer, LANs: roadmap

- introduction
- error detection, correction
- multiple access protocols
- LANs
  - addressing, ARP
  - Ethernet
  - switches
  - VLANs
- link virtualization: MPLS
- data center networking



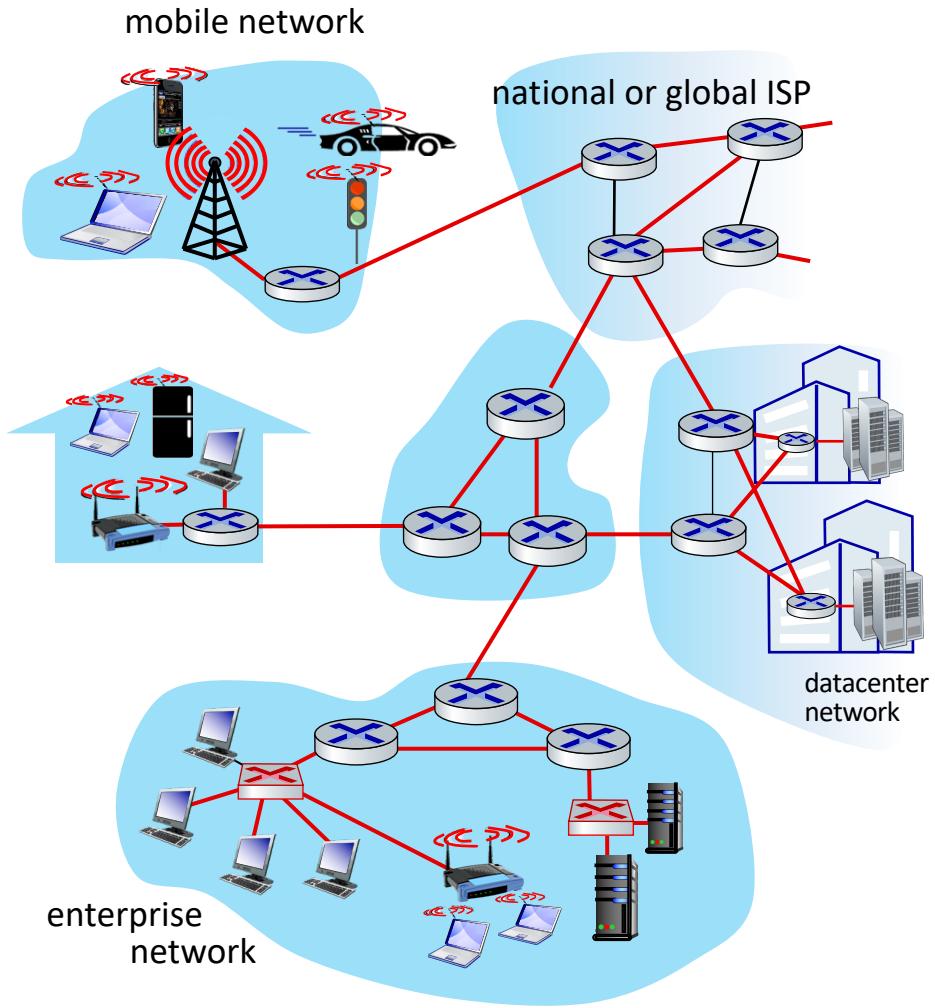
- a day in the life of a web request

# Link layer: introduction

terminology:

- hosts and routers: nodes
- communication channels that connect adjacent nodes along communication path: links
  - wired
  - wireless
  - LANs
- layer-2 packet: *frame*, encapsulates datagram

*link layer* has responsibility of transferring datagram from one node to **physically adjacent** node over a link



# Link layer: context

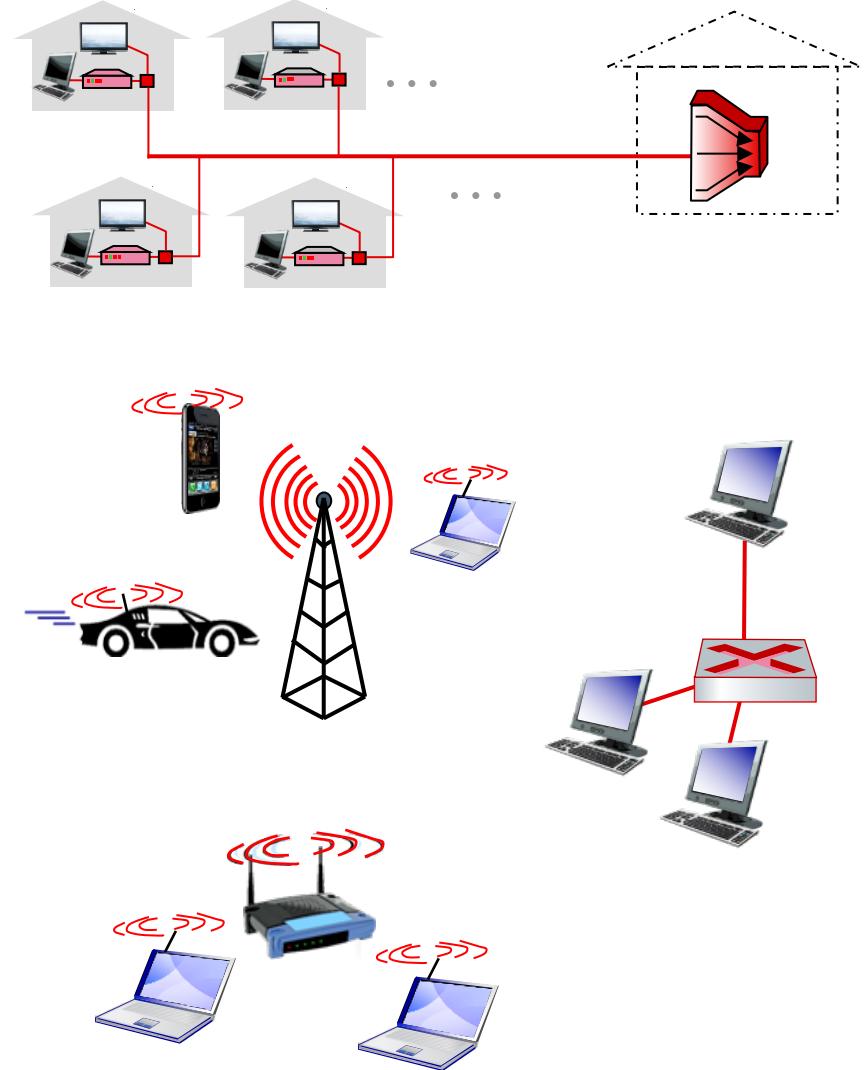
- datagram transferred by different link protocols over different links:
  - e.g., WiFi on first link, Ethernet on next link
- each link protocol provides different services
  - e.g., may or may not provide reliable data transfer over link

## transportation analogy:

- trip from Princeton to Provo
  - limo: Princeton to JFK
  - plane: JFK to SLC
  - train: SLC to Provo
- tourist = **datagram**
- transport segment = **communication link**
- transportation mode = **link-layer protocol**
- travel agent = **routing algorithm**

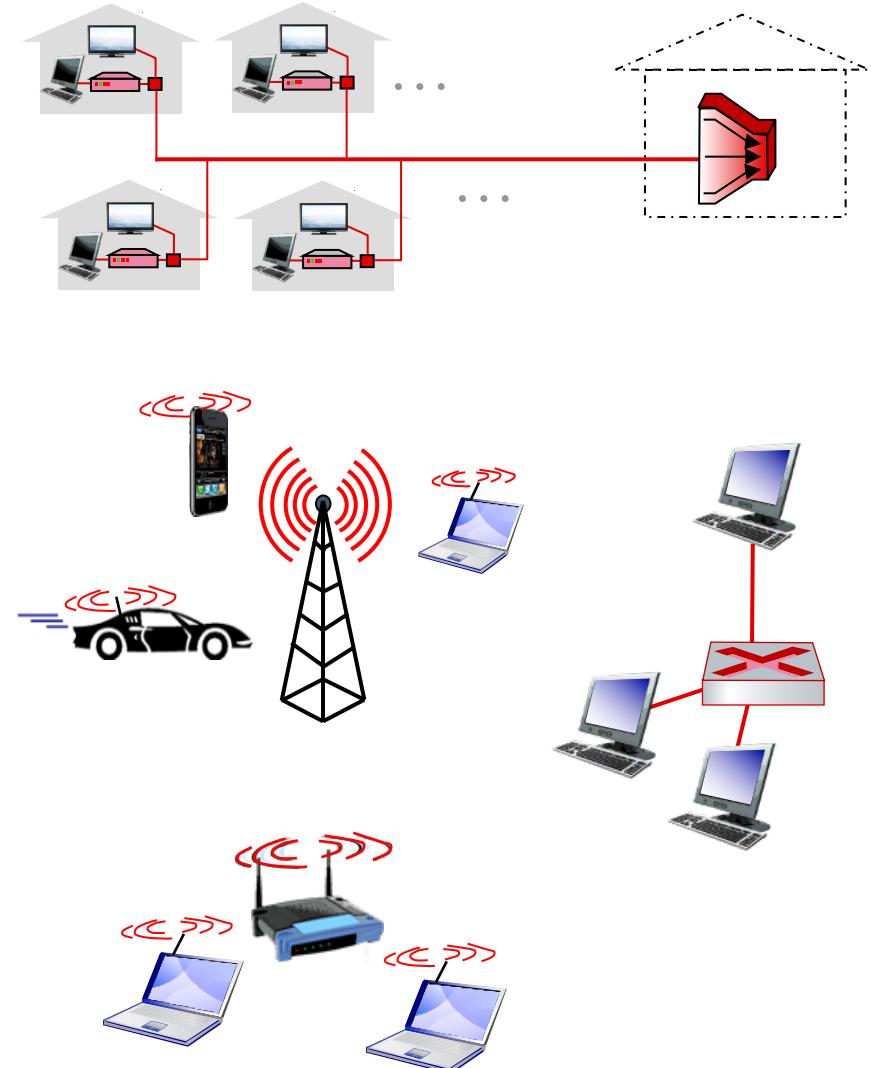
# Link layer: services

- **framing, link access:**
  - encapsulate datagram into frame, adding header, trailer
  - channel access if shared medium
  - “MAC” addresses in frame headers identify source, destination (different from IP address!)
- **reliable delivery between adjacent nodes**
  - we already know how to do this!
  - seldom used on low bit-error links
  - wireless links: high error rates
    - Q: why both link-level and end-to-end reliability?



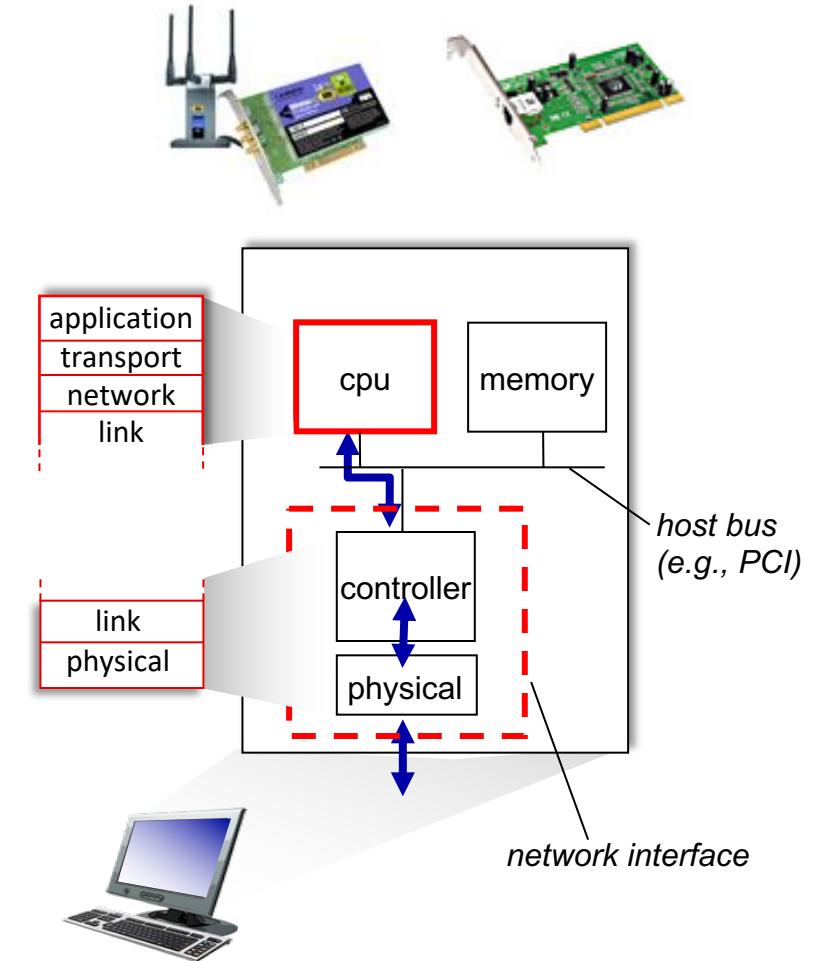
# Link layer: services (more)

- **flow control:**
  - pacing between adjacent sending and receiving nodes
- **error detection:**
  - errors caused by signal attenuation, noise.
  - receiver detects errors, signals retransmission, or drops frame
- **error correction:**
  - receiver identifies *and corrects* bit error(s) without retransmission
- **half-duplex and full-duplex:**
  - with half duplex, nodes at both ends of link can transmit, but not at same time

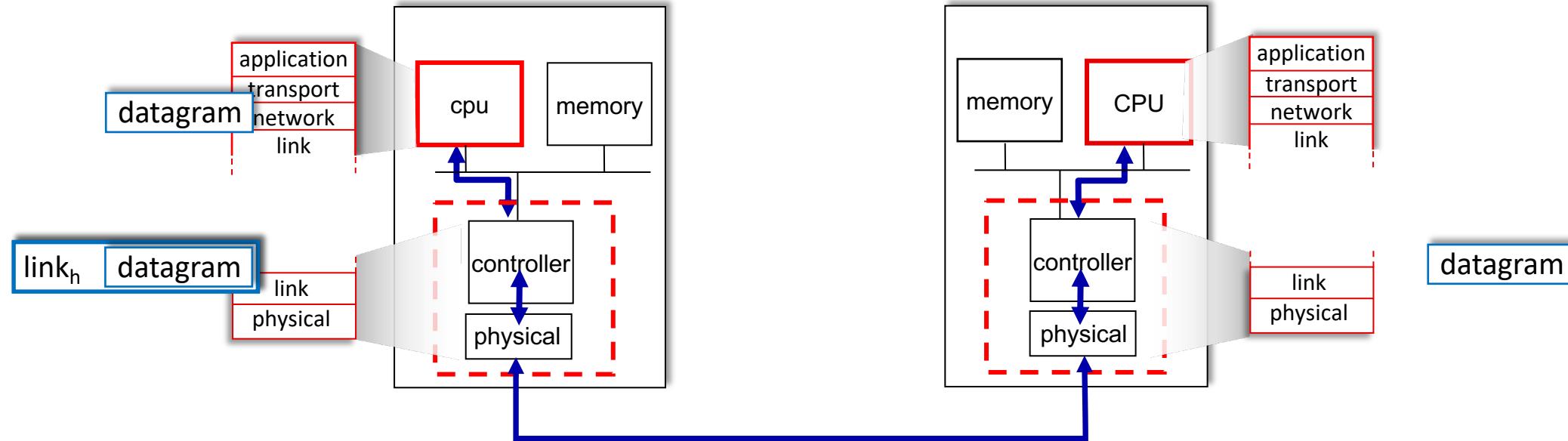


# Where is the link layer implemented?

- in each-and-every host
- link layer implemented in *network interface card* (NIC) or on a chip
  - Ethernet, WiFi card or chip
  - implements link, physical layer
- attaches into host's system buses
- combination of hardware, software, firmware



# Interfaces communicating



sending side:

- encapsulates datagram in frame
- adds error checking bits, reliable data transfer, flow control, etc.

receiving side:

- looks for errors, reliable data transfer, flow control, etc.
- extracts datagram, passes to upper layer at receiving side

# Link layer, LANs: roadmap

- introduction
- **error detection, correction**
- multiple access protocols
- LANs
  - addressing, ARP
  - Ethernet
  - switches
  - VLANs
- link virtualization: MPLS
- data center networking

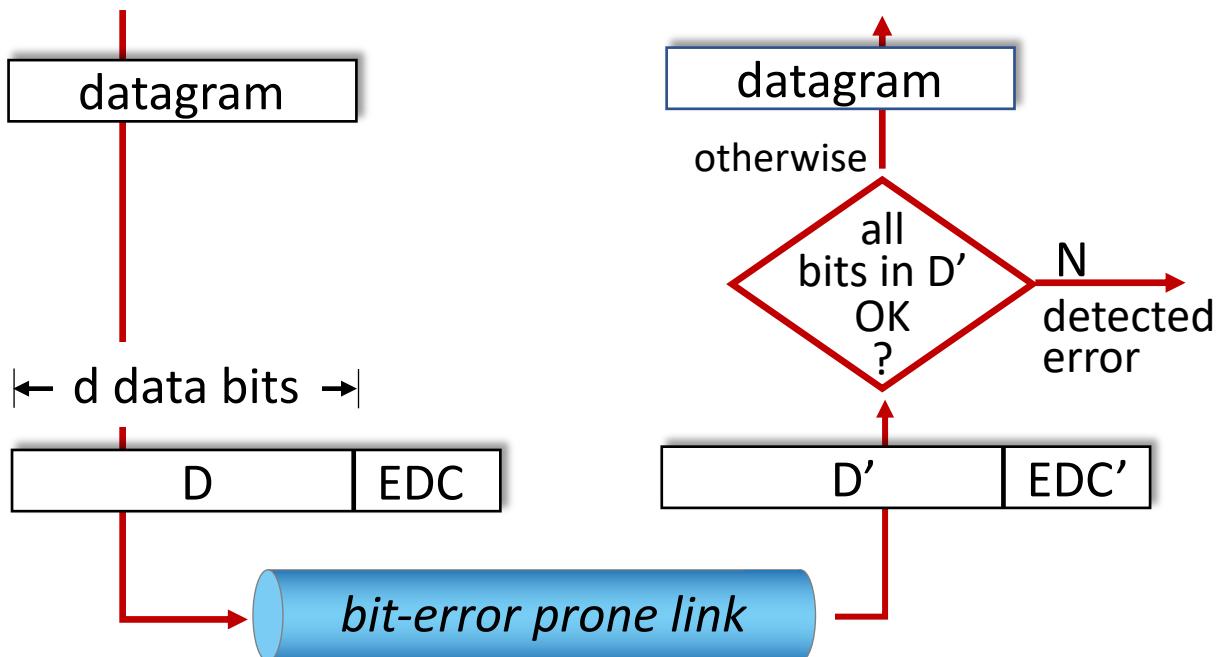


- a day in the life of a web request

# Error detection

EDC: error detection and correction bits (e.g., redundancy)

D: data protected by error checking, may include header fields



Error detection not 100% reliable!

- protocol may miss some errors, but rarely
- larger EDC field yields better detection and correction

# Link layer, LANs: roadmap

- introduction
- error detection, correction
- **multiple access protocols**
- LANs
  - addressing, ARP
  - Ethernet
  - switches
  - VLANs
- link virtualization: MPLS
- data center networking



- a day in the life of a web request

# Multiple access links, protocols

two types of “links”:

- point-to-point
  - point-to-point link between Ethernet switch, host
  - PPP for dial-up access
- broadcast (shared wire or medium)
  - old-fashioned Ethernet
  - upstream HFC in cable-based access network
  - 802.11 wireless LAN, cellular, satellite



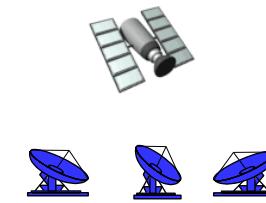
shared wire (e.g.,  
cabled Ethernet)



shared radio: 4G/5G



shared radio: WiFi



shared radio: satellite



humans at a party  
(shared air, acoustical)

# Multiple access protocols

- single shared broadcast channel
- two or more simultaneous transmissions by nodes: interference
  - *collision* if node receives two or more signals at the same time

## multiple access protocol

- distributed algorithm that determines how nodes share channel, i.e., determine when node can transmit
- communication about channel sharing must use channel itself!
  - no out-of-band channel for coordination

# An ideal multiple access protocol

*given:* multiple access channel of rate  $R$  bps

*Desirable characteristics:*

1. when one node wants to transmit, it can send at rate  $R$ .
2. when  $M$  nodes want to transmit, each can send at average rate  $R/M$
3. fully decentralized:
  - no special node to coordinate transmissions
  - no synchronization of clocks, slots
4. simple

# MAC protocols: taxonomy

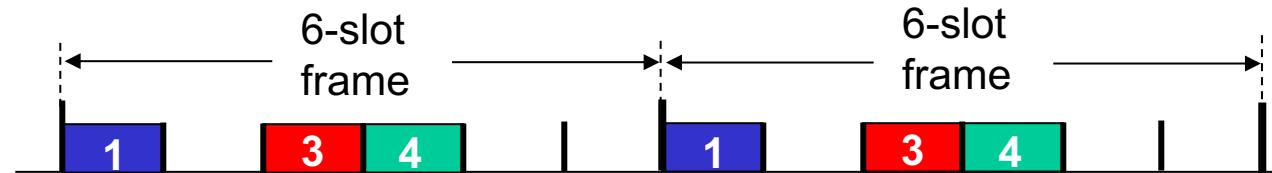
three broad classes:

- **channel partitioning**
  - divide channel into smaller “pieces” (time slots, frequency, code, space)
  - allocate piece to node for exclusive use
- *random access*
  - channel not divided, allow collisions
  - “recover” from collisions
- **“taking turns”**
  - nodes take turns, but nodes with more to send can take longer turns

# Channel partitioning MAC protocols: TDMA

## TDMA: time division multiple access

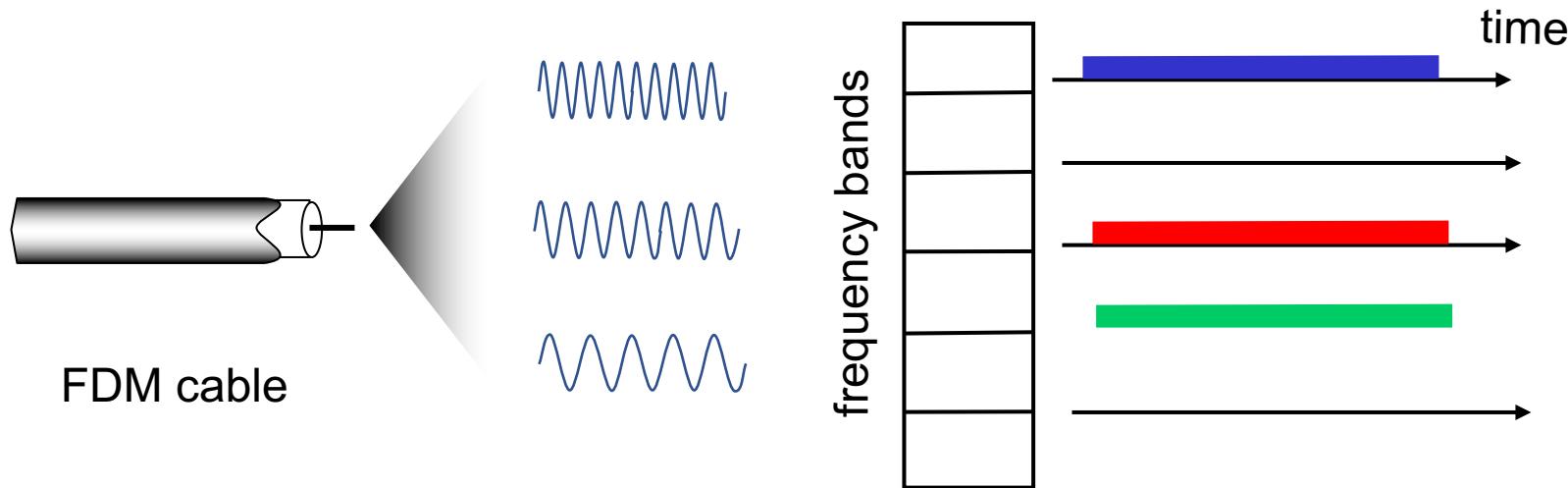
- access to channel in “rounds”
- each station gets fixed length slot (length = packet transmission time) in each round
- unused slots go idle
- example: 6-station LAN, 1,3,4 have packets to send, slots 2,5,6 idle



# Channel partitioning MAC protocols: FDMA

## FDMA: frequency division multiple access

- channel spectrum divided into frequency bands
- each station assigned fixed frequency band
- unused transmission time in frequency bands go idle
- example: 6-station LAN, 1,3,4 have packet to send, frequency bands 2,5,6 idle



# Random access protocols

- when node has packet to send
  - transmit at full channel data rate  $R$ .
  - no *a priori* coordination among nodes
- two or more transmitting nodes: “collision”
- **random access MAC protocol** specifies:
  - how to detect collisions
  - how to recover from collisions (e.g., via delayed retransmissions)
- examples of random access MAC protocols:
  - ALOHA, slotted ALOHA
  - CSMA/CD, CSMA/CA

# Slotted ALOHA

## assumptions:

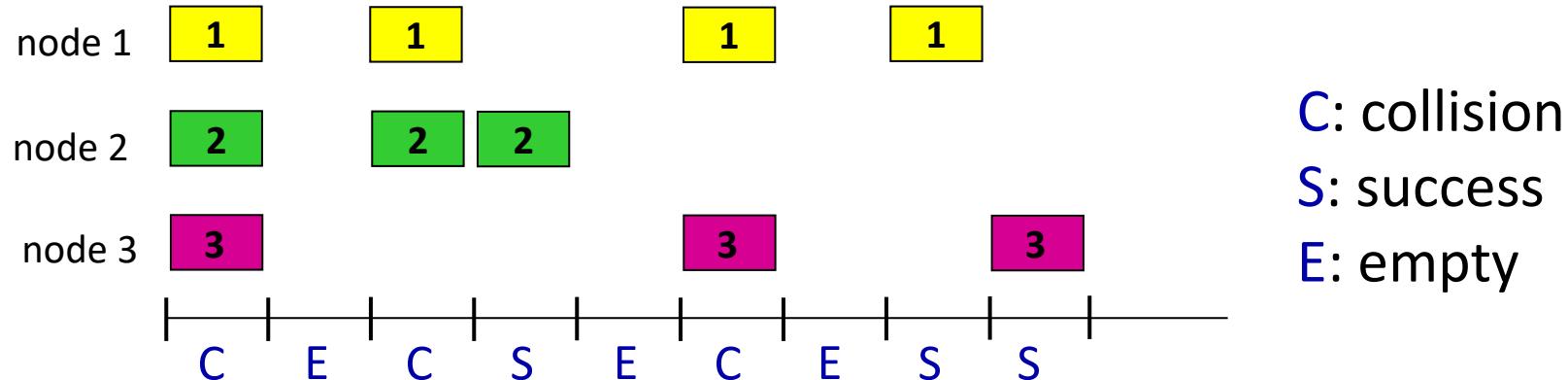
- all frames same size
- time divided into equal size slots (time to transmit 1 frame)
- nodes start to transmit only slot beginning
- nodes are synchronized
- if 2 or more nodes transmit in slot, all nodes detect collision

## operation:

- when node obtains fresh frame, transmits in next slot
  - *if no collision*: node can send new frame in next slot
  - *if collision*: node retransmits frame in each subsequent slot with probability  $p$  until success

randomization – *why?*

# Slotted ALOHA



## Pros:

- single active node can continuously transmit at full rate of channel
- highly decentralized: only slots in nodes need to be in sync
- simple

## Cons:

- collisions, wasting slots
- idle slots
- nodes may be able to detect collision in less than time to transmit packet
- clock synchronization

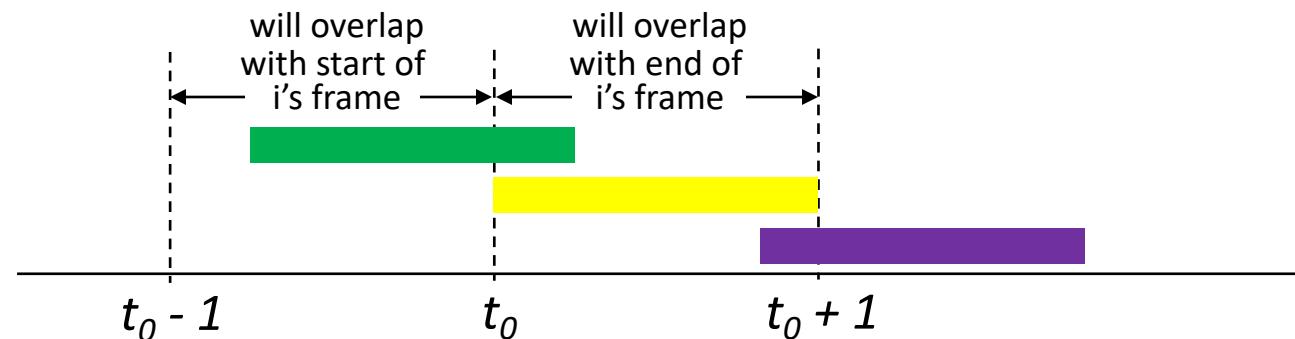
# Slotted ALOHA: efficiency

**efficiency:** long-run fraction of successful slots (many nodes, all with many frames to send)

- *suppose:*  $N$  nodes with many frames to send, each transmits in slot with probability  $p$ 
  - prob that given node has success in a slot =  $p(1-p)^{N-1}$
  - prob that *any* node has a success =  $Np(1-p)^{N-1}$
  - max efficiency: find  $p^*$  that maximizes  $Np(1-p)^{N-1}$
  - for many nodes, take limit of  $Np^*(1-p^*)^{N-1}$  as  $N$  goes to infinity, gives:  
*max efficiency =  $1/e = .37$*
- *at best:* channel used for useful transmissions 37% of time!

# Pure ALOHA

- unslotted Aloha: simpler, no synchronization
  - when frame first arrives: transmit immediately
- collision probability increases with no synchronization:
  - frame sent at  $t_0$  collides with other frames sent in  $[t_0-1, t_0+1]$



- pure Aloha efficiency: 18% !

# CSMA (carrier sense multiple access)

simple **CSMA**: listen before transmit:

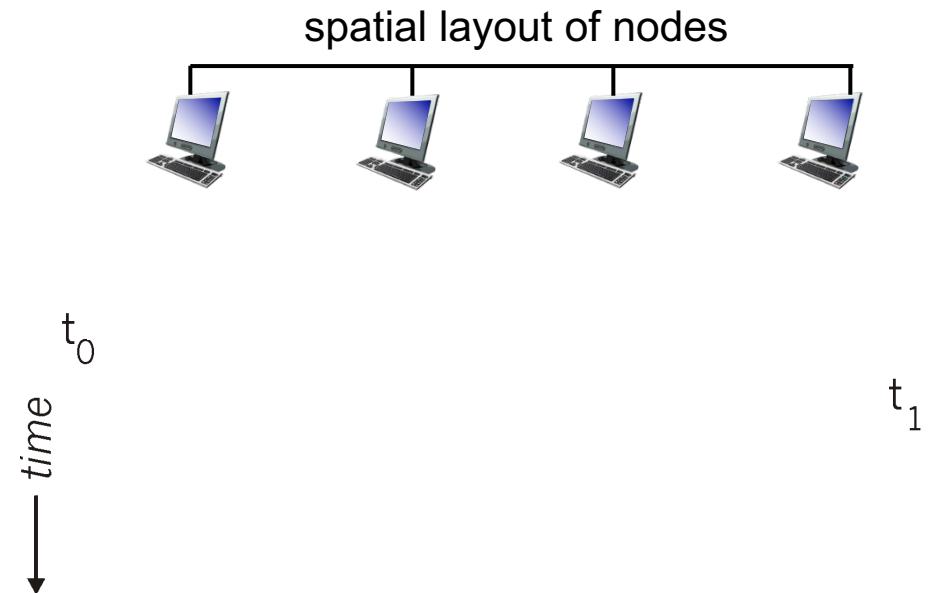
- if channel sensed idle: transmit entire frame
- if channel sensed busy: defer transmission
- human analogy: don't interrupt others!

**CSMA/CD**: CSMA with *collision detection*

- collisions *detected* within short time
- colliding transmissions aborted, reducing channel wastage
- collision detection easy in wired, difficult with wireless
- human analogy: the polite conversationalist

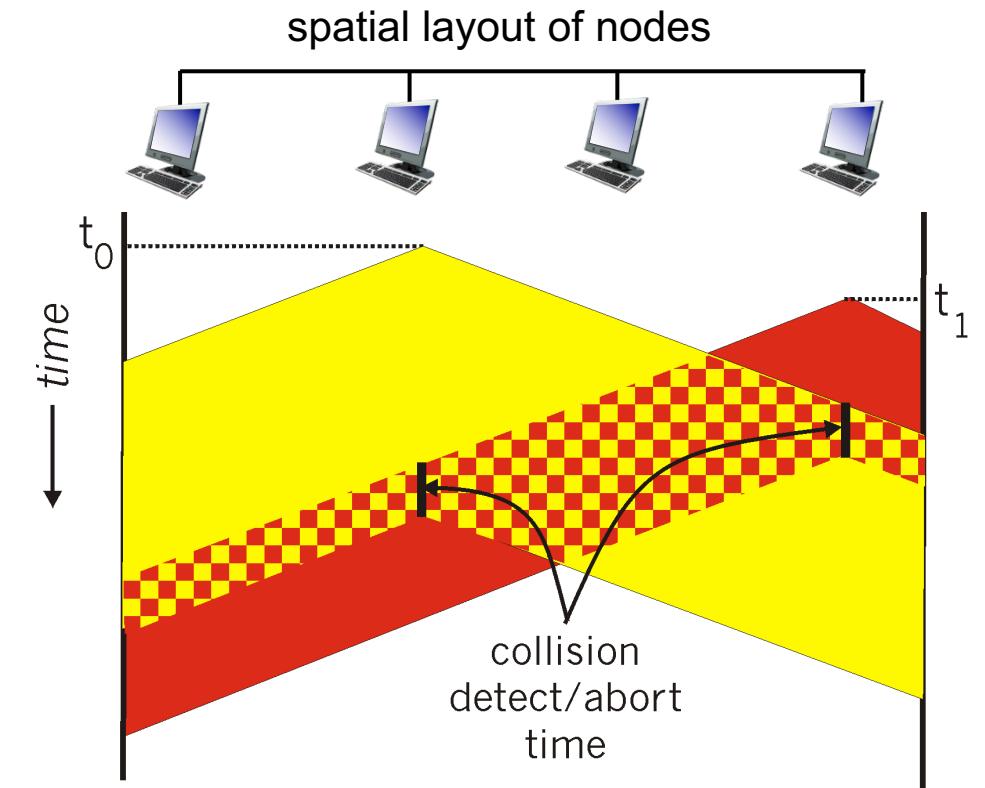
# CSMA: collisions

- collisions *can* still occur with carrier sensing:
  - propagation delay means two nodes may not hear each other's just-started transmission
- **collision:** entire packet transmission time wasted
  - distance & propagation delay play role in determining collision probability



# CSMA/CD:

- CSMA/CD reduces the amount of time wasted in collisions
  - transmission aborted on collision detection



# Ethernet CSMA/CD algorithm

1. NIC receives datagram from network layer, creates frame
2. If NIC senses channel:
  - if **idle**: start frame transmission.
  - if **busy**: wait until channel idle, then transmit
3. If NIC transmits entire frame without collision, NIC is done with frame!
4. If NIC detects another transmission while sending: abort, send jam signal
5. After aborting, NIC enters *binary (exponential) backoff*:
  - after  $m$ th collision, NIC chooses  $K$  at random from  $\{0, 1, 2, \dots, 2^m - 1\}$ . NIC waits  $K \cdot 512$  bit times, returns to Step 2
  - more collisions: longer backoff interval

# “Taking turns” MAC protocols

## channel partitioning MAC protocols:

- share channel *efficiently* and *fairly* at high load
- inefficient at low load: delay in channel access,  $1/N$  bandwidth allocated even if only 1 active node!

## random access MAC protocols

- efficient at low load: single node can fully utilize channel
- high load: collision overhead

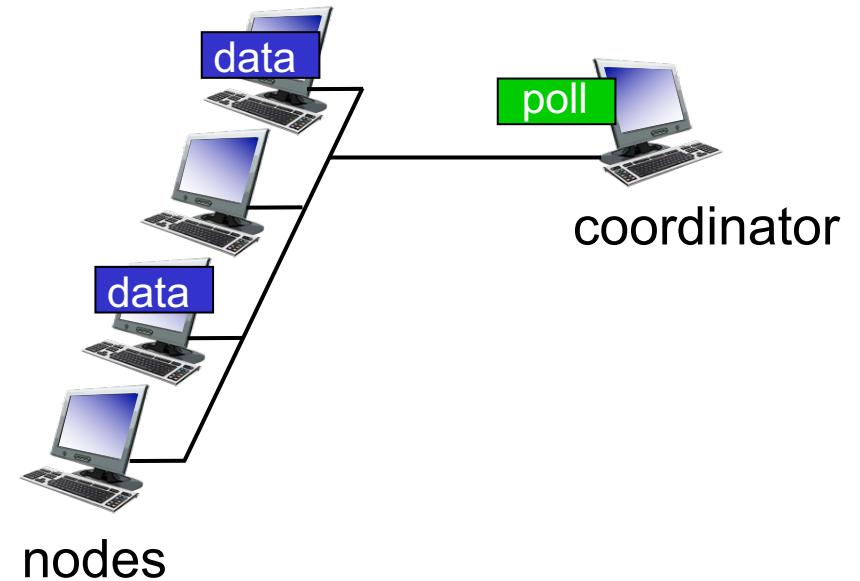
## “taking turns” protocols

- look for best of both worlds!

# “Taking turns” MAC protocols

## polling:

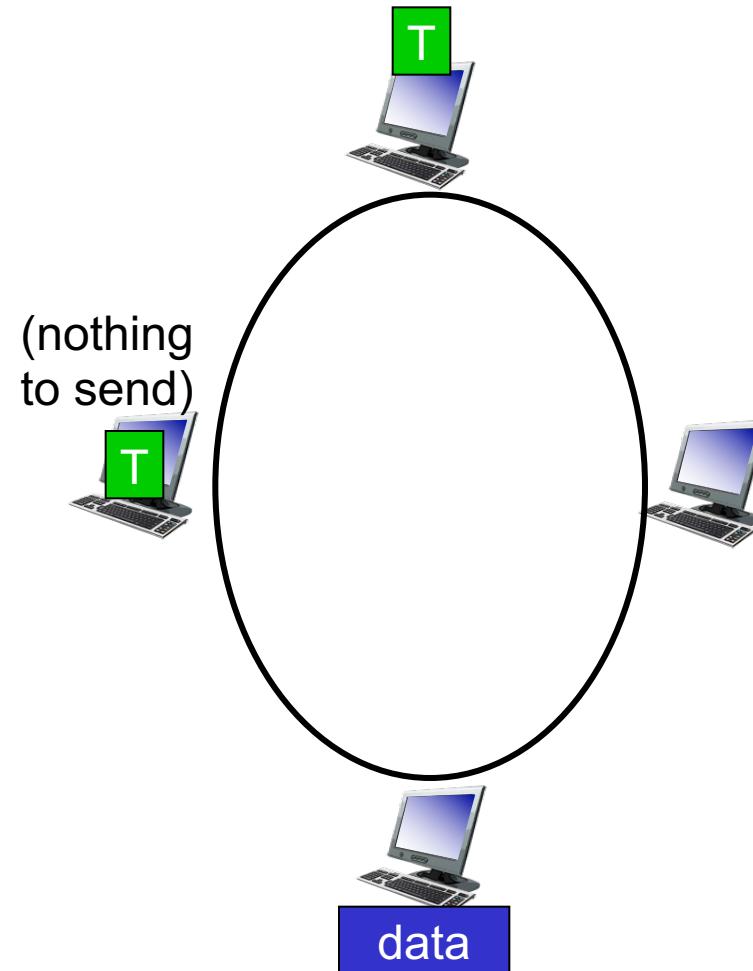
- master node “invites” other nodes to transmit in turn
- typically used with “dumb” devices
- concerns:
  - polling overhead
  - latency
  - single point of failure (master)



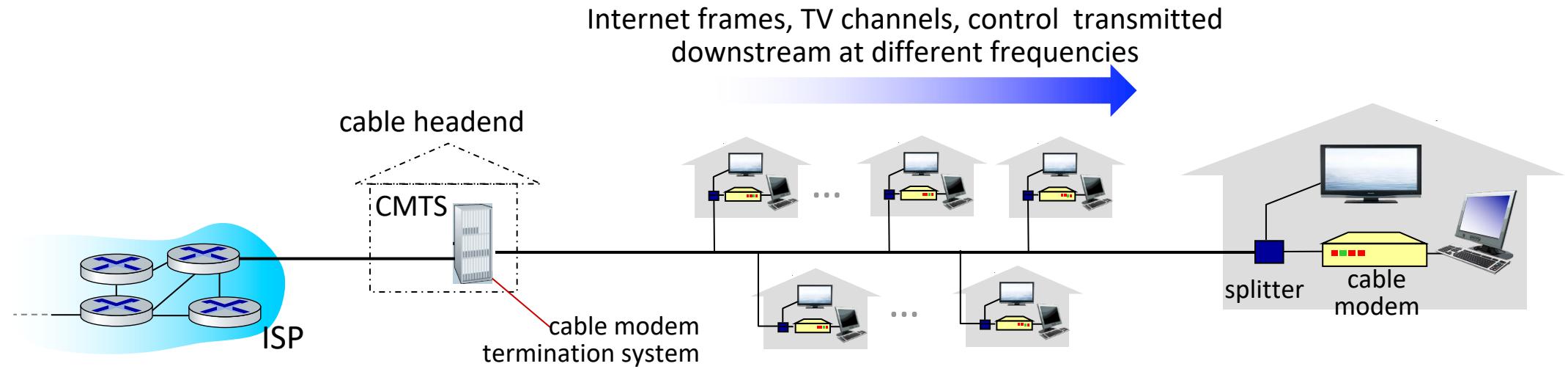
# “Taking turns” MAC protocols

## token passing:

- control *token* passed from one node to next sequentially.
- token message
- concerns:
  - token overhead
  - latency
  - single point of failure (token)



# Cable access network: FDM, TDM and random access!



- **multiple** downstream (broadcast) FDM channels: up to 1.6 Gbps/channel
  - single CMTS transmits into channels
- **multiple** upstream channels (up to 1 Gbps/channel)
  - **multiple access:** all users contend (random access) for certain upstream channel time slots; others assigned TDM

# Summary of MAC protocols

- **channel partitioning**, by time, frequency or code
  - Time Division, Frequency Division
- **random access (dynamic)**,
  - ALOHA, S-ALOHA, CSMA, CSMA/CD
  - carrier sensing: easy in some technologies (wire), hard in others (wireless)
  - CSMA/CD used in Ethernet
  - CSMA/CA used in 802.11
- **taking turns**
  - polling from central site, token passing
  - Bluetooth, FDDI, token ring

# Link layer, LANs: roadmap

- introduction
  - error detection, correction
  - multiple access protocols
  - **LANs**
    - addressing, ARP
    - Ethernet
    - switches
    - VLANs
  - link virtualization: MPLS
  - data center networking
- 
- a day in the life of a web request

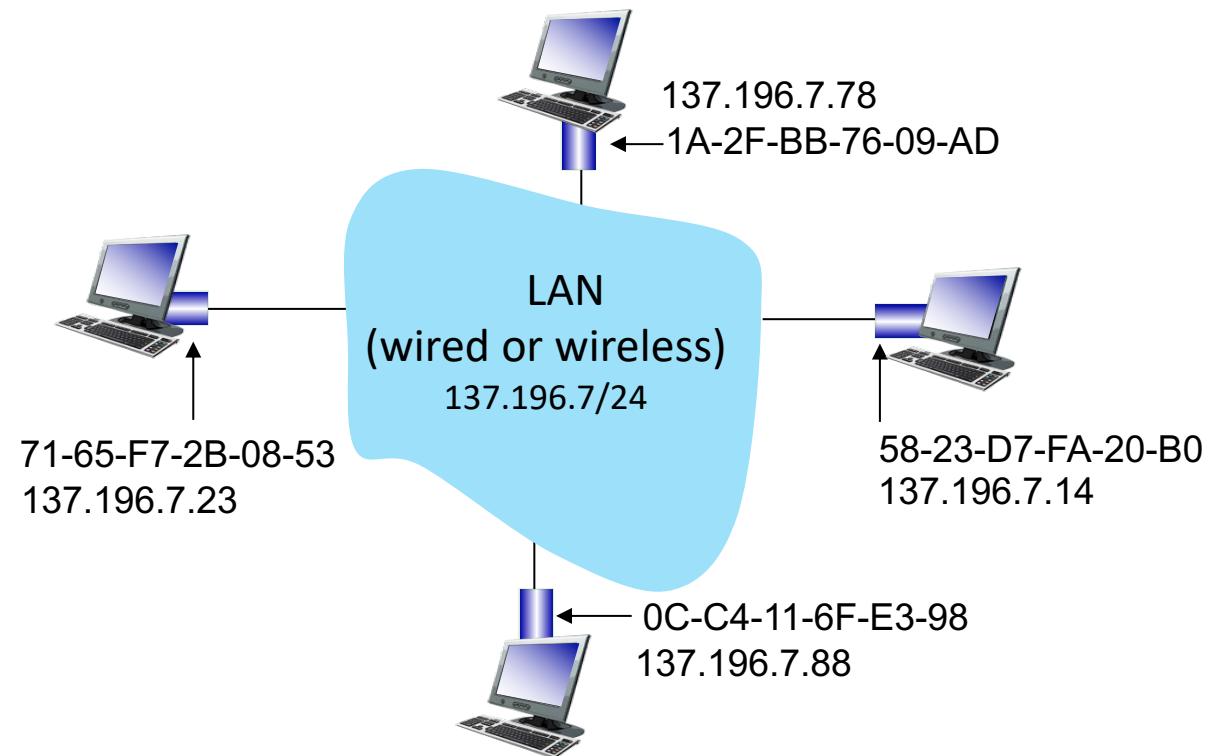
# MAC addresses

- 32-bit IP address:
    - *network-layer* address for interface
    - used for layer 3 (network layer) forwarding
    - e.g.: 128.119.40.136
  - MAC (or LAN or physical or Ethernet) address:
    - function: used “locally” to get frame from one interface to another physically-connected interface (same subnet, in IP-addressing sense)
    - 48-bit MAC address (for most LANs) burned in NIC ROM, also sometimes software settable
    - e.g.: 1A-2F-BB-76-09-AD
- hexadecimal (base 16) notation  
(each “numeral” represents 4 bits)*

# MAC addresses

each interface on LAN

- has unique 48-bit **MAC** address
- has a locally unique 32-bit IP address (as we've seen)

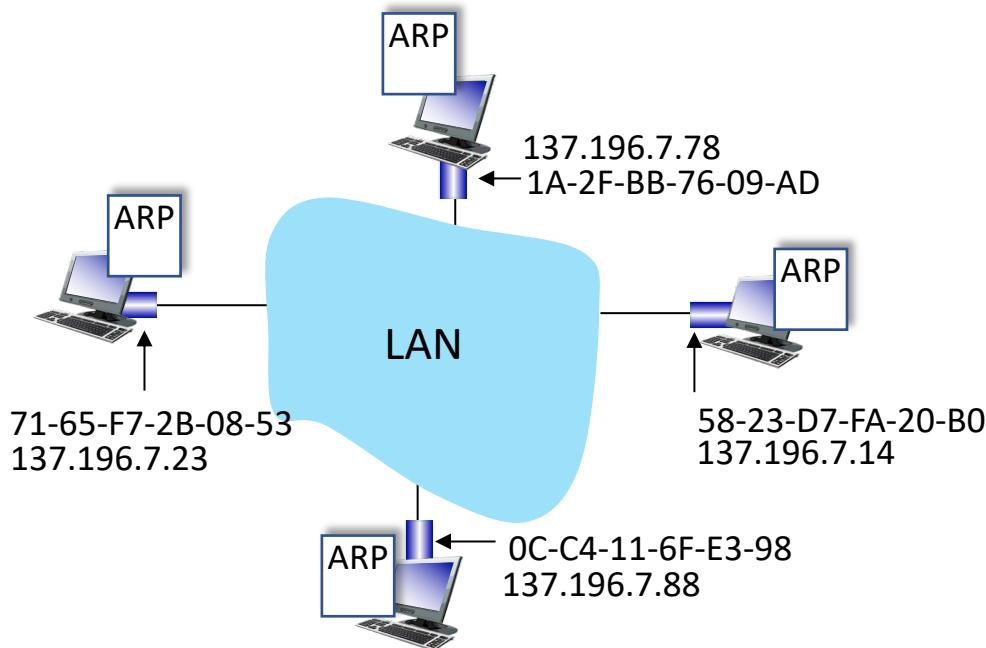


# MAC addresses

- MAC address allocation administered by IEEE
- manufacturer buys portion of MAC address space (to assure uniqueness)
- analogy:
  - MAC address: like Social Security Number
  - IP address: like postal address
- MAC flat address: portability
  - can move interface from one LAN to another
  - recall IP address *not* portable: depends on IP subnet to which node is attached

# ARP: address resolution protocol

*Question:* how to determine interface's MAC address, knowing its IP address?



**ARP table:** each IP node (host, router) on LAN has table

- IP/MAC address mappings for some LAN nodes:  
<IP address; MAC address; TTL>
- TTL (Time To Live): time after which address mapping will be forgotten (typically 20 min)

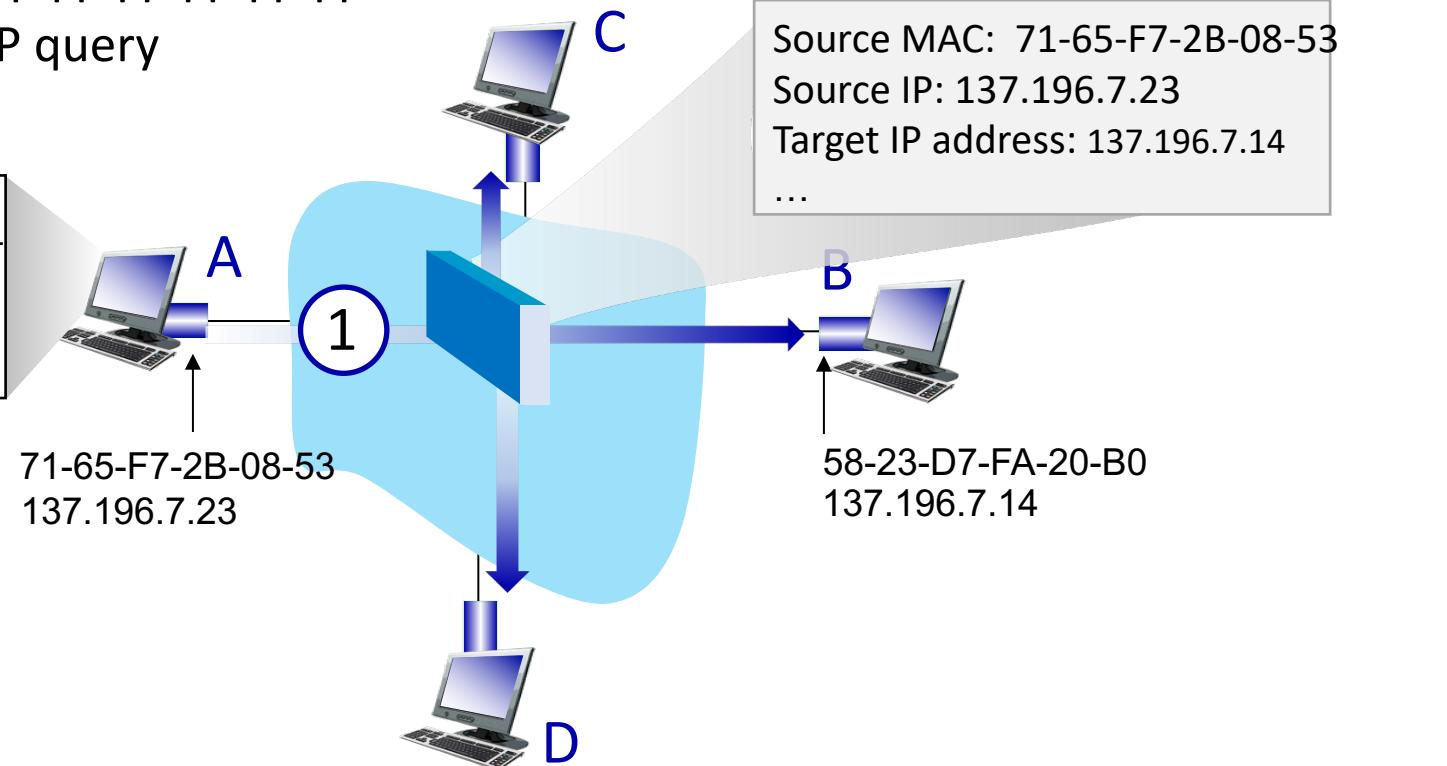
# ARP protocol in action

example: A wants to send datagram to B

- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address

- destination MAC address = FF-FF-FF-FF-FF-FF
  - all nodes on LAN receive ARP query

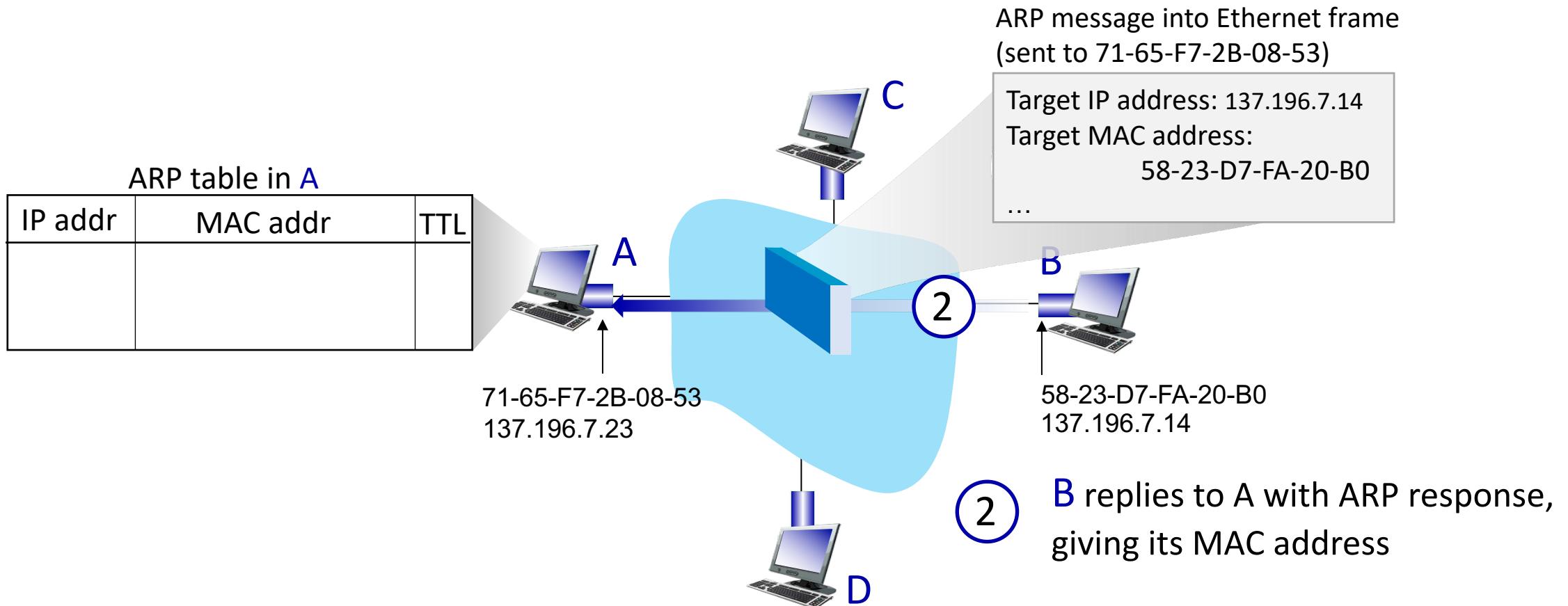
ARP table in A		
IP addr	MAC addr	TTL



# ARP protocol in action

example: A wants to send datagram to B

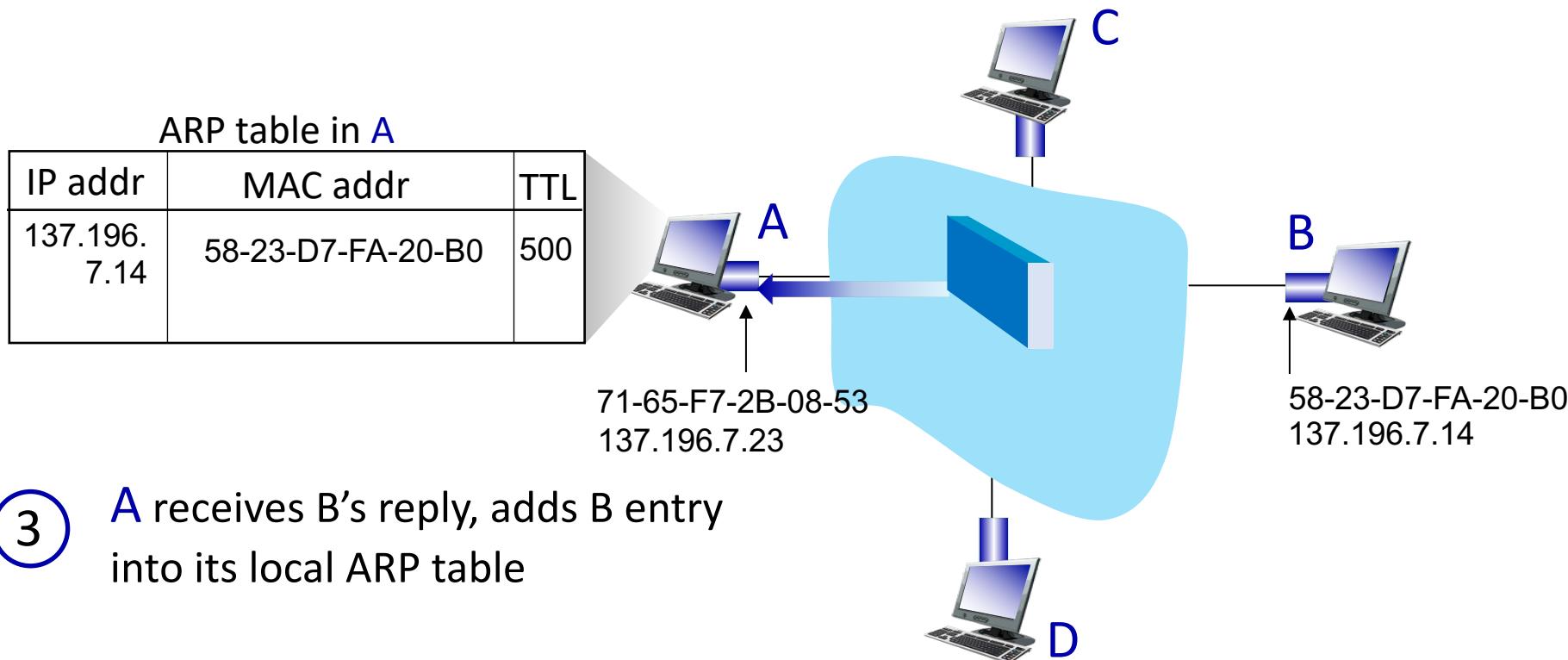
- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address



# ARP protocol in action

example: A wants to send datagram to B

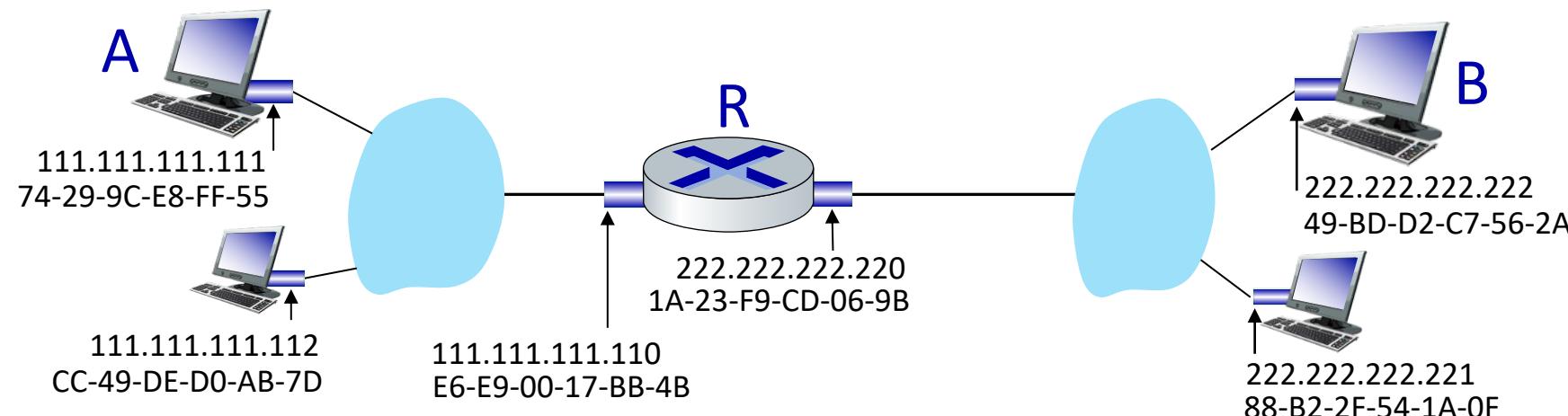
- B's MAC address not in A's ARP table, so A uses ARP to find B's MAC address



# Routing to another subnet: addressing

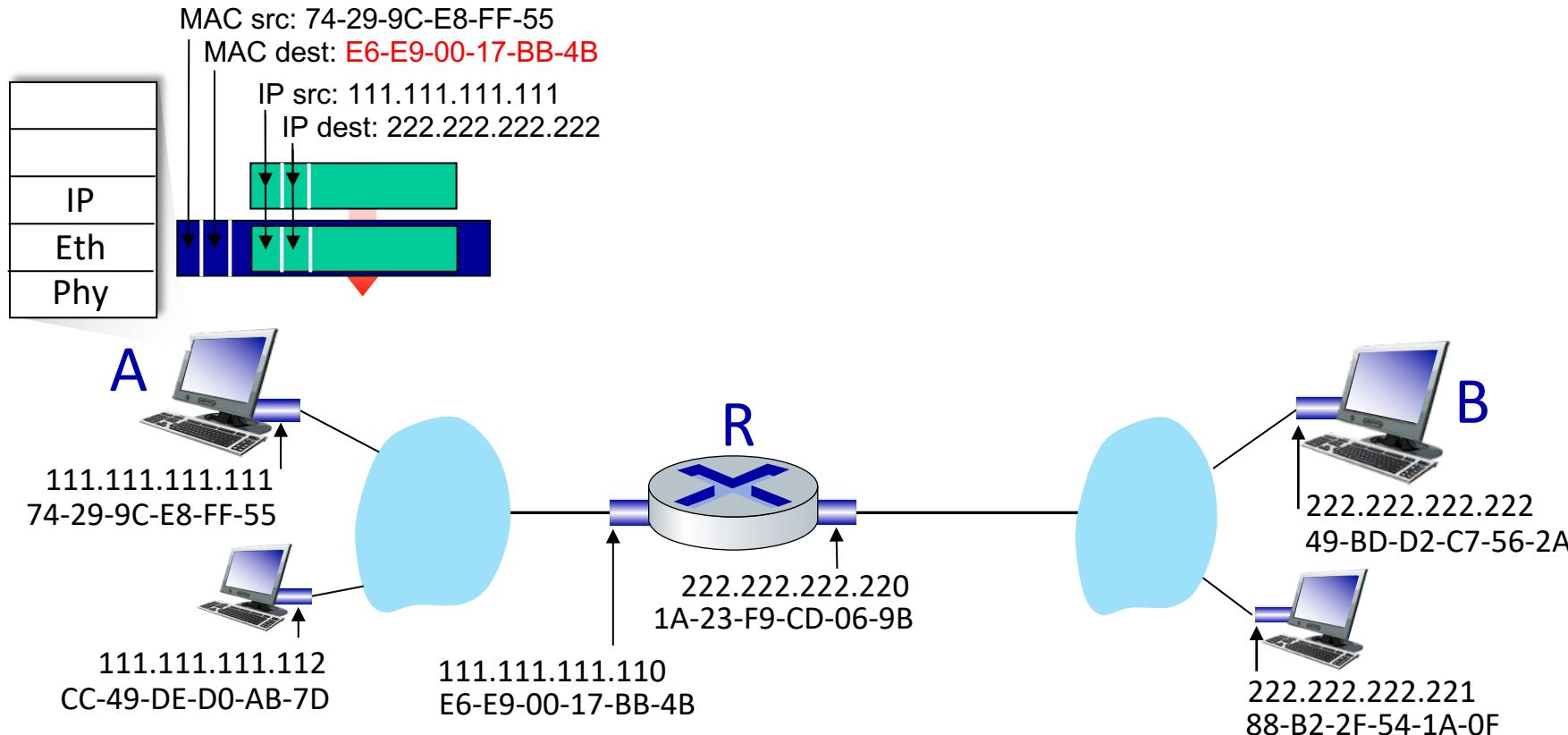
walkthrough: sending a datagram from *A* to *B* via *R*

- focus on addressing – at IP (datagram) and MAC layer (frame) levels
- assume that:
  - A knows B's IP address
  - A knows IP address of first hop router, R (how?)
  - A knows R's MAC address (how?)



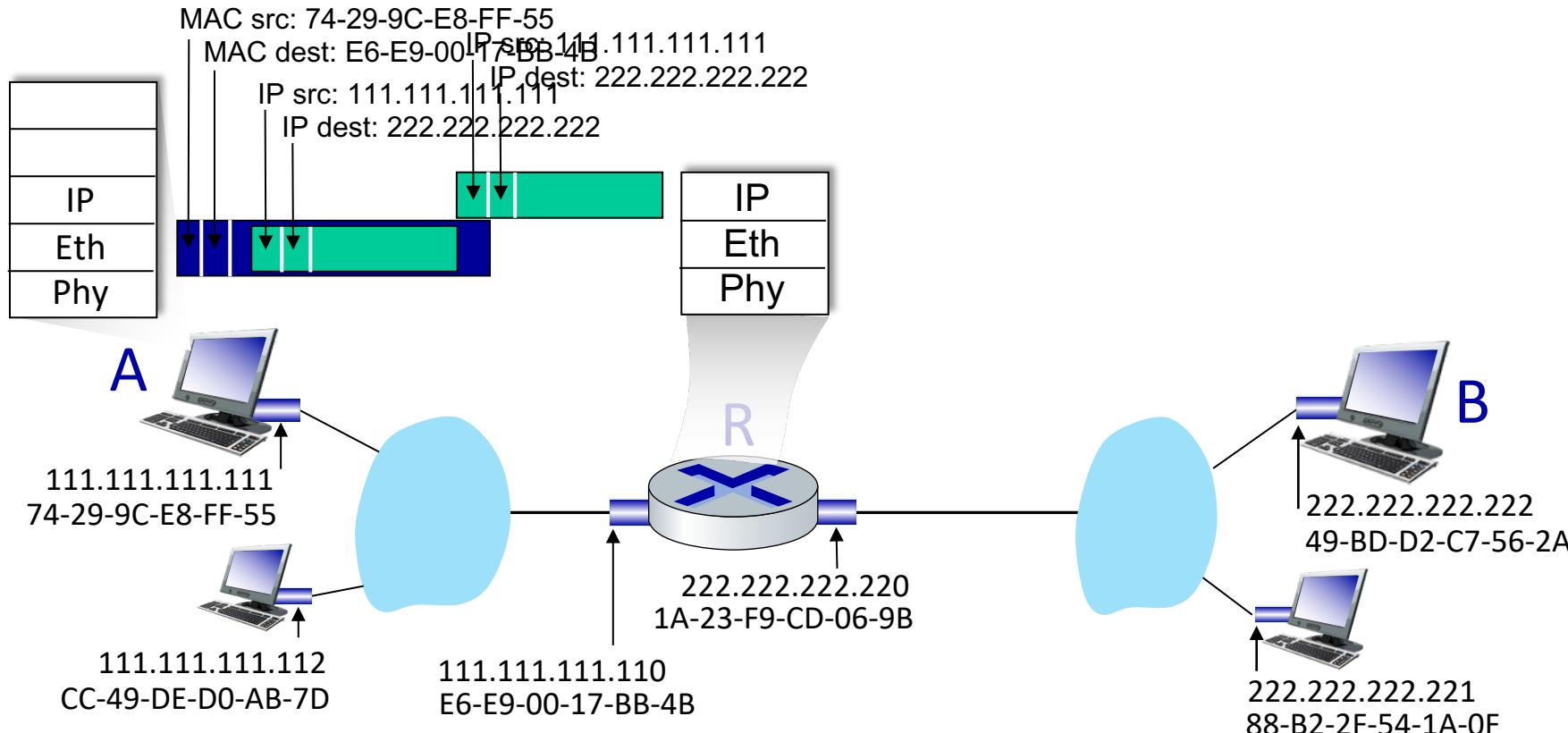
# Routing to another subnet: addressing

- A creates IP datagram with IP source A, destination B
- A creates link-layer frame containing A-to-B IP datagram
  - R's MAC address is frame's destination



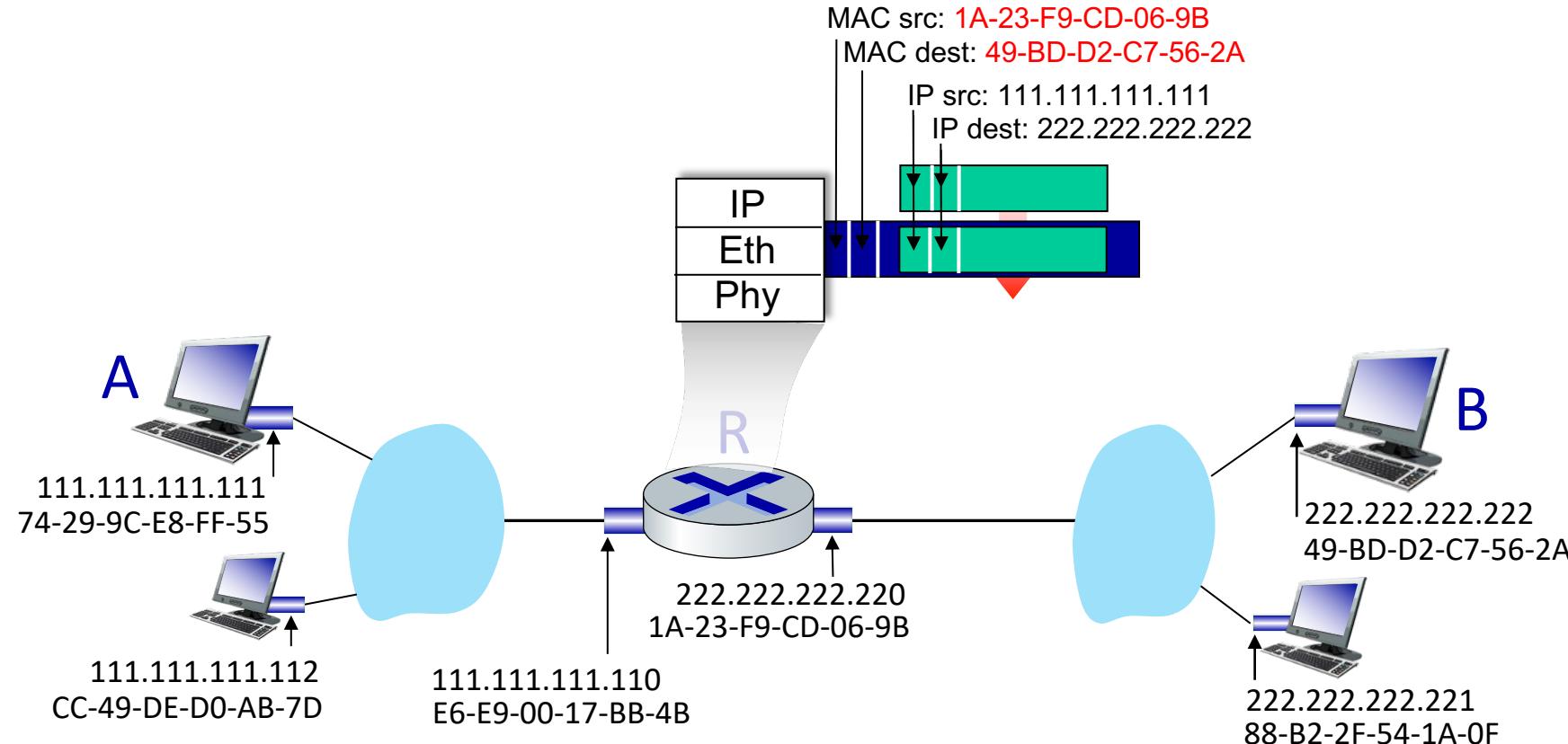
# Routing to another subnet: addressing

- frame sent from A to R
- frame received at R, datagram removed, passed up to IP



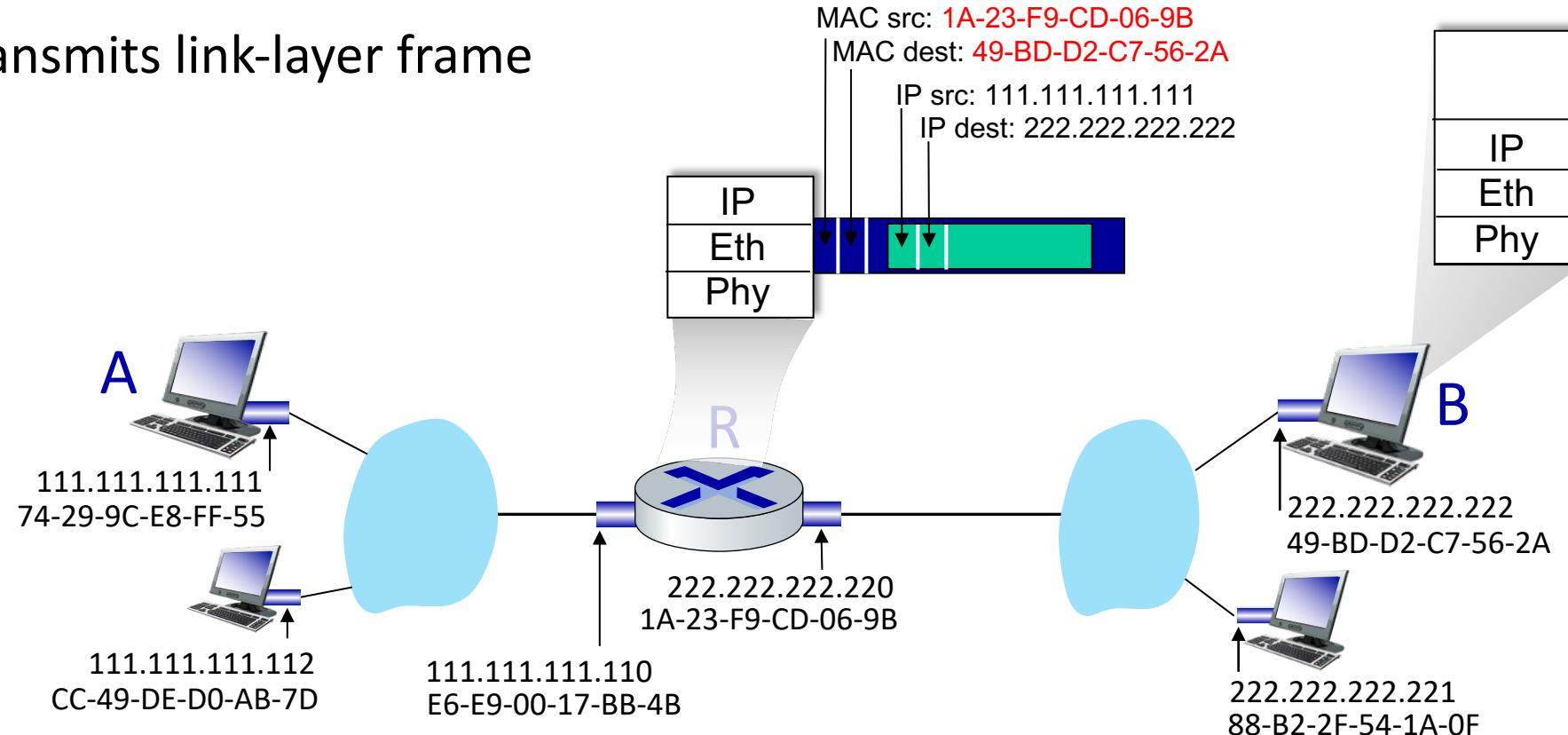
# Routing to another subnet: addressing

- R determines outgoing interface, passes datagram with IP source A, destination B to link layer
- R creates link-layer frame containing A-to-B IP datagram. Frame destination address: B's MAC address



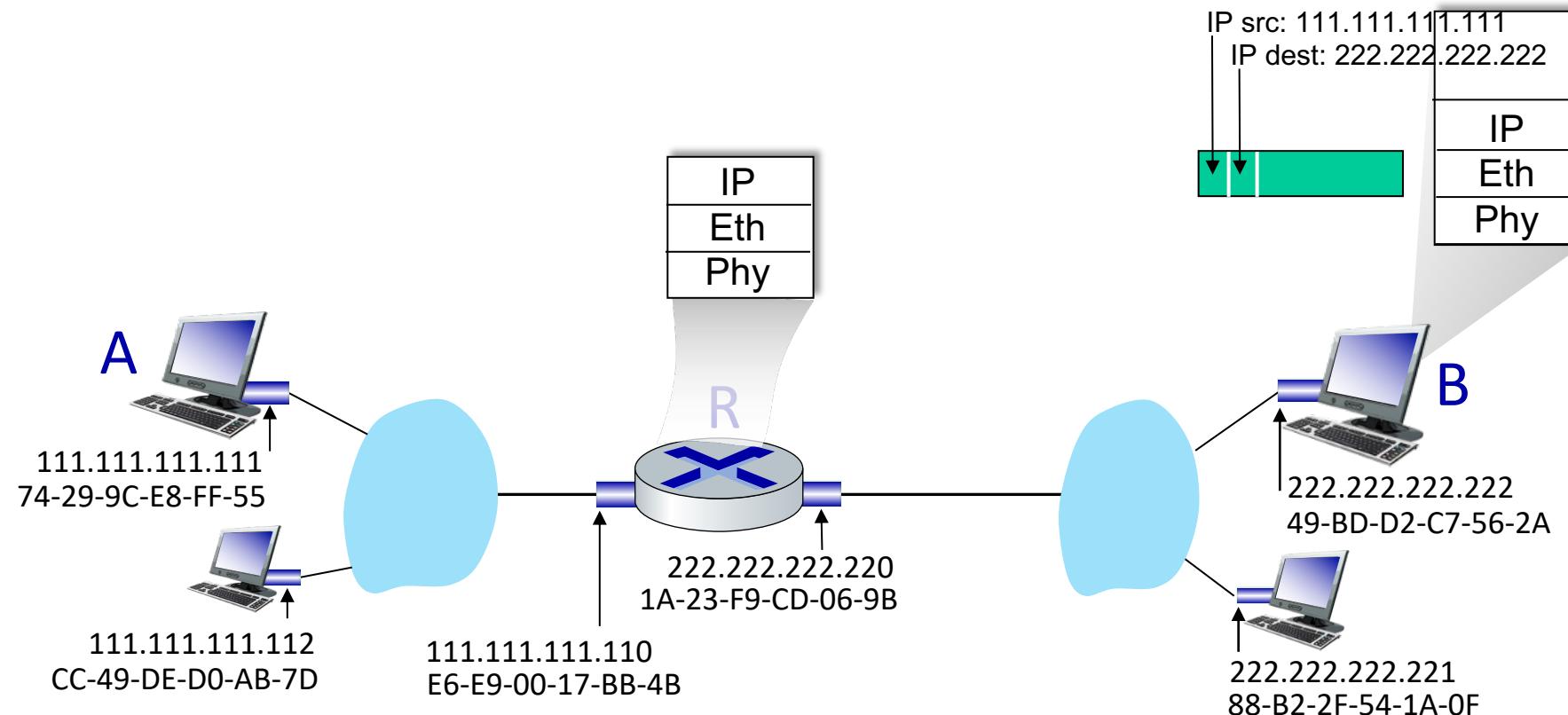
# Routing to another subnet: addressing

- R determines outgoing interface, passes datagram with IP source A, destination B to link layer
- R creates link-layer frame containing A-to-B IP datagram. Frame destination address: B's MAC address
- transmits link-layer frame



# Routing to another subnet: addressing

- B receives frame, extracts IP datagram destination B
- B passes datagram up protocol stack to IP



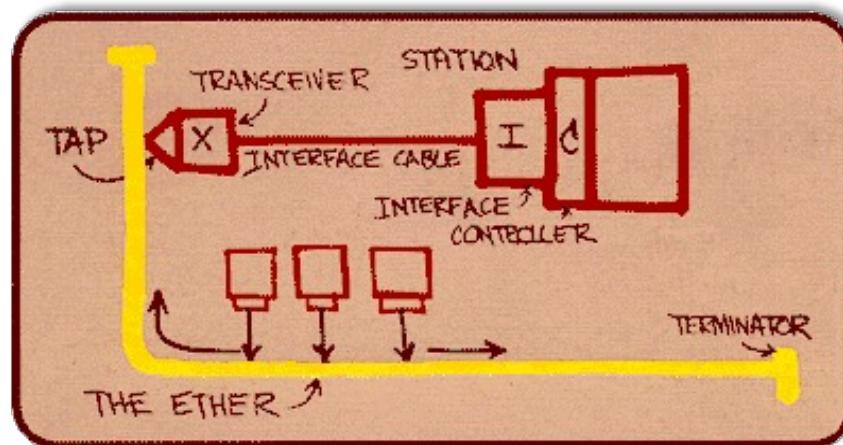
# Link layer, LANs: roadmap

- introduction
  - error detection, correction
  - multiple access protocols
  - **LANs**
    - addressing, ARP
    - **Ethernet**
    - switches
    - VLANs
  - link virtualization: MPLS
  - data center networking
- 
- a day in the life of a web request

# Ethernet

“dominant” wired LAN technology:

- first widely used LAN technology
- simpler, cheap
- kept up with speed race: 10 Mbps – 400 Gbps
- single chip, multiple speeds (e.g., Broadcom BCM5761)



*Metcalfe's Ethernet  
sketch*

# Ethernet: physical topology

- **bus:** popular through mid 90s
  - all nodes in same collision domain (can collide with each other)
- **switched:** prevails today
  - active link-layer 2 *switch* in center
  - each “spoke” runs a (separate) Ethernet protocol (nodes do not collide with each other)



# Ethernet frame structure

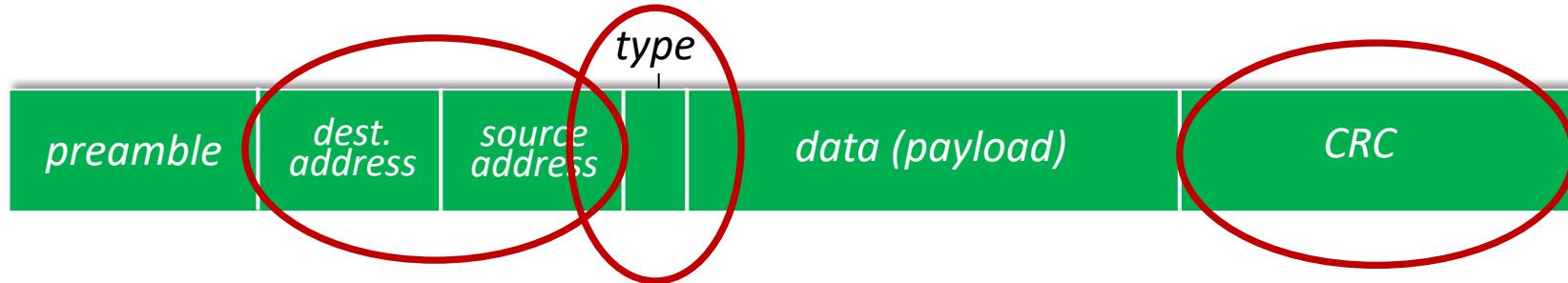
sending interface encapsulates IP datagram (or other network layer protocol packet) in **Ethernet frame**



*preamble*:

- used to synchronize receiver, sender clock rates
- 7 bytes of 10101010 followed by one byte of 10101011

# Ethernet frame structure (more)



- **addresses**: 6 byte source, destination MAC addresses
  - if adapter receives frame with matching destination address, or with broadcast address (e.g., ARP packet), it passes data in frame to network layer protocol
  - otherwise, adapter discards frame
- **type**: indicates higher layer protocol
  - mostly IP but others possible, e.g., Novell IPX, AppleTalk
  - used to demultiplex up at receiver
- **CRC**: cyclic redundancy check at receiver
  - error detected: frame is dropped

# Ethernet: unreliable, connectionless

- **connectionless:** no handshaking between sending and receiving NICs
- **unreliable:** receiving NIC doesn't send ACKs or NAKs to sending NIC
  - data in dropped frames recovered only if initial sender uses higher layer rdt (e.g., TCP), otherwise dropped data lost
- Ethernet's MAC protocol: unslotted **CSMA/CD with binary backoff**

# Link layer, LANs: roadmap

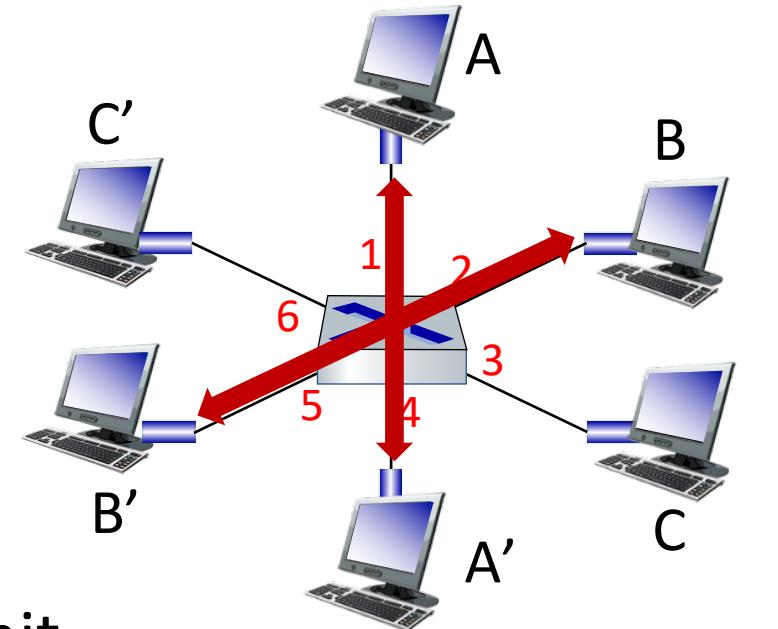
- introduction
  - error detection, correction
  - multiple access protocols
  - **LANs**
    - addressing, ARP
    - Ethernet
    - **switches**
    - VLANs
  - link virtualization: MPLS
  - data center networking
- 
- a day in the life of a web request

# Ethernet switch

- Switch is a **link-layer** device: takes an *active* role
  - store, forward Ethernet frames
  - examine incoming frame's MAC address, *selectively* forward frame to one-or-more outgoing links when frame is to be forwarded on segment, uses CSMA/CD to access segment
- **transparent**: hosts *unaware* of presence of switches
- **plug-and-play, self-learning**
  - switches do not need to be configured

# Switch: multiple simultaneous transmissions

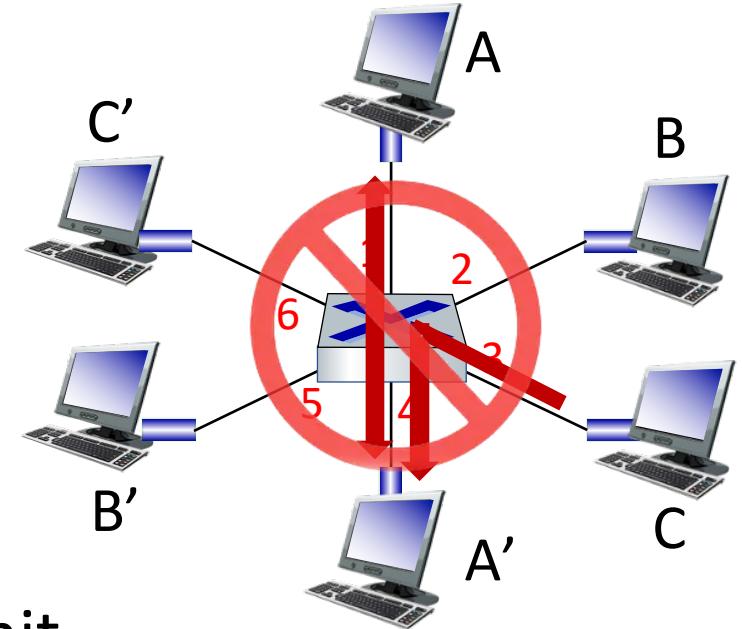
- hosts have dedicated, direct connection to switch
- switches buffer packets
- Ethernet protocol used on *each* incoming link, so:
  - no collisions; full duplex
  - each link is its own collision domain
- **switching:** A-to-A' and B-to-B' can transmit simultaneously, without collisions



switch with six  
interfaces (1,2,3,4,5,6)

# Switch: multiple simultaneous transmissions

- hosts have dedicated, direct connection to switch
- switches buffer packets
- Ethernet protocol used on *each* incoming link, so:
  - no collisions; full duplex
  - each link is its own collision domain
- **switching:** A-to-A' and B-to-B' can transmit simultaneously, without collisions
  - but A-to-A' and C to A' can *not* happen simultaneously



switch with six  
interfaces (1,2,3,4,5,6)

# Switch forwarding table

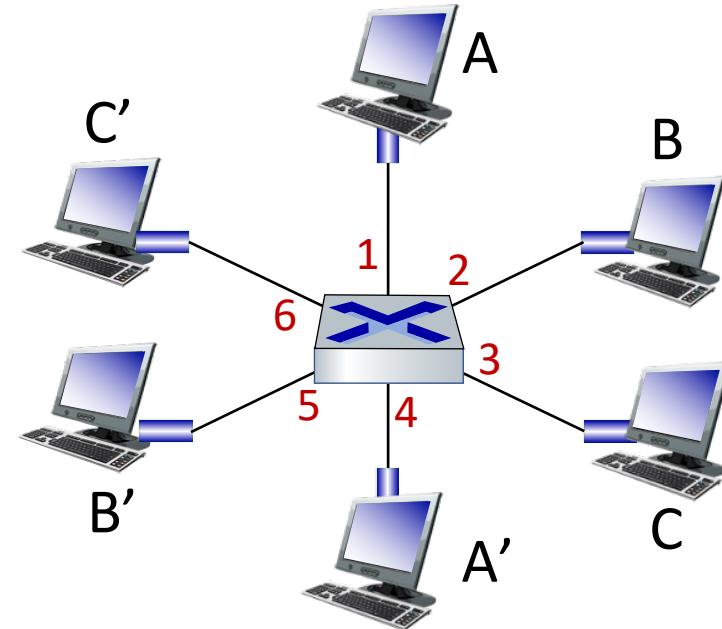
**Q:** how does switch know A' reachable via interface 4, B' reachable via interface 5?

**A:** each switch has a **switch table**, each entry:

- (MAC address of host, interface to reach host, time stamp)
- looks like a routing table!

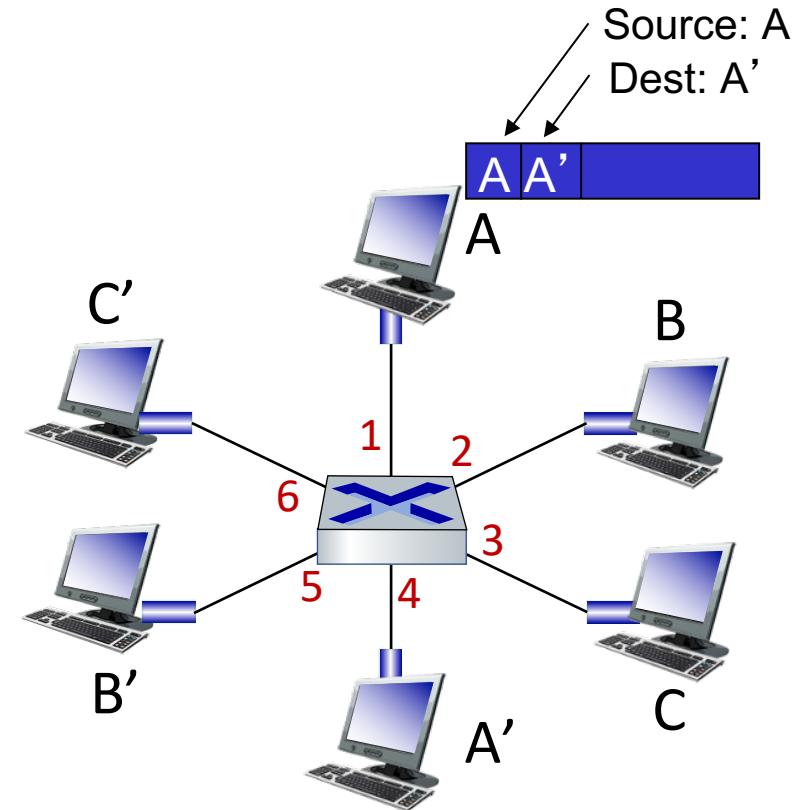
**Q:** how are entries created, maintained in switch table?

- something like a routing protocol?



# Switch: self-learning

- switch *learns* which hosts can be reached through which interfaces
  - when frame received, switch “learns” location of sender: incoming LAN segment
  - records sender/location pair in switch table



MAC addr	interface	TTL
A	1	60

*Switch table  
(initially empty)*

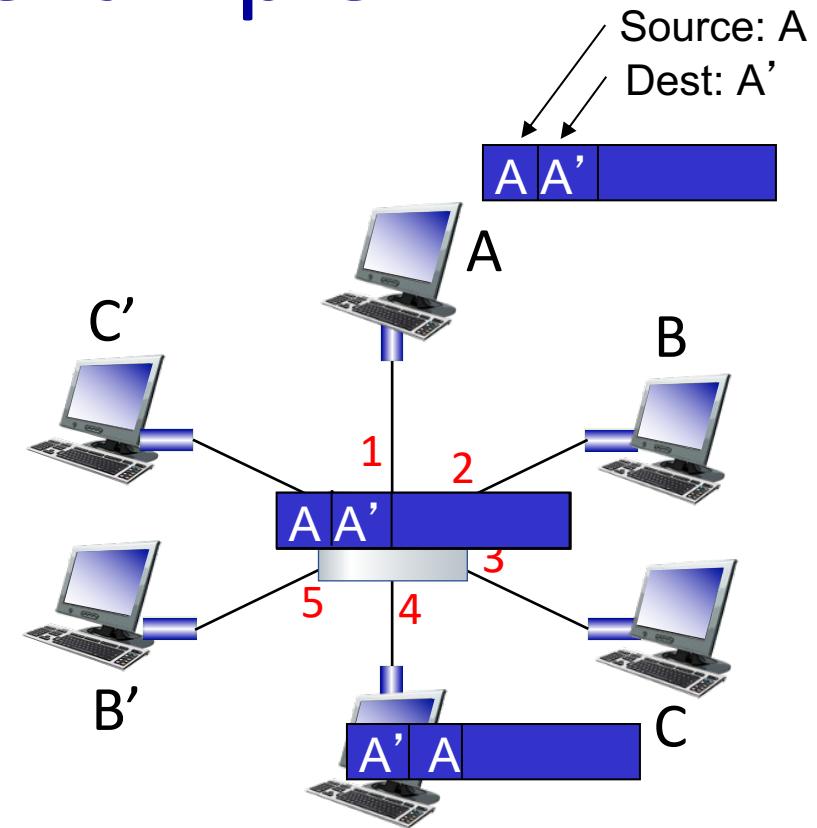
# Switch: frame filtering/forwarding

when frame received at switch:

1. record incoming link, MAC address of sending host
2. index switch table using MAC destination address
3. if entry found for destination
  - then {
    - if destination on segment from which frame arrived
      - then drop frame
      - else forward frame on interface indicated by entry
  - }
- else flood /\* forward on all interfaces except arriving interface \*/

# Self-learning, forwarding: example

- frame destination, A', location unknown: **flood**
- destination A location known: **selectively send on just one link**



MAC addr	interface	TTL
A	1	60
A'	4	60

*switch table  
(initially empty)*

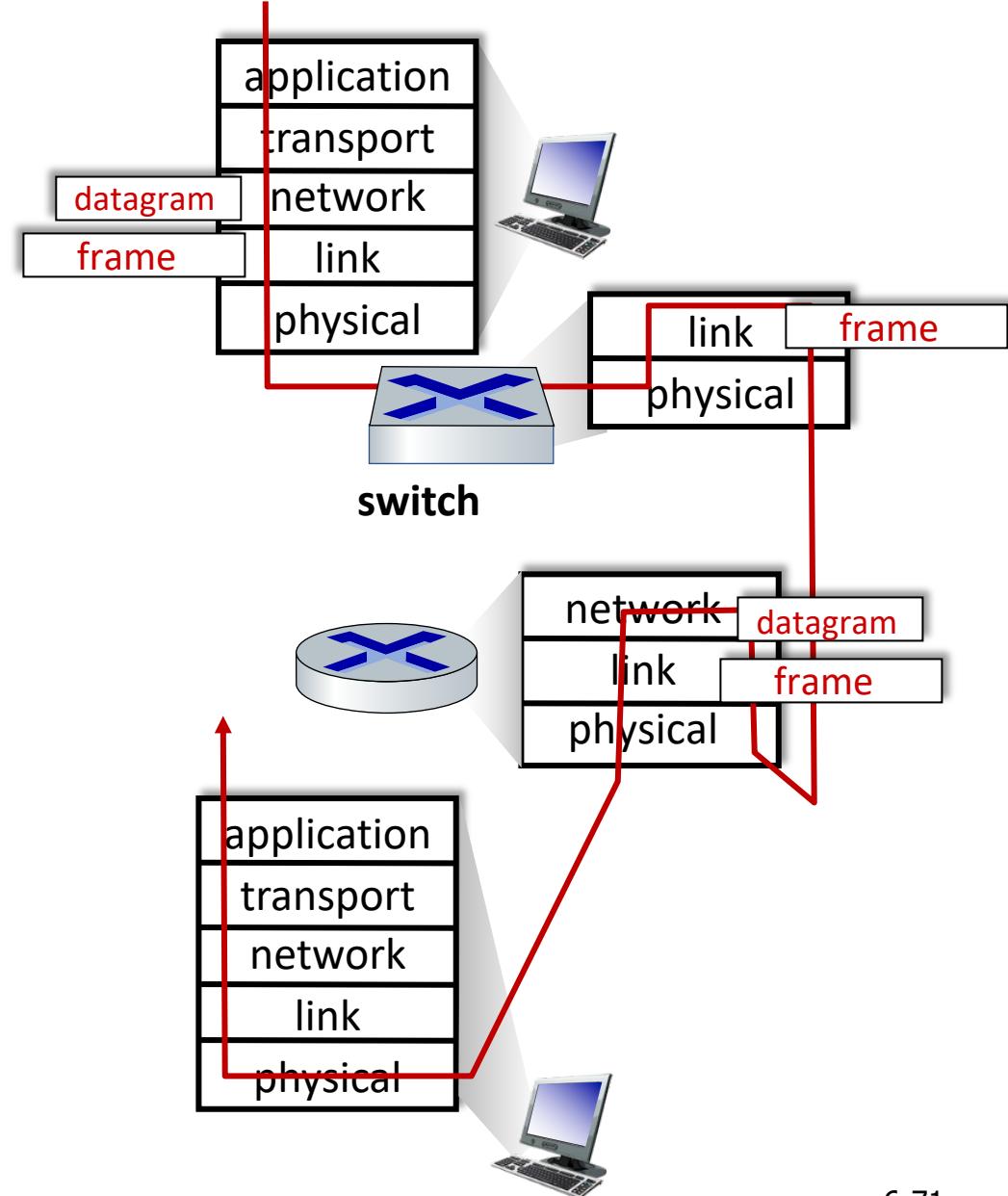
# Switches vs. routers

both are store-and-forward:

- *routers*: network-layer devices (examine network-layer headers)
- *switches*: link-layer devices (examine link-layer headers)

both have forwarding tables:

- *routers*: compute tables using routing algorithms, IP addresses
- *switches*: learn forwarding table using flooding, learning, MAC addresses



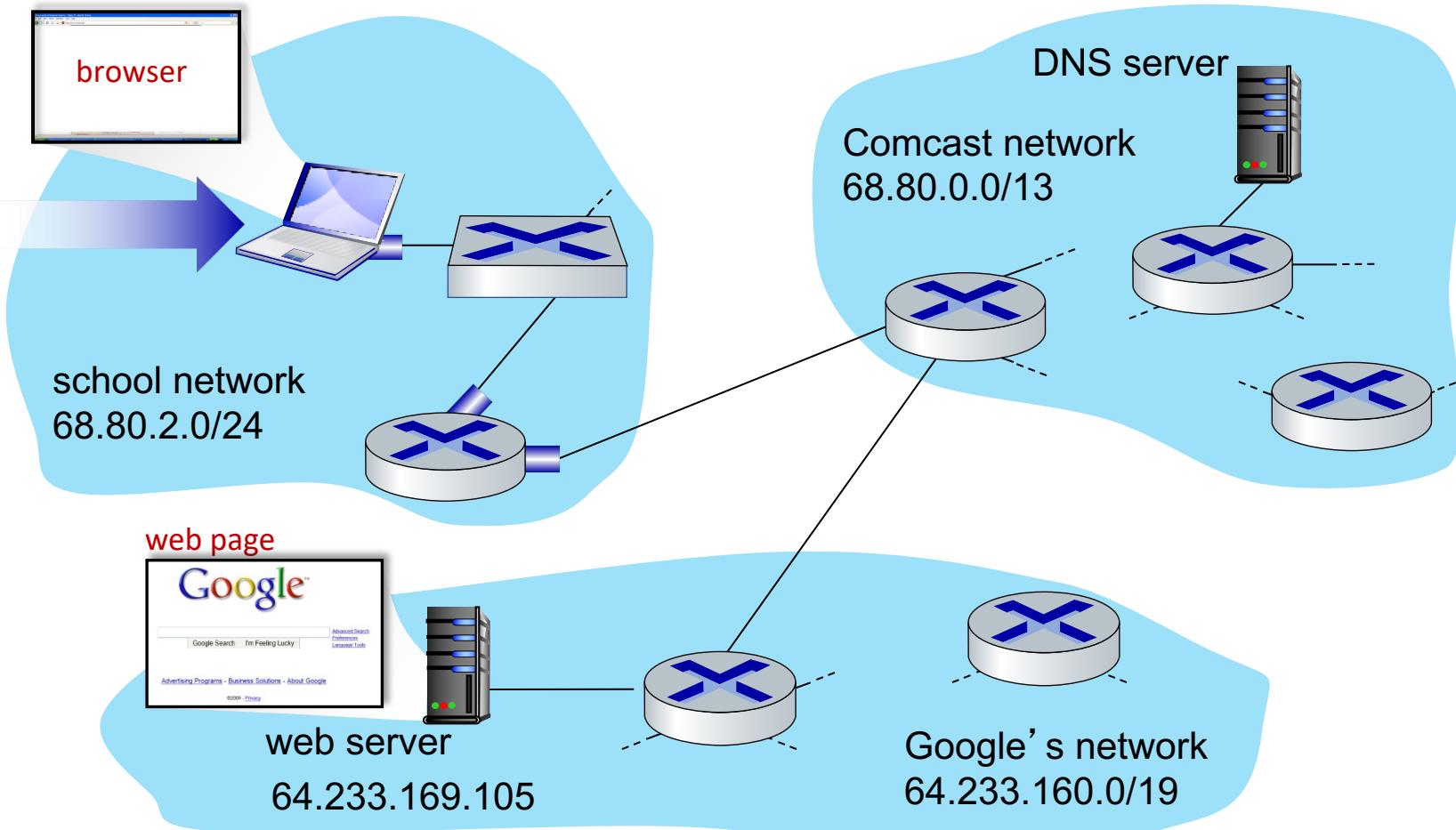
# Link layer, LANs: roadmap

- introduction
- error detection, correction
- multiple access protocols
- LANs
  - addressing, ARP
  - Ethernet
  - switches
  - VLANs
- link virtualization: MPLS
- data center networking



- a day in the life of a web request

# A day in the life: scenario

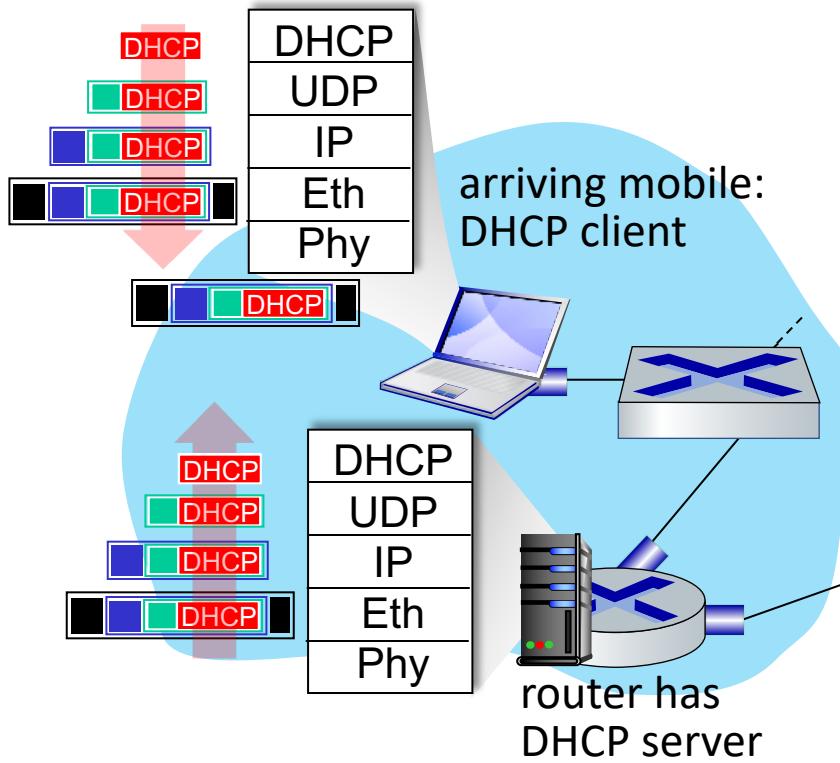


scenario:

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

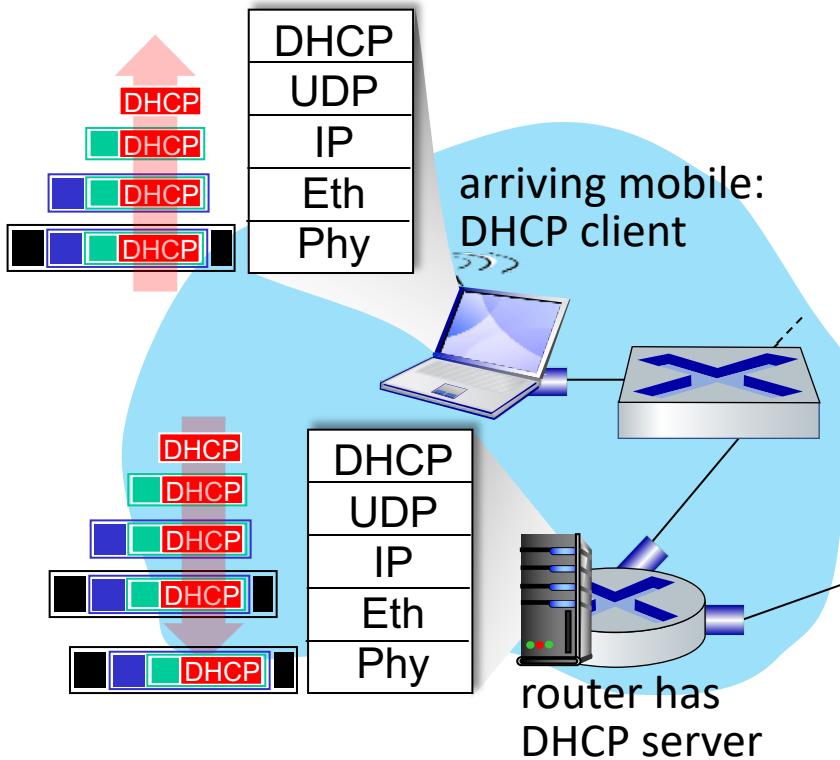
*Sounds simple!* !

# 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

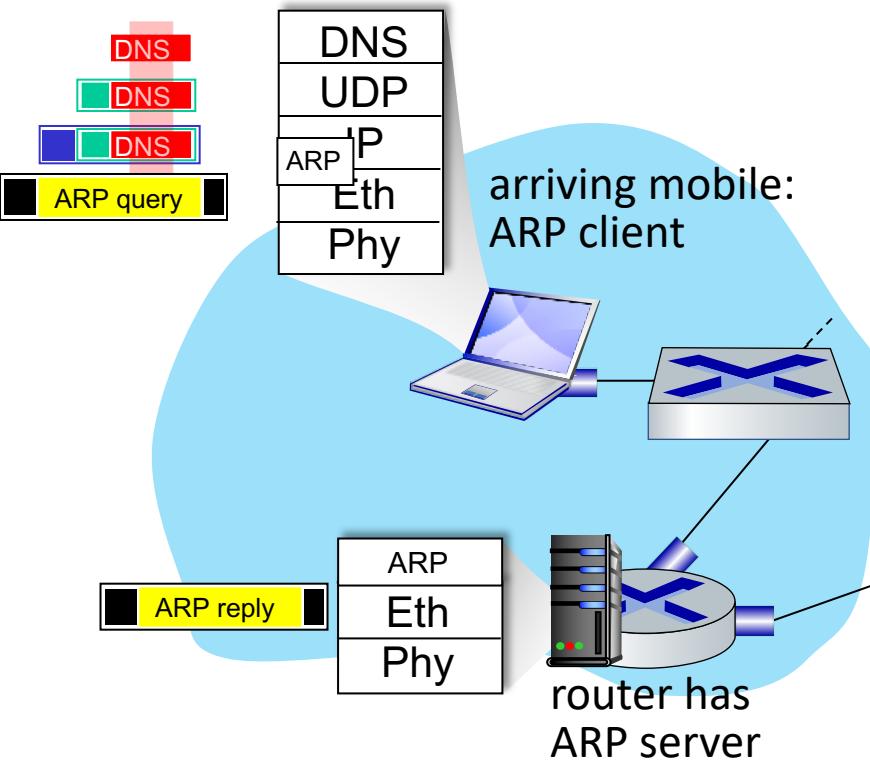
# A day in the life: connecting to the Internet



- 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

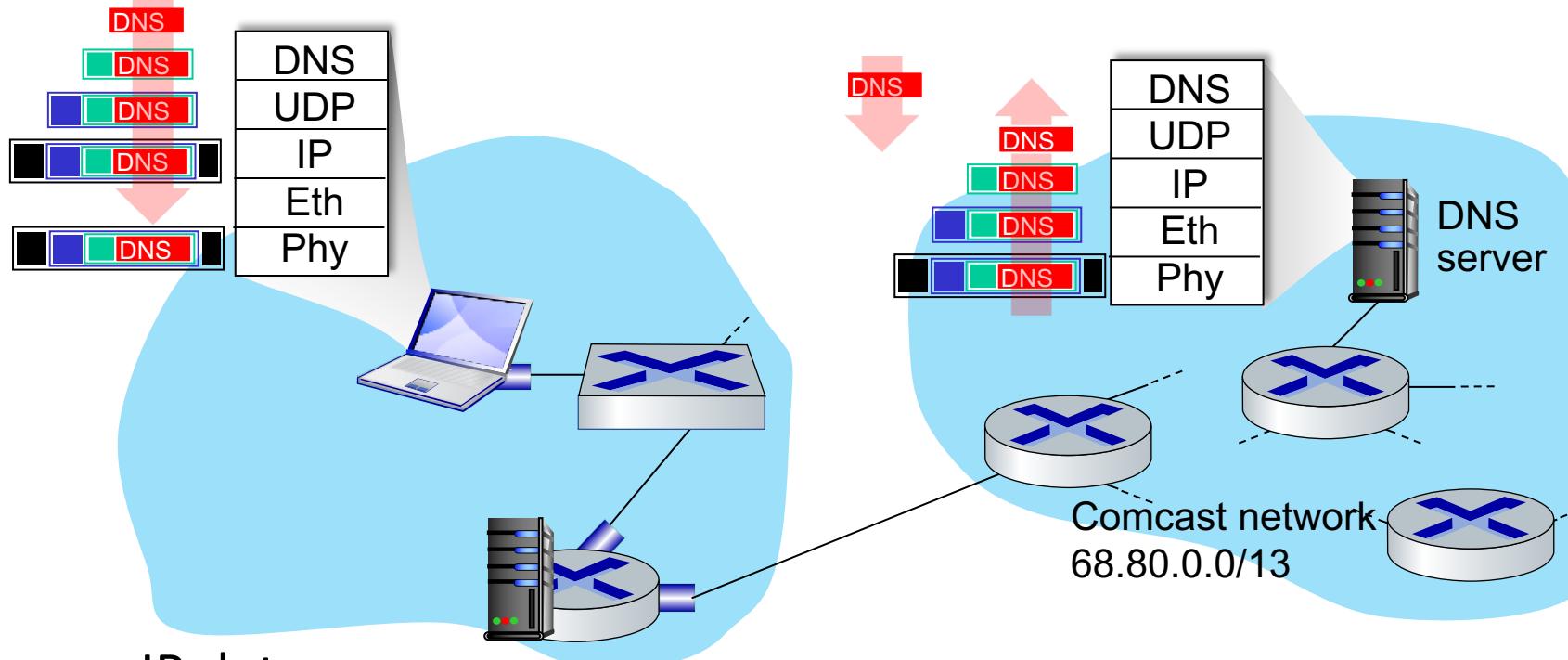
*Client now has IP address, knows name & addr of DNS server, IP address of its first-hop router*

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



- before sending **HTTP** request, need IP address of [www.google.com](http://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

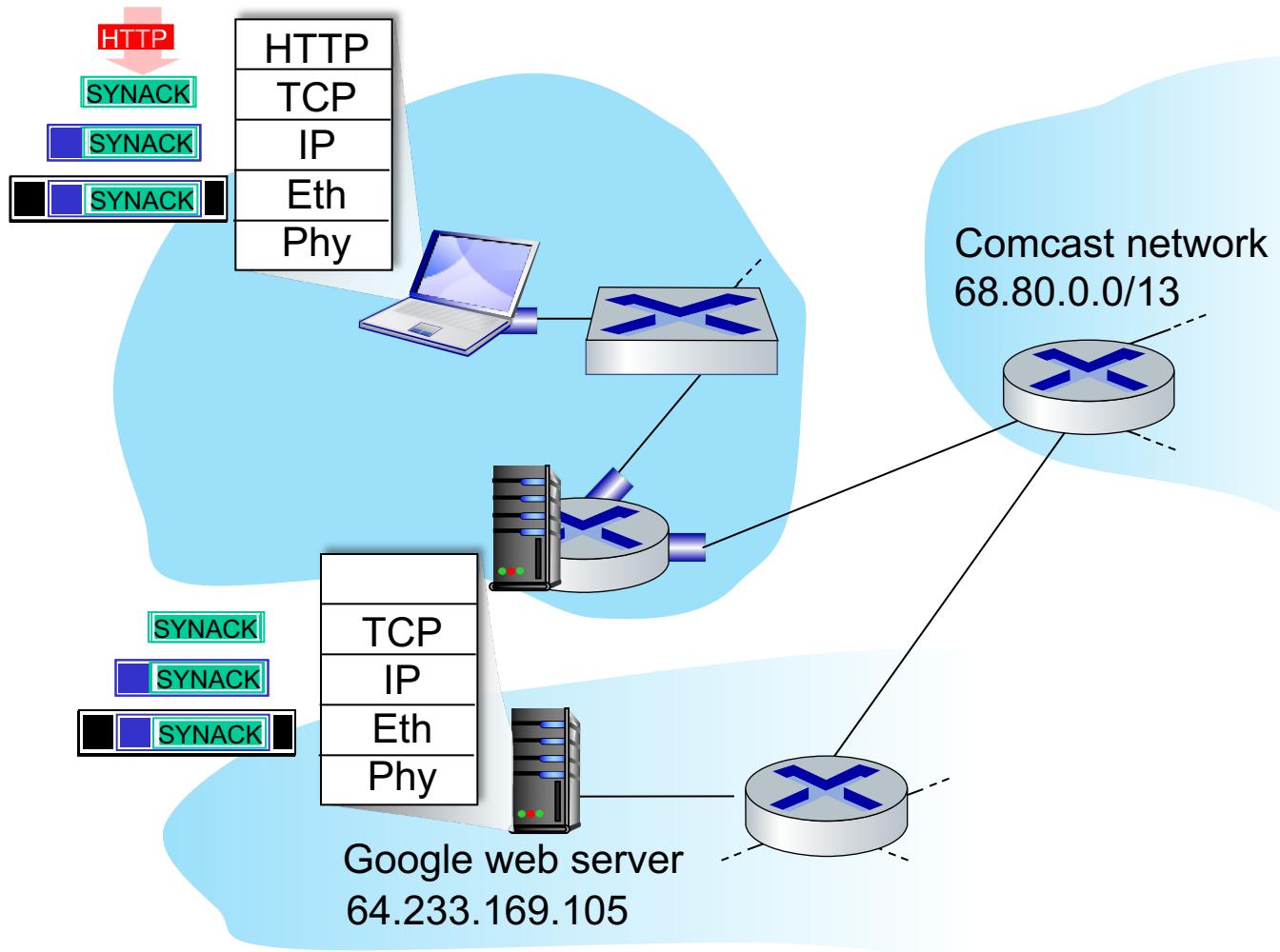
# A day in the life... using DNS



- IP datagram containing DNS query forwarded via LAN switch from client to 1<sup>st</sup> 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

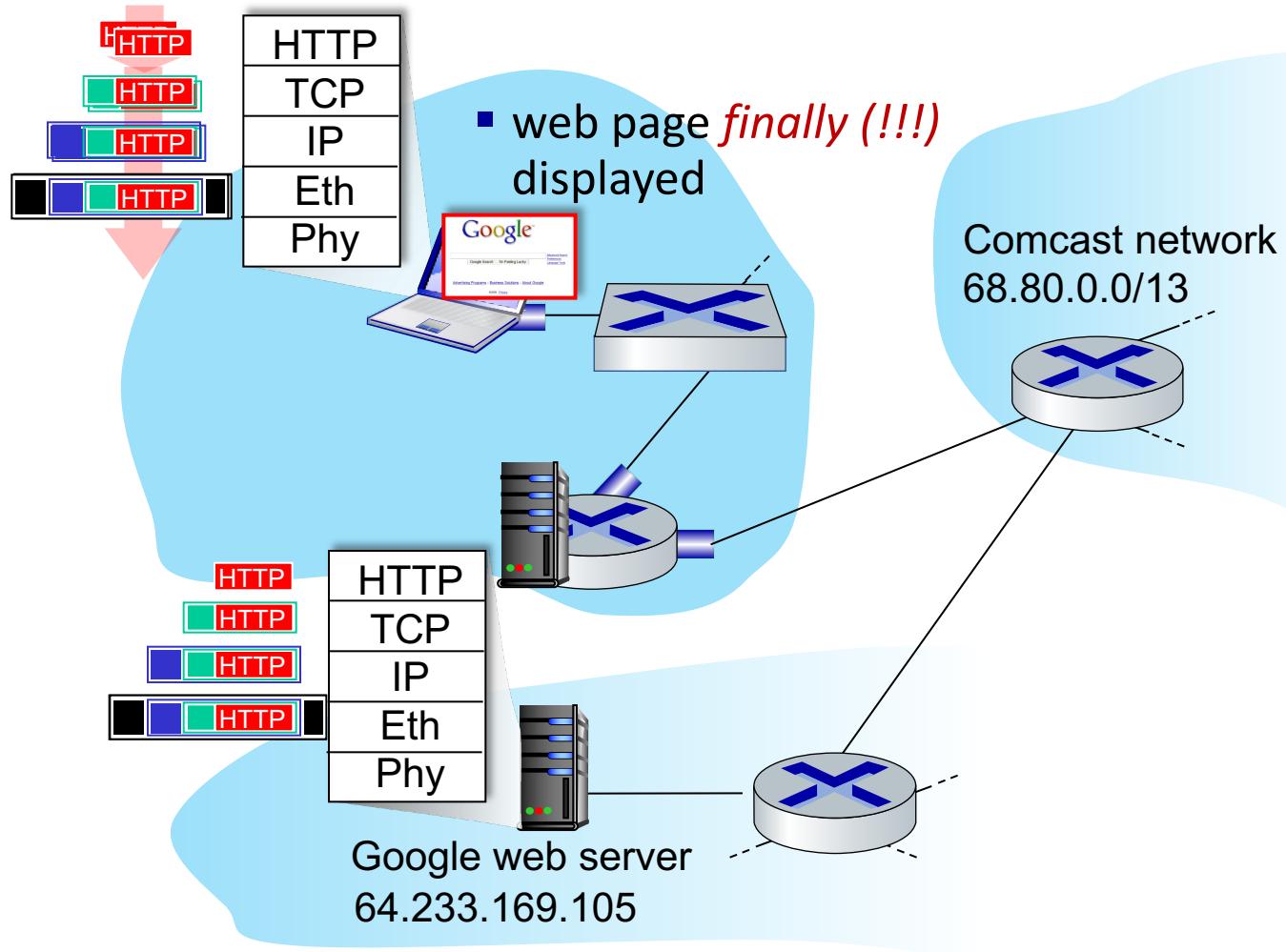
- demuxed to DNS
- DNS replies to client with IP address of [www.google.com](http://www.google.com)

# 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](http://www.google.com)
- web server responds with **HTTP reply** (containing web page)
- IP datagram containing HTTP reply routed back to client

# Chapter 7 outline

- **Introduction**

## Wireless

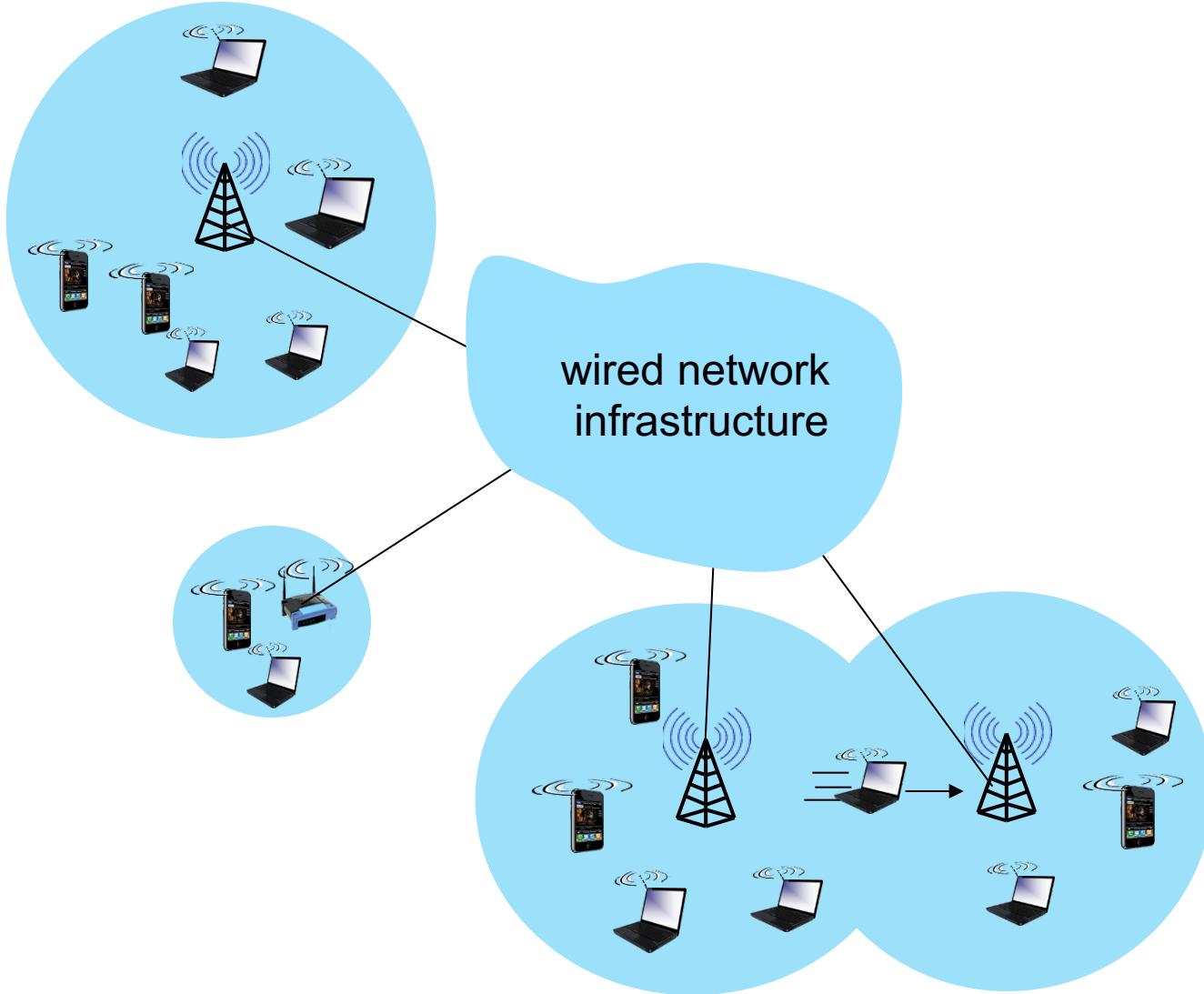
- Wireless Links and network characteristics
- WiFi: 802.11 wireless LANs
- Cellular networks: 4G and 5G



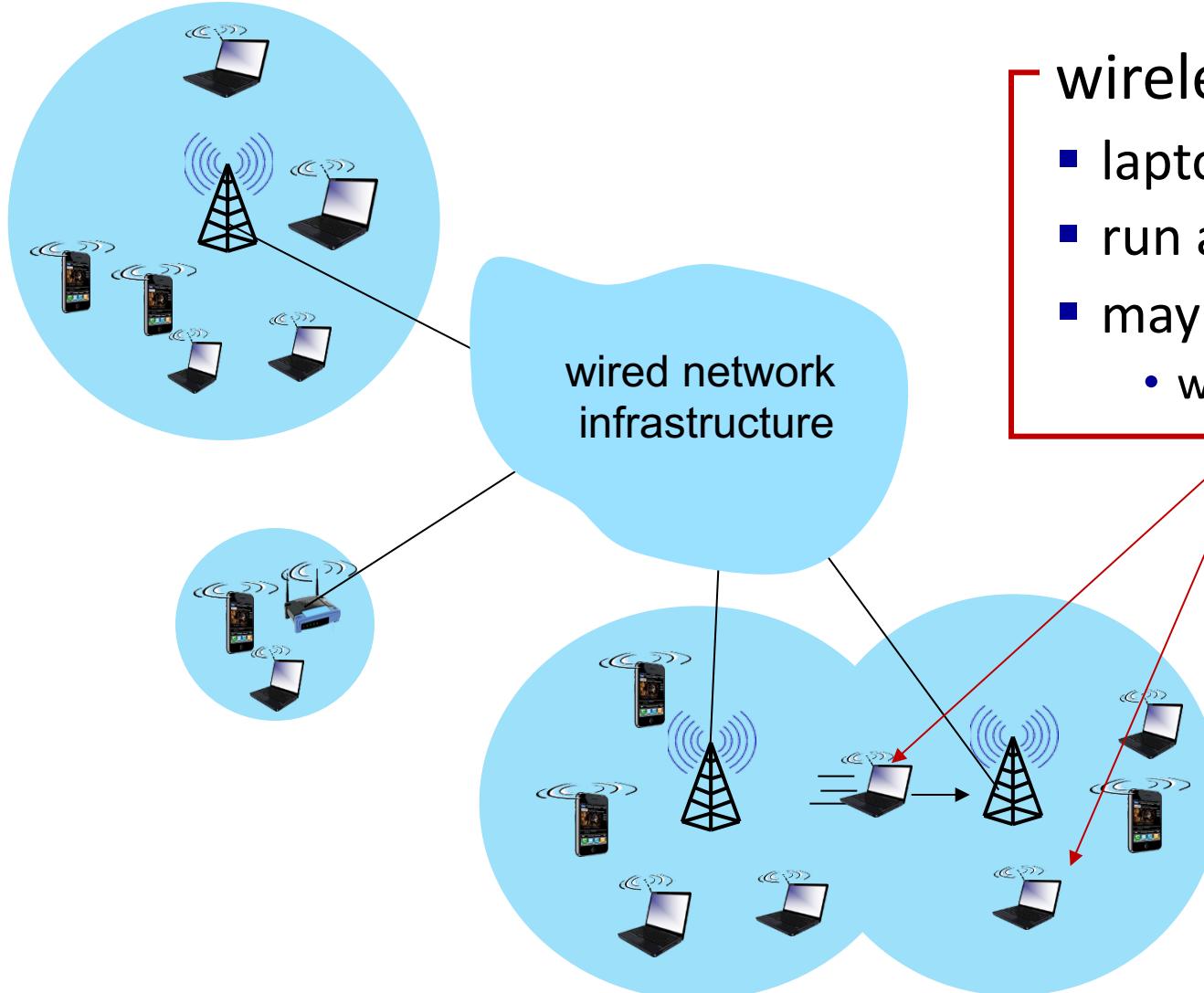
## Mobility

- Mobility management: principles
- Mobility management: practice
  - 4G/5G networks
  - Mobile IP
- Mobility: impact on higher-layer protocols

# Elements of a wireless network



# Elements of a wireless network

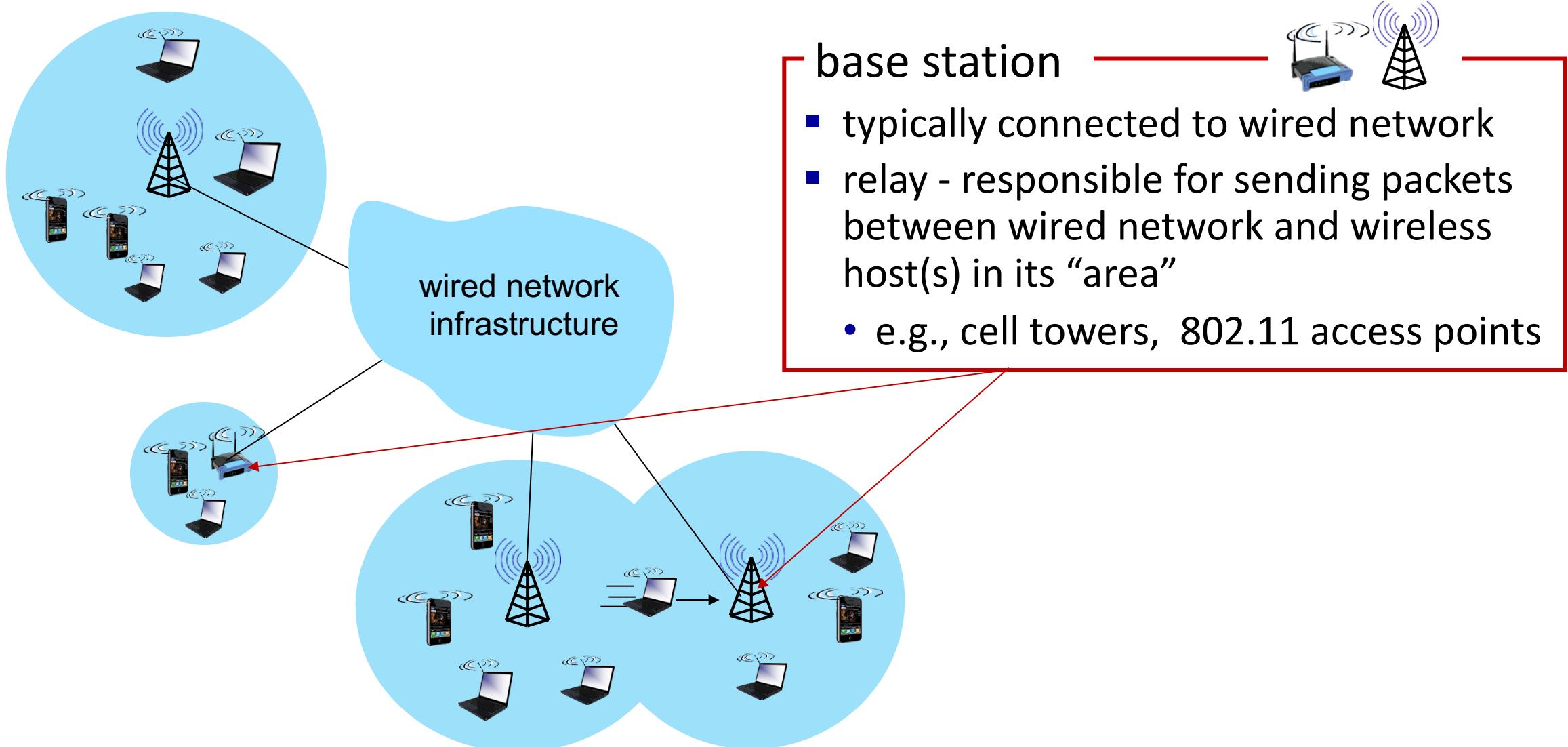


## wireless hosts

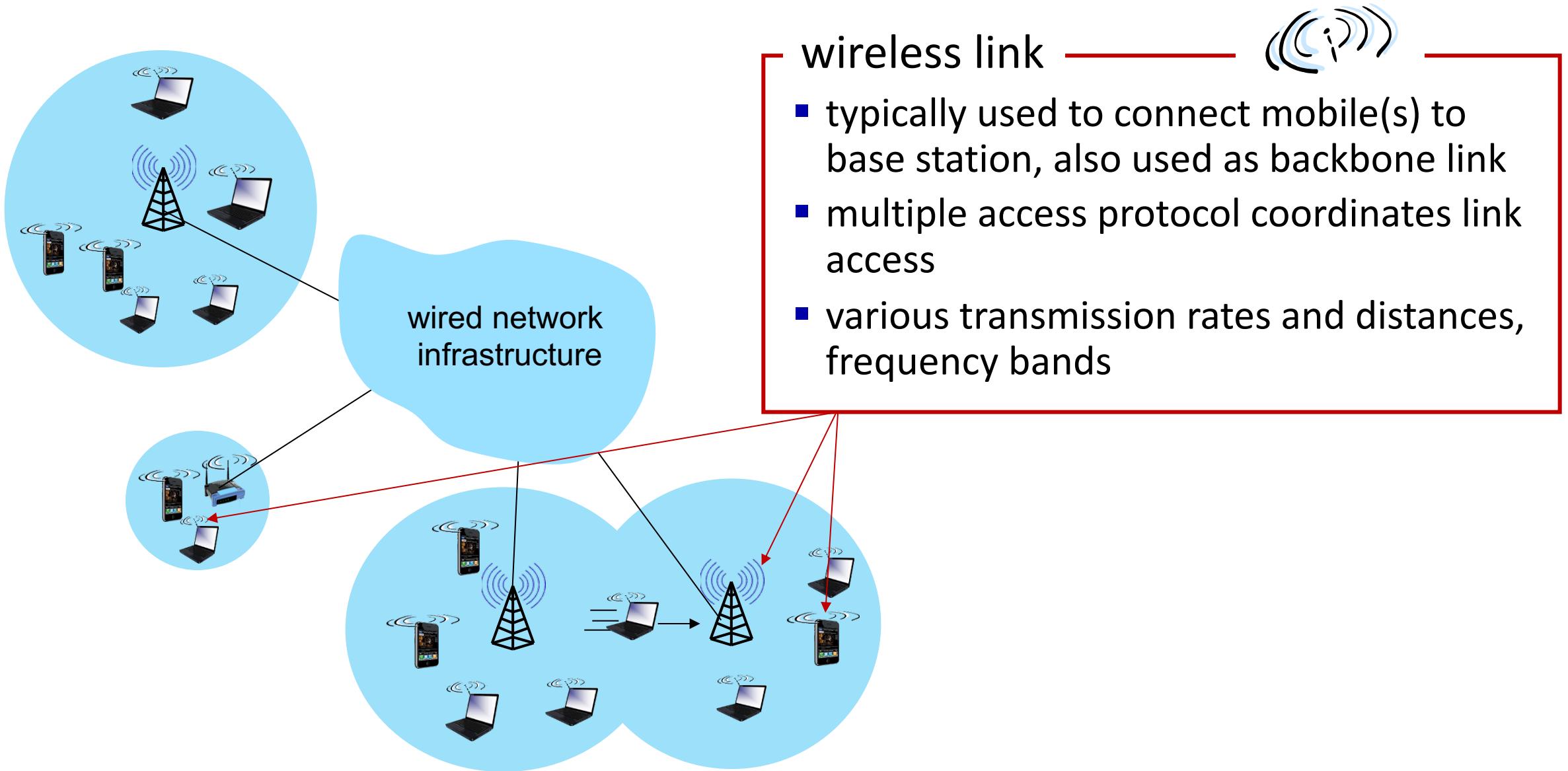
- laptop, smartphone, IoT
- run applications
- may be stationary (non-mobile) or mobile
  - wireless does *not* always mean mobility!



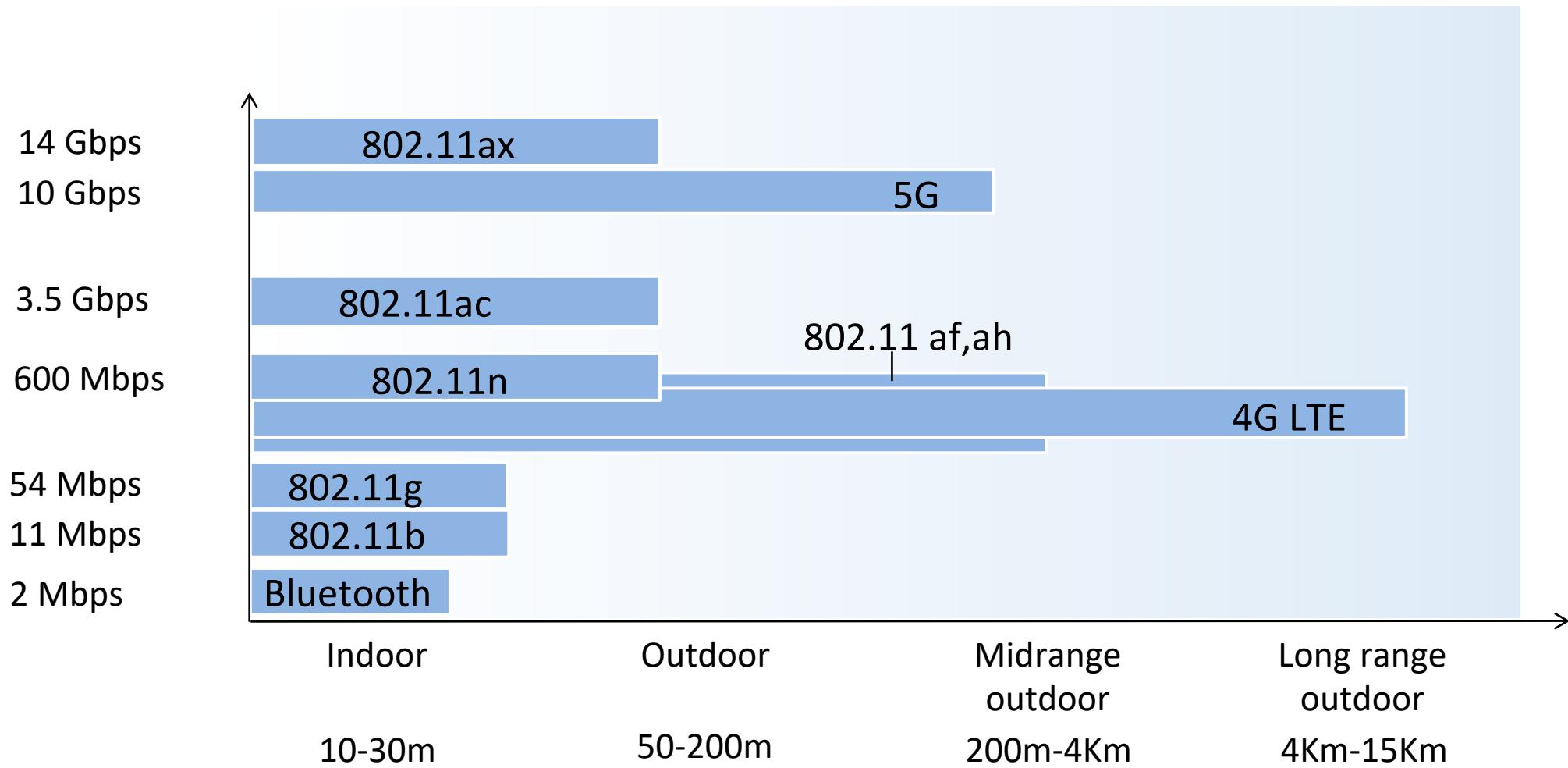
# Elements of a wireless network



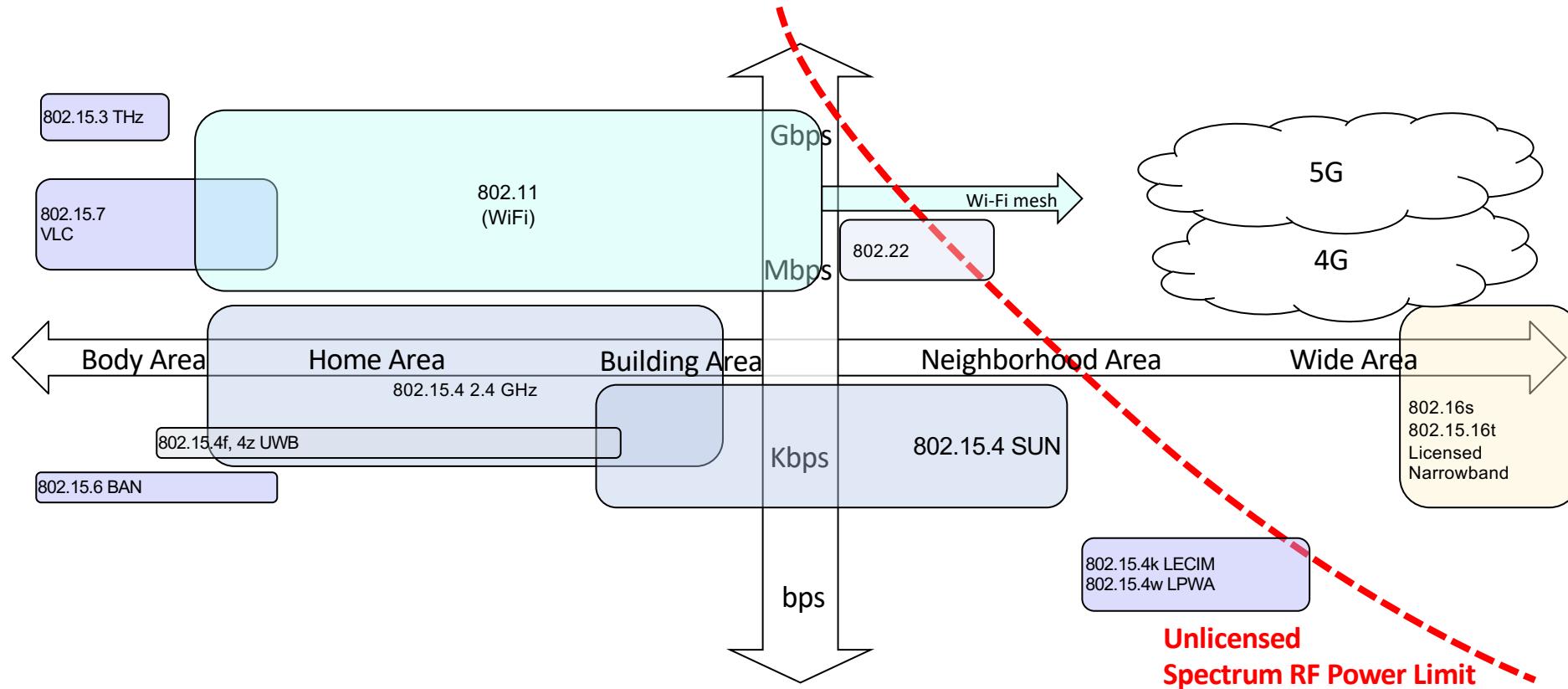
# Elements of a wireless network



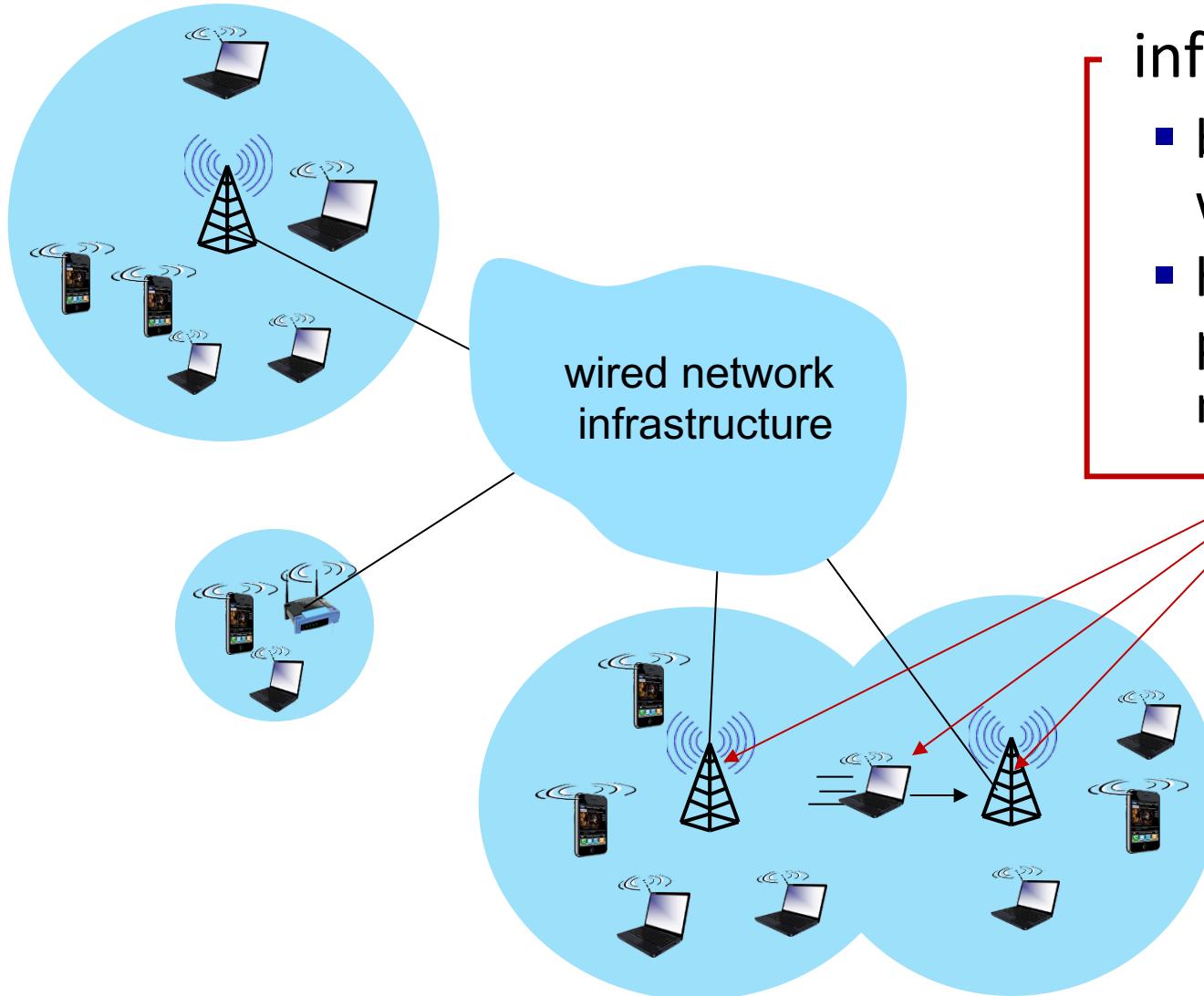
# Characteristics of selected wireless links



# Characteristics of selected wireless links



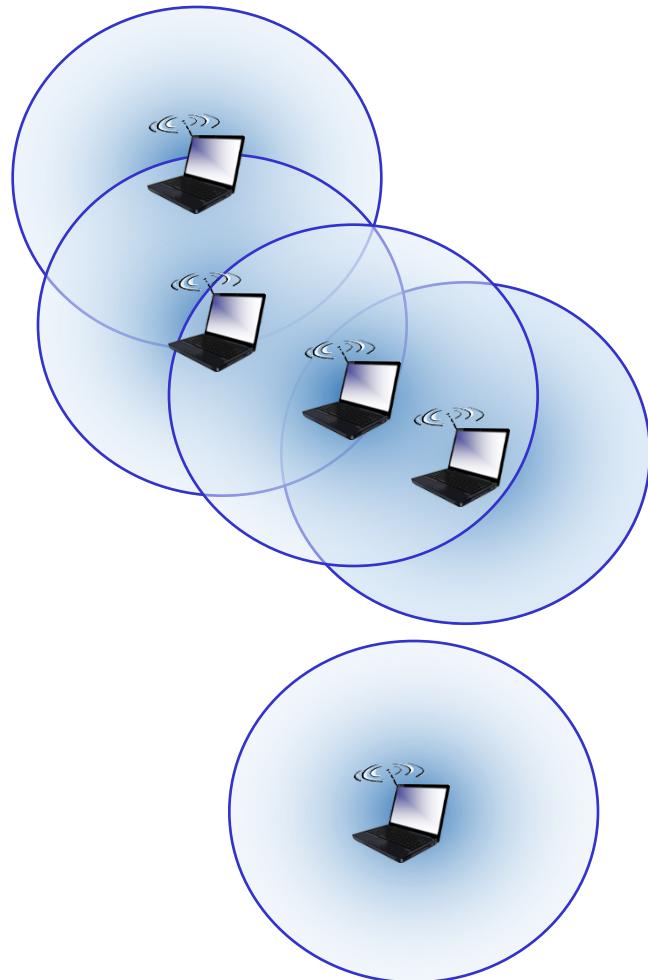
# Elements of a wireless network



## infrastructure mode

- base station connects mobiles into wired network
- handoff: mobile changes base station providing connection into wired network

# Elements of a wireless network



ad hoc mode

- no base stations
- nodes can only transmit to other nodes within link coverage
- nodes organize themselves into a network: route among themselves

# Chapter 7 outline

- Introduction

## Wireless

- Wireless links and network characteristics
- WiFi: 802.11 wireless LANs
- Cellular networks: 4G and 5G



## Mobility

- Mobility management: principles
- Mobility management: practice
  - 4G/5G networks
  - Mobile IP
- Mobility: impact on higher-layer protocols

# Wireless link characteristics (1)

*important* differences from wired link ....

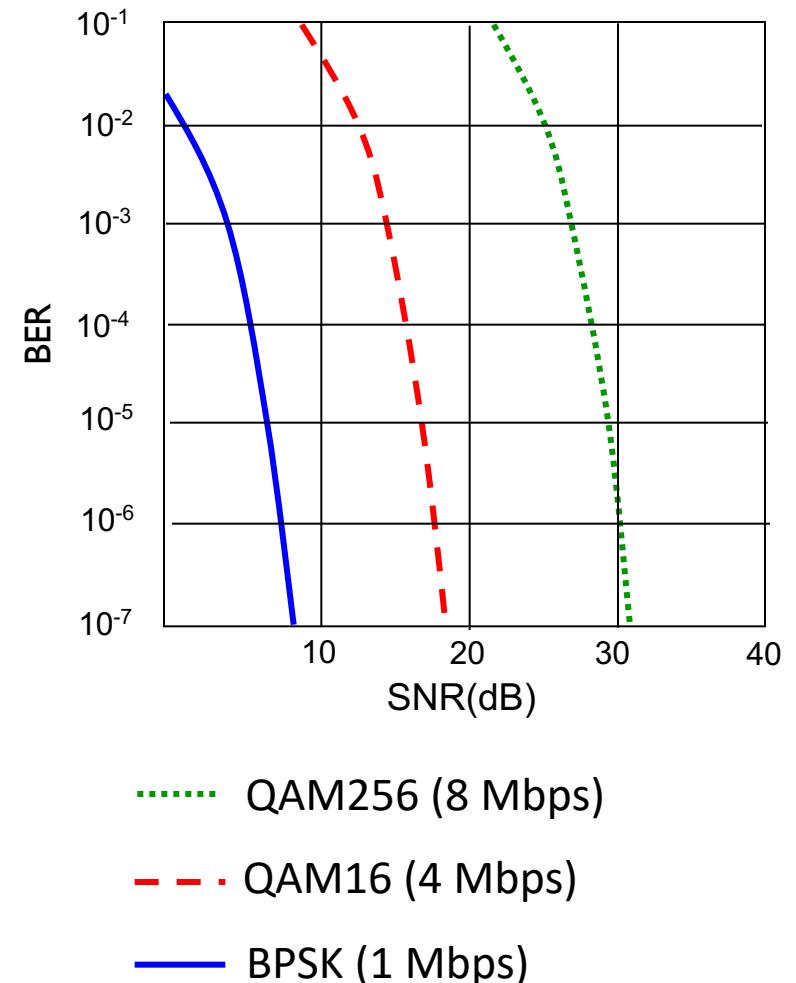
- **decreased signal strength:** radio signal attenuates as it propagates through matter (path loss)
- **interference from other sources:** wireless network frequencies (e.g., 2.4 GHz) shared by many devices (e.g., WiFi, cellular, motors): interference
- **multipath propagation:** radio signal reflects off objects ground, arriving at destination at slightly different times

.... make communication across (even a point to point) wireless link much more “difficult”



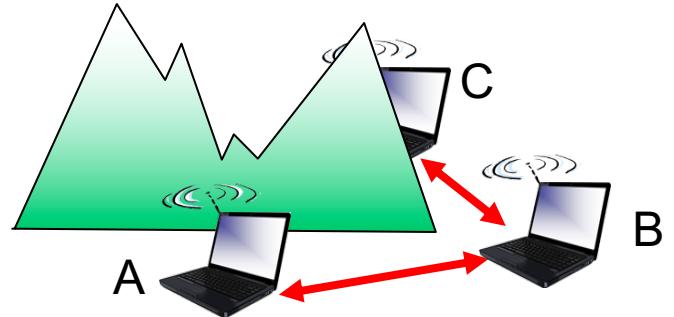
# Wireless link characteristics (2)

- SNR: signal-to-noise ratio
  - larger SNR – easier to extract signal from noise (a “good thing”)
- SNR versus BER tradeoffs
  - *given physical layer*: increase power -> increase SNR->decrease BER
  - *given SNR*: choose physical layer that meets BER requirement, giving highest throughput
    - SNR may change with mobility: dynamically adapt physical layer (modulation technique, rate)



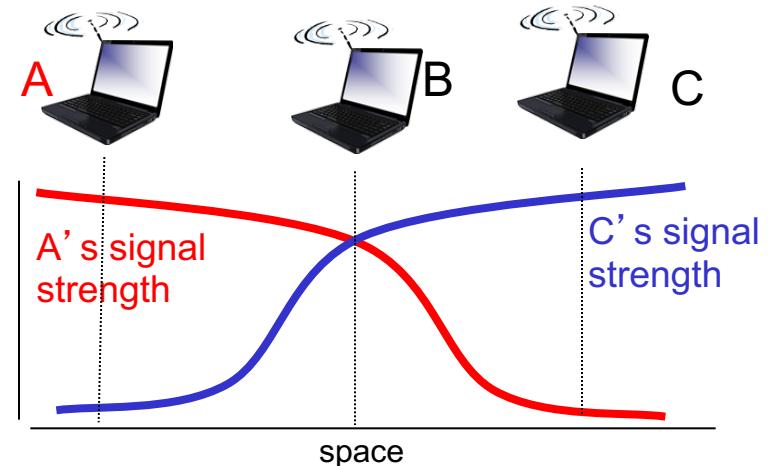
# Wireless link characteristics (3)

Multiple wireless senders, receivers create additional problems (beyond multiple access):



## Hidden terminal problem

- B, A hear each other
- B, C hear each other
- A, C can not hear each other means A, C unaware of their interference at B



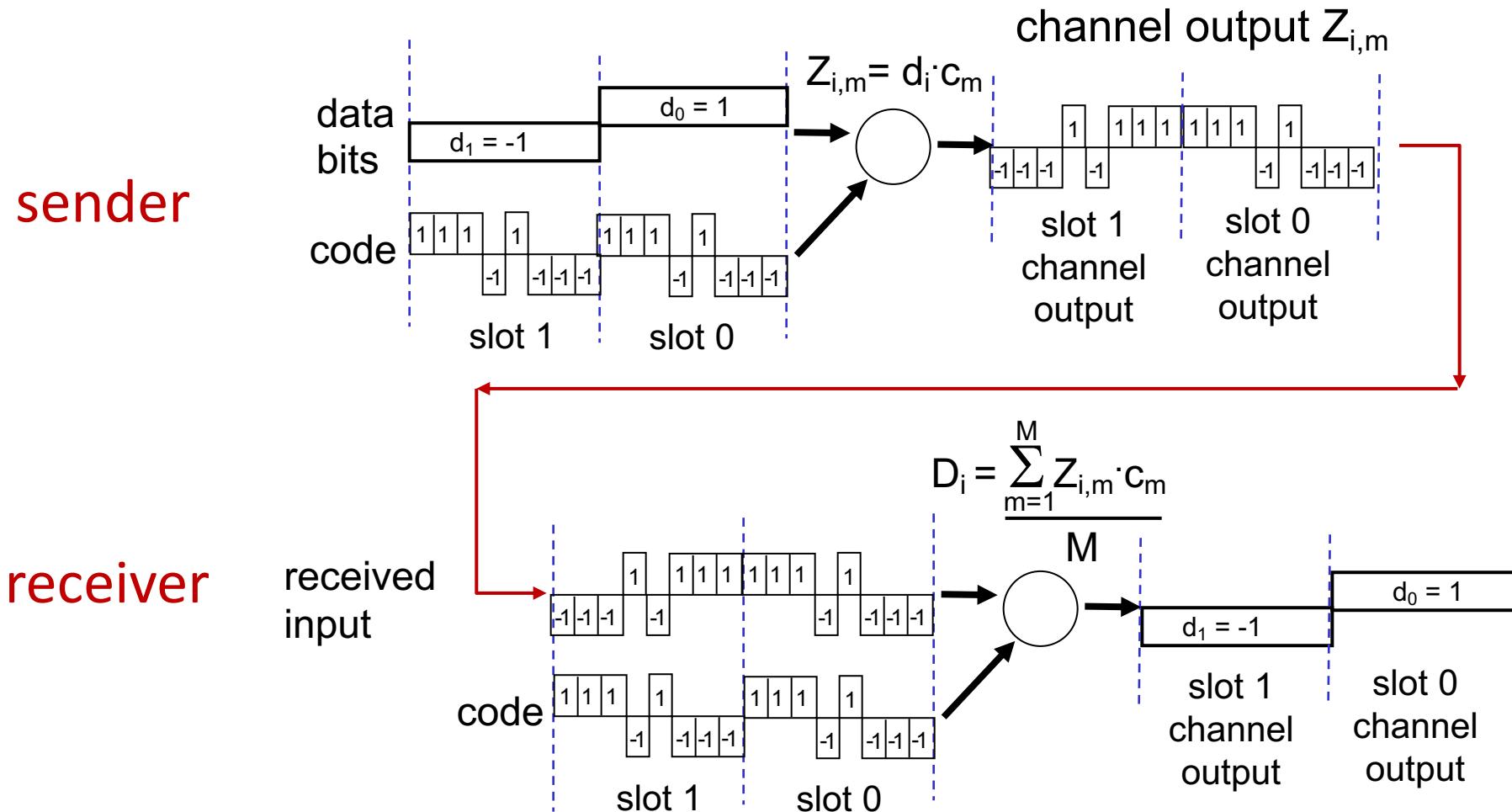
## Signal attenuation:

- B, A hear each other
- B, C hear each other
- A, C can not hear each other interfering at B

# Code Division Multiple Access (CDMA)

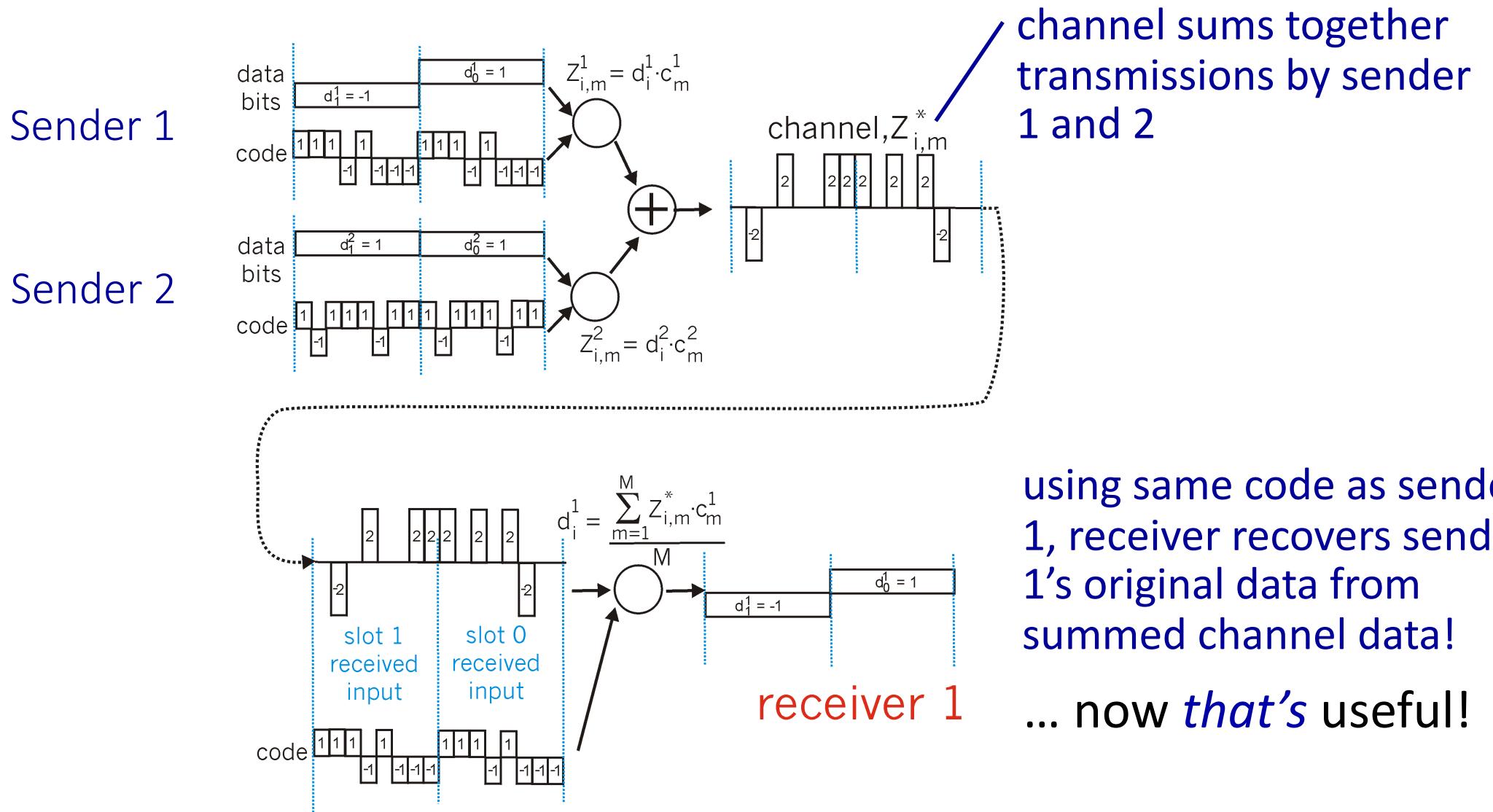
- unique “code” assigned to each user; i.e., code set partitioning
  - all users share same frequency, but each user has own “chipping” sequence (i.e., code) to encode data
  - allows multiple users to “coexist” and transmit simultaneously with minimal interference (if codes are “orthogonal”)
- **encoding:** inner product: (original data)  $\times$  (chipping sequence)
- **decoding:** summed inner-product: (encoded data)  $\times$  (chipping sequence)

# CDMA encode/decode



... but this isn't really useful yet!

# CDMA: two-sender interference



# Chapter 7 outline

- Introduction

## Wireless

- Wireless links and network characteristics
- WiFi: 802.11 wireless LANs
- Cellular networks: 4G and 5G



## Mobility

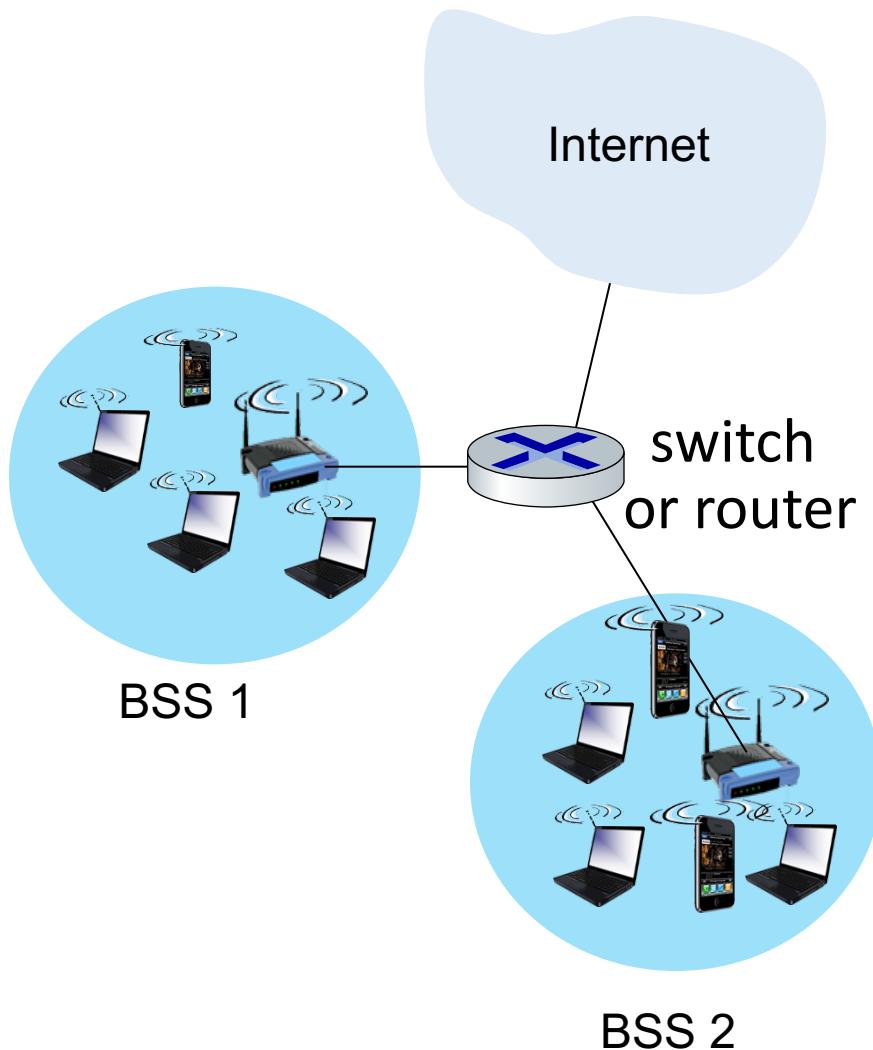
- Mobility management: principles
- Mobility management: practice
  - 4G/5G networks
  - Mobile IP
- Mobility: impact on higher-layer protocols

# IEEE 802.11 Wireless LAN

IEEE 802.11 standard	Year	Max data rate	Range	Frequency
802.11b	1999	11 Mbps	30 m	2.4 Ghz
802.11g	2003	54 Mbps	30m	2.4 Ghz
802.11n (WiFi 4)	2009	600 Mbps	70m	2.4, 5 Ghz
802.11ac (WiFi 5)	2013	3.47Gpbs	70m	5 Ghz
802.11ax (WiFi 6)	2020	14 Gbps	70m	2.4, 5 Ghz
802.11af	2014	35 – 560 Mbps	1 Km	unused TV bands (54-790 MHz)
802.11ah	2017	347Mbps	1 Km	900 Mhz

- all use CSMA/CA for multiple access, and have base-station and ad-hoc network versions

# 802.11 LAN architecture



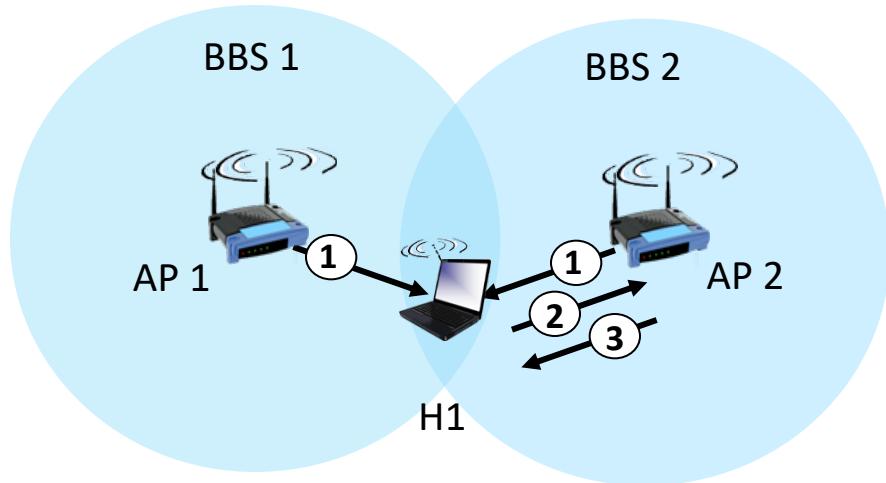
- wireless host communicates with base station
  - **base station = access point (AP)**
- **Basic Service Set (BSS) (aka “cell”)** in infrastructure mode contains:
  - wireless hosts
  - access point (AP): base station
  - ad hoc mode: hosts only

# 802.11: Channels, association

- spectrum divided into channels at different frequencies
  - AP admin chooses frequency for AP
  - interference possible: channel can be same as that chosen by neighboring AP!
- arriving host: must **associate** with an AP
  - scans channels, listening for *beacon frames* containing AP's name (SSID) and MAC address
  - selects AP to associate with
  - then may perform authentication [Chapter 8]
  - then typically run DHCP to get IP address in AP's subnet

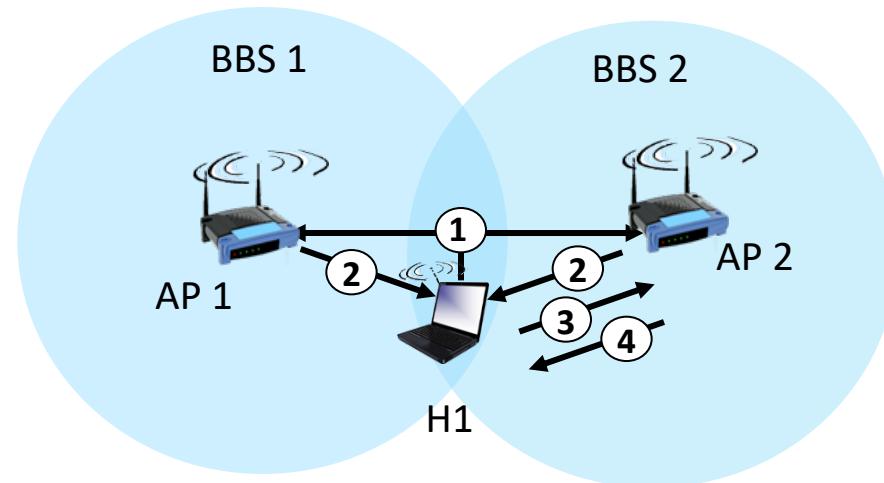


# 802.11: passive/active scanning



## passive scanning:

- (1) beacon frames sent from APs
- (2) association Request frame sent: H1 to selected AP
- (3) association Response frame sent from selected AP to H1

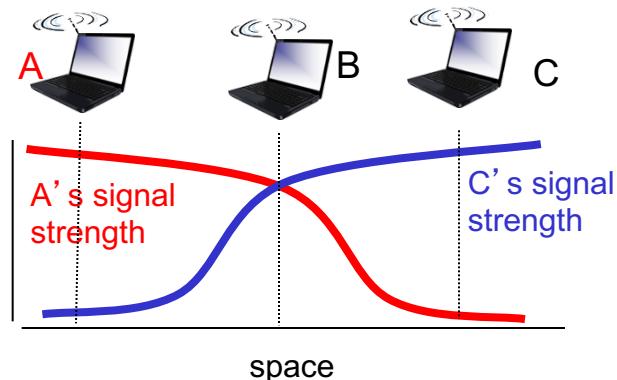
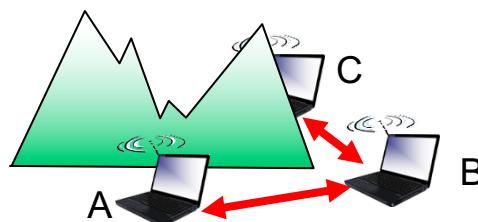


## active scanning:

- (1) Probe Request frame broadcast from H1
- (2) Probe Response frames sent from APs
- (3) Association Request frame sent: H1 to selected AP
- (4) Association Response frame sent from selected AP to H1

# IEEE 802.11: multiple access

- avoid collisions:  $2^+$  nodes transmitting at same time
- 802.11: CSMA - sense before transmitting
  - don't collide with detected ongoing transmission by another node
- 802.11: *no collision detection!*
  - difficult to sense collisions: high transmitting signal, weak received signal due to fading
  - can't sense all collisions in any case: hidden terminal, fading
  - goal: *avoid collisions*: CSMA/Collision Avoidance



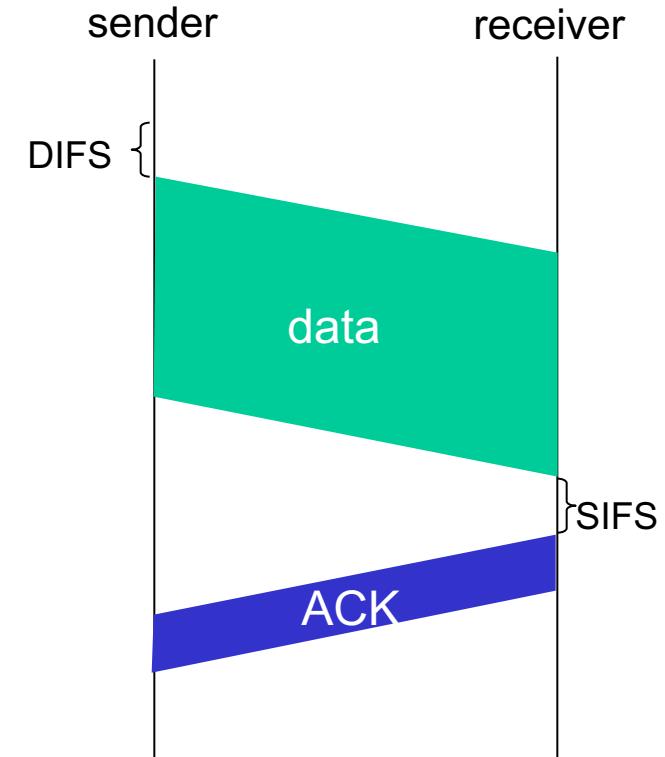
# IEEE 802.11 MAC Protocol: CSMA/CA

## 802.11 sender

- 1 if sense channel idle for **DIFS** then  
    transmit entire frame (no CD)
- 2 if sense channel busy then  
    start random backoff time  
    timer counts down while channel idle  
    transmit when timer expires  
    if no ACK, increase random backoff interval, repeat 2

## 802.11 receiver

if frame received OK  
    return ACK after **SIFS** (ACK needed due to hidden  
    terminal problem)

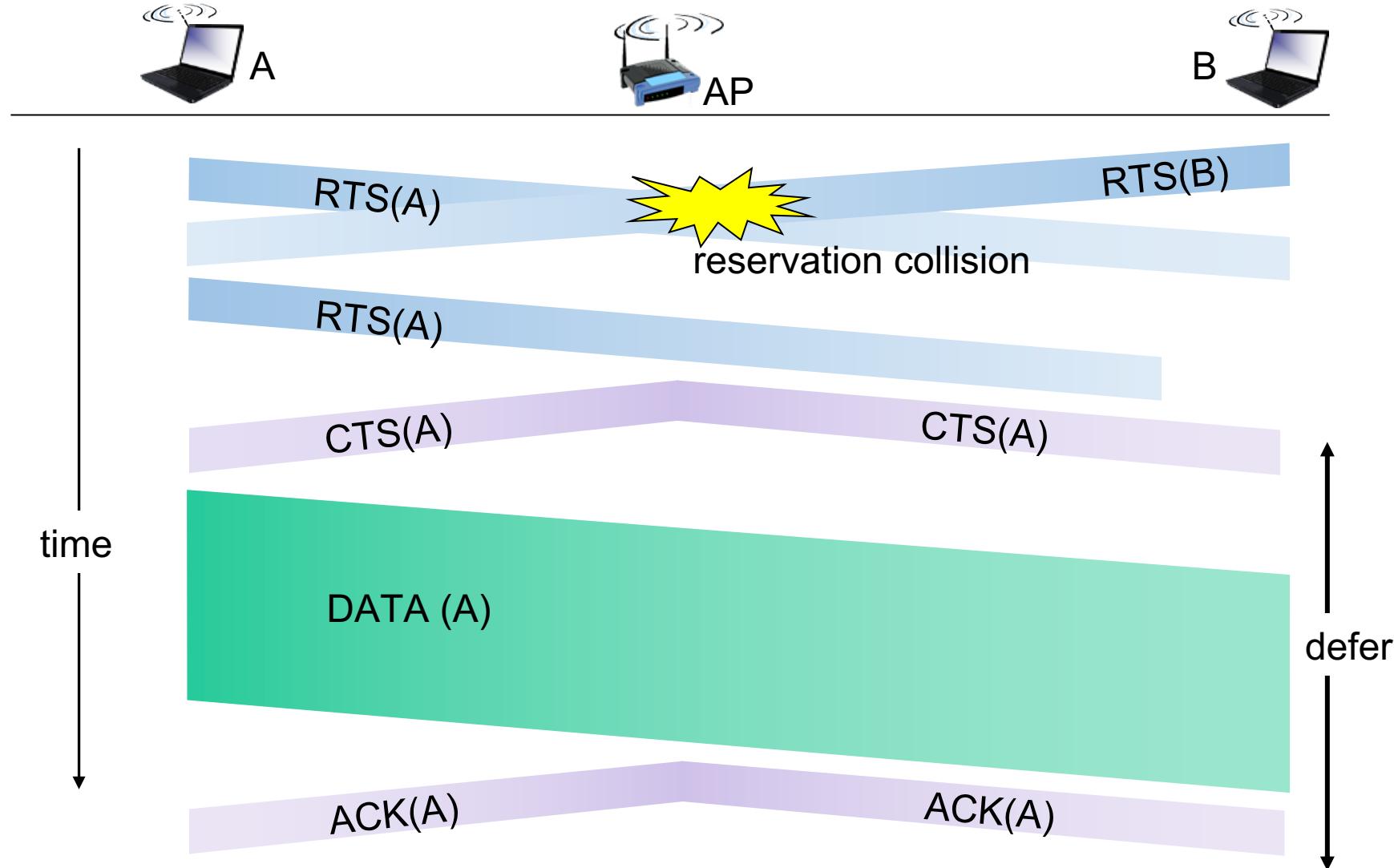


# Avoiding collisions (more)

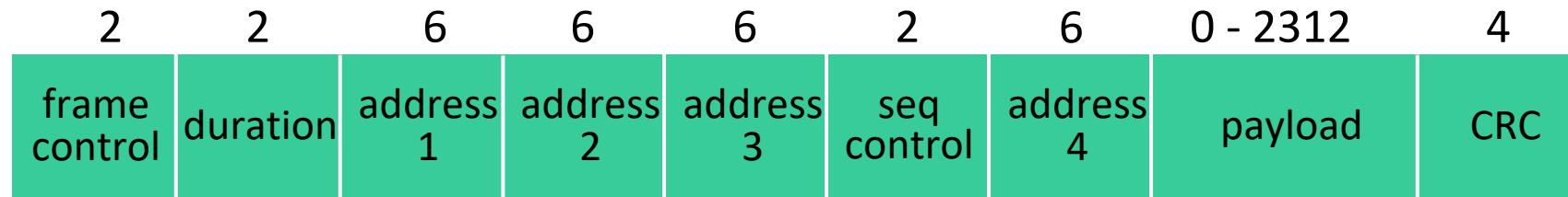
**idea:** sender “reserves” channel use for data frames using small reservation packets

- sender first transmits *small* request-to-send (RTS) packet to BS using CSMA
  - RTSs may still collide with each other (but they’re short)
- BS broadcasts clear-to-send CTS in response to RTS
- CTS heard by all nodes
  - sender transmits data frame
  - other stations defer transmissions

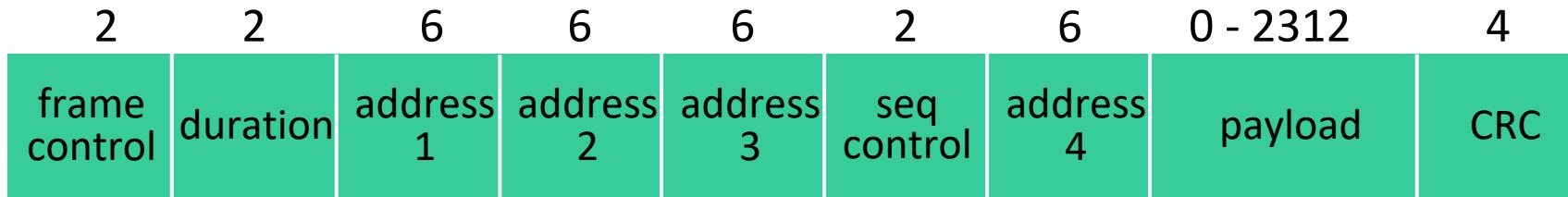
# Collision Avoidance: RTS-CTS exchange



# 802.11 frame: addressing



# 802.11 frame: addressing



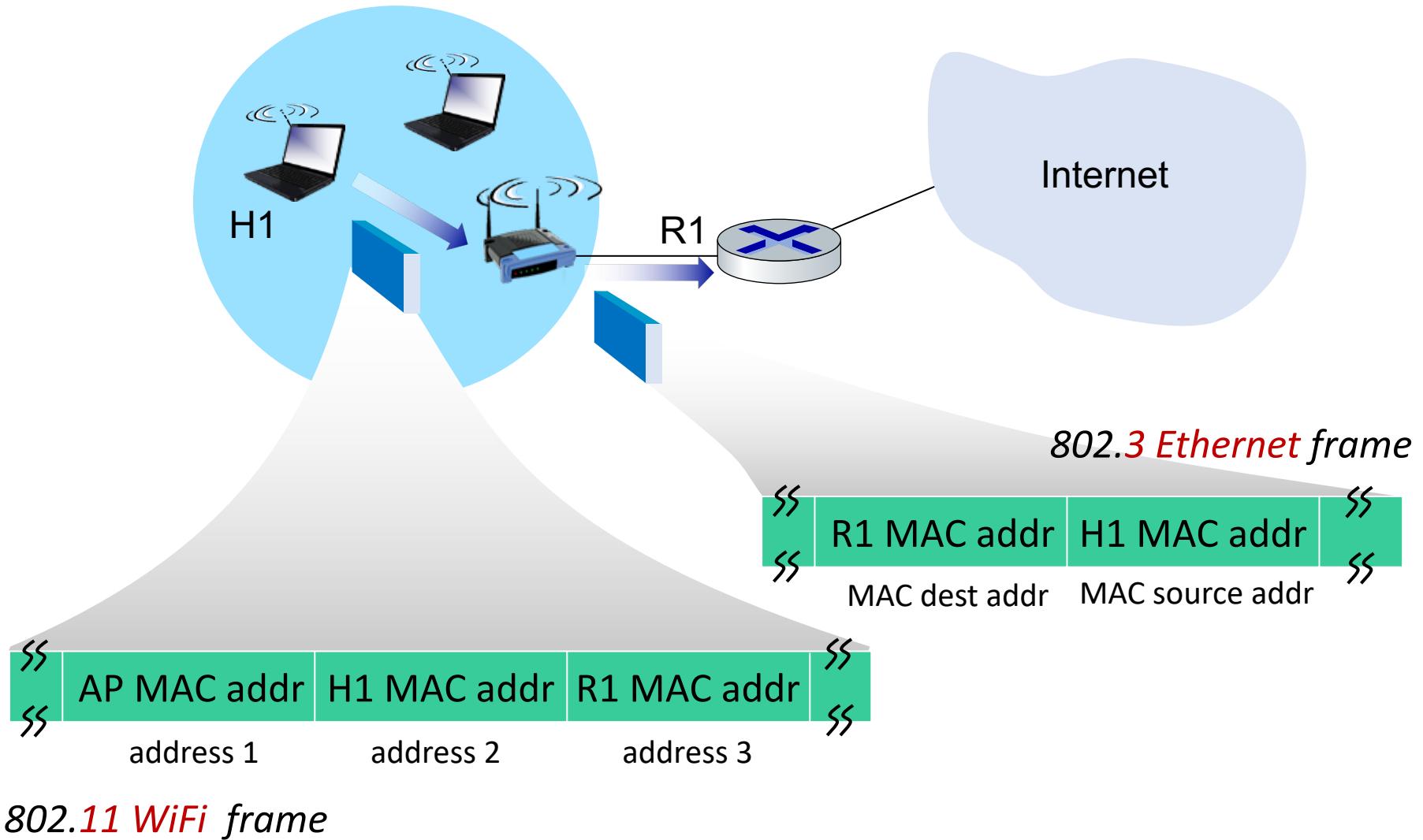
**Address 1:** MAC address of wireless host or AP to receive this frame

**Address 2:** MAC address of wireless host or AP transmitting this frame

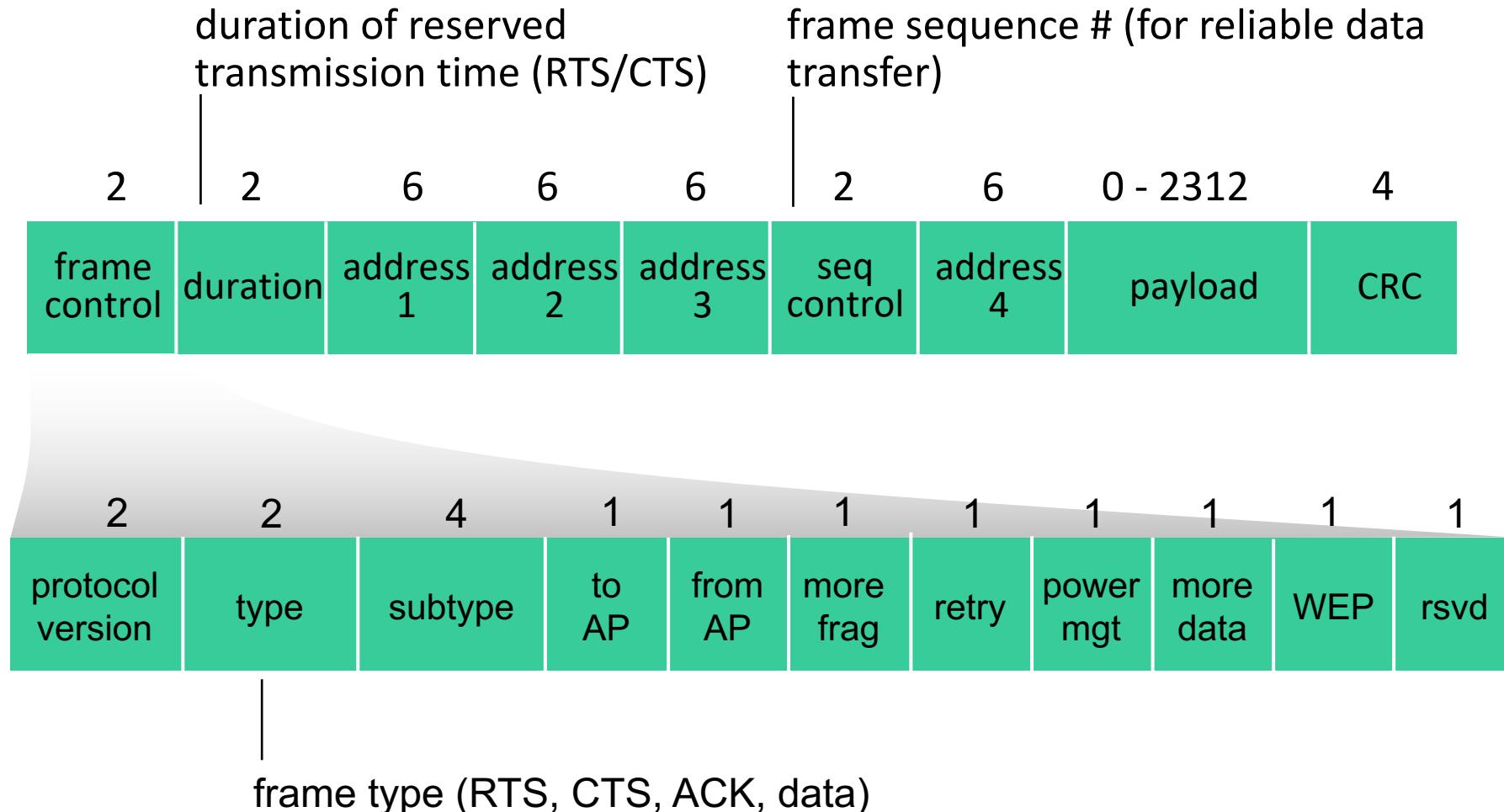
**Address 4:** used only in ad hoc mode

**Address 3:** MAC address of router interface to which AP is attached

# 802.11 frame: addressing



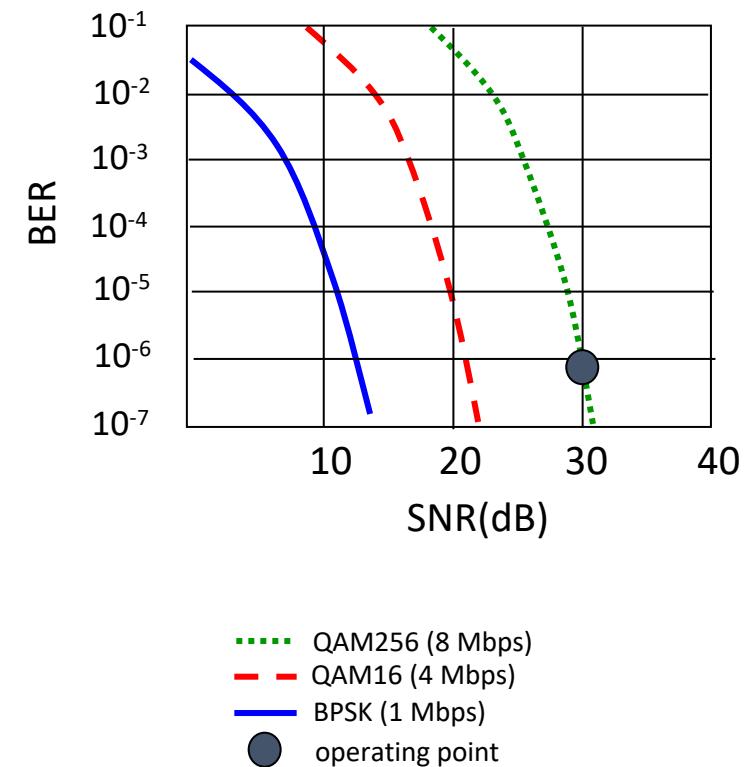
# 802.11 frame: addressing



# 802.11: advanced capabilities

## Rate adaptation

- base station, mobile dynamically change transmission rate (physical layer modulation technique) as mobile moves, SNR varies
  - SNR decreases, BER increase as node moves away from base station
  - When BER becomes too high, switch to lower transmission rate but with lower BER



# 802.11: advanced capabilities

## power management

- node-to-AP: “I am going to sleep until next beacon frame”
  - AP knows not to transmit frames to this node
  - node wakes up before next beacon frame
- beacon frame: contains list of mobiles with AP-to-mobile frames waiting to be sent
  - node will stay awake if AP-to-mobile frames to be sent; otherwise sleep again until next beacon frame

# Chapter 7 outline

- Introduction

## Wireless

- Wireless links and network characteristics
- WiFi: 802.11 wireless LANs
- Cellular networks: 4G and 5G



## Mobility

- Mobility management: principles
- Mobility management: practice
  - 4G/5G networks
  - Mobile IP
- Mobility: impact on higher-layer protocols

# 4G/5G cellular networks

- *the* solution for wide-area mobile Internet
- widespread deployment/use:
  - more mobile-broadband-connected devices than fixed-broadband-connected devices (5-1 in 2019)!
  - 4G availability: 97% of time in Korea (90% in US)
- transmission rates up to 100's Mbps
- technical standards: 3rd Generation Partnership Project (3GPP)
  - [www.3gpp.org](http://www.3gpp.org)
  - 4G: Long-Term Evolution (LTE)standard

# 4G/5G cellular networks

## *similarities to wired Internet*

- edge/core distinction, but both below to same carrier
- global cellular network: a network of networks
- widespread use of protocols we've studied: HTTP, DNS, TCP, UDP, IP, NAT, separation of data/control planes, SDN, Ethernet, tunneling
- interconnected to wired Internet

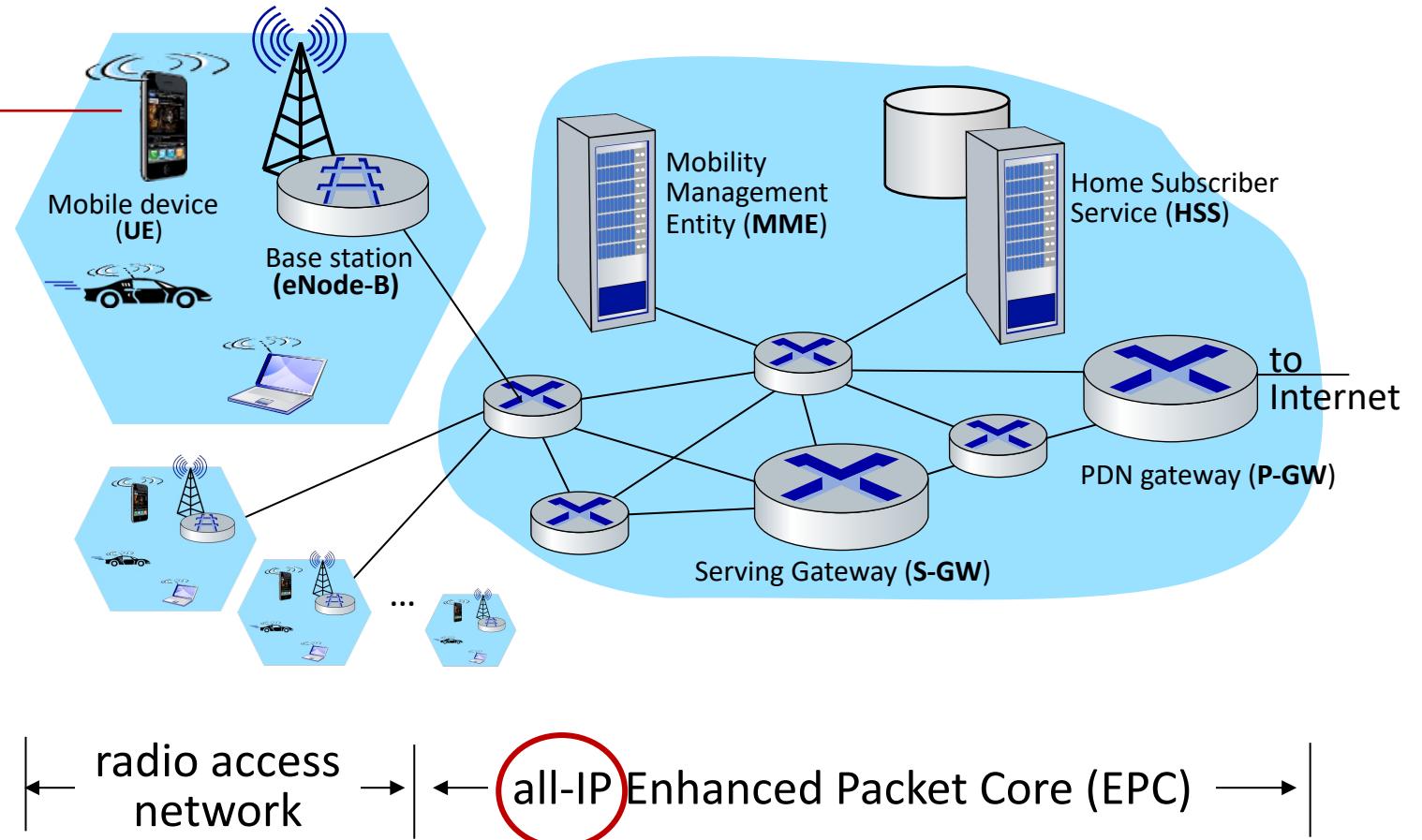
## *differences from wired Internet*

- different wireless link layer
- mobility as a 1<sup>st</sup> class service
- user “identity” (via SIM card)
- business model: users subscribe to a cellular provider
  - strong notion of “home network” versus roaming on visited nets
  - global access, with authentication infrastructure, and inter-carrier settlements

# Elements of 4G LTE architecture

## Mobile device:

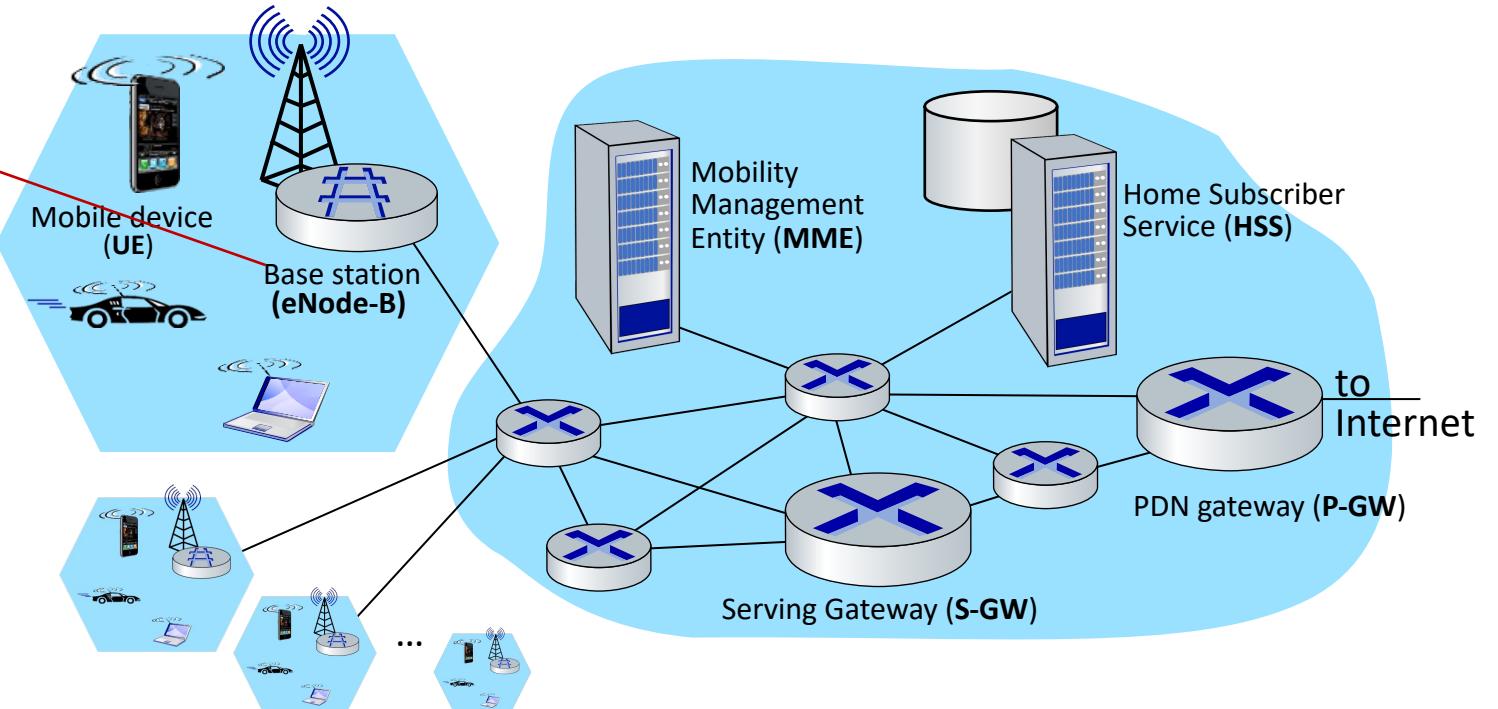
- smartphone, tablet, laptop, IoT, ... with 4G LTE radio
- 64-bit International Mobile Subscriber Identity (IMSI), stored on SIM (Subscriber Identity Module) card
- LTE jargon: User Equipment (UE)



# Elements of 4G LTE architecture

## Base station:

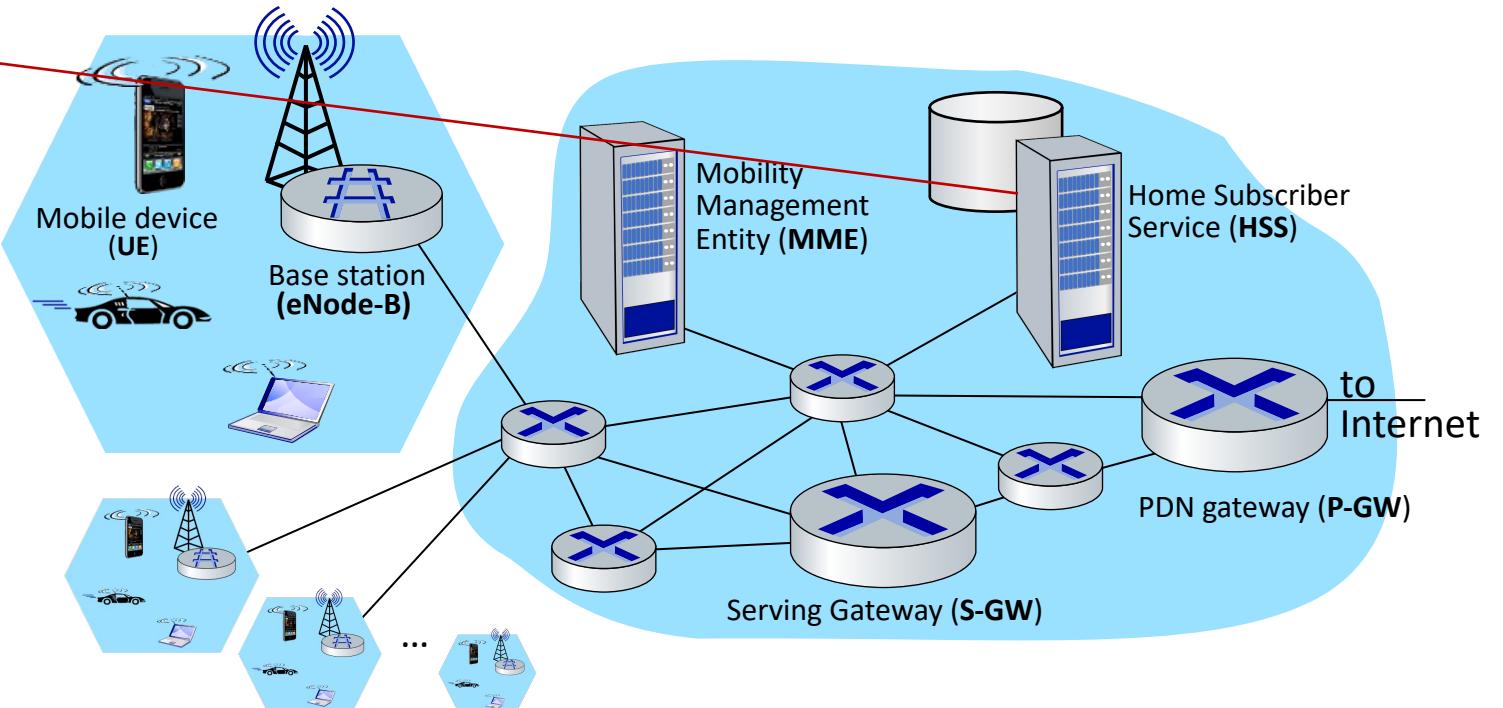
- at “edge” of carrier’s network
- manages wireless radio resources, mobile devices in its coverage area (“cell”)
- coordinates device authentication with other elements
- similar to WiFi AP but:
  - active role in user mobility
  - coordinates with nearby base stations to optimize radio use
- LTE jargon: eNode-B



# Elements of 4G LTE architecture

## Home Subscriber Service

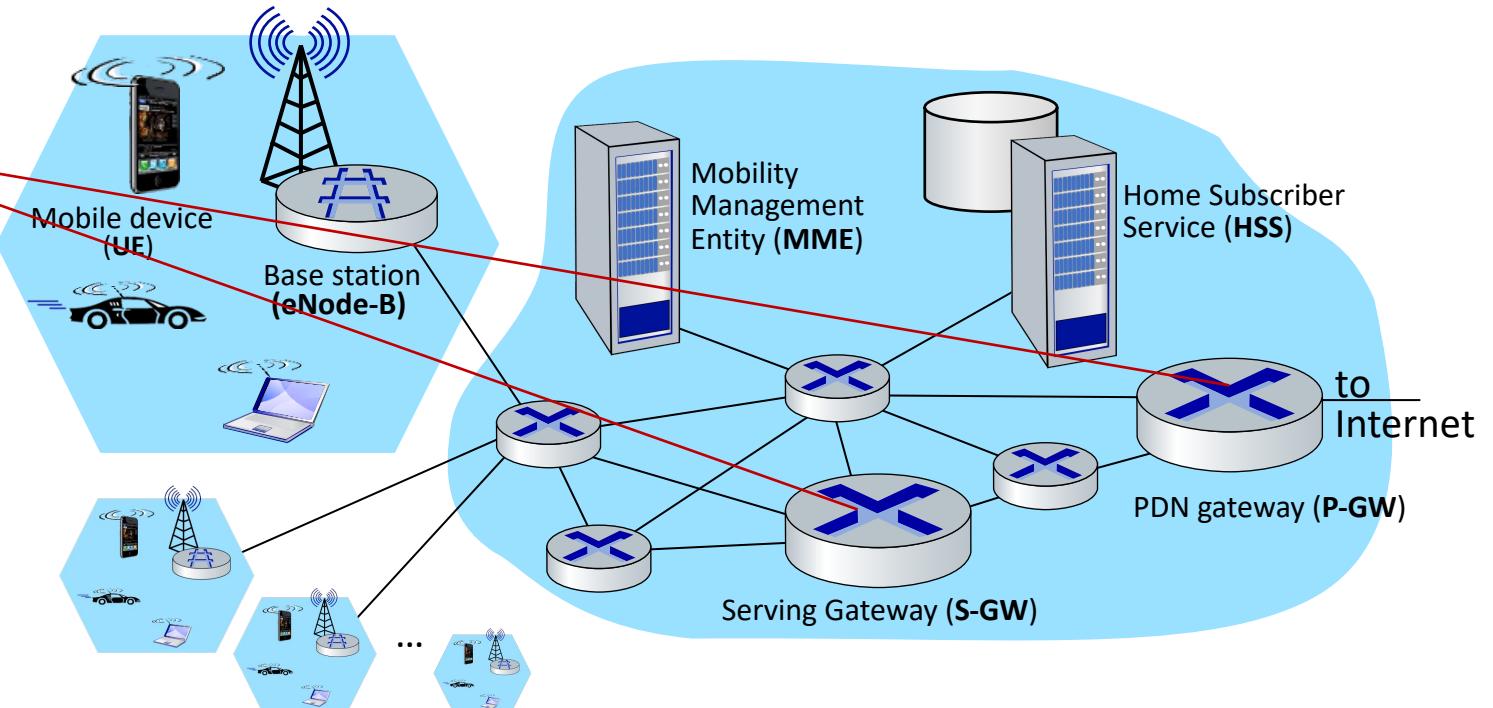
- stores info about mobile devices for which the HSS's network is their "home network"
- works with MME in device authentication



# Elements of 4G LTE architecture

## Serving Gateway (S-GW), PDN Gateway (P-GW)

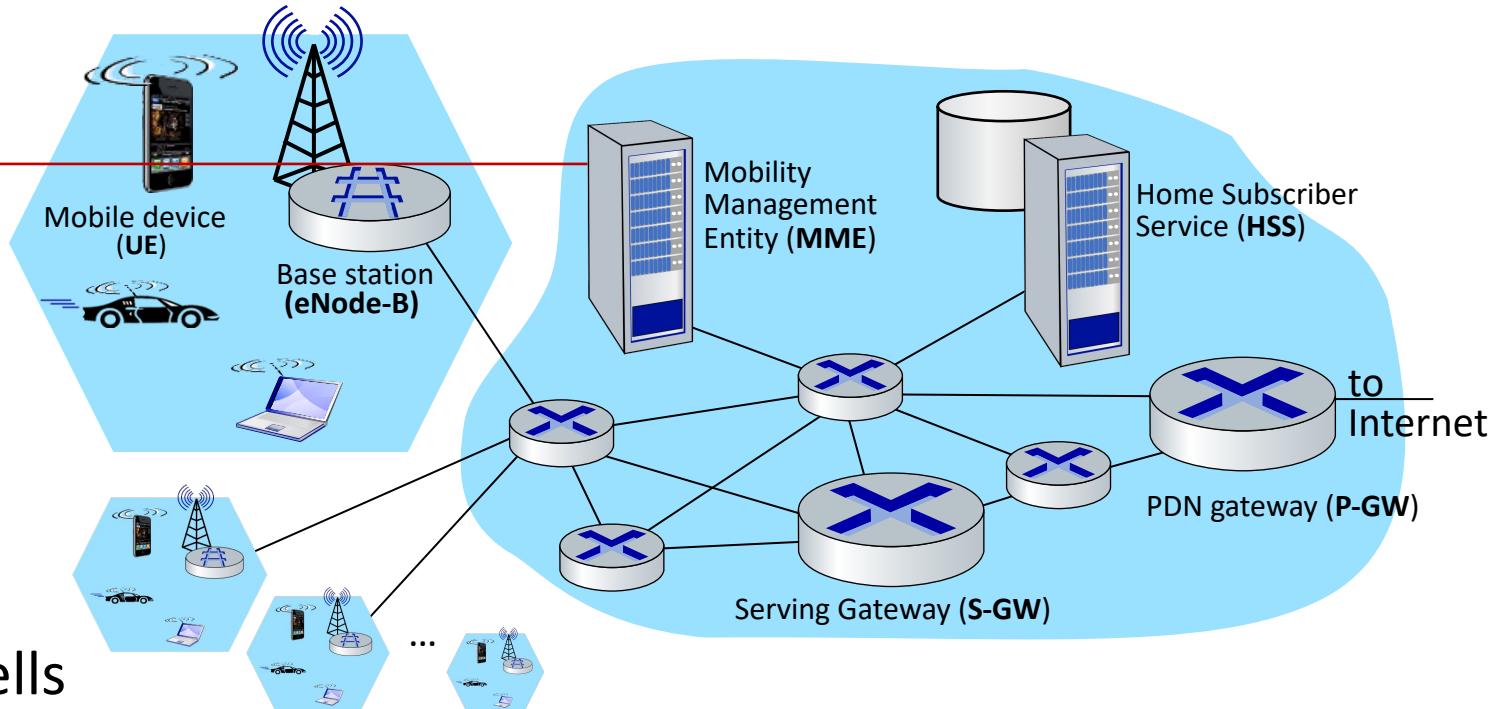
- lie on data path from mobile to/from Internet
- P-GW
  - gateway to mobile cellular network
  - Looks like any other internet gateway router
  - provides NAT services
- other routers:
  - extensive use of tunneling



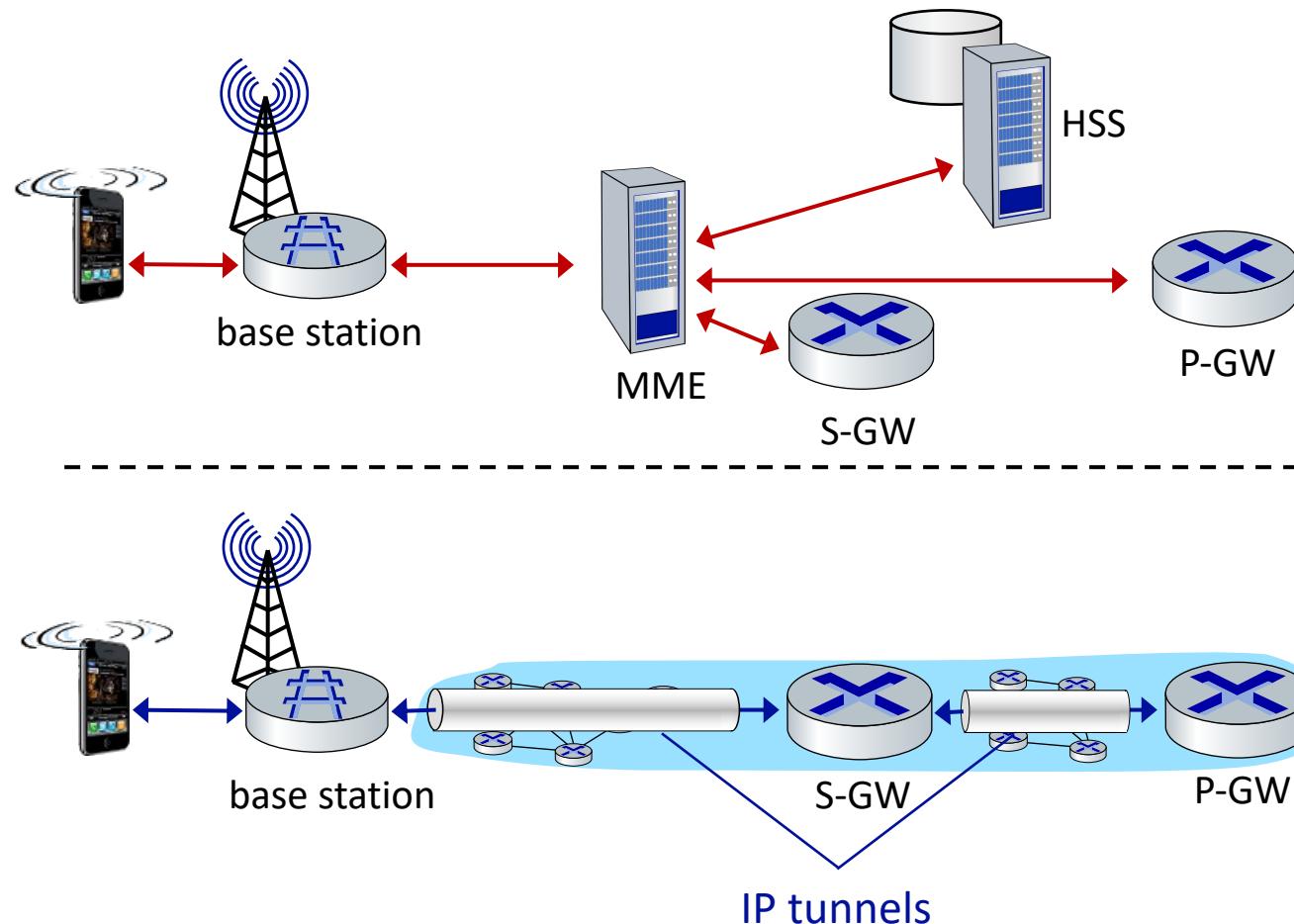
# Elements of 4G LTE architecture

## Mobility Management Entity

- device authentication (device-to-network, network-to-device) coordinated with mobile home network HSS
- mobile device management:
  - device handover between cells
  - tracking/paging device location
- path (tunneling) setup from mobile device to P-GW



# LTE: data plane control plane separation



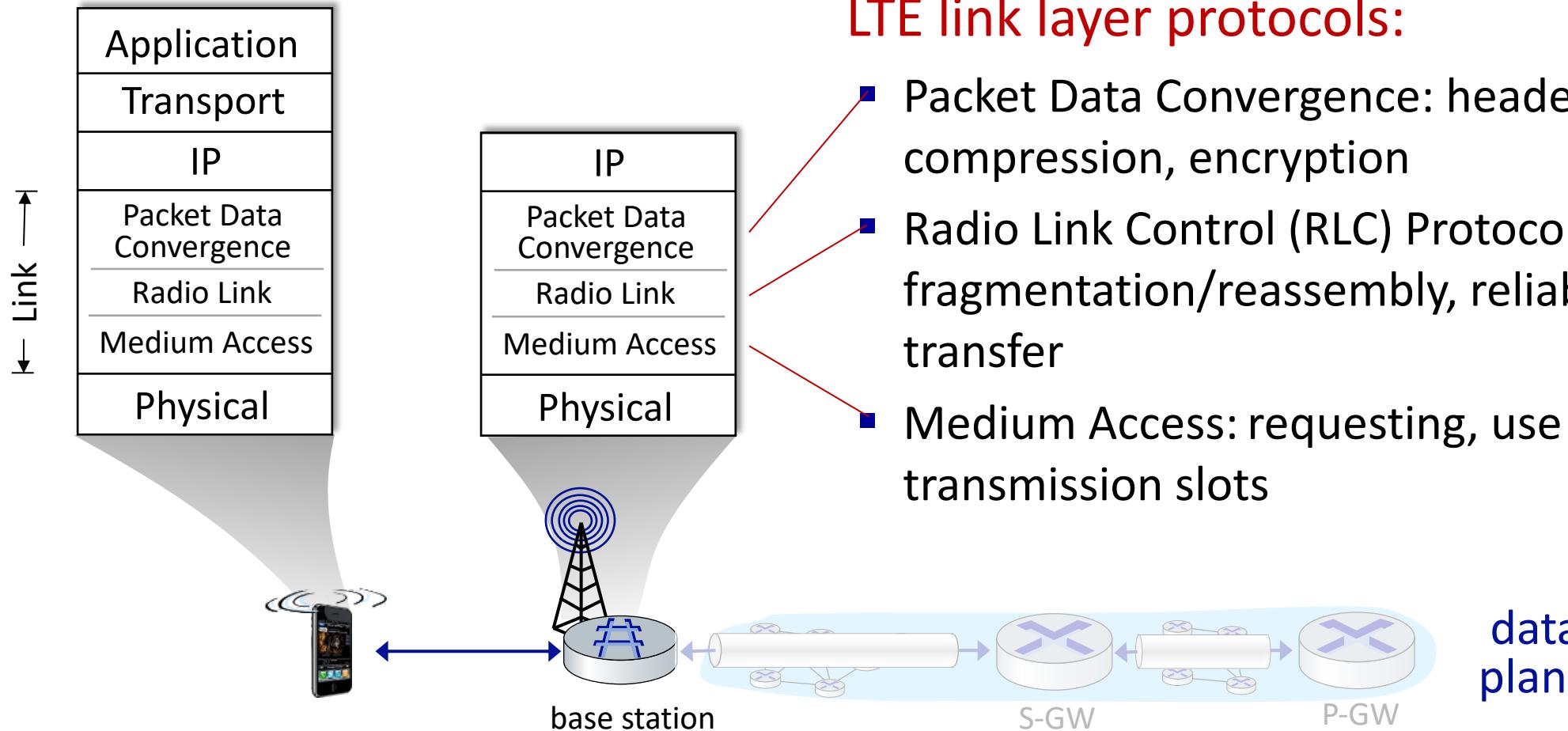
## control plane

- new protocols for mobility management , security, authentication

## data plane

- new protocols at link, physical layers
- extensive use of tunneling to facilitate mobility

# LTE data plane protocol stack: first hop

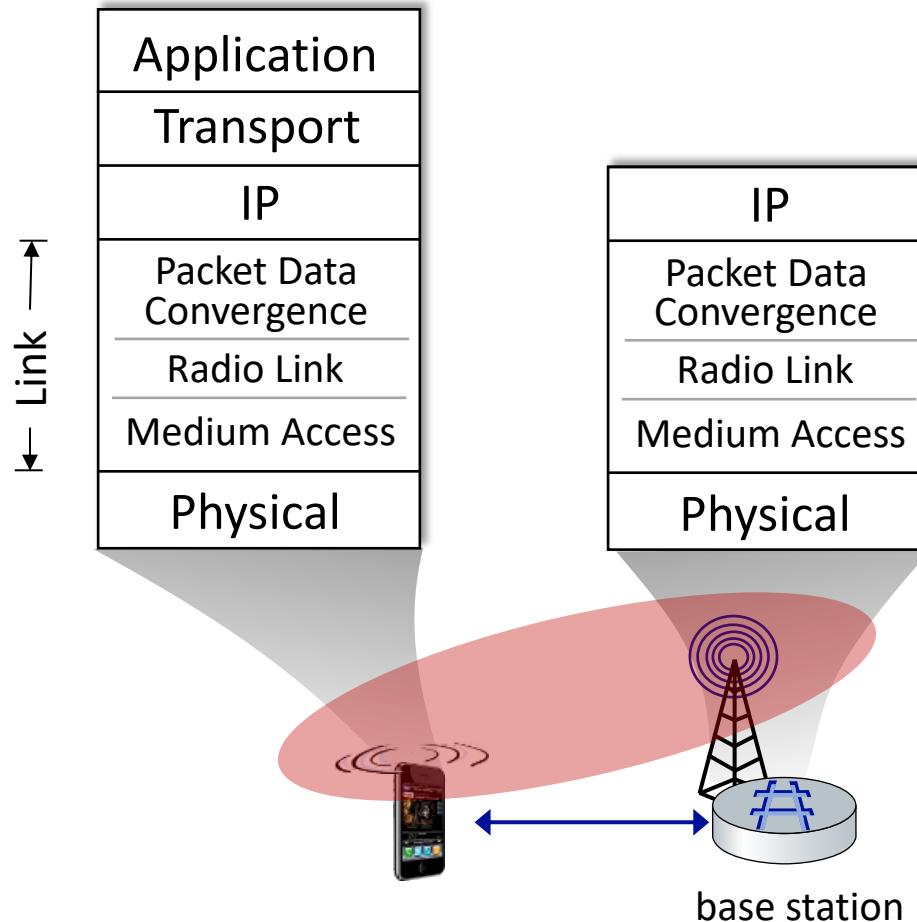


## LTE link layer protocols:

- Packet Data Convergence: header compression, encryption
- Radio Link Control (RLC) Protocol: fragmentation/reassembly, reliable data transfer
- Medium Access: requesting, use of radio transmission slots

data  
plane

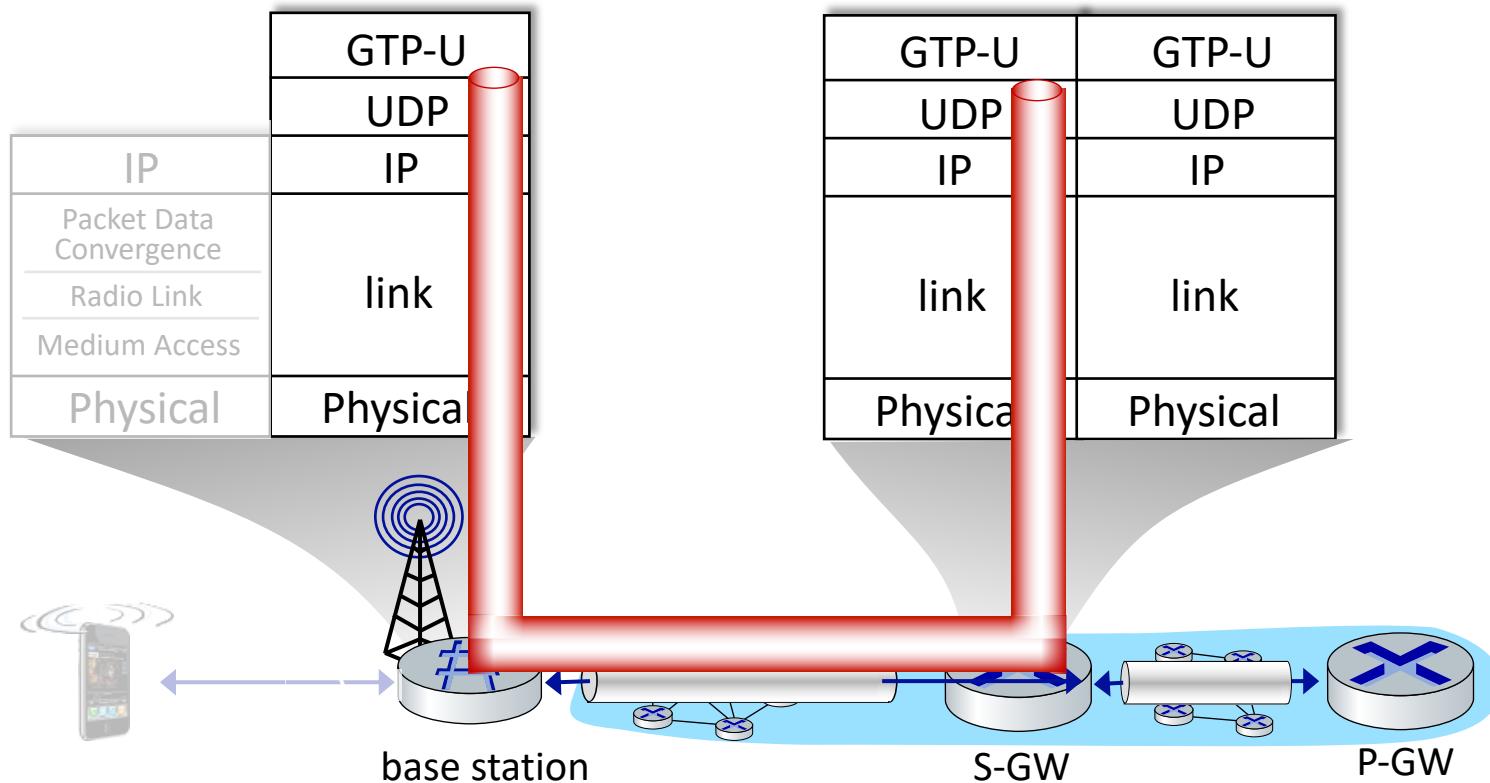
# LTE data plane protocol stack: first hop



## LTE radio access network:

- **downstream channel:** FDM, TDM within frequency channel (OFDM - orthogonal frequency division multiplexing)
  - “orthogonal”: minimal interference between channels
- **upstream:** FDM, TDM similar to OFDM
- each active mobile device allocated two or more 0.5 ms time slots over 12 frequencies
  - scheduling algorithm not standardized – up to operator
  - 100's Mbps per device possible

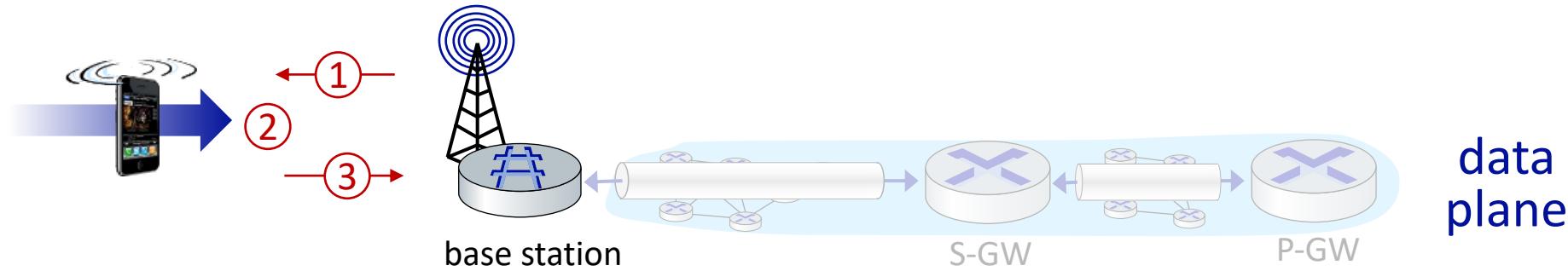
# LTE data plane protocol stack: packet core



## tunneling:

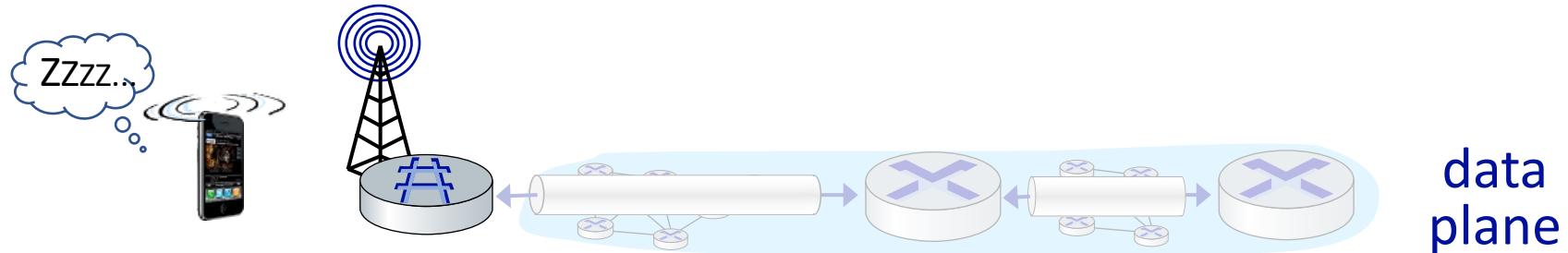
- mobile datagram encapsulated using GPRS Tunneling Protocol (GTP), sent inside UDP datagram to S-GW
- S-GW re-tunnels datagrams to P-GW
- supporting mobility: only tunneling endpoints change when mobile user moves

# LTE data plane: associating with a BS



- ① BS broadcasts primary synch signal every 5 ms on all frequencies
  - BSs from multiple carriers may be broadcasting synch signals
- ② mobile finds a primary synch signal, then locates 2<sup>nd</sup> synch signal on this freq.
  - mobile then finds info broadcast by BS: channel bandwidth, configurations; BS's cellular carrier info
  - mobile may get info from multiple base stations, multiple cellular networks
- ③ mobile selects which BS to associate with (*e.g.*, preference for home carrier)
- ④ more steps still needed to authenticate, establish state, set up data plane

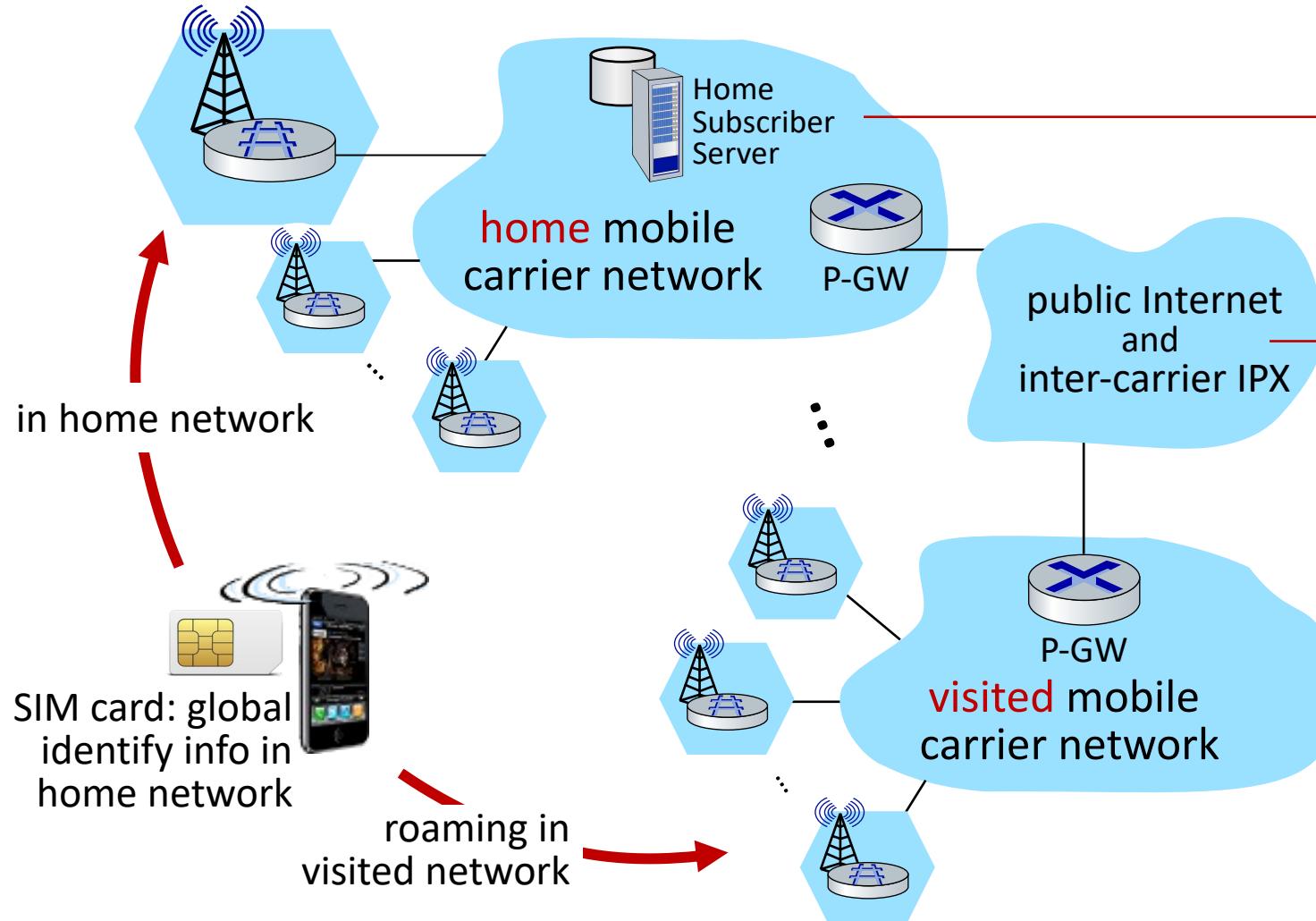
# LTE mobiles: sleep modes



as in WiFi, Bluetooth: LTE mobile may put radio to “sleep” to conserve battery:

- **light sleep:** after 100's msec of inactivity
  - wake up periodically (100's msec) to check for downstream transmissions
- **deep sleep:** after 5-10 secs of inactivity
  - mobile may change cells while deep sleeping – need to re-establish association

# Global cellular network: a network of IP networks



## home network HSS:

- identify & services info, while in home network and roaming

## all IP:

- carriers interconnect with each other, and public internet at exchange points
- legacy 2G, 3G: not all IP, handled otherwise

# On to 5G!

- **goal:** 10x increase in peak bitrate, 10x decrease in latency, 100x increase in traffic capacity over 4G
- **5G NR (new radio):**
  - two frequency bands: FR1 (450 MHz–6 GHz) and FR2 (24 GHz–52 GHz): millimeter wave frequencies
  - not backwards-compatible with 4G
  - MIMO: multiple directional antennae
- **millimeter wave frequencies:** much higher data rates, but over shorter distances
  - pico-cells: cells diameters: 10-100 m
  - massive, dense deployment of new base stations required

# Chapter 7 outline

- Introduction

## Wireless

- Wireless links and network characteristics
- WiFi: 802.11 wireless LANs
- Cellular networks: 4G and 5G

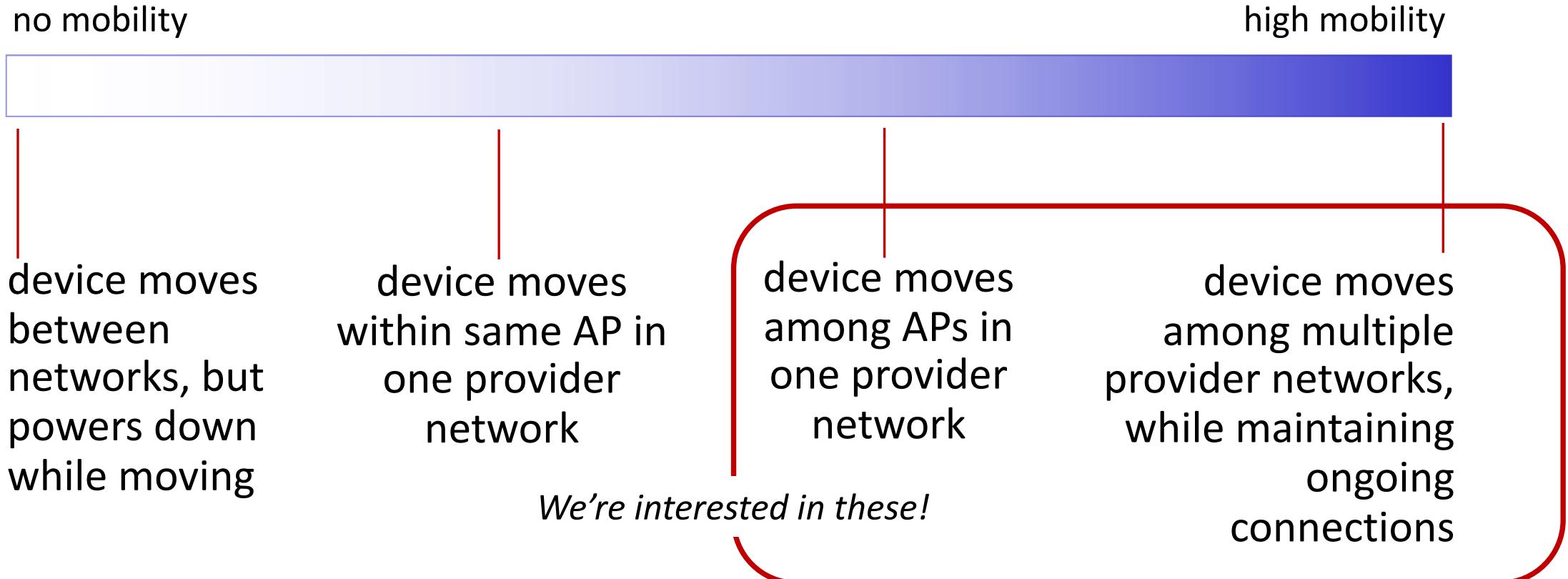


## Mobility

- Mobility management: principles
- Mobility management: practice
  - 4G/5G networks
  - Mobile IP
- Mobility: impact on higher-layer protocols

# What is mobility?

- spectrum of mobility, from the **network** perspective:



# Mobility approaches

- let network (routers) handle it:
  - routers advertise well-known name, address (e.g., permanent 32-bit IP address), or number (e.g., cell #) of visiting mobile node via usual routing table exchange
  - Internet routing could do this already *with no* changes! Routing tables indicate where each mobile located via longest prefix match!

# Mobility approaches

- let network (routers) handle it:
  - routers advertise well-known address (e.g., permanent 32-bit IP address), or number of visiting mobile node via usual routing table exchange
    - not scalable to billions of mobiles
  - Internet routing could do the same *with no changes!* Routing tables indicate where each mobile located via longest prefix match!
- let end-systems handle it: functionality at the “edge”
  - *indirect routing*: communication from correspondent to mobile goes through home network, then forwarded to remote mobile
  - *direct routing*: correspondent gets foreign address of mobile, send directly to mobile

# Contacting a mobile friend:

Consider friend frequently changing locations, how do you find him/her?

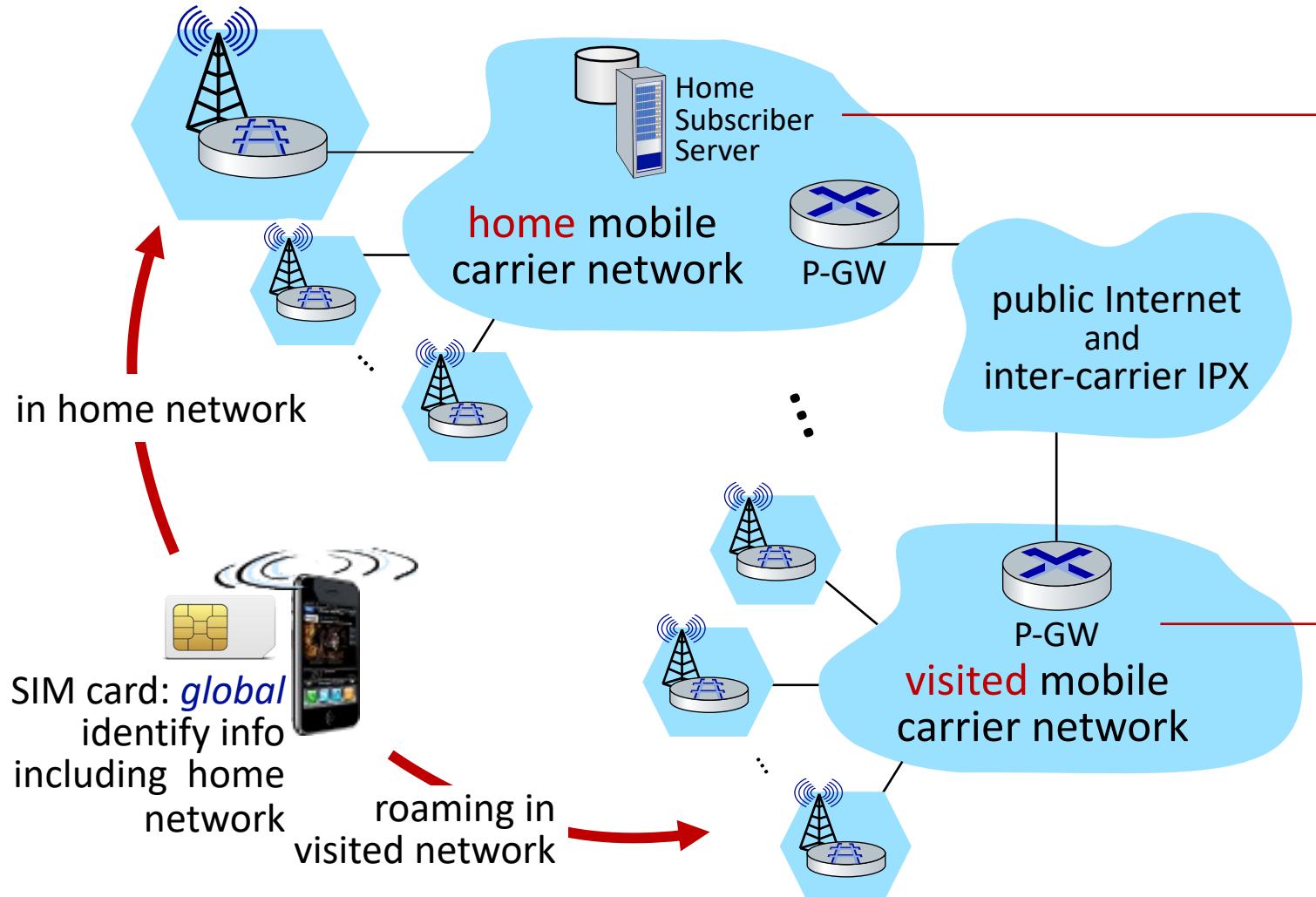
- search all phone books?
- expect her to let you know where he/she is?
- call his/her parents?
- Facebook!



The importance of having a “home”:

- a definitive source of information about you
- a place where people can find out where you are

# Home network, visited network: 4G/5G



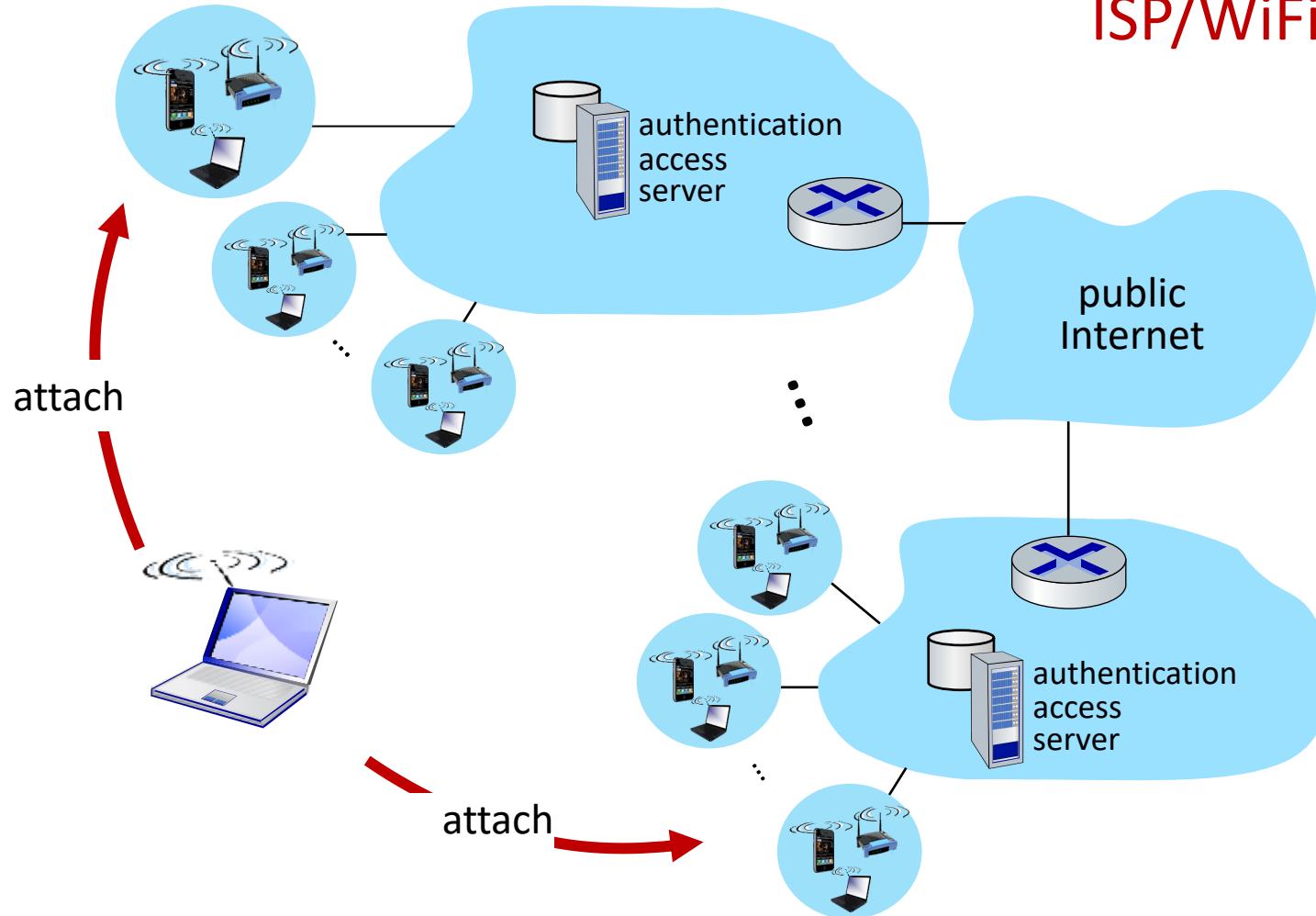
## home network:

- (paid) service plan with cellular provider, e.g., Verizon, Orange
- home network HSS stores identify & services info

## visited network:

- any network other than your home network
- service agreement with other networks: to provide access to visiting mobile

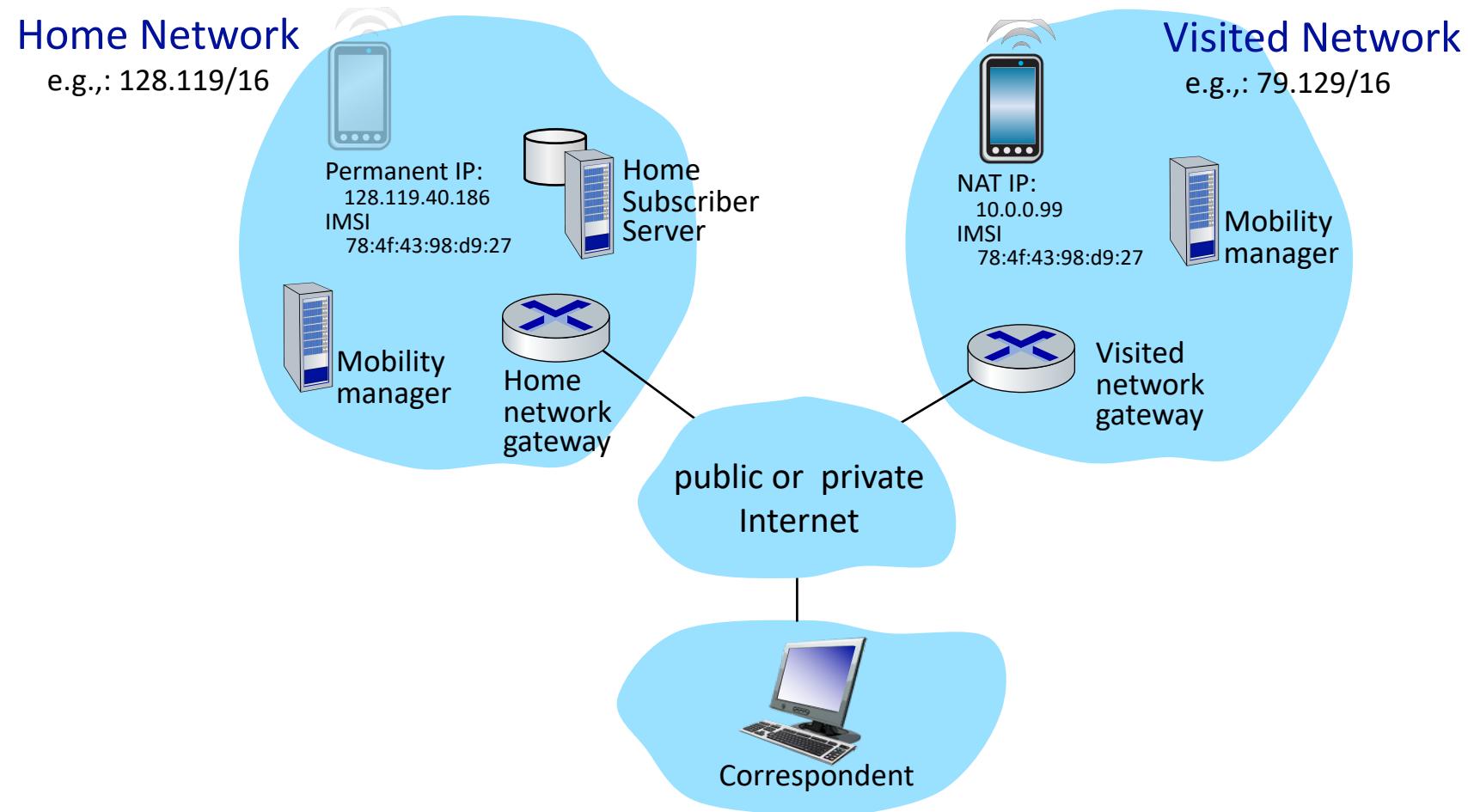
# Home network, visited network: ISP/WiFi



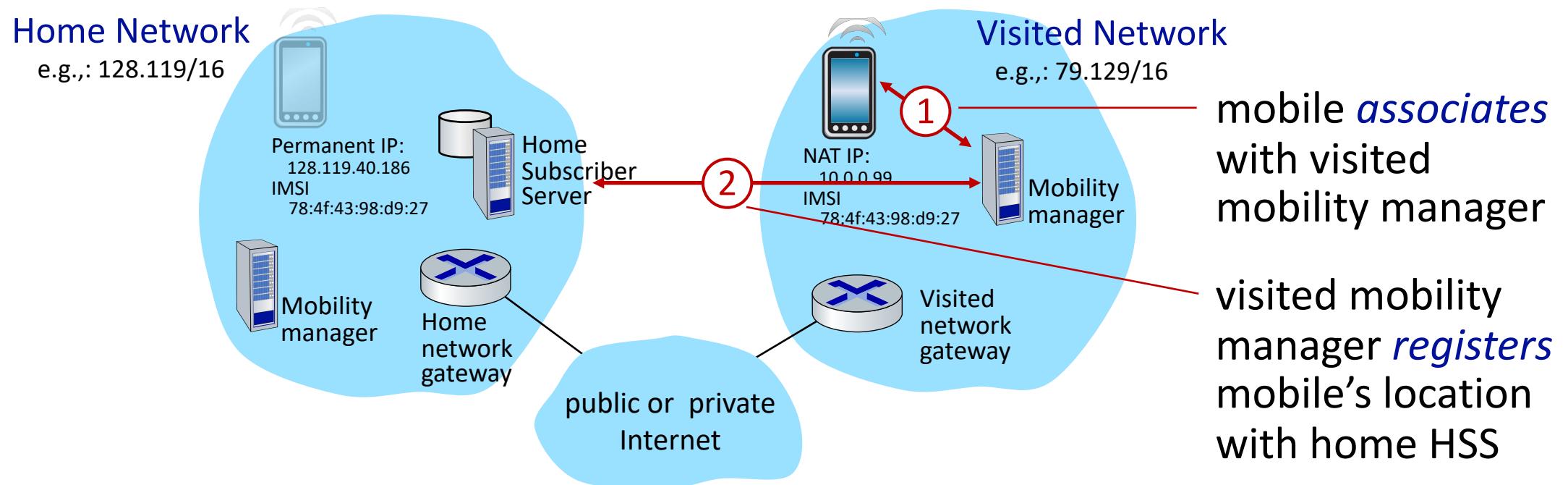
ISP/WiFi: no notion of global “home”

- credentials from ISP (e.g., username, password) stored on device or with user
- ISPs may have national, international presence
- different networks: different credentials
  - some exceptions (e.g., eduroam)
  - architectures exist (mobile IP) for 4G-like mobility, but not used

# Home network, visited network: generic



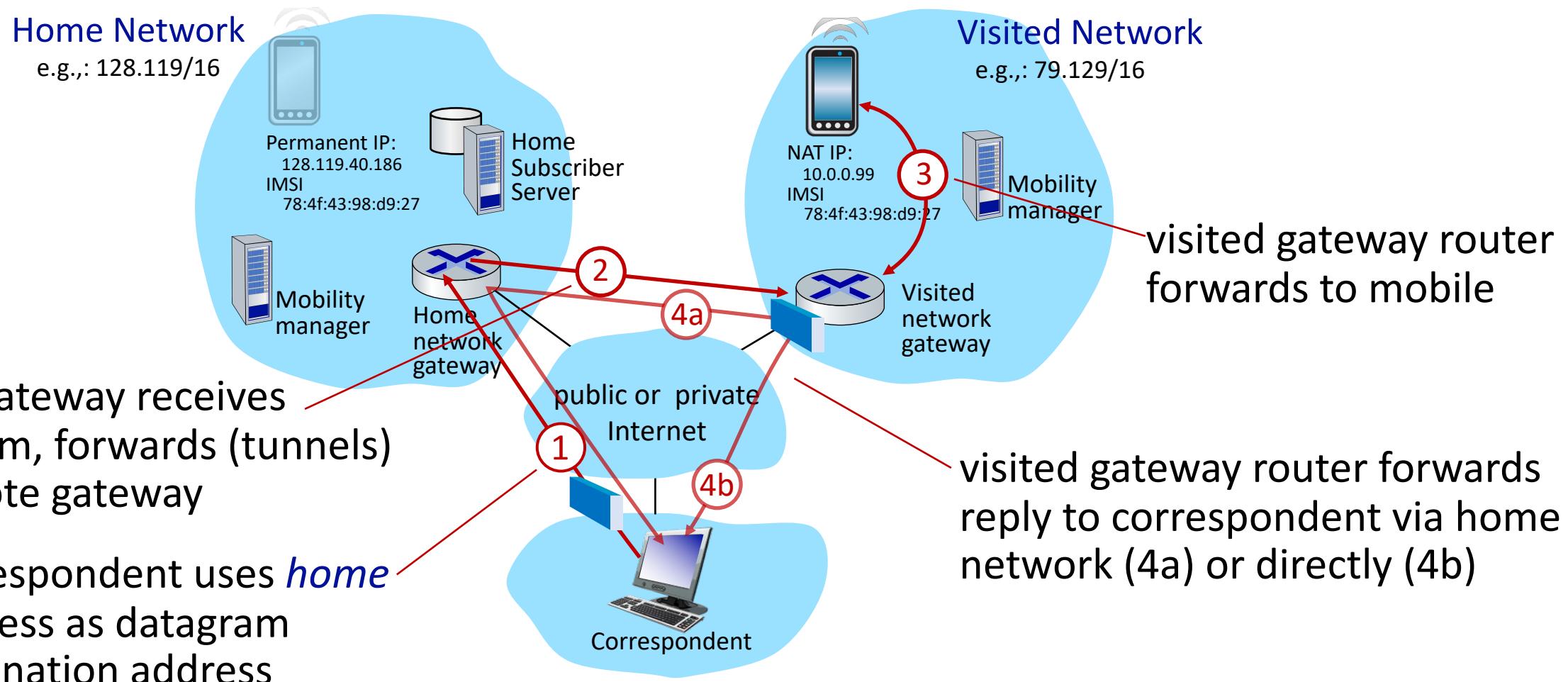
# Registration: home needs to know where you are!



end result:

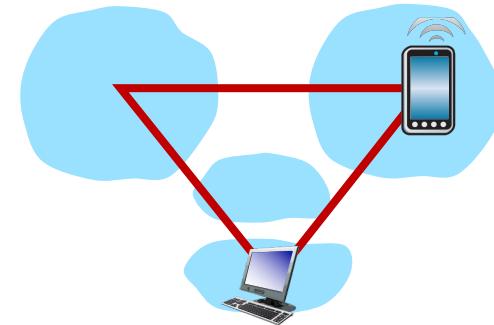
- visited mobility manager knows about mobile
- home HSS knows location of mobile

# Mobility with indirect routing

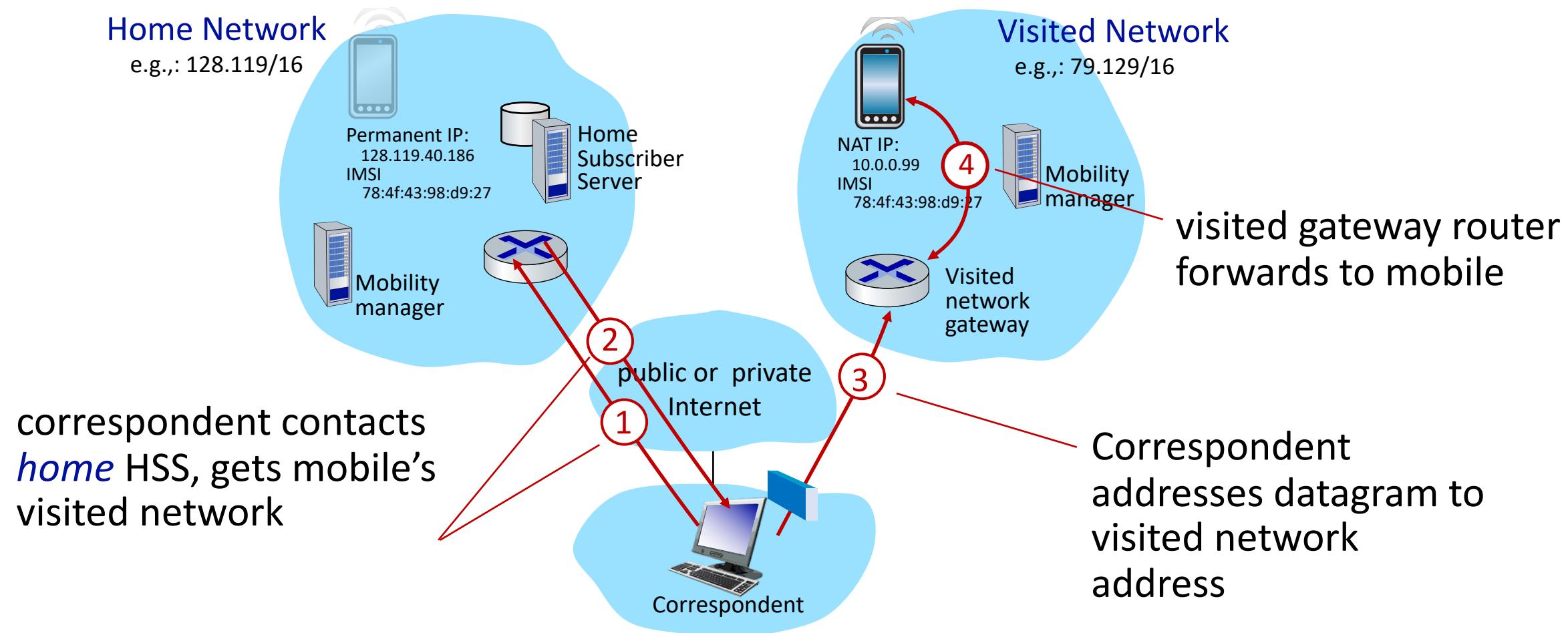


# Mobility with indirect routing: comments

- triangle routing:
  - inefficient when correspondent and mobile are in same network
- mobile moves among visited networks: transparent to correspondent!
  - registers in new visited network
  - new visited network registers with home HSS
  - datagrams continue to be forwarded from home network to mobile in new network
  - *on-going (e.g., TCP) connections between correspondent and mobile can be maintained!*



# Mobility with direct routing



# Mobility with direct routing: comments

- overcomes triangle routing inefficiencies
- *non-transparent to correspondent*: correspondent must get care-of-address from home agent
- what if mobile changes visited network?
  - can be handled, but with additional complexity

# Chapter 7 outline

- Introduction

## Wireless

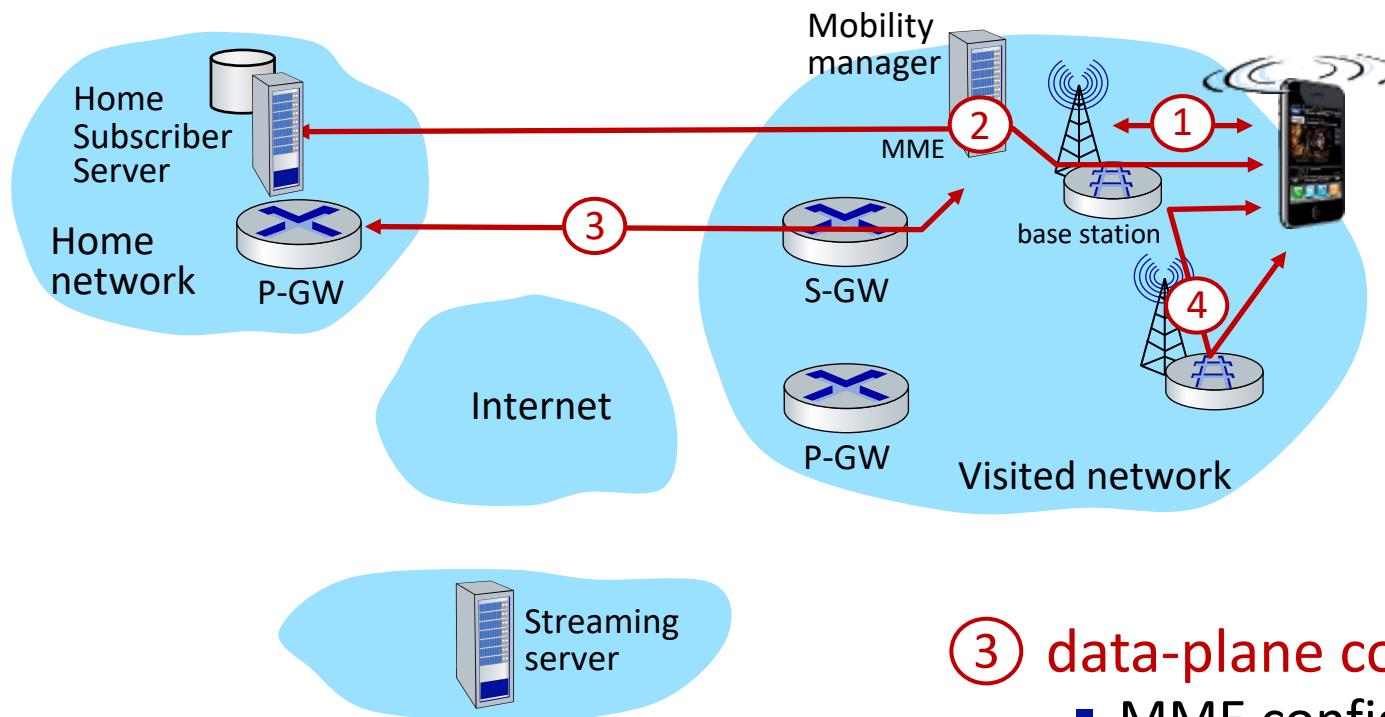
- Wireless links and network characteristics
- WiFi: 802.11 wireless LANs
- Cellular networks: 4G and 5G



## Mobility

- Mobility management: principles
- Mobility management: practice
  - 4G/5G networks
  - Mobile IP
- Mobility: impact on higher-layer protocols

# Mobility in 4G networks: major mobility tasks



## ④ mobile handover:

- mobile device changes its point of attachment to visited network

① **base station association:**

- covered earlier
- mobile provides IMSI – identifying itself, home network

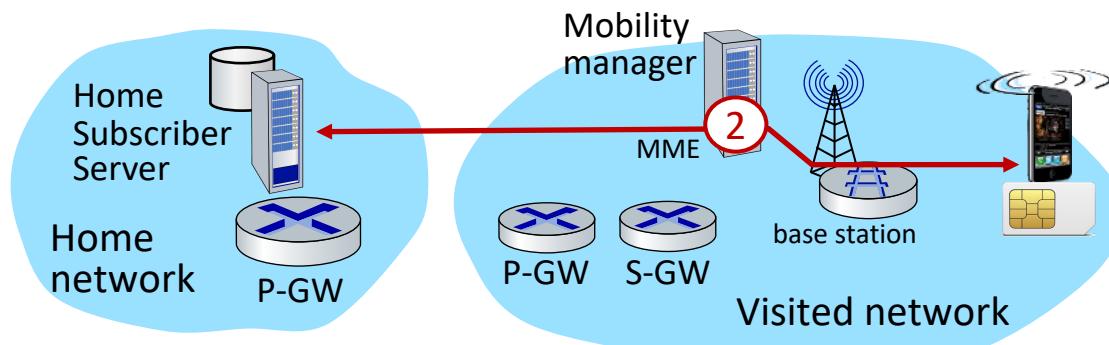
② **control-plane configuration:**

- MME, home HSS establish control-plane state - mobile is in visited network

## ③ data-plane configuration:

- MME configures forwarding tunnels for mobile
- visited, home network establish tunnels from home P-GW to mobile

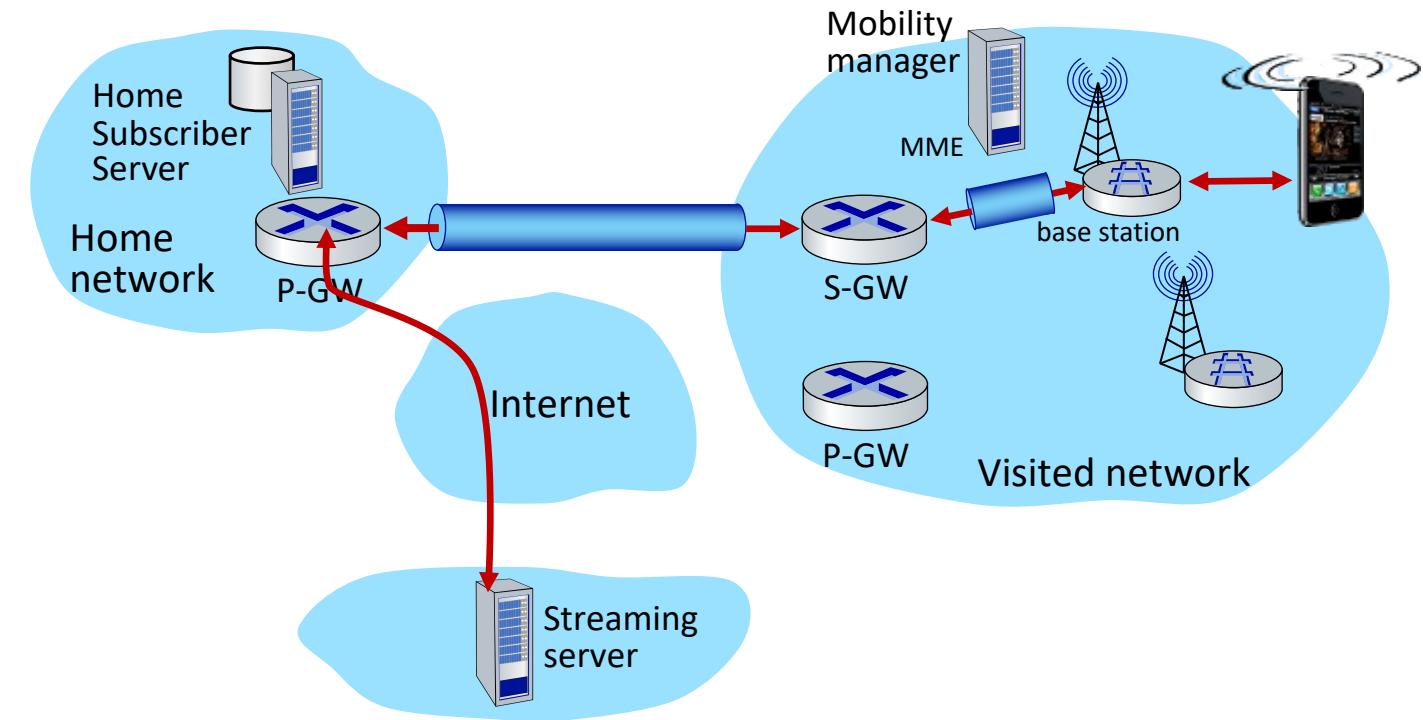
# Configuring LTE control-plane elements



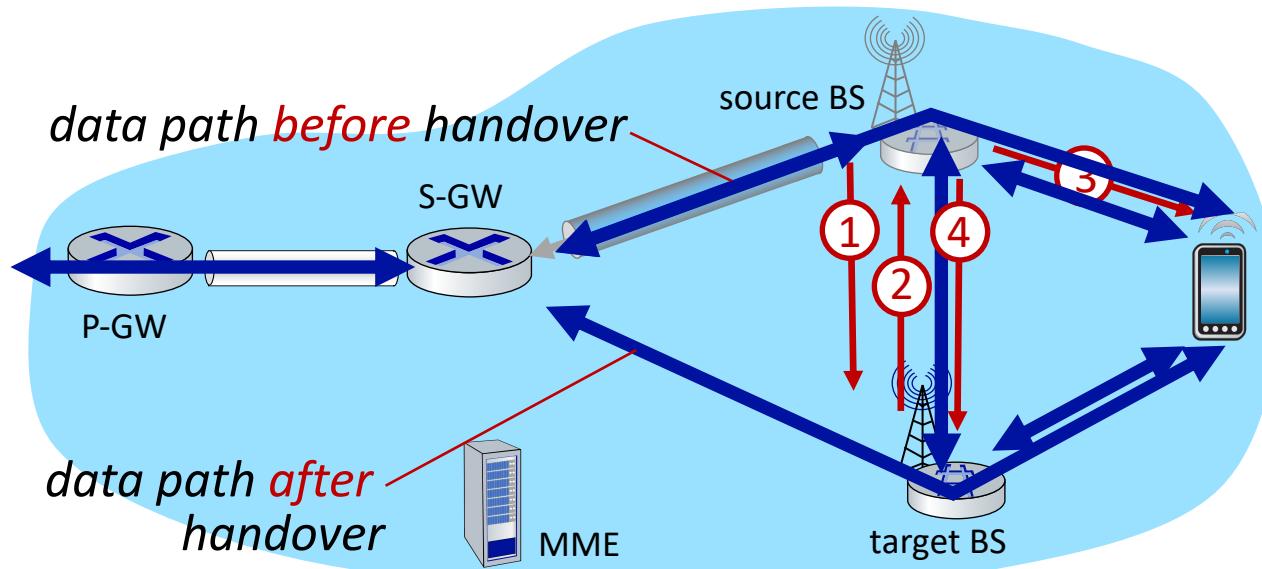
- Mobile communicates with local MME via BS control-plane channel
- MME uses mobile's IMSI info to contact mobile's home HSS
  - retrieve authentication, encryption, network service information
  - home HSS knows mobile now resident in visited network
- BS, mobile select parameters for BS-mobile data-plane radio channel

# Configuring data-plane tunnels for mobile

- **S-GW to BS tunnel:** when mobile changes base stations, simply change endpoint IP address of tunnel
- **S-GW to home P-GW tunnel:** implementation of indirect routing
- **tunneling via GTP (GPRS tunneling protocol):** mobile's datagram to streaming server encapsulated using GTP inside UDP, inside datagram

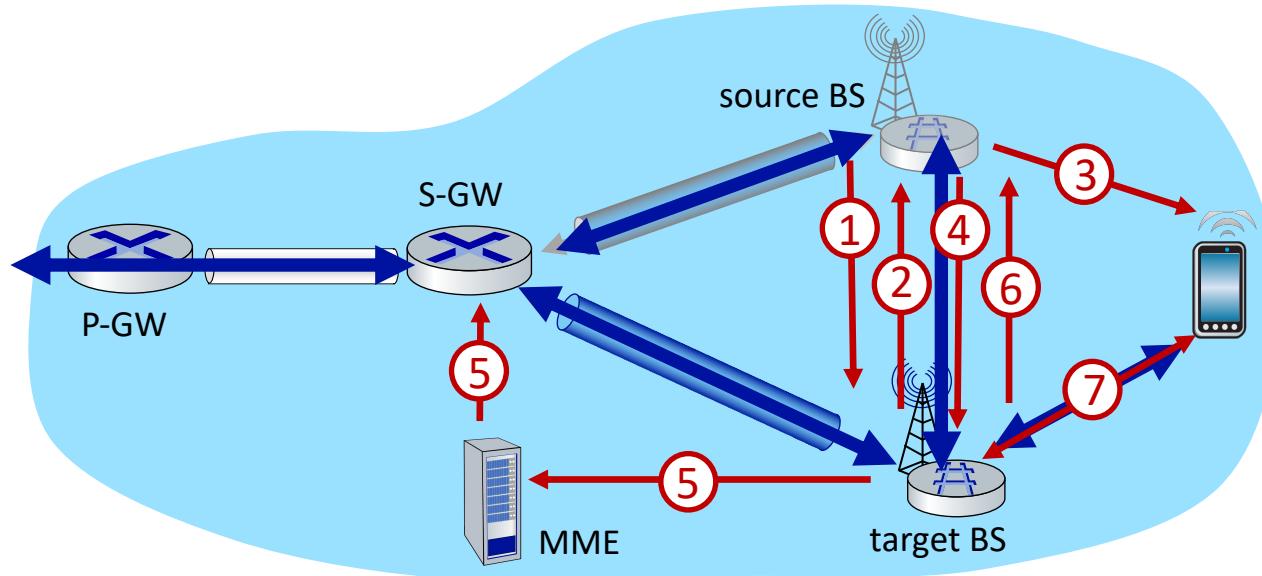


# Handover between BSs in same cellular network



- ① current (source) BS selects target BS, sends *Handover Request* message to target BS
- ② target BS pre-allocates radio time slots, responds with HR ACK with info for mobile
- ③ source BS informs mobile of new BS
  - mobile can now send via new BS - handover *looks complete* to mobile
- ④ source BS stops sending datagrams to mobile, instead forwards to new BS (who forwards to mobile over radio channel)

# Handover between BSs in same cellular network



- ⑤ target BS informs MME that it is new BS for mobile
- MME instructs S-GW to change tunnel endpoint to be (new) target BS

- ⑥ target BS ACKs back to source BS: handover complete, source BS can release resources
- ⑦ mobile's datagrams now flow through new tunnel from target BS to S-GW

# Mobile IP

- mobile IP architecture standardized ~20 years ago [RFC 5944]
  - long before ubiquitous smartphones, 4G support for Internet protocols
  - did not see wide deployment/use
  - perhaps WiFi for Internet, and 2G/3G phones for voice were “good enough” at the time
- mobile IP architecture:
  - indirect routing to node (via home network) using tunnels
  - mobile IP home agent: combined roles of 4G HSS and home P-GW
  - mobile IP foreign agent: combined roles of 4G MME and S-GW
  - protocols for agent discovery in visited network, registration of visited location in home network via ICMP extensions

# Wireless, mobility: impact on higher layer protocols

- logically, impact *should* be minimal ...
  - best effort service model remains unchanged
  - TCP and UDP can (and do) run over wireless, mobile
- ... but performance-wise:
  - packet loss/delay due to bit-errors (discarded packets, delays for link-layer retransmissions), and handover loss
  - TCP interprets loss as congestion, will decrease congestion window unnecessarily
  - delay impairments for real-time traffic
  - bandwidth a scarce resource for wireless links

# Chapter 7 summary

## Wireless

- Wireless Links and network characteristics
- WiFi: 802.11 wireless LANs
- Cellular networks: 4G and 5G



## Mobility

- Mobility management: principles
- Mobility management: practice
  - 4G/5G networks
  - Mobile IP
- Mobility: impact on higher-layer protocols

# Chapter 8 outline

- What is network security?
- Principles of cryptography
- Message integrity, authentication
- Securing e-mail
- Securing TCP connections: TLS
- Network layer security: IPsec
- Security in wireless and mobile networks
- Operational security: firewalls and IDS



# What is network security?

**confidentiality:** only sender, intended receiver should “understand” message contents

- sender encrypts message
- receiver decrypts message

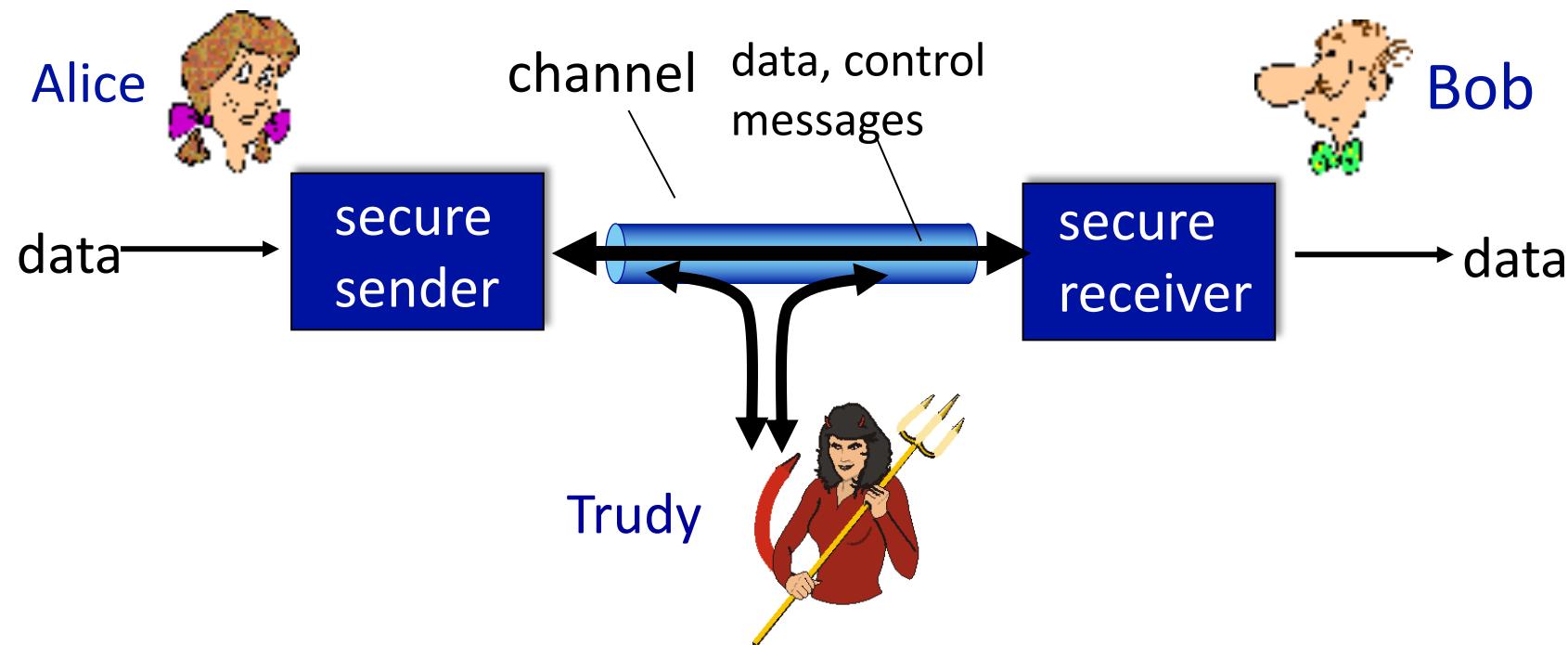
**authentication:** sender, receiver want to confirm identity of each other

**message integrity:** sender, receiver want to ensure message not altered (in transit, or afterwards) without detection

**access and availability:** services must be accessible and available to users

# Friends and enemies: Alice, Bob, Trudy

- well-known in network security world
- Bob, Alice want to communicate “securely”
- Trudy (intruder) may intercept, delete, add messages



# Friends and enemies: Alice, Bob, Trudy

Who might Bob and Alice be?

- ... well, *real-life* Bobs and Alices!
- Web browser/server for electronic transactions (e.g., on-line purchases)
- on-line banking client/server
- DNS servers
- BGP routers exchanging routing table updates
- other examples?

# There are bad people out there!

Q: What can a “bad person” do?

A: A lot!

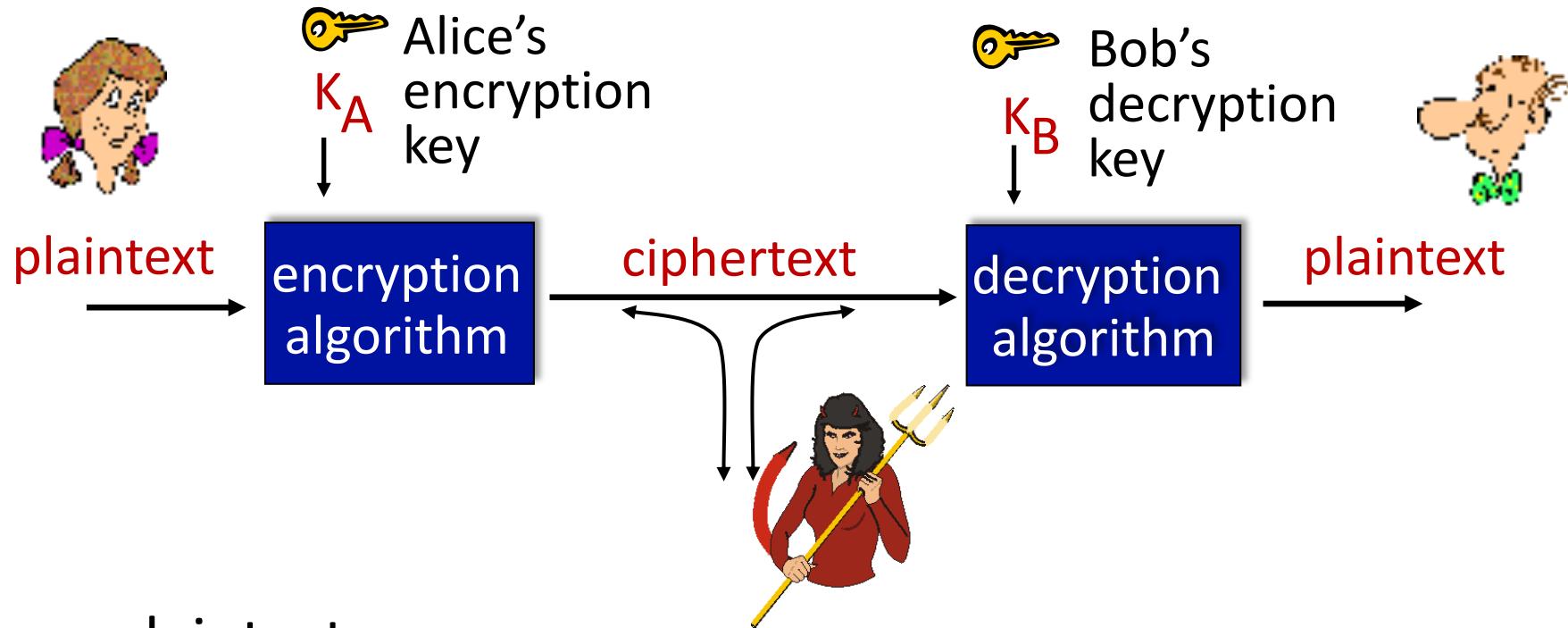
- **eavesdrop**: intercept messages
- actively **insert** messages into connection
- **impersonation**: can fake (spoof) source address in packet (or any field in packet)
- **hijacking**: “take over” ongoing connection by removing sender or receiver, inserting himself in place
- **denial of service**: prevent service from being used by others (e.g., by overloading resources)

# Chapter 8 outline

- What is network security?
- **Principles of cryptography**
- Message integrity, authentication
- Securing e-mail
- Securing TCP connections: TLS
- Network layer security: IPsec
- Security in wireless and mobile networks
- Operational security: firewalls and IDS



# The language of cryptography



$m$ : plaintext message

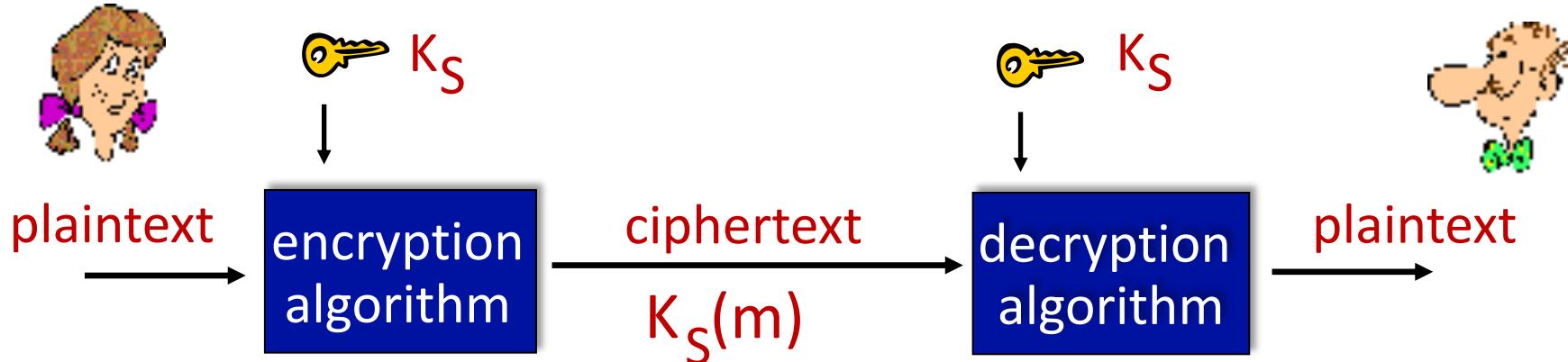
$K_A(m)$ : ciphertext, encrypted with key  $K_A$

$m = K_B(K_A(m))$

# Breaking an encryption scheme

- **cipher-text only attack:**  
Trudy has ciphertext she can analyze
- **two approaches:**
  - brute force: search through all keys
  - statistical analysis
- **known-plaintext attack:**  
Trudy has plaintext corresponding to ciphertext
  - e.g., in monoalphabetic cipher, Trudy determines pairings for a,l,i,c,e,b,o,
- **chosen-plaintext attack:**  
Trudy can get ciphertext for chosen plaintext

# Symmetric key cryptography



**symmetric key crypto:** Bob and Alice share same (symmetric) key:  $K$

- e.g., key is knowing substitution pattern in mono alphabetic substitution cipher

Q: how do Bob and Alice agree on key value?

# Simple encryption scheme

*substitution cipher:* substituting one thing for another

- monoalphabetic cipher: substitute one letter for another

plaintext: abcdefghijklmnopqrstuvwxyz

ciphertext: mnbvctxzasdfghjklpoiuytrewq

e.g.: Plaintext: bob. i love you. alice

ciphertext: nkn. s gktc wky. mgsbc



*Encryption key:* mapping from set of 26 letters  
to set of 26 letters

# A more sophisticated encryption approach

- n substitution ciphers,  $M_1, M_2, \dots, M_n$
  - cycling pattern:
    - e.g.,  $n=4$ :  $M_1, M_3, M_4, M_3, M_2; M_1, M_3, M_4, M_3, M_2; \dots$
  - for each new plaintext symbol, use subsequent substitution pattern in cyclic pattern
    - dog: d from  $M_1$ , o from  $M_3$ , g from  $M_4$
-  **Encryption key:** n substitution ciphers, and cyclic pattern
  - key need not be just n-bit pattern

# Symmetric key crypto: DES

## DES: Data Encryption Standard

- US encryption standard [NIST 1993]
- 56-bit symmetric key, 64-bit plaintext input
- block cipher with cipher block chaining
- how secure is DES?
  - DES Challenge: 56-bit-key-encrypted phrase decrypted (brute force) in less than a day
  - no known good analytic attack
- making DES more secure:
  - 3DES: encrypt 3 times with 3 different keys

# AES: Advanced Encryption Standard

- symmetric-key NIST standard, replaced DES (Nov 2001)
- processes data in 128 bit blocks
- 128, 192, or 256 bit keys
- brute force decryption (try each key) taking 1 sec on DES, takes 149 trillion years for AES

# Public Key Cryptography

## symmetric key crypto:

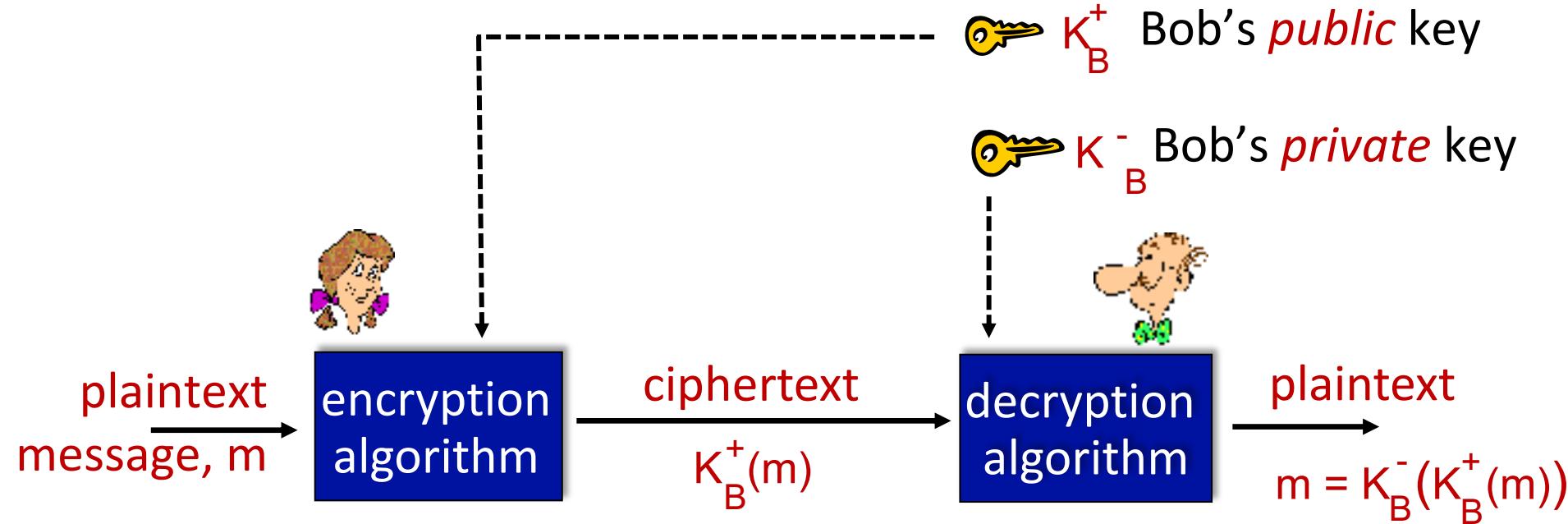
- requires sender, receiver know shared secret key
- Q: how to agree on key in first place (particularly if never “met”)?

## public key crypto

- *radically* different approach [Diffie-Hellman76, RSA78]
- sender, receiver do *not* share secret key
- *public* encryption key known to *all*
- *private* decryption key known only to receiver



# Public Key Cryptography



**Wow** - public key cryptography revolutionized 2000-year-old (previously only symmetric key) cryptography!

- similar ideas emerged at roughly same time, independently in US and UK

# Public key encryption algorithms

requirements:

- ① need  $K_B^+(\cdot)$  and  $K_B^-(\cdot)$  such that

$$K_B^-(K_B^+(m)) = m$$

- ② given public key  $K_B^+$ , it should be impossible to compute private key  $K_B^-$

**RSA:** Rivest, Shamir, Adelson algorithm

# Why is RSA secure?

- suppose you know Bob's public key ( $n, e$ ). How hard is it to determine  $d$ ?
- essentially need to find factors of  $n$  without knowing the two factors  $p$  and  $q$ 
  - fact: factoring a big number is hard

# RSA in practice: session keys

- exponentiation in RSA is computationally intensive
- DES is at least 100 times faster than RSA
- use public key crypto to establish secure connection, then establish second key – symmetric session key – for encrypting data

## session key, $K_s$

- Bob and Alice use RSA to exchange a symmetric session key  $K_s$
- once both have  $K_s$ , they use symmetric key cryptography

# Chapter 8 outline

- What is network security?
- Principles of cryptography
- **Authentication**, message integrity
- Securing e-mail
- Securing TCP connections: TLS
- Network layer security: IPsec
- Security in wireless and mobile networks
- Operational security: firewalls and IDS



# Authentication

**Goal:** Bob wants Alice to “prove” her identity to him

**Protocol ap1.0:** Alice says “I am Alice”



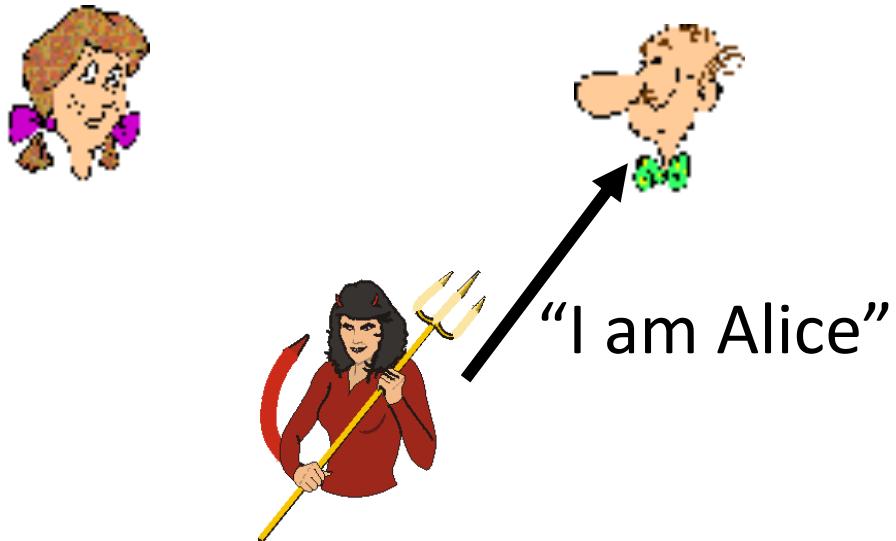
*failure scenario??*



# Authentication

**Goal:** Bob wants Alice to “prove” her identity to him

**Protocol ap1.0:** Alice says “I am Alice”



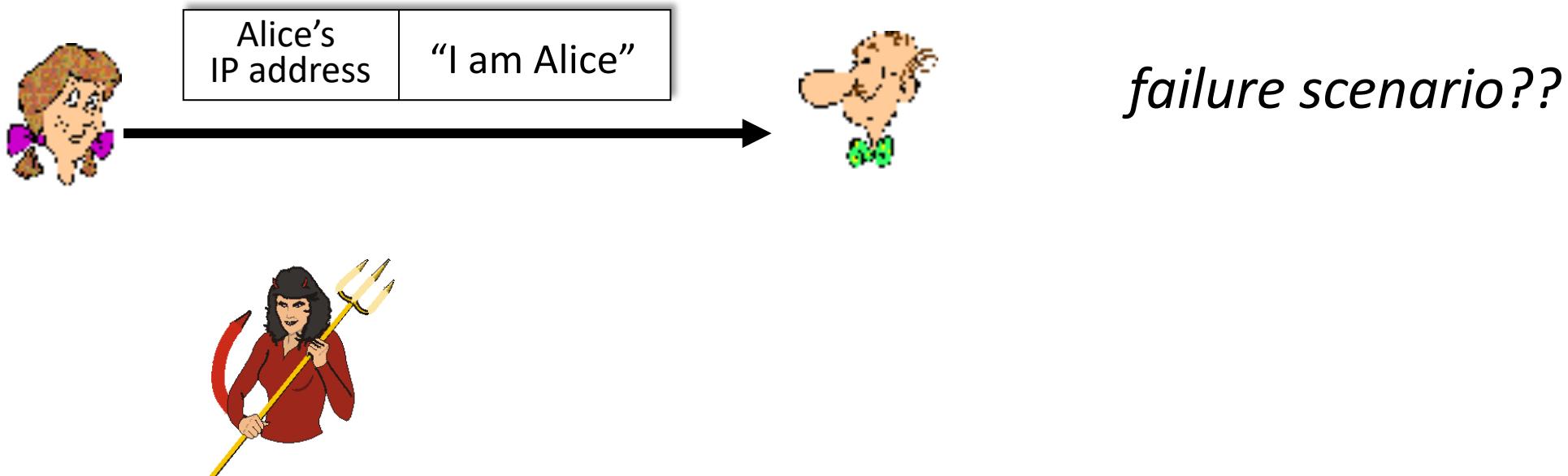
*in a network, Bob can not “see” Alice, so Trudy simply declares herself to be Alice*



# Authentication: another try

**Goal:** Bob wants Alice to “prove” her identity to him

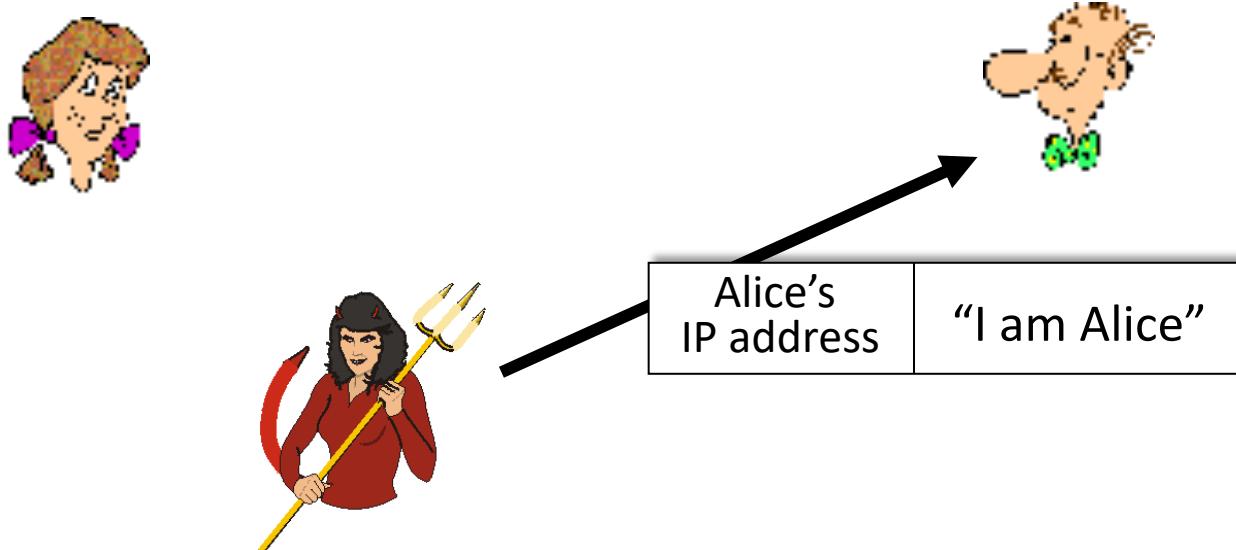
**Protocol ap2.0:** Alice says “I am Alice” in an IP packet containing her source IP address



# Authentication: another try

**Goal:** Bob wants Alice to “prove” her identity to him

**Protocol ap2.0:** Alice says “I am Alice” in an IP packet containing her source IP address

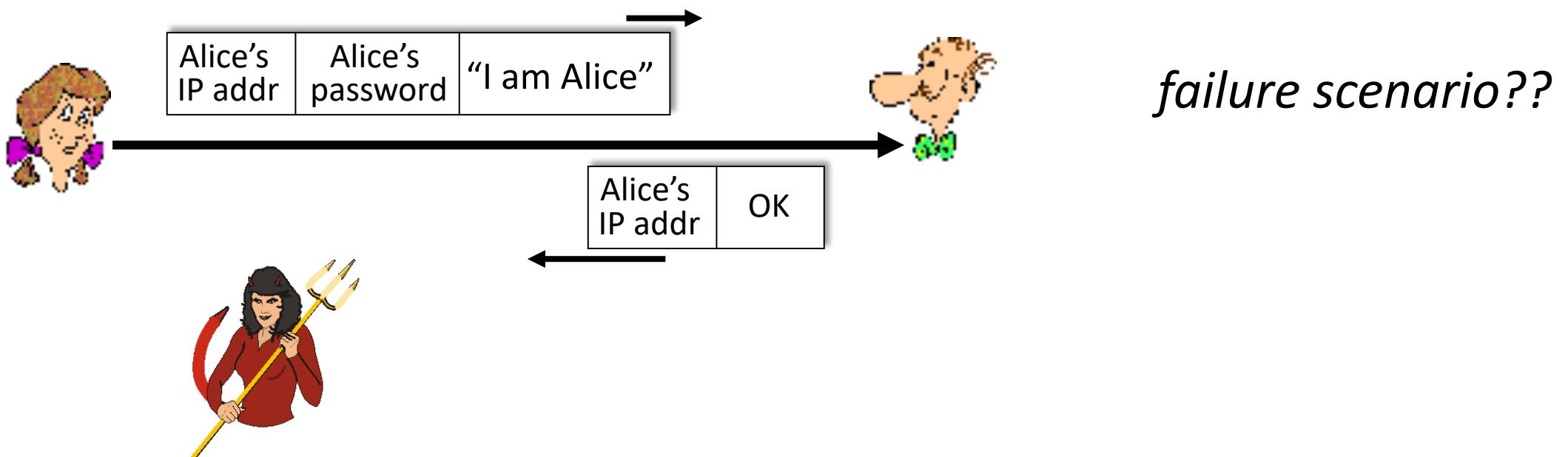


*Trudy can create  
a packet “spoofing”  
Alice’s address*

# Authentication: a third try

**Goal:** Bob wants Alice to “prove” her identity to him

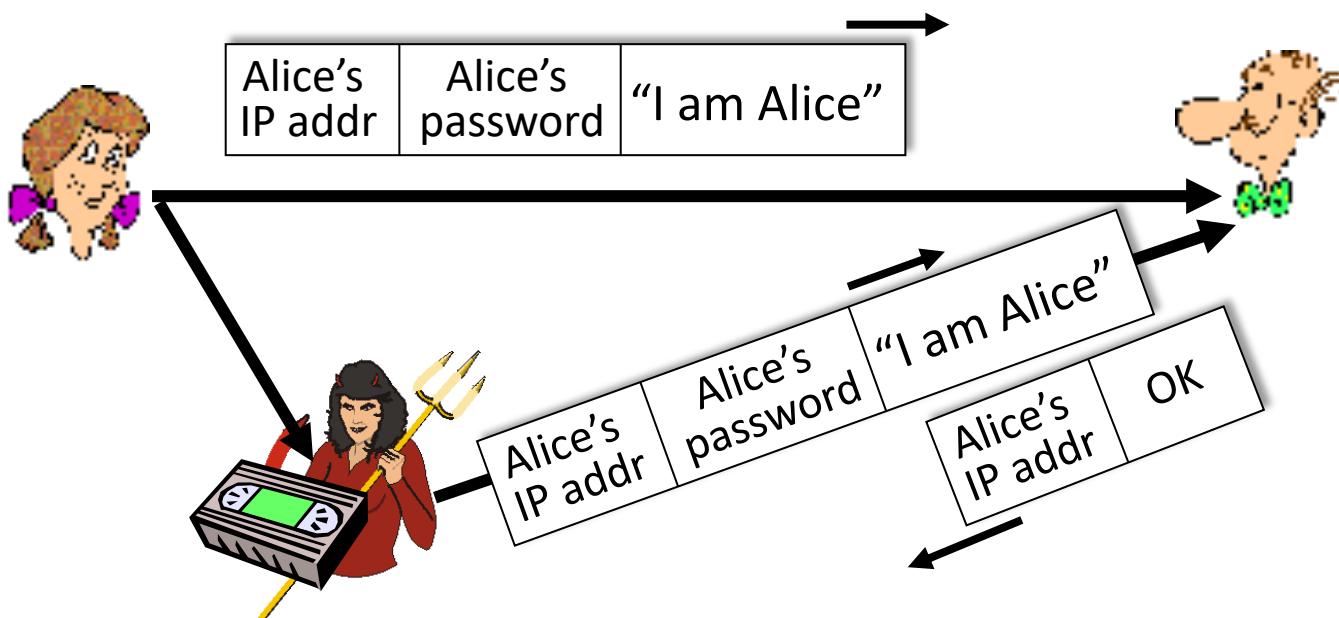
**Protocol ap3.0:** Alice says “I am Alice” and sends her secret password to “prove” it.



# Authentication: a third try

**Goal:** Bob wants Alice to “prove” her identity to him

**Protocol ap3.0:** Alice says “I am Alice” Alice says “I am Alice” and sends her secret password to “prove” it.

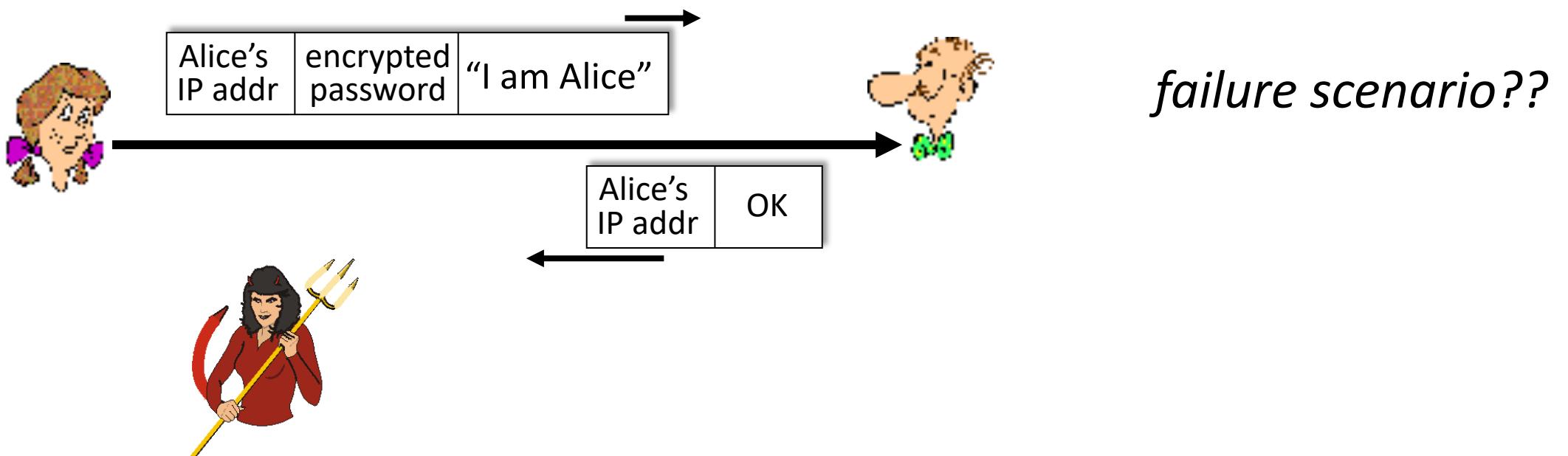


*playback attack:  
Trudy records  
Alice's packet  
and later  
plays it back to Bob*

# Authentication: a modified third try

**Goal:** Bob wants Alice to “prove” her identity to him

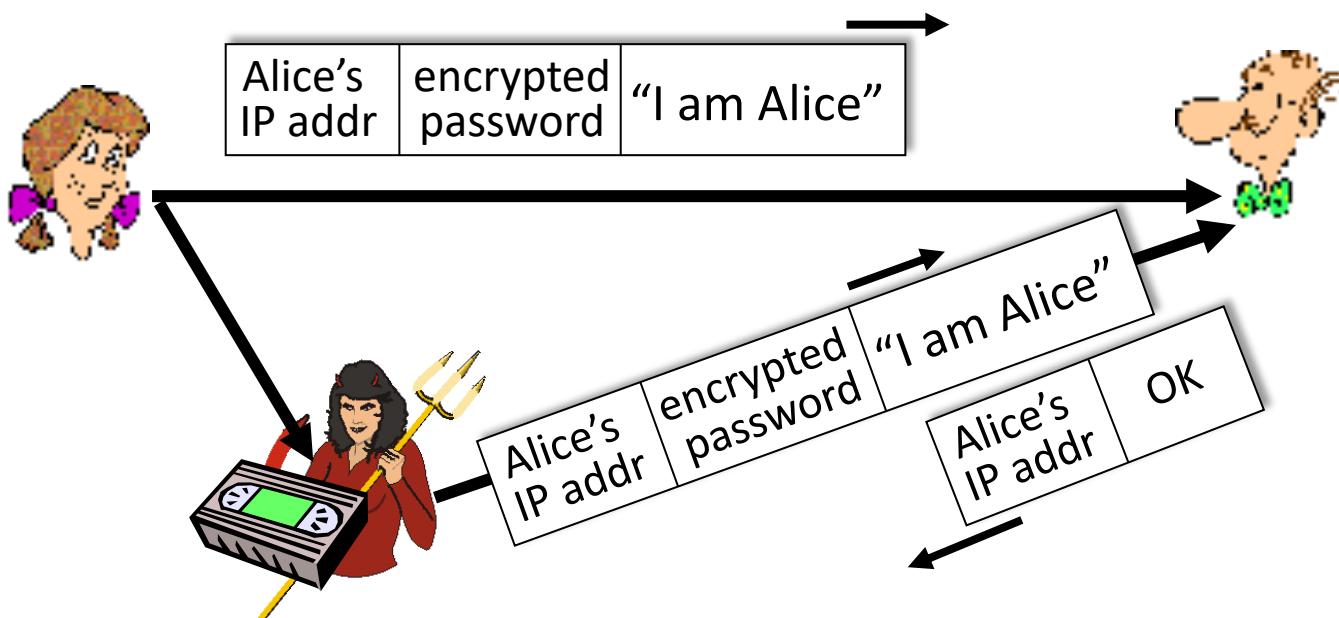
**Protocol ap3.0:** Alice says “I am Alice” and sends her encrypted secret password to “prove” it.



# Authentication: a modified third try

**Goal:** Bob wants Alice to “prove” her identity to him

**Protocol ap3.0:** Alice says “I am Alice” and sends her encrypted secret password to “prove” it.



*playback attack still works: Trudy records Alice's packet and later plays it back to Bob*

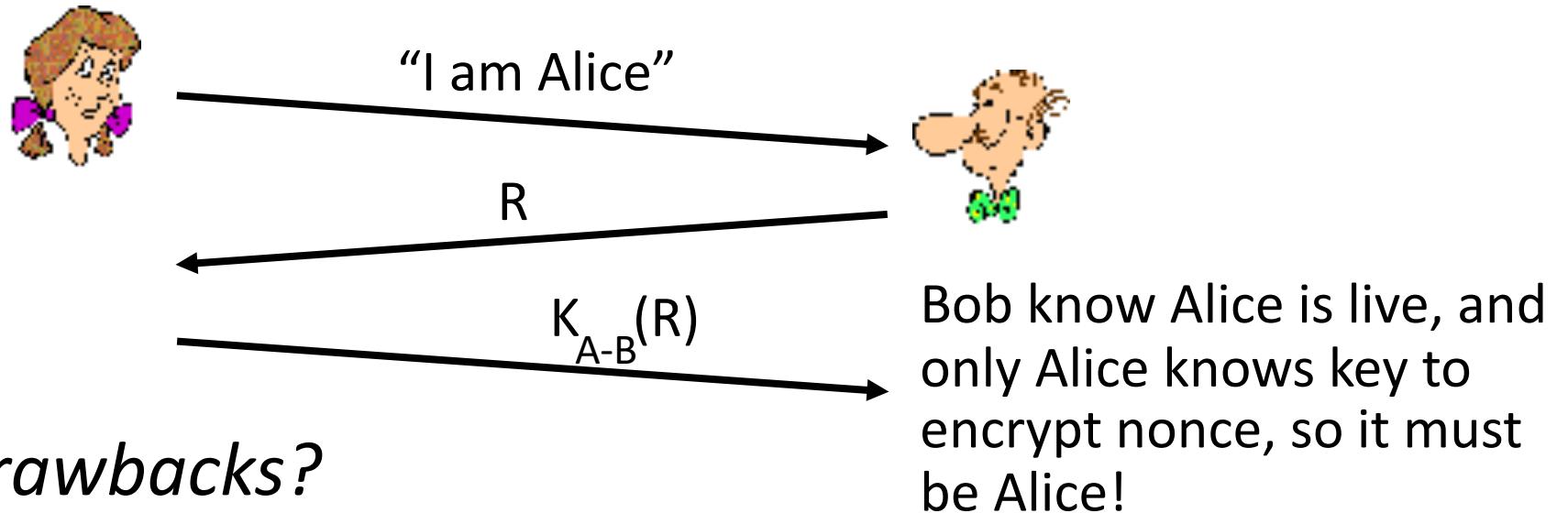
# Authentication: a fourth try

**Goal:** avoid playback attack

**nonce:** number (R) used only **once-in-a-lifetime**

**protocol ap4.0:** to prove Alice “live”, Bob sends Alice nonce, R

- Alice must return R, encrypted with shared secret key

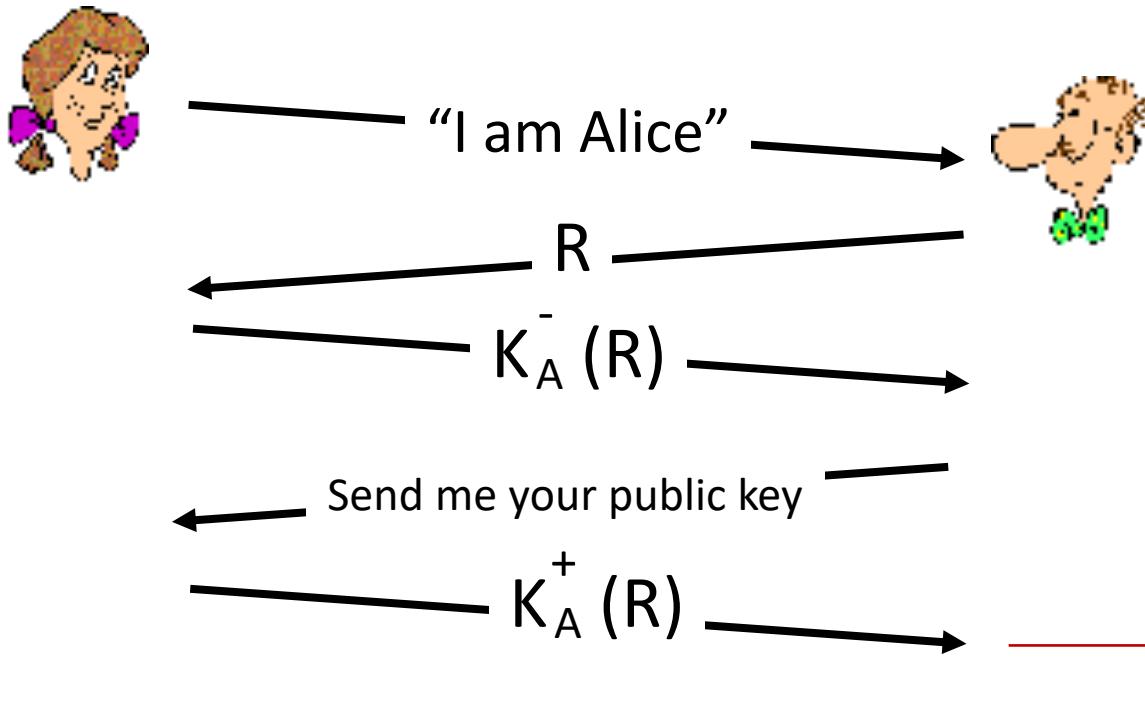


*Failures, drawbacks?*

# Authentication: ap5.0

ap4.0 requires shared symmetric key - can we authenticate using public key techniques?

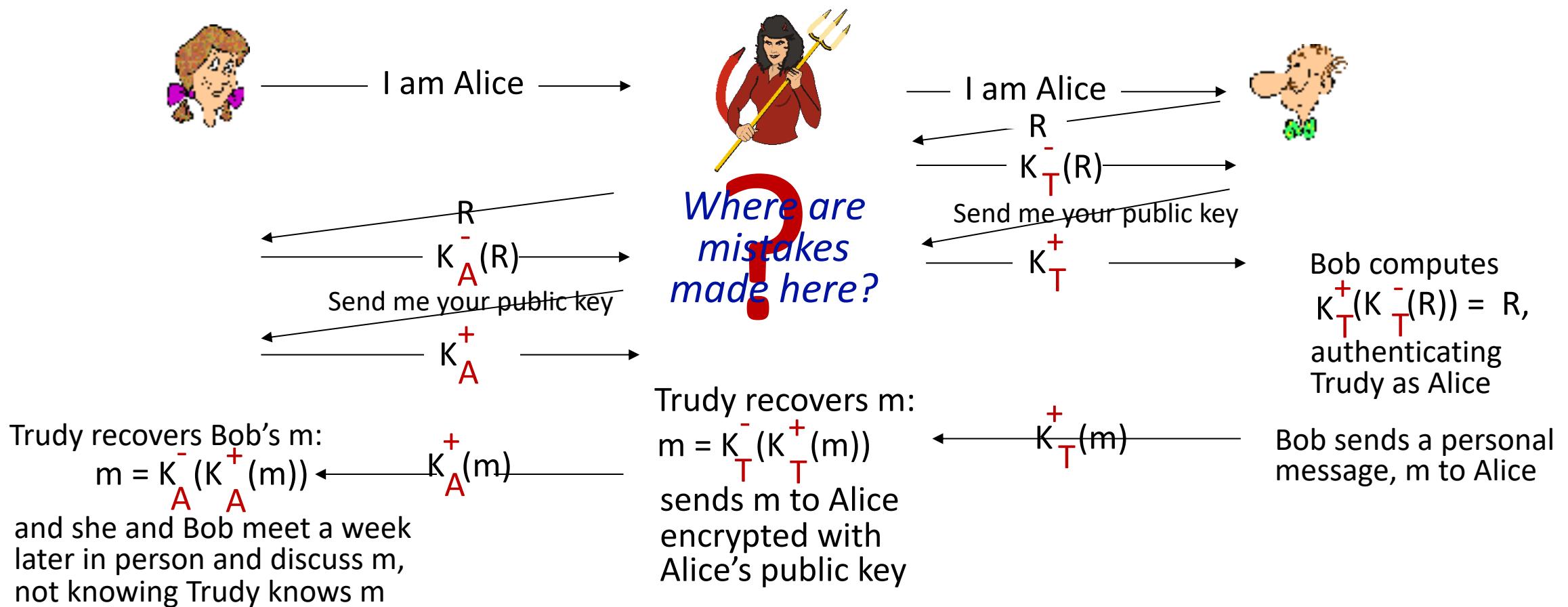
ap5.0: use nonce, public key cryptography



Bob computes  
 $K_A^+ (K_A^-(R)) = R$   
and knows only Alice could have the private key, that encrypted  $R$  such that  
 $K_A^+ (K_A^-(R)) = R$

# Authentication: ap5.0 – there's still a flaw!

man (or woman) in the middle attack: Trudy poses as Alice (to Bob) and as Bob (to Alice)



# Chapter 8 outline

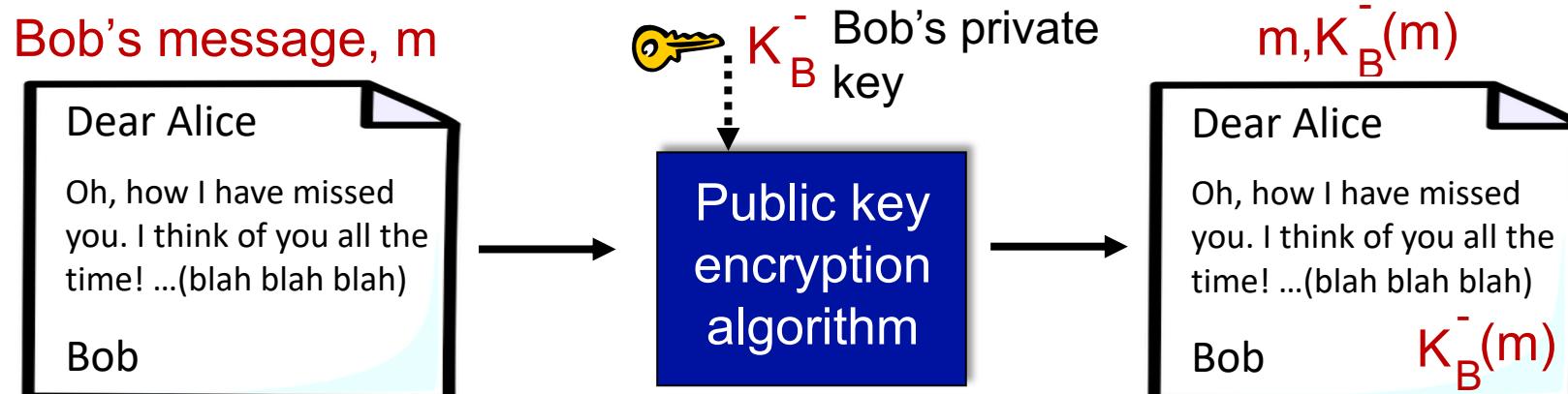
- What is network security?
- Principles of cryptography
- Authentication, **message integrity**
- Securing e-mail
- Securing TCP connections: TLS
- Network layer security: IPsec
- Security in wireless and mobile networks
- Operational security: firewalls and IDS



# Digital signatures

cryptographic technique analogous to hand-written signatures:

- sender (Bob) digitally signs document: he is document owner/creator.
- *verifiable, nonforgeable*: recipient (Alice) can prove to someone that Bob, and no one else (including Alice), must have signed document
- simple digital signature for message  $m$ :
  - Bob signs  $m$  by encrypting with his private key  $K_B^-$ , creating “signed” message,  $K_B^-(m)$



# Digital signatures

- suppose Alice receives msg  $m$ , with signature:  $m, K_B^-(m)$
- Alice verifies  $m$  signed by Bob by applying Bob's public key  $K_B^+$  to  $K_B^-(m)$  then checks  $K_B^+(K_B^-(m)) = m$ .
- If  $K_B^+(K_B^-(m)) = m$ , whoever signed  $m$  must have used Bob's private key

Alice thus verifies that:

- Bob signed  $m$
- no one else signed  $m$
- Bob signed  $m$  and not  $m'$

non-repudiation:

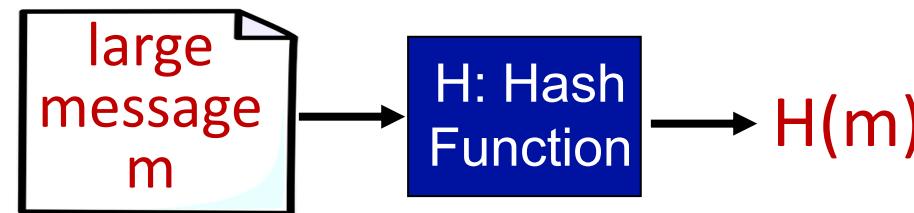
- ✓ Alice can take  $m$ , and signature  $K_B^-(m)$  to court and prove that Bob signed  $m$

# Message digests

computationally expensive to public-key-encrypt long messages

**goal:** fixed-length, easy- to-compute digital “fingerprint”

- apply hash function  $H$  to  $m$ , get fixed size message digest,  $H(m)$



**Hash function properties:**

- many-to-1
- produces fixed-size msg digest (fingerprint)
- given message digest  $x$ , computationally infeasible to find  $m$  such that  $x = H(m)$

# Internet checksum: poor crypto hash function

Internet checksum has some properties of hash function:

- produces fixed length digest (16-bit sum) of message
- is many-to-one

but given message with given hash value, it is easy to find another message with same hash value:

<u>message</u>	<u>ASCII format</u>
I O U 1	49 4F 55 31
0 0 . 9	30 30 2E 39
9 B O B	39 42 D2 42

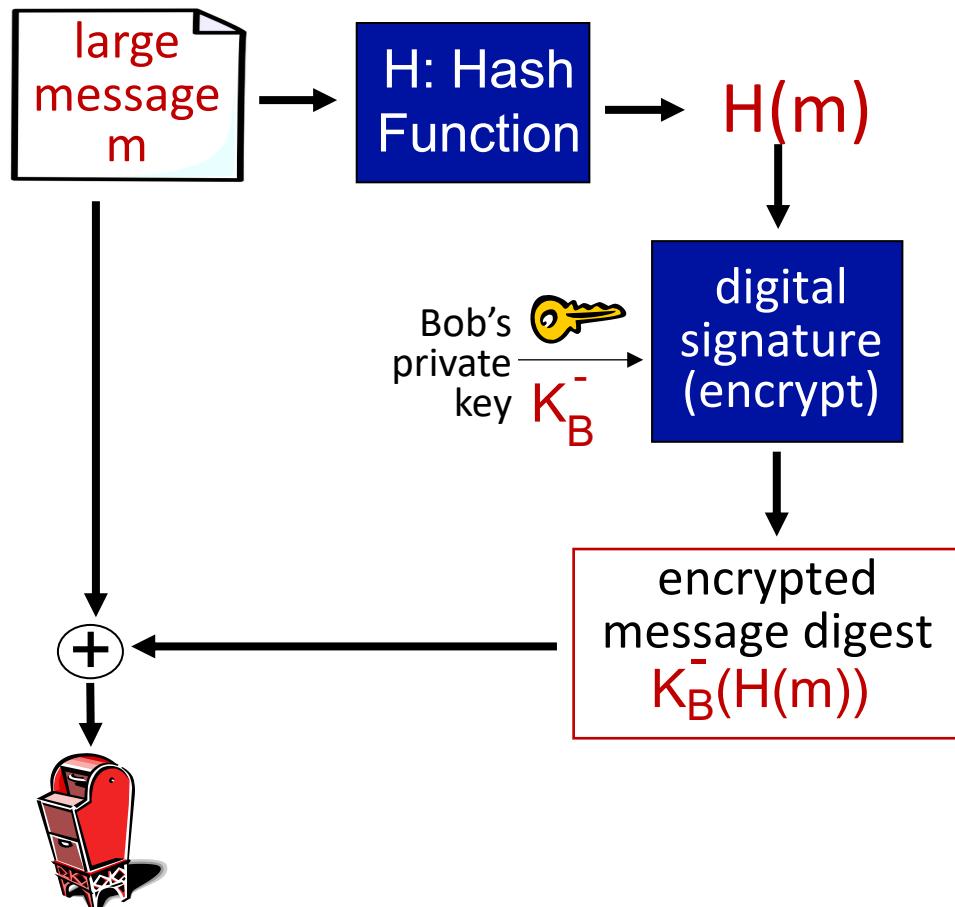
<u>message</u>	<u>ASCII format</u>
I O U 9	49 4F 55 <u>39</u>
0 0 . 1	30 30 2E <u>31</u>
9 B O B	39 42 D2 42

*different messages*

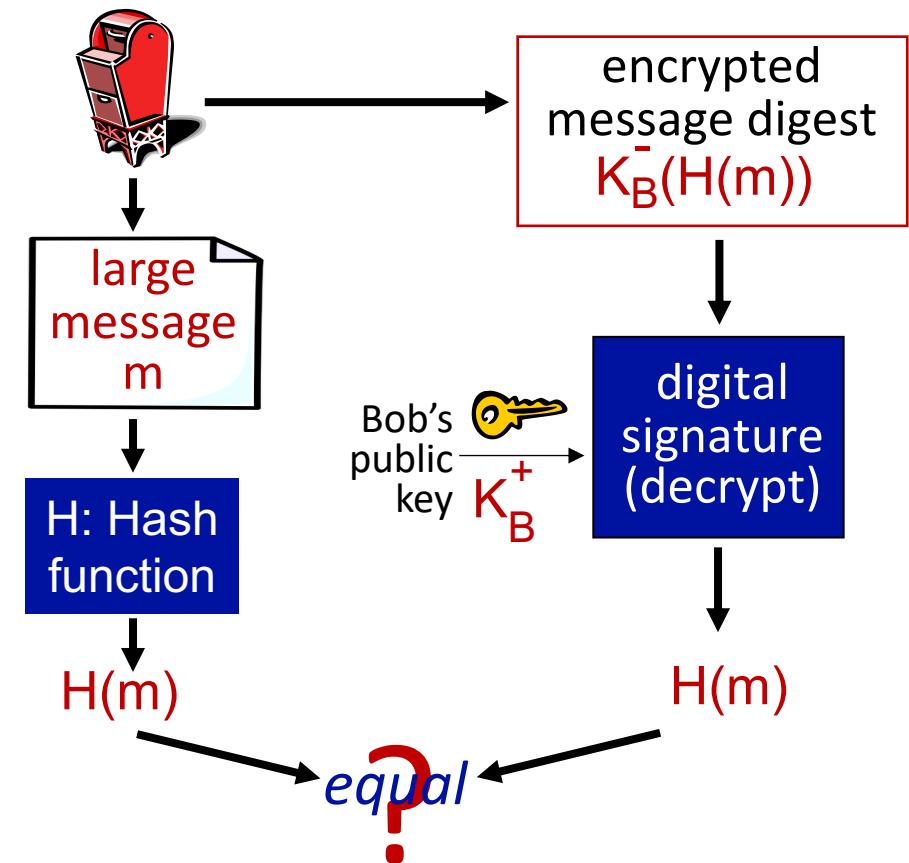
*but identical checksums!*

# Digital signature = signed message digest

Bob sends digitally signed message:



Alice verifies signature, integrity of digitally signed message:

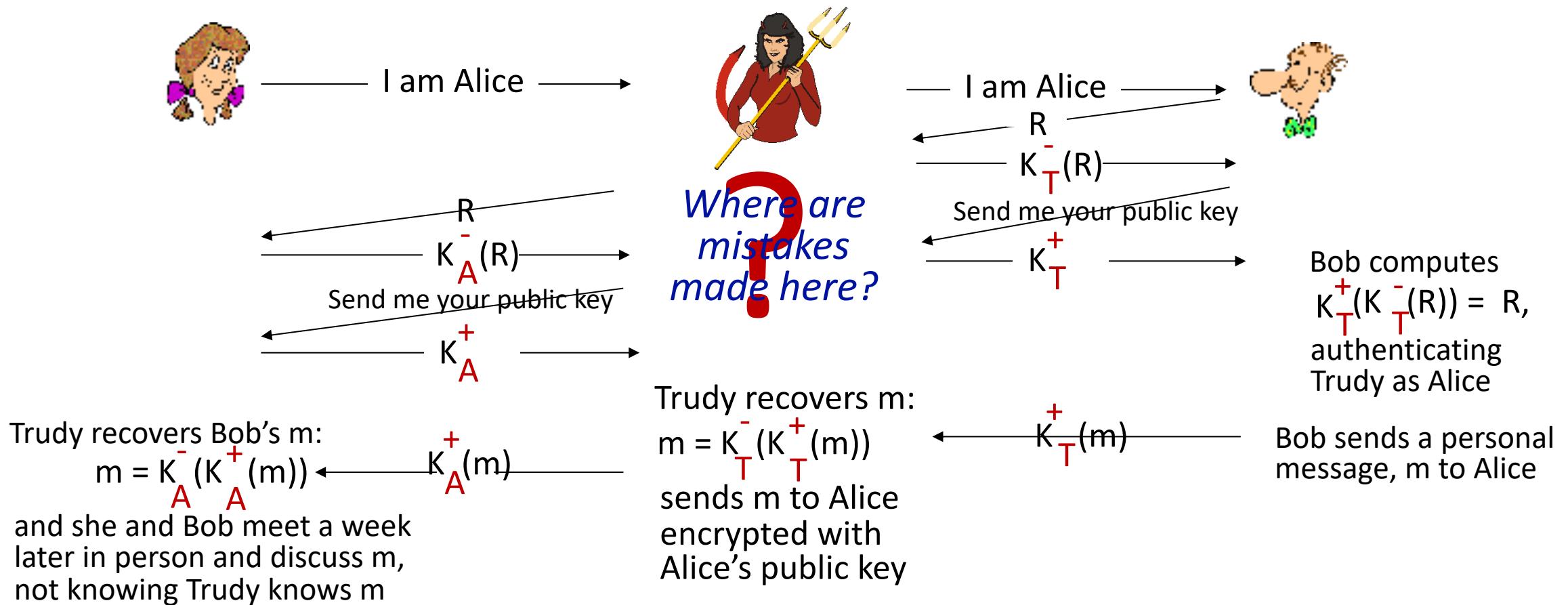


# Hash function algorithms

- MD5 hash function widely used (RFC 1321)
  - computes 128-bit message digest in 4-step process.
  - arbitrary 128-bit string  $x$ , appears difficult to construct msg  $m$  whose MD5 hash is equal to  $x$
- SHA-1 is also used
  - US standard [NIST, FIPS PUB 180-1]
  - 160-bit message digest

# Authentication: ap5.0 – let's fix it!!

Recall the problem: Trudy poses as Alice (to Bob) and as Bob (to Alice)



# Need for certified public keys

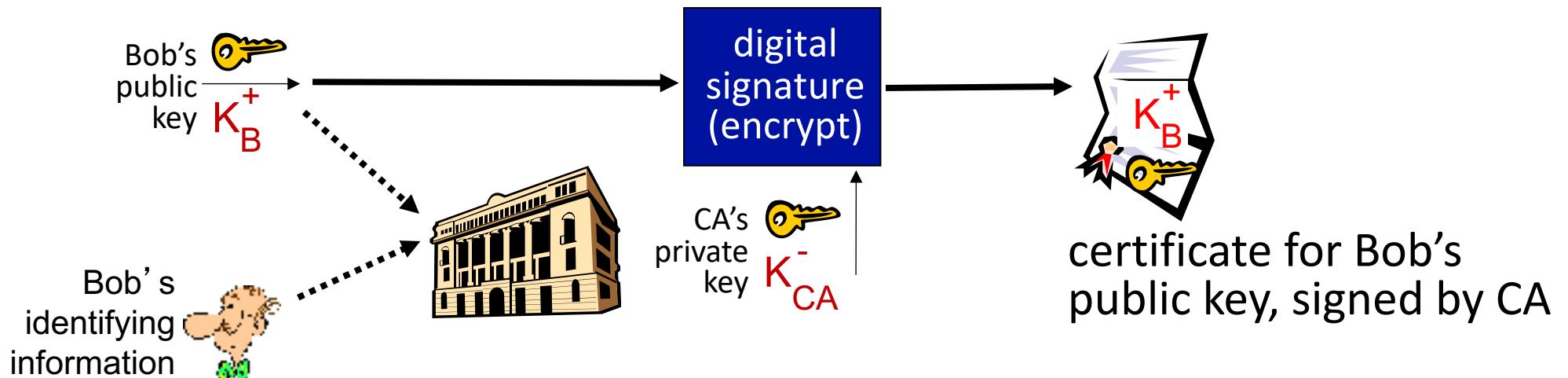
- motivation: Trudy plays pizza prank on Bob

- Trudy creates e-mail order:  
*Dear Pizza Store, Please deliver to me four pepperoni pizzas. Thank you, Bob*
- Trudy signs order with her private key
- Trudy sends order to Pizza Store
- Trudy sends to Pizza Store her public key, but says it's Bob's public key
- Pizza Store verifies signature; then delivers four pepperoni pizzas to Bob
- Bob doesn't even like pepperoni



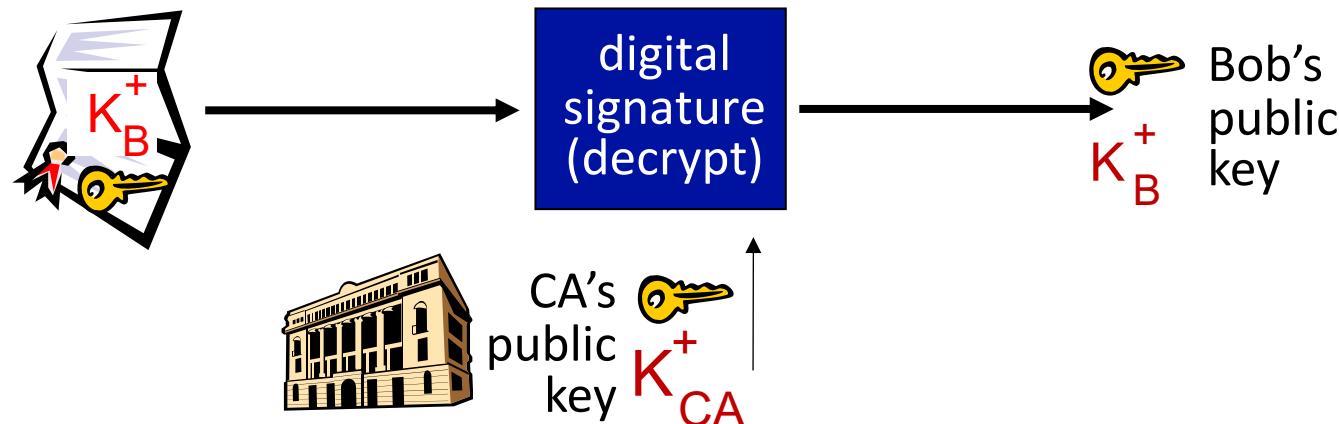
# Public key Certification Authorities (CA)

- certification authority (CA): binds public key to particular entity, E
- entity (person, website, router) registers its public key with CE provides “proof of identity” to CA
  - CA creates certificate binding identity E to E’s public key
  - certificate containing E’s public key digitally signed by CA: CA says “this is E’s public key”



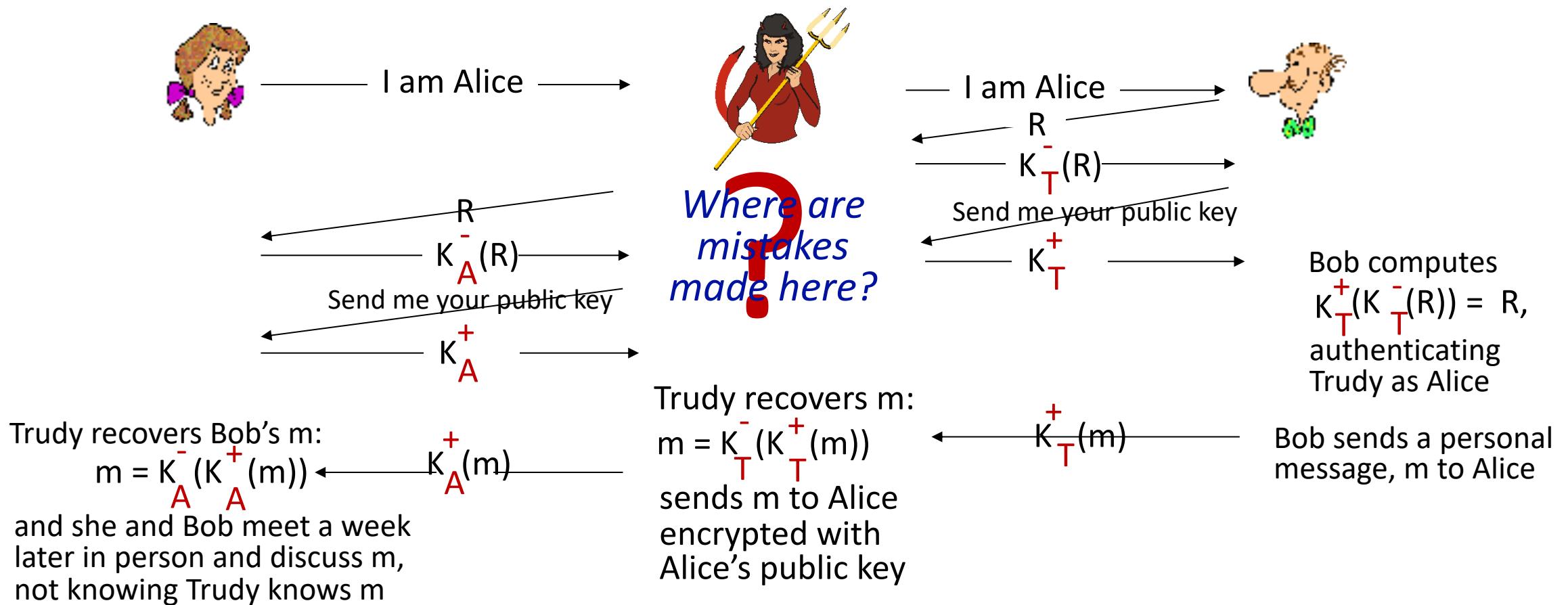
# Public key Certification Authorities (CA)

- when Alice wants Bob's public key:
  - gets Bob's certificate (Bob or elsewhere)
  - apply CA's public key to Bob's certificate, get Bob's public key



# Authentication: ap5.0 – let's fix it!!

Recall the problem: Trudy poses as Alice (to Bob) and as Bob (to Alice)



# Message Authentication Code

- How do you ensure data has not been changed?
- Alice send Bob  $(m, H(m))$
- Bob receives  $(m, H(m))$  and validates  $m$ .
- What is wrong with this?

# Message Authentication Code

- Hash needs to include authentication code
- $s$  = authentication code
- Instead of sending  $(m, H(m))$ , send  $(m, H(m + s))$
- This is called a Message Authentication Code (MAC)

# Chapter 8 outline

- What is network security?
- Principles of cryptography
- Authentication, message integrity
- Securing e-mail
- **Securing TCP connections: TLS**
- Network layer security: IPsec
- Security in wireless and mobile networks
- Operational security: firewalls and IDS



# Transport-layer security (TLS)

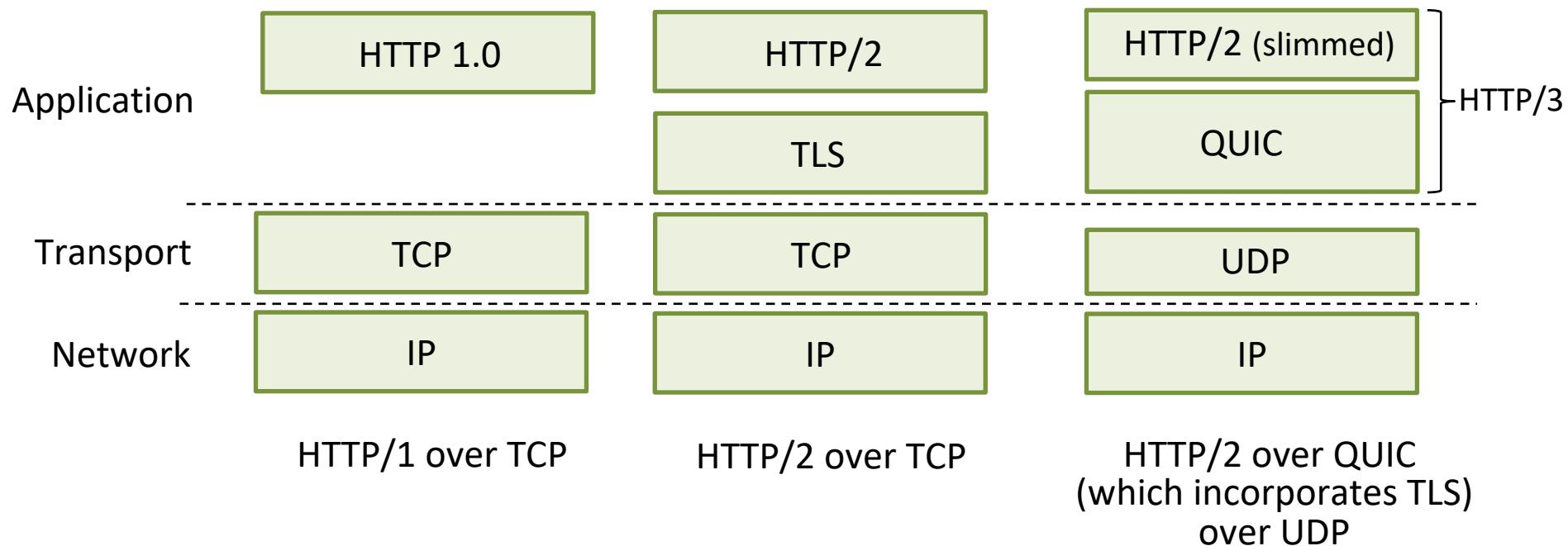
- widely deployed security protocol above the transport layer
    - supported by almost all browsers, web servers: https (port 443)
  - provides:
    - **confidentiality**: via *symmetric encryption*
    - **integrity**: via *cryptographic hashing*
    - **authentication**: via *public key cryptography*
  - history:
    - early research, implementation: secure network programming, secure sockets
    - secure socket layer (SSL) deprecated [2015]
    - TLS 1.3: RFC 8846 [2018]
- 
- all techniques we have studied!*

# Transport-layer security: what's needed?

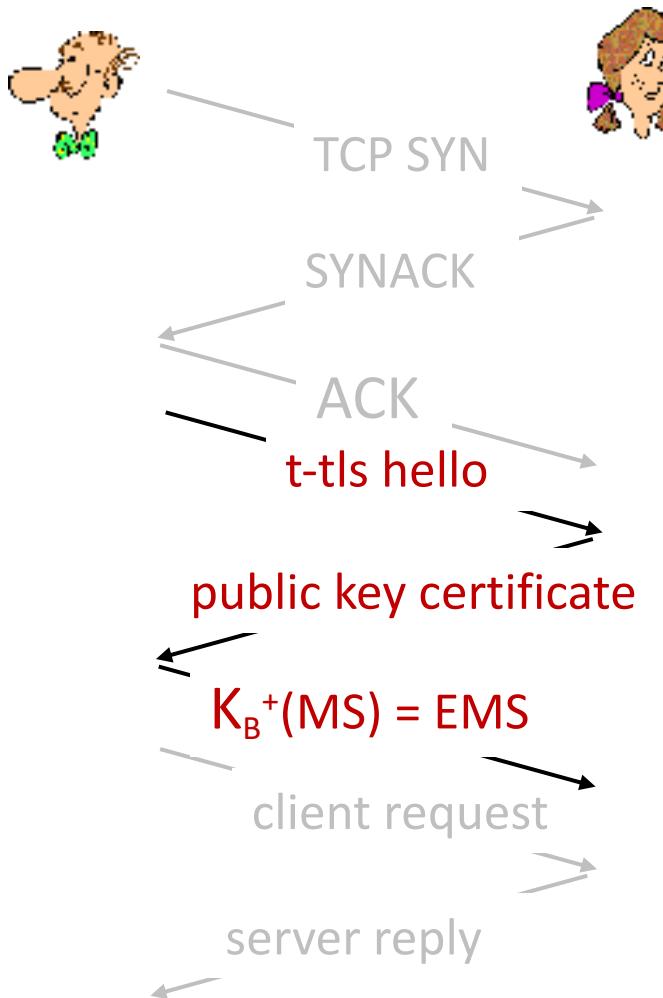
- let's *build* a toy TLS protocol, *t-tls*, to see what's needed!
- we've seen the “pieces” already:
  - **handshake**: Alice, Bob use their certificates, private keys to authenticate each other, exchange or create shared secret
  - **key derivation**: Alice, Bob use shared secret to derive set of keys
  - **data transfer**: stream data transfer: data as a series of records
    - not just one-time transactions
  - **connection closure**: special messages to securely close connection

# Transport-layer security (TLS)

- TLS provides an API that *any* application can use
- an HTTP view of TLS:



# t-tls: initial handshake



## t-tls handshake phase:

- Bob establishes TCP connection with Alice
- Bob verifies that Alice is really Alice
- Bob sends Alice a master secret key (MS), used to generate all other keys for TLS session
- potential issues:
  - 3 RTT before client can start receiving data (including TCP handshake)

# t-tls: cryptographic keys

- considered bad to use same key for more than one cryptographic function
  - different keys for message authentication code (MAC) and encryption
- four keys:
  - 🔑  $K_c$  : encryption key for data sent from client to server
  - 🔑  $M_c$  : MAC key for data sent from client to server
  - 🔑  $K_s$  : encryption key for data sent from server to client
  - 🔑  $M_s$  : MAC key for data sent from server to client
- keys derived from key derivation function (KDF)
  - takes master secret and (possibly) some additional random data to create new keys

# t-tls: encrypting data

- recall: TCP provides data *byte stream* abstraction
- Q: can we encrypt data in-stream as written into TCP socket?
  - A: where would MAC go? If at end, no message integrity until all data received and connection closed!
  - solution: break stream in series of “records”
    - each client-to-server record carries a MAC, created using  $M_c$
    - receiver can act on each record as it arrives
- t-tls record encrypted using symmetric key,  $K_c$ , passed to TCP:

$K_c($    $)$

# t-tls: encrypting data (more)

- possible attacks on data stream?
  - *Re-ordering*: man-in middle intercepts TCP segments and reorders (manipulating sequence #s in unencrypted TCP header)
  - Solution: use TLS sequence numbers (data, TLS-seq-# incorporated into MAC)
- *Replay*
- Solution: use nonce

# t-tls: connection close

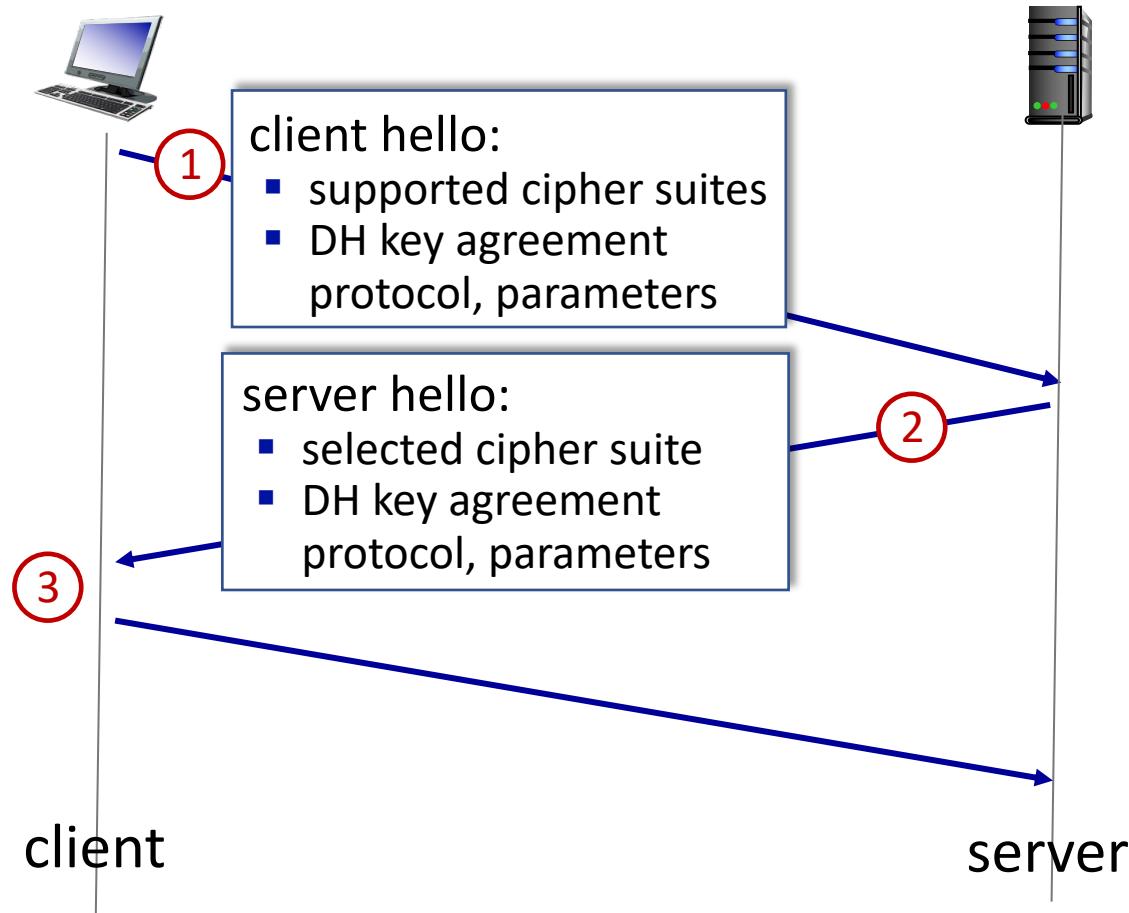
- truncation attack:
  - attacker forges TCP connection close segment
  - one or both sides thinks there is less data than there actually is
- solution: record types, with one type for closure
  - type 0 for data; type 1 for close
- MAC now computed using data, type, sequence #

$$K_C( \begin{array}{|c|c|c|c|} \hline length & type & data & MAC \\ \hline \end{array} )$$

# TLS: 1.3 cipher suite

- “cipher suite”: algorithms that can be used for key generation, encryption, MAC, digital signature
- TLS: 1.3 (2018): more limited cipher suite choice than TLS 1.2 (2008)
  - only 5 choices, rather than 37 choices
  - *requires* Diffie-Hellman (DH) for key exchange, rather than DH or RSA
  - combined encryption and authentication algorithm (“authenticated encryption”) for data rather than serial encryption, authentication
    - 4 based on AES
  - HMAC uses SHA (256 or 284) cryptographic hash function

# TLS 1.3 handshake: 1 RTT



- ① client TLS hello msg:
  - indicates cipher suites it supports
- ② server TLS hello msg chooses
  - key agreement protocol, parameters
  - cipher suite
  - server-signed certificate
- ③ client:
  - checks server certificate
  - generates key
  - can now make application request (e.g., HTTPS GET)

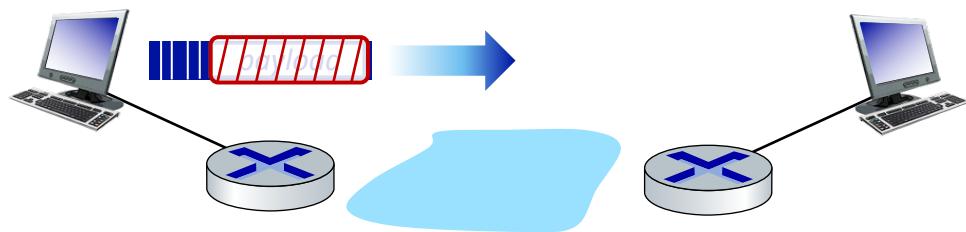
# Chapter 8 outline

- What is network security?
- Principles of cryptography
- Authentication, message integrity
- Securing e-mail
- Securing TCP connections: TLS
- **Network layer security: IPsec**
- Security in wireless and mobile networks
- Operational security: firewalls and IDS



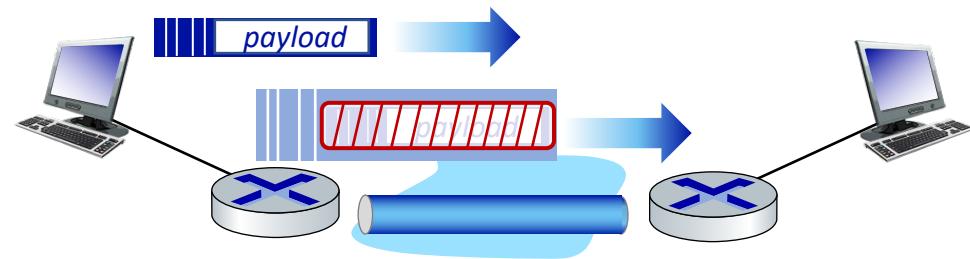
# IP Sec

- provides datagram-level encryption, authentication, integrity
  - for both user traffic and control traffic (e.g., BGP, DNS messages)
- two “modes”:



**transport mode:**

- *only* datagram *payload* is encrypted, authenticated



**tunnel mode:**

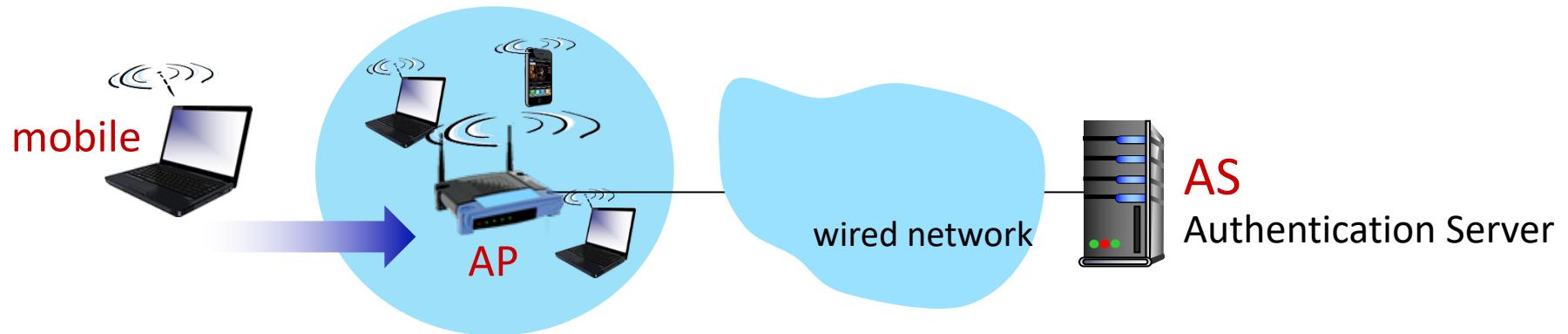
- entire datagram is encrypted, authenticated
- encrypted datagram encapsulated in new datagram with new IP header, tunneled to destination

# Chapter 8 outline

- What is network security?
- Principles of cryptography
- Authentication, message integrity
- Securing e-mail
- Securing TCP connections: TLS
- Network layer security: IPsec
- **Security in wireless and mobile networks**
  - 802.11 (WiFi)
  - 4G/5G
- Operational security: firewalls and IDS



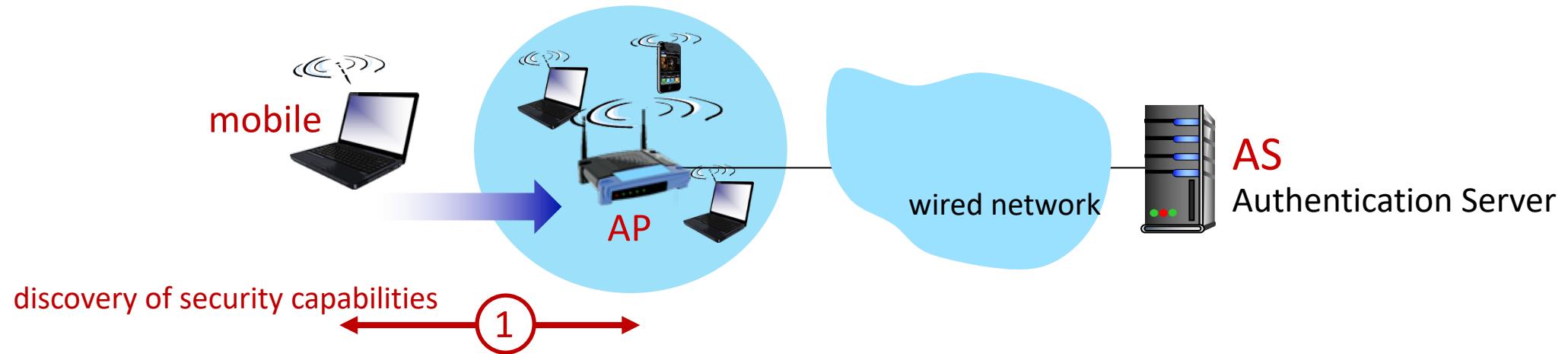
# 802.11: authentication, encryption



Arriving mobile must:

- associate with access point: (establish) communication over wireless link
- authenticate to network

# 802.11: authentication, encryption

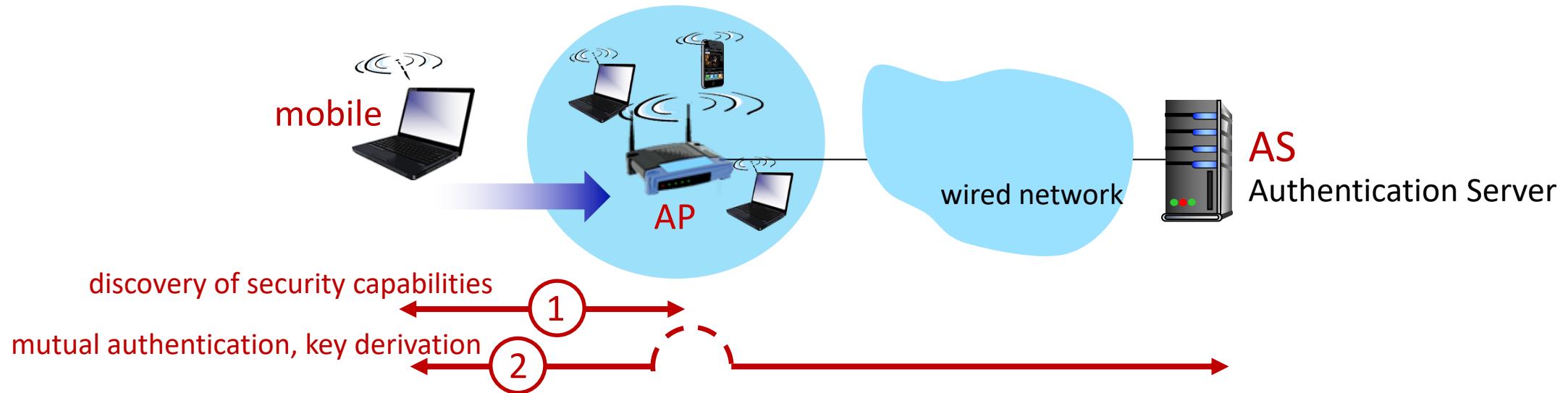


## ① discovery of security capabilities:

- AP advertises its presence, forms of authentication and encryption provided
- device requests specific forms authentication, encryption desired

although device, AP already exchanging messages, device not yet authenticated, does not have encryption keys

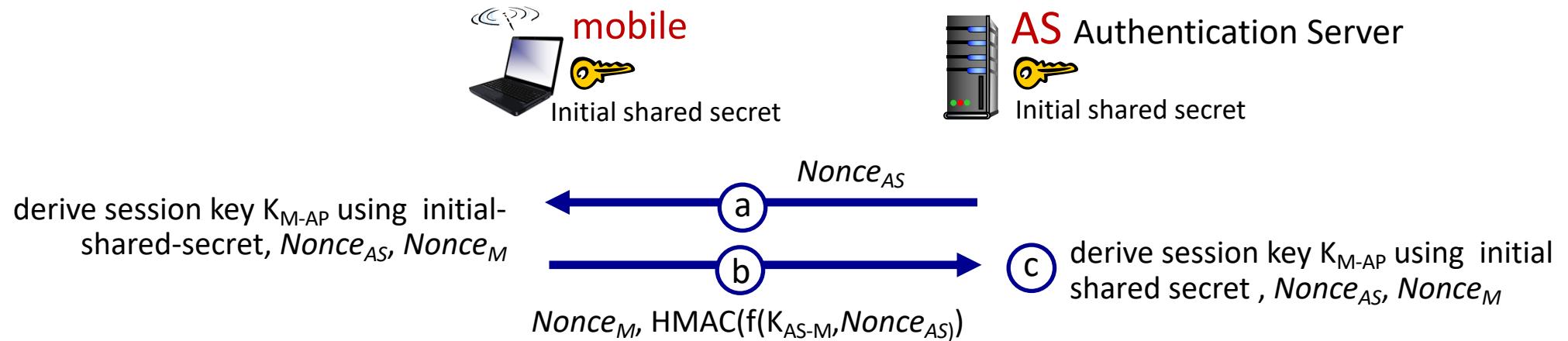
# 802.11: authentication, encryption



## ② mutual authentication and shared symmetric key derivation:

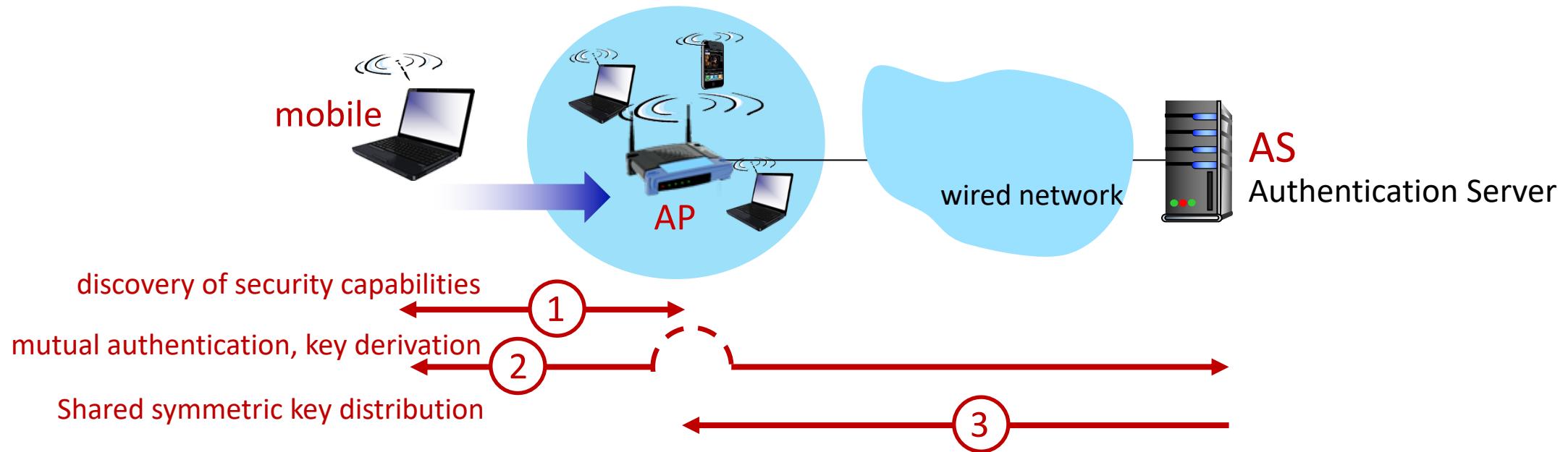
- AS, mobile already have shared common secret (e.g., password)
- AS, mobile use shared secret, nonces (prevent relay attacks), cryptographic hashing (ensure message integrity) to authenticating each other
- AS, mobile derive symmetric session key

# 802.11: WPA3 handshake



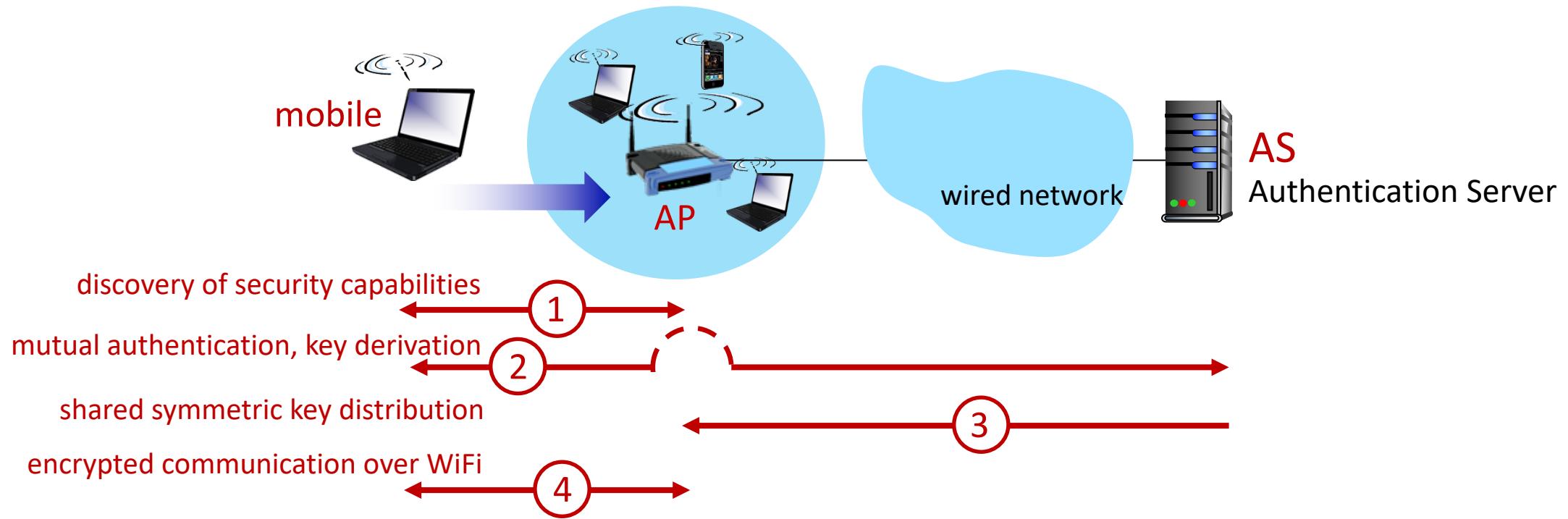
- AS generates  $\text{Nonce}_{AS}$ , sends to mobile
- mobile receives  $\text{Nonce}_{AS}$ 
  - generates  $\text{Nonce}_M$
  - generates symmetric shared session key  $K_{M-AP}$  using  $\text{Nonce}_{AS}$ ,  $\text{Nonce}_M$ , and initial shared secret
  - sends  $\text{Nonce}_M$ , and HMAC-signed value using  $\text{Nonce}_{AS}$  and initial shared secret
- AS derives symmetric shared session key  $K_{M-AP}$

# 802.11: authentication, encryption



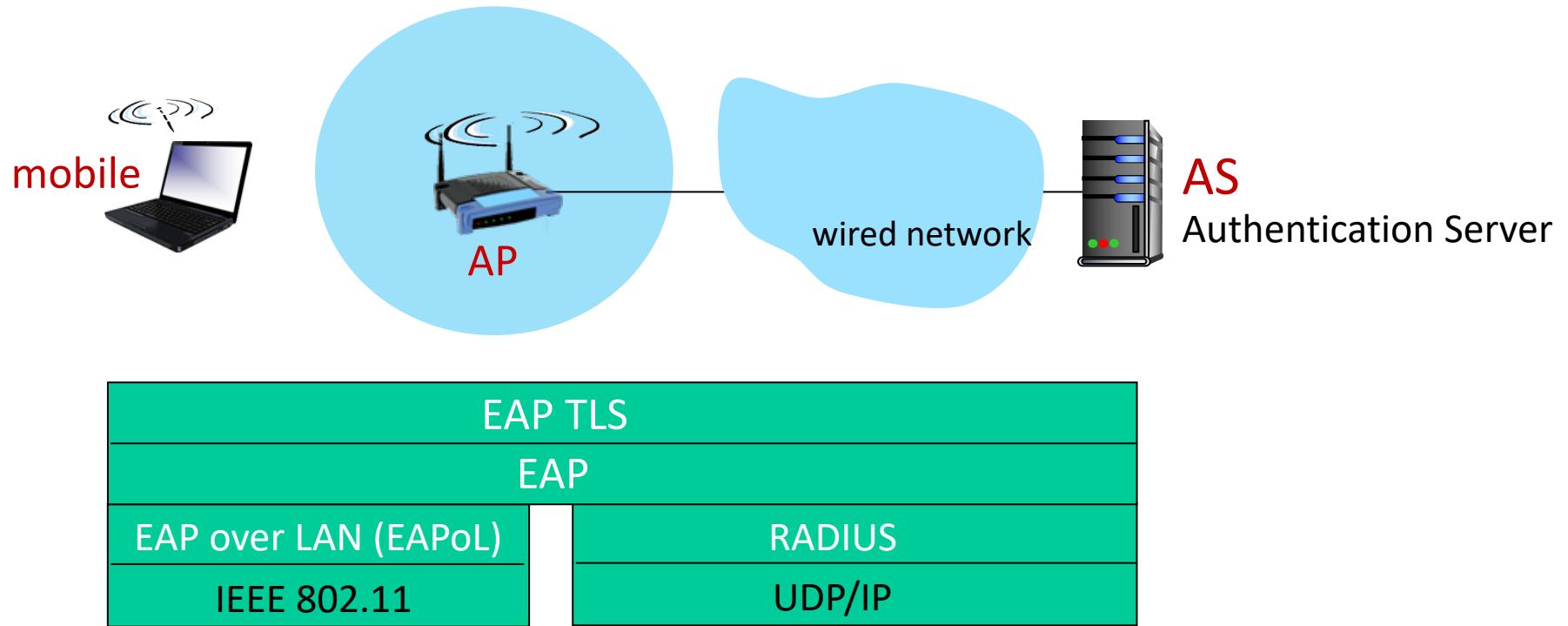
- ③ shared symmetric session key distribution (e.g., for AES encryption)
- same key derived at mobile, AS
  - AS informs AP of the shared symmetric session

# 802.11: authentication, encryption



- ④ encrypted communication between mobile and remote host via AP
- same key derived at mobile, AS
  - AS informs AP of the shared symmetric session

# 802.11: authentication, encryption



- Extensible Authentication Protocol (EAP) [RFC 3748] defines end-to-end request/response protocol between mobile device, AS

# Chapter 8 outline

- What is network security?
- Principles of cryptography
- Authentication, message integrity
- Securing e-mail
- Securing TCP connections: TLS
- Network layer security: IPsec
- **Security in wireless and mobile networks**
  - 802.11 (WiFi)
  - 4G/5G
- Operational security: firewalls and IDS



# Authentication, encryption in 4G LTE



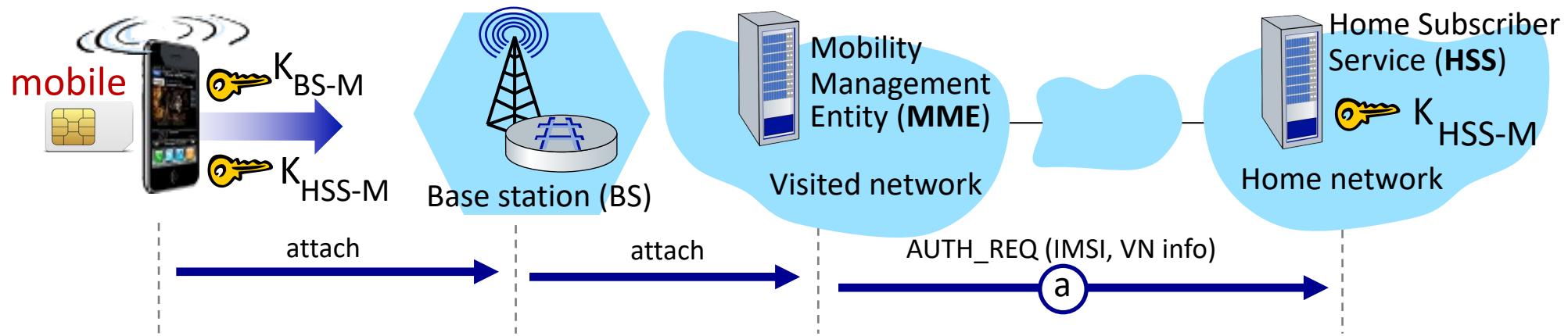
- arriving mobile must:
  - associate with BS: (establish) communication over 4G wireless link
  - authenticate itself to network, and authenticate network
- notable differences from WiFi
  - mobile's SIMcard provides global identity, contains shared keys
  - services in visited network depend on (paid) service subscription in home network

# Authentication, encryption in 4G LTE



- mobile, BS use derived session key  $K_{BS-M}$  to encrypt communications over 4G link
- MME in visited network + HSS in home network, together play role of WiFi AS
  - ultimate authenticator is HSS
  - trust and business relationship between visited and home networks

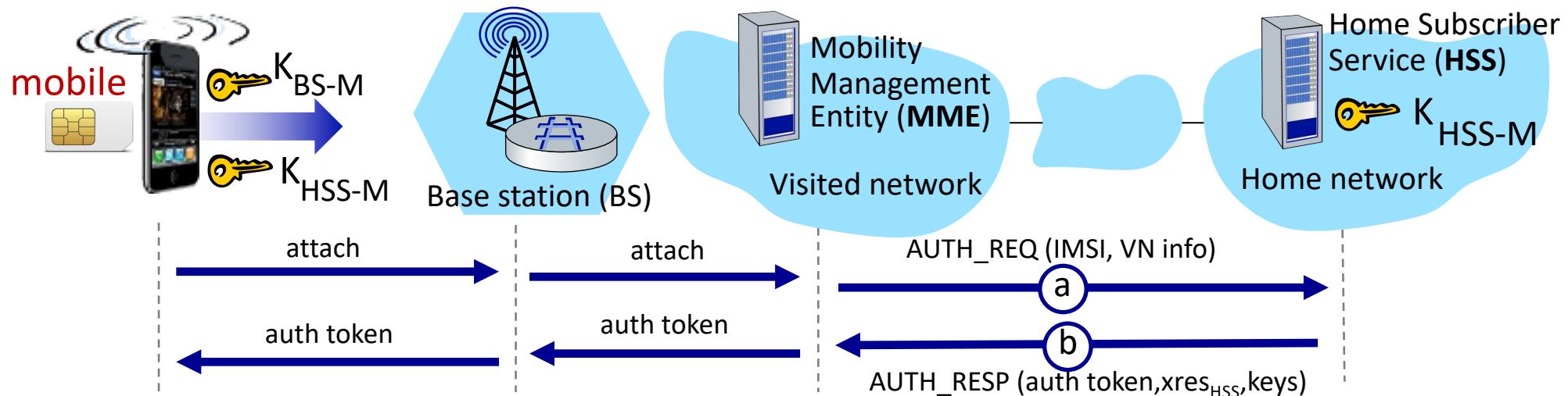
# Authentication, encryption in 4G LTE



## a) authentication request to home network HSS

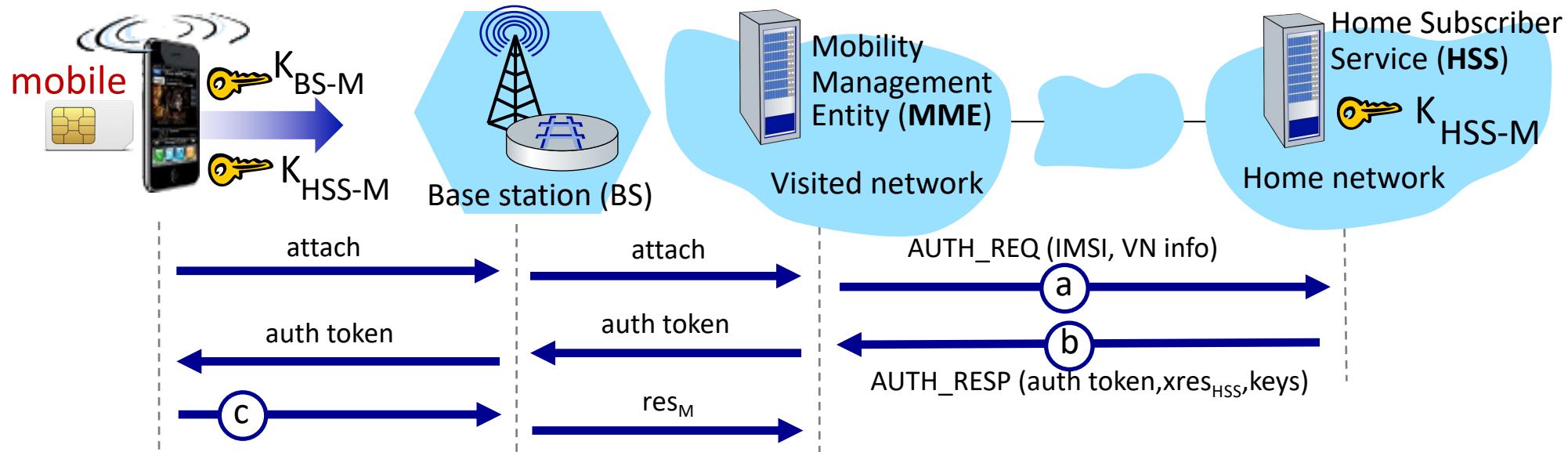
- mobile sends attach message (containing its IMSI, visited network info) relayed from BS to visited MME to home HSS
- IMSI identifies mobile's home network

# Authentication, encryption in 4G LTE



- ③ HSS use shared-in-advance secret key,  $K_{HSS-M}$ , to derive authentication token,  $auth\_token$ , and expected authentication response token,  $xres_{HSS}$
- $auth\_token$  contains info encrypted by HSS using  $K_{HSS-M}$ , allowing mobile to know that whoever computed  $auth\_token$  knows shared-in-advance secret
  - mobile has authenticated network
  - visited HSS keeps  $xres_{HSS}$  for later use

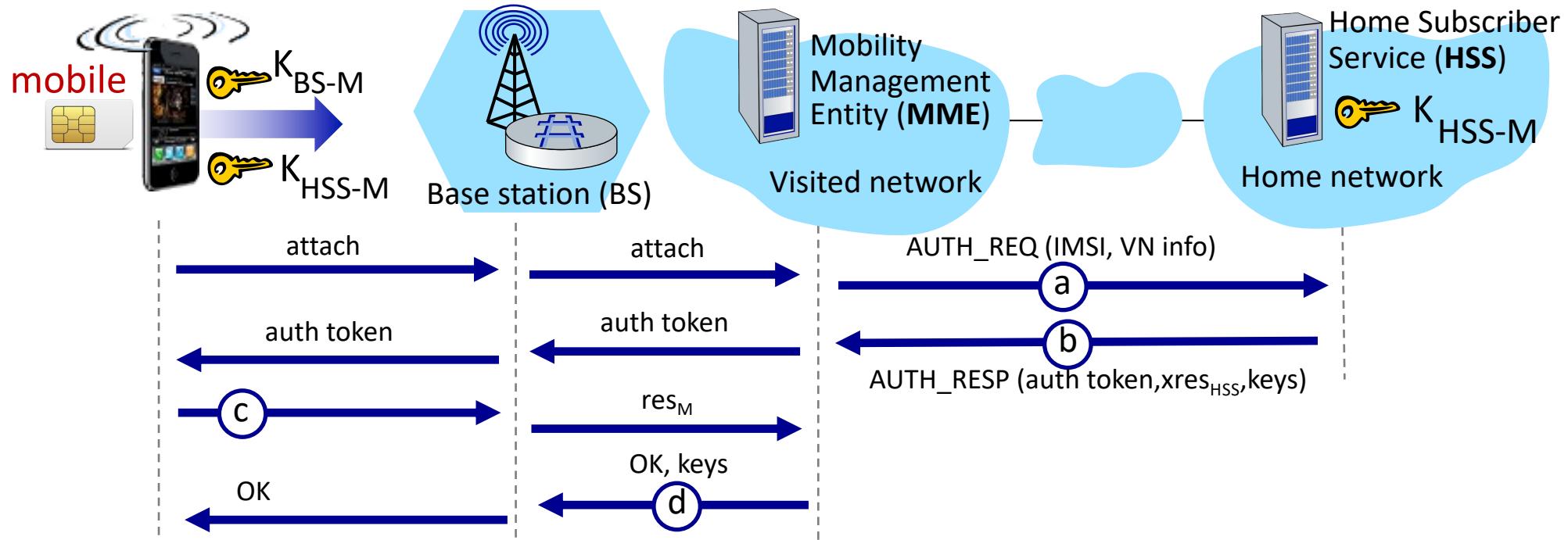
# Authentication, encryption in 4G LTE



## c) authentication response from mobile:

- mobile computes  $res_M$  using its secret key to make same cryptographic calculation that HSS made to compute  $xres_{HSS}$  and sends  $res_M$  to MME

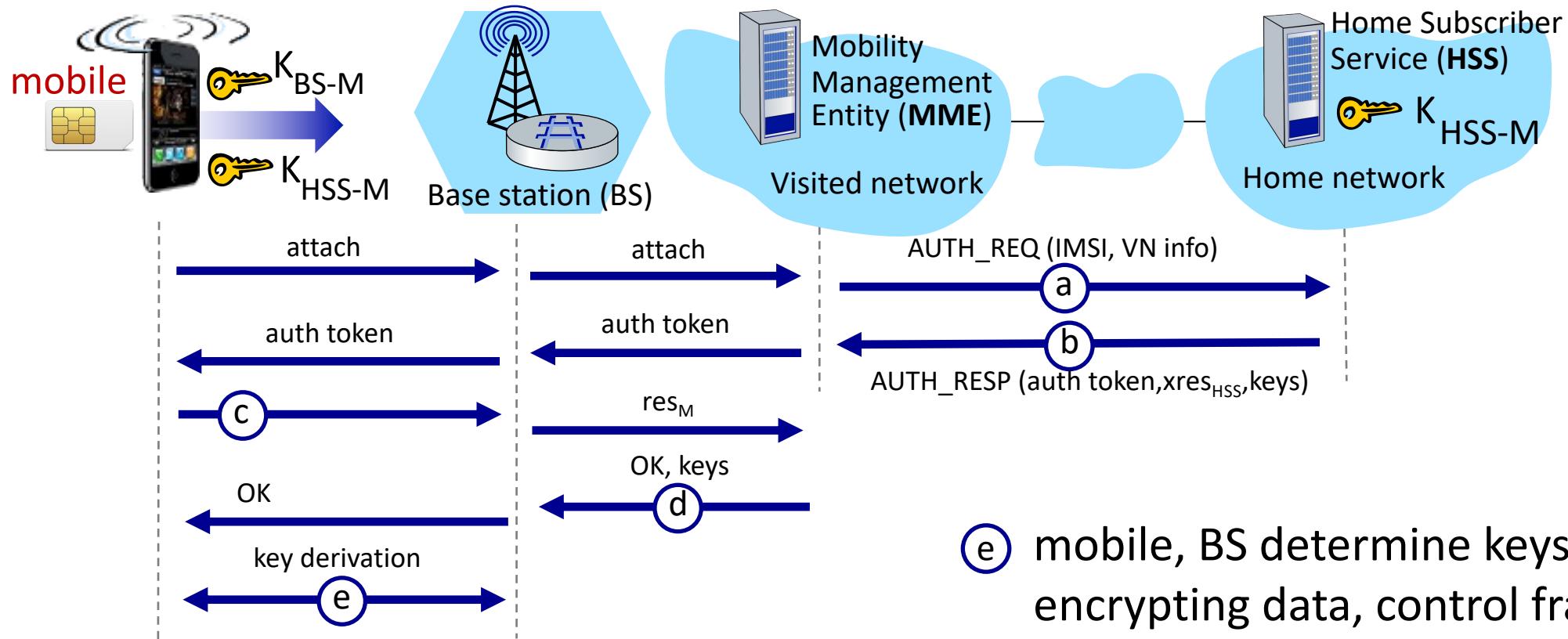
# Authentication, encryption in 4G LTE



d) mobile is authenticated by network:

- MMS compares mobile-computed value of  $res_M$  with the HSS-computed value of  $xres_{HSS}$ . If they match, mobile is authenticated ! (why?)
- MMS informs BS that mobile is authenticated, generates keys for BS

# Authentication, encryption in 4G LTE



- e) mobile, BS determine keys for encrypting data, control frames over 4G wireless channel
  - AES can be used

# Authentication, encryption: from 4G to 5G

- **4G:** MME in visited network makes authentication decision
- **5G:** home network provides authentication decision
  - visited MME plays “middleman” role but can still reject
- **4G:** uses shared-in-advance keys
- **5G:** keys not shared in advance for IoT
- **4G:** device IMSI transmitted in cleartext to BS
- **5G:** public key crypto used to encrypt IMSI

# Chapter 8 outline

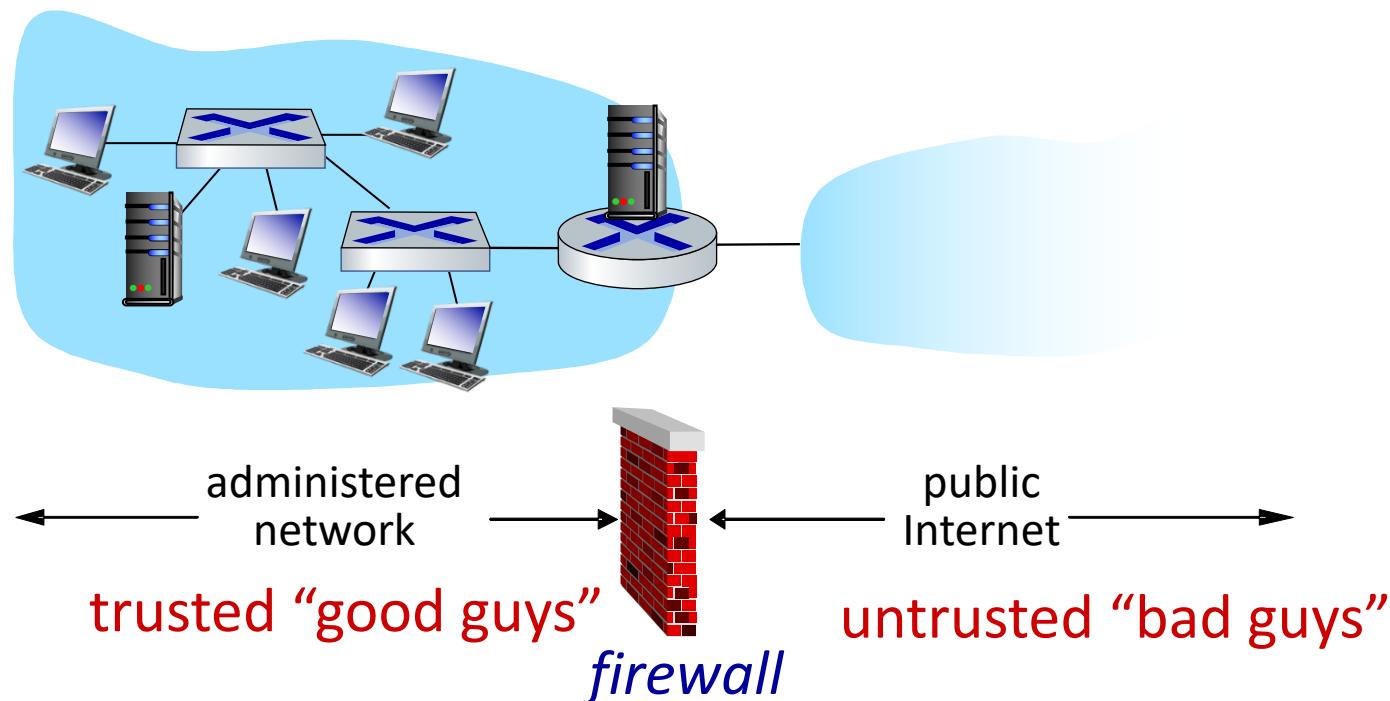
- What is network security?
- Principles of cryptography
- Authentication, message integrity
- Securing e-mail
- Securing TCP connections: TLS
- Network layer security: IPsec
- Security in wireless and mobile networks
- **Operational security: firewalls and IDS**



# Firewalls

**firewall**

isolates organization's internal network from larger Internet, allowing some packets to pass, blocking others



# Firewalls: why

prevent denial of service attacks:

- SYN flooding: attacker establishes many bogus TCP connections, no resources left for “real” connections

prevent illegal modification/access of internal data

- e.g., attacker replaces CIA’s homepage with something else

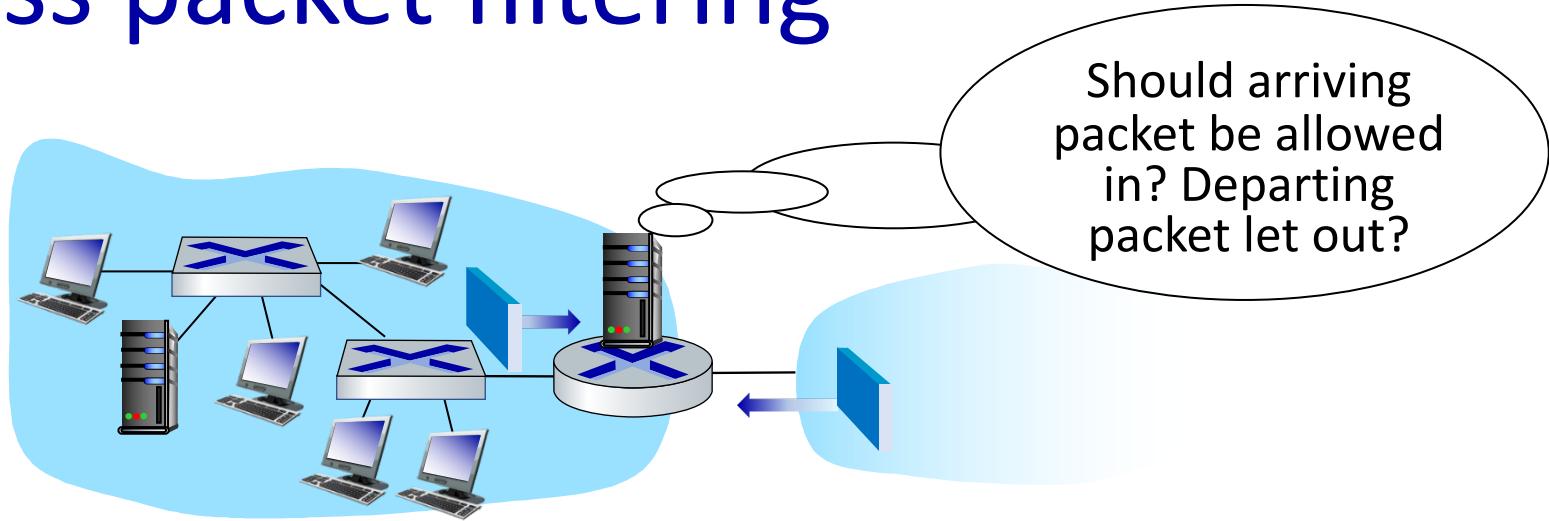
allow only authorized access to inside network

- set of authenticated users/hosts

three types of firewalls:

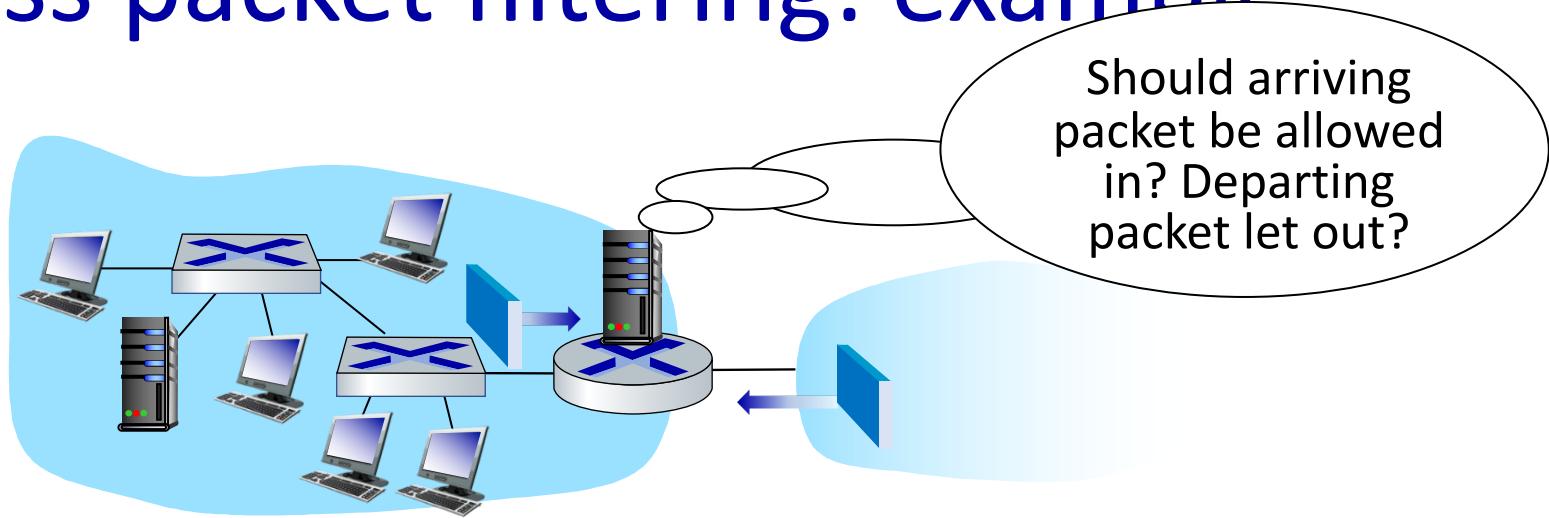
- stateless packet filters
- stateful packet filters
- application gateways

# Stateless packet filtering



- internal network connected to Internet via router **firewall**
- filters **packet-by-packet**, decision to forward/drop packet based on:
  - source IP address, destination IP address
  - TCP/UDP source, destination port numbers
  - ICMP message type
  - TCP SYN, ACK bits

# Stateless packet filtering: example



- **example 1:** block incoming and outgoing datagrams with IP protocol field = 17 and with either source or dest port = 23
  - **result:** all incoming, outgoing UDP flows and telnet connections are blocked
- **example 2:** block inbound TCP segments with ACK=0
  - **result:** prevents external clients from making TCP connections with internal clients, but allows internal clients to connect to outside

# Stateless packet filtering: more examples

Policy	Firewall Setting
no outside Web access	drop all outgoing packets to any IP address, port 80
no incoming TCP connections, except those for institution's public Web server only.	drop all incoming TCP SYN packets to any IP except 130.207.244.203, port 80
prevent Web-radios from eating up the available bandwidth.	drop all incoming UDP packets - except DNS and router broadcasts.
prevent your network from being used for a smurf DoS attack.	drop all ICMP packets going to a "broadcast" address (e.g. 130.207.255.255)
prevent your network from being tracerouted	drop all outgoing ICMP TTL expired traffic

# Access Control Lists

**ACL:** table of rules, applied top to bottom to incoming packets: (action, condition) pairs: looks like OpenFlow forwarding (Ch. 4)!

action	source address	dest address	protocol	source port	dest port	flag bit
allow	222.22/16	outside of 222.22/16	TCP	> 1023	80	any
allow	outside of 222.22/16	222.22/16	TCP	80	> 1023	ACK
allow	222.22/16	outside of 222.22/16	UDP	> 1023	53	---
allow	outside of 222.22/16	222.22/16	UDP	53	> 1023	----
deny	all	all	all	all	all	all

# Stateful packet filtering

- *stateless packet filter*: heavy handed tool

- admits packets that “make no sense,” e.g., dest port = 80, ACK bit set, even though no TCP connection established:

action	source address	dest address	protocol	source port	dest port	flag bit
allow	outside of 222.22/16	222.22/16	TCP	80	> 1023	ACK

- *stateful packet filter*: track status of every TCP connection

- track connection setup (SYN), teardown (FIN): determine whether incoming, outgoing packets “makes sense”
  - timeout inactive connections at firewall: no longer admit packets

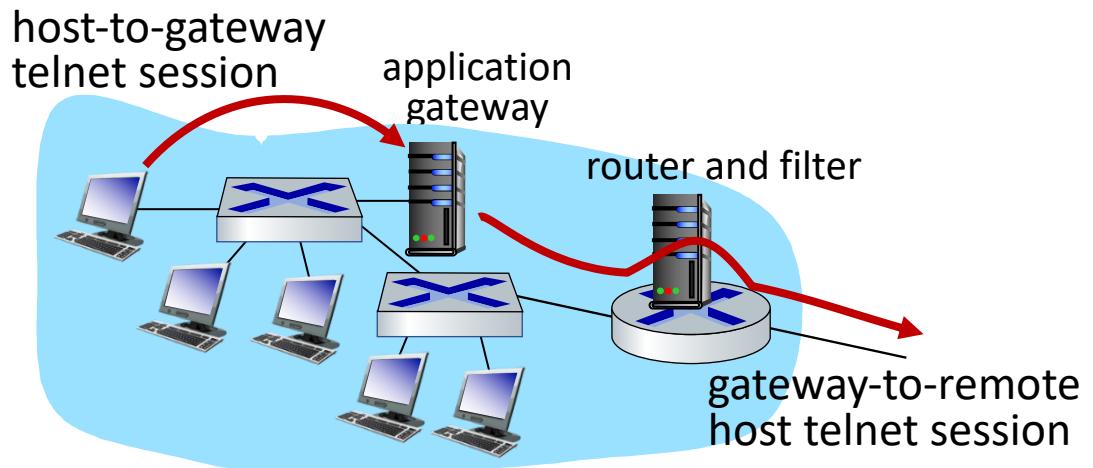
# Stateful packet filtering

ACL augmented to indicate need to check connection state table before admitting packet

action	source address	dest address	proto	source port	dest port	flag bit	check connection
allow	222.22/16	outside of 222.22/16	TCP	> 1023	80	any	
allow	outside of 222.22/16	222.22/16	TCP	80	> 1023	ACK	X
allow	222.22/16	outside of 222.22/16	UDP	> 1023	53	---	
allow	outside of 222.22/16	222.22/16	UDP	53	> 1023	----	X
deny	all	all	all	all	all	all	

# Application gateways

- filter packets on application data as well as on IP/TCP/UDP fields.
- *example:* allow select internal users to telnet outside



1. require all telnet users to telnet through gateway.
2. for authorized users, gateway sets up telnet connection to dest host
  - gateway relays data between 2 connections
3. router filter blocks all telnet connections not originating from gateway

# Limitations of firewalls, gateways

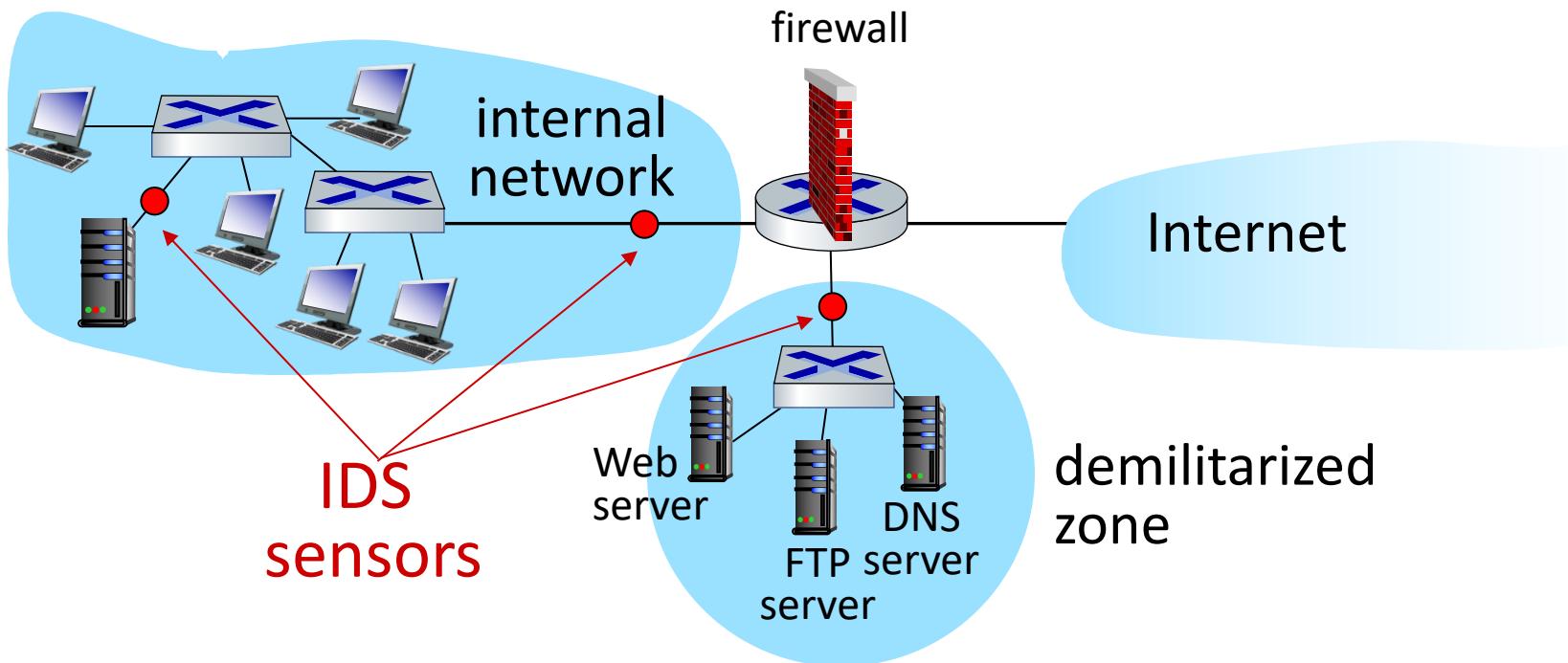
- **IP spoofing:** router can't know if data "really" comes from claimed source
- if multiple apps need special treatment, each has own app. gateway
- client software must know how to contact gateway
  - e.g., must set IP address of proxy in Web browser
- filters often use all or nothing policy for UDP
- ***tradeoff:*** degree of communication with outside world, level of security
- many highly protected sites still suffer from attacks

# Intrusion detection systems

- packet filtering:
  - operates on TCP/IP headers only
  - no correlation check among sessions
- IDS: intrusion detection system
  - deep packet inspection: look at packet contents (e.g., check character strings in packet against database of known virus, attack strings)
  - examine correlation among multiple packets
    - port scanning
    - network mapping
    - DoS attack

# Intrusion detection systems

multiple IDSs: different types of checking at different locations



# Network Security (summary)

basic techniques.....

- cryptography (symmetric and public key)
- message integrity
- end-point authentication

.... used in many different security scenarios

- secure email
- secure transport (TLS)
- IP sec
- 802.11, 4G/5G

operational security: firewalls and IDS



# **Concurrency**

# Concurrency

- Concurrency vs parallelism?
- Concurrency: one system doing multiple things. They might be done in parallel, they might not.
- Parallelism: doing two tasks at once.
- Before multicore systems, how did multiple things get done?
- “Concurrency is not Parallelism” by Rob Pike

# Concurrency and Networking

- Why do concurrency and networking often go together?
- The network is slow (compared to the computer)!
- What do you do while you wait?

# Concurrency

- Four major methods:
  - Processes
  - Threads (most common)
  - Micro-threads
  - Async (does not provide parallelism)

# Processes

- Pretty much the same as starting your program twice at the same time
- Whole program is copied
- Slow and heavy
- No shared memory between processes
  - Requires explicit sharing, such as a file or a socket

# Threads

- Lighter alternative to processes
  - 10x faster creation than processes
- Threads provided by OS
- Memory is shared between threads (good and bad)

# Threads

## Shared Memory

- How do you deal with shared memory?
- What if two threads are writing to one memory location at once?
- Atomic operations
- Synchronization techniques
  - Locks/Mutex
  - Semaphore
  - Message Queue

# Micro-threads

- Instead of OS controlled threads, programming language controlled threads
- Much lighter weight than normal threads
- Let the library deal with OS threads

# I/O Multiplexing

- While you are waiting for I/O, do something else
- Give operating system list of sockets (or files) you are waiting for
- OS will let you know when one is ready to be acted on
- select, poll, epoll

# Async

- One task runs at a time, but never blocks
- Tasks explicitly yield control to another task
- An event loop manages which tasks are ready to run
- Does not give you parallel execution
- You don't have to worry about the problems associated with parallel execution

# Python

- Python has strange behavior for concurrency
- Global Interpreter Lock
- Not a problem for these labs because you are waiting for the network

# Thread/Process Pool

- It is expensive to keep creating new threads/processes for every job
- Keep a list of threads/processes
- Give a task to an available thread/process
- If no threads/processes are available, block
- Once a thread/process is done, it is given another task

**Some programmers, when confronted with  
a problem, think “I know, I’ll solve it with  
threads!”. have Now problems. two they**