

Chapter I

Intro to Networks

© *The materials in these notes are adapted from Computer Networking: A Top Down Approach, 6th edition, by Jim Kurose, Keith Ross*

Chapter I: introduction

our goal:

- ❖ get “feel” and terminology
- ❖ more depth, detail
later in course
- ❖ approach:
 - use Internet as example

overview:

- ❖ what’s the Internet?
- ❖ what’s a protocol?
- ❖ network edge; hosts, access net, physical media
- ❖ network core: packet/circuit switching, Internet structure
- ❖ performance: loss, delay, throughput
- ❖ security
- ❖ protocol layers, service models
- ❖ history

Chapter I: roadmap

I.1 what *is* the Internet?

I.2 network edge

- end systems, access networks, links

I.3 network core

- packet switching, circuit switching, network structure

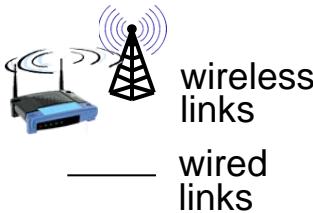
I.4 delay, loss, throughput in networks

I.5 protocol layers, service models

I.6 networks under attack: security

I.7 history

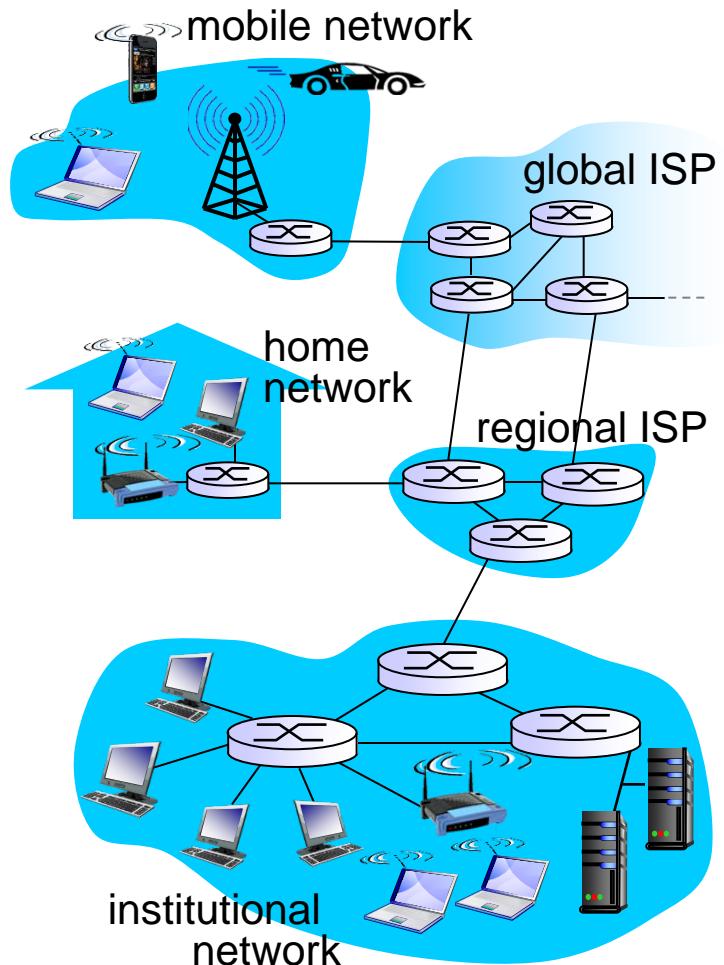
What's the Internet: “nuts and bolts” view



- ❖ millions of connected computing devices:
 - *hosts = end systems*
 - running *network apps*

- ❖ communication links
 - fiber, copper, radio, satellite
 - transmission rate: *bandwidth*

- ❖ *Packet switches: forward packets (chunks of data)*
 - *routers and switches*



“Fun” internet appliances



IP picture frame
<http://www.ceiva.com/>



Web-enabled toaster +
weather forecaster



Internet
refrigerator



Slingbox: watch,
control cable TV remotely



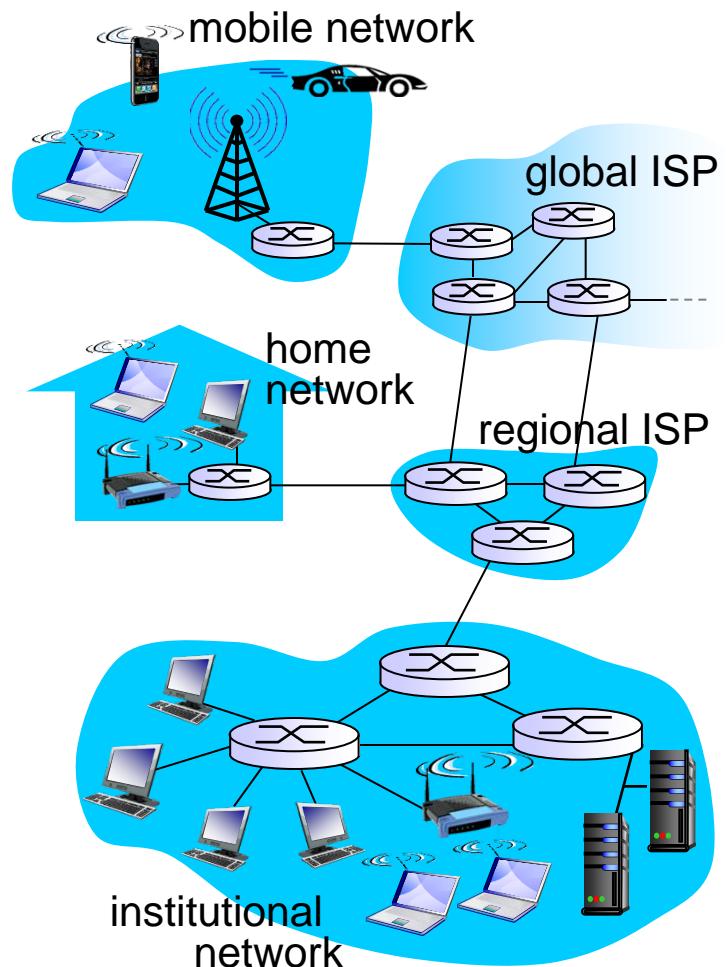
Tweet-a-watt:
monitor energy use



Internet phones

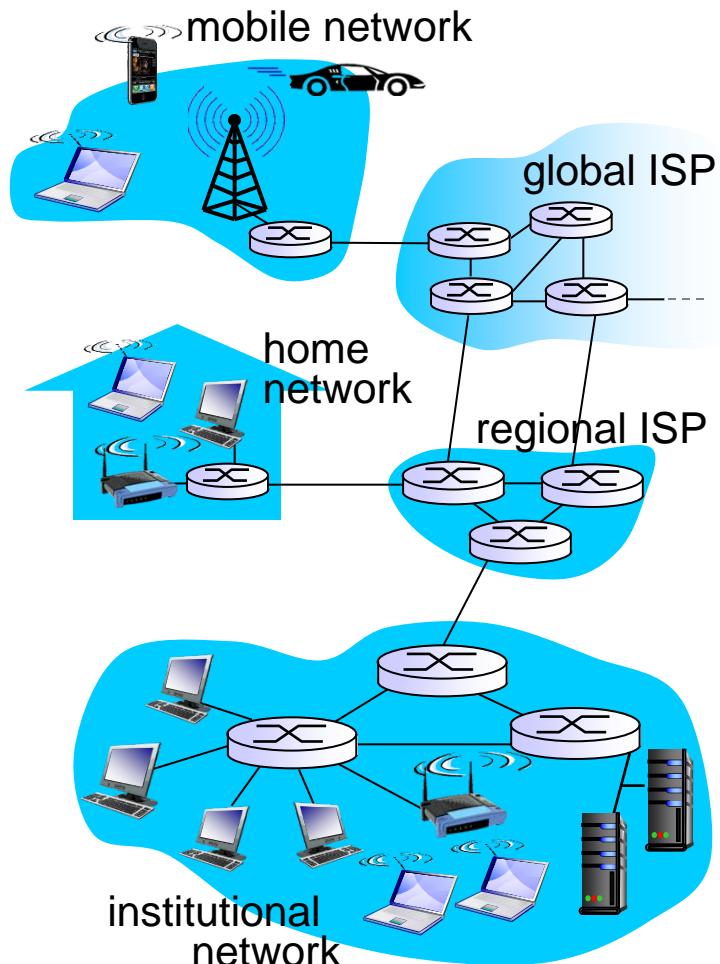
What's the Internet: “nuts and bolts” view

- ❖ *Internet: “network of networks”*
 - Interconnected ISPs
- ❖ *protocols* control sending, receiving of msgs
 - e.g., TCP, IP, HTTP, Skype, 802.11
- ❖ *Internet standards*
 - RFC: Request for comments
 - IETF: Internet Engineering Task Force



What's the Internet: a service view

- ❖ *Infrastructure that provides services to applications:*
 - Web, VoIP, email, games, e-commerce, social nets, ...
- ❖ *provides programming interface to apps*
 - hooks that allow sending and receiving app programs to “connect” to Internet
 - provides service options, analogous to postal service



What's a protocol?

human protocols:

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

... specific msgs sent

... specific actions taken
when msgs received, or
other events

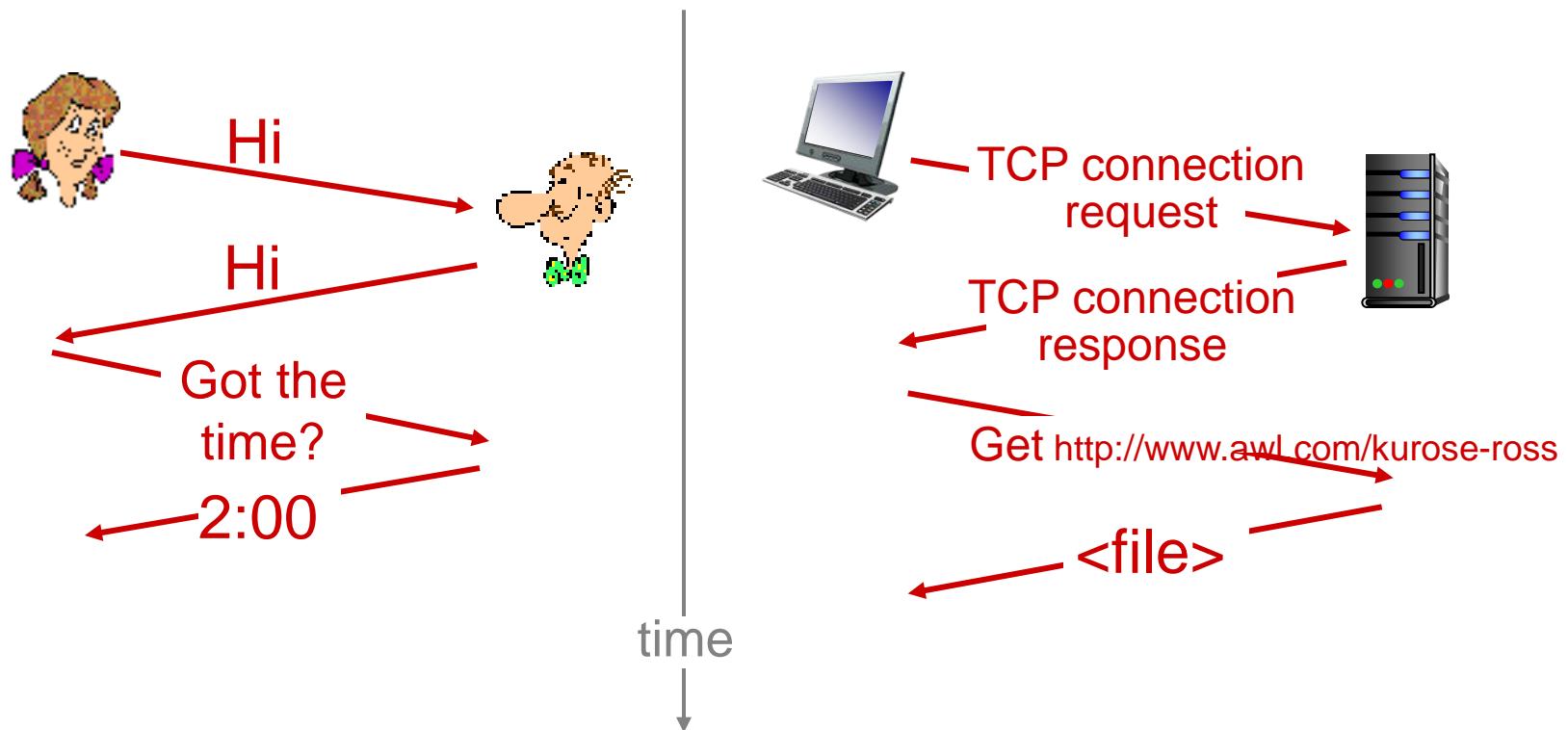
network protocols:

- ❖ machines rather than humans
- ❖ all communication activity in Internet governed by protocols

*protocols define format, order
of msgs 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 I: roadmap

I.1 what *is* the Internet?

I.2 network edge

- end systems, access networks, links

I.3 network core

- packet switching, circuit switching, network structure

I.4 delay, loss, throughput in networks

I.5 protocol layers, service models

I.6 networks under attack: security

I.7 history

A closer look at network structure:

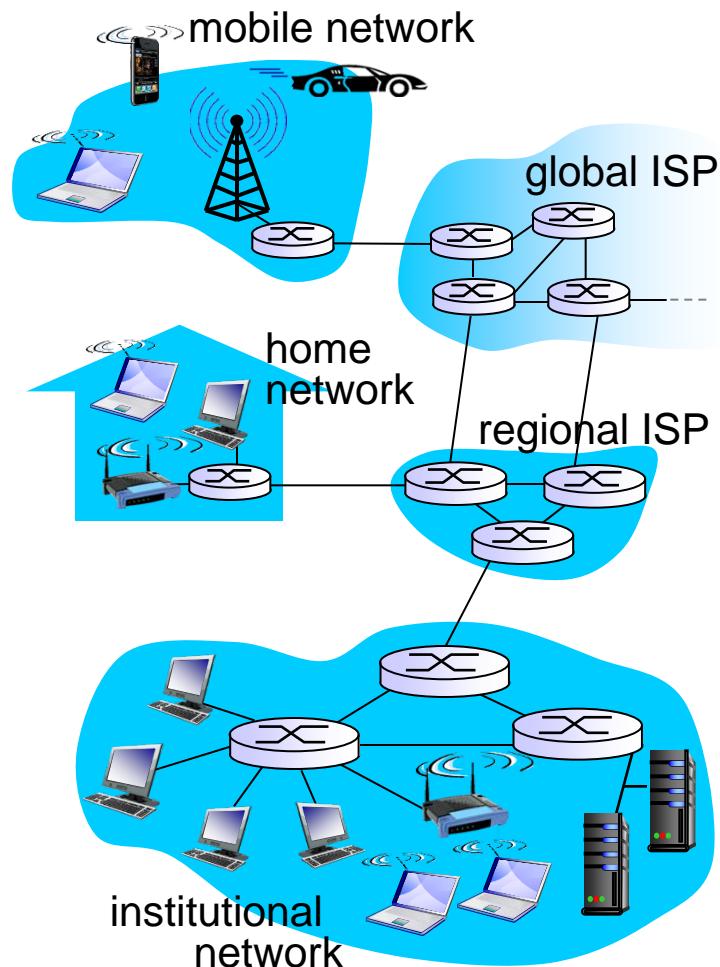
❖ *network edge:*

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

❖ *access networks, physical media:* wired, wireless communication links

❖ *network core:*

- interconnected routers
- network of networks



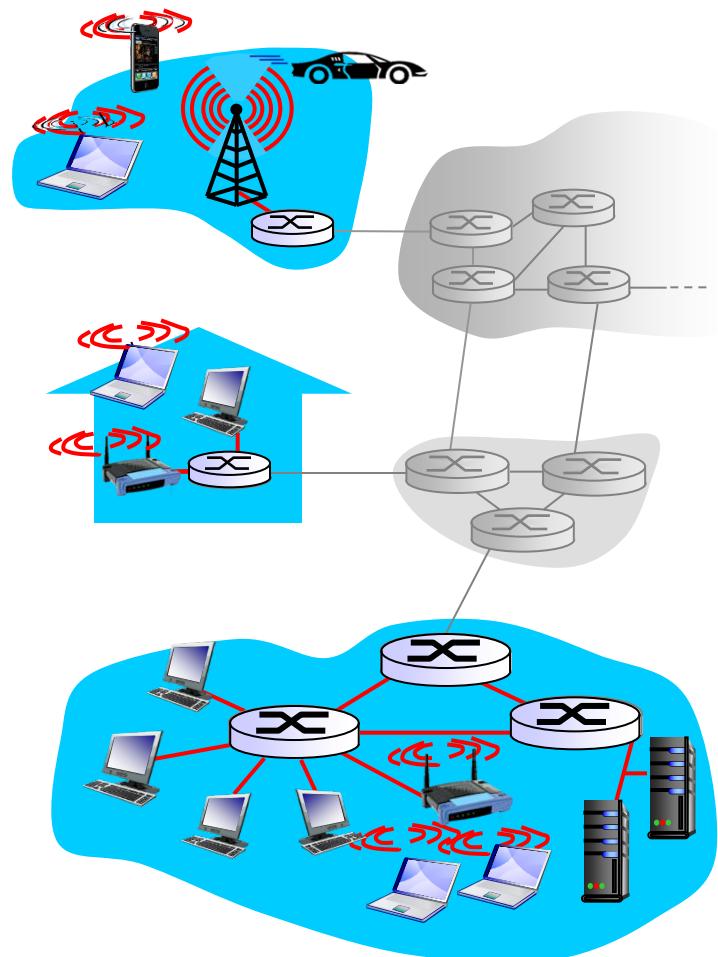
Access networks and physical media

Q: How to connect end systems to edge router?

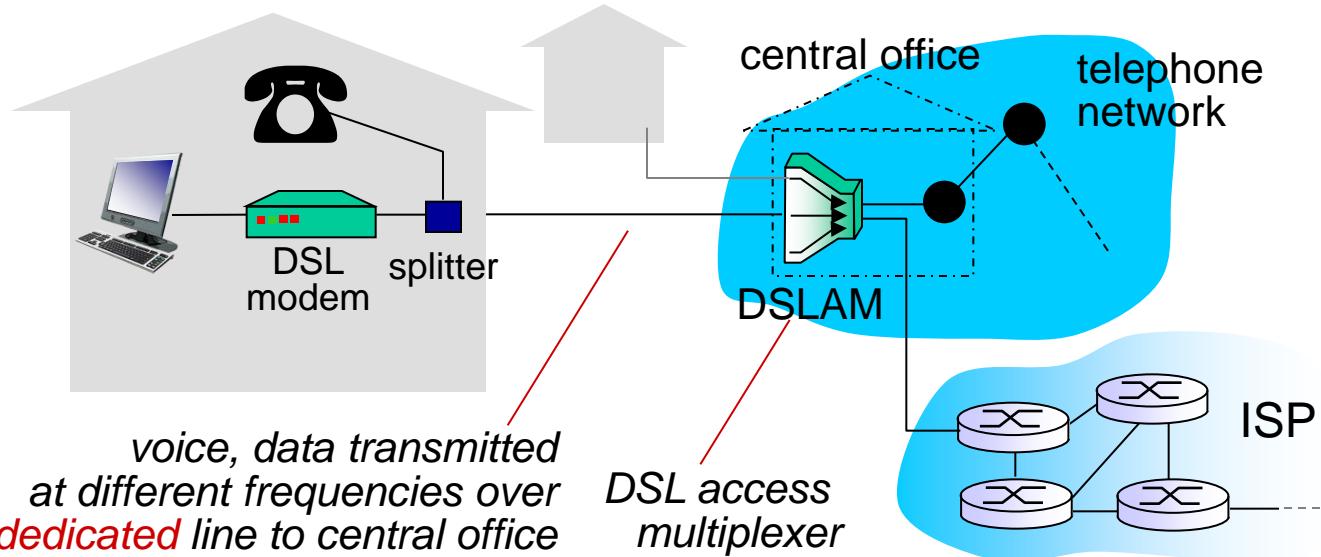
- ❖ residential access nets
- ❖ institutional access networks (school, company)
- ❖ mobile access networks

keep in mind:

- ❖ bandwidth (bits per second) of access network?
- ❖ shared or dedicated?

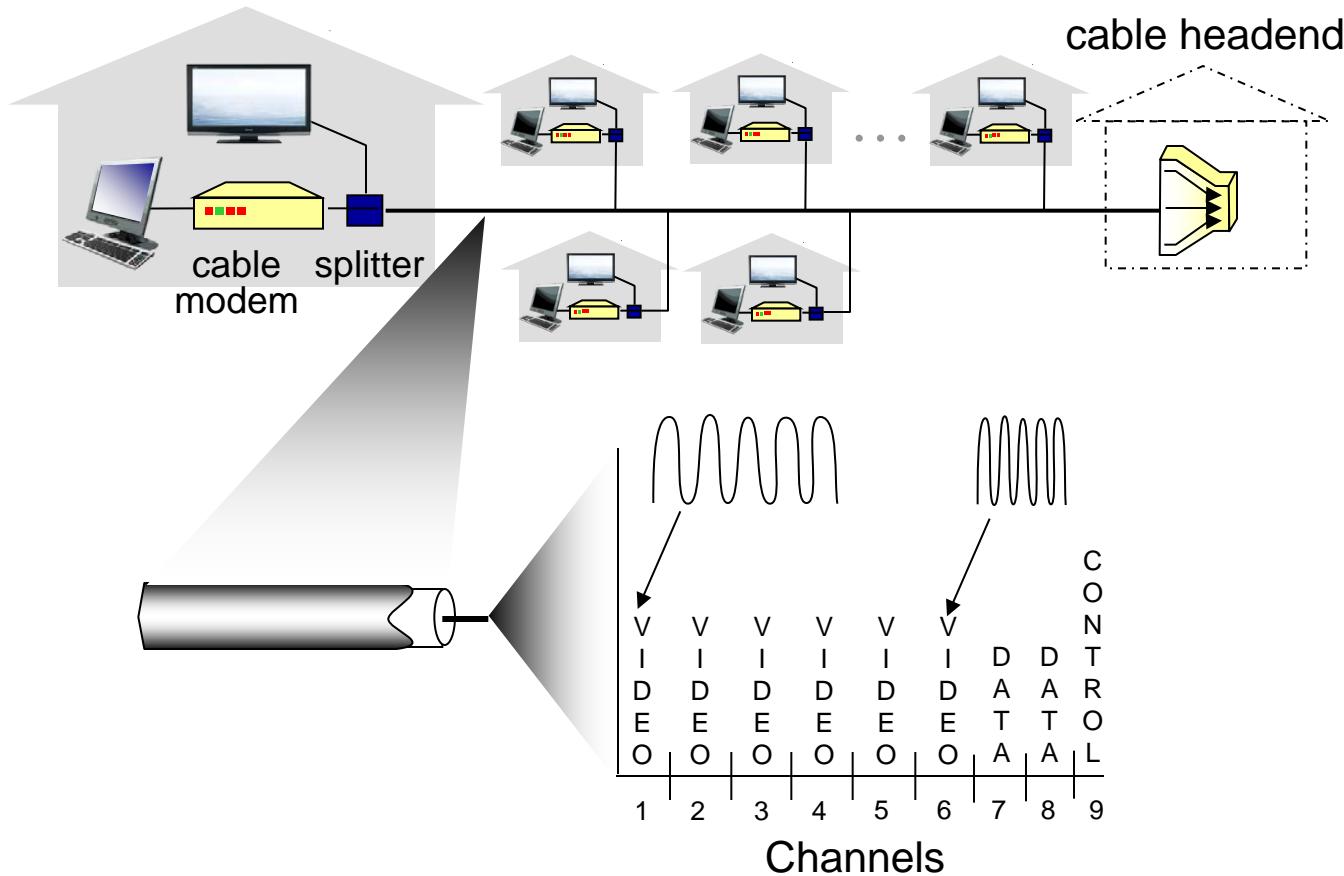


Access net: digital subscriber line (DSL)



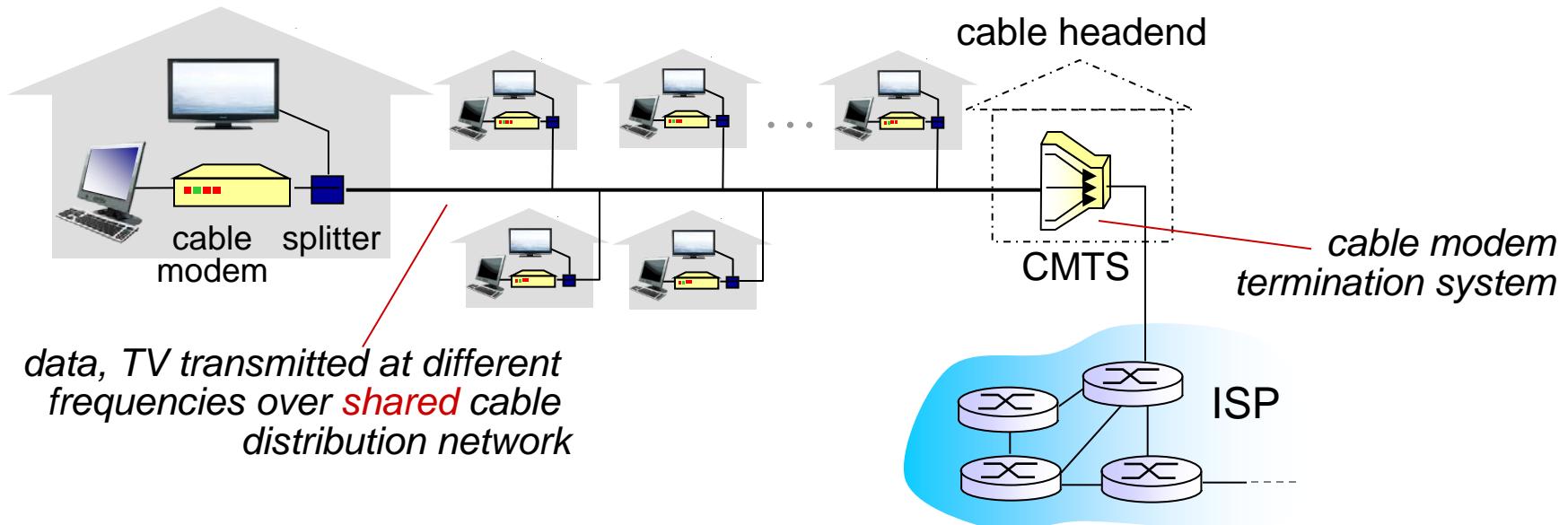
- ❖ use **existing** telephone line to central office DSLAM
 - data over DSL phone line goes to Internet
 - voice over DSL phone line goes to telephone net
- ❖ < 2.5 Mbps upstream transmission rate (typically < 1 Mbps)
- ❖ < 24 Mbps downstream transmission rate (typically < 10 Mbps)

Access net: cable network



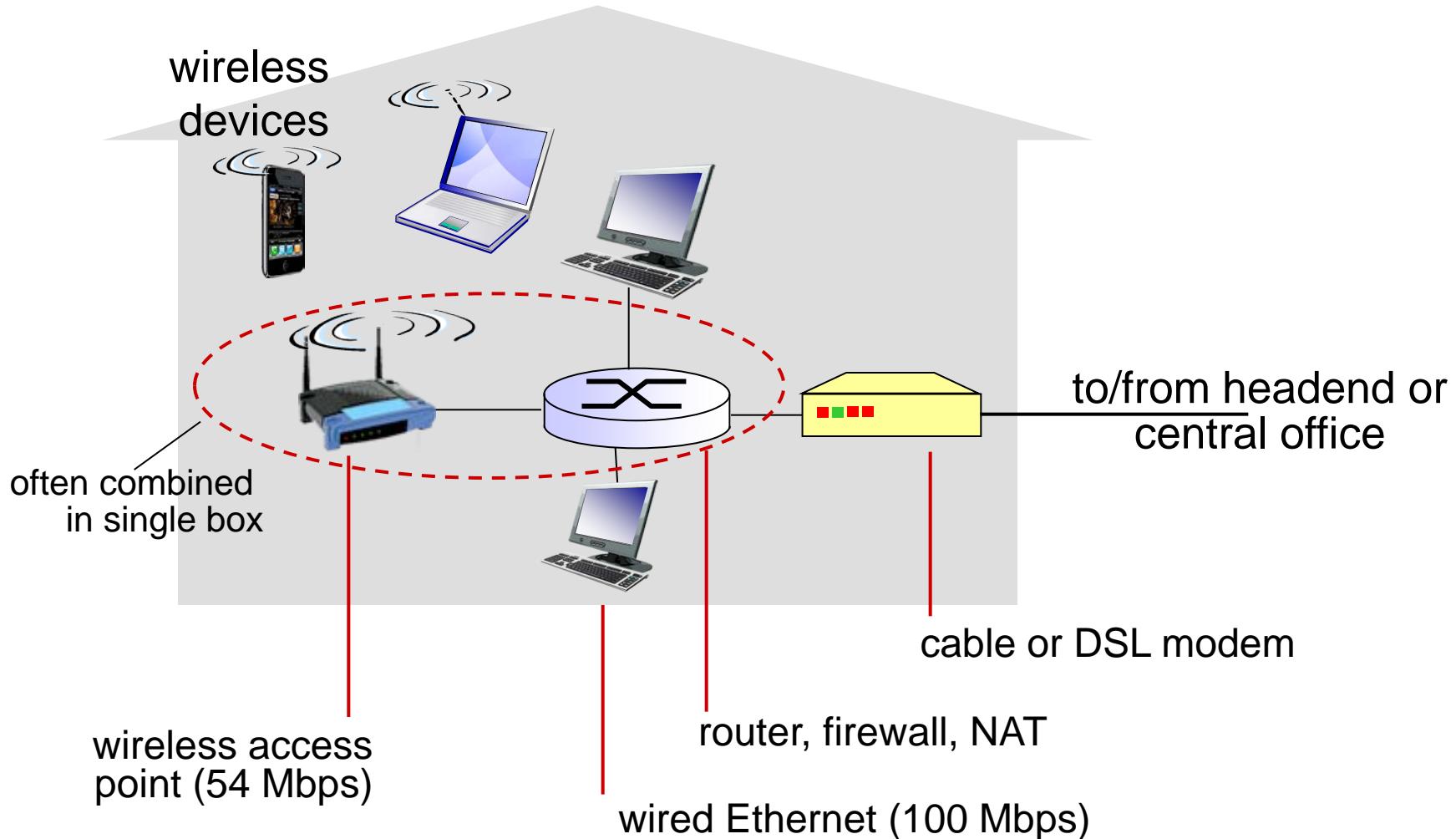
frequency division multiplexing: different channels transmitted in different frequency bands

Access net: cable network

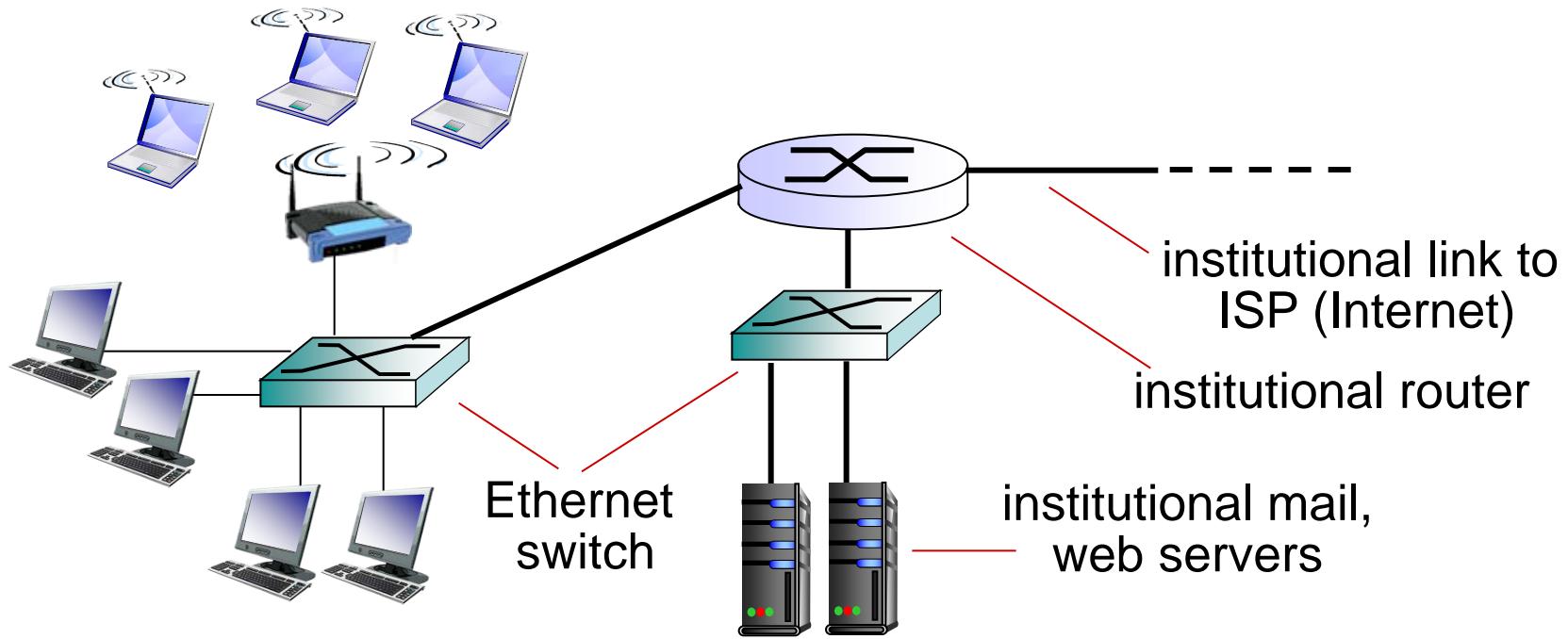


- ❖ HFC: hybrid fiber coax
 - asymmetric: up to 30Mbps downstream transmission rate, 2 Mbps upstream transmission rate
- ❖ network of cable, fiber attaches homes to ISP router
 - homes **share access network** to cable headend
 - unlike DSL, which has dedicated access to central office

Access net: home network



Enterprise access networks (Ethernet)



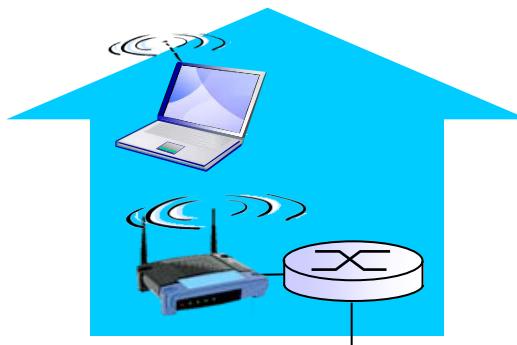
- ❖ typically used in companies, universities, etc
- ❖ 10 Mbps, 100Mbps, 1Gbps, 10Gbps transmission rates
- ❖ today, end systems typically connect into Ethernet switch

Wireless access networks

- ❖ shared wireless access network connects end system to router
 - via base station aka “access point”

wireless LANs:

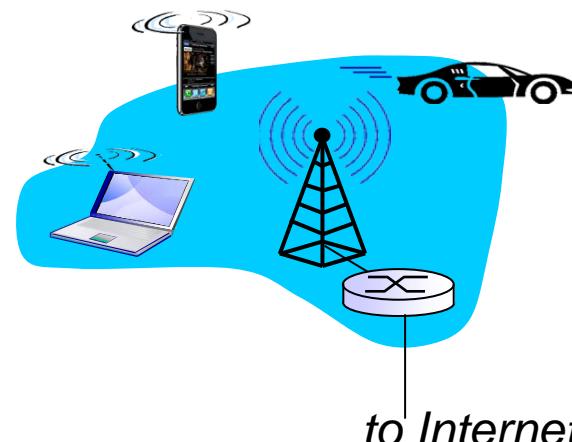
- within building (100 ft)
- 802.11b/g (WiFi): 11, 54 Mbps transmission rate



to Internet

wide-area wireless access

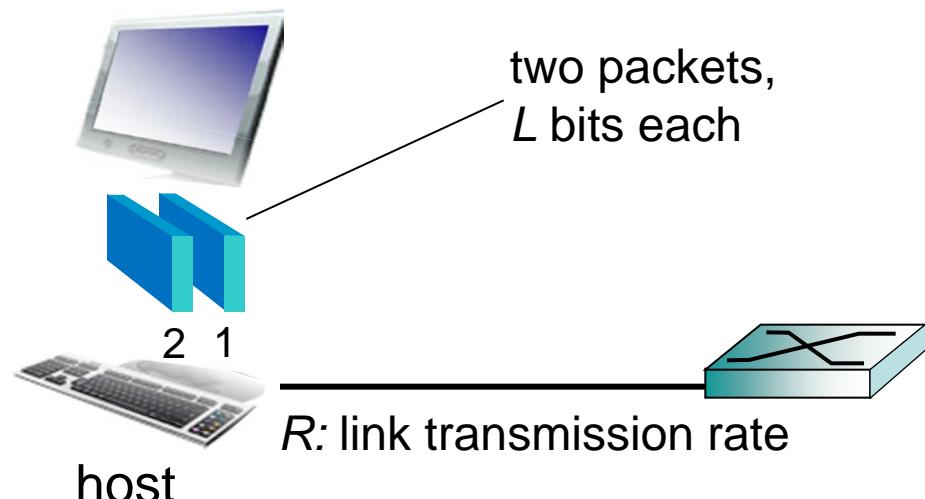
- provided by telco (cellular) operator, 10's km
- between 1 and 10 Mbps
- 3G, 4G: LTE



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*



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

Physical media

- ❖ **bit:** propagates between transmitter/receiver pairs
- ❖ **physical link:** what lies between transmitter & receiver
- ❖ **guided media:**
 - signals propagate in solid media: copper, fiber, coax
- ❖ **unguided media:**
 - signals propagate freely, e.g., radio

twisted pair (TP)

- ❖ two insulated copper wires
 - Category 5: 100 Mbps, 1 Gbps Ethernet
 - Category 6: 10Gbps



Physical media: coax, fiber

coaxial cable:

- ❖ two concentric copper conductors
- ❖ bidirectional
- ❖ broadband:
 - multiple channels on cable
 - HFC



fiber optic cable:

- ❖ glass fiber carrying light pulses, each pulse a bit
- ❖ high-speed operation:
 - high-speed point-to-point transmission (e.g., 10's-100's Gbps transmission rate)
- ❖ low error rate:
 - repeaters spaced far apart (up to 100 km)
 - immune to electromagnetic noise



Physical media: radio

- ❖ signal carried in electromagnetic spectrum
- ❖ no physical “wire”
- ❖ bidirectional
- ❖ propagation environment effects:
 - reflection
 - obstruction by objects
 - interference

radio link types:

- ❖ terrestrial microwave
 - e.g. up to 45 Mbps channels
- ❖ LAN (e.g., WiFi)
 - 11Mbps, 54 Mbps
- ❖ wide-area (e.g., cellular)
 - 3G cellular: ~ few Mbps
- ❖ satellite
 - Kbps to 45Mbps channel (or multiple smaller channels)
 - 270 msec end-end delay
 - geosynchronous versus low altitude

Chapter I: roadmap

I.1 what *is* the Internet?

I.2 network edge

- end systems, access networks, links

I.3 network core

- packet switching, circuit switching, network structure

I.4 delay, loss, throughput in networks

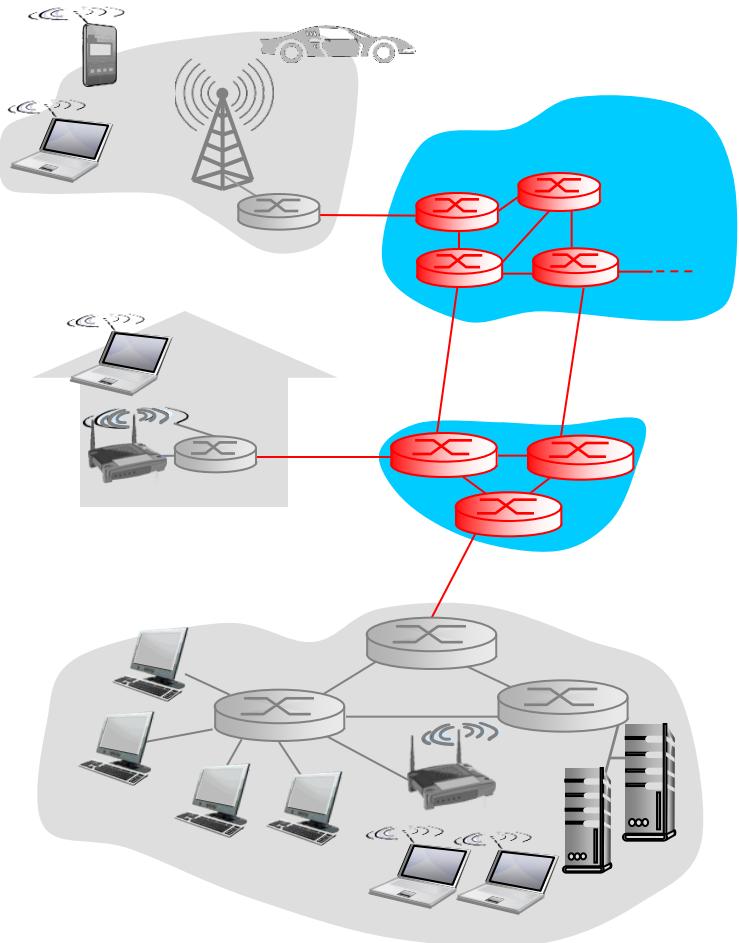
I.5 protocol layers, service models

I.6 networks under attack: security

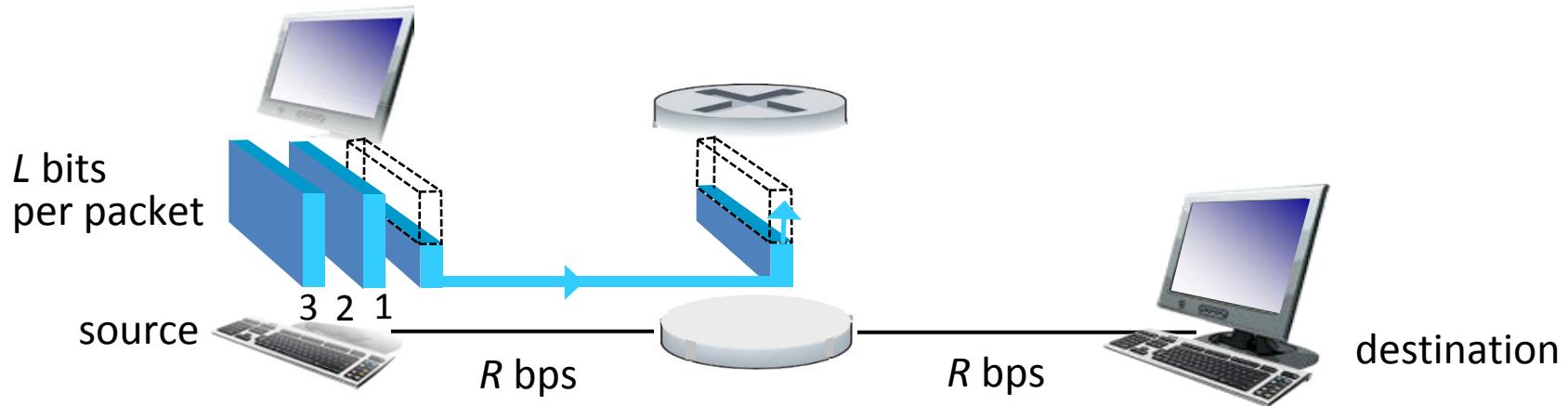
I.7 history

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



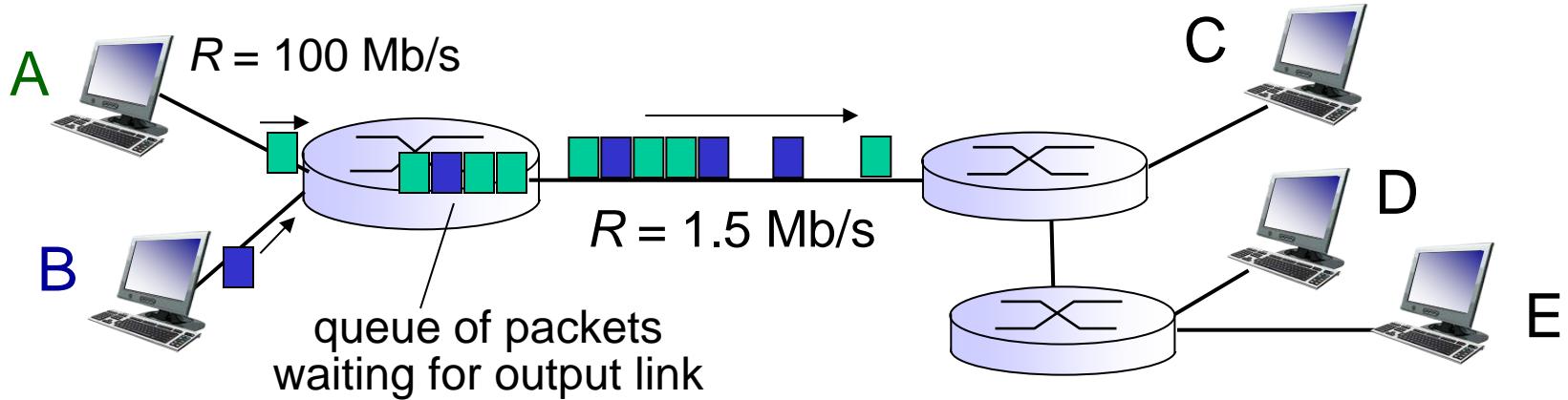
- ❖ takes L/R seconds to transmit (push out) L -bit packet into link at R bps
- ❖ **store and forward:** entire packet must arrive at router before it can be transmitted on next link
- ❖ end-end delay = $2L/R$ (assuming zero propagation delay)

one-hop numerical example:

- $L = 7.5$ Mbits
- $R = 1.5$ Mbps
- one-hop transmission delay = 5 sec

} more on delay shortly ...

Packet Switching: queueing delay, loss



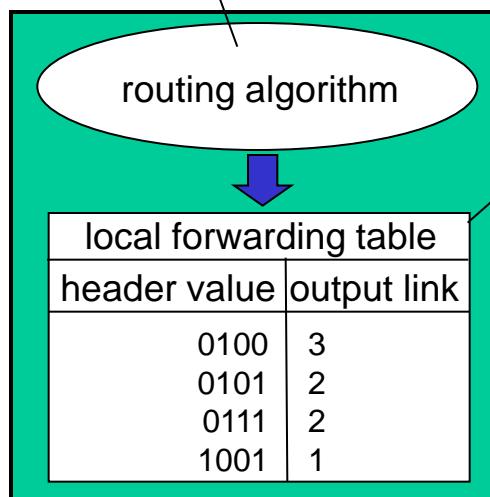
queuing and loss:

- ❖ If arrival rate (in bits) to link exceeds transmission rate of link for a period of time:
 - packets will queue, wait to be transmitted on link
 - packets can be dropped (lost) if memory (buffer) fills up

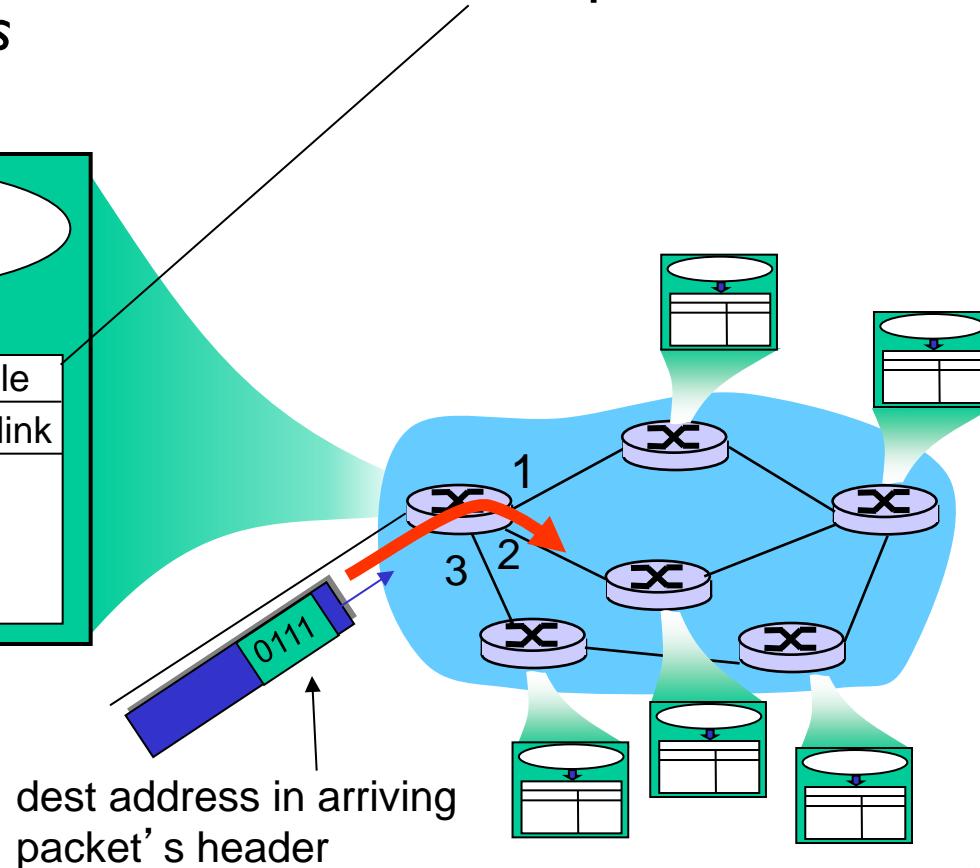
Two key network-core functions

routing: determines source-destination route taken by packets

- *routing algorithms*



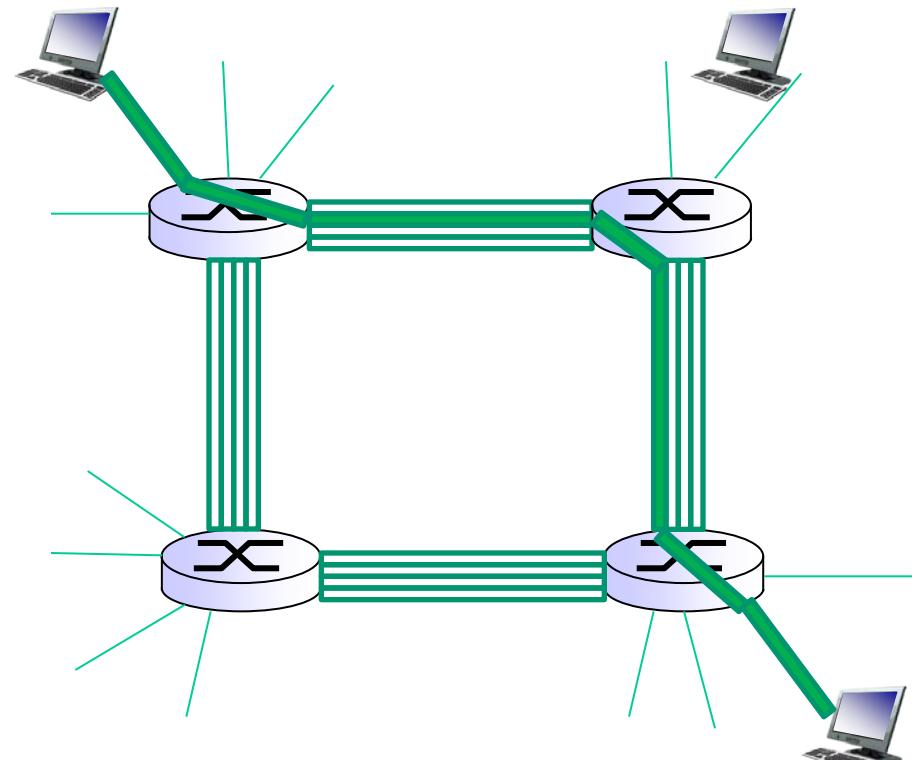
forwarding: move packets from router's input to appropriate router output



Alternative core: circuit switching

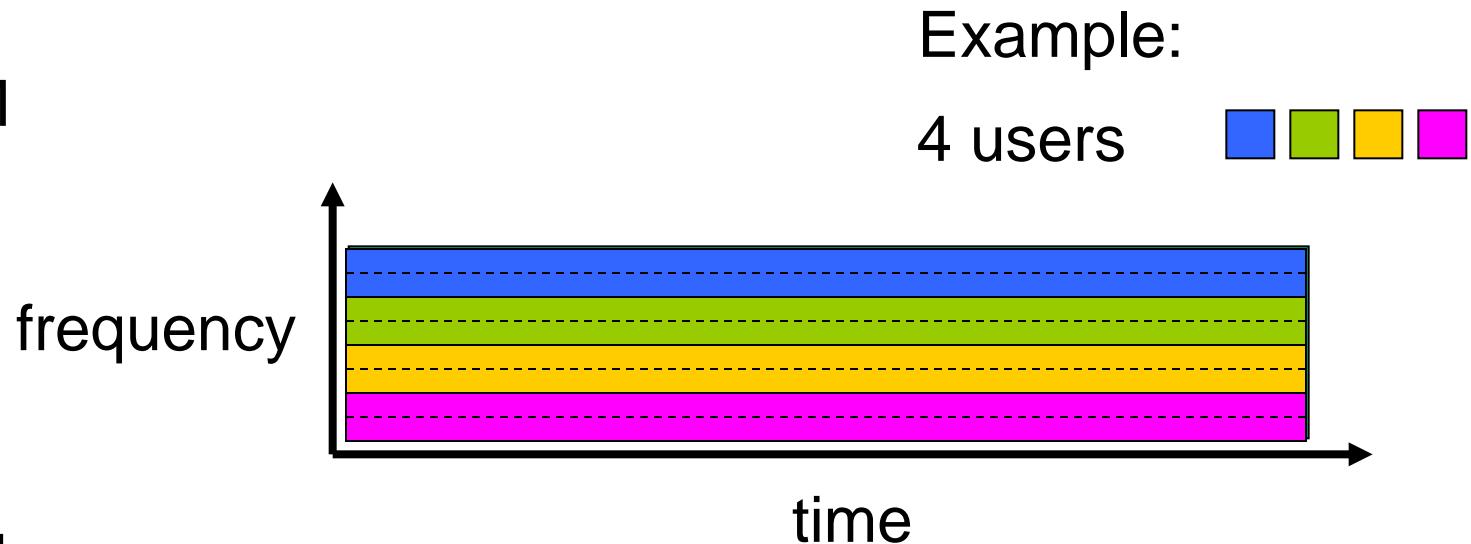
end-end resources allocated
to, reserved for “call”
between source & dest:

- ❖ In diagram, each link has four circuits.
 - call gets 2nd circuit in top link and 1st 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

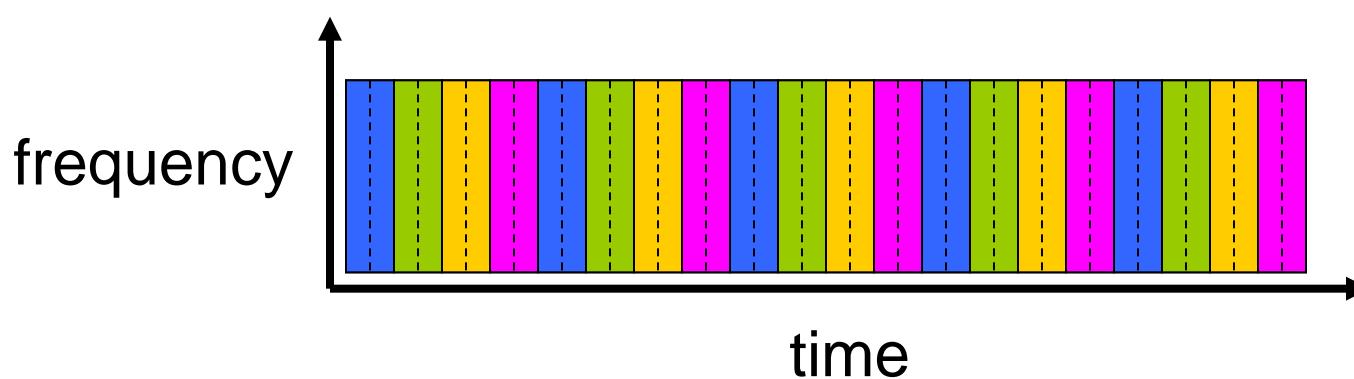


Circuit switching: FDM versus TDM

FDM



TDM

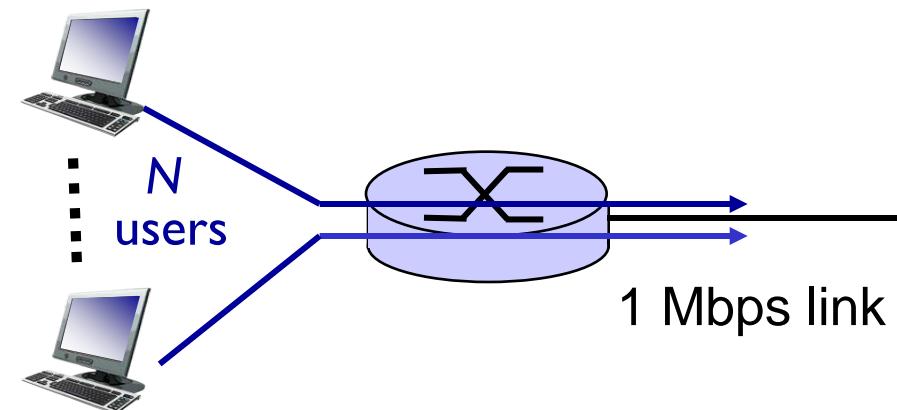


Packet switching versus circuit switching

packet switching allows more users to use network!

example:

- 1 Mb/s link
- each user:
 - 100 kb/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 *

Q: how did we get value 0.0004?

Q: what happens if > 35 users ?

* Check out the online interactive exercises for more examples

Packet switching versus circuit switching

is packet switching a “slam dunk winner?”

- ❖ great for bursty data
 - resource sharing
 - simpler, no call setup
- ❖ excessive congestion possible: packet delay and loss
 - protocols needed for reliable data transfer, congestion control
- ❖ Q: How to provide circuit-like behavior?
 - bandwidth guarantees needed for audio/video apps
 - still an unsolved problem (chapter 7)

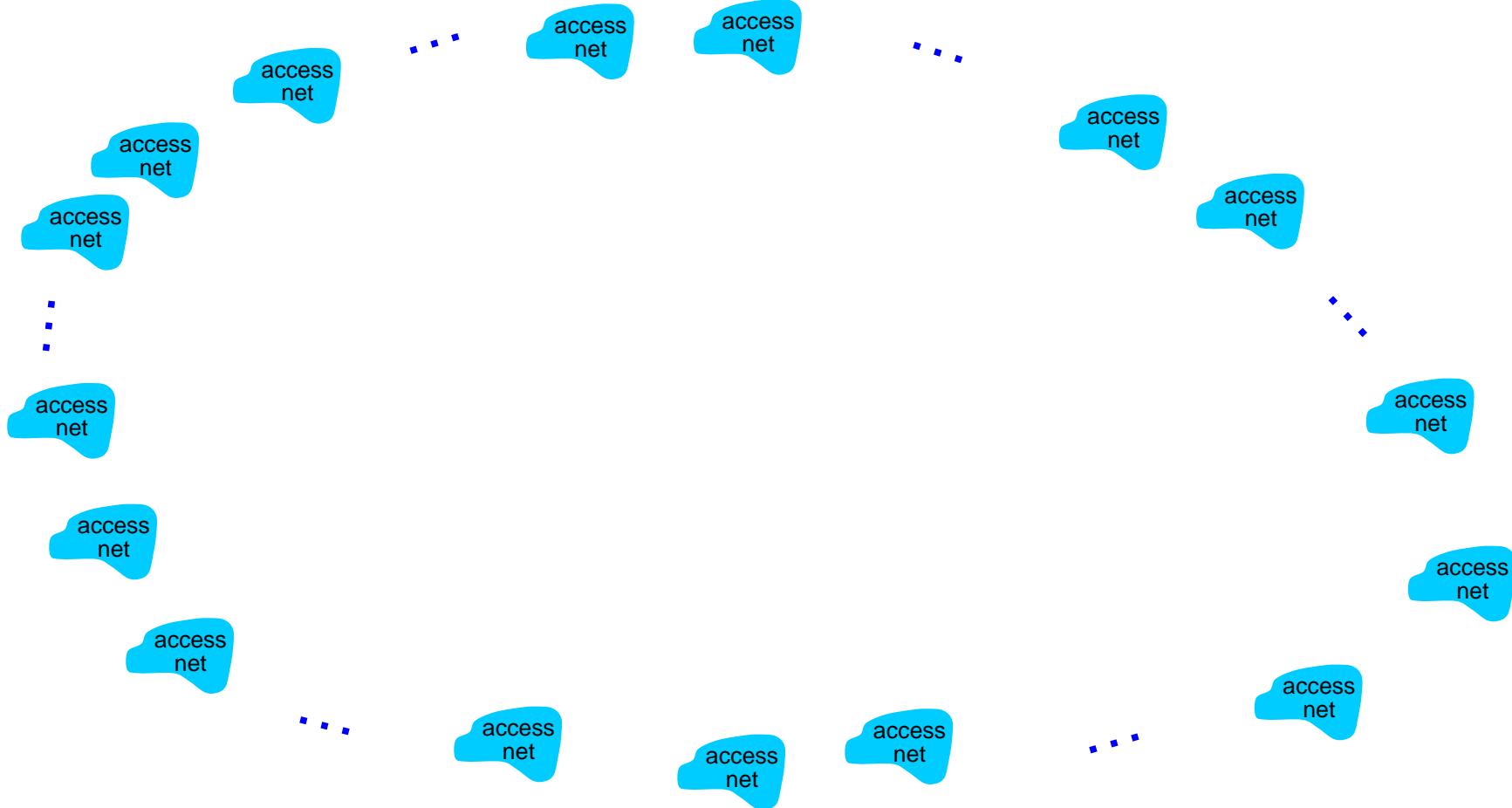
Q: human analogies of reserved resources (circuit switching)
versus on-demand allocation (packet-switching)?

Internet structure: network of networks

- ❖ End systems connect to Internet via **access ISPs** (Internet Service Providers)
 - Residential, company and university ISPs
- ❖ Access ISPs in turn must be interconnected.
 - ❖ So that any two hosts can send packets to each other
- ❖ Resulting network of networks is very complex
 - ❖ Evolution was driven by **economics** and **national policies**
- ❖ Let's take a stepwise approach to describe current Internet structure

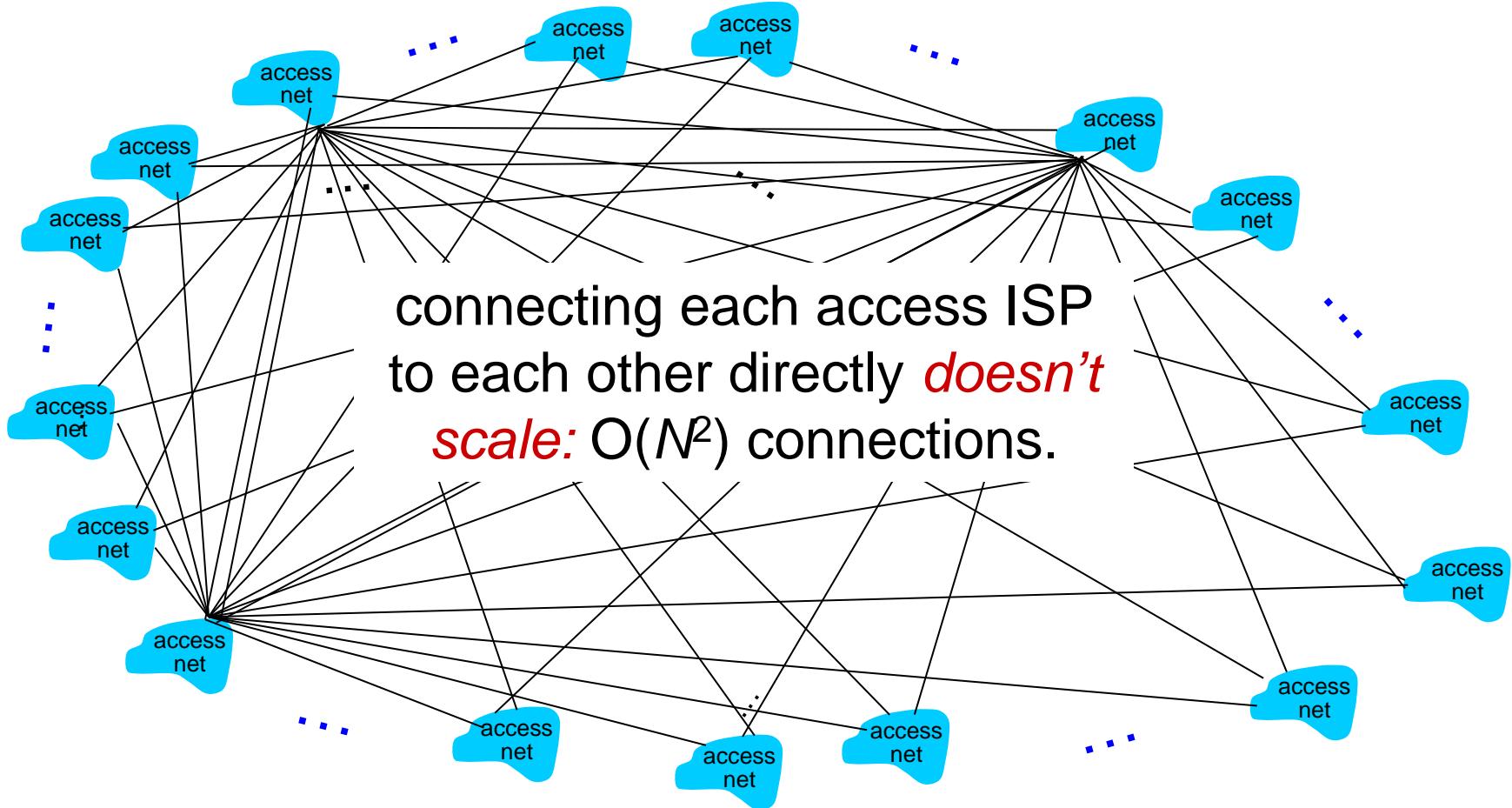
Internet structure: network of networks

Question: given *millions* of access ISPs, how to connect them together?



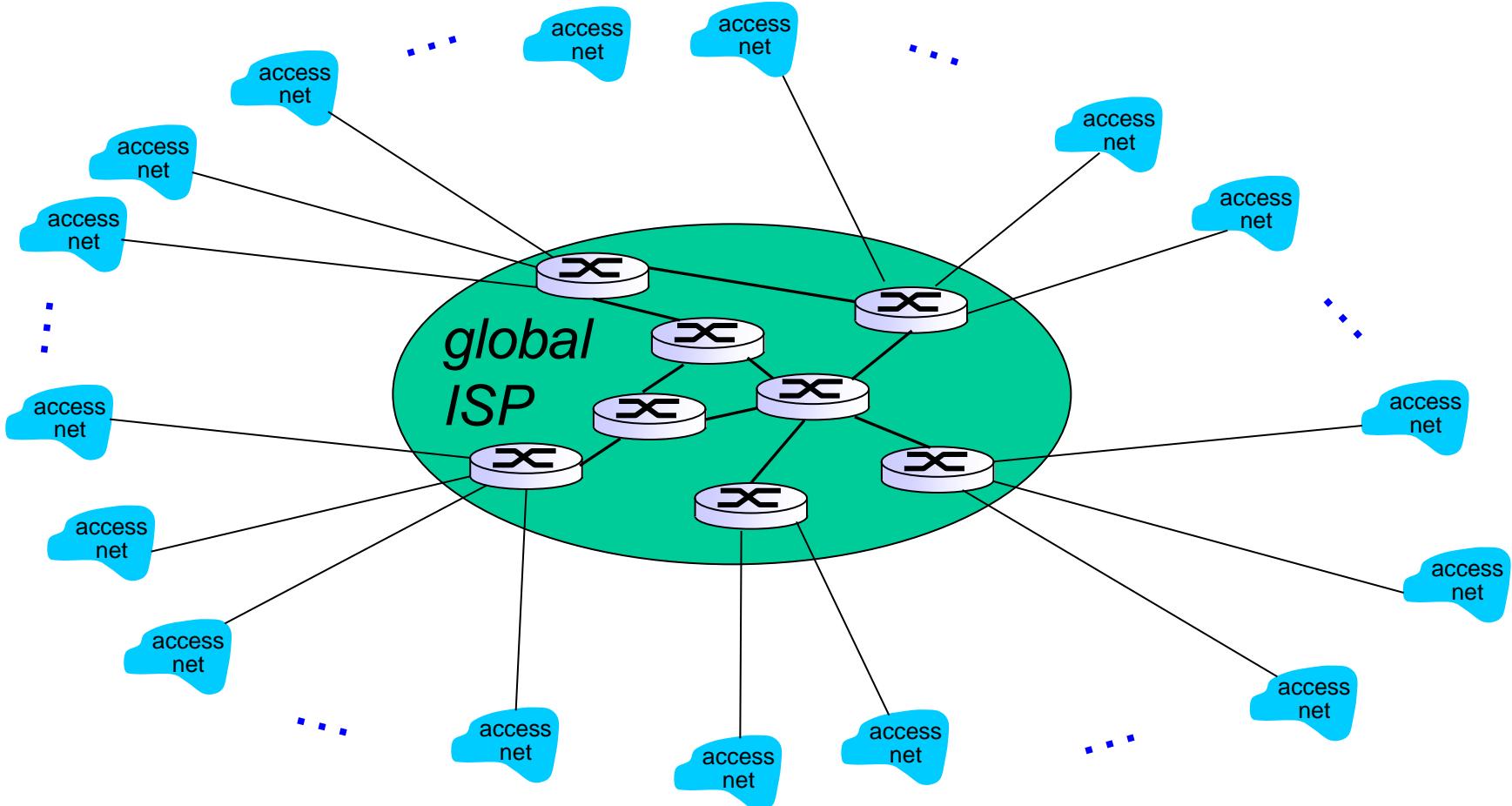
Internet structure: network of networks

Option: connect each access ISP to every other access ISP?



Internet structure: network of networks

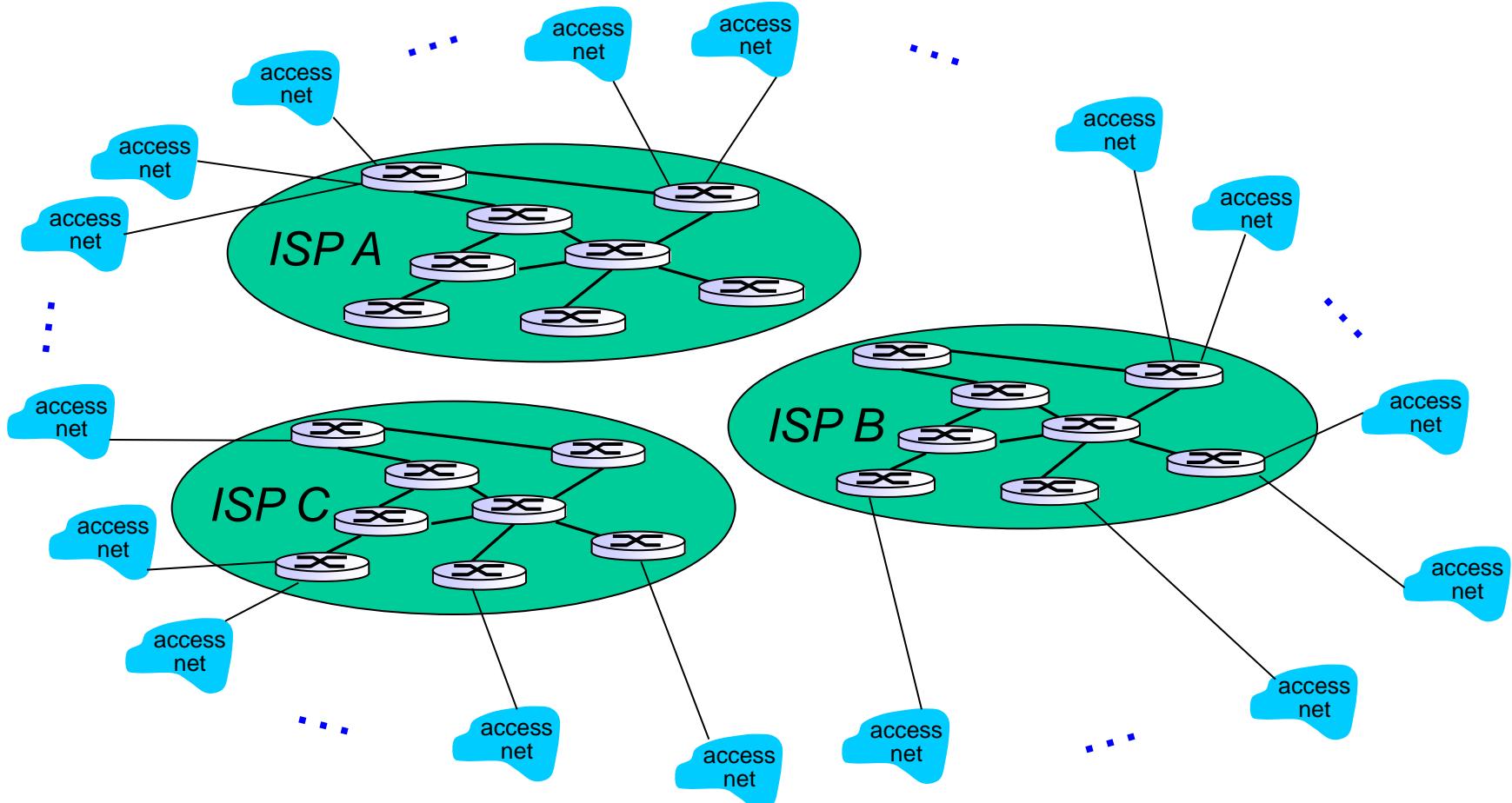
Option: connect each access ISP to a global transit ISP? *Customer and provider ISPs have economic agreement.*



Internet structure: network of networks

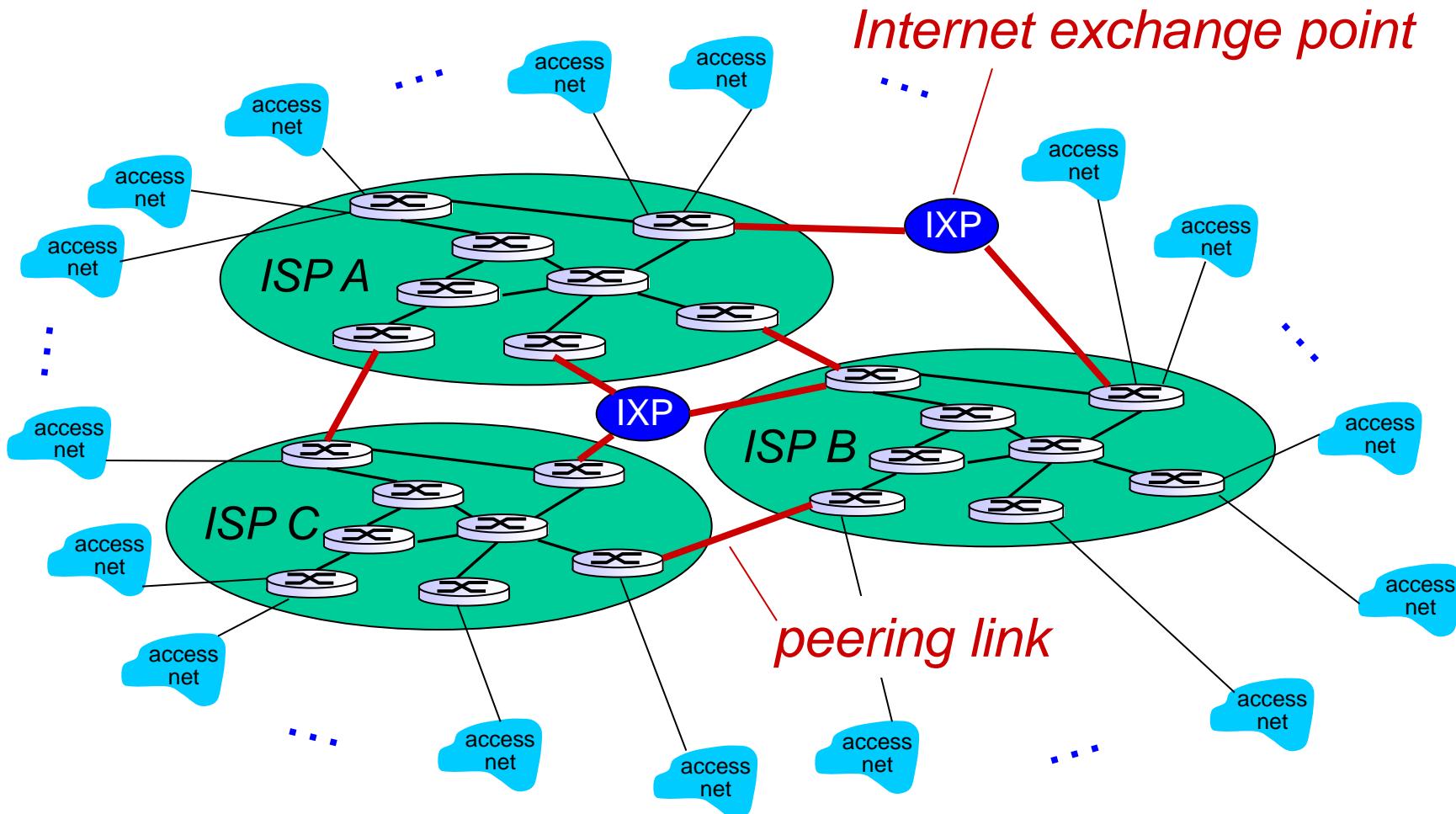
But if one global ISP is viable business, there will be competitors

....



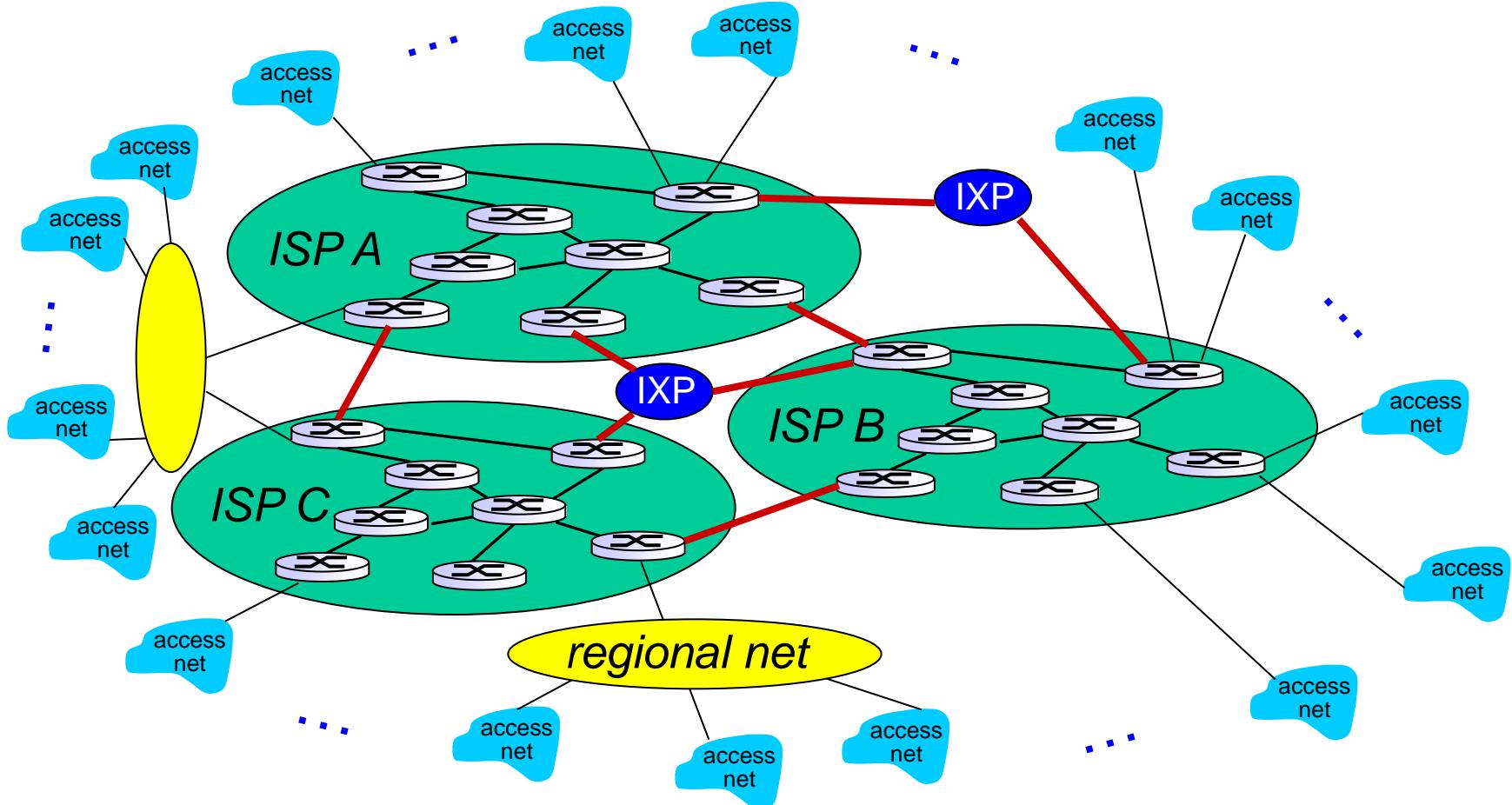
Internet structure: network of networks

But if one global ISP is viable business, there will be competitors
.... which must be interconnected



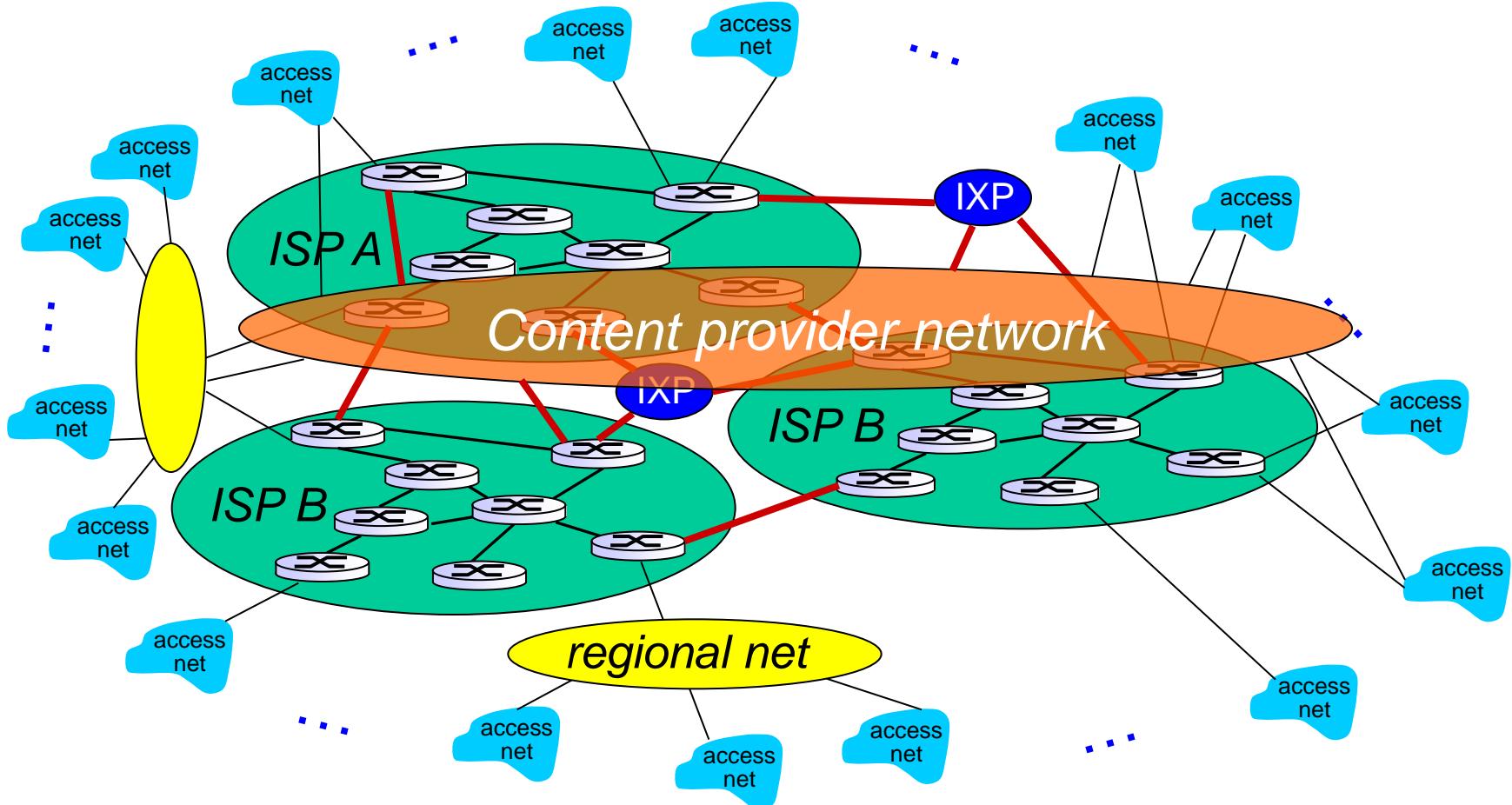
Internet structure: network of networks

... and regional networks may arise to connect access nets to ISPs

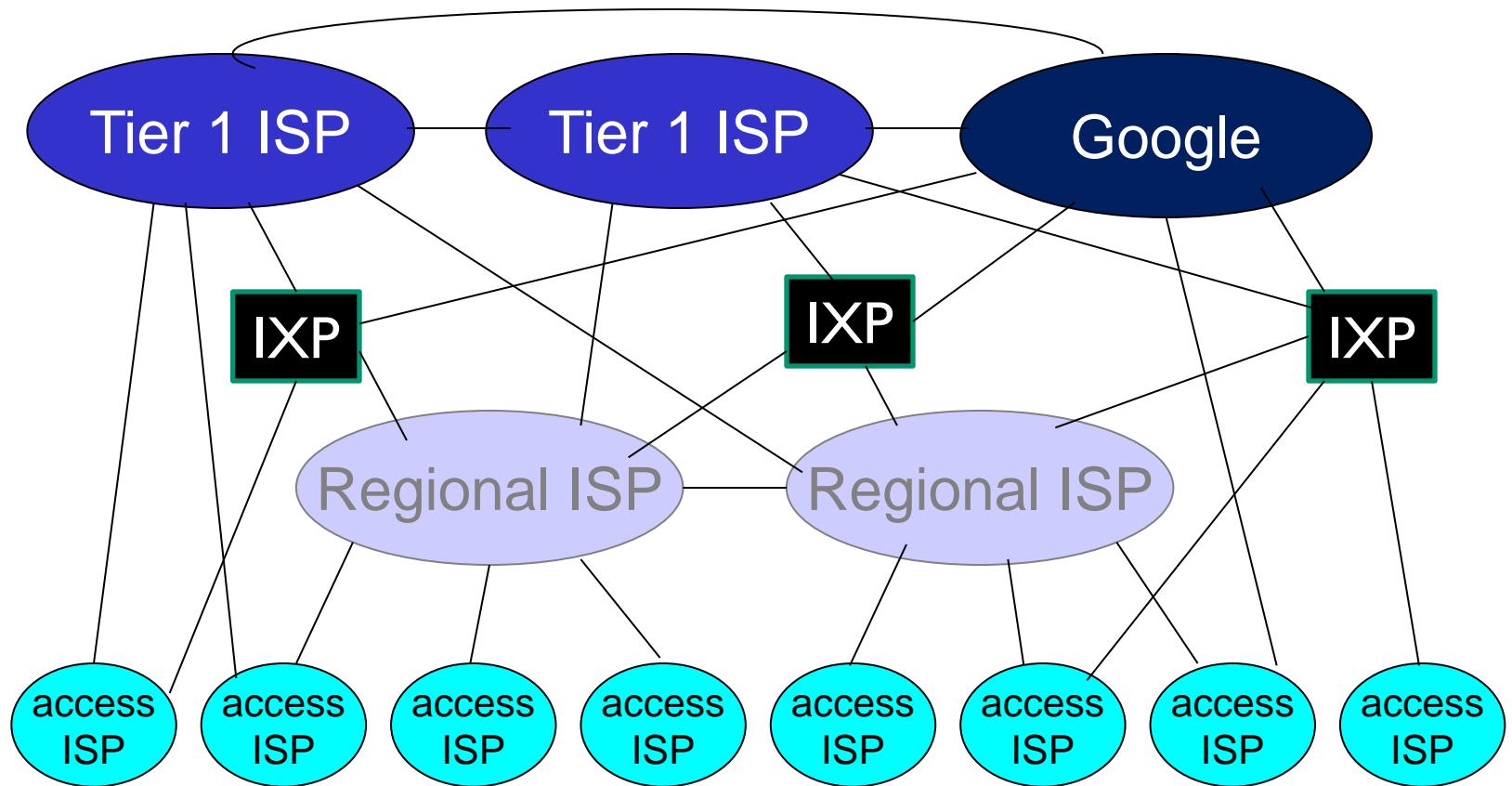


Internet structure: network of networks

... and content provider networks (e.g., Google, Microsoft, Akamai) may run their own network, to bring services, content close to end users

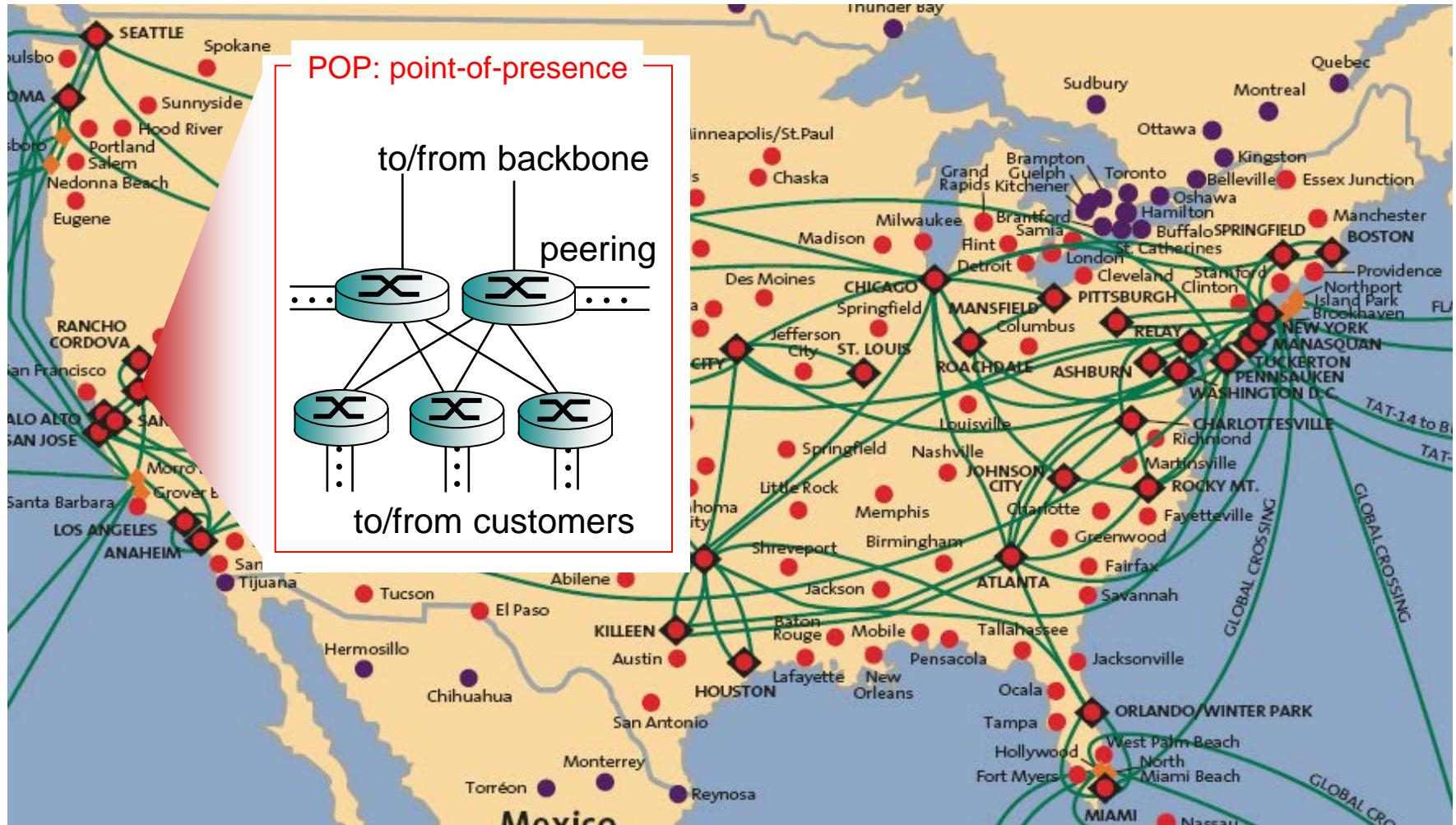


Internet structure: network of networks



- ❖ at center: small # of well-connected large networks
 - “tier-1” commercial ISPs (e.g., Level 3, Sprint, AT&T, NTT), national & international coverage
 - content provider network (e.g, Google): private network that connects its data centers to Internet, often bypassing tier-1, regional ISPs

Tier-I ISP: e.g., Sprint



Chapter I: roadmap

I.1 what *is* the Internet?

I.2 network edge

- end systems, access networks, links

I.3 network core

- packet switching, circuit switching, network structure

I.4 delay, loss, throughput in networks

I.5 protocol layers, service models

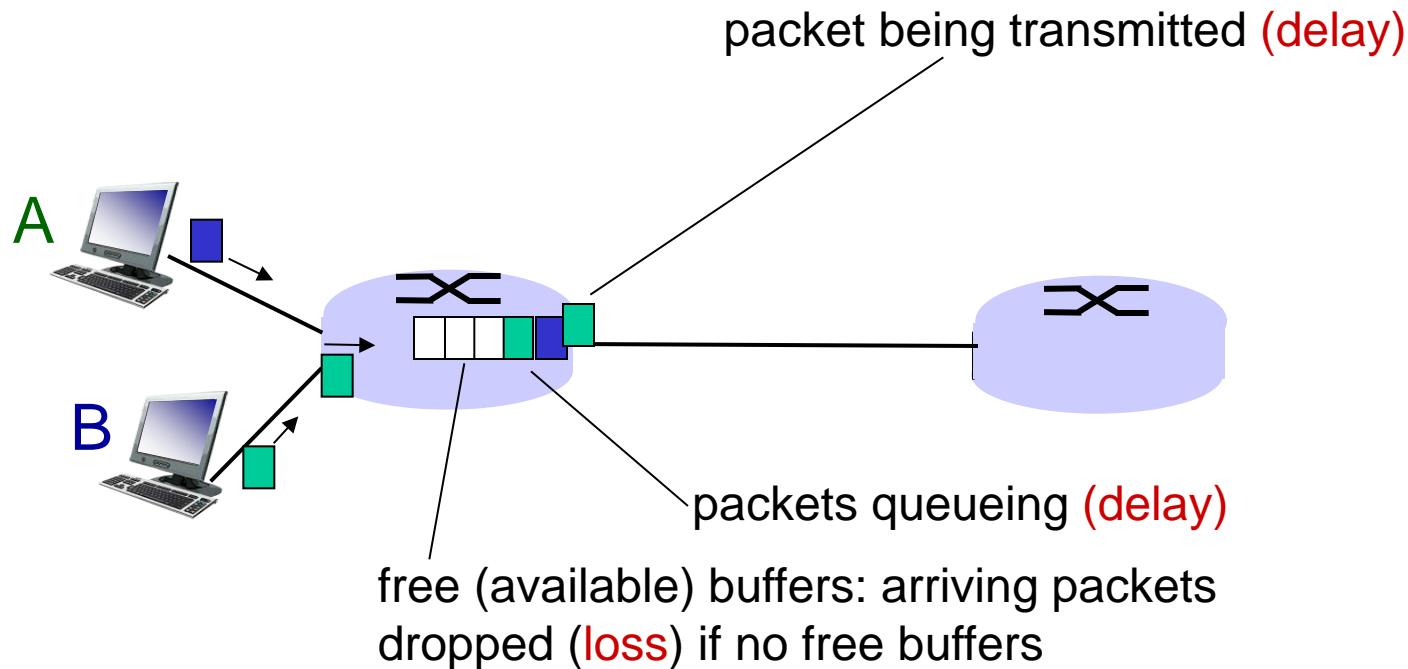
I.6 networks under attack: security

I.7 history

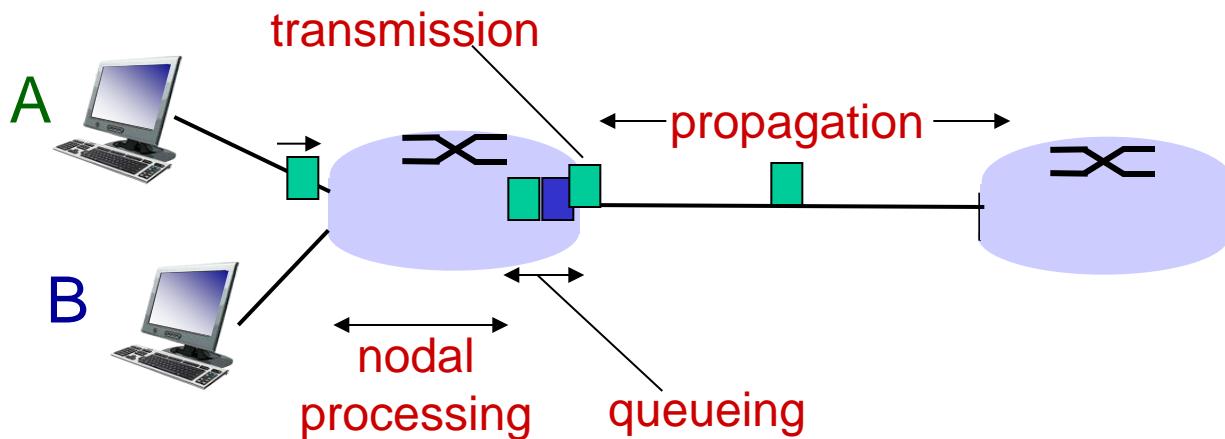
How do loss and delay occur?

packets queue in router buffers

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



Four sources of packet delay



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

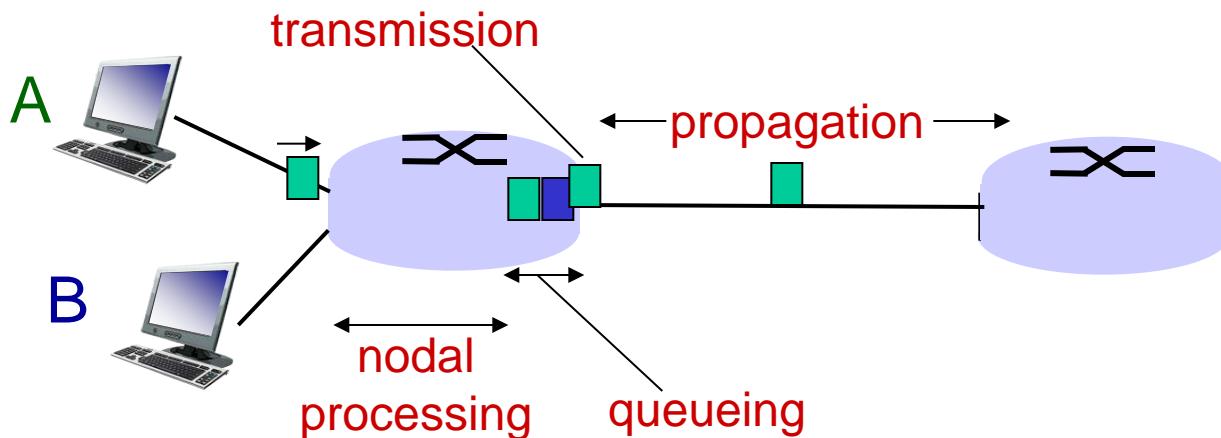
d_{proc} : nodal processing

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

d_{queue} : queueing delay

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

Four sources of packet delay



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

d_{trans} : transmission delay:

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

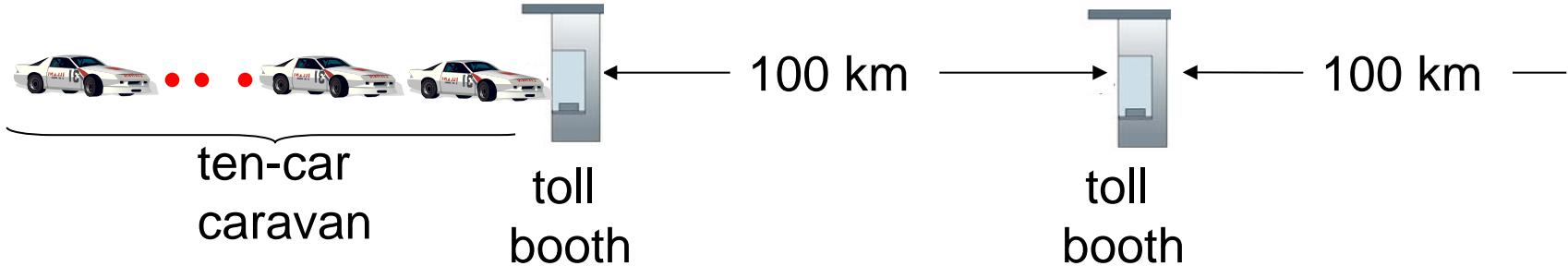
d_{trans} and d_{prop}
very different

d_{prop} : propagation delay:

- d : length of physical link
- s : propagation speed in medium ($\sim 2 \times 10^8 \text{ m/sec}$)
- $d_{\text{prop}} = d/s$

* Check out the Java applet for an interactive animation on trans vs. prop delay

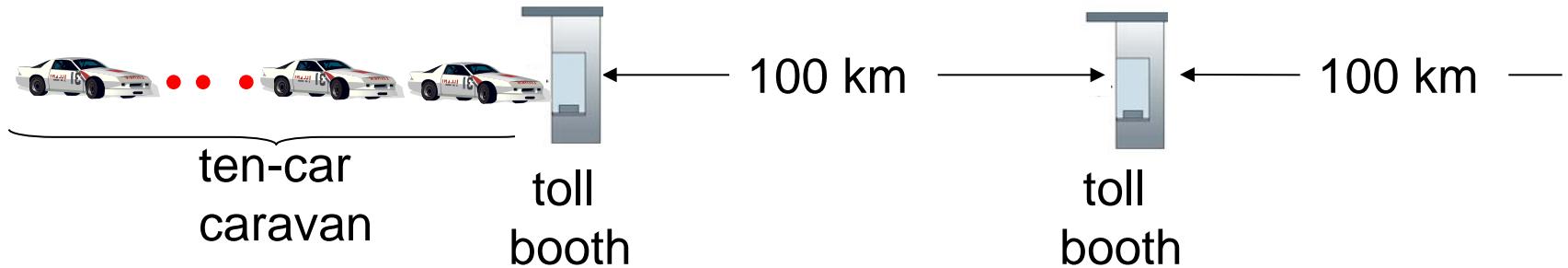
Caravan analogy



- ❖ 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:
 $100\text{km}/(100\text{km/hr}) = 1\text{ hr}$
- A: 62 minutes

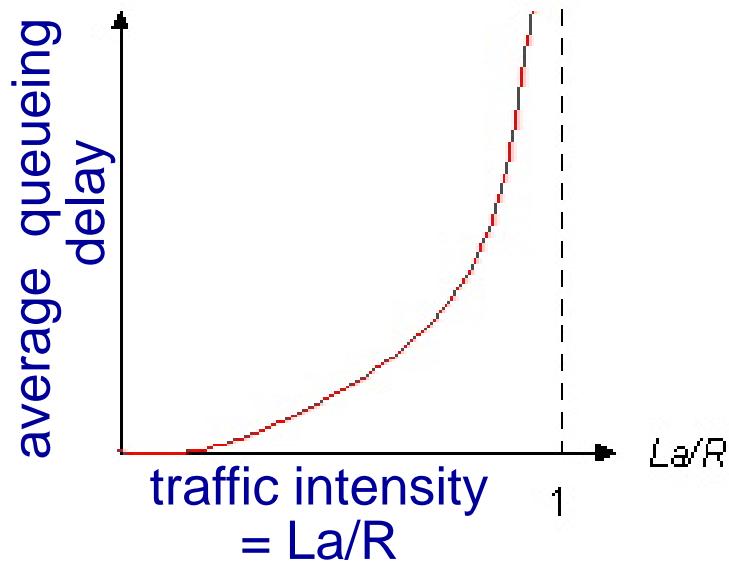
Caravan analogy (more)



- ❖ 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, 1st car arrives at second booth; three cars still at 1st booth.

Queueing delay (revisited)

- ❖ R : link bandwidth (bps)
- ❖ L : packet length (bits)
- ❖ a : average packet arrival rate



- ❖ $La/R \sim 0$: avg. queueing delay small
- ❖ $La/R \rightarrow 1$: avg. queueing delay large
- ❖ $La/R > 1$: more “work” arriving than can be serviced, average delay infinite!



$La/R \sim 0$

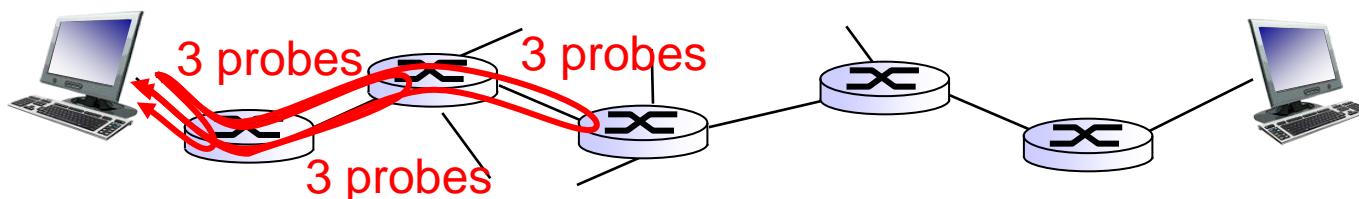


$La/R \rightarrow 1$

* Check out the Java applet for an interactive animation on queueing and loss

“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
 - router i will return packets to sender
 - sender times interval between transmission and reply.



“Real” Internet delays, routes

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

3 delay measurements from
gaia.cs.umass.edu to cs-gw.cs.umass.edu

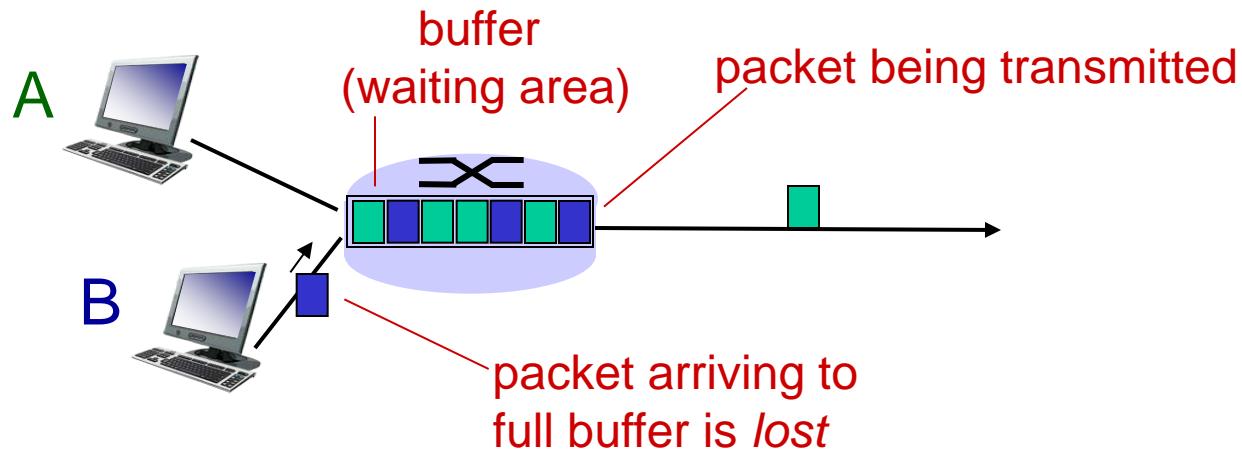
1	cs-gw (128.119.240.254)	1 ms	1 ms	2 ms
2	border1-rt-fa5-1-0.gw.umass.edu (128.119.3.145)	1 ms	1 ms	2 ms
3	cht-vbns.gw.umass.edu (128.119.3.130)	6 ms	5 ms	5 ms
4	jn1-at1-0-0-19.wor.vbns.net (204.147.132.129)	16 ms	11 ms	13 ms
5	jn1-so7-0-0-0.wae.vbns.net (204.147.136.136)	21 ms	18 ms	18 ms
6	abilene-vbns.abilene.ucaid.edu (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

trans-oceanic link

* Do some traceroutes from exotic countries at www.traceroute.org

Packet loss

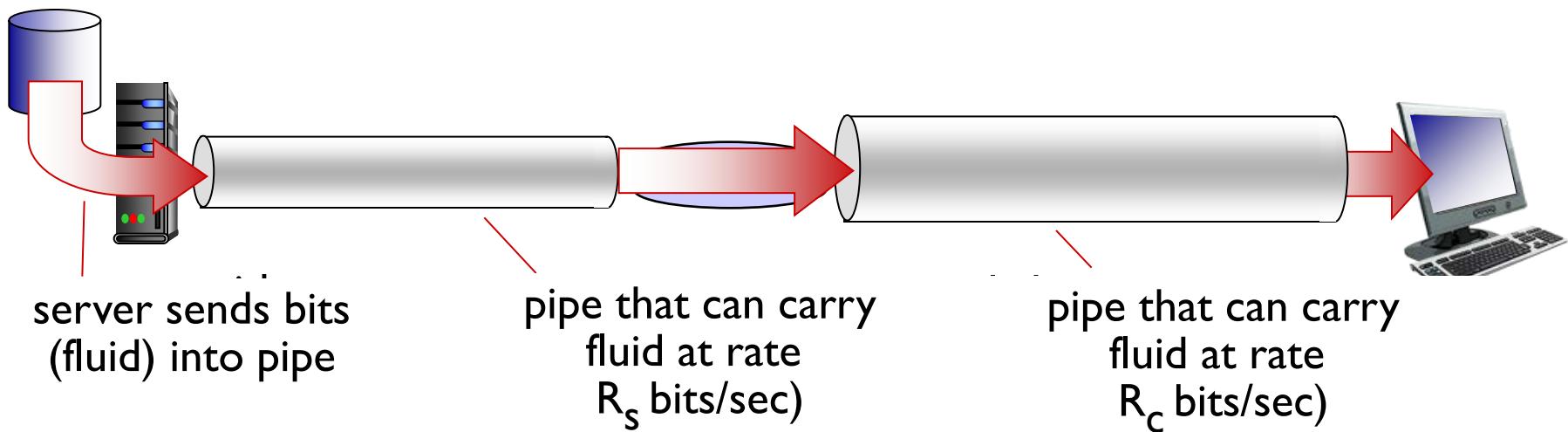
- ❖ queue (aka buffer) preceding link in buffer has finite capacity
- ❖ packet arriving to full queue dropped (aka lost)
- ❖ lost packet may be retransmitted by previous node, by source end system, or not at all



* Check out the Java applet for an interactive animation on queuing and loss

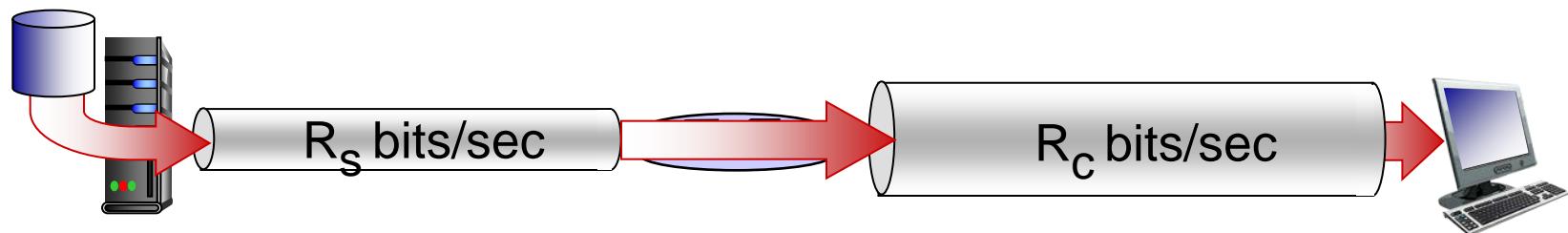
Throughput

- ❖ **throughput:** rate (bits/time unit) at which bits transferred between sender/receiver
 - *instantaneous:* rate at given point in time
 - *average:* rate over longer period of time

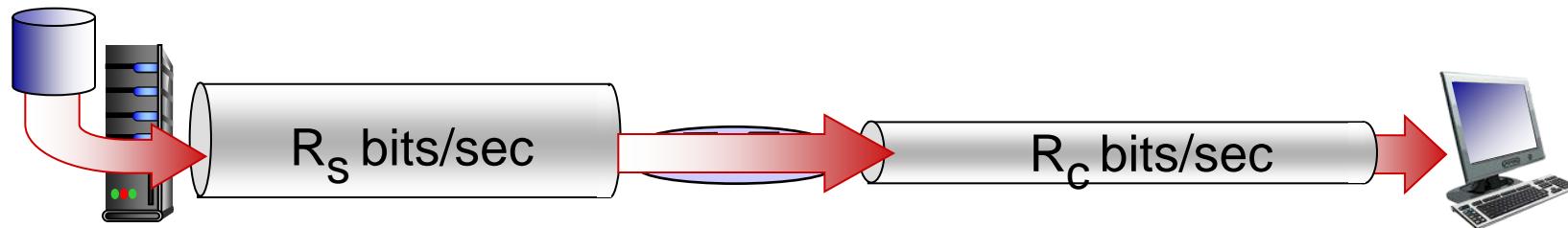


Throughput (more)

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



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

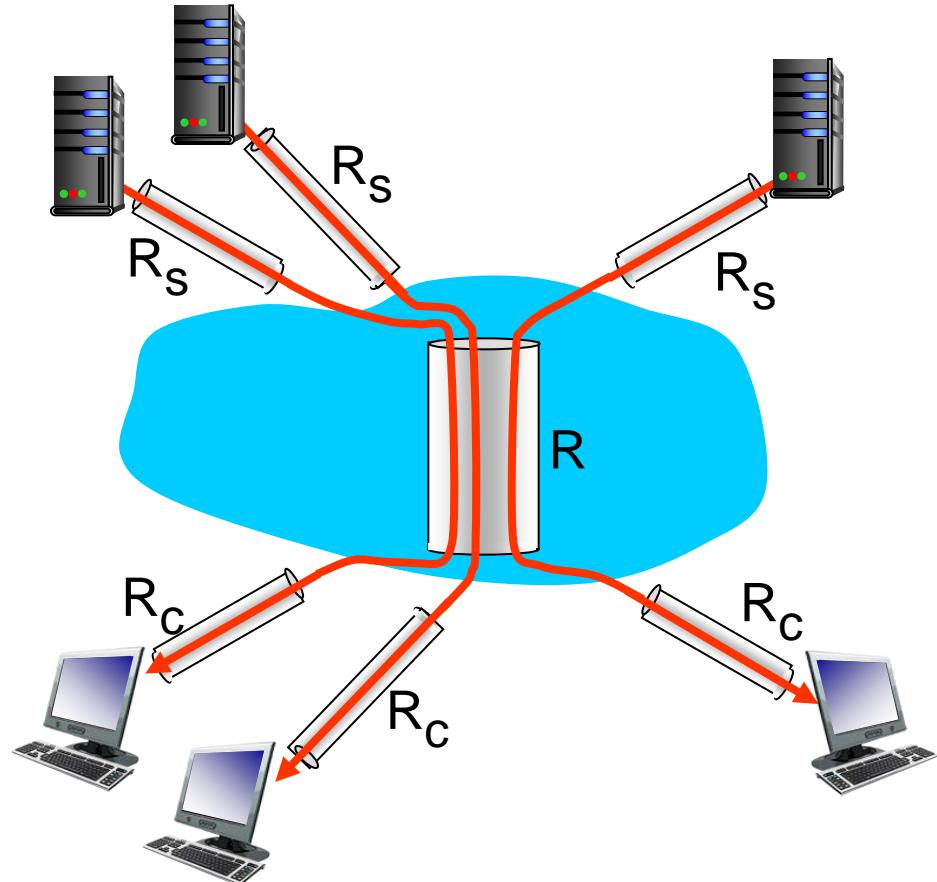


bottleneck link

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

Throughput: Internet scenario

- ❖ per-connection end-end throughput: $\min(R_c, R_s, R/10)$
- ❖ in practice: R_c or R_s is often bottleneck



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

Chapter I: roadmap

I.1 what *is* the Internet?

I.2 network edge

- end systems, access networks, links

I.3 network core

- packet switching, circuit switching, network structure

I.4 delay, loss, throughput in networks

I.5 protocol layers, service models

I.6 networks under attack: security

I.7 history

Protocol “layers”

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

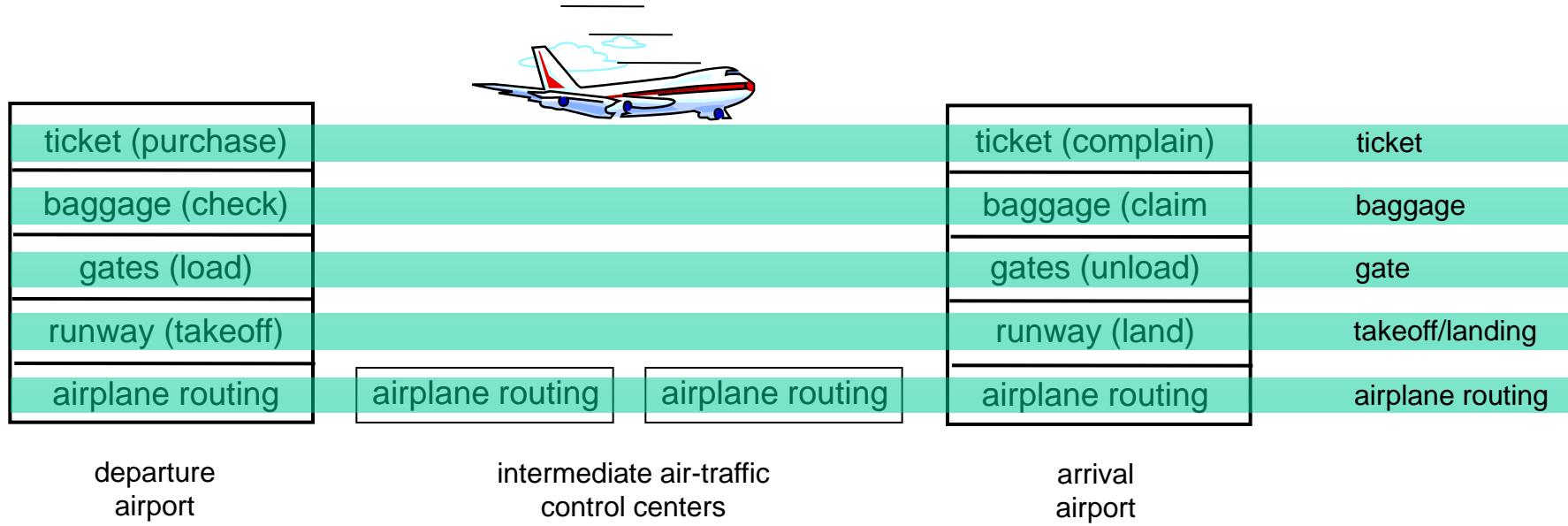
- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

Question:

is there any hope of
organizing structure of
network?

.... or at least our
discussion of networks?

Layering of airline functionality



layers: each layer implements a service

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

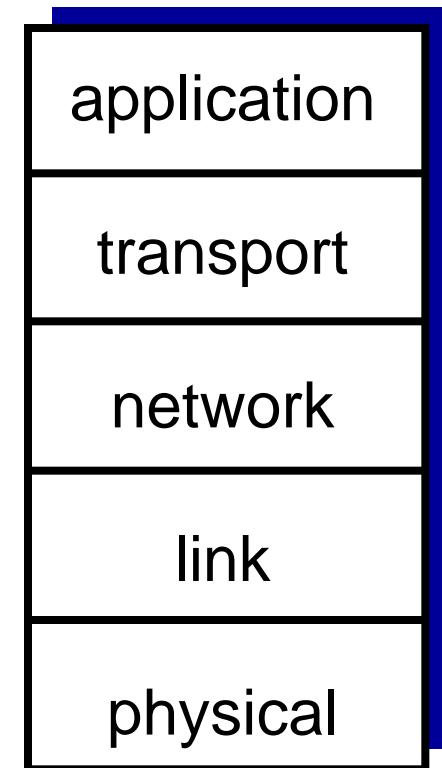
Why layering?

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 of implementation of layer's service transparent to rest of system
 - e.g., change in gate procedure doesn't affect rest of system
- ❖ layering considered harmful?

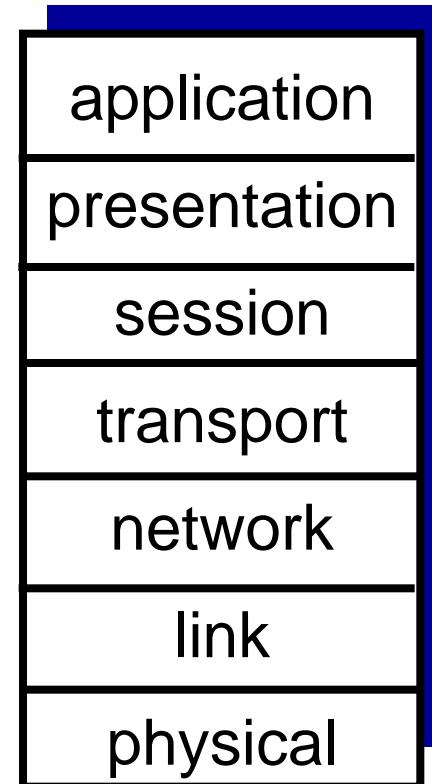
Internet protocol stack

- ❖ *application*: supporting network applications
 - FTP, 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”

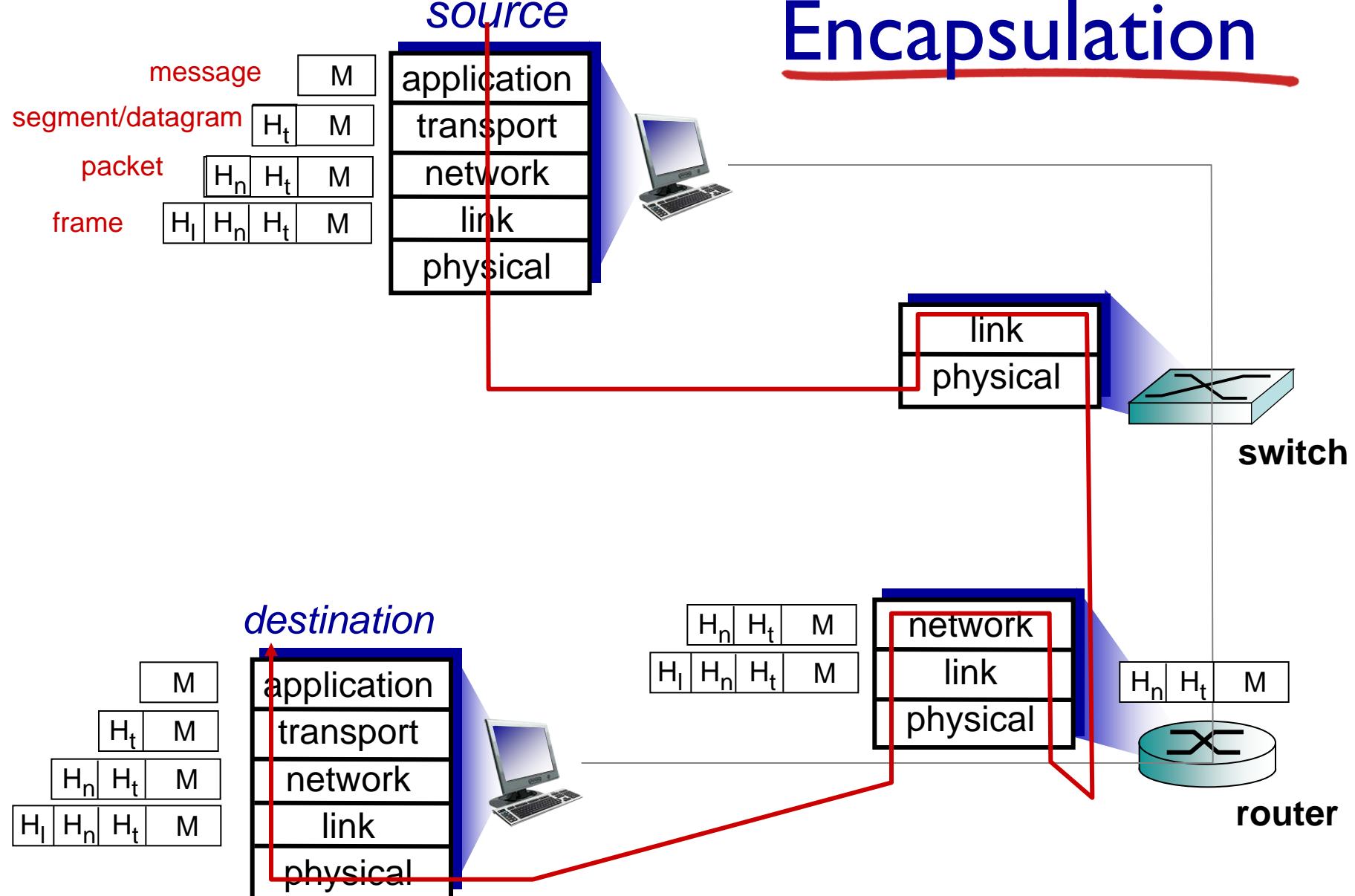


ISO/OSI reference model

- ❖ ***presentation:*** allow applications to interpret meaning of data, e.g., encryption, compression, machine-specific conventions
- ❖ ***session:*** synchronization, checkpointing, recovery of data exchange
- ❖ Internet stack “missing” these layers!
 - these services, *if needed*, must be implemented in application
 - needed?



Encapsulation



Chapter I: roadmap

I.1 what *is* the Internet?

I.2 network edge

- end systems, access networks, links

I.3 network core

- packet switching, circuit switching, network structure

I.4 delay, loss, throughput in networks

I.5 protocol layers, service models

I.6 networks under attack: security

I.7 history

Network security

- ❖ field of network security:
 - how bad guys can attack computer networks
 - how we can defend networks against attacks
 - how to design architectures that are immune to attacks
- ❖ Internet not originally designed with (much) security in mind
 - *original vision:* “a group of mutually trusting users attached to a transparent network” ☺
 - Internet protocol designers playing “catch-up”
 - security considerations in all layers!

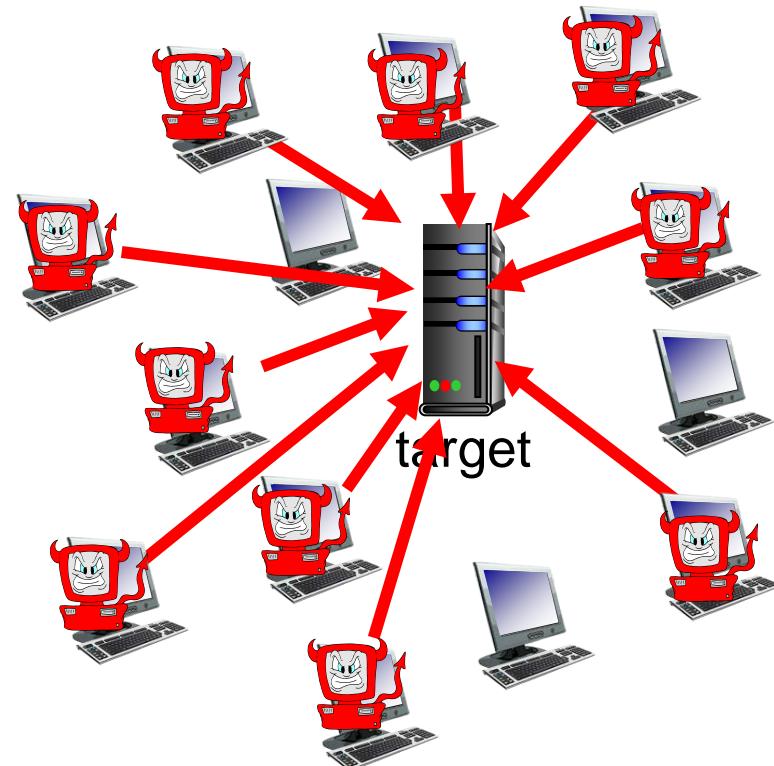
Bad guys: put malware into hosts via Internet

- ❖ malware can get in host from:
 - **virus**: self-replicating infection by receiving/executing object (e.g., e-mail attachment)
 - **worm**: self-replicating infection by passively receiving object that gets itself executed
- ❖ **spyware malware** can record keystrokes, web sites visited, upload info to collection site
- ❖ infected host can be enrolled in **botnet**, used for spam. DDoS attacks

Bad guys: attack server, network infrastructure

Denial of Service (DoS): attackers make resources (server, bandwidth) unavailable to legitimate traffic by overwhelming resource with bogus traffic

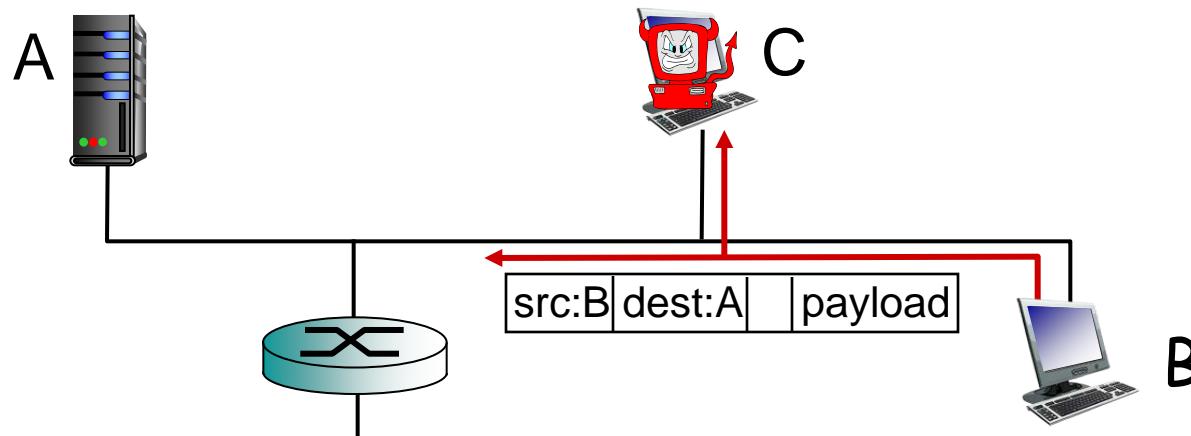
1. select target
2. break into hosts around the network (see botnet)
3. send packets to target from compromised hosts



Bad guys can sniff packets

packet “sniffing”:

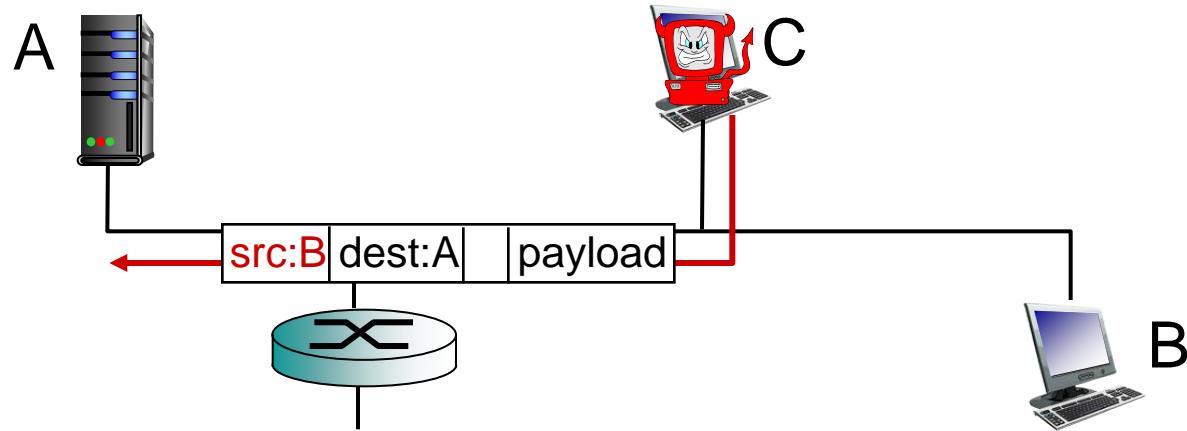
- broadcast media (shared ethernet, wireless)
- promiscuous network interface reads/records all packets (e.g., including passwords!) passing by



- ❖ wireshark software used for end-of-chapter labs is a (free) packet-sniffer

Bad guys can use fake addresses

IP spoofing: send packet with false source address



... lots more on security (throughout, Chapter 8)

Chapter I: roadmap

I.1 what *is* the Internet?

I.2 network edge

- end systems, access networks, links

I.3 network core

- packet switching, circuit switching, network structure

I.4 delay, loss, throughput in networks

I.5 protocol layers, service models

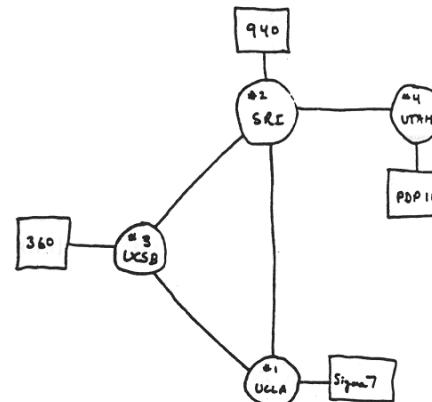
I.6 networks under attack: security

I.7 history

Internet history

1961-1972: Early packet-switching principles

- ❖ 1961: Kleinrock - queueing theory shows effectiveness of packet-switching
- ❖ 1964: Baran - packet-switching in military nets
- ❖ 1967: ARPAnet conceived by Advanced Research Projects Agency
- ❖ 1969: first ARPAnet node operational
- ❖ 1972:
 - ARPAnet public demo
 - NCP (Network Control Protocol) first host-host protocol
 - first e-mail program
 - ARPAnet has 15 nodes



Internet history

1972-1980: Internetworking, new and proprietary nets

- ❖ 1970: ALOHAnet satellite network in Hawaii
- ❖ 1974: Cerf and Kahn - architecture for interconnecting networks
- ❖ 1976: Ethernet at Xerox PARC
- ❖ late 70' s: proprietary architectures: DECnet, SNA, XNA
- ❖ late 70' s: switching fixed length packets (ATM precursor)
- ❖ 1979: ARPAnet has 200 nodes

Cerf and Kahn's
internetworking principles:

- minimalism, autonomy - no internal changes required to interconnect networks
- best effort service model
- stateless routers
- decentralized control

define today's Internet architecture

Internet history

1980-1990: new protocols, a proliferation of networks

- ❖ 1983: deployment of TCP/IP
- ❖ 1982: smtp e-mail protocol defined
- ❖ 1983: DNS defined for name-to-IP-address translation
- ❖ 1985: ftp protocol defined
- ❖ 1988: TCP congestion control
- ❖ new national networks: Csnet, BITnet, NSFnet, Minitel
- ❖ 100,000 hosts connected to confederation of networks

Internet history

1990, 2000's: commercialization, the Web, new apps

- ❖ early 1990's: ARPAnet decommissioned
- ❖ 1991: NSF lifts restrictions on commercial use of NSFnet (decommissioned, 1995)
- ❖ early 1990s: Web
 - hypertext [Bush 1945, Nelson 1960's]
 - HTML, HTTP: Berners-Lee
 - 1994: Mosaic, later Netscape
 - late 1990's: commercialization of the Web

late 1990's – 2000's:

- ❖ more killer apps: instant messaging, P2P file sharing
- ❖ network security to forefront
- ❖ est. 50 million host, 100 million+ users
- ❖ backbone links running at Gbps

Internet history

2005-present

- ❖ ~750 million hosts
 - Smartphones and tablets
- ❖ Aggressive deployment of broadband access
- ❖ Increasing ubiquity of high-speed wireless access
- ❖ Emergence of online social networks:
 - Facebook: soon one billion users
- ❖ Service providers (Google, Microsoft) create their own networks
 - Bypass Internet, providing “instantaneous” access to search, email, etc.
- ❖ E-commerce, universities, enterprises running their services in “cloud” (eg, Amazon EC2)

Introduction: summary

covered a “ton” of material!

- ❖ Internet overview
- ❖ what’s a protocol?
- ❖ network edge, core, access network
 - packet-switching versus circuit-switching
 - Internet structure
- ❖ performance: loss, delay, throughput
- ❖ layering, service models
- ❖ security
- ❖ history

you now have:

- ❖ context, overview, “feel” of networking
- ❖ more depth, detail to follow!

Chapter 2

Application Layer

© *The materials in these notes are adapted from Computer Networking: A Top Down Approach, 6th edition, by Jim Kurose, Keith Ross*

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Chapter 2: application layer

our goals:

- ❖ conceptual, implementation aspects of network application protocols
 - transport-layer service models
 - client-server paradigm
 - peer-to-peer paradigm
- ❖ learn about protocols by examining popular application-level protocols
 - HTTP
 - FTP
 - SMTP / POP3 / IMAP
 - DNS
- ❖ creating network applications
 - socket API

Some network apps

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

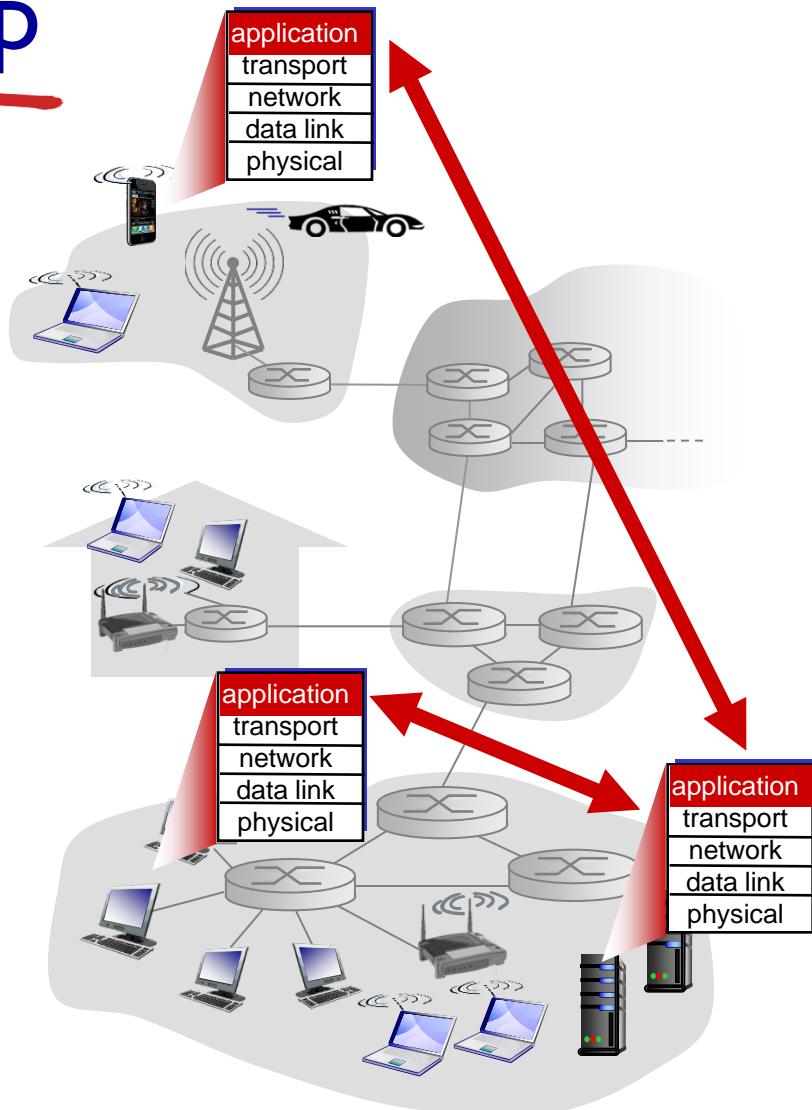
Creating a network app

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

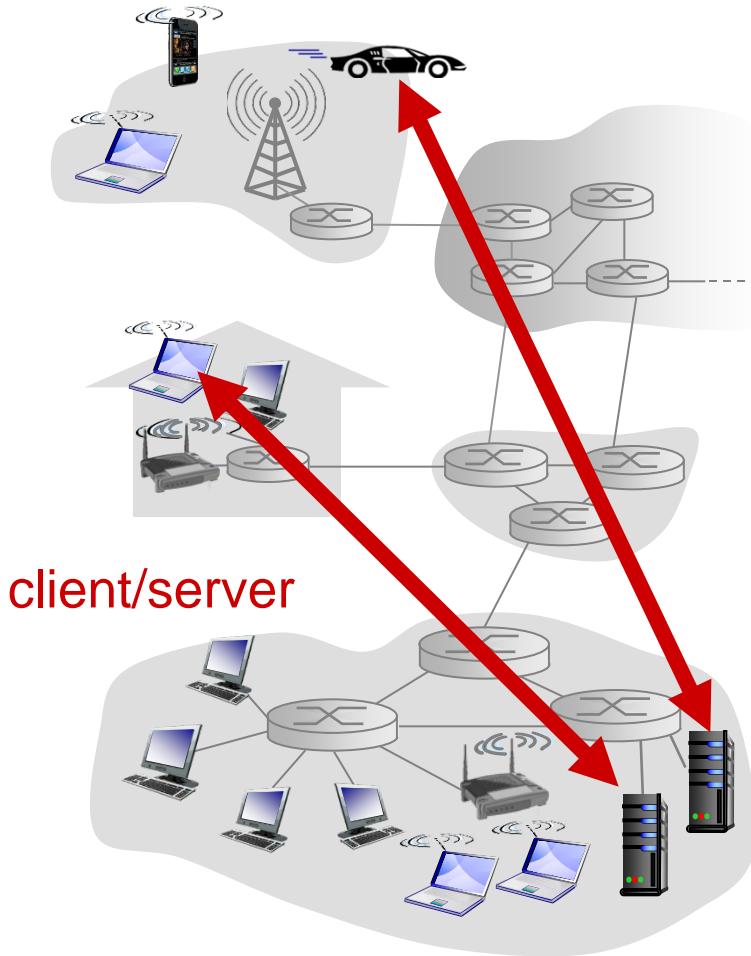


Application architectures

possible structure of applications:

- ❖ client-server
- ❖ peer-to-peer (P2P)

Client-server architecture



server:

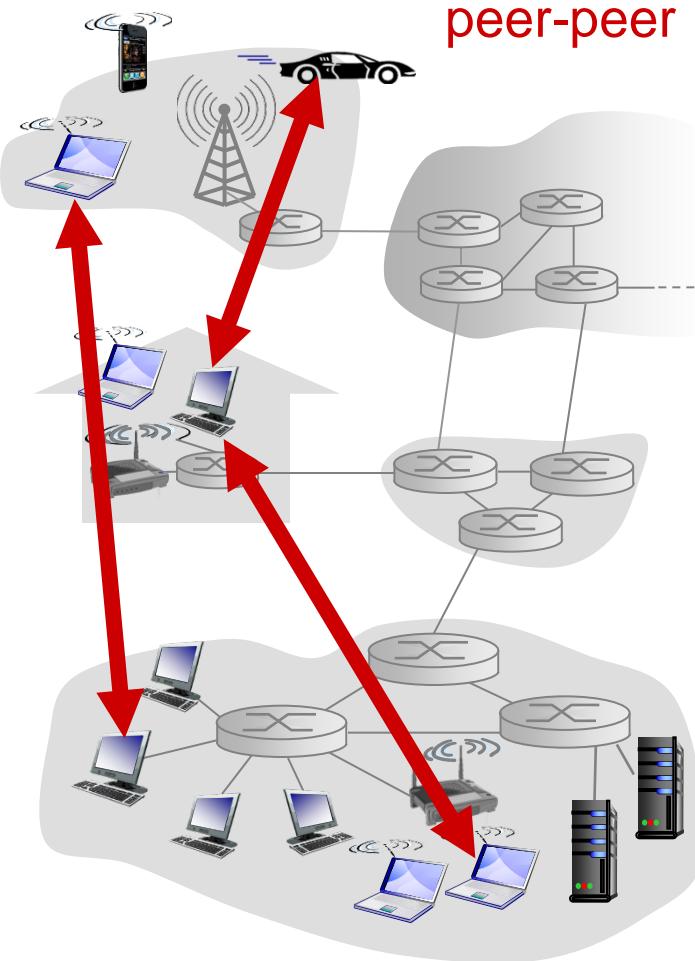
- ❖ always-on host
- ❖ permanent IP address
- ❖ data centers for scaling

clients:

- ❖ communicate with server
- ❖ may be intermittently connected
- ❖ may have dynamic IP addresses
- ❖ do not communicate directly with each other

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, as well as new service demands
- ❖ peers are intermittently connected and change IP addresses
 - complex management



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

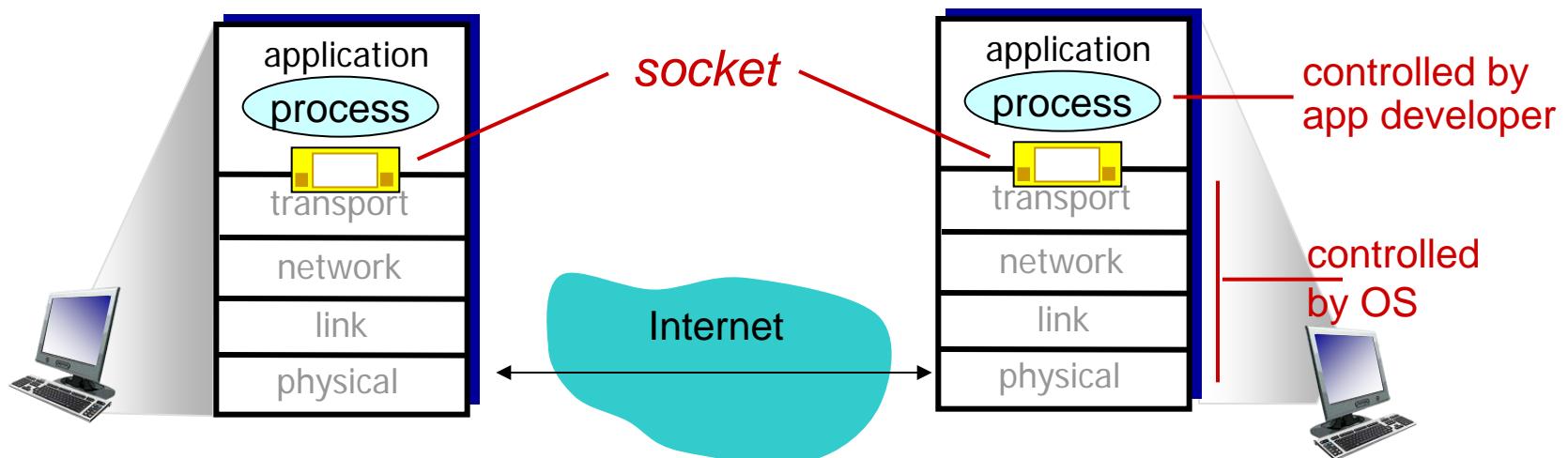
client process: process that initiates communication

server process: process that waits to be contacted

- ❖ aside: 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



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
- ❖ more shortly...

App-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
- ❖ 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

timing

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

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

security

- ❖ encryption, data integrity,

...

Transport service requirements: common apps

application	data loss	throughput	time sensitive
file transfer	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, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100'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 apps: application, transport protocols

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

Securing TCP

TCP & UDP

- ❖ no encryption
- ❖ clear-text passwds sent into socket traverse Internet in clear text

SSL

- ❖ provides encrypted TCP connection
- ❖ data integrity
- ❖ end-point authentication

SSL is at app layer

- ❖ Apps use SSL libraries, which “talk” to TCP

SSL socket API

- ❖ clear-text passwds sent into socket traverse Internet encrypted
- ❖ See Chapter 8

Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Web and HTTP

First, a review...

- ❖ *web page* consists of *objects*
- ❖ object can be HTML file, JPEG image, Java applet, audio file,...
- ❖ web page consists of *base HTML-file* which includes *several referenced objects*
- ❖ each object is addressable by a *URL*, e.g.,

www.someschool.edu/someDept/pic.gif

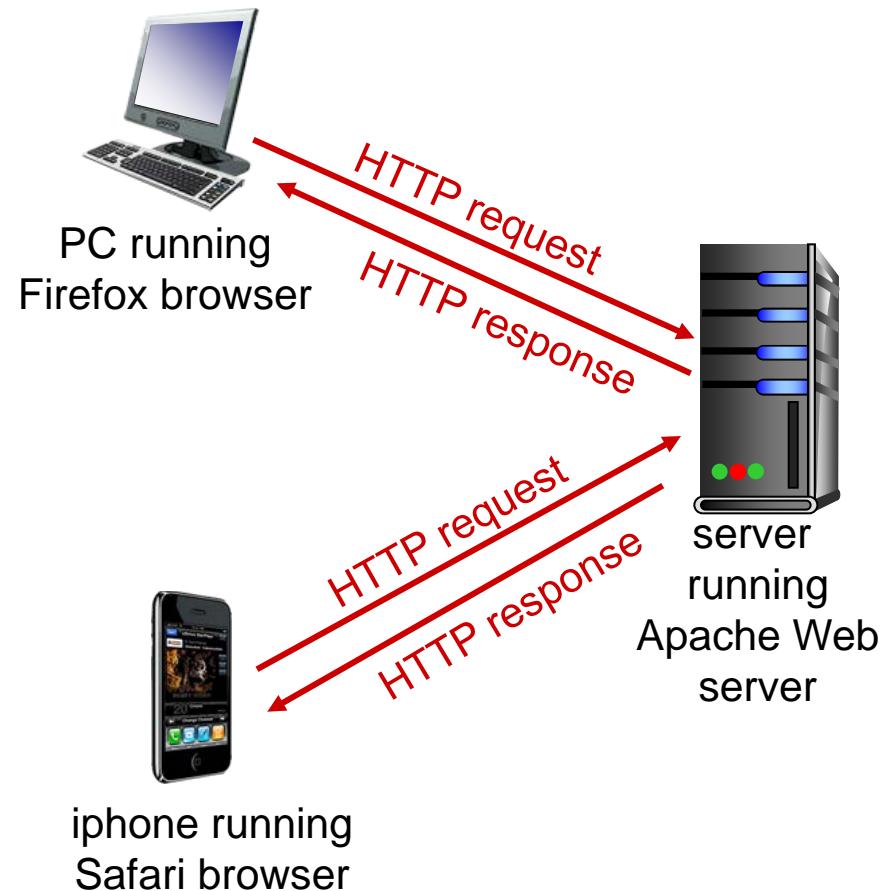
host name

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)

uses TCP:

- ❖ client initiates TCP connection (creates socket) to server, port 80
- ❖ 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

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

aside

HTTP connections

non-persistent HTTP

- ❖ at most one object sent over TCP connection
 - connection then closed
- ❖ downloading multiple objects required multiple connections

persistent HTTP

- ❖ multiple objects can be sent over single TCP connection between client, server

Non-persistent HTTP

suppose user enters URL:

`www.someSchool.edu/someDepartment/home.index`

(contains text,
references to 10
jpeg images)

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

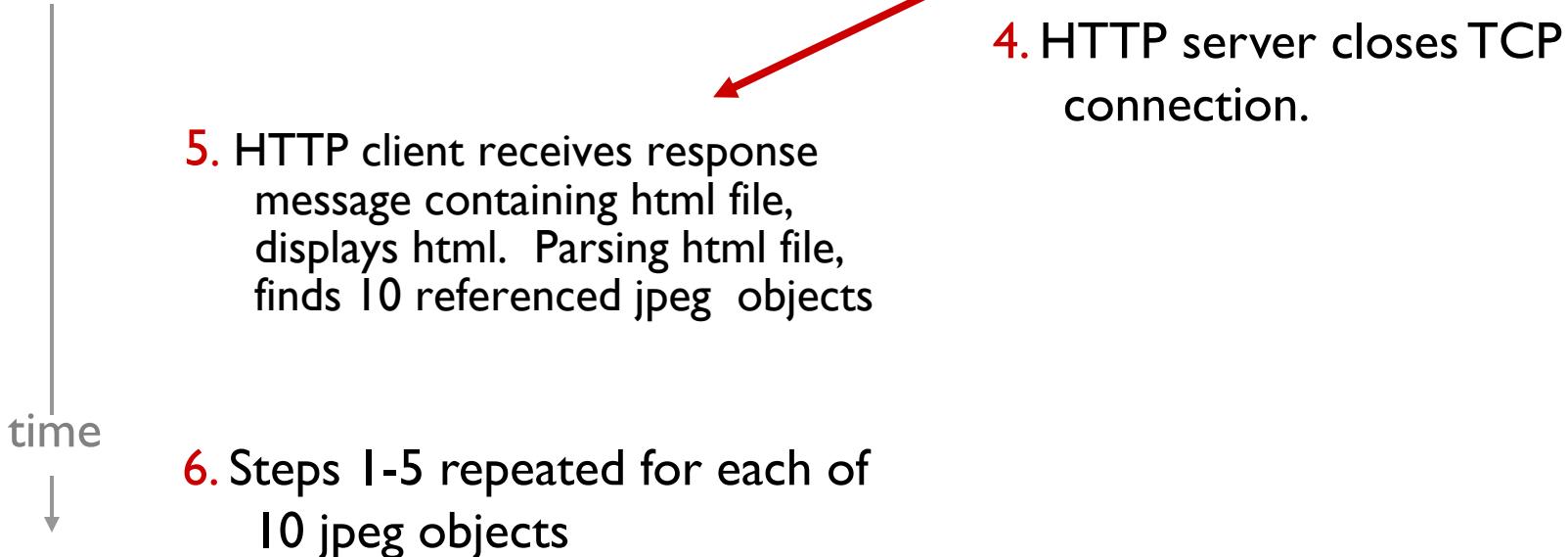
Ib. HTTP server at host `www.someSchool.edu` waiting for TCP connection at port 80. “accepts” connection, notifying client

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

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

time
↓

Non-persistent HTTP (cont.)

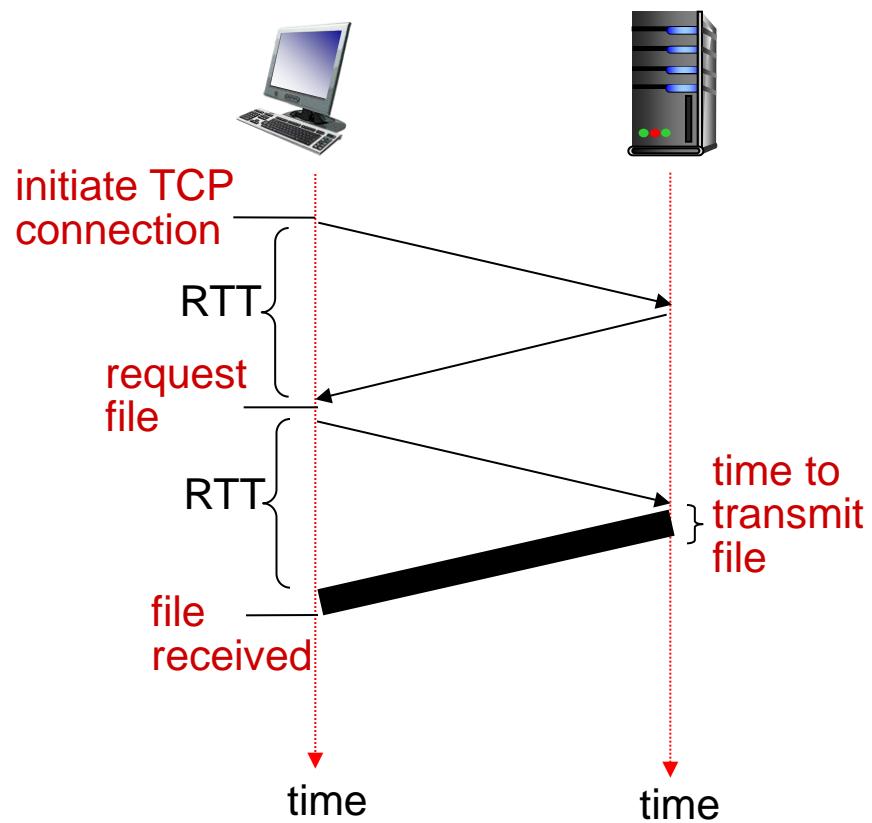


Non-persistent HTTP: response time

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

HTTP response time:

- ❖ one RTT to initiate TCP connection
- ❖ one RTT for HTTP request and first few bytes of HTTP response to return
- ❖ file transmission time
- ❖ non-persistent HTTP response time = $2\text{RTT} + \text{file transmission time}$



Persistent HTTP

non-persistent HTTP issues:

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

persistent HTTP:

- ❖ 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

HTTP request message

- ❖ two types of HTTP messages: *request, response*
- ❖ **HTTP request message:**
 - ASCII (human-readable format)

request line
(GET, POST,
HEAD commands)

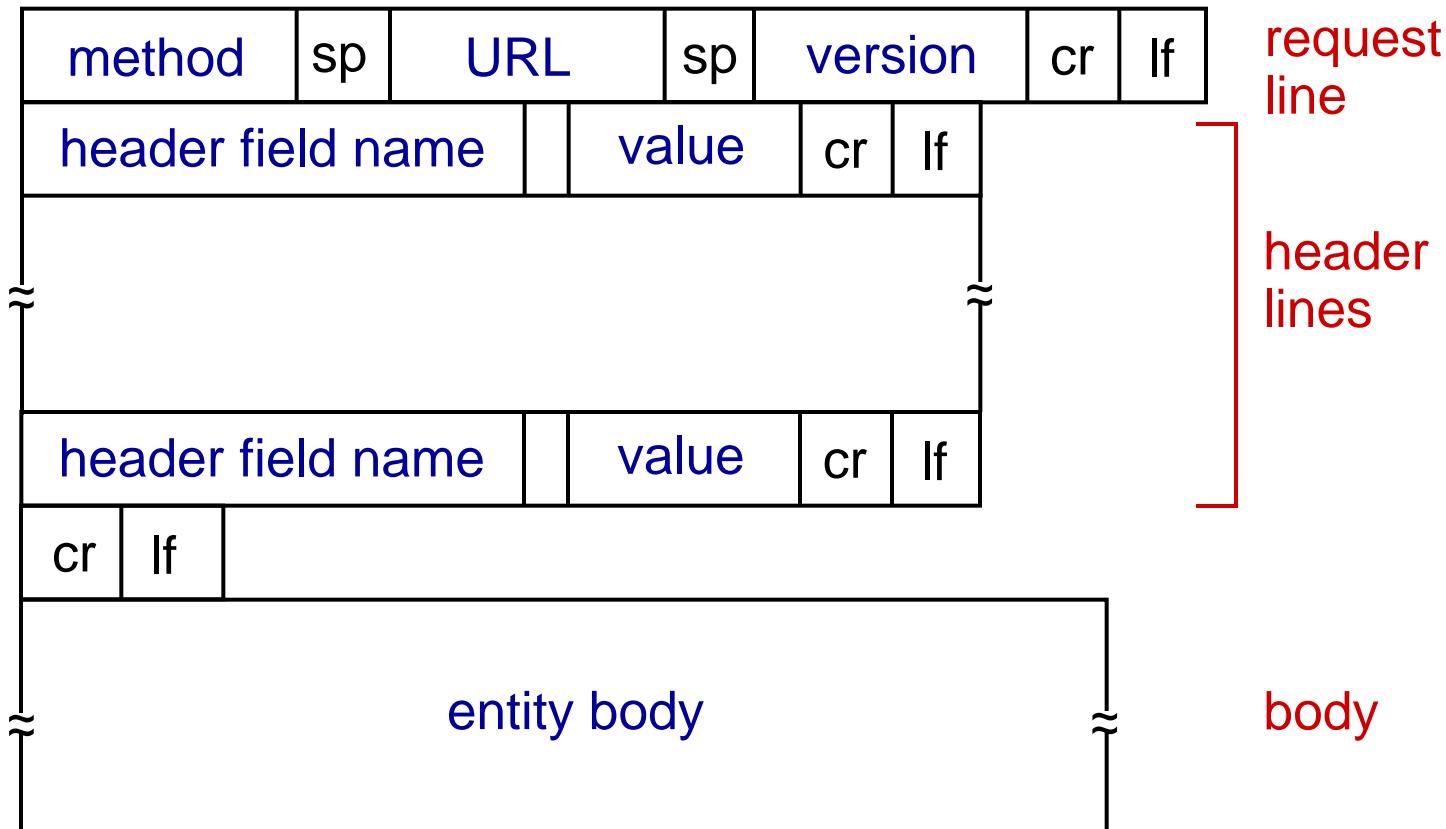
header
lines

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

```
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

HTTP request message: general format



Uploading form input

POST method:

- ❖ web page often includes form input
- ❖ input is uploaded to server in entity body

URL method:

- ❖ uses GET method
- ❖ input is uploaded in URL field of request line:

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

Method types

HTTP/1.0:

- ❖ GET
- ❖ POST
- ❖ HEAD
 - asks server to leave requested object out of response

HTTP/1.1:

- ❖ GET, POST, HEAD
- ❖ PUT
 - uploads file in entity body to path specified in URL field
- ❖ DELETE
 - deletes file specified in the URL field

HTTP response message

status line

(protocol

status code

status phrase)

header
lines

data, e.g.,
requested
HTML file

```
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\nETag: "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\n
```

```
data 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 msg

301 Moved Permanently

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

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

I. Telnet to your favorite Web server:

telnet H

opens TCP connection to port 80
(default HTTP server port) at cis.poly.edu.
anything typed in sent
to port 80 at cis.poly.edu

2. type in a GET HTTP request:

**GET /~ross/ HTTP/1.1
Host: cis.poly.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)

User-server state: cookies

many Web sites use cookies

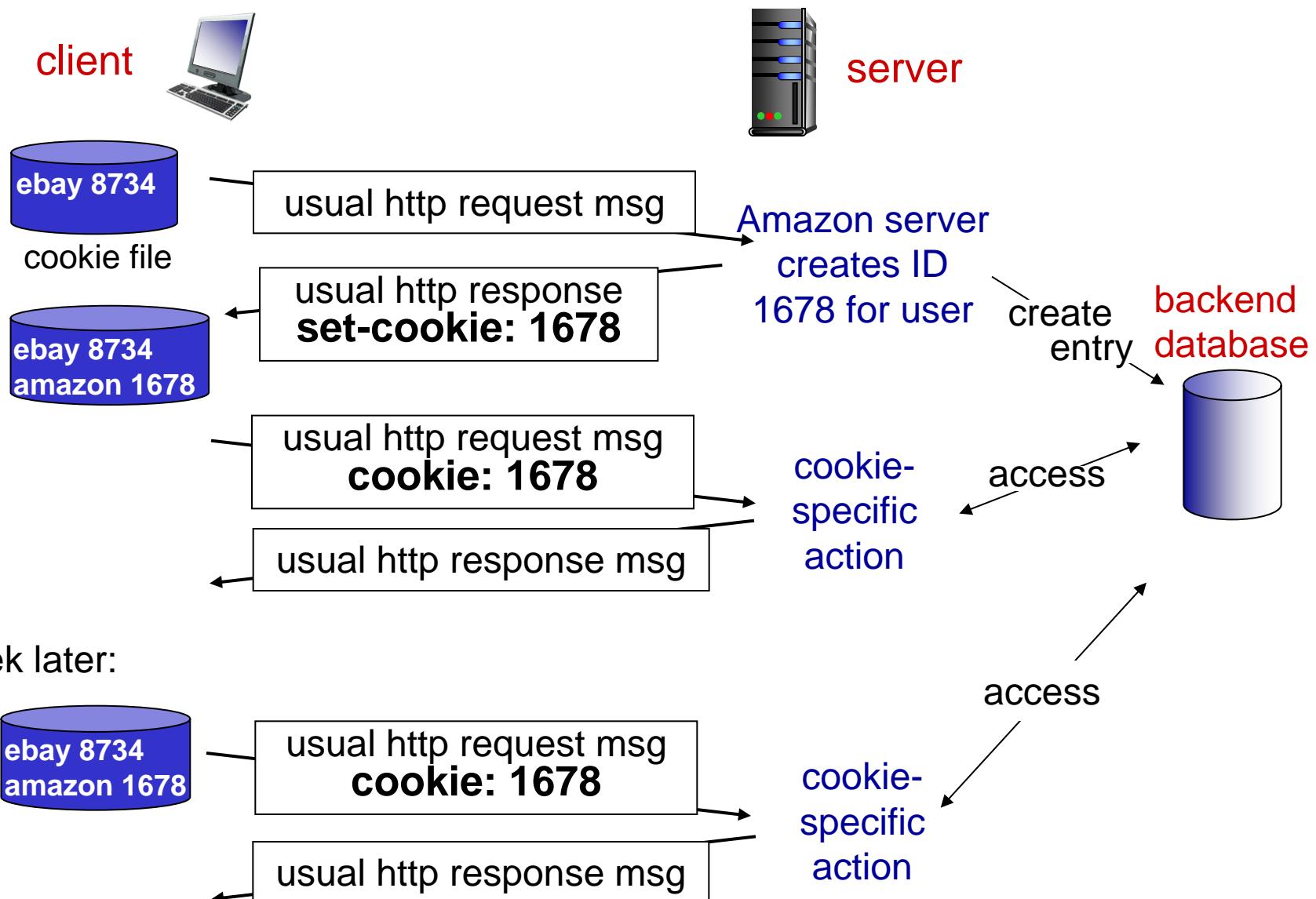
four components:

- 1) cookie header line of
HTTP *response*
message
- 2) cookie header line in
next HTTP *request*
message
- 3) cookie file kept on
user's host, managed
by user's browser
- 4) back-end database at
Web site

example:

- ❖ Susan always access Internet from PC
- ❖ visits specific e-commerce site for first time
- ❖ when initial HTTP requests arrives at site, site creates:
 - unique ID
 - entry in backend database for ID

Cookies: keeping “state” (cont.)



Cookies (continued)

*what cookies can be used
for:*

- ❖ authorization
- ❖ shopping carts
- ❖ recommendations
- ❖ user session state (Web e-mail)

*aside
cookies and privacy:*

- ❖ cookies permit sites to learn a lot about you
- ❖ you may supply name and e-mail to sites

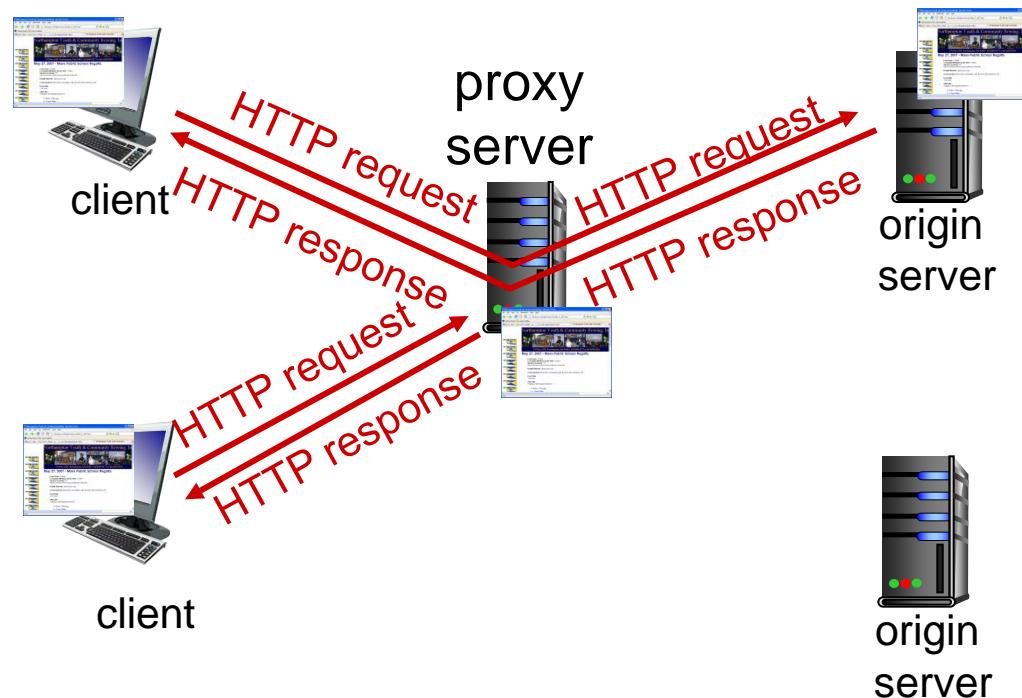
how to keep “state”:

- ❖ protocol endpoints: maintain state at sender/receiver over multiple transactions
- ❖ cookies: http messages carry state

Web caches (proxy server)

goal: satisfy client request without involving origin server

- ❖ user sets browser: Web accesses via cache
- ❖ browser sends all HTTP requests to cache
 - object in cache: cache returns object
 - else cache requests object from origin server, then returns object to client



More about Web caching

- ❖ cache acts as both client and server
 - server for original requesting client
 - client to origin server
 - ❖ typically cache is installed by ISP (university, company, residential ISP)
- why Web caching?*
- ❖ reduce response time for client request
 - ❖ reduce traffic on an institution's access link
 - ❖ Internet dense with caches: enables “poor” content providers to effectively deliver content (so too does P2P file sharing)

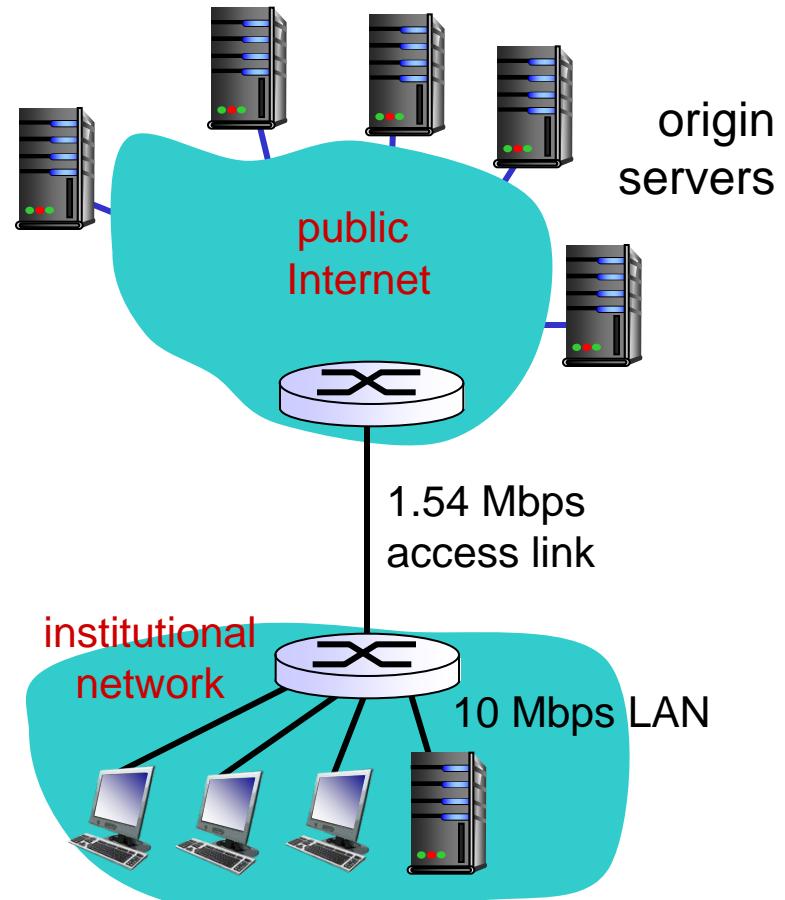
Caching example:

assumptions:

- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

consequences:

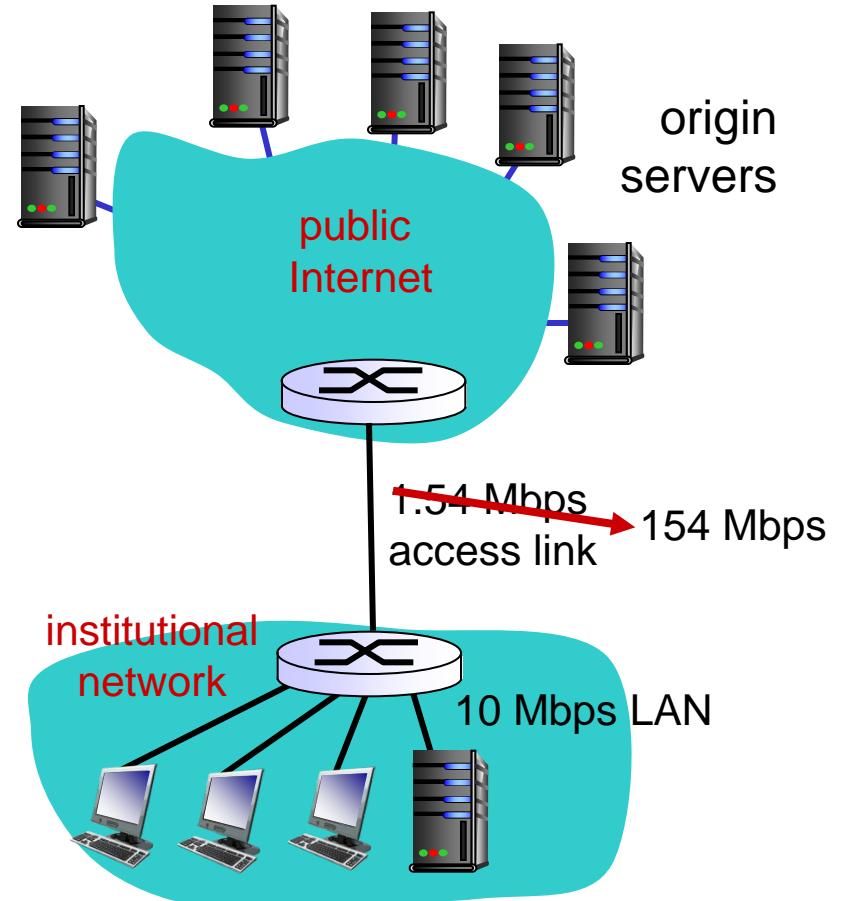
- ❖ LAN utilization: 15% *problem!*
- ❖ access link utilization = **99%**
- ❖ total delay = Internet delay + access delay + LAN delay
= 2 sec + minutes + usecs



Caching example: fatter access link

assumptions:

- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: ~~1.54 Mbps~~ \rightarrow 154 Mbps



consequences:

- ❖ LAN utilization: 15%
- ❖ access link utilization = ~~99%~~ \rightarrow 0.99%
- ❖ total delay = Internet delay + access delay + LAN delay
 $= 2 \text{ sec} + \cancel{\text{minutes}} + \cancel{\text{usecs}} \rightarrow \text{msecs}$

Cost: increased access link speed (not cheap!)

Caching example: install local cache

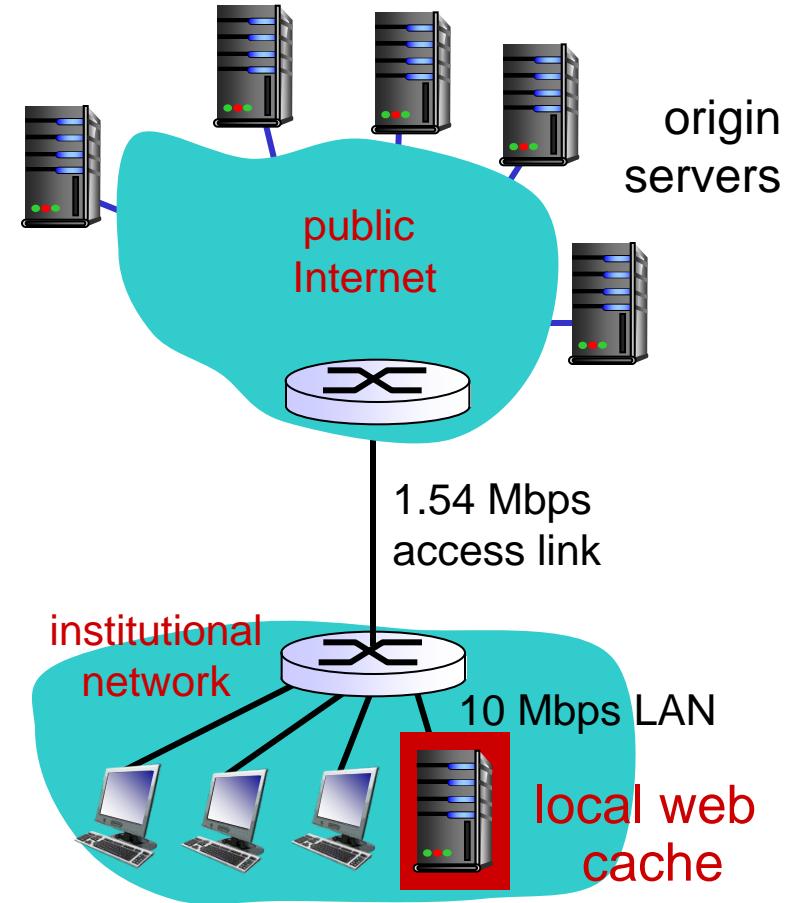
assumptions:

- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

consequences:

- ❖ LAN utilization: 15%
- ❖ access link utilization = ?
- ❖ total delay = ?

How to compute link utilization, delay?

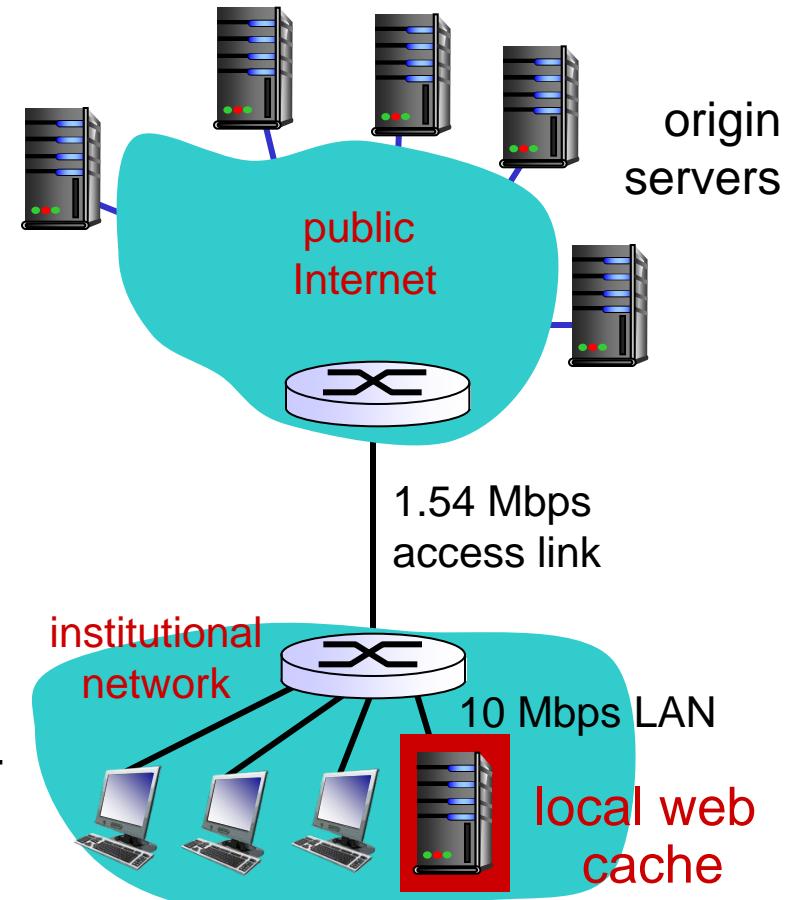


Cost: web cache (cheap!)

Caching example: install local cache

Calculating access link utilization, delay with cache:

- ❖ suppose cache hit rate is 0.4
 - 40% requests satisfied at cache,
60% requests satisfied at origin
- ❖ access link utilization:
 - 60% of requests use access link
- ❖ data rate to browsers over access link
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
 - utilization = $0.9 / 1.54 = .58$
- ❖ total delay
 - $= 0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
 - $= 0.6 (2.01) + 0.4 (0.01)$
 - $= \sim 1.2 \text{ secs}$
 - less than with 1.54 Mbps link (and cheaper too!)



Conditional GET

- ❖ **Goal:** don't send object if cache has up-to-date cached version

- no object transmission delay

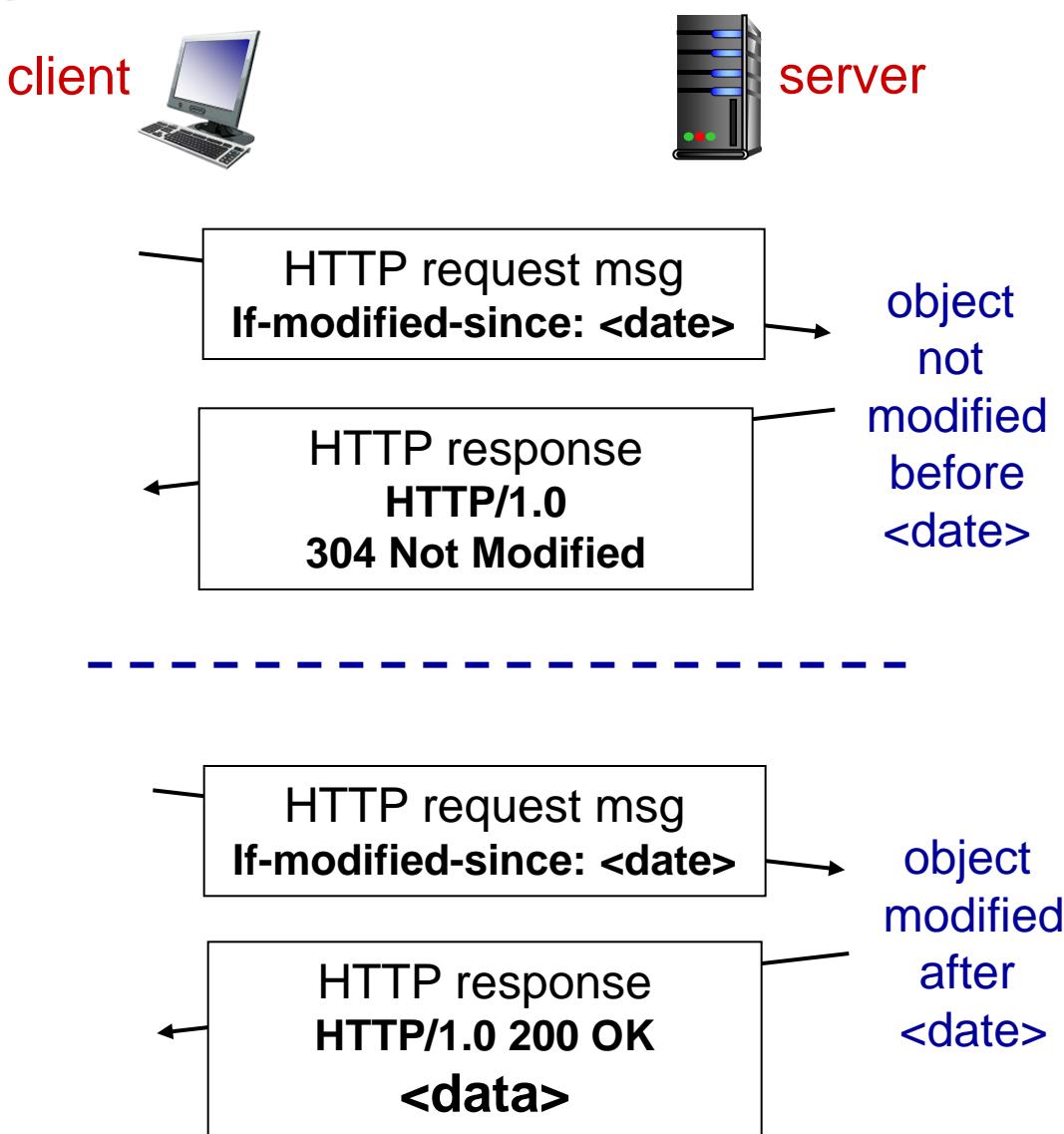
- lower link utilization

- ❖ **cache:** specify date of cached copy in HTTP request

If-modified-since:
<date>

- ❖ **server:** response contains no object if cached copy is up-to-date:

HTTP/1.0 304 Not Modified



Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

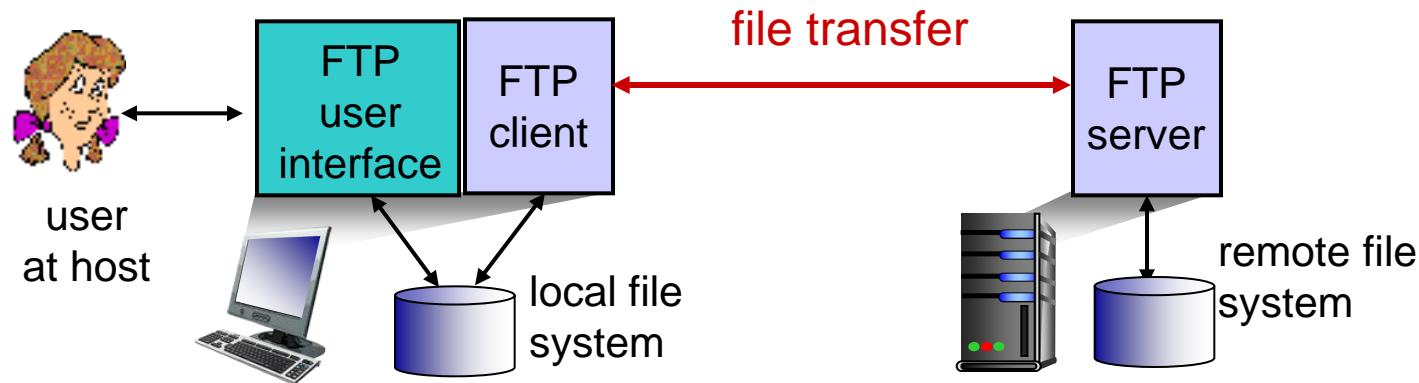
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

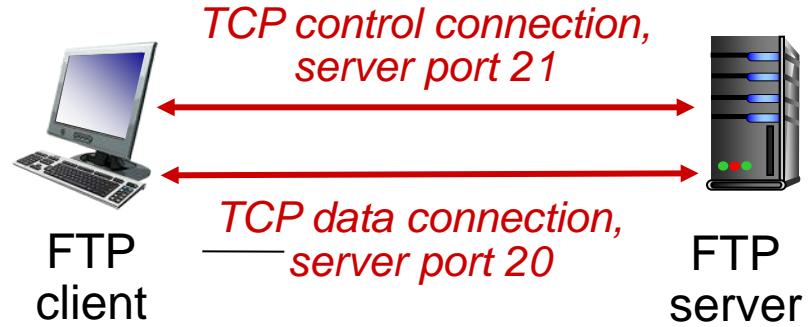
FTP: the file transfer protocol



- ❖ transfer file to/from remote host
- ❖ client/server model
 - *client*: side that initiates transfer (either to/from remote)
 - *server*: remote host
- ❖ ftp: RFC 959
- ❖ ftp server: port 21

FTP: separate control, data connections

- ❖ FTP client contacts FTP server at port 21, using TCP
- ❖ client authorized over control connection
- ❖ client browses remote directory, sends commands over control connection
- ❖ when server receives file transfer command, *server* opens 2nd TCP data connection (for file) to client
- ❖ after transferring one file, server closes data connection



- ❖ server opens another TCP data connection to transfer another file
- ❖ control connection: “*out of band*”
- ❖ FTP server maintains “state”: current directory, earlier authentication

FTP commands, responses

sample commands:

- ❖ sent as ASCII text over control channel
- ❖ **USER *username***
- ❖ **PASS *password***
- ❖ **LIST** return list of file in current directory
- ❖ **RETR *filename*** retrieves (gets) file
- ❖ **STOR *filename*** stores (puts) file onto remote host

sample return codes

- ❖ status code and phrase (as in HTTP)
- ❖ **331 Username OK, password required**
- ❖ **125 data connection already open; transfer starting**
- ❖ **425 Can't open data connection**
- ❖ **452 Error writing file**

Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

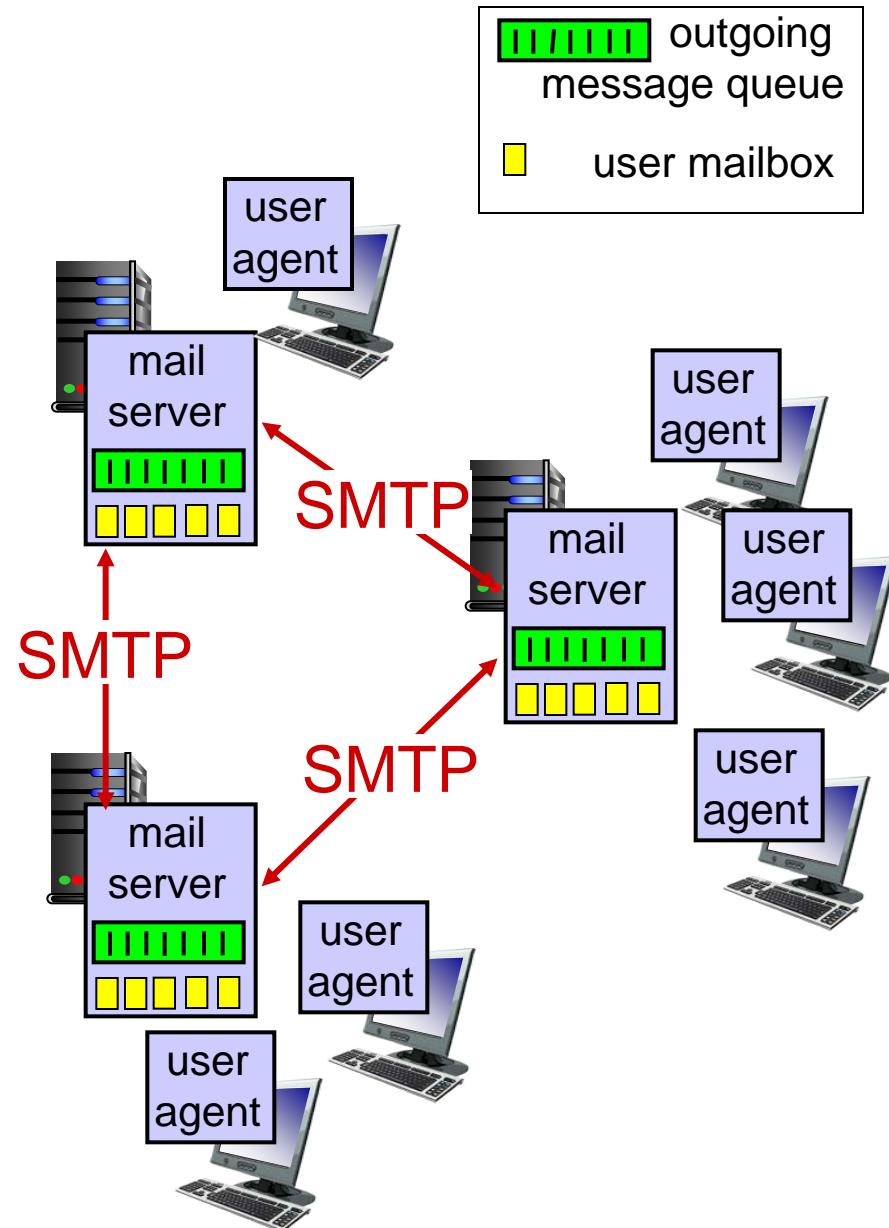
Electronic mail

Three major components:

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

User Agent

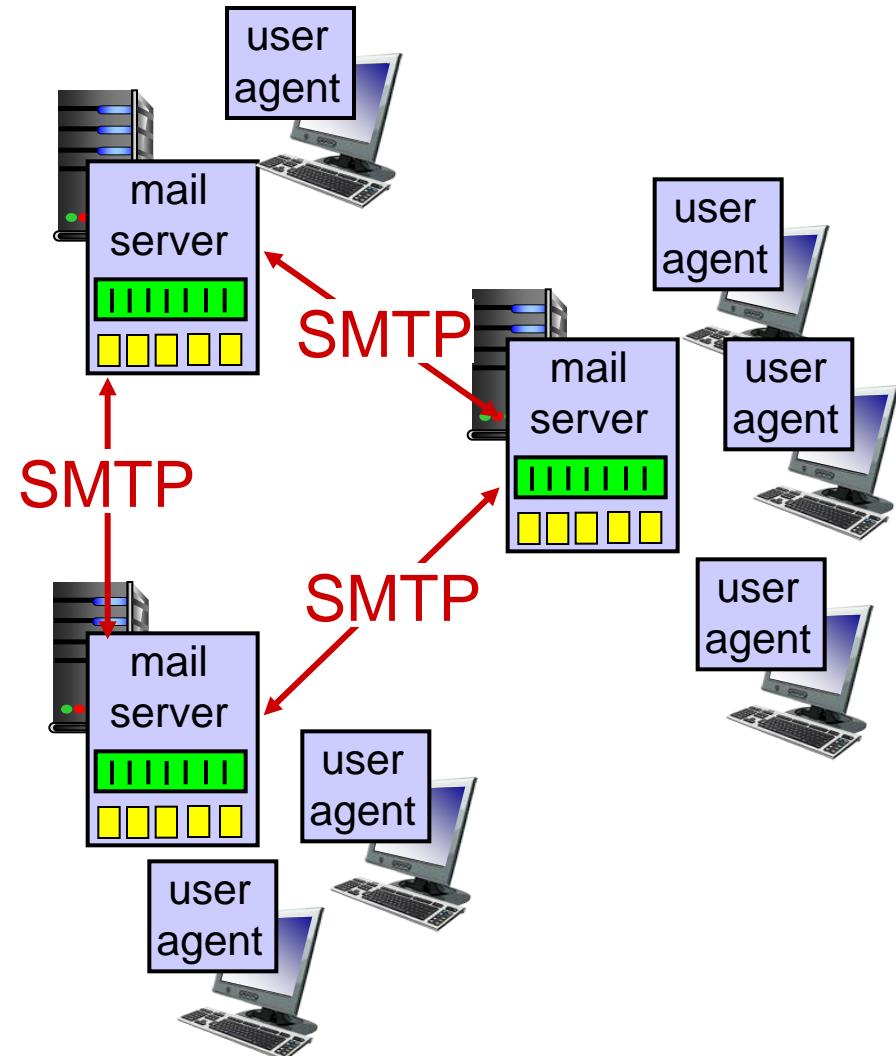
- ❖ a.k.a. “mail reader”
- ❖ composing, editing, reading mail messages
- ❖ e.g., Outlook, Thunderbird, iPhone mail client
- ❖ outgoing, incoming messages stored on server



Electronic 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

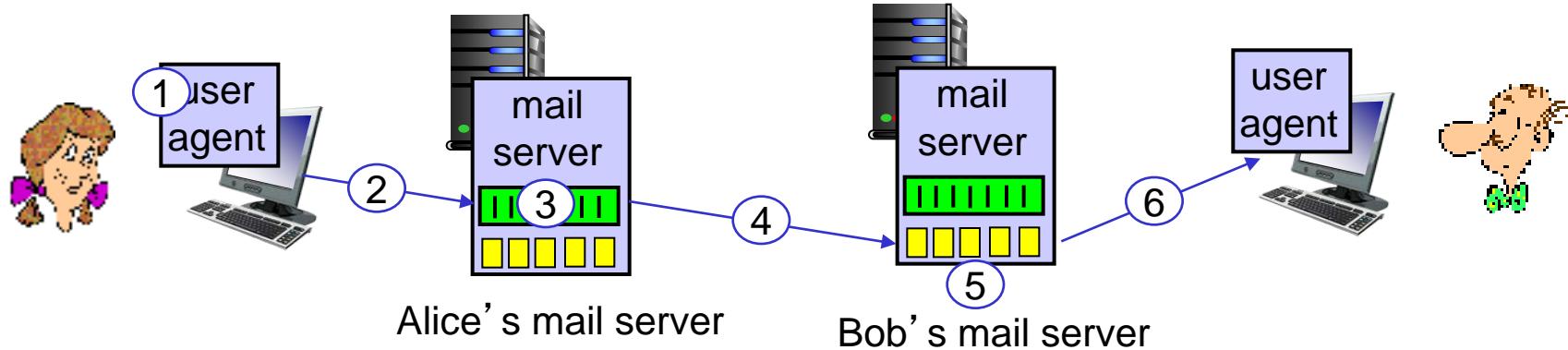


Electronic Mail: SMTP [RFC 2821]

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

Scenario: Alice sends message to Bob

- 1) Alice uses UA to compose 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
```

Try SMTP interaction for yourself:

- ❖ `telnet servername 25`
- ❖ see 220 reply from server
- ❖ enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client (reader)

SMTP: final words

- ❖ 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

comparison with HTTP:

- ❖ HTTP: pull
- ❖ SMTP: push
- ❖ both have ASCII command/response interaction, status codes
- ❖ HTTP: each object encapsulated in its own response msg
- ❖ SMTP: multiple objects sent in multipart msg

Mail message format

SMTP: protocol for
exchanging email msgs

RFC 822: standard for text
message format:

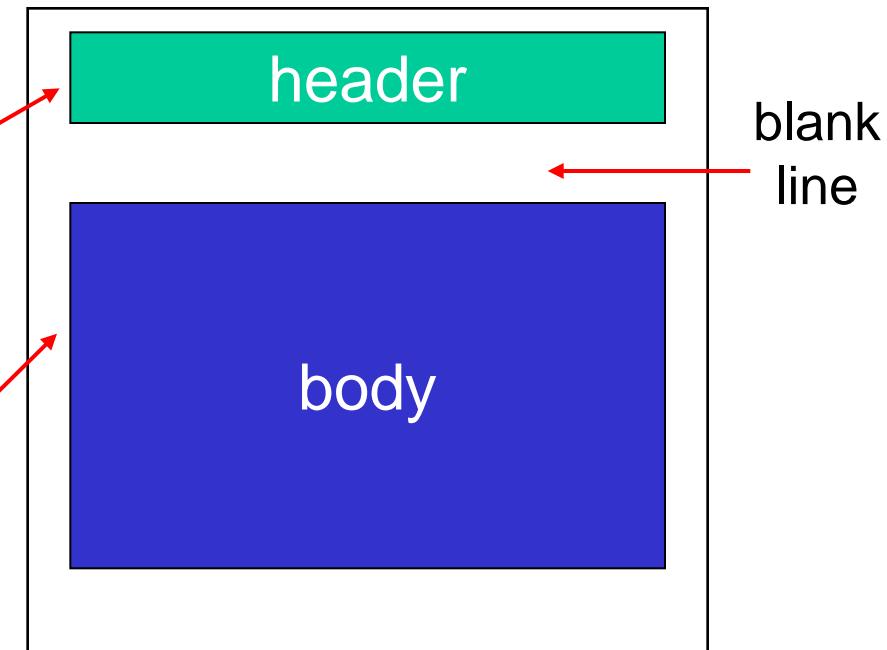
❖ header lines, e.g.,

- To:
- From:
- Subject:

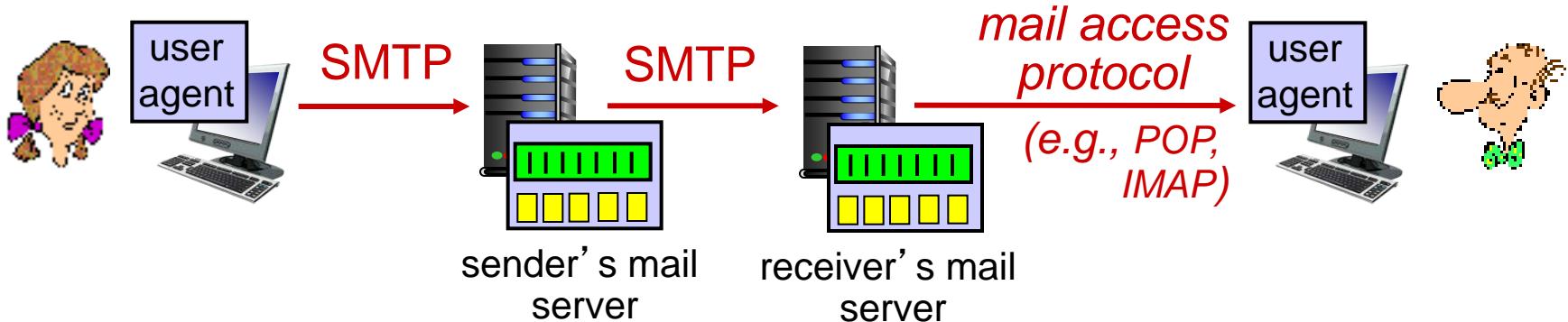
*different from SMTP MAIL
FROM, RCPT TO:
commands!*

❖ Body: the “message”

- ASCII characters only



Mail access protocols



- ❖ **SMTP:** delivery/storage to receiver's server
- ❖ mail access protocol: retrieval from server
 - **POP:** Post Office Protocol [RFC 1939]: authorization, download
 - **IMAP:** Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
 - **HTTP:** gmail, Hotmail, Yahoo! Mail, etc.

POP3 protocol

authorization phase

- ❖ client commands:
 - **user**: declare username
 - **pass**: password
- ❖ server responses
 - +OK
 - -ERR

transaction phase, client:

- ❖ **list**: list message numbers
- ❖ **retr**: retrieve message by number
- ❖ **dele**: delete
- ❖ **quit**

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 (more) and IMAP

more about POP3

- ❖ previous example uses POP3 “download and delete” mode
 - Bob cannot re-read e-mail if he changes client
- ❖ POP3 “download-and-keep”: copies of messages on different clients
- ❖ POP3 is stateless across sessions

IMAP

- ❖ keeps all messages in one place: at server
- ❖ allows user to organize messages in folders
- ❖ keeps user state across sessions:
 - names of folders and mappings between message IDs and folder name

Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 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., www.yahoo.com - 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

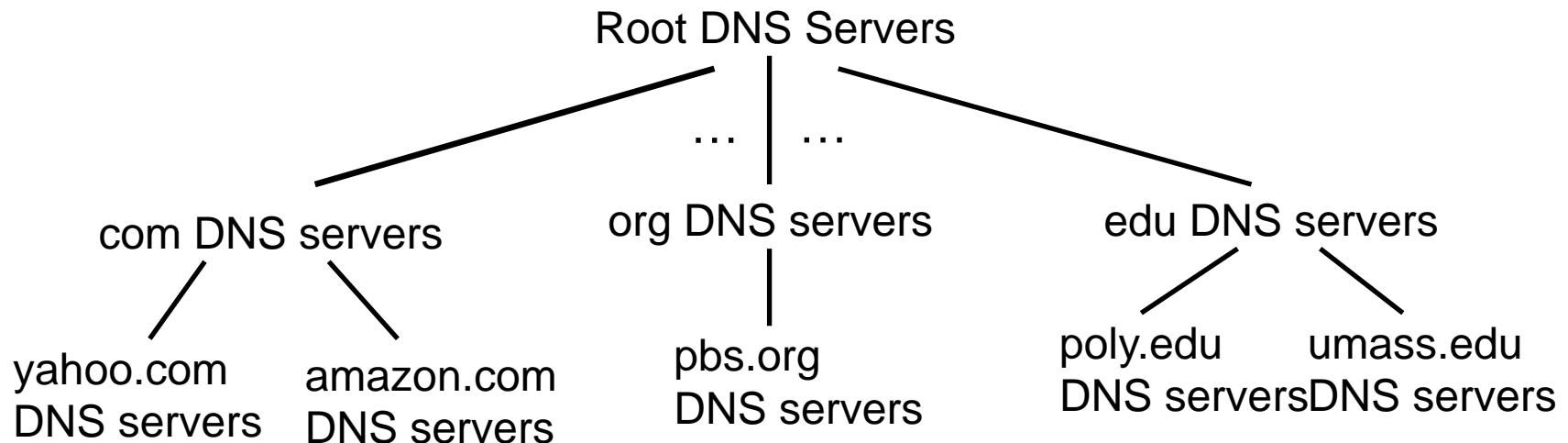
- ❖ hostname to IP address translation
- ❖ host aliasing
 - canonical, alias names
- ❖ mail server aliasing
- ❖ load distribution
 - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

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

A: *doesn't scale!*

DNS: a distributed, hierarchical database

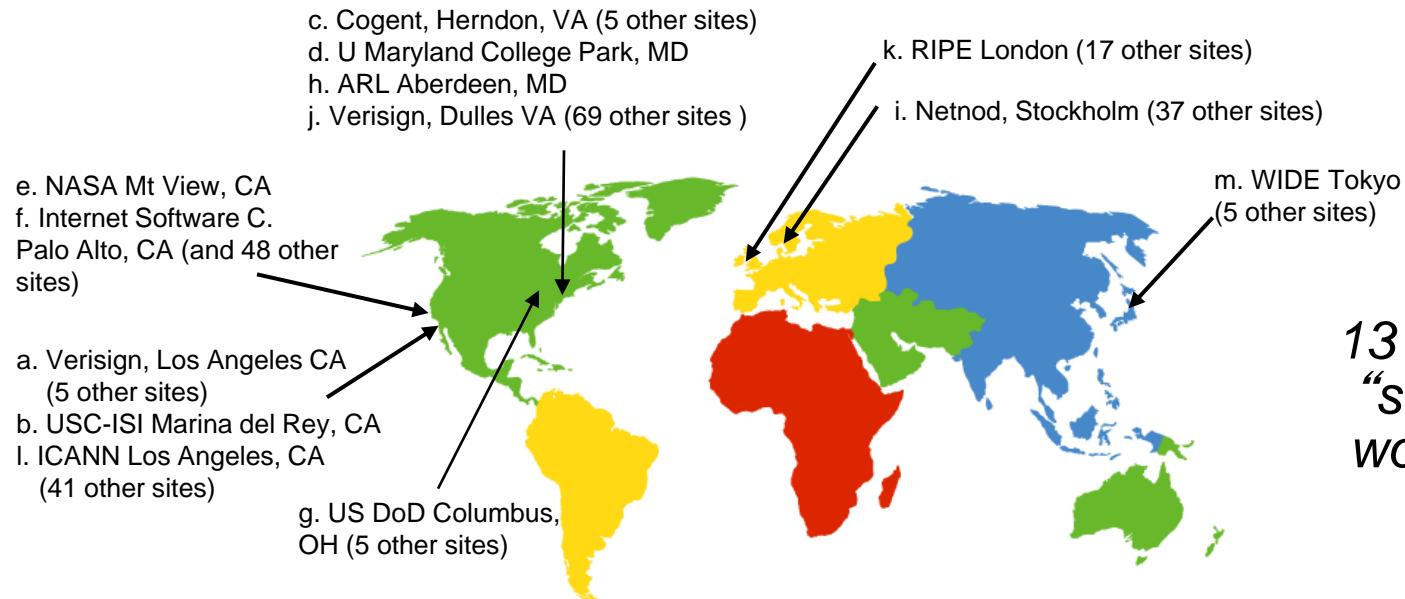


client wants IP for www.amazon.com; 1st approx:

- ❖ 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

- ❖ contacted by local name server that can not resolve name
- ❖ root name server:
 - contacts authoritative name server if name mapping not known
 - gets mapping
 - returns mapping to local name server



*13 root name
“servers”
worldwide*

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.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .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 server

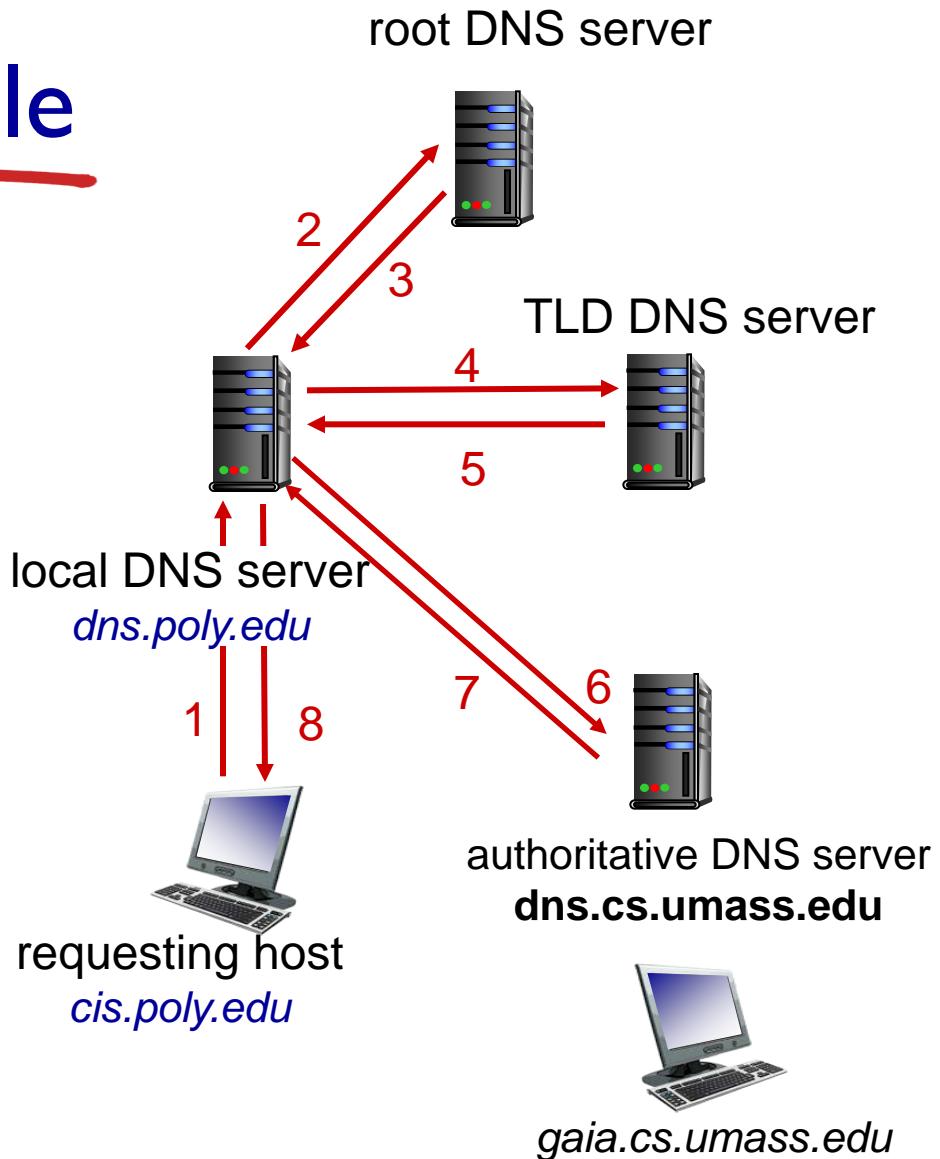
- ❖ does not strictly belong to hierarchy
- ❖ each ISP (residential ISP, company, university) has one
 - also called “default name server”
- ❖ when host makes DNS query, query is sent to its local DNS server
 - has local cache of recent name-to-address translation pairs (but may be out of date!)
 - acts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

DNS name resolution example

- host at `cis.poly.edu` wants IP address for `gaia.cs.umass.edu`

iterated query:

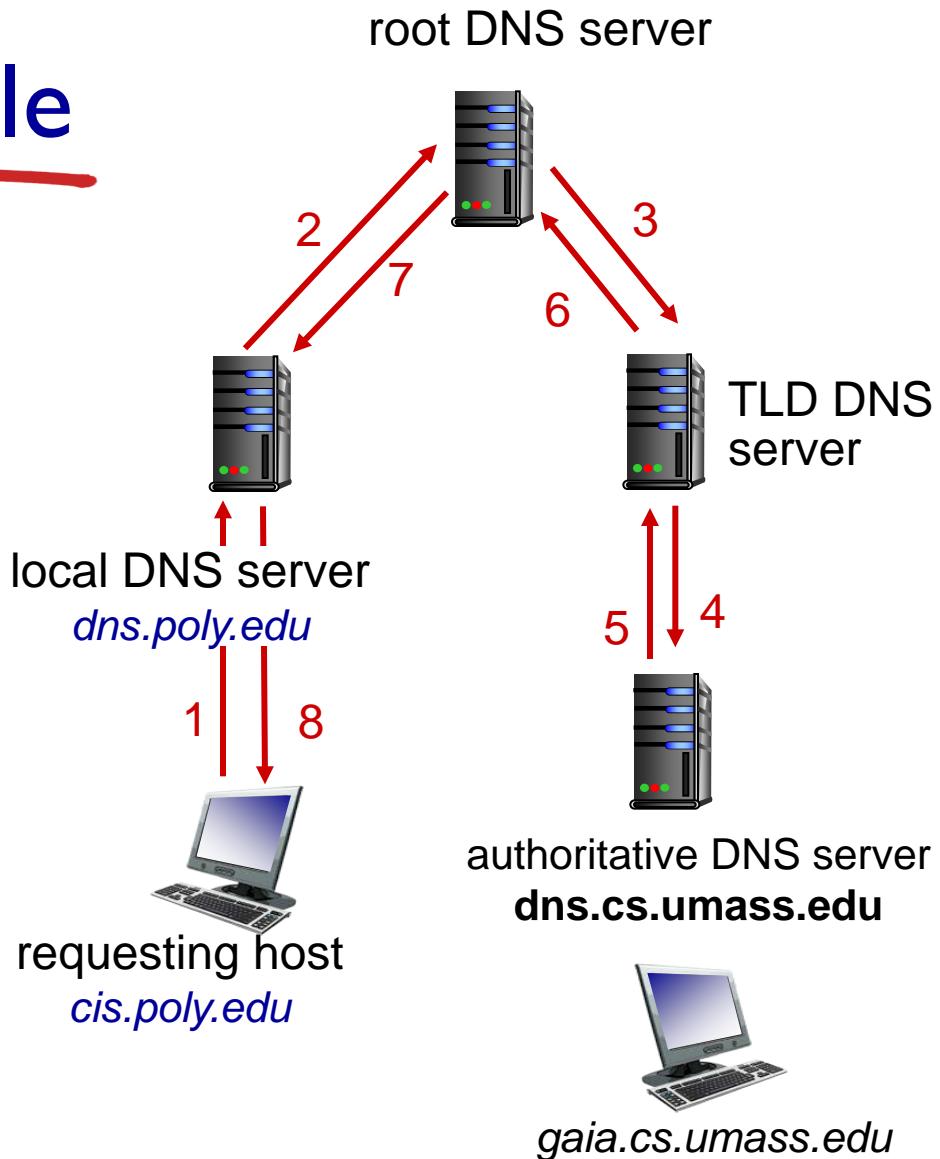
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”



DNS name resolution example

recursive query:

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



DNS: caching, updating 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 db 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

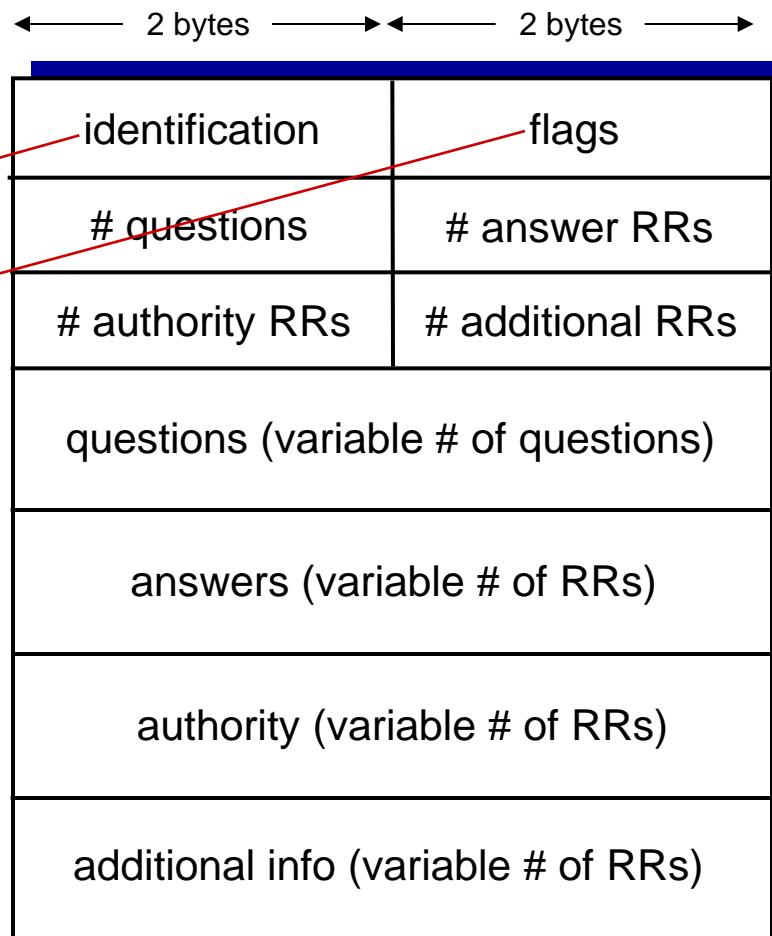
- **value** is name of mailserver associated with **name**

DNS protocol, messages

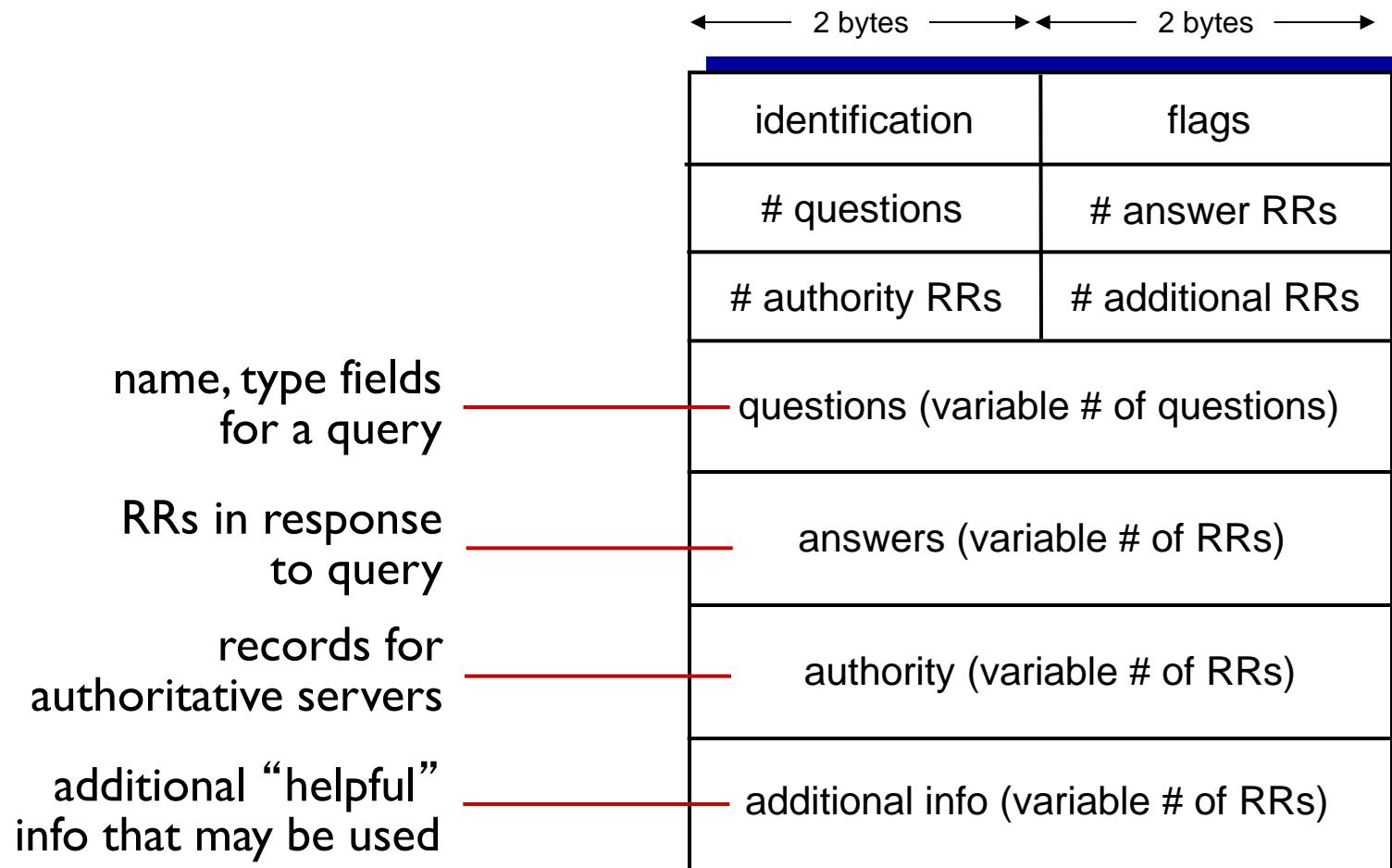
- ❖ *query* and *reply* messages, both with same *message format*

msg 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



```
$ dig redhat.com

; <>> DiG 9.7.3-RedHat-9.7.3-2.el6 <>> redhat.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 62863
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 4, ADDITIONAL: 3

;; QUESTION SECTION:
;redhat.com.           IN      A

;; ANSWER SECTION:
redhat.com.        37      IN      A      209.132.183.81

;; AUTHORITY SECTION:
redhat.com.        73      IN      NS     ns4.redhat.com.
redhat.com.        73      IN      NS     ns3.redhat.com.
redhat.com.        73      IN      NS     ns2.redhat.com.
redhat.com.        73      IN      NS     ns1.redhat.com.

;; ADDITIONAL SECTION:
ns1.redhat.com.    73      IN      A      209.132.186.218
ns2.redhat.com.    73      IN      A      209.132.183.2
ns3.redhat.com.    73      IN      A      209.132.176.100

;; Query time: 13 msec
;; SERVER: 209.144.50.138#53(209.144.50.138)
;; WHEN: Thu Jan 12 10:09:49 2012
;; MSG SIZE  rcvd: 164
```

Inserting records into DNS

- ❖ example: new startup “Network Utopia”
- ❖ register name `networkutopia.com` at *DNS registrar* (e.g., Network Solutions)
 - provide names, IP addresses of authoritative name server (primary and secondary)
 - registrar inserts two RRs into .com TLD server:
`(networkutopia.com, dns1.networkutopia.com, NS)`
`(dns1.networkutopia.com, 212.212.212.1, A)`
- ❖ create authoritative server type A record for `www.networkutopia.com`; type MX record for `networkutopia.com`

Attacking DNS

DDoS attacks

- ❖ Bombard root servers with traffic
 - Not successful to date
 - Traffic Filtering
 - Local DNS servers cache IPs of TLD servers, allowing root server bypassed
- ❖ Bombard TLD servers
 - Potentially more dangerous

Redirect attacks

- ❖ Man-in-middle
 - Intercept 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

Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

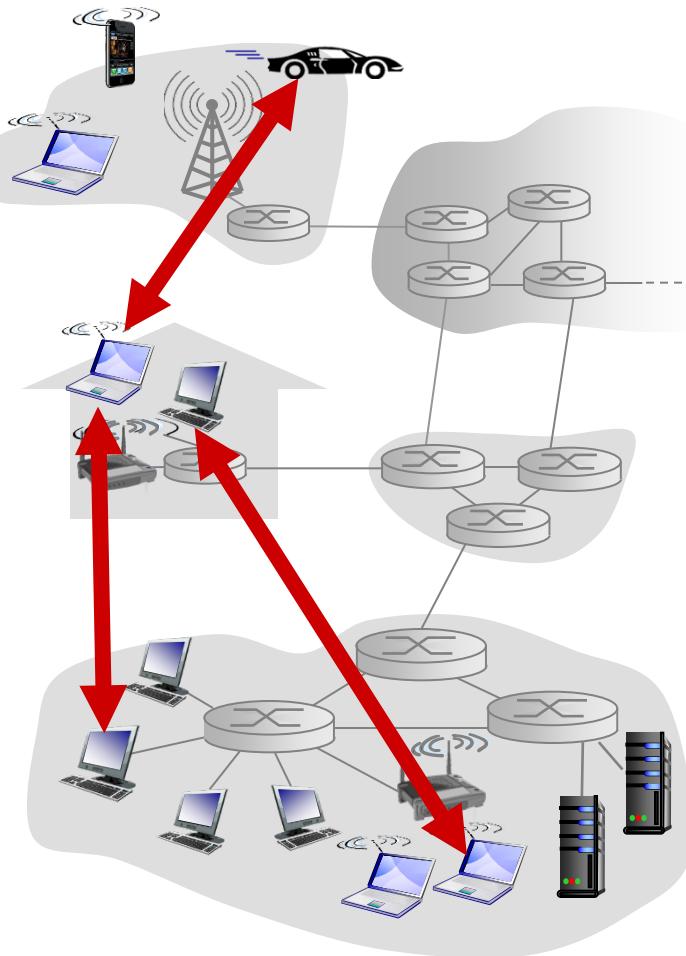
2.7 socket programming with UDP and TCP

Pure P2P architecture

- ❖ no always-on server
- ❖ arbitrary end systems directly communicate
- ❖ peers are intermittently connected and change IP addresses

examples:

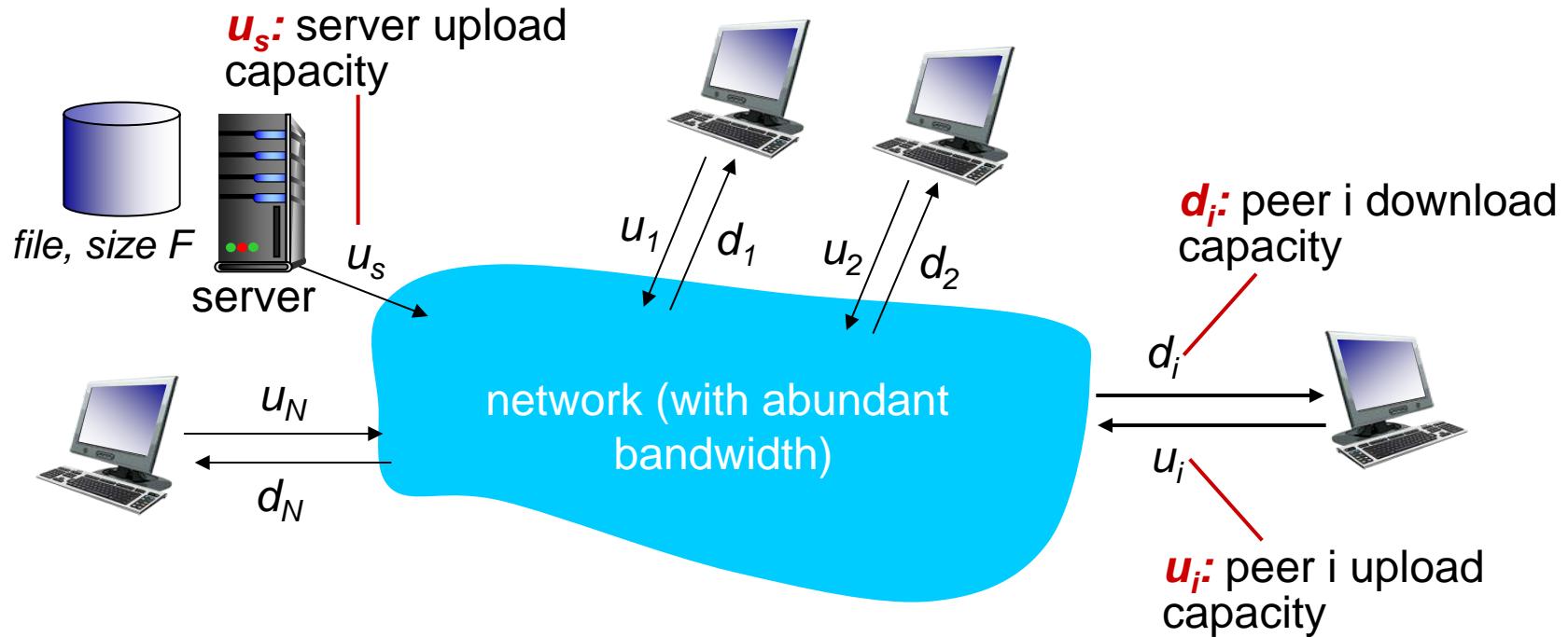
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)



File distribution: client-server vs P2P

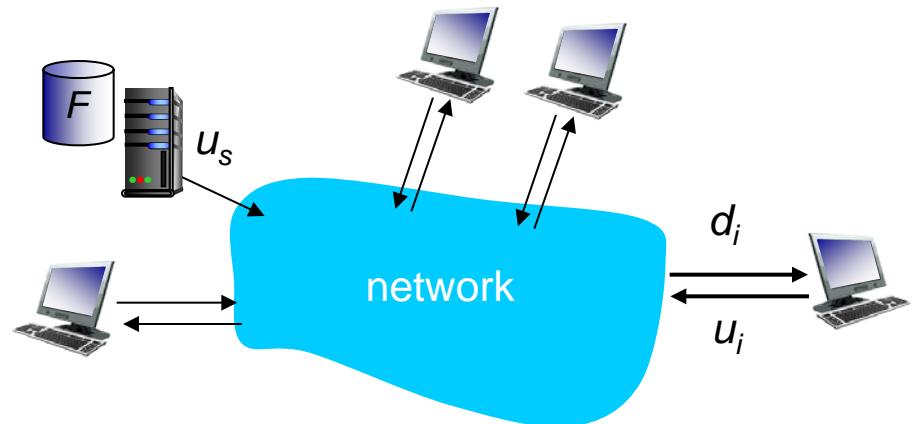
Question: how much time to distribute file (size F) from one server to N peers?

- 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
 - max client download time: F/d_{\min}



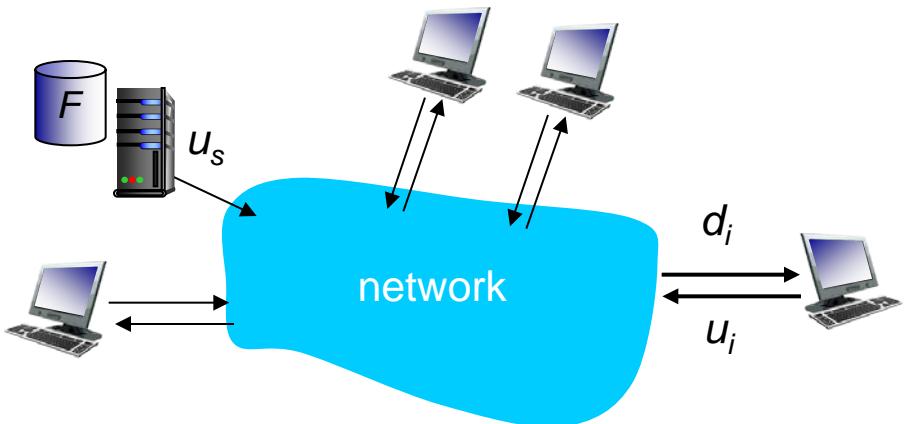
*time to distribute F
to N clients using
client-server approach*

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

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
 - max client download time: F/d_{\min}
- ❖ **clients:** as aggregate must download NF bits
 - max upload rate (limiting max download rate) is $u_s + \sum u_i$



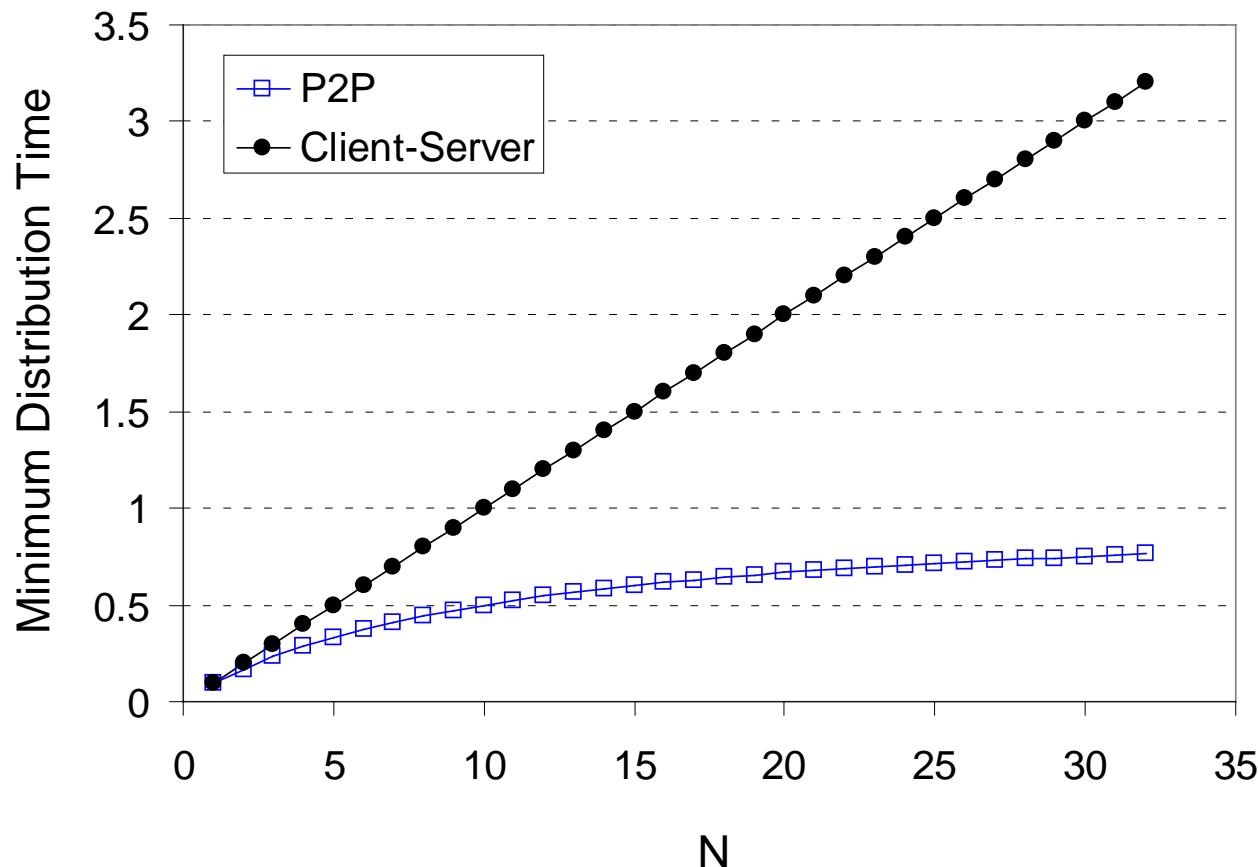
time to distribute F
to N clients using
P2P approach

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

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, $u_s = 10u$, $d_{min} \geq u_s$

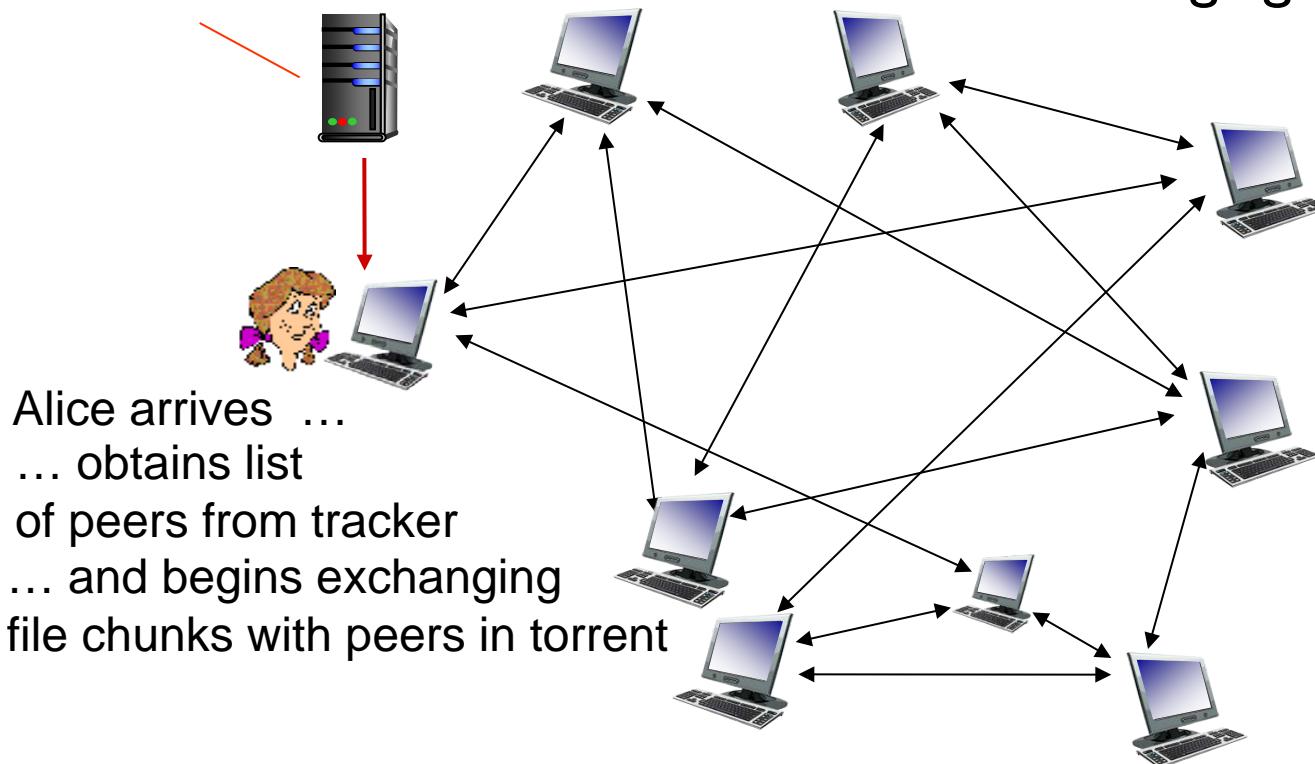


P2P file distribution: BitTorrent

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

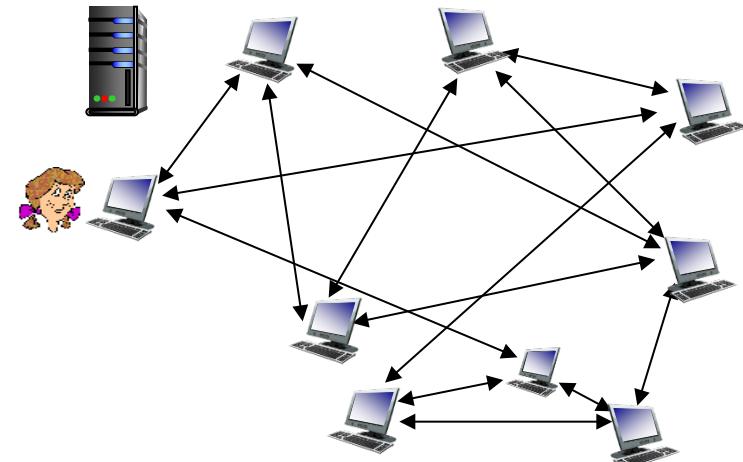
tracker: tracks peers
participating in torrent

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:

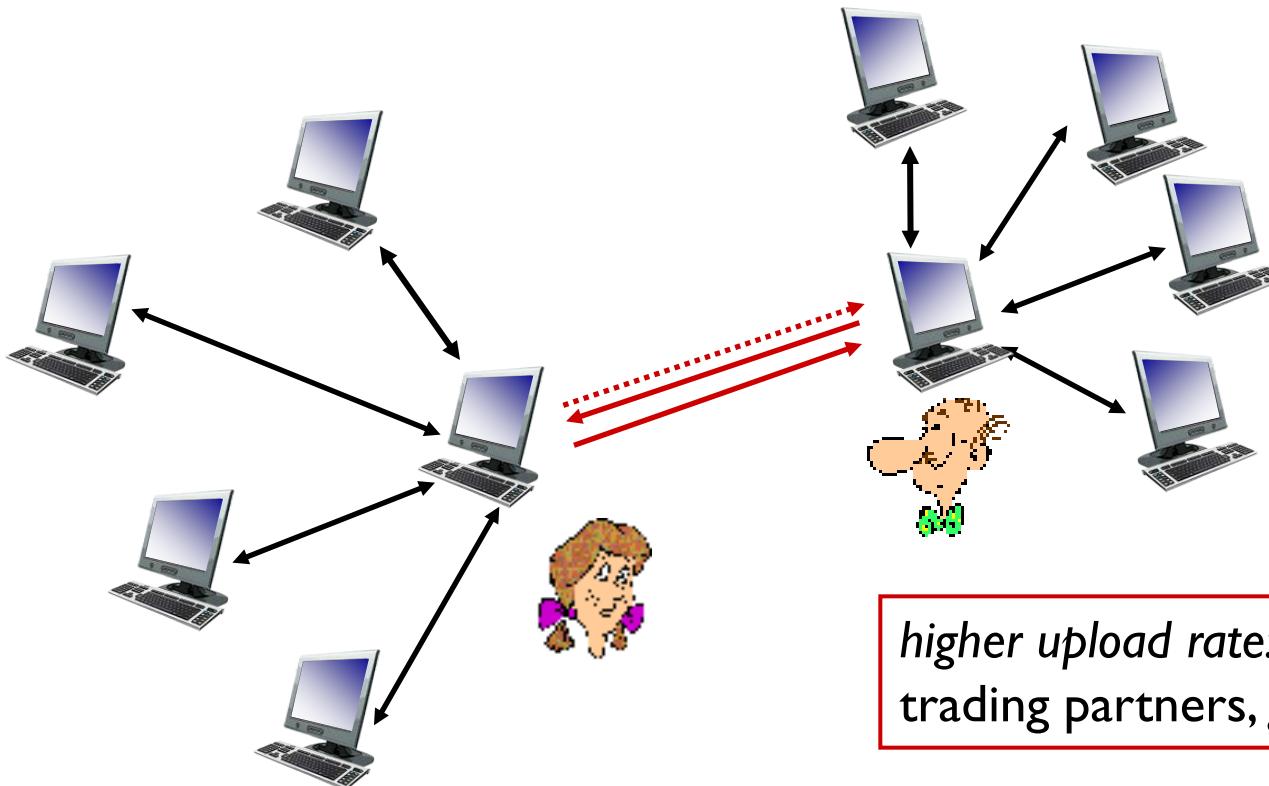
- ❖ at any given time, different peers have different subsets of file chunks
- ❖ periodically, Alice asks each peer for list of chunks that they have
- ❖ Alice requests missing chunks from peers, **rarest first**

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



higher upload rate: find better trading partners, get file faster !

Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

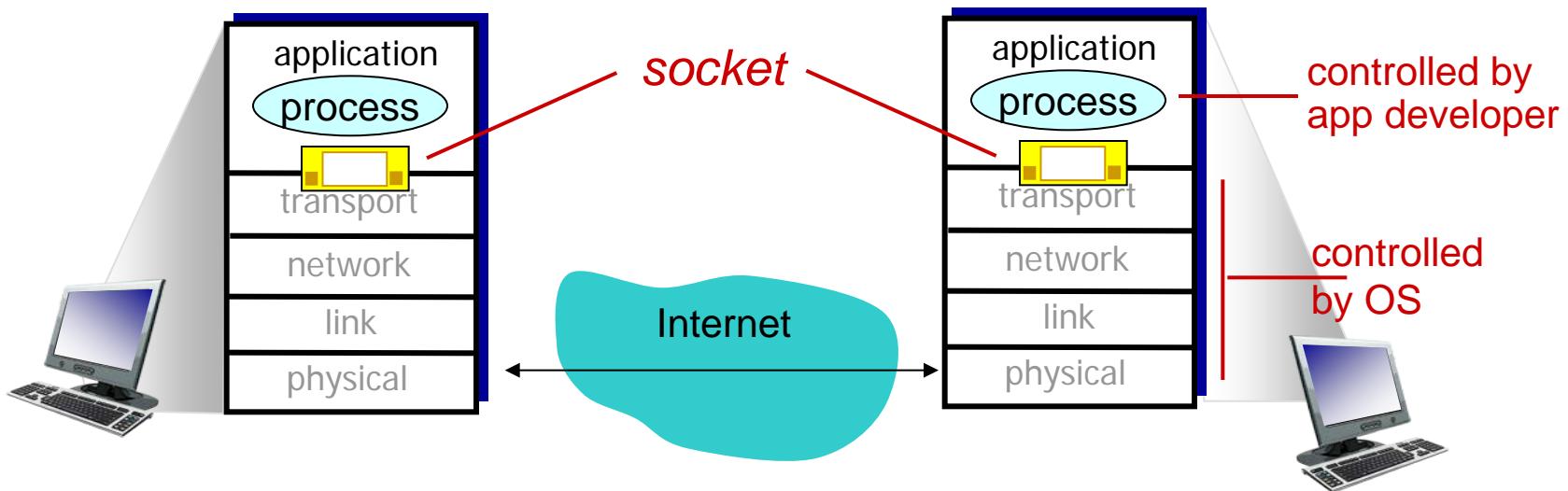
2.6 P2P applications

2.7 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 the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the 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
- ❖ rcvr 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

server (running on serverIP)

create socket, port= x:

```
serverSocket =  
socket(AF_INET,SOCK_DGRAM)
```

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:

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

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket
close
clientSocket

Example app: UDP client

Python UDPCClient

```
include Python's socket  
library → from socket import *  
  
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,(serverName, serverPort))  
  
read reply characters from  
socket into string → modifiedMessage, serverAddress =  
clientSocket.recvfrom(2048)  
  
print out received string  
and close socket → print modifiedMessage  
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 1:
Read from UDP socket into
message, getting client's
address (client IP and port) -----> message, clientAddress = serverSocket.recvfrom(2048)
                                            modifiedMessage = message.upper()
send upper case string -----> serverSocket.sendto(modifiedMessage, clientAddress)
back to this client
```

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'      //here, 127.0.0.1
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

create TCP socket for
server, remote port 12000



SOCK_STREAM

No need to attach server
name, port



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 1:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

create TCP welcoming
socket →

server begins listening for
incoming TCP requests →

loop forever →

server waits on accept()
for incoming requests, new
socket created on return →

read bytes from socket (but
not address as in UDP) →

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

Socket API in C Programming Language

- ❖ What is a socket?
 - The **point** where a **local application process** attaches to the **network**
 - An **interface** between an **application** and the **network**
 - An application creates the socket
- ❖ The interface defines operations for
 - Creating a socket
 - Attaching a socket to the network
 - Sending and receiving messages through the socket
 - Closing the socket

Socket

- ❖ **Socket Family**
 - PF_INET denotes the Internet family
 - PF_UNIX denotes the Unix pipe facility
 - PF_PACKET denotes direct access to the network interface (i.e., it bypasses the TCP/IP protocol stack)
- ❖ **Socket Type**
 - SOCK_STREAM is used to denote a byte stream
 - SOCK_DGRAM is an alternative that denotes a message oriented service, such as that provided by UDP

Creating a Socket

```
int sockfd = socket(address_family, type,  
protocol);
```

- ❖ The socket number returned is the socket descriptor for the newly created socket
- ❖ `int sockfd = socket (PF_INET, SOCK_STREAM, 0);`
- ❖ `int sockfd = socket (PF_INET, SOCK_DGRAM, 0);`

The combination of `PF_INET` and `SOCK_STREAM` implies TCP

Client-Serve Model with TCP

Server

- Passive open
- Prepares to accept connection, does not actually establish a connection

Server invokes

```
int bind (int socket, struct sockaddr *address,  
          int addr_len)  
int listen (int socket, int backlog)  
int accept (int socket, struct sockaddr  
*address,  
            int *addr_len)
```

Client-Serve Model with TCP

Bind

- Binds the newly created socket to the specified address i.e. the network address of the local participant (the server)
- Address is a data structure which combines IP and port

Listen

- Defines how many connections can be pending on the specified socket

Client-Serve Model with TCP

Accept

- Carries out the passive open
- Blocking operation
 - **Does not return until a remote participant has established a connection**
 - When it does, it returns a new socket that corresponds to the new established connection and the address argument contains the remote participant's address

Client-Serve Model with TCP

Client

- Application performs active open
- It says who it wants to communicate with

Client invokes

```
int connect (int socket, struct sockaddr  
*address,  
             int addr_len)
```

Connect

- Does not return until TCP has successfully established a connection at which application is free to begin sending data
- Address contains remote machine's address

Client-Serve Model with TCP

In practice

- The client usually specifies only remote participant's address and let's the system fill in the local information
- Whereas a server usually listens for messages on a well-known port
- A client does not care which port it uses for itself, the OS simply selects an unused one

Client-Serve Model with TCP

Once a **connection is established**, the application process invokes two operation

```
int send (int socket, char *msg, int msg_len,  
          int flags)
```

```
int recv (int socket, char *buff, int buff_len,  
          int  
flags)
```

Example Application: Client

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 5432
#define MAX_LINE 256

int main(int argc, char * argv[])
{
    FILE *fp;
    struct hostent *hp;
    struct sockaddr_in sin;
    char *host;
    char buf[MAX_LINE];
    int s;
    int len;
    if (argc==2) {
        host = argv[1];
    }
    else {
        fprintf(stderr, "usage: simplex-talk host\n");
        exit(1);
    }
```

Example Application: Client

```
/* translate host name into peer's IP address */
hp = gethostbyname(host);
if (!hp) {
    fprintf(stderr, "simplex-talk: unknown host: %s\n", host);
    exit(1);
}
/* build address data structure */
bzero((char *)&sin, sizeof(sin));
sin.sin_family = AF_INET; /* Internet Address*/
bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
sin.sin_port = htons(SERVER_PORT);
/* active open PF_INET is protocol family*/
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simplex-talk: socket");
    exit(1);
}
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("simplex-talk: connect");
    close(s);
    exit(1);
}
/* main loop: get and send lines of text */
while (fgets(buf, sizeof(buf), stdin)) {
    buf[MAX_LINE-1] = '\0';
    len = strlen(buf) + 1;
    send(s, buf, len, 0);
}
```

Example Application: Server

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define SERVER_PORT 5432
#define MAX_PENDING 5
#define MAX_LINE 256

int main()
{
    struct sockaddr_in sin;
    char buf[MAX_LINE];
    int len;
    int s, new_s;
    /* build address data structure */
    bzero((char *)&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(SERVER_PORT);

    /* setup passive open */
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("simplex-talk: socket");
        exit(1);
    }
```

Example Application: Server

```
if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
    perror("simplex-talk: bind");
    exit(1);
}
listen(s, MAX_PENDING);
/* wait for connection, then receive and print text */
while(1) {
    if ((new_s = accept(s, (struct sockaddr *)&sin, &len)) < 0) {
        perror("simplex-talk: accept");
        exit(1);
    }
    while (len = recv(new_s, buf, sizeof(buf), 0))
        fputs(buf, stdout);
    close(new_s);
}
}
```

Chapter 2: summary

our study of network apps 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
 - FTP
 - SMTP, POP, IMAP
 - DNS
 - P2P: BitTorrent, DHT
- ❖ 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 being communicated

important themes:

- ❖ control vs. data msgs
 - in-band, out-of-band
- ❖ centralized vs. decentralized
- ❖ stateless vs. stateful
- ❖ reliable vs. unreliable msg transfer
- ❖ “complexity at network edge”

Chapter 2

Additional Slides

