

# **CHAPTER 2**

# **APPLICATION LAYER**

**Abdullah Yasin Nur**

**1**

# **CHAPTER 2: OUTLINE**

---

**2.1 principles of network applications**

**2.2 Web and HTTP**

**2.3 electronic mail**  
• SMTP, POP3, IMAP

**2.4 DNS**

**2.5 P2P applications**

**2.6 video streaming and content distribution networks**

**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
  - content distribution networks

- 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

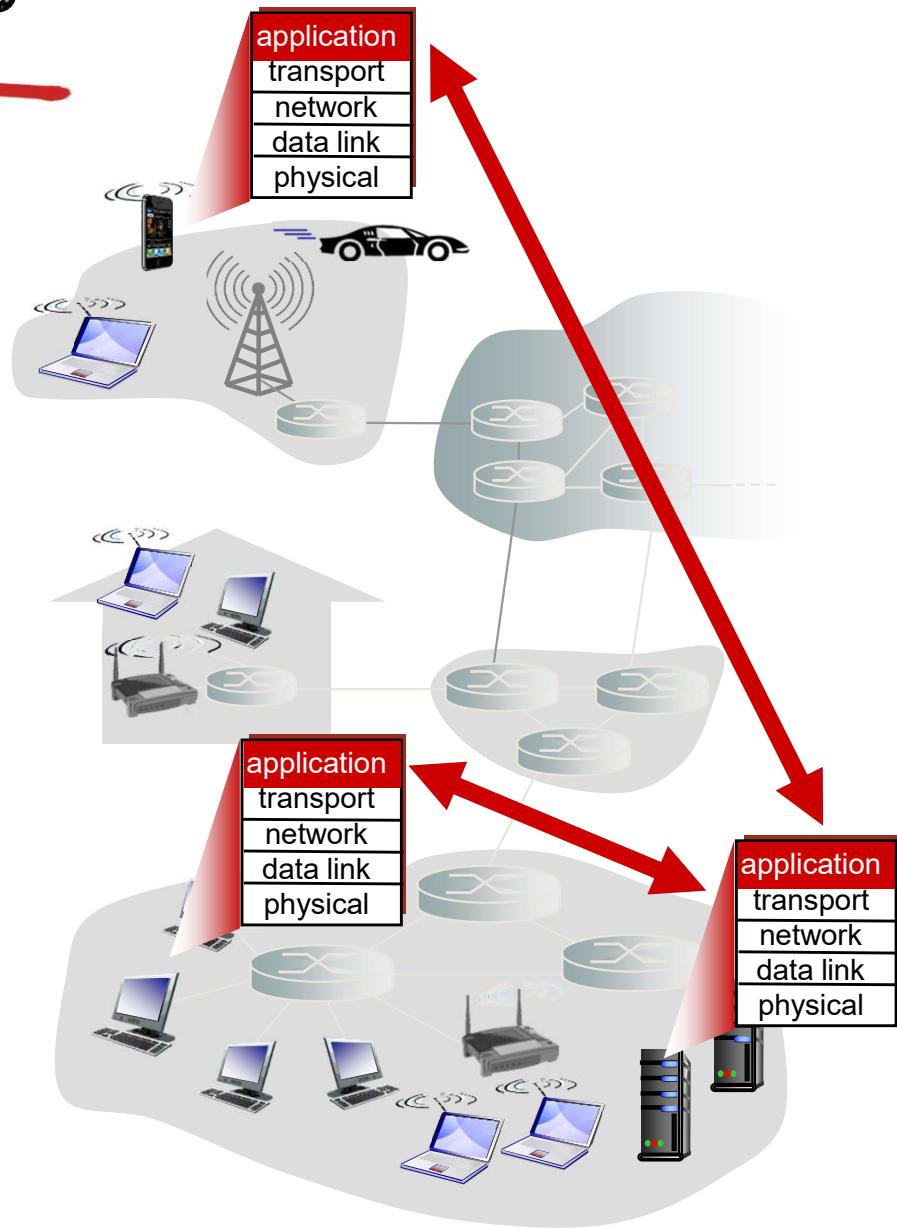
# CREATING A NETWORK APP

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



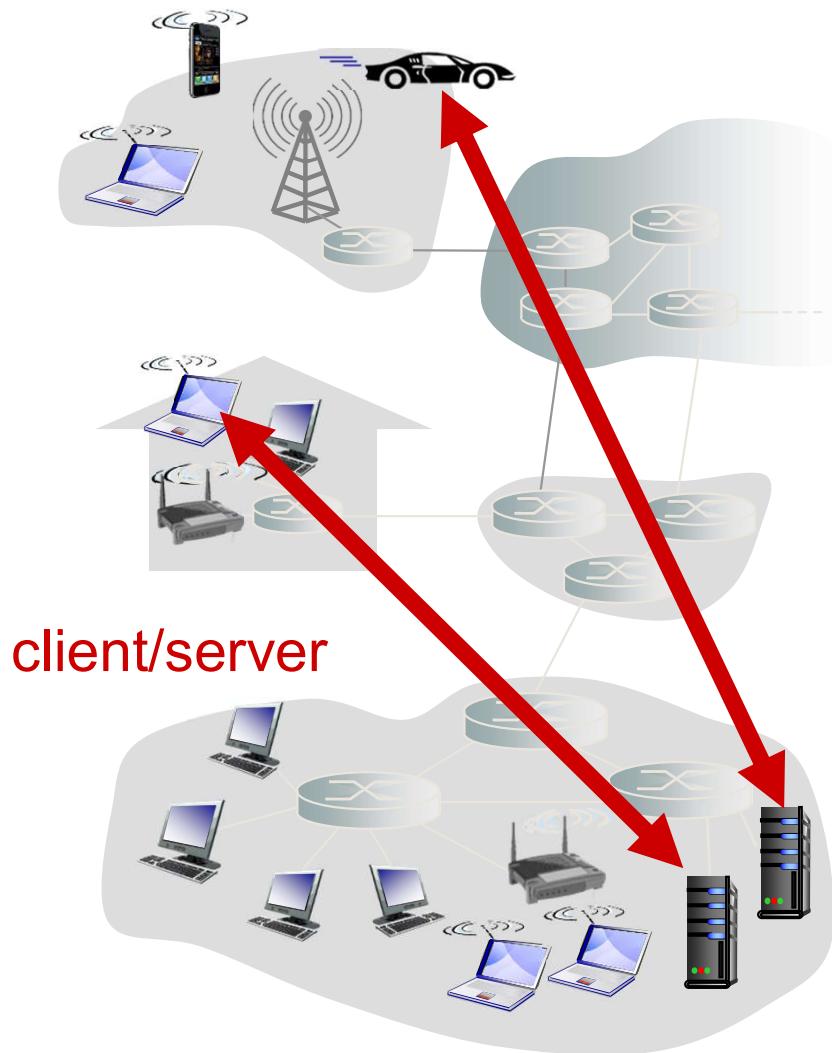
# APPLICATION ARCHITECTURES

---

possible structure of applications:

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

# CLIENT-SERVER ARCHITECTURE



## server:

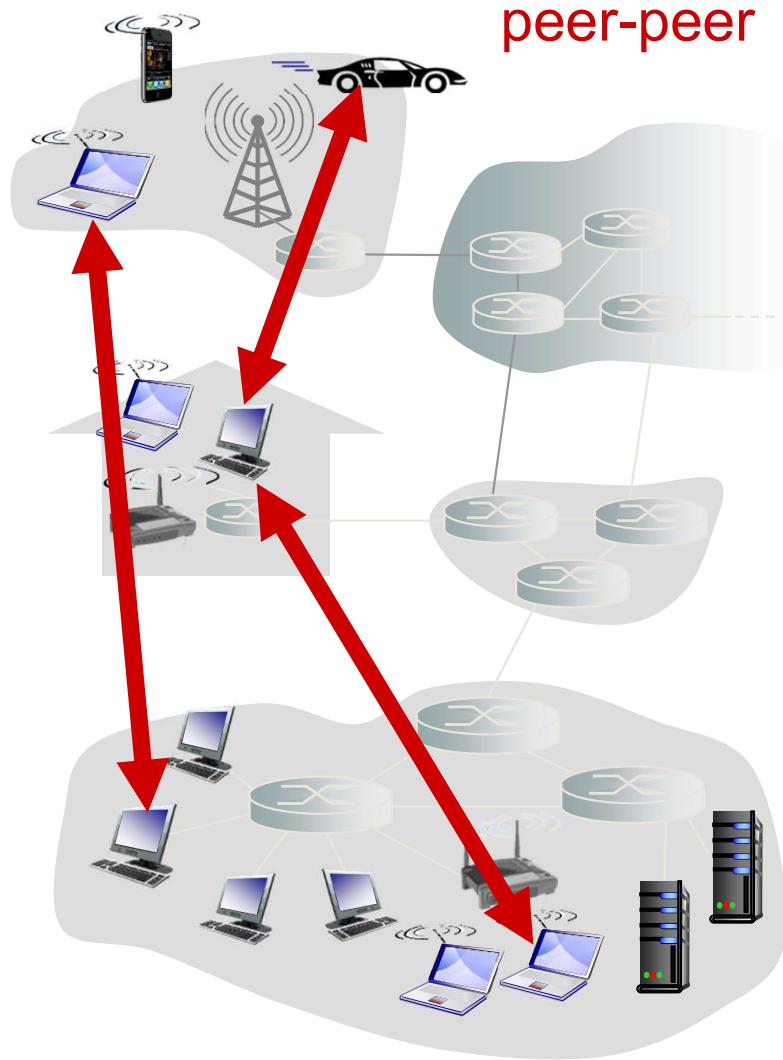
- always-on host
- permanent IP address
- data centers for scaling

## 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

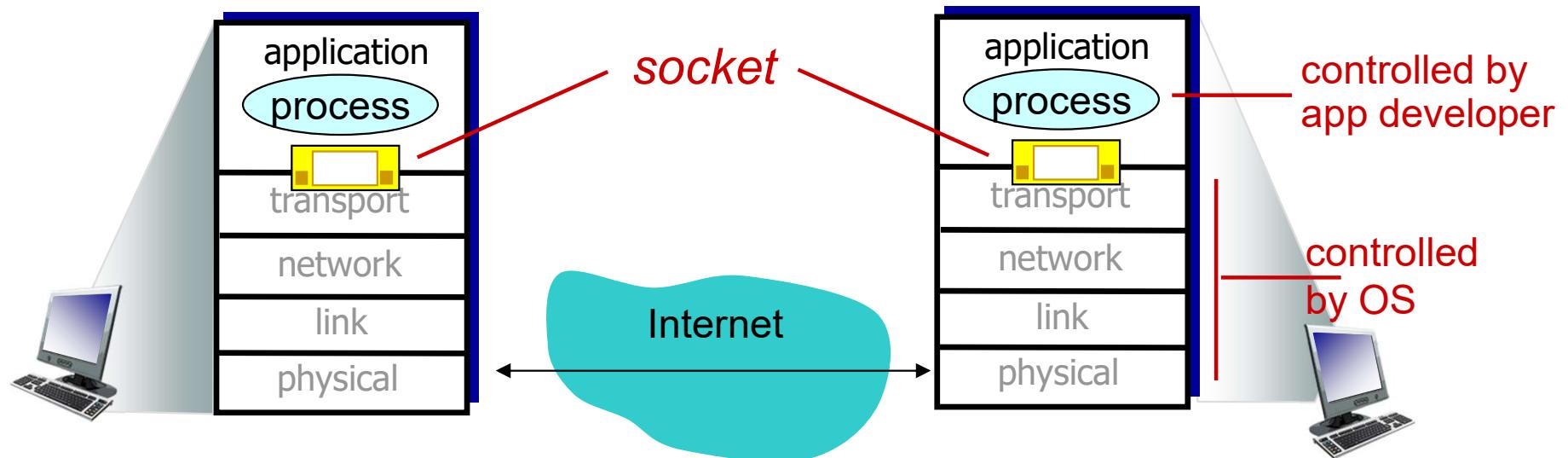
*client process*: process that initiates communication

*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 cs.uno.edu web server:
  - IP address: 151.101.1.193
  - port number: 80

# 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**

<b>application</b>	<b>data loss</b>	<b>throughput</b>	<b>time sensitive</b>
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
- cleartext passwords sent into socket traverse Internet in cleartext

## SSL

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

## SSL is at app layer

- apps use SSL libraries, that “talk” to TCP

## SSL socket API

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

# **CHAPTER 2: OUTLINE**

---

2.1 principles of network  
applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and  
content distribution  
networks

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**

*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

---

## *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)

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

1b. 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.)

time  
↓

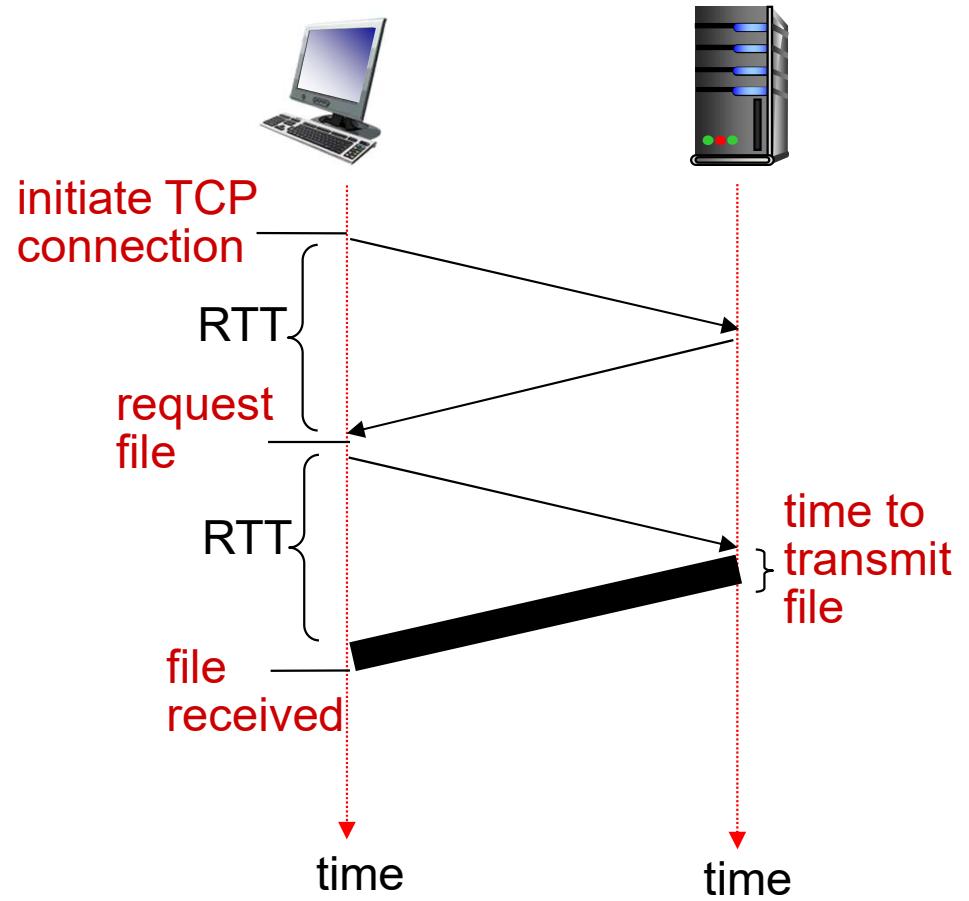
4. HTTP server closes TCP connection.
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects
6. Steps 1-5 repeated for each of 10 jpeg objects

# 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)

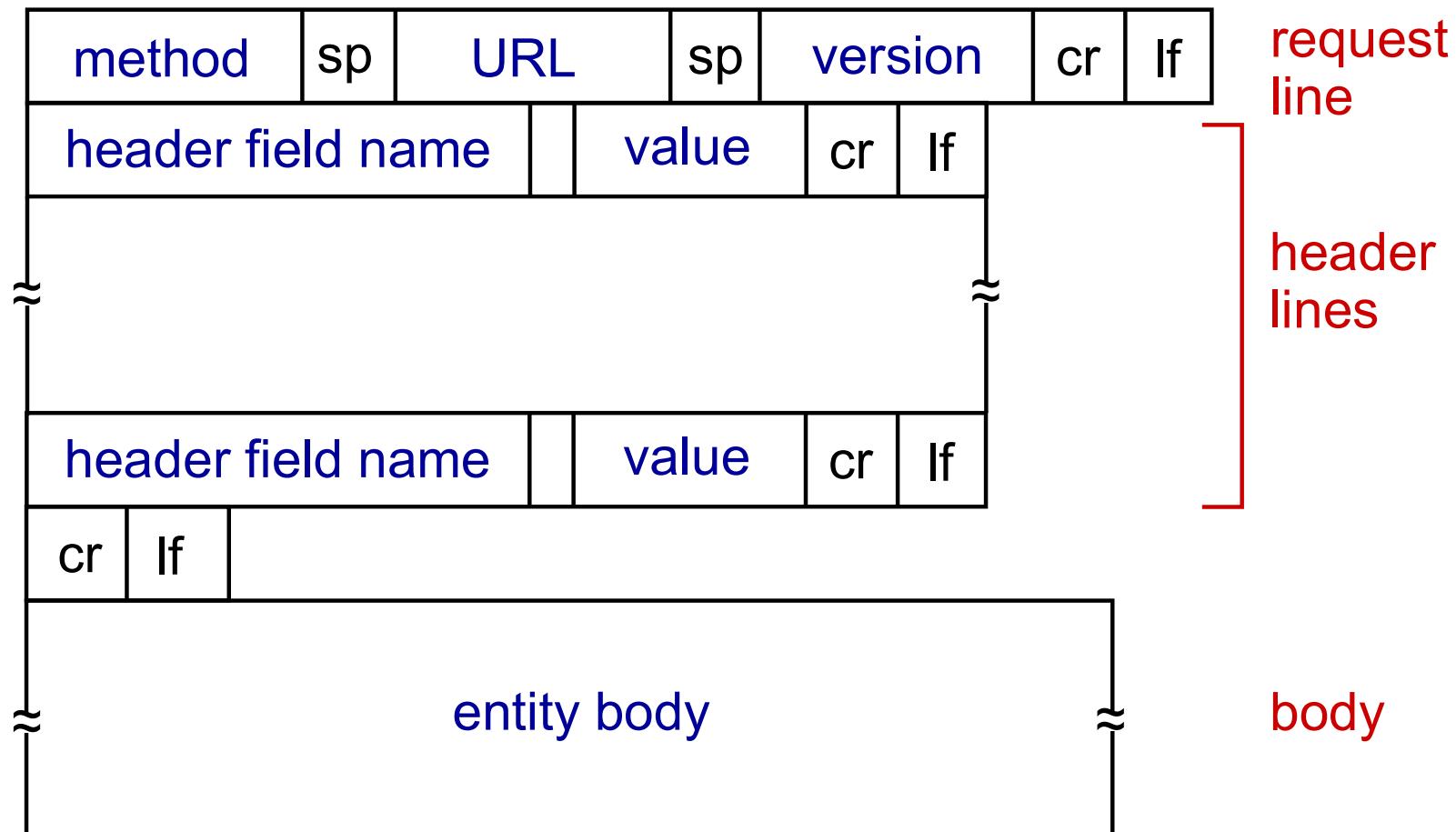
header  
lines

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\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 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**

# USER-SERVER STATE: COOKIES

---

many Web sites use  
cookies

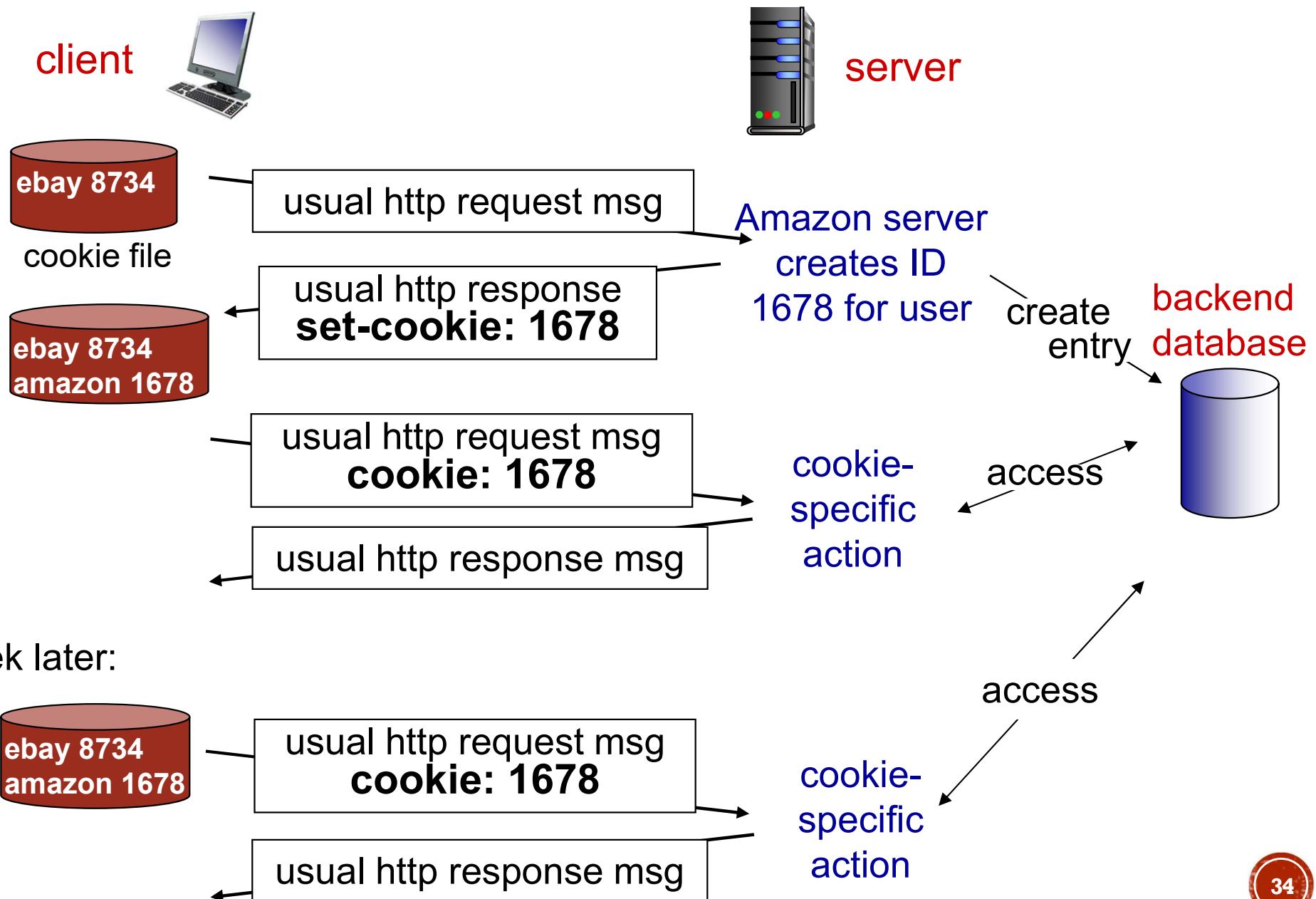
*four components:*

- 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

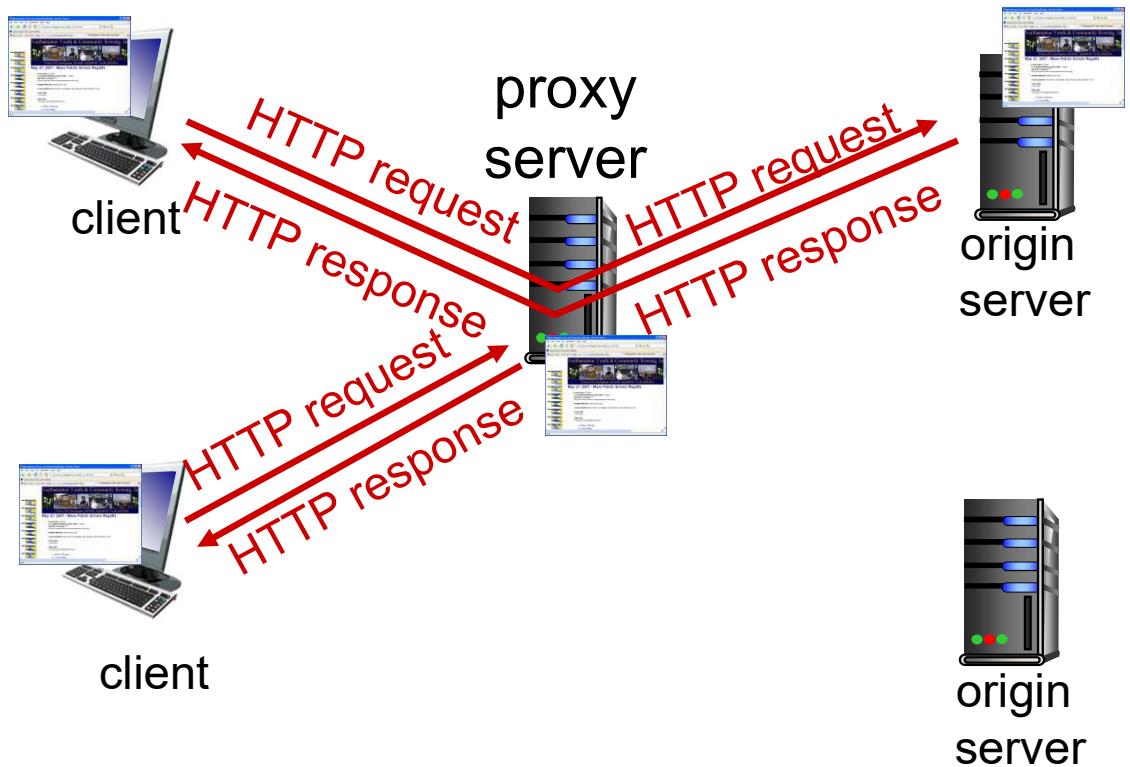
*how to keep “state”:*

- 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)

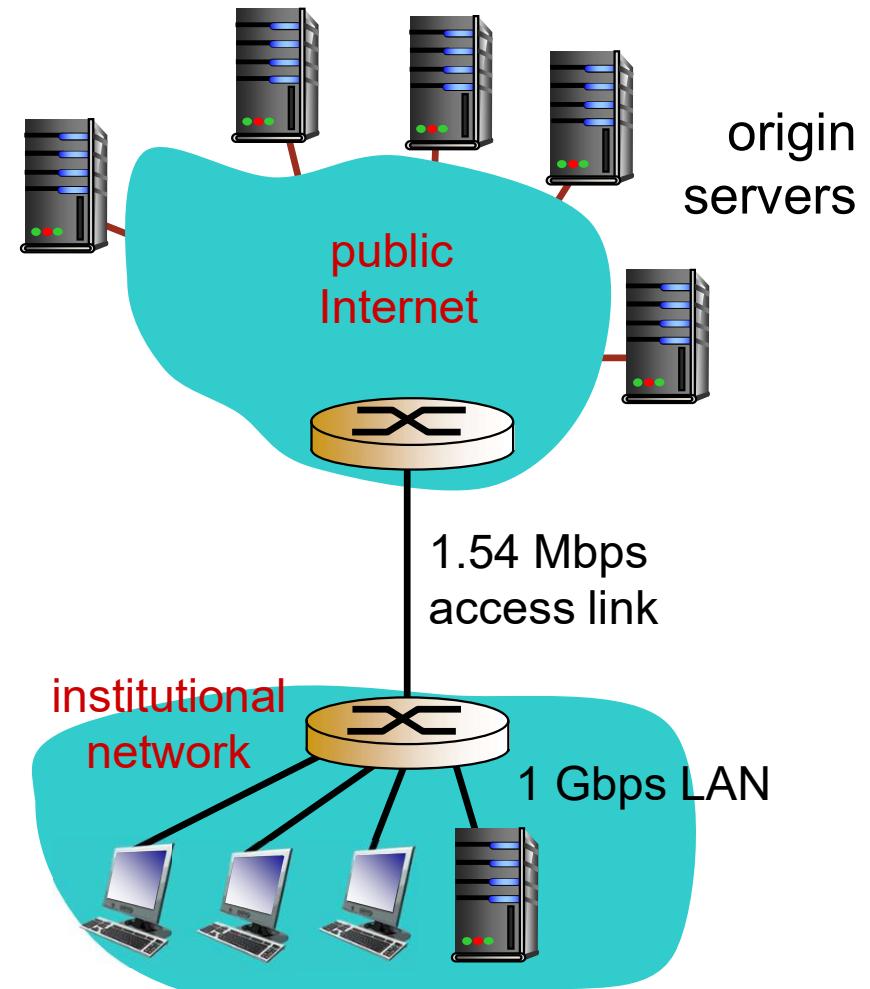
# CACHING EXAMPLE:

## *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

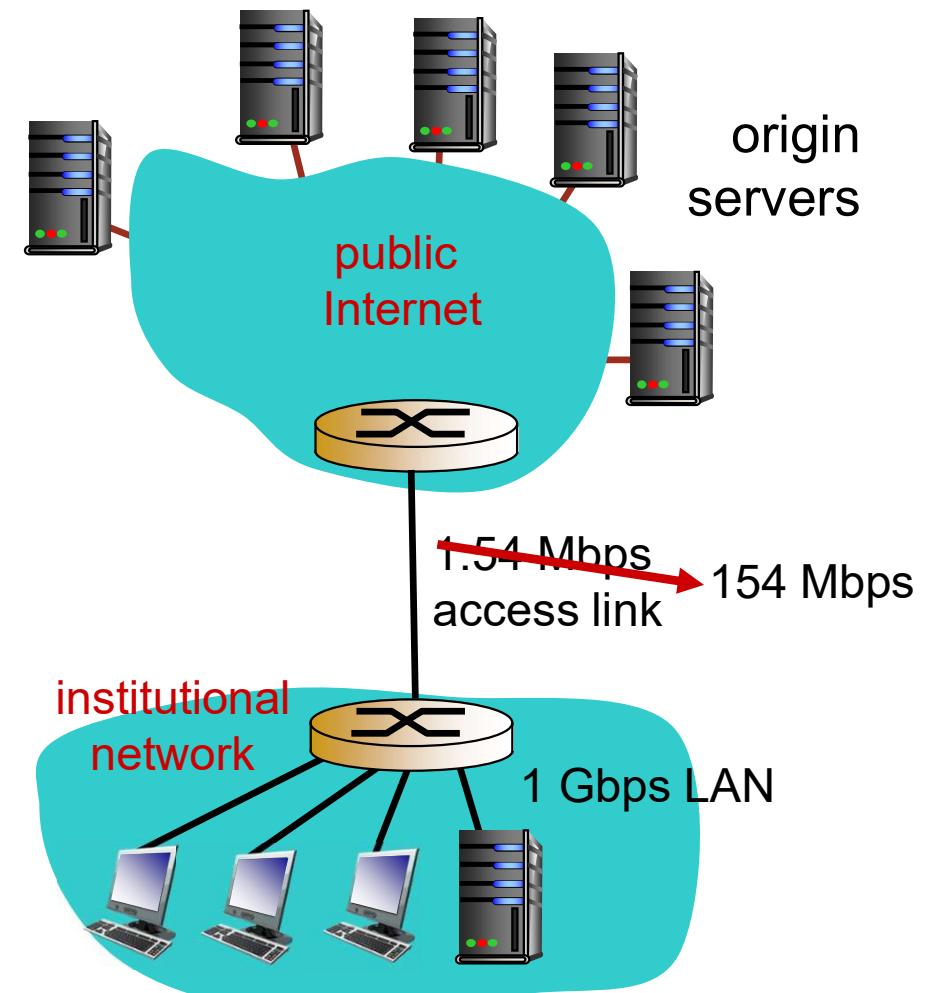


# CACHING EXAMPLE: FATTER ACCESS LINK

## *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~~ 154 Mbps

## **consequences:**



**Cost:** increased access link speed (not cheap!)

# CACHING EXAMPLE: INSTALL LOCAL CACHE

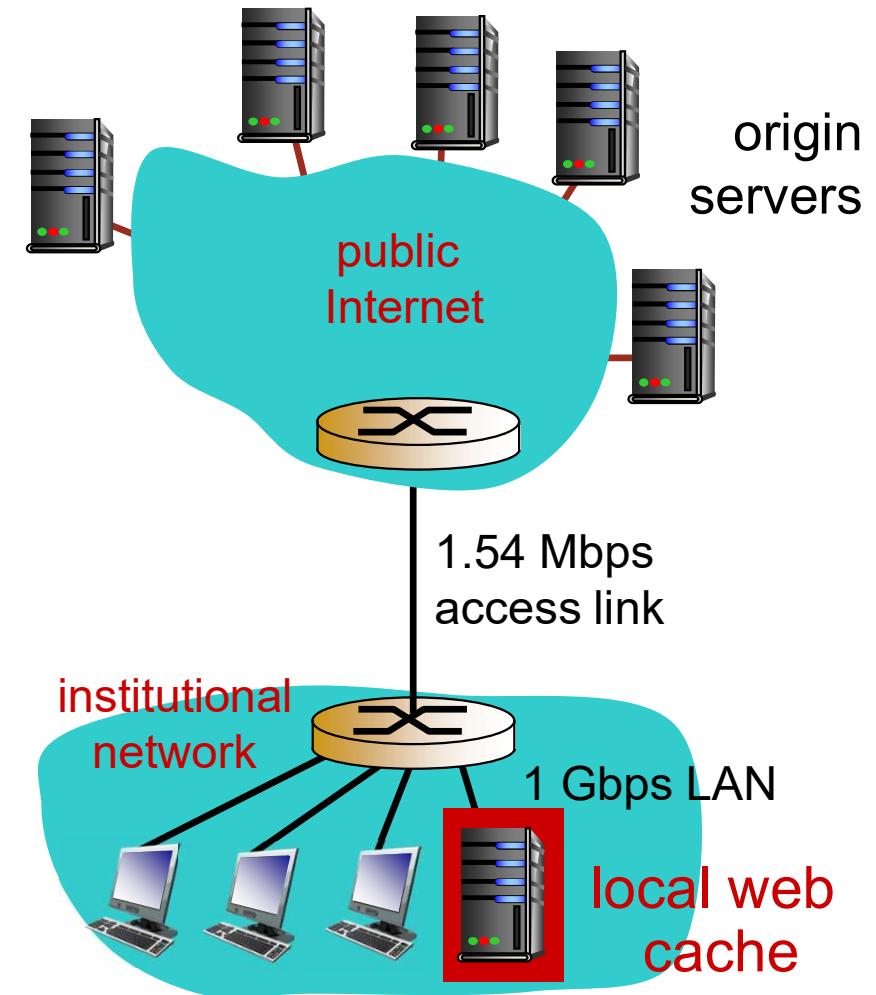
## *assumptions:*

- 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?*

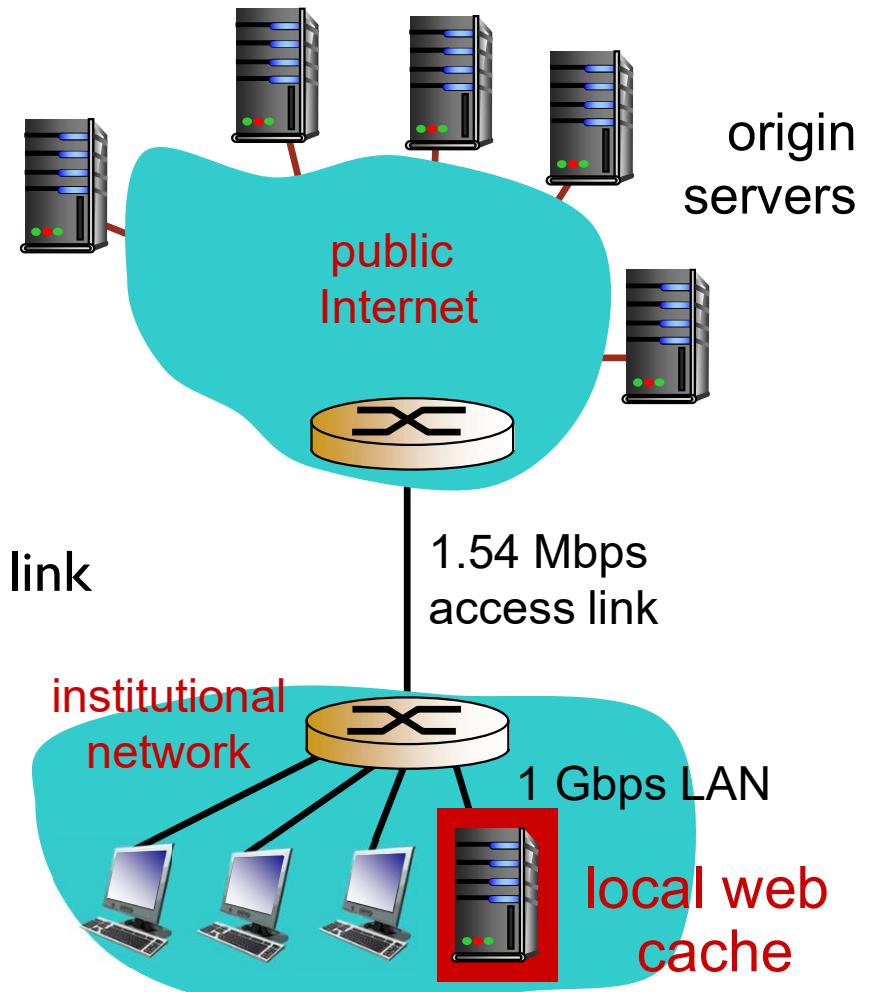


*Cost:* web cache (cheap!)

# 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
    - $= 0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
    - $= 0.6 (2.01) + 0.4 (\sim \text{msecs}) = \sim 1.2 \text{ secs}$
    - less than with 154 Mbps link (and  
cheaper too!)



# CONDITIONAL GET

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

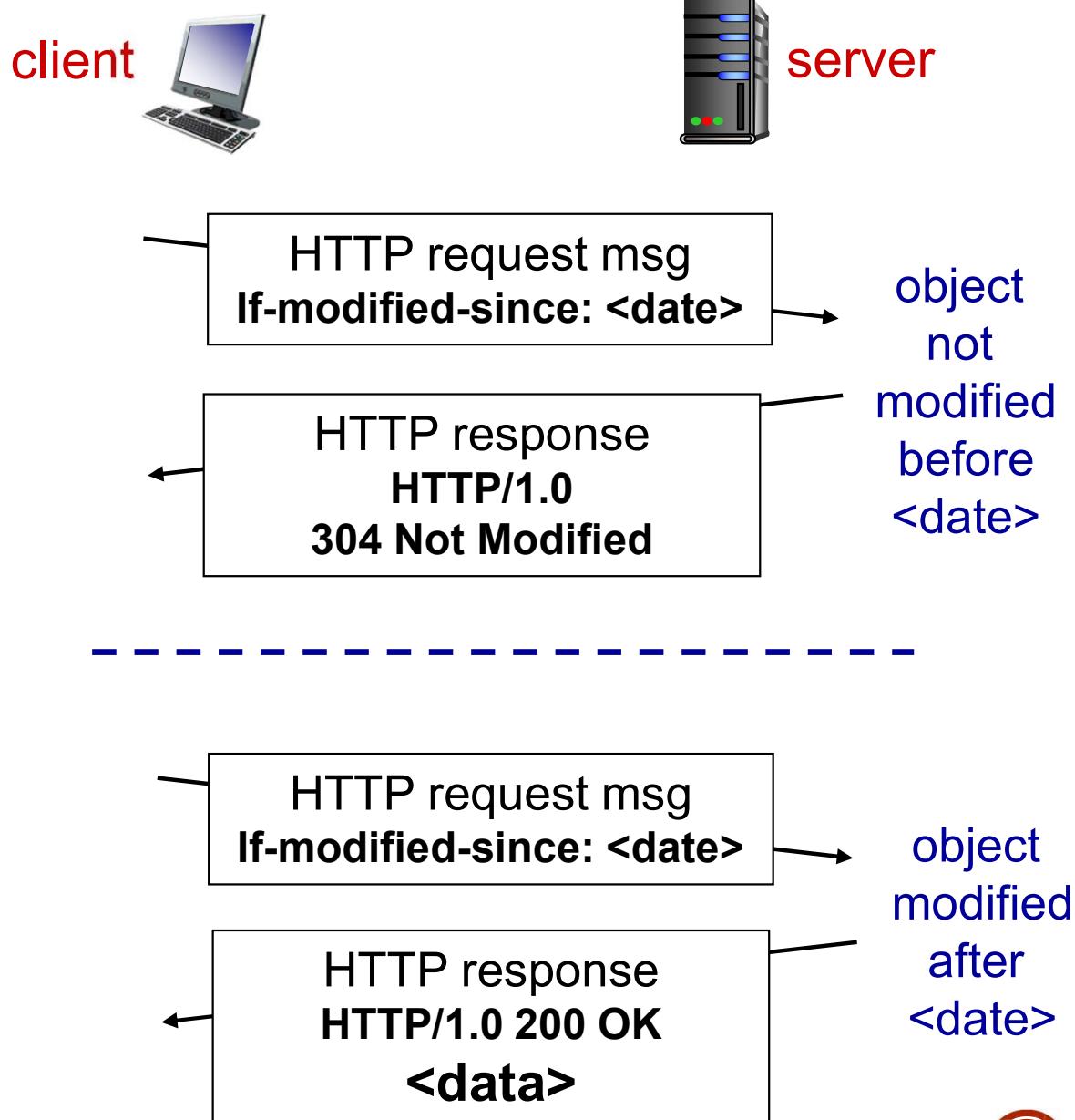
- no object transmission delay
  - lower link utilization

- **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

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and  
content distribution  
networks

2.7 socket programming with  
UDP and TCP

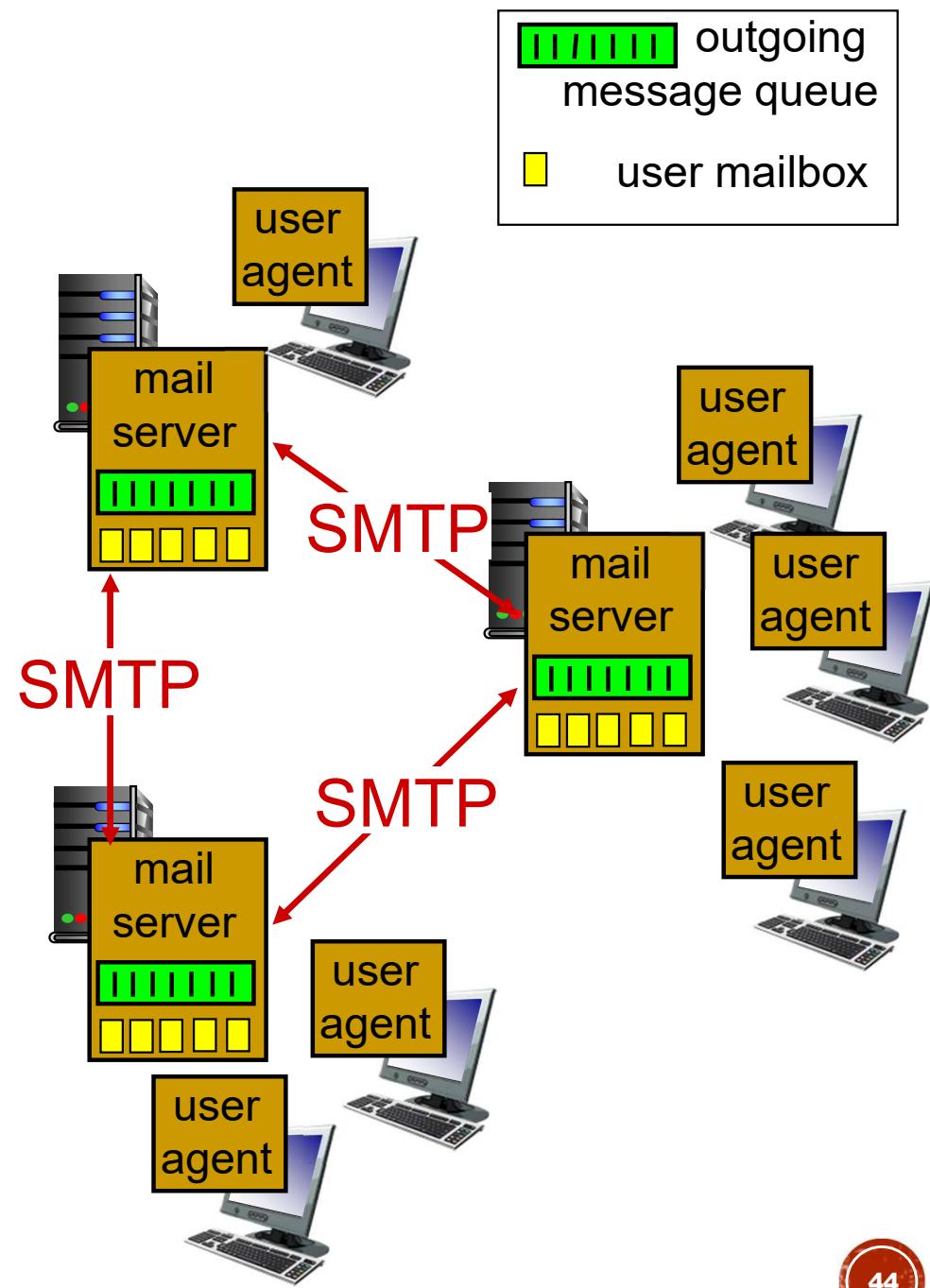
# ELECTRONIC MAIL

*Three major components:*

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

## *User Agent*

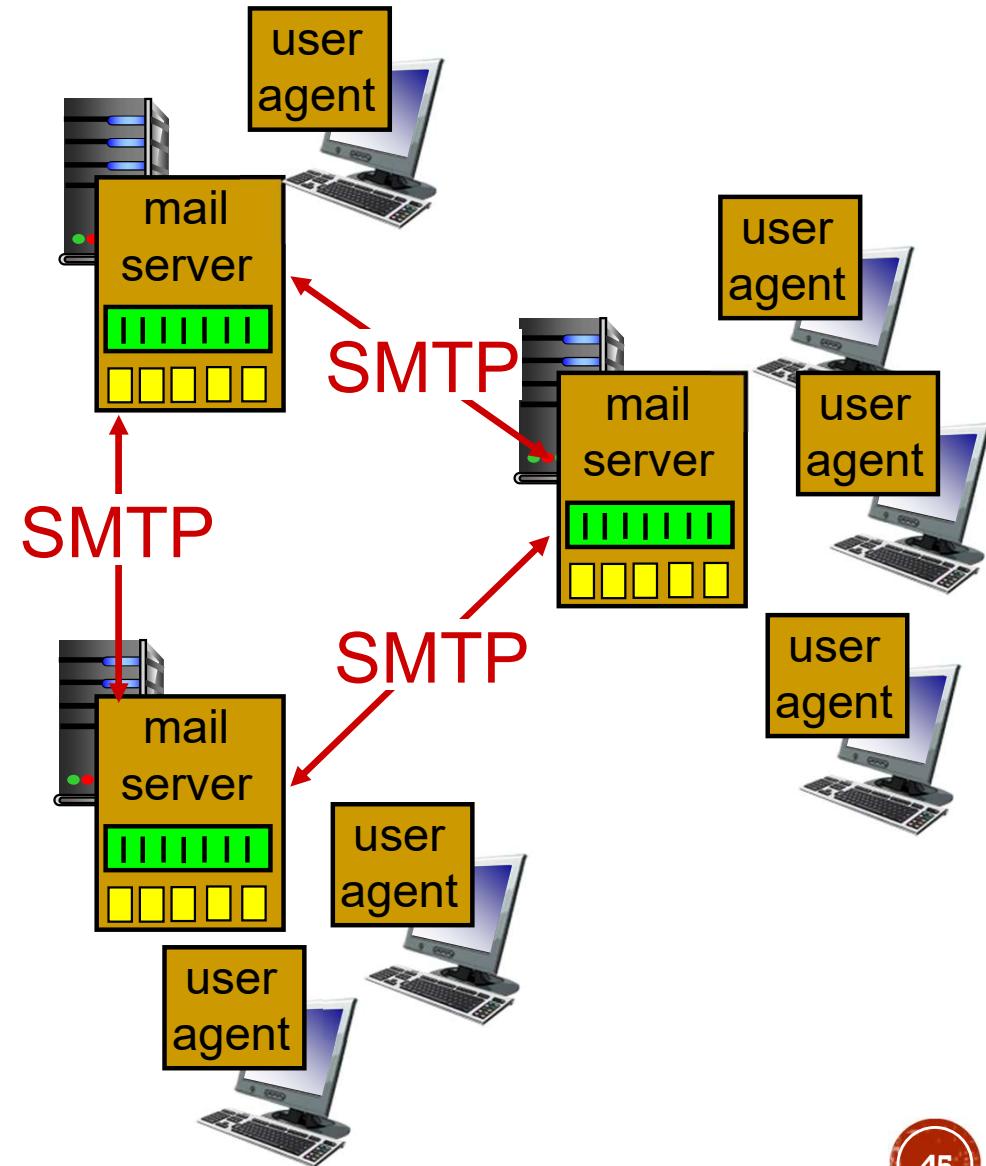
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



# 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



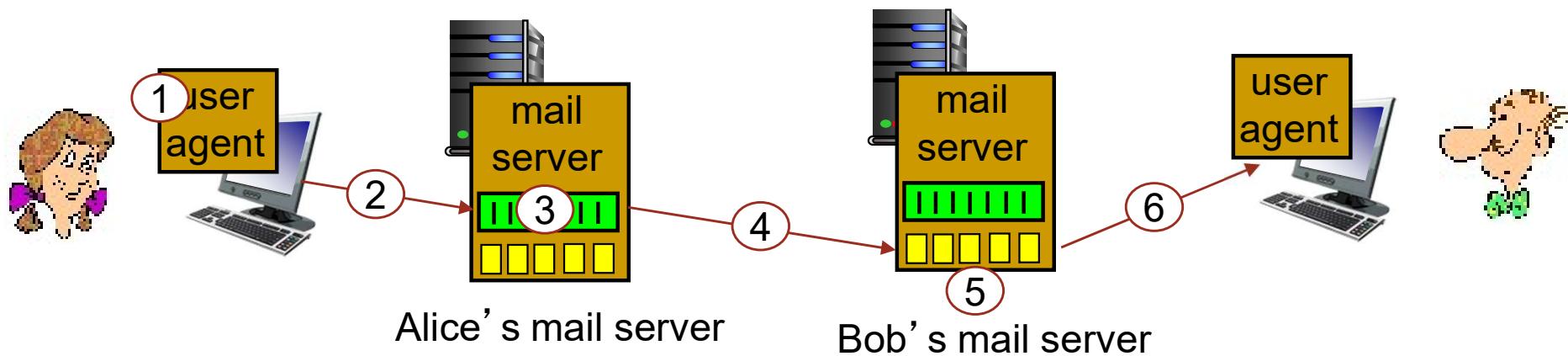
# ELECTRONIC MAIL: SMTP [RFC 2821]

---

- 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)
  - **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



# **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 message
- SMTP: multiple objects sent in multipart message

# MAIL MESSAGE FORMAT

SMTP: protocol for  
exchanging email  
messages

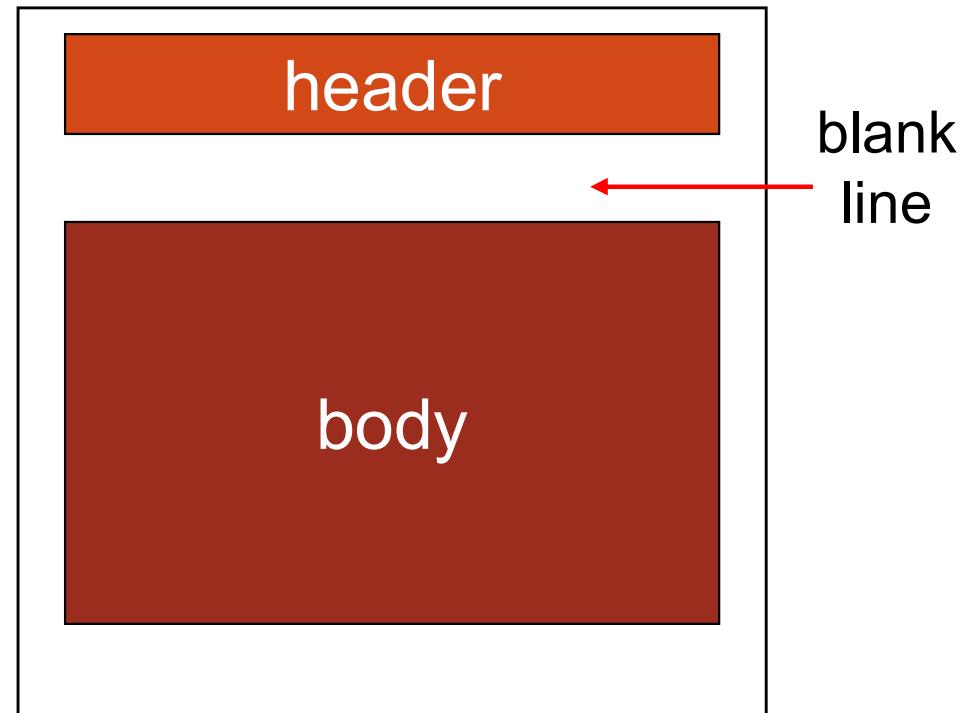
RFC 822: standard for text  
message format:

- header lines, e.g.,

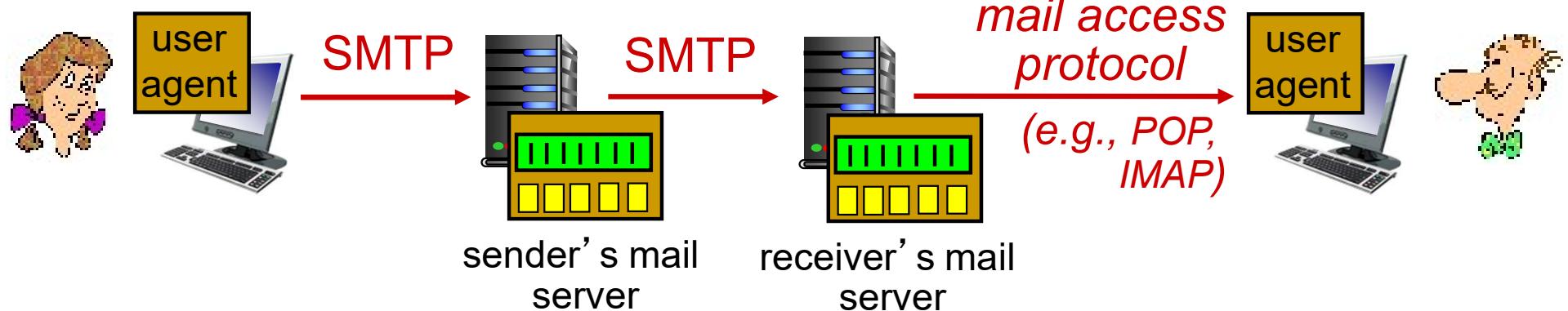
- To:
- From:
- Subject:

*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 messages 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

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and  
content distribution  
networks

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*

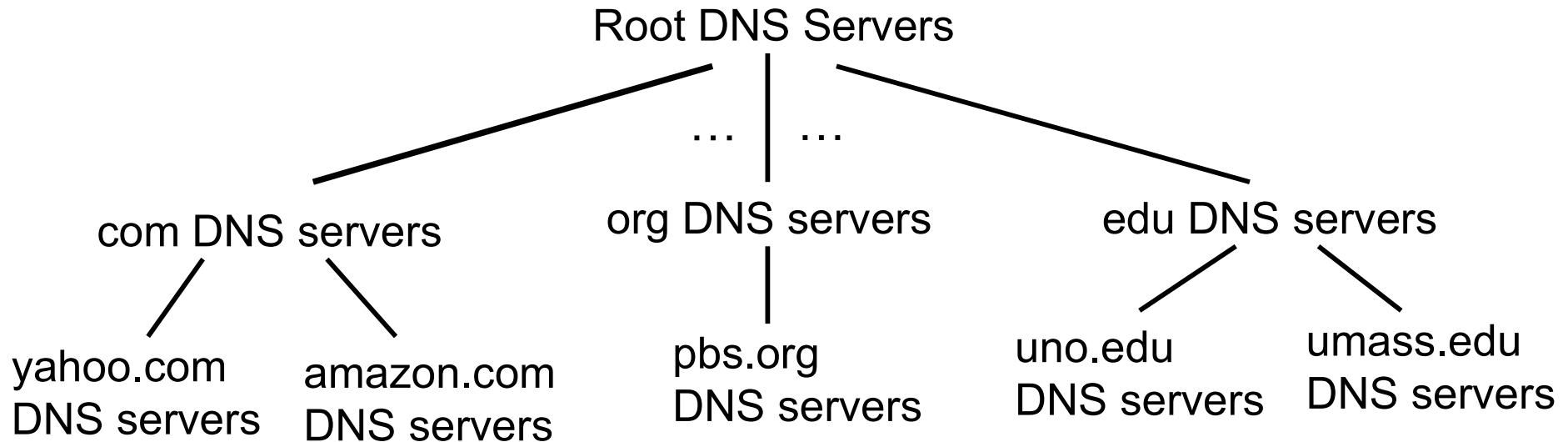
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

## *why not centralize DNS?*

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

*A: doesn't scale!*

# DNS: A DISTRIBUTED, HIERARCHICAL DATABASE

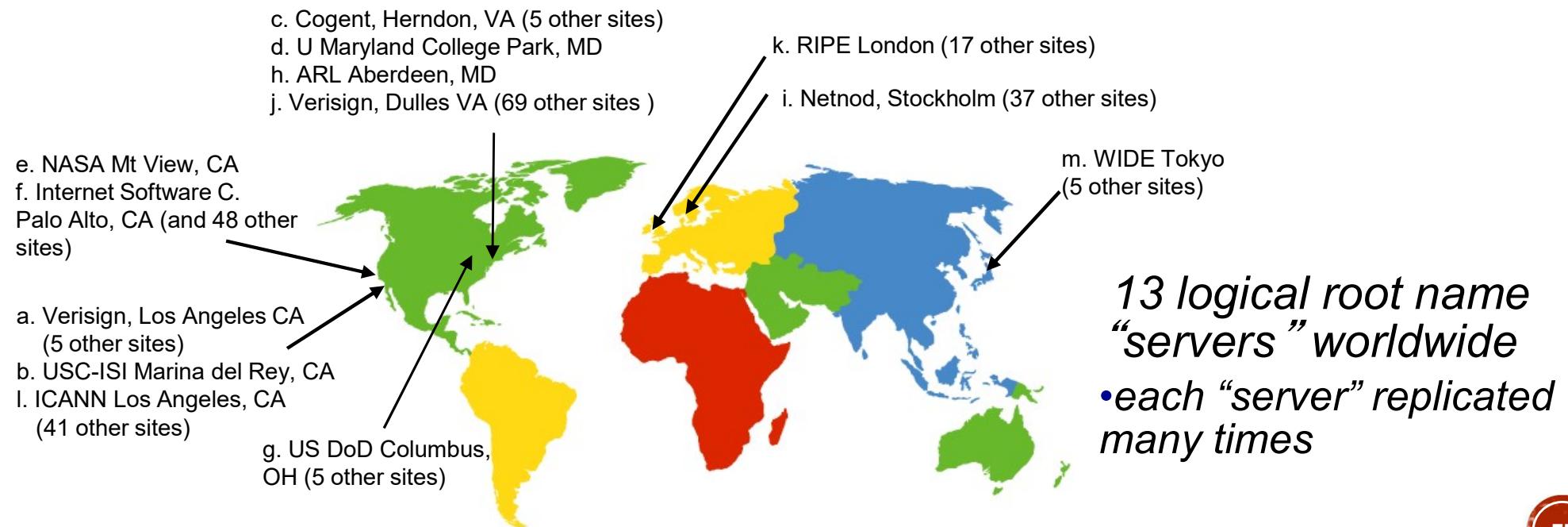


*client wants IP for www.amazon.com; 1<sup>st</sup> approximation:*

- 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

- 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



# 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

---

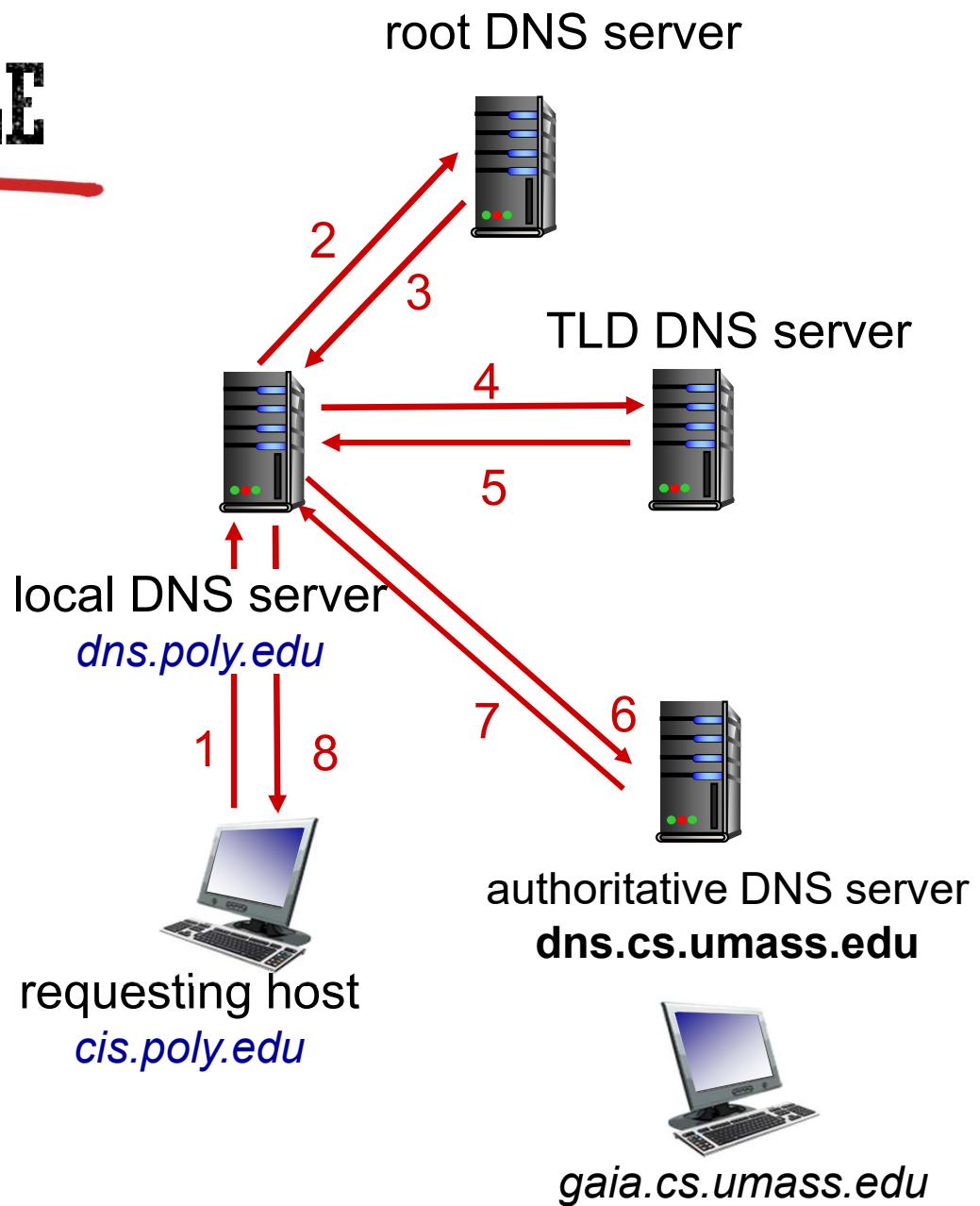
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

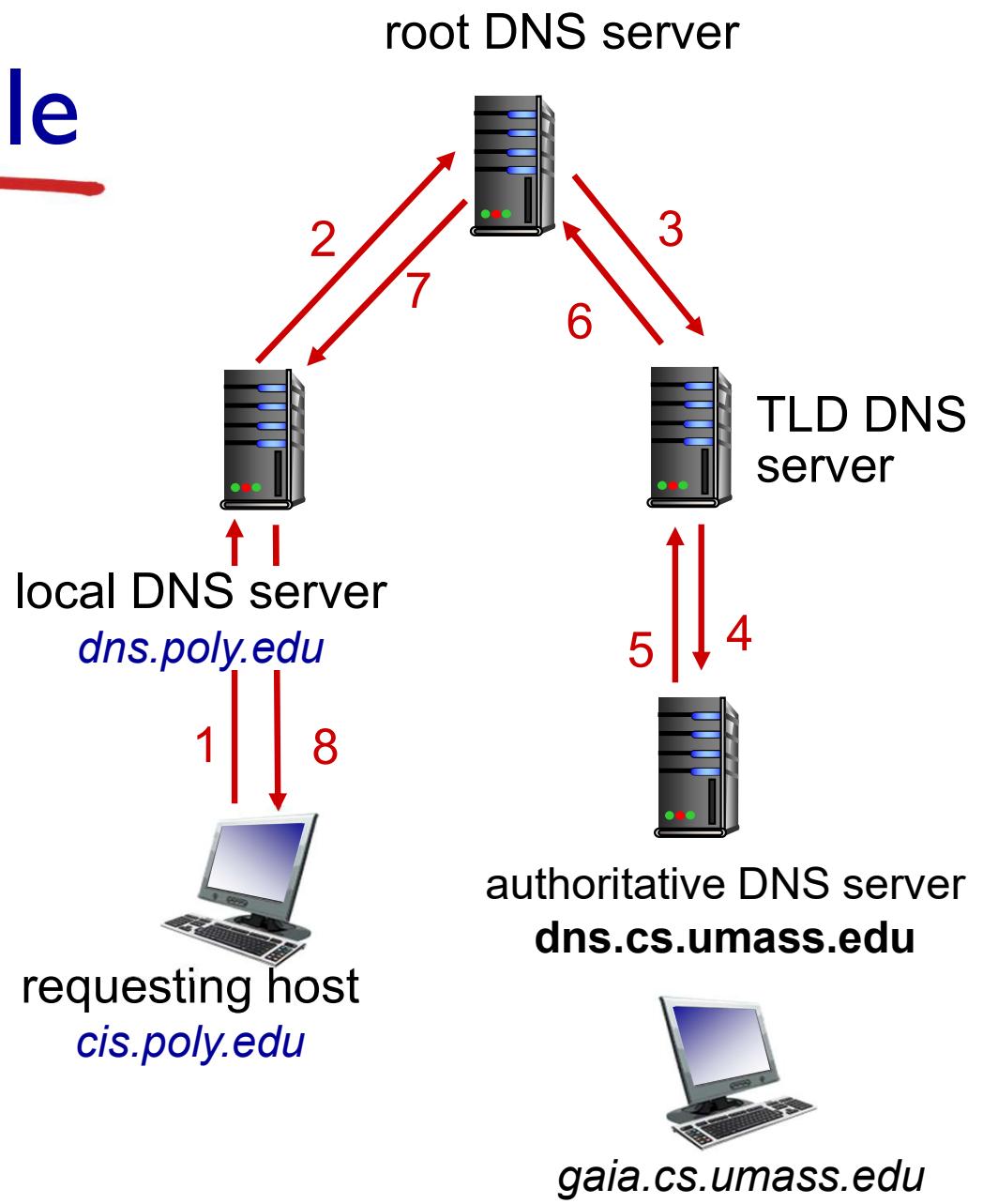
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”



# 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 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**

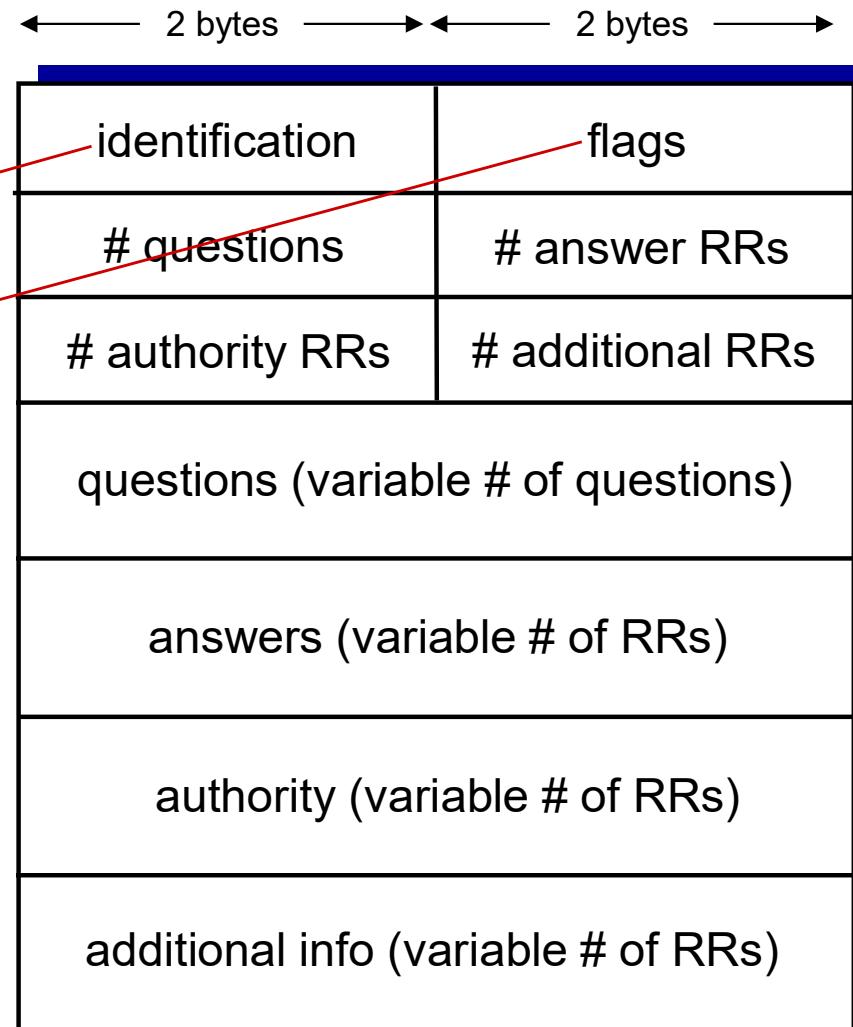
- **value** is name of mailserver  
associated with **name**

# DNS PROTOCOL, MESSAGES

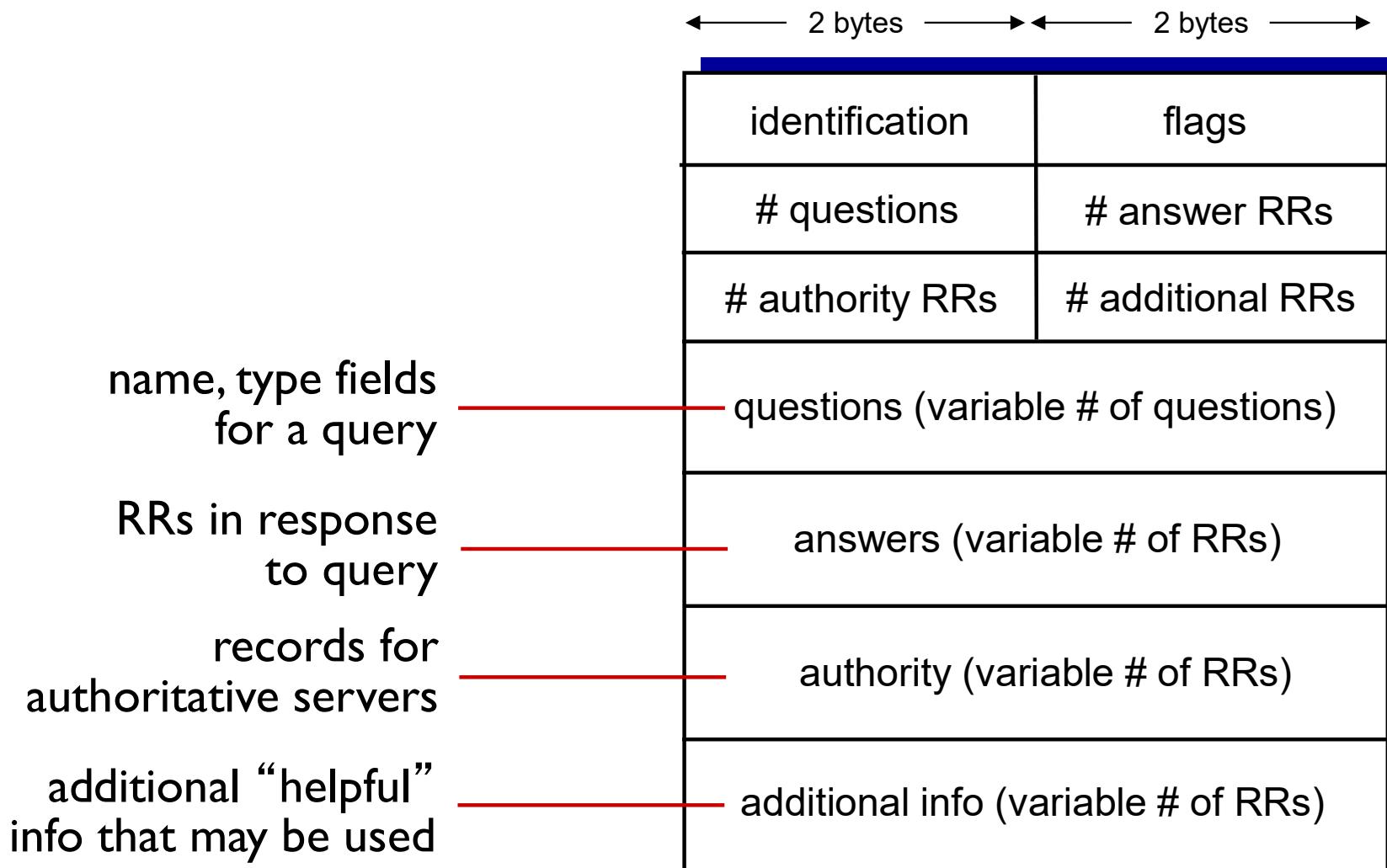
- *query* and *reply* messages, both with same *message format*

## 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



# 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 two RRs into .com TLD server:  
`(networkuptopia.com, dns1.networkuptopia.com, NS)`  
`(dns1.networkuptopia.com, 212.212.212.1, A)`
- create authoritative server type A record for  
`www.networkuptopia.com`; type MX record for `networkuptopia.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 bypass
- bombard TLD servers
  - potentially more dangerous

- The most recent large-scale DDoS attack took place on October 21, 2016.
- The attack targeted the DNS infrastructure of DYN and took some of the most popular sites offline including Netflix, Twitter, CNN, PayPal, and New York Times

# 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 bypass
- 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

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

**2.5 P2P applications**

2.6 video streaming and content distribution networks

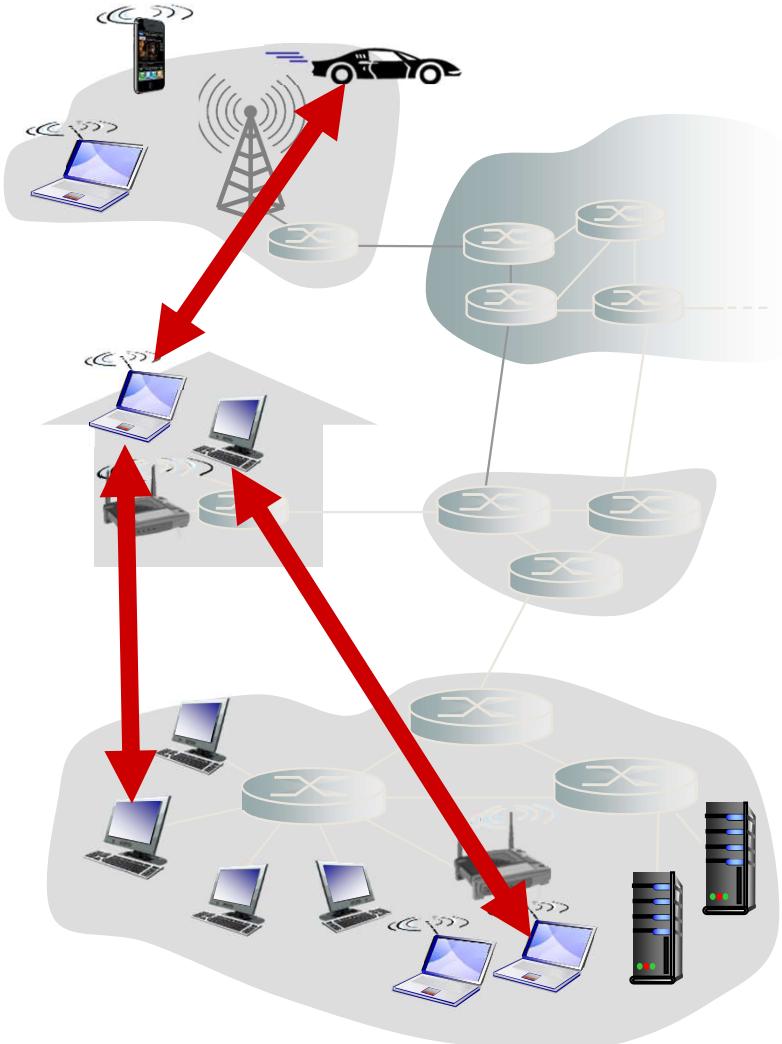
2.7 socket programming with UDP and TCP

# PURE P2P ARCHITECTURE

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

*examples:*

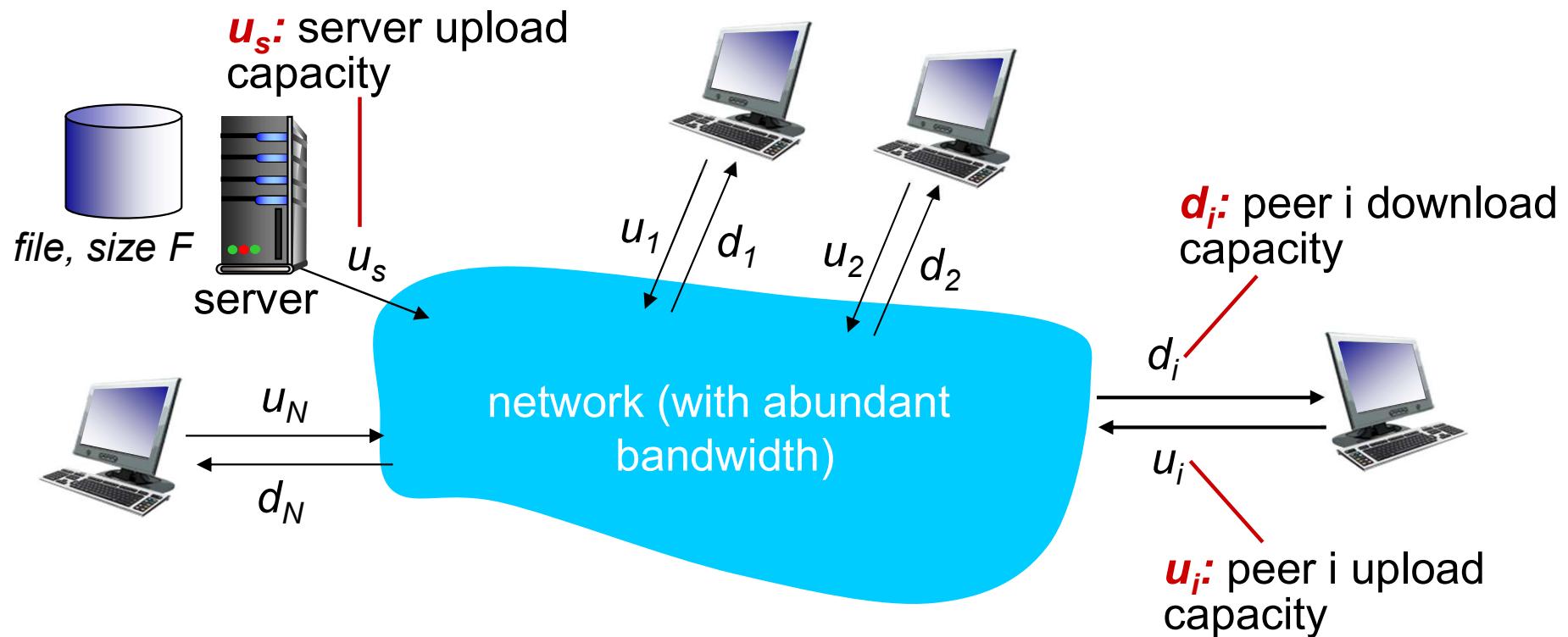
- file distribution (BitTorrent)
- Streaming (Xunlei KanKan)
- VoIP (Skype)



# 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

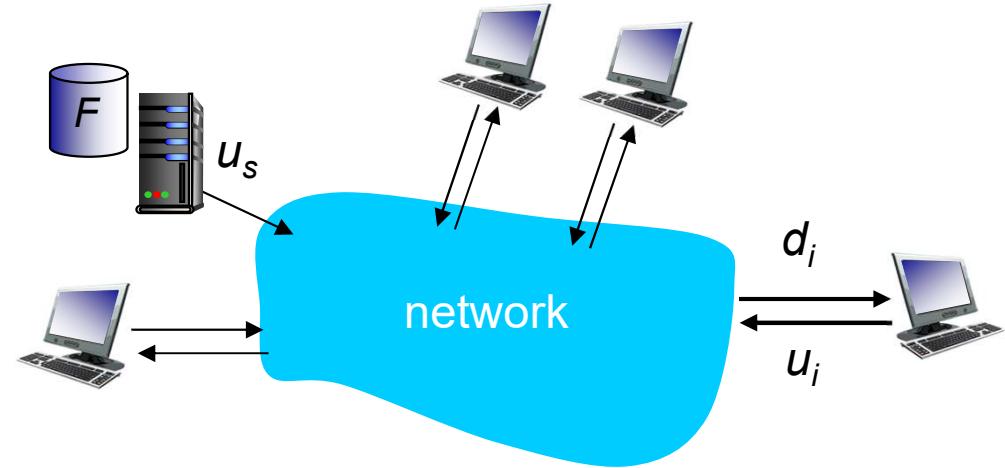


# FILE DISTRIBUTION TIME: CLIENT-SERVER

- **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
  - min client download time:  $F/d_{min}$



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

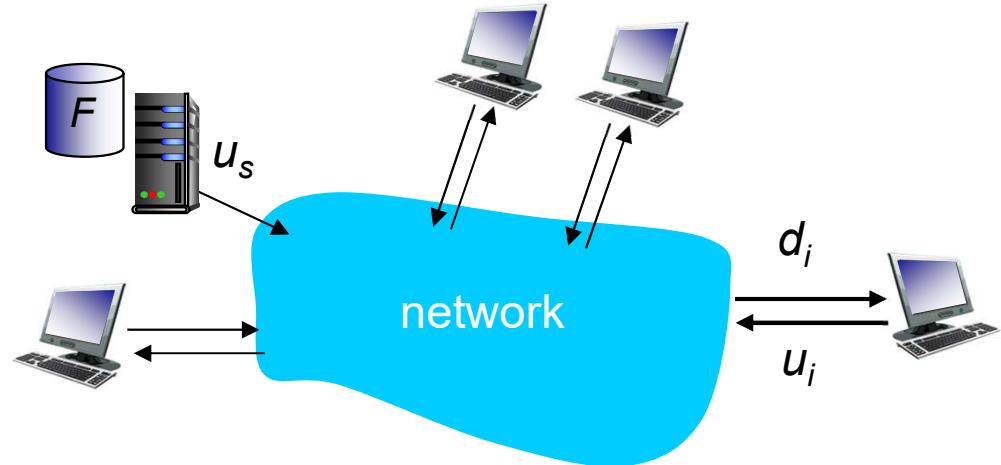
$$D_{c-s} \geq \max\{NF/u_s, F/d_{min}\}$$

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

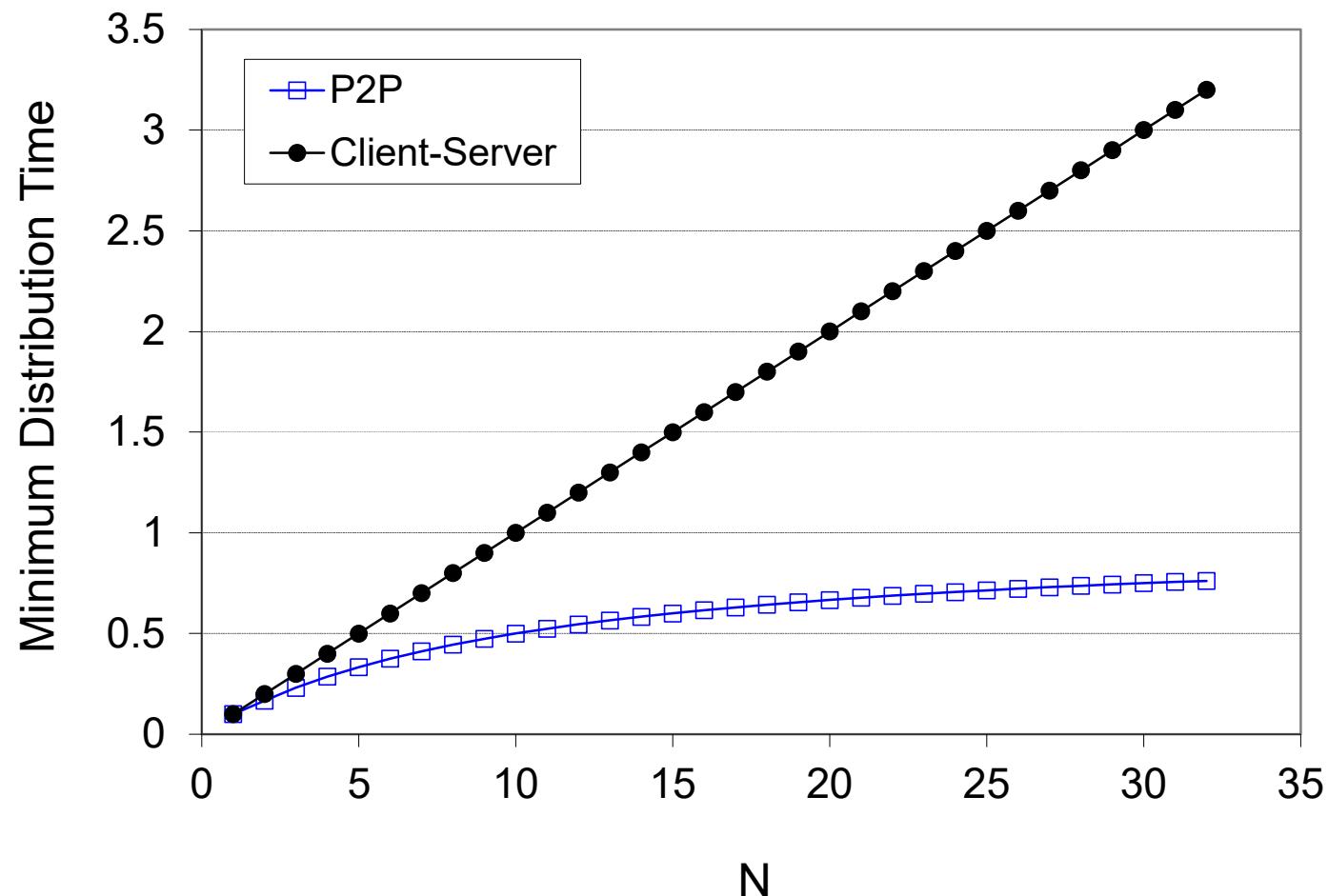
$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

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

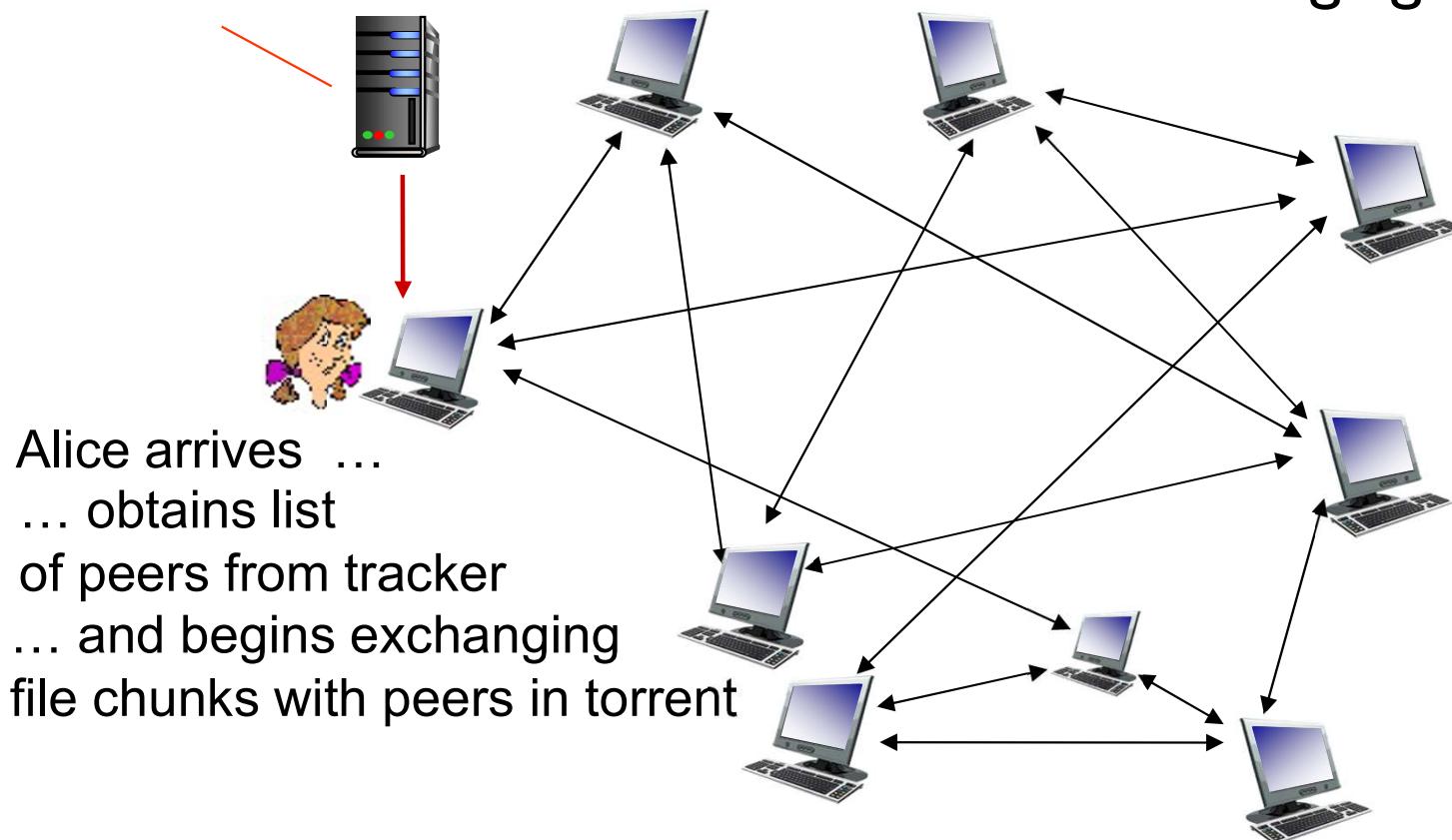


# P2P FILE DISTRIBUTION: BITTORRENT

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

**tracker:** tracks peers  
participating in torrent

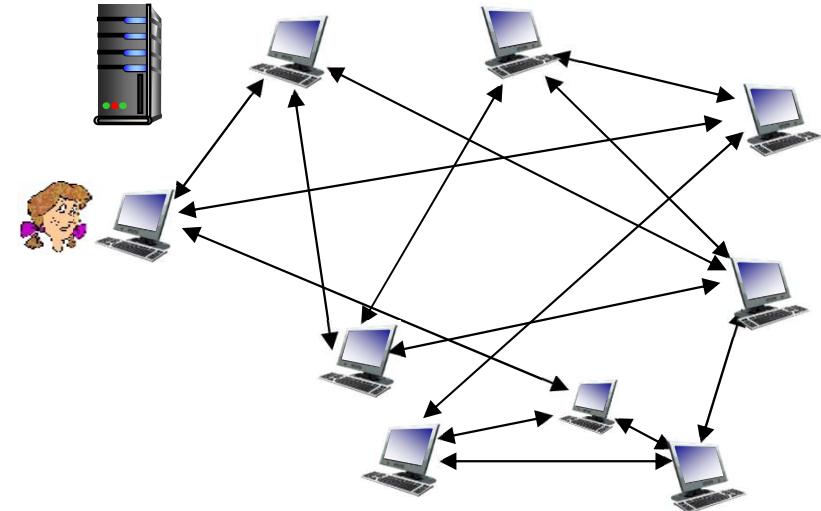
**torrent:** group of peers  
exchanging chunks of a file



# 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:*

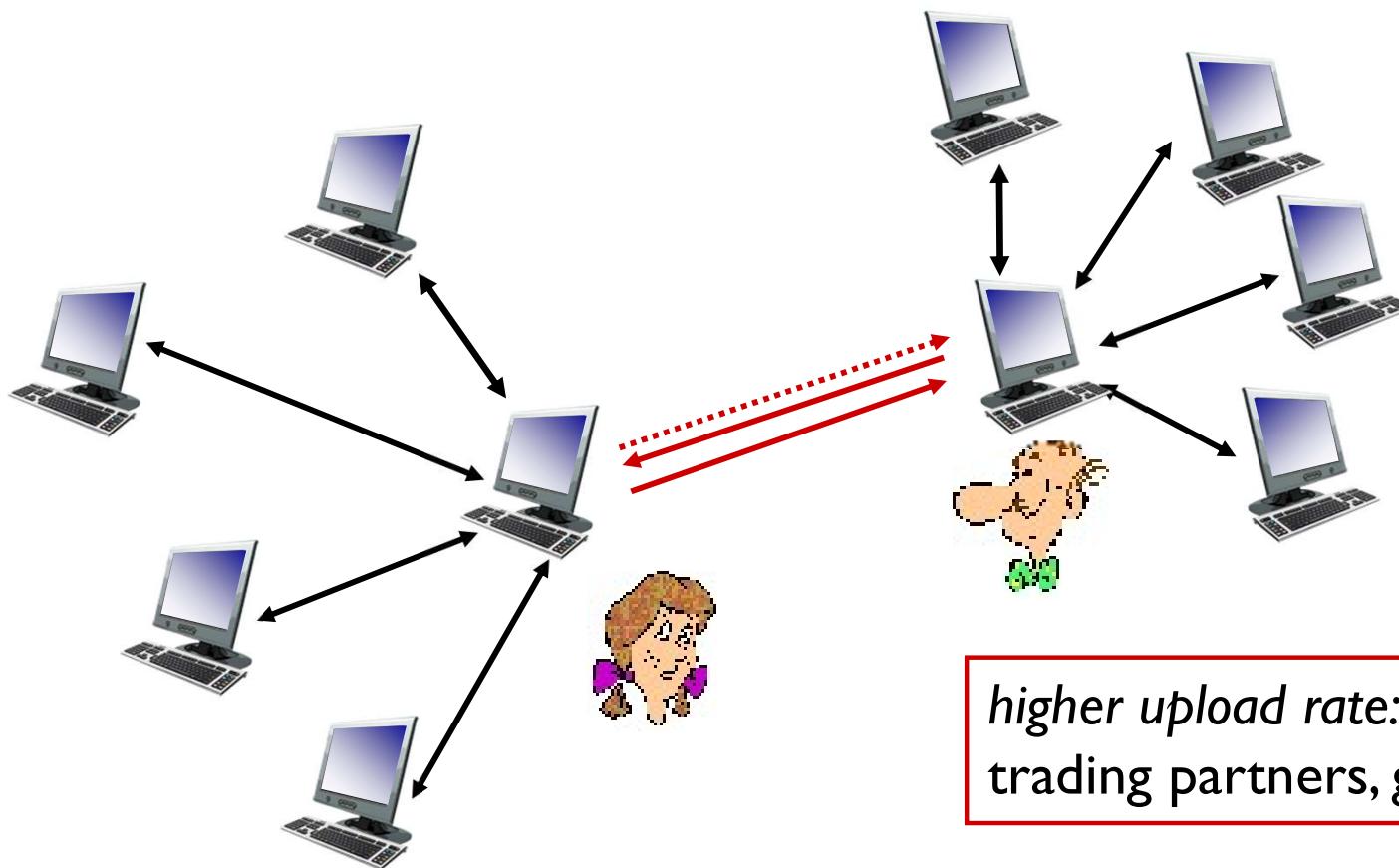
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

## *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

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks (CDNs)

2.7 socket programming with UDP and TCP

# VIDEO STREAMING AND CDNS: CONTEXT

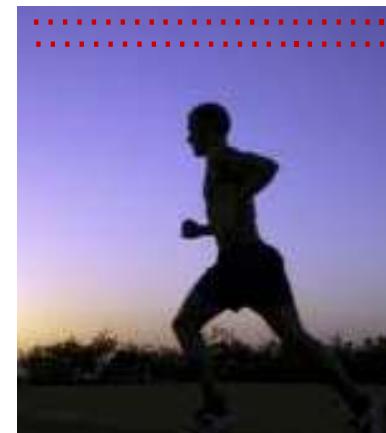
- video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube: 37%, 16% of downstream residential ISP traffic
  - ~1B YouTube users, ~75M Netflix users
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users 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  $i$

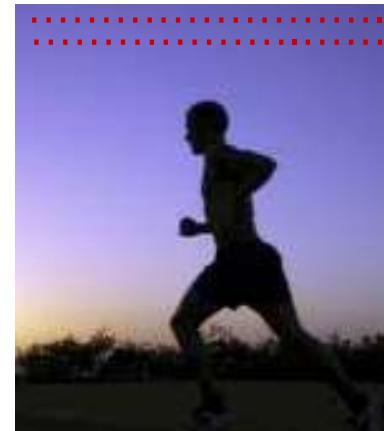


frame  $i+1$

# 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 I (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, < 1 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  $i$

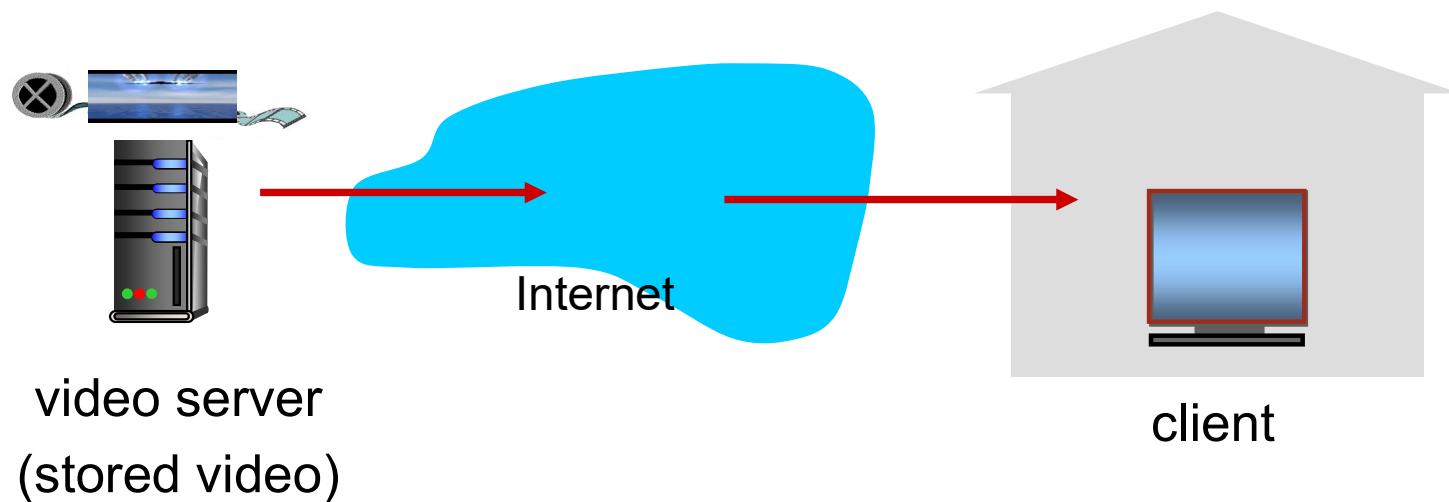


frame  $i+1$

# STREAMING STORED VIDEO:

---

simple scenario:



# CONTENT DISTRIBUTION NETWORKS

---

- *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*

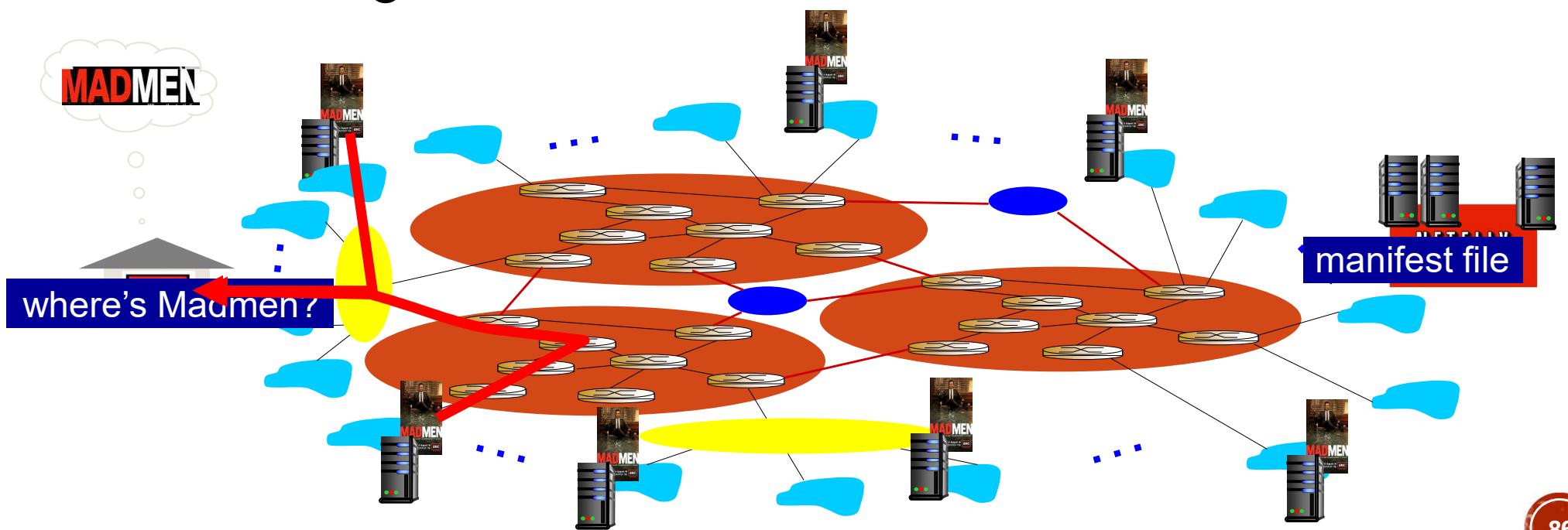
# CONTENT DISTRIBUTION NETWORKS

---

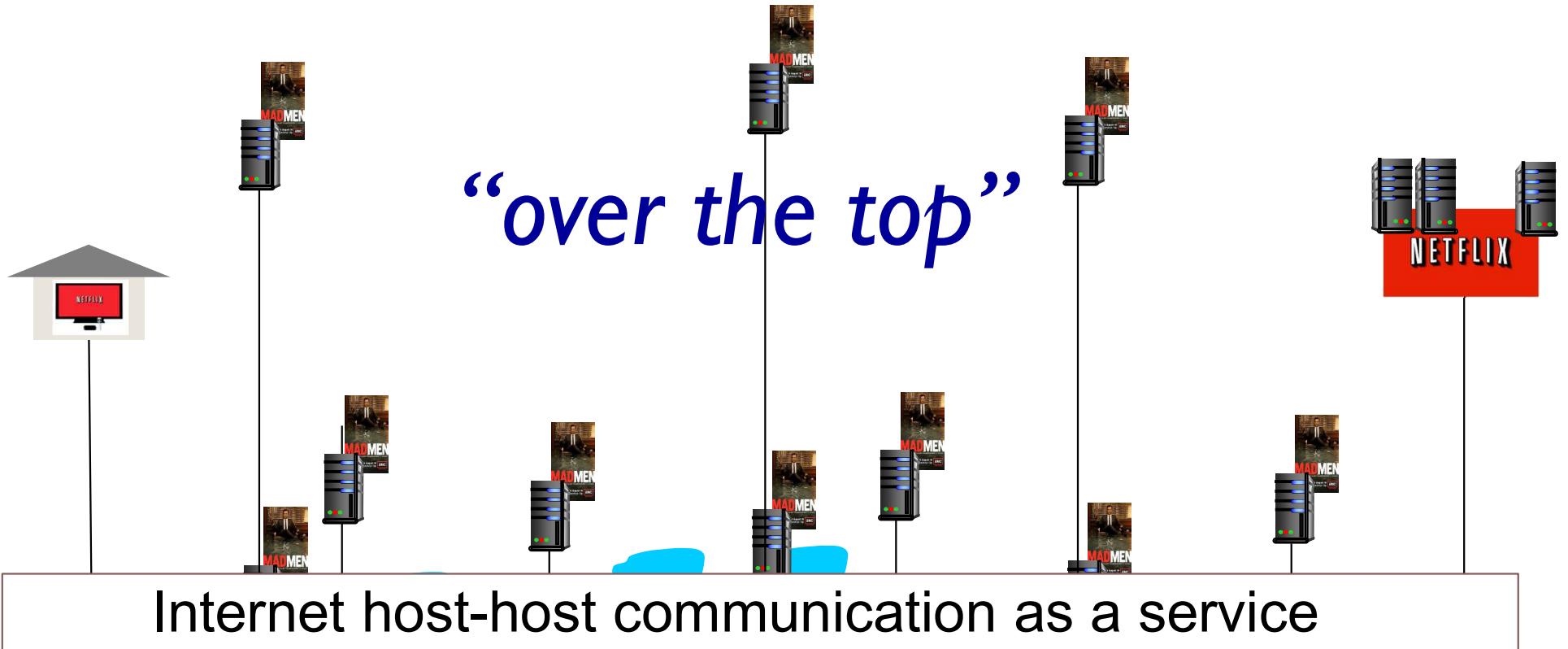
- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?
  
- *option 2:* store/serve multiple copies of videos at multiple geographically distributed sites (*CDN*)
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - used by Akamai, 1700 locations
  - *bring home:* smaller number (10's) of larger clusters in POPs 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)

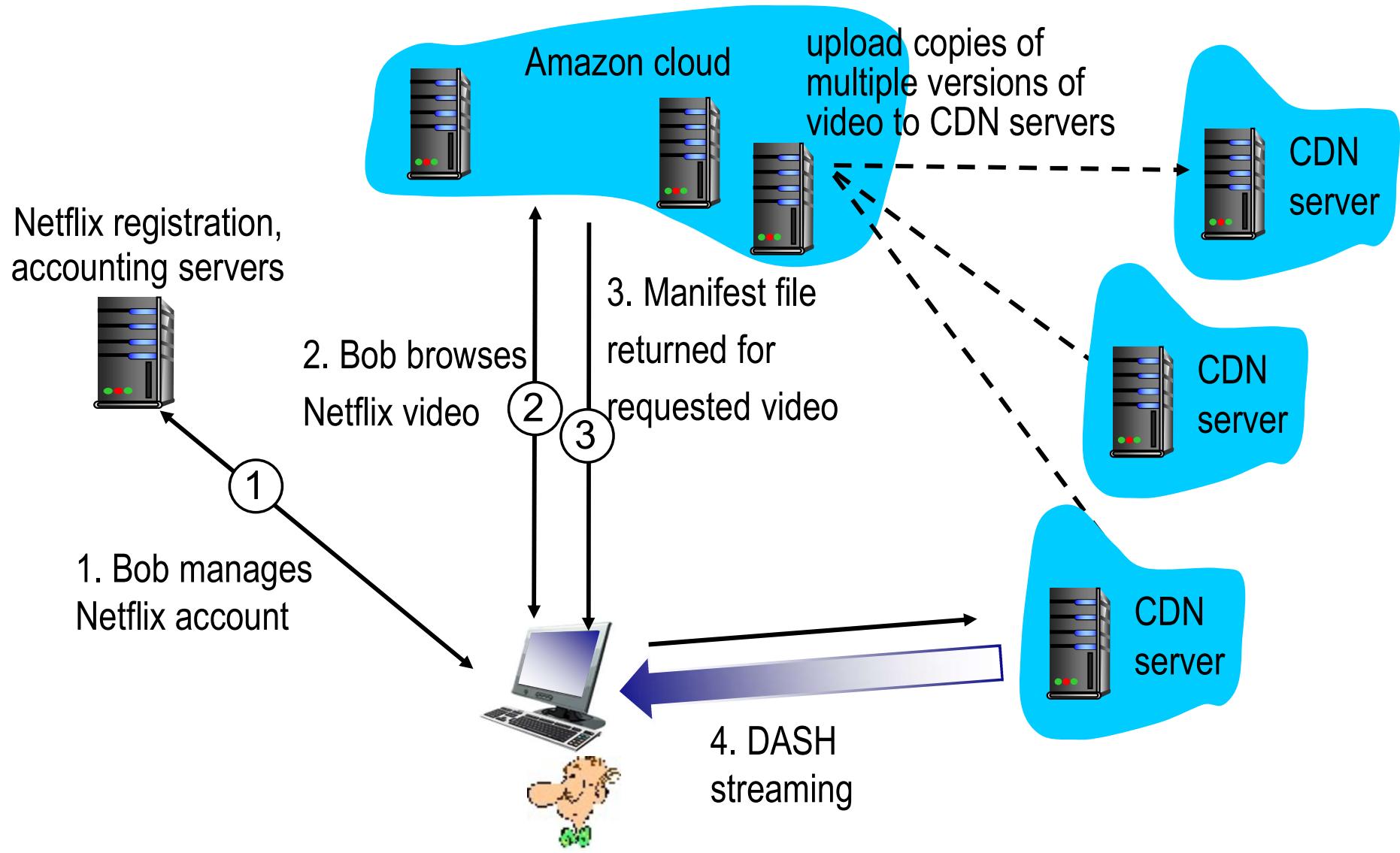


*OTT challenges:* coping with a congested Internet

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

*more .. in chapter 7*

# CASE STUDY: NETFLIX



# 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
  - SMTP, POP, 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:*

- control vs. messages
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable message transfer
- “complexity at network edge”