Chapter 2: Application Layer

Layering in Internet protocol stack

Applications

… built on ...

Reliable (or unreliable) transport

… built on ...

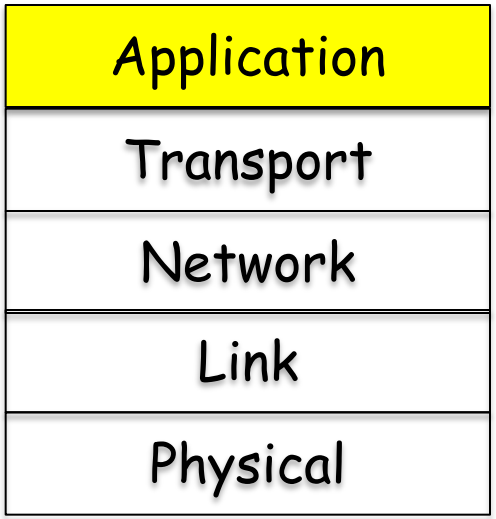
Best-effort global packet delivery

… built on ...

Best-effort local packet delivery

… built on ...

Physical transfer of bits



Modified based on source slide by Dr. Scott Shenker (UC Berkeley)
at The Future of Networking, and the Past of Protocols

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
  - SMTP / POP3 / IMAP
  - DNS
  - P2P
  - Video streaming

- creating network applications
  - socket API

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/TCP in Java~~

(replaced with TA slides)

Question:
What applications do you use in your everyday life?

What do you do over the Internet
in your everyday life?

Some network apps

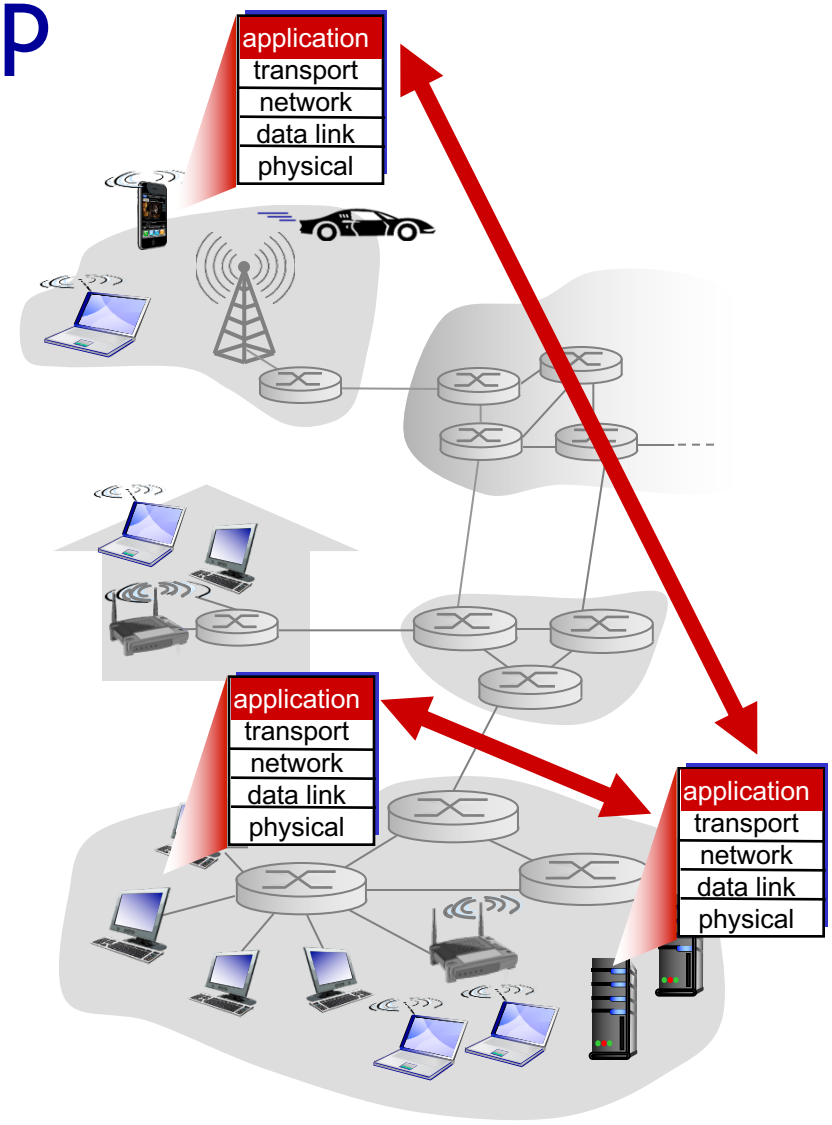
- Search (Google)
- Web
- E-mail
- Video streaming (YouTube, Hulu, Netflix)
- Voice over IP (e.g., Skype, hangouts)
- Social networking (Facebook)
- Multi-user network game
- real-time video conferencing (Zoom, Skype, Facetime),
- text messaging
- …
- remote login
- P2P file sharing
- multi-user network games
- Emerging ones:
  - AR/VR/MR/XR (see video)

Creating a network app

write programs that:

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

no need to write software for network-core devices



Question:
Do all end systems play the the same role?

Example: Use your laptop to do Google Search

Hints:

Role of your laptop

Role of Google Server

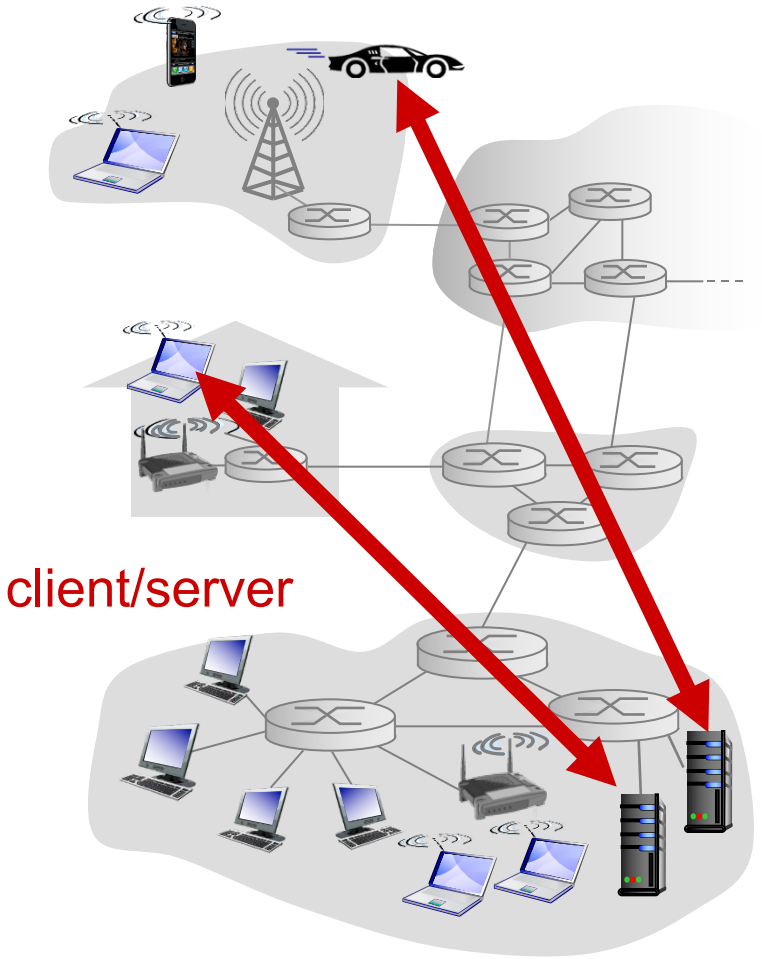
Same or different requirements on the Google Server and your laptop?

Application architectures

possible structure of applications:

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

Client-server architecture



server:

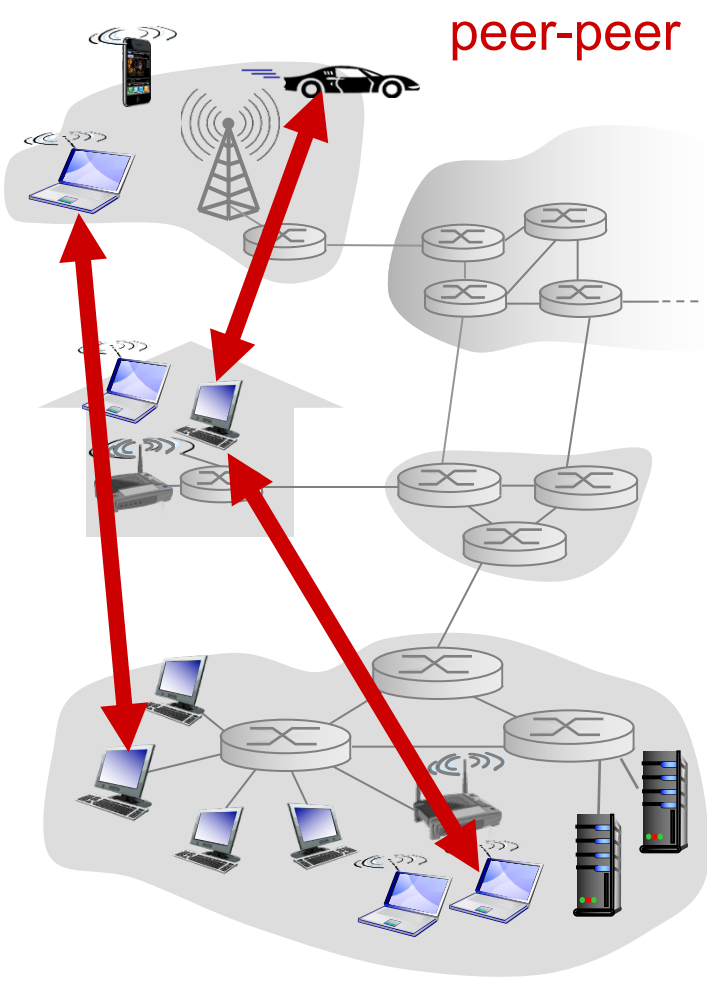
- always-on host
- permanent IP address
- data centers for scaling

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

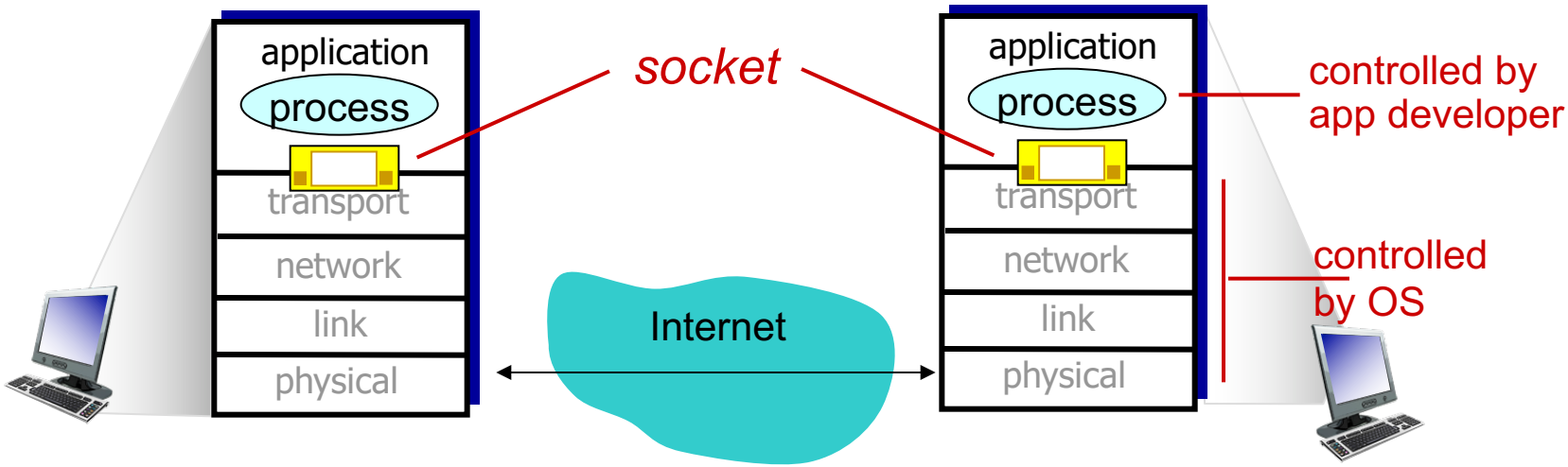
client process: process that initiates communication

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
- Example: to send HTTP message to google.com web server:
  - IP address: 172.217.1.36
  - 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
- e.g., SPDY (Google)

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 is there 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

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

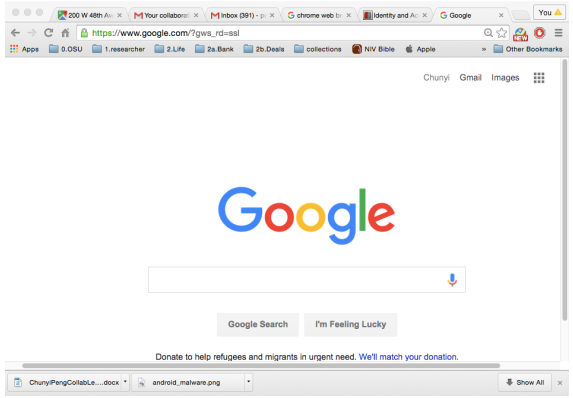
2.7 ~~socket programming with UDP/TCP in Java~~

Web and HTTP

- Most popular or killer application on the Internet!
- Client-server model

PC, phone, pad
running web browser
(chrome, Firefox, Safari, …)

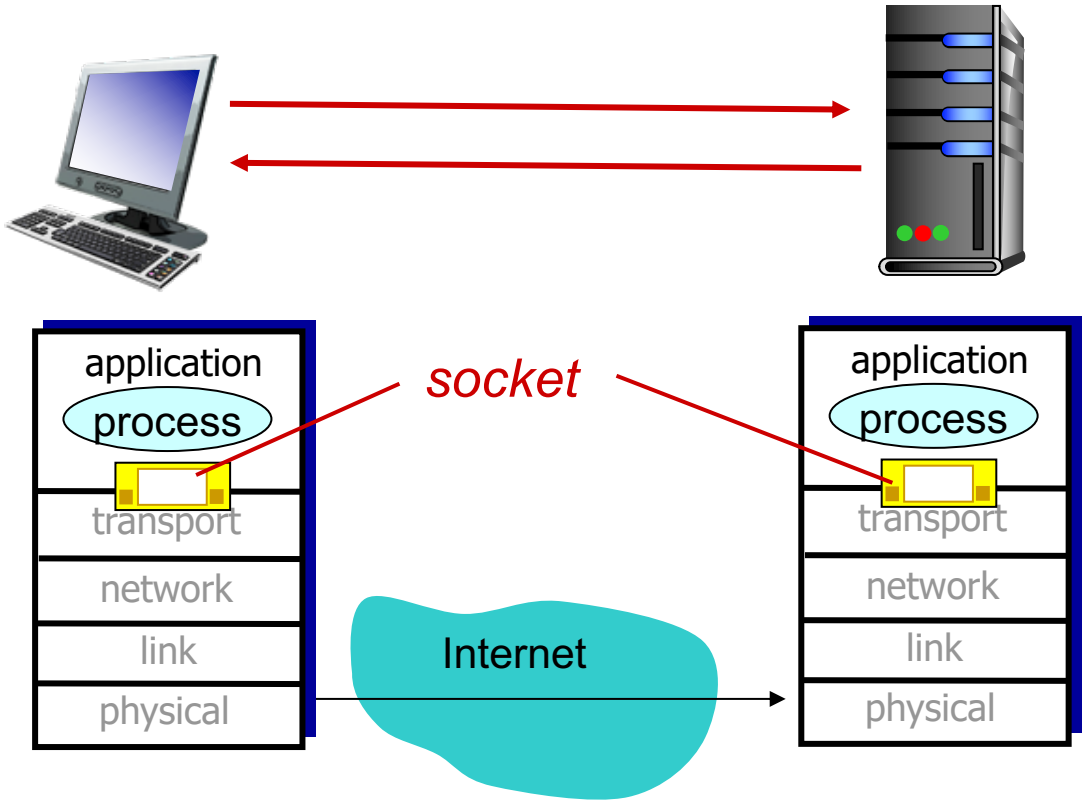
e.g., running
Apache Web
server



Google

4 Questions in Web and HTTP

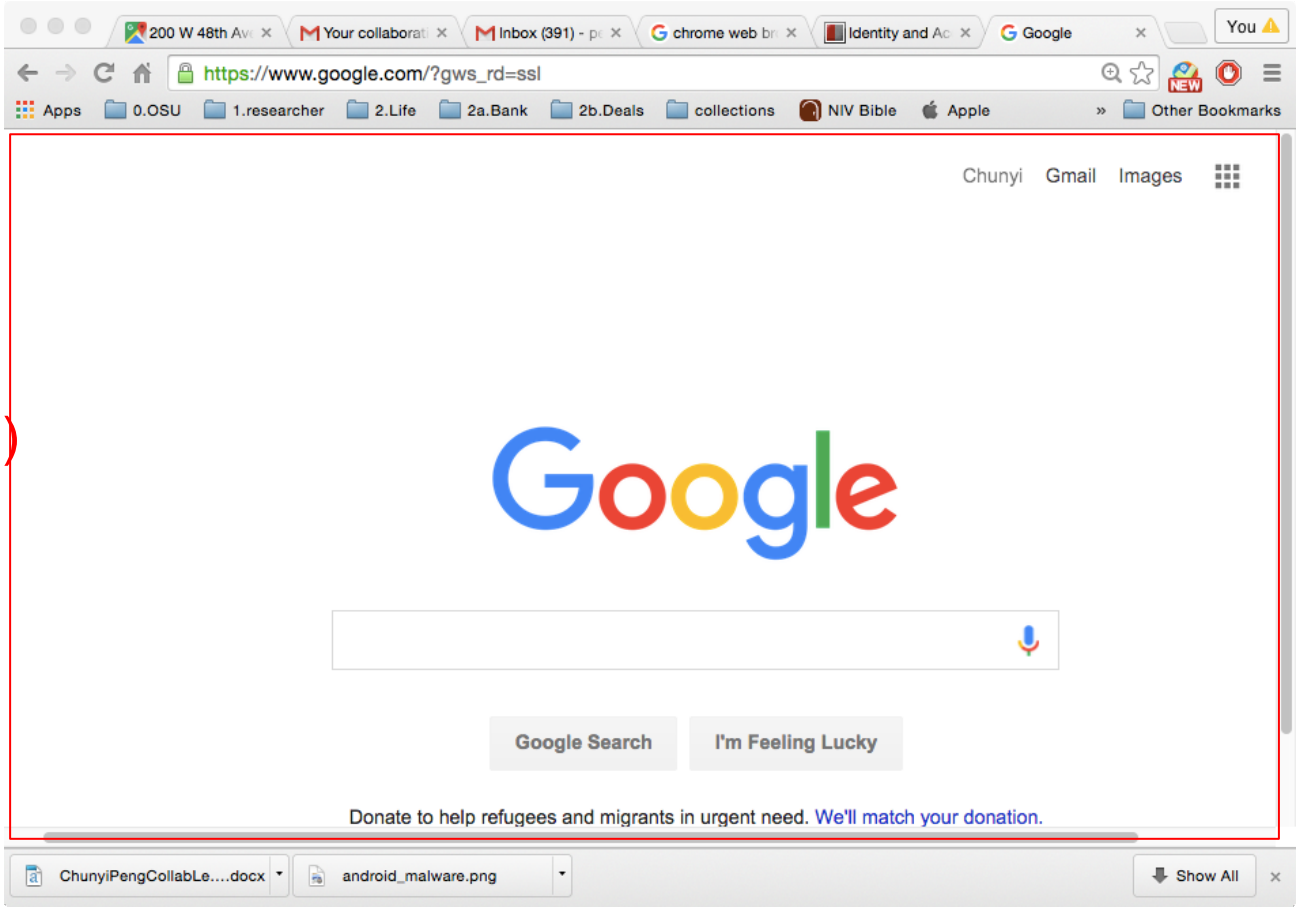
- What content?
  - Web pages
- How to transfer?
  - HTTP connections
- In what messages?
  - HTTP messages
- Advanced features



Web and Web browser

- On the client
  - Loading web pages in Web browser

A Web page
for google.com
(right-click: view source)



Web page source (example)

```
<img alt="Google" height="92" id="hplogo"
src="/images/branding/googlelogo/2x/googlel
ogo_color_272x92dp.png" style="padding-
top:109px" width="272"
onload="window.lol&&lol()">

<div class="_mSc">
Say "Ok Google" to start a voice search.
</div>
<div class="gb_6c gb_ec" … ….”>
Donate to help refugees and migrants in
urgent need.
<a class="gb_dc"
href="https://onetoday.google.com/page/refug
eerelief" rel="nofollow">We'll match your
donation.</a>
</div>
```

Donate to help refugees and migrants in urgent need. We'll match your donation.

Web Pages

- *Web page* consists of *objects*
  - object can be of any file format, such as 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.,

How to transfer?

HTTP: hypertext transfer protocol

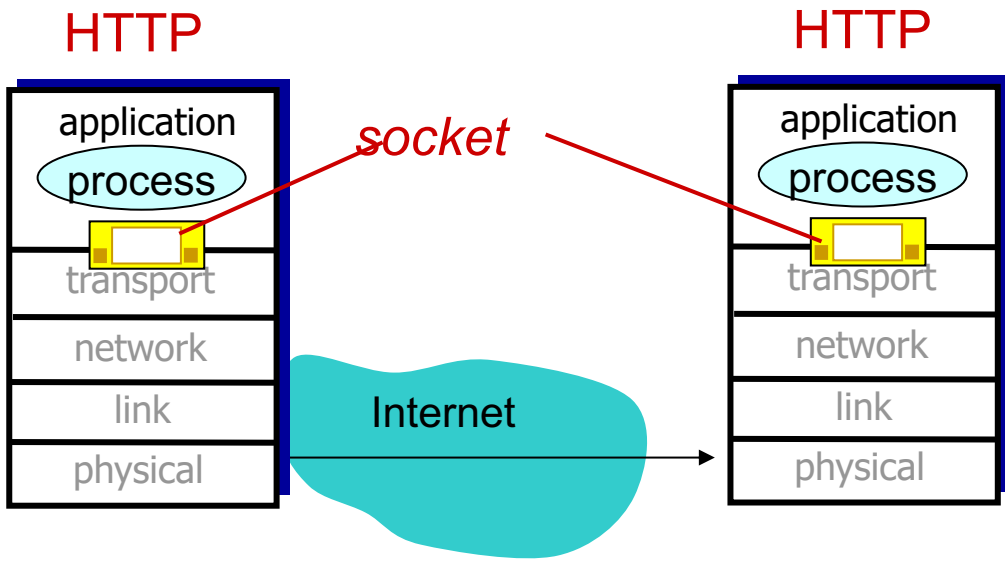
HTTP overview

- 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 a TCP connection (creates socket) to server, port 80
- server accepts the 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

HTTP connections

non-persistent HTTP

- at most one object sent over each 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 overview

Step A: For the base-html file,

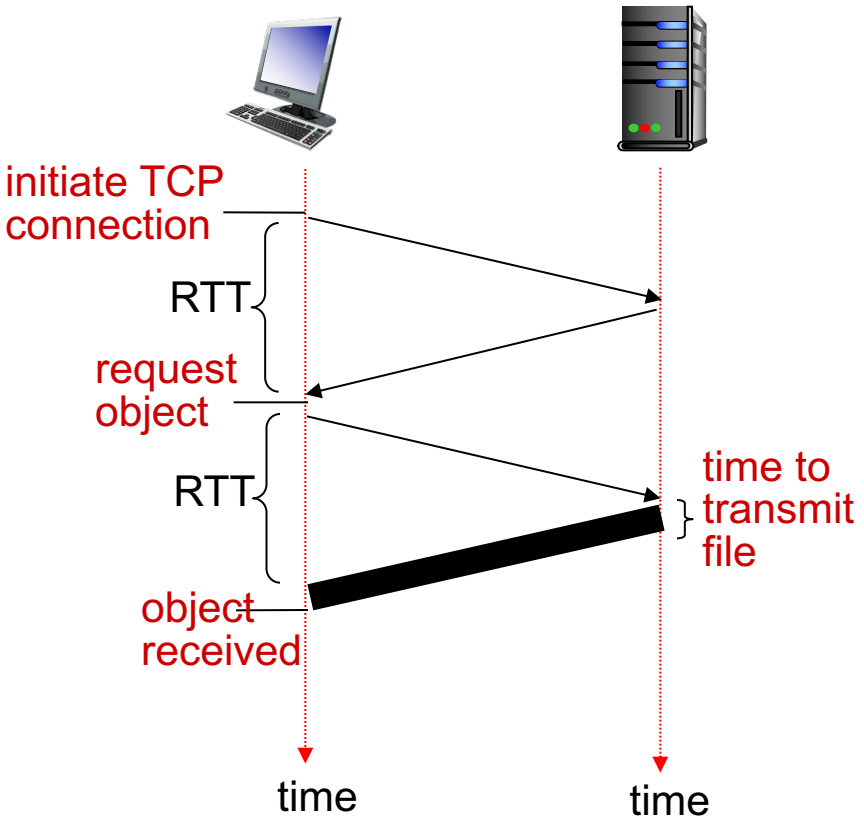
1st RTT (definition of round-trip time): set up TCP connection

2nd RTT: send and receive HTTP request/response for the base-html file

Step B: For each referenced object (find out *all* referenced objects from the base-html file):

1st RTT: set up TCP connection

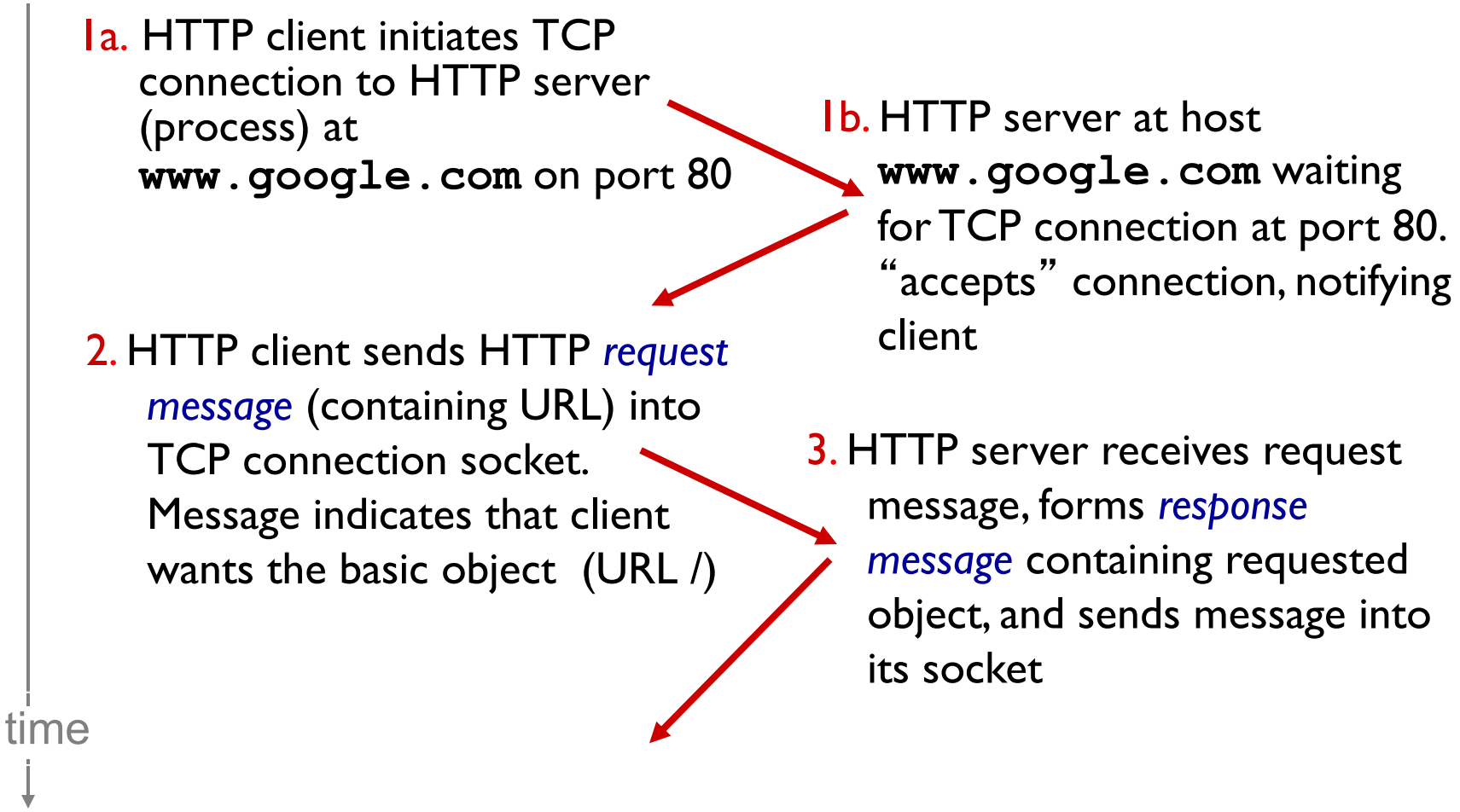
2nd RTT: send and receive HTTP request/response on each object



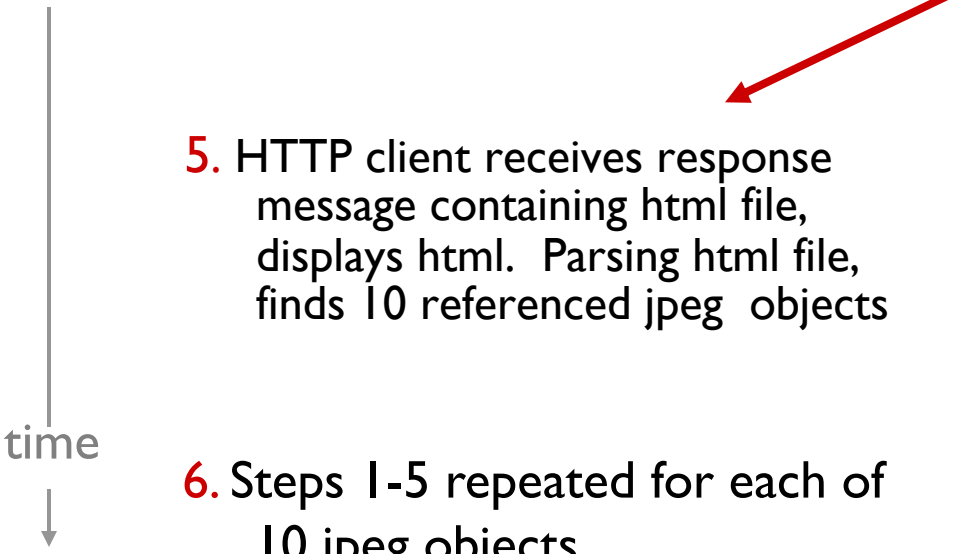
Non-persistent HTTP

suppose user enters URL:
`http://www.google.com`

(Let us assume it contains text, references to 10 jpeg images)



Non-persistent HTTP (cont.)



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

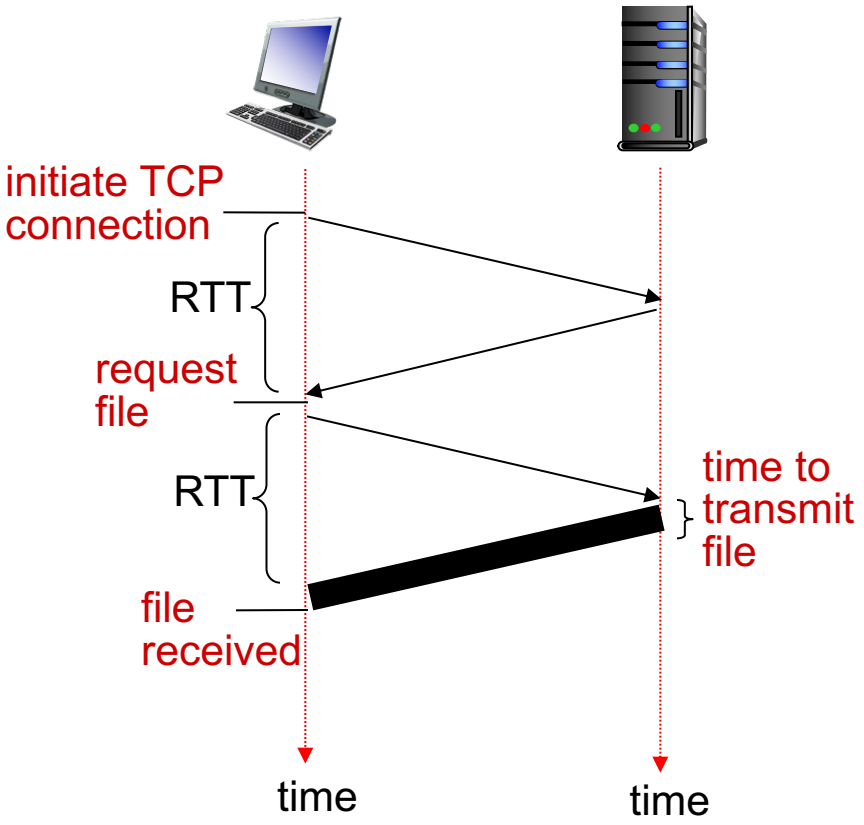
Non-persistent HTTP: response time

RTT (definition): round-trip 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 =

  2RTT+ file transmission time



Parallel non-Persistent HTTP & Persistent HTTP

non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection

Solution #1: parallel TCP connections

- Base HTML: one TCP connection (initial)
- Parallel TCP connections to fetch multiple referenced objects

- ***Limitation***: consume more server (OS) resources

Solution #2: persistent HTTP:

- server leaves connection open after sending response
- subsequent HTTP messages over the open connection
- client sends requests as soon as it encounters a referenced object
- One RTT for each referenced object

Non-persistent HTTP vs. persistent HTTP

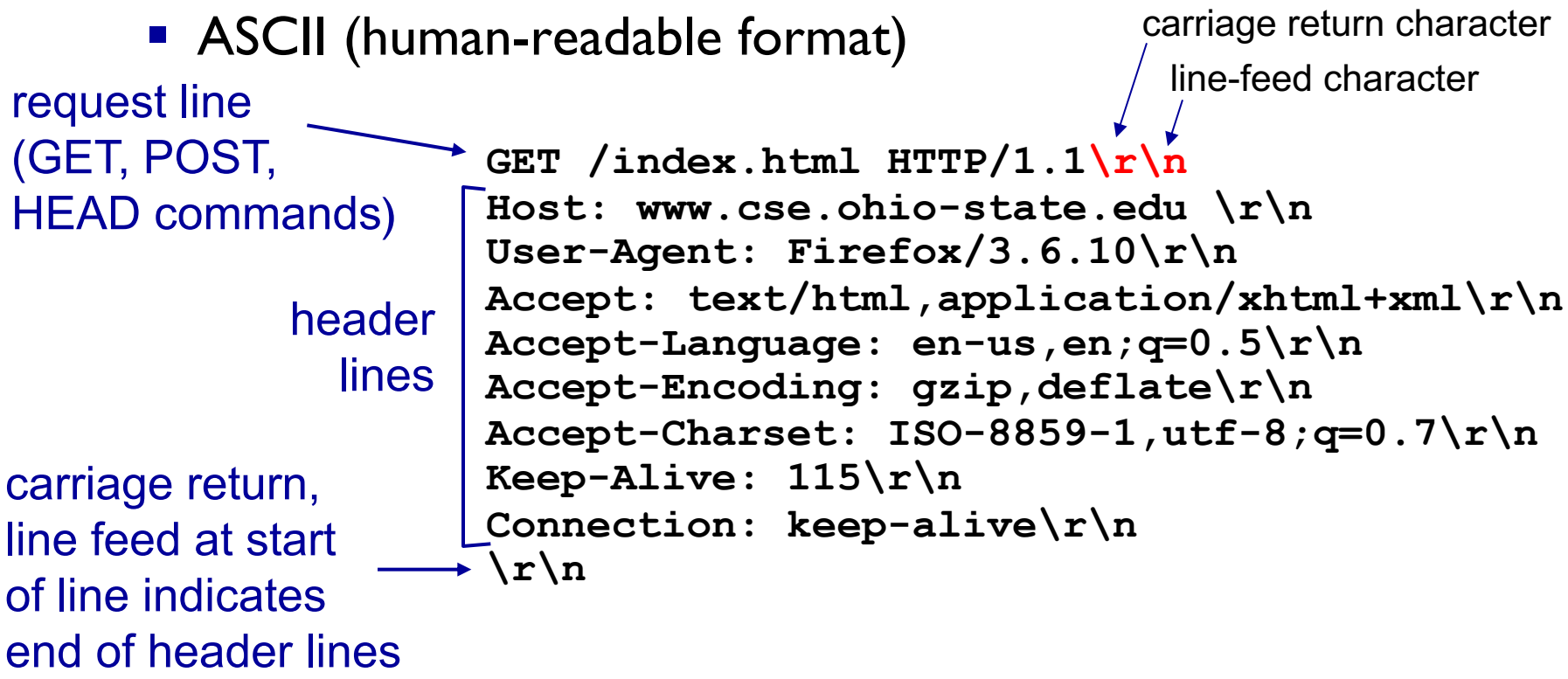
- A web page (base html) with 10 referenced objects. How long to complete all?
  - Ignore the object transmission time
  - non-persistent HTTP (with 1 object one time)
    - 2RTT + 2RTT* 10 = 22 RTT
  - Persistent HTTP
    - 2RTT + 10 RTT = 12 RTT
  - non-persistent HTTP (5 objects in parallel)
    - 2RTT + 2RTT (1~5 objects) + 2RTT (6~10 objects) = 6RTT

In what messages?

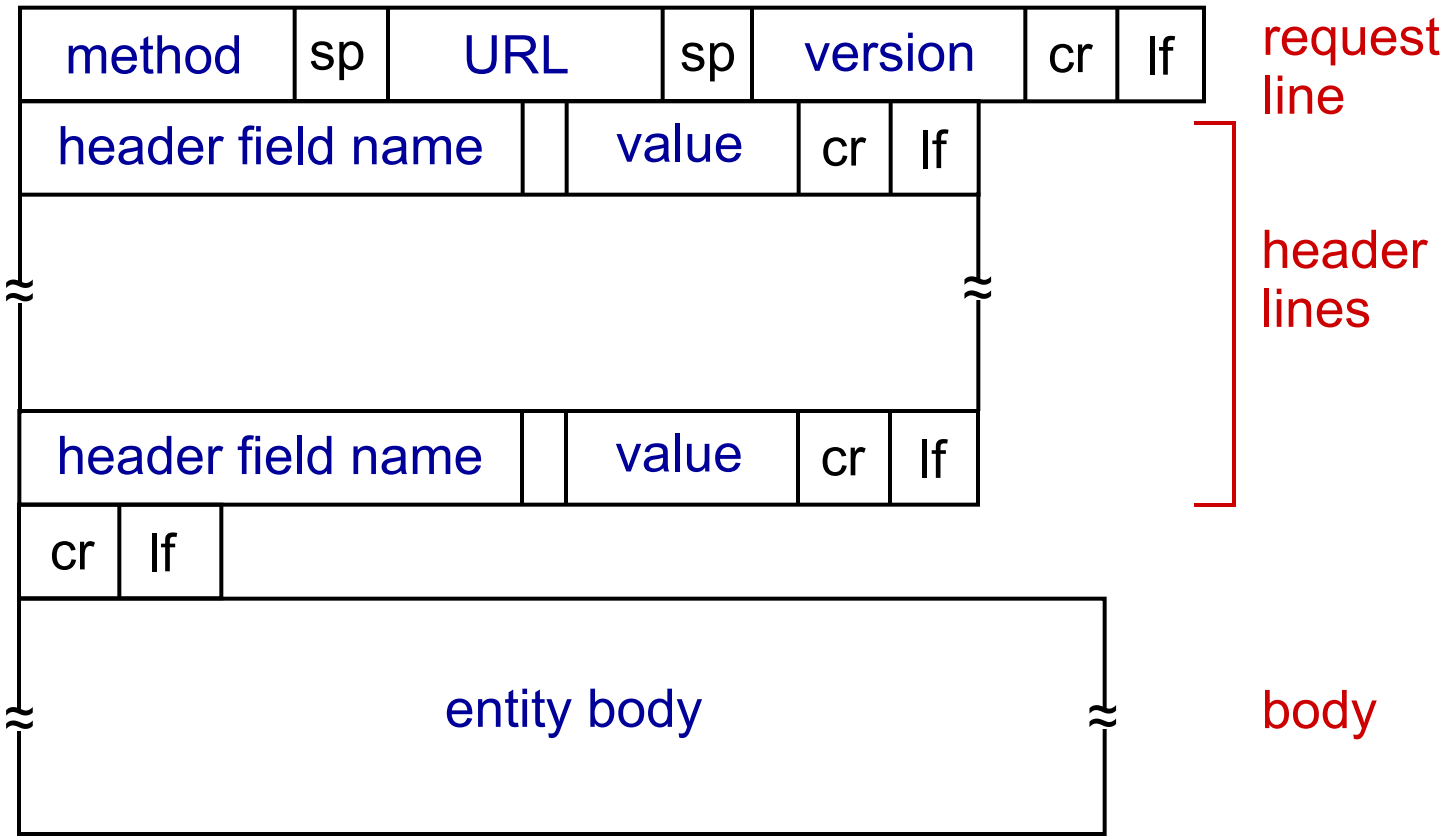
Format and order of messages
in HTTP Protocols

HTTP request message

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



HTTP request message: general format



Method types

POST method:

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

HEAD method:

- Similar to GET method
- Leaves out the requested object

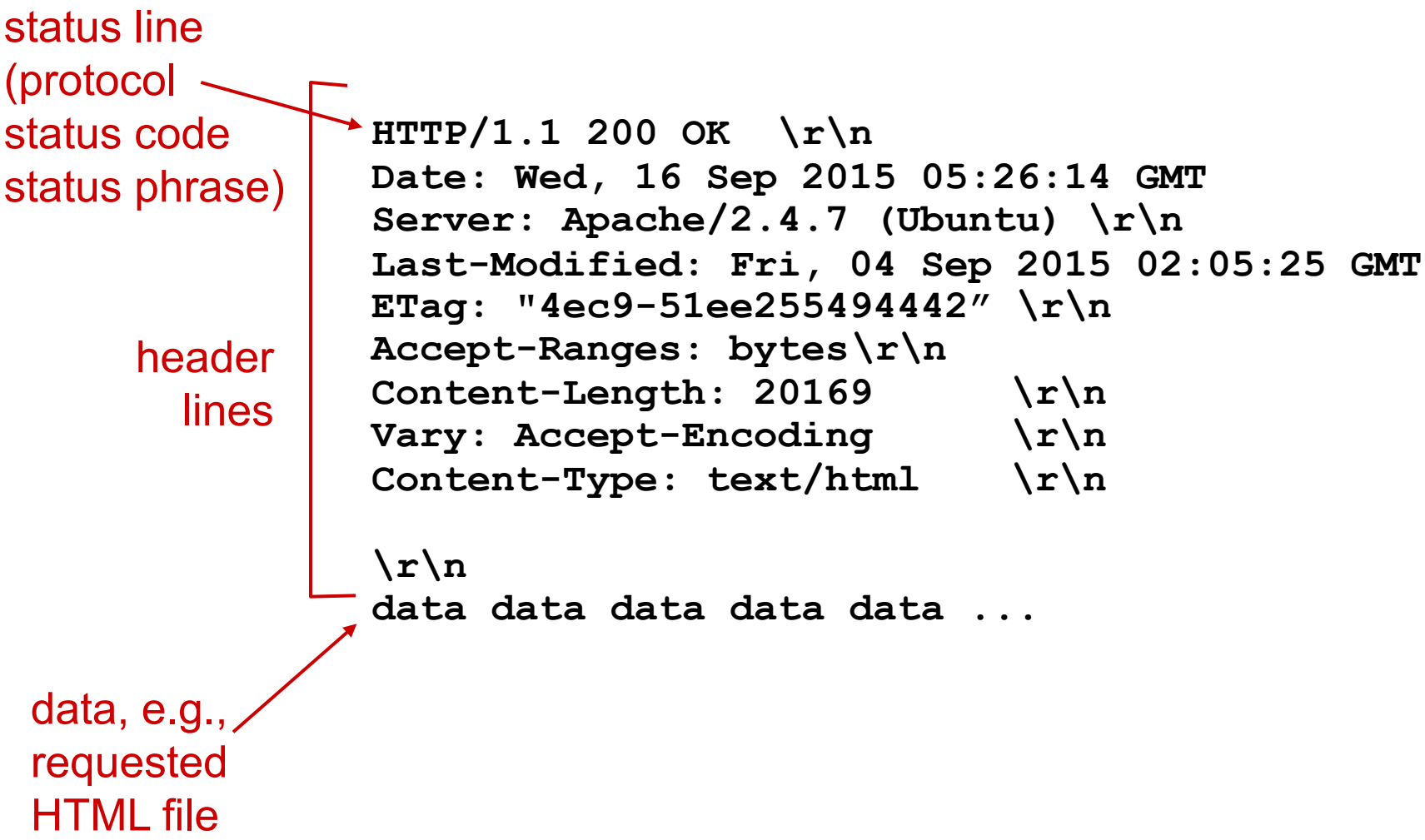
HTTP/1.0:

- GET
- POST
- HEAD

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



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

HTTP/2 (optional)

HTTP/2 (originally named http/2.0):

- next improved version of HTTP (RFC7560)
  - Derived from earlier Google's SPDY
  - 50.6% of all the websites used HTTP/2 (in April 2021)
    - Source: https://w3techs.com/technologies/details/ce-http2
  - 97% of used Web browsers have the capabilities
- designed to improve throughput of client-server connections
- new features:
  - Multiplexing multiple streams over one HTTP2.0 connection; request-response pipelining
  - Header compression
  - Server push (preemptive transfer to clients)
- https://en.wikipedia.org/wiki/HTTP/2

Advanced Features

Questions on basic HTTP

- HTTP is *stateless*
  - The server can't infer whether the client visits her before
- However, it is convenient and friendly to know that
  - Amazon example
- Your experience at Amazon
- Q: How to do that?

- In CS@UCLA, we all (120 clients) want to observe the same video at Youtube
- Basic HTTP: many connections over many communication links
  - Throughput bottleneck
- Your experience at Youtube
- Q: How to solve it?

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

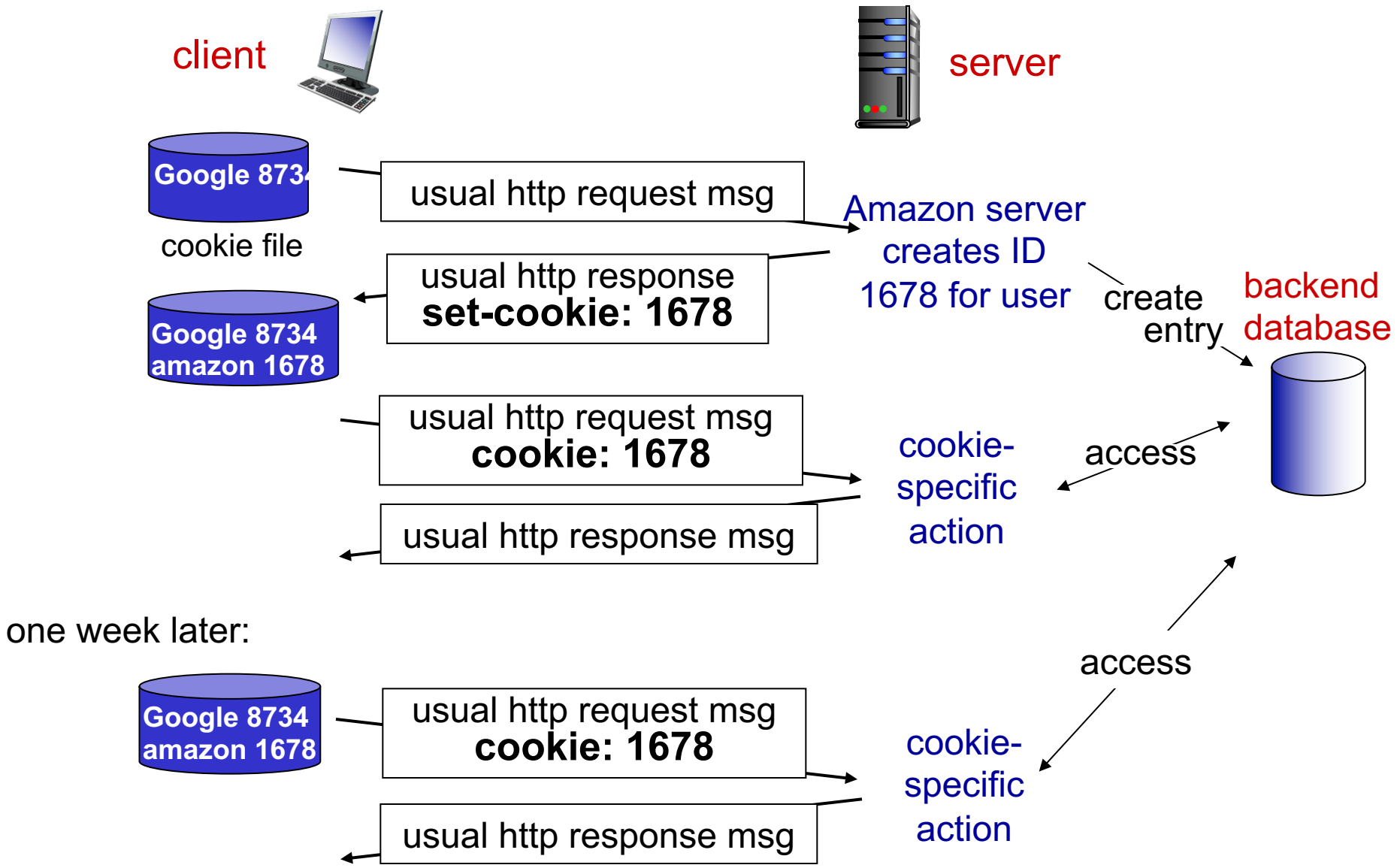
3) cookie file kept on user's host, managed by user's browser

4) back-end database at Web site

Example:

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

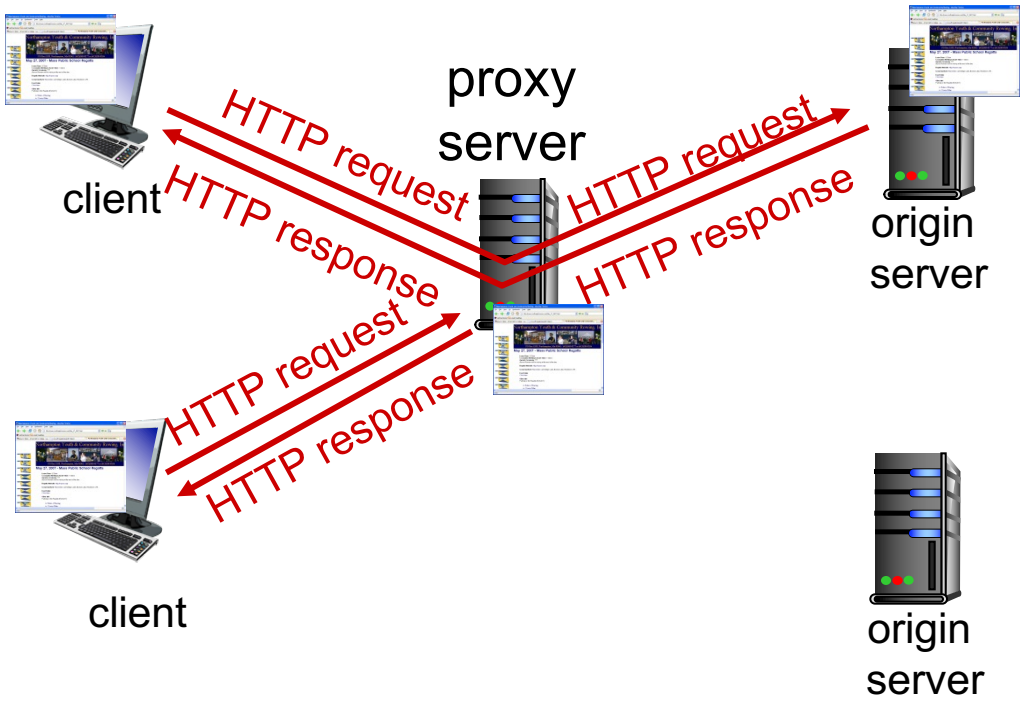
Cookies: keeping "state"



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



Question:
What if the page is updated?

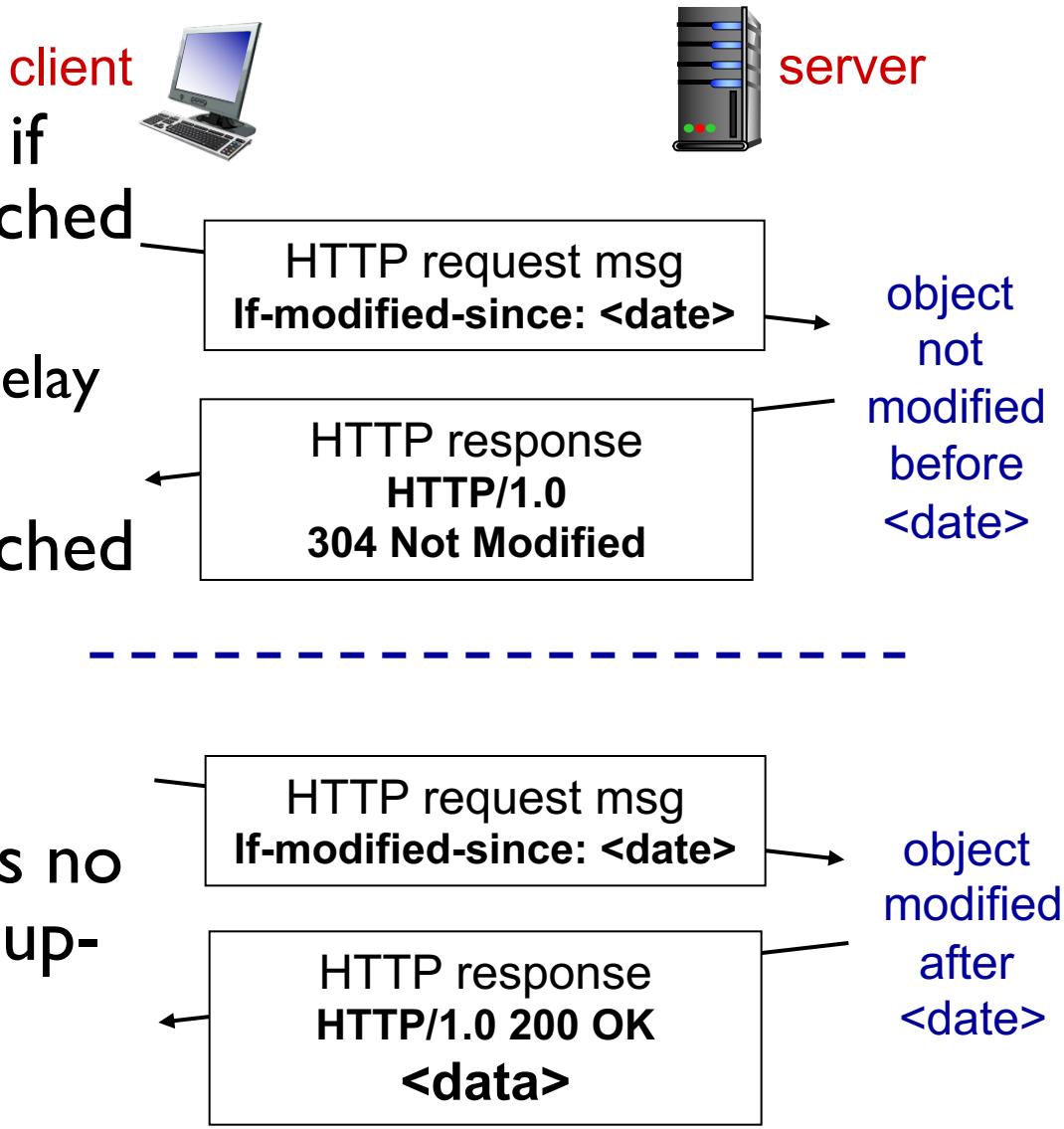
Conditional GET

- *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
```



Summary: Web and HTTP

- Web pages: basic HTML file and objects
- Client-Server model, TCP
- HTTP connections
  - Persistent connection
  - non-persistent connection
  - Non-persistent connection with parallel TCP connections
- HTTP messages
  - Request: GET
  - Response: 200 OK, 404 etc
- Advanced features:
  - Cookie
  - Cache

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 in Java~~

Question

- Client-Server application architecture
  - Web (HTTP)
    - Example: Alice does a google search
    - Client: Alice's host (laptop)
    - Server: Google
  - Other apps: comparison with web
    - Email
    - Ex: Alice (gmail) sends an email to Bob (UCLA)
    - Client: ?
    - Server: ?

Electronic mail (Email)

Two components:

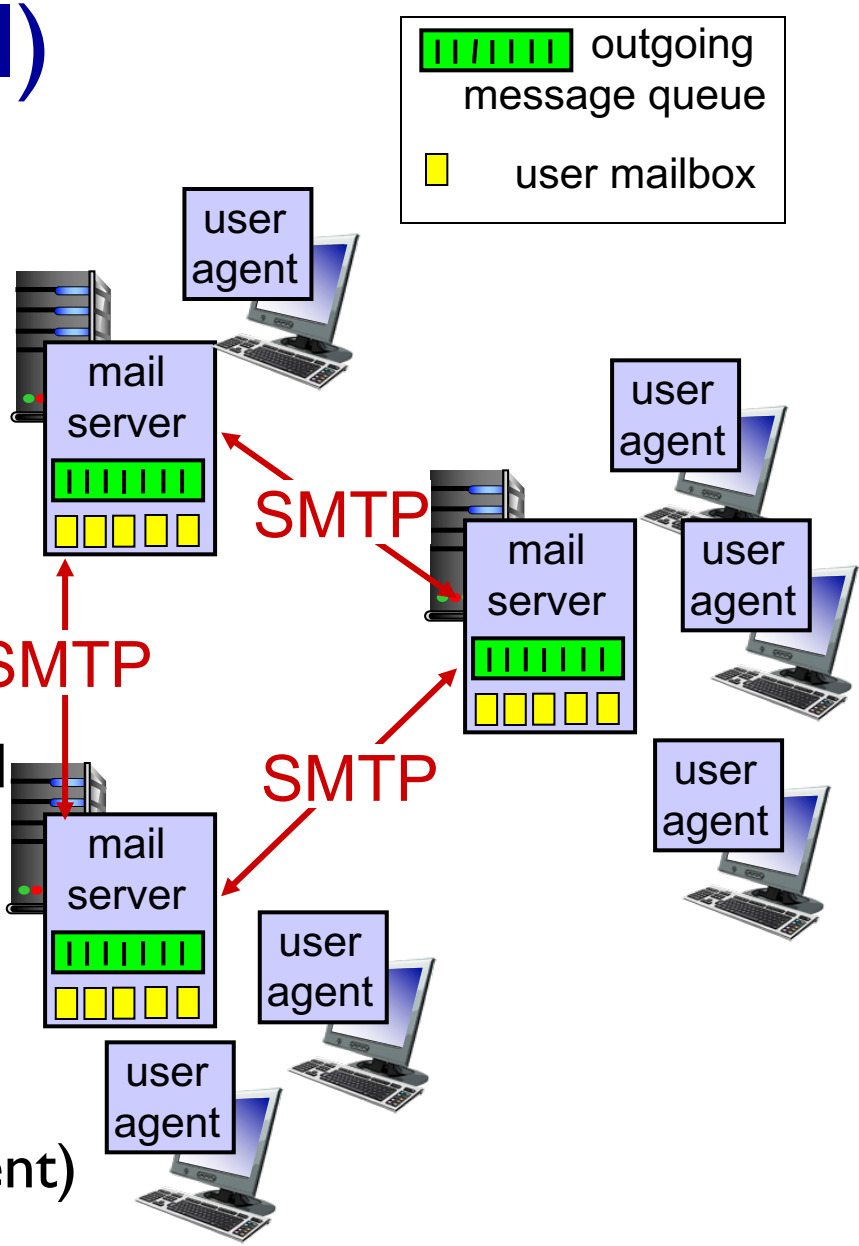
- user agents
- mail servers

User Agent: a.k.a. "mail reader"

- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server

mail servers:

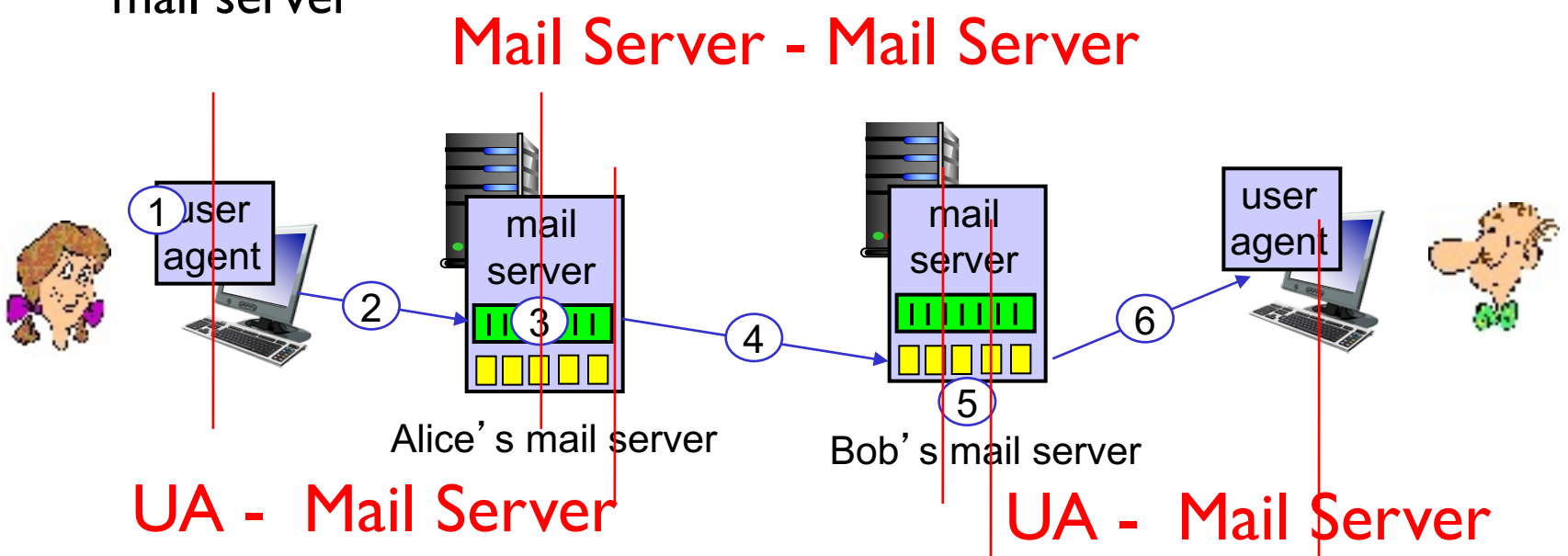
- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages



How is an email sent out?

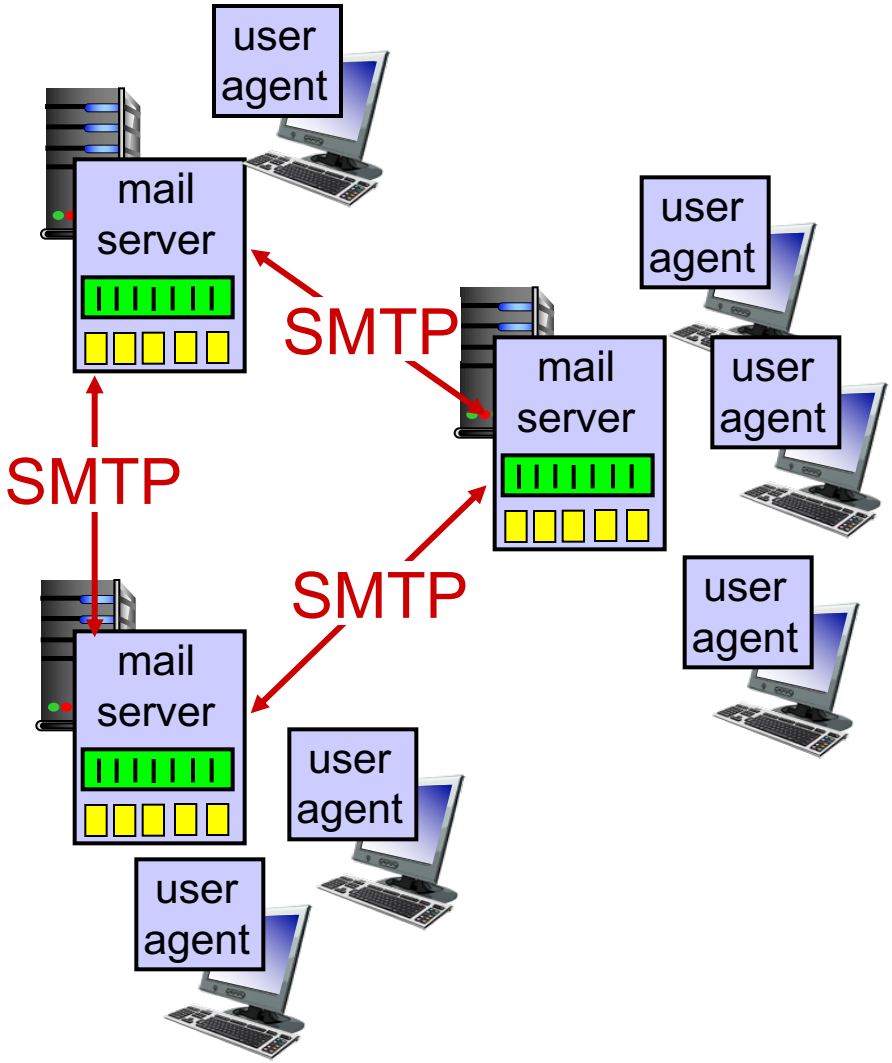
Scenario: Alice sends message to Bob

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) Alice's mail server opens TCP connection with Bob's mail server
4) Alice's mail server 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



Simple Mail Transfer protocol: SMTP

- *SMTP protocol* between mail servers to send email messages
  - "client": sending mail server
  - "server": receiving mail server



Email: 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

Sample SMTP interaction

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

SMTP Messages

- Requests:
  - HELO,
  - MAIL FROM,
  - RCPT TO,
  - DATA,
  - QUIT
- Responses, e.g.,:
  - **`250 XXX ok`**
  - **`221 XXX closing connection`**

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
- HTTP: ASCII command/response interaction, status codes
- SMTP: ASCII for everything (incl. msgs)

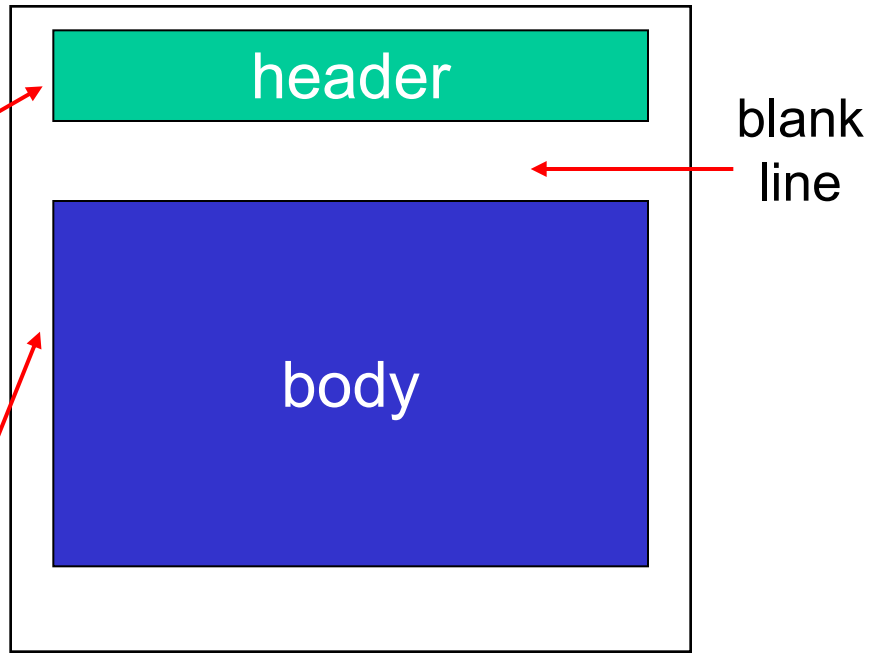
Mail message format

RFC 822: standard for text message format:

SMTP: protocol for exchanging email msgs

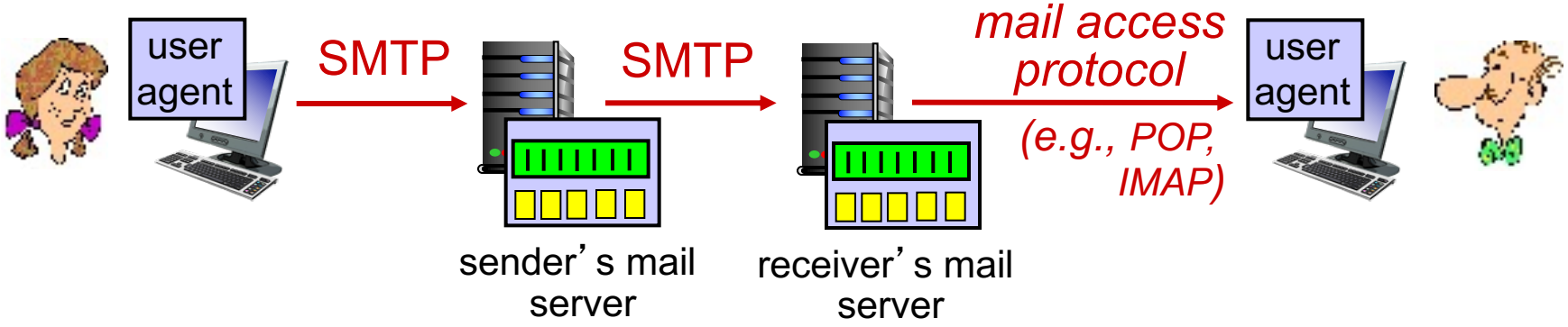
- Header lines, e.g.,
  - To: xxx@purdue.edu
  - From: xxxx@purdue.edu
  - Subject:: cs422….

  different from SMTP MAIL FROM, RCPT TO: commands!
- Body: the "message"
  - ASCII characters only



Q: Are email messages the same as protocol (SMTP) messages?

Mail access protocols



- SMTP: delivery/storage to receiver's server
- mail access protocol: retrieval from server
  - POP: Post Office Protocol authorization, download
  - IMAP: Internet Mail Access Protocol
  - more features, including manipulation of stored msgs on server
  - HTTP: Gmail, Yahoo! Mail, OFFICE365, 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`**

update phase, server:

- e.g., delete

```
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 2 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

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/TCP in Java~~

DNS: Domain Name System

Basic function: find the IP address, given the host name (name → IP address translation)

- #1: Why DNS?
- #2: How DNS works?
  - High-level principle and design choices
  - Details in DNS (focus on key techniques only)

Why DNS?

- Application: host-to-host, process-to-process communication
  - Process identifier: IP address and port number
  - Example-1: http://173.194.204.99:80
  - Example-2: http://176.32.103.205:80

- But, how can we know and remember the destination IP address?
  - Google?
  - Amazon?
  - Facebook?

Why DNS? (cont'd)

Internet hosts, routers:

- "name", e.g., www.google.com - used by humans
- IP address (32 bit) - 173.194.204.99, used for addressing datagrams, used by routers

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

DNS: Essential service

- hostname to IP address translation
  - *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: Other Services

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

DNS: Domain Name System

- #1: Why DNS?

- #2: How DNS works?
  - What is the DNS (database) structure?
  - How to query an entry?
  - How to cache and update an entry?
  - What is a DNS record format?
  - DNS protocol

DNS structure

Design question #1: why not a centralized database?

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

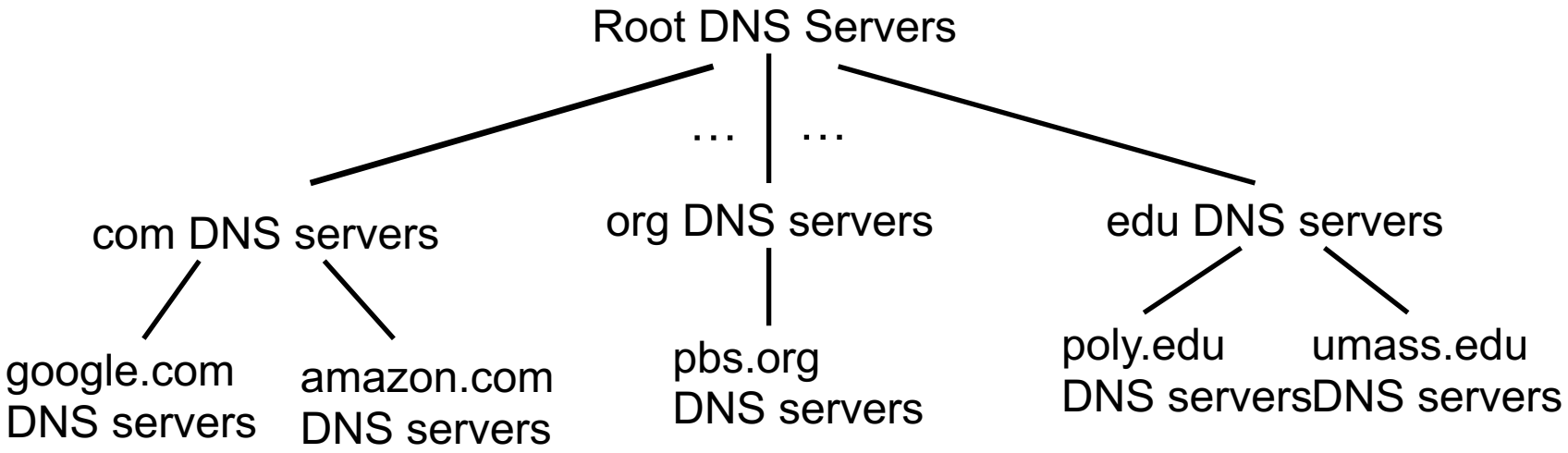
A: doesn't scale!

distributed database implemented in hierarchy of many *name servers*

Key design concepts in DNS

- Names are hierarchical
  - No flat names (e.g., university_ucla_cs_kiwi) for hosts
  - Hierarchical names (e.g., kiwi.cs.ucla.edu)
    - Highest hierarchy: edu, com, gov, org, … us, jp, fr, ..
    - Next-level hierarchy: ucla, mit, ..google,ibm, ...ca, il, …
  - Hierarchical names form the name space hierarchy
- Name servers (that resolve names) are also organized into hierarchy
  - Each name server handles a small portion of the name space hierarchy
- Name resolution follows the hierarchy to resolve names to IP addresses

DNS: a distributed, hierarchical database

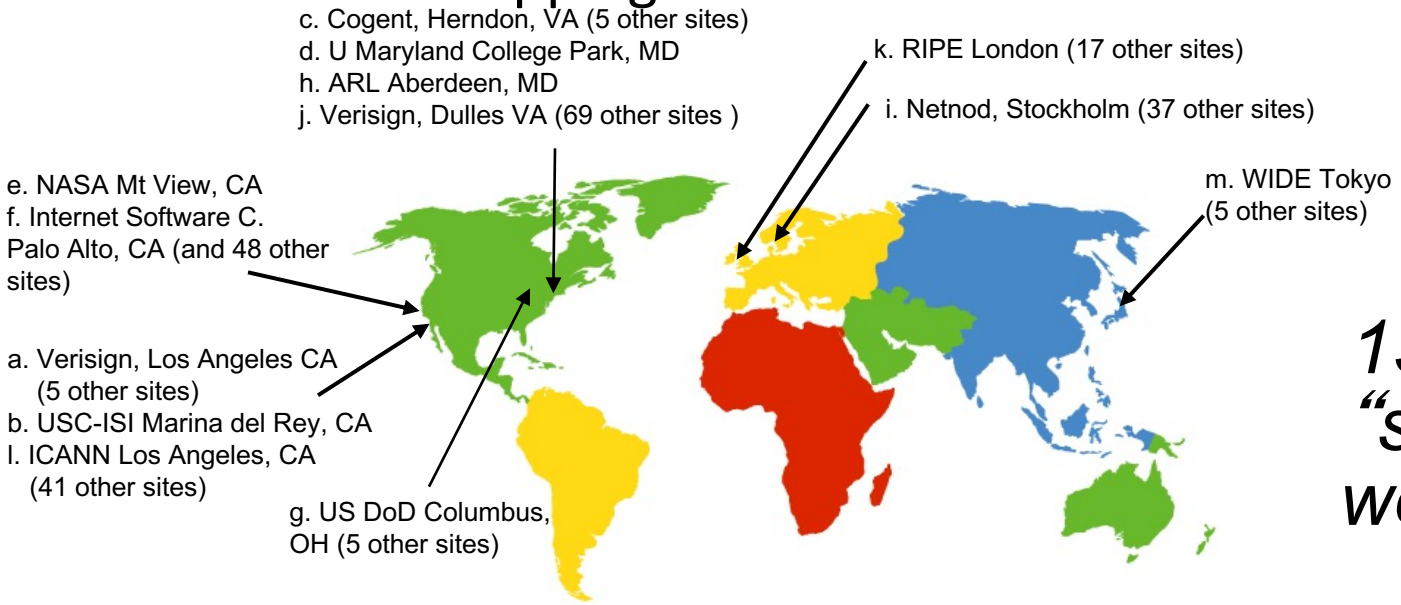


client wants IP for www.amazon.com; 1st approx:

- 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 unknown
  - 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

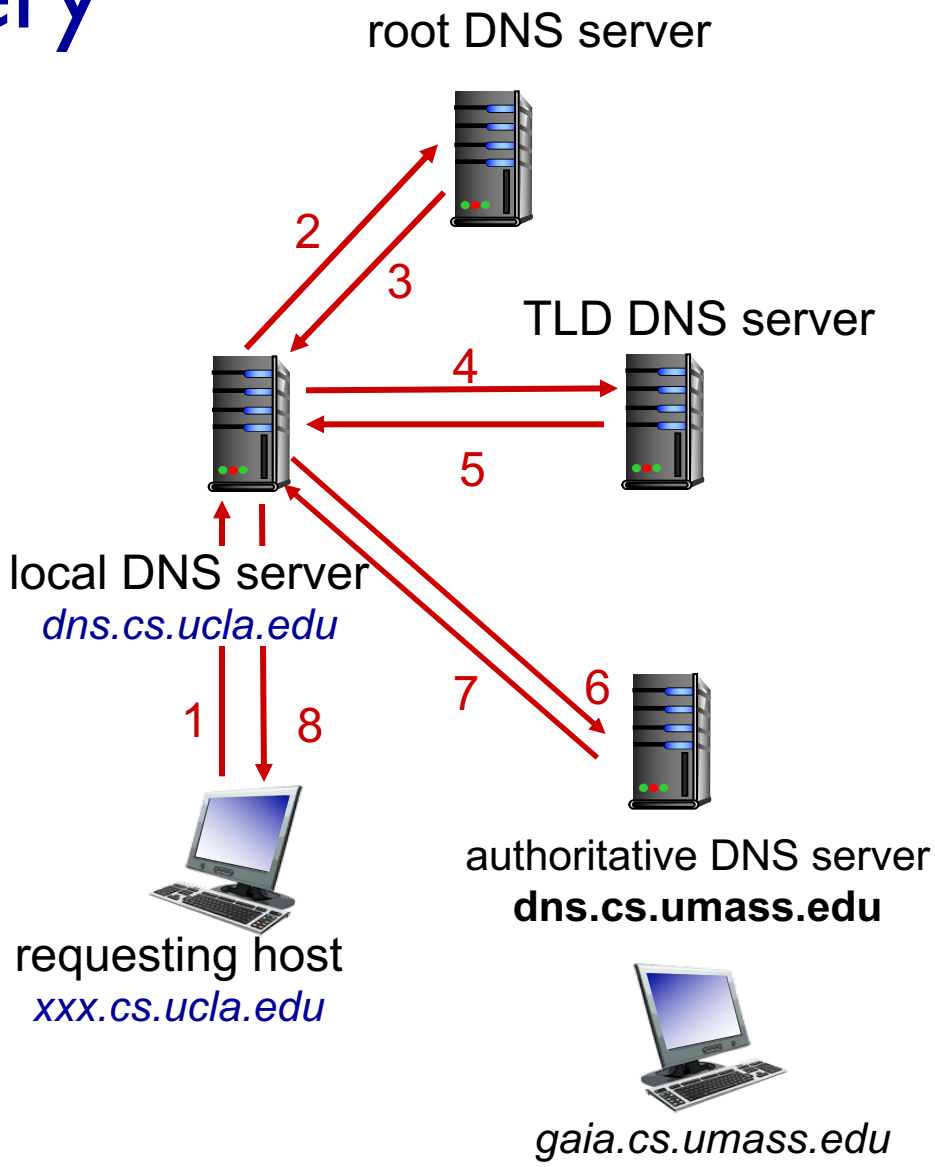
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

How to query?

Two resolution approaches

DNS iterated query example

- host at cs.ucla.edu wants IP address for gaia.cs.umass.edu



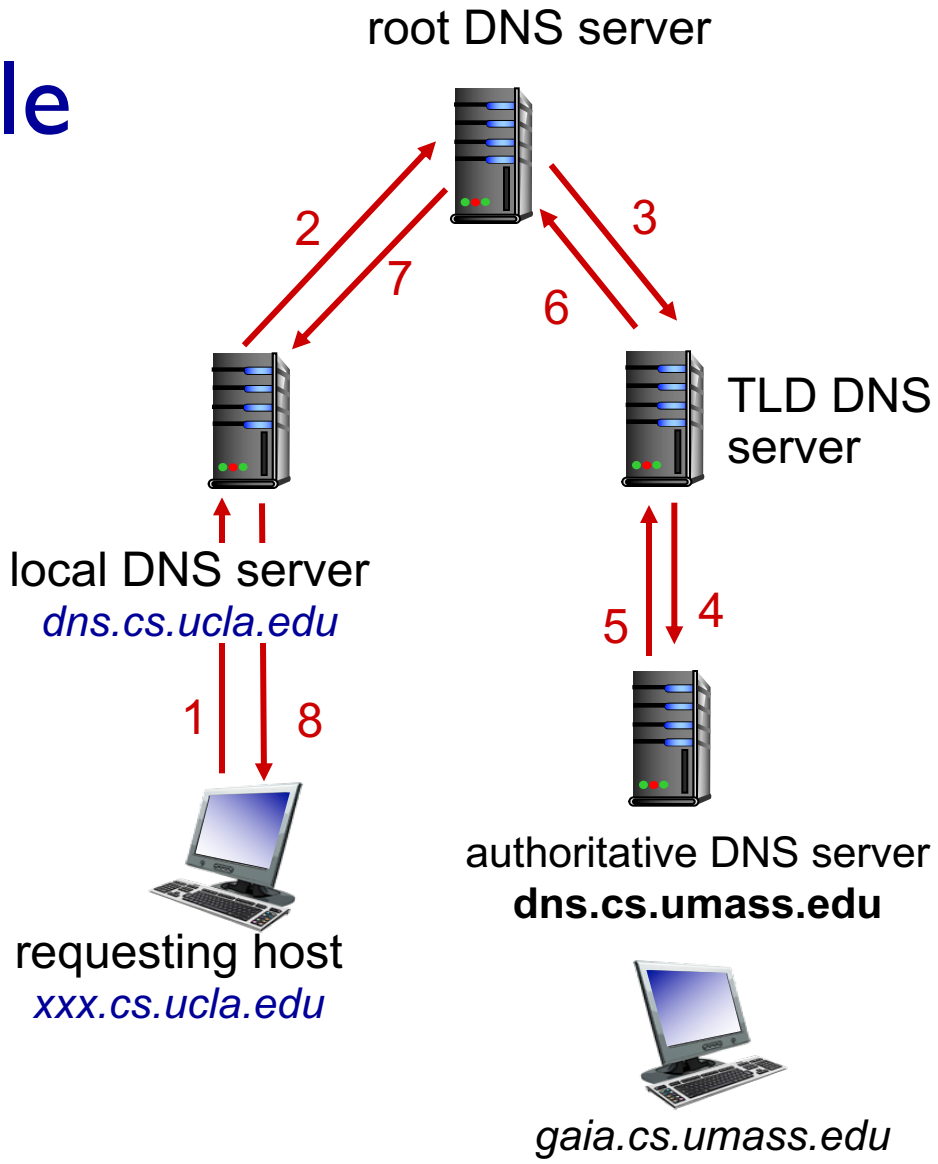
iterated query:

- 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:

- Higher-level servers provide query answers
- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



DNS caching

Question: Is it needed to repeat the whole query procedure every time upon a query request?

- Heavy load
- Long response time
- Unnecessary in most cases

A: bad performance, not necessary

DNS caching in hierarchy of many *name servers*

DNS: caching, updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (time-to-live, TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited

Q: what is the problem with caching?

- 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**

Try DNS yourself (nslookup)

(Can also use the online site: https://centralops.net/co/NsLookup.aspx)

> nslookup amazon.com

>nslookup cs.ucla.edu

Authoritative name servers

>nslookup -type=NS cs.ucla.edu

> nslookup –type=A cs.ucla.edu

Canonical name (alias)

>nslookup -type=CNAME amazon.com

>nslookup -type=CNAME ucla.edu

Mail server

>nslookup -type=MX gmail.com

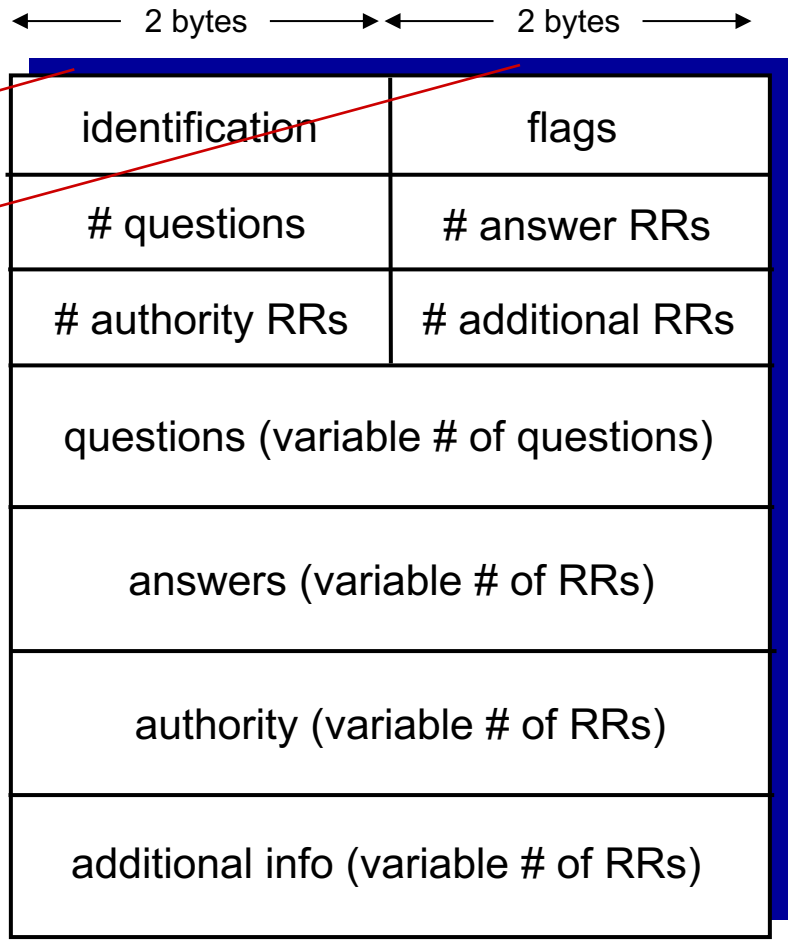
>nslookup -type=MX ucla.edu

DNS protocol: messages

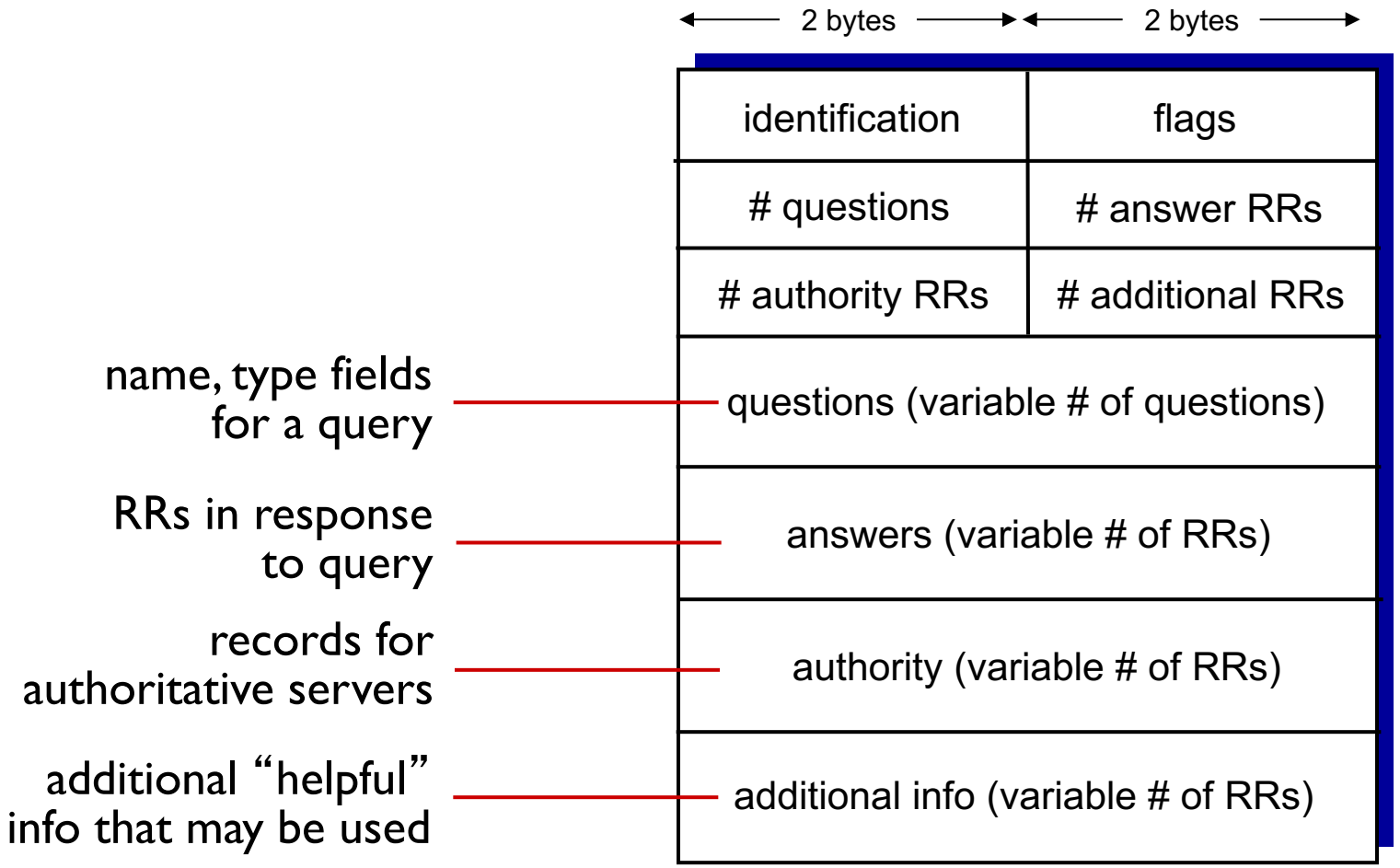
- *Mainly over UDP*
- *query* and *reply* messages, both with same *message format*

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



Questions

- Difference between DNS Message vs. HTTP/SMTP Messages?
  - HTTP, FTP, SMTP
    - ASCII
    - Field: flexible length (sp)
  - DNS: fixed-length (bytes) for each field
- Difference between DNS protocol message and DNS records
  - Protocol message: exchanged between host and local DNS resolver, between various DNS resolvers
  - DNS records: stored at DNS resolvers (entry in the database)

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/TCP in Java~~

Pure P2P architecture

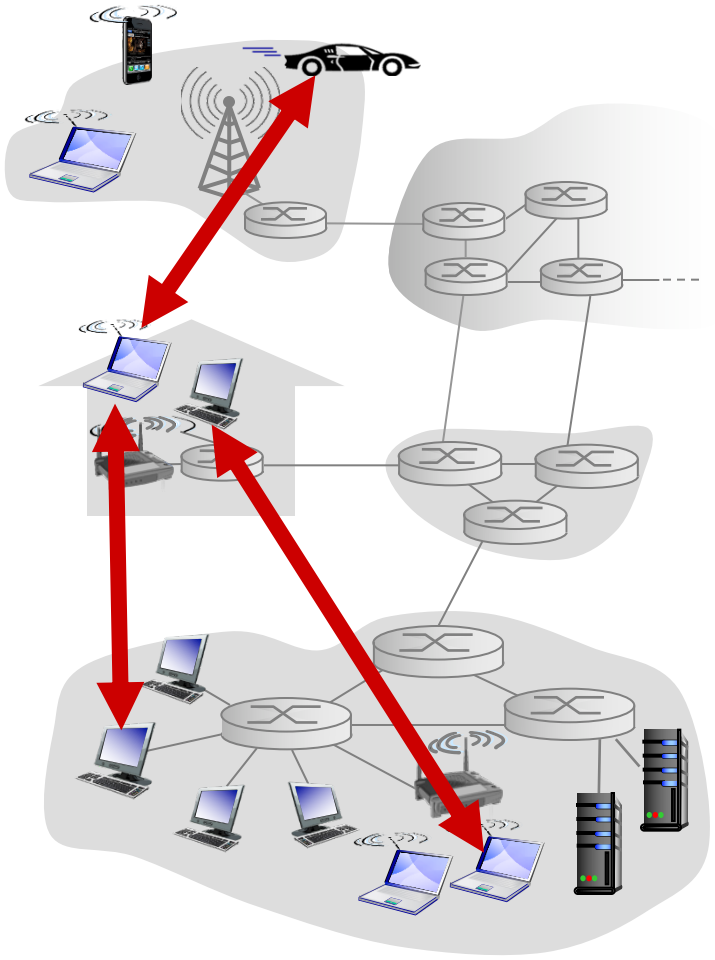
- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

Examples:

- BitTorrent, Skype (VoIP)

One Topic:

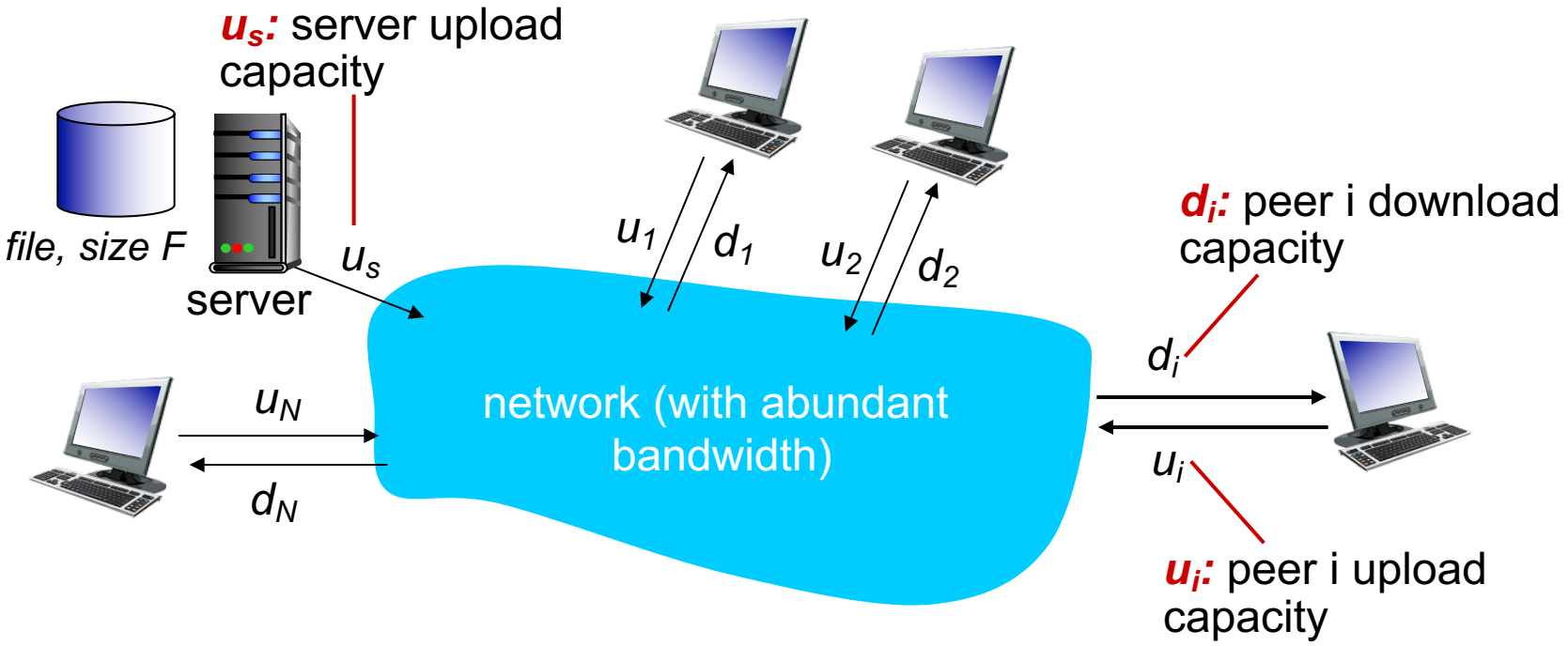
- File distribution



File distribution: client-server vs. P2P

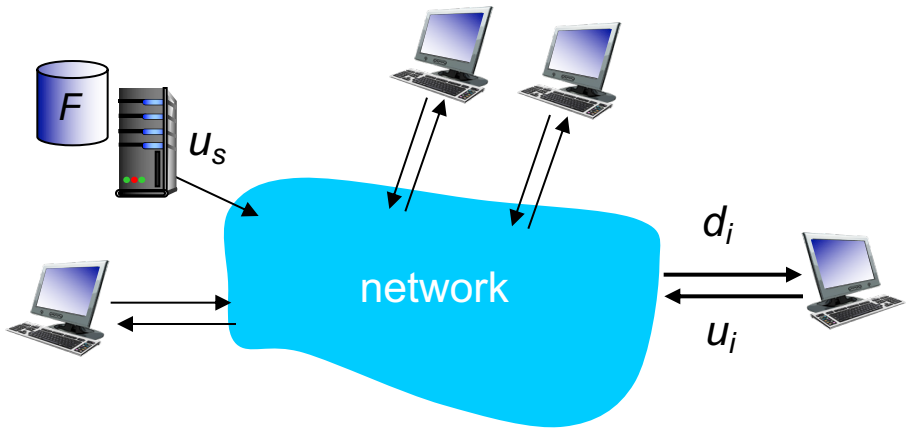
Question: how much time to distribute file (size *F*) from one server to *N* peers?

- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must send (upload) *one* file copy per client:
  - time to send one copy: F/u_s
  - time to send for N clients: $F/(u_s/N) = NF/u_s$



- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}

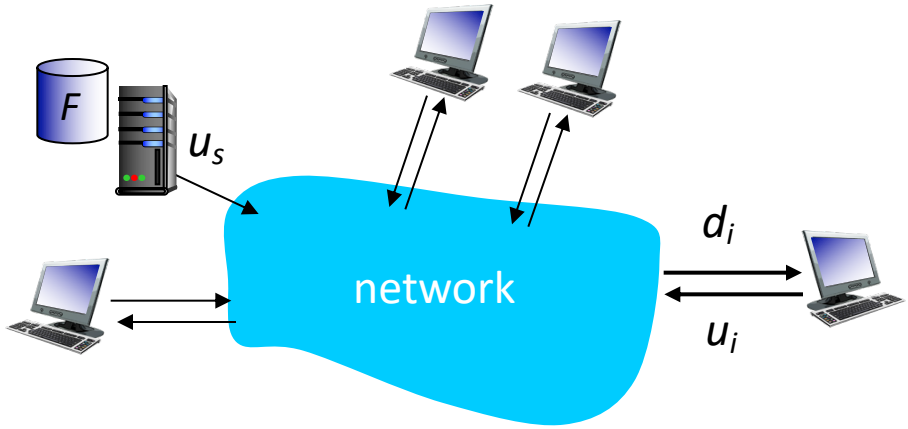
time to distribute F to N clients using client-server approach

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

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 (limting max download rate) is $u_s + Su_i$



time to distribute F to N clients using P2P approach

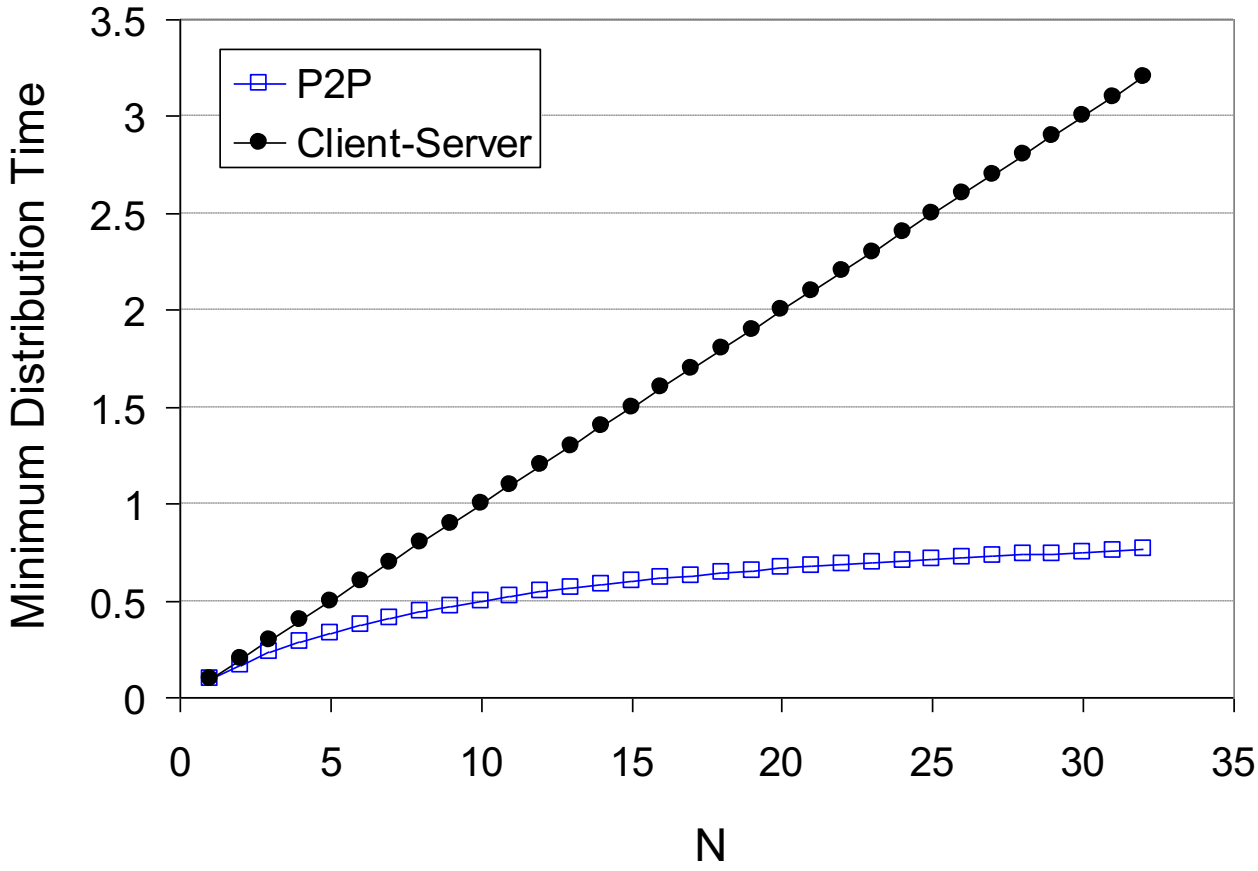
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + Su_i)\}$$

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



BitTorrent (optional)

- Read textbook

Main ideas:

- Divide the file into multiple chunks
  - Each chunk can be downloaded and uploaded independently
- Each peer download and upload chunks
- Peers find other peers and learn the available files for transfer from the trackers
  - Trackers: special severs that assist p2p communications among peers
  - Trackerless design is also available via distributed hash table (DHT)
- Tit for tat as incentive mechanism

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/TCP in Java~~

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*

frame *i+1*

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

Multimedia: video

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

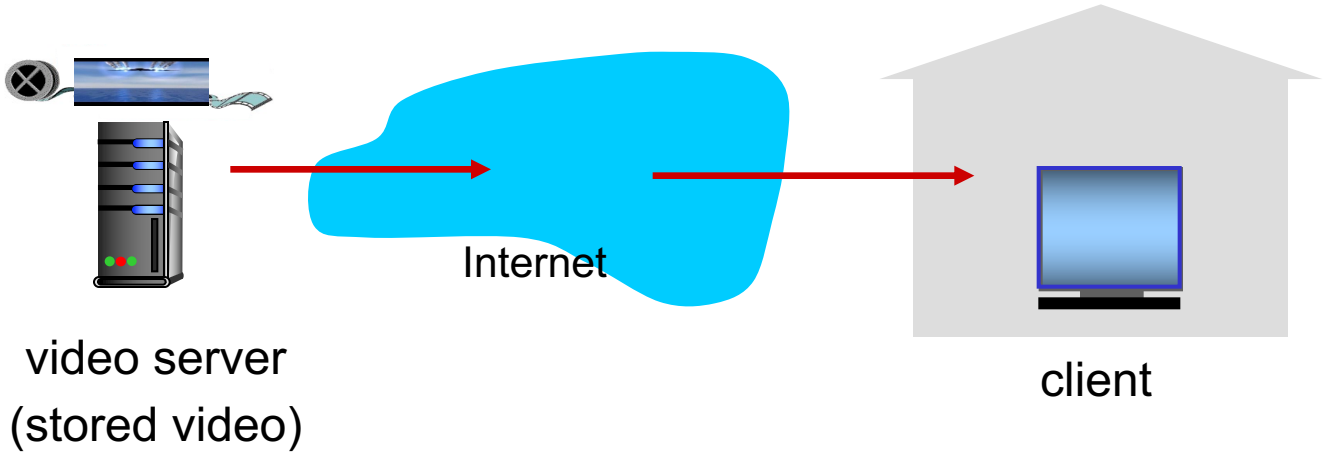
frame *i*

frame *i+1*

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

Streaming stored video:

simple scenario:



Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *server:*
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - *manifest file:* provides URLs for different chunks
- *client:*
  - periodically measures server-to-client bandwidth
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or has high available bandwidth)

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*

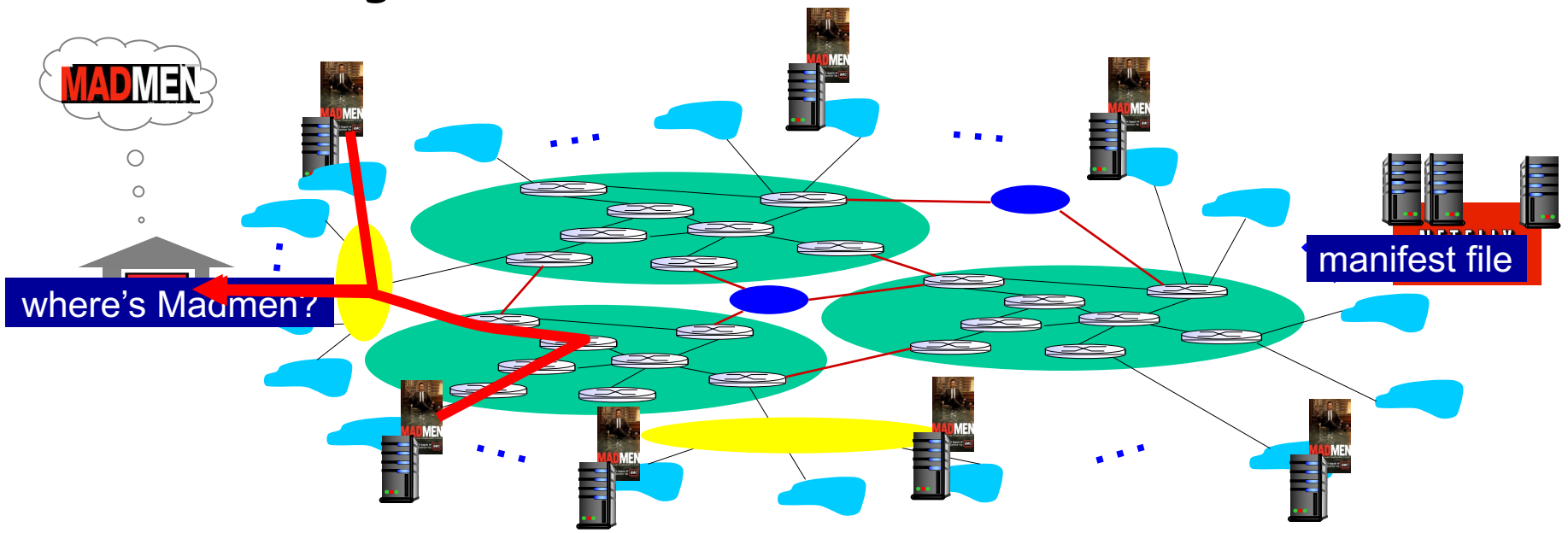
Content distribution networks

- *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 (points of presence) 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

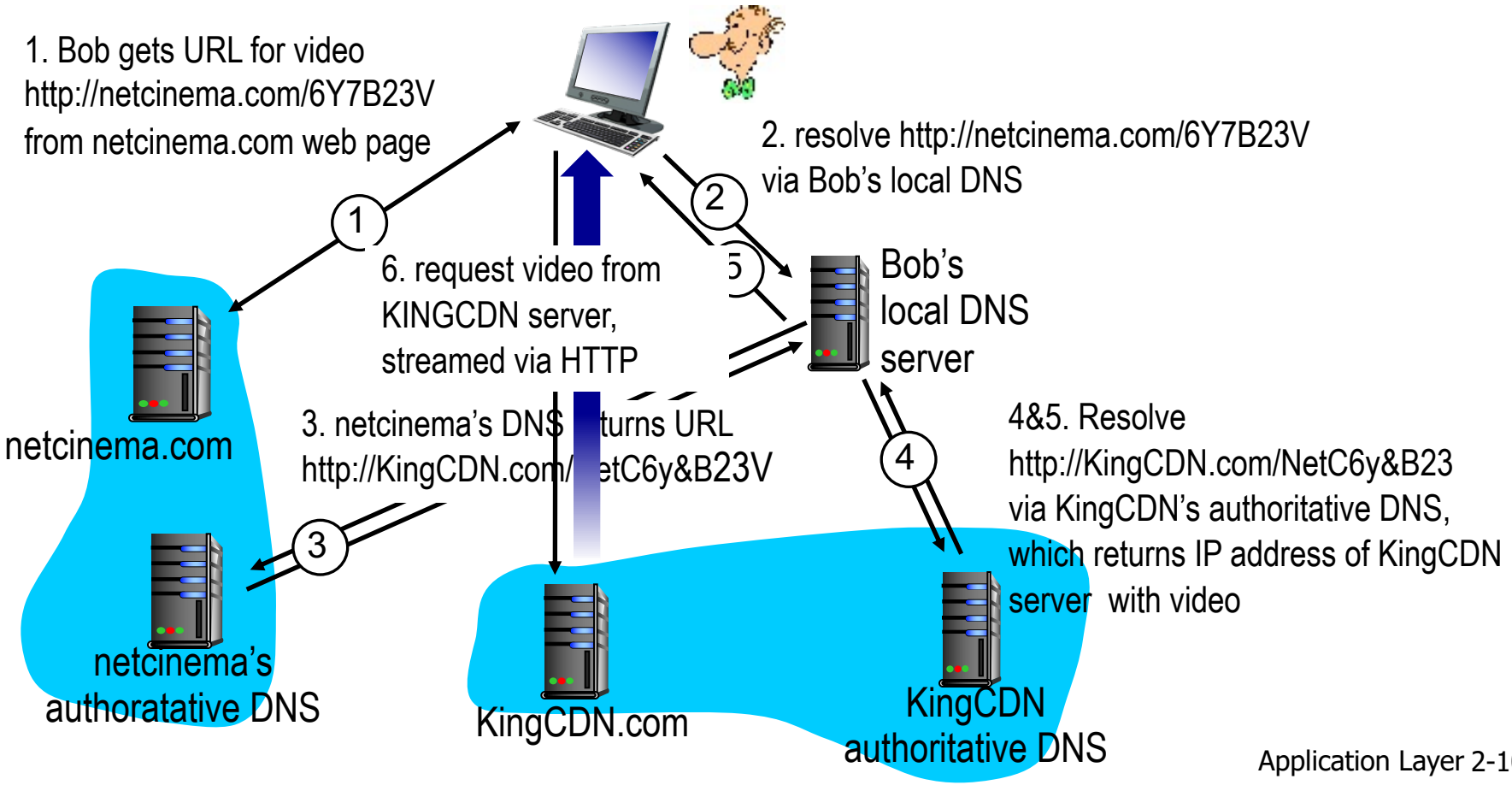
- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

more .. in chapter 9

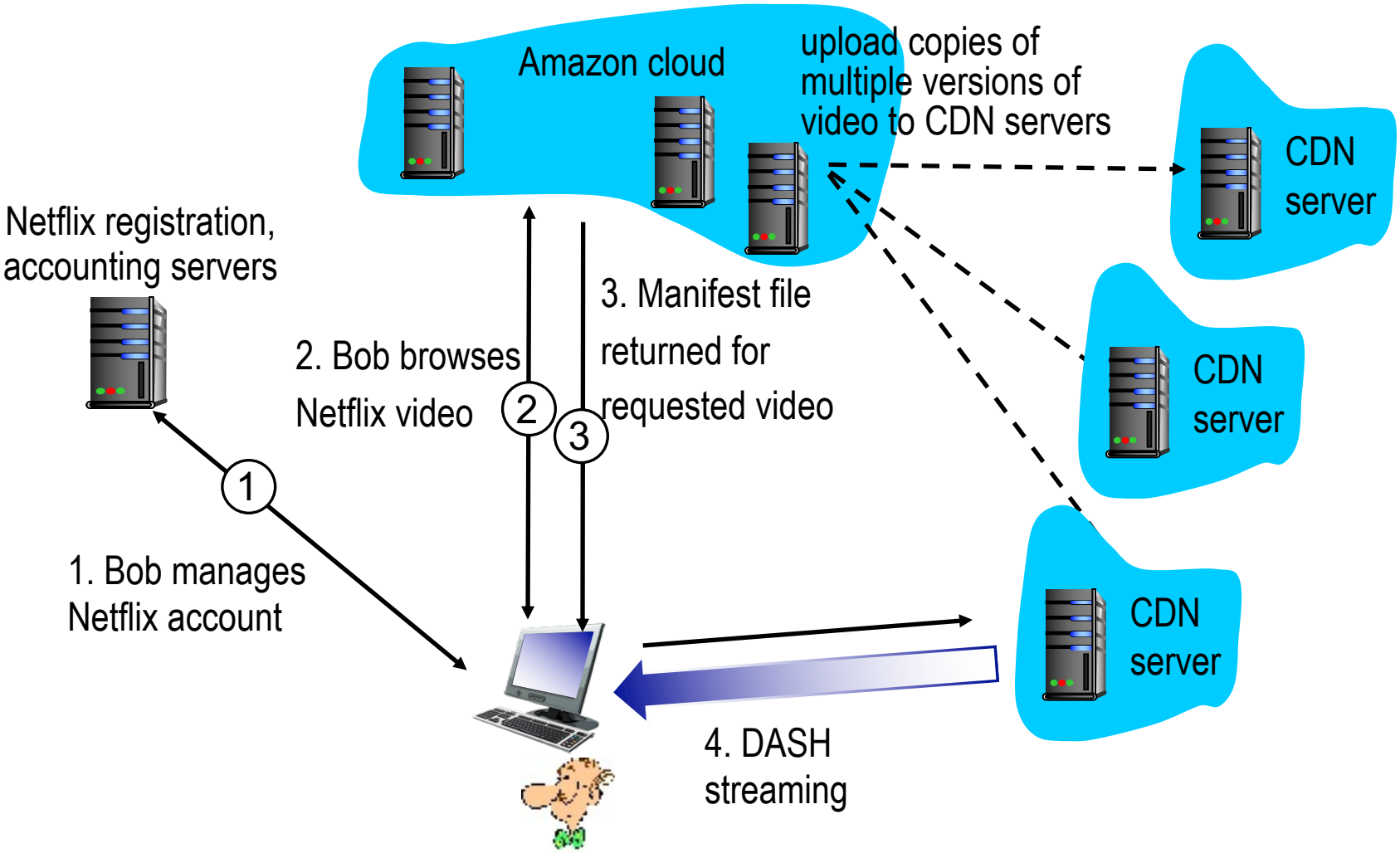
CDN content access: a closer look

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

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



Case study: Netflix



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
- socket programming:

  BSD TCP/UDP sockets

  (in TA's sessions)