

CPSC 441

Computer Networks

Majid Ghaderi
Department of Computer Science
University of Calgary

Chapter 2: application layer

our goal:

- ❖ learn about protocols by examining popular application-level protocols
 - HTTP
 - SMTP / POP3 / IMAP
 - DNS

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

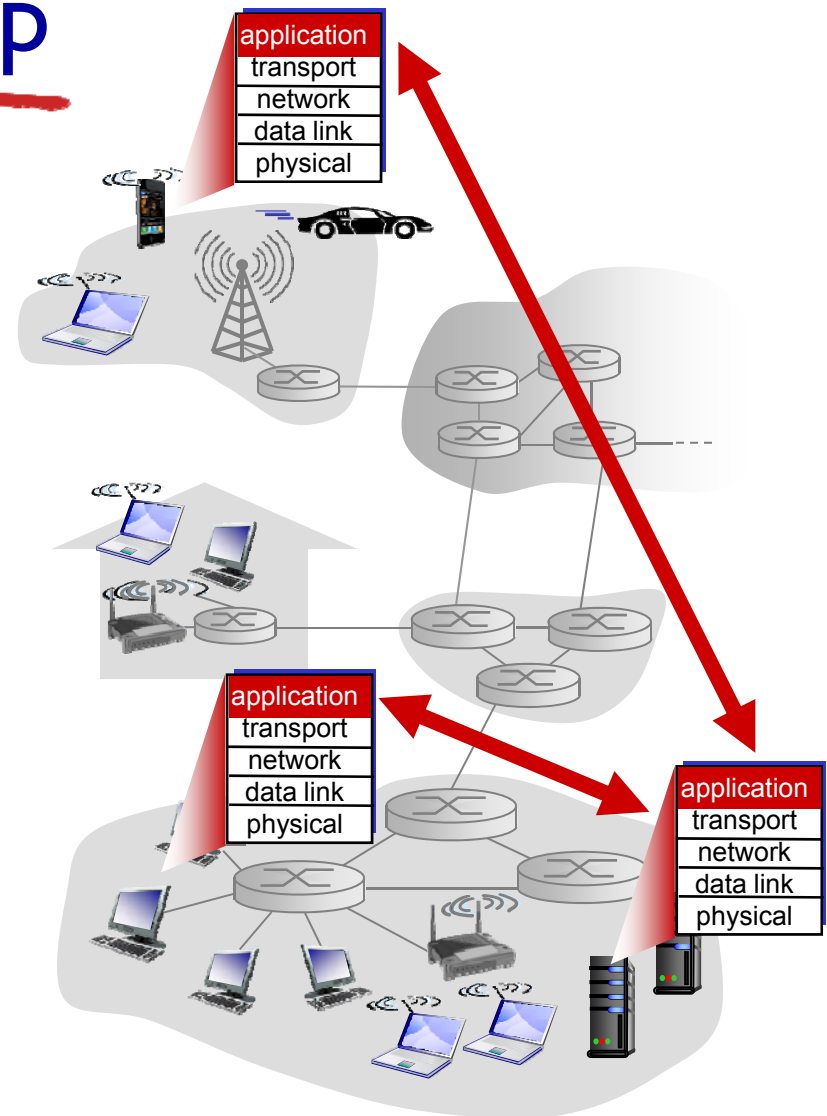
Creating a network app

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

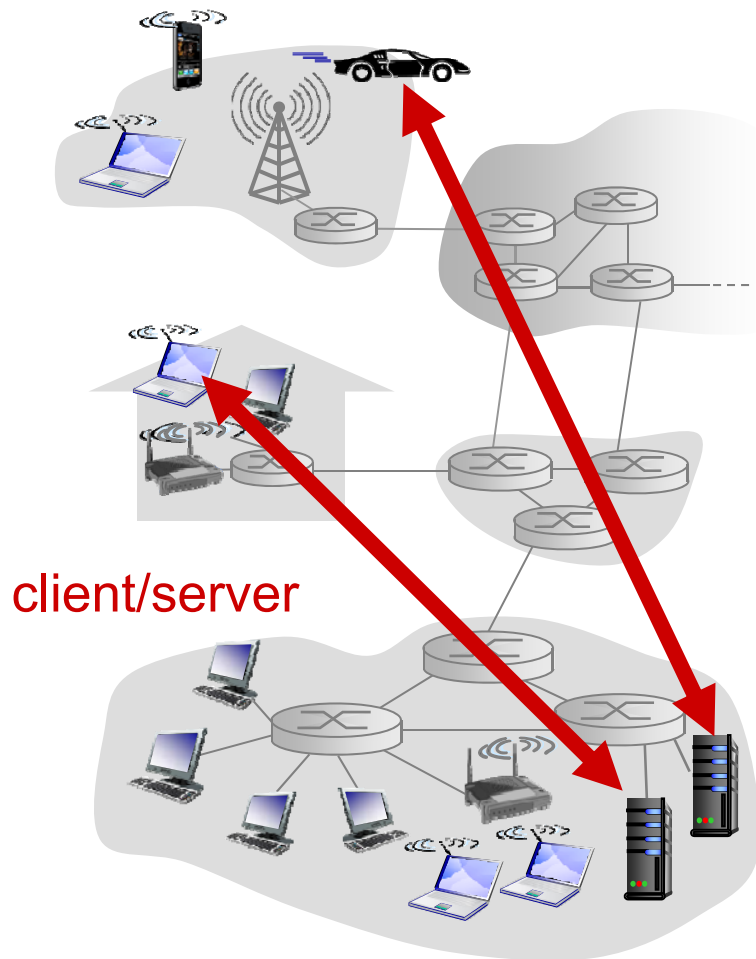


Application architectures

possible structure of applications:

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

Client-server architecture



server:

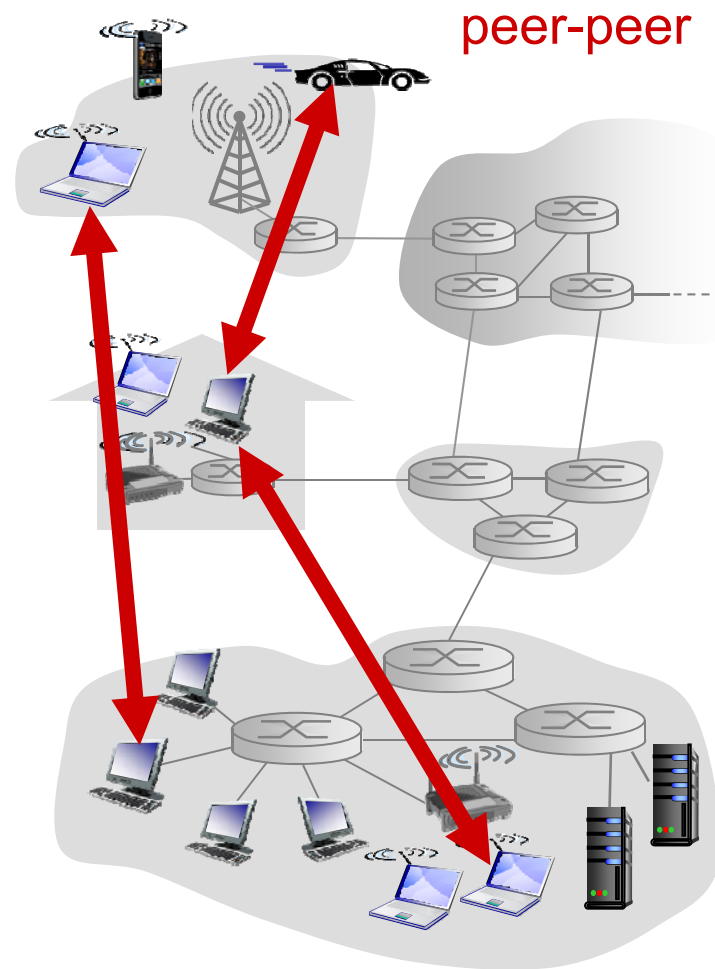
- ❖ always-on host
- ❖ permanent IP address
- ❖ data centers for scaling

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

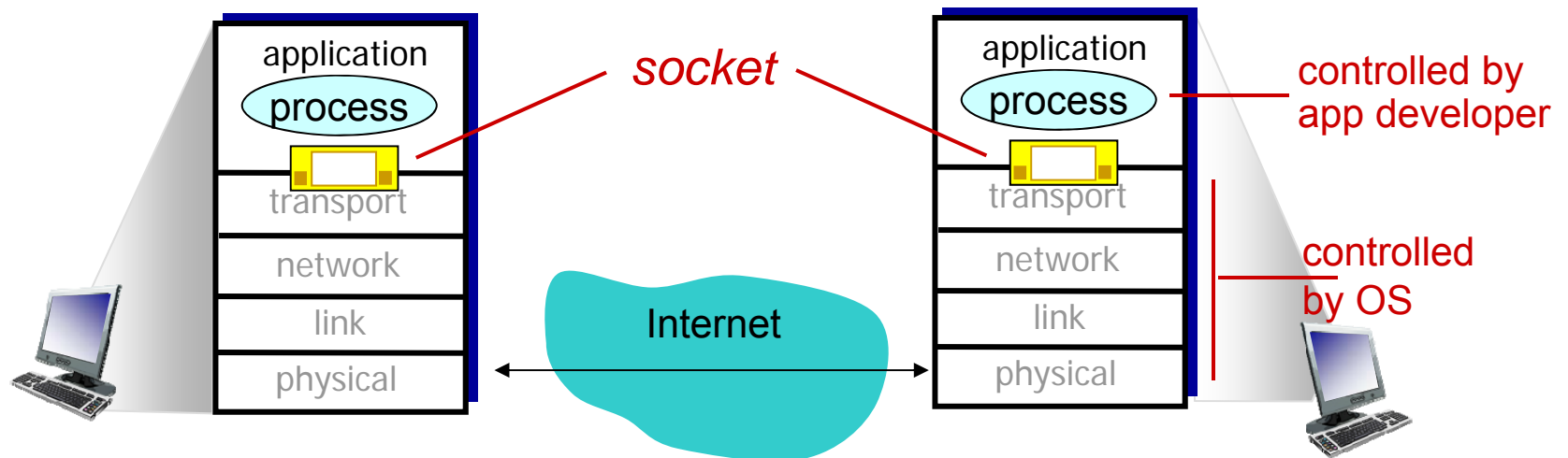
client process: process that initiates communication

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,

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

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

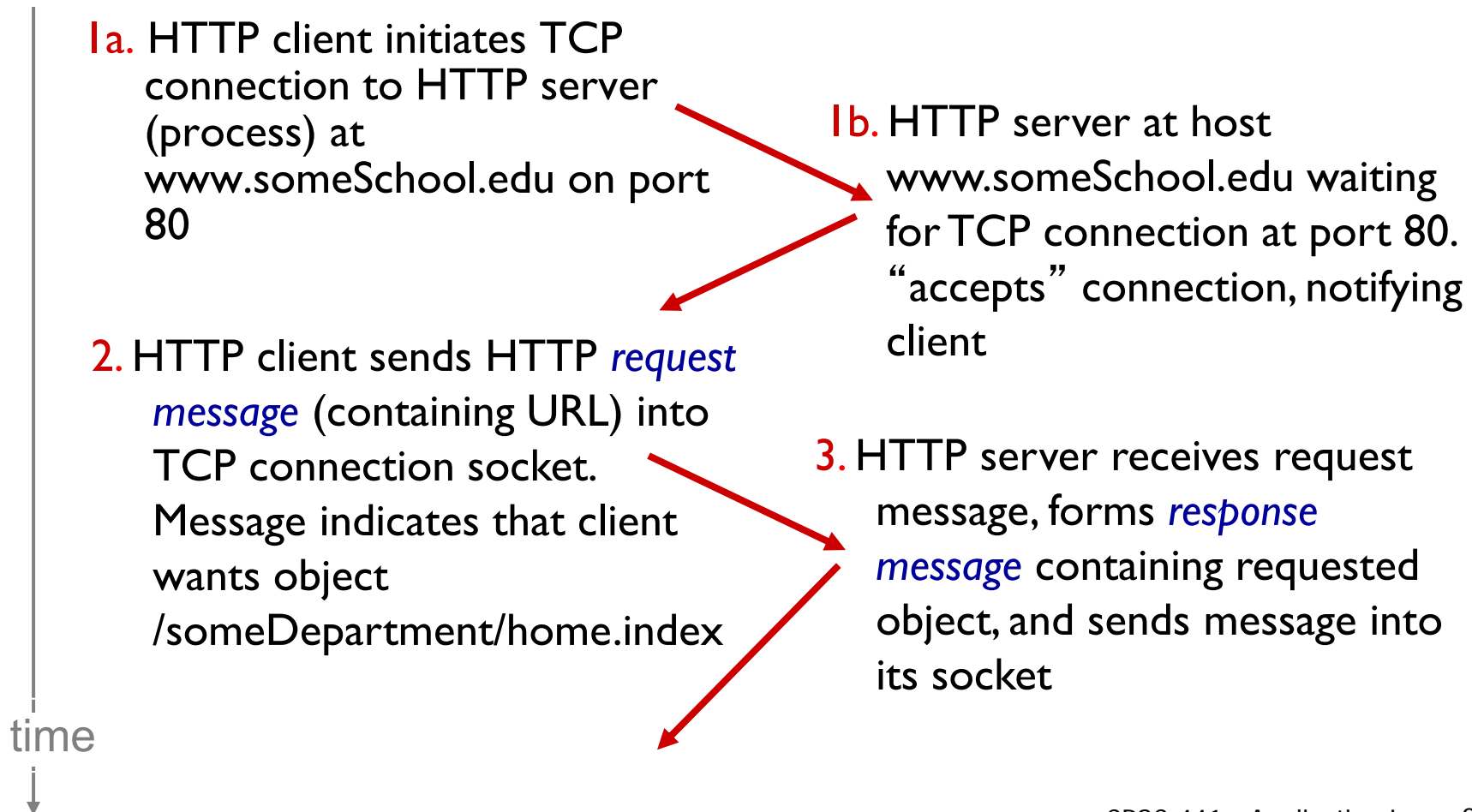
- ❖ multiple objects can be sent over single TCP connection between client, server

Non-persistent HTTP

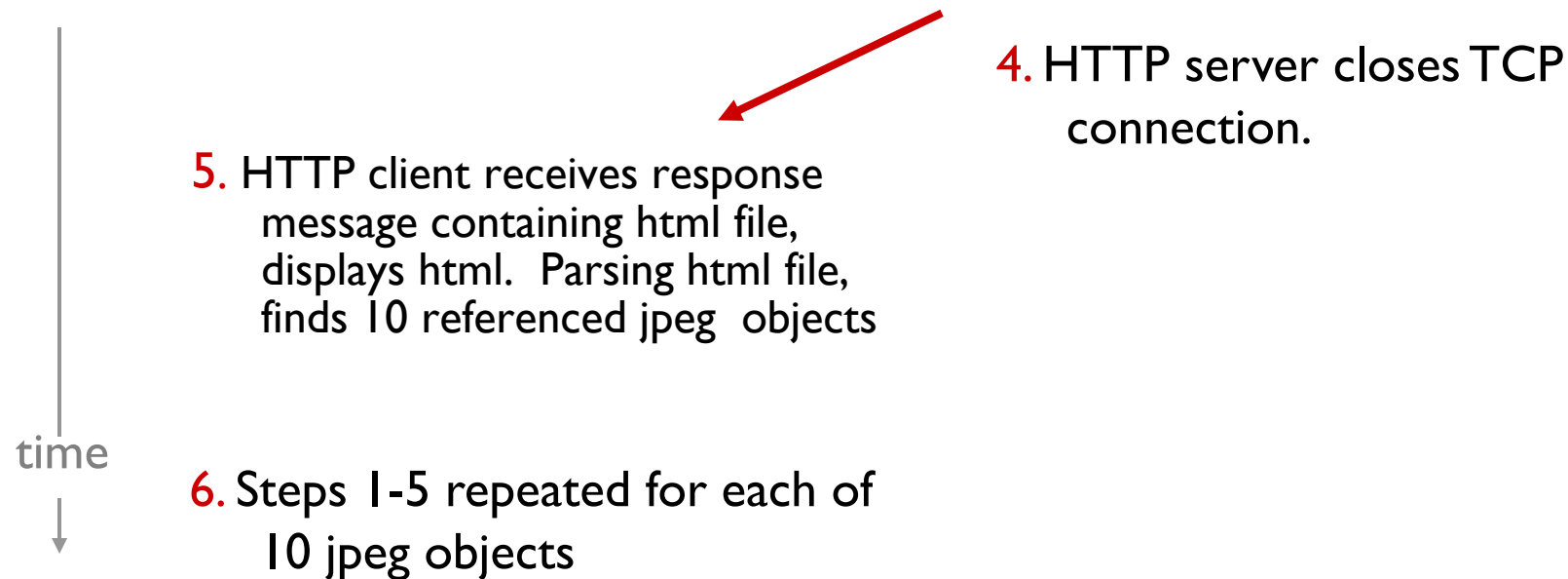
suppose user enters URL:

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

(contains text,
references to 10
jpeg images)



Non-persistent HTTP (cont.)

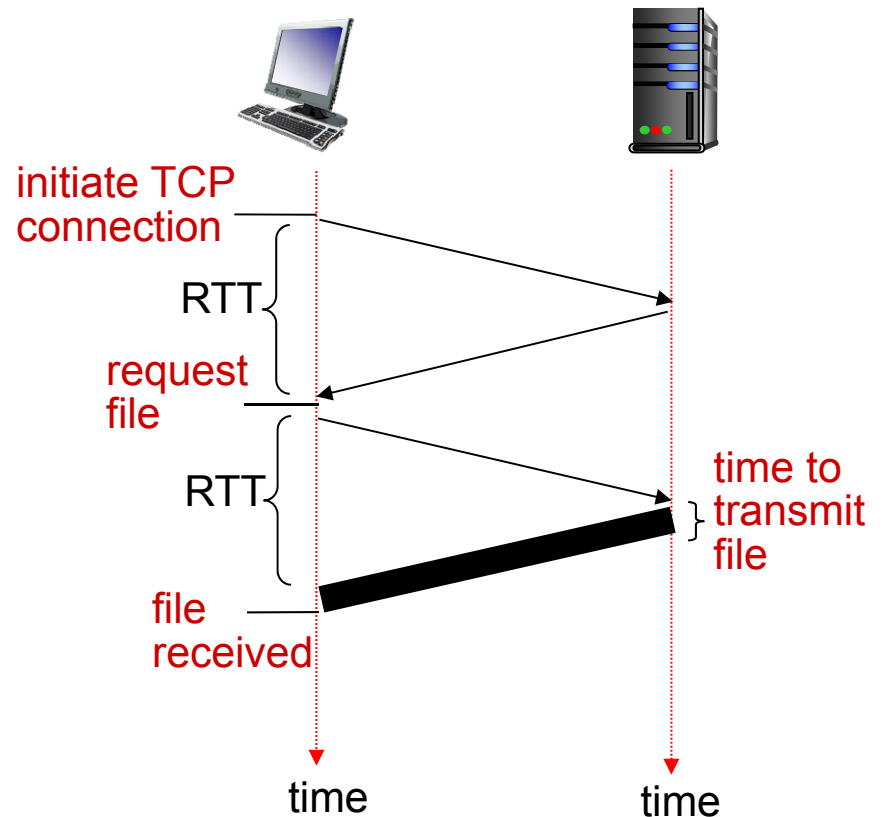


Non-persistent HTTP: response time

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)

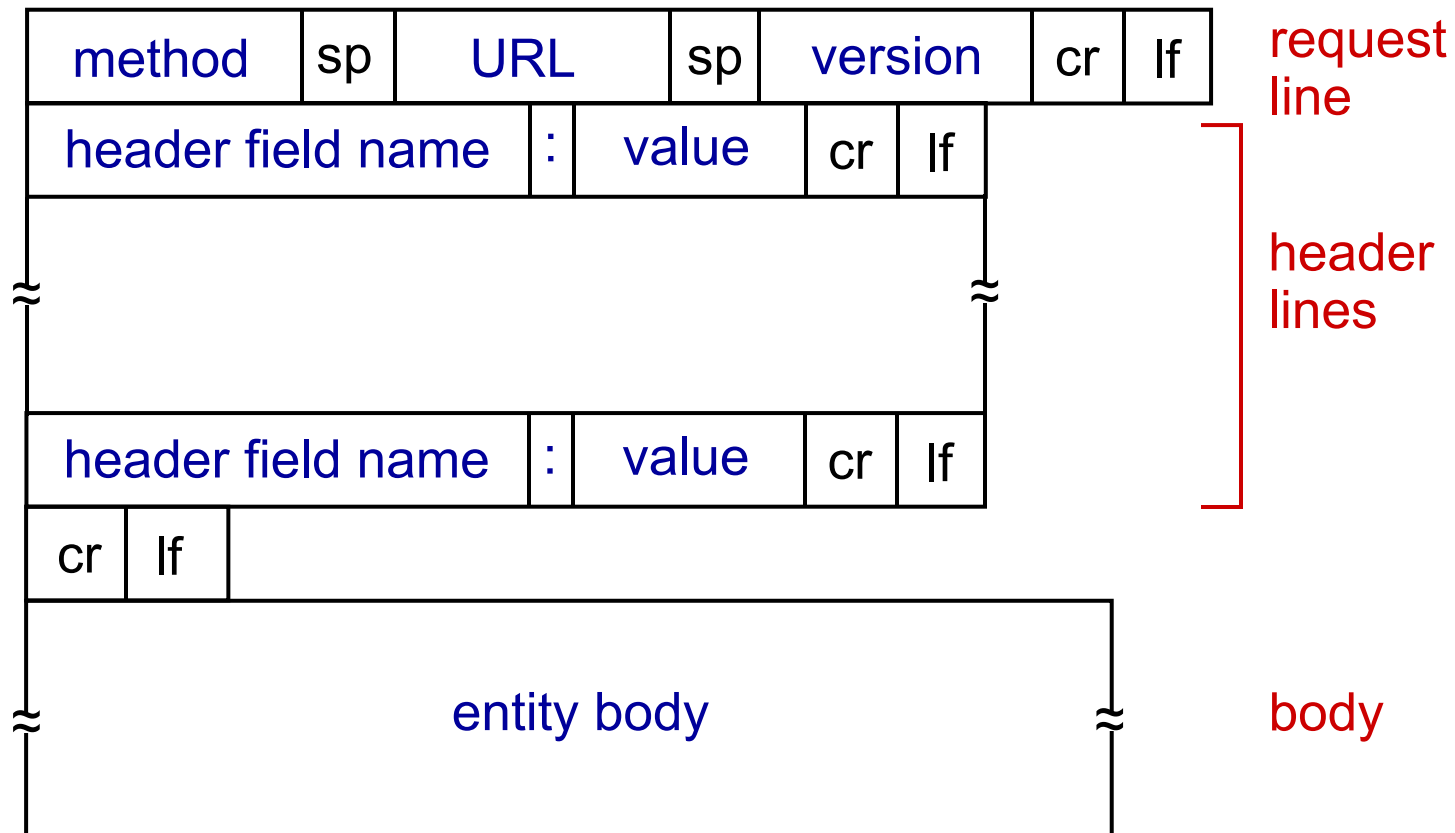
header
lines

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



HTTP response message

status line
(protocol
status code
status phrase)

header
lines

data, e.g.,
requested
HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-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

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

User-server state: cookies

many Web sites use cookies

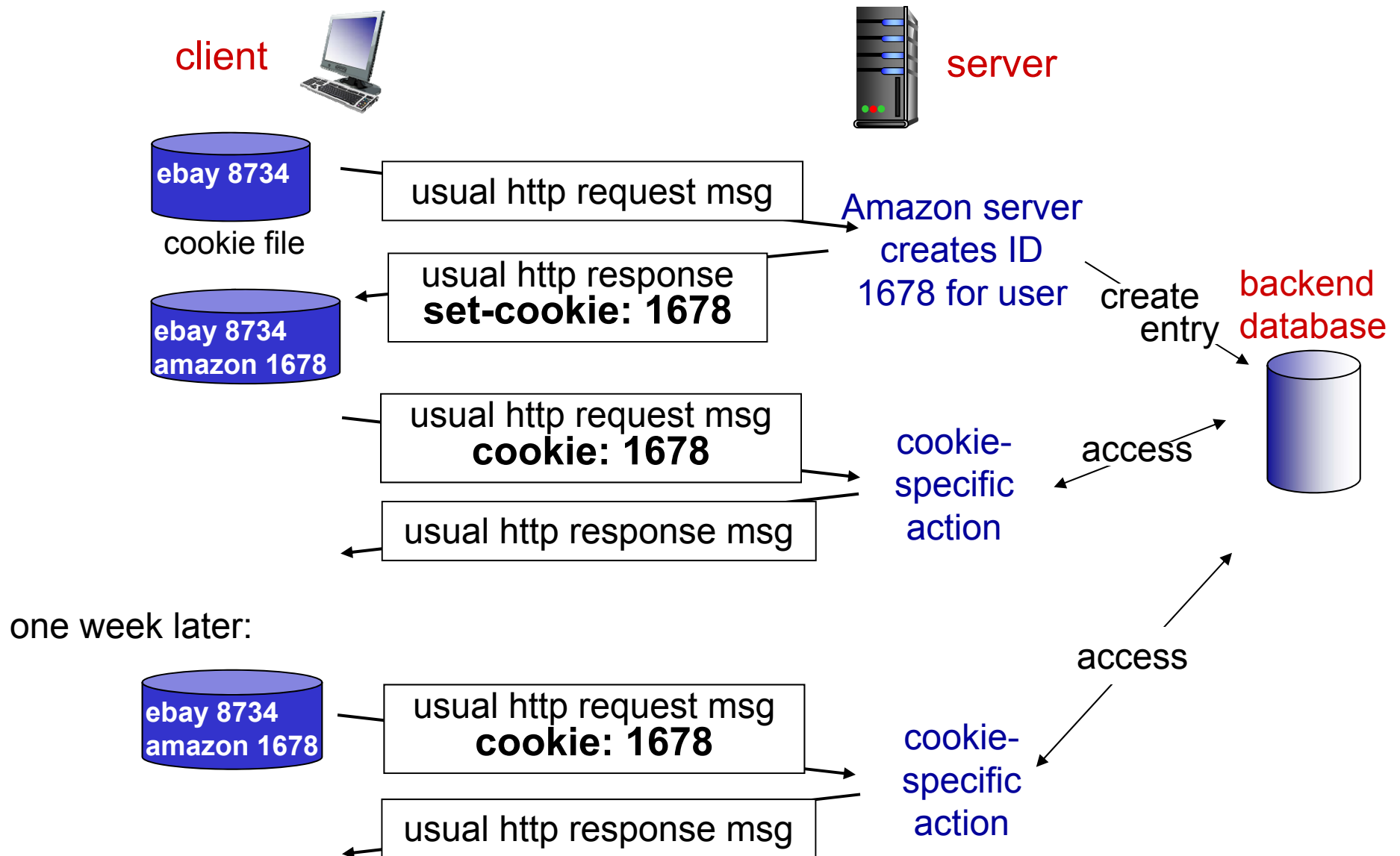
four components:

- 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)

cookies and privacy: aside

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

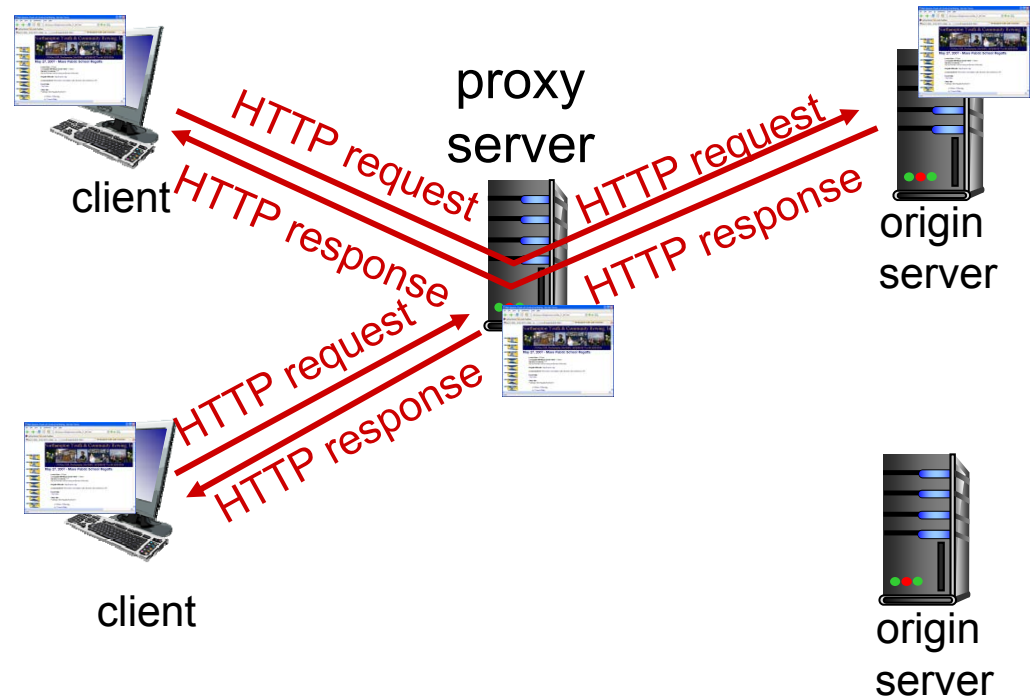
how to keep “state”:

- ❖ 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

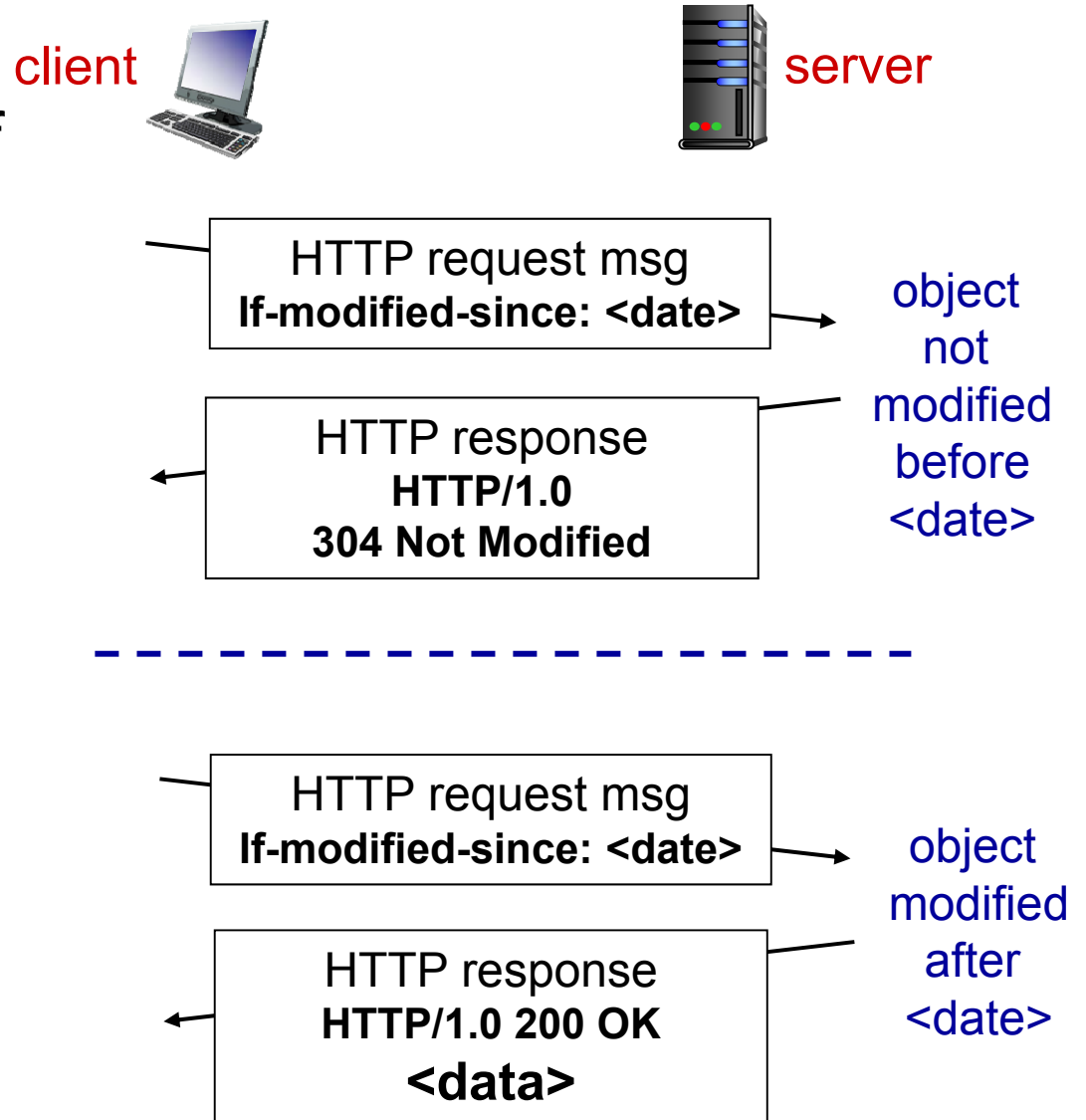
- ❖ cache acts as both client and server
 - server for original requesting client
 - client to origin server
- ❖ typically cache is installed by ISP (university, company, residential ISP)

why Web caching?

- ❖ reduce response time for client request
- ❖ reduce traffic on an institution's access link

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

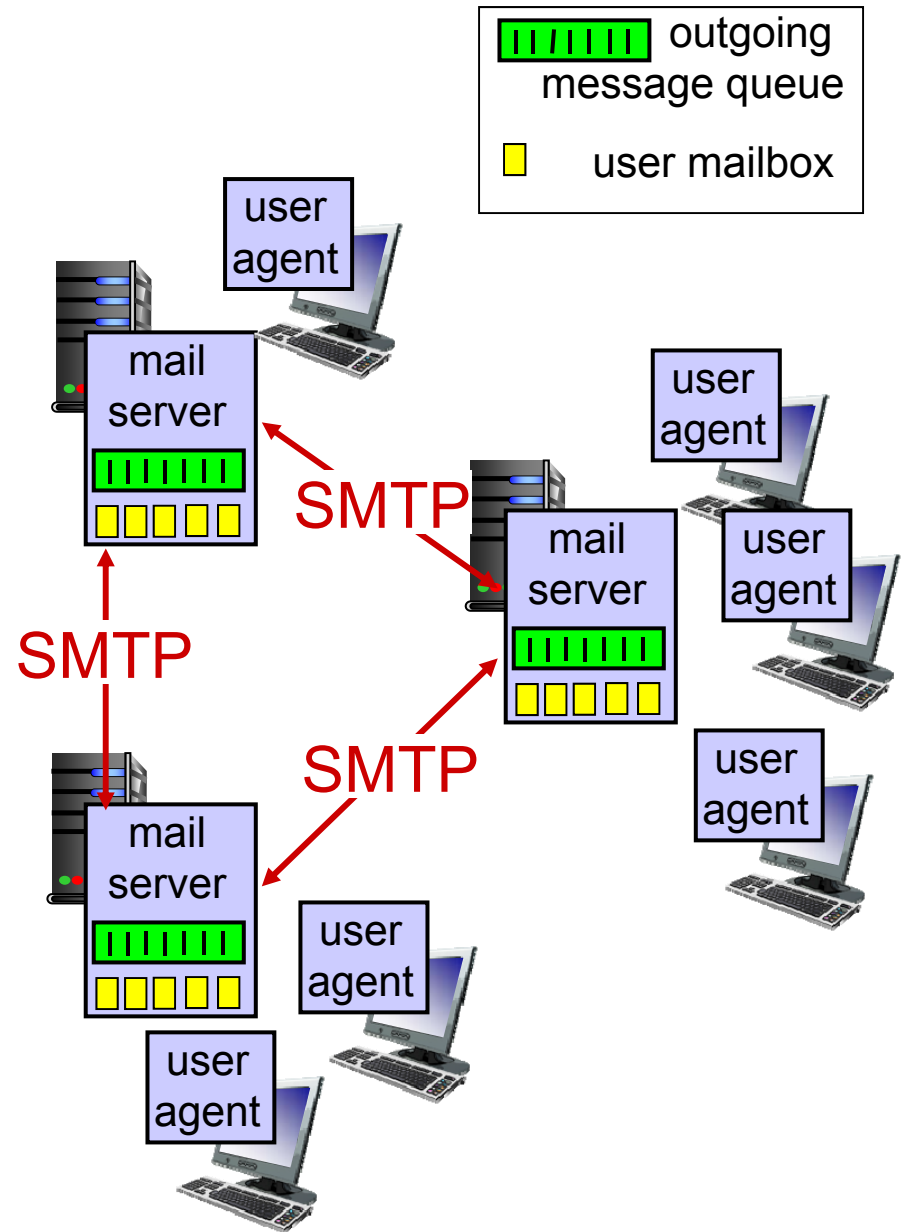
Electronic mail

Three major components:

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

User Agent

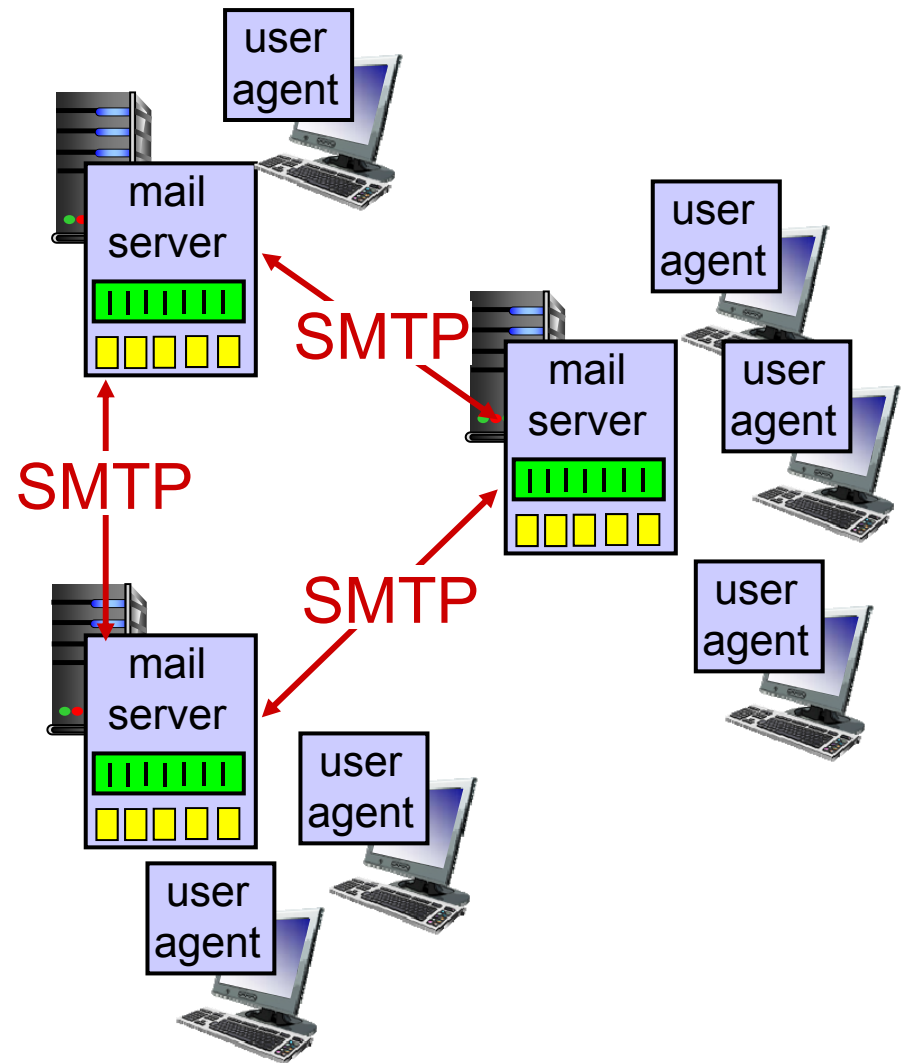
- ❖ a.k.a. “mail reader”
- ❖ composing, editing, reading mail messages
- ❖ e.g., Outlook, Thunderbird
- ❖ outgoing, incoming messages stored on server



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

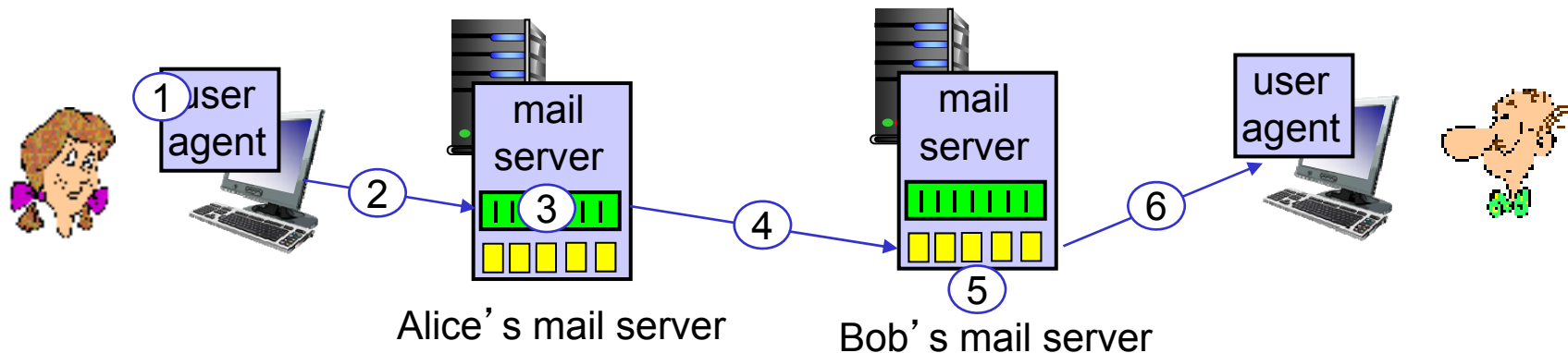


Electronic Mail: SMTP [RFC 2821]

- ❖ uses TCP to reliably transfer email message from client to server, port 25
- ❖ command/response interaction
 - **commands:** ASCII text
 - **response:** status code and phrase

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



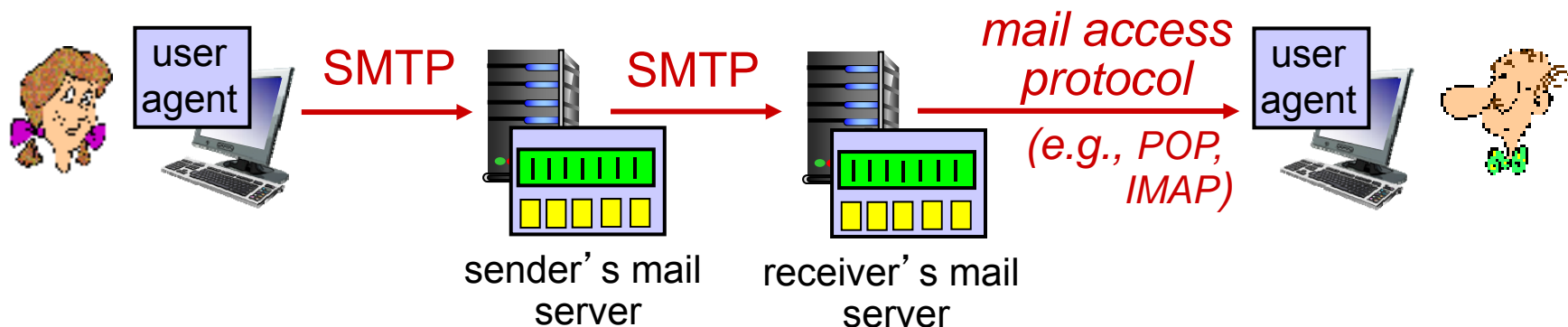
SMTP features

- ❖ 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

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.

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

DNS: domain name system

Internet hosts:

- 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

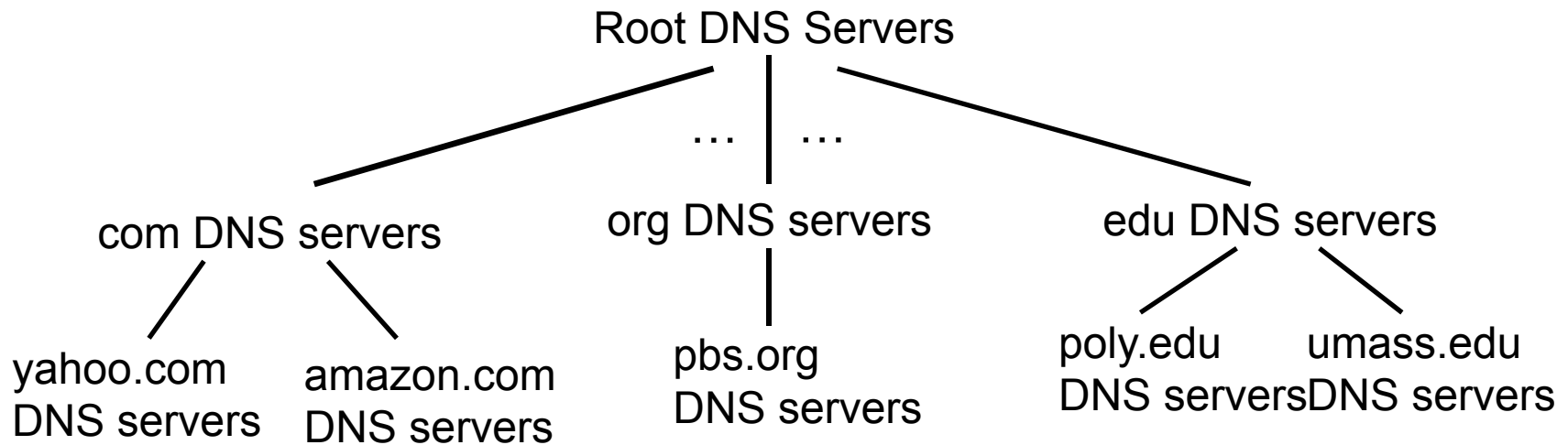
- ❖ hostname to IP address translation
- ❖ host aliasing
 - canonical, alias names
- ❖ load distribution
 - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

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

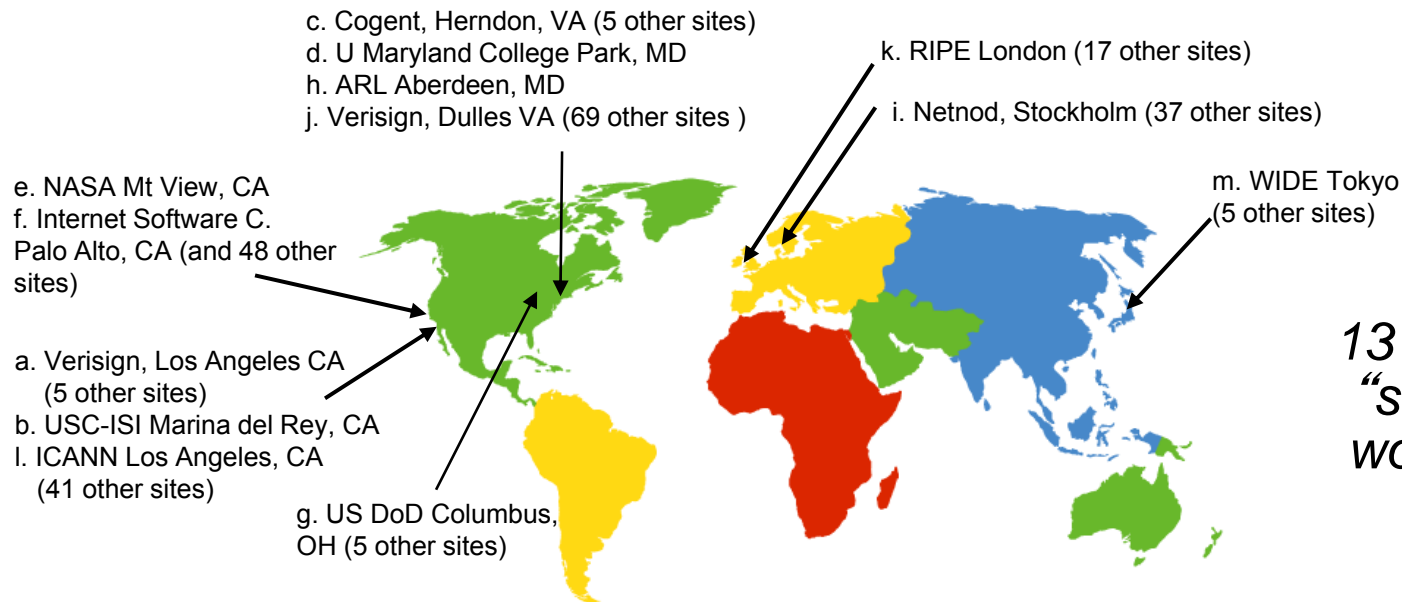
A: doesn't scale!

DNS: a distributed, hierarchical database



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

authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts

Local DNS name server

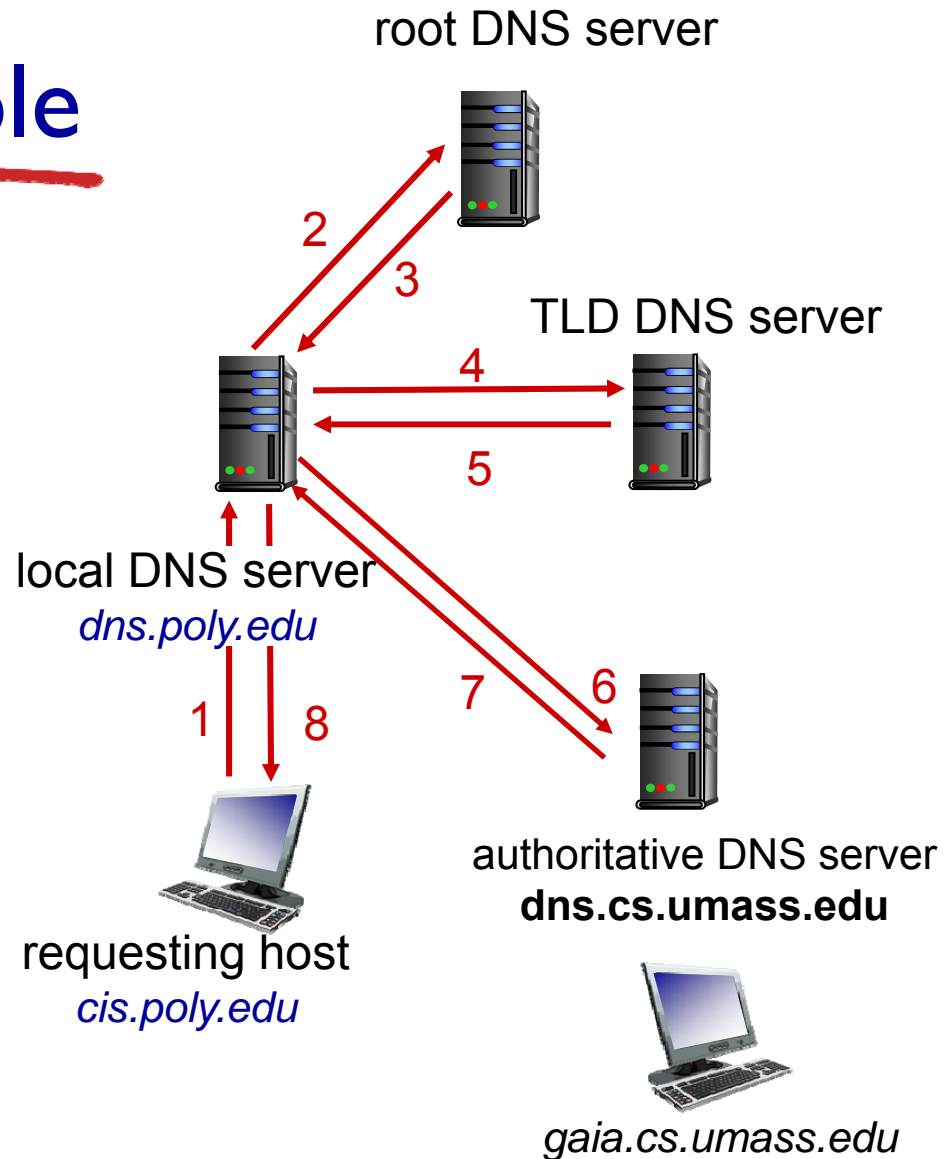
- ❖ each ISP (residential ISP, company, university) has one
 - also called “default name server”
- ❖ when host makes DNS query, query is sent to its local DNS server
 - has local cache of recent name-to-address translation pairs (but may be out of date!)
 - acts as proxy, forwards query into hierarchy

DNS name resolution example

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

iterated query:

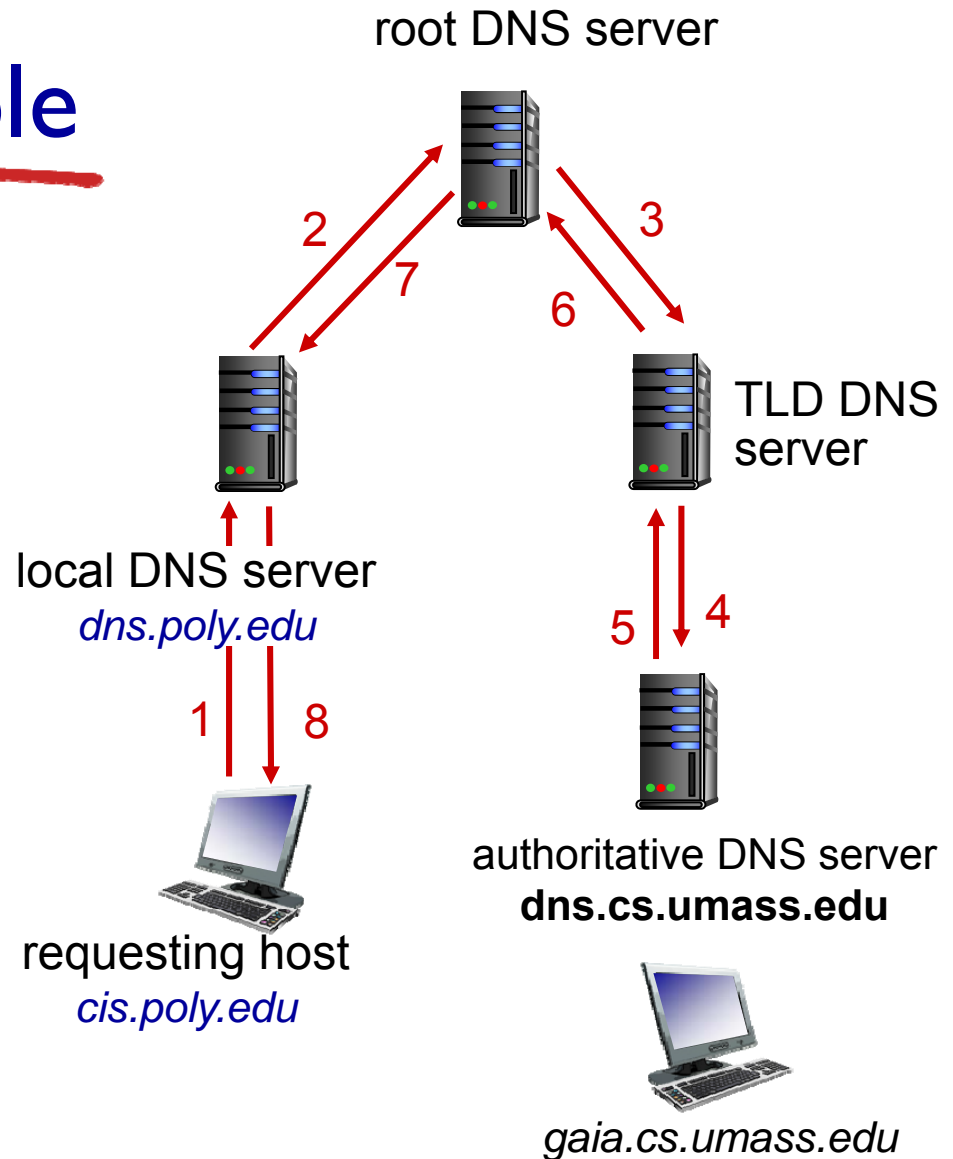
- ❖ contacted server replies with name of server to contact
- ❖ “I don’t know this name, but ask this server”



DNS name resolution example

recursive query:

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



DNS 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**

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
 - 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 bypass
 - ❖ bombard TLD servers
 - potentially more dangerous
- ❖ Oct 21, 2016
 - IoT DDoS attack on Dyn, a major provider of DNS services

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

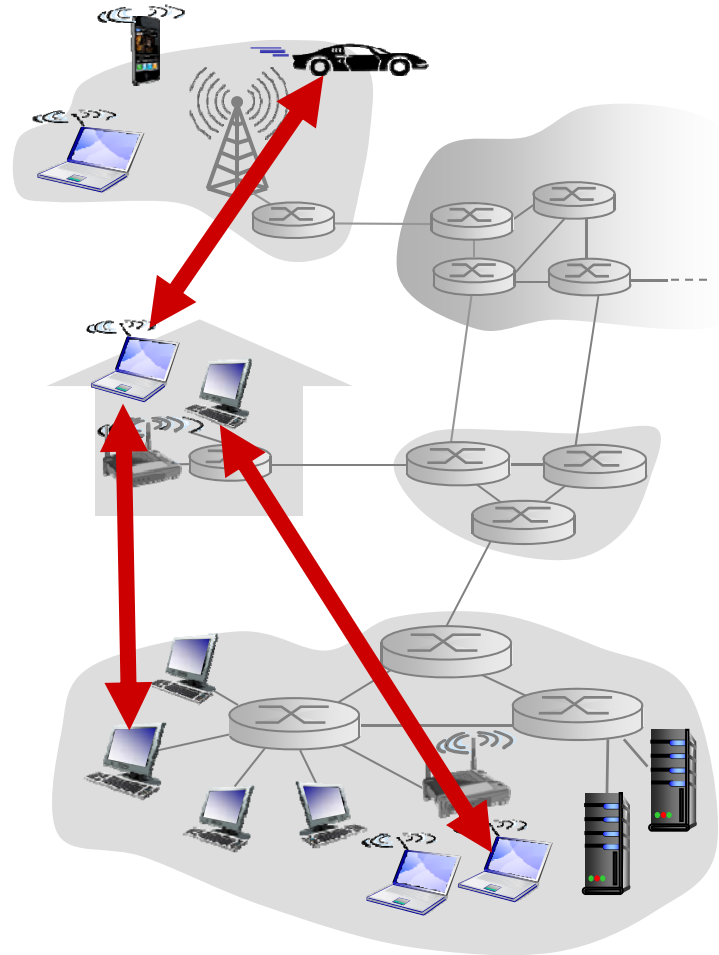
2.6 Video streaming and content distribution networks

P2P architecture

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

examples:

- file distribution (BitTorrent)
- VoIP (Skype)

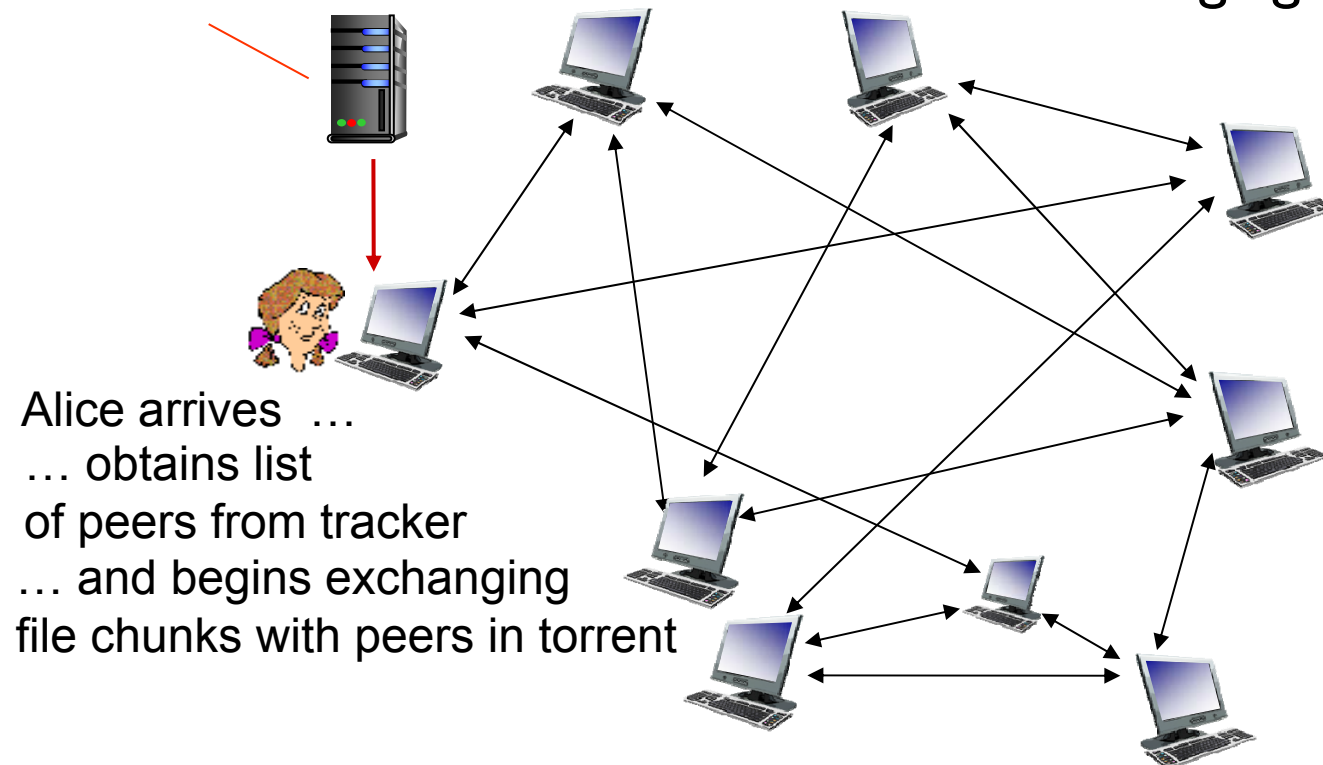


P2P file distribution: BitTorrent

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

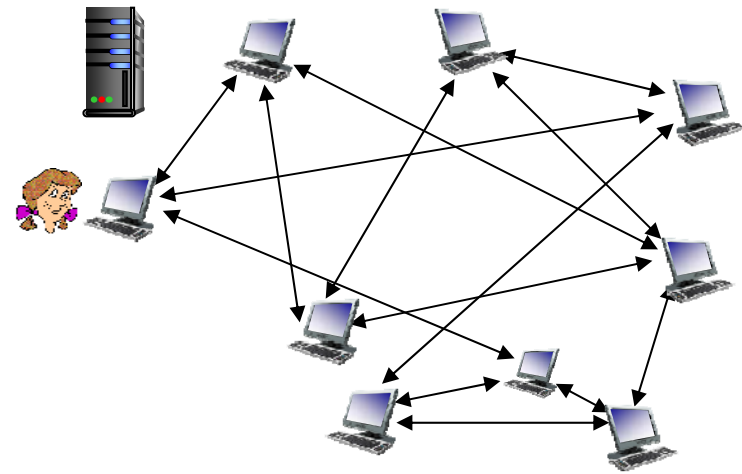
tracker: tracks peers participating in torrent

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



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

Video Streaming and CDNs: context

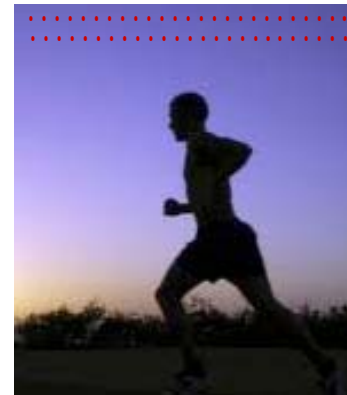
- video traffic: major consumer of Internet bandwidth
 - Netflix, YouTube: 37%, 16% of downstream residential ISP traffic
 - ~1B YouTube users, ~75M Netflix users
- challenge: scale - how to reach ~1B users?
 - single mega-video server won't work (why?)
- challenge: heterogeneity
 - different users have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

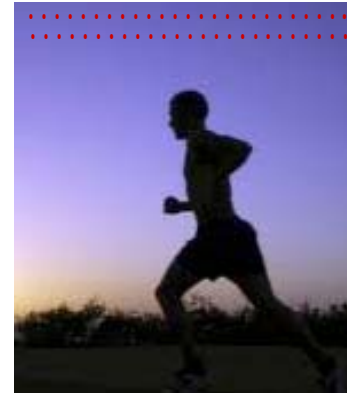


frame $i+1$

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

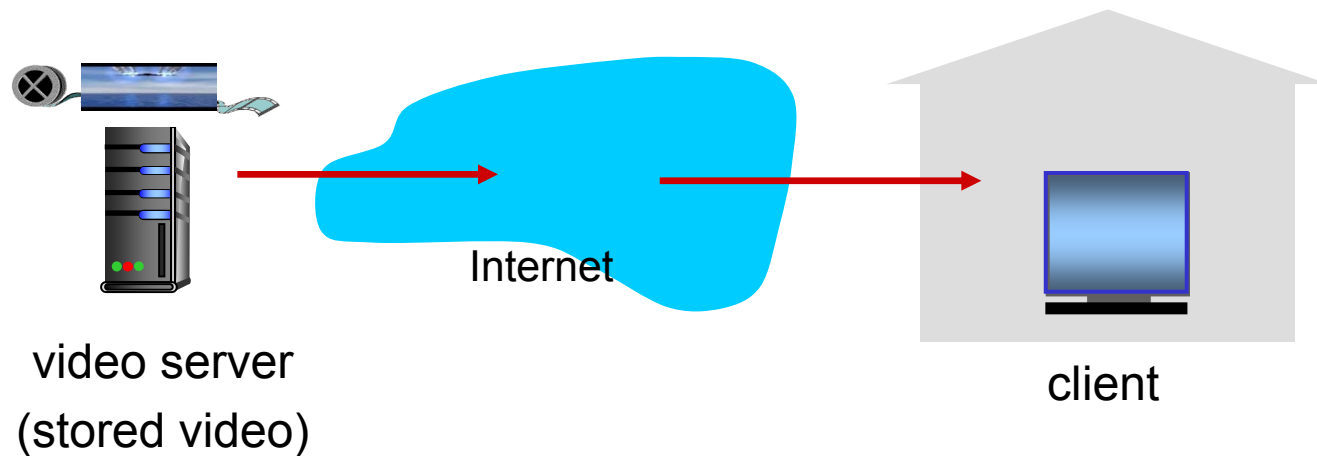
temporal coding example: instead of sending complete frame at $i+1$, send only differences from frame i



frame $i+1$

Streaming stored video:

simple scenario:



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*:

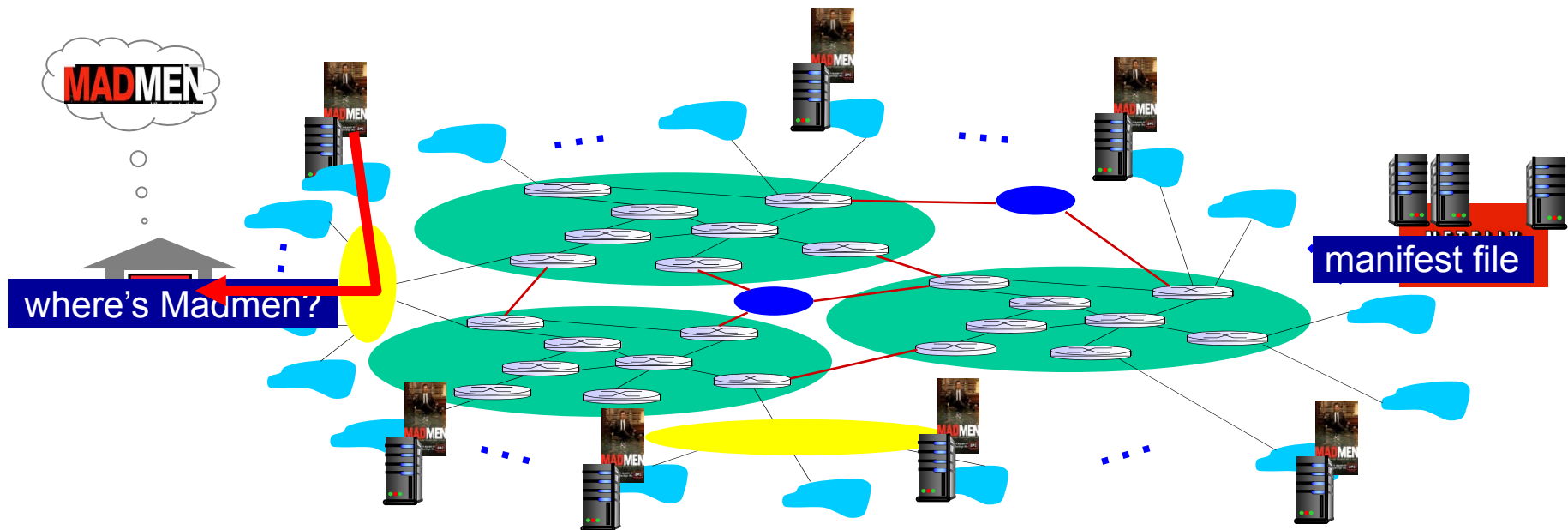
- periodically measures server-to-client bandwidth
- consulting manifest, requests one chunk at a time
 - chooses maximum coding rate sustainable given current bandwidth
 - can choose different coding rates at different points in time (depending on available bandwidth at time)

Content distribution networks

- ❖ *challenge*: how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?
- ❖ *solution*: store multiple copies of videos at multiple geographically distributed sites (*CDN*)
 - push CDN servers close to users

Content Distribution Networks (CDNs)

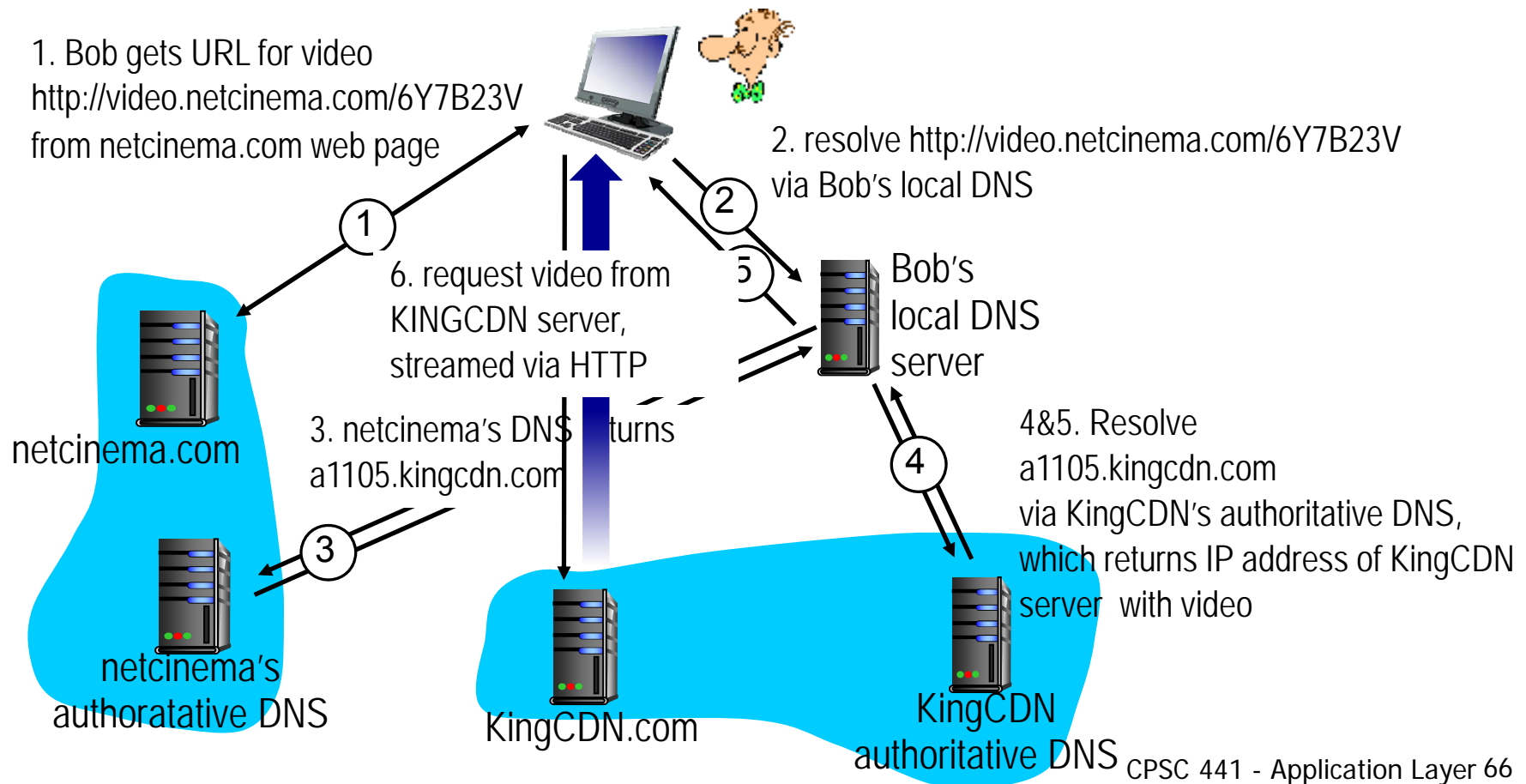
- CDN: stores copies of content at CDN nodes
 - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
 - directed to nearby copy, retrieves content



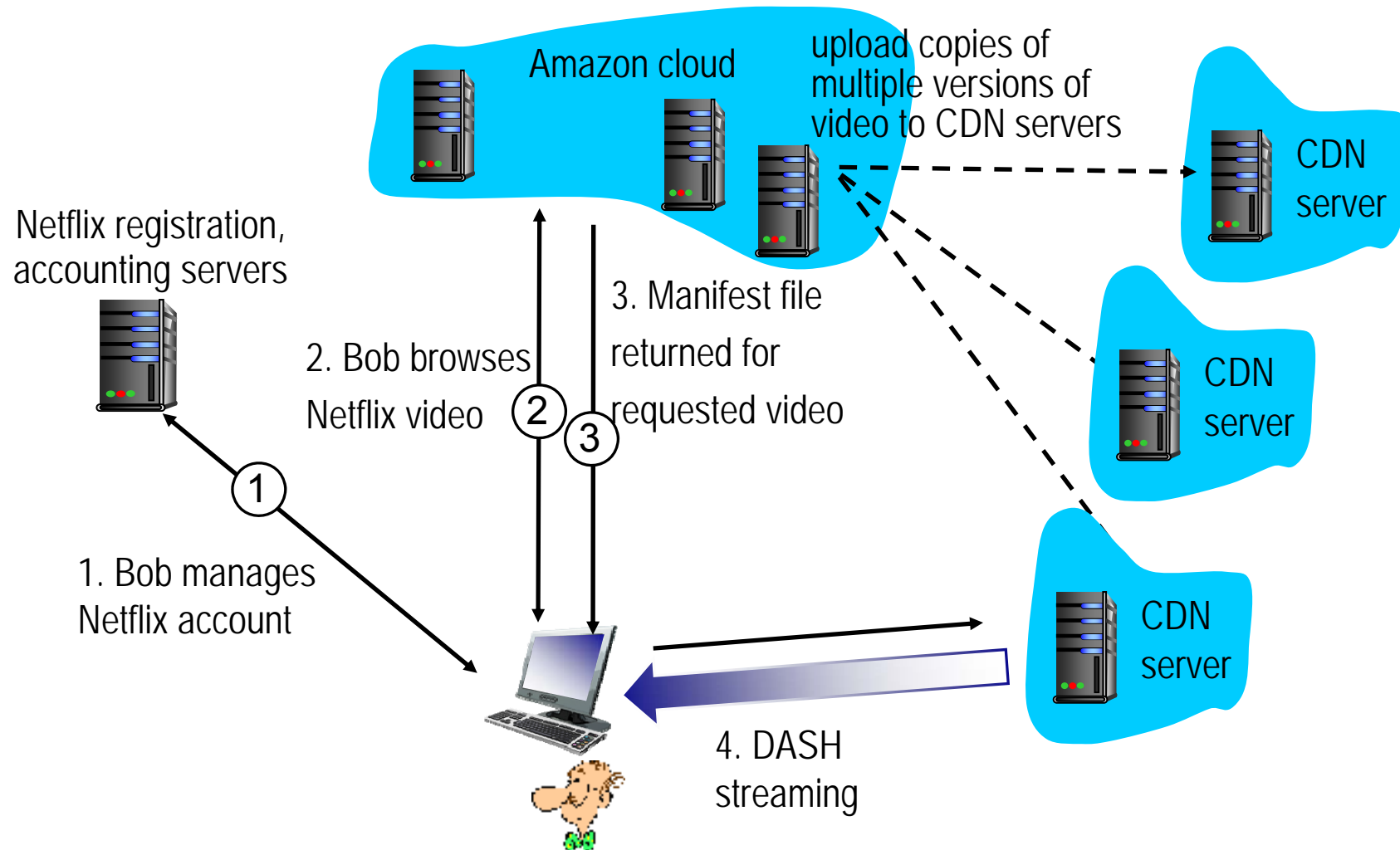
CDN content access: a closer look

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

- video stored in CDN at `a1105.Kingcdn.com`



Case study: Netflix



Acknowledgement

- ❖ These notes are adapted from the publishers material.
- ❖ All material copyright 1996-2016
J. F Kurose and K. W. Ross
All Rights Reserved.

