Chapter 2
Application Layer
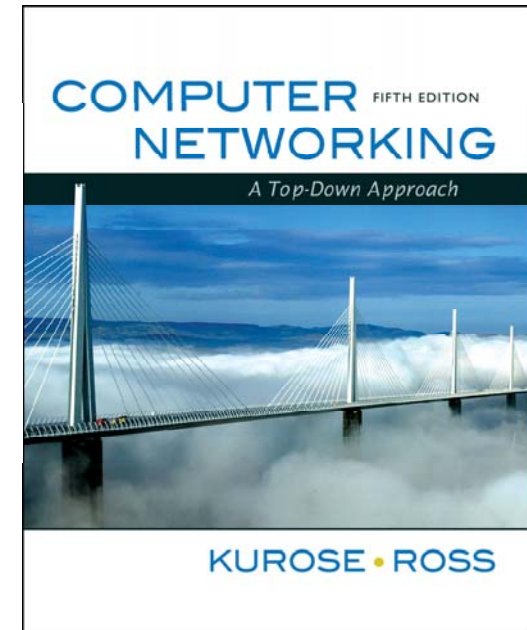
A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers).
They're in PowerPoint form so you can add, modify, and delete slides
(including this one) and slide content to suit your needs. They obviously
represent a *lot* of work on our part. In return for use, we only ask the
following:
❑ If you use these slides (e.g., in a class) in substantially unaltered form,
that you mention their source (after all, we'd like people to use our book!)
❑ If you post any slides in substantially unaltered form on a www site, that
you note that they are adapted from (or perhaps identical to) our slides, and
note our copyright of this material.

Thanks and enjoy! JFK/KWR

*Computer Networking:
A Top Down Approach,*
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April
2009.

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - ❖ SMTP, POP3, IMAP
- 2.5 DNS

- 2.6 P2P applications
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP

Chapter 2: Application Layer

Our goals:

□ conceptual, implementation aspects of network application protocols

  ❖ transport-layer service models

  ❖ client-server paradigm

  ❖ peer-to-peer paradigm

□ learn about protocols by examining popular application-level protocols

  ❖ HTTP

  ❖ FTP

  ❖ SMTP / POP3 / IMAP

  ❖ DNS

□ programming network applications

  ❖ socket API

Some network apps

- e-mail
- web
- instant messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video clips

- voice over IP
- real-time video conferencing
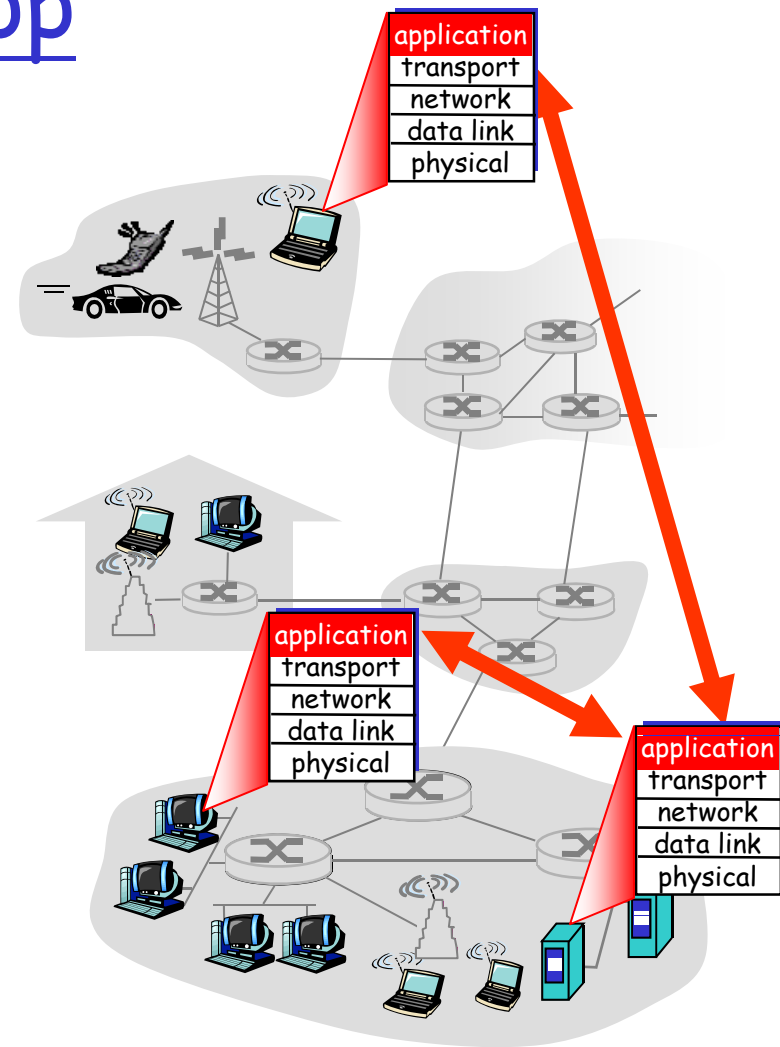- grid computing
-
-
-

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
transport
network
data link
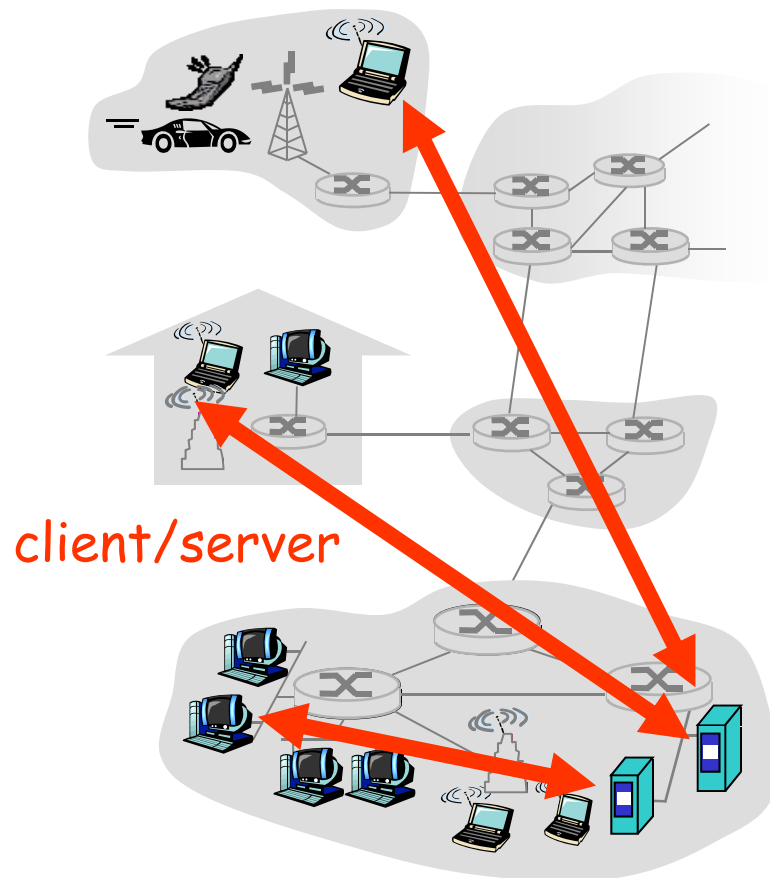physical

application
transport
network
data link
physical

application
transport
network
data link
physical

Chapter 2: Application layer

Application architectures

□ Client-server

□ Peer-to-peer (P2P)

□ Hybrid of client-server and P2P

Client-server architecture

client/server

server:

- ❖ always-on host
- ❖ permanent IP address
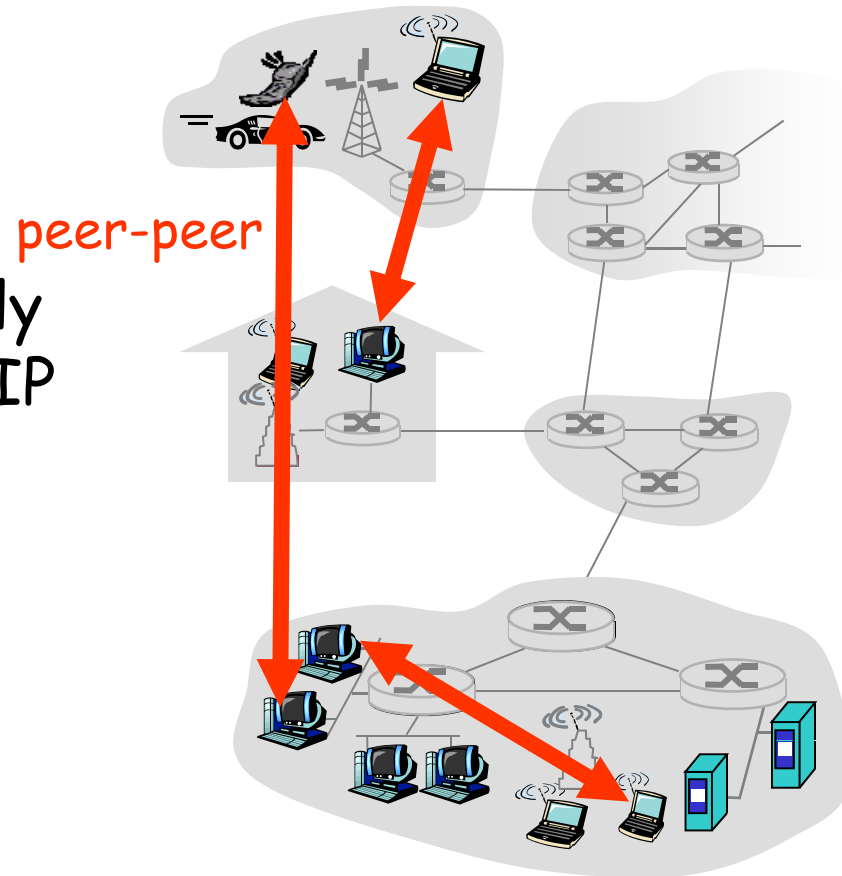- ❖ server farms for scaling

clients:

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

Pure P2P architecture

□ *no* always-on server

□ arbitrary end systems directly communicate

□ peers are intermittently connected and change IP addresses

Highly scalable but difficult to manage

peer-peer

Hybrid of client-server and P2P

Skype

* ❖ voice-over-IP P2P application
* ❖ centralized server: finding address of remote party:
* ❖ client-client connection: direct (not through server)

Instant messaging

* ❖ chatting between two users is P2P
* ❖ centralized service: client presence detection/location
    * user registers its IP address with central server when it comes online
    * user contacts central server to find IP addresses of buddies

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
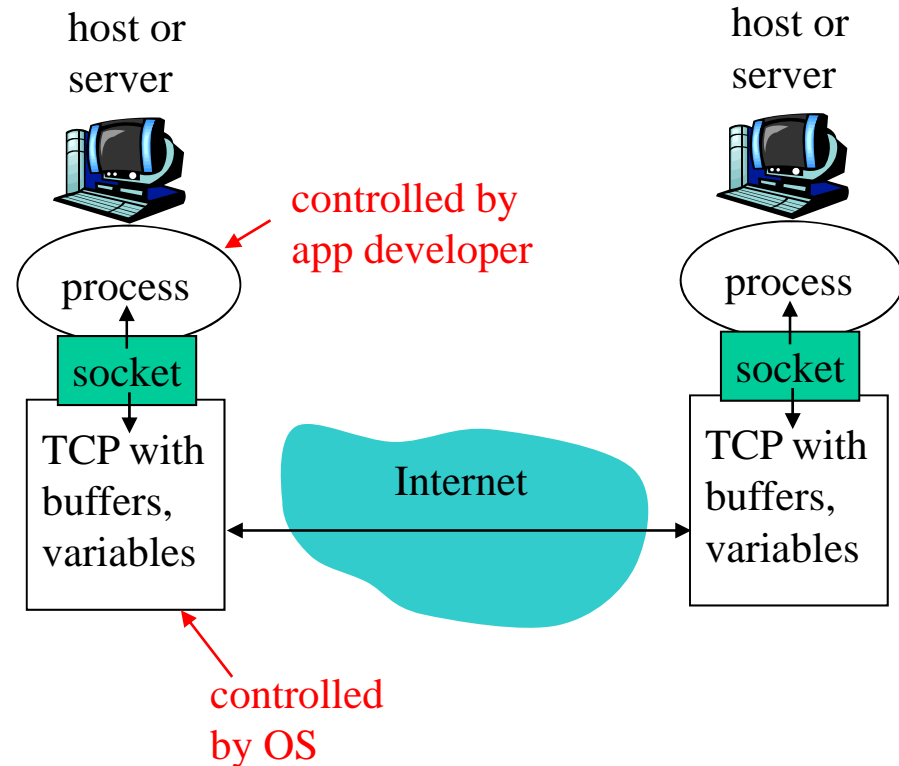
Client process: process that initiates communication

Server process: process that waits to be contacted

□ Note: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its **socket**
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process

host or server

process

controlled by app developer

socket

TCP with buffers, variables

controlled by OS

Internet

host or server

process

socket

TCP with buffers, variables

- API: (1) choice of transport protocol; (2) ability to fix a few parameters (lots more on this later)

Addressing processes

□ to receive messages, process must have *identifier*

□ host device has unique 32-bit IP address

□ *Q:* does IP address of host suffice for identifying the 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

Public-domain protocols:

□ defined in RFCs

□ allows for interoperability

□ e.g., HTTP, SMTP

Proprietary protocols:

□ e.g., Skype

What transport service does an app need?

Data loss

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

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 of 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
instant messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *connection-oriented:* setup required between client and server processes
- *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 guarantees, security

UDP service:

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

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 (eg Youtube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	typically UDP

Chapter 2: Application layer

Web and HTTP

<u>First some jargon</u>

☐ 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

☐ Example URL:

`www.someschool.edu/someDept/pic.gif`

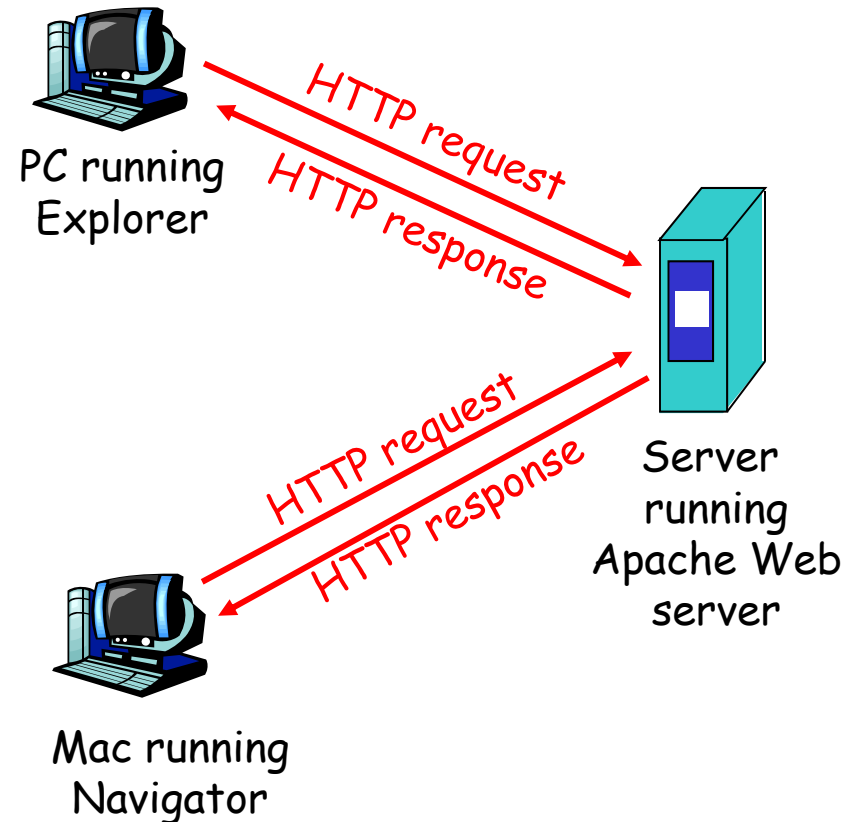host name   path name

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests

PC running Explorer

HTTP request

HTTP response

Mac running Navigator

HTTP request

HTTP response

Server running Apache Web server

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

Nonpersistent HTTP

- At most one object is sent over a TCP connection.

Persistent HTTP

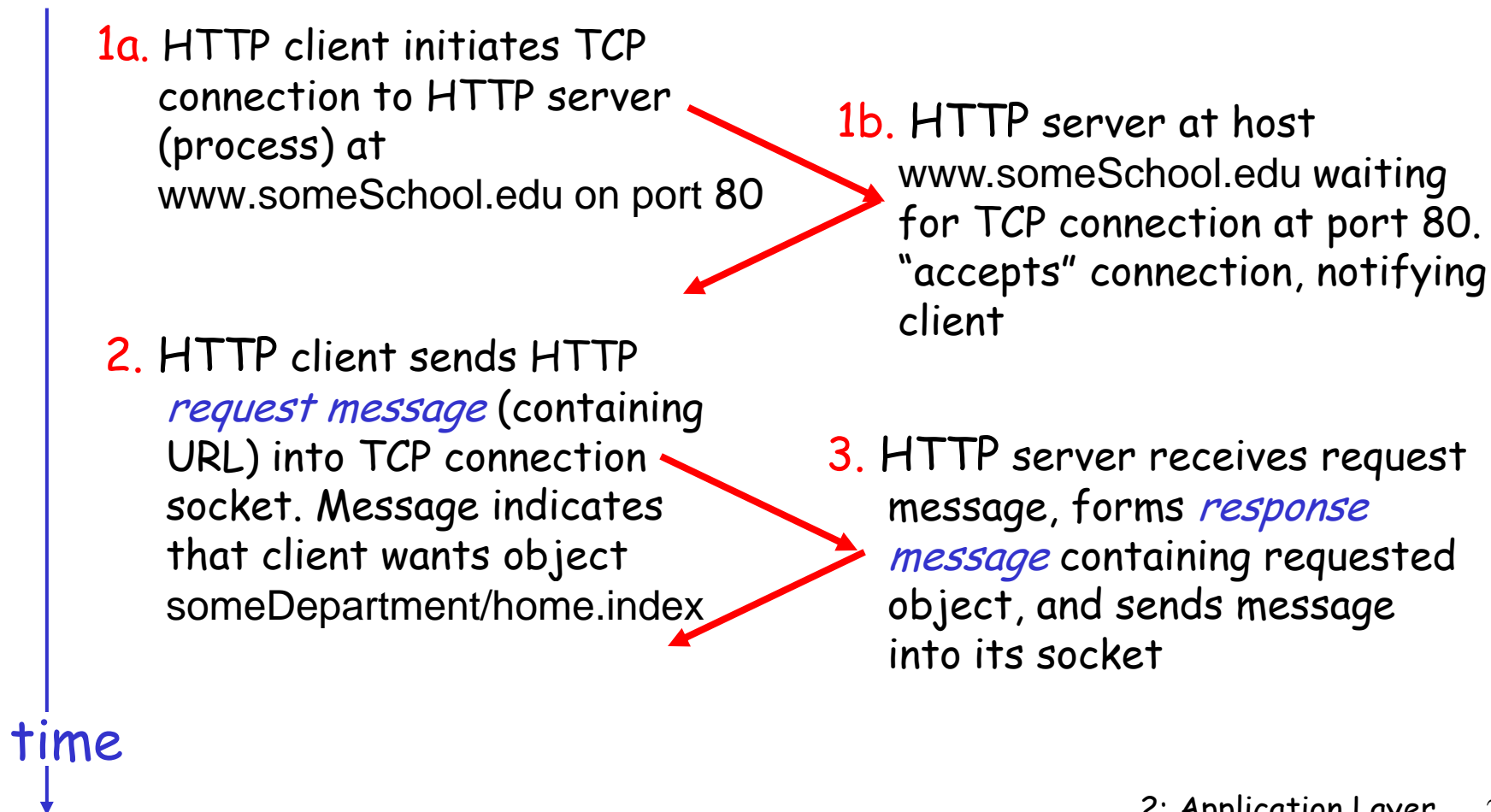- Multiple objects can be sent over single TCP connection between client and server.
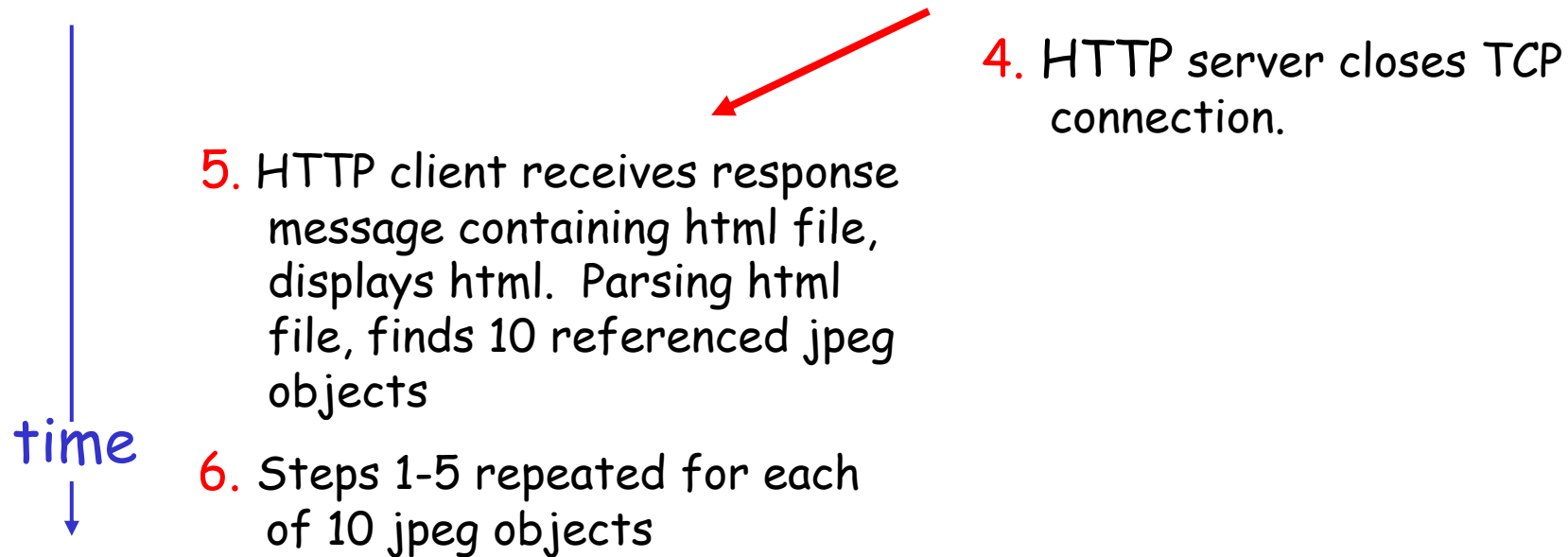
Nonpersistent HTTP

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

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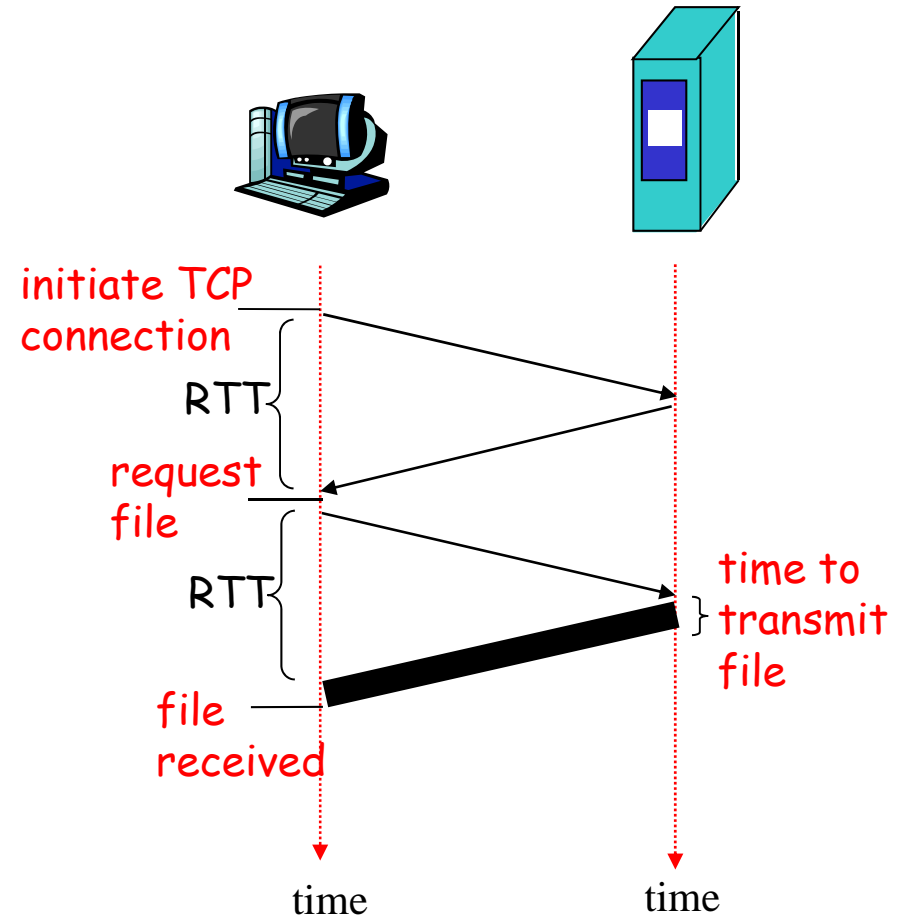
6. Steps 1-5 repeated for each of 10 jpeg objects

Non-Persistent HTTP: Response time

Definition of RTT: time for a small packet to travel from client to server and back.

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

total = 2RTT+transmit time

initiate TCP connection

RTT

request file

RTT

file received

time to transmit file

time   time

Persistent HTTP

Nonpersistent 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:

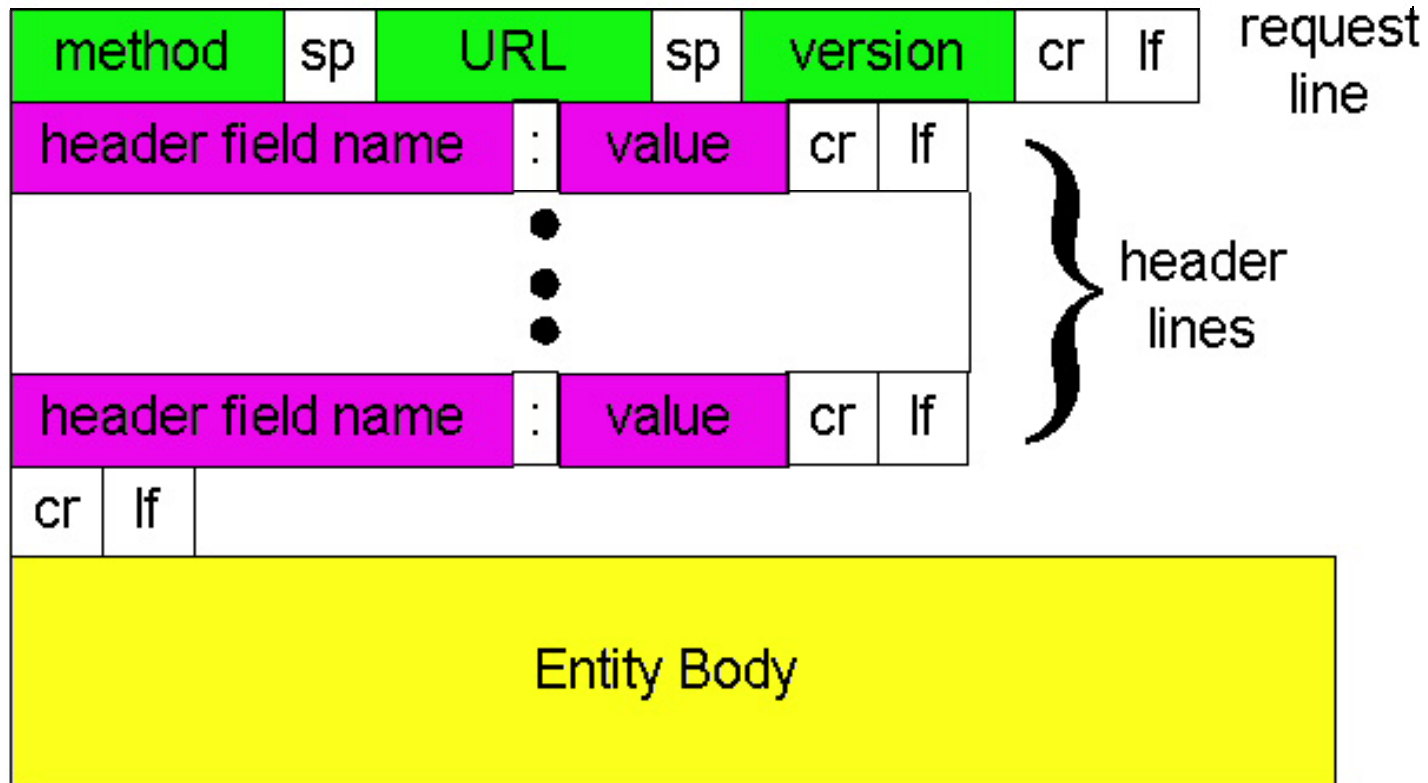 ❖ ASCII (human-readable format)

request line
(GET, POST,
HEAD commands)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language:fr
```

header
lines

Carriage return,
line feed
indicates end
of message

(extra carriage return, line feed)

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:

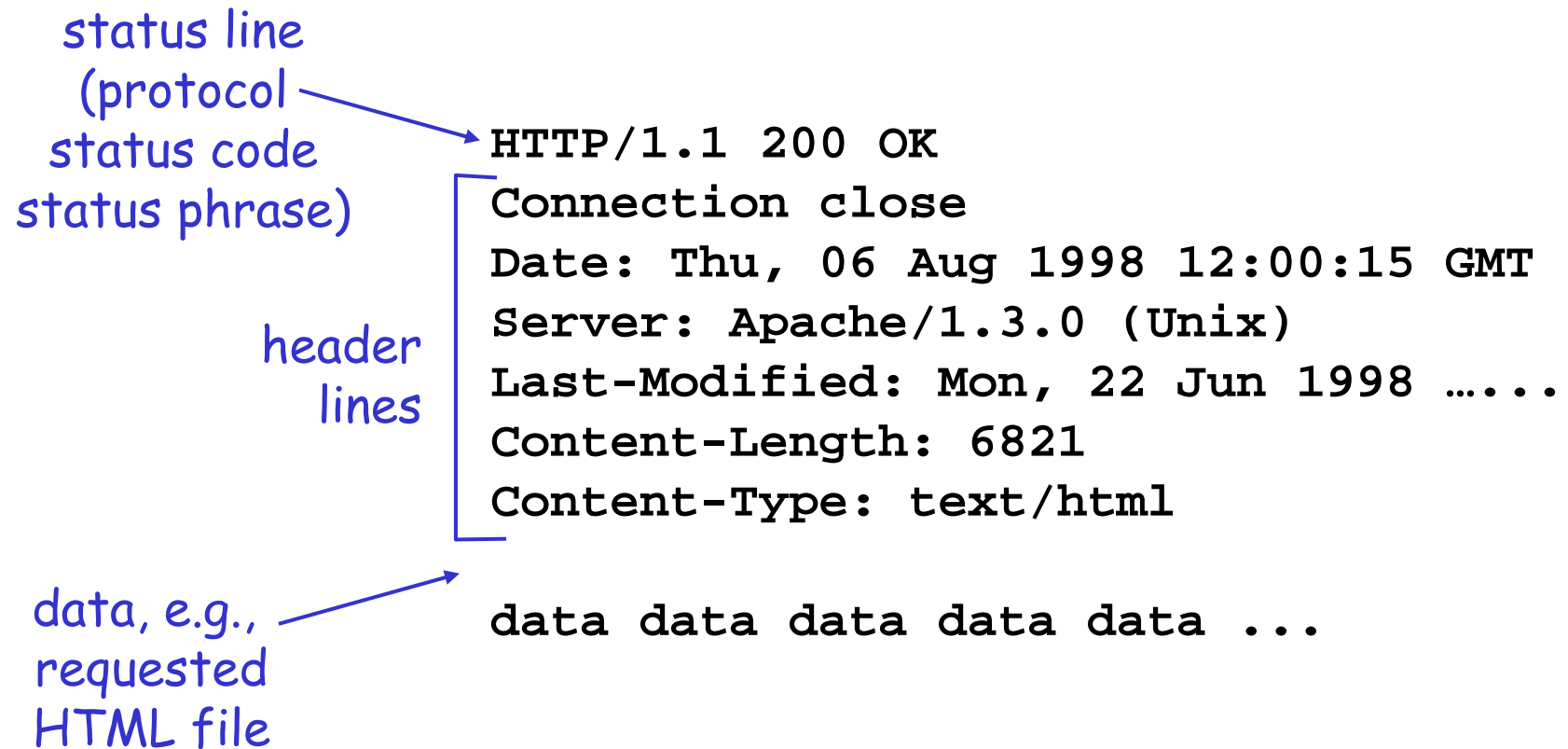`www.somesite.com/animalsearch?monkeys&banana`

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
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

HTTP response status codes

In first line in server->client response message.

A few sample codes:

200 OK

❖ request succeeded, requested object later in this message

301 Moved Permanently

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

400 Bad Request

❖ request message not understood by server

404 Not Found

❖ requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

`telnet cis.poly.edu 80`

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

2. Type in a GET HTTP request:

`GET /~ross/ HTTP/1.1`
`Host: cis.poly.edu`

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

3. Look at response message sent by HTTP server!

User-server state: cookies
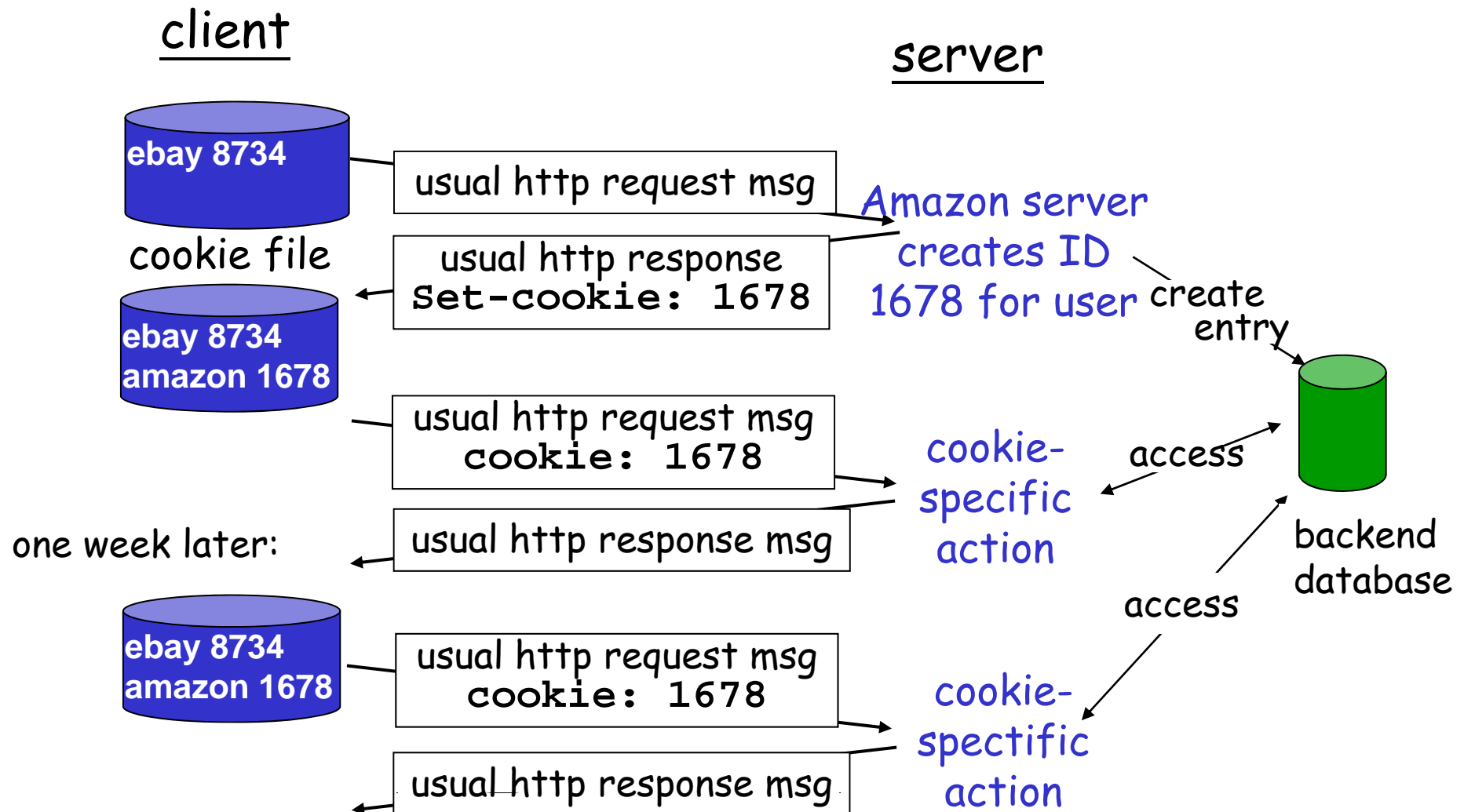
Many major Web sites use cookies

Four components:

    1) cookie header line of HTTP *response* message

    2) cookie header line in 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 always 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.)

client

server

ebay 8734

cookie file

ebay 8734
amazon 1678

one week later:

ebay 8734
amazon 1678

usual http request msg

usual http response
Set-cookie: 1678

usual http request msg
cookie: 1678

usual http response msg

usual http request msg
cookie: 1678

usual http response msg

Amazon server
creates ID
1678 for user

create
entry

cookie-
specific
action

access

cookie-
spectific
action

access

backend
database

Cookies (continued)

What cookies can bring:

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

How to keep "state":

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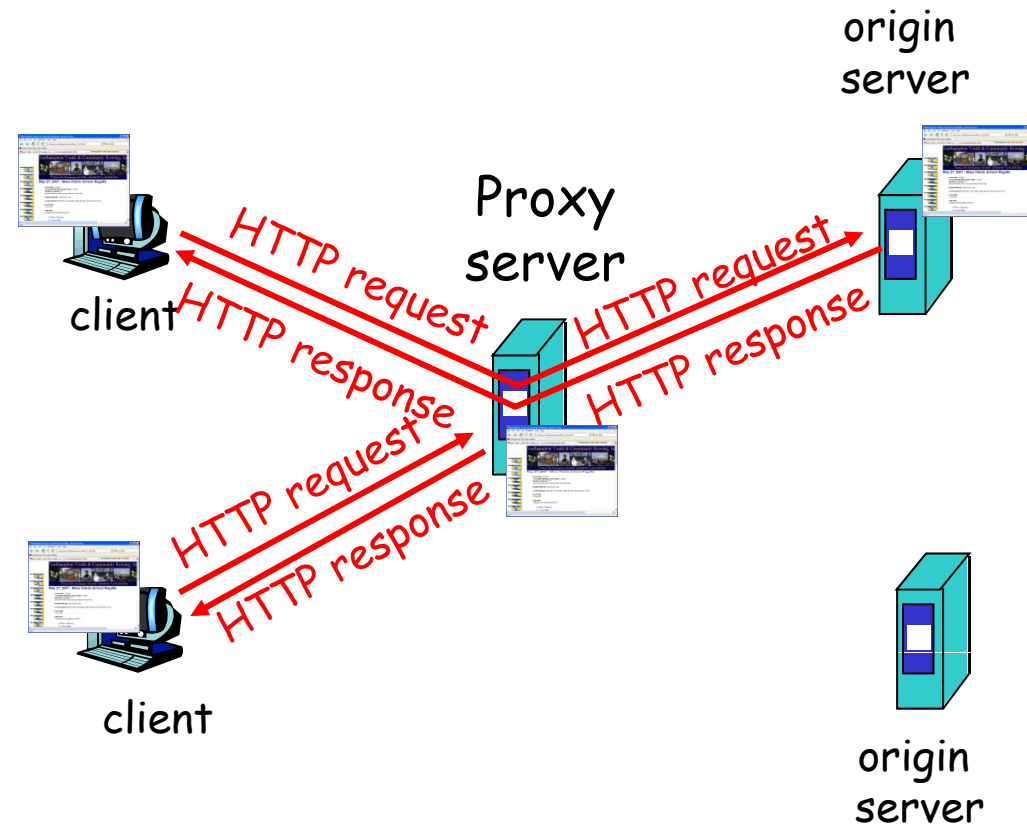- ❑ cookies: http messages carry state

──── aside ────

Cookies and privacy:

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

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

origin server

Proxy server

client

HTTP request
HTTP response

HTTP request
HTTP response

HTTP request
HTTP response

HTTP request
HTTP response

client

origin server

More about Web caching

- cache acts as both client and 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 (but so does P2P file sharing)

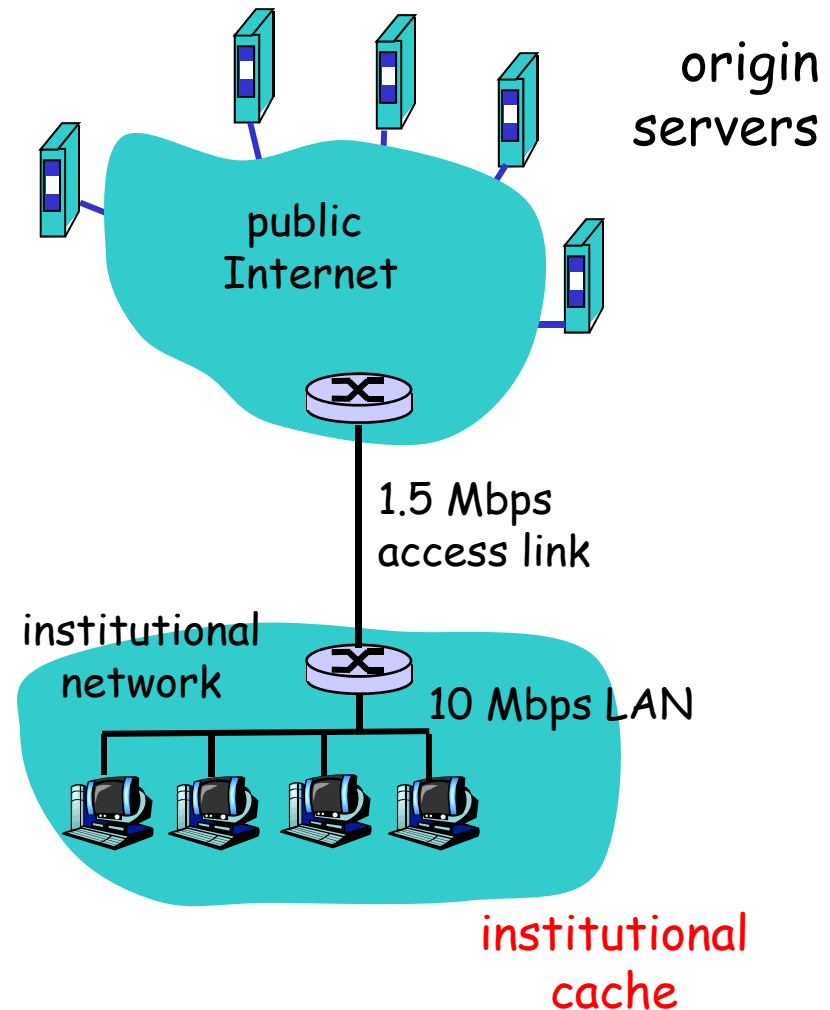# Caching example

Assumptions

- average object size = 100,000 bits
- avg. request rate from institution's browsers to origin servers = 15/sec
- delay from institutional router to any origin server and back to router = 2 sec

Consequences

- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay
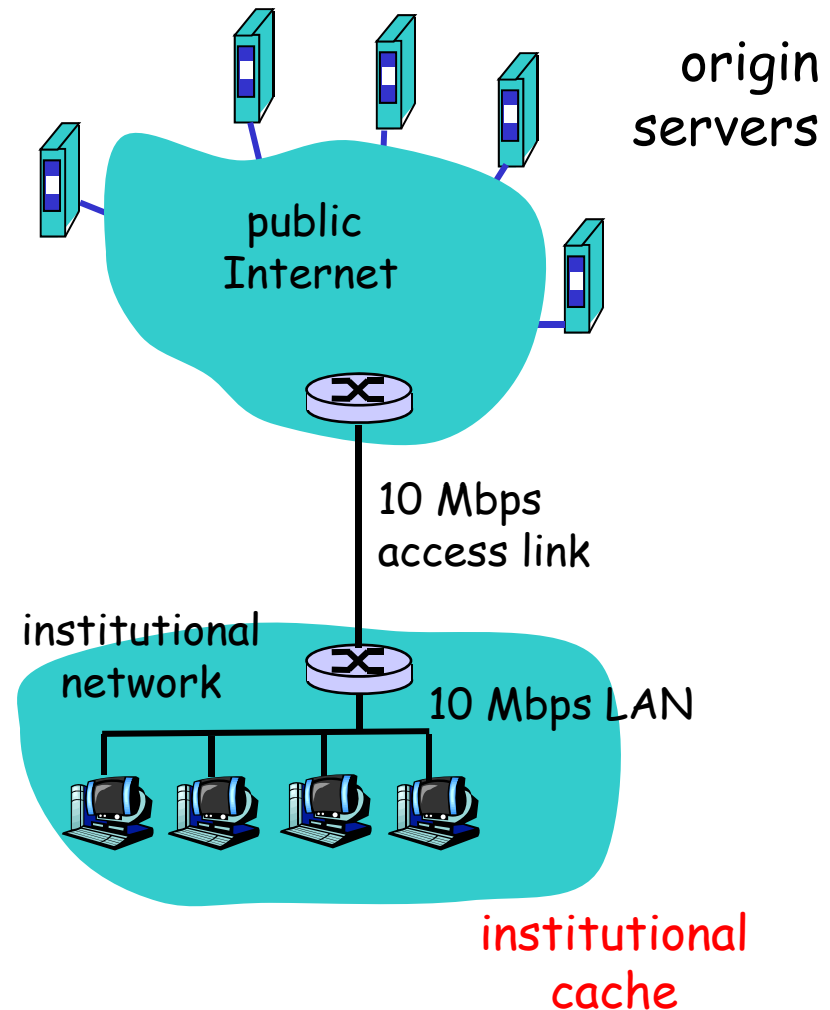 = 2 sec + minutes + milliseconds

origin servers

public Internet

1.5 Mbps access link

institutional network

10 Mbps LAN

institutional cache

Caching example (cont)

possible solution

☐ increase bandwidth of access link to, say, 10 Mbps

consequence

☐ utilization on LAN = 15%

☐ utilization on access link = 15%

☐ Total delay = Internet delay + access delay + LAN delay

  = 2 sec + msecs + msecs

☐ often a costly upgrade

origin servers

public Internet

10 Mbps access link
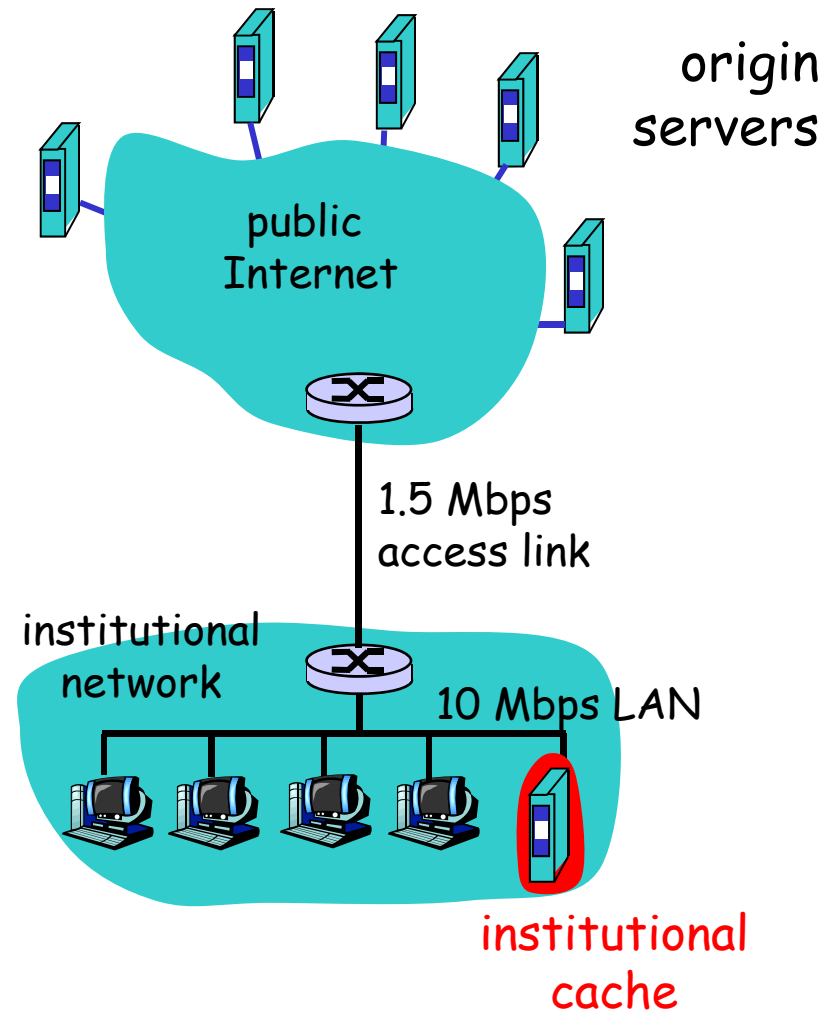
institutional network

10 Mbps LAN

institutional cache

Caching example (cont)

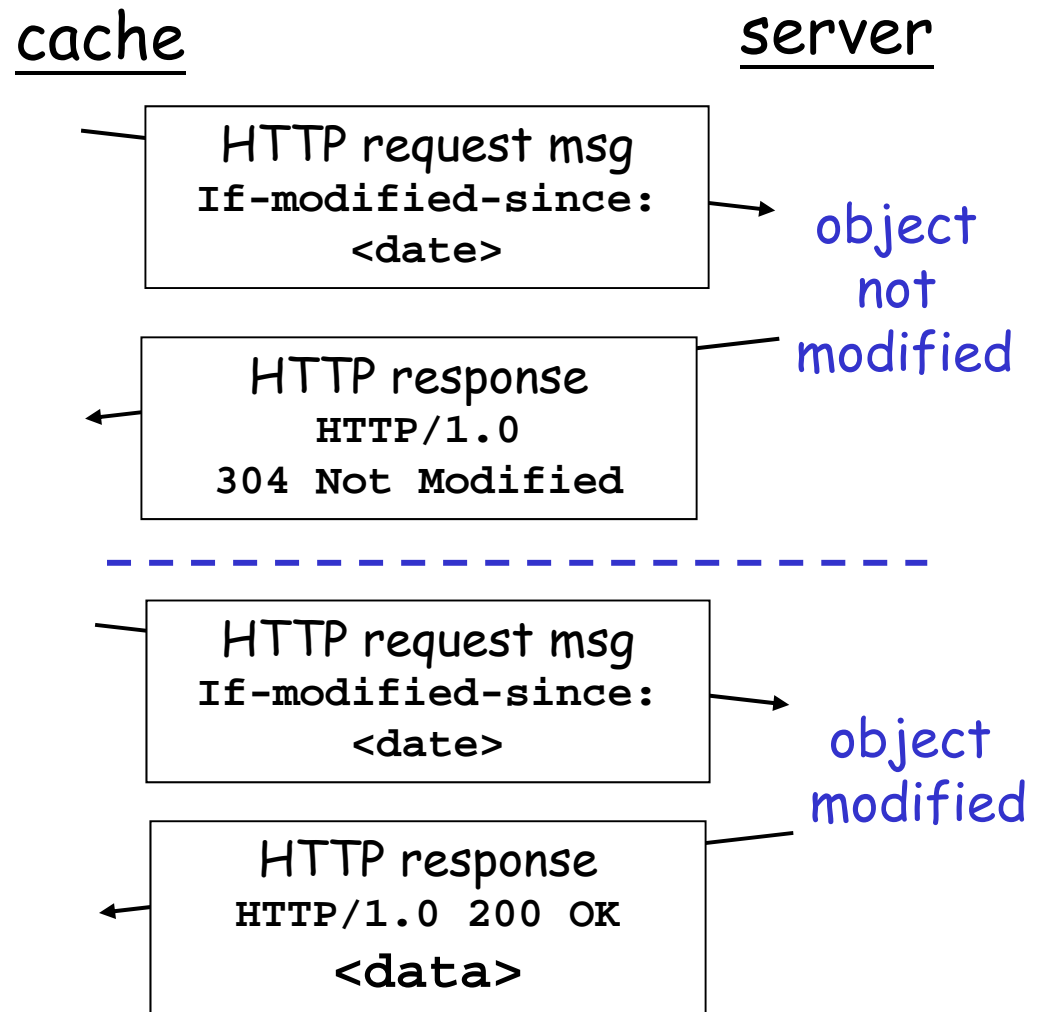possible solution: install cache

□ suppose hit rate is 0.4

consequence

□ 40% requests will be satisfied almost immediately

□ 60% requests satisfied by origin server

□ utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)

□ total avg delay = Internet delay + access delay + LAN delay = .6*(2.01) secs + .4*milliseconds < 1.4 secs

origin servers

public Internet

1.5 Mbps access link

institutional network
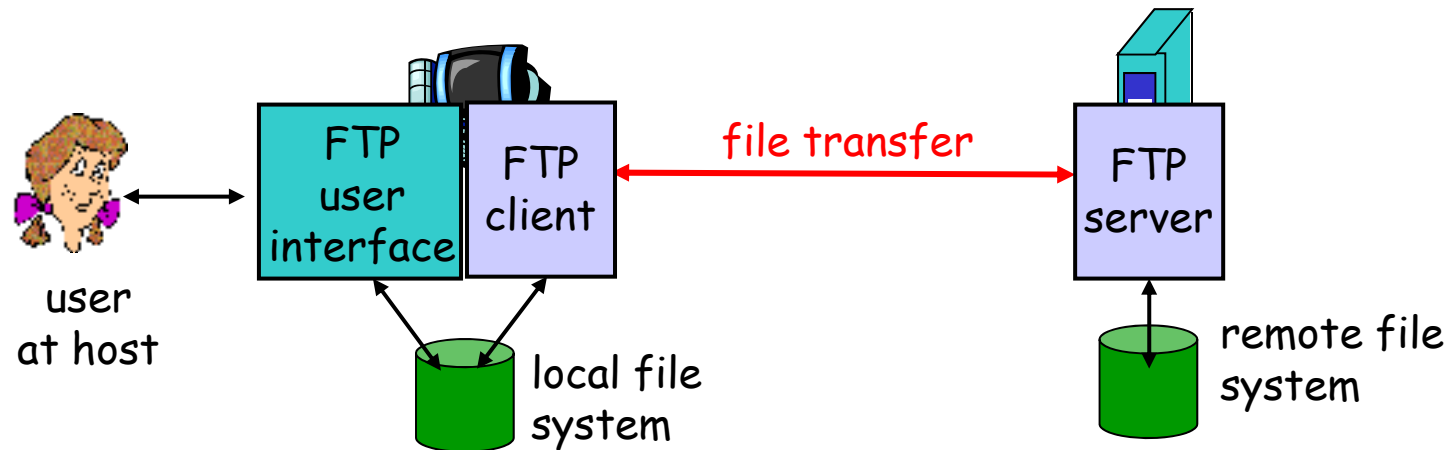
10 Mbps LAN

institutional cache

Conditional GET

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

cache   server

HTTP request msg
If-modified-since:
<date>

object not modified

HTTP response
HTTP/1.0
304 Not Modified

HTTP request msg
If-modified-since:
<date>

object modified

HTTP response
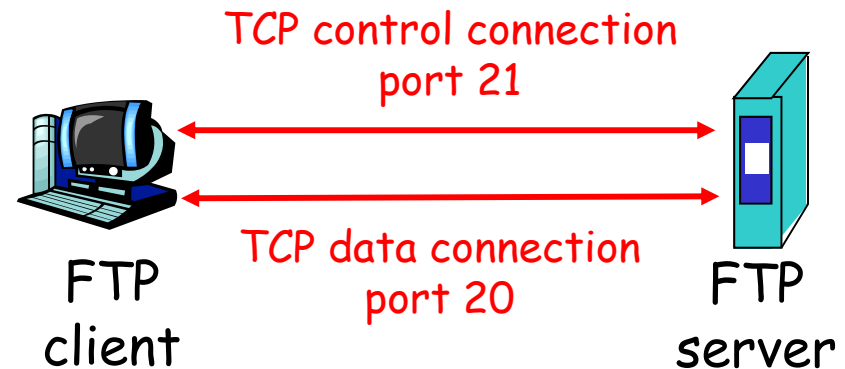**HTTP/1.0 200 OK**
<data>

Chapter 2: Application layer

FTP: the file transfer protocol

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

FTP: separate control, data connections

TCP control connection
port 21

TCP data connection
port 20

FTP client

FTP server

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

- server opens another TCP data connection to transfer another file.
- control connection: "out of band"
- FTP server maintains "state": current directory, earlier authentication

FTP commands, responses

Sample commands:

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

Sample return codes

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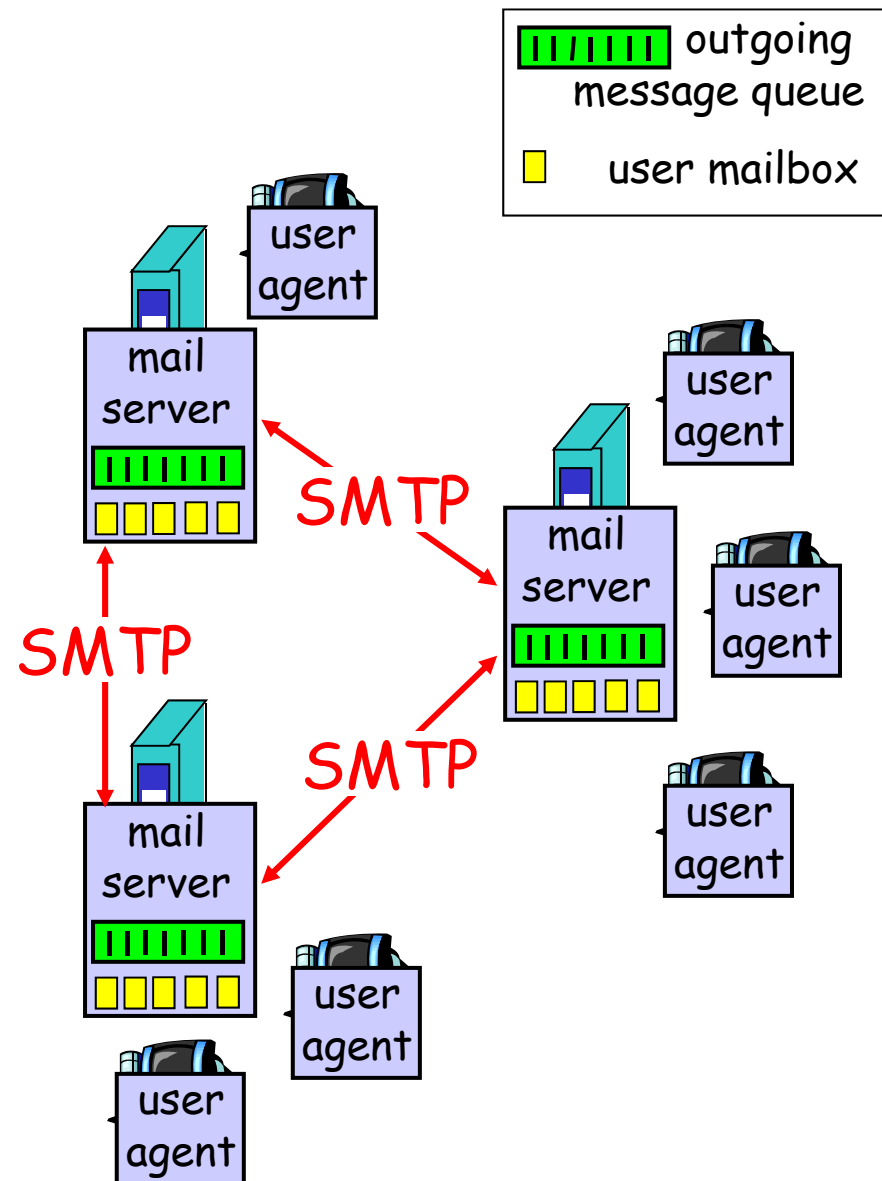- **452 Error writing file**

Chapter 2: Application layer

Electronic Mail

Three major components:

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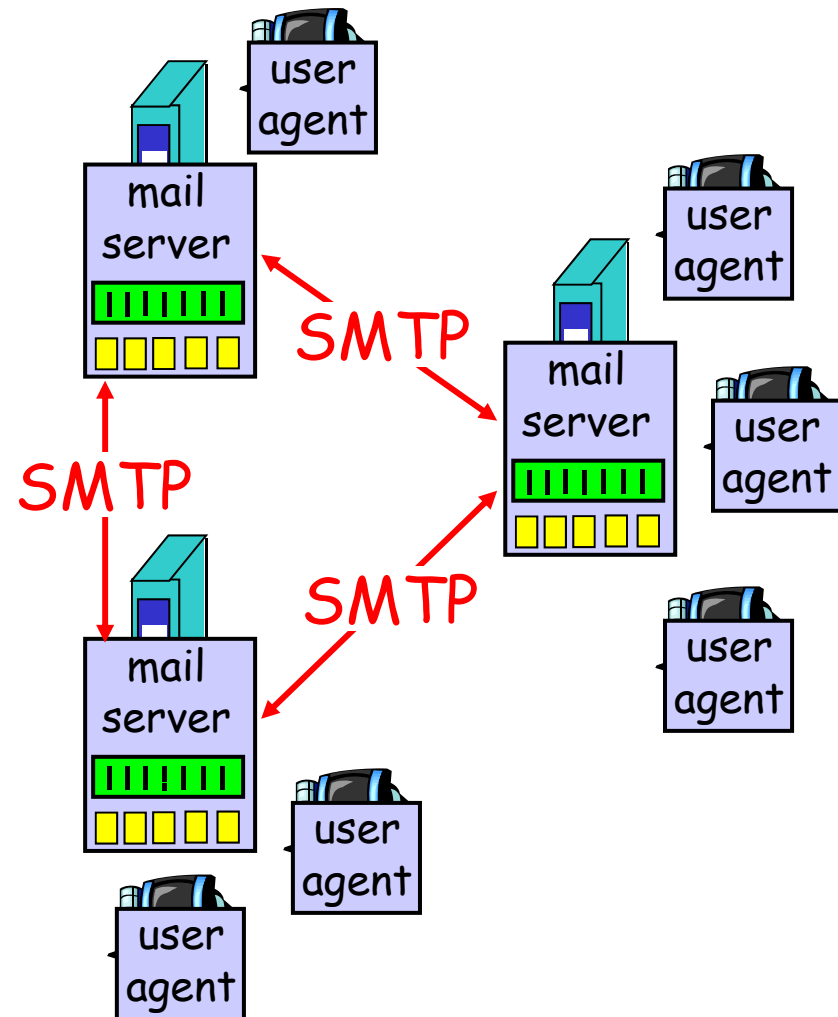- mail servers
- simple mail transfer protocol: SMTP

User Agent

- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Mozilla Thunderbird
- outgoing, incoming messages stored on server

outgoing message queue

user mailbox

user agent

mail server

SMTP

SMTP

SMTP

mail server

user agent

user agent

user agent

mail server

user agent

user agent

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
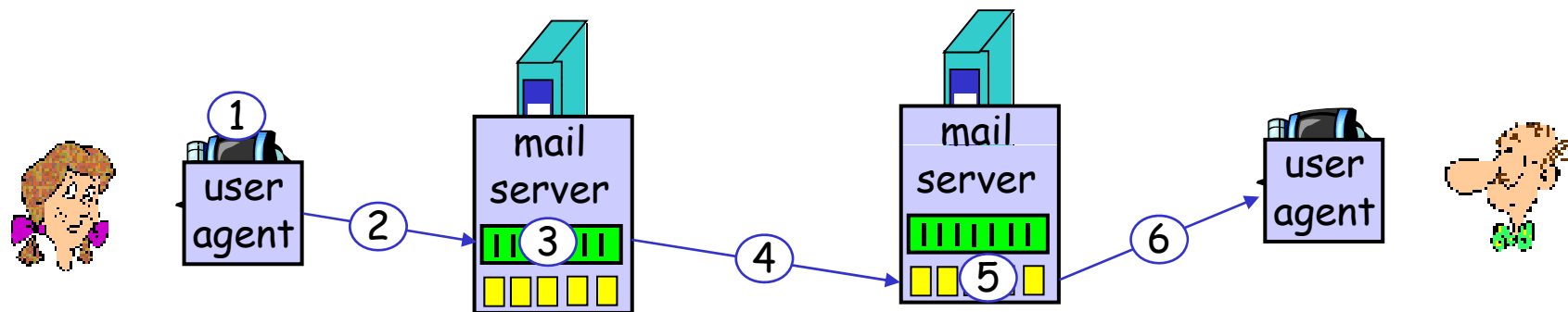 - ❖ client: sending mail server
  - ❖ "server": receiving mail server

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
   ❖ 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 and "to" `bob@someschool.edu`

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

3) Client side of SMTP opens TCP connection with Bob's mail server

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

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

6) Bob invokes his user agent to read message

Sample SMTP interaction

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

Try SMTP interaction for yourself:

□ `telnet servername 25`

□ see 220 reply from server

□ enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

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

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

Comparison with HTTP:

- HTTP: pull
- SMTP: push

- both have ASCII command/response interaction, status codes

- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg
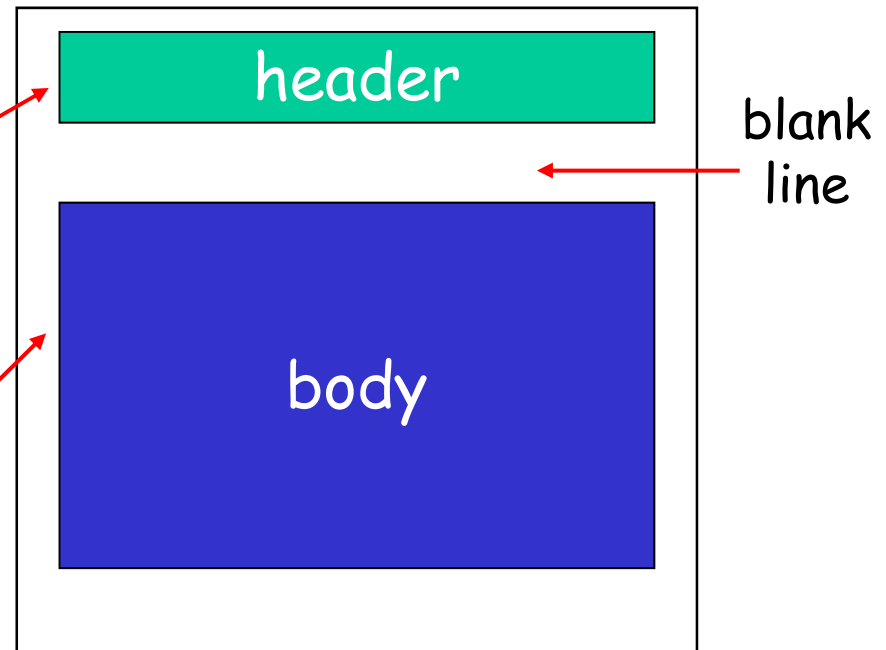
Mail message format

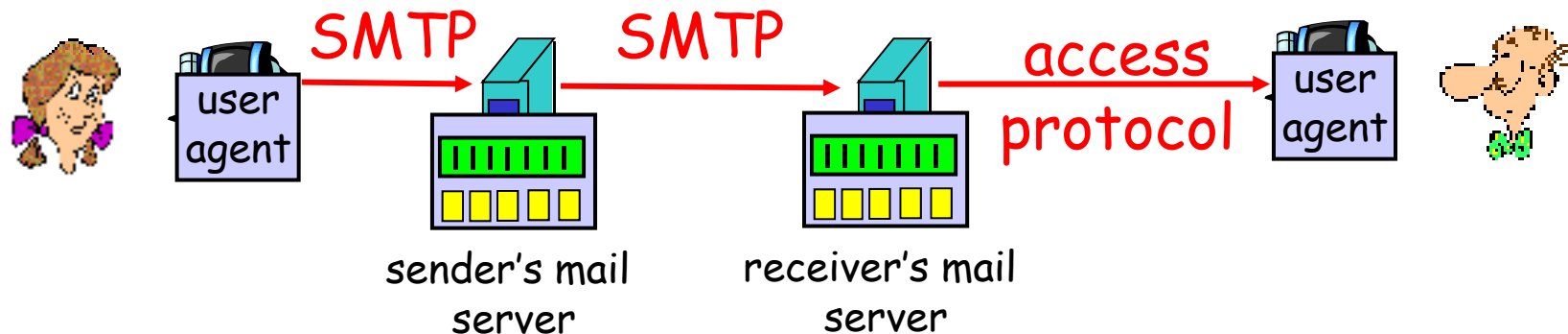SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

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

  different from SMTP commands!

- body
  - the "message", ASCII characters only

header

body

blank line

Mail access protocols

SMTP   SMTP   access protocol

user agent   sender's mail server   receiver's mail server   user agent

□ SMTP: delivery/storage to receiver's server
□ Mail access protocol: retrieval from server
  ❖ POP: Post Office Protocol [RFC 1939]
    • authorization (agent <-->server) and download
  ❖ IMAP: Internet Mail Access Protocol [RFC 1730]
    • more features (more complex)
    • manipulation of stored msgs on server
  ❖ HTTP: gmail, Hotmail, Yahoo! Mail, etc.

POP3 protocol

authorization phase

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

transaction phase, client:

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

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

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

POP3 (more) and IMAP

More about POP3

- Previous example uses "download and delete" mode.
- Bob cannot re-read e-mail if he changes client
- "Download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

IMAP

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

Chapter 2: Application layer

DNS: Domain Name System

People: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., ww.yahoo.com - used by humans

Q: map between IP addresses and name ?

Domain Name System:

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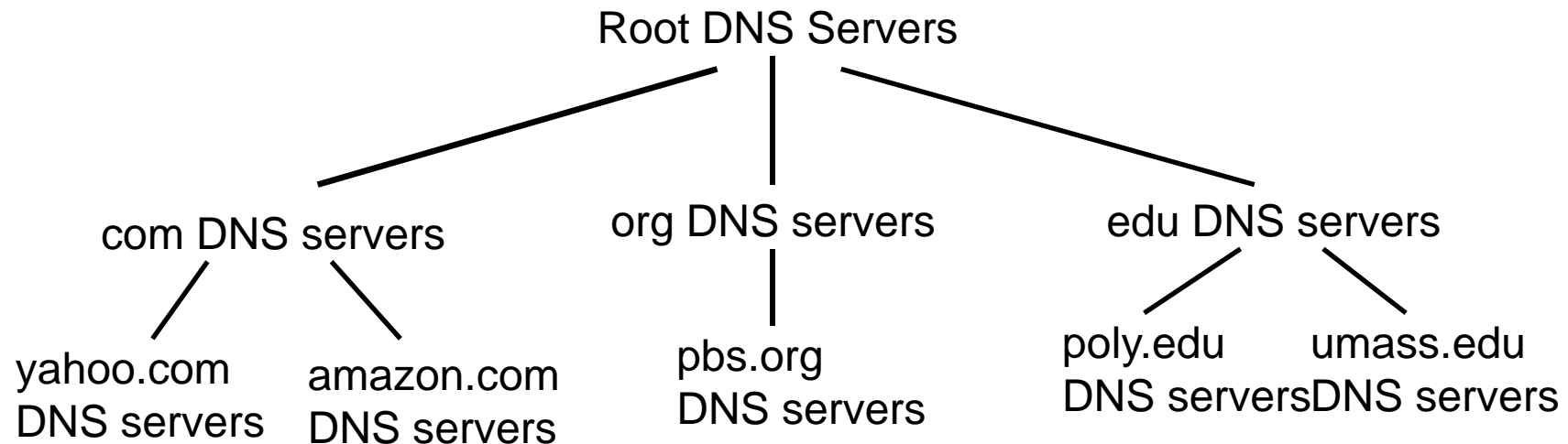 - complexity at network's "edge"

DNS

DNS services

- hostname to IP address translation
- host aliasing
  - Canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?

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

doesn't *scale!*

Distributed, Hierarchical Database

Root DNS Servers

com DNS servers   org DNS servers   edu DNS servers

yahoo.com   amazon.com   pbs.org   poly.edu   umass.edu
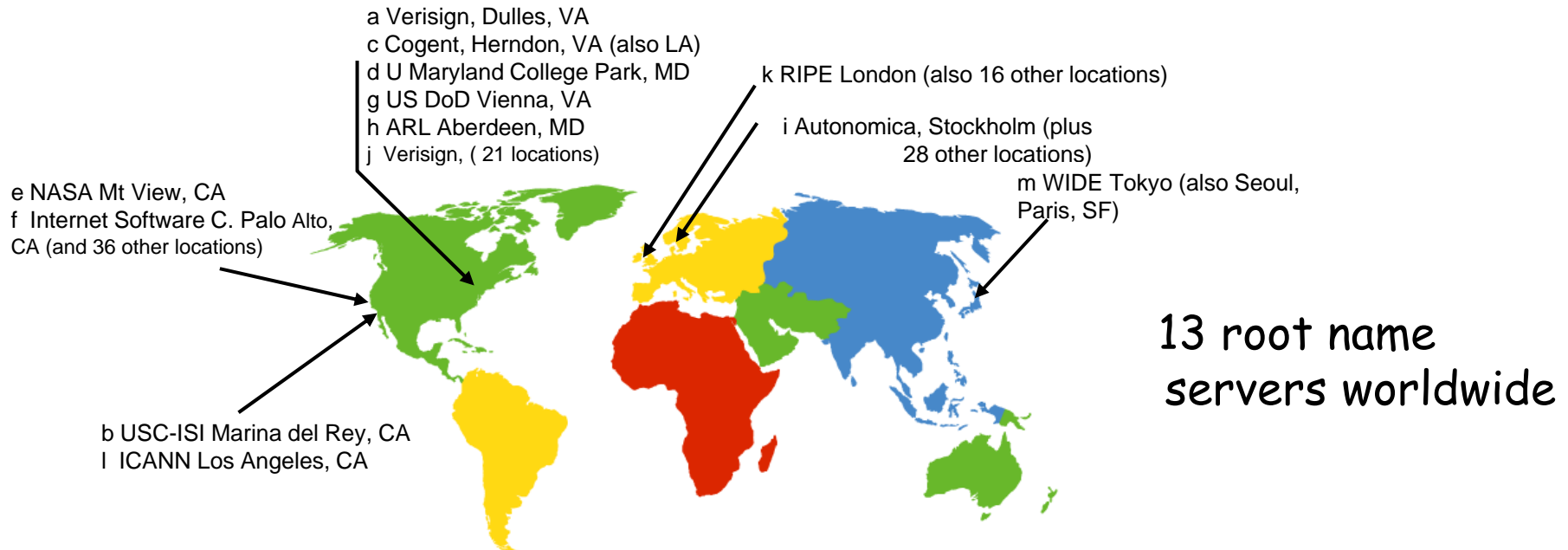DNS servers   DNS servers   DNS servers   DNS serversDNS servers

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

☐ client queries a 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

a Verisign, Dulles, VA
c Cogent, Herndon, VA (also LA)
d U Maryland College Park, MD
g US DoD Vienna, VA
h ARL Aberdeen, MD
j Verisign, (21 locations)

k RIPE London (also 16 other locations)

i Autonomica, Stockholm (plus 28 other locations)

m WIDE Tokyo (also Seoul, Paris, SF)

e NASA Mt View, CA
f Internet Software C. Palo Alto, CA (and 36 other locations)

b USC-ISI Marina del Rey, CA
l ICANN Los Angeles, CA

13 root name servers worldwide

TLD and Authoritative Servers

□ **Top-level domain (TLD) servers:**

❖ responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.

❖ Network Solutions maintains servers for com TLD

❖ Educause for edu TLD

□ **Authoritative DNS servers:**

❖ organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web, mail).

❖ can be maintained by organization or service provider

Local Name Server

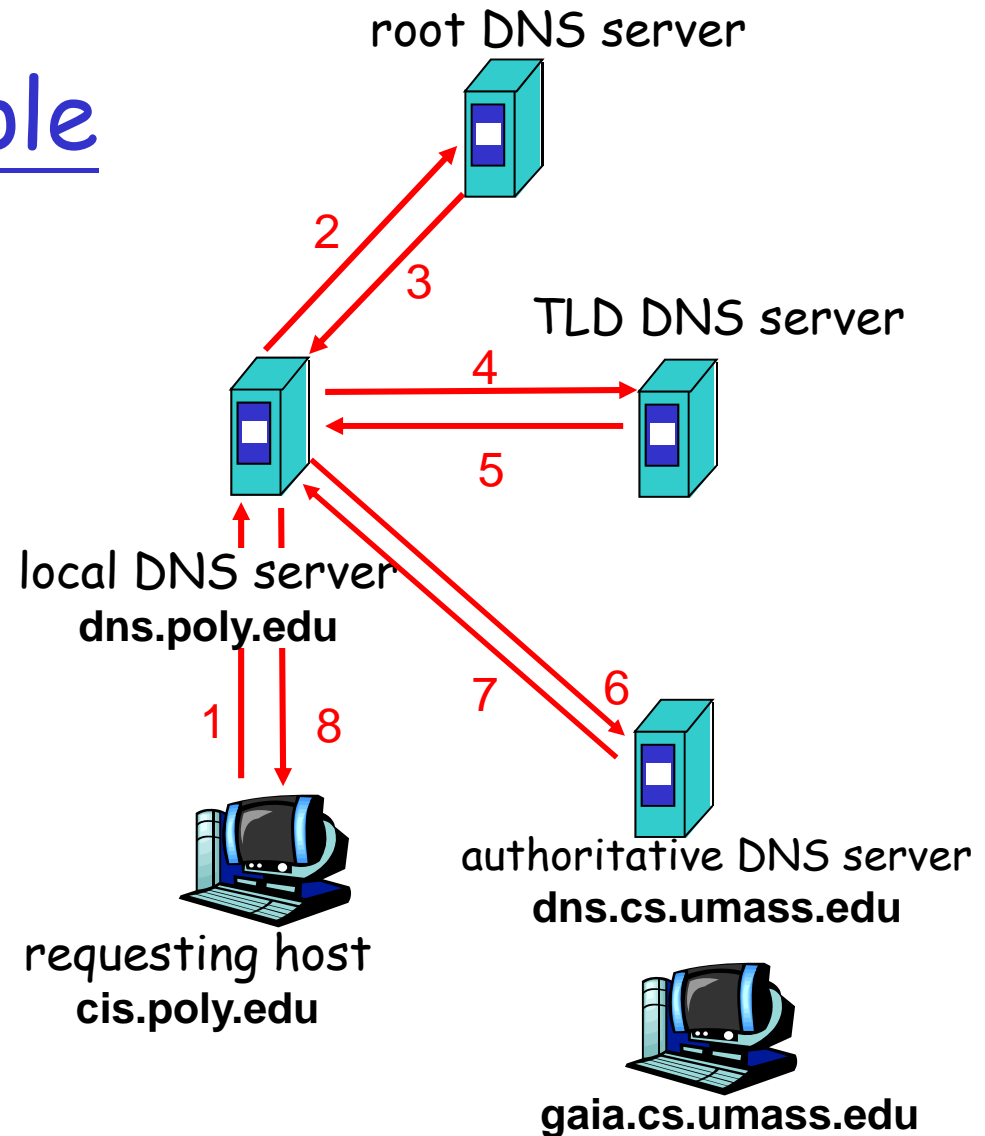□ does not strictly belong to hierarchy

□ 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

  ❖ 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"

root DNS server

TLD DNS server

local DNS server
dns.poly.edu

authoritative DNS server
dns.cs.umass.edu

requesting host
cis.poly.edu

gaia.cs.umass.edu

1 2 3 4 5 6 7 8

DNS name resolution example

recursive query:

- puts burden of name resolution on contacted name server
- heavy load?

root DNS server

2

3

7

6

local DNS server
dns.poly.edu

TLD DNS server

5

4

1

8

authoritative DNS server
dns.cs.umass.edu

requesting host
cis.poly.edu

gaia.cs.umass.edu

DNS: caching and updating records

□ once (any) name server learns mapping, it *caches* mapping
  ❖ cache entries timeout (disappear) after some time
  ❖ TLD servers typically cached in local name servers
    • Thus root name servers not often visited

□ update/notify mechanisms under design by IETF
  ❖ RFC 2136
  ❖ http://www.ietf.org/html.charters/dnsind-charter.html

DNS records

DNS: distributed db storing resource records (RR)

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

- Type=A
  - ❖ `name` is hostname
  - ❖ `value` is IP address

- Type=NS
  - ❖ `name` is domain (e.g. foo.com)
  - ❖ `value` is hostname of authoritative name server for this domain

- Type=CNAME
  - ❖ `name` is alias name for some "canonical" (the real) name
    `www.ibm.com` is really
    `servereast.backup2.ibm.com`
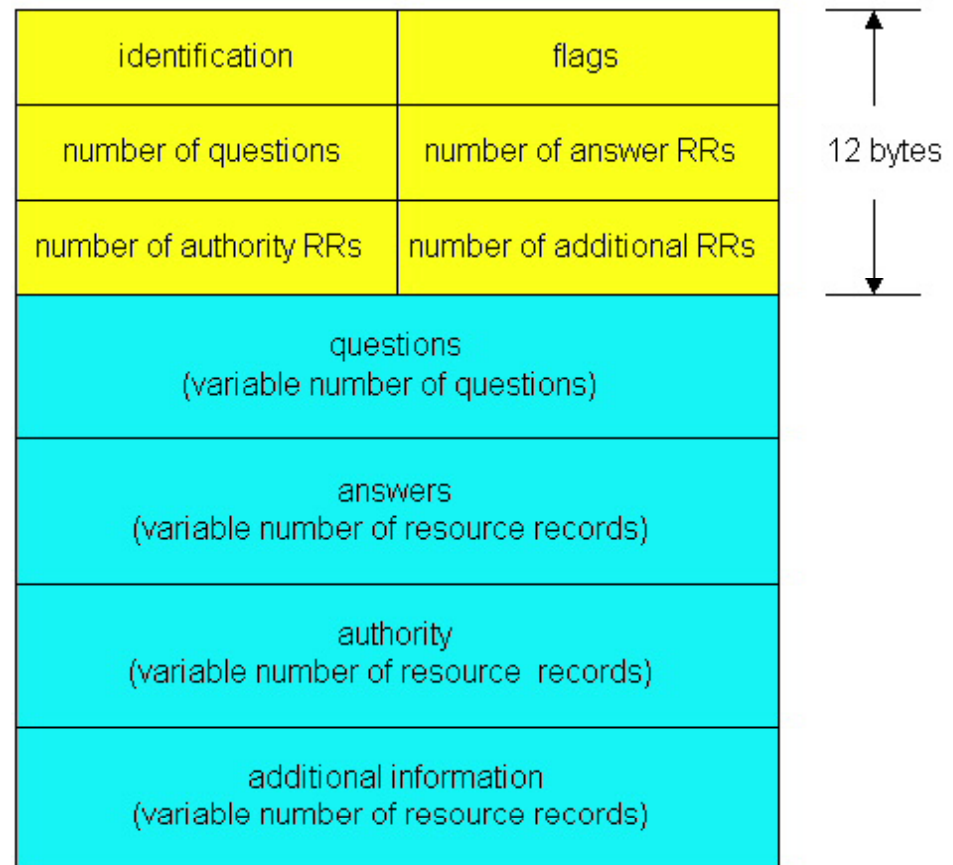 - ❖ `value` is canonical name

- Type=MX
  - ❖ `value` is name of mailserver associated with `name`

DNS protocol, messages

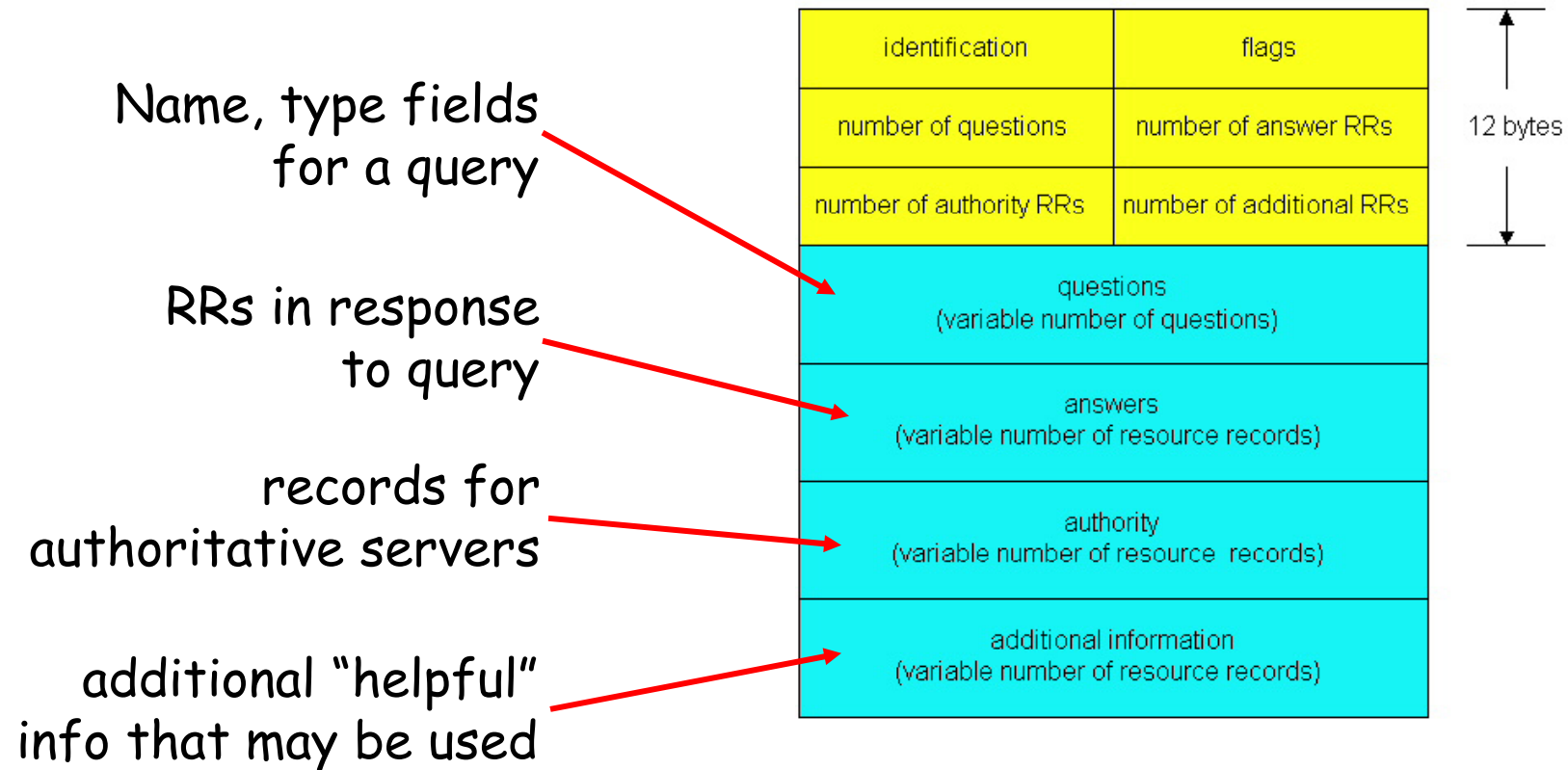DNS protocol : *query* and *reply* messages, both with
same *message format*

msg header

□ identification: 16 bit #
for query, reply to query
uses same #

□ flags:

❖ query or reply
❖ recursion desired
❖ recursion available
❖ reply is authoritative

identification	flags
number of questions	number of answer RRs
number of authority RRs	number of additional RRs

12 bytes

questions
(variable number of questions)

answers
(variable number of resource records)

authority
(variable number of resource records)

additional information
(variable number of resource records)

DNS protocol, messages

Name, type fields for a query

RRs in response to query

records for authoritative servers

additional "helpful" info that may be used

identification	flags
number of questions	number of answer RRs
number of authority RRs	number of additional RRs

12 bytes

questions
(variable number of questions)

answers
(variable number of resource records)

authority
(variable number of resource records)

additional information
(variable number of resource records)

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:

```
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
```
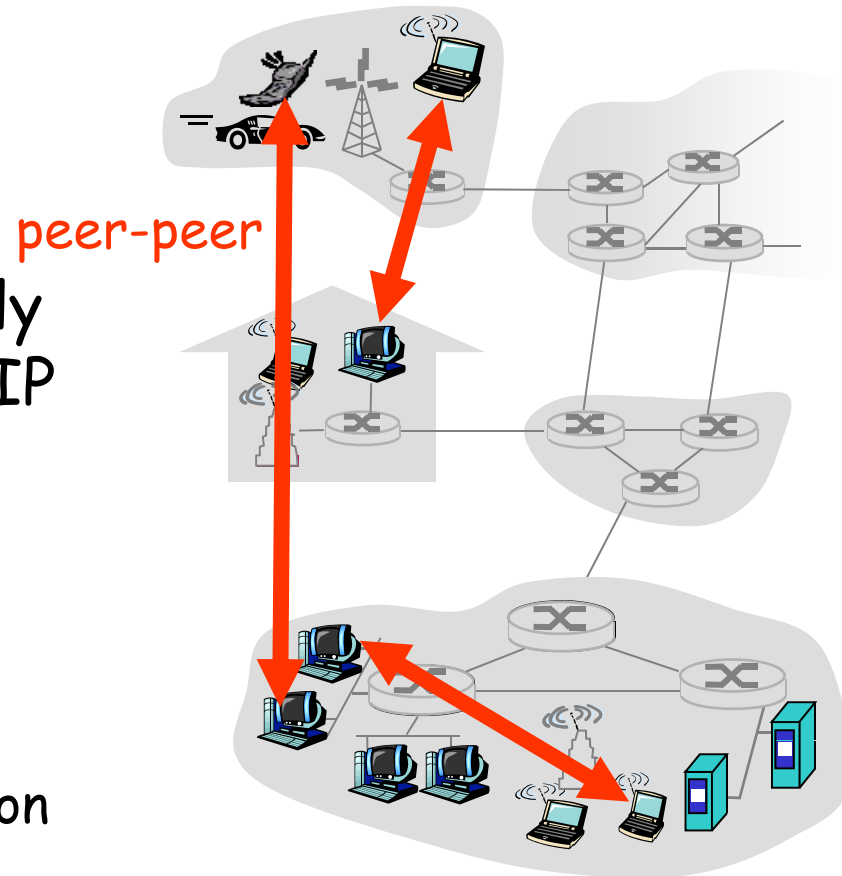
□ create authoritative server Type A record for www.networkuptopia.com; Type MX record for networkutopia.com

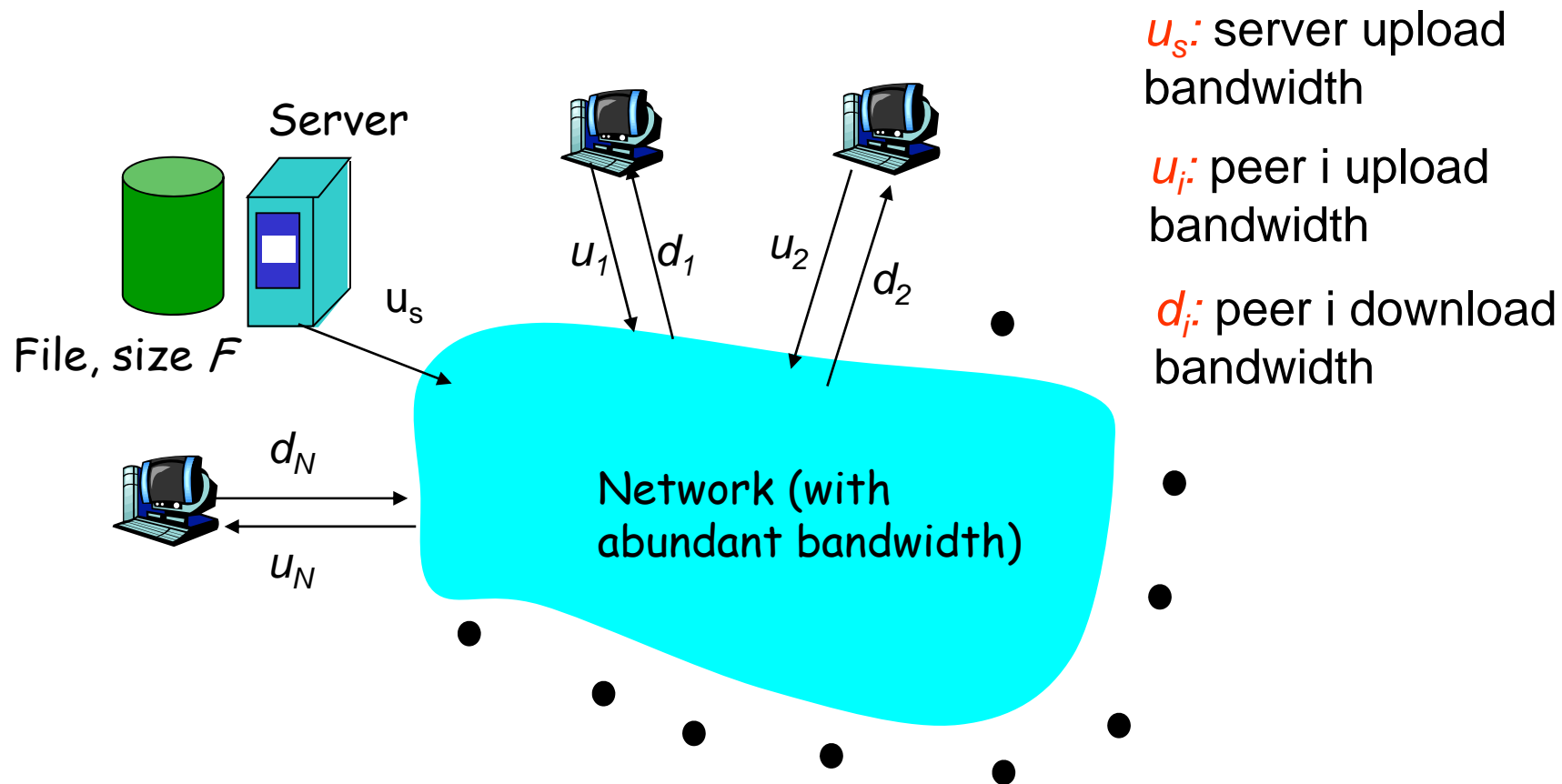□ How do people get IP address of your Web site?

Chapter 2: Application layer

Pure P2P architecture

□ *no* always-on server

□ arbitrary end systems directly communicate

□ peers are intermittently connected and change IP addresses

□ Three topics:
   ❖ File distribution
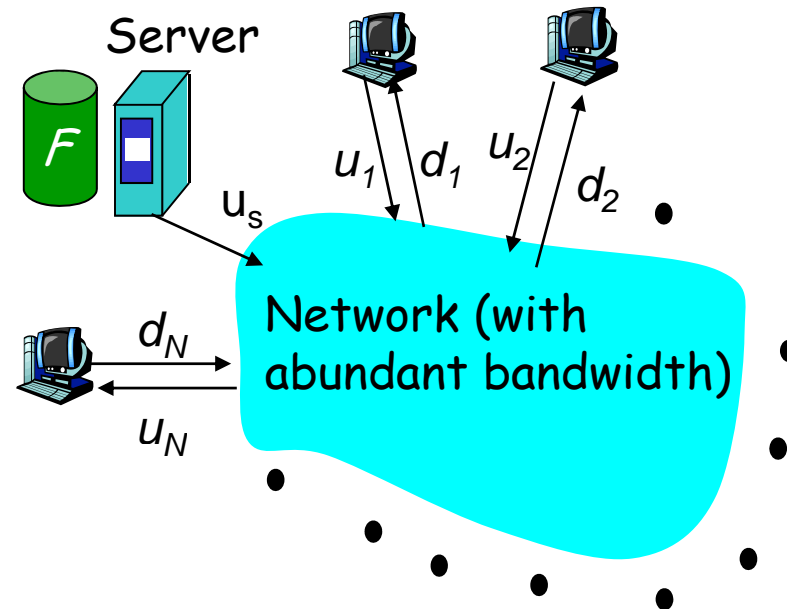   ❖ Searching for information
   ❖ Case Study: Skype

peer-peer

File Distribution: Server-Client vs P2P

Question : How much time to distribute file from one server to *N peers*?

Server

File, size F

u_s

u_1 d_1 u_2 d_2

d_N

u_N

Network (with abundant bandwidth)

u_s: server upload bandwidth

u_i: peer i upload bandwidth

d_i: peer i download bandwidth

File distribution time: server-client

□ **server sequentially sends N copies:**
  ❖ NF/u_s time

□ **client i takes F/d_i time to download**

Server

u_s, u_1, d_1, u_2, d_2
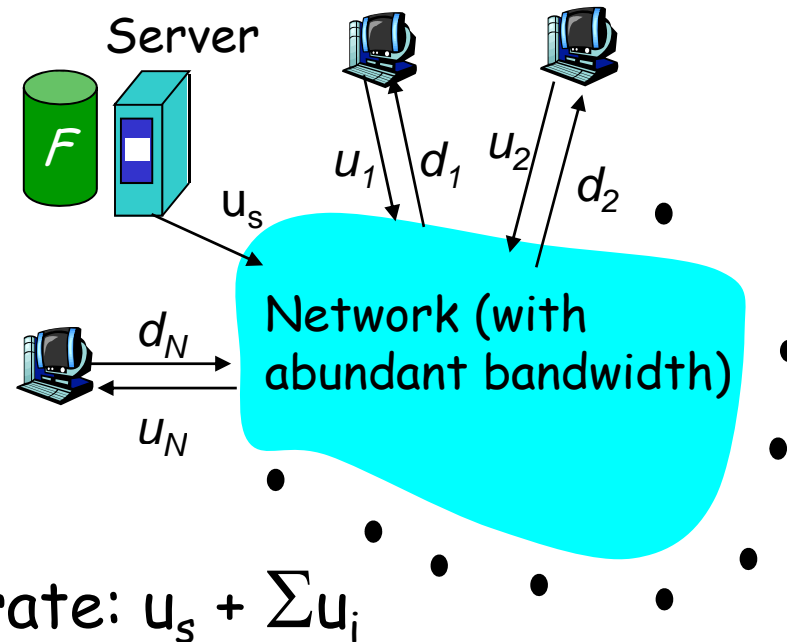
d_N

u_N

Network (with abundant bandwidth)

Time to distribute F to N clients using client/server approach $= d_{cs} = \max \{ NF/u_s, F/\min_i(d_i) \}$

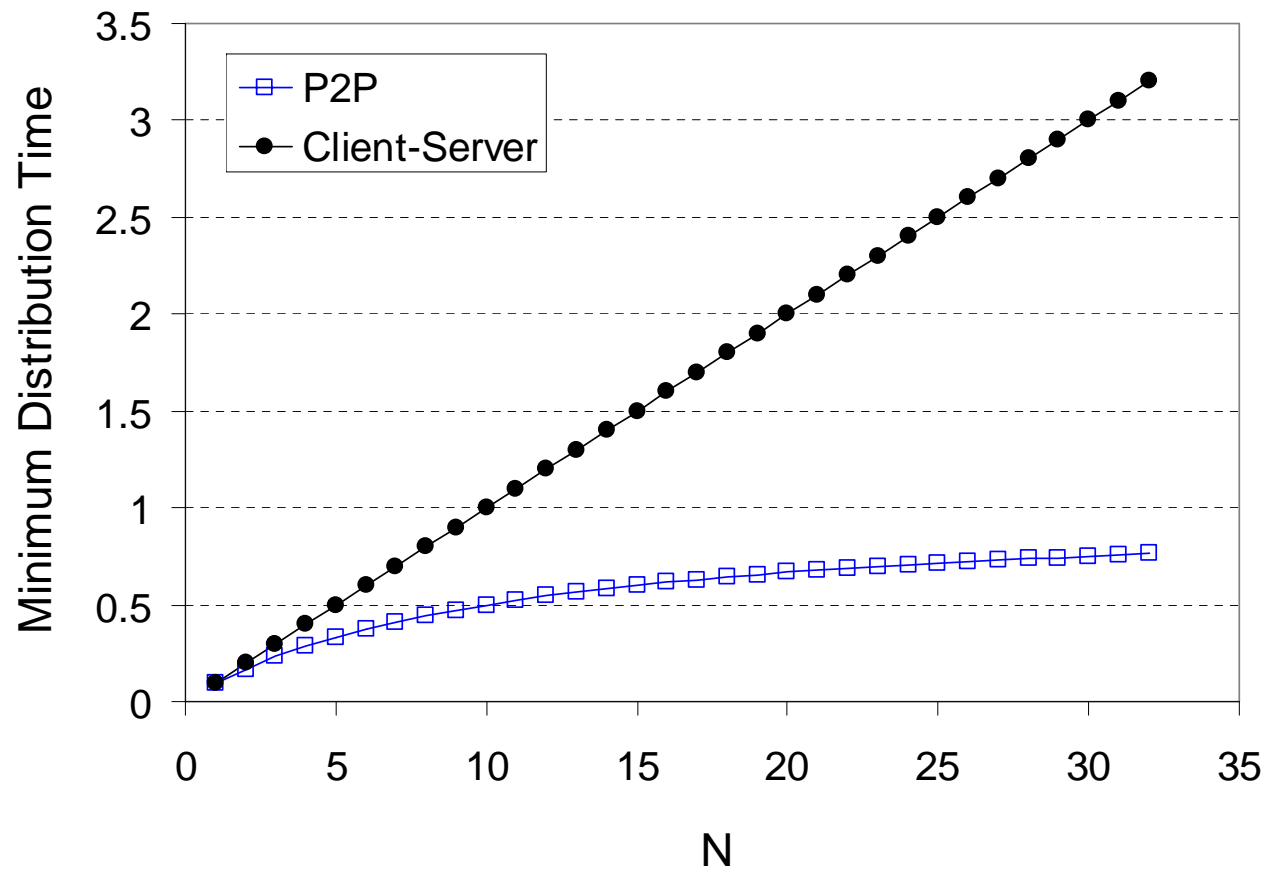increases linearly in N (for large N)

File distribution time: P2P

□ server must send one copy: F/u_s time

□ client i takes F/d_i time to download

□ NF bits must be downloaded (aggregate)

   □ fastest possible upload rate: $u_s + \Sigma u_i$

Server

u_1 d_1 u_2 d_2

u_s

d_N

u_N

Network (with abundant bandwidth)

$$d_{P2P} = \max \left\{ F/u_s, F/min(d_i)_i, NF/(u_s + \Sigma u_i) \right\}$$

Server-client vs. P2P: example

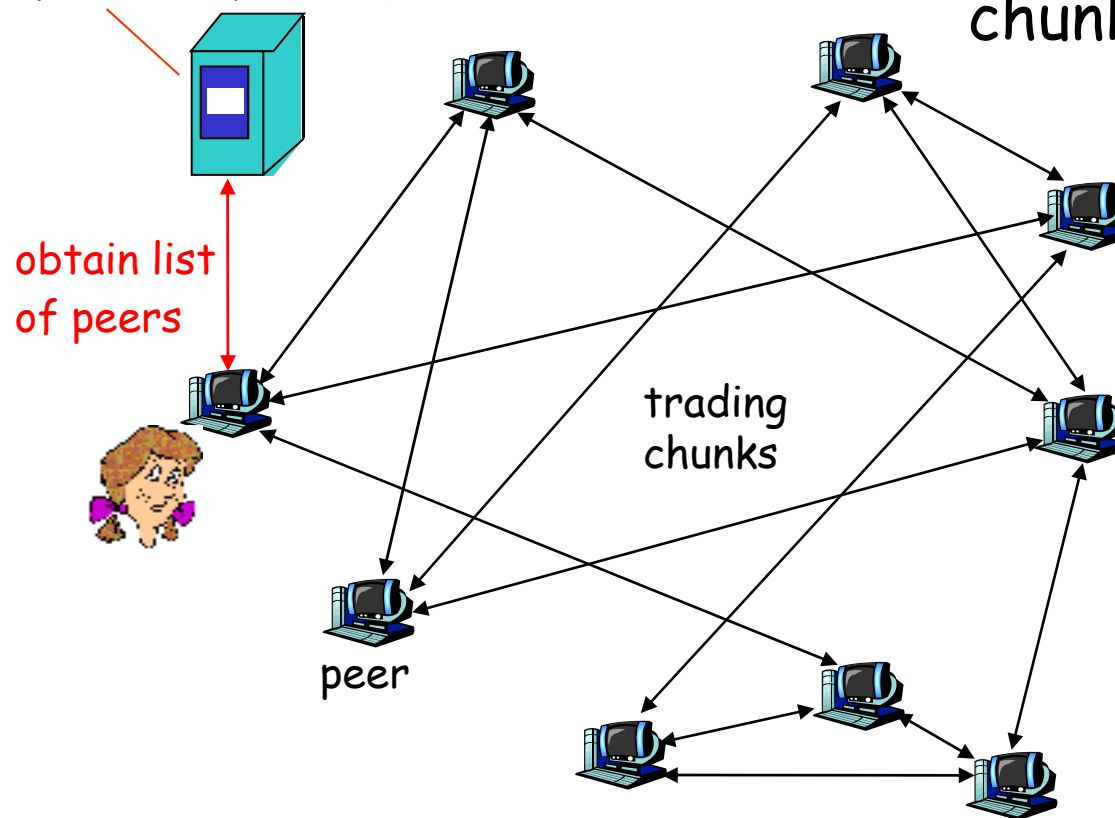Client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$
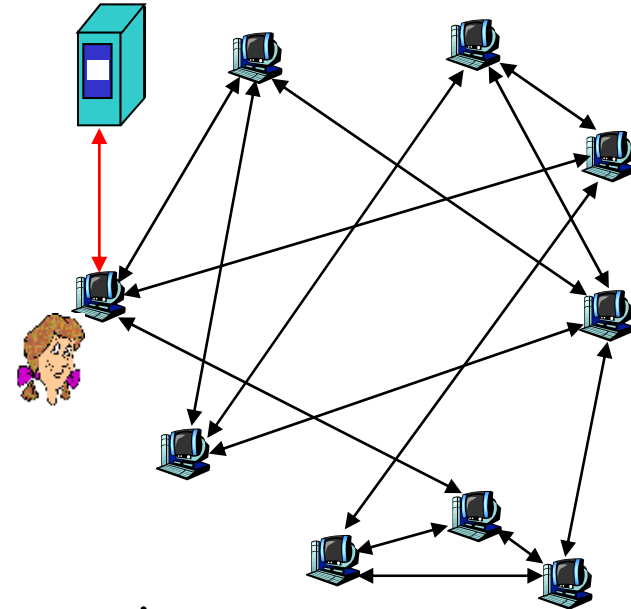
File distribution: BitTorrent

□ P2P file distribution

tracker: tracks peers participating in torrent

torrent: group of peers exchanging chunks of a file

obtain list of peers

trading chunks

peer

BitTorrent (1)

- file divided into 256KB *chunks*.
- peer joining torrent:
  - has no chunks, but will accumulate them over time
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers.
- peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain
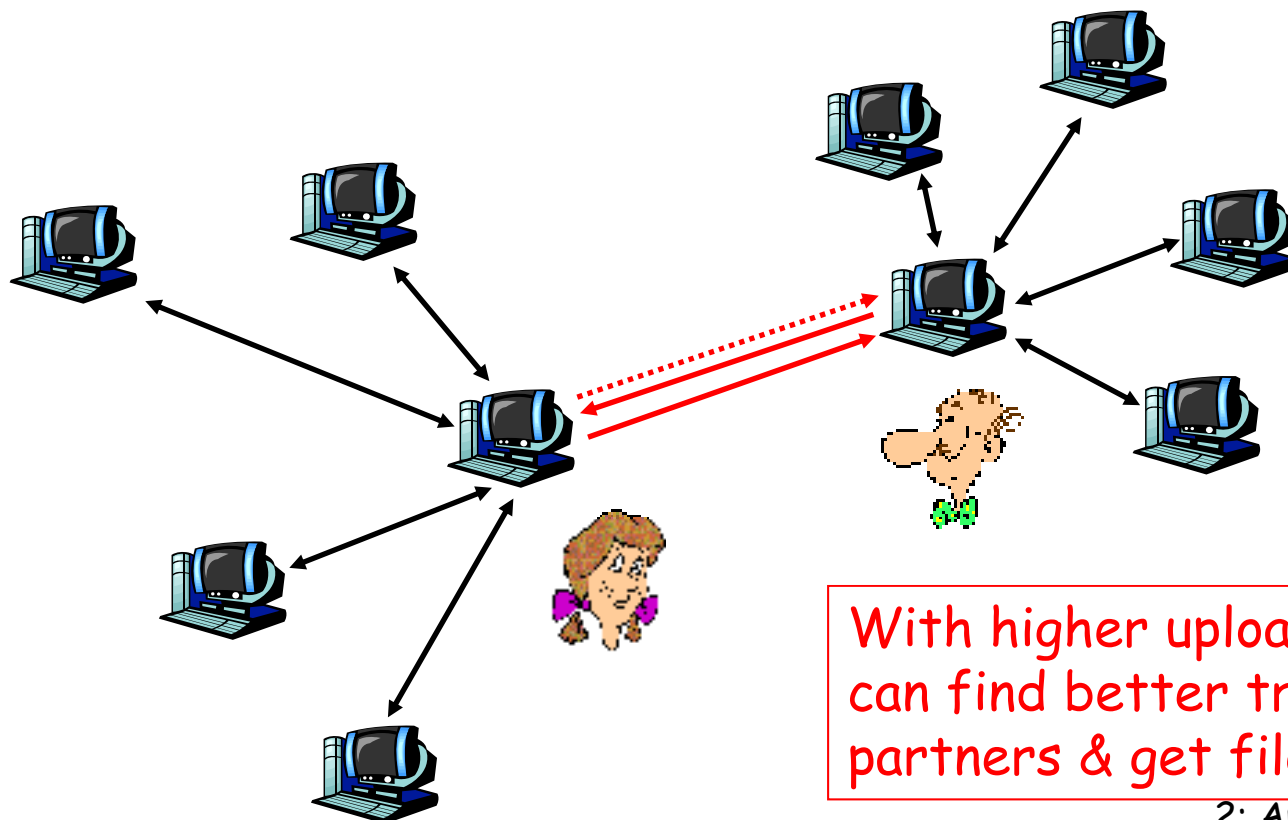
BitTorrent (2)

Pulling Chunks

- at any given time, different peers have different subsets of file chunks

- periodically, a peer (Alice) asks each neighbor for list of chunks that they have.

- Alice sends requests for her missing chunks
  - ❖ rarest first

Sending Chunks: tit-for-tat

- Alice sends chunks to four neighbors currently sending her chunks *at the highest rate*
  - ❖ re-evaluate top 4 every 10 secs

- every 30 secs: randomly select another peer, starts sending chunks
  - ❖ newly chosen peer may join top 4
  - ❖ "optimistically unchoke"

BitTorrent: Tit-for-tat

(1) Alice "optimistically unchokes" Bob
(2) Alice becomes one of Bob's top-four providers; Bob reciprocates
(3) Bob becomes one of Alice's top-four providers

With higher upload rate, can find better trading partners & get file faster!

Distributed Hash Table (DHT)

- DHT = distributed P2P database
- Database has (key, value) pairs;
    - key: ss number; value: human name
    - key: content type; value: IP address
- Peers query DB with key
    - DB returns values that match the key
- Peers can also insert (key, value) peers

DHT Identifiers

- Assign integer identifier to each peer in range $[0, 2^n-1]$.
  - Each identifier can be represented by n bits.
- Require each key to be an integer in same range.
- To get integer keys, hash original key.
  - eg, key = h("Led Zeppelin IV")
  - This is why they call it a distributed "hash" table

How to assign keys to peers?

- Central issue:
  - Assigning (key, value) pairs to peers.
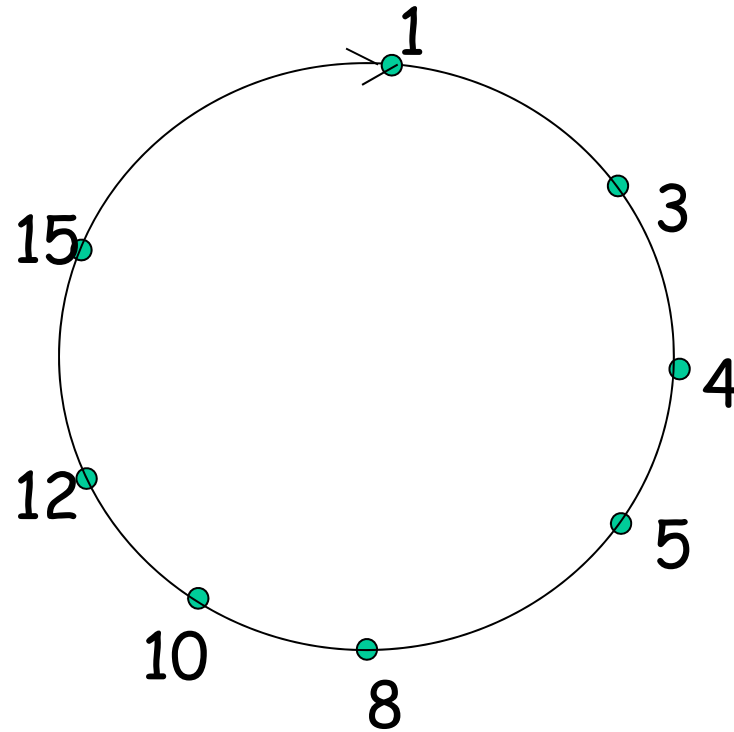- Rule: assign key to the peer that has the closest ID.
- Convention in lecture: closest is the immediate successor of the key.
- Ex: n=4; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

Circular DHT (1)

Circle with peers labeled: 1, 3, 4, 5, 8, 10, 12, 15

□ Each peer *only* aware of immediate successor and predecessor.

□ "Overlay network"

Circle DHT (2)

O(N) messages on avg to resolve query, when there are N peers

I am

Who's resp for key 1110 ?

Define <u>closest</u> as closest successor

0001

0011

1111

1110

1110

0100

1110

1110

1100

0101

1110

1110

1010

1000

Circular DHT with Shortcuts

- Each peer keeps track of IP addresses of predecessor, successor, short cuts.
- Reduced from 6 to 2 messages.
- Possible to design shortcuts so O(log N) neighbors, O(log N) messages in query

Peer Churn

•To handle peer churn, require each peer to know the IP address of its two successors.

• Each peer periodically pings its two successors to see if they are still alive.

□ Peer 5 abruptly leaves

□ Peer 4 detects; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8's immediate successor its second successor.

□ What if peer 13 wants to join?

P2P Case study: Skype

□ inherently P2P: pairs of users communicate.

□ proprietary application-layer protocol (inferred via reverse engineering)

□ hierarchical overlay with SNs

□ Index maps usernames to IP addresses; distributed over SNs

Skype clients (SC)

Skype login server

Supernode (SN)

Peers as relays

□ **Problem when both Alice and Bob are behind "NATs".**

- ❖ NAT prevents an outside peer from initiating a call to insider peer

□ **Solution:**

- ❖ Using Alice's and Bob's SNs, Relay is chosen
- ❖ Each peer initiates session with relay.
- ❖ Peers can now communicate through NATs via relay

Chapter 2: Application layer

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

socket

a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of **bytes** from one process to another

controlled by
application
developer

controlled by
operating
system

process

socket

TCP with
buffers,
variables

host or
server

internet

process

socket

TCP with
buffers,
variables

host or
server

controlled by
application
developer

controlled by
operating
system

Socket programming *with TCP*

Client must contact server

□ server process must first be running

□ server must have created socket (door) that welcomes client's contact

Client contacts server by:

□ creating client-local TCP socket

□ specifying IP address, port number of server process

□ When client creates socket: client TCP establishes connection to server TCP

□ When contacted by client, server TCP creates new socket for server process to communicate with client

   ❖ allows server to talk with multiple clients

   ❖ source port numbers used to distinguish clients (more in Chap 3)

┌─ application viewpoint ─────────────

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Client/server socket interaction: TCP

Server (running on `hostid`)   **Client**

create socket,
port=**x**, for
incoming request:
welcomeSocket =
   ServerSocket()

wait for incoming   ← — — *TCP* — — →   create socket,
connection request   *connection setup*   connect to `hostid`, port=**x**
connectionSocket =   clientSocket =
welcomeSocket.accept()   Socket()

                               send request using
read request from   clientSocket
connectionSocket

write reply to
connectionSocket   read reply from
                                        clientSocket

close   close
connectionSocket   clientSocket

Stream jargon

□ A stream is a sequence of characters that flow into or out of a process.

□ An input stream is attached to some input source for the process, e.g., keyboard or socket.
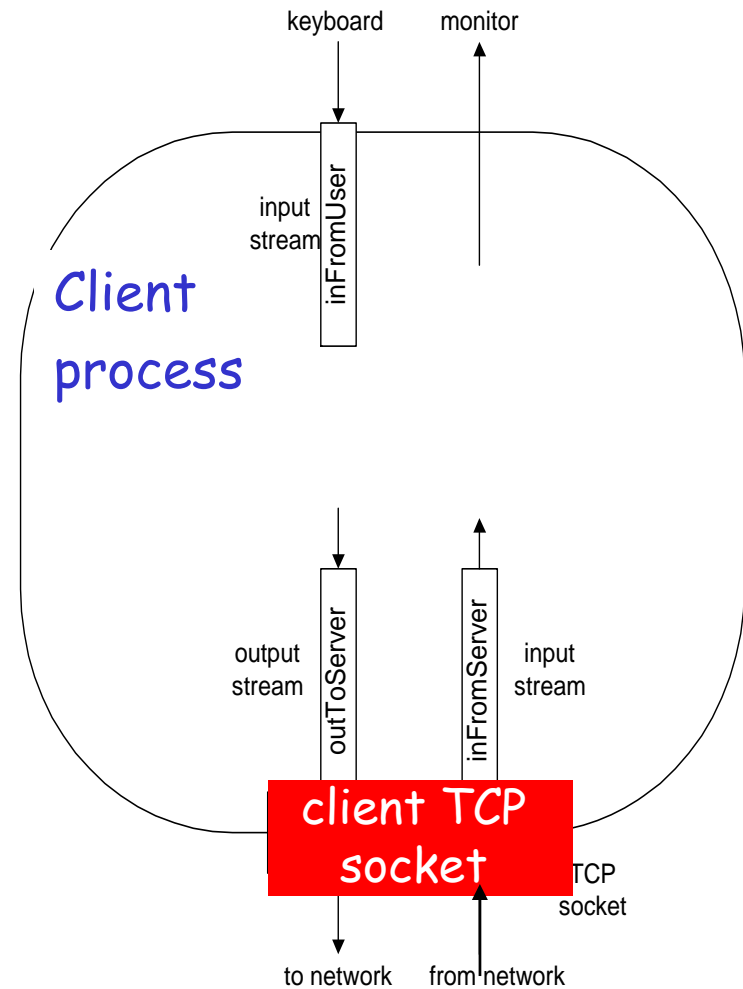
□ An output stream is attached to an output source, e.g., monitor or socket.

keyboard   monitor

Client
process

input
stream   inFromUser

output   outToServer   inFromServer   input
stream   stream

client TCP socket   TCP
                         socket

to network   from network

Socket programming with TCP

Example client-server app:

1) client reads line from standard input (`inFromUser` stream) , sends to server via socket (`outToServer` stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (`inFromServer` stream)

Example: Java client (TCP)

```java
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream →

Create client socket, connect to server →

Create output stream attached to socket →

Example: Java client (TCP), cont.

Create
input stream
attached to socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));
```

```
sentence = inFromUser.readLine();
```

Send line
to server

```
outToServer.writeBytes(sentence + '\n');
```

Read line
from server

```
modifiedSentence - inFromServer.readLine();
```

```
System.out.println("FROM SERVER: " + modifiedSentence);
```

```
clientSocket.close();
```

```
      }
    }
```

Example: Java server (TCP)

```java
import java.io.*;
import java.net.*;

class TCPServer {

    public static void main(String argv[]) throws Exception
    {
        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {

            Socket connectionSocket = welcomeSocket.accept();

            BufferedReader inFromClient =
                new BufferedReader(new
                InputStreamReader(connectionSocket.getInputStream()));
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

Create output stream, attached to socket →
```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

Read in line from socket →
```
clientSentence = inFromClient.readLine();
```

```
capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Write out line to socket →
```
outToClient.writeBytes(capitalizedSentence);
      }
    }
  }
```

End of while loop, loop back and wait for another client connection

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS

- 2.6 P2P applications
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP

Socket programming *with UDP*

UDP: no "connection" between client and server

- □ no handshaking
- □ sender explicitly attaches IP address and port of destination to each packet
- □ server must extract IP address, port of sender from received packet

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

┌─ application viewpoint ──────────

UDP provides <u>unreliable</u> transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP

Server (running on `hostid`)   Client

create socket,
port= x.
serverSocket =
DatagramSocket()

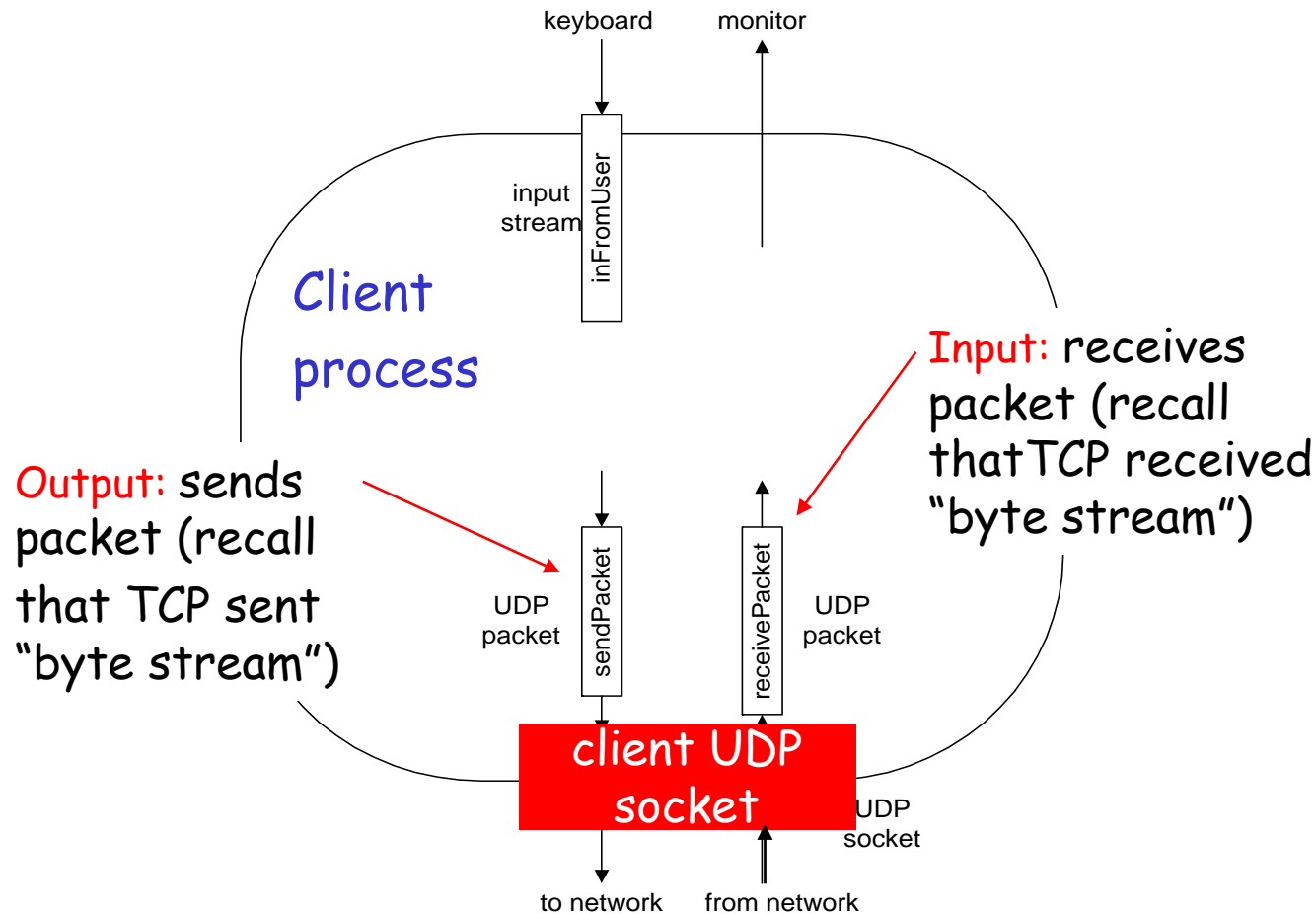   create socket,
                       clientSocket =
                       DatagramSocket()

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

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

                       read datagram from
                       clientSocket

                       close
                       clientSocket

Example: Java client (UDP)

Example: Java client (UDP)

```java
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        DatagramSocket clientSocket = new DatagramSocket();

        InetAddress IPAddress = InetAddress.getByName("hostname");

        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];

        String sentence = inFromUser.readLine();

        sendData = sentence.getBytes();
```

Create input stream

Create client socket

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```
clientSocket.send(sendPacket);

DatagramPacket receivePacket =
  new DatagramPacket(receiveData, receiveData.length);
```

Read datagram from server

```
clientSocket.receive(receivePacket);

String modifiedSentence =
  new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
  }
}
```

Example: Java server (UDP)

```java
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
   {

    DatagramSocket serverSocket = new DatagramSocket(9876);

    byte[] receiveData = new byte[1024];
    byte[] sendData  = new byte[1024];

    while(true)
     {

      DatagramPacket receivePacket =
         new DatagramPacket(receiveData, receiveData.length);

      serverSocket.receive(receivePacket);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

Example: Java server (UDP), cont

String sentence = new String(receivePacket.getData());

Get IP addr
port #, of
sender
InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

Create datagram
to send to client
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress,
                port);

Write out
datagram
to socket
serverSocket.send(sendPacket);
    }
  }
}

End of while loop,
loop back and wait for
another datagram

Chapter 2: Summary

our study of network apps now complete!

- □ application architectures
  - ❖ client-server
  - ❖ P2P
  - ❖ hybrid
- □ application service requirements:
  - ❖ reliability, bandwidth, delay
- □ Internet transport service model
  - ❖ connection-oriented, reliable: TCP
  - ❖ unreliable, datagrams: UDP

- □ specific protocols:
  - ❖ HTTP
  - ❖ FTP
  - ❖ SMTP, POP, IMAP
  - ❖ DNS
  - ❖ P2P: BitTorrent, Skype
- □ socket programming

Chapter 2: Summary

Most importantly: learned about *protocols*

□ **typical request/reply message exchange:**
  - ❖ client requests info or service
  - ❖ server responds with data, status code

□ **message formats:**
  - ❖ headers: fields giving info about data
  - ❖ data: info being communicated

Important themes:

□ control vs. data msgs
  - ❖ in-band, out-of-band

□ centralized vs. decentralized

□ stateless vs. stateful

□ reliable vs. unreliable msg transfer

□ "complexity at network edge"