软件学院

COMPUTER NETWORK

CHAPTER 2

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Chapter 2: application layer

our goals:

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

Chapter 2.1: outline

2.1 principles of network applications

- 2.1.1 Network Application Architecture
- 2.1.2 Processes Communication
- 2.1.3 Transport Services Available to Applications
- 2.1.4 Transport Services Provided by the Internet
- 2.1.5 Application-Layer Protocols
- 2.1.6 Network Applications Covered in This Book

Some network apps

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

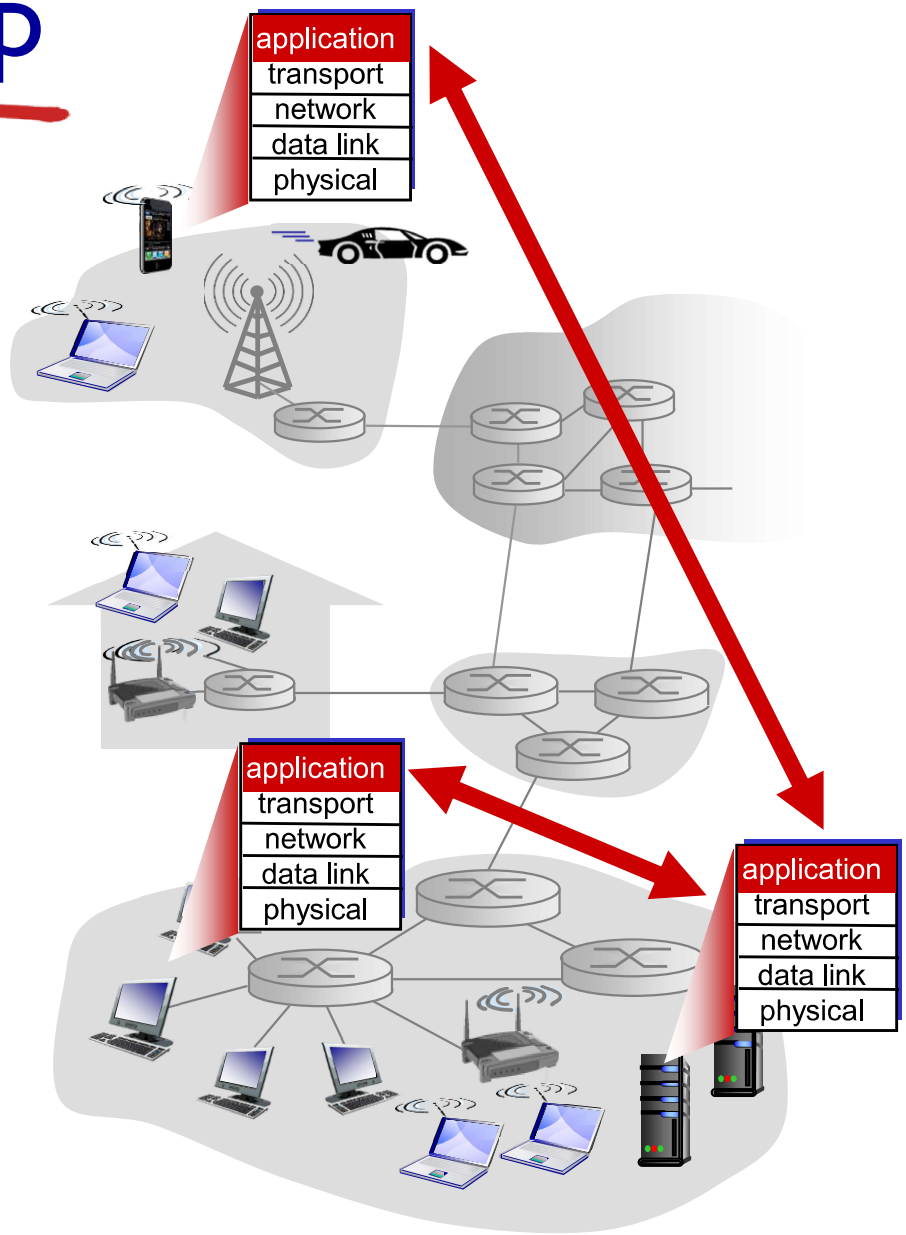
Creating a network app

write programs that:

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

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

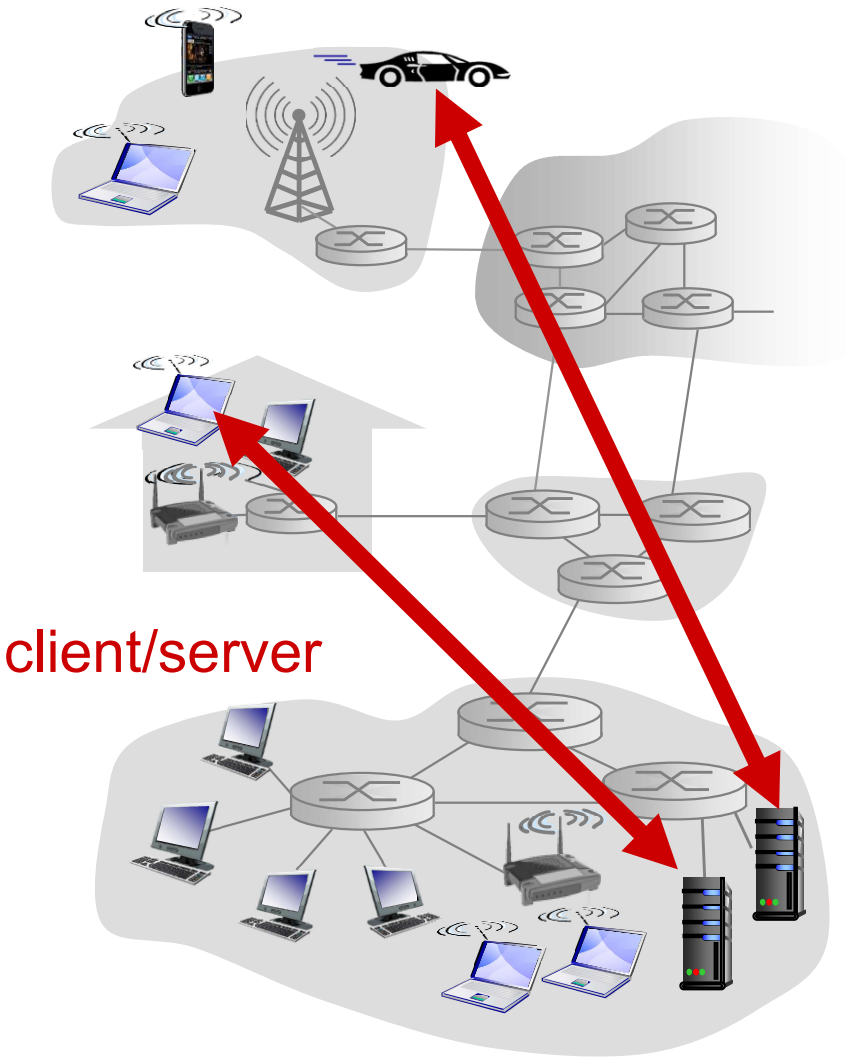


Application architectures

possible structure of applications:

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

Client-server architecture



server:

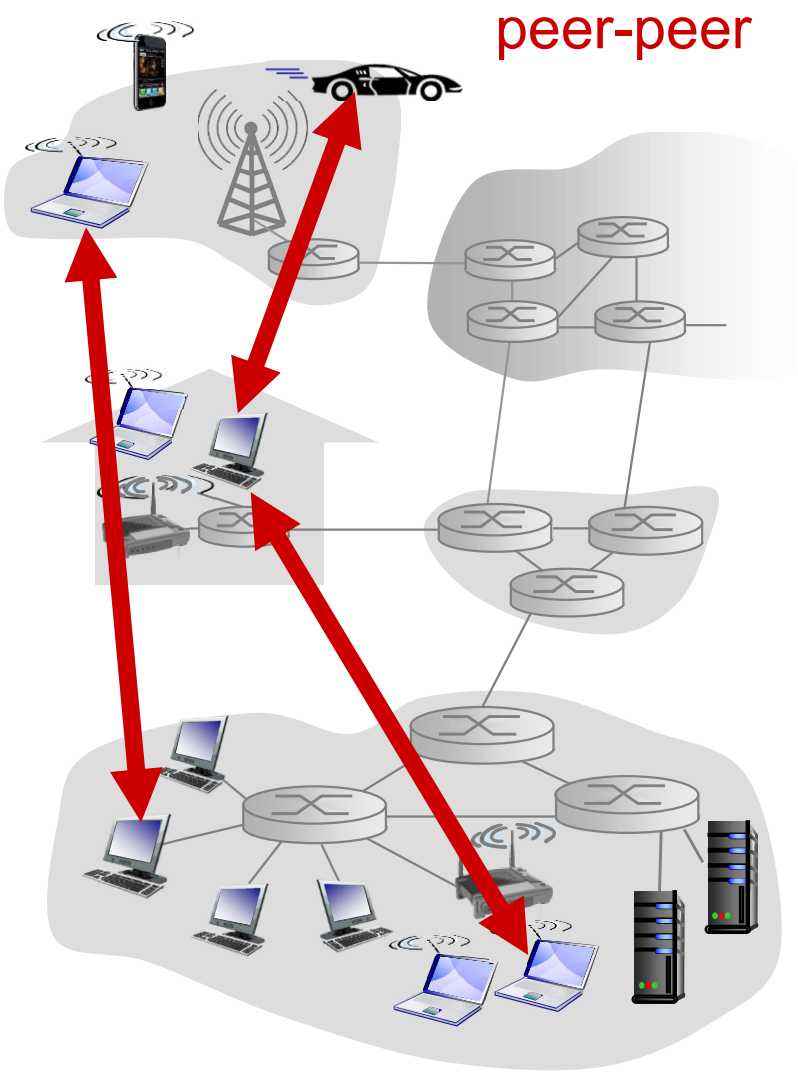
- always-on host
- permanent IP address
- Web\FTP\Telnet\email
- data centers for scaling

clients:

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

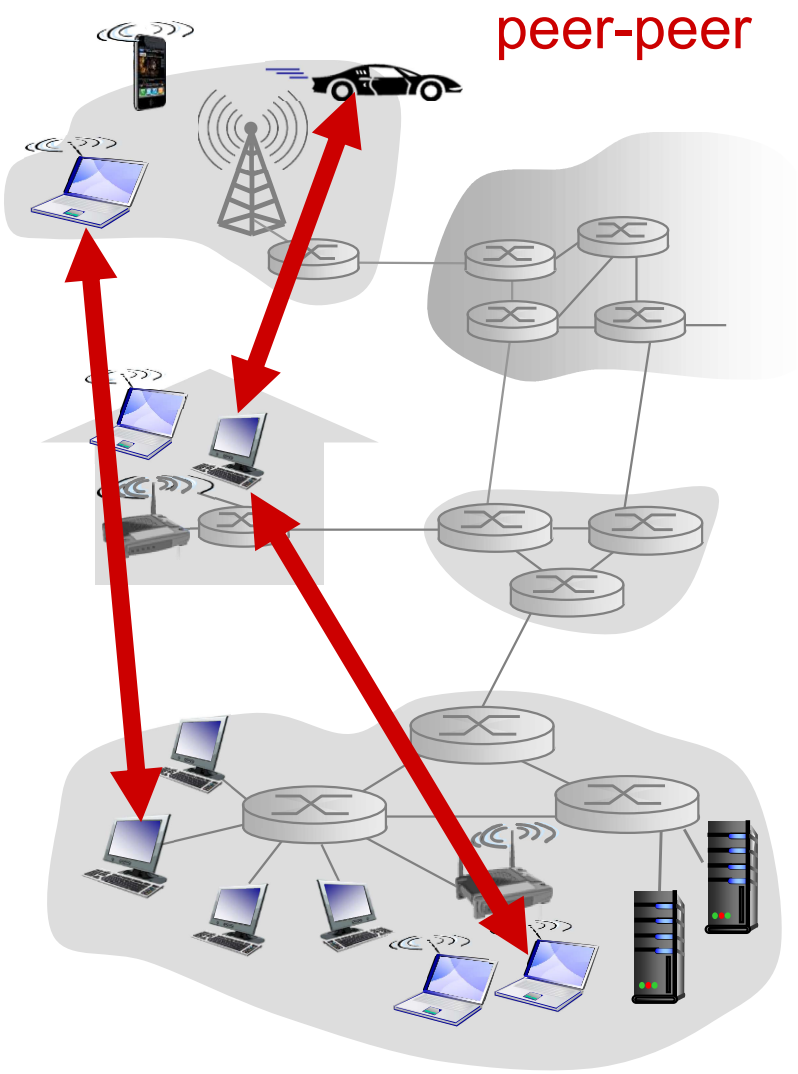
P2P architecture

- *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
  - BitTorrent, eMule, Skpye, PPTV, Thunder



P2P architecture

- cost effective
  - normally don't require significant server infrastructure and server bandwidth
- three major challenges:
  - ISP Friendly
  - Security
  - Incentives



hybrid of client-server and P2P architecture

- Instant messaging applications
  - 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
  - chatting between two users is P2P
- Skype
  - voice-over-IP (VoIP) P2P application
  - centralized server: finding address of remote party:
  - client-client connection: direct (not through server)

Chapter 2.1: outline

2.1 principles of network applications

- 2.1.1 Network Application Architecture
- 2.1.2 Processes Communication
- 2.1.3 Transport Services Available to Applications
- 2.1.4 Transport Services Provided by the Internet
- 2.1.5 Application-Layer Protocols
- 2.1.6 Network Applications Covered in This Book

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
  - sending process: creates and sends messages into the
  - network
  - receiving process: receives these messages and possibly responds by sending messages back

Processes communicating

clients, servers

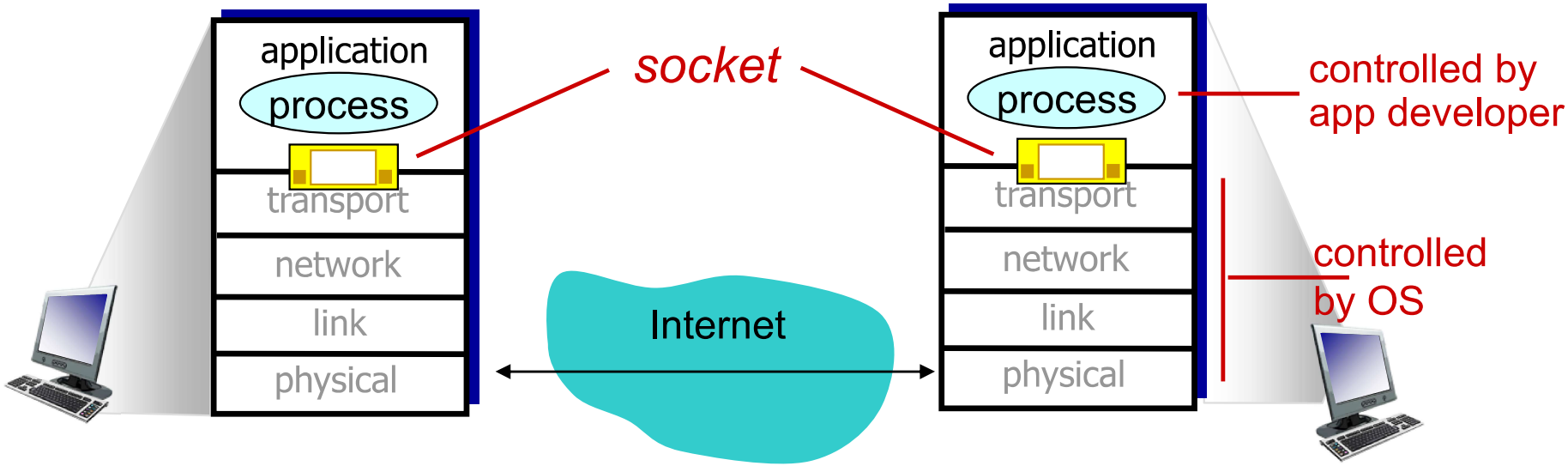
client process: process that initiates communication

server process: process that waits to be contacted

- *in the Web application a client browser process exchanges messages with a Web server process*
- *aside: applications with P2P architectures have client processes & server processes*
  - *Peer A asks Peer B to send a file, Peer A is the client and Peer B is the server in the context of this specific communication session.*

Sockets

- process sends/receives messages to/from its socket
- process is analogous to a house, 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
  - little control of the transport-layer side of the socket



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…

Chapter 2.1: outline

2.1 principles of network applications

- 2.1.1 Network Application Architecture
- 2.1.2 Processes Communication
- 2.1.3 Transport Services Available to Applications
- 2.1.4 Transport Services Provided by the Internet
- 2.1.5 Application-Layer Protocols
- 2.1.6 Network Applications Covered in This Book

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 (loss-tolerant application)

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": bandwidth-sensitive applications
- other apps ("elastic apps") make use of whatever throughput they get

security

- encryption, data integrity, end-point authentication

…

Chapter 2.1: outline

2.1 principles of network applications

- 2.1.1 Network Application Architecture
- 2.1.2 Processes Communication
- 2.1.3 Transport Services Available to Applications
- 2.1.4 Transport Services Provided by the Internet
- 2.1.5 Application-Layer Protocols
- 2.1.6 Network Applications Covered in This Book

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:

- *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 guarantee, security

UDP service:

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

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

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

Securing TCP

TCP & UDP

- no encryption
- cleartext passwds sent into socket traverse Internet in cleartext

SSL

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

SSL is at app layer

- Apps use SSL libraries, which "talk" to TCP

SSL socket API

- cleartext passwds sent into socket traverse Internet encrypted
- See Chapter 7

Chapter 2.1: outline

2.1 principles of network applications

- 2.1.1 Network Application Architecture
- 2.1.2 Processes Communication
- 2.1.3 Transport Services Available to Applications
- 2.1.4 Transport Services Provided by the Internet
- 2.1.5 Application-Layer Protocols
- 2.1.6 Network Applications Covered in This Book

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax（语法）:
  - what fields in messages & how fields are delineated
- message semantics（语义）
  - 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

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Chapter 2.2: outline

2.2 The Web and HTTP

- 2.2.1 Overview of HTTP
- 2.2.2 Non-Persistent and Persistent Connections
- 2.2.3 HTTP Message Format
- 2.2.4 User-Server Interaction: Cookies
- 2.2.5 Web Caching
- 2.2.6 The Conditional GET

Web and HTTP

- before 1990s, Internet unknown outside of the academic and research communities
- early 1990s, major new application arrived on the scene—the World Wide Web
  - The Web was the first Internet application that caught the general public's eye
  - The Web operates on demand
  - Graphics、interact with pages and sites、many killer applications
  - Hyperlinks and search engines help us navigate through an ocean of Web sites

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

Chapter 2.2: outline

2.2 The Web and HTTP

- 2.2.1 Overview of HTTP
- 2.2.2 Non-Persistent and Persistent Connections
- 2.2.3 HTTP Message Format
- 2.2.4 User-Server Interaction: Cookies
- 2.2.5 Web Caching
- 2.2.6 The Conditional GET

HTTP connections

non-persistent HTTP

- each request/response pair be sent over a separate TCP connection
- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

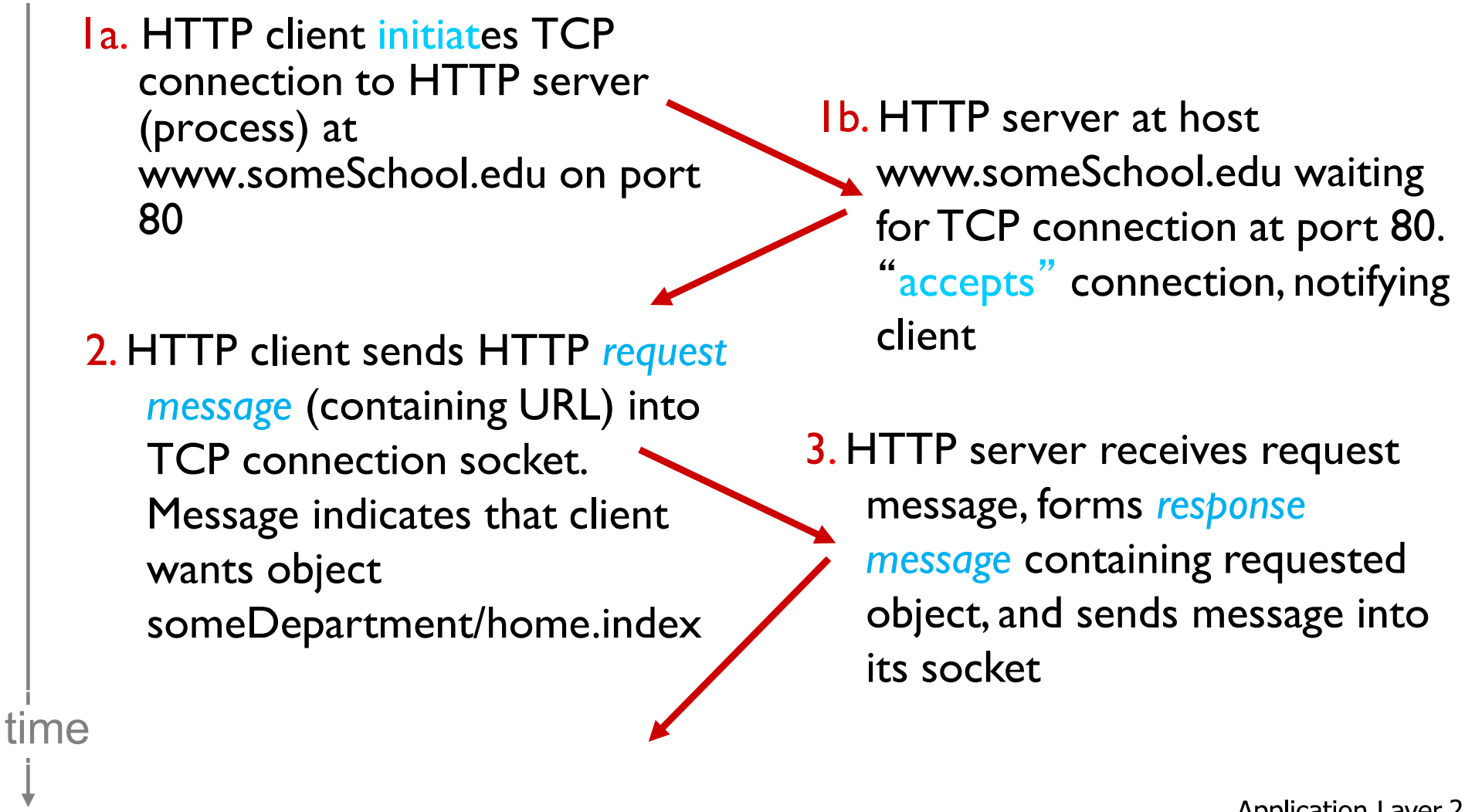
persistent HTTP

- all of the requests and their corresponding responses be sent over the same TCP connection
- multiple objects can be sent over single TCP connection between client, server

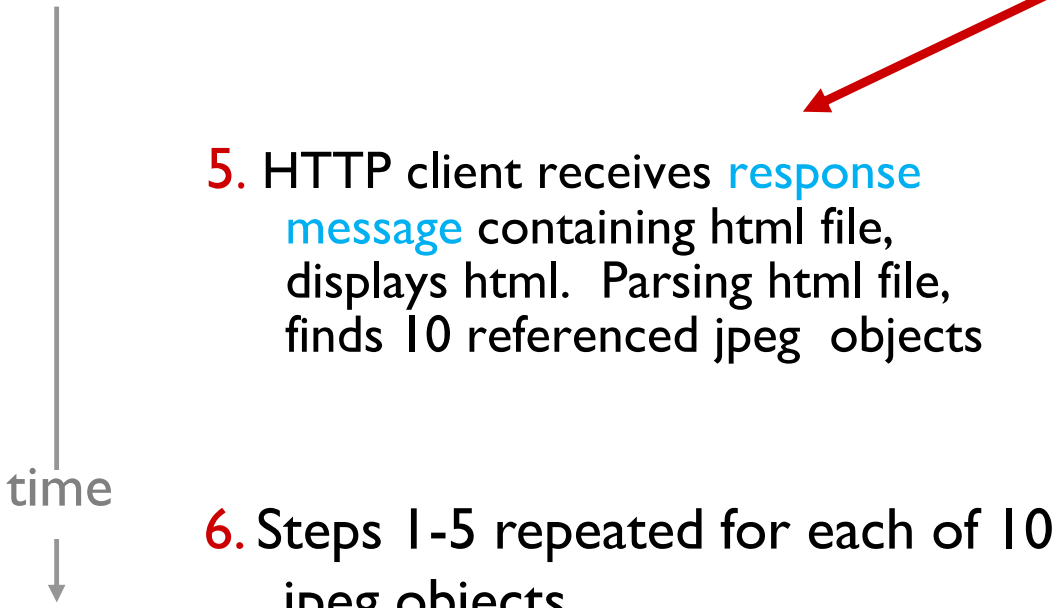
Non-persistent HTTP

suppose user enters URL:
`www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)



Non-persistent HTTP (cont.)



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

- each TCP connection is **closed** after the server sends the object—the connection **does not persist** for other objects
- request **one** web page, **11 TCP connections** are generated

Discussion

- **Question**: A user requests a Web page that consists of some text and 3 images. The browser's cache is empty. For this page, the client's browser:
- *A. sends 1 http request message and receives 1 http response messages.*
- *B. sends 1 http request message and receives 3 http response messages.*
- *C. sends 3 http request message and receives 3 http response messages.*
- *D. sends 4 http request message and receives 4 http response messages.*
- **Answer**: D

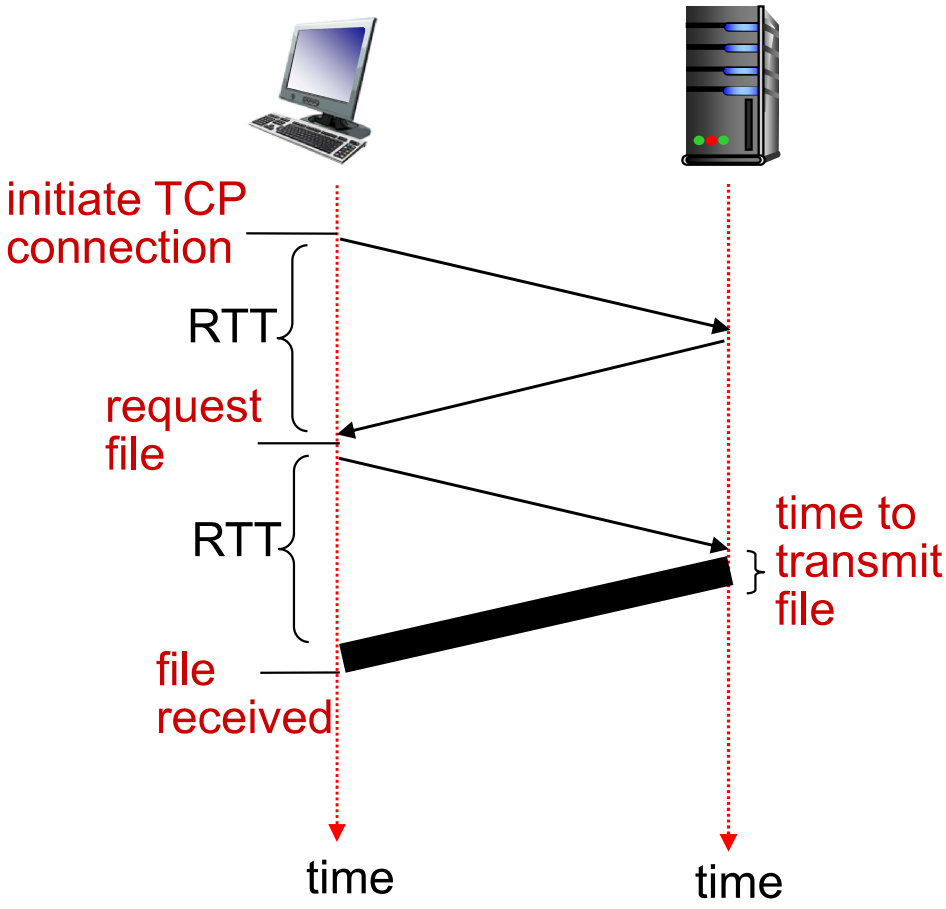
Non-persistent HTTP: response time

estimate the amount of time

that elapses from when a client requests the base HTML file until the entire file is received by the client

RTT (definition):

- time for a small packet to travel from client to server and back
- includes processing delays, queuing delays, and propagation delays



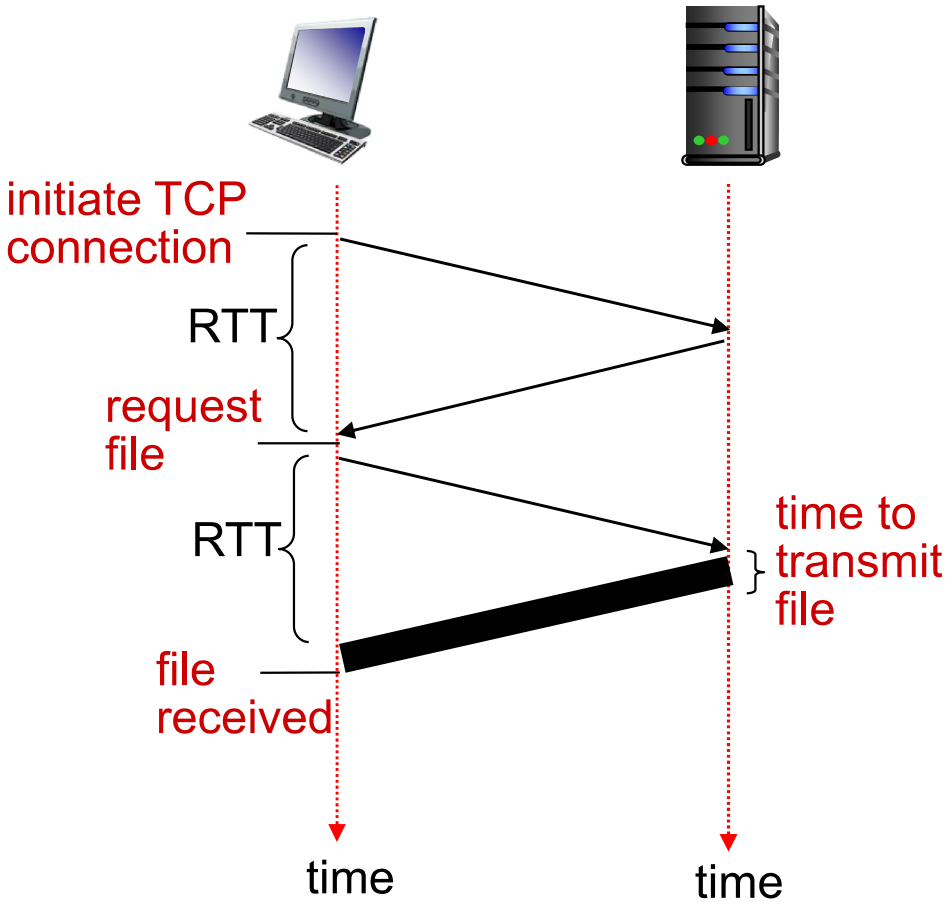
Non-persistent HTTP: response time

Three-way handshake:

- the client sends a small TCP segment to the server,
- the server acknowledges and responds with a small TCP segment,
- and, finally, the client acknowledges back to the server

HTTP response time:

- one RTT to initiate TCP connection
- one RTT to request and receive an object
- file transmission time
- non-persistent HTTP response time = 2RTT+ file transmission time



Persistent HTTP

non-persistent HTTP issues:

- a brand-new connection must be established and maintained for each requested object, OS overhead for each TCP connection
- TCP buffers must be allocated and TCP variables must be kept in both client and server
- can place a significant burden on the Web server
- each object suffers a delivery delay of more than two RTTs

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

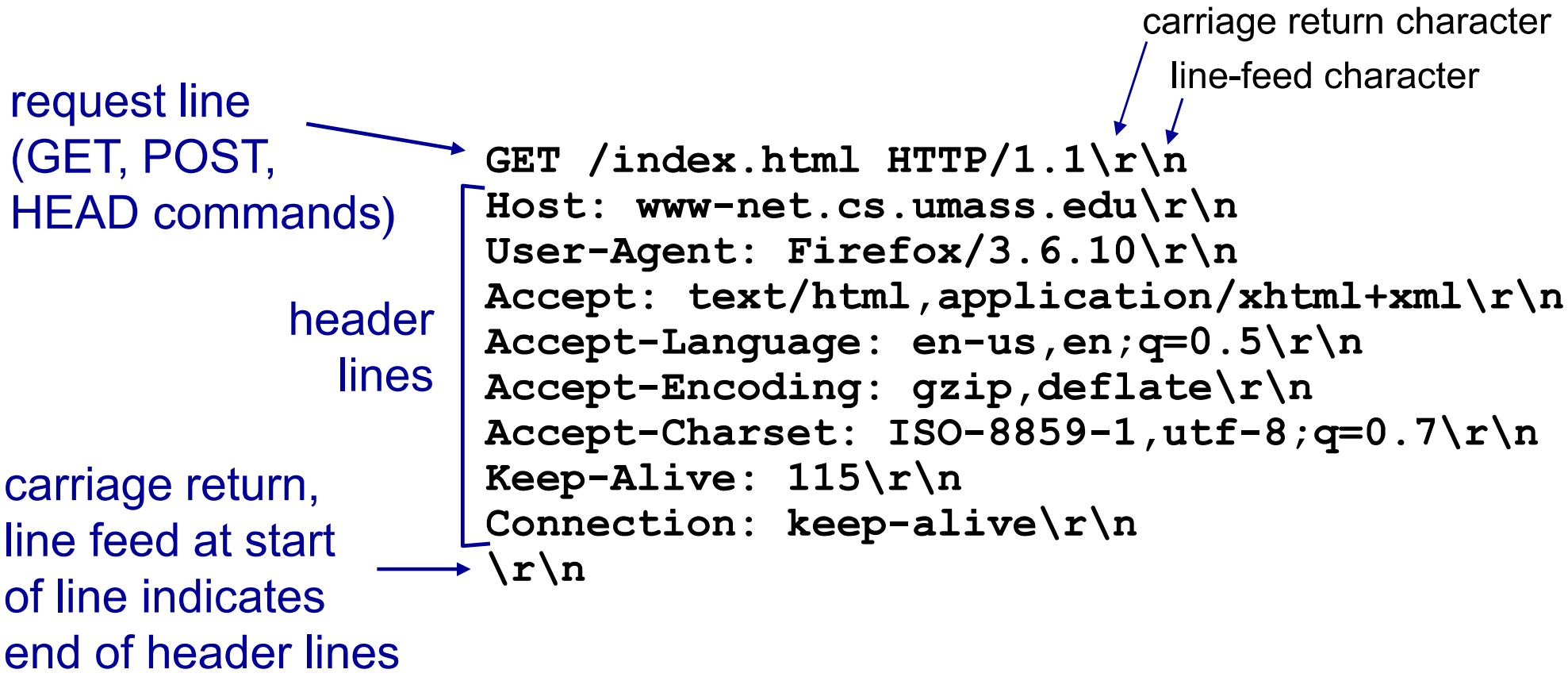
Chapter 2.2: outline

2.2 The Web and HTTP

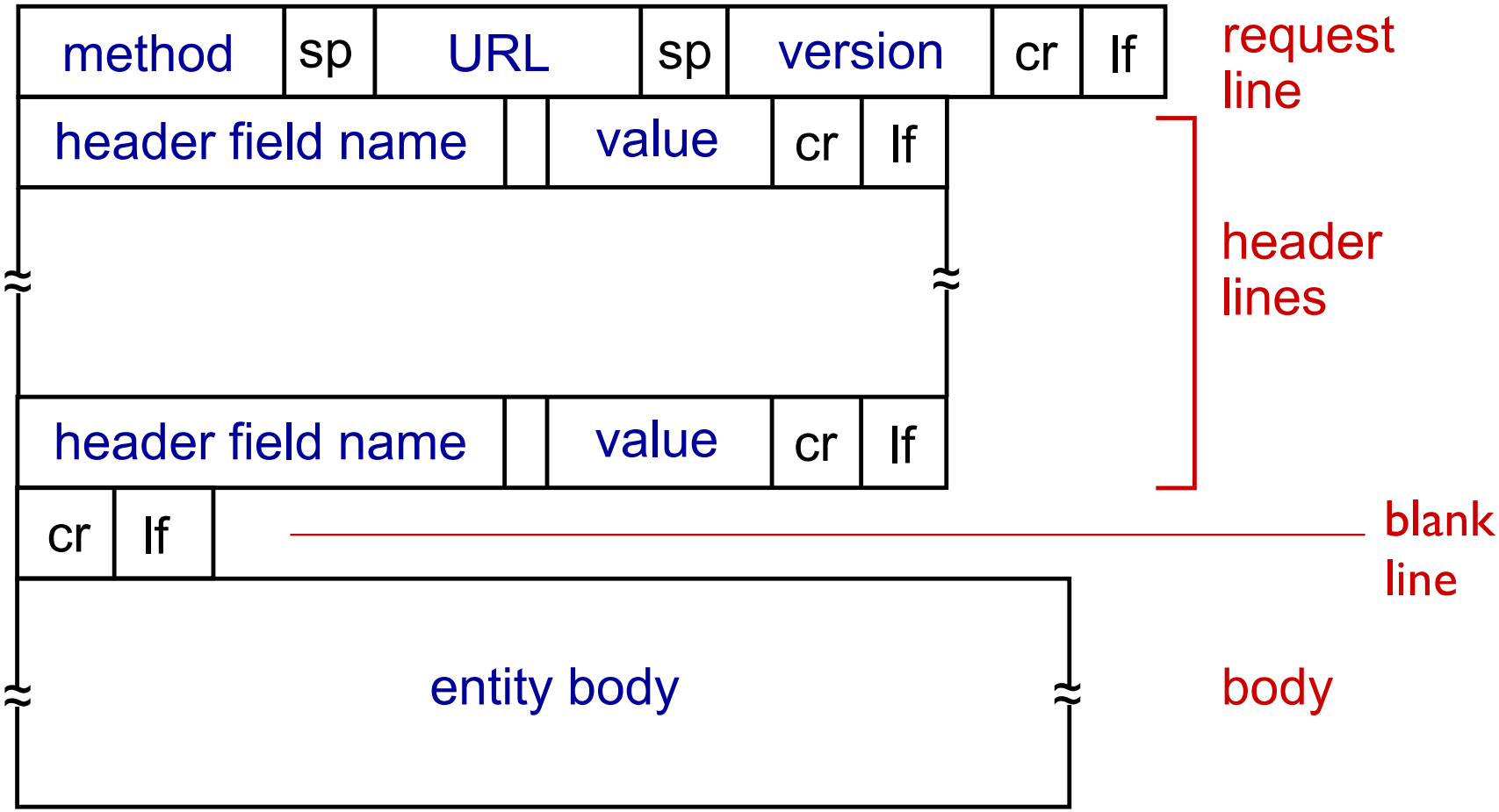
- 2.2.1 Overview of HTTP
- 2.2.2 Non-Persistent and Persistent Connections
- 2.2.3 HTTP Message Format
- 2.2.4 User-Server Interaction: Cookies
- 2.2.5 Web Caching
- 2.2.6 The Conditional GET

HTTP request message

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



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 (for debugging)

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

server retrieves the object from its file system, inserts the object into the response message, and sends the response message

status line (version status code phrase)

protocol version

status code

status message

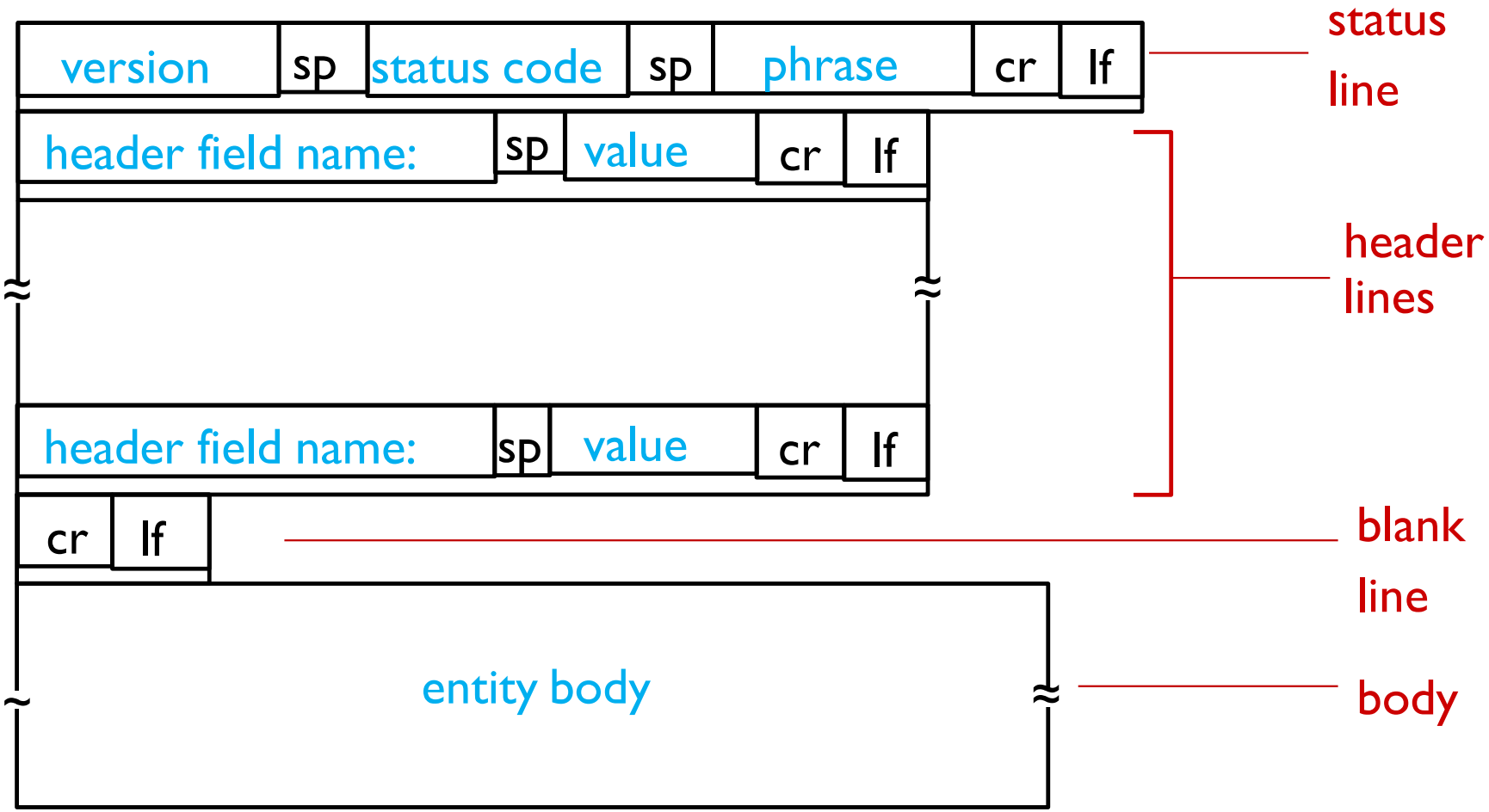
header lines

blank lines

entity body, 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 message : general format



HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK

- request succeeded, requested object later in this msg

301 Moved Permanently

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

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet www.baidu.com 80
```

opens TCP connection to port 80 (default HTTP server port) at www.baidu.com. anything typed in sent to port 80 at www.baidu.com

2. type in a GET HTTP request:

```
GET /index.html/ HTTP/1.1
Host: www.baidu.com
```

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

3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

Chapter 2.2: outline

2.2 The Web and HTTP

- 2.2.1 Overview of HTTP
- 2.2.2 Non-Persistent and Persistent Connections
- 2.2.3 HTTP Message Format
- 2.2.4 User-Server Interaction: Cookies
- 2.2.5 Web Caching
- 2.2.6 The Conditional GET

User-server state: cookies

many Web sites use cookies

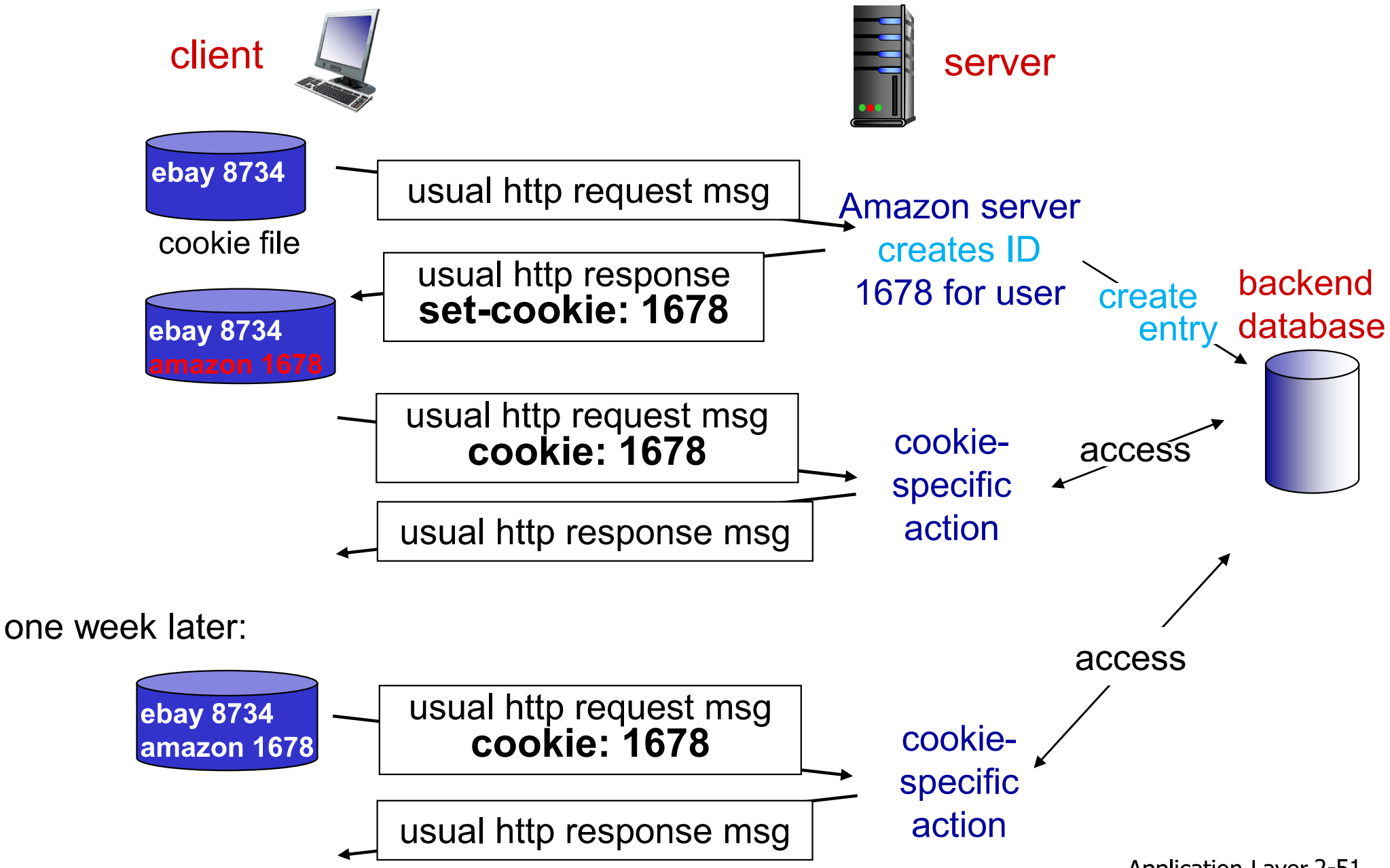
four components:

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

example:

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

Cookies: keeping "state" (cont.)



Cookies (continued)

what cookies can be used for:

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

aside

cookies and privacy:

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

how to keep "state":

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

Chapter 2.2: outline

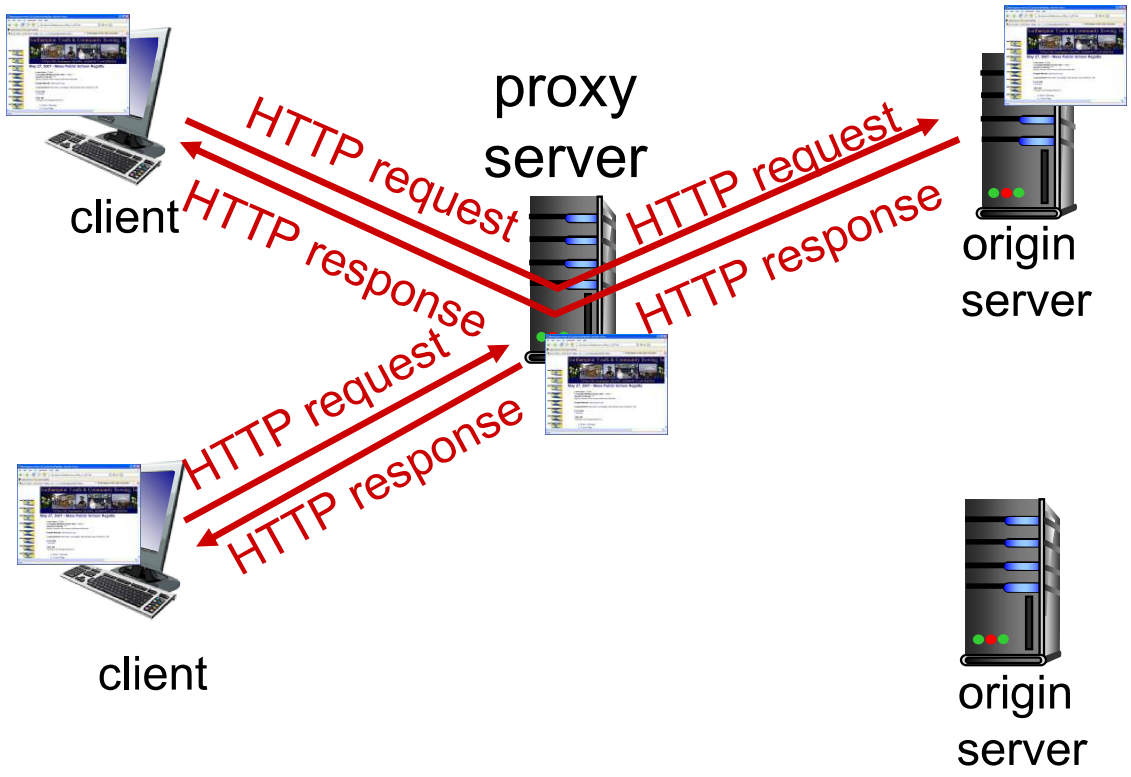
2.2 The Web and HTTP

- 2.2.1 Overview of HTTP
- 2.2.2 Non-Persistent and Persistent Connections
- 2.2.3 HTTP Message Format
- 2.2.4 User-Server Interaction: Cookies
- 2.2.5 Web Caching
- 2.2.6 The Conditional GET

Web caches (proxy server)

goal: satisfy client request without involving origin server

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



More about Web caching

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

why Web caching?

- reduce response time for client request
- reduce traffic on an institution's access link
- Internet dense with caches: enables "poor" content providers to effectively deliver content (so too does P2P file sharing)

Caching example:

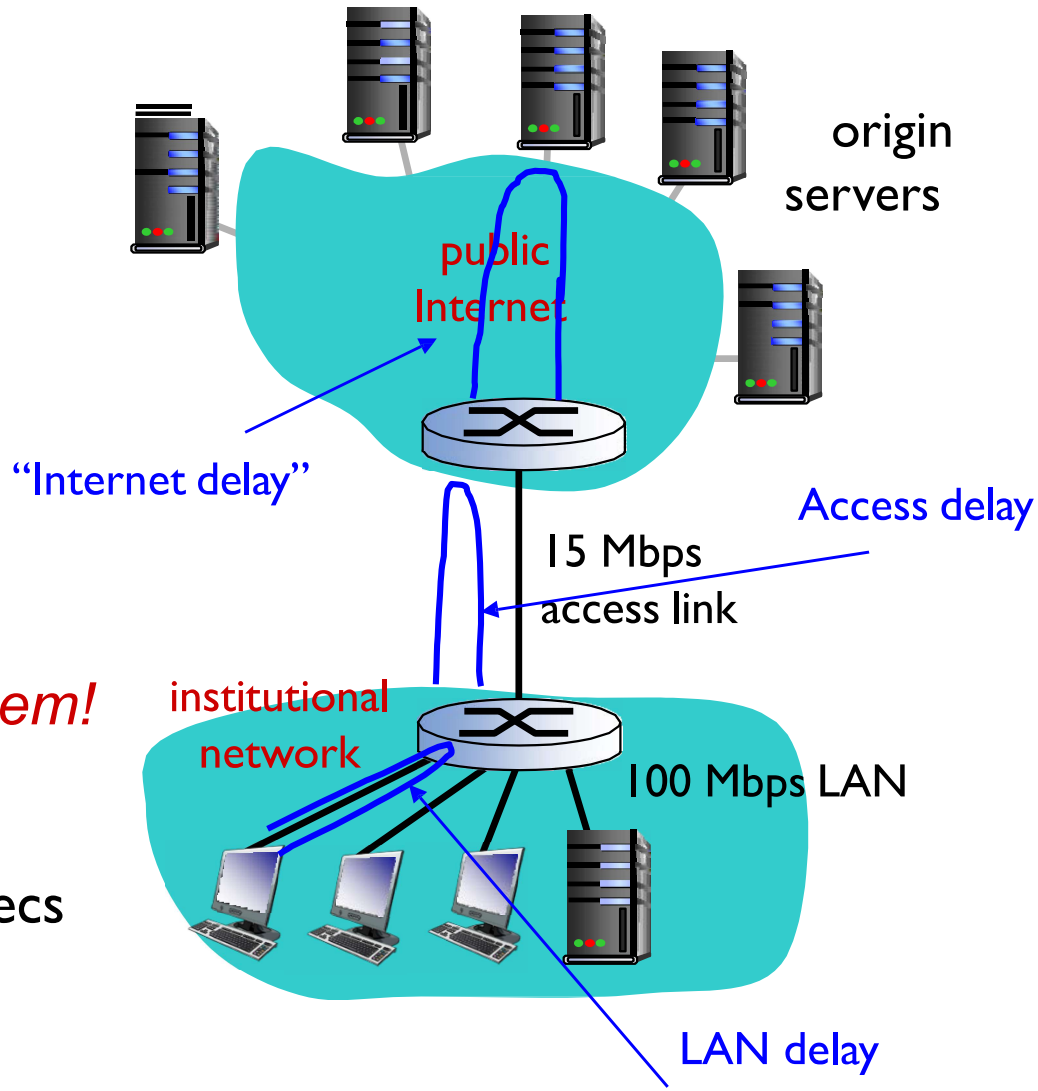
assumptions:

- access link rate: 15 Mbps
- avg request rate from browsers to origin servers: 15/sec
- avg object size: 1Mbits
- RTT from the router on the Internet side to any origin server: 2 sec

consequences:

- total delay = Internet delay + access delay + LAN delay
- LAN traffic intensity = 0.15
- access link traffic intensity = 1
- total delay =2 sec + minutes + msecs

problem!



LAN traffic intensity = (15 requests/sec)*(1 Mbits/request)/(100Mbps)=0.15
access link traffic intensity = (15 requests/sec)*(1Mbits/request/(15Mbps)=1

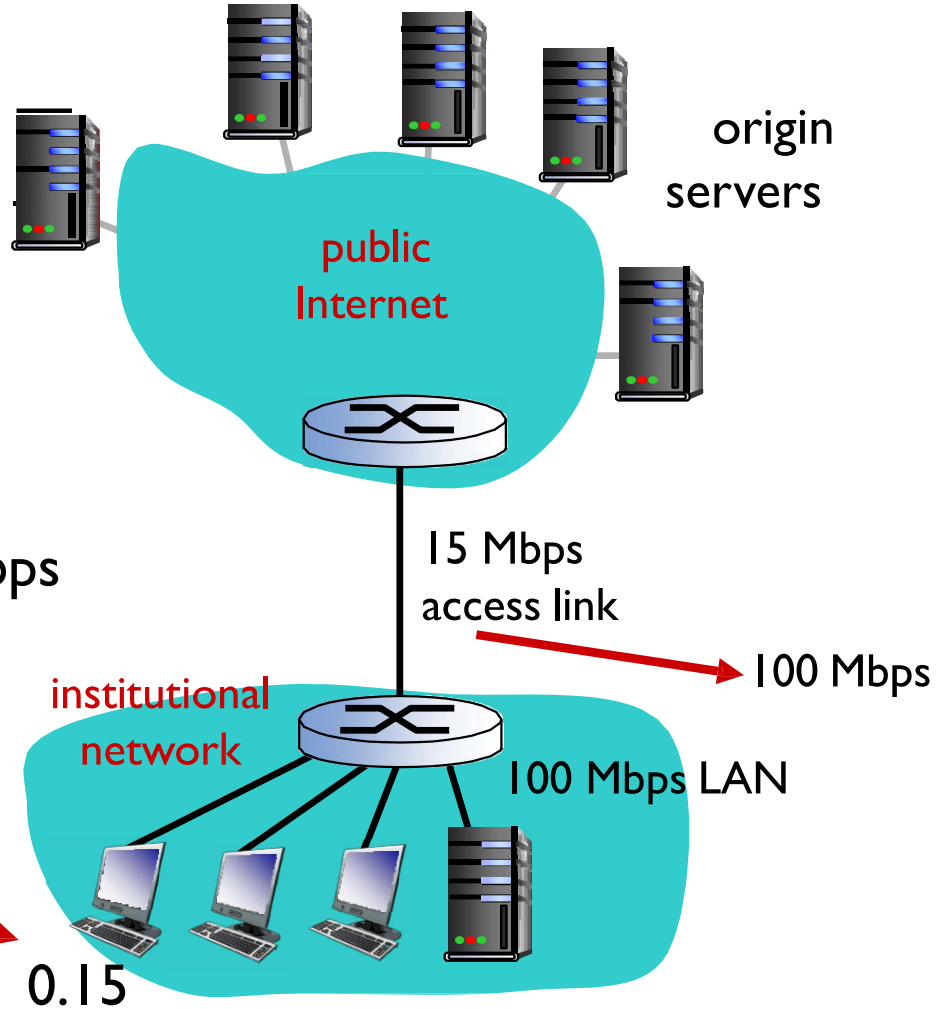
Caching example: fatter access link

assumptions:

- avg object size: 1Mbits
- avg request rate from browsers to origin servers:15/sec
- RTT from the router on the Internet side to any origin server: 2 sec
- access link rate: 15 Mbps

consequences:

- total delay = Internet delay + access delay + LAN delay
- LAN traffic intensity = 0.15
- access link traffic intensity = 1
- total delay = 2 sec + minutes + msecs



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

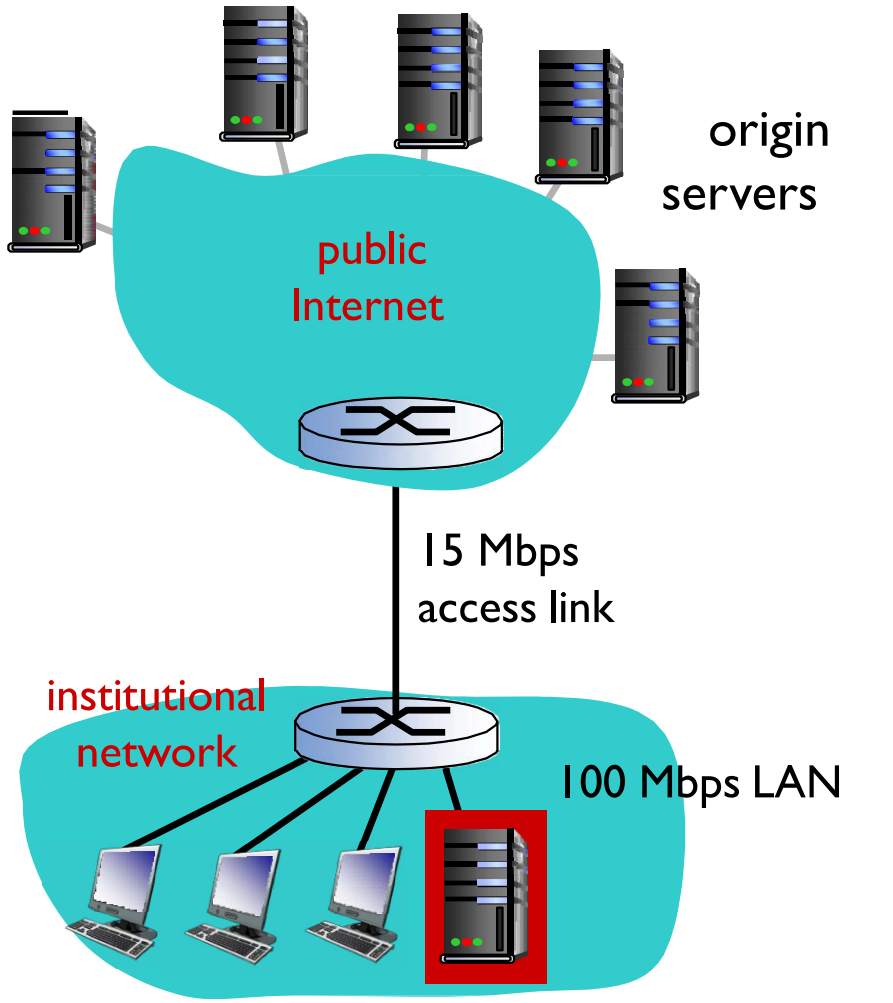
- avg object size: 1Mbits
- avg request rate from browsers to origin servers:15/sec
- RTT from the router on the Internet side to any origin server: 2 sec
- access link rate: 15 Mbps

consequences:

- total delay = Internet delay + access delay + LAN delay

How to compute traffic intensity and delay?

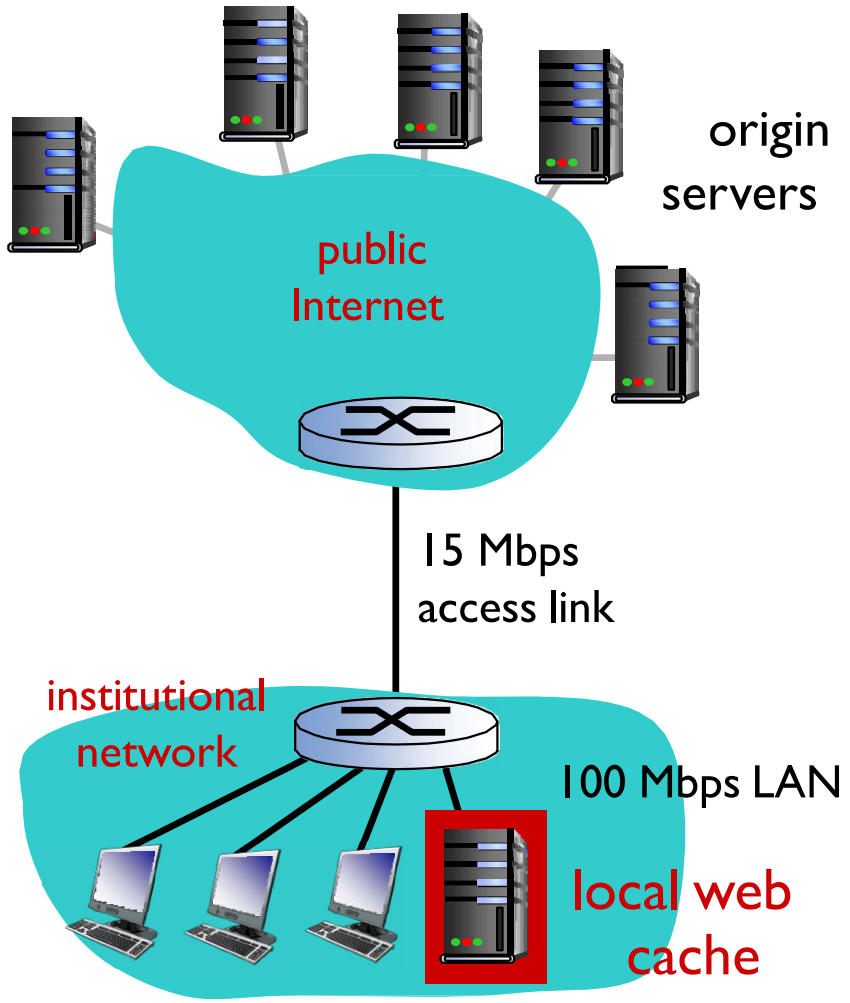
Cost: web cache (cheap!)



Caching example: install local cache

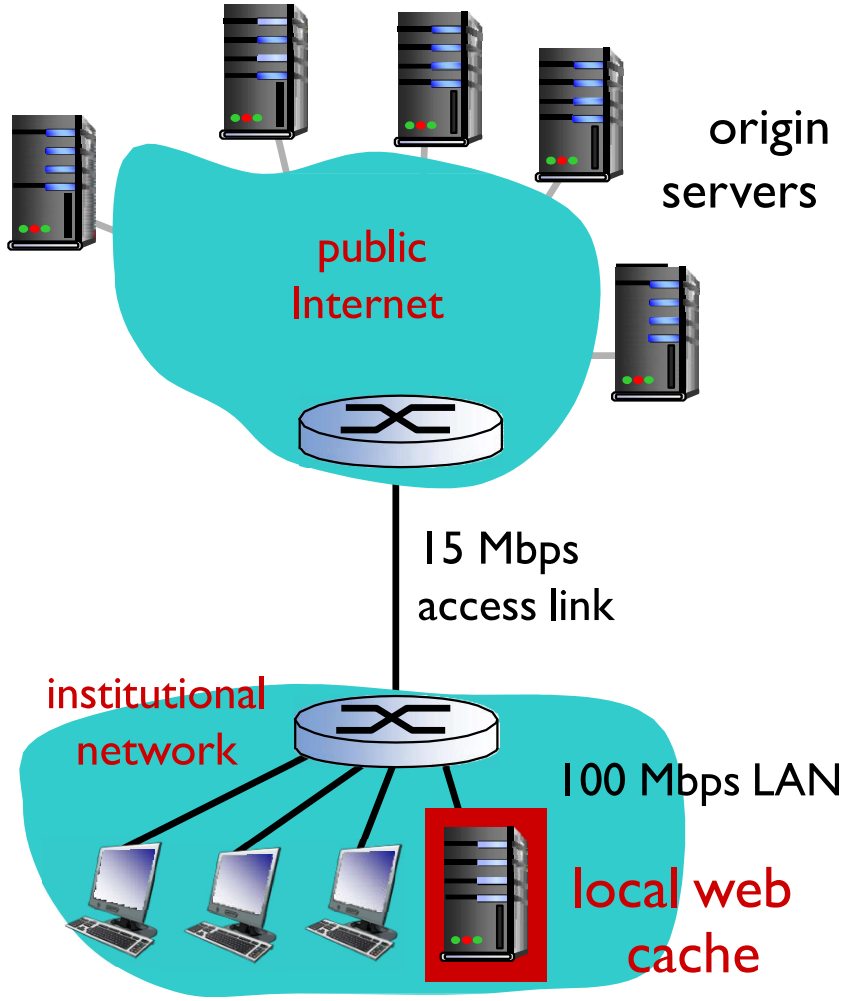
Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin
- access link utilization:
  - 60% of requests use access link
- data rate to browsers over access link = 0.6*15Mbps = 9 Mbps
  - access link traffic intensity = 9/15 = 0.6
  - access delay is negligible (tens of milliseconds)



Caching example: install local cache

- total delay
  - = 0.6 * (Internet delay) +0.4 * (LAN delay)
    = 0.6*2.0 + 0.4 (~msecs)
    = ~ 1.2 secs
  - less than with 100 Mbps link (2 secs)and cheaper too!
- Content Distribution Networks(CDNs)
  - Web caches are increasingly playing an important role in the Internet.
  - installs many geographically distributed caches throughout the Internet
  - thereby localizing much of the traffic



Chapter 2.2: outline

2.2 The Web and HTTP

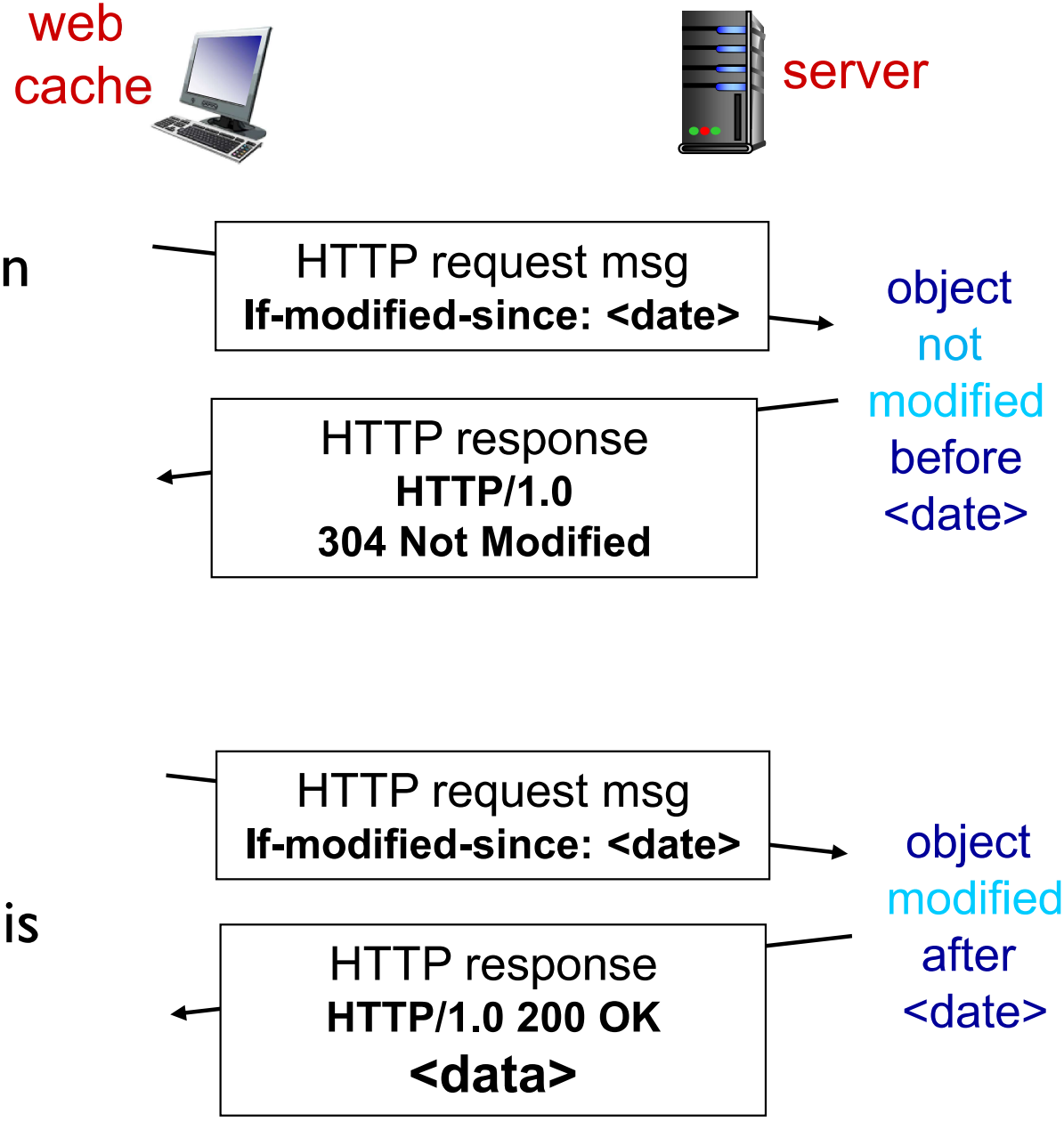
- 2.2.1 Overview of HTTP
- 2.2.2 Non-Persistent and Persistent Connections
- 2.2.3 HTTP Message Format
- 2.2.4 User-Server Interaction: Cookies
- 2.2.5 Web Caching
- 2.2.6 The Conditional GET

Conditional GET

- *Goal:* origin server don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *cache*: specify date of cached copy in HTTP request

  `If-modified-since: <date>`
- *server*: response contains no object if cached copy is up-to-date:

  `HTTP/1.0 304 Not Modified`



Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

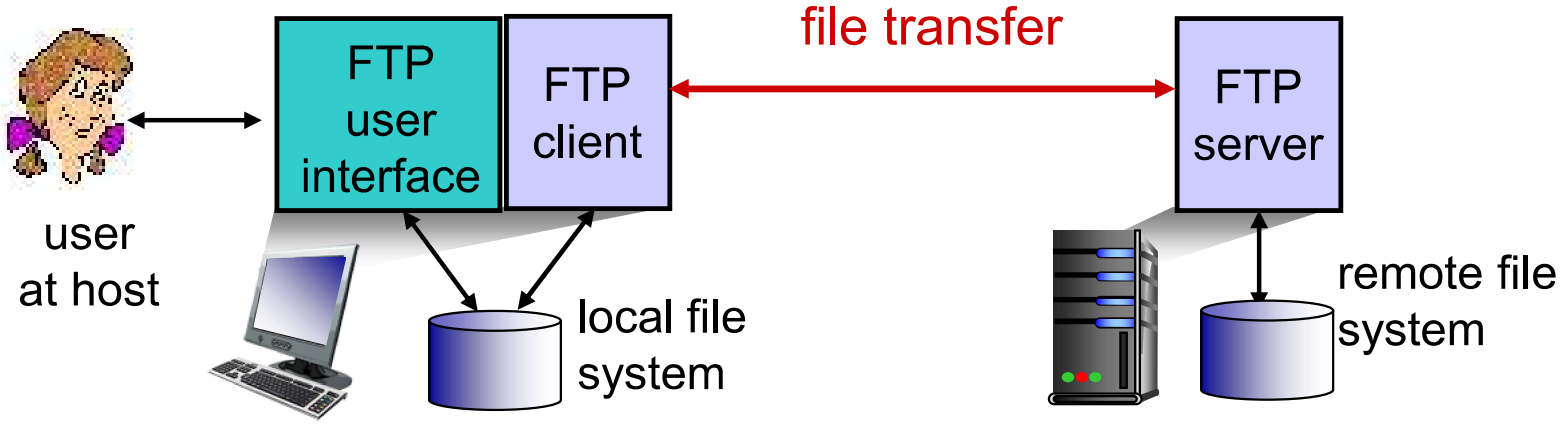
2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

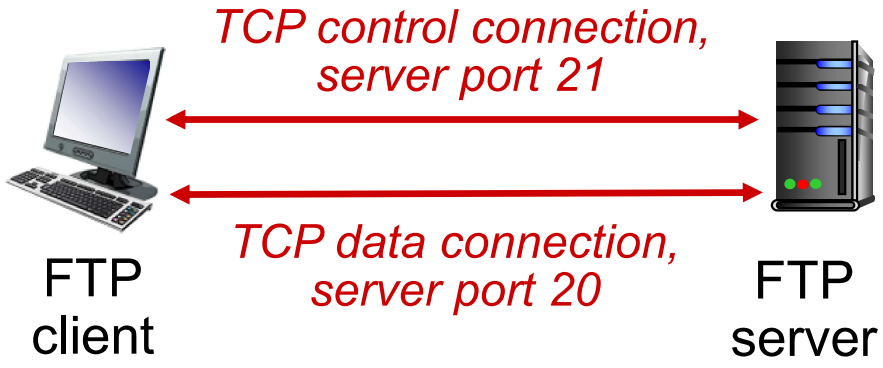
FTP: the file transfer protocol



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

FTP: separate control, data connections

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



- server opens another TCP data connection to transfer another file
- control connection: *"out of band"* (http, smtp:"in-band")
- FTP server maintains "state": current directory, earlier authentication

FTP commands, responses

sample commands:

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

  stores (puts) file onto remote host

sample return codes

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

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

Electronic mail

- Electronic mail has been around since the beginning of the Internet
- As with ordinary postal mail, e-mail is an asynchronous communication medium
- electronic mail is fast, easy to distribute, and inexpensive
- Modern e-mail has many powerful features
  - messages with attachments, hyperlinks, HTML-formatted text, and embedded photos.

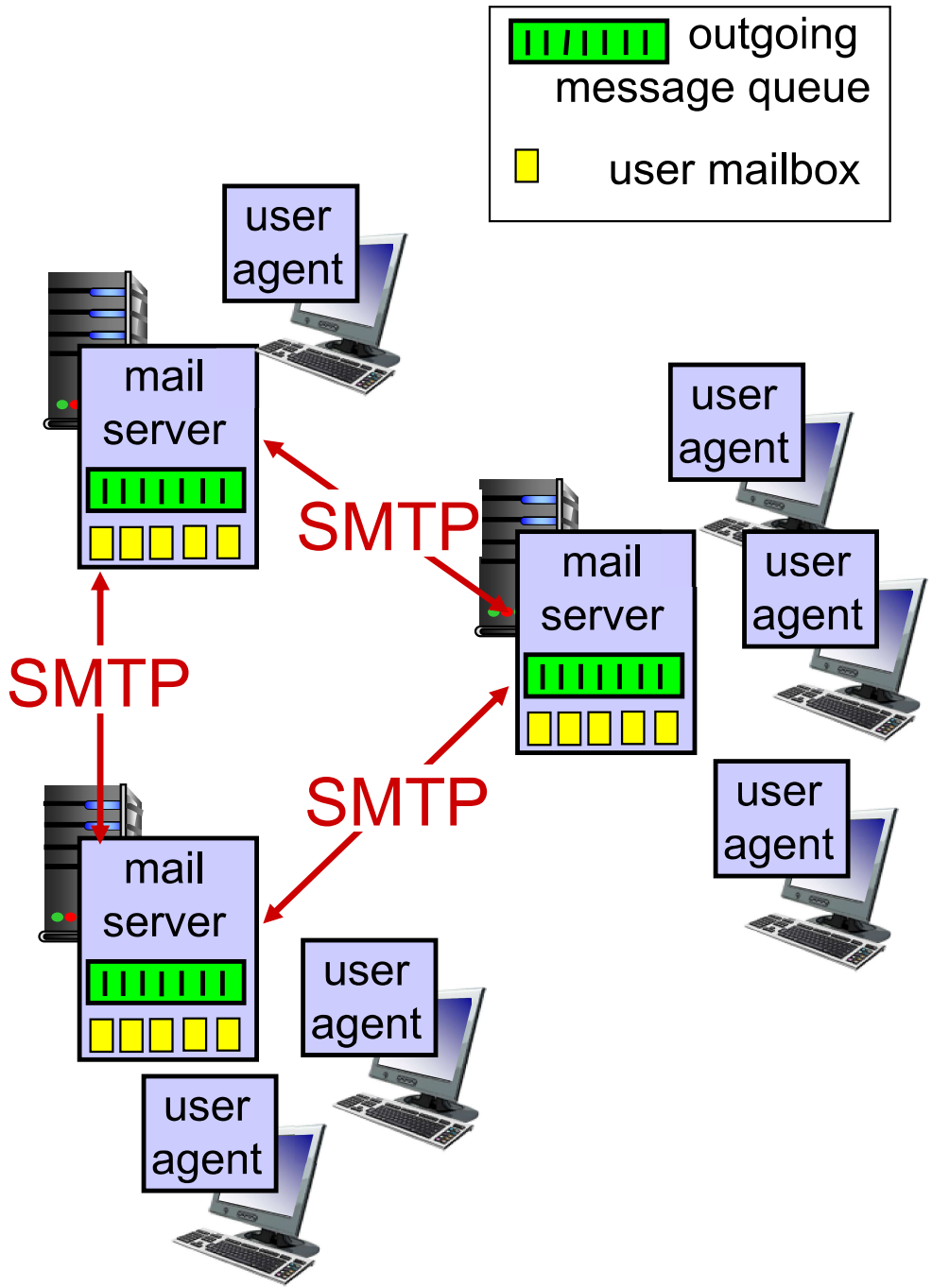
Electronic mail

Three major components:

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

User Agent

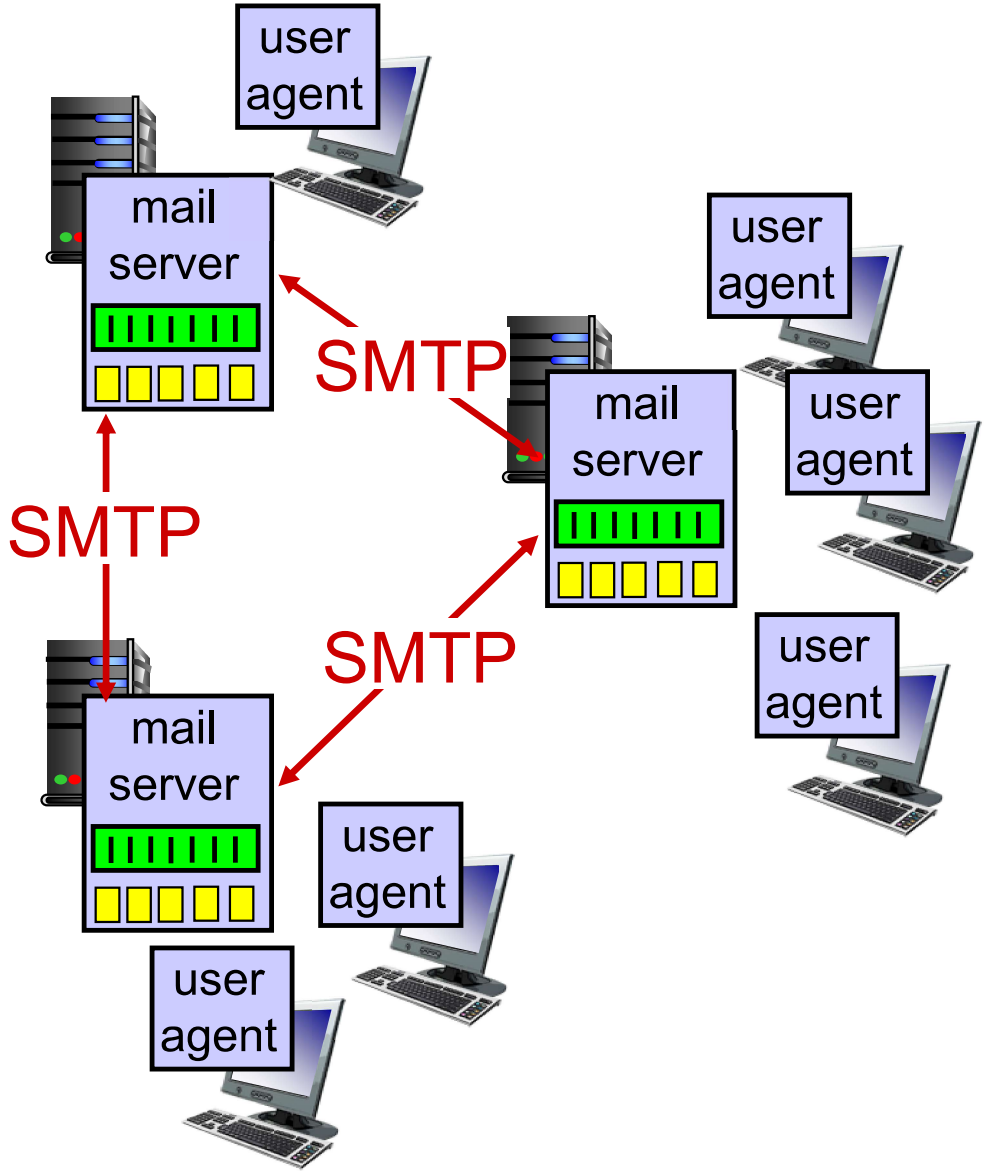
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



Electronic mail: mail servers

mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages
- *SMTP protocol* between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server



Electronic Mail: SMTP [RFC 2821]

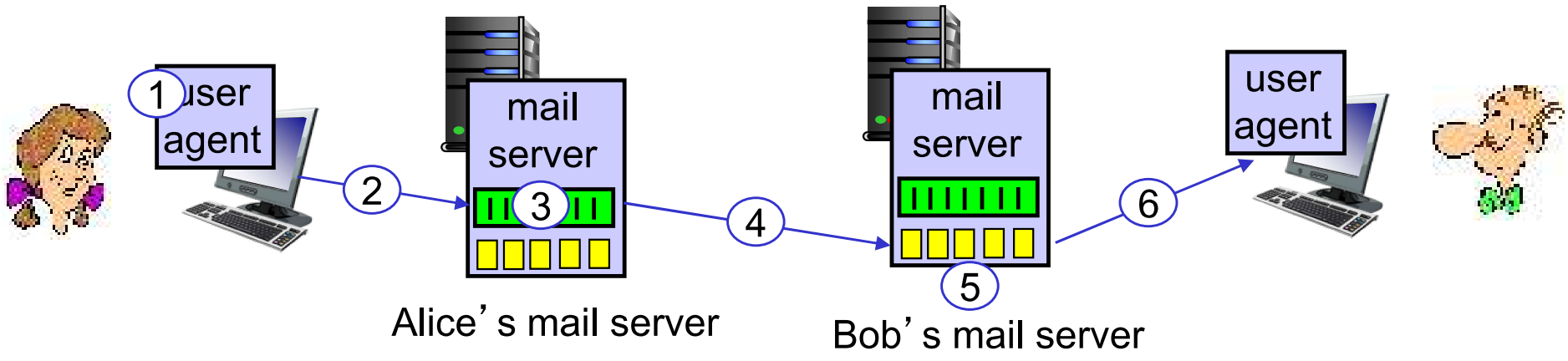
- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP, FTP)
  - commands: ASCII text
  - response: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

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message "to" `bob@someschool.edu`
2) Alice's UA sends message to her mail server; message placed in message queue
3) client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message

电子邮件地址：

收件人的邮箱名@邮箱所在主机的域名



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

SMTP: final words

comparison with HTTP:

similar characteristics:

- both have ASCII command/response interaction, status codes
- both persistent HTTP and SMTP use persistent connections

different characteristics:

- HTTP: pull
- SMTP: push
- SMTP requires each message to be in 7-bit ASCII format
- HTTP: each object encapsulated in its own response msg.
- SMTP: multiple objects sent in multipart msg(one)

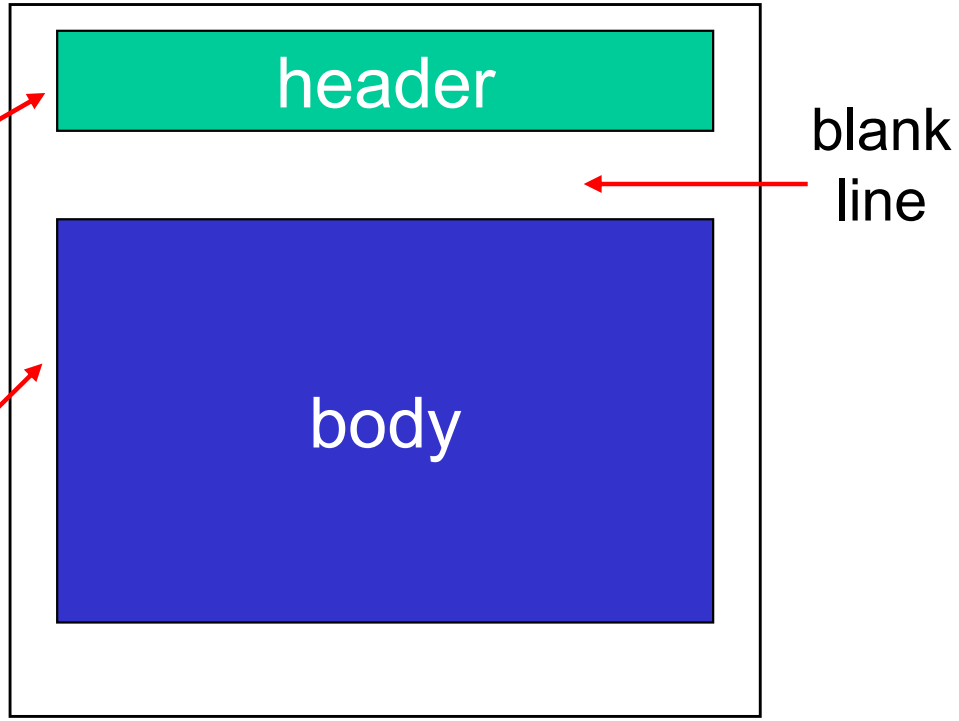
Mail message format

SMTP: protocol for exchanging email msgs

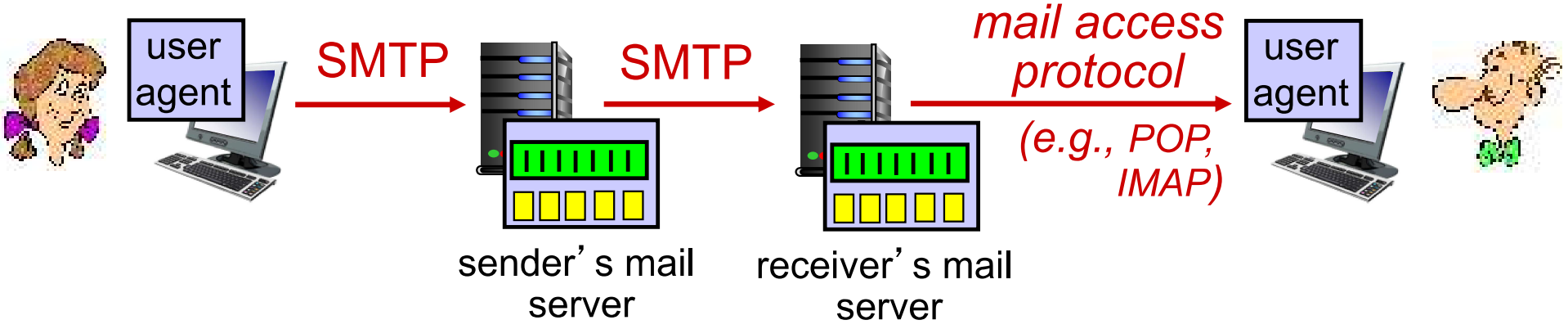
RFC 5322: standard for text message format:

- header lines, e.g.,
  - To: (required)
  - From: (required)
  - Subject: (optional)

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



Mail access protocols



- SMTP: delivery/storage to receiver's server
- mail access protocol: retrieval from server
  - POP3: Post Office Protocol [RFC 1939]: authorization, download
  - IMAP: Internet Mail Access Protocol [RFC 3501]: more features, including manipulation of stored msgs on server
  - HTTP: Gmail, Hotmail, Yahoo, QQmail; university mail, company 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`

Update

telnet mailServer 110

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

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

POP3 (more) and IMAP

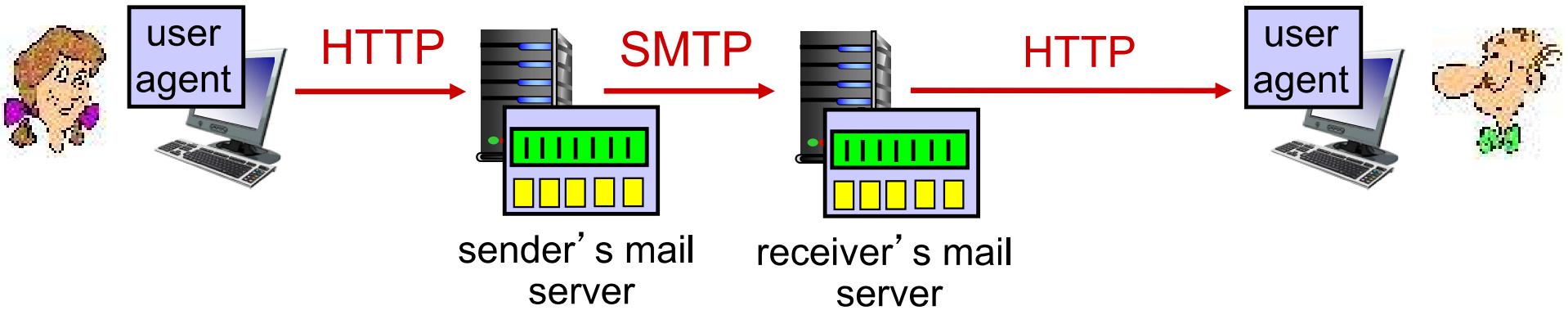
more about POP3

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

IMAP

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

Web-Based E-Mail



- Gmail, Hotmail, Yahoo, QQmail; university mail, company mail, etc
- user agent is Web browser
- The user agent communicates with its remote mailbox via HTTP
- The e-mail message is sent from Bob's mail server to Bob's browser using the HTTP protocol

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

DNS: domain name system

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

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

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

Domain Name System:

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

DNS: services, structure

DNS services

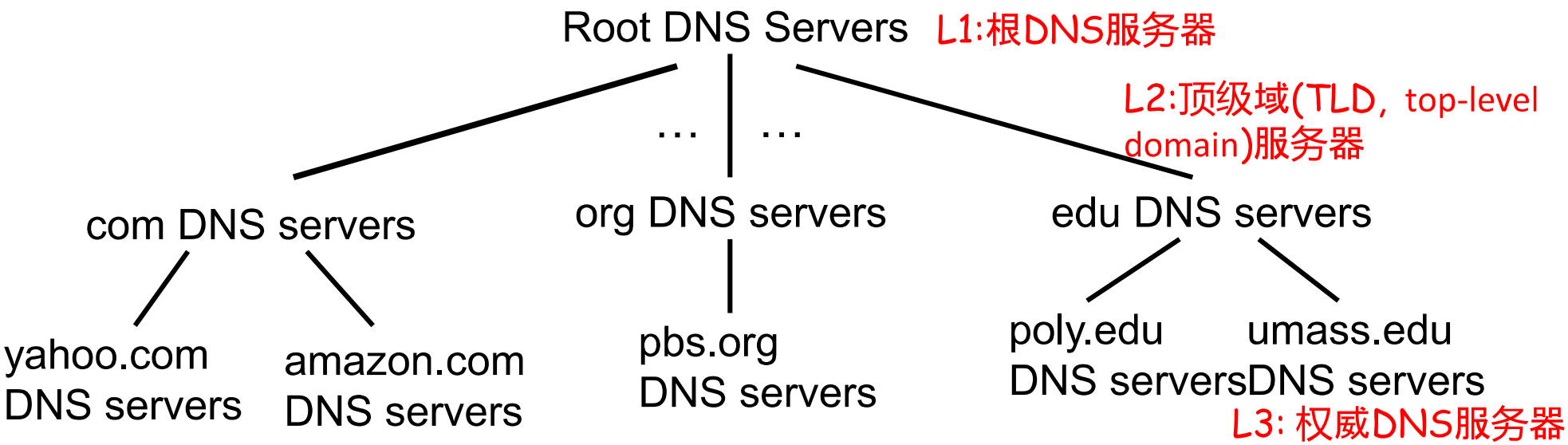
- hostname to IP address translation
- host aliasing（主机别名）
  - canonical（规范主机名）
  - alias names（更易记忆）
- mail server aliasing
  - alias names to canonical hostname and ip
- load distribution（负载分配）
  - replicated（冗余）Web servers: many IP addresses correspond to one name

why not centralize DNS?

- single point of failure单点故障
- traffic volume通信容量
- distant centralized database
  远距离的集中是数据库
- maintenance
  - Keep all records and update frequently

A: doesn't scale!

DNS: a distributed, hierarchical database

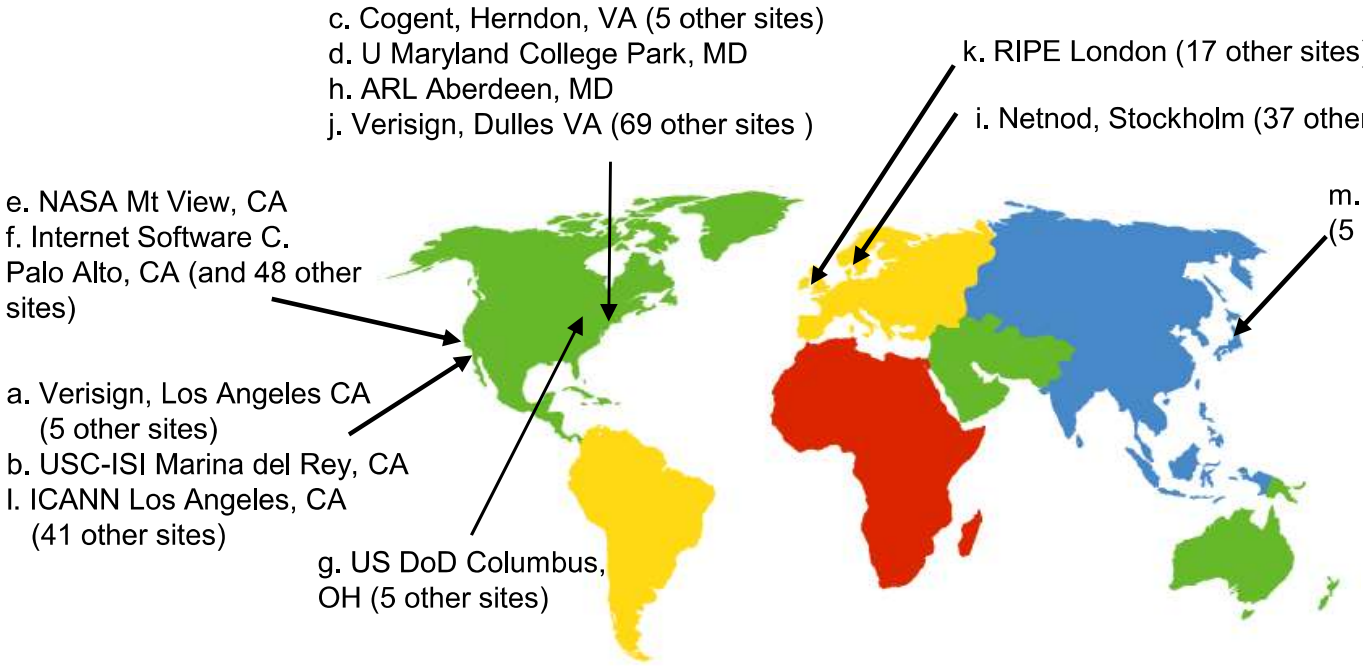


client wants IP for www.amazon.com; 1st approx:

- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS: root name servers

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



13 root name "servers" worldwide

247 root servers (2011)

TLD, authoritative servers

top-level domain (TLD) servers:

- responsible for com, org, net, edu, gov, and all top-level country domains, e.g.: uk, fr, ca, jp
- Verisign Global Registry Services company maintains servers for .com TLD
- Educause company maintains servers 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 (e.g. university, company) themselves or service provider (pay fee)

Local DNS name server

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"默认名称服务器
  - provide IP addresses of one or more of its local DNS server when host connects to ISP
- 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

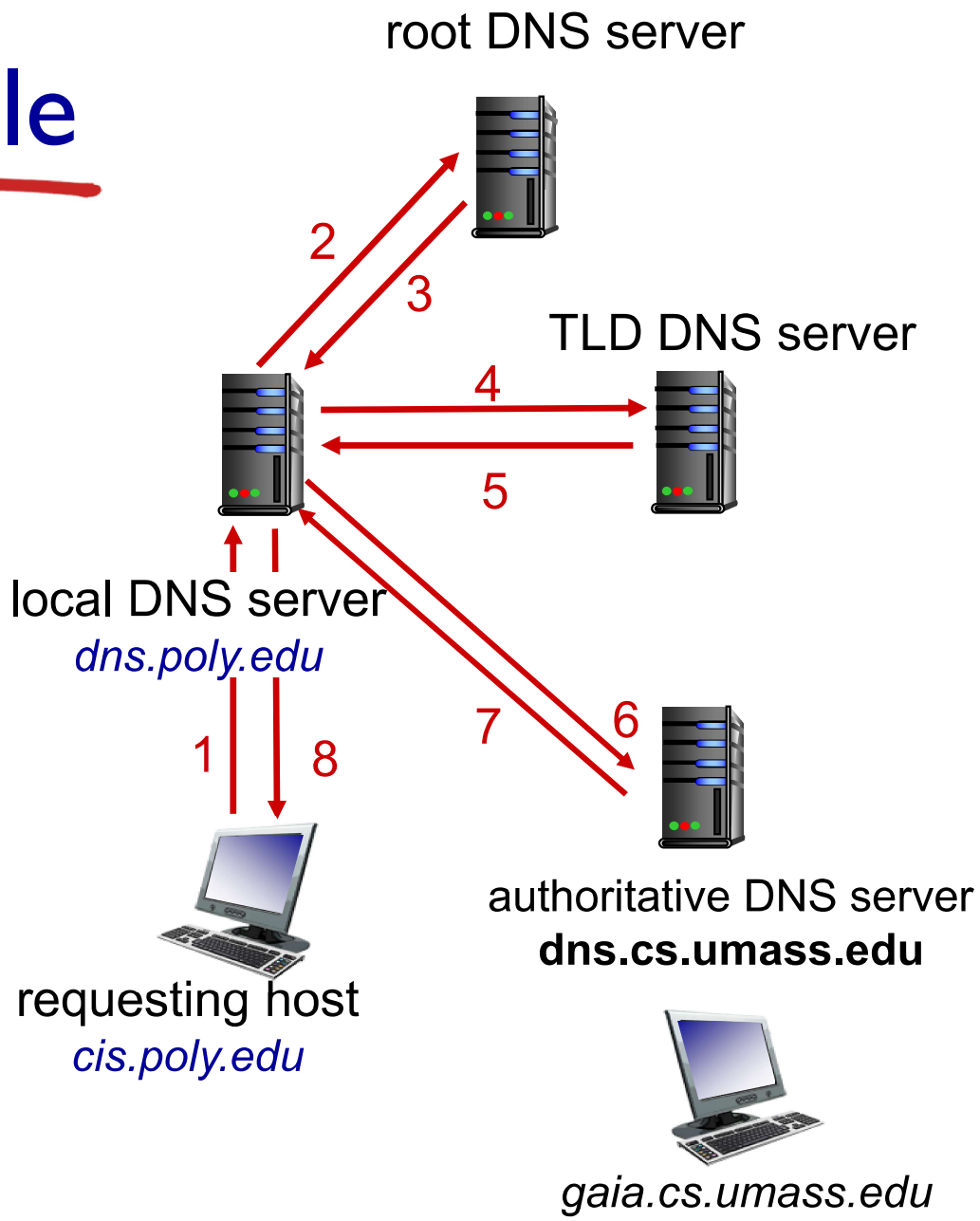
DNS name resolution example

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

iterated (迭代) query:

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"

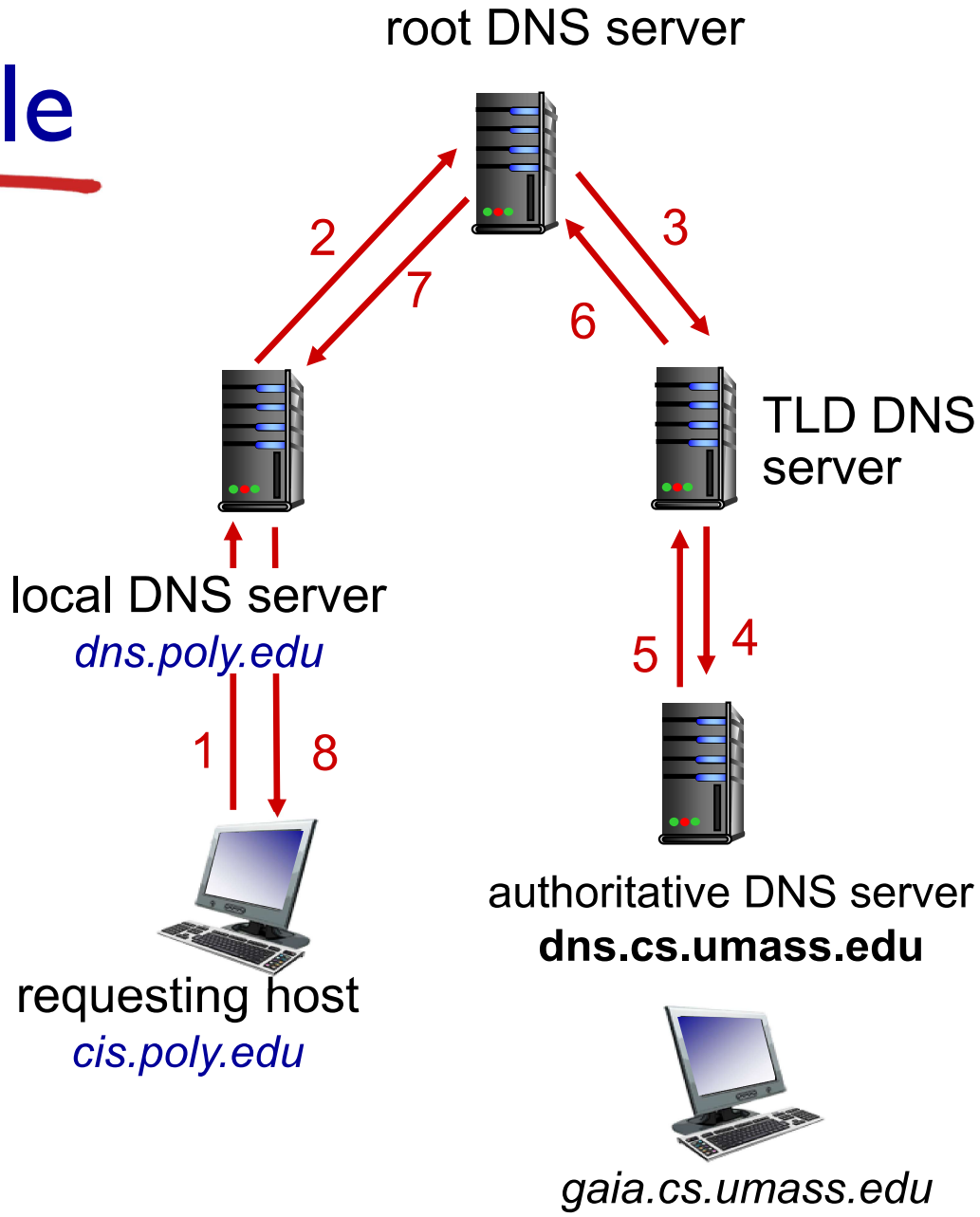
- note: step1 and step8 is recursive query(递归查询).



DNS name resolution example

recursive (递归) query:

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



DNS: caching, updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS records

DNS: distributed db storing resource records (RR)

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

TTL: time to live of the RR, time to be cached

type=A

- **`name`** is hostname
- **`value`** is IP address
- standard hostname-to-IP address (relay1.bar.foo.com, 145.37.93.126, A)

type=NS

- **`name`** is domain (e.g., foo.com)
- **`value`** is hostname of authoritative name server for this domain
- route DNS queries further along in query chain, e.g. (foo.com, dns.foo.com, NS)

type=CNAME

- **`name`** is alias name for some "canonical" (the real) name
- **`value`** is canonical name
- provide querying hosts canonical name for a hostname (foo.com, relay1.bar.foo.com, CNAME)

type=MX

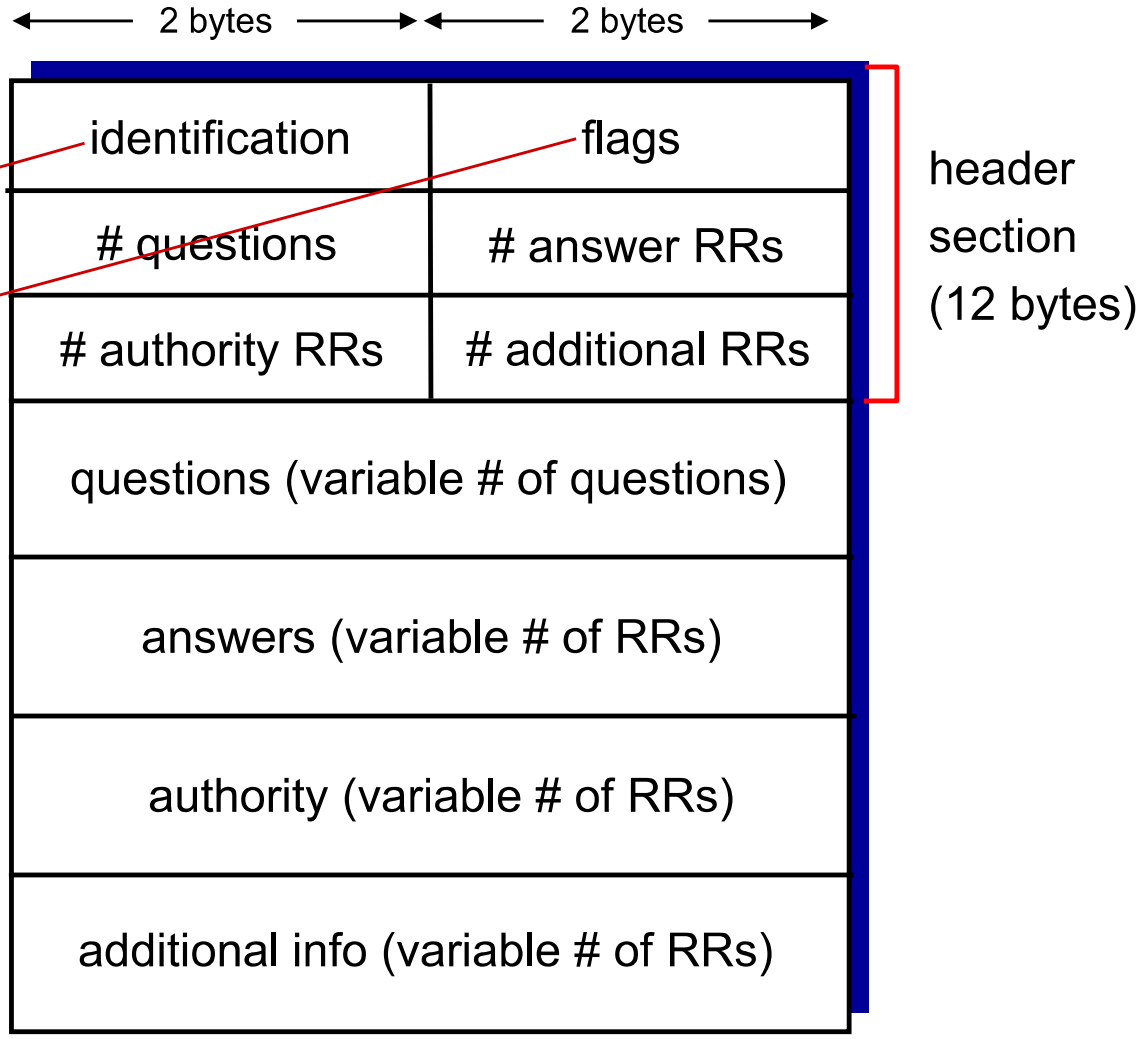
- **Value** is canonical name of a mail server associated with alias **Name**, e.g., (foo.com, mail.bar.foo.com, MX)
- same alias for mails server and one of other servers

DNS protocol, messages

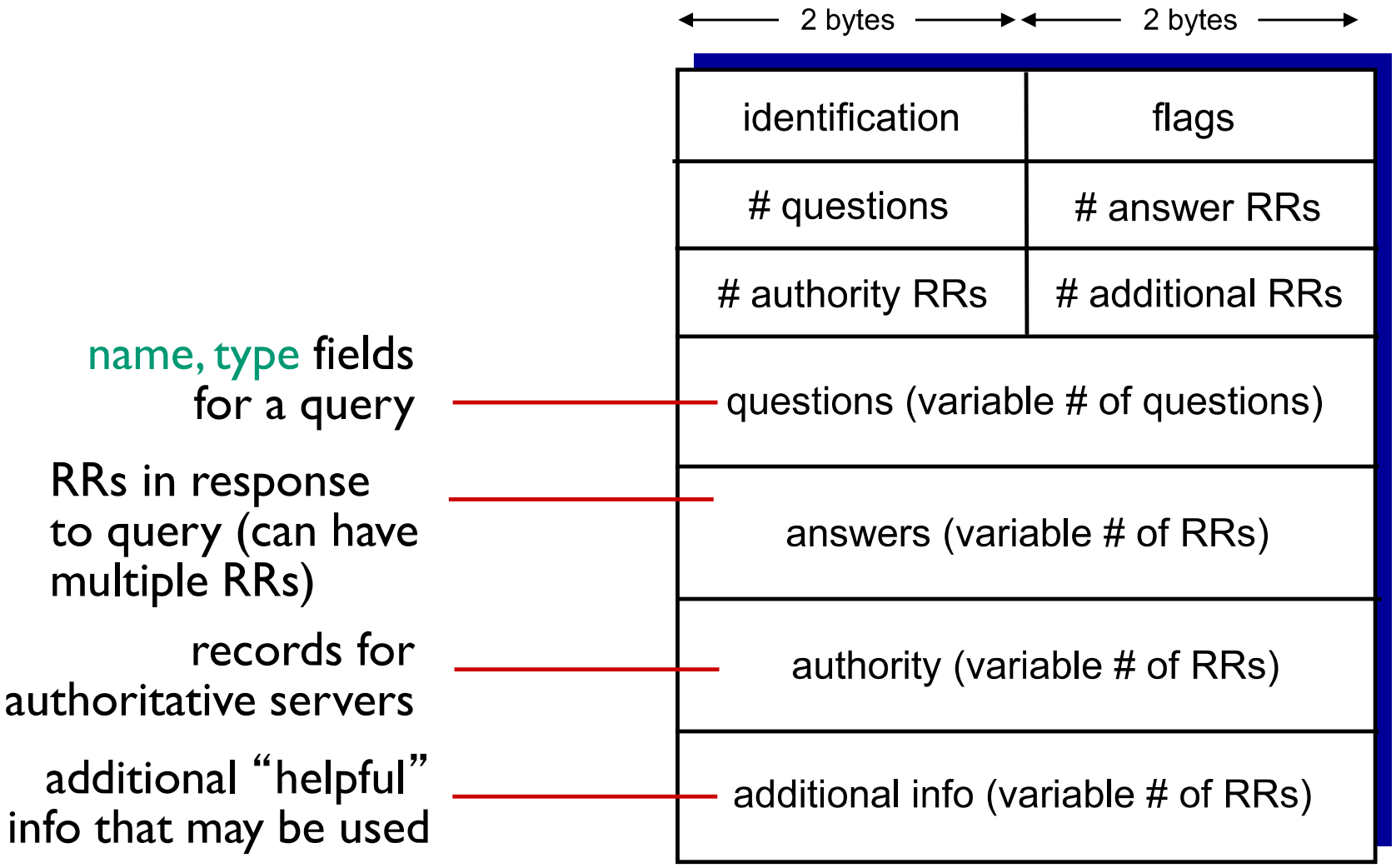
- *query* and *reply* messages, both with same *message format*

msg header

- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query(0) or reply(1)
  - reply is authoritative
  - recursion desired
  - recursion available



DNS protocol, messages



nslookup Tool

- How would you like to send a DNS query message directly from the host you're working on to some DNS server?
- **nslookup program**, available from most Windows and UNIX platforms

```
C:\Users\scu>nslookup
默认服务器:  public1.114dns.com
Address:  114.114.114.114
```

```
C:\Users\scu>nslookup -qt=A www.baidu.com
服务器:  public1.114dns.com
Address:  114.114.114.114

非权威应答:
名称:    www.a.shifen.com
Addresses:  14.215.177.39
          14.215.177.38
Aliases:  www.baidu.com
```

nslookup Tool

- nslookup
- nslookup -qt=A www.baidu.com
- nslookup -qt=NS qq.com
- nslookup -qt=MX qq.com
- nslookup –qt=CNAME qq.com

```
C:\Users\scu>nslookup -qt=MX qq.com
服务器:  public1.114dns.com
Address:  114.114.114.114

非权威应答:
qq.com  MX preference = 10, mail exchanger = mx3.qq.com
qq.com  MX preference = 20, mail exchanger = mx2.qq.com
qq.com  MX preference = 30, mail exchanger = mx1.qq.com
```

```
C:\Users\scu>nslookup -qt=NS qq.com
服务器:  public1.114dns.com
Address:  114.114.114.114

非权威应答:
qq.com  nameserver = ns3.qq.com
qq.com  nameserver = ns4.qq.com
qq.com  nameserver = ns2.qq.com
qq.com  nameserver = ns1.qq.com
```

```
C:\Users\scu>nslookup -qt=CNAME qq.com
服务器:  public1.114dns.com
Address:  114.114.114.114

qq.com
        primary name server = ns1.qq.com
        responsible mail addr = webmaster.qq.com
        serial  = 1330914143
        refresh = 3600 (1 hour)
        retry   = 300 (5 mins)
        expire  = 86400 (1 day)
        default TTL = 300 (5 mins)
```

Inserting records into DNS

- example: new startup "Network Utopia"
- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - Internet Corporation for Assigned Names and Numbers (ICANN,互 联网名称与数字地址分配机构) accredit(授权) the various registrars (list of accredited http://www.internic.net)
  - 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 mail.networkutopia.com

Attacking DNS

DDoS attacks

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

Redirect attacks

- Man-in-middle(中间人攻击）
  - Intercept queries, and returns bogus replies
- DNS poisoning（毒害攻击）
  - Send bogus（伪造）relies to DNS server, which caches

Exploit DNS for DDoS（放大攻击）

- Send queries with spoofed（欺骗）source address: target IP
- Requires amplification

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

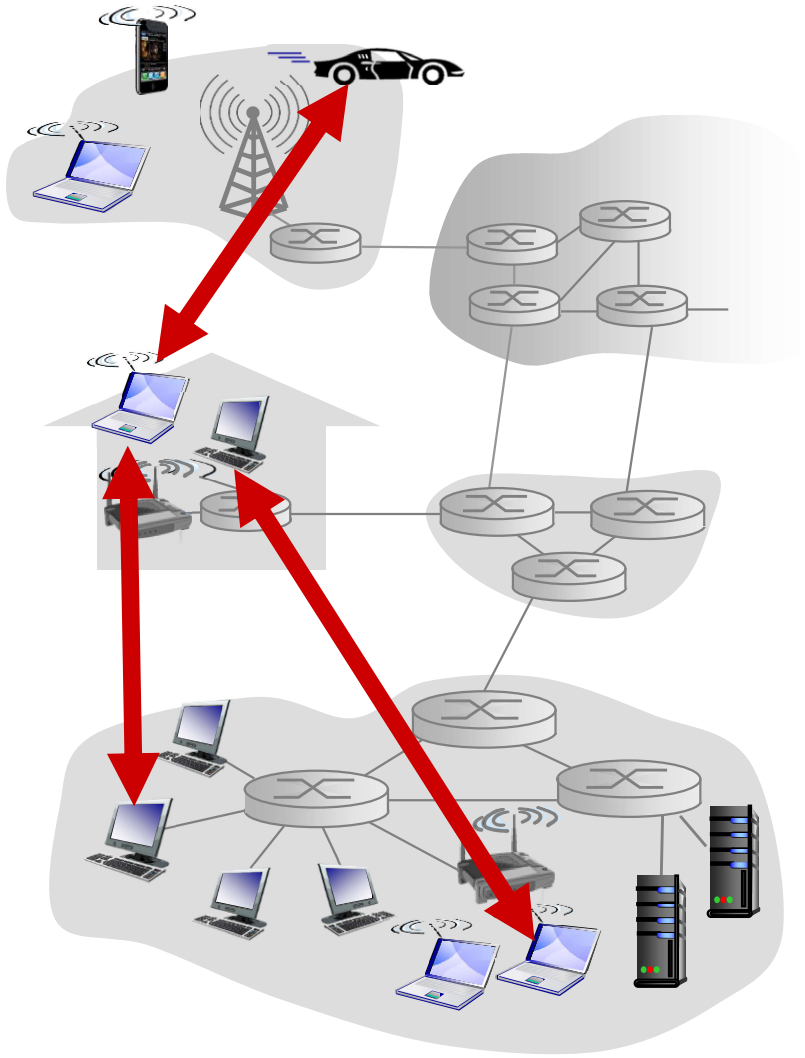
2.7 socket programming with UDP and TCP

Pure P2P architecture

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

examples:

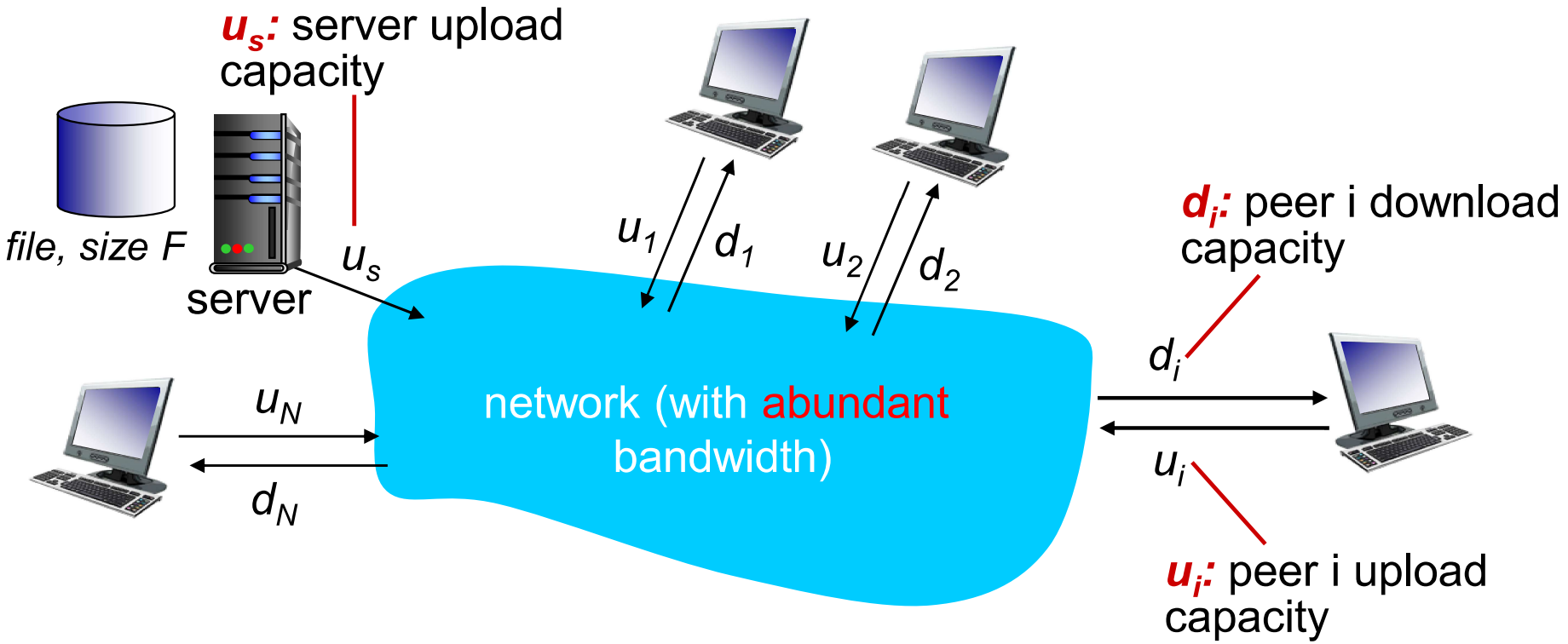
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)



File distribution: client-server vs P2P

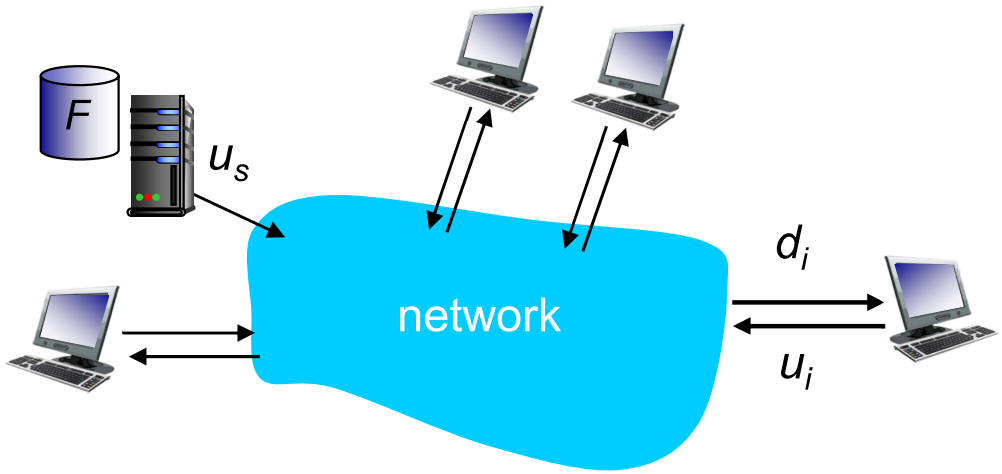
Question: how much time to distribute file (size *F*) from one server to *N* *peers*?

- peer upload/download capacity is limited resource



File distribution time: **client-server**

- *server transmission:* must sequentially send (upload) *N* file copies:
  - time to send one copy: F/u_s
  - time to send N copies: NF/u_s
- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}



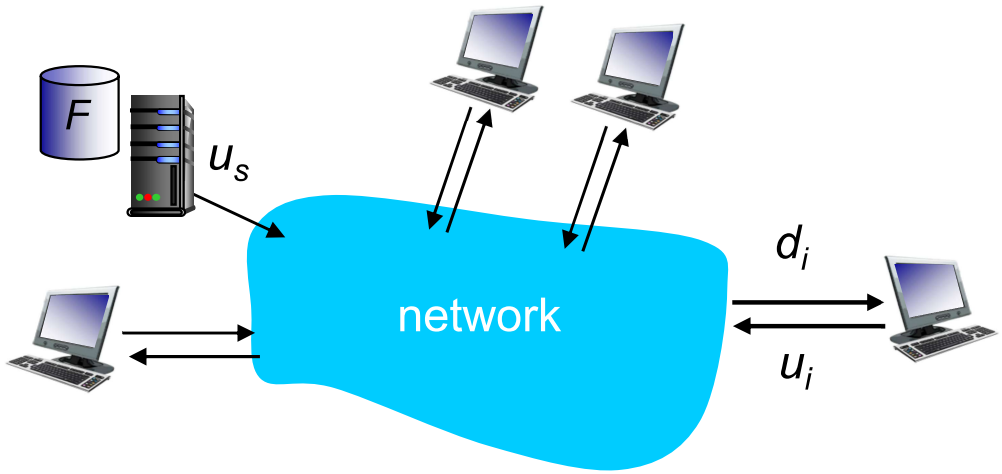
time to distribute F to N clients using client-server approach

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

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 $u_s + \Sigma u_i$



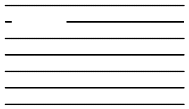
time to distribute F to N clients using P2P approach

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

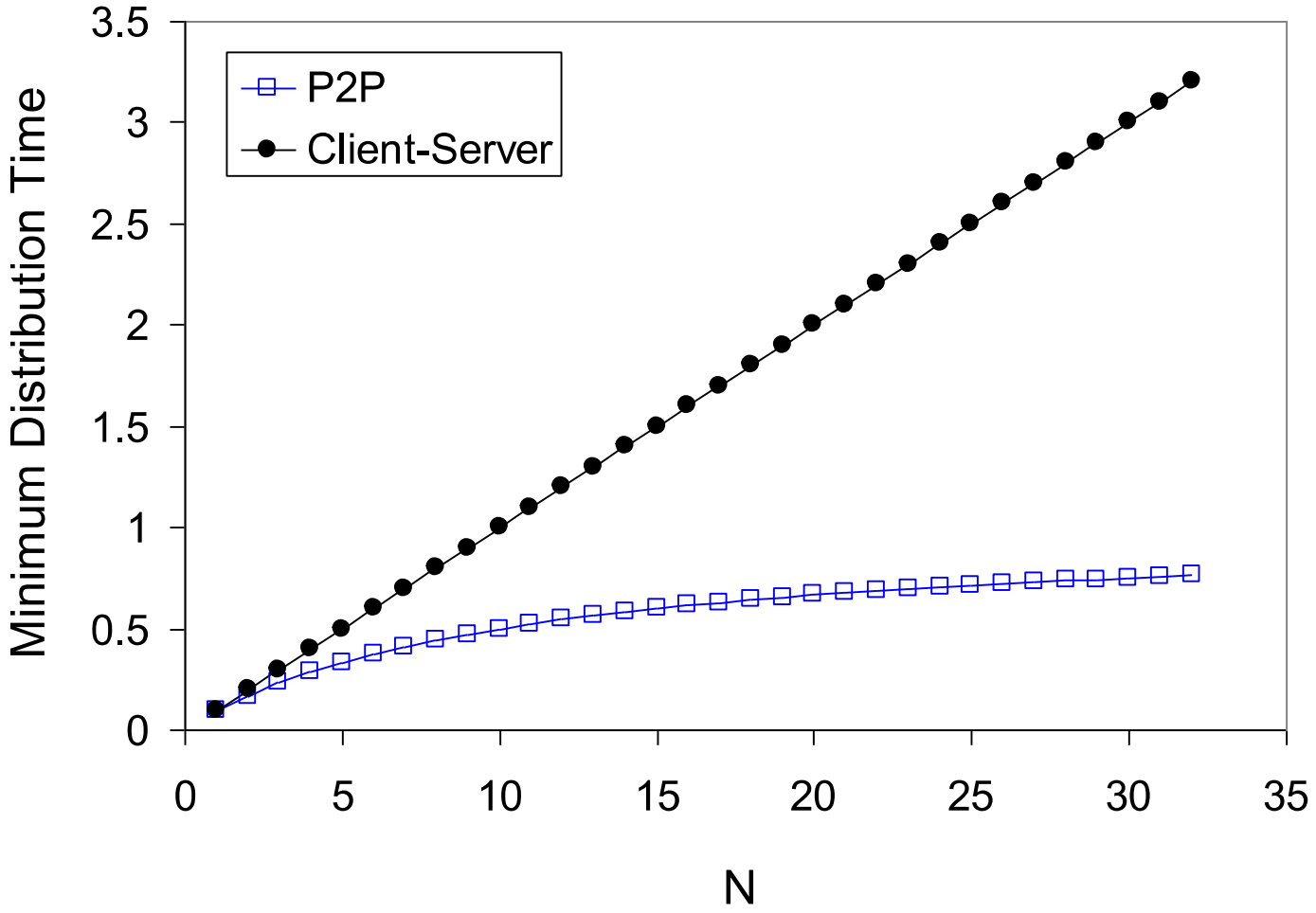
increases linearly in N …

… but so does this, as each peer brings service capacity

Client-server vs. P2P: example



client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$

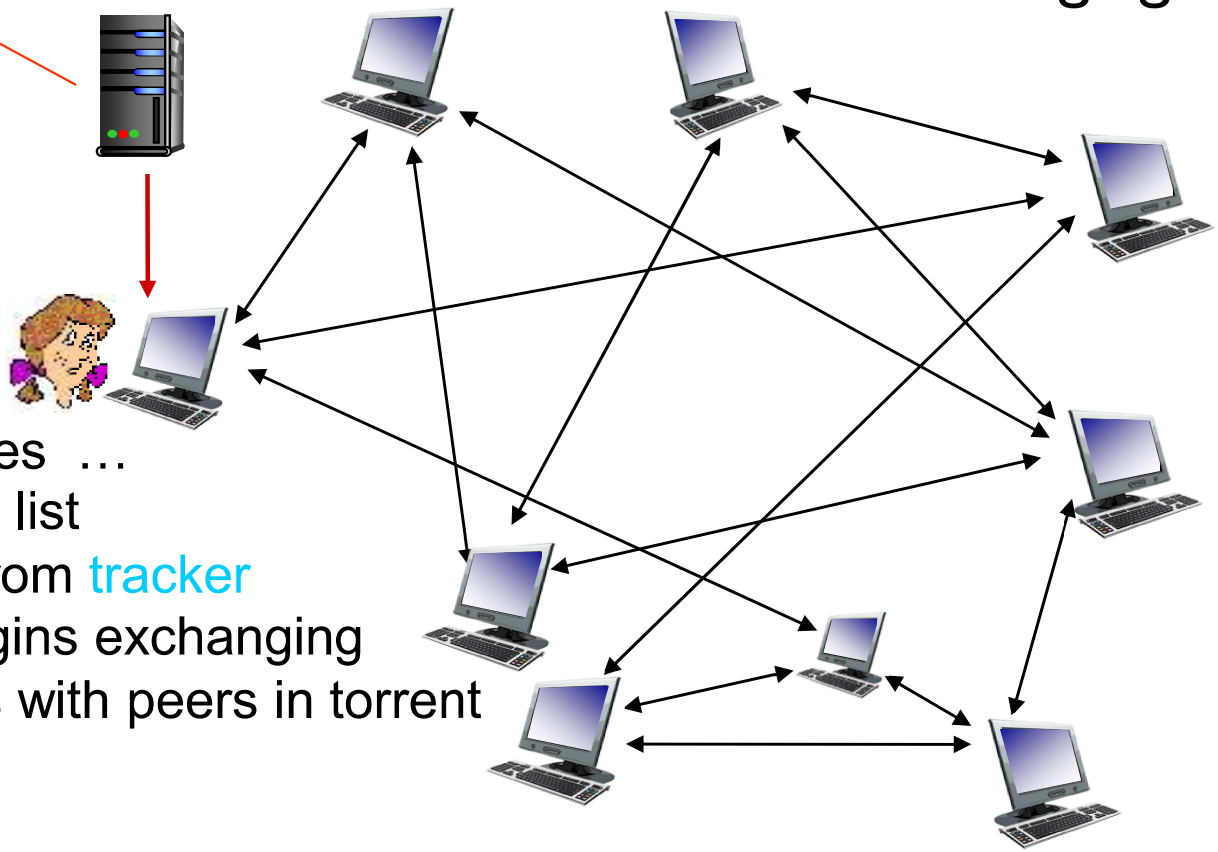


P2P file distribution: BitTorrent

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

tracker: tracks peers participating in torrent

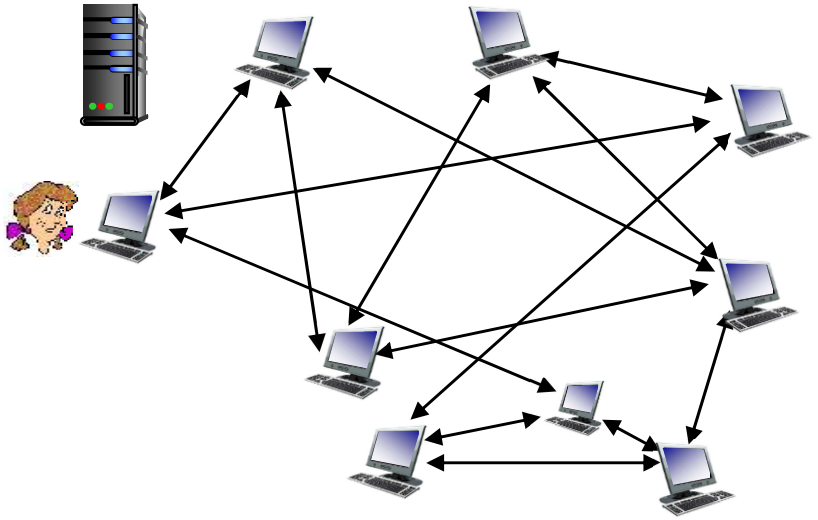
torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

P2P file distribution: BitTorrent

- peer joining torrent:
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
  - has no chunks, but will accumulate them over time from other peers



- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
  - *churn (搅动):* peers may come and go
  - once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

requesting chunks:

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

sending chunks: tit-for-tat

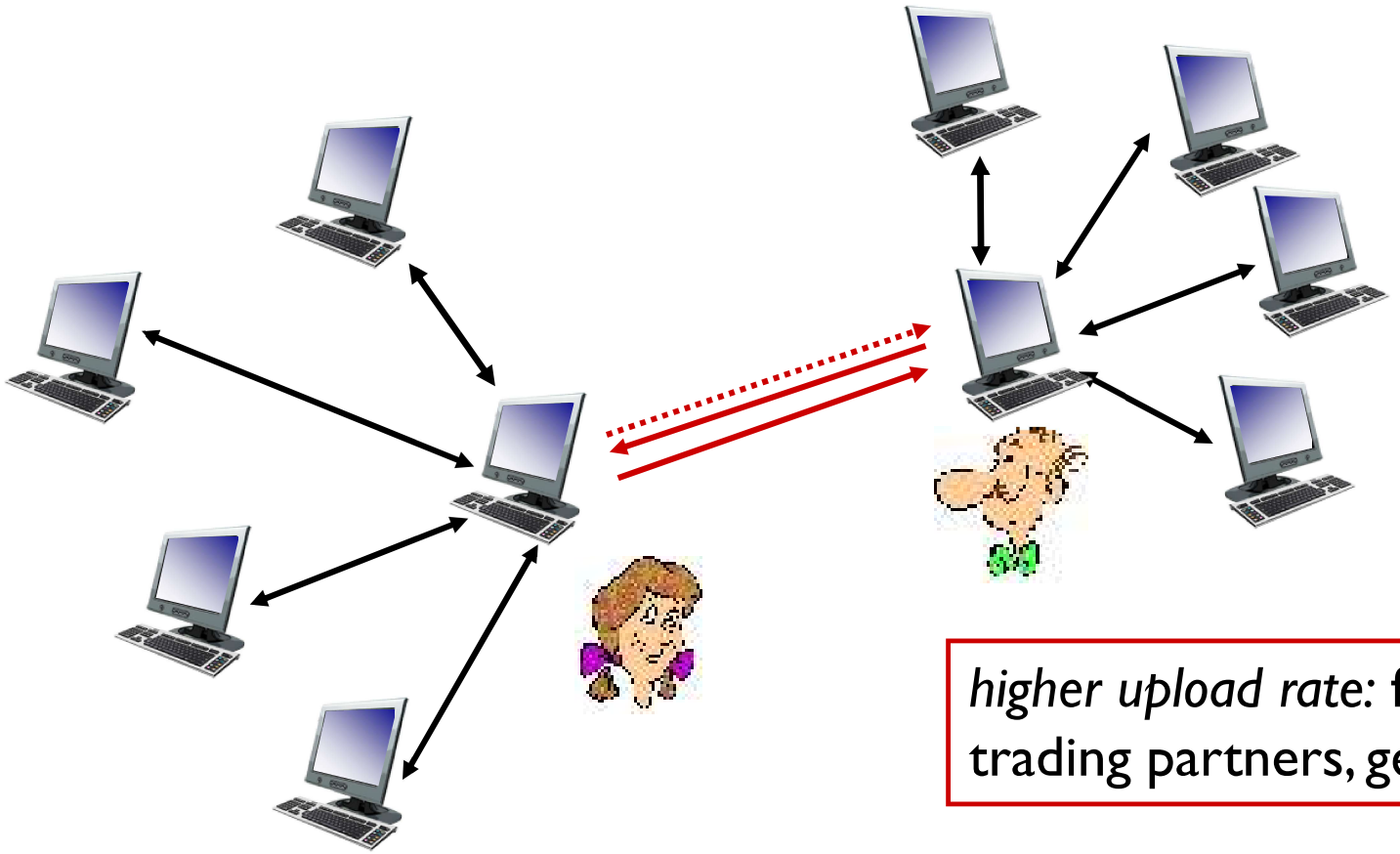
- Alice **sends** chunks to those four peers currently **sendi**ng her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke（疏通）" this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates (酬谢))

(3) Bob becomes one of Alice's top-four providers



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

Distributed Hash Table (DHT)

- DHT: a *distributed P2P database*
- Simple database has (key, value) pairs;examples:
  - key: ss number; value: human name
  - key: movie title; value: IP address
- Distribute the (key, value) pairs over the (millions of peers)
  - a peer queries DHT with key
    - DHT returns values that match the key
  - peers can also insert (key, value) pairs

Q: how to assign keys to peers?

- central issue:
  - assigning (key, value) pairs to peers.
- basic idea:
  - convert each key to an integer
  - Assign integer to each peer
  - put (key,value) pair in the peer that is closest to the key

DHT identifiers

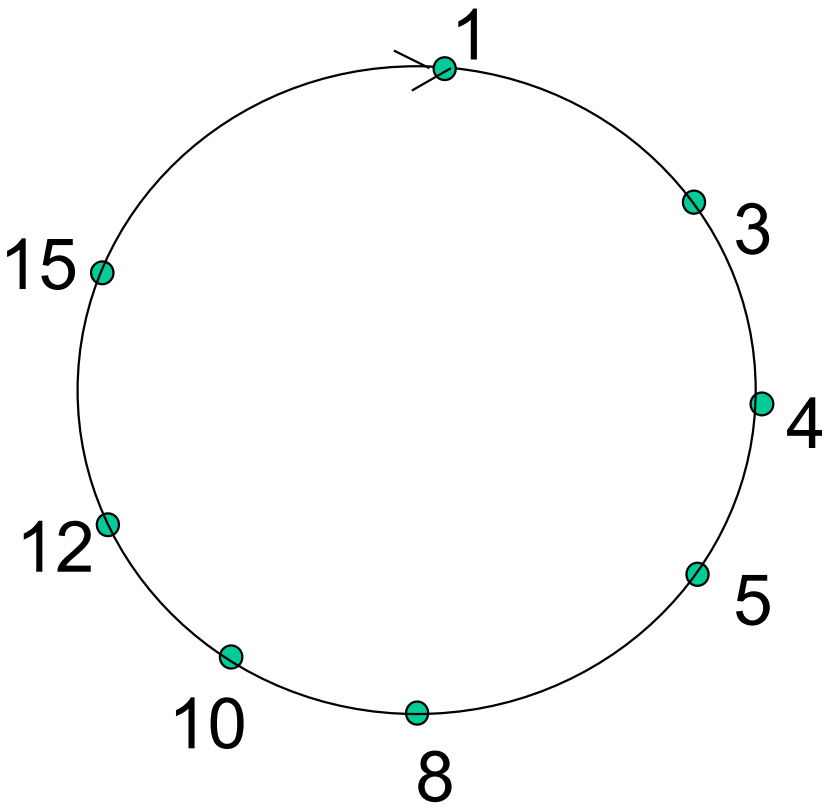
- assign integer identifier to each peer in range $[0,2^n-1]$ for some *n*.
  - each identifier represented by *n* bits.

- require each key to be an integer in same range
- to get integer key, hash original key
  - e.g., key = hash("Led Zeppelin IV")
  - this is why its is referred to as a *distributed "hash" table*

Assign keys to peers

- rule: assign key to the peer that has the *closest*(最邻近）ID.
- convention in lecture: closest is the *immediate successor* of the key.
- e.g., *n*=4; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

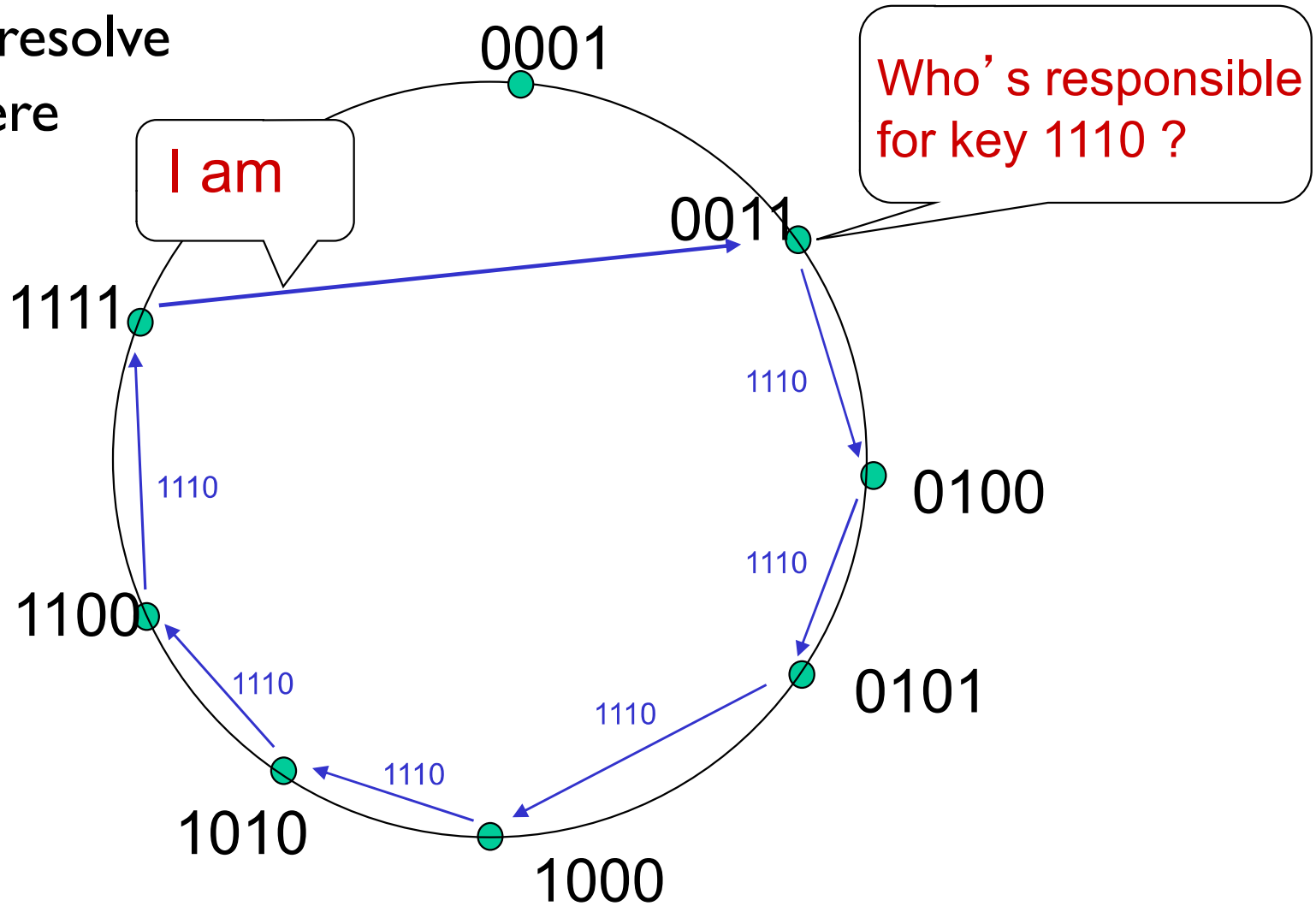
Circular DHT (1)



- each peer *only* aware of immediate successor and predecessor.
- "overlay network"

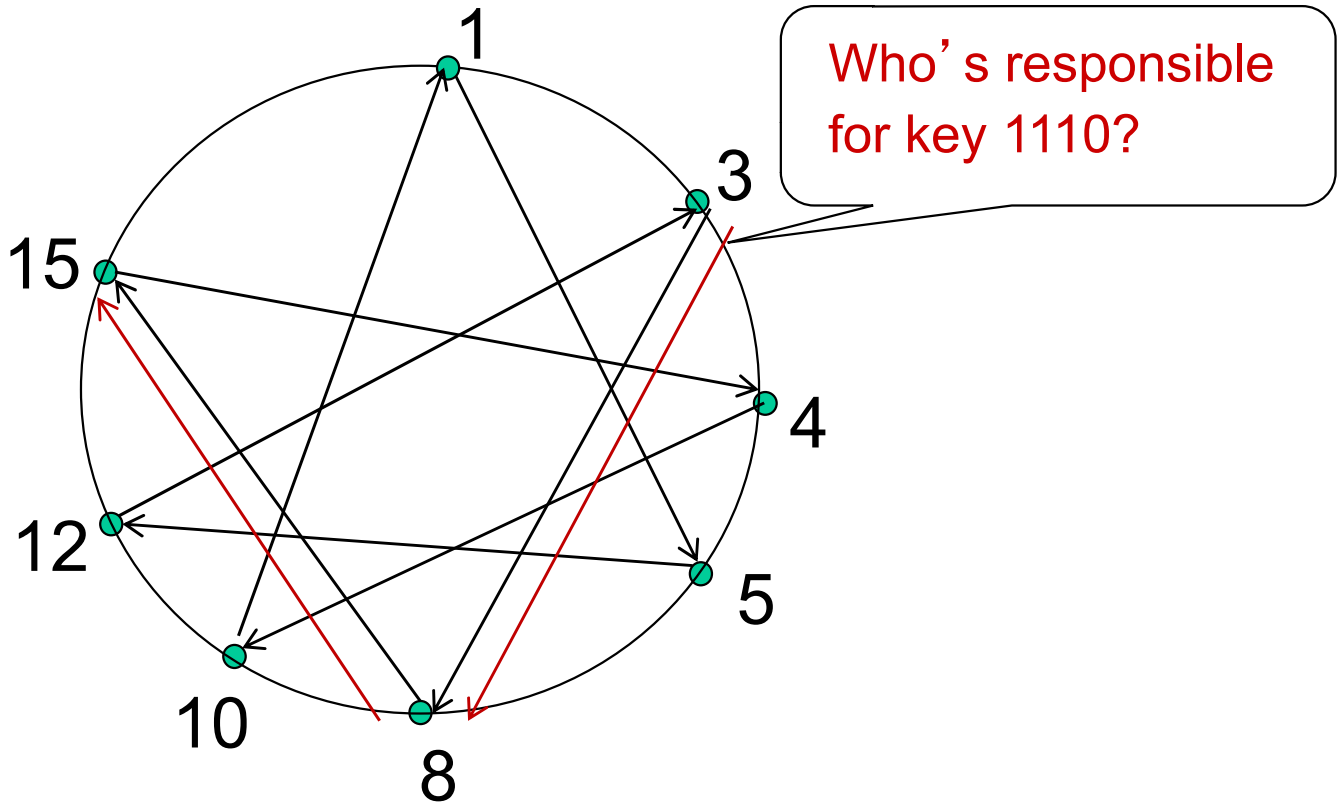
Circular DHT (1)

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



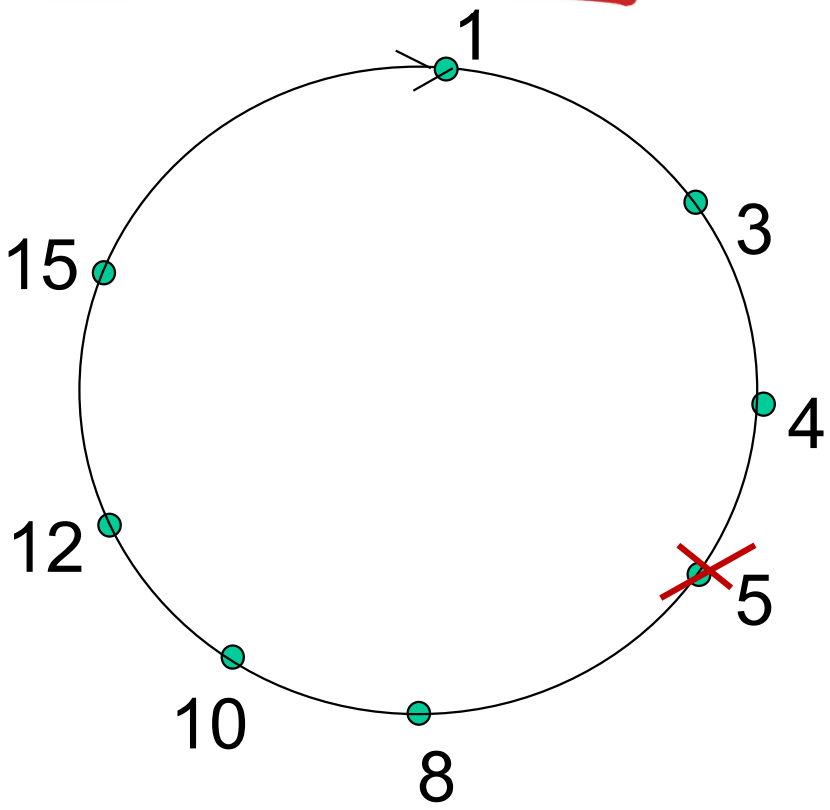
Define closest as closest successor

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



handling peer churn:

- peers may come and go (churn)
- each peer knows address of its two successors
- each peer periodically pings its two successors to check aliveness
- if immediate successor leaves, choose next successor as new immediate successor

example: peer 5 abruptly leaves

- peer 4 detects peer 5 departure; 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?

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

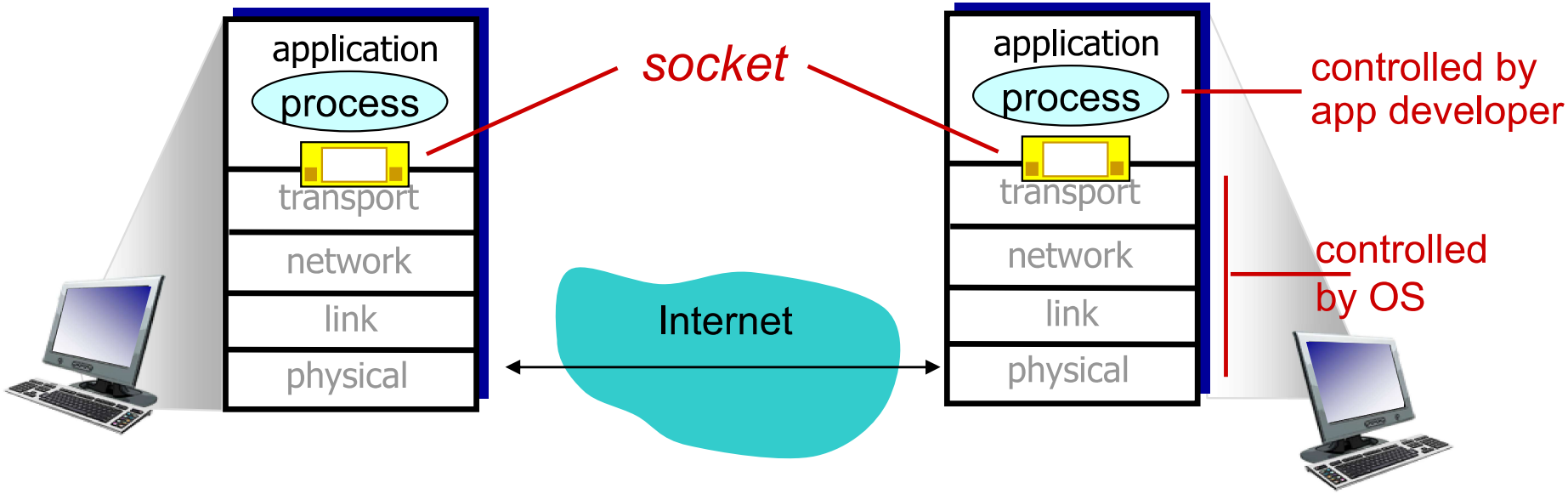
2.6 P2P applications

2.7 socket programming with UDP and TCP

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Socket programming *with UDP*

UDP: no "connection" between client & server

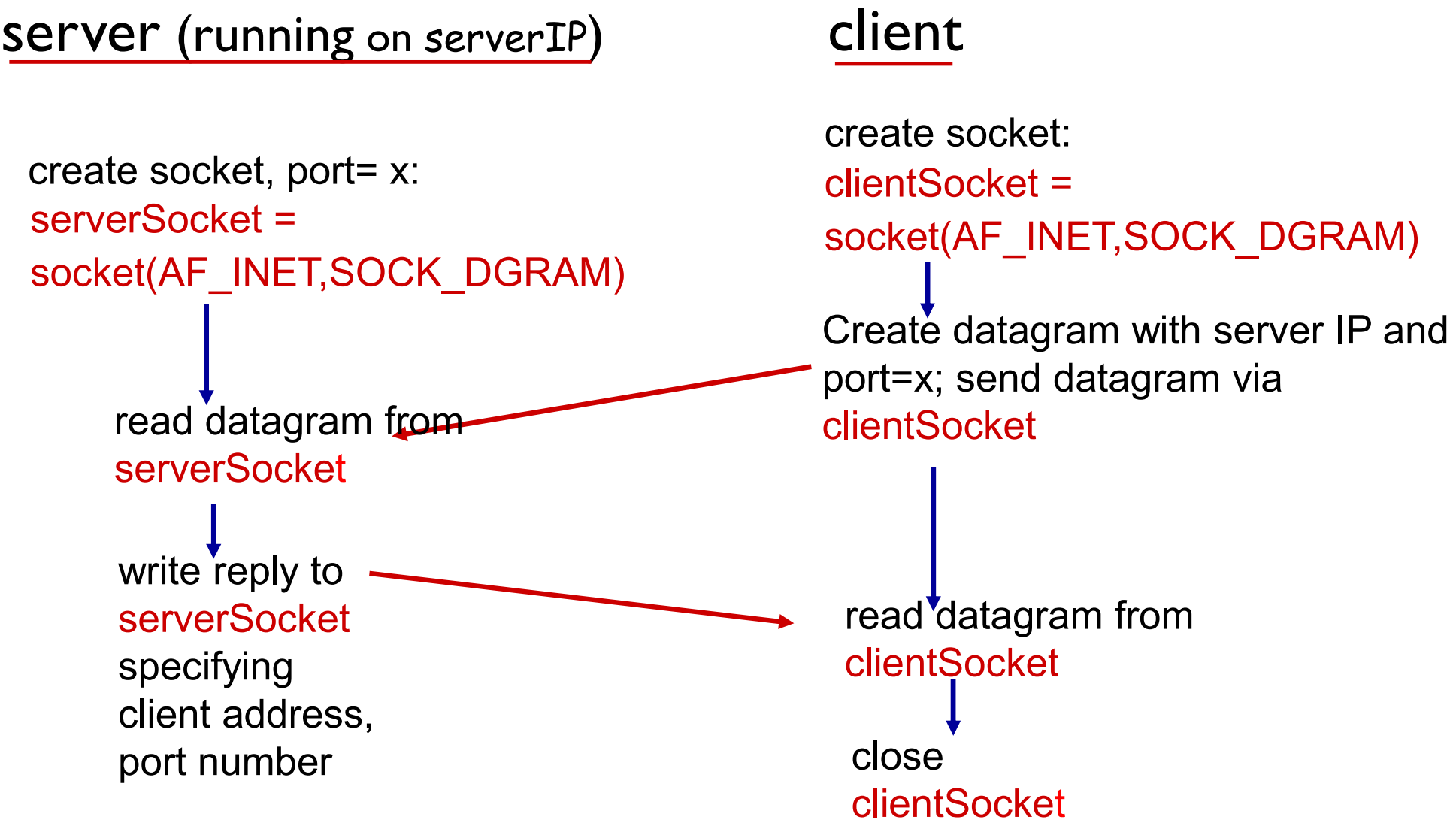
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



Example app: UDP client

Python UDPClient

include Python's socket library

```
from socket import *
```

```
serverName = 'hostname'
```

```
serverPort = 12000
```

create UDP socket for server

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

get user keyboard input

```
message = raw_input('Input lowercase sentence:')
```

Attach server name, port to message; send into socket

```
clientSocket.sendto(message,(serverName, serverPort))
```

read reply characters from socket into string

```
modifiedMessage, serverAddress =
                      clientSocket.recvfrom(2048)
```

print out received string and close socket

```
print modifiedMessage
```

```
clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

create UDP socket

bind socket to local port number 12000

loop forever

Read from UDP socket into message, getting client's address (client IP and port)

send upper case string back to this client

Socket programming *with TCP*

client must contact server

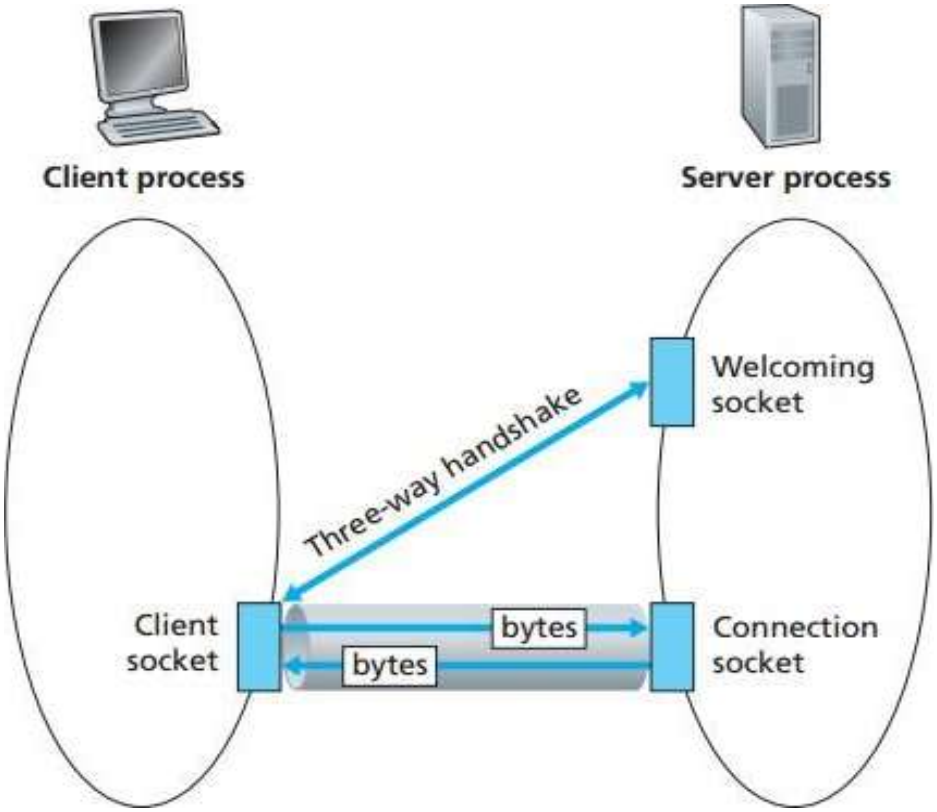
- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

Socket programming *with TCP*

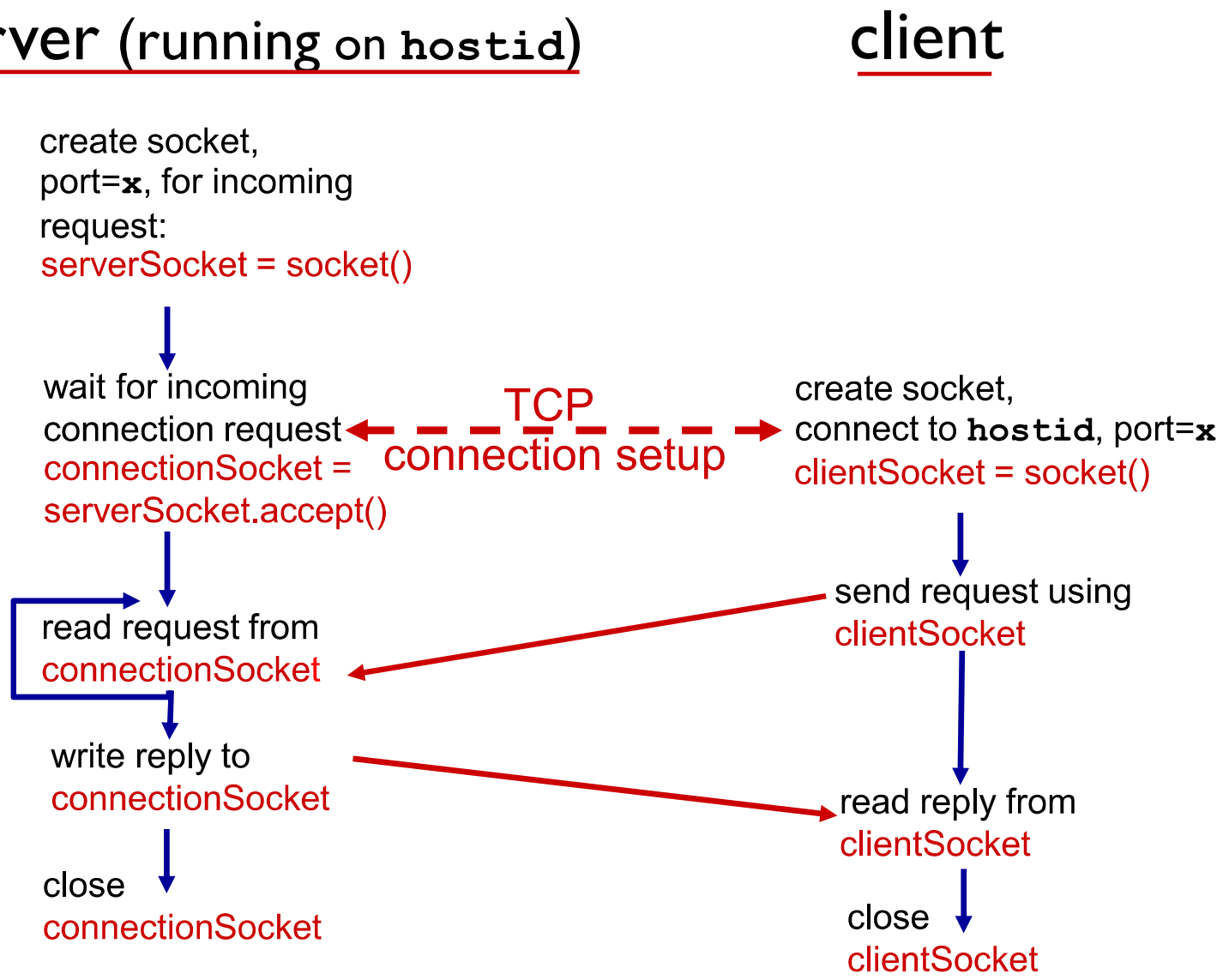
- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (Chap 3)
  - welcoming door is a TCP socket object called serverSocket
  - newly created socket called connectionSocket.



application viewpoint:

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

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

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

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

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

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

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

Chapter 2: summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, DHT
- socket programming: TCP, UDP sockets

Chapter 2: summary

most importantly: learned about protocols!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated

important themes:

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