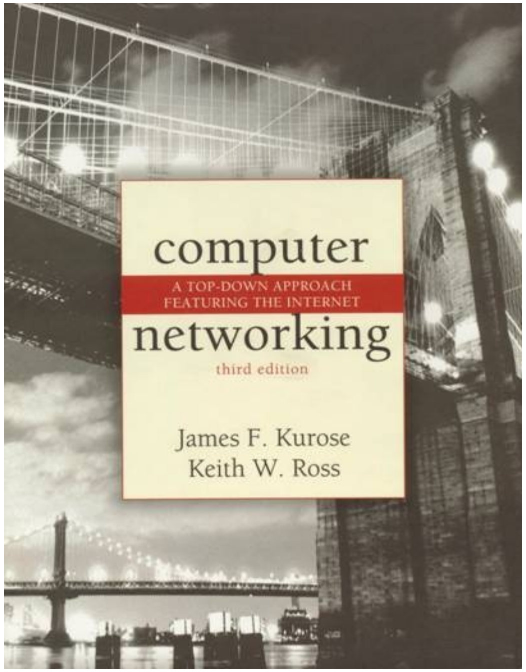
# Chapter 2
# Application Layer




*Computer Networking: A Top Down Approach Featuring the Internet,*

3rd edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2004

## 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

# 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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# Chapter 2: Application Layer

**Our goals:**

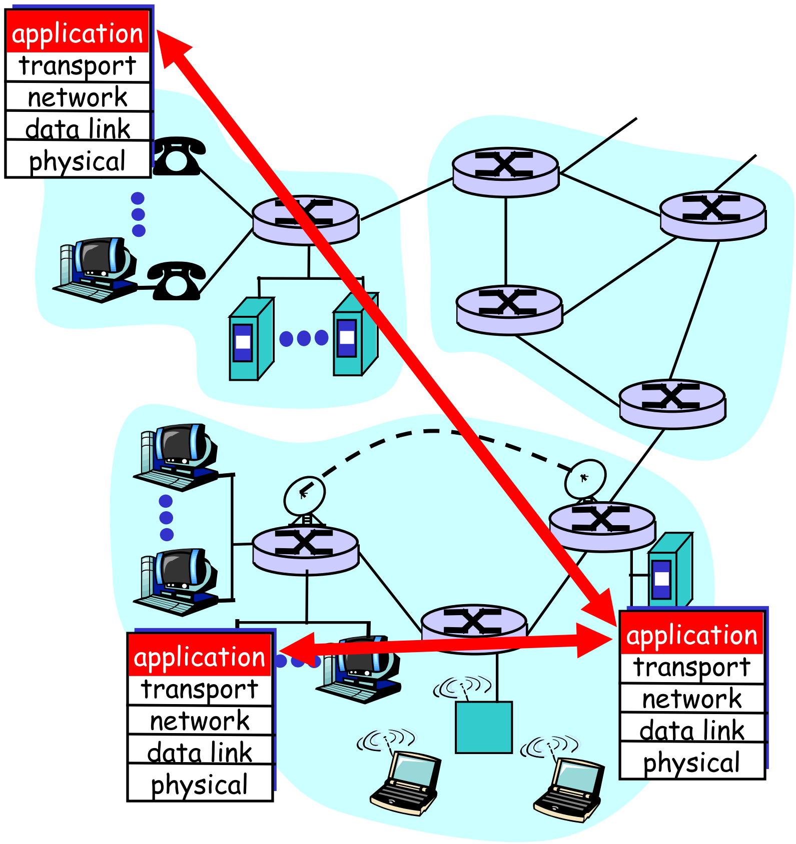
- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- programming network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

# Some network apps

- E-mail
- Web
- Instant messaging
- Remote login
- P2P file sharing
- Multi-user network games
- Streaming stored video clips
- Internet telephone
- Real-time video conference
- Massive parallel computing

# Creating a network app

- Write programs that
  - run on different end systems and
  - communicate over a network.
  - e.g., Web: Web server software communicates with browser software
- No software written for devices in network core
  - Network core devices do not function at app layer
  - This design allows for rapid app development



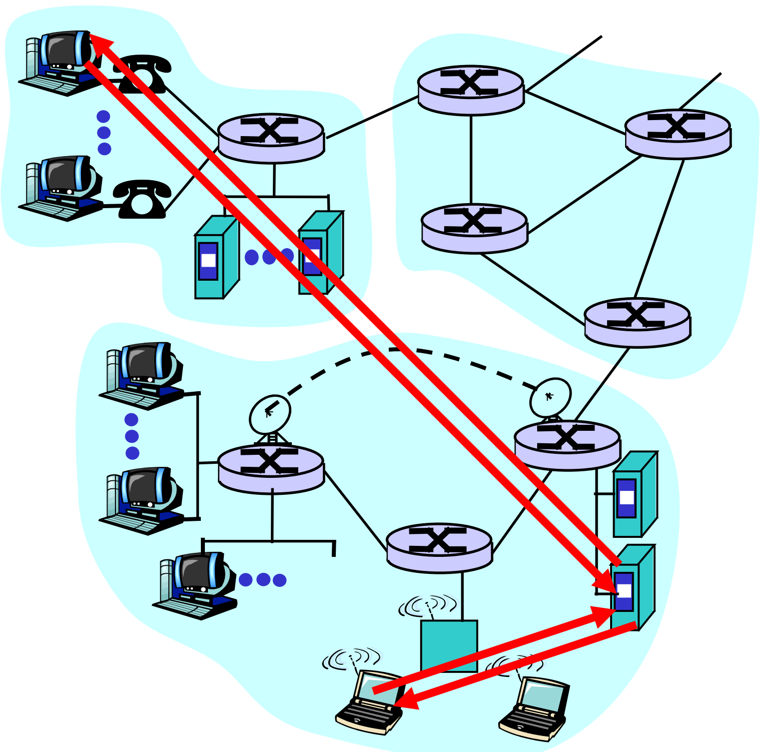
# Chapter 2: Application layer

- 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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# Application architectures

- Client-server
- Peer-to-peer (P2P)
- Hybrid of client-server and P2P

# Client-server archicture



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

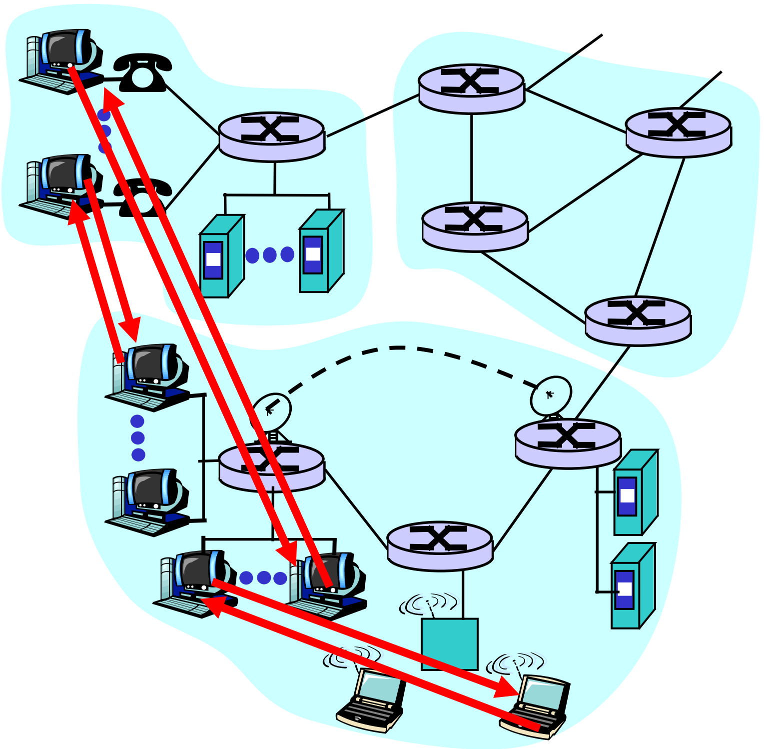
clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

# Pure P2P architecture

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

- Highly scalable

- But difficult to manage



# Hybrid of client-server and P2P

## Napster

- File transfer P2P
- File search centralized:
  - Peers register content at central server
  - Peers query same central server to locate content

## Instant messaging

- Chatting between two users is P2P
- Presence detection/location centralized:
  - 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

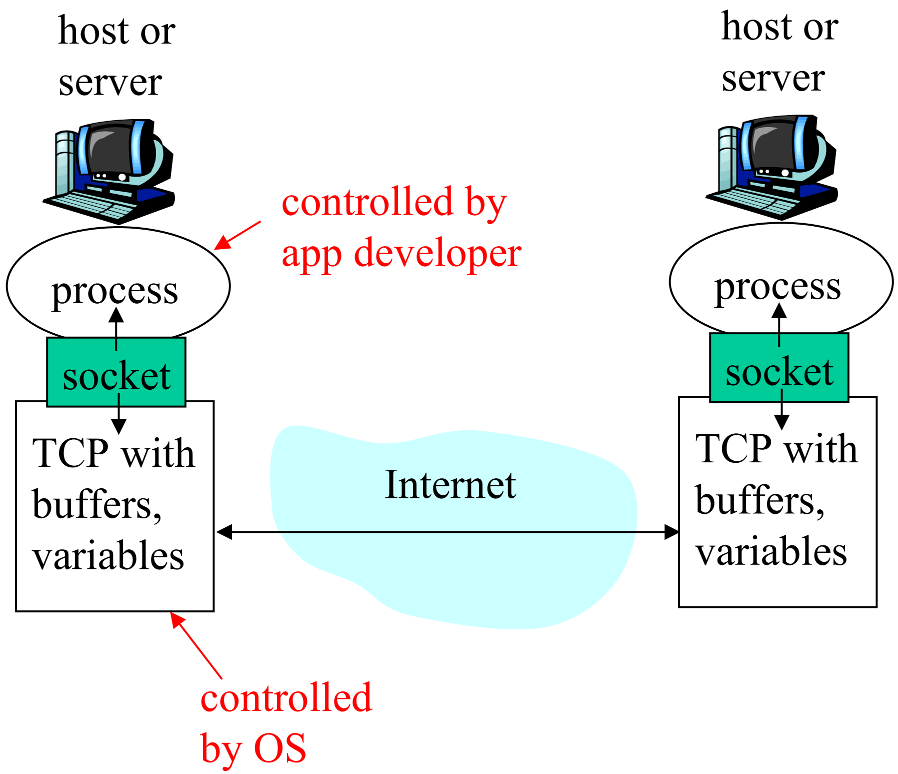
Client process: process that initiates communication

Server process: process that waits to be contacted

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

# Processes communicating across network

- 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




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

# Addressing processes:

- For a process to receive messages, it must have an identifier
- Every host has a unique 32-bit IP address
- Q: does the IP address of the host on which the process runs suffice for identifying the process?
- Answer: No, many processes can be running on same host
- Identifier includes both the IP address and port numbers associated with the process on the host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- More on this later

# App-layer protocol defines

- Types of messages exchanged, eg, request & response messages
- Syntax of message types: what fields in messages & how fields are delineated
- Semantics of the fields, ie, meaning of information in fields
- Rules for when and how processes send & respond to messages

Public-domain protocols:

- defined in RFCs
- allows for interoperability
- eg, HTTP, SMTP

Proprietary protocols:

- eg, KaZaA

# 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"

**Bandwidth**
- some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
- other apps ("elastic apps") make use of whatever bandwidth they get

# Transport service requirements of common apps

| Application | Data loss | Bandwidth | 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<br>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 providing:* timing, minimum bandwidth guarantees

## UDP service:

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

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 | proprietary (e.g. RealNetworks) | TCP or UDP |
| Internet telephony | proprietary (e.g., Dialpad) | typically UDP |

# Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# Web and HTTP

First some jargon

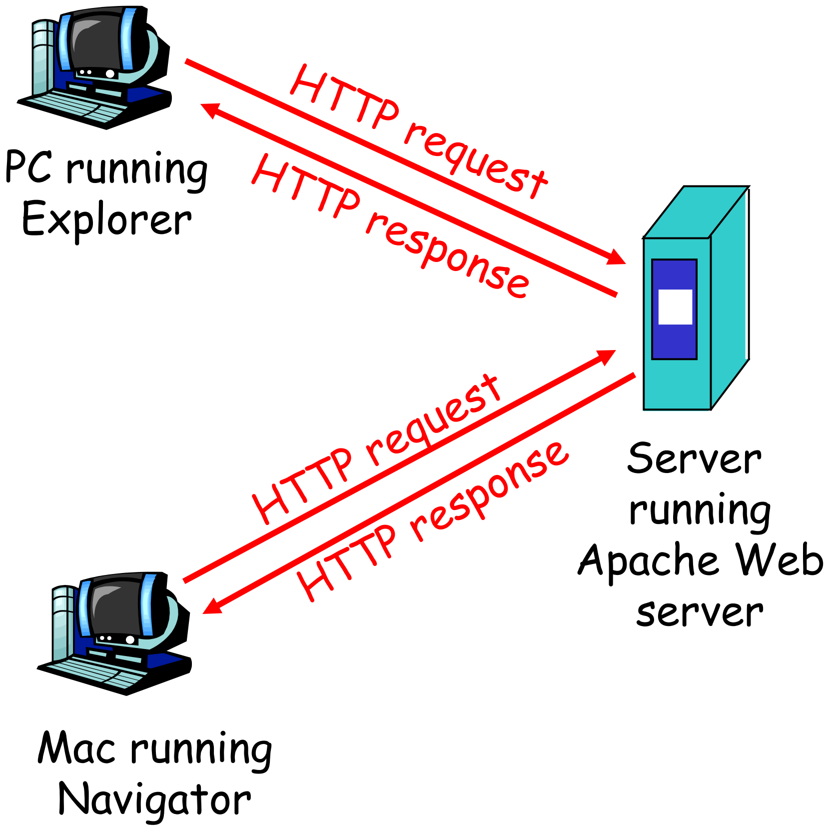
- Web page consists of objects
- An object is a file such as an HTML file, a JPEG image, a Java applet, an audio file,…
- A Web page consists of a base HTML-file and several referenced objects
- The base HTML file references the other objects in the page with the object's URLs (Uniform Resource Locators)
- Example URL:

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

# 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
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2616



# 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.
- HTTP/1.0 uses nonpersistent HTTP

**Persistent HTTP**

- Multiple objects can be sent over single TCP connection between client and server.
- A new connection need not be set up for the transfer of each Web object
- HTTP/1.1 uses persistent connections in default mode – can be configured to use nonpersistent connection

# 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.)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 references to the 10 jpeg objects

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

time

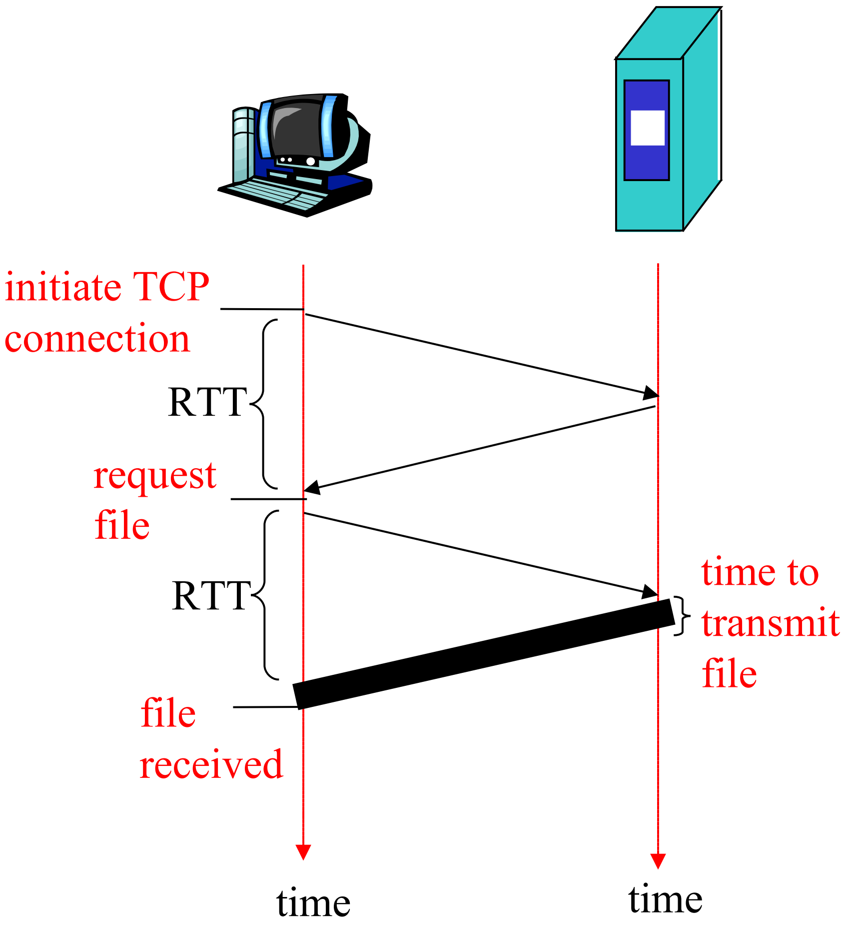
# Response time modeling

Definition of RTT: time to send 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



# Persistent HTTP

**Nonpersistent HTTP issues:**

- requires 2 RTTs per object
- OS must work and allocate host resources for each TCP connection
- but 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 are sent over connection

**Two versions of persistent connections:**

**Persistent without pipelining:**

- client issues new request only when previous response has been received
- one RTT for each referenced object

**Persistent with pipelining:**

- default in HTTP/1.1
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

# HTTP request message

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

request line (GET, POST, HEAD commands)

header lines

Carriage return, line feed indicates end of message

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

(extra carriage return, line feed)

# Explanation of the example

**`GET /somedir/page.html HTTP/1.1`**

-- Request to return the object `/somedir/page.html`

-- The browser implements version HTTP/1.1

**`Host: www.someschool.edu`**

-- Specifies the host on which the object resides

**`User-agent: Mozilla/4.0`**

-- Specifies the browser type that is making the request

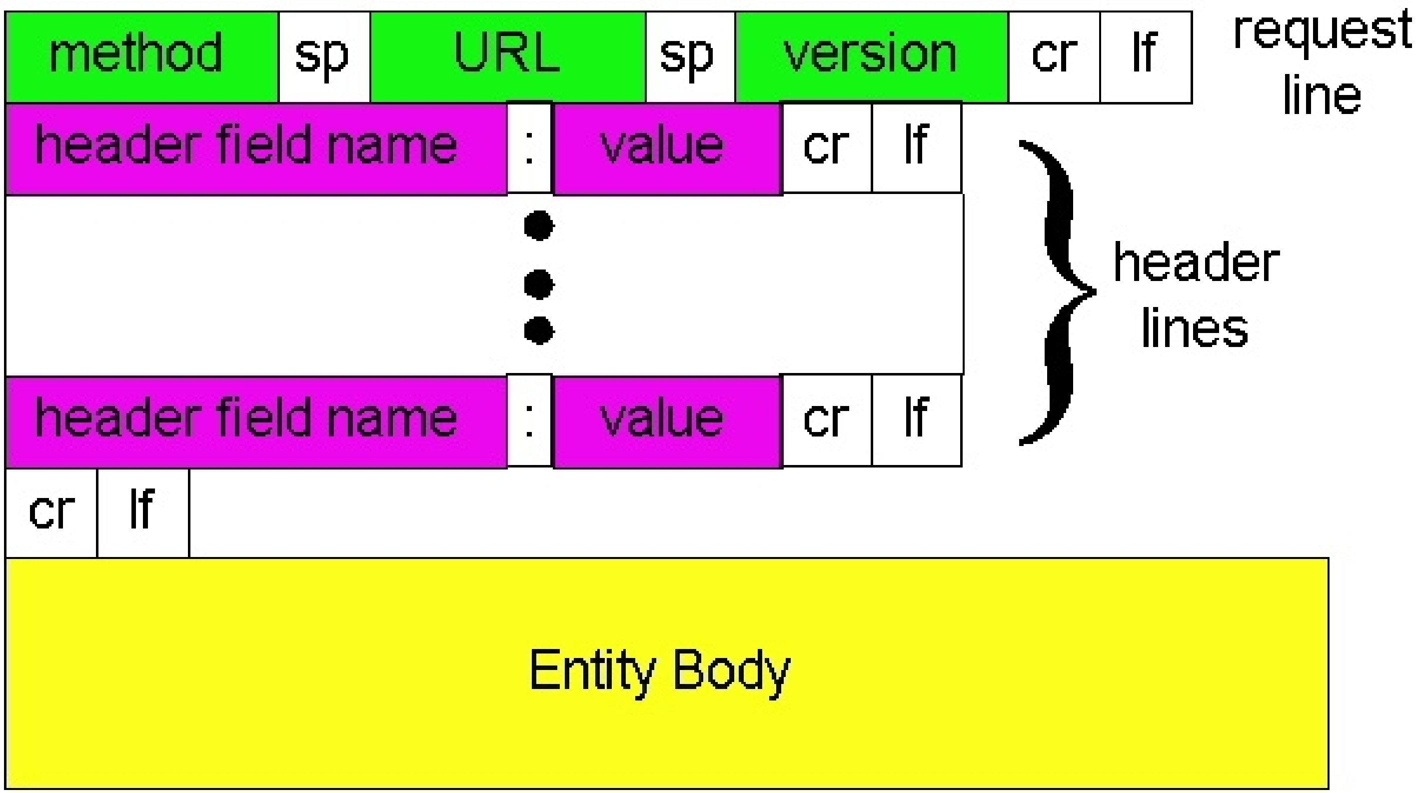
**`Connection: close`**

-- Indicates that the connection SHOULD NOT be considered `persistent'. It wants the server to close the connection after the current request/response is complete

**`Accept-language:fr`**

-- Indicates that the user prefers to receive a French version of the object

# HTTP request message: general format



# Method types

## HTTP/1.0

- GET : Return the object
- POST : Send information to be stored on the server
- HEAD
  - Return only information about the object, such as how old it is, but not the object itself

## HTTP/1.1

- GET, POST, HEAD
- PUT
  - Uploads a new copy of existing object in entity body to path specified in URL field
- DELETE
  - deletes object specified in the URL field

# 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
```

# HTTP response message

An HTTP response consists of the following:

1. A status line, which indicates the success or failure of the request
2. Header lines: A description of the information in the response. This is the metadata or meta information
3. The actual information requested

status line
(protocol status code status phrase)

header lines

blank line

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 www.eurecom.fr 80
```

Opens TCP connection to port 80 (default HTTP server port) at www.eurecom.fr. Anything typed is sent to port 80 at www.eurecom.fr

2. Type in a GET HTTP request:

```
GET /~ross/index.html HTTP/1.0
```

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!

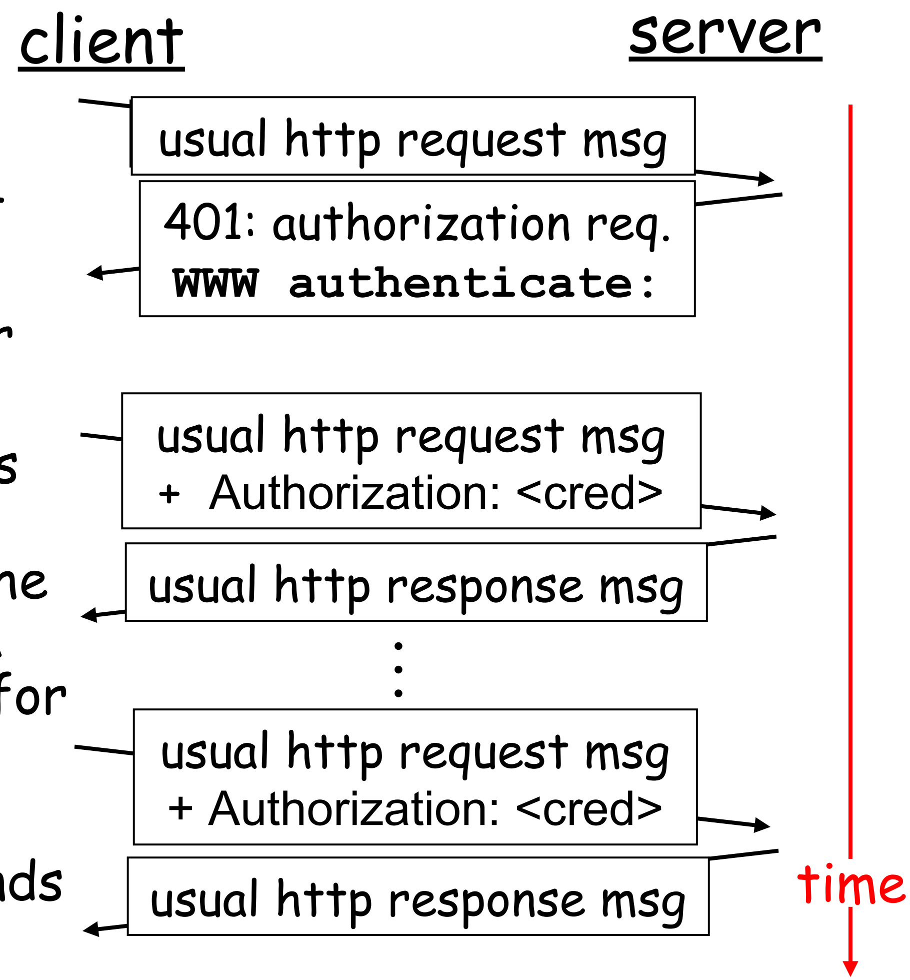
# User-server interaction: authorization

**Authorization :** control access to server content

- authorization credentials: typically name, password
- **stateless:** client must present authorization in *each* request
  - Includes authorization: header line in each request
  - While the browser remains open, the username and password are cached, so the user is not prompted for a username and a password for each request
  - if no authorization: header, server refuses access, sends

    **WWW authenticate:**

    header line in response



# User-server state: cookies

Many major Web sites use cookies

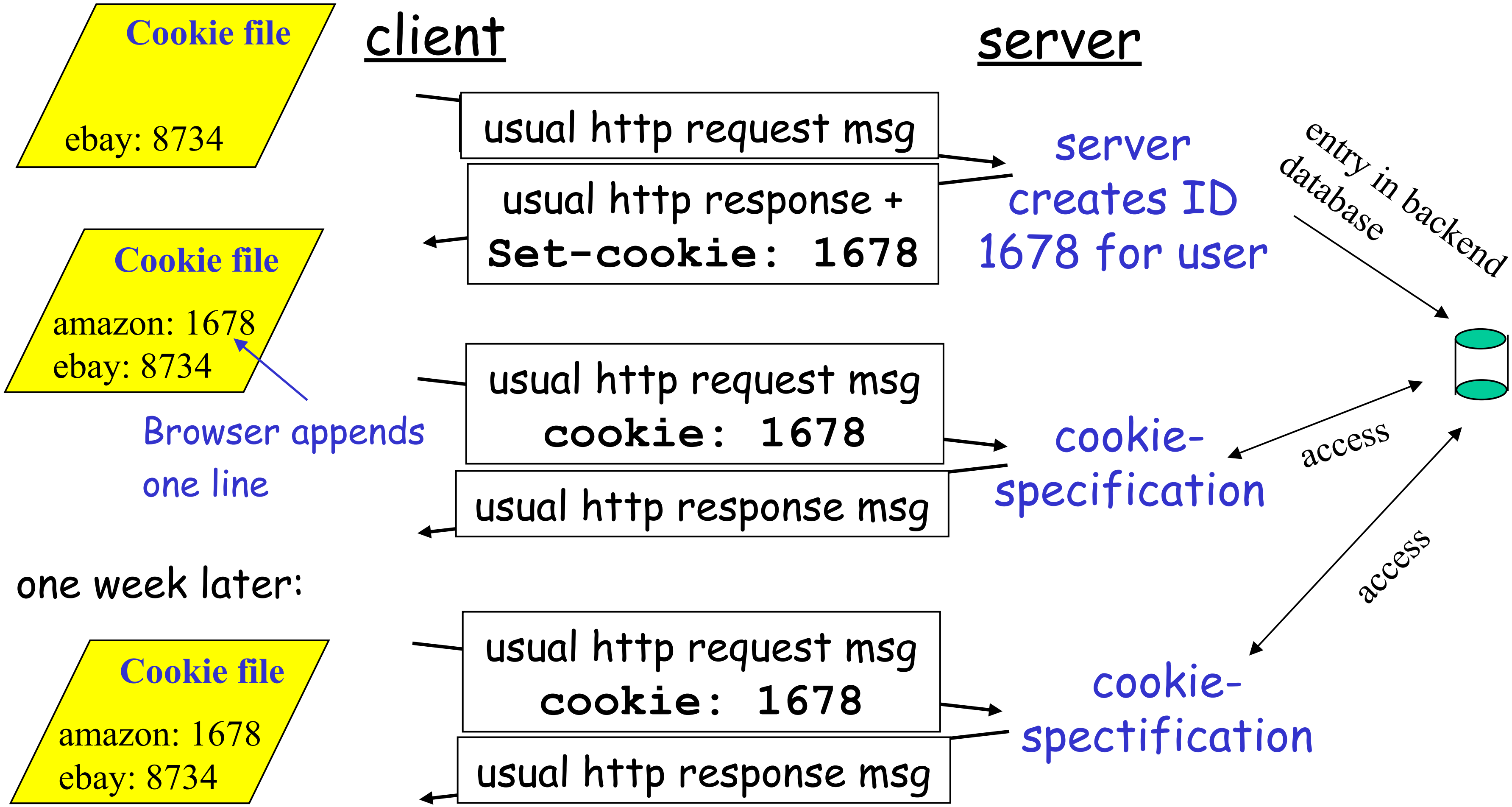
Four components of cookie technology:

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

Example:

- Susan access Internet always from same PC
- She visits a specific e-commerce site for first time
- When initial HTTP requests arrives at site, site creates a unique ID and creates an entry in backend database for ID

# Cookies: keeping "state" (cont.)



# Cookies (continued)

**What cookies can bring:**

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

aside

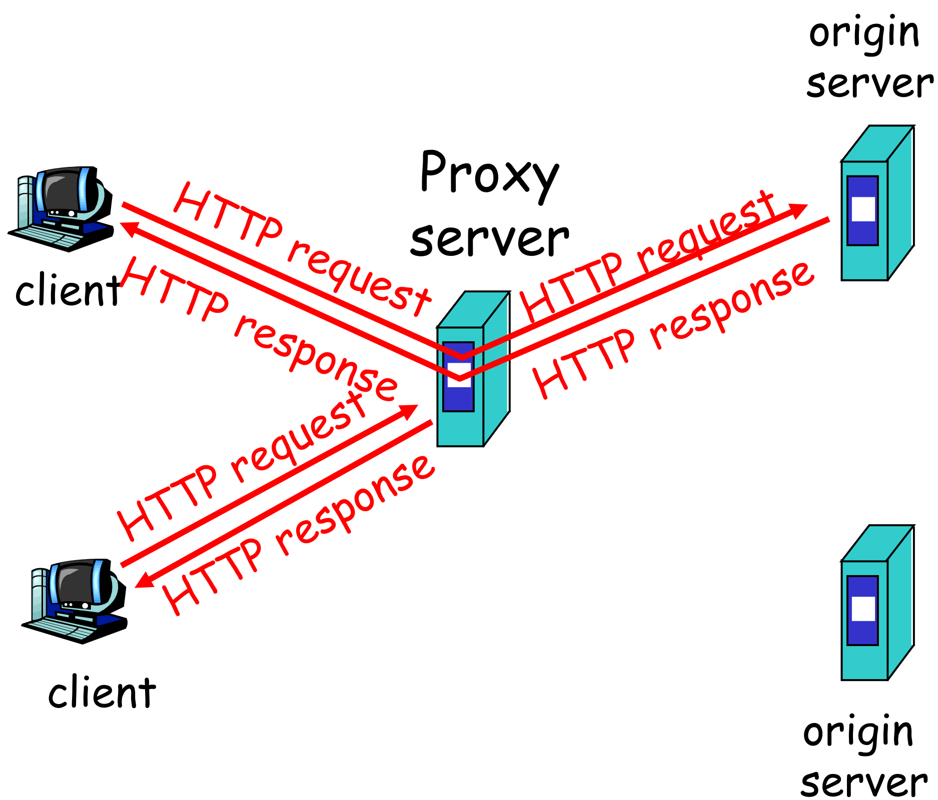
**Cookies and privacy:**

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites
- search engines use redirection & cookies to learn yet more
- advertising companies obtain info across 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



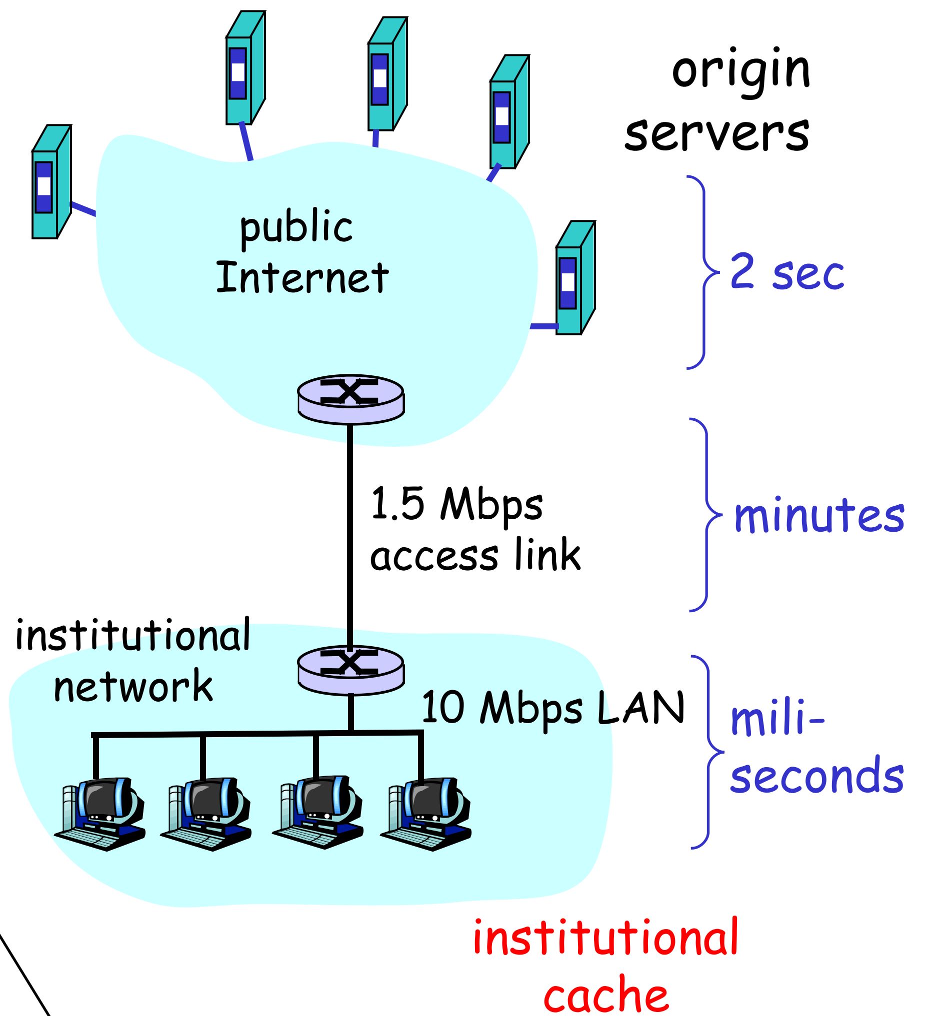
# More about Web caching

- 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

(100,000 bits x 15/sec )/1.5 Mbps

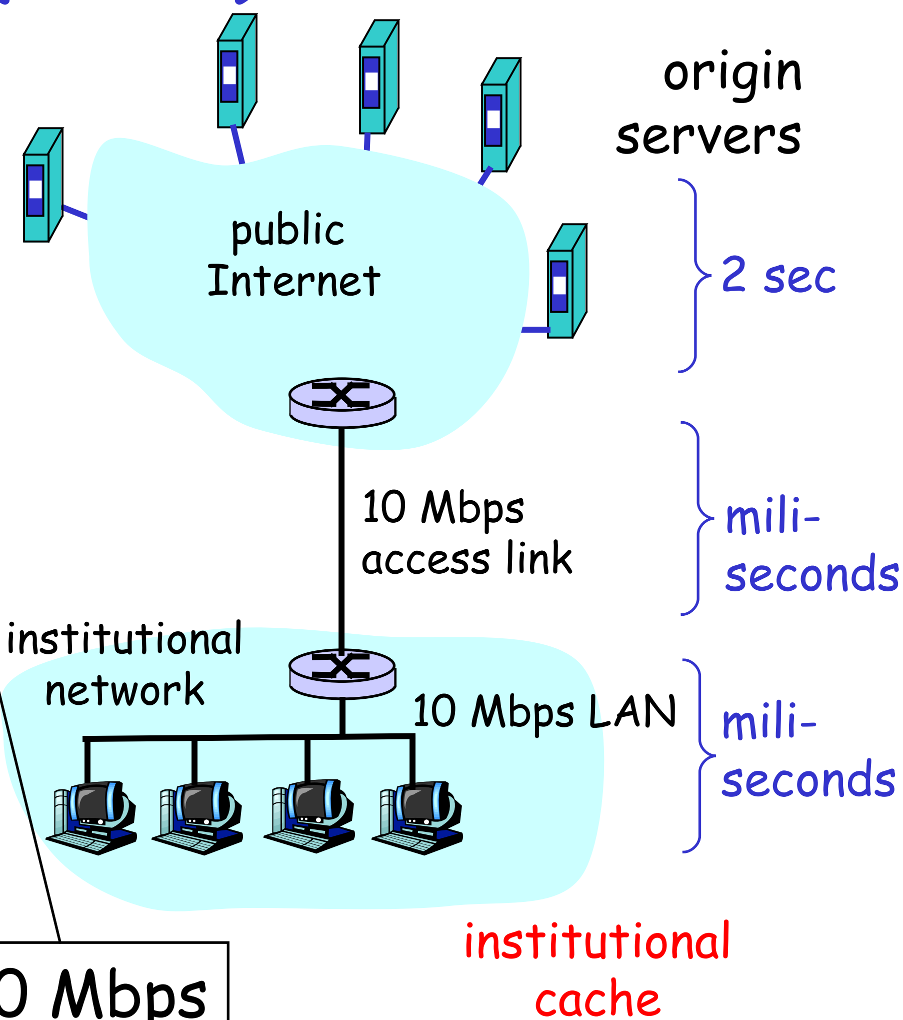
# Caching example (cont)

<u>Possible solution</u>

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

<u>Consequences</u>

- 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




(100,000 bits x 15/sec )/10 Mbps

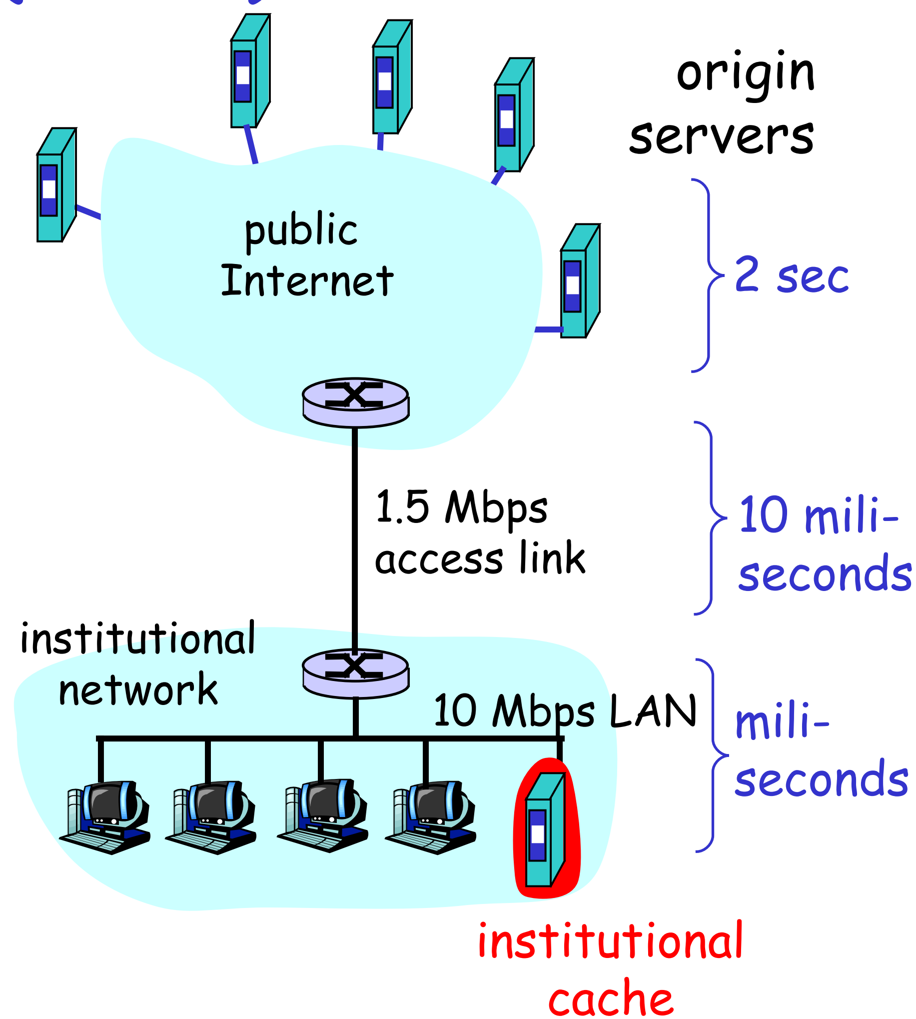
# Caching example (cont)

Install cache

- suppose hit rate is .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 + milliseconds < 1.4 sec

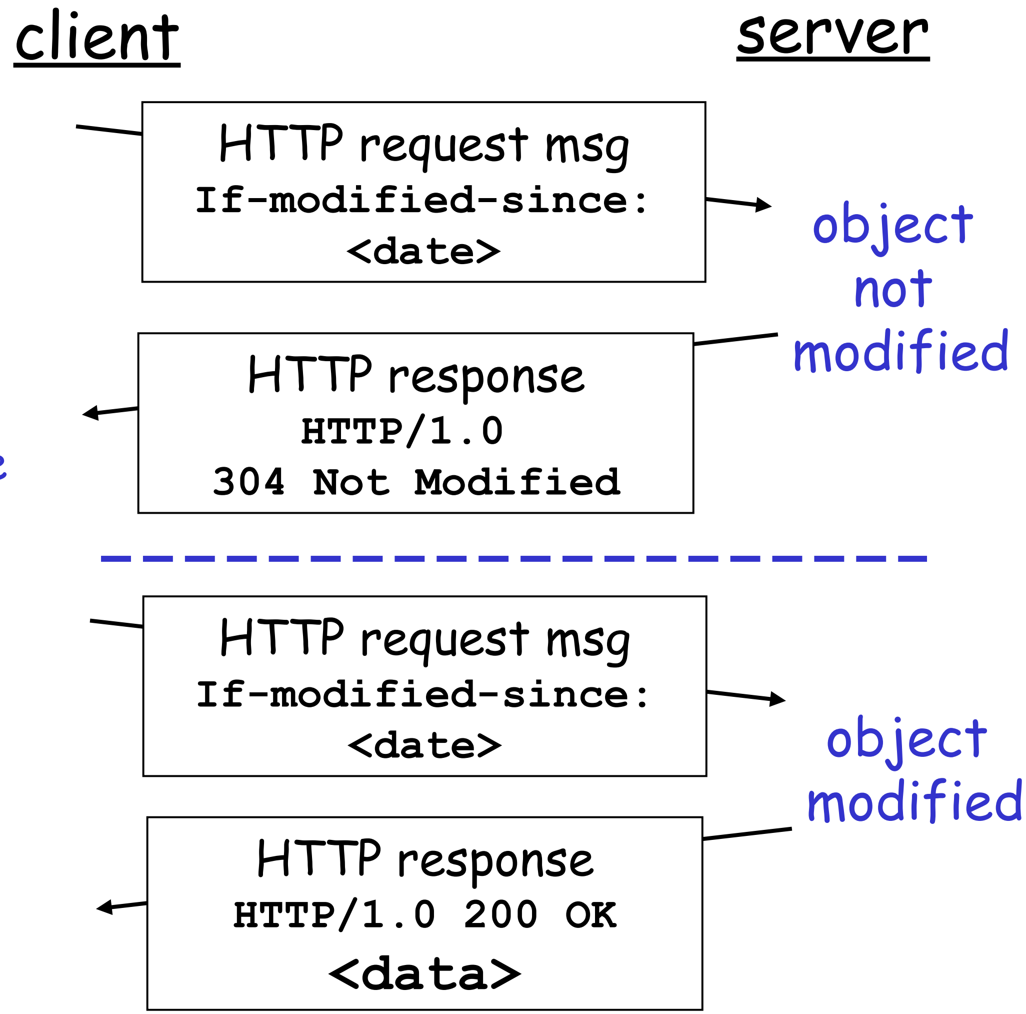


# Conditional GET: client-side caching

- **Goal:** don't send object if client has up-to-date cached version
- client: specify date of cached copy in HTTP request

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

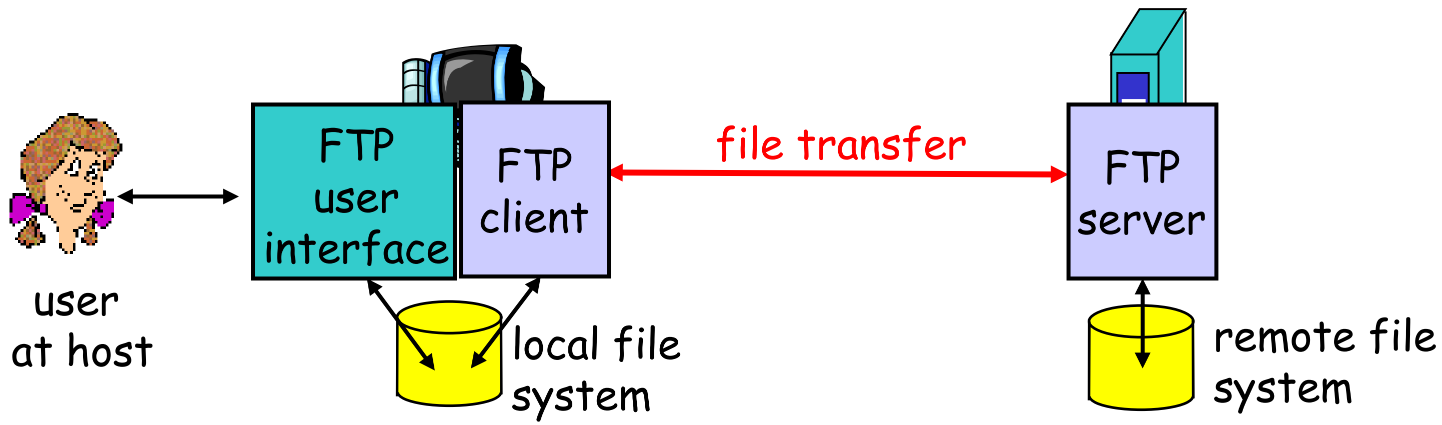
  **`HTTP/1.0 304 Not Modified`**



# 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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

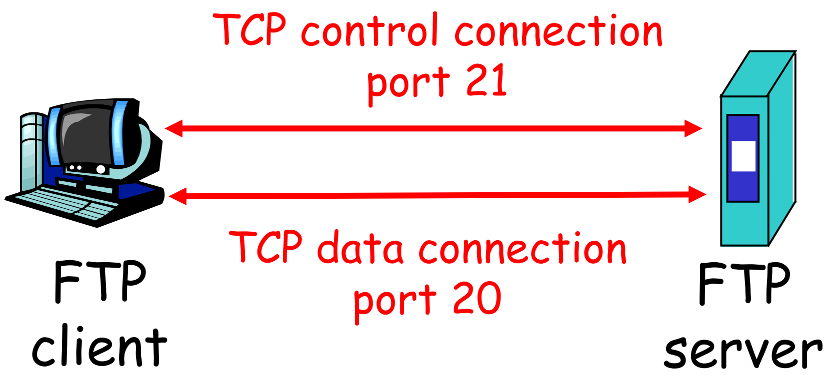
# FTP: the file transfer protocol




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

# FTP: separate control, data connections

- FTP client contacts FTP server at port 21, specifying TCP as transport protocol
- Client obtains authorization over control connection -- username, password
- Client browses remote directory by sending commands over control connection.
- When server receives a command for a file transfer, the server opens a TCP data connection to client
- After transferring one file, server closes connection.




- Server opens a second 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

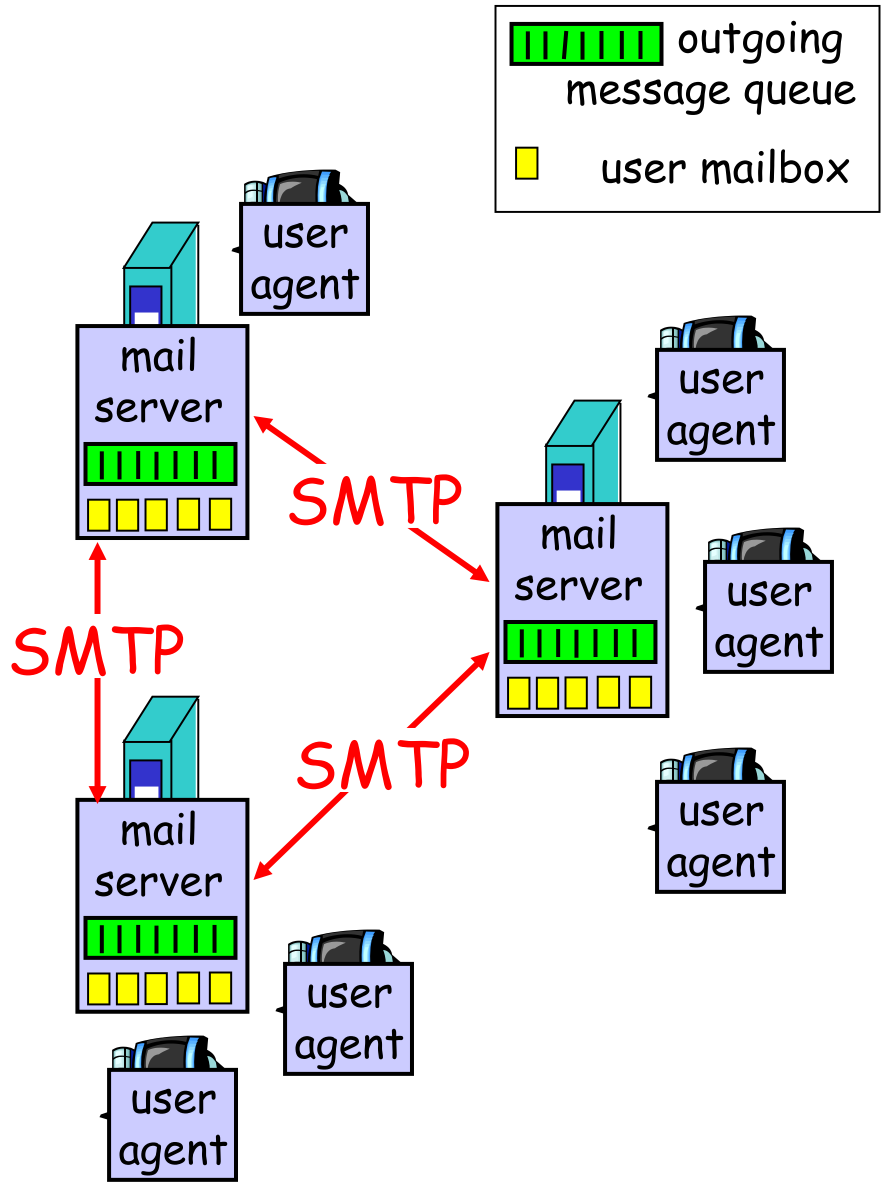
status code and phrase (as in HTTP)

- **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

- 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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# Electronic Mail



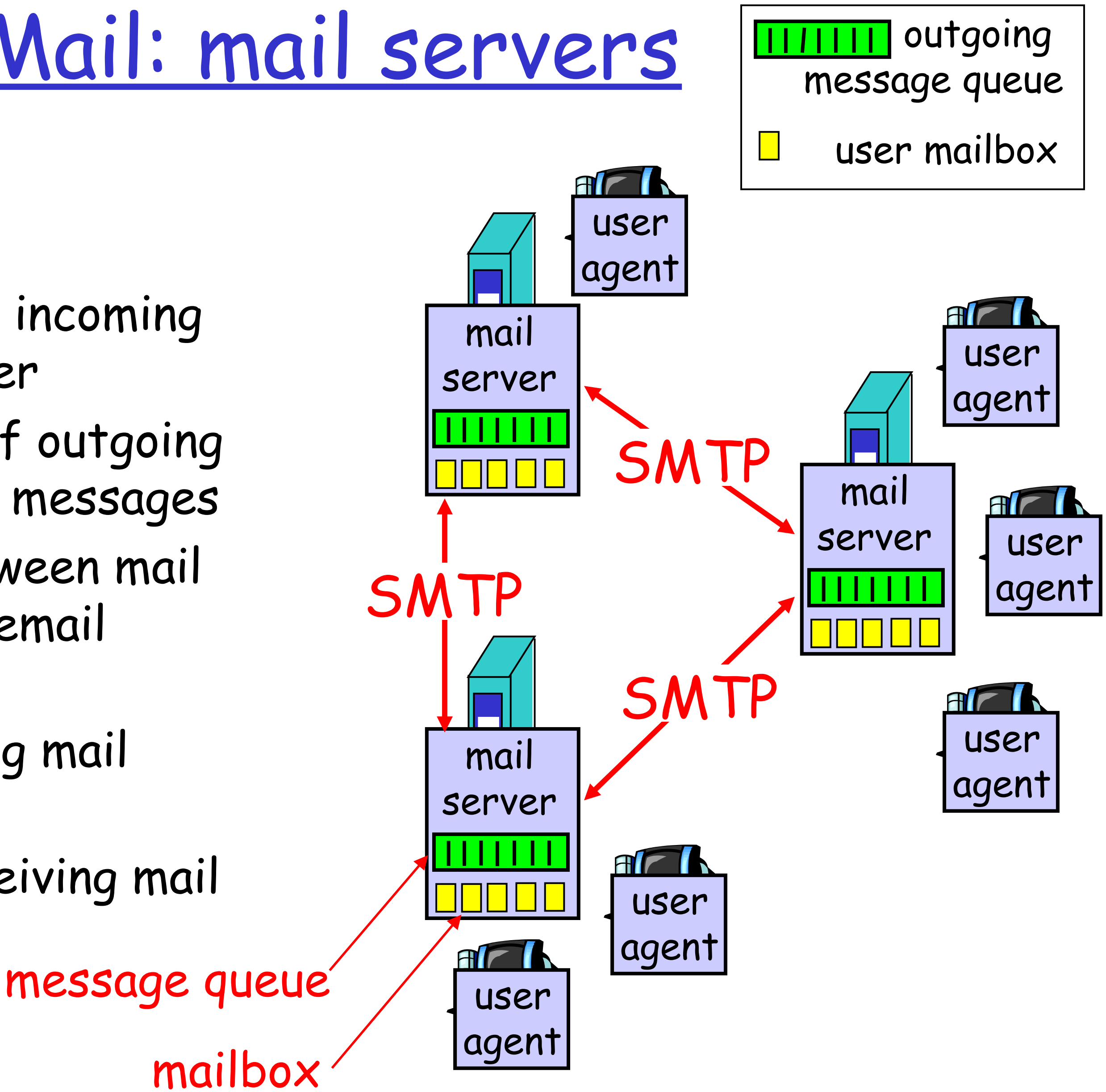

Three major components of a mail system:

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

User Agent

- Also known as "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- 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

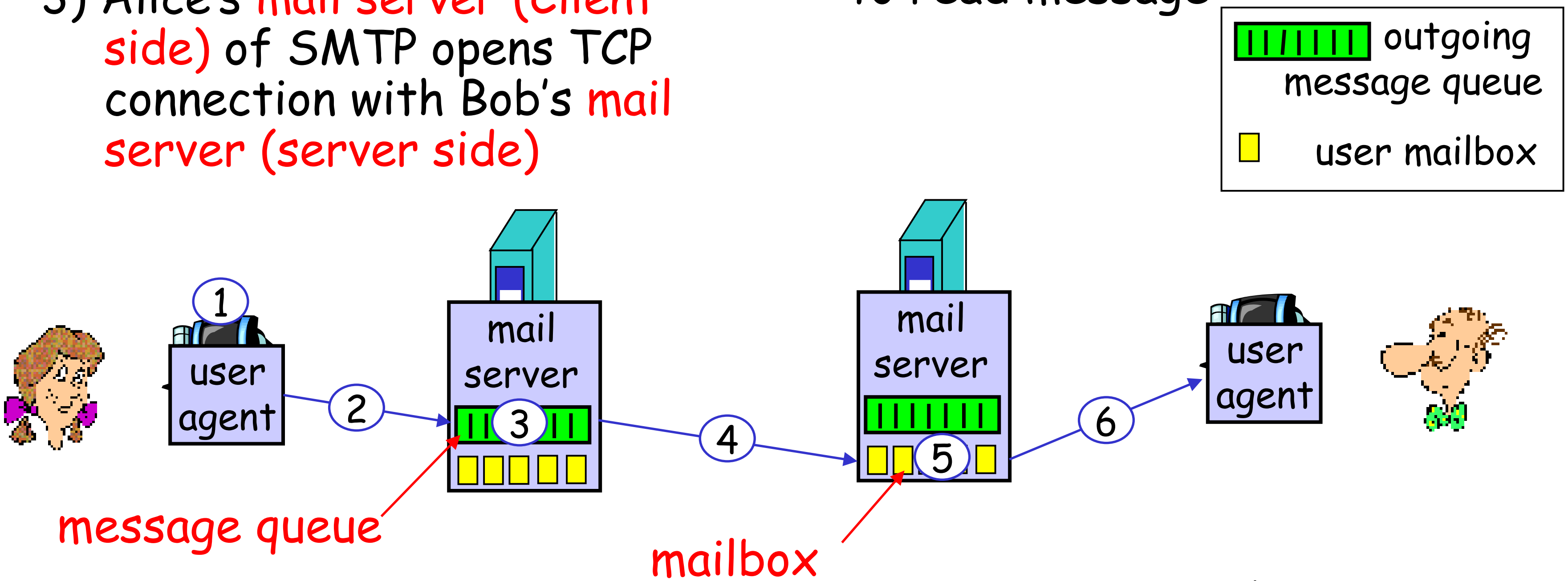
# Electronic Mail: SMTP [RFC 2821]

Simple Mail Transfer Protocol (SMTP)

- 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 user agent to compose message and "to" `bob@someschool.edu`
2) Alice's user agent sends message to her mail server; message placed in message queue
3) Alice's mail server (Client side) of SMTP opens TCP connection with Bob's mail server (server side)
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

The following transcript begins as soon as the TCP connection is established:

server

client

```
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 protocol (client's point of view)
- SMTP: push protocol
- both have ASCII command/ response interaction, status codes
- HTTP does not require message to be in 7-bit ASCII
- HTTP: one object in one response message
- SMTP: multiple objects can be sent in one message

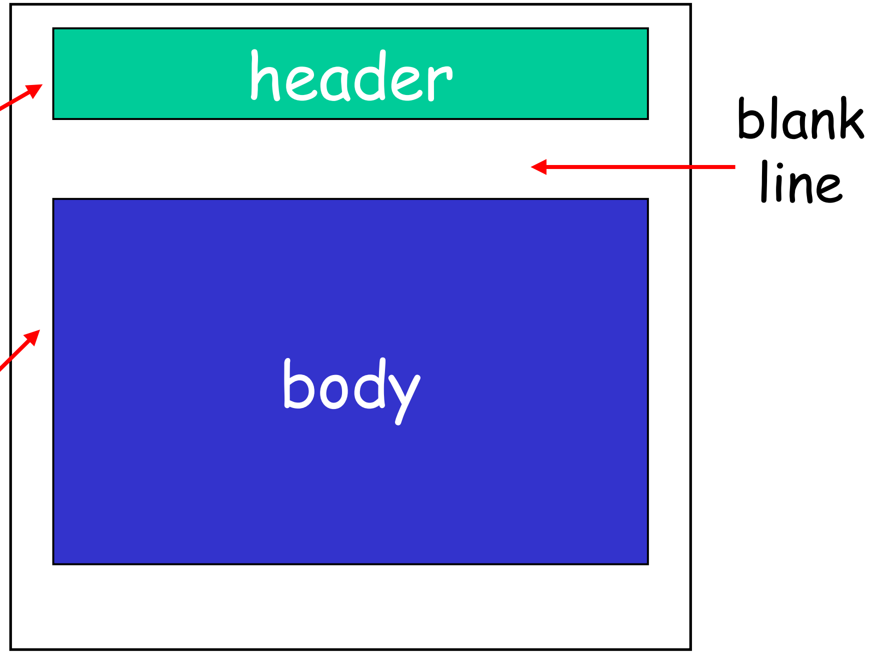
# Mail message format

SMTP: protocol for exchanging email messages

RFC 822: standard for text message format:

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

  *different from SMTP commands!*
- body
  - the “message”, ASCII characters only



# Message format: multimedia extensions

- MIME (Multipurpose Internet Mail extensions) : multimedia mail extension, RFC 2045, 2056
- additional lines in message header declare MIME content type

MIME version

method used to encode data

multimedia data type, subtype, parameter declaration

encoded data

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
```

# MIME types

**`Content-Type: type/subtype; parameters`**

Currently, seven types are defined:

## (1) Text

- example subtypes: **`plain`**, **`html`**

## (2) Image

- example subtypes: **`jpeg`**, **`gif`**

## (3) Audio

- example subtypes: **`basic`** (8-bit mu-law encoded), **`32kadpcm`** (32 kbps Adaptive Differential Pulse Code Modulation coding)

## (4) Video

- example subtypes: **`mpeg`**, **`quicktime`**

## (5)Application

- other data that must be processed by reader before "viewable"
- example subtypes: **`msword`**, **`octet-stream`**

## (6) Multipart

- one or more different sets of data are combined in a single body
- example subtypes: mixed, alternative (alternative version of the same information)

## (7) Message

- encapsulate another mail message
- example subtypes: rfc822, partial

# Example of Multipart Type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=StartOfNextPart

--StartOfNextPart
Dear Bob, Please find a picture of a crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
base64 encoded data .....
.........................
......base64 encoded data
--StartOfNextPart
Do you want the reciple?
```

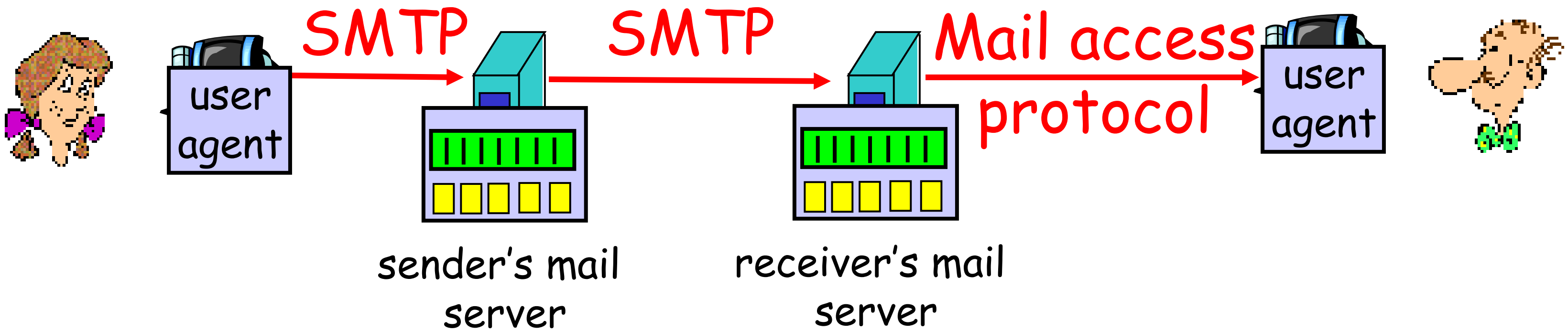
# Header line inserted by the receiving server

```
Received: from crepes.fr by hamburger.edu; 12 Oct 98
   15:27:39 GMT
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
```

inserted by the receiving server

# Mail access protocols




- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP3: Post Office Protocol, version 3 [RFC 1939]
    - authorization (agent <--> server) and download
  - IMAP: Internet Mail Access Protocol [RFC 2060]
    - more features (more complex)
    - manipulation of stored messages on server
  - HTTP: Hotmail , Yahoo! Mail, etc.

# POP3 protocol

(1) Client opens a TCP connection to the mail server on port 110

(2) authorization phase

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

(3) transaction phase, client:

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

(4) update phase : mail server deletes the messages marked for deletion

```
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
```

Message size

Message number

# 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" mode: 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

- 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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# DNS: Domain Name System

People: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., gaia.cs.umass.edu - used by humans

Q: map between IP addresses and name ?

Domain Name System:

- *A distributed database* implemented in hierarchy of many *name servers*
- *An application-layer protocol* that allows host, routers, name servers to communicate to *resolve* names (address/name translation)
  - DNS provides a core Internet function, implemented as application-layer protocol
  - DNS is an example of the Internet design philosophy of placing complexity at network's "edge"

# DNS

DNS services

- Hostname to IP address translation
- Host aliasing
  - Canonical and alias names
  - Relay1.west-coast.enterprise.com
  - enterprise.com and www.enterprise.com
- Mail server aliasing
  - bob@hotmail.com
  - Relay1.west-coast.hotmail.com
- Load distribution
  - Replicated Web servers: set of IP addresses for one canonical name

Canonical name

Alias name

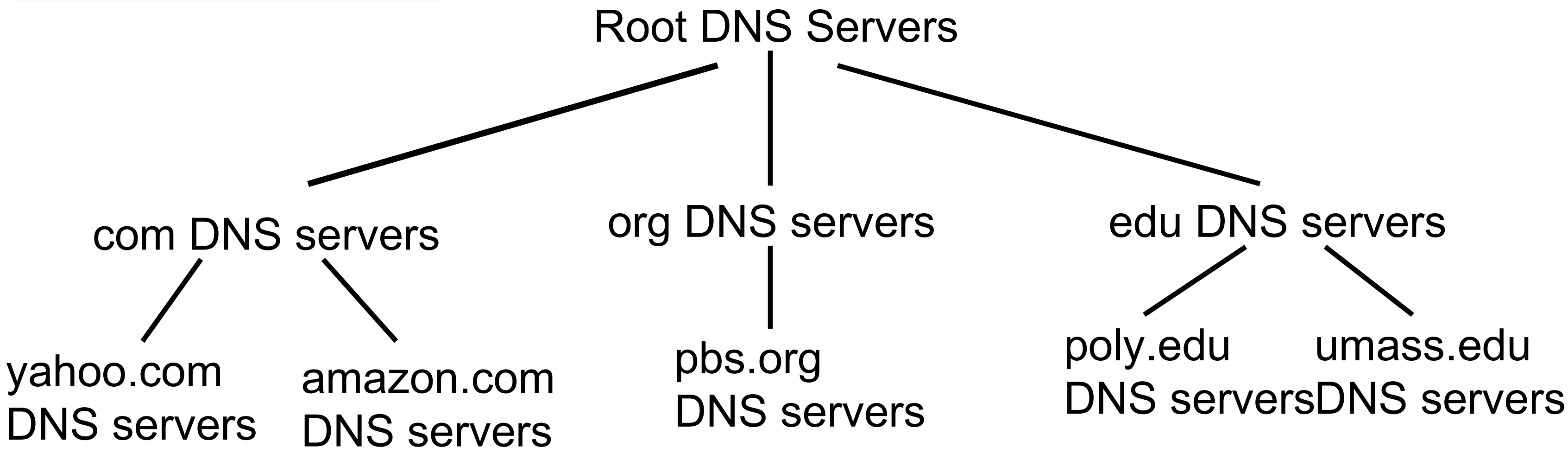
# DNS

**Why not centralize DNS?**

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

doesn't *scale!*

# Distributed, Hierarchical Database




## 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 name servers

The DNS is a distributed design

- A large number of name servers organized in a hierarchical fashion and distributed around the world
- no server has all name-to-IP address mappings

There are four types of name servers:

(1) root name servers

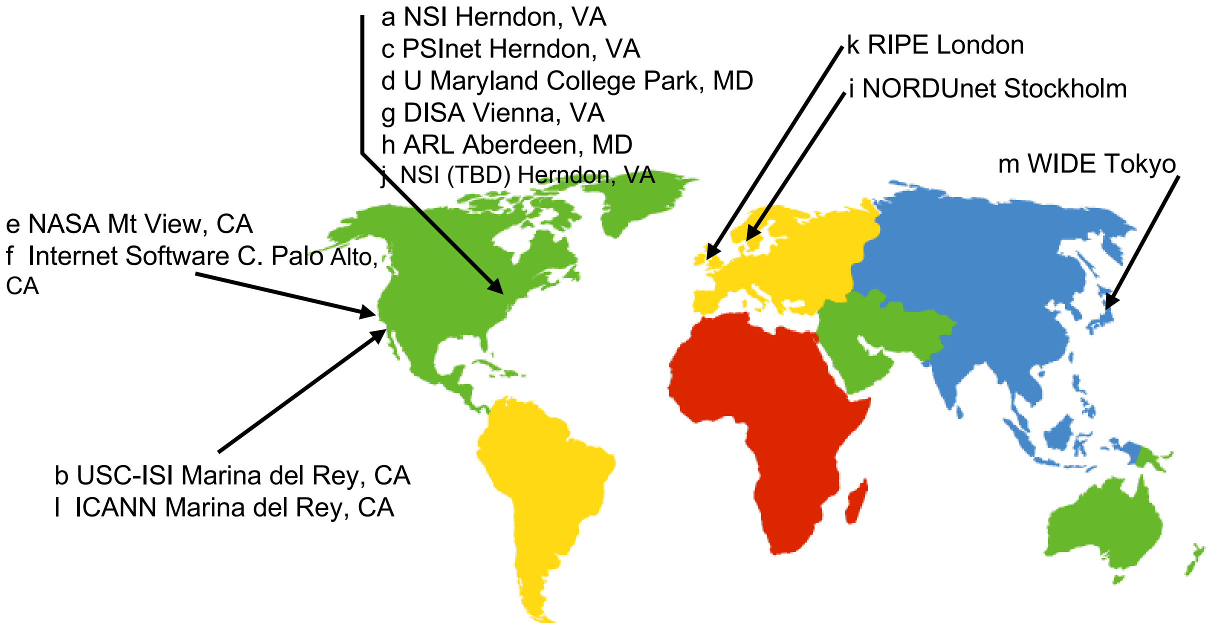
(2) top level name servers ( to be explained next)

(3) authoritative name servers:
- for a host: stores that host's IP address, name
- can perform name/address translation for that host's name

(4) local name servers:
- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

# DNS: Root name servers

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




13 root name servers worldwide

# TLD and Authoritative Servers

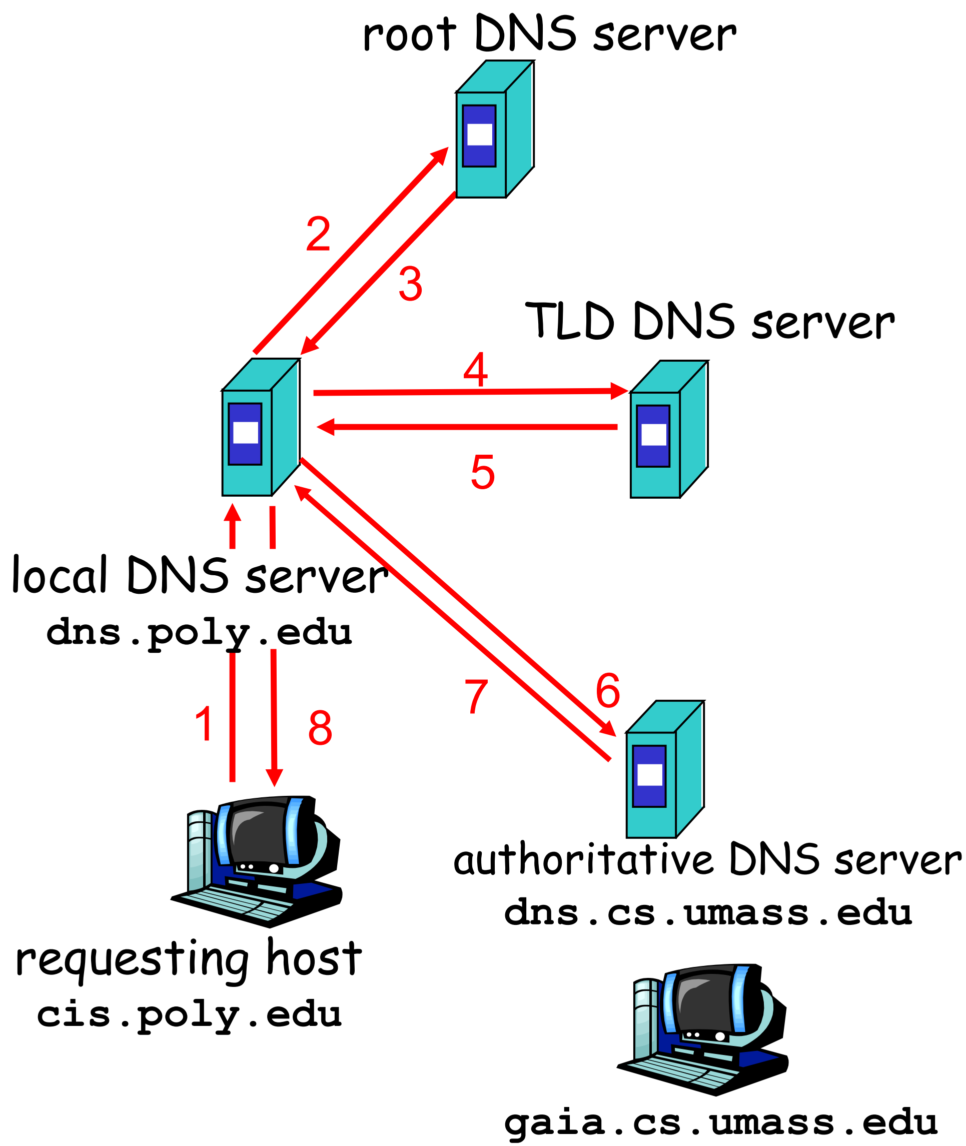
- Top-level domain (TLD) servers: responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
  - The company Network Solutions maintains servers for com TLD
  - The company Educause maintains servers for edu TLD
- Authoritative DNS servers: organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web and 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 a host makes a DNS query, query is sent to its local DNS server
  - Acts as a proxy, forwards query into hierarchy.

# Example

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



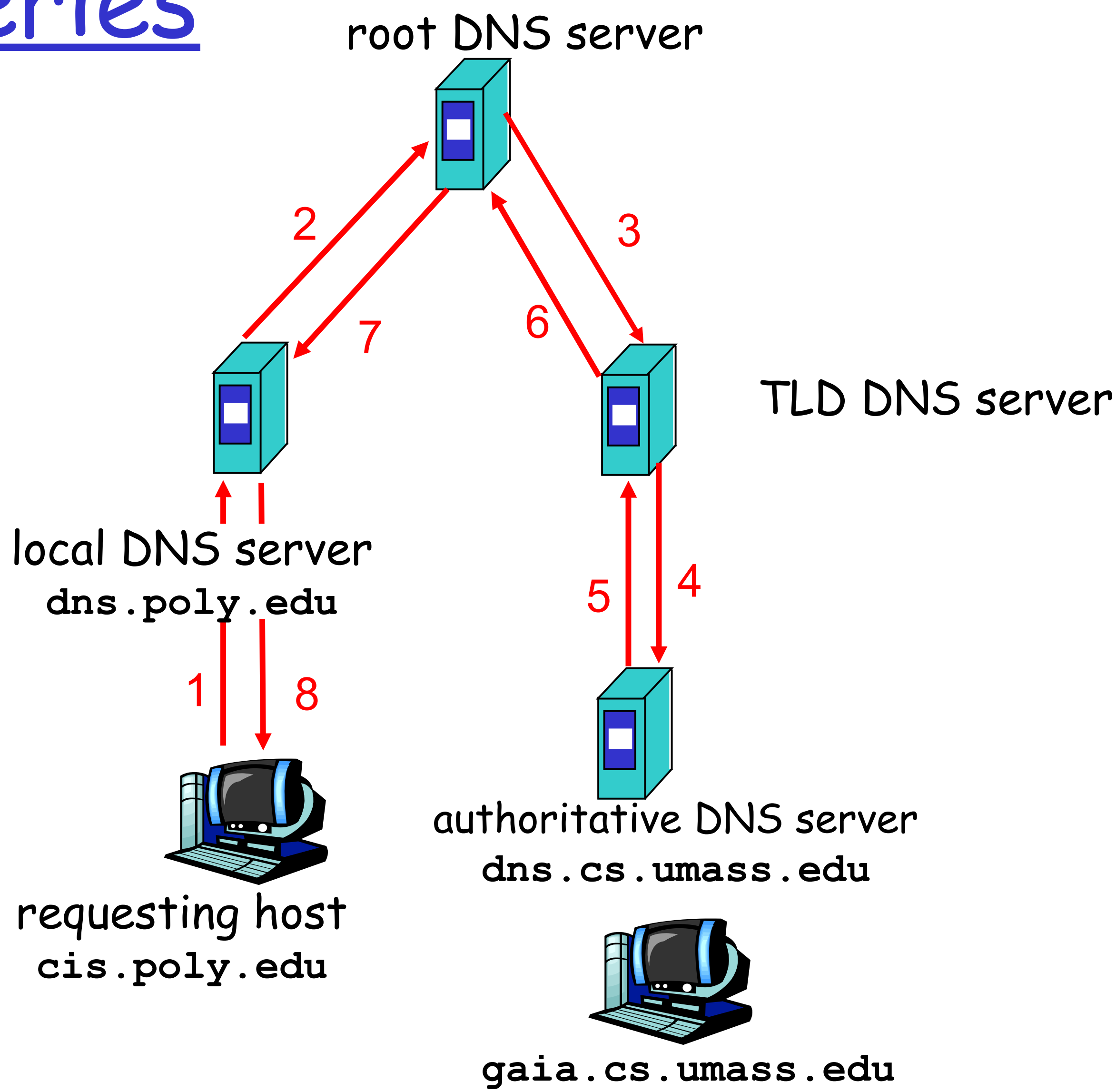
# Recursive queries

## recursive query:

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

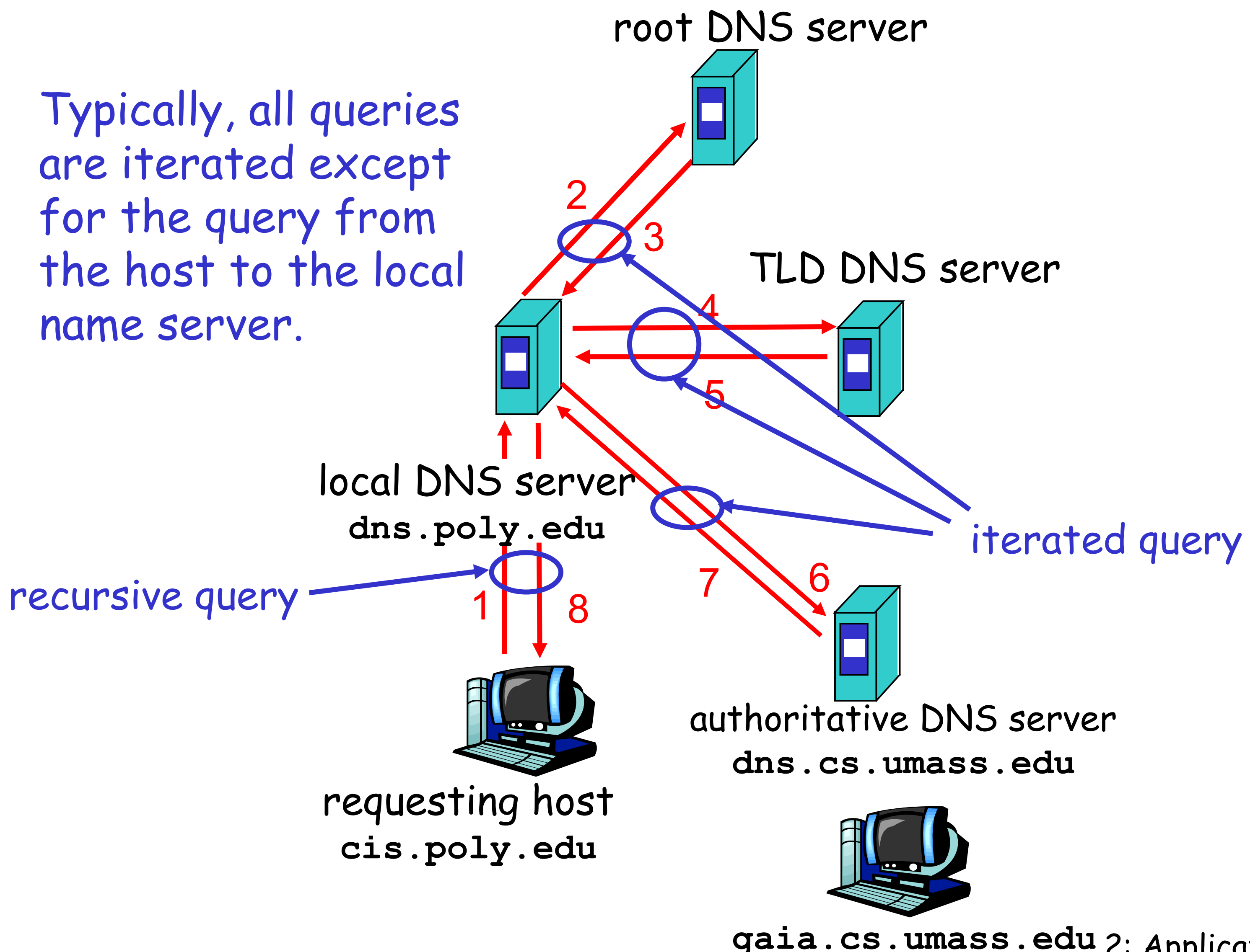
## iterated query:

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



# Recursive and iterated queries

Typically, all queries are iterated except for the query from the host to the local name server.



# DNS: caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (usually two days)
- Until recently, the contents of each DNS servers were configured statically from a configuration file created by a system manager.
- More recently, an UPDATE option has been added to the DNS protocol to allow data to be added or deleted from the database via DNS messages.
- DNS dynamic update mechanism is specified in RFC 2136

# DNS records

DNS: distributed database storing resource records (RR)

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

- Type=A
  - **`name`** is hostname
  - **`value`** is IP address
  - (relay1.bar.foo.com, 145.37.93.126, A)
- Type=NS
  - **`name`** is domain (e.g. foo.com)
  - **`value`** is host name of an authoritative name server for this domain
  - (foo.com, dns.foo.com, NS)
- Type=CNAME
  - **`name`** is alias name for some "canonical" (the real) name

    `www.ibm.com` is really

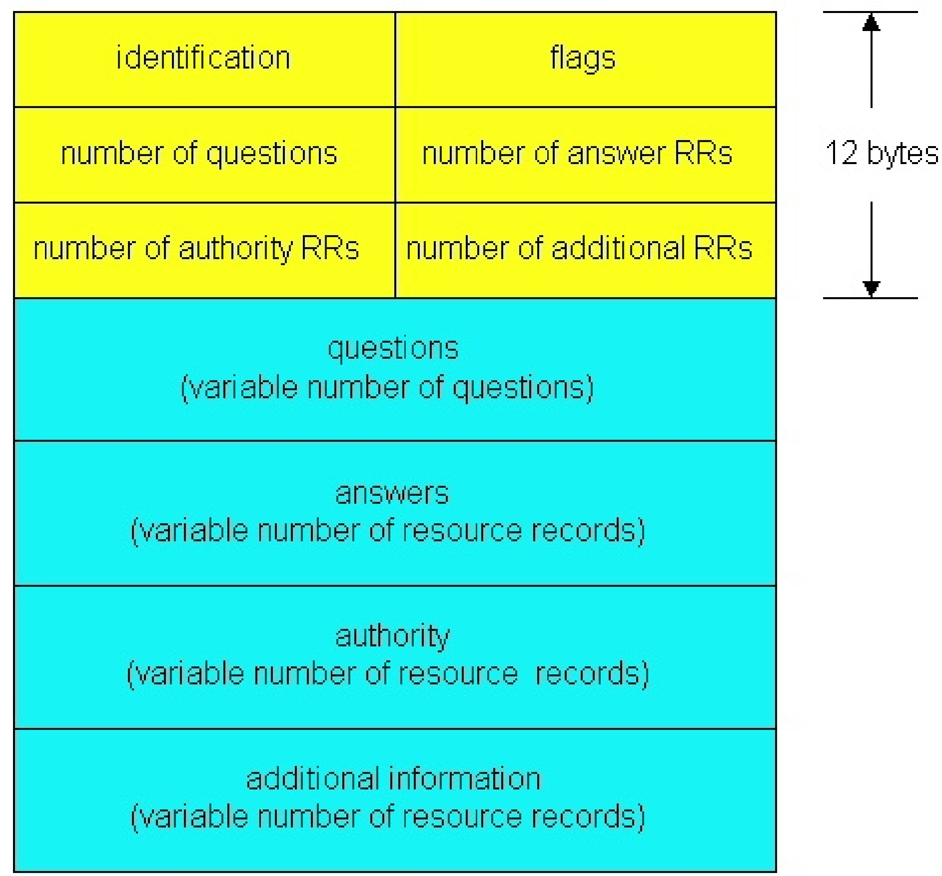
    `servereast.backup2.ibm.com`
  - **`value`** is canonical name
  - (foo.com, relay1.bar.foo.com, CNAME)
- Type=MX
  - **`name`** is alias name for some mail server
  - **`value`** is the canonical name of the mail server
  - (foo.com, mail.bar.foo.com, MX)

# DNS protocol, messages

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

message header

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



# DNS protocol, messages

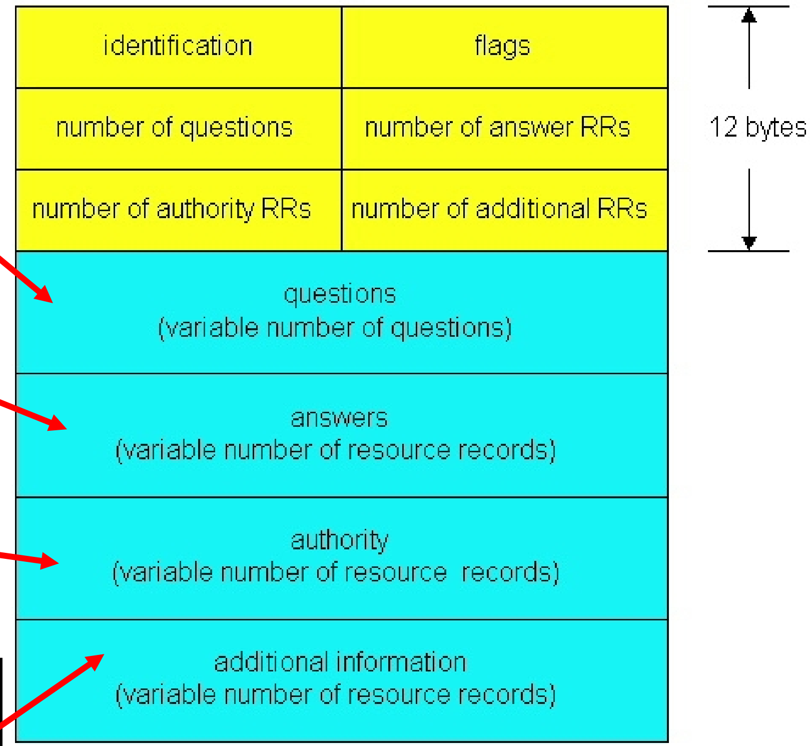
Name, type fields for a query

- RRs in response to query
- A hostname can have multiple IP addresses

records for other authoritative servers

additional “helpful” information that may be used

e.g. IP address for the canonical hostname of the mail server



# Inserting records into DNS

- Example: just created startup "Network Utopia"
- Register name networkuptopia.com at a registrar (e.g., Network Solutions)
  - Need to provide registrar with names and IP addresses of your authoritative name server (primary and secondary)
  - dns1.networkutopia.com, 212.212.212.1
  - dns2.networkutopia.com, 212.212.212.2
- Registrar inserts two RRs into the com TLD server:

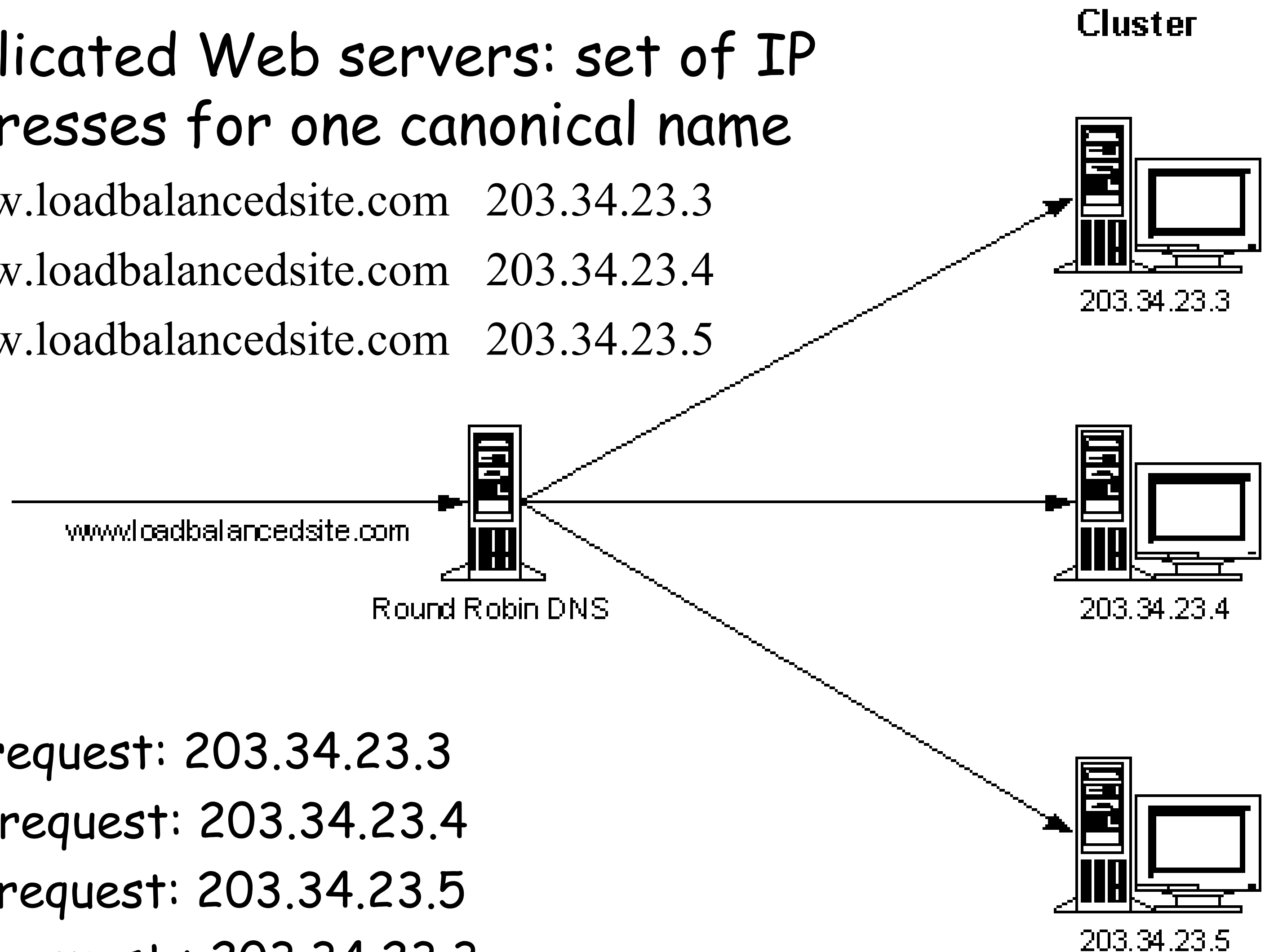
  (networkutopia.com, dns1.networkutopia.com, NS)
  (dns1.networkutopia.com, 212.212.212.1, A)

- Put in authoritative server a type A record for www.networkuptopia.com and a type MX record for mail.networkutopia.com
- How do people get the IP address of your Web site?

# DNS load balancing (DNS Round Robin)

- Replicated Web servers: set of IP addresses for one canonical name

  www.loadbalancedsite.com 203.34.23.3
  www.loadbalancedsite.com 203.34.23.4
  www.loadbalancedsite.com 203.34.23.5




- 1st request: 203.34.23.3
- 2nd request: 203.34.23.4
- 3rd request: 203.34.23.5
- 4th request : 203.34.23.3

# Chapter 2: Application layer

- 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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# File distribution problem

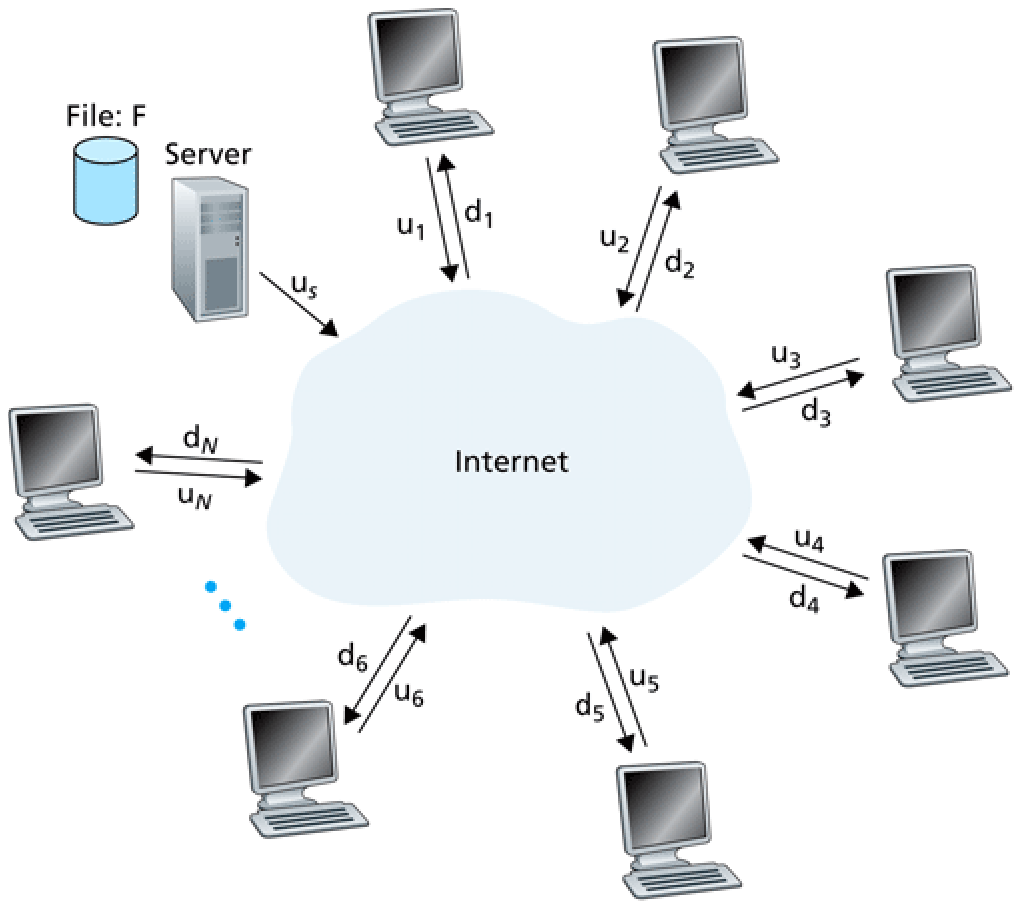



**Figure 2.24** ♦ An illustrative file distribution problem

# File distribution time

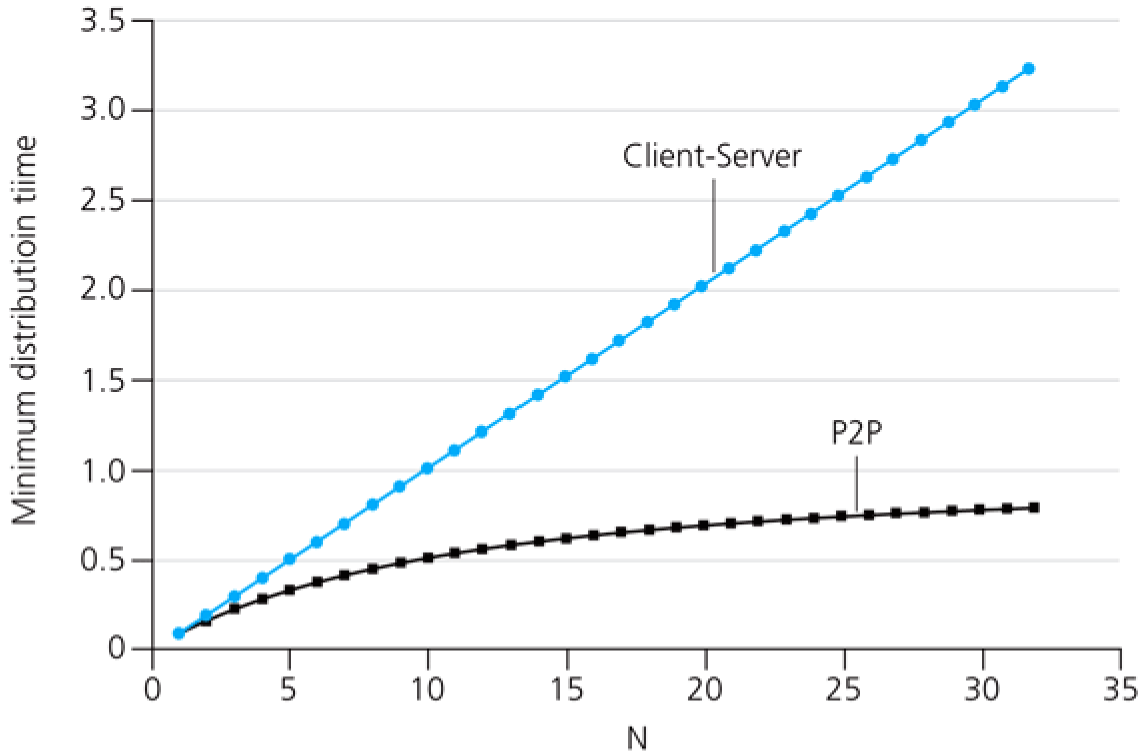



**Figure 2.25** ♦ Distribution time for P2P and client-server architectures

# File distribution with BT

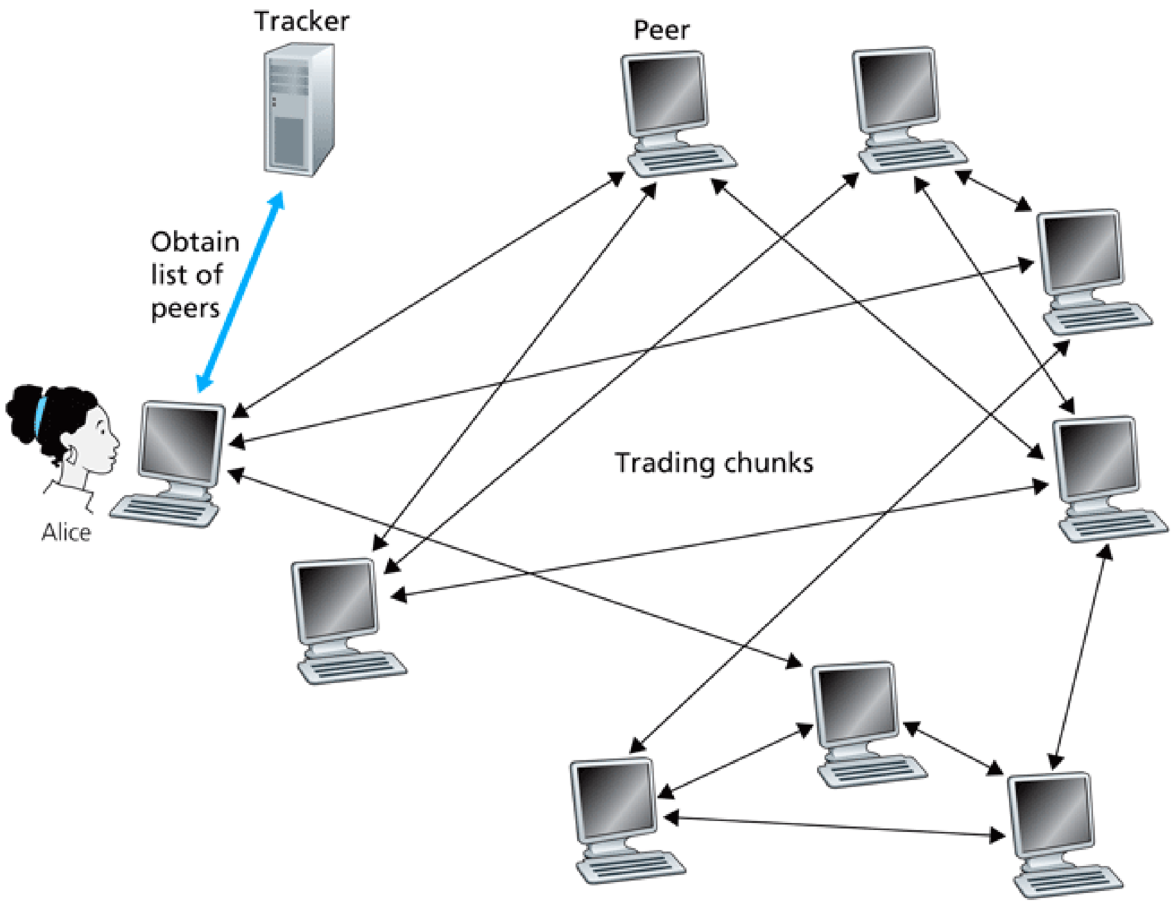



**Figure 2.26** ♦ File distribution with BitTorrent

# P2P file sharing

Example

- Alice runs P2P client application on her notebook computer
- Intermittently connects to Internet; gets new IP address for each connection
- Asks for "Hey Jude"
- Application displays other peers that have copy of Hey Jude.
- Alice chooses one of the peers, Bob.
- File is copied from Bob's PC to Alice's notebook: HTTP
- While Alice downloads, other users uploading from Alice.
- Alice's peer is both a Web client and a transient Web server.

*All peers are servers = highly scalable!*

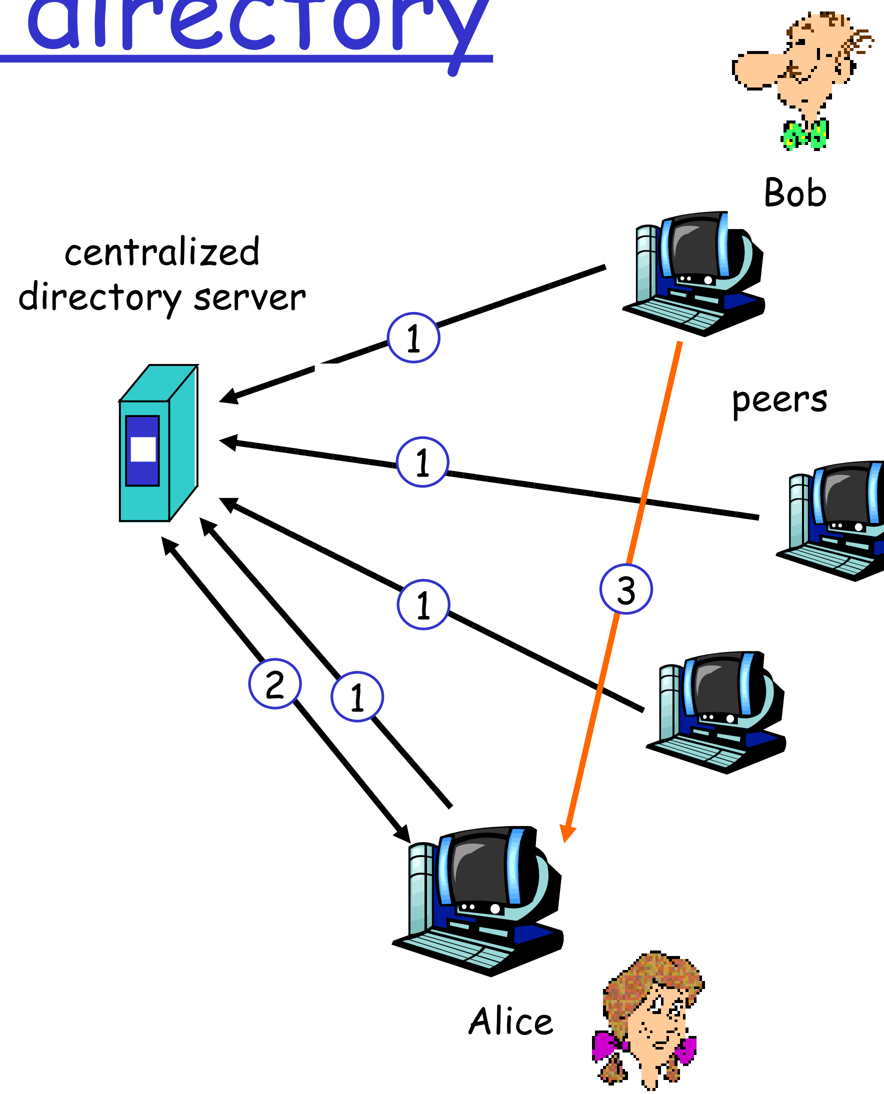
# P2P: centralized directory

original "Napster" design

1) when peer connects, it informs central server:
   - IP address
   - content

2) Alice queries for "Hey Jude"

3) Alice requests file from Bob



# P2P: problems with centralized directory

- Single point of failure
- Performance bottleneck
- Copyright infringement

file transfer is decentralized, but locating content is highly centralized

# Query flooding: Gnutella

- fully distributed
  - no central server
- public domain protocol
- many Gnutella clients implementing protocol

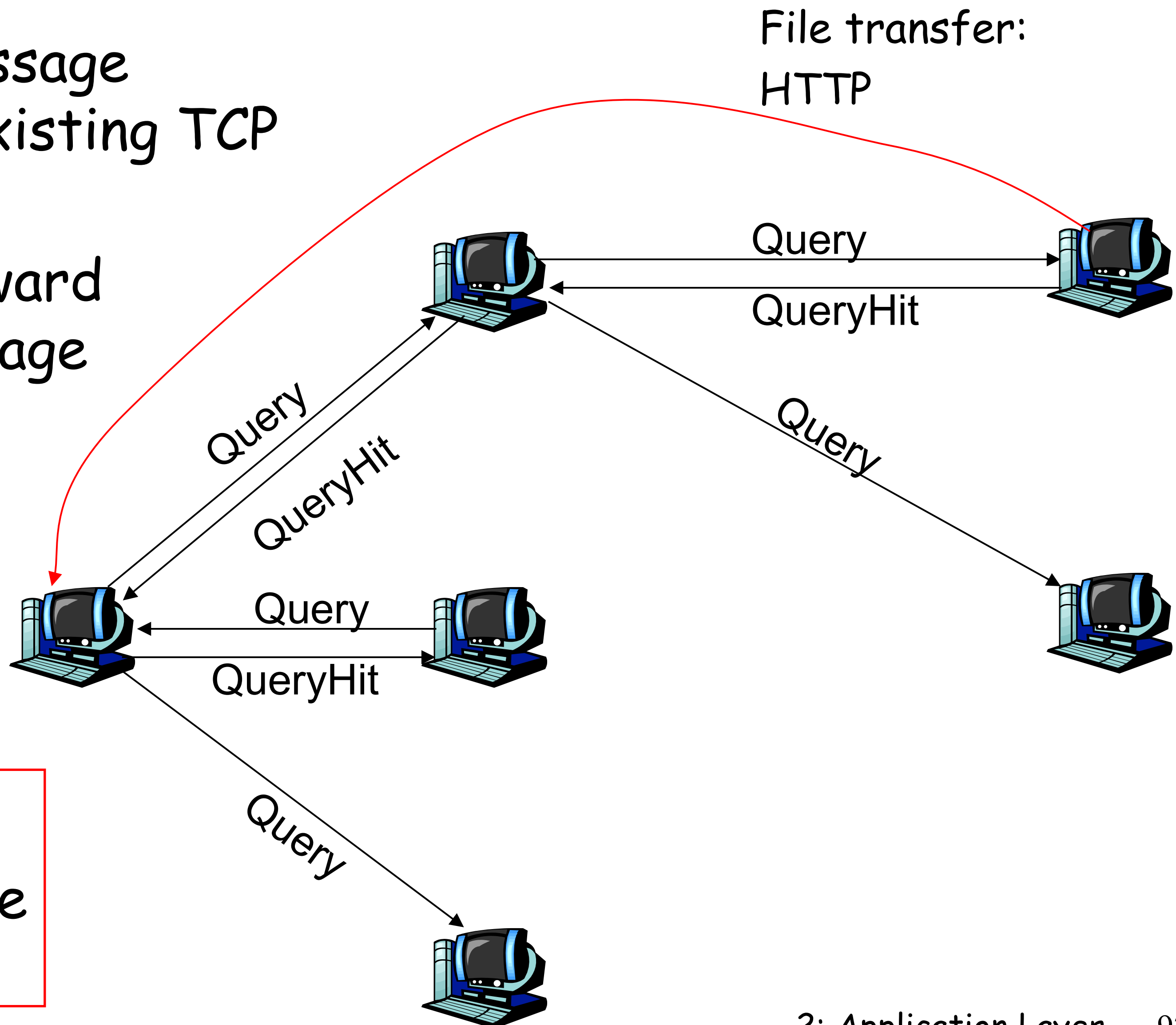
overlay network: graph

- edge between peer X and Y if there's a TCP connection
- all active peers and edges is overlay net
- Edge is not a physical link
- Given peer will typically be connected with < 10 overlay neighbors

# Gnutella: protocol

- Query message sent over existing TCP connections
- peers forward Query message
- QueryHit sent over reverse path

Scalability: limited scope flooding



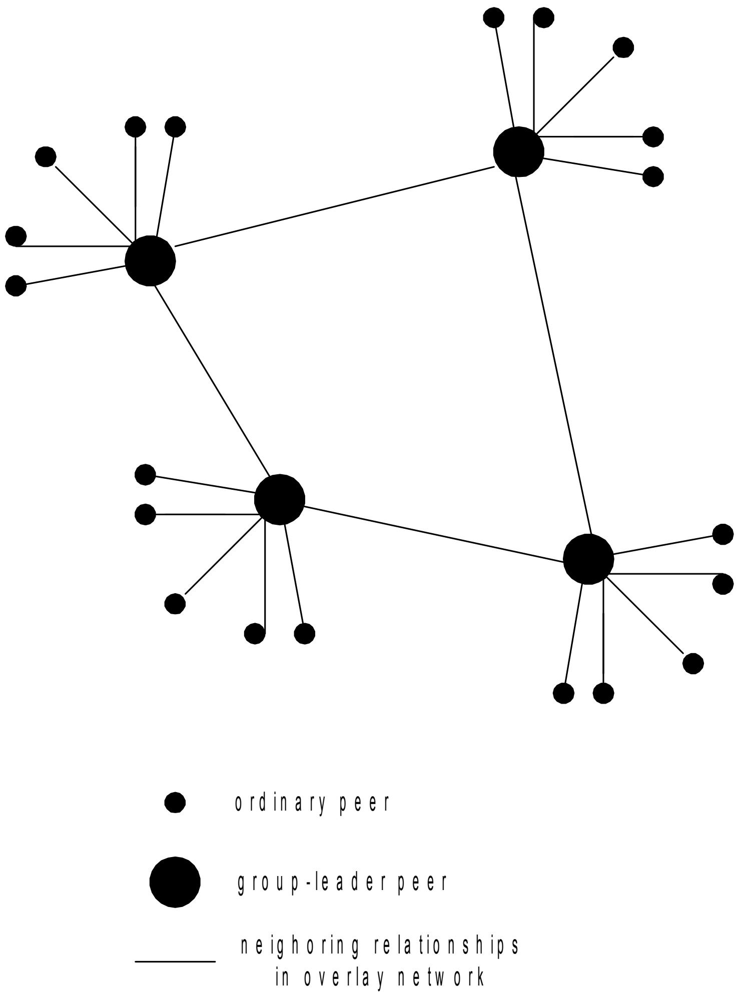
# Gnutella: Peer joining

1. Joining peer X must find some other peer in Gnutella network: use list of candidate peers
2. X sequentially attempts to make TCP with peers on list until connection setup with Y
3. X sends Ping message to Y; Y forwards Ping message.
4. All peers receiving Ping message respond with Pong message
5. X receives many Pong messages. It can then setup additional TCP connections

Peer leaving: see homework problem!

# Exploiting heterogeneity: KaZaA

- Each peer is either a group leader or assigned to a group leader.
  - TCP connection between peer and its group leader.
  - TCP connections between some pairs of group leaders.
- Group leader tracks the content in all its children.



# KaZaA: Querying

- Each file has a hash and a descriptor
- Client sends keyword query to its group leader
- Group leader responds with matches:
  - For each match: metadata, hash, IP address
- If group leader forwards query to other group leaders, they respond with matches
- Client then selects files for downloading
  - HTTP requests using hash as identifier sent to peers holding desired file

# Kazaa tricks

- Request queuing
  - Limitation on the number of simultaneous uploads
- Incentive priorities
  - Give priority to users who have uploaded more files than they have downloaded
- Parallel downloading
  - Use the byte-range header of HTTP to request different portion of the file from different peers

# 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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# 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 applications
- 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 Java

- Advantages:
  - Cross platform without recompiling
  - Easy programming with high-level API
- Preparation:
  - JDK (Java Development Kit)
    - http://java.sun.com
  - Free IDE(Integrated Development Environment): Eclipse (optional)
    - http://www.eclipse.org

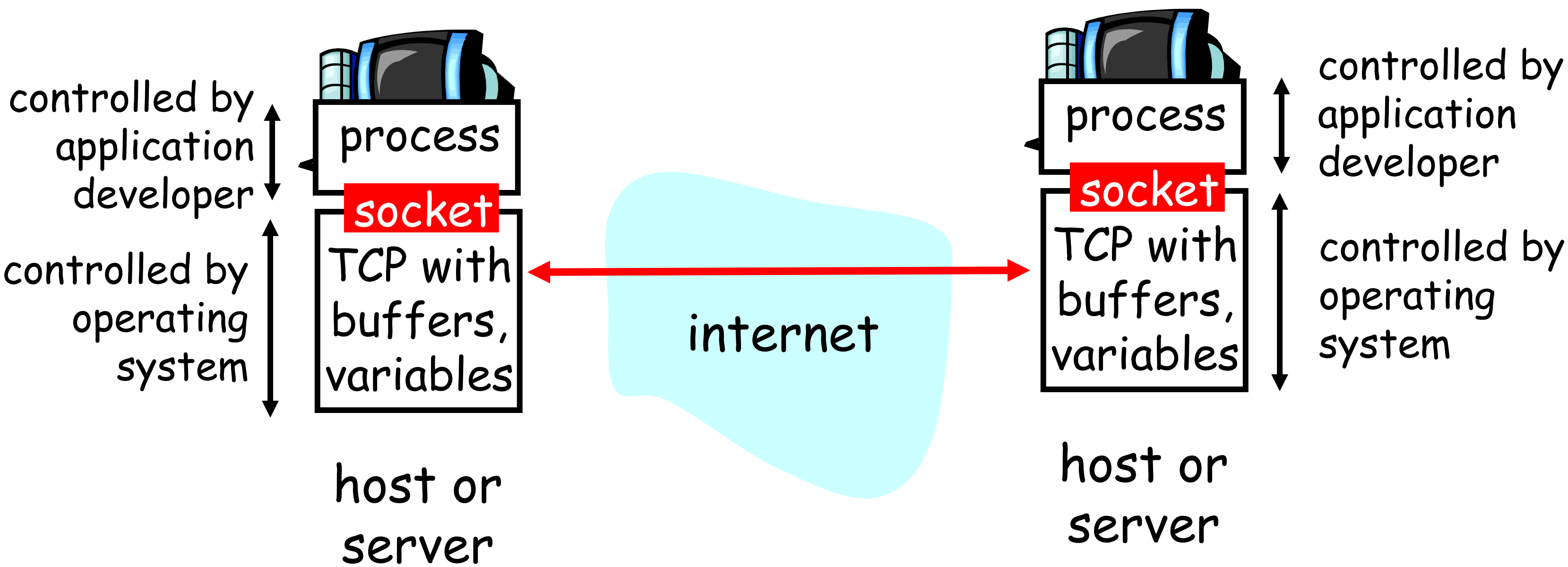
# Introduction to the Java Programming Language

- Object-Oriented
  - Classes
    - Methods
    - Members
    - The public static void Main method
  - Inheritance
- No “pointers”, just “references”
- Stream-based I/O
- Exception handling

# Socket-programming using TCP

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

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



# 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*

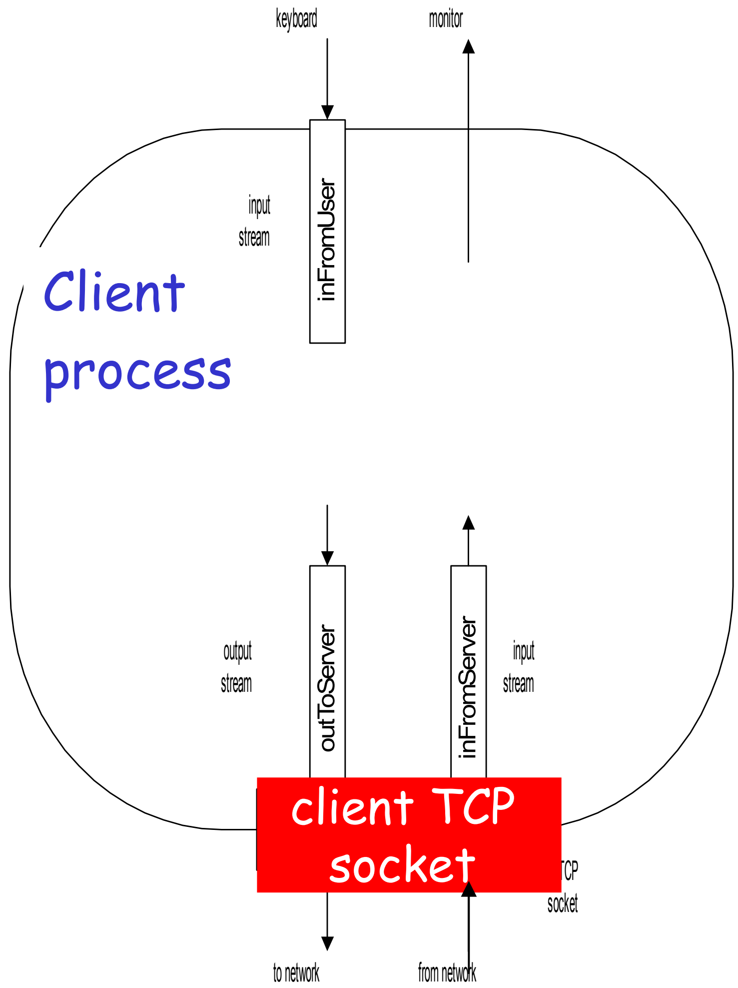
# Stream jargon

- 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, eg, keyboard or socket.
- An output stream is attached to an output source, eg, monitor or socket.

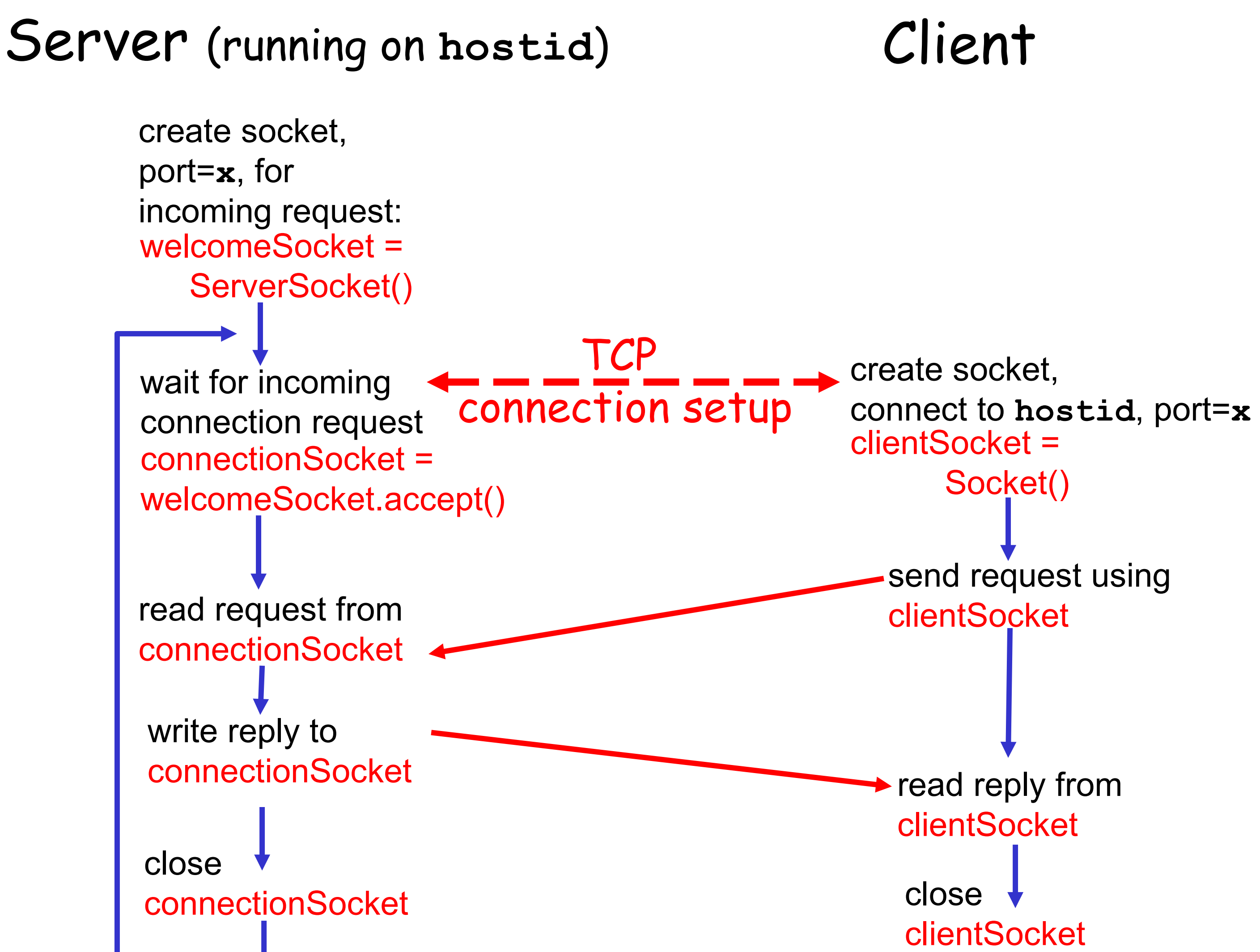
# Socket programming with TCP

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)



# Client/server socket interaction: TCP



# Introduction to Java Sockets: TCP

- Class:
  - Java.net.Socket
  - Java.net.ServerSocket
- Stream-based I/O
- InputStream = socket.getInputStream()
- OutputStream = socket.getOutputStream()

# Socket Programming using Java: TCP Client

- A "Socket" object, making connection requests
- Parameters:
  - Remote ip address
  - Remote tcp port
  - Local ip address (optional)
  - Local port (optional)
- (Code Example)

# Example: Java client (TCP)

java.io package contains classes for input and output streams

java.net package contains classes for network support

```
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
- attach to standard input

- Create client socket
- connect to server

- Create output stream
- attach to clientSocket

# Example: Java client (TCP), cont.

- Create input stream
- attach to clientSocket

Send line to server

Read line from server

Close the clientSocket

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

sentence = inFromUser.readLine();

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

modifiedSentence = inFromServer.readLine();

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

clientSocket.close();

    }
}
```

# Socket Programming using Java: TCP Server

- A "ServerSocket" object, waiting for incoming connection requests.
- Then, a new "Socket" object initiated to communicate with the remote "Socket" object.
- A "ServerSocket" then continues waiting for incoming connection requests.
- Parameters:
  - Local tcp port
  - Local ip address (optional)
- (Code Example)

# Example: Java server (TCP)

```
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 a new socket, connectionSocket, when contacted by a client

- Create input stream
- attach to socket, connectionSocket

# Example: Java server (TCP), cont

- Create output stream
- Attach to socket, connectionSocket

Read in line from socket

Write out line to socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

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

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 file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# 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 unreliable transfer of groups of bytes ("datagrams") between client and server*

# Introduction to Java Sockets: UDP

- Class:
  - DatagramSocket
  - DatagramPacket
- Packet-based I/O

# Client/server socket interaction: UDP

## Server (running on `hostid`)

create socket,
port=**x**, for
incoming request:
serverSocket =
DatagramSocket()

read request from
serverSocket

write reply to
serverSocket
specifying client
host address,
port number

## Client

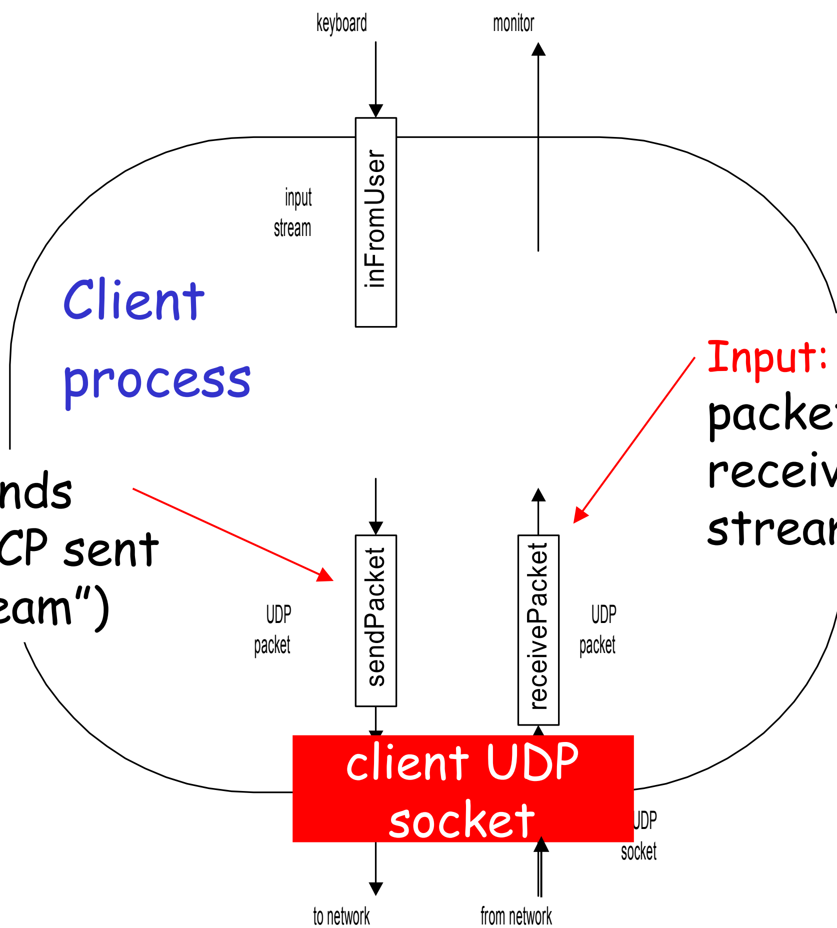
create socket,
clientSocket =
DatagramSocket()

Create, address (**hostid, port=x**,
send datagram request
using clientSocket

read reply from
clientSocket

close
clientSocket

# Example: Java client (UDP)




Output: sends packet (TCP sent "byte stream")

Input: receives packet (TCP received "byte stream")

# Socket Programming using Java: UDP Client

- A "DatagramSocket" object used to send a "DatagramPacket" without establishing connections.
- Parameters:
  - Remote ip address
  - Remote udp port
  - Local ip address (optional)
  - Local udp port (optional)
- (Code Example)

# Example: Java client (UDP)

- Create input stream
- attach to standard input

Create client socket

Translate hostname to IP address using DNS

Covert string to array of bytes

```
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();
```

# Example: Java client (UDP), cont.

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

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

Send packet to server

```
clientSocket.send(sendPacket);
```

Creat place holder for receiving packet

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

Read packet from server

```
clientSocket.receive(receivePacket);
```

Extract data from receivePacket

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

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

# Socket Programming using Java: UDP Server

- A “DatagramSocket” object used to receive “DatagramPacket” objects on a specified port
- Parameters:
  - Local udp port
  - Local ip address (optional)
- (Code Example)

# Example: Java server (UDP)

Create datagram socket at port 9876

Create space for received packet

Receive packet

```
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);
```

# Example: Java server (UDP), cont

Extract data from receivePacket

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

Extract IP addr port #, of sender

```
InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

                String capitalizedSentence = sentence.toUpperCase();
```

Covert string to array of bytes

```
sendData = capitalizedSentence.getBytes();
```

Create packet to send to client

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

Write out packet to socket

```
serverSocket.send(sendPacket);
   }
  }
}
```

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

# Javadoc

- Available from http://java.sun.com
- Documentation for standard java APIs
  - Socket
  - ServerSocket
  - DatagramSocket
  - DatagramPacket
- Example 1:
  - Connecting to a remote socket with specified local port
- Example 2:
  - Accepting connections on a specified ip address

# Exception Handling

- Network programs encounter “exceptions“ like:
  - Failure during initializing sockets
  - Failure during establishing connection
  - Failure during data transmission
- Programmers should write codes to handle these “exceptions“

# Summary

- A ***socket*** is one end-point of a two-way communication link between two programs running on the network.
- A ***ip socket*** is identified by a ***ip address*** and a ***tcp/udp port***.
- For TCP,
  - Use ServerSocket to accept connections
  - Use Socket to make connection requests
- For UDP,
  - Use DatagramSocket to send/receive DatagramPackets

# Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

# Building a simple Web server

Build a server that do the following:

- handles only one HTTP request
- accepts the request
- parses header
- obtains requested file from server's file system
- creates HTTP response message:
  - header lines + file
- sends response to client

- after creating server, you can request file using a browser (eg IE explorer)
- see  text for details

# Example: a Java Web Server

This part is the same as TCP server example

```
import j ava.io.*;
import j ava.net.*;
import java.util.*;
class Webserver {
  public static void main(String argv[]) throws Exception
  {
    String requestMessageLine;
    String fileName;
    ServerSocket listenSocket = new ServerSocket(6789);
    Socket connectionSocket = listenSocket.accept();
    BufferedReader inFromClient =
      new BufferedReader(new InputStreamReader(
           connectionSocket.get!nputStream()));
    DataOutputStream outToClient =
      new DataOutputStream(
           connectionSocket.getOutputStream());
```

Create welcoming socket at port 6789

- Wait, on welcoming socket for contact by client
- Create a new socket, connectionSocket, when contacted by a client

- Create input stream
- attach to socket, connectionSocket

- Create output stream
- Attach to socket, connectionSocket

Continue on next page

# Example: a Java Web Server, cont

Read the 1st line of the HTTP request "GET file-name HTTP/1.0"

```
requestMessageLine = inFromClient.readLine();
```

Parse the requestMessageLine to obtain the words

```
StringTokenizer tokenizedLine =
        new StringTokenizer(requestMessageLine);
```

Determine whether the 1st word is "get"

```
if (tokenizedLine.nextToken().equals("GET")){
```

Extract the filename

```
fileName = tokenizedLine.nextToken();
```

Remove the slash that may precede the filename

```
if (fileName.startsWith("/") == true )
        fileName = fileName.substring(1);
```

Continue on next page

# Example: a Java Web Server, cont

Creates a new File instance, file

Obtain the length of the file

- Creates a FileInputStream, inFile
- Attach the stream to the file, filename

Create a byte array, fileInBytes

Read the file inFile to the byte array fileInBytes

```
File file = new File(fileName);

int numOfBytes = (int) file.length();

FileInputStream inFile = new FileInputStream (fileName);

byte[] fileInBytes = new byte[numOfBytes];

inFile.read(fileInBytes);
```

Continue on next page

# Example: a Java Web Server, cont

Write HTTP response status line to output stream

Write HTTP response header line (Content-Type) to output stream

Write HTTP response header line (Content-Length) to output stream

Write the file to output stream

```
outToClient.writeBytes(
                "HTTP/I.0 200 Document Follows\r\n");
   if (fileName.endsWith(".jpg"))
      outToClient.writeBytes("Content-Type:image/jpeg\r\n");
   if (fileName.endsWith(".gif"))
     outToClient.writeBytes("Content-Type:image/gif\r\n");
   outToClient.writeBytes("Content-Length: " +
             numOfBytes + "\r\n");
   outToClient.writeBytes("\r\n") ;
   outToClient.write(fileInBytes, 0, numOfBytes);
   connectionSocket.close();
   }
 else System.out.println("Bad Request Message");
 }
}
```

# Socket programming: references

C-language tutorial (audio/slides):

- "Unix Network Programming" (J. Kurose),

http://manic.cs.umass.edu/~amldemo/courseware/intro.

Java-tutorials:

- "All About Sockets" (Sun tutorial), http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html
- "Socket Programming in Java: a tutorial," http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html

# 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
- 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

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