

# Chapter 2

## Application Layer

### A note on the use of these ppt slides:

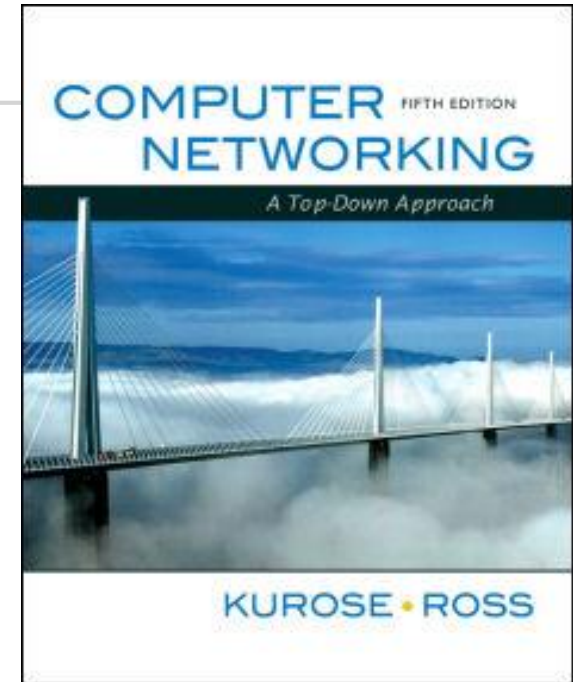
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

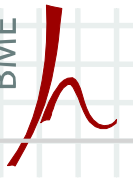
Thanks and enjoy! JFK/KWR

All material copyright 1996-2010

J.F Kurose and K.W. Ross, All Rights Reserved



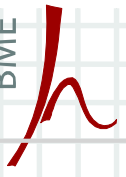
**Computer Networking: A  
Top Down Approach  
Featuring the Internet,  
5th edition.  
Jim Kurose, Keith Ross  
Pearson Addison-Wesley,  
2009.**



# Chapter 2: Application layer

---

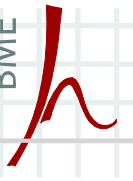
- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing



# 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



# 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
- ...
- ...
-

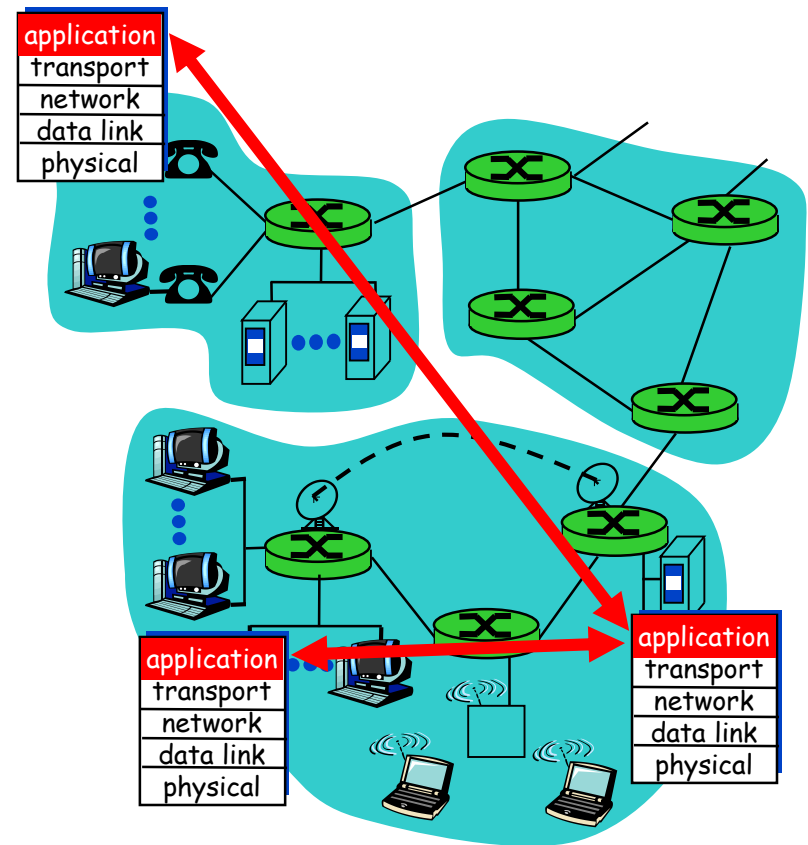
# Creating a network app

## Write programs that

- run on different end systems and
- communicate over a network.
- e.g., Web: Web server software communicates with browser software

## Little software written for devices in network core

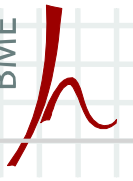
- network core devices do not run user application code
- application on end systems allows for rapid app development, propagation



# 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

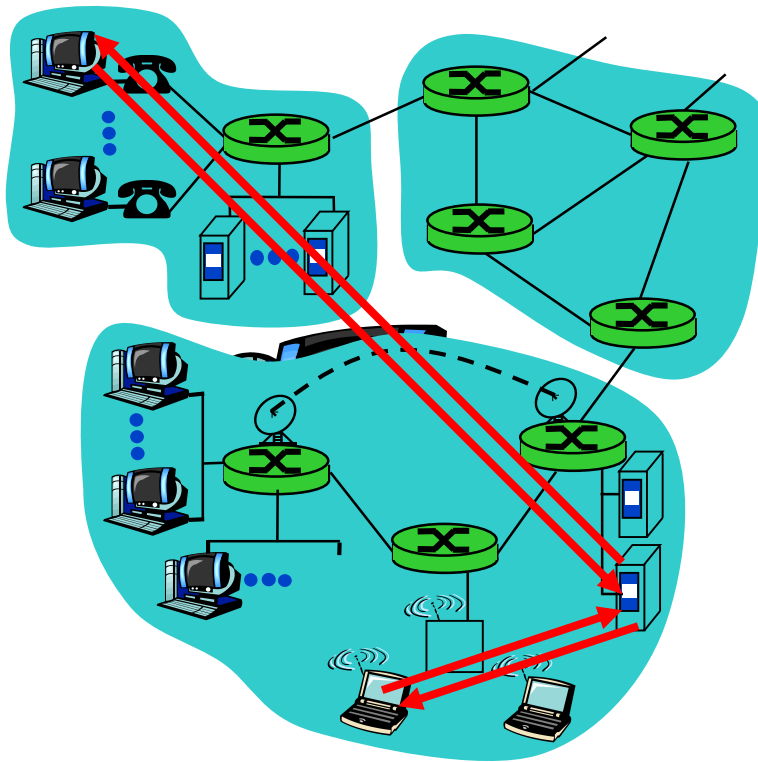


# Application architectures

---

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

# Client-server architecture



## Server:

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

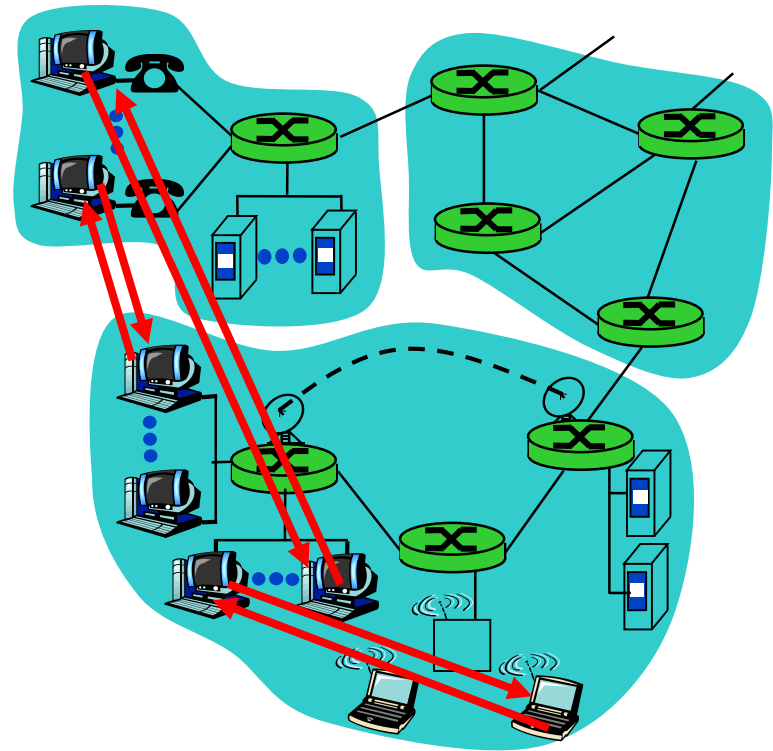
## Clients:

- 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



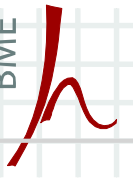
# Hybrid of client-server and P2P

## Skype

- Internet telephony app
- Finding address of remote party: centralized server(s)
- Client-client connection is direct (not through server)

## 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



# Process communication

**Process:** program running within a host

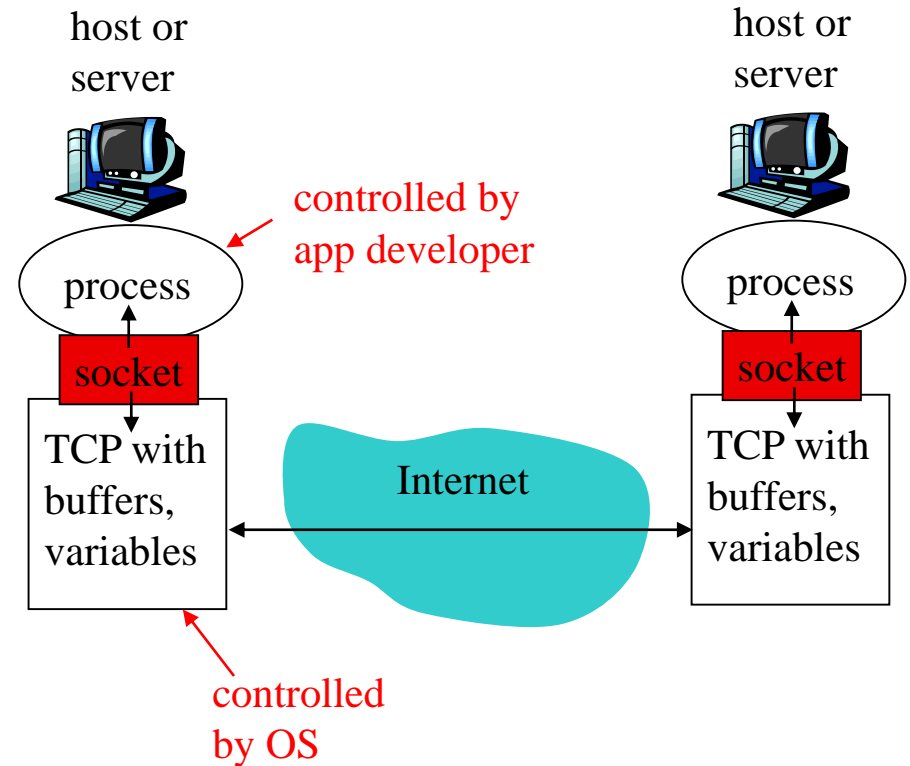
- Within same host, two processes communicate using **inter-process communication** (defined by OS).
- Processes in different hosts communicate by exchanging **messages**

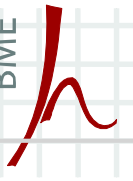
**Client process:** process that initiates communication

**Server process:** process that waits to be contacted

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

- 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)

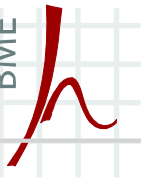




# Addressing processes

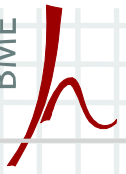
---

- 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?



# 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?
  - **Answer:** NO, many processes can be running on same host
- *Identifier* includes both **IP address** and **port numbers** associated with process on host
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- To send HTTP message to gaia.cs.umass.edu web server:
  - **IP address:** 128.119.245.12
  - **Port number:** 80
- More shortly...



# App-layer protocol defines

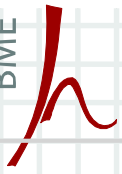
- Types of messages exchanged
  - e.g., request, response
- Message syntax
  - what fields in messages & how fields are delineated
- Message semantics
  - meaning of information in fields
- Rules for when and how processes send & respond to messages

## Public-domain protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

## Proprietary protocols:

- e.g., 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

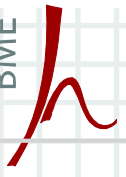
- Some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

## Throughput

- Some apps (e.g., multimedia) require minimum amount of bandwidth to be “effective”
- Other apps (“elastic apps”) make use of whatever bandwidth they get

## Security

- Encryption, data integrity,  
...



# 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 video: 10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
instant messaging	no loss	elastic	yes and no



# Internet transport protocols services

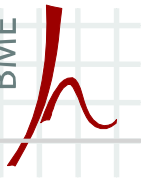
## TCP service:

- *Connection-oriented*: setup required between client and server processes
- *Reliable transport* between sending and receiving process
- *Flow control*: sender won't overwhelm receiver
- *Congestion control*: throttle sender when network overloaded
- *Does not provide*: timing, minimum bandwidth guarantees

## UDP service:

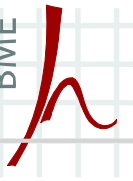
- Unreliable data transfer between sending and receiving process
- Does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

Q: Why bother? Why is there a UDP?



# Internet apps: Application, transport protocols

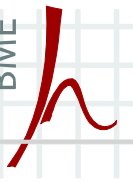
<b>Application</b>	<b>Application layer protocol</b>	<b>Underlying transport protocol</b>
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Vonage, Dialpad)	typically UDP



# 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



# Web and HTTP

## First some jargon

- **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 (Uniform Resource Locator)**
- **Example URL:**

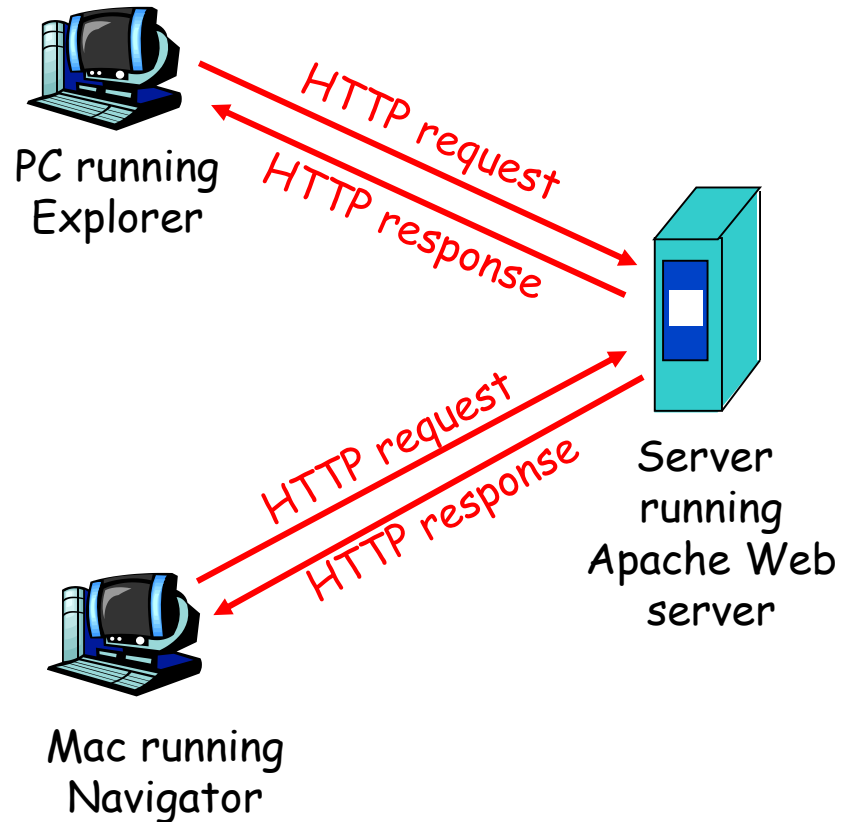
`www.someschool.edu/someDept/pic.gif`

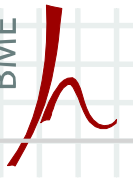
host name

path name

## 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 2068





# 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

**Protocols that maintain “state”<sup>aside</sup> 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
- HTTP/1.1 uses persistent connections in default mode

# 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

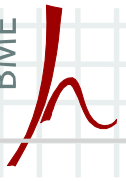
1b. HTTP server at host

`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'd)

- 
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects
  4. HTTP server closes TCP connection.
  6. Steps 1-5 repeated for each of 10 jpeg objects

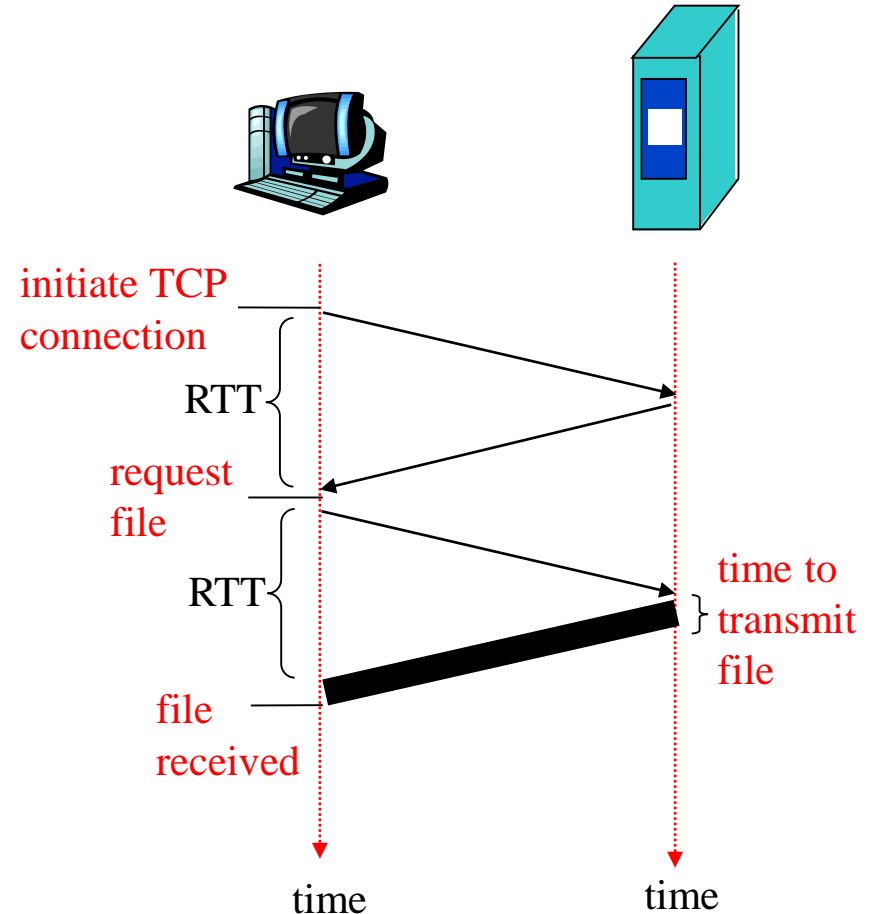
# Non-Persistent HTTP: Response time

**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**



## Nonpersistent HTTP issues

- Requires 2 RTTs per object
- OS overhead for *each* TCP connection
- Browsers often open parallel TCP connections to fetch referenced objects

## Persistent HTTP

- Server leaves connection open after sending response
- Subsequent HTTP messages between same client/server sent over open connection

## Persistent *without* pipelining

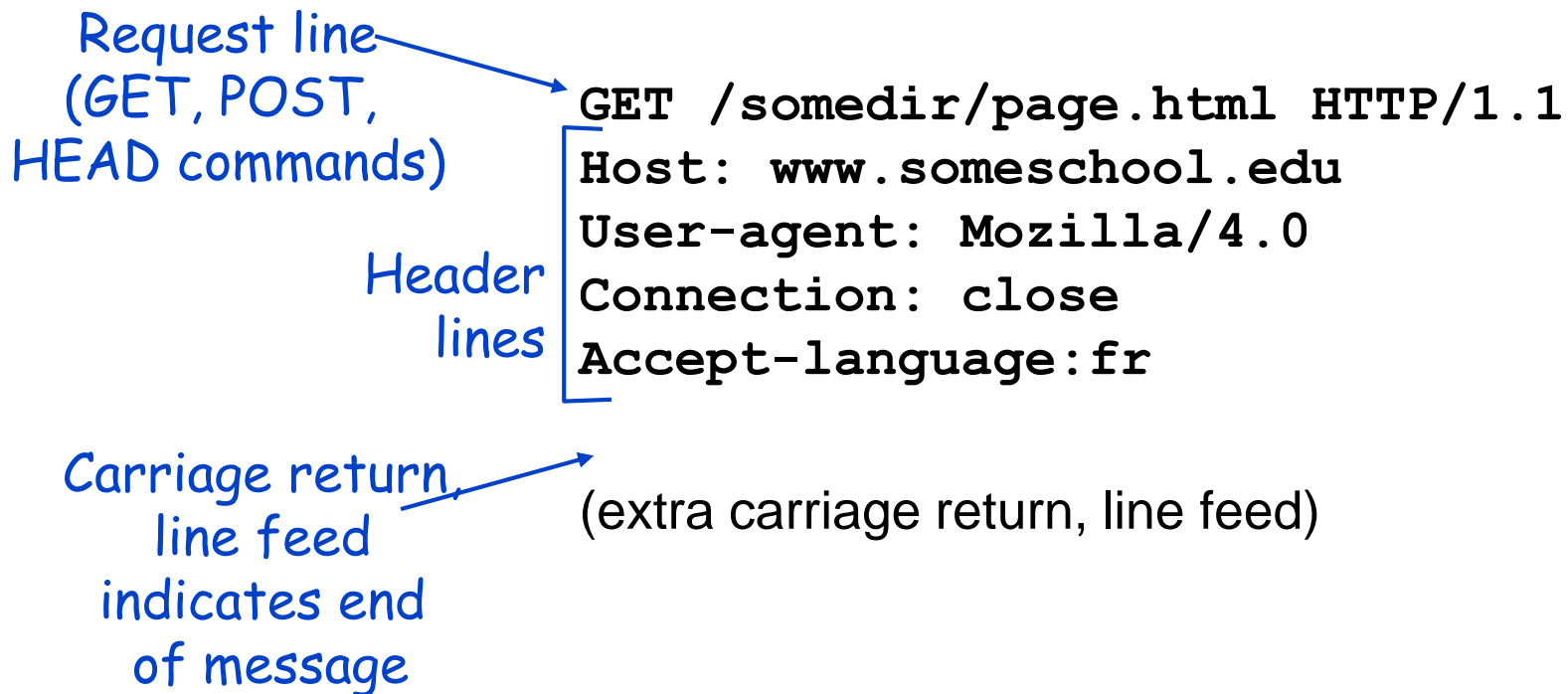
- Client issues new request only when previous response has been received
- One RTT for each referenced object

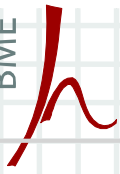
## Persistent *with* pipelining

- 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

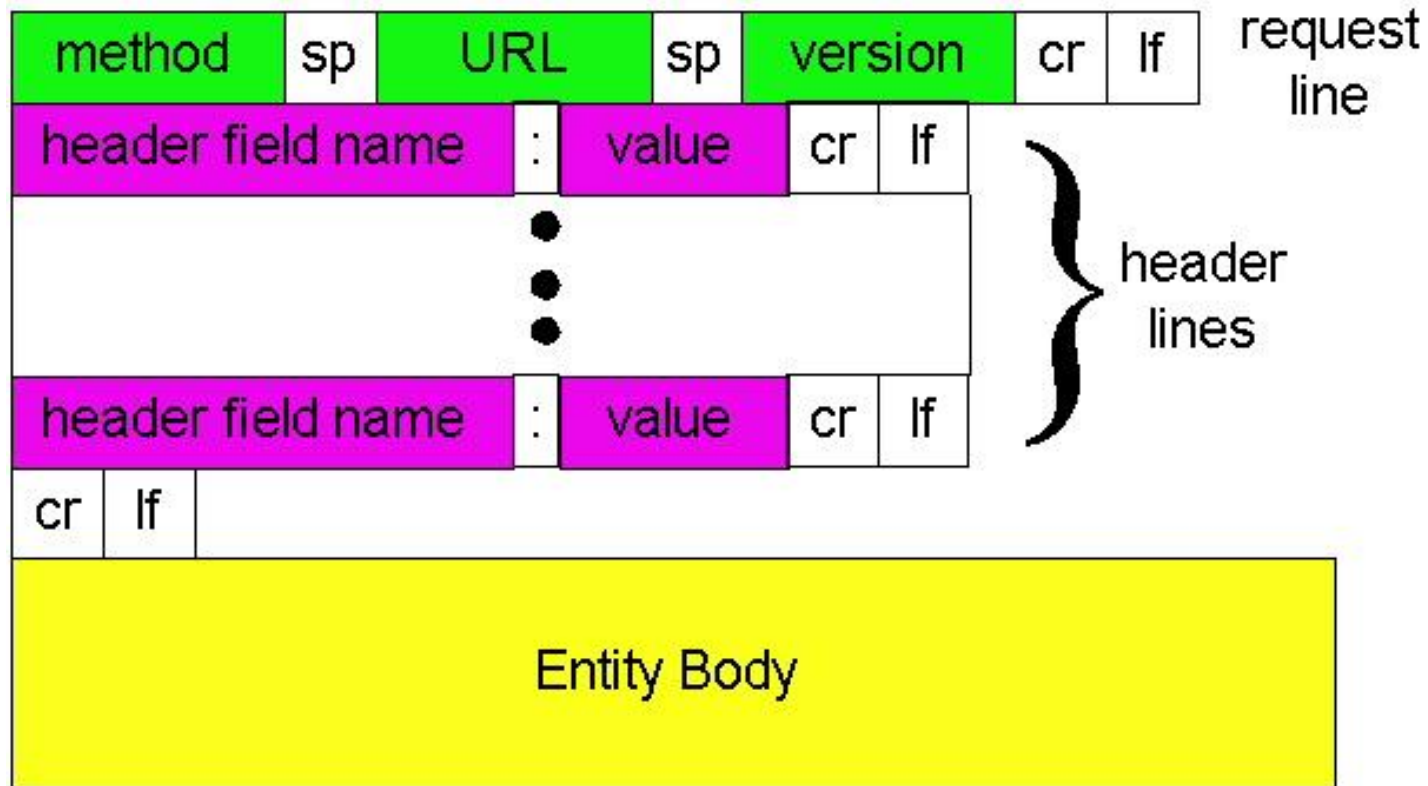
# HTTP request message

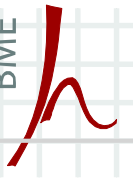
- Two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)





# 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`

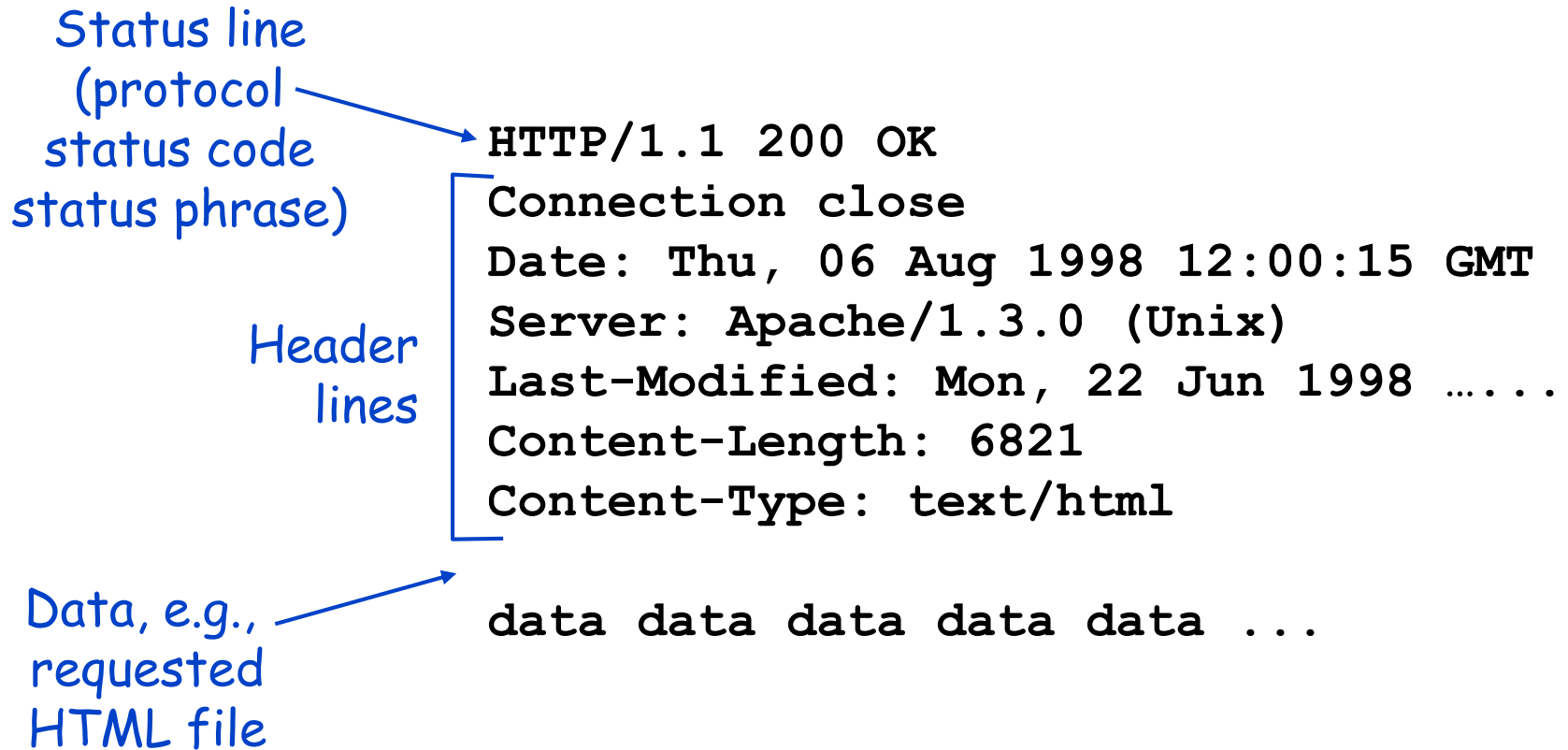
## HTTP/1.0

- GET
- POST
- HEAD
  - Asks server to leave requested object out of response

## HTTP/1.1

- GET, POST, HEAD
- PUT
  - Uploads file in entity body to path specified in URL field
- DELETE
  - Deletes file specified in the URL field

# HTTP response message





# HTTP response status codes

In first line in server->client response message.

A few sample codes:

## 200 OK

- Request succeeded, requested object later in this message

## 301 Moved Permanently

- Requested object moved, new location specified later in this message (Location:)

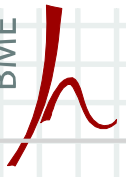
## 400 Bad Request

- Request message not understood by server

## 404 Not Found

- Requested document not found on this server

## 505 HTTP Version Not Supported



# Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

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

2. Type in a GET HTTP request:

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

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

3. Look at response message sent by HTTP server!

Many major Web sites use cookies

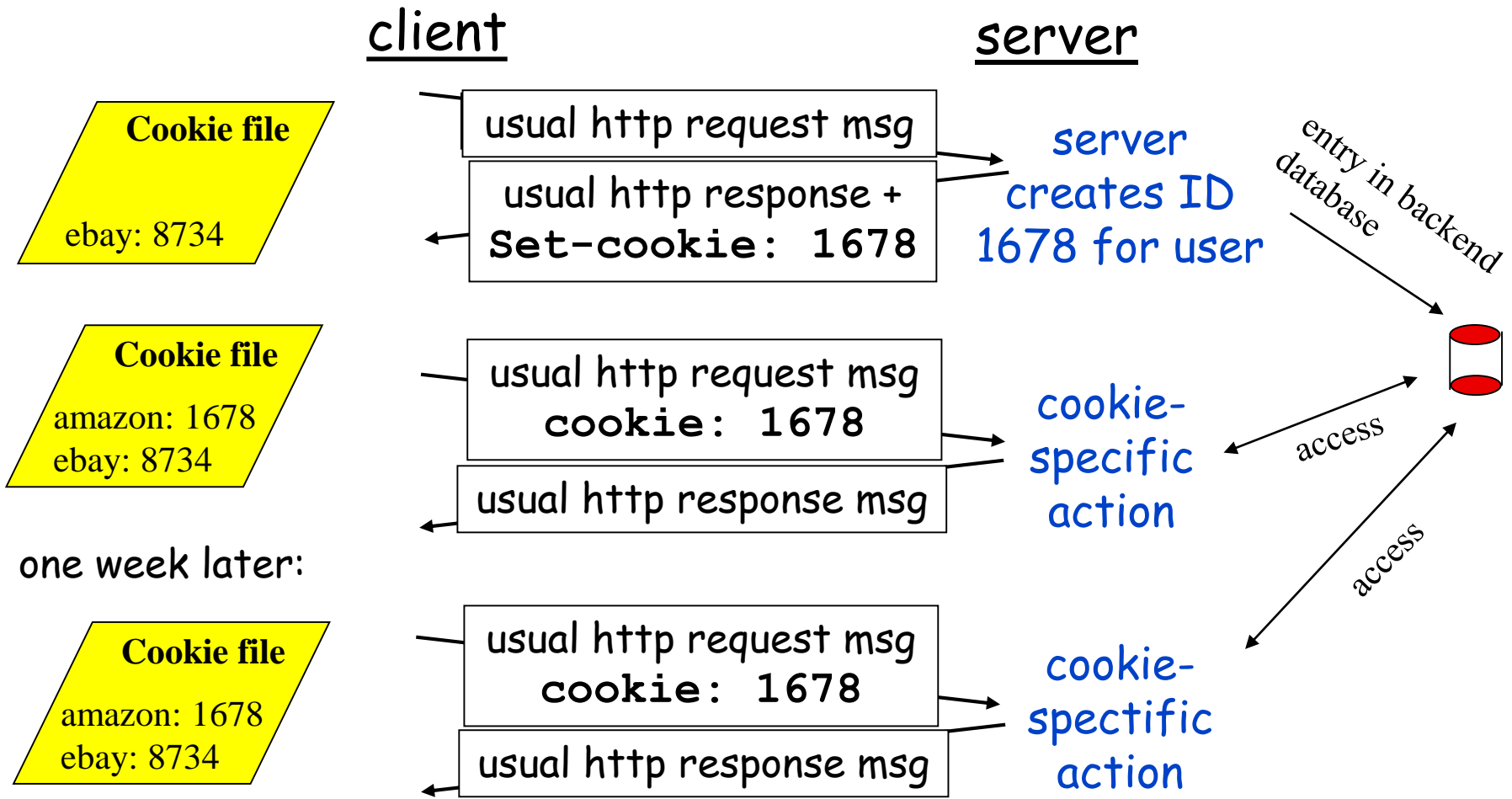
## Four components:

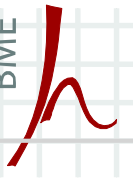
- 1) Cookie header line of HTTP *response* message
- 2) Cookie header line in HTTP *request* message
- 3) Cookie file kept on user's host, managed by user's browser
- 4) Back-end database at Web site

## Example:

- Susan 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’d)





# Cookies (continued)

## What cookies can bring:

- Authorization
- Shopping carts
- Recommendations
- User session state (Web e-mail)

## How to keep “state”:

- Protocol endpoints: maintain state at sender/receiver over multiple transactions
- Cookies: http messages carry state

aside

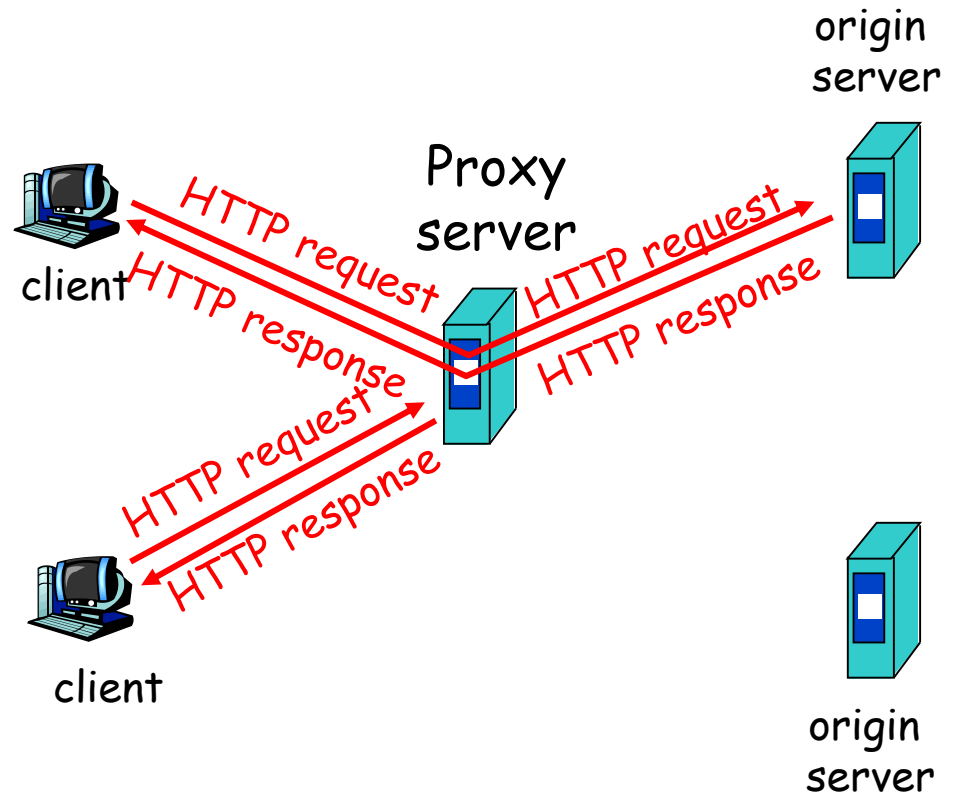
## Cookies and privacy:

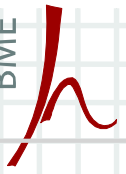
- Cookies permit sites to learn a lot about you
- You may supply name and e-mail to sites

# Web caches (proxy server)

**Goal:** satisfy client request without involving origin server

- User sets browser: Web accesses via cache
- Browser sends all HTTP requests to cache
  - Object in cache: cache returns object
  - Else cache requests object from origin server, then returns object to client





# 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)

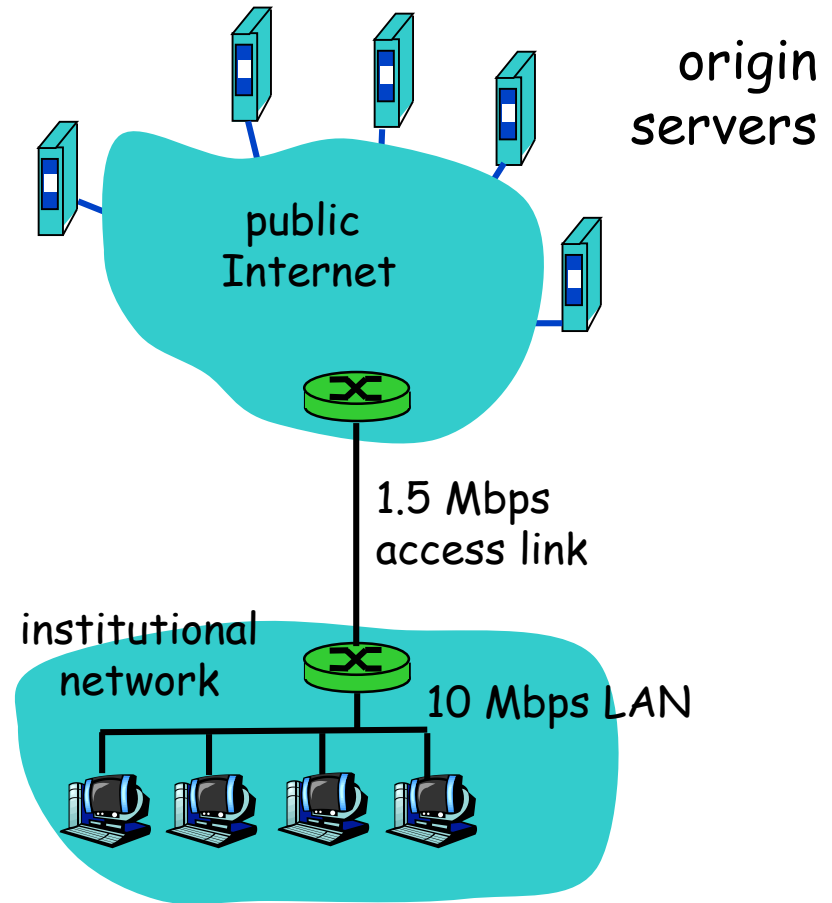
# Caching example

## Assumptions

- Average object size = 100,000 bits
- Avg. request rate from institution's browsers to origin servers = 15/sec
- Delay from Internet edge 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 + secs + msecs



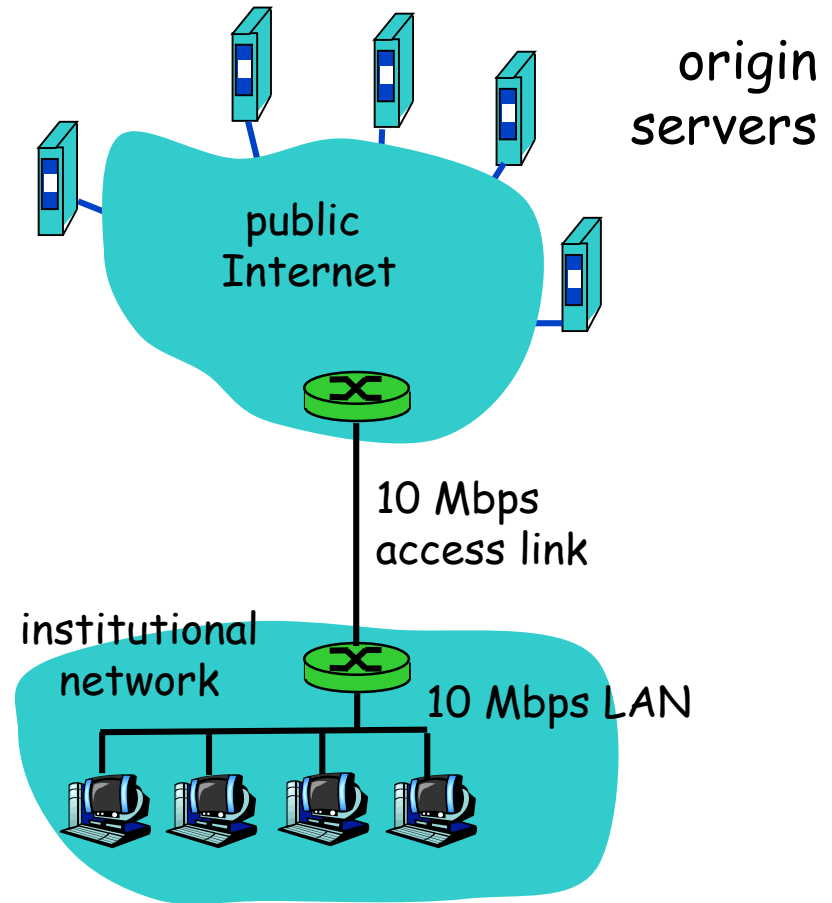
# Caching example (cont'd)

## Possible solution

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

## Consequences

- Utilization on LAN = 15%
- Utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay  
= 2 sec + msec + msec
- Often a costly upgrade



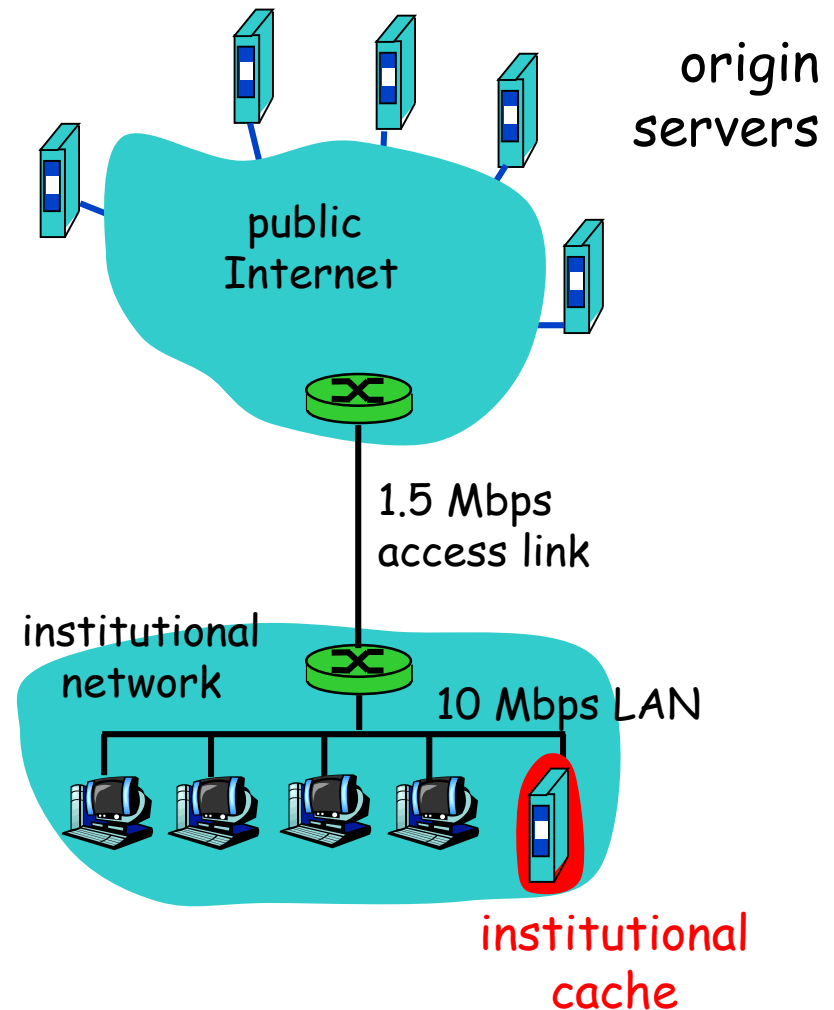
# Caching example (cont'd)

## Install cache

- Suppose hit rate is 0.4

## Consequence

- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- Utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
- Total avg delay = Internet delay + access delay + LAN delay =  $0.6 \cdot (2.01) \text{ secs} + 0.4 \cdot \text{msecs} < 1.4 \text{ secs}$

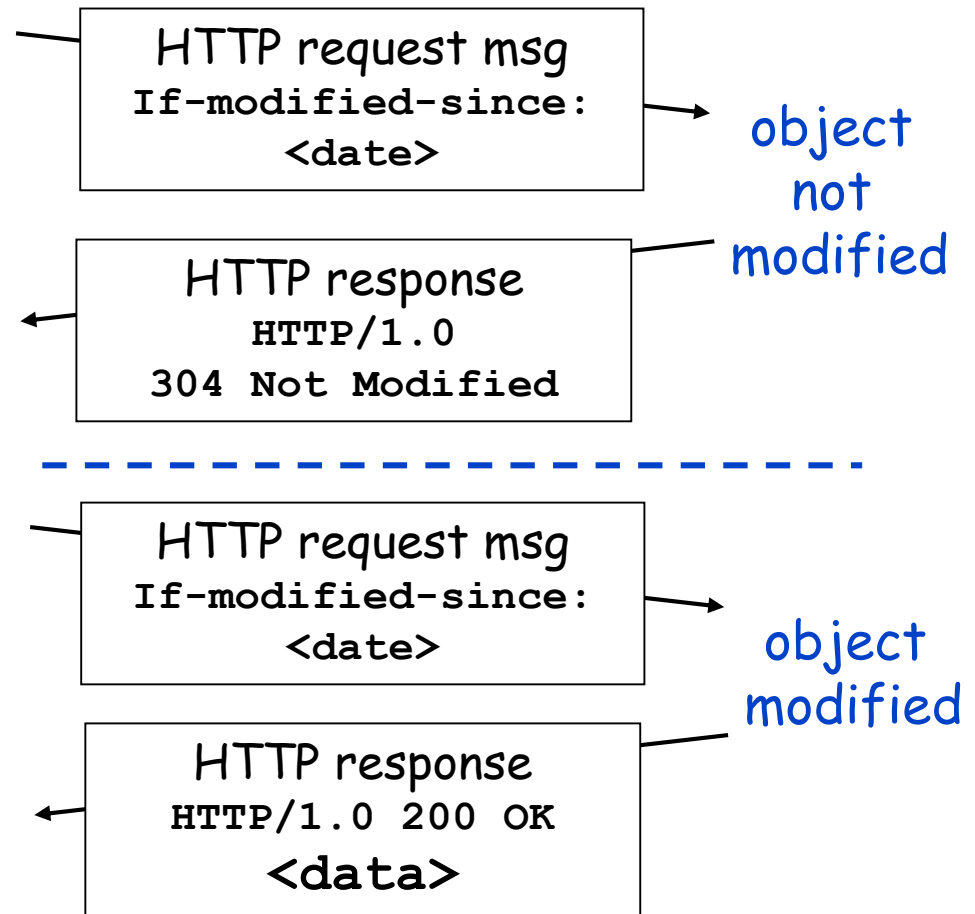


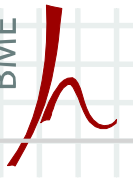
# Conditional GET

- **Goal:** don't send object if cache has up-to-date cached version
- Cache: specify date of cached copy in HTTP request  
`If-modified-since:  
 <date>`
- Server: response contains no object if cached copy is up-to-date:  
`HTTP/1.0 304 Not  
 Modified`

cache

server



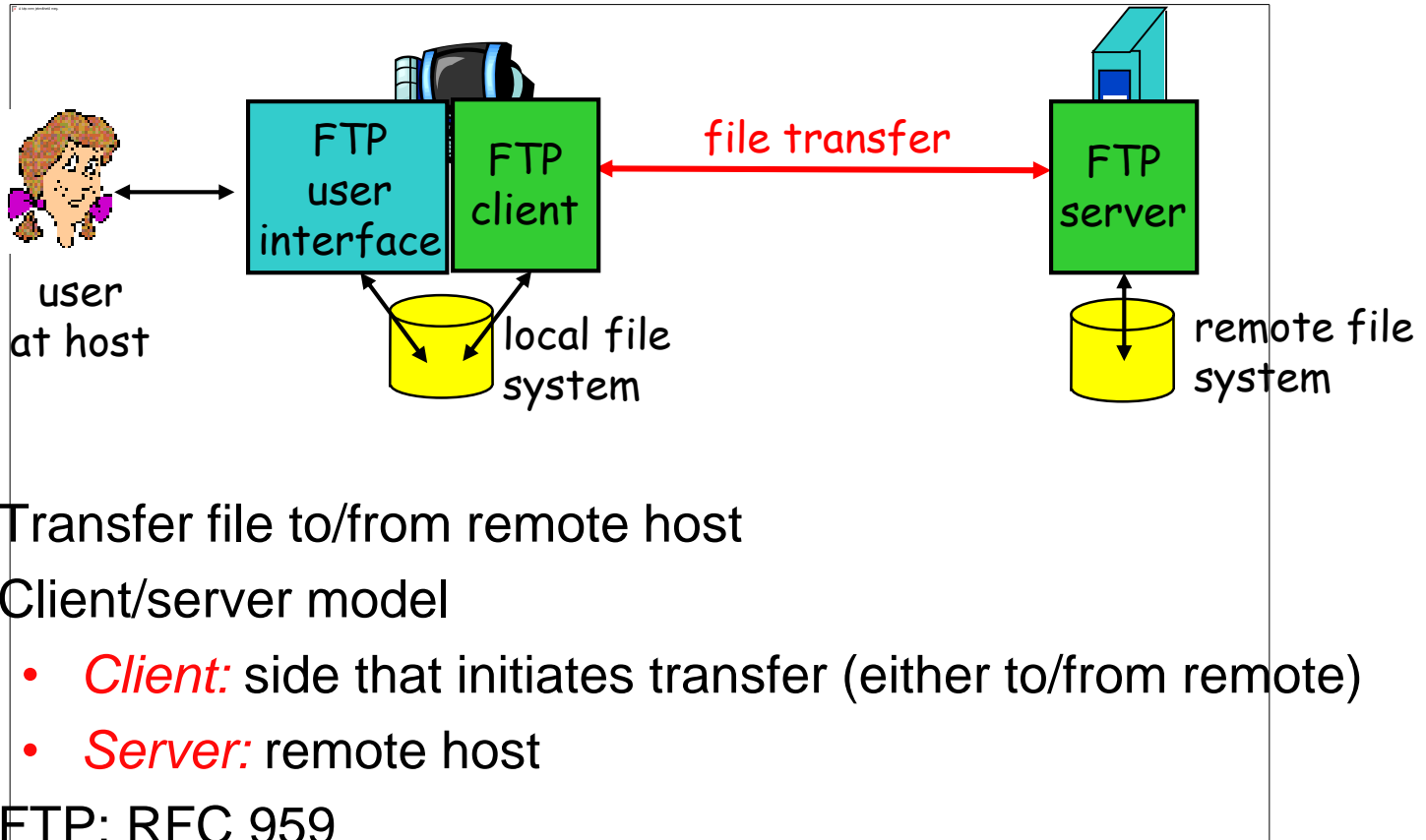


# Chapter 2: Application layer

---

- 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

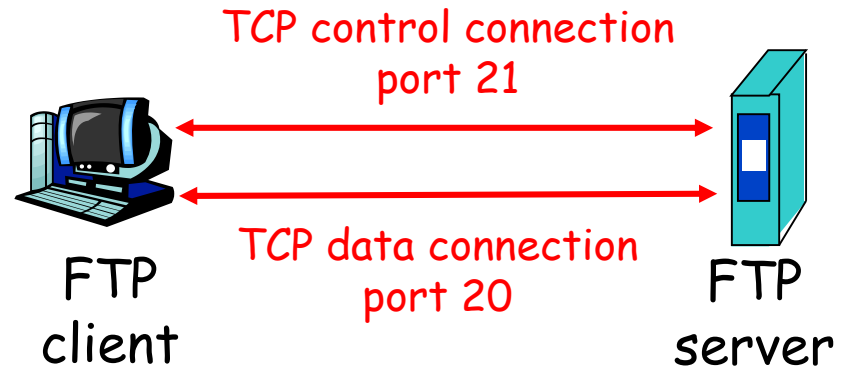
# FTP: The File Transfer Protocol



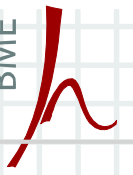
- Transfer file to/from remote host
- Client/server model
  - *Client*: side that initiates transfer (either to/from remote)
  - *Server*: remote host
- FTP: RFC 959
- FTP server: port 21

# FTP: Separate control, data connections

- FTP client contacts FTP server at port 21, specifying TCP as transport protocol
- Client obtains authorization over control connection
- Client browses remote directory by sending commands over control connection
- When server receives file transfer command, server opens 2<sup>nd</sup> TCP connection (for file) to client
- After transferring one file, server closes data connection



- Server opens another TCP data connection to transfer another file
- Control connection: “out of band”
- FTP server maintains “state”: current directory, earlier authentication



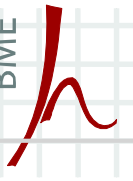
# FTP commands, responses

## Sample commands

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

## Sample return codes

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



# Chapter 2: Application layer

---

- 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

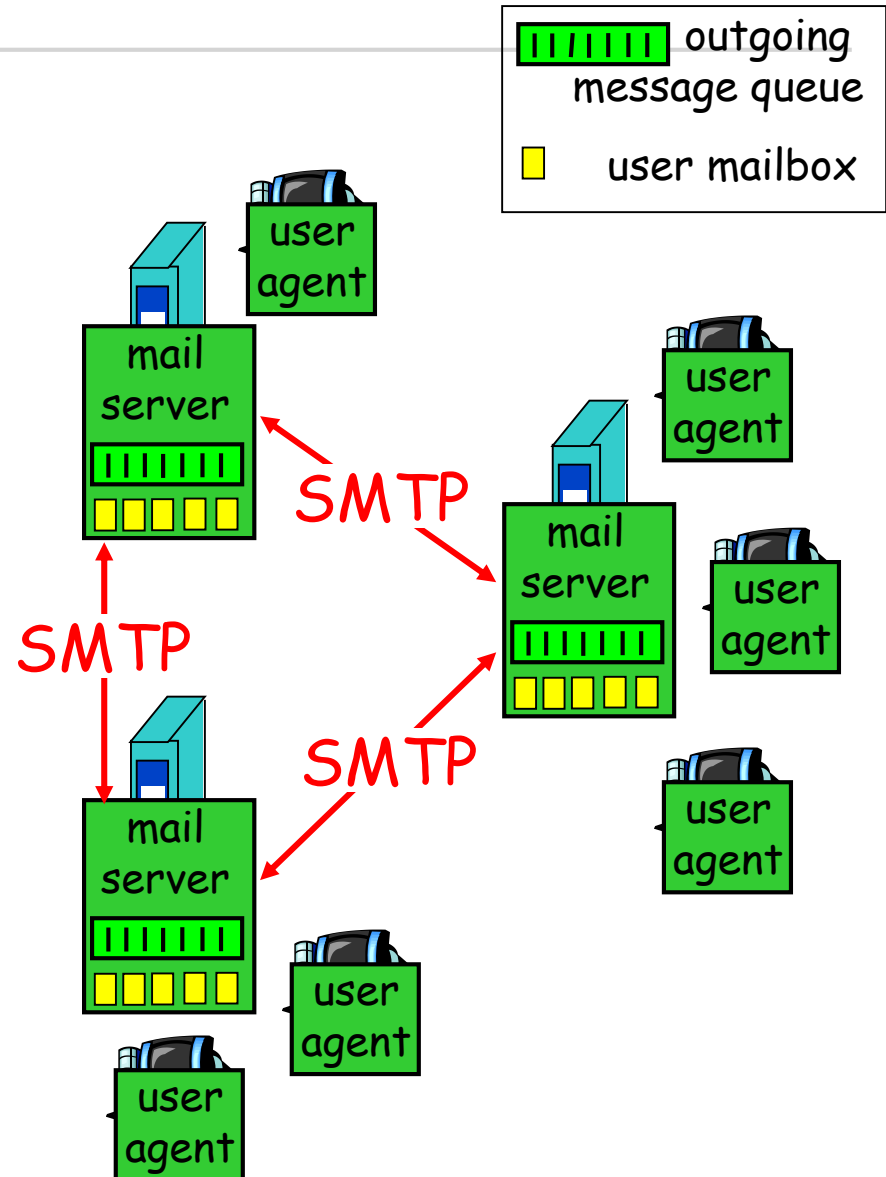
# Electronic mail

## Three major components:

- User agents
- Mail servers
- Simple mail transfer protocol: SMTP

## User Agent

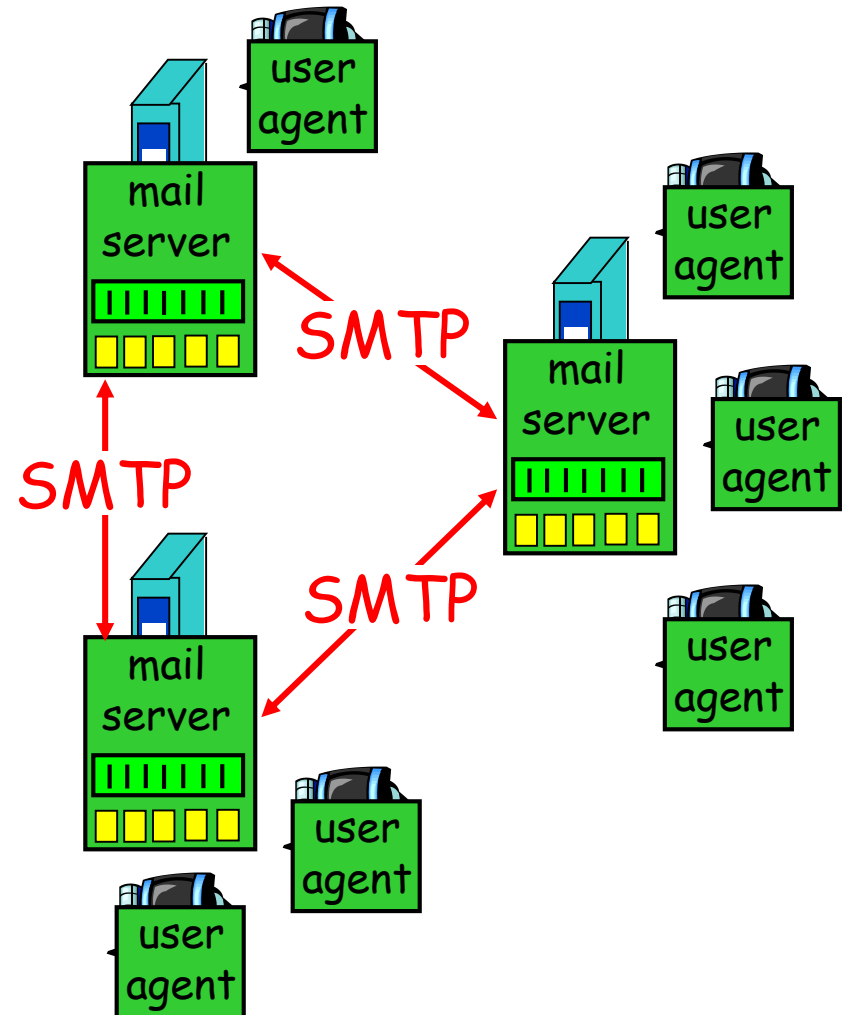
- A.k.a. “mail reader”
- Composing, editing, reading mail messages
- E.g., Eudora, Outlook, elm, Netscape Messenger
- Outgoing, incoming messages stored on server

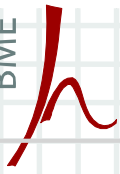


# Electronic mail: Mail servers

## 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



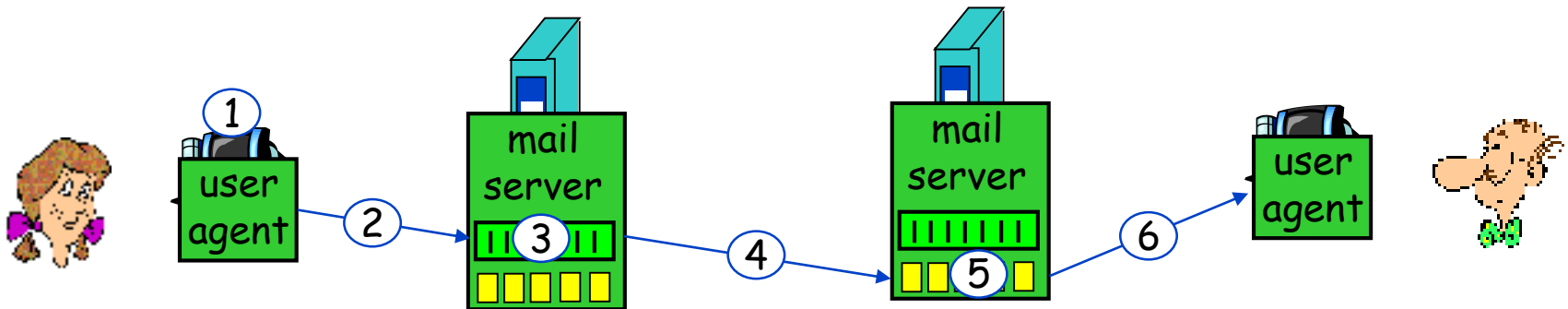


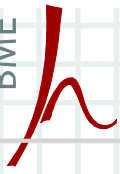
# Electronic mail: SMTP [RFC 2821]

- Uses TCP to reliably transfer email message from client to server, port 25
- Direct transfer: sending server to receiving server
- Three phases of transfer
  - Handshaking (greeting)
  - Transfer of messages
  - Closure
- Command/response interaction
  - **Commands:** ASCII text
  - **Response:** status code and phrase
- Messages must be in 7-bit ASCII

# Scenario: Alice sends message to Bob

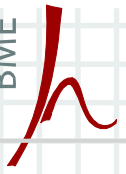
- 1) Alice uses UA to compose message and “to” bob@someschool.edu
- 2) Alice’s UA sends message to her mail server; message placed in message queue
- 3) Client side of SMTP opens TCP connection with Bob’s mail server
- 4) SMTP client sends Alice’s message over the TCP connection
- 5) Bob’s mail server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





# Sample SMTP interaction

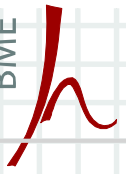
```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like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



# Try SMTP interaction for yourself

- `telnet servername 25`
- See 220 reply from server
- Enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

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



# SMTP: final words

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

## Comparison with HTTP:

- HTTP: pull
- SMTP: push
- Both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg

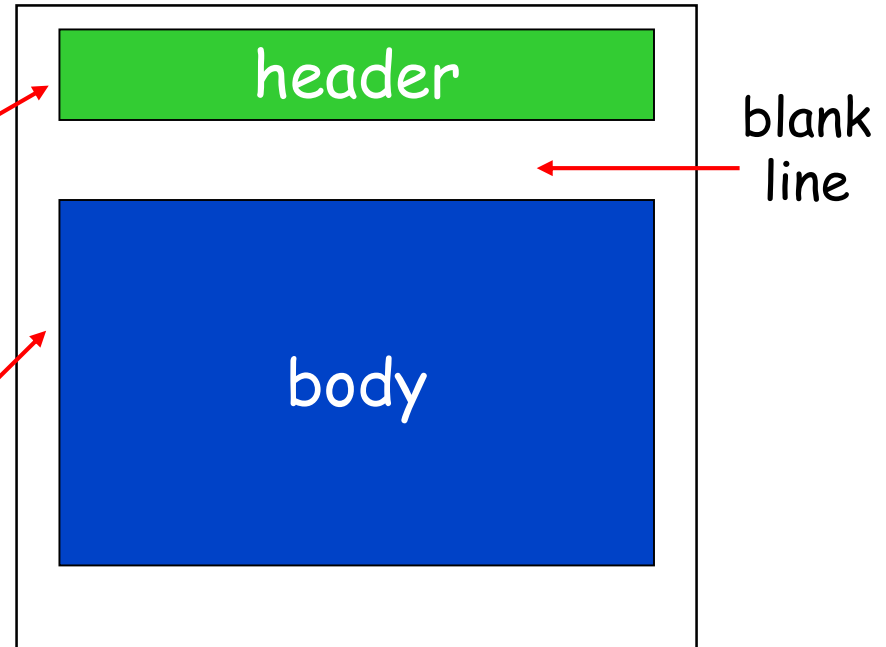
# Mail message format

SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

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

*Different from SMTP commands!*
- Body
  - The “message”, ASCII characters only



# Message format: Multimedia extensions

- MIME: Multimedia mail extension, RFC 2045, 2056
- Additional lines in msg header declare MIME content type

MIME version

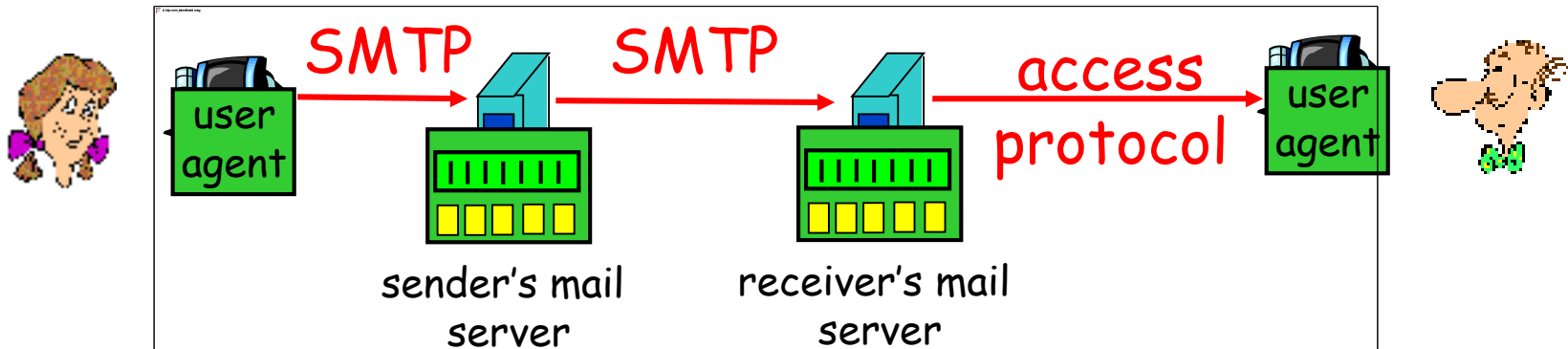
method used  
to encode data

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
```

# Mail access protocols



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

# POP3 protocol

## Authorization phase

- Client commands
  - **user**: declare username
  - **pass**: password
- Server responses
  - **+OK**
  - **-ERR**

## Transaction phase, client

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

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

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



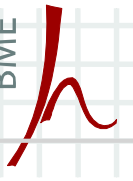
# POP3 (more) and IMAP

## More about POP3

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

## IMAP

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



# Chapter 2: Application layer

---

- 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

# 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:** Map between IP addresses and name?

**Domain Name System**

- *Distributed database* implemented in hierarchy of many *name servers*
- *Application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
  - Note: core Internet function, implemented as application-layer protocol
  - Complexity at network’s “edge”

## DNS services

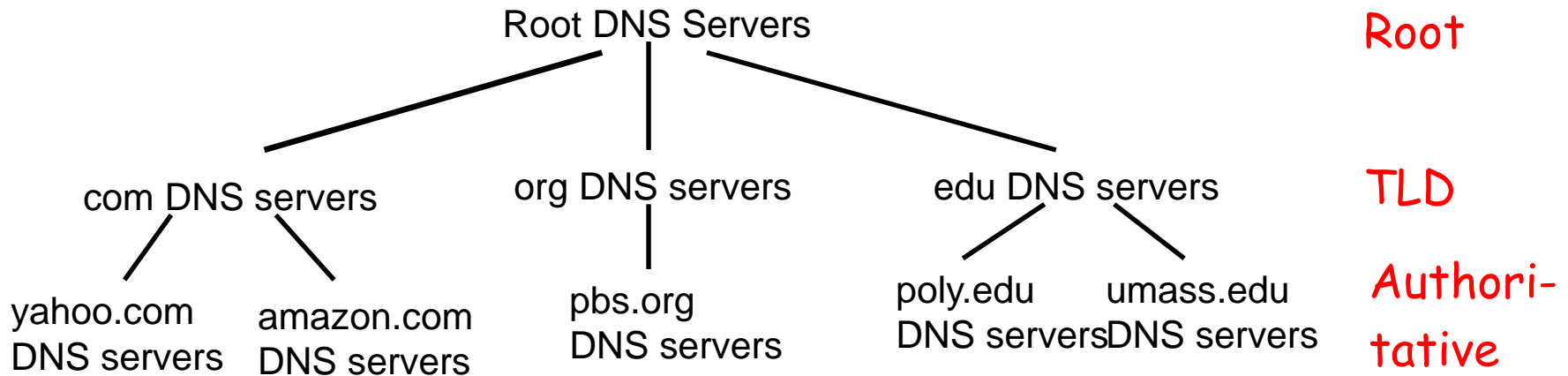
- Hostname to IP address translation
- Host aliasing
  - Canonical and alias names
- Mail server aliasing
- Load distribution
  - Replicated Web servers: set of IP addresses for one canonical name

## Why not centralize DNS?

- Single point of failure
- Traffic volume
- Distant centralized database
- Maintenance

*Doesn't scale!*

# Distributed, hierarchical database

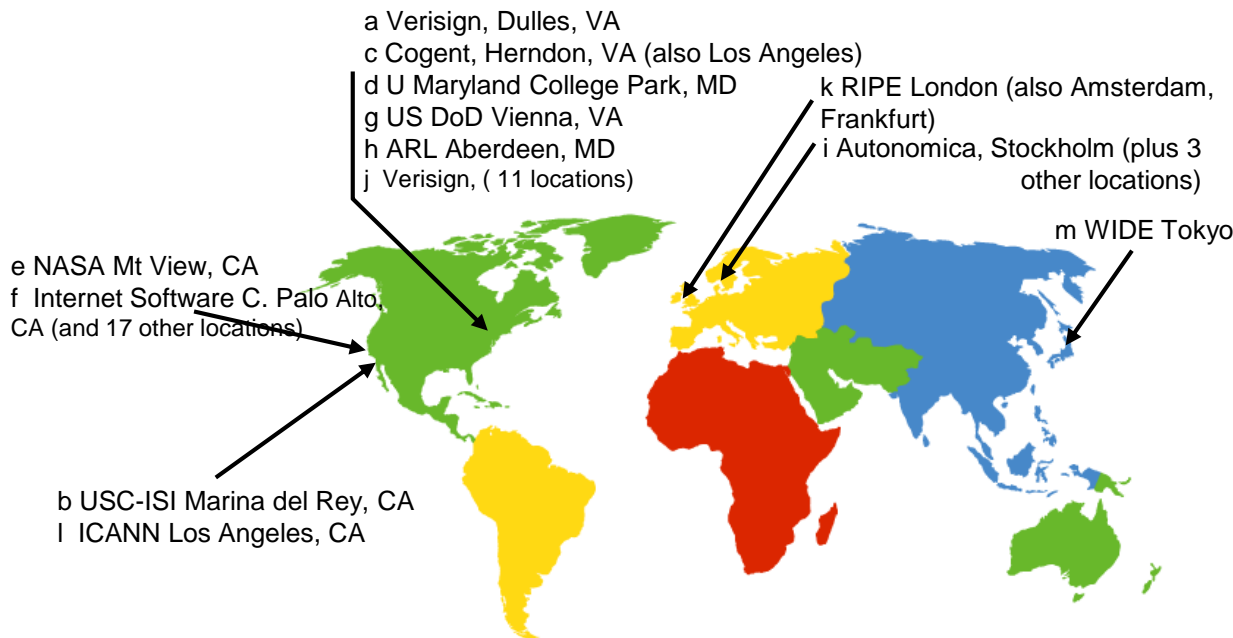


## Client wants IP for [www.amazon.com](http://www.amazon.com); 1<sup>st</sup> approx:

- 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](http://www.amazon.com)

# DNS: Root name servers

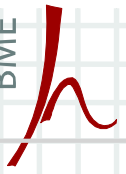
- Contacted by local name server that can not resolve name
- Root name server
  - Contacts authoritative name server if name mapping not known
  - Gets mapping
  - Returns mapping to local name server



13 root name servers worldwide

# TLD and Authoritative servers

- **Top-level domain (TLD) servers:** responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp
  - Network solutions maintains servers for com TLD
  - Educause for edu TLD
- **Authoritative DNS servers:** organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web and mail)
  - Can be maintained by organization or service provider



# Local name server

---

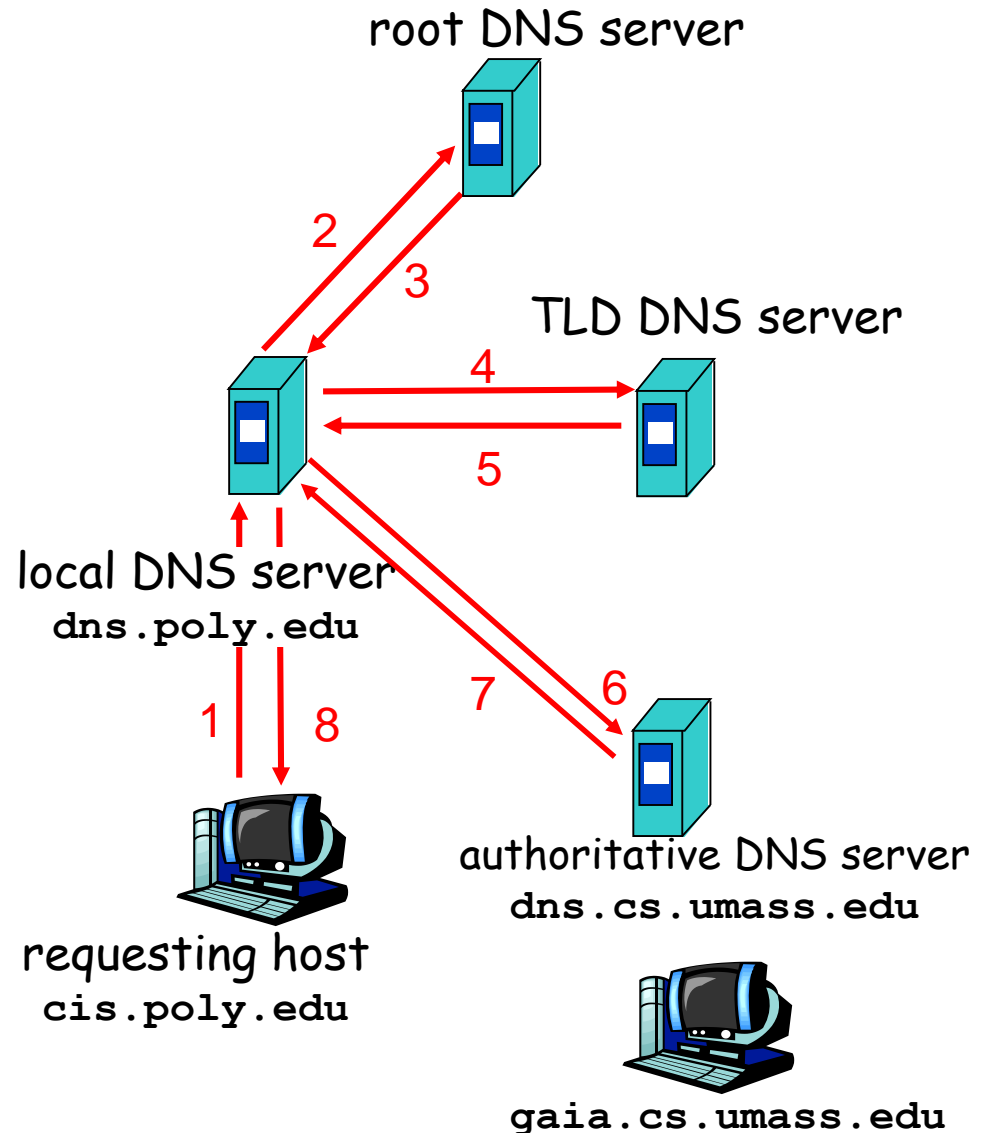
- Does not strictly belong to the hierarchy
- Each ISP (residential ISP, company, university) has one
  - Also called “default name server”
- When a host makes a DNS query, query is sent to its local DNS server
  - Acts as a proxy, forwards query into hierarchy

# Example

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

## Iterated query:

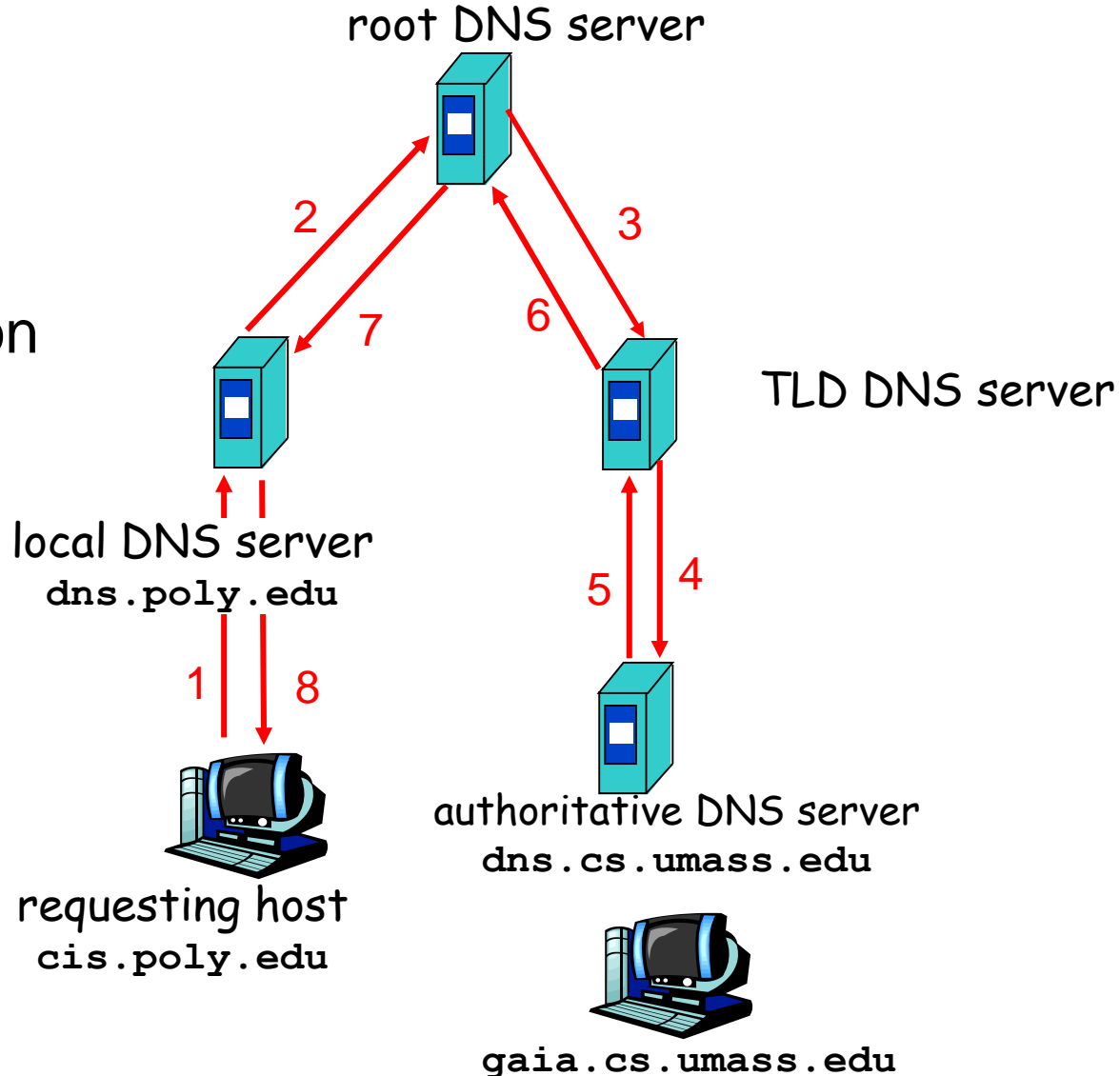
- Contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”



# Recursive queries

## Recursive query:

- Puts burden of name resolution on contacted name server
- Heavy load?



# DNS: Caching and updating records

- Once (any) name server learns mapping, it *caches* mapping
  - Cache entries timeout (disappear) after some time
  - TLD servers typically cached in local name servers
    - Thus root name servers not often visited
- Update/notify mechanisms under design by IETF
  - RFC 2136
  - <http://www.ietf.org/html.charters/dnsind-charter.html>

DNS: distributed db storing resource records (RR)

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

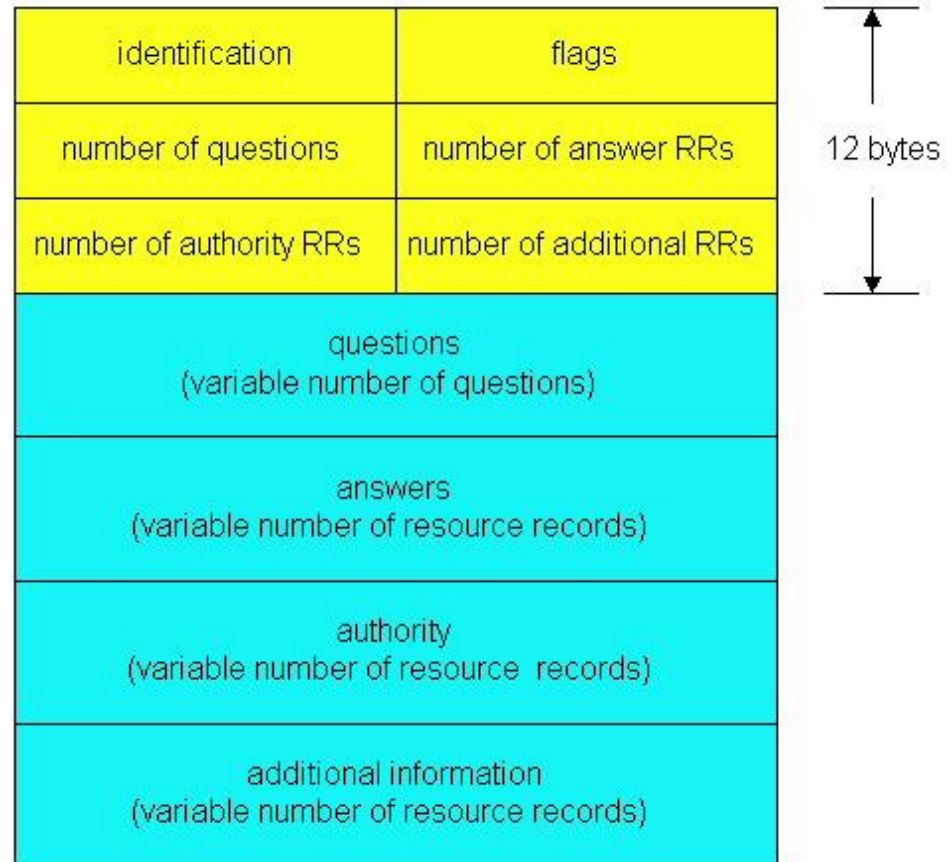
- Type=A
  - **name** is hostname
  - **value** is IP address
- Type=NS
  - **name** is domain (e.g. foo.com)
  - **value** is hostname of authoritative name server for this domain
- Type=CNAME
  - **name** is alias name for some “canonical” (the real) name  
www.ibm.com is really  
servereast.backup2.ibm.com
  - **value** is canonical name
- Type=MX
  - **value** is name of mailserver associated with **name**

# DNS protocol, messages

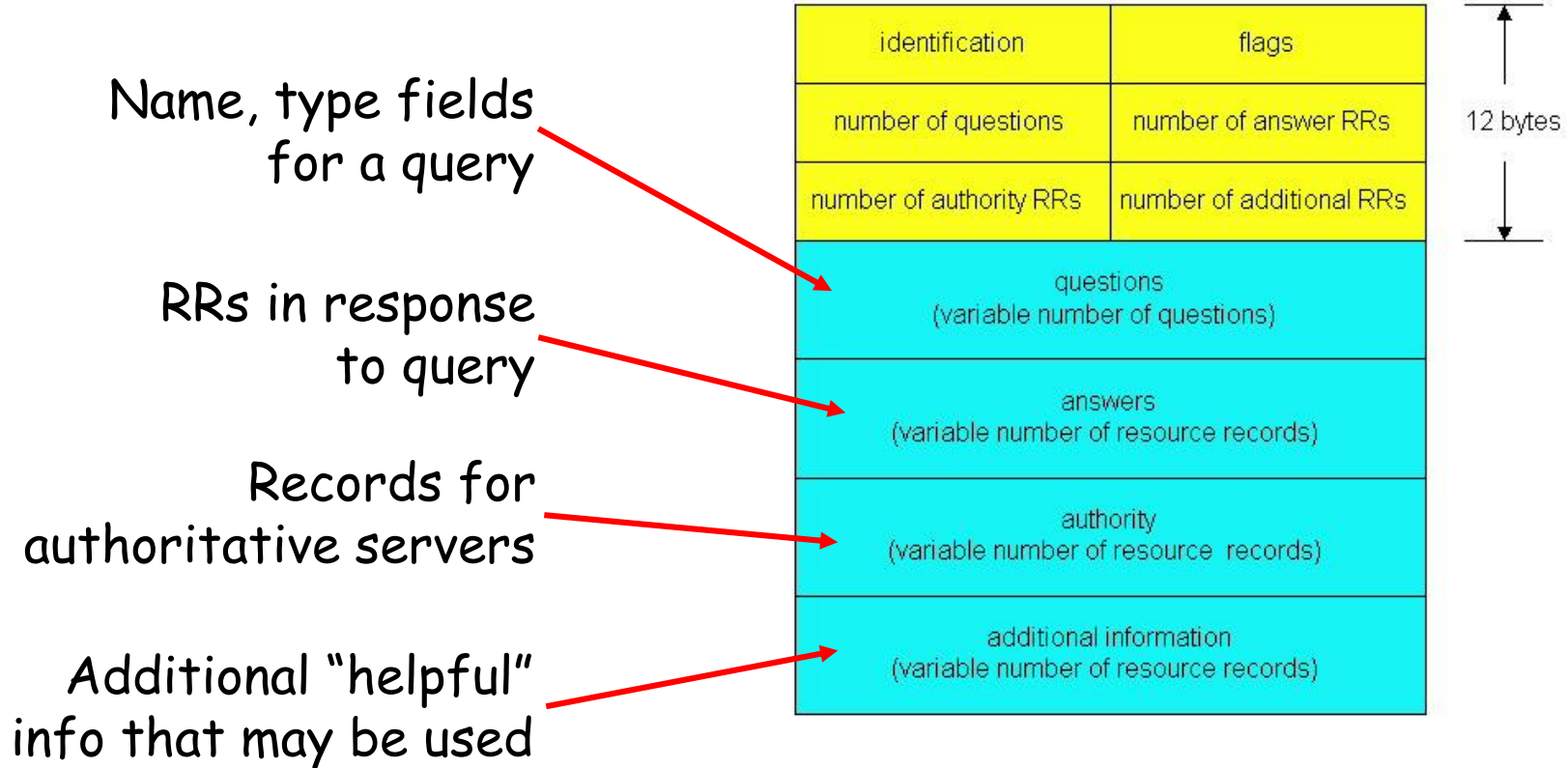
DNS protocol: *query* and *reply* messages, both with the same *message format*

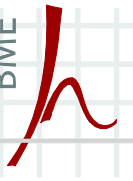
## Msg header

- **Identification**: 16 bit # for query, reply to query uses same #
- **Flags**
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



# DNS protocol, messages





# Inserting records into DNS

- Example: just created startup “Network Utopia”
- Register name networkutopia.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)
  - Registrar inserts two RRs into the com TLD server:

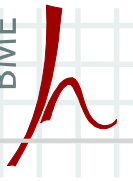
```
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
```

- Put in authoritative server Type A record for `www.networkutopia.com` and Type MX record for `networkutopia.com`
- **How do people get the IP address of your Web site?**

# 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



# P2P file sharing

## Example

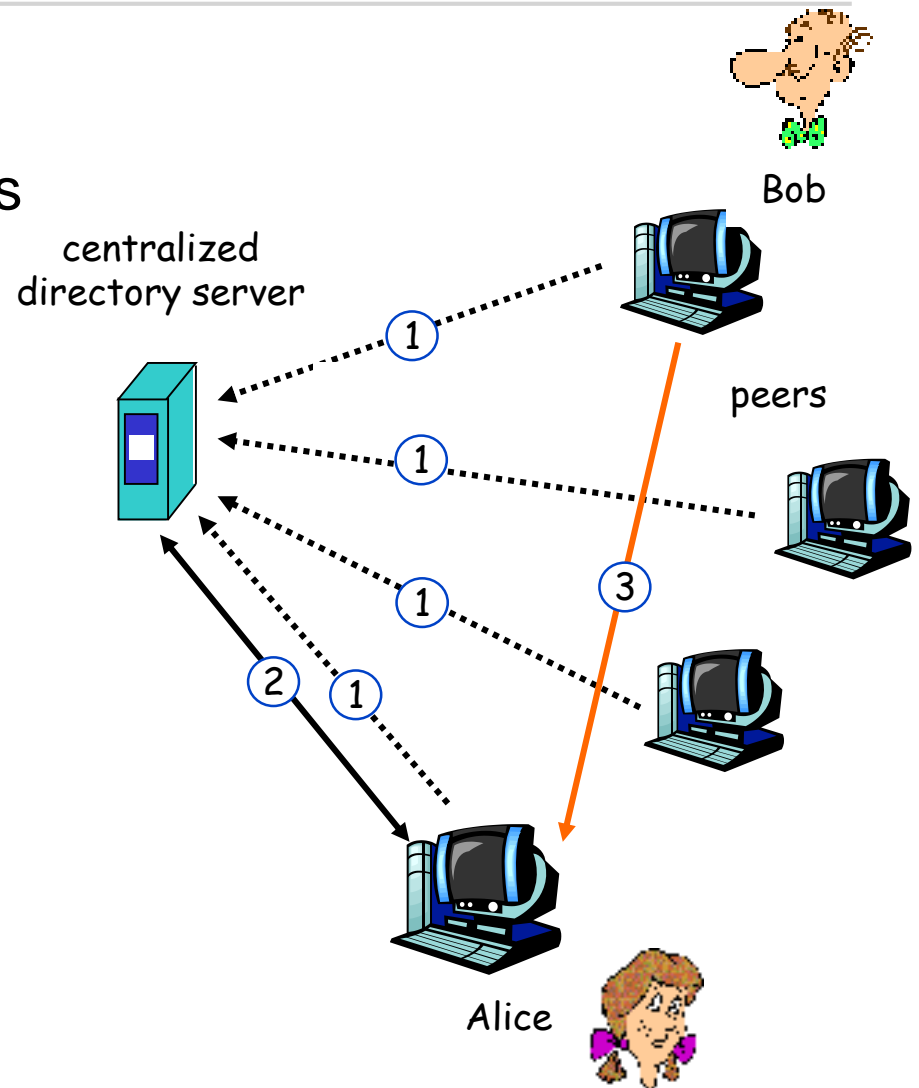
- Alice runs P2P client application on her notebook computer
- Intermittently connects to Internet; gets new IP address for each connection
- Asks for “Hey Jude”
- Application displays other peers that have copy of Hey Jude.
- Alice chooses one of the peers, Bob.
- File is copied from Bob’s PC to Alice’s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

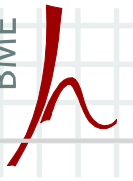
**All peers are servers = highly scalable!**

# 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

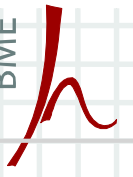




# P2P: Problems with centralized directory

- Single point of failure
- Performance bottleneck
- Copyright infringement

File transfer is decentralized, but locating content is highly centralized



# Query flooding: Gnutella

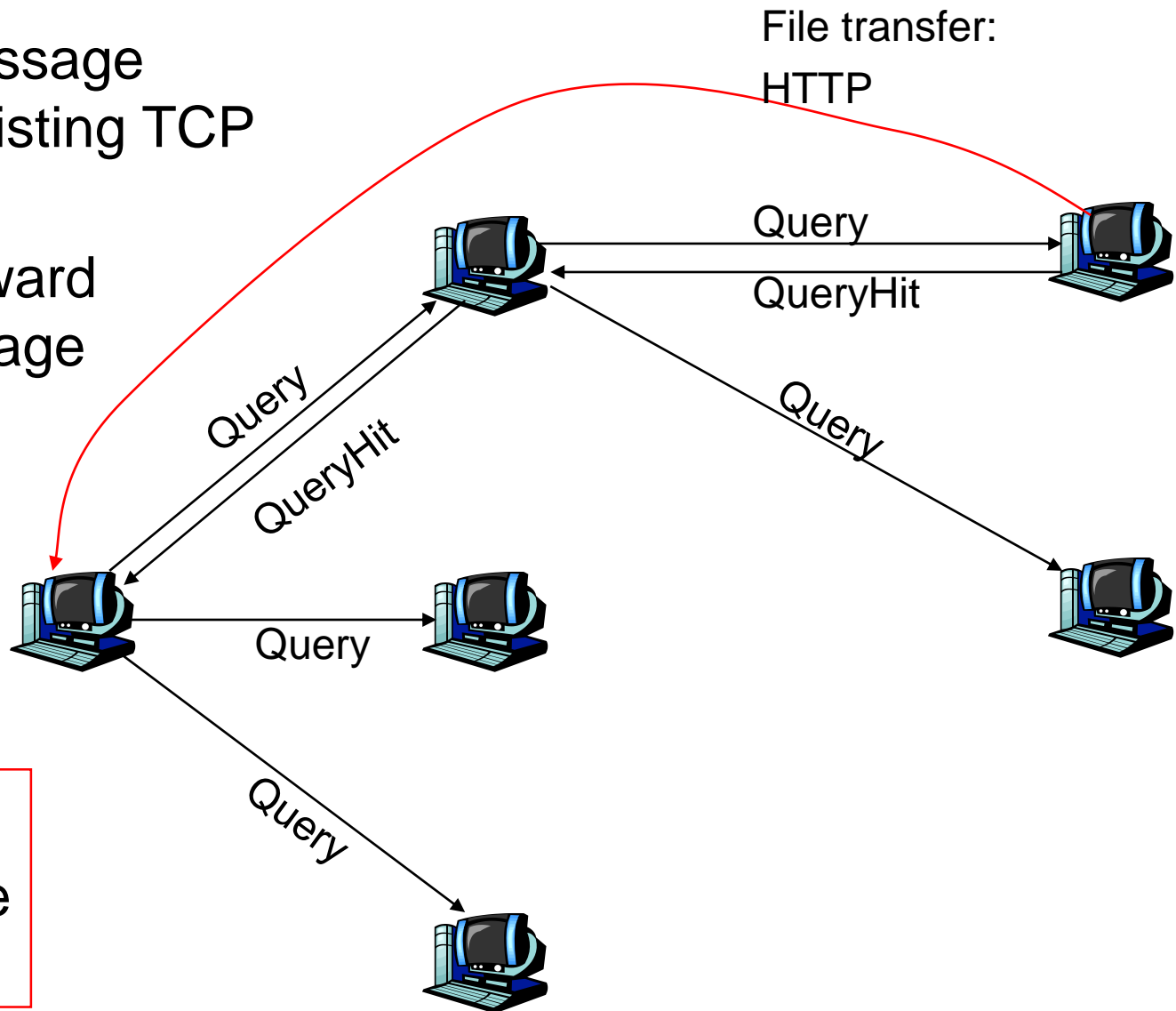
- Fully distributed
  - No central server
- Public domain protocol
- Many Gnutella clients implementing protocol

## Overlay network: Graph

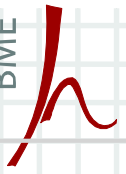
- Edge between peer X and Y if there's a TCP connection
- All active peers and edges is overlay net
- Edge is not a physical link
- Given peer will typically be connected with  $< 10$  overlay neighbors

# Gnutella: Protocol

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

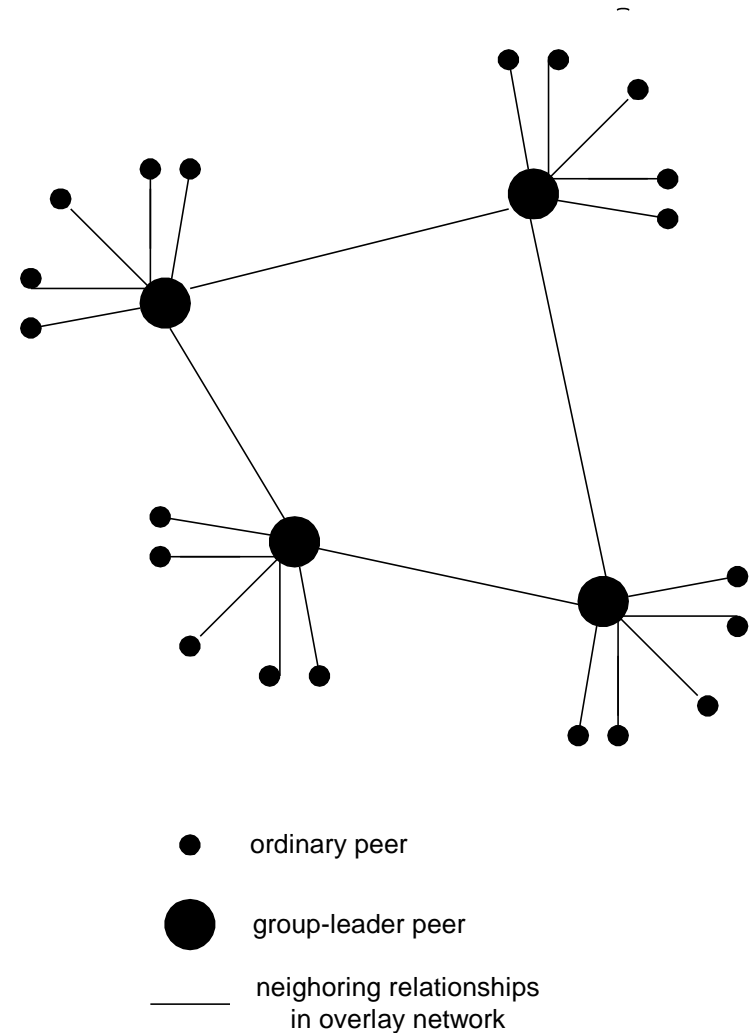


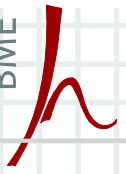
Scalability:  
limited scope  
flooding



# 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

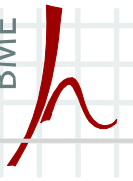
- Each file has a hash and a descriptor
- Client sends keyword query to its group leader
- Group leader responds with matches
  - For each match: metadata, hash, IP address
- If group leader forwards query to other group leaders, they respond with matches
- Client then selects files for downloading
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

# 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



# 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”