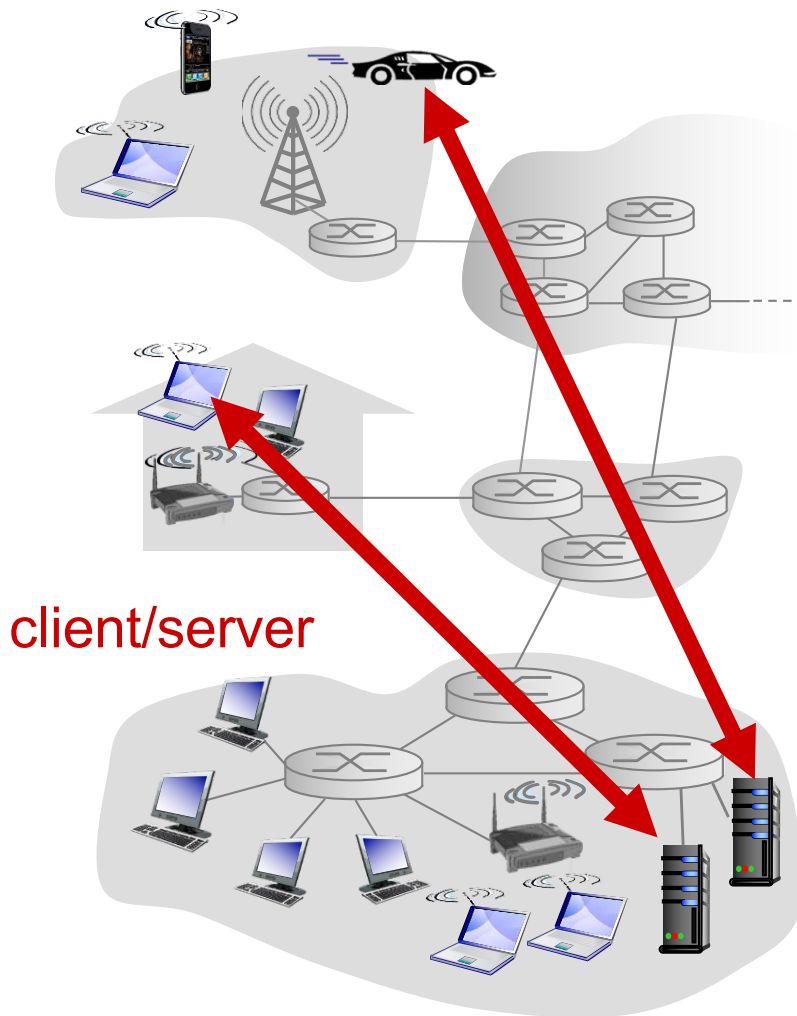


# Logistics

## ❖ Reminder: Assignments

- Web server programming assignment due Sun 10pm
- Chapter 2 and reading HW2 (Ditto)

# Recap: Client-server architecture



## server:

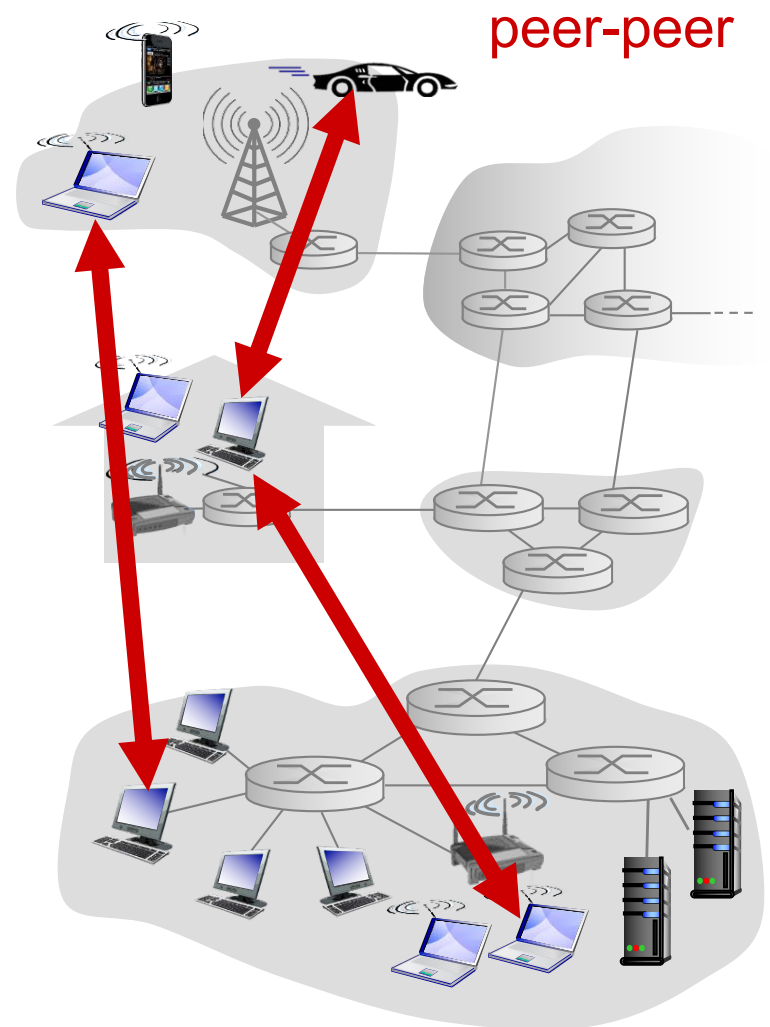
- ❖ always-on host
- ❖ permanent IP address
- ❖ data centers for scaling

## clients:

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

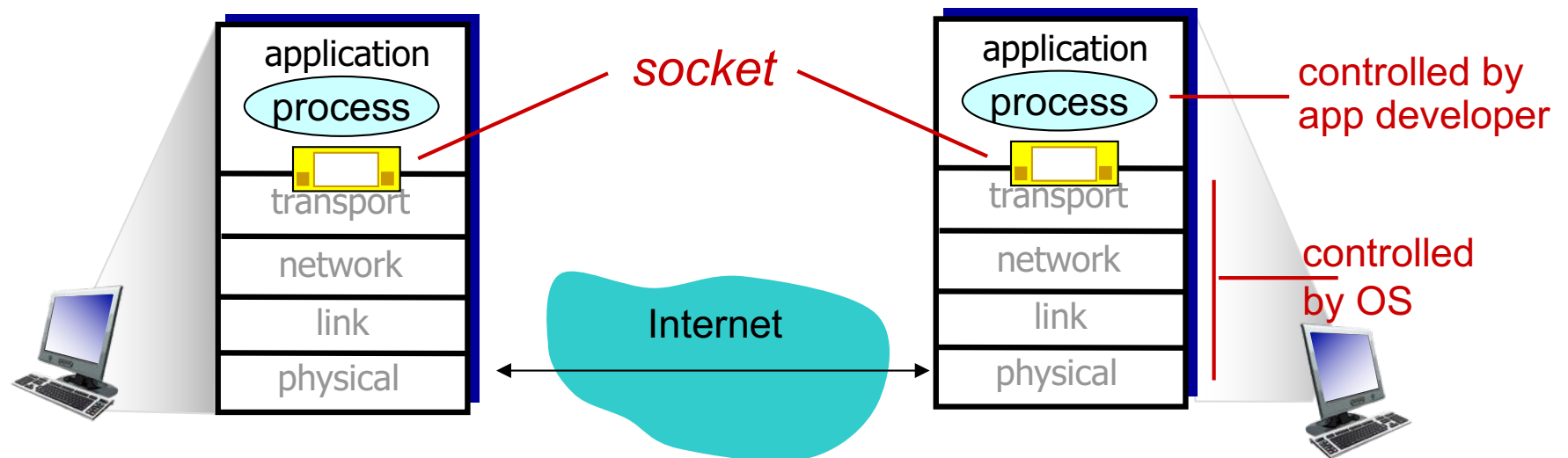
# Recap: P2P architecture

- ❖ *no* always-on server
- ❖ arbitrary end systems directly communicate
- ❖ peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- ❖ peers are intermittently connected and change IP addresses
  - complex management



# Recap: Sockets

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



# Recap: “Packets” vs. “connections”

## ❖ Connection-oriented transport

- Mimics a “serial cable”
- Sends streams of bytes in order, gets them to destination reliably
- Example: TCP



## ❖ Packet-oriented (“datagram”) transport

- Take advantage of the underlying packet network
- Send short, independent messages
- May not get to destination, or get there in order
- Example: UDP

## ❖ Application Examples?

# Web and HTTP

*First, a review...*

- ❖ *web page* consists of *objects*
- ❖ object can be HTML file, JPEG image, Java applet, audio file,...
- ❖ web page consists of *base HTML-file* which includes *several referenced objects*
- ❖ each object is addressable by a *URL*, e.g.,

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

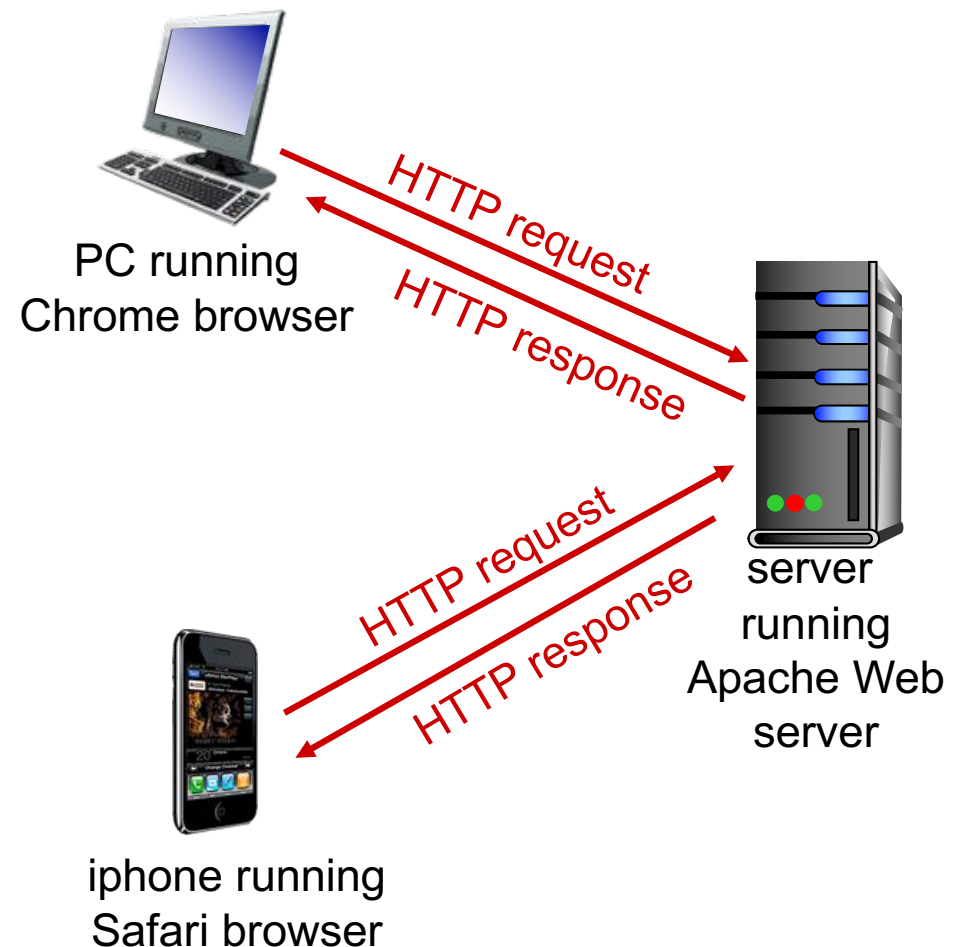
host name

path name

# HTTP overview

## HTTP: hypertext transfer protocol

- ❖ Web's application layer protocol
- ❖ client/server model
  - **client:** browser that requests, receives, (using HTTP protocol) and “displays” Web objects
  - **server:** Web server sends (using HTTP protocol) objects in response to requests



# HTTP overview (continued)

## *uses TCP:*

- ❖ client initiates TCP connection (creates socket) to server, port 80
- ❖ server accepts TCP connection from client
- ❖ HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- ❖ TCP connection closed

## *HTTP is “stateless”*

- ❖ server maintains no information about past client requests

## *aside* protocols that maintain “state” are complex!

- ❖ past history (state) must be maintained
- ❖ if server/client crashes, their views of “state” may be inconsistent, must be reconciled



# HTTP connections

## *non-persistent HTTP*

- ❖ at most one object sent over TCP connection
  - connection then closed
- ❖ downloading multiple objects required multiple connections

## *persistent HTTP*

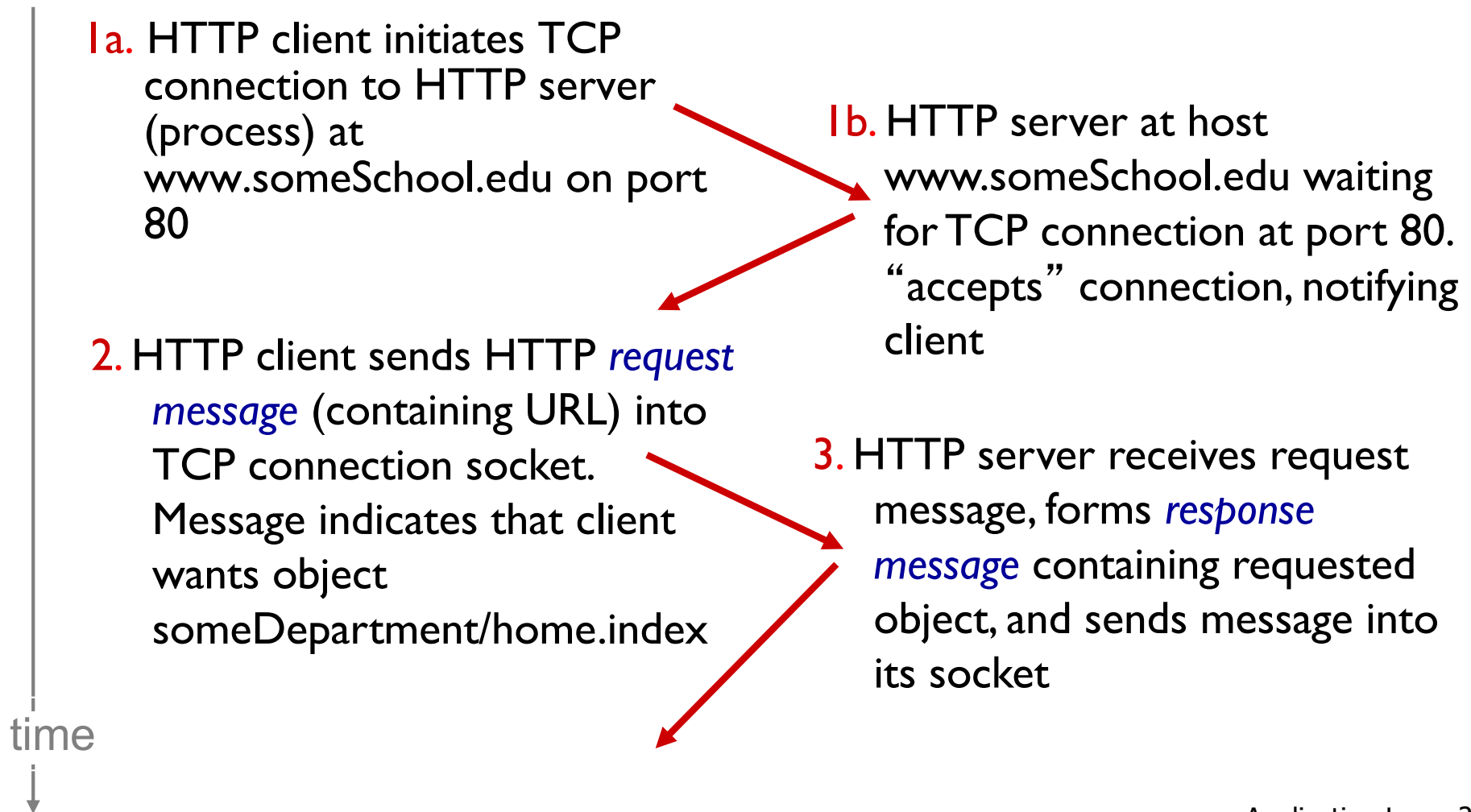
- ❖ multiple objects can be sent over single TCP connection between client, server

# Non-persistent HTTP

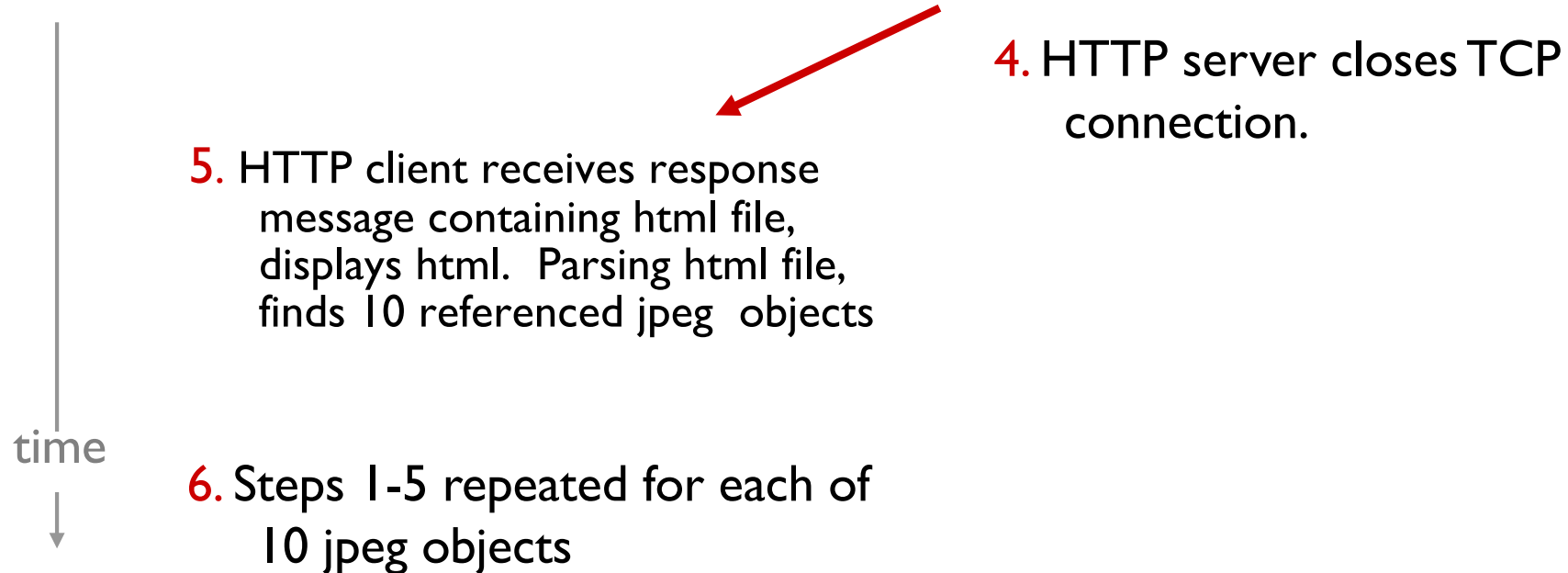
suppose user enters URL:

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

(contains text,  
references to 10  
jpeg images)



# Non-persistent HTTP (cont.)

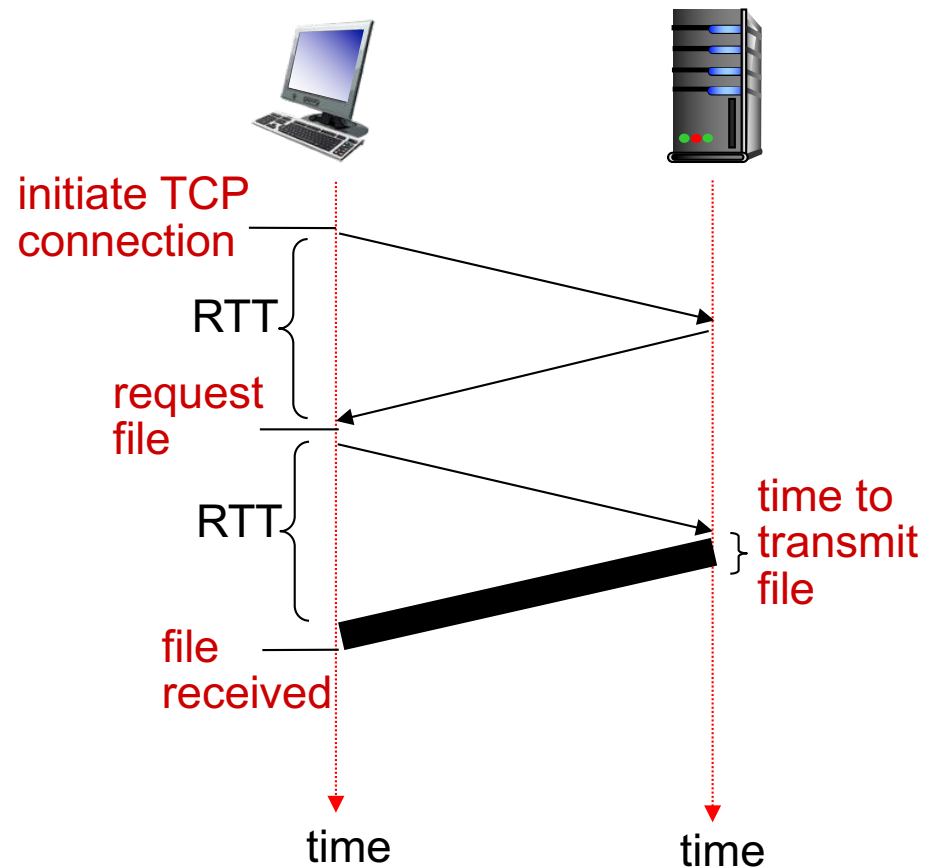


# Non-persistent HTTP: response time

**RTT (definition):** time for a small packet to travel from client to server and back

**HTTP response time:**

- ❖ one RTT to initiate TCP connection
- ❖ one RTT for HTTP request and first few bytes of HTTP response to return
- ❖ file transmission time
- ❖ non-persistent HTTP response time =  
 $2\text{RTT} + \text{file transmission time}$



# Persistent HTTP

## *non-persistent HTTP issues:*

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

## *persistent HTTP:*

- ❖ server leaves connection open after sending response
- ❖ subsequent HTTP messages between same client/server sent over open connection
- ❖ client sends requests as soon as it encounters a referenced object
- ❖ as little as one RTT for all the referenced objects

# HTTP request message

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

request line  
(GET, POST,  
HEAD commands)

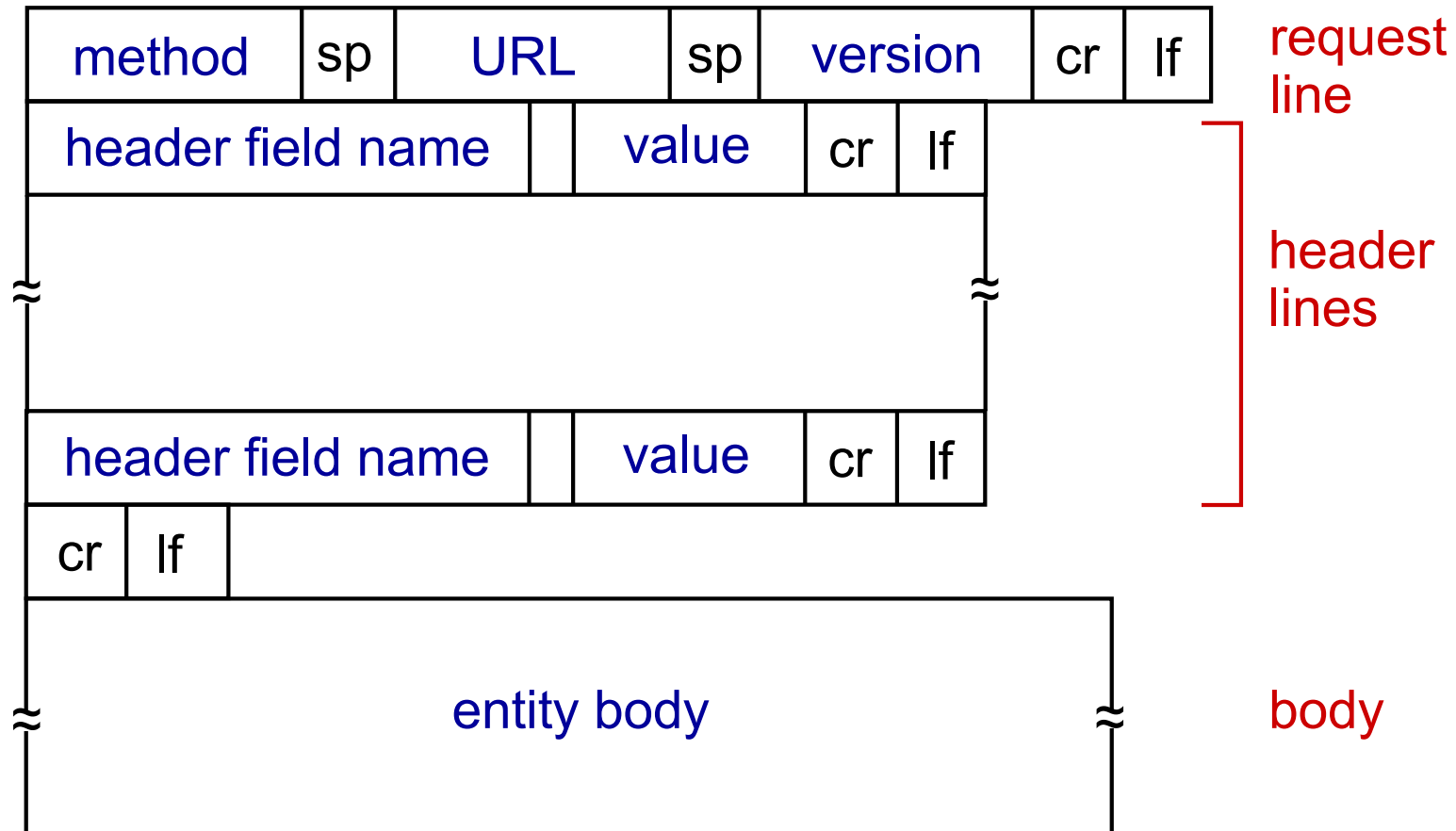
header  
lines

carriage return,  
line feed at start  
of line indicates  
end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

carriage return character  
line-feed character

# HTTP request message: general format



# Uploading form input

## POST method:

- ❖ web page often includes form input
- ❖ input is uploaded to server in entity body

## URL method:

- ❖ uses GET method
- ❖ input is uploaded in URL field of request line:

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



# Method types

## HTTP/1.0:

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

## HTTP/1.1:

- ❖ GET, POST, HEAD
- ❖ PUT
  - uploads file in entity body to path specified in URL field
- ❖ DELETE
  - deletes file specified in the URL field

# HTTP response message

status line  
(protocol  
status code  
status phrase)

header  
lines

data, e.g.,  
requested  
HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
1\r\n
\r\n
data data data data data ...
```

# HTTP response status codes

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

## **200 OK**

- request succeeded, requested object later in this msg

## **301 Moved Permanently**

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

## **400 Bad Request**

- request msg not understood by server

## **404 Not Found**

- requested document not found on this server

## **505 HTTP Version Not Supported**

# Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet 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!

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

# User-server state: cookies

many Web sites use cookies

*four components:*

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

*example:*

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

# Cookies: keeping “state” (cont.)

client



server



usual http request msg

Amazon server  
creates ID  
1678 for user

create  
entry

backend  
database



usual http response  
**set-cookie: 1678**

usual http request msg  
**cookie: 1678**

cookie-  
specific  
action

access

usual http response msg

one week later:



usual http request msg  
**cookie: 1678**

cookie-  
specific  
action

access

usual http response msg

# Cookies (continued)

*what cookies can be used for:*

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

*cookies and privacy:* aside

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

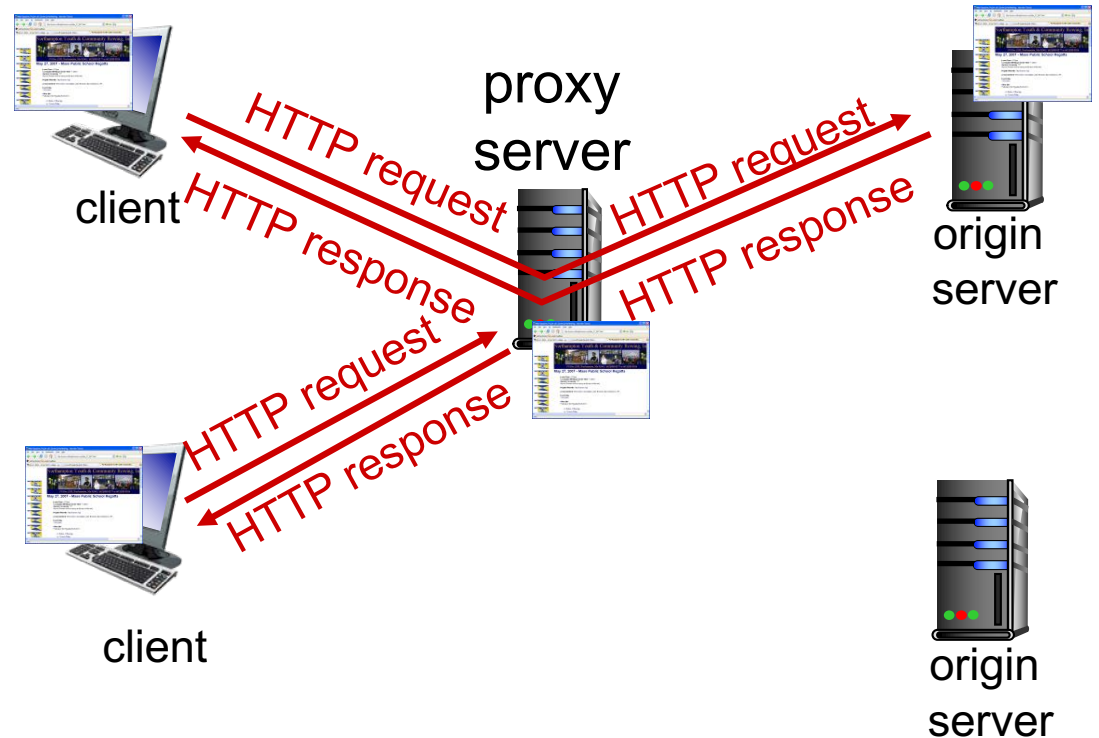
*how to keep “state”:*

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

# Web caches (proxy server)

*goal:* satisfy client request without involving origin server

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





# More about Web caching

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

## *why Web caching?*

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

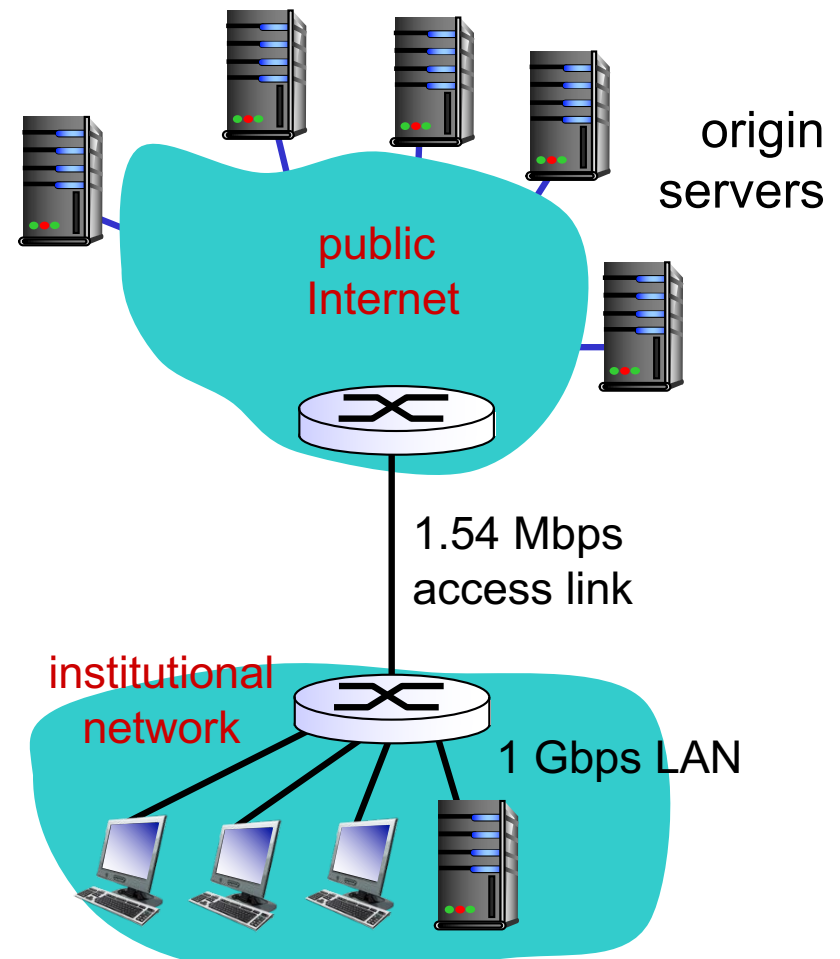
# Caching example:

## *assumptions:*

- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

## *consequences:*

- ❖ LAN utilization: 15%
- ❖ access link utilization = **99%** *problem!*
- ❖ total delay = Internet delay + access delay + LAN delay  
= 2 sec + minutes + usecs



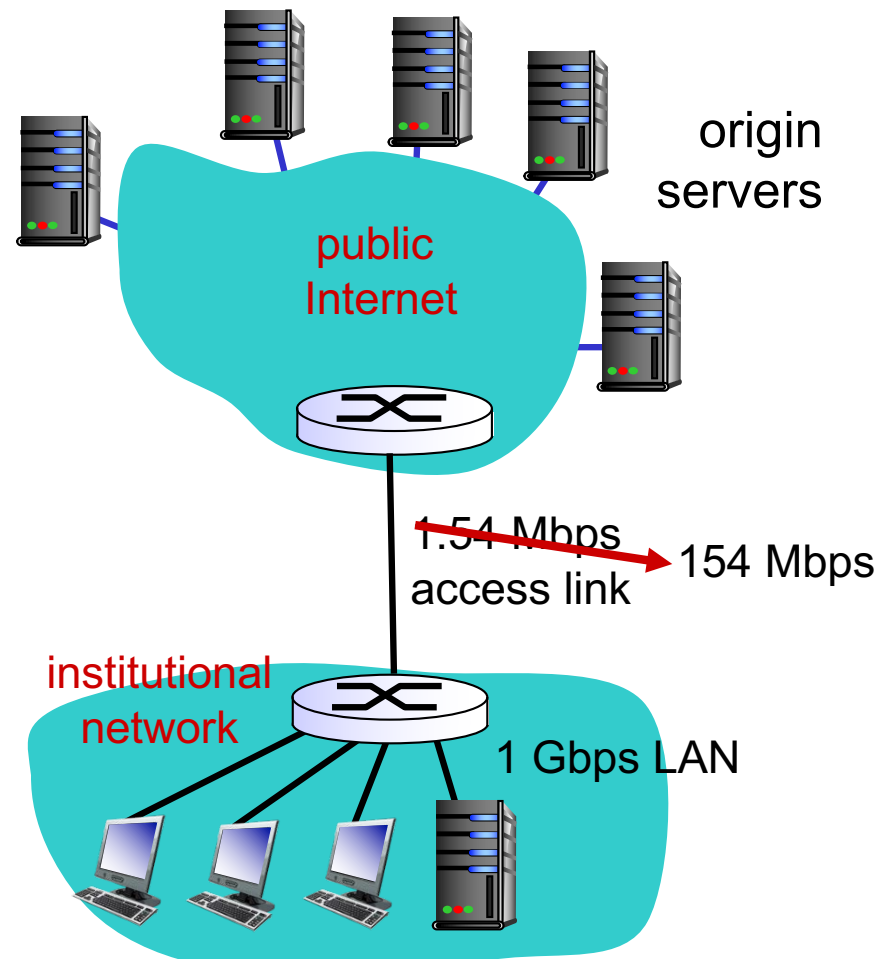
# Caching example: fatter access link

## *assumptions:*

- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: ~~1.54 Mbps~~ → 154 Mbps

## *consequences:*

- ❖ LAN utilization: 15%
- ❖ access link utilization = ~~99%~~ → 9.9%
- ❖ total delay = Internet delay + access delay + LAN delay  
= 2 sec + ~~minutes~~ → msecs



**Cost:** increased access link speed (not cheap!)

# Caching example: install local cache

## *assumptions:*

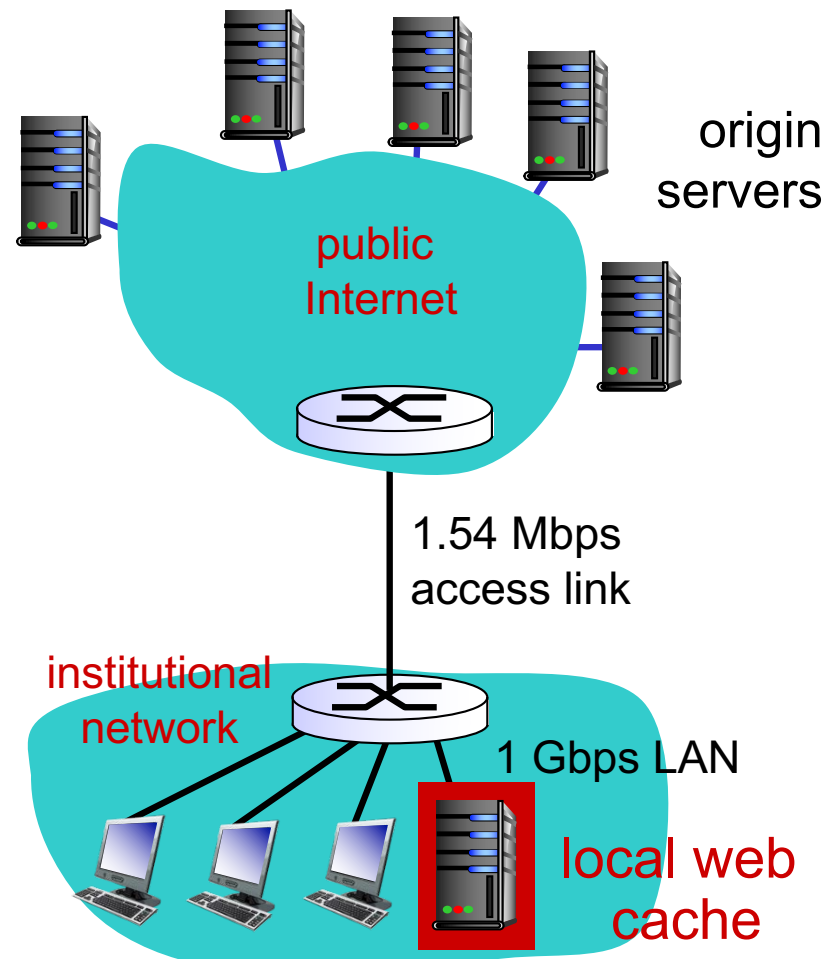
- ❖ avg object size: 100K bits
- ❖ avg request rate from browsers to origin servers: 15/sec
- ❖ avg data rate to browsers: 1.50 Mbps
- ❖ RTT from institutional router to any origin server: 2 sec
- ❖ access link rate: 1.54 Mbps

## *consequences:*

- ❖ LAN utilization: 15%
- ❖ access link utilization = ?
- ❖ total delay = ?

*How to compute link utilization, delay?*

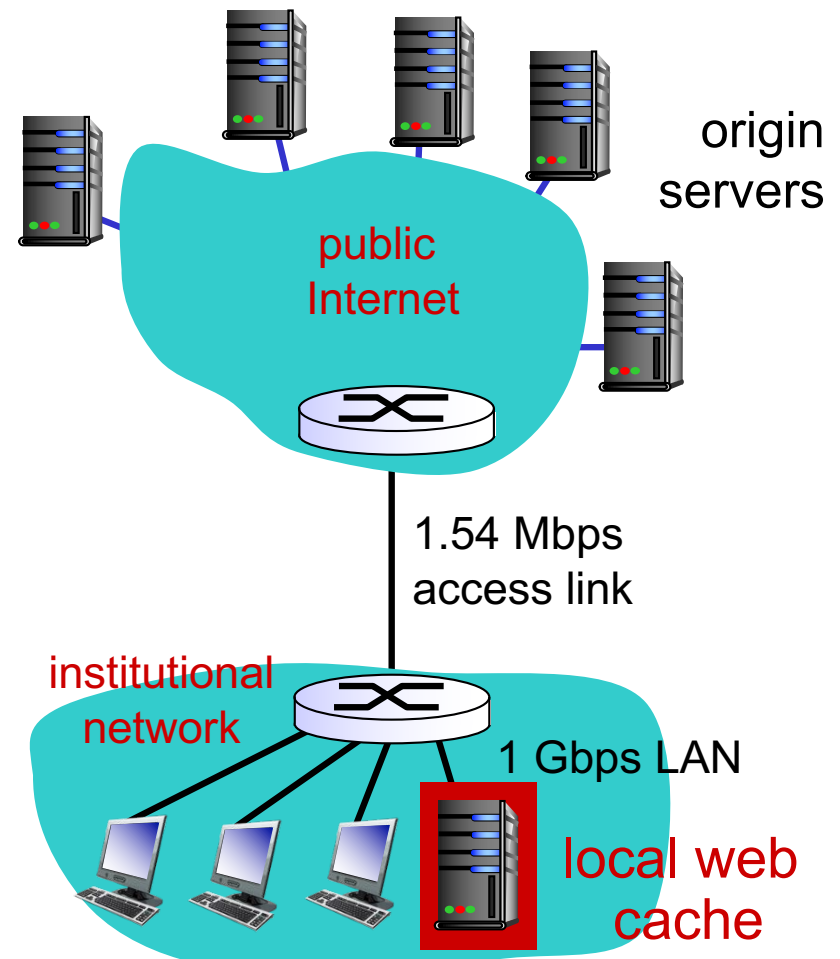
*Cost:* web cache (cheap!)



# Caching example: install local cache

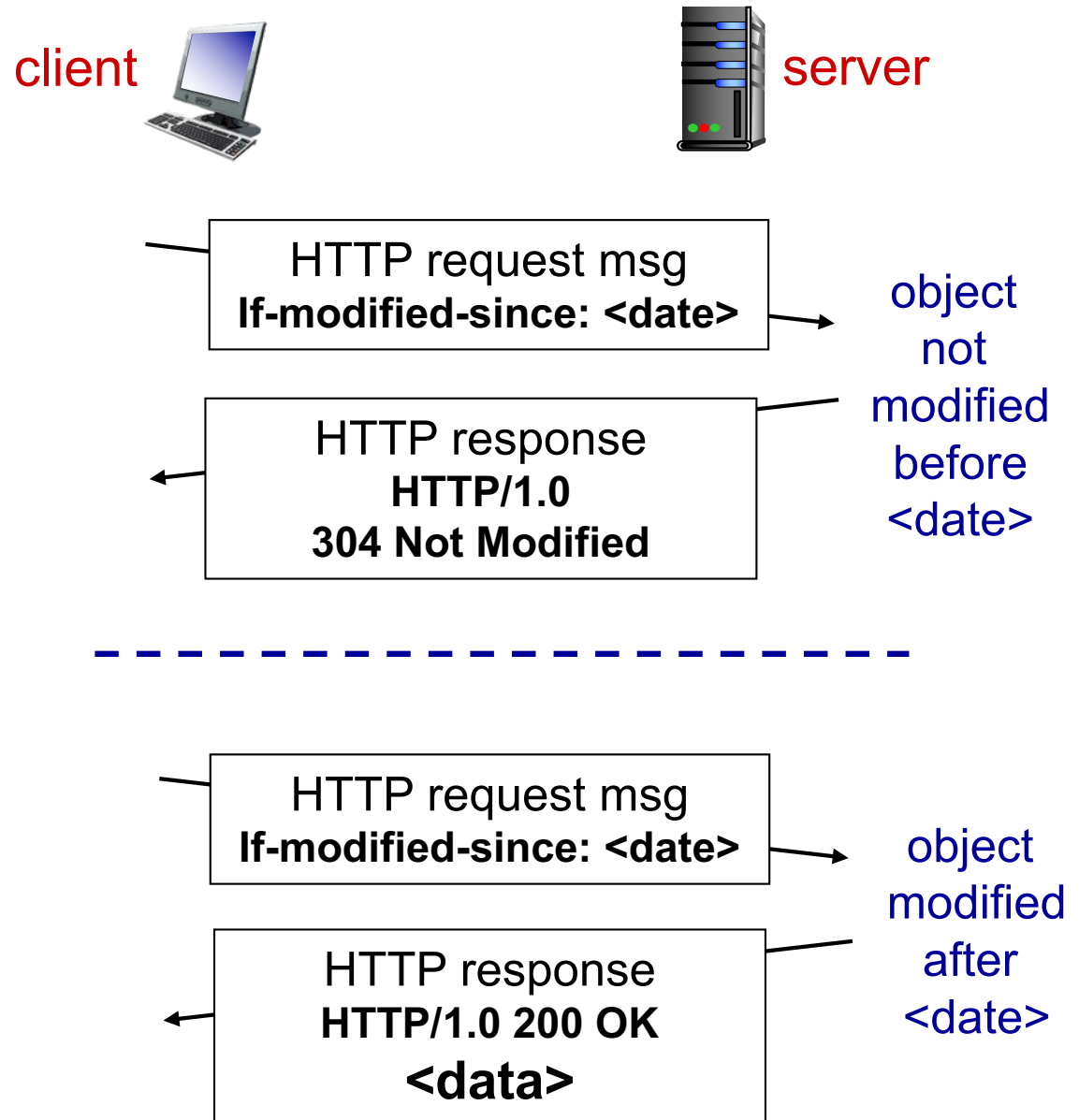
## *Calculating access link utilization, delay with cache:*

- ❖ suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin
- ❖ access link utilization:
  - 60% of requests use access link
- ❖ data rate to browsers over access link =  $0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$ 
  - utilization =  $0.9 / 1.54 = .58$
- ❖ total delay
  - =  $0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
  - =  $0.6 (2.01) + 0.4 (\sim \text{msecs})$
  - =  $\sim 1.2 \text{ secs}$
  - less than with 154 Mbps link (and cheaper too!)

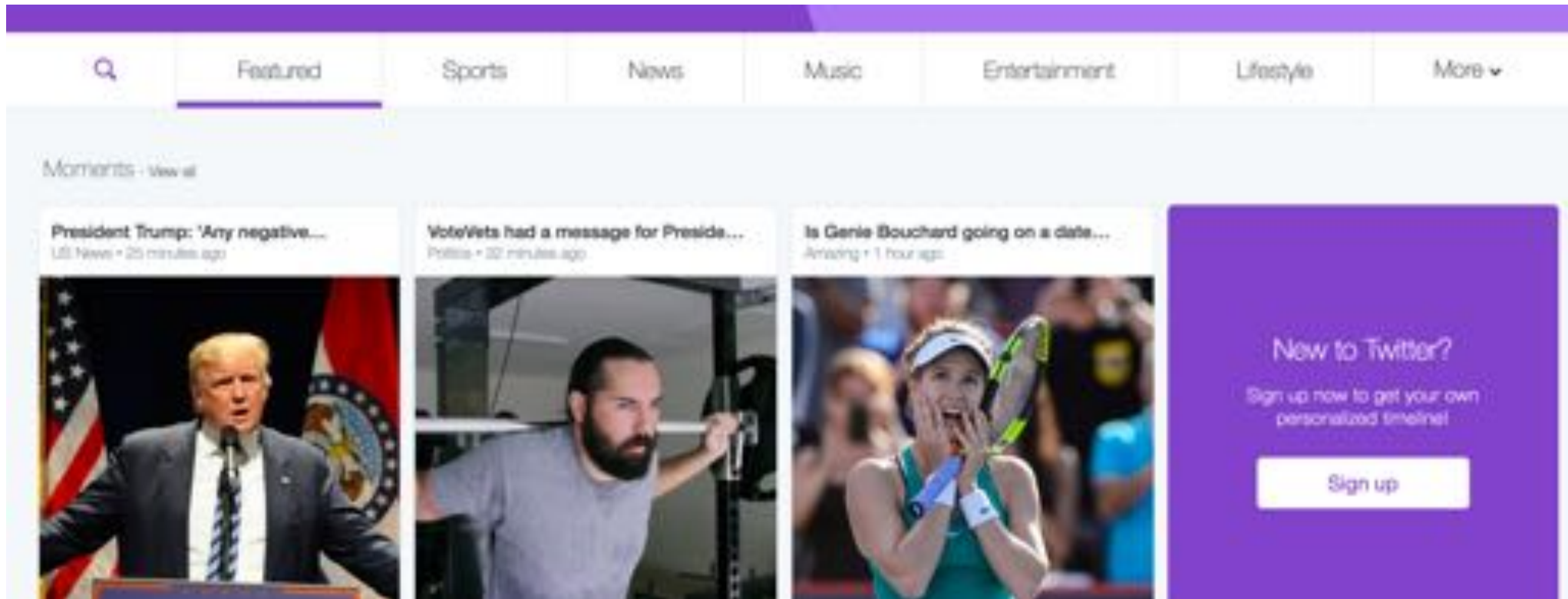


# Conditional GET

- ❖ **Goal:** don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- ❖ **cache:** specify date of cached copy in HTTP request  
**If-modified-since:**  
**<date>**
- ❖ **server:** response contains no object if cached copy is up-to-date:  
**HTTP/1.0 304 Not Modified**



# AJAX (REST)



# AJAX (REST)

- ❖ Use HTTP to operate *live applications* asynchronously
- ❖ Usually Javascript apps (running in a browser)
  - E.g., Facebook.com, Twitter.com, Etc.
  - But can be native mobile apps as well
- ❖ Basic idea:
  - Javascript code loads in browser
  - When you do something, it sends an XML or JSON request back to the server
  - Server responds with a data structure



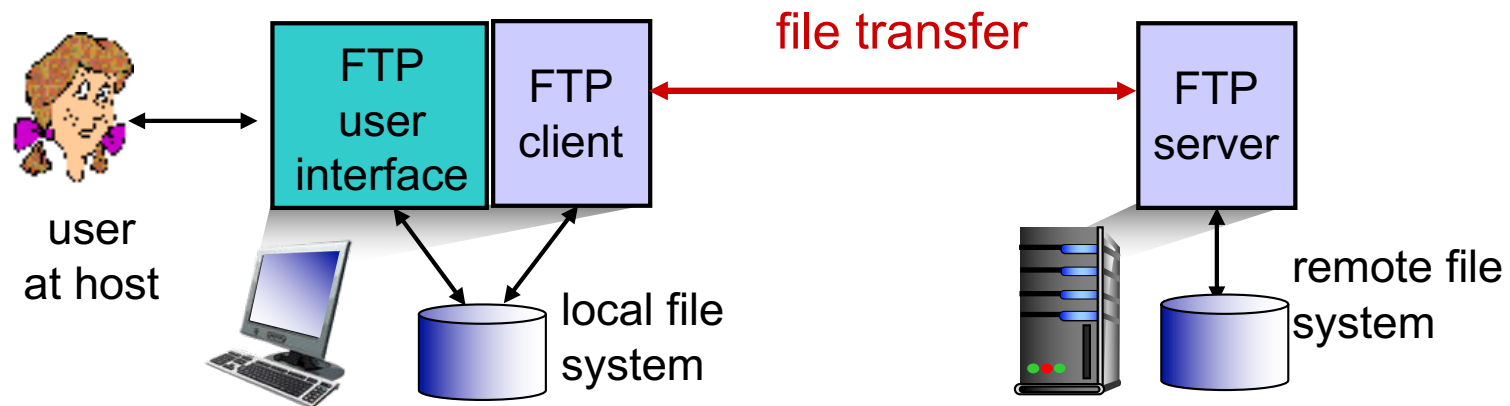
---

```
GET /user
Accept: application/json+userdb
```

```
200 OK
Content-Type: application/json+userdb

{
  "users": [
    {
      "id": 1,
      "name": "Emil",
      "country": "Sweden",
      "links": [
        {
          "href": "/user/1",
          "rel": "self",
          "method": "GET"
        },
        {
          "href": "/user/1",
          "rel": "edit",
          "method": "PUT"
        },
        {
          "href": "/user/1",
          "rel": "delete",
          "method": "DELETE"
        }
      ]
    },
    {
      "id": 2,
      "name": "Ola",
      "country": "Sweden",
      "links": [
        {
          "href": "/user/2",
          "rel": "self",
          "method": "GET"
        },
        {
          "href": "/user/2",
          "rel": "edit",
          "method": "PUT"
        },
        {
          "href": "/user/2",
          "rel": "delete",
          "method": "DELETE"
        }
      ]
    }
  ]
}
```

# 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

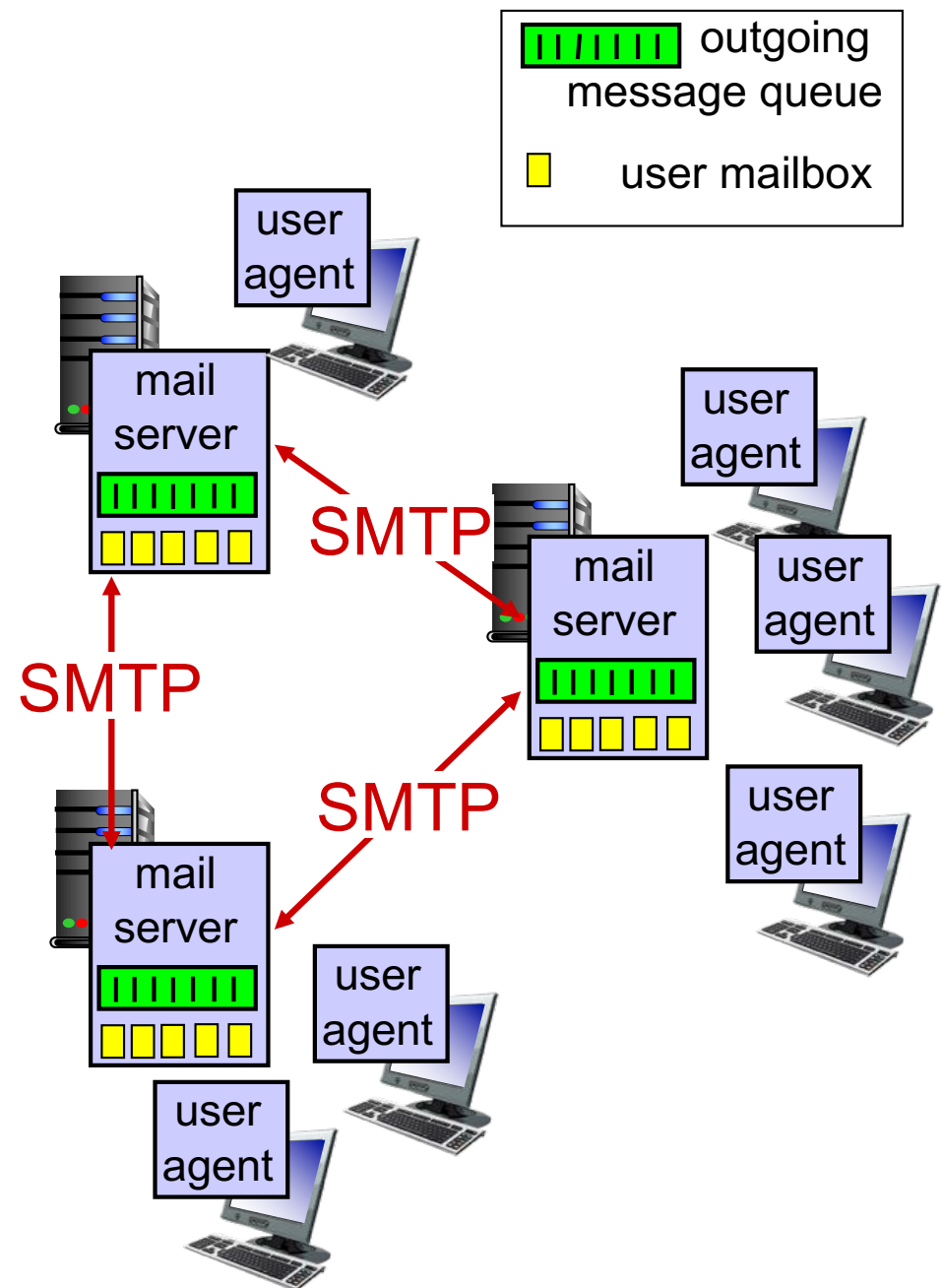
# Electronic mail

## *Three major components:*

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

## *User Agent*

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



# Electronic Mail: SMTP [RFC 2821]

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

# 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:

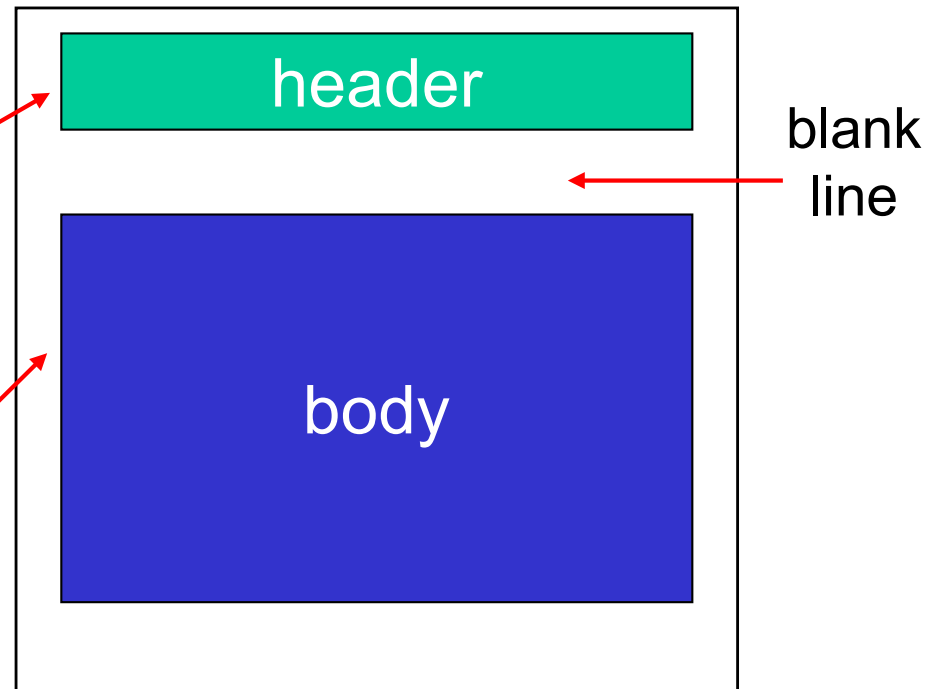
❖ header lines, e.g.,

- To:
- From:
- Subject:

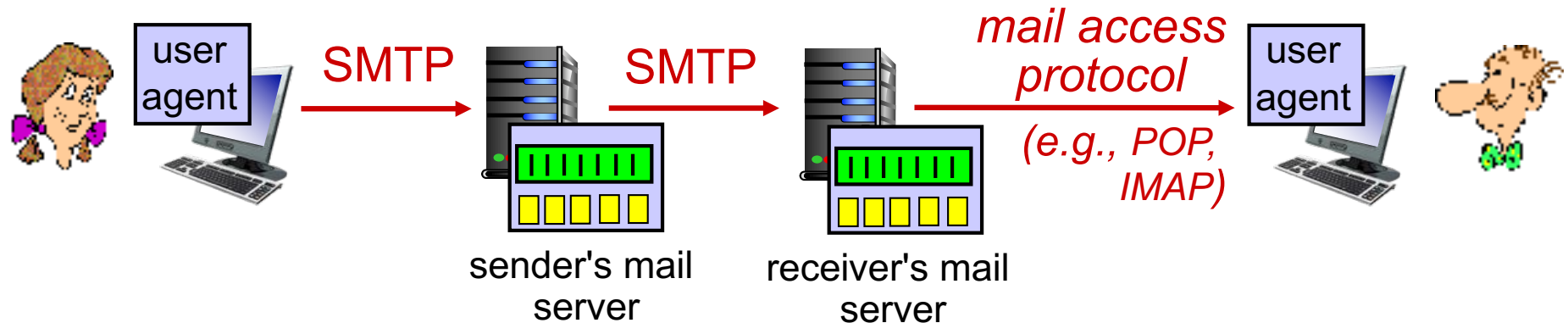
*different* from SMTP MAIL  
FROM, RCPT TO:  
commands!

❖ Body: the “message”

- ASCII characters only

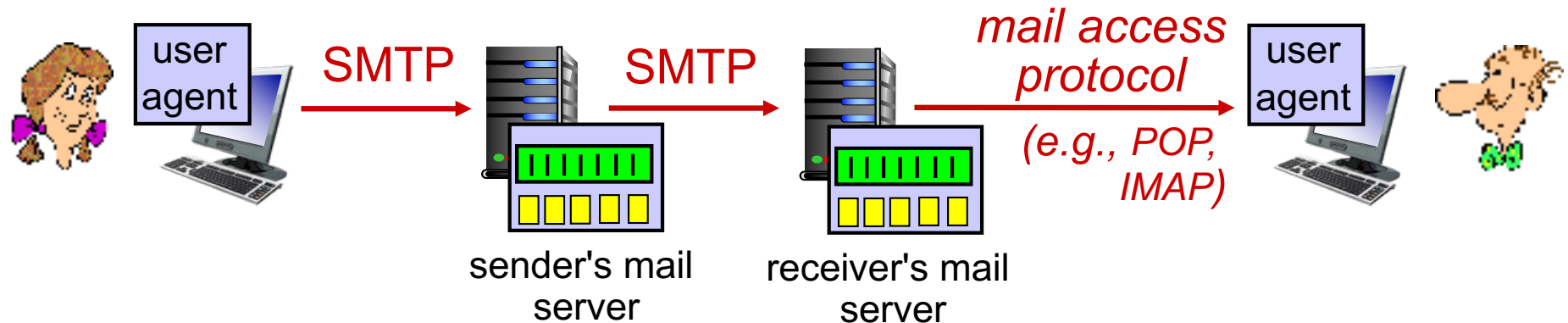


# Mail access protocols



- ❖ **SMTP:** delivery/storage to receiver's server
- ❖ mail access protocol: retrieval from server
  - **POP:** Post Office Protocol [RFC 1939]: authorization, download
  - **IMAP:** Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  - **HTTP:** gmail, Hotmail, Yahoo! Mail, etc.

# Mail as an overlay



- ❖ SMTP allows mail relaying
- ❖ You can even send through multiple SMTP servers
  - What's wrong with “open relays”?



# Chapter 2: outline



# Chapter 2: outline

## 2.1 principles of network applications

- app architectures
- app requirements

## 2.2 Web and HTTP

## 2.3 FTP

## 2.4 electronic mail

- SMTP, POP3, IMAP

## 2.5 DNS

## 2.6 P2P applications

## 2.7 socket programming with UDP and TCP

# DNS: domain name system

*people:* many identifiers:

- SSN, name, passport #

*Internet hosts, routers:*

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

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

## *Domain Name System:*

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

# DNS: services, structure

## *DNS services*

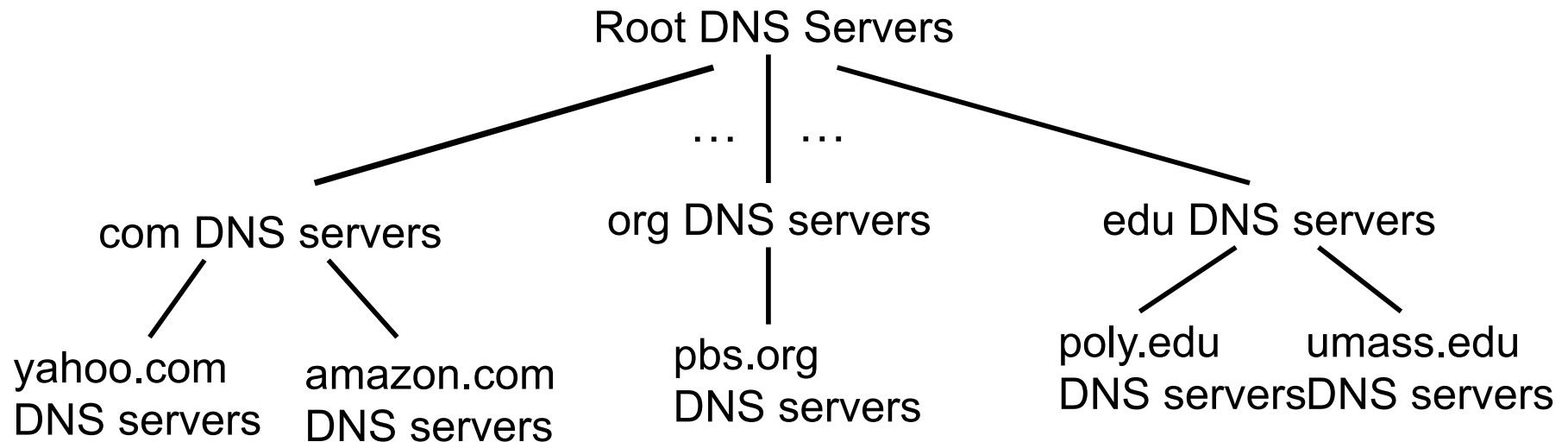
- ❖ hostname to IP address translation
- ❖ host aliasing
  - canonical, alias names
- ❖ mail server aliasing
- ❖ load distribution
  - replicated Web servers: many IP addresses correspond to one name

## *why not centralize DNS?*

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

*A: doesn't scale!*

# DNS: a distributed, hierarchical database

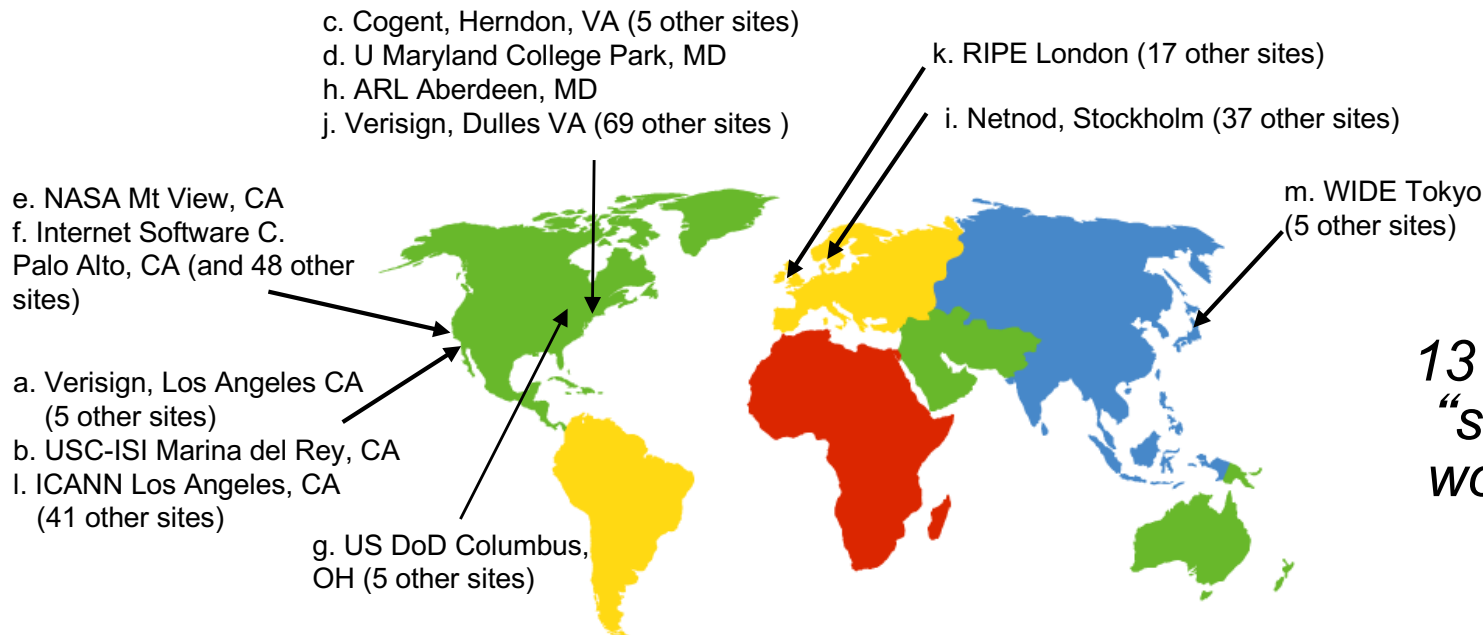


*client wants IP for www.amazon.com; 1<sup>st</sup> approx:*

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

# DNS: root name servers

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



*13 root name  
“servers”  
worldwide*

# TLD, authoritative servers

## *top-level domain (TLD) servers:*

- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TLD

## *authoritative DNS servers:*

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

# Local DNS name server

- ❖ does not strictly belong to hierarchy
- ❖ each ISP (residential ISP, company, university) has one
  - also called “default name server”
- ❖ when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

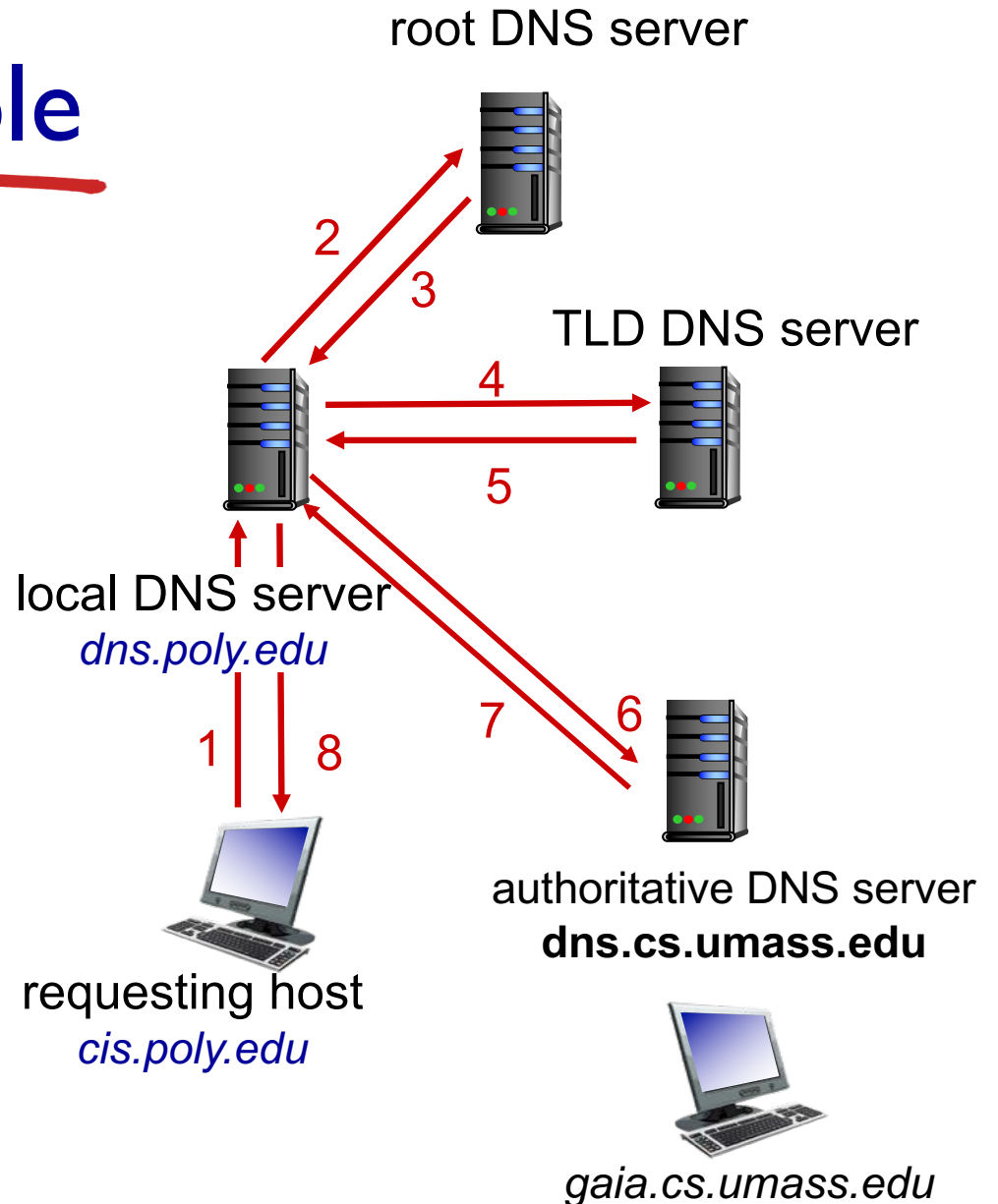


# DNS name resolution example

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

## *iterated query:*

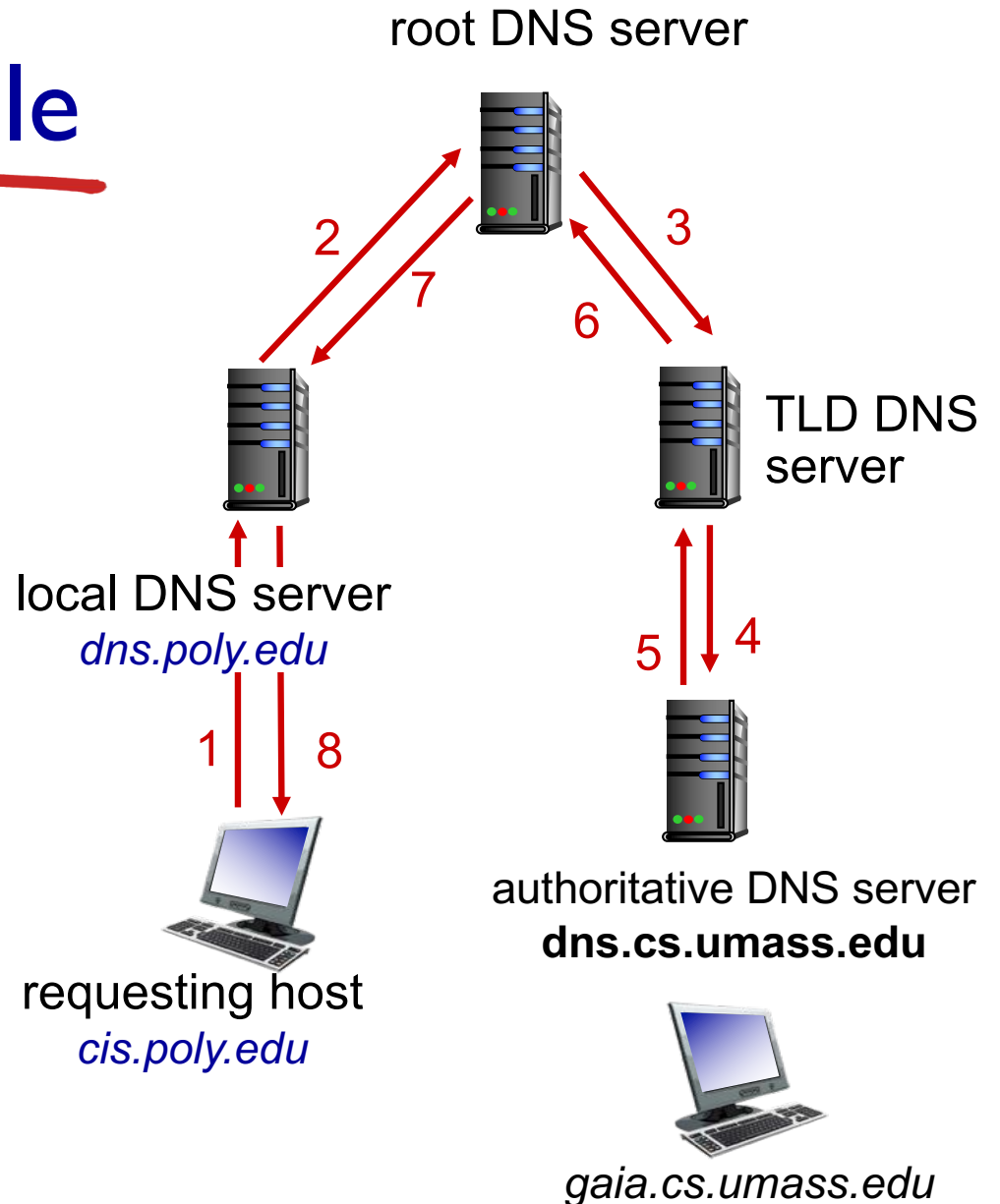
- ❖ contacted server replies with name of server to contact
- ❖ “I don't know this name, but ask this server”



# DNS name resolution example

## *recursive query:*

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



# DNS: caching, updating records

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