

CSCI-1680

HTTP & Web

Chen Avin



Administrivia

- **This Week:**
 - HW3 (delay to Thursday)
 - TCP Milestone II
- **Coming week: HTTP, E-mail, DNS**



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

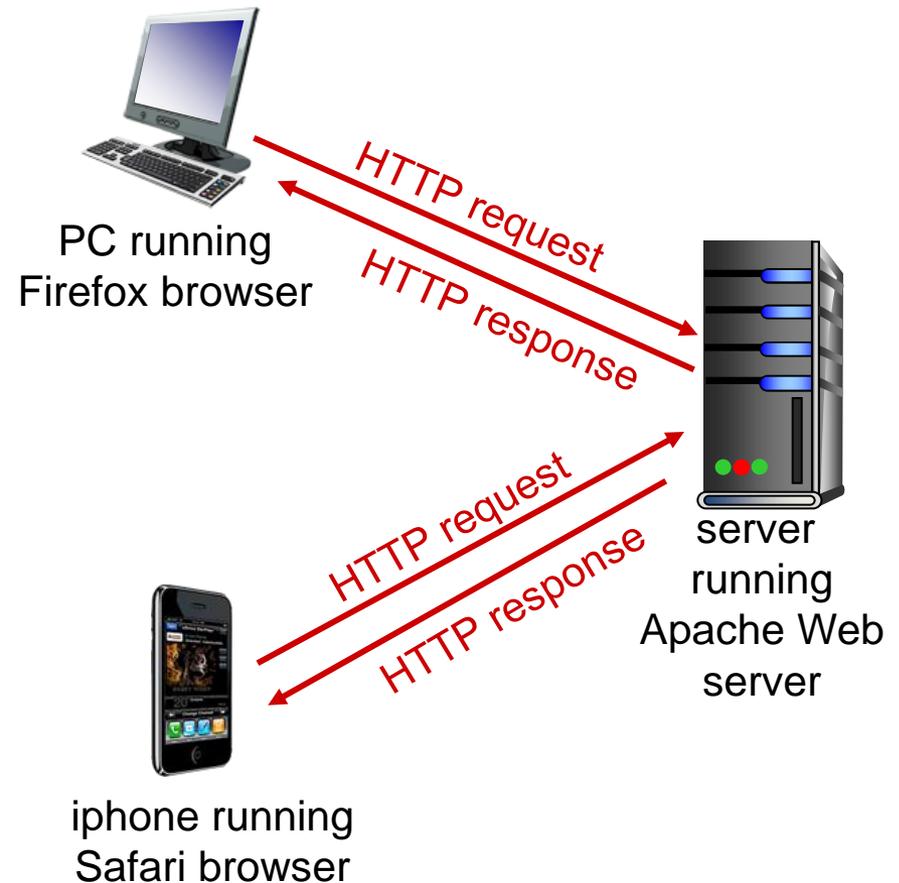
path name



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)

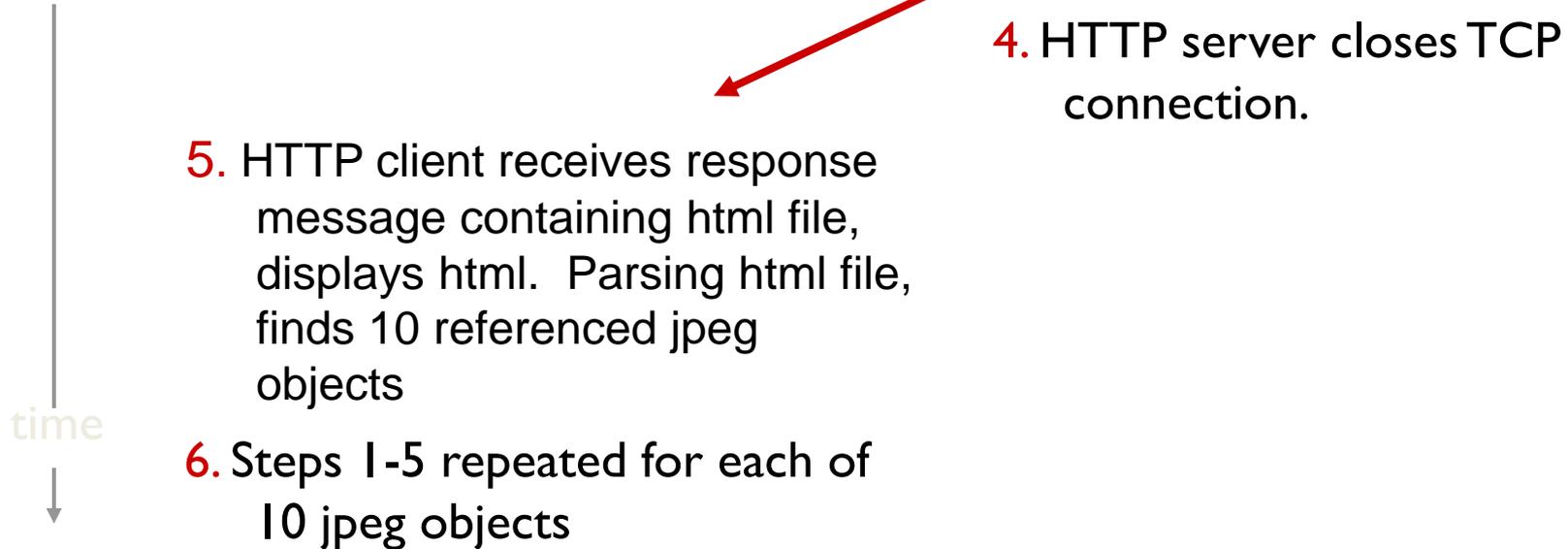
1a. HTTP client initiates TCP connection to HTTP server (process) at `www.someSchool.edu` on port 80

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

Non-persistent HTTP (cont.)

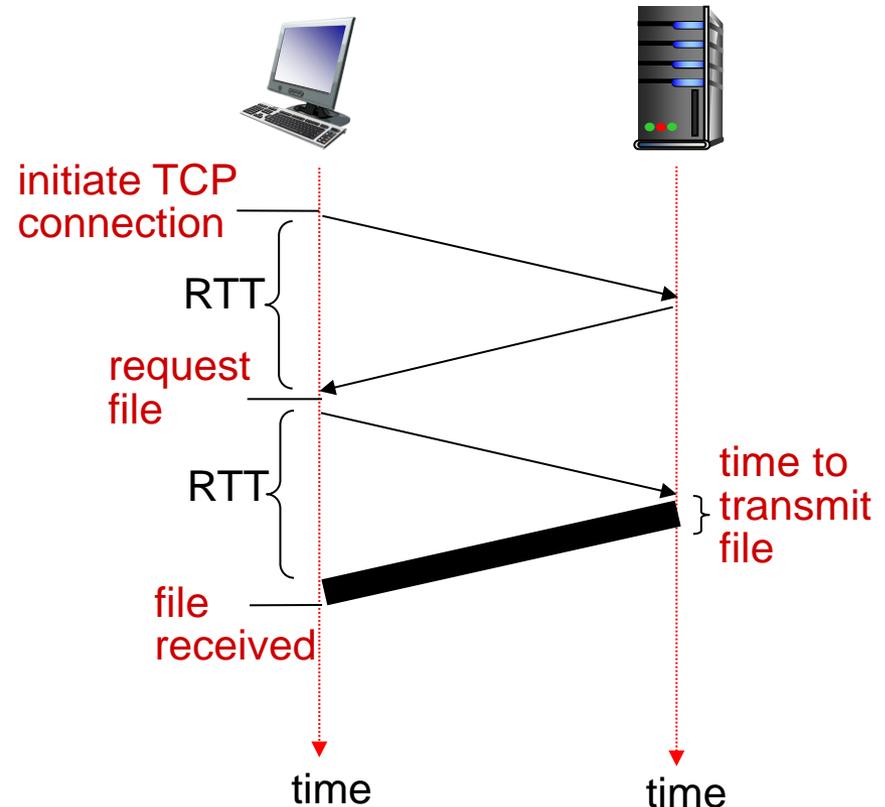


Non-persistent HTTP: response time

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 =
 $2RTT + \text{file transmission time}$

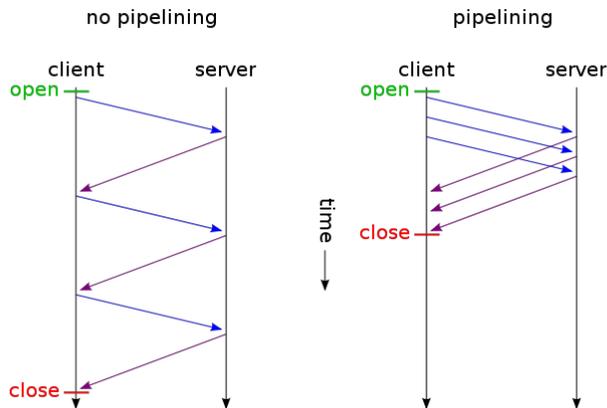


Persistent HTTP

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
- With/without pipelining



HTTP request message

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

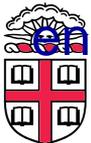
request line
(GET, POST,
HEAD commands)

header
lines

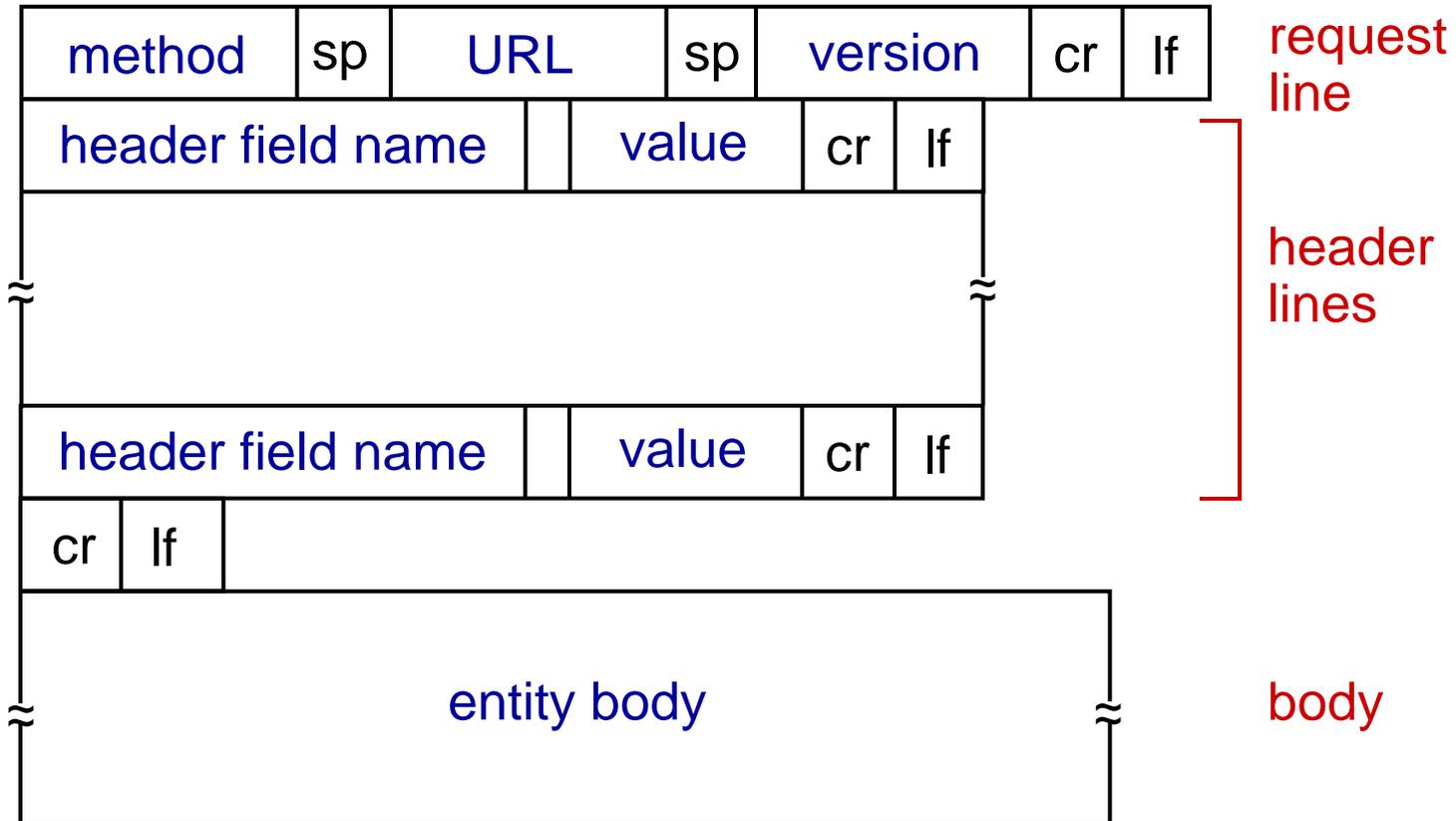
carriage return,
line feed at start
of line indicates
end of header lines

```
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)

```
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
```

header
lines

```
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, e.g.,
requested
HTML file

```
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

usual http response
set-cookie: 1678

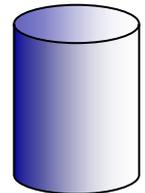
usual http request msg
cookie: 1678

usual http response msg

Amazon server
creates ID
1678 for user

create
entry

backend
database



access

cookie-
specific
action

access

cookie-
specific
action

one week later:



usual http request msg
cookie: 1678

usual http response msg



Cookies (continued)

what cookies can be used for:

- 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:

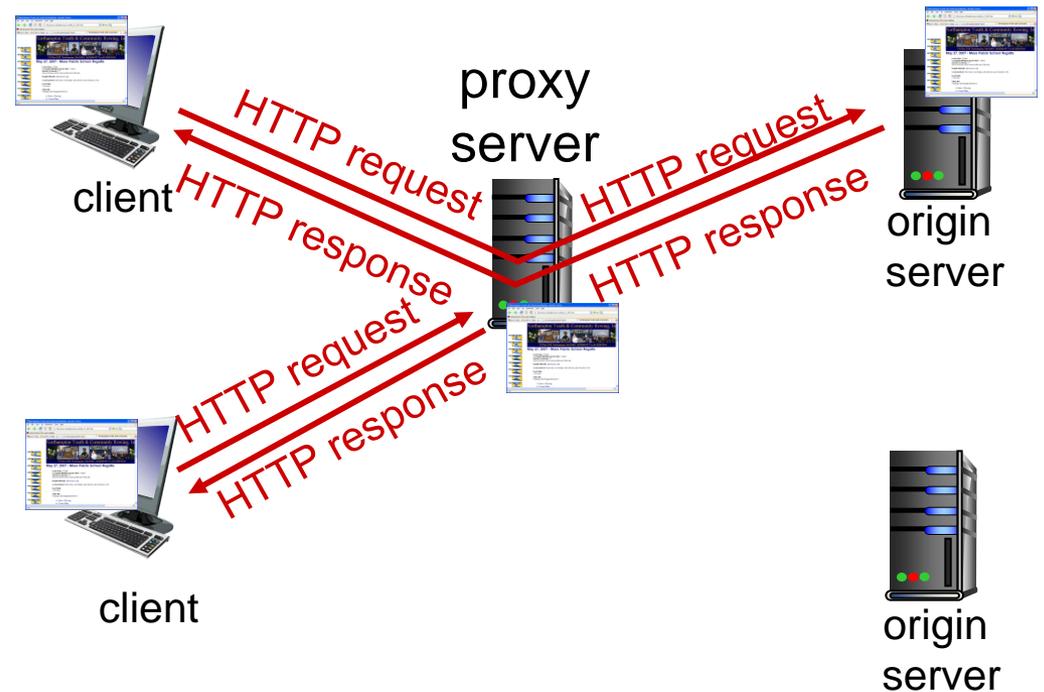
- ❖ cookies permit sites to learn a lot about you
- ❖ you may supply name and e-mail to sites



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**
 - 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:

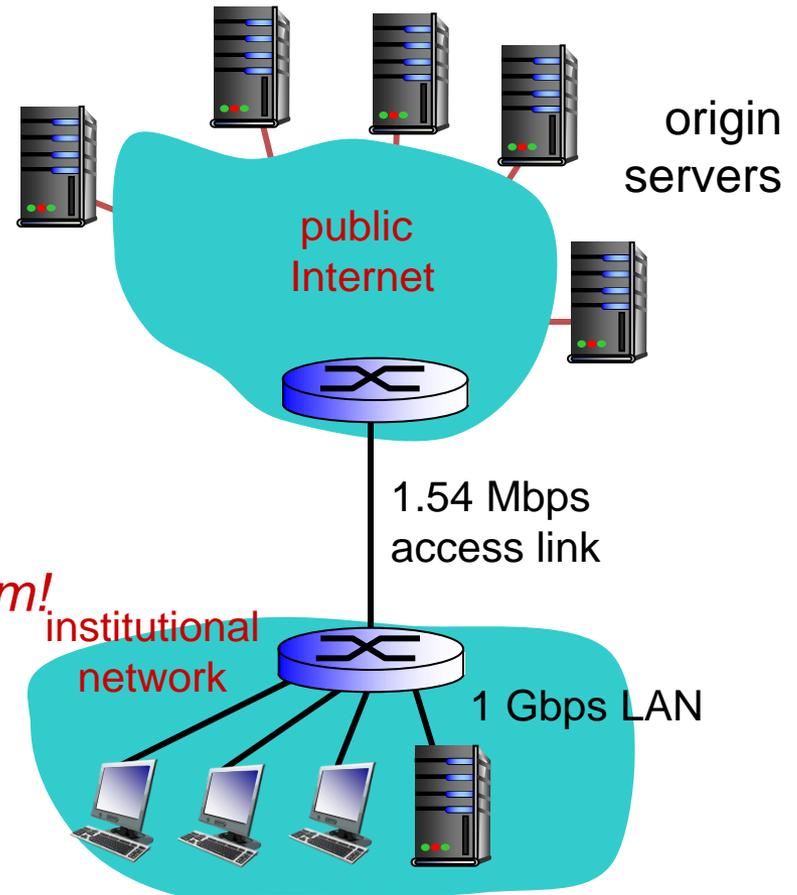
assumptions:

- ❖ 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%
- ❖ total delay = Internet delay + access delay + LAN delay
= 2 sec + minutes + usecs

problem!



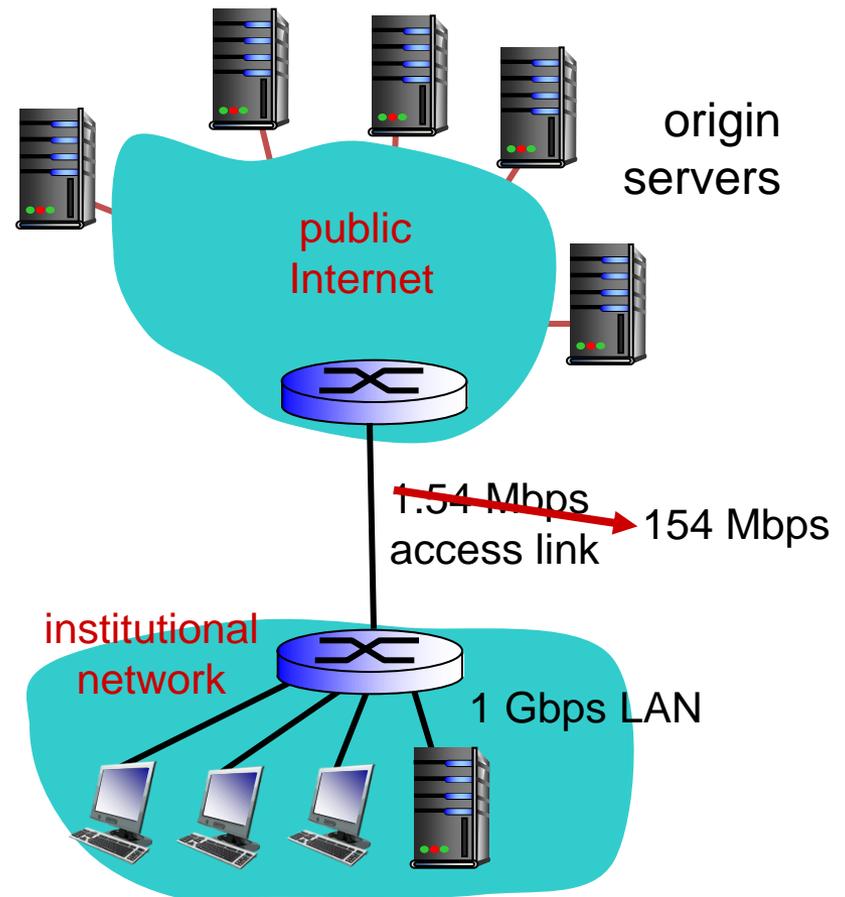
Caching example: fatter access link

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 + usecs
→ msec



Cost: increased access link speed (not cheap!)



Caching example: install local cache

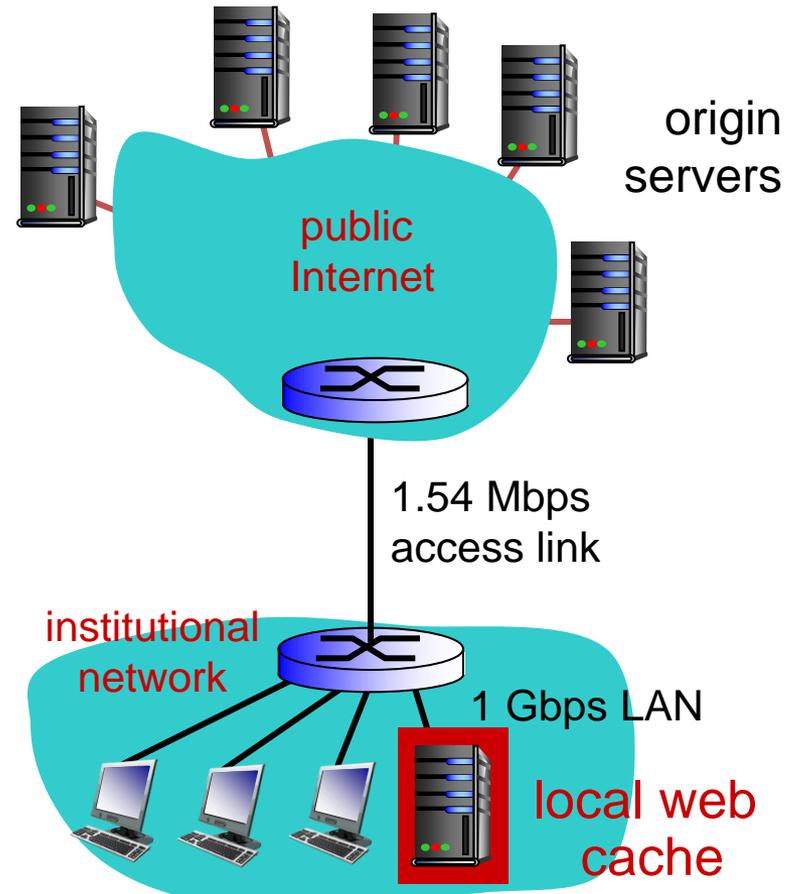
assumptions:

- ❖ 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?



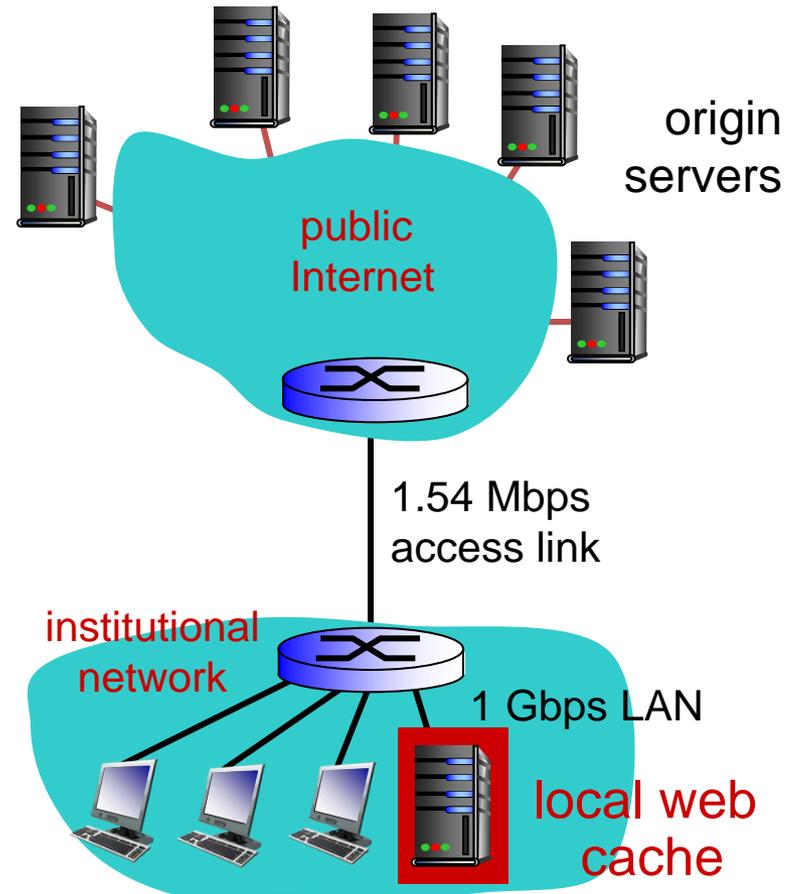
Cost: web cache (cheap!)



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

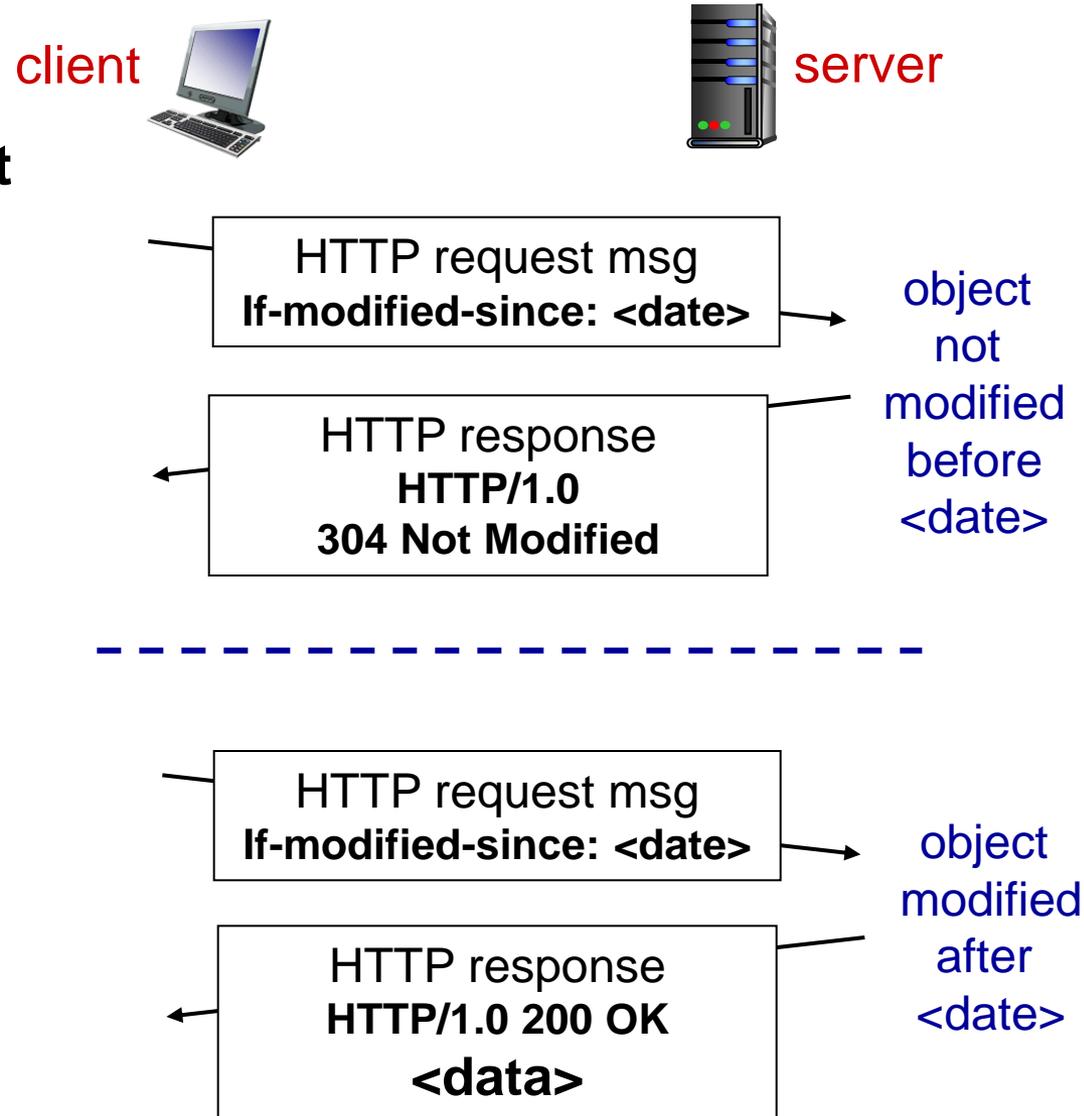
- no object transmission delay
- lower link utilization

- **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



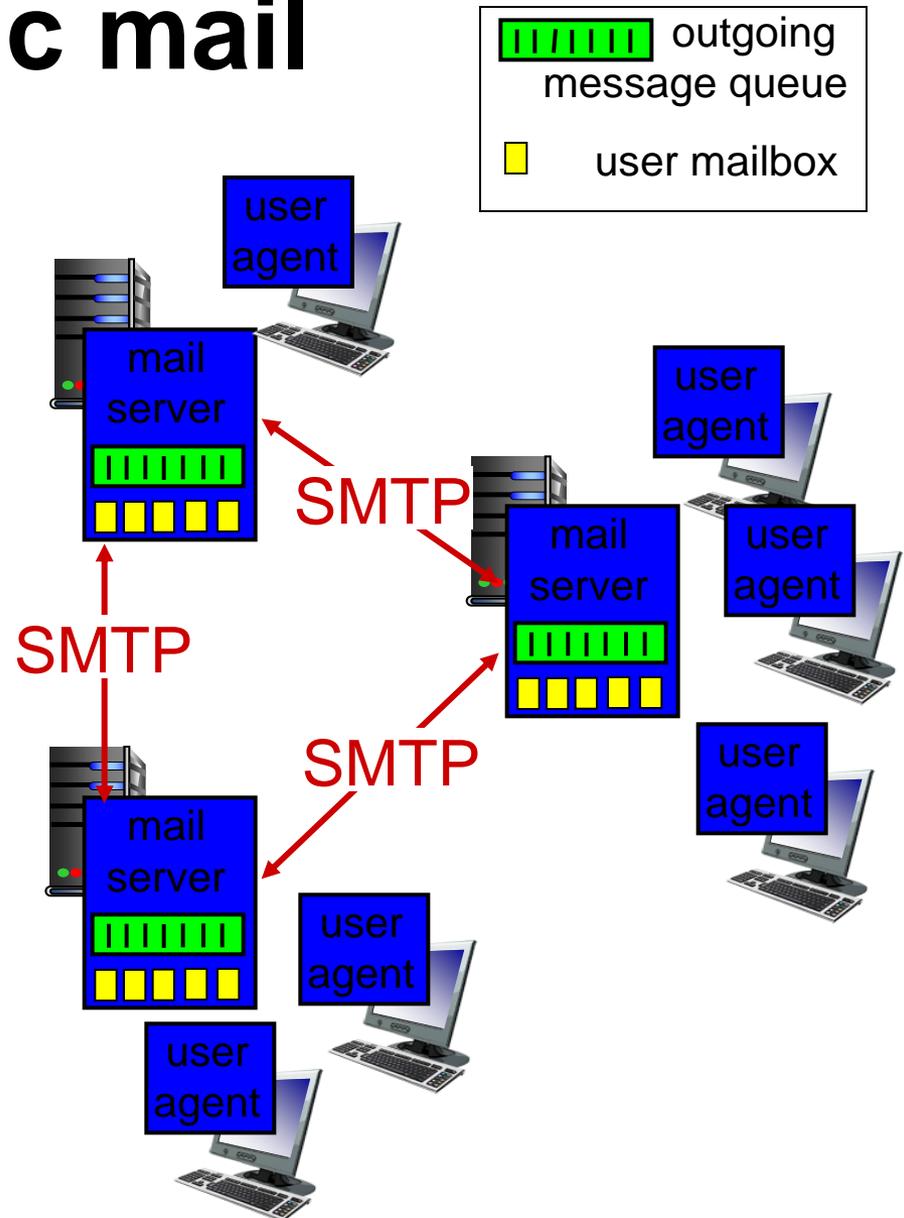
Electronic mail

Three major components:

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

User Agent

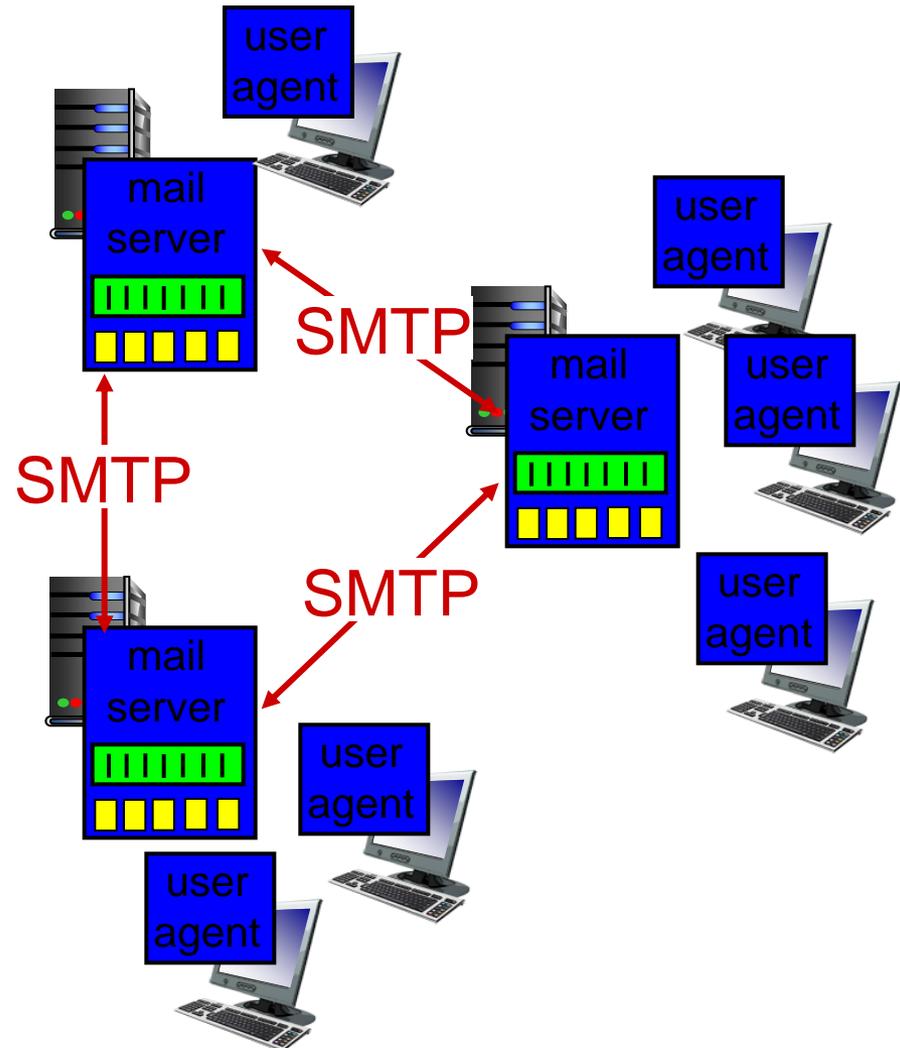
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



Electronic mail: mail servers

mail servers:

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



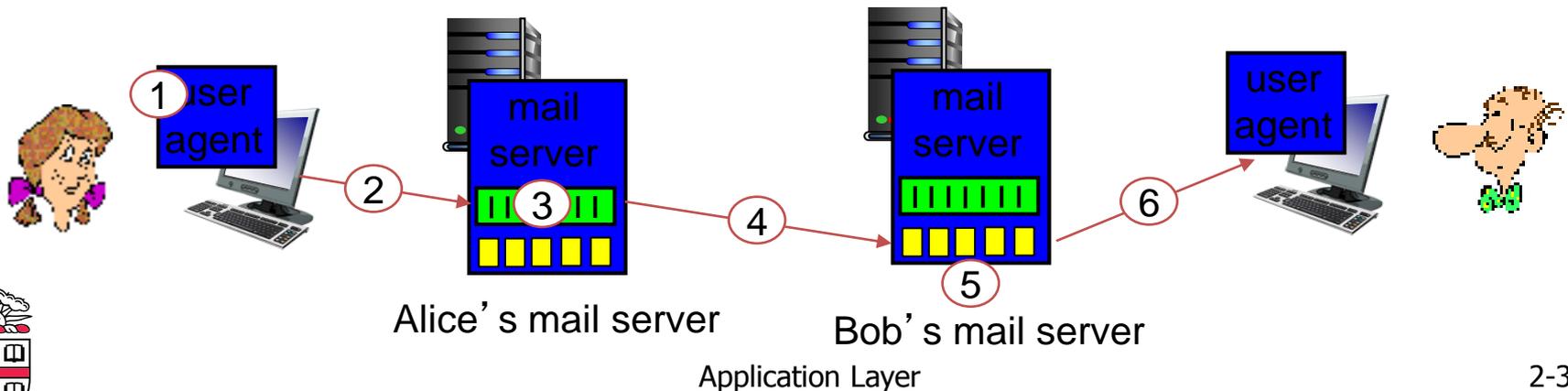
Electronic Mail: SMTP [RFC 2821]

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



Scenario: Alice sends message to Bob

- 1) Alice uses UA to compose message "to" bob@somechool.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**



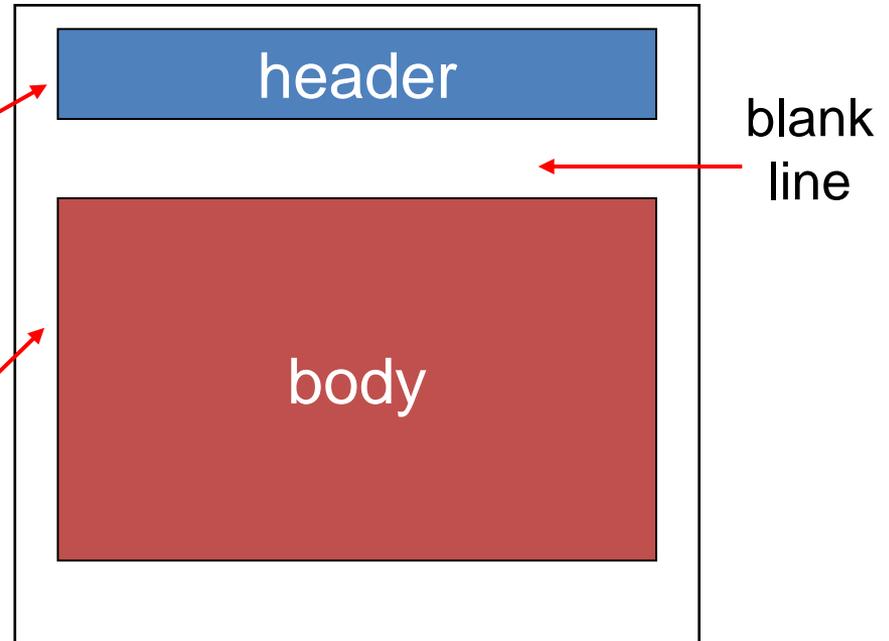
Mail message format

SMTP: protocol for exchanging email msgs

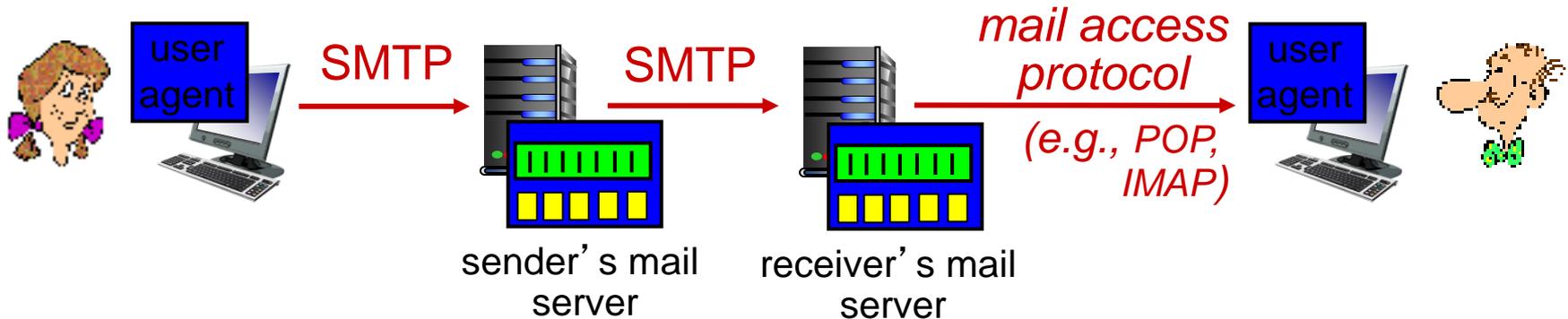
RFC 822: standard for text message format:

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

different from SMTP 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.



POP3 protocol

authorization phase

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

transaction phase, **client:**

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

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

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



POP3 (more) and IMAP

more about POP3

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

IMAP

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

