



# Applied Sciences Faculty

## Management Information Systems

**Dr. Yakup BAKIŞ**



# Application Layer

# Application layer: overview

- Principles of network applications
- **Web and HTTP**
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



# Web and HTTP

*First, a quick review...*

- web page consists of *objects*, each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL*, e.g.,  
`www.someschool.edu/someDept/pic.gif`

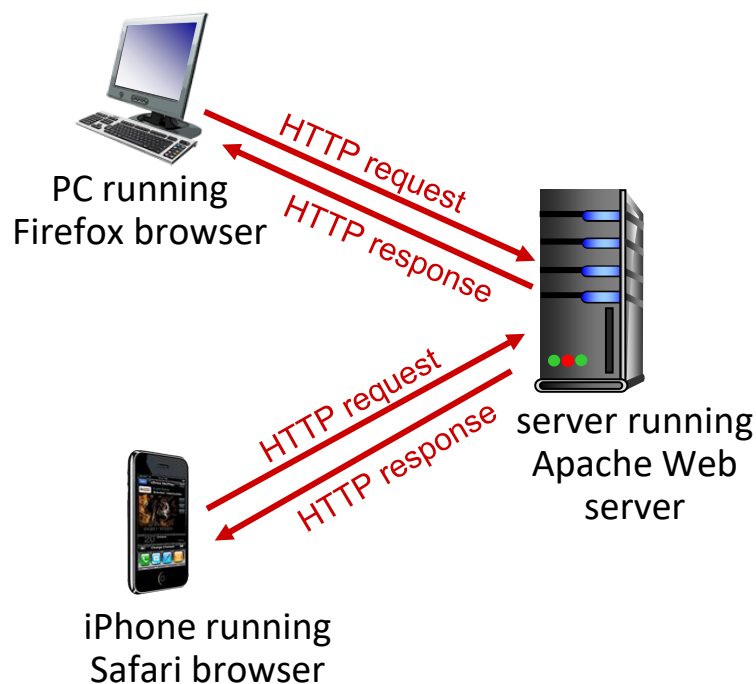
host name

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)

### *HTTP 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: two types

### *Non-persistent HTTP*

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

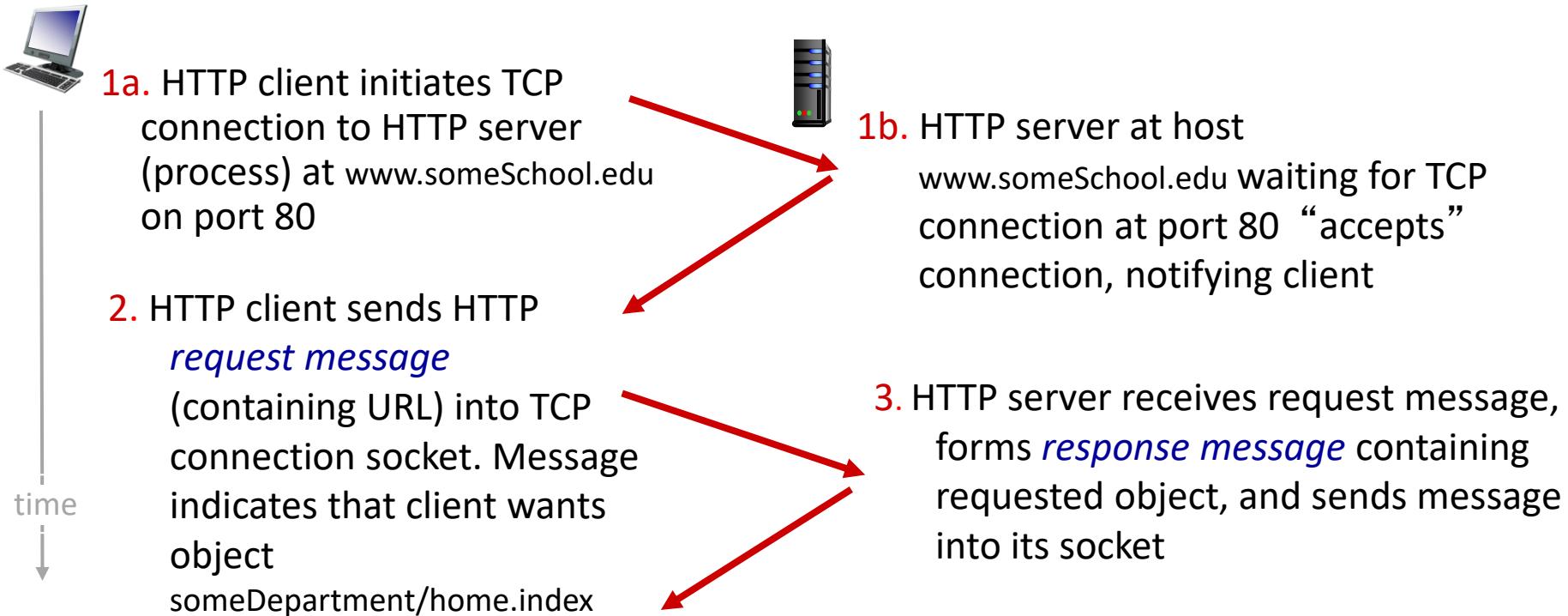
downloading multiple objects required multiple connections

### *Persistent HTTP*

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

# Non-persistent HTTP: example

User enters URL: `www.someSchool.edu/someDepartment/home.index`  
(containing text, references to 10 jpeg images)



# Non-persistent HTTP: example (cont.)

User enters URL: `www.someSchool.edu/someDepartment/home.index`  
(containing text, references to 10 jpeg images)



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

4. HTTP server closes TCP connection.

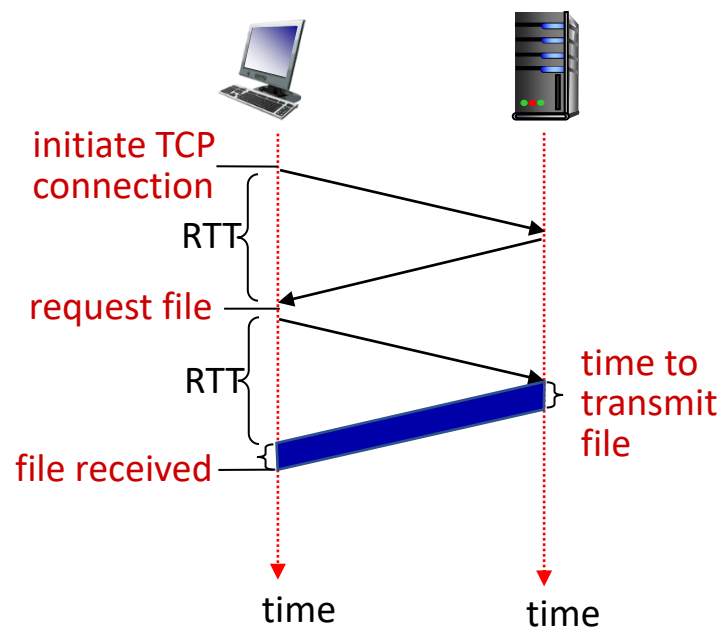


# Non-persistent HTTP: response time

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

**HTTP response time (per object):**

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



*Non-persistent HTTP response time = 2RTT + file transmission time*

## Persistent HTTP (HTTP 1.1)

### *Non-persistent HTTP issues:*

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

### *Persistent HTTP (HTTP1.1):*

- 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 (cutting response time in half)

# 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

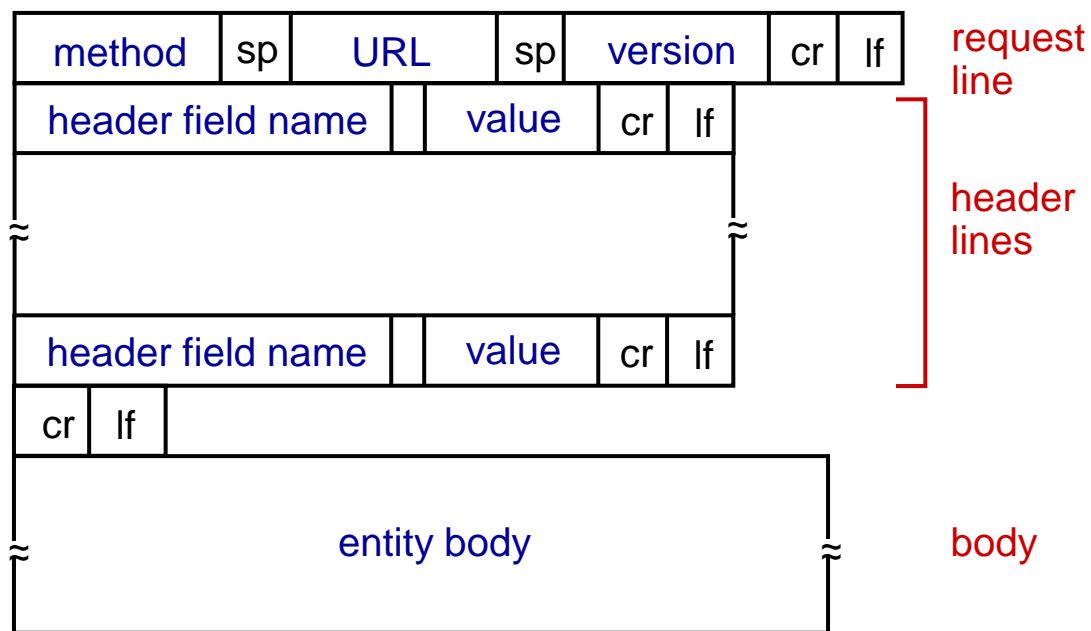
carriage return character  
line-feed character

```
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, line  
feed at start of line  
indicates end of header  
lines

\* Check out the online interactive exercises for more  
examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# HTTP request message: general format



# Other HTTP request messages

## POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):  
`www.someSite.com/animalsearch?monkeys&banana`

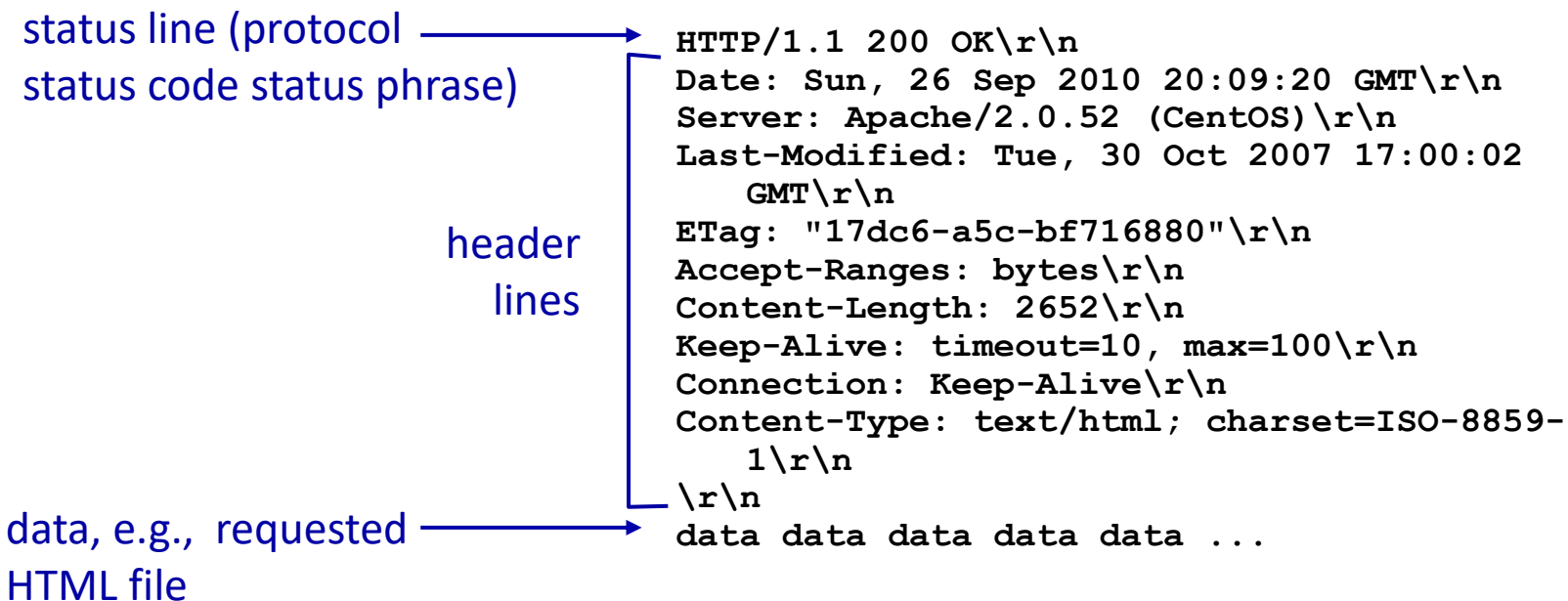
## HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

## PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

# HTTP response message



\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# 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 message

## 301 Moved Permanently

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

## 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 gaia.cs.umass.edu 80
```

- opens TCP connection to port 80 (default HTTP server port) at gaia.cs.umass.edu.
- anything typed in will be sent to port 80 at gaia.cs.umass.edu

## 2. type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1  
Host: gaia.cs.umass.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!

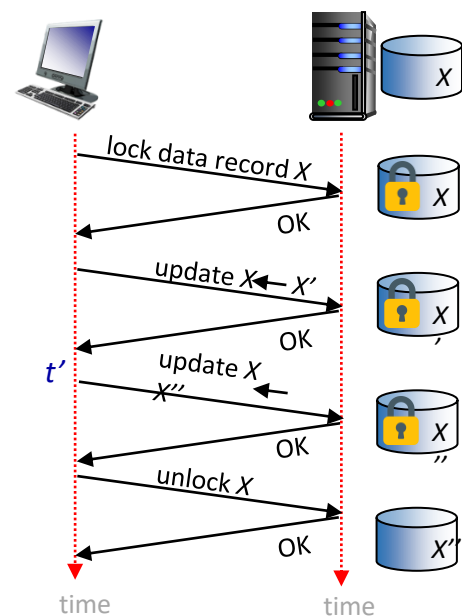
(or use Wireshark to look at captured HTTP request/response)

# Maintaining user/server state: cookies

Recall: HTTP GET/response interaction is *stateless*

- no notion of multi-step exchanges of HTTP messages to complete a Web “transaction”
  - no need for client/server to track “state” of multi-step exchange
  - all HTTP requests are independent of each other
  - no need for client/server to “recover” from a partially-completed-but-never-completely-completed transaction

a **stateful protocol**: client makes two changes to X, or none at all



**Q:** what happens if network connection or client crashes at  $t'$  ?

# Maintaining user/server state: cookies

Web sites and client browser use *cookies* to maintain some state between transactions

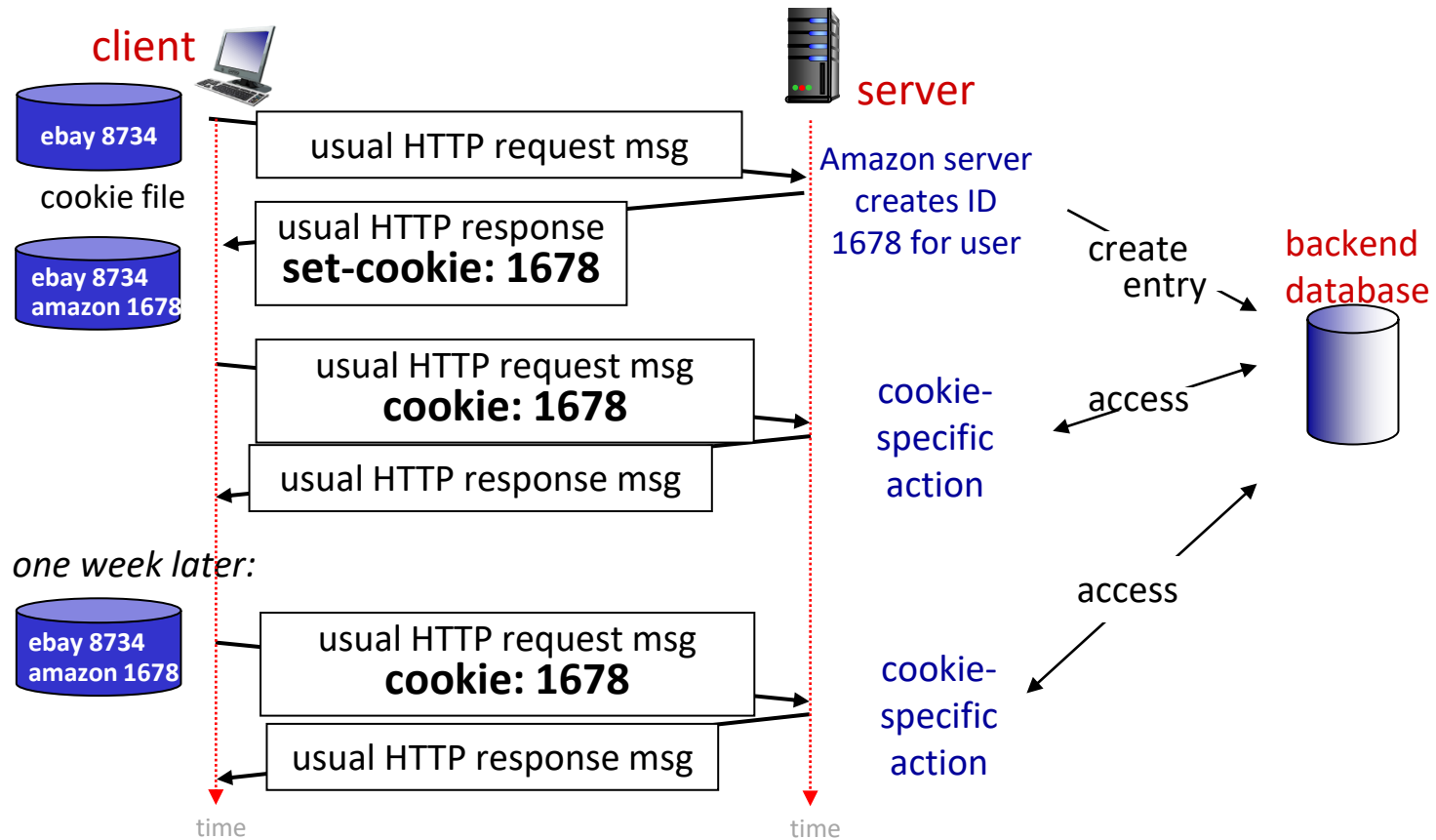
## *four components:*

- 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 uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka "cookie")
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

# Maintaining user/server state: cookies



# HTTP cookies: comments

## *What cookies can be used for:*

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

## *Challenge: How to keep state:*

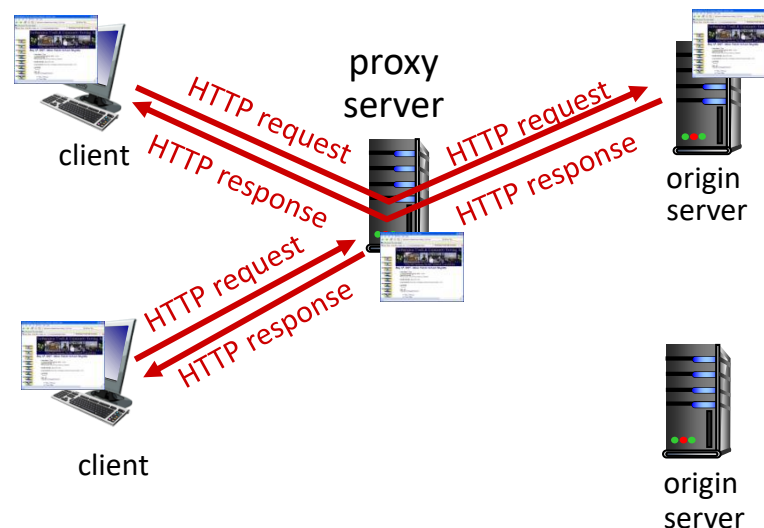
- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: HTTP messages carry state

- aside
- ### *cookies and privacy:*
- cookies permit sites to *learn* a lot about you on their site.
  - third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

# Web caches (proxy servers)

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

- user configures browser to point to a **Web cache**
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, then returns object to client



# Web caches (proxy servers)

- Web 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
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches
  - enables "poor" content providers to more effectively deliver content

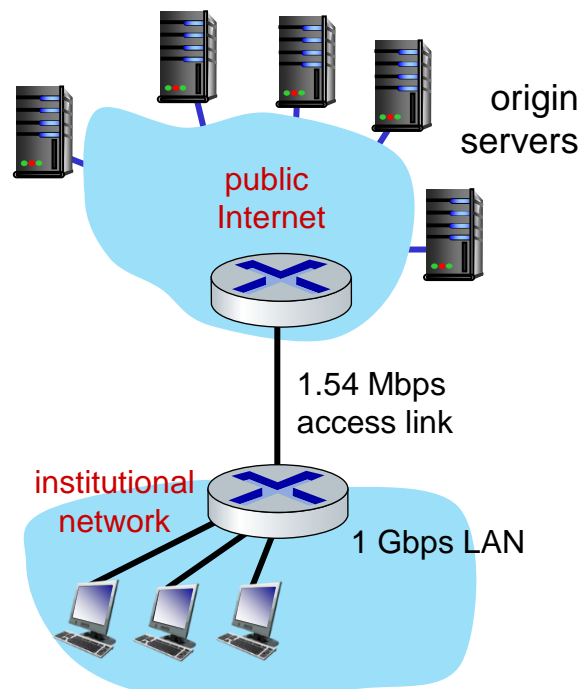
# Caching example

## Scenario:

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

## Performance:

- LAN utilization: .0015
  - access link utilization = .97 *problem: large delays at high utilization!*
  - end-end delay = Internet delay + access link delay + LAN delay
- = 2 sec + minutes + usecs



# Caching example: buy a faster access link

## Scenario:

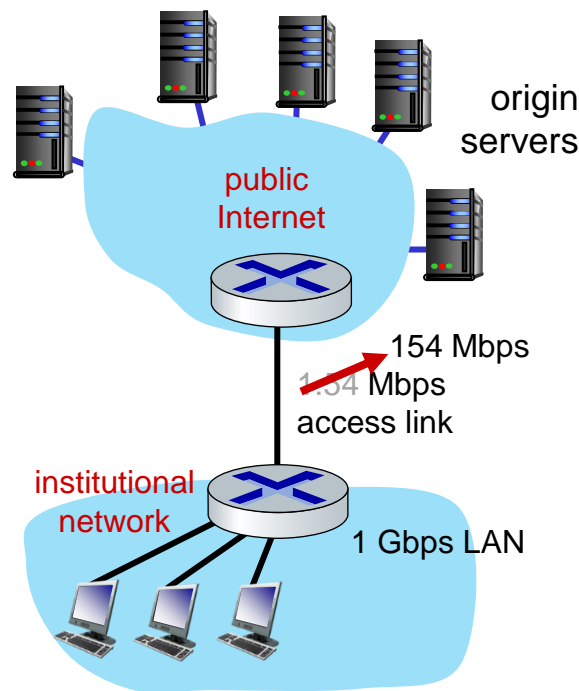
- access link rate: ~~1.54~~ <sup>154</sup> Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

## Performance:

- LAN utilization: .0015
- access link utilization = ~~.97~~ <sup>.0097</sup>
- end-end delay = Internet delay + access link delay + LAN

delay

**Cost:** faster access link (2 \$expensive) + use ~~2 sec~~ <sup>2 msec</sup>



# Caching example: install a web cache

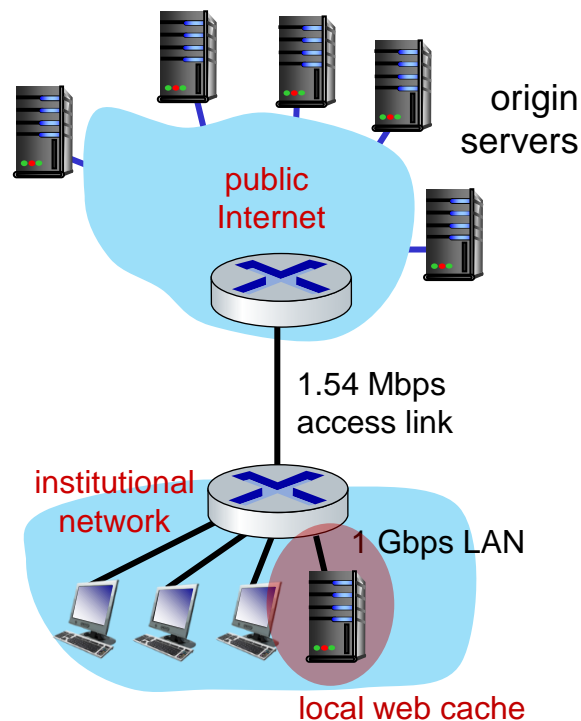
## Scenario:

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

## Performance:

- LAN utilization: .?
  - access link utilization = ?
  - average end-end delay = ?
- How to compute link utilization, delay?*

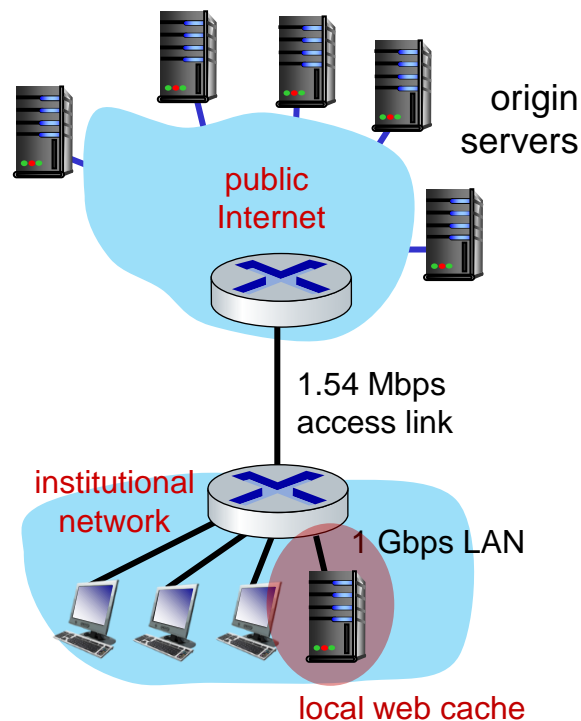
**Cost:** web cache (cheap!)



# Caching example: install a web cache

## Calculating access link utilization, end-end delay with cache:

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



*lower average end-end delay than with 154 Mbps link (and cheaper too!)*

# Conditional GET

**Goal:** don't send object if cache has up-to-date cached version

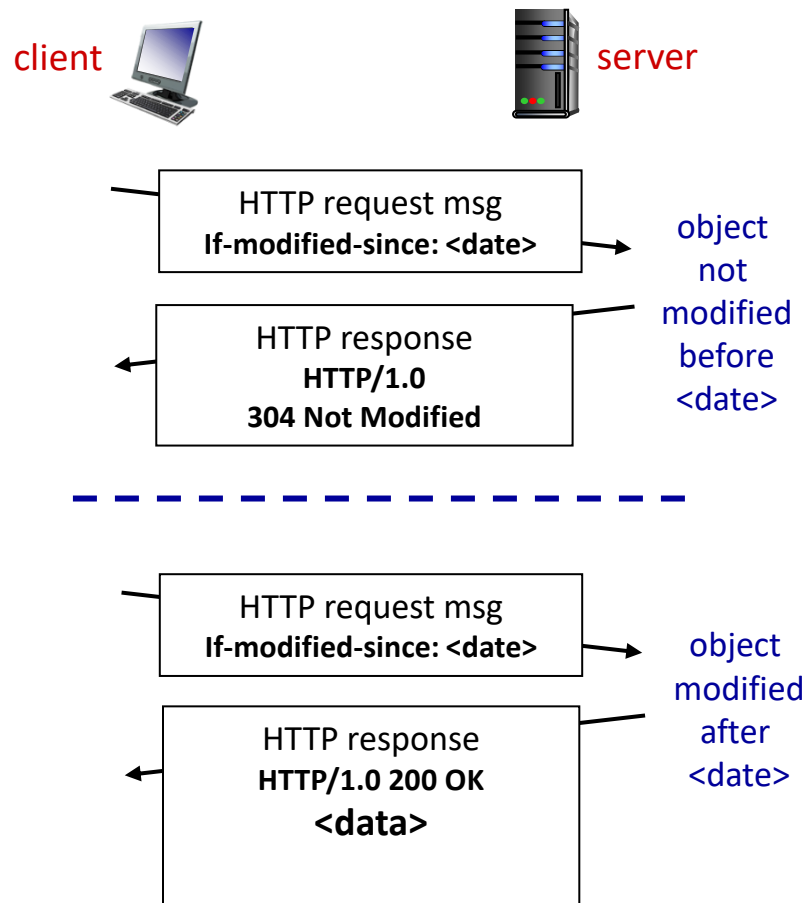
- no object transmission delay
- lower link utilization

- **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**



# HTTP/2

*Key goal:* decreased delay in multi-object HTTP requests

HTTP1.1: introduced **multiple, pipelined GETs** over single TCP connection

- server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- with FCFS, small object may have to wait for transmission (**head-of-line (HOL) blocking**) behind large object(s)
- loss recovery (retransmitting lost TCP segments) stalls object transmission

# HTTP/2

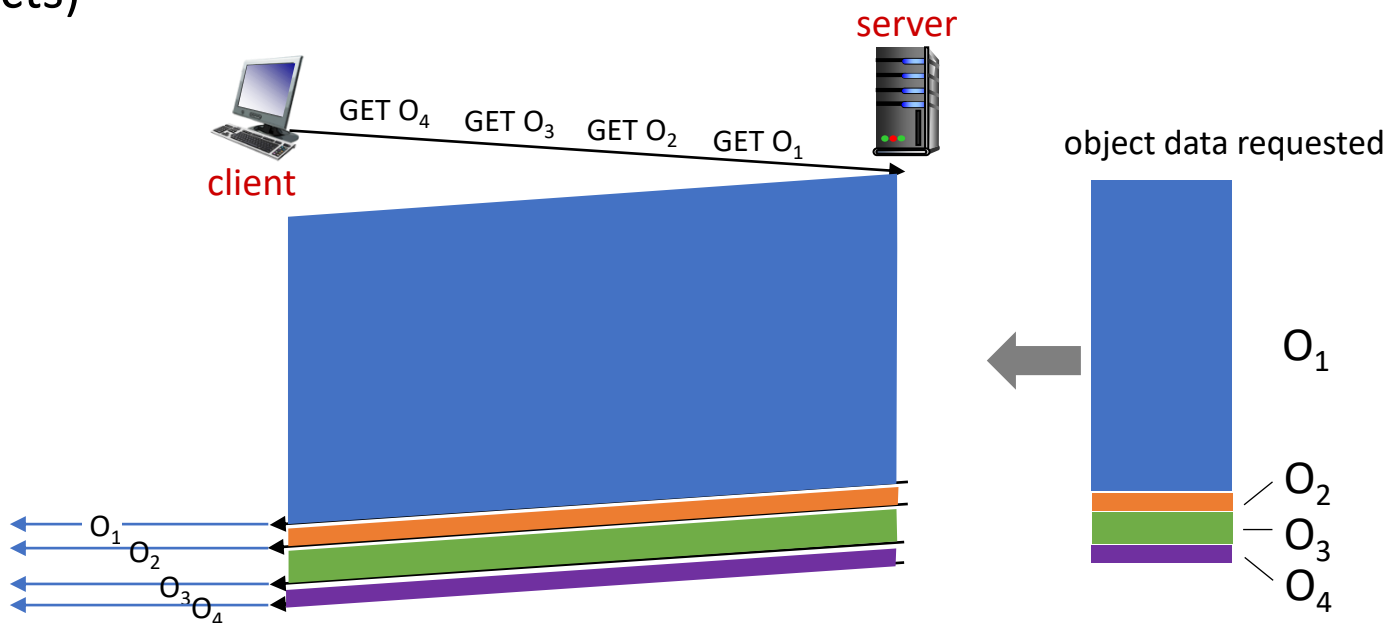
*Key goal:* decreased delay in multi-object HTTP requests

*HTTP/2:* [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:

- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking

# HTTP/2: mitigating HOL blocking

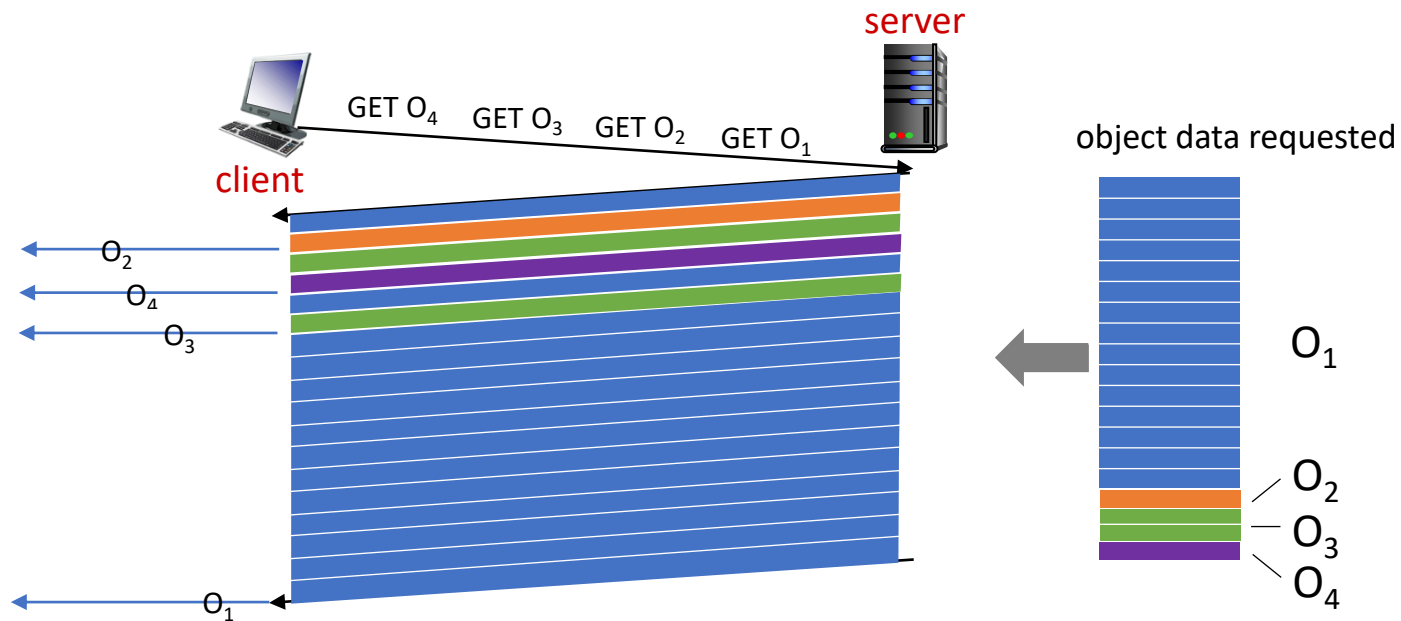
HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)



*objects delivered in order requested:  $O_2$ ,  $O_3$ ,  $O_4$  wait behind  $O_1$*

# HTTP/2: mitigating HOL blocking

HTTP/2: objects divided into frames, frame transmission interleaved



*O<sub>2</sub>, O<sub>3</sub>, O<sub>4</sub> delivered quickly, O<sub>1</sub> slightly delayed*

# HTTP/2 to HTTP/3

*Key goal:* decreased delay in multi-object HTTP requests

HTTP/2 over single TCP connection means:

- recovery from packet loss still stalls all object transmissions
  - as in HTTP 1.1, browsers have incentive to open multiple parallel TCP connections to reduce stalling, increase overall throughput
- no security over vanilla TCP connection
- **HTTP/3:** adds security , per object error- and congestion-control (more pipelining) over UDP
  - more on HTTP/3 in transport layer

# Application layer: overview

- Principles of network applications
- Web and HTTP
- **E-mail, SMTP, IMAP**
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



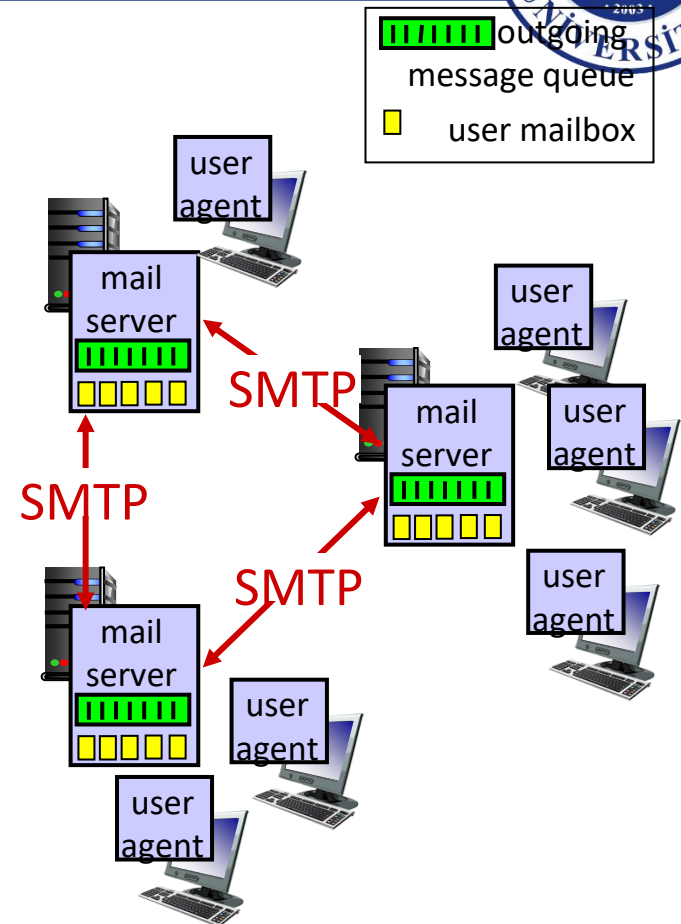
# E-mail

## Three major components:

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

## User Agent

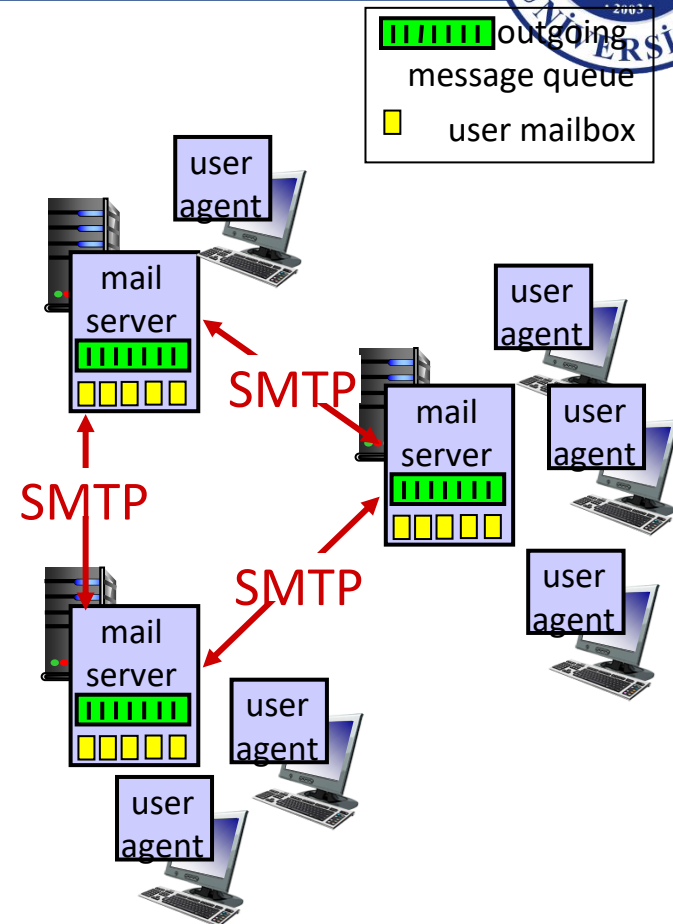
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server



# E-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

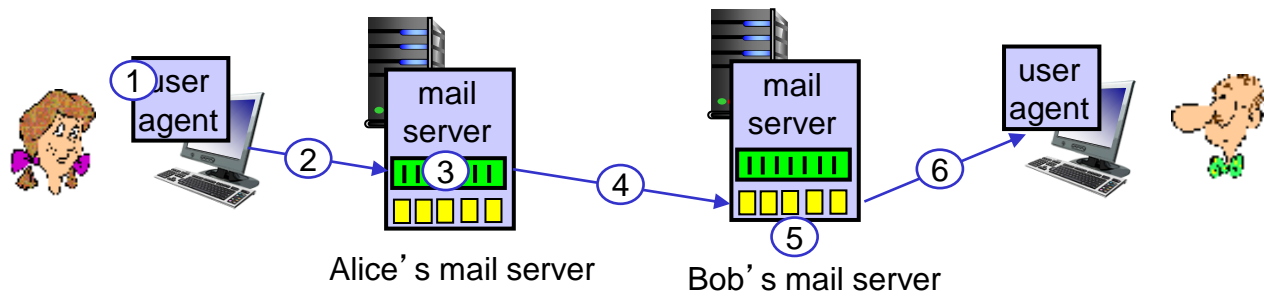


## E-mail: the RFC (5321)

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

# Scenario: Alice sends e-mail to Bob

- 1) Alice uses UA to compose e-mail message "to" bob@some school.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 e-mail client (reader)

*Note: this will only work if <servername> allows telnet connections to port 25 (this is becoming increasingly rare because of security concerns)*

# SMTP: closing observations

## *comparison with HTTP:*

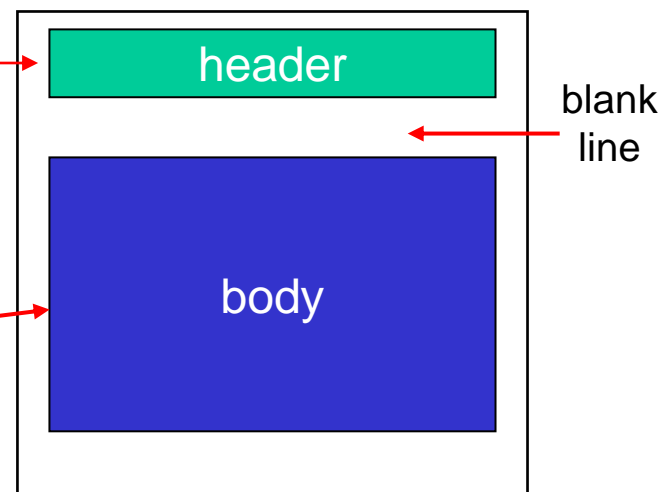
- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message
- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

# Mail message format

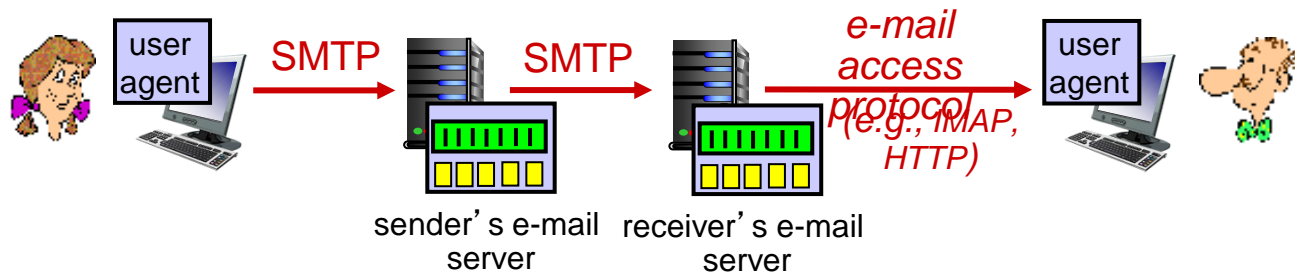
SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

RFC 822 defines *syntax* for e-mail message itself (like HTML)

- header lines, e.g.,
  - To:
  - From:
  - Subject:these lines, within the body of the email message area different from SMTP MAIL FROM; RCPT TO: commands!
- Body: the “message” , ASCII characters only



# Mail access protocols



- **SMTP**: delivery/storage of e-mail messages to receiver's server
- mail access protocol: retrieval from server
  - **IMAP**: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- **HTTP**: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of SMTP (to send), IMAP (or POP) to retrieve e-mail messages