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 applications

2.7 Socket programming with TCP

2.8 Socket programming with UDP

FAQ

- Can I do the homework by myself?
- Where/how do I submit?
  - In person (AFTER class), in dropbox, not both
  - One submission per person or per team?
- Can I schedule individual meeting because I missed your office hours?
- How do I access the website?
- Other questions: homework, book, office hours....

Web and HTTP

First, a review…

- web page (or "document") consists of objects
- object can be HTML file, JPEG image, Java applet, audio file,…
- Most web pages consist of base HTML-file which includes several referenced objects
- each object is addressable by a URL
- example URL:

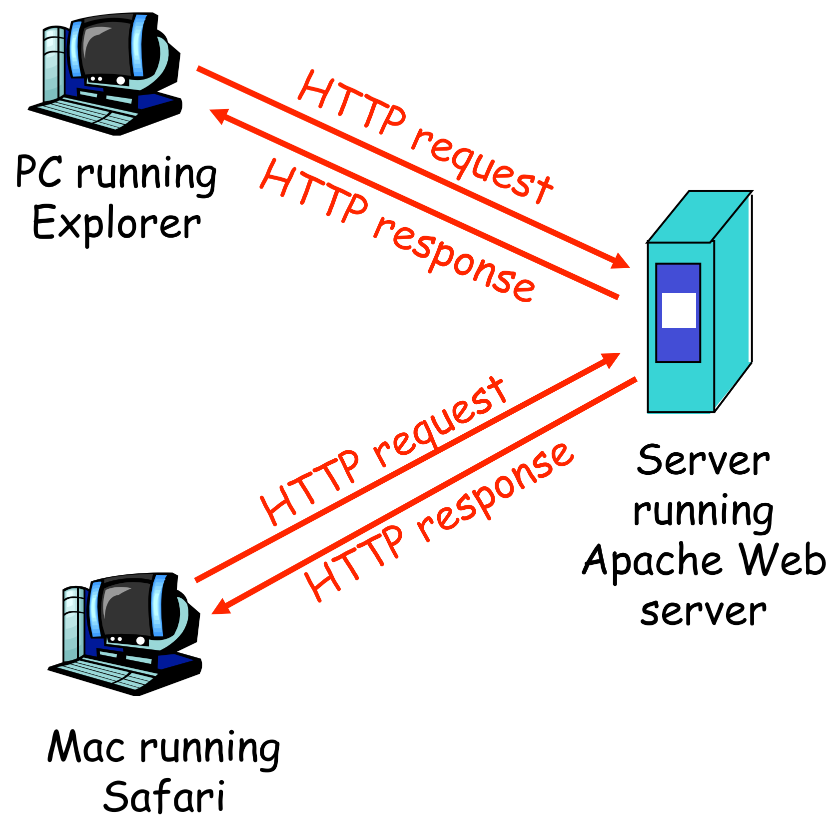
$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol [RFC1945, RFC2616] from 1999:
- HTTP 1.1: http://tools.ietf.org/html/rfc2616
- http://en.wikipedia.org/wiki/HTTP/2

- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests



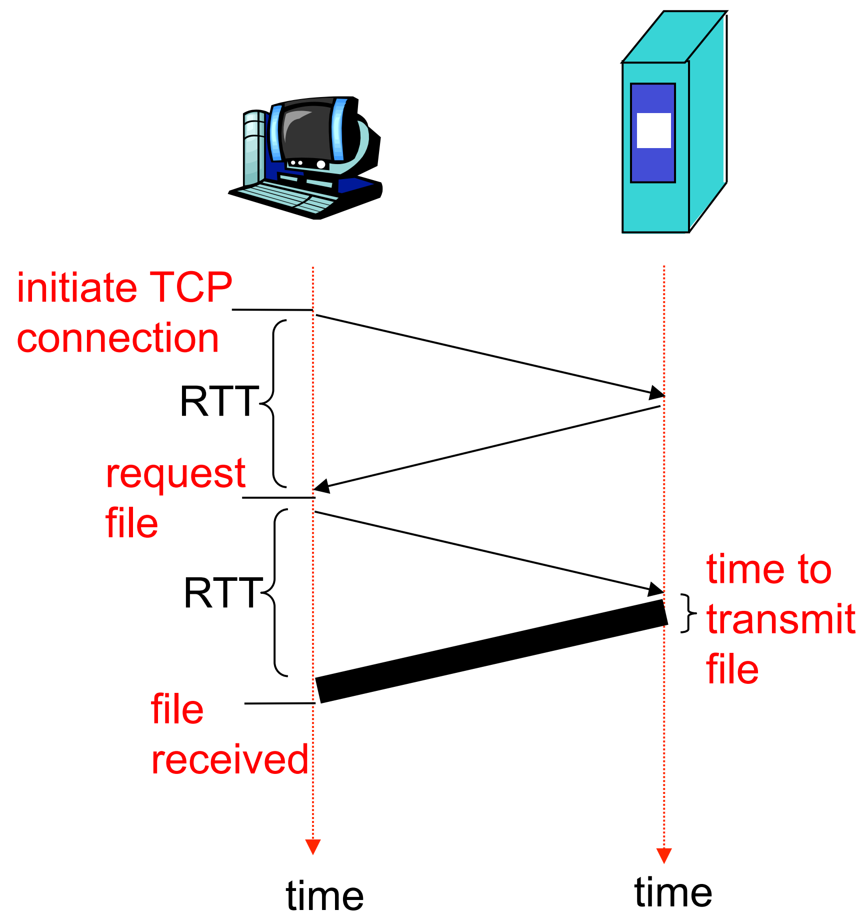
RTT & Response Time

definition of Round-Trip Time (RTT): time for 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+transmission time



HTTP overview (continued)

Uses TCP:

- client initiates TCP connection (creates socket) to server, port 80

  (https: port 443)
- 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.

persistent HTTP

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

Nonpersistent HTTP

suppose user enters URL:

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

(contains text, references to 10 jpeg images)

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

1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Nonpersistent HTTP (cont.)

4. HTTP server closes TCP connection.

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

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

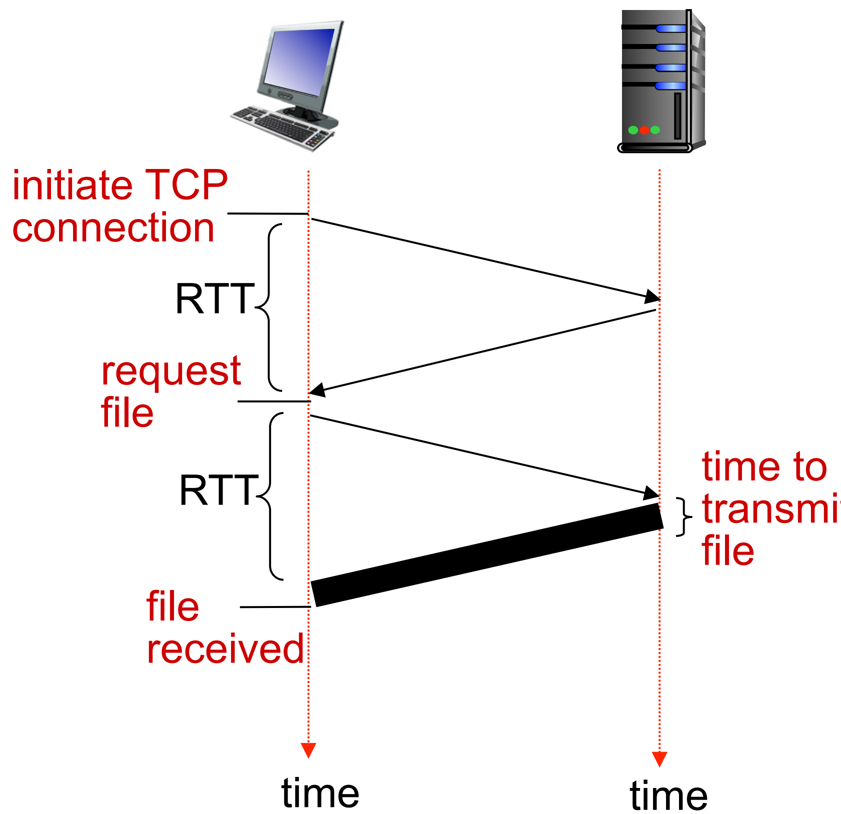
time

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
- file transmission time
- non-persistent HTTP response time =
  2RTT+ file transmission time



Non-Persistent vs. Persistent HTTP

non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- speedup: browsers often open [serial or] parallel (5-10) TCP connections to fetch referenced objects

persistent HTTP

- server leaves connection open after sending response, waiting for requests
- 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
- speedup: pipelining of requests and responses

HTTP overview

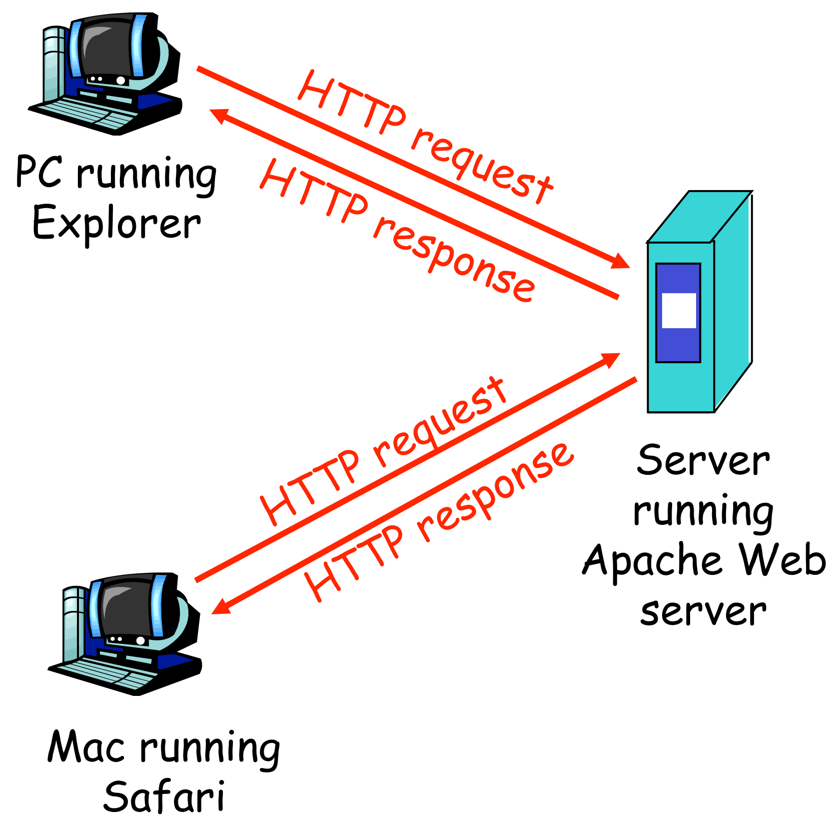
HTTP: hypertext transfer protocol

- Web's application layer protocol [RFC1945, RFC2616] from 1999:
- **HTTP 1.1:** http://tools.ietf.org/html/rfc2616
- http://en.wikipedia.org/wiki/HTTP/2

A protocol that allows browsers to talk to servers

Two types of HTTP messages:

1. *request*
2. *response*



HTTP request message

- ASCII (human-readable format)

carriage return character

object identifier

line-feed character

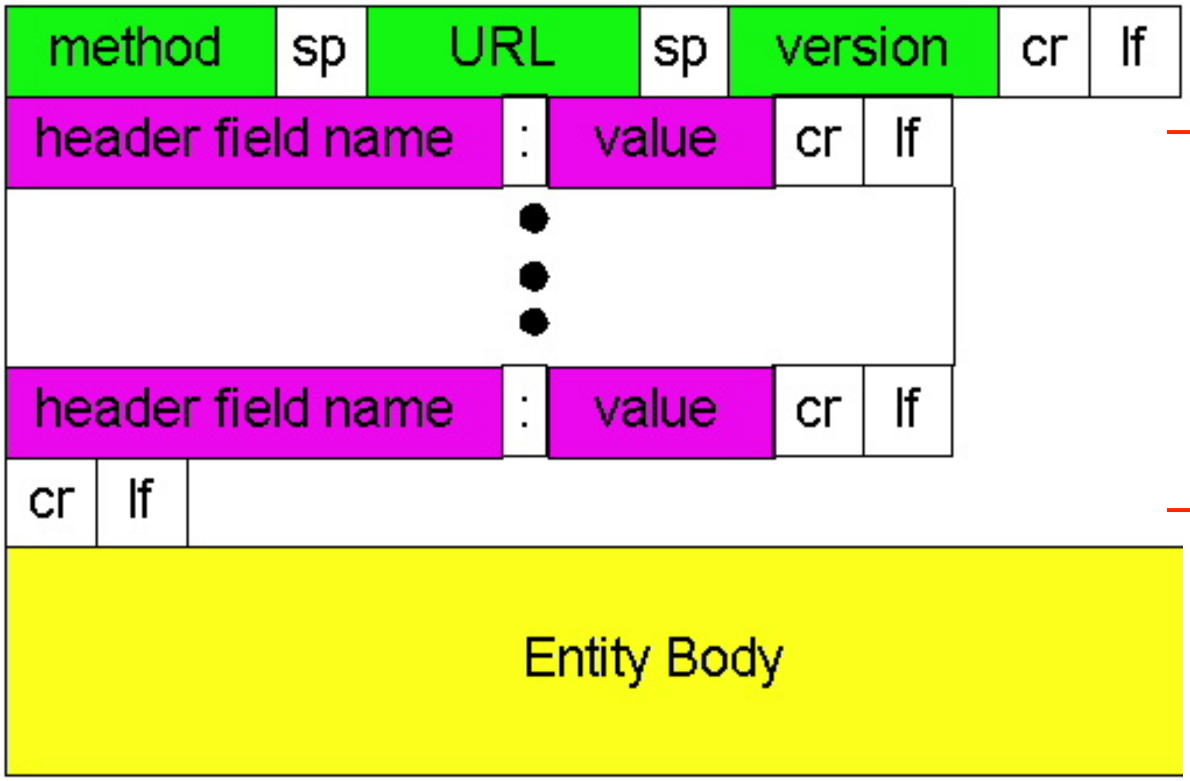
request line (GET, HEAD, POST method)

```
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
```

header lines

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

HTTP request message: general format



request line

request header lines

Body
Empty with GET
Full with POST

See HTTP 1.1: http://www.ietf.org/rfc/rfc2616.txt

Method types

- GET
- HEAD
  - asks server to leave requested object out of response
- POST
  - Still a request, but used to fill a form (e.g. search)
- PUT
  - uploads file in entity body to path specified in URL field (web publishing)
- DELETE
  - deletes file specified in the URL field

in HTTP 1.0 and HTTP/1.1

only in HTTP/1.1

Uploading form input

POST method:

- web page often includes form input
- input is uploaded to server in entity body
- answer of server depends on input

URL method:

- uses GET method
- input is uploaded in (extended) URL field of request line:

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

The role of your browser

- Your browser takes your input and constructs HTTP compliant requests

- Taking into account
  - browser type and version
  - configuration (e.g. language, type of TCP connections)
  - and user input

HTTP response message

status line: (protocol␣status code␣status phrase)

response header lines

```
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
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n (or Connection: close)
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

data, e.g., requested HTML file

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:) Client can retrieve new URL

  400 Bad Request
  - request msg not understood by server

  404 Not Found
  - requested document not found on this server

  505 HTTP Version Not Supported

- For more see HTTP 1.1: http://www.ietf.org/rfc/rfc2616.txt

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 is 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)

Try out HTTP (client side)

http://odysseas.calit2.uci.edu/doku.php/public:teaching-eecs148-s16

1. ssh to your favorite Web server:

```
telnet odysseas.calit2.uci.edu 80
```

opens TCP connection to port 80 (default HTTP server port) at odysseas.calit2.uci.edu. Anything typed in, is sent there.

2. type in a GET HTTP request:

```
GET /doku.php/public:teaching-eecs148-s16/ HTTP/1.1
Host: odysseas.calit2.uci.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)

Try out HTTP -variations

http://odysseas.calit2.uci.edu/doku.php/public:teaching-eecs148-s16

1. ssh to your favorite Web server:

```
telnet odysseas.calit2.uci.edu 80
```

2. type in a GET HTTP request:

```
GET /doku.php/public:teaching-eecs148-s16/ HTTP/1.1
Host: odysseas.calit2.uci.edu
```

4. Try variations …

```
GET /doku.php/public:teaching-eecs148-s16 HTTP/1.1
Host: odysseas.calit2.uci.edu

GET /doku.php/public:teaching-eecs148-s16/ HTTP/1.1
Host: odysseas.calit2.uci.edu
Keep-Alive: timeout=100, max=100
[repeat request]

HEAD /doku.php/public:teaching-eecs148-s16 HTTP/1.1
Host: odysseas.calit2.uci.edu
```

Add-on 1: User-server state: cookies

many Web sites use cookies

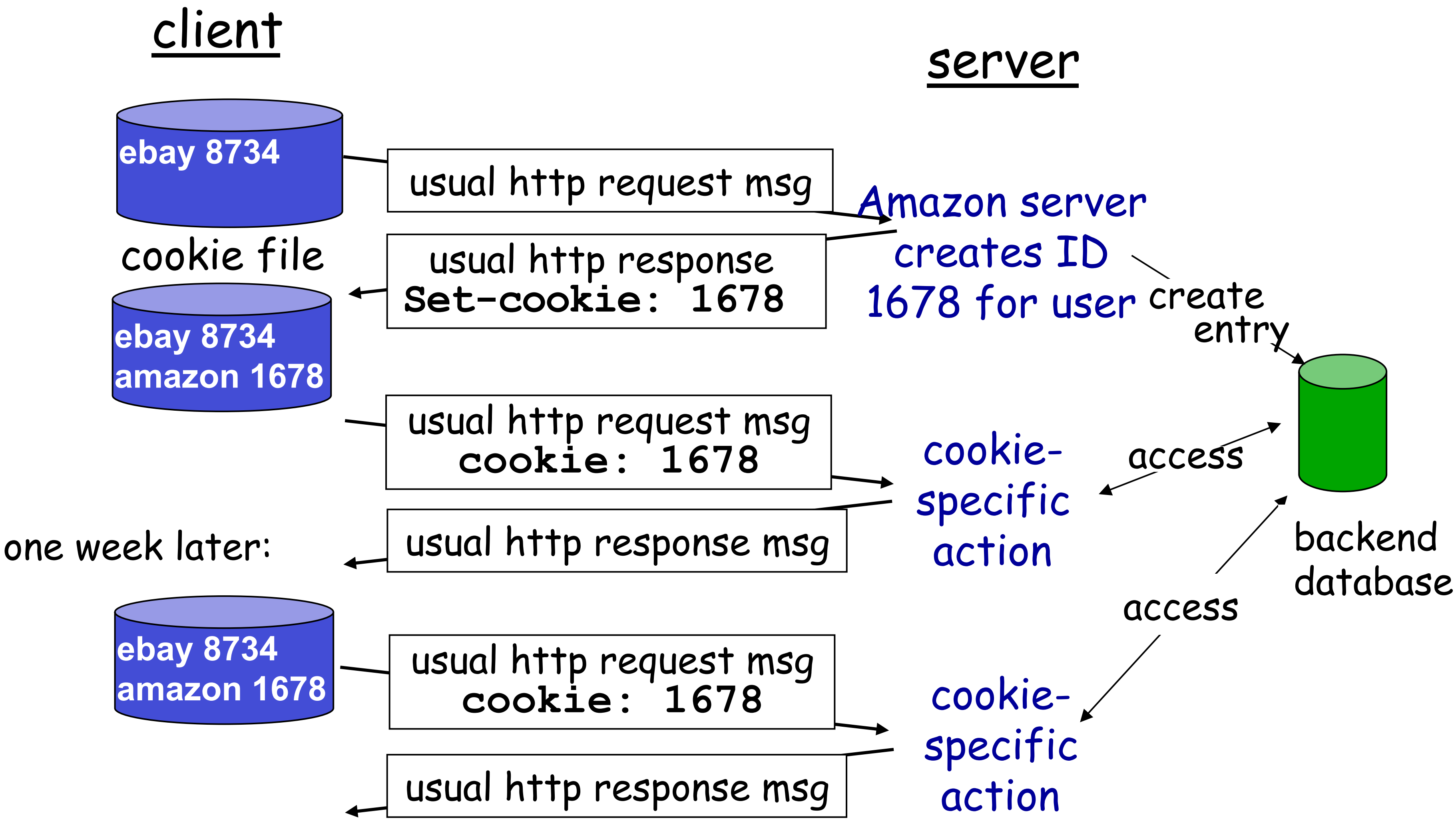
four components:

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 always accesses Internet from laptop
- 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.)



Cookies (continued)

cookies can be used for:

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

aside

cookies and privacy:

- cookies permit sites to learn a lot about you
- you may register+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

FAQ on cookies

Q1: If I delete my cookie, next time I visit the same webserver, will I be assigned the same? Or will there be a mapping between the new and old cookie?

A1: Not through HTTP (stateless). Although the server could do the mapping.

FAQ on cookies

Q2: If somebody steals my cookies file (or overhears my cookies sent in the clear over wireless), can he login (i.e., authenticate) as me?

A2: Yes, as long as the cookie is still valid.

FAQ on cookies

Q2 cont'd: Are there any mechanisms in place to prevent others from using my cookie?

A2 cont'd: YES, through authentication mechanisms.

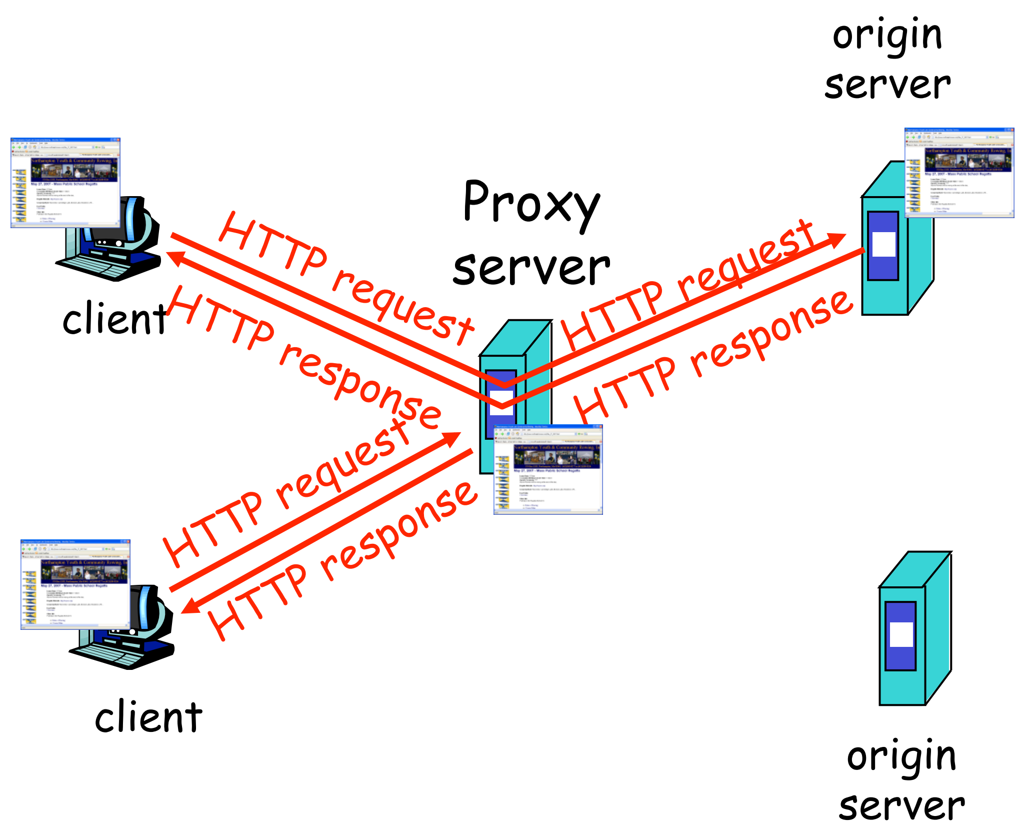
E.g. see http://sconce.ics.uci.edu/cs203-w12/lec4-web-auth.pdf

- HTTPS encodes the cookie so it can't be overheard
- Session ID is part of the cookie
  - While you are logged & active in the website the cookie is valid; then the server makes it invalid.
- Server provides cookie:authenticator (the latter is not easily forgeable)
- The mechanisms are not bullet-proof but put the bar higher.

Add-on 2: 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?

- to reduce response time for client request
- to reduce traffic on an institution's access link.
- Internet dense with caches: enables "poor" content providers to effectively deliver content

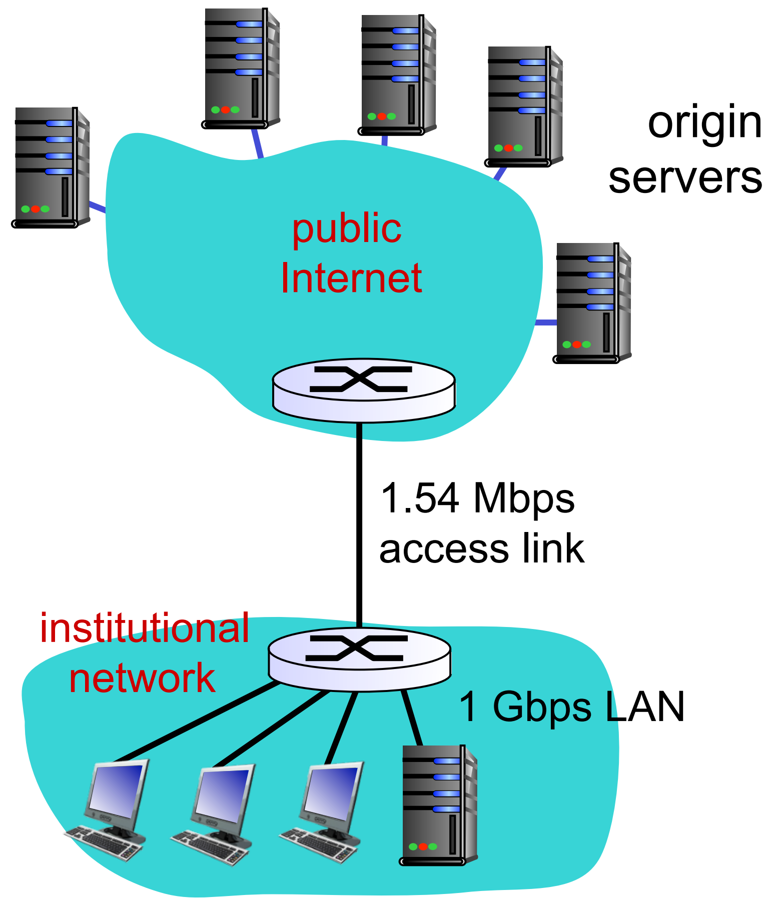
Caching example:

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



Caching example: faster access link

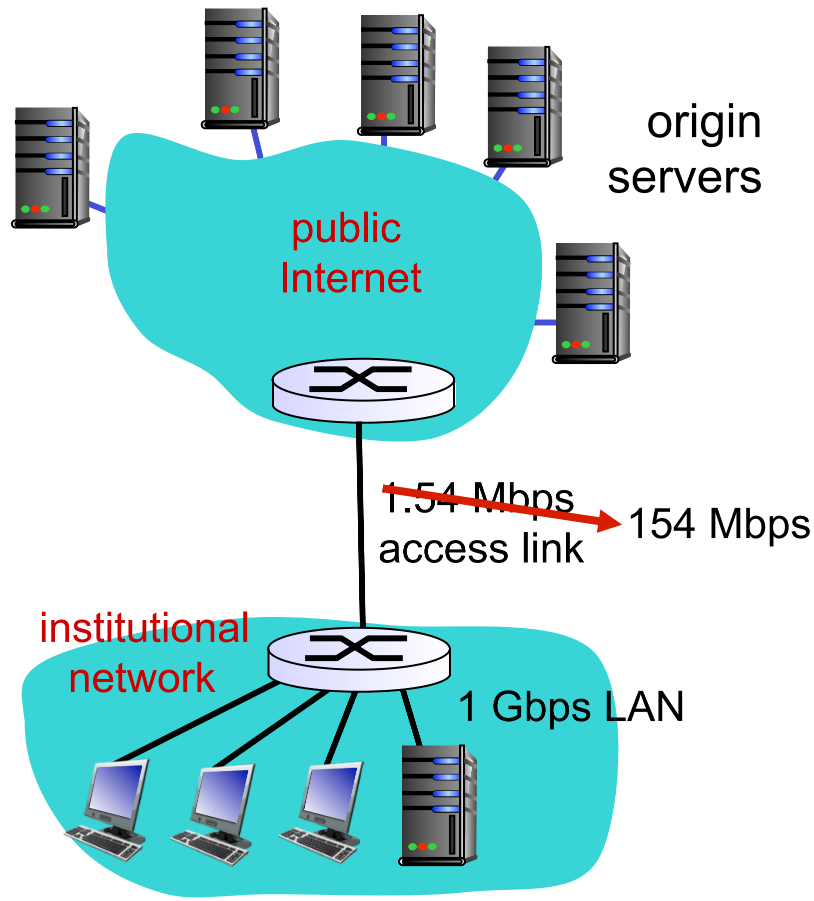
assumptions:

- 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%~~ 9.9%
- access link utilization = 99%
- total delay = Internet delay + access delay + LAN delay

  = 2 sec + ~~minutes~~ msecs + usecs



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

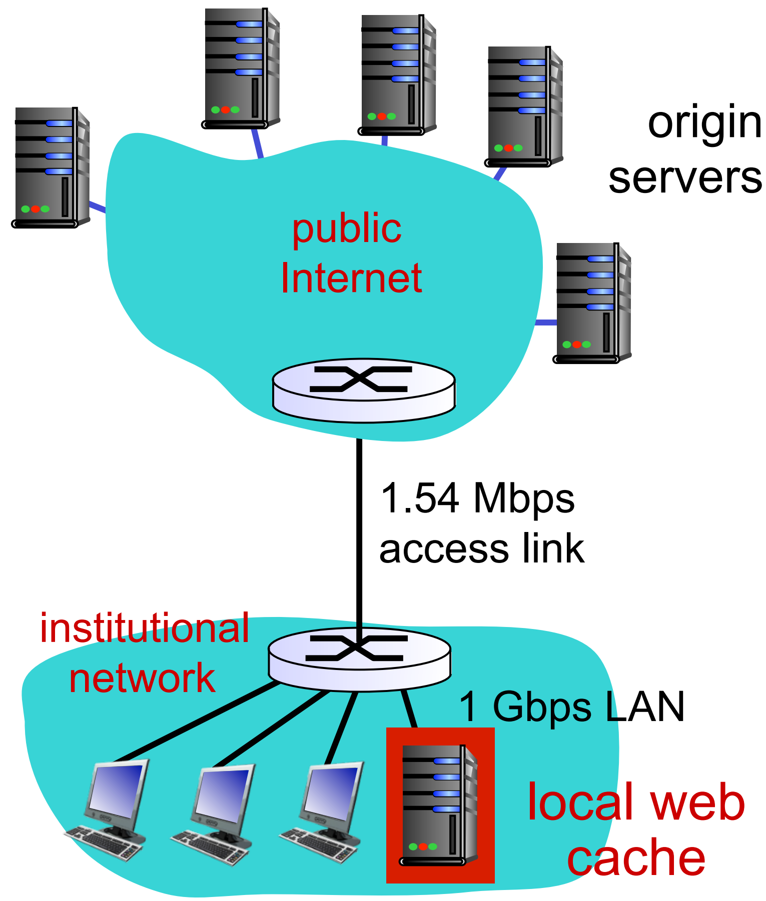
- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

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

How to compute link utilization, delay?

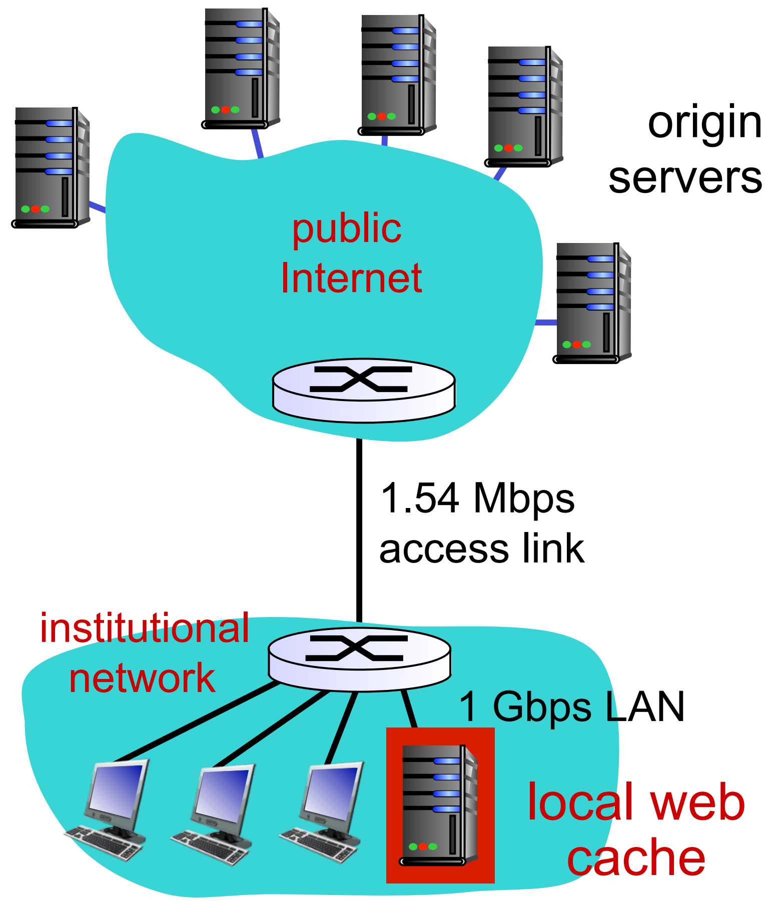
Cost: web cache (cheap!)



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 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58
- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs)
  - = ~ 1.2 secs
  - less than with 154 Mbps link (and cheaper too!)

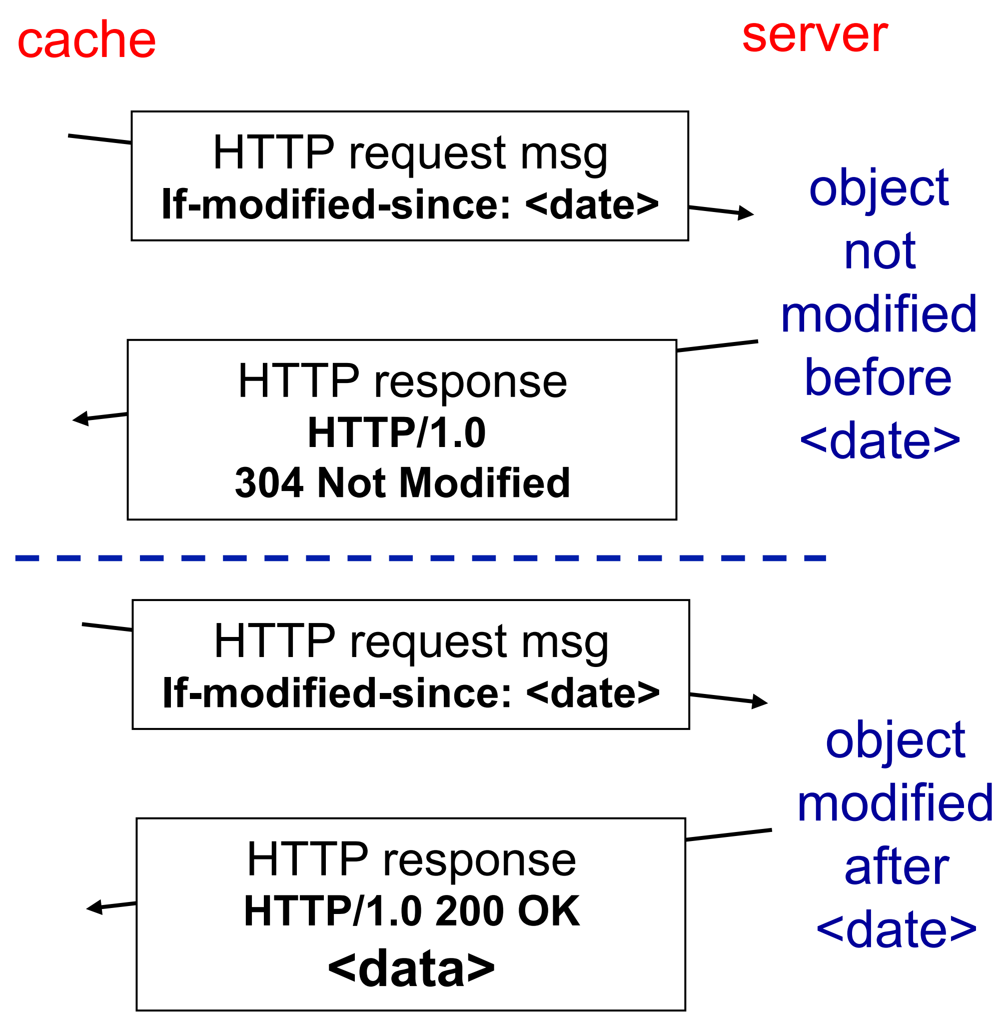


Conditional GET

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

  If-modified-since: <date>
- server: response contains no object if cached copy is up-to-date:

  HTTP/1.0 304 Not Modified



Chapter 2: Application layer

2.1 Principles of network applications

2.2 Web and HTTP

RFC2616: http://tools.ietf.org/html/rfc2616

Look at your browser's preferences/settings

Funny: http://www.adamgrant.net/#!originals/c1ckh page 4+: Chrome, Firefox vs IE, Safari

2.3 FTP

2.4 Electronic Mail

- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 Socket programming with TCP

2.8 Socket programming with 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

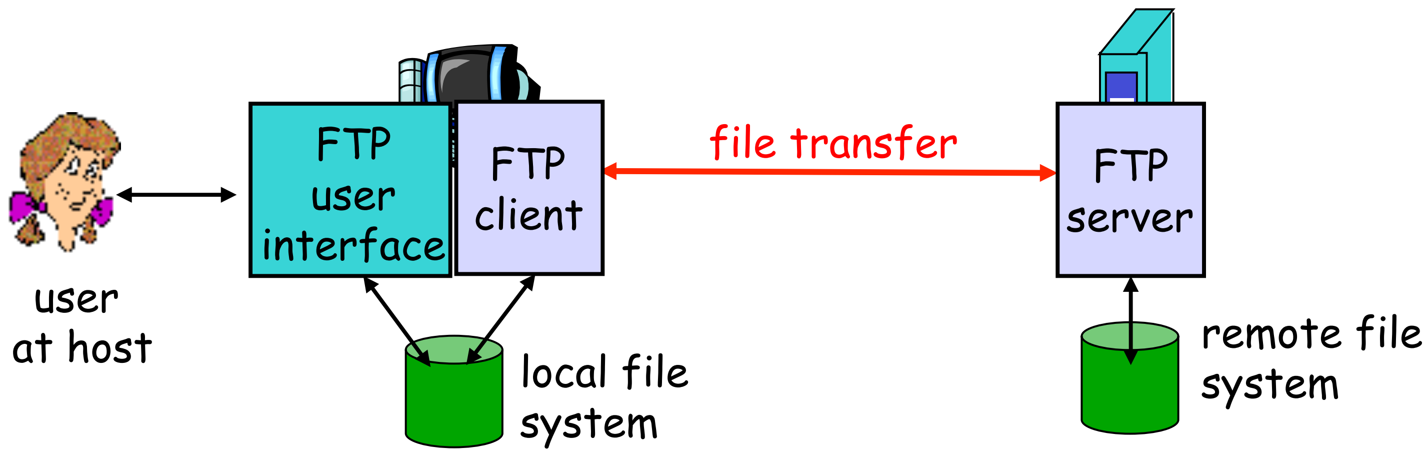
2.5 DNS

2.6 P2P applications

2.7 Socket programming with TCP

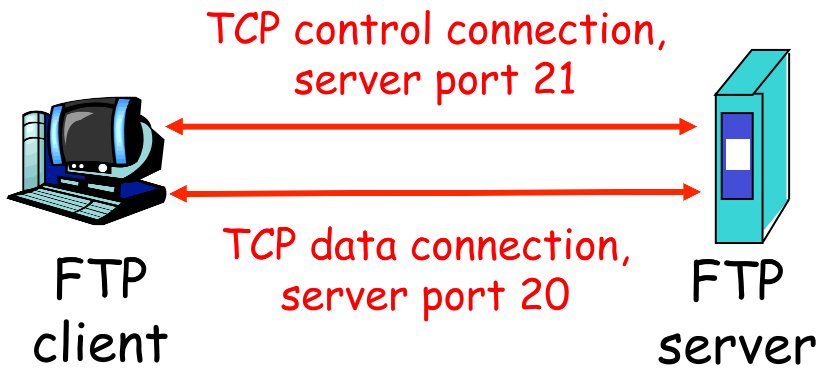
2.8 Socket programming with UDP

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 server: port 21
- Many client programs: ftp, winscp, filezilla, putty, dropbox ….
- ftp: RFC 959, since 1985: http://www.ietf.org/rfc/rfc959.txt

FTP: separate control, data connections



- FTP client contacts FTP server at port 21, uses TCP as the transport protocol
- client authorized over control connection
- client browses remote directory by sending commands over control connection.
- when server receives file transfer command, server opens *2nd* TCP connection (data connection) 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 files in current directory (ls)
- `RETR filename` retrieves (gets) file
- `STOR filename` stores (puts) file onto remote host
- `QUIT` closes connection (bye)

sample return codes

- status code and optional 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`

For details see RFC 959: http://tools.ietf.org/html/rfc959