TDTS04/11: Computer Networks

Instructor: Niklas Carlsson

Email: niklas.carlsson@liu.se

Notes derived from "*Computer Networking: A Top Down Approach*", by Jim Kurose and Keith Ross, Addison-Wesley.

The slides are adapted and modified based on slides from the book's companion Web site, as well as modified slides by Anirban Mahanti and Carey Williamson.

Outline

- Background
- Introduction to App-Layer Protocols
- Brief History of WWW
- Architecture
- HTTP Connections
- HTTP Format
- Web Performance
- Cookies

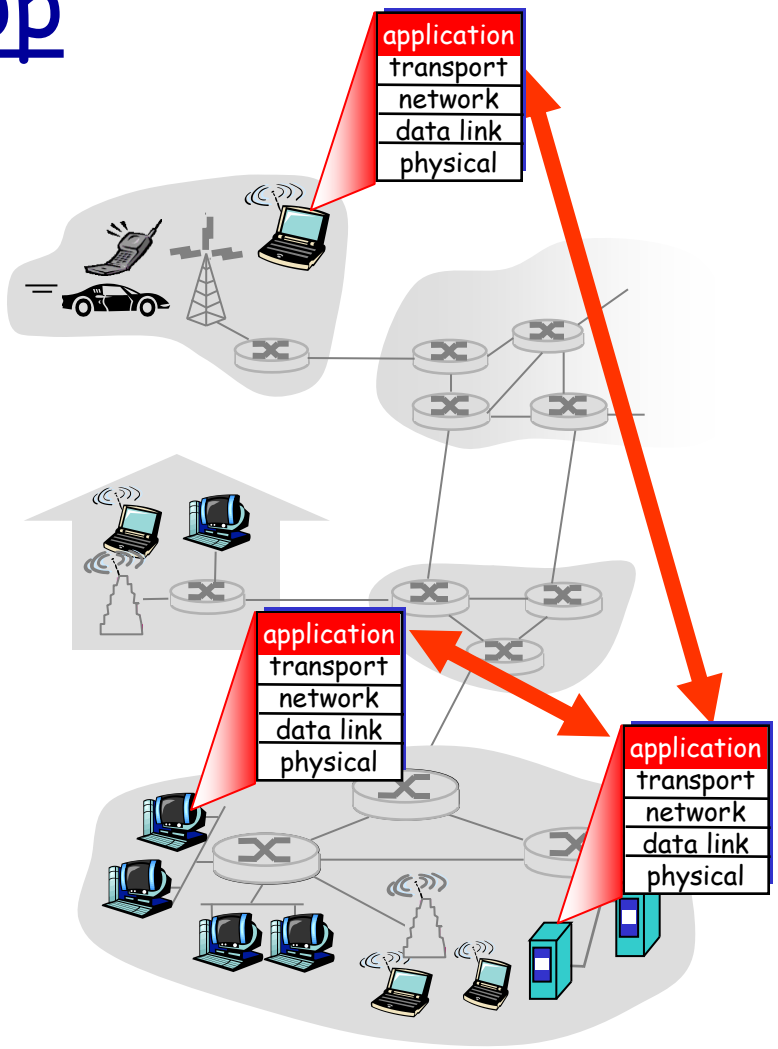
Creating a network app

write programs that

- run on (different) *end systems*
- communicate over network

No need to write software for network-core devices

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



App-layer protocols

protocols define format, order of msgs sent and received among network entities, and actions taken on msg transmission, receipt

App-layer protocols

protocols define format, order of msgs sent and received among network entities, and actions taken on msg transmission, receipt

public-domain protocols:

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

proprietary protocols:

- e.g., Skype, Spotify

Application architectures

- client-server
- peer-to-peer (P2P)
- hybrid of client-server and P2P

Network applications: some jargon

Process: program running within a host.

- within same host, two processes communicate using inter-process communication (IPC, defined by OS).
- processes running on different hosts communicate with an application-layer protocol

User agent: interfaces with user "above" and network "below".

- implements user interface & application-level protocol
  - Web: browser
  - E-mail: mail reader
  - streaming audio/video: media player

Processes communicating

Process: program running within a host.

- processes in different hosts communicate by exchanging messages

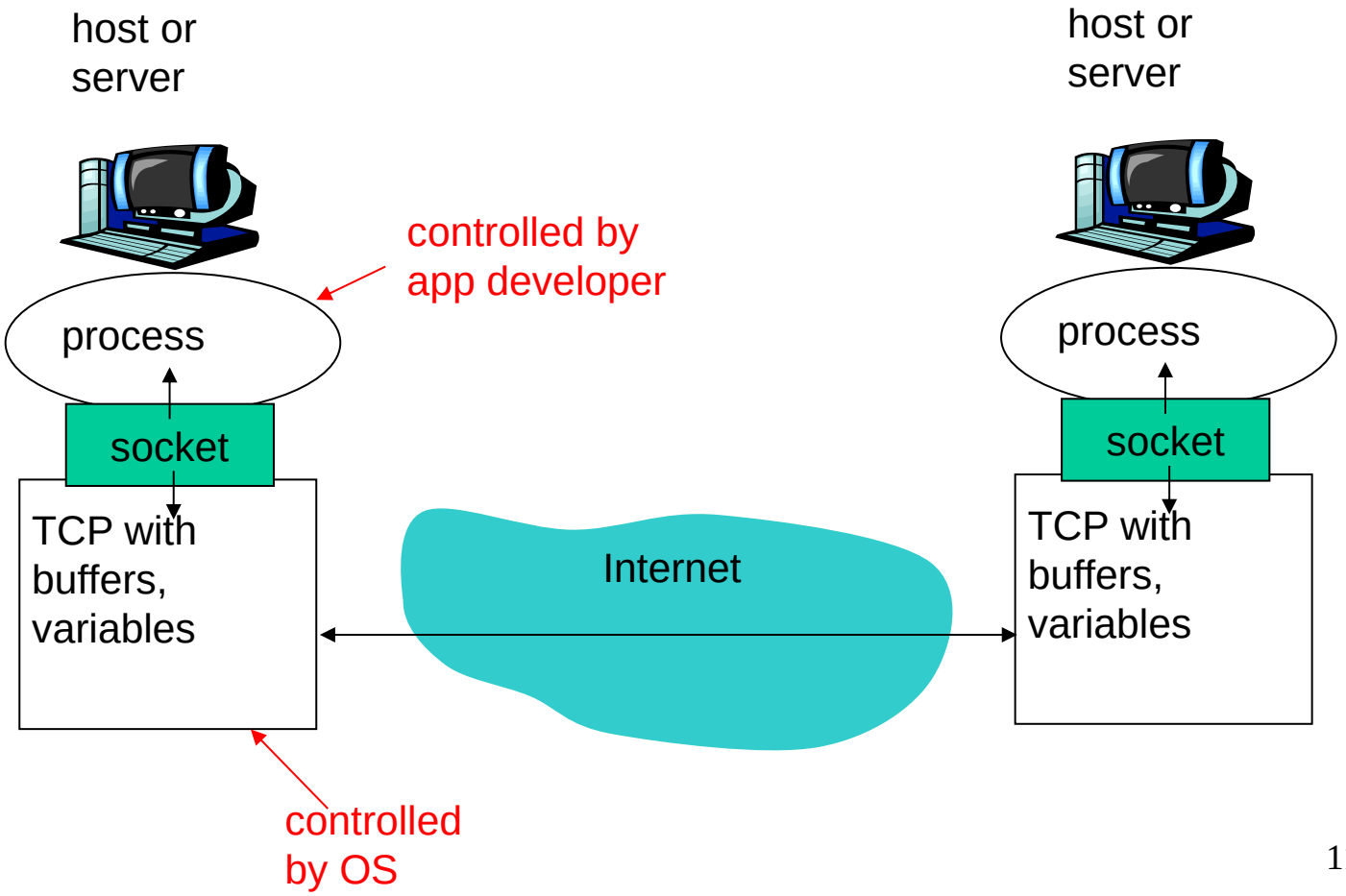
Client-server paradigm

client process: process that initiates communication

server process: process that waits to be contacted

Sockets

- process sends/receives messages to/from its socket



Addressing processes:

- For a process to receive messages, it must have an identifier
- Identifier includes both the IP address and port numbers associated with the process on the host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- More on this later

Addressing processes:

- For a process to receive messages, it must have an identifier
- Every host has a unique 32-bit IP address
- Q: does the IP address of the host on which the process runs suffice for identifying the process?
- Answer: No, many processes can be running on same host

- Identifier includes both the IP address and port numbers associated with the process on the host.

- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- More on this later

What transport service does an app need?

Protocol: Connection oriented or not?

Connection oriented:

- *Hand shaking*
  - *Explicit setup phase for logical connection*
  - *Connection release afterwards*
- *Establishes state information about the connection*
- *Mechanisms for*
  - *reliable data transfer, error control, flow control, etc.*
- *Guarantees that data will arrive (eventually)*

Protocol: Connection oriented or not?

Connection oriented:

- *Hand shaking*
  - *Explicit setup phase for logical connection*
  - *Connection release afterwards*
- *Establishes state information about the connection*
- *Mechanisms for*
  - *reliable data transfer, error control, flow control, etc.*
- *Guarantees that data will arrive (eventually)*

Connection less:

- No handshaking
- No (significant) state information (at end points or in network)
- No mechanisms for flow control etc.
- No guarantees of arrival (or when)
- Simpler (and faster?)

Protocol: Connection oriented or not?

Connection oriented:

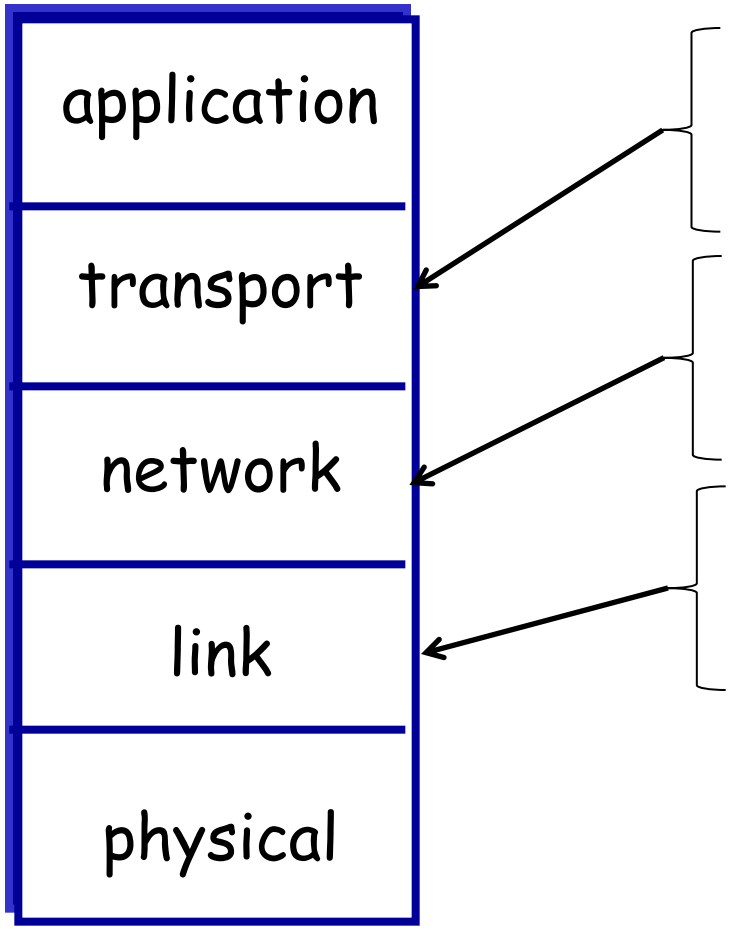
- *Hand shaking*
  - *Explicit setup phase for logical connection*
  - *Connection release afterwards*
- *Establishes state information about the connection*
- *Mechanisms for*
  - *reliable data transfer, error control, flow control, etc.*
- *Guarantees that data will arrive (eventually)*

Connection less:

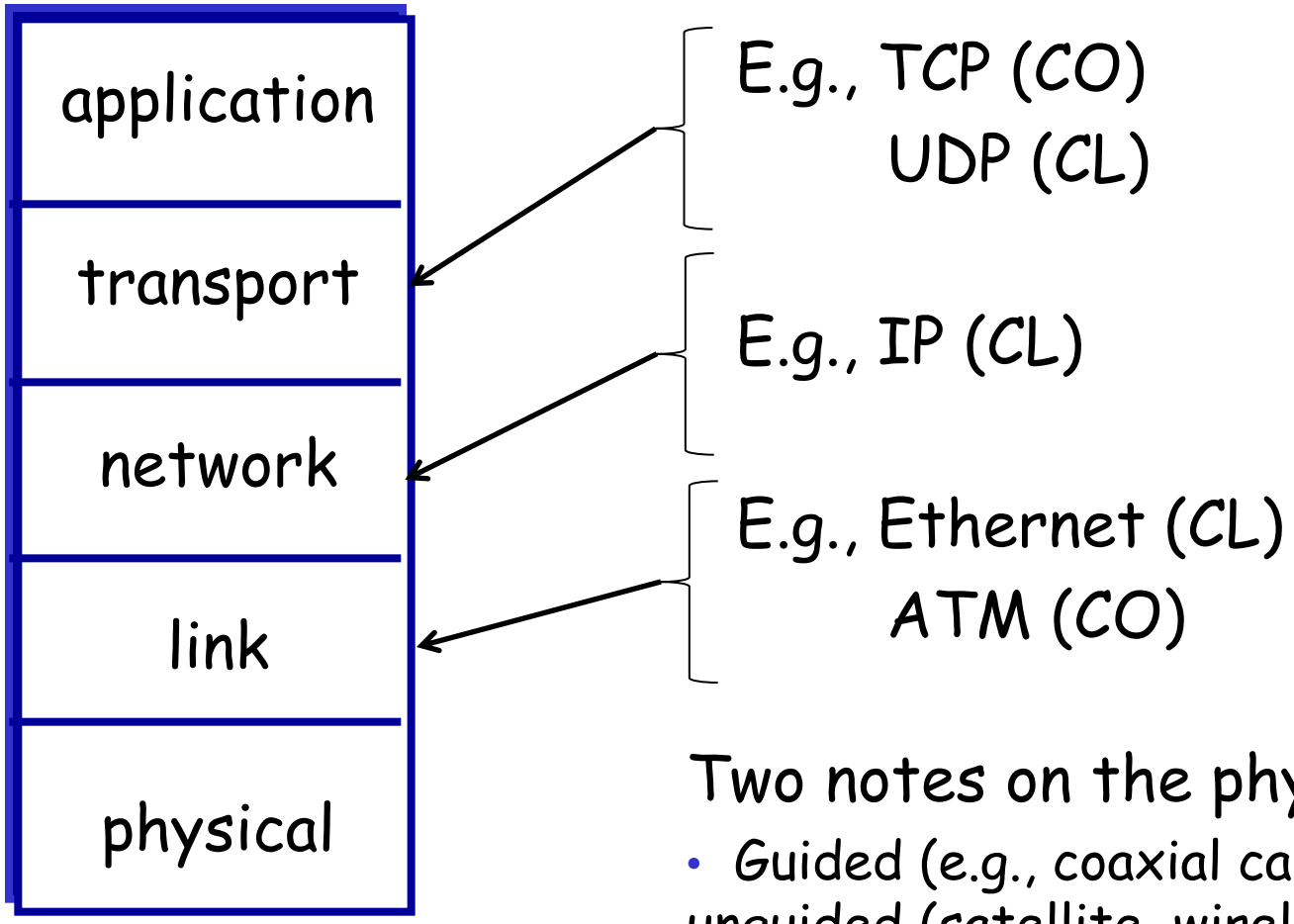
- No handshaking
- No (significant) state information (at end points or in network)
- No mechanisms for flow control etc.
- No guarantees of arrival (or when)
- Simpler (and faster?)

Which is the best? It depends on
(i) what it is used for, and
(ii) what it is built ontop of …

Internet protocol stack



Internet protocol stack



Two notes on the physical layer:

- Guided (e.g., coaxial cable, fiber, etc) vs. unguided (satellite, wireless, etc.)
- Signaling, modulation, encoding, etc,

What transport service does an app need?

Data loss

Bandwidth

Timing

What transport service does an app need?

Data loss

- some apps (e.g., file transfer, telnet) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

Timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

Bandwidth

- most apps ("elastic apps") make use of whatever bandwidth they get
- other apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"

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 throughput guarantees, security

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, throughput guarantee, or security

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 2821]	
remote terminal access	Telnet [RFC 854]	
Web	HTTP [RFC 2616]	
file transfer	FTP [RFC 959]	
streaming multimedia	proprietary (e.g., RealNetworks, youtube, netflix, spotify)	
Internet telephony	proprietary (e.g., Dialpad, skype)	

Internet apps: application, transport protocols

Application	Application layer protocol	Underlying transport protocol
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, youtube, netflix, spotify)	TCP (or UDP)
Internet telephony	proprietary (e.g., Dialpad, skype)	UDP or TCP typically UDP

- WWW terminology and HTTP overview

Some "Web" Terminology

- Web page may contain links to other pages (sometimes also called Web Objects)
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web pages are "Hypertexts"
  - One page points to another

- Each object is addressable by a URL:

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

protocol host name path name

HTTP overview

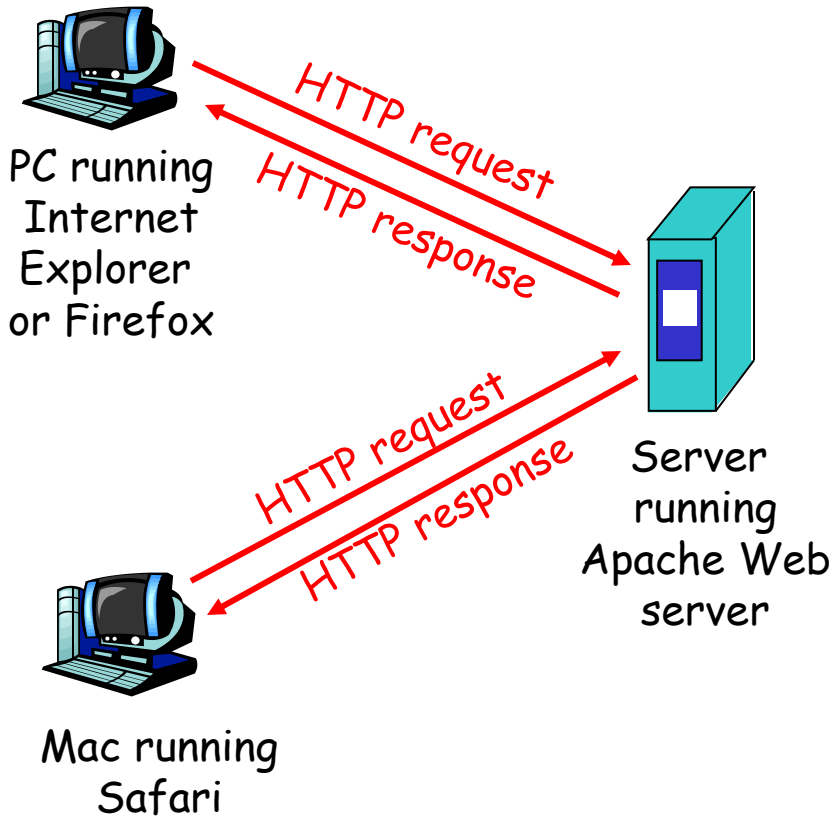
HTTP: hypertext transfer protocol

- Web's application layer protocol

HTTP overview

HTTP: hypertext transfer protocol

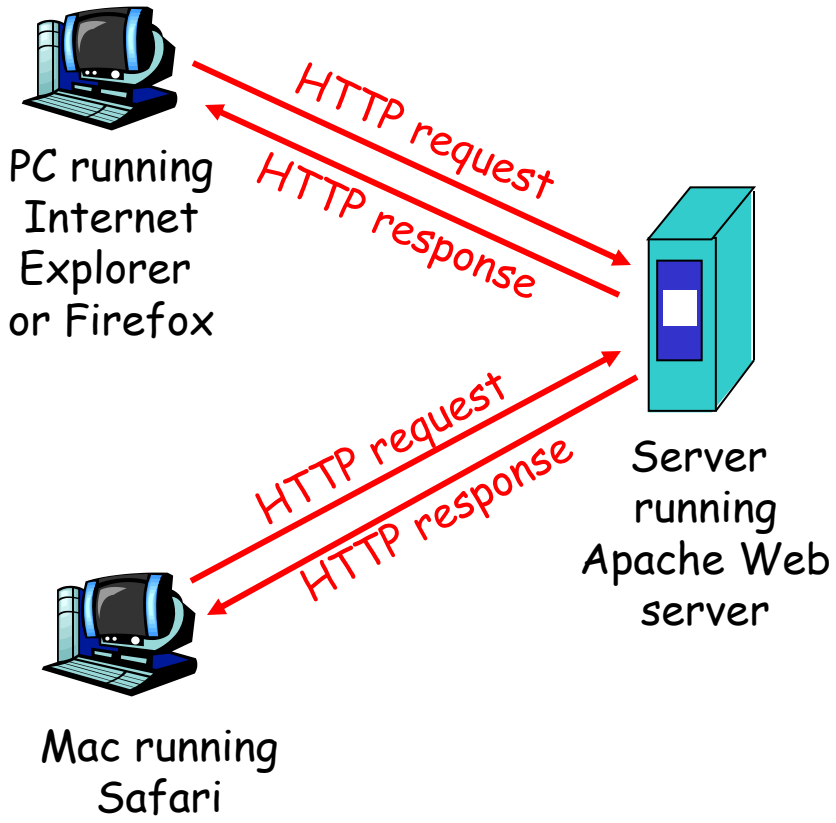
- Web's application layer protocol



HTTP overview

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 2616



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

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

Outline

- Introduction to App Layer Protocols
- Brief History of WWW
- Architecture
- HTTP Connections
- HTTP Format
- Web Performance
- Cookies

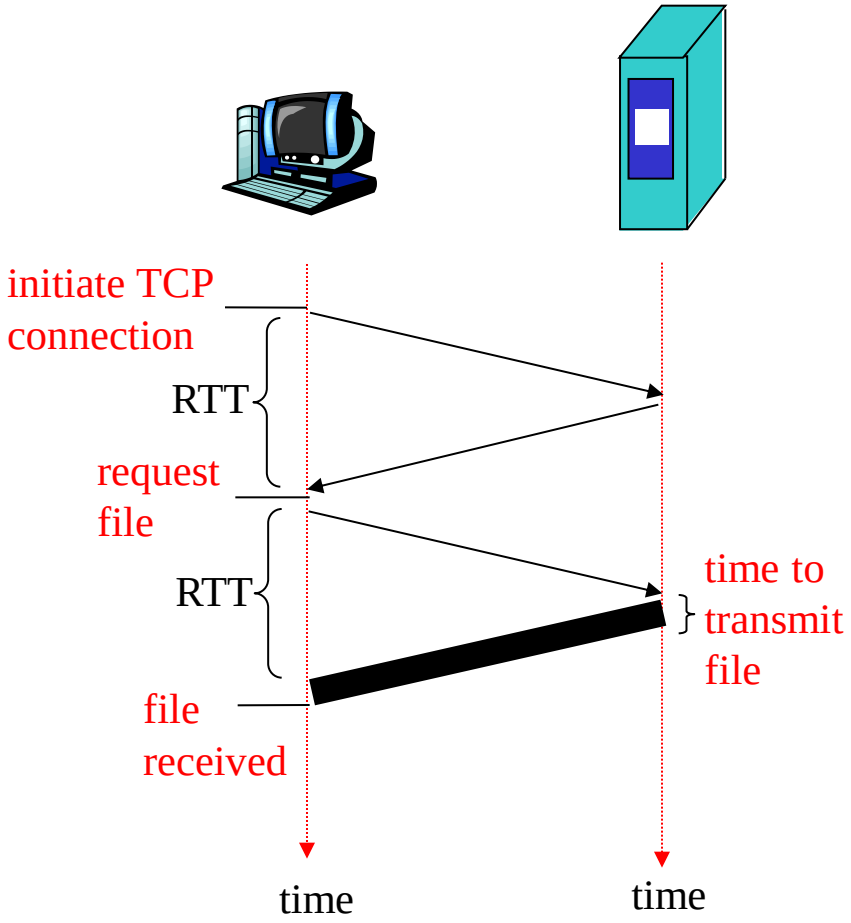
Response time modeling

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 = 2*RTT+transmit time



HTTP connections

Non-persistent HTTP

- At most one object is sent over a TCP connection.
- HTTP/1.0 uses non-persistent HTTP

Persistent HTTP

- Multiple objects can be sent (one at a time) over a single TCP connection between client and server.
- HTTP/1.1 uses persistent connections in default mode
  - Pipelined
  - Non-pipelined

Persistent HTTP

Nonpersistent HTTP issues:

- requires 2 RTTs per object
- OS must work and allocate host resources for each TCP connection
- but 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 are sent over connection

Persistent without pipelining:

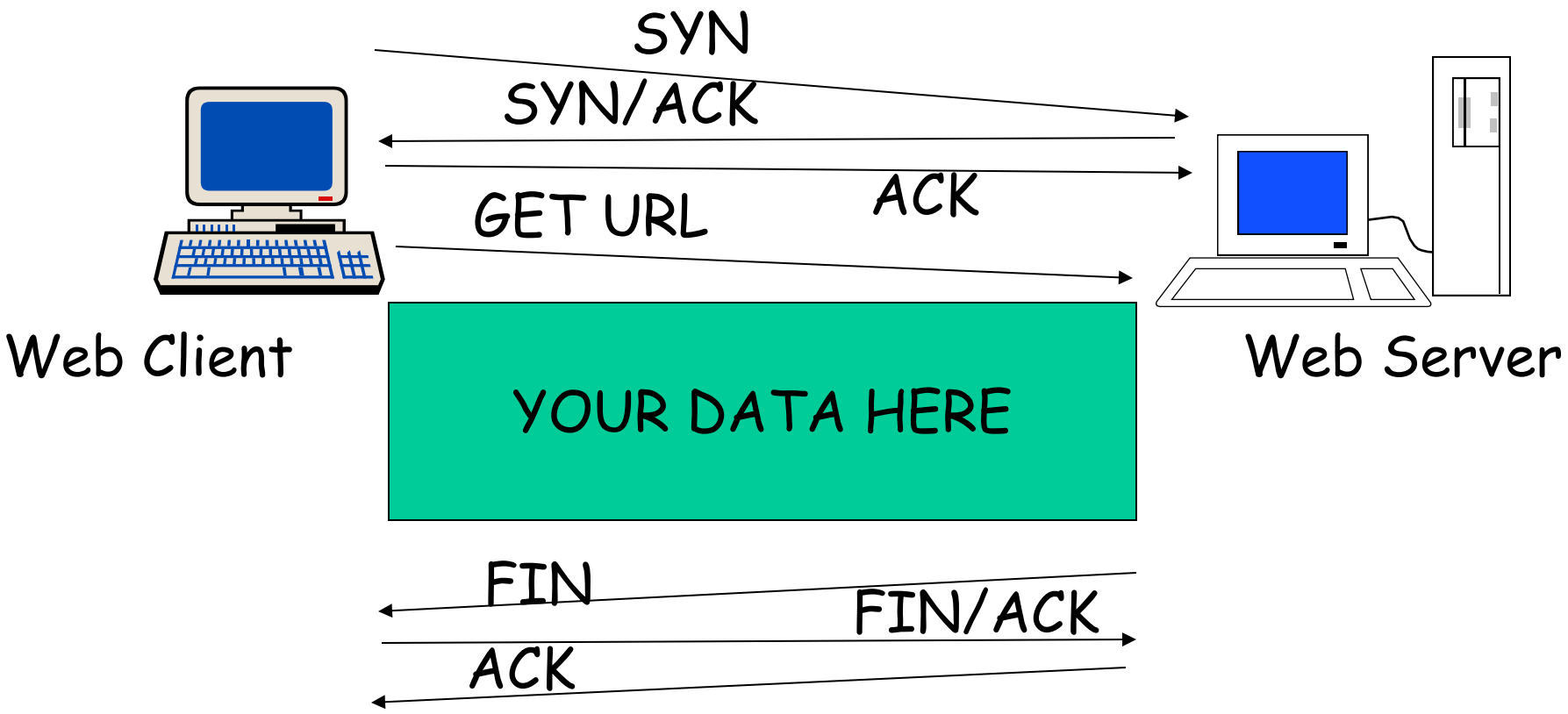
- client issues new request only when previous response has been received
- one RTT for each referenced object

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

Network View: HTTP and TCP

- TCP is a connection-oriented protocol



Example Web Page

page.html

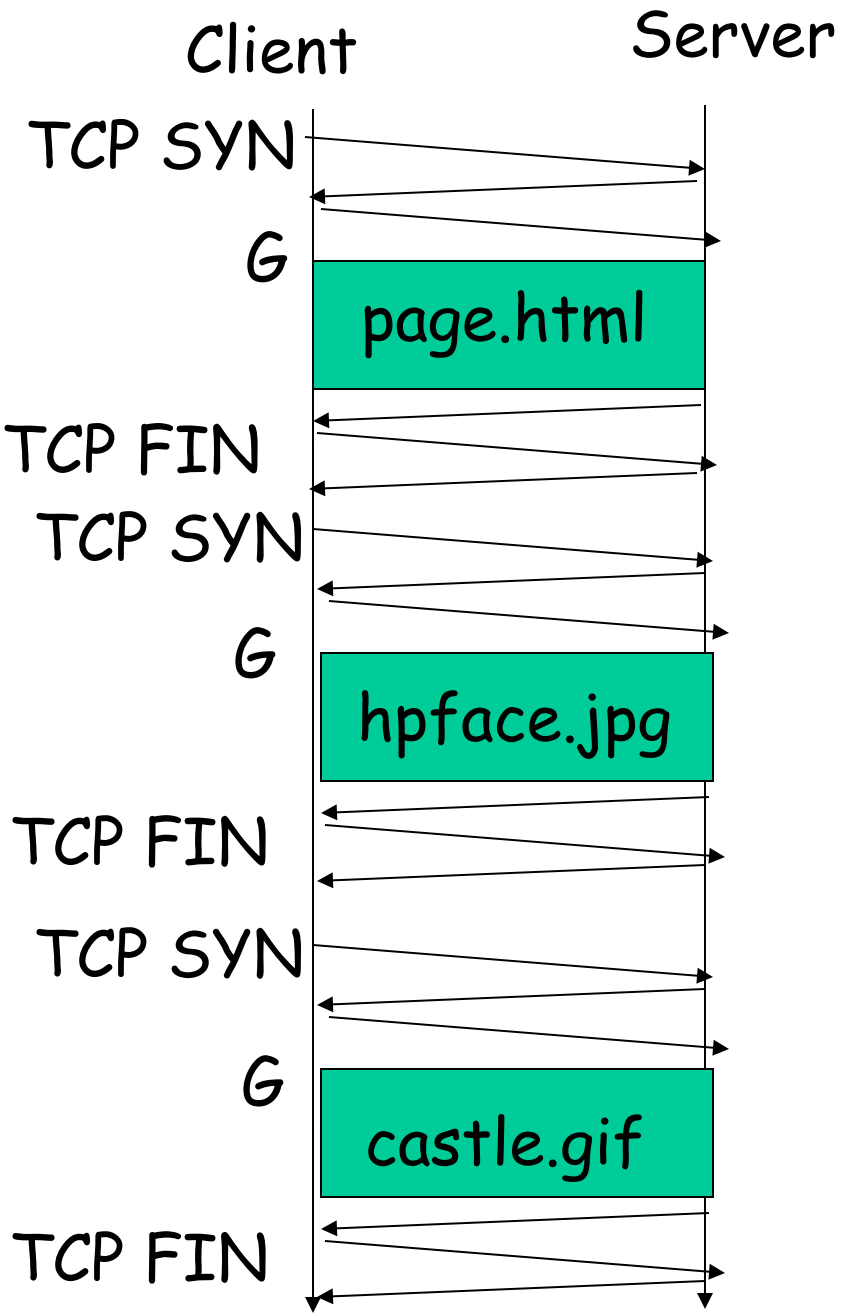
Harry Potter Movies

As you all know,
the new HP book
will be out in June
and then there will
be a new movie
shortly after that…

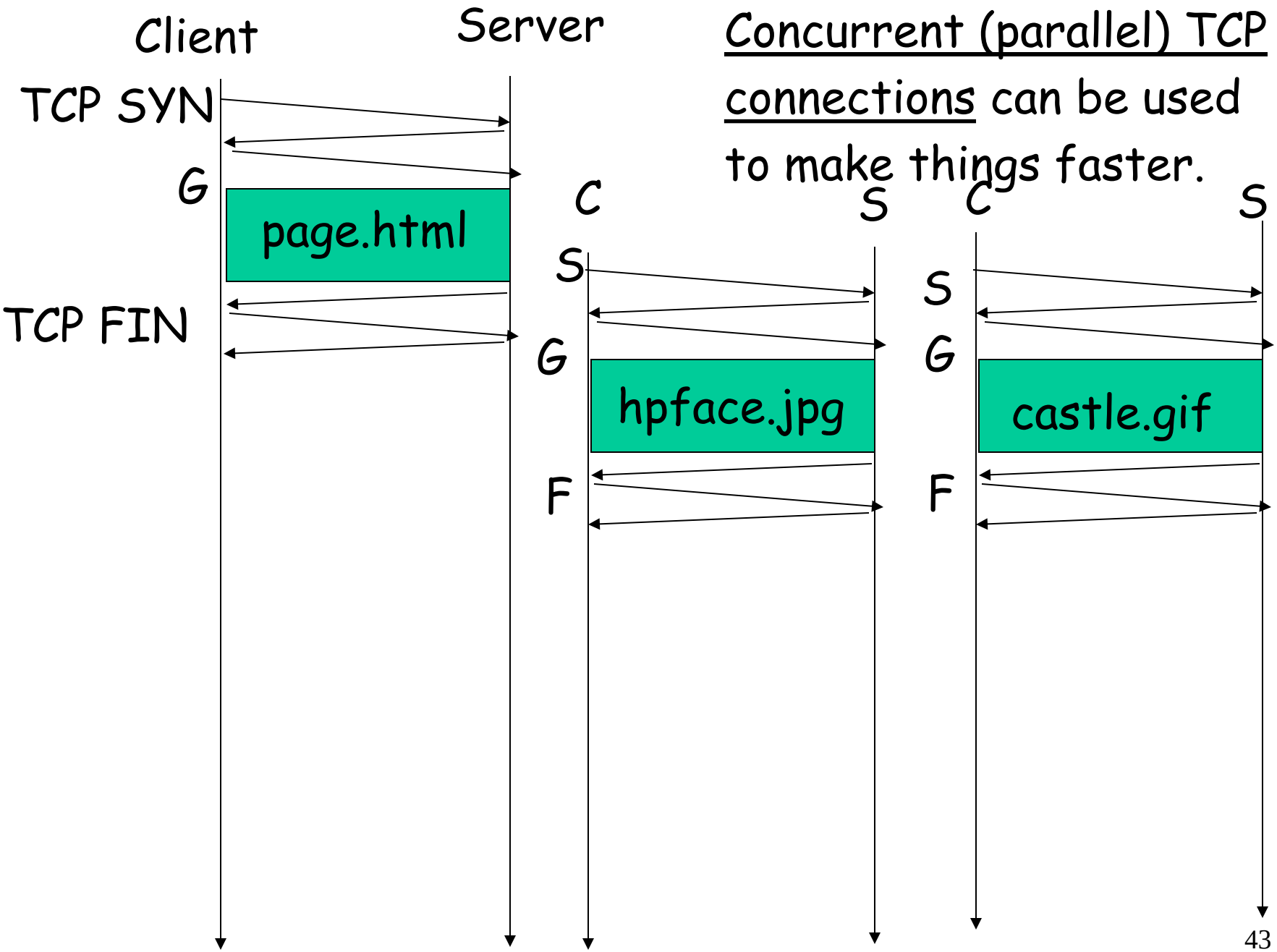
hpface.jpg

castle.gif

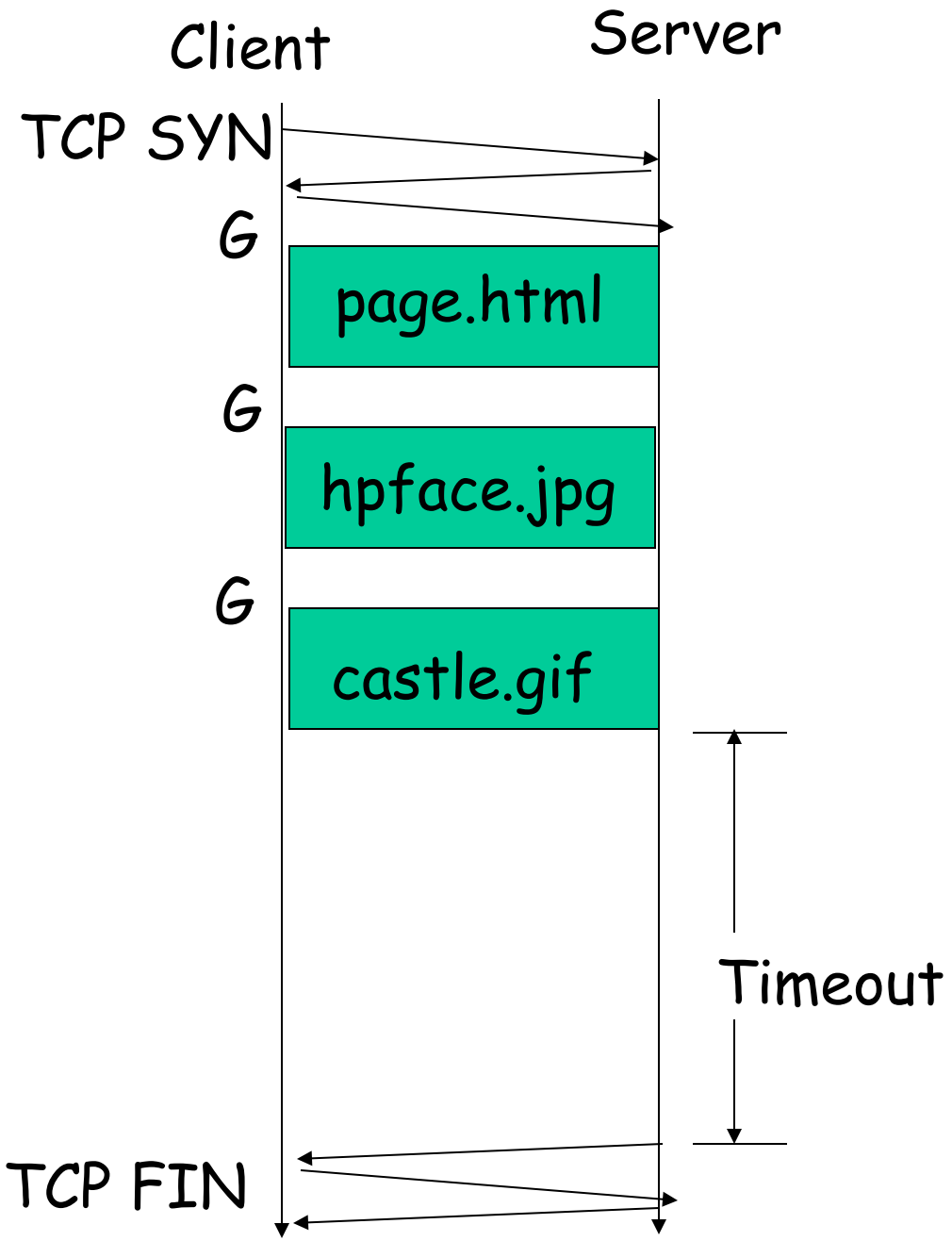
"Harry Potter and
the Bathtub Ring



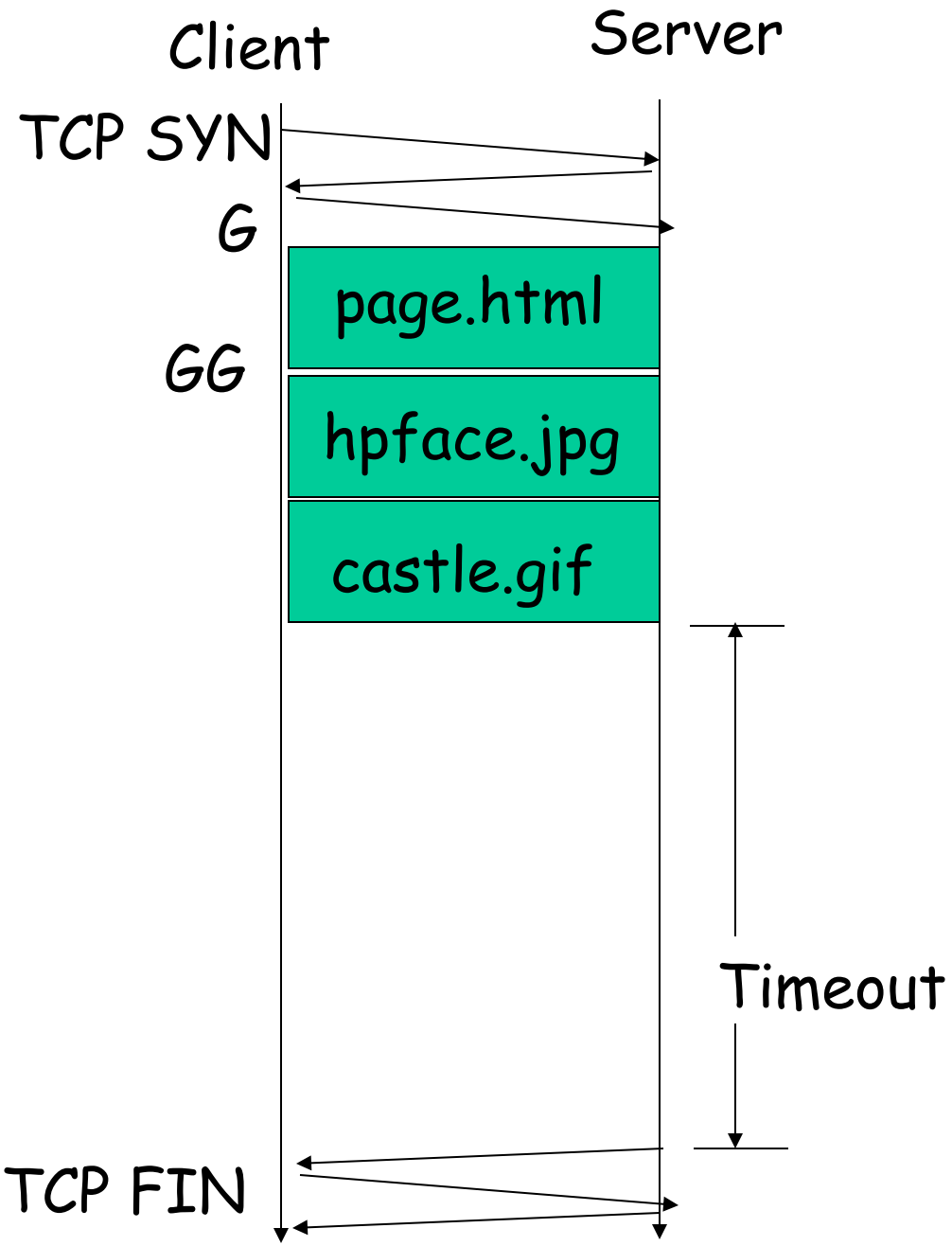
The "classic" approach in HTTP/1.0 is to use one HTTP request per TCP connection, serially.



Concurrent (parallel) TCP connections can be used to make things faster.



The "persistent HTTP" approach can re-use the same TCP connection for Multiple HTTP transfers, one after another, serially Amortizes TCP overhead, but maintains TCP state longer at server.



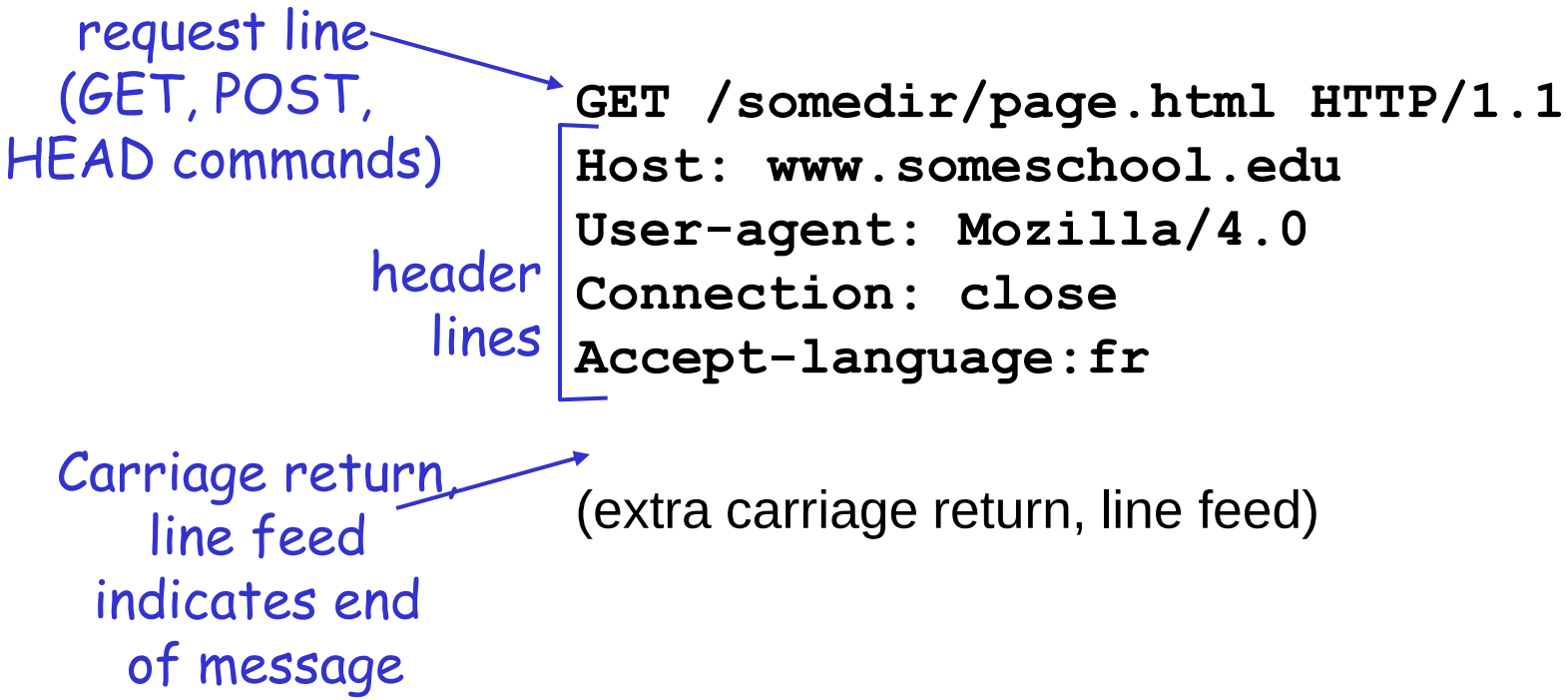
The "pipelining" feature in HTTP/1.1 allows requests to be issued asynchronously on a persistent connection. Requests must be processed in proper order. Can do clever packaging.

Outline

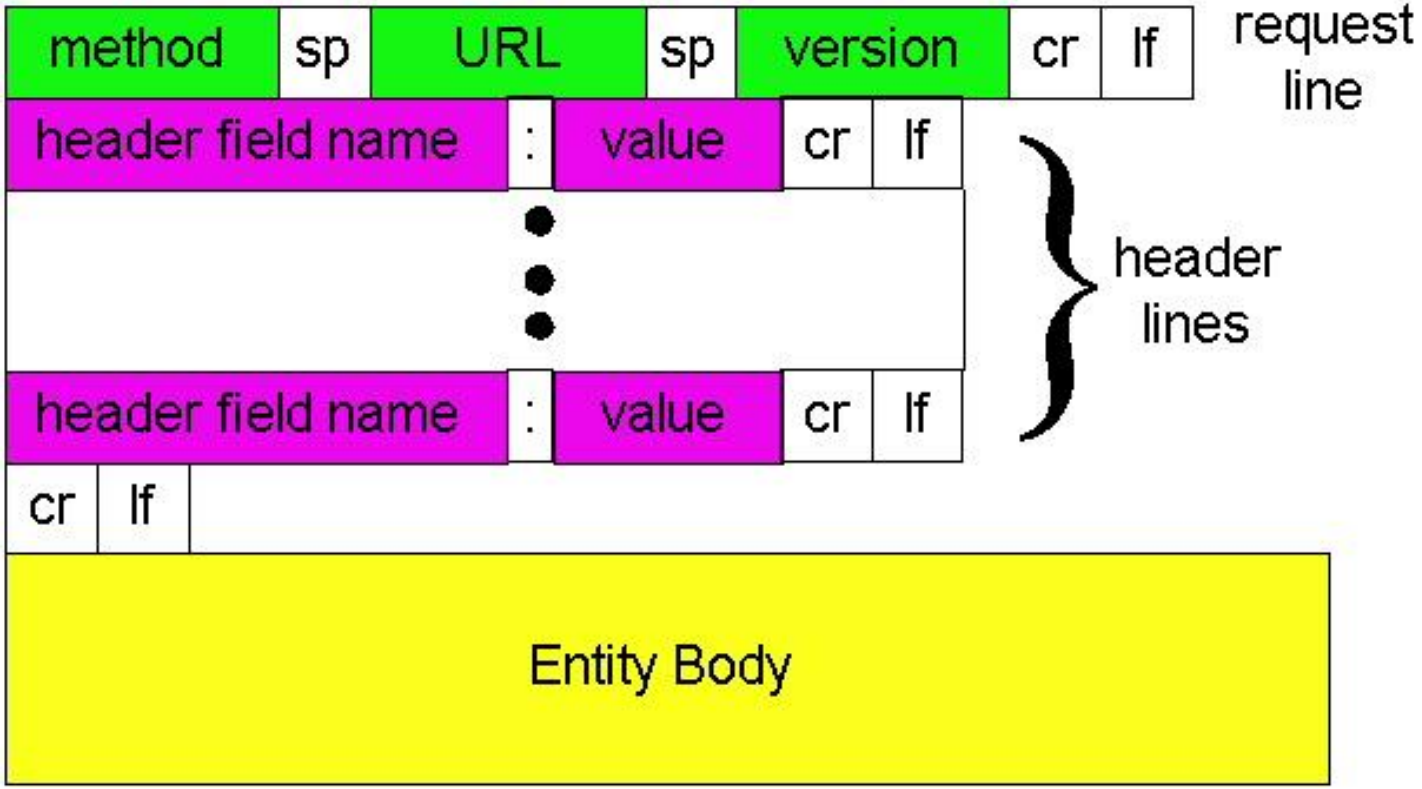
- Introduction to App Layer Protocols
- Brief History of WWW
- Architecture
- HTTP Connections
- HTTP Format
- Web Performance
- Cookies

HTTP request message

- HTTP request message:
  - ASCII (human-readable format)



HTTP request message: general format



HTTP Methods

- GET: retrieve a file (95% of requests)
- HEAD: just get meta-data (e.g., mod time)
- POST: submitting a form to a server
- PUT: store enclosed document as URI
- DELETE: removed named resource
- LINK/UNLINK: in 1.0, gone in 1.1
- TRACE: http "echo" for debugging (added in 1.1)
- CONNECT: used by proxies for tunneling (1.1)
- OPTIONS: request for server/proxy options (1.1)

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet www.eurecom.fr 80
```

Opens TCP connection to port 80 (default HTTP server port) at www.eurecom.fr. Anything typed in sent to port 80 at www.eurecom.fr

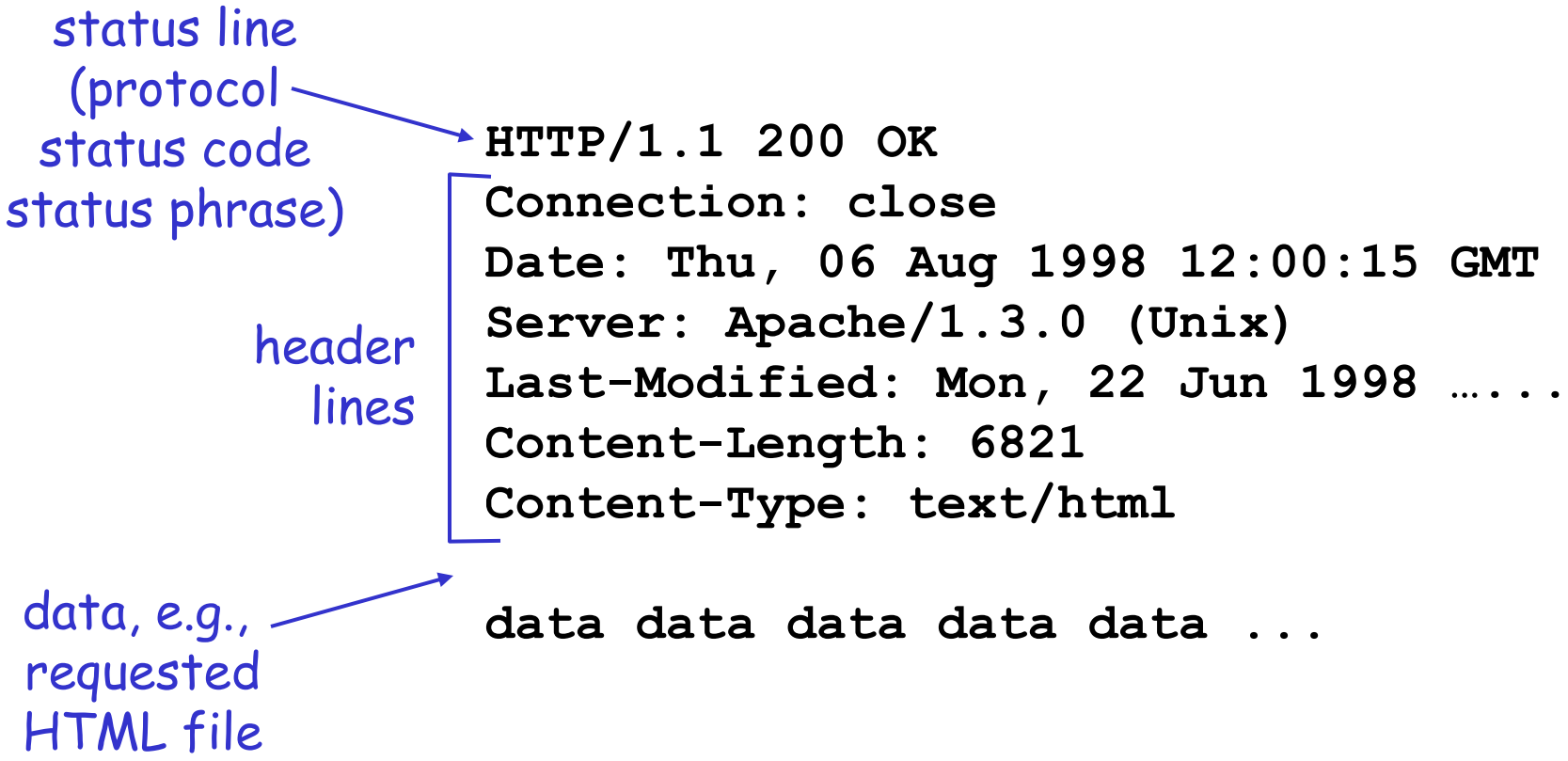
2. Type in a GET HTTP request:

```
GET /~ross/index.html HTTP/1.0
```

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!

HTTP response message



HTTP Response Status Codes

- 1XX: Informational (def'd in 1.0, used in 1.1)

  100 Continue, 101 Switching Protocols

- 2XX: Success

  200 OK, 206 Partial Content

- 3XX: Redirection

  301 Moved Permanently, 304 Not Modified

- 4XX: Client error

  400 Bad Request, 403 Forbidden, 404 Not Found

- 5XX: Server error

  500 Internal Server Error, 503 Service Unavailable, 505 HTTP Version Not Supported

HTTP Response Status Codes

- 1XX: Informational (def'd in 1.0, used in 1.1)

  100 Continue, 101 Switching Protocols

- 2XX: Success

  200 OK, 206 Partial Content

- 3XX: Redirection

  301 Moved Permanently, 304 Not Modified

- 4XX: Client error

  400 Bad Request, 403 Forbidden, 404 Not Found

- 5XX: Server error

  500 Internal Server Error, 503 Service Unavailable, 505 HTTP Version Not Supported

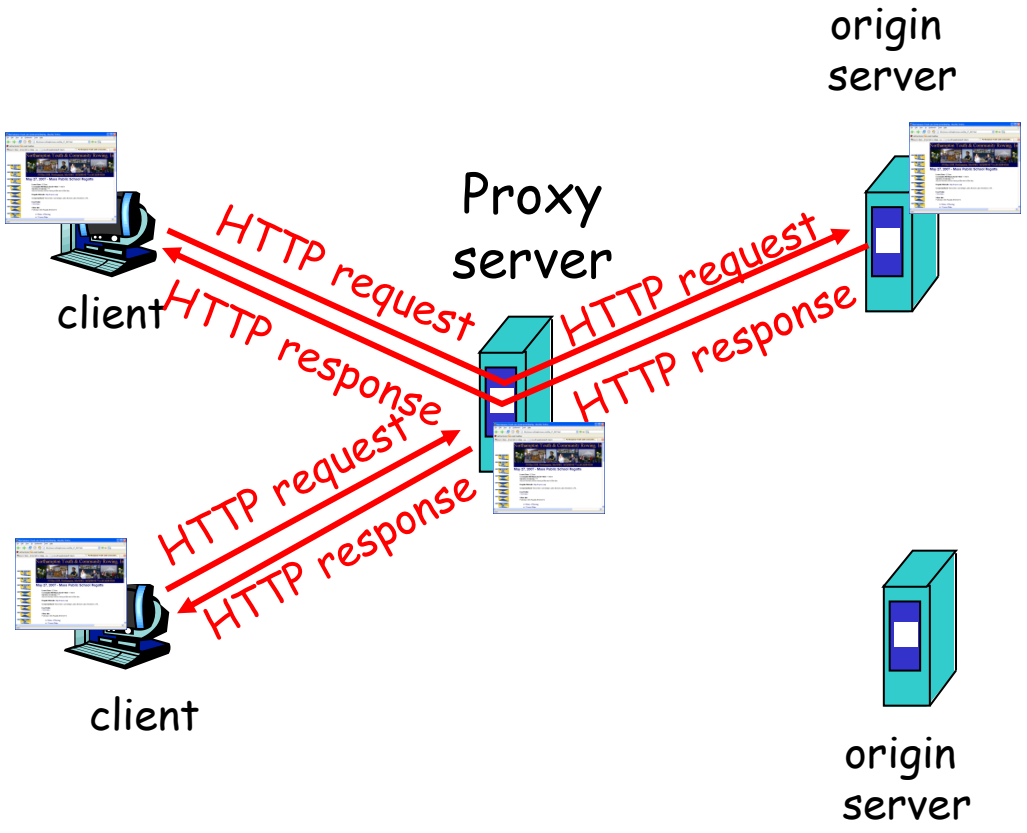
Outline

- Introduction to App Layer Protocols
- Brief History of WWW
- Architecture
- HTTP Connections
- HTTP Format
- Web Performance
- Cookies

Web caches (proxy server)

Goal: satisfy client request without involving origin server

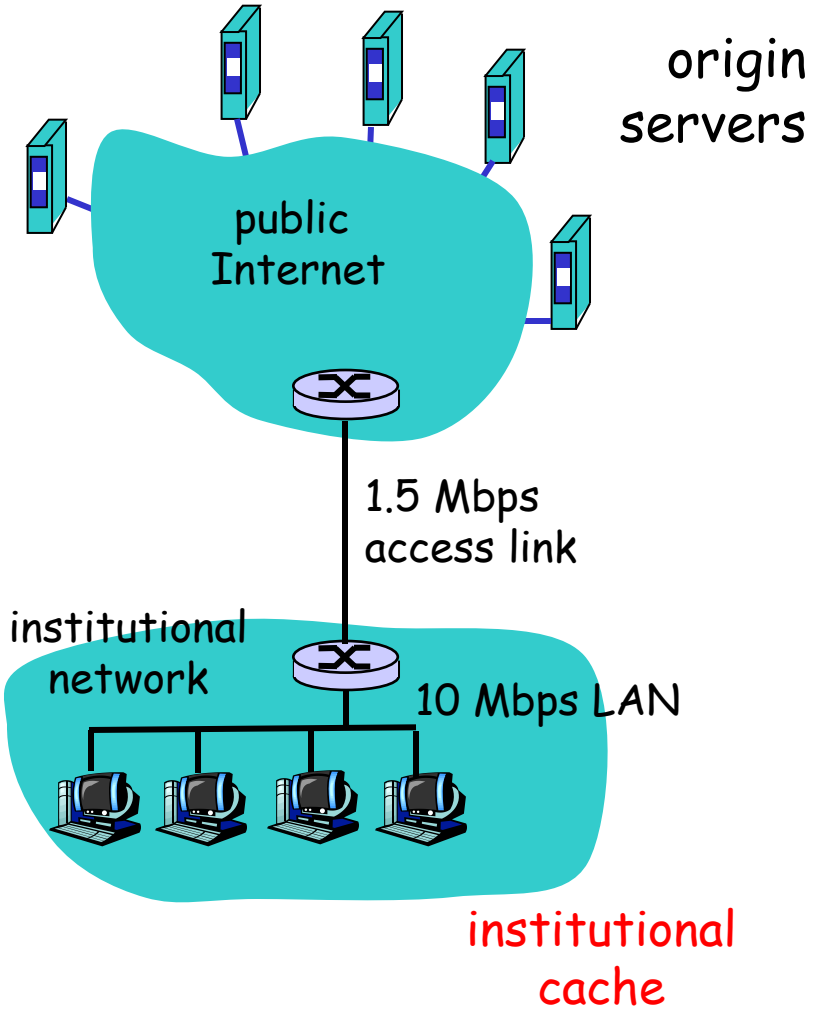
- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



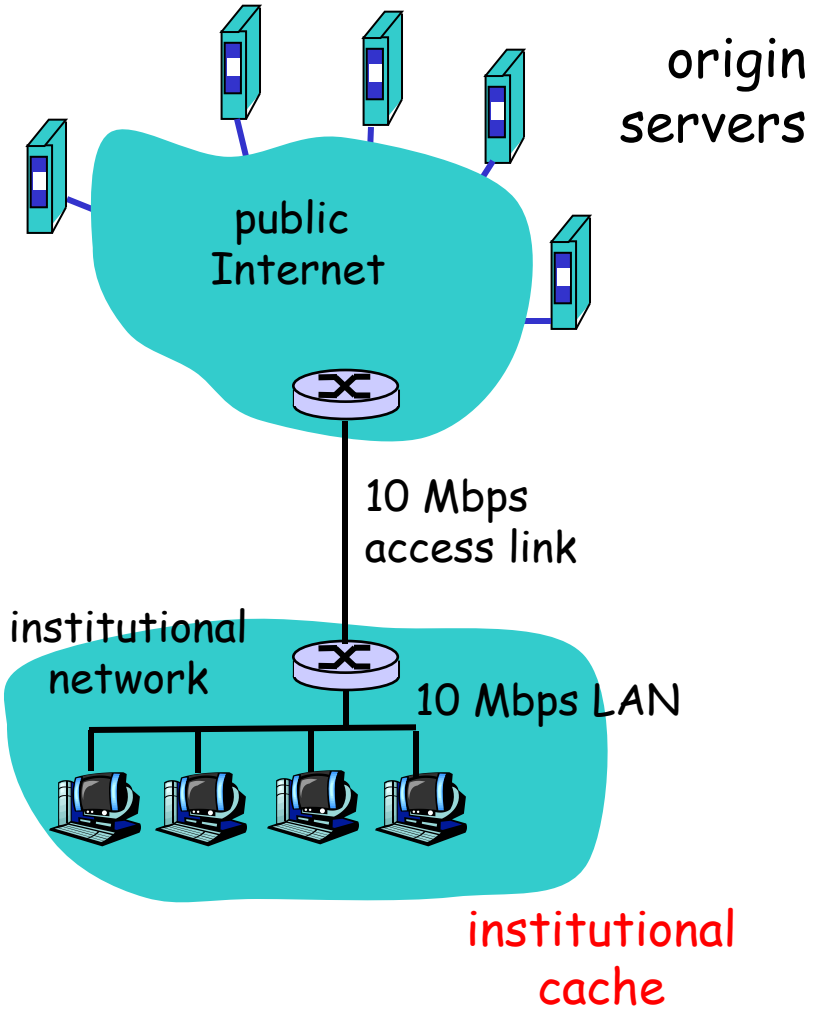
More about Web caching

- cache acts as both client and server
- typically cache is installed by ISP (university, company, residential ISP)

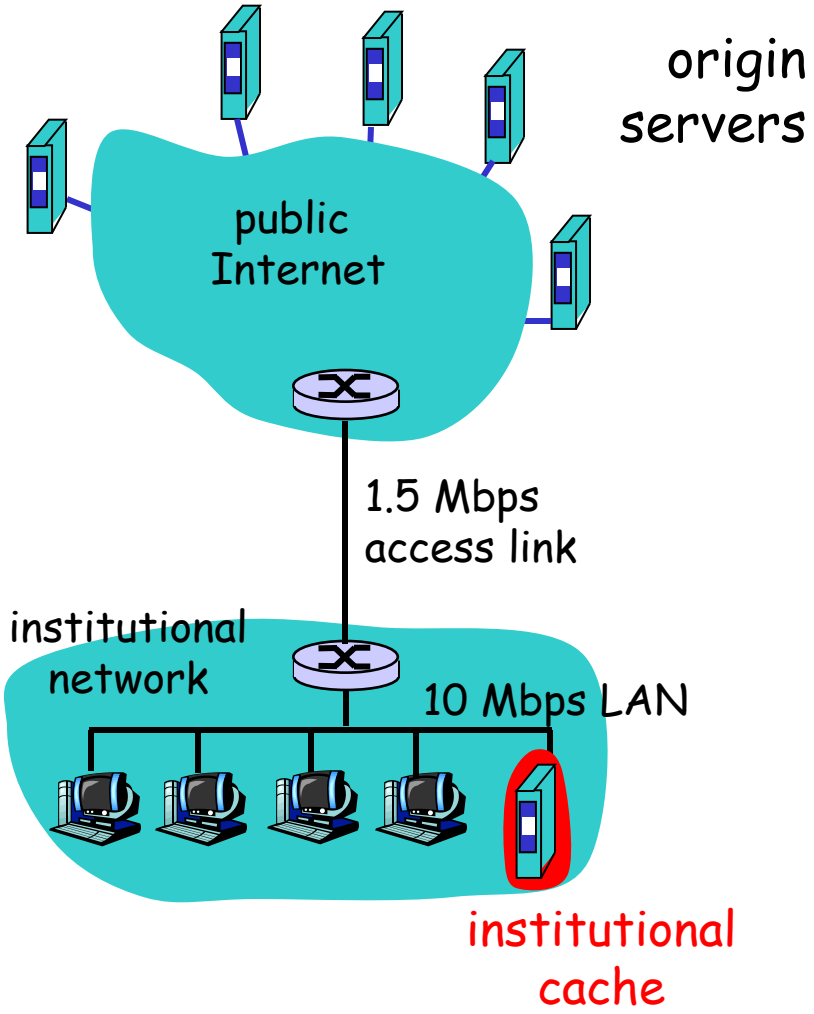
Caching example



Caching example (cont)



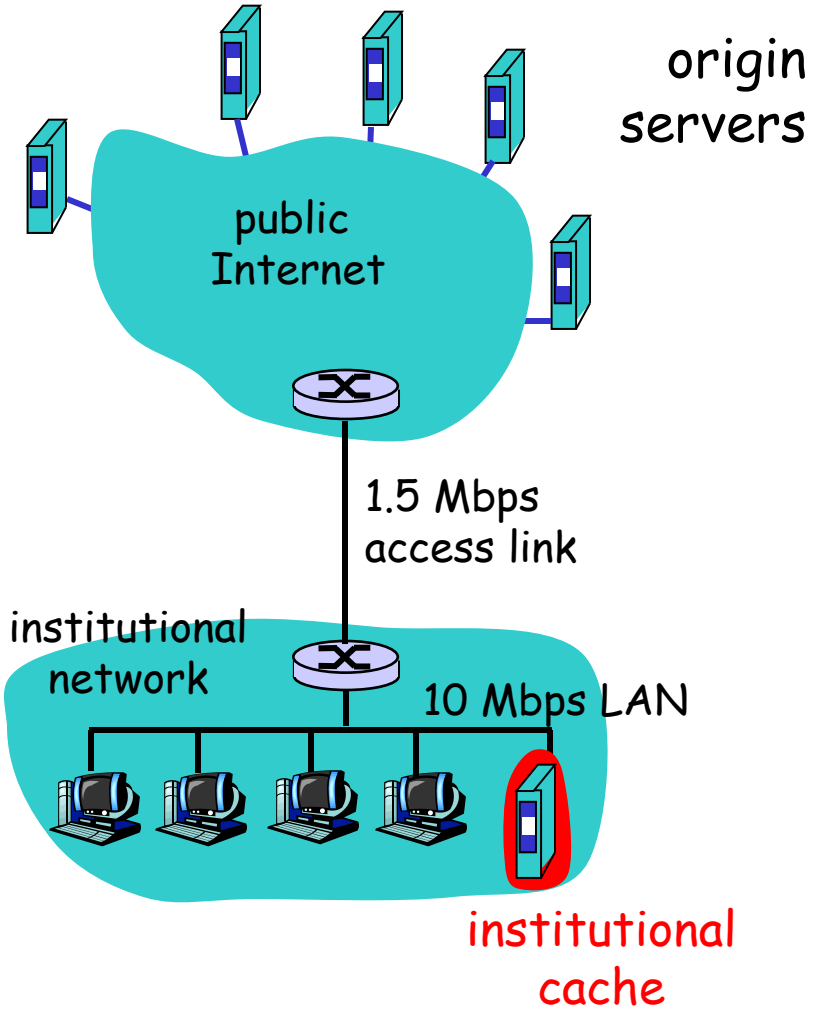
Caching example (cont)



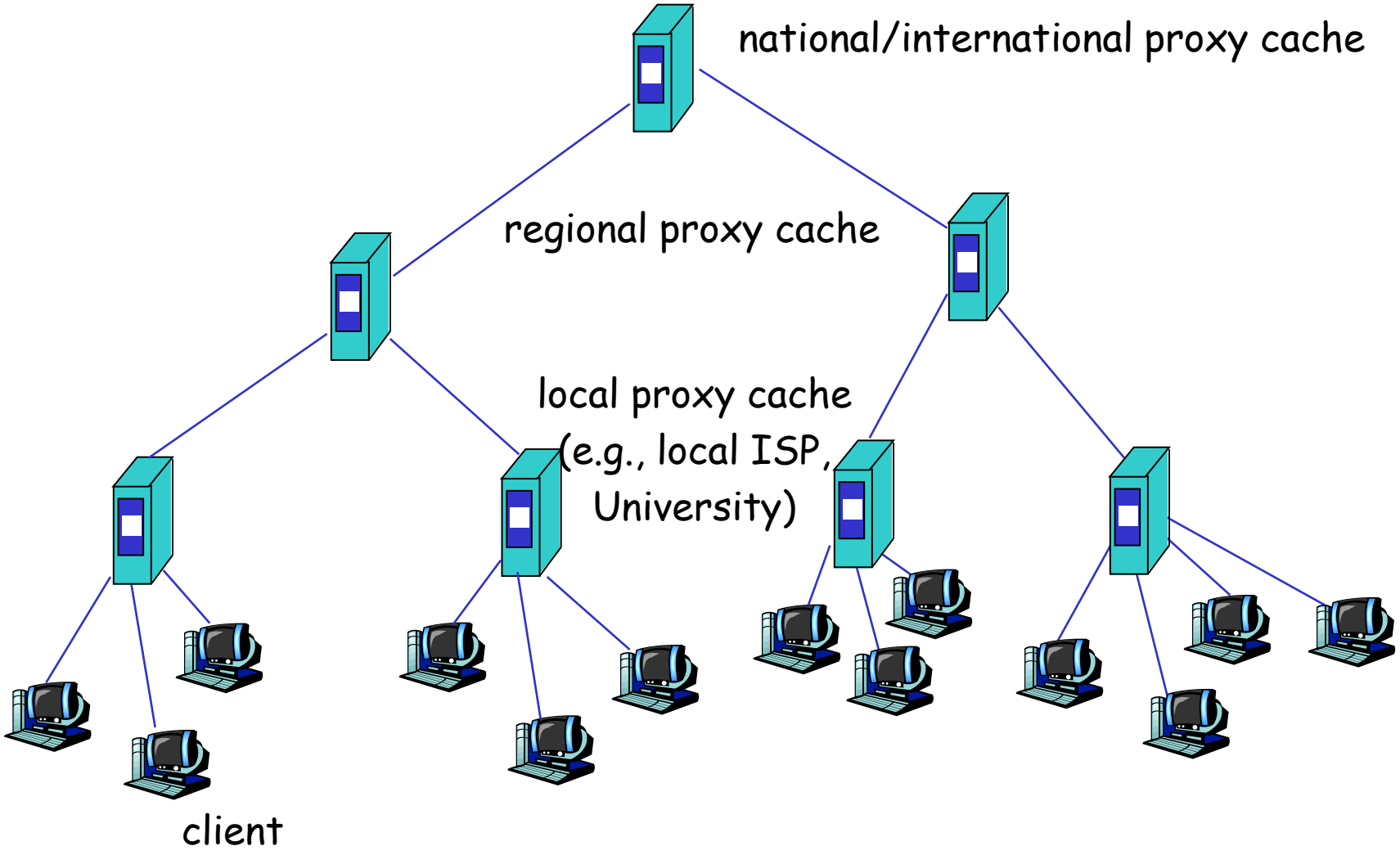
Caching example (cont)

why Web caching?

- Reduce response time for client request
- Reduce traffic on an institution's access link.
- Offloads server
  - Internet dense with caches: enables "poor" content providers to effectively deliver content (but so does P2P file sharing)



Web Caching Hierarchy



Some Issues

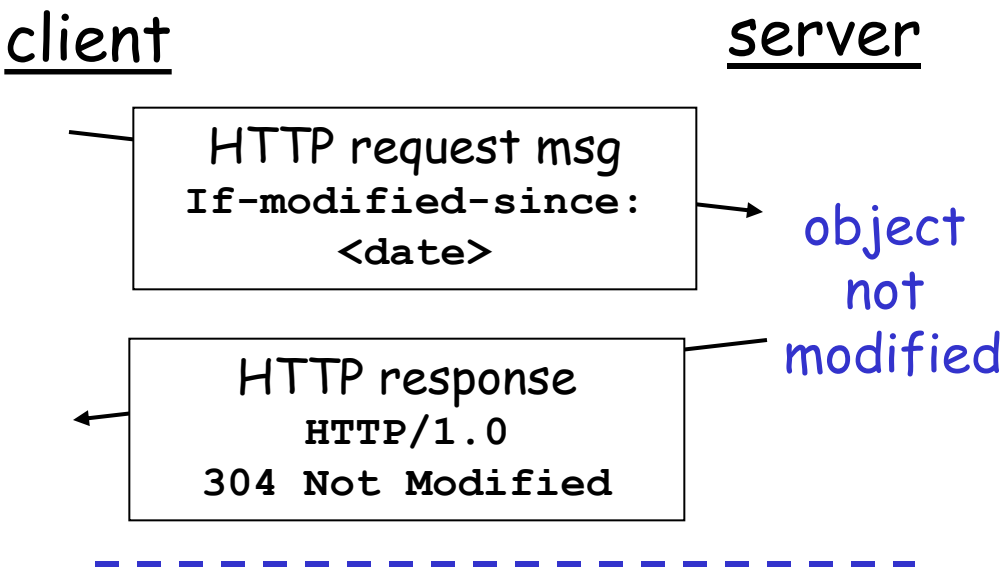
- Not all objects can be cached
  - E.g., dynamic objects, copyrighted material
- Cache consistency
  - strong
  - weak
- Cache Replacement Policies
  - Variable size objects
  - Varying cost of not finding an object (a "miss") in the cache
- Prefetch?
  - A large fraction of the requests are one-timers

Weak Consistency

- Each cached copy has a TTL beyond which it must be validated with the origin server
- TTL = freshness life time – age
  - freshness life time: often heuristically calculated; sometimes based on MAX_AGE or EXPIRES headers
  - age = current time (at client) – timestamp on object (time at which server generated response)
- Age Penalty?

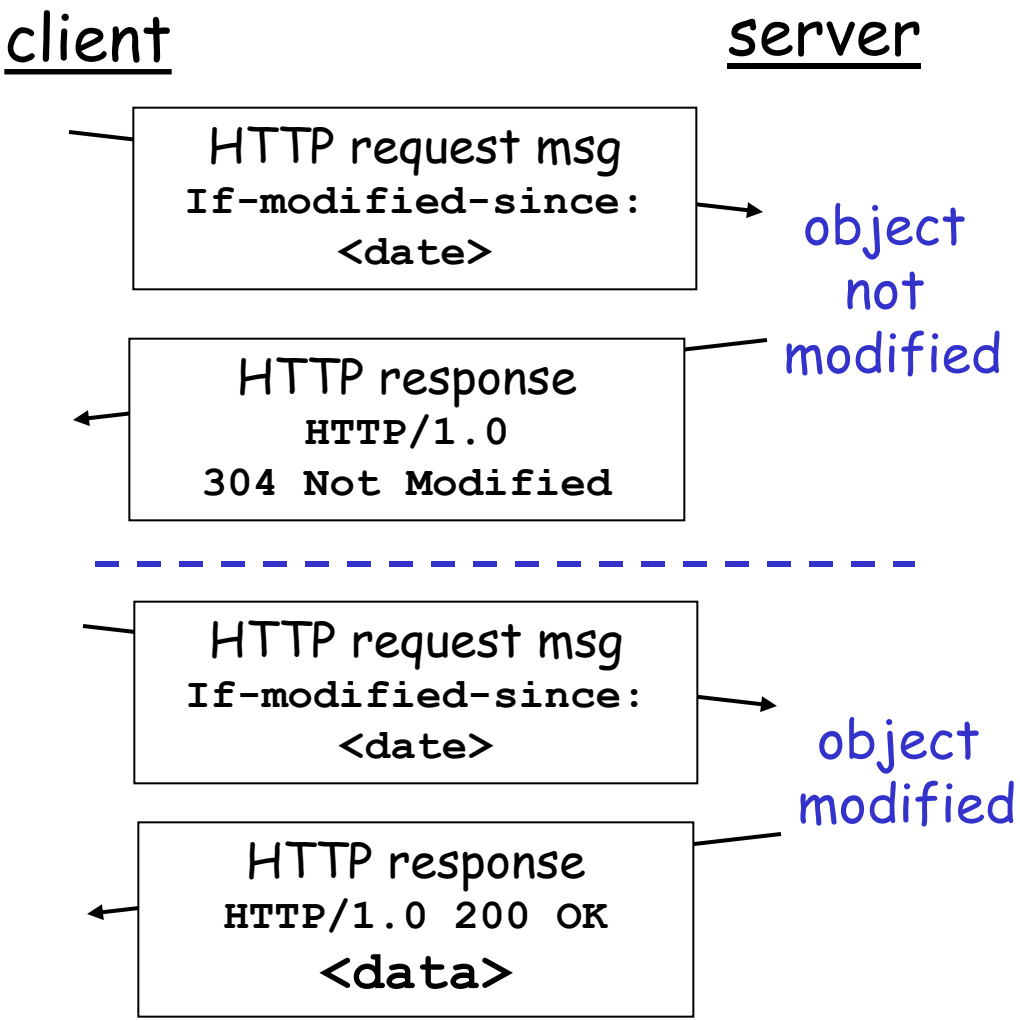
Conditional GET: client-side caching

- Goal: don't send object if client has up-to-date cached version
- client: specify date of cached copy in HTTP request
- server: response contains no object if cached copy is up-to-date.



Conditional GET: client-side caching

- Goal: don't send object if client has up-to-date cached version
- client: specify date of cached copy in HTTP request
- server: response contains no object if cached copy is up-to-date.



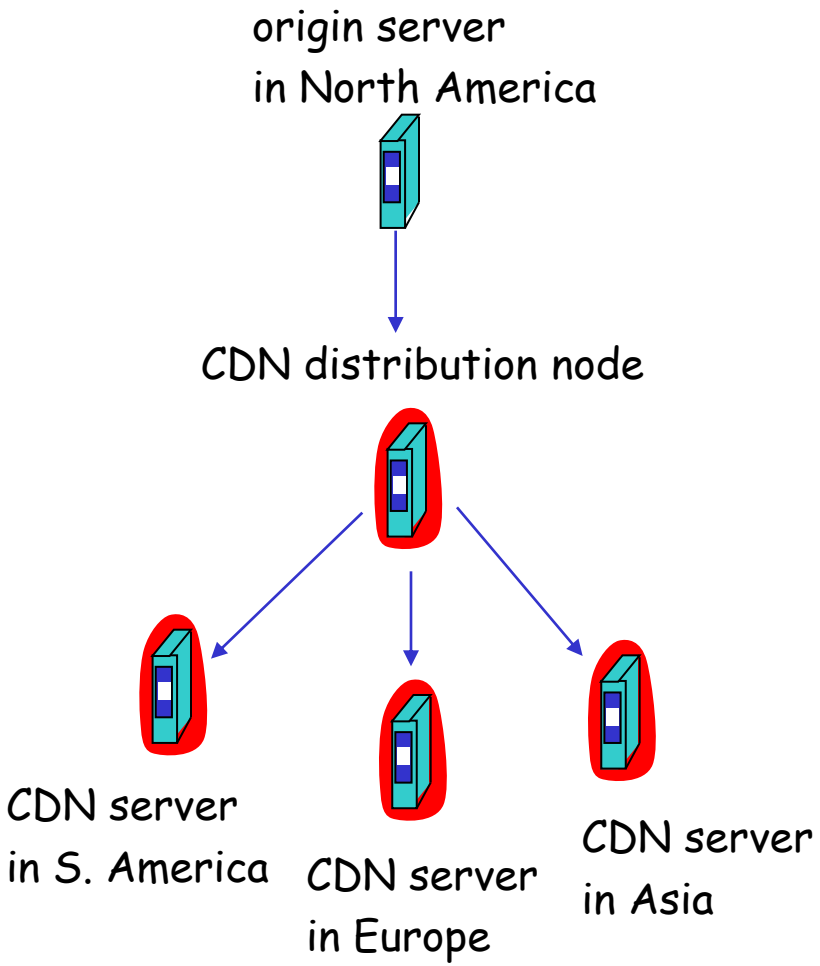
Content distribution networks (CDNs)

- The content providers are the CDN customers.

Content replication

- CDN company installs hundreds of CDN servers throughout Internet
  - in lower-tier ISPs, close to users
- CDN replicates its customers' content in CDN servers. When provider updates content, CDN updates servers

Different approaches …



Cookies: keeping "state"

Many major Web sites use cookies

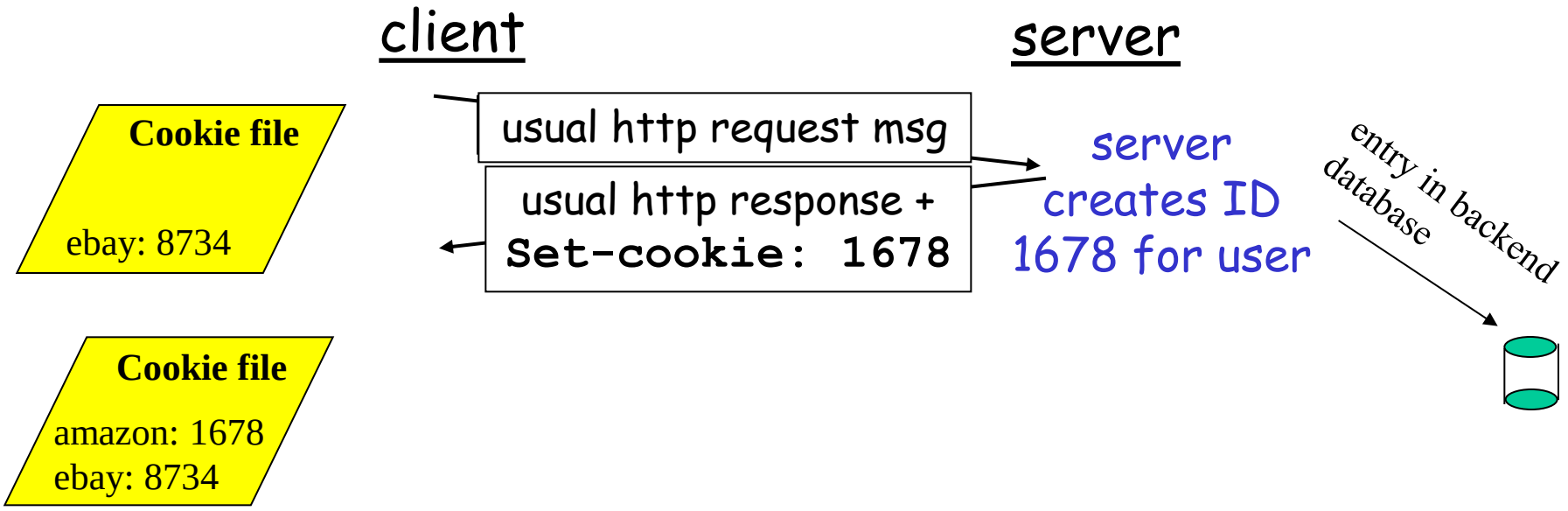
Four components:

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

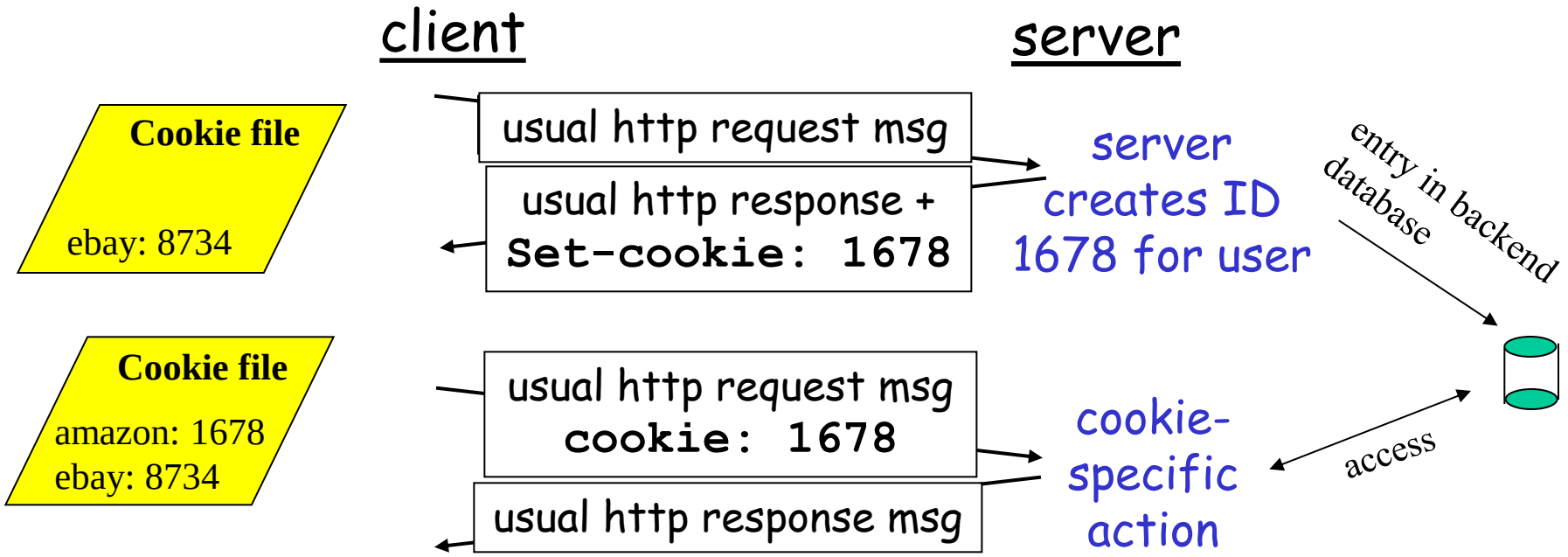
Example:

- User visits a specific e-commerce site …

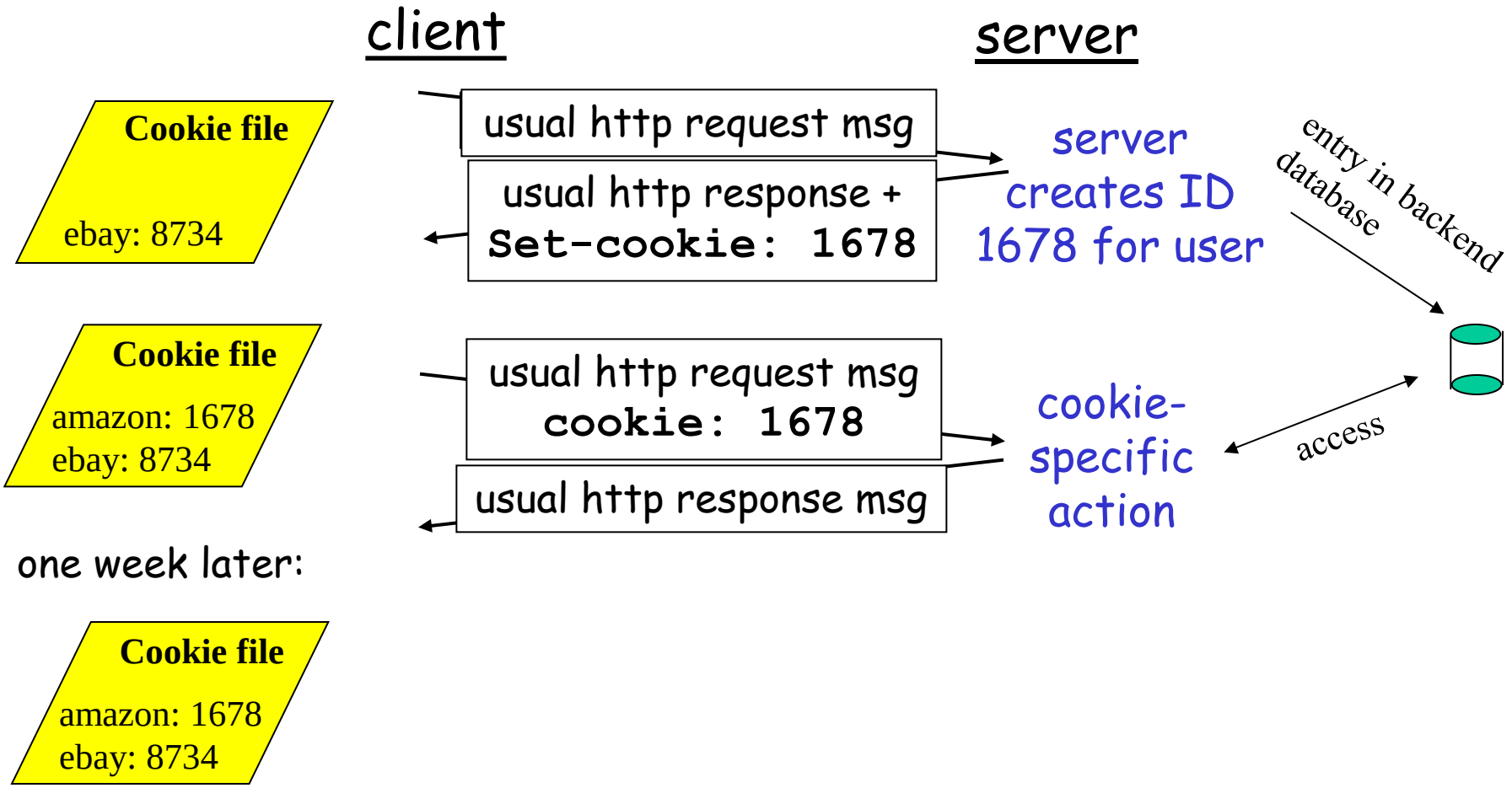
Cookies: keeping "state" (cont.)



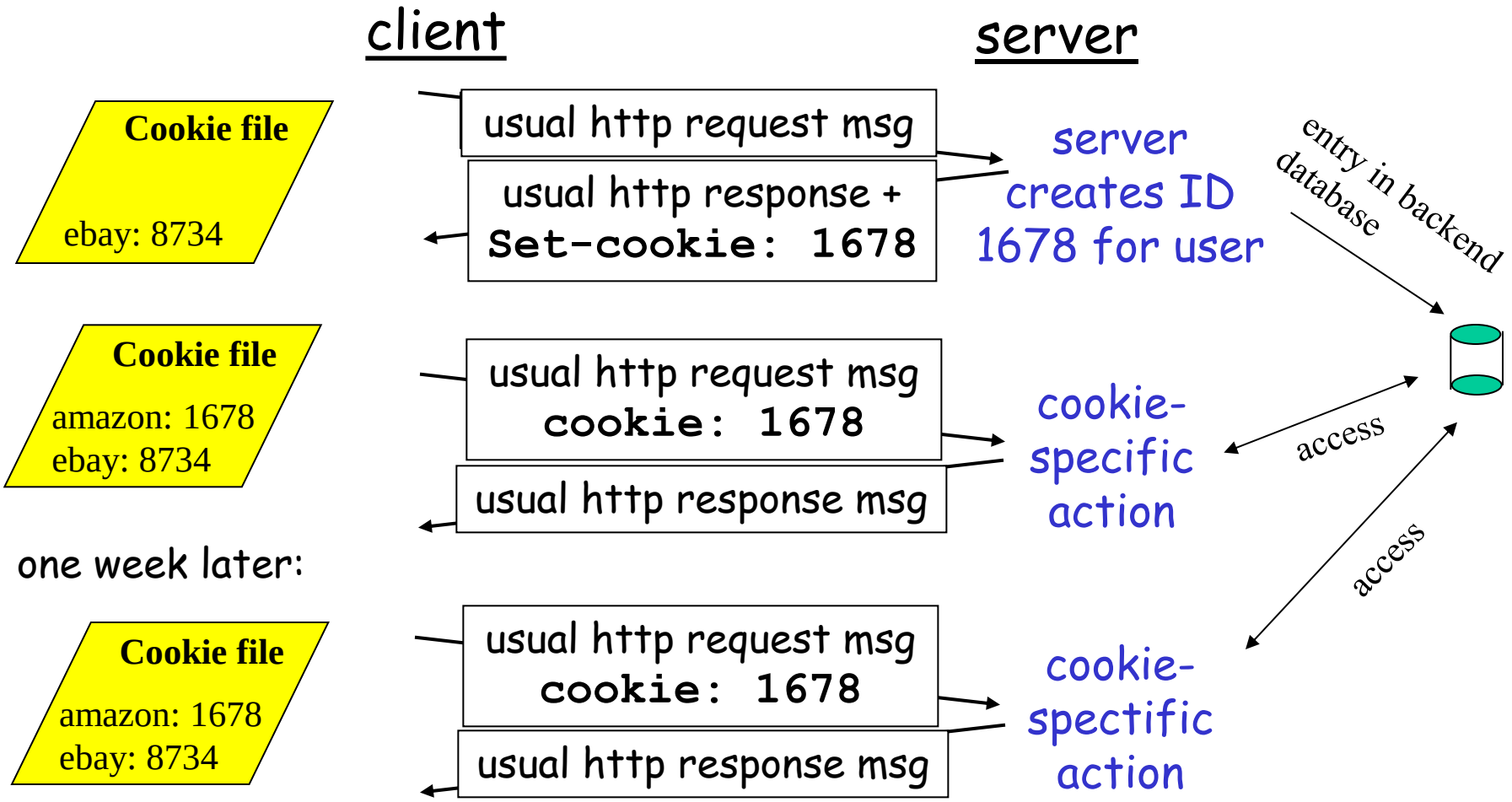
Cookies: keeping "state" (cont.)



Cookies: keeping "state" (cont.)



Cookies: keeping "state" (cont.)



Cookies (continued)

What cookies can bring:

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

aside

Cookies and privacy:

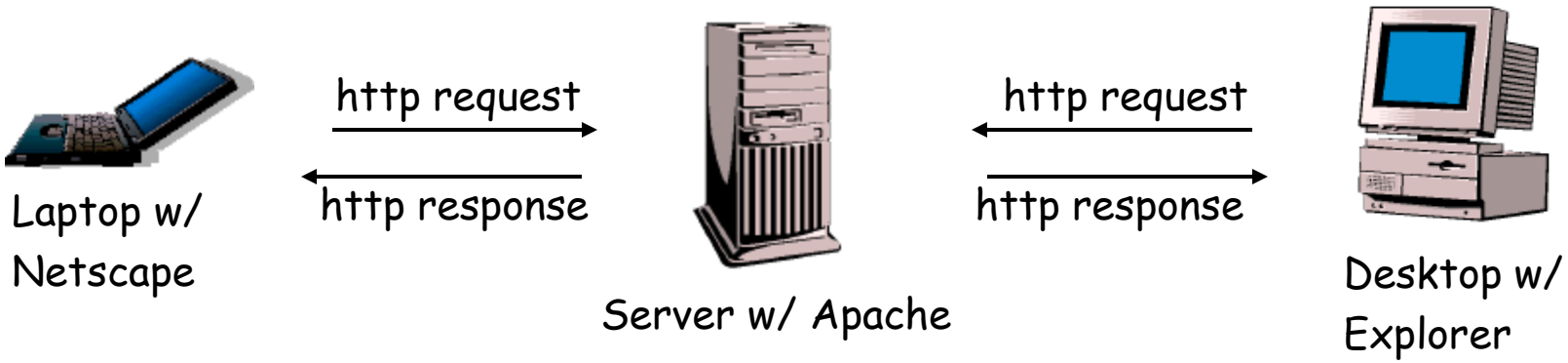
- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites
- search engines use redirection & cookies to learn yet more
- advertising companies obtain info across sites

Web & HTTP

- The major application on the Internet
  - A (very) large fraction of traffic is HTTP
- Client/server model:
  - Clients make requests, servers respond to them
  - Done mostly in ASCII text (helps debugging!)
- Various headers and commands
- Web Caching & Performance
- Content Distribution Networks

More slides …

Introduction to HTTP



- HTTP: HyperText Transfer Protocol
  - Communication protocol between clients and servers
  - Application layer protocol for WWW
- Client/Server model:
  - Client: browser that requests, receives, displays object
  - Server: receives requests and responds to them
- Protocol consists of various operations
  - Few for HTTP 1.0 (RFC 1945, 1996)
  - Many more in HTTP 1.1 (RFC 2616, 1999)

Request Generation

- User clicks on something
- Uniform Resource Locator (URL):
  - `http://www.cnn.com`
  - `http://www.cpsc.ucalgary.ca`
  - `https://www.paymybills.com`
  - `ftp://ftp.kernel.org`
- Different URL schemes map to different services
- Hostname is converted from a name to a 32-bit IP address (DNS lookup, if needed)
- Connection is established to server (TCP)

What Happens Next?

- Client downloads HTML document
  - Sometimes called "container page"
  - Typically in text format (ASCII)
  - Contains instructions for rendering (e.g., background color, frames)
  - Links to other pages

- Many have embedded objects:
  - Images: GIF, JPG (logos, banner ads)
  - Usually automatically retrieved
    - I.e., without user involvement
    - can control sometimes (e.g. browser options, junkbusters)

```
<html>
<head>
<meta
name="Author"
content="Erich Nahum">
<title> Linux Web
Server Performance
</title>
</head>
<body text="#00000">
<img width=31
height=11
src="ibmlogo.gif">
<img
src="images/new.gif>
<h1>Hi There!</h1>
Here's lots of cool
linux stuff!
<a href="more.html">
Click here</a>
for more!
</body>
</html>
```

Sample html file

Web Server Role

- Respond to client requests, typically a browser
  - Can be a proxy, which aggregates client requests
  - Could be search engine spider or robot
- May have work to do on client's behalf:
  - Is the client's cached copy still good?
  - Is client authorized to get this document?
- Hundreds or thousands of simultaneous clients
- Hard to predict how many will show up on some day (e.g., "flash crowds", diurnal cycle, global presence)
- Many requests are in progress concurrently

HTTP Request Format

```
GET /images/penguin.gif HTTP/1.0
User-Agent: Mozilla/0.9.4 (Linux 2.2.19)
Host: www.kernel.org
Accept: text/html, image/gif, image/jpeg
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: B=xh203jfsf; Y=3sdkfjej
<cr><lf>
```

- Messages are in ASCII (human-readable)
- Carriage-return and line-feed indicate end of headers
- Headers may communicate private information
  (browser, OS, cookie information, etc.)

Request Types

Called Methods:

- GET: retrieve a file (95% of requests)
- HEAD: just get meta-data (e.g., mod time)
- POST: submitting a form to a server
- PUT: store enclosed document as URI
- DELETE: removed named resource
- LINK/UNLINK: in 1.0, gone in 1.1
- TRACE: http "echo" for debugging (added in 1.1)
- CONNECT: used by proxies for tunneling (1.1)
- OPTIONS: request for server/proxy options (1.1)

Response Format

```
HTTP/1.0 200 OK
Server: Tux 2.0
Content-Type: image/gif
Content-Length: 43
Last-Modified: Fri, 15 Apr 1994 02:36:21 GMT
Expires: Wed, 20 Feb 2002 18:54:46 GMT
Date: Mon, 12 Nov 2001 14:29:48 GMT
Cache-Control: no-cache
Pragma: no-cache
Connection: close
Set-Cookie: PA=wefj2we0-jfjf
<cr><lf>
<data follows…>
```

- Similar format to requests (i.e., ASCII)

Response Types

- 1XX: Informational (def'd in 1.0, used in 1.1)

  `100 Continue, 101 Switching Protocols`

- 2XX: Success

  `200 OK, 206 Partial Content`

- 3XX: Redirection

  `301 Moved Permanently, 304 Not Modified`

- 4XX: Client error

  `400 Bad Request, 403 Forbidden, 404 Not Found`

- 5XX: Server error

  `500 Internal Server Error, 503 Service Unavailable, 505 HTTP Version Not Supported`

Outline of an HTTP Transaction

- This section describes the basics of servicing an HTTP GET request from user space
- Assume a single process running in user space, similar to Apache 1.3
- We'll mention relevant socket operations along the way

```
initialize;
forever do {
  get request;
  process;
  send response;
  log request;
}
```

server in a nutshell

Readying a Server

```
s = socket(); /* allocate listen socket */
bind(s, 80);  /* bind to TCP port 80    */
listen(s);    /* indicate willingness to accept */
while (1) {
    newconn = accept(s); /* accept new connection */b
```

- First thing a server does is notify the OS it is interested in WWW server requests; these are typically on TCP port 80. Other services use different ports (e.g., SSL is on 443)
- Allocate a socket and bind()'s it to the address (port 80)
- Server calls listen() on the socket to indicate willingness to receive requests
- Calls accept() to wait for a request to come in (and blocks)
- When the accept() returns, we have a new socket which represents a new connection to a client

Processing a Request

```
remoteIP = getsockname(newconn);
remoteHost = gethostbyname(remoteIP);
gettimeofday(currentTime);
read(newconn, reqBuffer, sizeof(reqBuffer));
reqInfo = serverParse(reqBuffer);
```

- getsockname() called to get the remote host name
  - for logging purposes (optional, but done by most)
- gethostbyname() called to get name of other end
  - again for logging purposes
- gettimeofday() is called to get time of request
  - both for Date header and for logging
- read() is called on new socket to retrieve request
- request is determined by parsing the data
  - "GET /images/jul4/flag.gif"

Processing a Request (cont)

```
fileName = parseOutFileName(requestBuffer);
fileAttr = stat(fileName);
serverCheckFileStuff(fileName, fileAttr);
open(fileName);
```

- stat() called to test file path
  - to see if file exists/is accessible
  - may not be there, may only be available to certain people
  - "/microsoft/top-secret/plans-for-world-domination.html"
- stat() also used for file meta-data
  - e.g., size of file, last modified time
  - "Has file changed since last time I checked?"
- might have to stat() multiple files and directories
- assuming all is OK, open() called to open the file

Responding to a Request

```
read(fileName, fileBuffer);
headerBuffer = serverFigureHeaders(fileName, reqInfo);
write(newSock, headerBuffer);
write(newSock, fileBuffer);
close(newSock);
close(fileName);
write(logFile, requestInfo);
```

- read() called to read the file into user space
- write() is called to send HTTP headers on socket
  (early servers called write() for *each header!)*
- write() is called to write the file on the socket
- close() is called to close the socket
- close() is called to close the open file descriptor
- write() is called on the log file

Summary of Web and HTTP

- The major application on the Internet
  - Majority of traffic is HTTP (or HTTP-related)
- Client/server model:
  - Clients make requests, servers respond to them
  - Done mostly in ASCII text (helps debugging!)
- Various headers and commands
  - Too many to go into detail here
  - Many web books/tutorials exist (e.g., Krishnamurthy & Rexford 2001)