
Network Applications:
HTTP/1.0/1.1/2;
High-Performance Server Design (Per Thread)

Qiao Xiang

<https://qiaoxiang.me/courses/cnns-xmuf21/index.shtml>

10/19/2021

Outline

- Admin and recap
- HTTP/1.0
 - Basic design
 - Dynamic content
- HTTP “acceleration”
- Network server design

Admin

- HTTP server assignment (part 1) to be posted later today

- Plagiarism is NOT allowed in any assignment
 - Discussion and reading are encouraged, but copying is NOT!
 - If you reuse others' (e.g., classmates, friends, Stackoverflow, CSDN) code as part of your solution, please mark it in comments.

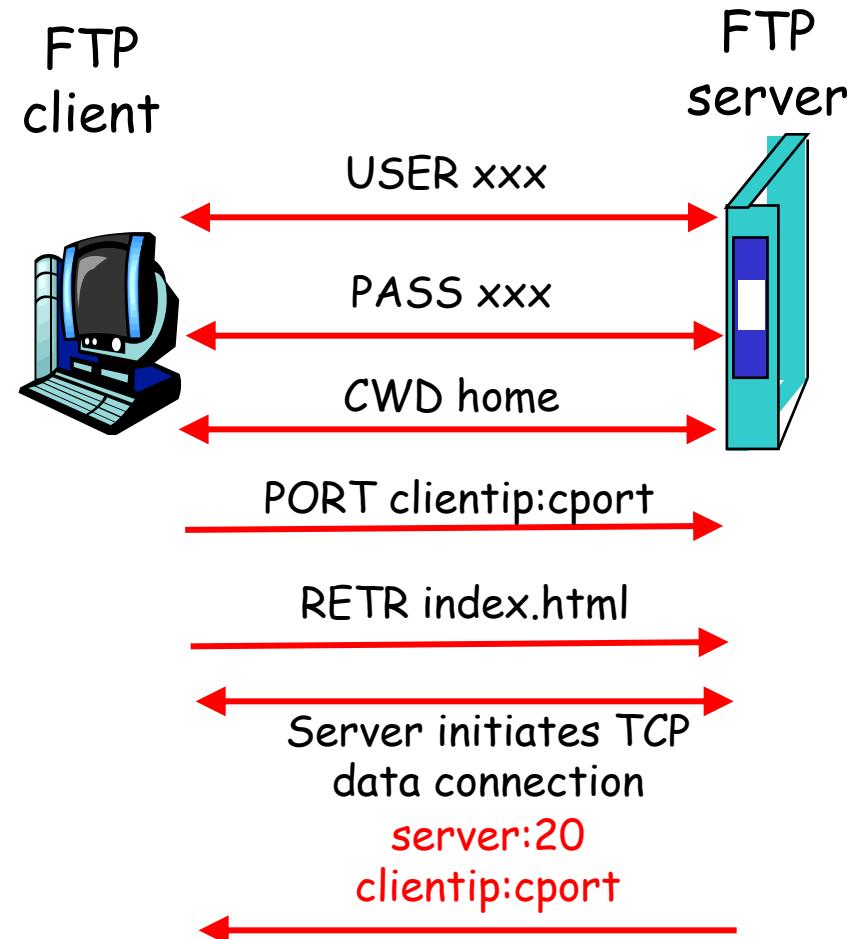
Recap: FTP

□ A stateful protocol

- state established by commands such as
 - USER/PASS, CWD, TYPE

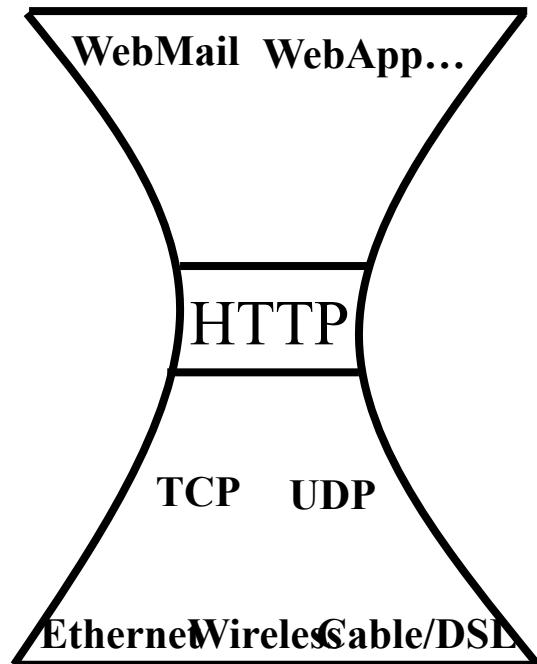
□ Multiple TCP connections

- A control connection
- Data connections
 - Two approaches: PORT vs PASV
 - GridFTP: concurrent data connections; block data transfer mode



Recap: HTTP

- Wide use of HTTP for Web applications
- Example: RESTful API
 - RESTful design
 - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
 - <http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html>



Recap: HTTP

❑ C-S app serving Web pages

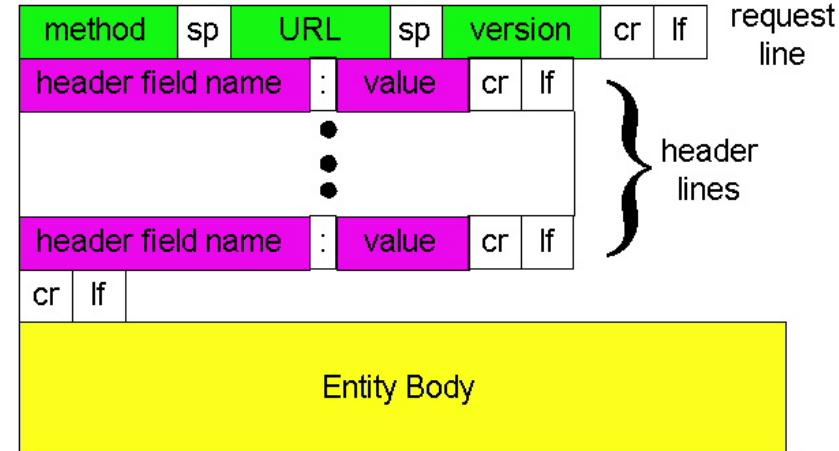
- message format

- request/response line, header lines, entity body

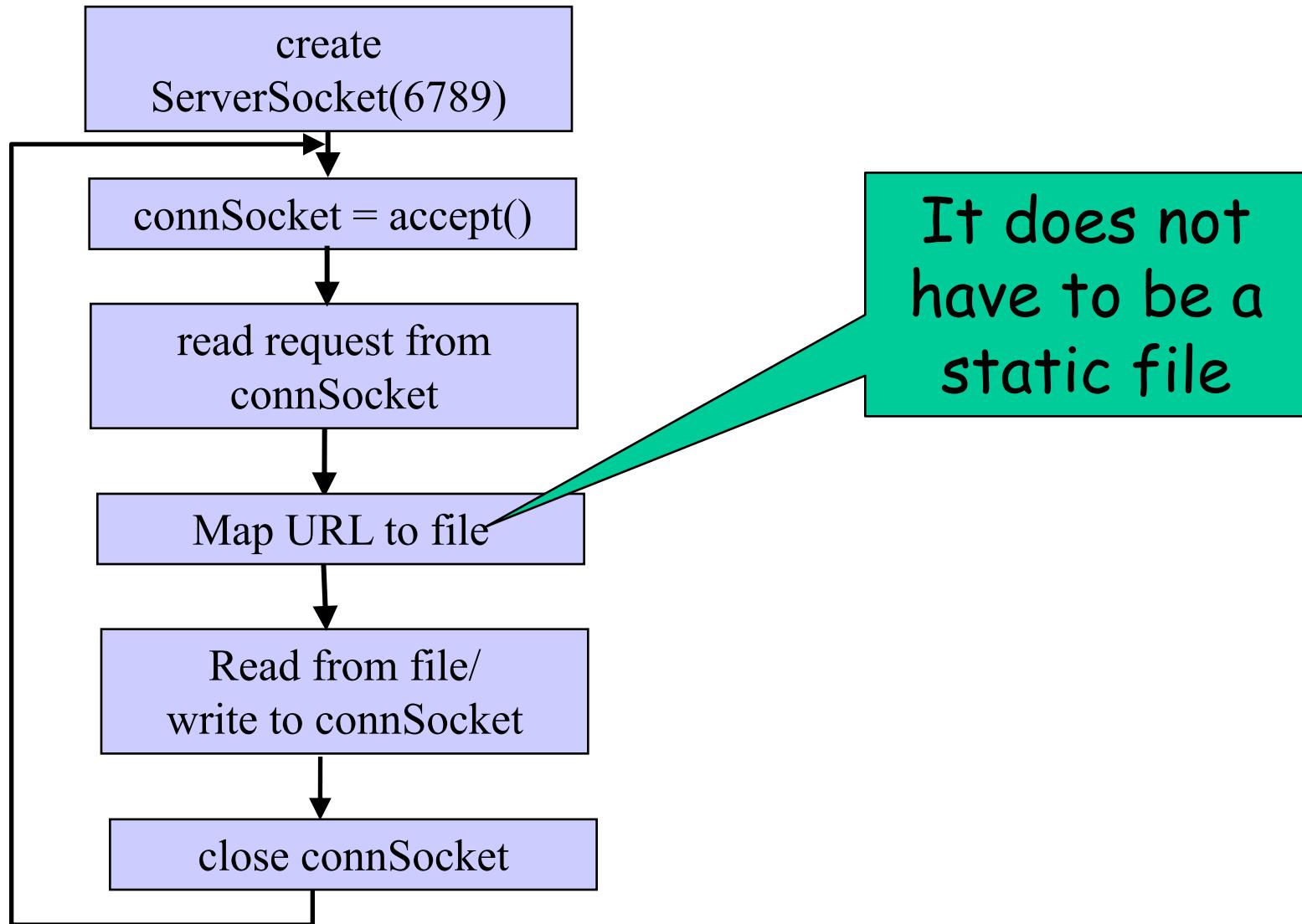
- simple methods, rich headers

- message flow

- stateless server, thus states such as cookie and authentication are needed in each message



Recap: Basic HTTP/1.0 Server



Outline

- Admin and recap
- HTTP/1.0
 - Basic design
 - *Dynamic content*

Dynamic Content Pages

- There are multiple approaches to make dynamic web pages:
 - Embed code into pages (**server side include**)
 - http server includes an interpreter for the type of pages
 - Invoke external programs (http server is agnostic to the external program execution)
 - E.g., **Common Gateway Interface (CGI)**

`http://www.cs.yale.edu/index.shtml`

`http://www.cs.yale.edu/cgi-bin/ureserve.pl`

`http://www.google.com/search?q=Yale&sourceid=chrome`

Example SSI

- See programming/examples-java-socket/BasicWebServer/ssi/index.shtml, header.shtml, ...

Example SSI

- See programming/examples-java-socket/BasicWebServer/ssi/index.shtml, header.shtml, ...

- To enable ssi, need configuration to tell the web server (see conf/apache-htaccess)
 - <https://httpd.apache.org/docs/2.2/howto/htaccess.html> (Server Side Includes example)

CGI: Invoking External Programs

□ Two issues

- Input: Pass HTTP request parameters to the external program
- Output: Redirect external program output to socket

Example: Typical CGI Implementation

- Starts the executable as a child process
 - Passes HTTP request as environment variables
 - <http://httpd.apache.org/docs/2.2/env.html>
 - CGI standard: <http://www.ietf.org/rfc/rfc3875>
 - Redirects input/output of the child process to the socket

Example: CGI

□ Example:

- GET /search?q=Yale&sourceid=chrome HTTP/1.0
- **setup environment variables, in particular**
`$QUERY_STRING=q=Yale&sourceid=chrome`
- **start search and redirect its input/output**

<https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html>

Example

- <http://172.28.229.215/BasicWebServer/cgi/price.cgi?appl>

```
#!/usr/bin/perl -w

$company = $ENV{'QUERY_STRING'};
print "Content-Type: text/html\r\n";
print "\r\n";

print "<html>";
print "<h1>Hello! The price is ";

if ($company =~ /appl/) {
    my $var_rand = rand();
    print 450 + 10 * $var_rand;
} else {
    print "150";
}

print "</h1>";
print "</html>";
```

Client Using Dynamic Pages

- See ajax.html and wireshark for client code example

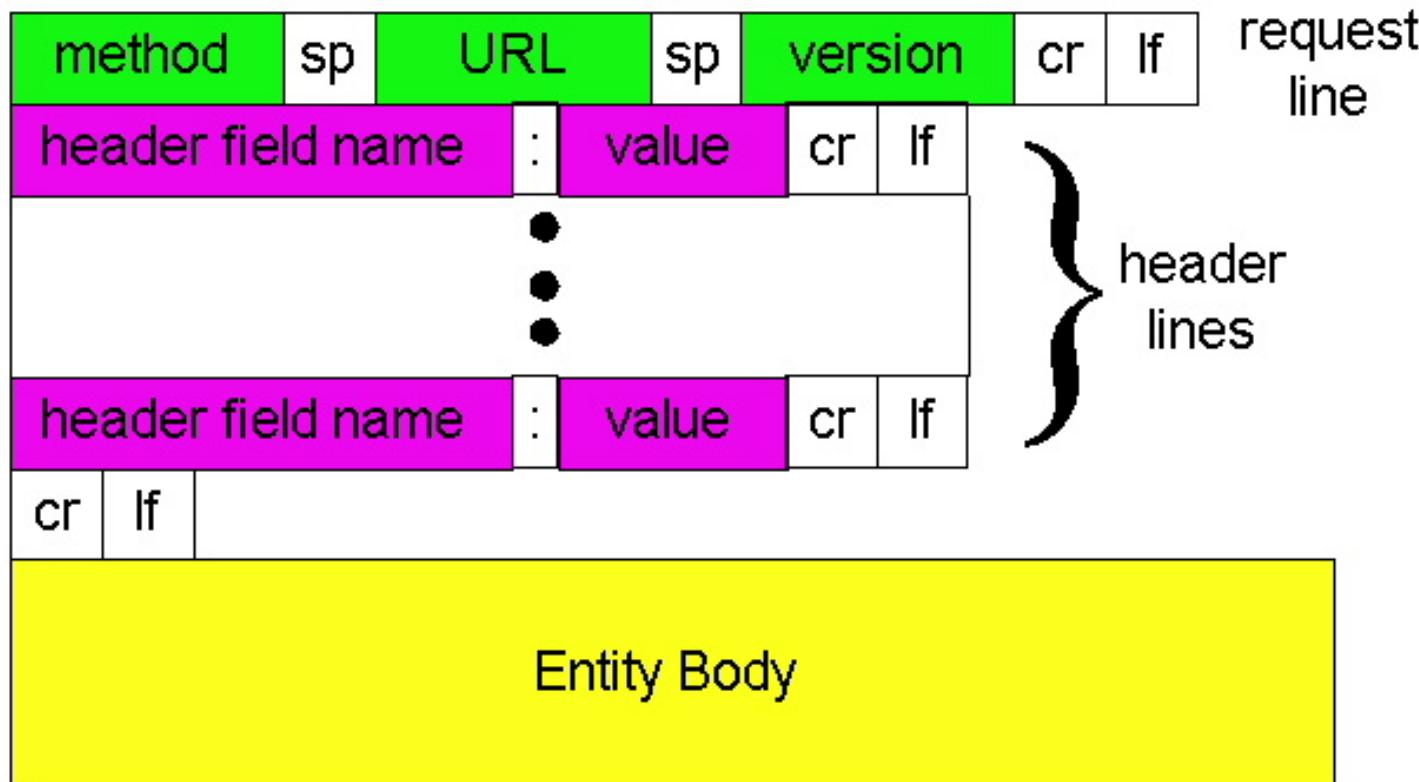
<http://172.28.229.215/BasicWebServer/cgi/ajax.html>

Discussions

- What features are missing in HTTP that we have covered so far?

HTTP: POST

- If an HTML page contains forms or parameter too large, they are sent using POST and encoded in message body



HTTP: POST Example

POST /path/script.cgi HTTP/1.0

User-Agent: MyAgent

Content-Type: application/x-www-form-urlencoded

Content-Length: 15

item1=A&item2=B

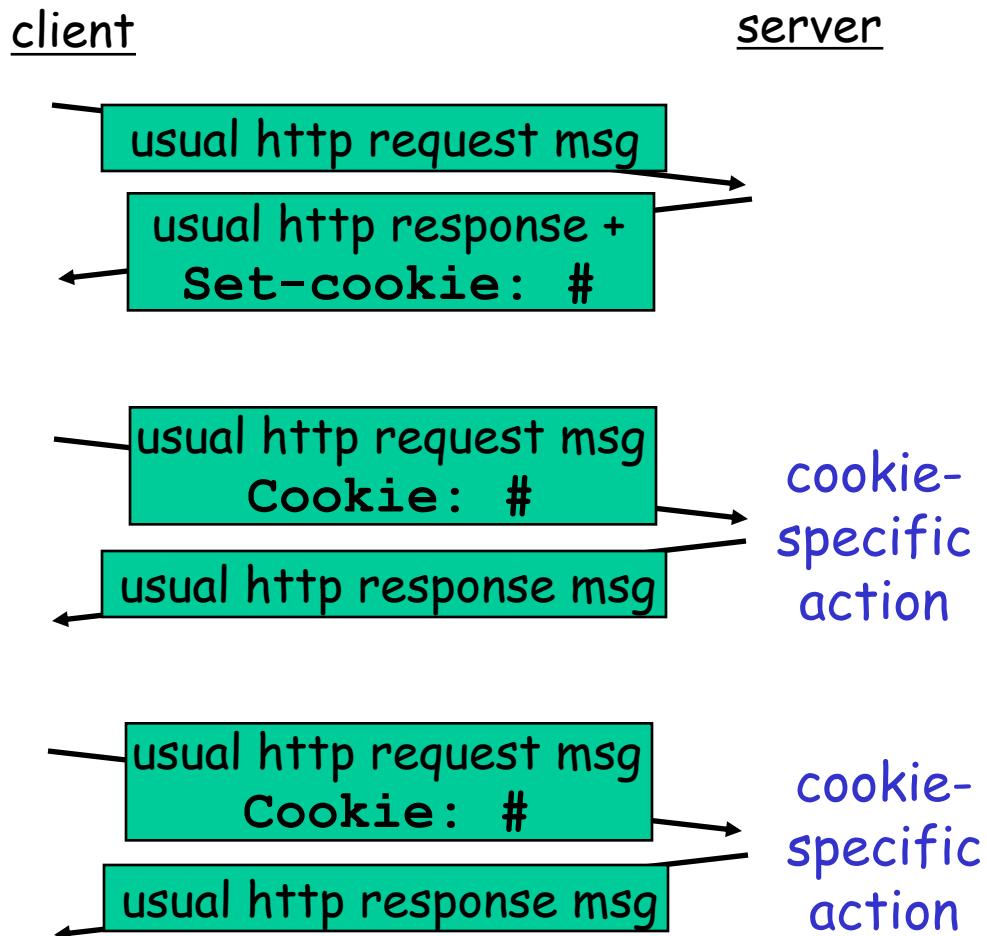
Example using nc:

<https://github.com/programming-examples/java-socket/tree/BasicWebServer/nc/>

Stateful User-server Interaction: Cookies

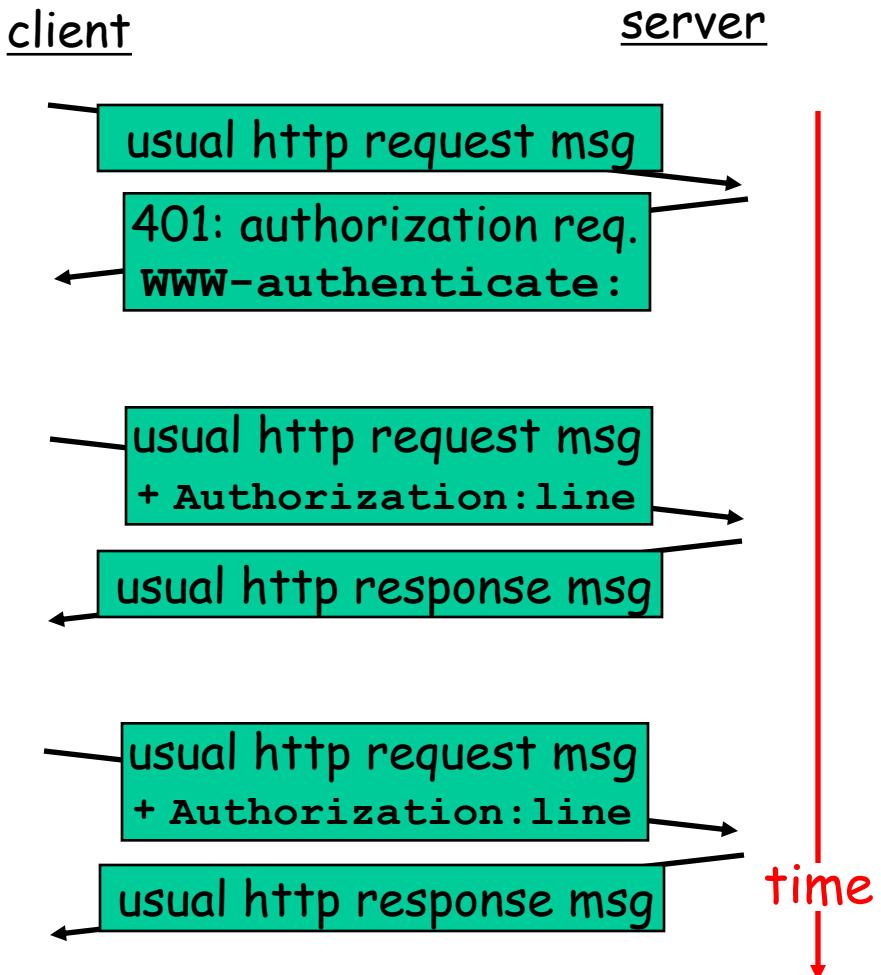
Goal: no explicit application level session

- ❑ Server sends “cookie” to client in response msg
 - **Set-cookie:** 1678453
- ❑ Client presents cookie in later requests
 - **Cookie:** 1678453
- ❑ Server matches presented-cookie with server-stored info
 - authentication
 - remembering user preferences, previous choices



Authentication of Client Request

- Authentication goal:** control access to server documents
- **stateless:** client must present authorization in each request
 - **authorization:** typically name, password
 - Authorization: header line in request
 - if no authorization presented, server refuses access, sends **www-authenticate:** header line in response



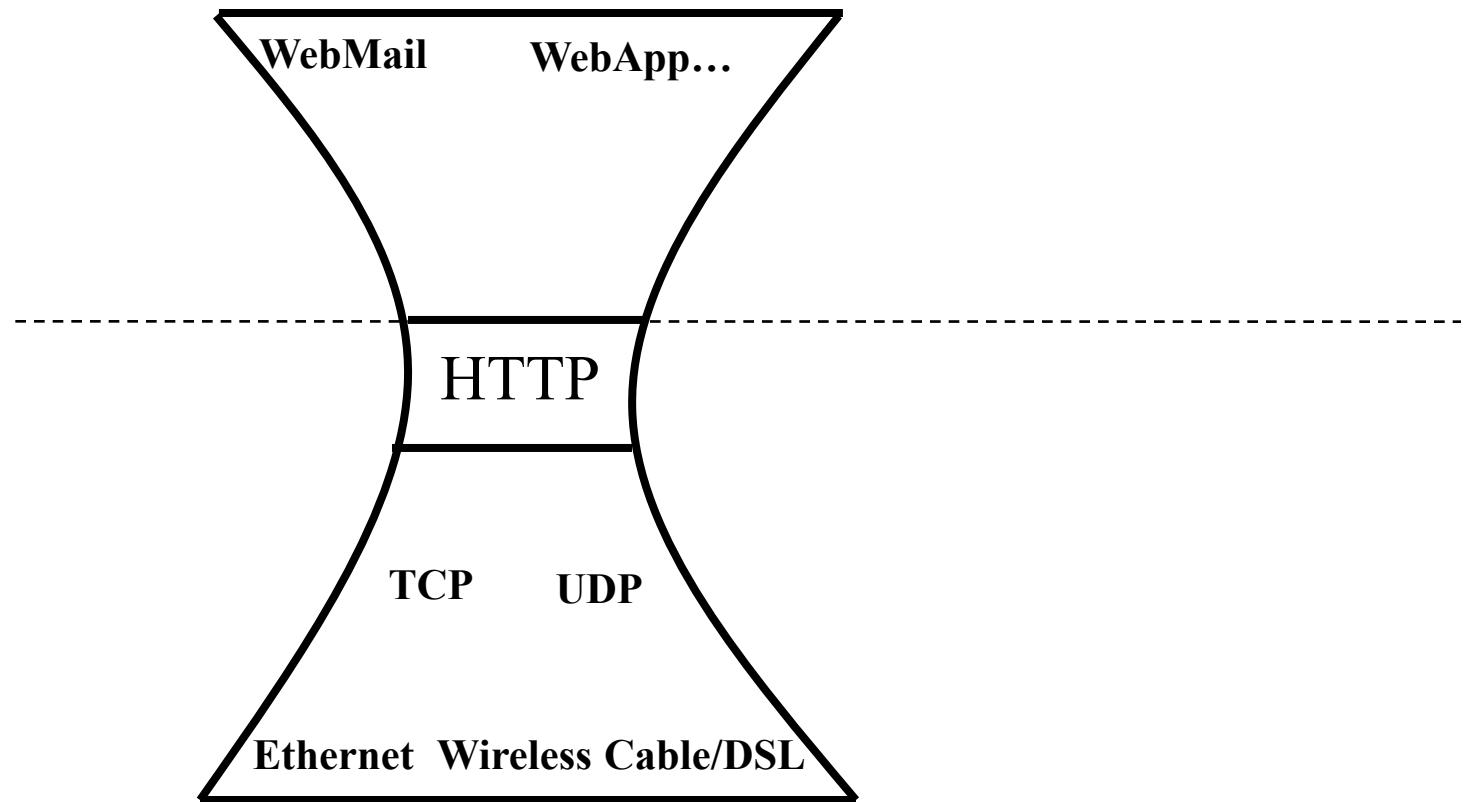
Browser caches name & password so that user does not have to repeatedly enter it.

Example: Amazon S3

□ Amazon S3 API

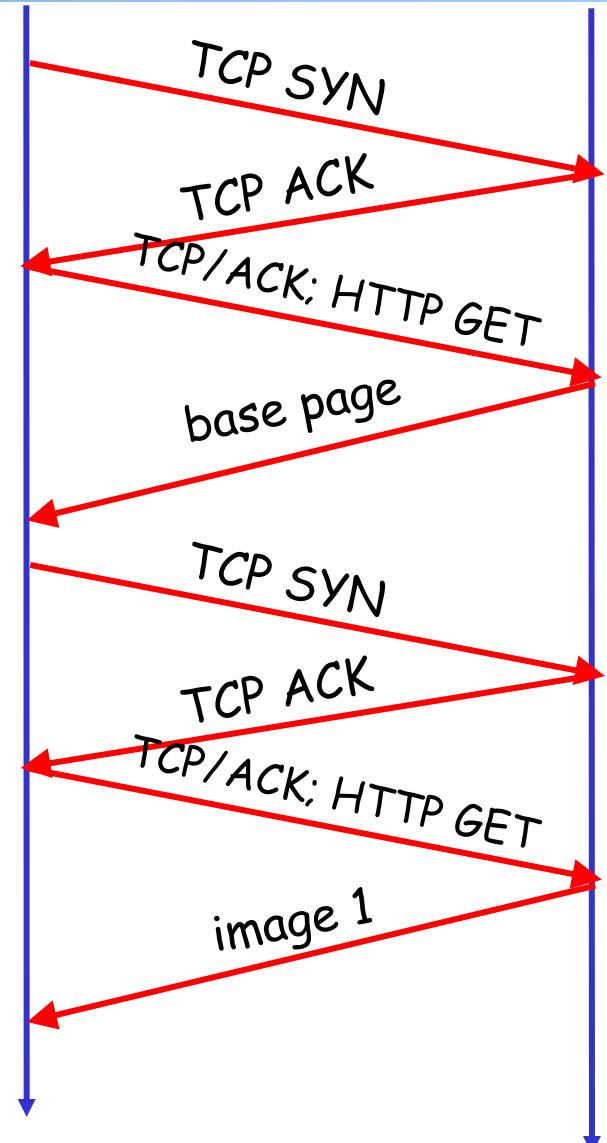
- <http://docs.aws.amazon.com/AmazonS3/latest/API/APIRest.html>

HTTP as the Thin Waist



Protocol Flow of Basic HTTP/1.0

- ≥ 2 RTTs per object:
 - TCP handshake --- 1 RTT
 - client request and server responds --- at least 1 RTT
(if object can be contained in one packet)



Outline

- Admin and recap
- HTTP/1.0
- *HTTP "acceleration"*

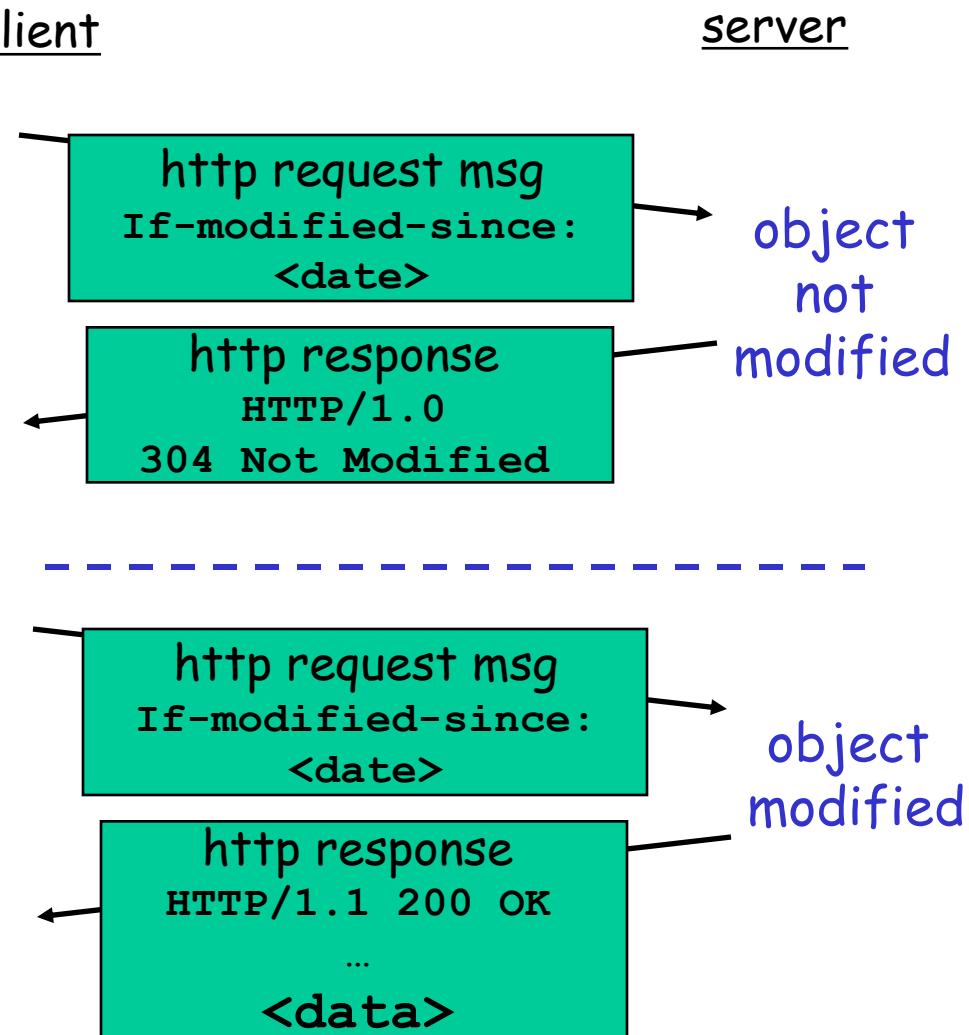
Substantial Efforts to Speedup Basic HTTP/1.0

- Reduce the number of objects fetched [Browser cache]
- Reduce data volume [Compression of data]
- Header compression [HTTP/2]
- Reduce the latency to the server to fetch the content [Proxy cache]
- Remove the extra RTTs to fetch an object [Persistent HTTP, aka HTTP/1.1]
- Increase concurrency [Multiple TCP connections]
- Asynchronous fetch (multiple streams) using a single TCP [HTTP/2]
- Server push [HTTP/2]



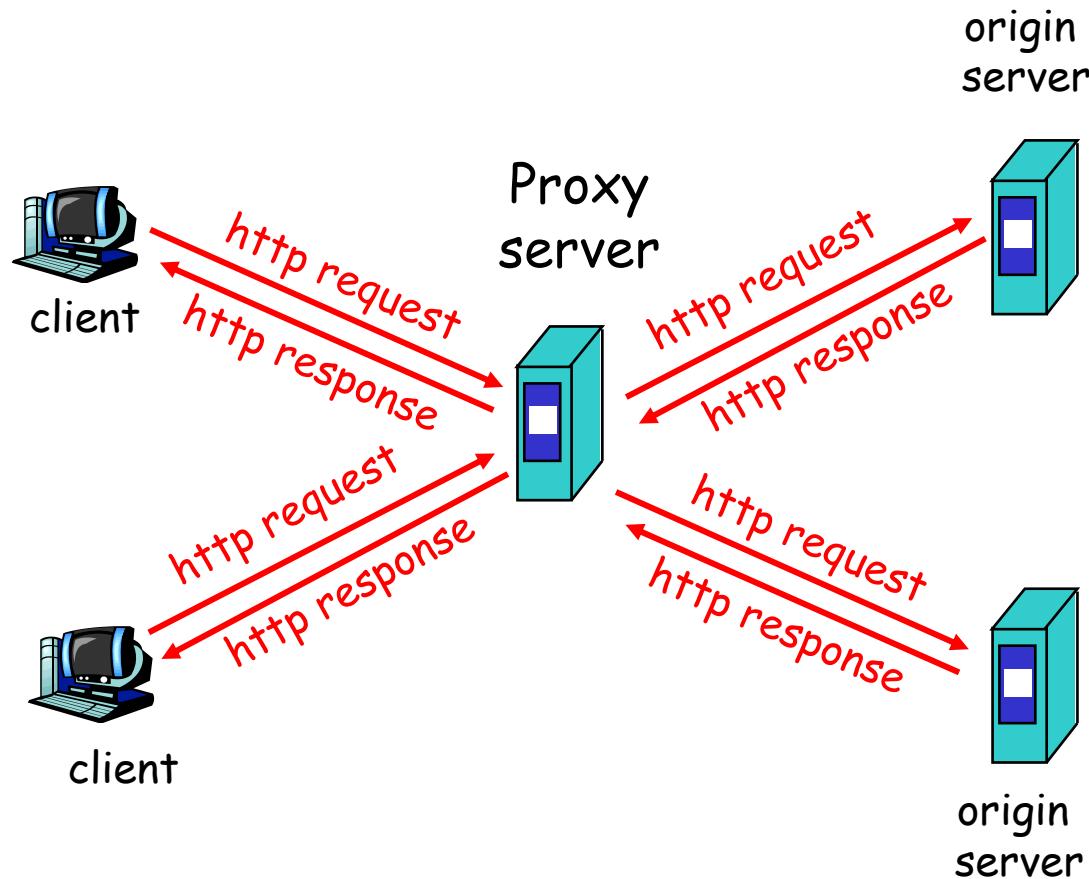
Browser Cache and Conditional GET

- **Goal:** don't send object if client has up-to-date stored (cached) version
- client: specify date of cached copy in http request
If-modified-since:
 <date>
- server: response contains no object if cached copy up-to-date:
HTTP/1.0 304 Not Modified



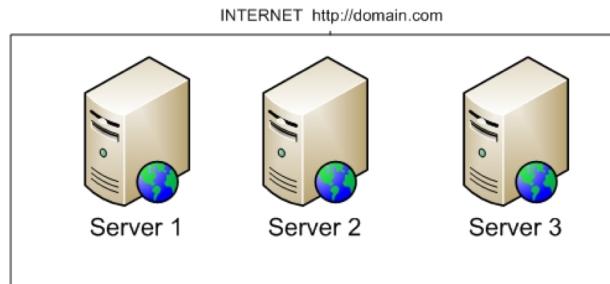
Web Caches (Proxy)

Goal: satisfy client request without involving origin server

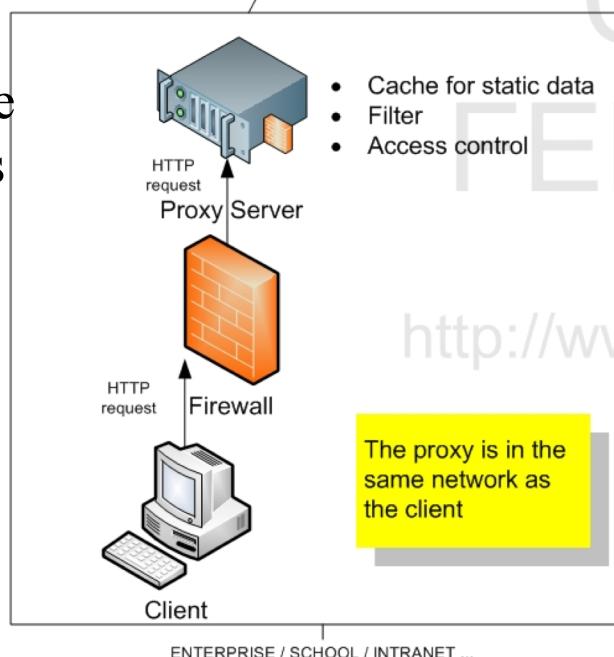


Two Types of Proxies

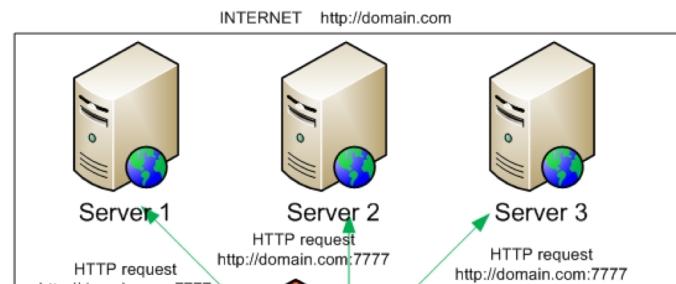
(FORWARD) PROXY server



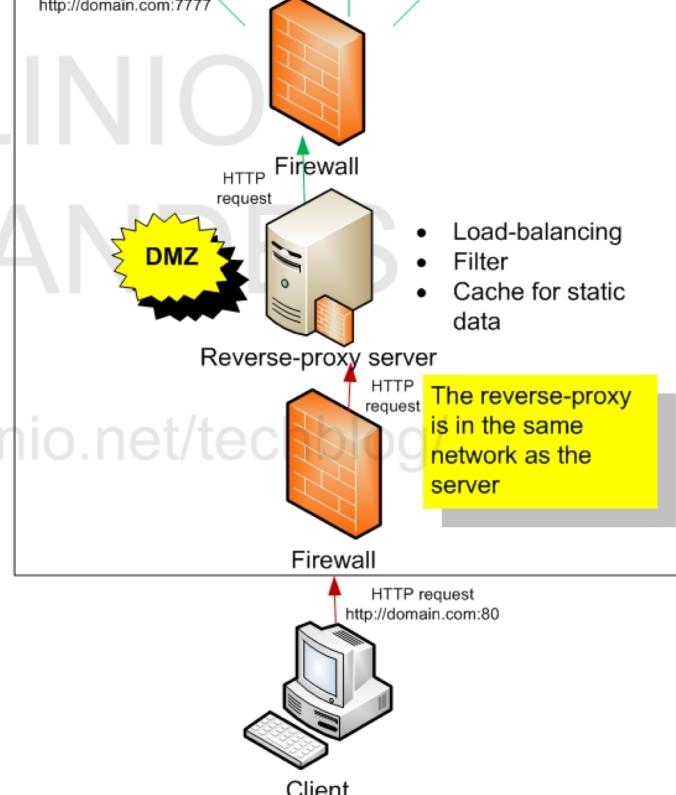
Typically
in the same
network as
the client



REVERSE-PROXY server



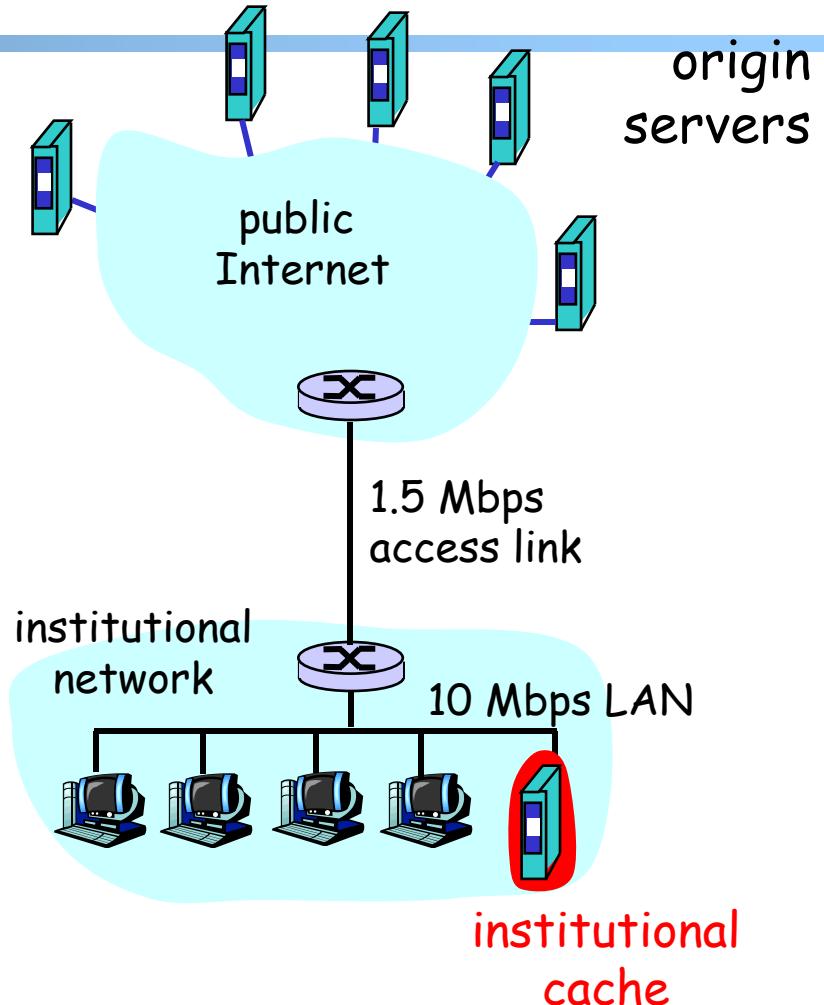
Typically in
the same
network as
the server



Benefits of Forward Proxy

Assume: cache is “close” to client (e.g., in same network)

- smaller response time: cache “closer” to client
- decrease traffic to distant servers
 - link out of institutional/local ISP network often is bottleneck



No Free Lunch: Problems of Web Caching

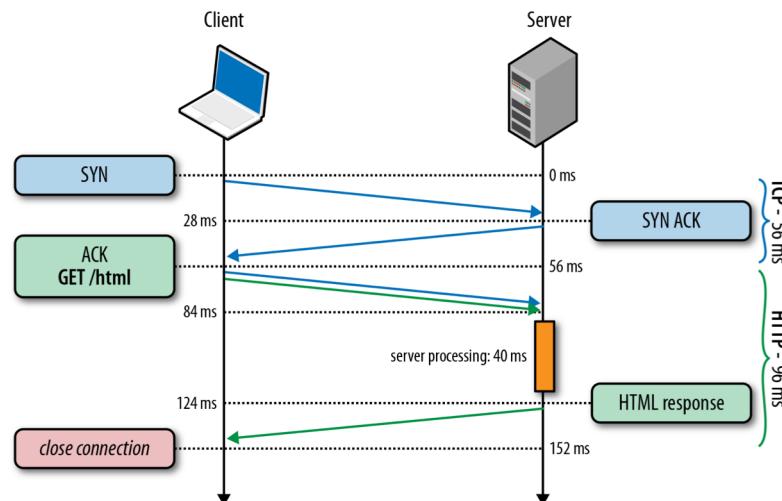
- The major issue of web caching is how to maintain consistency
- Two ways
 - pull
 - Web caches periodically pull the web server to see if a document is modified
 - push
 - whenever a server gives a copy of a web page to a web cache, they sign a lease with an expiration time; if the web page is modified before the lease, the server notifies the cache

HTTP/1.1: Persistent (keepalive/pipelining) HTTP

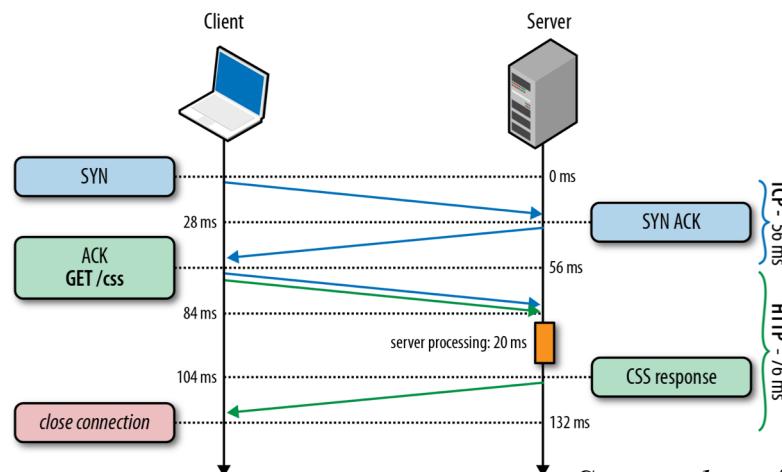
- HTTP/1.0 allows a single request outstanding, while HTTP/1.1 allows request pipelining
 - On same TCP connection: server parses request, responds, parses new request, ...
 - Client sends requests for all referenced objects as soon as it receives base HTML
- Benefit
 - Fewer RTTs
 - See Joshua Graessley WWDC 2012 talk: 3x within iTunes

HTTP/1.0, Keep-Alive, Pipelining

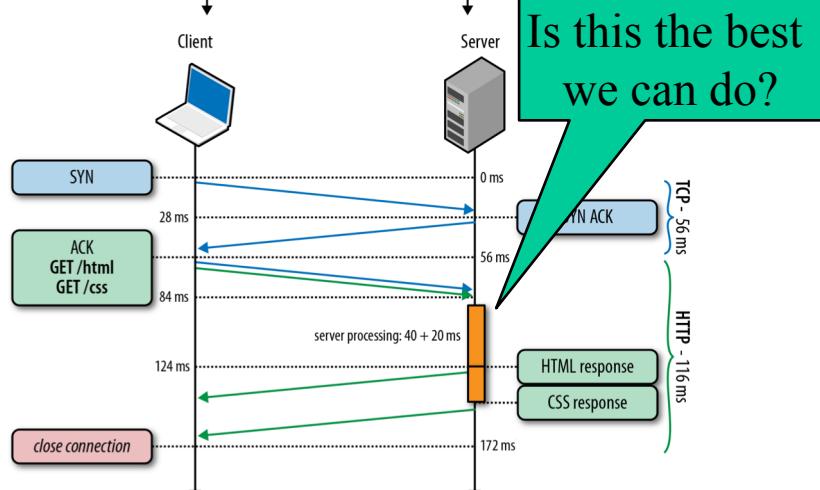
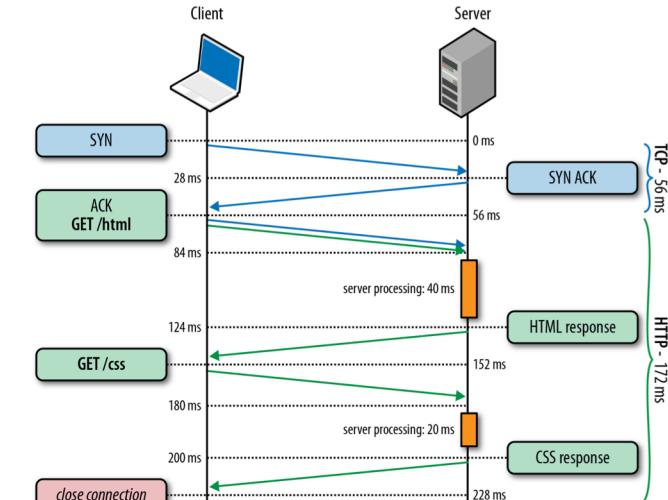
TCP connection #1, Request #1: HTML request



TCP connection #2, Request #2: CSS request

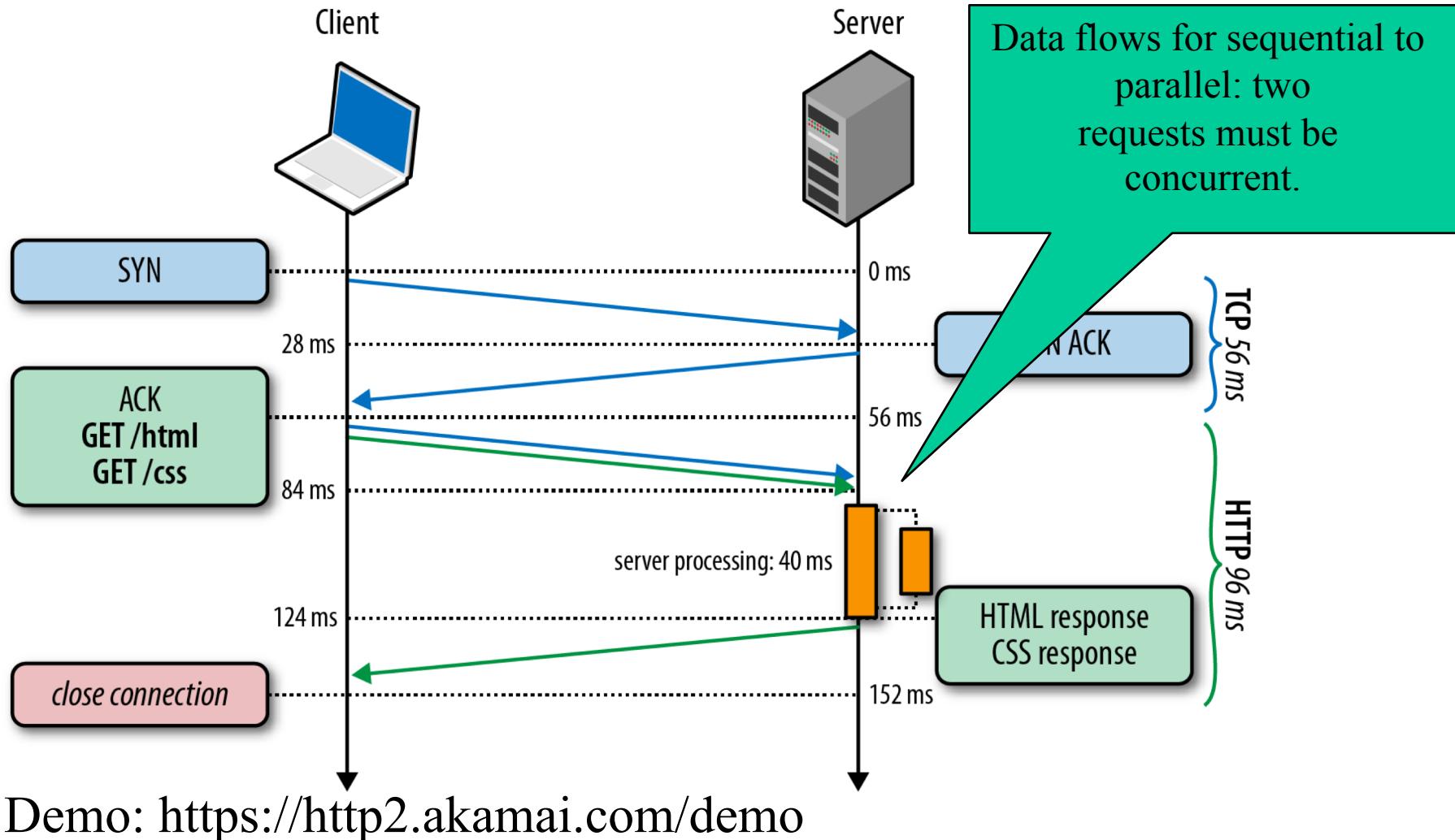


TCP connection #1, Request #1-2: HTML + CSS



Source: <http://chimera.labs.oreilly.com/books/123000000545/ch11.html>

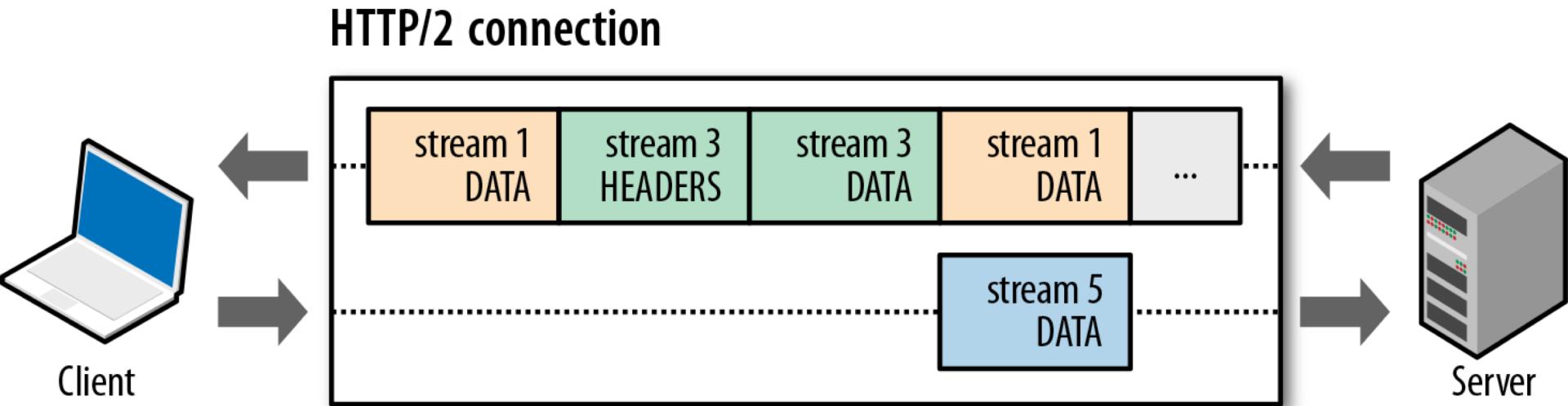
HTTP/2 Basic Idea: Remove Head-of-Line Blocking in HTTP/1.1



Observing HTTP/2

- `export SSLKEYLOGFILE=/tmp/keylog.txt`
- Start Chrome, e.g.,
 - Mac: /Applications/Google Chrome.app/Contents/MacOS/Google Chrome
 - Ubuntu: firefox
- Visit HTTP/2 pages, such as
<https://http2.akamai.com>
- Wireshark:
 - Mac: Wireshark -> preferences -> protocol -> TSL (pre)-master-secret log file name
 - Ubuntu: edit -> preferences -> protocol -> SSL (pre)-master-secret log file name

HTTP/2 Design: Multi-Streams



Bit	+0..7	+8..15	+16..23	+24..31
0		Length		Type
32	Flags			
40	R		Stream Identifier	
...			<i>Frame Payload</i>	

HTTP/2 Binary Framing

HTTP/2 Header Compression

Request headers

:method	GET
:scheme	https
:host	example.com
:path	/resource
user-agent	Mozilla/5.0 ...
custom-hdr	some-value

Static table

1	:authority	
2	:method	GET
...
51	referer	
...
62	user-agent	Mozilla/5.0 ...
63	:host	example.com
...



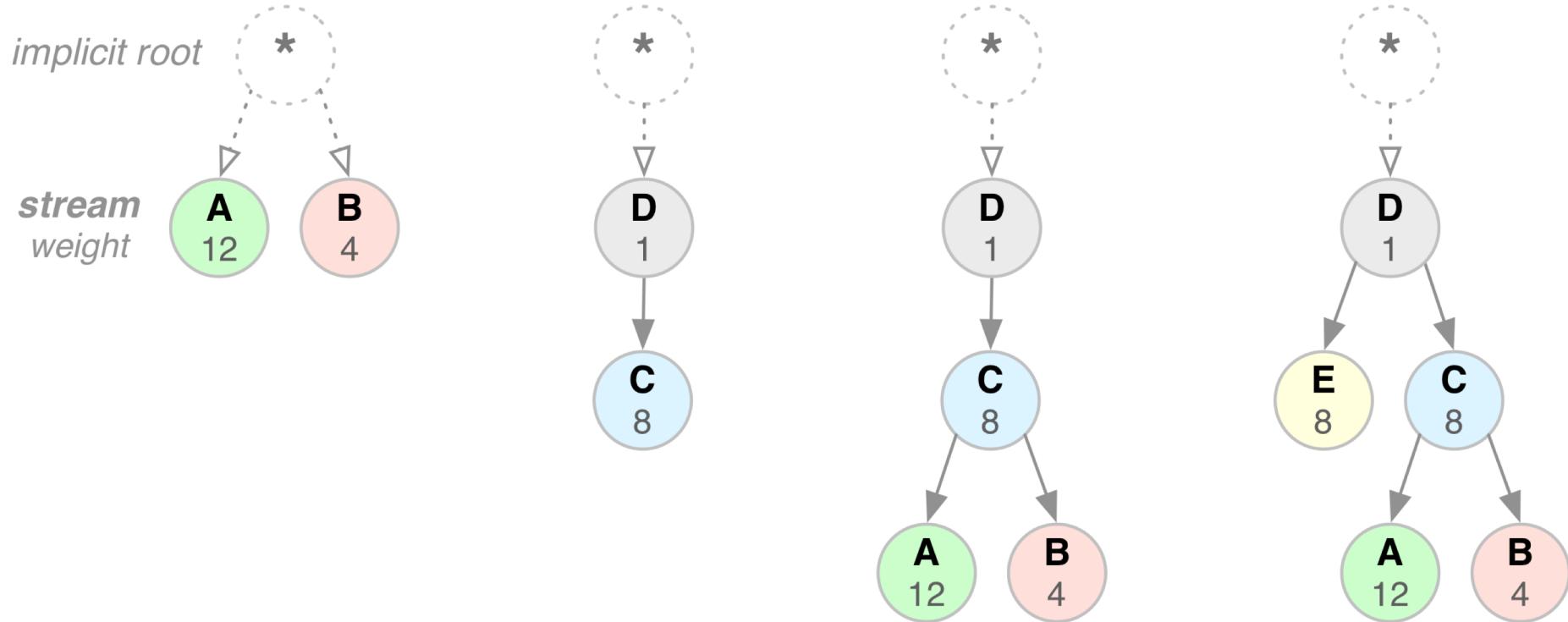
Encoded headers

2
7
63
19
Huffman("/resource")
62
Huffman("custom-hdr")
Huffman("some-value")

Dynamic table

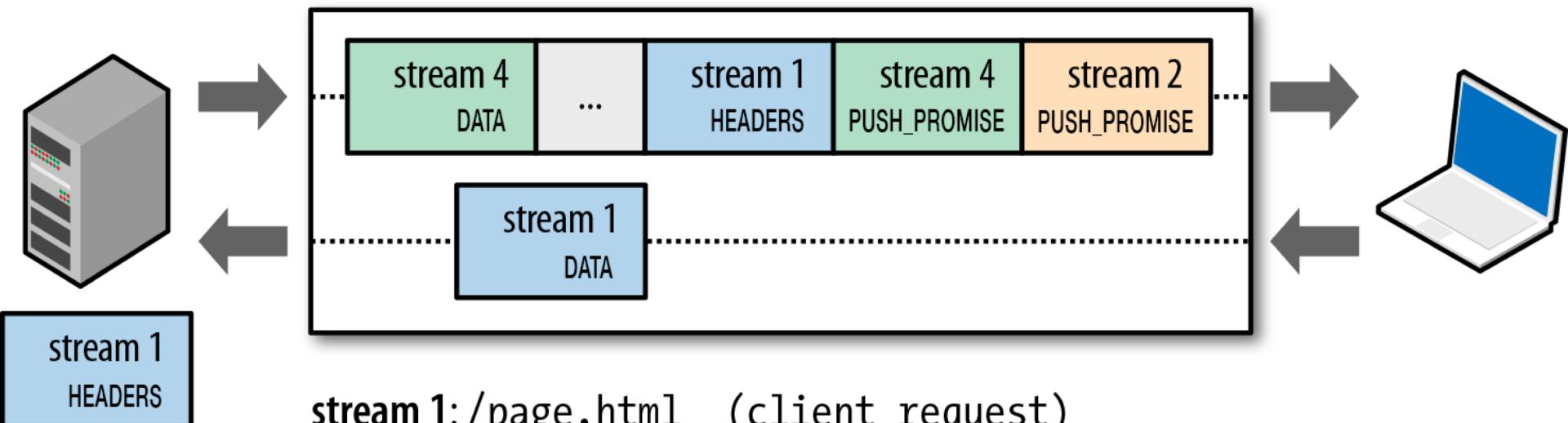


HTTP/2 Stream Dependency and Weights



HTTP/2 Server Push

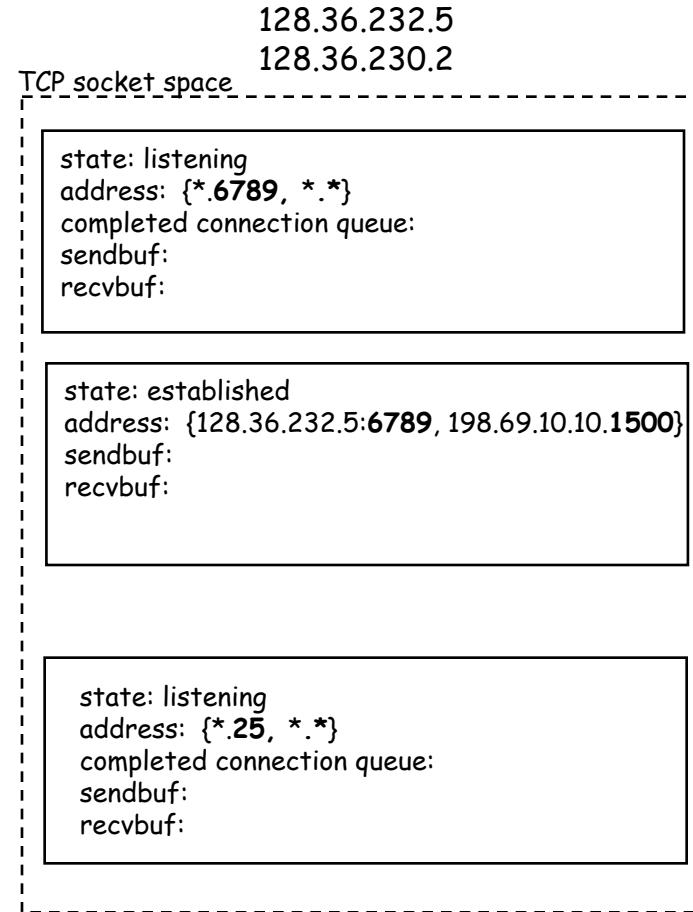
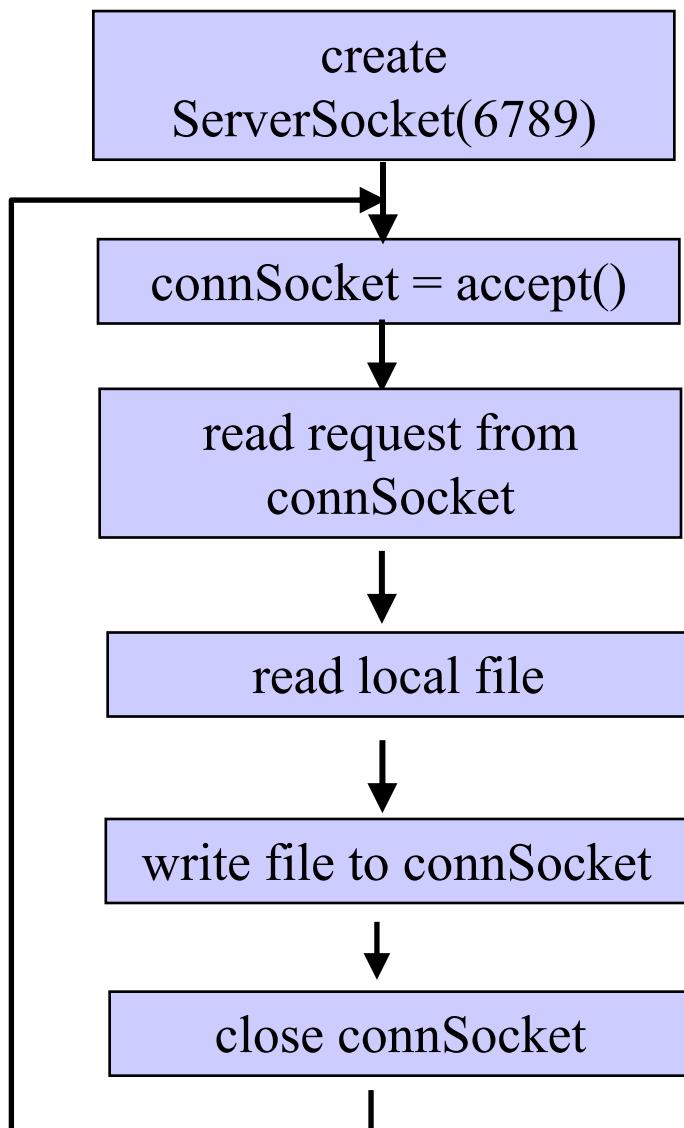
HTTP/2 connection



Outline

- Admin and recap
- HTTP/1.0
- HTTP "acceleration"
- *Network server design*

WebServer Implementation



Discussion: what does each step do and how long does it take?

Demo

- Try TCPServer
- Start two TCPClient
 - Client 1 starts early but stops
 - Client 2 starts later but inputs first