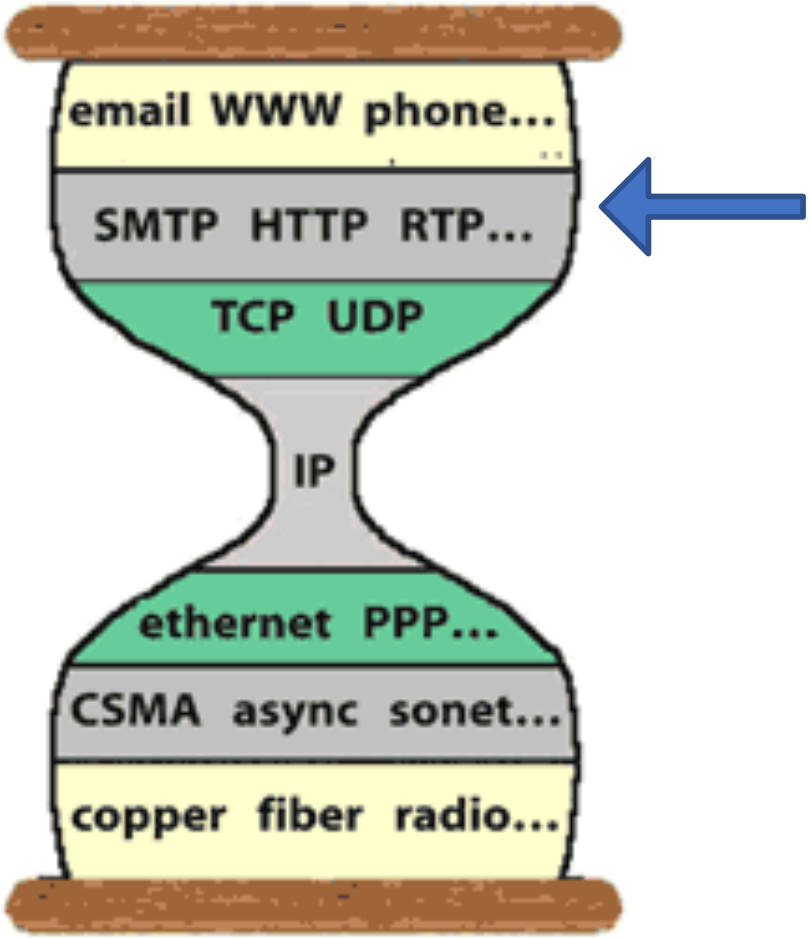
Lecture-2: HTTP

- Non-persistent HTTP, persistent HTTP; pipelining
- Web caching
- A brief intro to HTTP/2 (FYI)



What we covered in lecture-1

- Concepts:
  - **Internet**: made of a huge number of hosts and routers, interconnected by physical and wireless links
  - **Host**: a computer running applications and bunch of protocols to let apps exchange data with each other
  - **Router**: a packet switch running bunch of protocols to move packets toward their destinations
- Protocols are organized in layers:
  - Application protocols
  - Transport protocols
  - Network protocols
  - Link layer protocols
  - Physical layer
- How to calculate packet delays as they move across one hop

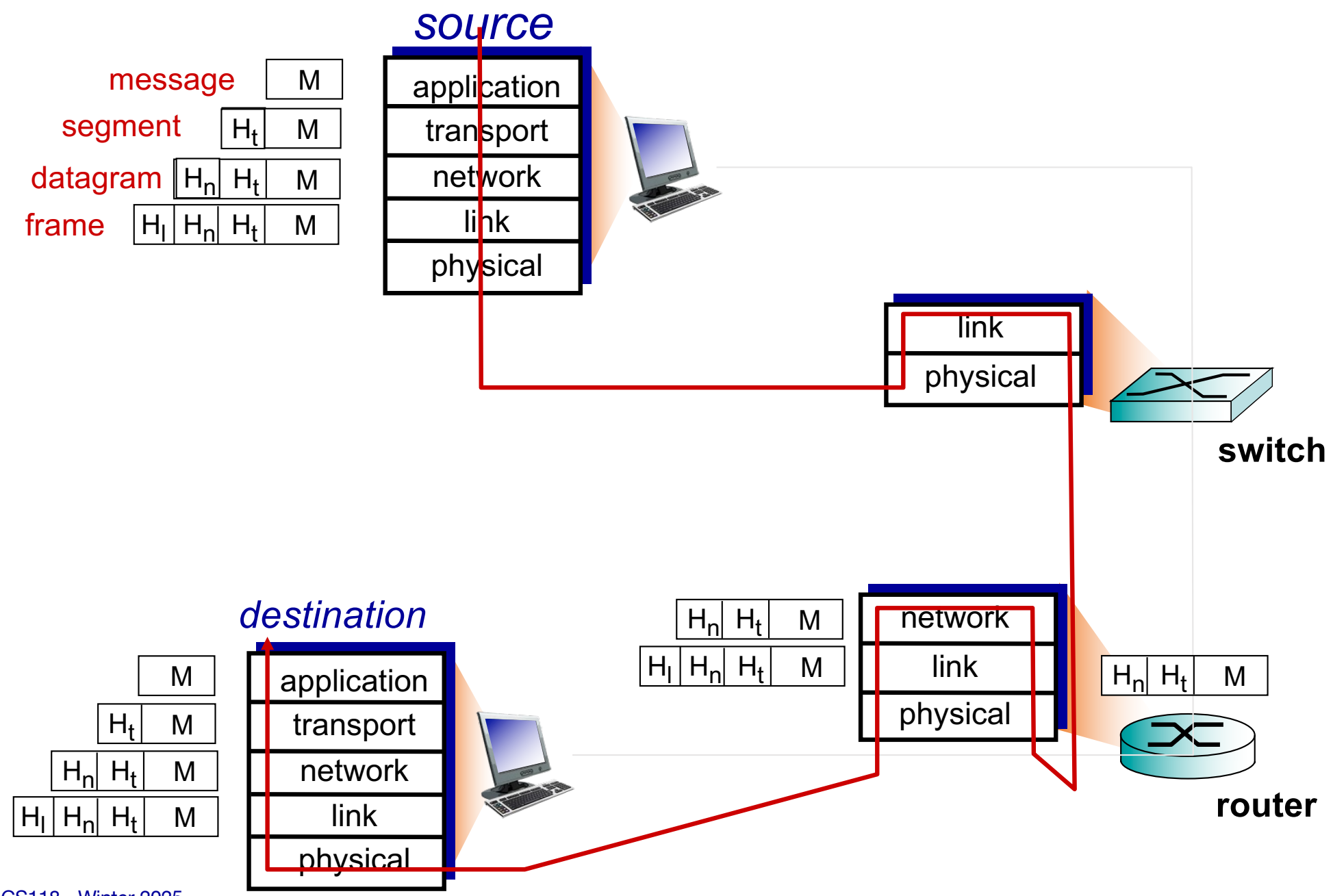
Why Layering?

Decomposed complex delivery into fundamental components

- Explicit structure allows identification, relationship of complex system's pieces
  - layered *reference model* for discussion

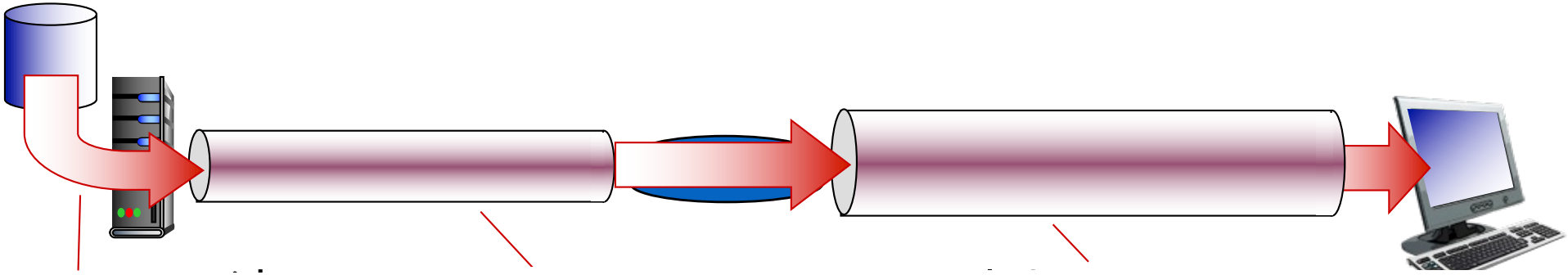
- Modularization eases maintenance, updating of system
  - change of implementation of layer's service transparent to the rest of system

Encapsulation & Decapsulation



Throughput

- *throughput:* rate (bits/time unit) at which bits transferred between sender/receiver
  - *instantaneous:* rate at given point in time
  - *average:* rate over longer period of time



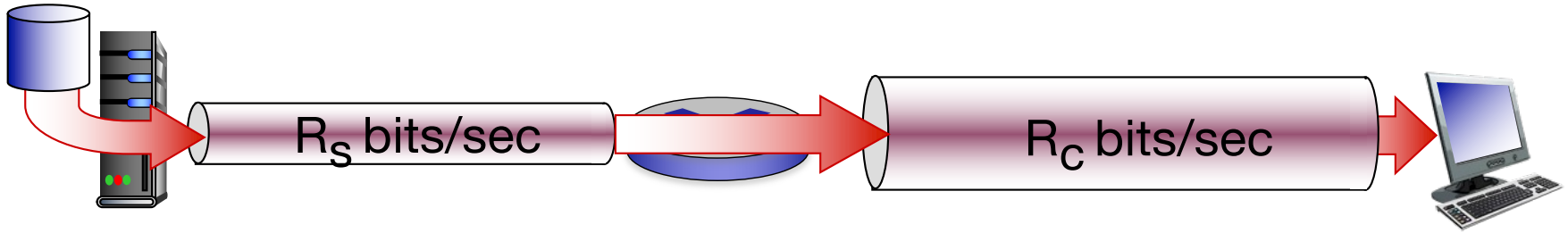
server sends bits (fluid) into pipe

pipe that can carry fluid at rate R_s bits/sec)

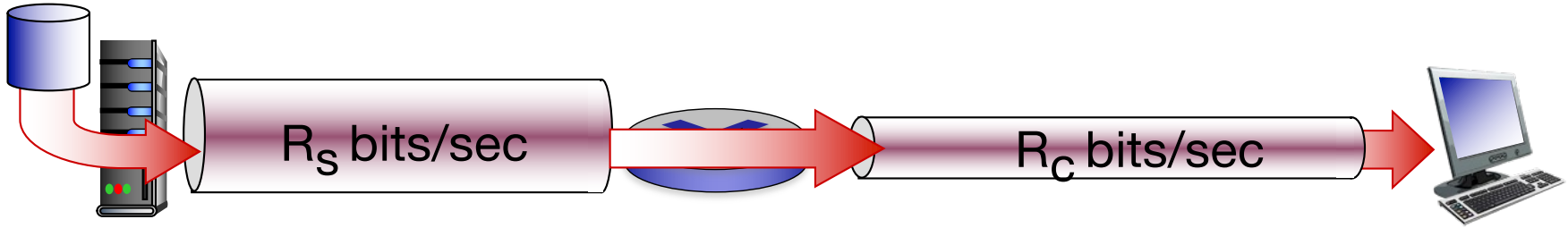
pipe that can carry fluid at rate R_c bits/sec)

Throughput (more)

- $R_s < R_c$ What is average end-end throughput?



- $R_s > R_c$ What is average end-end throughput?



The bottleneck link on end-end path that constrains end-end throughput

How to develop an Internet app?

- 3 basic concepts on Monday
  - Internet application processes
  - Internet sockets
  - Binding with address and port
- **What are the procedures to construct an Internet application?**

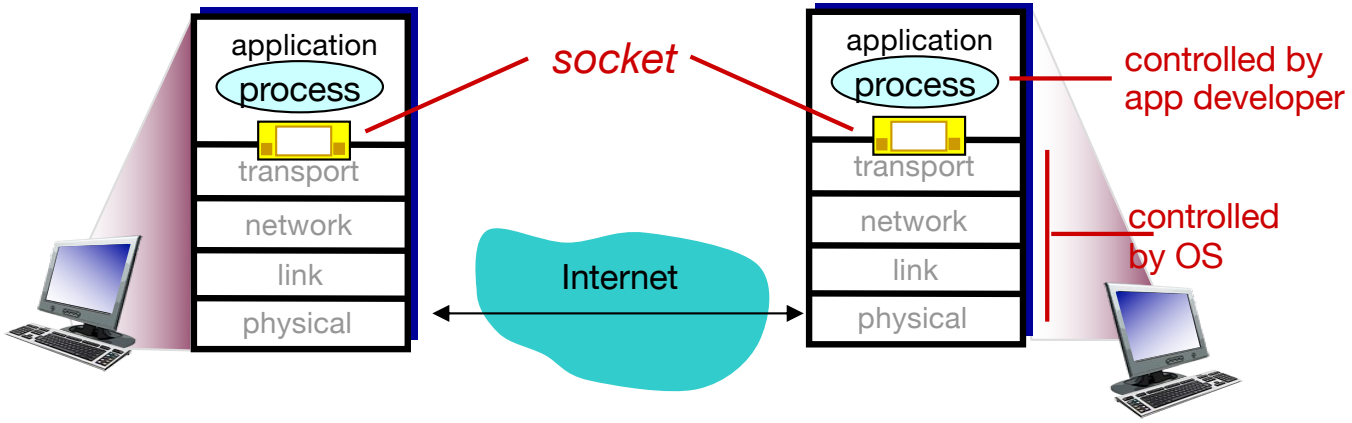
3 main steps

  - Create an application process
    - Which also needs to create an Internet socket
  - Select one from 2 transport services offered by the Internet (via sockets)
  - Define your own application protocol for your Internet application

Step 1: Internet application

- Create an Internet application process at two hosts/end systems

- Each above process creates an Internet socket (via socket API)
  - As you create a socket, you need to select what transport service to use



Step 2: Select from 2 Internet transport services

1. TCP service:

- **Connection-oriented:** setup required between client and server processes
- **Reliable transfer** between sending and receiving process
- **Control:**
  - **flow control:** sender won't overwhelm receiver
  - **congestion control:** throttle sender when network overloaded
- **does NOT provide:** timing, minimum throughput guarantee, security

2. UDP service:

- **Connectionless: no connection setup**
- **Unreliable data transfer** between sending and receiving process
- **No control**
  - You control the sending rate
- **does NOT provide**: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

What transport service to choose for your app?

Data integrity

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

Timing

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

Throughput

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

Security

- encryption, data integrity, …

Apps requirements on transport service

Application	Data loss	Throughput	Time sensitive?
File transfer	no loss	elastic	no
Email	no loss	elastic	no
Web page	no loss	elastic	no
Real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
Streaming audio/video	loss-tolerant	same as above	yes, few secs

Choices of transport protocols services by apps

Application	Application layer protocol	Transport protocol
File transfer	FTP [RFC 959]	**TCP**
Email	SMTP [RFC 5321]	**TCP**
Web page	HTTP 1.1 [RFC 7230]	**TCP**
Real-time audio/video	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	**TCP or UDP**
Streaming audio/video	HTTP [RFC 7230], DASH	**TCP**

Step 3: Define your application-layer protocol:

- **Types of messages exchanged,**
  - e.g., request, response
- **Message syntax:**
  - what fields in messages & how fields are delineated
- **Message semantics**
  - meaning of information in fields
- **Rules** for when and how processes send & respond to messages

Open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

Proprietary protocols:

- e.g., Skype

Choices of transport protocols services by apps

Application	Application layer protocol	Transport protocol
File transfer	**FTP [RFC 959]**	TCP
Email	**SMTP [RFC 5321]**	TCP
Web page	**HTTP 1.1 [RFC 7230]**	TCP
Real-time audio/video	**SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)**	TCP or UDP
Streaming audio/video	**HTTP [RFC 7230], DASH**	TCP

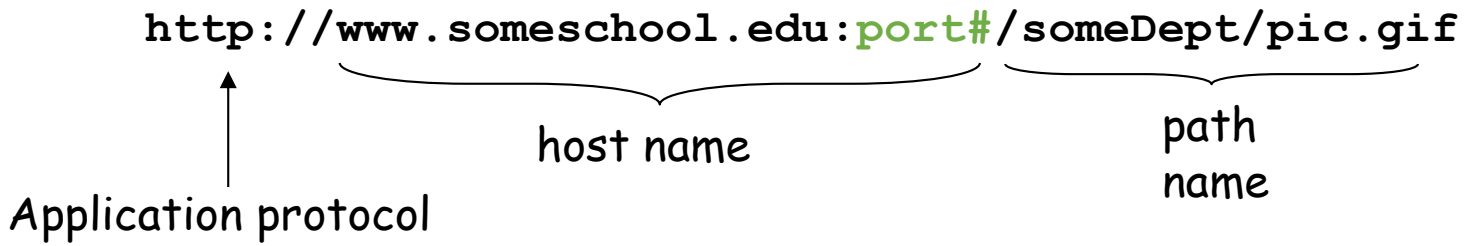
Summary: How to develop an Internet app?

- What are the procedures to construct an Internet application?
  - Create an application process (executing application program)
    - Which also needs to create an Internet socket
  - Select from 2 of the transport services offered by the Internet (via sockets)
  - Define your own application protocol for your Internet application

Let's look at exactly what data is exchanged

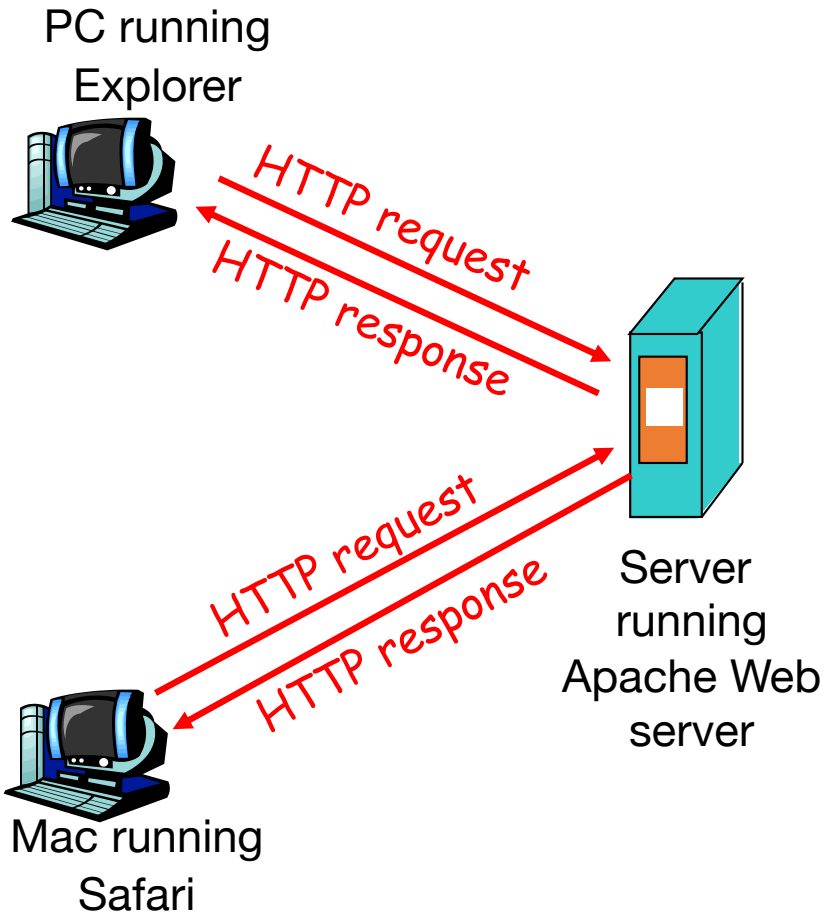
Web and HTTP

- Web page: normally consists of
  - base HTML-file, which includes
  - several referenced objects
- An object can be another HTML file, JPEG image, Java applet, audio file,…
- Each object is addressable by a URL (Universal Resource Locator)



HTTP: HyperText Transfer Protocol

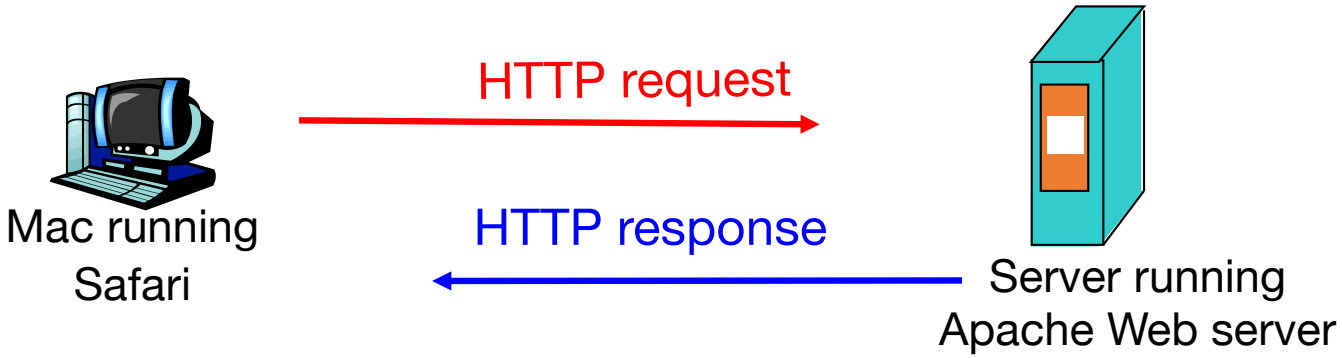
- Web's application layer protocol
- Client/Server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- HTTP/1.0: non-persistent connection
- HTTP/1.1: persistent connection
  - May also pipelining



HTTP runs over TCP

- Client initiates TCP connection to server on port 80
  - creates socket
- Server accepts TCP connection request from client
- HTTP data exchanged between browser (HTTP client) and Web server (HTTP server)
- When done: close the TCP connection
  - Which side initiates the close? Later

Now we got the big picture



- Client (browser) speaks first
  - Set up a TCP connection (det
  - Send HTTP request
- Server answers the request
  - HTTP is "stateless": server maintains no information about past requests

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

Exactly how HTTP request & reply messages look like?

HTTP request message example

FYI

Written in ASCII (human-readable)

```
http://www.httpforever.com:port#/some-dir/pic.gif
```

host name

optional, default value: 80

path name

carriage return character

line-feed character

method URL version

request line

```
GET /index.html HTTP/1.1\r\n
Host: www.httpforever.com\r\n
User-Agent: Chrome/131.0.0.0\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
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

header lines

A blank line indicates end of header

Optional message body

Method types

HTTP/1.0

- GET
- POST
  - Web page often includes form input
  - User input sent from client to server in entity body of HTTP POST request message
- HEAD
  - Requesting the header only (i.e. response does not include the requested object)

HTTP/1.1

- GET, POST, HEAD
- PUT
  - uploads (or completely replace) file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field from the server
- and a few others
  - See the protocol specification RFC2616

HTTP response message

FYI

status line (status code, status phrase)

header lines

A blank line

Data: e.g., requested HTML file

```
HTTP/1.1 200 OK \r\n
Server: nginx/1.18.0 (Ubuntu)\r\n
Date: Wed, 08 Jan 2025 03:34:37 GMT\r\n
Content-Type: text/html\r\n
Content-Length: 5124\r\n
Last-Modified: Wed, 22 Mar 2023 14:54:48 GMT\r\n
Connection: keep-alive\r\n
ETag: "641b16b8-1404”\r\n
Accept-Ranges: bytes\r\n
\r\n
```

Optional message body

```
data data data data data ...
```

HTTP response status codes

- Appears in the first line in server→client response message:
- A few 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 (Location:)

400 Bad Request

- request message not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP request for yourself

1. Telnet to a Web server:

telnet www.httpforever.com 80

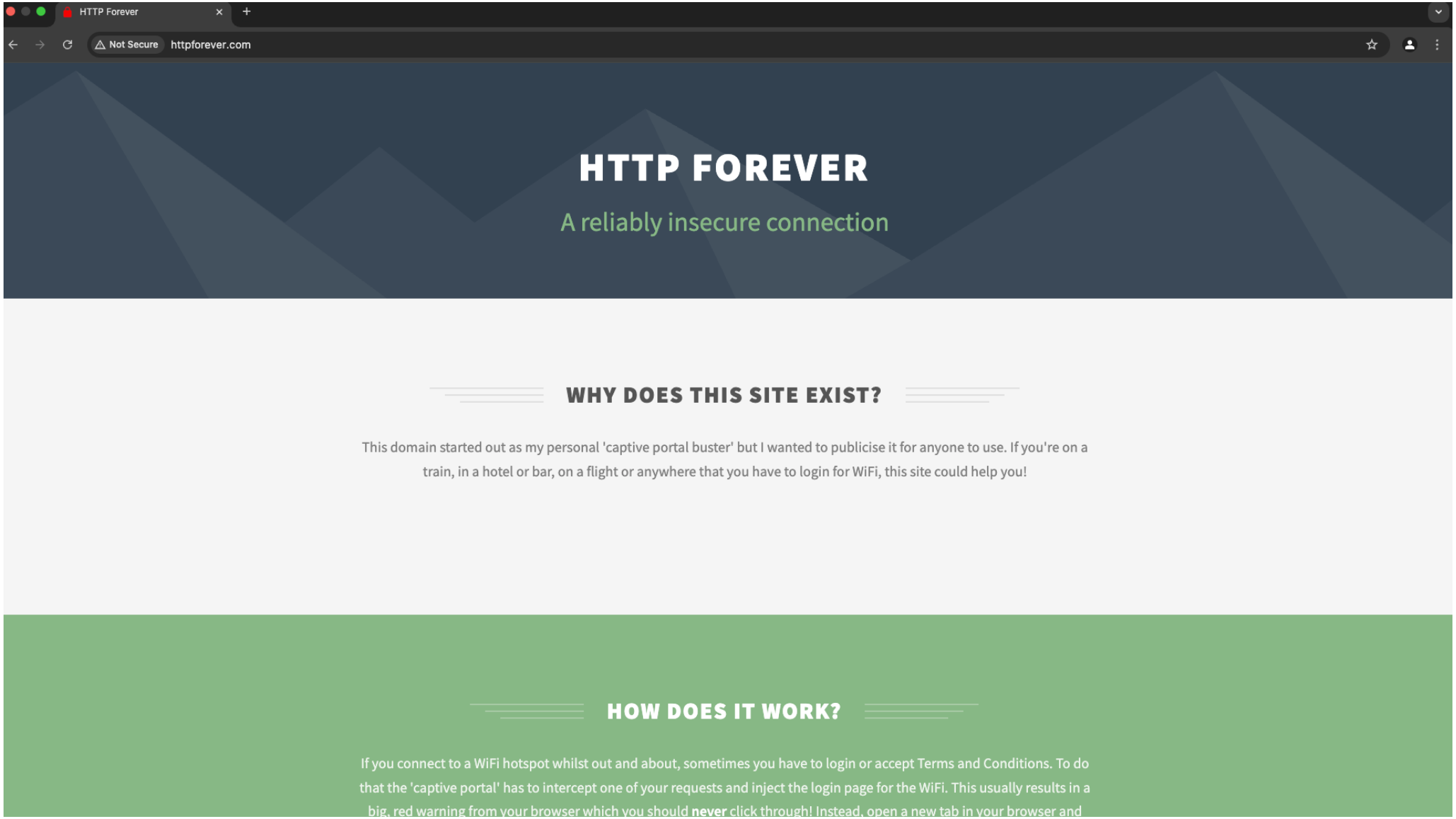
Opens TCP connection to port 80
(default HTTP server port) at **www.httpforever.com**
Anything typed in is sent
to port 80 at www.httpforever.com

2. Type in a GET HTTP request:

GET /index.html HTTP/1.1
Host: www.httpforever.com

By typing this in (hit carriage return ***twice***), you send this minimal (but complete) GET request to HTTP server

3. Look at response message from the HTTP server!


HTTP Forever
Not Secure
httpforever.com
HTTP FOREVER
A reliably insecure connection
WHY DOES THIS SITE EXIST?
This domain started out as my personal 'captive portal buster' but I wanted to publicise it for anyone to use. If you're on a train, in a hotel or bar, on a flight or anywhere that you have to login for WiFi, this site could help you!
HOW DOES IT WORK?
If you connect to a WiFi hotspot whilst out and about, sometimes you have to login or accept Terms and Conditions. To do that the 'captive portal' has to intercept one of your requests and inject the login page for the WiFi. This usually results in a big, red warning from your browser which you should never click through! Instead, open a new tab in your browser and

http://www.httpforever.com/index.html

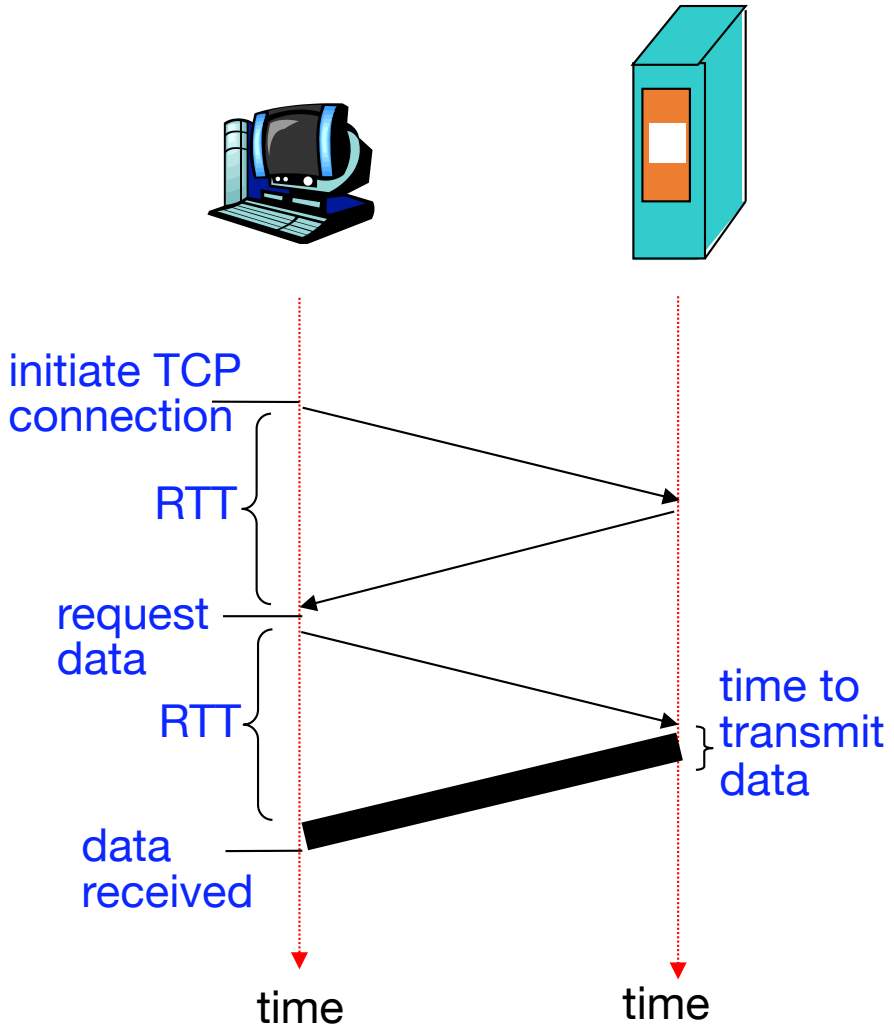
```
<!DOCTYPE HTML>
<html>
<head>
<title>HTTP Forever</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<meta name="description" content="A site that will always be available over HTTP!" />
<meta name="keywords" content="HTTP WiFi Captive Portal" />
<!--[if lte IE 8]><script src="css/ie/html5shiv.js"></script><![endif]-->
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js" integrity="sha256-
FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/skel/3.0.1/skel.min.js" integrity="sha256-
3e+NvOq+D/yeJy1qrWpYkEUr6SlOCL5mHpc2nZfX74E=" crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/skel-layers/2.2.1/skel.min.js" integrity="sha256-
6xgf/CipbscdlAaUOAAlWVmpfPy9V5cQvZejxXSEfcw=" crossorigin="anonymous"></script>
<script src="js/init.min.js"></script>
<noscript>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/skel-layers/2.2.1/skel.min.css"
integrity="sha256-HoTbojxjAGIeiQMgAD2nqi6adFwcOUwoiPnr7mC7qBs=" crossorigin="anonymous" />
<link rel="stylesheet" href="css/style.min.css" />
<link rel="stylesheet" href="css/style-wide.min.css" />
</noscript>
<!--[if lte IE 8]><link rel="stylesheet" href="css/ie/v8.min.css" /><![endif]-->
</head>
<body class="landing">
<section id="banner">
<h2>HTTP FOREVER</h2>
<p>A reliably insecure connection</p>
</section>
<div class="wrapper style1">
<section class="container">
<header class="major">
<h2>Why does this site exist?</h2>
```

Non-Persistent HTTP (HTTP 1.0)

RTT: time between client sending a *small packet* to server and receiving the reply

Time needed to fetch a single web object:

- one RTT to set up TCP connection
- one RTT for HTTP request and first byte of HTTP response to reach the client
- object transmission time



Total = 2RTT + object transmission time

Total time to fetch a simple web page

User clicks URL www.httpforever.com/index.html

Host www.httpforever.com runs HTTP server process, waiting for incoming requests

1. HTTP client sends TCP connection open request to www.httpforever.com

2. HTTP server responds with "accept TCP connection"

3. HTTP client sends HTTP request message asking for index.html

4. HTTP server receives request message, forms a reply containing requested object, and sends it.

5. HTTP client receives response message (index.html file), parsing the html file, finds 15 referenced objects

6. Steps 1-5 repeated for each of the 15 objects

As soon as the server receives ACK for the reply (index.html), the server closes TCP connection.

time

Total time: 2 x (15 +1) = 32 RTTs, plus transmission time for the total 16 files

Q: How to improve?

At most **one** object is sent over **one** TCP connection

After opening a TCP connection, use it to fetch send multiple objects (**Persistent HTTP**)

open multiple TCP connections in parallel, one for each object

Persistent HTTP

Reuse the same TCP connection to transfer multiple objects

- Server leaves connection open after sending response
- Subsequent HTTP messages over the open connection
- Client sends requests as soon as it encounters a referenced object
- One RTT for each referenced object

Persistent HTTP (HTTP 1.1)

User clicks URL www.httpforever.com/index.html

Host www.httpforever.com runs HTTP server process, waiting for incoming requests

1. HTTP client sends TCP connection open request to www.httpforever.com

2. HTTP server responds with "accept TCP connection"

3. HTTP client sends HTTP request message asking for object index.html

4

5. HTTP client receives response message (index.html file), parsing the html file, finds 15 referenced objects

After sending out first object, keep the connection open for a short time period, so if there is next request, it can use the same connection

At Step 5: use the established TCP connection to send 15 HTTP requests for the 15 referenced objects ⇨ total time needed: 17RTT + trans. Time for 16 objects

time

HTTP 1.0 vs 1.1 summary

Nonpersistent HTTP(1.0)

- At most one object is fetched over a single TCP connection between client and server.
- Described in RFC1945

  http://tools.ietf.org/html/**rfc1945**

Can set up multiple TCP connections in parallel to speed up data fetching

Persistent HTTP (1.1)

- Multiple objects can be fetched over a single TCP connection.
- Described in RFC2616

⇦ Does HTTP 1.1 need to do the same?

Persistent HTTP: one more detail

Persistent ***without*** pipelining:

- client issues next request after the previous response *has been received*
- Total delay: one RTT for *each* object plus data transfer time (after TCP connection setup)

Persistent ***with*** pipelining:

- client sends requests as soon as it sees a referenced object
- Total delay: one RTT plus data transfer time for all objects (after TCP connection setup)

Three factors in HTTP fetching

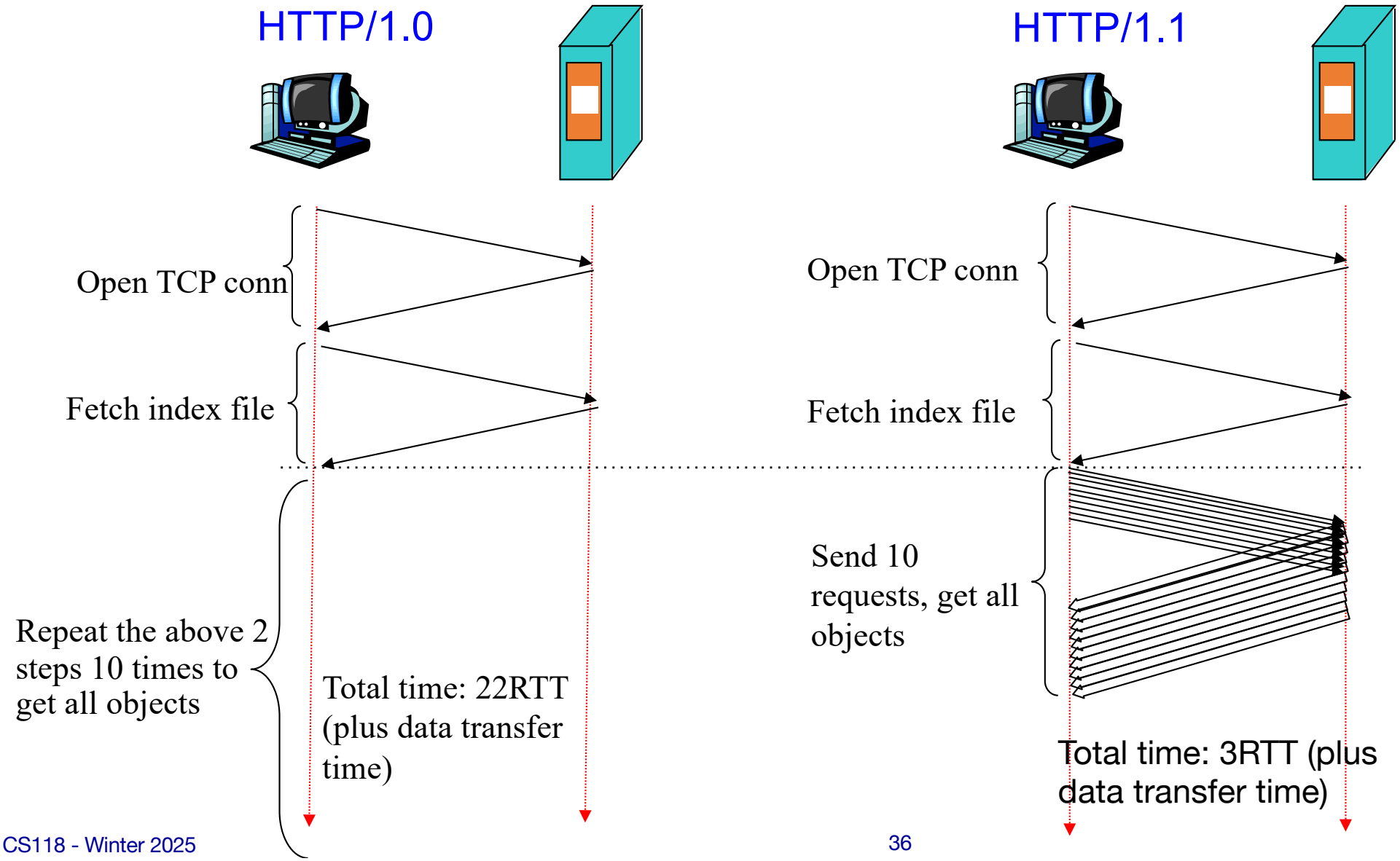
	1. Set up parallel connections, or not	2. Use persistent connection or not	3. Use, or not use, pipelining
HTTP1.0	Can do	No	No
HTTP1.1	Can do	Yes	Can do both

Example: Non-persistent HTTP vs. persistent HTTP

- A web page (base html) with 10 (small) referenced objects. How long for client to receive all them all?
  - Ignore the object transmission time
  - Non-persistent HTTP (with 1 object one time)
    - 2RTT + 2RTT* 10 = 22 RTT
  - Persistent HTTP
    - 2RTT + 10 RTT = 12 RTT
  - Non-persistent HTTP (5 objects in parallel)
    - 2RTT + 2RTT (1~5 objects) + 2RTT (6~10 objects) = 6RTT

HTTP/1.0 vs HTTP/1.1 with pipelining

Fetching one index.html file ***and*** 10 referenced jpeg files



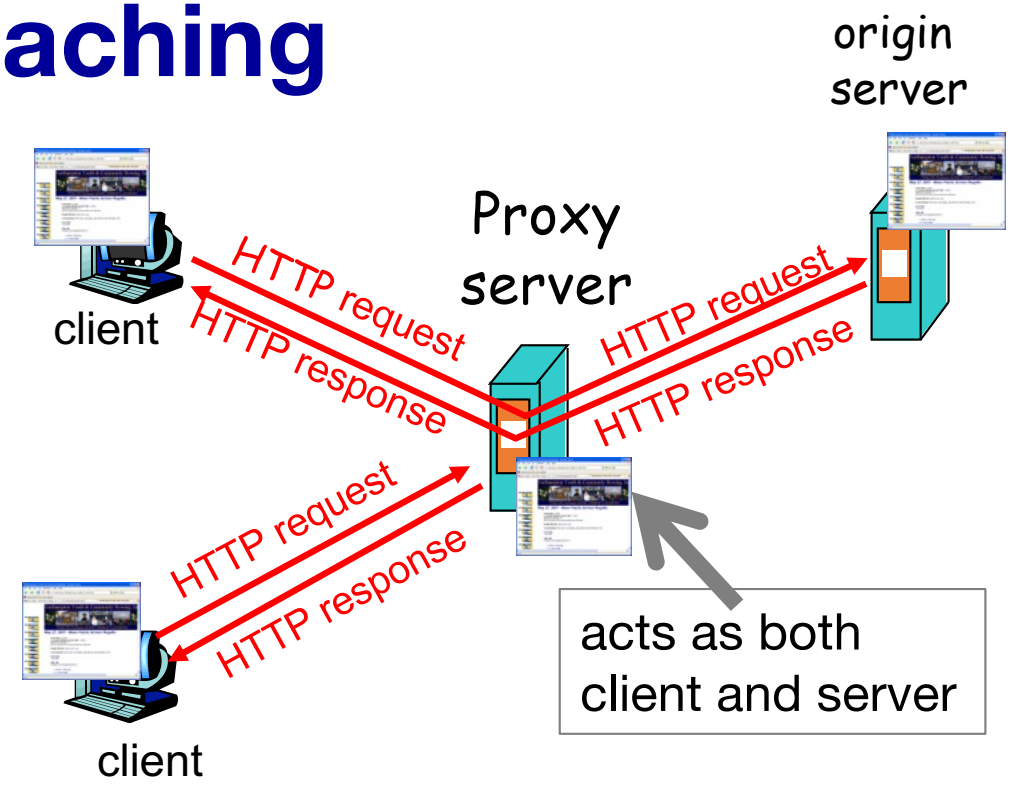
How to scale web services?

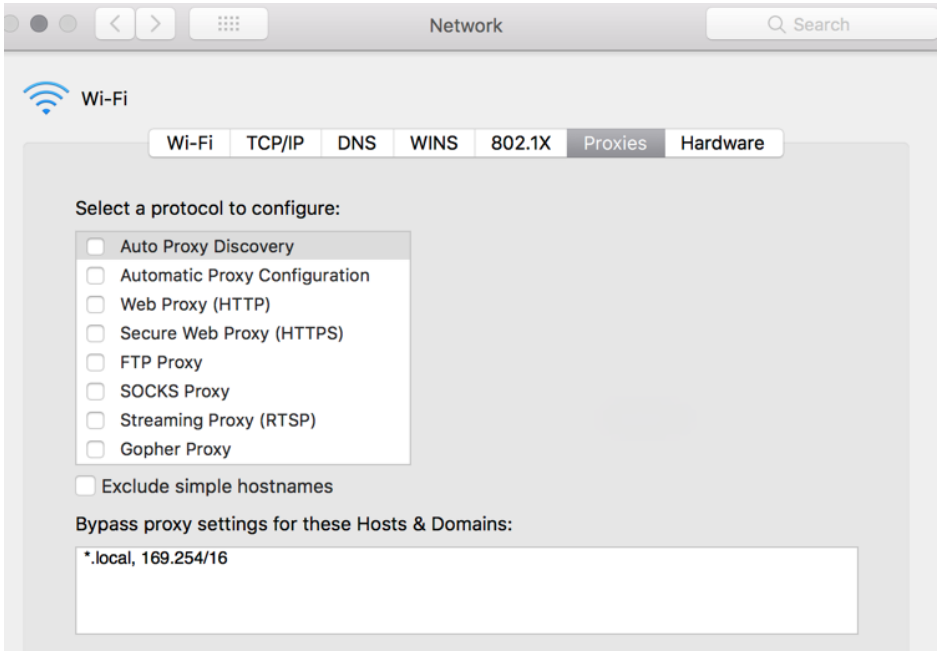
- Popular websites may receive thousands of requests per second
- Each web server can only handle limited number of users at any given time
- Popular web contents requested by many users

Cache popular contents, serve user requests from caches

Web caching

- Configure each browser to send web requests to a proxy server (cache)
  - For each request: open a TCP connection with the cache
- Cache:
  - *If* a requested object in cache: returns the object
  - *else* fetches the object from server, then returns object to client, and save a copy





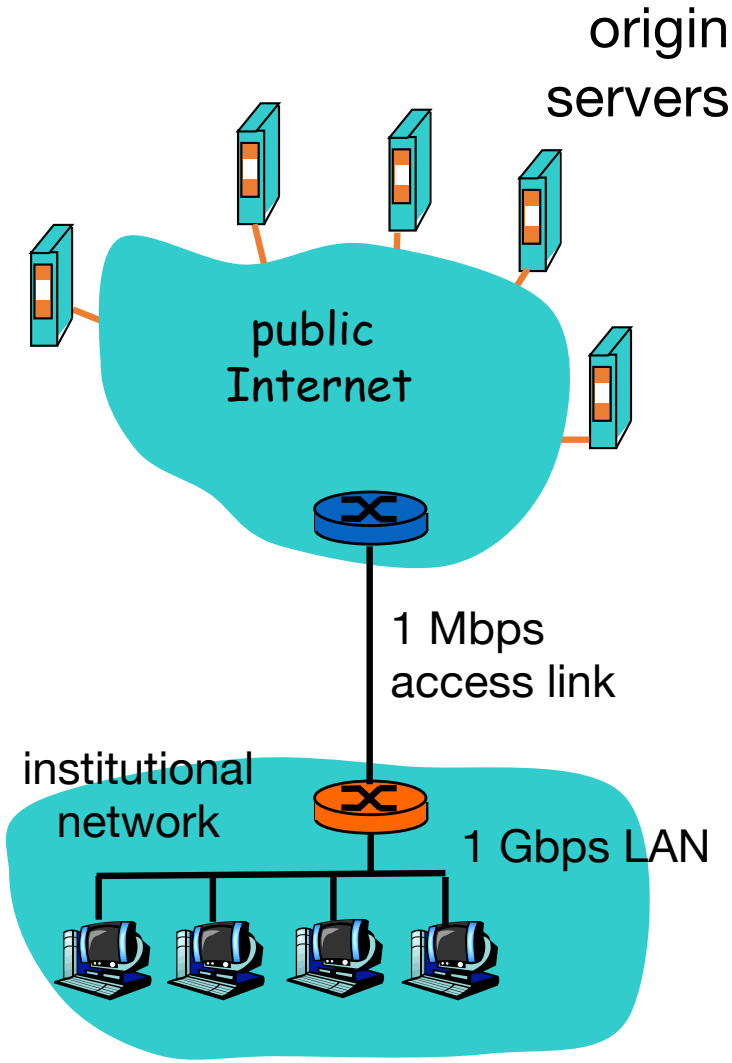
Example: without caching

Assumptions

- avg. web request rate = 10 reqs/sec
- average object size = 100,000 bits
  - 10 objects → 1M bits
- RTT from institutional router to any web server and back = 500msec

Consequences

- utilization on LAN = 1%
- utilization on access link = 100%
  - Queuing delay at institutional router: may grow without bound
- delay = 500msec + queuing delay



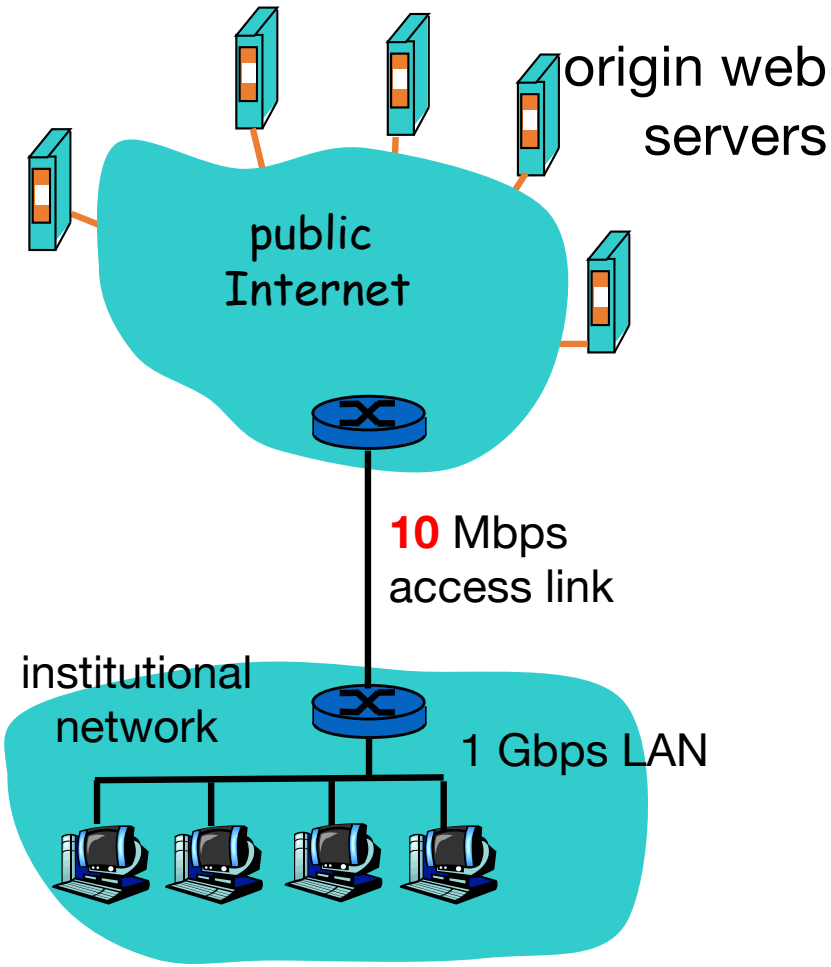
Option-1: buy more bandwidth

Increase bandwidth of access link to 10 Mbps

Consequences

- Can be costly
- utilization on LAN = 1%
- utilization on access link = 10%
- delay for each request

  ≈ 500msec+queueing delay (negligible)



Option-2: Adding a local cache

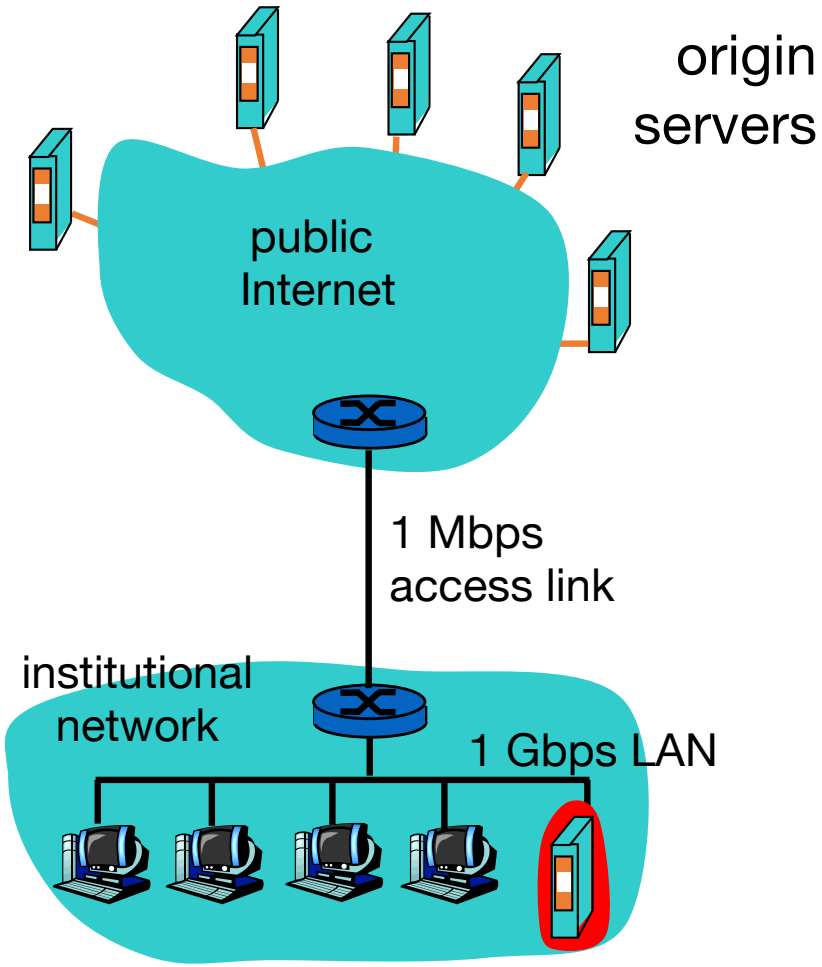
Consequences

- assume hit rate = 40% requests will be satisfied by the data in the cache (delay ≈ 10msec)
- Remaining 60% requests served by origin servers
- utilization of access link reduced to 60% → much smaller queueing delay (say 30 msec)
- avg delay for each object

  = 0.6x(Internet delay + access delay) + 0.4x(LAN+cache delay)

  = 0.6x(530 msec) + 0.4 x (1 msec)

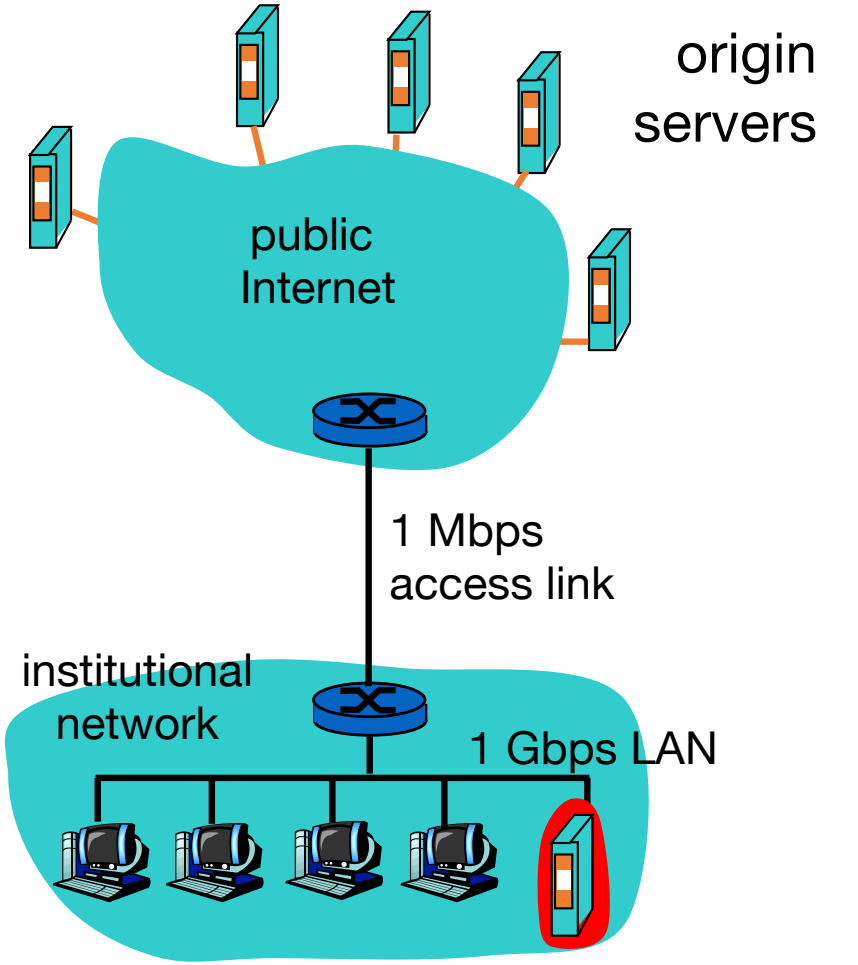
  = 313msec



Institutional cache

Having a local cache: there are issues

- Configuring browsers to use a local cache:
  - What if the local cache fails?
  - What if the browser moves?
- Two more questions
  1. What if the cached contents are obsolete?
  2. What about secured HTTP connections?

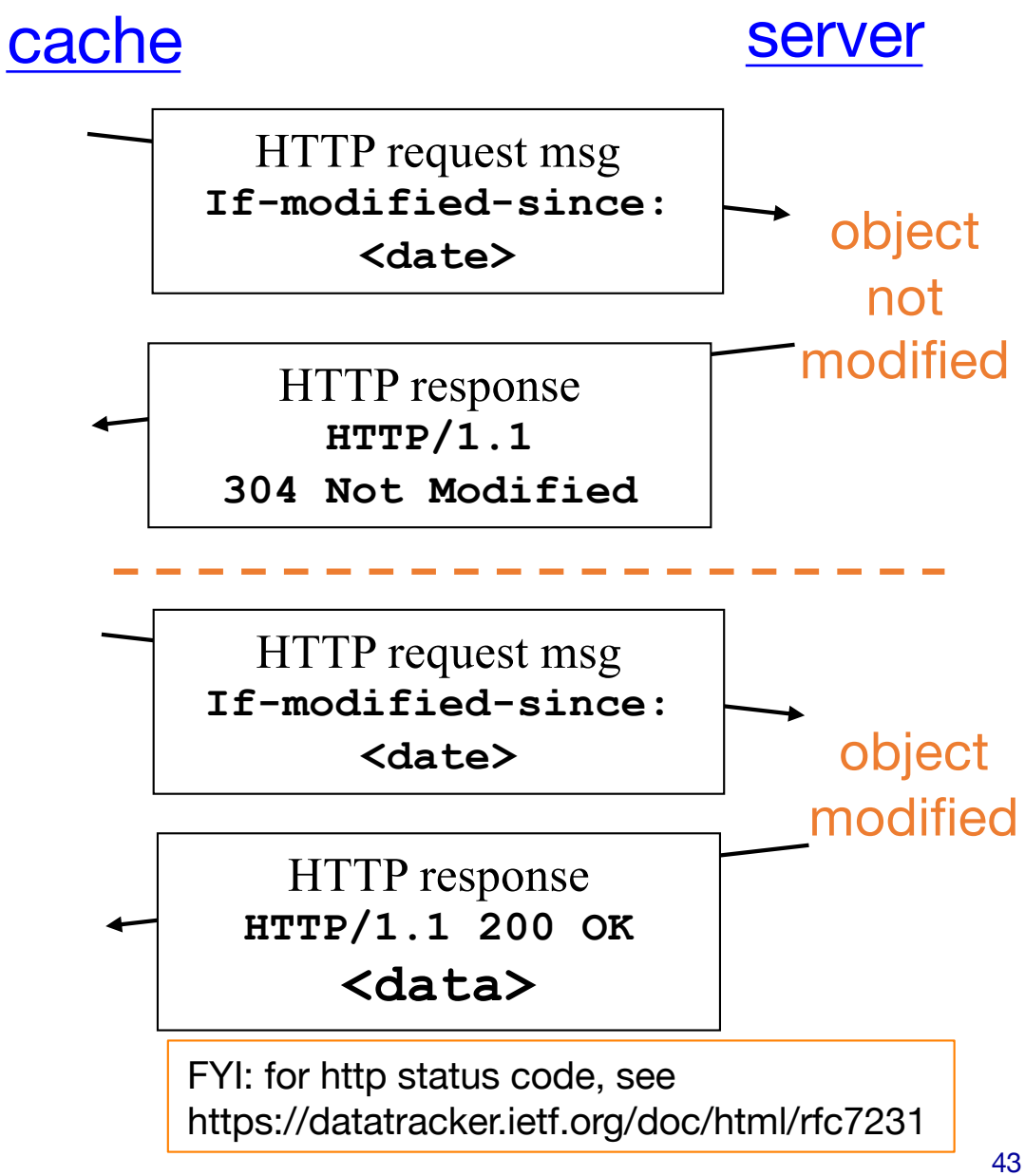


Answer to Q1: use HTTP Conditional GET

- Fetch content only if it has changed since previous fetch
- 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/1.1 304 Not Modified



Answer to Q2: HTTPS and HTTP Proxy

- HTTPS:
  - Browser connects to, and authenticates web server
  - Encrypt all communications between the two ends
- A cache by local ISP can run HTTPS with web server; your browser won't run HTTPS with the cache
- Web caching today: performed by Content Distribution Networks (CDNs)
  - CDN providers: Akamai, Fastly, CloudFlare, others
  - Your browser connects to a CDN server via HTTPs
    - Websites pay CDN providers and share crypto keys with them
    - *How* does your browser know *which CDN box to connect to?*

History HTTP

https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP

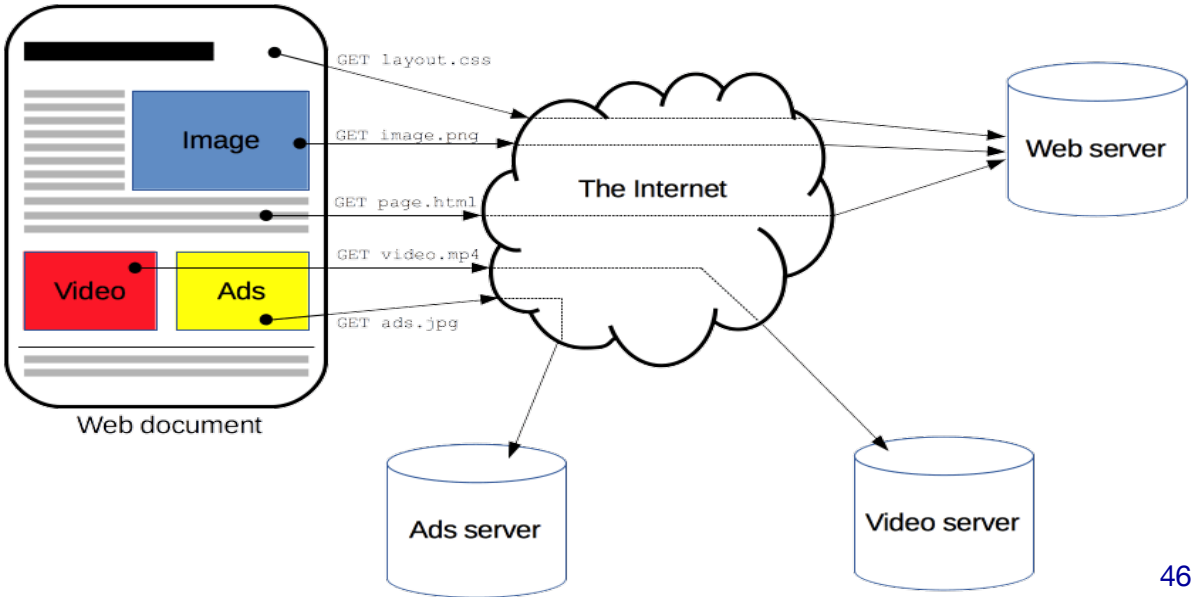
Sir Tim Berners-Lee 2016 ACM A.M. Turing Lecture, May 29, 2018
https://www.youtube.com/watch?v=BaMa4u4Fio4

Why HTTP/2

HTTP/1.1 with pipelining: not good enough

Some measurement numbers: an average Web application is

- composed of more than 90 objects
- fetched from more than 15 distinct hosts
- totaling more than 1.3MB of transferred data
  - On average each object <15KB
    - Some can be very big, e.g. large image files
    - That means some others can be very small

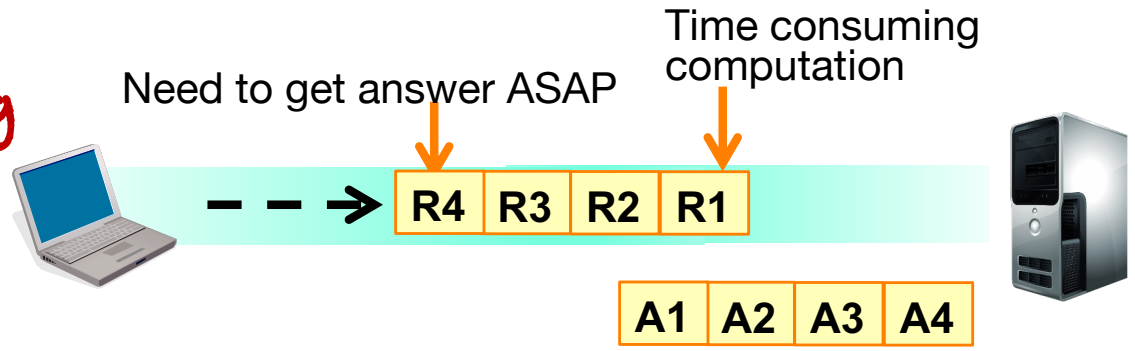


HTTP/1.1's performance issues

important

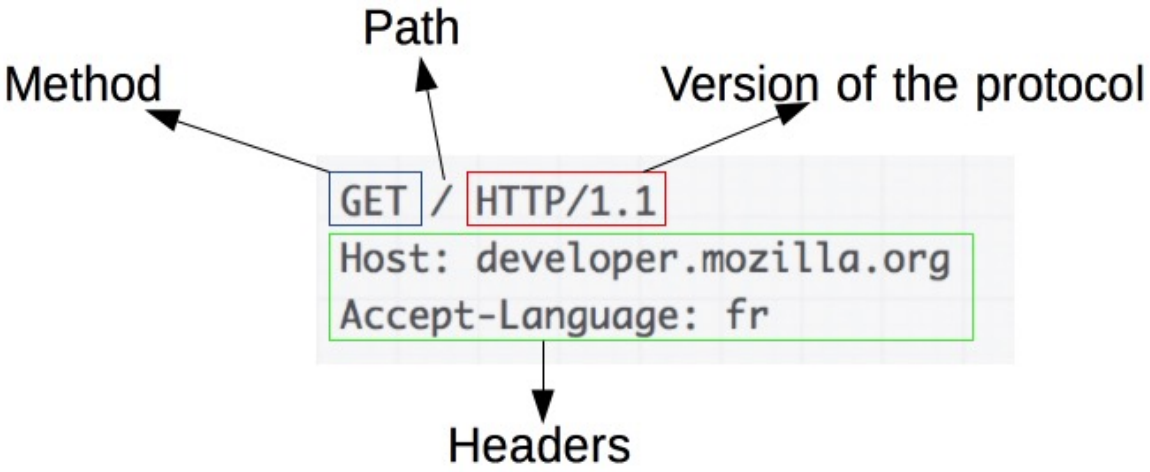
1. Head-of-line blocking: HTTP/1.1 handles all requests in strict sequential order
   - A request for a large file, or some dynamic computation, can take time, blocking all requests following it

Header-of-line blocking

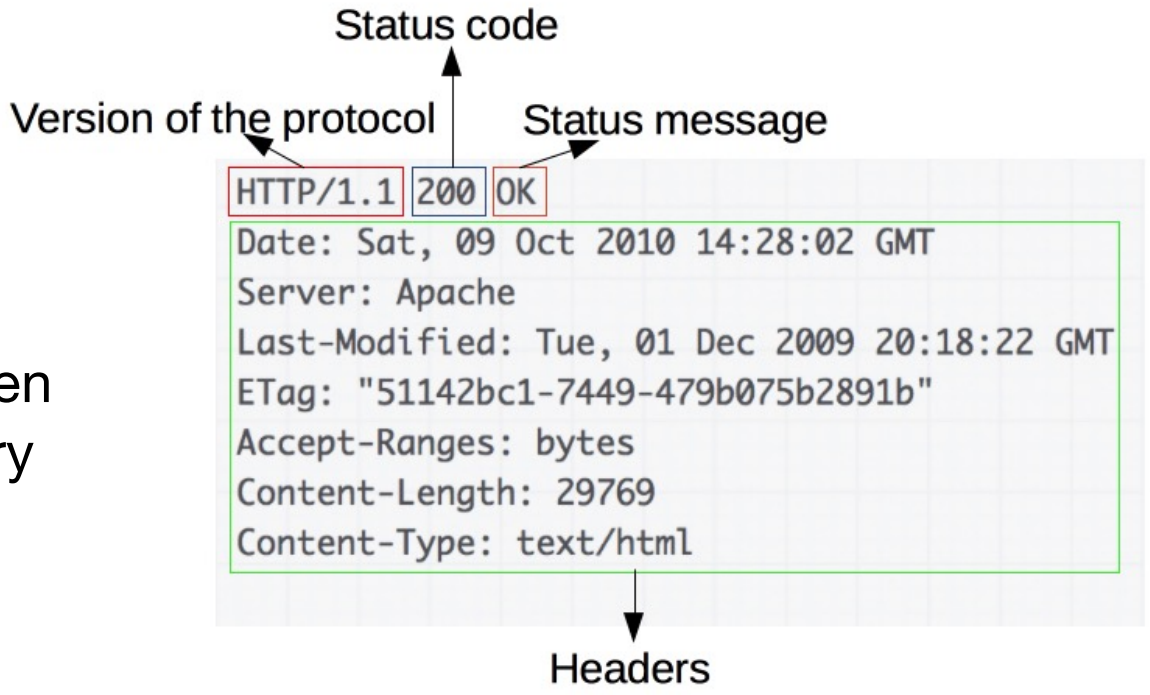


   - Work-around: open multiple TCP connections
2. Big size HTTP header with repetitive information carried in queries
   - No work-around

HTTP request and response message formats

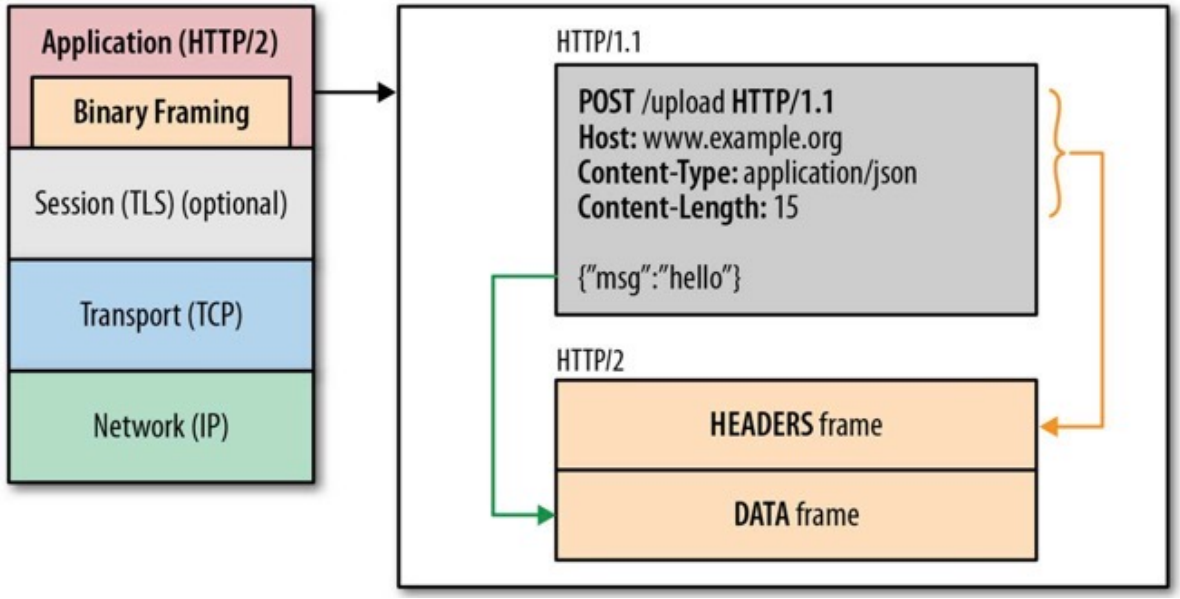


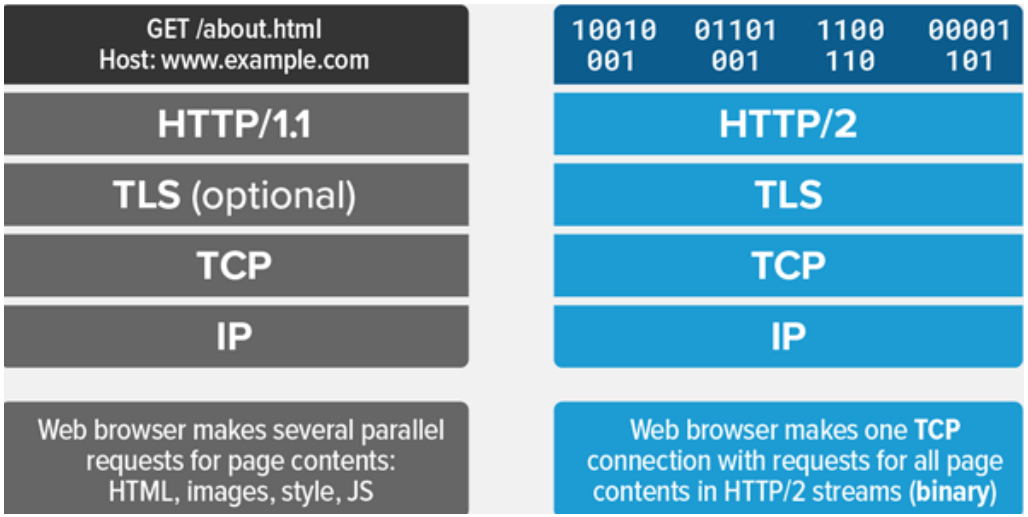
- HTTP headers are ASCII-encoded
- Requests/response between the same 2 end nodes carry repeated information



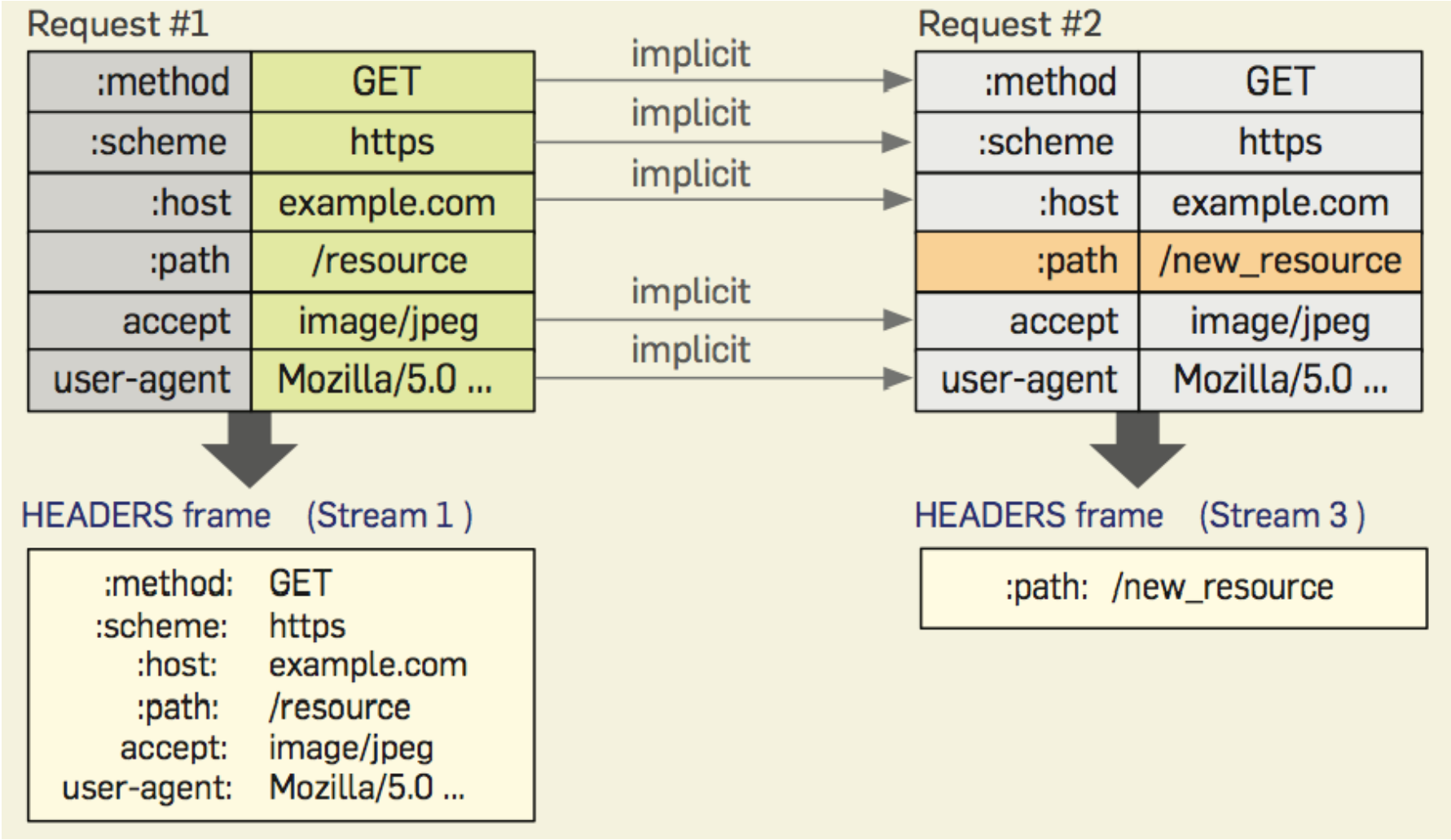
HTTP/2's major new features

- Binary encoding
- Header compression
- "frame" as the basic unit
- Use a single TCP connection between browser—server
  - Each HTTP request → a stream
  - streams are multiplexed, in priority order
- Server push





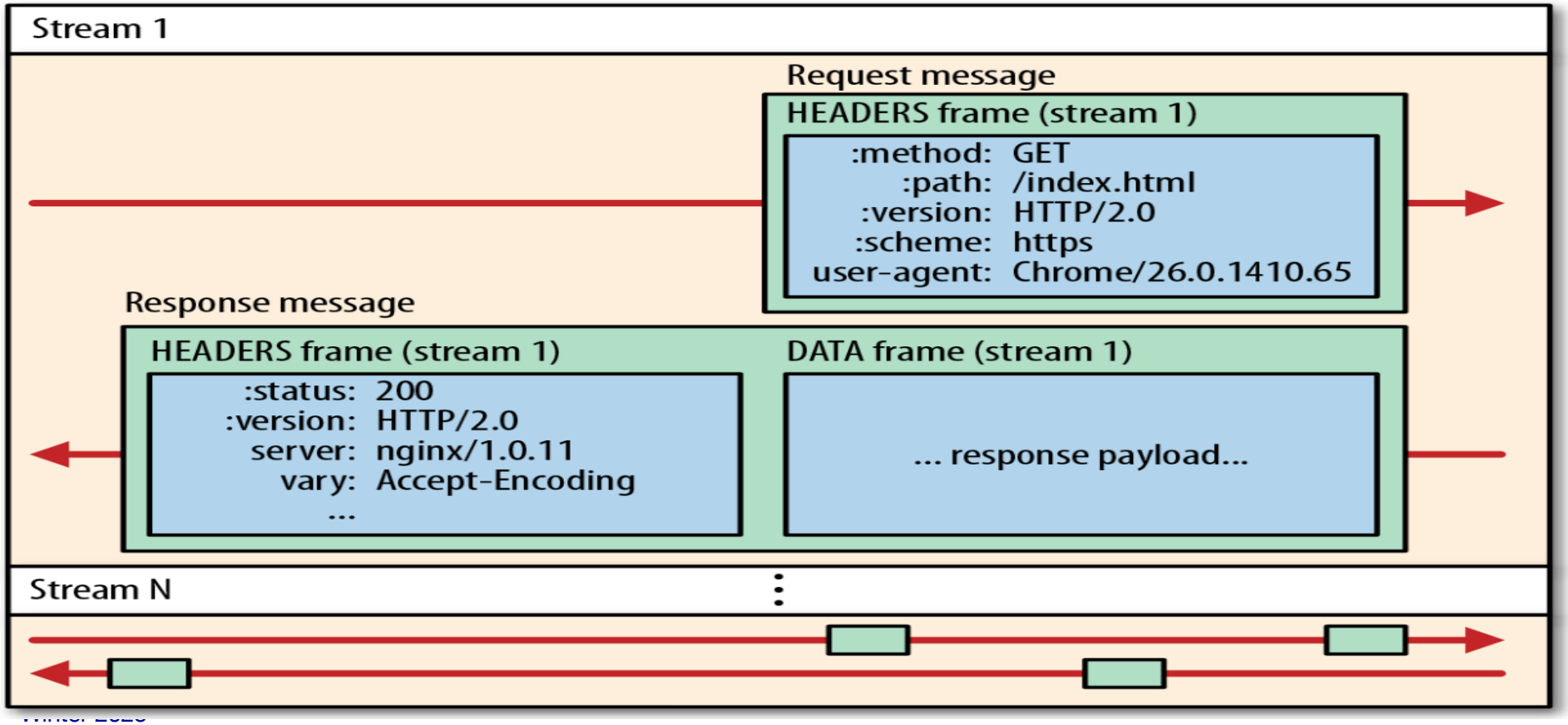
HTTP/2: Header Compression



- Both browser & server keep a header table until the TCP connection closes

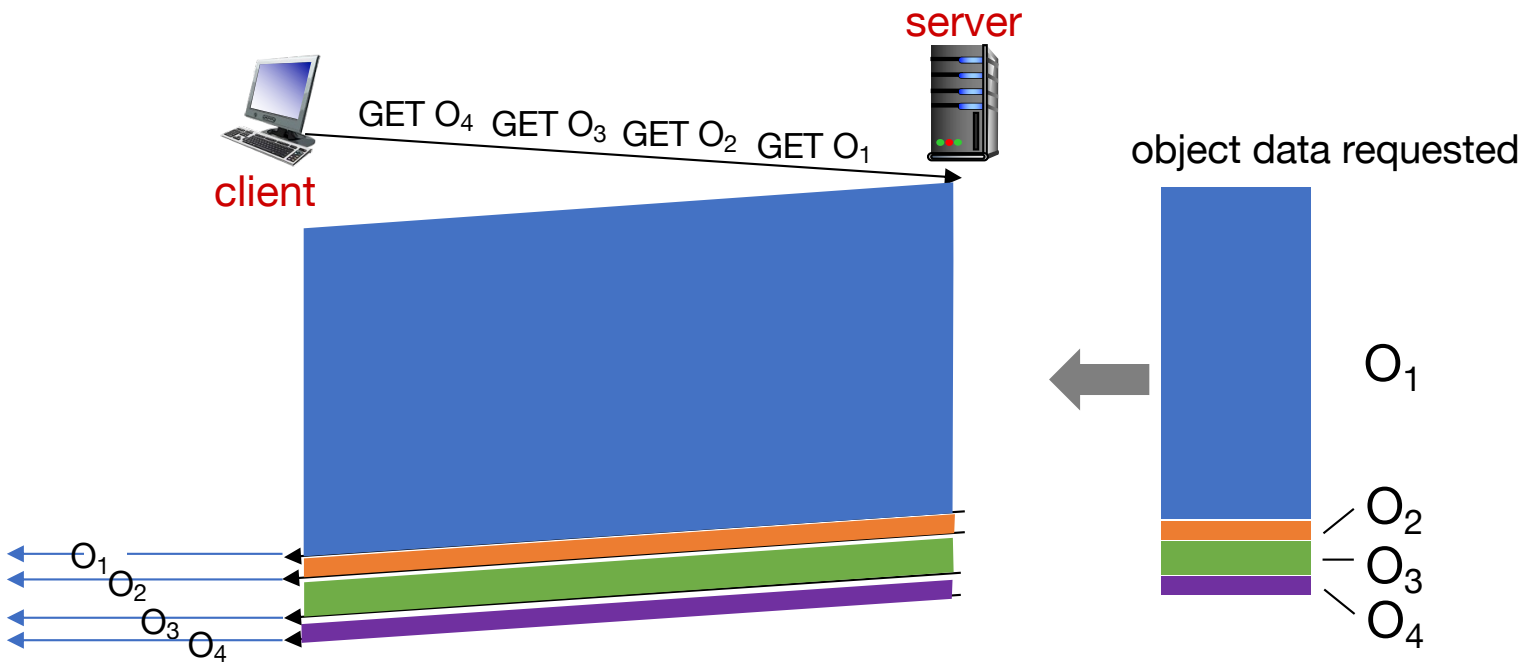
HTTP/2.0: Frame, Message, Stream

- Frame: basic communication unit
- Message: an HTTP request, or response
  - encoded in one or multiple frames
- Stream: a virtual channel with priority, carrying frames in both directions



HTTP/2: Mitigating HOL blocking

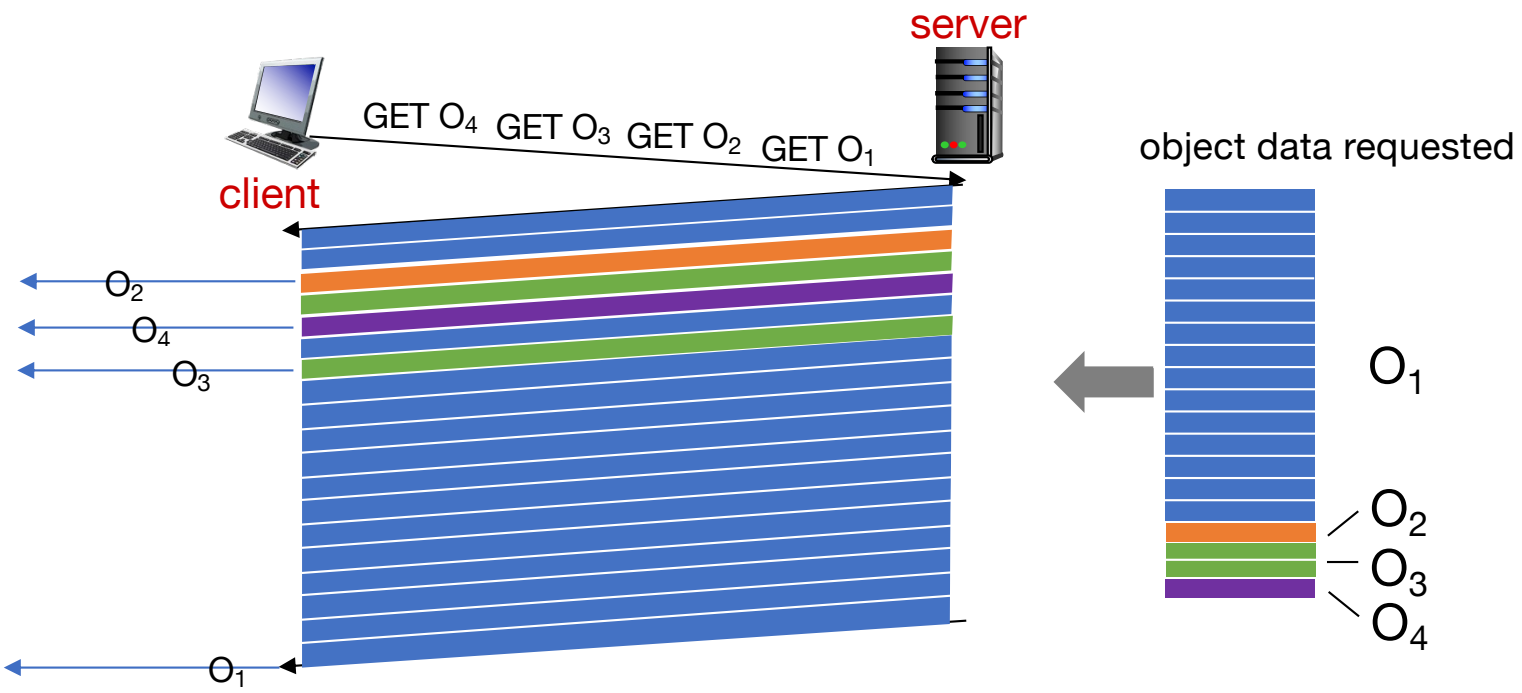
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



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_2, O_3, O_4 delivered quickly, O_1's finish-time slightly delayed

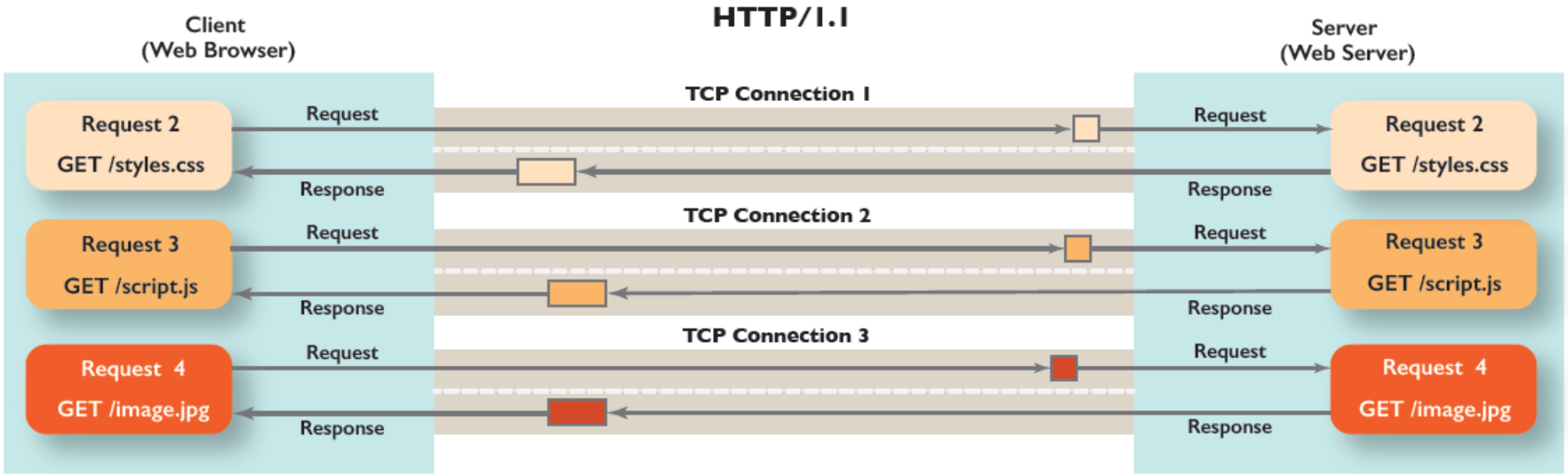
- What if 2nd frame of O_1 gets lost: can O_2, O_3, and O_4 be delivered to the browser app before the loss is recovered?

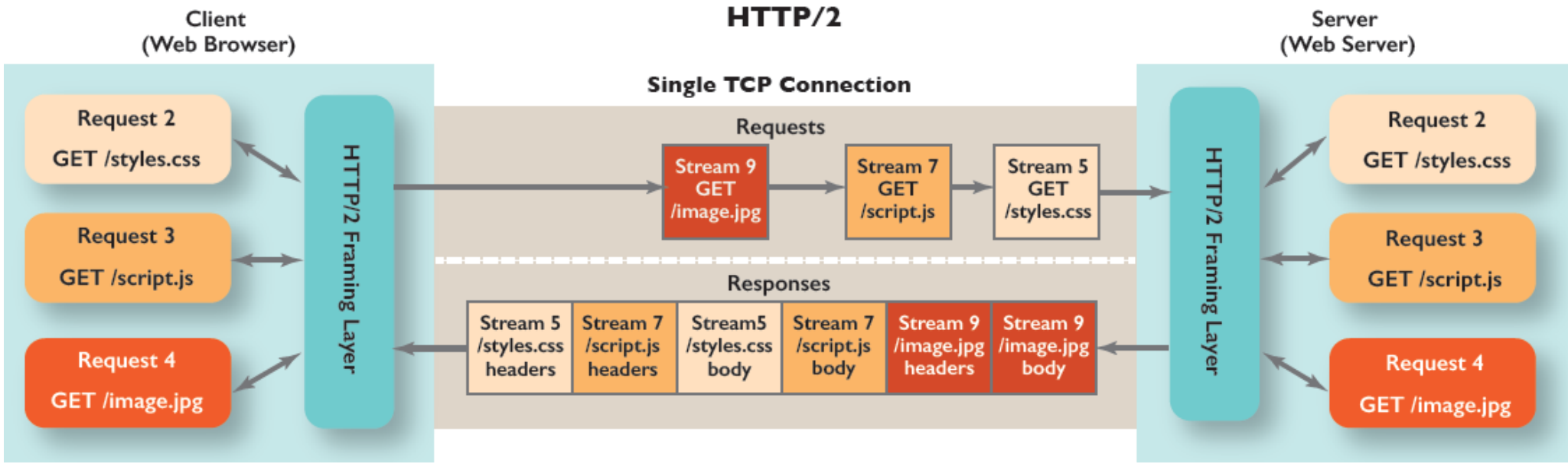
HTTP/2 Performance Improvements

- Reduced HTTP header overhead
  - Binary encoding
  - Header compression
- Attempted to remove head-of-line blocking
  - Multiple streams, one for each http request/reply
  - Big messages are broken down to multiple frames
  - Frames from all streams can be interleaved
- Above approaches avoids HOL *at HTTP level*
  - Single TCP connection between client-server → packet losses still lead to head-of-line blocking

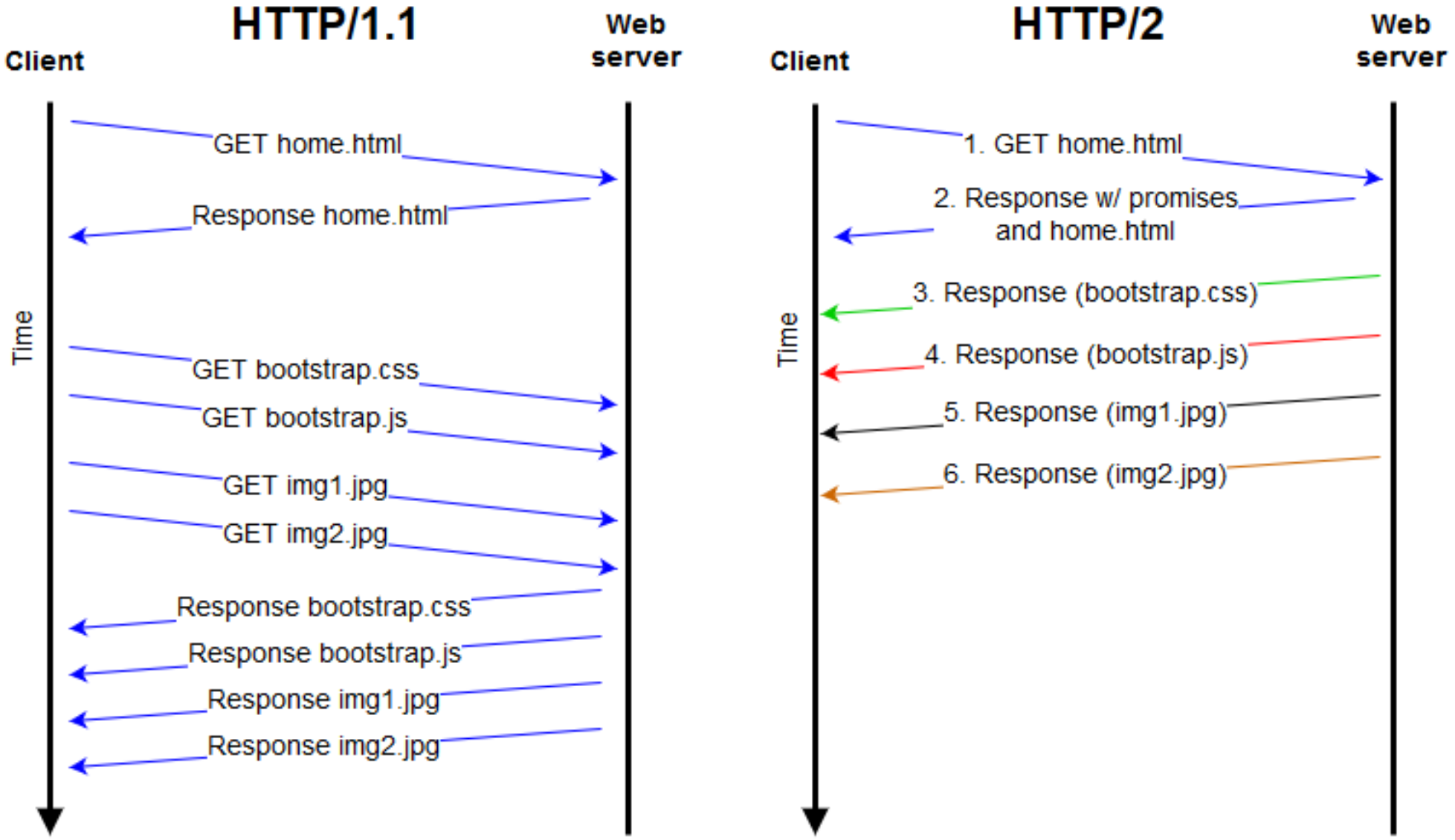
HTTP/2: a single TCP connection for multiple streams

- Streams are prioritized





HTTP/2 server push



HTTP/2 to HTTP/3

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