CS 158A Computer Networks

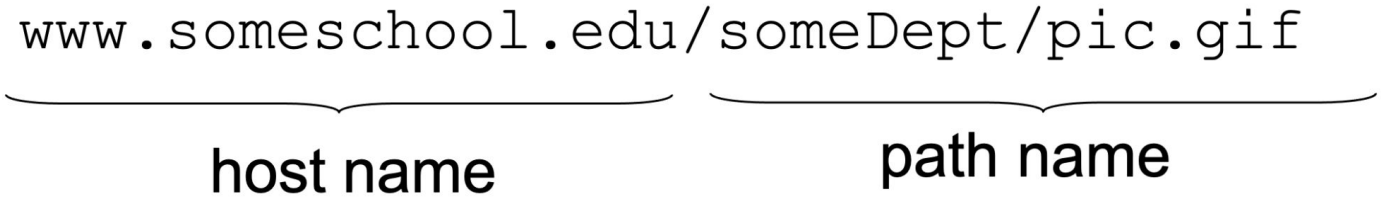
8. Web and HTTP

San José State University

Web and HTTP

First, a quick review…

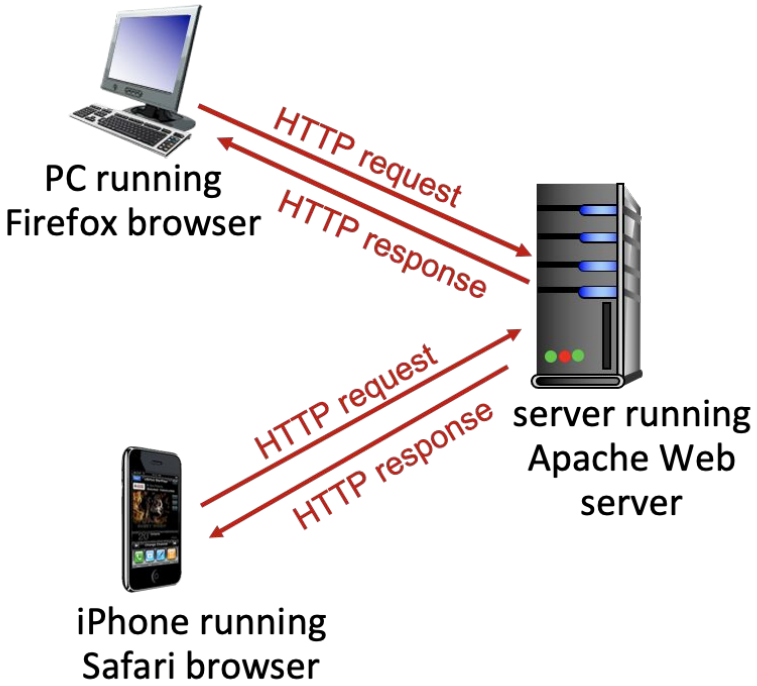
- web page consists of objects, each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of base HTML-file which includes several referenced objects, each addressable by a URL (Uniform Resource Locator), e.g.,



HTTP overview

HTTP: hypertext transfer protocol

- Web's application-layer protocol
- client/server model:
  - client: browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - server: Web server sends (using HTTP protocol) objects in response to requests



HTTP overview (continued)

- HTTP (/1, /2) 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 connections: two types

Non-persistent HTTP

- TCP connection opened
- at most one object sent over TCP connection
- TCP connection closed

E.g. downloading multiple objects required multiple connections

Persistent HTTP

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

Non-persistent HTTP: example

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

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

Non-persistent HTTP: example

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

4. HTTP server closes TCP connection.

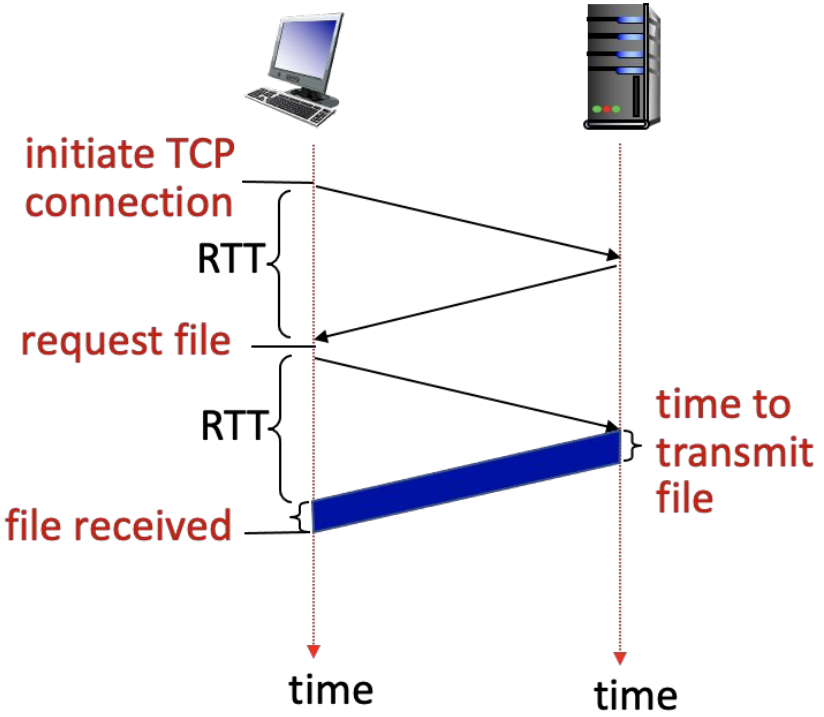
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

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



Persistent HTTP (HTTP 1.1)

- Non-persistent HTTP issues:
  - requires 2 RTTs per object
  - OS overhead for each TCP connection
  - browsers often open multiple parallel TCP connections to fetch referenced objects in parallel
- Persistent HTTP (HTTP1.1):
  - server leaves connection open after sending response
  - subsequent HTTP messages between same client/server sent over open connection
  - client sends requests as soon as it encounters a referenced object
  - as little as one RTT for all the referenced objects (cutting response time in half)

HTTP request message

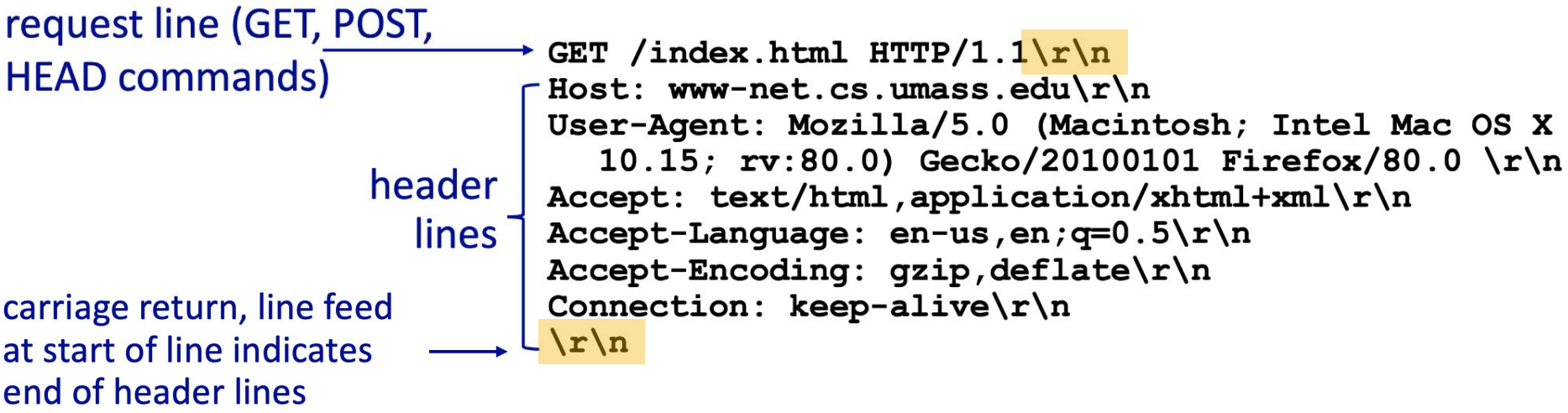
- two types of HTTP messages: request, response
- HTTP request message:
  - ASCII (human-readable format)

request line (GET, POST, HEAD commands) → `GET /index.html HTTP/1.1\r\n`

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

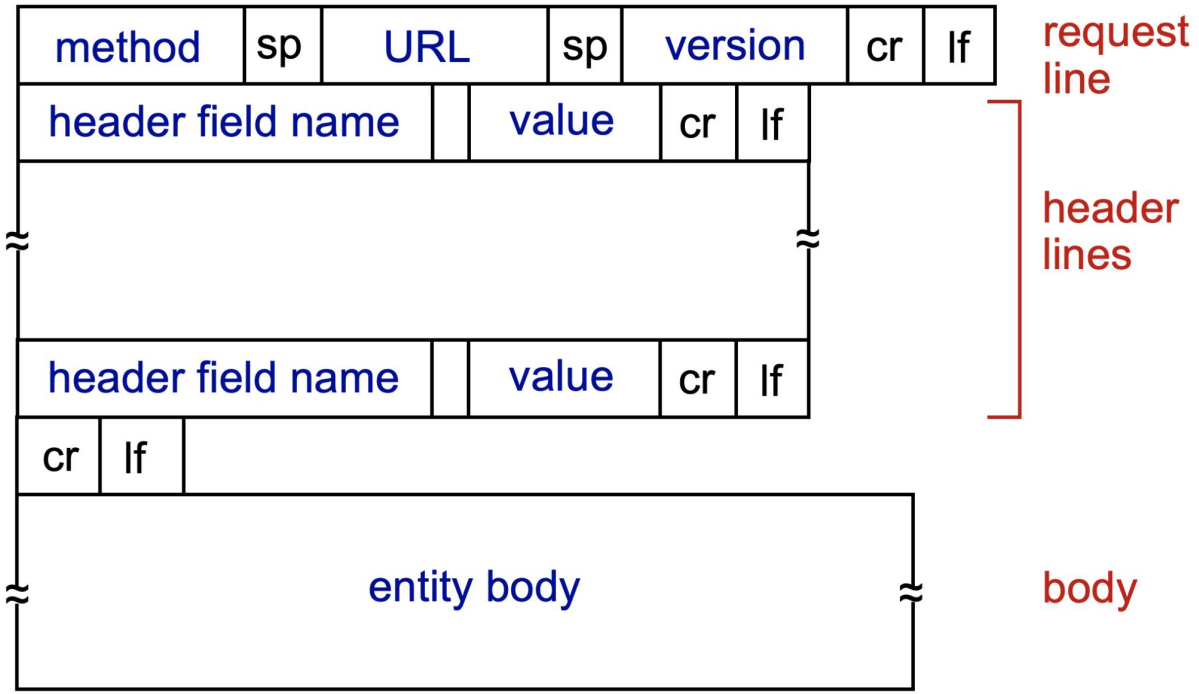
HTTP request message

- two types of HTTP messages: request, response
- HTTP request message:
  - ASCII (human-readable format)



* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP request message: general format

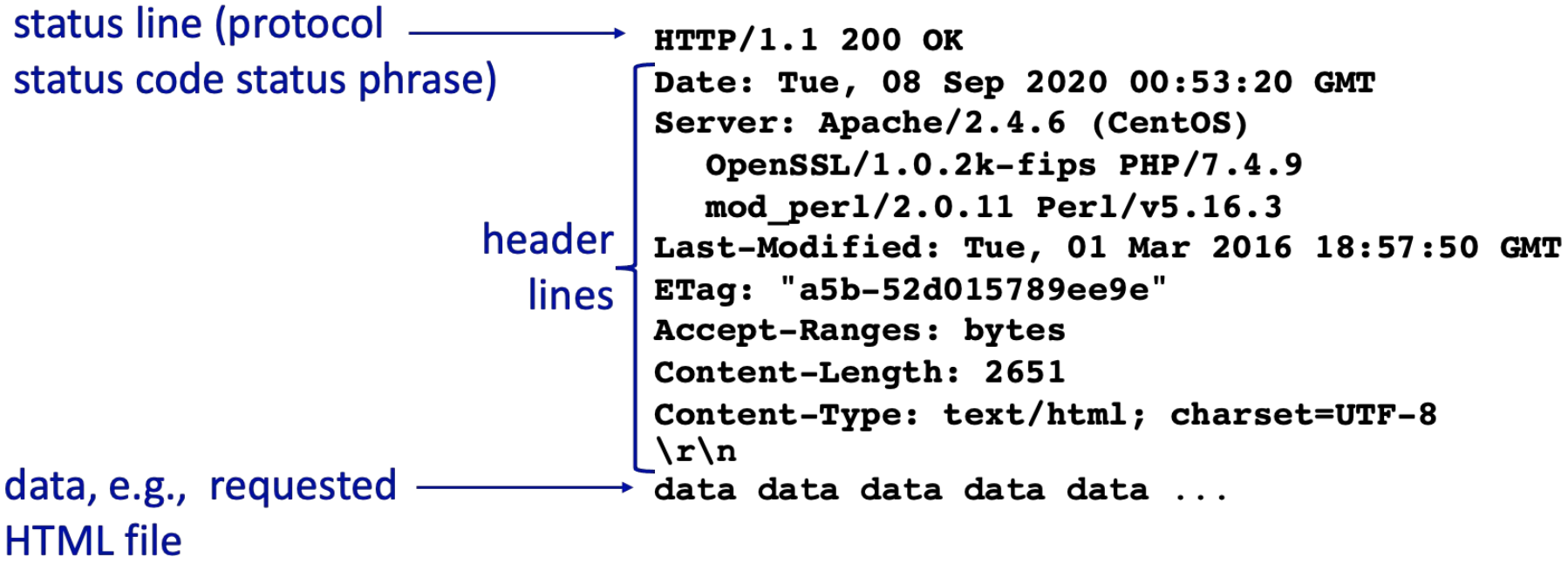


The request/status line and headers must all end with <CR><LF> (that is, a carriage return followed by a line feed). The empty line must consist of only <CR><LF> and no other whitespace.

Other HTTP request messages

- POST method:
  - web page often includes form input
  - user input sent from client to server in entity body of HTTP POST request message
- GET method (for sending data to server):
  - include user data in URL field of HTTP GET request message (following a '?'): **`www.somesite.com/animalsearch?monkeys&banana`**
- HEAD method:
  - requests headers (only) that would be returned if specified URL were requested with an HTTP GET method.
- PUT method:
  - uploads new file (object) to server
  - completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

HTTP response message



HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

  200 OK
  - request succeeded, requested object later in this message

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

  400 Bad Request
  - request msg not understood by server

  404 Not Found
  - requested document not found on this server

  505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

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

```
% nc -c -v gaia.cs.umass.edu 80
```

2. type in a GET HTTP request:
    - by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

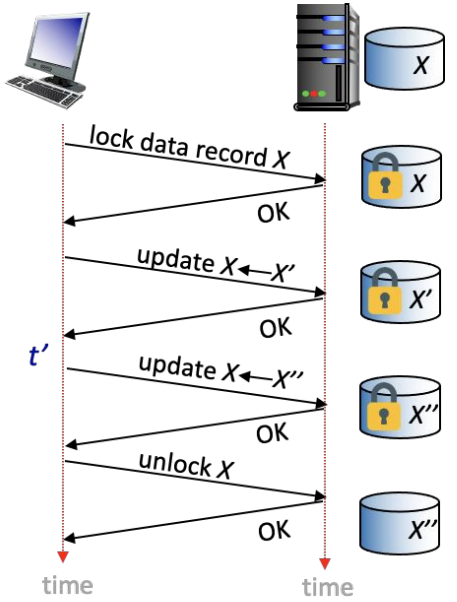
```
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

3. look at response message sent by HTTP server!
(or use Wireshark to look at captured HTTP request/response)

Maintaining user/server state: cookies

- Recall: HTTP GET/response interaction is stateless
- no notion of multi-step exchanges of HTTP messages to complete a Web "transaction"
  - no need for client/server to track "state" of multi-step exchange
  - all HTTP requests are independent of each other
  - no need for client/server to "recover" from a partially-completed-but-never-completely-completed transaction

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



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

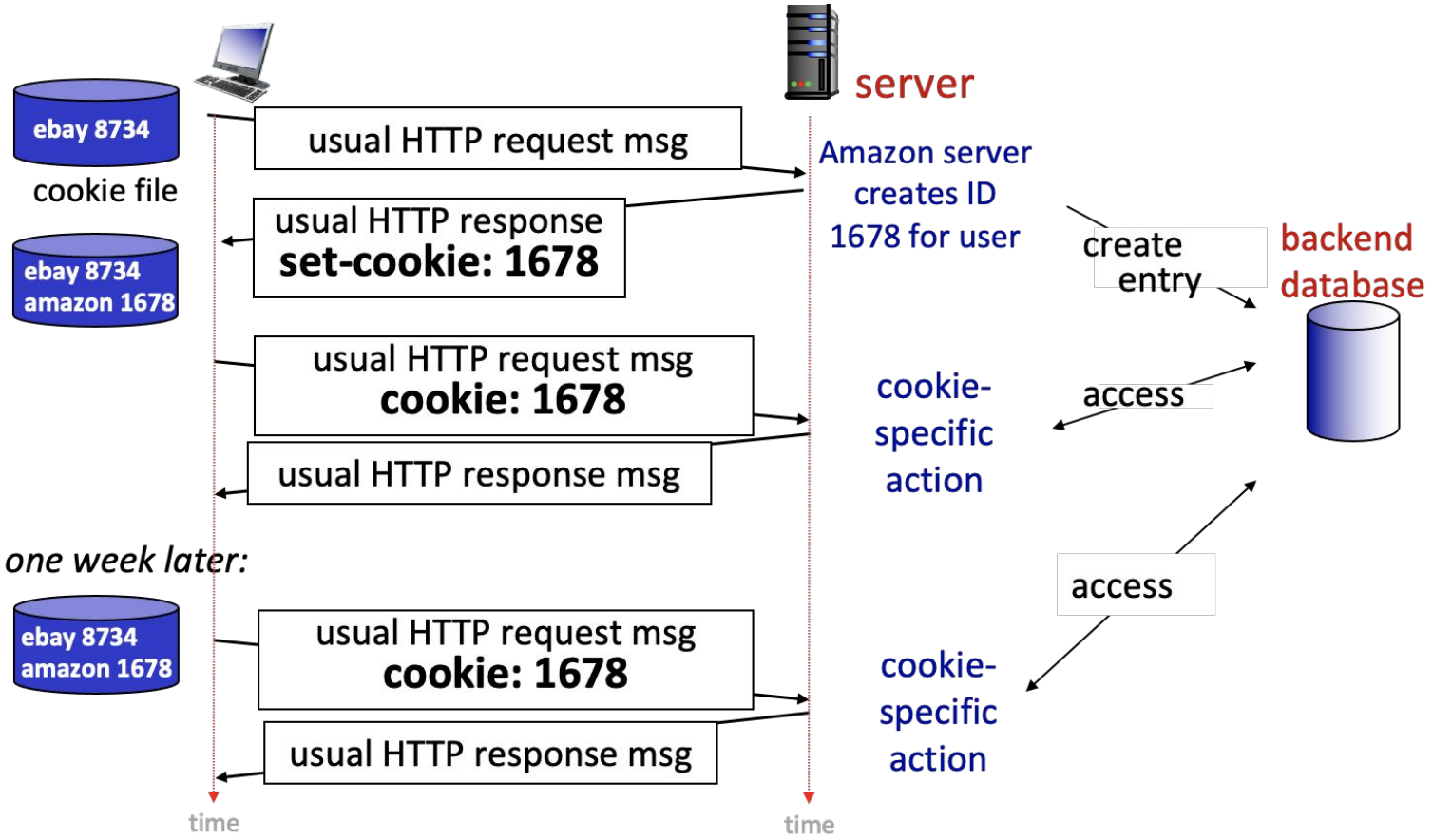
Maintaining user/server state: cookies

- Web sites and client browser use cookies to maintain some state between transactions
- four components:
  1. cookie header line of HTTP response message
  2. cookie header line in next HTTP request message
  3. cookie file kept on user's host, managed by user's browser
  4. back-end database at Web site

Maintaining user/server state: cookies example

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

Maintaining user/server state: cookies

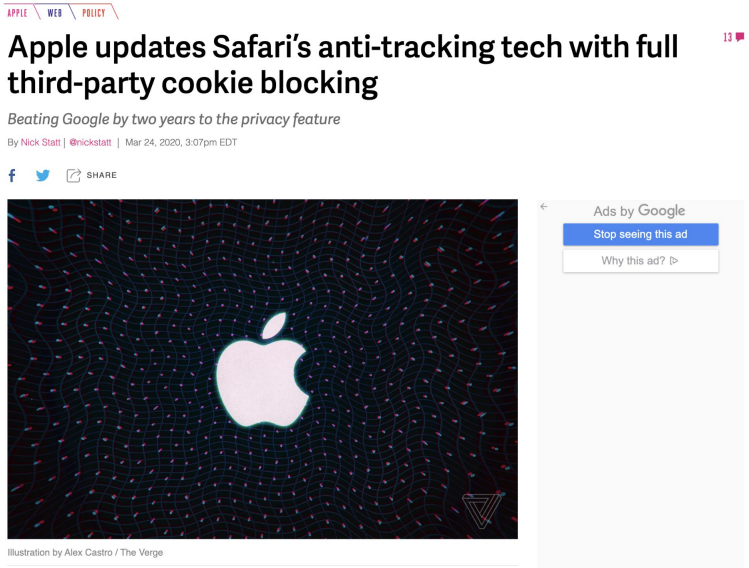


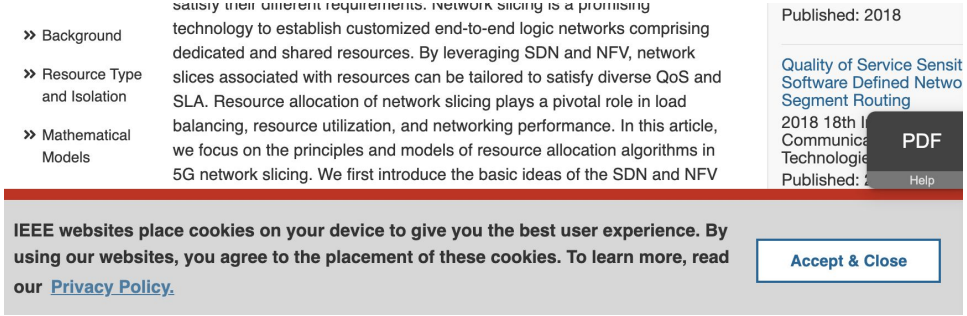
HTTP cookies: comments

- What cookies can be used for:
  - authorization
  - shopping carts
  - recommendations
  - user session state (Web e-mail)
- Challenge: How to keep state?
  - at protocol endpoints: maintain state at sender/receiver over multiple transactions
  - in messages: cookies in HTTP messages carry state

Cookies and Privacy

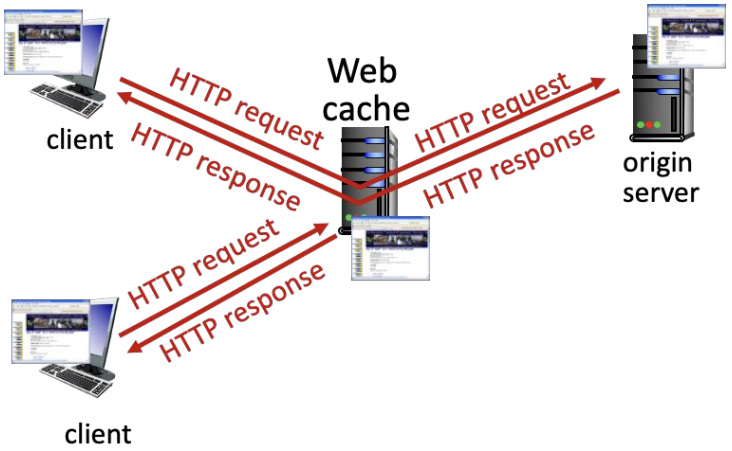
- cookies permit sites to learn a lot about you on their site
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites





Web caches

- Goal: satisfy client requests without involving origin server
  - user configures browser to point to a (local) Web cache
  - browser sends all HTTP requests to cache
    - if object in cache: cache returns object to client
    - else cache requests object from origin server, caches received object, then returns object to client



Web caches (aka proxy servers)

- Web cache acts as both client and server
  - server for original requesting client
  - client to origin server
- server tells cache about object's allowable caching in response header:

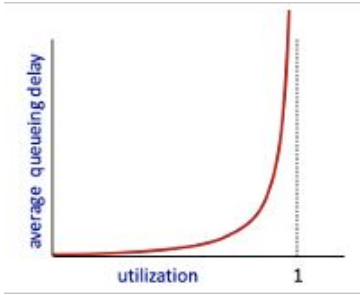
```
Cache-Control: max-age=<seconds>
```

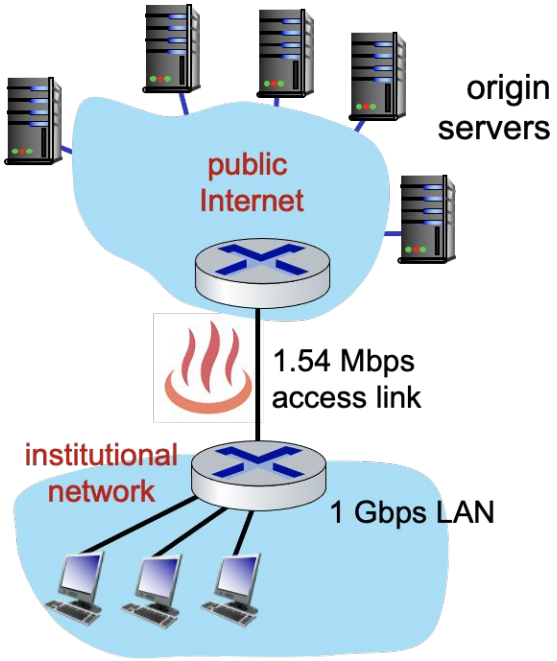
```
Cache-Control: no-cache
```

Why Web caching?

- reduce response time for client request
  - cache is closer to client
- reduce traffic on an institution’s access link
- Internet is dense with caches
  - enables “poor” content providers to more effectively deliver content

Caching example





Scenario:

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

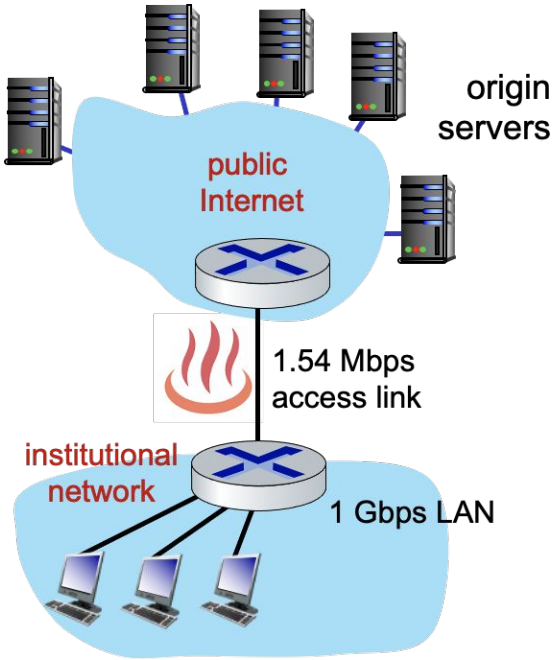
Performance:

- access link utilization = .97
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs

Option 1: buy a faster access link

Scenario:

- access link rate: 1.54 Mbps → 154Mbs
- RTT from institutional router to server: 2 sec
- web object size: 100K bits
- average request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps



Performance:

- access link utilization = .97 → .0097
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs → msecs

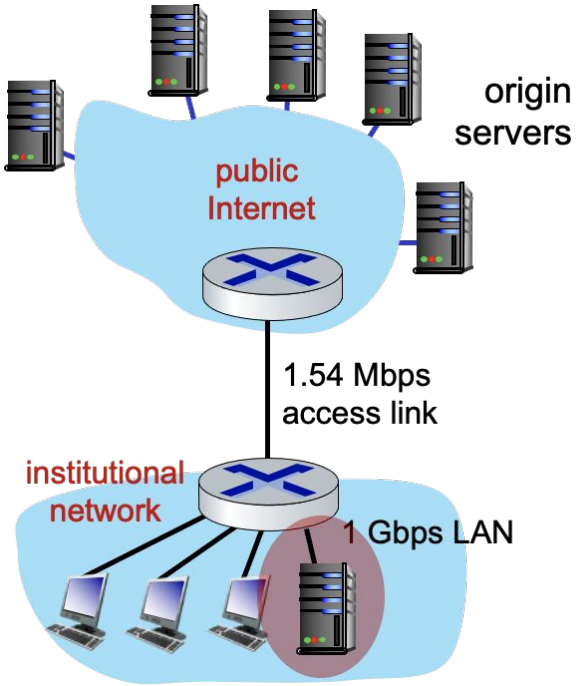
Option 2: install a web cache

Faster link: Expensive!

How about web cache? (cheap!)

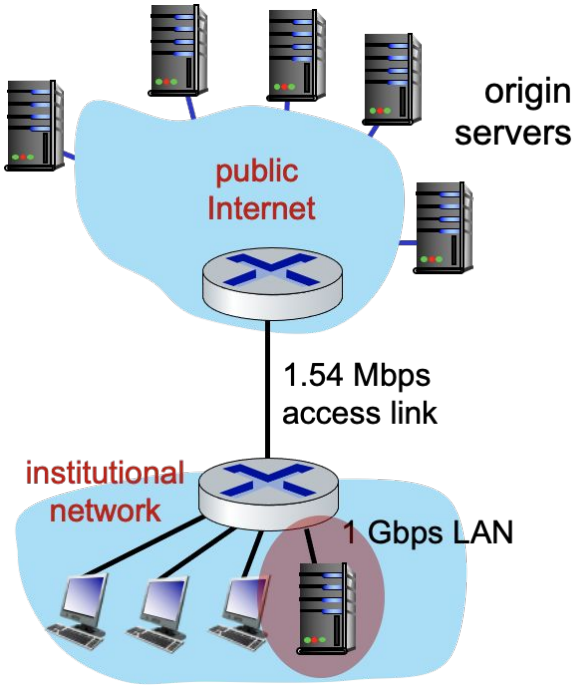
Performance:

- LAN utilization: .?
- access link utilization = ?
- average end-end delay = ?



Calculating performance with cache

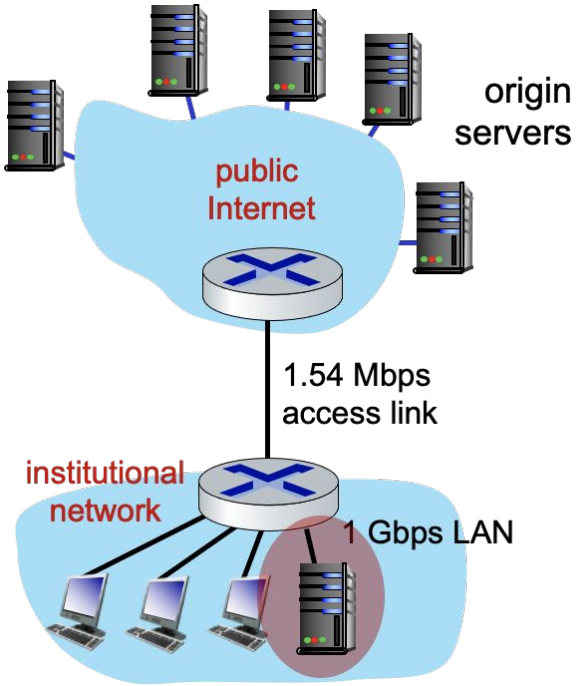
- suppose cache hit rate is 0.4:
  - 40% requests served by cache, with low (msec) delay
  - 60% requests satisfied at origin
- rate to browsers over access link
  = 0.6 * 1.54 Mbps = .9 Mbps
- access link utilization = 0.9/1.54
  = .58 means low (msec) queueing delay at access link
- average end-end delay:
  - = 0.6 * (delay from origin servers)
    + 0.4 * (delay when satisfied at cache)



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

Calculating performance with cache

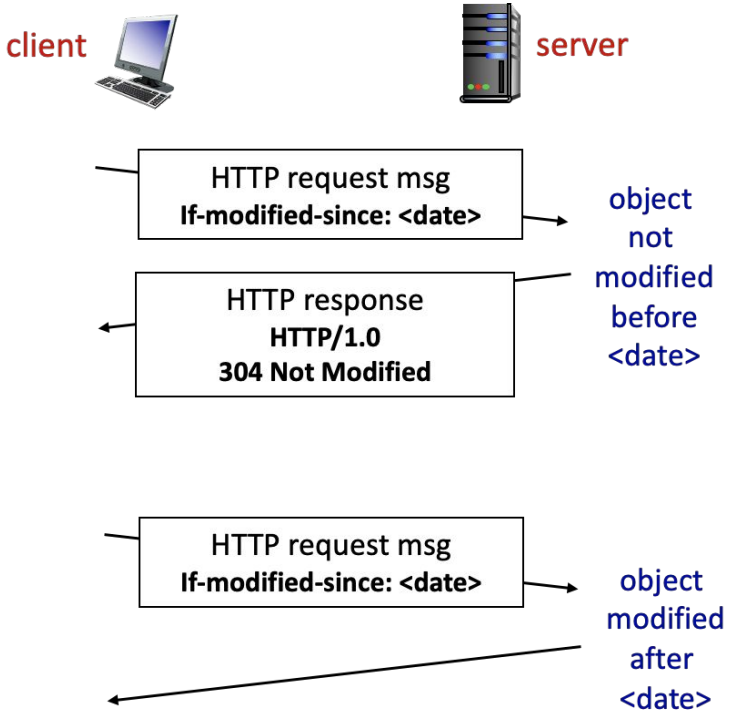
- suppose cache hit rate is 0.4:
  - 40% requests served by cache, with low (msec) delay
  - 60% requests satisfied at origin
- rate to browsers over access link
  = 0.6 * 1.54 Mbps = .9 Mbps
- access link utilization = 0.9/1.54
  = .58 means low (msec) queueing delay at access link
- average end-end delay:
  - = 0.6 * (delay from origin servers)
    + 0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs



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

Conditional GET

- Goal: don't send object if cache has up-to-date cached version
  - no object transmission delay (or use of network resources)
- client: specify date of cached copy in HTTP request
  - `If-modified-since: <date>`
- server: response contains no object if cached copy is up-to-date:
  - `HTTP/1.0 304 Not Modified`



HTTP/2

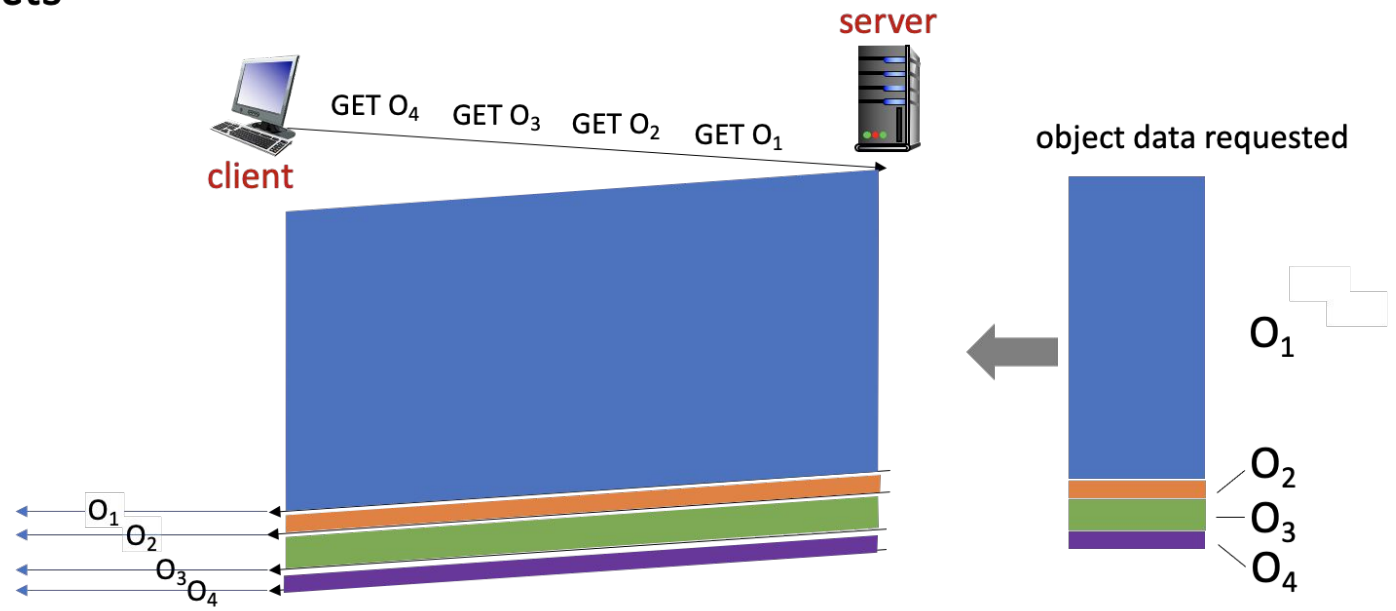
- Key goal: decreased delay in multi-object HTTP requests
- HTTP1.1: introduced multiple, pipelined GETs over single TCP connection
  - server responds in-order (FCFS: first-come-first-served scheduling) to GET requests
  - with FCFS, small object may have to wait for transmission
    - head-of-line (HOL) blocking behind large object(s)
  - loss recovery (retransmitting lost TCP segments) stalls object transmission

HTTP/2

- Key goal: decreased delay in multi-object HTTP requests
- HTTP/2: [RFC 7540, 2015] increased flexibility at server in sending objects to client:
  - methods, status codes, most header fields unchanged from HTTP 1.1
  - transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
    - divide objects into frames, schedule frames to mitigate HOL blocking
  - push unrequested objects to client

HTTP/2: mitigating HOL blocking

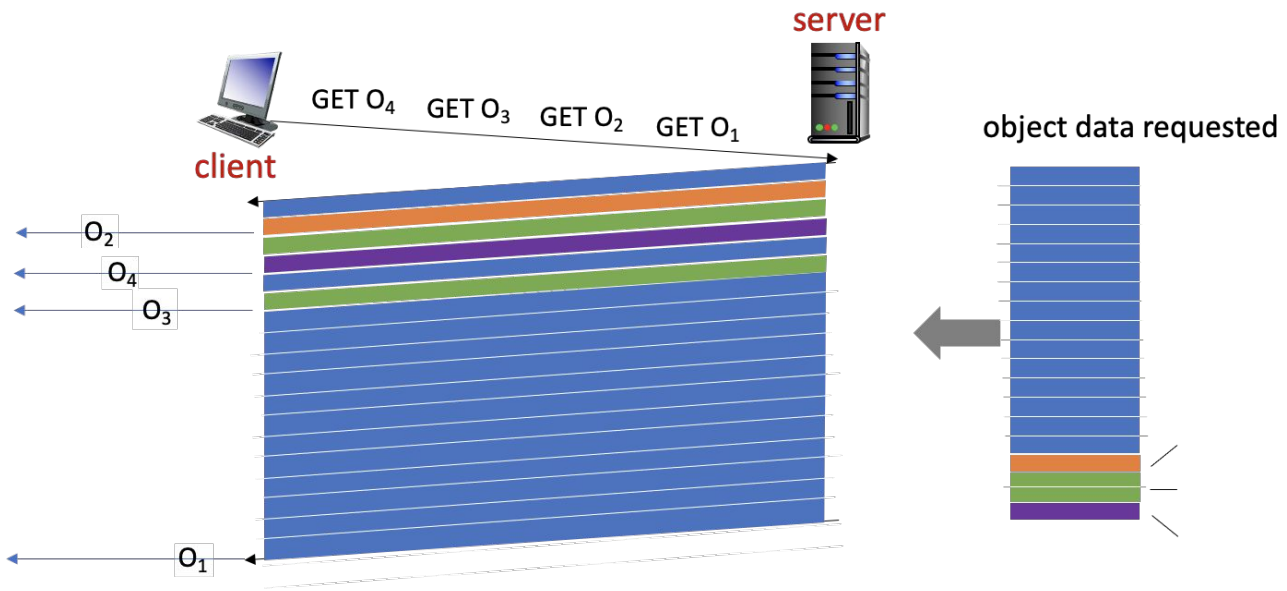
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



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

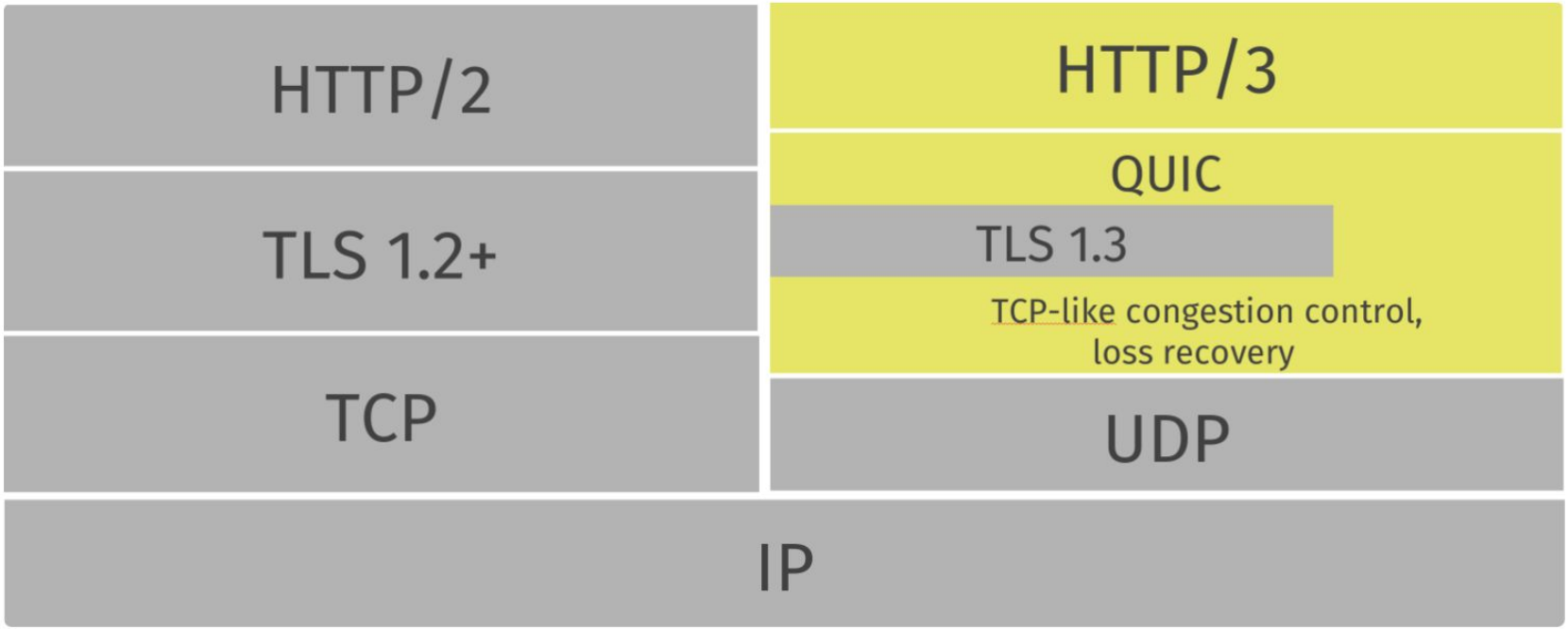


O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP/2 to HTTP/3

- 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
- HTTP/3: adds security, per object error- and congestion-control (more pipelining) over UDP
  - more on HTTP/3 in transport layer

HTTP/3 + QUIC Protocol features



HTTP/3 over QUIC stack overview

https://http3-explained.haxx.se/en/the-protocol

Connection ID

- Each connection possesses a set of connection identifiers, or connection IDs
- Connection IDs are independently selected by endpoints; each endpoint selects the connection IDs that its peer uses
- The primary function of these connection IDs is to ensure that changes in addressing at lower protocol layers (UDP, IP, and below) do not cause packets for a QUIC connection to be delivered to the wrong endpoint
- By taking advantage of the connection ID, connections can thus migrate between IP addresses and network interfaces in ways TCP never could.
- For instance, migration allows an in-progress download to move from a cellular network connection to a faster wifi connection when the user moves their device into a location offering wifi