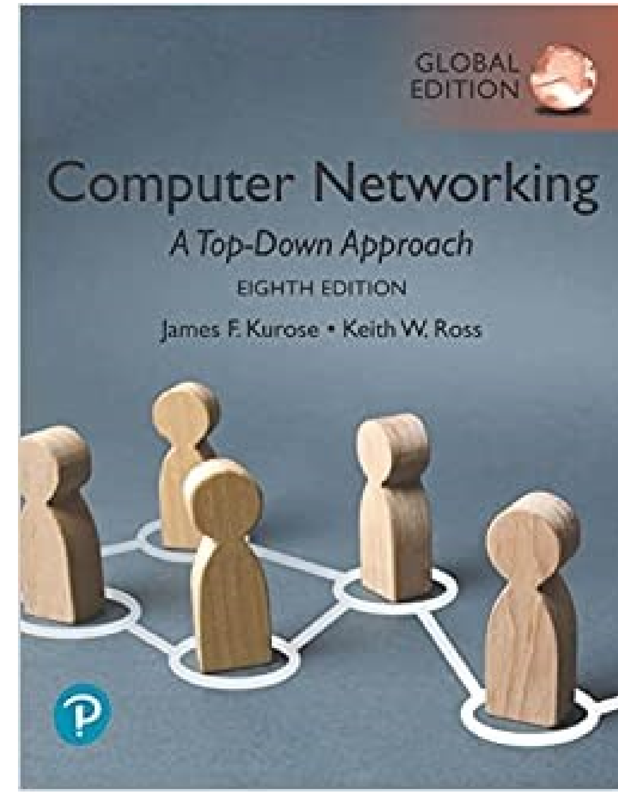
Chapter 2
Application Layer – part 1

School of Computing
Gachon Univ.
Joon Yoo



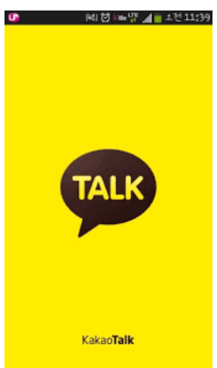
Computer Networking: A Top-Down Approach
8th edition (Global edition)
Jim Kurose, Keith Ross
Pearson, 2021

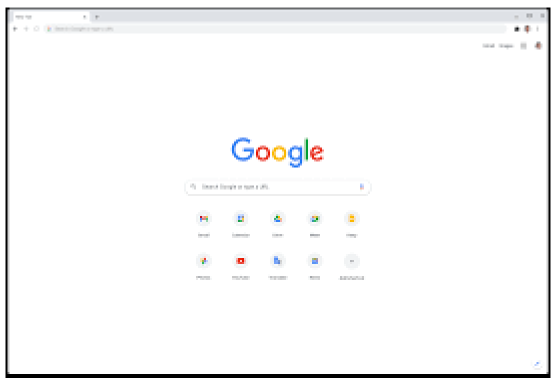
Many slides from J.F Kurose and K.W. Ross

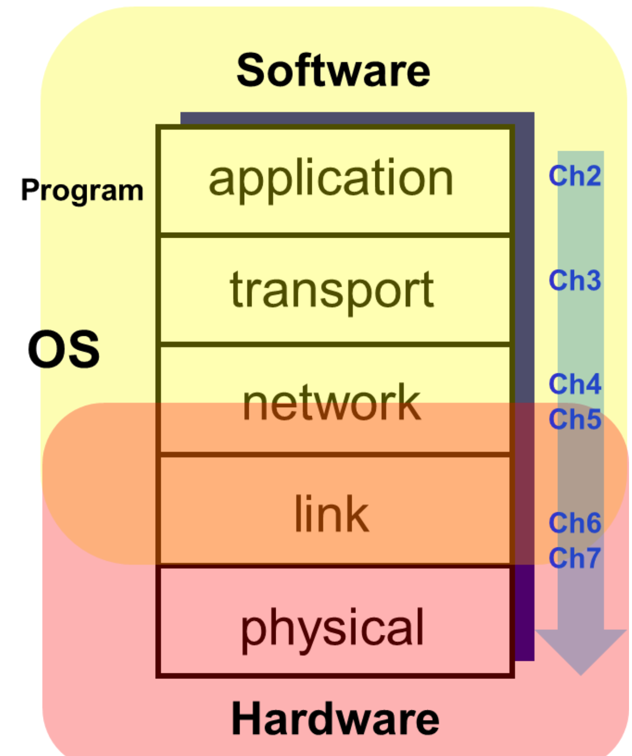
가천대학교
Gachon University

Question

Network Application vs. Application Layer?







Chapter 2: outline

Chapter 2: application layer

our goals:

- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - SMTP / POP3 / IMAP
  - DNS
- creating network applications
  - socket API

Some network apps

- social networking
- **Web (HTTP)**
- text messaging
- **e-mail (SMTP)**
- multi-user network games
- streaming stored video: YouTube, Netflix **(DASH)**
- P2P file sharing
- voice over IP (e.g., Skype)
- real-time video conferencing
- Internet search
- IP address search **(DNS)**
- remote login
- …

Q: your favorites?

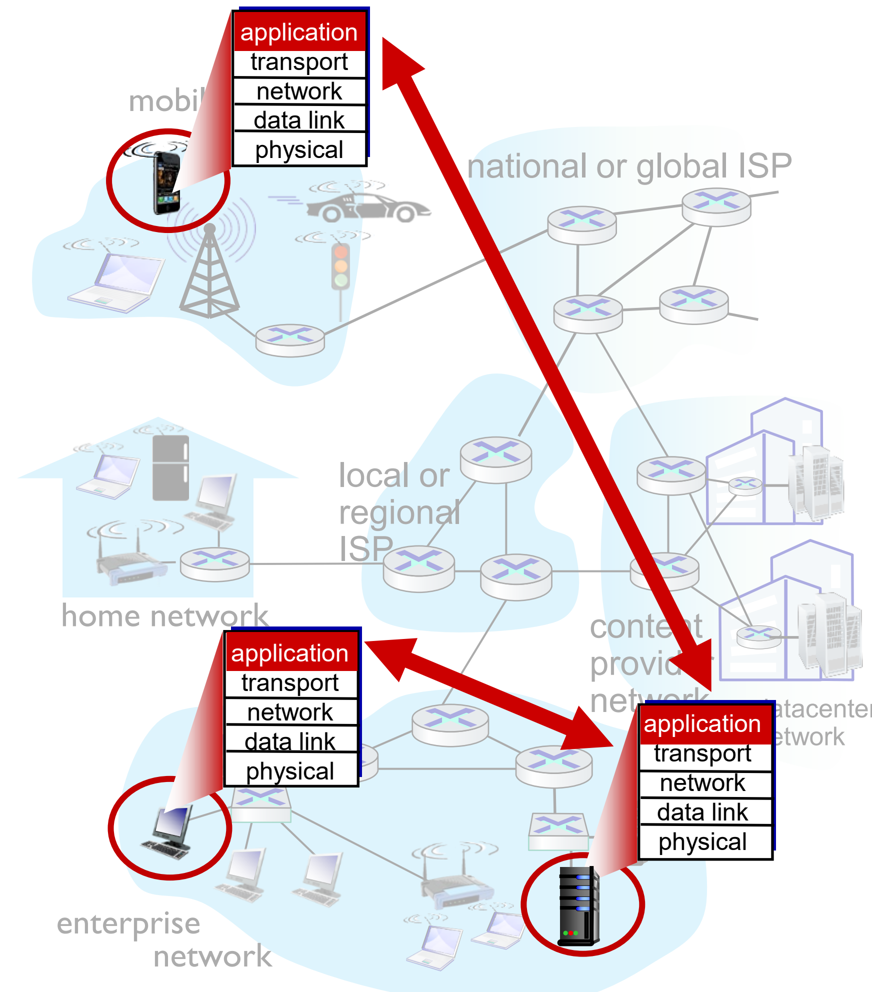
Creating a network app

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

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



Client-server paradigm

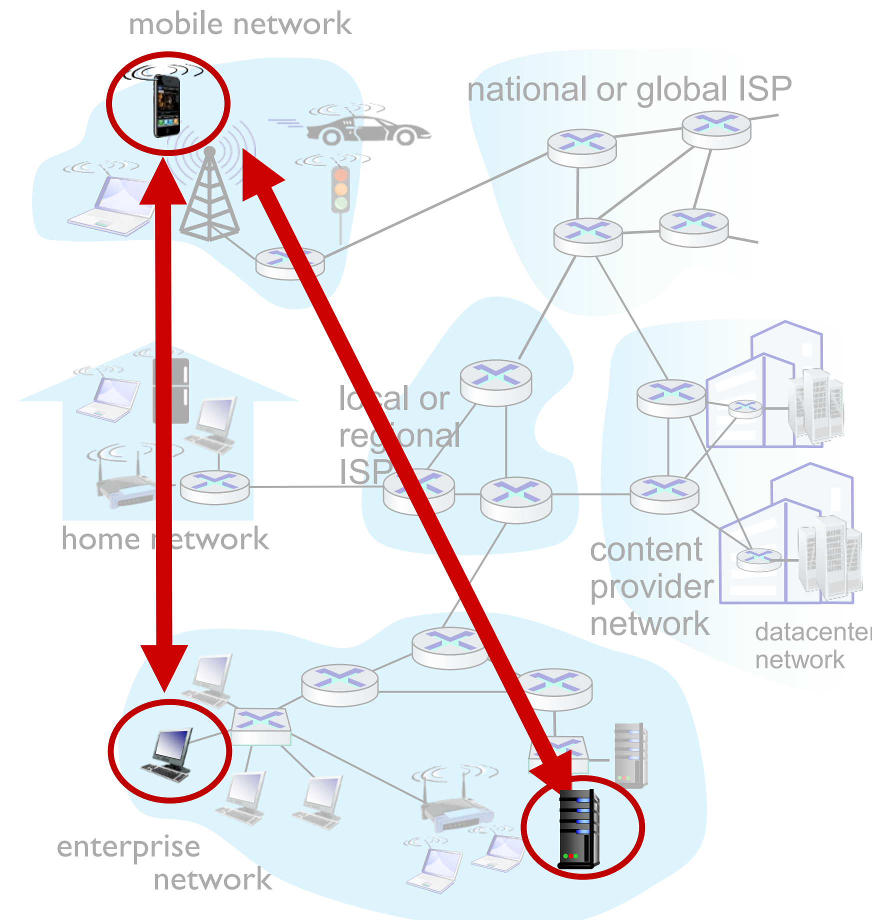
server:

- always-on host
- permanent IP address
- often in data centers, for scaling

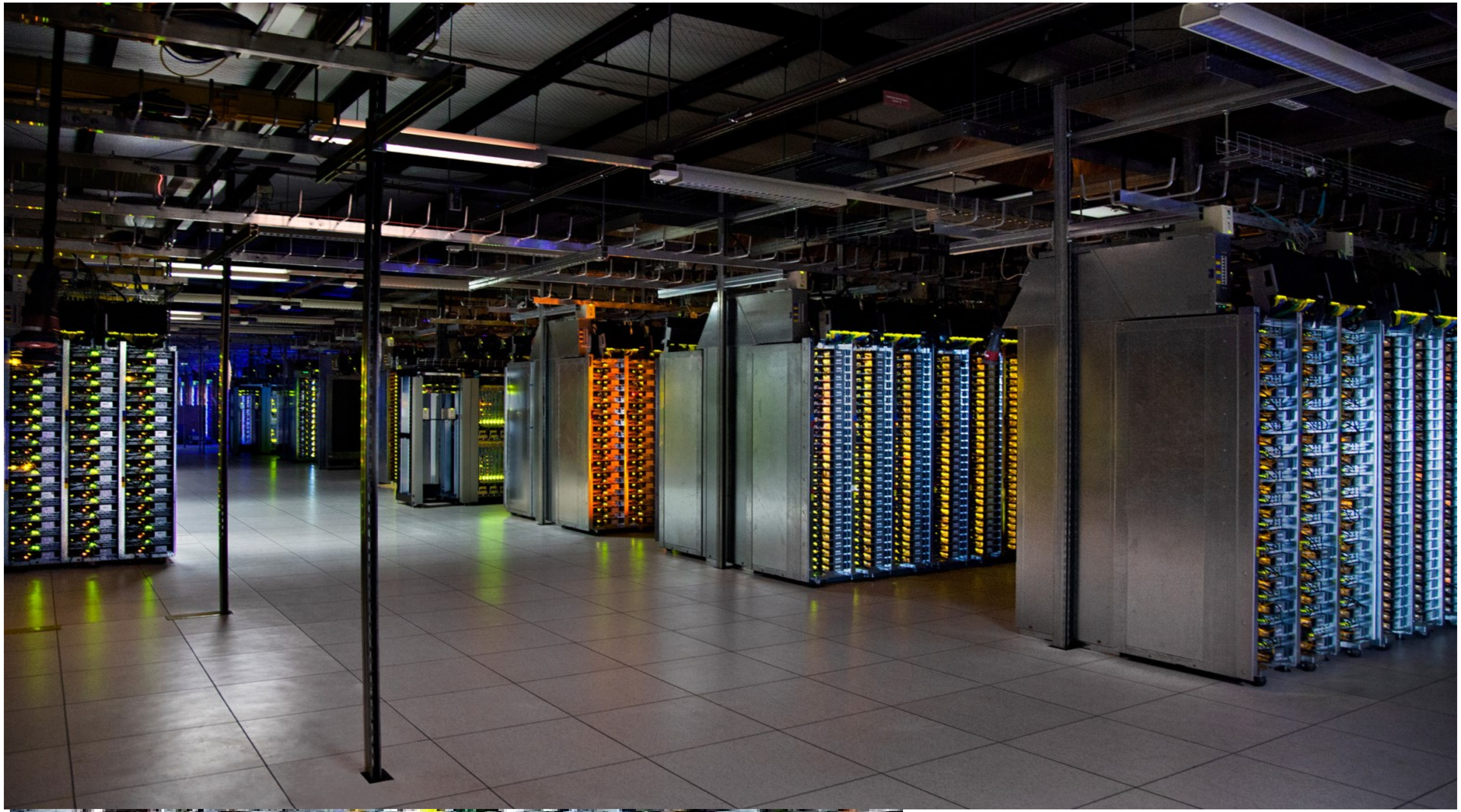
clients:

- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP

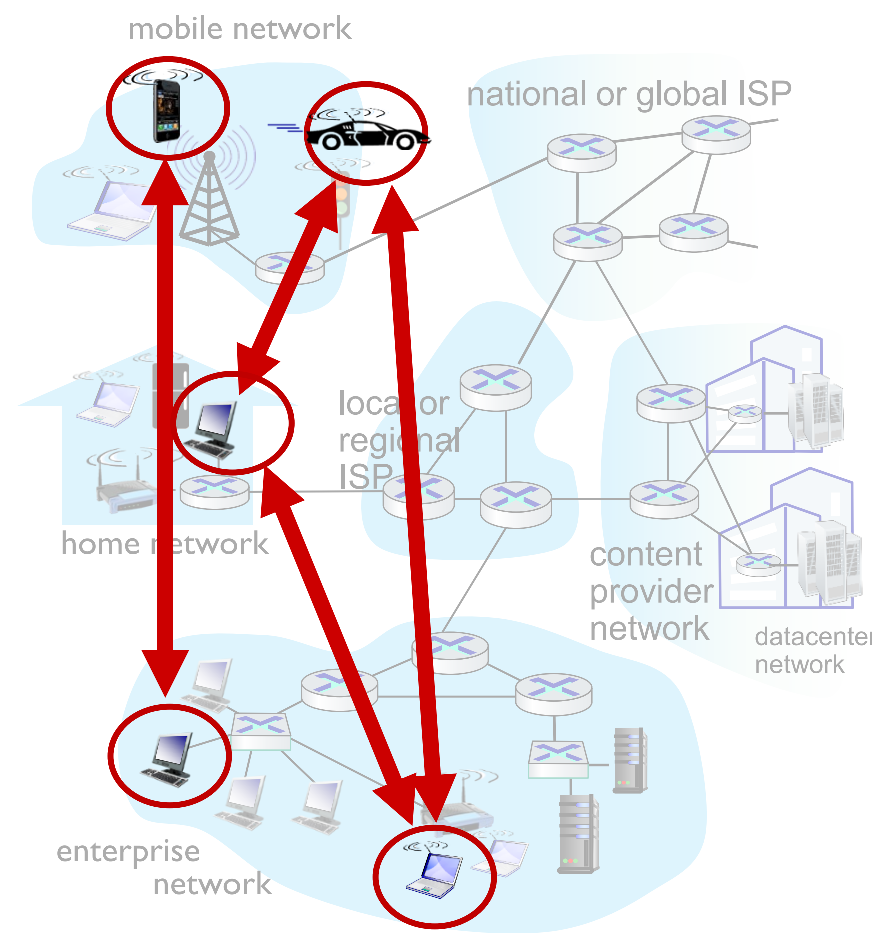


Web Servers?
Data center



Peer-to-peer (P2P) architecture

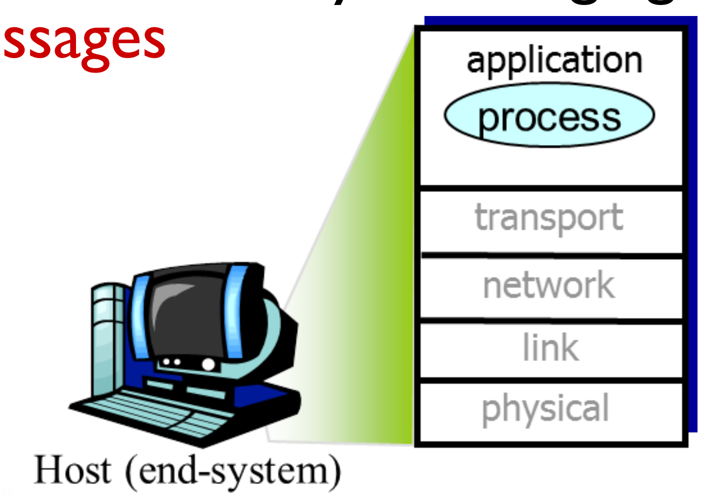
- usually ***no*** always-on server
- arbitrary end systems (peers) directly communicate
- peers request service from other peers, provide service in return to other peers
  - peers are intermittently connected and change IP addresses
- examples: P2P file sharing (BitTorrent), Skype (Voice/video call)



Processes communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (IPC)
- processes in different hosts communicate by exchanging messages



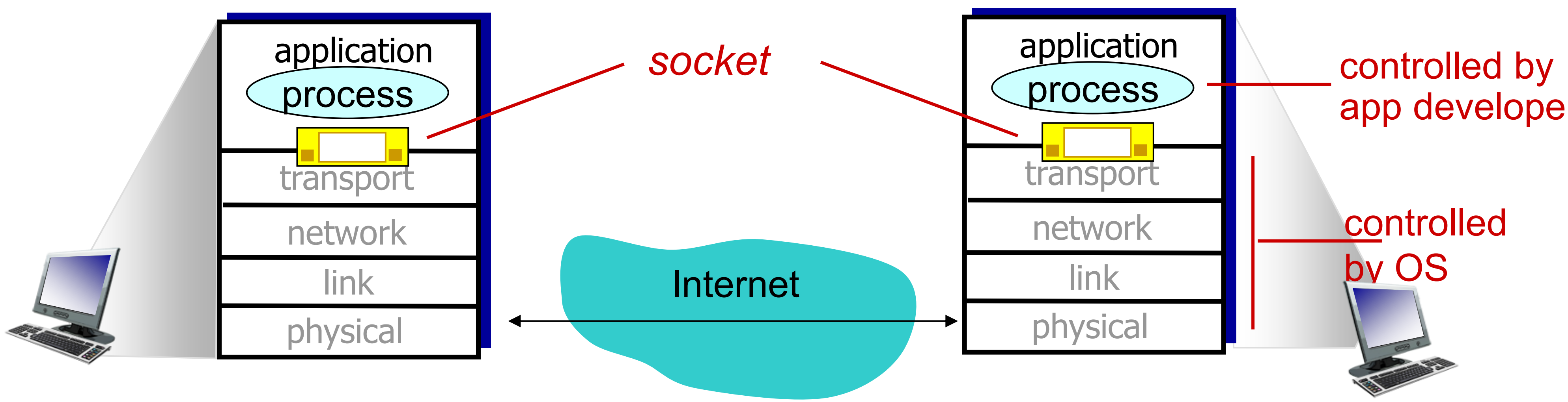
clients, servers

client process: process that initiates communication

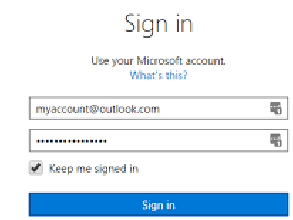
server process: process that waits to be contacted

Sockets

- process sends/receives messages to/from its socket
- **Socket**
  - *Software* ***interface*** between the application layer and transport-layer protocol
  - **Application Programming Interface (API)** between application and network
- Network programming ≈ socket programming



Addressing processes (IP address)

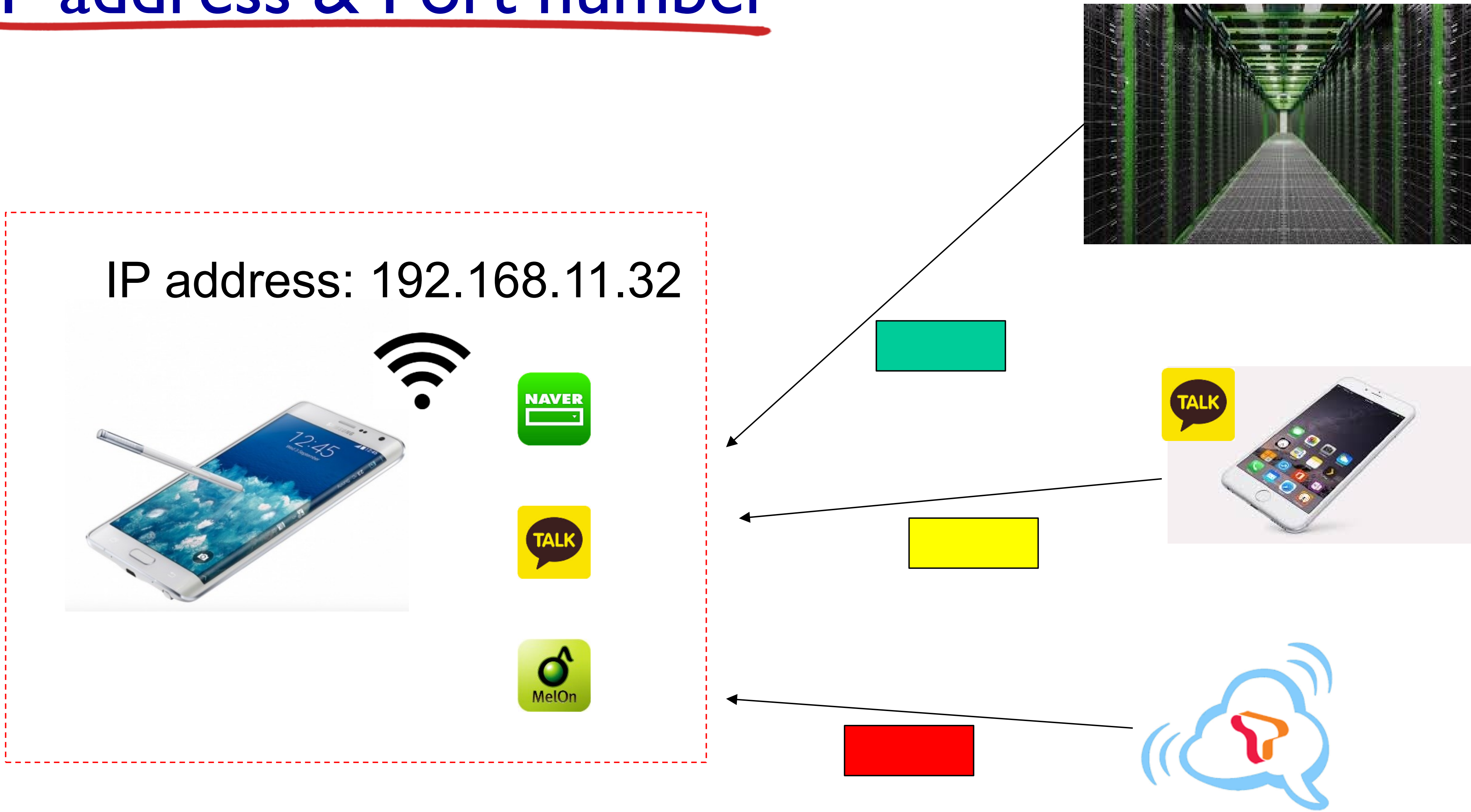


- to receive messages, process must have *identifier (id)*
- host device has unique 32-bit **IP address**

```
C:\Users\jyoo>ipconfig

Windows IP 구성

이더넷 어댑터 로컬 영역 연결:

   연결별 DNS 접미사. . . . :
   링크-로컬 IPv6 주소 . . . . : fe80::51f3:ffd1:6e46:8d13%14
   IPv4 주소 . . . . . . . . . : 192.168.0.11
   서브넷 마스크 . . . . . . . : 255.255.255.0
   기본 게이트웨이 . . . . . . : 192.168.0.1
```

- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host

IP address & Port number



Addressing processes

- *identifier* includes both IP address and port numbers associated with process on host.
- example server port numbers:
  - HTTP server: 80
  - mail server: 25
- to send an HTTP message to sw.gachon.ac.kr web server:
  - IP address: 222.122.41.206
  - port number: 80

App-layer protocol defines

- types of messages:
  - 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 IETF RFCs[1]
- allows for interoperability
- e.g., HTTP, SMTP

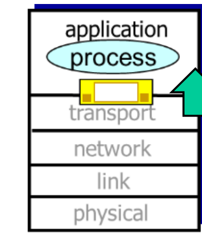
proprietary protocols:

- e.g., Skype, Zoom, WebEx

- Application vs. Application Layer protocol
  - Read Chapter 2.1.5 in textbook!

1) A Request for Comments (RFC) is from the Internet Engineering Task Force (IETF), the principal technical development and standards-setting bodies for the Internet. https://en.wikipedia.org/wiki/Request_for_Comments

What transport layer service does an app need?



data integrity

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

timing

- some apps (e.g., Internet telephony, interactive games) require **<u>low delay</u>** 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

Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	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
interactive games	loss-tolerant	Kbps+	yes, 10's msec

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *connection-oriented:* **setup** required between client and server processes

UDP service:

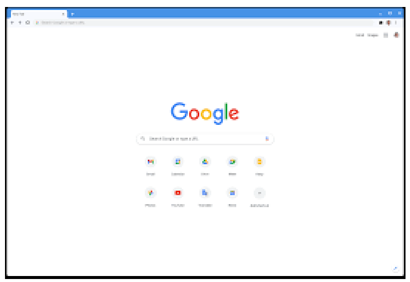
- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, or connection setup,
- But lightweight, so faster than TCP!

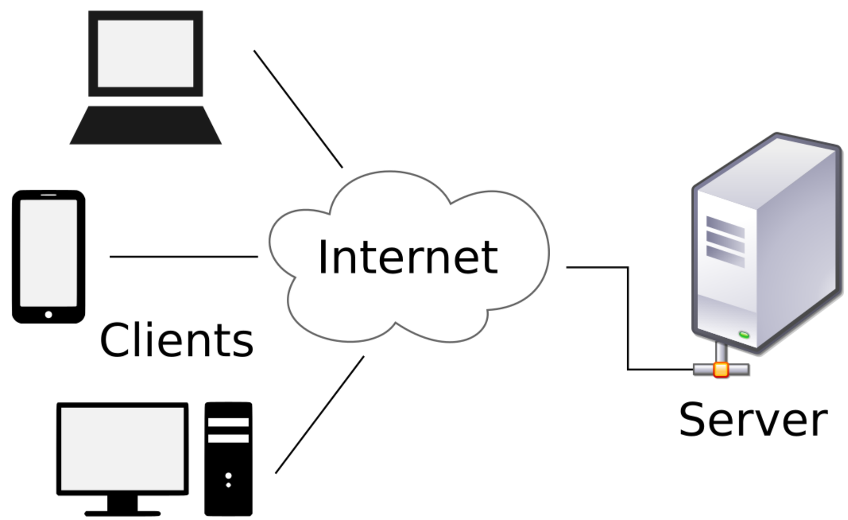
Internet transport protocols services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Question

How can chrome fetch the Web page from the server?





Chapter 2: outline

2.1 principles of network applications

- app architectures
- app requirements

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.6 Video Streaming and CDN

2.7 Socket Programming

Web and HTTP

First, a quick review…

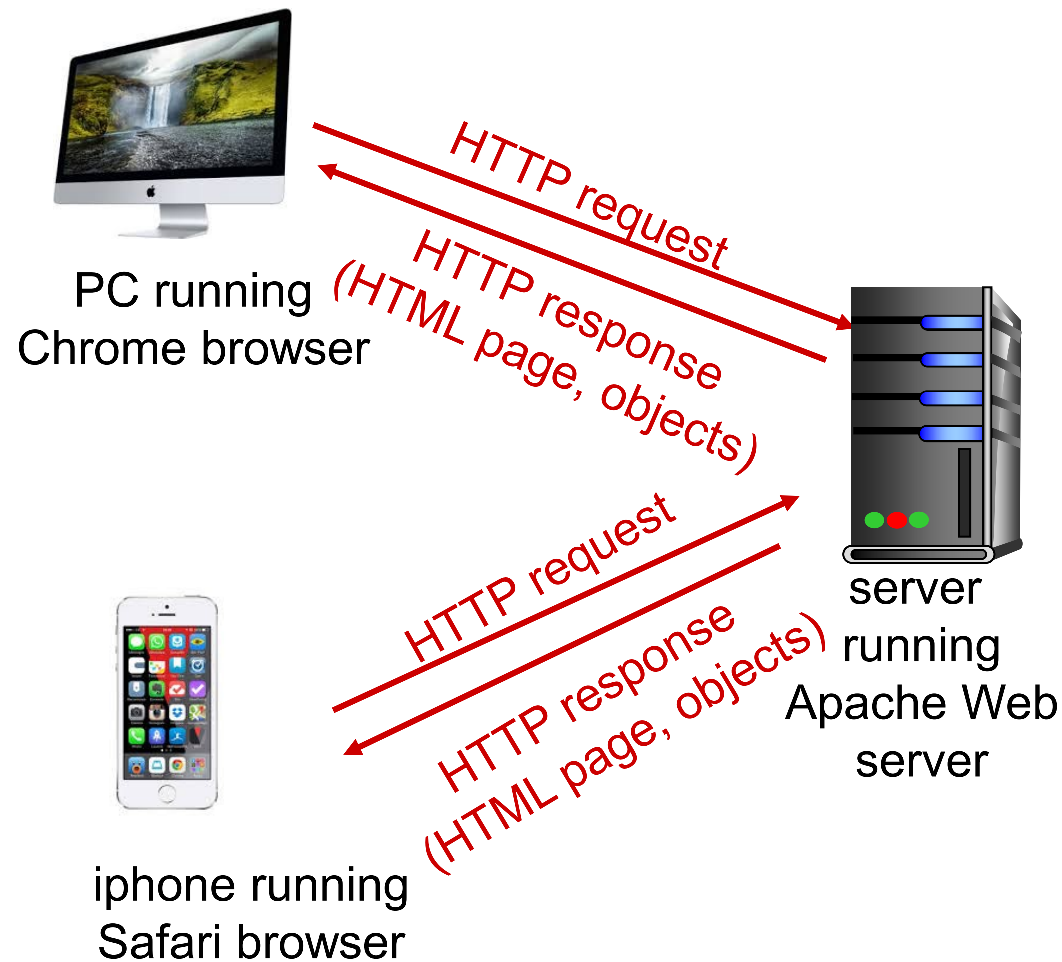
- web page consists of *objects,* each of which can be stored on different Web servers
  - object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL,* e.g.,

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

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *Web client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *Web server:* Web server sends (using HTTP protocol) objects in response to 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 <u>no</u> information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained at ____________

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

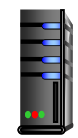
3. HTTP server receives request message, forms **HTTP** ***response message*** containing requested object, and sends message into its socket

time

Non-persistent HTTP: example (cont.)

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

HTTP connection: response time

Tx. delay?

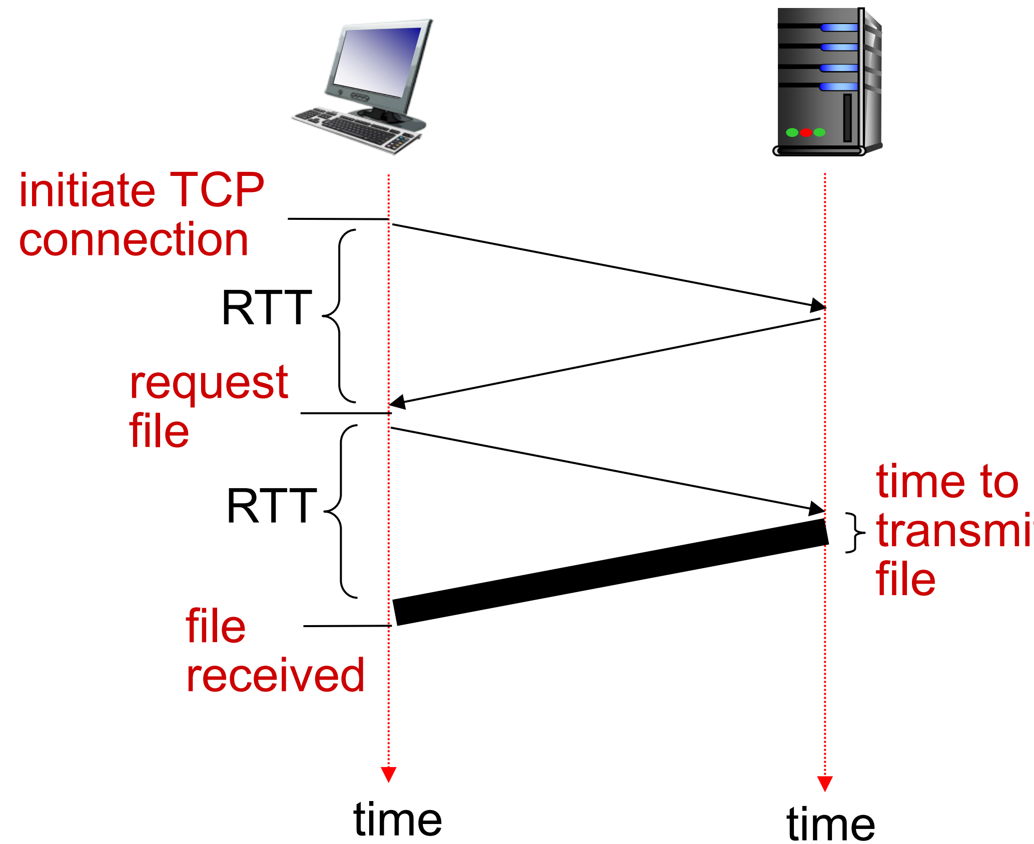
Round-trip time (RTT): time for a small packet to travel from client to server and back

- What type of delays does RTT include? (Recall Ch. 1.4)

HTTP 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 (why include this?)
- non-persistent HTTP response time =

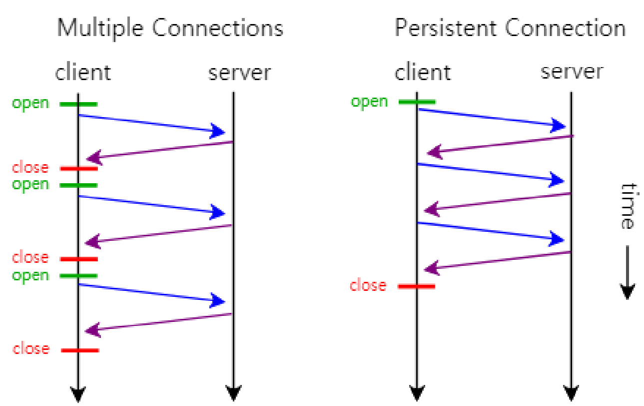
  2RTT+ file transmission time



Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:

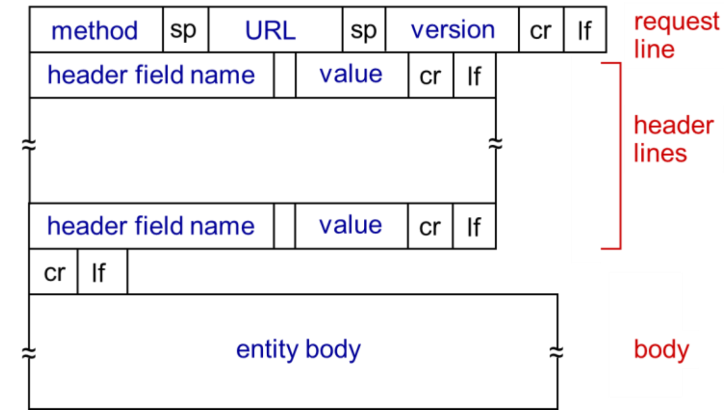
- requires **2 RTTs per object**



Persistent HTTP (HTTP1.1):

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- as little as **1 RTT per object**
  - (No need for new TCP connection)
- HTTP server closes connection after certain time (timeout)

HTTP request message



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

[request line]
method (GET, POST,...)
URL version

header lines

```
GET /index.html HTTP/1.1
Host: www-net.cs.umass.edu
Keep-Alive: 115
Connection: keep-alive
User-Agent: Firefox/3.6.10
Accept-Language: en-us,en;q=0.5
...
```

HTTP response message

status line
(protocol
status code
status phrase)

header
lines

Persistent or non-persistent?

data, e.g.,
requested
HTML file

```
HTTP/1.1 200 OK
Date: Tue, 12 Sep 2022 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 12 Sep 2022 11:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
(data data data data data ... )
```

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

- 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

2. type in a GET HTTP request:

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

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

3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

Cookies: keeping state at server

But sometimes server wishes to…
- ✓ restrict user access
- ✓ serve content as a function of user identity
- ☞ **Cookies** allow sites to keep track of users

many Web sites use cookies

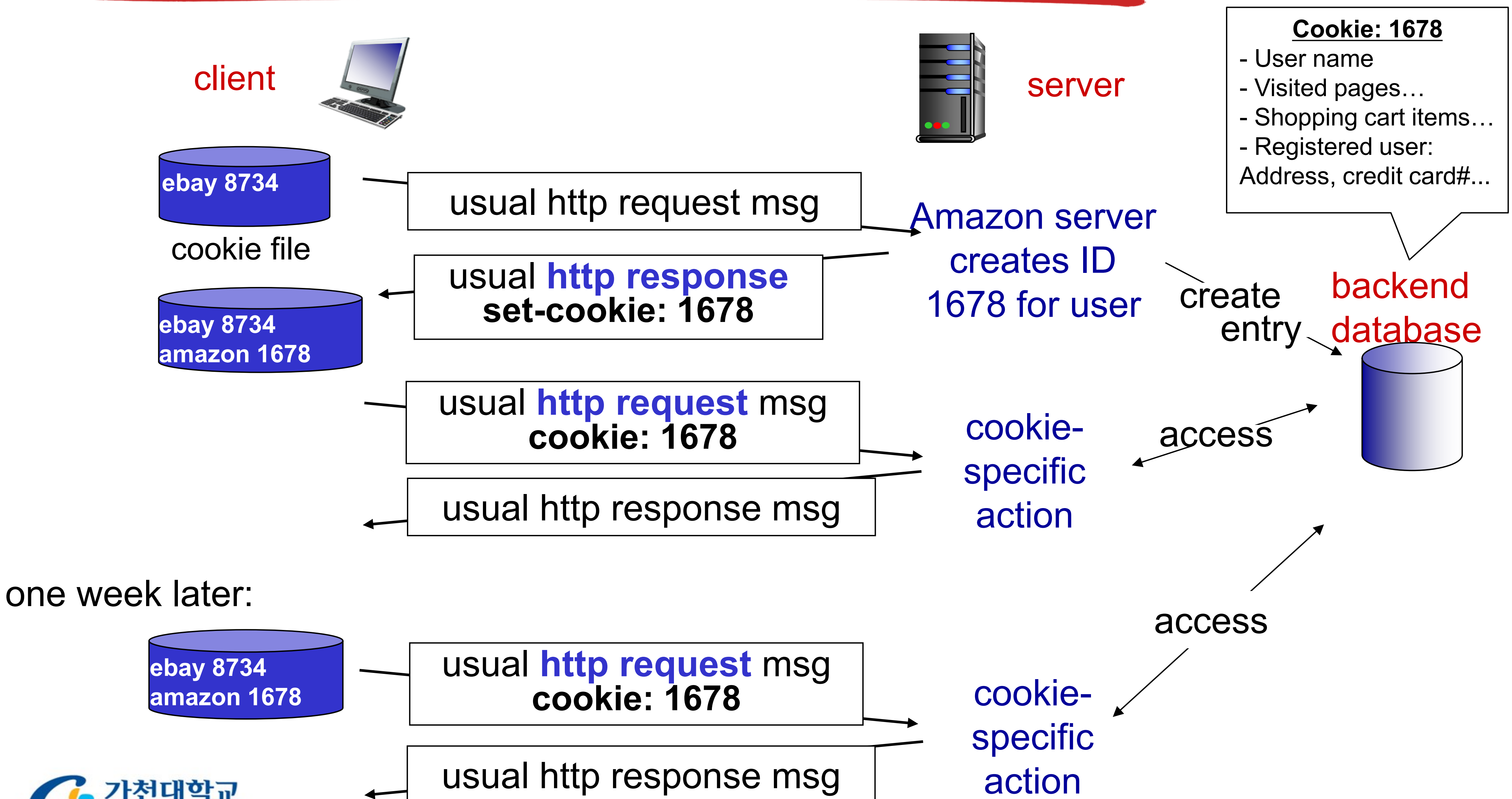
four components:

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

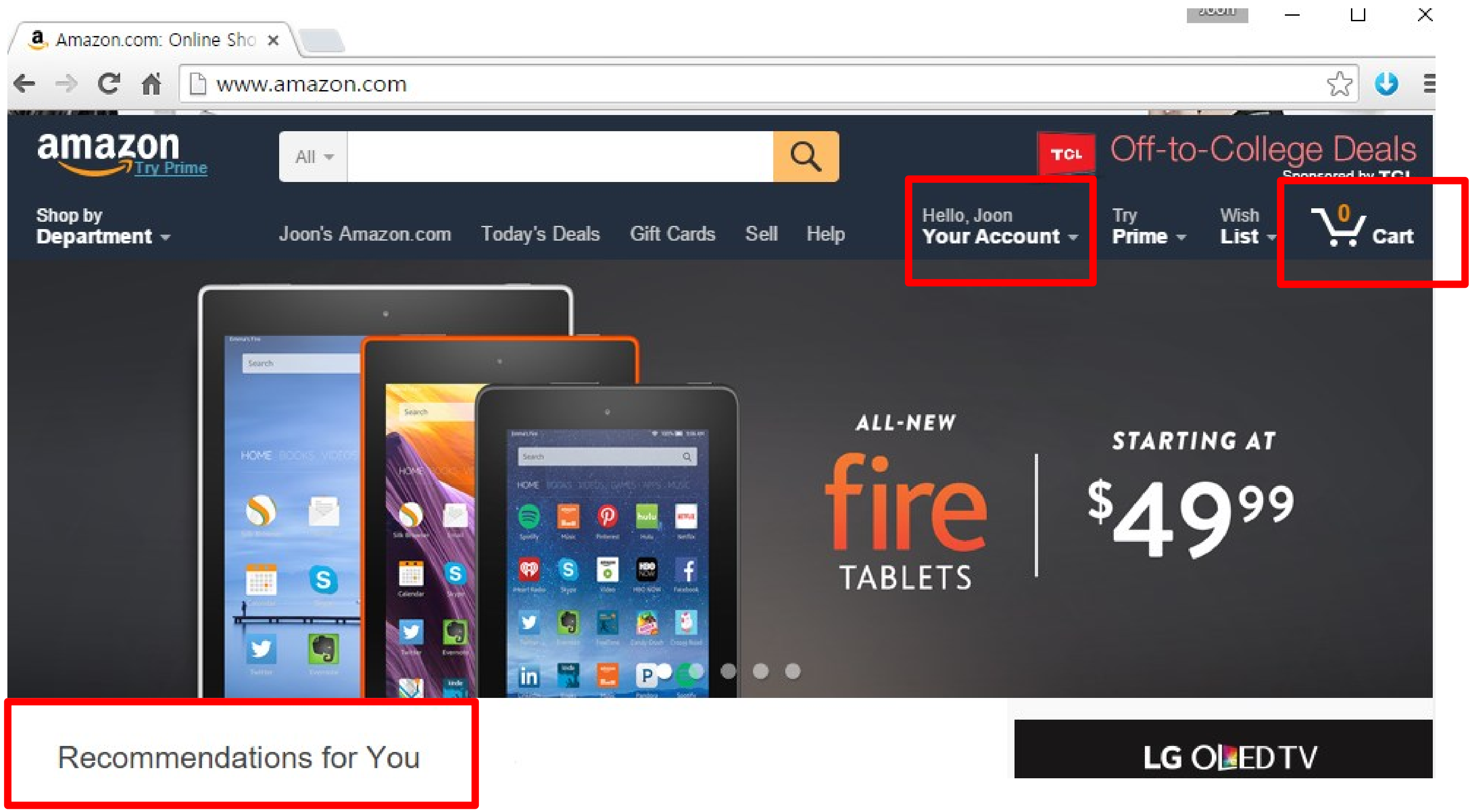
example:

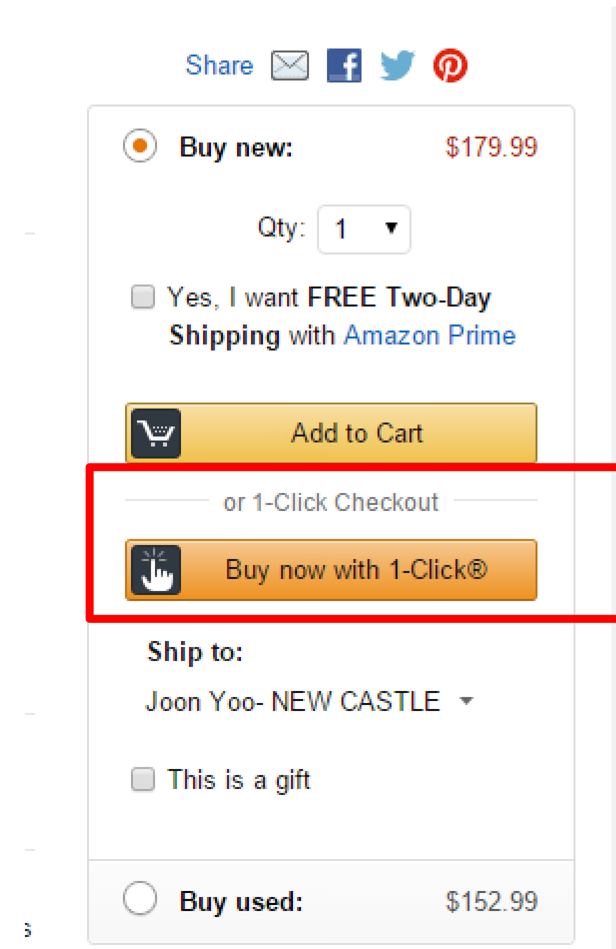
- Susan uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka "cookie")
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to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

Cookies: keeping “state” (cont.)



Cookies used in Amazon.com





Cookies (continued)

what cookies can be used for:

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

aside

cookies and privacy:

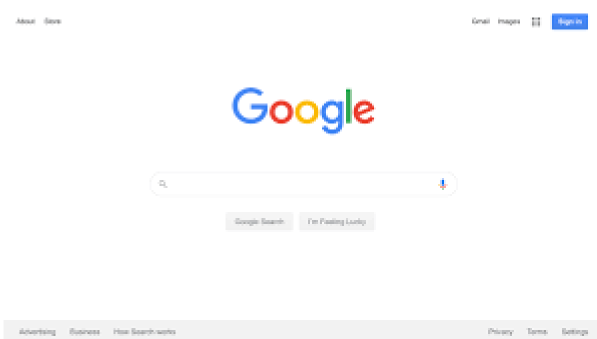
- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

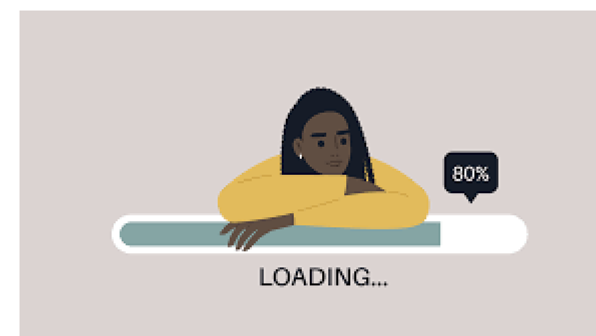
how to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Question

Why are **popular** Web pages fast and some others slow?

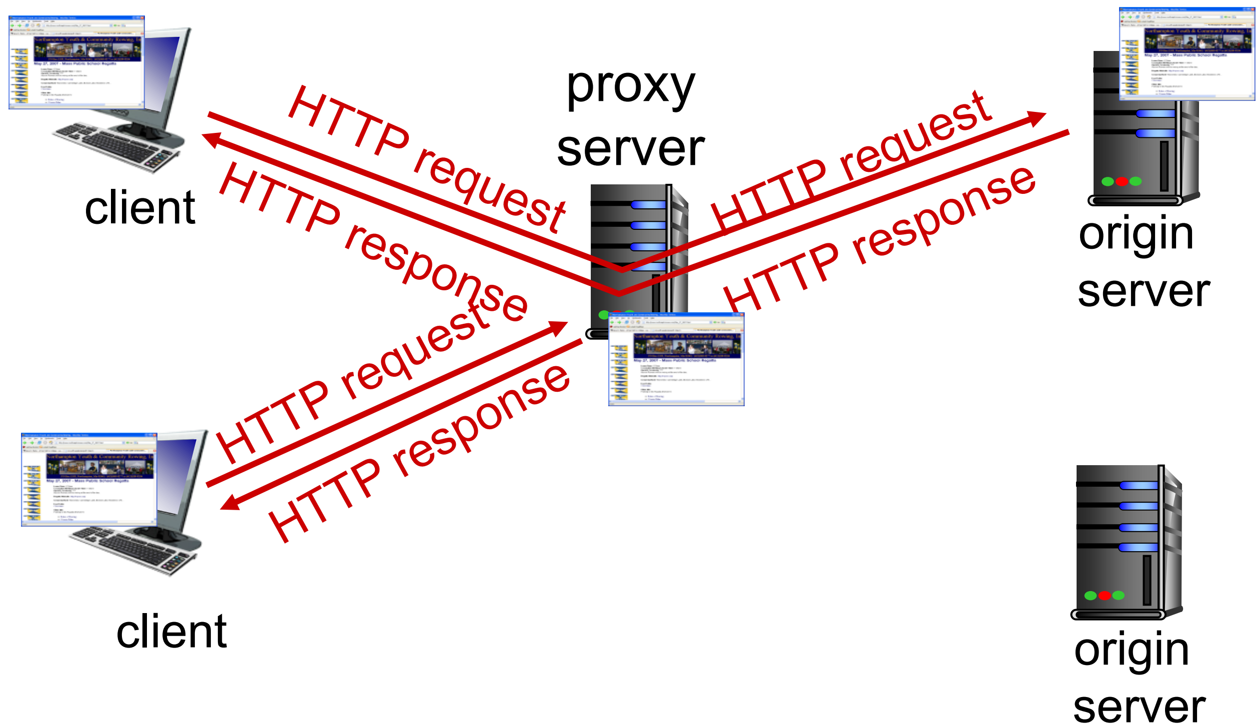




Web caches (proxy server)

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
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

why Web caching?

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

Caching example:

Assumptions:

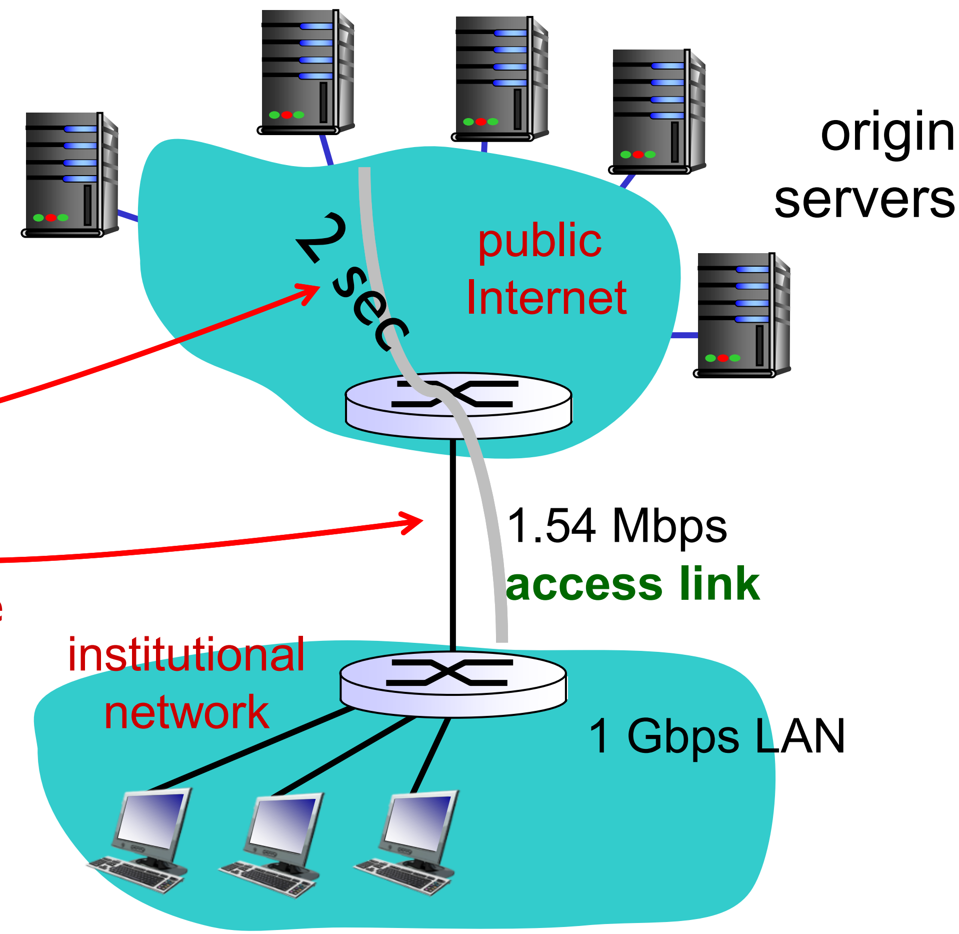
- avg object size (100kbits), avg request rate from browsers to origin servers (15/sec)
  - avg data rate to browsers: 100kbit x 15/sec = 1.50 Mbps
  - RTT from institutional router to any origin server: 2 sec
- **access link** rate: 1.54 Mbps

Consequences:

- LAN utilization: **0.0015**
- access link utilization >= 97%
- Total delay = Internet delay + access link delay + LAN delay

  = 2 sec + <u>**minutes**</u> + usecs

problem: large delays at high utilization!



Possible solution: fatter access link

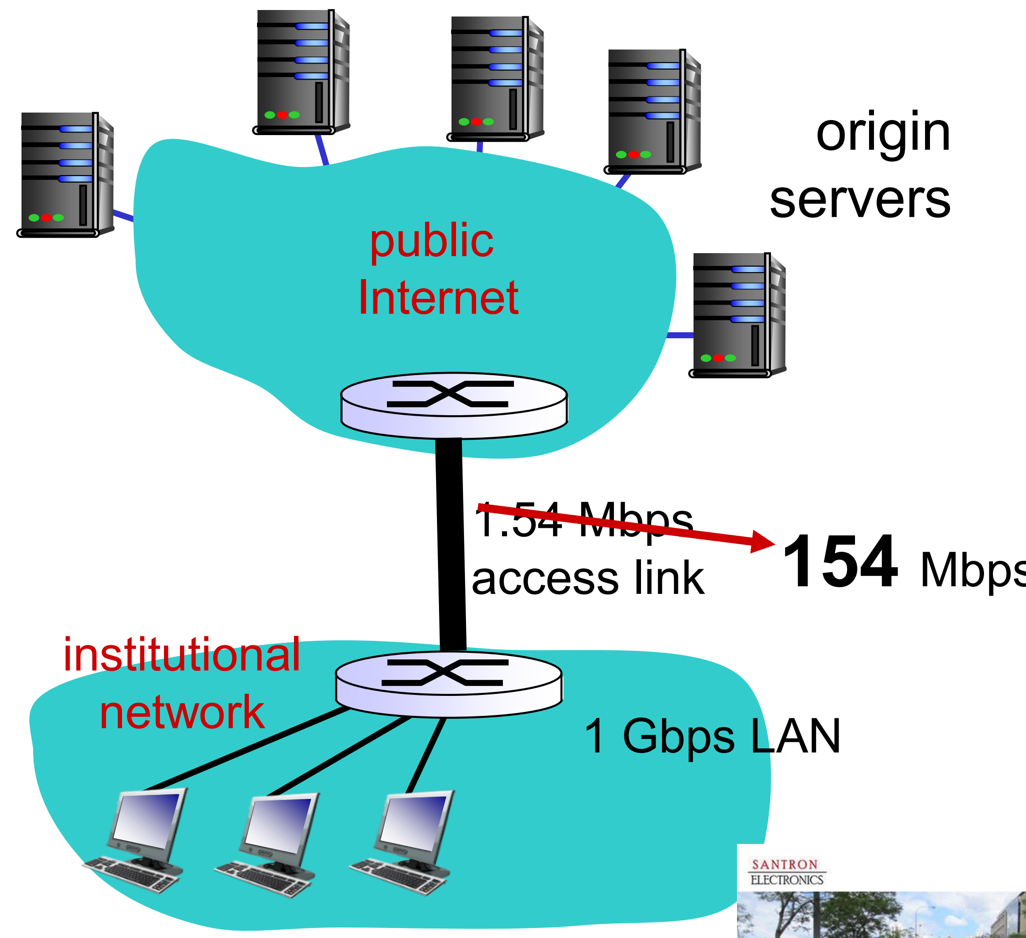
assumptions:

- avg object size (100kbits), avg request rate from browsers to origin servers (15/sec)
  - avg data rate to browsers: 100kbit x 15/sec = 1.50 Mbps
  - RTT from institutional router to any origin server: 2 sec
- **access link** rate: ~~1.54 Mbps~~ **154** Mbps

consequences:

- LAN utilization: 0.0015
- access link utilization = ~~97%~~ .0097
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + ~~minutes~~ msecs + usecs = **~3 secs**

Cost: increased access link speed (not cheap!)



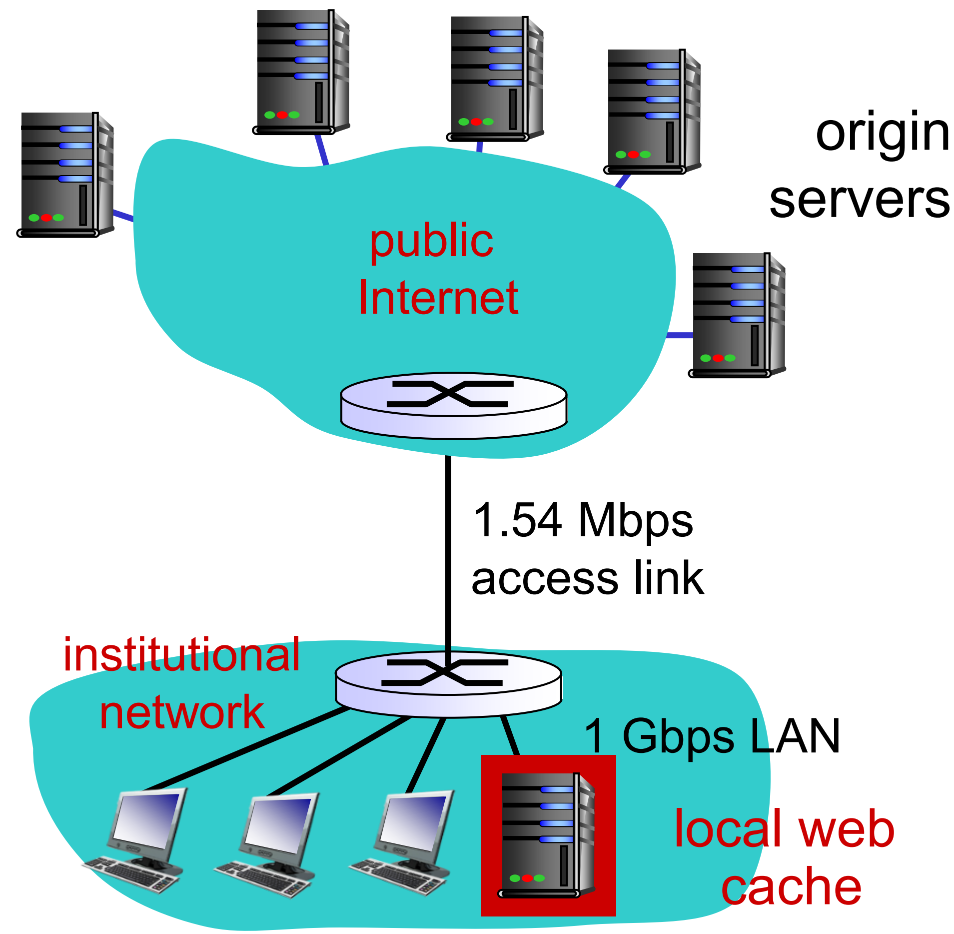
Caching example: install local Web cache

assumptions:

- avg object size (100kbits), avg request rate from browsers to origin servers (15/sec)
  - avg data rate to browsers: 100kbit x 15/sec = 1.50 Mbps
  - RTT from institutional router to any origin server: 2 sec
- **access link** rate: 1.54 Mbps

Performance:

- LAN utilization: ?
- access link utilization = ?
- total delay = Internet delay + access delay + LAN delay = ?

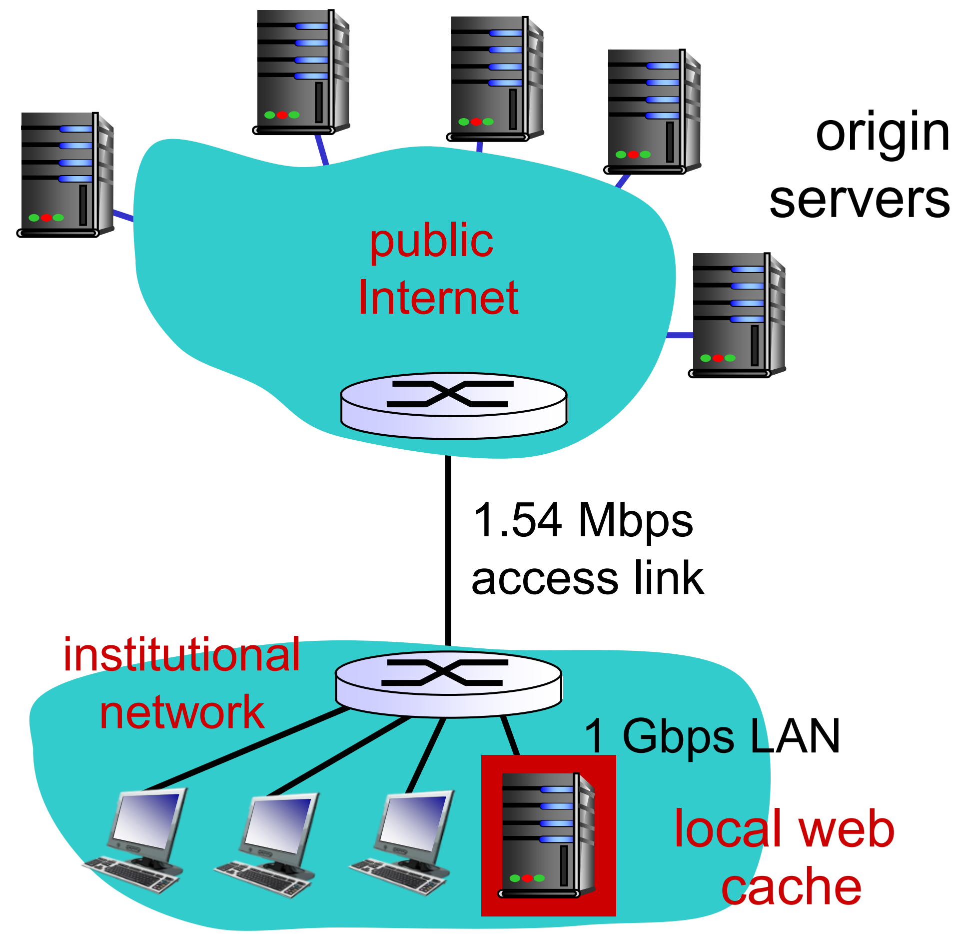


Cost: web cache server (cheap!)

Caching example: install local cache

Calculating access link utilization, delay with cache:

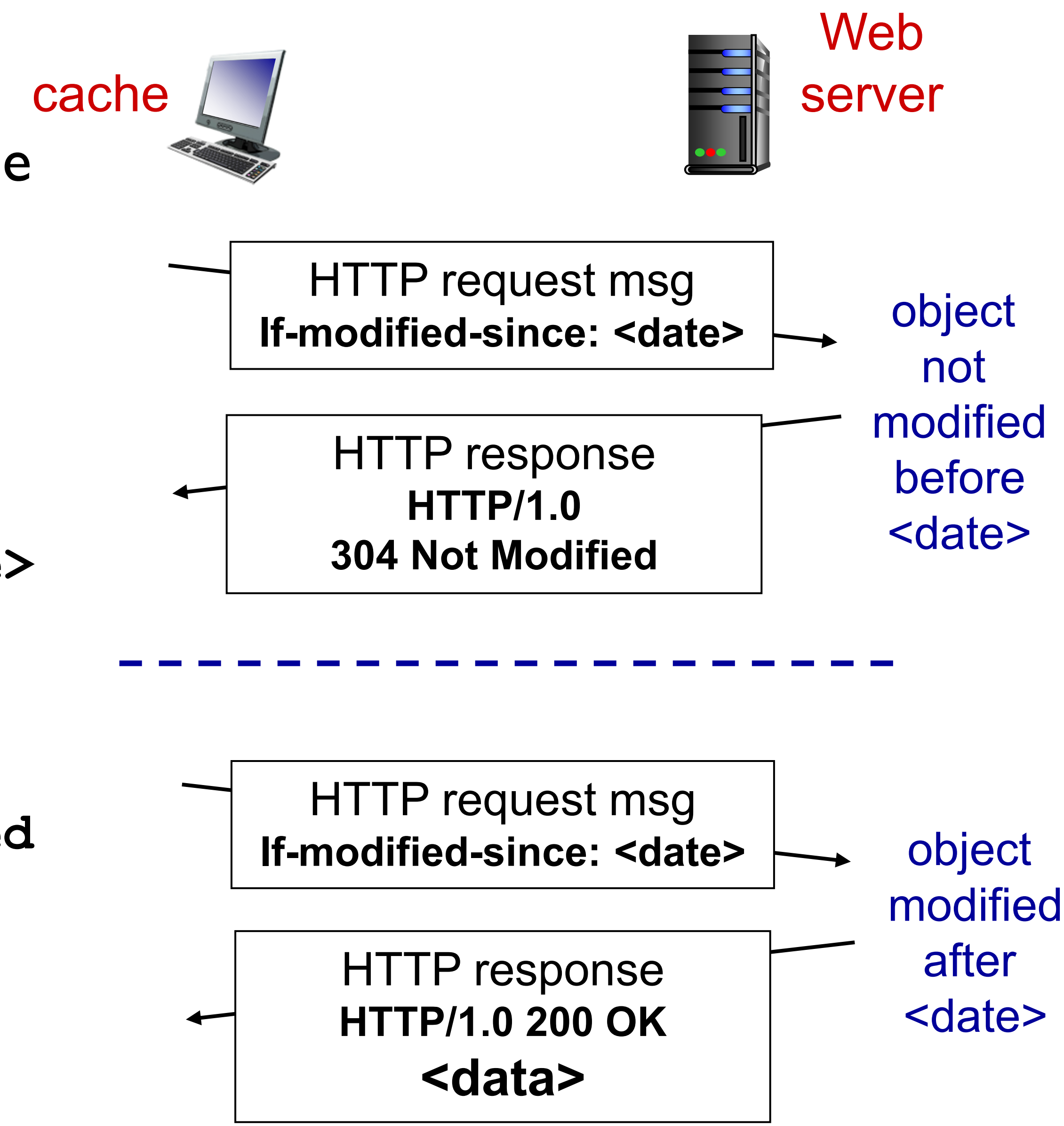
- suppose cache hit rate is **0.4**
  - 40% requests will be satisfied almost immediately !!
  - 60% requests satisfied by origin server
- access link utilization:
  - 60% of requests use access link
- data rate to browsers over access link = 0.6*1.50 Mbps = 0.9 Mbps
  - utilization = 0.9/1.54 = **0.58** (from 0.97)
- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~ms) = **~ 1.2 secs**
  - less than with 154 Mbps link (~3 secs) and cheaper too!



- reduce **response time** for client request
- reduce **traffic** on an institution's access link

Conditional GET

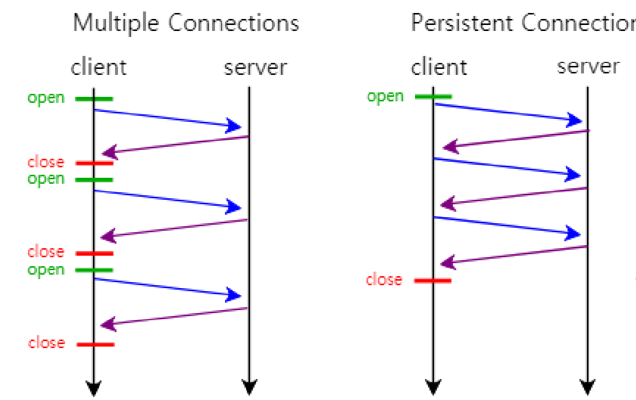
- *Goal:* don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *cache:* specify date of cached copy in HTTP request
  If-modified-since: <date>
- *server:* response contains no object if cached copy is up-to-date:
  HTTP/1.0 304 Not Modified



HTTP/2

Key goal: decreased delay in multi-object HTTP requests

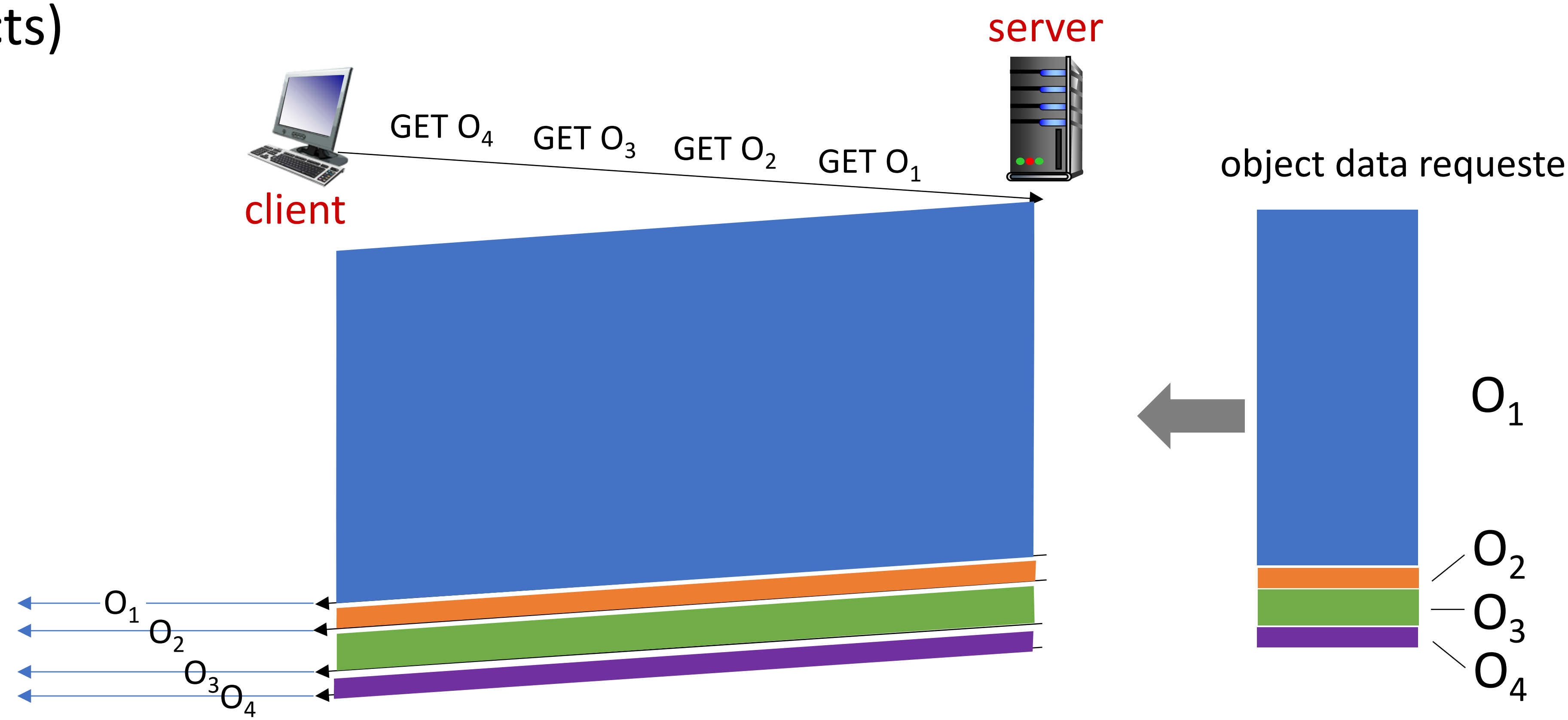
<u>*HTTP1.1:*</u> [RFC 2068,1997] introduced multiple, pipelined GETs over single TCP connection server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests



- with FCFS, small object may have to wait for transmission (head-of-line (HOL) blocking) behind large object(s)

HTTP/2: mitigating HOL blocking

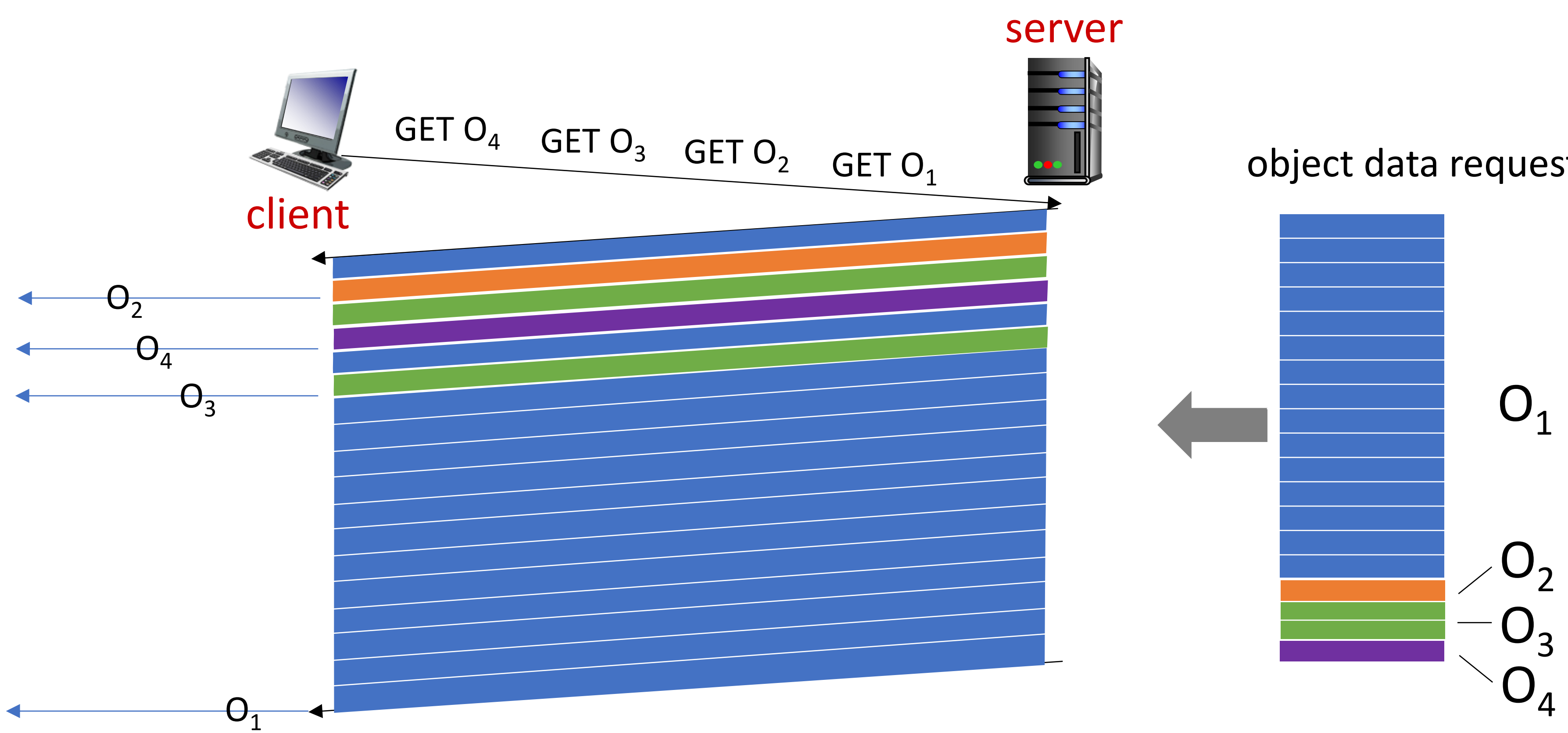
HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)



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

Key goal: decreased delay in multi-object HTTP requests

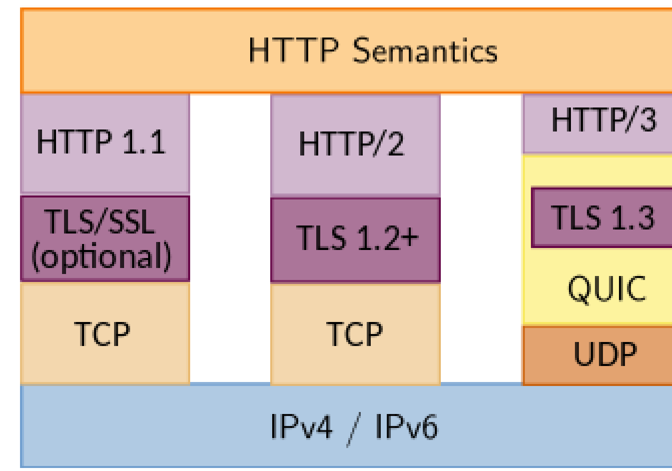
HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:

- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- divide objects into frames, schedule frames to mitigate HOL blocking
- Most browsers (Chrome, IE, Safari,…) support HTTP/2
- HTTP/2 is used by 35.5% of all websites. (as of Sep. 2023)
  - https://w3techs.com/technologies/details/ce-http2

HTTP/2 to HTTP/3

Key goal: decreased delay in multi-object HTTP requests

- HTTP/3 standardized in June 2022
  - Builds on the concepts established by HTTP/2
  - Supported by 94% of running Web[1] browsers, and 26% of to 10M websites
- HTTP/3
  - transport?: HTTP/1.1 and HTTP/2 used TCP, but HTTP/3 uses **QUIC**
  - maybe more on QUIC in transport layer



1) https://caniuse.com/?search=HTTP%2F3