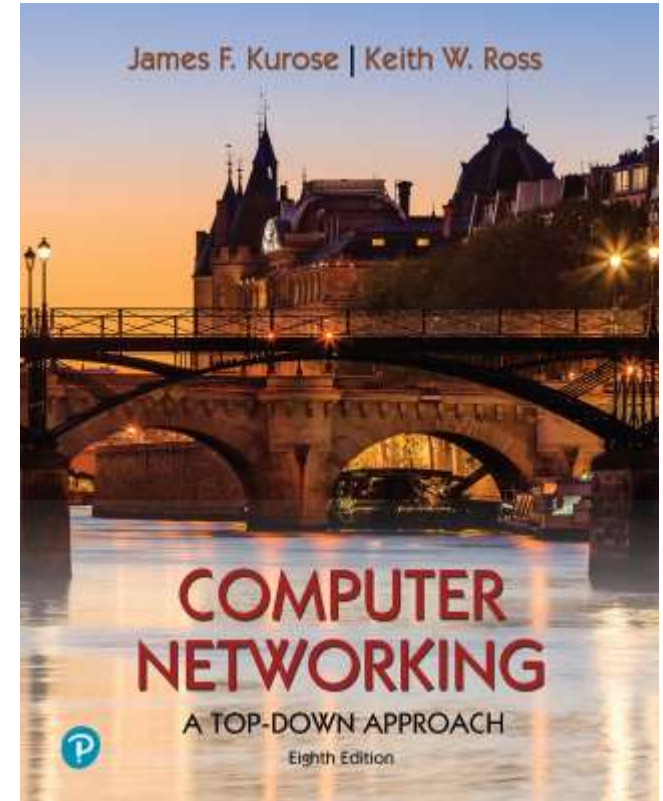# Internet Architecture
TCP/IP Protocol Suite
IPC - Concept of Socket
HTTP 1.1, 2 and 3
Cookies
Web Caching

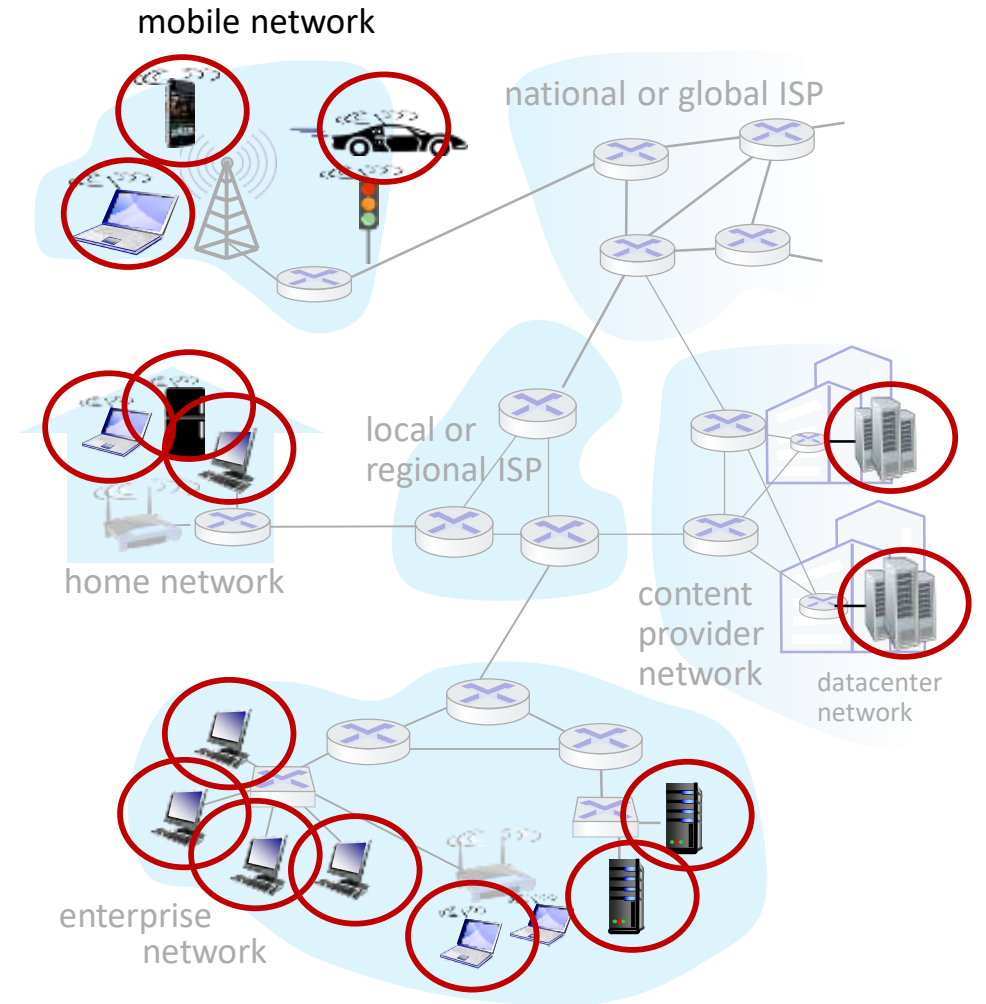*Computer Networking: A Top-Down Approach*

8th edition
Jim Kurose, Keith Ross
Pearson, 2020

A closer look at Internet structure

Network edge:

- hosts: clients and servers
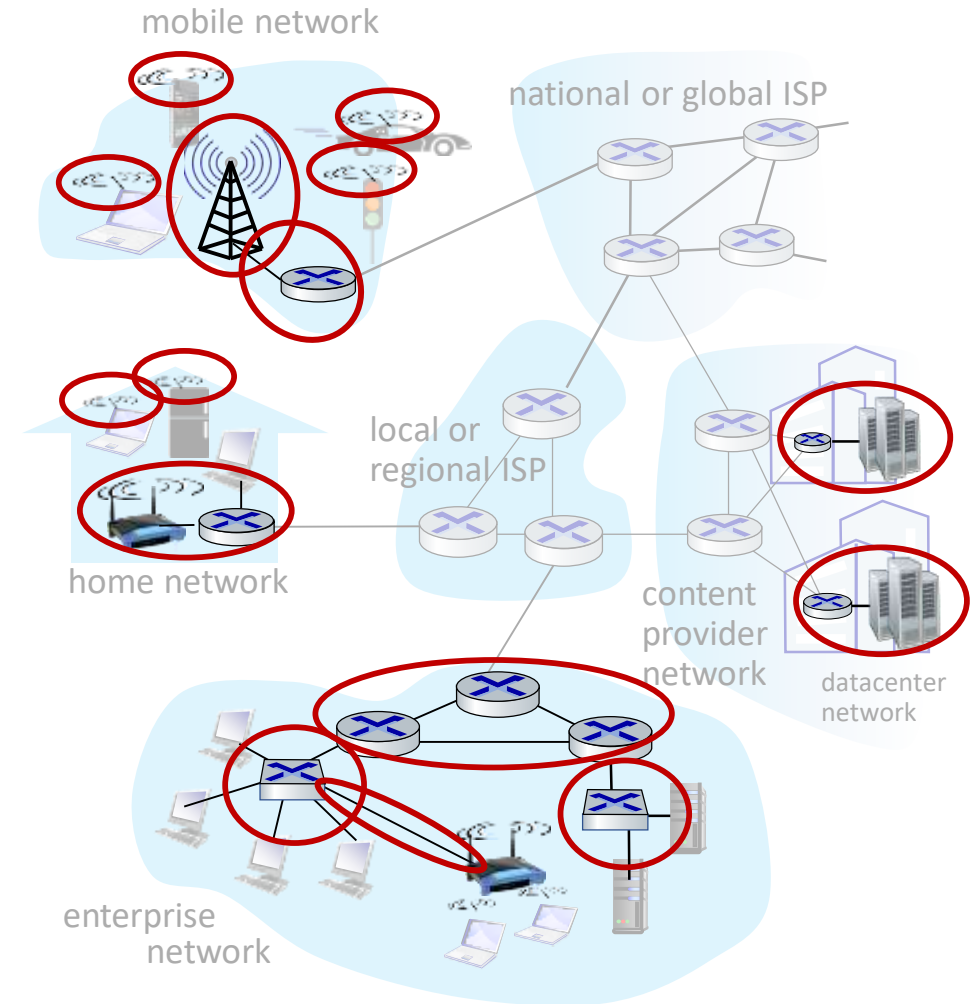- servers often in data centers

mobile network

national or global ISP

local or regional ISP

home network

content provider network

datacenter network

enterprise network

A closer look at Internet structure

Network edge:

- ■ hosts: clients and servers
- ■ servers often in data centers

Access networks, physical media:

- ■ wired, wireless communication links

mobile network

national or global ISP

local or regional ISP

home network

content provider network

datacenter network

enterprise network

A closer look at Internet structure
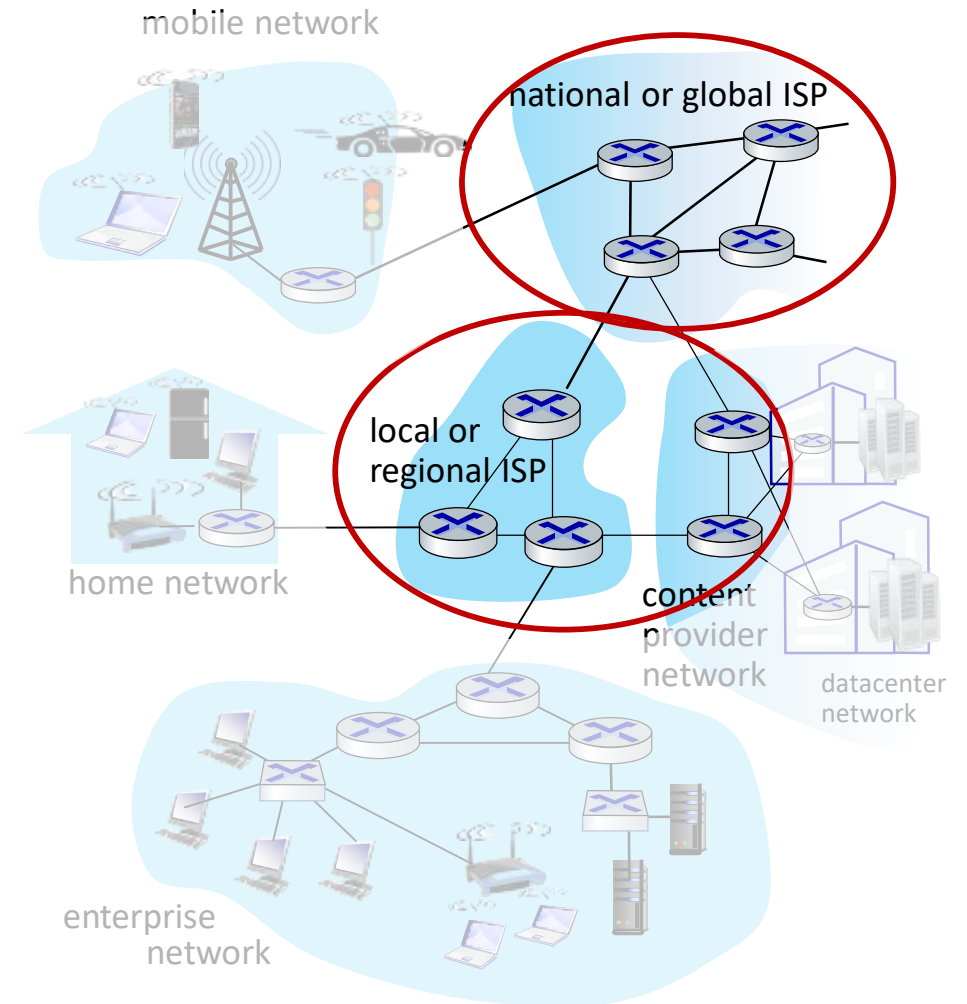
Network edge:

- hosts: clients and servers

- servers often in data centers

Access networks, physical media:

- wired, wireless communication links

Network core:

- interconnected routers
- network of networks

mobile network

national or global ISP

local or regional ISP

home network

content provider network

datacenter network

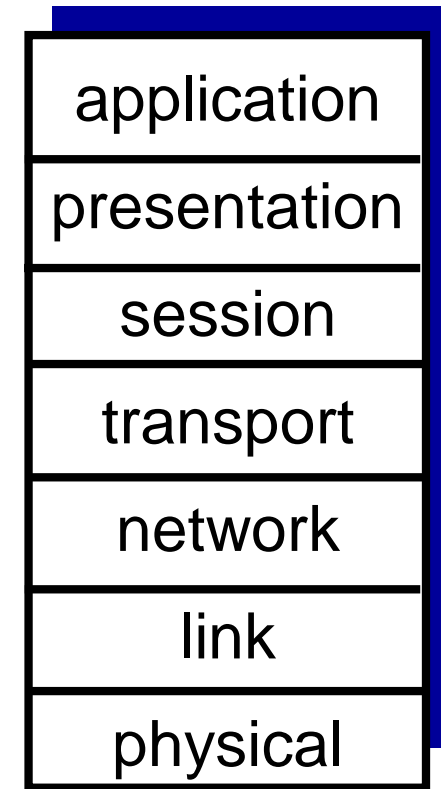enterprise network

Protocol "layers" and reference models

Networks are complex,
with many "pieces":

- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

ISO/OSI reference model
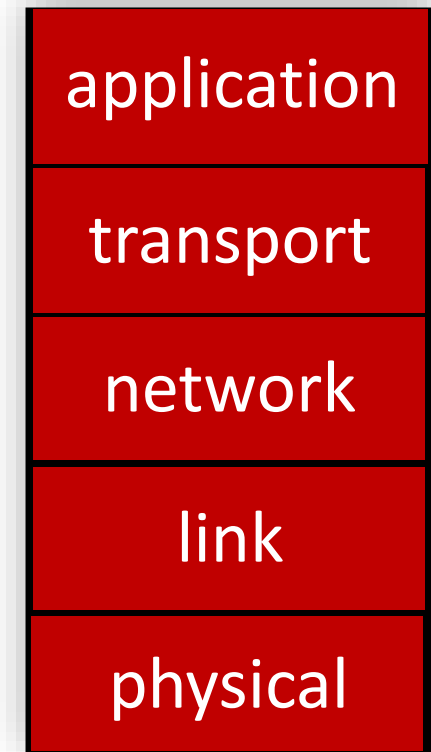
Two layers not found in Internet protocol stack!

- *presentation:* allow applications to interpret meaning of data, e.g., encryption, compression, machine-specific conventions

- *session:* synchronization, checkpointing, recovery of data exchange

- Internet stack "missing" these layers!
  - these services, *if needed,* must be implemented in application
  - needed?

application
presentation
session
transport
network
link
physical

The seven layer OSI/ISO reference model

Layered TCP/IP stack

■ *application:* supporting network applications
  • HTTP, IMAP, SMTP, DNS
■ *transport:* process-process data transfer
  • TCP, UDP
■ *network:* routing of datagrams from source to destination
  • IP, routing protocols
■ *link:* data transfer between neighboring network elements
  • Ethernet, 802.11 (WiFi), PPP
■ *physical:* bits "on the wire"

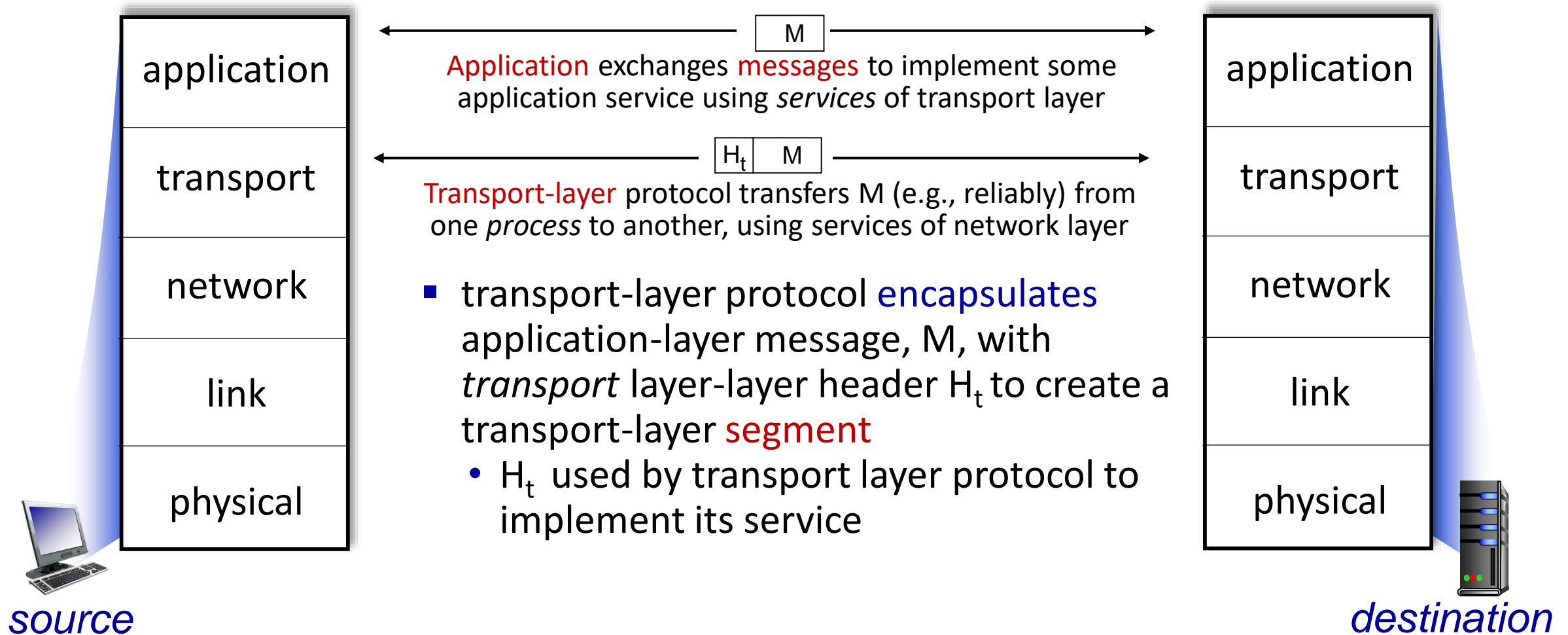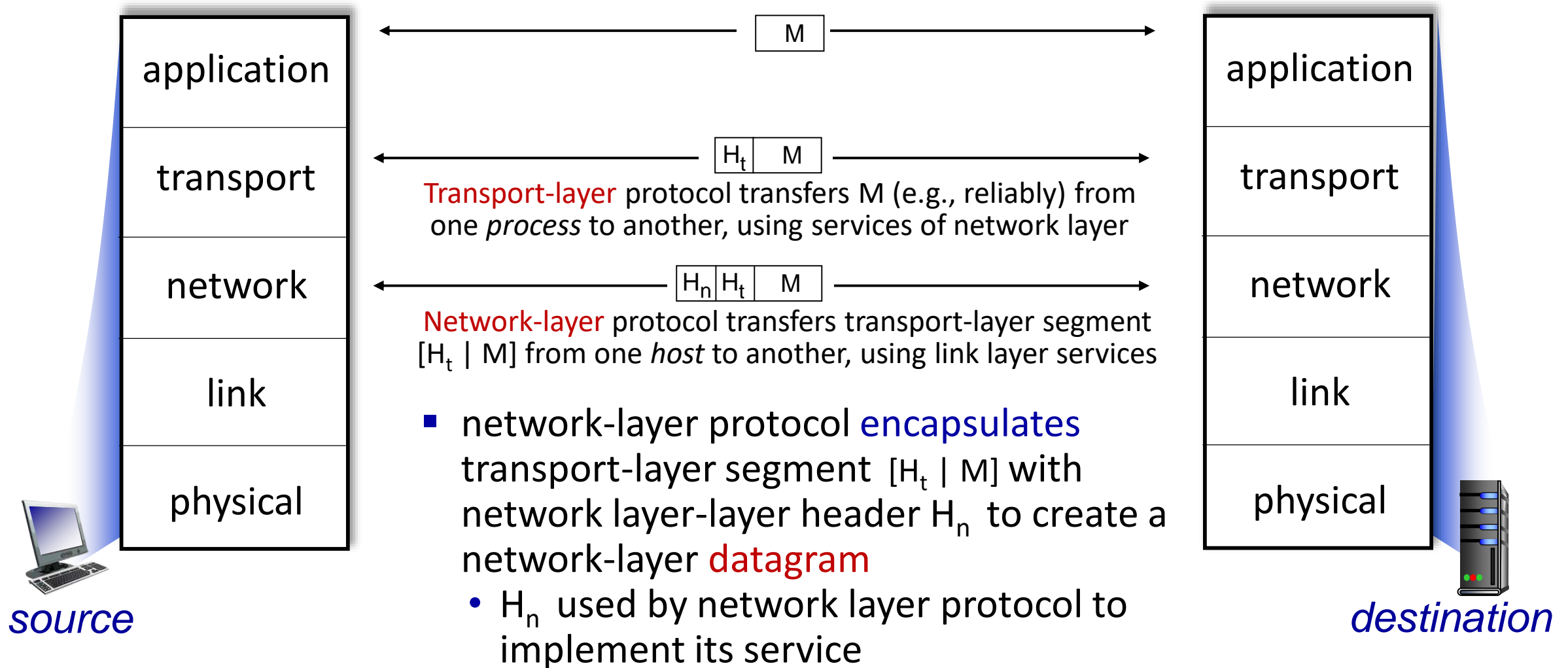| application |
| --- |
| transport |
| network |
| link |
| physical |

Why layering?

Approach to designing/discussing complex systems:

- explicit structure allows identification, relationship of system's pieces
  - layered *reference model* for discussion
- modularization eases maintenance, updating of system
  - change in layer's service *implementation*: transparent to rest of system
  - e.g., change in gate procedure doesn't affect rest of system

Services, Layering and Encapsulation

application

transport

network

link

physical

source

application

transport

network

link

physical

destination

M

Application exchanges messages to implement some application service using *services* of transport layer

H_t | M

Transport-layer protocol transfers M (e.g., reliably) from one *process* to another, using services of network layer

- transport-layer protocol encapsulates application-layer message, M, with *transport* layer-layer header H_t to create a transport-layer segment
  - H_t used by transport layer protocol to implement its service

Services, Layering and Encapsulation

application

transport

network

link

physical

source

M

Transport-layer protocol transfers M (e.g., reliably) from one *process* to another, using services of network layer

H_t M

Network-layer protocol transfers transport-layer segment $[H_t \mid M]$ from one *host* to another, using link layer services

H_n H_t M

- network-layer protocol encapsulates transport-layer segment $[H_t \mid M]$ with network layer-layer header H_n to create a network-layer datagram
  - H_n used by network layer protocol to implement its service

application

transport

network

link

physical

destination

Services, Layering and Encapsulation

Network-layer protocol transfers transport-layer segment [H_t | M] from one *host* to another, using link layer services

Link-layer protocol transfers datagram [H_n| [H_t |M] from *host* to neighboring host, using network-layer services

- link-layer protocol encapsulates network datagram [H_n| [H_t |M], with link-layer header H_l to create a link-layer frame

source *destination*

Services, Layering and Encapsulation

Encapsulation: an end-end view

message   M

segment H_t M

datagram H_n H_t M

frame H_l H_n H_t M

source

| application |
| transport |
| network |
| link |
| physical |

| link |
| physical |

switch

destination

M

H_t M

H_n H_t M

H_l H_n H_t M

| application |
| transport |
| network |
| link |
| physical |

H_n H_t M

H_l H_n H_t M

| network |
| link |
| physical |

H_n H_t M

router

"Real" Internet delays and routes

- what do "real" Internet delay & loss look like?

- **traceroute** program: provides delay measurement from source to router along end-end Internet path towards destination. For all *i:*
  - sends three packets that will reach router *i* on path towards destination (with time-to-live field value of *i*)
  - router *i* will return packets to sender
  - sender measures time interval between transmission and reply

Real Internet delays and routes

traceroute: gaia.cs.umass.edu to www.eurecom.fr

3 delay measurements from
gaia.cs.umass.edu to cs-gw.cs.umass.edu

```
1  cs-gw (128.119.240.254)  1 ms  1 ms  2 ms
2  border1-rt-fa5-1-0.gw.umass.edu (128.119.3.145)  1 ms  1 ms  2 ms
3  cht-vbns.gw.umass.edu (128.119.3.130)  6 ms 5 ms 5 ms
4  jn1-at1-0-0-19.wor.vbns.net (204.147.132.129)  16 ms 11 ms 13 ms
5  jn1-so7-0-0-0.wae.vbns.net (204.147.136.136)  21 ms 18 ms 18 ms
6  abilene-vbns.abilene.ucaid.edu (198.32.11.9)  22 ms  18 ms  22 ms
7  nycm-wash.abilene.ucaid.edu (198.32.8.46)  22 ms  22 ms  22 ms
8  62.40.103.253 (62.40.103.253)  104 ms 109 ms 106 ms
9  de2-1.de1.de.geant.net (62.40.96.129)  109 ms 102 ms 104 ms
10  de.fr1.fr.geant.net (62.40.96.50)  113 ms 121 ms 114 ms
11  renater-gw.fr1.fr.geant.net (62.40.103.54)  112 ms  114 ms  112 ms
12  nio-n2.cssi.renater.fr (193.51.206.13)  111 ms  114 ms  116 ms
13  nice.cssi.renater.fr (195.220.98.102)  123 ms  125 ms  124 ms
14  r3t2-nice.cssi.renater.fr (195.220.98.110)  126 ms  126 ms  124 ms
15  eurecom-valbonne.r3t2.ft.net (193.48.50.54)  135 ms  128 ms  133 ms
16  194.214.211.25 (194.214.211.25)  126 ms  128 ms  126 ms
17  * * *
18  * * *
19  fantasia.eurecom.fr (193.55.113.142)  132 ms  128 ms  136 ms
```

3 delay measurements
to border1-rt-fa5-1-0.gw.umass.edu

trans-oceanic link

looks like delays
decrease! Why?

* means no response (probe lost, router not replying)

* Do some traceroutes from exotic countries at www.traceroute.org

Application Layer Protocol - HTTP

Processes communicating

process: program running within a host

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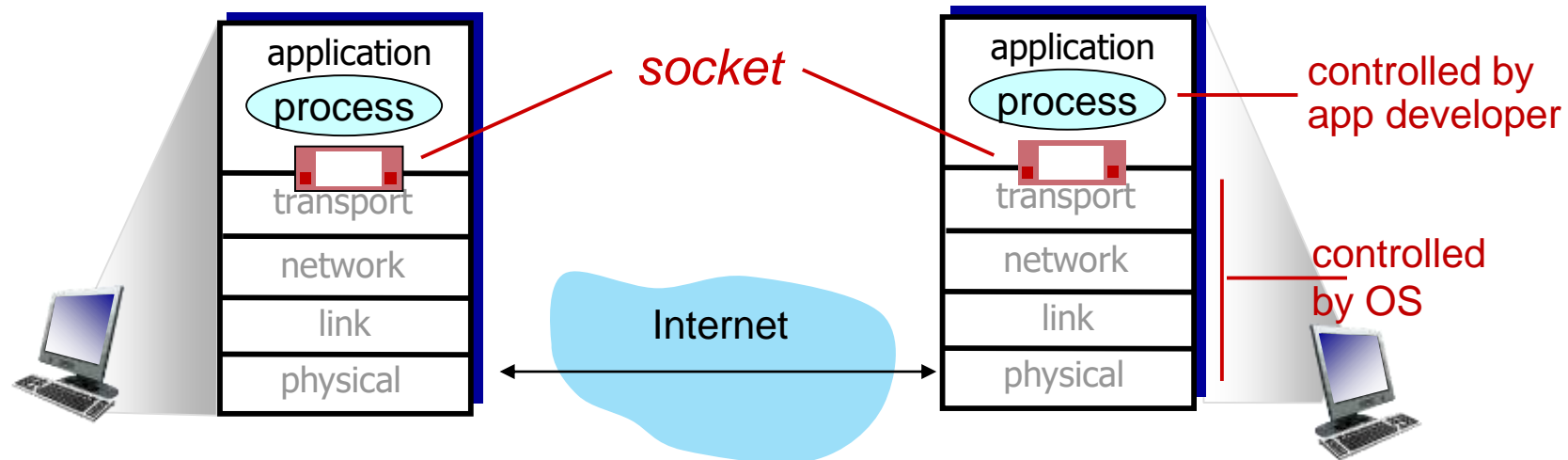- processes in different hosts communicate by exchanging messages

clients, servers
- *client process:* process that initiates communication
- *server process:* process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its socket

- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side

Addressing processes

- to receive messages, process must have *identifier*

- host device has unique 32-bit IP address

- *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

- *identifier* includes both IP address and port numbers associated with process on host.

- example port numbers:
  - HTTP server: 80
  - mail server: 25

- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80

- more shortly...

An application-layer protocol defines:

- 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, Zoom

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.,
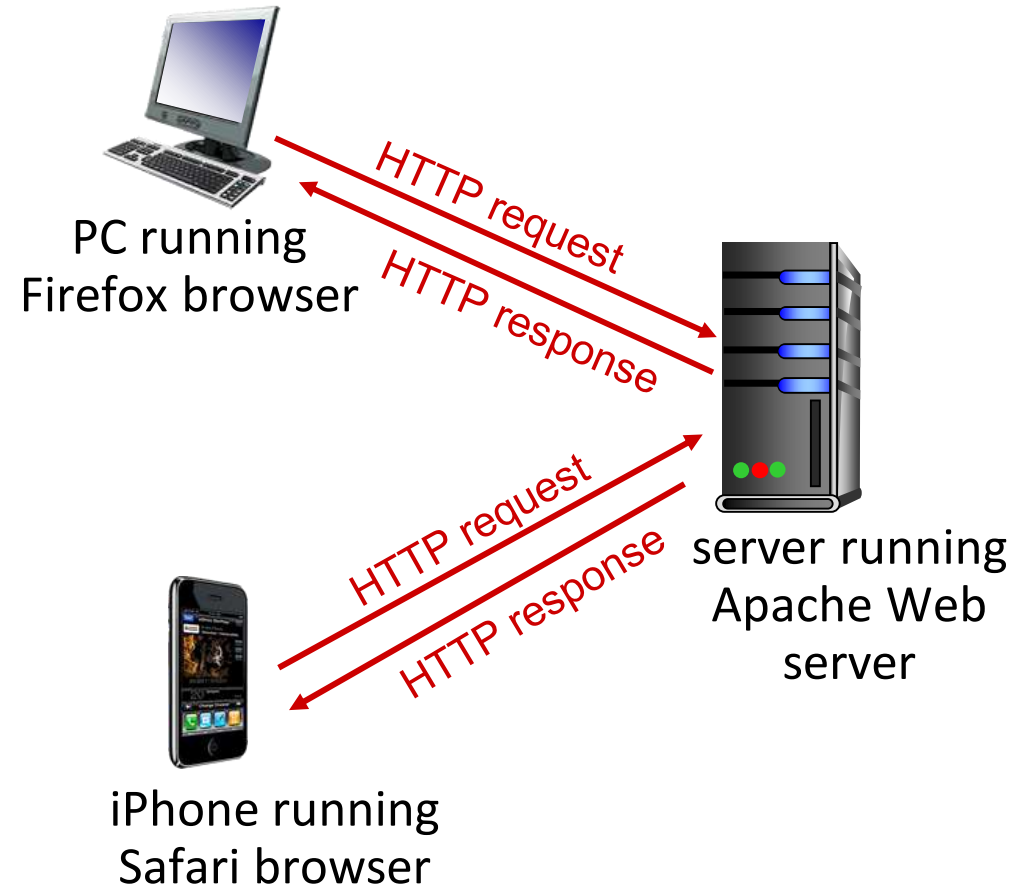
```
www.someschool.edu/someDept/pic.gif
```

host name   path name

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

PC running
Firefox browser

HTTP request

HTTP response

server running
Apache Web
server

HTTP request

HTTP response

iPhone running
Safari browser

HTTP overview (continued)

HTTP 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

HTTP connections: two types

Non-persistent HTTP

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

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
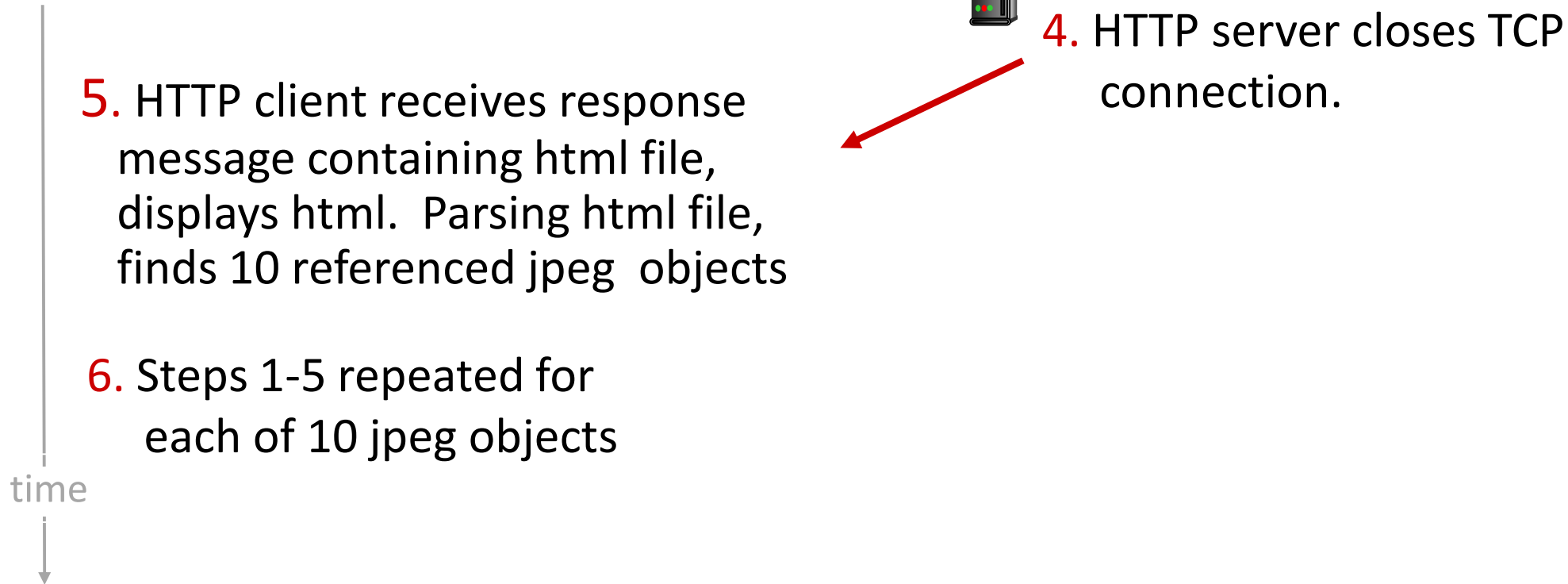
2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

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

time

Non-persistent HTTP: example (cont.)

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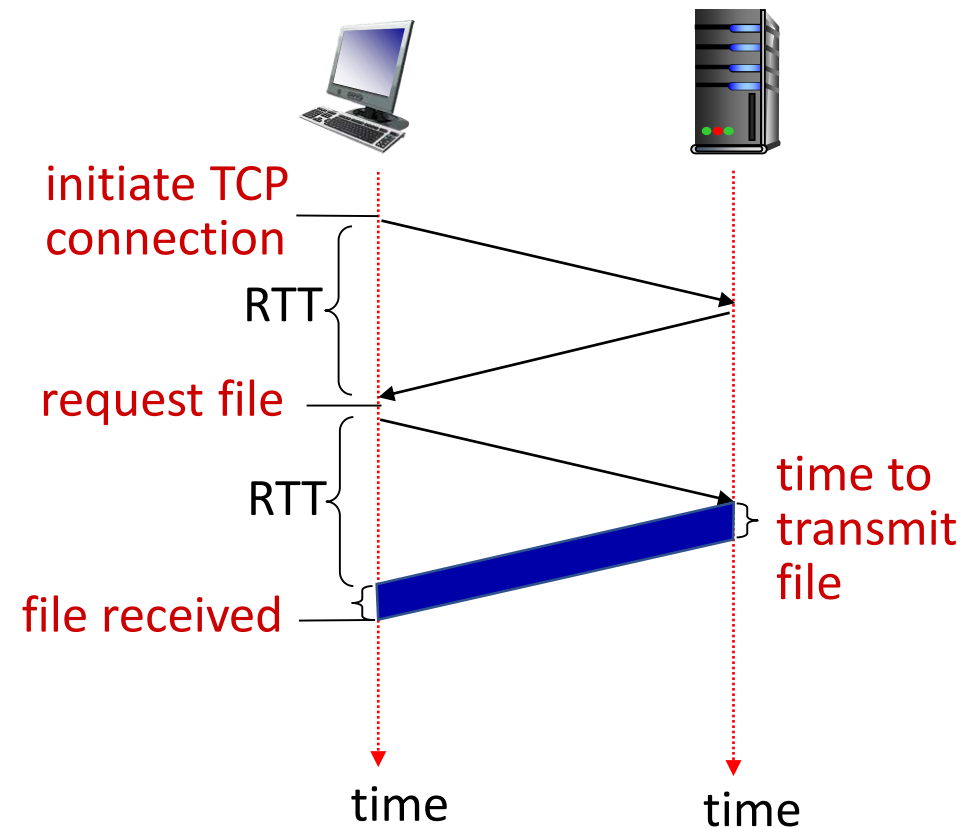(containing text, references to 10 jpeg images)

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

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

initiate TCP connection

RTT

request file

RTT

file received

time to transmit file

time   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:
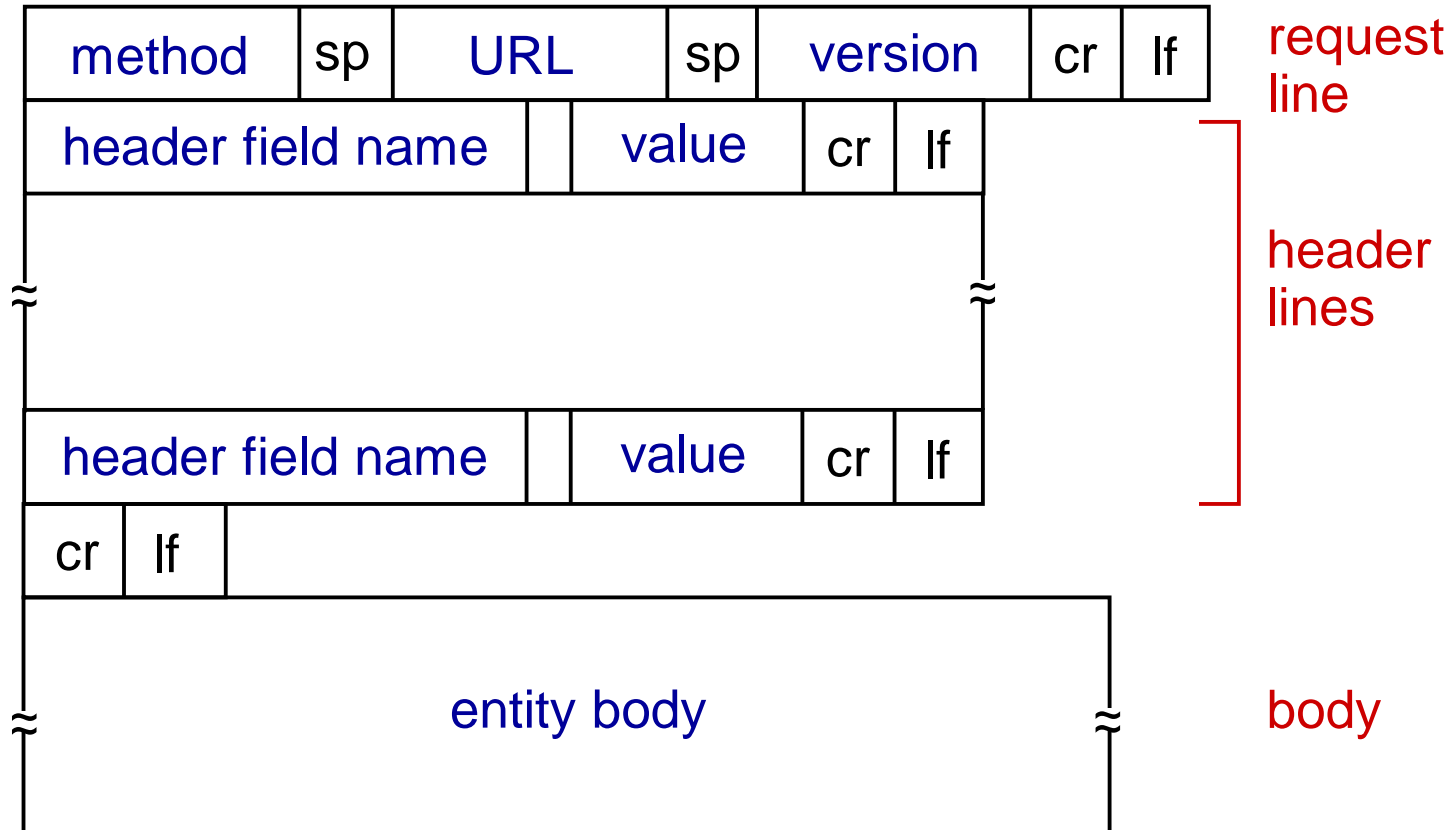 - ASCII (human-readable format)

carriage return character

line-feed character

request line (GET, POST, HEAD commands)

carriage return, line feed
at start of line indicates
end of header lines

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

HTTP request message: general format

| method | sp | URL | sp | version | cr | lf | request line |

| header field name | | value | cr | lf |

| header field name | | value | cr | lf |

header lines

| cr | lf |

| entity body | body |

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

status line (protocol
status code status phrase) ———————➔ **HTTP/1.1 200 OK**

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

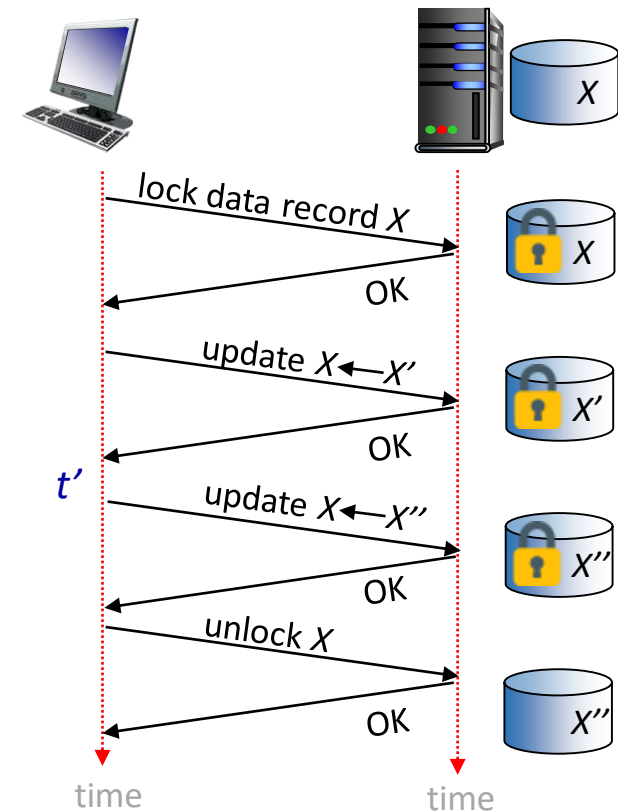- requested document not found on this server

505 HTTP Version Not Supported

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

lock data record X
OK
update X ← X'
OK
t'
update X ← X''
OK
unlock X
OK
time   time

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

Maintaining user/server state: cookies

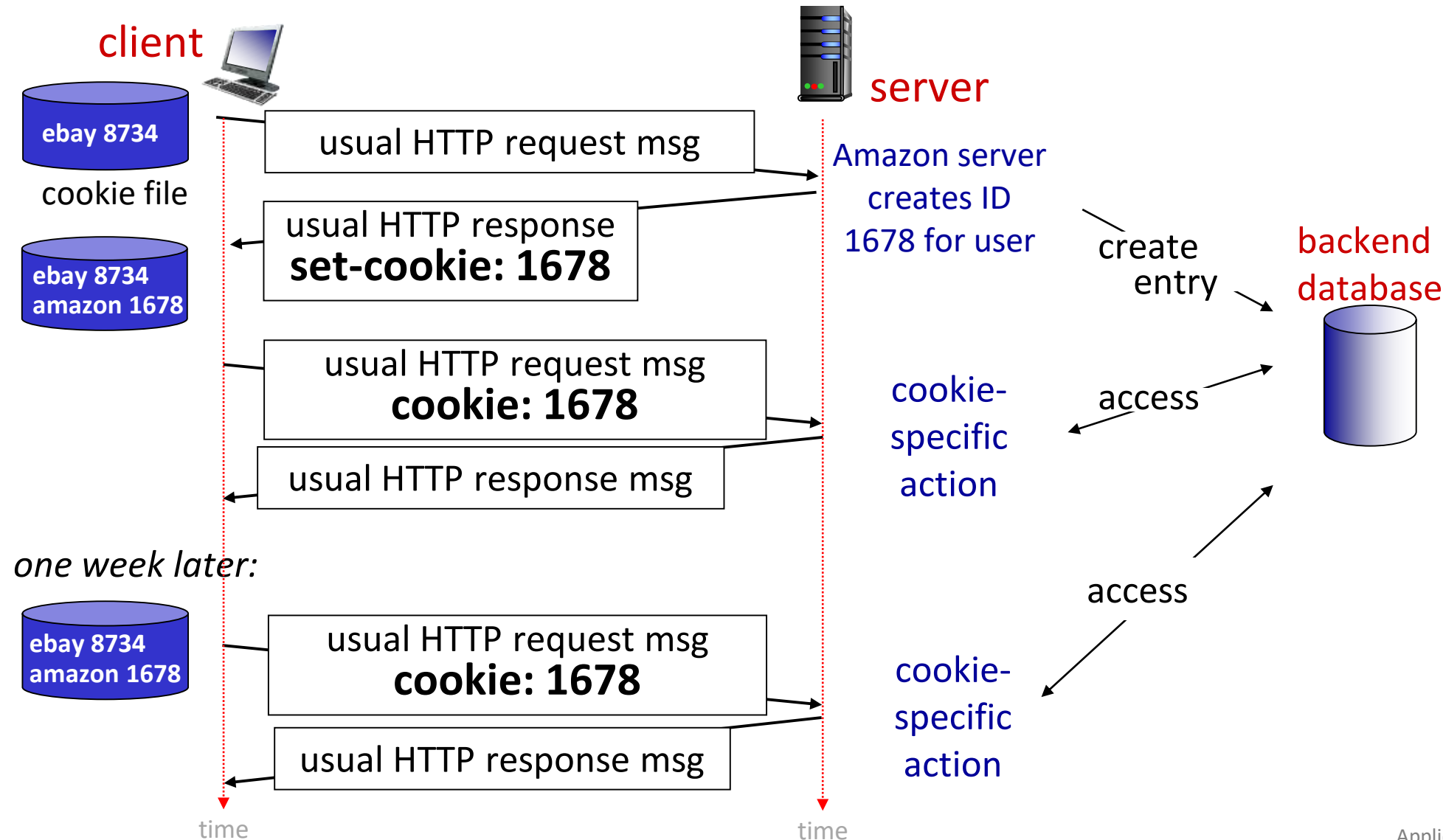Web sites and client browser use *cookies* to maintain some state between transactions

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

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

client

ebay 8734

cookie file

ebay 8734
amazon 1678

server

usual HTTP request msg

Amazon server
creates ID
1678 for user

usual HTTP response
set-cookie: 1678

create
entry

backend
database

usual HTTP request msg
cookie: 1678

cookie-
specific
action

access

usual HTTP response msg

one week later:

ebay 8734
amazon 1678

usual HTTP request msg
cookie: 1678

access

usual HTTP response msg

cookie-
specific
action

time

time

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

Cookies: tracking a user's browsing behavior

Cookies can be used to:

- track user behavior on a given website (first party cookies)
- track user behavior across multiple websites (third party cookies) without user ever choosing to visit tracker site (!)
- tracking may be *invisible* to user:
  - rather than displayed ad triggering HTTP GET to tracker, could be an invisible link

third party tracking via cookies:

- disabled by default in Firefox, Safari browsers
- disabled in Chrome browser in 2023

GDPR (EU General Data Protection Regulation) and cookies

"Natural persons may be associated with online identifiers [...] such as internet protocol addresses, cookie identifiers or other identifiers [...].

This may leave traces which, in particular when combined with unique identifiers and other information received by the servers, may be used to create profiles of the natural persons and identify them."

GDPR, recital 30 (May 2018)

when cookies can identify an individual, cookies are considered personal data, subject to GDPR personal data regulations
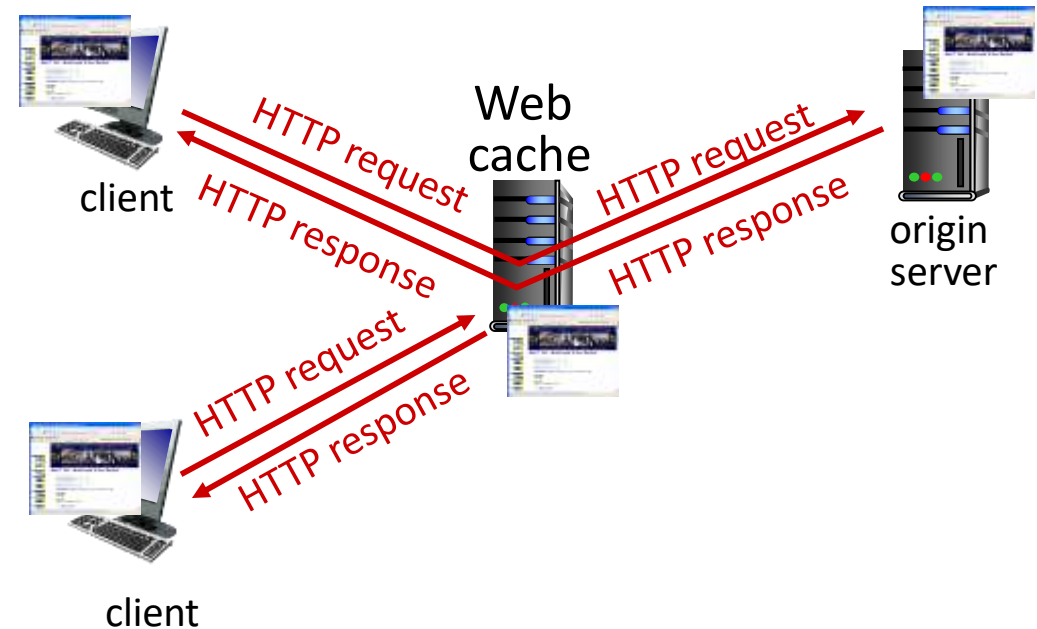

User has explicit control over whether or not cookies are allowed

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
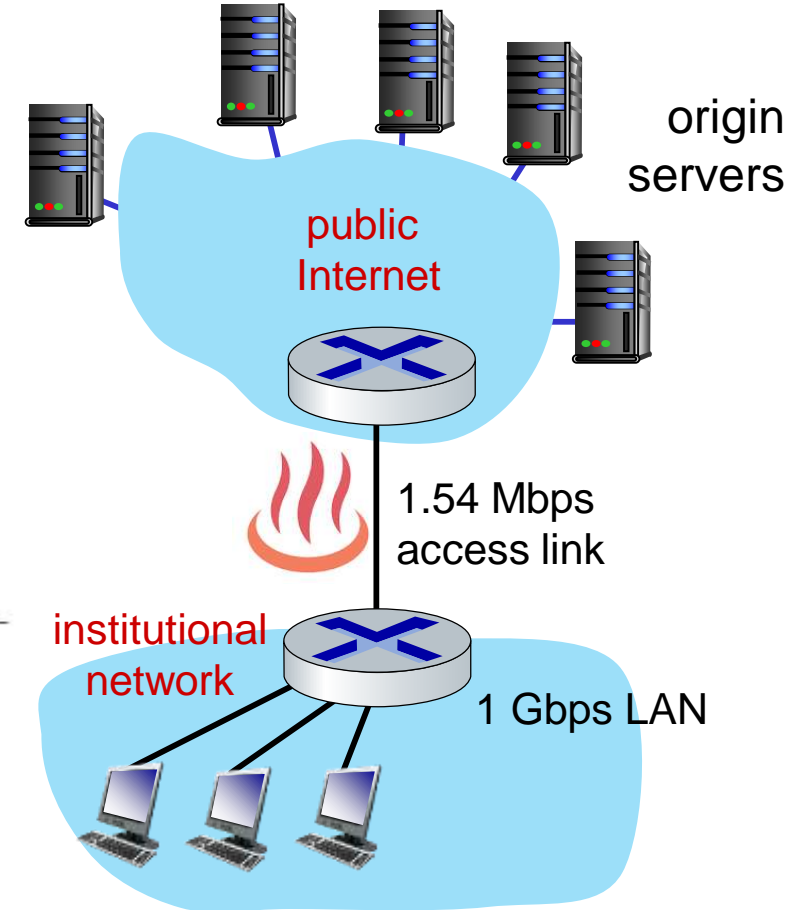
Caching example

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   *problem:* large queueing delays at high utilization!
- LAN utilization: .0015
- end-end delay = Internet delay +
                              access link delay + LAN delay
                    = 2 sec + minutes + usecs

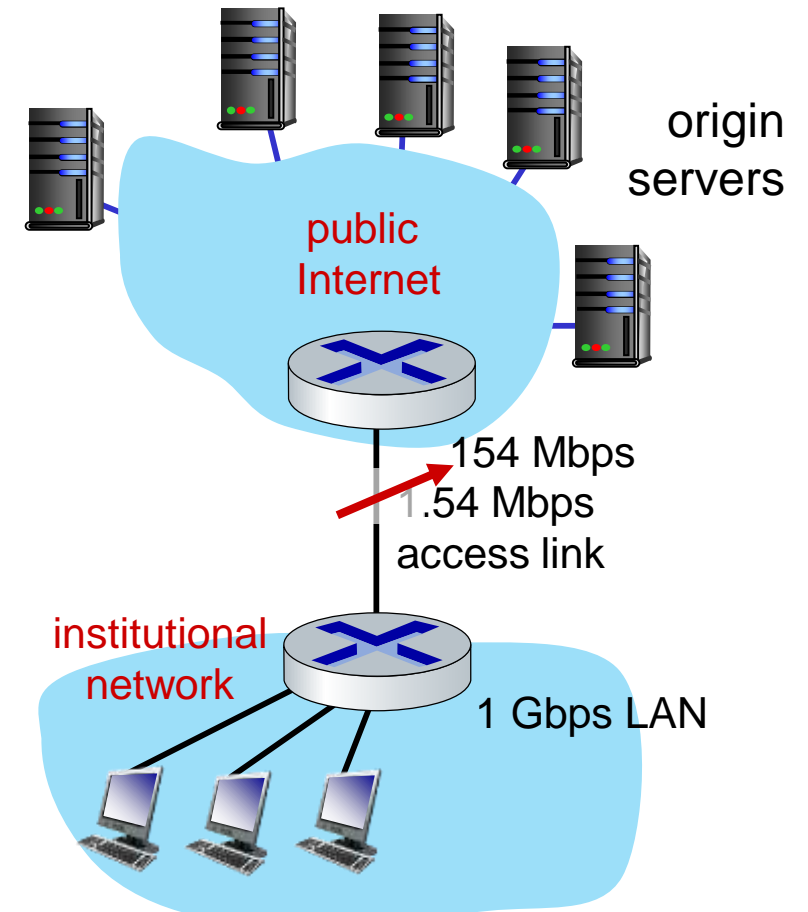# Option 1: buy a faster access link

Scenario:

- access link rate: ~~1.54~~ Mbps   **154 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~~ → **.0097**
- LAN utilization: .0015
- end-end delay = Internet delay + access link delay + LAN delay

  = 2 sec + ~~minutes~~ + usecs   **msecs**

Cost: faster access link (expensive!)

origin servers

public Internet

154 Mbps
1.54 Mbps access link

institutional network

1 Gbps LAN
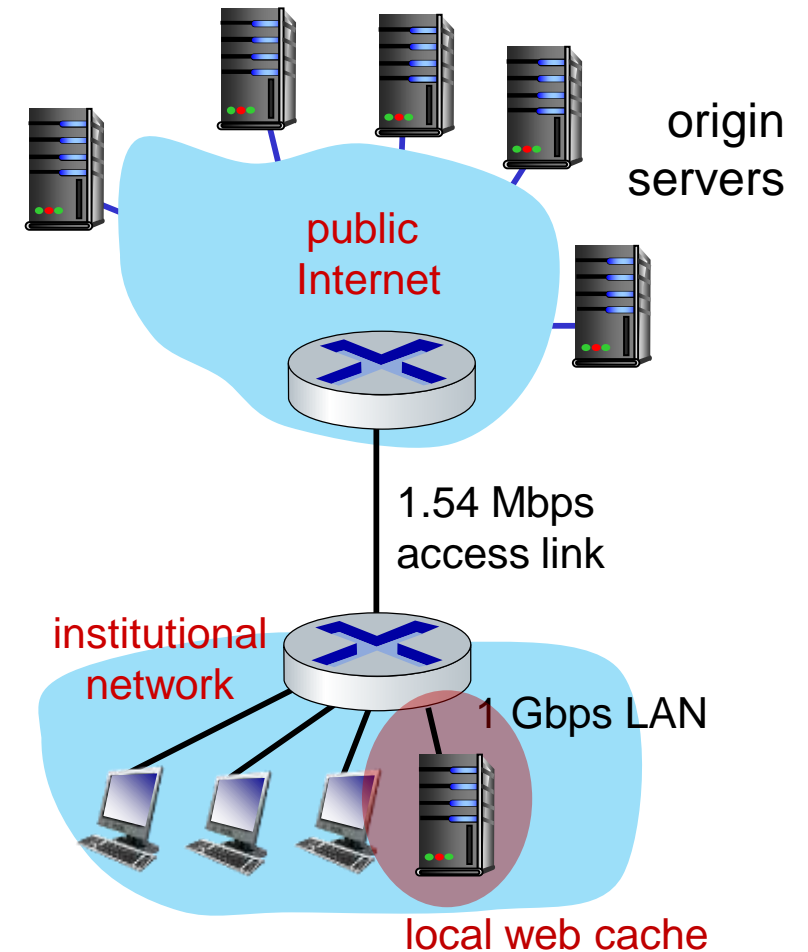
Option 2: install a web cache

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

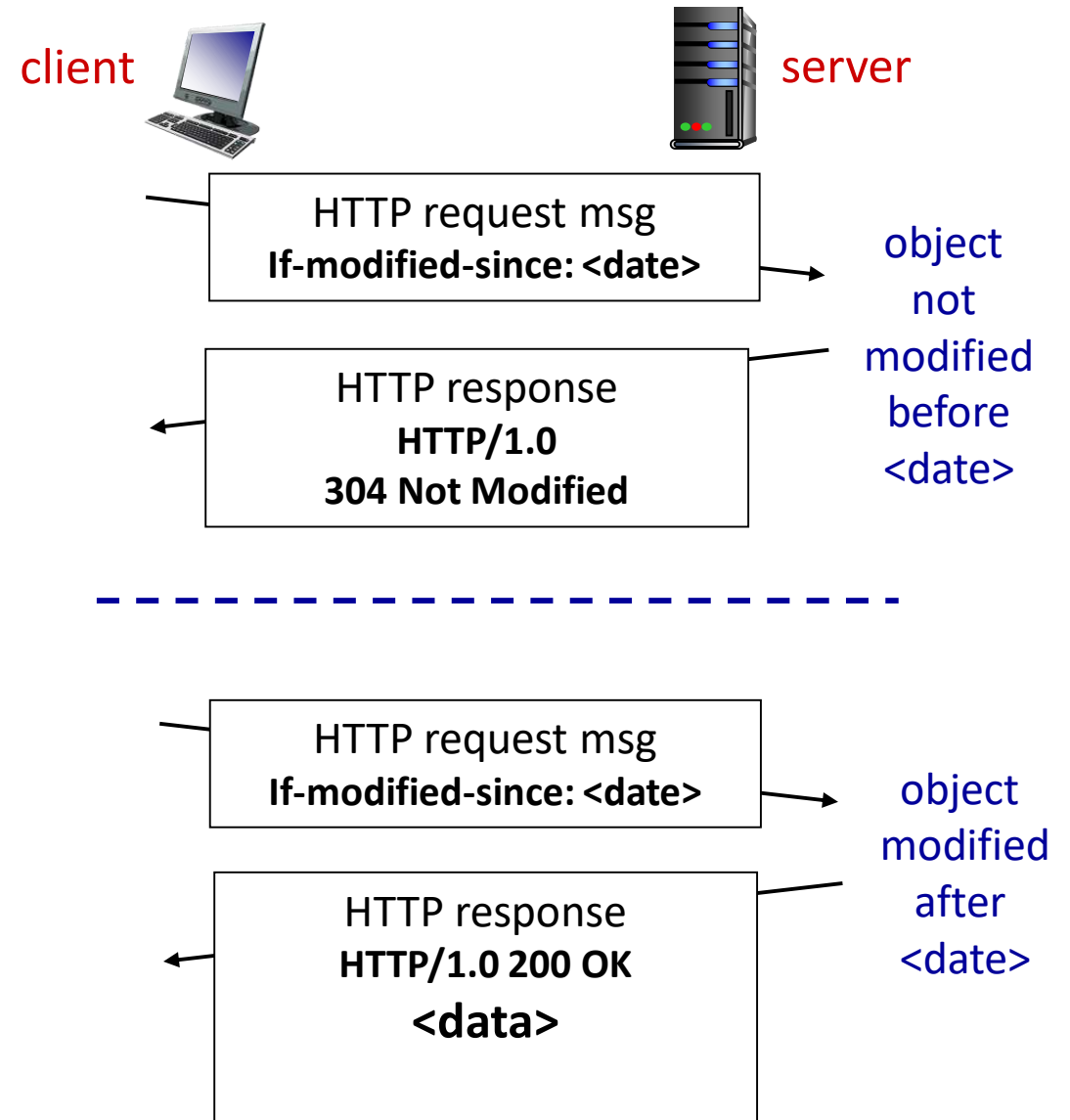*Cost:* web cache (cheap!)

Performance:

- LAN utilization: .?
- access link utilization = ?   *How to compute link utilization, delay?*
- average end-end delay = ?

origin servers

public Internet

1.54 Mbps access link

institutional network

1 Gbps LAN

local web cache

Browser caching: Conditional GET

Goal: don't send object if browser has up-to-date cached version
- no object transmission delay (or use of network resources)

■ *client:* specify date of browser-cached copy in HTTP request
If-modified-since: <date>

■ *server:* response contains no object if browser-cached copy is up-to-date:
HTTP/1.0 304 Not Modified

client   server

HTTP request msg
If-modified-since: <date>

object not modified before <date>

HTTP response
**HTTP/1.0
304 Not Modified**

HTTP request msg
If-modified-since: <date>

object modified after <date>

HTTP response
**HTTP/1.0 200 OK
<data>**

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)
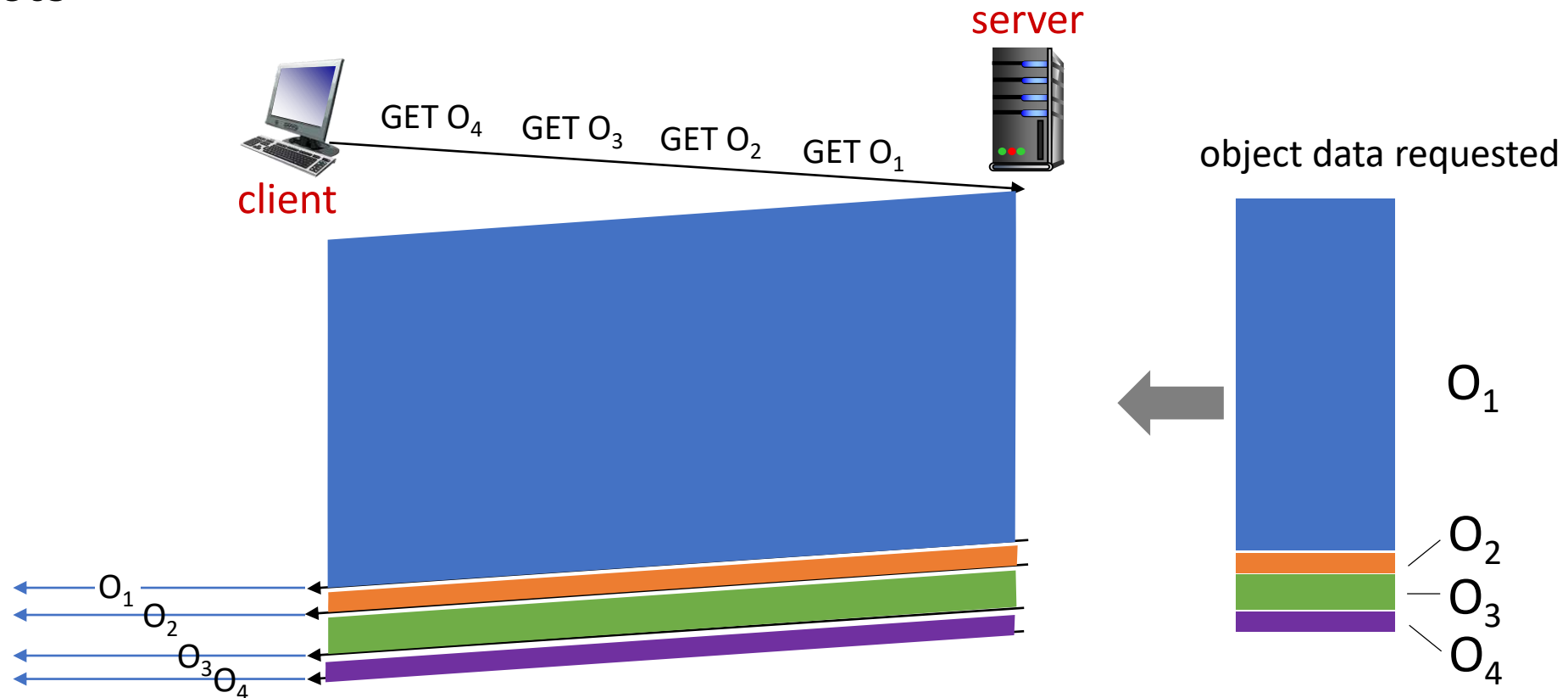- loss recovery (retransmitting lost TCP segments) stalls object transmission

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)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking
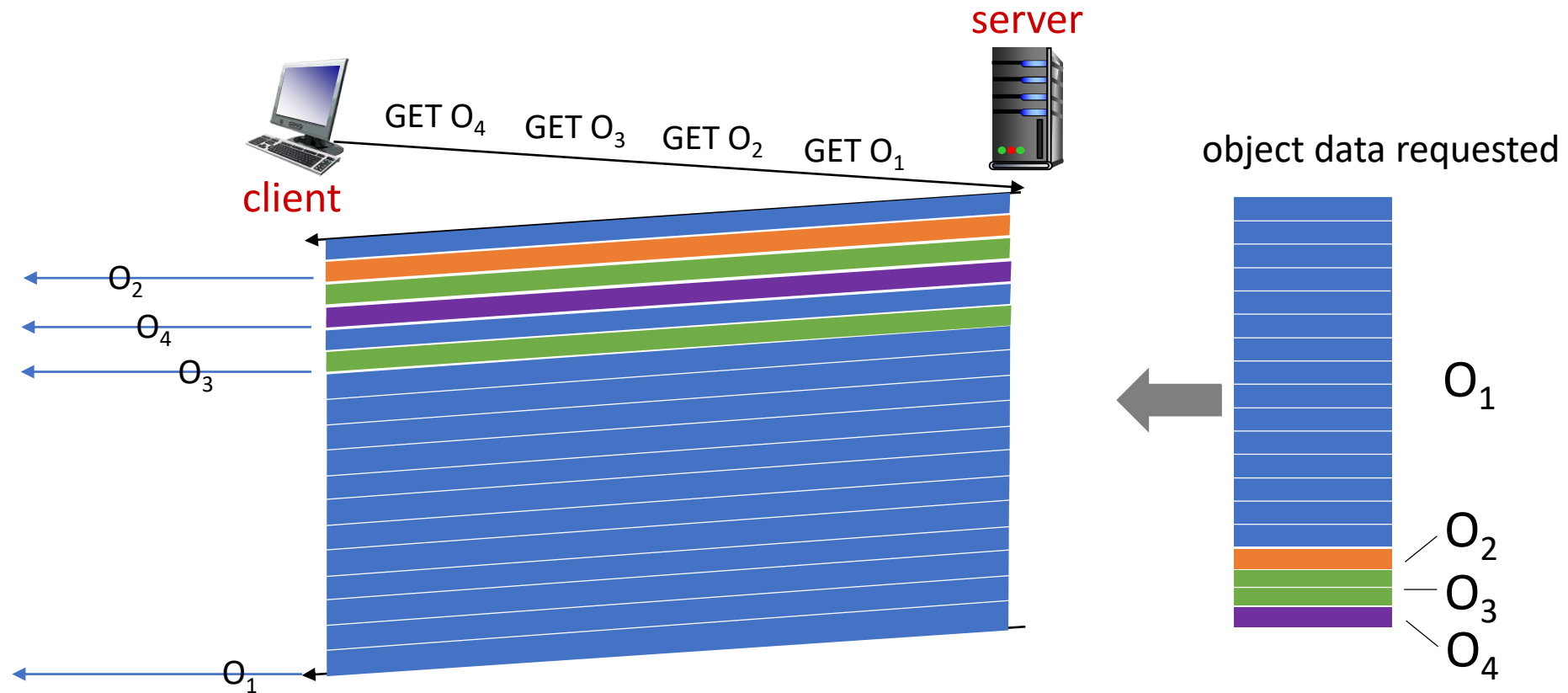
HTTP/2: mitigating HOL blocking

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

- 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