UNIVERSITY OF WATERLOO

CS 456/656
Computer Networks

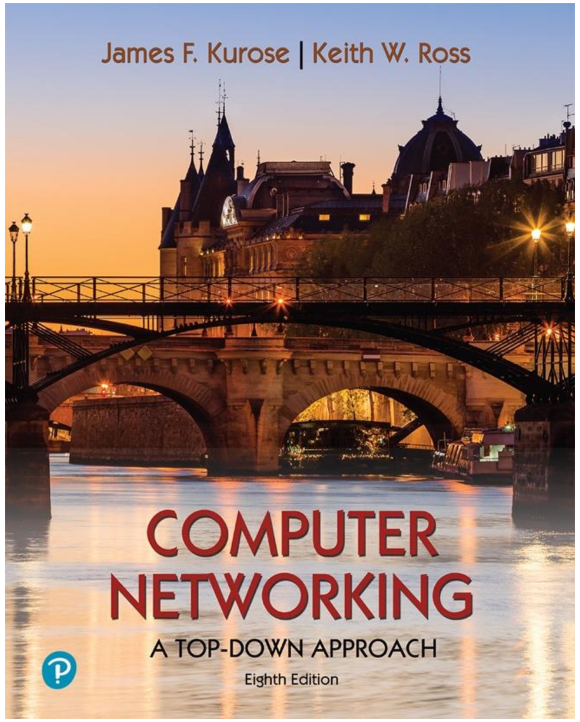
Lecture 3: Application Layer – Part 1

Mina Tahmasbi Arashloo and Bo Sun

Fall 2024

A note on slides

Adapted from the slides that accompany this book.



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Computer networks are complex systems

- They have many pieces
  - Hosts, routers/switches (network devices), links, protocols, …
- They can get quite large
  - Thousands if not millions of hosts and devices
- They are often shared among many traffic flows
- They have to provide many services to distributed applications

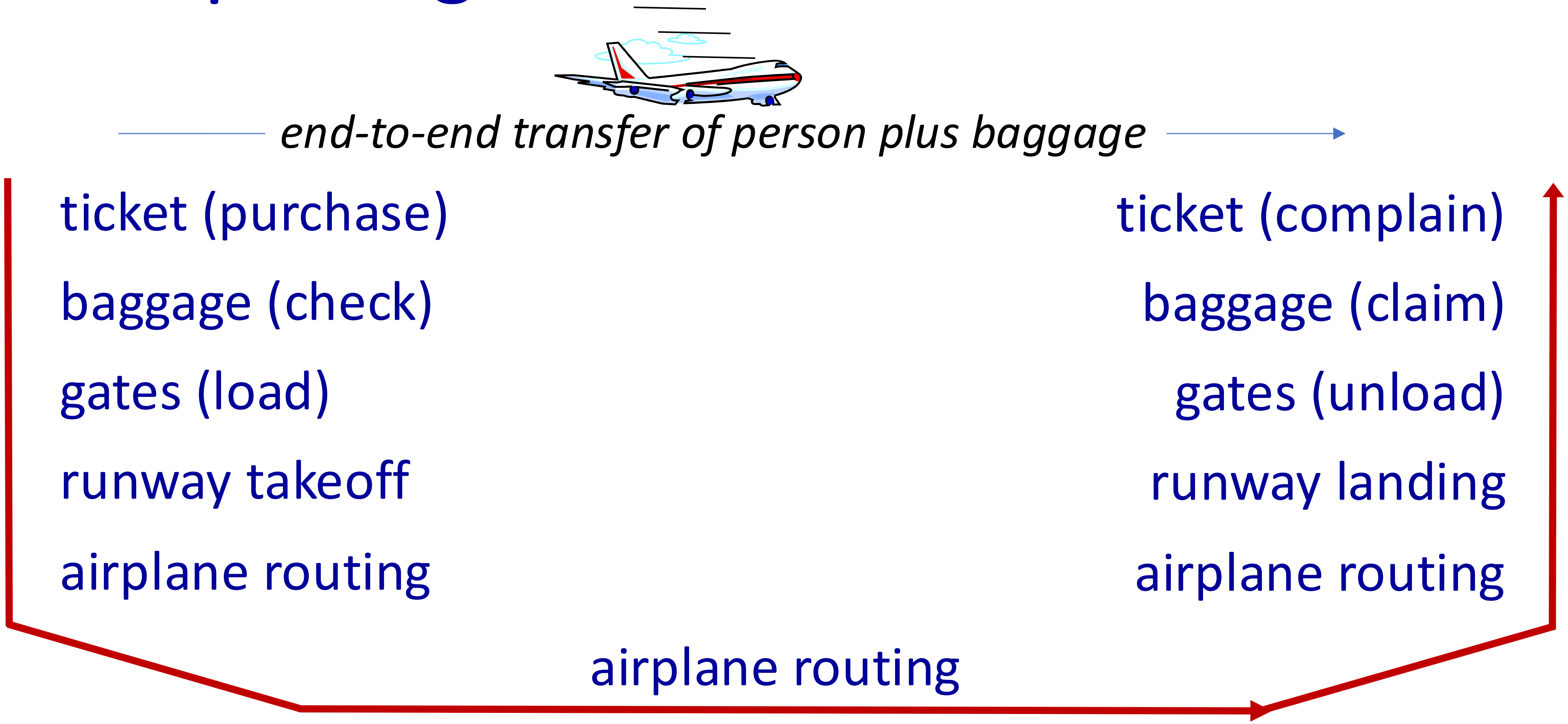
Computer networks are complex systems

- They have many pieces
  - Hosts, routers/switches (network devices), links, protocols, …
- They can get quite large
  - Thousands if not millions of hosts and devices
- They are often shared among many traffic flows
- They have to provide many services to distributed applications

Is there any hope of organizing all the functionality a network should provide?

Let’s look at another complex system for inspiration…

Example: organization of air travel



end-to-end transfer of person plus baggage

ticket (purchase)

baggage (check)

gates (load)

runway takeoff

airplane routing

ticket (complain)

baggage (claim)

gates (unload)

runway landing

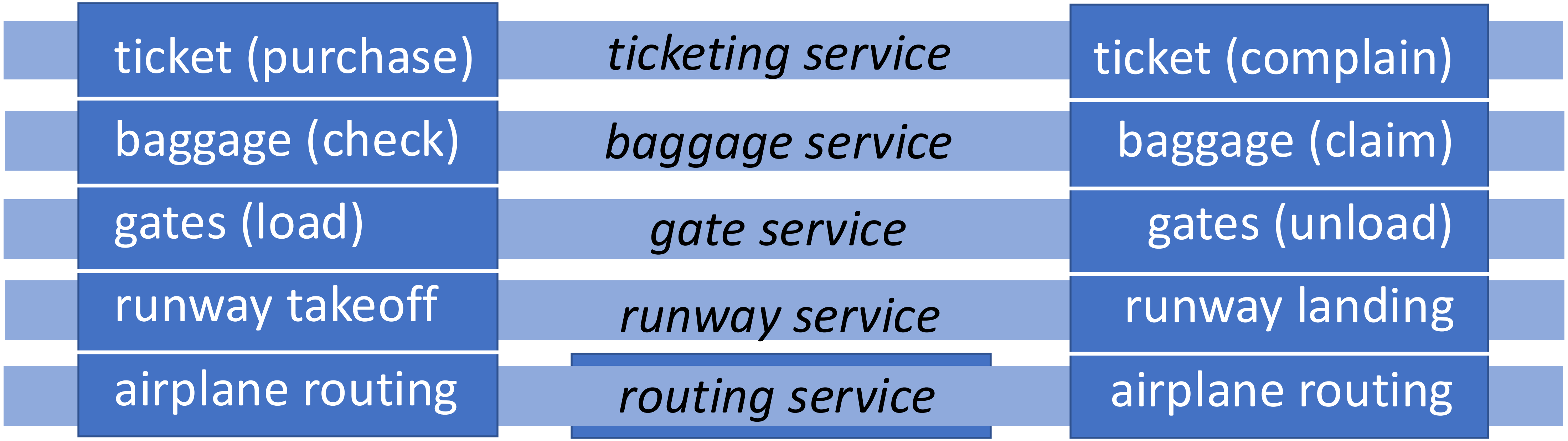
airplane routing

airplane routing

How would you *define/discuss* the *system* of airline travel?

- a series of steps, involving many services

Example: organization of air travel



ticket (purchase)	*ticketing service*	ticket (complain)
baggage (check)	*baggage service*	baggage (claim)
gates (load)	*gate service*	gates (unload)
runway takeoff	*runway service*	runway landing
airplane routing	*routing service*	airplane routing

layers: each layer implements a service

- via its own internal-layer actions
- relying on services provided by layer below

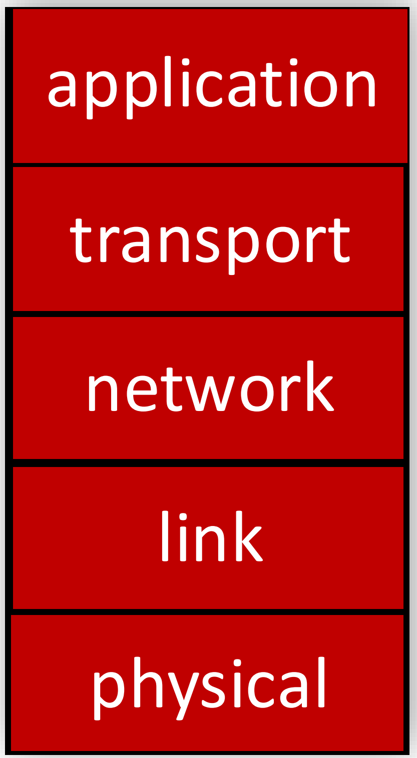
Why layering?

Approach to designing/discussing complex systems:

- explicit *structure* allows identification of system’s pieces and their relationships
  - layered *reference model* for discussion
- *modularization* eases maintenance and 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

The layered Internet protocol 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
- *data link:* data transfer between neighboring network elements
  - Ethernet, 802.11 (WiFi), PPP
- *physical:* bits "on the wire"



application

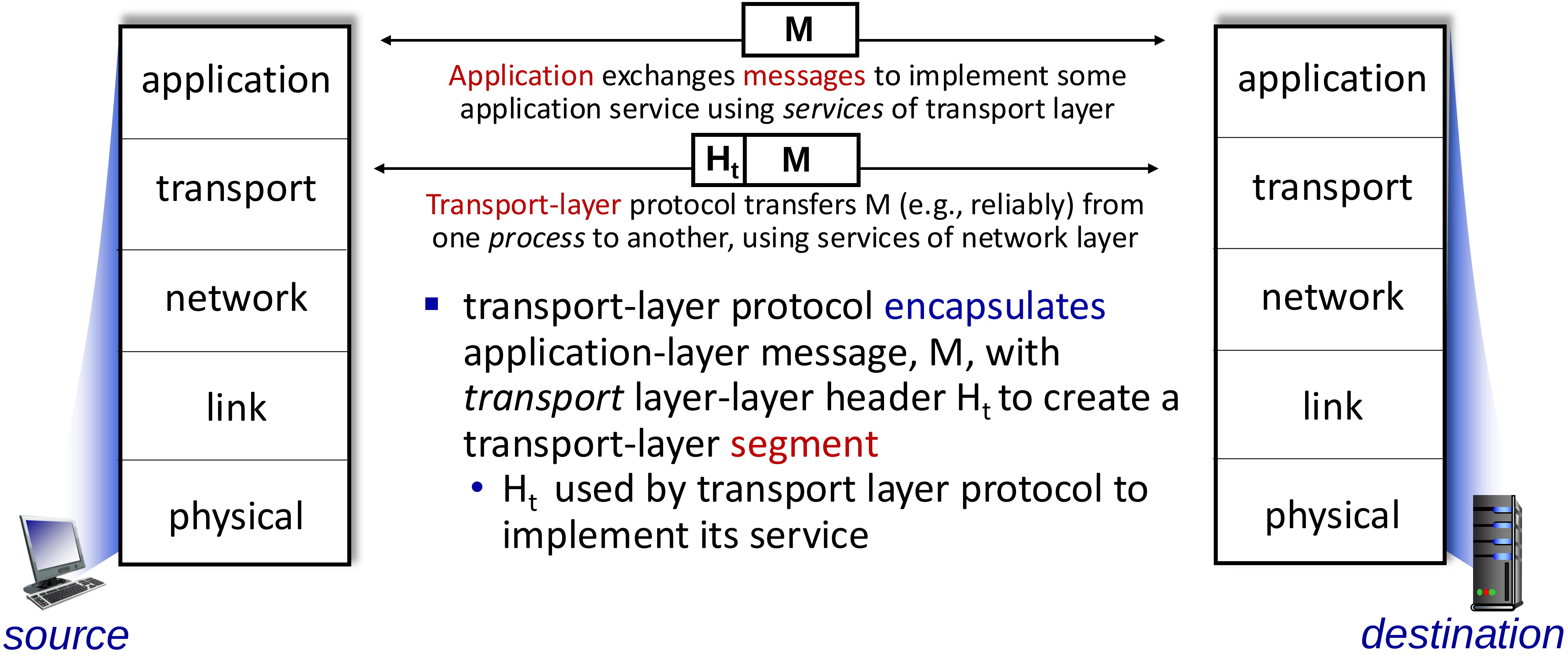
transport

network

link

physical

Services, Layering and Encapsulation

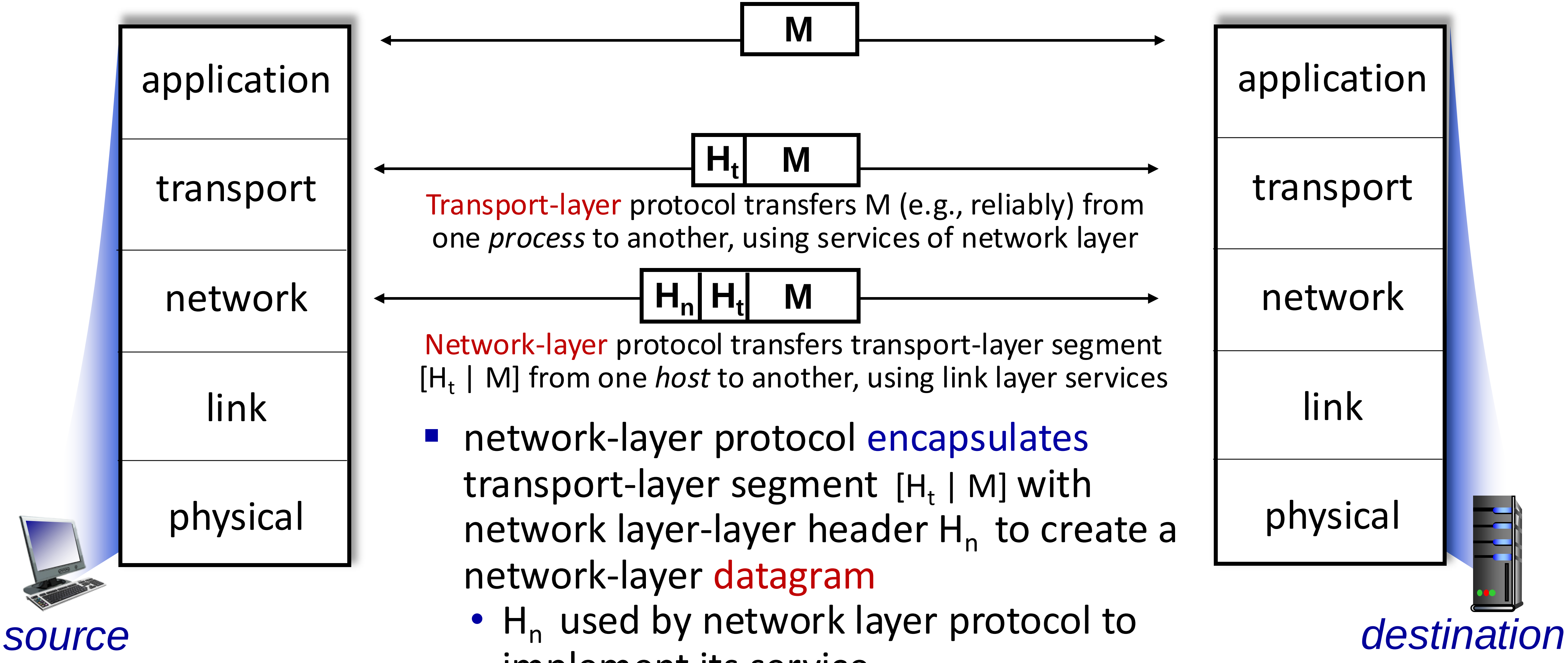


Application exchanges messages to implement some application service using *services* of transport layer

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

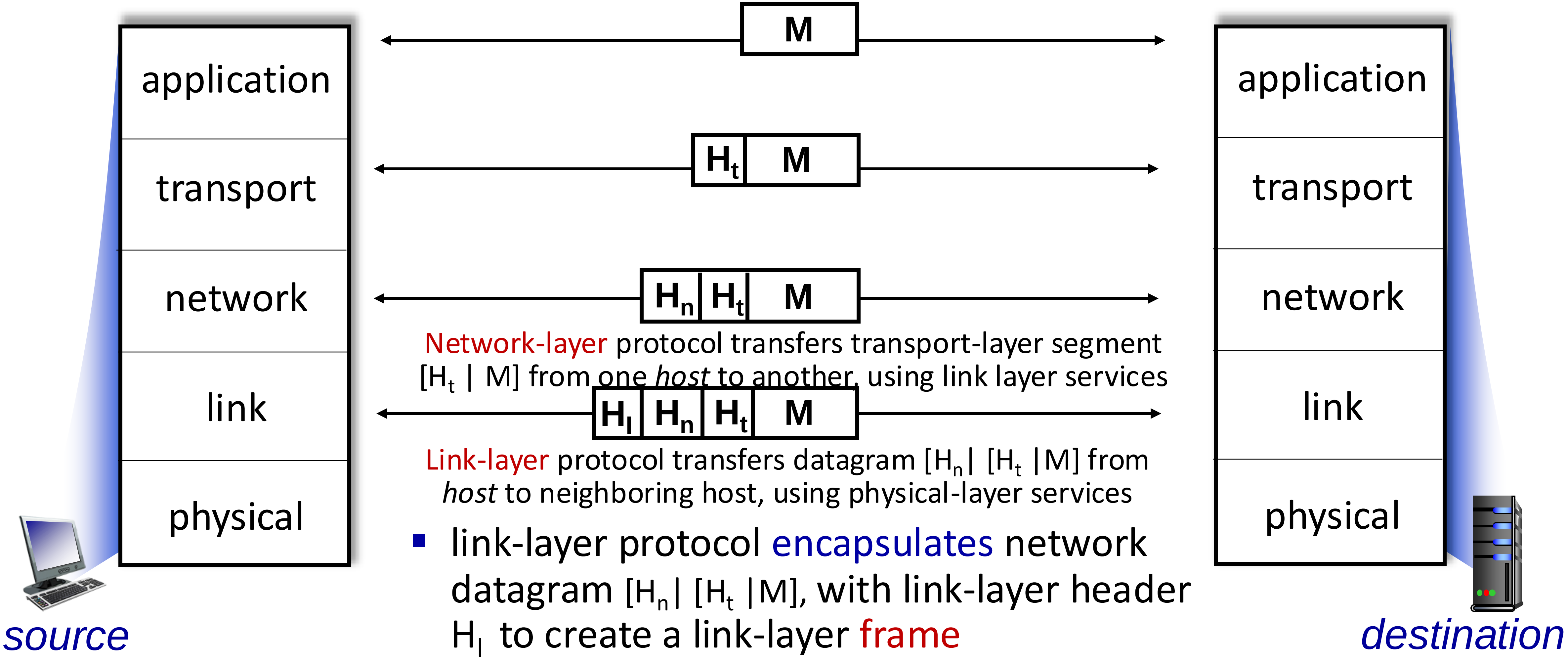


Transport-layer protocol transfers M (e.g., reliably) from one *process* to another, using services of network layer

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

- network-layer protocol encapsulates transport-layer segment [H_t | 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

Services, Layering and Encapsulation



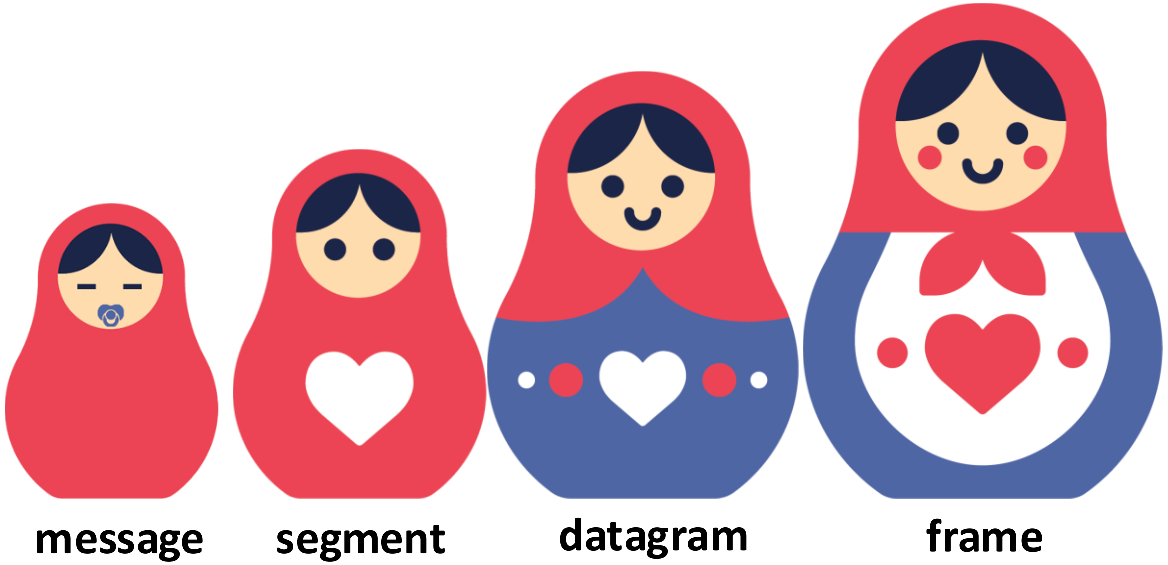
Network-layer protocol transfers transport-layer segment [H_t | M] from one *host* to another, using link layer services

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

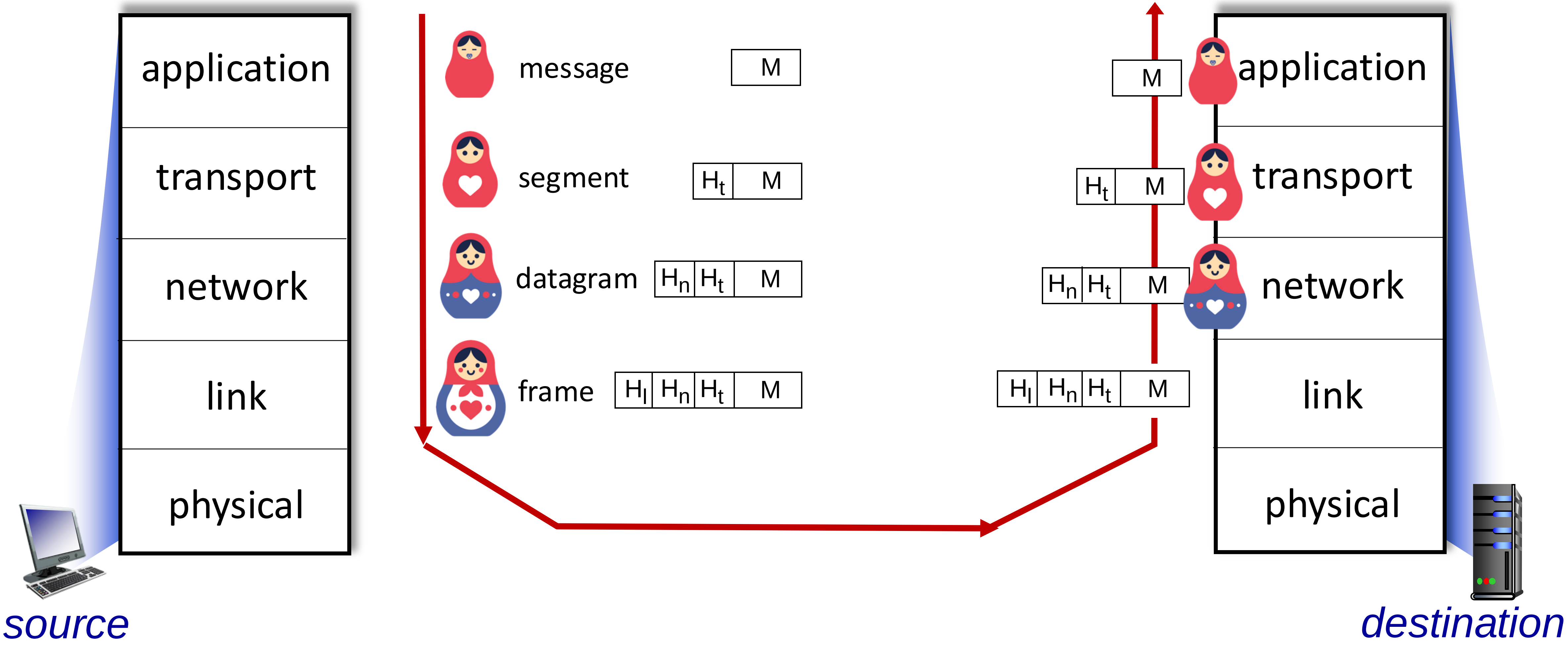
Encapsulation

Matryoshka dolls (stacking dolls)

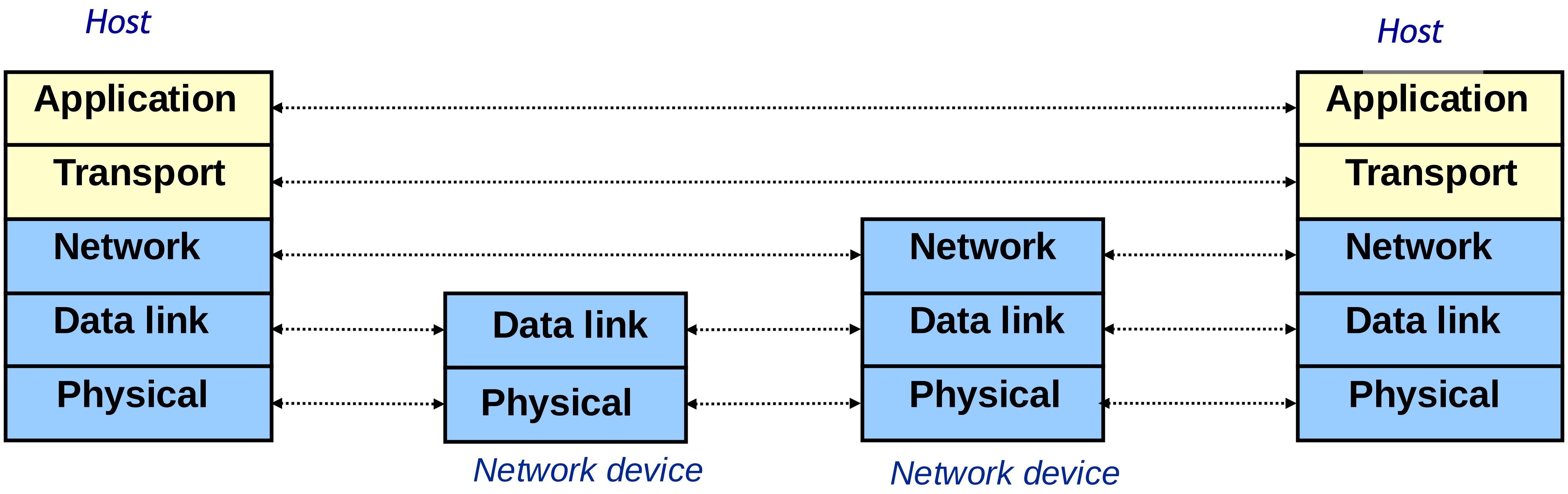


Credit: https://dribbble.com/shots/7182188-Babushka-Boi

Common Layers in Today’s Networks

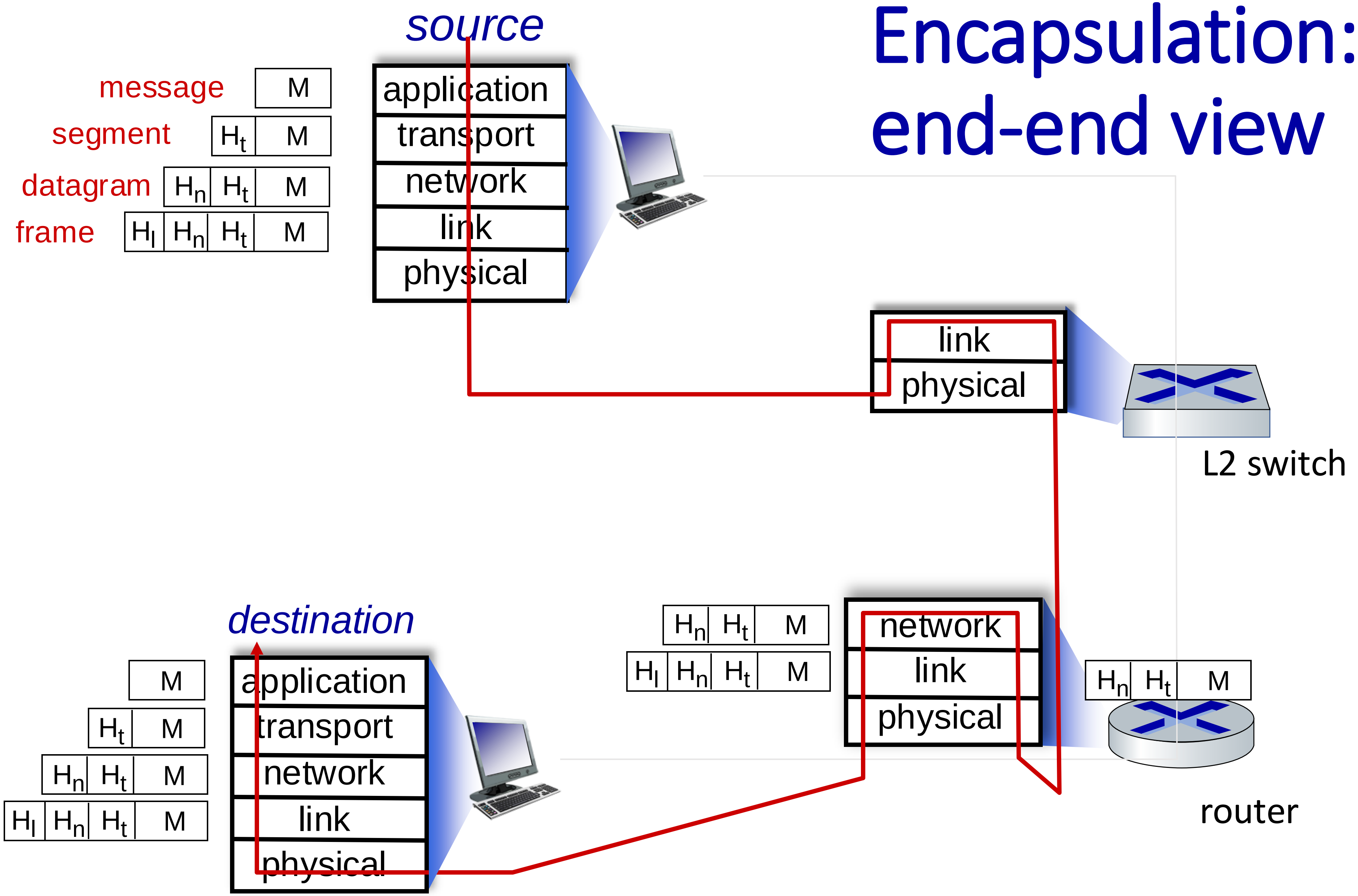


Common Layers in Today's Networks



- The end-hosts typically implement all layers of the stack.
- Depending on their functionality, devices in the network implement all or a subset of the layers.

Encapsulation: an end-end view



source

message M

segment H_t M

datagram H_n H_t M

frame H_l H_n H_t M

application

transport

network

link

physical

link

physical

L2 switch

H_n H_t M

H_l H_n H_t M

network

link

physical

H_n H_t M

router

destination

M

H_t M

H_n H_t M

H_l H_n H_t M

application

transport

network

link

physical

We will study networks one layer at a time

- For the next several weeks, we will discuss the common layers in today’s networks
- Starting from the top -- application layer
- All the way to the data link layer

We will study networks one layer at a time

- The Application Layer
- The Transport Layer
- The Network Layer
- The Data Link Layer

We will study networks one layer at a time

- <u>The Application Layer</u>
- The Transport Layer
- The Network Layer
- The Data Link Layer

Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
- learn about protocols by examining popular application-layer protocols and infrastructure
- programming network applications
  - socket API

Some network apps

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

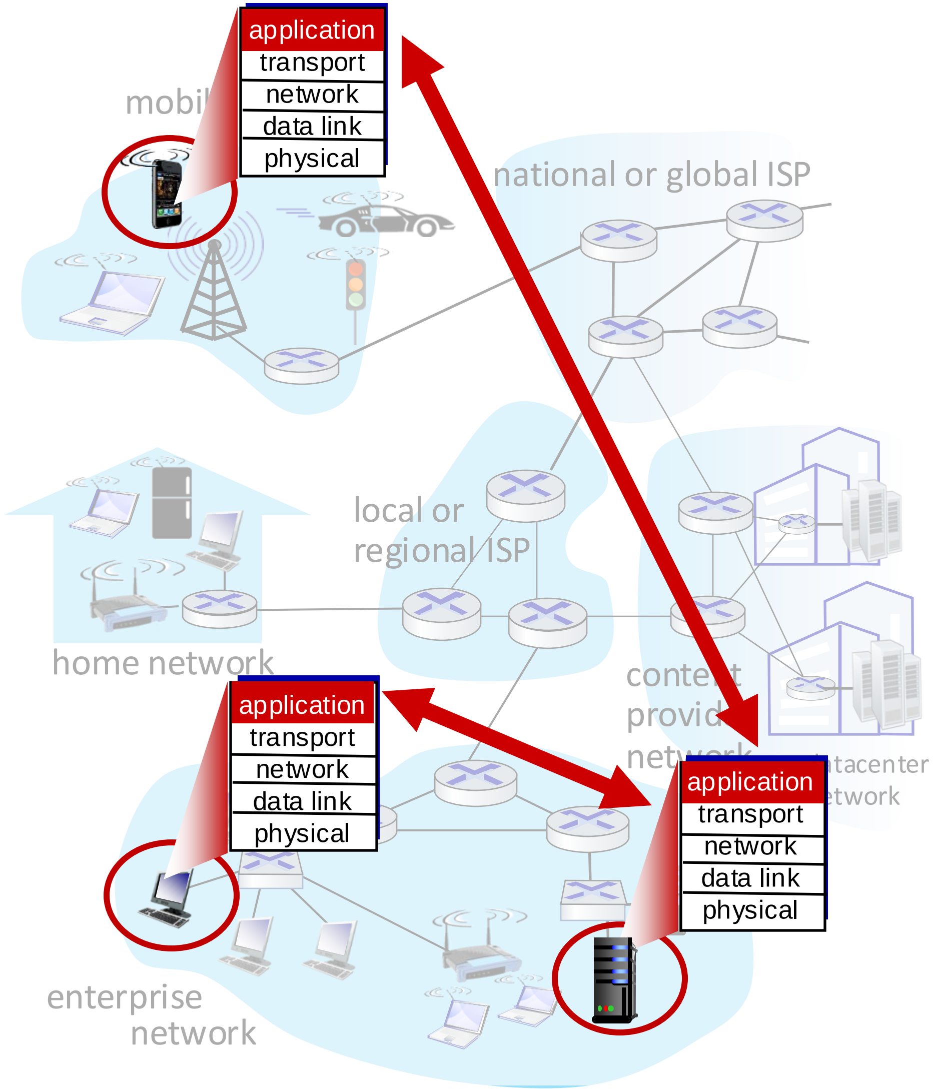
Creating a network app

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 and propagation



Client-server paradigm

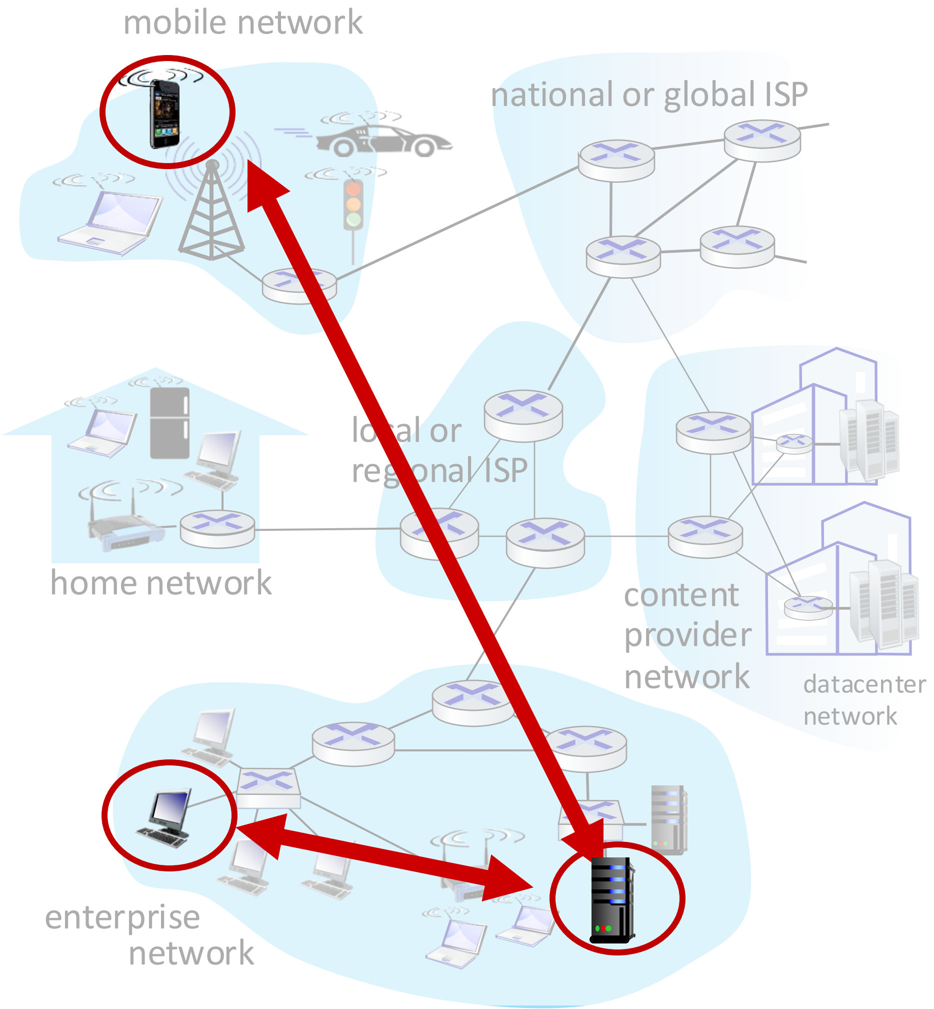
server:

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

clients:

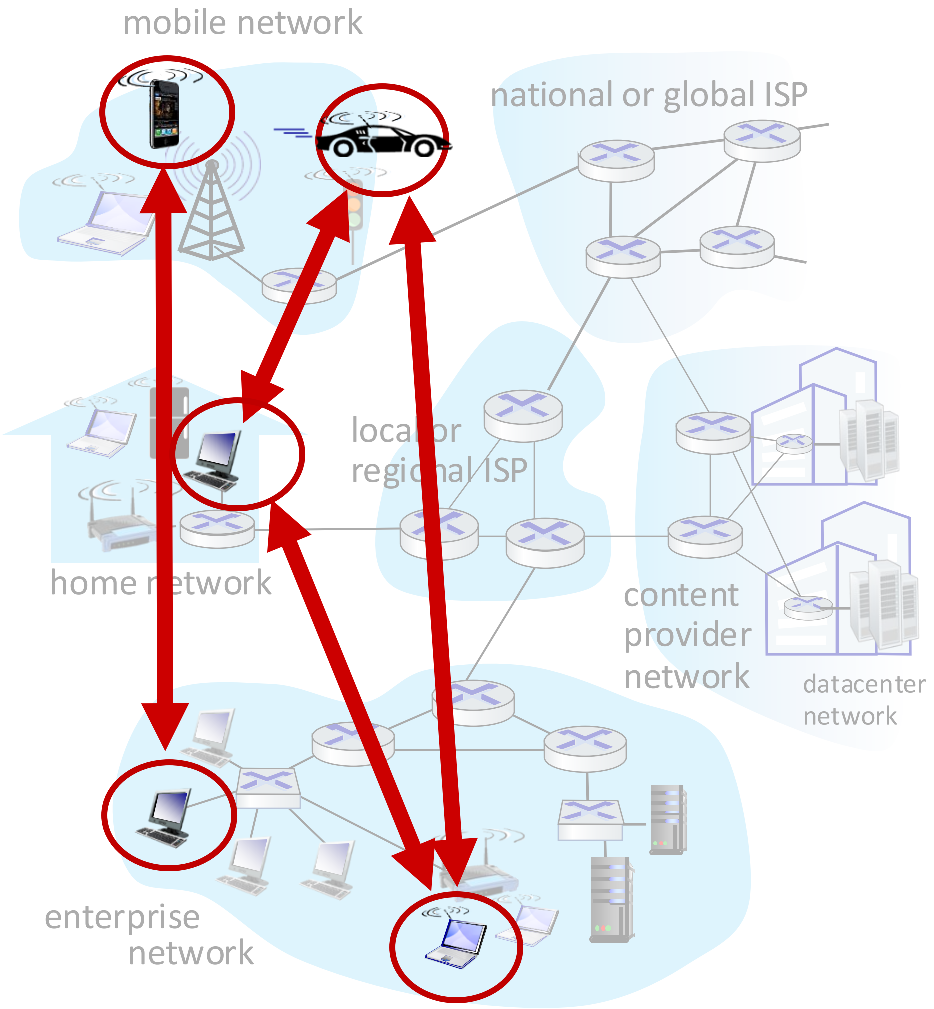
- contact, communicate with server
- may be intermittently connected
- may have dynamic network addresses
- do *not* communicate directly with each other

- examples: Web applications



Peer-to-peer (P2P) architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change network addresses
  - complex management
- example: P2P file sharing [BitTorrent]



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 vs proprietary protocols

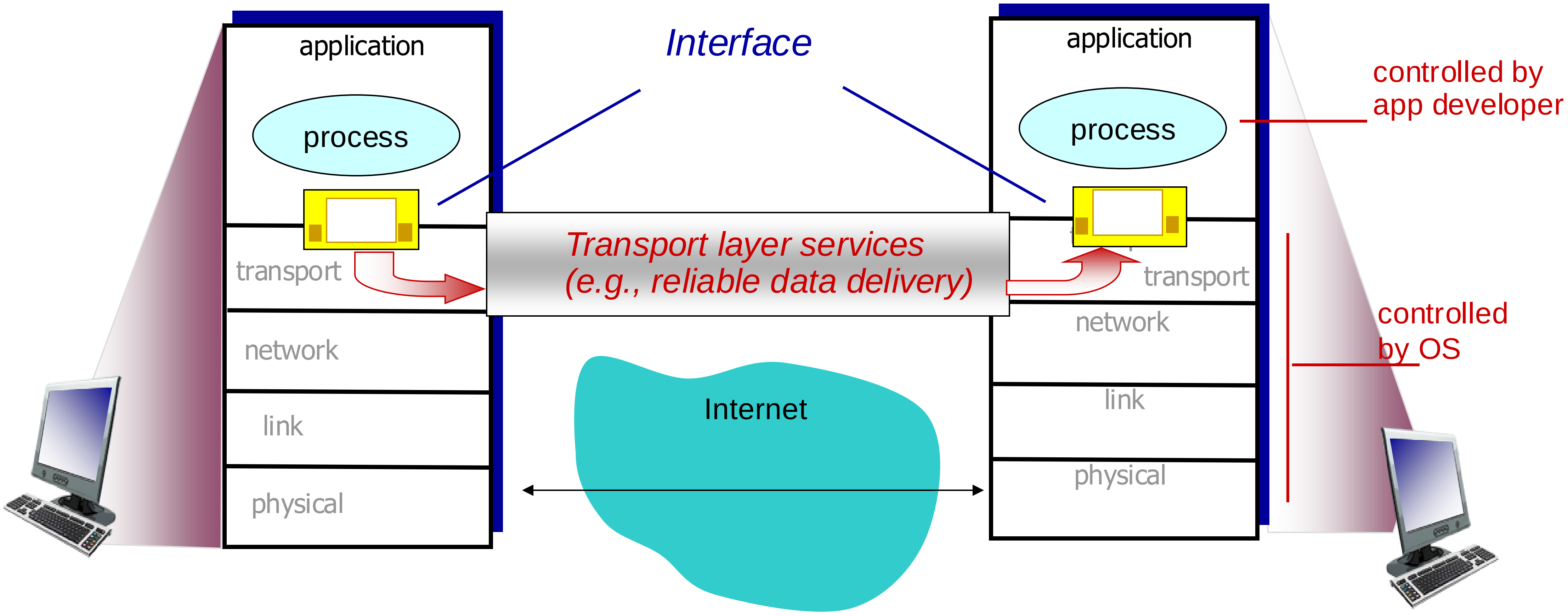
open protocols:

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

proprietary protocols:

- e.g., Skype, Zoom

The application layer relies on the transport layer



What transport service may an app need?

data integrity

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

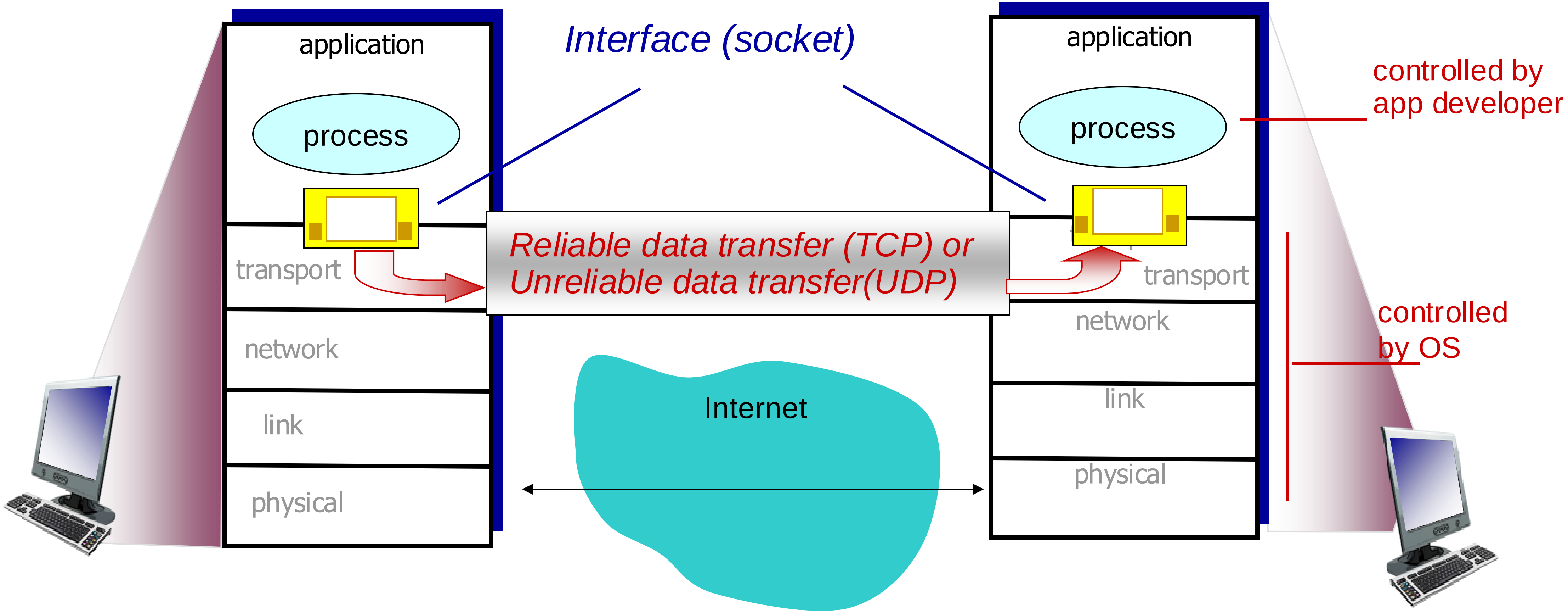
security

- encryption, data integrity, …

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
text messaging	no loss	elastic	yes and no

Internet applications rely on Internet transport protocols



Internet transport protocols services

Reliable connection-based service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *connection-oriented:* setup required between client and server processes
- *does not provide:* timing, minimum throughput guarantee, security

Unreliable connection-less service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Example: TCP

t tr

Q: why bother with this one?
Why is there a UDP?

vices

Example: UDP

Reliable connection-based service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won’t overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *connection-oriented:* setup required between client and server processes
- *does not provide:* timing, minimum throughput guarantee, security

Unreliable connection-less service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Internet applications use Internet transport protocols

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP [RFC 7230, 9110]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7230], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Examples applications we will discuss

- Web applications: client-server
- E-Mail: client-server
- Video streaming: client-server
- P2P file distribution: peer-to-peer

Example 1: Web applications

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.,

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

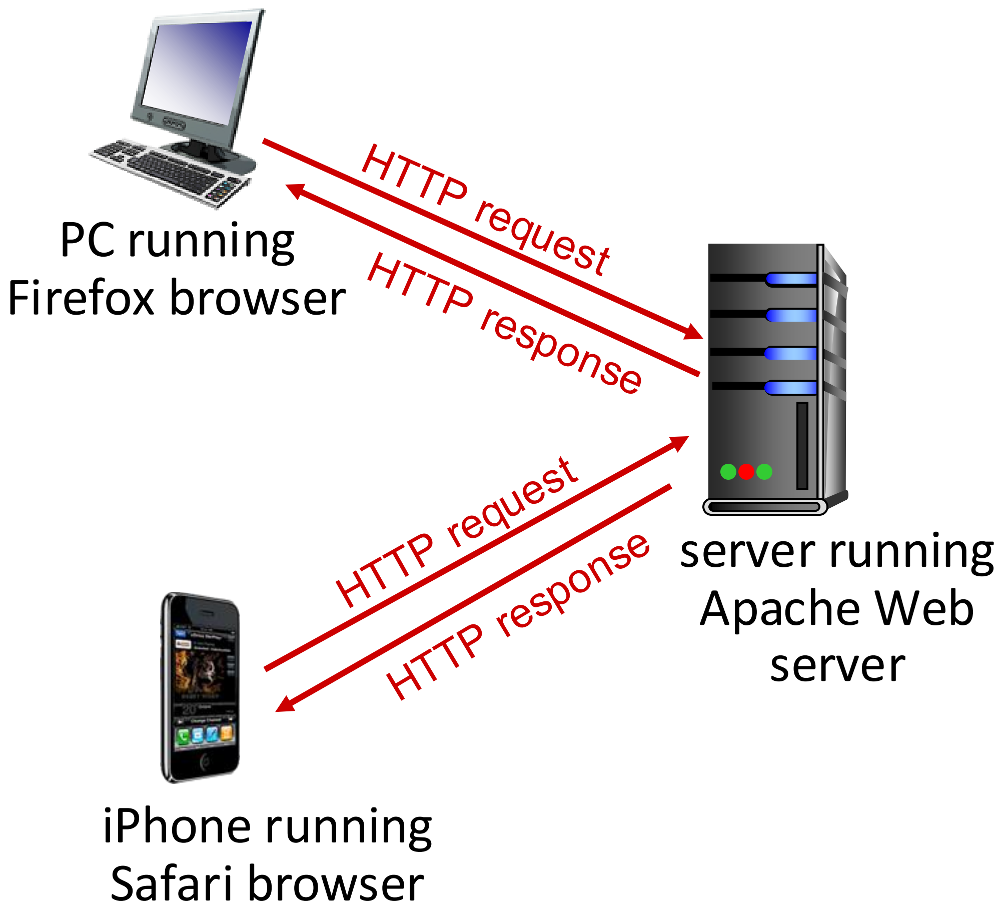
In-class Exercise

- Suppose you want to implement a simple web server and a browser
- The user will enter the URL to the object they want to access
  - *Say, the HTML file for https://cs.uwaterloo.ca/*
- The file is stored in a server in the CS department, where your web server program is also running
- Your browser should retrieve the file and display it.
- How do you have the browser and server coordinate to retrieve the file?

HTTP overview

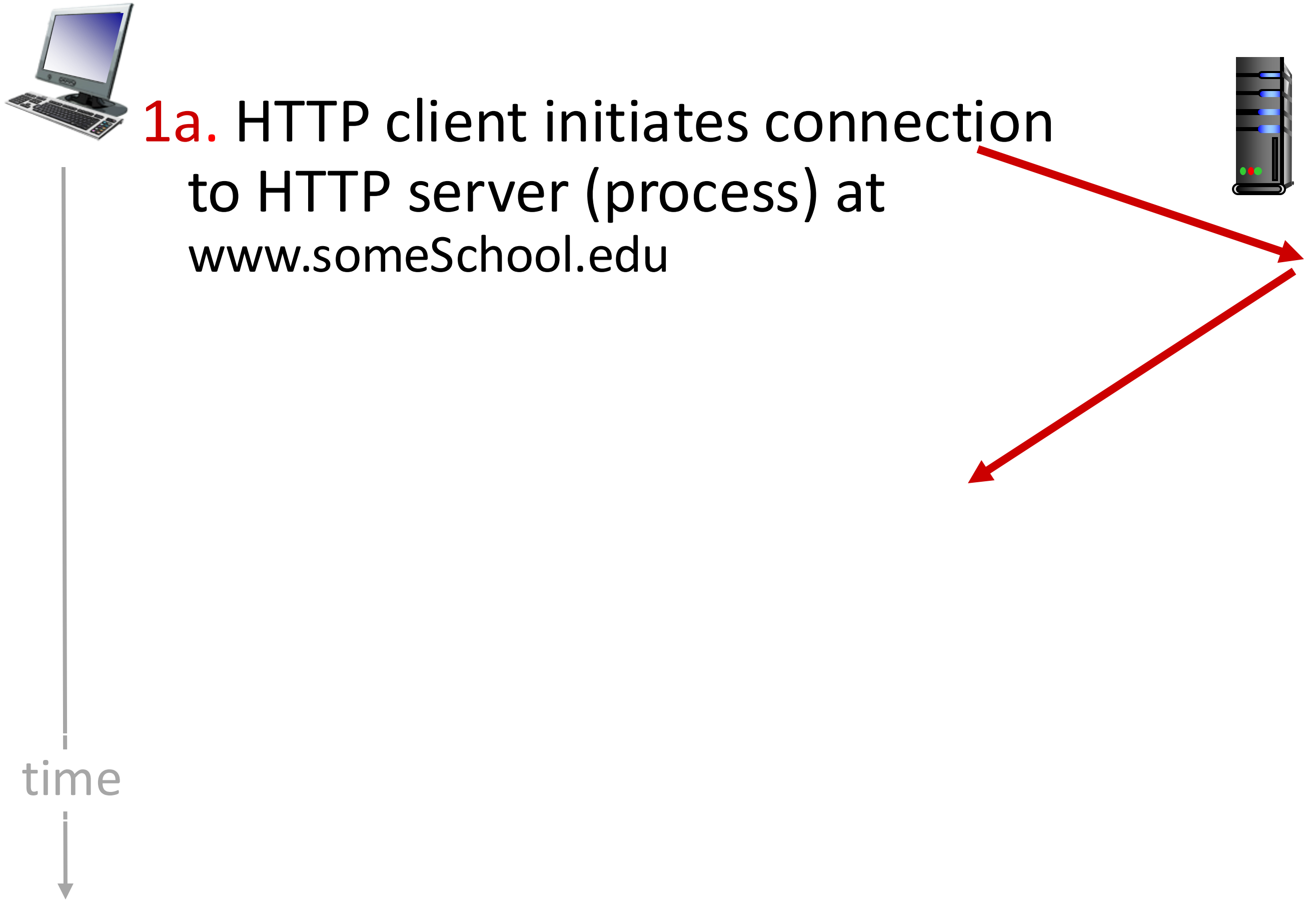
HTTP: hypertext transfer protocol

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

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**

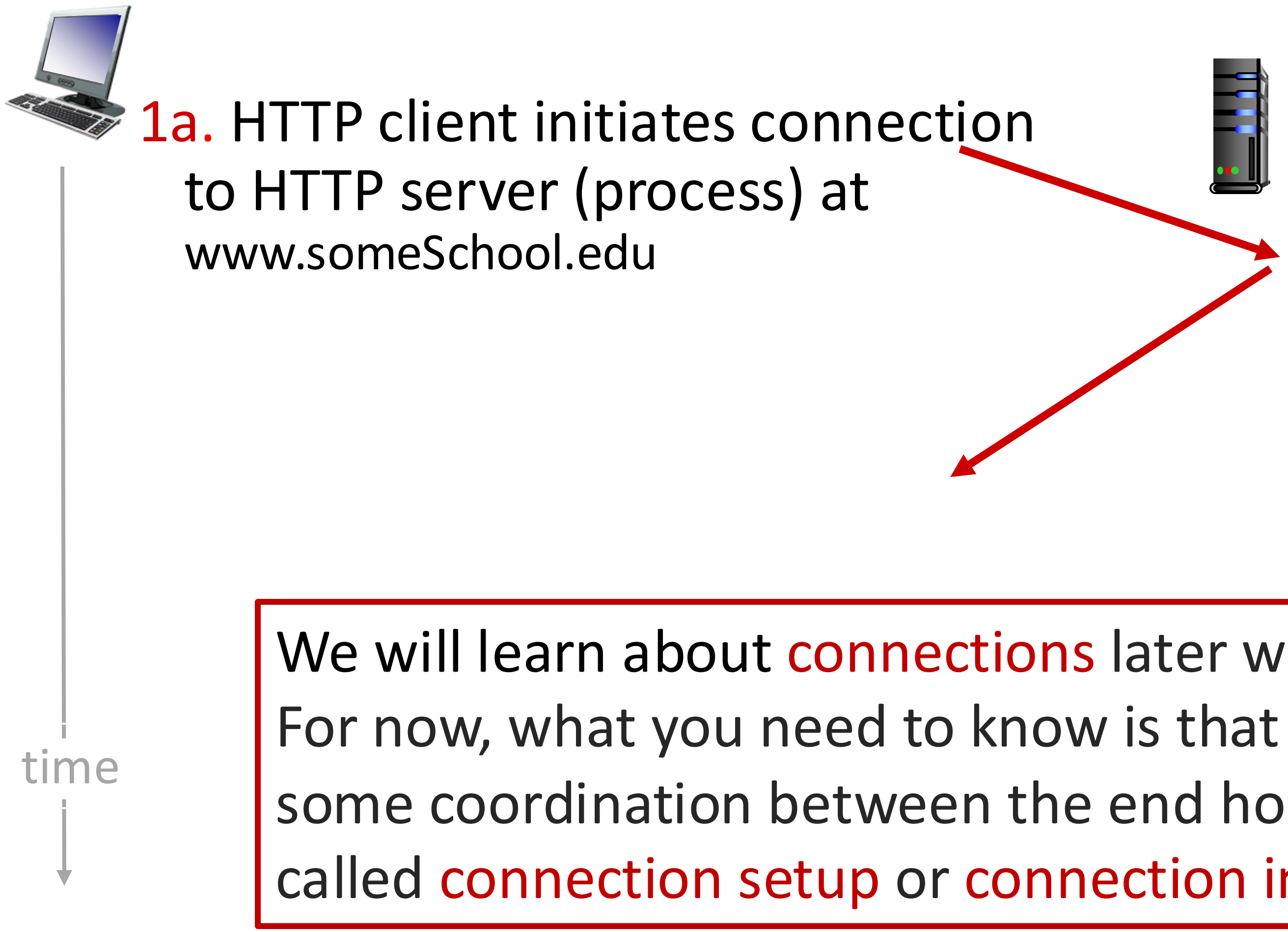


1a. HTTP client initiates connection to HTTP server (process) at www.someSchool.edu

1b. HTTP server at host www.someSchool.edu "accepts" connection, notifying client

HTTP example

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**



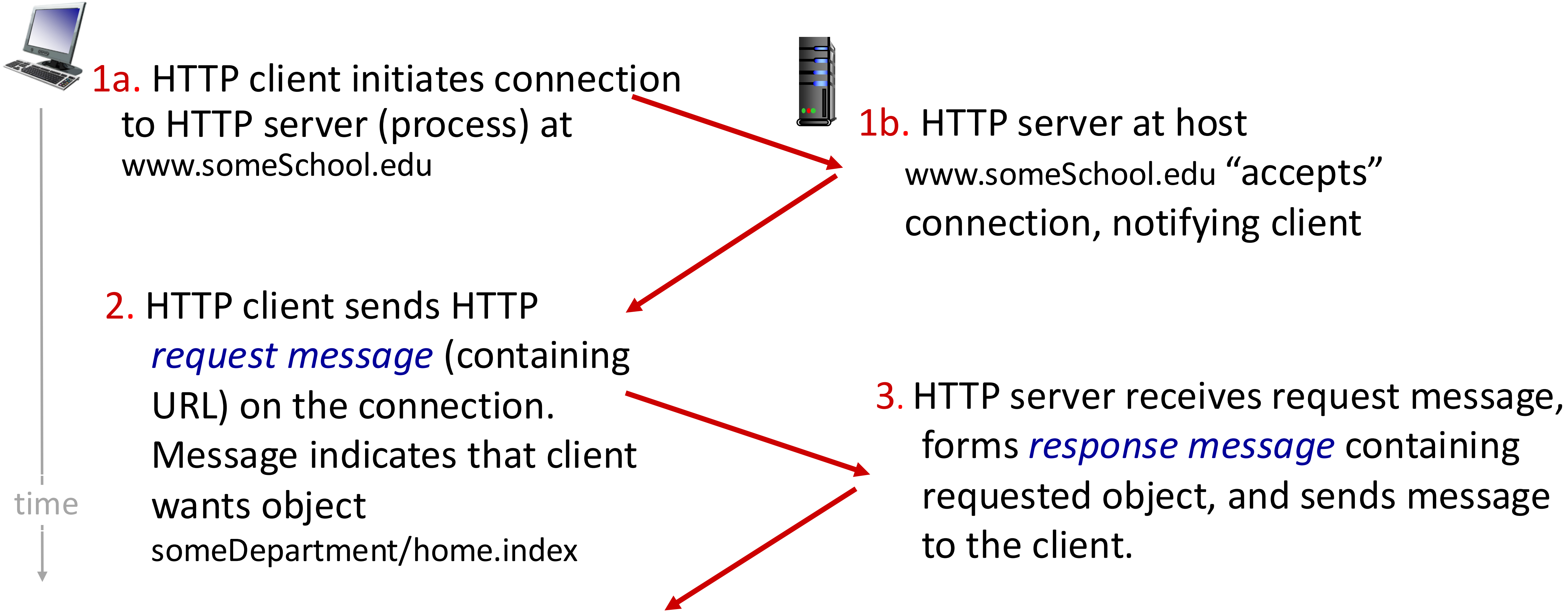
1a. HTTP client initiates connection to HTTP server (process) at www.someSchool.edu

1b. HTTP server at host www.someSchool.edu “accepts” connection, notifying client

We will learn about connections later when we discuss the transport layer. For now, what you need to know is that some transport protocols require some coordination between the end hosts before data transfer. That’s called connection setup or connection initiation.

HTTP example (cont.)

User enters URL: `www.someSchool.edu/someDepartment/home.index`



1a. HTTP client initiates connection to HTTP server (process) at www.someSchool.edu

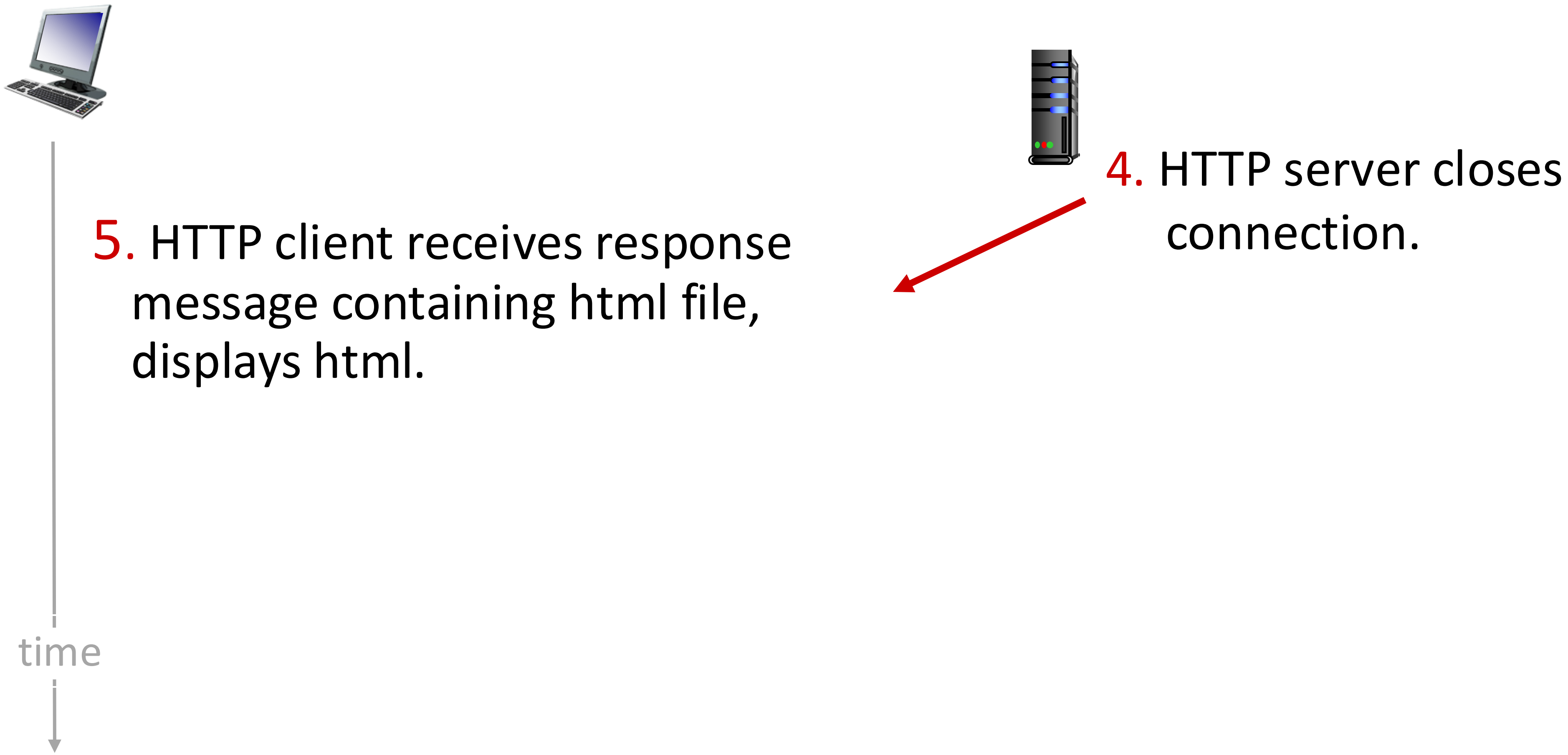
1b. HTTP server at host www.someSchool.edu "accepts" connection, notifying client

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

3. HTTP server receives request message, forms *response message* containing requested object, and sends message to the client.

HTTP example (cont.)

User enters URL: `www.someSchool.edu/someDepartment/home.index`



4. HTTP server closes connection.

5. HTTP client receives response message containing html file, displays html.

So, what did we learn about HTTP?

- Two types of HTTP messages: *request, response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
- HTTP is stateless
  - Server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

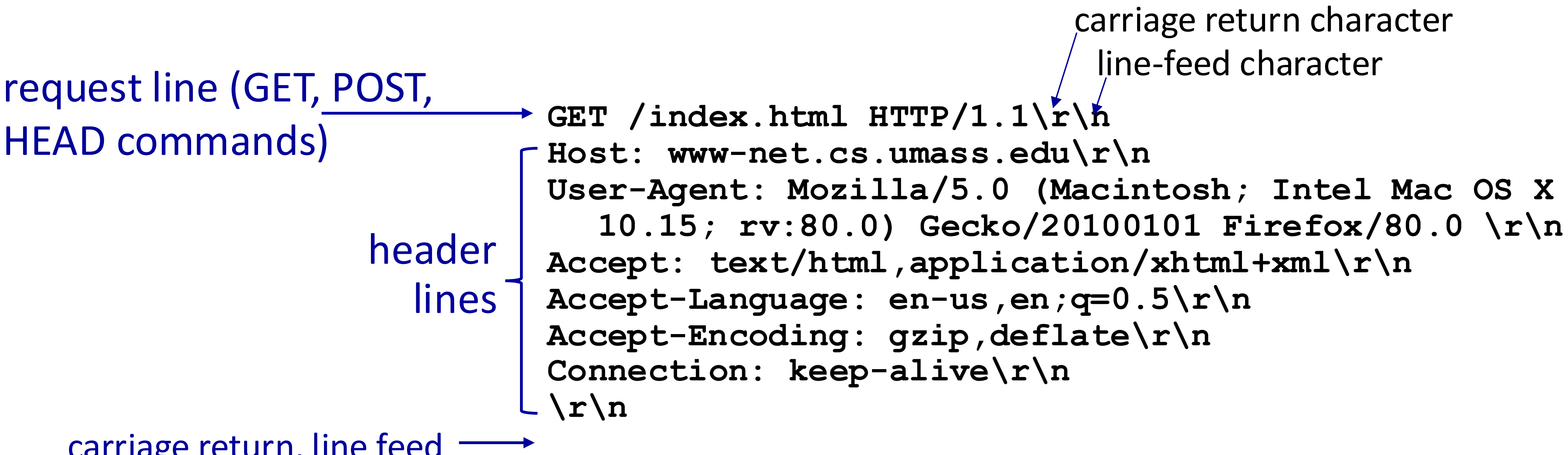
- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

So, what did we learn about HTTP?

- Two types of HTTP messages: *request, response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
- HTTP is stateless
  - Server maintains no information about past client requests

HTTP request message

- ASCII (human-readable format)

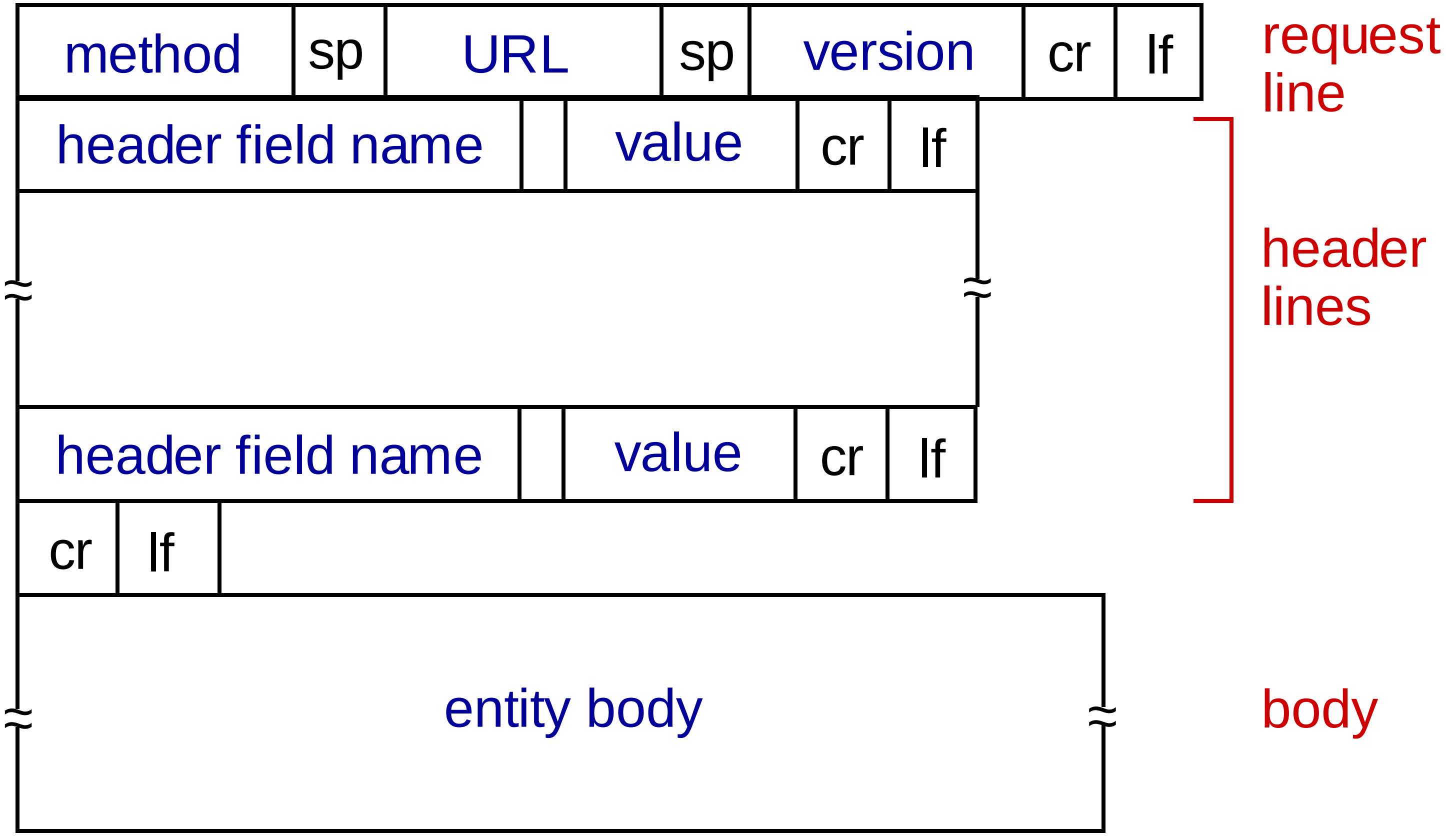


carriage return character
line-feed character
request line (GET, POST, HEAD commands)
header lines
carriage return, line feed at start of line indicates end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
   10.15; rv:80.0) Gecko/20100101 Firefox/80.0 \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
Connection: keep-alive\r\n
\r\n
```

* 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

header field name value cr lf

≈ ≈

header field name value cr lf

cr lf

≈ entity body ≈

request line

header lines

body

Other HTTP request messages

GET method

- Requests the object at the specified URL

POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

- Can also be done with a GET request by including user data in URL field of HTTP GET request message (following a ‘?’):

```
www.somesite.com/animalsearch?monkeys&banana
```

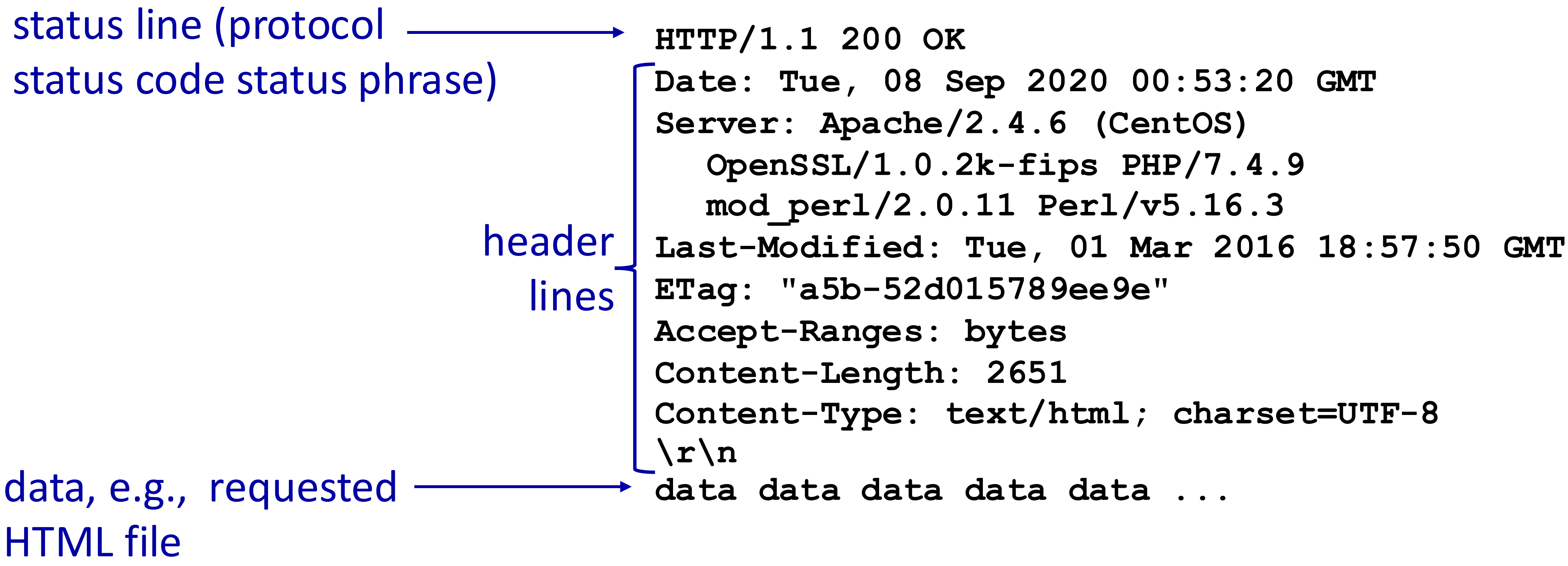
HEAD method:

- 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 PUT HTTP request message

HTTP response message



status line (protocol status code status phrase)

```
HTTP/1.1 200 OK
Date: Tue, 08 Sep 2020 00:53:20 GMT
Server: Apache/2.4.6 (CentOS)
   OpenSSL/1.0.2k-fips PHP/7.4.9
   mod_perl/2.0.11 Perl/v5.16.3
Last-Modified: Tue, 01 Mar 2016 18:57:50 GMT
ETag: "a5b-52d015789ee9e"
Accept-Ranges: bytes
Content-Length: 2651
Content-Type: text/html; charset=UTF-8
\r\n
data data data data data ...
```

header lines

data, e.g., requested HTML file

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

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

So, what did we learn about HTTP?

- Two types of HTTP messages: *request, response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
- HTTP is stateless
  - Server maintains no information about past client requests

So, what did we learn about HTTP?

- Two types of HTTP messages: *request*, *response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
- HTTP is stateless
  - Server maintains no information about past client requests

HTTP connections: two types

Non-persistent HTTP

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

downloading multiple objects required multiple connections

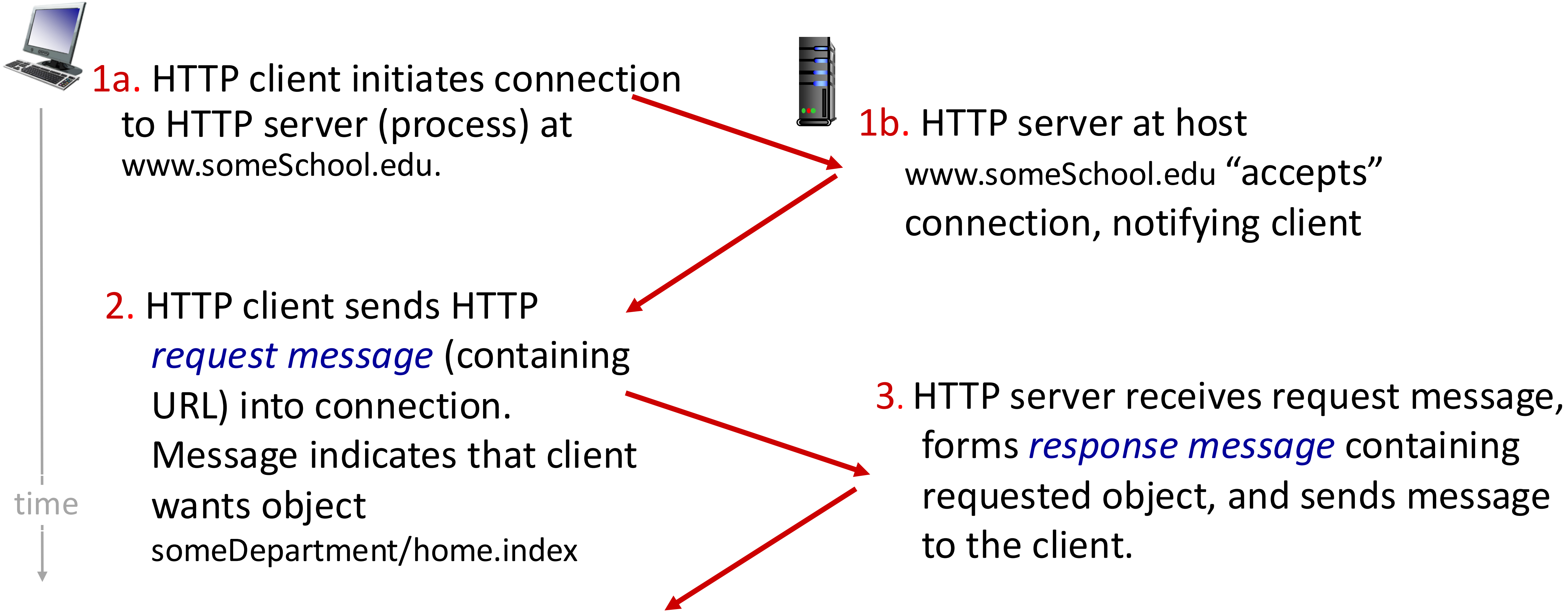
Persistent HTTP

1. Connection opened
2. multiple objects can be sent over *single* connection between client, and that server
3. 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 connection to HTTP server (process) at www.someSchool.edu.

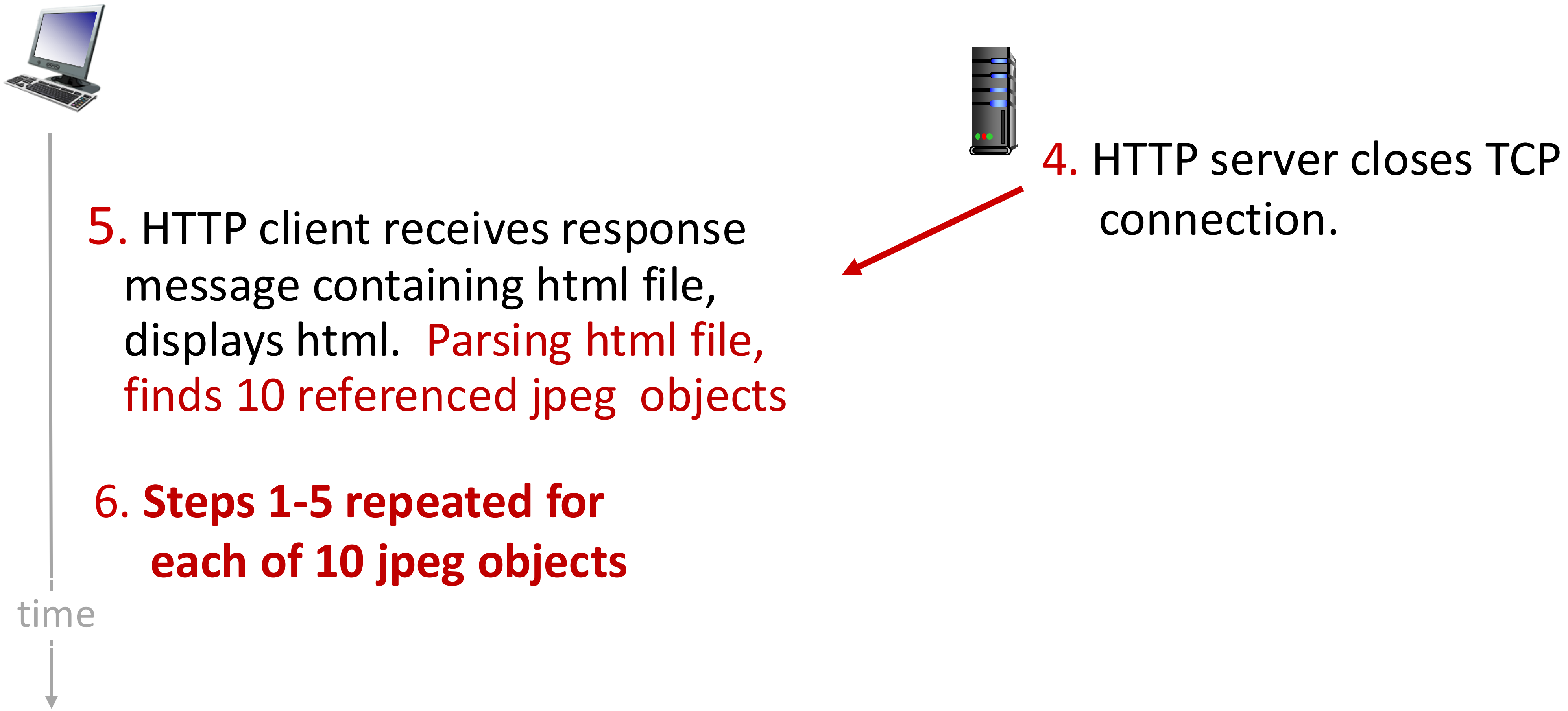
1b. HTTP server at host www.someSchool.edu "accepts" connection, notifying client

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

3. HTTP server receives request message, forms *response message* containing requested object, and sends message to the client.

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

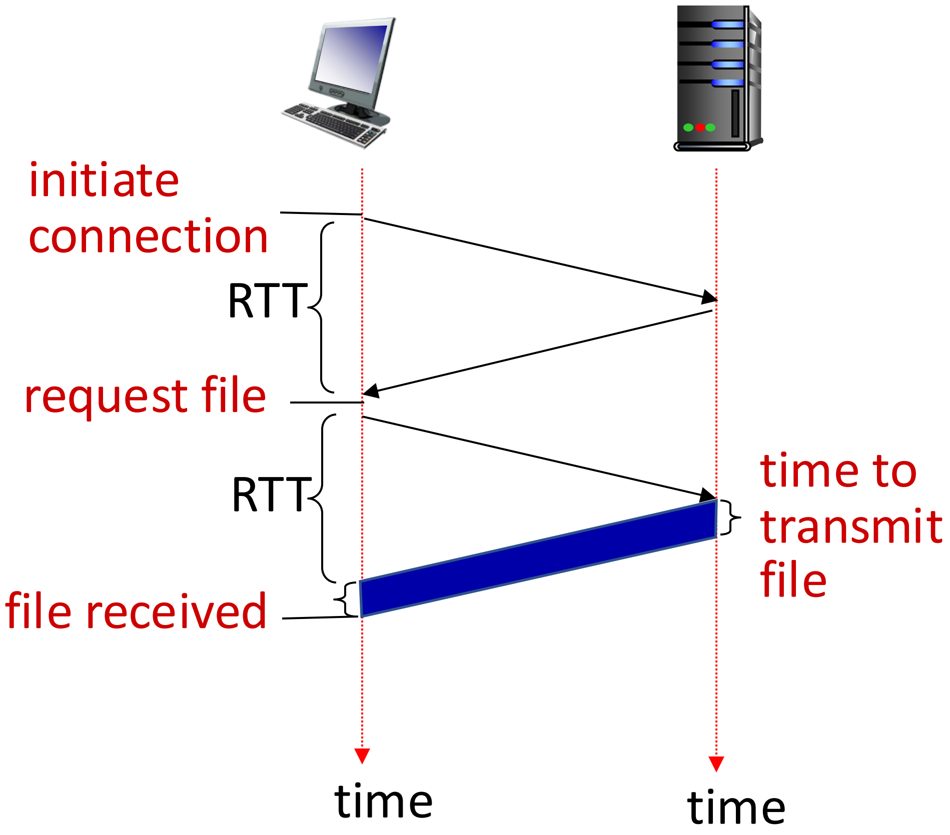
6. **Steps 1-5 repeated for each of 10 jpeg objects**

Non-persistent HTTP: response time

RTT (definition): time for a packet to travel from client to server and back

HTTP response time (per object):

- one RTT to initiate 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

Non-persistent HTTP issues

Non-persistent HTTP issues:

- A separate connection for each object
- Higher response time
  - One object: *2RTT+ file transmission time*
  - N objects: *2N*RTT+ (sum of file transmission time for the N objects)*
  - browsers often open multiple parallel TCP connections to fetch referenced objects in parallel to improve response time.
- Higher resource overhead:
  - The end host operating system incurs overhead for maintaining *each* connection

Persistent HTTP (HTTP 1.1)

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over the already established connection
- client sends requests as soon as it encounters a referenced object
- Lower response time:
  - Response time for the first object: *2RTT + file transmission time*
  - Response time for the next (N – 1) objects: *RTT + file transmission time*
  - As little as one RTT for almost all the referenced objects
  - cutting response time in half

Persistent HTTP (HTTP 1.1)

- Lower response time:
  - Response time for the first object: *2RTT + file transmission time*
  - Response time for the next (N – 1) objects: *RTT + file transmission time*
  - As little as one RTT for almost all the referenced objects
  - cutting overall response time ~in half
- Lower resource overhead
  - No need to have multiple open connections to the same server to improve response time.

Persistent HTTP (HTTP 1.1)

Q: why didn't we do this from the beginning?

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over the already established connection
- client sends requests as soon as it encounters a referenced object
- Lower response time
- Lower resource overhead

So, what did we learn about HTTP?

- Two types of HTTP messages: *request, response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
  - Non-persistent vs persistent connection
- HTTP is stateless
  - Server maintains no information about past client requests

So, what did we learn about HTTP?

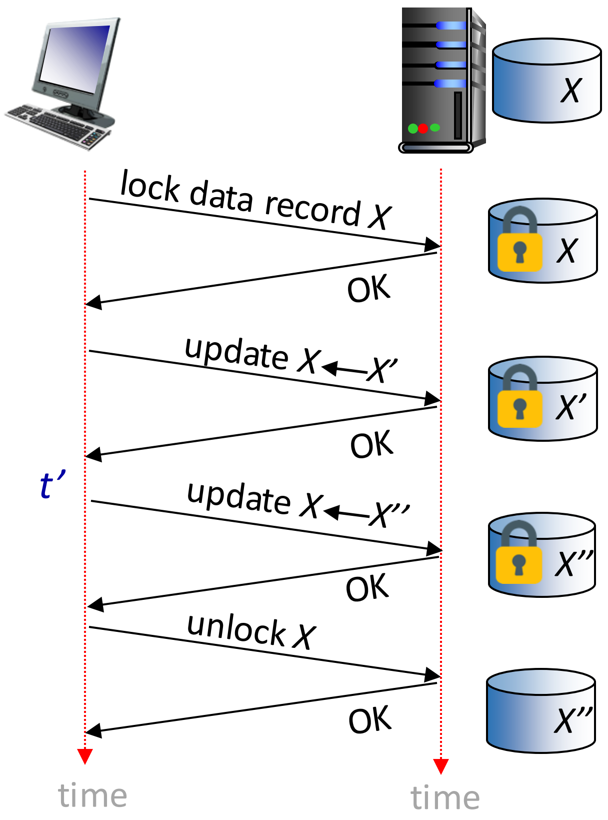
- Two types of HTTP messages: *request*, *response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
  - Non-persistent vs persistent connection
- HTTP is stateless
  - Server maintains no information about past client requests

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

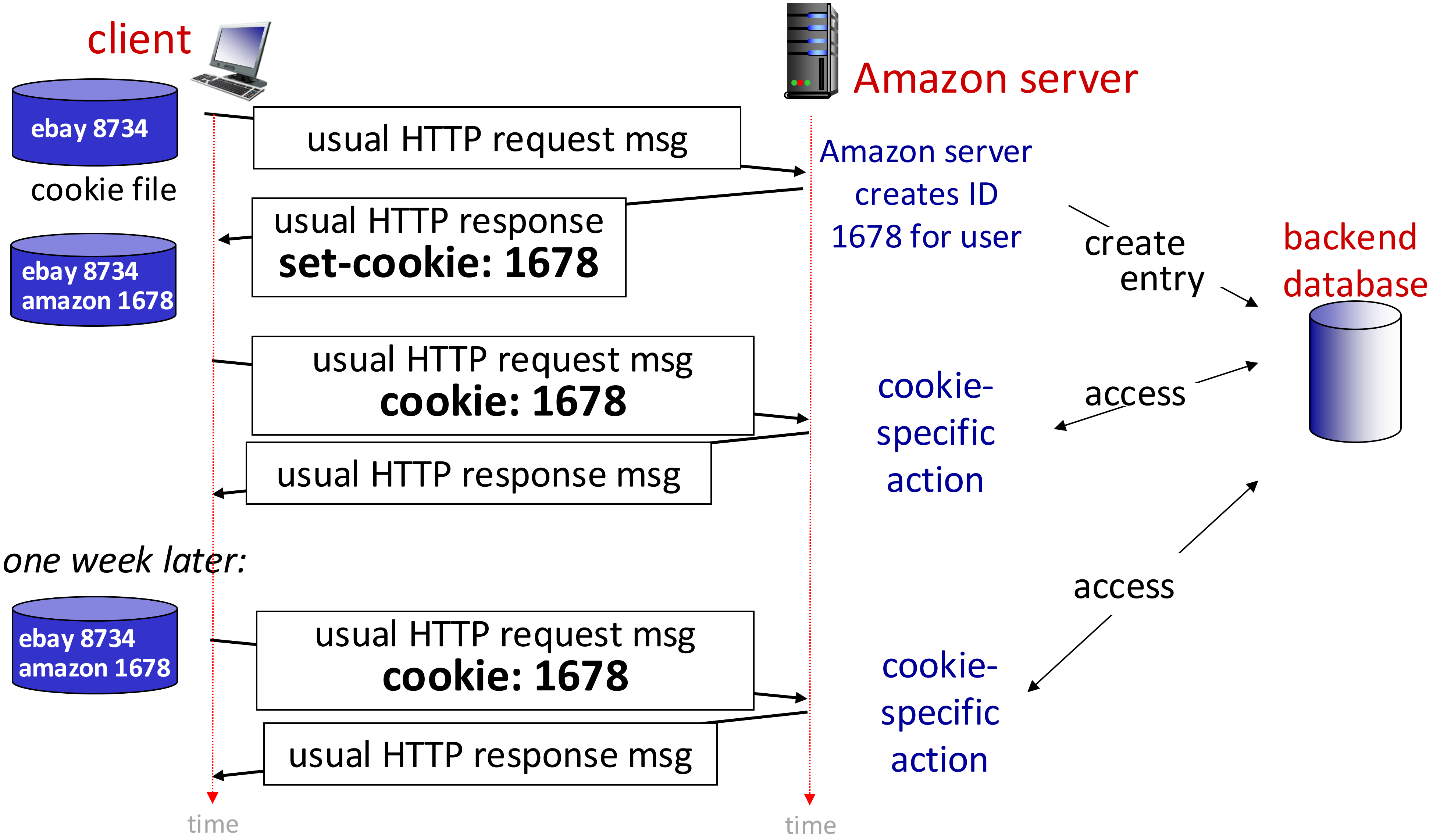
four components:

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

Example:

- Susan uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP request 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

aside

cookies and privacy:

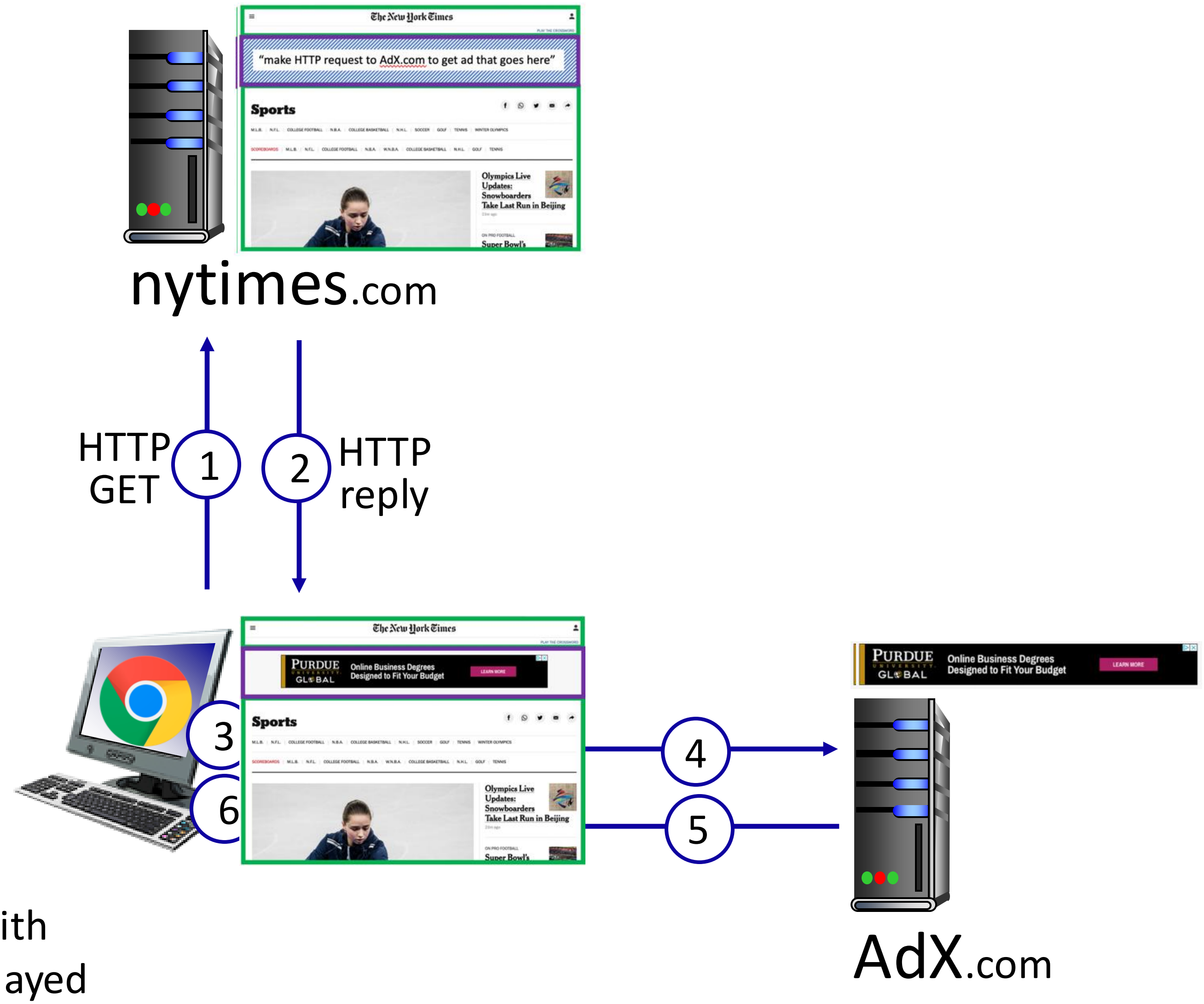
- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

Example: displaying a NY Times web page

1 2 GET base html file from nytimes.com

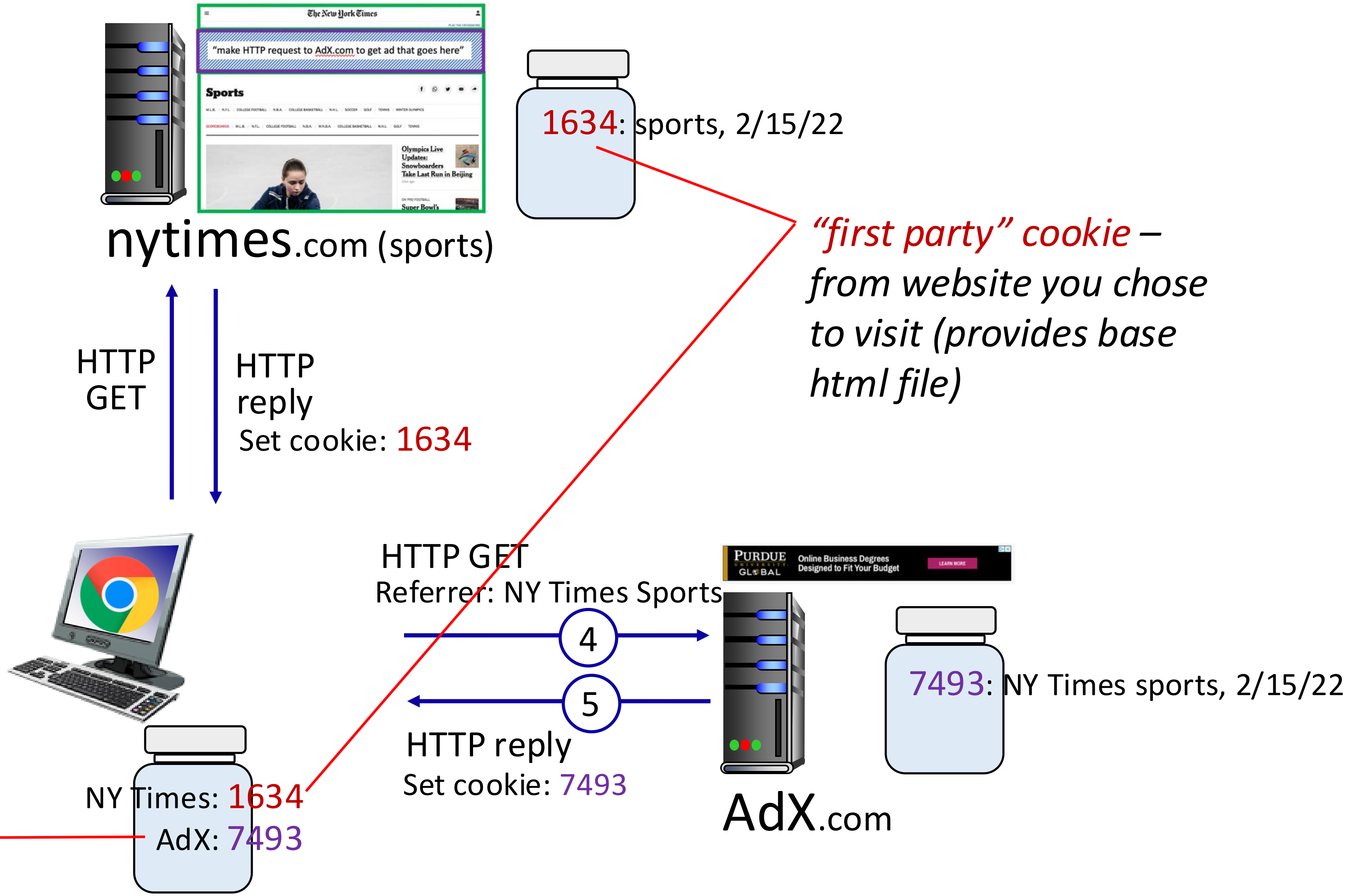
4 5 fetch ad from AdX.com

7 display composed page


The New York Times
"make HTTP request to AdX.com to get ad that goes here"
Sports
Olympics Live Updates: Snowboarders Take Last Run in Beijing
Super Bowl's
nytimes.com
HTTP GET
1
2
HTTP reply
The New York Times
PURDUE UNIVERSITY GLOBAL
Online Business Degrees Designed to Fit Your Budget
LEARN MORE
Sports
3
6
4
5
AdX.com

NY times page with embedded ad displayed

Cookies: tracking a user’s browsing behavior



"make HTTP request to AdX.com to get ad that goes here"

nytimes.com (sports)

1634: sports, 2/15/22

HTTP GET

HTTP reply
Set cookie: 1634

HTTP GET
Referrer: NY Times Sports

4

5

HTTP reply
Set cookie: 7493

AdX.com

7493: NY Times sports, 2/15/22

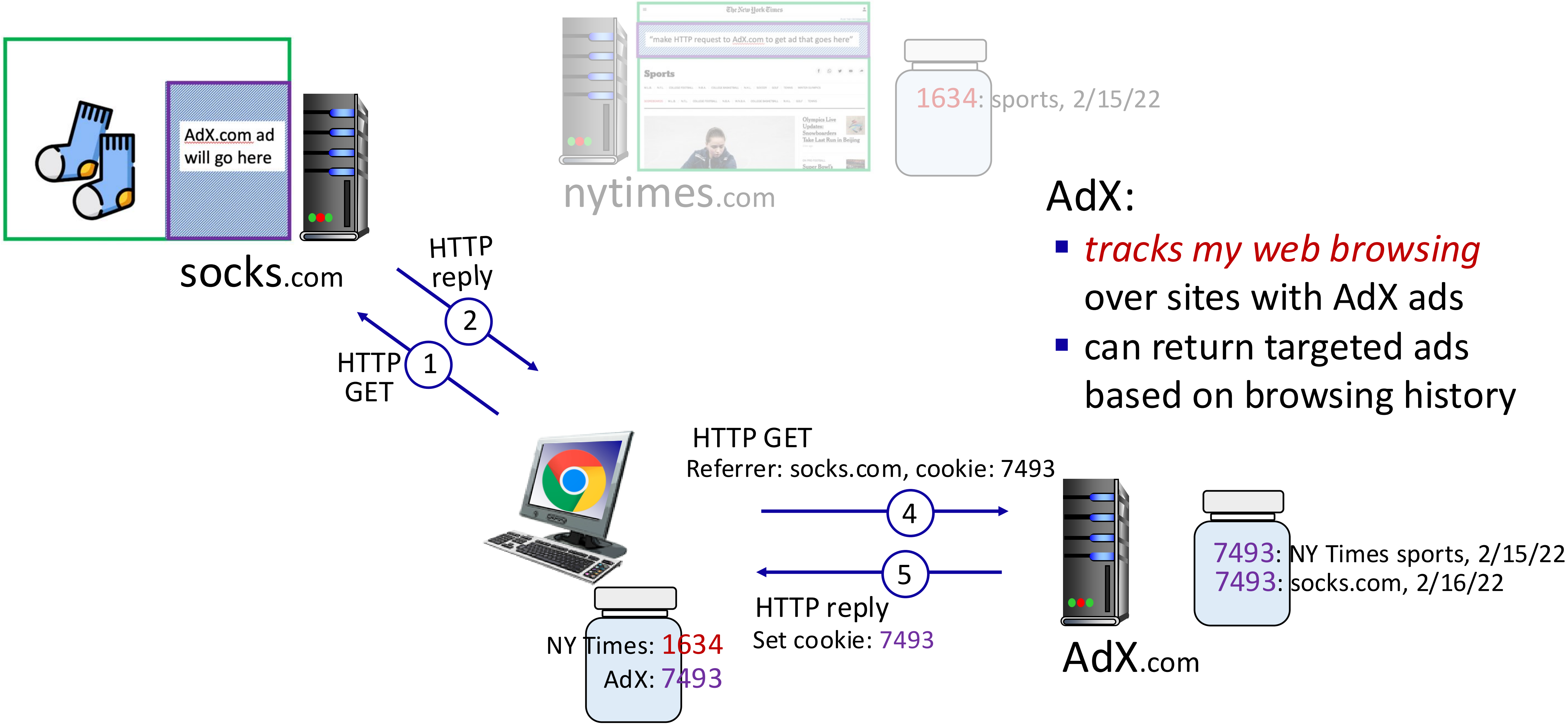
NY Times: 1634
AdX: 7493

The New York Times
Sports
Olympics Live Updates: Snowboarders Take Last Run in Beijing
Super Bowl's
PURDUE UNIVERSITY GLOBAL
Online Business Degrees Designed to Fit Your Budget
LEARN MORE

“first party” cookie – *from website you chose to visit (provides base html file)*

“third party” cookie – *from website you did __not__ choose to visit*

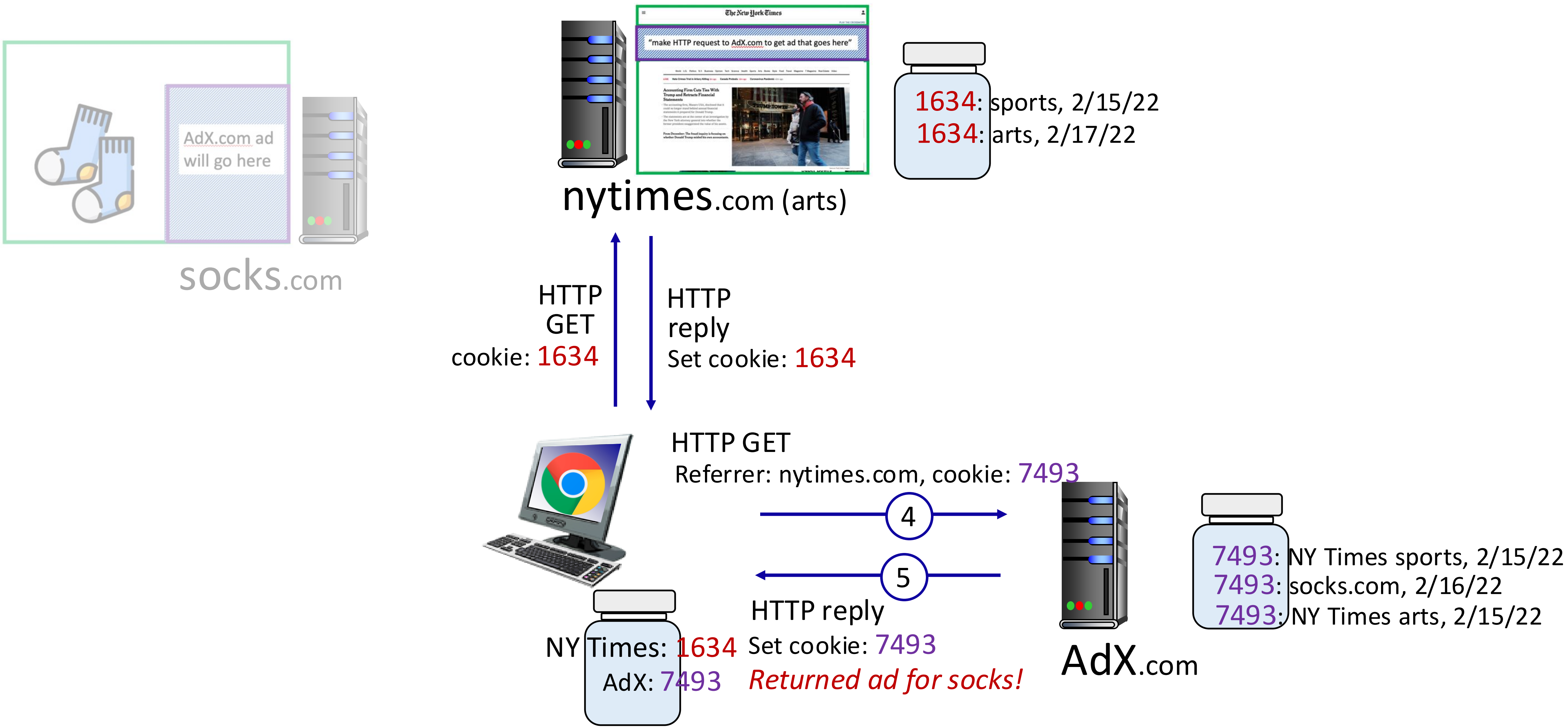
Cookies: tracking a user's browsing behavior



AdX:

- *tracks my web browsing* over sites with AdX ads
- can return targeted ads based on browsing history

Cookies: tracking a user's browsing behavior (one day later)



AdX.com ad will go here

socks.com

"make HTTP request to AdX.com to get ad that goes here"

nytimes.com (arts)

1634: sports, 2/15/22
1634: arts, 2/17/22

HTTP GET
cookie: 1634

HTTP reply
Set cookie: 1634

HTTP GET
Referrer: nytimes.com, cookie: 7493

4

5

HTTP reply
Set cookie: 7493
Returned ad for socks!

NY Times: 1634
AdX: 7493

AdX.com

7493: NY Times sports, 2/15/22
7493: socks.com, 2/16/22
7493: NY Times arts, 2/15/22

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:

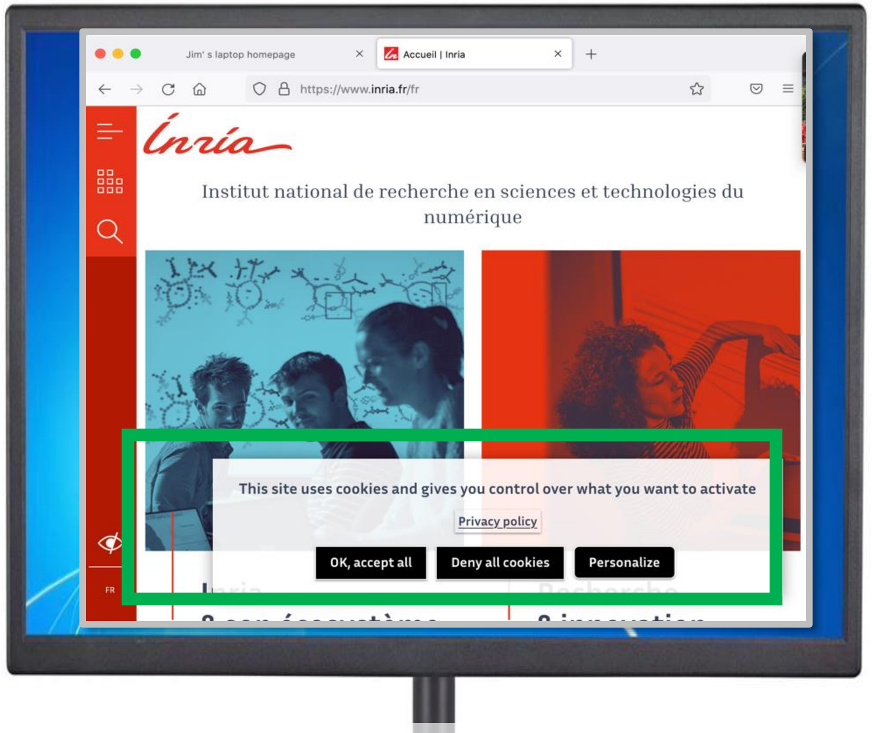
- disabled by default in Firefox, Safari browsers
- to be disabled in Chrome browser in 2023

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



Institut national de recherche en sciences et technologies du numérique

This site uses cookies and gives you control over what you want to activate

User has explicit control over whether or not cookies are allowed

So, what did we learn about HTTP?

- Two types of HTTP messages: *request, response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
  - Non-persistent vs persistent connection
- HTTP is stateless
  - Server maintains no information about past client requests

So, what did we learn about HTTP?

- Two types of HTTP messages: *request, response*
- HTTP Connection: Built on top of a reliable Connection-based transport service
  - Non-persistent vs persistent connection
- ~~HTTP is stateless~~
  - ~~Server maintains no information about past client requests~~
- HTTP can be stateful
  - E.g., cookies

Improving web application performance

- Non-persistent HTTP connections → persistent HTTP connections
- Cookies can help improve performance

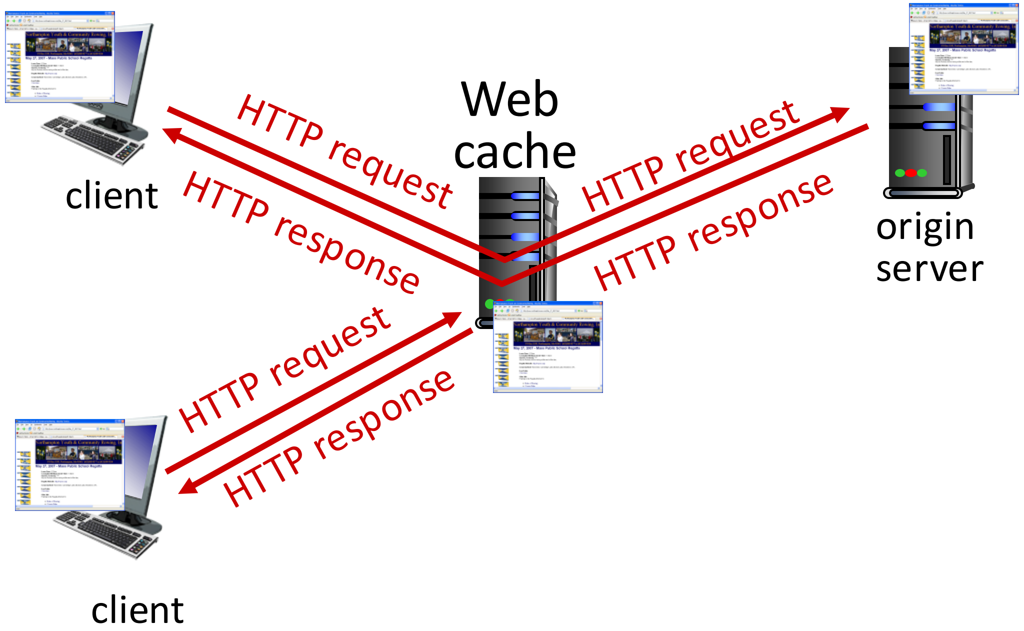
Improving web application performance

- Non-persistent HTTP connections → persistent HTTP connections
- Cookies can help improve performance
- Caching!
  - Web caching
  - Brower caching

Web caches

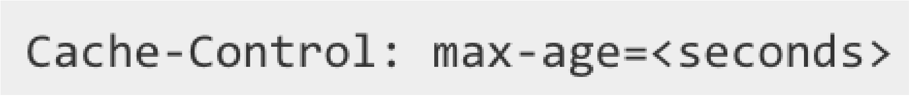
Goal: satisfy client requests without involving origin server

- 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

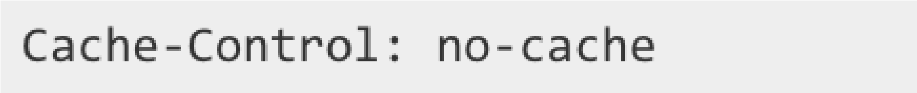


Web caches (aka proxy servers)

- 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

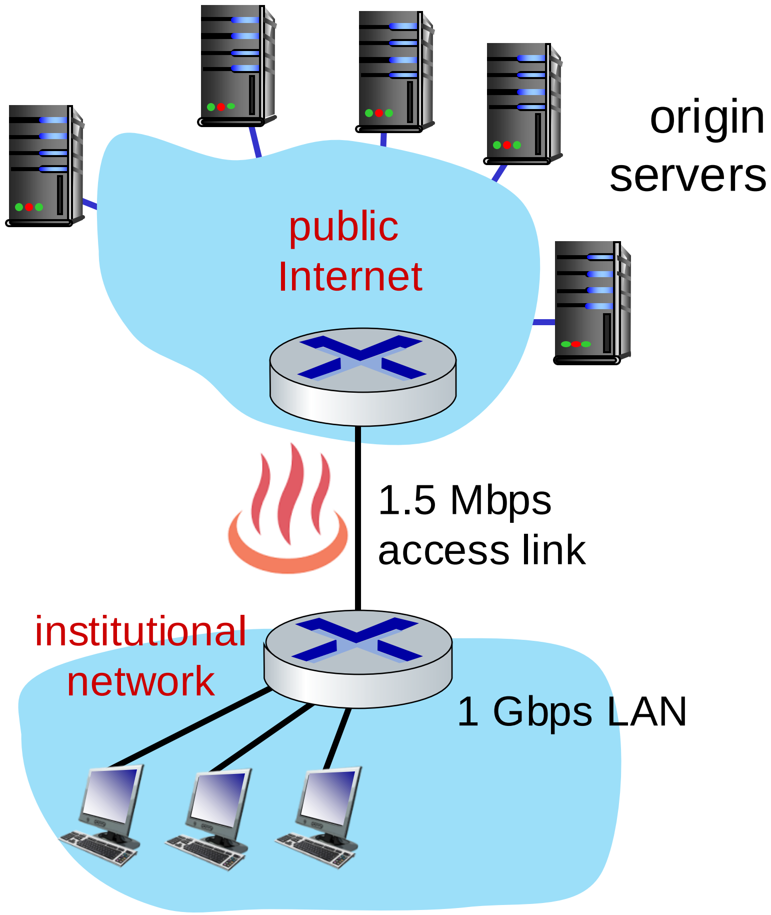
Caching example

Scenario:

- access link rate: 1.5 Mbps
- RTT from institutional router to server: 2 sec
- average web object size: 750K bits
- average request rate from browsers to origin servers: 1.8/sec

Question:

- What is the average delay for a web object crossing the access link?
  - Mostly affected by queuing delay
  - Avg queuing delay = $\bar{x}/(1-\lambda\bar{x})$, where λ is the number of objects per second, and $\bar{x}$ is the average transmission time of each object.
- What is the average response time?
  - Response time = Internet delay + access link delay + LAN delay (negligible)



Caching example

Scenario:

- access link rate: 1.5 Mbps
- RTT from institutional router to server: 2 sec
- average web object size: 750K bits
- average request rate from browsers to origin servers: 1.8/sec

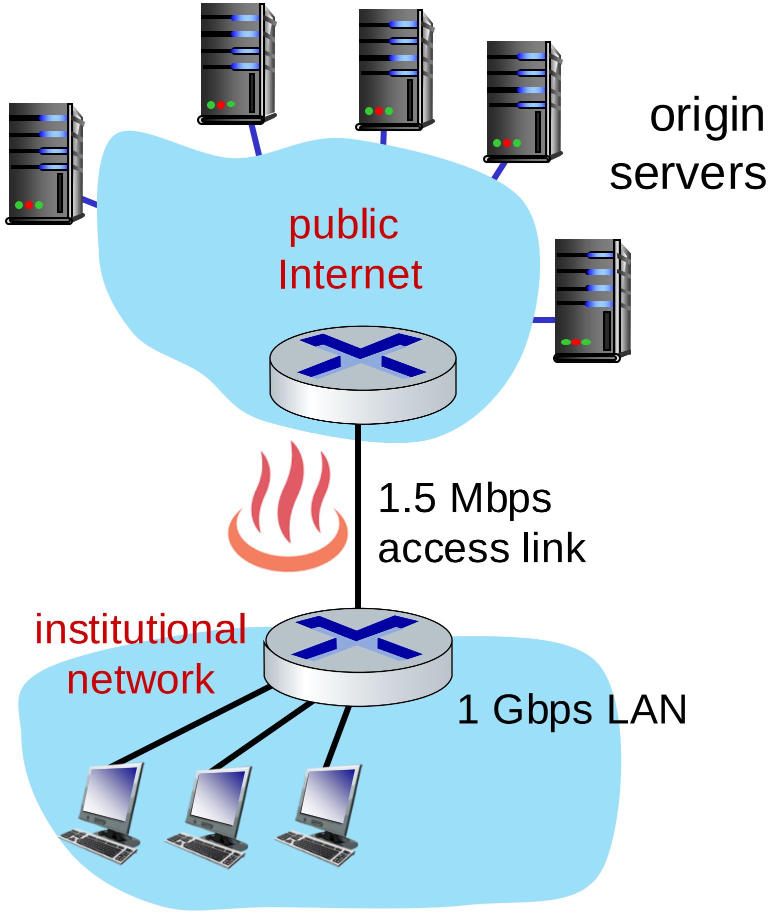
Question:

- What is the average delay for a web object crossing the access link?

delay ~= queueing delay = $\frac{\bar{x}}{1-\lambda\bar{x}} = \frac{0.75/1.5}{1 - 1.8*0.75/1.5} = 5$ secs

- What is the average response time?

Response time ~= 2 secs + 5 secs = 7 secs



problem: large queueing delays and internet delay!

Option 1: buy a faster access link

Scenario:

- access link rate: ~~1.5 Mbps~~ 15 Mbps
- RTT from institutional router to server: 2 sec
- average web object size: 750K bits
- average request rate from browsers to origin servers: 1.8/sec

Question:

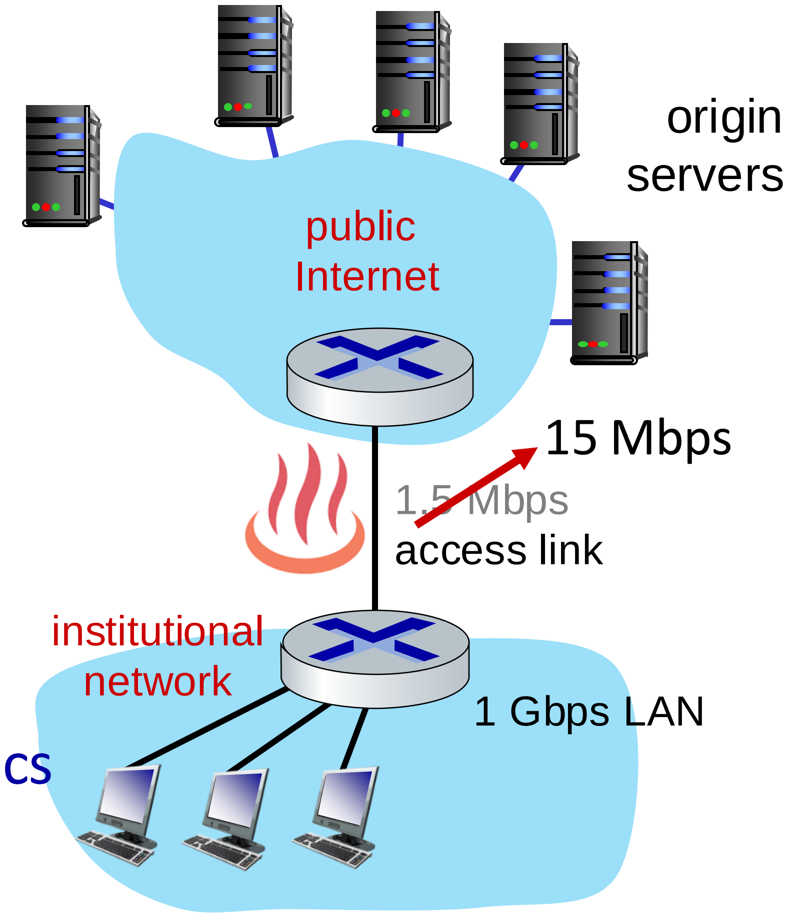
- What is the average delay for a web object crossing the access link?

delay ~= queueing delay = $\frac{\bar{x}}{1-\lambda\bar{x}} = \frac{0.75/15}{1-1.8*0.75/15} = 0.055$ secs

- What is the average response time?

Response time ~= 2 secs + 0.055 secs = 2.055 secs

Cost: faster access link (expensive!)



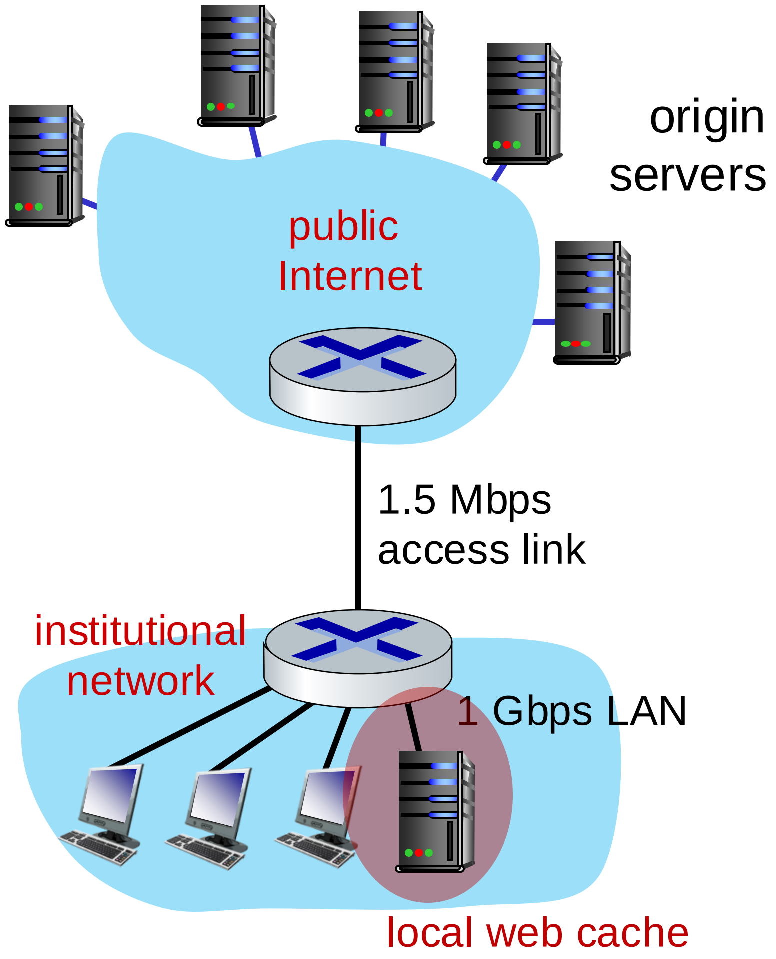
Option 2: install a web cache

Scenario:

- access link rate: 1.5 Mbps
- RTT from institutional router to server: 2 sec
- web object size: 750K bits
- average request rate from browsers to origin servers: 1.8/sec
- Web cache hit ratio is 0.6:
  - 60% requests served by cache, with negligible delay
  - 40% requests served by origin servers

Question:

- What is the average delay for a web object crossing the access link?
- What is the average response time?



Cost: web cache (cheap!)

Browser caching: Conditional GET

Goal: don’t send object if browser has up-to-date cached version

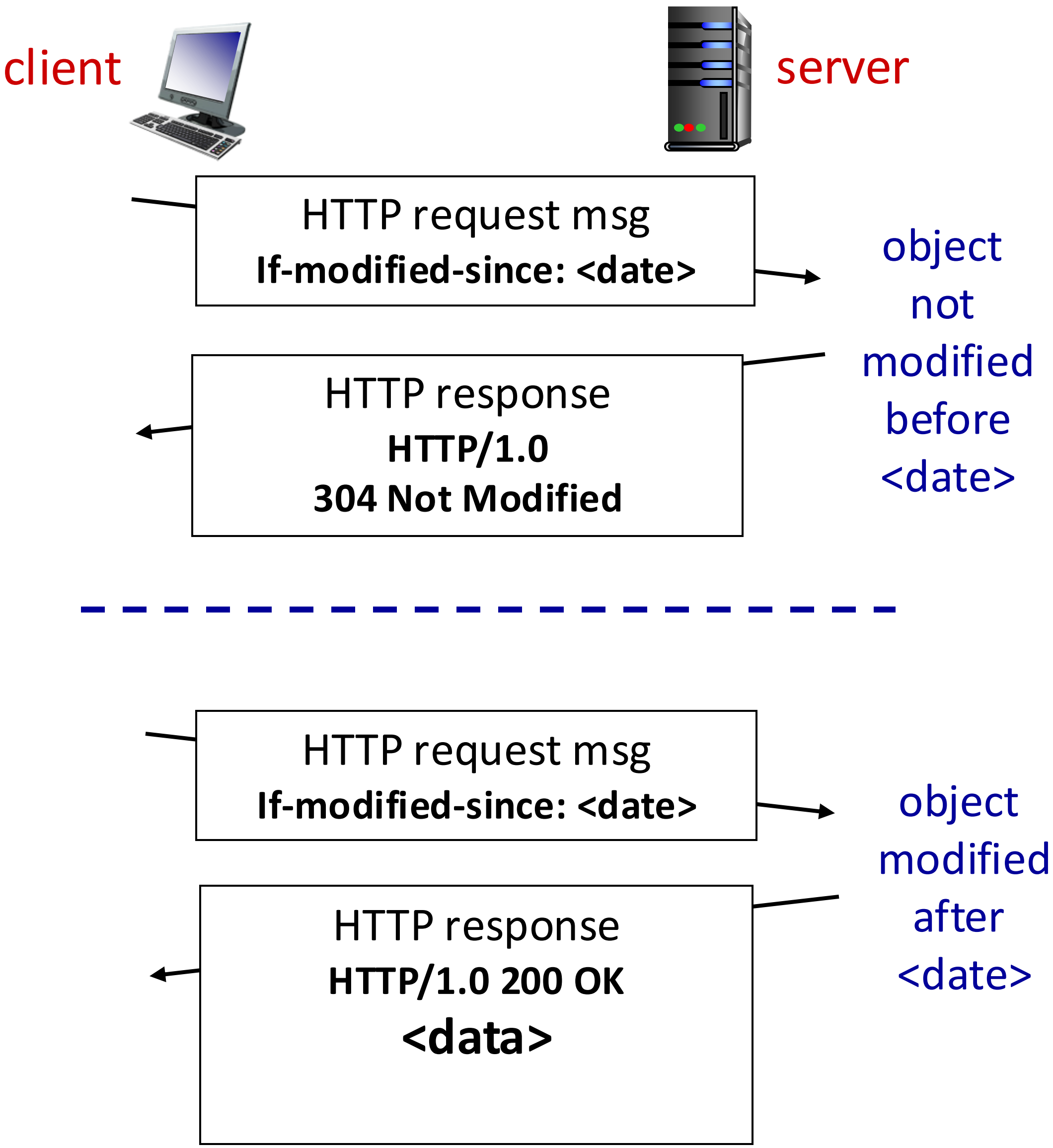
- no object transmission delay (or use of network resources)

- *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>

Improving web application performance

- Non-persistent HTTP connections → persistent HTTP connections
- Cookies can help improve performance
- Caching!
  - Web caching
  - Brower caching

Improving web application performance

- Non-persistent HTTP connections → persistent HTTP connections
- Cookies can help improve performance
- Caching!
  - Web caching
  - Brower caching
- HTTP/2 and HTTP/3

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)
  - Specially if objects ahead of them are lost and have to be retransmitted.

HTTP/2

Key goal: decreased delay in multi-object HTTP requests

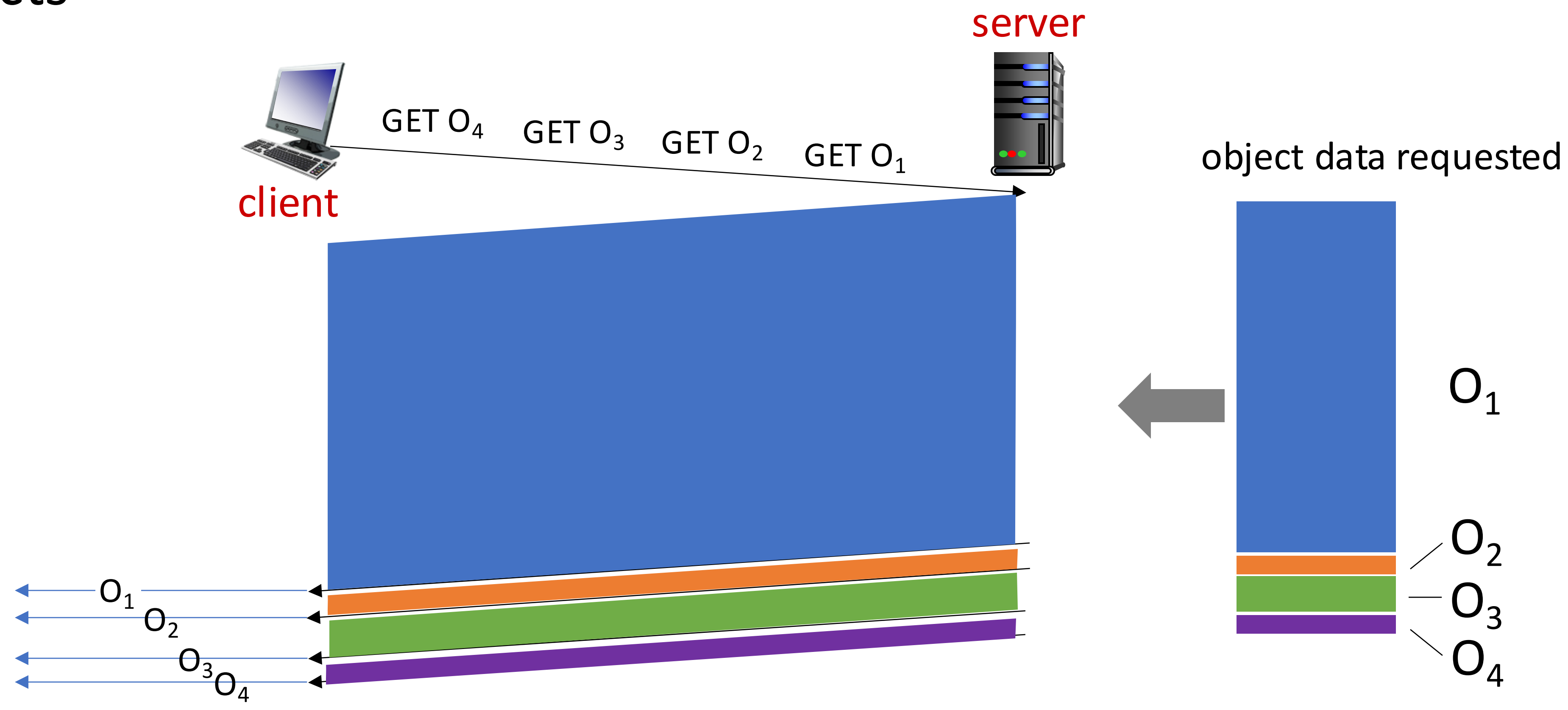
HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:

- 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

Overloaded term, different from link layer frames

HTTP/2: mitigating HOL blocking

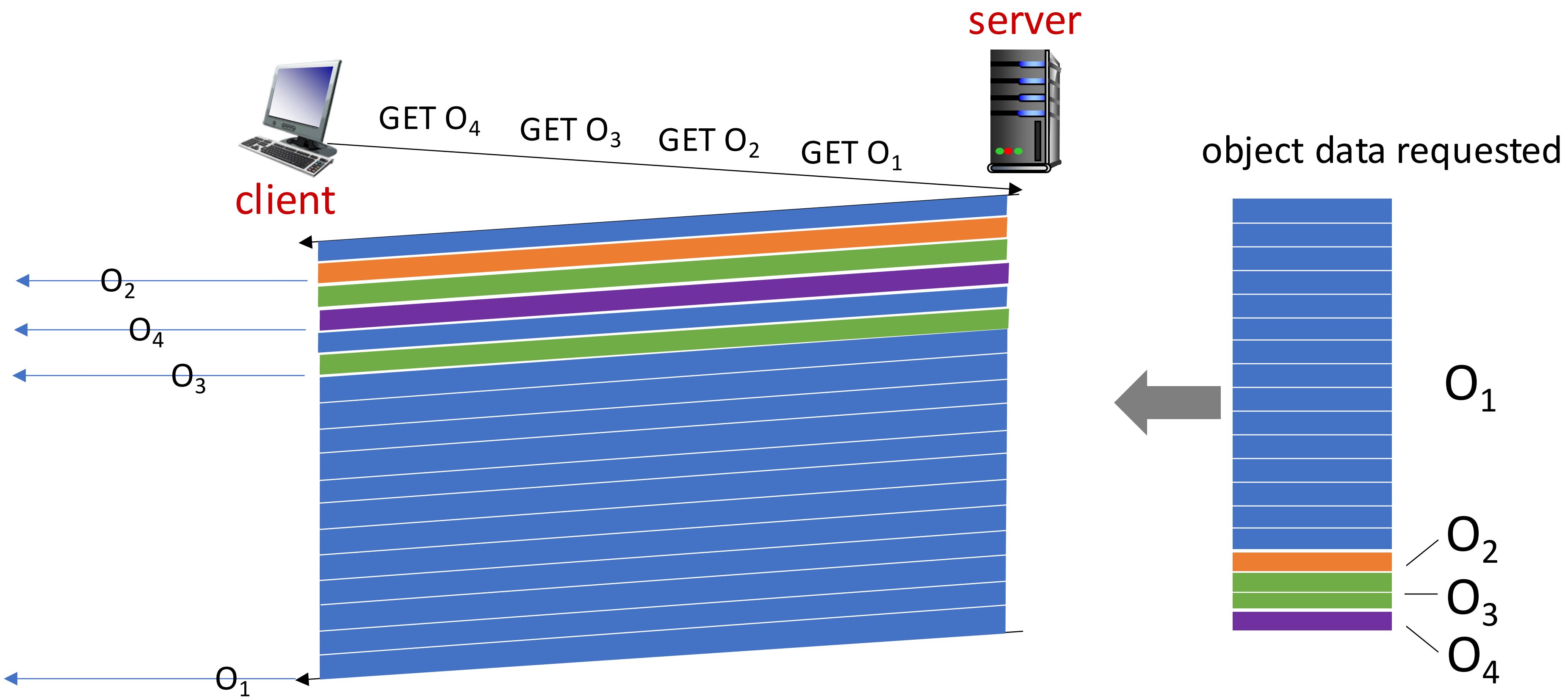
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



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 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