Chapter 2
Application Layer

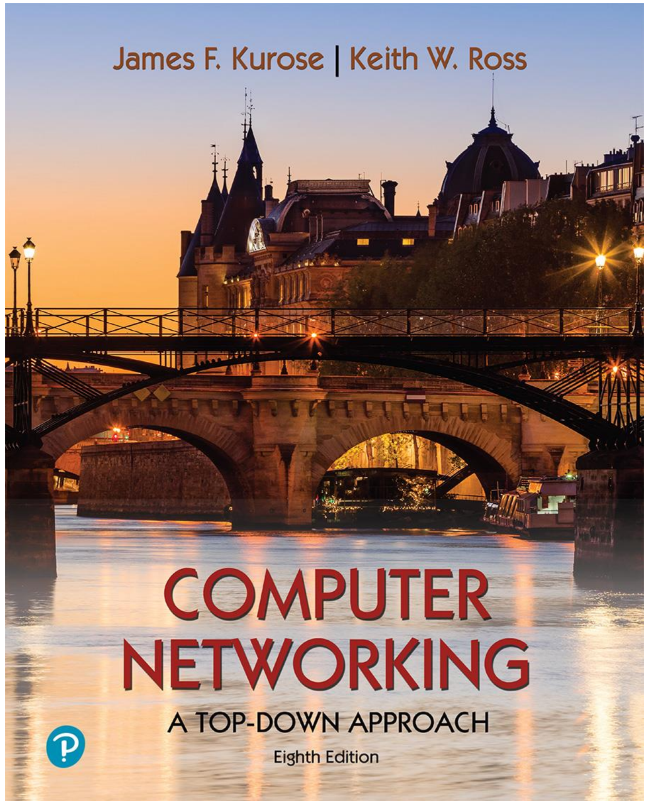
A note on the use of these PowerPoint slides:

We’re making these slides freely available to all (faculty, students, readers). They’re in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we’d like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

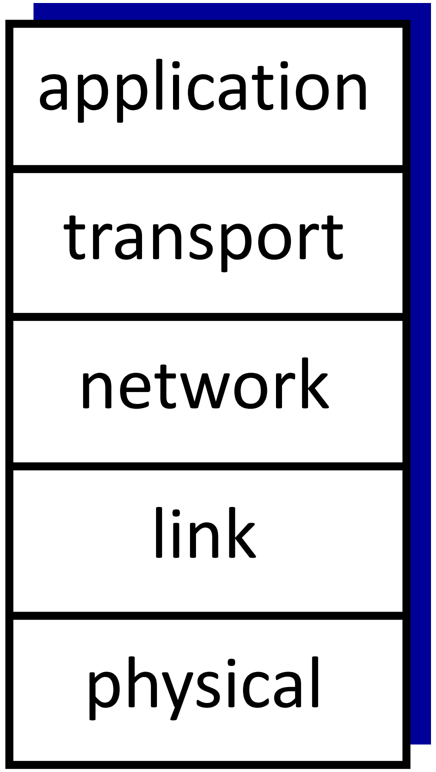
Thanks and enjoy! JFK/KWR



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

Internet protocol stack

- *application:* supporting network applications
  - IMAP, SMTP, HTTP
- *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

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS
- 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
- Internet search
- remote login
- …

Q: *your* favorites?

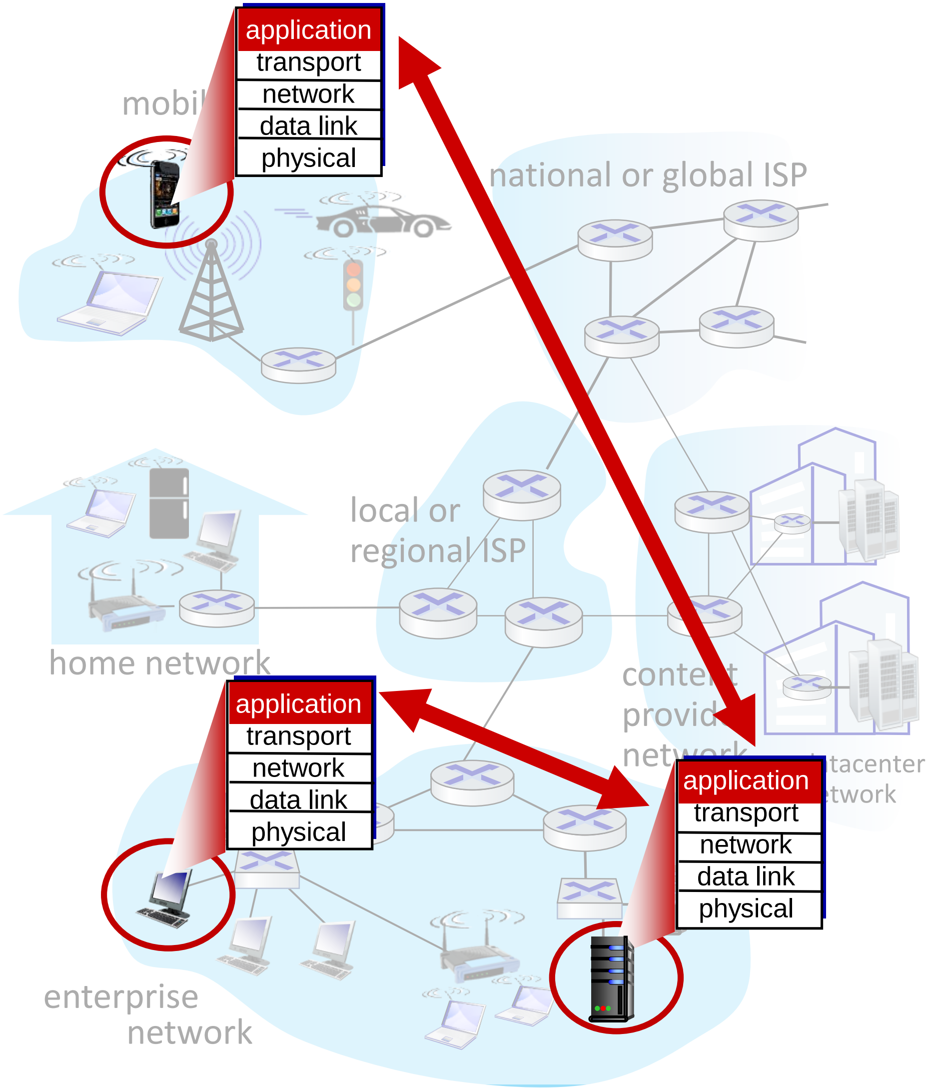
Creating a network app

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

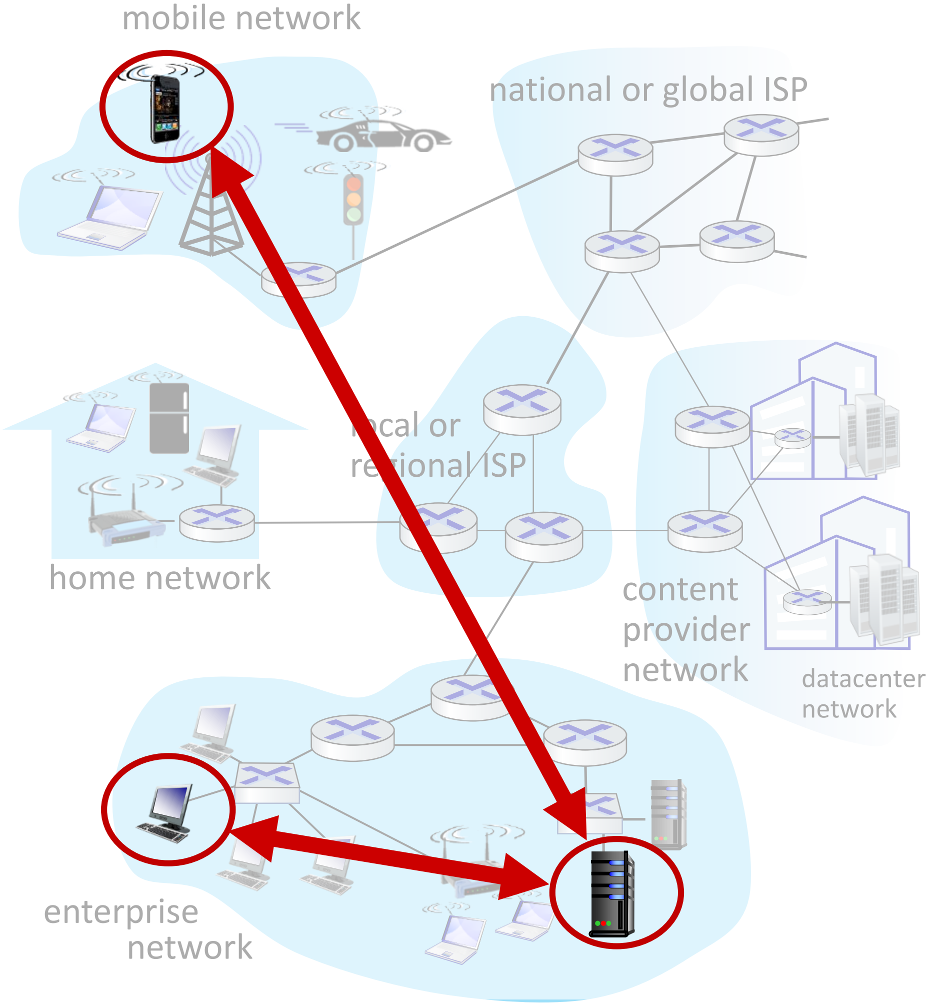
server:

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

clients:

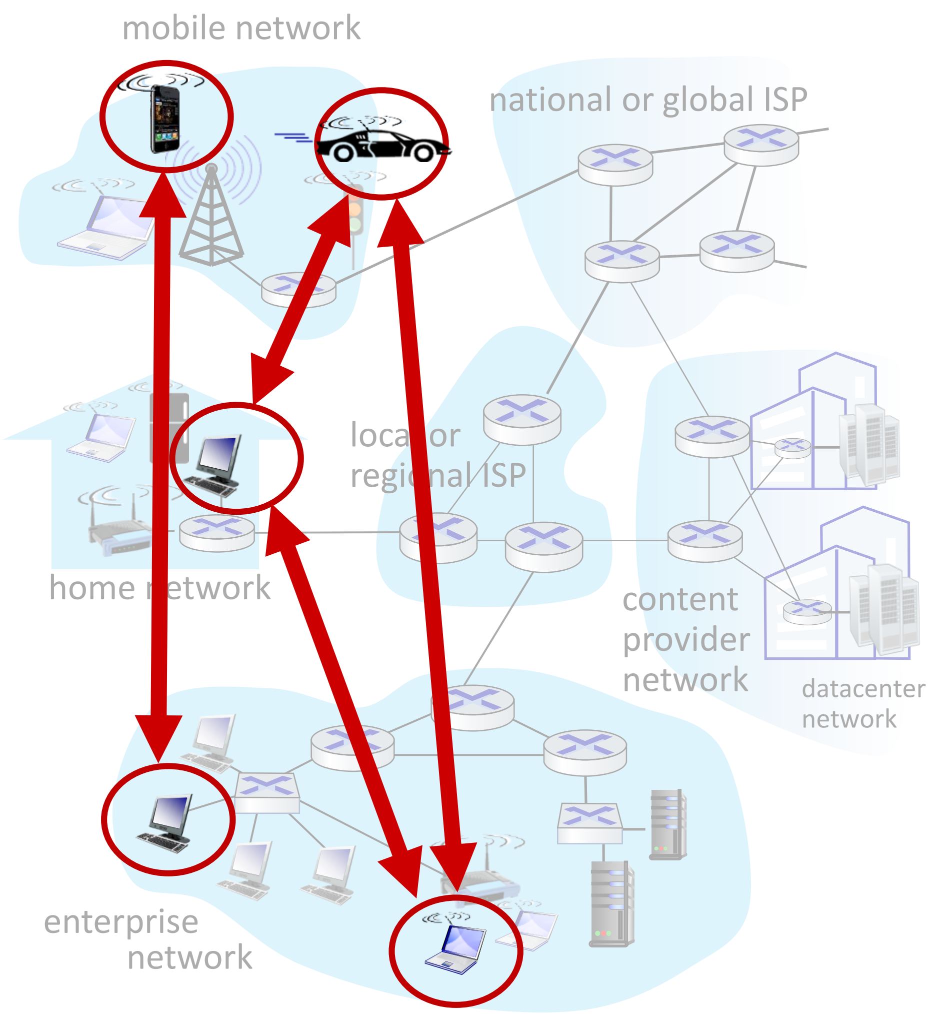
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



Peer-peer 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 IP addresses
  - complex management
- example: P2P file sharing



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

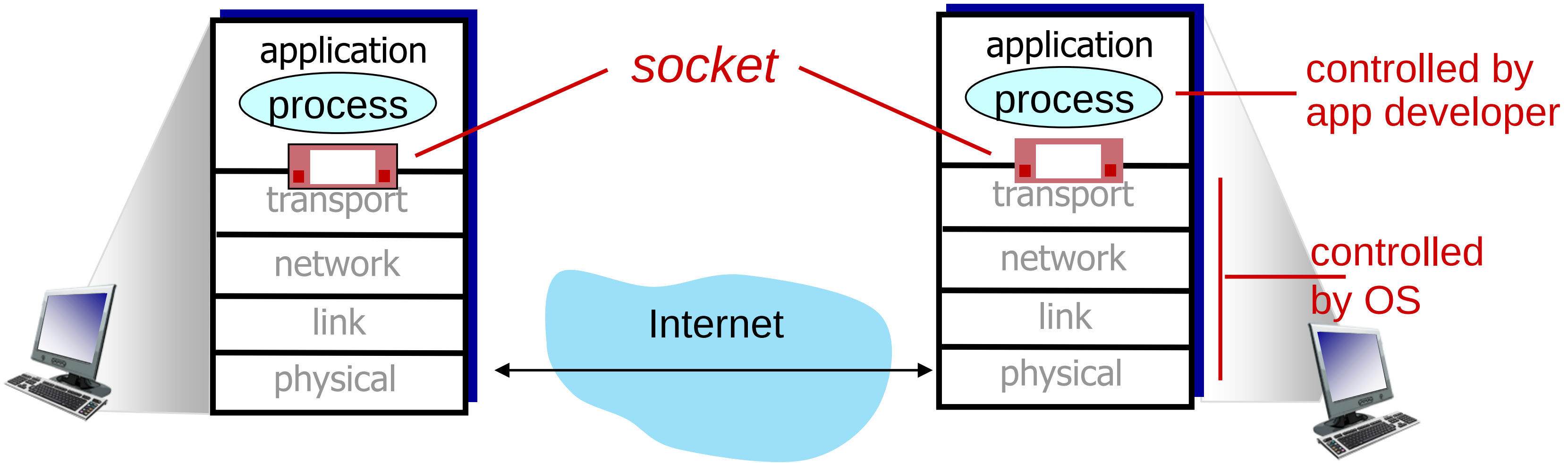
client process: process that initiates communication

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



socket

controlled by app developer

controlled by OS

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

What transport service does 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 transport protocols services

TCP 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

UDP service:

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

Q: why bother? *Why* is there a UDP?

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

FTP: File Transfer Protocol
SMTP: Simple Mail Transfer Protocol
HTTP: HyperText Transfer Protocol
SIP: Session Initiation Protocol

RTP: Real-time Transport Protocol
DASH: Dynamic Adaptive Streaming over HTTP
WOW: World of Warcraft
FPS: First Person Shooters

Securing TCP

Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn

TLS socket API

- cleartext sent into socket traverse Internet *encrypted*
- see Chapter 8

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

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.,

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

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Simple HTML</title>
</head>
<body>
<h1> <img src="bzulogo.png" alt="bzulogo" width="40" height="40">
Birzeit University</h1>
<p>Birzeit University offers graduate and undergraduate programs in
information technology, engineering, sciences...</p>
<img src="bzu.jpg" alt="bzu" width="300" height="300">

<h2>Faculties</h2>
 <ul>
  <li>Engineering and Technology</li>
  <li>Arts</li>
  <li>Business</li>
  <li>...</li>
</ul>
<a href="https://www.w3schools.com">Visit W3Schools.com to learn
more about HTML!</a>
</body>
</html>
```

Birzeit University

Birzeit University offers graduate and undergraduate programs in information technology, engineering, sciences...

Faculties

- Engineering and Technology
- Arts
- Business
- ...

Visit W3Schools.com to learn more about HTML!

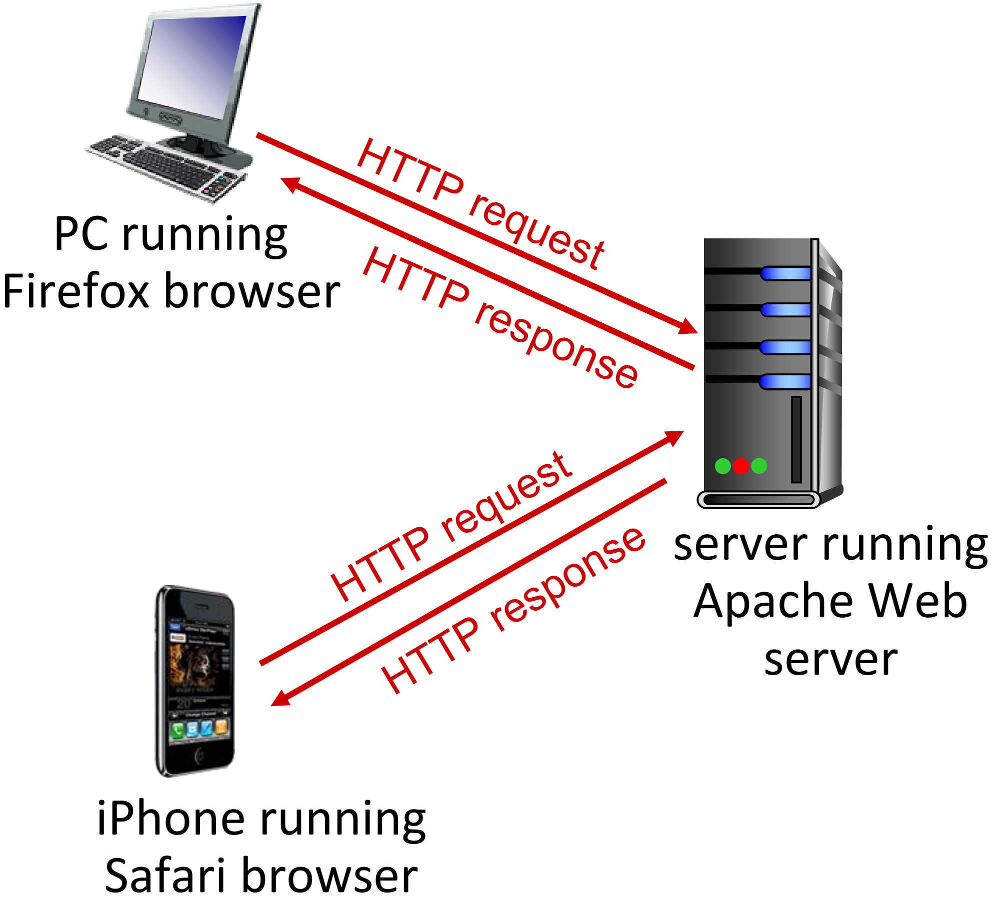
Some HTML Tags

Tag	**Description**
<html> … </html>	Declares the Web page to be written in HTML
<head> … </head>	Delimits the page's head
<title> … </title>	Defines the title (not displayed on the page)
<body> … </body>	Delimits the page's body
<h *n*> … </h *n*>	Delimits a level *n* heading
 … 	Set … in boldface
<i> … </i>	Set … in italics
<center> … </center>	Center … on the page horizontally
…	Brackets an unordered (bulleted) list
…	Brackets a numbered list
	Starts a list item (there is no)
 	Forces a line break here
<p>	Starts a paragraph
<hr>	Inserts a horizontal rule
<img src="...">	Displays an image here
 … 	Defines a hyperlink

HTTP overview

HTTP: hypertext transfer protocol

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



HTTP overview (continued)

HTTP 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

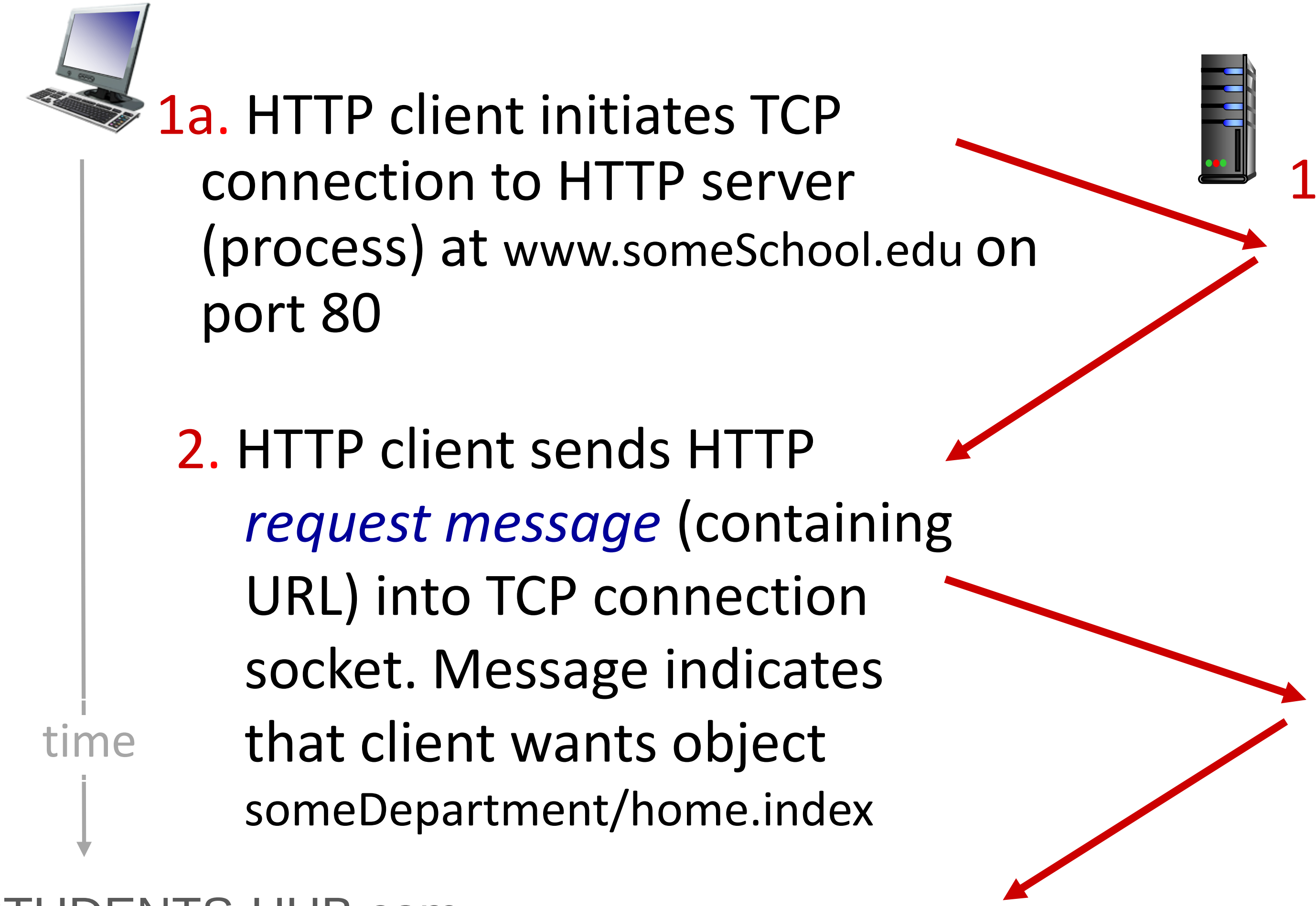
Persistent HTTP

- 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

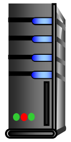
2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

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

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

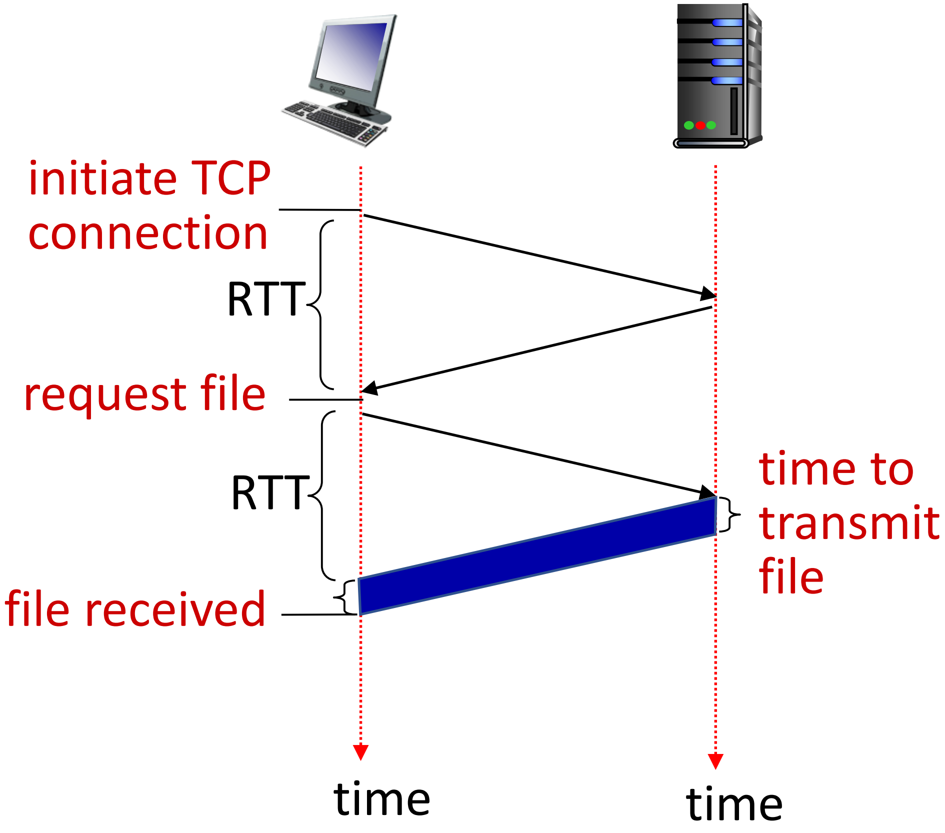
time

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 *(FTT)*



initiate TCP connection

RTT

request file

RTT

file received

time to transmit file

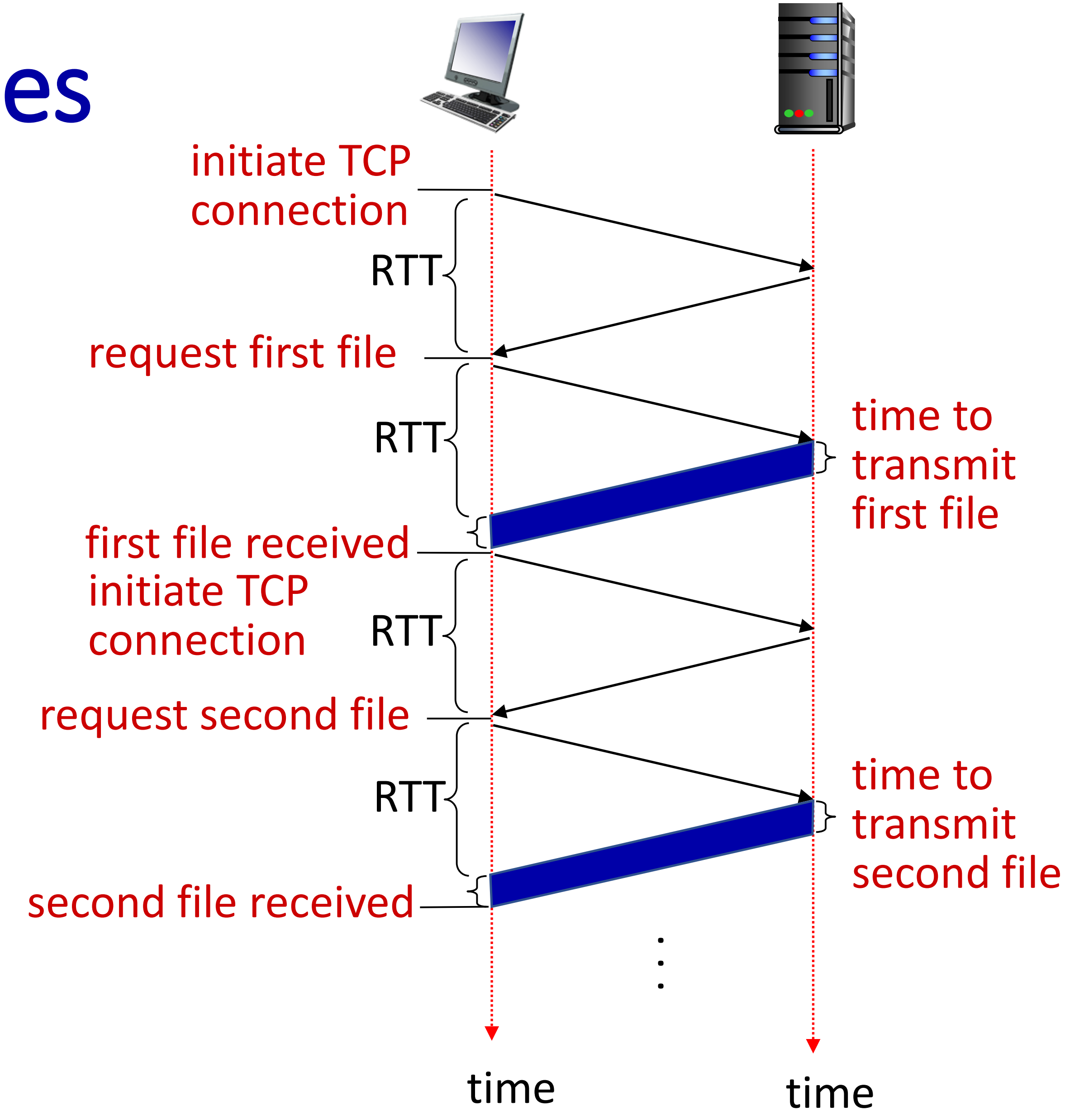
time

time

Non-persistent HTTP response time (per object) = 2RTT + file transmission time (FTT)

Non-persistent HTTP: issues

- requires 2 RTTs per object
  - *Non-persistent response time (N objects)*
    $= \sum_{i=1}^{N}(RTT + RTT + FTT_i)$
    $= 2 \times N \times RTT + \sum_{i=1}^{N} FTT_i$
- OS overhead for *each* TCP connection
- solution: browsers often open *multiple parallel TCP connections* to fetch referenced objects in parallel
  - users can configure some browsers to control the degree of parallelism



initiate TCP connection

RTT

request first file

RTT

time to transmit first file

first file received

initiate TCP connection

RTT

request second file

RTT

time to transmit second file

second file received

time

time

Non-persistent HTTP: types

A. Non-Persistent – with no parallel TCP connections:

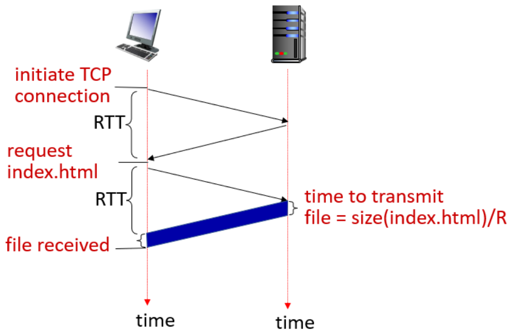
- Each object takes two RTTs, i.e., one to initiate TCP connection and the other for HTTP request and first few bytes of HTTP response to return
- Example: Consider a webpage with 6 embedded images. The client would:
  1) Open a TCP connection for image 1, download it, then close the connection
  2) Repeat the process for each of the remaining 5 images

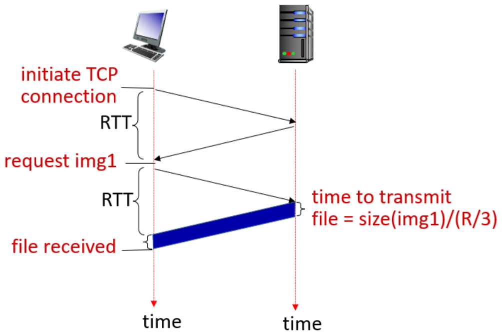
B. Non-Persistent – with parallel TCP connections:

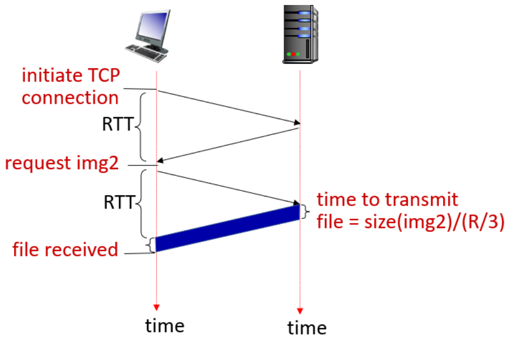
- The client opens multiple TCP connections at the same time to request multiple objects in parallel from the server
- Example: Consider a webpage with 6 embedded images. The client might open 3 connections at the same time:
  1) Download images 1, 2, and 3 concurrently
  2) After those connections close, open 3 new connections to download images 4, 5, and 6

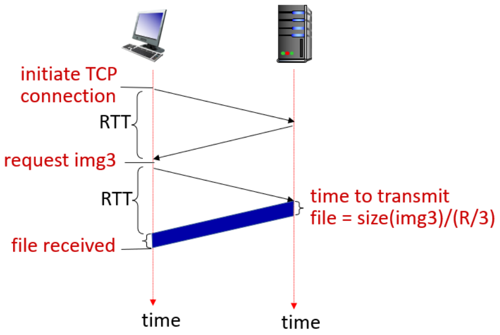
Non-persistent HTTP - With Parallel TCP Connections

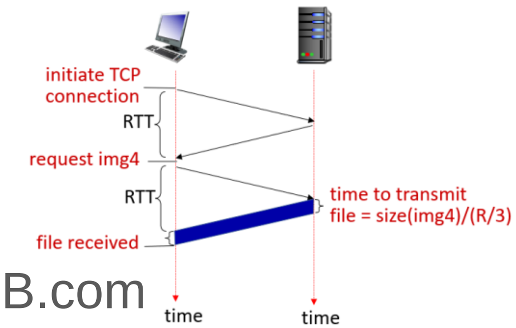
Consider a webpage with 6 embedded images. The client uses non-persistent HTTP with 3 parallel connections.

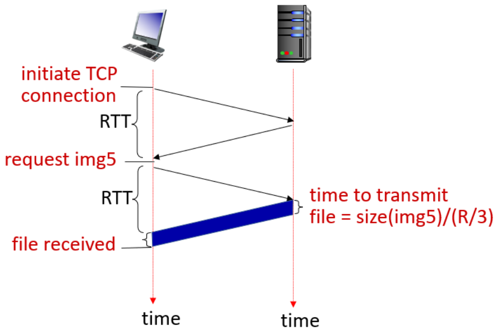


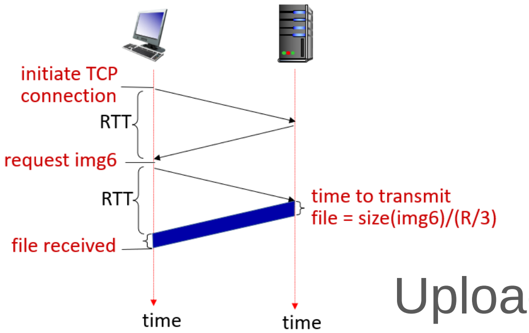












Persistent HTTP (HTTP 1.1)

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

Persistent HTTP Connection:

A. **Non-Pipelined:** the client sends a request to the server and waits for the full response before sending the next request over the same TCP connection

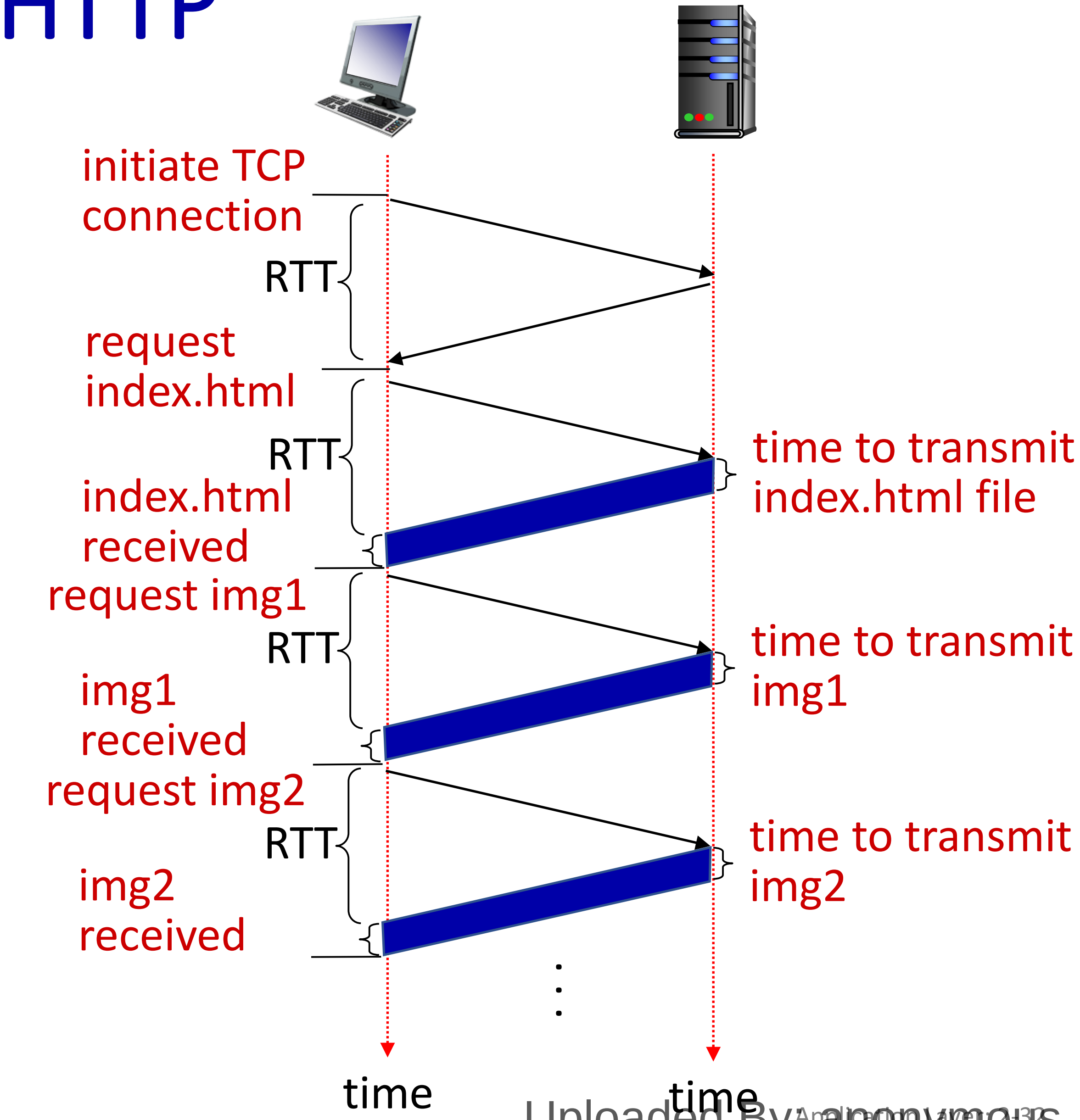
B. **Pipelined:** the client sends multiple requests to the server without waiting for the previous response to arrive. The responses are returned by the server in the same order the requests were received

Non-Pipelined Persistent HTTP

response time (N objects)

$$= RTT + \sum_{i=1}^{N}(RTT + FTT_i)$$

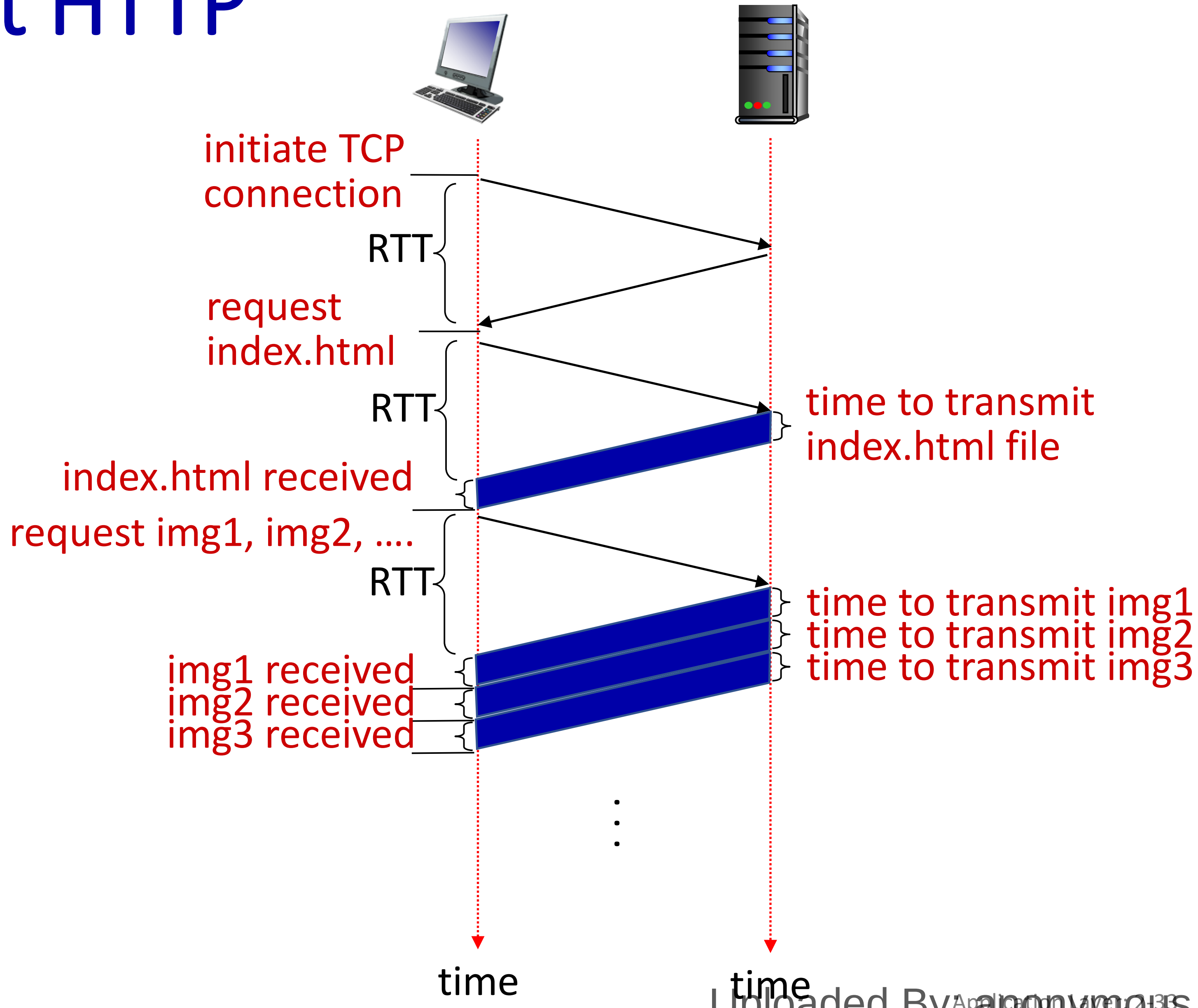
$$= (N + 1) \times RTT + \sum_{i=1}^{N} FTT_i$$



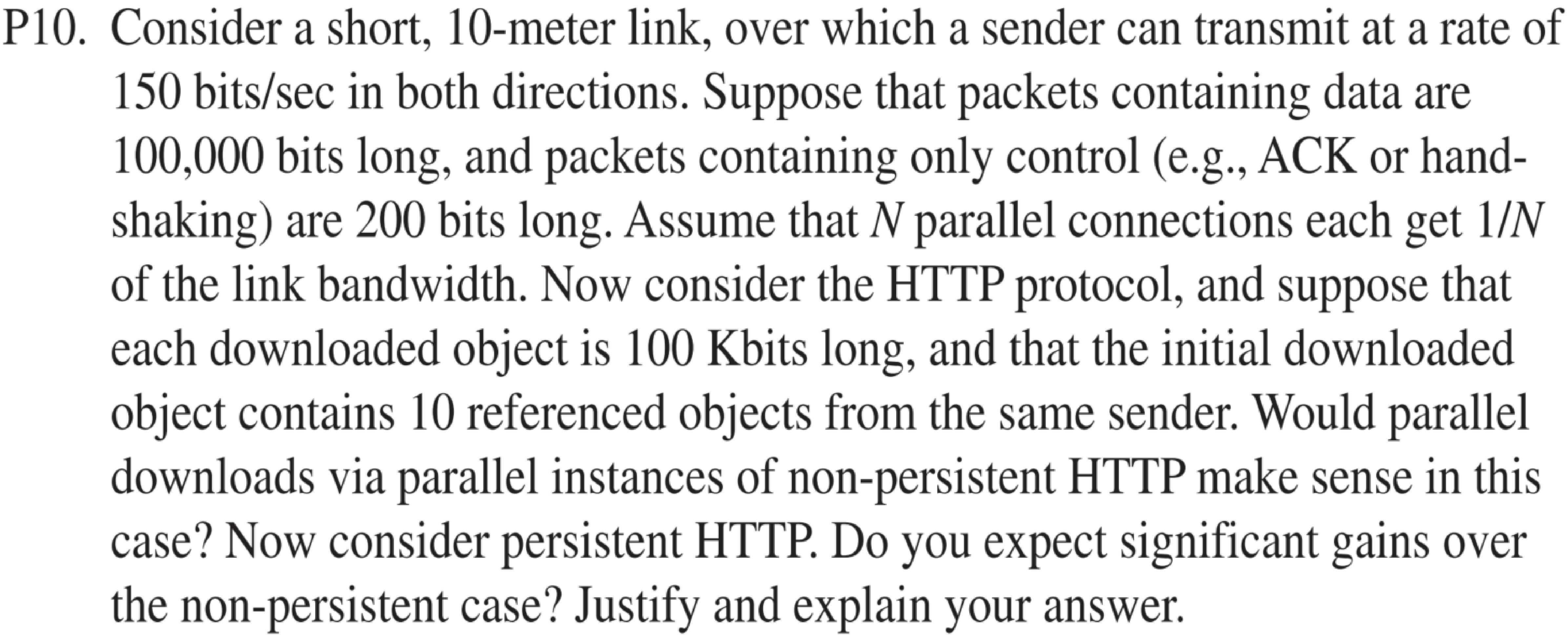
Pipelined Persistent HTTP

response time (N objects)

$$= 3 \times RTT + \sum_{i=1}^{N} FTT_i$$



Example



P10. Consider a short, 10-meter link, over which a sender can transmit at a rate of 150 bits/sec in both directions. Suppose that packets containing data are 100,000 bits long, and packets containing only control (e.g., ACK or handshaking) are 200 bits long. Assume that *N* parallel connections each get 1/*N* of the link bandwidth. Now consider the HTTP protocol, and suppose that each downloaded object is 100 Kbits long, and that the initial downloaded object contains 10 referenced objects from the same sender. Would parallel downloads via parallel instances of non-persistent HTTP make sense in this case? Now consider persistent HTTP. Do you expect significant gains over the non-persistent case? Justify and explain your answer.

Example - Solution

Note that each downloaded object can be completely put into one data packet. Let Tp denote the one-way propagation delay between the client and the server.

First consider parallel downloads using non-persistent connections. Parallel downloads would allow 10 connections to share the 150 bits/sec bandwidth, giving each just 15 bits/sec. Thus, the total time needed to receive all objects is given by:

$$(200/150+Tp + 200/150 + Tp + 200/150+Tp + 100{,}000/150+ Tp)$$
$$+ (200/(150/10)+Tp + 200/(150/10) + Tp + 200/(150/10)+Tp + 100{,}000/(150/10)+ Tp)$$
$$= 7377 + 8*Tp \text{ (seconds)}$$

Now consider a persistent HTTP connection. The total time needed is given by:

$$(200/150+Tp + 200/150 + Tp + 200/150+Tp + 100{,}000/150+ Tp)$$
$$+ \ 10*(200/150+Tp + 100{,}000/150+ Tp)$$
$$=7351 + 24*Tp \text{ (seconds)}$$

Assuming the speed of light is $300*10^6$ m/sec, then $Tp=10/(300*10^6)=0.03$ microsec. Tp is therefore negligible compared with transmission delay.

Thus, we see that persistent HTTP is not significantly faster (less than 1 percent) than the non-persistent case with parallel download.

Example

Suppose within your Web browser you click on a link to obtain a web page. Suppose that the Web page associated with the link contains exactly one object; the base HTML file. Moreover, suppose the base HTML file indexes four more objects. The first and the second indexed reside on the same server hosting the base HTML file. The remaining indexed objects reside on a different server than the server hosting the base HTML file. Assume that $\boldsymbol{RTT_0}$ denotes the RTT between the local host and each server containing an object. Assuming $\boldsymbol{t_{trans}}$ transmission time for each object, find the total amount of time that elapses from when the client clicks on the link until the client receives all 5 objects (i.e., base HTML and 4 indexed objects) with:

1) Persistent HTTP without pipelining.

Total time = $2\ RTT_0 + t_{trans} + 2\ RTT_0 + 2\ t_{trans} + 3\ RTT_0 + 2\ t_{trans} = 7\ RTT_0 + 5\ t_{trans}$

2) Persistent HTTP with pipelining.

Total time = $2\ RTT_0 + t_{trans} + RTT_0 + 2\ t_{trans} + 2\ RTT_0 + 2\ t_{trans} = 5\ RTT_0 + 5\ t_{trans}$

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)

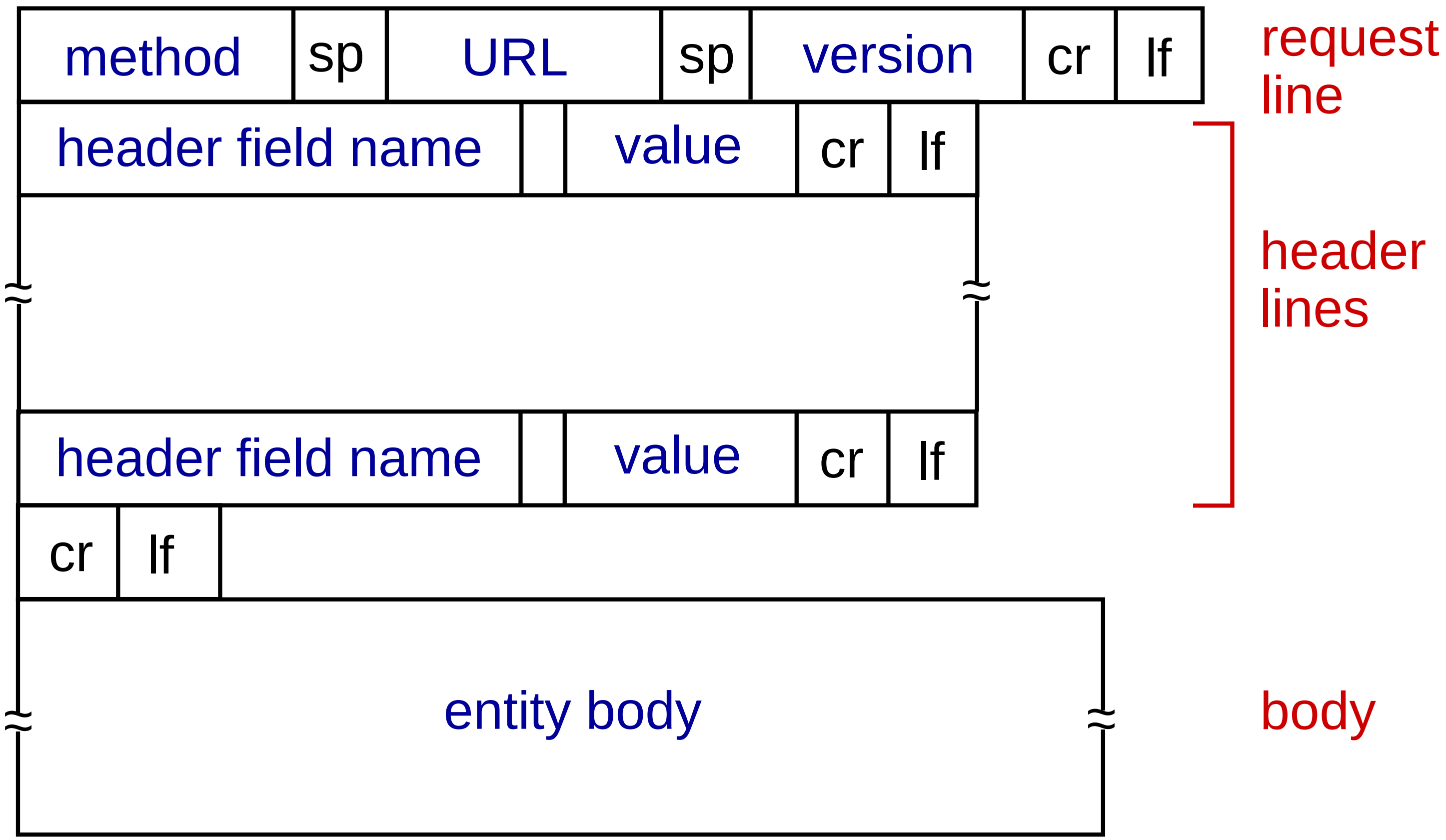
header lines

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: Firefox/3.6.10\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
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\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

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`

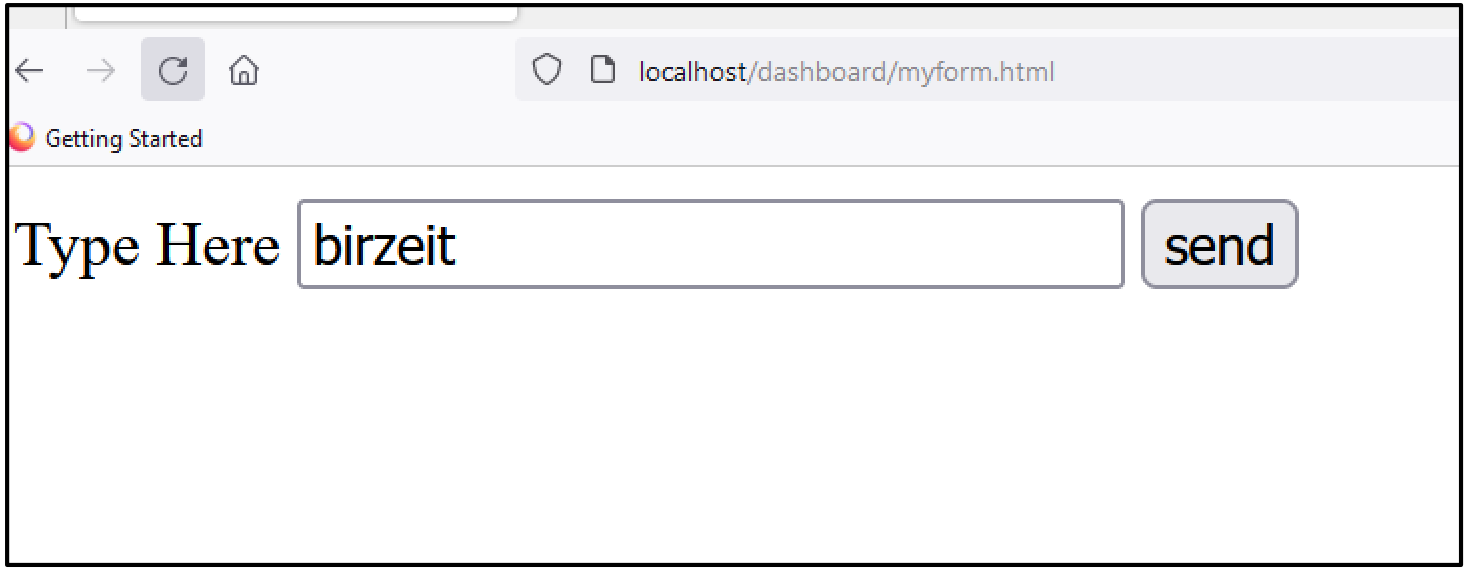
HEAD method:

- 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

Form



localhost/dashboard/myform.html

Getting Started

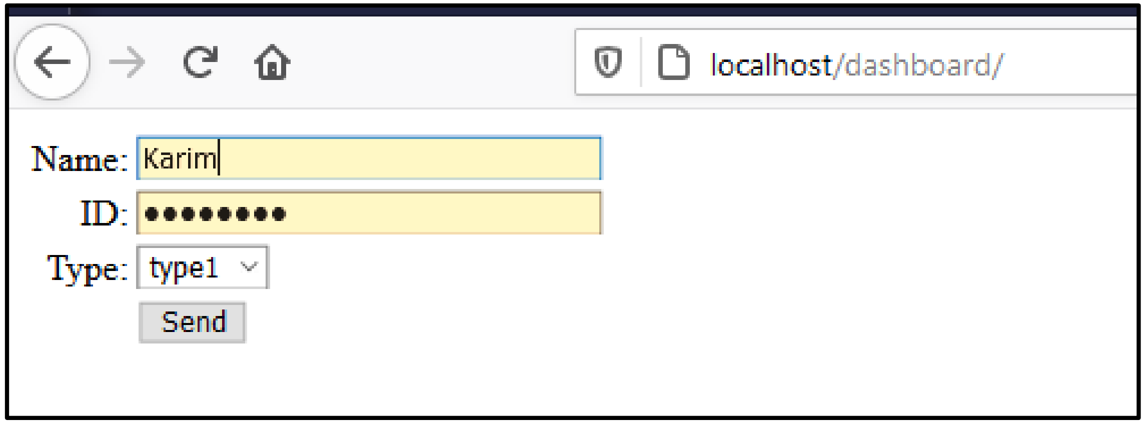
Type Here birzeit send

```
<html>
<head><title>Simple Form</title></head>
<body>
 <form method="get" action="getedata.php" >

Type Here <input type="text" name="name"  size="30"/>
   <input type="submit" value ="send" />

 </form>
</body>
</html>
```

```
<html>
<head><title>Registration Form</title></head>
<body>
 <form method=“post" action="test.php"
enctype="multipart/form-data">
  <table border=0>
   <tr><td align=right>Name:</td><td><input type="text"
name="Name"  size="30"/></td></tr>
   <tr><td align=right>ID:</td><td><input type="password"
name="SID"  size="30"/></td></tr>
   <tr><td align=right>Type:</td><td><select name="type">
     <option value="type1">type1</option>
     <option value="type2">type2</option>
     <option value="type3">type2</option>
     </select></td></tr>
   <tr><td></td><td><input type="submit" name="Send"
value="Send" /></td></tr>
  </table>
 </form>
</body>
</html>
```



localhost/dashboard/

Name: Karim

ID: ●●●●●●●●●

Type: type1

Send

HTTP response message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   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 ...
```

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

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!

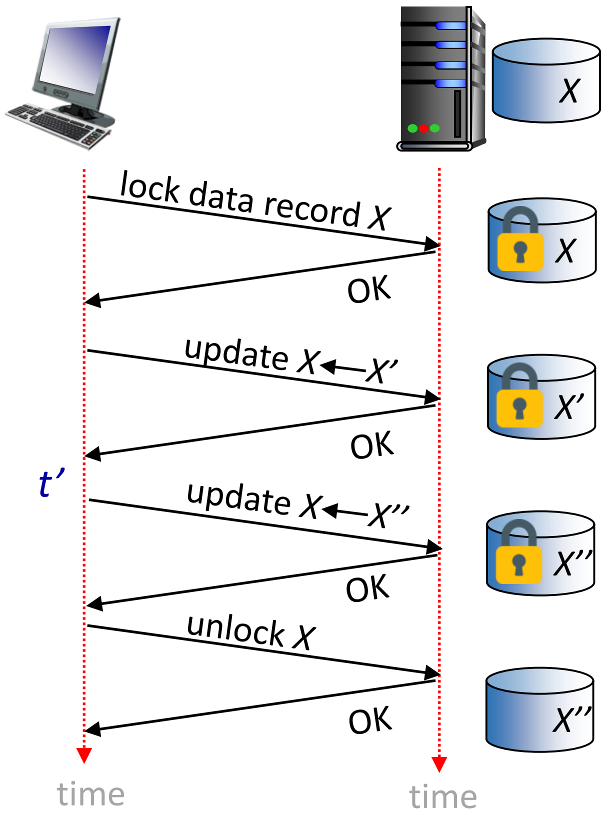
(or use Wireshark to look at captured HTTP request/response)

Maintaining user/server state: cookies

Recall: HTTP request/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

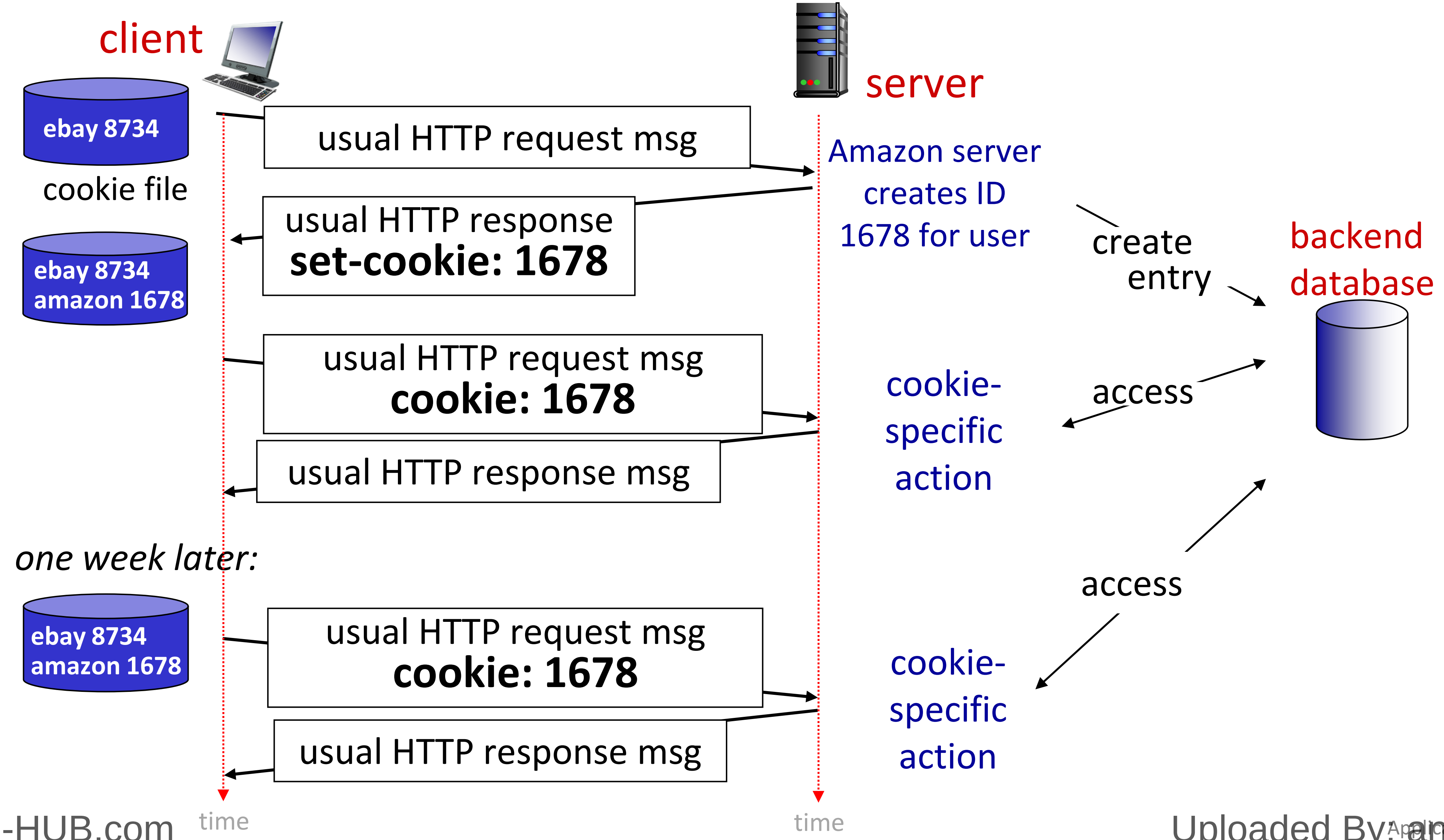
3) cookie file kept on user's host, managed by user's browser

4) back-end database at Web site

Example:

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

Maintaining user/server state: cookies



HTTP cookies: comments

What cookies can be used for:

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

Challenge: How to keep state:

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

aside

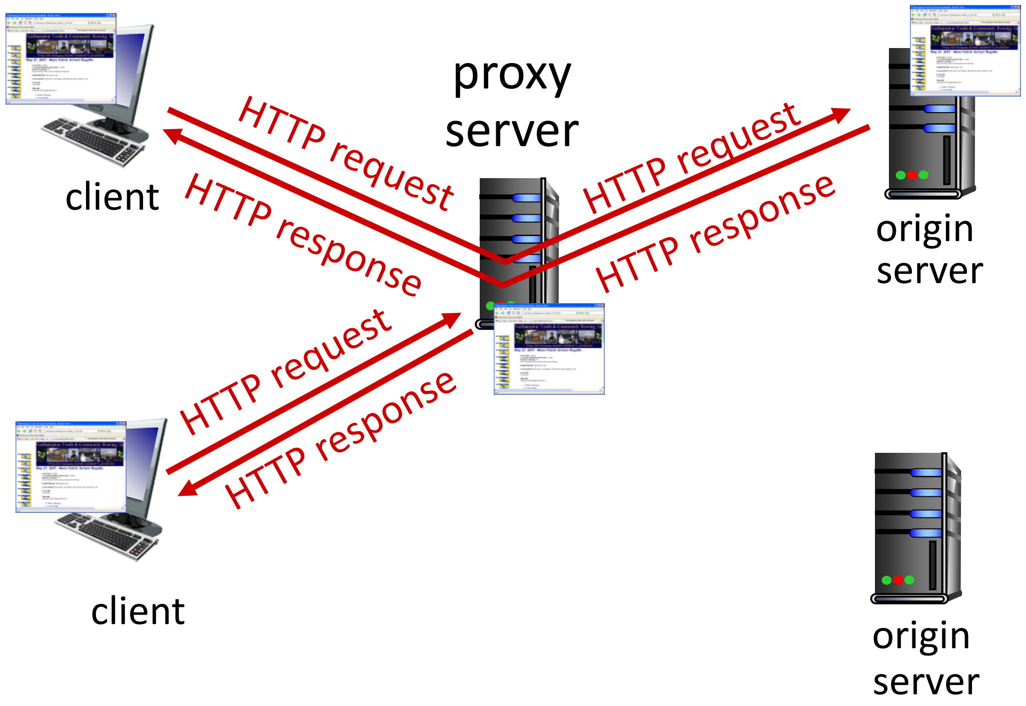
cookies and privacy:

- 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

Web caches (proxy servers)

Goal: satisfy client request without involving origin server

- user configures browser to point to a *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 (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`

- typically, cache is installed by ISP (university, company, …)

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

The total response time —that is, the time from the browser's request of an object until its receipt of the object— is the sum of the LAN delay, the access delay (that is, the delay between the two routers), and the Internet delay.

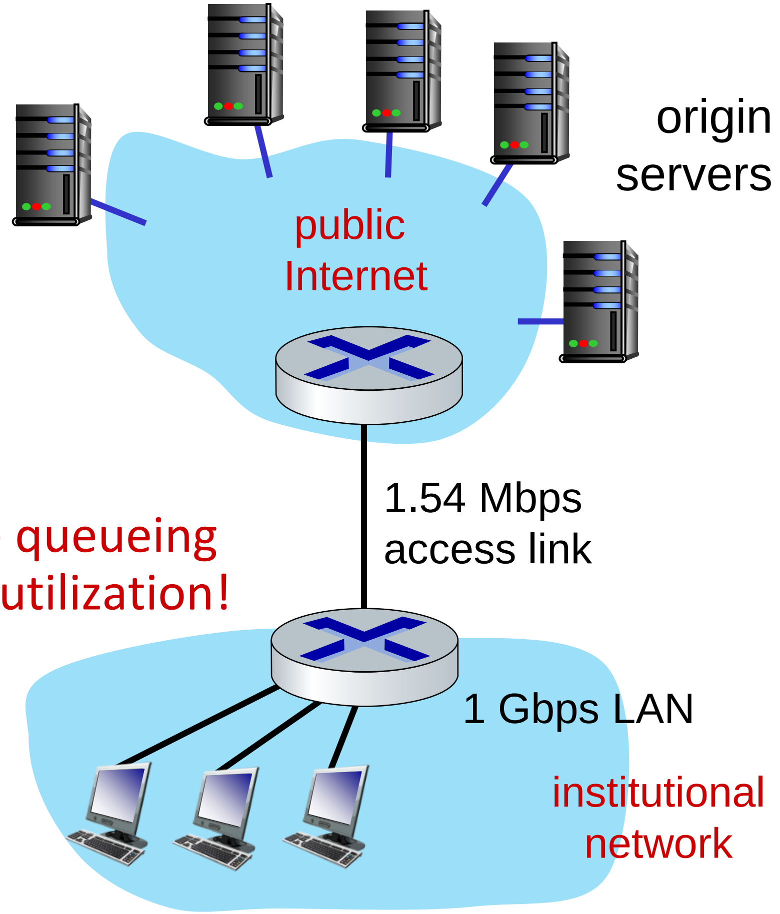
Scenario:

- 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 request/sec
  - average data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization = 0.0015
- access link utilization = 0.97
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + minutes + usecs

problem: large queueing delays at high utilization!



Option 1: buy a faster access link

Scenario:

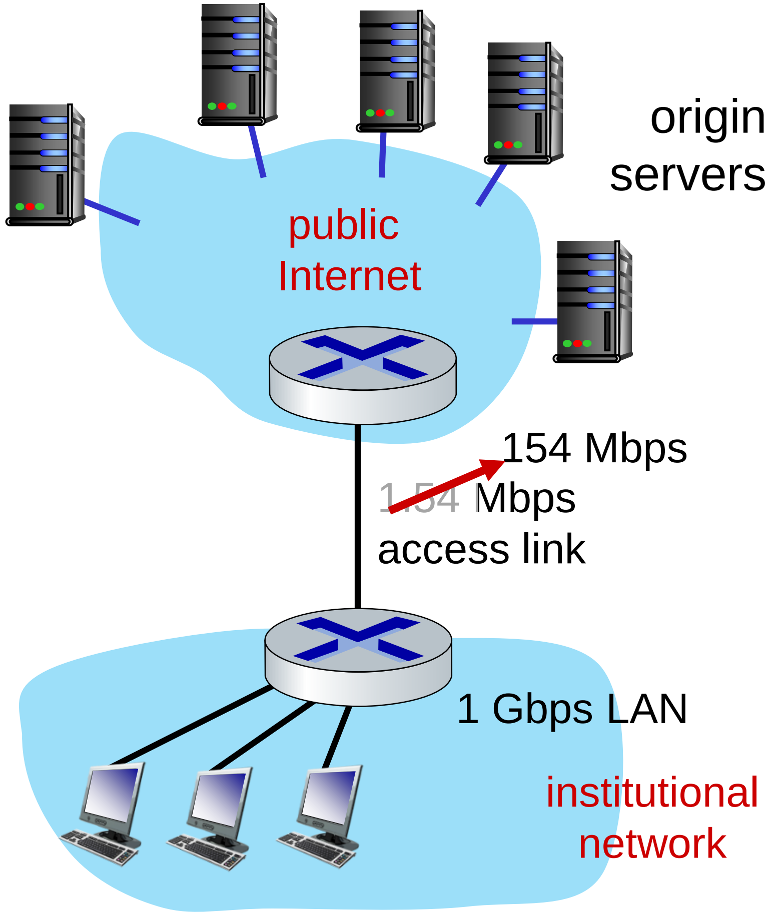
- access link rate: ~~1.54~~ Mbps → 154 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15 request/sec
    - avg data rate to browsers: 1.50 Mbps

Performance:

- LAN utilization: 0.0015
- access link utilization = ~~.97~~ → .0097
- end-end delay = Internet delay + access link delay + LAN delay

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

Cost: faster access link (expensive!)



Option 2: install a web cache

Scenario:

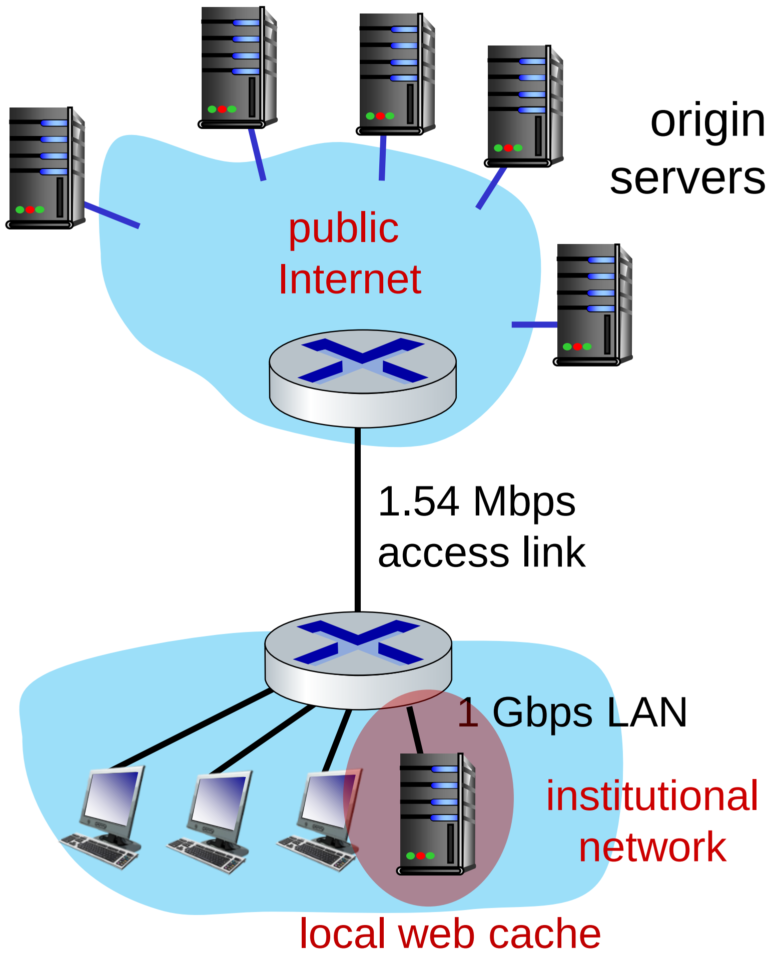
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15 request/sec
  - avg data rate to browsers: 1.50 Mbps

Performance:

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

How to compute link utilization, delay?

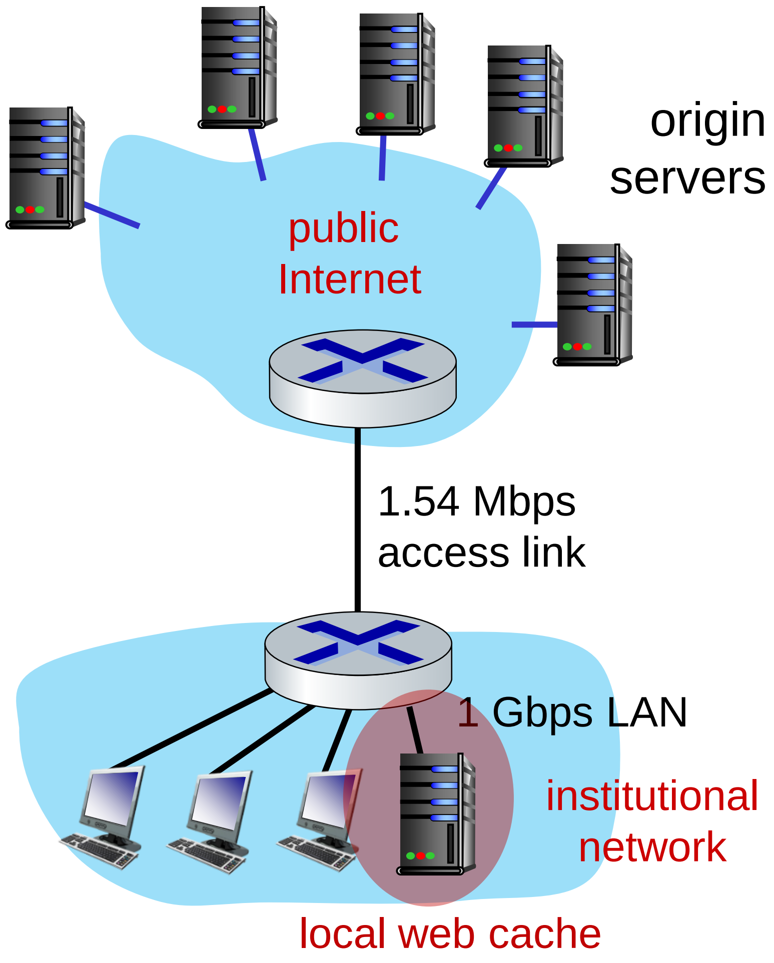
Cost: web cache (cheap!)



Calculating access link utilization, end-end delay with cache

suppose cache hit rate is 0.4:

- 40% requests served by cache, with low (msec) delay
- 60% requests satisfied at origin
  - data rate to browsers over access link
    = 0.6 * 1.50 Mbps = 0.9 Mbps
  - utilization = 0.9/1.54 = 0.58 means low (msec) queueing delay at access link
  - average end-end delay
    = 0.6 * (delay from origin servers)
    + 0.4 * (delay when satisfied at cache)
    = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs



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

Conditional GET

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

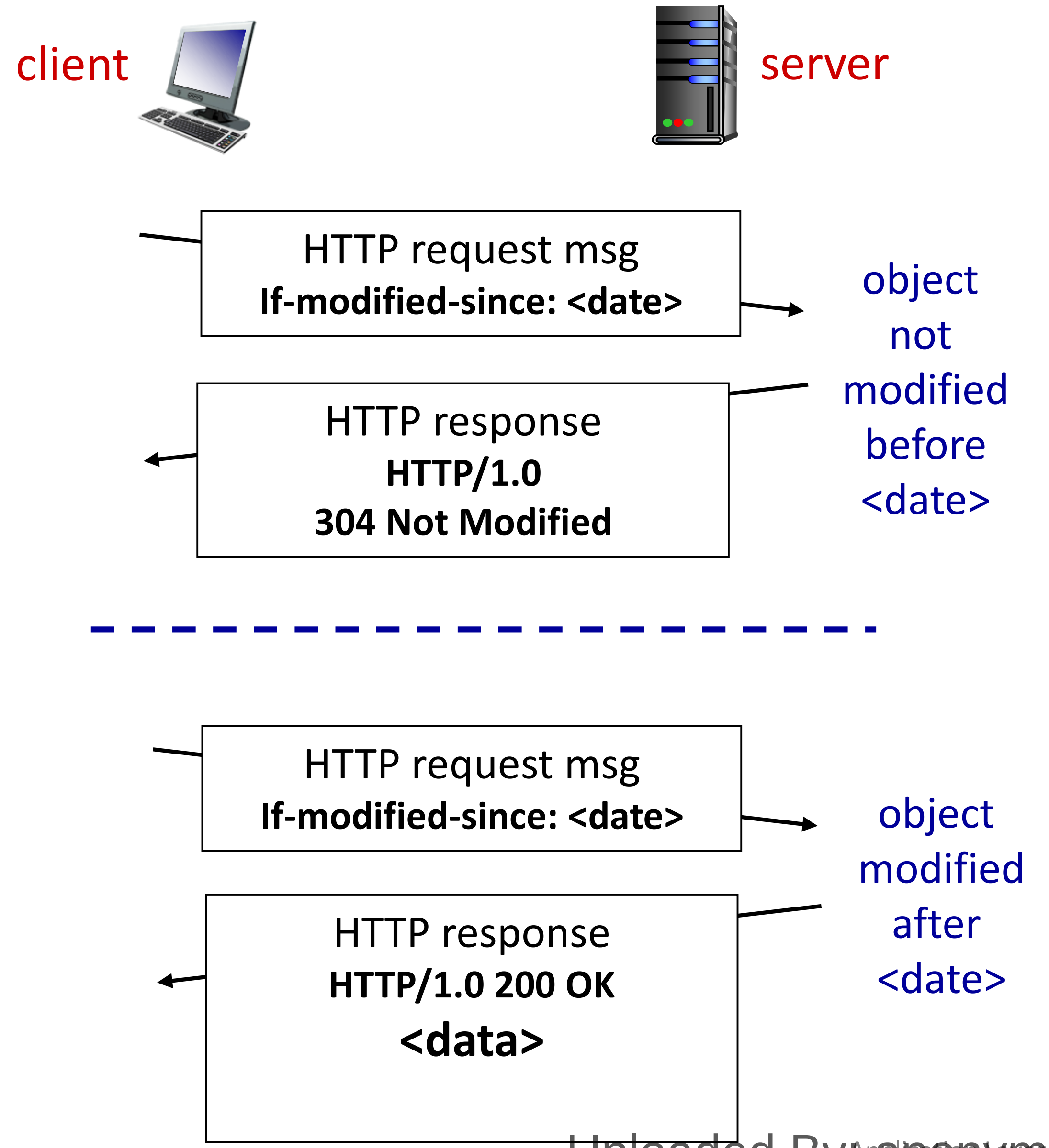
- no object transmission delay
- lower link utilization

- *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



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>

Example (HTTP GET)

A client is sending an HTTP GET message to a web server. Suppose the client-to-server HTTP GET message is the following:

```
GET /kurose_ross_sandbox/interactive/quotation1.htm HTTP/1.1
Host: gaia.cs.umass.edu
Accept: text/plain, text/html, image/jpeg, image/gif, audio/mp4, audio/mpeg, video/mp4, video/wmv,
Accept-Language: en-us, en-gb;q=0.1, en;q=0.4, fr, fr-ch, ar, cs
If-Modified-Since: Mon, 14 Oct 2024 11:43:20 -0700
User Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
```

1) What is the name of the file that is being retrieved in this GET message?

    quotation1.htm

2) What version of HTTP is the client running?

    HTTP/1.1

3) True or False: The client will accept jpeg images.

    True

Example (HTTP GET)

A client is sending an HTTP GET message to a web server. Suppose the client-to-server HTTP GET message is the following:

```
GET /kurose_ross_sandbox/interactive/quotation1.htm HTTP/1.1
Host: gaia.cs.umass.edu
Accept: text/plain, text/html, image/jpeg, image/gif, audio/mp4, audio/mpeg, video/mp4, video/wmv,
Accept-Language: en-us, en-gb;q=0.1, en;q=0.4, fr, fr-ch, ar, cs
If-Modified-Since: Mon, 14 Oct 2024 11:43:20 -0700
User Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
```

4) What is the client's preferred version of English?

American English

5) True or False: The client will accept the German language.

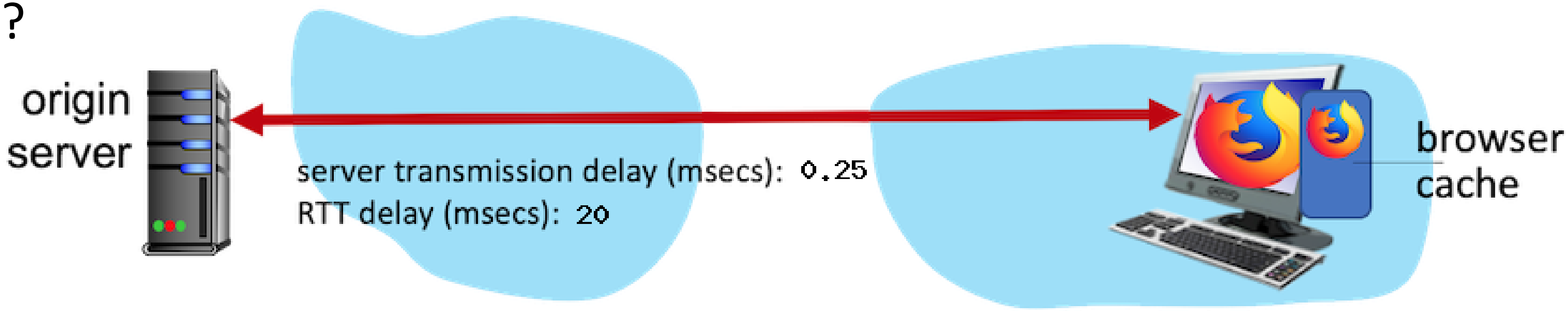
False

6) True or False: The client already has a cached copy of the file.

True

Example (Browser Caching)

Consider an HTTP server and client as shown in the figure below. Suppose that the RTT delay between the client and server is 20 msecs; the time a server needs to transmit an object into its outgoing link is 0.25 msecs; and any other HTTP message not containing an object has a negligible (zero) transmission time. Suppose the client again makes 50 requests, one after the other, waiting for a reply to a request before sending the next request. Assume the client is using HTTP 1.1 and the IF-MODIFIED-SINCE header line. Assume 60% of the objects requested have NOT changed since the client downloaded them (before these 50 downloads are performed). How much time elapses (in milliseconds) between the client transmitting the first request, and the completion of the last request?



50*0.60*(20)+50*0.40*(20+0.25) = 1005 msec

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

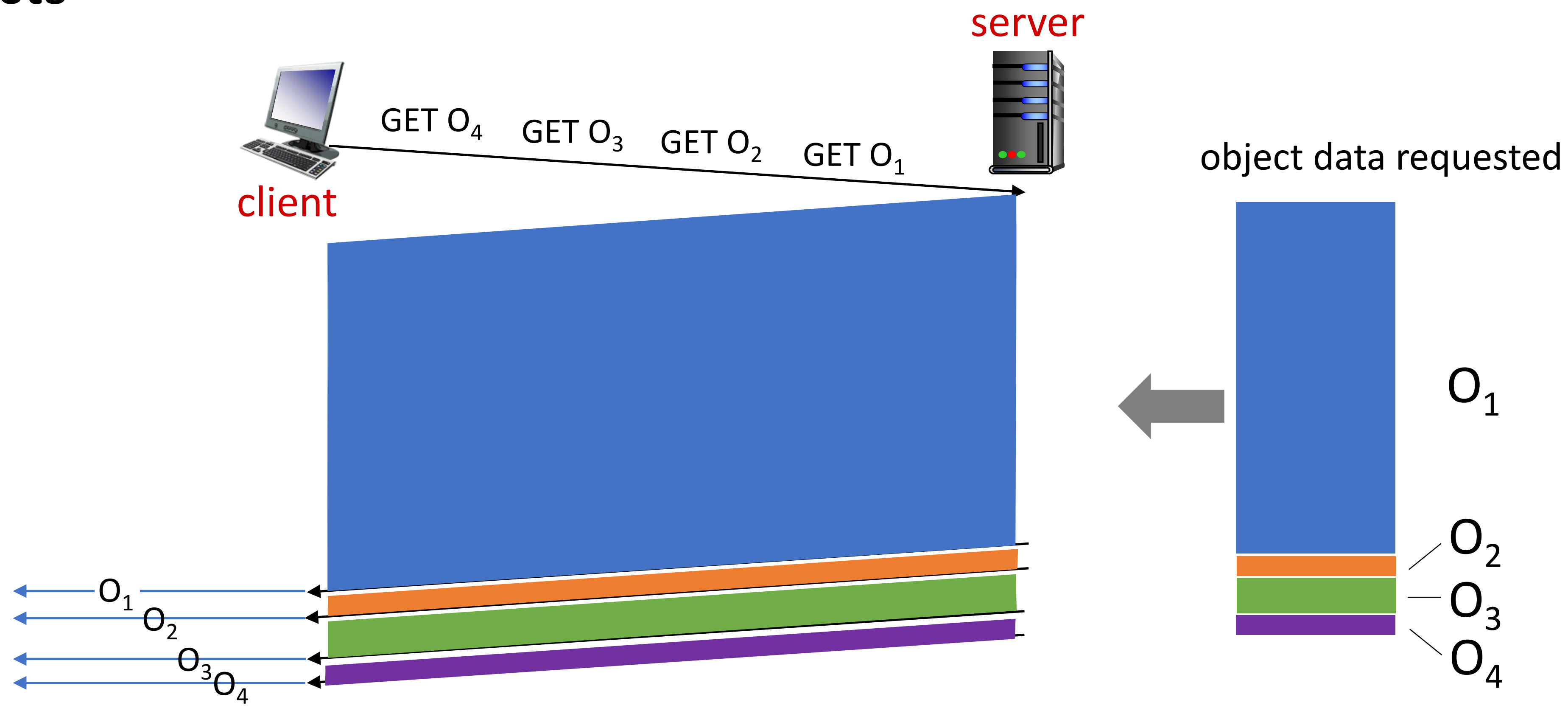
Key goal: decreased delay in multi-object HTTP requests

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

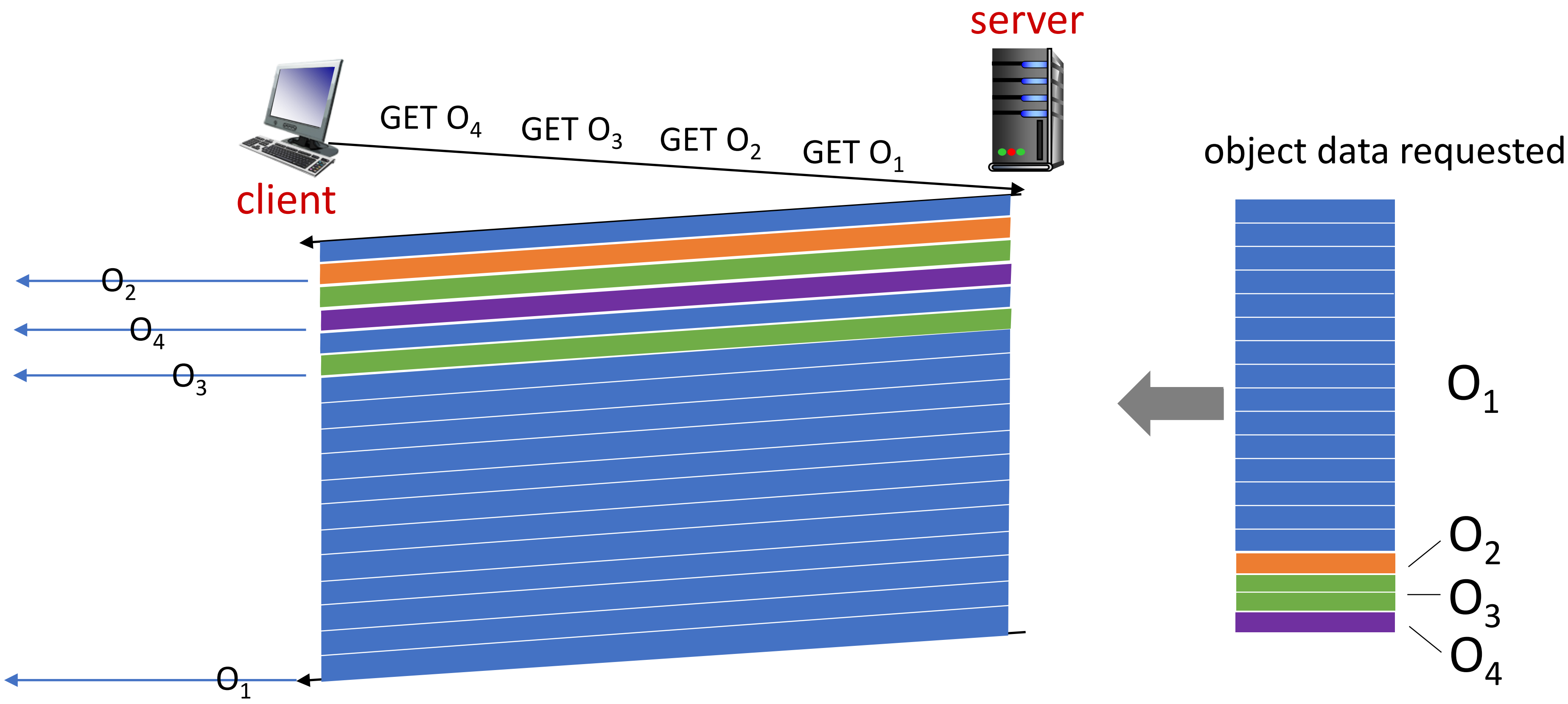
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



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

Key goal: decreased delay in multi-object HTTP requests

HTTP/2 over single TCP connection means:

- recovery from packet loss still stalls all object transmissions
  - as in HTTP 1.1, browsers have incentive to open multiple parallel TCP connections to reduce stalling, increase overall throughput
- no security over vanilla TCP connection
- HTTP/3: adds security, per object error- and congestion-control (more pipelining) over UDP
  - more on HTTP/3 in transport layer

Application layer: overview

- Principles of network applications
- Web and HTTP
- **E-mail, SMTP, IMAP**
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

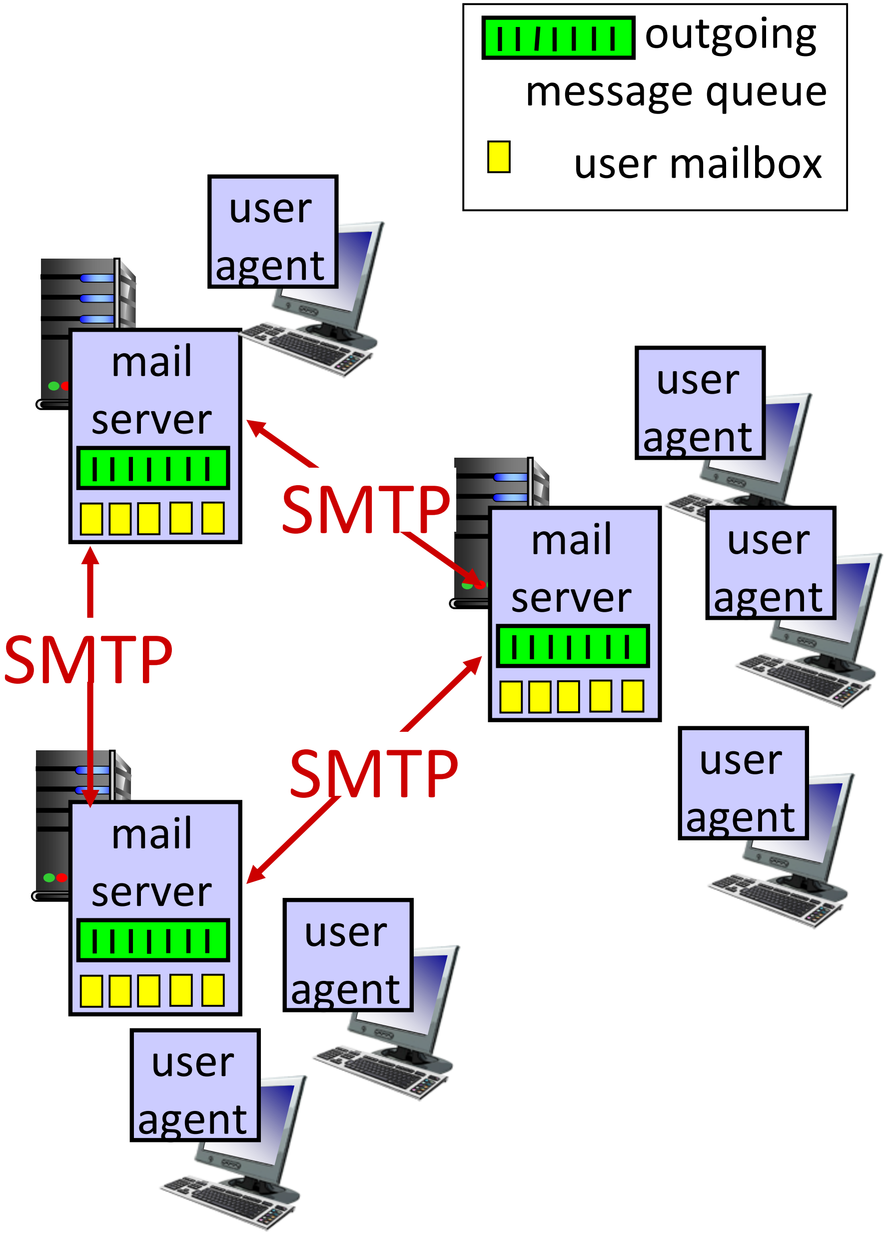
E-mail

Three major components:

- user agents
- mail servers
- simple mail transfer protocol: SMTP

User Agent

- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server



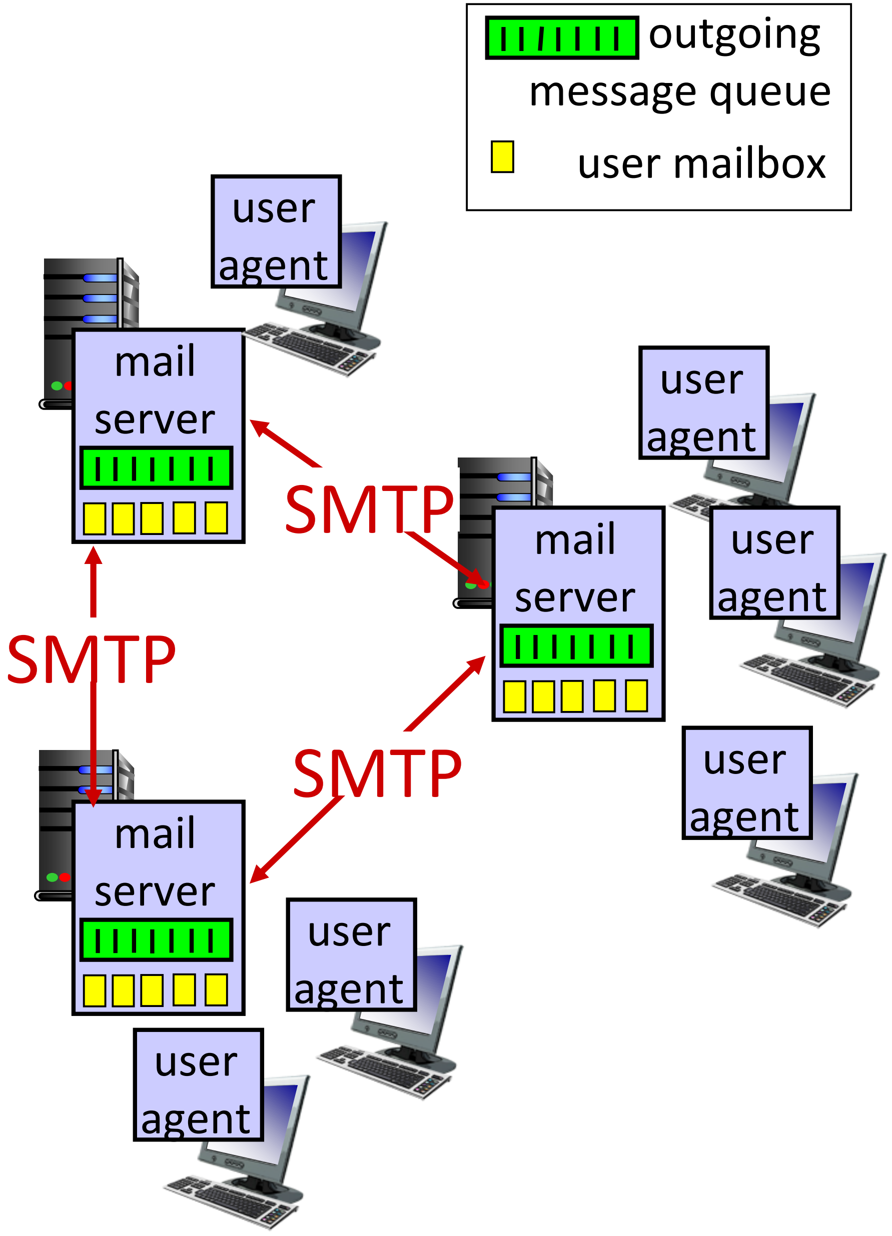
E-mail: mail servers

mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages

SMTP protocol between mail servers to send email messages

- client: sending mail server
- “server”: receiving mail server



Scenario: Alice sends e-mail to Bob

1) Alice uses UA to compose e-mail message "to" bob@someschool.edu

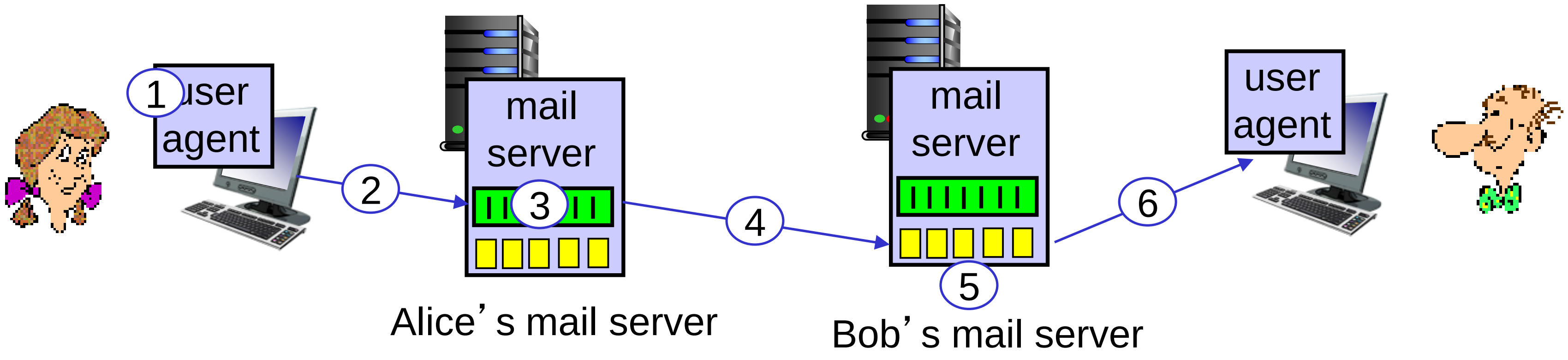
2) Alice's UA sends message to her mail server; message placed in message queue

3) client side of SMTP opens TCP connection with Bob's mail server

4) SMTP client sends Alice's message over the TCP connection

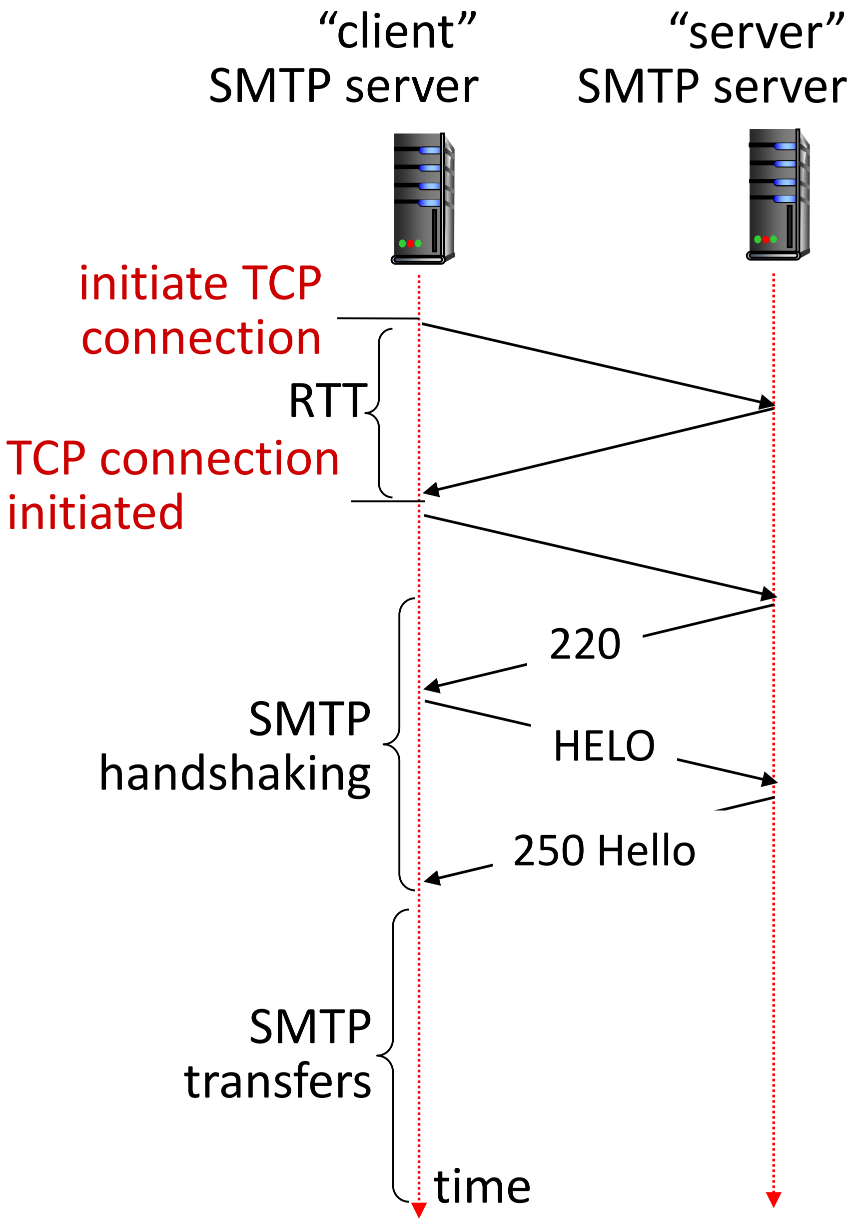
5) Bob's mail server places the message in Bob's mailbox

6) Bob invokes his user agent to read message



SMTP RFC (5321)

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
  - direct transfer: sending server (acting like client) to receiving server
- three phases of transfer
  - SMTP handshaking (greeting)
  - SMTP transfer of messages
  - SMTP closure
- command/response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase



Sample SMTP interaction

220 — SMTP Service ready

221 — Service closing

250 — Requested action taken and completed

354 — Start message input and end with

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

Try SMTP interaction for yourself:

```
telnet <servername> 25
```

- see 220 reply from server
- enter HELO, MAIL FROM:, RCPT TO:, DATA, QUIT commands

above lets you send email without using e-mail client (reader)

Note: this will only work if <servername> allows telnet connections to port 25 (this is becoming increasingly rare because of security concerns)

SMTP: closing observations

comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

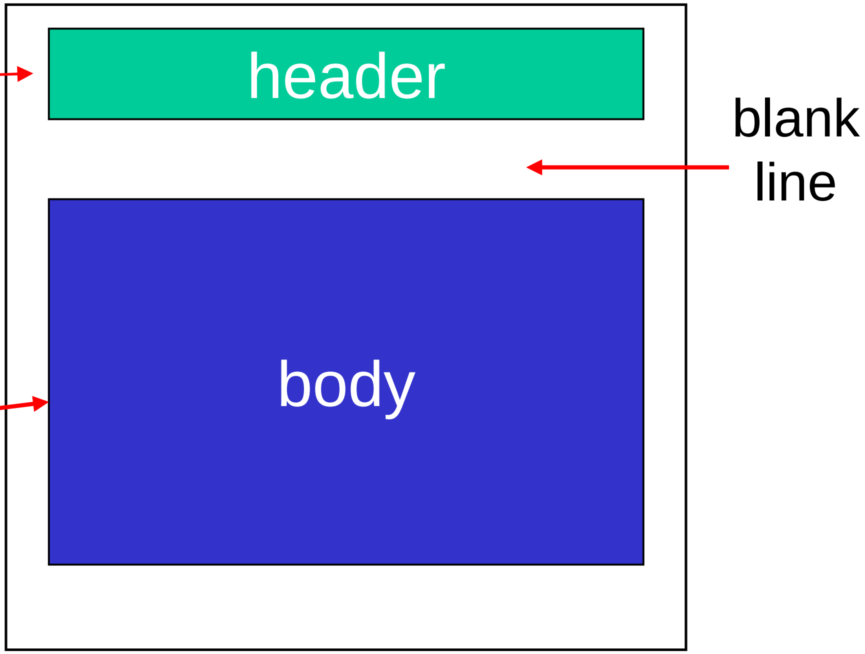
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 5321 (like RFC 7231 defines HTTP)

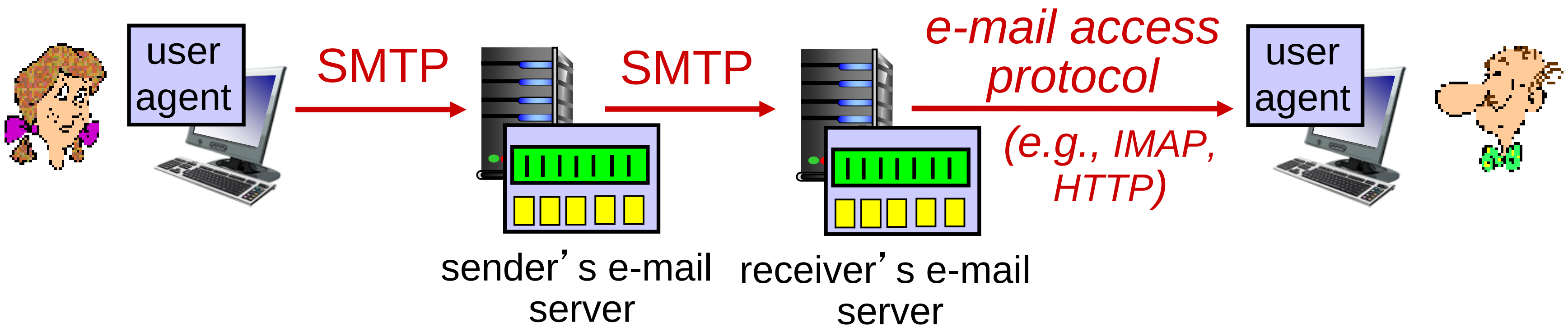
RFC 2822 defines *syntax* for e-mail message itself (like HTML defines syntax for web documents)

- header lines, e.g.,
  - To:
  - From:
  - Subject:

  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the “message” , ASCII characters only



Mail access protocols


user agent

SMTP

SMTP

e-mail access protocol

(e.g., IMAP, HTTP)

user agent

sender's e-mail server

receiver's e-mail server

- SMTP: delivery/storage of e-mail messages to receiver's server
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
- HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of SMTP (to send), IMAP (or POP) to retrieve e-mail messages

Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS

- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

DNS: Domain Name System

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., cs.umass.edu - used by humans

Q: how to map between IP address and name, and vice versa ?

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, *implemented as application-layer protocol*
  - complexity at network’s “edge”

DNS: services, structure

DNS services

- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

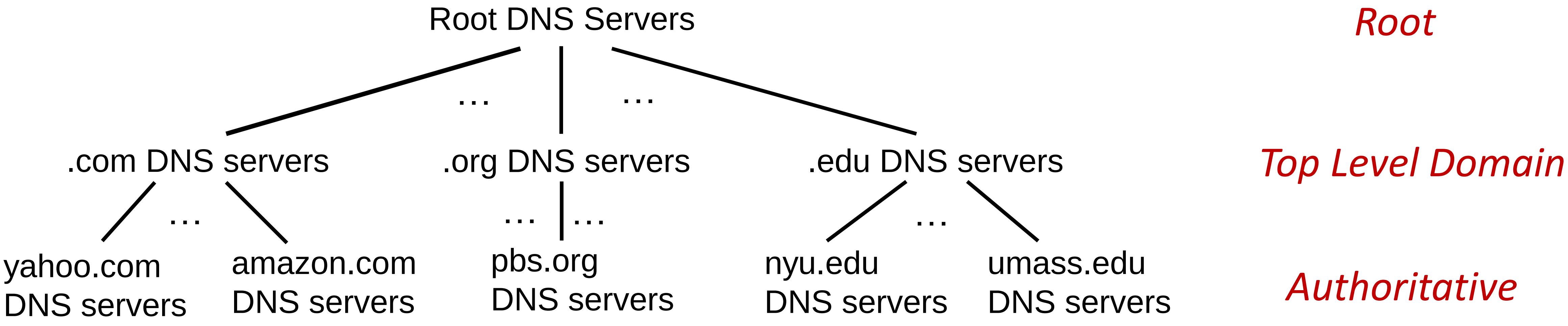
Q: Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance

A: doesn't scale!

- Comcast DNS servers alone: 600B DNS queries per day

DNS: a distributed, hierarchical database



Root DNS Servers

Root

.com DNS servers

.org DNS servers

.edu DNS servers

Top Level Domain

yahoo.com DNS servers

amazon.com DNS servers

pbs.org DNS servers

nyu.edu DNS servers

umass.edu DNS servers

Authoritative

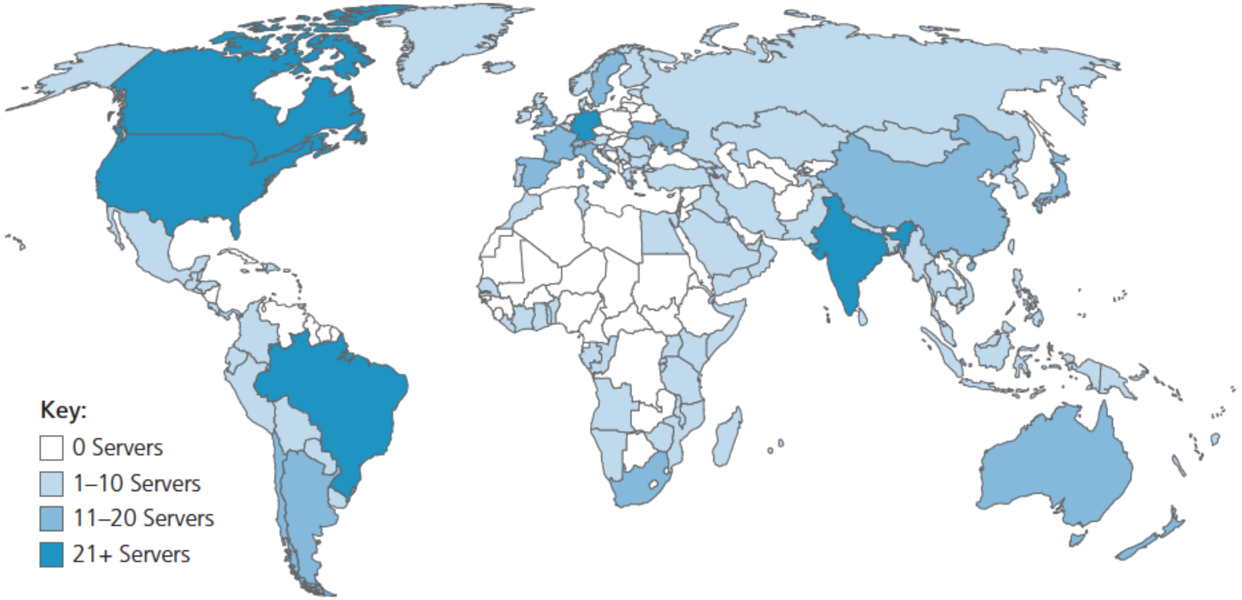
Client wants IP address for www.amazon.com; 1st approximation:

- client queries root server to find .com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS: root name servers

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication and message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name "servers" worldwide each "server" replicated many times (~200 servers in US)



TLD: authoritative servers

Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp
- Network Solutions: authoritative registry for .com, .net TLD
- Educause: .edu TLD

Authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

Local DNS name servers

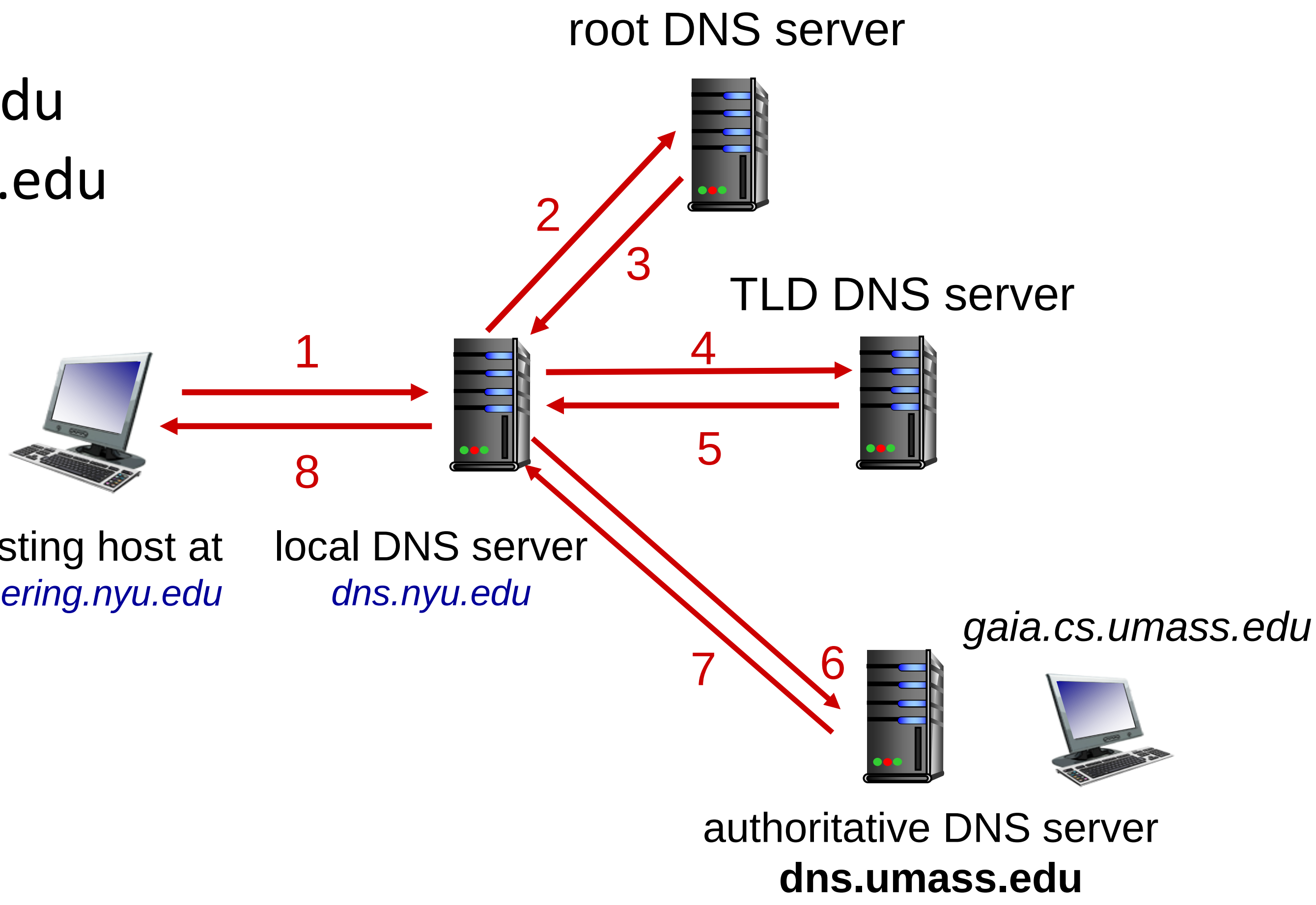
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

DNS name resolution: iterated query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

Iterated query:

- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”



root DNS server

TLD DNS server

requesting host at *engineering.nyu.edu*

local DNS server *dns.nyu.edu*

gaia.cs.umass.edu

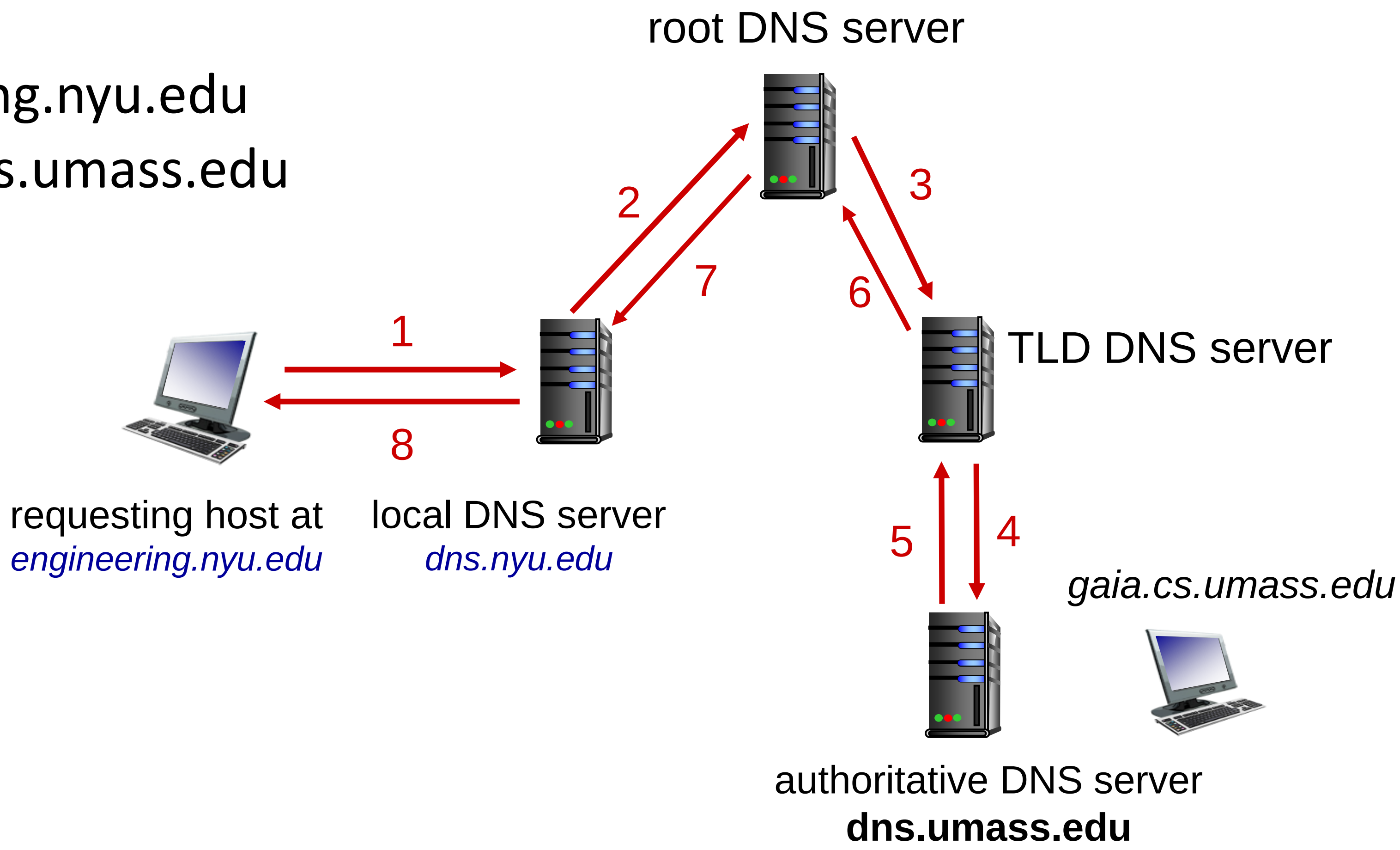
authoritative DNS server **dns.umass.edu**

DNS name resolution: recursive query

Example: host at engineering.nyu.edu wants IP address for gaia.cs.umass.edu

Recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



Caching, Updating DNS Records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - Thus, root name servers not often visited
- cached entries may be *out-of-date* (best-effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire!
- update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS records

DNS: distributed database storing resource records (RR)

RR format: (`name, value, type, ttl`)

type = A

- `name` is hostname
- `value` is IP address
  - Example: (relay1.bar.foo.com, 145.37.93.126, A)

type = NS

- `name` is domain (e.g., foo.com)
- `value` is hostname of authoritative name server for this domain
  - Example: (foo.com, dns.foo.com, NS)

type = CNAME

- `name` is alias hostname for some “canonical” (the real) hostname
- `value` is canonical hostname
  - *www.ibm.com* is really *servereast.backup2.ibm.com*
  - Example: (foo.com, relay1.bar.foo.com, CNAME)

type = MX

- `value` is the canonical name of a mail server that has an alias hostname `name`
  - Example: (foo.com, mail.bar.foo.com, MX)

Inserting records into DNS

Example: new startup "Network Utopia"

- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts `NS`, `A` RRs into `.com` TLD server:

    `(networkutopia.com, dns1.networkutopia.com, NS)`

    `(dns1.networkutopia.com, 212.212.212.1, A)`
- create authoritative server locally with IP address `212.212.212.1`
  - type `A` record for `www.networkuptopia.com`
  - type `MX` record for `mail.networkutopia.com`

Example:

Assume a company “birzeit” has two DNS servers: dns1.birzeit.com with IP address 77.167.21.7 and dns2.birzeit.com with IP address 77.167.21.40,

What resource records (RRs) do you need to provide to the upper-level “.com” Registrar?

(birzeit.com, dns1.birzeit.com, NS)
(dns1.birzeit.com, 77.167.21.7, A)

(birzeit.com, dns2.birzeit.com, NS)
(dns2.birzeit.com, 77.167.21.40, A)

Example:

Suppose you open a startup company “encs3320” and want to set up your company network. Your network has the following servers:

- DNS server: “dns1.encs3320.com” with IP as “128.119.12.40”
- Web server: “encs3320.com” with two IP as “128.119.12.55” and “128.119.12.56”. The web server also has a name as www.encs3320.com
- Email server: “mail.encs3320.com” with IP as “128.119.12.60”. Your company’s email address is “username@encs3320.com”

a) What resource records (RRs) do you need to provide to the upper-level “.com” Registrar?

(encs3320.com, dns1.encs3320.com, NS)
(dns1.encs3320.com, 128.119.12.40, A)

b) What RRs do you need to put in your company’s DNS server?

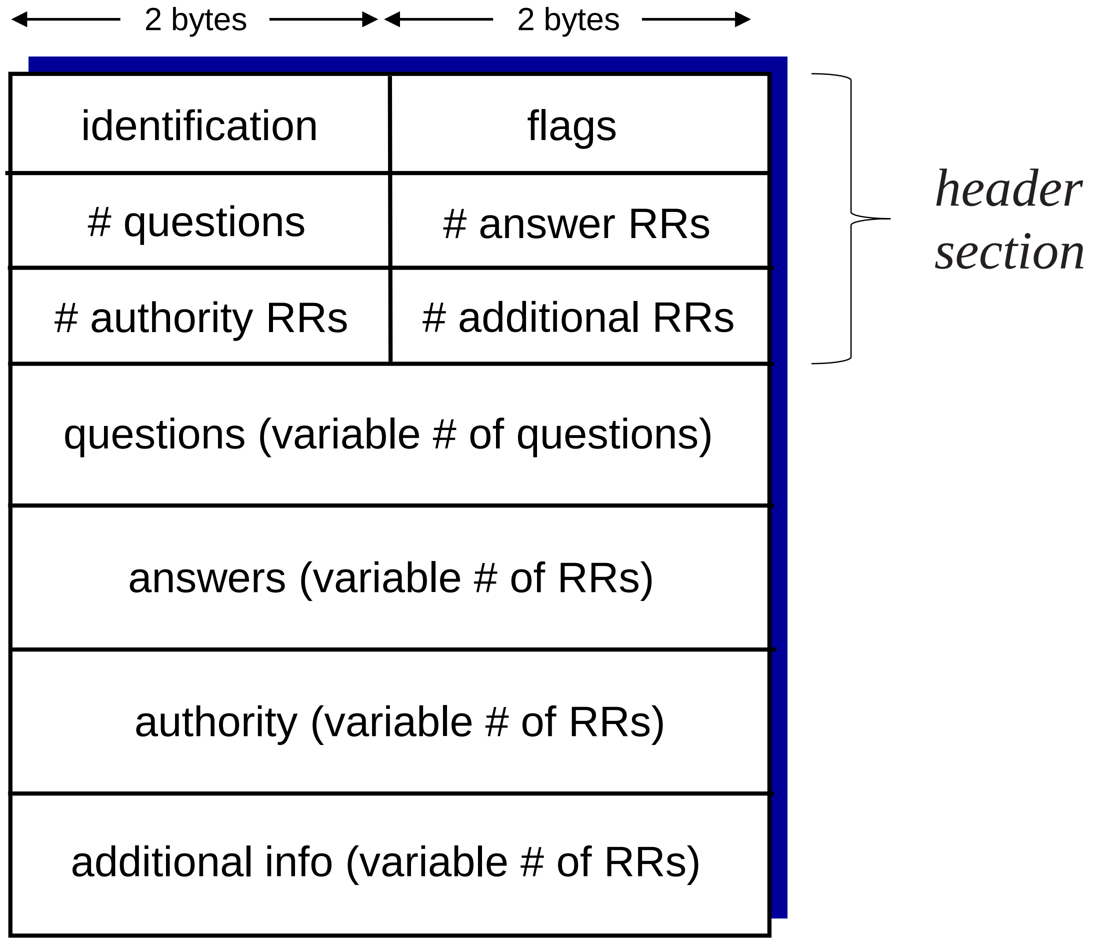
(encs3320.com, 128.119.12.55, A)
(encs3320.com, 128.119.12.56, A)
(www.encs3320.com, encs3320.com, CNAME)
(encs3320.com, mail.encs3320.com, MX)
(mail.encs3320.com, 128.119.12.60, A)

DNS protocol messages

DNS *query* and *reply* messages, both have same *format:*

message header:

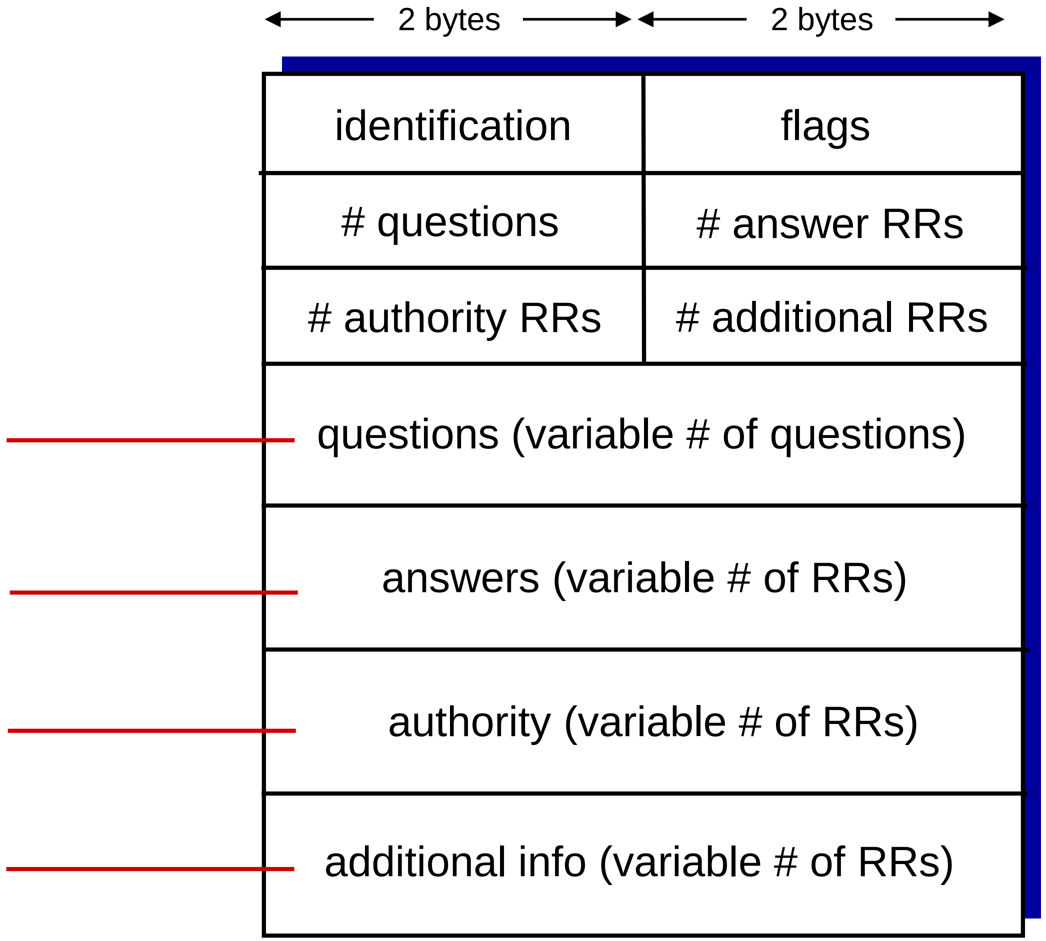
- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query (0) or reply (1)
  - recursion desired
  - recursion available
  - reply is authoritative
- four number-of fields: indicate the number of occurrences of the four types of data sections that follow the header



2 bytes	2 bytes
identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

DNS protocol messages

DNS *query* and *reply* messages, both have same *format:*



name, type fields for a query

RRs in response to query

records for authoritative servers

other "helpful" info that may be used

DNS security

DDoS attacks

- bombard root servers with traffic
  - not successful to date
  - traffic filtering
  - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
  - potentially more dangerous
  - Mirai malware: for almost a full day, Amazon, Twitter, Netflix, Github and Spotify were disturbed

Redirect attacks

- man-in-the-middle
  - intercept DNS queries from hosts and returns bogus replies
- DNS poisoning
  - send bogus relies to DNS server, which caches

Exploit DNS for DDoS

- send queries with spoofed source address: target IP
- requires amplification

DNSSEC
[RFC 4033]

Example:

Suppose within your Web browser you submit a URL to obtain a web page. Assume the following:

a) The base HTML file indexes two **(2) objects**. Both objects reside on the **same server** hosting the base HTML file (Origin Server 1).

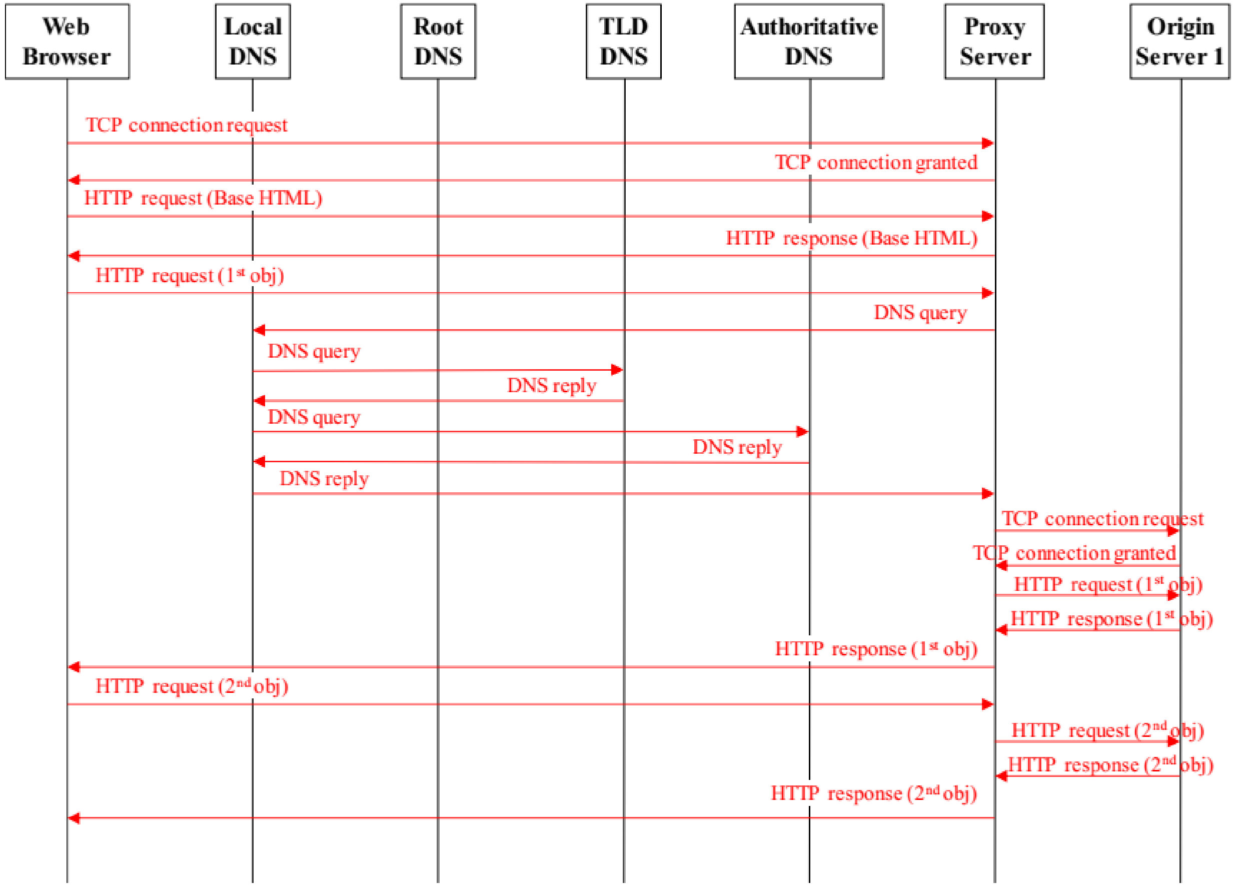
b) The **local proxy server is used**, and has **no existing TCP** connections established.

c) The base HTML file **is cached and is up-to-date**. On the other hand, the **two objects are not cached**.

d) The IP address of the server hosting the base HTML file is not known to the local proxy server.

e) If needed, an **iterative DNS query** is used, and the IP address of only the **TLD DNS** is known to the Local DNS. In addition, the requested IP address is only cached by the authoritative DNS server.

f) Persistent HTTP without pipelining is used.

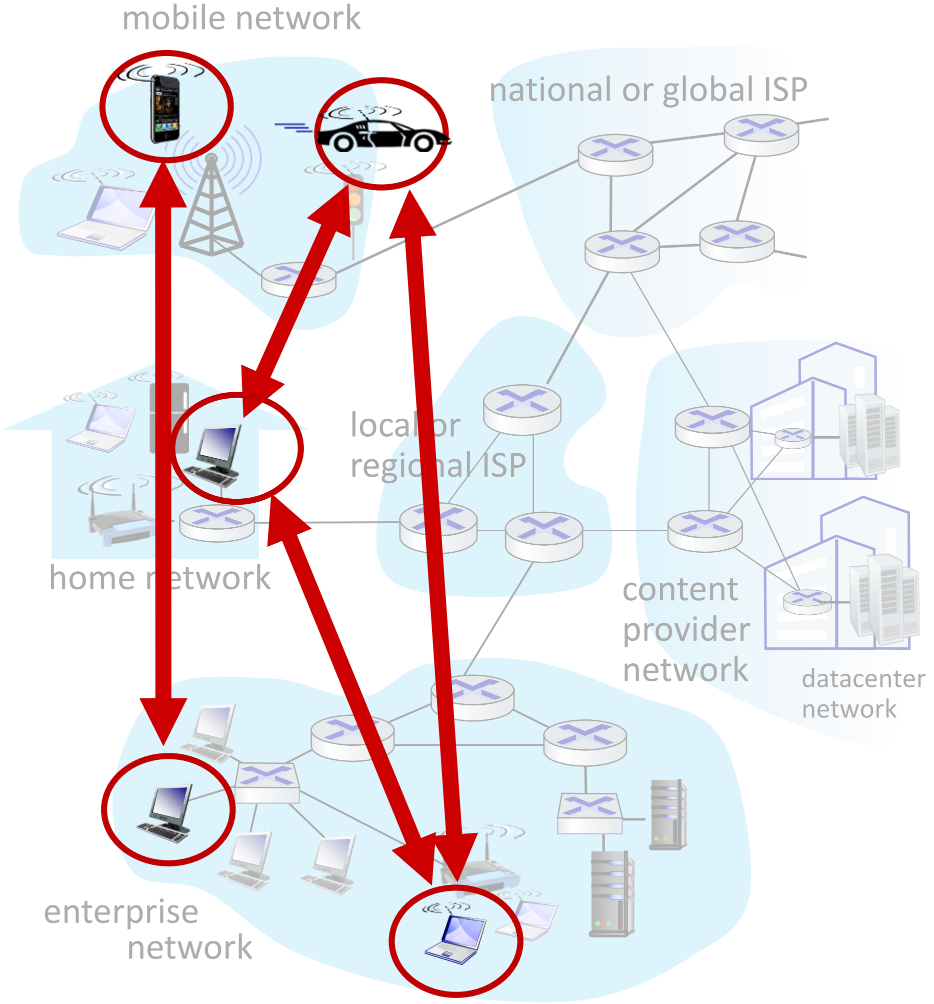


Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- **P2P applications**
- video streaming and content distribution networks
- socket programming with UDP and TCP

Peer-to-peer (P2P) architecture

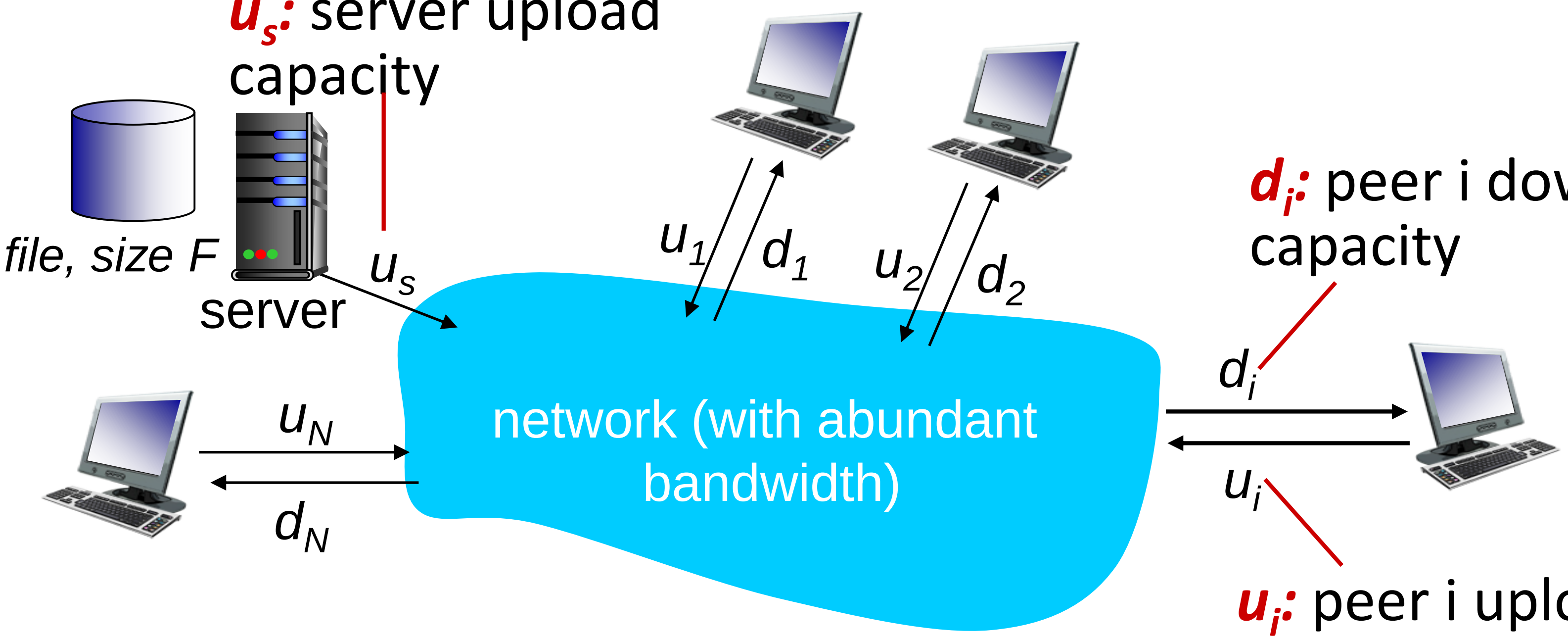
- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)



File distribution: client-server vs P2P

Q: how much time to distribute file (size *F*) from one server to *N peers*?

- peer upload/download capacity is limited resource



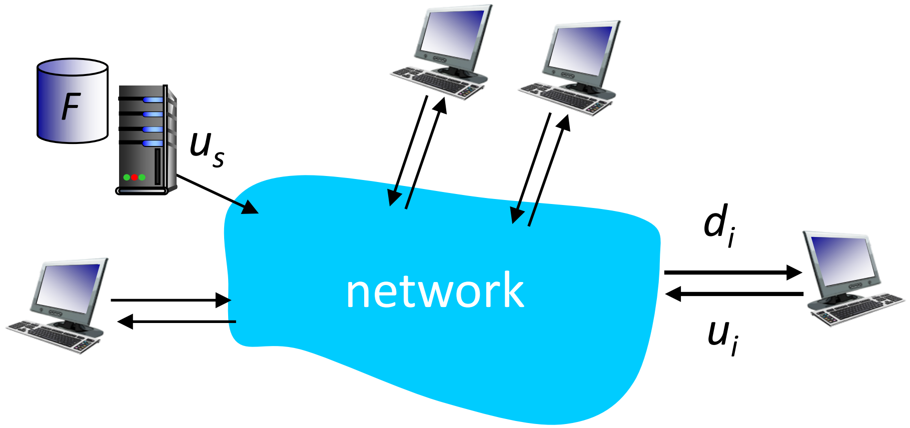
u_s: server upload capacity

d_i: peer i download capacity

u_i: peer i upload capacity

File distribution time: client-server

- *server transmission:* must send (upload) *N* file copies
  - time to send one copy: F/u_s
  - time to send *N* copies: NF/u_s
- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}



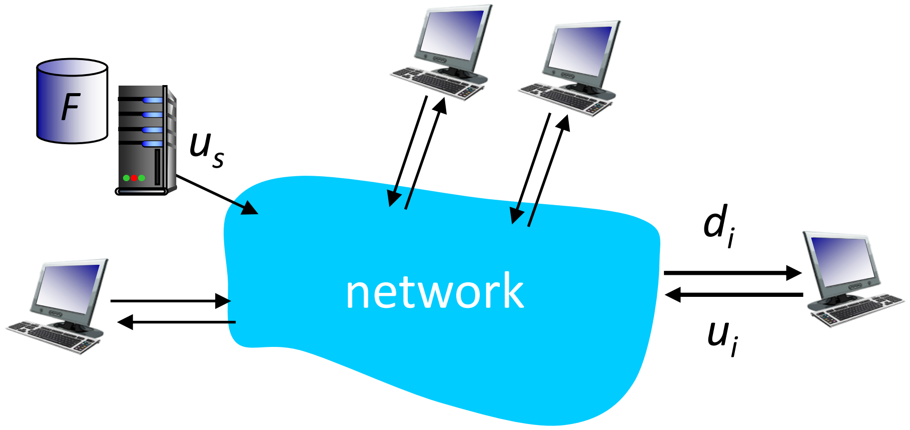
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} \geq max\{NF/u_s\,,\; F/d_{min}\}$$

increases linearly in N

File distribution time: P2P

- *server transmission:* must upload at least one copy
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients:* as aggregate must download *NF* bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$



time to distribute F to N clients using P2P approach

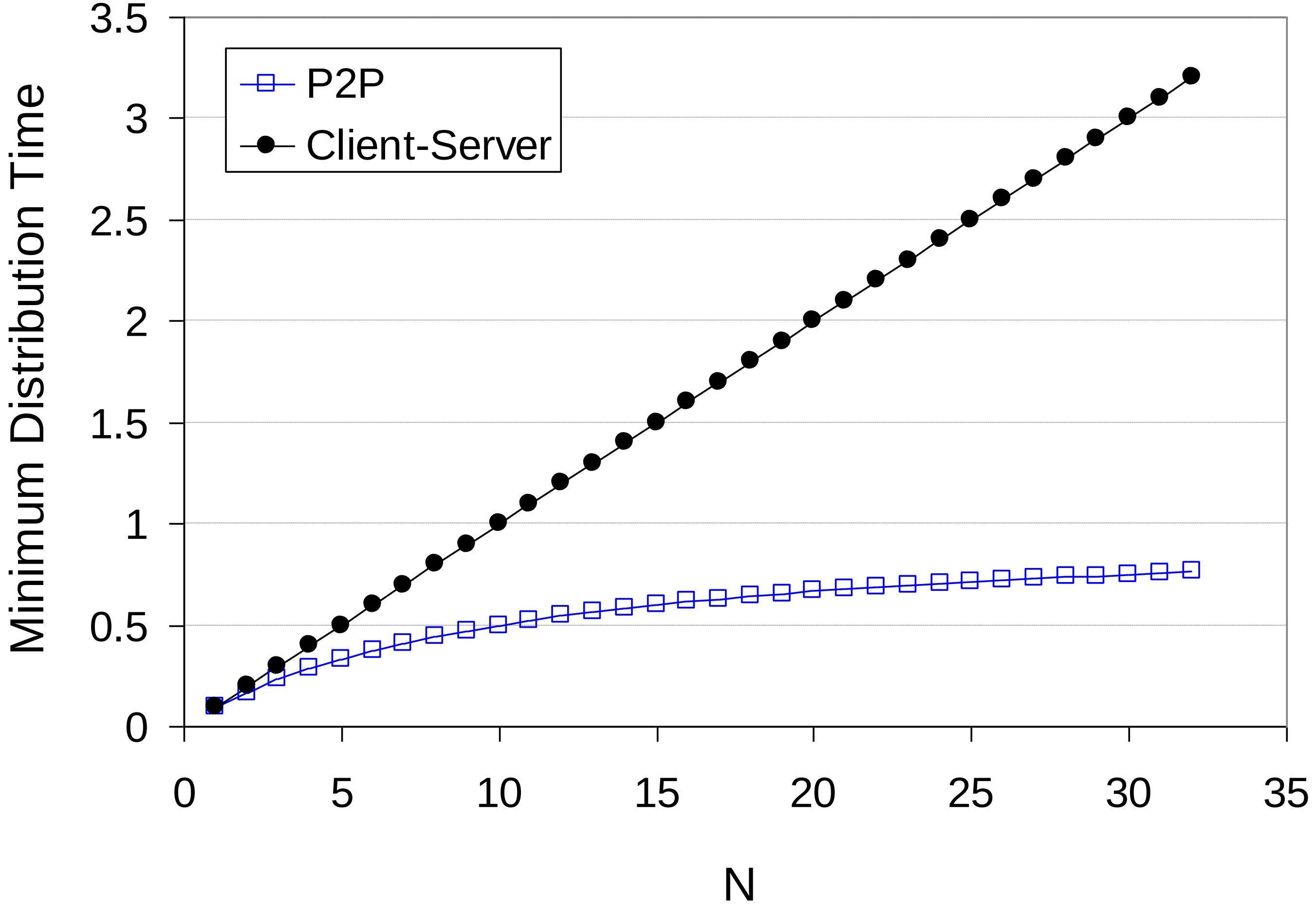
$$D_{P2P} \geq max\{F/u_s\ ,\ F/d_{min}\ ,\ NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

… but so does this, as each peer brings service capacity

Client-server vs. P2P: example

client upload rate = u, F/u = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$

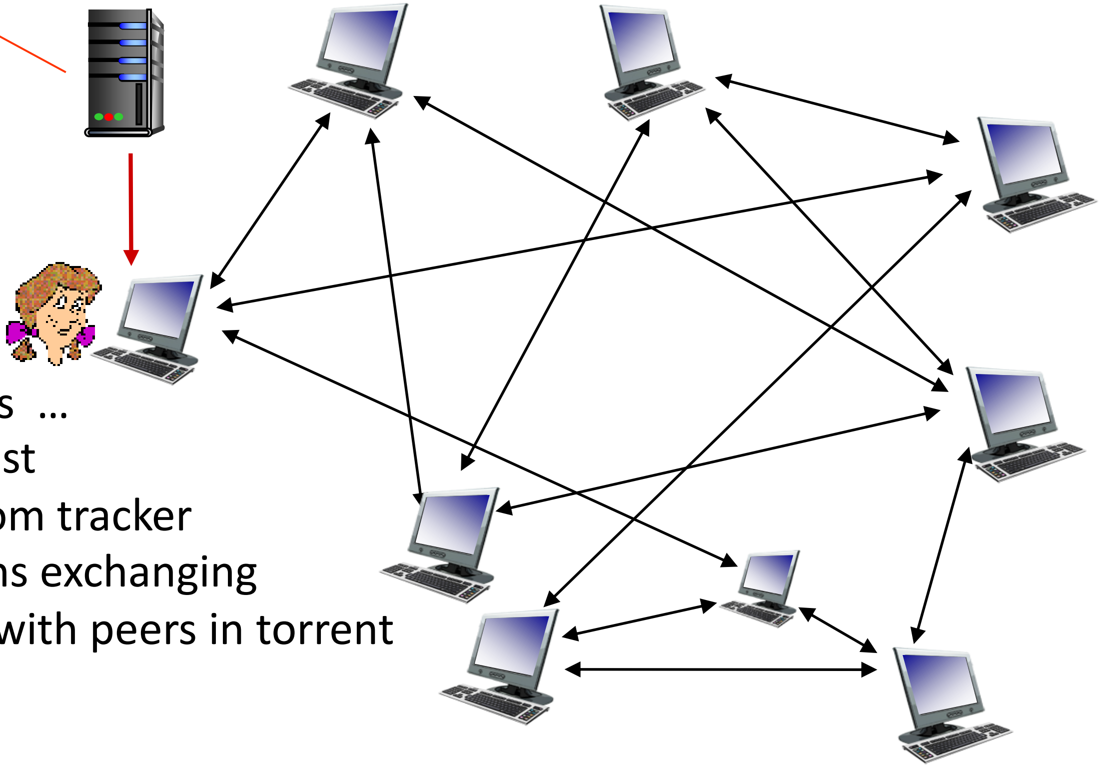


P2P file distribution: BitTorrent

- file divided into 256 KBytes chunks
- peers in torrent send/receive file chunks

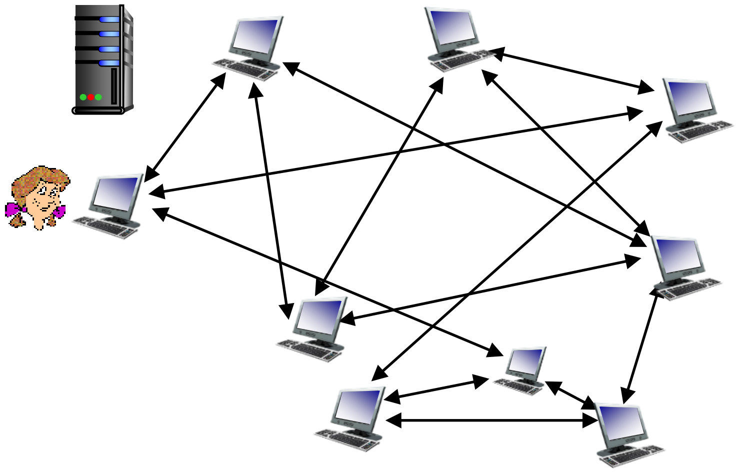
tracker: tracks peers participating in torrent

torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

P2P file distribution: BitTorrent



- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers (say 50), connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

Requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, **rarest first** technique

Sending chunks: tit-for-tat

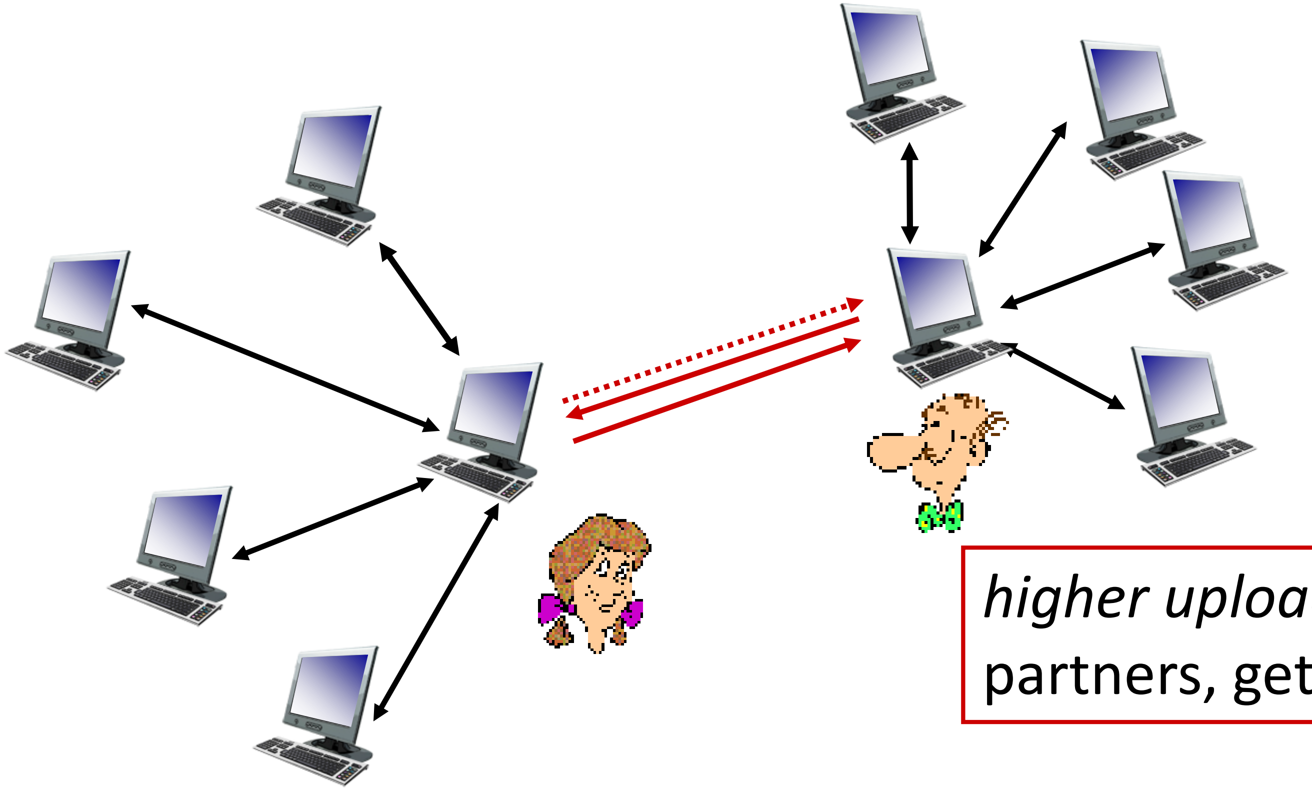
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - “optimistically unchoke” this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers



higher upload rate: find better trading partners, get file faster !

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution: distributed, application-level infrastructure*

Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)

frame *i*

temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

frame *i+1*

Multimedia: video

- CBR (constant bit rate): video encoding rate fixed
- VBR (variable bit rate): video encoding rate changes as amount of spatial, temporal coding changes
- examples:
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, 64Kbps – 12 Mbps)

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)

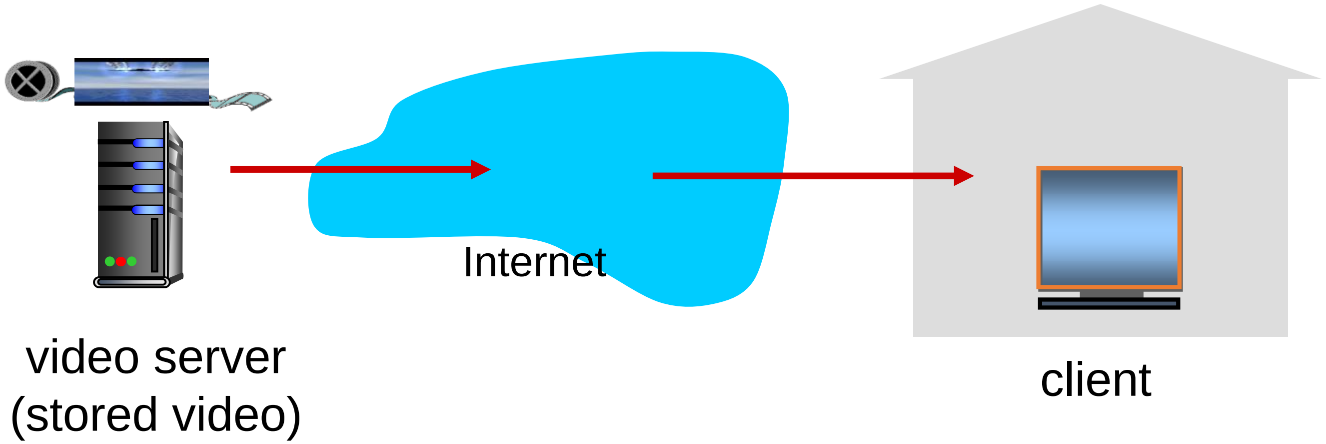
frame *i*

temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

frame *i+1*

Streaming stored video

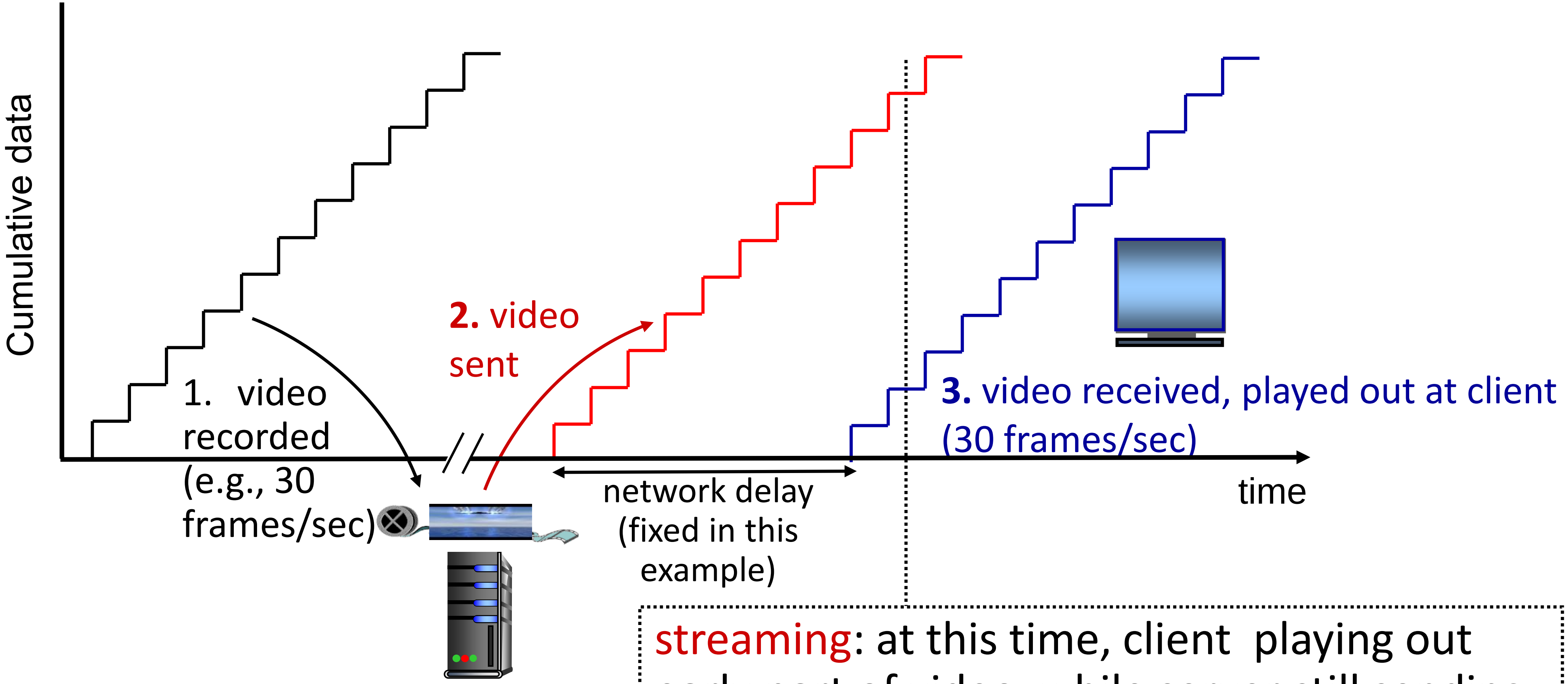
simple scenario:



Main challenges:

- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, in access network, in network core, at video server)
- packet loss and delay due to congestion will delay playout, or result in poor video quality

Streaming stored video



Cumulative data

1. video recorded (e.g., 30 frames/sec)

2. video sent

3. video received, played out at client (30 frames/sec)

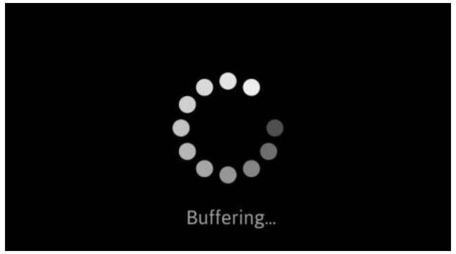
network delay (fixed in this example)

time

streaming: at this time, client playing out early part of video, while server still sending later part of video

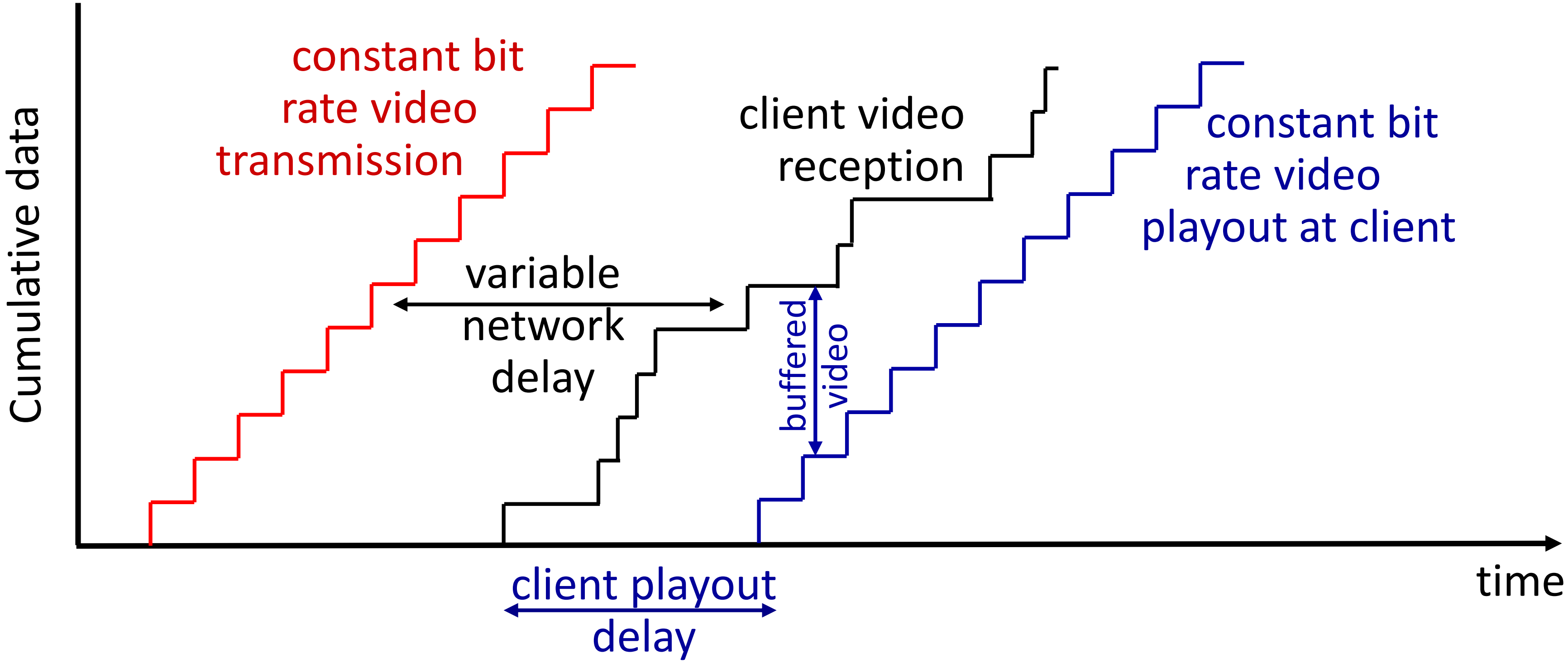
Streaming stored video: challenges

- continuous playout constraint: once client playout begins, playback must match original timing
  - … but network delays are variable (jitter), so will need client-side buffer to match playout requirements



- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted

Streaming stored video: playout buffering



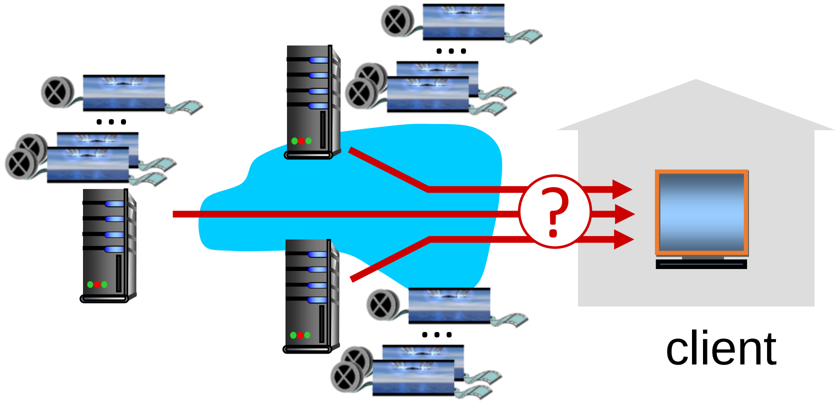
- *client-side buffering and playout delay:* compensate for network-added delay, delay jitter

Streaming multimedia: DASH

Dynamic, Adaptive
Streaming over HTTP

server:

- divides video file into multiple chunks
- each chunk encoded at multiple different rates
- different rate encodings stored in different files
- files replicated in various CDN nodes
- *manifest file:* provides URLs for different chunks

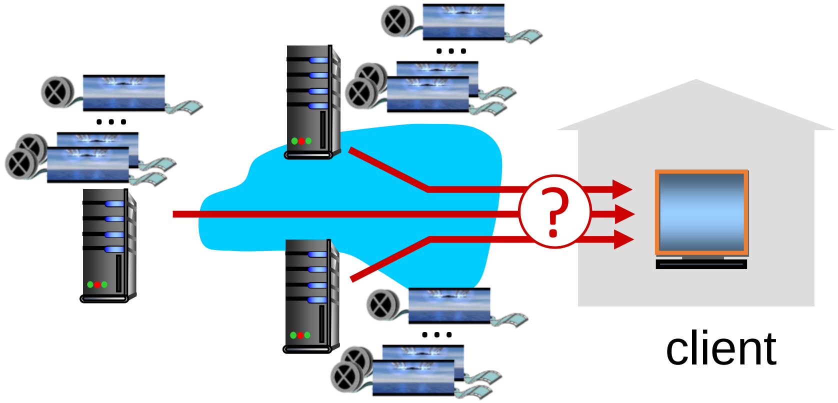


client:

- periodically estimates server-to-client bandwidth
- consulting manifest, requests one chunk at a time
  - chooses maximum coding rate sustainable given current bandwidth
  - can choose different coding rates at different points in time (depending on available bandwidth at time), and from different servers

Streaming multimedia: DASH

- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or has high available bandwidth)



Streaming video = encoding + DASH + playout buffering

Content distribution networks (CDNs)

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- option 1: single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long (and possibly congested) path to distant clients
  - multiple copies of video sent over outgoing link

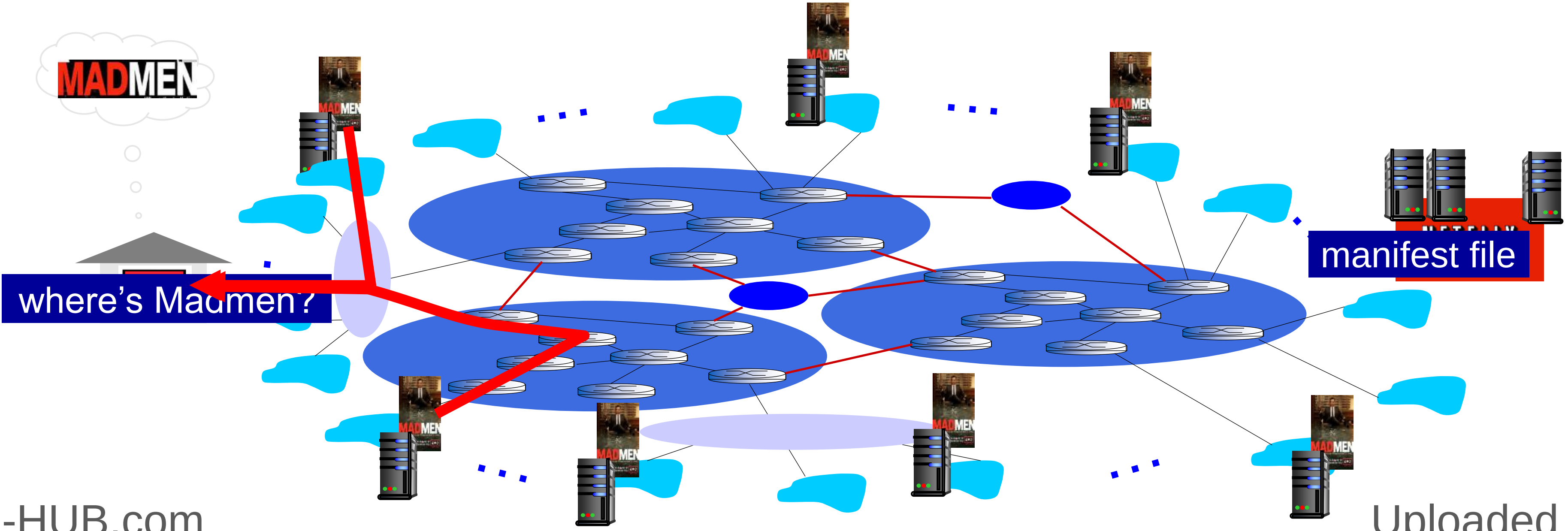
....quite simply: this solution *doesn't scale*

Content distribution networks (CDNs)

- *challenge:* how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?
- option 2: store/serve multiple copies of videos at multiple geographically distributed sites *(CDN)*
  - *enter deep:* push CDN servers deep into many access networks of ISPs
    - close to users
    - Akamai: 240,000 servers deployed in more than 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in PoPs/IXPs near (but not within) access networks
    - used by Limelight

Content distribution networks (CDNs)

- CDN: stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN, service provider returns manifest
  - using manifest, client directed to nearby copy, retrieves content at highest supported rate
  - may choose different rate or copy if network path congested



Content distribution networks (CDNs)

Internet host-host communication as a service

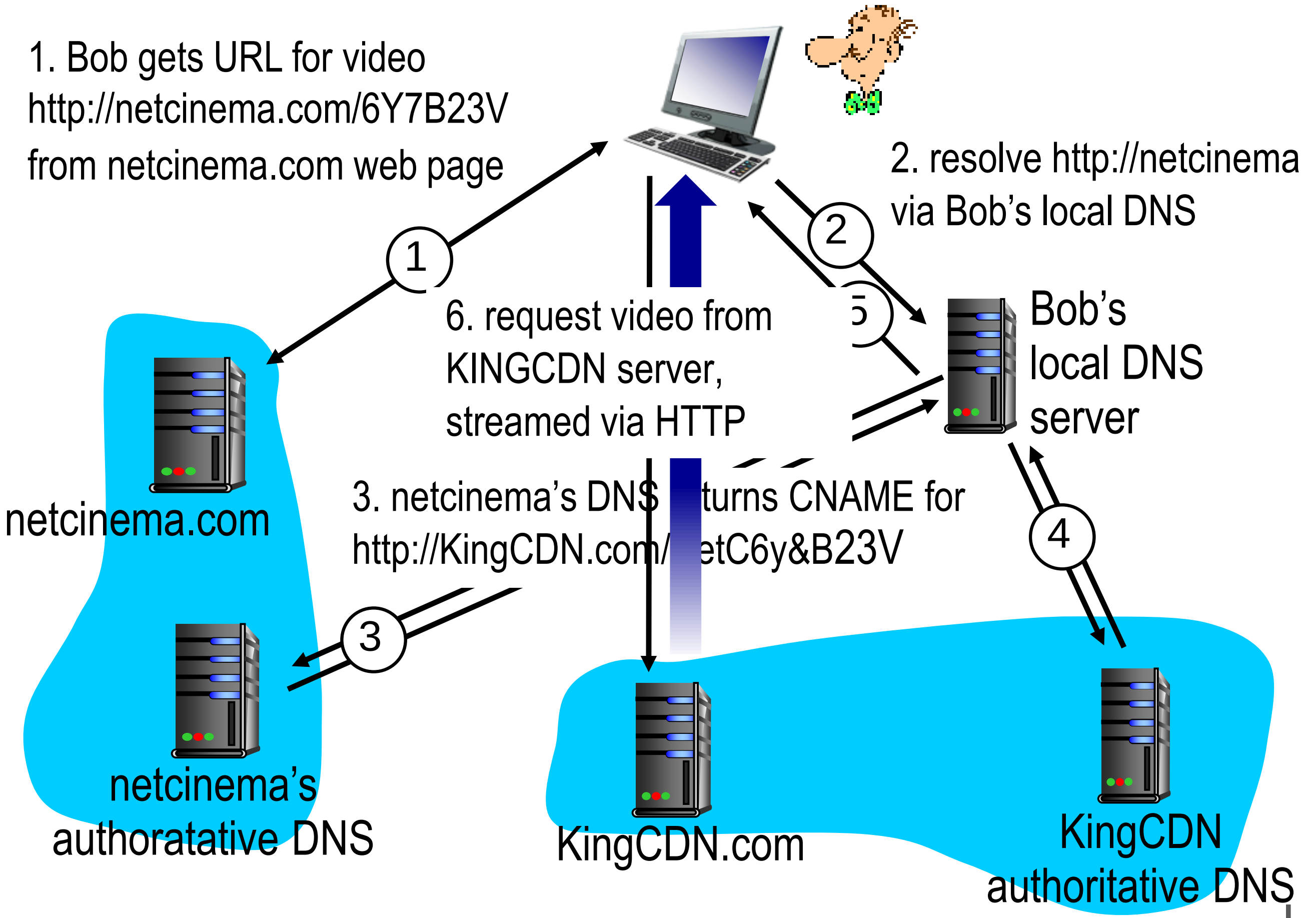
OTT challenges: coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V

- video stored in CDN at http://KingCDN.com/NetC6y&B23V



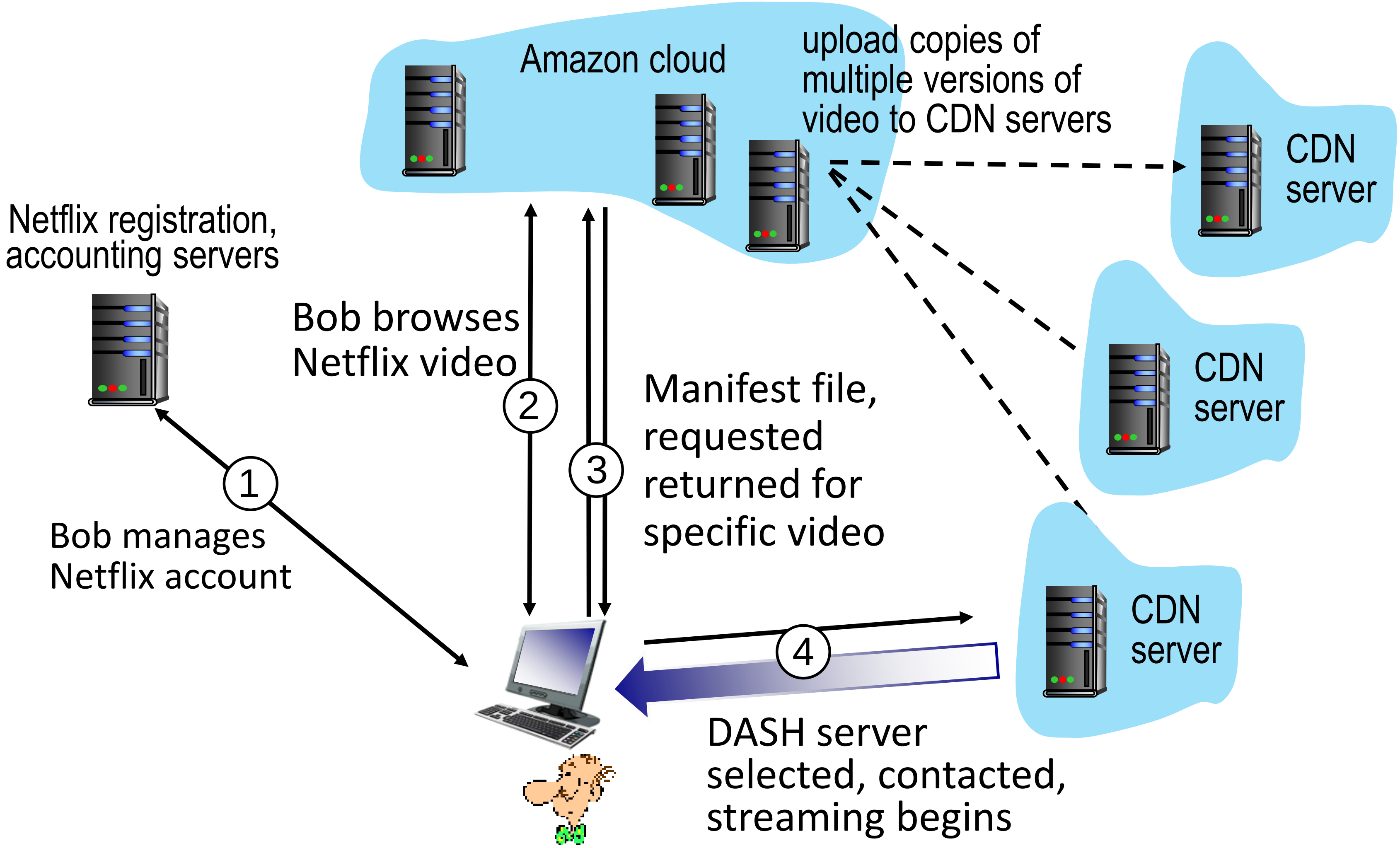
1. Bob gets URL for video http://netcinema.com/6Y7B23V from netcinema.com web page

2. resolve http://netcinema.com/6Y7B23V via Bob's local DNS

3. netcinema's DNS turns CNAME for http://KingCDN.com/ etC6y&B23V

6. request video from KINGCDN server, streamed via HTTP

Case study: Netflix



Amazon cloud

upload copies of multiple versions of video to CDN servers

CDN server

CDN server

CDN server

Netflix registration, accounting servers

① Bob manages Netflix account

② Bob browses Netflix video

③ Manifest file, requested returned for specific video

④ DASH server selected, contacted, streaming begins

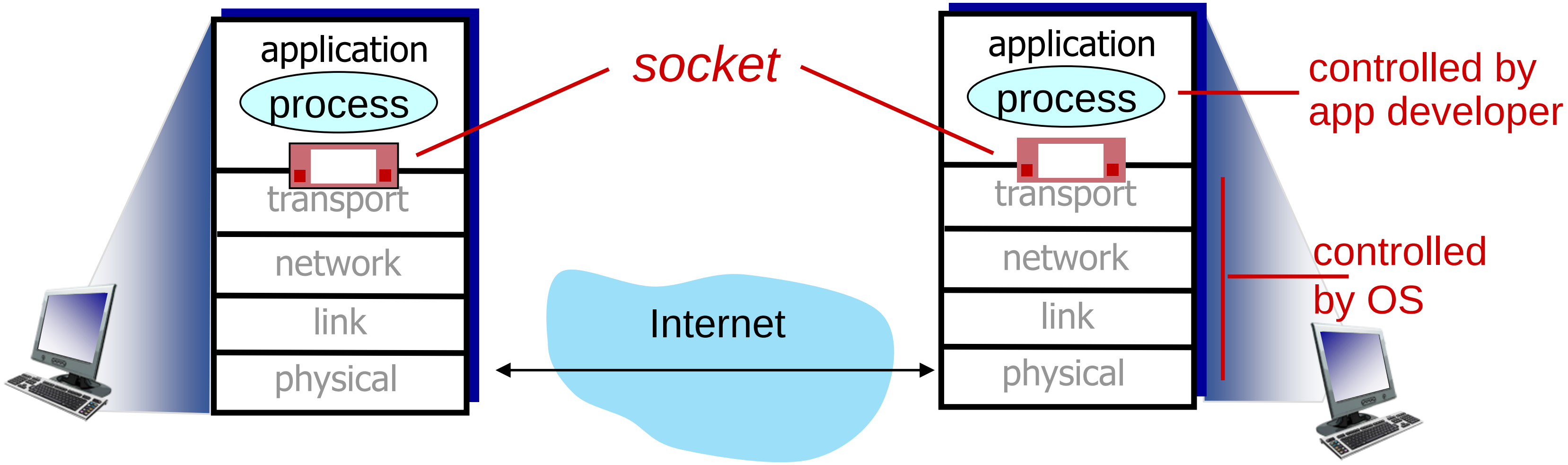
Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming with UDP

UDP: no "connection" between client & server

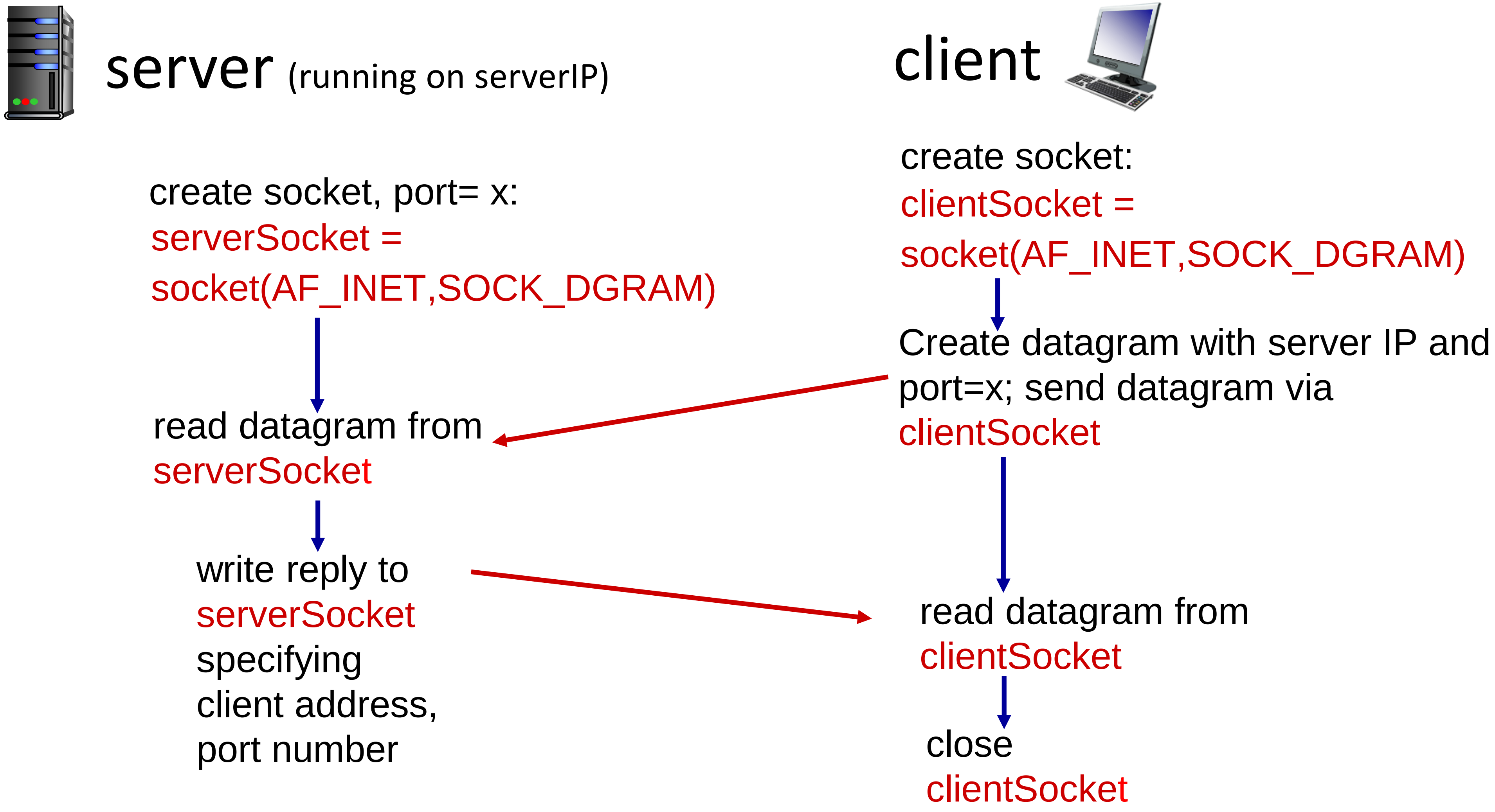
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: UDP client

Python UDPClient

include Python’s socket library

```
from socket import *
serverName = ‘hostname’
serverPort = 12000
```

create UDP socket for server

```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```

get user keyboard input

```
message = input(’Input lowercase sentence:’)
```

attach server name, port to message; send into socket

```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```

read reply characters from socket into string

```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```

print out received string and close socket

```
print (modifiedMessage.decode())
clientSocket.close()
```

Example app: UDP server

Python UDPServer

create UDP socket →

bind socket to local port number 12000 →

loop forever →

Read from UDP socket into message, getting client's address (client IP and port) →

send upper case string back to this client →

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                                    clientAddress)
```

Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

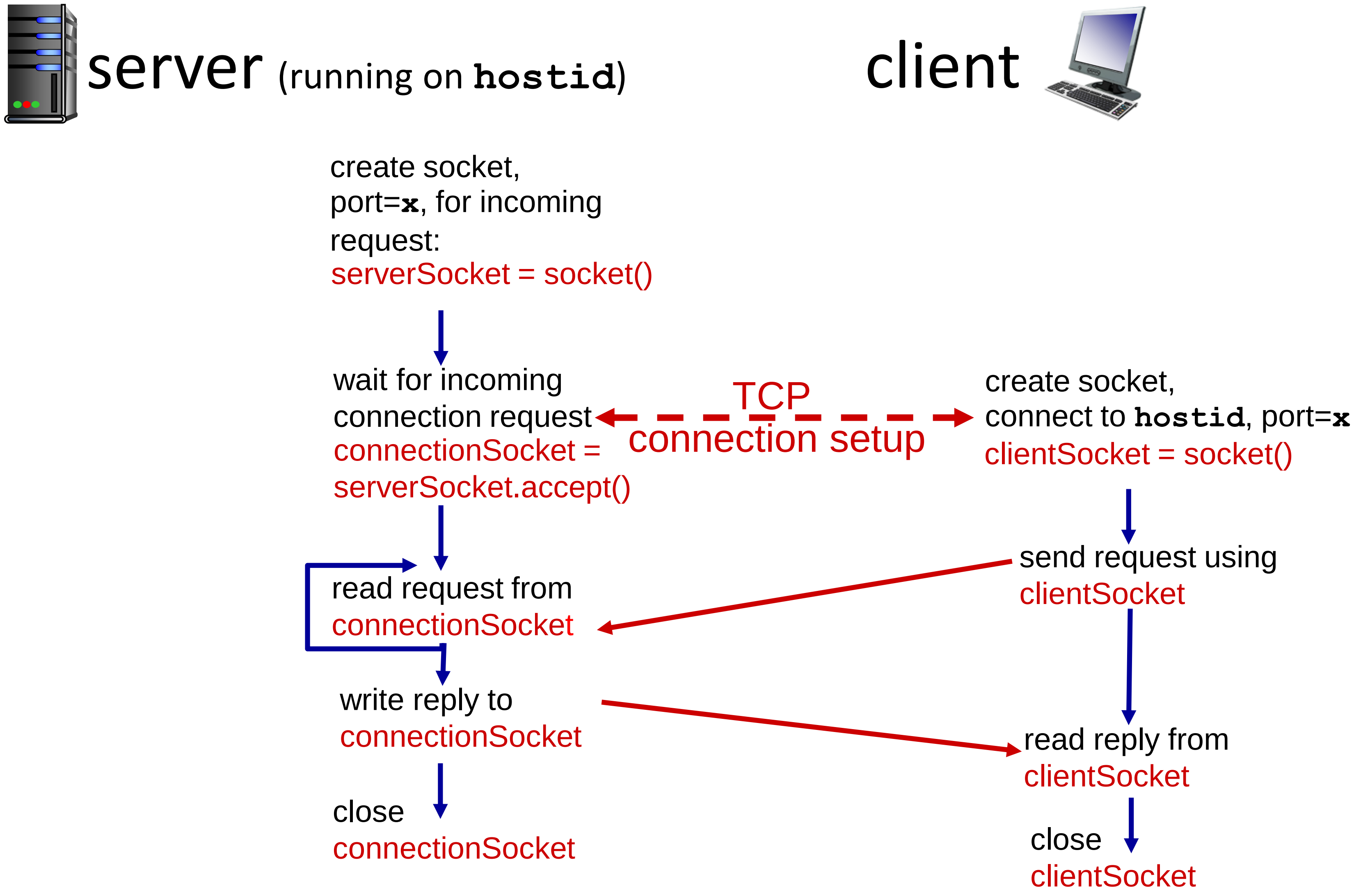
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

Application viewpoint

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



server (running on **`hostid`**)

client

create socket,
port=**`x`**, for incoming
request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **`hostid`**, port=**`x`**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

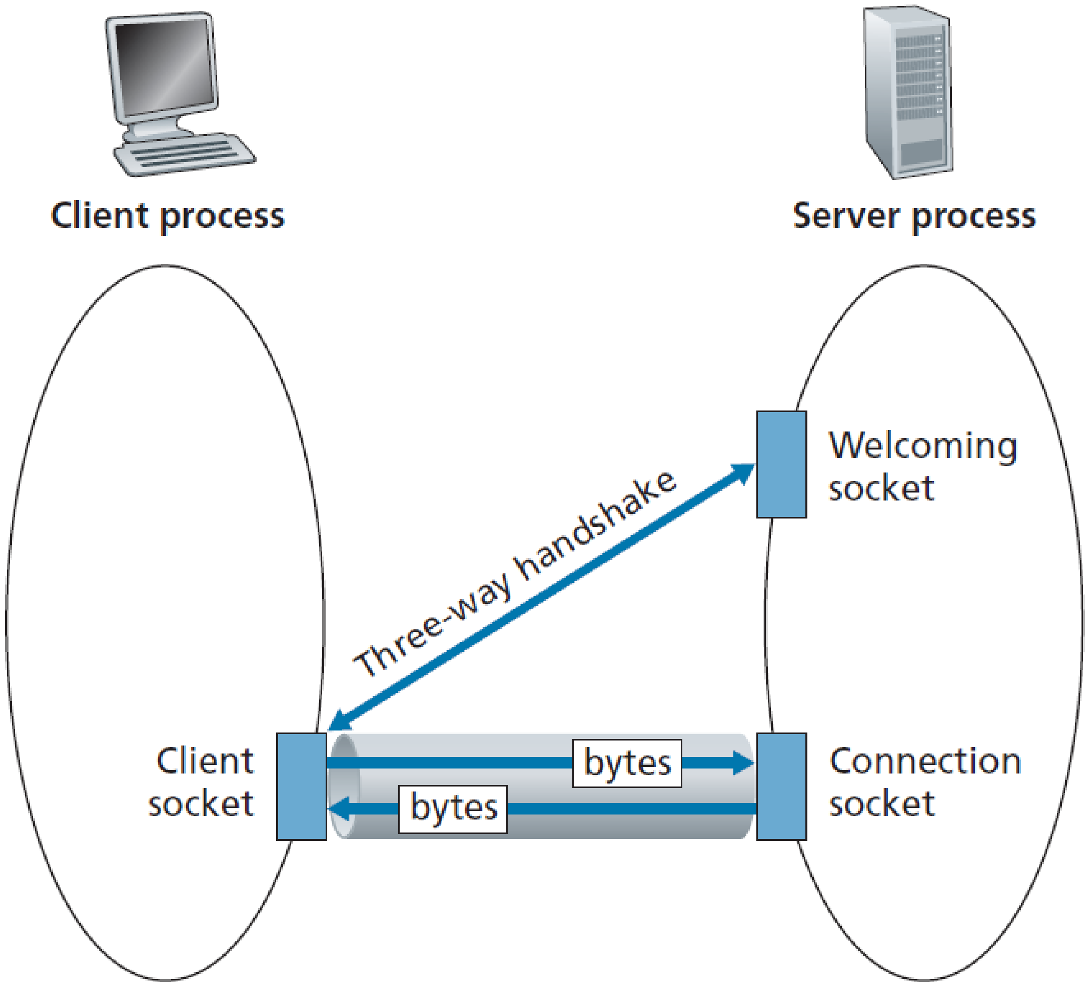
write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

The TCPServer process has two sockets



Example app: TCP client

Python TCPClient

create TCP socket for server, remote port 12000 →

No need to attach server name, port →

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input(‘Input lowercase sentence:’)
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print (‘From Server:’, modifiedSentence.decode())
clientSocket.close()
```

Example app: TCP server

Python TCPServer

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print (‘The server is ready to receive’)
while True:
     connectionSocket, addr = serverSocket.accept()

     sentence = connectionSocket.recv(1024)
     capitalizedSentence = sentence.decode().upper()
     connectionSocket.send(capitalizedSentence.encode())
     connectionSocket.close()
```

Chapter 2: Summary

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS
  - P2P: BitTorrent
- video streaming, CDNs
- socket programming:
  TCP, UDP sockets

Chapter 2: Summary

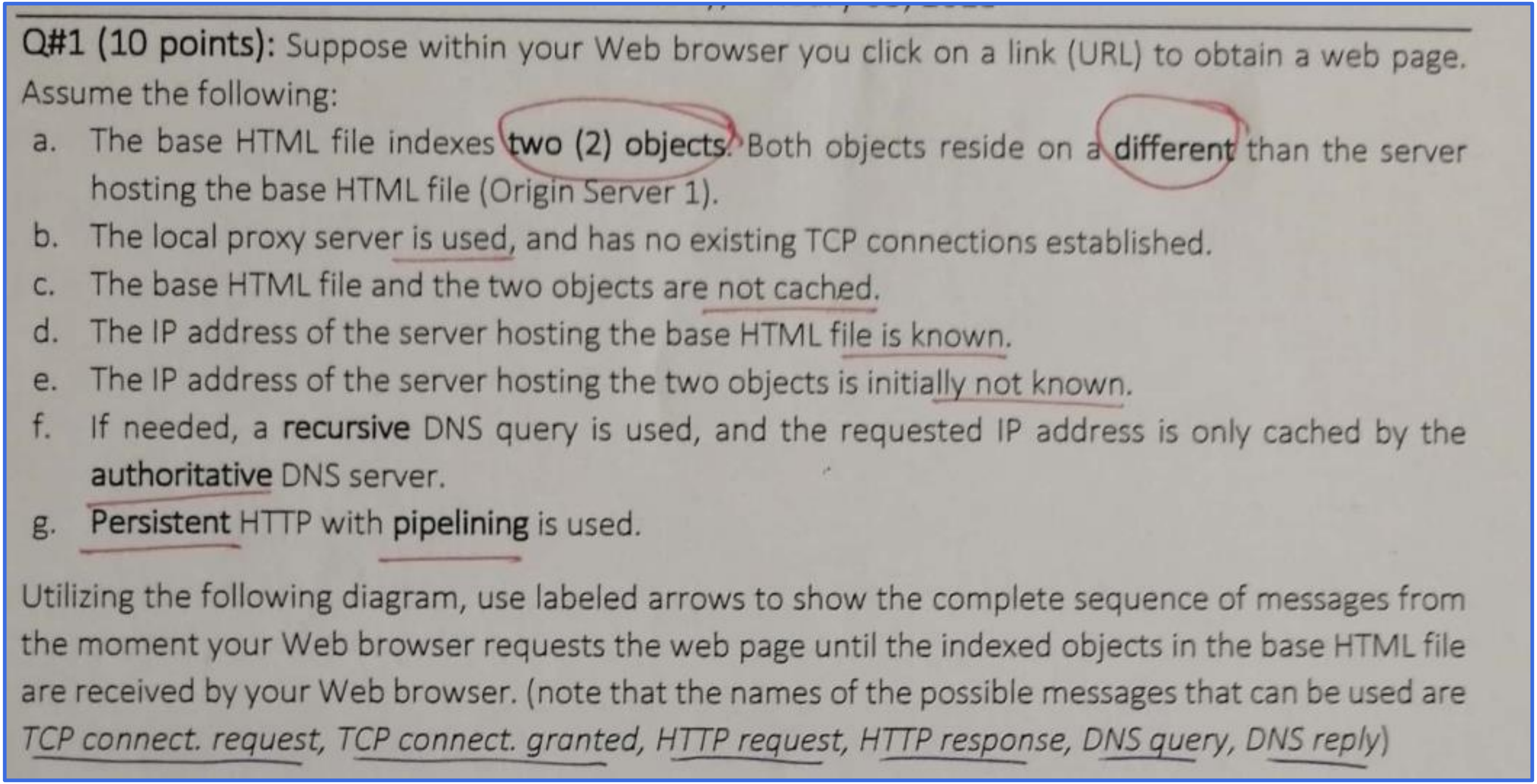
Most importantly: learned about *protocols*!

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - *headers*: fields giving info about data
  - *data:* info(payload) being communicated

important themes:

- centralized vs. decentralized
- stateless vs. stateful
- scalability
- reliable vs. unreliable message transfer
- "complexity at network edge"

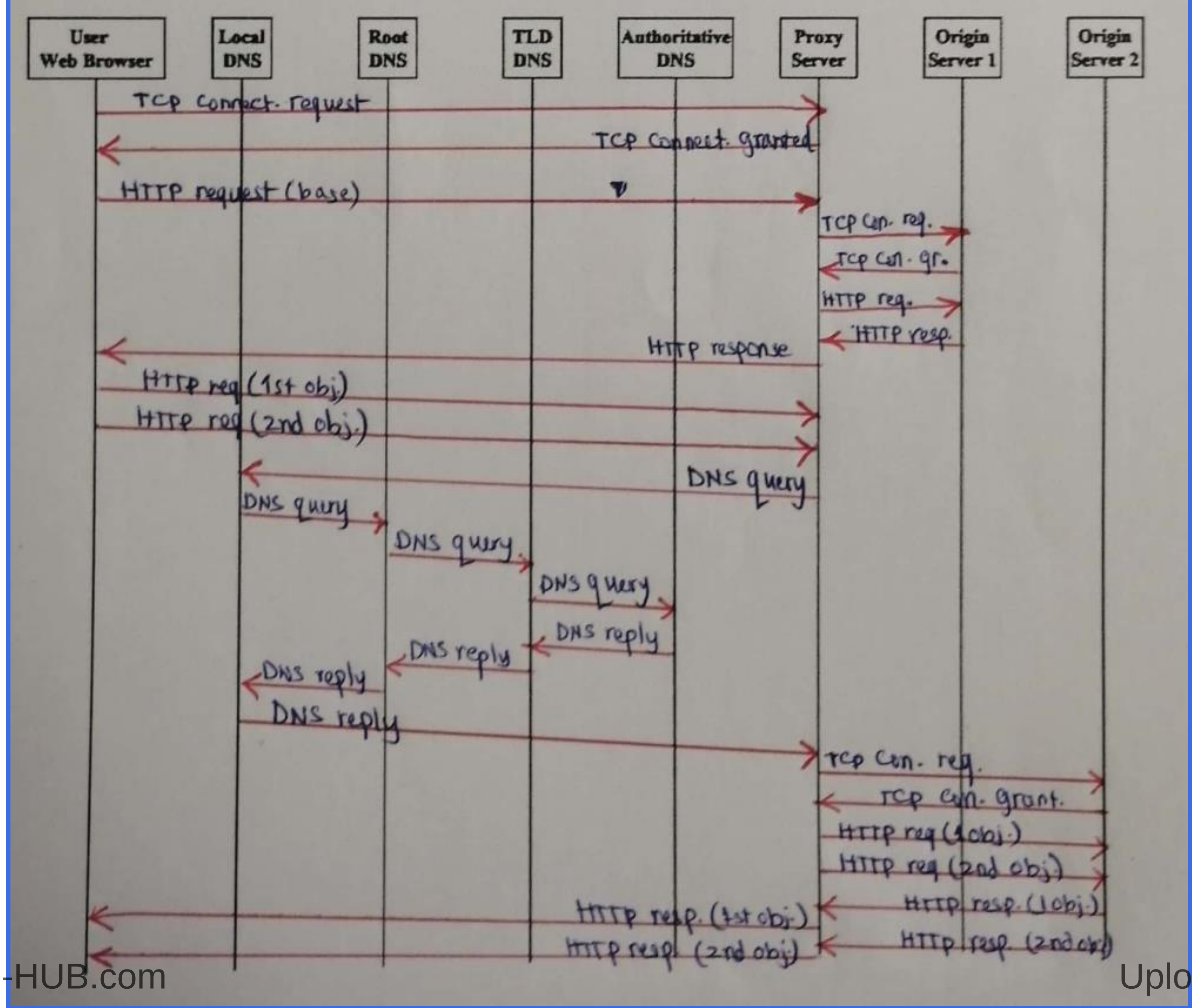
Additional Chapter 2 slides



Q#1 (10 points): Suppose within your Web browser you click on a link (URL) to obtain a web page. Assume the following:

a. The base HTML file indexes **two (2) objects**. Both objects reside on a **different** than the server hosting the base HTML file (Origin Server 1).
b. The local proxy server is used, and has no existing TCP connections established.
c. The base HTML file and the two objects are not cached.
d. The IP address of the server hosting the base HTML file is known.
e. The IP address of the server hosting the two objects is initially not known.
f. If needed, a **recursive** DNS query is used, and the requested IP address is only cached by the **authoritative** DNS server.
g. **Persistent** HTTP with **pipelining** is used.

Utilizing the following diagram, use labeled arrows to show the complete sequence of messages from the moment your Web browser requests the web page until the indexed objects in the base HTML file are received by your Web browser. (note that the names of the possible messages that can be used are *TCP connect. request, TCP connect. granted, HTTP request, HTTP response, DNS query, DNS reply*)


User Web Browser
Local DNS
Root DNS
TLD DNS
Authoritative DNS
Proxy Server
Origin Server 1
Origin Server 2
TCP connect. request
TCP connect. granted
HTTP request (base)
TCP Con. req.
TCP Con. gr.
HTTP req.
HTTP resp.
HTTP response
HTTP req (1st obj.)
HTTP req (2nd obj.)
DNS query
DNS query
DNS query
DNS query
DNS reply
DNS reply
DNS reply
DNS reply
TCP Con. req.
TCP Con. grant.
HTTP req (1 obj.)
HTTP req (2nd obj.)
HTTP resp. (1 obj.)
HTTP resp. (2nd obj.)
HTTP resp. (1st obj.)
HTTP resp. (2nd obj.)