

# Chapter 2:

# Application Layer – Part 1

Instructor: Zhuozhao Li

Lab: Qing Wang

Department of Computer Science and Engineering

# Quick review

- Access networks
- Performance
  - Transmission delay
  - Queuing delay
  - Processing delay
  - Propagation delay
  - Bandwidth vs. throughput
- Packet-switching vs. circuit-switching
- Network security

# Quick review

- HW1 available on Sakai
- Deadline: 2 weeks, March 23, 2022

# 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 (Youku, Migu)
  - P2P file sharing
  - voice over IP (e.g., Skype)
  - real-time video conferencing
  - Internet search
  - remote login
  - ...
- Q: *your* favorites?

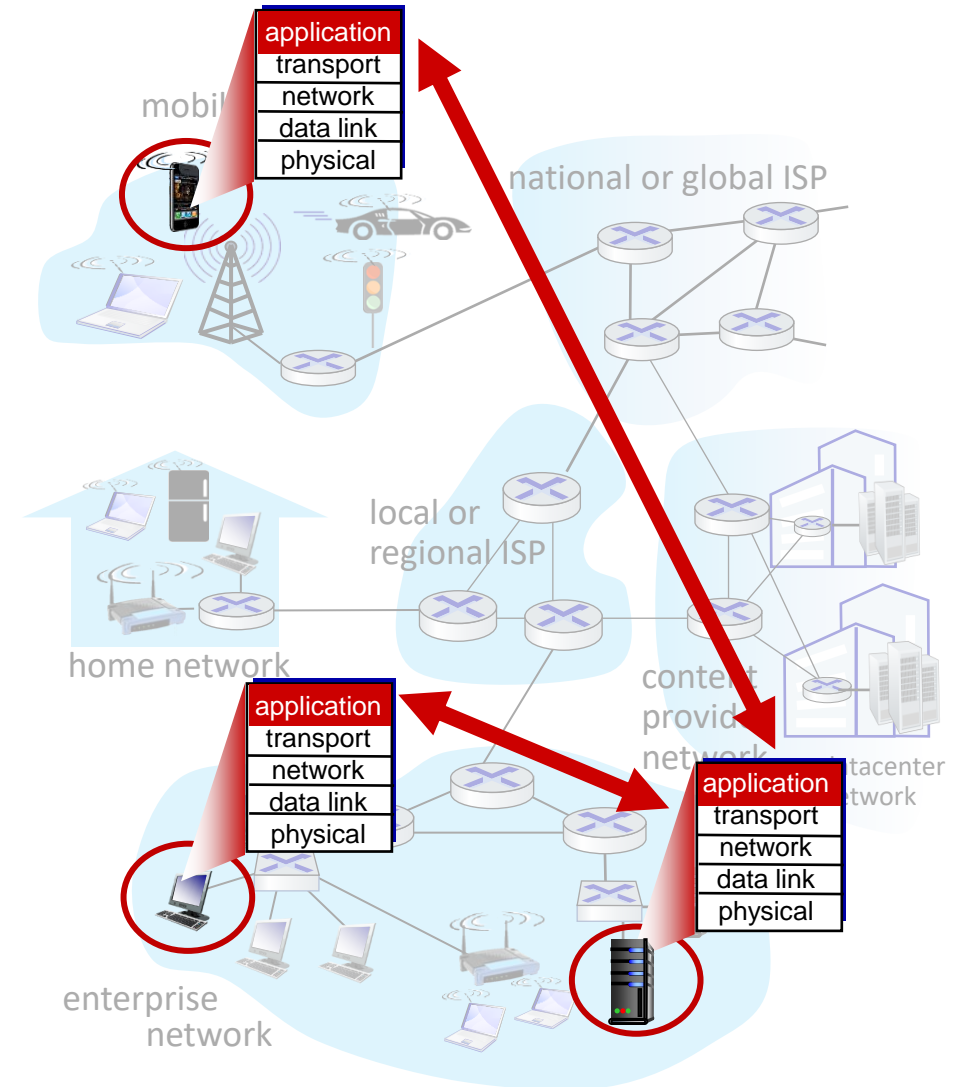
# Creating a network app

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



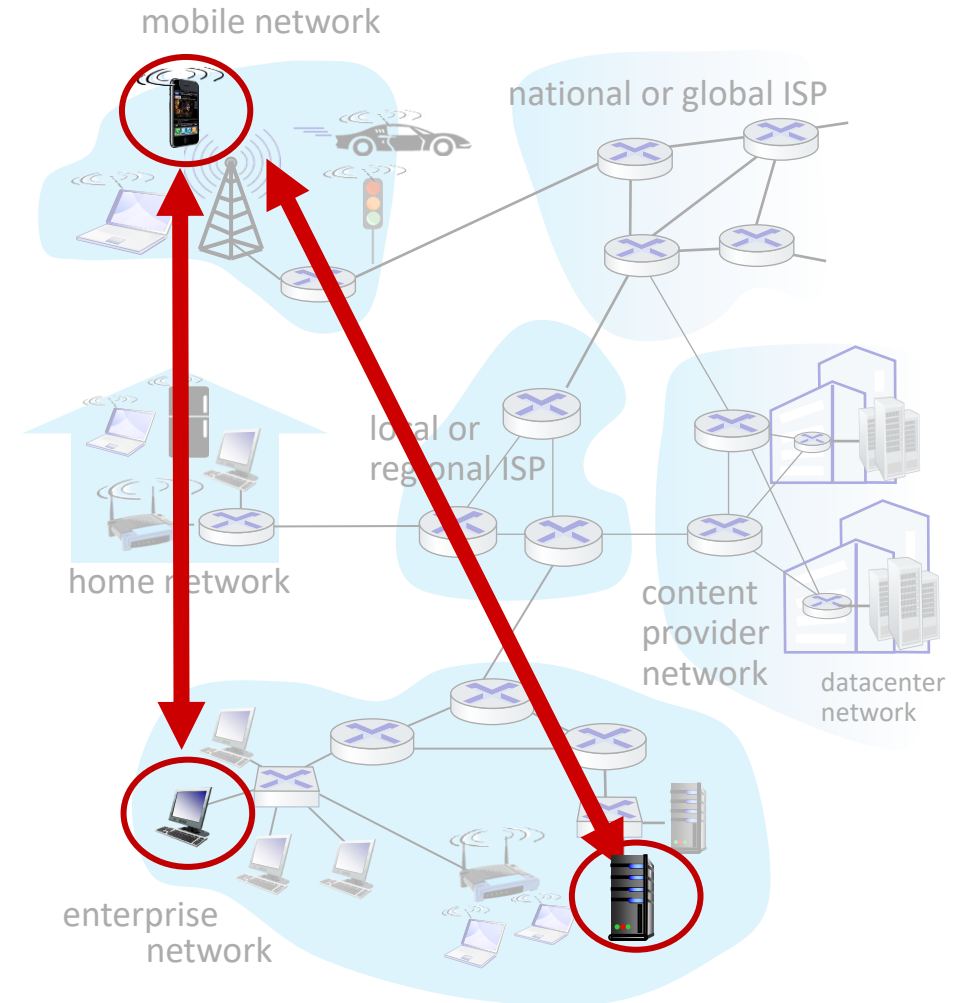
# Client-server paradigm

## server:

- always-on – long-running
- permanent IP address
- often in data centers, for scaling

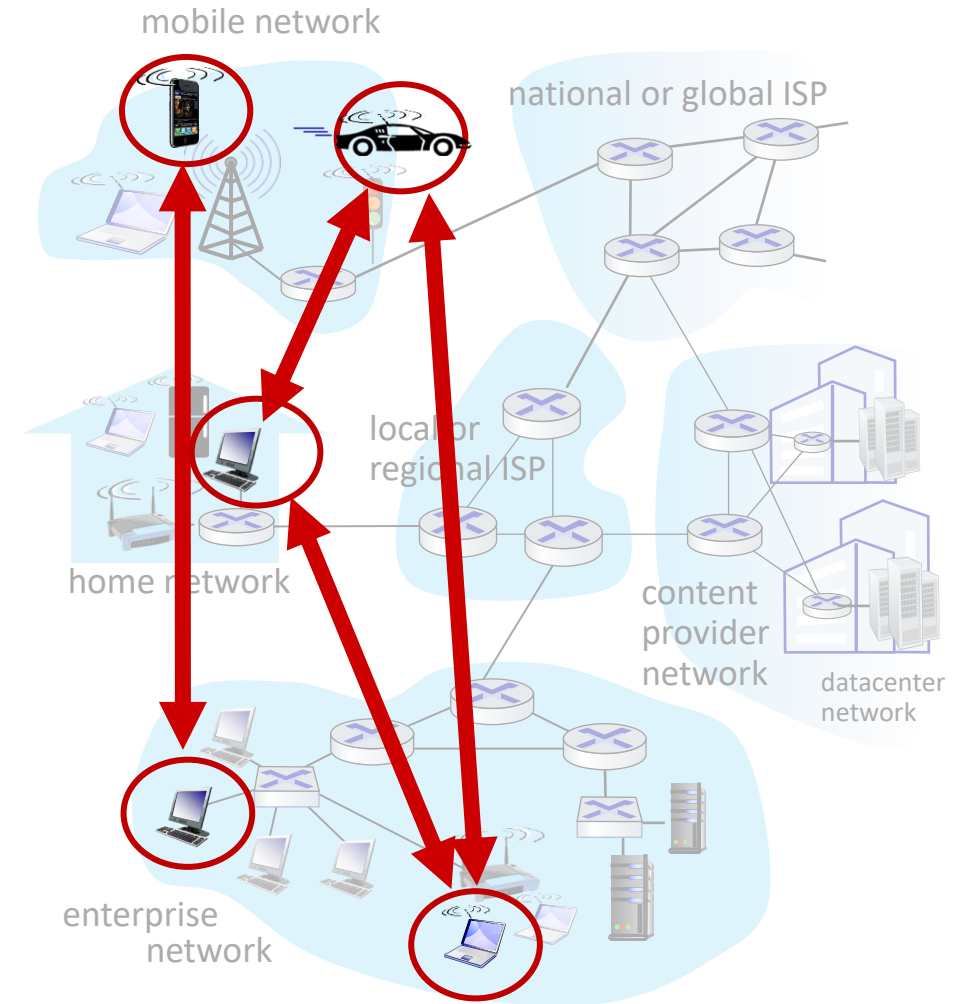
## clients:

- 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-to-peer (P2P) paradigm

- *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

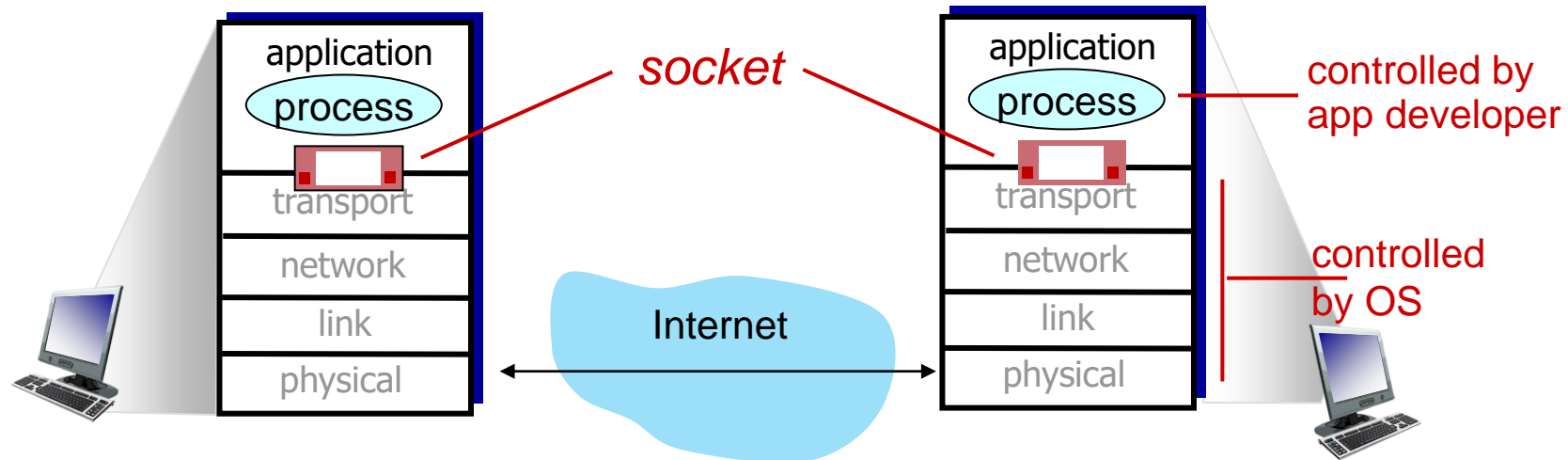
*client process*: process that initiates communication

*server process*: process that waits to be contacted

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

# Sockets (套接字)

- process sends/receives messages to/from its **socket**
- socket analogous to courier (快递员)
  - sending process shoves message out courier
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side



# Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host
- *identifier* includes both **IP address** and **port numbers** associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - **IP address:** 128.119.245.12
  - **port number:** 80
- more shortly...

# An application-layer protocol defines:

- **types of messages exchanged**,
  - e.g., request, response
- **message syntax**:
  - what fields in messages & how fields are delineated
- **message semantics**
  - meaning of information in fields
- **rules** for when and how processes send & respond to messages

## **open protocols:**

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

## **proprietary protocols:**

- e.g., Skype, Zoom

# 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
- *does not provide*: timing, minimum throughput guarantee, security
- *connection-oriented*: setup required between client and server processes

## *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.

# 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



# 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 (Uniform Resource Locator)*, e.g.,

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

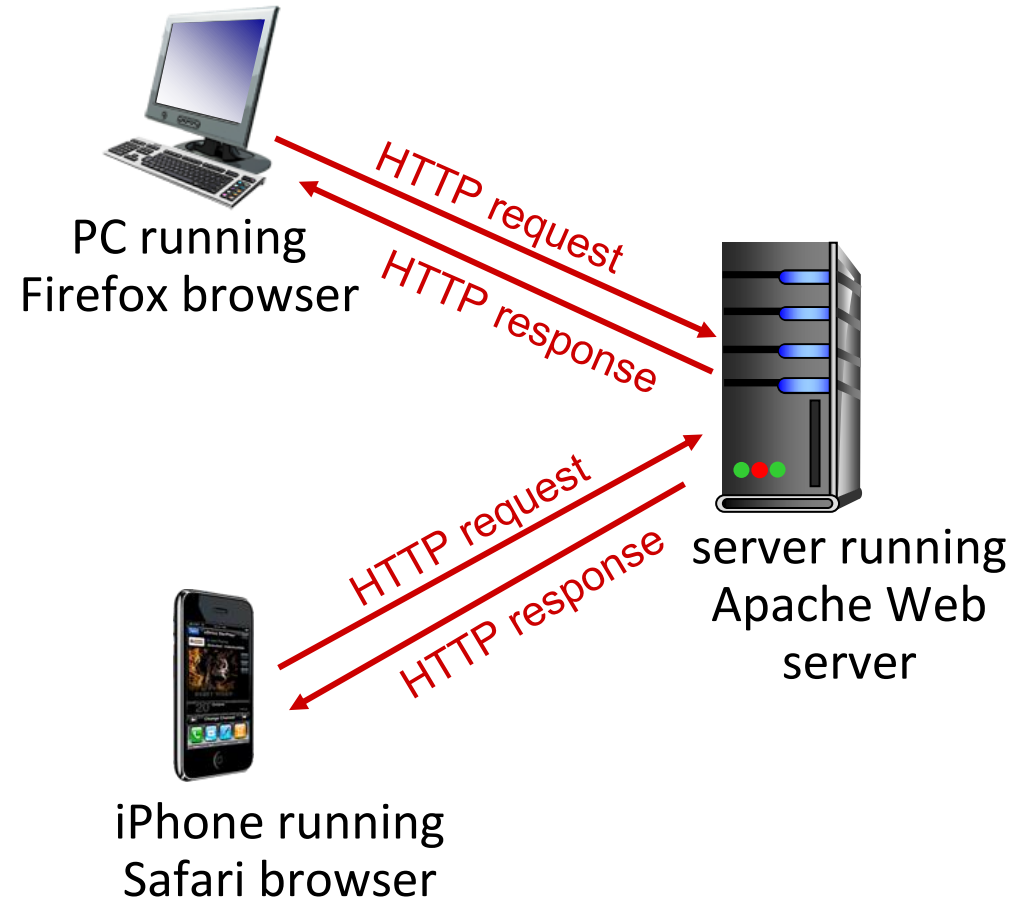
host name

path name

# HTTP overview

## HTTP: hypertext transfer protocol

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



# 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 (HTTP 1.0)*

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 (HTTP 1.1)*

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

# Non-persistent HTTP: example

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



**1a.** HTTP client initiates TCP connection to HTTP server (process) at `www.someSchool.edu` on port 80



**1b.** HTTP server at host `www.someSchool.edu` waiting for TCP connection at port 80 “accepts” connection, notifying client

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

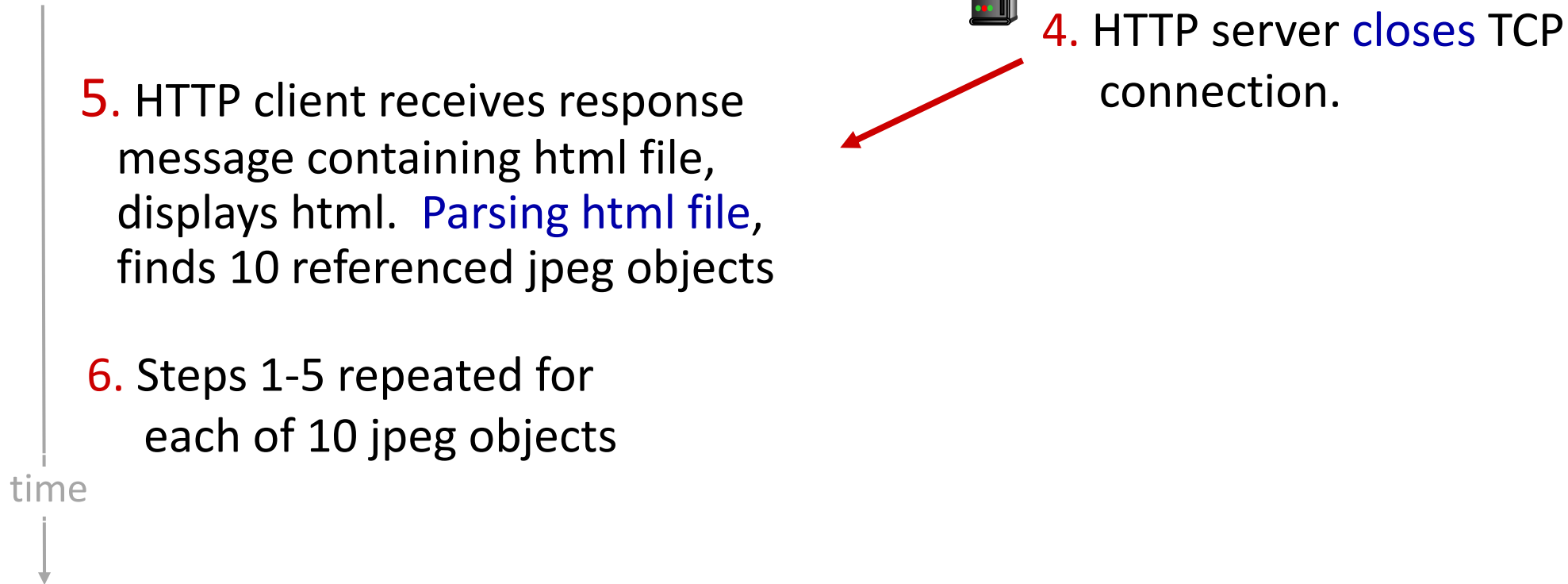
**3.** HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time



# Non-persistent HTTP: example (cont.)

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

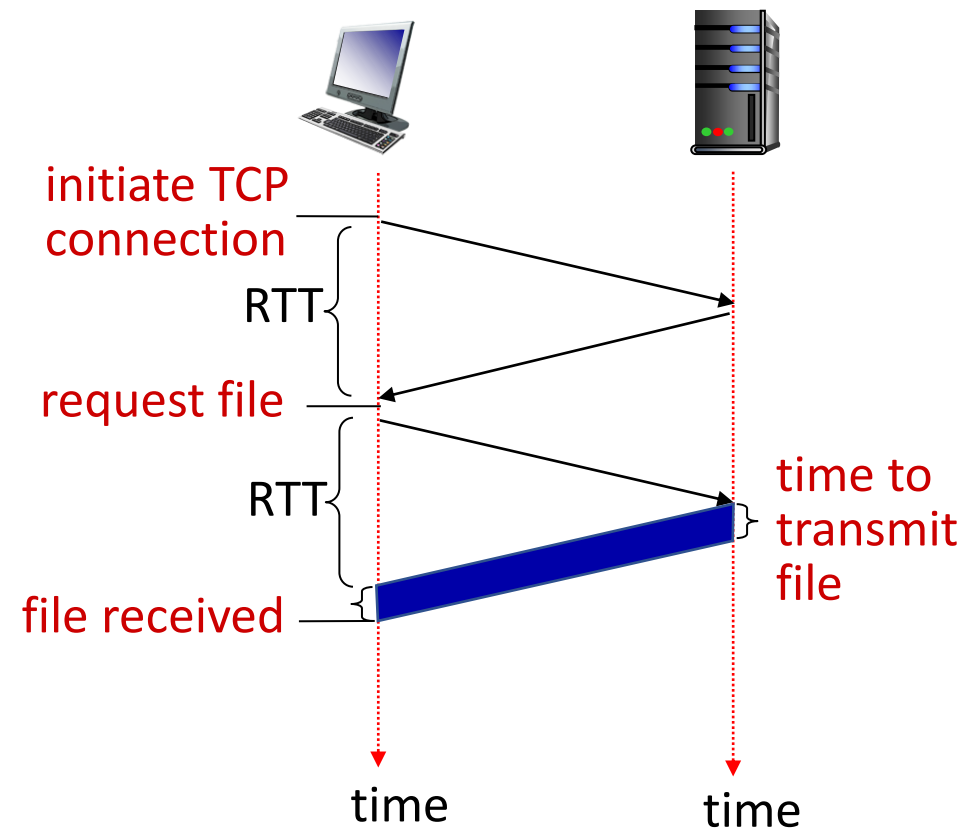


# Non-persistent HTTP: response time

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

**HTTP response time (per object):**

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



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



# Persistent HTTP (HTTP 1.1)

## *Non-persistent HTTP issues:*

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

## *Persistent HTTP (HTTP1.1):*

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)

# HTTP request message

Start line
Message headers
Blank line
Message body

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

request line (GET, POST,  
HEAD commands)

header  
lines

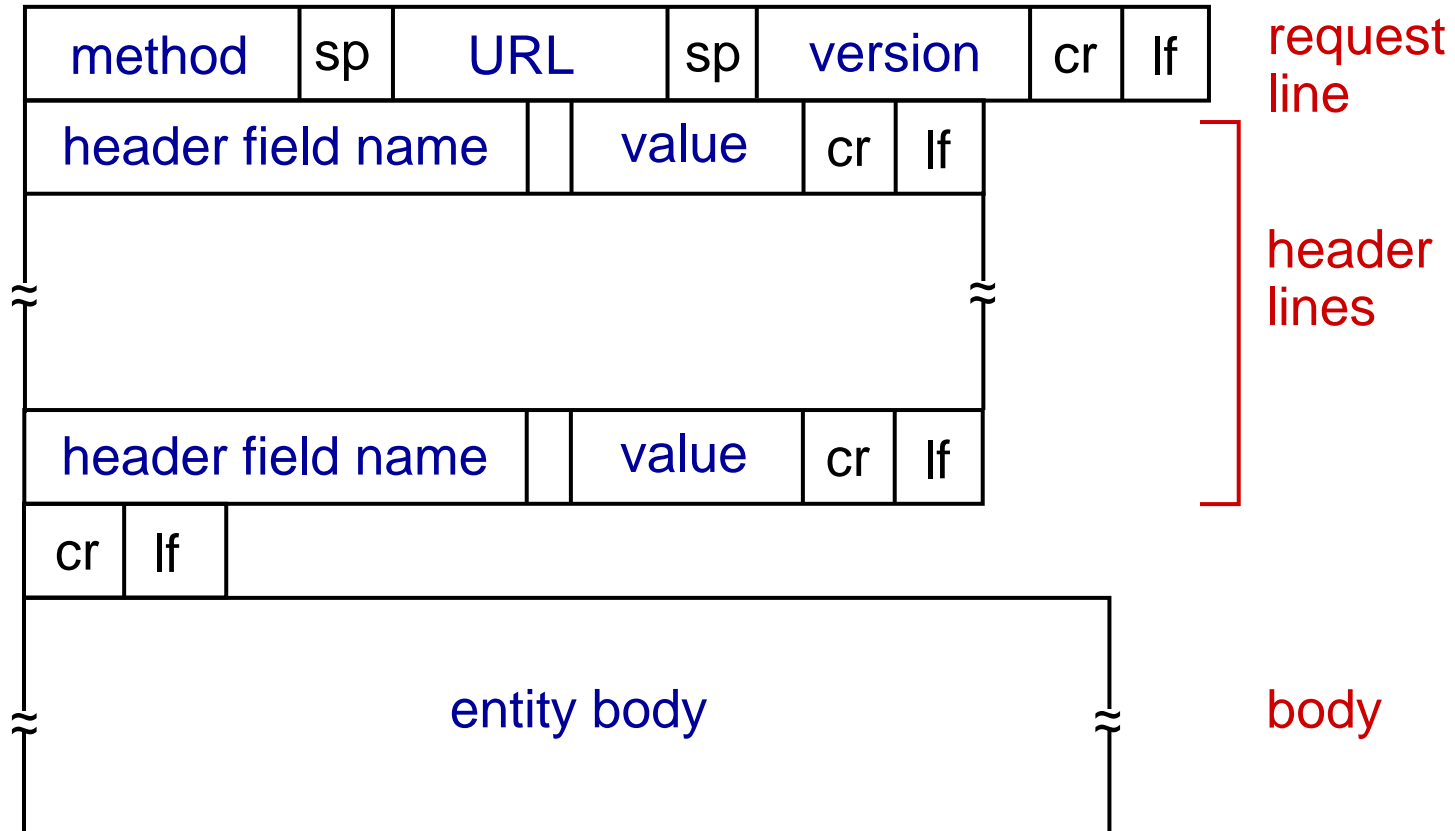
carriage return, line feed  
at start of line indicates  
end of header lines

```
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
```

carriage return character  
line-feed character

\* Check out the online interactive exercises for more  
examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# HTTP request message: general format



# 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`

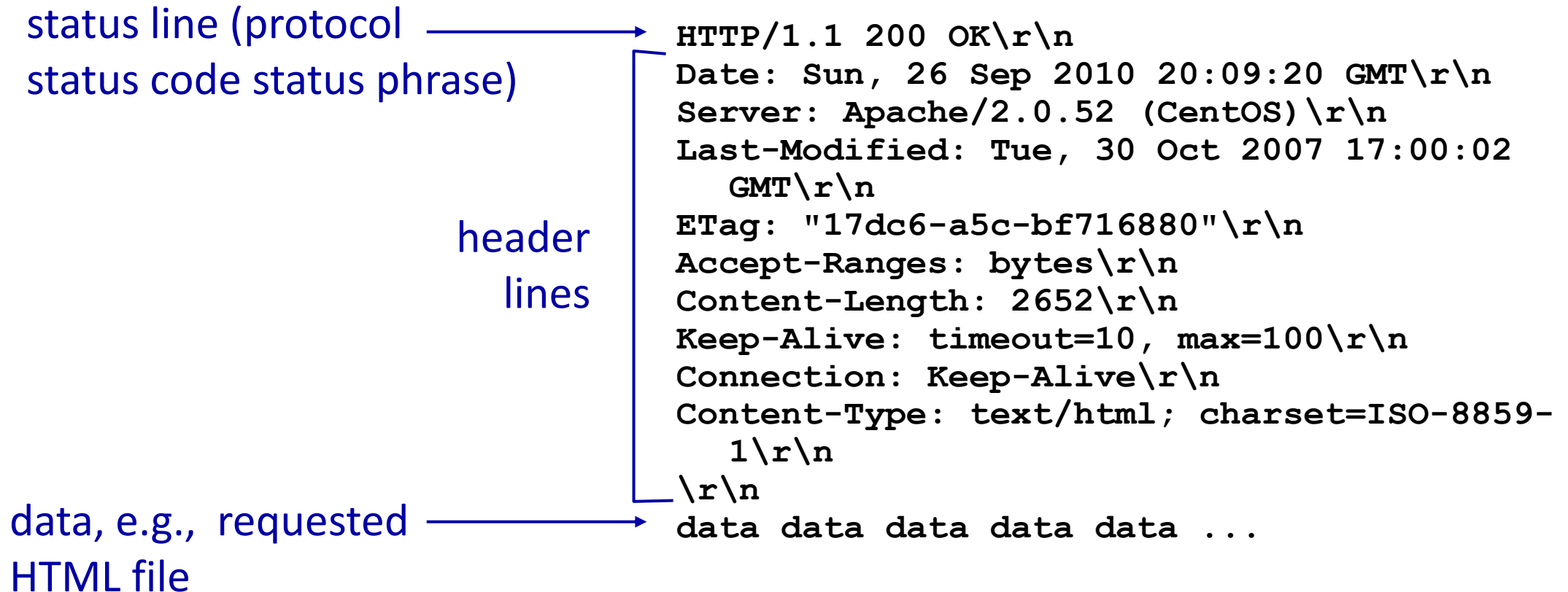
## HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

## PUT method (HTTP 1.1):

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

# HTTP response message



\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](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

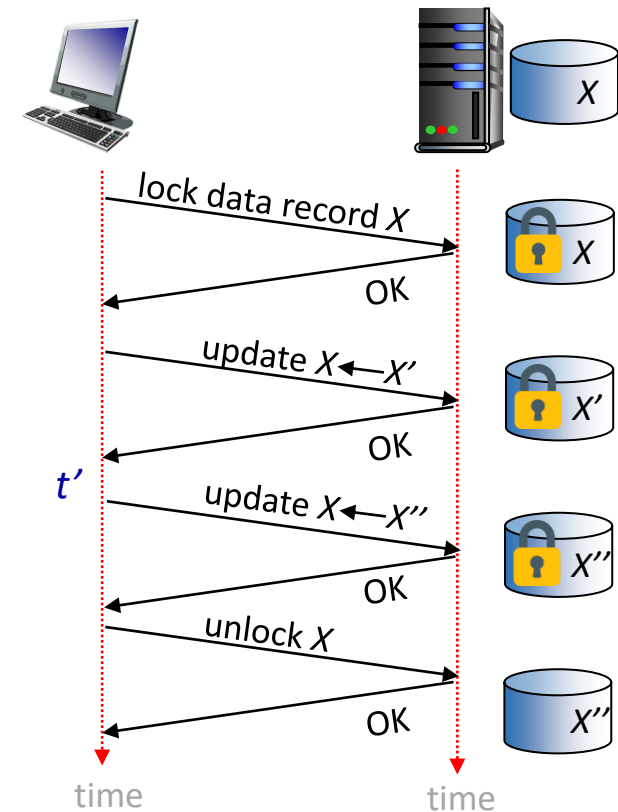
1. Informational responses ( 100 – 199 )
2. Successful responses ( 200 – 299 )
3. Redirection messages ( 300 – 399 )
4. Client error responses ( 400 – 499 )
5. Server error responses ( 500 – 599 )

# 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:*

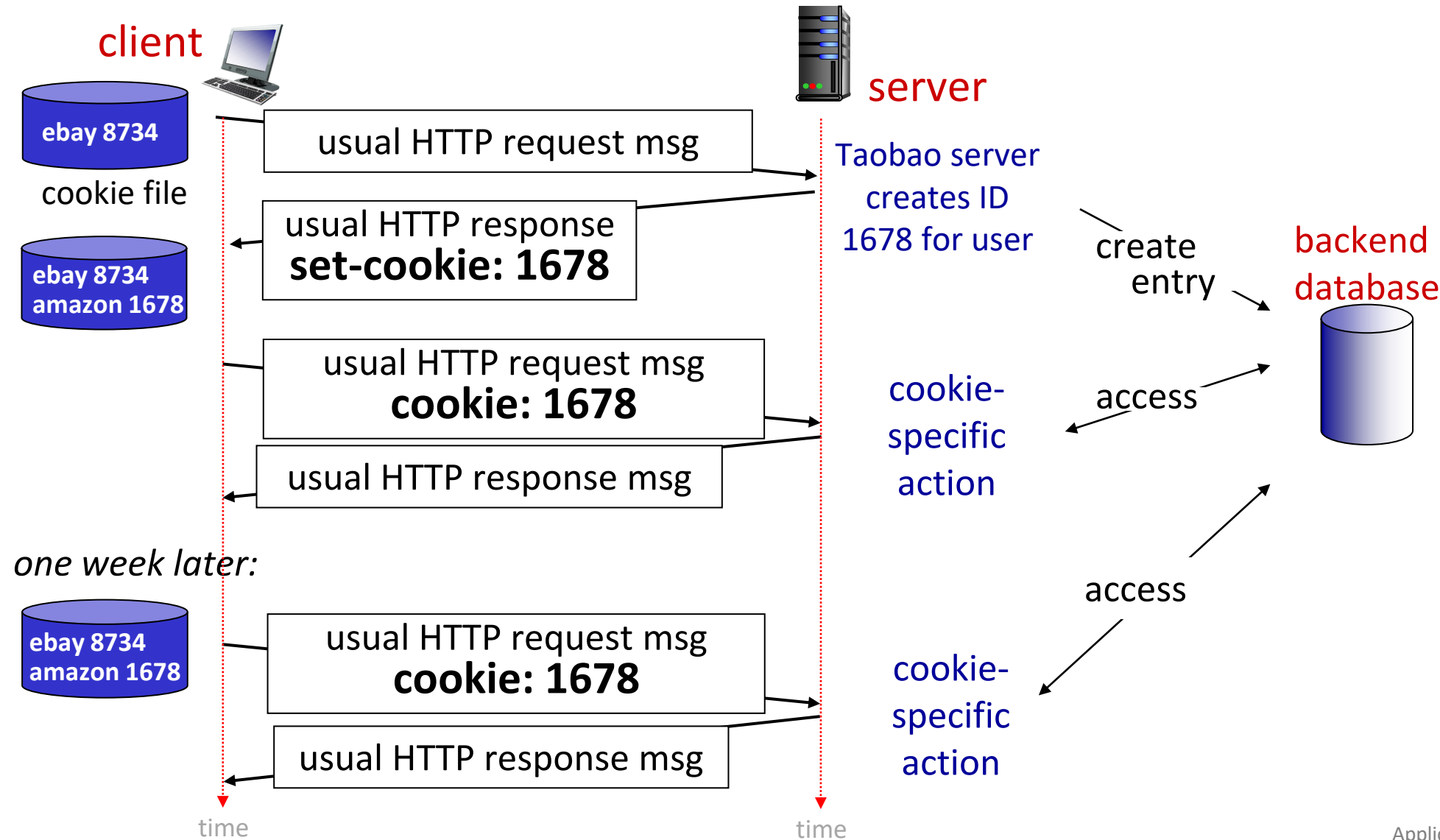
- 1) cookie header line of HTTP *response* message
- 2) cookie header line in next HTTP *request* message
- 3) cookie file kept on user's host, managed by user's browser
- 4) back-end database at Web site

## Example:

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

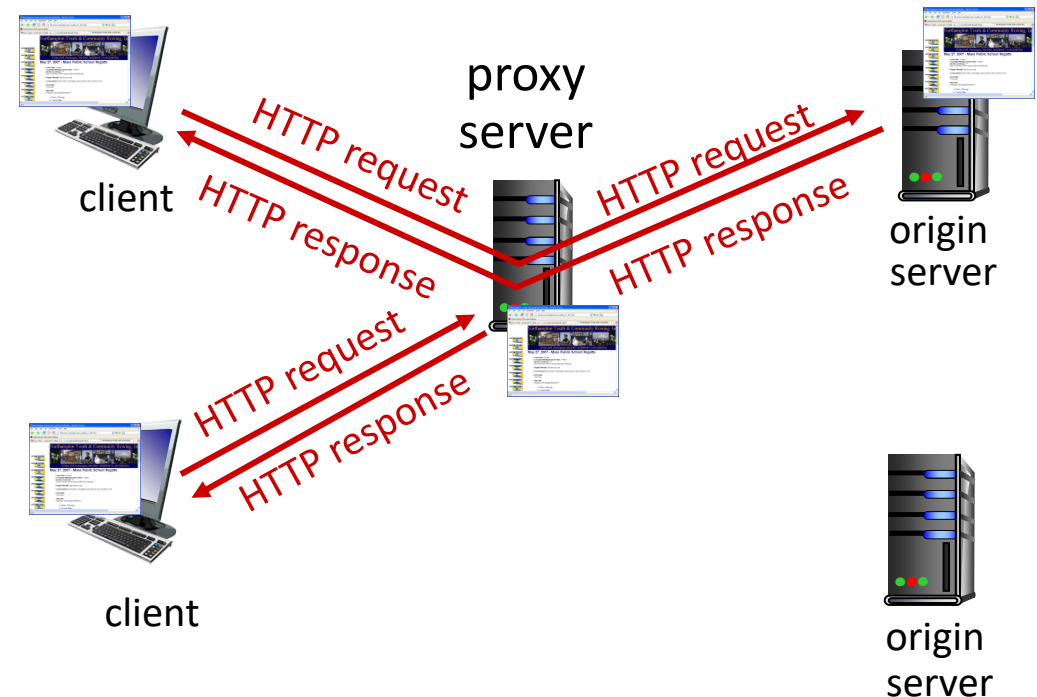
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

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

## *Why* Web caching?

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

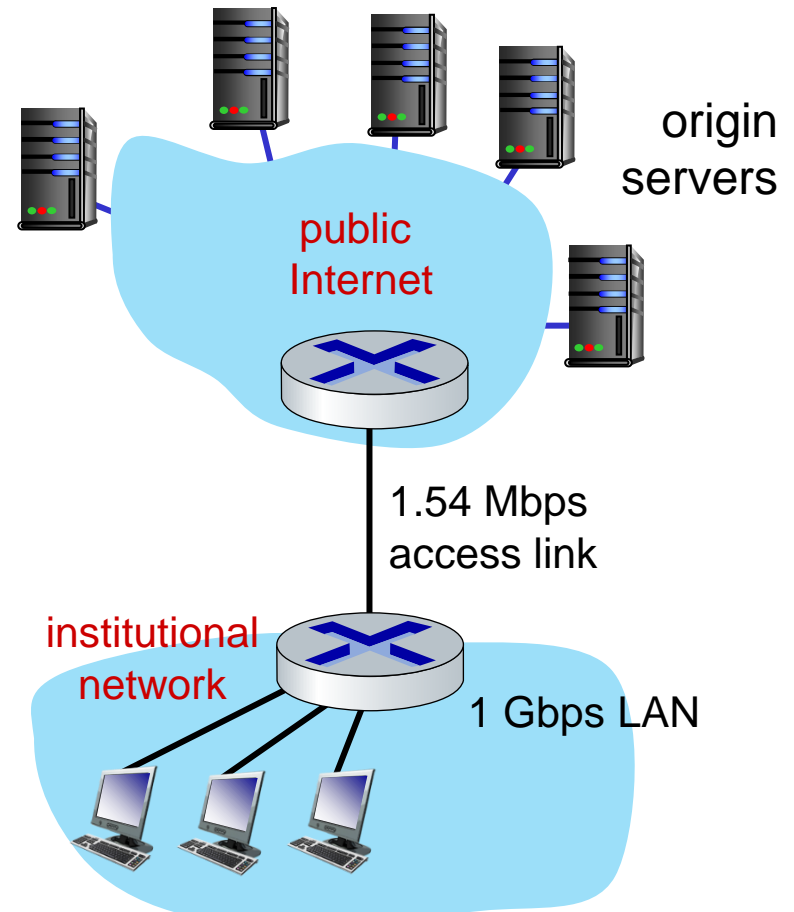
# Caching example

## Scenario:

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

## Performance:

- access link utilization = .97 *problem: large delays at high utilization!*
- LAN utilization: .0015
- end-end delay = Internet delay +  
access link delay + LAN delay  
= 2 sec + minutes + usecs



# Caching example: buy a faster access link

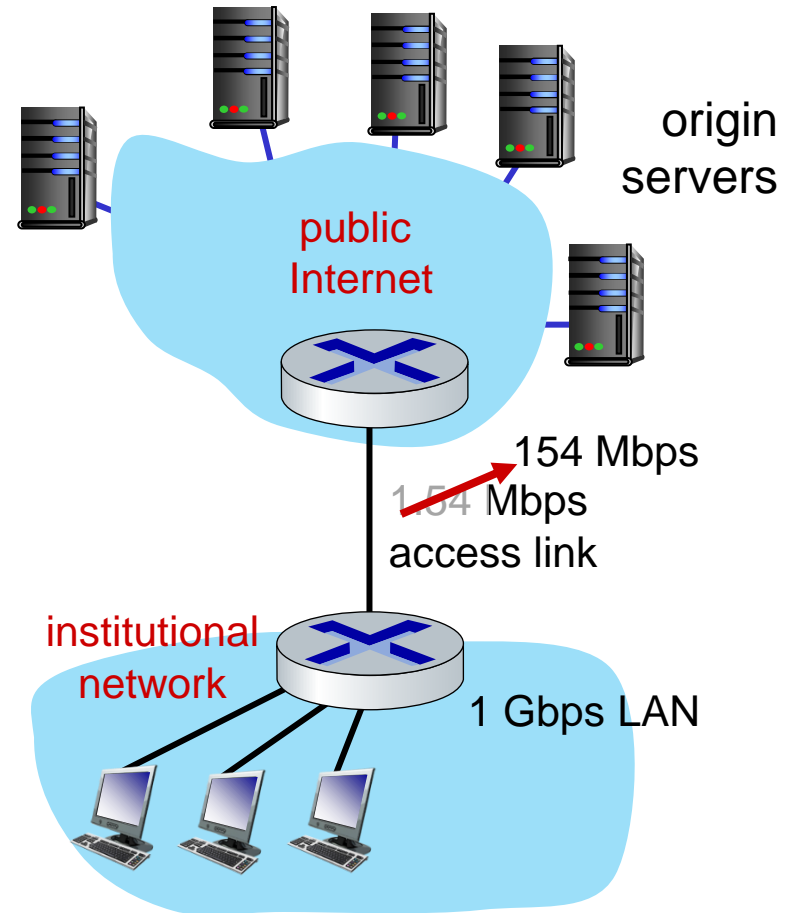
## Scenario:

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

## Performance:

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

**Cost:** faster access link (expensive!) → msec



# Caching example: install a web cache

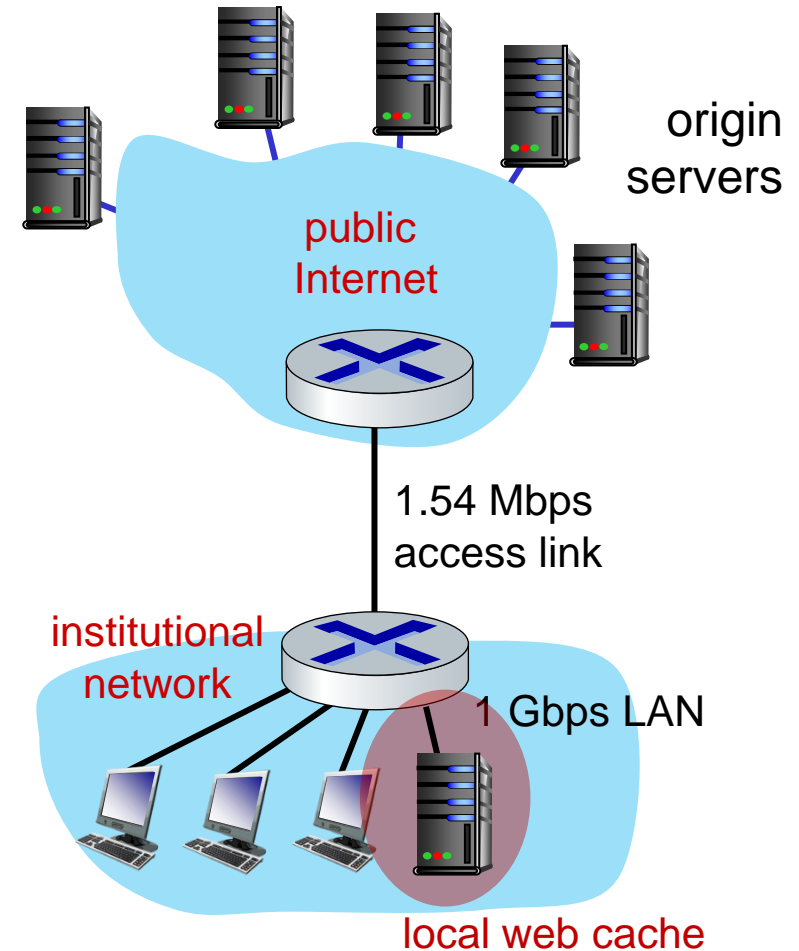
## Scenario:

- 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/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!)

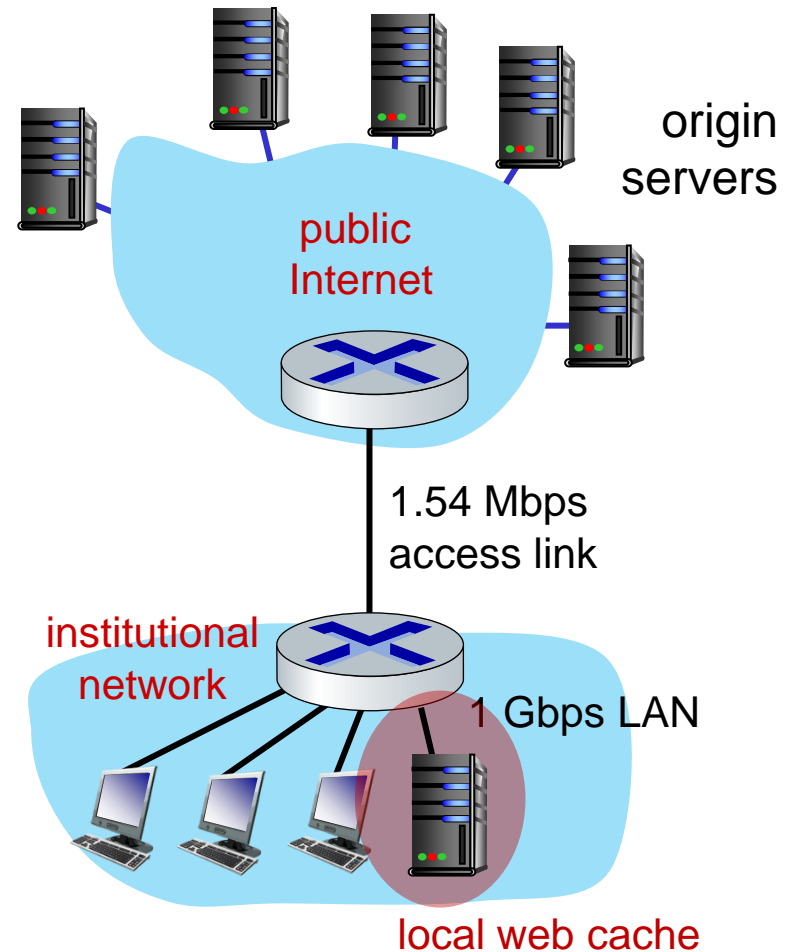


# Caching example: install a web cache

## Calculating access link utilization, end-end delay with cache:

- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link  
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
- utilization  $= 0.9 / 1.54 = .58$
- average end-end delay (*estimation*)  
 $= 0.6 * (\text{delay from origin servers})$   
 $+ 0.4 * (\text{delay when satisfied at cache})$   
 $= 0.6 * (2.01) + 0.4 * (\sim \text{msecs}) = \sim 1.2 \text{ secs}$

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

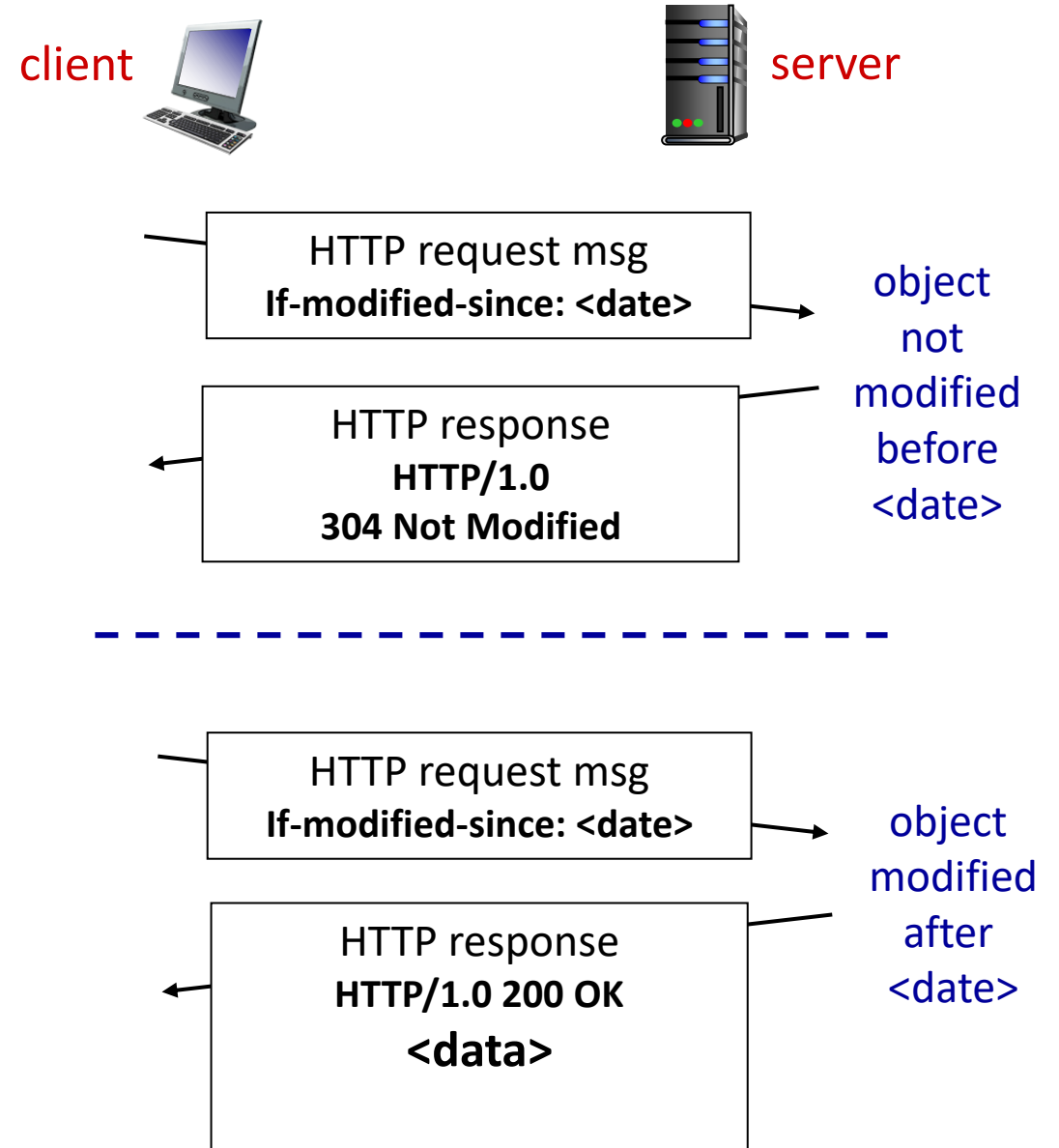




# Conditional GET

**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**



# 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

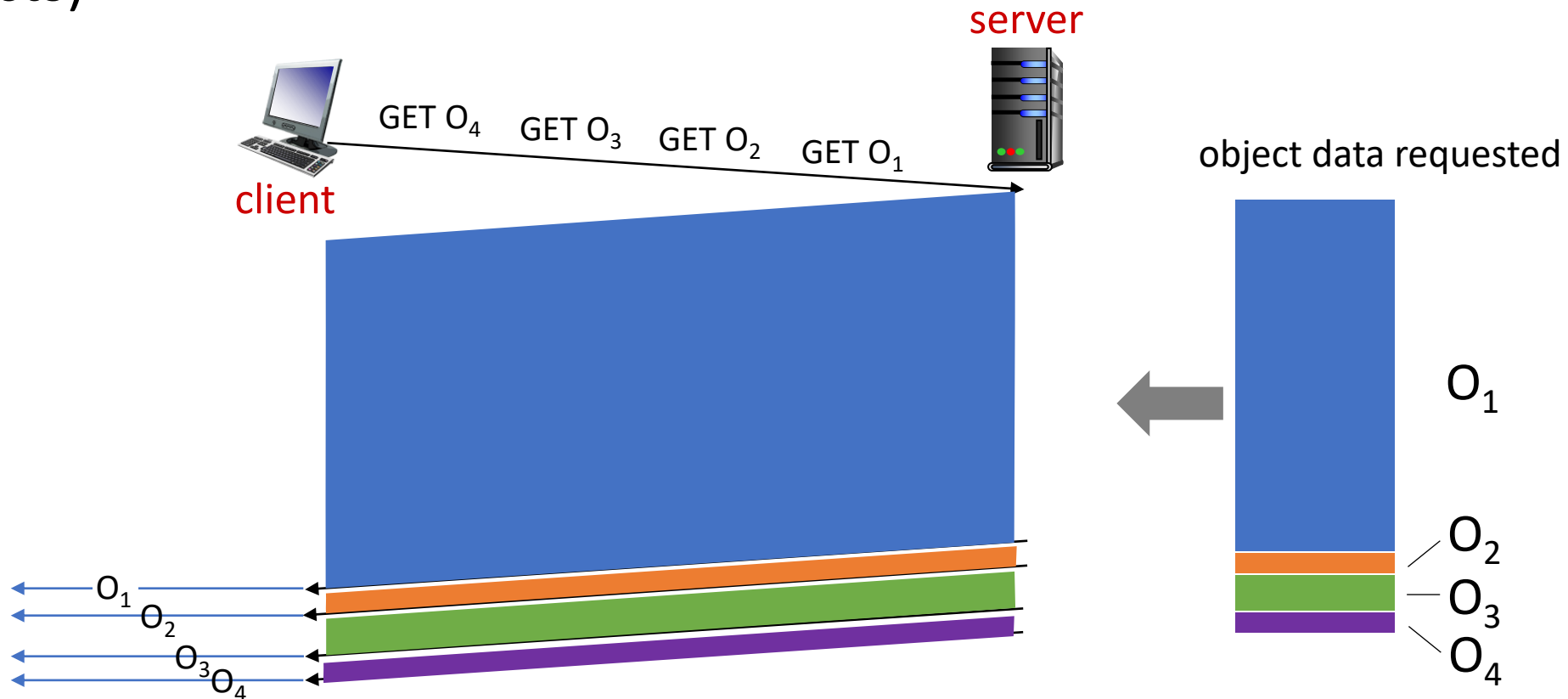
*Key goal:* decreased delay in multi-object HTTP requests

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

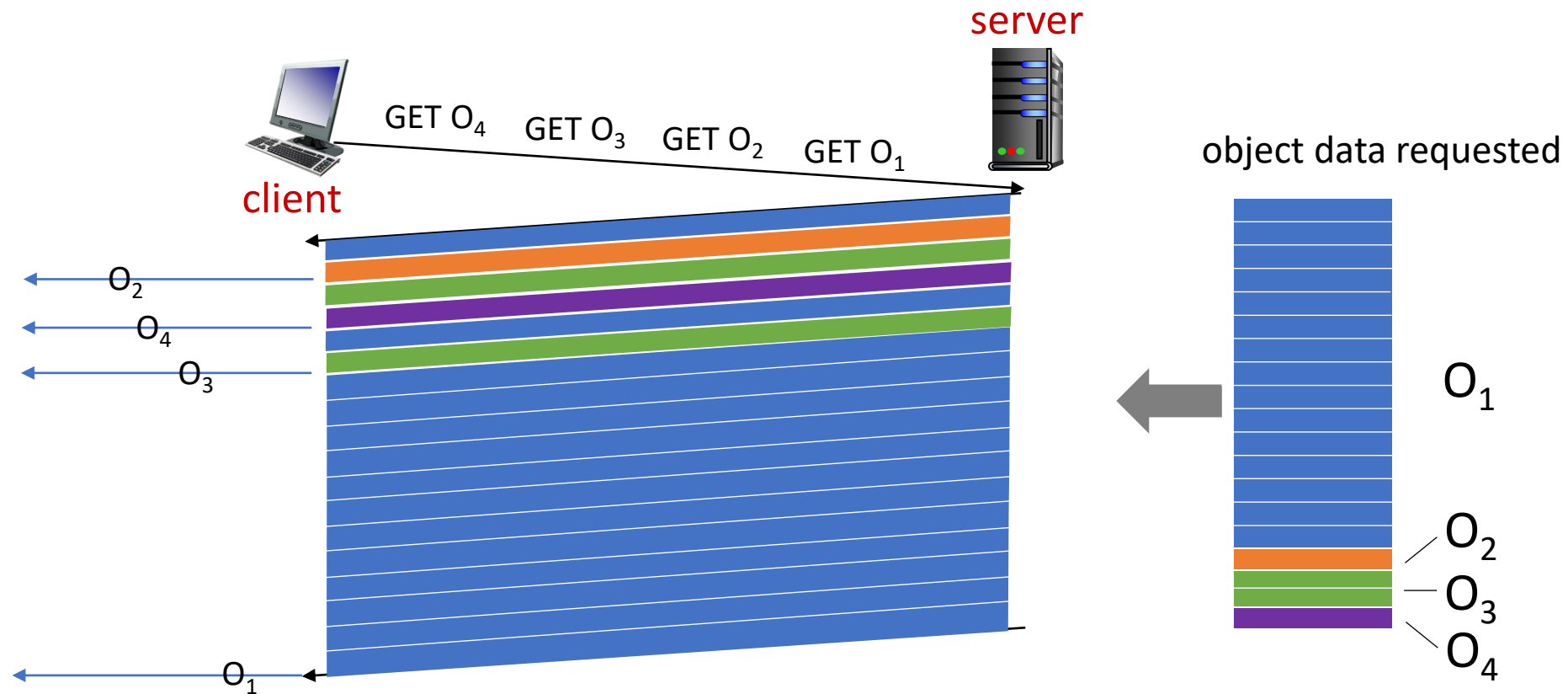
HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)



*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<sub>2</sub>, O<sub>3</sub>, O<sub>4</sub> delivered quickly, O<sub>1</sub> 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



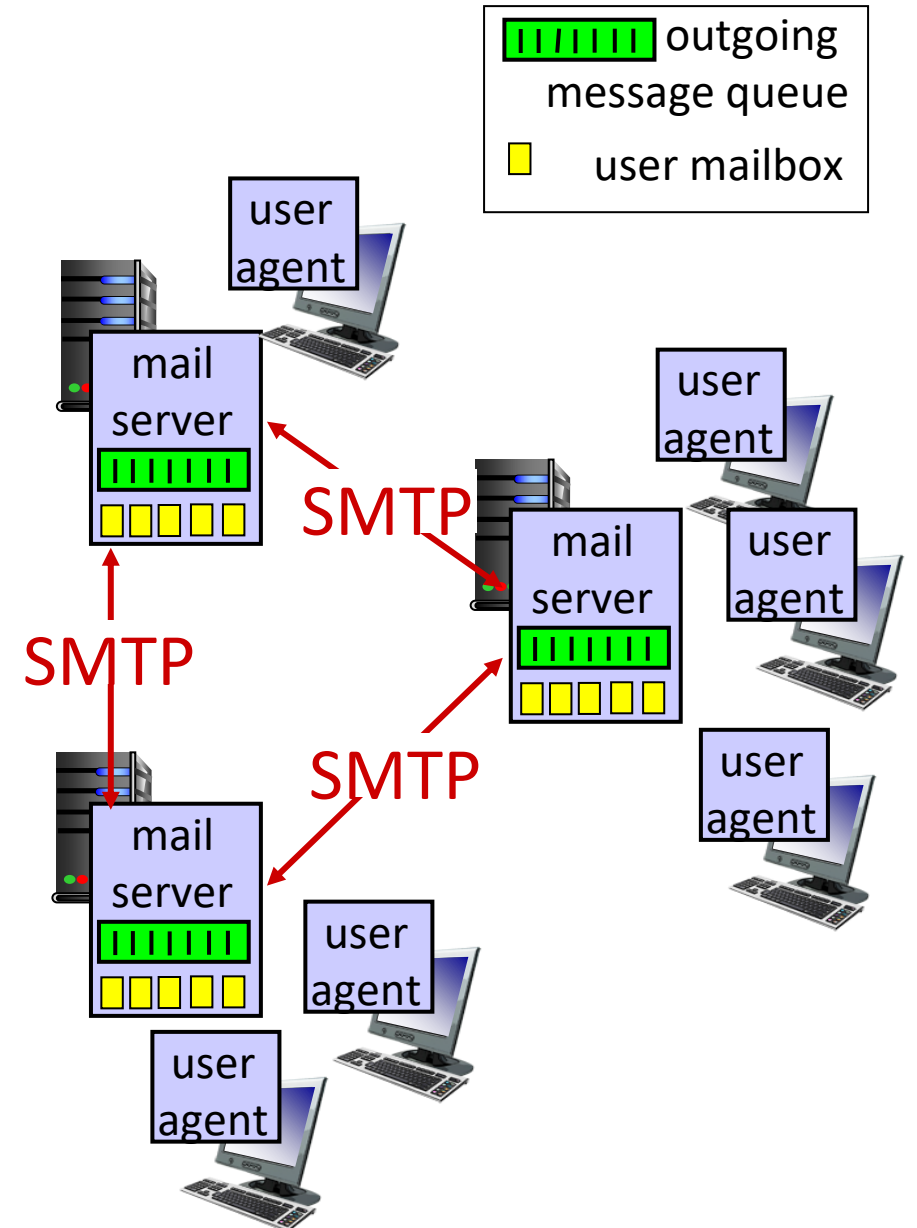
# E-mail

## 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, phone mail client
- outgoing, incoming messages stored on server

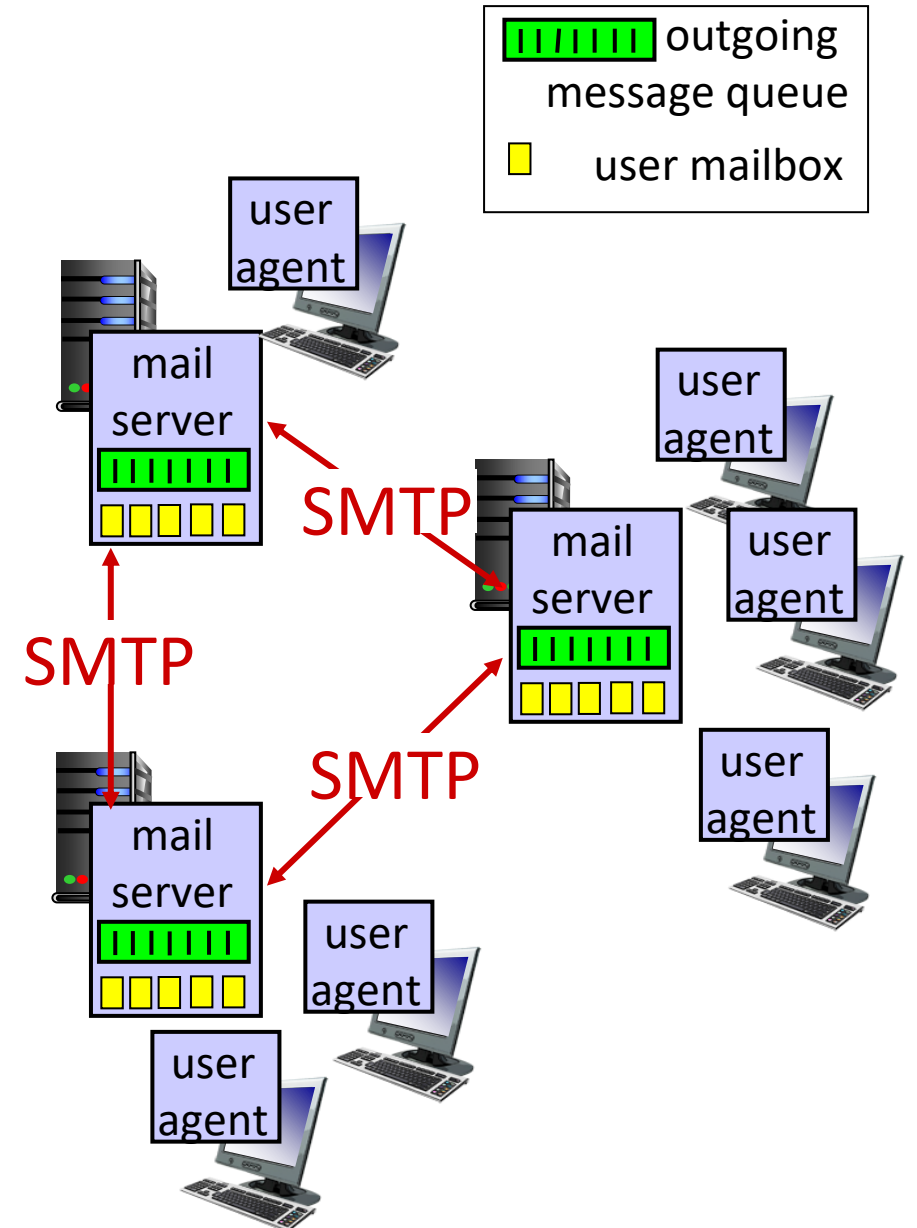




# E-mail: mail servers

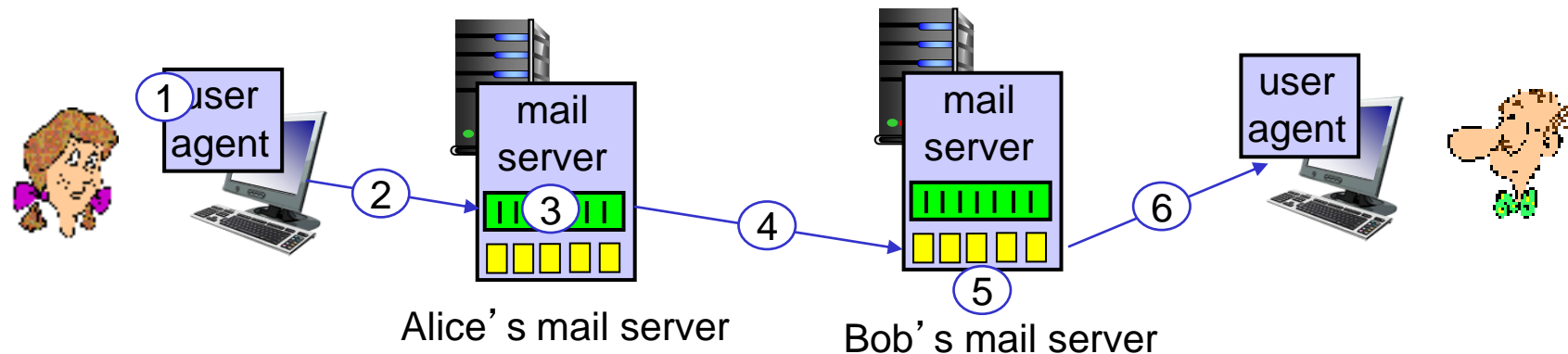
## mail servers:

- *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



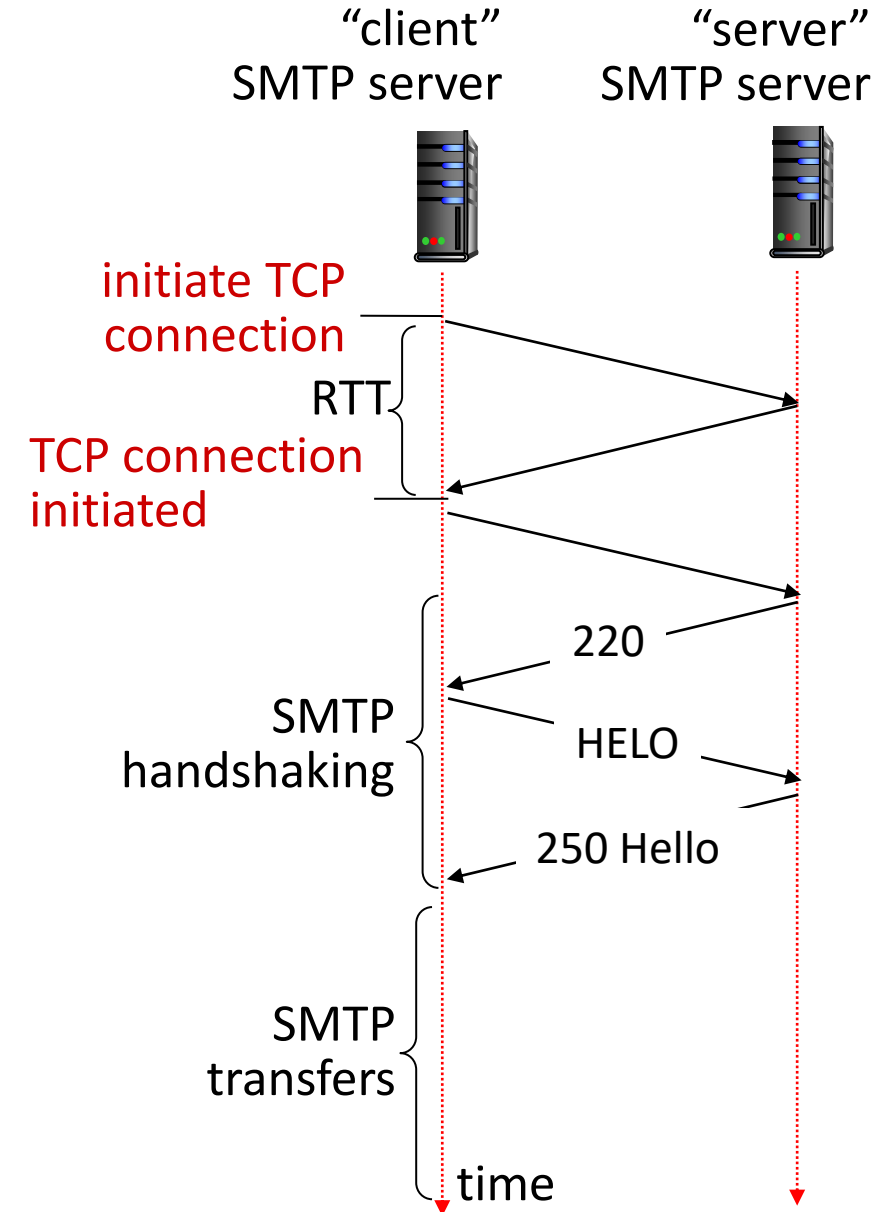
# Scenario: Alice sends e-mail to Bob

- 1) Alice uses UA to compose e-mail message "to" bob@some school.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: from 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

```
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
```

# SMTP: closing observations

## *comparison with HTTP:*

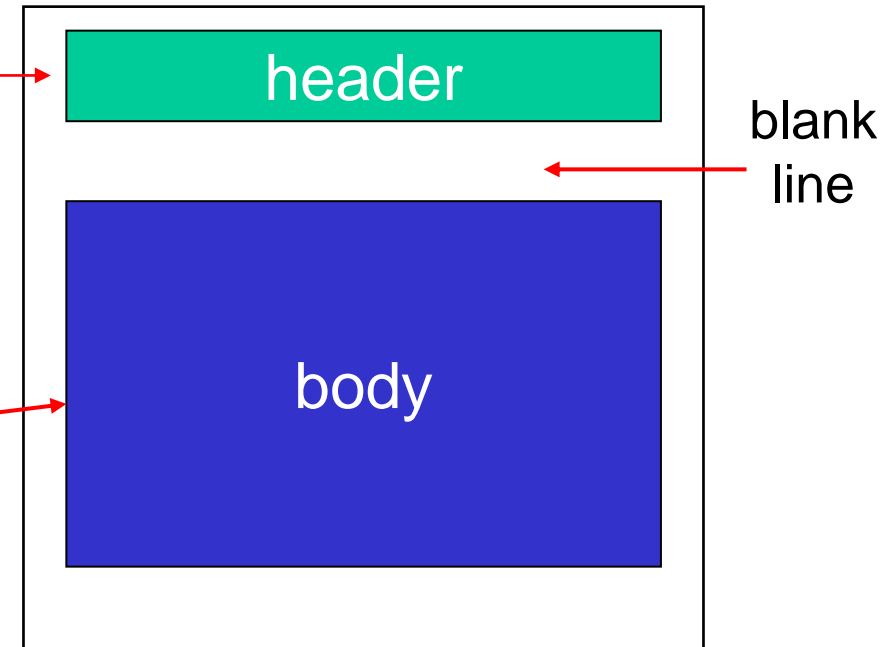
- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in a multipart message
- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses CRLF.CRLF to determine end of message

# Mail message format

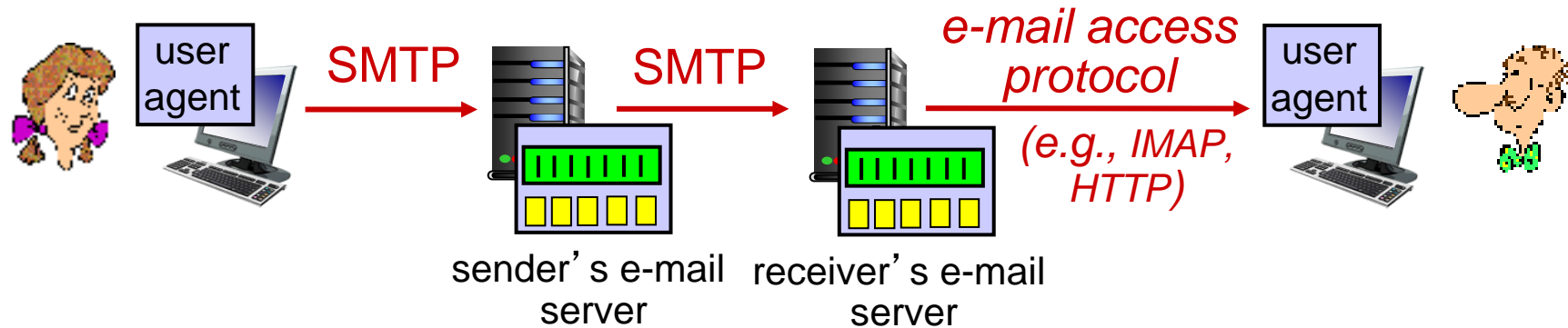
SMTP: protocol for exchanging e-mail messages, defined in RFC 531 (like HTTP)

RFC 822 defines *syntax* for e-mail message itself (like HTML)

- 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



- **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:

- student ID, 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

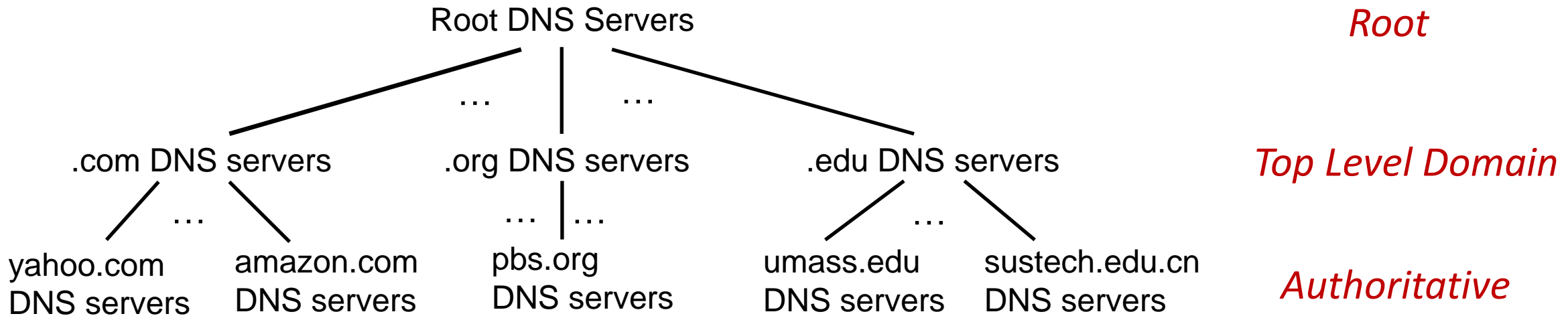
## *Q: Why not centralize DNS?*

- 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



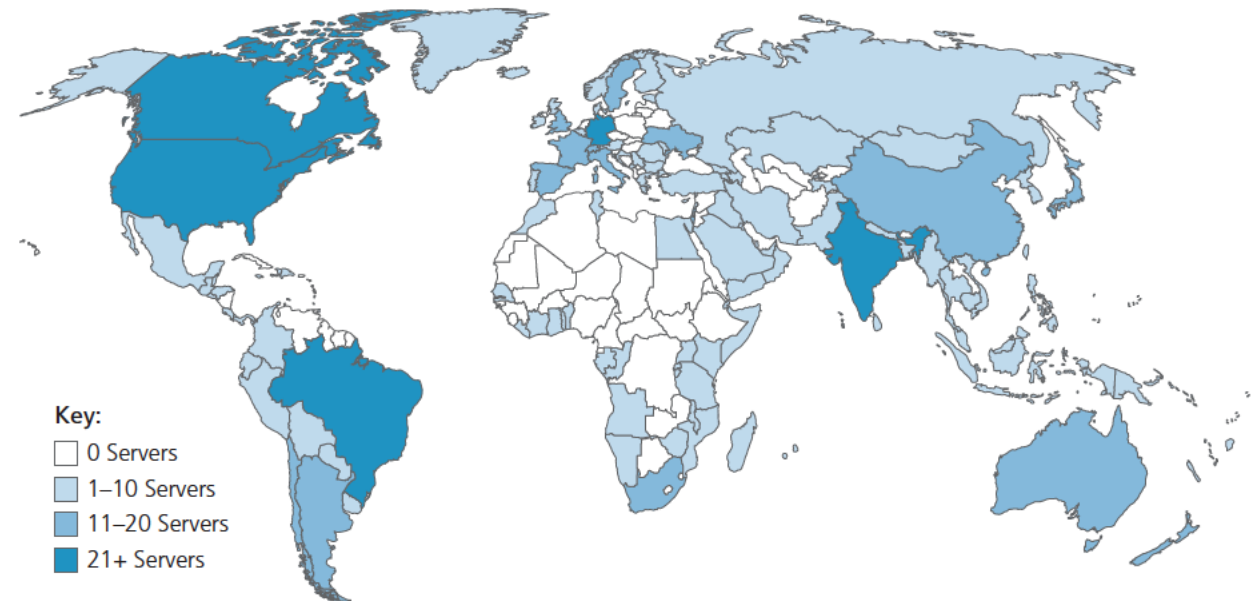
Client wants IP address for `www.amazon.com`; 1<sup>st</sup> approximation:

- 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)



# Top-Level Domain, and 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

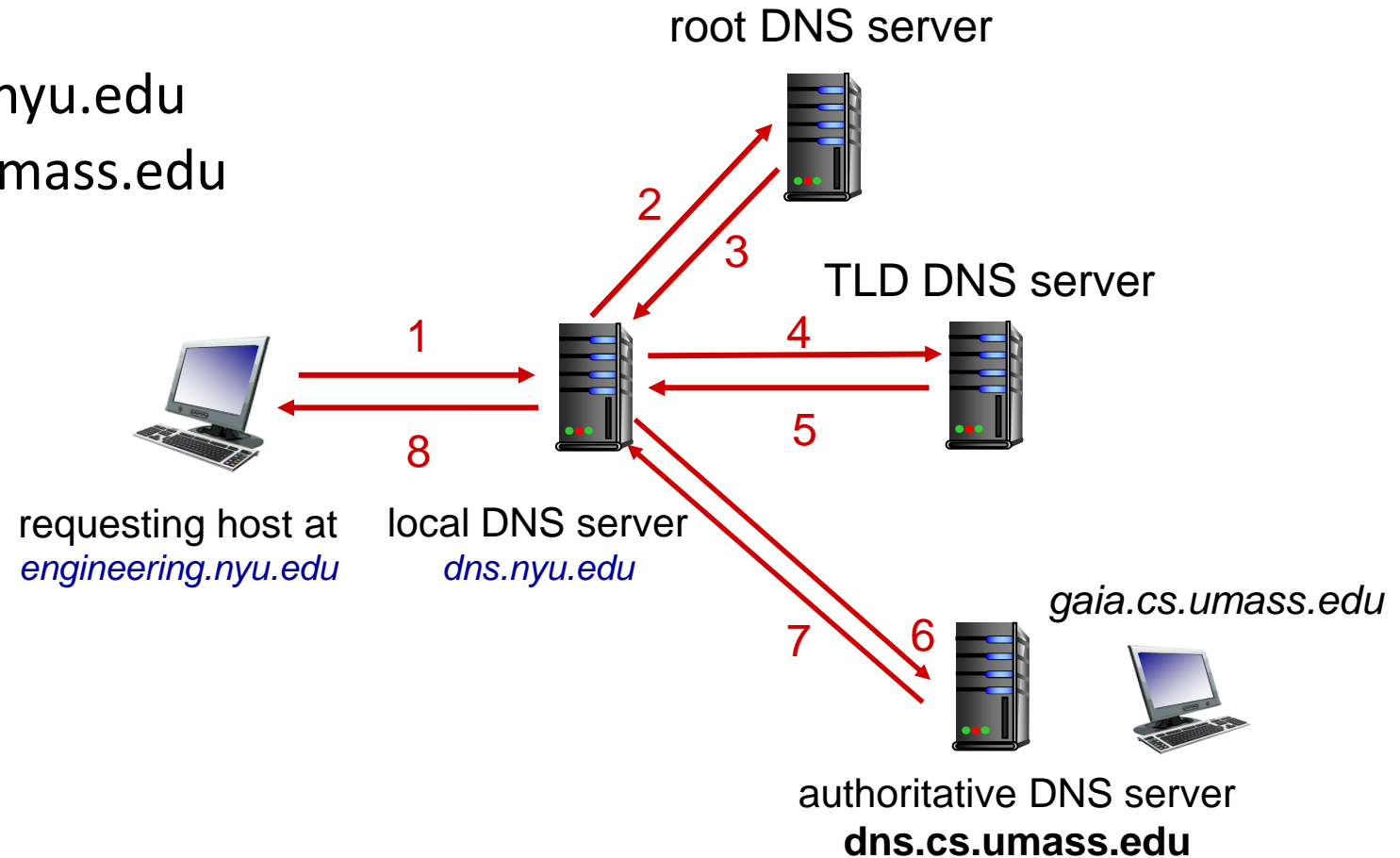
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

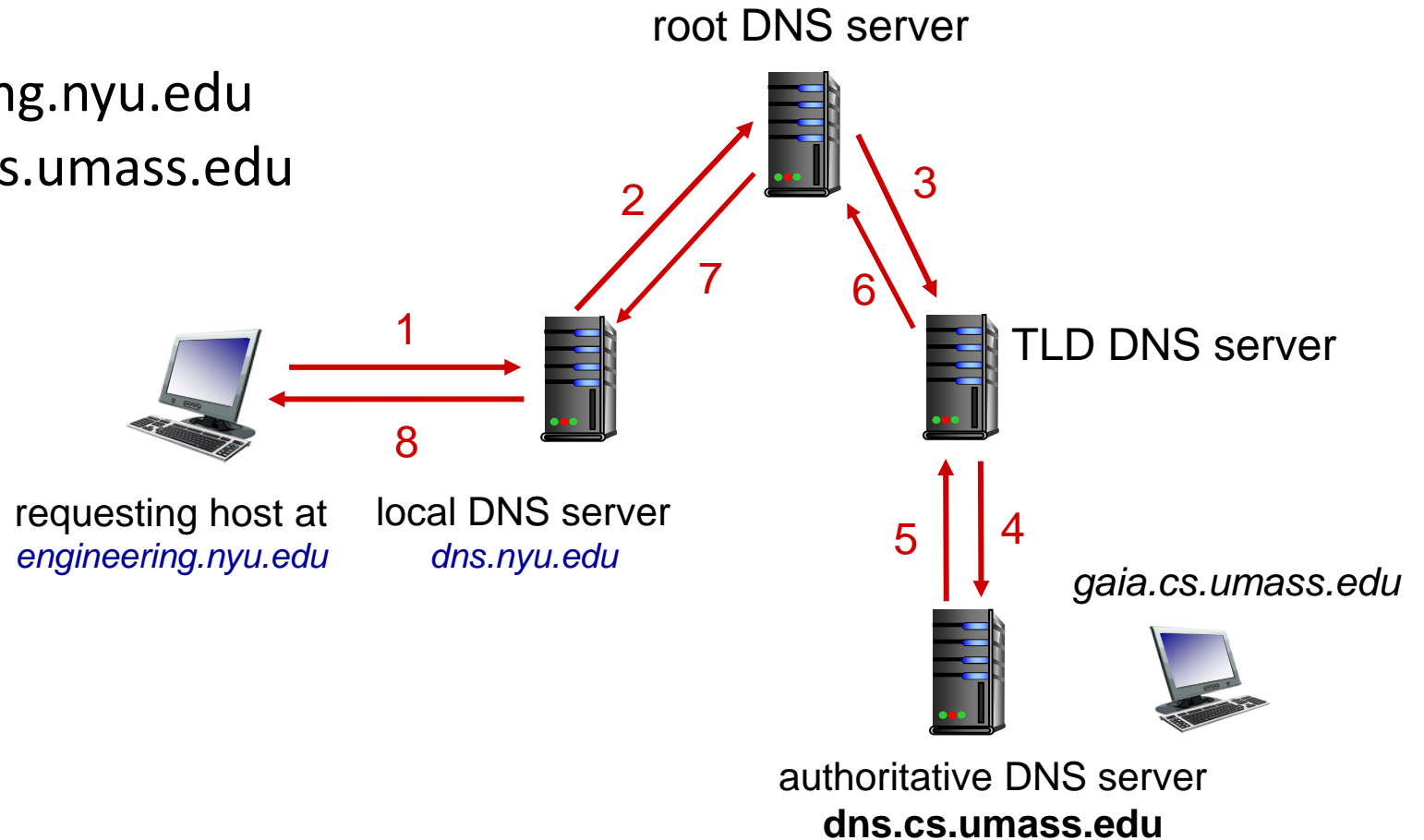


# DNS name resolution: recursive query

**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 *cached* 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!

# DNS records

**DNS:** distributed database storing resource records (RR)

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

## type=A

- name is hostname
- value is IP address

## type=NS

- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

## type=CNAME

- name is alias name for some “canonical” (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name

## type=MX

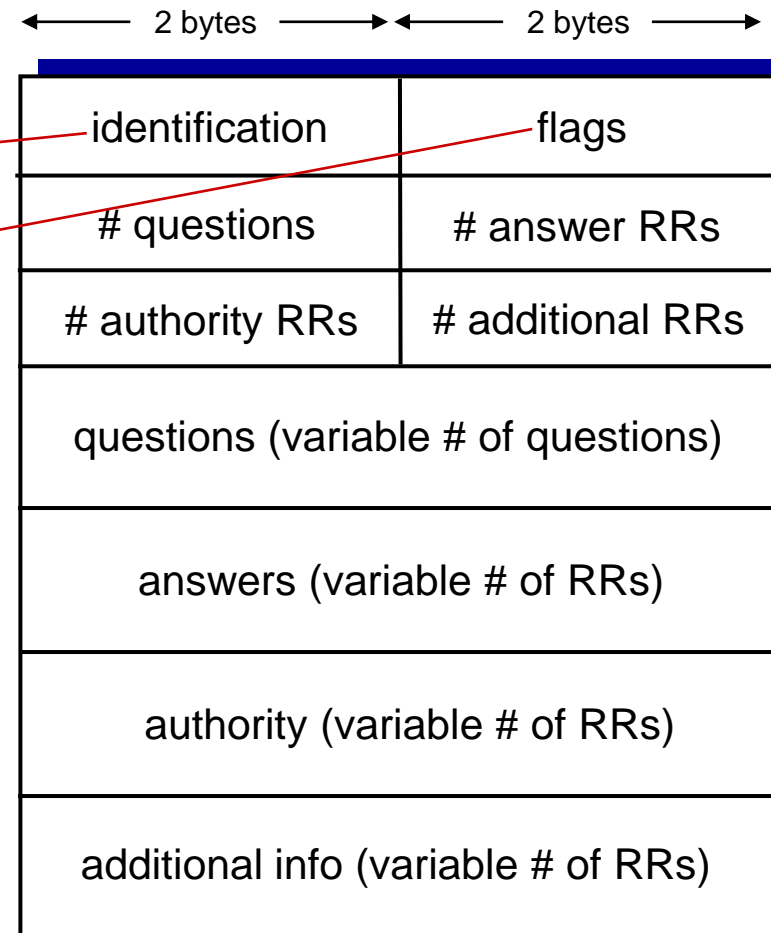
- value is name of mailserver associated with name

# DNS protocol messages

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

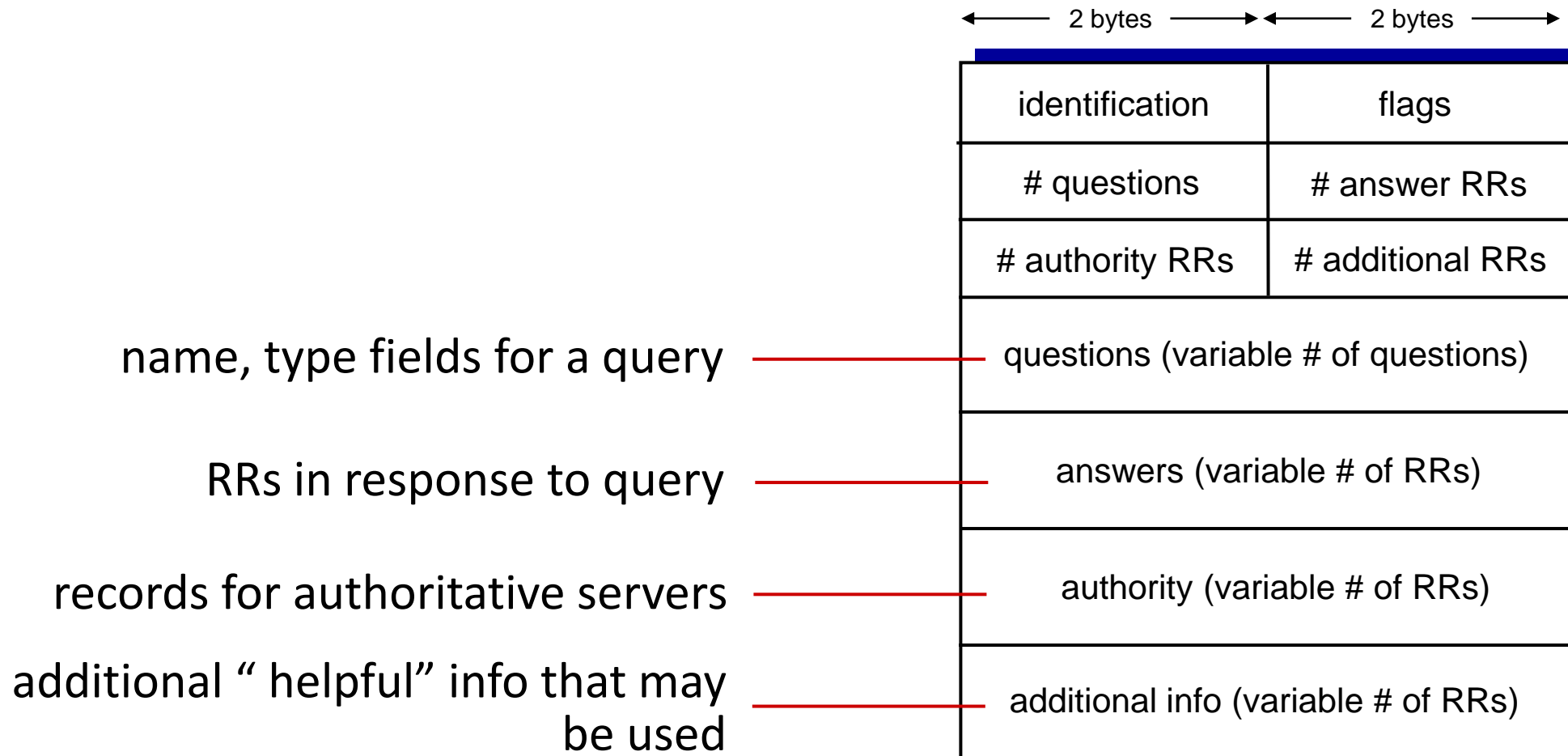
message header:

- **identification**: 16 bit # for query, reply to query uses same #
- **flags**:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



# DNS protocol messages

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



# Inserting records into DNS

Example: new startup “Network Utopia”

- register name networkutopia.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.networkutopia.com`
  - type MX record for `networkutopia.com`

# 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

## Redirect attacks

- man-in-middle
  - intercept DNS queries
- DNS poisoning
  - send bogus replies to DNS server, which caches

## Exploit DNS for DDoS

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

DNSSEC  
[RFC 4033]