

## Application Layer Introduction



# Network Applications

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

***Q:*** *your favorites?*

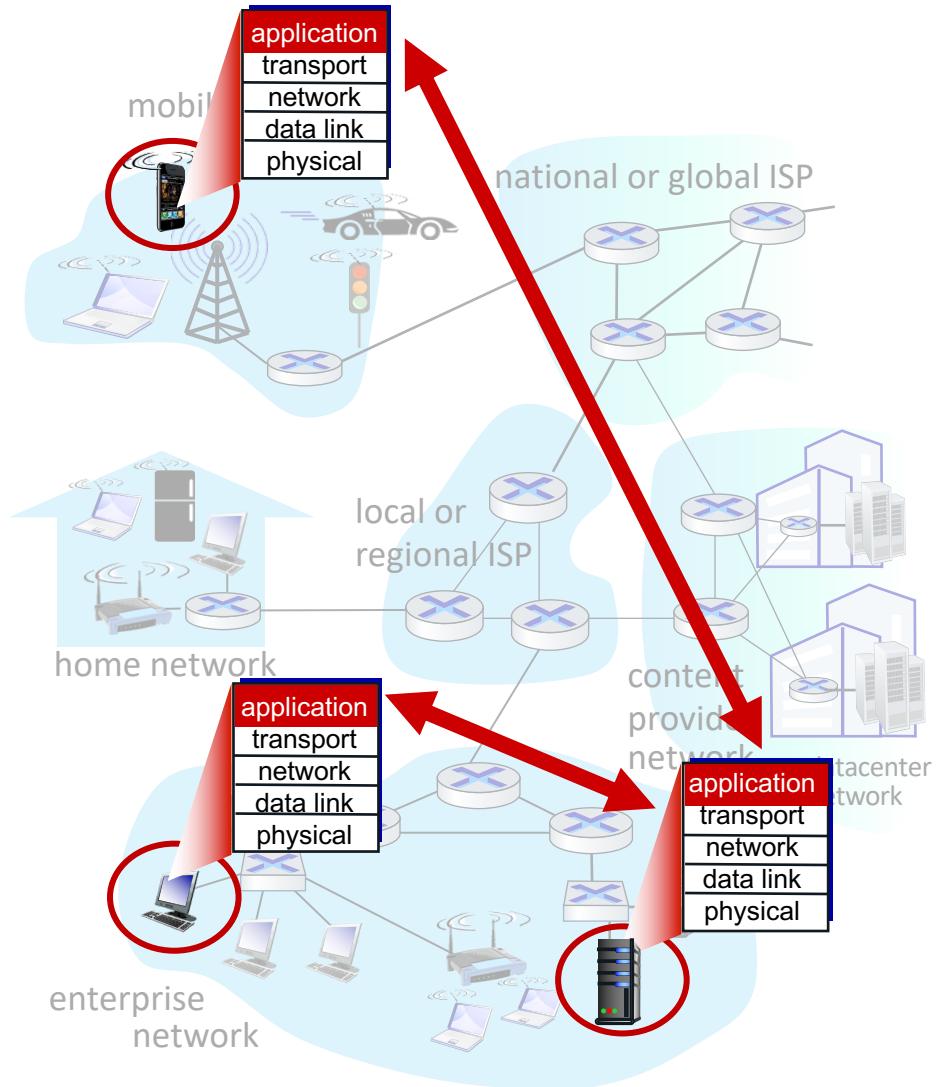
# How to Create A Network App

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



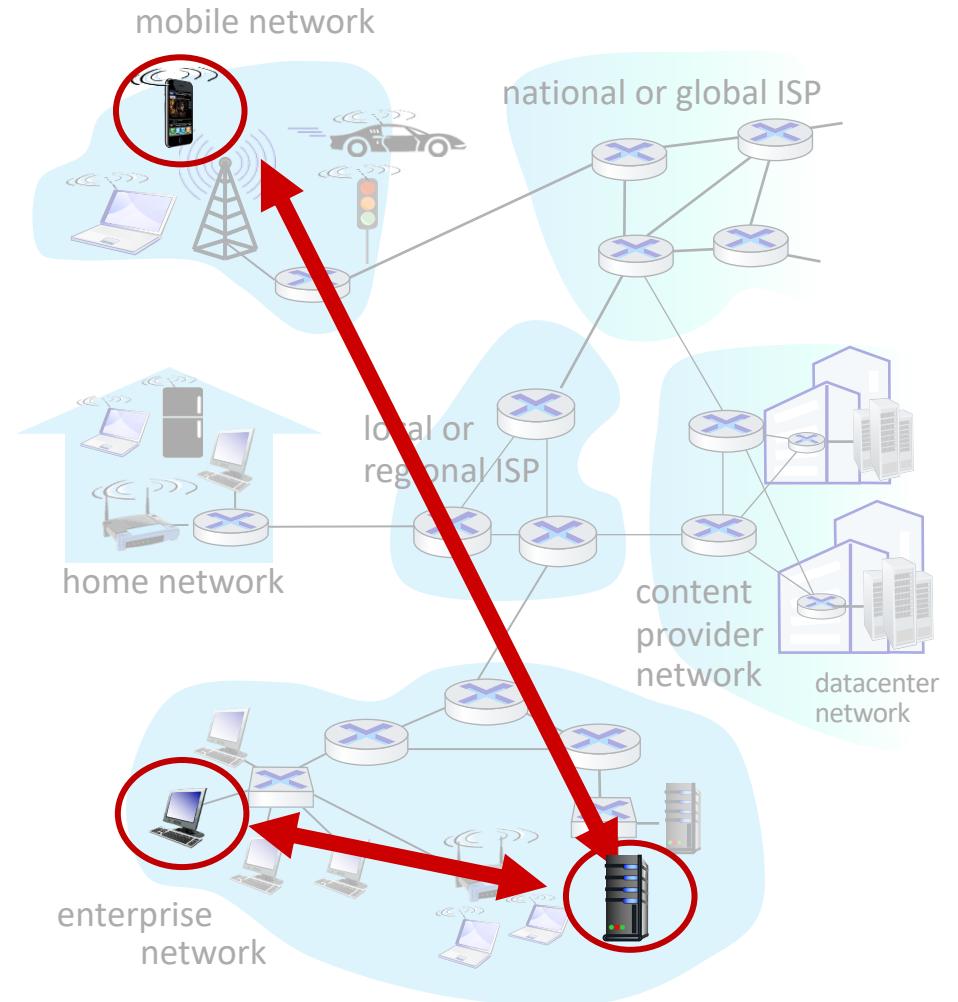
# Client-Server Paradigm

## server:

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

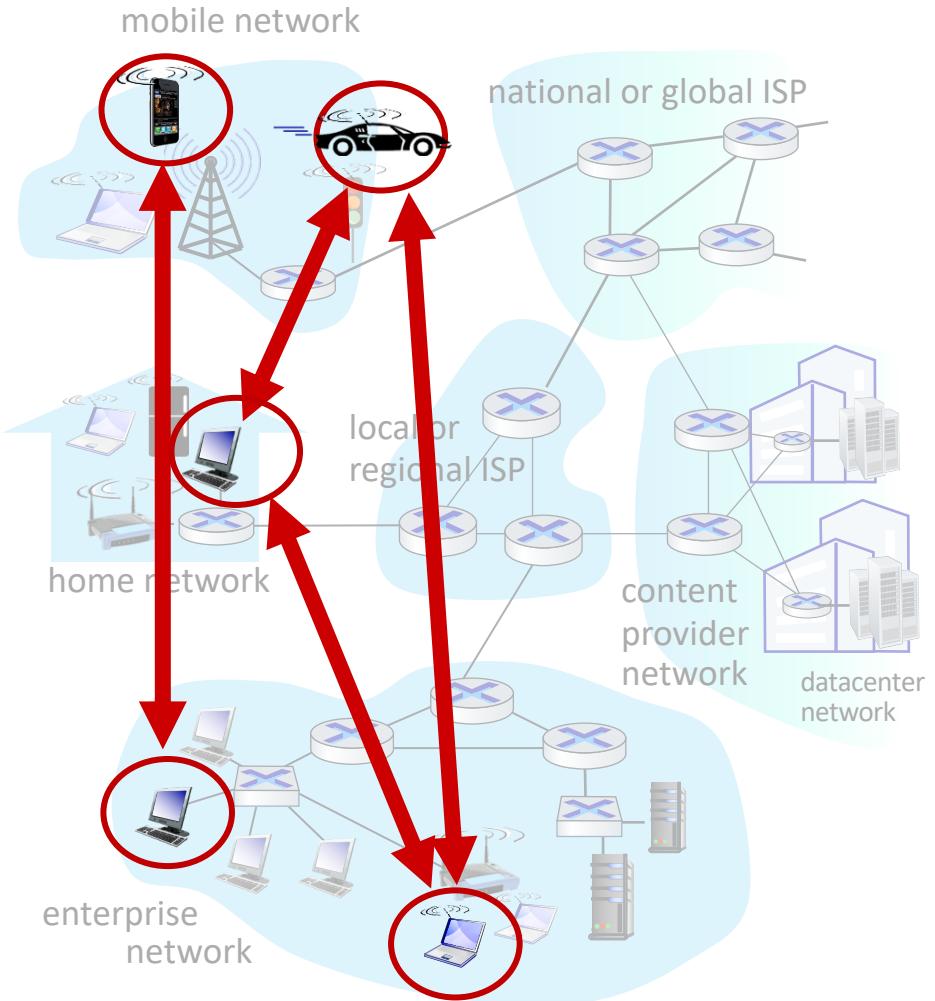
## clients:

- 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 Communication

*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

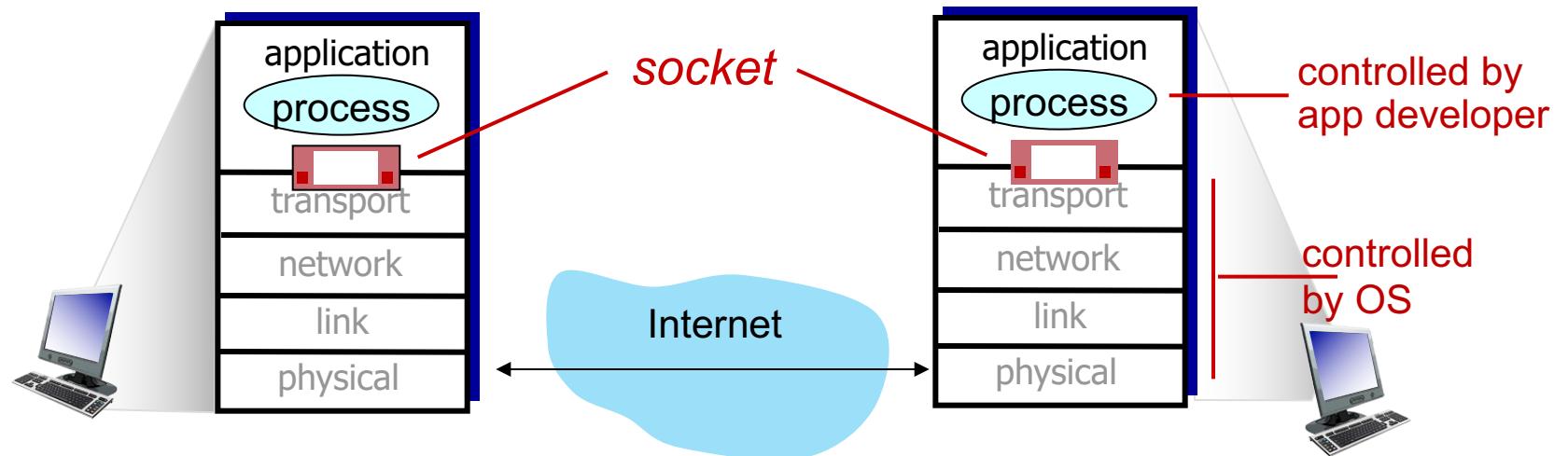
*client process*: process that initiates communication

*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



# 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

## 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

## timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

## security

- encryption, data integrity, ...

# Transport Service Requirements

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 Protocol 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

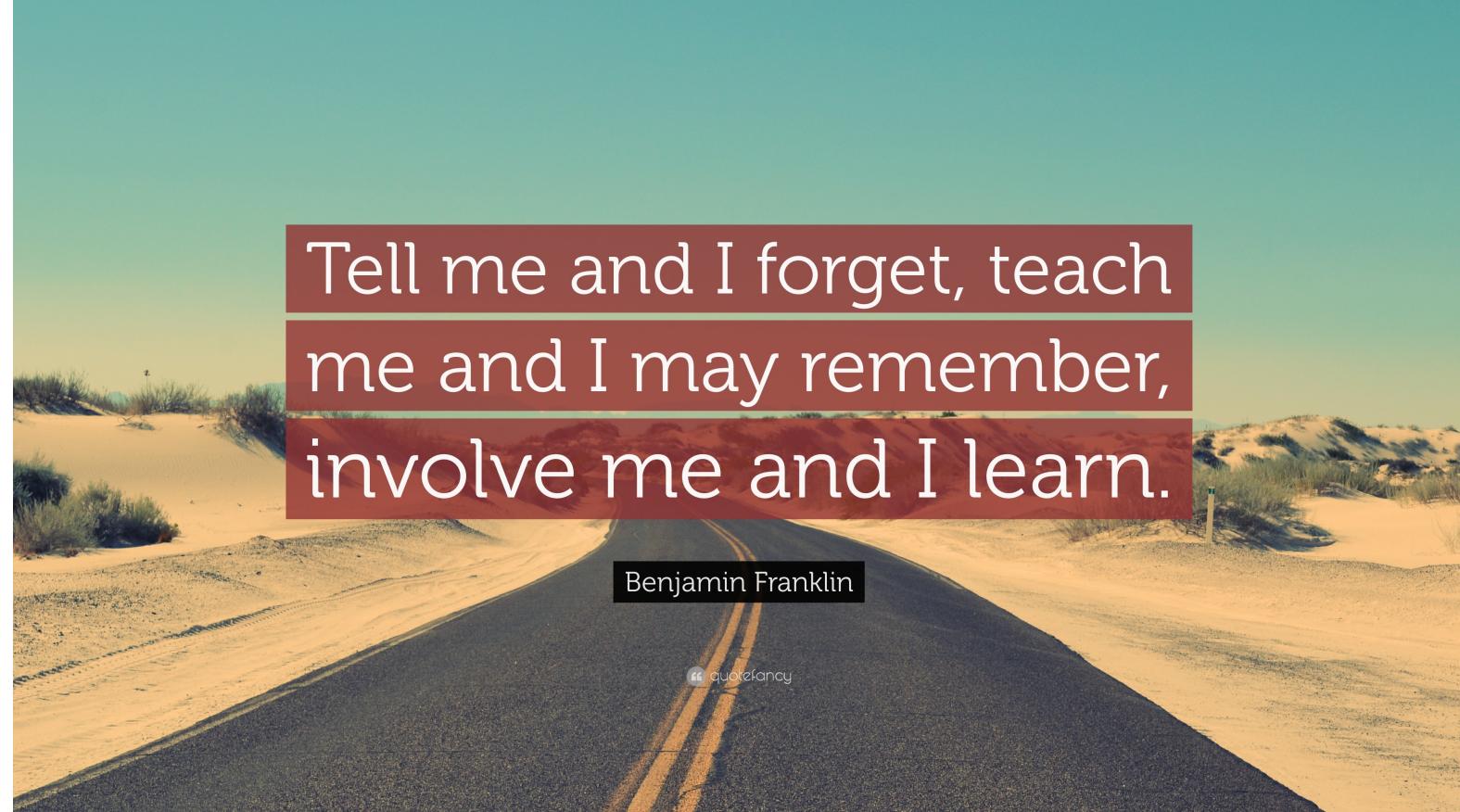
## UDP service:

- ***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 Applications and Transport Protocols

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



## Web and HTTP (Part 1)



# Web and HTTP

*First, a quick review...*

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

www.someschool.edu/someDept/pic.gif

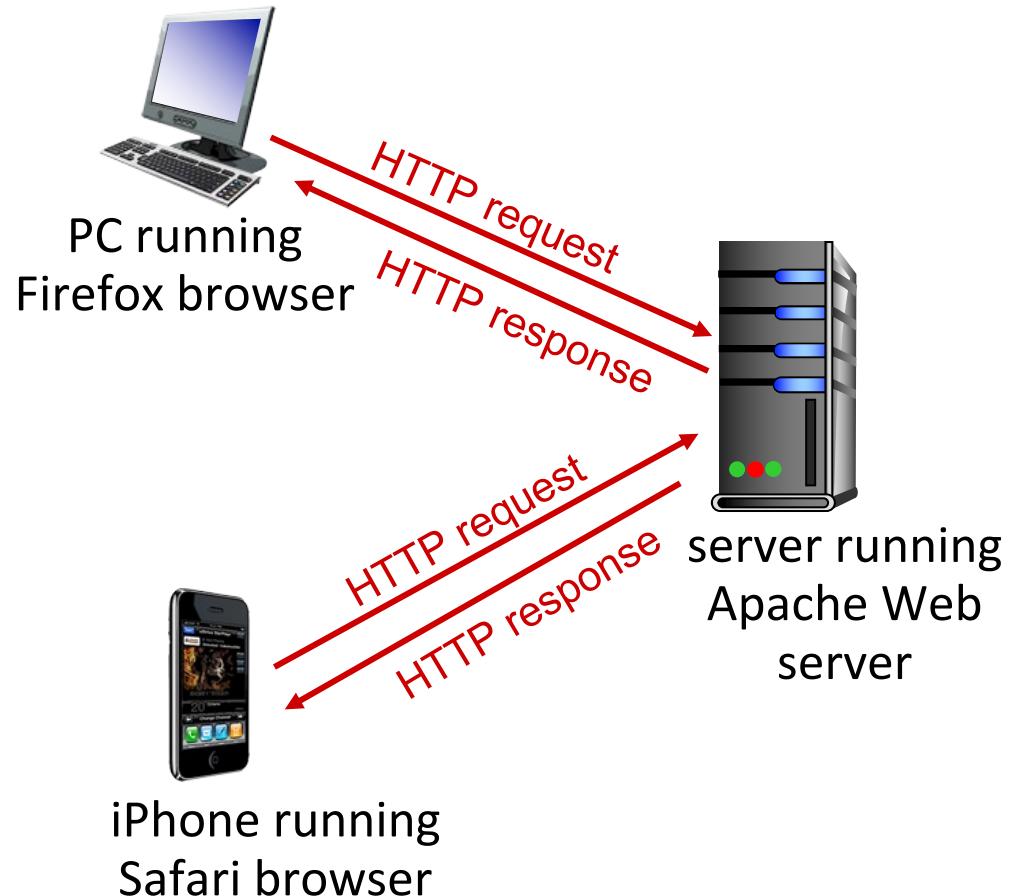
host name

path name

# HTTP Overview

## HTTP: hypertext transfer protocol

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



# 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

time ↓

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

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



- time ↓
1. User enters URL
2. HTTP client sends TCP connection request to port 80 of the server
3. HTTP server accepts connection request
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

4. HTTP server closes TCP connection.

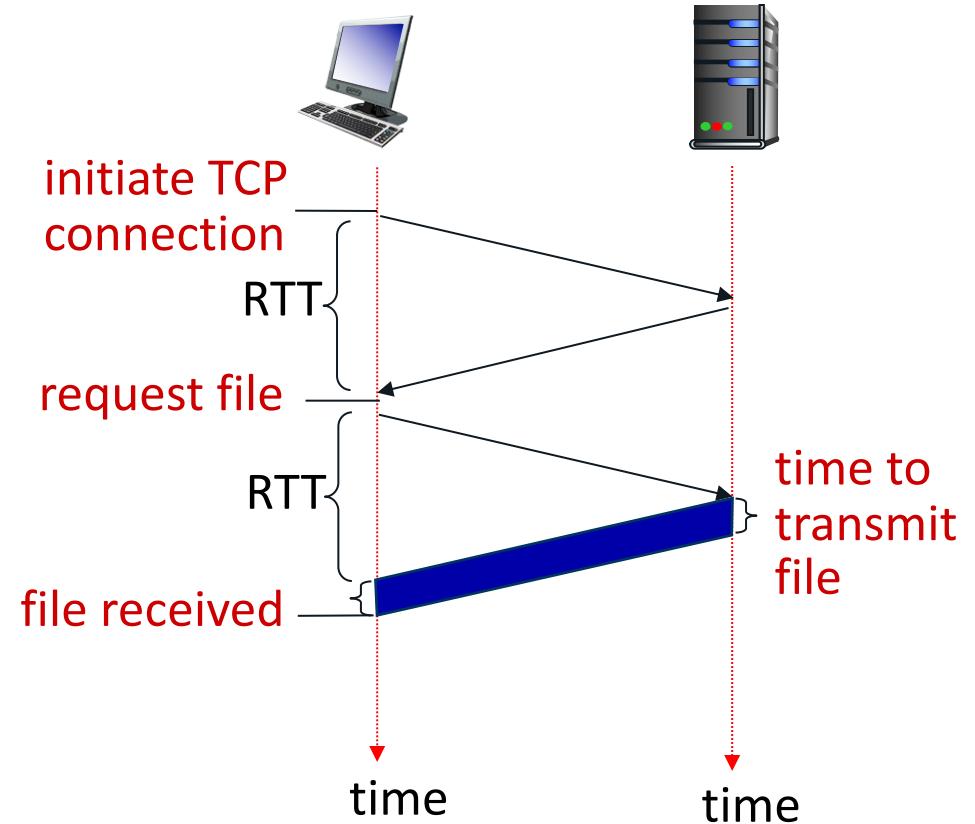


# Non-Persistent HTTP: Response Time

**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



$$\text{Non-persistent HTTP response time} = 2\text{RTT} + \text{file transmission time}$$

# Persistent HTTP (http 1.1)

## *Non-persistent HTTP issues:*

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

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

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

# HTTP Request Message

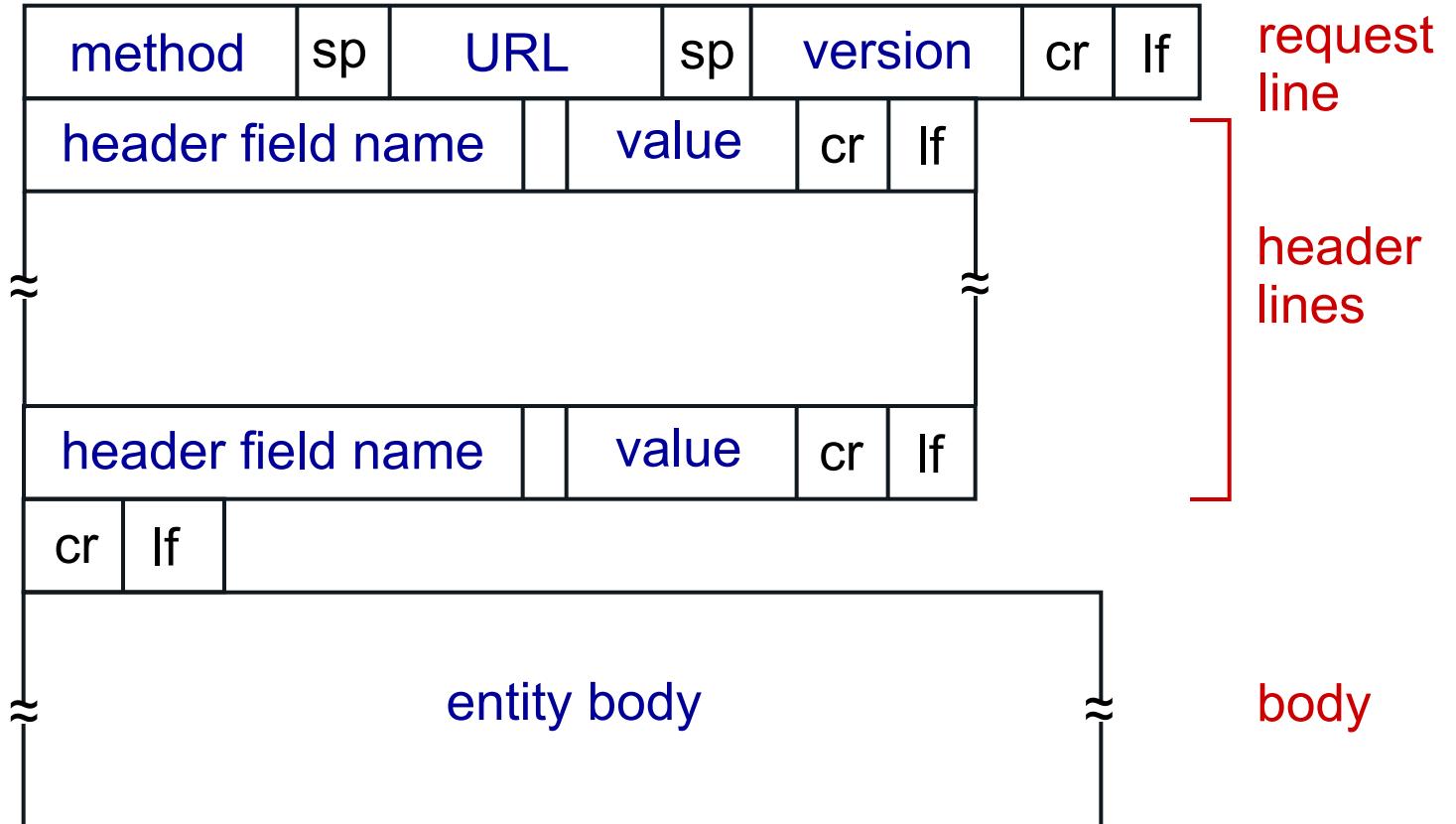
- two types of HTTP messages: *request, response*
- **HTTP request message:**
  - ASCII (human-readable format)

request line (GET, POST,  
HEAD commands) →

carriage return character  
line-feed character

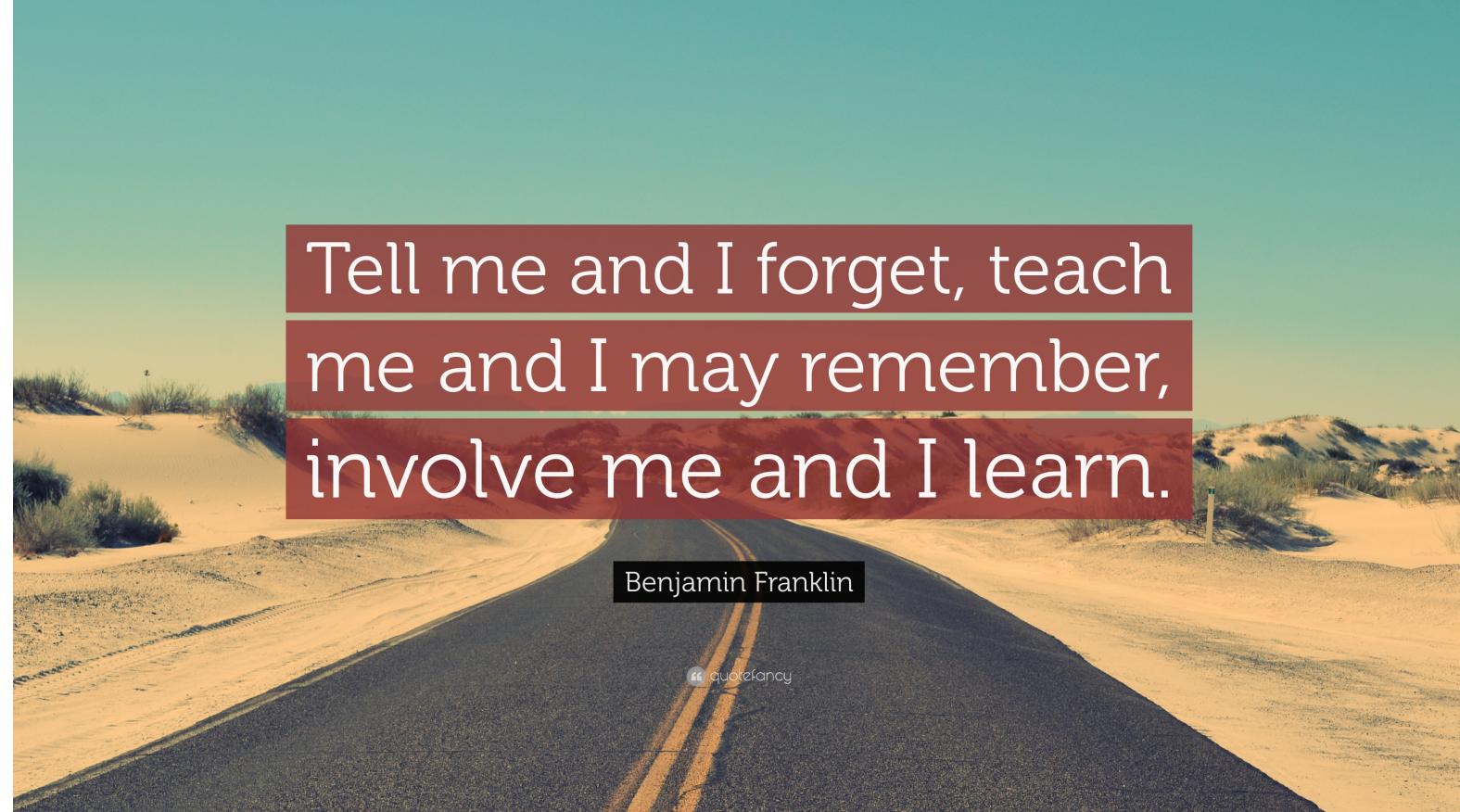
carriage return, line feed →  
at start of line indicates  
end of header lines

# HTTP Request Message: General Format



# HTTP Response Message

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



Tell me and I forget, teach  
me and I may remember,  
involve me and I learn.

Benjamin Franklin

" quotefancy

## Web and HTTP (Part 2)



# Maintaining User/Server State: Cookies

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

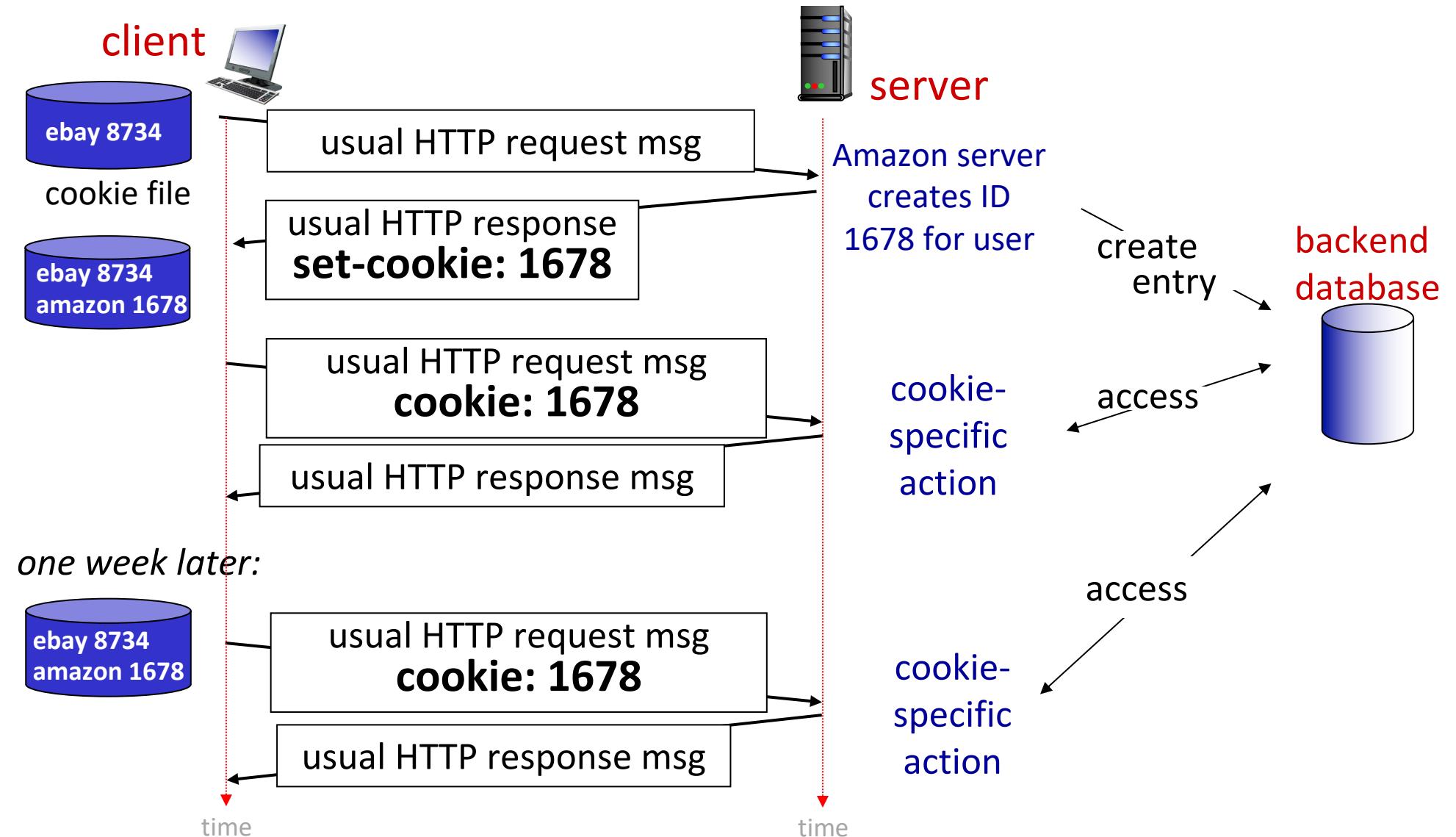
*four components:*

- 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



## *What cookies can be used for:*

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

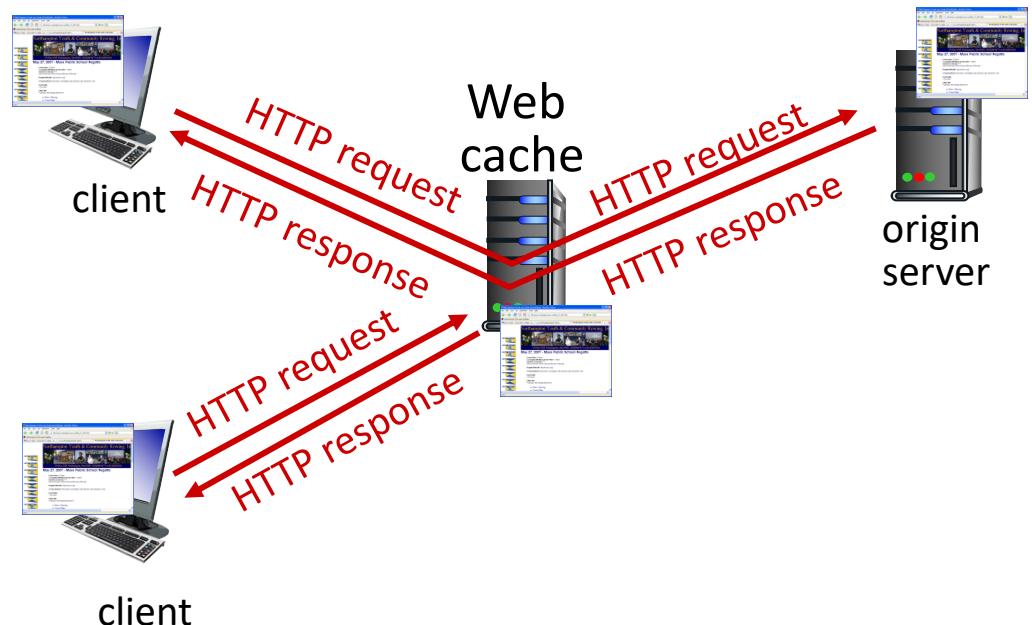
aside  
*cookies and privacy:*

- 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

**Goal:** satisfy client requests without involving origin server

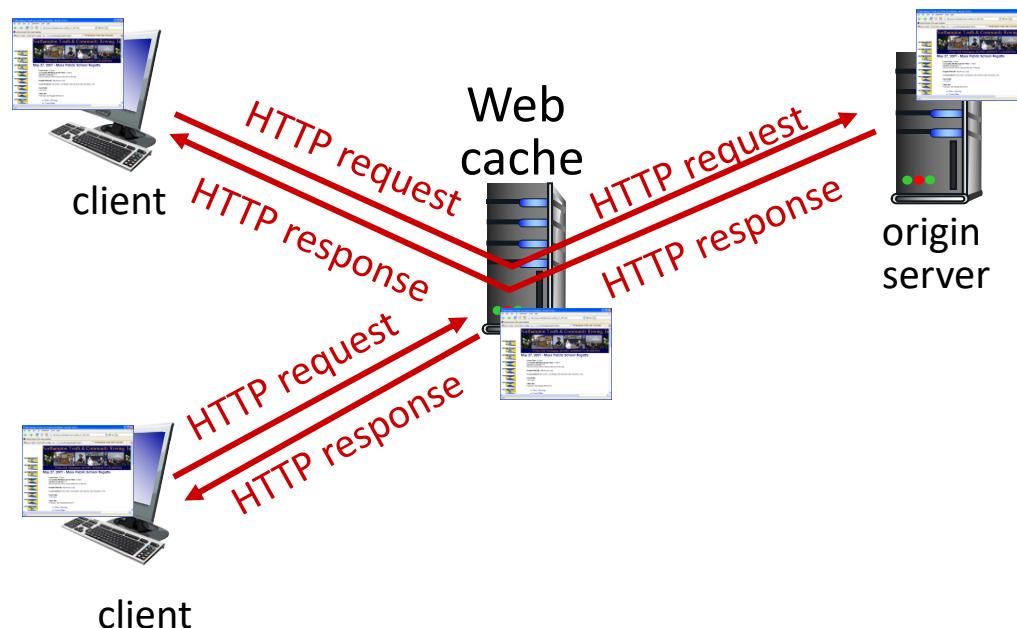
- user configures browser to point to a (local) *Web cache*
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, then returns object to client



# Web Caches

**Goal:** satisfy client requests without involving origin server

- user configures browser to point to a (local) **Web cache**
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, then returns object to client



- Web cache acts as both client and server
  - server for original requesting client
  - client to origin server
- server tells cache about object's allowable caching in response header:

```
Cache-Control: max-age=<seconds>
```

```
Cache-Control: no-cache
```

## *Why* Web caching?

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

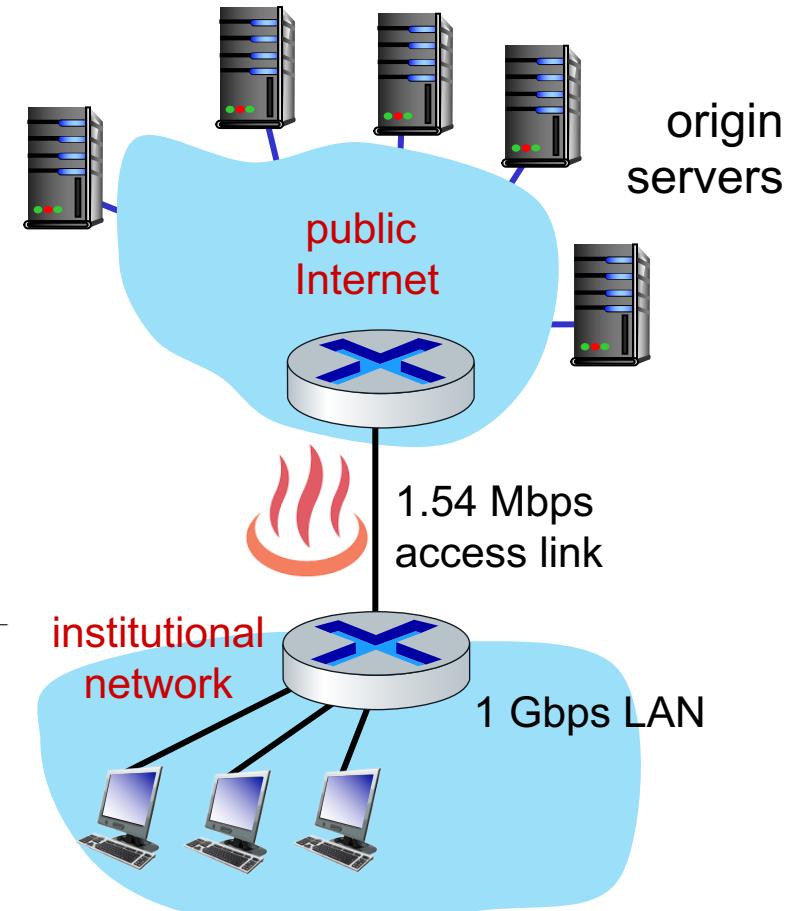
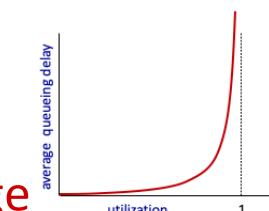
# Caching Example

## Scenario:

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

## Performance:

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



# Caching Example with a Web Cache

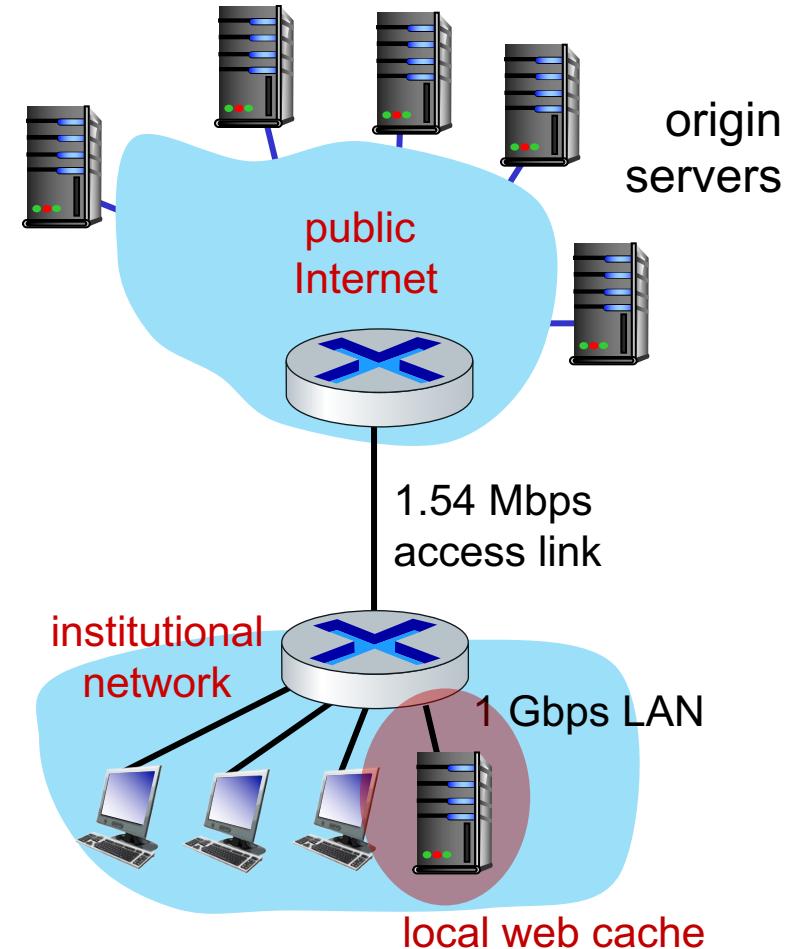
## *Scenario:*

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

*Cost:* web cache (cheap!)

## *Performance:*

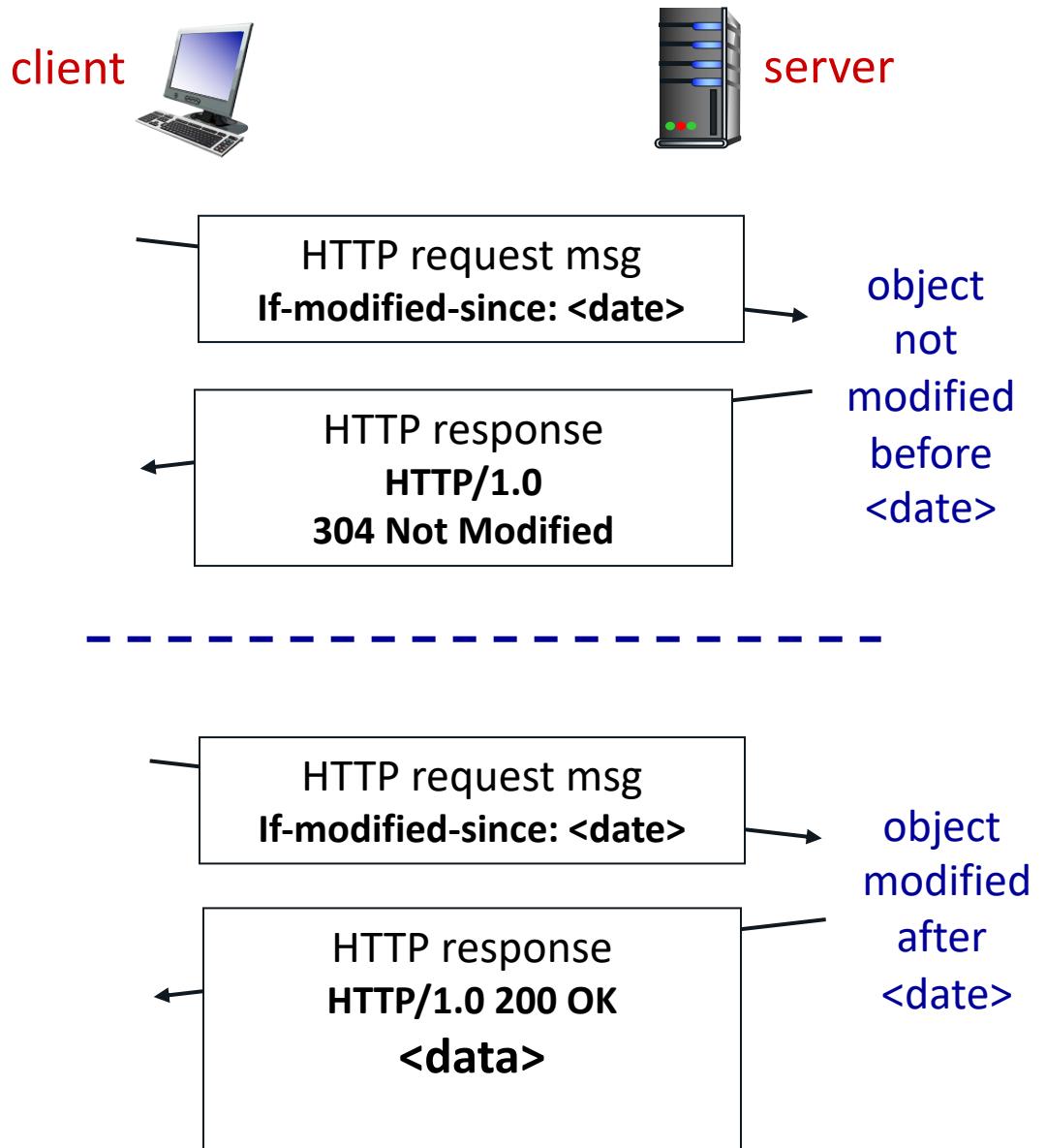
- LAN utilization
- access link utilization
- average end-end delay



# Conditional GET

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

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



*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

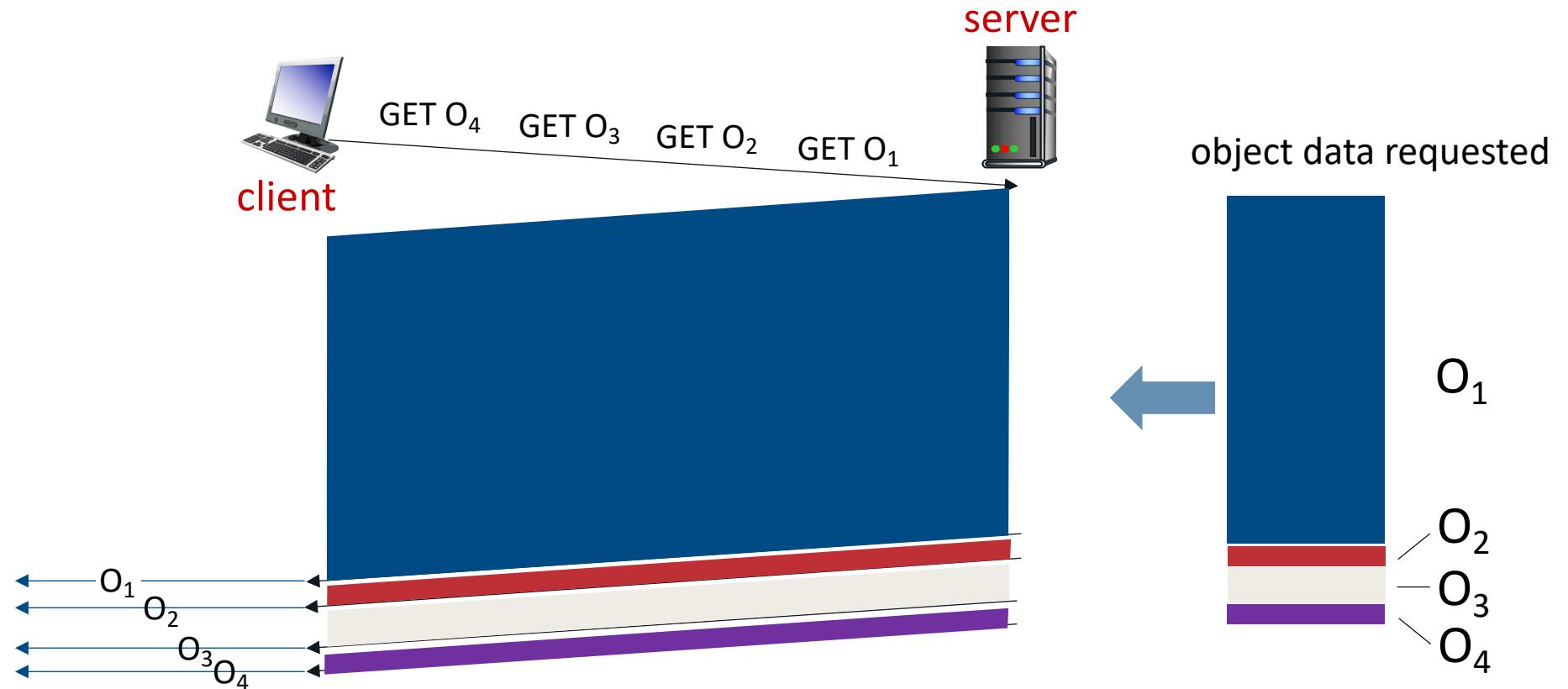
*Key goal:* decreased delay in multi-object HTTP requests

**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

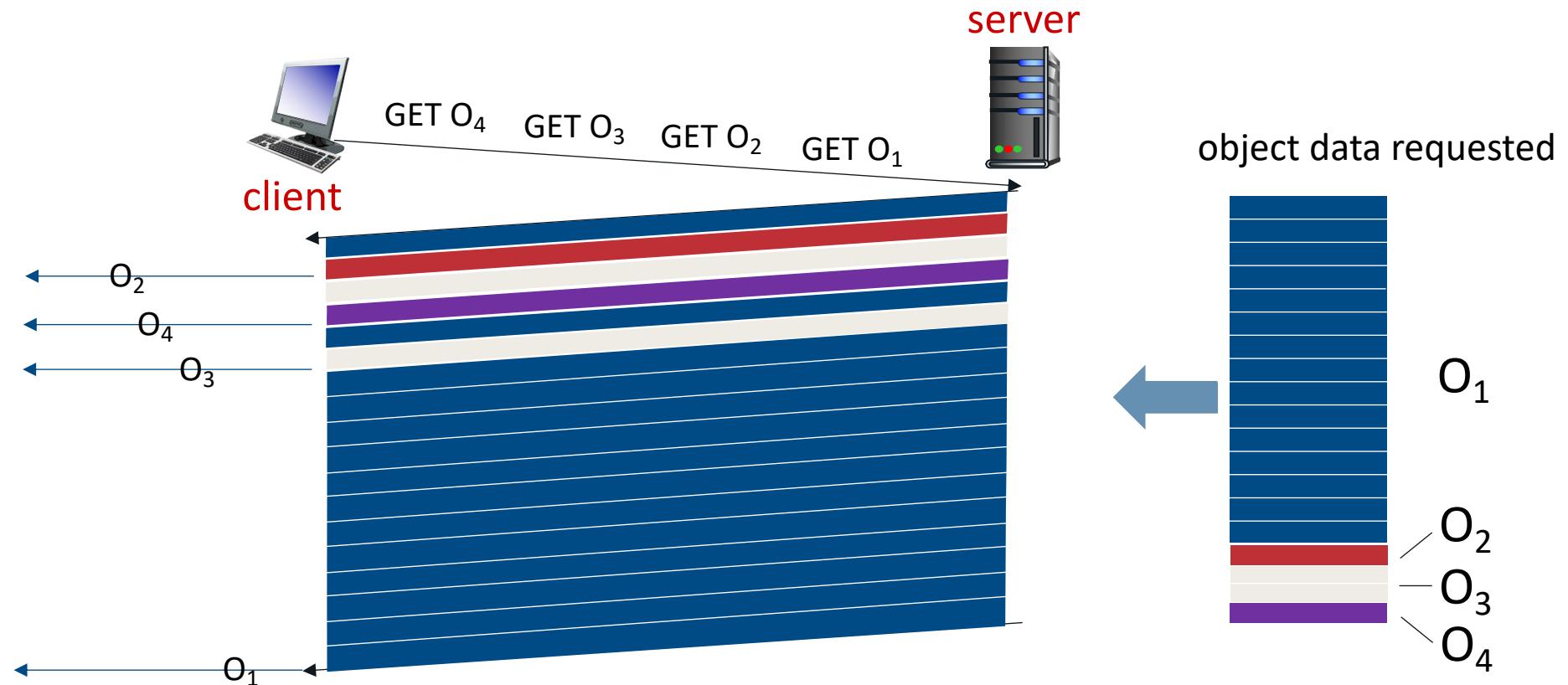
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



*objects delivered in order requested: O<sub>2</sub>, O<sub>3</sub>, O<sub>4</sub> wait behind O<sub>1</sub>*

# HTTP/2: Mitigating HOL Blocking

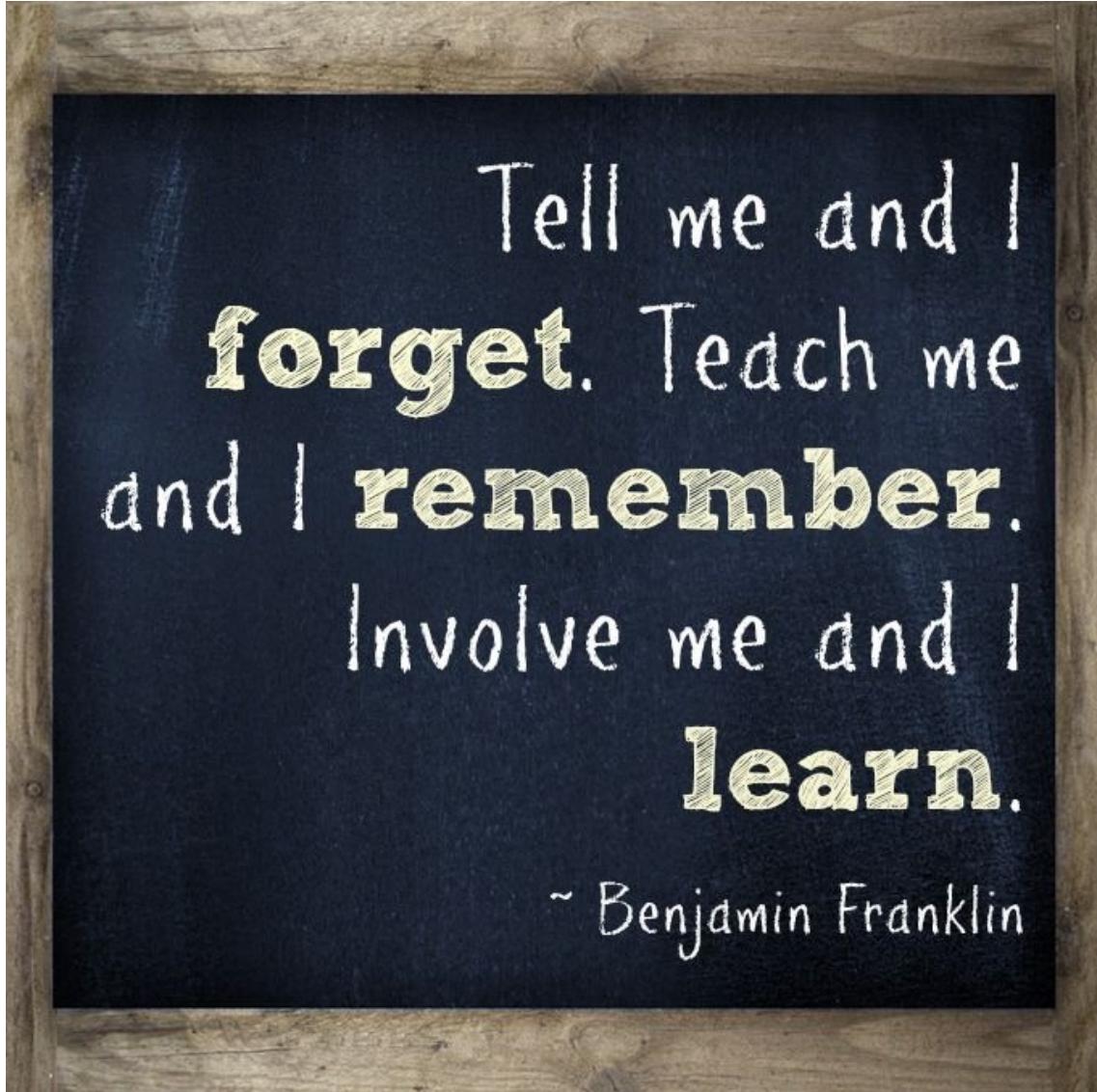
HTTP/2: objects divided into frames, frame transmission interleaved



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

HTTP/2 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



# CSI6201: Networking Technologies

## E-mail

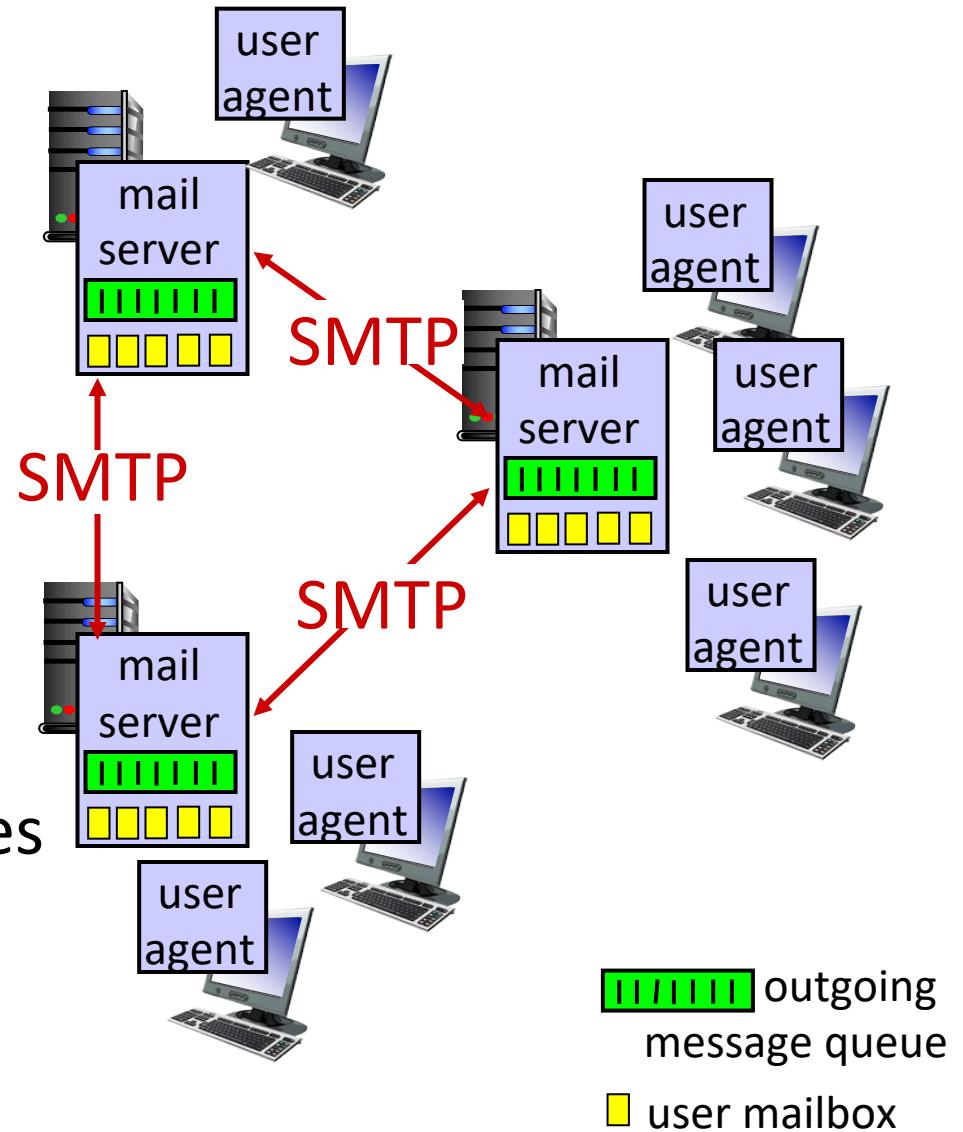


## Three major components:

- 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



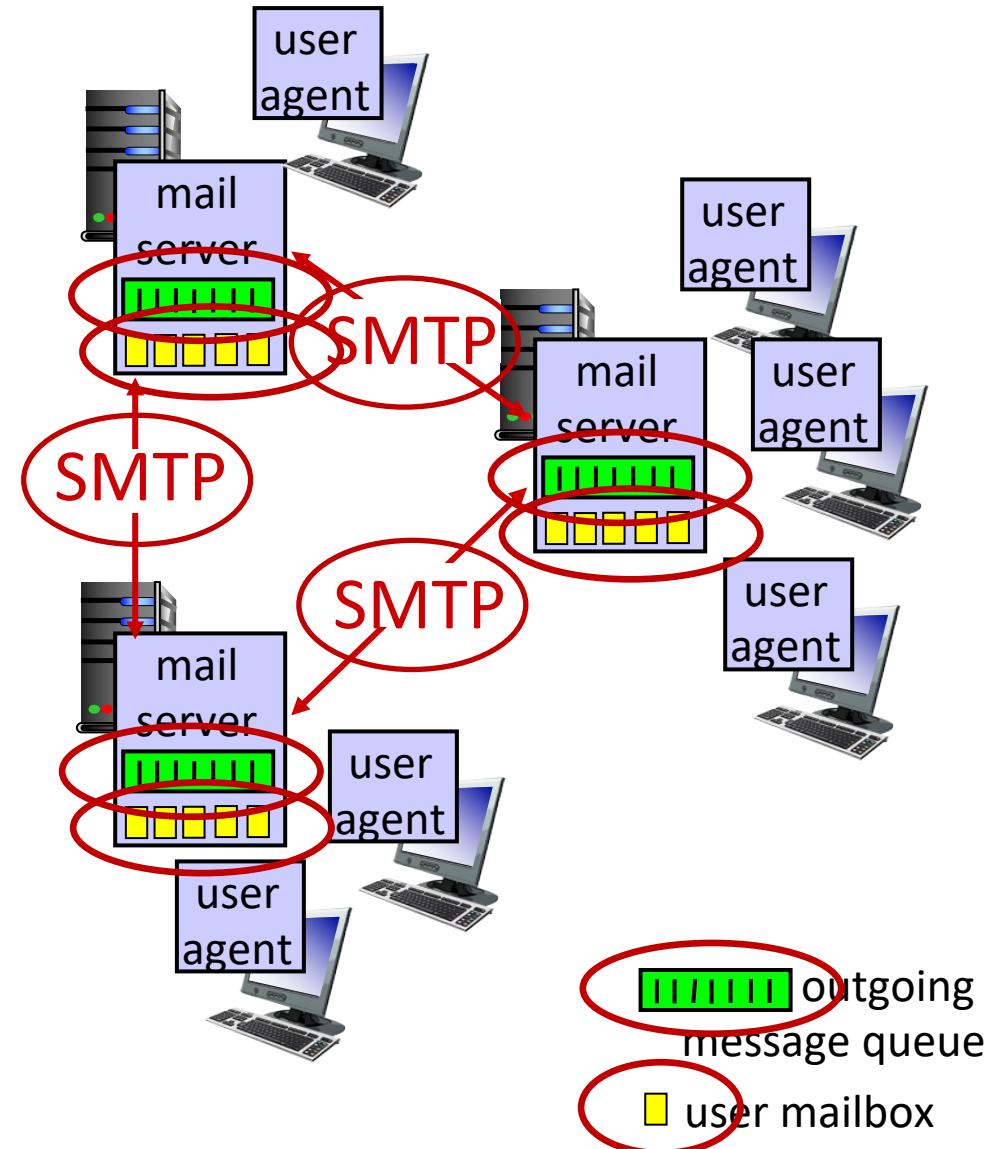
# E-mail: Mail Servers

## mail servers:

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

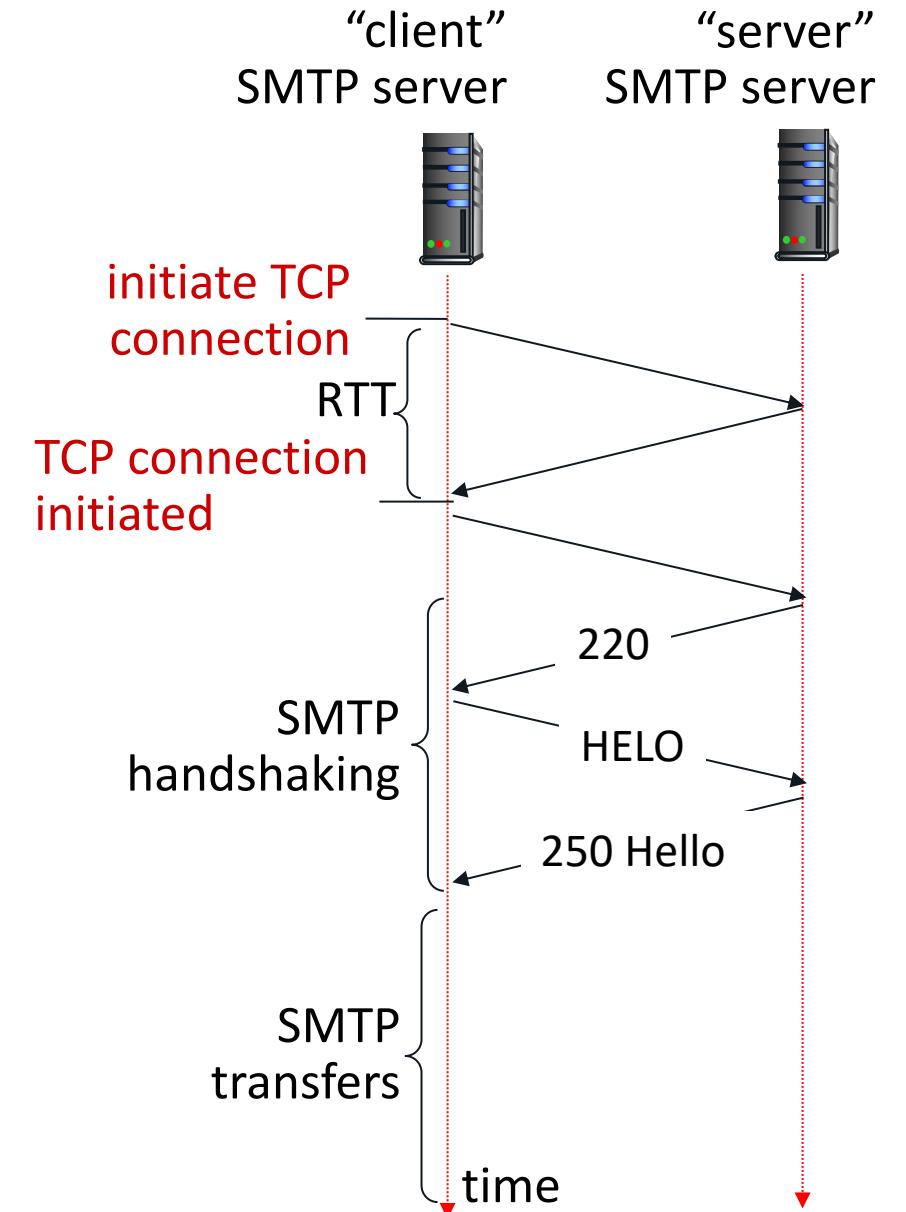
**SMTP protocol** between mail servers to send email messages

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



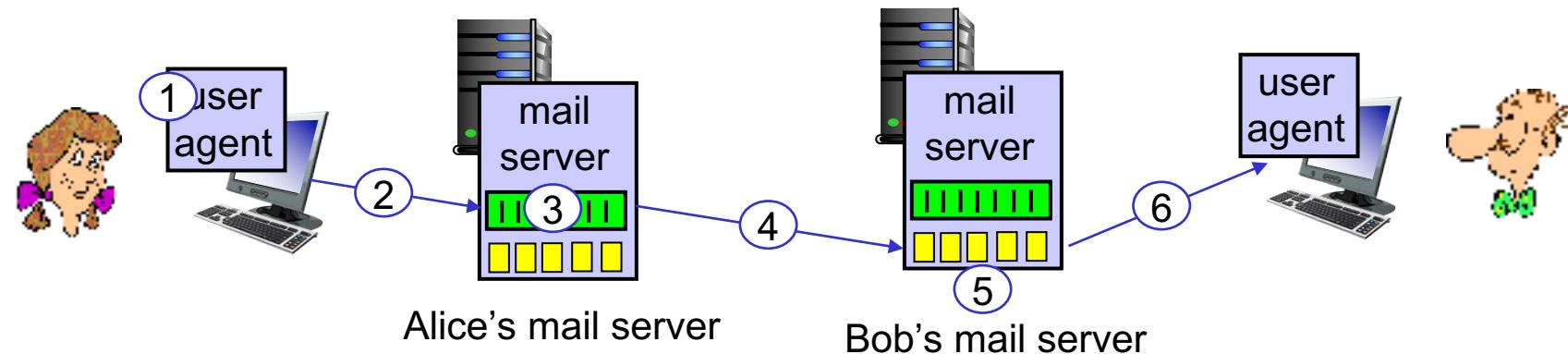
# SMTP RFC

- 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)



# 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 using SMTP; message placed in message queue
- 3) client side of SMTP at mail server 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



## *comparison with HTTP:*

- HTTP: client pull
- SMTP: client 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

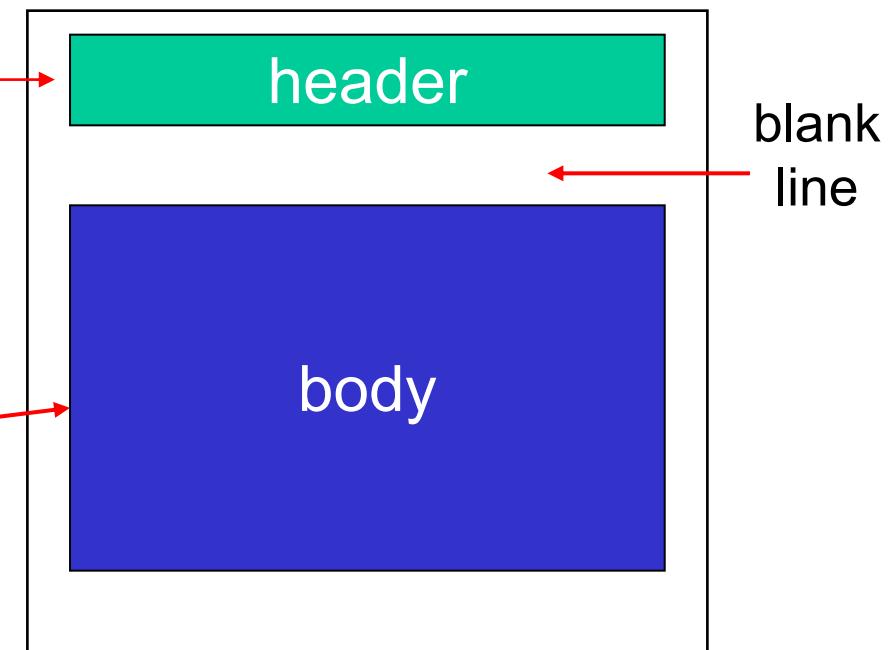
# Mail Message Format

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

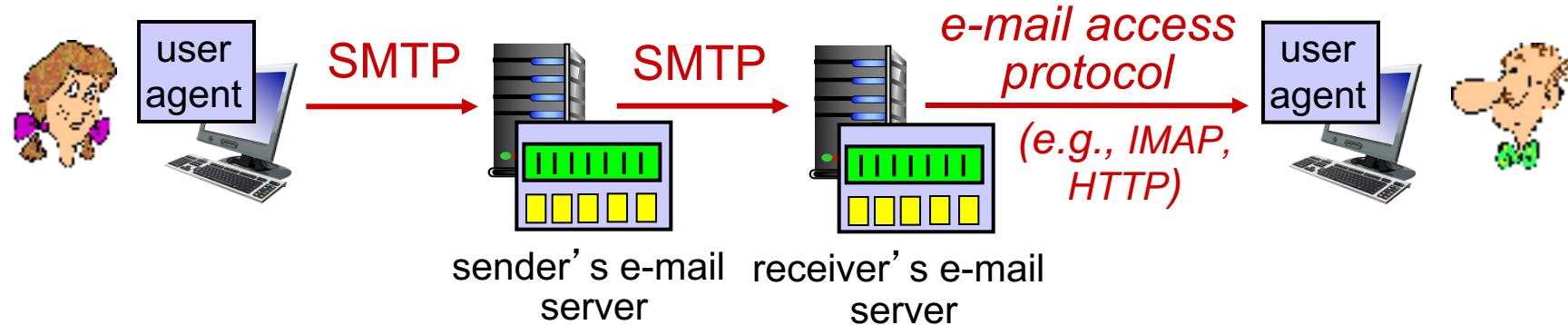
RFC 2822 defines *syntax* for e-mail message itself (like HTML defines syntax for web documents)

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

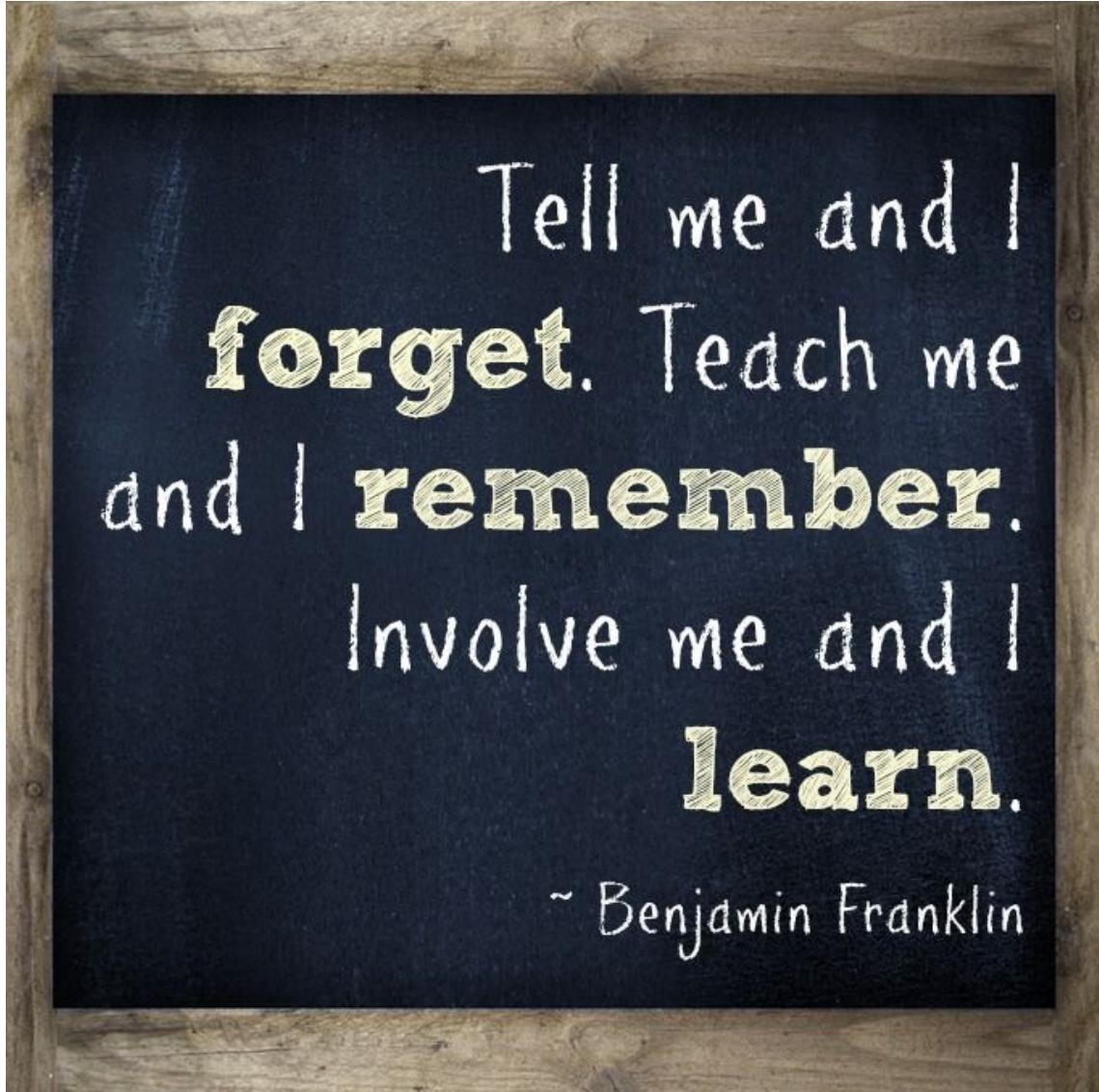
these lines, within the body of the email message area different from ~~SMTP MAIL FROM:,  
RCPT TO: commands!~~
- Body: the “message”, ASCII characters only



# Retrieving E-mail: 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 STMP (to send), IMAP (or POP) to retrieve e-mail messages



## Domain Name System (DNS)



# 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 (DNS):

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

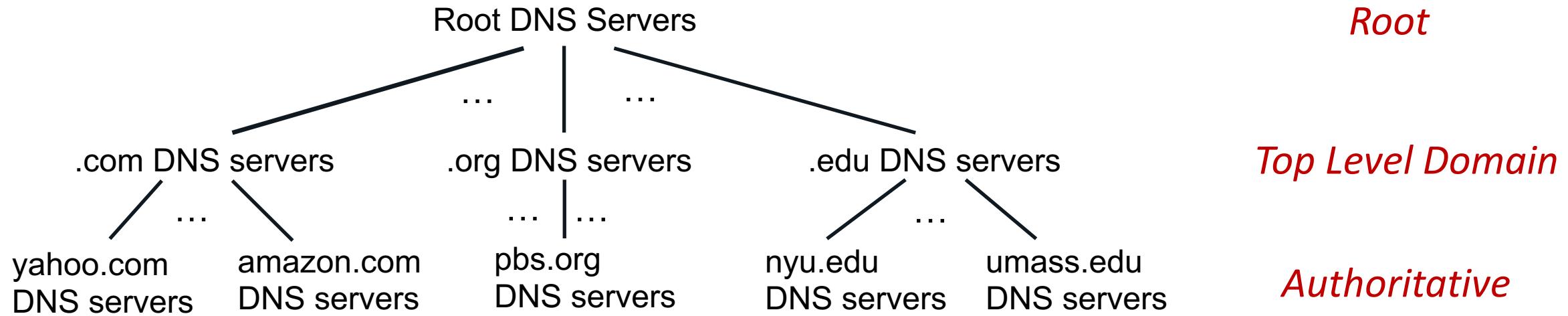
## DNS services:

- 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

# DNS: A Distributed and Hierarchical Database

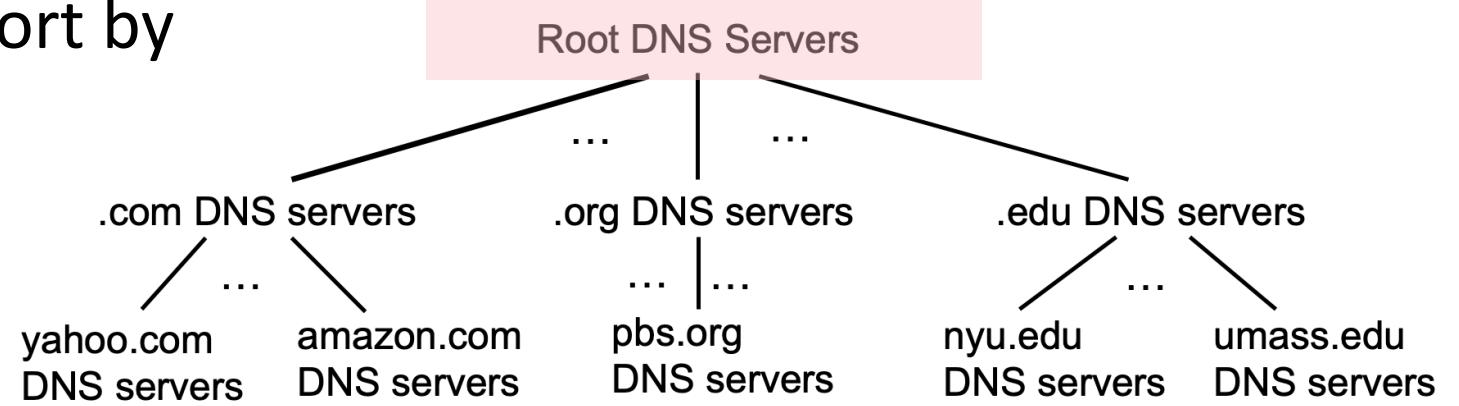


Client wants IP address for [www.amazon.com](http://www.amazon.com); 1<sup>st</sup> approximation:

- 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

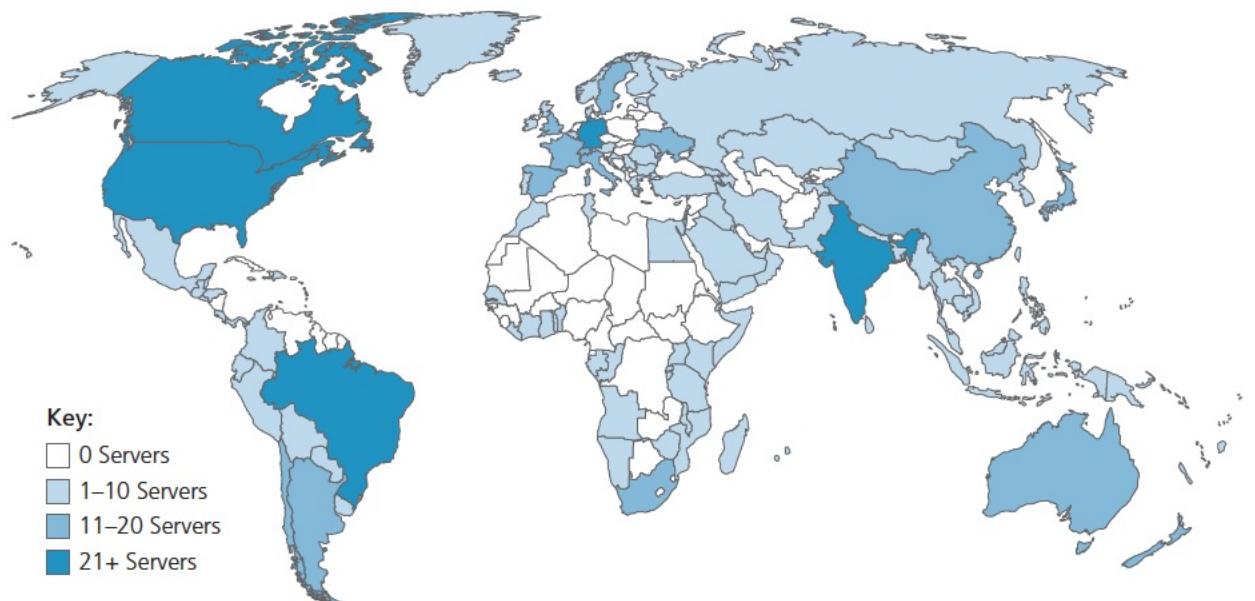
- official, contact-of-last-resort by name servers that can not resolve name



# 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, message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

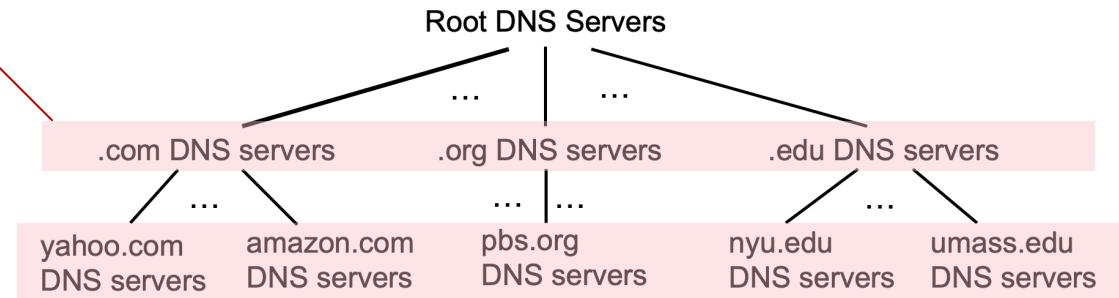
13 logical root name “servers” worldwide each “server” replicated many times



# 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

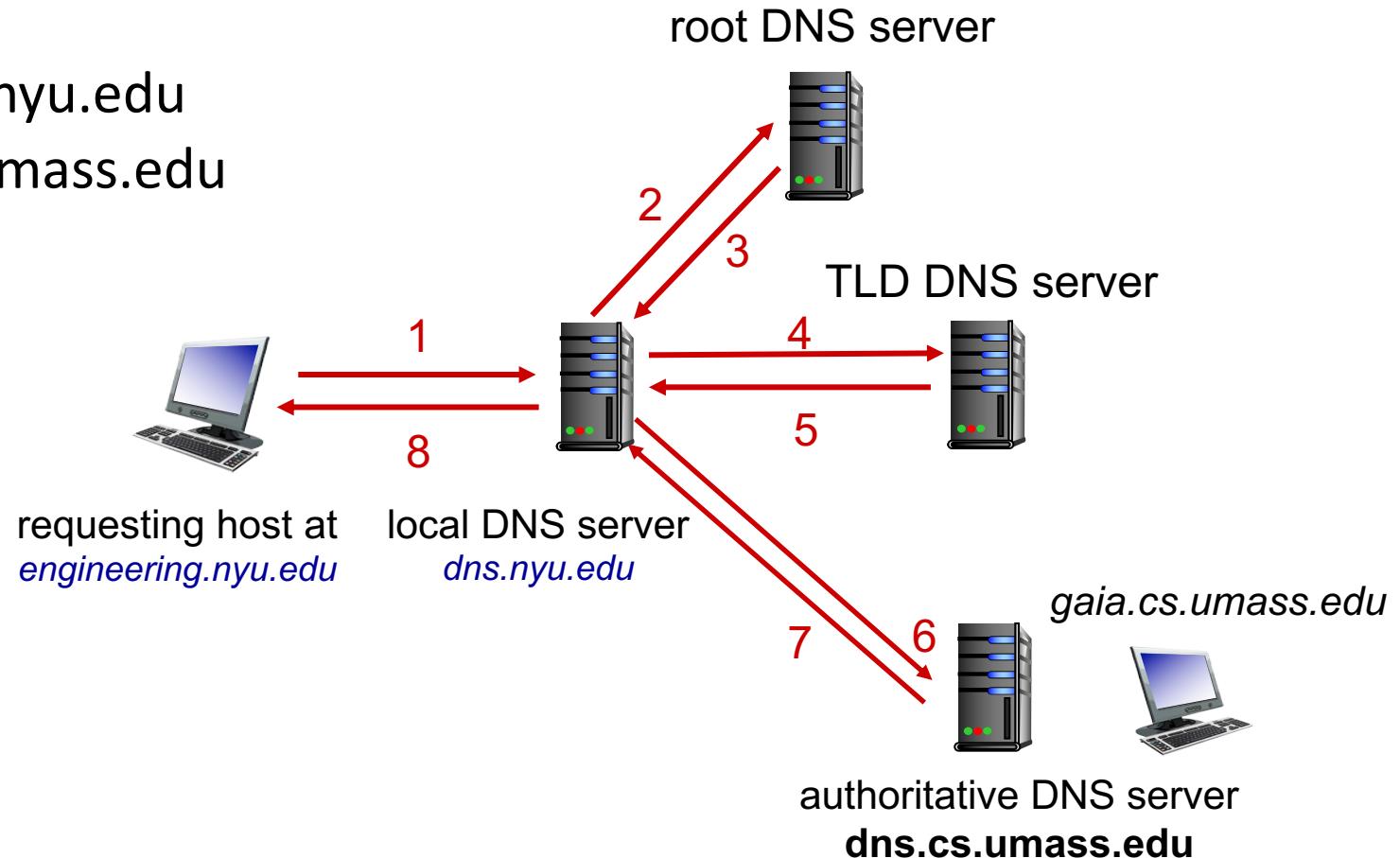
- when host makes DNS query, it is sent to its *local* DNS server
  - Local DNS server returns reply, answering:
    - from its local cache of recent name-to-address translation pairs (possibly out of date!)
    - forwarding request into DNS hierarchy for resolution
  - each ISP has local DNS name server; to find yours:
    - MacOS: % scutil --dns
    - Windows: >ipconfig /all
- local DNS server doesn't strictly belong to 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”

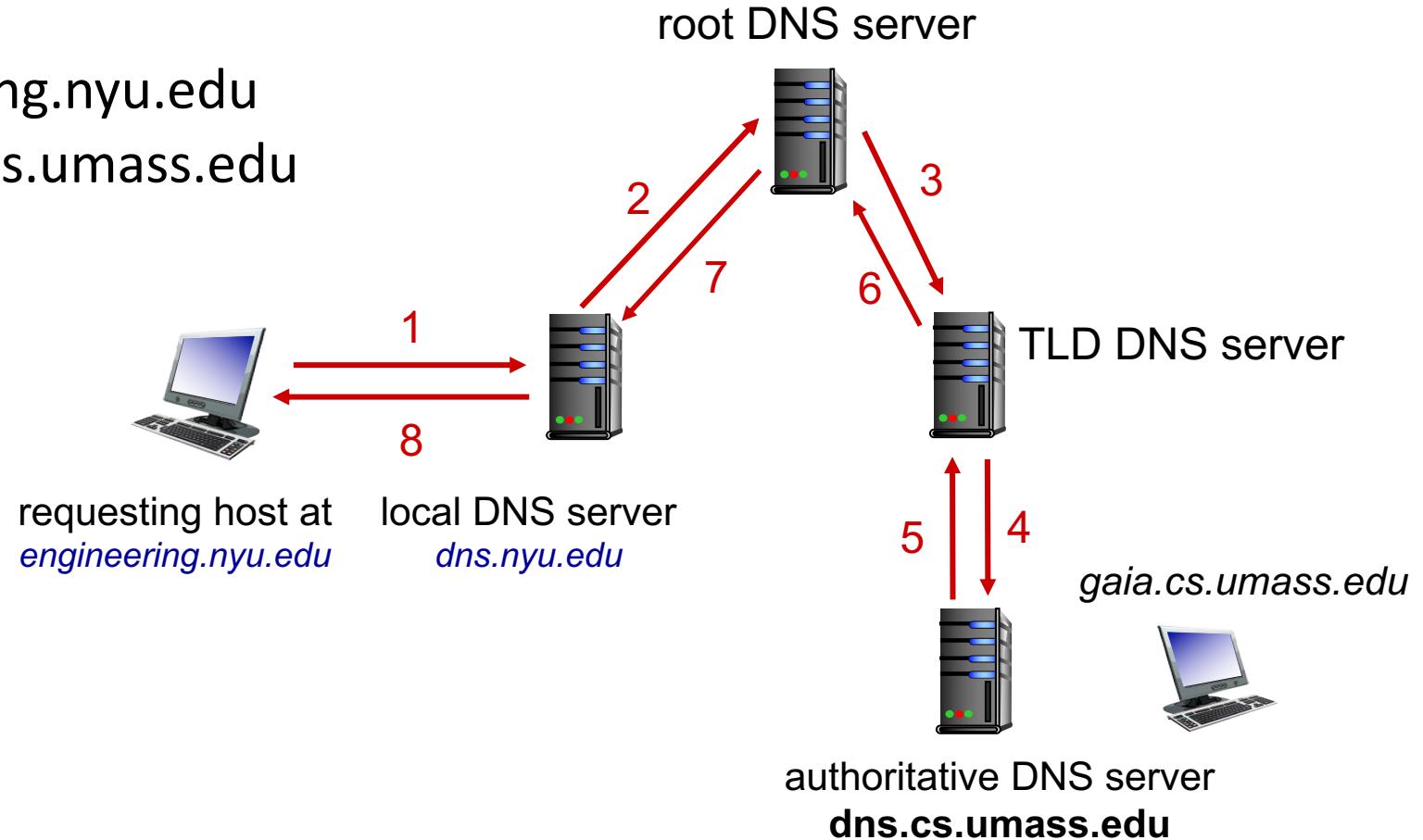


# DNS Name Resolution: Recursive Query

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 DNS Information

- once (any) name server learns mapping, it *caches* mapping, and *immediately* returns a cached mapping in response to a query
  - caching improves response time
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
- cached entries may be *out-of-date*
  - if named host changes IP address, may not be known Internet-wide until all TTLs expire!
  - *best-effort name-to-address translation!*

**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 severeast.backup2.ibm.com
- value is canonical name

## type=MX

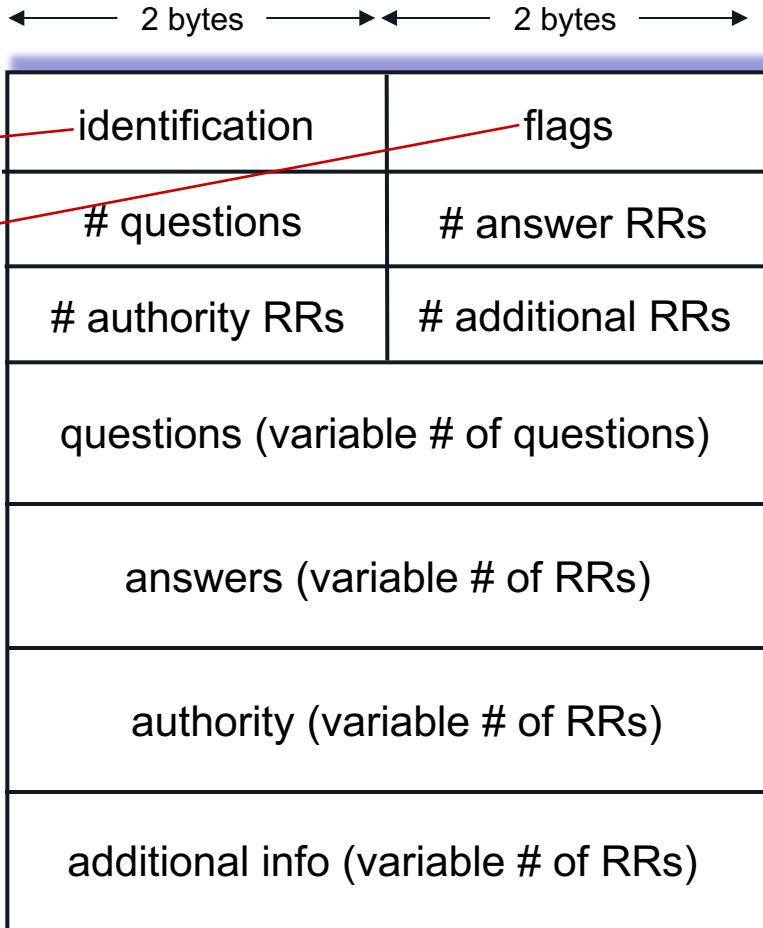
- value is name of SMTP mail server associated with name

# DNS Protocol Messages

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

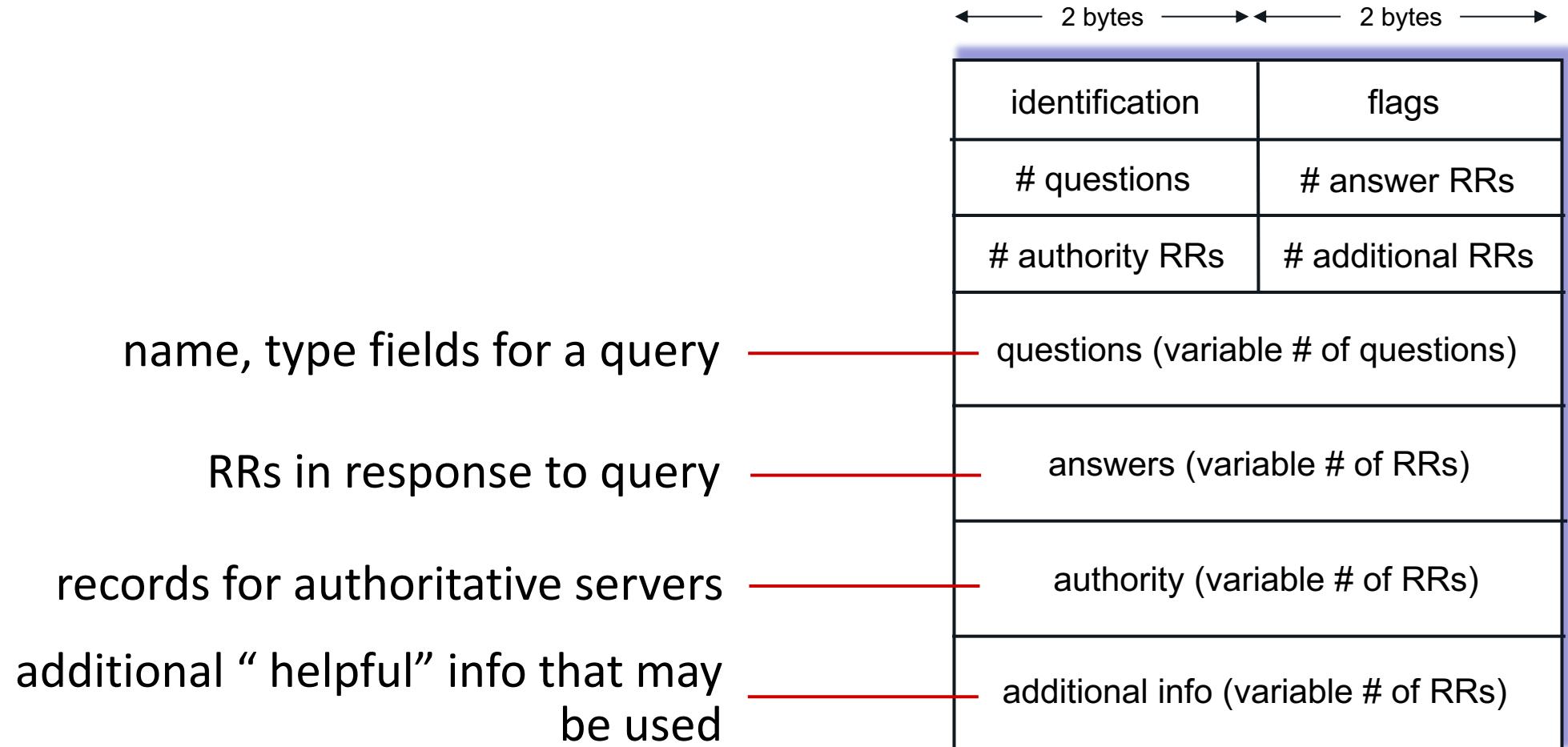
message header:

- **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*:



# Getting Your Info 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 networkutopia.com

