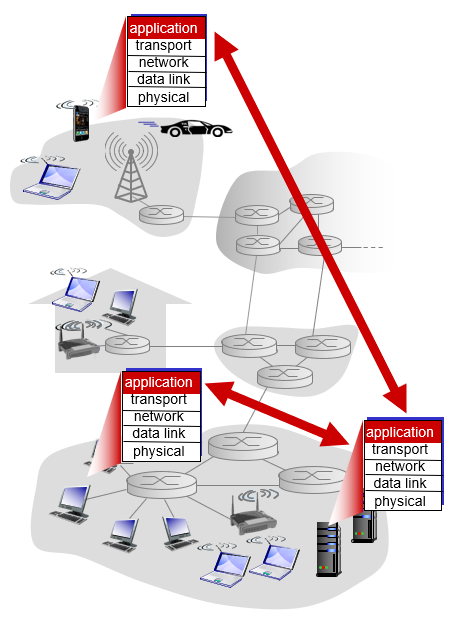
# Application Layer Protocols

Some Network Apps

* E-mail
* Web
* Text messaging
* P2P file sharing
* Massively Multiplayer Online, or MMO, games
* Streaming stored video (YouTube, Hulu, Netflix)
* Voice over IP, or VoIP (e.g. Skype, imo, WhatsApp)
* Real-time video conferencing (e.g. Teams, Discord)
* Social networking (e.g. Discord, Facebook)
* Search (e.g. Google, Bing)

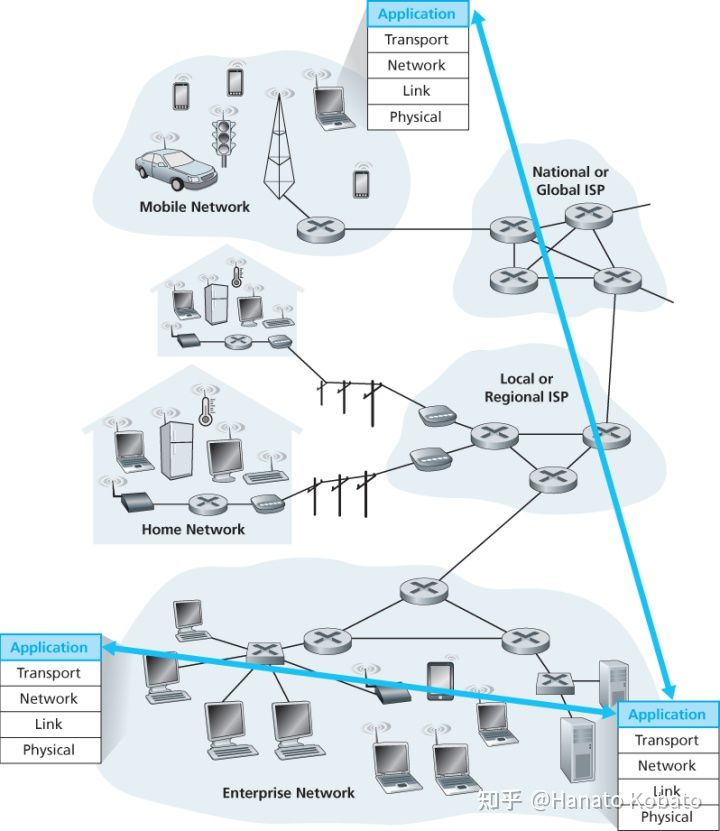
## Creating a Network App

* Write programs that run on different end systems and communicate over network (commonly the Internet); e.g. web server software communicates with browser.
* No need to write software for network-core devices, as they do not run user apps.
* Applications on end systems allows for rapid app development, propagation



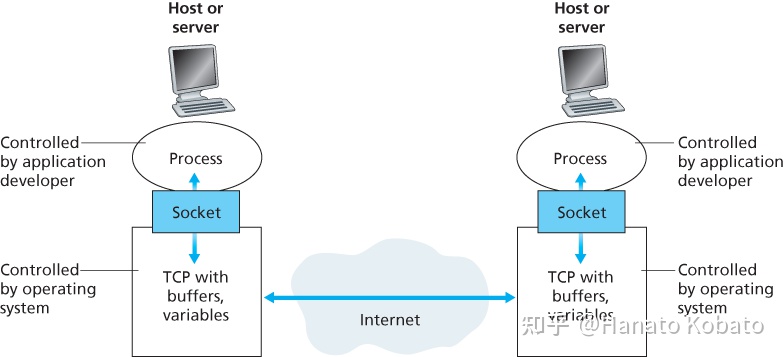
## Processes Communicating

* Most applications consist of pairs of communicating processes.
* 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 and servers use processes to communicate
  + Client process: process that initiates communication
  + Server process: process that waits to be contacted
* Applications with P2P architectures have both client and server processes



## Sockets: Application Interfaces

* Any message sent from one process to another must go through the underlying network, more specifically through a socket.
* A **socket** is the interface between the app layer and the transport layer within a host.
  + It is analogous to a door…
    - sending process pushes the message out the door
    - sending process relies on transport infrastructure (think of it as courier delivery) in between to deliver the message to the socket (or “door”) at receiving process
  + also referred to as the **Application Programming Interface (API)** between the app and the network, since the socket is the programming interface with which network applications are built.

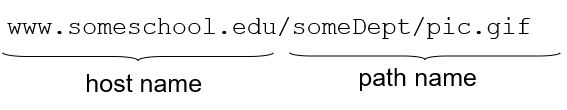


## Addressing Processes

* In order to send postal mail to a particular destination, that destination needs to have an address. Likewise, in order for a process running on one host to send packets to a process running on another host, the receiving process must have an **identifier**
* Host devices have unique 32-bit IP addresses for identification on the Internet.
  + **Q:** Does the IP address of the host (on which the process runs) suffice for identifying the process?
  + **A:** NO, because *many* processes can be running on the same host
* The identifier also needs the **port number** associated with the particular process running on the other host. Popular applications have been assigned specific port numbers.
  + A web server is assigned the port number, 80.
  + A mail server process (using SMTP protocol) is assigned by port number 25.
* For instance, to send an HTTP message to the gaia.cs.umass.edu web server, the following information is required:
  + IP address: 128.119.245.12
  + Port number: 80

## Web and HTTP (A review)

* The Web was first conceived in 1989 by Tim Berners-Lee at the European Particle Physics Lab (CERN) in Geneva.
  + The original idea was to develop a database of info, but that was difficult, so he decided to use a **hypertext** network of information. With hypertext, any doc can contain a link to any other doc.
  + CERN’s first web browser was created in 1990 and released on the Internet in 1991; by the end of 1993, there was the Mosaic browser and about 200 web servers; now, no one knows for sure how many web servers there are!
* The Web is a good example of a [two-tier client server architecture](file:///C:\Users\mmbil\Documents\Saad%20Billah\Massey%20Courses\S2Y1%20Courses\158235%20-%20Networks,%20Security,%20Privacy\Notes\Week%202%20Notes.docx#_Two-tier_architecture).
  + Each client needs an app layer software package called a **web browser**.
  + Each server on the network that will act as a web server needs an app layer software package called a **web server**; example: Microsoft, Apache.
* To get a page from the Web, the user must type the URL for the page he/she wants.
  + The URL specifies the Internet address of the web server and the directory and name of the specific page wanted.
  + If no directory/page is specified, the web server will provide whatever page has been defined as the homepage.
* Web pages consist of the base HTML file which includes several referenced objects
  + Objects can be HTML/CSS files, JPG images, Java applets, audio files, etc.
  + Each object is also addressable by a URL (uniform resource locater), e.g.



## The Web’s Protocol (HTTP)

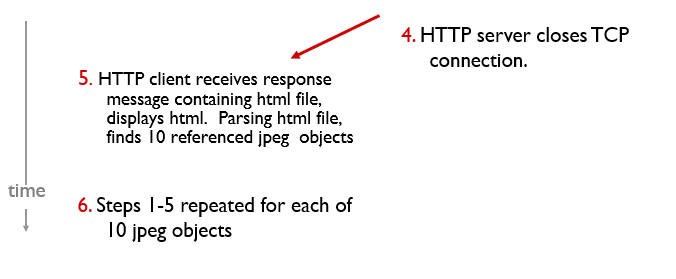
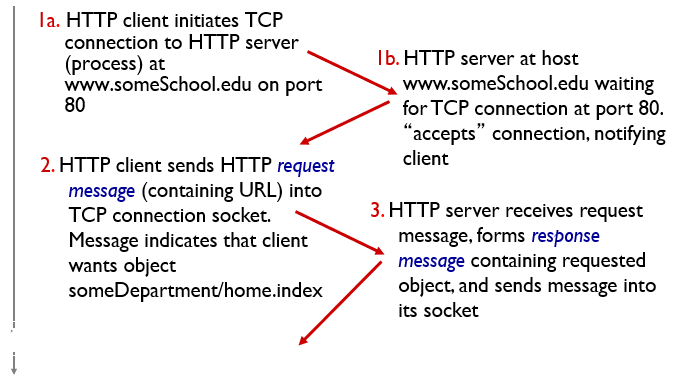
* For the requests from the web browser to be understood by the web server, they must use the same standard **protocol** or language.
* The standard protocol for communication between browser and server is **Hypertext Transfer Protocol (HTTP)**. It is the web’s app layer protocol and is based on a 2-tier client-server model
  + Client: web browser that issues **HTTP requests**, receives web objects (using HTTP protocol) and then “displays” them
  + Server: in response to request, web server sends back an **HTTP response** (using HTTP protocol), which is the requested object or an error message
  + This request-response dialogue occurs for every file transferred between the client and the server (so if there are images associated with the HTML page, they will each need separate requests-responses).
* HTTP uses TCP transfer messages from one process to another…
  + Client initiates TCP connection (creates socket) to server (port 80)
  + Server accepts TCP connection from client
  + HTTP messages (i.e. app-layer protocol messages) are exchanged between browser and web server
  + TCP connection closed
* HTTP is a “stateless” protocol…
  + Server maintains no information about past client requests
* Why is HTTP stateless?
  + Protocols that maintain “state” are complex; past history (state) must be maintained; also, if server/client crashes, their views of “state” may be inconsistent and thus need to be reconciled.

## HTTP connections

* The standard is **non-persistent HTTP**…
  + At most, one object is sent over TCP connection and then the connection is immediately closed.
  + Downloading multiple objects require multiple TCP connections
* Another type is **persistent HTTP**…
  + Multiple objects can be sent over a single TCP connection between the client and server

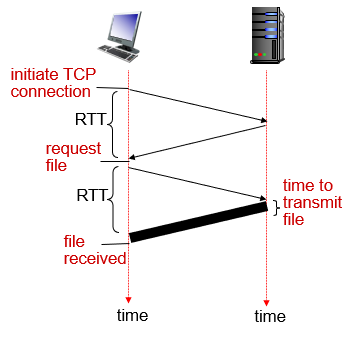
## Non-persistent HTTP

* Suppose someone enters the following URL (which contains text and references to 10 jpeg images): [www.someSchool.edu/someDepartment/home.index](http://www.someSchool.edu/someDepartment/home.index)



## Non-persistent HTTP (response time + issues)

* Round Trip Time (RTT) = time for a small packet to travel from client to server and back
* Non-persistent HTTP response time = 2RTT + file transmission time
  + 1 RTT to initiate TCP connection
  + 1 RTT for HTTP request and 1st few bytes of HTTP response to return
  + File transmission time



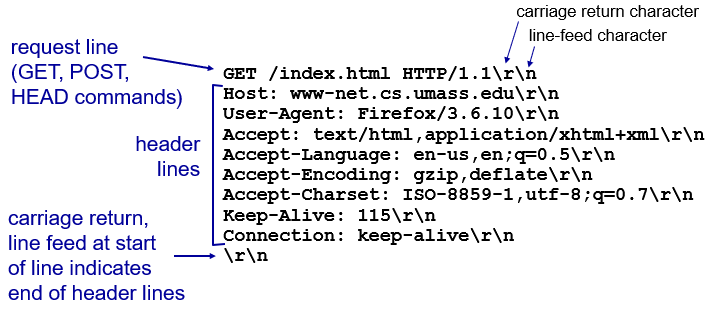
* Non-persistent HTTP issues:
  + Requires 2 RTTs per object
  + OS overhead for *each* TCP connection
  + Browsers often open parallel TCP connections to fetch referenced objects

## Persistent HTTP

* 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 1 RTT for all the referenced objects

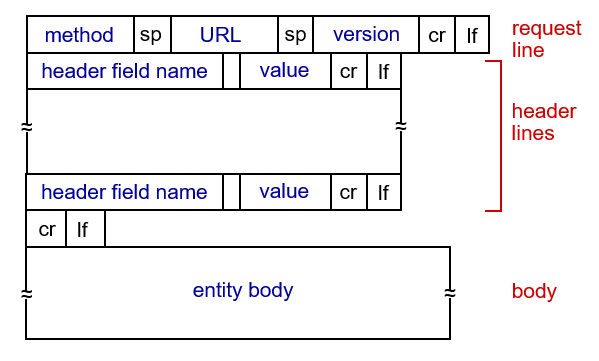
## HTTP request message

* The HTTP request (and HTTP response) are…
  + examples of the packets (introduced in [Week 1](Week%201%20Notes.docx)) that are produced by the app layer and sent down to the other layers for transmission through the network.
  + are simple text files that take the info provided by the app (e.g. the URL to get) and format it in a structured way (i.e. in ASCII format) so that the receiver of the message can clearly understand it.
* Notes:
  + Carriage return character = returns device position to the beginning of a line of text
  + Line-feed character = starts a new line of text; same as pressing ENTER



## HTTP request message: general format

* An HTTP request from a web browser to a web server has three parts; first and third parts are required; second part is optional…
  + The **request line**, which starts with a command (e.g. GET), provides the web page, and ends with the HTTP version number that the browser understands (so that there is no confusion between different versions)
  + The **request header**, which contains a variety of optional info; e.g. web browser being used, date, and so on.
  + The **request body**, which contains info sent to the server, such as info that user has typed into a form.



## Uploading form input

* POST method…
  + Web page often includes form input
  + Input is uploaded to server in entity body
* URL method…
  + Uses GET method
  + Input is uploaded in URL field of request line:

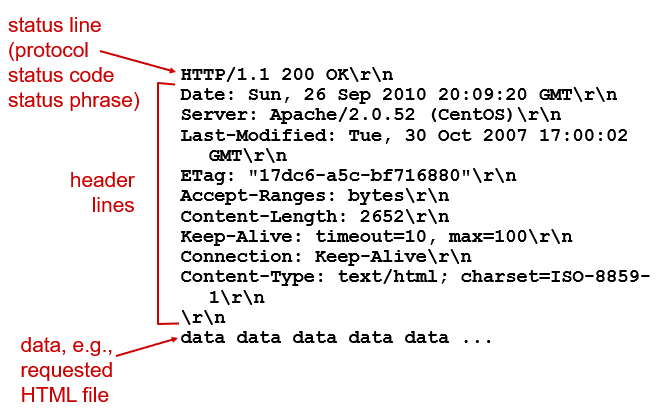


## Method Types

* HTTP/1.0 has the following methods…
  + GET: for requesting a webpage
  + POST: for submitting form input
  + HEAD: asks for a response identical to GET, but without the response body
* HTTP/1.1 has these additional methods…
  + PUT: uploads file in entity body to path specified in URL field (or replaces all current representations of the target resource with the request payload)
  + DELETE: deletes file specified in the URL field

## HTTP response message

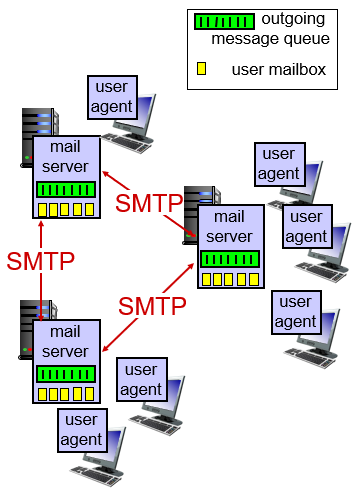
* The format of an HTTP response is very similar to the HTTP request; it too has three parts, with the first required and the last two optional:
  + The **response status**, which contains the HTTP version number the server has used, a *status code* (e.g. 200 means “OK”, 404 means “not found”, and a reason phrase (a text description of status code).
  + The **response header**, which contains a variety of optional info, e.g. web server being used (like Apache), the date, and exact URL of the page.
  + The **response body**, which is the webpage itself

****

## HTTP response status codes

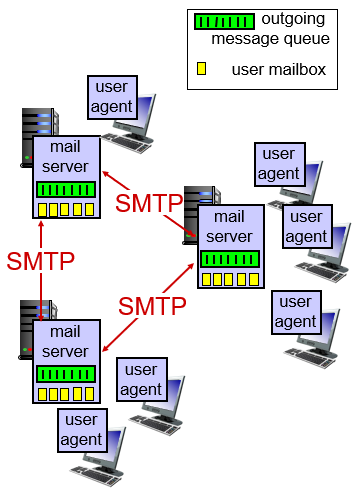
* Status code appears in 1st line in the HTTP response message
* Some sample codes:
  + 200 – OK: request succeeded, requested object later in this msg
  + 301 – Moved Permanently: requested object moved, new location specified later in the message (Location: …)
  + 400 – Bad Request: request msg not understood by server
  + 404 – Not Found: requested doc not found on this server
  + 505 – HTTP Version Not Supported

## Electronic Mail

* With **electronic mail** (or **email**), users create and send messages to one user, several users or all users on a **distribution list**.
  + Email was one of the earliest applications on the internet and is still heavily used today for communication.
  + Most email software enables users to send text messages, attach files from different sources, and filter messages.
* Several ‘emailing’ standards have been developed to ensure compatibility between different email software packages.
  + A software package conforming to a certain standard can send messages that are formatted using its rules.
  + Another package that understands that particular standard can then relay the message to its destination. But if it receives the message in an unknown format, it may be unable to process correctly.
  + Many email packages send using one standard but can understand messages sent in several different standards.
* The most common standard is SMTP (Simple Mail Transfer Protocol).
  + Other common standards include CMC (Common Messaging Calls) and X.400.
  + All these standards can be translated to the standard that the company uses using translation software.

## How Email Works

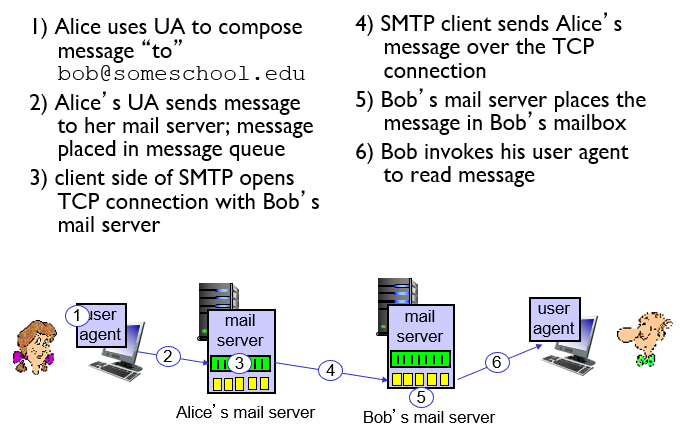
* Email works similar to how the Web works, but is a bit more complex.
* SMTP usually implemented as a 2-tier thick client-server architecture app.
* Three major components (user agents, SMTP, mail servers)…
  + Each client runs an app layer software package called a **mail user agent (**a.k.a. an **email client** or **mail reader**), which…
    - Allows users to create, read, and edit email messages
    - Formats composed message into an SMTP packet with info like sender’s and destinations addresses
    - Sends outgoing messages for storage on a **mail server**
    - Examples of email clients are Outlook and Thunderbird.
  + The mail server runs a special app layer software package called a **mail transfer agent**, a.k.a. **mail server software**, which…
    - Reads the SMTP packet to find destination address.
    - Sends the packet on its way through the network from mail server to mail server until it reaches the destination
  + The mail transfer agent on the destination server then stores the message in the receiver’s mailbox on that server, where it sits waiting to be checked



## How SMTP Works

* Uses TCP to reliably transfer email message from client to mail server (port 25)
  + Direct transfer: sending server to receiving server
  + Three phases of transfer:
    - Handshaking (greeting)
    - Transfer of messages
    - Closure
  + Command/response interaction (like HTTP, FTP)
    - Commands: ASCII text
    - Response: status code and phrase
  + Messages must be in 7-bit ASCII

## Example: Alice sends email to Bob



## Sample SMTP interaction

* C = client
* S = server
* HELO = hello message; greeting
* RCPT TO = receipt to; send to
* DATA = this is the data being sent
* QUIT = quitting the connection

**S: 220 hamburger.edu**

**C: HELO crepes.fr (meaning “this is 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**

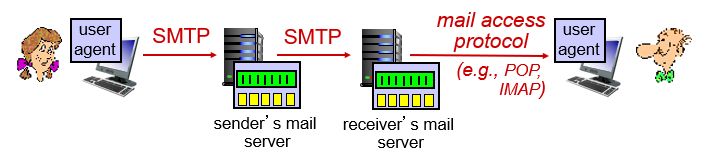
## SMTP: final words

* 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

*comparison with HTTP:*

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

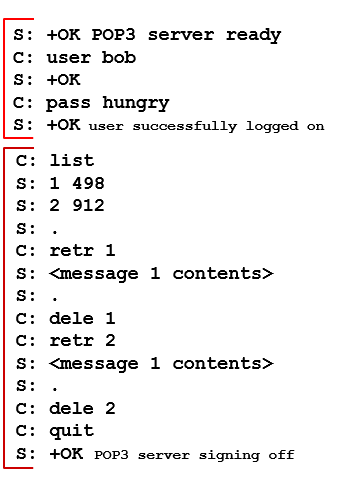
## Mail access protocols



* SMTP: delivery/storage to receiver’s server
* mail access protocol: retrieval from server
  + POP: Post Office Protocol [RFC 1939]: authorization, download
  + IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  + HTTP: gmail, Hotmail, Yahoo! Mail, etc.

## POP3 protocol

***authorization phase***

* client commands:
  + **user:** declare username
  + **pass:** password
* server responses
  + **+OK**
  + **-ERR**

***transaction phase****,* client:

* **list:** list message numbers
* **retr:** retrieve message by number
* **dele:** delete
* **quit**

## POP3 (more)

*more about POP3*

* previous example uses POP3 “download and delete” mode
  + Bob cannot re-read e-mail if he changes client
* POP3 “download-and-keep”: copies of messages on different clients
* POP3 is stateless across sessions

## IMAP

* keeps all messages in one place: at server
* allows user to organize messages in folders
* keeps user state across sessions:
  + names of folders and mappings between message IDs and folder name

## DNS: domain name system

* *people* have many identifiers: SSN, name, passport number, etc
* *Internet hosts, routers* only have*:*
  + IP address (32 bit) - used for addressing datagrams
  + “name”, e.g., www.yahoo.com - used by humans
* *Question:* how to map between IP address and name, and vice versa?
* *Answer:* use the Domain Name System (DNS)!
  + *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: e.g. replicated Web servers: many IP addr correspond to one name

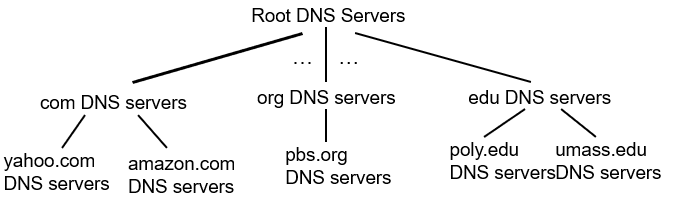
*Question:* why not centralize DNS?

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

*Answer:* it just doesn’t scale!

## DNS: a distributed, hierarchical database

*client wants IP for www.amazon.com; 1st approx:*

* 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