# Application Layer

- Reference
  - Chapter 02, *Computer Networking: A Top Down Approach,* 6/E, Jim Kurose, Keith Ross, Addison-Wesley
  - Adapted from part of the slides provided by the authors



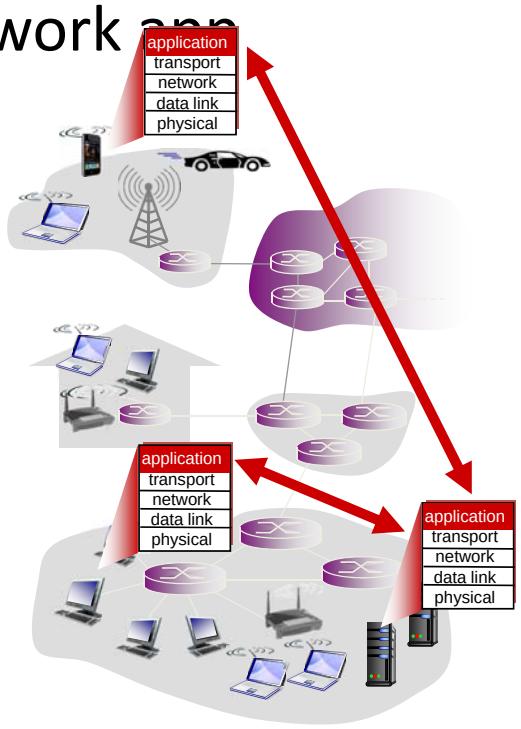
# Some network apps

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

# Creating a network app

write programs that:

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software



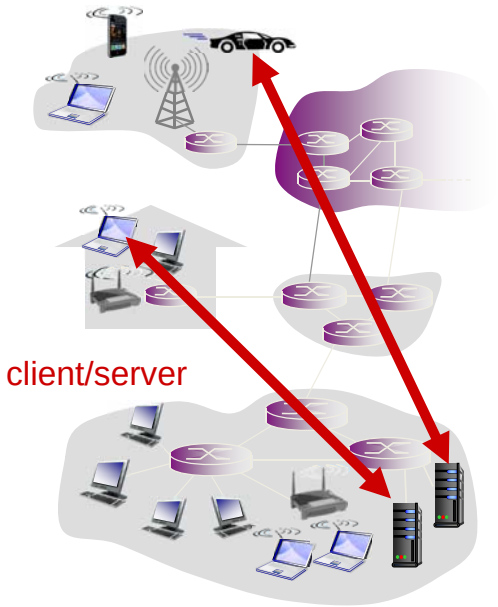

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



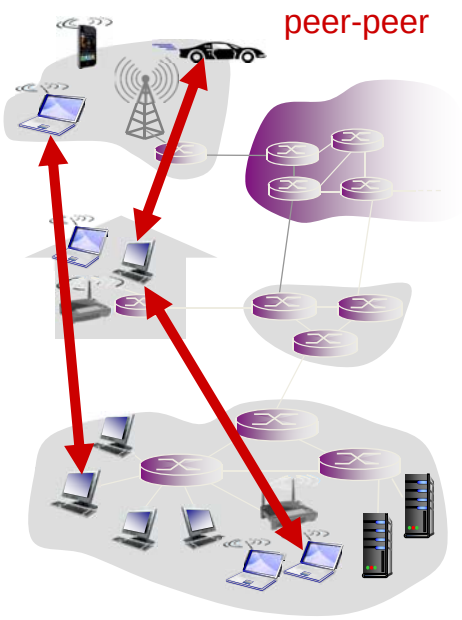
# Client-server architecture

- server:
  - always-on host
  - permanent IP address
  - data centers for scaling
- clients:
  - communicate with server
  - may be intermittently connected
  - may have dynamic IP addresses
  - do NOT communicate directly with each other



# P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management

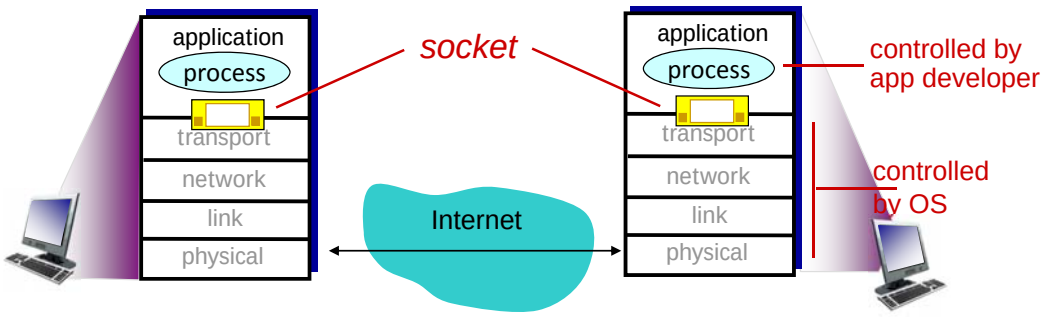





# Processes

- *process:* program running within a host
  - Message: the unit/format of data at the application layer
  - processes in different hosts communicate by exchanging messages
- Clients and servers
  - *client process:* process that initiates communication
  - *server process:* process that waits to be contacted
- applications with P2P architectures have client processes & server processes

# Sockets




- Socket: the service interface provided by the transport layer to the application layer on a host (client or server)
- process sends/receives messages to/from its socket
  - A process is addressed by an identifier including both IP address (to address host) and port number (to address a specific process on a host)
- 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



# 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,
- *SSL service:*
  - provides encrypted TCP connection
  - data integrity
  - end-point authentication
  - SSL is on top of TCP

# Socket programming in Java

- *TCP:*
  - Socket class used on server side: class ServerSocket
  - Socket class used on client side: class Socket
- *UDP*: Socket class used on both client and server side: DatagramSocket

*Application Example:*

1. Client reads a line of characters (data) from its keyboard, displays the data on its screen, and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.
5. Repeat Steps 1 – 4 until "Bye." is read by the client.

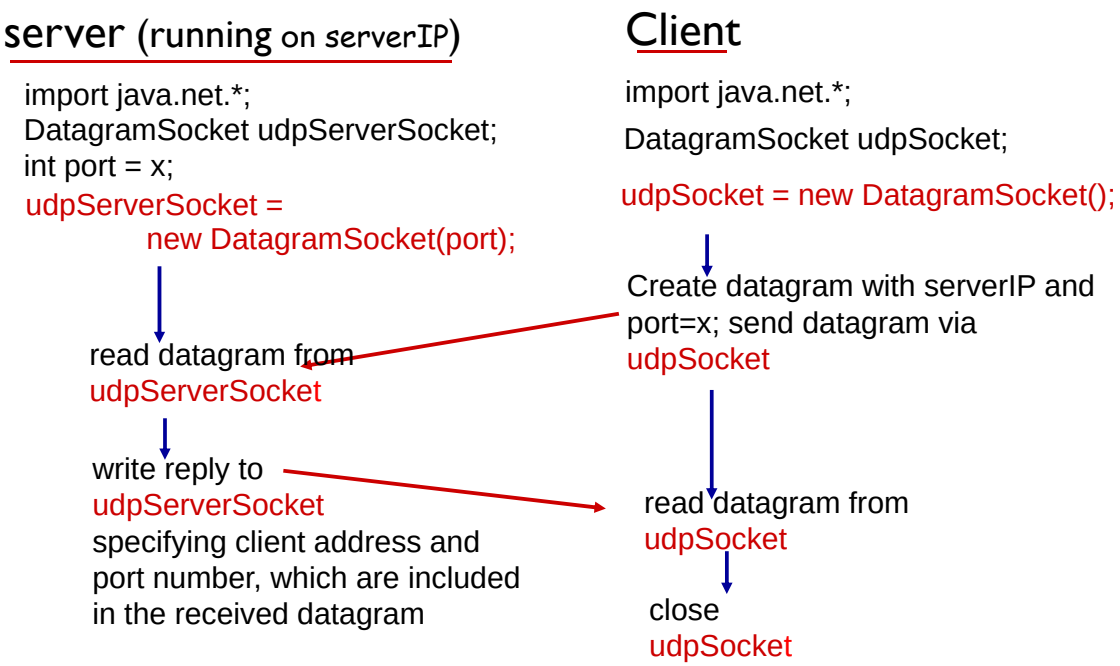


# Socket programming *with UDP*

- Application viewpoint:
  - UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server
- UDP: no "connection" between client & server
  - no handshaking before sending data
  - sender explicitly attaches IP destination address and port # to each packet
  - rcvr extracts sender IP address and port# from received packet
- UDP: transmitted data may be lost or received out-of-order

➔ See source codes after reading the next slide:
- UDP_ToUpperCase/UDPClient.java
- UDP_ToUpperCase/UDPServer.java

# Client/server socket interaction: UDP



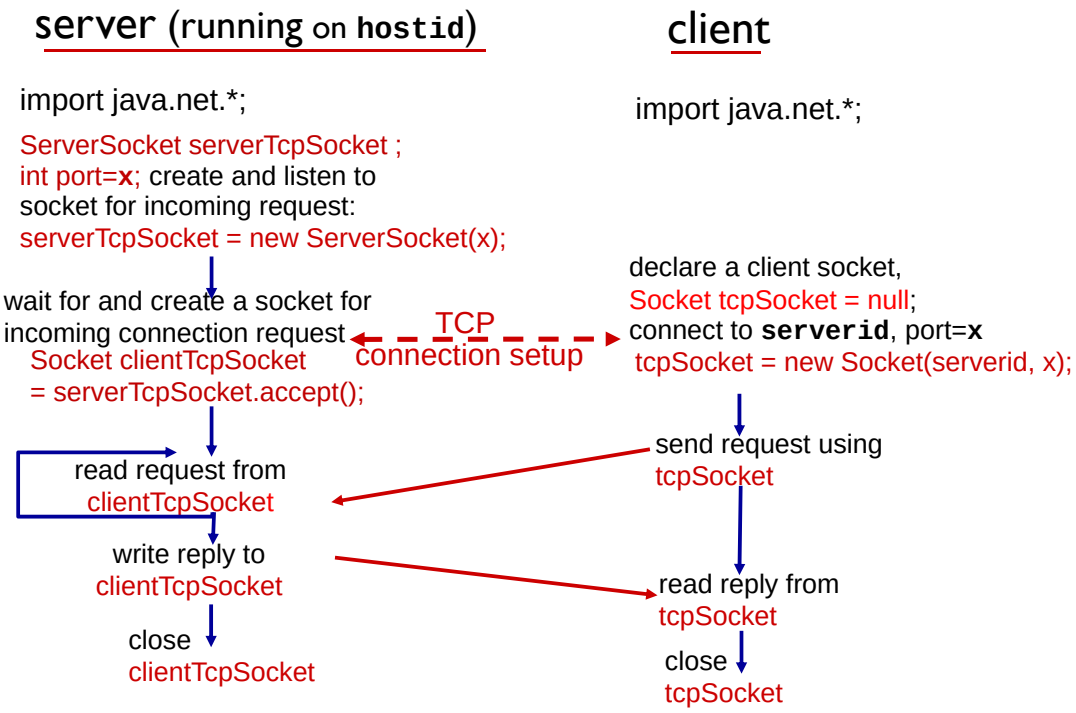



# Socket programming *with TCP*

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

➔ See source codes after reading the next slide:
- TCP_ToUpperCase/TCPClient.java
- TCP_ToUpperCase/TCPServer.java

# Client/server socket interaction: TCP






# TCP Server using Multiple Threads

- Multiple threads are used by TCP server to support multiple clients
- Upon accepting a connection request from a client, TCP Server starts a new thread designated to handle the TCP connection with this client.

  new TCPMultiServerThread(serverTCPSocket.accept()).start();
- One of the two ways to start a new thread in Java
  - Extend the Thread class:

    public class TCPMultiServerThread extends Thread { … }
    - TCPMultiServerThread: subclass or extending class, Thread: supper class
  - The extending class must override the run() method of the Thread class, which is the entry point for the new thread

    public void run() { … }
  - To start a new thread, create an instance of the extending class and call start()

    new TCPMultiServerThread(serverTCPSocket.accept()).start();
  - This new thread stops at the end of the run() method
  - Each thread may have a name, in the constructor of the extending class,

    super("TCPMultiServerThread")

➔ See TCP_toUpperCase_MultiThread/*.java

# Internet apps: application, transport protocols

| application | application layer protocol | underlying transport protocol |
|---|---|---|
| e-mail | SMTP [RFC 2821] | TCP |
| remote terminal access | Telnet [RFC 854] | TCP |
| Web | HTTP [RFC 2616] | TCP |
| file transfer | FTP [RFC 959] | TCP |
| streaming multimedia | HTTP (e.g., YouTube), RTP [RFC 1889] | TCP or UDP |
| Internet telephony | SIP, RTP, proprietary (e.g., Skype) | TCP or UDP |



# HTTP Overview

- *web page* consists of *objects*
  - **Objects** can be HTML file, JPEG image, Java applet, audio file,…
  - web page consists of base HTML-file which includes several referenced objects
  - each object is addressable by a URL, e.g.,

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

host name | path name

- 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 specifications ([RFC 1945] and [RFC 2616])
  - define only the communication protocol between the client HTTP program and the server HTTP program.
  - have nothing to do with how a client interprets a Web page

# HTTP Connections

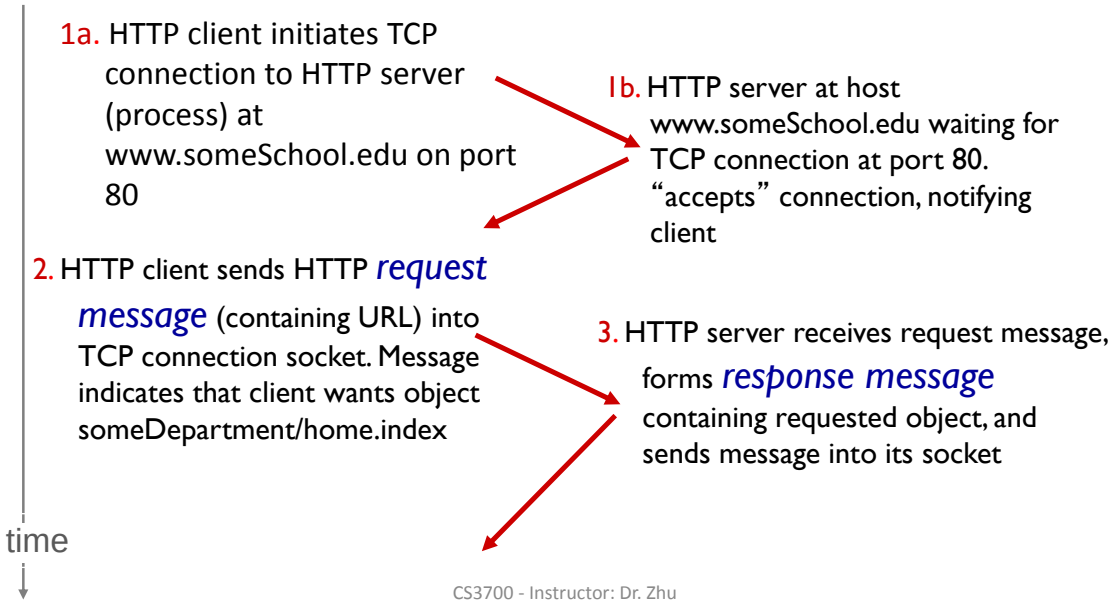
- HTTP uses TCP:
  - client initiates TCP connection (creates socket) to server, port 80
  - server accepts TCP connection from client
  - HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
  - TCP connection closed
- HTTP is "stateless"
  - server maintains no information about past client requests
- non-persistent HTTP v.s. persistent HTTP
  - non-persistent HTTP: at most one object sent over TCP connection; connection is then closed
    - downloading multiple objects required multiple connections
  - persistent HTTP: multiple objects can be sent over single TCP connection between client, server



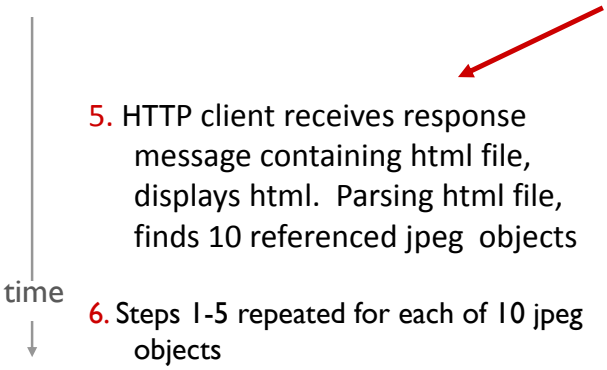
# Non-persistent HTTP

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

(contains text, references to 10 jpeg images)



# Non-persistent HTTP (cont.)




4. HTTP server closes TCP connection.

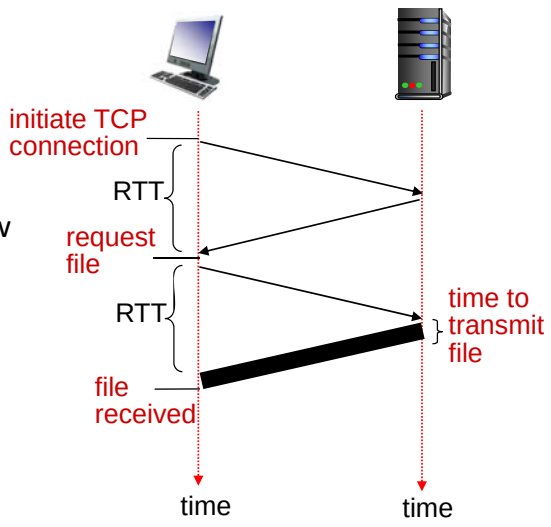
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

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



# Non-persistent HTTP: response time

- RTT (definition): time for a small packet to travel from client to server and back
- HTTP response time:
  - one RTT to initiate TCP connection
  - one RTT for HTTP request and first few bytes of HTTP response to return
  - file transmission time
  - non-persistent HTTP response time = 2RTT+ file transmission time



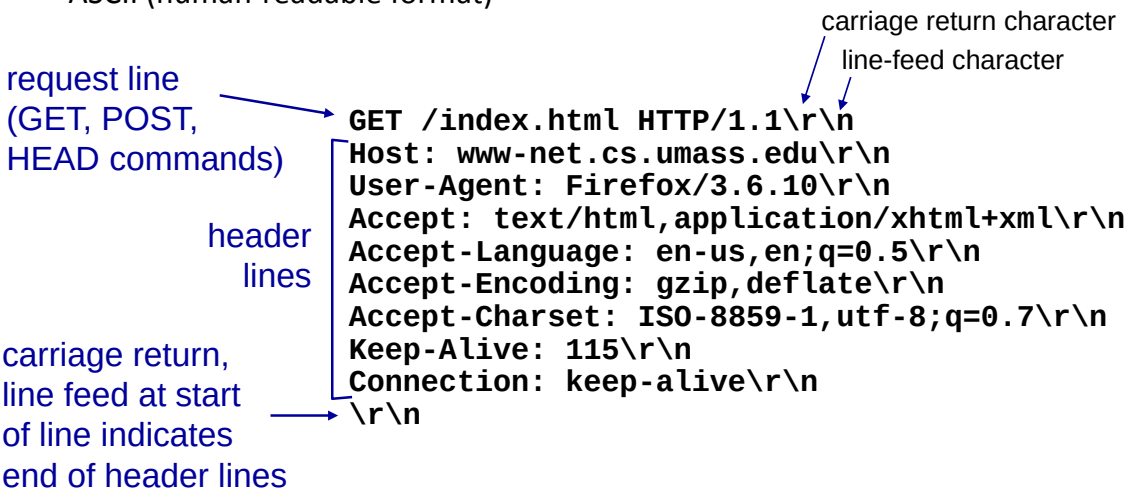
# Persistent HTTP

- 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 one RTT for all the referenced objects

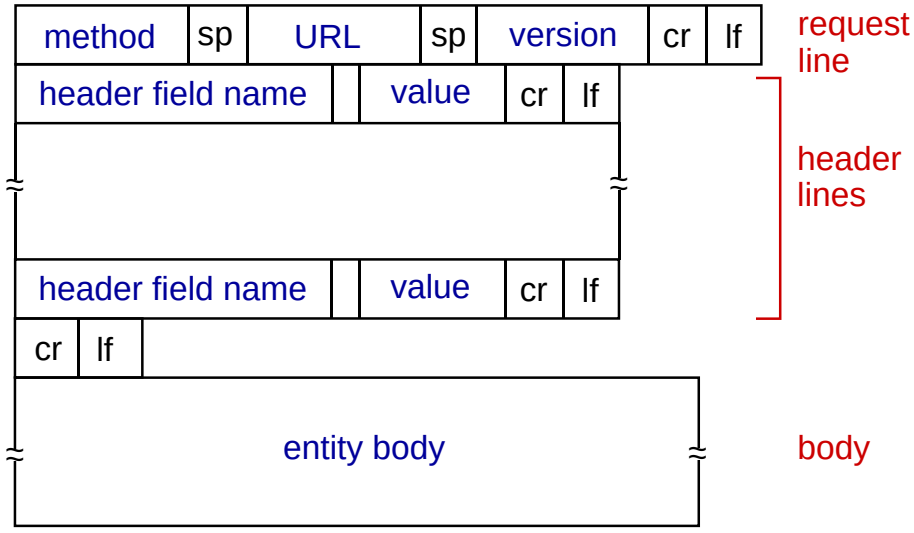


# HTTP request message

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



# HTTP request message: general format

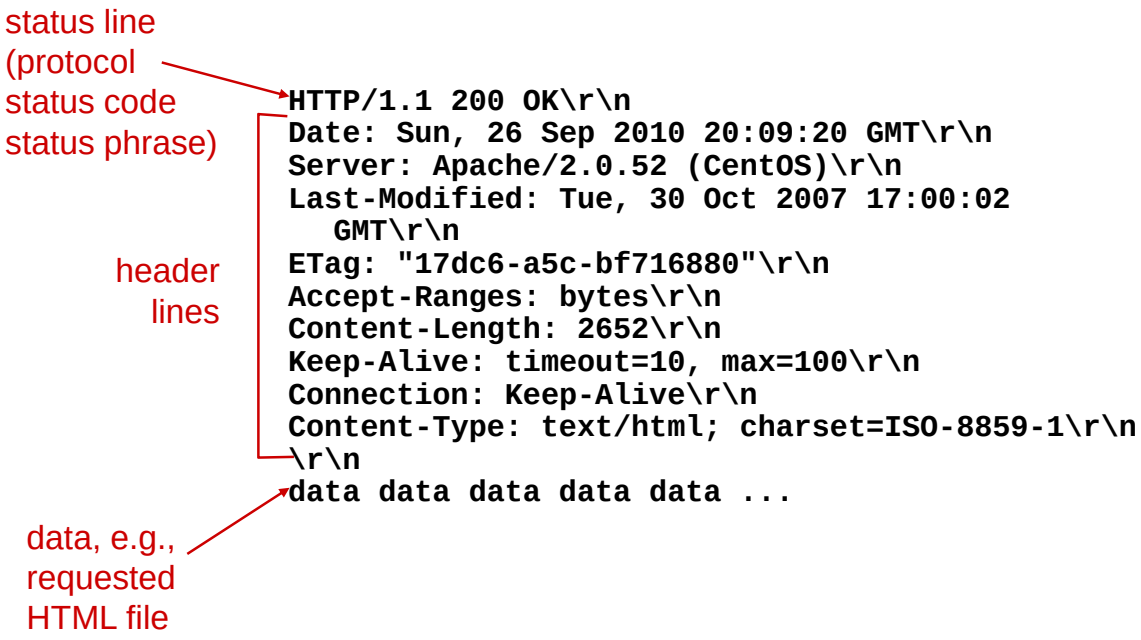





# HTTP Request message: Method types

- HTTP/1.0:
  - GET: Just request a webpage by leaving the "Entity Body" empty
  - POST: Upload form input to server in "entity body" when user fills out a form
    - URL Method: Use GET method and upload input in URL field of request line
      `www.somesite.com/animalsearch?monkeys&banana`
  - HEAD: = GET but server leaves requested object out of response(for debugging)
- HTTP/1.1:
  - GET, POST, HEAD
  - PUT: Upload file (object) in entity body to path specified in URL field on a server
  - DELETE: Delete file specified in the URL field on a server

# HTTP response message: from Server to Client






# 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 msg

  301 Moved Permanently

  – requested object moved, new location specified later in this msg (Location:)

  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) using Telnet

1. Telnet to a Web server of your choice

```
telnet www.msudenver.edu 80
```

opens TCP connection to port 80 (default HTTP server port) at www.msudenver. anything typed in will be sent to port 80 at www.msudenver

2. type in a GET HTTP request:

```
GET /about/ HTTP/1.1
Host: www.msudenver.edu
```

by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

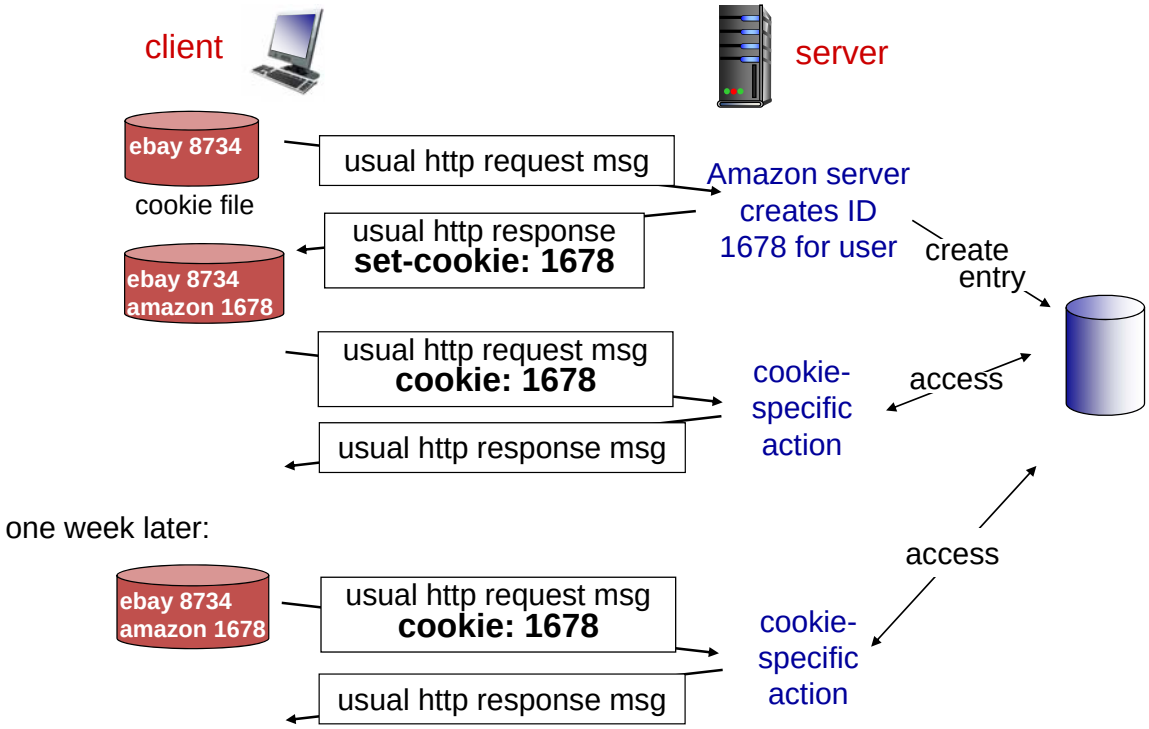
3. look at response message sent by HTTP server!



# Cookies: keep User-Server State

- cookies can be used for:
  - authorization
  - shopping carts
  - recommendations
  - user session state (Web e-mail)
- how to keep "state":
  - protocol endpoints: maintain state at sender/receiver over multiple transactions
  - cookies: http messages carry state
- Cookies: an invasion of privacy
  - cookies permit sites to learn a lot about you
  - you may supply name and e-mail to sites that may sell your information to a third party

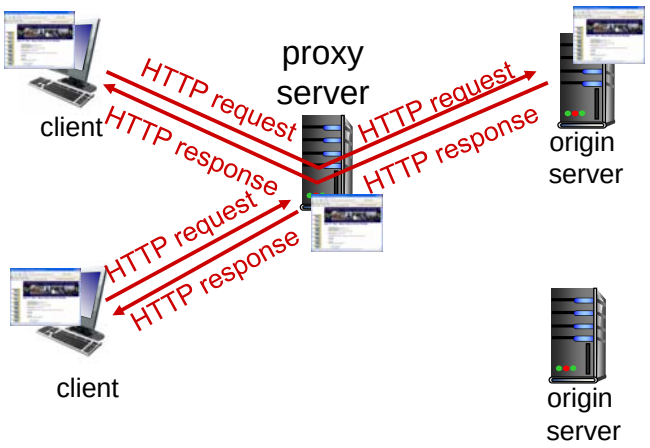
# Cookies: keep User-Server State (Cont.)






# Web caches (proxy server)

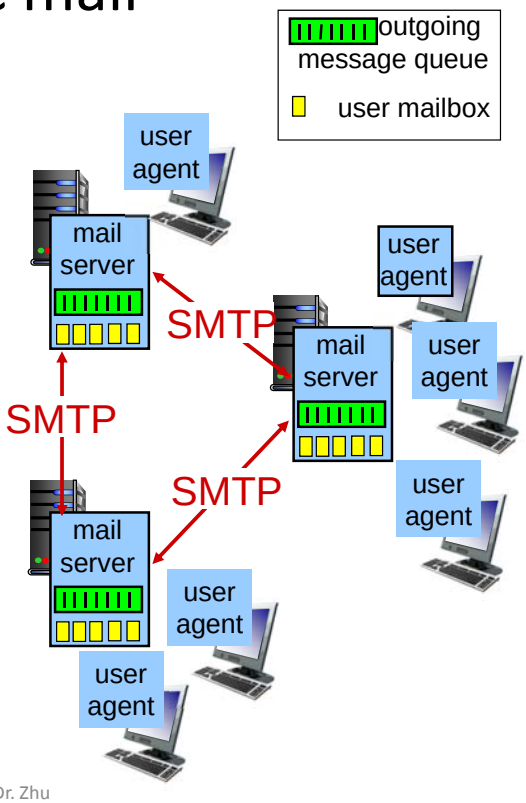
- goal: satisfy client request without involving origin server
  - user sets browser: Web accesses via cache
  - browser sends all HTTP requests to cache
    - object in cache: cache returns object
    - else cache requests object from origin server, then returns object to client




- cache acts as both client and server
- typically cache is installed by ISP (university, company, residential ISP)
- Web caching:
  - reduce response time for client request
  - reduce traffic on an institution's access link
  - enables "poor" content providers to effectively deliver content

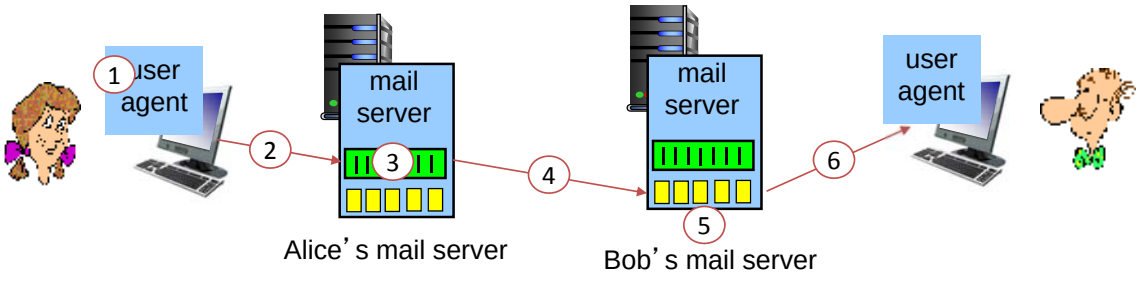
# Electronic mail

- Three major components:
  - user agents
  - mail servers
  - simple mail transfer protocol: SMTP
- User Agent (a.k.a. “mail reader”)
  - composing, editing, reading mail messages
  - Outlook, Thunderbird, iPhone mail client, …
  - PUSH outgoing mail onto server via SMTP
  - PULL incoming mail from server via mail access protocols
- mail servers:
  - mailbox contains incoming messages for user
  - message queue of outgoing (to be sent) mail messages
  - SMTP protocol is used between mail servers to send email messages
    - client: mail server sending a mail
    - server: mail server receiving a mail






# Scenario: Alice sends message to Bob




1) Alice uses UA to compose message “to” `bob@someschool.edu`
2) Alice’s UA PUSHes message to her mail server via SMTP; 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

*Direct TCP connection between two mail servers w/o any intermediate mail sever

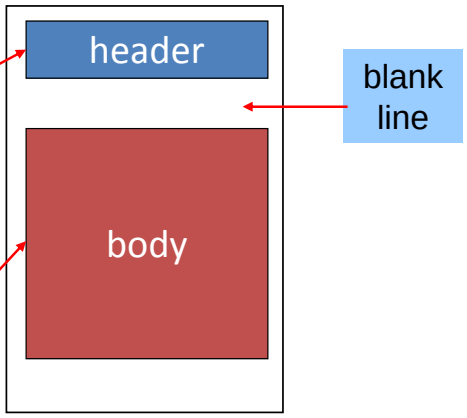
# Electronic Mail: SMTP [RFC 2821]

- uses TCP to reliably transfer email message from client to server
- Server, the receiving mail server, listens at port 25
- direct transfer: from sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction (like HTTP, FTP)
  - commands: ASCII text
    **HELO, MAIL FROM, RCPT TO, DATA, QUIT**
  - response: status code and phrase, ASCII text
    status code: **220, 250, 354, 221**, etc.
- Mail messages (DATA) must be in 7-bit ASCII
  - Each character is encoded using 8 bits, a Byte



# Mail message format

- SMTP: protocol for exchanging email messages
- RFC 822: standard for text message format:
  - header lines, e.g.,
    **To: … \r\n**
    **From: … \r\n**
    **Subject: … \r\n**
    *NOT* SMTP MAIL FROM, RCPT TO commands!
  - Blank line between header and body, e.g.,
    **\r\n**
  - Body: the "message", e.g.,
    **how are you doing?\r\n**
    **etc.\r\n**
  - CRLF . CRLF marks end of message, e.g.,
    **.\r\n**



# 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: To: bob@hamburger.edu
C: From: alice@crepes.fr
C: Subject: ketchup and pickles
C:
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```



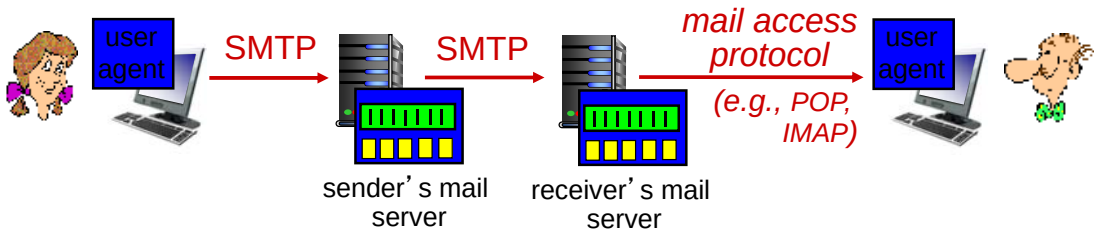
# Try SMTP interaction for yourself:

- **`telnet hamburger.edu 25`**
  - Metro email server: mail.mscd.edu
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

➔ an email might be sent to your email account at mail.mscd.edu sometime, but most likely it will be classified as a junk mail.

➔ Most likely, at certain point, the command will be reject due to blocked telnet port, invalid address, unauthorized sender issues, etc.

# Mail access protocols



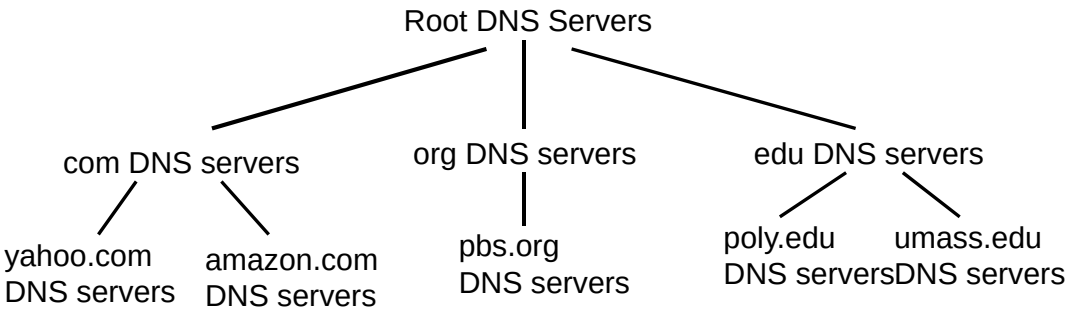

- SMTP: PUSH mail from UA to sender's server or from sender's server to receiver's server
- mail access protocol: retrieval from server (PULL)
  - POP: Post Office Protocol [RFC 1939]: authorization, download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  - HTTP: gmail, Hotmail, Yahoo! Mail, etc.



# DNS: 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)
  - core Internet function
  - complexity at network's "edge"
- *DNS services*
  - hostname to IP address translation
  - host aliasing: canonical (real) names and aliases
  - mail server aliasing
  - load distribution
    - replicated Web servers: many IP addresses correspond to one name
- *why not centralize DNS?*
  - single point of failure
  - traffic volume
  - distant centralized database
  - maintenance

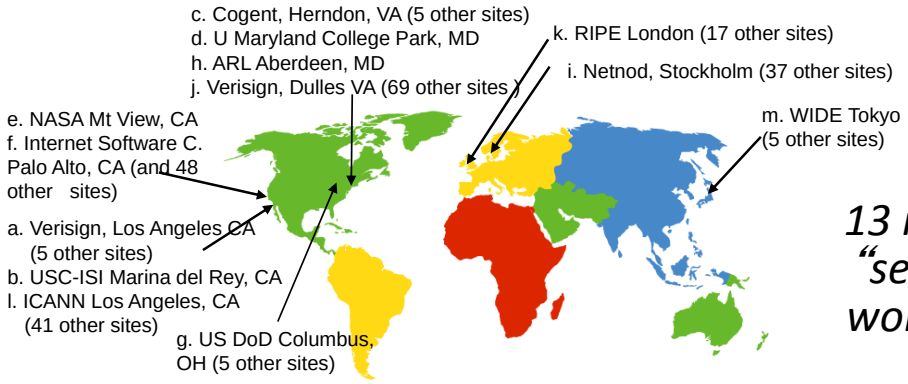
# DNS: a distributed, hierarchical database




- 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



# DNS: root name servers




*13 root name "servers" worldwide*

- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server

# TLD servers 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.: uk, fr, ca, jp
  - Network Solutions maintains servers for .com TLD
  - Educause for .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 server

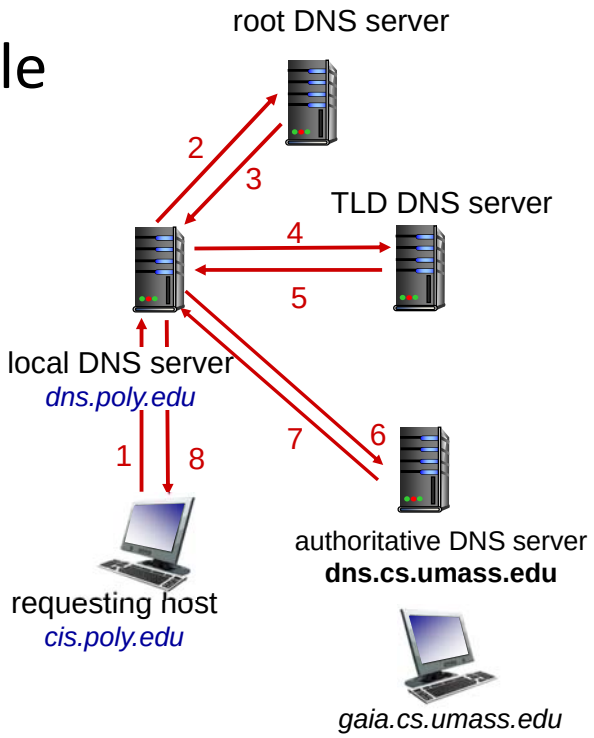
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

# DNS name resolution example

- host at cis.poly.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"

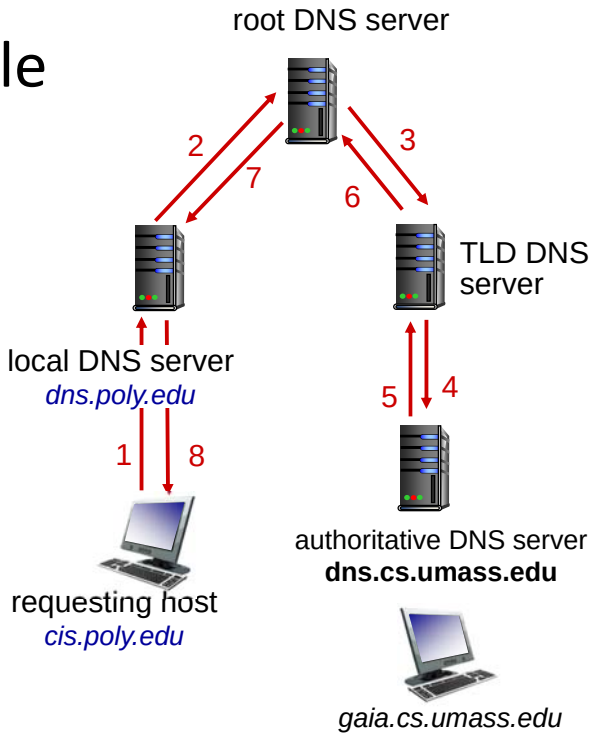





# DNS name resolution example

recursive query:

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



# DNS: caching, updating 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
- 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



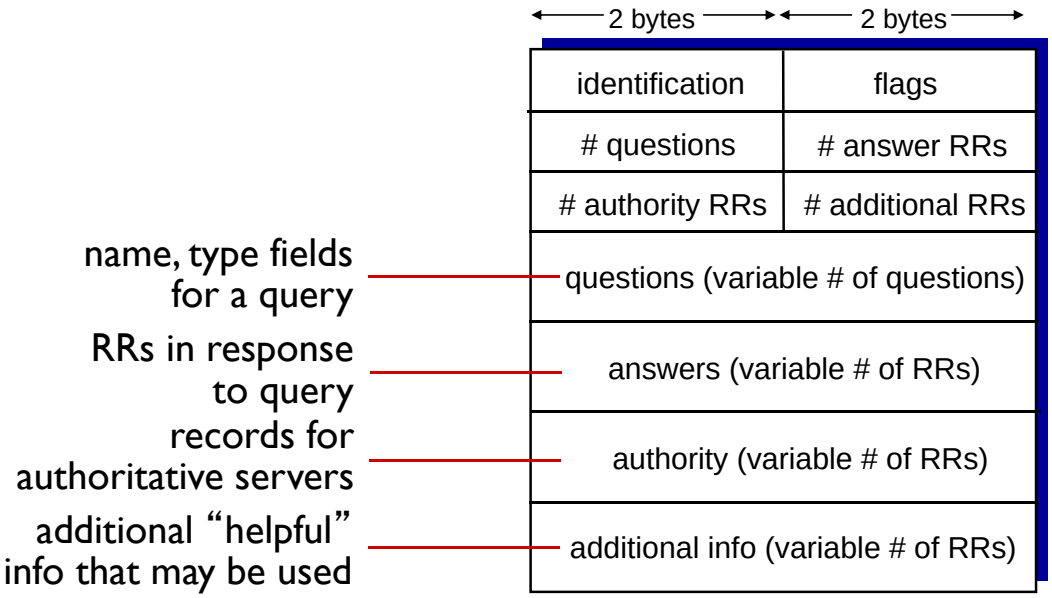
# DNS records

- *DNS:* distributed db 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., amazon.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:
  - **`name`** is domain (e.g., amazon.com)
  - **`value`** is is name of mailserver associated with **`name`**

# DNS protocol, messages






# 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 two RRs into .com TLD server:

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

    (**`dns1.networkutopia.com, 212.212.212.1, A`**)
- create authoritative server type A record for www.networkuptopia.com; type MX record for networkutopia.com

# Attacking DNS

- 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 queries
  - DNS poisoning
    - Send bogus relies to DNS server, which caches
- Exploit DNS for DDoS
  - Send queries with spoofed source address: target IP
  - Requires amplification