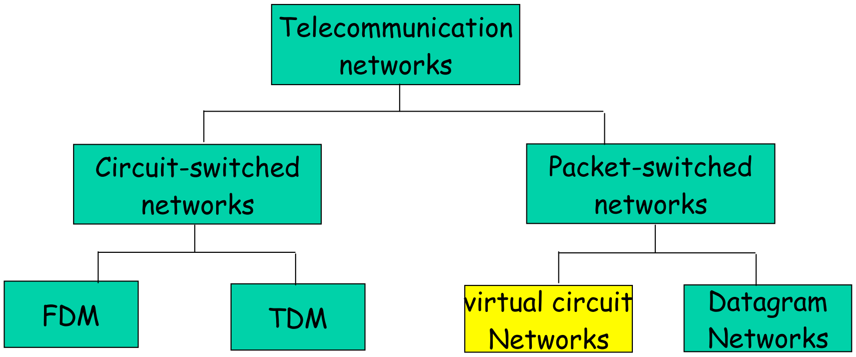
What is a virtual circuit network?



Telecommunication networks

Circuit-switched networks

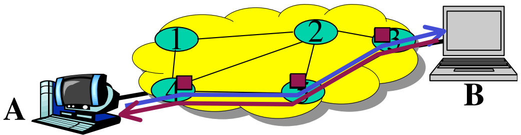
Packet-switched networks

FDM

TDM

virtual circuit Networks

Datagram Networks



Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP

Tue 4/8

- 2.3 FTP
- 2.4 Electronic Mail

Next week

- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP

Thu 4/10

- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing

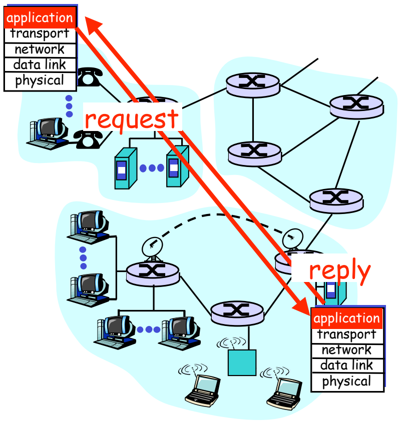
Applications and application-layer protocols

- Application

- Application-layer protocols

- Client:

- Server



Q: how does one process "identify" the other process with which it wants to communicate?

Network applications: some jargons

- Process: program running within a host.
- two processes within the same host communicate using interprocess communication (defined by operating system)
- Processes running in different hosts communicate through an application-layer protocol
- user agent: software process.
  - Web:browser
  - E-mail: mail reader
- API: Application Programming Interface
  - defines interface between application and transport layer
- socket: Internet API
  - two processes communicate by sending data into socket, reading data out of socket

Internet transport protocols services

TCP service

- connection-oriented: setup connection between client and server first
- reliable data delivery between the two ends
- flow control: sender won't overwhelm receiver
- congestion control: throttle sender when network overloaded
- does not provide:
  - Timing
  - bandwidth guarantees

UDP service

- unreliable data transfer between sending and receiving process
- does not provide:
  - connection setup
  - Reliability
  - flow control
  - congestion control
  - timing, or
  - bandwidth guarantee

World Wide Web

- Web page
- Object: HTML file, JPEG image, Java applet, audio file,...
- Each object is addressable by a URL (Universal resource locator)

  app://host_name:port#/path_and_file_name

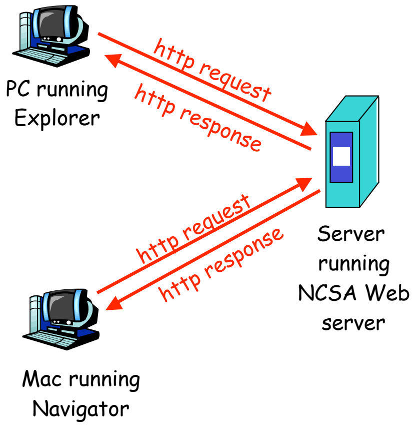
ex: http://www.cs.ucla.edu/classes/spring03/cs118/slides.html

- Web browser: User agent for Web
- Web server:

The Web and http protocol

http: hypertext transfer protocol

- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- http1.0: RFC 1945
- http1.1: RFC 2068



The http protocol: more

Use TCP transport service

http is "stateless"

- server maintains *no information* about *past* client requests

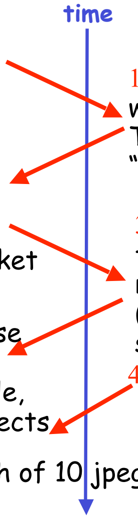
aside

Protocols that maintain "state" are more complex!
past history (state) must be maintained
if server/client crashes, their views of "state" may be inconsistent, must be reconciled

http example (cont.)

fetch www.someSchool.edu/someDepartment/home.index

(contains text, references to 10 jpeg images)



1a. http client initiates TCP connection to http server (process) at www.SomeSchool.edu

1b. http server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection

2. http client sends http *request message* (containing URL) into TCP connection socket

3. http server receives request, forms *response msg* containing requested object (someDepartment/home.index), sends message into socket

4. http server closes TCP conn.

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, persistent connections

Non-persistent

- http/1.0: server parses request, responds, closes TCP connection
- 2 RTTs to fetch object
  - TCP connection
  - object request/transfer
- many browsers open multiple parallel connections

Persistent

- default for htp/1.1
- on same TCP connection: server parses request, responds, parses new request,..
- client sends requests for all referenced objects as soon as it receives base HTML.
- fewer RTTs

http message format: request

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

request line (GET, POST, HEAD commands)

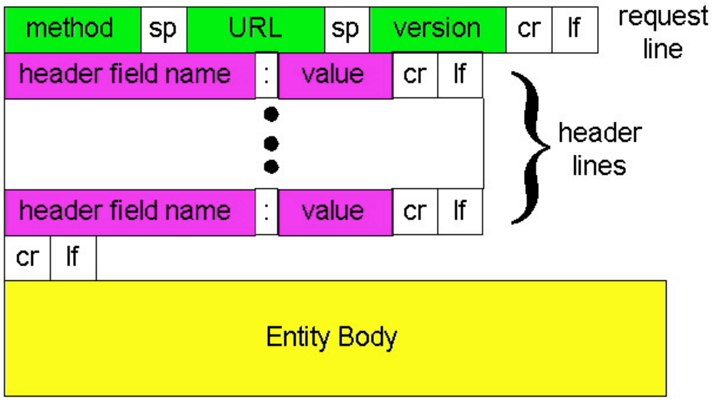
header lines

```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif,image/jpeg
Accept-language:fr
```

(extra carriage return, line feed)

Carriage return, line feed indicates end of message

http request message: general format



method sp URL sp version cr lf

request line

header field name : value cr lf

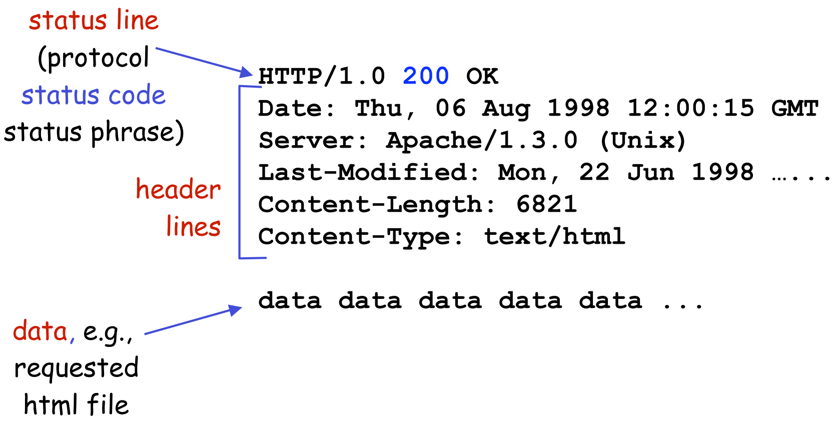
header field name : value cr lf

header lines

cr lf

Entity Body

http message format: response



status line (protocol status code status phrase)

header lines

data, e.g., requested html file

```
HTTP/1.0 200 OK
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

http response status codes

In first line in server->client response message.

A few sample codes:

200 OK

301 Moved Permanently

400 Bad Request

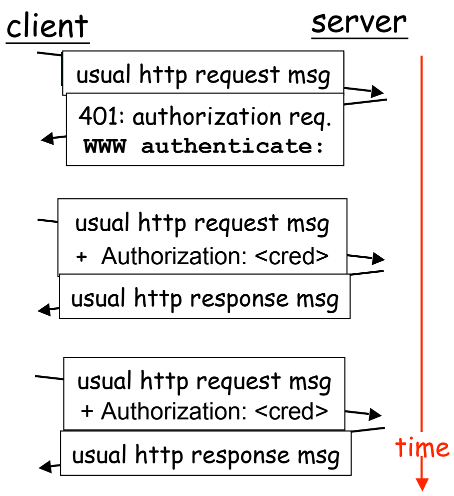
404 Not Found

505 HTTP Version Not Supported

User-server interaction: authentication

control access to the content

- authorization credentials: typically name, password
- stateless: client must present authorization in *each* request
  - authorization: header line in each request
  - if no authorization: header, server refuses access



client

server

usual http request msg

401: authorization req.
`WWW authenticate:`

usual http request msg
+ Authorization: <cred>

usual http response msg

usual http request msg
+ Authorization: <cred>

usual http response msg

time

Cookies: keeping "state"

Many major Web sites use cookies

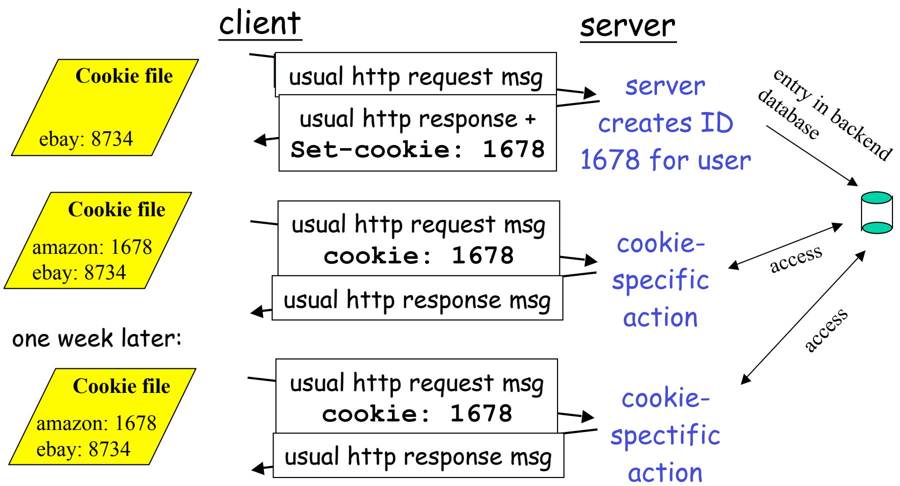
Four components:

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

Example:

- Susan access Internet always from same PC

Cookies: keeping "state" (cont.)



client

server

Cookie file

ebay: 8734

usual http request msg

usual http response +
`Set-cookie: 1678`

server
creates ID
1678 for user

entry in backend
database

Cookie file

amazon: 1678
ebay: 8734

usual http request msg
`cookie: 1678`

usual http response msg

cookie-
specific
action

access

one week later:

Cookie file

amazon: 1678
ebay: 8734

usual http request msg
`cookie: 1678`

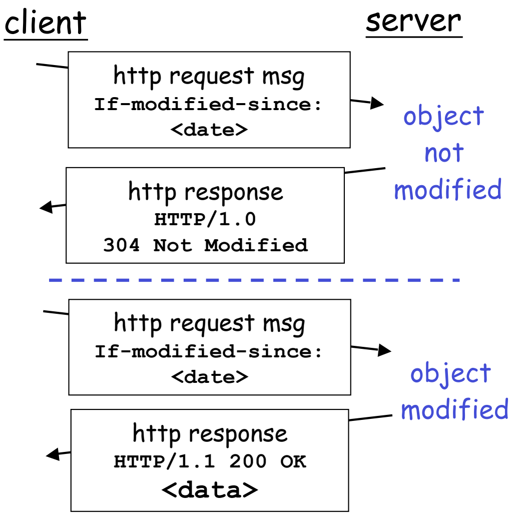
usual http response msg

cookie-
spectific
action

access

Conditional GET: client-side caching

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



client

server

http request msg
`If-modified-since: <date>`

object
not
modified

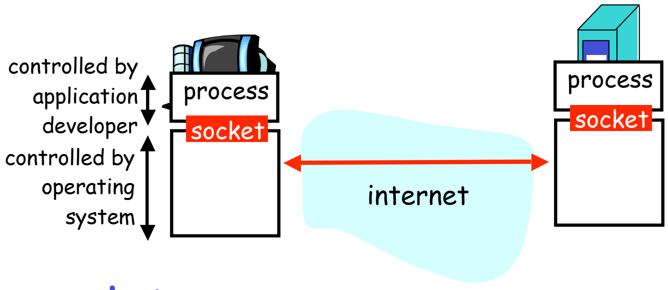
http response
`HTTP/1.0`
`304 Not Modified`

http request msg
`If-modified-since: <date>`

object
modified

http response
`HTTP/1.1 200 OK`
`<data>`

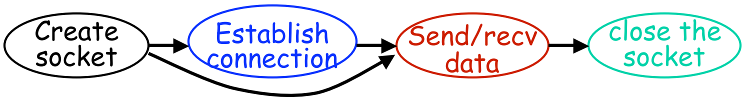
Socket programming



socket

a *host-local*, *application-created/owned*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process

Socket API:



Socket functional calls

- socket (): Create a socket
- bind(): bind a socket to a local IP address and port #
- listen(): passively waiting for connections
- connect(): initiating connection to another socket
- accept(): accept a new connection
- Write(): write data to a socket
- Read(): read data from a socket
- sendto(): send a datagram to another UDP socket
- recvfrom(): read a datagram from a UDP socket
- close(): close a socket (tear down the connection)

Socket programming with TCP

Client must contact server

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

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process

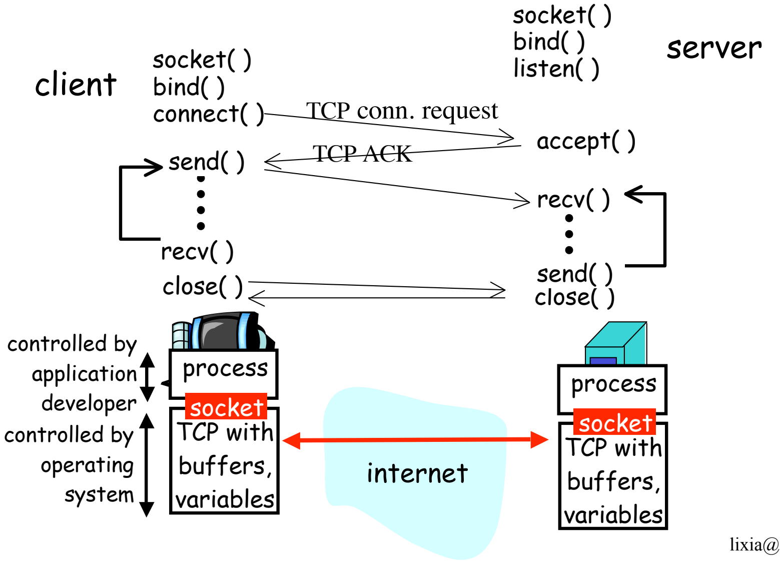
- When client creates socket: client TCP establishes connection to server TCP
- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients

application viewpoint

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

Socket-programming using TCP

TCP service: reliable byte stream transfer



client

socket( )
bind( )
connect( )

send( )

recv( )
close( )

server

socket( )
bind( )
listen( )

accept( )

recv( )

send( )
close( )

TCP conn. request

TCP ACK

controlled by application developer

controlled by operating system

process

socket

TCP with buffers, variables

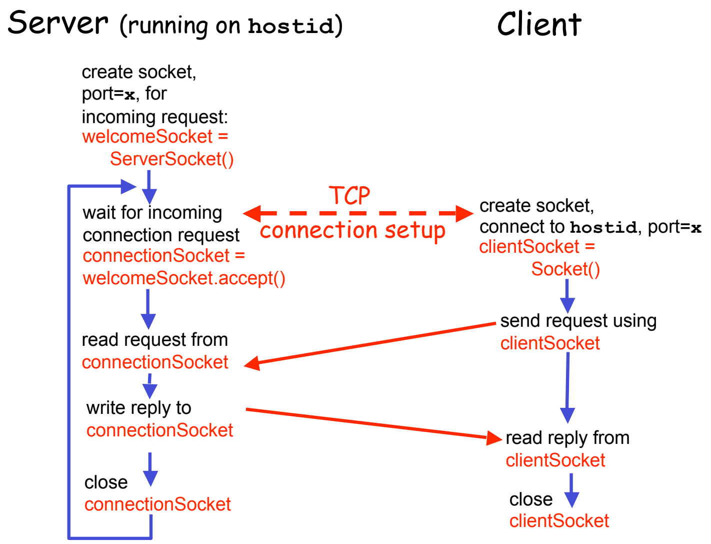
internet

process

socket

TCP with buffers, variables

Client/server socket interaction: TCP



Server (running on hostid)

Client

create socket,
port=x, for
incoming request:
welcomeSocket =
ServerSocket()

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

TCP
connection setup

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

send request using
clientSocket

read request from
connectionSocket

write reply to
connectionSocket

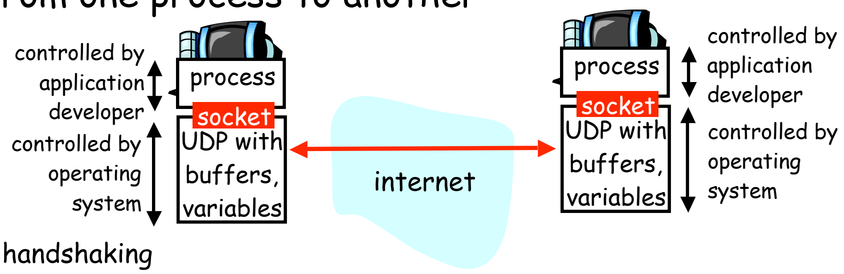
read reply from
clientSocket

close
connectionSocket

close
clientSocket

Socket-programming using UDP

UDP service: unreliable transfer of data blocks from one process to another

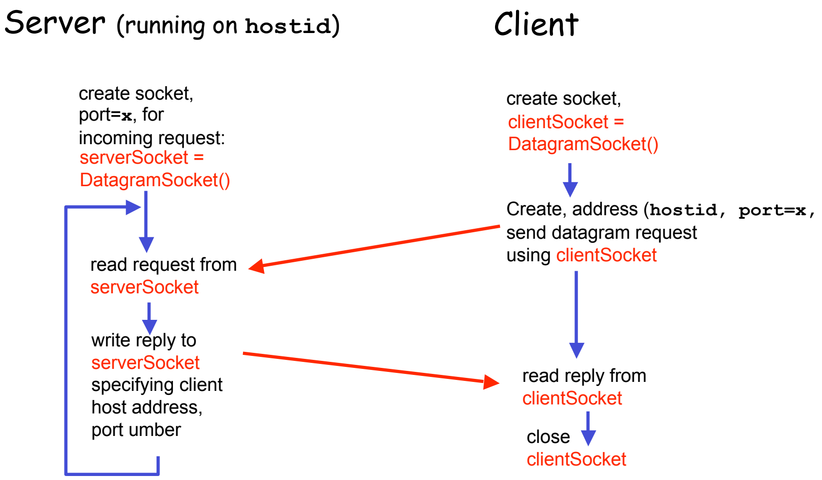


- ❖no handshaking
- ❖sender explicitly attaches IP address and port of destination
- ❖transmitted data may be received with bit error, out of order, or lost

application viewpoint

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

Client/server socket interaction: UDP



Server (running on **hostid**)

Client

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

read request from
serverSocket

write reply to
serverSocket
specifying client
host address,
port umber

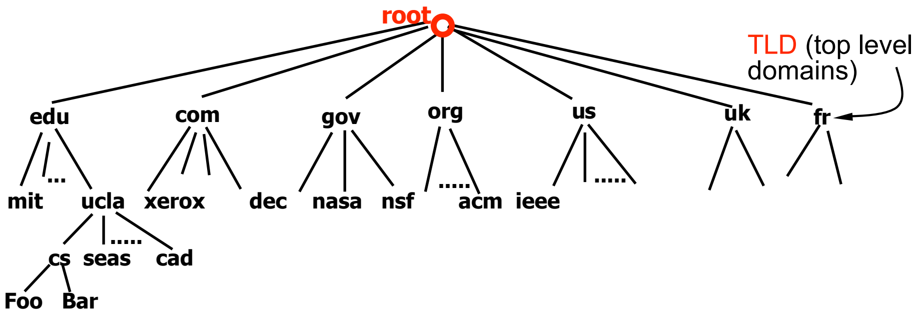
create socket,
clientSocket =
DatagramSocket()

Create, address (**hostid, port=x,**
send datagram request
using clientSocket

read reply from
clientSocket

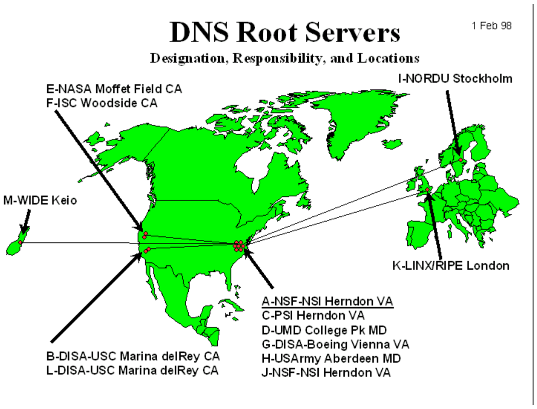
close
clientSocket

Domain Name System



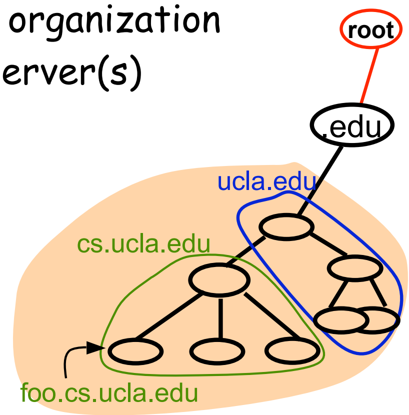
DNS: Root name servers

- 13 root name servers worldwide
  - holding identical DNS database
- Your DNS query goes to local DNS server, for names it cannot resolve, it contact one of the root servers
- root name server:
  - If it knows the exact answer, reply
  - Otherwise reply with the pointer to another name server



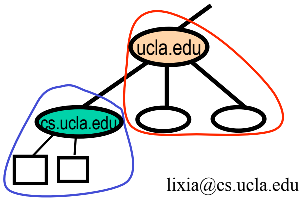
DNS as a distributed database

- entire DNS name space is divided to a hierarchy of zones
  - zone: a *continuous* sub-space in the DNS name tree
  - a zone may contain domains at different levels
- each zone is controlled by an organization
- Each zone has its own name server(s)



What's in the zone's master file:

1. data that defines the top node of the zone
   - including a list all the servers for the zone
2. authoritative data for all nodes in the zone
   - all RRs for all of the nodes from the top node to leaf nodes(that are outside of any subzone)
3. data that describes delegated subzones
   - Domain name, owner, etc
4. "glue data": *IP address(es)* for subzone's name server(s)



DNS example

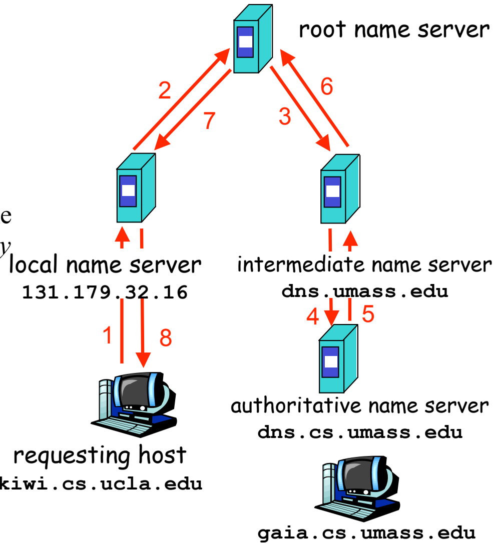
host kiwi**.cs.ucla.edu** wants IP address of **gaia.cs.umass.edu**

1. Contacts its local DNS server, **131.179.32.16** (dns.cs.ucla.edu)

2. **dns.cs.ucla.edu** contacts root name server, *if necessary*

3. root name server contacts umass name server, **dns.umass.edu**, *if necessary*

4. **dns.umass.edu** contacts the authoritative name server, **dns.cs.umass.edu**, if necessary



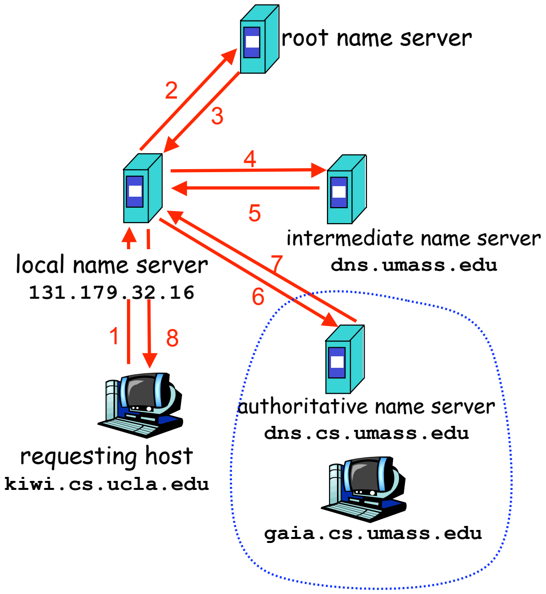
DNS: iterated queries

recursive query:

- puts burden of name resolution on contacted name server
- heavy load?

iterated query:

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS Performance

- Virtual each and all Internet applications invoke DNS lookup
- use both replication and caching to improve performance
  - Each domain has one or more secondary servers
  - servers cache recent query results
    - buffer recently resolved names and addresses till their "time-to-live" expires

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. foo.com)
  - **value** is IP address of authoritative name server for this domain

Type=CNAME
name is an alias name for some "canonical" (the real) name
value is canonical name

Type=MX
value is hostname of mailserver associated with **name**

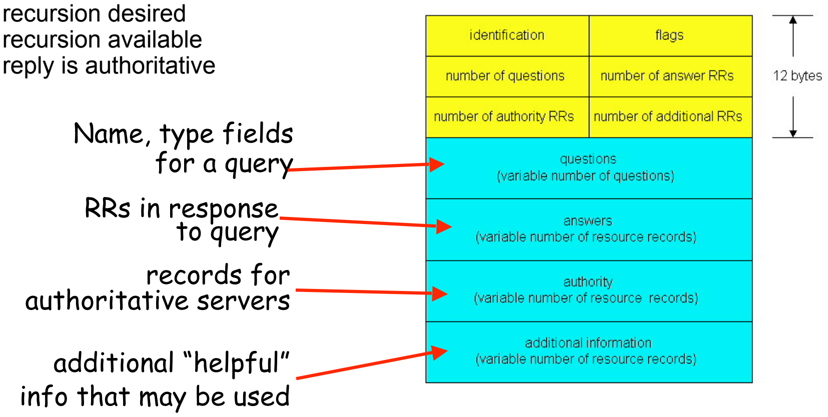
DNS protocol, messages

DNS protocol : *query* and *reply* messages, use same *message format*

msg header

identification: 16 bit # for query, reply to query uses same #

flags: query or reply
recursion desired
recursion available
reply is authoritative



Name, type fields for a query

RRs in response to query

records for authoritative servers

additional "helpful" info that may be used

identification	flags
number of questions	number of answer RRs
number of authority RRs	number of additional RRs

questions
(variable number of questions)

answers
(variable number of resource records)

authority
(variable number of resource records)

additional information
(variable number of resource records)

How to use DNS in practice?

Two popular programs you can use:

- "host" – look up host names using domain servers
  - Command: host [-l] [-v] [-w] [-r] [-d] [-t query type] host [server]
  - Manual page: man host
- "nslookup" – query Internet name servers interactively
  - Command: nslookup [-options…] [host-to-find | –[server]]
  - Manual page: man nslookup