Chapter 2: Application Layer

last updated 22/09/03

Chapter goals:

- conceptual + implementation aspects of network application protocols
  - client server paradigm
  - service models
- learn about protocols by examining popular application-level protocols

More chapter goals

- specific protocols:
  - http
  - ftp
  - smtp
  - pop
  - dns
- programming network applications
  - socket programming

Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
    - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
    - Content distribution networks vs. Web Caching

Applications and application-layer protocols

Applications: communicating, distributed processes

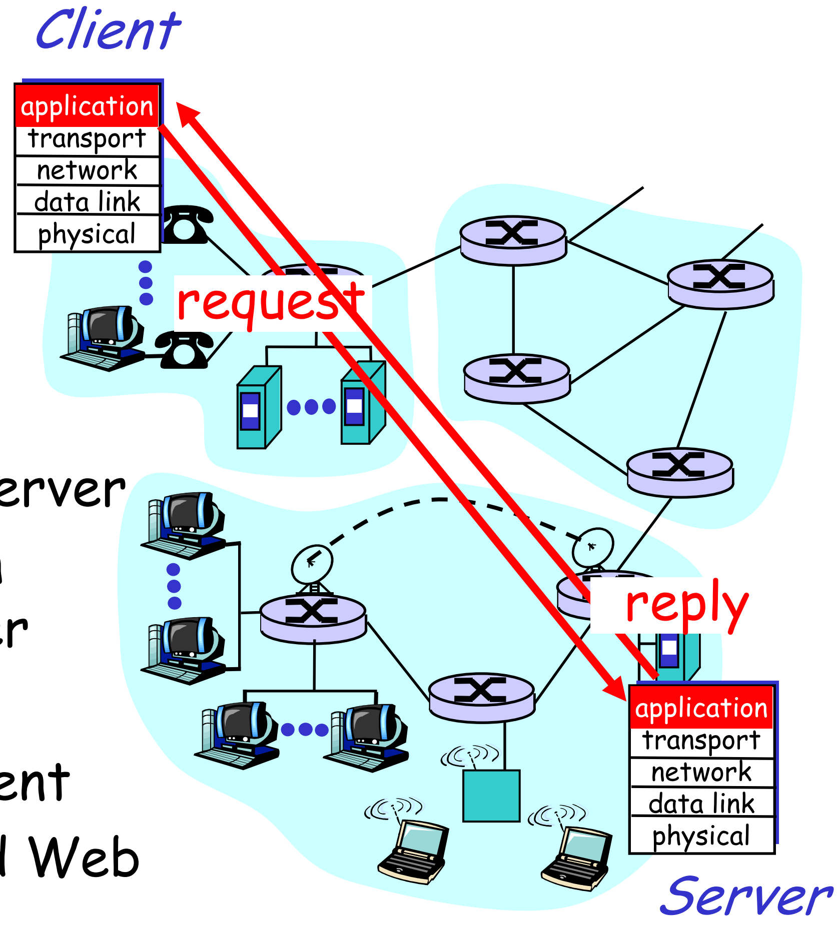
- running the "user space" of network hosts
- which exchange messages among themselves
- Network Applications are applications which involves interactions of processes implemented in multiple hosts connected by a network. Examples: the web, email, file transfer
- Within the same host, processes communicate with interprocess communication defined by the OS (Operating System).
- Processes running in different hosts communicate with an application-layer protocol

Application-layer protocols

- a "piece" of Application (apps)
- define messages exchanged by apps and actions taken
- uses services provided by lower layer protocols

Client-server paradigm

Typical network app has two pieces: *client* and *server*



Client:

- initiates contact with server ("speaks first")
- typically requests service from server
- for Web, client is implemented in browser; for e-mail, in mail reader

Server:

- provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail

Application-layer protocols (cont).

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

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

- IP address of host running other process
- "port number" - allows receiving host to determine to which local process the message should be delivered

… lots more on this later.

What transport service does an app need?

Data loss

- some apps (e.g., audio) can tolerate some loss
- other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Bandwidth

- some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
- other apps ("elastic apps") make use of whatever bandwidth they get

Timing

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

Transport service requirements of common apps

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps up	yes, 100's msec
financial apps	no loss	elastic	yes and no

Services provided by Internet transport protocols

TCP service:

- *connection-oriented:* setup required between client, server
- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not providing:* timing, minimum bandwidth guarantees

UDP service:

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

Internet apps: their protocols and transport protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	smtp [RFC 821]	TCP
remote terminal access	telnet [RFC 854]	TCP
Web	http [RFC 2068]	TCP
file transfer	ftp [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
remote file server	NFS	TCP or UDP
Internet telephony	proprietary (e.g., Vocaltec)	typically UDP

Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

The Web: some jargon

- Web page:
  - consists of "objects"
  - addressed by a URL
- Most Web pages consist of:
  - base HTML page, and
  - several referenced objects.
- URL has two components: host name and path name:

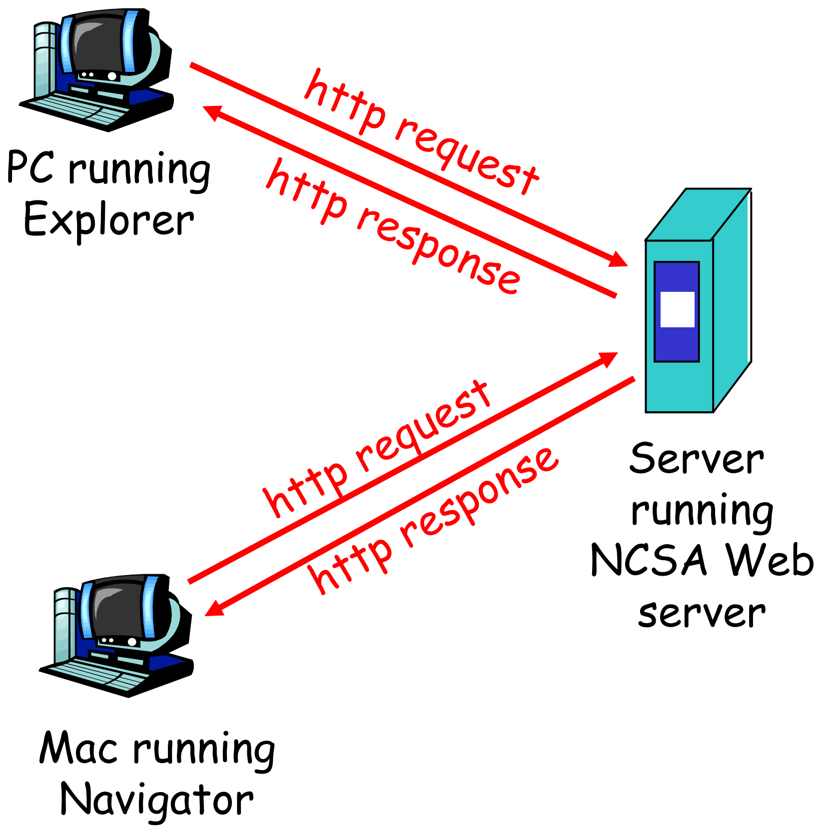
- User agent for Web is called a browser:
  - MS Internet Explorer
  - Netscape Communicator
- Server for Web is called Web server:
  - Apache (public domain)
  - MS Internet Information Server

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

The Web: the http protocol

http: hypertext transfer protocol

- Web's application layer 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

http: TCP transport service:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- http messages (application-layer protocol messages) exchanged between browser (http client) and Web server (http server)
- TCP connection closed

http is "stateless"

- server maintains no information about past client requests

aside

Protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

http example

Suppose user enters URL
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. Port 80 is default for http server.

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

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

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

time

http example (cont.)

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

time

4. http server closes TCP connection.

Non-persistent and persistent connections

Non-persistent

- HTTP/1.0
- server parses request, responds, and closes TCP connection
- At least 2 RTTs (Round Trip Time) to fetch each object
- Repeated 10 times for 10 objects. Each object transfer suffers from slow start

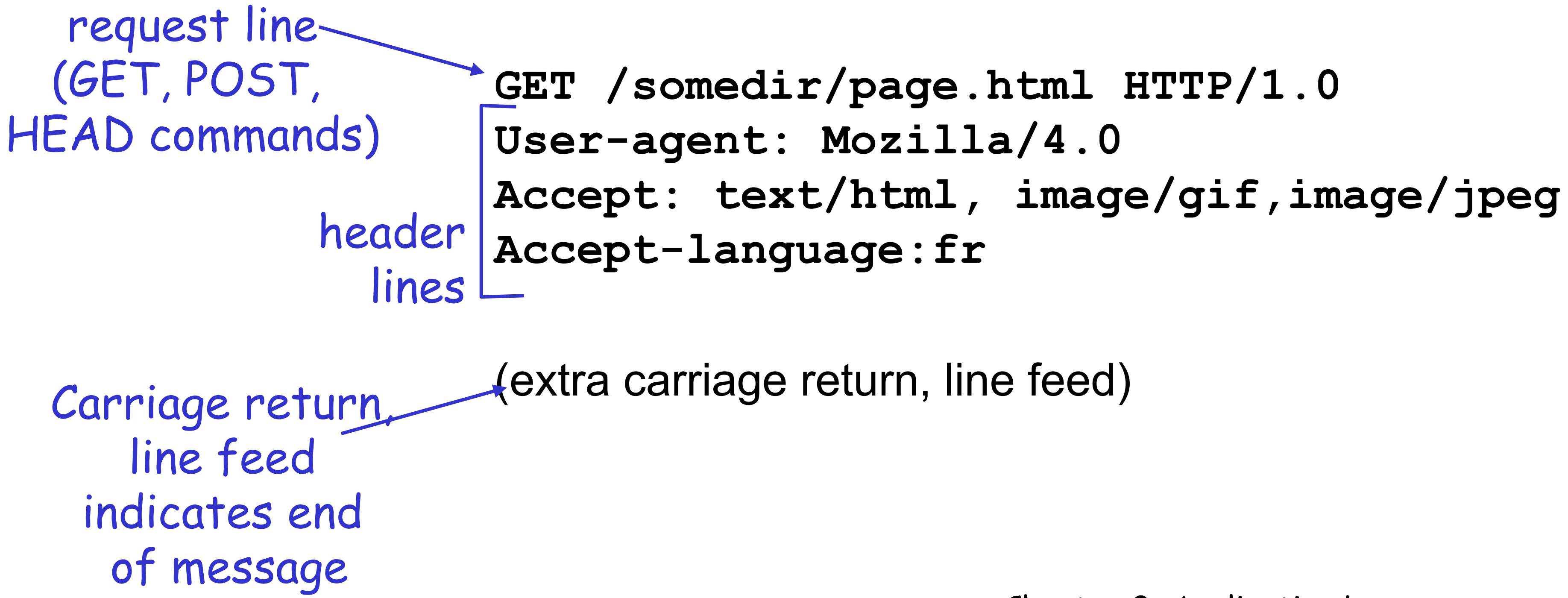
But most 1.0 browsers use parallel TCP connections.

Persistent

- default for HTTP/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 and less slow start.

http message format: request

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



request line (GET, POST, HEAD commands)

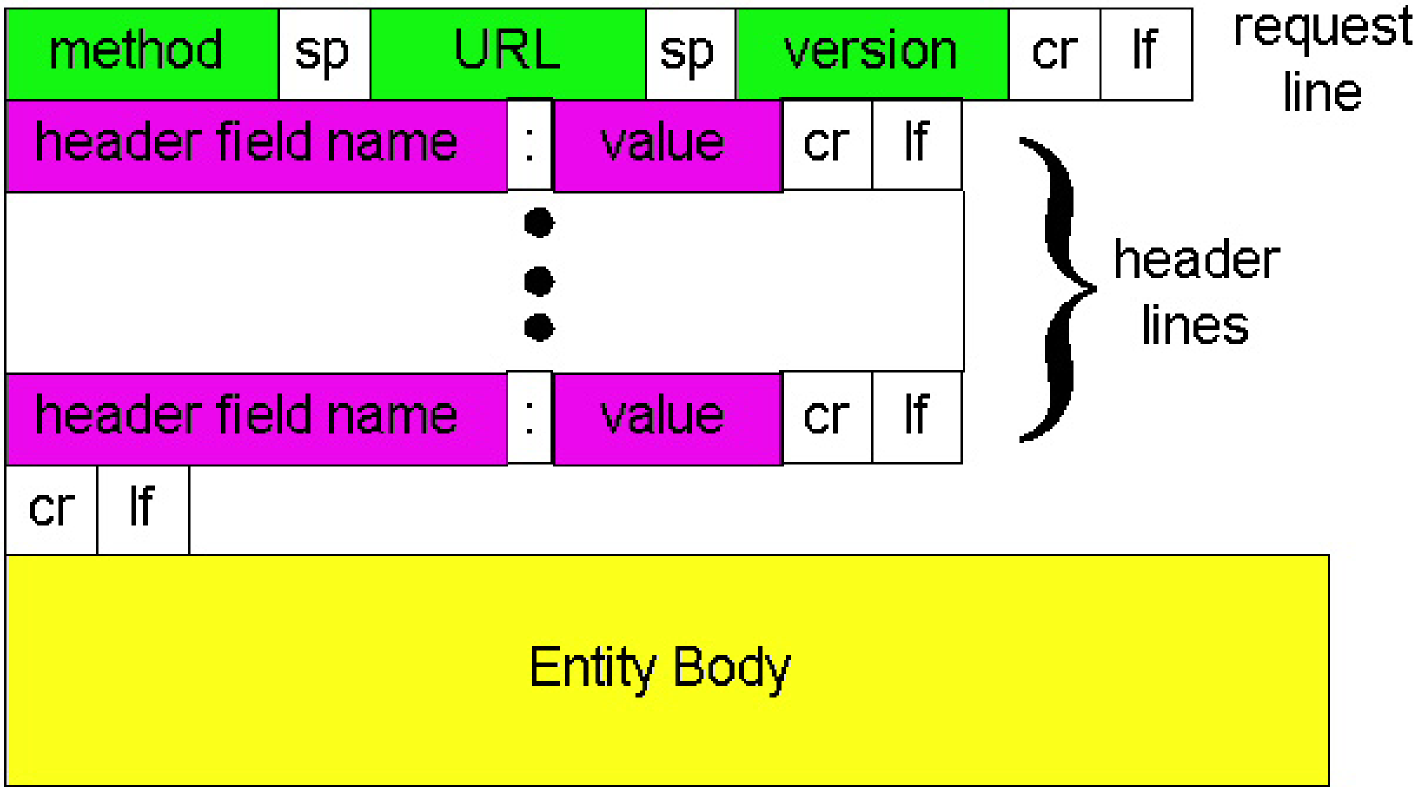
```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif,image/jpeg
Accept-language:fr
```

header lines

(extra carriage return, line feed)

Carriage return, line feed indicates end of message

http request message: general format



http request message: more info

- http/1.0 has only three request *methods*
  - *GET:*
  - *POST:* for forms. Uses *Entity Body* to transfer form info
  - *HEAD:* Like *GET* but response does not actually return any info. This is used for debugging/test purposes

- http/1.1 has two additional request *methods*
  - *PUT:* Allows uploading object to web server
  - *DELETE:* Allows deleting object from web server

http message format: respone

status line
(protocol
status code
status phrase)

status code

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`

- request succeeded, requested object later in this message

`301 Moved Permanently`

- requested object moved, new location specified later in this message (Location:)

`400 Bad Request`

- request message not understood by server

`404 Not Found`

- requested document not found on this server

`505 HTTP Version Not Supported`

Trying out http (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

Opens TCP connection to port 80 (default http server port) at cis.poly.edu. Anything typed in sent

to port 80 at cis.poly.edu.

2. Type in a GET http request:

```
GET /~ross/index.html HTTP/1.0
```

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!

Try telnet www.cs.ust.hk 80

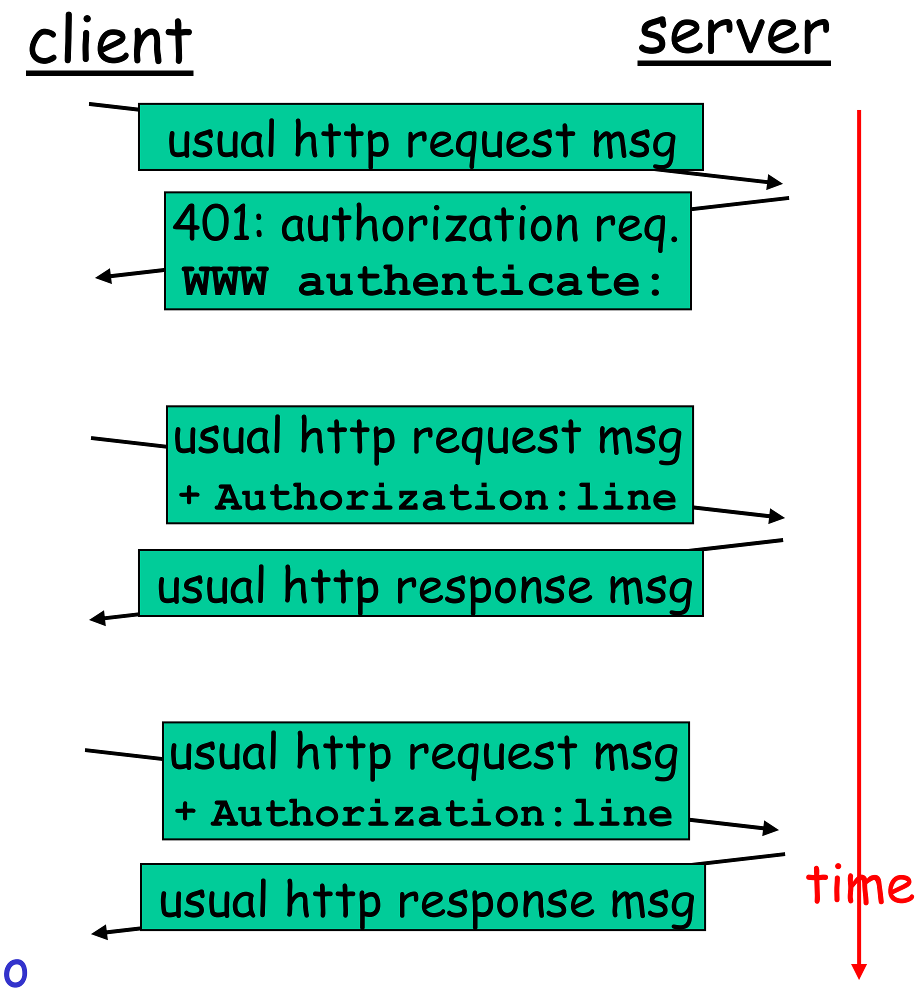
User-server interaction: authentication

Authentication goal: control access to server documents

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

    `WWW authenticate:`

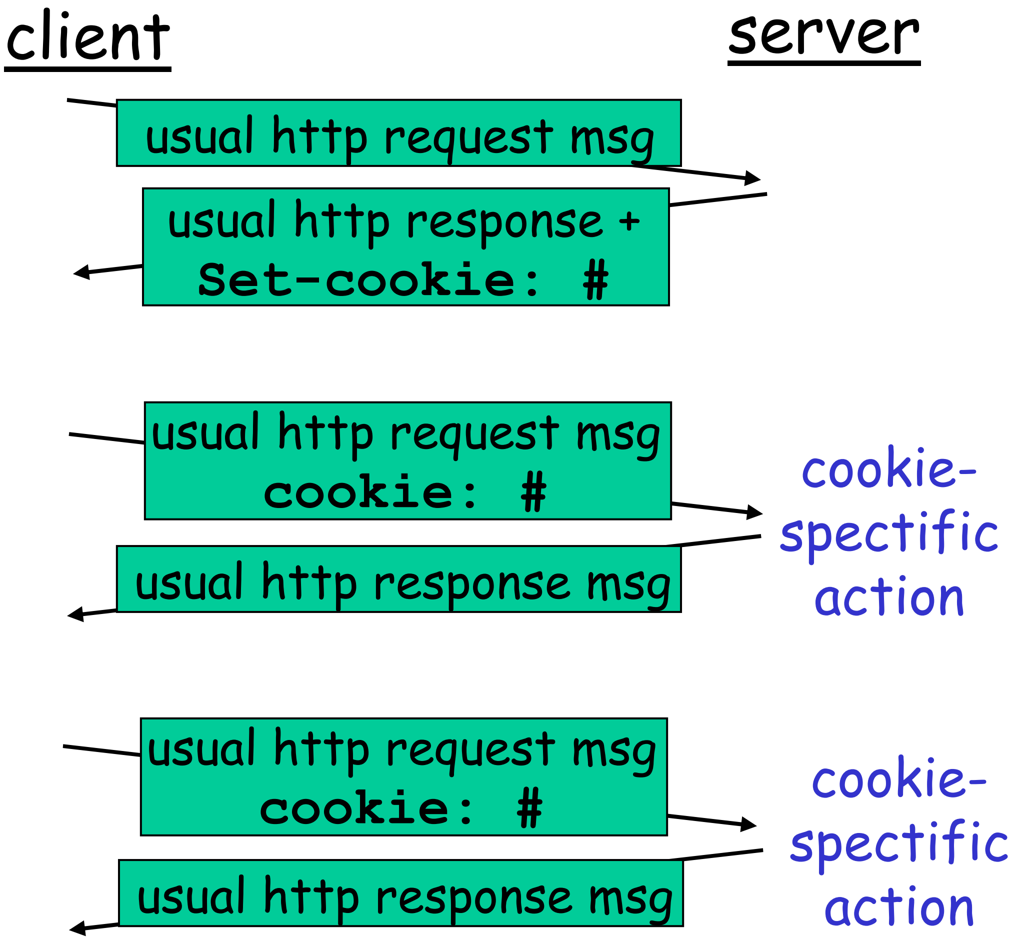
    header line in response



Browser caches name & password so that user does not have to repeatedly enter it.

User-server interaction: cookies

- server sends "cookie" to client in response msg
  `Set-cookie: 1678453`
- client stores & presents cookie in later requests
  `cookie: 1678453`
- server matches presented-cookie with server-stored info
  - authentication
  - remembering user preferences, previous choices



client

server

usual http request msg

usual http response +
Set-cookie: #

usual http request msg
cookie: #

usual http response msg

cookie-specific action

usual http request msg
cookie: #

usual http response msg

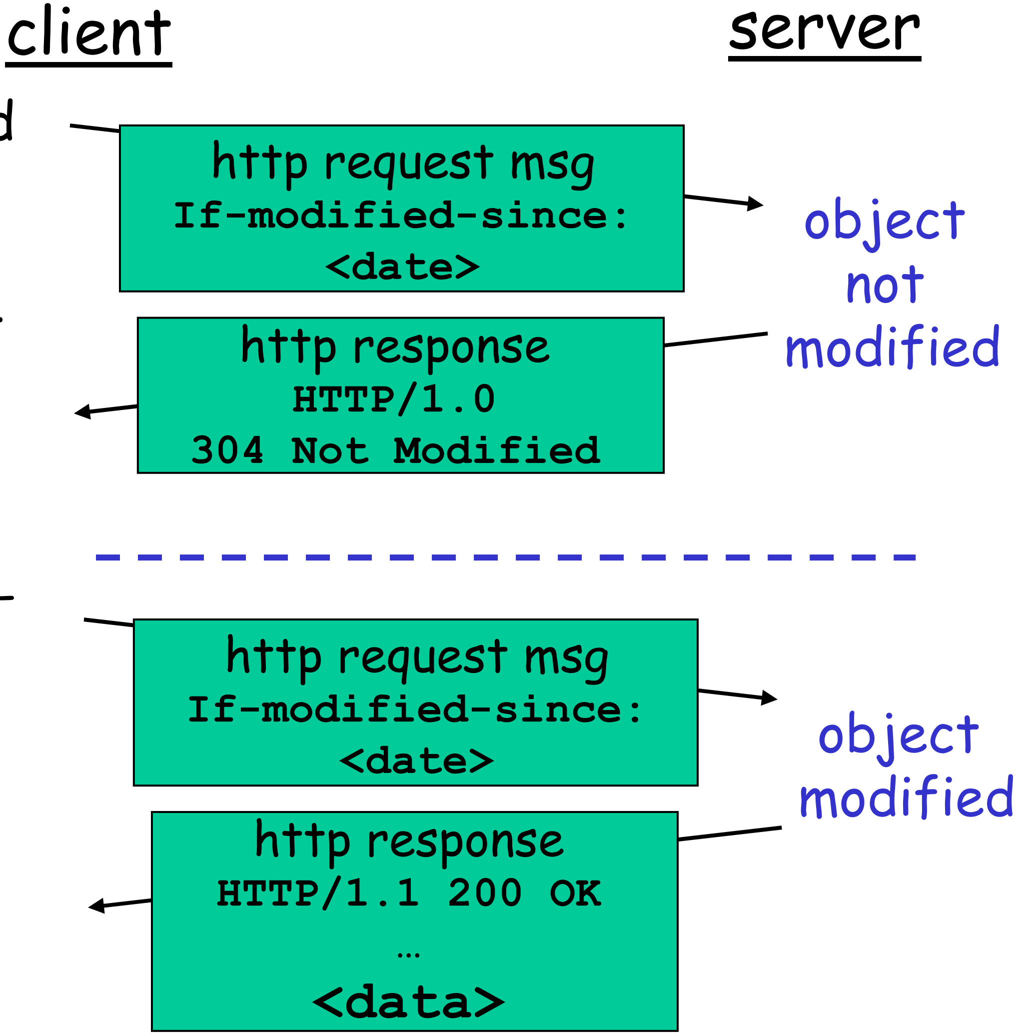
cookie-specific action

Cookie example

```
telnet www.google.com 80

Trying 216.239.33.99...
Connected to www.google.com.
Escape character is '^]'.

GET /index.html HTTP/1.0

HTTP/1.0 200 OK
Date: Wed, 10 Sep 2003 08:58:55 GMT
Set-Cookie:
    PREF=ID=43bd8b0f34818b58:TM=1063184203:LM=1063184203:
    S=DDqPgTb56Za88O2y; expires=Sun, 17-Jan-2038 19:14:07 GMT;
    path=/; domain=.google.com
.
.
```

User-server interaction: conditional GET

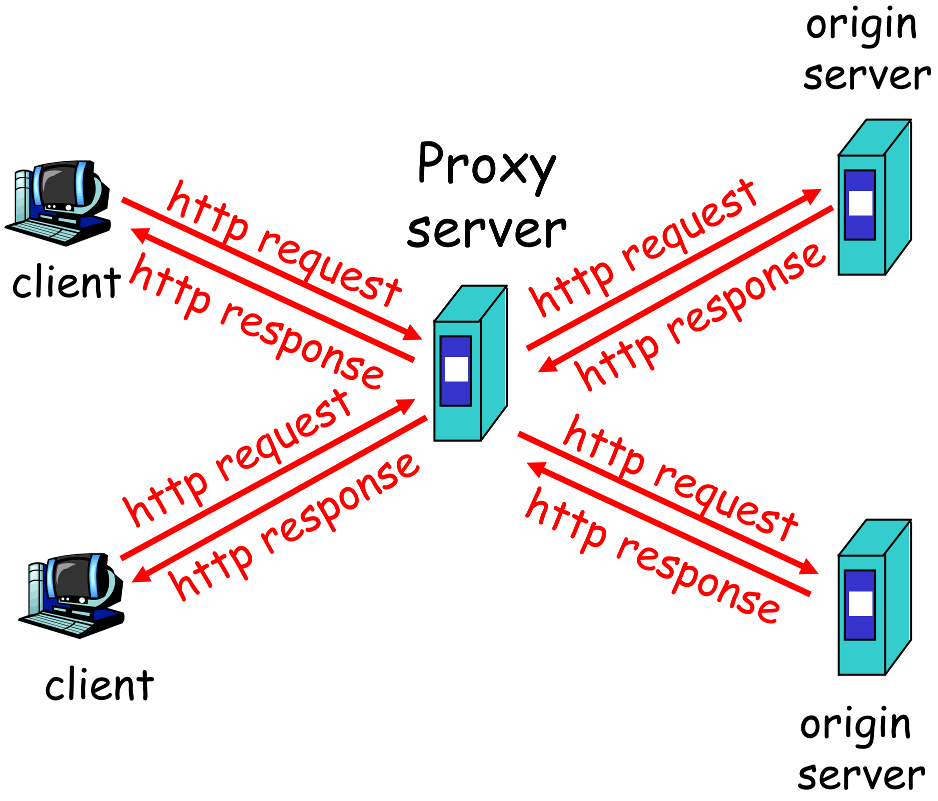
- Goal: don't send object if client has up-to-date stored (cached) version
- client: specify date of cached copy in http request
  `If-modified-since: <date>`
- server: response contains no object if cached copy up-to-date:
  `HTTP/1.0 304 Not Modified`



Web Caches (proxy server)

Goal: satisfy client request without involving origin server

- user sets browser: Web accesses via web cache
- client sends all http requests to web cache
  - if object at web cache, web cache immediately returns object in http response
  - else requests object from origin server, then returns http response to client



More about Web caching

- Cache acts as both client and server
- Cache can do up-to-date check using `If-modified-since` HTTP header
  - Issue: should cache take risk and deliver cached object without checking?
  - Heuristics are used.
- Typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

- Reduce response time for client request.
- Reduce traffic on an institution's access link.
- Internet dense with caches enables "poor" content providers to effectively deliver content

Caching example (1)

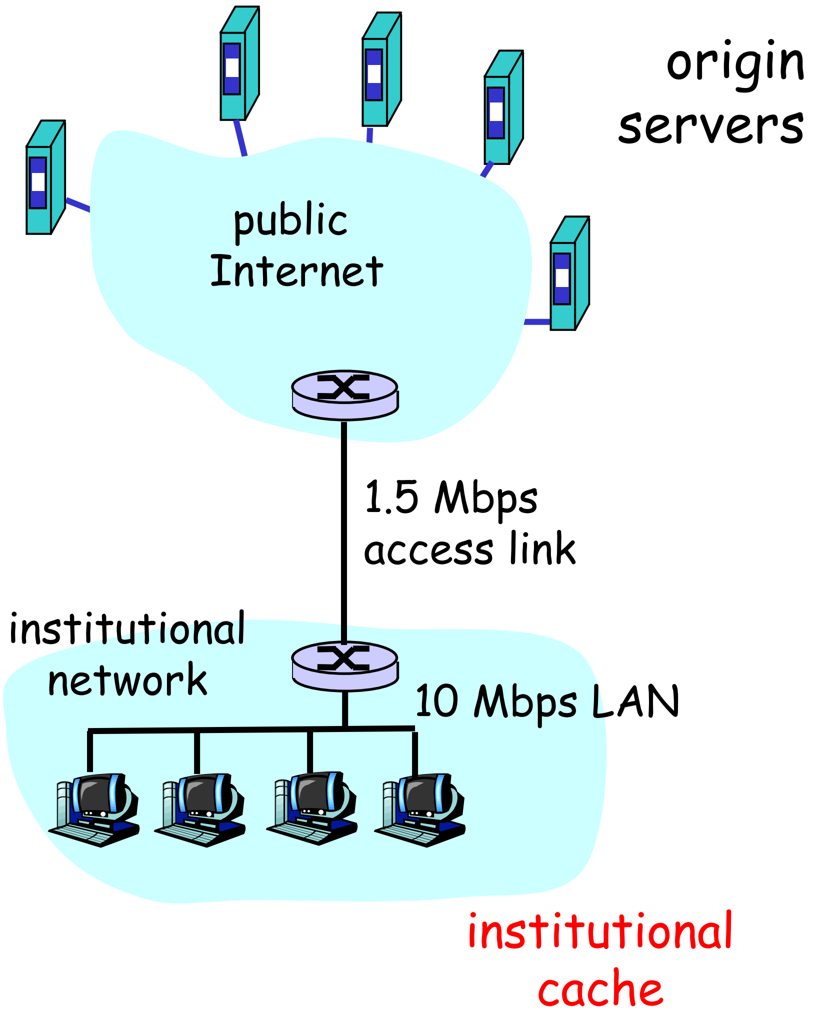
Assumptions

- average object size = 100,000 bits
- avg. request rate from institution's browser to origin serves = 15/sec
- delay from institutional router to any origin server and back to router = 2 sec

Consequences

- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay

= 2 sec + minutes + milliseconds



Caching example (2)

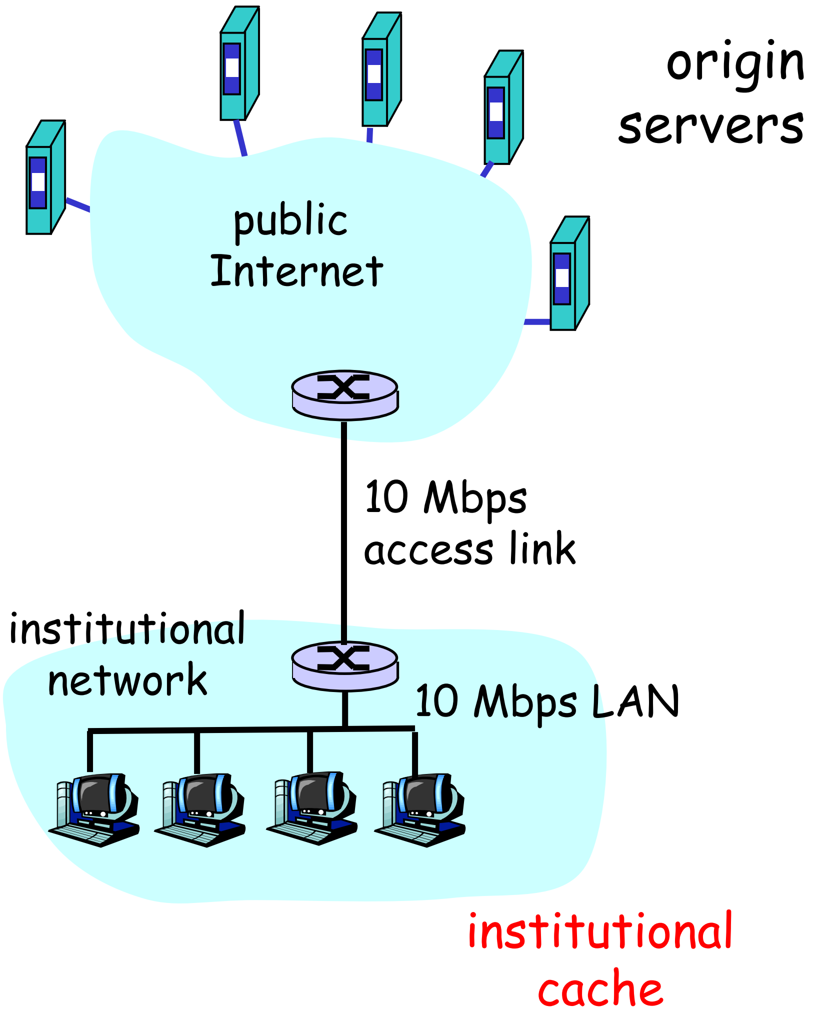
Possible solution

- increase bandwidth of access link to, say, 10 Mbps

Consequences

- utilization on LAN = 15%
- utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay

  = 2 sec + msecs + msecs
- often a costly upgrade



Caching example (3)

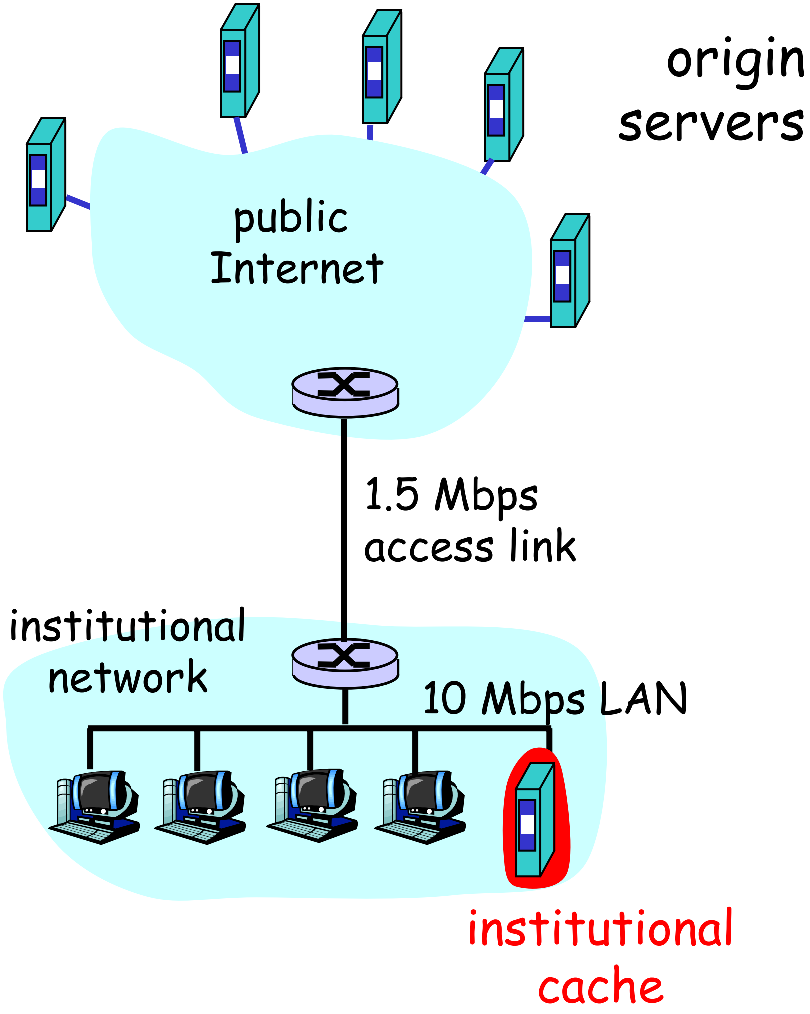
Install cache

- suppose hit rate is .4

Consequence

- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
- total delay = Internet delay + access delay + LAN delay

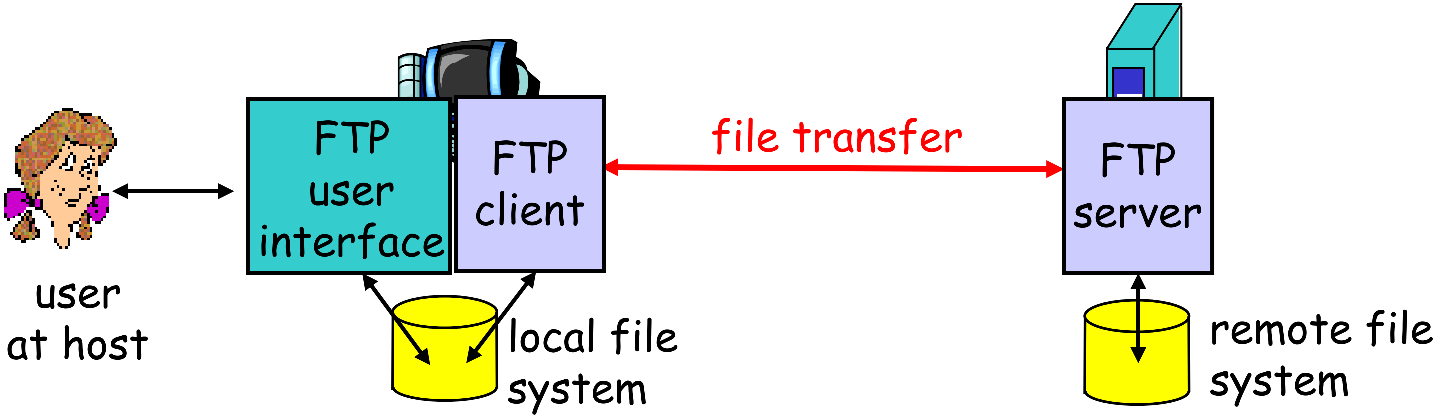
= .6*2 sec + .6*.01 secs + milliseconds < 1.3 secs



Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

ftp: the file transfer protocol

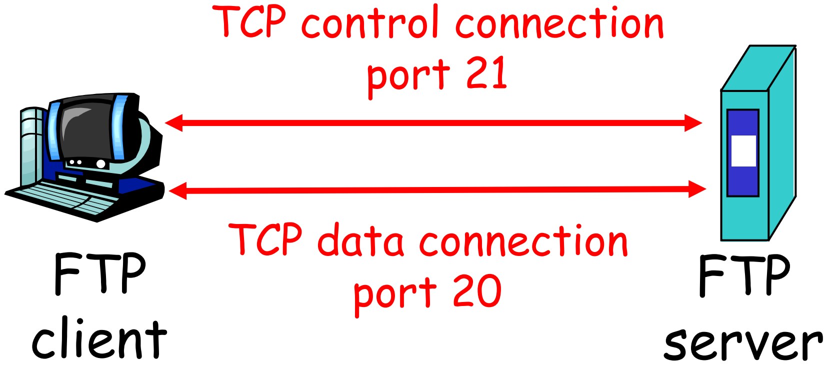


- transfer file to/from remote host
- client/server model
  - *client:* side that initiates transfer (either to/from remote)
  - *server:* remote host
- ftp: RFC 959
- ftp server: port 21

ftp: separate control, data connections

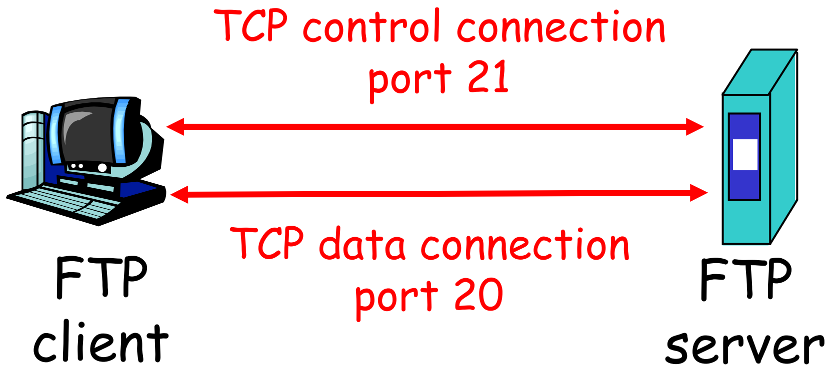
- ftp client contacts ftp server at port 21, specifying TCP as transport protocol
- two parallel TCP connections opened:
  - control: exchange commands, responses between client, server.

    "out of band control"
  - data: file data to/from server
- ftp server maintains "state": current directory, earlier authentication



ftp: separate control, data connections

- When server receives request for file transfer it opens a TCP data connection to client on port 20.
- After transferring one file, server closes connection
- When next request for file transfer arrives server opens new TCP data connection on port 20



ftp commands, responses

Sample commands:

- sent as ASCII text over control channel
- **`USER`** ***`username`***
- **`PASS`** ***`password`***
- **`LIST`** return list of file in current directory
- **`RETR filename`** retrieves (gets) file
- **`STOR filename`** stores (puts) file onto remote host

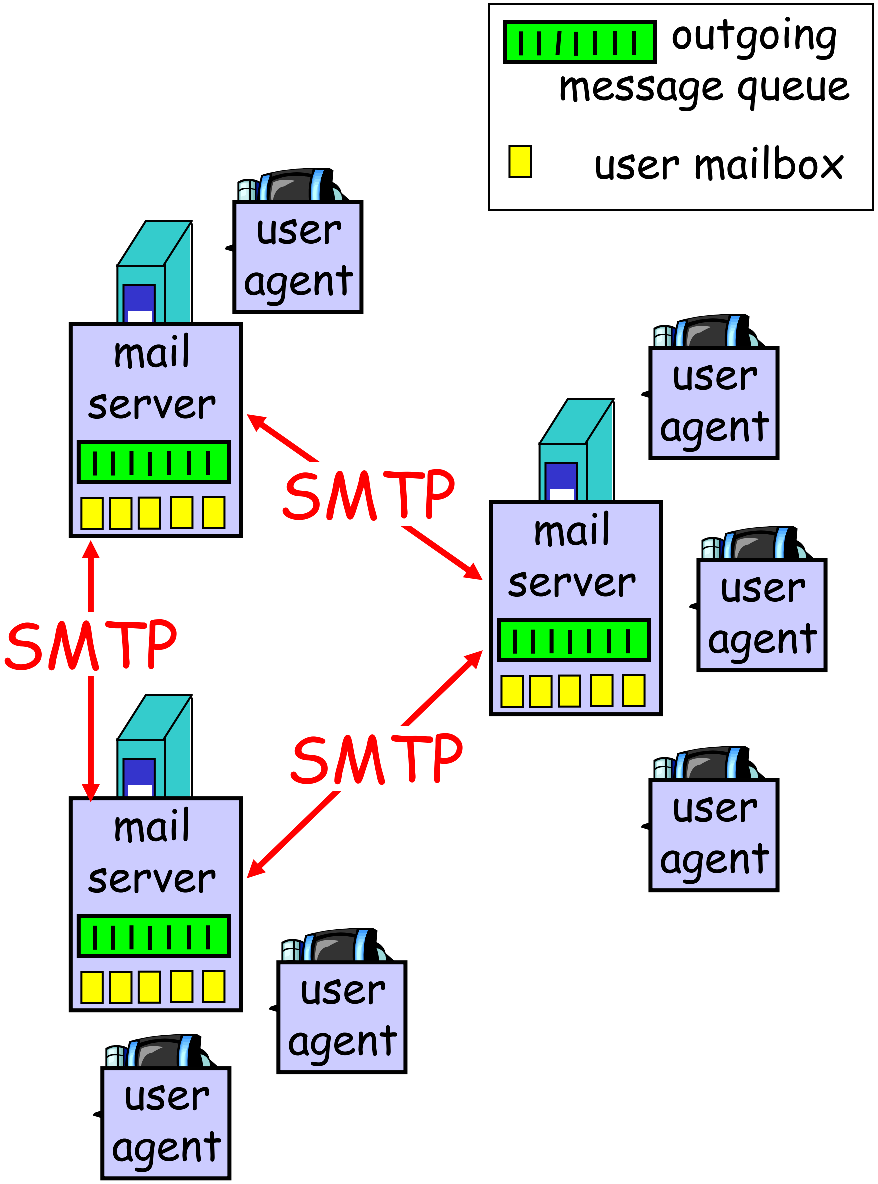
Sample return codes

- status code and phrase (as in http)
- **`331 Username OK, password required`**
- **`125 data connection already open; transfer starting`**
- **`425 Can't open data connection`**
- **`452 Error writing file`**

Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

Electronic Mail



Three major components:

- user agents
- mail servers
- simple mail transfer protocol: smtp

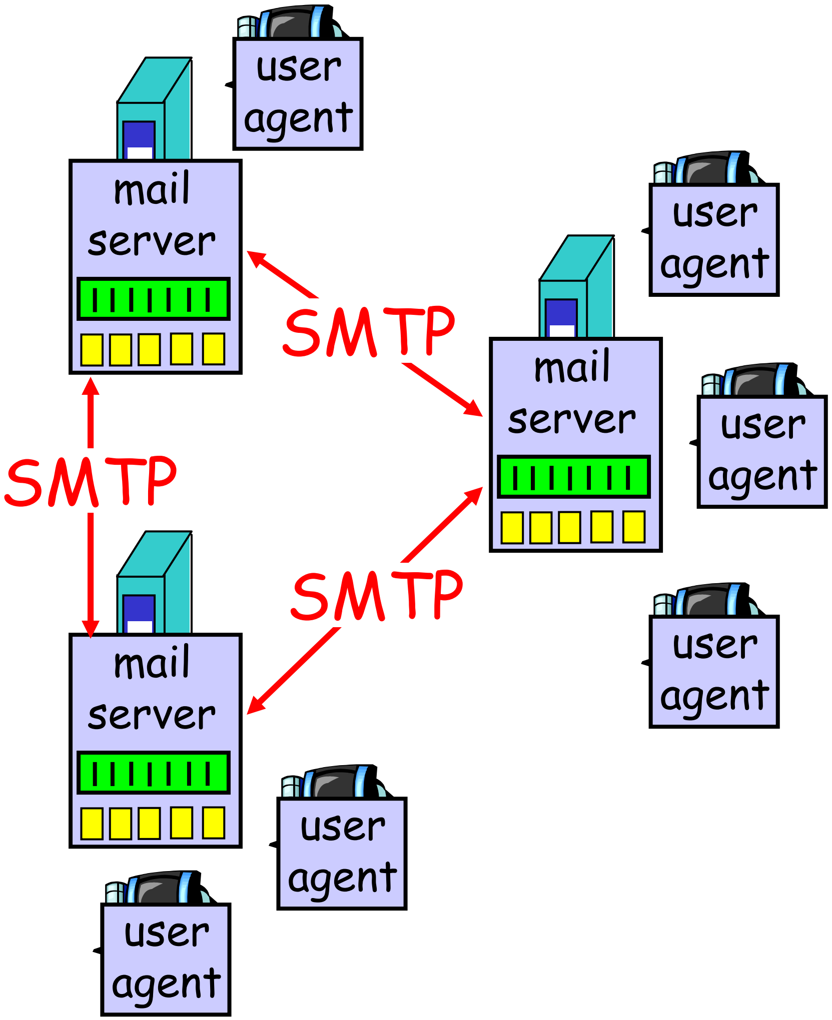
User Agent

- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- outgoing, incoming messages stored on server

Electronic Mail: mail servers

Mail "Servers"

- mailbox contains incoming messages (yet to be read) 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



Electronic Mail: smtp [RFC 821]

- uses tcp to reliably transfer email msg from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message and "to" `bob@someschool.edu`

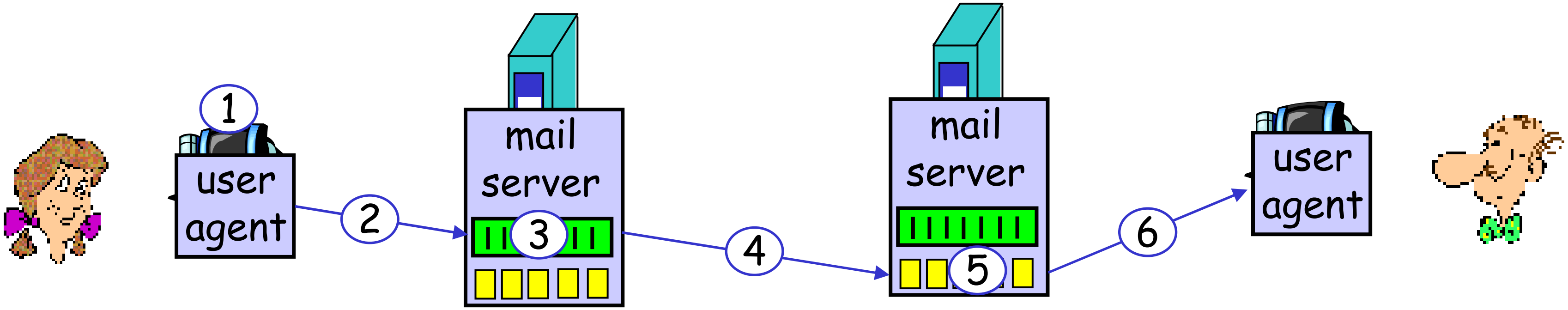
2) Alice's UA sends message to her mail server; message placed in message queue

3) Client side of SMTP opens TCP connection with Bob's mail server

4) SMTP client sends Alice's message over the TCP connection

5) Bob's mail server places the message in Bob's mailbox

6) Bob invokes his user agent to read message



Sample smtp interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C:   How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

Try SMTP interaction for yourself:

- `telnet servername 25`
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client (reader)

smtp: final words

- smtp uses persistent connections
- smtp requires that message (header & body) be in 7-bit ascii
- certain character strings are not permitted in message (e.g., `CRLF.CRLF`). Thus message has to be encoded (usually into either base-64 or quoted printable)
- smtp server uses `CRLF.CRLF` to determine end of message

Comparison with http

- http: pull
- email: push
- both have ASCII command/response interaction, status codes
- http: each object is encapsulated in its own response message
- smtp: multiple objects message sent in a multipart message

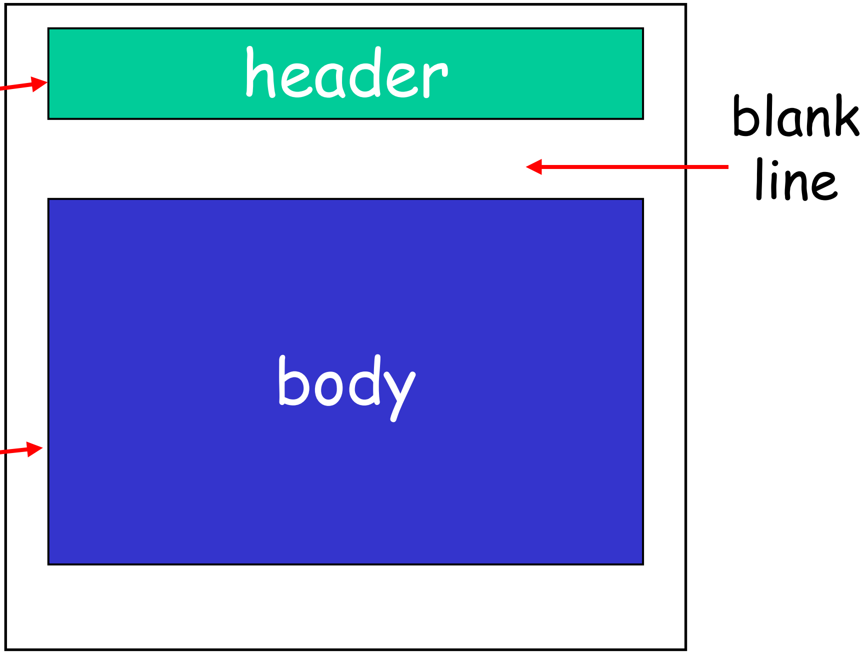
Mail message format

smtp: protocol for exchanging email msgs

RFC 822: standard for text message format:

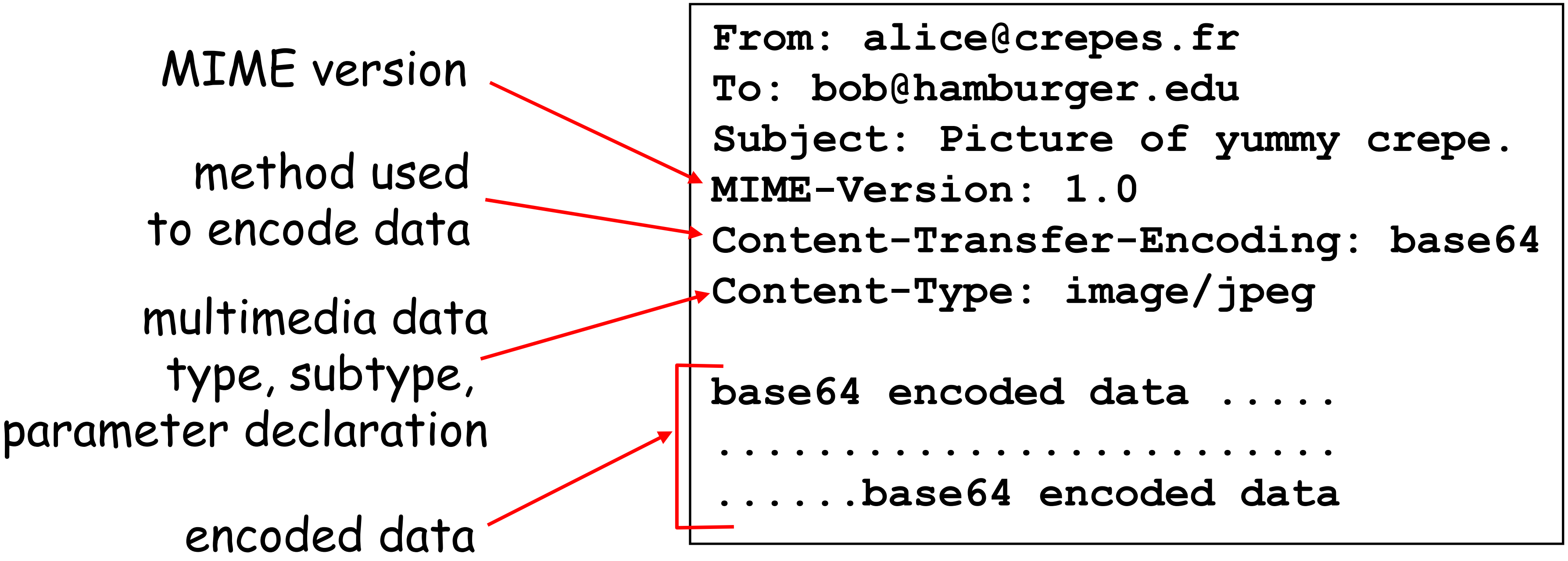
- header lines, e.g.,
  - To:
  - From:
  - Subject:

  different from smtp commands!
- body
  - the "message", ASCII characters only



Message format: multimedia extensions

- MIME: (Multipurpose Internet Mail Extensions) multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type



MIME version

method used to encode data

multimedia data type, subtype, parameter declaration

encoded data

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
```

MIME types

`Content-Type: type/subtype; parameters`

Text

- example subtypes: **`plain`**, **`html`**

Image

- example subtypes: **`jpeg`**, **`gif`**

Audio

- exampe subtypes: **`basic`** (8-bit mu-law encoded), **`32kadpcm`** **(32 kbps coding)**

Video

- example subtypes: **`mpeg`**, **`quicktime`**

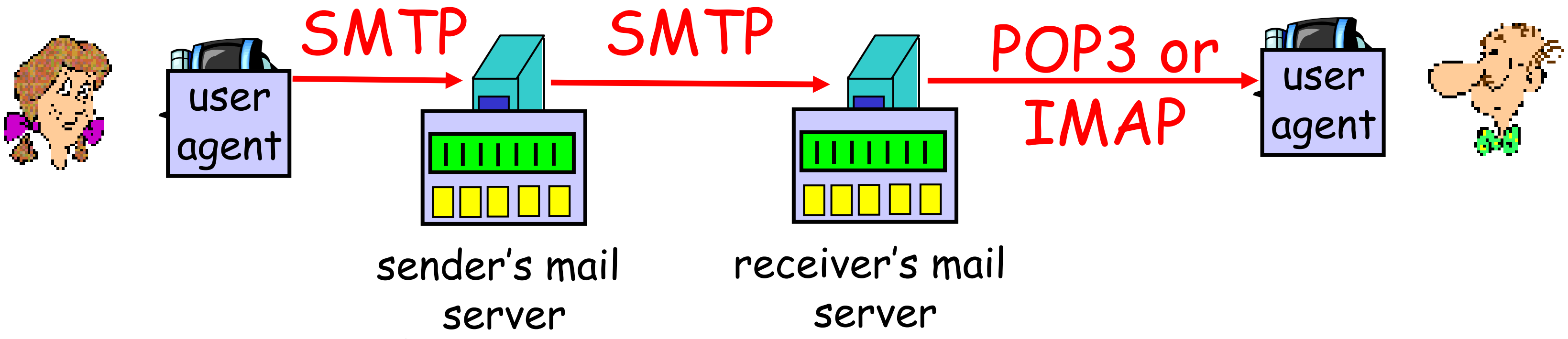
Application

- other data that must be processed by reader before "viewable"
- example subtypes: **`msword`**, **`octet-stream`**

Multipart Type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=StartOfNextPart

--StartOfNextPart
Dear Bob, Please find a picture of a crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
base64 encoded data .....
.........................
......base64 encoded data
--StartOfNextPart
Do you want the recipe?
```

Mail access protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - HTTP: Hotmail , Yahoo! Mail, etc.

POP3 protocol

authorization phase

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

```
S: +OK POP3 server ready
C: user alice
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

transaction phase, client:

- `list`: list message numbers
- `retr`: retrieve message by number
- `dele`: delete
- `quit`

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 (more) and IMAP

More about POP3

- Previous example uses "download and delete" mode.
- Bob cannot re-read e-mail if he changes client
- "Download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

IMAP

- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name

Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

DNS: Domain Name System

People: many identifiers:

- SSN, name, Passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., gaia.cs.umass.edu - used by humans

Q: map between IP addresses and name ?

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
  - note: core Internet function implemented as application-layer protocol
  - complexity at network's "edge"

DNS name servers

Why not centralize DNS?

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

doesn't *scale!*

- no server has all name-to-IP address mappings

local name servers:

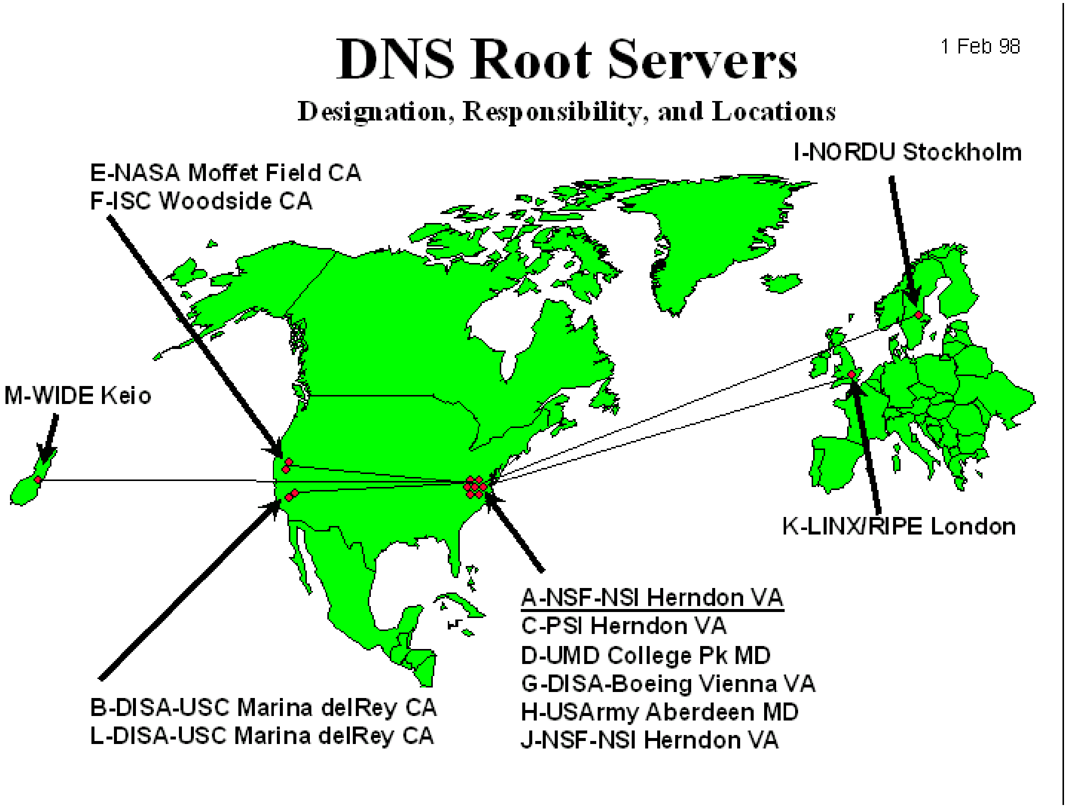
- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

authoritative name server:

- for a host: stores that host's IP address, name
- can perform name/address translation for that host's name

DNS: Root name servers

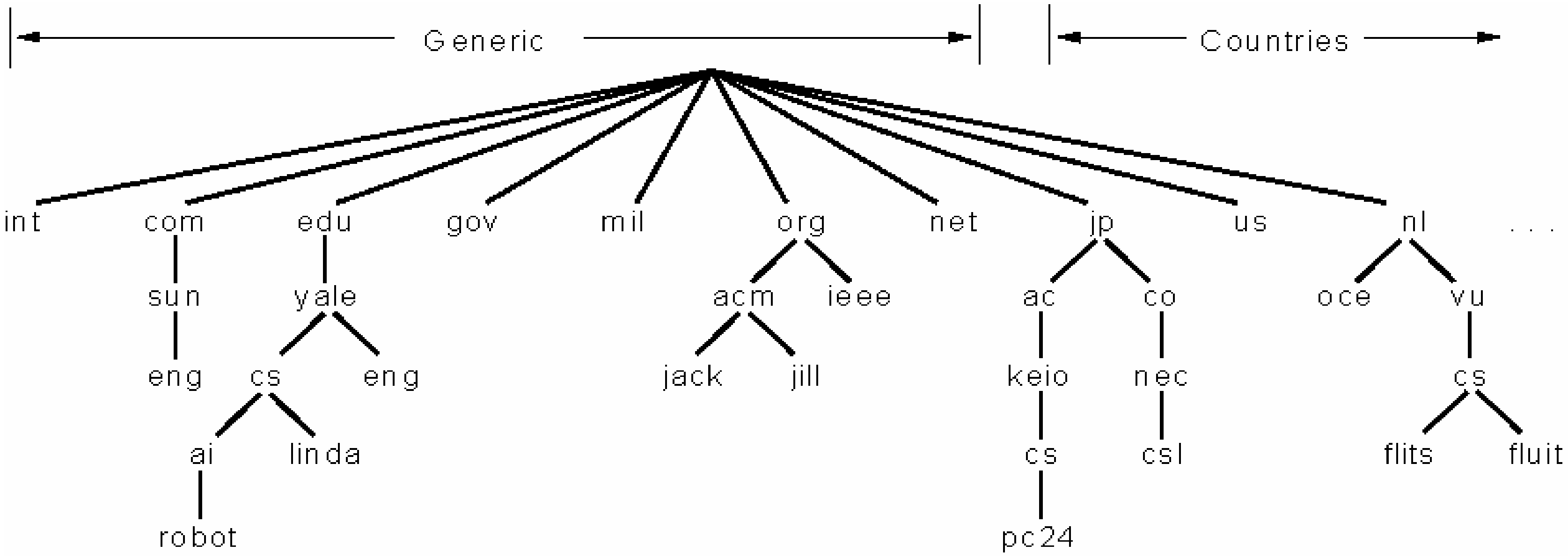
- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server
- ~ dozen root name servers worldwide



2. DNS

- Defined in RFCs 1034 and 1035.
- Hierarchical, domain-based naming scheme, and uses distributed database system.

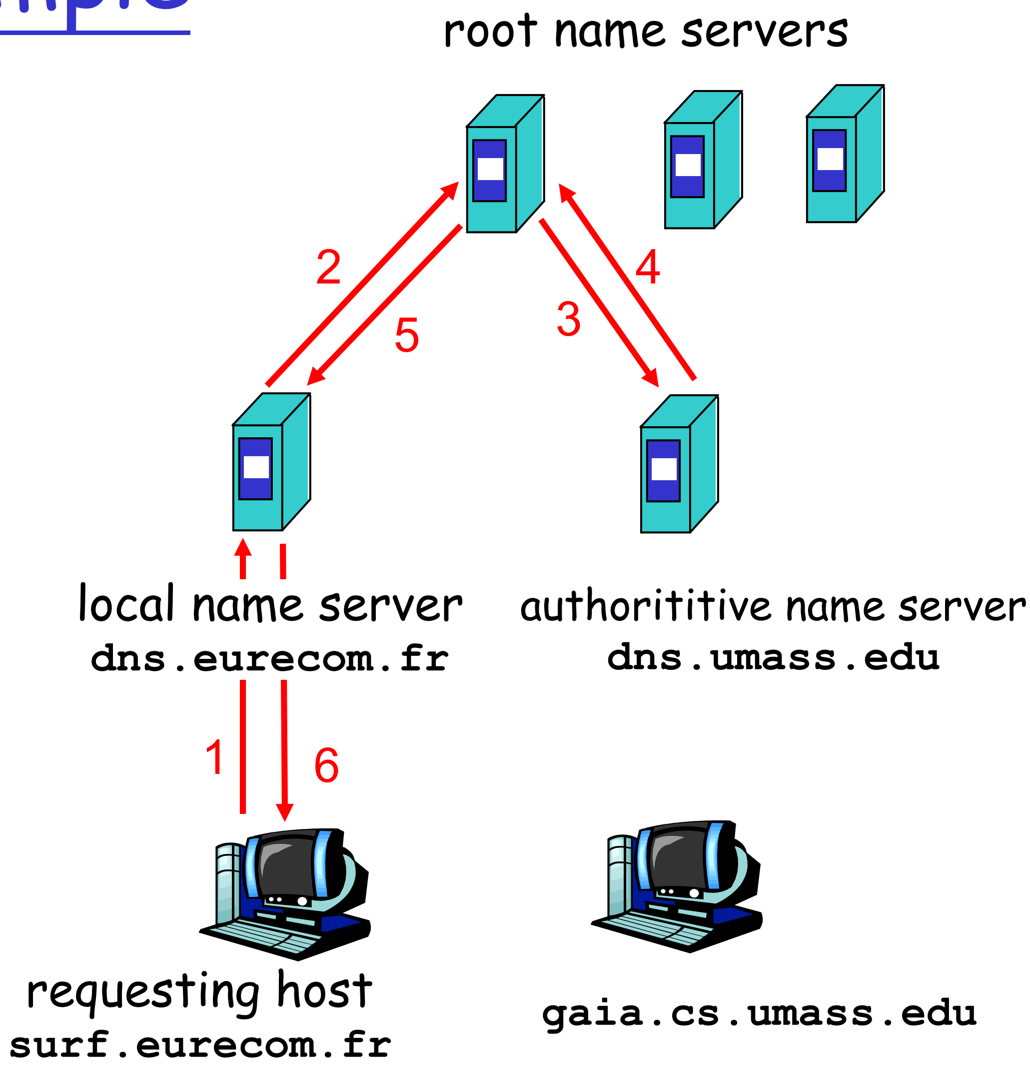
Illustration from Tanenbaum



Simple DNS example

host **`surf.eurecom.fr`** wants IP address of **`gaia.cs.umass.edu`**

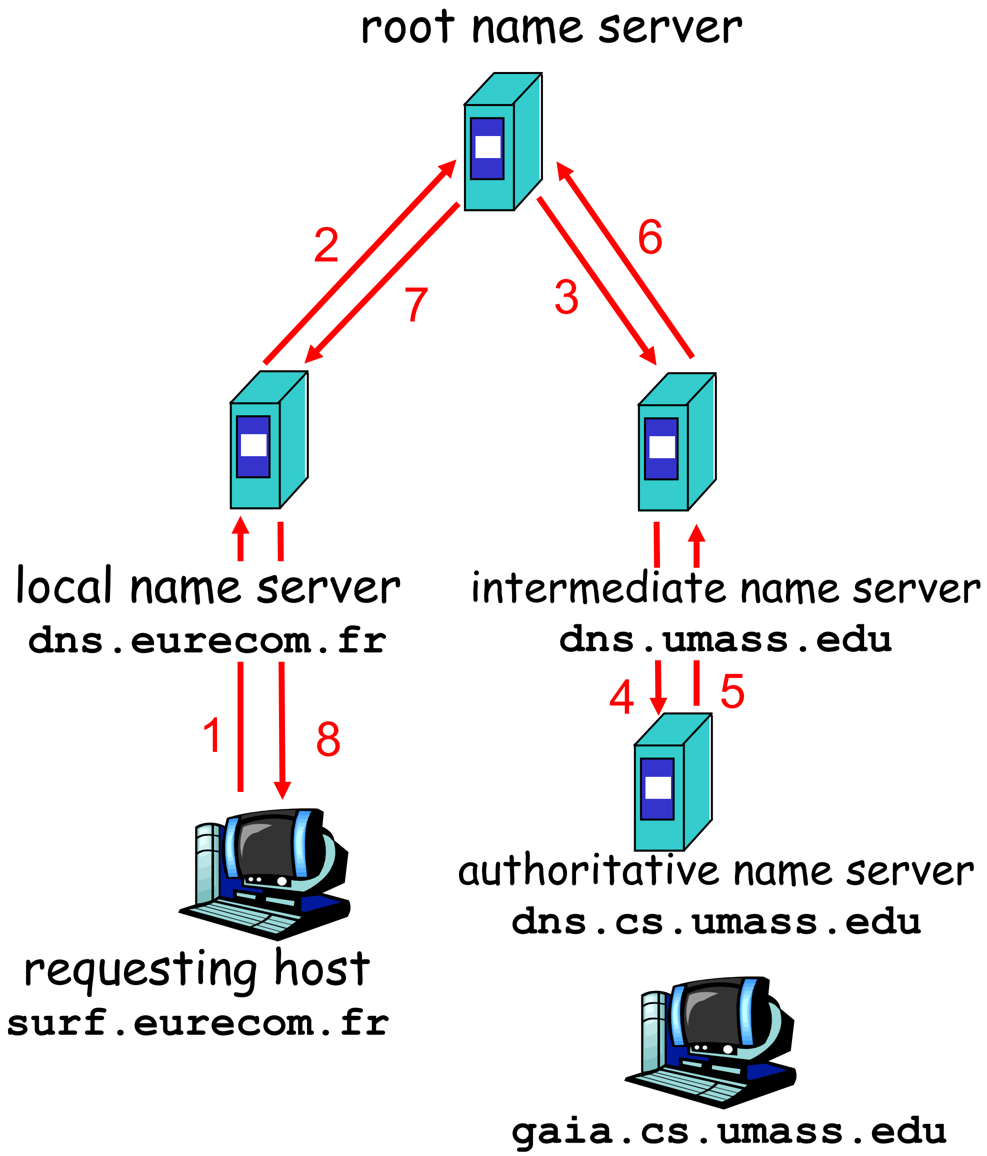
1. Contacts its local DNS server, **`dns.eurecom.fr`**
2. **`dns.eurecom.fr`** contacts root name server, **if necessary**
3. root name server contacts authoritative name server, **`dns.umass.edu`**, **if necessary**



DNS example

Root name server:

- may not know authoritative name server
- may know *intermediate name server:* who to contact to find authoritative name server



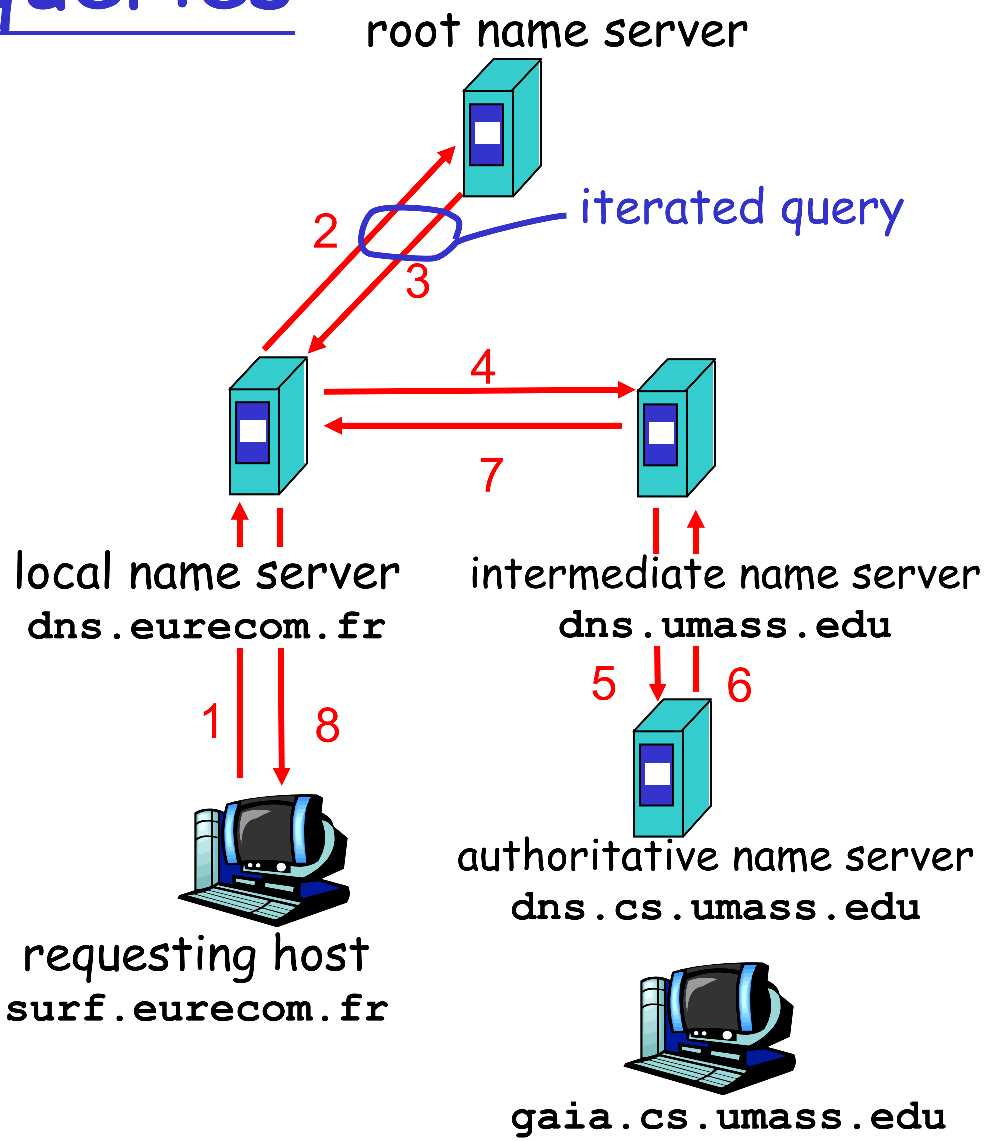
DNS: iterated queries

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: caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time
- update/notify mechanisms under design by IETF
  - RFC 2136
  - http://www.ietf.org/html.charters/dnsind-charter.html

DNS records

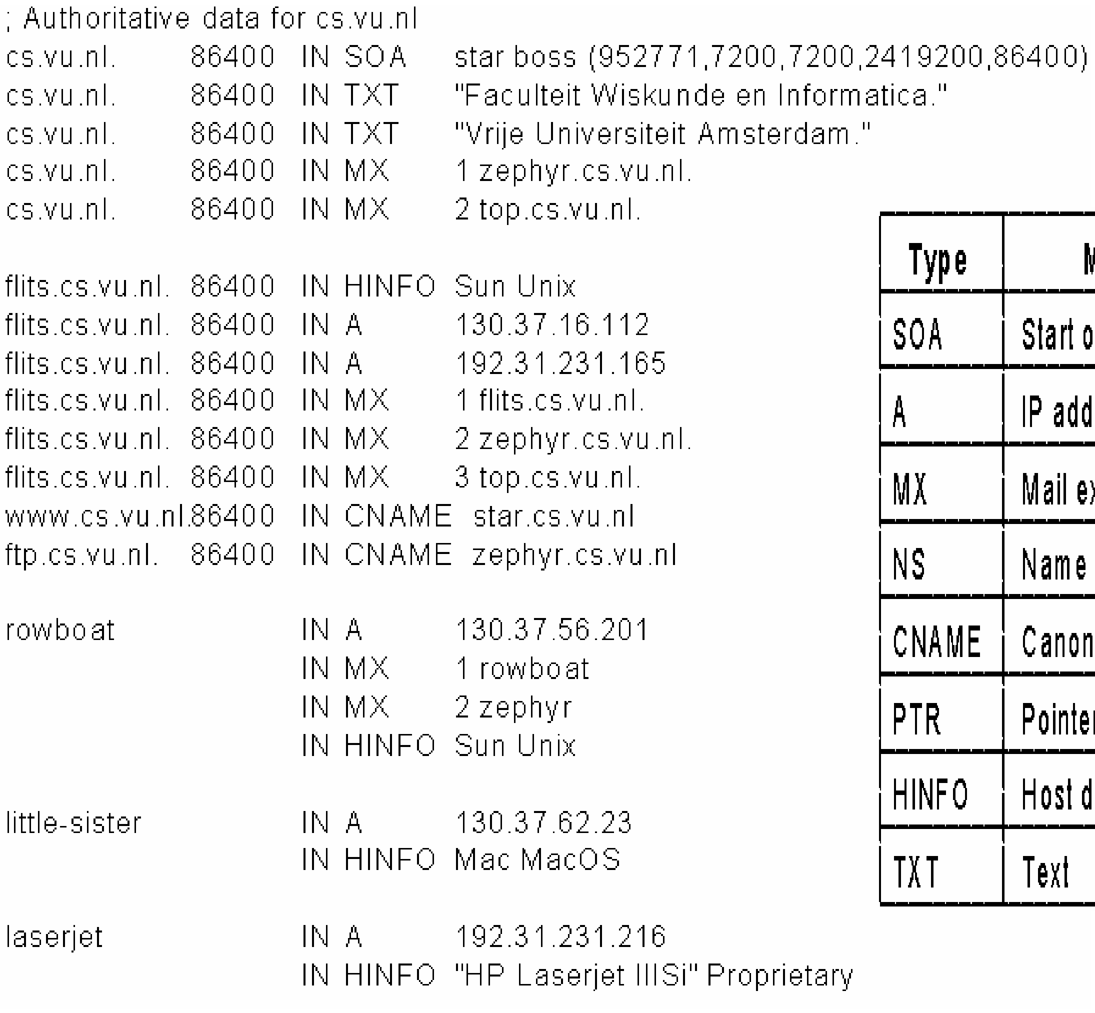
DNS: distributed db storing resource records (RR)

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

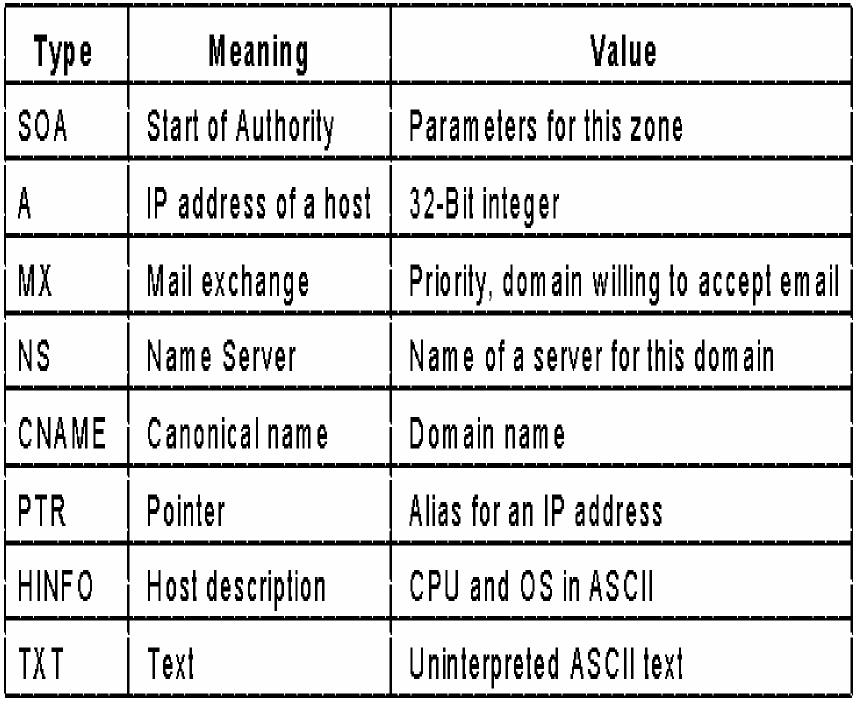
- Type=A
  - **`name`** is hostname
  - **`value`** is IP address
- Type=NS
  - **`name`** is domain (e.g. foo.com)
  - **`value`** is IP address of authoritative name server for this domain
- Type=CNAME
  - **`name`** is an alias name for some "cannonical" (the real) name
  - **`value`** is cannonical name
- Type=MX
  - **`value`** is hostname of mailserver associated with **`name`**

2. Resource Record

From Tanenbaum



```
; Authoritative data for cs.vu.nl
cs.vu.nl.       86400  IN  SOA    star boss (952771,7200,7200,2419200,86400)
cs.vu.nl.       86400  IN  TXT    "Faculteit Wiskunde en Informatica."
cs.vu.nl.       86400  IN  TXT    "Vrije Universiteit Amsterdam."
cs.vu.nl.       86400  IN  MX     1 zephyr.cs.vu.nl.
cs.vu.nl.       86400  IN  MX     2 top.cs.vu.nl.

flits.cs.vu.nl. 86400  IN  HINFO  Sun Unix
flits.cs.vu.nl. 86400  IN  A      130.37.16.112
flits.cs.vu.nl. 86400  IN  A      192.31.231.165
flits.cs.vu.nl. 86400  IN  MX     1 flits.cs.vu.nl.
flits.cs.vu.nl. 86400  IN  MX     2 zephyr.cs.vu.nl.
flits.cs.vu.nl. 86400  IN  MX     3 top.cs.vu.nl.
www.cs.vu.nl.86400     IN  CNAME  star.cs.vu.nl
ftp.cs.vu.nl.   86400  IN  CNAME  zephyr.cs.vu.nl

rowboat                IN  A      130.37.56.201
                       IN  MX     1 rowboat
                       IN  MX     2 zephyr
                       IN  HINFO  Sun Unix

little-sister          IN  A      130.37.62.23
                       IN  HINFO  Mac MacOS

laserjet               IN  A      192.31.231.216
                       IN  HINFO  "HP Laserjet IIISi" Proprietary
```



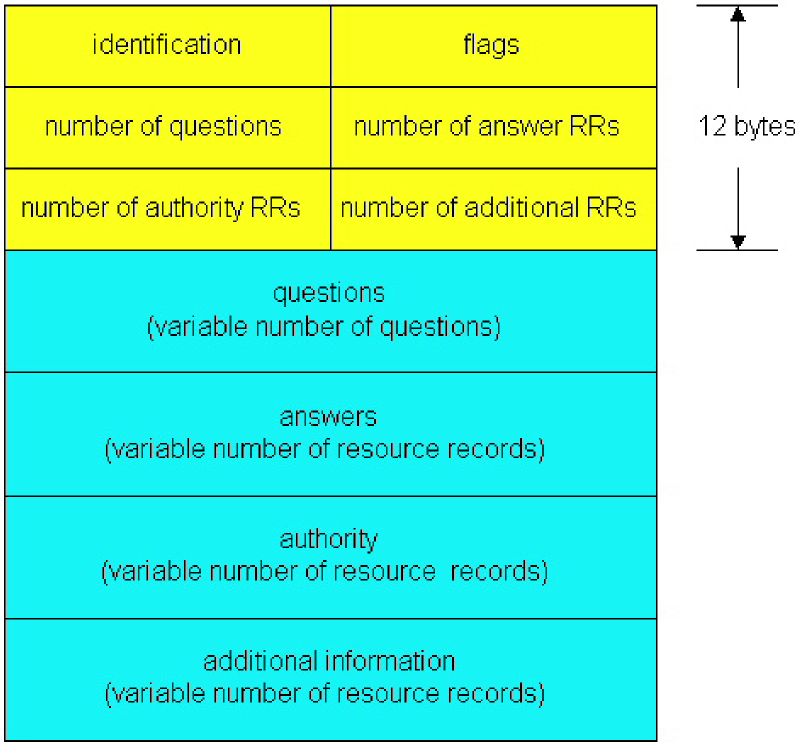
Type	Meaning	Value
SOA	Start of Authority	Parameters for this zone
A	IP address of a host	32-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name Server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
HINFO	Host description	CPU and OS in ASCII
TXT	Text	Uninterpreted ASCII text

DNS protocol, messages

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

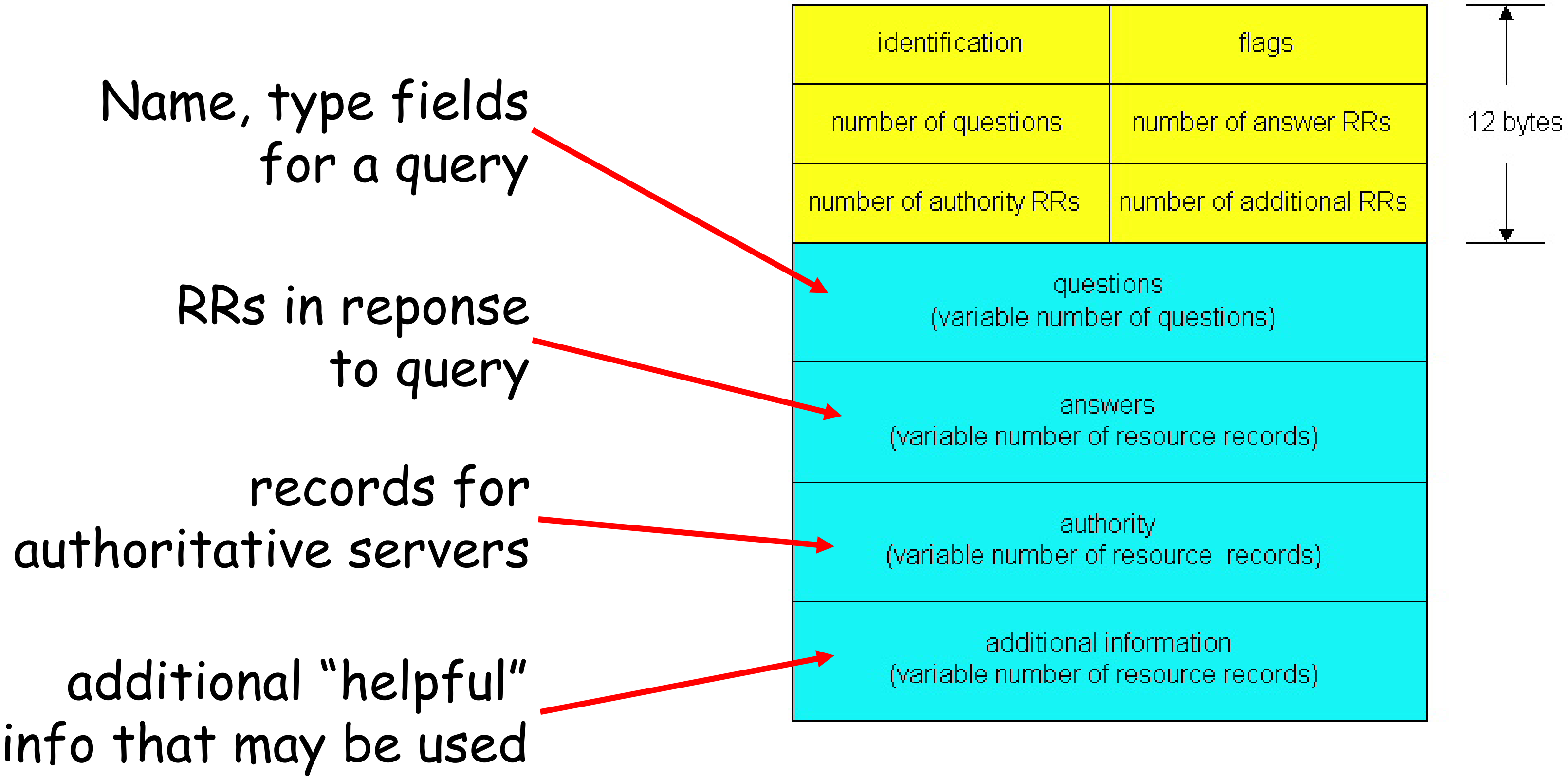
msg header

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



identification	flags	12 bytes
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)		

DNS protocol, messages



Name, type fields for a query

RRs in reponse 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

12 bytes

questions
(variable number of questions)

answers
(variable number of resource records)

authority
(variable number of resource records)

additional information
(variable number of resource records)

Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

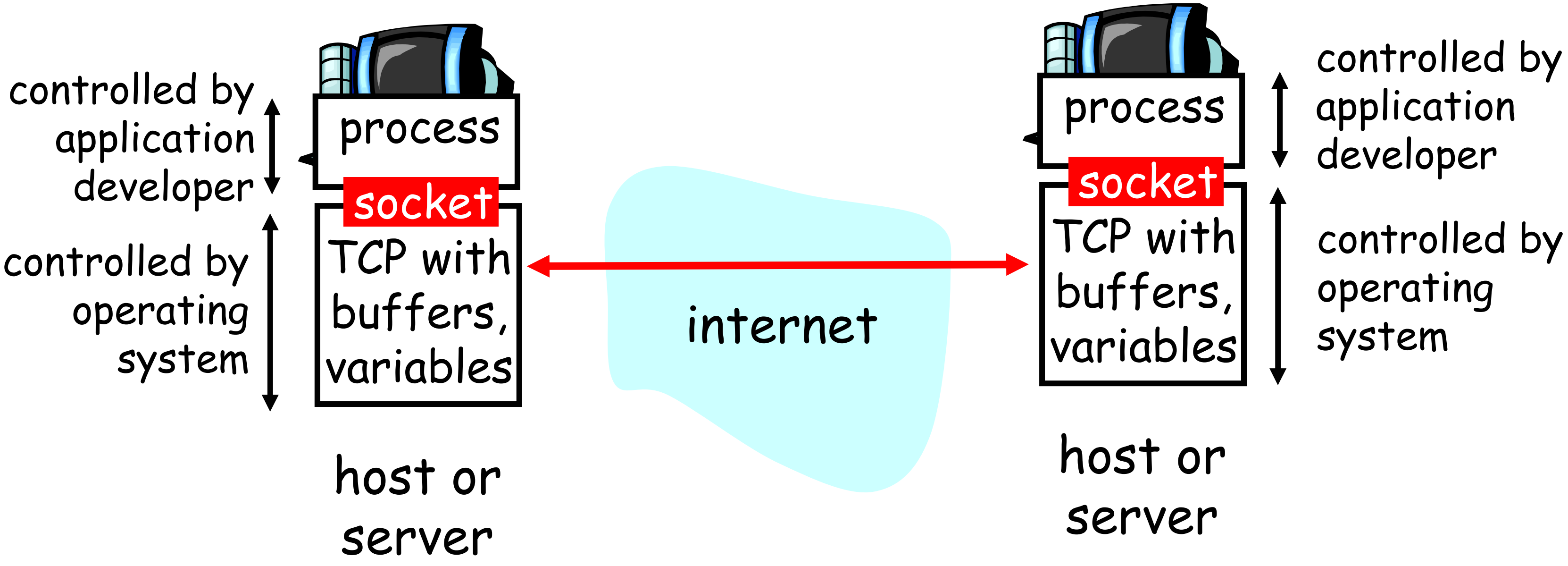
socket

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

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of **bytes** from one process to another



controlled by application developer

controlled by operating system

process

socket

TCP with buffers, variables

host or server

internet

process

socket

TCP with buffers, variables

host or server

controlled by application developer

controlled by operating system

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
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint

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

Stream jargon

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, eg, keyboard or socket.
- An output stream is attached to an output source, eg, monitor or socket.

Socket programming with TCP

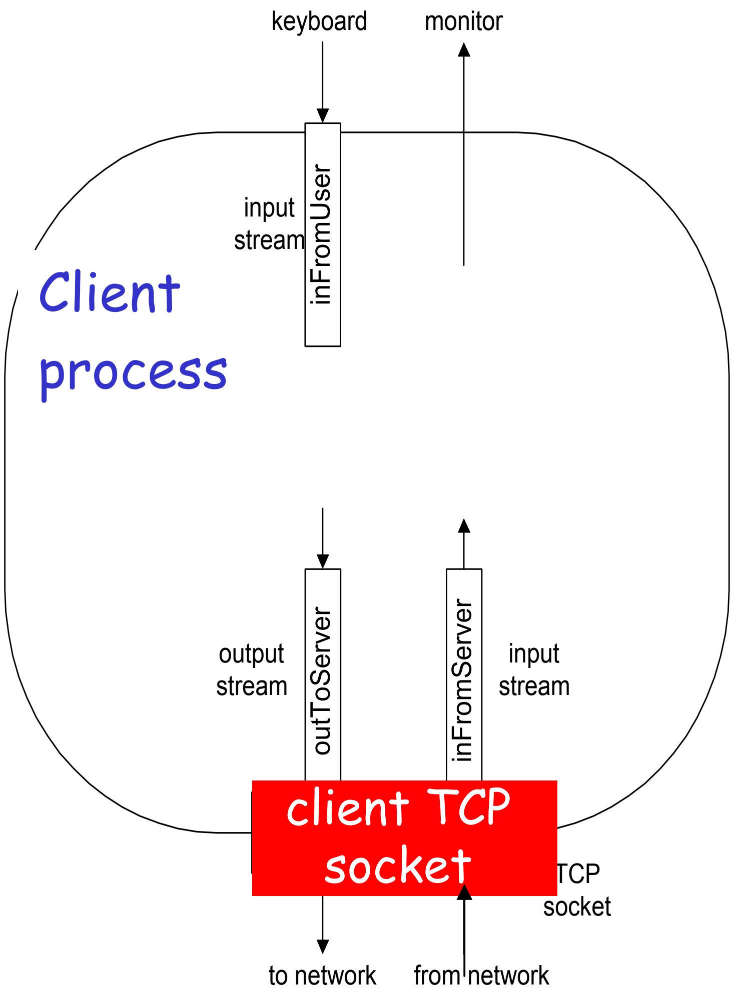
Example client-server app:

1) client reads line from standard input (`inFromUser` stream) , sends to server via socket (`outToServer` stream)

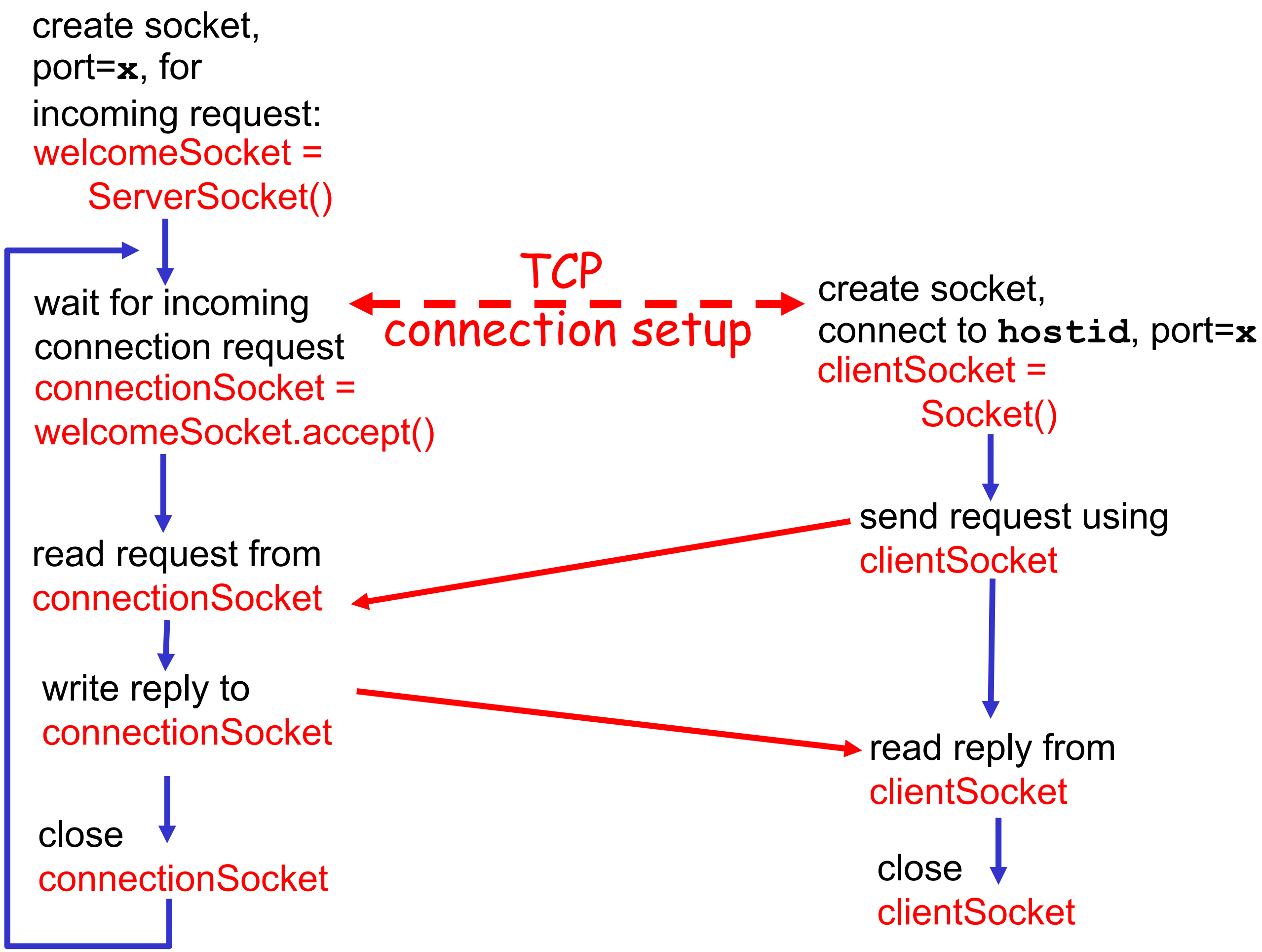
2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (`inFromServer` stream)



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

Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;
```

Create input stream

```
        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));
```

Create client socket, connect to server

```
        Socket clientSocket = new Socket("hostname", 6789);
```

Create output stream attached to socket

```
        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

Example: Java client (TCP), cont.

Create input stream attached to socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
```

Send line to server

```
outToServer.writeBytes(sentence + '\n');
```

Read line from server

```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

    }
}
```

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;
```

Create welcoming socket at port 6789

```
     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {
```

Wait, on welcoming socket for contact by client

```
        Socket connectionSocket = welcomeSocket.accept();
```

Create input stream, attached to socket

```
       BufferedReader inFromClient =
         new BufferedReader(new
         InputStreamReader(connectionSocket.getInputStream()));
```

Example: Java server (TCP), cont

Create output stream, attached to socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

Read in line from socket

```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Write out line to socket

```
outToClient.writeBytes(capitalizedSentence);
}
```

End of while loop, loop back and wait for another client connection

```
  }
}
```

Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

Socket programming *with UDP*

UDP: no "connection" between client and server

- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

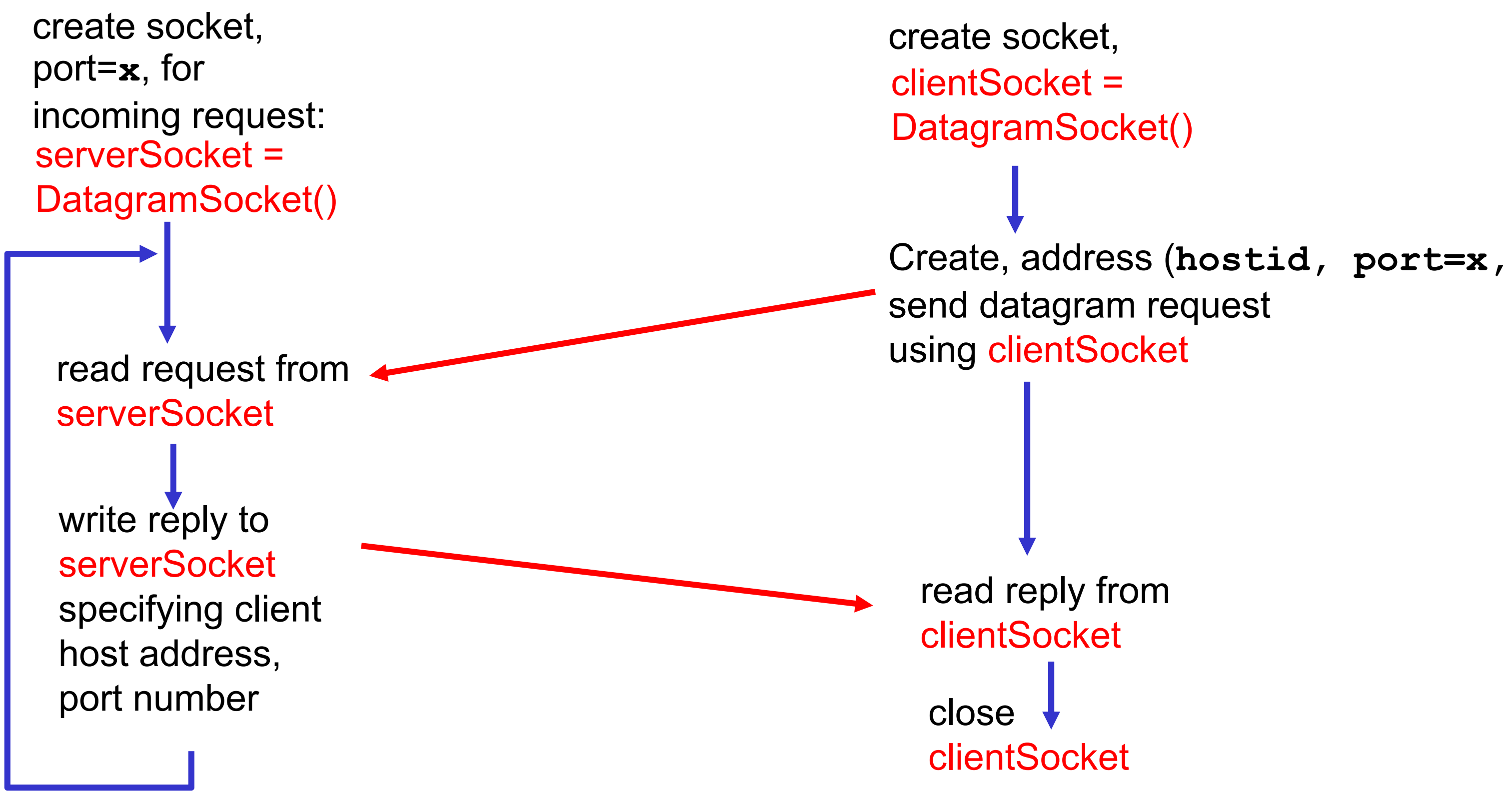
application viewpoint

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 number

create socket,
clientSocket = DatagramSocket()

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

read reply from clientSocket

close clientSocket

TCP vs. UDP

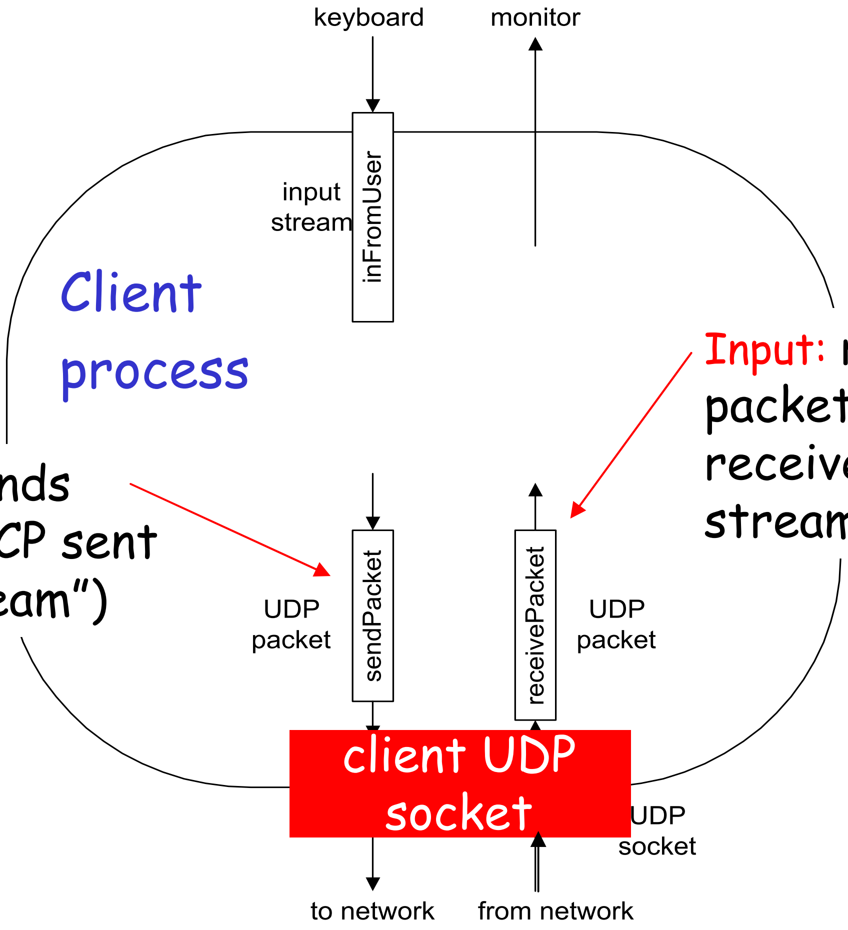
TCP

1. **Socket()**
   - Connection steam established: Data goes in one end of pipe and out the other. Pipe stays open until it is closed.
2. **ServerSocket()**
   - A special type of socket that sits waiting for a knock from a client to open connection. Leads to *handshaking*.

UDP

1. **DatagramSocket()**
   - Data sent as individual packets of bytes. Each packet contains all addressing info. No concept of open “pipe”.
2. No handshaking!
   - A DatagramSocket waits to receive each packet

Example: Java client (UDP)



Output: sends packet (TCP sent "byte stream")

Input: receives packet (TCP received "byte stream")

Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {
```

Create input stream

```
      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));
```

Create client socket

```
      DatagramSocket clientSocket = new DatagramSocket();
```

Translate hostname to IP address using DNS

```
      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```
    DatagramPacket sendPacket =
      new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```
    clientSocket.send(sendPacket);

    DatagramPacket receivePacket =
      new DatagramPacket(receiveData, receiveData.length);
```

Read datagram from server

```
    clientSocket.receive(receivePacket);

    String modifiedSentence =
       new String(receivePacket.getData());

    System.out.println("FROM SERVER:" + modifiedSentence);
    clientSocket.close();
    }
}
```

Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {
```

Create datagram socket at port 9876

```
      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {
```

Create space for received datagram

```
          DatagramPacket receivePacket =
             new DatagramPacket(receiveData, receiveData.length);
```

Receive datagram

```
           serverSocket.receive(receivePacket);
```

Example: Java server (UDP), cont

```
String sentence = new String(receivePacket.getData());
```

Get IP addr
port #, of
sender

```
InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

                String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();
```

Create datagram
to send to client

```
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress,
                        port);
```

Write out
datagram
to socket

```
serverSocket.send(sendPacket);
   }
  }
}
```

End of while loop,
loop back and wait for
another datagram

Chapter 2 outline

- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

Building a simple Web server

- handles one HTTP request
- accepts the request
- parses header
- obtains requested file from server's file system
- creates HTTP response message:
  - header lines + file
- sends response to client
- after creating server, you can request file using a browser (e.g. IE explorer)
- see text for details

Chapter 2 outline

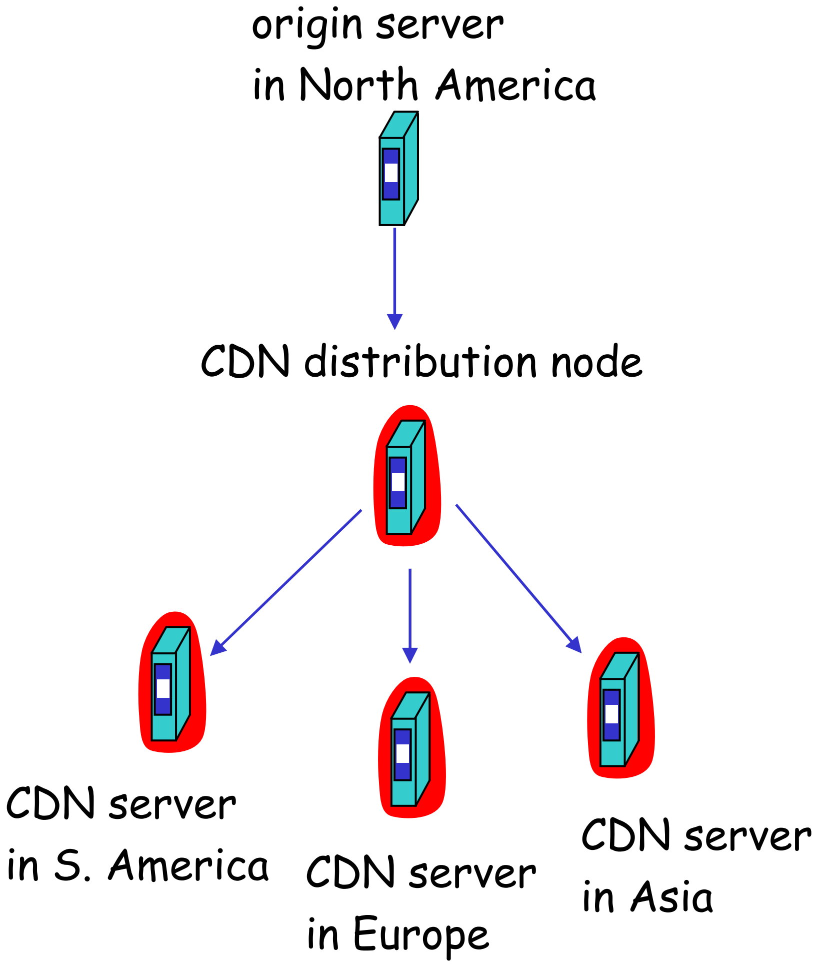
- 2.1 Principles of app layer protocols
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Content distribution networks vs. Web Caching

Content distribution networks (CDNs)

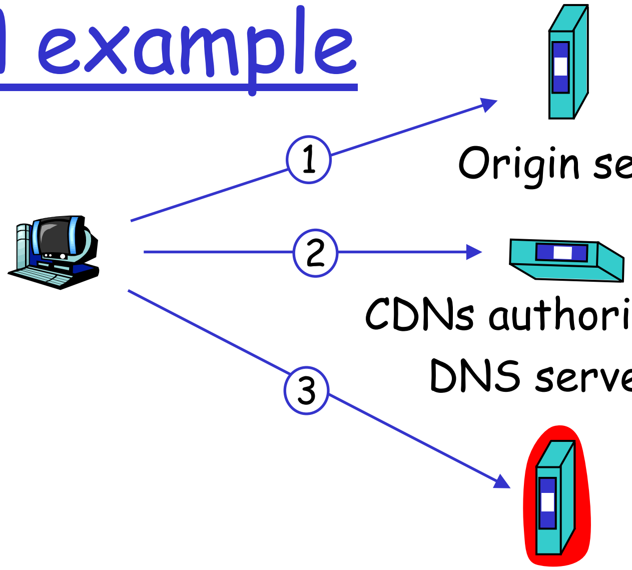
- The content providers are the CDN customers

Content replication

- CDN company installs hundreds of CDN servers throughout Internet
  - in lower-tier ISPs, close to users
- CDN replicates its customers' content in CDN servers. When provider updates content, CDN updates servers



CDN example



HTTP request for
www.foo.com/sports/sports.html

Origin server

DNS query for www.cdn.com

CDNs authoritative
DNS server

HTTP request for
www.cdn.com/www.foo.com/sports/ruth.gif

Nearby
CDN server

origin server

- www.foo.com
- distributes HTML
- Replaces:

  http://www.foo.com/sports.ruth.gif

  with

  http://www.cdn.com/www.foo.com/sports/ruth.gif

CDN company

- cdn.com
- distributes gif files
- uses its authoritative DNS server to route redirect requests

More about CDNs

routing requests

- CDN creates a "map", indicating distances from leaf ISPs and CDN nodes
- when query arrives at authoritative DNS server:
  - server determines ISP from which query originates
  - uses "map" to determine best CDN server

not just Web pages

- streaming stored audio/video
- streaming real-time audio/video

Web Caching vs. CDN

Both Web Caching and CDN replicate content

- Web Caching: Content replicated on demand as function of user requests
- CDN: Content replicated by content provider

P2P

As well as retrieving objects from content providers/proxy caches/CDNs it is also possible for edge-machines to retrieve content from other edge-machines. This approach is known as Peer-To-Peer (P2P).

For more on P2P see textbook.

Chapter 2: Summary

Our study of network apps now complete!

- application service requirements:
  - reliability, bandwidth, delay
- client-server paradigm
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
- socket programming
- content distribution
  - caches, CDNs
  - P2P

Chapter 2: Summary

Most importantly: learned about *protocols*

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated
- control vs. data msgs
  - in-based, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"
- security: authentication