Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chapter 2
APPLICATIONS

Computer Networks

Timothy Roscoe

Summer 2007

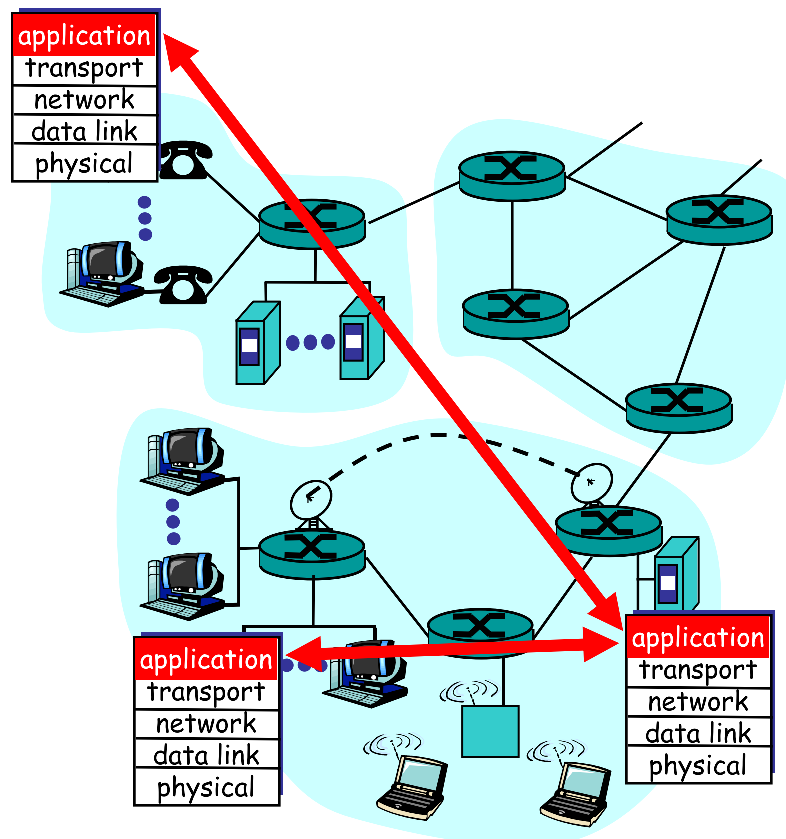
Informatik
Computer Science

Overview

- This week: Learn specific application layer protocols
  - HTTP, FTP, SMTP, POP, DNS, etc.
  - learn about protocols by examining popular application-level protocols
  - conceptual and implementation aspects of network application protocols
  - client-server paradigm
  - service models
- Next week: How to program network applications?
  - Socket API for Java and C

Applications vs. Application-Layer Protocols

- Application: communicating, distributed process
  - running in network hosts in "user space"
  - exchange messages to implement application
  - e.g. email, ftp, web
- Application-layer protocol
  - one part of application
  - define messages exchanged by applications and actions taken
  - use communication services provided by transport layer protocols (TCP, UDP)

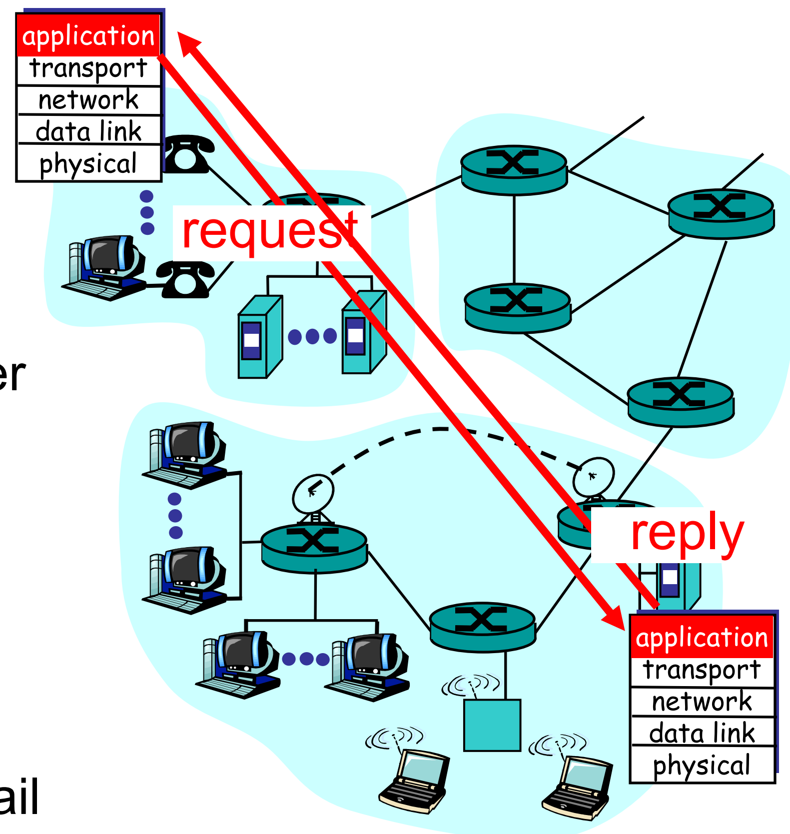


Network applications: some jargon

- Process: program running within a host
  - within same host, two processes *can* communicate using interprocess communication (defined by the Operating System).
  - processes running on different hosts must communicate with an application-layer protocol through messages
- User agent: software process, interfacing with user "above" and network "below"
  - implements application-level protocol
  - Examples
    - Web: browser
    - E-mail: mail reader
    - streaming audio/video: media player

Client-server paradigm

Typical network app has two parts: Client and Server



Client

- initiates contact with server ("client speaks first")
- typically requests service from server
- Web: client implemented in browser
- email: client in mail reader

Server

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

API: Application Programming Interface

- Defines interface between application and transport layers
- Most common Internet API: "*sockets*"
- two processes communicate by sending data into socket, reading data out of socket

- How does a process identify the other process with which it wants to communicate?
  - IP ("Internet Protocol") 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) require 100% reliable data transfer

Timing

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

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

Common transport requirements

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	loss-tolerant	elastic	no
real-time audio/video, e.g VoIP	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 150 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

Internet transport protocol services

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 provide timing, minimum 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
- Why bother? Why is there a UDP service at all?!?

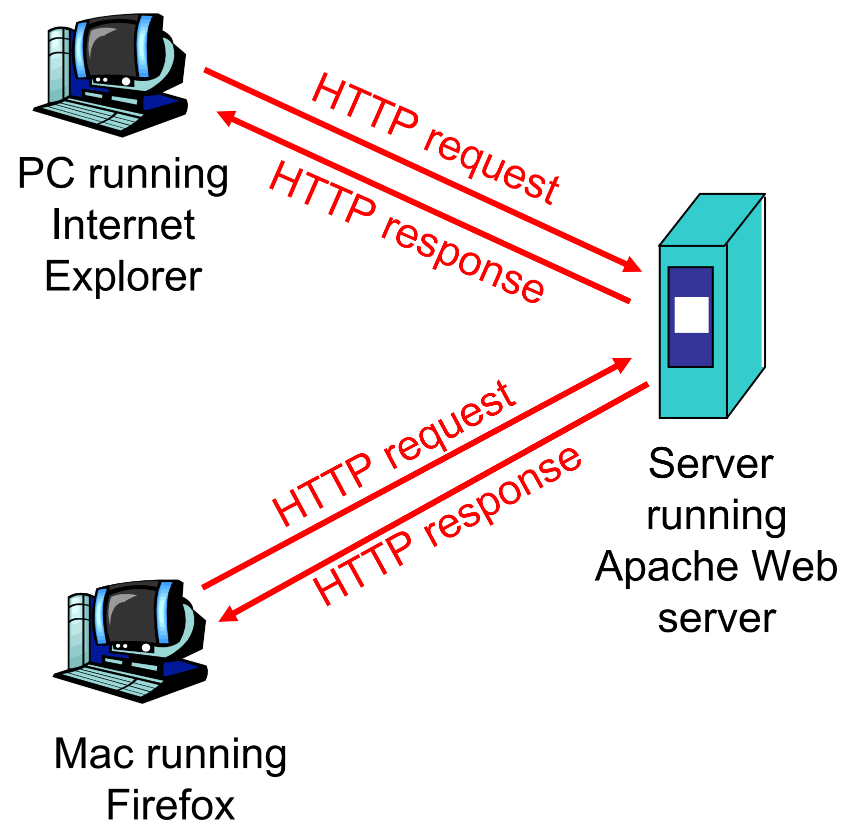
Internet apps: application, transport protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 821]	TCP
remote terminal access	telnet [RFC 854]	TCP
World-wide web	HTTP [RFC 2068]	TCP
file transfer	ftp [RFC 959]	TCP
streaming multimedia	RTP, RTSP, etc.	TCP or UDP
remote file server	NFS, SMB	TCP or UDP
Internet telephony	SIP, Skype, etc.	typically UDP

The Web: The HTTP protocol

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, and "displays" Web objects
  - *server:* Web server sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2616



More on the HTTP protocol

- 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

Example for HTTP

Suppose user enters URL **`http://www.inf.ethz.ch/education/index.html`**
(assume that web page contains text, references to 10 jpeg images)

1. HTTP client initiates TCP connection to HTTP server (process) at **`www.inf.ethz.ch`**. Port 80 is default for HTTP server.

2. HTTP server at host **`www.inf.ethz.ch`** waiting for TCP connection at port 80, "accepts" connection, notifies client

3. HTTP client sends HTTP request message (containing URL) into TCP connection socket

4. HTTP server receives request message, forms *response message* containing requested object (**`index.html`** in directory **`education`**), sends message into socket

time

Example for HTTP (continued)

5. HTTP server closes TCP connection

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

Then…

Steps 1-6 repeated for each of the 10 jpeg objects

time

Non-persistent vs. persistent connections

Non-persistent

- HTTP/1.0
- server parses request, responds, closes TCP connection
- 2 RTTs (round-trip-time) to fetch object
  – TCP connection
  – object request/transfer
- each transfer suffers from TCP's initially slow sending rate
- many browsers open multiple parallel 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, less slow start

HTTP message format: request

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

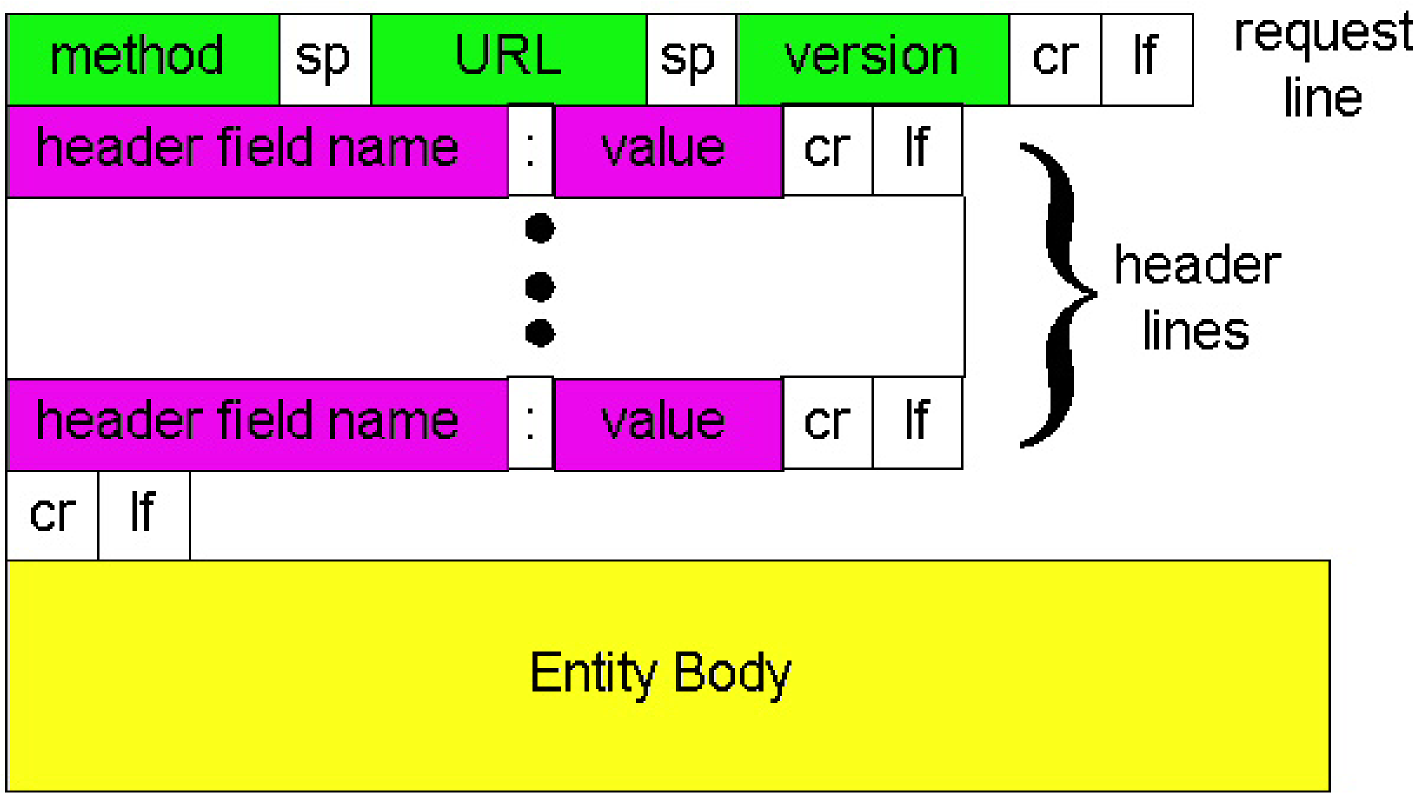
request line
(GET, POST,
HEAD commands)

```
GET /somedir/page.html HTTP/1.1
Host: www.servername.com
User-agent: Mozilla/4.0
Accept-language: de

(extra carriage return, line feed)
```

header
lines

Carriage return
and line feed
indicate end
of message

HTTP request message: the general format



HTTP message format: response

status line
(protocol
status code
status phrase)

HTTP/1.1 200 OK

header
lines

Date: Tue, 27 Mar 2007 12:03:35 GMT
Server: Apache/1.3.33 (Unix)
Connection: close
Content-Type: text/html; charset=iso-8859-1
Last-Modified: Mon, 22 Jun 1998 …
Content-Length: 6821

data, e.g.
requested
html file

<HTML><HEAD> ...

HTTP response status codes

First line of 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

An aside on Telnet

- Remote (character) terminal access [RFC 854, 1983!]
  - Uses TCP transport, port 23
  - Lots of in-band control codes
  - Surprisingly complex (15 pages + 40 further RFCs!)
  - No security (encryption, etc.) until 2000.
  - Largely superceded by Secure Shell (**`ssh`**)
  - Hardly used any more…

But…

- Standard in Unix: **`telnet <host> [<port>]`**
- Most Internet protocols are intentionally *text based*
  - Ease of implementation, debugging, testing
  - telnet is fantastically useful for protocol hacking…

Ultra-minimalist web browsing

1. Telnet to a Web server:
   telnet people.inf.ethz.ch 80

2. Type in a GET HTTP request:
   GET /troscoe/ HTTP/1.0

3. Check out response message sent by HTTP server…

- Opens TCP connection to port 80 (default HTTP server port) at people.inf.ethz.ch.
- Anything typed in sent to people.inf.ethz.ch port 80
- By typing this followed by a blank line (hit return twice), you send this minimal (but complete) GET request to the HTTP server

But why doesn't this work for something useful like www.sbb.ch?

More modern ultra-minimalist web browsing

- Lots of web sites on the same machine
- Only one port 80
- Need to say which site you want

1. telnet www.sbb.ch 80

2. Type in a GET HTTP request:

   GET /index.html HTTP/1.0
   Host: www.sbb.ch

3. Should work a lot better…

Sneak peek:

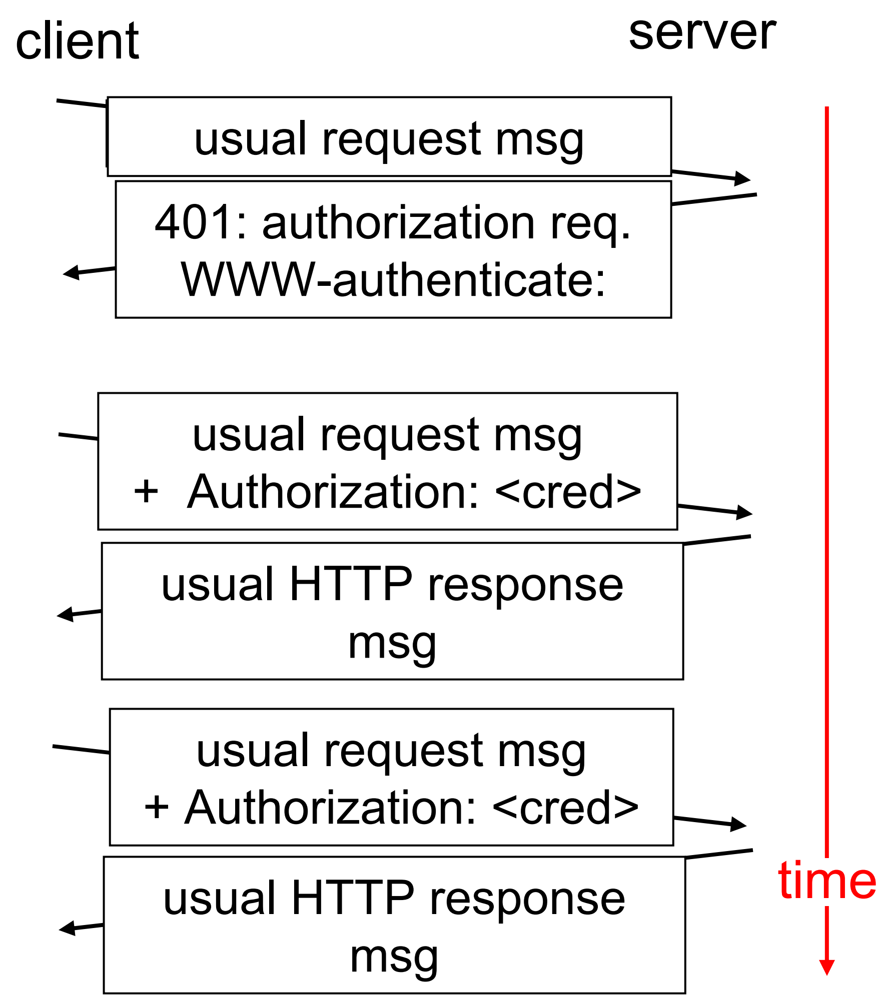
- "www.sbb.ch" is the *name* of the site, but not its *address*
- One *address* can have many *names*
- More on this later with DNS…

HTTP user-server interaction: authentication

- Authentication: control access to server content
- authorization credentials: typically name and password
- stateless: client must present authorization in *each* request
  - authorization: header line in each request
  - if no authorization: header, server refuses access, sends

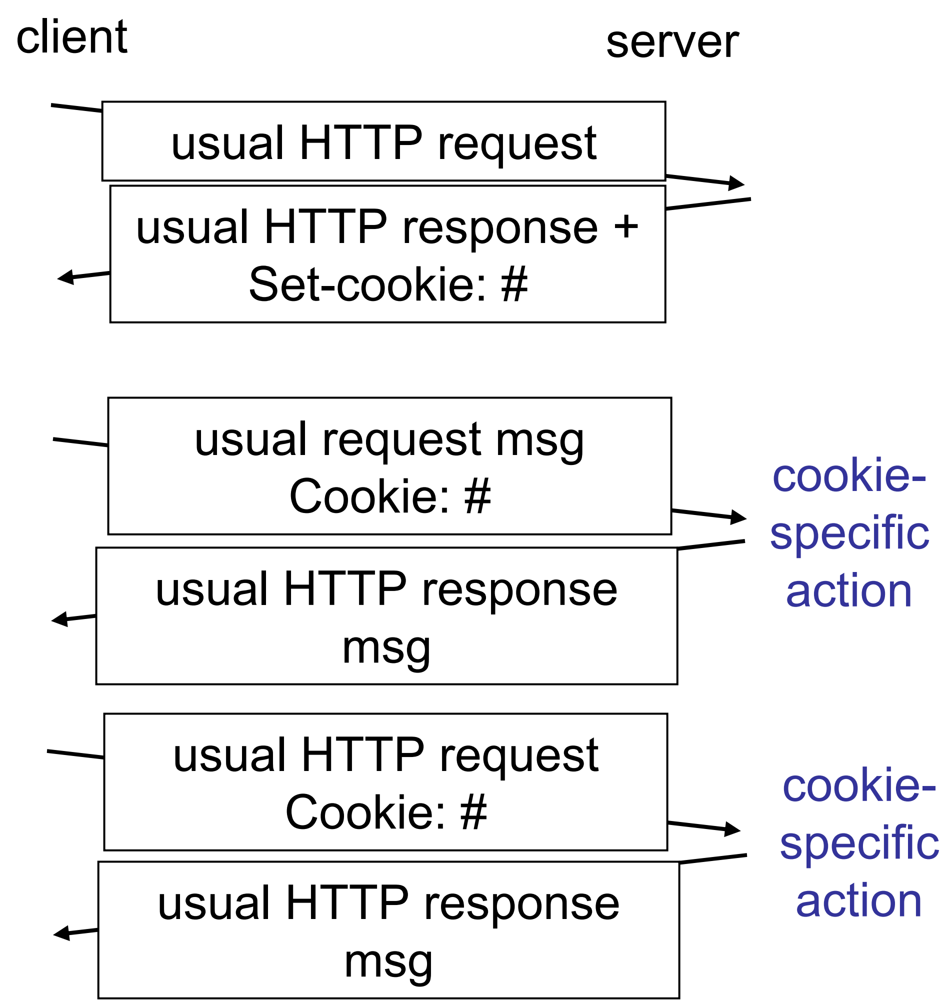
    `WWW authenticate:`

    header line in response



Cookies: keeping "state"

- server-generated # , server-remembered #, later used for
  - authentication
  - remembering user preferences
  - remembering previous choices
  - (…privacy?)
- server sends "cookie" to client in response msg
  `Set-cookie: 1678453`
- client presents cookie in later requests
  `Cookie: 1678453`



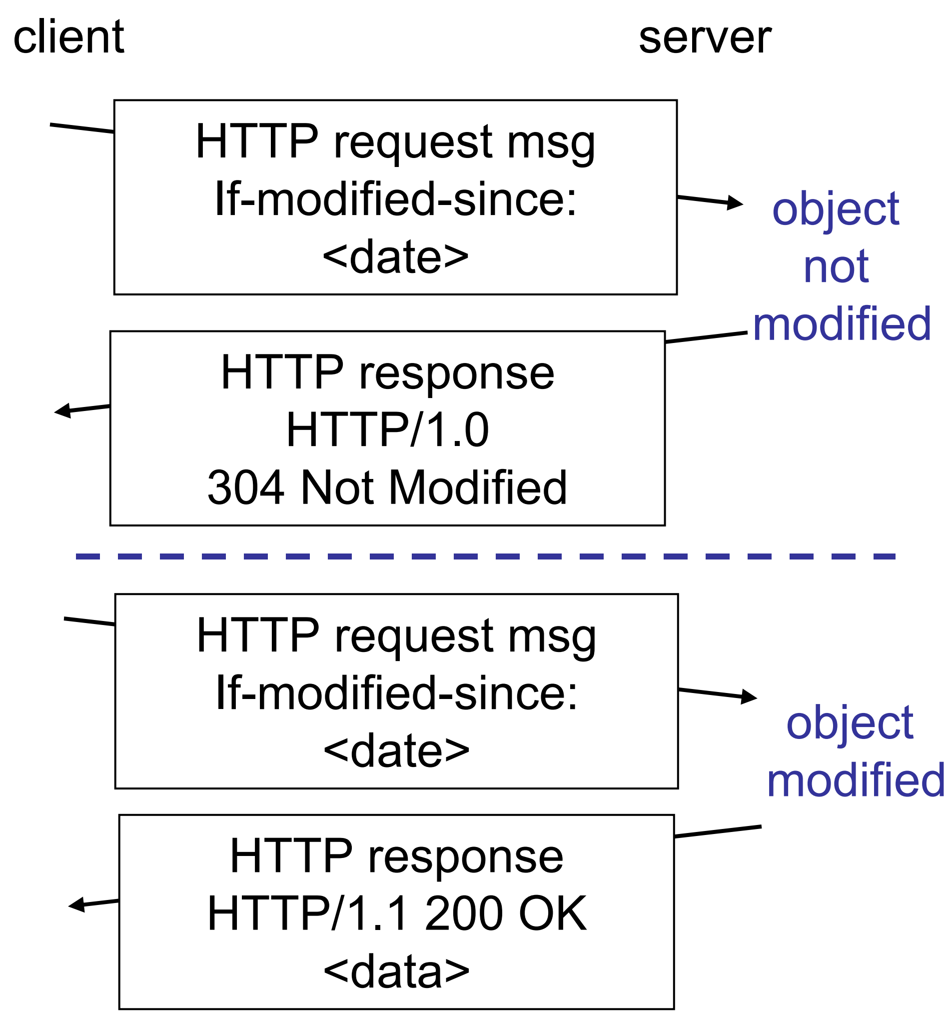
A recent cookie from Google

```
Set-Cookie: PREF=ID=313e7de24f3b48a3:
    TM=1175005089:LM=1175005089:S=OoXbqHqVOejOEVmc;
    expires=Sun, 17-Jan-2038 19:14:07 GMT;
    path=/;
    domain=.google.com
```

- Expires: when to throw this cookie away
- Domain: who to present this cookie to
- Path: which URLs to present this cookie with
- The rest: known only to Google (but…)

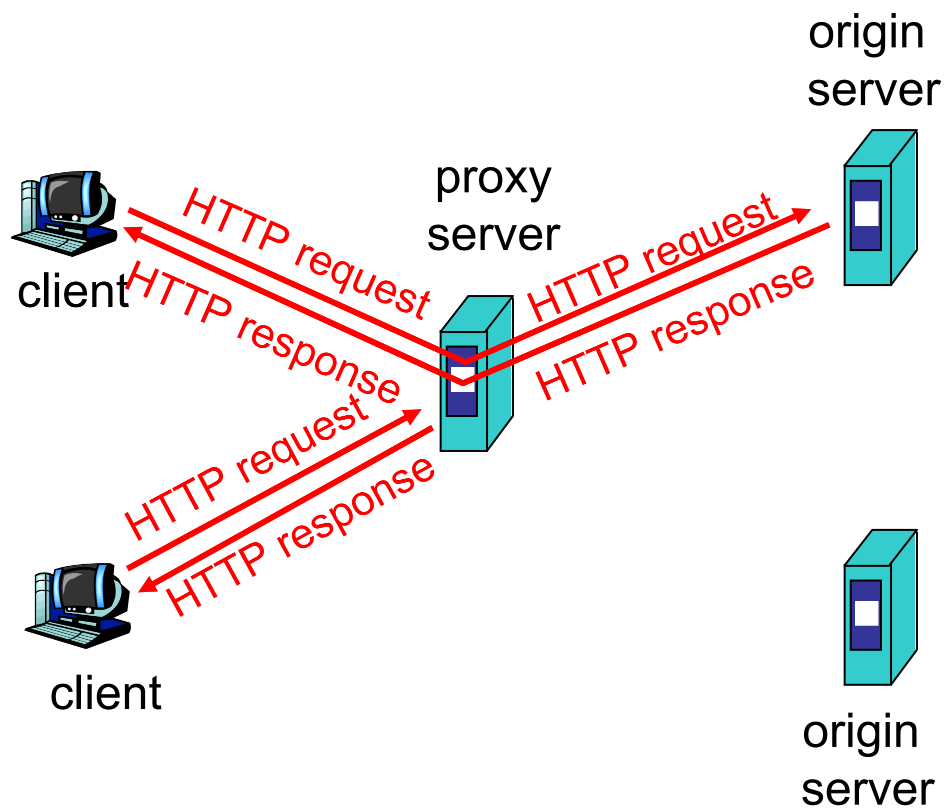
Conditional GET: client-side caching

- Goal: don't send object if client has up-to-date cached version
- Client: specify date of cached copy in HTTP request
  If-modified-since: <date>
- Server: response contains no object if cached copy is up-to-date:
  HTTP/1.0 304 Not Modified



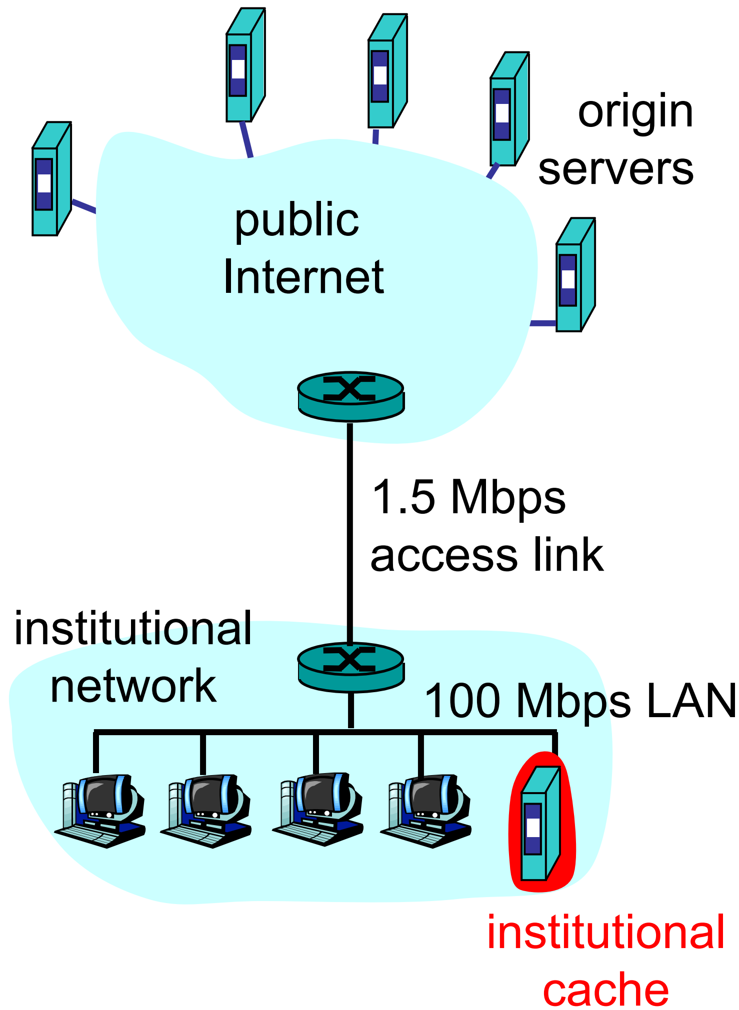
Web Caches (a.k.a. proxy server)

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



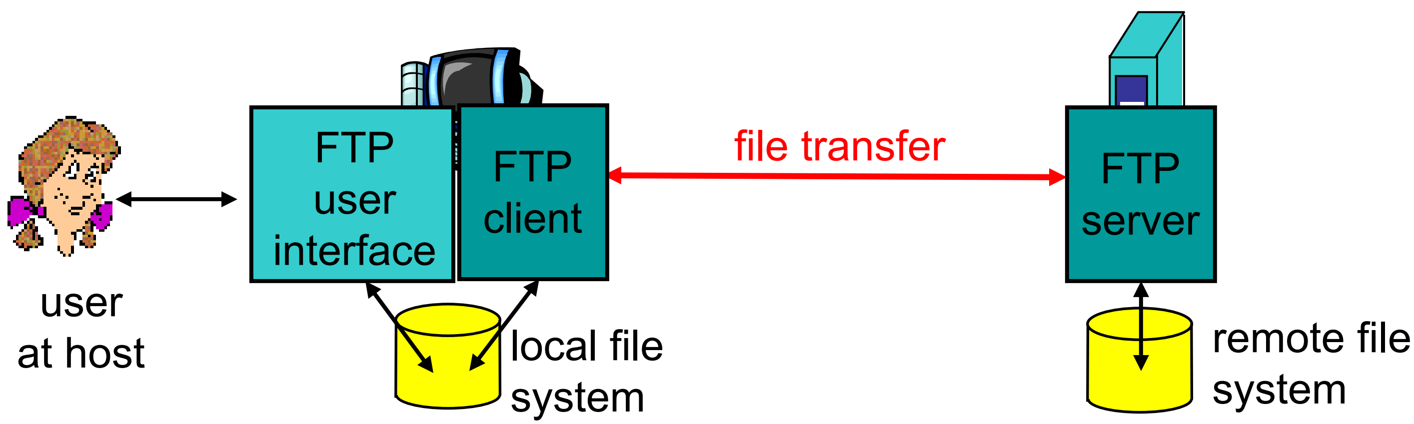
Why Web Caching?

- Assumption: cache is “close” to client (e.g. in same network)
- Smaller response time: cache “closer” to client
- Decrease traffic to distant servers
- Link out of institutional/local ISP network is often a bottleneck



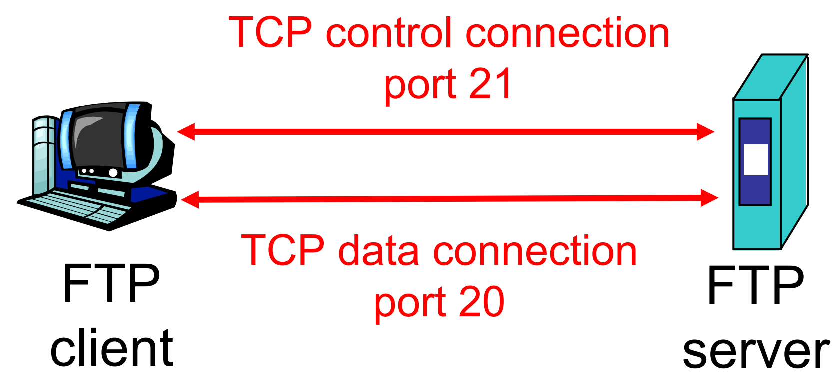
ftp: The file transfer protocol

- 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 and 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 commands and responses

Sample commands

- sent as ASCII text over control channel
- **`USER`** ***`username`***
- **`PASS`** ***`password`***
- **`LIST`** returns list of files 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`**

Good taste in protocol implementation

- Jon Postel in RFC 791:

  "In general, an implementation should be conservative in its sending behaviour and liberal in its receiving behaviour"

- The hardest thing about protocol implementation is "expecting the unexpected".
- People send you the strangest stuff…

- Worst-case example: *electronic mail*

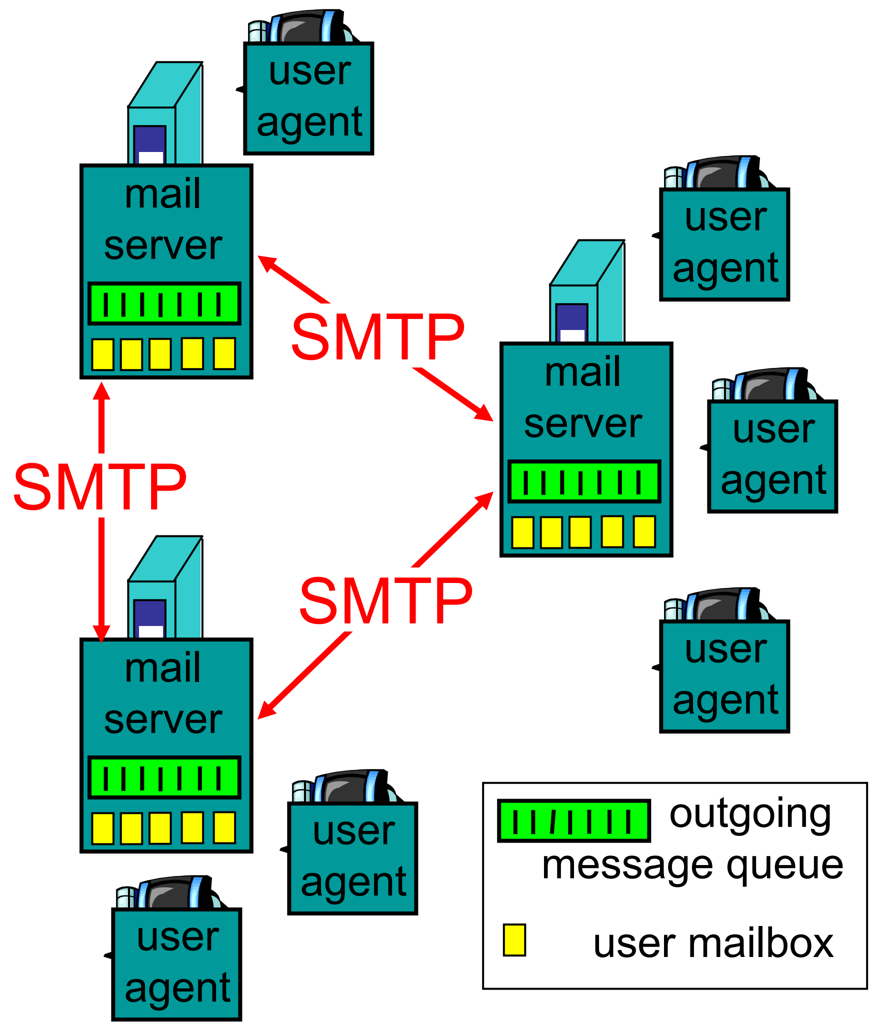
Electronic Mail

Three major components

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

User Agent

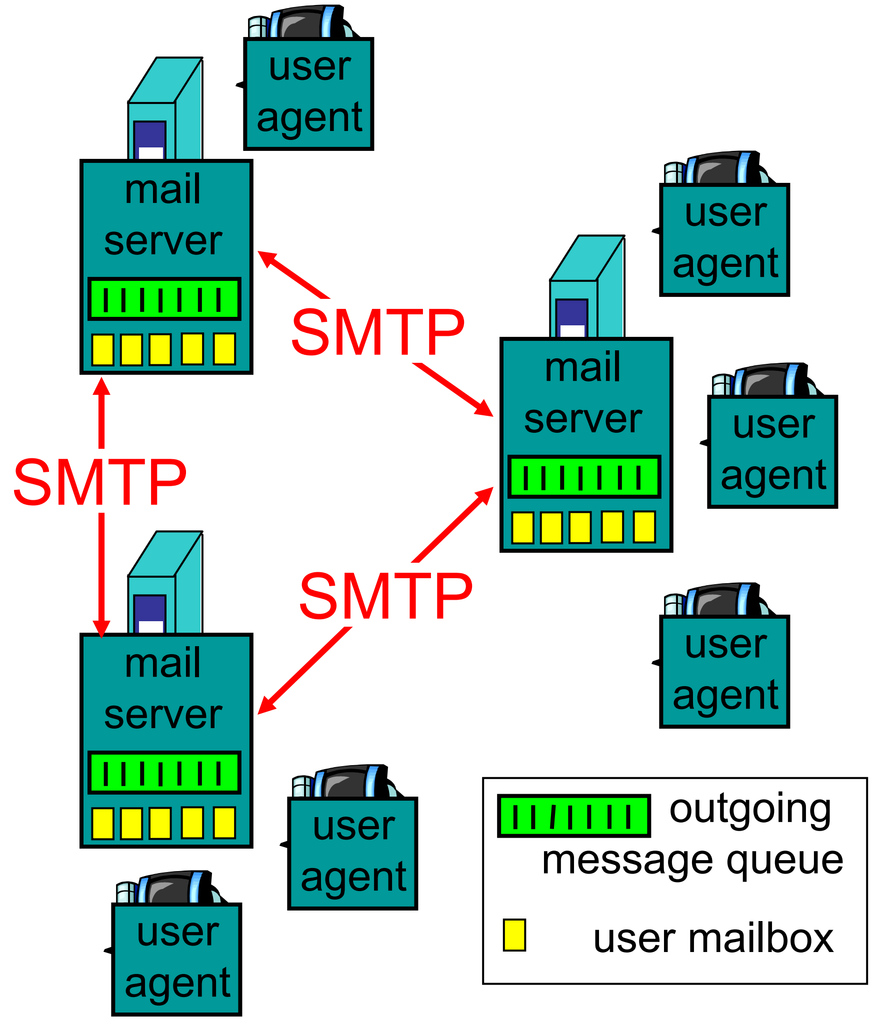
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- Examples: Outlook, Netscape Messenger, elm, Eudora
- outgoing, incoming messages stored on server



Electronic Mail: 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

- Why not sending directly?



Electronic Mail: SMTP

- uses TCP to reliably transfer email message from client to server, on port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshake (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - commands: ASCII text
  - response: status code and phrase
- SMTP: RFC 821

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: From: Alice <alice@crepes.fr>
C: To: Bob <bob@hamburger.edu>
C: Subject: Fancy lunch?
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
```

SMTP "issues"

- Trademark of Hormel Foods, Inc.
- Pork, mechanically recovered chicken, additives
- Inexplicably, a delicacy in Hawaii…
- Immortalized by Monty Python
  - Spam, spam, spam, spam, …
  - Unwanted, typically anonymous / forged email

SMTP: more details

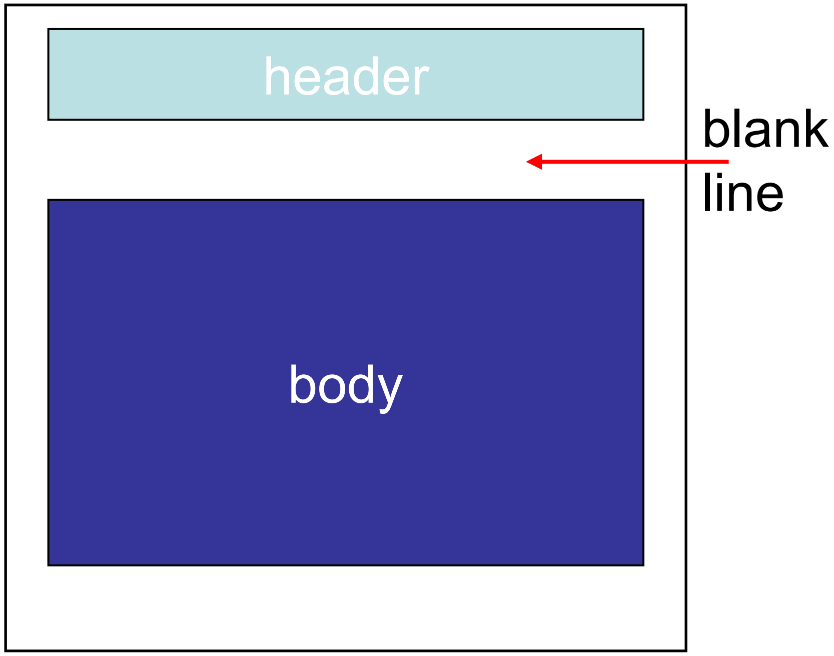
- Persistent connections
- Requires message (header & body) to be in 7-bit ASCII
- certain character strings not permitted in msg (e.g., **`CRLF.CRLF`**, which is used to determine the end of a message by the server).
- ⇒ msg must be encoded (usually base-64 or quoted-printable)

Comparison with HTTP

- HTTP: pull, email: push
- both have ASCII command/response interaction and status codes
- HTTP: each object encapsulated in its own response msg (1.0), or by use of content-length field (1.1)
- SMTP: multiple objects sent in multipart msg (as we will see on the next slides)

Mail message format

- SMTP: protocol for exchanging email msgs
- RFC (2)822: standard for text message format:
- header lines, e.g.
  - To:
  - From:
  - Subject:
  - (!) Caution: these are not SMTP commands! They are like the header of a letter, whereas SMTP commands are like the address on the envelope
- body
  - the “message”
  - ASCII characters only



Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2046, …
- additional lines in message 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

/9j/4RRARXhpZgAASUkqAAgAAAAKAA8BA
gAOAAAAhgAAABABAgAGAAAApgAAABIBAw
ABAAAAAQAAABoBBQABAAAA3gAAABsBBQA
...
```

MIME types

`Content-Type: type/subtype; parameters`

Text

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

Image

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

Audio

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

Video

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

Application

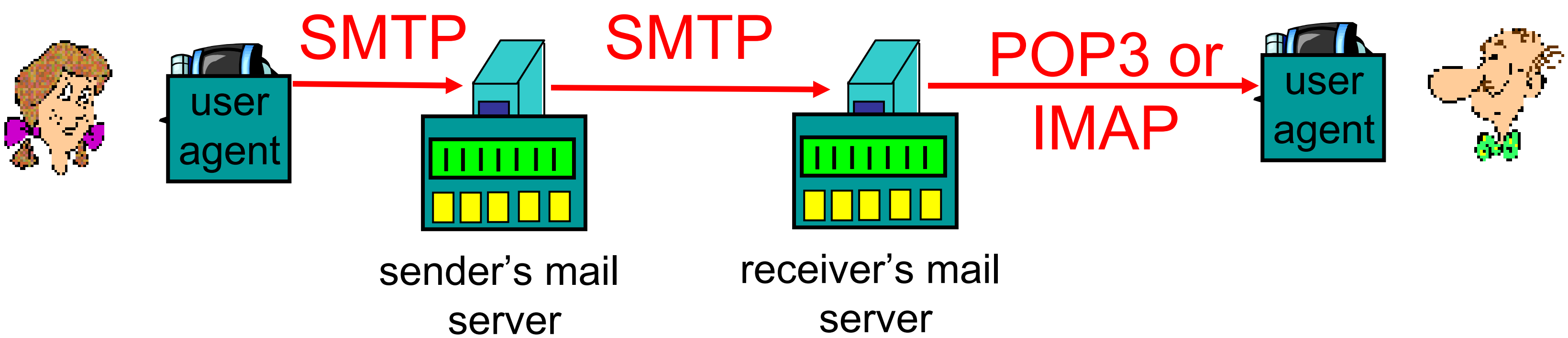
- other data that must be processed by reader before “viewable”
- example subtypes: **`msword, octet-stream`**

MIME Multipart Type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=98766789

--98766789
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain

Dear Bob,
Please find a picture of a crepe.
--98766789
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
/9j/4RRARXhpZgAASUkqAAgAAAAKAA8BAgAOAAAAhgAAABABAgAGAAAApg
AAABIBAwABAAAAAQAAABoBBQABAAAA3gAAABsBBQA ...

--98766789--
```

Mail access protocols

- SMTP: delivery/storage to receiver's server
  - In the old days, their own machine...
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 2060]
    - more features (more complex)
    - manipulation of stored messages on server
  - HTTP: Hotmail, Yahoo! Mail, Google Mail, etc.



POP3 protocol

Authorization phase

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

Transaction phase

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

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

```
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 2 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

DNS: Domain Name System

People have many identifiers

- passport number, AHV number, student number, name, etc.

Internet hosts, routers

- IP address (129.132.130.152); used for addressing datagrams
- Name (`photek.ethz.ch`); used by humans

- We need a map from names to IP addresses (and vice versa?)

Domain Name System

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol* host, routers, name servers to communicate to *resolve* names (name/address translation)
  - note: is a core Internet function, but only 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

…it does not *scale!*

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

local name servers

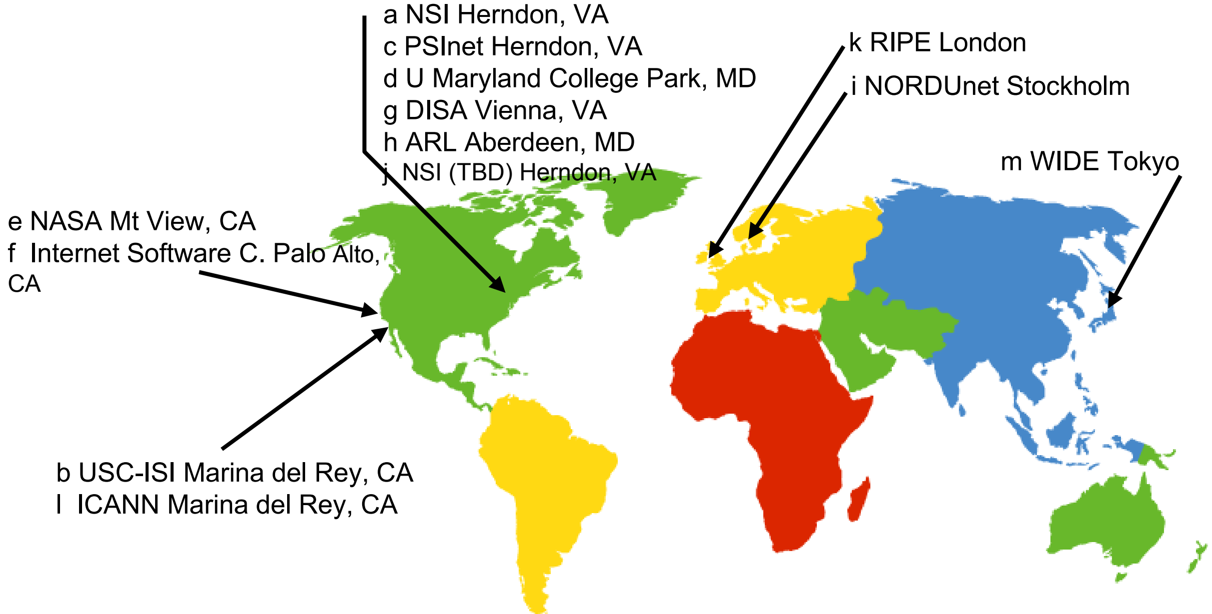
- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

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

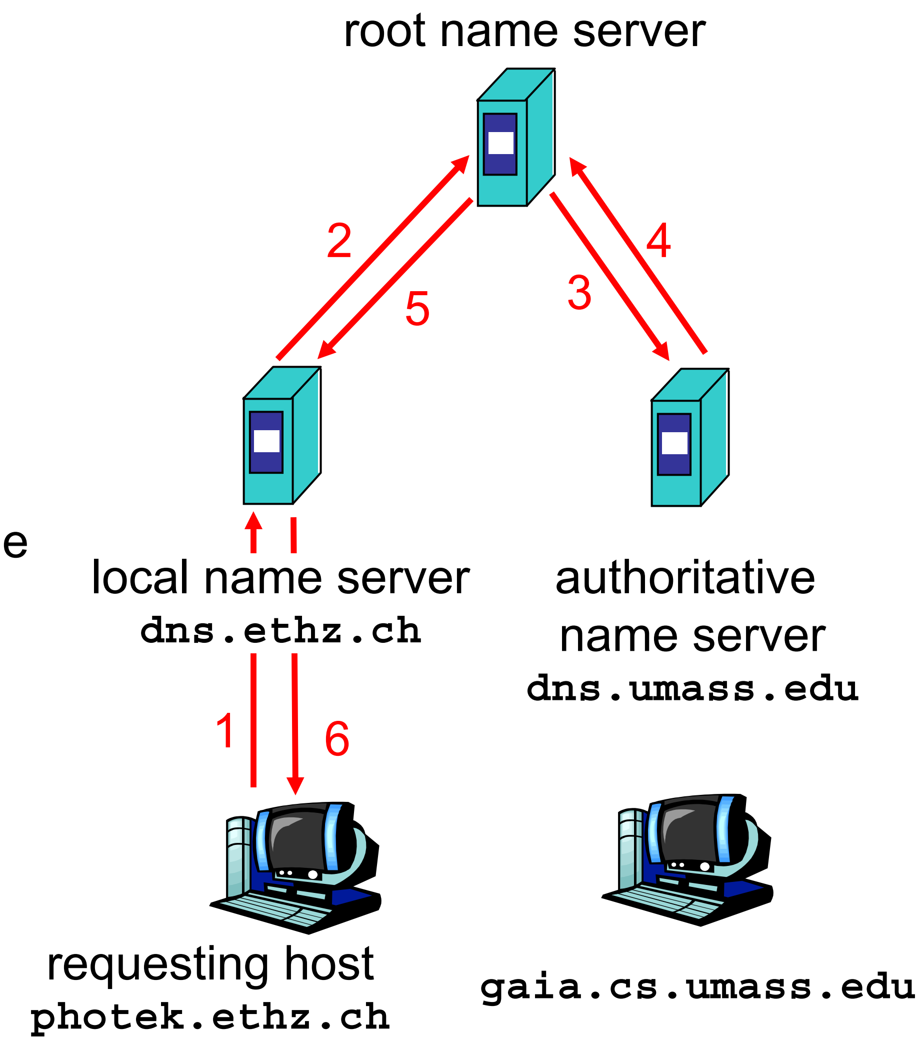
- contacted by local name server that cannot resolve name
- root name server
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server
  - Until recently, 13 root name servers worldwide



Simple DNS example

- host **photek.ethz.ch** wants IP address of **gaia.cs.umass.edu**

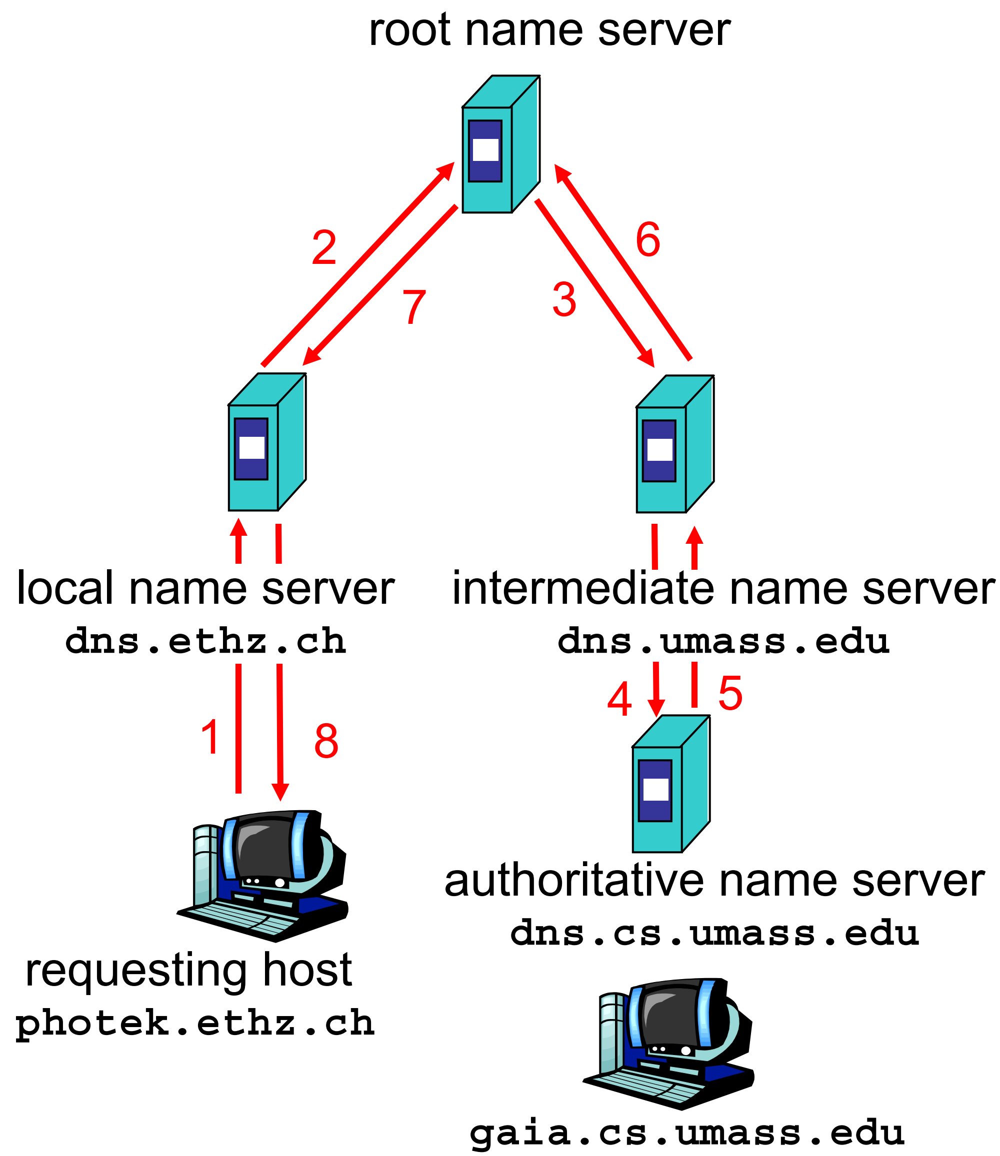
1. contact local DNS server, **dns.ethz.ch** (the "primary resolver")
2. **dns.ethz.ch** contacts root name server, if necessary
3. root name server contacts authoritative name server, **dns.umass.edu**, if necessary



DNS extended example

Root name server:

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



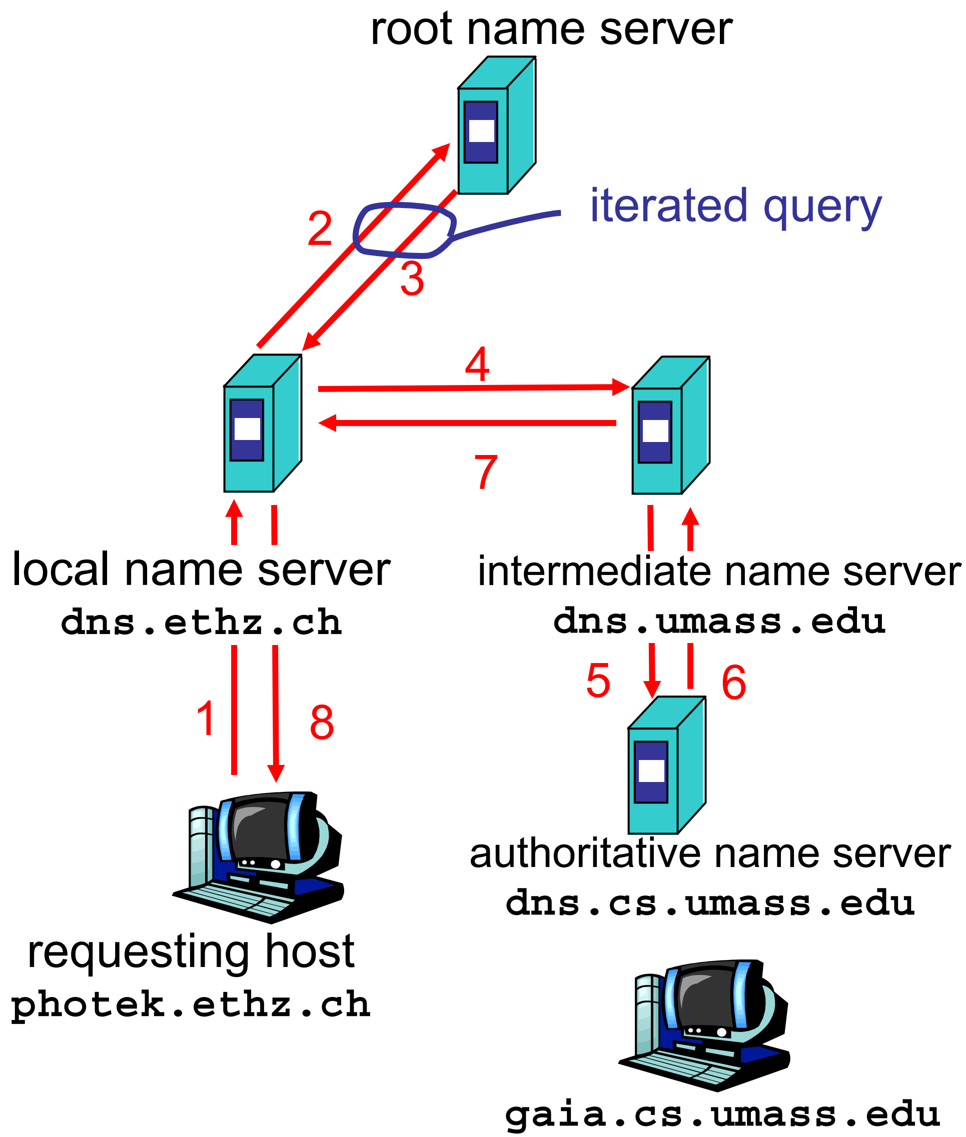
DNS Iterated queries

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 resource records

DNS: distributed database storing resource records (RR)

RR format: **(name, ttl, class, type, value)**

- 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 alias name for some "canonical" (the real) name
    `www.ibm.com` is really
    `servereast.backup2.ibm.com`
  - **`value`** is canonical name
- Type=MX
  - **`value`** is name of mail server associated with **`name`**

Example of DNS lookup

```
$ dig www.sbb.ch

; <<>> DiG 9.3.2-P1 <<>> www.sbb.ch
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 18725
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2,
   ADDITIONAL: 0

;; QUESTION SECTION:
;www.sbb.ch.                    IN      A

;; ANSWER SECTION:
www.sbb.ch.             30      IN      A       194.150.245.35

;; AUTHORITY SECTION:
sbb.ch.                 11      IN      NS      ns2.sbb.ch.
sbb.ch.                 11      IN      NS      ns1.sbb.ch.

;; Query time: 3 msec
;; SERVER: 129.132.98.12#53(129.132.98.12)
;; WHEN: Tue Mar 27 17:25:24 2007
;; MSG SIZE  rcvd: 80
```

More complex example of DNS lookup

```
$ dig www.inf.ethz.ch
; <<>> DiG 9.3.2-P1 <<>> www.inf.ethz.ch
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12816
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 3,
   ADDITIONAL: 5

;; QUESTION SECTION:
;www.inf.ethz.ch.                IN      A

;; ANSWER SECTION:
www.inf.ethz.ch.         86400   IN      CNAME   www-css.ethz.ch.
www-css.ethz.ch.         86400   IN      A       129.132.46.11

;; AUTHORITY SECTION:
ethz.ch.                 86400   IN      NS      scsnms.switch.ch.
ethz.ch.                 86400   IN      NS      dns1.ethz.ch.
ethz.ch.                 86400   IN      NS      dns3.ethz.ch.

;; ADDITIONAL SECTION:
dns1.ethz.ch.            86400   IN      A       129.132.98.12
dns3.ethz.ch.            86400   IN      A       129.132.250.2
scsnms.switch.ch.        106745  IN      A       130.59.1.30
scsnms.switch.ch.        106745  IN      A       130.59.10.30
scsnms.switch.ch.        141765  IN      AAAA    2001:620::1
```

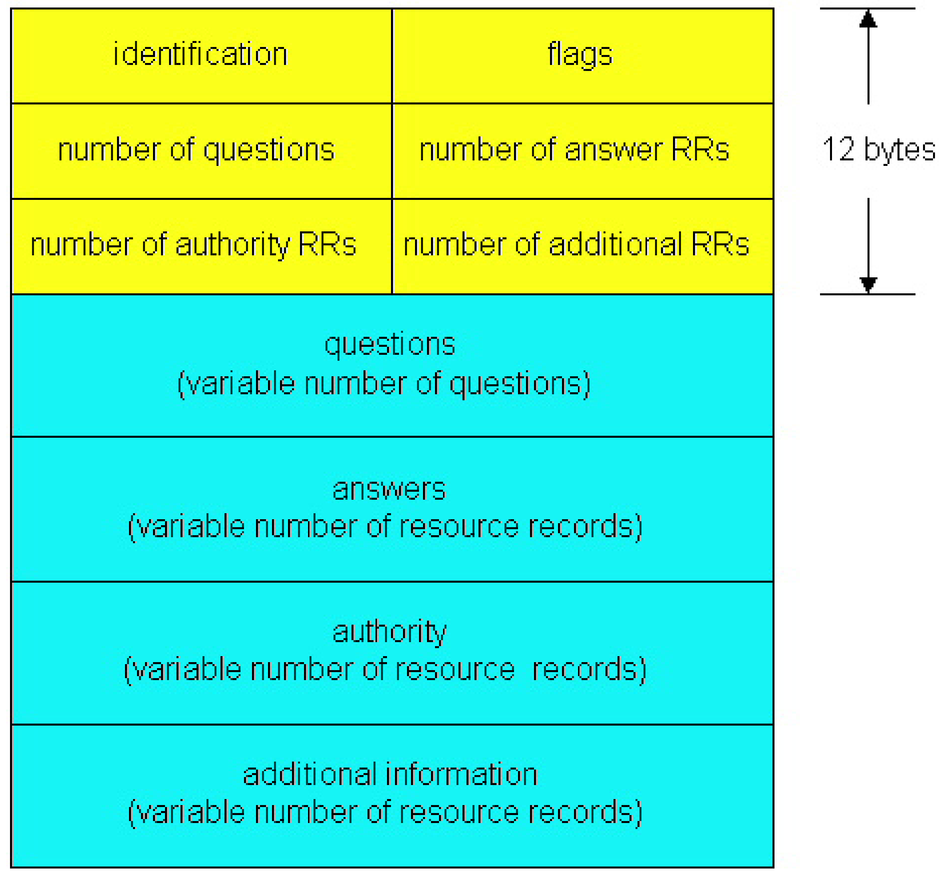
DNS protocol, messages

DNS protocol

- *query* and *reply* messages, both with same *message format*

msg header

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



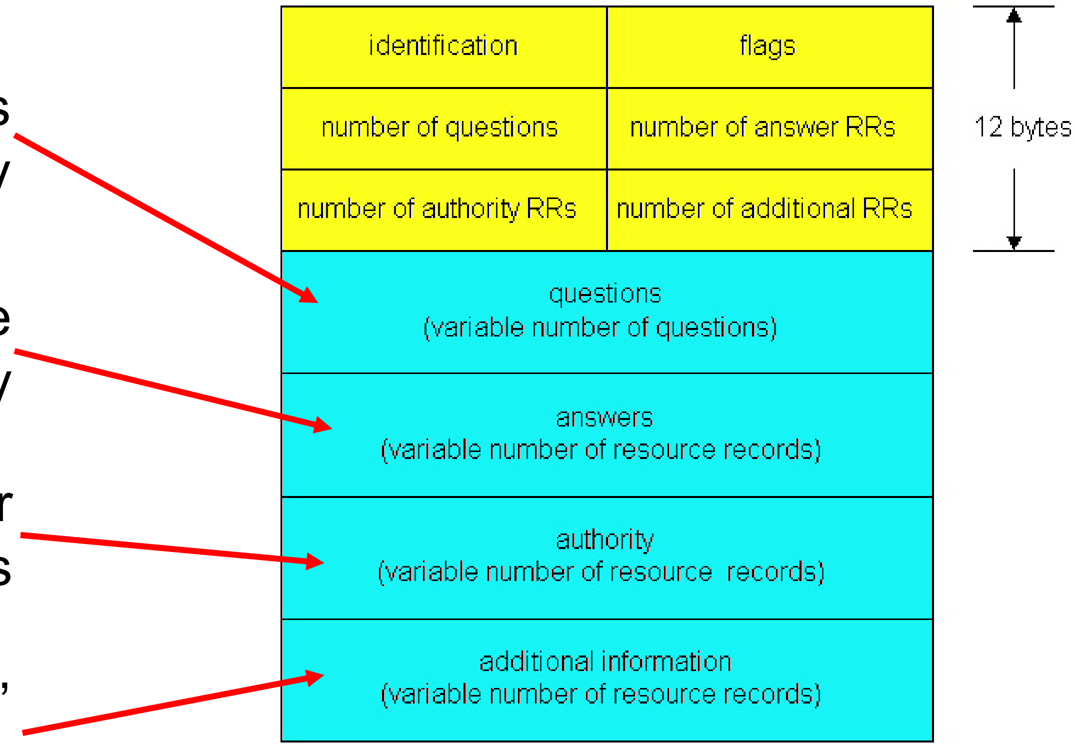
DNS protocol, messages

Name, type fields
for a query

RRs in response
to query

records for
authoritative servers

additional "helpful"
info that may be used



Note: unlike others we have seen, DNS is a *binary* protocol!

Other Internet application protocols

- … are numerous…
- File systems: NFS, SMB, AFS, etc.
- Encrypted sessions: SSH, SSL, TLS
- Filesharing: BitTorrent, Kazaa, …
- Netnews: NNTP
- Network Management: SNMP
- Games: DOOM (port 666, naturally)
- Historical artifacts: ECHO, DISCARD, CHARGEN, QUOTE, DAYTIME, TIME, FINGER

- Next: programming application protocols using sockets.