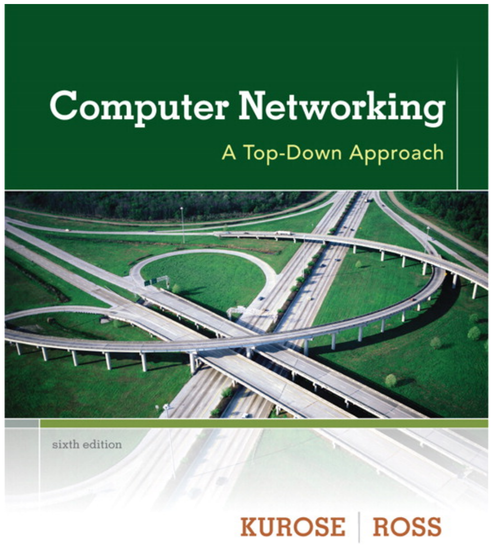
Chapter 2
Application Layer



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

Chapter 2: outline

Chapter 2: application layer

our goals:

- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- creating network applications
  - socket API

Some network apps

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

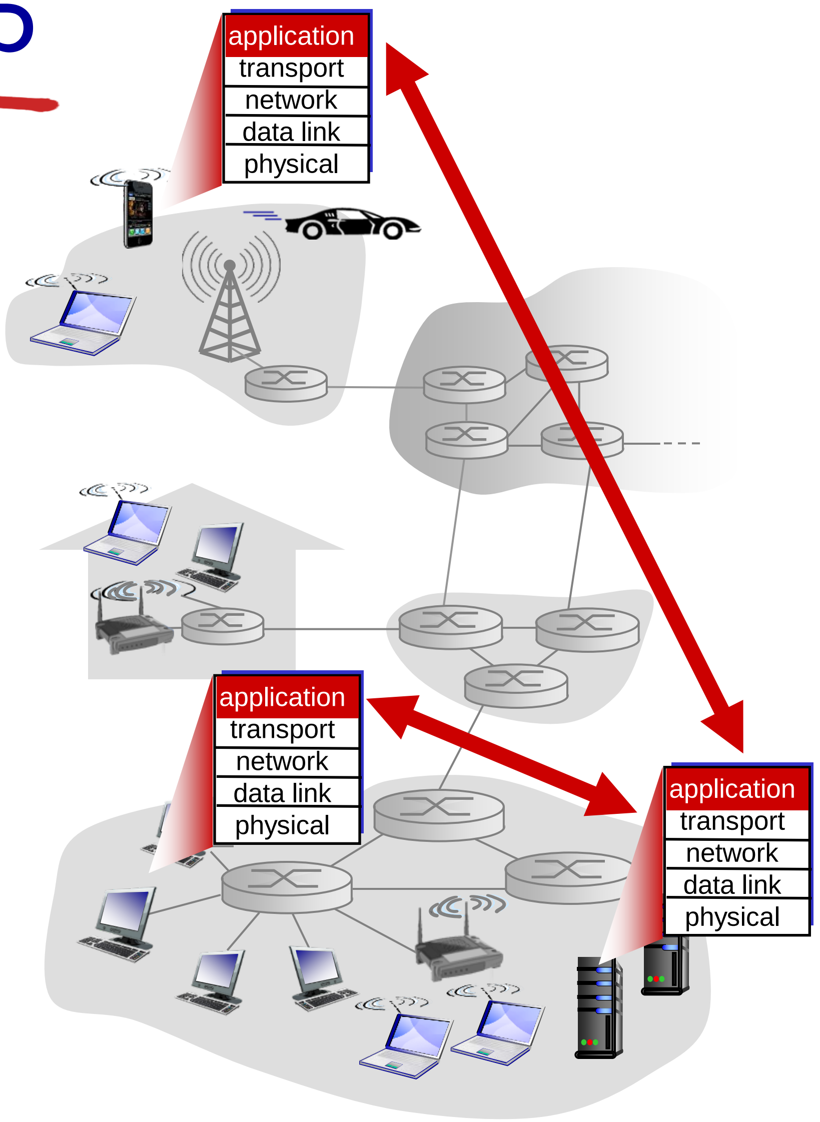
Creating a network app

write programs that:

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

no need to write software for network-core devices

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

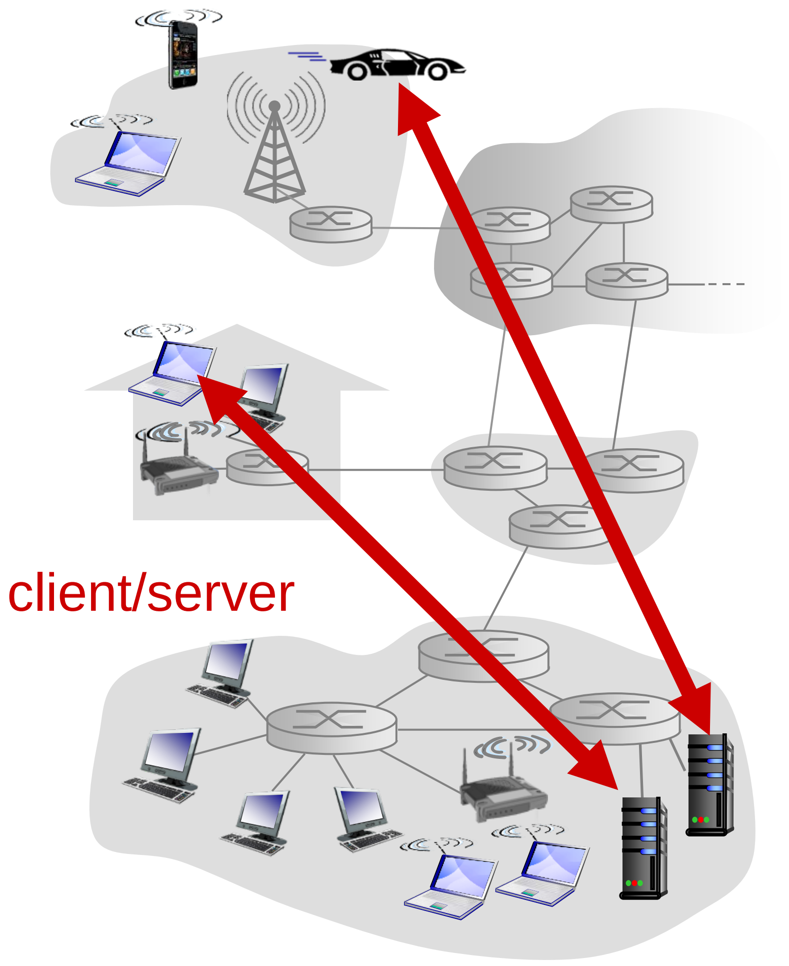


Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)
- Hybrid of client-server and P2P

Client-server architecture



server:

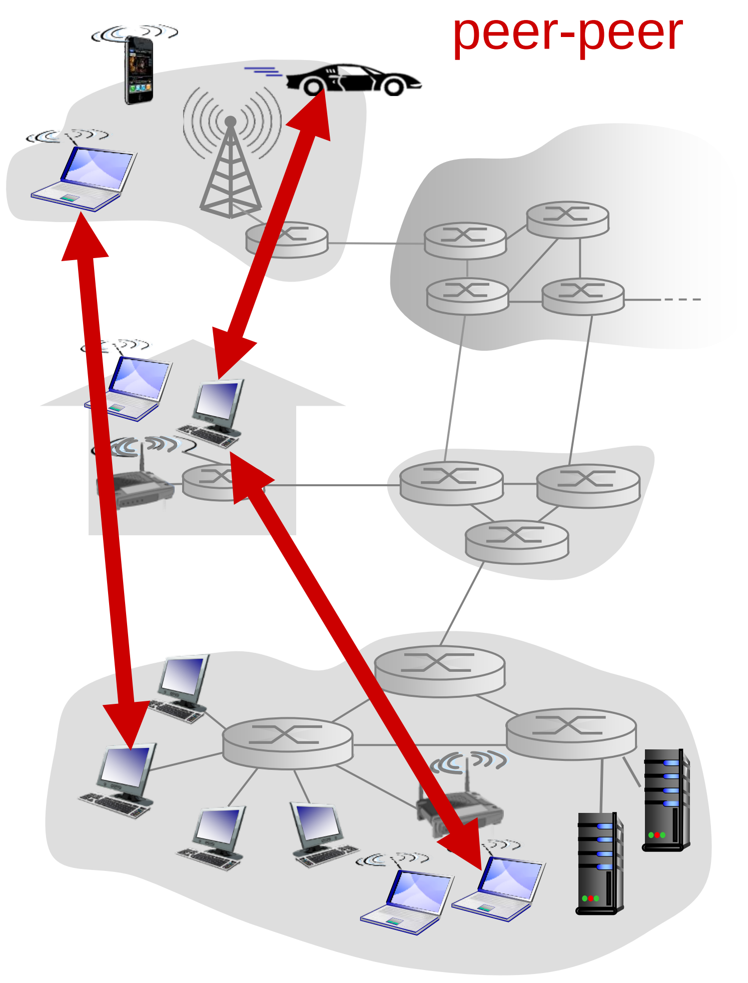
- always-on host
- permanent IP address
- data centers for scaling

clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

P2P architecture

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



Hybrid of client-server and P2P

- Skype
  - P2P VoIP application
  - Centralized server: finding address of remote receiver
  - Client-client connection: direct (not through server)
- Instant messaging
  - Chat between two users is P2P
  - Centralized: detection / location of the presence of the client
  - The user registers its IP address with central server to connect
  - The user contacts central server to find IP addresses of receiver

Processes communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

clients, servers

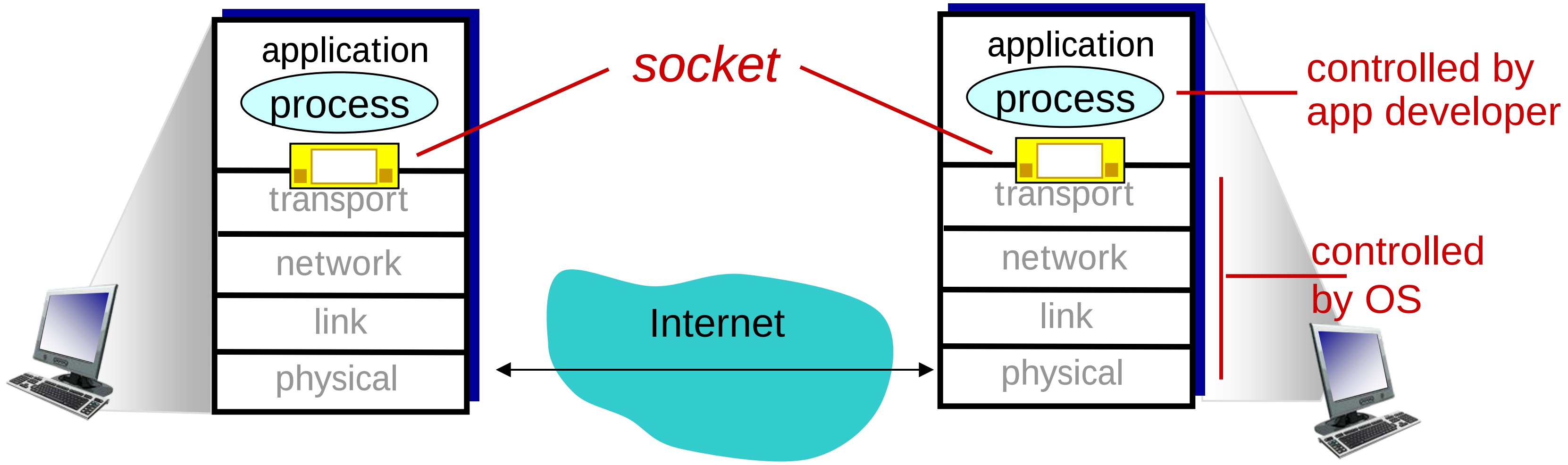
client process: process that initiates communication

server process: process that waits to be contacted

- aside: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host
- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

What transport service does an app need?

data integrity

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

timing

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

throughput

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

security

- encryption, data integrity, …

Transport service requirements: common apps

application	data loss	throughput	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: 5kbps-1Mbps video:10kbps-5Mbps	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
text messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

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

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, orconnection setup,

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

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

Chapter 2: outline

Web and HTTP

First, a review…

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

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

URL - Uniform Resource Locator

- A Uniform Resource Locator (URL) is used to address a document (or other data) on the web.
- A web address, like

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

follows these syntax rules:

`protocol://domain:port/path/filename`

  - `protocol:` defines the type of Internet service (www service uses protocol **http or https**)
  - `domain:` defines the Internet **domain name**
  - `Port:` defines the **port number** at the host (default for http is **80**)
  - `path:` defines a **path** at the server (If omitted: the root directory of the site)
  - `filename:` defines the name of a document or resource

HTML
HyperText Mark-up Language (RFC 1866)

- HTML is the set of markup symbols or codes inserted in a file intended for display on a World Wide Web browser page
- The markup tells the Web browser how to display a Web page's words and images for the user
- Each individual markup code is referred to as a tag
  - Some tags come in pairs that indicate when some display effect is to begin and when it is to end.

HTML Links - Hyperlinks

- HTML links are hyperlinks
- You can click on a link and jump to another document
- In HTML, links are defined with the **<a>** tag:

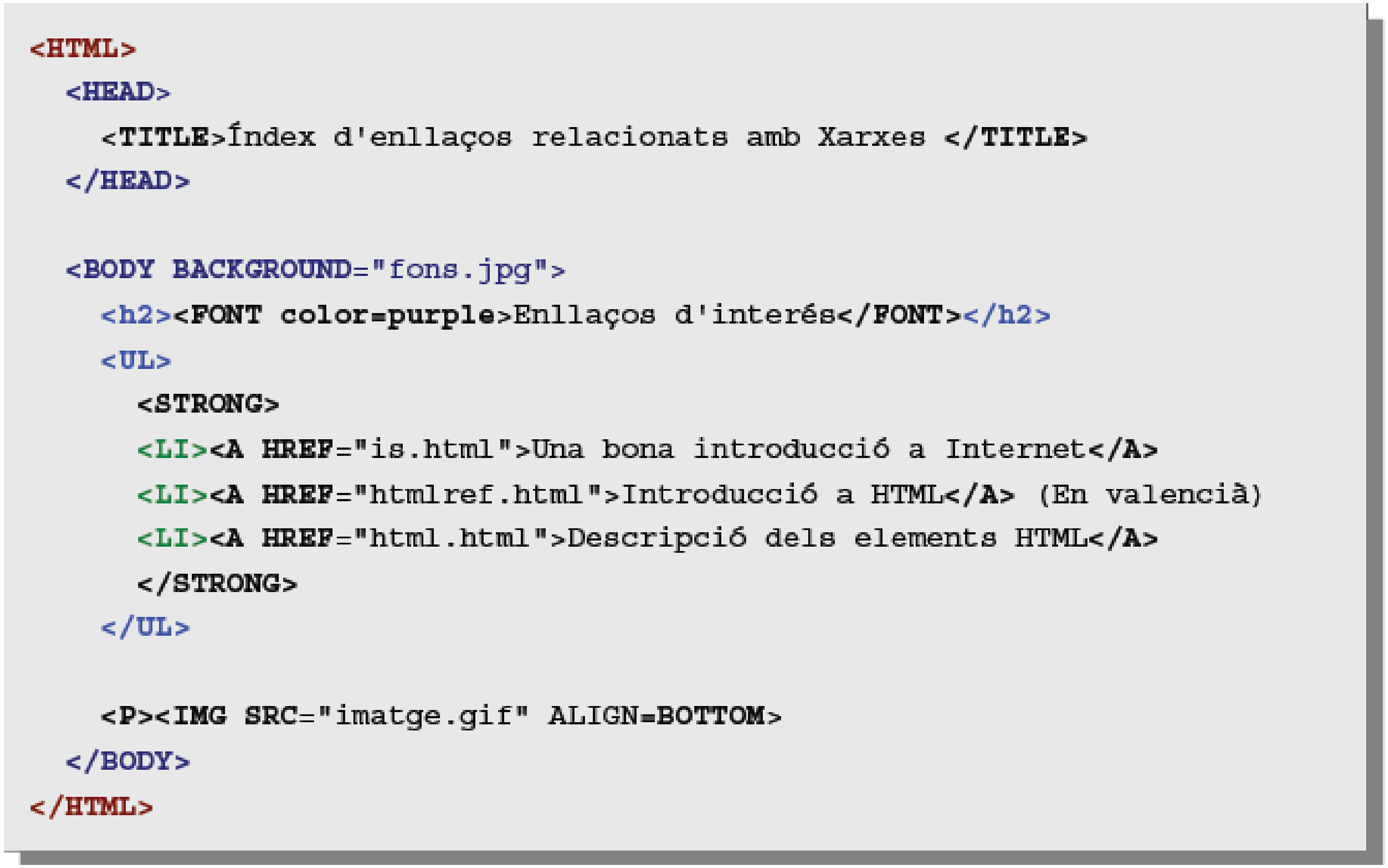
  <a href="*url*">*link text*</a>

- Example:

<a href="https://www.w3schools.com/html/">Visit our HTML tutorial</a>

- The href attribute specifies the destination address (https://www.w3schools.com/html/) of the link
- Local Links: <a href="html_images.asp">HTML Images</a>
  - A local link (link to the same web site) is specified with a relative URL (without http://www....).

HTML Example

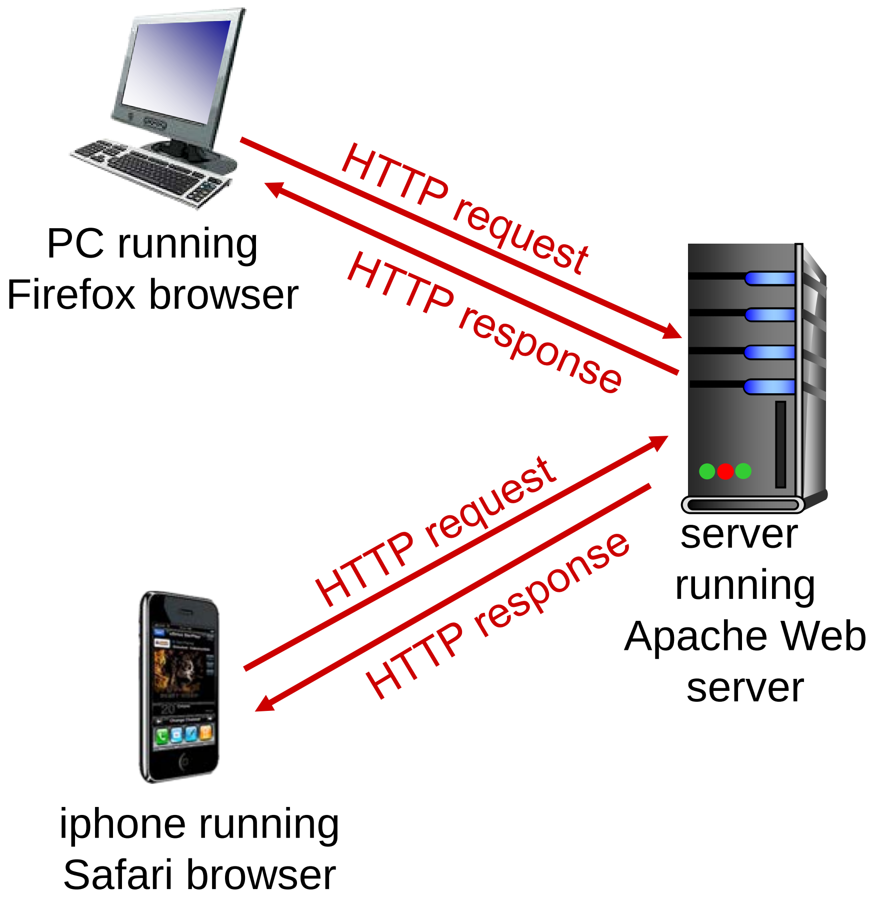


```html
<HTML>
  <HEAD>
    <TITLE>Índex d'enllaços relacionats amb Xarxes </TITLE>
  </HEAD>

  <BODY BACKGROUND="fons.jpg">
    <h2><FONT color=purple>Enllaços d'interés</FONT></h2>
    <UL>
      <STRONG>
      <LI><A HREF="is.html">Una bona introducció a Internet</A>
      <LI><A HREF="htmlref.html">Introducció a HTML</A> (En valencià)
      <LI><A HREF="html.html">Descripció dels elements HTML</A>
      </STRONG>
    </UL>

    <P><IMG SRC="imatge.gif" ALIGN=BOTTOM>
  </BODY>
</HTML>
```

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests
- Versions:
  - HTTP 1.0 (RFC 1945)
  - HTTP 1.1 (RFC 2616)
  - HTTP 2.0 (RFC 7540)


PC running Firefox browser
HTTP request
HTTP response
server running Apache Web server
HTTP request
HTTP response
iphone running Safari browser

HTTP overview (continued)

uses TCP:

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

HTTP is "stateless"

- server maintains no information about past client requests

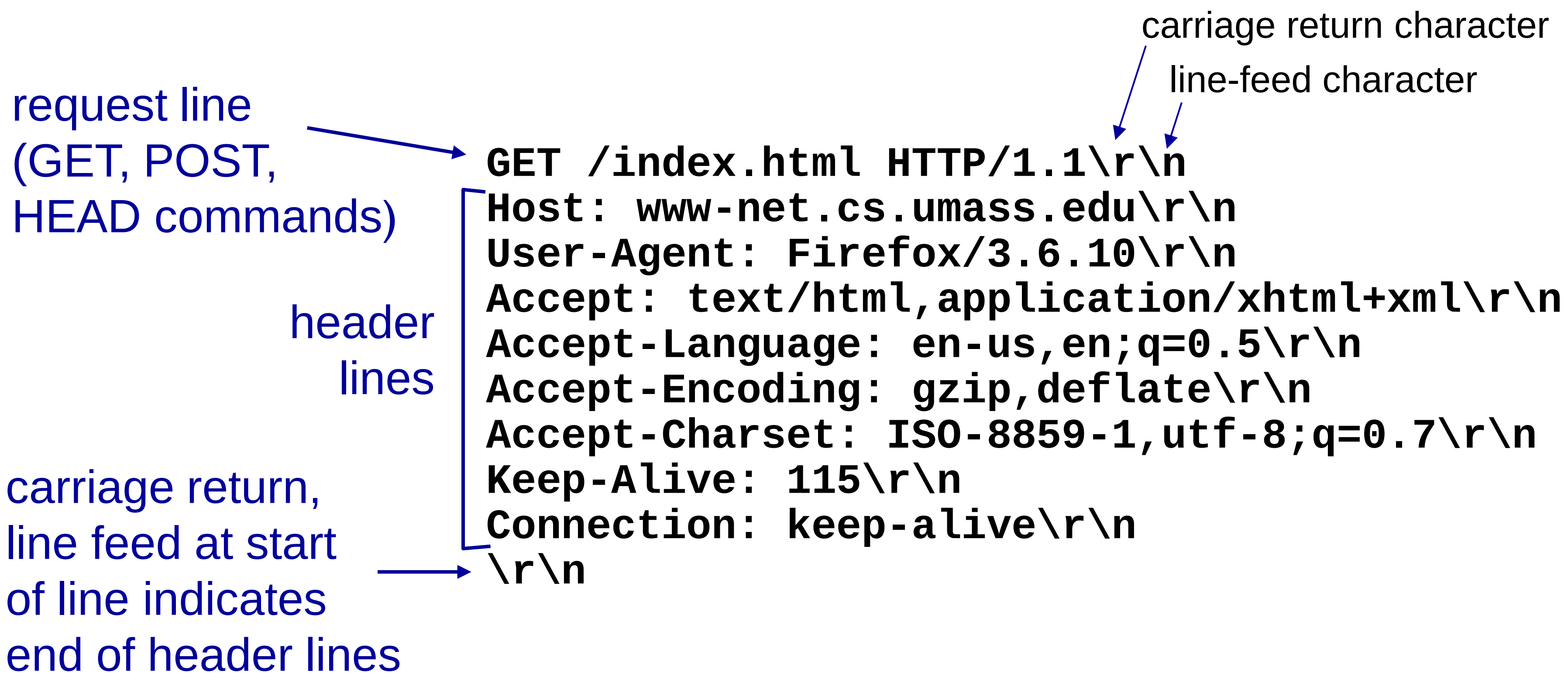
aside

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

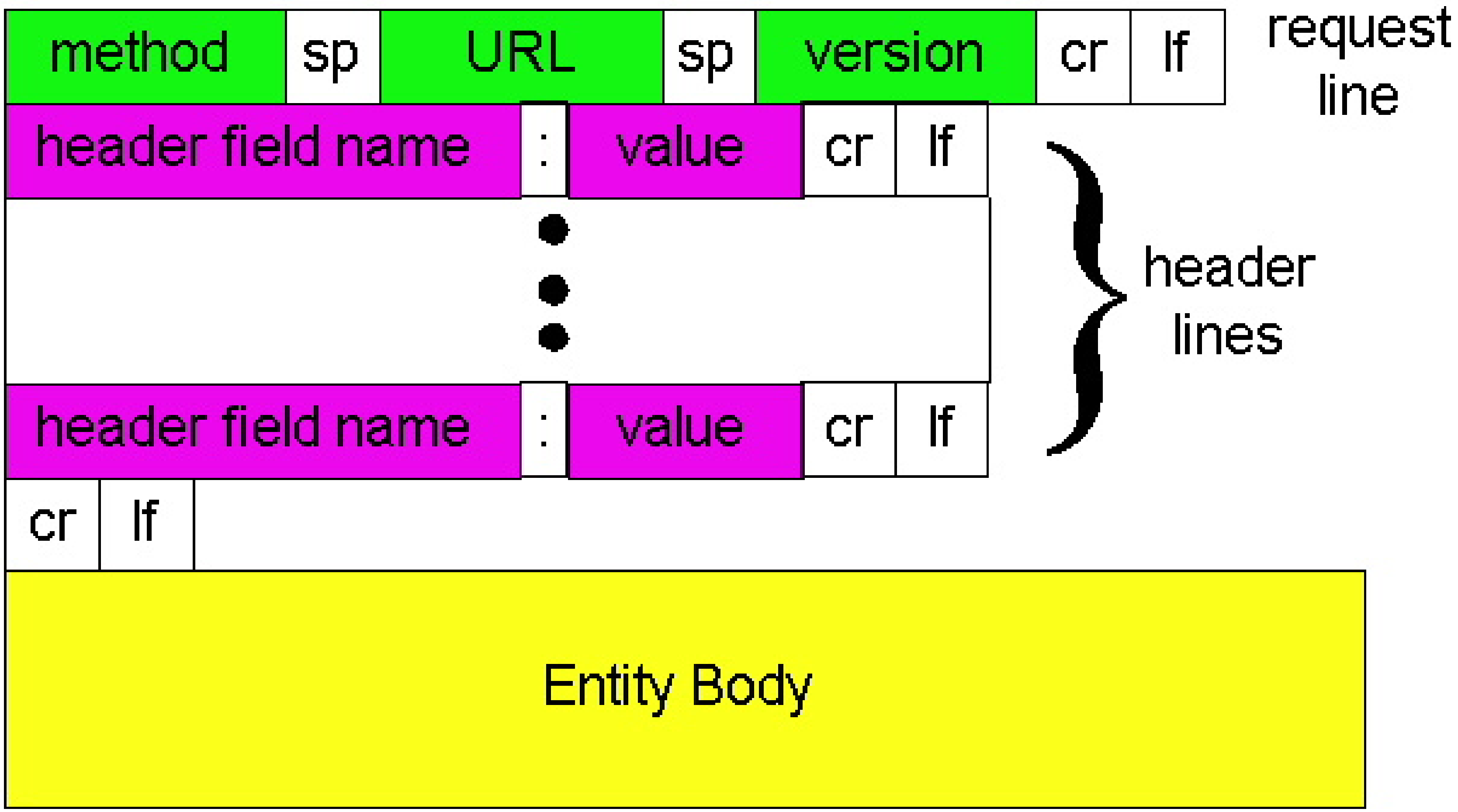
HTTP request message

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


carriage return character
line-feed character
request line (GET, POST, HEAD commands)
header lines
carriage return, line feed at start of line indicates end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP request message: general format



Method types

HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

Uploading form input

POST method:

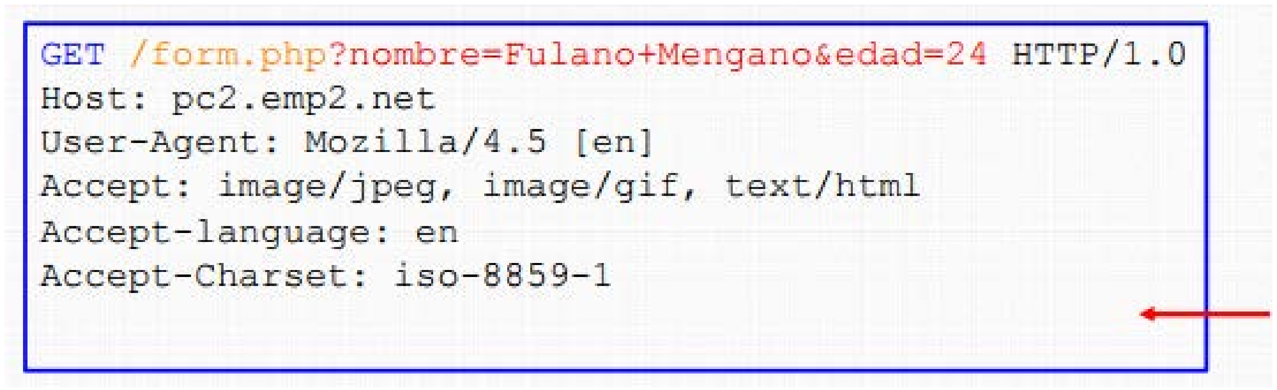
- web page often includes form input
- input is uploaded to server in entity body

URL method:

- uses GET method
- input is uploaded in URL field of request line:

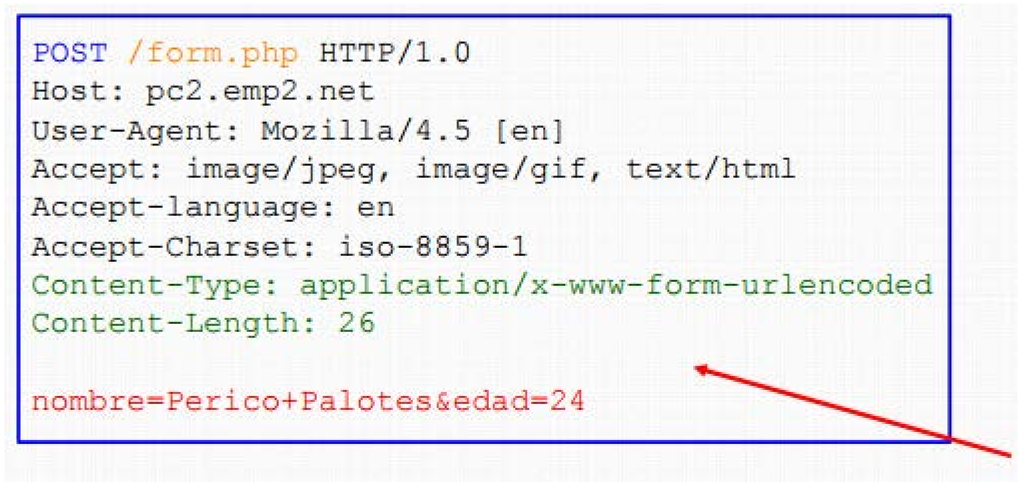
```
www.somesite.com/animalsearch?monkeys&banana
```

Example of GET and POST methods



```
GET /form.php?nombre=Fulano+Mengano&edad=24 HTTP/1.0
Host: pc2.emp2.net
User-Agent: Mozilla/4.5 [en]
Accept: image/jpeg, image/gif, text/html
Accept-language: en
Accept-Charset: iso-8859-1

```

Blank line indicates the end of the header



```
POST /form.php HTTP/1.0
Host: pc2.emp2.net
User-Agent: Mozilla/4.5 [en]
Accept: image/jpeg, image/gif, text/html
Accept-language: en
Accept-Charset: iso-8859-1
Content-Type: application/x-www-form-urlencoded
Content-Length: 26

nombre=Perico+Palotes&edad=24
```

Blank line indicates the end of the header

HTTP response message

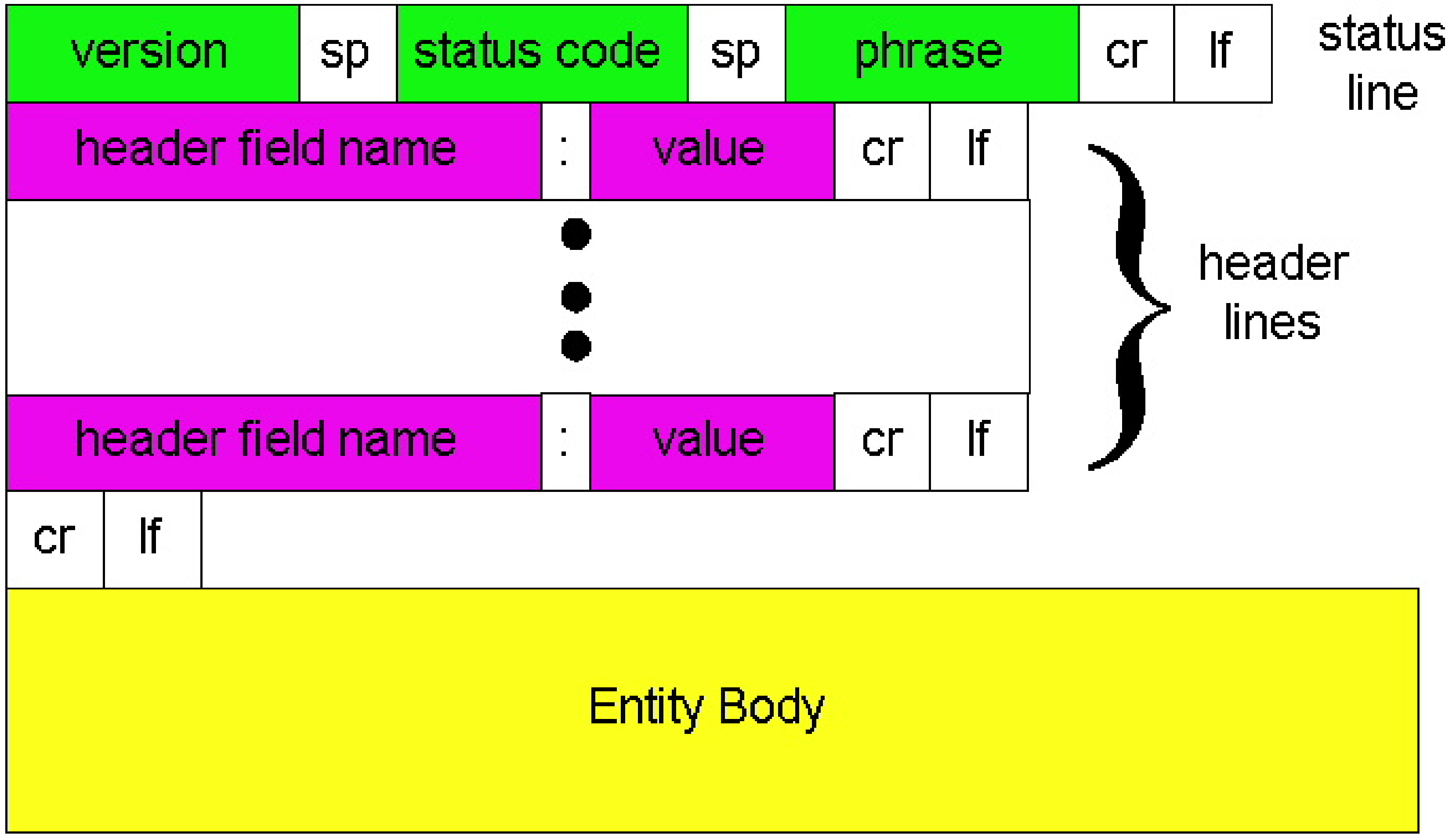
status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

HTTP response message



HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- status codes:
- **1xx Informational:**
  - This class of status code **indicates** information to the client, consisting only of the Status-Line and optional headers, and is terminated by an empty line
  - There are no required headers for this class of status code
- **2xx Success:**
  - This class of status code indicates that the **client's request was successfully received, understood, and accepted**
  - This class of status code indicates that the client's request was successfully received, understood, and accepted.

HTTP response status codes

- **3xx Redirection:**
  - This class of status code indicates that further action needs to be taken by the **user agent** in order to fulfill the request, as **redirect a request to another URL**
  - The action required MAY be carried out by the user agent without interaction with the user if and only if the method used in the second request is GET or HEAD
  - A user agent should not automatically redirect a request more than five times, since such redirections usually indicate an infinite loop.

HTTP response status codes

- **4xx Client Error**
  - The 4xx class of status code is intended for cases in which the **client seems to have erred**
  - Except when responding to a HEAD request, the server SHOULD include an explanation of the error situation, and whether it is a temporary or permanent condition
- **5xx Server Error**
  - Response status codes beginning with the digit "5" indicate cases in which the **server is aware that it has erred or is incapable of performing the request**
  - Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition

HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

`200 OK`
  - request succeeded, requested object later in this msg

`301 Moved Permanently`
  - requested object moved, new location specified later in this msg (Location:)

`400 Bad Request`
  - request msg not understood by server

`404 Not Found`
  - requested document not found on this server

`505 HTTP Version Not Supported`

HTTP Headers

- The header fields are transmitted after the request or response line (the first line of a message)
- Header fields are colon-separated name-value pairs in clear-text string format, terminated by a carriage return (CR) and line feed (LF) character sequence
- The end of the header section is indicated by an empty field, resulting in the transmission of two consecutive CR-LF pairs
- HTTP 1.0 defines 16 headers, none mandatory
- HTTP 1.1 defines 46 headers, only one (Host:) is mandatory since HTTP/1.1

Some important headers

- Request:
  - User-Agent: The browser used by the client
  - Accept: Content-Types that are acceptable for the response
  - Host: The domain name of the server, mandatory since HTTP/1.1
- Response:
  - Content-Type: The MIME type of the body, e.g. text/html, image/gif
  - Content-Length: The length of the request body in octets (8-bit bytes)
  - Date: The date and time that the message was originated

Request – Response Example

Request example

```
GET /img/sic/pixelb.gif HTTP/1.1
User-Agent:Mozilla/4.76 (Windows NT 5.0; U)
Host: sic.uji.es
Accept: image/gif, image/x-xbitmap, image/jpeg
Accept-Charset: iso-8859-1,*,utf-8
```

Response example

```
HTTP/1.1 200 OK
Date: Fri, 07 Dec 2009 13:02:54 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.21
Last-Modified: Fri, 16 Jun 2008 16:49:46 GMT
Accept-Ranges: bytes
Content-Length: 799
Content-Type: image/gif

<fitxer pixelb.gif>
```

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/ HTTP/1.1
Host: cis.poly.edu
```

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

3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

HTTP connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections
- Used by default in HTTP / 1.0
- It can be modified including the header connection

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server
- Used by default in HTTP / 1.1
- It can be modified including the header connection

Non-persistent HTTP

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 on port 80

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. Message indicates that client wants object someDepartment/home.index

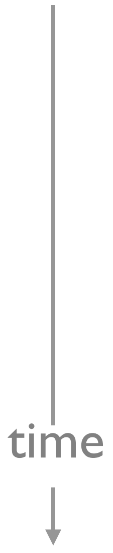
3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Non-persistent HTTP (cont.)

4. HTTP server closes TCP connection.

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



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

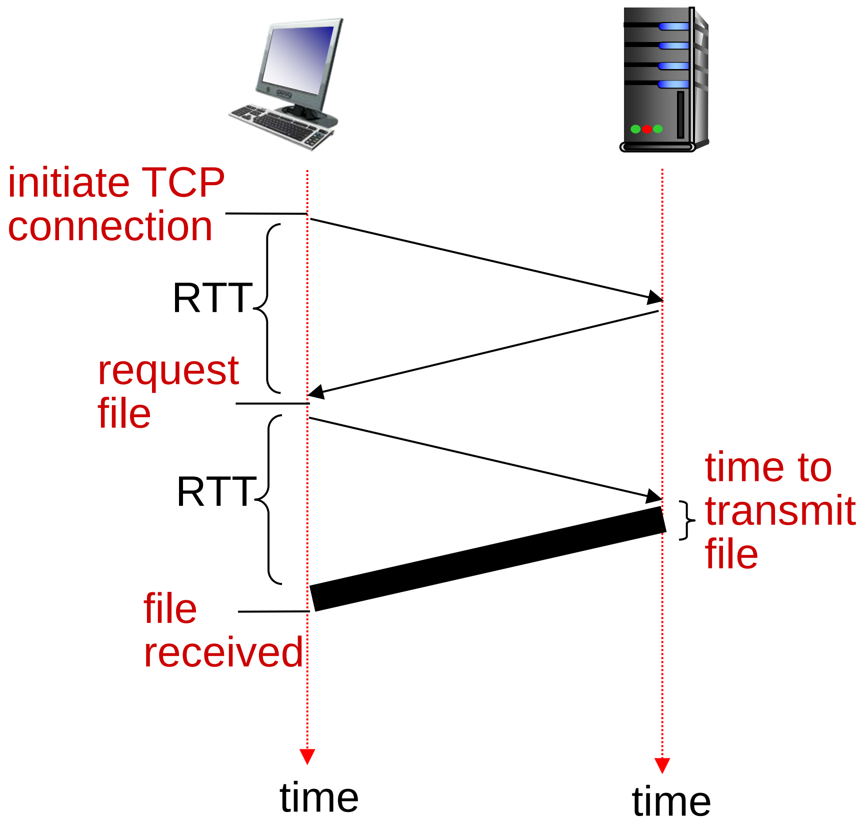
Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back

HTTP response time:

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2RTT+ file transmission time



Non-persistent HTTP disadvantages

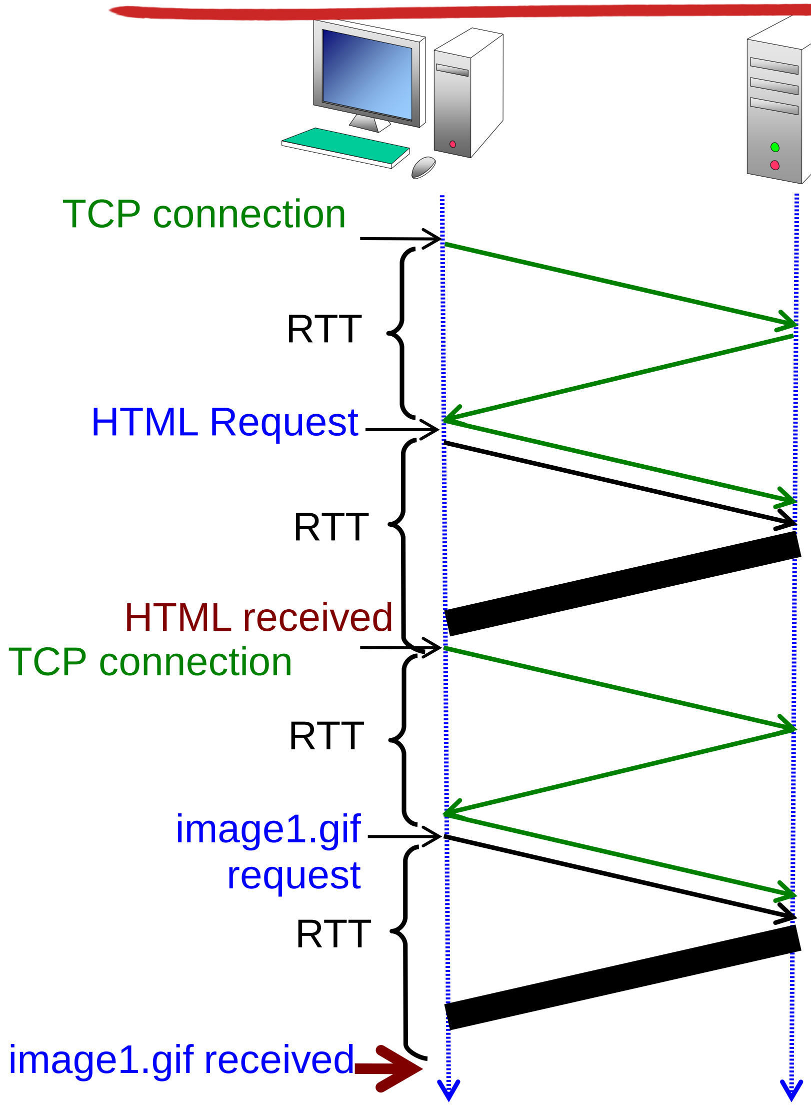
- It needs 2 RTT for each object
- The OS must control each TCP connection
- Browsers typically open several concurrent TCP connections to transport the referenced objects
  - Overload on servers
  - A maximum of 4 concurrent connections are recommended on the same server
- The client can request that the server keep the connection open
  - Header : Connection: Keep_alive
- The server can accept or not the request
  - Header : Connection: close

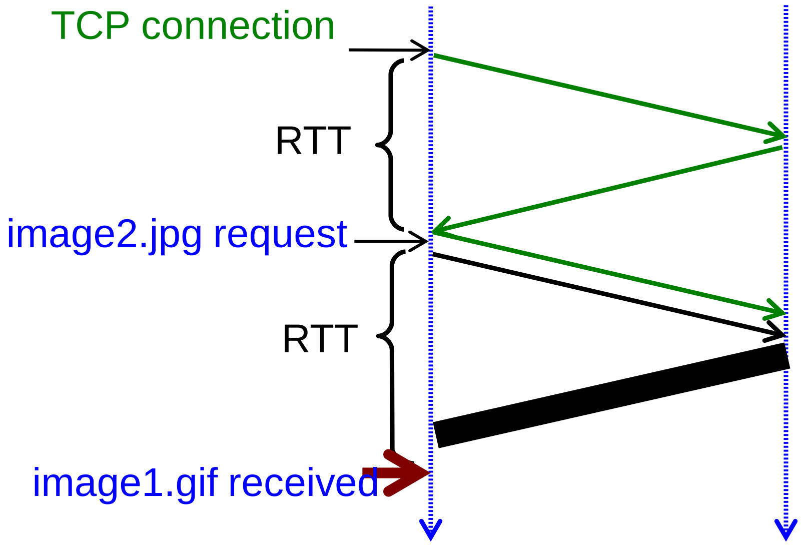
Persistent HTTP

- Default behaviour of HTTP 1.1 and enhanced versions
- Server leaves connection open after sending response
- Subsequent HTTP messages between same client/server sent over open connection
  - Headers:

    ```
    Connection: keep-alive <CR><LF>
    Keep-alive: valor_segs <CR><LF>
    ```
- Client sends requests as soon as it receives the previous demanded object
- Pipelining;
  - Client sends requests as soon as it encounters a referenced object without waiting to receive the previous demanded object
  - as little as one RTT for all the referenced objects

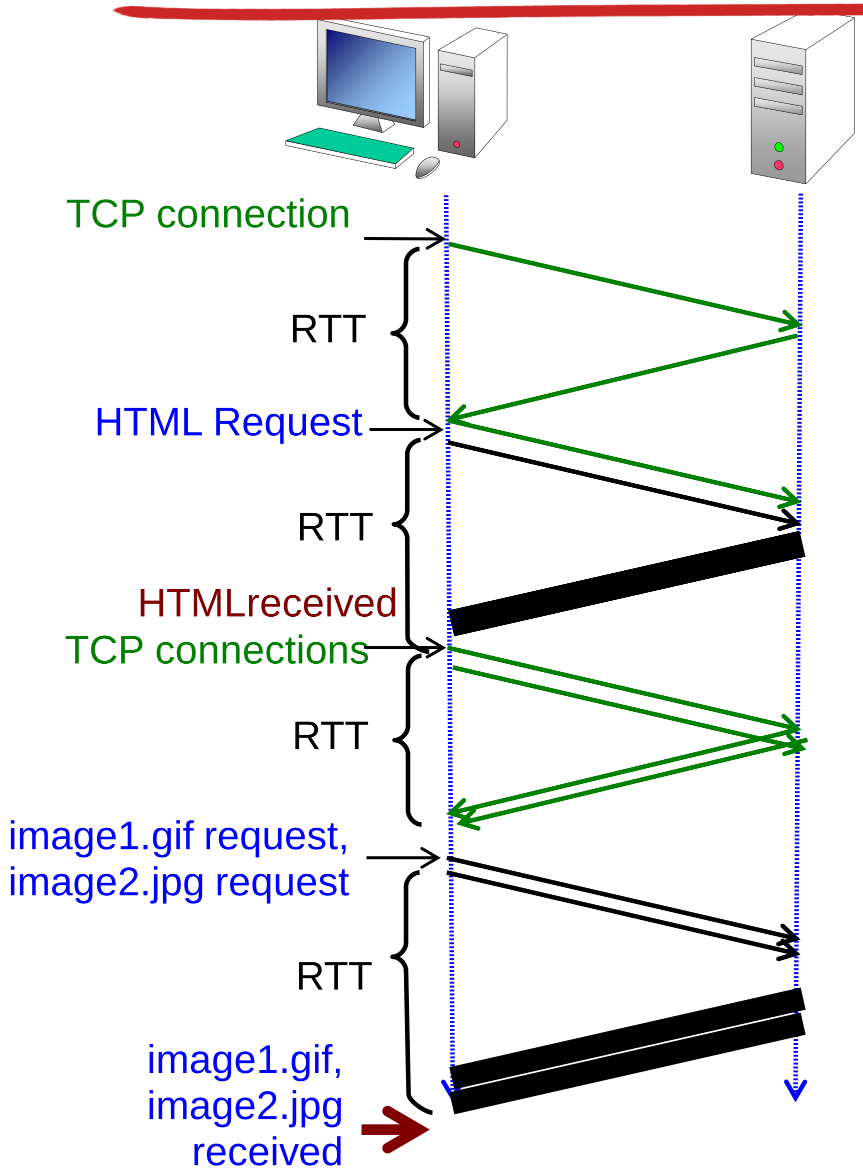
Non-persistent HTTP response time





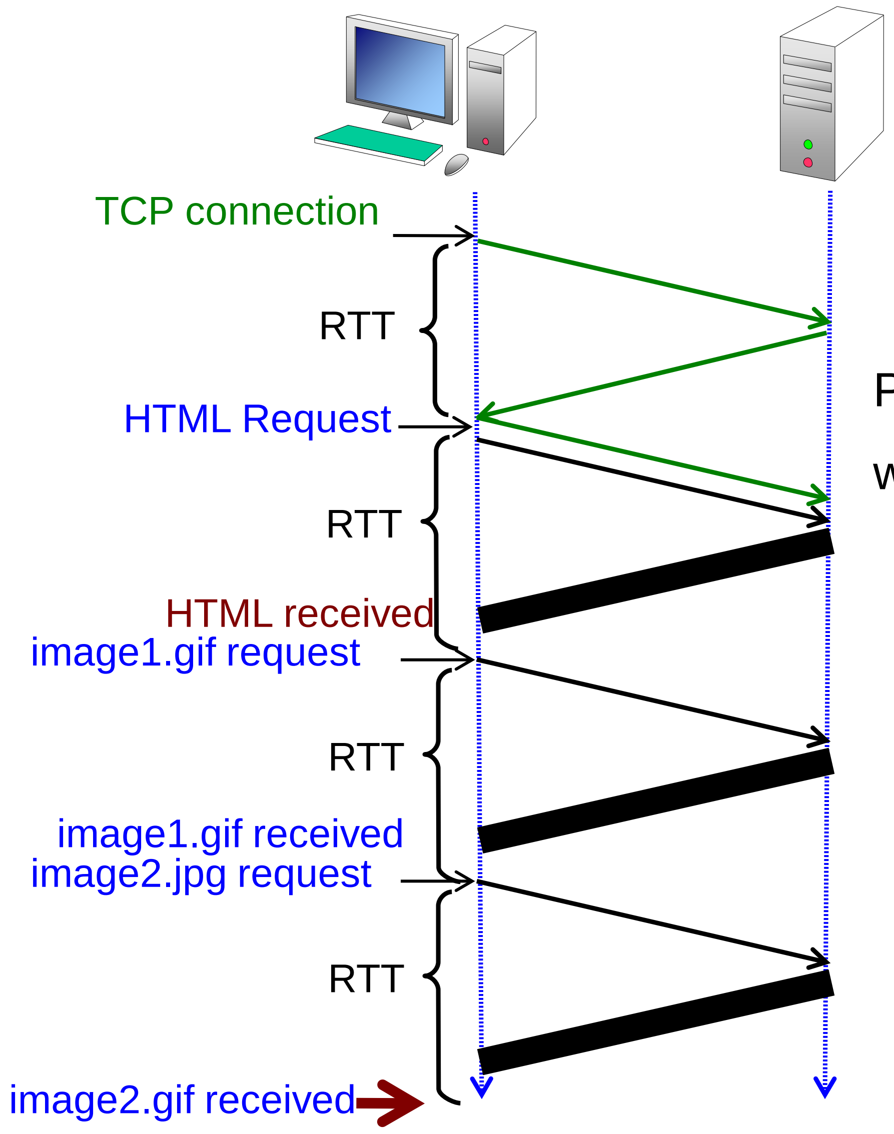
non-persistent HTTP connection
(without pipelining)

Non-persistent HTTP response time



non-persistent HTTP connection (with pipelining)

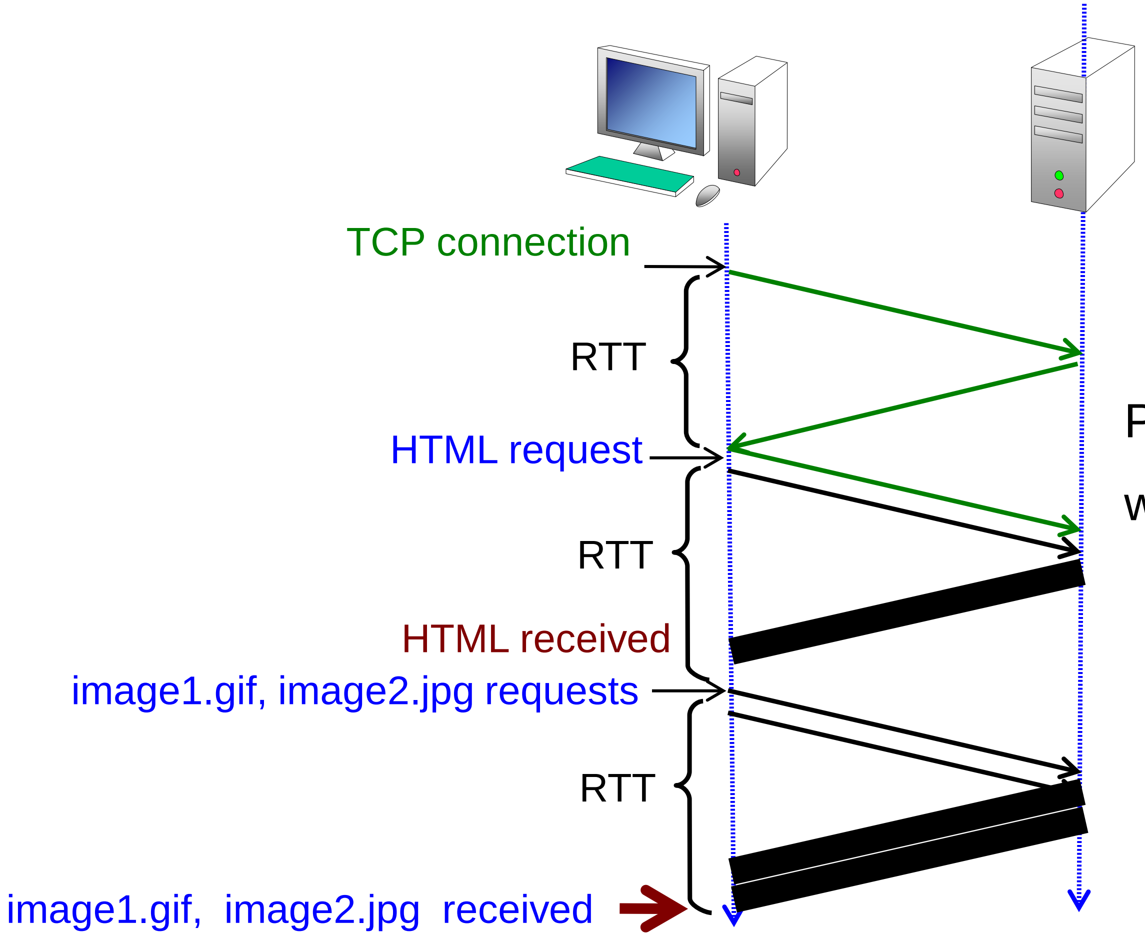
Persistent HTTP response time



Persistent HTTP connection

without pipelining

Persistent HTTP response time


TCP connection
RTT
HTML request
RTT
HTML received
image1.gif, image2.jpg requests
RTT
image1.gif, image2.jpg received

Persistent HTTP connection

with pipelining

Text from: https://http2.github.io/faq/#why-revise-http

Why revise HTTP?

- Loading a Web page is more resource intensive than ever, and loading all of those assets efficiently is difficult
- HTT/1.0 and HTTP/1.1have used multiple TCP connections to issue parallel requests.
  - However, there are limits to this; if too many connections are used, it's both counter-productive
    - TCP congestion control is effectively negated, leading to congestion events that hurt performance and the network and it's fundamentally unfair (because browsers are taking more than their share of network resources).
- At the same time, the large number of requests means a lot of duplicated data "on the wire"
- Both of these factors means that HTTP/1.1 requests have a lot of overhead associated with them

What are the key differences to HTTP/1.x?

- At a high level, HTTP/2:
  - is binary, instead of textual
  - is fully multiplexed, instead of ordered and blocking
    - *can therefore use one connection for parallelism*
  - uses header compression to reduce overhead
  - allows servers to "push" responses proactively into client caches
  - Flow control
    - *Flow control is used for both individual streams and for the connection as a whole*
  - Stream priority

Why is HTTP/2 binary?

- Binary protocols are more efficient to parse, more compact "on the wire", and most importantly
- They are much less error-prone, compared to textual protocols like HTTP/1.x, because they often have a number of affordances to "help" with things like whitespace handling, capitalization, line endings, blank lines and so on.
- It's true that HTTP/2 isn't usable through telnet, but we already have some tool support, such as a Wireshark plugin

Why is HTTP/2 multiplexed?

- HTTP/1.x has a problem called "head-of-line blocking," where effectively only one request can be outstanding on a connection at a time.
  - HTTP/1.1 tried to fix this with pipelining, but it didn't completely address the problem (a large or slow response can still block others behind it)
  - Additionally, pipelining has been found very difficult to deploy, because many intermediaries and servers don't process it correctly.
  - This forces clients to use a number of heuristics (often guessing) to determine what requests to put on which connection to the origin when
  - this can severely impact performance, often resulting in a "waterfall" of blocked requests.
- Multiplexing addresses these problems by allowing multiple request and response messages to be in flight at the same time; it's even possible to intermingle parts of one message with another on the wire.
- This, in turn, allows a client to use just one connection per origin to load a page.

Why do we need header compression?

- If you assume that a page has about 80 assets (which is conservative in today's Web), and each request has 1400 bytes of headers (again, not uncommon, thanks to Cookies, Referer, etc.), it takes at least 7-8 round trips to get the headers out "on the wire." That's not counting response time - that's just to get them out of the client.
  - This is because of TCP's Slow Start mechanism, which paces packets out on new connections based on how many packets have been acknowledged – effectively limiting the number of packets that can be sent for the first few round trips.
- In comparison, even mild compression on headers allows those requests to get onto the wire within one roundtrip – perhaps even one packet.
- This overhead is considerable, especially when you consider the impact upon mobile clients, which typically see round-trip latency of several hundred milliseconds, even under good conditions.

What's the benefit of Server Push?

- When a browser requests a page, the server sends the HTML in the response, and then needs to wait for the browser to parse the HTML and issue requests for all of the embedded assets before it can start sending the JavaScript, images and CSS.
- Server Push potentially allows the server to avoid this round trip of delay by "pushing" the responses it thinks the client will need into its cache.
- However, Pushing responses is not "magical" – if used incorrectly, it can harm performance. Correct use of Server Push is an ongoing area of experimentation and research.

Flow Control

- Using streams for multiplexing introduces contention over use of the TCP connection, resulting in blocked streams. A flow-control scheme ensures that streams on the same connection do not destructively interfere with each other. Flow control is used for both individual streams and for the connection as a whole.
- Flow control is defined to protect endpoints that are operating under resource constraints
- For example, a proxy needs to share memory between many connections and also might have a slow upstream connection and a fast downstream one
- Flow-control addresses cases where the receiver is unable to process data on one stream yet wants to continue to process other streams in the same connection.

Stream Priority

- The purpose of prioritization is to allow an endpoint to express how it would prefer its peer to allocate resources when managing concurrent streams
- Most importantly, priority can be used to select streams for transmitting frames when there is limited capacity for sending
- A client can assign a priority for a new stream by including prioritization information in the HEADERS frame that opens the stream
  - At any other time, the PRIORITY frame can be used to change the priority of a stream.

HTTP/2

- If a request is made without knowing if the server supports HTTP/2, Client includes headers: "Upgrade" and "HTTP2-Settings"

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64 url encoding of HTTP/2 SETTINGS payload>
```

- If Server doesn't support HTTP/2, it will answer:

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html
        …
```

- If Server supports HTTP/2, it will answer:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[HTTP/2 connection ...
```

User-server state: cookies

many Web sites use cookies

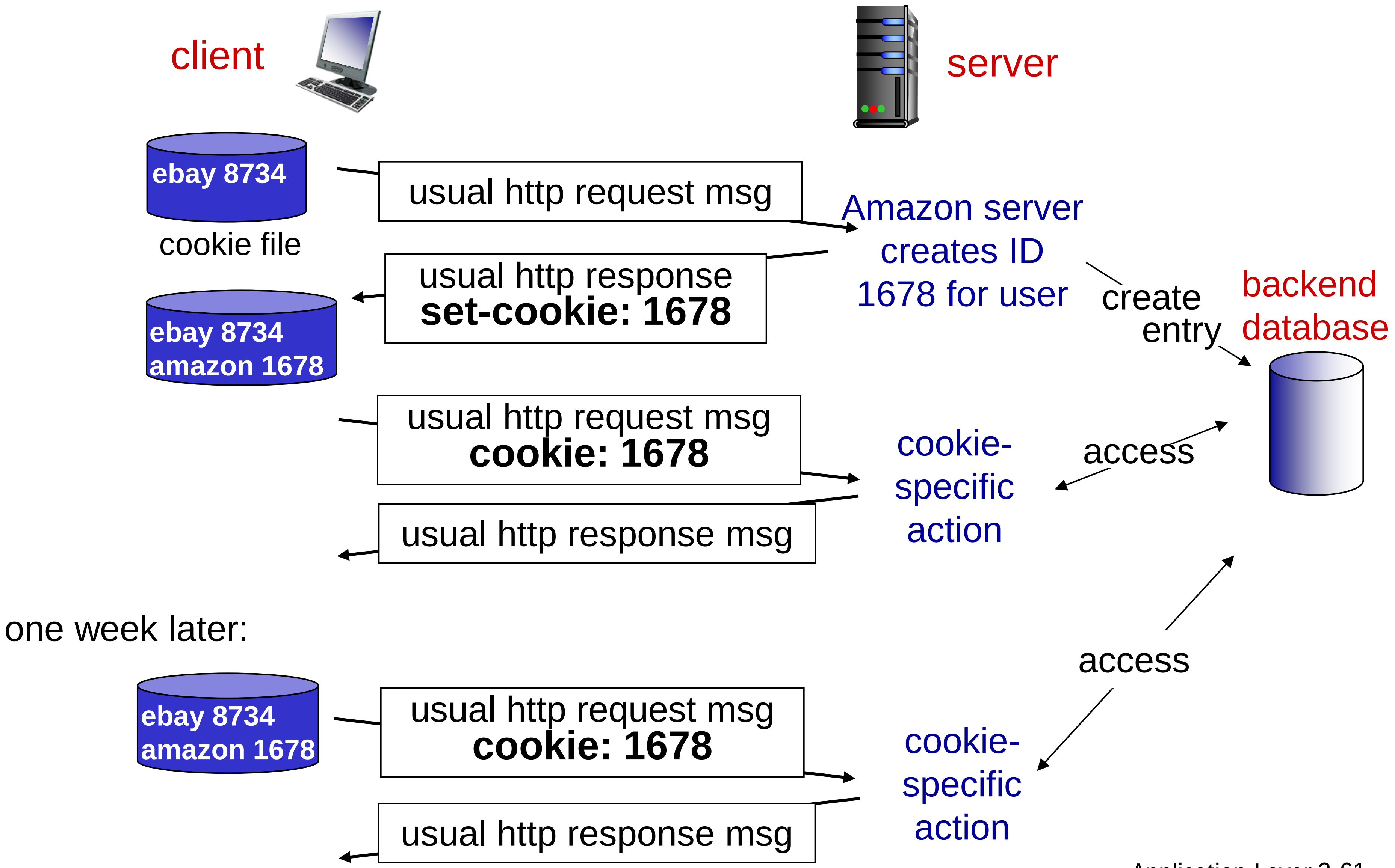
four components:

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

example:

- Susan always access Internet from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping "state" (cont.)



Cookies (continued)

what cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

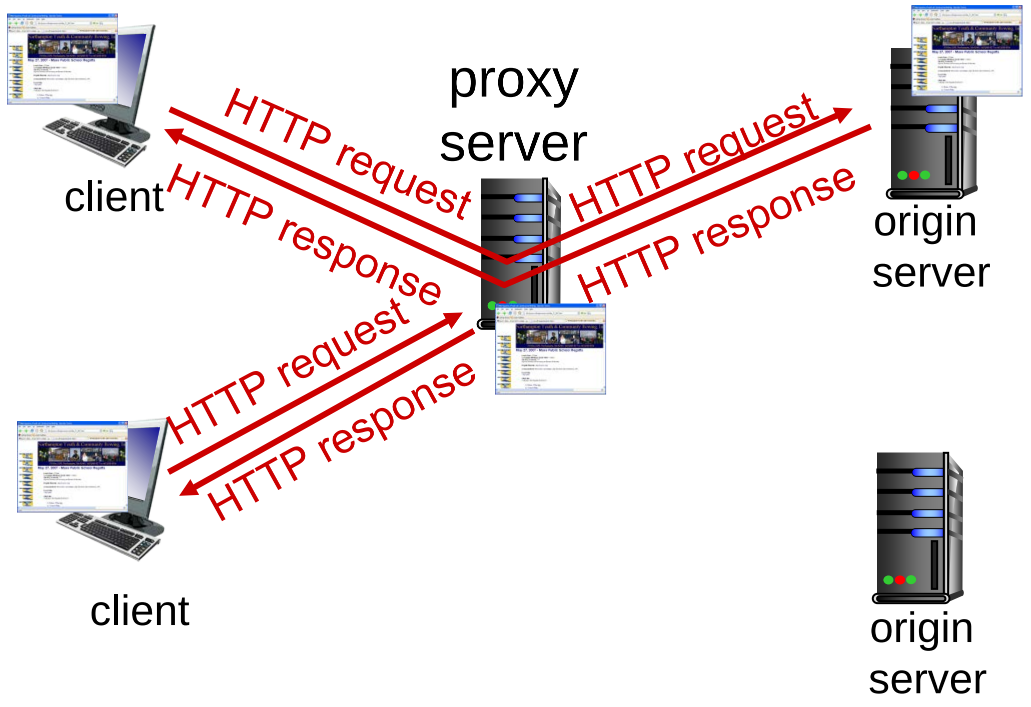
how to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Web caches (proxy server)

goal: satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



More about Web caching

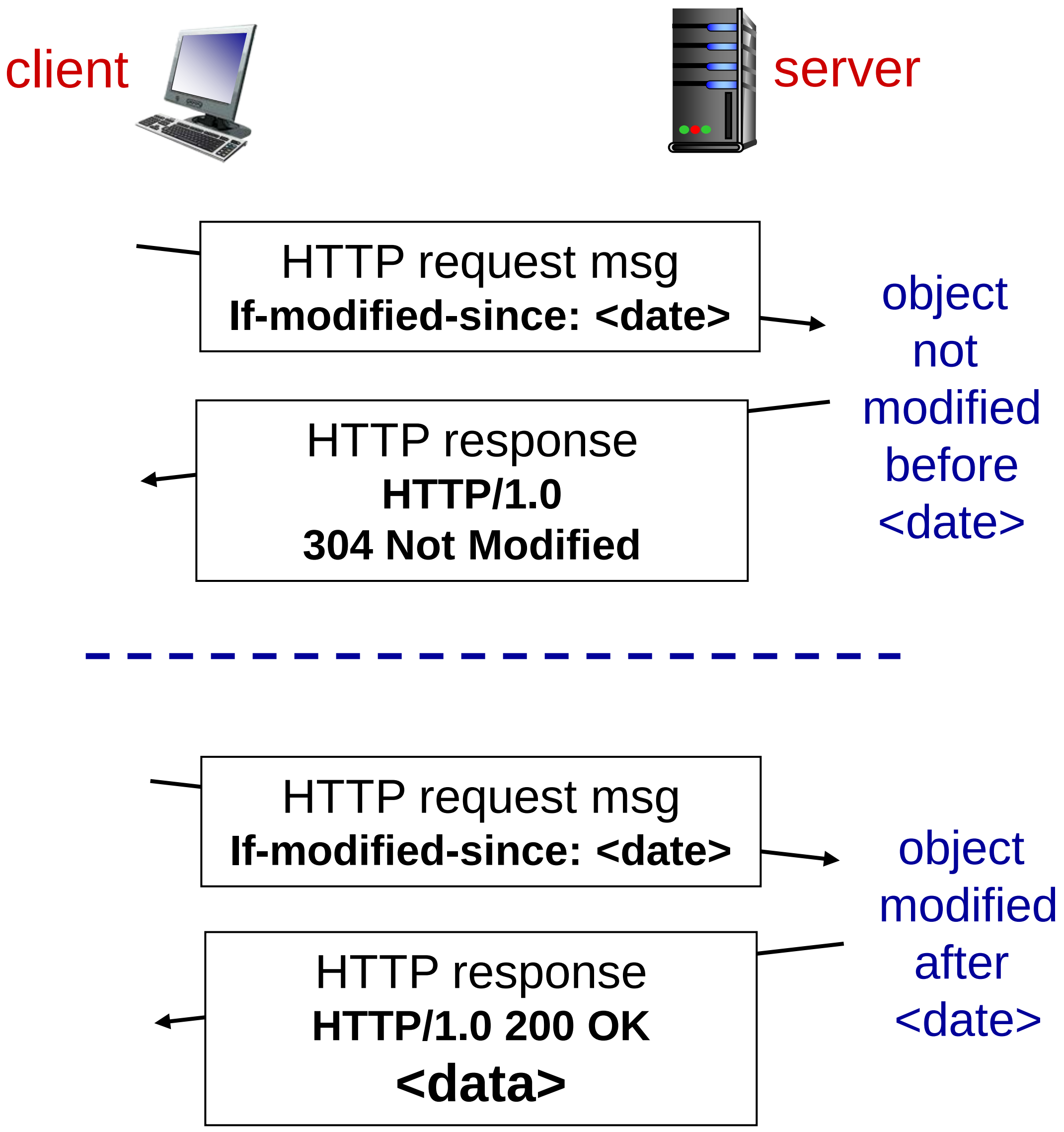
- cache acts as both client and server
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

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 (so too does P2P file sharing)

Conditional GET

- *Goal:* don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *cache:* 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`



Conditional GET Example

Request Header Example:

```
GET /easy/http/ HTTP/1.1
If-Modified-Since: Wed, 13 Sep 2009 22:51:57 GMT; length=45531
Referer: http://www.google.com/search?q=http+tutorial
User-Agent: Mozilla/4.76 (Windows NT 5.0; U)
Host: www.jmarshall.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg
Accept-Encoding: gzip
Accept-Charset: iso-8859-1,utf-8
```

- Response Header Example:

```
HTTP/1.1 304 Not Modified
Date: Fri, 07 Dec 2010 11:29:00 GMT
Server: mod_jk FrontPage/4.0.4.3 Confluence Apache/1.3.20 (Unix)
```

Chapter 2: outline

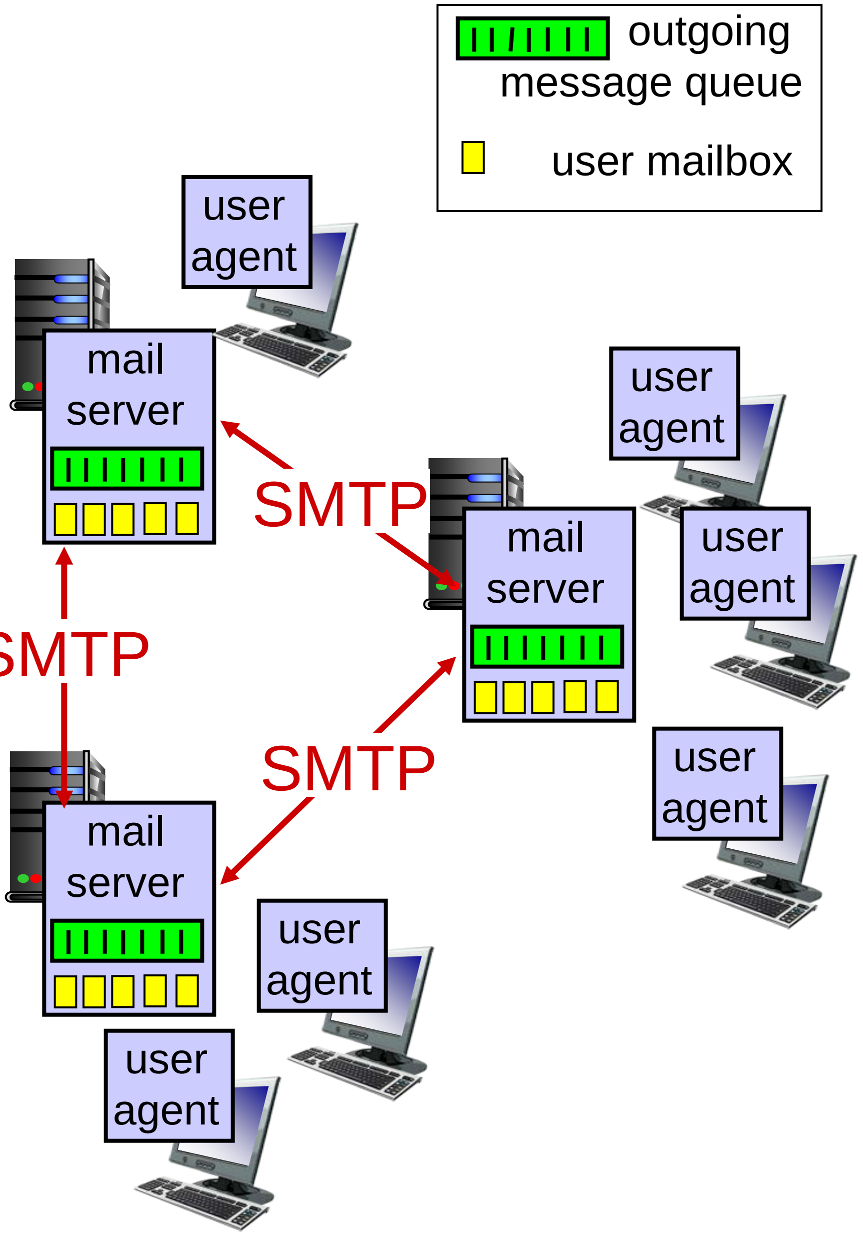
Electronic mail

Three major components:

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

User Agent

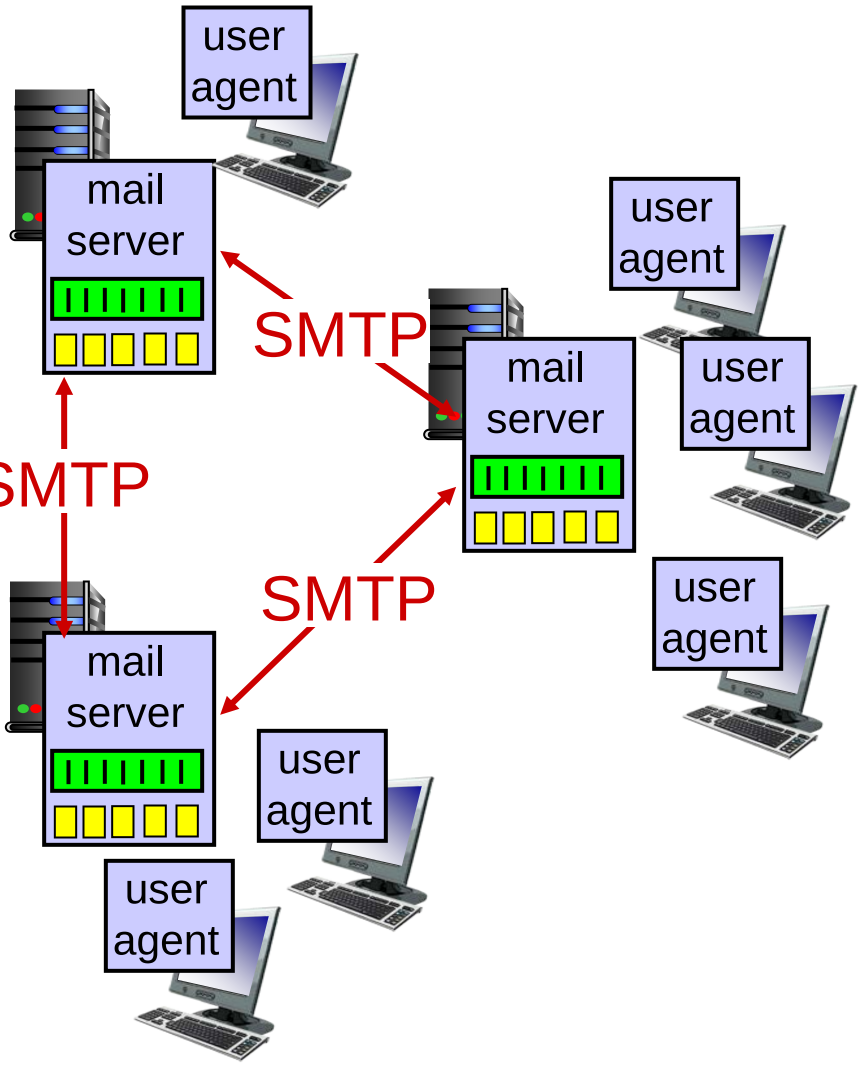
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



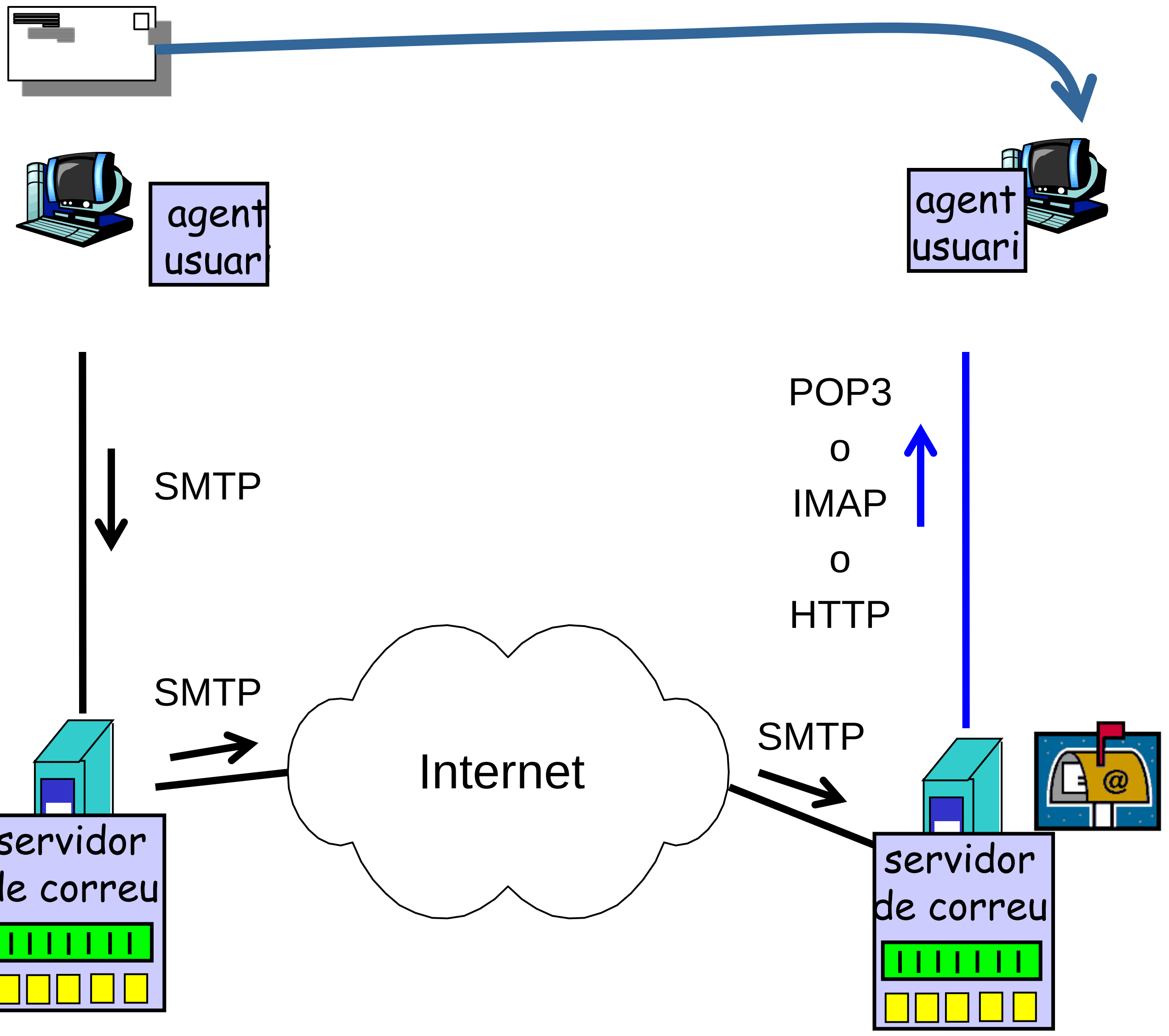
Electronic mail: mail servers

mail servers:

- *mailbox* contains incoming messages 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



Email Addresses

- Format:

  mailbox@ mail_domain

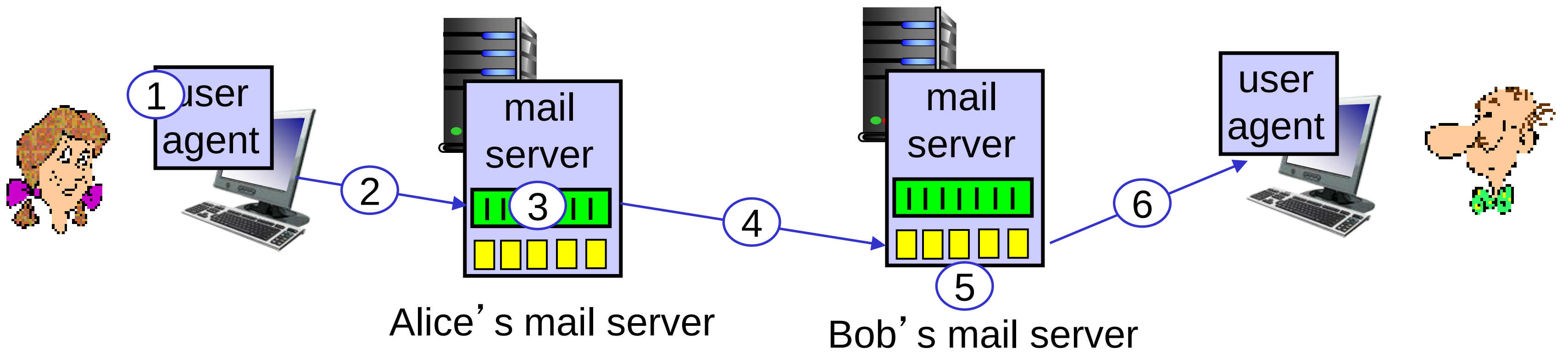
- Usually **mailbox** is the user in the destination end system
  - *Aliases* can be created
    - e.g.: profes@redes.upv.es = {joan, pepe, ana}@redes.upv.es
  - Format of **mail_domain**: names separates by periods (e.g.: disca.upv.es)
  - *A* **mail_domain** can be a host name (e.g..: zoltar.redes.upv.es)

Electronic Mail: SMTP [RFC 5321]

- uses TCP to reliably transfer email message 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 (like HTTP, FTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCI

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message "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



SMTP Protocol Commands

- Commands and responses in ASCII-7 format
- Client commands
  - Always ending with <CR><LF>

HELO domain name	Identify the source connection
MAIL FROM: <source address**>**	Sender
RCPT TO: <destination address**>**	Destination
DATA	Introduction of data
QUIT	Close connection with the SMTP server
RSET	Reset SMTP server connection
HELP	Show help about accepted commands
EXPN <email address>	Expands a mailing list
VRFY <email address>	Verify the existence of add. email

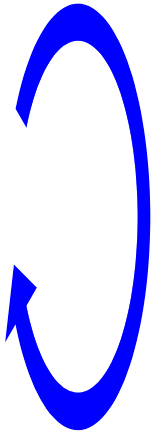
SMTP Protocol

- HELO source.end.system
- Transaction SMTP, 3 steps:
  - MAIL FROM: <ental@macasa.es>
  - RCPT TO: <entalaltre@tacasa.es>
  - DATA

    hello

    bla, bla, bla,...

    .
- QUIT



SMTP Responses

- Server Responses:
  - Three-digit numbers, blank space and text information
  - The answers are usually in a line of text. Some answers using multiple lines of text
- Processing DATA command
  - The server sends a message to the client, and asked to start sending the message
  - The client after sending the message enter the sequence <CR> <LF>. <CR> <LF> to mark end of 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 message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

comparison with HTTP:

- HTTP: pull
- SMTP: push

- both have ASCII command/response interaction, status codes

- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg

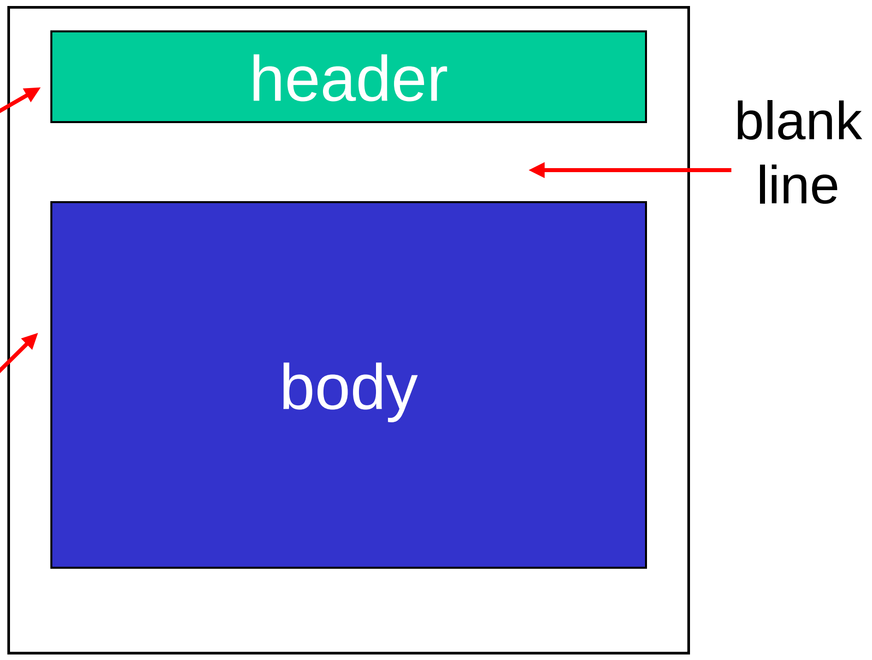
Mail message format

SMTP: protocol for exchanging email msgs

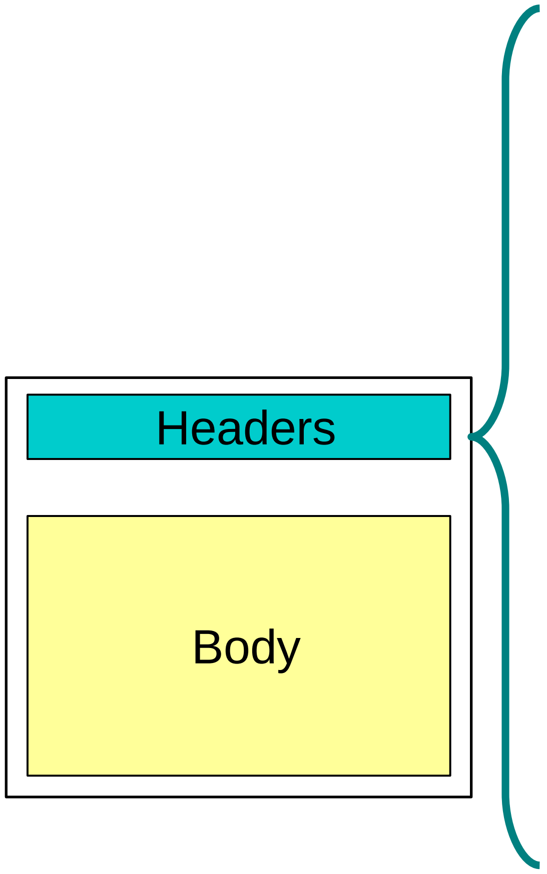
RFC 822: standard for text message format:

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

  different from SMTP MAIL FROM, RCPT TO: commands!
- Body: the "message"
  - ASCII characters only



Message Headers



From:	Source
To:	Main destination
Cc:	Carbon Copy (Secondary destination)
Bcc:	Blind Carbon Copy
Subject:	Topic of the email content
Received:	List of servers that forward the email
Date:	Date and time the email was compossed
Reply-To:	Message address for return mail
Message-Id:	Message Identifier
In-Reply-To:	Message-Id of the message being replied to

…

X-mailer:	Identifies the software the sender used to send the message.

…

email headers definition and example

Email Example

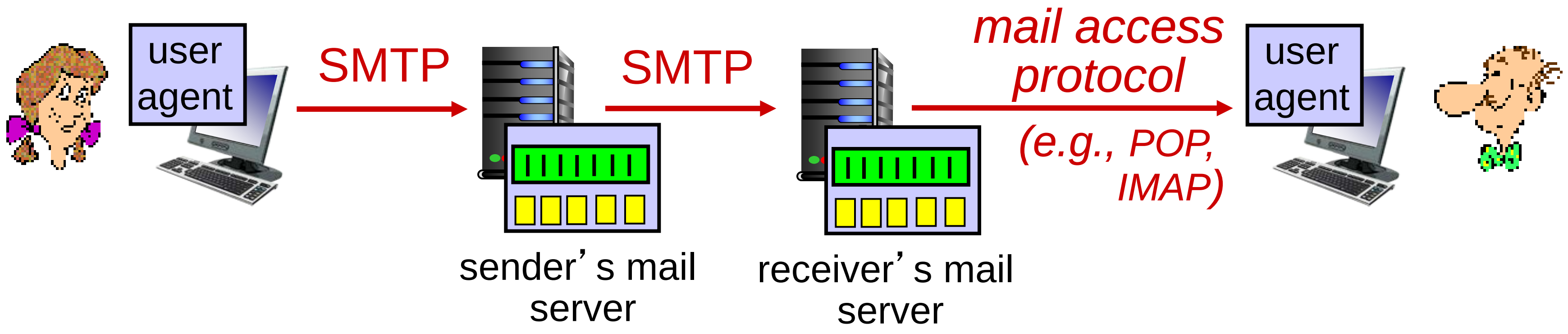
Headers

```
Return-Path: <pepe@disca.upv.es>
Received: from vega.upv.es by disca.upv.es (SMI-8.6/SMI-SVR4)
        id CAA03766; Wed, 24 Jan 2010 02:13:22 GMT
Received: from mailman.endymion.com (antares.cc.upv.es [158.42.3.1])
        by vega.upv.es (8.8.4/8.8.5) with SMTP id BAA00996
        for <juan@disca.upv.es>; Wed, 24 Jan 2010 01:56:46 +0100 (MET)
From: pepe@disca.upv.es
Message-Id: <201001240056.BAA00996@vega.upv.es>
To: juan@disca.upv.es
Subject: leeme
Date: Wed, 24 Jan 2010 01:56:46 +0000

Hola,
bla, bla, bla, bla, bla.
Adeu
```

Blank Line

Body

Mail access protocols

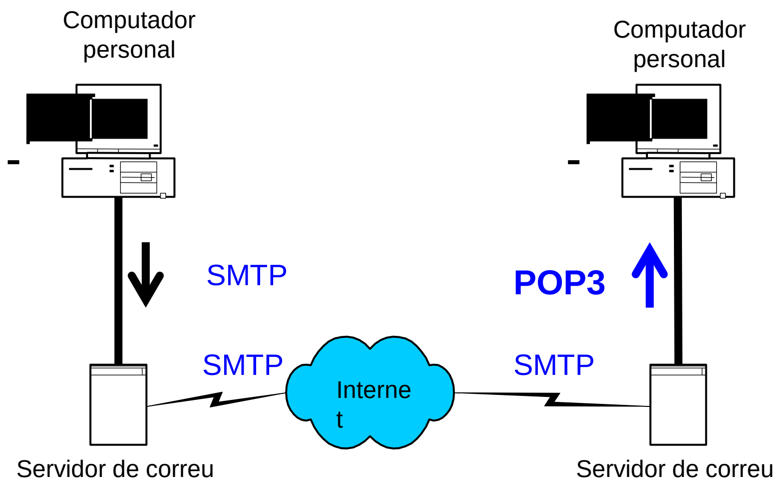


- SMTP: delivery/storage to receiver's server
- mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]: authorization, download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs on server
  - HTTP: gmail, Hotmail, Yahoo!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

Protocol POP3

POP3: *Post Office Protocol* (RFC 1939)

- Allows the user to see (read) your email previously received and stored your local server
- Uses TCP connection, port 110



Fases d’una sessió POP3

- Authorization
  - User name and password
- Transaction
  - Retrieve messages, mark messages to be deleted, statistics shown
- Update
  - After the quit command from the mail client, the message store is updated (delete the messages marked) and the session ends.

POP3 Responses

- All responses from the server will start with +OK or –ERR.
- There is usually more information displayed after this status value (+OK means your command was successful and -ERR that it failed) to indicate why there was an error, or what the result of the command was if it succeeded.
- <CR><LF>. <CR><LF> indicates the end of the response.

POP3 protocol

authorization phase

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

transaction phase, client:

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

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

POP3 (more) and IMAP

more about POP3

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

IMAP

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

MIME
(Multipurpose Internet Mail Extensions)

- MIME is a standard which was proposed by Bell Communications in 1991 in order to expand upon the limited capabilities of email, and in particular to allow documents (such as images, sound, and text) to be inserted in a message. It was originally defined by RFCs 1341 and 1342 in June 1992.

MIME

- MIME adds the following features to email service:
  - Be able to send multiple attachments with a single message;
  - Unlimited message length;
  - Use of character sets other than 7bit -ASCII code;
  - Use of rich text (layouts, fonts, colors, etc)
  - Binary attachments (executables, images, audio or video files, etc.), which may be divided if needed.

MIME Headers

- MIME uses special header directives to describe the format used in a message body, so that the email client can interpret it correctly:
  - `MIME-Version:`
    - This is the version of the MIME standard used in the message. Currently only version 1.0 exists. □
  - `Content-type:`
    - Describes the data's type and subtype. It can include a "charset" parameter, separated by a semi-colon, defining which character set to use.
  - `Content-Transfer-Encoding:`
    - Defines the encoding used in the message body

MIME Data Types

- The primary data types, sometimes called "discrete data types," are:
  - text: readable text data text/rfc822 [RFC822];
    - text/plain [RFC2646];
    - text/html [RFC2854] .
  - image: binary data representing digital images:
    - image/jpeg;
    - image/gif;
    - image/png.
  - audio: digital sound data:
    - audio/basic; audio/wav
  - video: video data:
    - video/mpeg
  - application: Other binary data: application/octet-stream;
    - application/pdf
  - multipart: messages which include multiple attachments, which may even be nested
    - To do so, MIME allows for a standard called boundary. This is an arbitrary string defined as an attribute in the Content-type header: Content-Type: multipart/mixed; boundary="------------020005090303070203010601"

MIME Encoding Formats

- To transfer binary data, MIME offers five encoding formats which can be used in the header transfer-encoding:
  - 7bit: 7-bit text format (for messages without accented characters);
  - 8bit: 8-bit text format;
  - quoted-printable: Quoted-Printable format, recommended for messages which use a 7-bit alphabet (such as when there are accent marks);
  - base64: Base 64, recommended for sending binary files as attachments;
  - binary: binary format; not recommended.

MIME Example

```
From: "Roser Peix" <roser@upvnet.upv.es>
To: profes_eui@upv.es
MIME-Version: 1.0
Content-type: Multipart/Mixed; boundary=Message-Boundary-15761
Subject: Documents

--Message-Boundary-15761
Content-type: text/plain; charset=ISO-8859-1
Content-transfer-encoding: Quoted-printable

Attached to this email you will find the document requested.

--Message-Boundary-15761
Content-type: Application/Octet-stream; name=enero.pdf; type=Unknown
Content-transfer-encoding: BASE64

JVBERi0xLjIgDQol4uPP0w0KIA0KOCAwIG9iag0KPDwNCi9MZW5ndGggOSAwIFINCi9Ga
Wx0ZXIgL0ZsYXRlRGVjb2RlIA0KPj4NCnN0cmVhbQ0KSInMl19vpDgWxT9BfQdLq5F6pY
TBfzDm0QWuChkKaEPVtFr9tppdaR9G6n7Zr7/XYBuoogKVriStlqJWI
--Message-Boundary-15761--
```

Chapter 2: outline

DNS: domain name system

people: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., www.yahoo.com - used by humans

Q: how to map between IP address and name, and vice versa ?

Domain Name System:

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

DNS: services, structure

DNS services

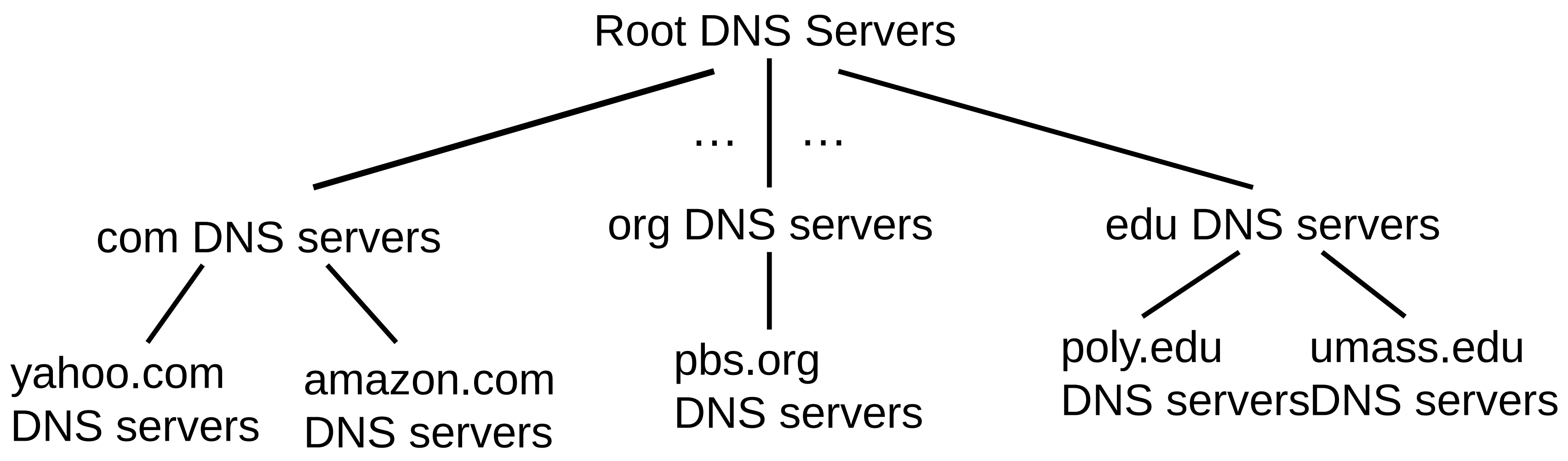
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

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

A: doesn't scale!

DNS: a distributed, hierarchical database

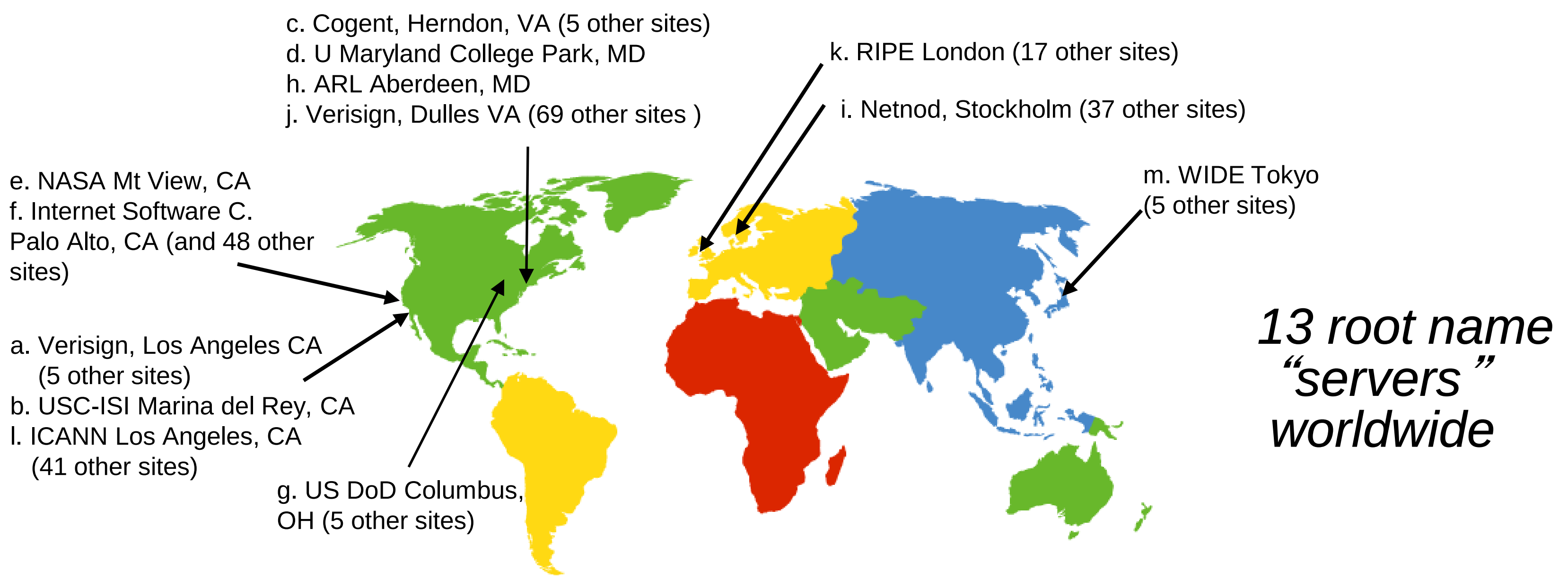


client wants IP for www.amazon.com; 1st approx:

- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

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


c. Cogent, Herndon, VA (5 other sites)
d. U Maryland College Park, MD
h. ARL Aberdeen, MD
j. Verisign, Dulles VA (69 other sites)
k. RIPE London (17 other sites)
i. Netnod, Stockholm (37 other sites)
e. NASA Mt View, CA
f. Internet Software C. Palo Alto, CA (and 48 other sites)
m. WIDE Tokyo (5 other sites)
a. Verisign, Los Angeles CA (5 other sites)
b. USC-ISI Marina del Rey, CA
l. ICANN Los Angeles, CA (41 other sites)
g. US DoD Columbus, OH (5 other sites)
13 root name "servers" worldwide

TLD, authoritative servers

top-level domain (TLD) servers:

- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TLD

authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

Local DNS name server

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

DNS Name Resolution

- Clients's name domain query includes:
  - name to resolve
  - name type
- Sends the query to its local DNS name server
  - Usually, the query from the requesting host to the local DNS server is recursive, and the remaining queries are iterative
- DNS application can use UDP or TCP
  - UDP most used (less overload)
  - Port 53

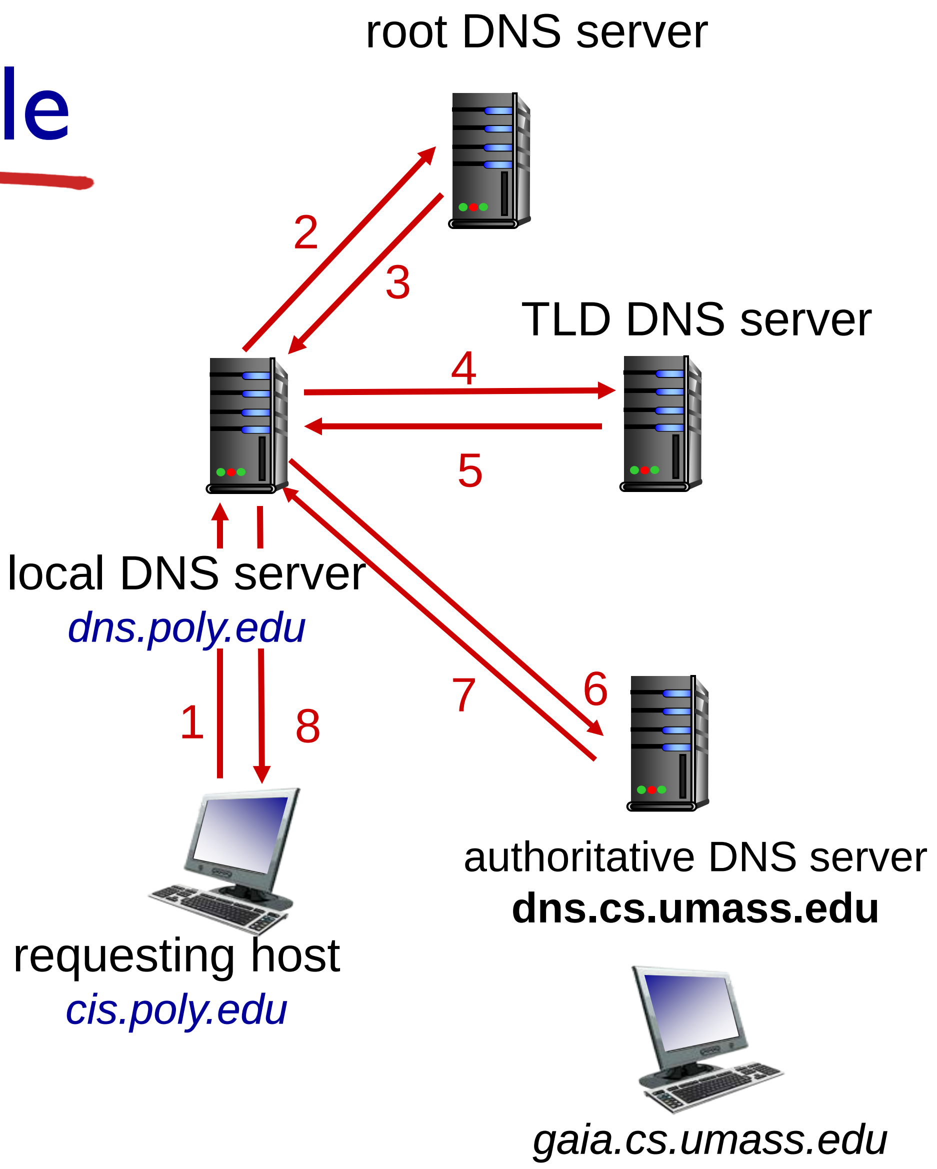
- If the local DNS server can not resolve the query:
  - It forwards the query received from the client to a root DNS server
  - The root DNS server takes note of the last pat of the name domain suffix and returns to the local DNS server a list of IP addresses for TLD servers responsible for this domain.
  - The local DNS server then resends the query message to one of these TLD servers.
  - The TLD server takes note of the previous suffix and responds with the IP address of the authoritative DNS server that domain.
  - Finally, the local DNS server resends the query message directly to the authoritative DNS server, which responds with the IP address of the requested host by the client.

DNS name resolution example

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu

iterated query:

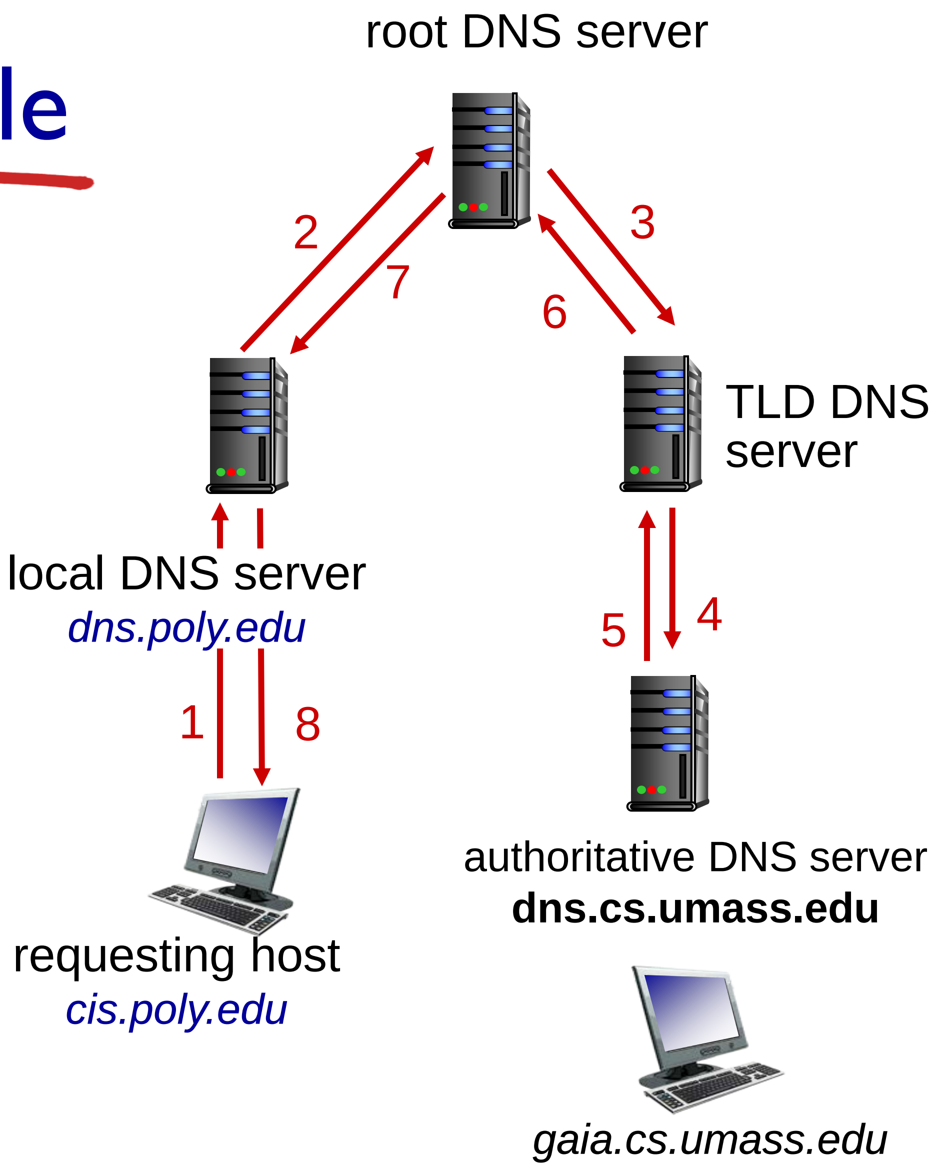
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?
- for one hostname, eight DNS messages were sent: four query messages and four reply messages!



DNS: caching, updating records

- Let's see how DNS caching reduces this query traffic
- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS caching

- When the server can not resolve a name, it looks in its cache
  - If the server resolves it from its cache, it sends the response to the client saying that is obtained from the cache (nonauthoritative) and indicates the name and IP address of the server that provided it
  - If it is not in its cache, it forward the query to another server

DNS records

DNS: distributed db storing resource records (RR)

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

type=A

- **`name`** is hostname
- **`value`** is IPv4 address

type=AAAA

- **`name`** is hostname
- **`value`** is IPv6 address

type=NS

- **`name`** is domain (e.g., foo.com)
- **`value`** is hostname of authoritative name server for this domain

type=CNAME

- **`name`** is alias name for some "canonical" (the real) name
- **`www.ibm.com`** is really **`servereast.backup2.ibm.com`**
- **`value`** is canonical name

type=MX

- **`value`** is name of mailserver associated with **`name`**

type=PTR

- **`nom`** is IP address
- **`value`** is the domain name mapped to that IP

Mail Domain Name

- To know which machine serves a mail domain, DNS server is used

```
nslookup
> set type=MX
> disca.upv.es

disca.upv.es    MX preference = 10, mail exchanger = smtp1.cc.upv.es
disca.upv.es    MX preference = 15, mail exchanger = smtp2.cc.upv.es
disca.upv.es    MX preference = 25, mail exchanger = mail.rediris.es
smtp1.cc.upv.es internet address = 158.42.250.32
smtp2.cc.upv.es internet address = 158.42.250.31
```

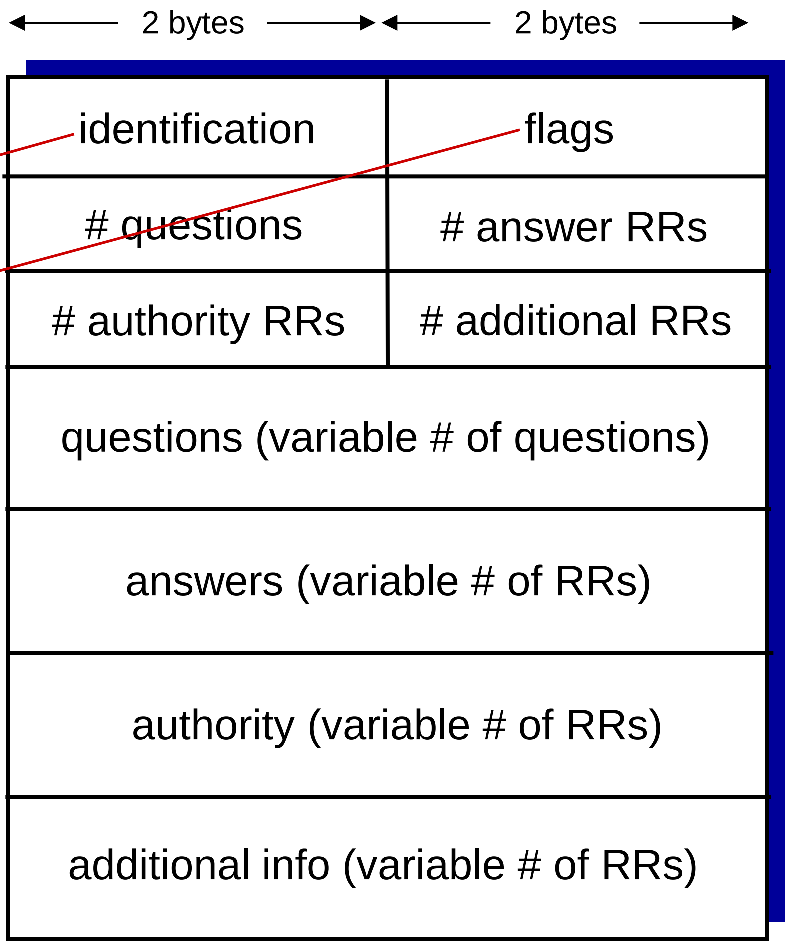
Lower preference = Highest priority

DNS protocol, messages

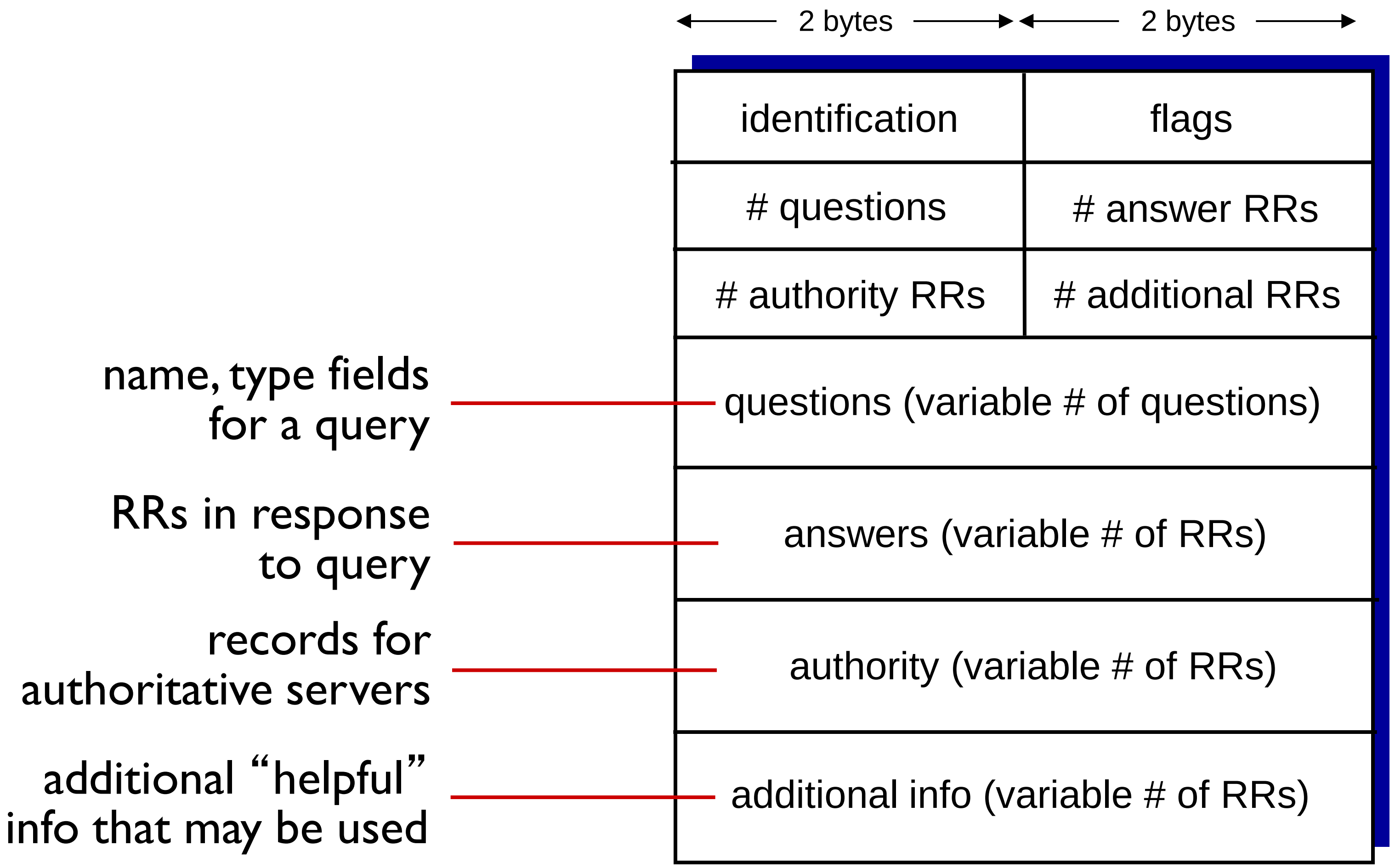
- *query* and *reply* messages, both with same *message format*

msg header

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


2 bytes
2 bytes
identification
flags
# questions
# answer RRs
# authority RRs
# additional RRs
questions (variable # of questions)
answers (variable # of RRs)
authority (variable # of RRs)
additional info (variable # of RRs)

DNS protocol, messages


2 bytes
2 bytes
identification
flags
# questions
# answer RRs
# authority RRs
# additional RRs
questions (variable # of questions)
answers (variable # of RRs)
authority (variable # of RRs)
additional info (variable # of RRs)
name, type fields for a query
RRs in response to query
records for authoritative servers
additional "helpful" info that may be used

Inserting records into DNS

- example: new startup "Network Utopia"
- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts two RRs into .com TLD server:
    `(networkutopia.com, dns1.networkutopia.com, NS)`
    `(dns1.networkutopia.com, 212.212.212.1, A)`
- create authoritative server type A record for www.networkuptopia.com; type MX record for networkutopia.com

Chapter 2: summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- specific protocols:
  - HTTP
  - SMTP, POP, IMAP
  - DNS

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

important themes:

- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"