Lecture 3

Application Layer

Chapter 2: Application layer

2.1 Principles of network applications

2.2 Web and HTTP

2.3 Electronic Mail
   – SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 Video streaming and content distribution networks

2.7 Socket programming with UDP and TCP

Some network apps

- E-mail

- Web

- Instant messaging

- Remote login

- P2P file sharing

- Multi-user network games

- Streaming stored video
  (YouTube, Hulu, Netflix…)

- VoIP(e.g. Skype)

- Real-time video conference

- Social Networking…

Creating a network app

Write programs that
- run on different end systems and
- communicate over a network.
- e.g., Web: Web server software communicates with browser software

little software written for devices in network core
- network core devices do not run user application code
- application on end systems allows for rapid app development, propagation
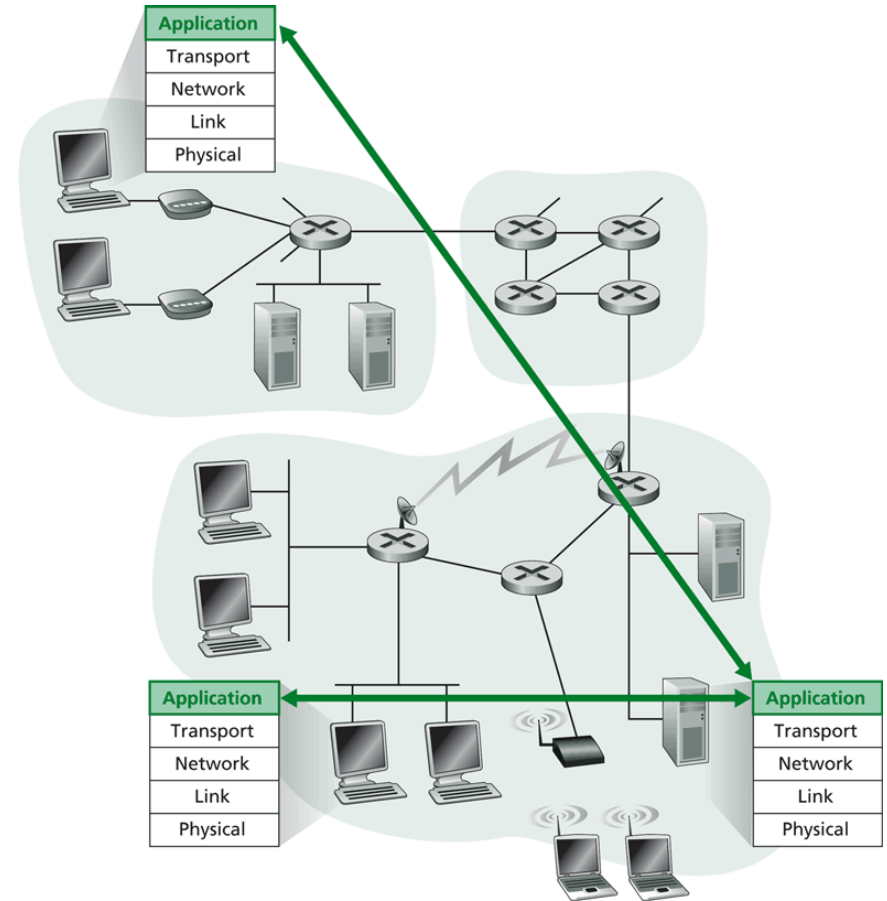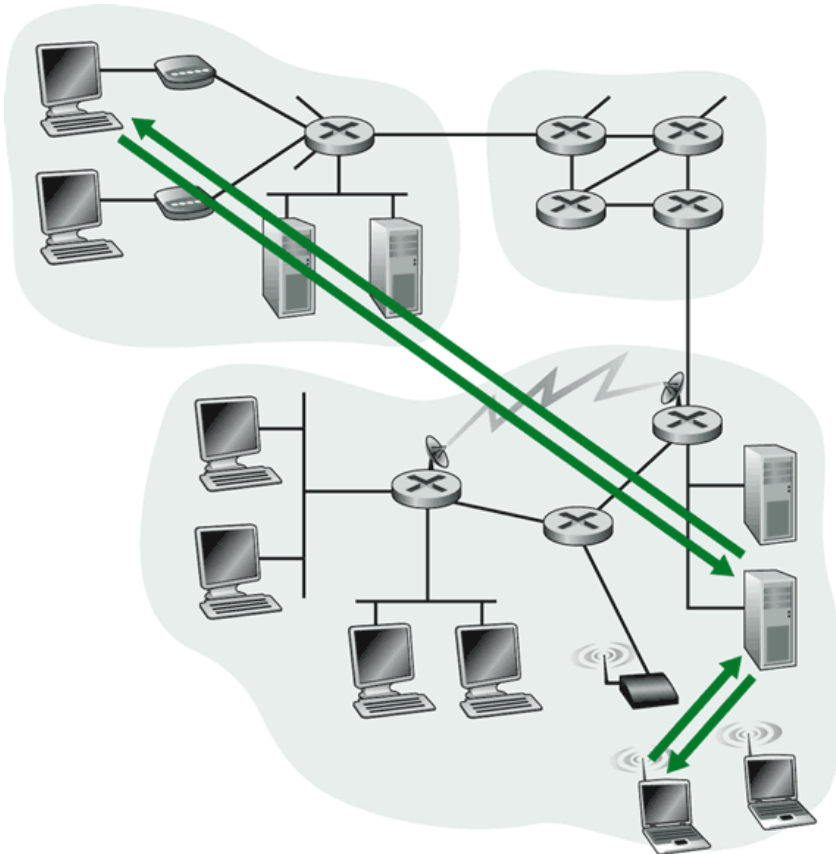
Client-server architecture

server:
- – always-on host
- – permanent IP address
- – server farms for scaling

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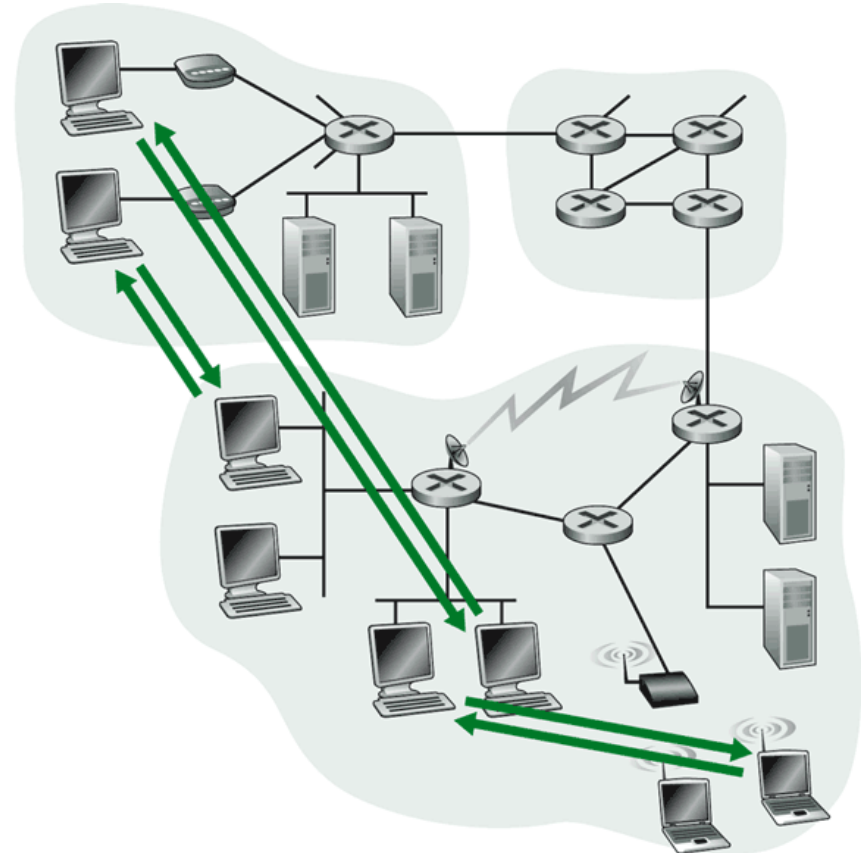- – do not communicate directly with each other

e.g:

P2P architecture

- no always-on server

- arbitrary end systems directly communicate

- peers are intermittently connected and change IP addresses

- e.g:

Highly scalable

But difficult to manage

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

Client process: process that initiates communication

Server process: process that waits to be contacted

- Note: 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 which brings message to socket at receiving process

- API: (1) choice of transport protocol; (2) ability to fix a few parameters

8
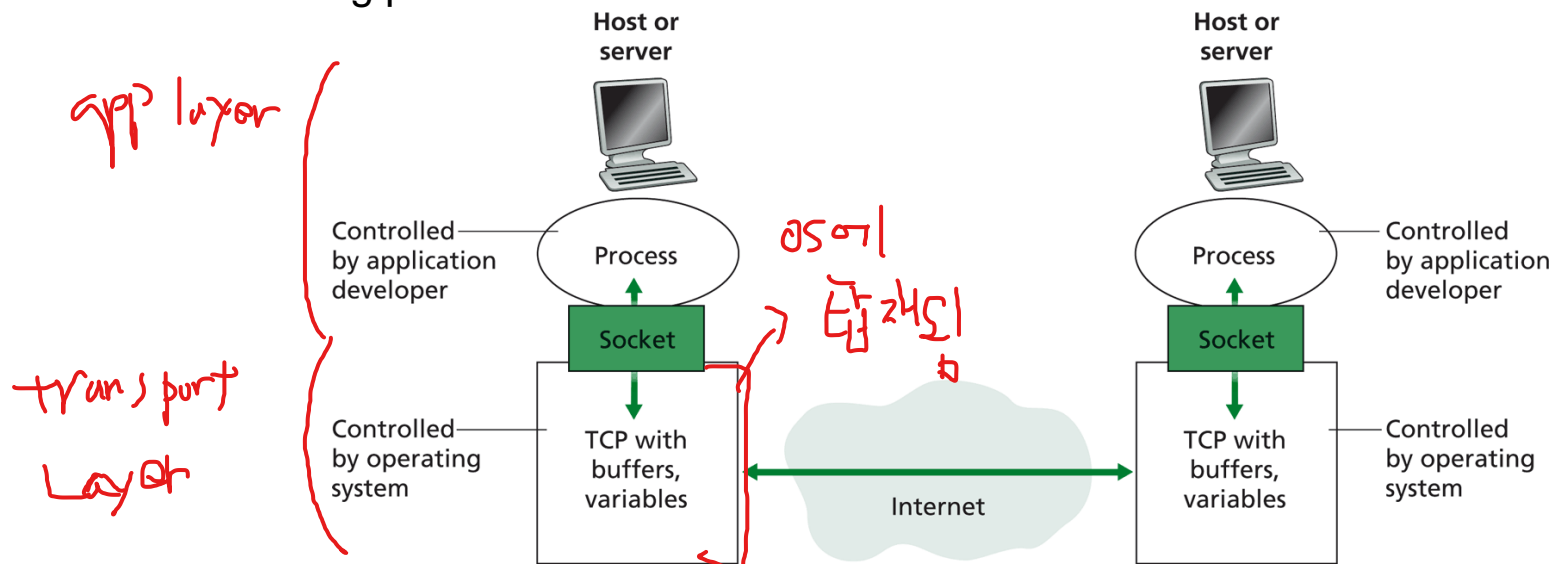
Addressing processes

- For a process to receive messages, it must have an identifier

- A host has a unique32-bit IP address

- Q: does the IP address of the host on which the process runs suffice for identifying the process?

- Identifier includes both the IP address and port numbers associated with the process on the host.

- Example port numbers:
  - HTTP server: 80
  - Mail server: 25

- To send HTTP messages to www.kw.ac.kr web server:
  - IP address: 223.194.1.80
  - Port number: 80

- More on this later

App-layer protocol defines

- Types of messages exchanged: e.g., request & response messages

- 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

Public-domain 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., audio) can tolerate some loss

- other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Throughput

- 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: 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
instant 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*: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

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]	
remote terminal access	Telnet [RFC 854]	*TCP*
Web	HTTP [RFC 2616]	
file transfer	FTP [RFC 959]	
streaming multimedia	HTTP(e.g. YouTube), RTP [RFC 1889]	*TCP or UDP*
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	

Securing TCP

TCP & UDP

- no encryption

- cleartext passwds sent into socket traverse Internet in cleartext

SSL

- provides encrypted TCP connection

- data integrity

- end-point authentication

SSL is at app layer

- apps use SSL libraries, that "talk" to TCP

SSL socket API

- cleartext passwords sent into socket traverse Internet encrypted

- see Chapter 8

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

- Example URL:

```
www.someschool.edu/someDept/pic.gif
```
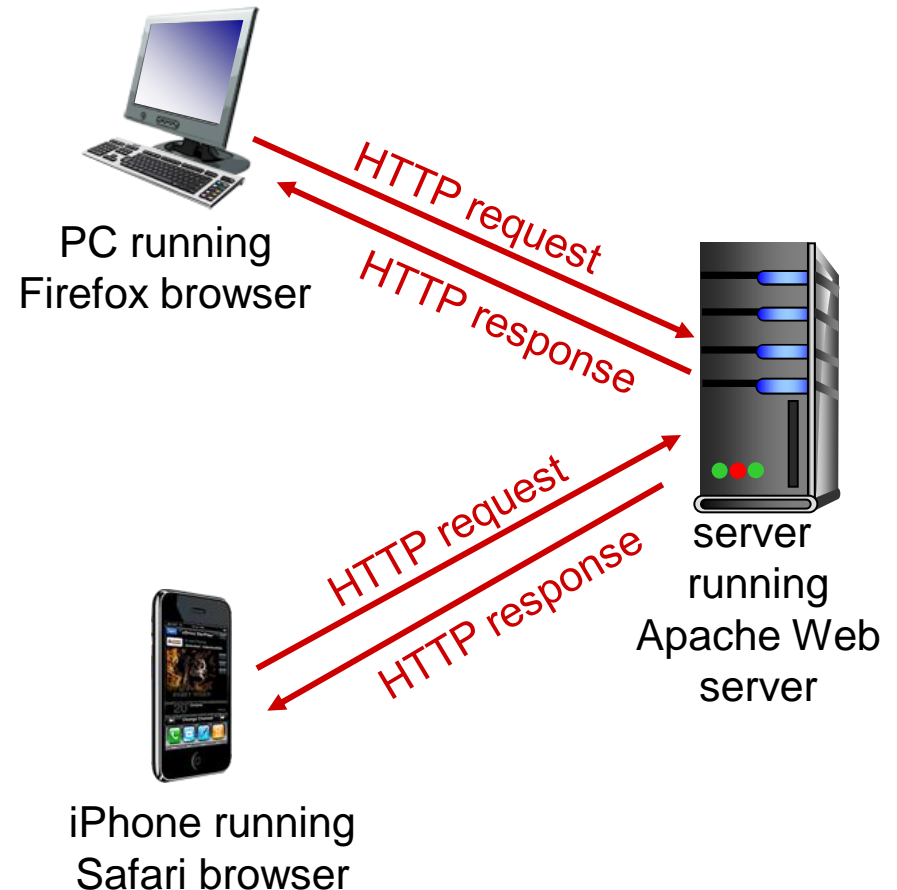
host name   path name

HTTP overview

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

- HTTP 1.0: RFC 1945

- HTTP 1.1: RFC 2068

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

18

HTTP connections

Non-persistent HTTP

- At most one object is sent over a TCP connection.
  - Connection then closed

- Downloading multiple objects requires multiple connections

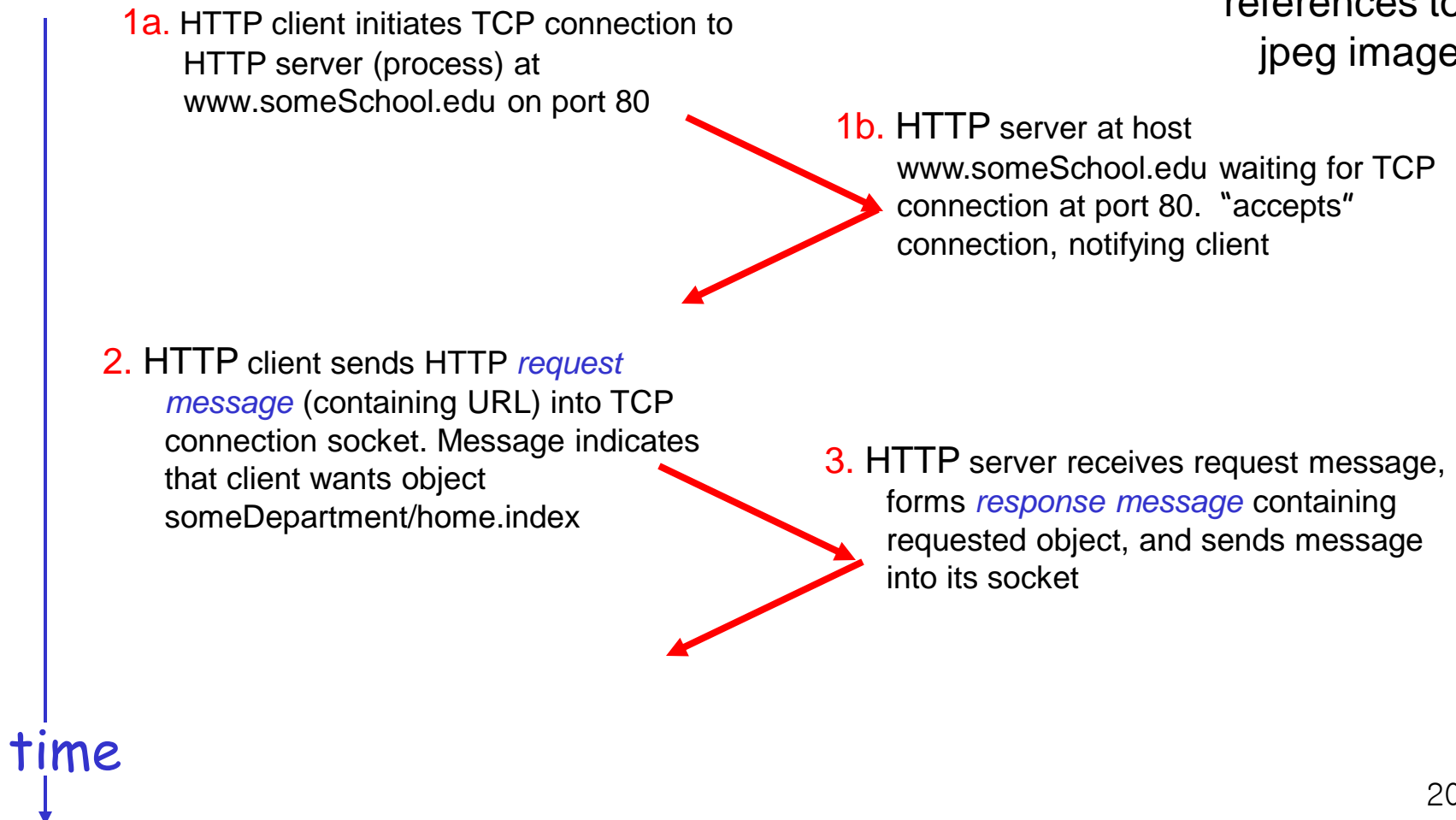- HTTP/1.0 uses nonpersistent HTTP

Persistent HTTP

- Multiple objects can be sent over single TCP connection between client and server.

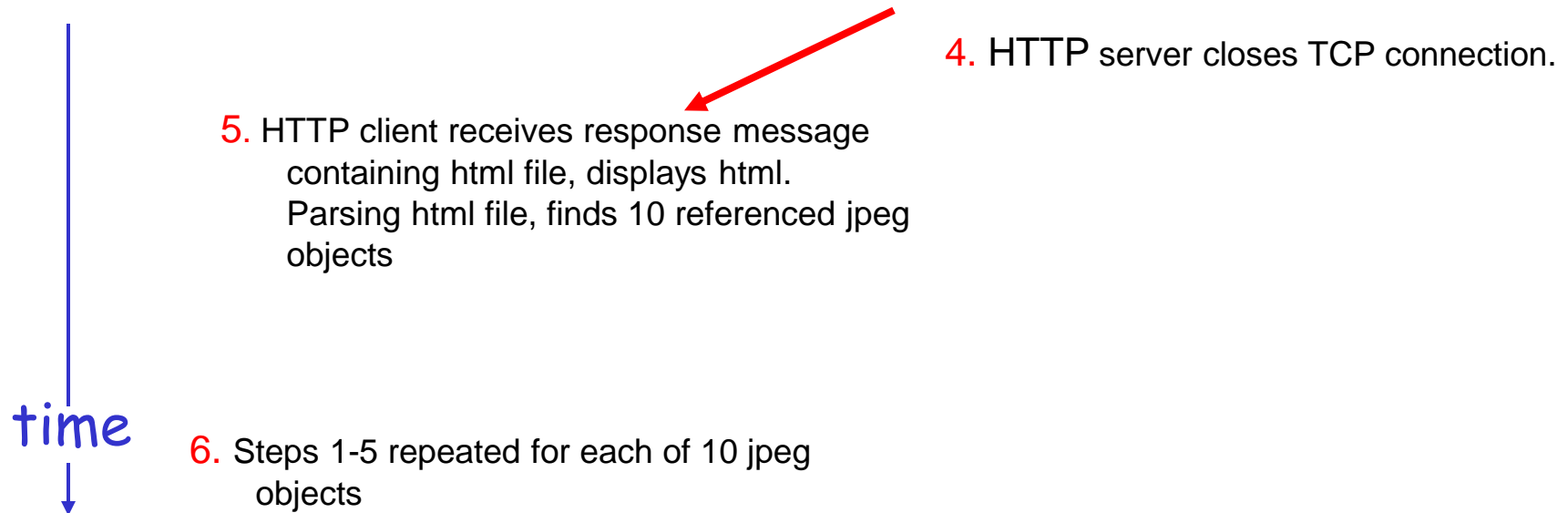- HTTP/1.1 uses persistent connections in default mode

Nonpersistent 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

Nonpersistent 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

time

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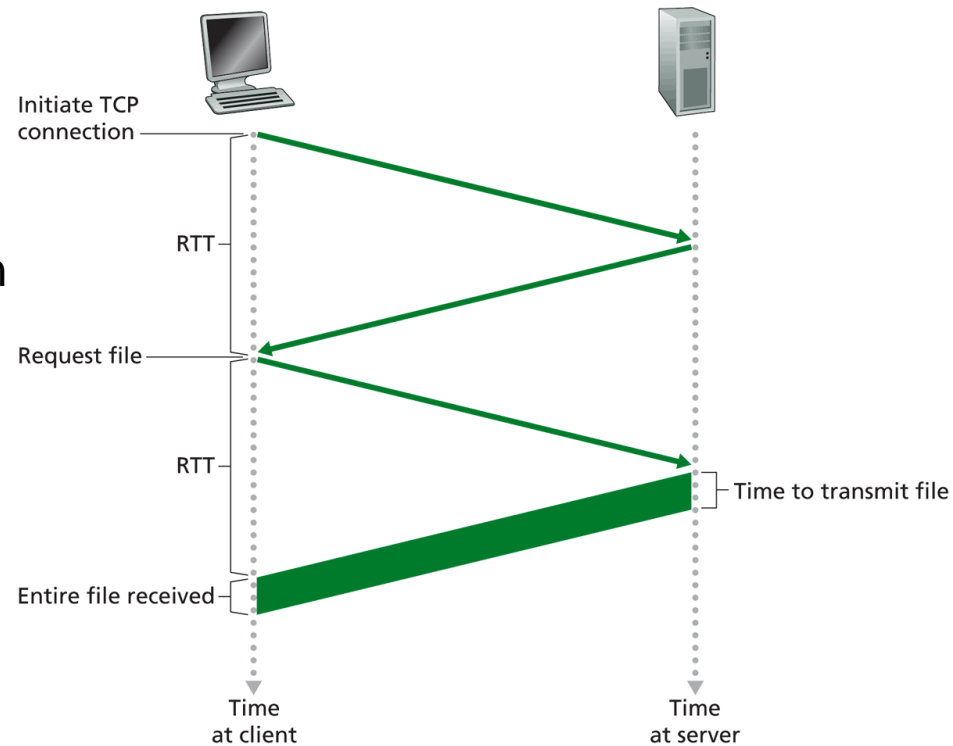
Response time modeling

Definition of RTT: time to send a small packet to travel from client to server and back.
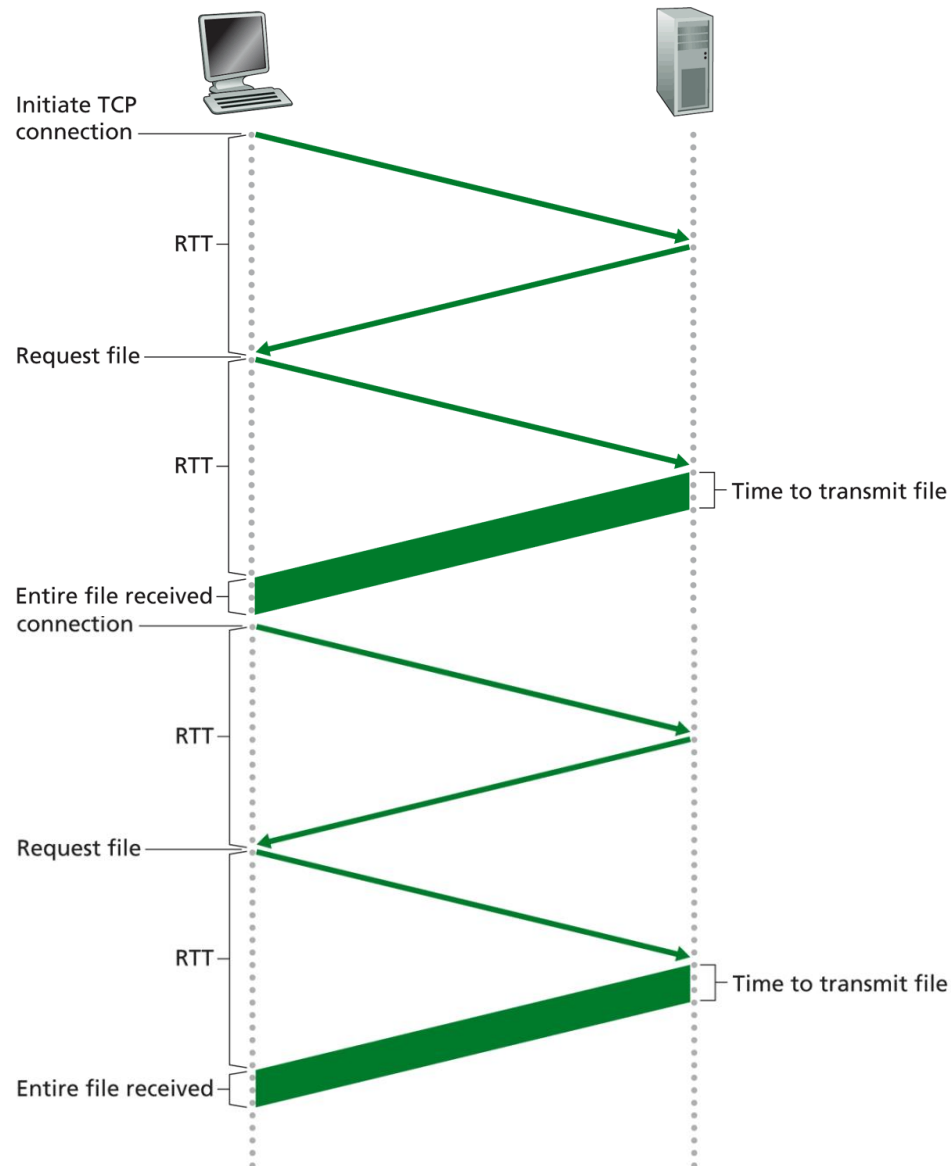
Response time:

- one RTT to initiate TCP connection

- one RTT for HTTP request and first few bytes of HTTP response to return

- file transmission time

total = 2RTT+transmit time

Non-persistent HTTP example: one (embedded) object
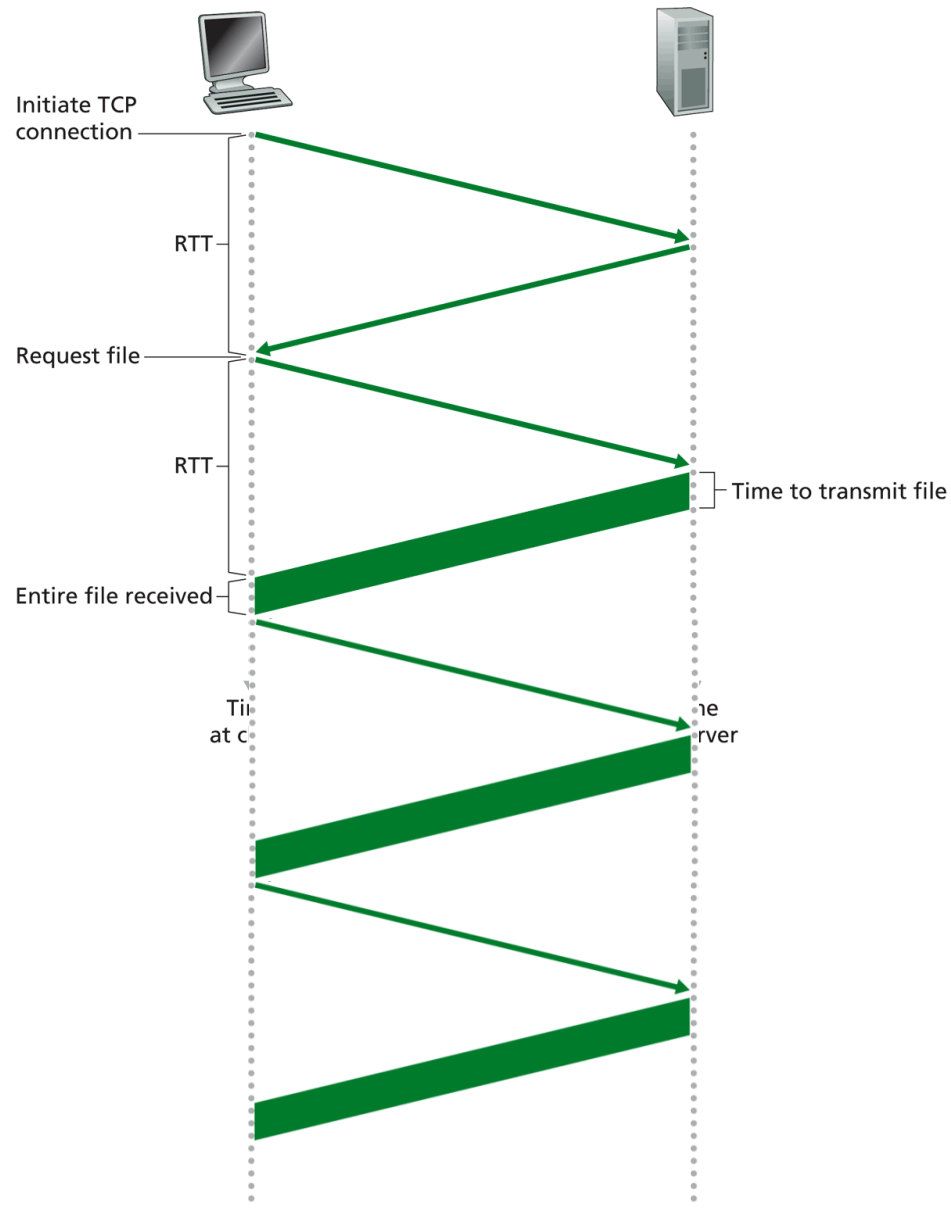
Persistent HTTP

Non-persistent HTTP issues:

- requires 2 RTTs per object

- OS overhead for *each* TCP connection

- browsers often open parallel TCP connections to fetch referenced objects

Persistent HTTP

- server leaves connection open after sending response

- subsequent HTTP messages between same client/server sent over open connection

- client sends requests as soon as it encounters a referenced object

- as little as one RTT for all the referenced objects

Persistent HTTP example: two (embedded) objects

Initiate TCP connection

RTT

Request file

RTT

Time to transmit file

Entire file received

Time at client

Time at server

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)

```
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
```

header
lines

carriage return,
line feed at start
of line indicates
end of header lines

HTTP request message: general format

Request line — method sp URL sp Version cr lf

Header lines — header field name: sp value cr lf
header field name: sp value cr lf

Blank line — cr lf
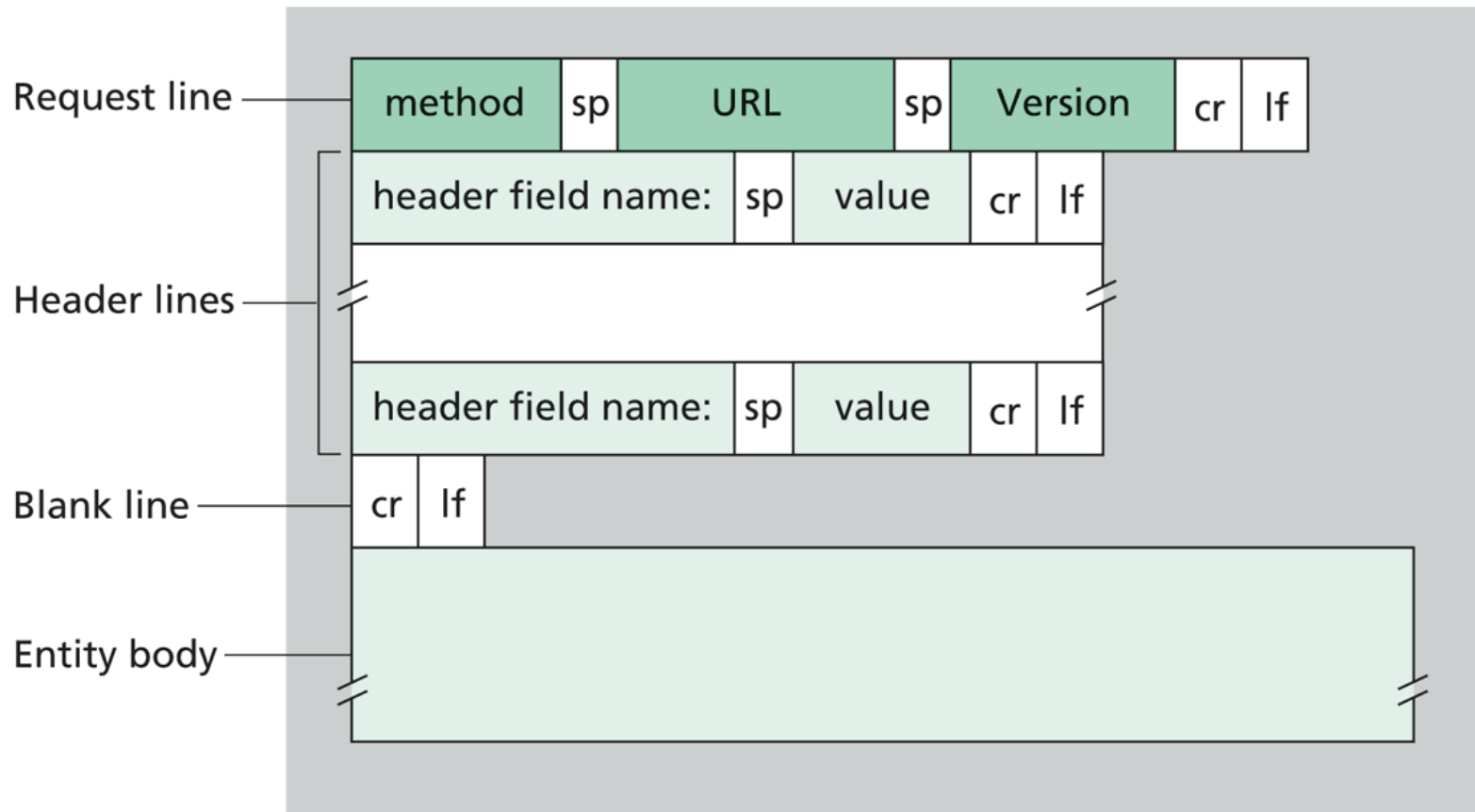
Entity body

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
```

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

HTTP response message

status line
(protocol
status code
status phrase)

```
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 ...
```

header
lines

data, e.g.,
requested
HTML file

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 gaia.cs.umass.edu 80` opens TCP connection to port 80
(default HTTP server port)
at gaia.cs.umass. edu.
anything typed in will be sent
to port 80 at gaia.cs.umass.edu

2. type in a GET HTTP request:

`GET /kurose_ross/interactive/index.php HTTP/1.1`
`Host: gaia.cs.umass.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)

User-server state: cookies
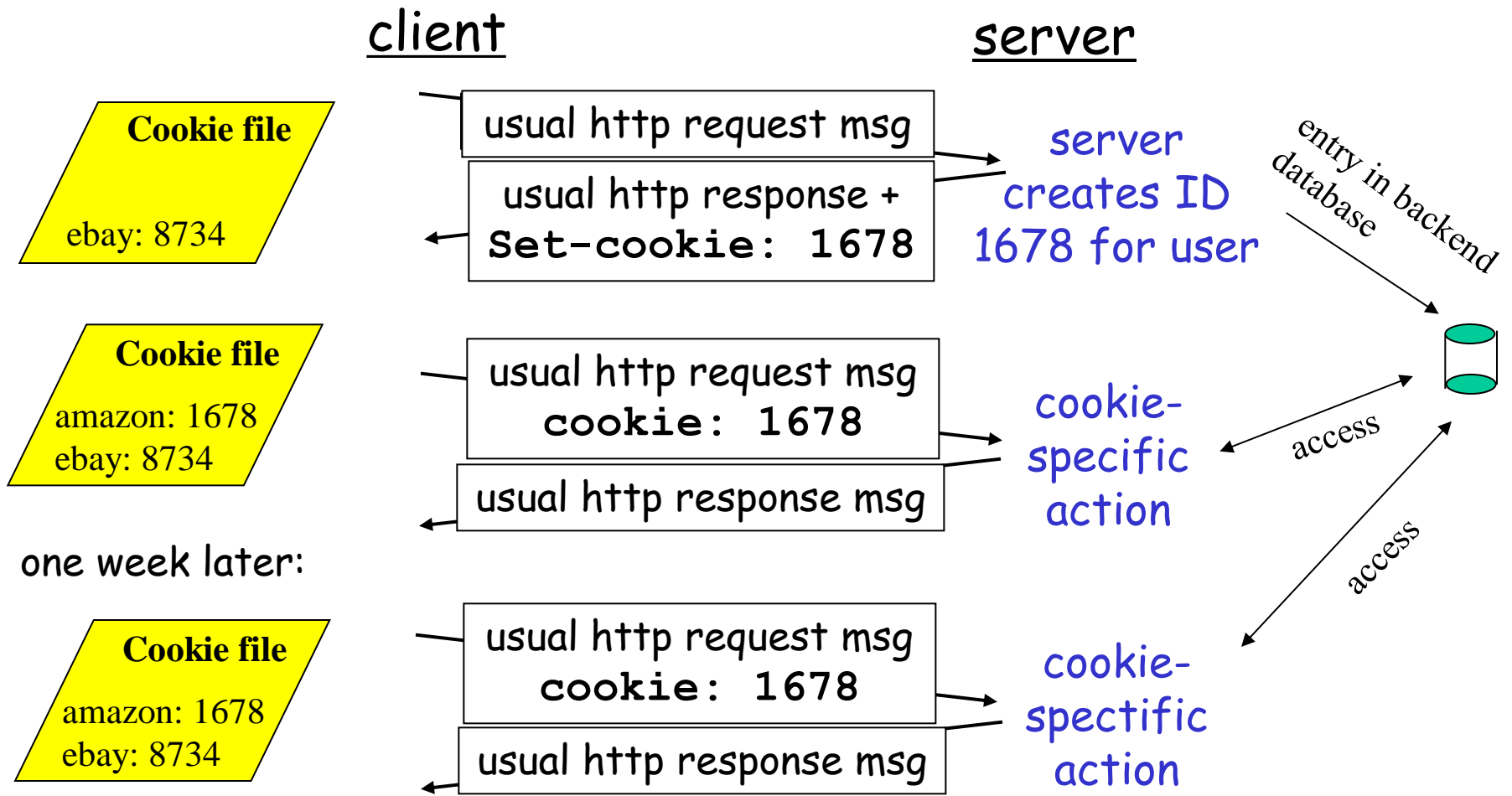
Many major Web sites use cookies

1) cookie header line of HTTP *response* message
2) cookie header line in 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 access Internet always from same PC
– She visits a specific e-commerce site for first time
– When initial HTTP requests arrives at site, site creates a unique ID and creates an entry in backend database for ID

Cookies: keeping "state" (cont.)

client   server

Cookie file

ebay: 8734

usual http request msg
usual http response + **Set-cookie: 1678**

server creates ID 1678 for user

entry in backend database

Cookie file

amazon: 1678
ebay: 8734

usual http request msg **cookie: 1678**
usual http response msg

cookie-specific action

access

one week later:

Cookie file

amazon: 1678
ebay: 8734

usual http request msg **cookie: 1678**
usual http response msg

cookie-spectific action

access

Cookies (continued)

what cookies can be used for:

- authorization

- shopping carts

- recommendations

- user session state (Web e-mail)

how to keep "state":

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

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

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
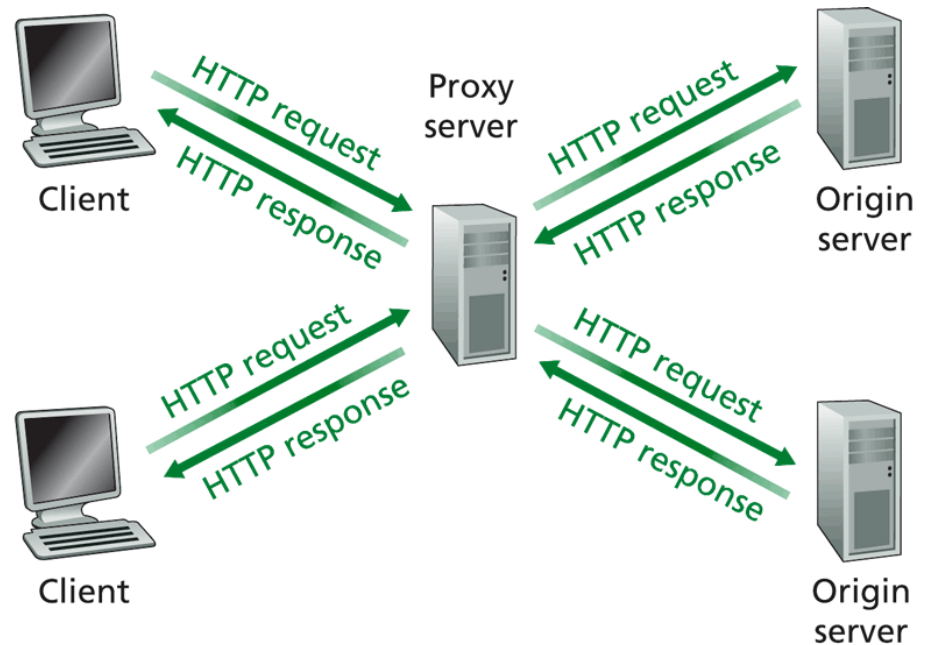
More about Web caching

- 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 (but so does P2P file sharing)
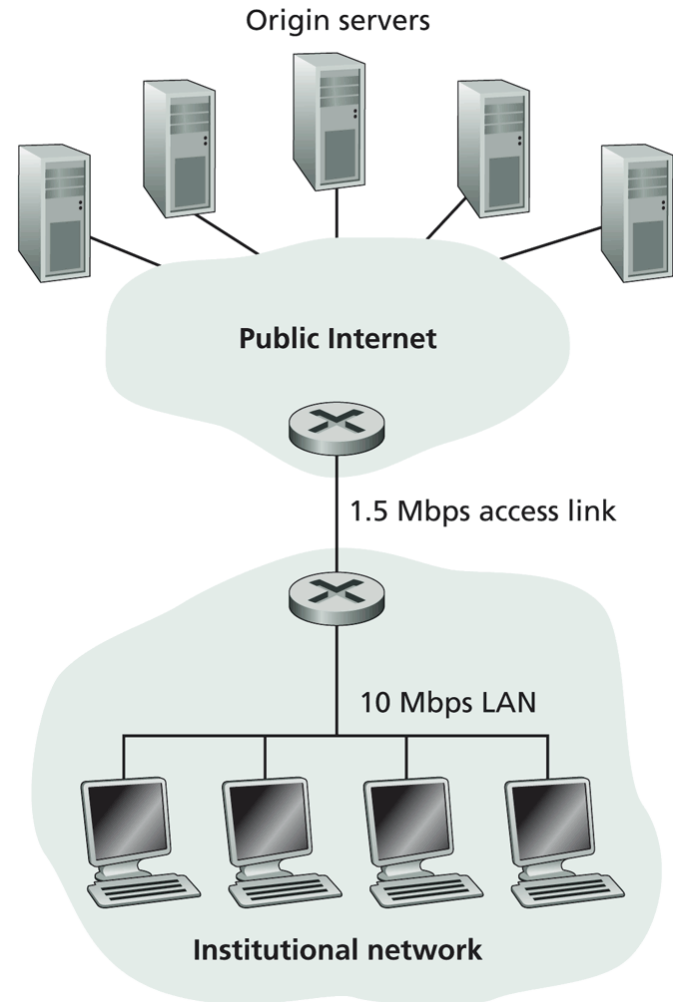
Caching example

Assumptions

- average object size = 100,000 bits

- avg. request rate from institution's browsers to origin servers = 15/sec

- delay from **edge** 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

Origin servers

Public Internet

1.5 Mbps access link

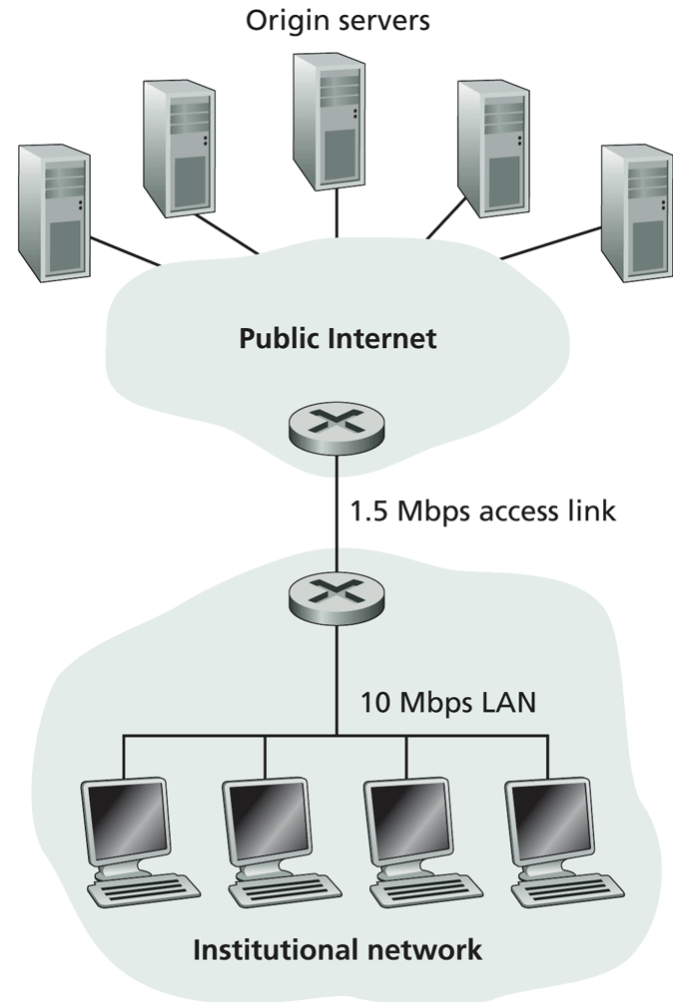10 Mbps LAN

Institutional network

Caching example (cont)

Possible solution

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

Consequences

- utilization on LAN = ?

- utilization on access link = ?

- Total delay = ?

- often a costly upgrade

Origin servers

Public Internet

1.5 Mbps access link

10 Mbps LAN
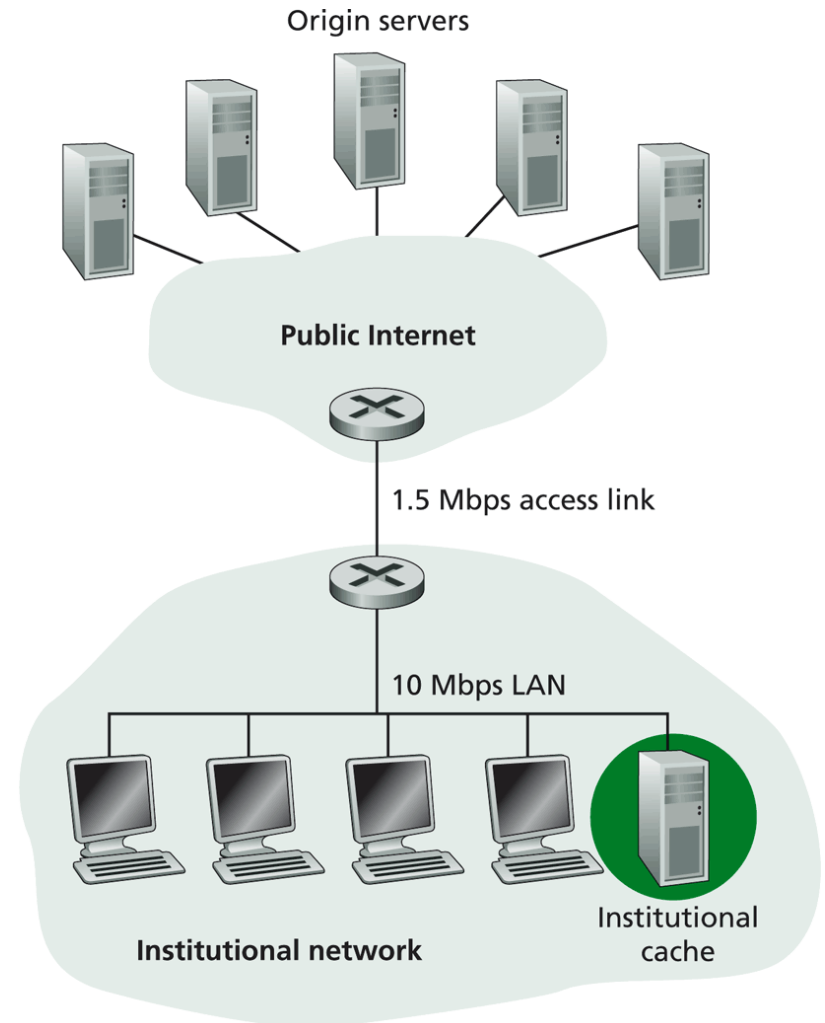
Institutional network

Caching example (cont)
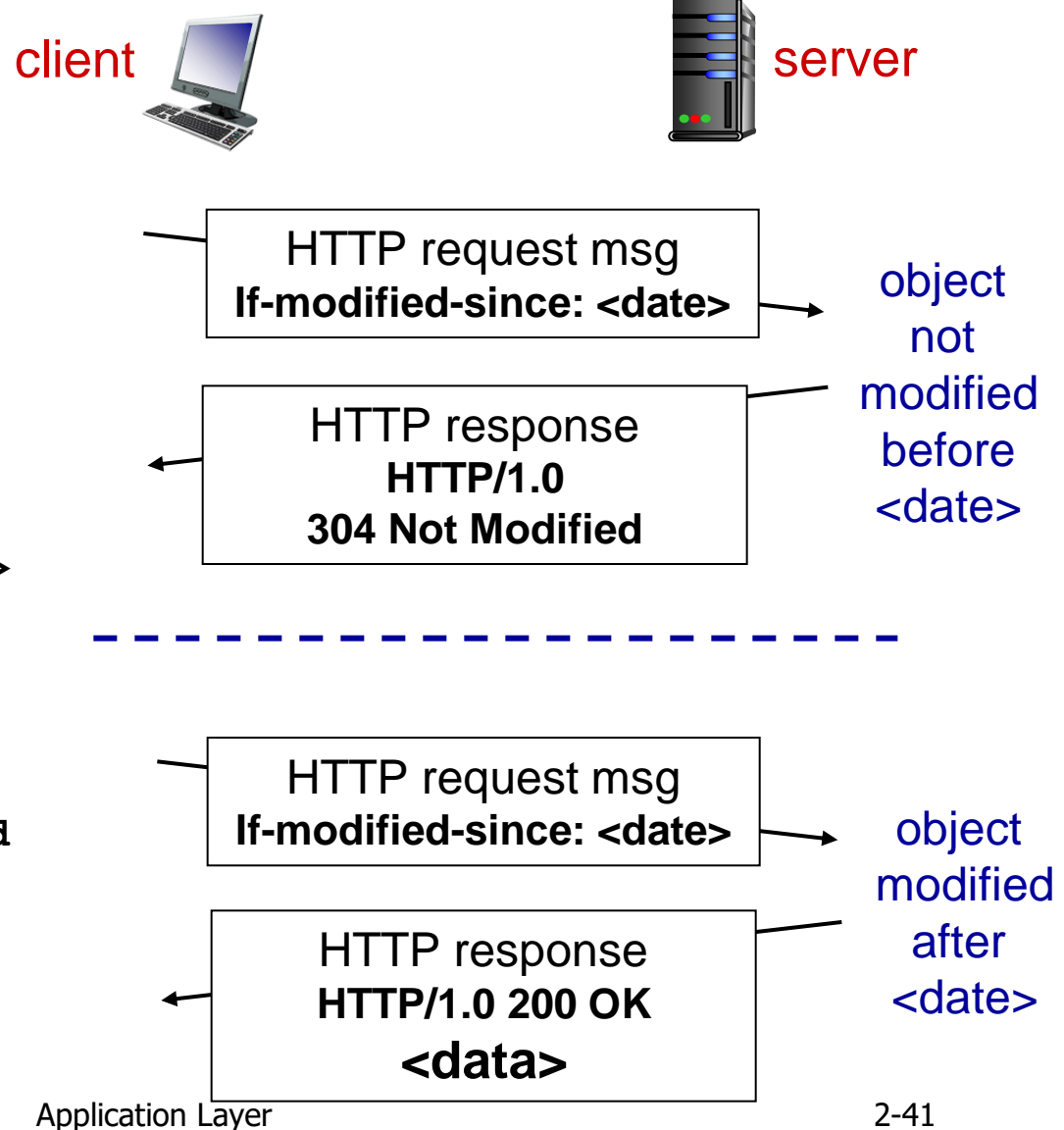
Install cache

- 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 avg delay = Internet delay + access delay + LAN delay = ?

Origin servers

Public Internet

1.5 Mbps access link

10 Mbps LAN

Institutional network

Institutional cache

Conditional GET

client   server

- *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

HTTP request msg
If-modified-since: <date>

object not modified before <date>

HTTP response
**HTTP/1.0
304 Not Modified**

- -

HTTP request msg
If-modified-since: <date>

object modified after <date>

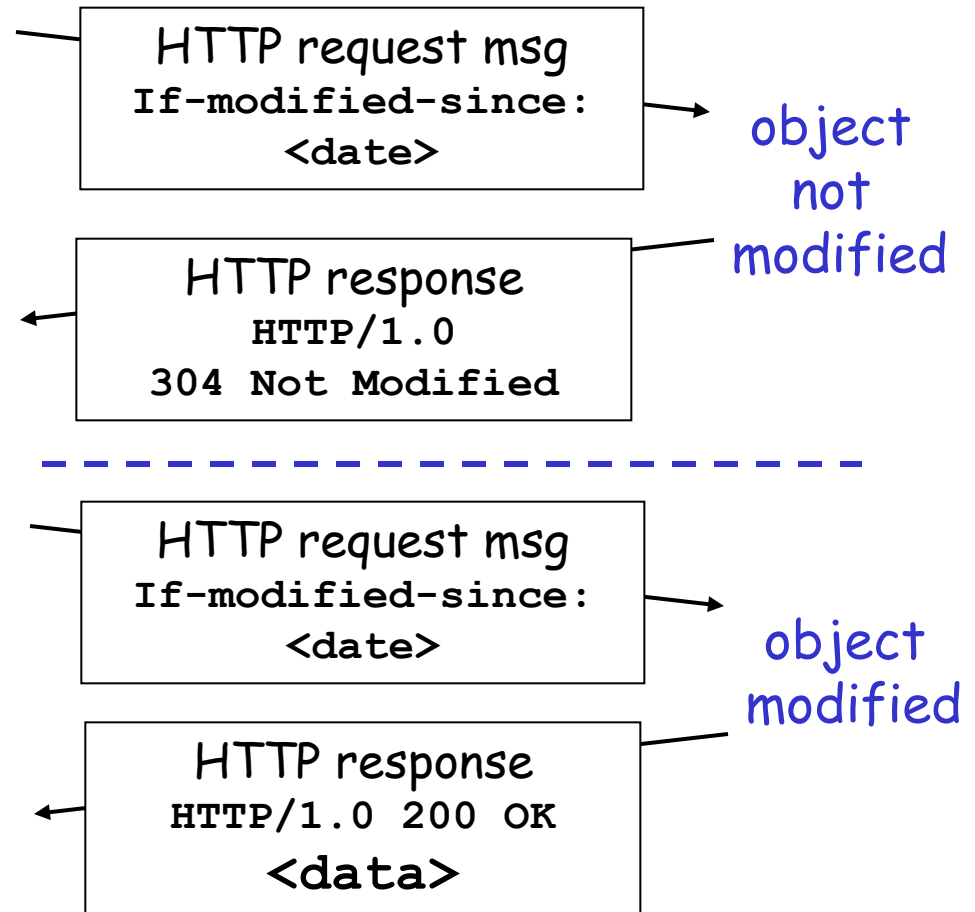HTTP response
**HTTP/1.0 200 OK
<data>**

Application Layer

Conditional GET

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

- 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`

cache

server

HTTP request msg
**If-modified-since:
<date>**

object not modified

HTTP response
**HTTP/1.0
304 Not Modified**

- -

HTTP request msg
**If-modified-since:
<date>**

object modified

HTTP response
HTTP/1.0 200 OK
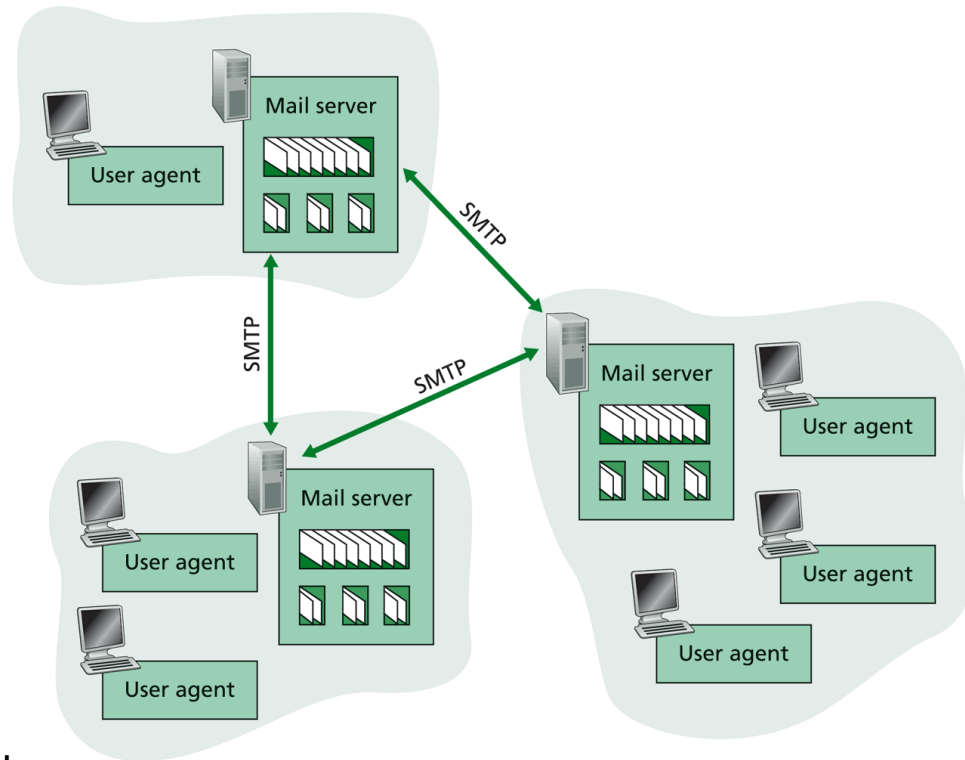**<data>**

Electronic Mail

Three major components:

- user agents

- mail servers

- simple mail transfer protocol: SMTP

<u>User Agent</u>

- a.k.a. "mail reader"

- composing, editing, reading mail messages

- e.g., Outlook, Netscape Messenger, 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
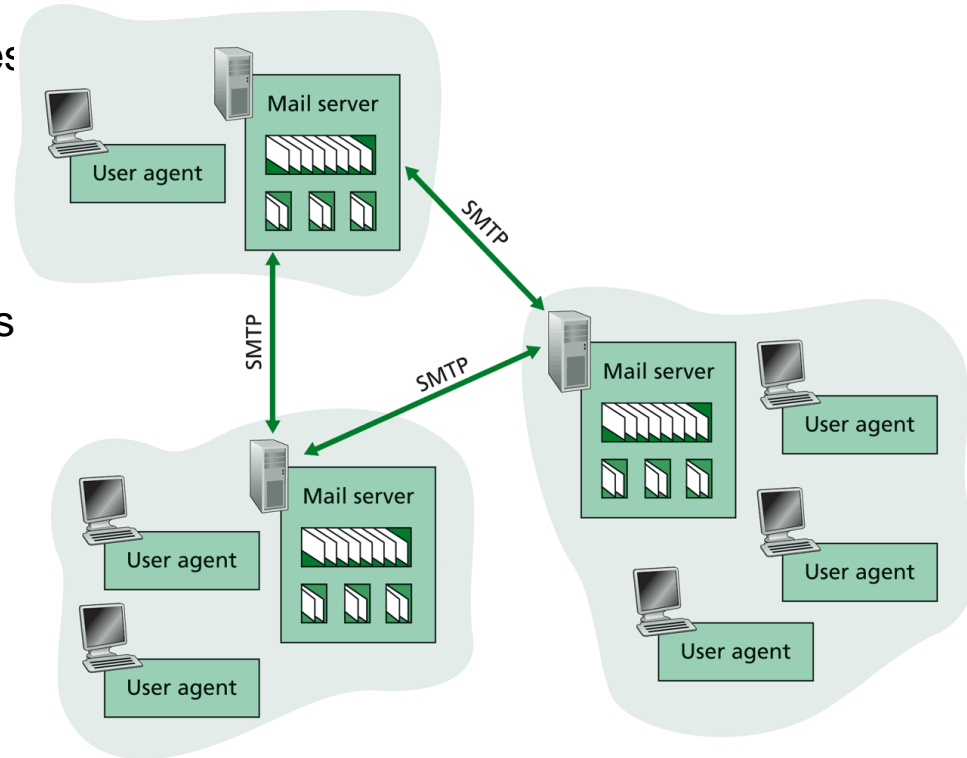
Electronic Mail: SMTP [RFC 2821]

- 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
  - 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
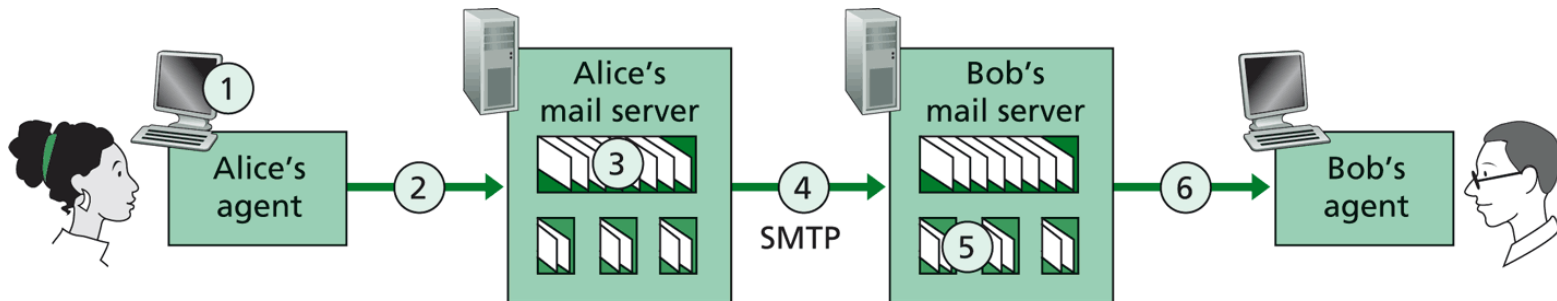
Sample SMTP interaction

```
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 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
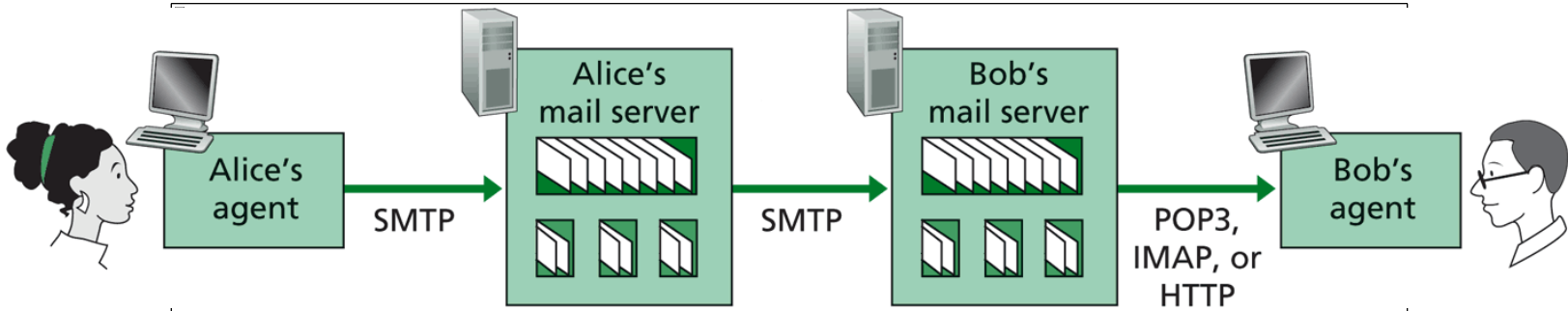
Mail message format

SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

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

    different from SMTP commands!

- body
  - the "message", ASCII characters only

header

body

blank line
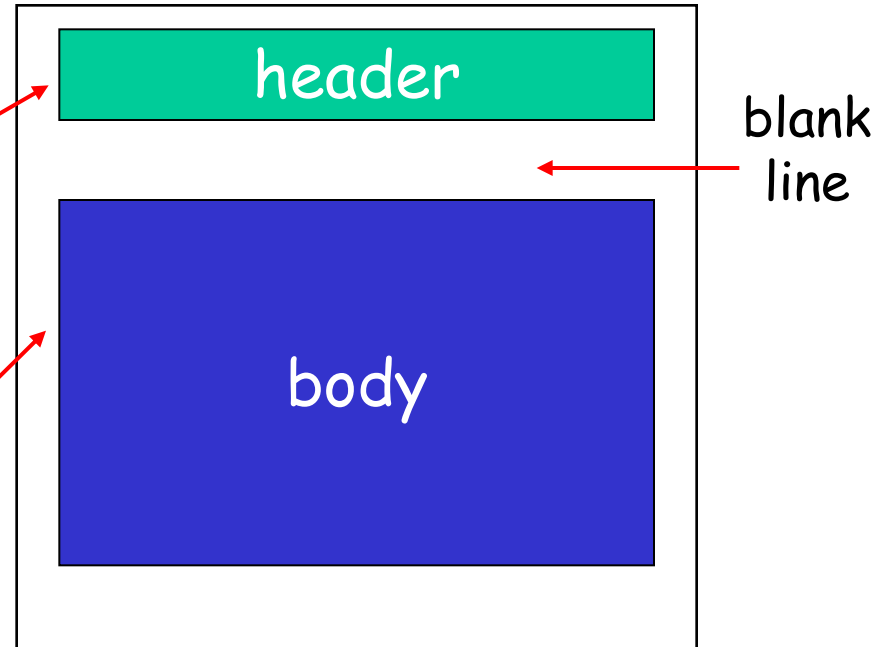
Mail access protocols

- 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**

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 "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

DNS: Domain Name System

People: many identifiers:
- – SSN, name, passport #

Internet hosts, routers:
- – IP address (32 bit) - used for addressing datagrams
- – "name", e.g., ww.yahoo.com - 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"
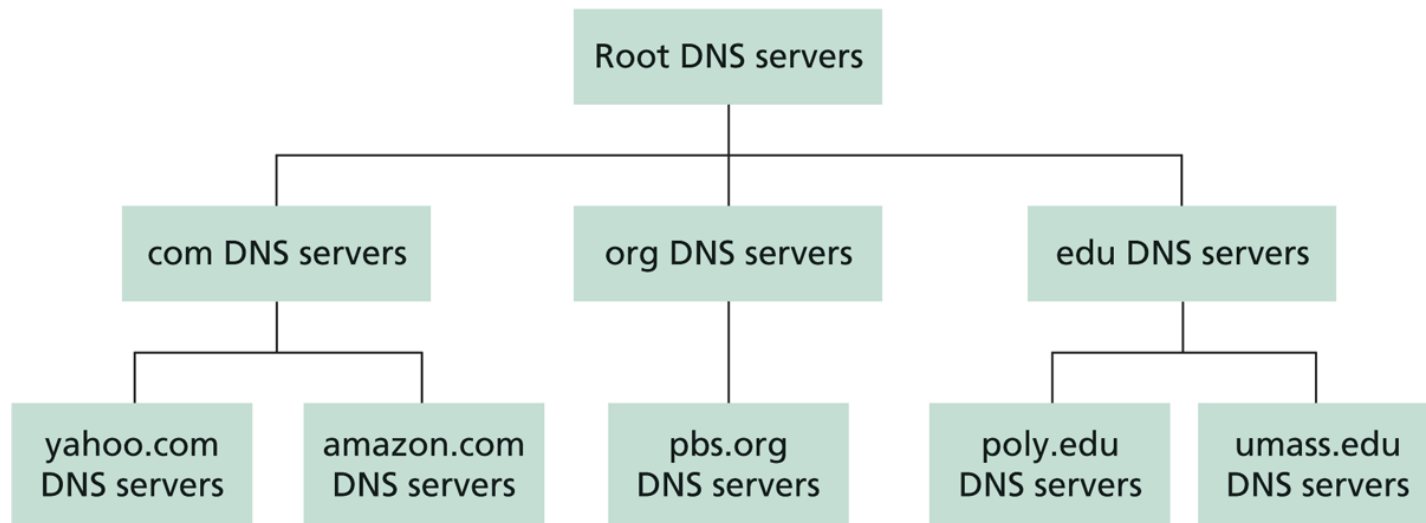
DNS

DNS services

- Hostname to IP address translation

- Host aliasing
  - Canonical and alias names

- Mail server aliasing

- Load distribution
  - Replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?

- single point of failure

- traffic volume

- distant centralized database

- maintenance

doesn't *scale!*

Distributed, Hierarchical Database

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

- Client queries a 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

a. Verisign, Dulles, VA
c. Cogent, Herndon, VA (also Los Angeles)
d. U Maryland College Park, MD
g. US DoD Vienna, VA
h. ARL Aberdeen, MD
j. Verisign, (11 locations)

e. NASA Mt View, CA
f. Internet Software C.
Palo Alto, CA
(and 17 other locations)

i. Autonomica, Stockholm
(plus 3 other locations)
k. RIPE London
(also Amsterdam,
Frankfurt)

b. USC-ISI Marina del Rey, CA
l. ICANN Los Angeles, CA

m. WIDE Tokyo

13 root name servers
worldwide

TLD and Authoritative Servers

- Top-level domain (TLD) servers: responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
    - Network solutions maintains servers for com TLD
    - Educause for edu TLD

- Authoritative DNS servers: organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web and mail).
    - Can be maintained by organization or service provider

Local Name Server

- Does not strictly belong to hierarchy

- Each ISP (residential ISP, company, university) has one.
  - Also called "default name server"

- When a host makes a 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 a proxy, forwards query into hierarchy.

Example

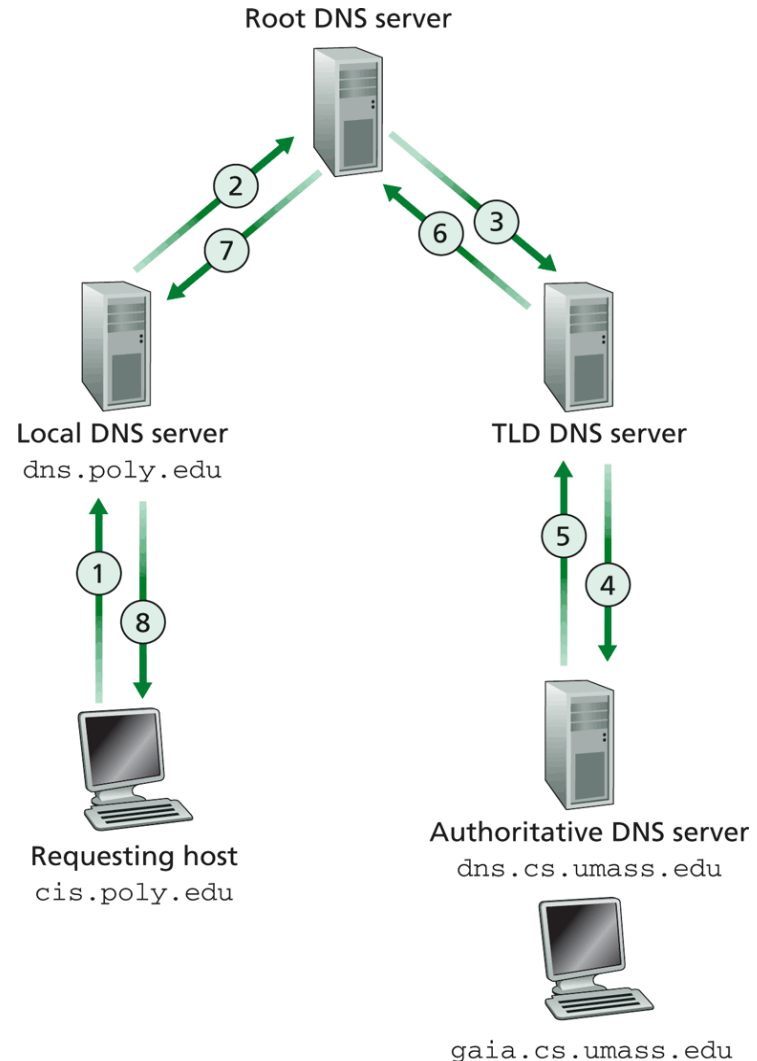- Host at cis.poly.edu wants IP address for gaia.cs.umass.edu
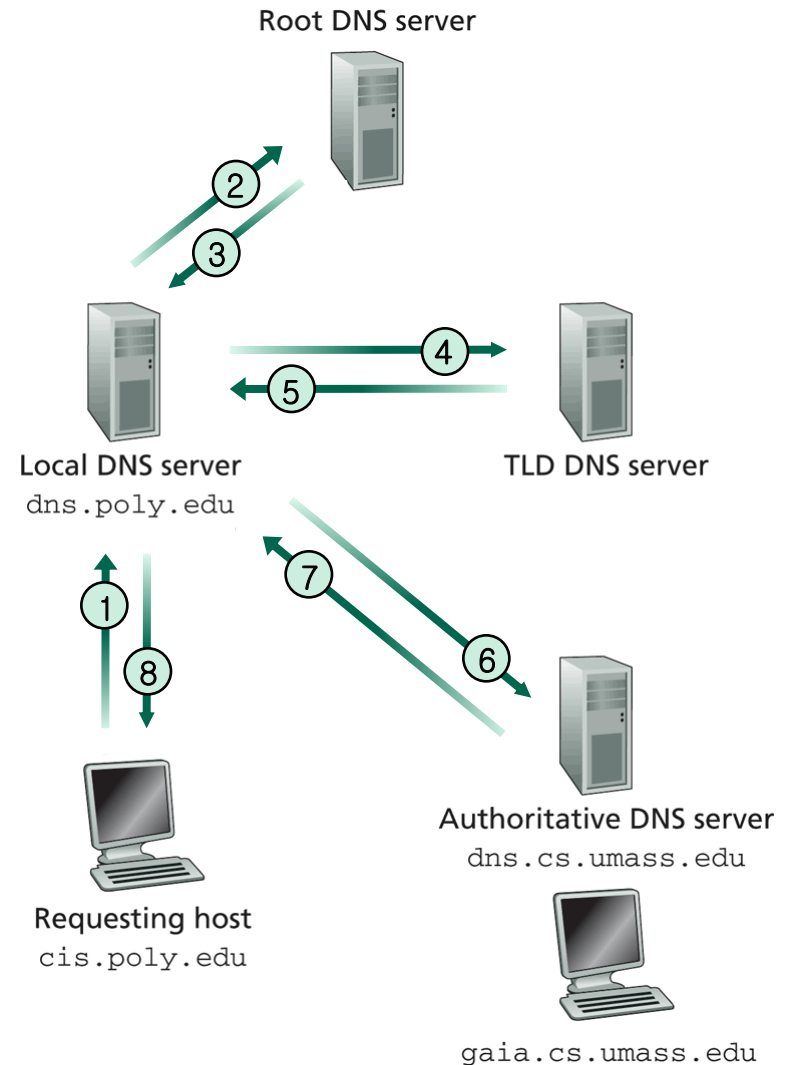
Recursive queries

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
    - 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 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 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
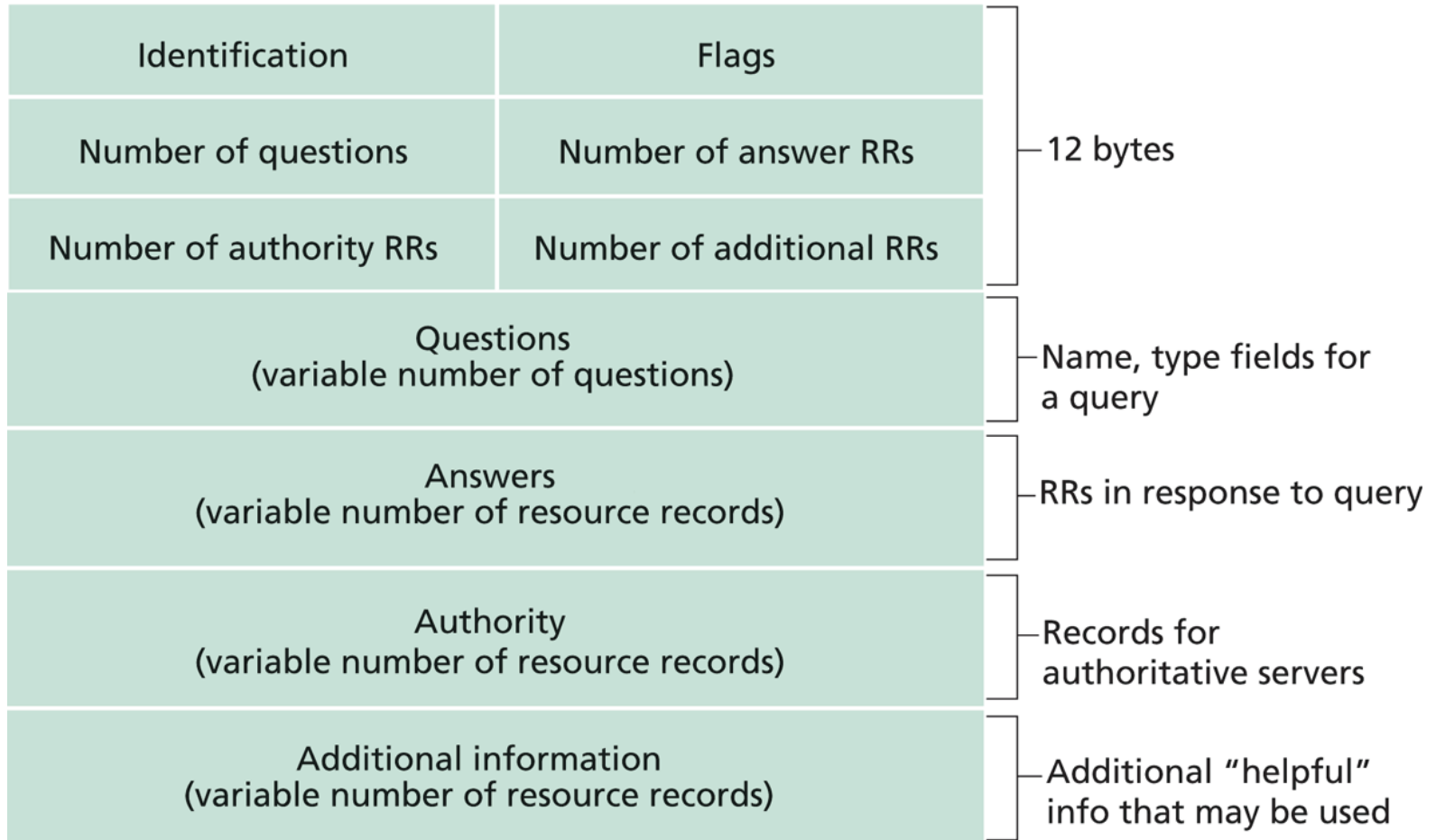 - value is name of mailserver associated with name

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
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)

DNS protocol, messages

Identification	Flags
Number of questions	Number of answer RRs
Number of authority RRs	Number of additional RRs

— 12 bytes

Questions (variable number of questions)

— Name, type fields for a query

Answers (variable number of resource records)

— RRs in response to query

Authority (variable number of resource records)

— Records for authoritative servers

Additional information (variable number of resource records)

— 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