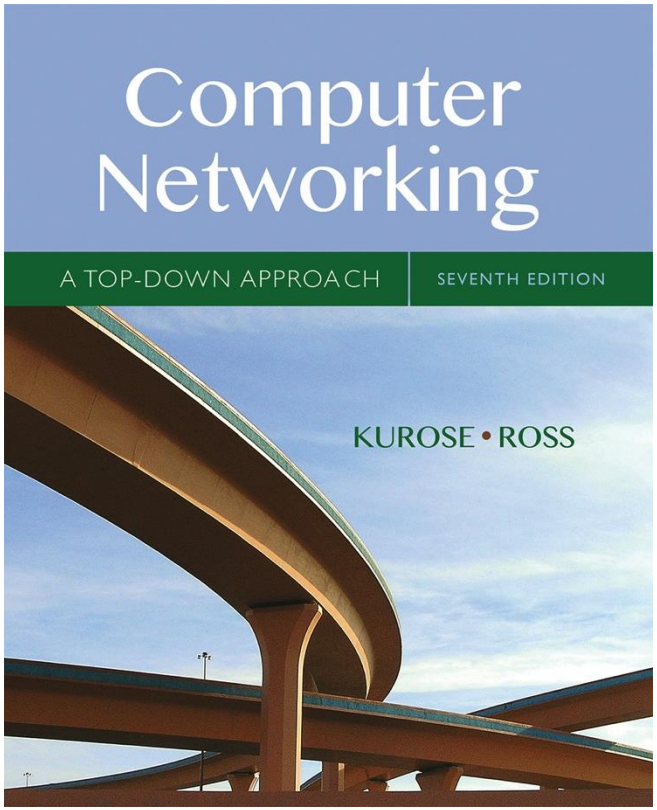
Chapter 2
Application Layer



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Network Applications

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
  - SFTP
  - SMTP / POP3 / IMAP
  - DNS
- creating network applications
  - socket API

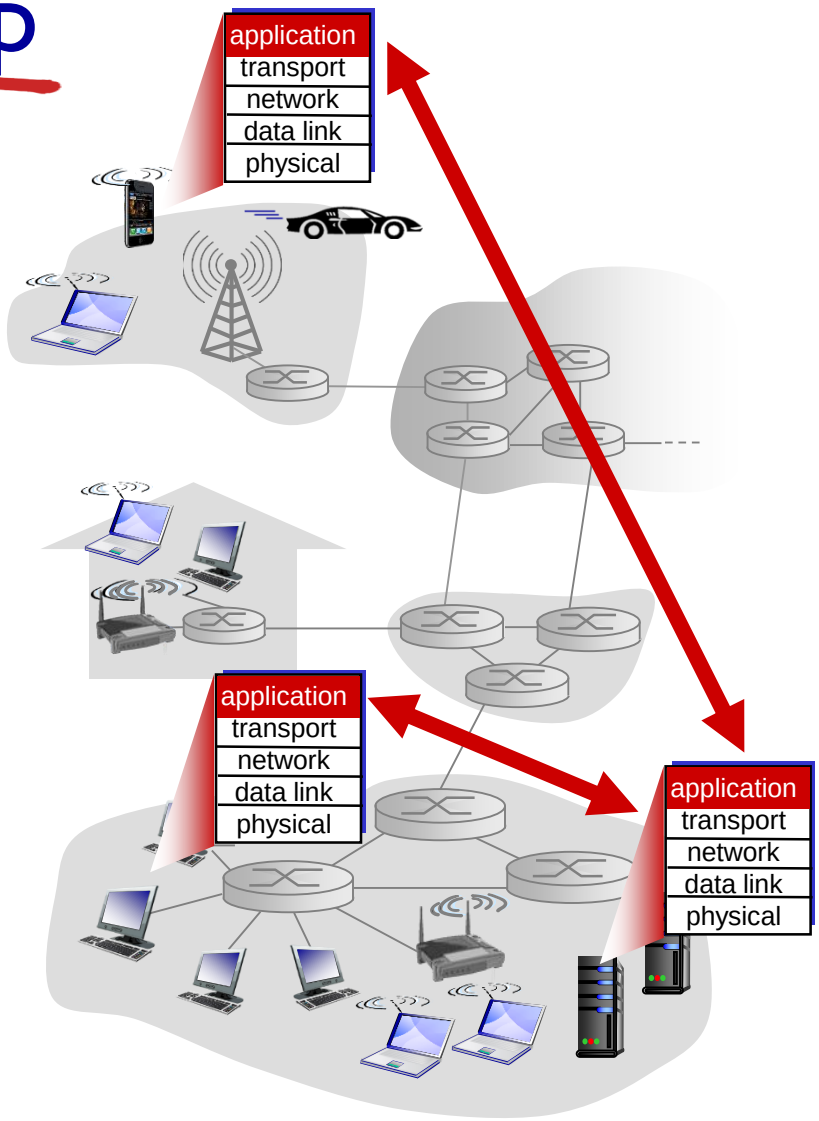
Some network apps

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming 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

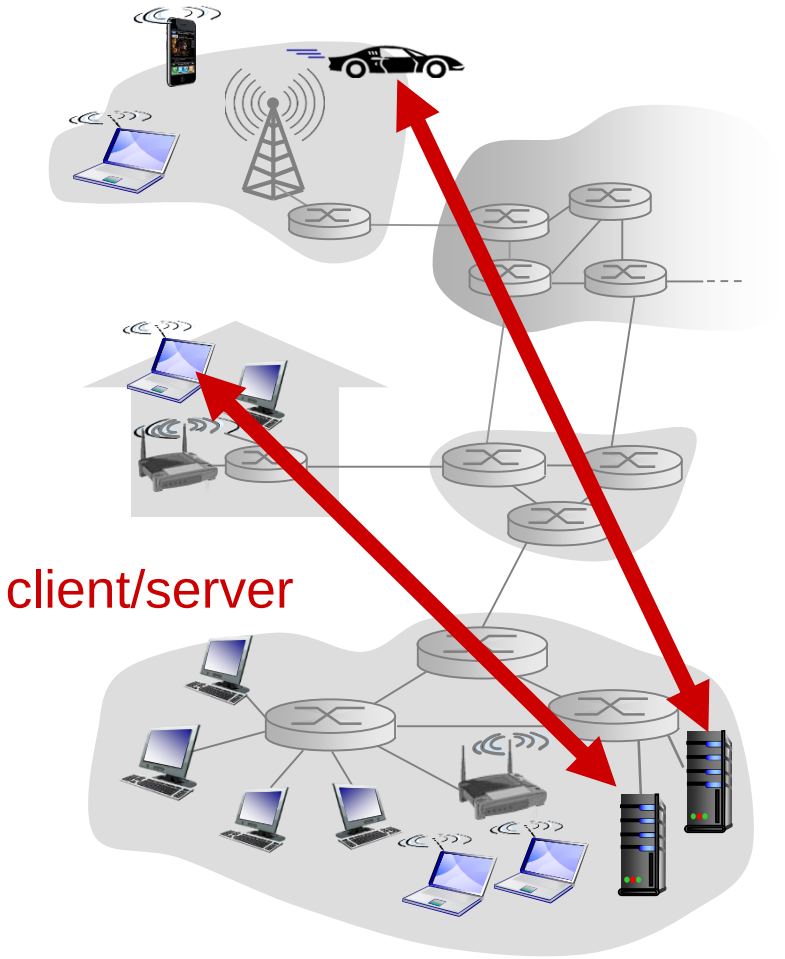
- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)

Client-server architecture



server:

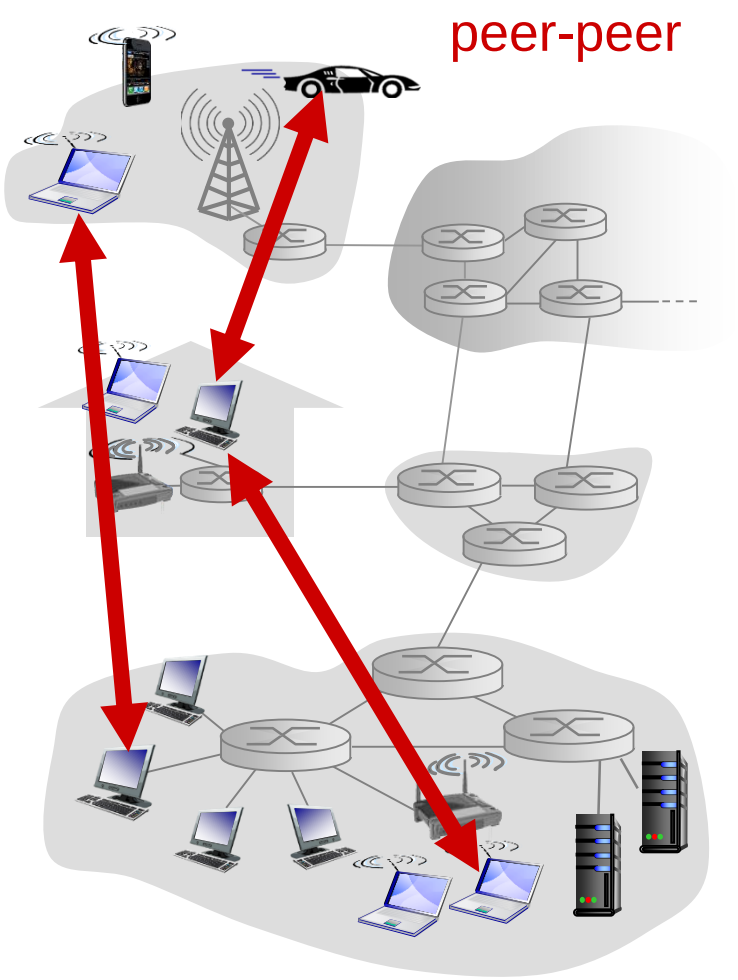
- always-on host
- permanent IP address
- data centers for scaling

clients:

- communicate with server
- intermittently connected
- use dynamic IP addresses
- do not normally communicate with each other

P2P architecture

- *a client is also a server*
- communicate with each other directly
- peers request services from other peers, or provide services in return
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management



Host Processes communicating

process: program running within a host/machine

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

clients, servers

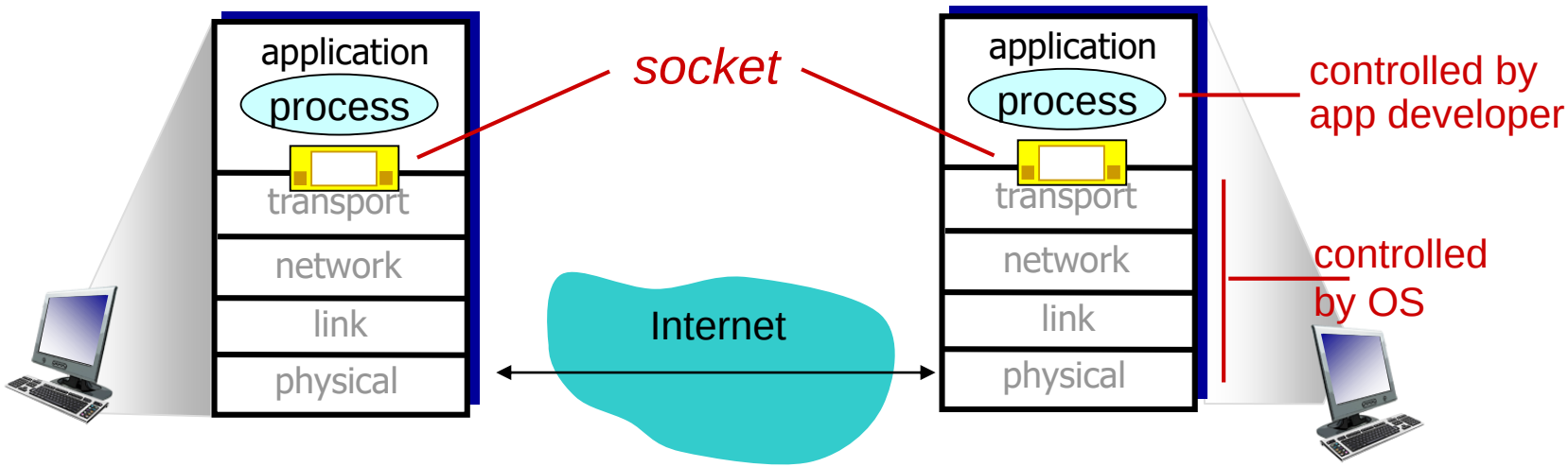
client process: process that initiates communication

server process: process that waits to be contacted

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

Sockets (or Berkeley 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 an *identifier*
- *host device* has a *unique* 32-bit IP address
- *Q:* is an IP address suffice for identifying the process?
  - *A:* no, ***many*** processes can be running on same host, sharing the **same** IP address

- *identifier* includes both IP address and port numbers.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to www.uvic.ca web server:
  - IP address: 142.104.197.120
  - port number: 80
- more shortly…

Application-layer protocols

- 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

Which transport service required?

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	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100's msec yes and no

Internet transport protocol 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, or connection setup,

Q: why bother? Why is there a UDP?

Internet applications and 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

See also: Well-known Internet Services, Protocols and Port Numbers

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

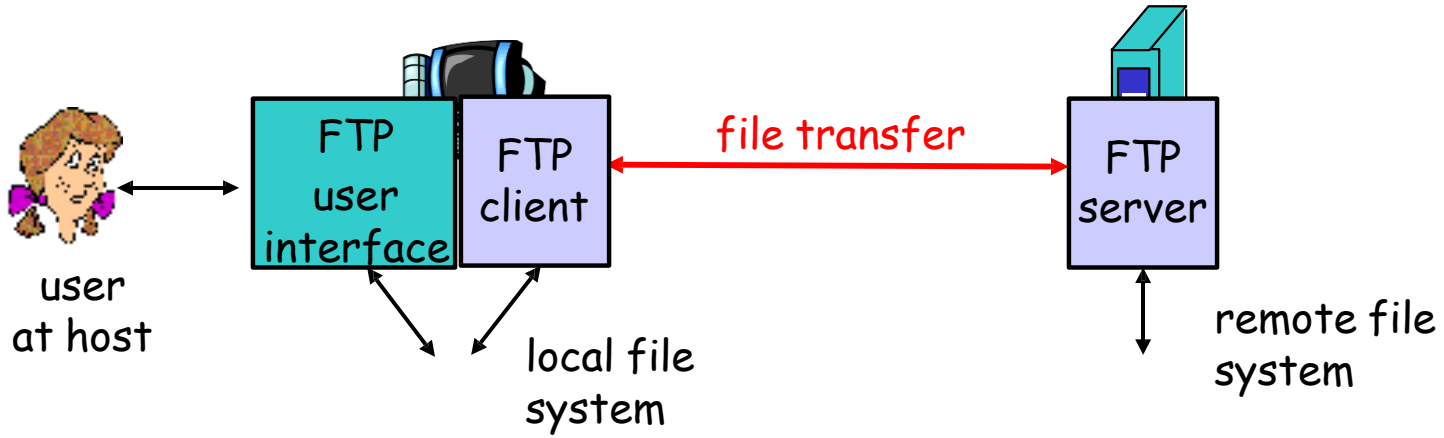
- apps use SSL libraries, that "talk" to TCP

SSL socket API

- cleartext passwords sent into socket traverse Internet encrypted
- see Chapter 8

SFTP

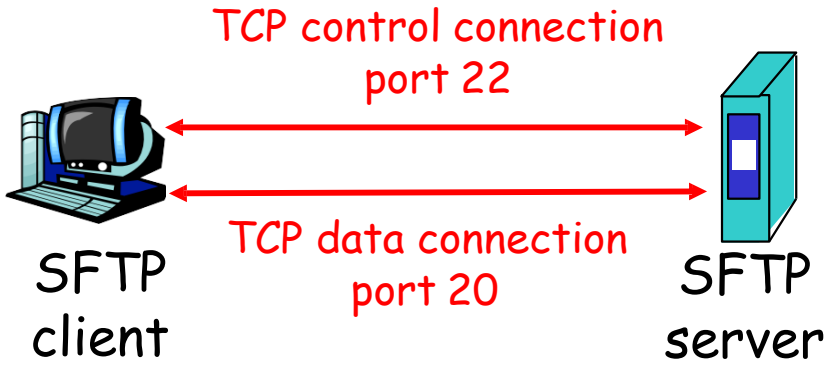
SFTP: secure file transfer protocol



- transfer file to/from remote host
- client/server model
  - *client:* side that initiates transfer (either to/from remote)
  - *server:* remote host
- ftp: RFC 959
- sftp server: port 22

FTP: separate control, data connections

- FTP client contacts SFTP server at port 22, TCP is transport protocol
- client authorized over control connection
- client browses remote directory by sending commands over control connection.
- when server receives file transfer command, server opens 2^{nd} TCP connection (for file) to client
- after transferring one file, server closes data connection.



- server opens another TCP data connection to transfer another file.
- control connection: "out of band"
- FTP server maintains "state": current directory, earlier authentication

```
MacBook-Pro-Retina-Mantis:~ mcheng$ sftp unix.uvic.ca
Password:
Connected to unix.uvic.ca.
sftp> dir
Archive          Deleted Items    Drafts           INBOX            Junk E-mail
Sent Items       mail             www              www-dev
sftp> cd www
sftp> dir
ex1              lab1             lab2             lab3             lab4
phishing.html
sftp> cd ex1
sftp> dir
autumn.jpg       fox.jpg          fox_Fotor.jpg    test.html        test1.html
sftp> get fox.jpg
Fetching /home7/60/mcheng/www/ex1/fox.jpg to fox.jpg
/home7/60/mcheng/www/ex1/fox.jpg                100%  856KB   4.8MB/s   00:00
sftp> quit
MacBook-Pro-Retina-Mantis:~ mcheng$
```

SFTP commands and responses

Sample commands:

- sent as ASCII text over control channel
- **USER *username***
- **PASS *password***
- **DIR** return list of file in current directory
- **GET filename** retrieves file
- **PUT filename** stores 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**

WWW and HTTP

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.,

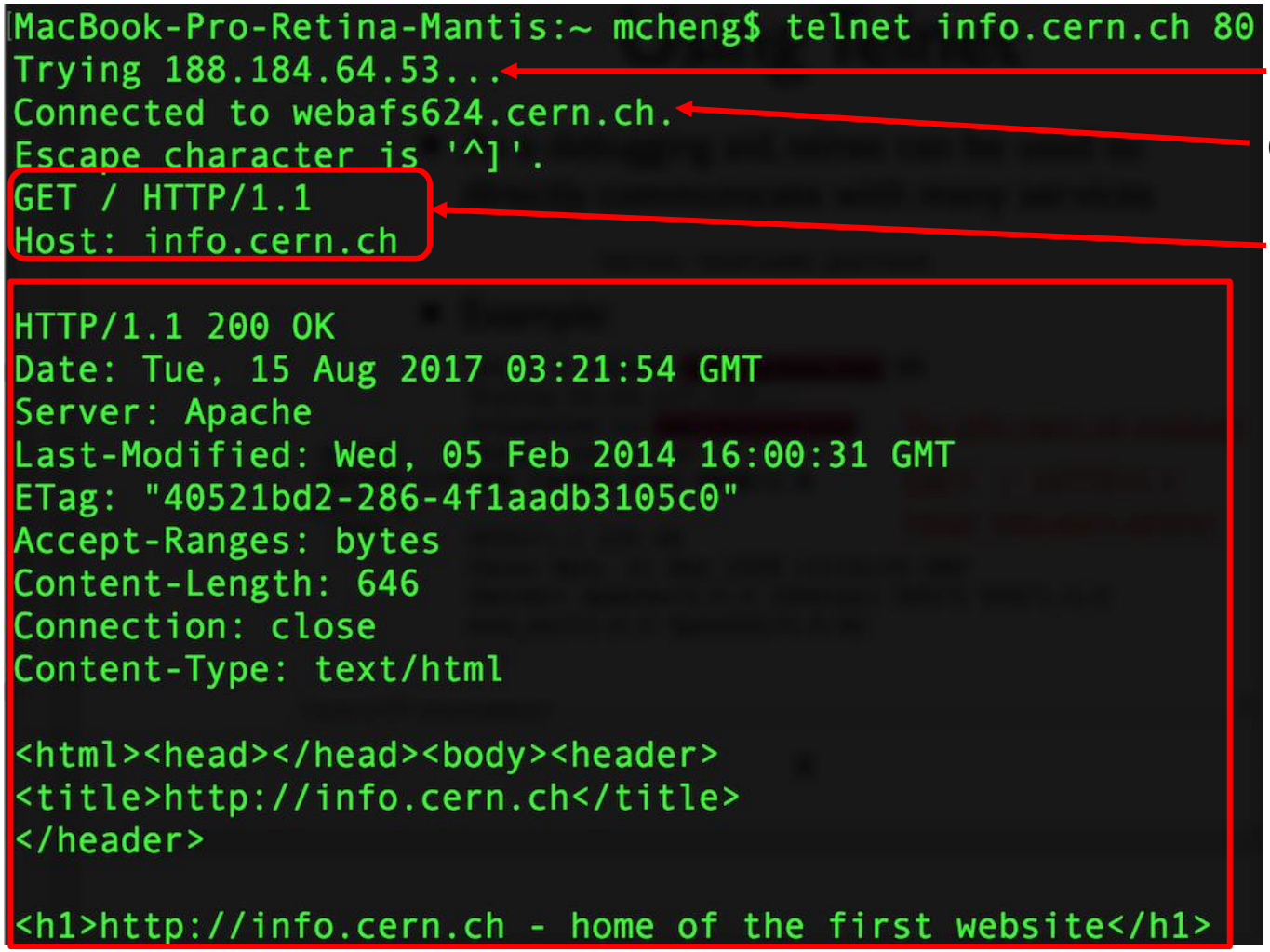
`www.someschool.edu/someDept/pic.gif`

host name | path name

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests


MacBook-Pro-Retina-Mantis:~ mcheng$ telnet info.cern.ch 80
Trying 188.184.64.53...
Connected to webafs624.cern.ch.
Escape character is '^]'.
GET / HTTP/1.1
Host: info.cern.ch
HTTP/1.1 200 OK
Date: Tue, 15 Aug 2017 03:21:54 GMT
Server: Apache
Last-Modified: Wed, 05 Feb 2014 16:00:31 GMT
ETag: "40521bd2-286-4f1aadb3105c0"
Accept-Ranges: bytes
Content-Length: 646
Connection: close
Content-Type: text/html
<html><head></head><body><header>
<title>http://info.cern.ch</title>
</header>
<h1>http://info.cern.ch - home of the first website</h1>
connection request
connection response
HTTP request
HTTP response

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

Stateful protocols are complex!

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

HTTP connections

non-persistent HTTP (1.0 default)

- at most one object sent over a TCP connection; connection then closed
- downloading multiple objects required multiple connections

persistent HTTP (1.1 default)

- multiple objects can be sent over single TCP connection between client, server

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

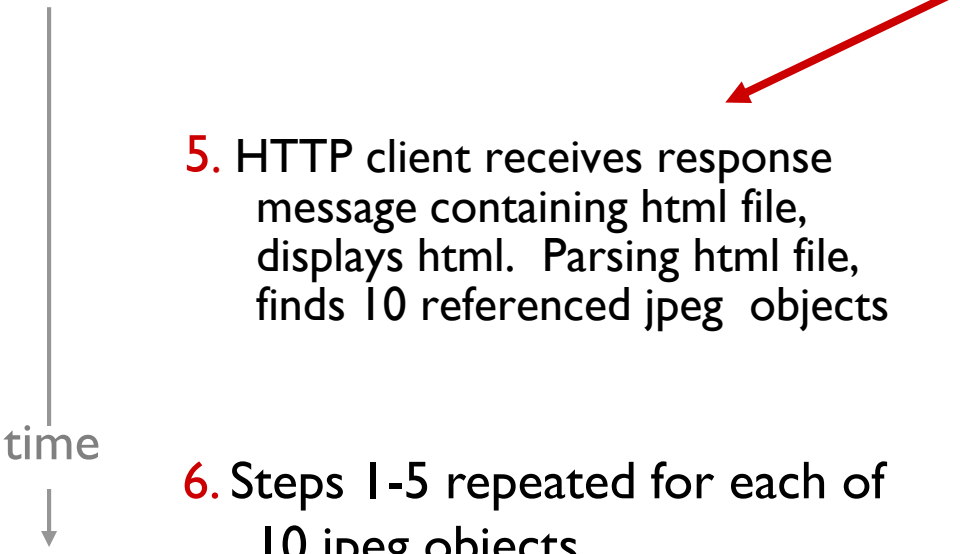
1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

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

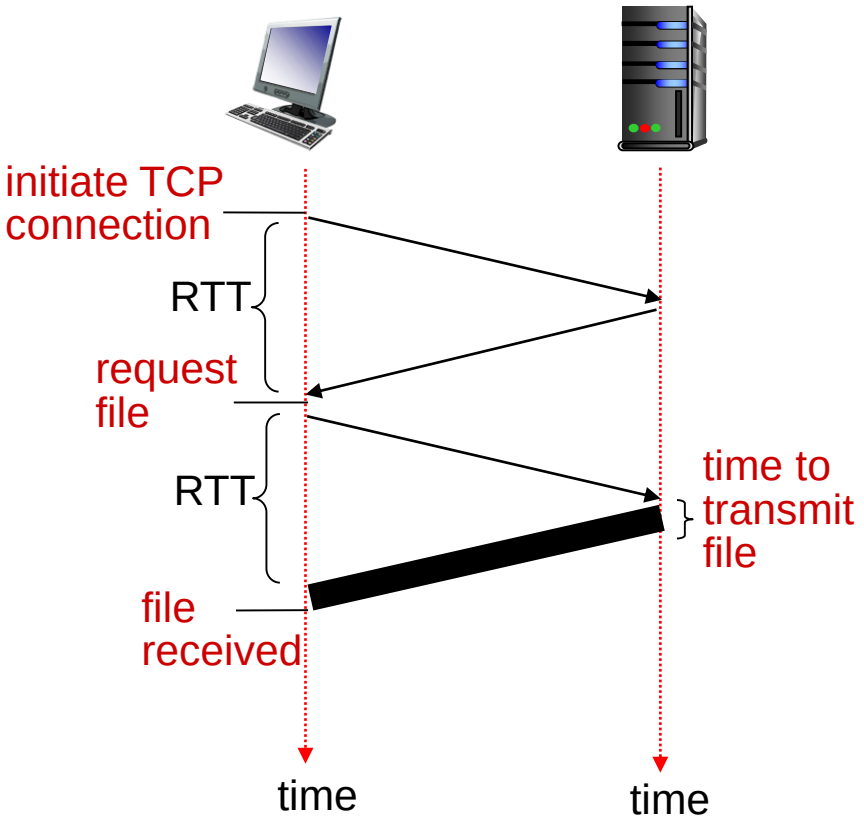
6. Steps 1-5 repeated for each of 10 jpeg objects

Non-persistent HTTP: response time

RTT (Round-Trip-Time): 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 response to return
- plus file transmission time
- non-persistent HTTP response time = 2 x RTT+ transmission time



Persistent HTTP

non-persistent HTTP issues:

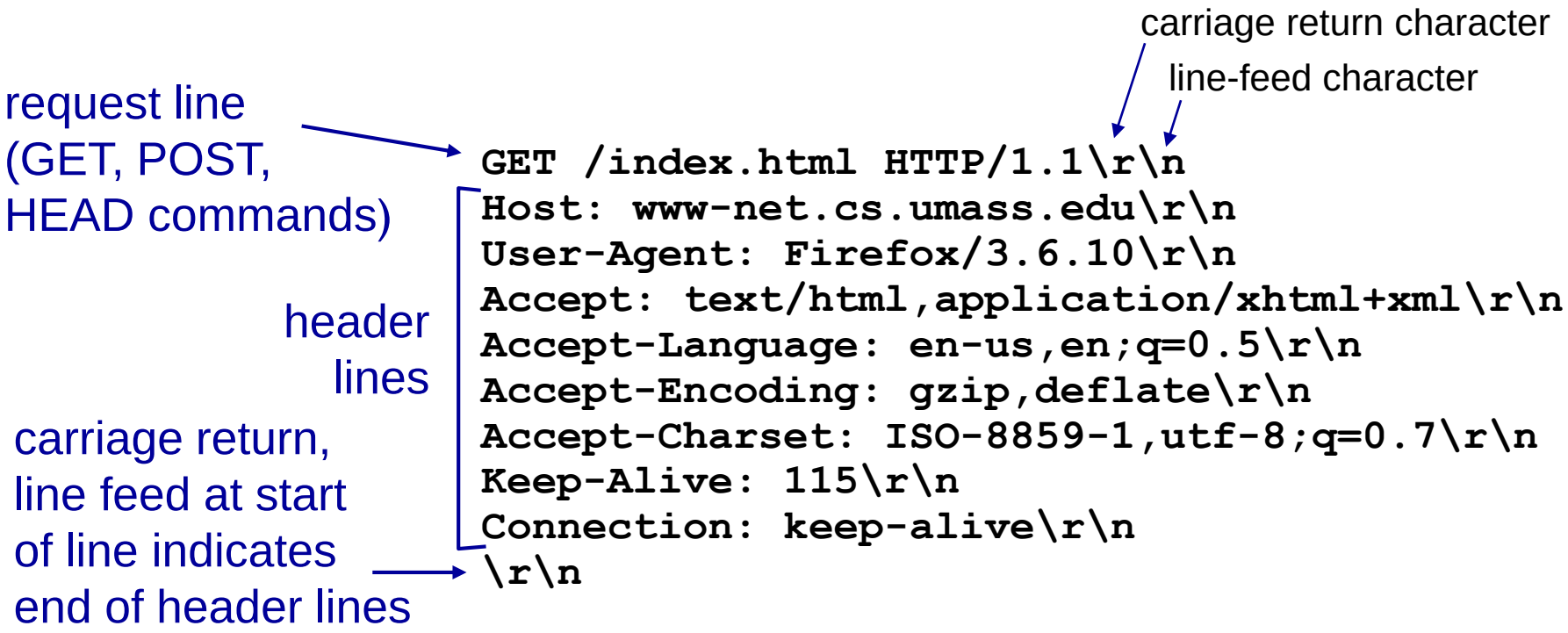
- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

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

HTTP request message

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



* Check out: http://gaia.cs.umass.edu/kurose_ross/interactive/

Python Socket Programming

client.py

```
1  #
2  # this is a simple test of using socket in Python
3  # the host "info.cern.ch" still shows the world's first webpage
4  #
5  # Try also: curl info.cern.ch
6  # this will produce an identical output
7  #
8
9  from socket import *
10
11 s = socket( AF_INET, SOCK_STREAM )
12
13 s.connect( ("info.cern.ch", 80) )
14
15 s.send("GET / HTTP/1.1\nHost: info.cern.ch\n\n")
16
17 print s.recv(1024)
18
```

TCP stream

HTTP connection

HTTP request

```
MacBook-Pro-Retina-Mantis:Desktop mcheng$ python client.py
HTTP/1.1 200 OK
Date: Thu, 03 Aug 2017 02:15:21 GMT
Server: Apache
Last-Modified: Wed, 05 Feb 2014 16:00:31 GMT
ETag: "40521bd2-286-4f1aadb3105c0"
Accept-Ranges: bytes
Content-Length: 646
Connection: close
Content-Type: text/html

<html><head></head><body><header>
<title>http://info.cern.ch</title>
</header>

<h1>http://info.cern.ch - home of the first website</h1>
<p>From here you can:</p>
<ul>
<li><a href="http://info.cern.ch/hypertext/WWW/TheProject.html">Browse the first
 website</a></li>
<li><a href="http://line-mode.cern.ch/www/hypertext/WWW/TheProject.html">Browse
the first website using the line-mode browser simulator</a></li>
```

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 ...
```

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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

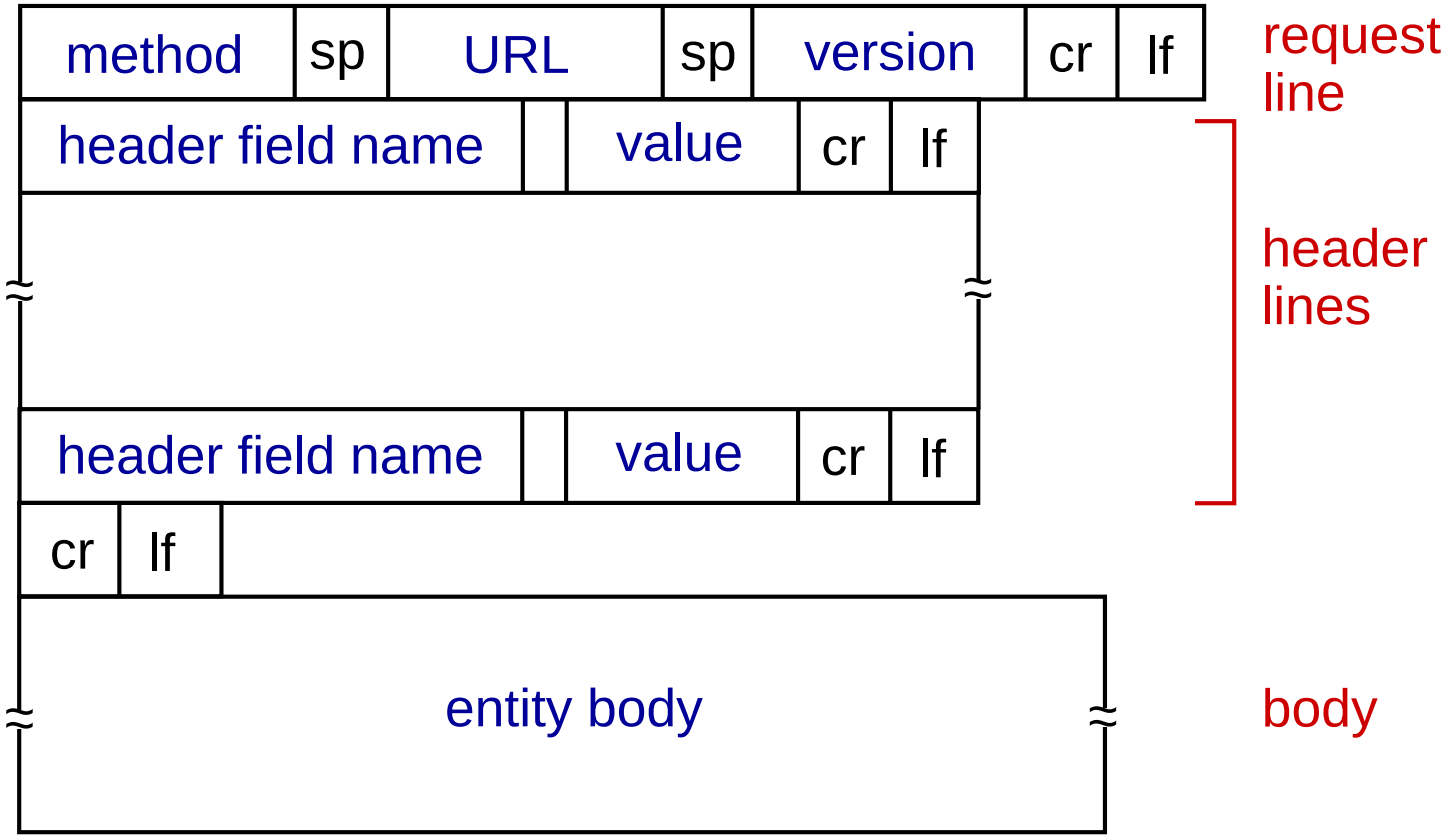
- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

HTTP request message: general format



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

User-server state: cookies

many Web sites use cookies

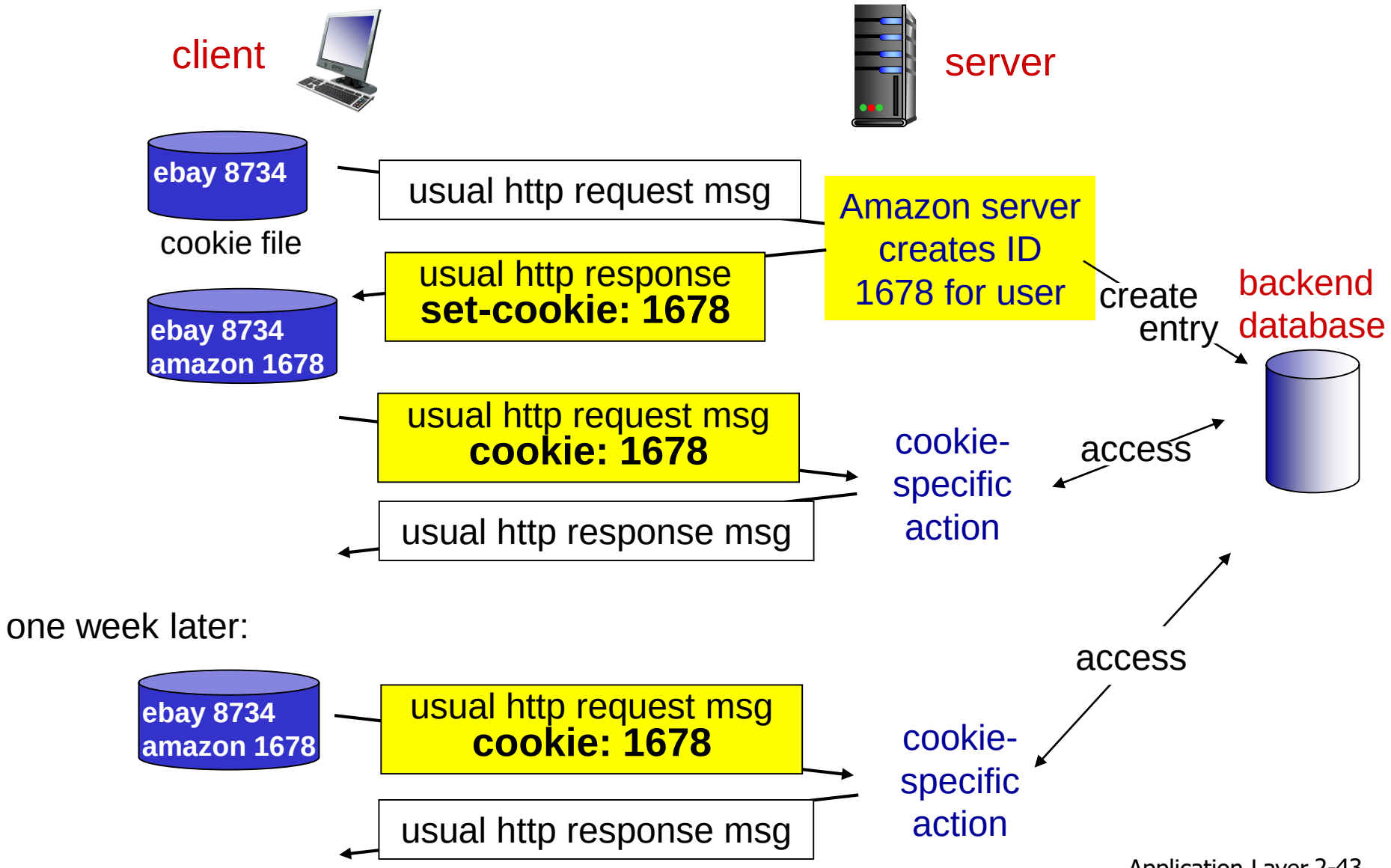
four components:

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

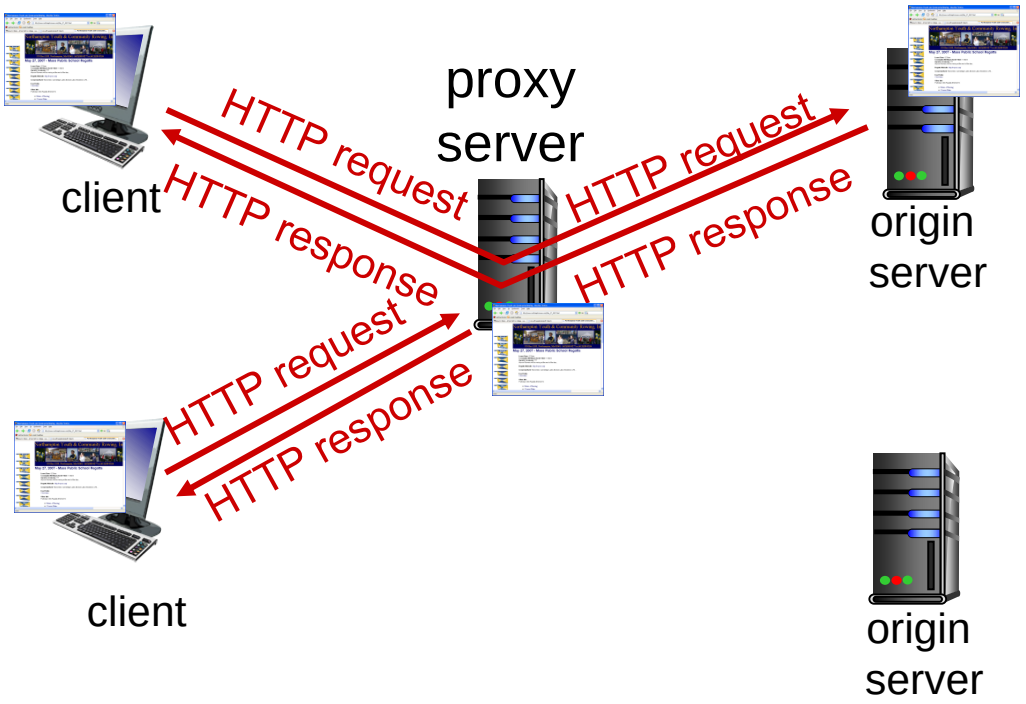
how to keep "state":

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

Web proxy server

goal: satisfy client requests without involving origin server

- user sets browser: web accesses via proxy
- browser sends all HTTP requests to proxy
  - If requested object in proxy then returns object
  - else proxy requests object from origin server, then returns object to client and keeps a copy



More about Web proxy

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

why Web proxy?

- reduce response time for client requests
- reduce traffic on an institution's access link
- Internet dense with proxies: enables "poor" content providers to effectively deliver content (so too does P2P file sharing)

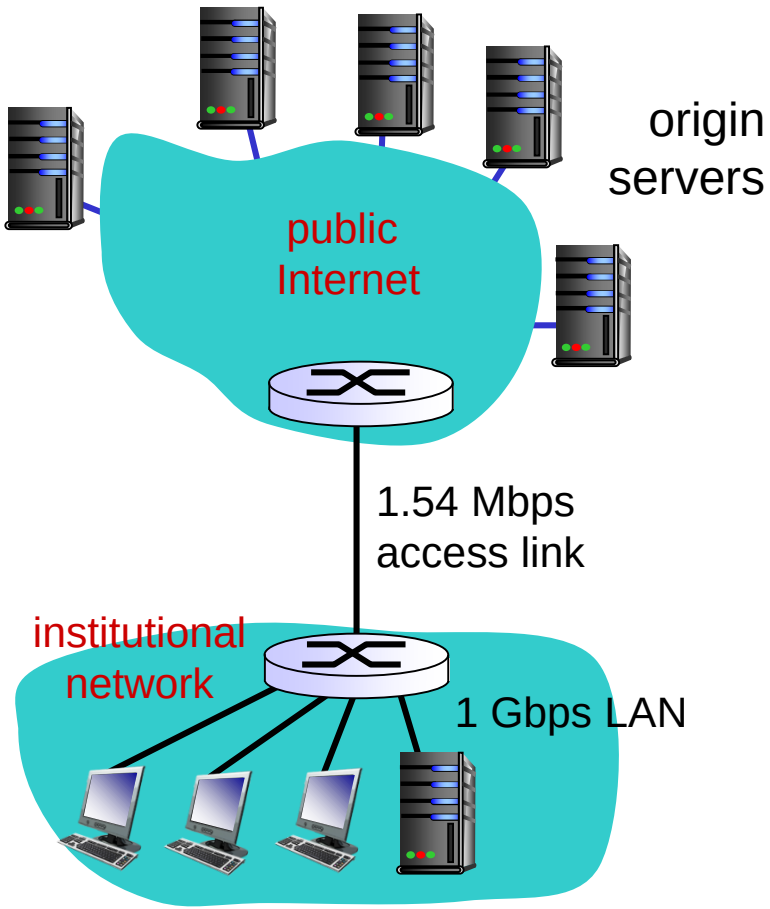
Proxy example:

assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

- LAN utilization: 0.15% *problem!*
- access link utilization = 99%
- total delay = RTT delay + access (queueing) delay + LAN delay
  = 2 sec + minutes + 100 microsecs



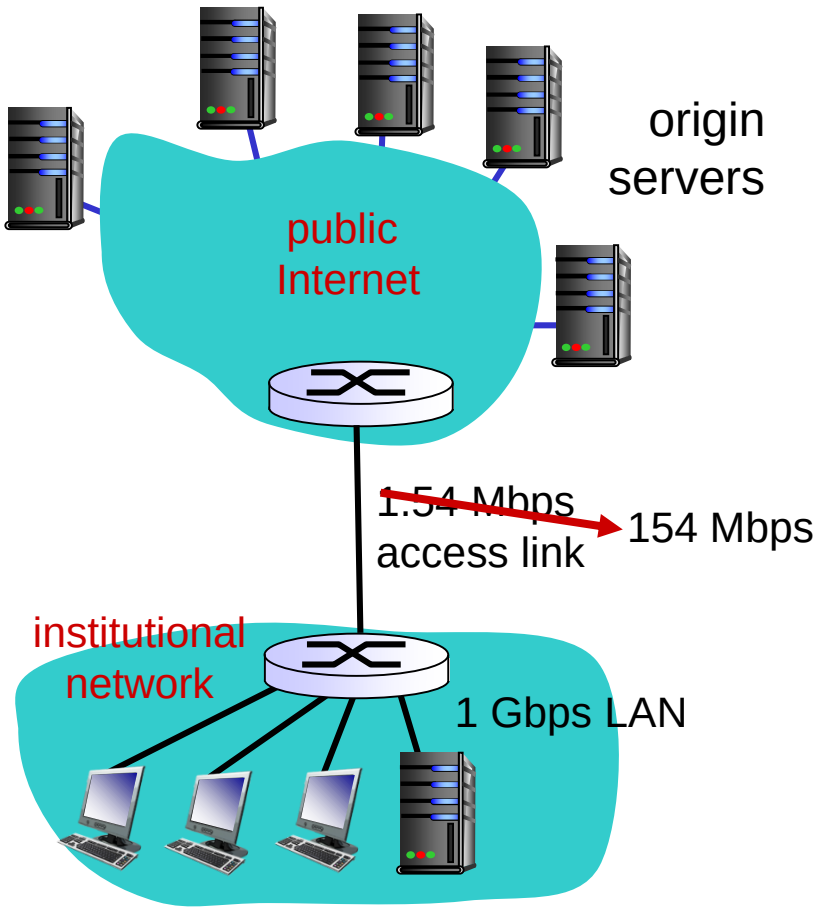
Proxy example: faster access link (100x)

assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec (why?)
- access link rate: ~~1.54 Mbps~~ 154 Mbps

consequences:

- LAN utilization: 0.15%
- access link utilization = ~~99%~~ 9.9%
- total delay = RTT delay + access (queueing) delay + LAN delay
  = 2 sec + ~~minutes~~ + 100 microsecs
  millisecs



Cost: increased access link speed (not cheap!)

Proxy example: install local proxy

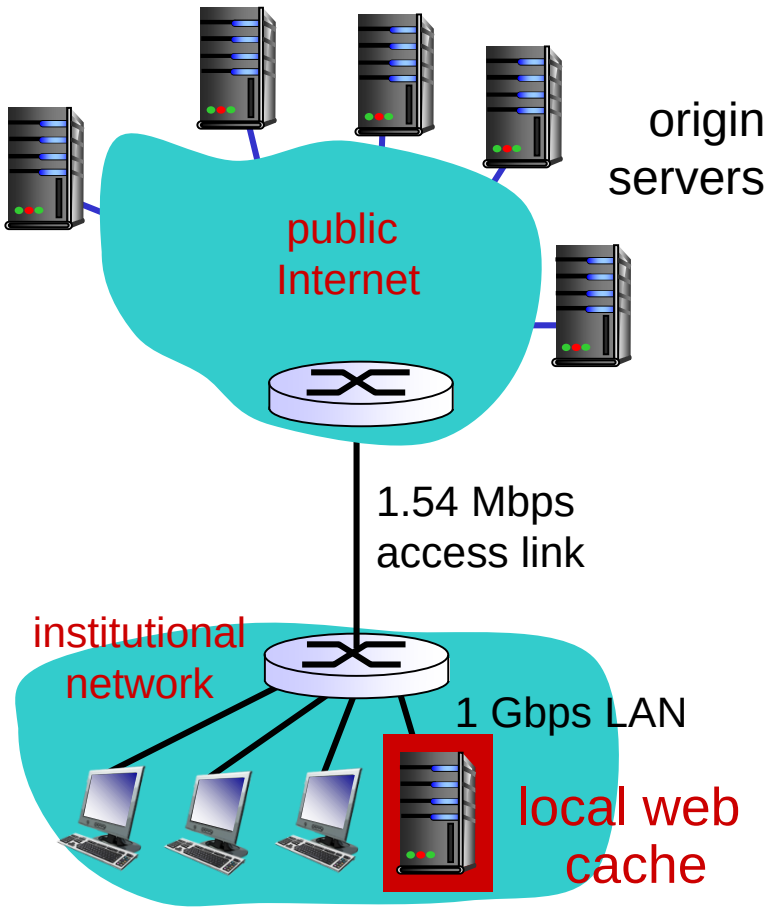
assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

- LAN utilization: 0.15%
- access link utilization = ?
- total delay = ?

How to compute link utilization, delay?



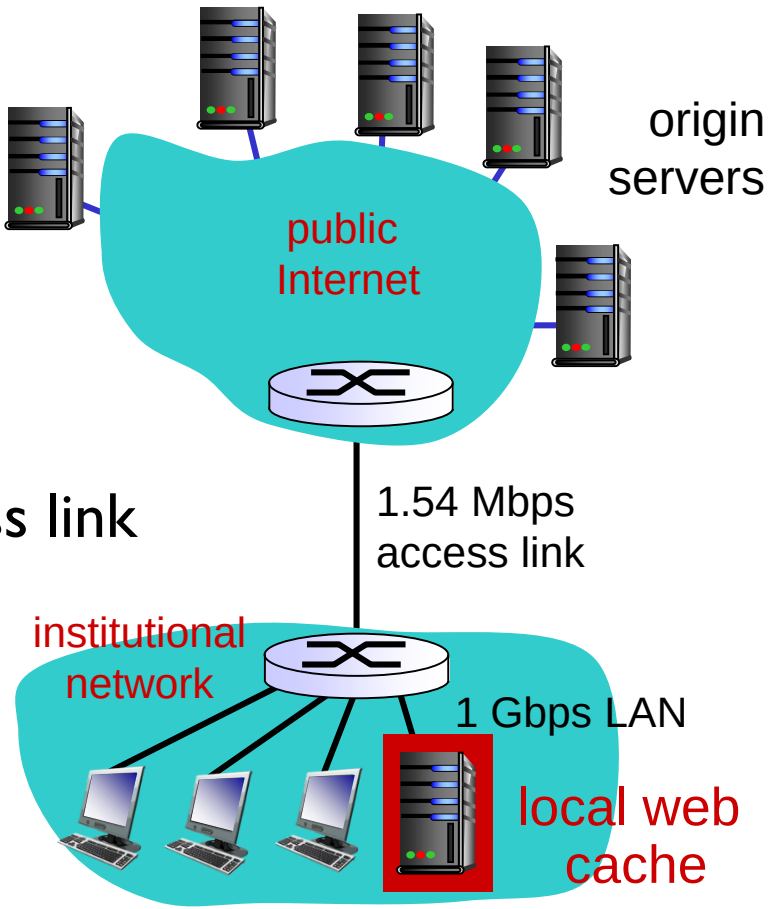
Cost: web proxy (relatively cheap!)

Proxy example: install local proxy

Calculating access link utilization, delay with proxy:

- suppose proxy hit rate is 0.4
  - 40% requests satisfied at proxy, 60% requests satisfied at origin
- access link utilization:
  - 60% of requests use access link
- data rate to browsers over access link

  = 0.6*1.50 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58
- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at proxy)
  - = 0.6 (2.0) + 0.4 (~msecs) = ~ 1.2 secs
  - less than with 154 Mbps link (and cheaper too!)

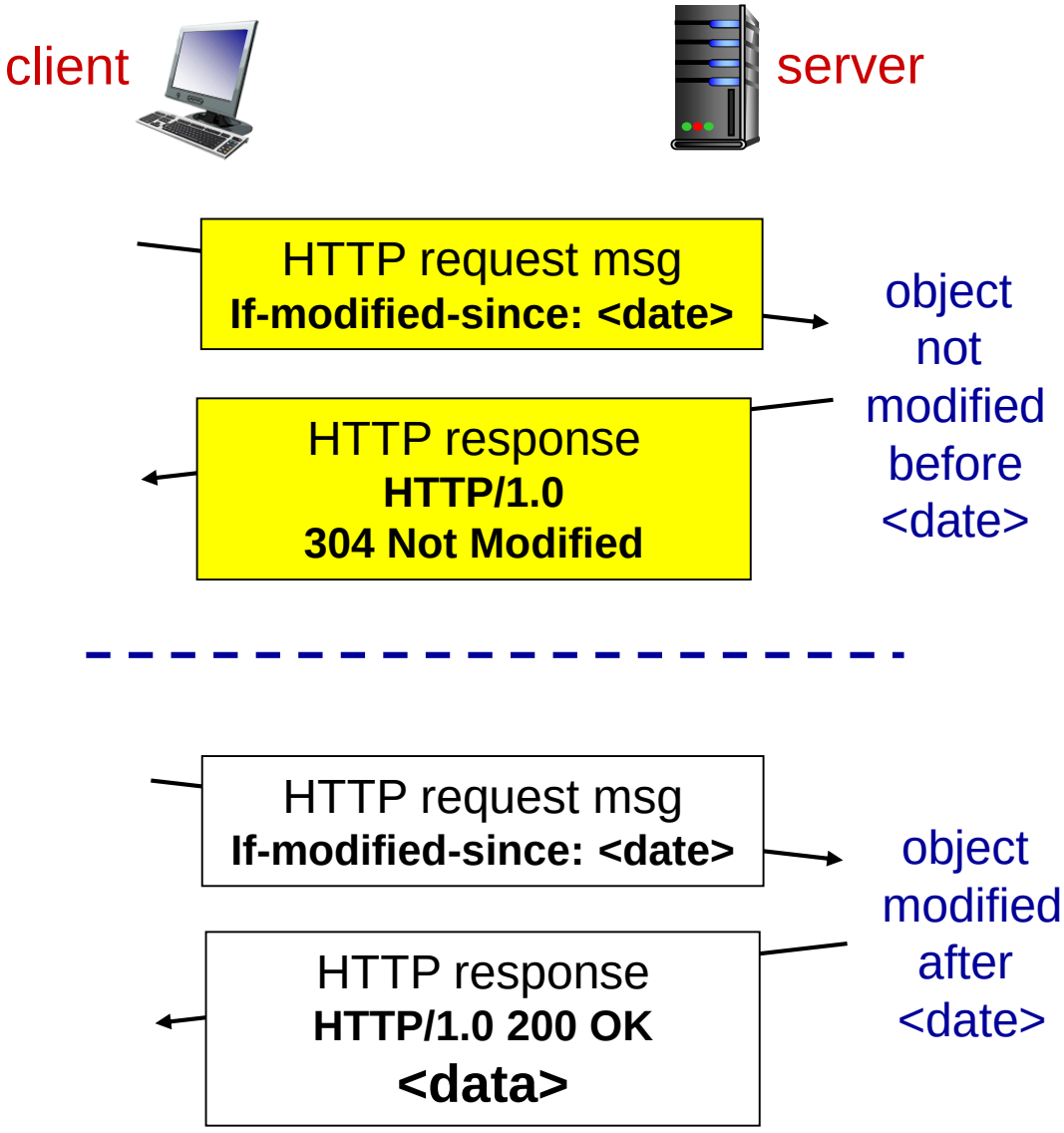


Conditional GET (used by proxy/browser)

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



Email

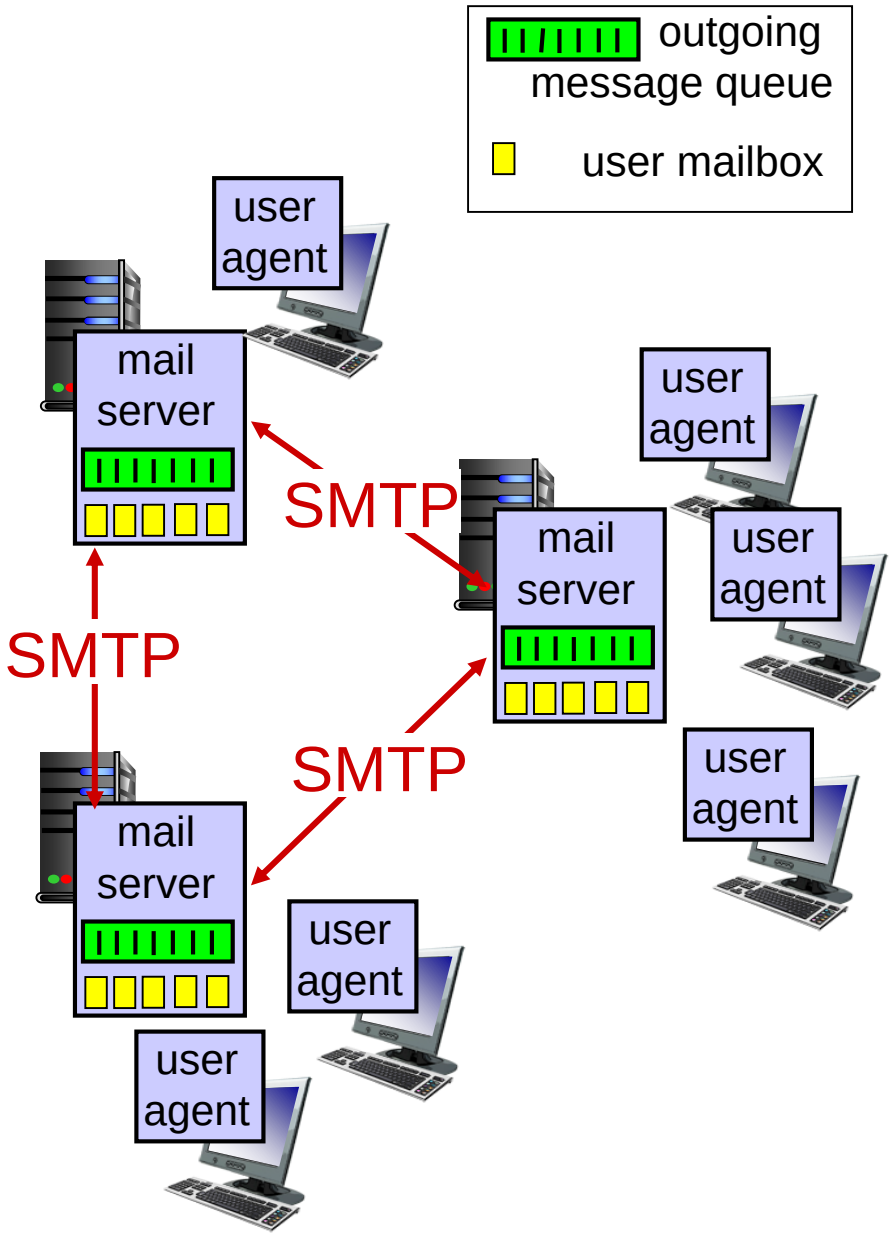
Electronic mail

Three major components:

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

User Agent

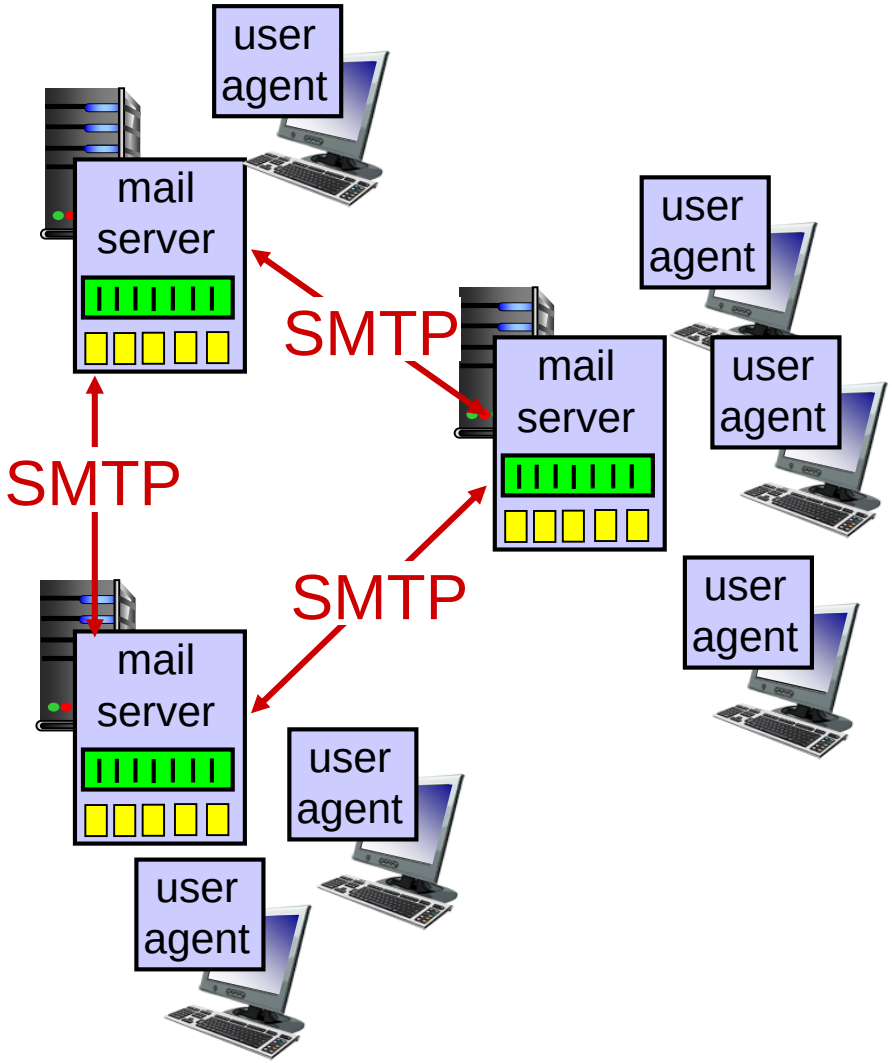
- "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, OS X mail client
- outgoing, incoming messages stored on server



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

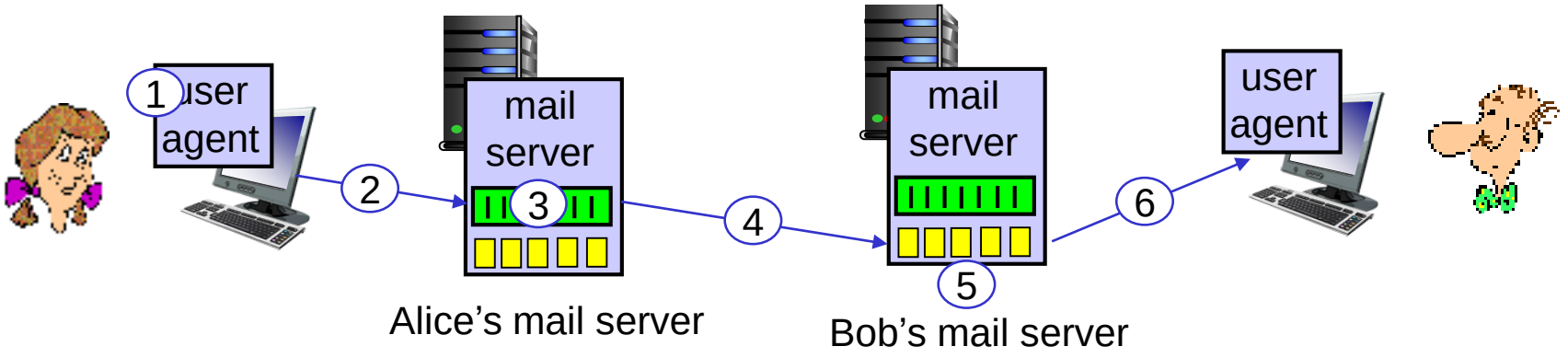


Electronic Mail: SMTP [RFC 2821]

- 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 (like HTTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII

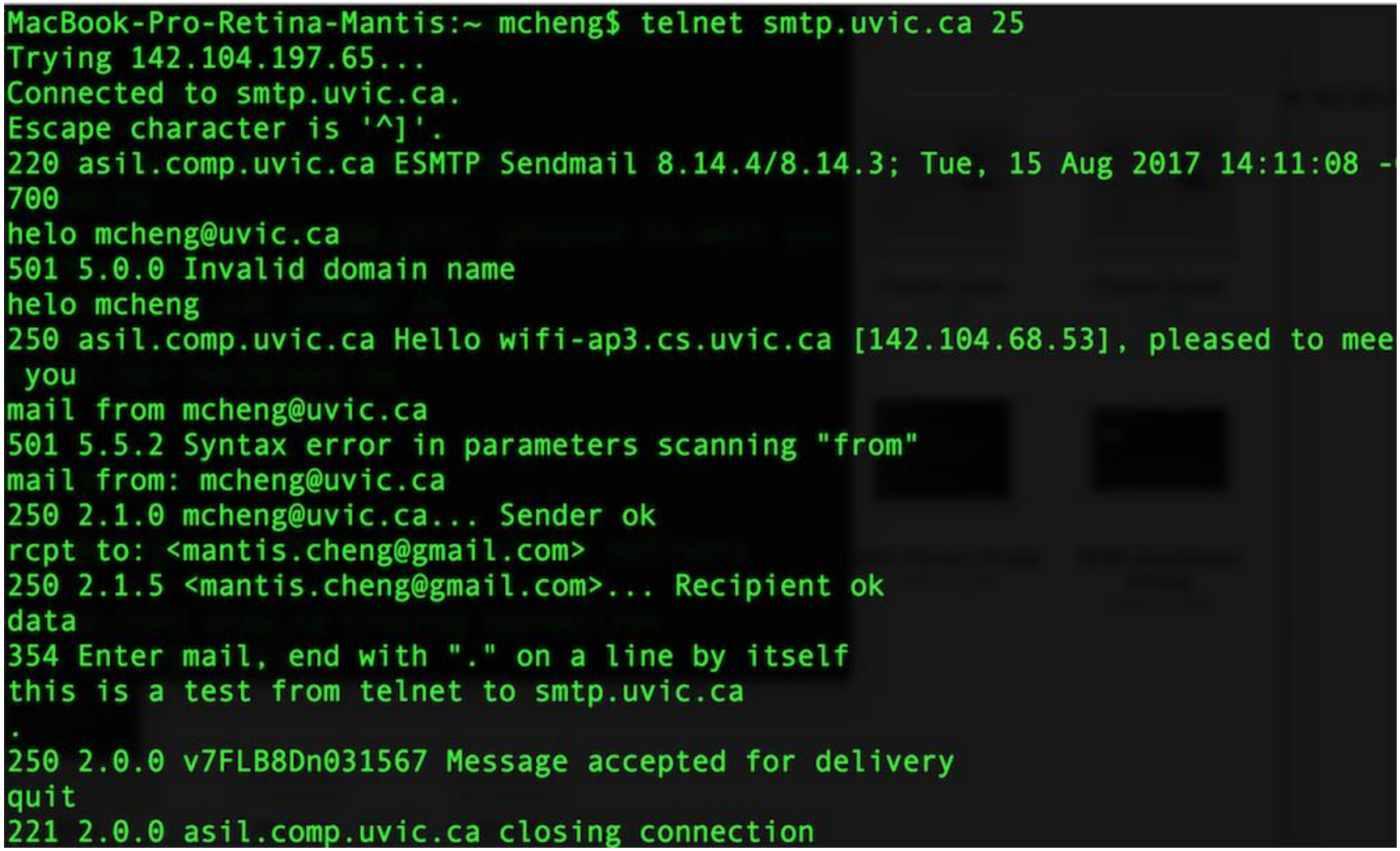
Scenario: Alice sends message to Bob

1) Alice uses UA to compose message "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 (port 25)

```
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
```


MacBook-Pro-Retina-Mantis:~ mcheng$ telnet smtp.uvic.ca 25
Trying 142.104.197.65...
Connected to smtp.uvic.ca.
Escape character is '^]'.
220 asil.comp.uvic.ca ESMTP Sendmail 8.14.4/8.14.3; Tue, 15 Aug 2017 14:11:08 -
700
helo mcheng@uvic.ca
501 5.0.0 Invalid domain name
helo mcheng
250 asil.comp.uvic.ca Hello wifi-ap3.cs.uvic.ca [142.104.68.53], pleased to mee
you
mail from mcheng@uvic.ca
501 5.5.2 Syntax error in parameters scanning "from"
mail from: mcheng@uvic.ca
250 2.1.0 mcheng@uvic.ca... Sender ok
rcpt to: <mantis.cheng@gmail.com>
250 2.1.5 <mantis.cheng@gmail.com>... Recipient ok
data
354 Enter mail, end with "." on a line by itself
this is a test from telnet to smtp.uvic.ca
.
250 2.0.0 v7FLB8Dn031567 Message accepted for delivery
quit
221 2.0.0 asil.comp.uvic.ca closing connection

SMTP direction interaction: (SPAM!!!)

```
1. Open a SSH session to "linux.csc.uvic.ca".

> ssh linux.csc.uvic.ca   // or directly if telnet is installed

2. Try telnet to "smtp.uvic.ca" with port number 25

> telnet smtp.uvic.ca 25
...
> helo mcheng     // your netlink ID
...
> mail from: mcheng@uvic.ca    // your UVic email
...
> rcpt to: <mantis.cheng@gmail.com>   // your other email addr
...
> data
...
> this is a test from telnet directly to smtp.uvic.ca
> .
...
> quit
```

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 message
- SMTP: multiple objects sent in multipart message

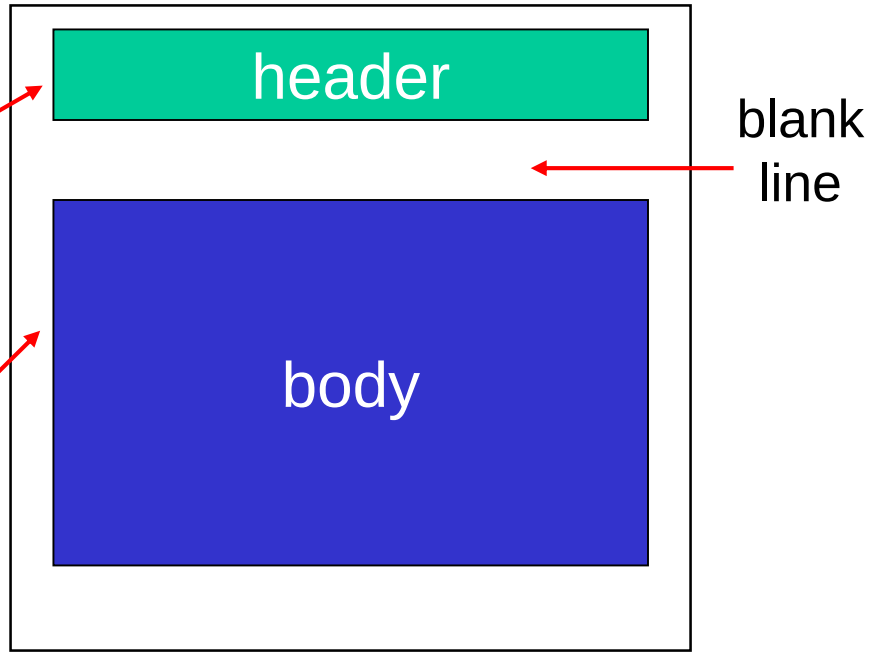
Mail message format

SMTP: protocol for exchanging email messages

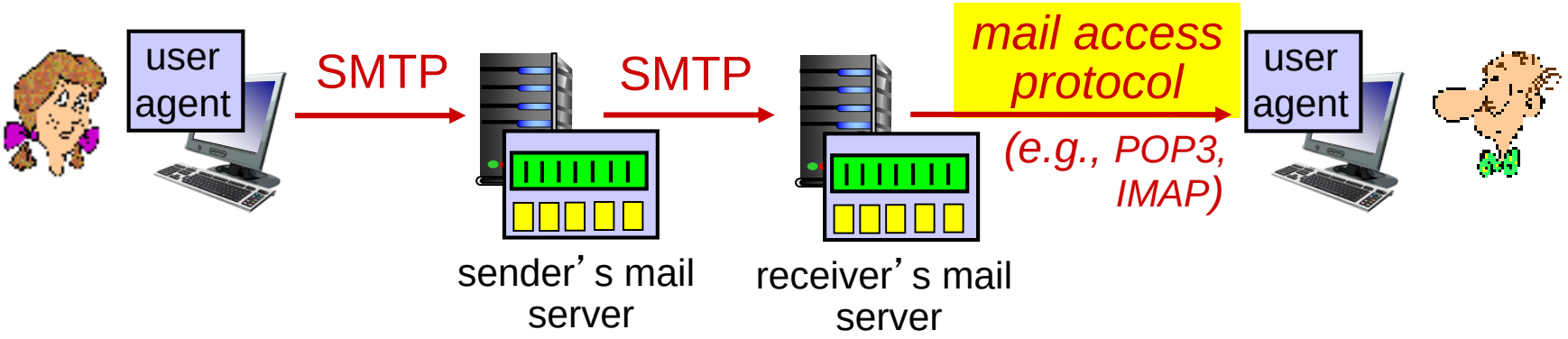
RFC 822: standard for text message format:

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

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



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 messages on server
  - HTTP: gmail, Hotmail, Yahoo! Mail, etc.

POP3 protocol (port 110)

authorization phase

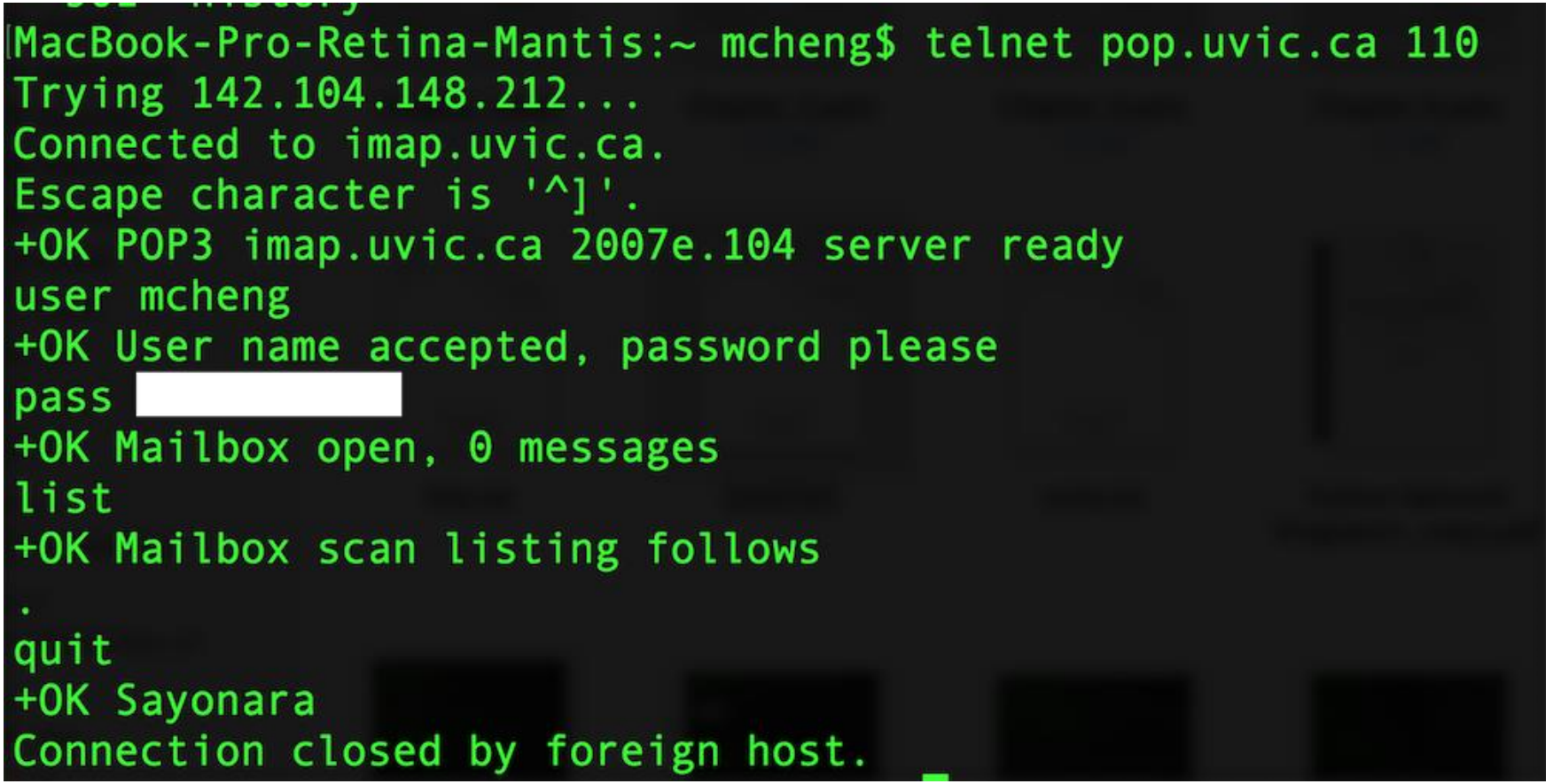
- client commands:
  - **`user`**: declare username
  - **`pass`**: password
- server responses
  - **`+OK`**
  - **`-ERR`**

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
```


MacBook-Pro-Retina-Mantis:~ mcheng$ telnet pop.uvic.ca 110
Trying 142.104.148.212...
Connected to imap.uvic.ca.
Escape character is '^]'.
+OK POP3 imap.uvic.ca 2007e.104 server ready
user mcheng
+OK User name accepted, password please
pass
+OK Mailbox open, 0 messages
list
+OK Mailbox scan listing follows
.
quit
+OK Sayonara
Connection closed by foreign host.

POP3 direction interaction:

```
1. Open a SSH session to "linux.csc.uvic.ca".

> ssh linux.csc.uvic.ca   // or directly if telnet is installed

2. Try telnet to "pop.uvic.ca" with port number 110

> telnet pop.uvic.ca 110
...
> user mcheng   //  your netlink ID
...
> pass xxxxxxx  // your netlink password
...
> list
...
> retr 1
...

> quit
...
```

POP3 vs IMAP

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

DNS

DNS: domain name system

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

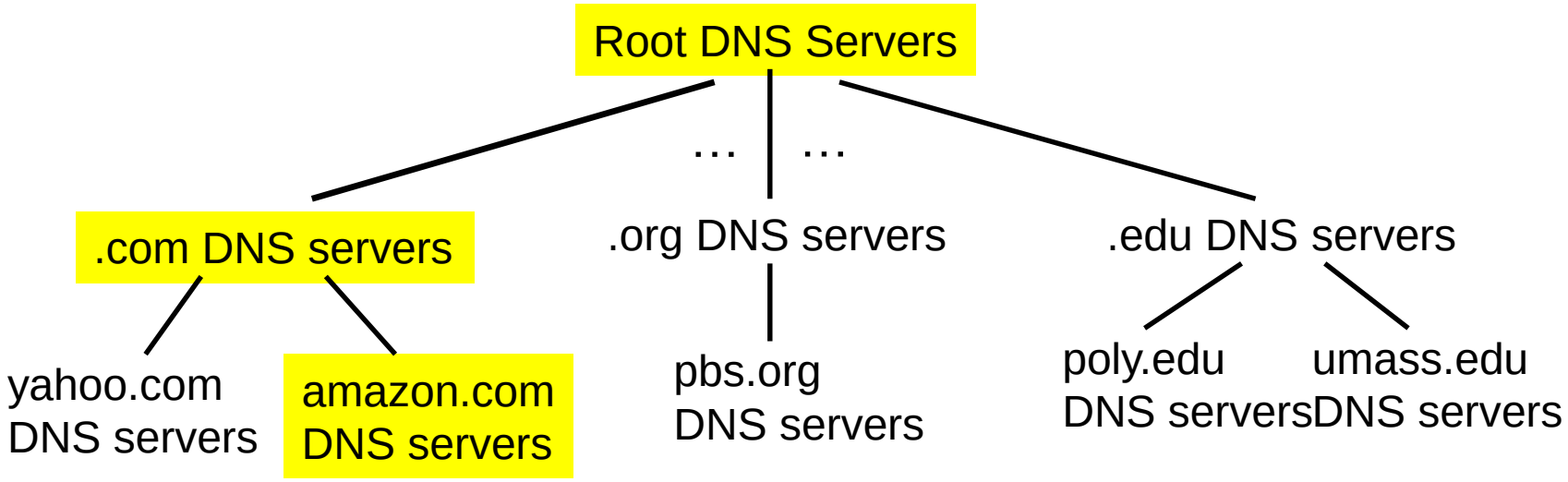
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated web servers: many IP addresses correspond to one name

why not centralize DNS?

- single point of failure
- traffic volume, not scalable!
- distant centralized database
- difficult to maintain

A: doesn't scale!

DNS: a distributed, hierarchical database

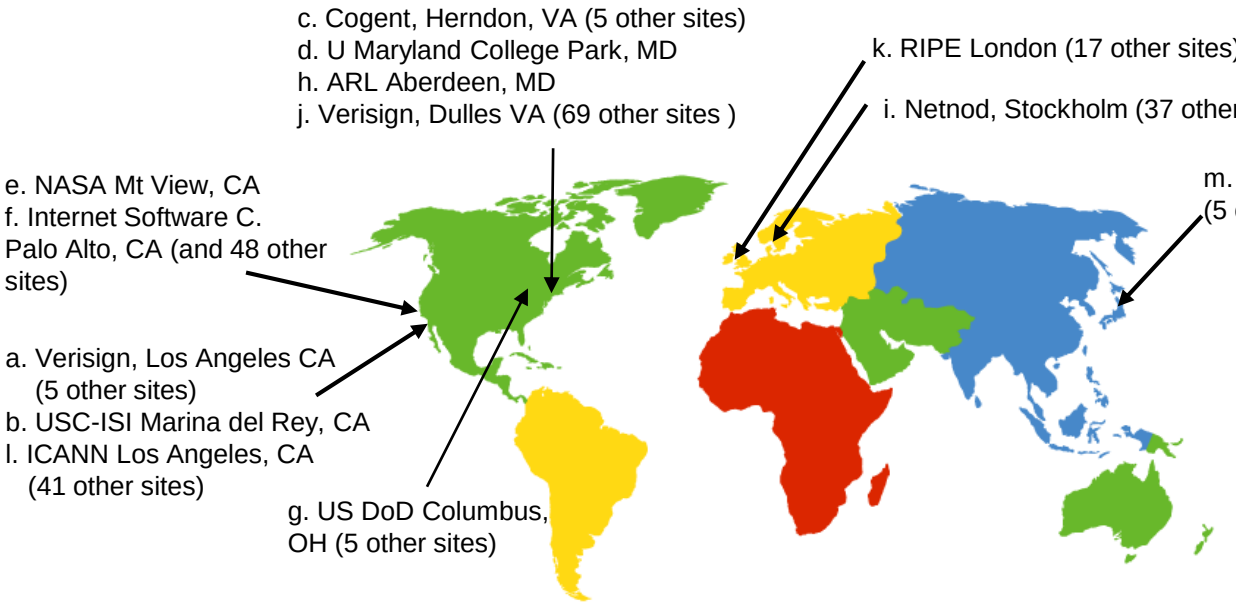


client wants IP for www.amazon.com; 1st approximation:

- client contacts its local DNS server
- local DNS queries root server to find .com DNS server
- local DNS queries .com DNS server to get amazon.com DNS server
- local DNS 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 servers if name mapping not known
  - gets mapping
  - returns mapping to local name server



13 logical root name "servers" worldwide

- *each "server" replicated many times*

List of Root name servers

HOSTNAME	IP ADDRESSES	MANAGER
a.root-servers.net	198.41.0.4, 2001:503:ba3e::2:30	VeriSign, Inc.
b.root-servers.net	192.228.79.201, 2001:500:200::b	University of Southern California (ISI)
c.root-servers.net	192.33.4.12, 2001:500:2::c	Cogent Communications
d.root-servers.net	199.7.91.13, 2001:500:2d::d	University of Maryland
e.root-servers.net	192.203.230.10, 2001:500:a8::e	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241, 2001:500:2f::f	Internet Systems Consortium, Inc.
g.root-servers.net	192.112.36.4, 2001:500:12::d0d	US Department of Defense (NIC)
h.root-servers.net	198.97.190.53, 2001:500:1::53	US Army (Research Lab)
i.root-servers.net	192.36.148.17, 2001:7fe::53	Netnod
j.root-servers.net	192.58.128.30, 2001:503:c27::2:30	VeriSign, Inc.
k.root-servers.net	193.0.14.129, 2001:7fd::1	RIPE NCC
l.root-servers.net	199.7.83.42, 2001:500:9f::42	ICANN
m.root-servers.net	202.12.27.33, 2001:dc3::35	WIDE Project

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 organizations or service providers

Local DNS name server

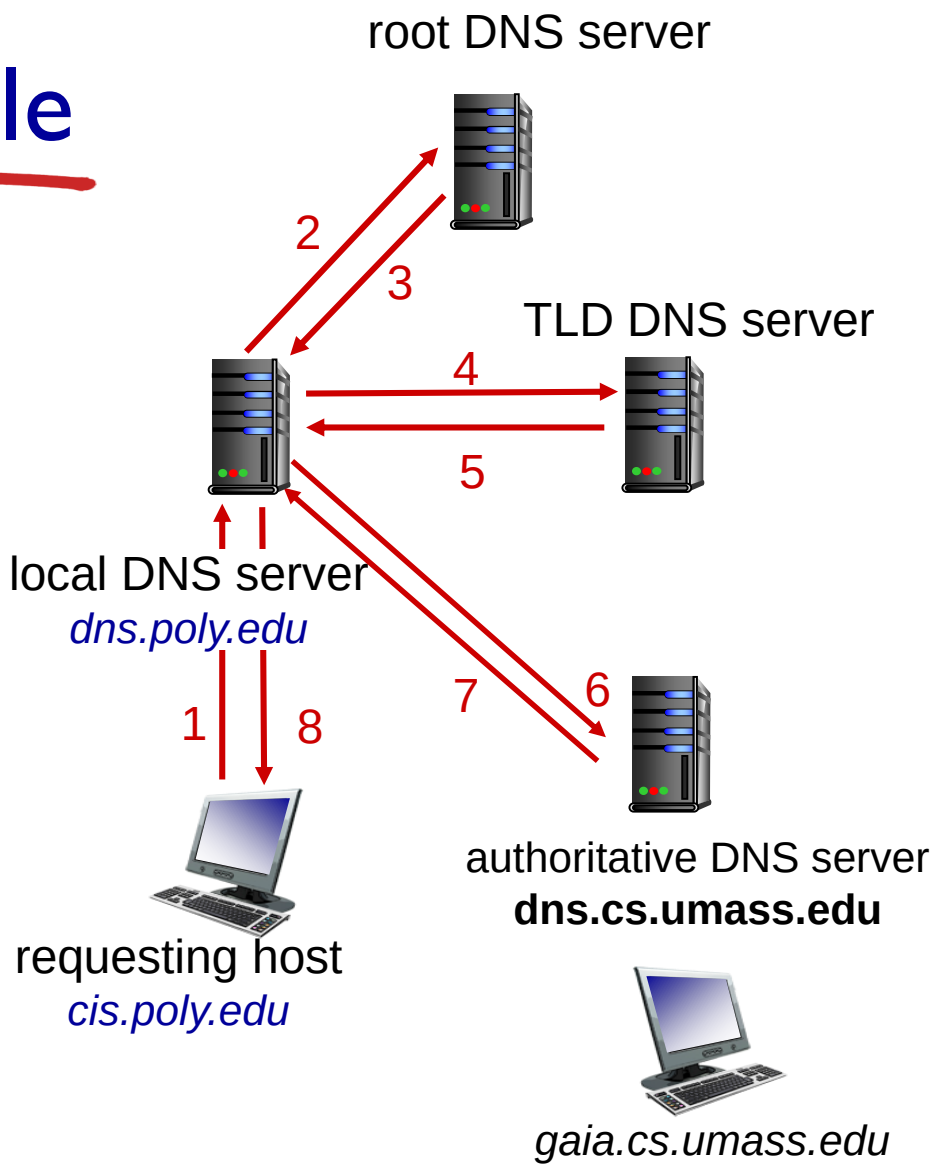
- does not strictly belong to DNS hierarchy
- each ISP (residential ISP, company, university) has one
  - called "default name server"
- when host makes DNS query, query is sent to its local DNS server first
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  -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

DNS name resolution example

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

iterated query:

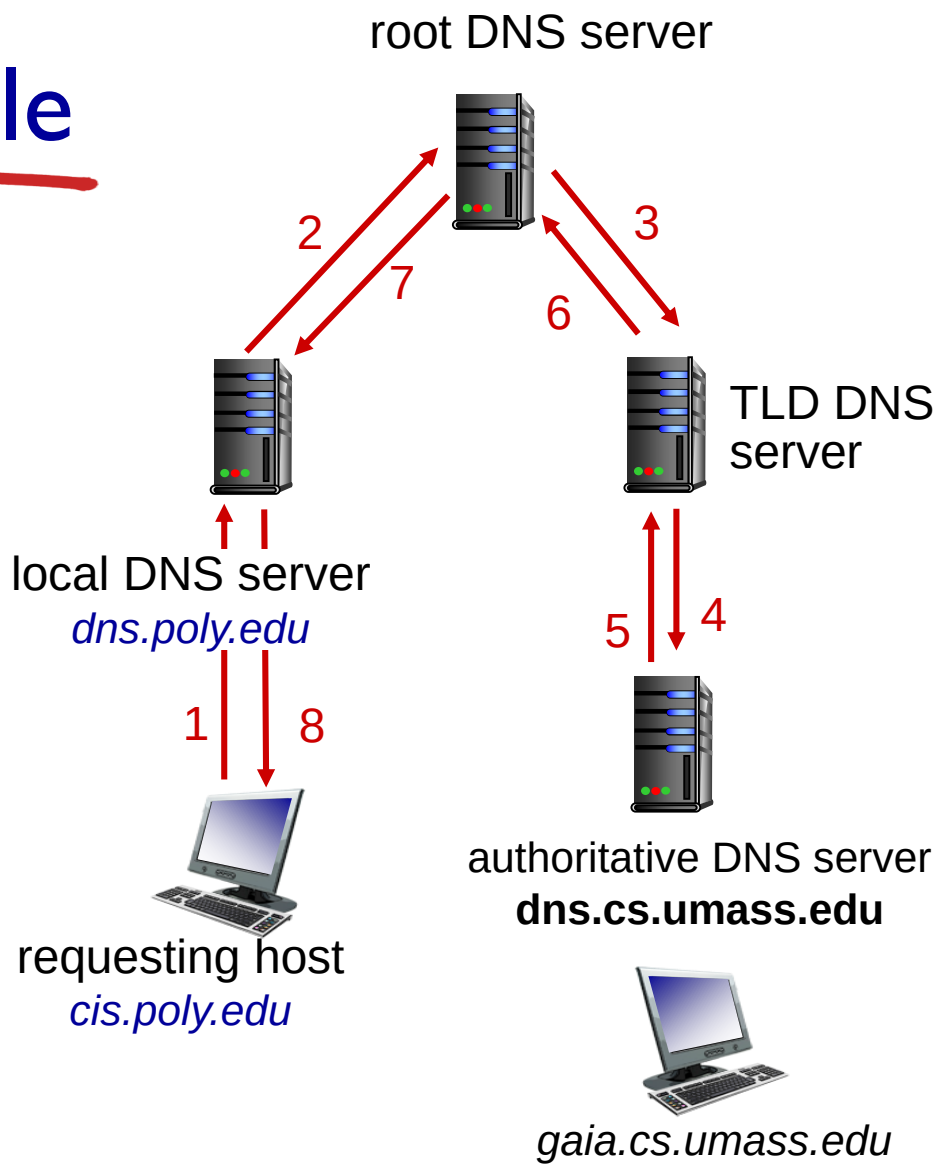
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



DNS: caching, updating records

- once a name server learns a mapping, it *caches* the mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically are 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 a well-known host changes its IP address, it may not be known Internet-wide until all cached names expire

DNS records

DNS: distributed database 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**

dig uvic.ca any

```
;; QUESTION SECTION:
;uvic.ca.                       IN      ANY

;; ANSWER SECTION:
uvic.ca.                299     IN      SOA     dns1.uvic.ca. netadmin.uvic
.ca. 1504013426 10800 300 2592000 300
uvic.ca.                299     IN      NS      ns4.uvic.ca.
uvic.ca.                299     IN      NS      dns1.uvic.ca.
uvic.ca.                299     IN      NS      dns2.uvic.ca.
uvic.ca.                299     IN      A       142.104.197.120
uvic.ca.                299     IN      TXT     "MS=5CB2F7F278B331C49BA7700
D1ECDBDF3044C94C6"
uvic.ca.                299     IN      TXT     "google-site-verification=H
o7KjuRFxgLp5rCvp4SeC-E2zCcPTaqaYzwqQyyMgaY"
uvic.ca.                299     IN      LOC     48 28 1.200 N 123 19 8.400
W 62.00m 1m 10000m 10m
uvic.ca.                299     IN      MX      0 smtpr.uvic.ca.
uvic.ca.                299     IN      MX      0 smtps.uvic.ca.
uvic.ca.                299     IN      MX      0 smtpv.uvic.ca.
uvic.ca.                299     IN      MX      0 smtpw.uvic.ca.
uvic.ca.                299     IN      MX      0 smtpy.uvic.ca.
uvic.ca.                299     IN      MX      0 smtpz.uvic.ca.
uvic.ca.                299     IN      MX      10 smtpu.uvic.ca.
```

dig @a.root-servers.net www.uvic.ca

```
;; QUESTION SECTION:
;www.uvic.ca.                   IN      A

;; AUTHORITY SECTION:
ca.                     172800  IN      NS      any.ca-servers.ca.
ca.                     172800  IN      NS      d.ca-servers.ca.
ca.                     172800  IN      NS      c.ca-servers.ca.
ca.                     172800  IN      NS      j.ca-servers.ca.

;; ADDITIONAL SECTION:
any.ca-servers.ca.      172800  IN      A       199.4.144.2
any.ca-servers.ca.      172800  IN      AAAA    2001:500:a7::2
d.ca-servers.ca.        172800  IN      A       199.19.4.1
d.ca-servers.ca.        172800  IN      AAAA    2001:500:97::1
c.ca-servers.ca.        172800  IN      A       185.159.196.2
c.ca-servers.ca.        172800  IN      AAAA    2620:10a:8053::2
j.ca-servers.ca.        172800  IN      A       198.182.167.1
j.ca-servers.ca.        172800  IN      AAAA    2001:500:83::1
```

dig @any.ca-servers.ca www.uvic.ca

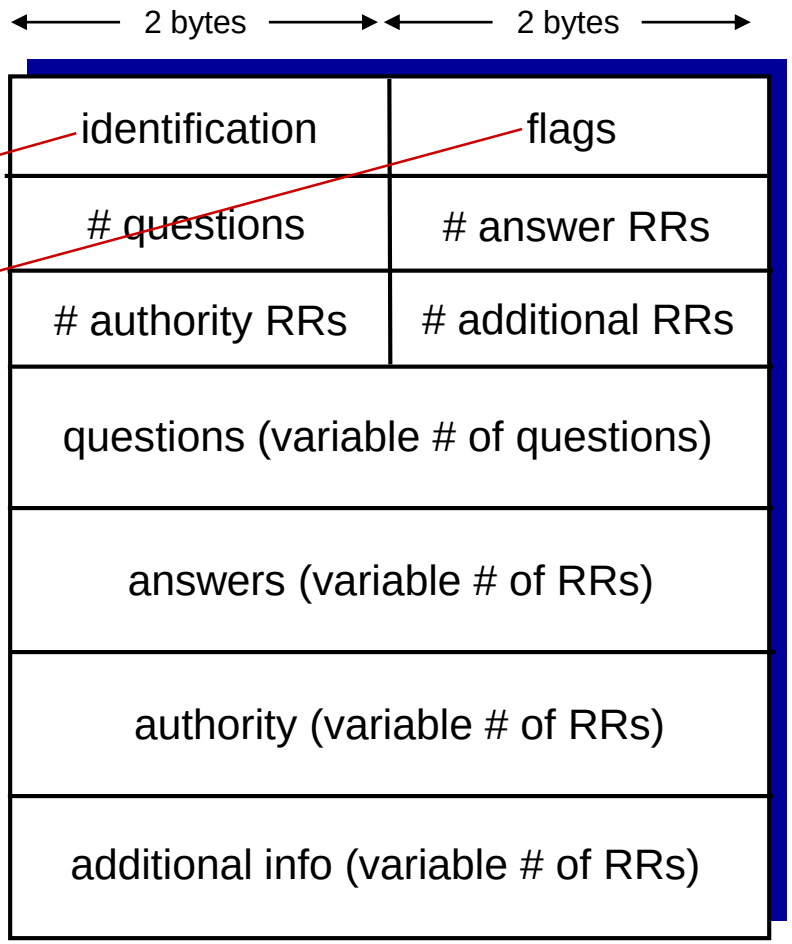
```
;; QUESTION SECTION:
;www.uvic.ca.                   IN      A

;; AUTHORITY SECTION:
uvic.ca.                86400   IN      NS      ns5.uvic.ca.
uvic.ca.                86400   IN      NS      ns4.uvic.ca.
uvic.ca.                86400   IN      NS      dns2.uvic.ca.
uvic.ca.                86400   IN      NS      dns1.uvic.ca.

;; ADDITIONAL SECTION:
ns4.uvic.ca.            86400   IN      A       206.12.58.50
ns5.uvic.ca.            86400   IN      A       206.12.58.52
dns1.uvic.ca.           86400   IN      A       142.104.6.1
dns2.uvic.ca.           86400   IN      A       142.104.80.2
```

DNS protocol, messages

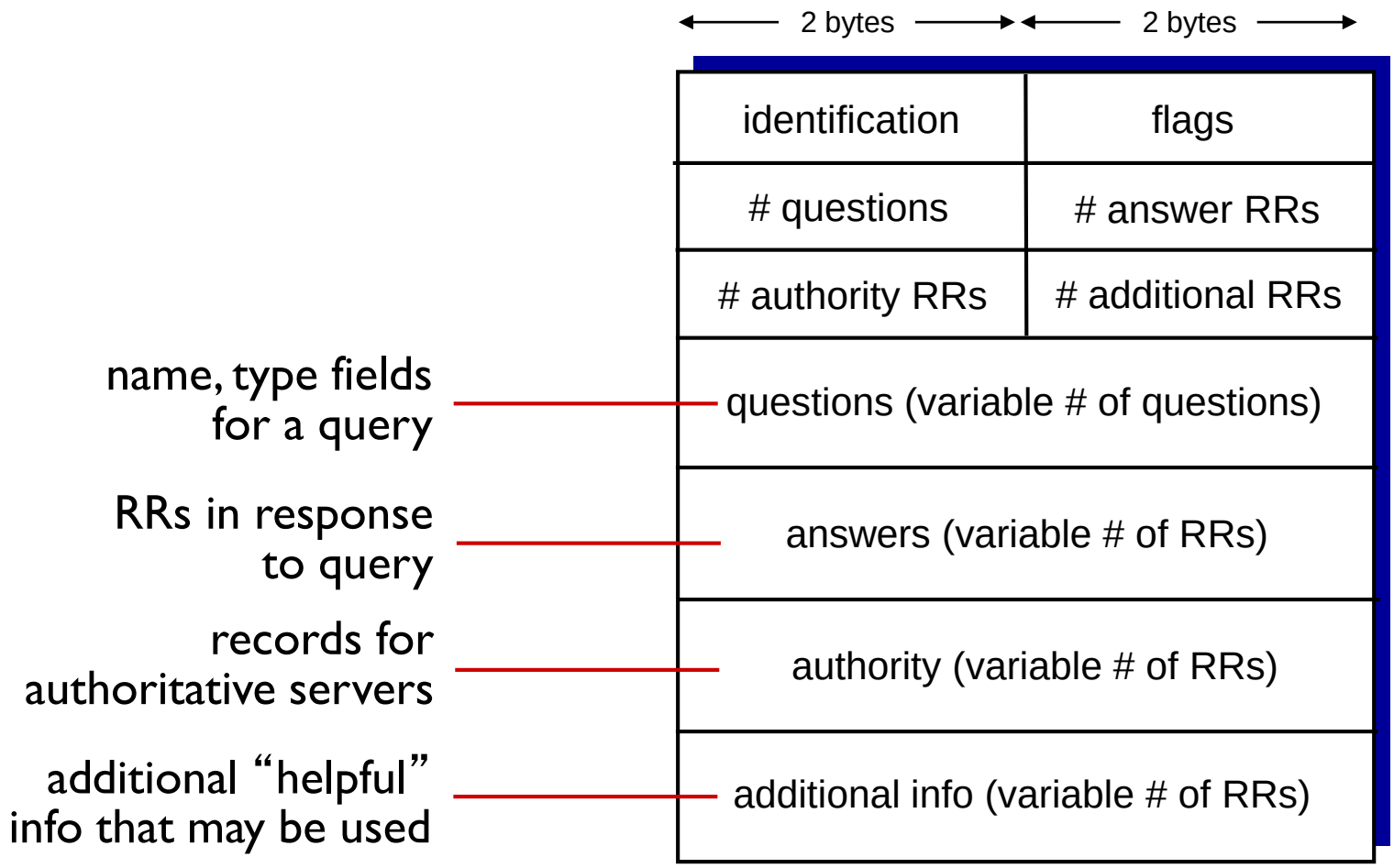
- *query* and *reply* messages, both use same *format*

message header

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



DNS protocol, messages



Inserting records into DNS

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

Attacking DNS

DDoS attacks

- bombard root servers with traffic
  - not successful to date
  - traffic filtering
  - local DNS servers cache IPs of TLD servers, allowing root server bypass
- bombard TLD servers
  - potentially more dangerous

redirect attacks

- man-in-middle
  - Intercept queries
- DNS poisoning
  - Send bogus replies to DNS server, which caches

exploit DNS for DDoS

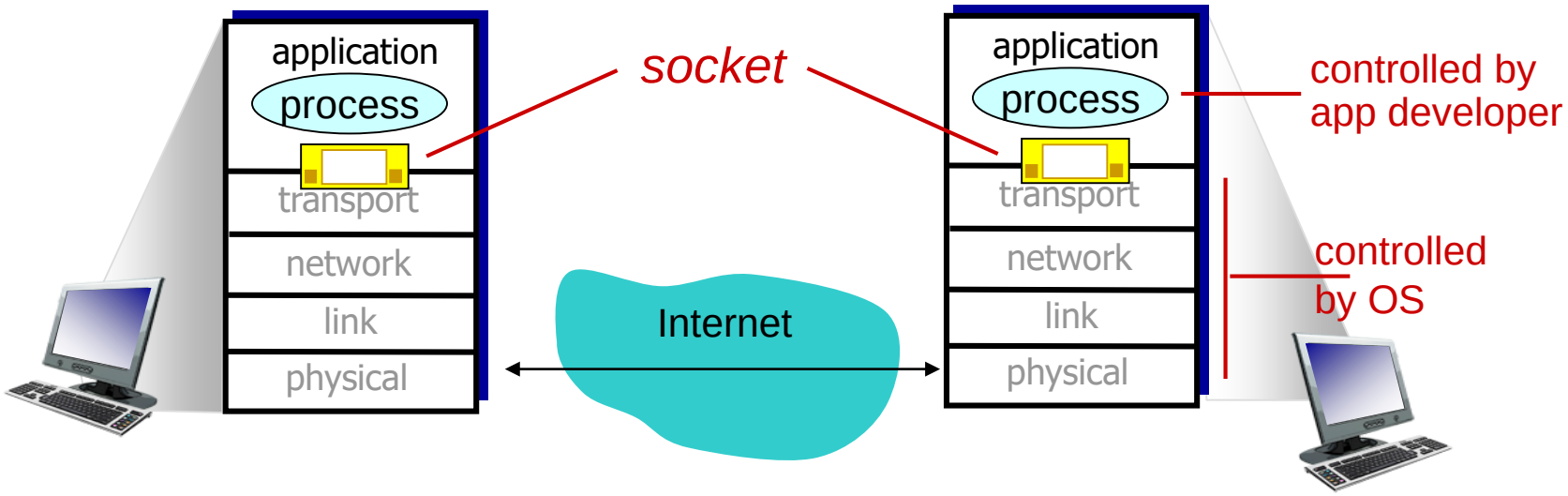
- send queries with spoofed source address: target IP
- requires amplification

BSD Unix Sockets

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable *datagram*
- *TCP:* reliable, byte *stream-oriented*

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Python Network Programming

Socket connection in Python

- Every machine/host has an IP address.
- Network applications/services have port #s.
- Some ports are well-known; some are transient.
- Each endpoint of a network communication is represented by an IP address and port #.
- In Python, you write the endpoint as:

  ("info.cern.ch", 80)
  (142.129.20.9, 4431)
- In BSD Unix, this is called a "socket".

Socket types

- There are two types of sockets: streams (TCP) and datagrams (UDP).
- Stream (TCP) socket transmits and receives in sequence of bytes with no framing, i.e., unstructured.
- SOCK_STREAM type denotes TCP streams.
- Datagram socket transmits and receives in well-defined structures called (UDP) datagrams.
- SOCK_DGRAM type denotes UDP datagrams.

UDP sockets in Python

- To create a UDP socket in Python:

```
from socket import *
s = socket(AF_INET, SOCK_DGRAM)
```

- To transmit a datagram message (size limited by the network or physical layer):

```
s.sendto("hello",(("server.com",9999))
data, r = s.recvfrom(2048)
```

- No connection is required!

Socket programming *with UDP*

UDP: no "connection" between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet (i.e., **sendto**)
- receiver extracts sender IP address and port# from received packet n (i.e., **recvfrom**)

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

Application viewpoint:

- UDP provides *unreliable* transfer of "datagrams" between client and server
- each datagram is a "self-contained" unit; **sendto**() and **recvfrom**() operate on each unit independently

TCP sockets in Python (client)

- To create a TCP socket in Python:

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
```

- To transmit a byte stream, establish a connection first:

```
s.connect((”server.com”,9999))
s.send("GET /index.html HTTP/1.0\n\n")
data = s.recv(10000)
s.close()
```

TCP sockets in Python (server)

- To receive from a TCP socket in Python:

```
s = socket(AF_INET,SOCK_STREAM)

s.bind(“”,9999)

s.listen(2)

r, addr= s.accept()

r.send(“response”)
```

accepting connection at port 9999

ready to accept connections

wait for a connection

send back a response

sender’s remote address (IP, port #)

Socket programming *with TCP*

client must contact server

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

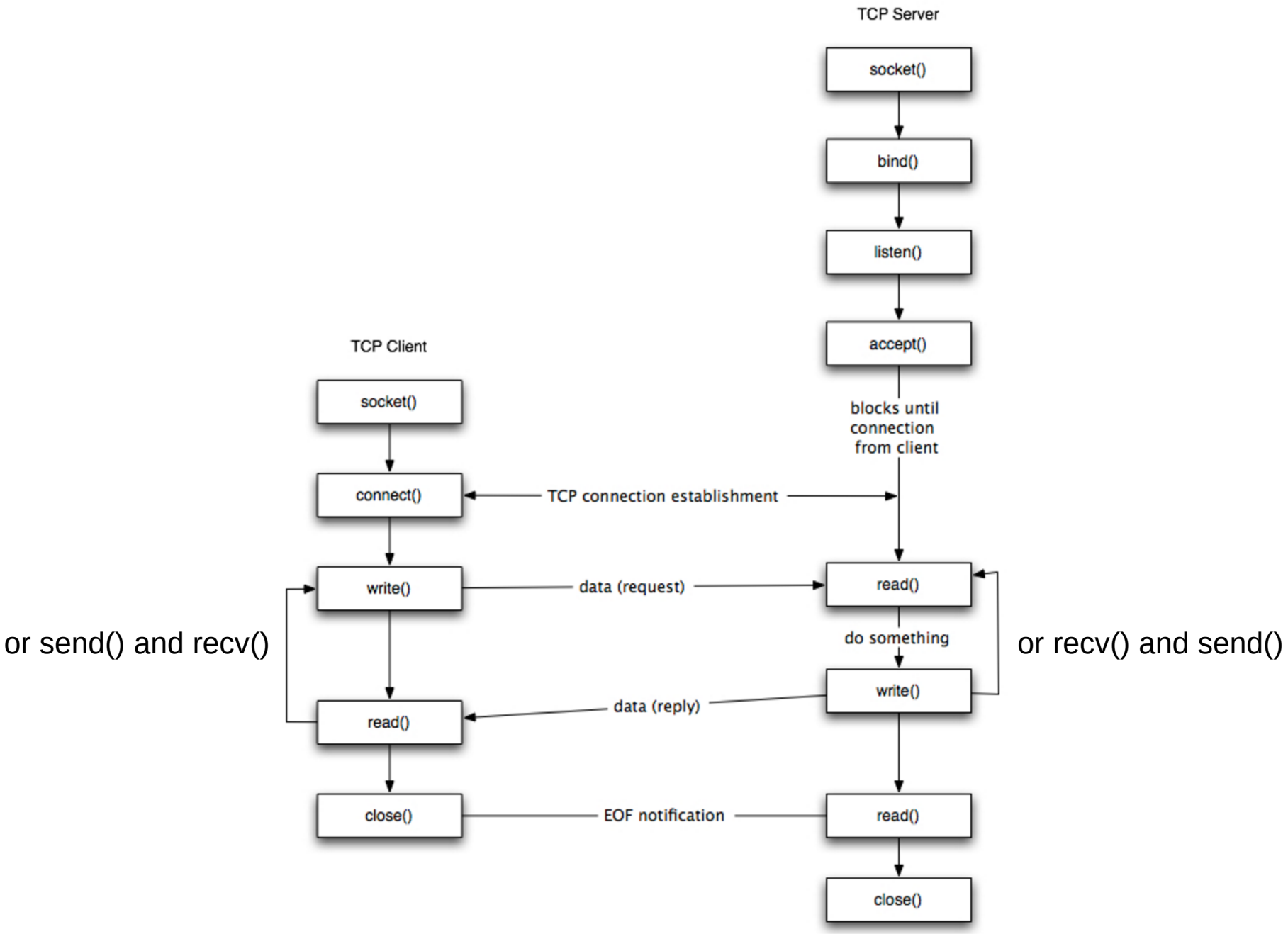
client contacts server by:

- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates **new** socket* for server process to communicate with that **particular** client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

TCP provides reliable, in-order byte-stream transfer between client and server


TCP Server
socket()
bind()
listen()
accept()
blocks until connection from client
TCP Client
socket()
connect()
TCP connection establishment
write()
data (request)
read()
do something
or send() and recv()
or recv() and send()
write()
data (reply)
read()
close()
EOF notification
read()
close()