# Application Layer

# Application layer

our goals:

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

# Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

2.4 DNS

2.5 P2P applications

2.7 socket programming with UDP and TCP

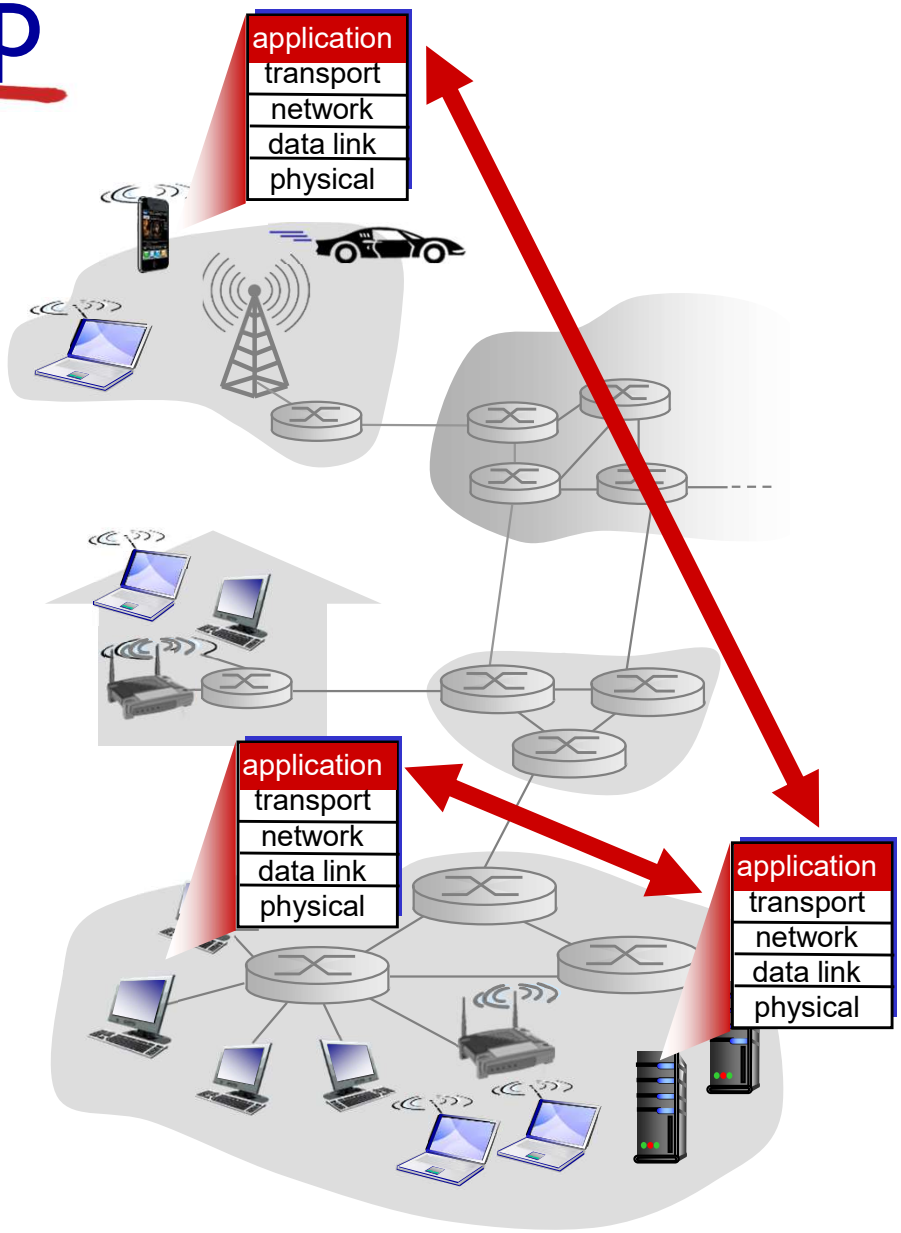
# Some network apps

- 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

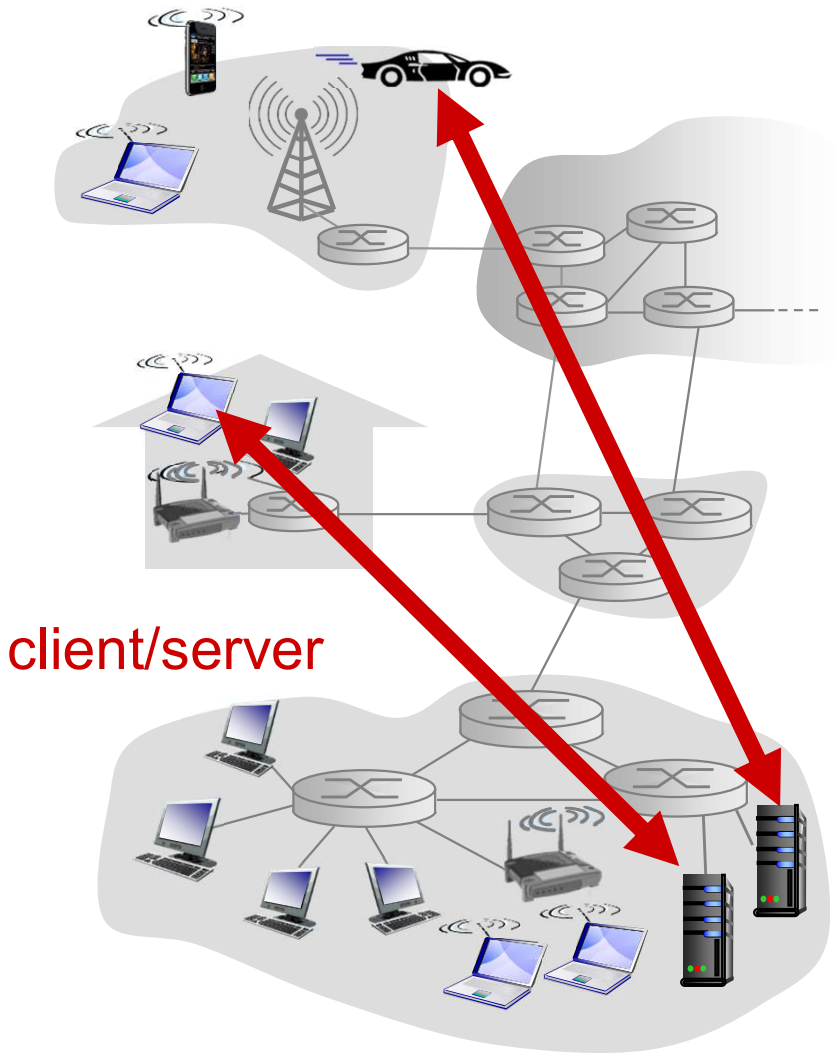
- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

# Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P) ex: bittorrent

# Client-server architecture




server:

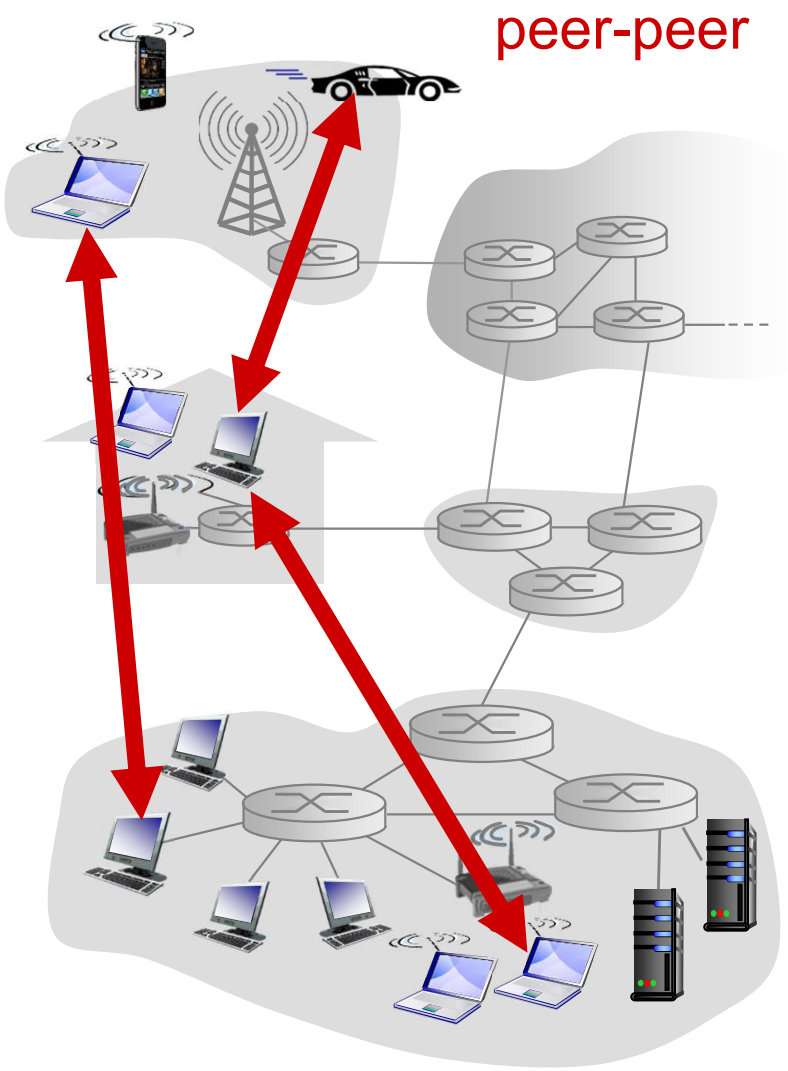
- always-on host
- permanent IP address
- data centers for scaling

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



# 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

can make laptop web server by installing iis

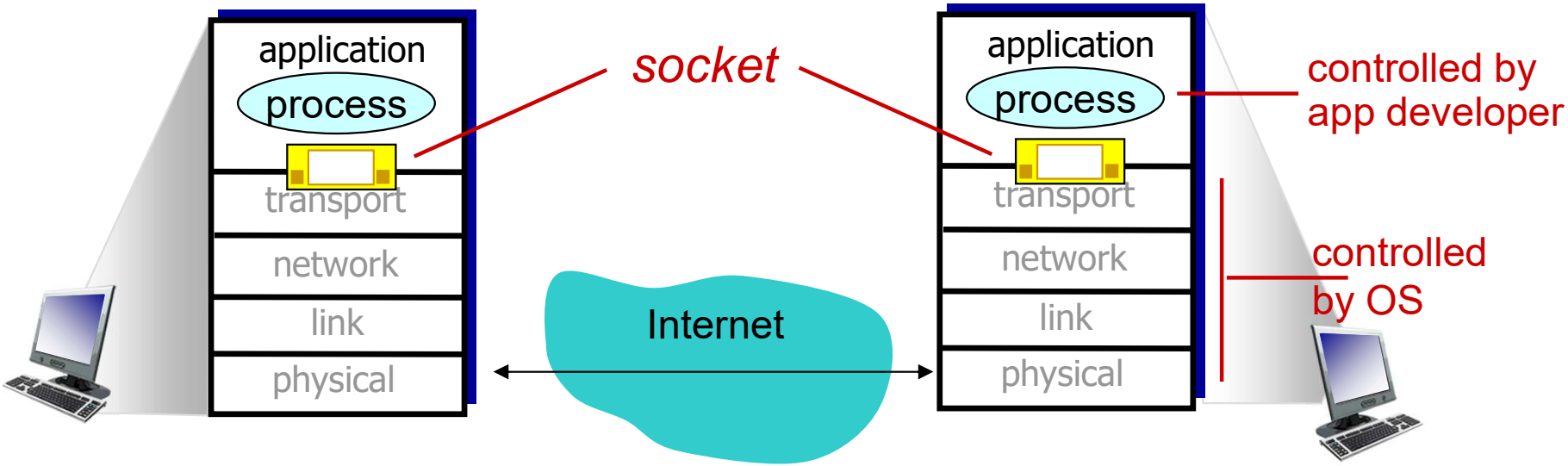
for linux install apache

# Sockets

need address of machine and address of process to send message
<IP, port#>
the pair is called the socket

port number is address of process

- 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

well known ports are reserved
port is 16 bit, ip 32 bit
only use ports larger that 1024

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25

https: 443

- to send HTTP message to stmarytx.edu web server:
  - IP address: 54.175.71.212
  - 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?

## reliable data Transfer

- 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, …

TCP - reliable, use if need reliable transmission, connection oriented protocol, flow control, provides acknowledgement
http is protocol used by web browser, apps need reliable data transfer so they will use tcp
flow control - sender and receiver synchronize so that nothing is dropped

UDP - not so reliable, no flow control, no connection oriented, no acknowledgement, but FAST

# 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<br>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<br>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, or connection setup,

Q: why bother? Why is there a UDP?

# Internet apps: application, transport protocols

| application | application layer protocol | transport protocol |
|---|---|---|
| file transfer/download | FTP [RFC 959] | TCP |
| e-mail | SMTP [RFC 5321] | TCP |
| Web documents | HTTP 1.1 [RFC 7320] | TCP |
| Internet telephony | SIP [RFC 3261], RTP [RFC 3550], or proprietary | TCP or UDP |
| streaming audio/video | HTTP [RFC 7320], DASH | TCP |
| interactive games | WOW, FPS (proprietary) | UDP or TCP |

neither is secure

ssl - secure socket layer NOT A LAYER IN THE TCP/IP PROTOCOL

it enhances TCP & UDP

# Securing TCP

## TCP & UDP

- no encryption
- clear text passwords 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

- clear text passwords sent into socket traverse Internet encrypted

# Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

2.4 DNS

2.5 P2P applications

2.7 socket programming with UDP and TCP

# Web and HTTP

rendering - your browser renders the file sent by the server

*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

# 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 connections

*non-persistent HTTP*

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

*persistent HTTP*

- 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

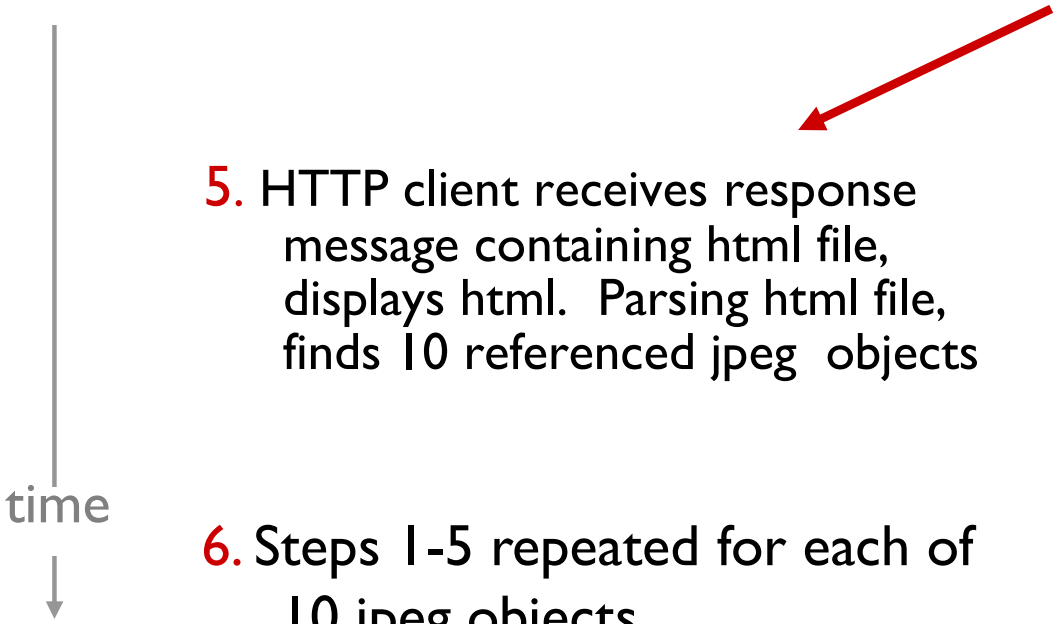
1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

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

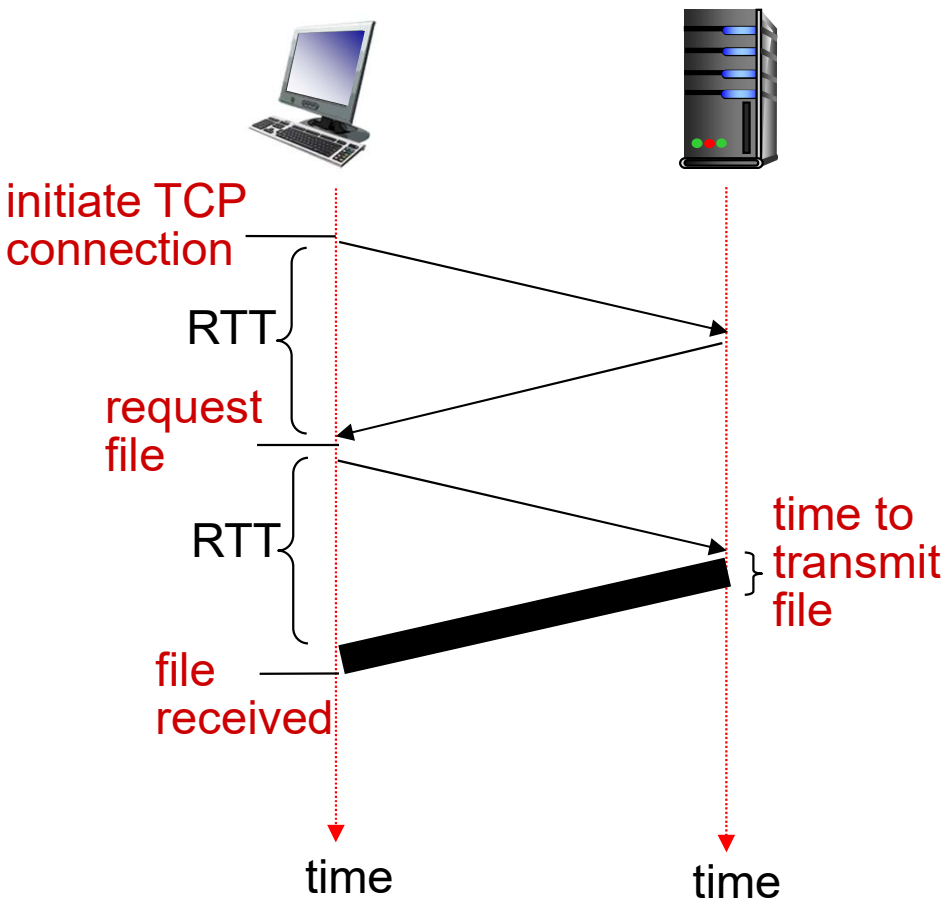
# Non-persistent HTTP: response time

RTT (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



# Persistent HTTP

*non-persistent HTTP issues:*

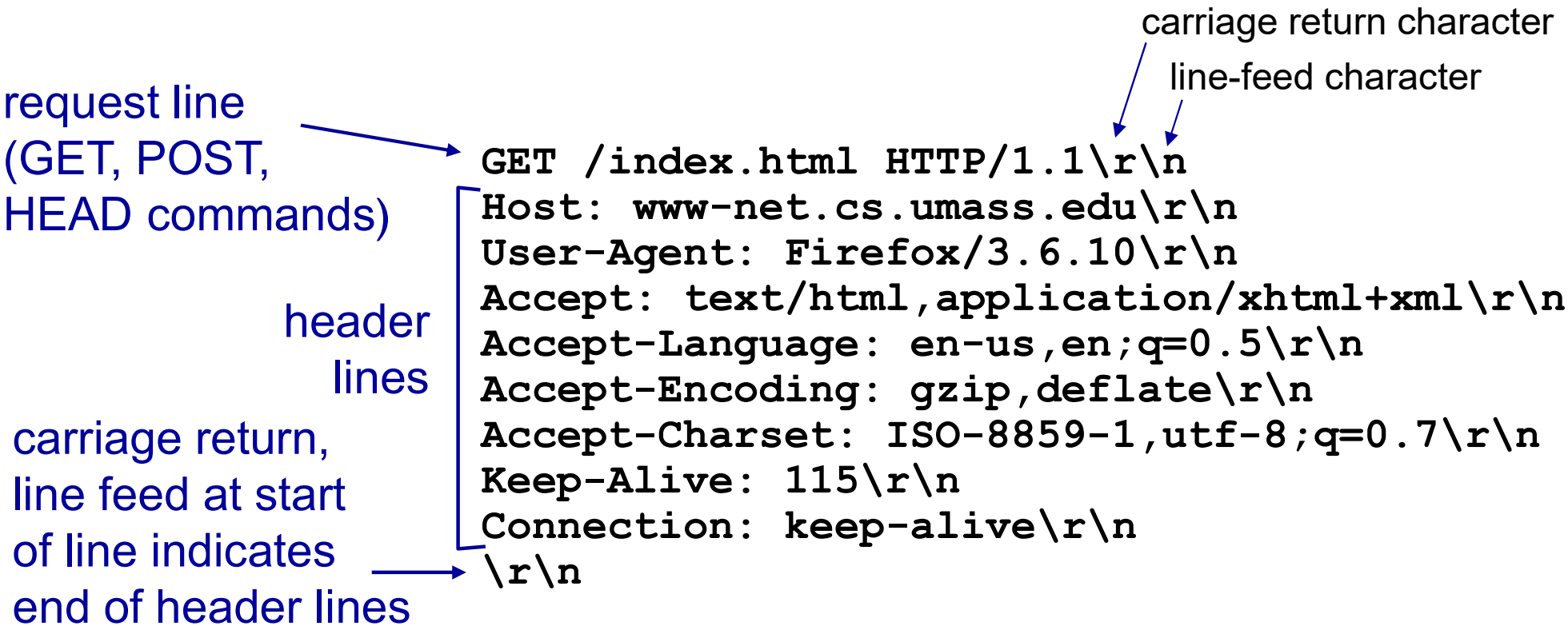
- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

*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

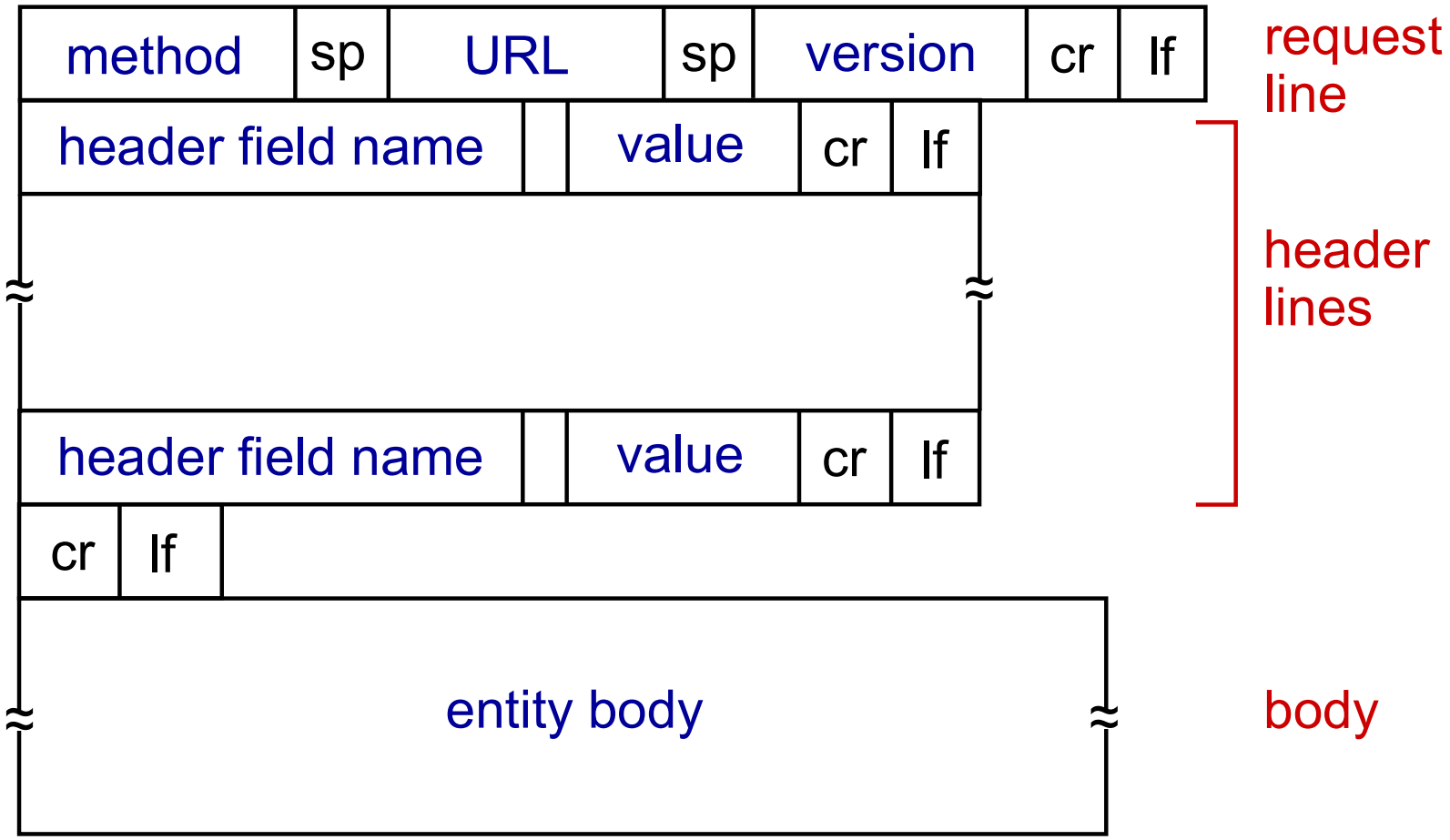
# HTTP request message

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




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

# 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

# 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**

# 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 Web sites use cookies

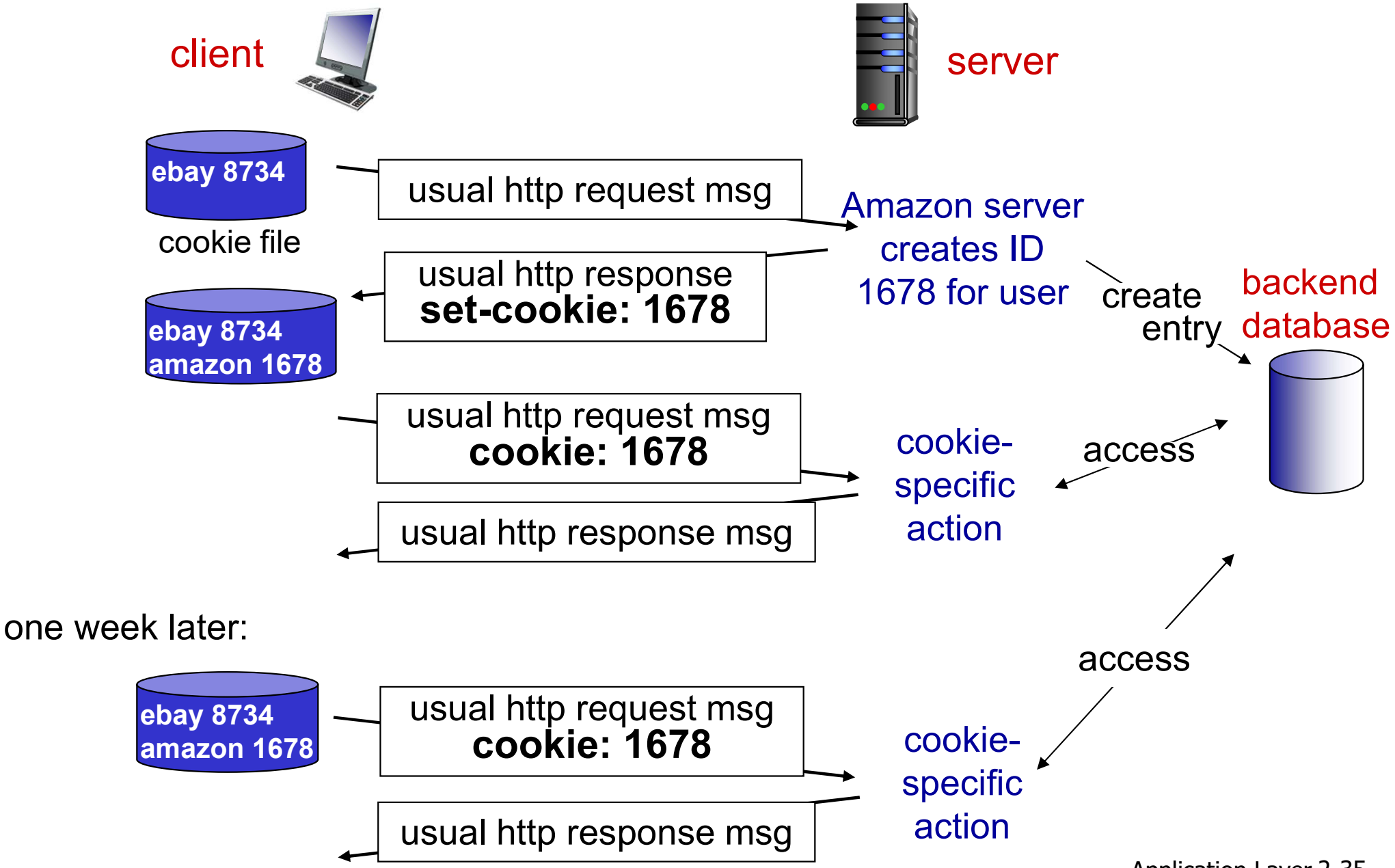
*four components:*

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

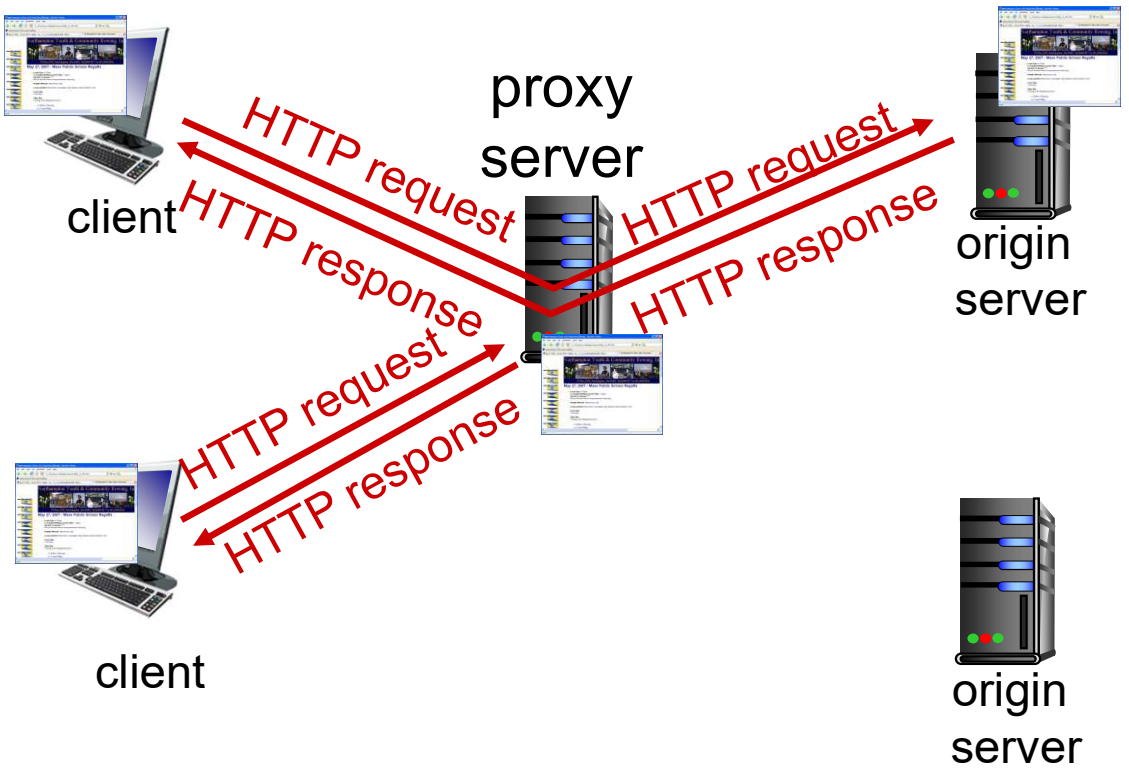
*how to keep "state":*

- 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?*

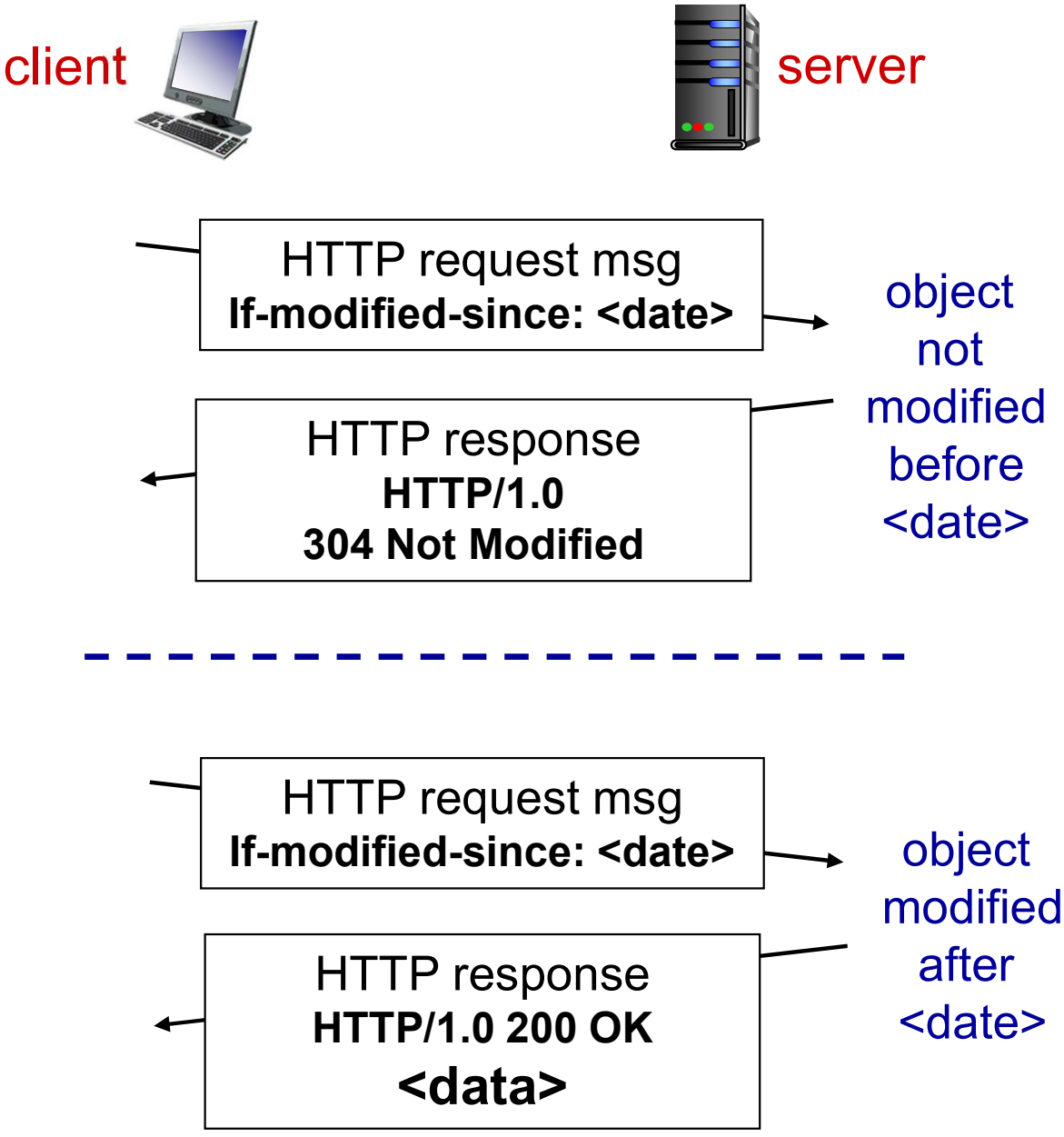
- reduce response time for client request
- reduce traffic on an institution's access link
- Internet dense with caches: enables "poor" content providers to effectively deliver content (so too does P2P file sharing)

# 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`



# HTTP/2

*Key goal:* decreased delay in multi-object HTTP requests

*HTTP1.1:* introduced multiple, pipelined GETs over single TCP connection

- server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- with FCFS, small object may have to wait for transmission (head-of-line (HOL) blocking) behind large object(s)
- loss recovery (retransmitting lost TCP segments) stalls object transmission

# HTTP/2

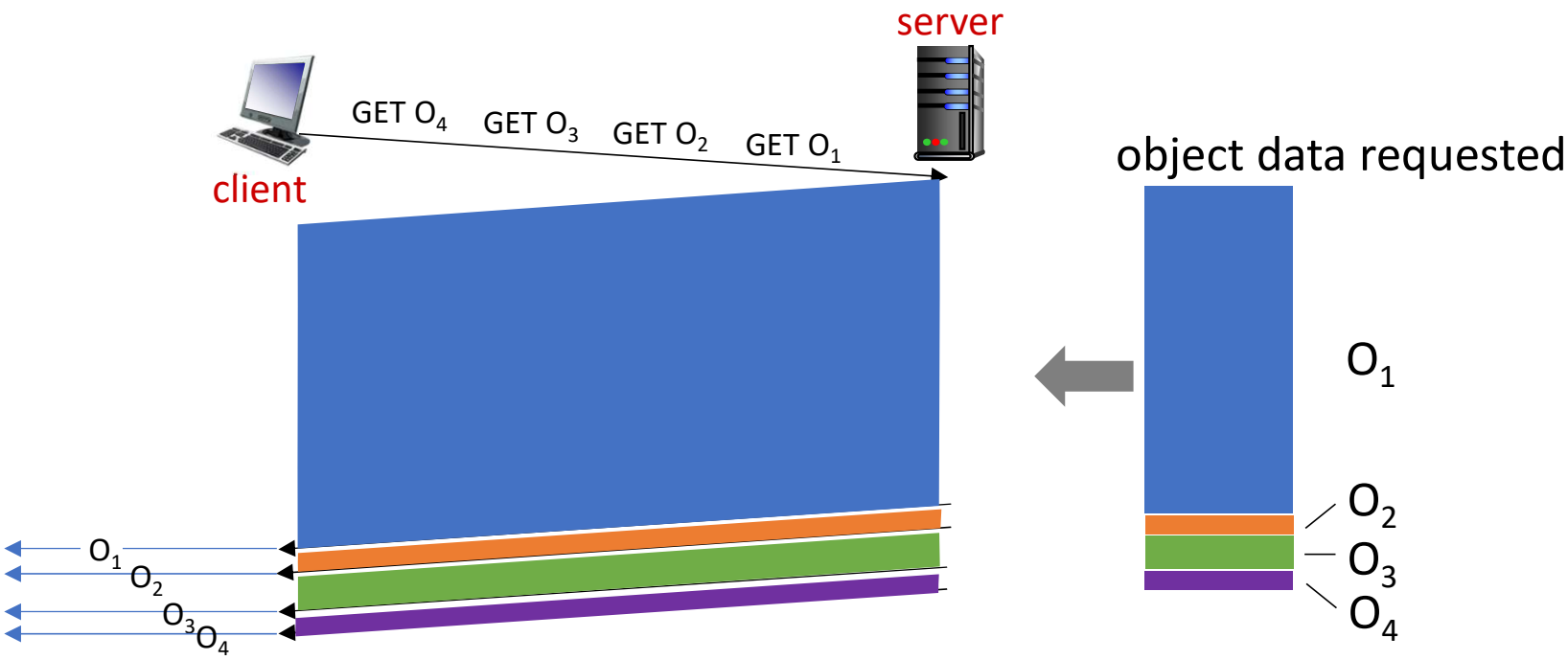
*Key goal:* decreased delay in multi-object HTTP requests

*HTTP/2:* [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:

- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking

# HTTP/2: mitigating HOL blocking

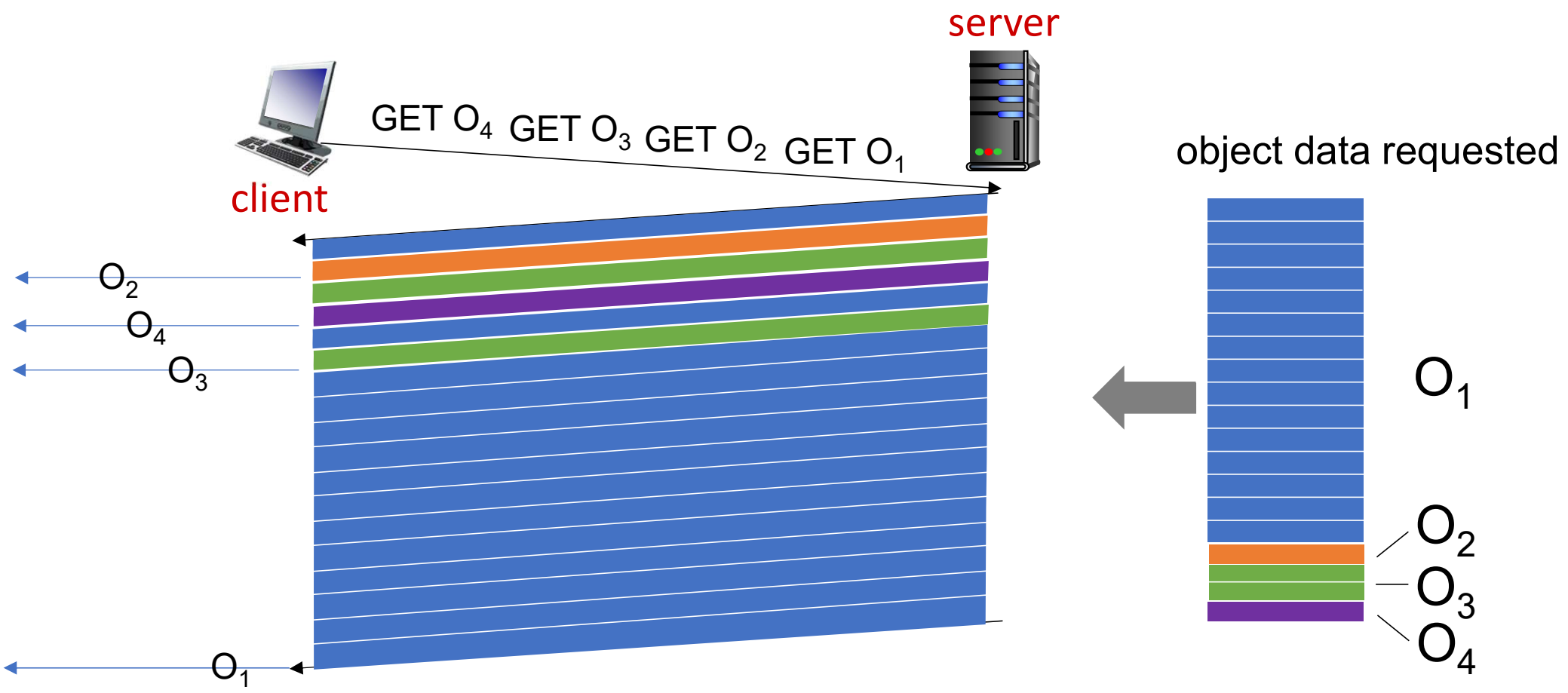
HTTP 1.1: client requests 1 large object (e.g., video file, and 3 smaller objects)




*objects delivered in order requested: $O_2$, $O_3$, $O_4$ wait behind $O_1$*

# HTTP/2: mitigating HOL blocking

HTTP/2: objects divided into frames, frame transmission interleaved




*$O_2$, $O_3$, $O_4$ delivered quickly, $O_1$ slightly delayed*

# HTTP/2 to HTTP/3

*Key goal:* decreased delay in multi-object HTTP requests

HTTP/2 over single TCP connection means:

- recovery from packet loss still stalls all object transmissions
  - as in HTTP 1.1, browsers have incentive to open multiple parallel TCP connections to reduce stalling, increase overall throughput
- no security over vanilla TCP connection
- HTTP/3: adds security , per object error- and congestion-control (more pipelining) over UDP
  - more on HTTP/3 in transport layer

# Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

2.4 DNS

2.5 P2P applications

2.7 socket programming with UDP and TCP

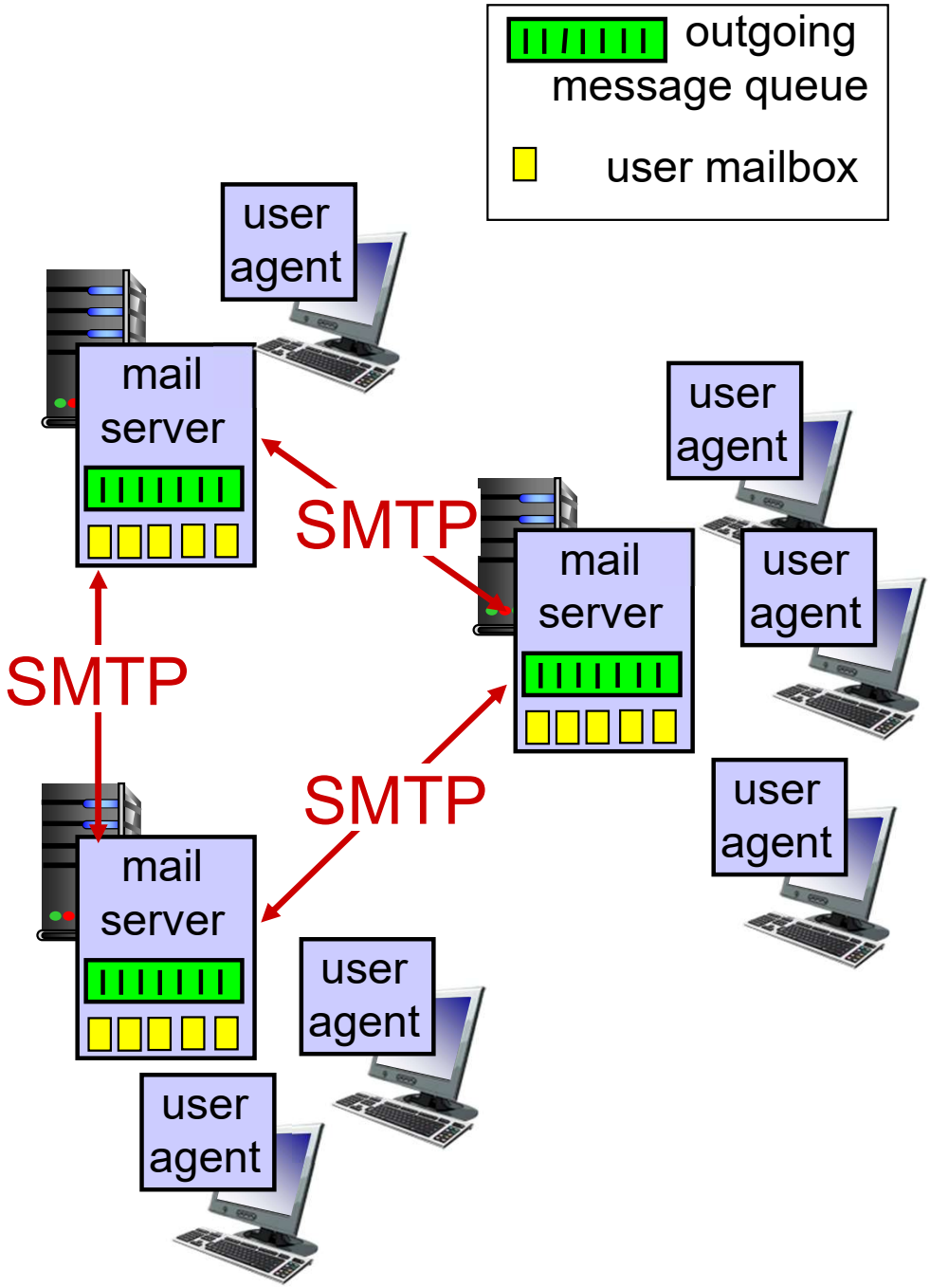
# Electronic mail

*Three major components:*

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

*User Agent*

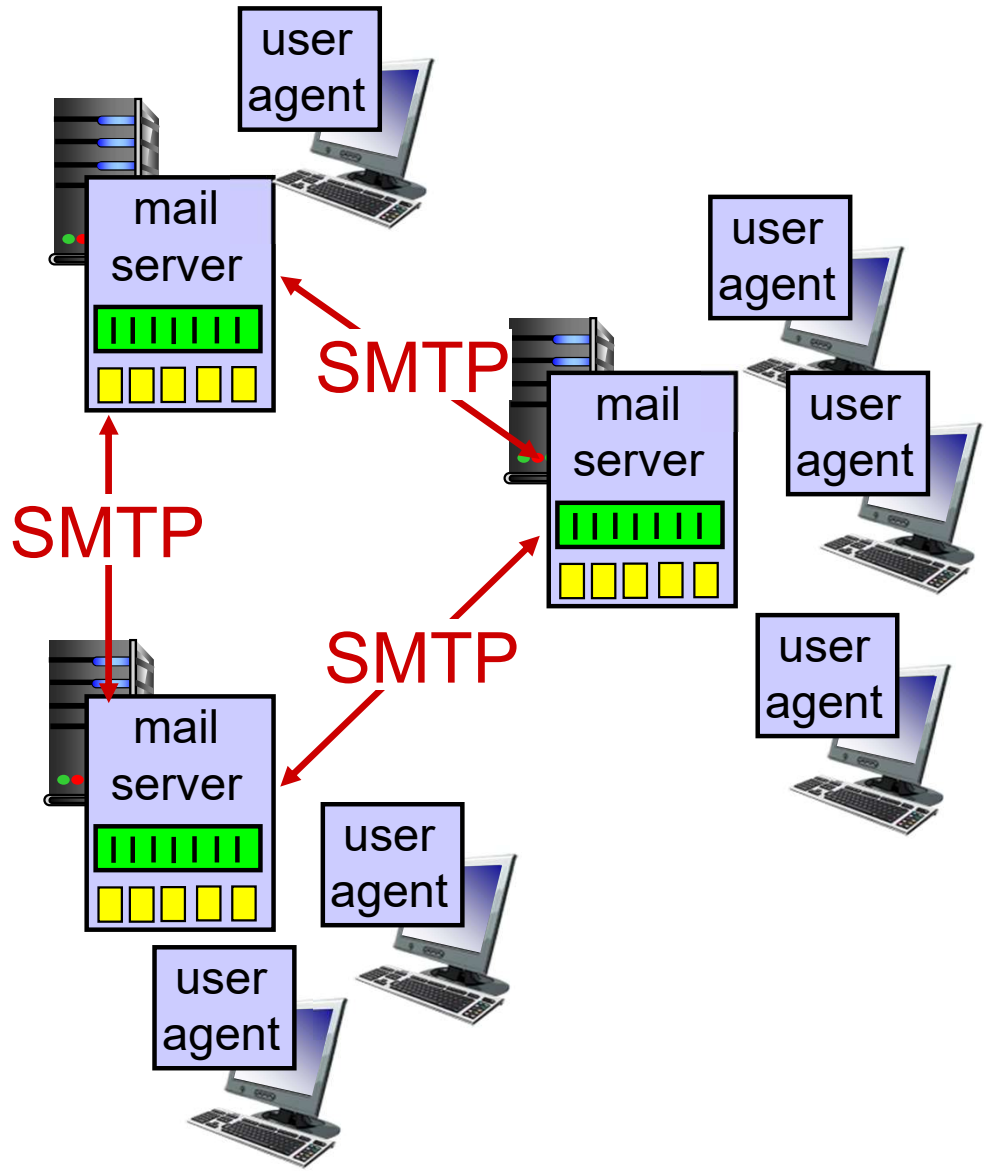
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



# 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

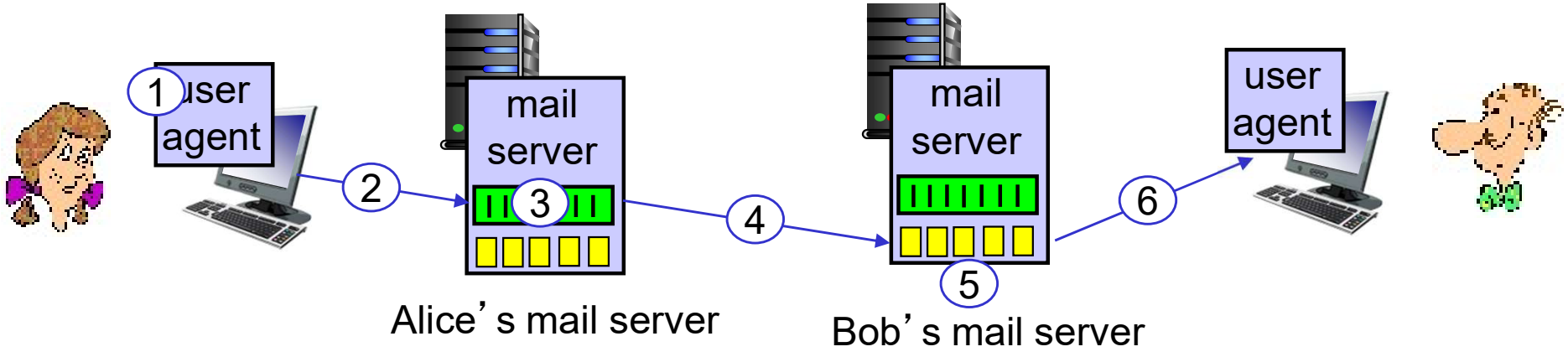


# Electronic Mail: SMTP

- 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 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



# Sample SMTP interaction

```
C: Telnet <Mail Server> 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
```

# 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 message
- SMTP: multiple objects sent in multipart message

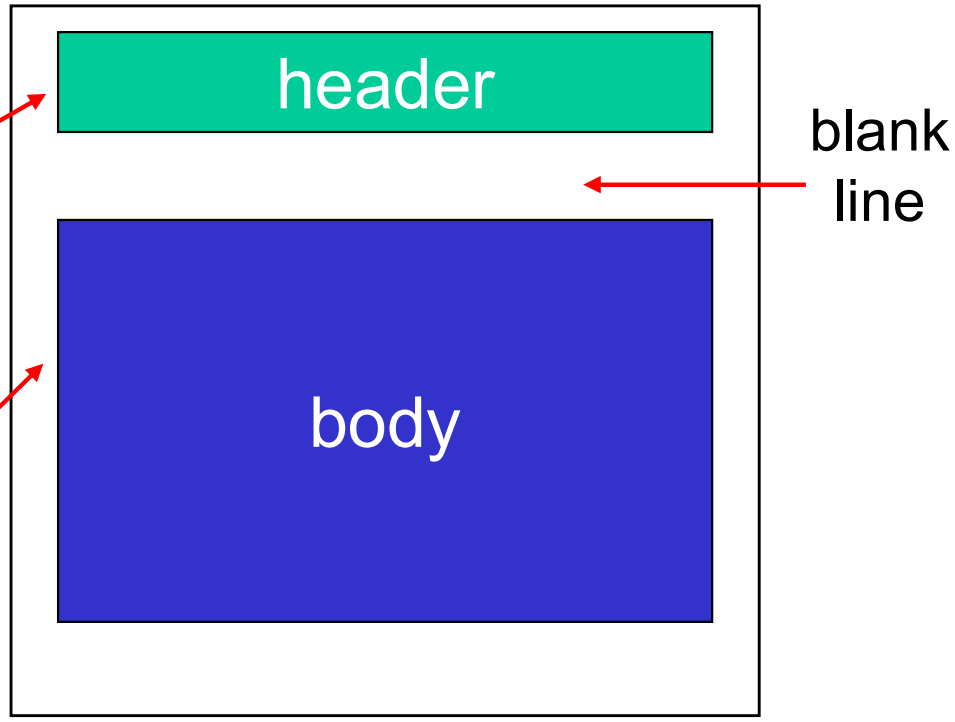
# Mail message format

SMTP: protocol for exchanging email messages

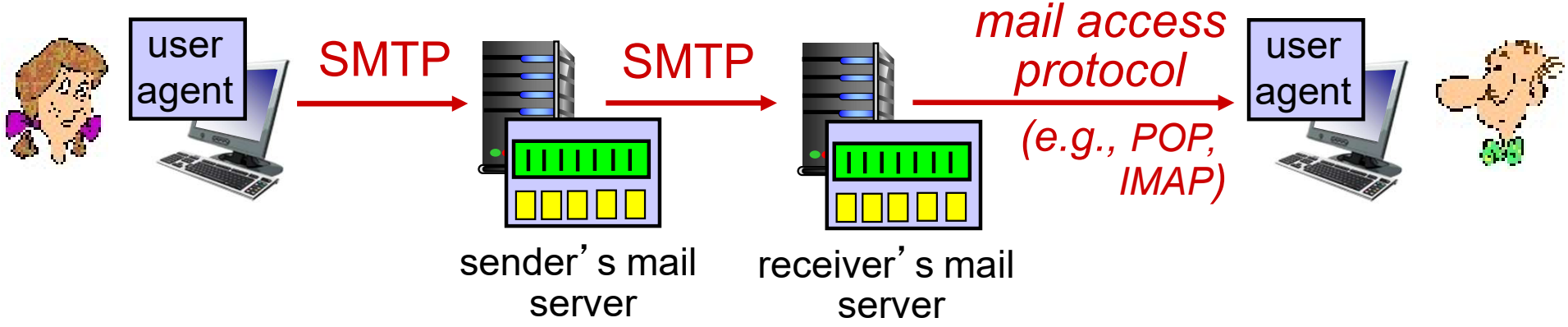
RFC 822: standard for text message format:

- 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
  - IMAP: Internet Mail Access Protocol [RFC 3501]: messages stored on server, IMAP provides retrieval, deletion, folders of stored messages on server
  - HTTP: gmail, Hotmail, Yahoo!Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

# Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

2.4 DNS

2.5 P2P applications

2.7 socket programming with UDP and TCP

# 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*

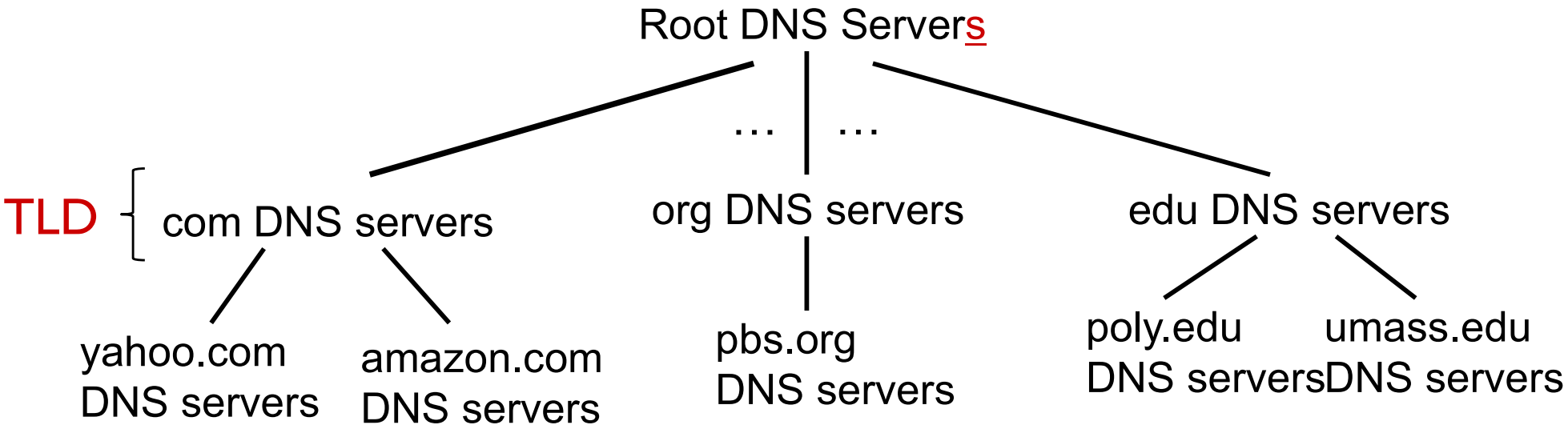
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

*why not centralize DNS?*

- single point of failure
- traffic volume
- distant centralized database
- maintenance
  - frequent update for every new host.

*A: doesn't scale!*

# DNS: a distributed, hierarchical database

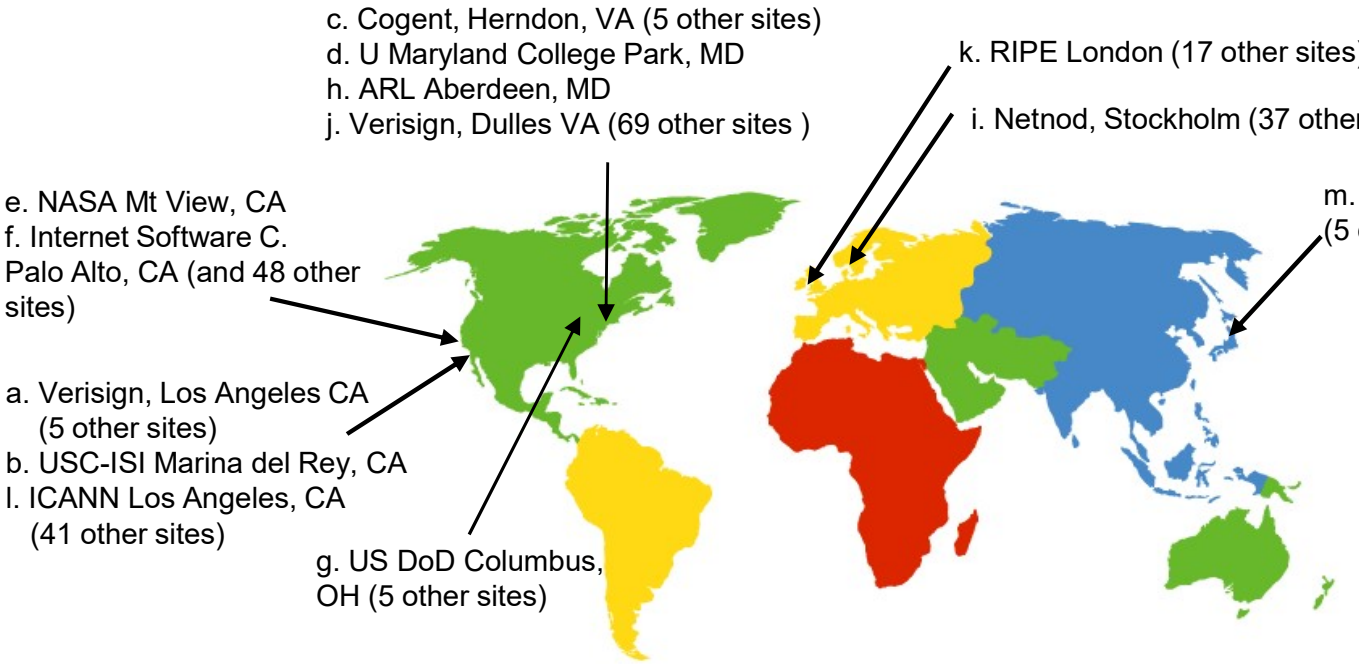



*client wants IP for www.amazon.com; 1st approximation:*

- client queries one of the root servers to find com DNS server
  - returns IP addresses for TLD (top-level domain) servers
- client queries one of the TLD servers (.com DNS server)
  - returns the IP address of an authoritative server for amazon.com
- 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




*13 logical root name "servers" worldwide*

- *each "server" replicated many times*

# 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

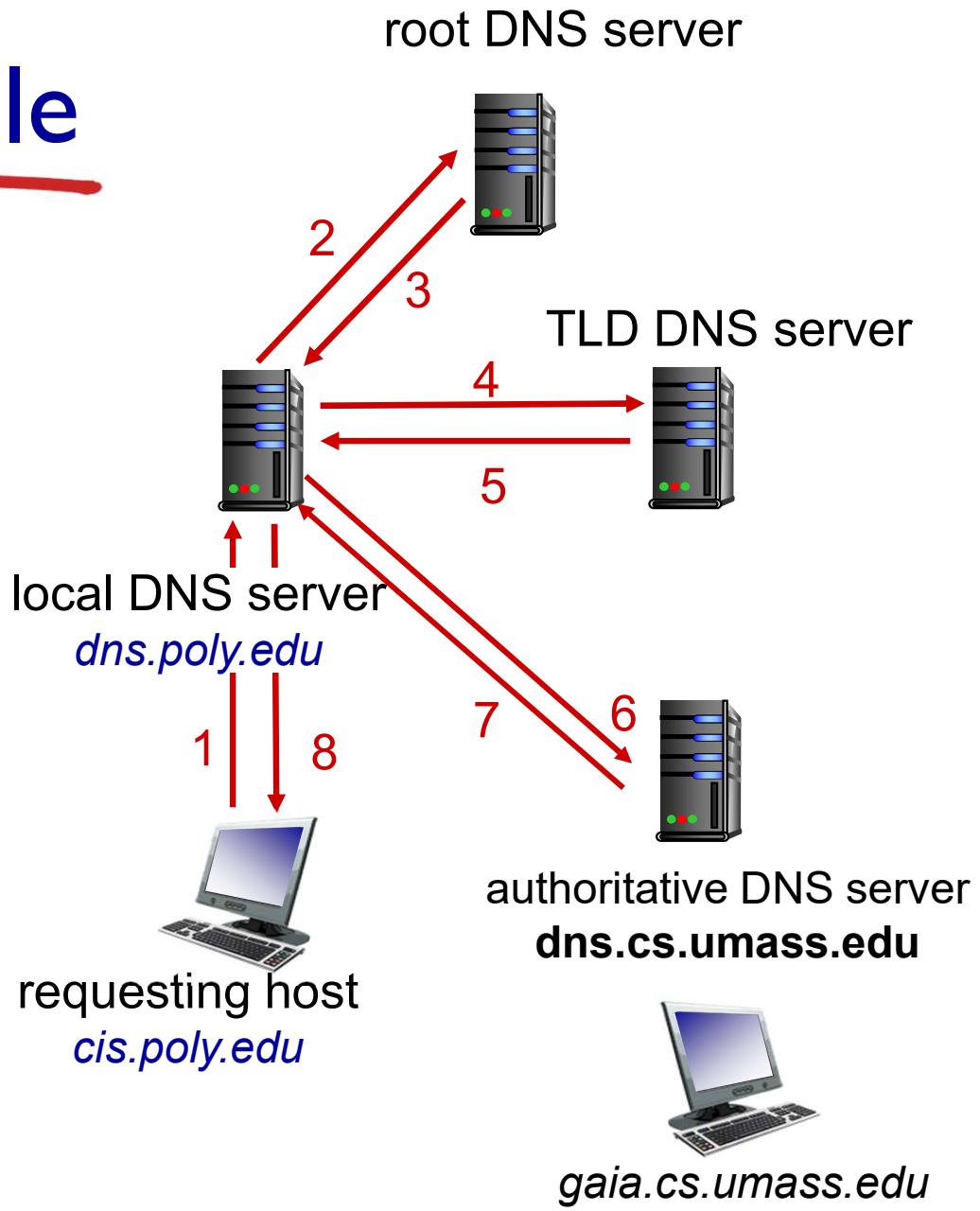
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts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

# DNS name resolution example

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

*iterative query:*

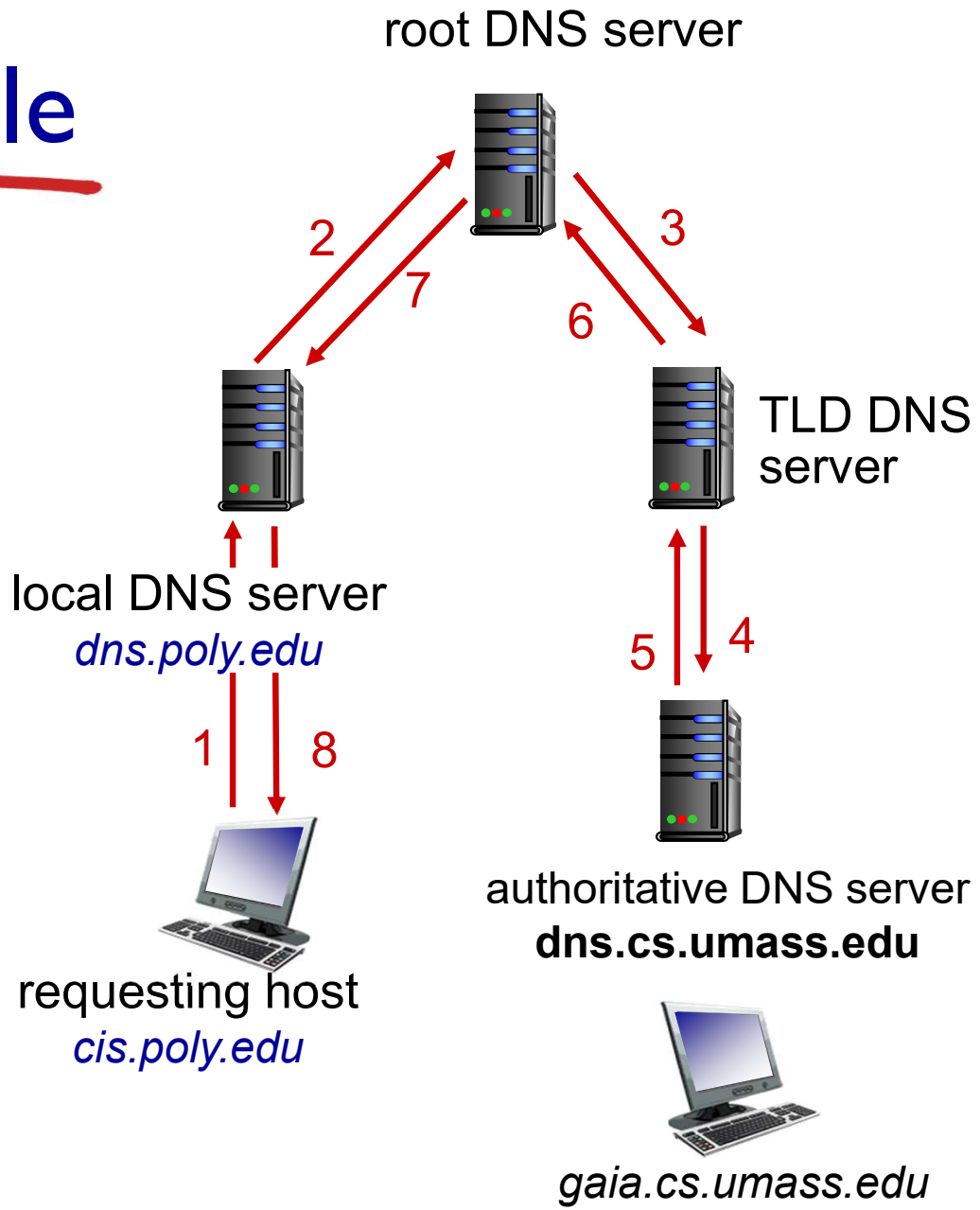
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



# 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 (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 is decentralized

# DNS records

*DNS:* distributed database storing resource records (RR)

RR format: `(name, value, type, ttl)`
time to live

## type=A

- **`name`** is hostname
- **`value`** is IP address
- (relay1.bar.foo.com, 145.37.93.126, A)

## type=NS

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

## 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
- (foo.com, relay1.bar.foo.com, CNAME)

## type=MX

- **`value`** is name of mailserver associated with **`name`**
- foo.com, mail.bar.foo.com, MX)

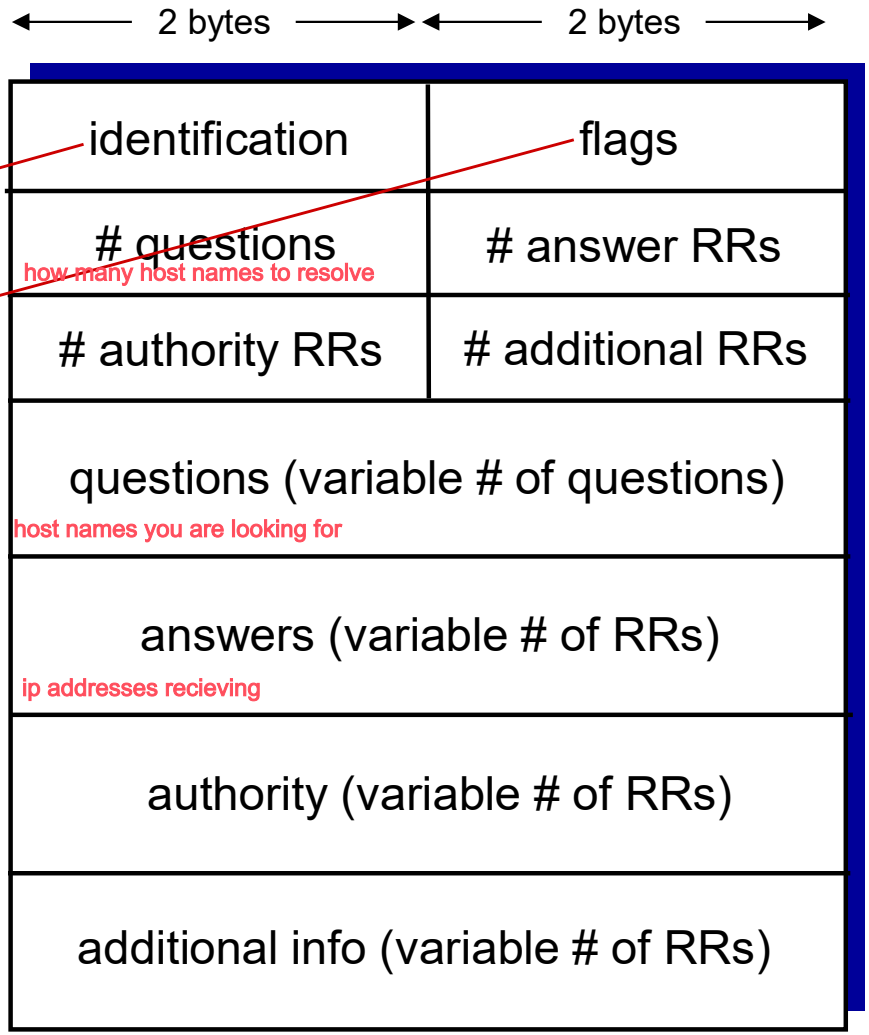
linux: dig - give url and it will give host name

# DNS protocol, messages

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

message header

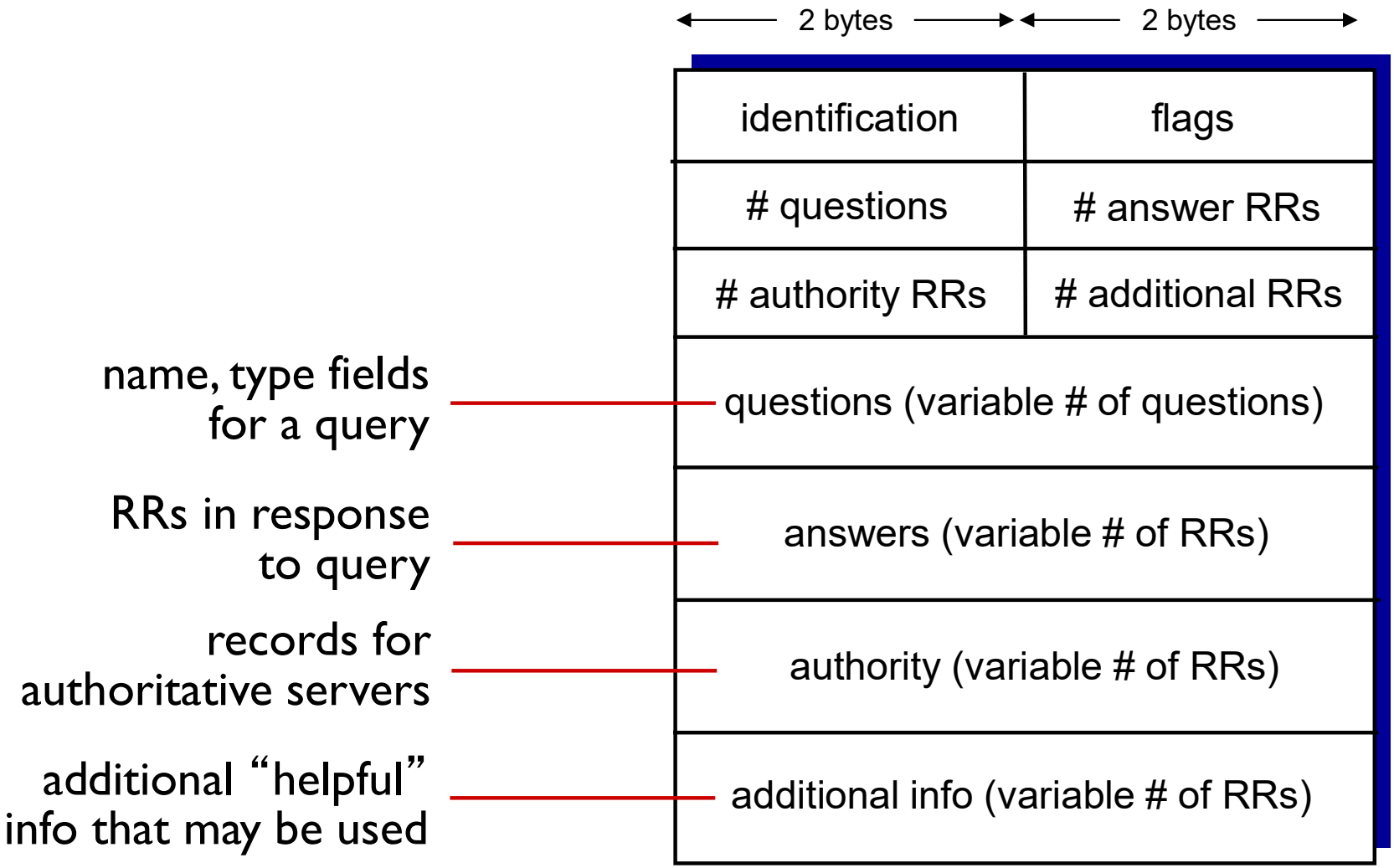
- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative




recursive or iterative can be set through the flag

computer that will host website is web server: www.cybermary.com ip 1.1.1.1
computer for authoritative dns: dns.cybermary.com ip 2.2.2.2, inside of here contains:
www.cybermary,com 1.1.1.1 A, and www.cybermary.com mail.cybermary.com MX, and mail.cybermary.com 3.3.3.3 A
computer for mail server: mail.cybermary.com ip 3.3.3.3, insider of here contains folders for employees
to get into dns tree, go to tld company and pay to be put in the tree (company such as network solutions), need from you
(1) www.cybermary.com dns.cybermary.com NS, (2) dns.cybermary.com 2.2.2.2 A

# DNS protocol, messages



# 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 (these entries are inside the authoritative DNS server `dns1.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 relies to DNS server, which caches

## exploit DNS for DDoS

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

# Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

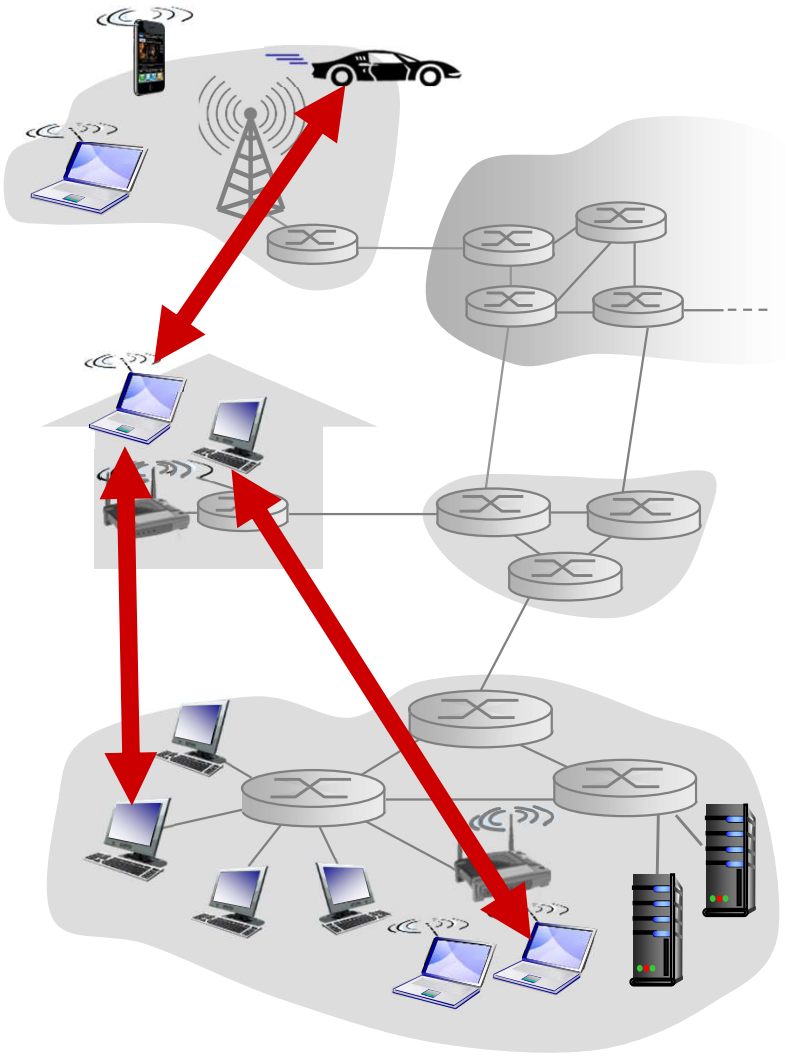
2.4 DNS

2.5 P2P applications

2.7 socket programming with UDP and TCP

# Pure P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses



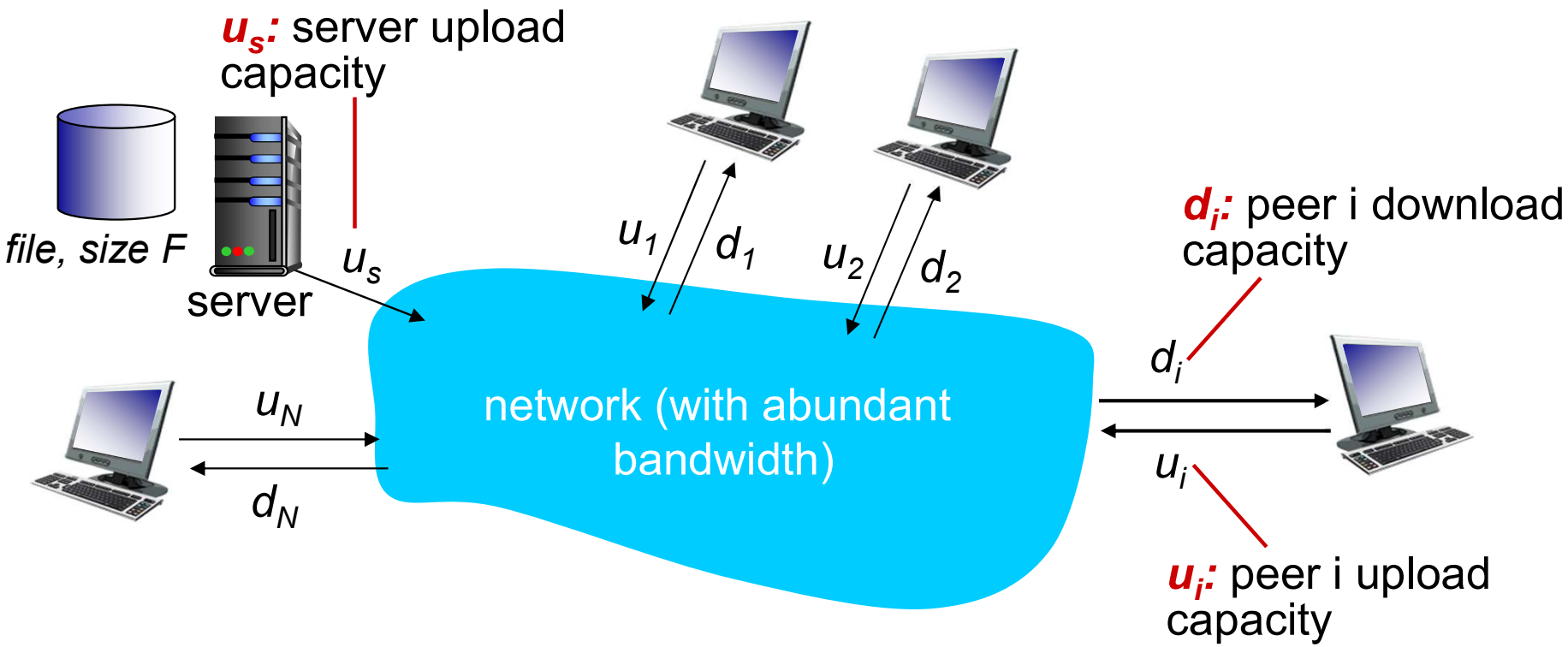
*examples:*

- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)

# File distribution: client-server vs P2P

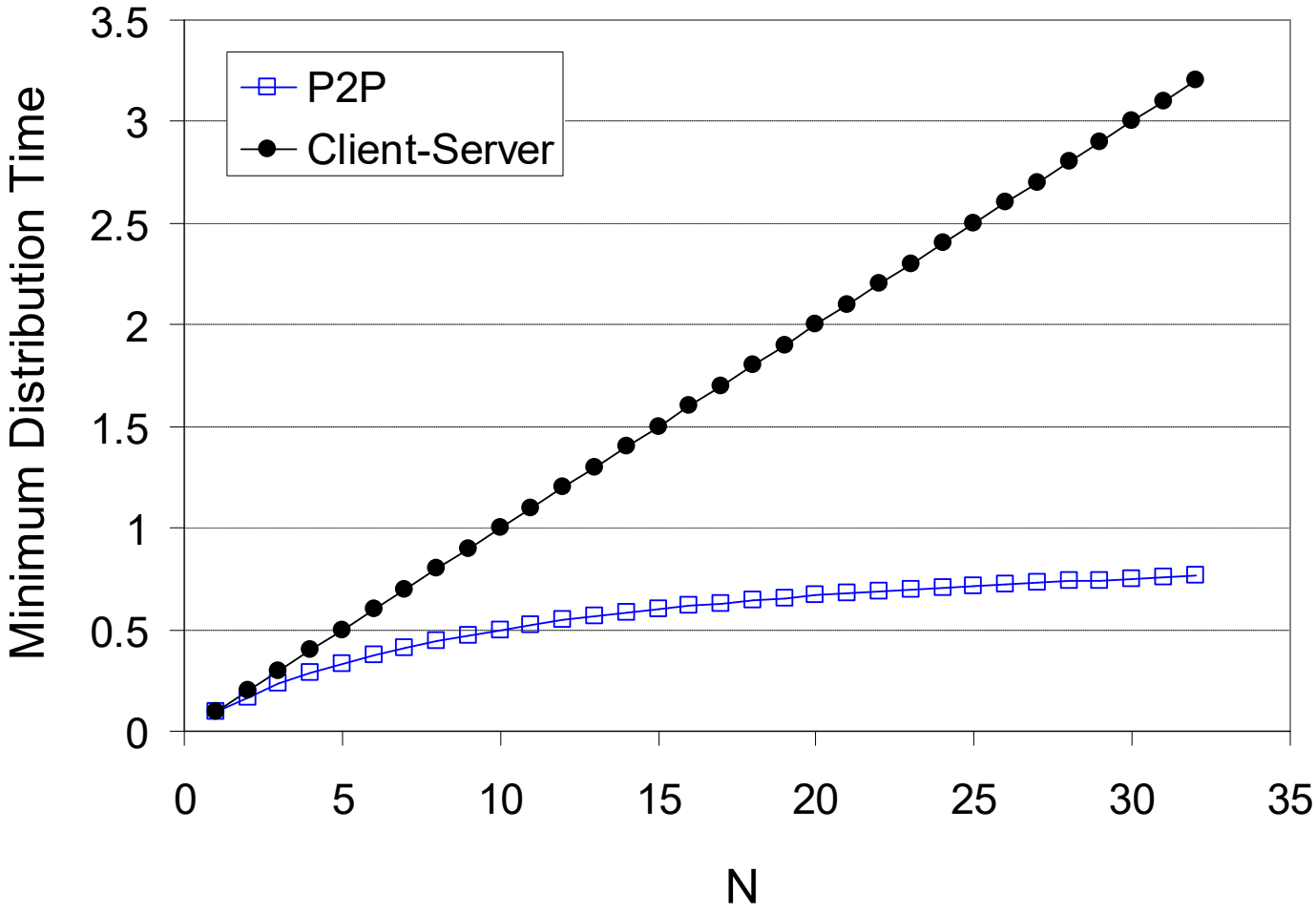
*Question:* how much time to distribute file (size *F*) from one server to *N* peers?

- peer upload/download capacity is limited resource



# Client-server vs. P2P: example

client upload rate = $u$, $F/u$ = 1 hour, $u_s = 10u$, $d_{min} \geq u_s$

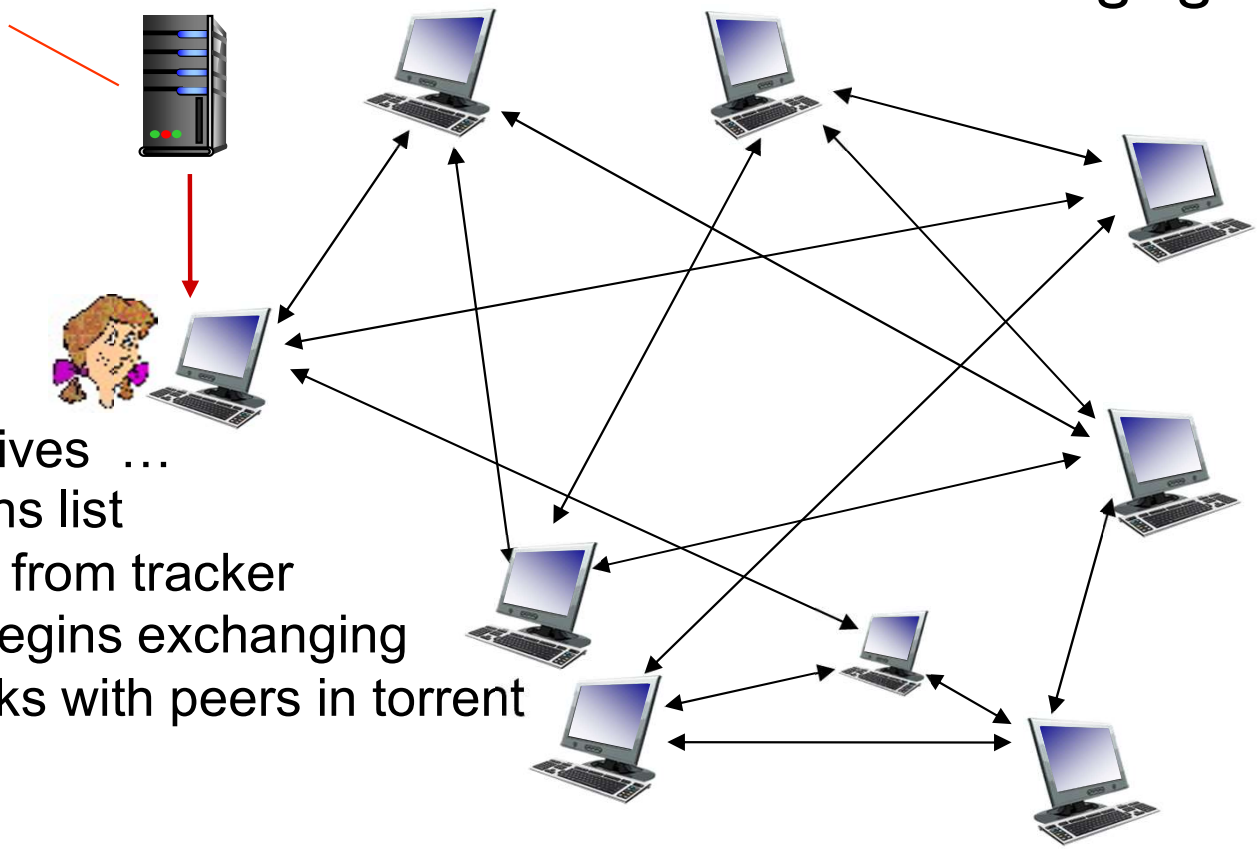


# P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

*tracker:* tracks peers participating in torrent

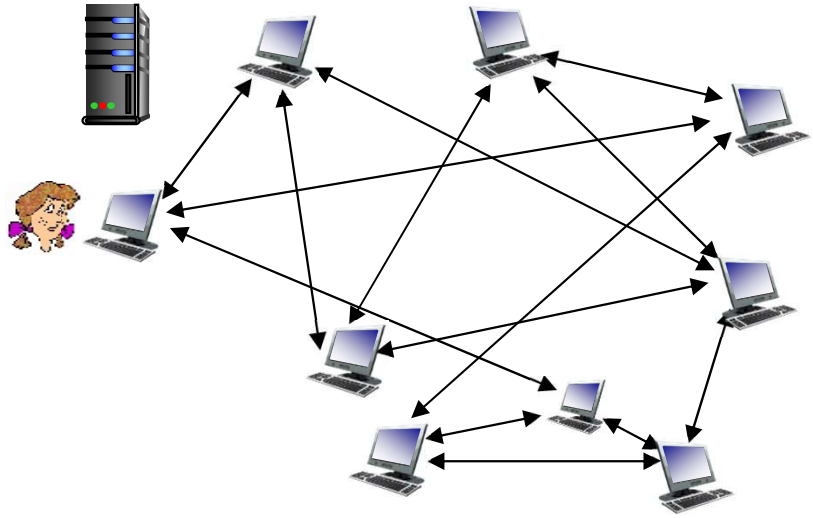
*torrent:* group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

# P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")



- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

# BitTorrent: requesting, sending file chunks

*requesting chunks:*

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

*sending chunks: tit-for-tat*

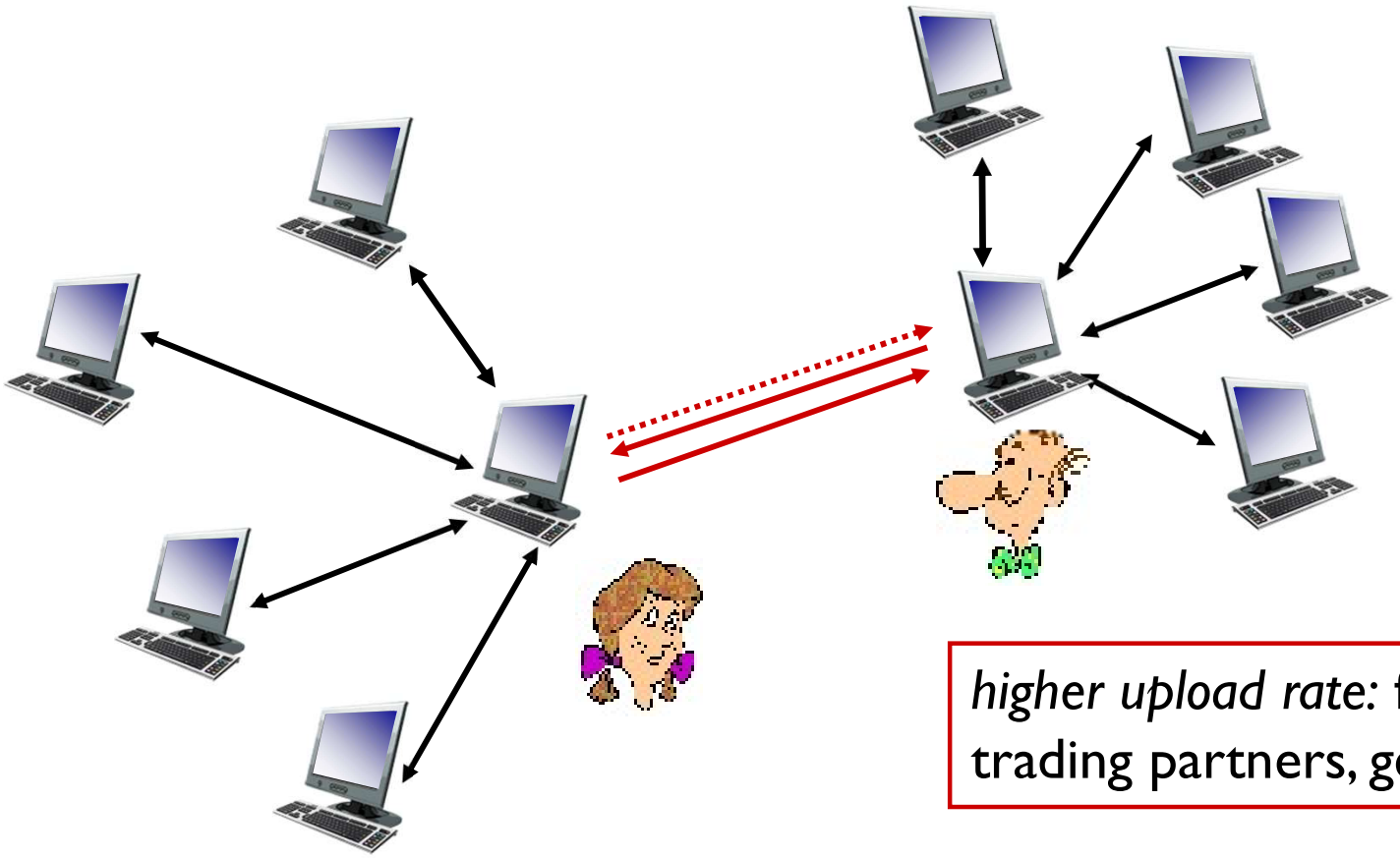
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

# BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers



*higher upload rate:* find better trading partners, get file faster !

# Outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

2.4 DNS

2.5 P2P applications

2.7 socket programming with UDP and TCP

server.py

```
from socket import *

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM) # address family inet (ipv4), protocol TCP
serverSocket.bind(('', serverPort)) # Link object to a socket

serverSocket.listen(1) # how many clients you will listen to at a time

print ("The server is ready to recieve")

while True:
connectionSocket, address = serverSocket.accept() # return connection socket and client address
print (address)

# recv for tcp, recvfrom for udp
message = connectionSocket.recv(2048).decode() # Receive 2048 bit of data, this returns 2 parameters
modifiedMessage = message.upper()
connectionSocket.send(modifiedMessage.encode())

connectionSocket.close()
```

client.py

```
from socket import *

serverName = '127.0.0.1'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_STREAM)
cleintSocket.connect((serverName, serverPort)) # Knock on the door

message = input("Input a lowercase sentence: ")

# sendto() is for UDP, send() is TCP
clientSocket.send(message.encode()) # what are you sending

modifiedMessage = clientSocket.recv(2048)

print (modifiedMessage.decode())

clientSocket.close()
```

server.py

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM) # address family inet (ipv4), socket protocol UDP
serverSocket.bind(('', serverPort)) # Link object to a socket

print ("The server is ready to recieve")

while True:
message, clientAddress = serverSocket.recvfrom(2048) # Receive 2048 bit of data, this returns 2 parameters
modifiedMessage = message.decode().upper()
serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

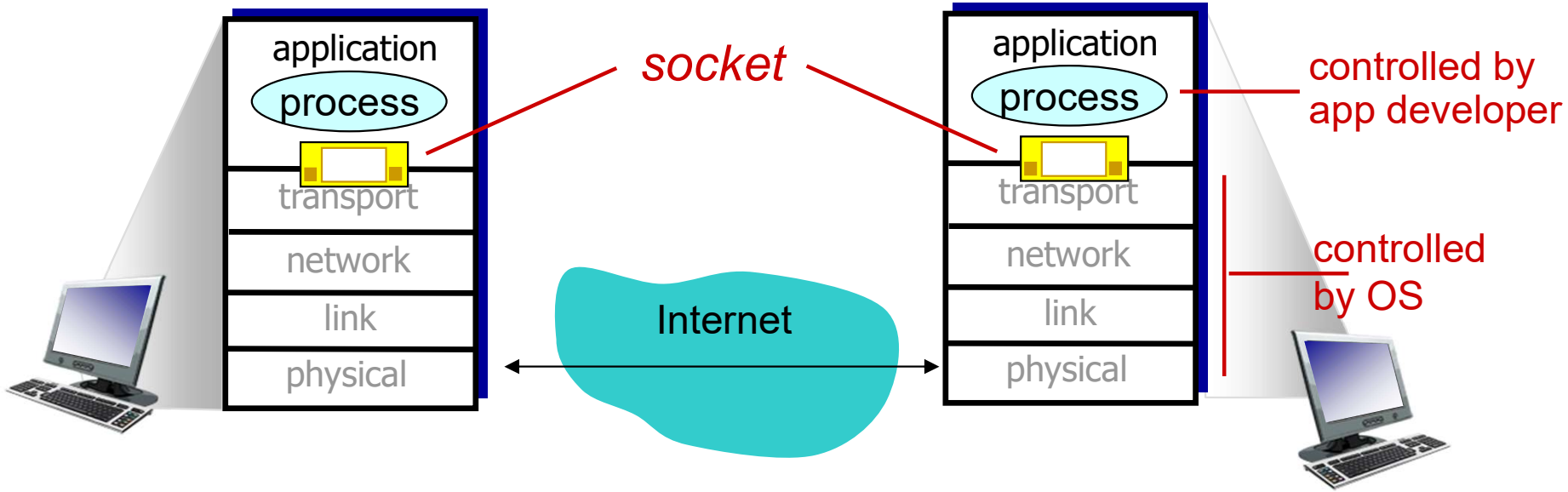
client.py

```
from socket import *

serverName = '127.0.0.1'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_DGRAM)

message = input("Input a lowercase sentence: ")

# sendto() is for UDP, send() is TCP
clientSocket.sendto(message.encode(), (serverName, serverPort)) # what are you sending, and where

modifiedMessage, serverAddress = clientSocker.recvfrom(2048)

print (modifiedMessage.decode())

clientSocket.close()
```

# 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

# Socket programming *with UDP*

## UDP: Connectionless communication

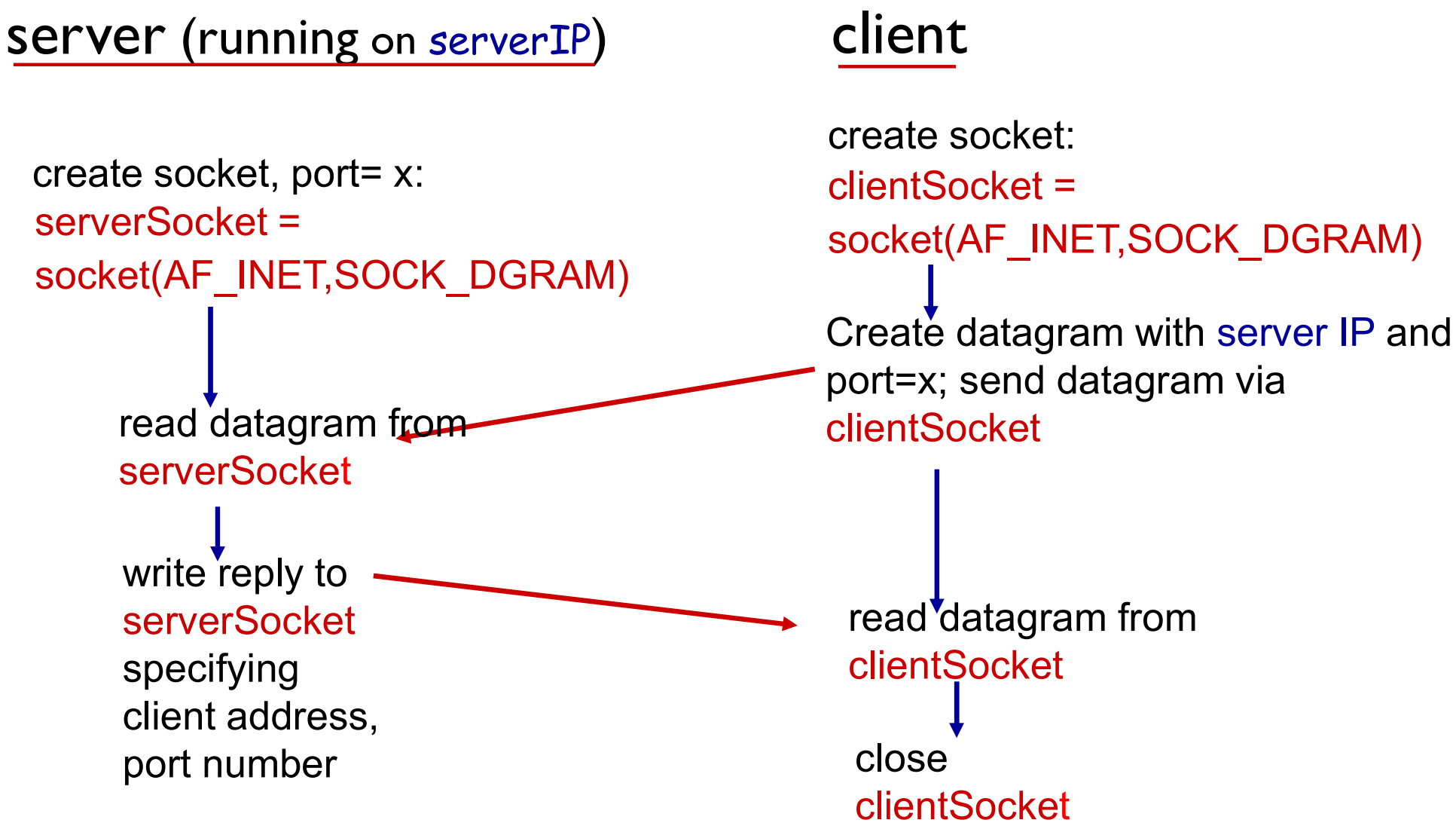
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

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

## Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

# Client/server socket interaction: UDP



# Example app: UDP client

*Python UDPClient*

include Python's socket library

```
from socket import *
serverName = 'hostname'
serverPort = 12000
```

create UDP socket for server

```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```

get user keyboard input

```
message = raw_input('Input lowercase sentence:')
```

Attach server name, port to message; send into socket

```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```

read reply characters from socket into string

```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```

print out received string and close socket

```
print modifiedMessage.decode()
clientSocket.close()
```

# Example app: UDP server

*Python UDPServer*

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

create UDP socket

bind socket to local port number 12000

loop forever

Read from UDP socket into message, getting client's address (client IP and port)

send upper case string back to this client

# 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:

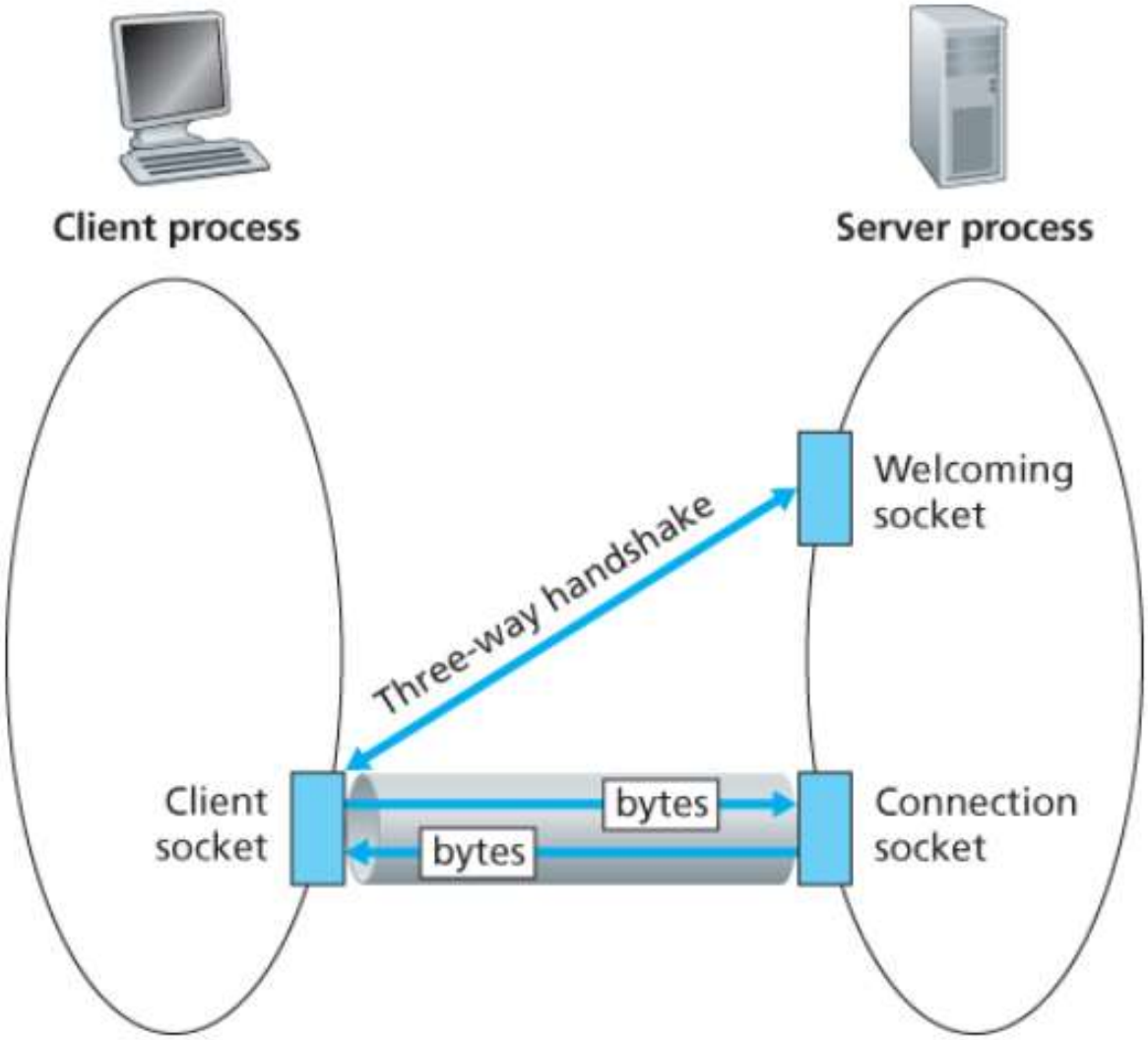
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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

application viewpoint:

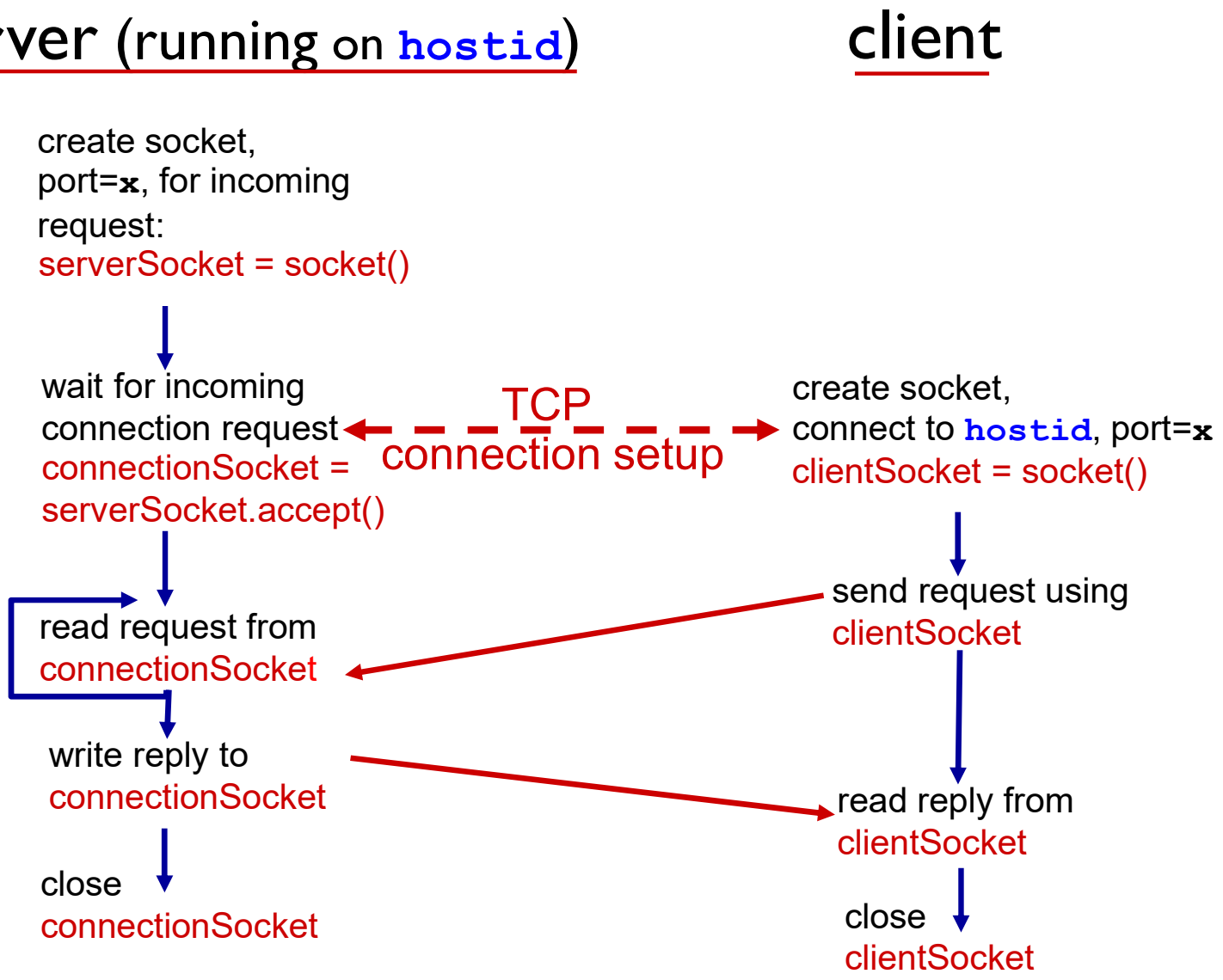
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

# Socket programming *with TCP*




The TCPServer process has two sockets

# Client/server socket interaction: TCP



# Example app: TCP client

*Python TCPClient*

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

# Example app: TCP server

*Python TCPServer*

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                        encode())

    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

# Set 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
  - P2P: BitTorrent
- socket programming: TCP, UDP sockets

# Set 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(payload) being communicated

*important themes:*

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