Application Layer

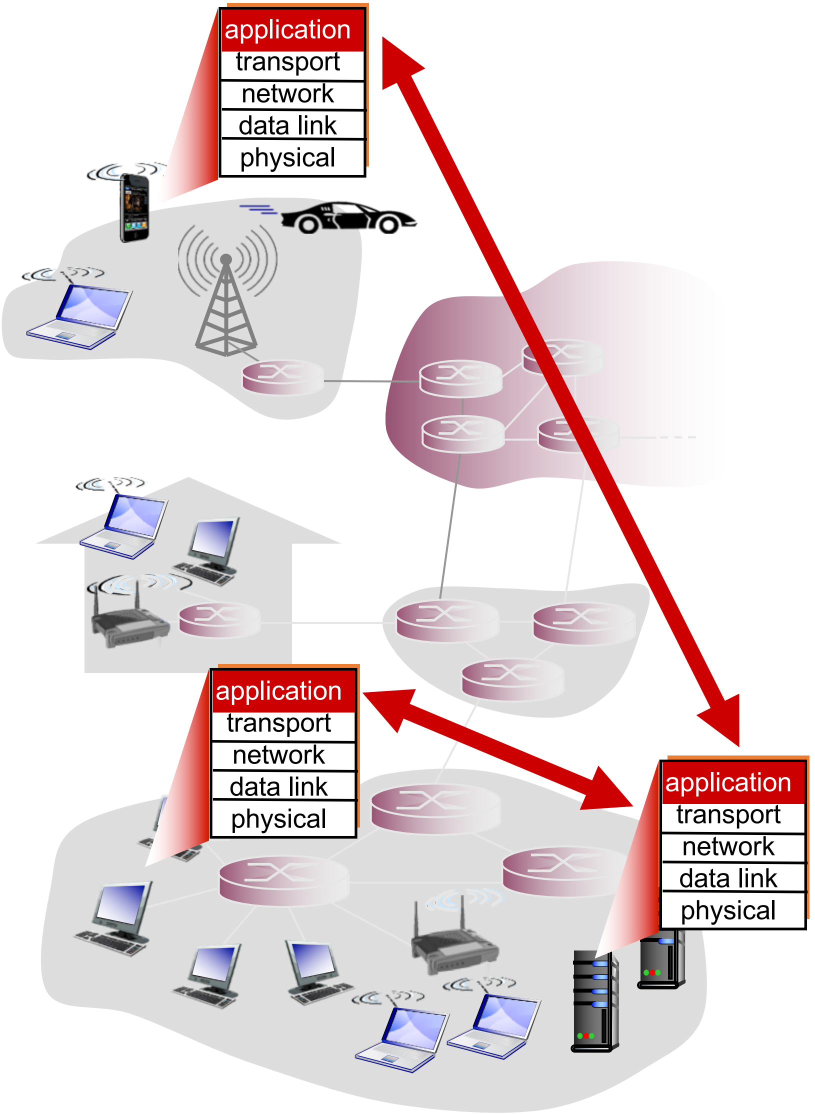
CMPS 4750/6750: Computer Networks

Agenda

- Principles of Network Applications
- Case Studies
  - Web and HTTP
  - Email
  - Domain Name System (DNS)
  - Peer-to-Peer File Sharing
- Socket Programming with UDP and TCP

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



Some network apps

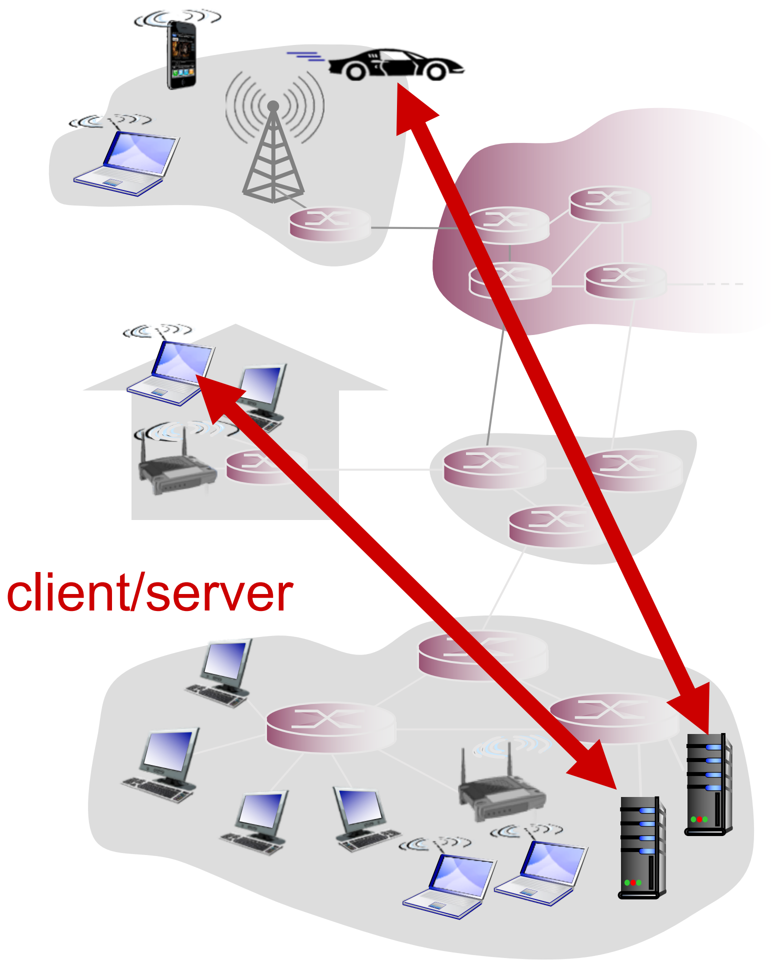
- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- search
- …

Application architectures

possible structure of applications:

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

Client-server architecture



server:

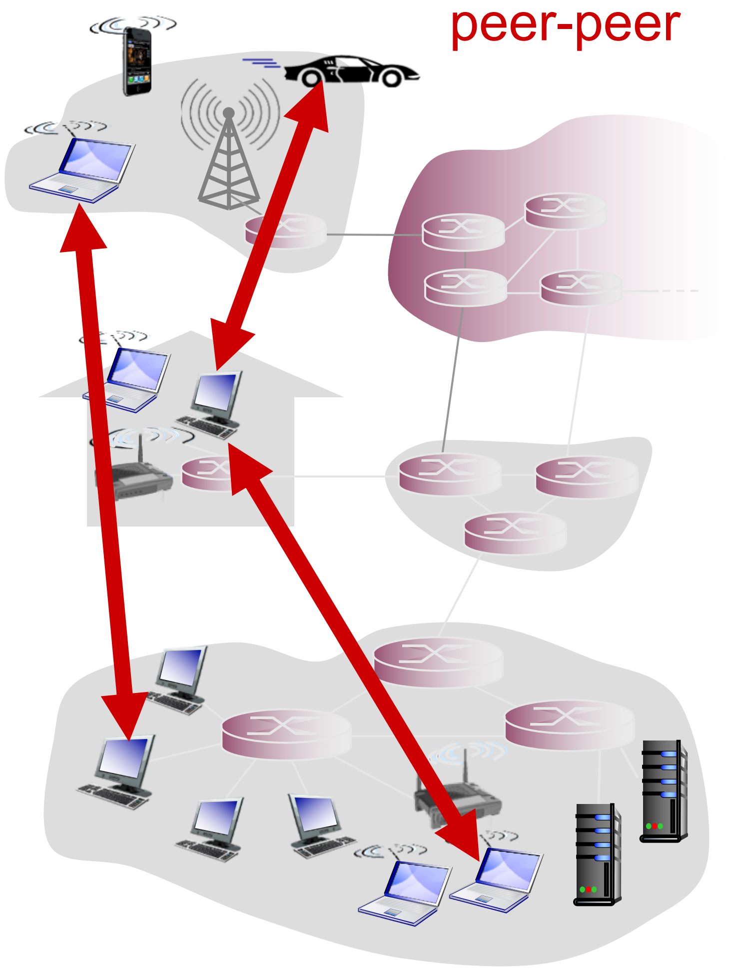
- always-on host
- permanent IP address
- data centers for scaling

clients:

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

Peer-to-peer (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: in a P2P application, a process can be both a client process & a server process

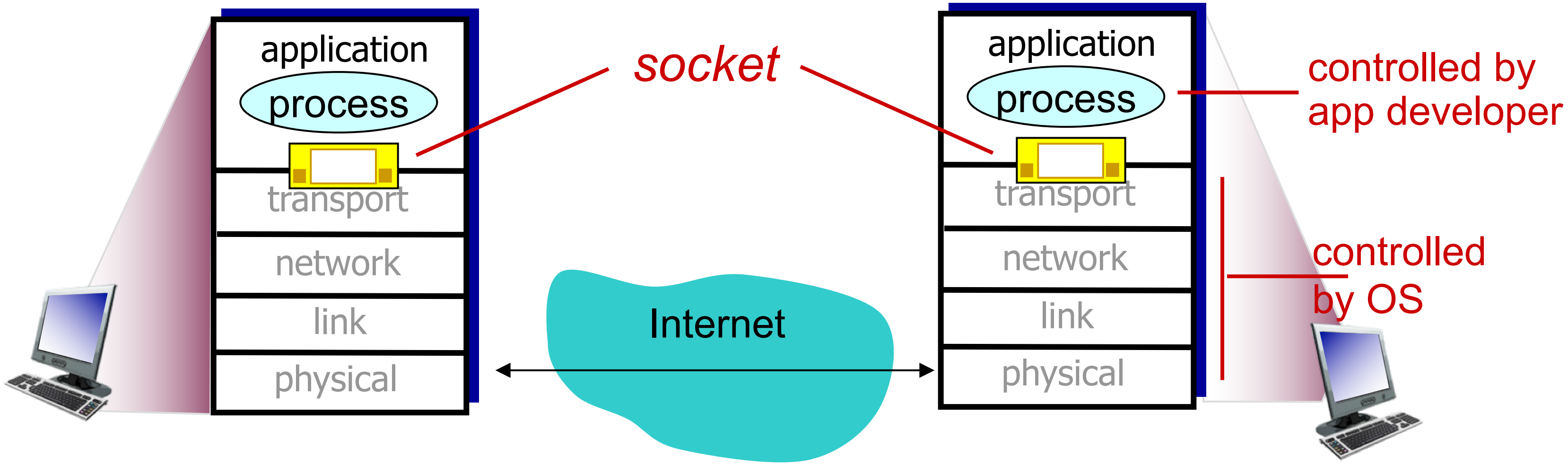
Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* no, *many* processes can be running on same host

- *identifier* includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to cs.tulane.edu web server:
  - IP address: 129.81.226.25
  - port number: 80
- more shortly…

Socket

- 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



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, authentication

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, 100s of msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100s of msec
text messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

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

UDP service:

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

Internet apps: application, transport protocols

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

Agenda

- Principles of Network Applications
- Case Studies
  - Web and HTTP
  - Email
  - Domain Name System (DNS)
  - Peer-to-Peer File Sharing
- Socket Programming with UDP and TCP

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.,

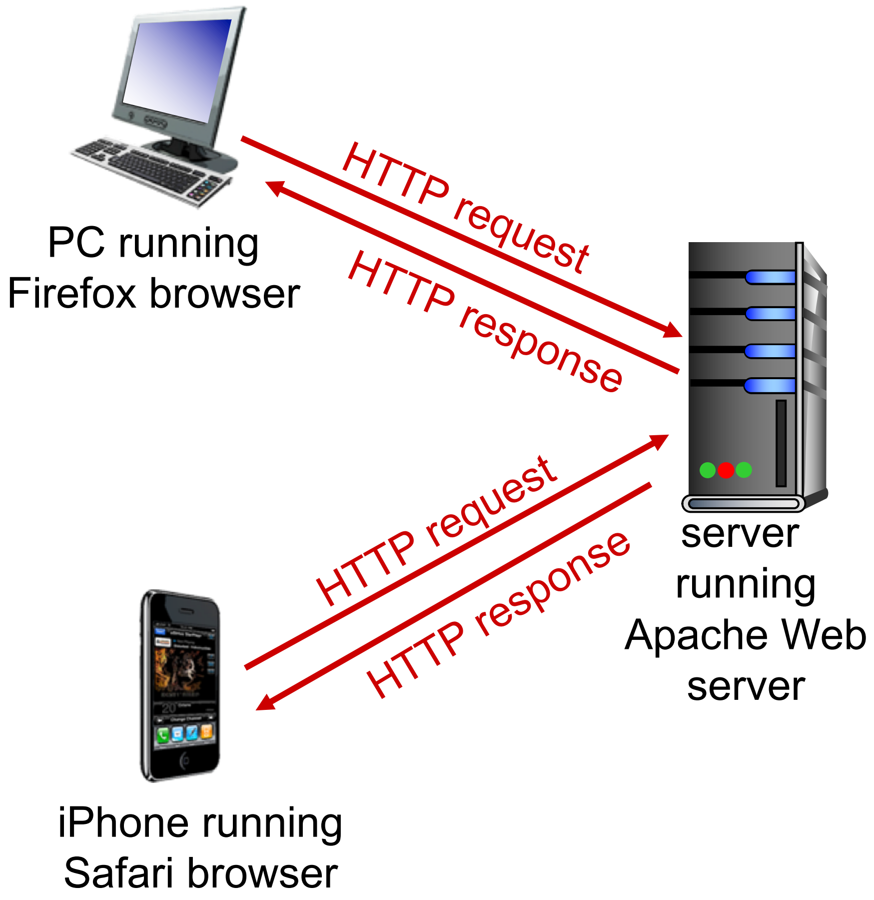
`www.someschool.edu/someDept/pic.gif`

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

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 v1.0: RFC 1945
- HTTP/1.1: RFC 2616



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

Non-persistent HTTP

suppose user enters URL: (contains text, references to 10 jpeg images)

`www.someSchool.edu/someDepartment/home.index`

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.

time

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

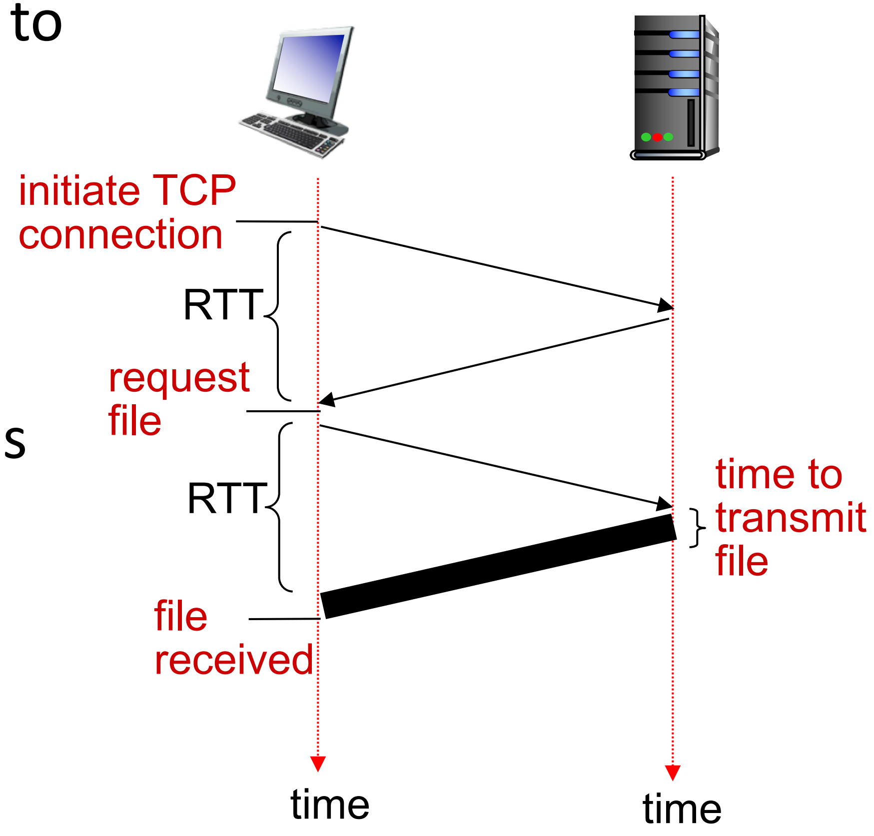
Non-persistent HTTP: response time

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 request and first few bytes of HTTP response to return
- non-persistent HTTP response time =

  2RTT+ file transmission time



Non-persistent HTTP with parallel TCP connections

- What is the total time to retrieve a webpage that consists of a base HTML file and and 10 JPEG images?
  - Assume the objects are very small and ignore transmission time
- uses serial TCP connections: $11 \cdot 2RTT$
- use 5 parallel TCP connections: $3 \cdot 2RTT$

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 (pipelining)
- What is the total time to retrieve a webpage that consists of a base HTML file and and 10 JPEG images using persistent HTTP?

2RTT+RTT = 3RTT

HTTP request message

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

carriage return character

line-feed character

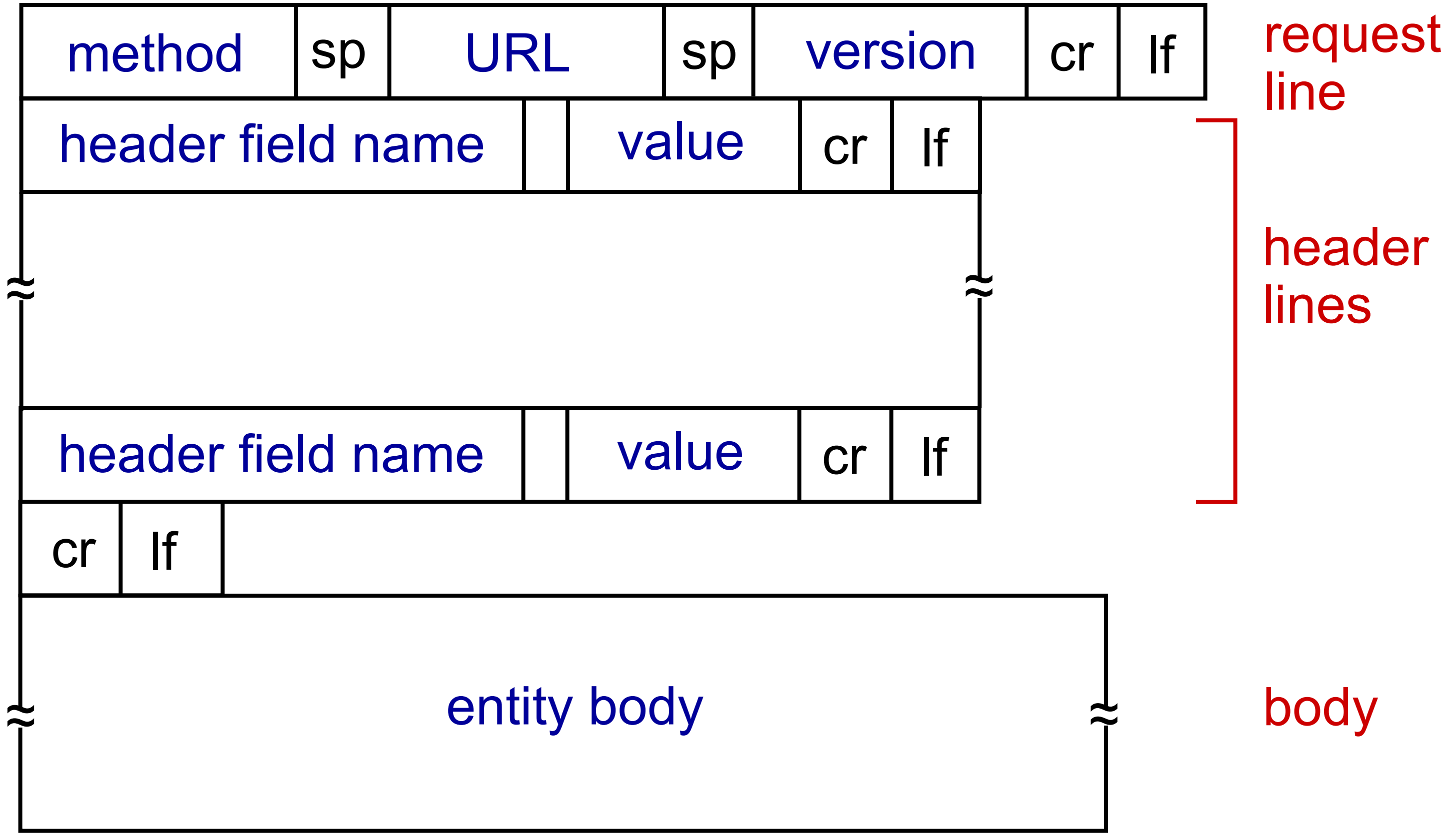
request line (GET, POST, HEAD commands)

header lines

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

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

HTTP request message: general format



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
```

HTTP response message

status line (protocol status code status phrase)

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

header lines

data, e.g., requested HTML file

HTTP response status codes

- 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`

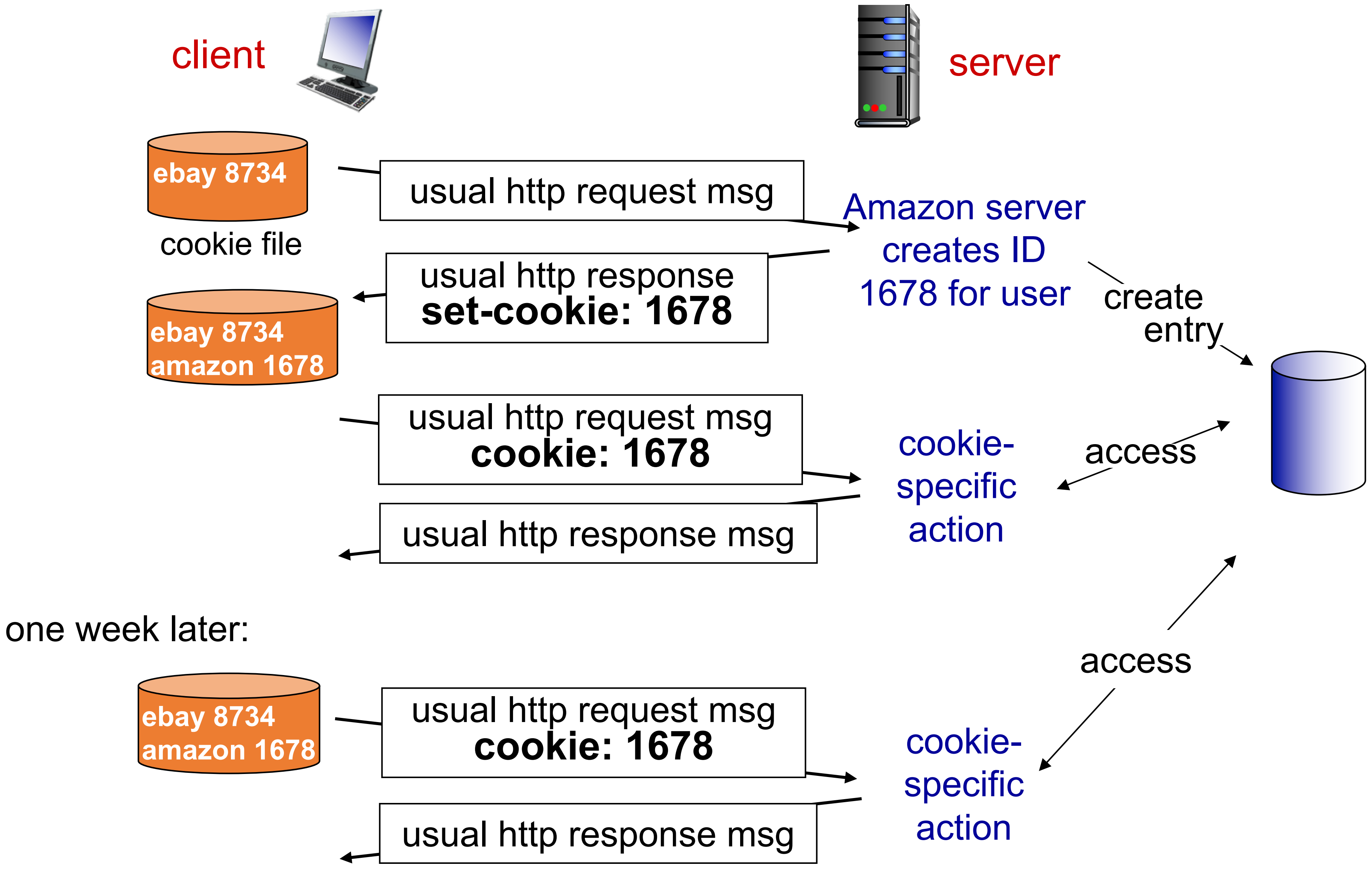
- request msg not understood by server

`404 Not Found`

- requested document not found on this server

`505 HTTP Version Not Supported`

Cookies: keeping “state”



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

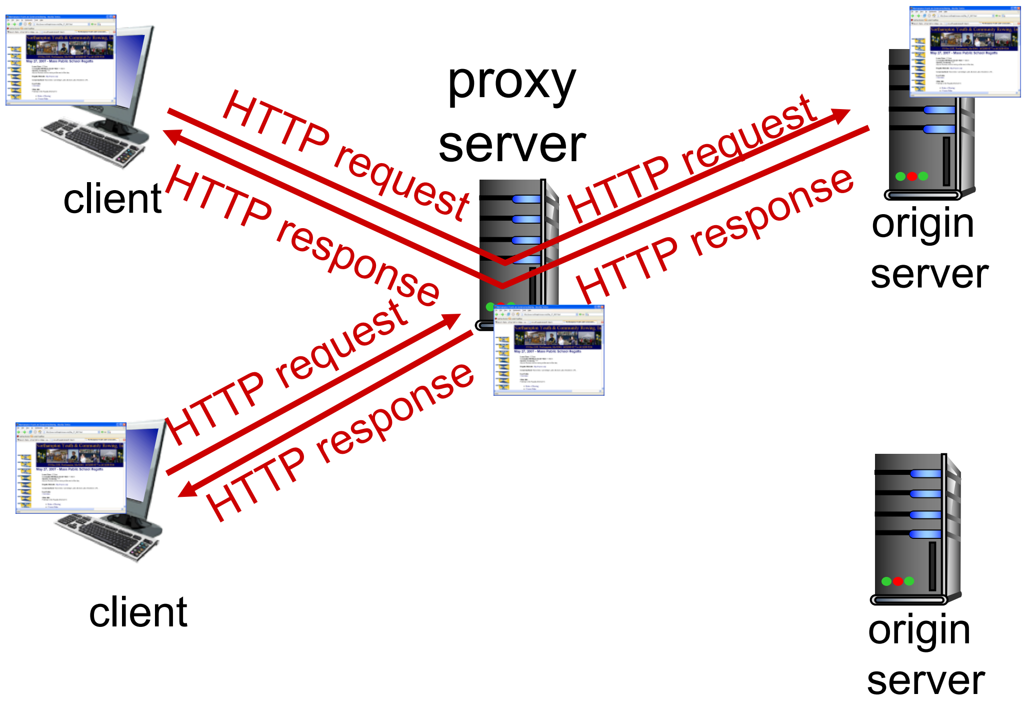
how to keep "state":

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

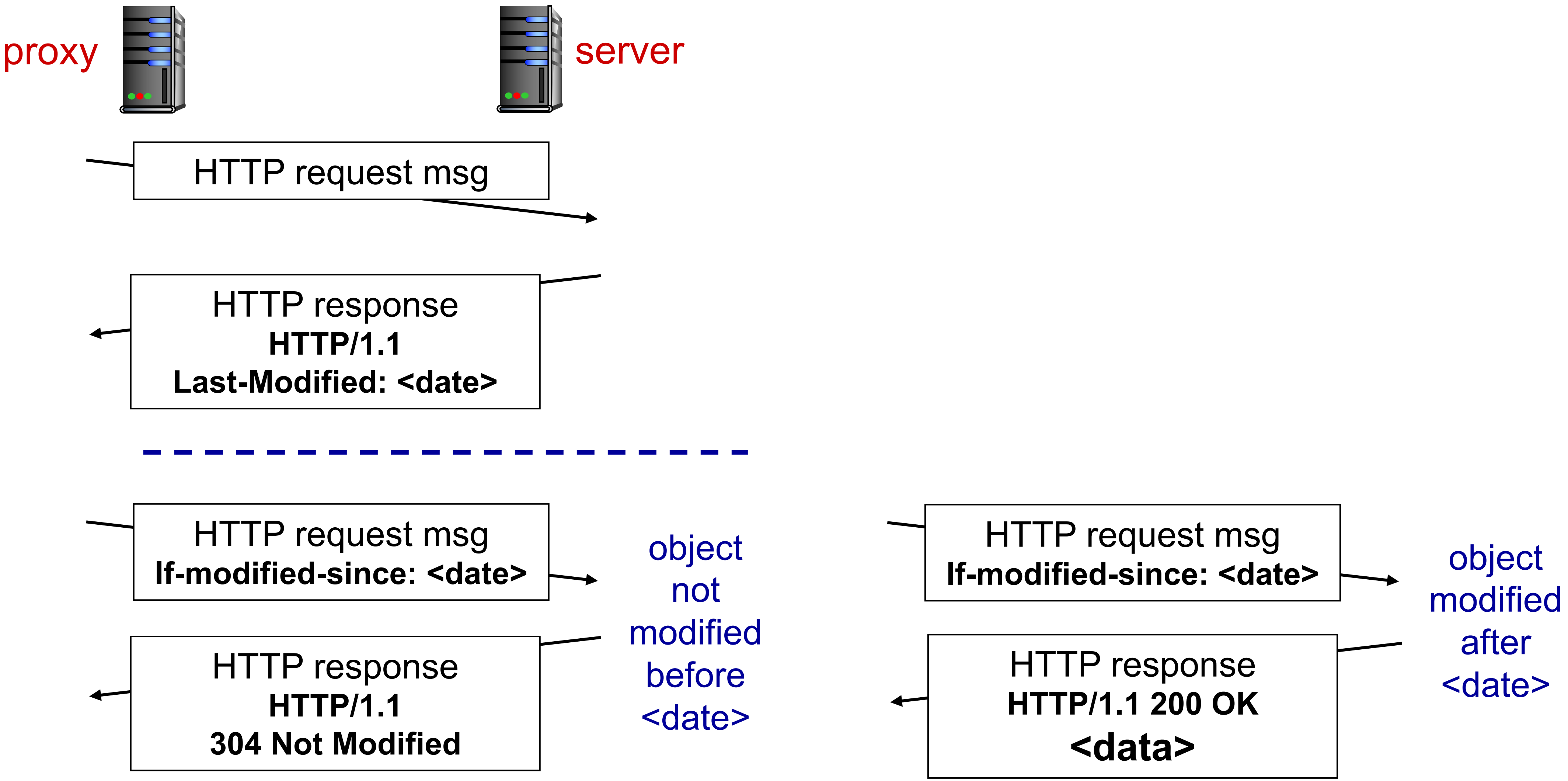
Web caches (proxy server)

- *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



Conditional GET



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)

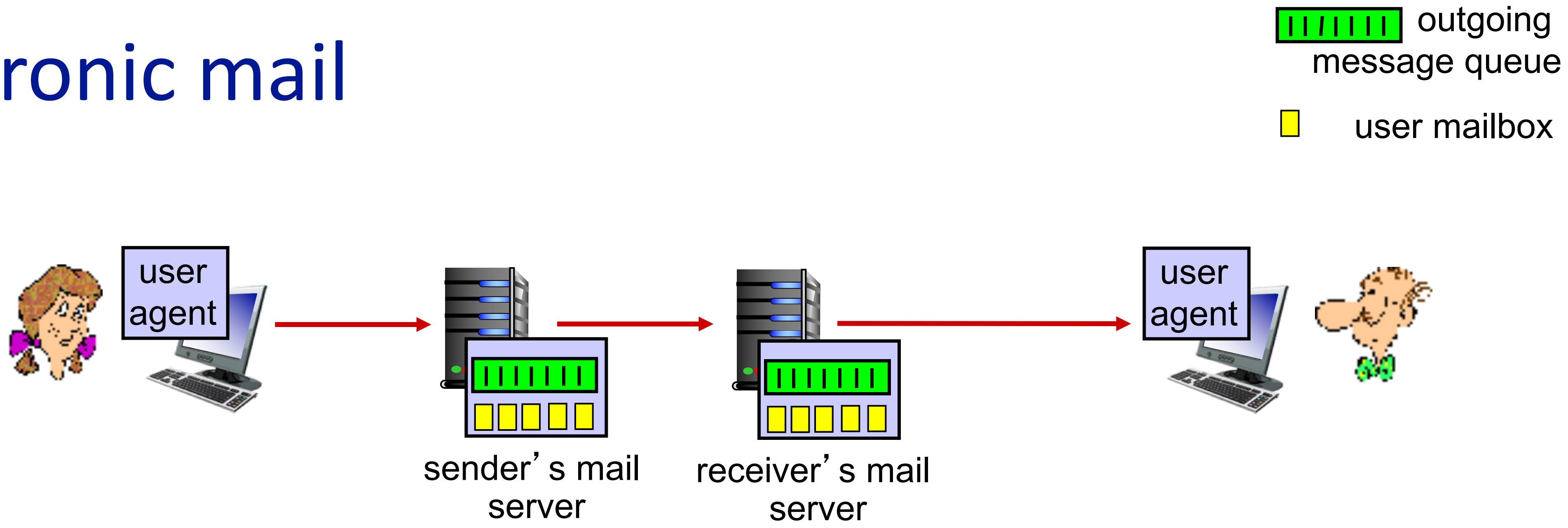
why Web caching?

- 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

Agenda

- Principles of Network Applications
- Case Studies
  - Web and HTTP
  - Email
  - Domain Name System (DNS)
  - Peer-to-Peer File Sharing
- Socket Programming with UDP and TCP

Electronic mail



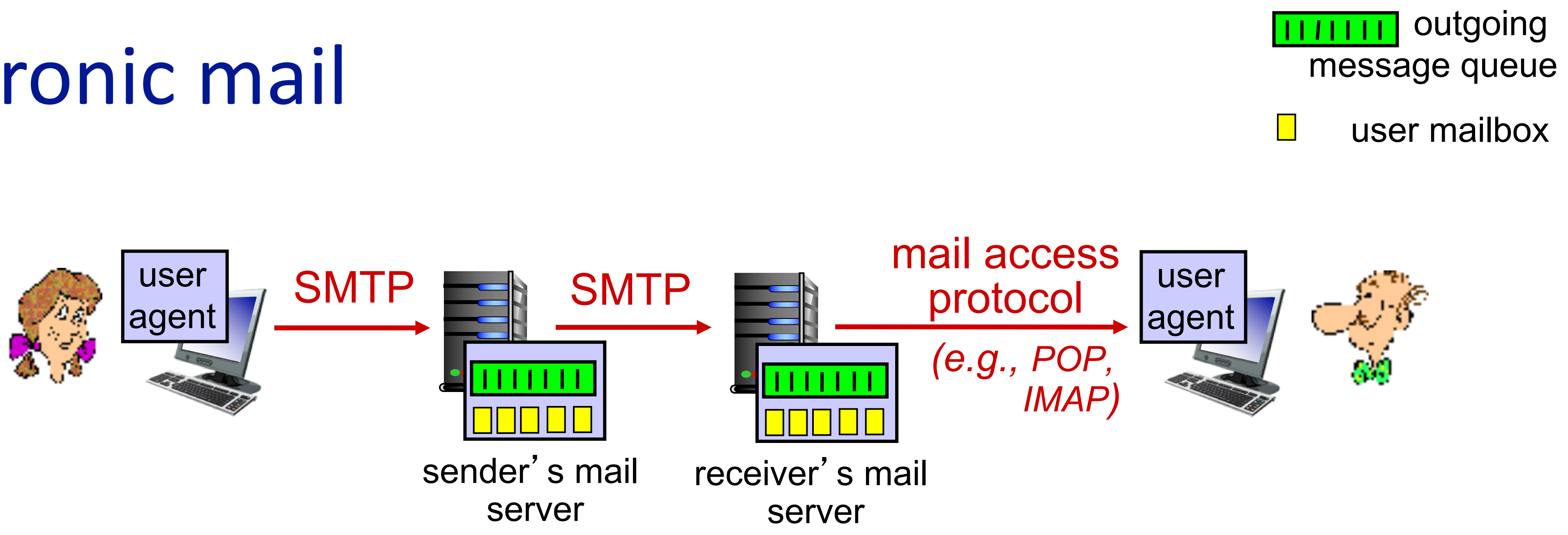
User Agent

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

Mail Server

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages

Electronic mail



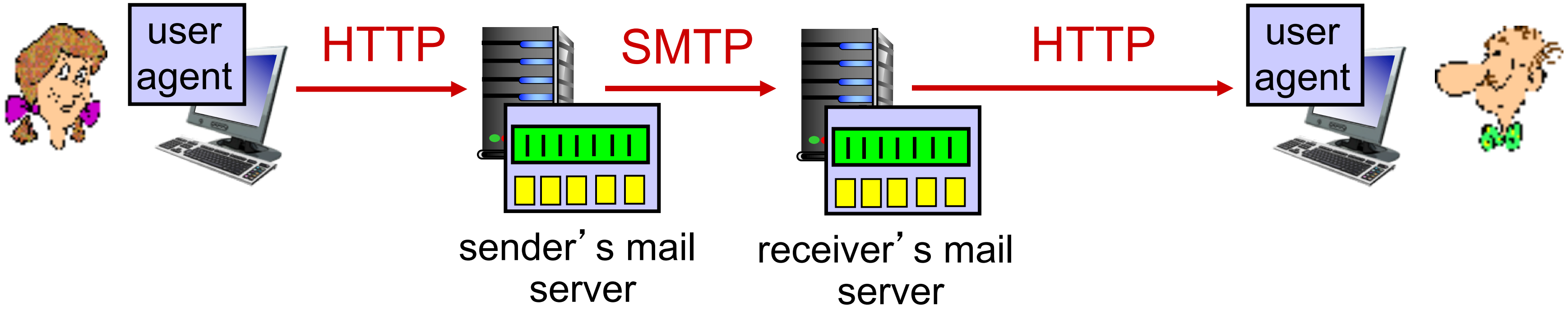
SMTP (Simple Mail Transfer Protocol)

- between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server
- Uses TCP, port 25, RFC 2821

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.

Electronic mail



Scenario: Alice sends message to Bob

1) Alice uses UA to compose message "to" bob@someschool.edu

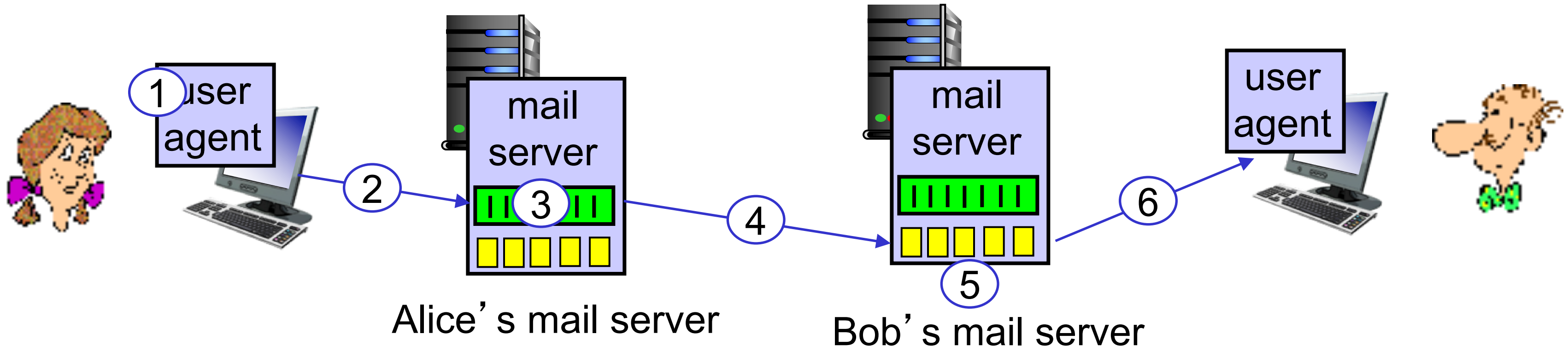
2) Alice's UA sends message to her mail server; message placed in message queue

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

handshaking (greeting)

```
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
```

transfer of messages

```
C: From: alice@crepes.fr
C: To: bob@hamburger.edu
C: Subject: Searching for the meaning of life.
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
```

closure

```
C: QUIT
S: 221 hamburger.edu closing connection
```

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:

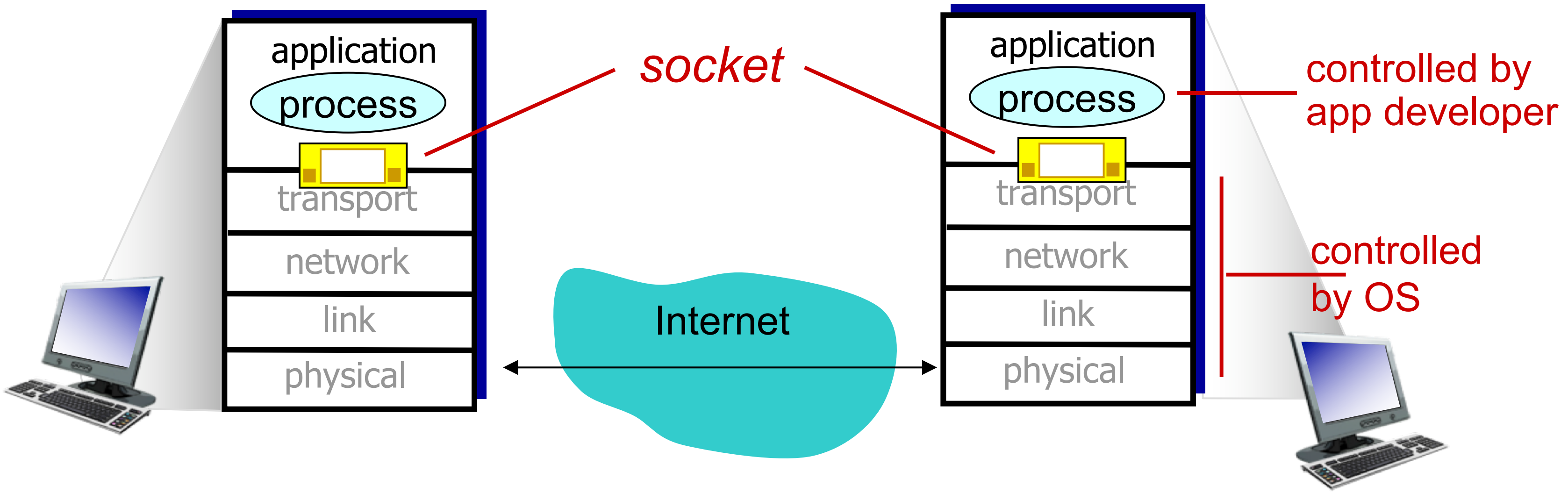
- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message

Agenda

- Principles of Network Applications
- Case Studies
  - Web and HTTP
  - Email
  - Domain Name System (DNS)
  - Peer-to-Peer File Sharing
- Socket Programming with UDP and TCP

Socket programming

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



Socket programming

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: no "connection" between client & server

- 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

server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: *Python UDPClient*

include Python's socket library → `from socket import *`

```
serverName = 'hostname'

serverPort = 12000
```

create UDP socket → `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: *Python UDPServer*

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 →

```
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)
```

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:

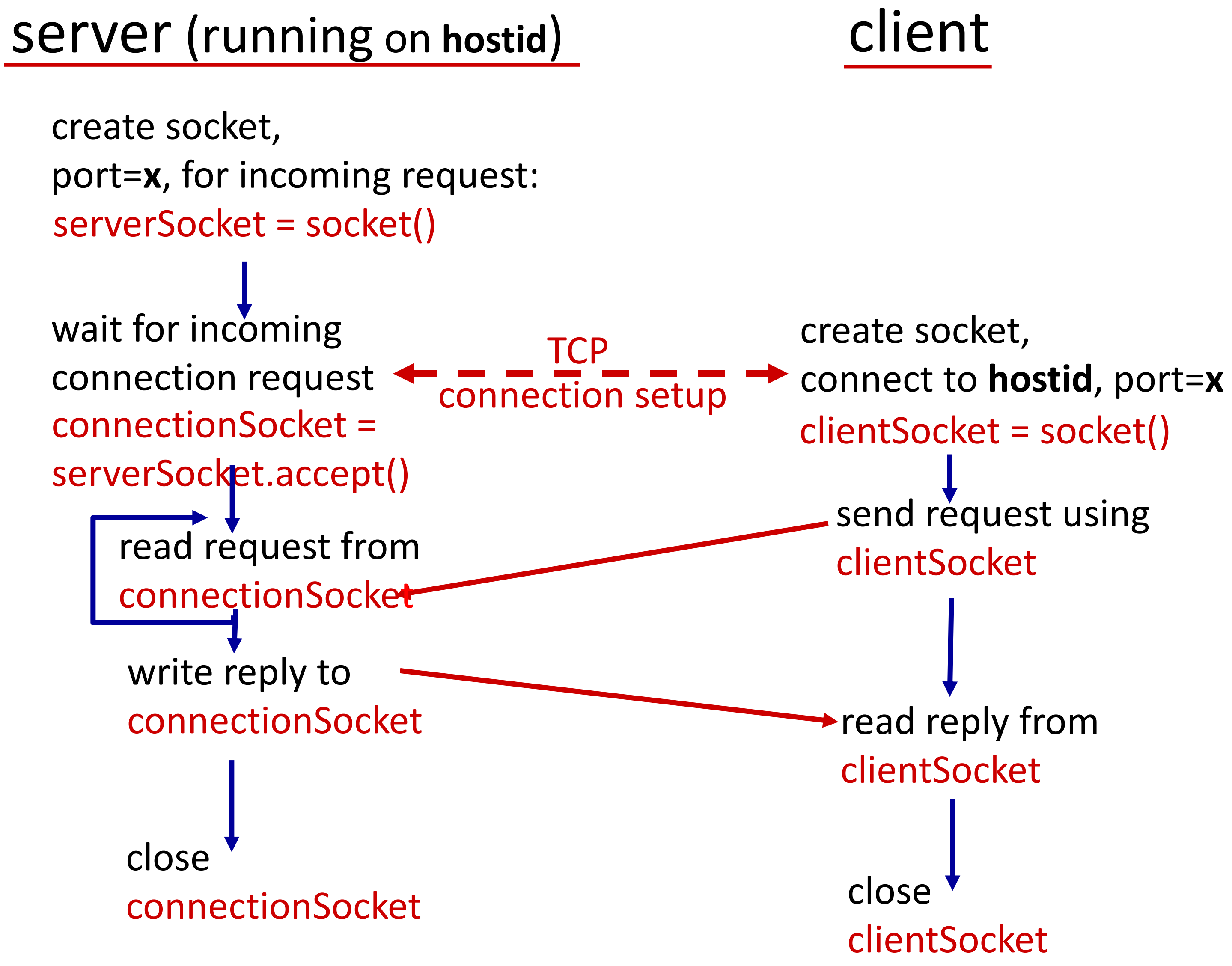
- creating TCP socket, specifying IP address, port number of server process
- client TCP establishes connection to server TCP

- 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 IP addresses/port numbers used to distinguish clients

application viewpoint:

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

Client/server socket interaction: TCP



Example app: *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 → `clientSocket = socket(AF_INET, SOCK_STREAM)`

No need to attach server name, port → `clientSocket.send(sentence.encode())`

Example app: *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 → `serverSocket = socket(AF_INET,SOCK_STREAM)`
- server begins listening for incoming TCP requests → `serverSocket.listen(1)`
- loop forever → `while True:`
- server waits on accept() for incoming requests, new socket created on return → `connectionSocket, addr = serverSocket.accept()`
- read bytes from socket (but not address as in UDP) → `sentence = connectionSocket.recv(1024).decode()`
- close connection to this client (but *not* welcoming socket) → `connectionSocket.close()`

Lab assignment 1

- Programming assignment: develop a simple web server that is able to
  - accept and parse one HTTP GET request, get the requested file from the server's file system and create an HTTP response message.
  - if the requested file is not present in your server, return a '404 Not Found' error message
  - using multithreading to serve multiple requests simultaneously (graduate students)
  - ✓ skeleton code in Python is provided
- Wireshark lab: DNS
  - get familiar with nslookup and ifconfig/ipconfig
  - use Wireshark to observe DNS in action
- Due on Mar 16
- Hand in: (1) complete code and screen shots of client browser; (2) Lab report

Agenda

- Principles of Network Applications
- Case Studies
  - Web and HTTP
  - Email
  - Domain Name System (DNS)
  - Peer-to-Peer File Sharing
- 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

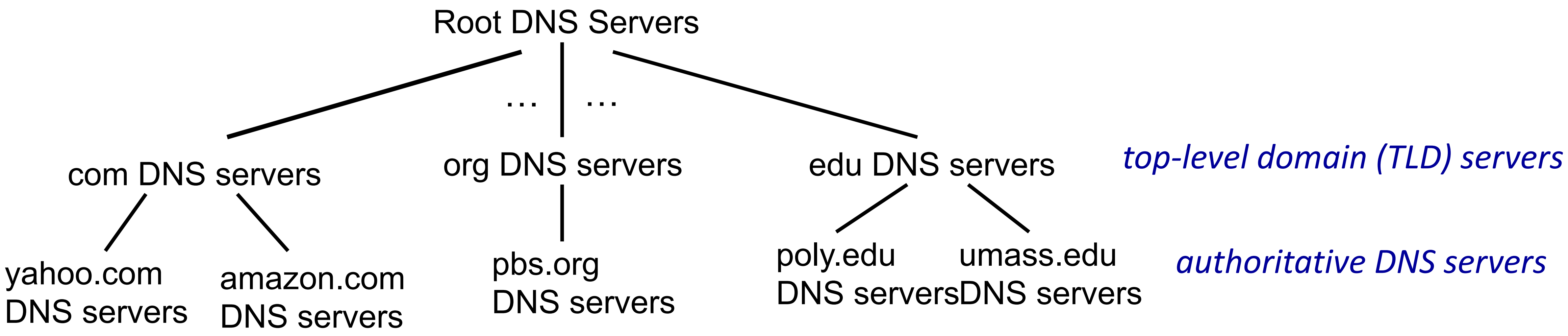
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

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

A: doesn't scale!

DNS: a distributed, hierarchical database



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

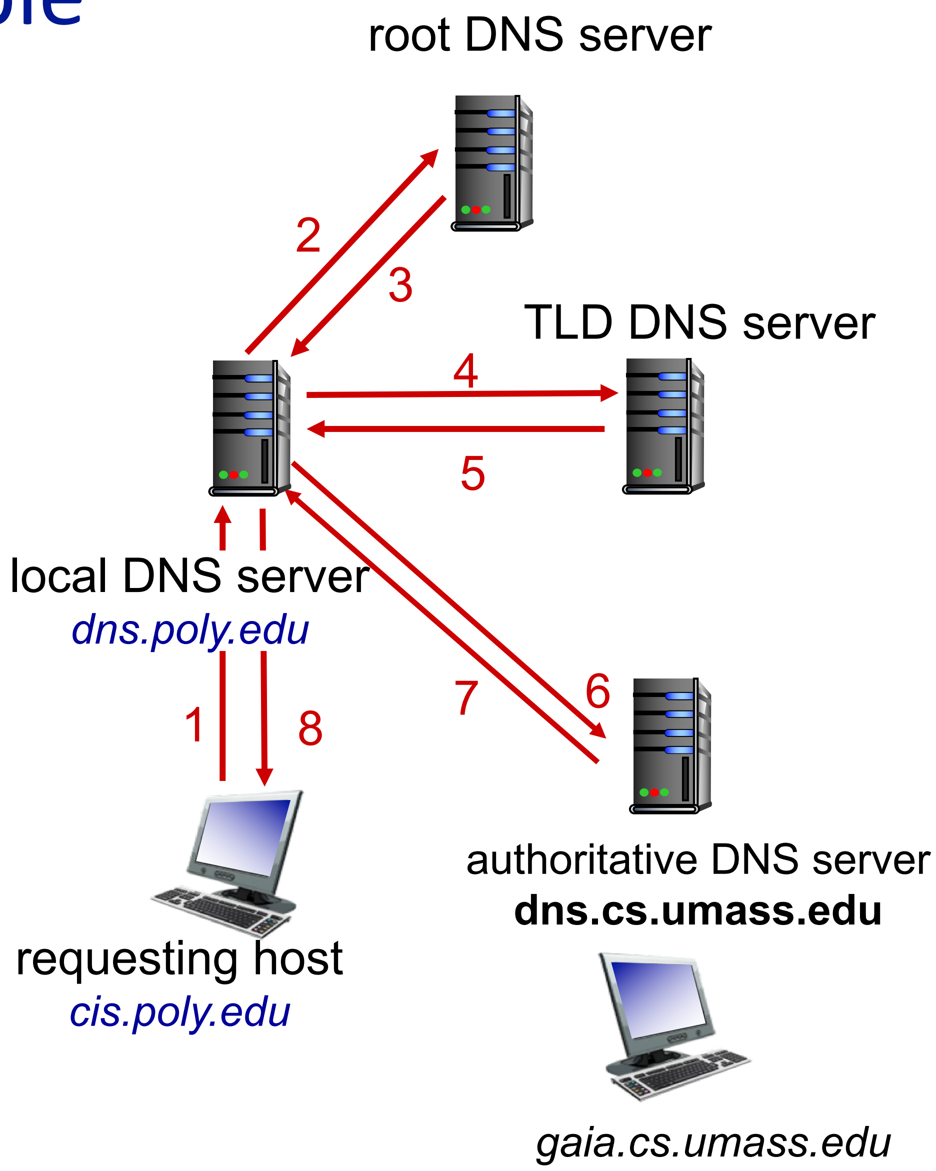
- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS name resolution example

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

iterated query:

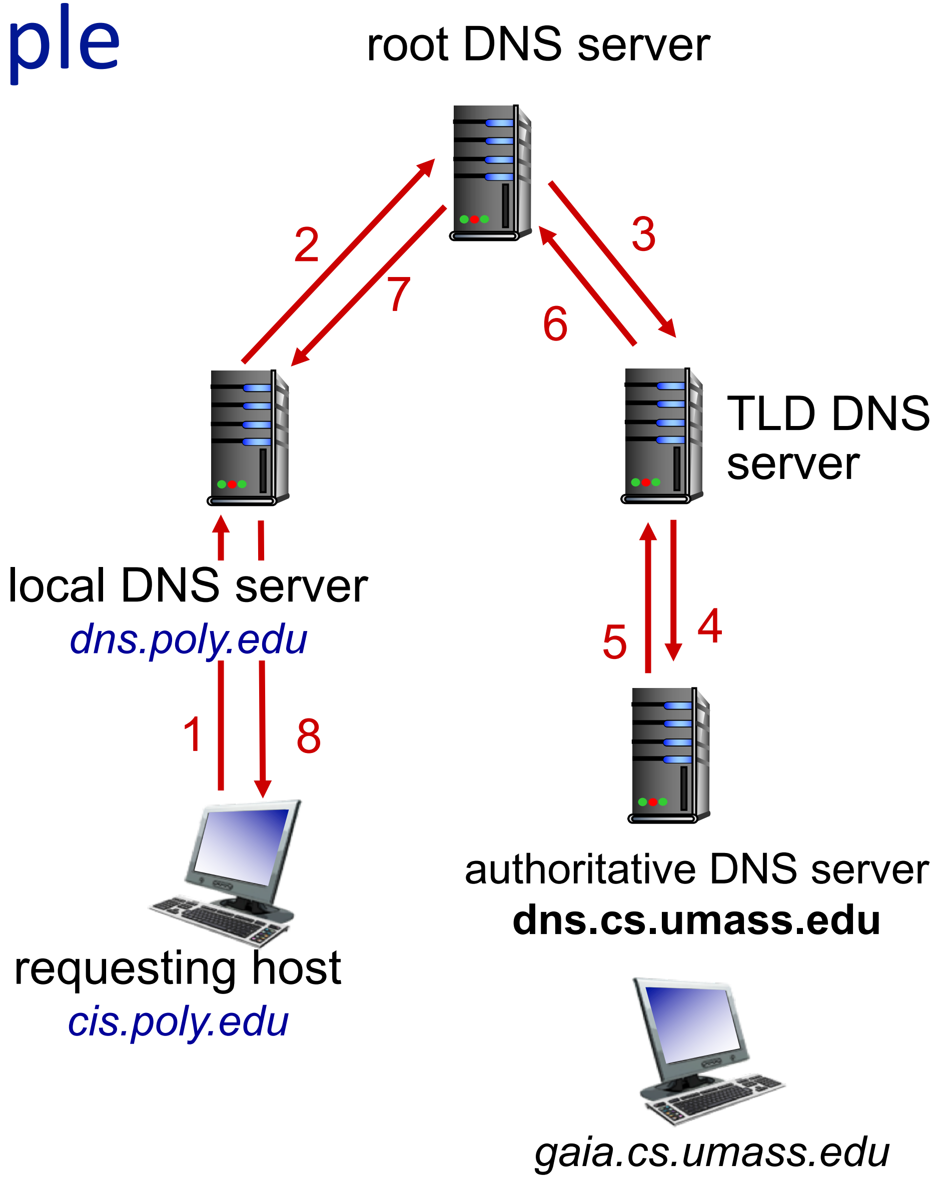
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



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 (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 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

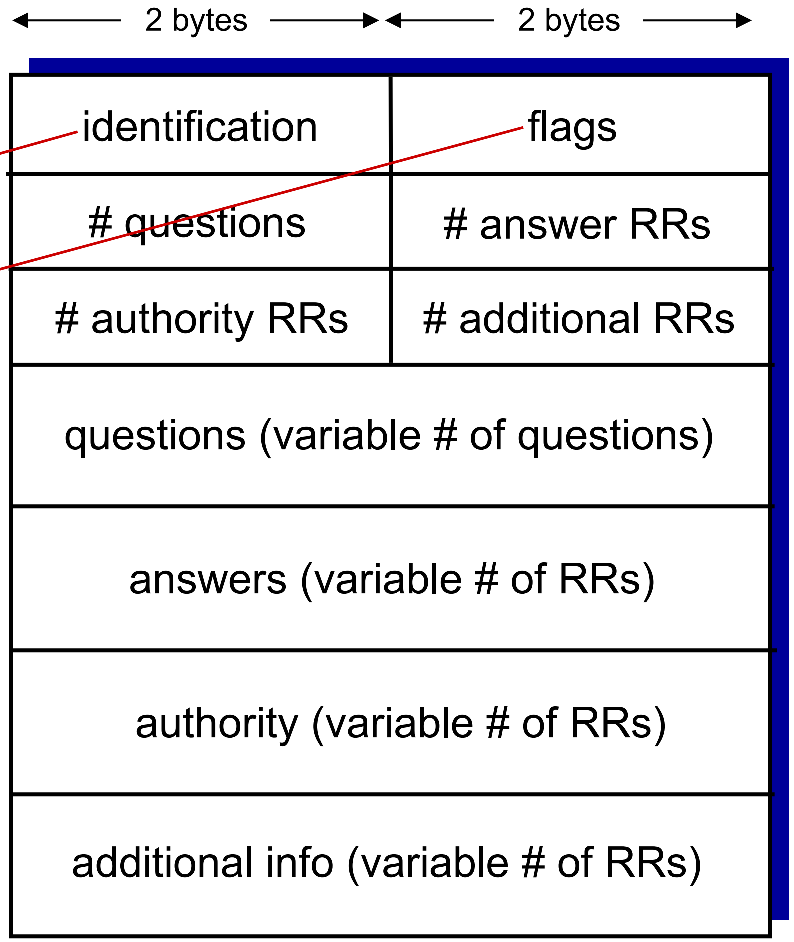
- **value** is name of mailserver associated with **name**

DNS protocol, messages

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

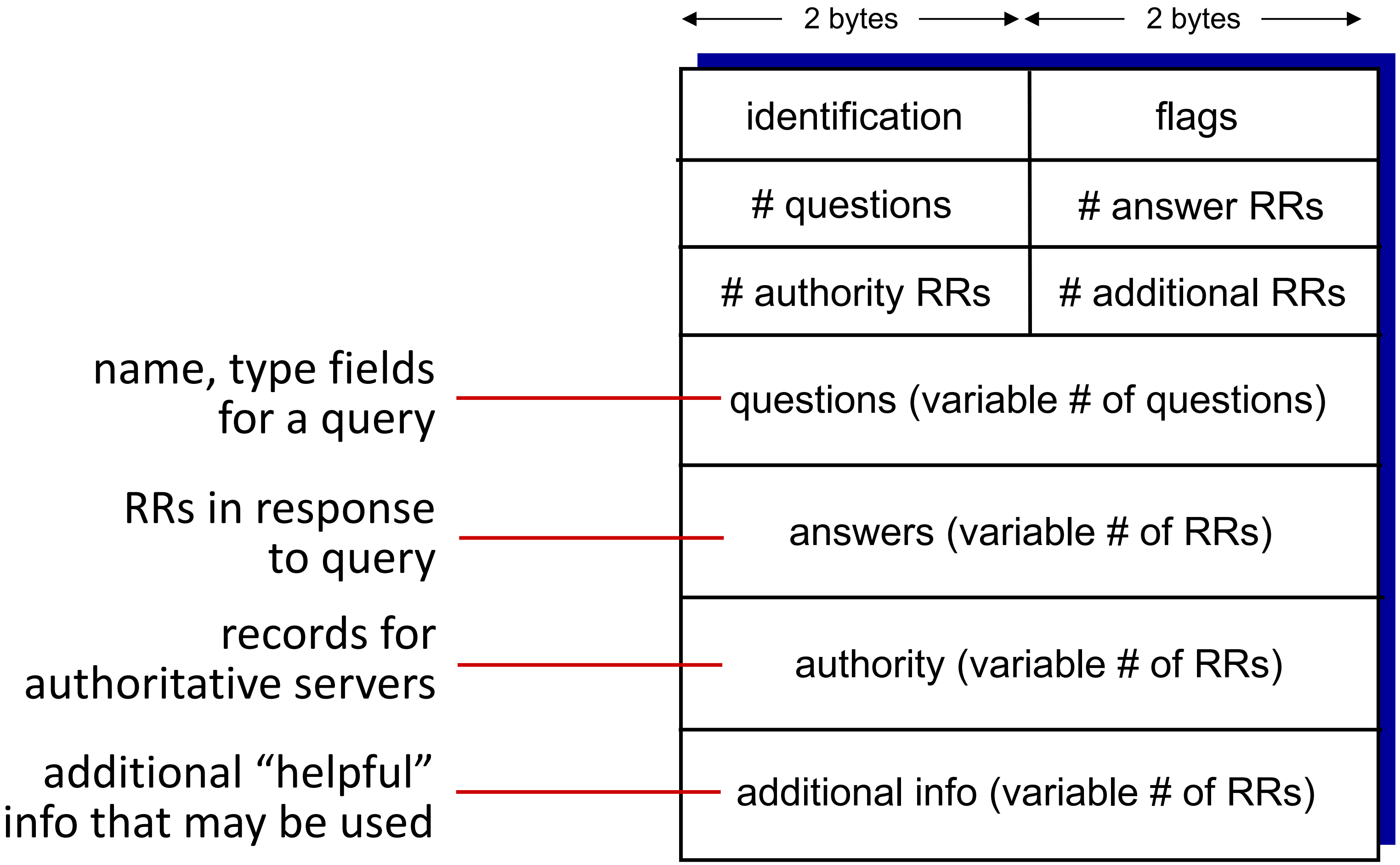
message header

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



DNS protocol, messages

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



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 A and type MX records for mail.networkutopia.com

Agenda

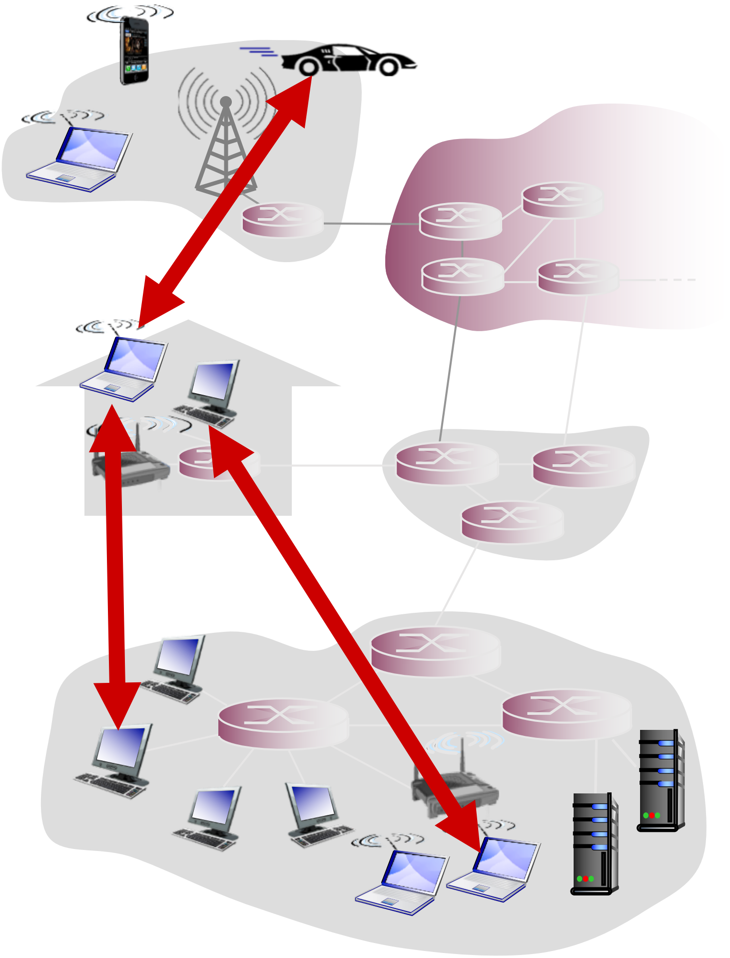
- Principles of Network Applications
- Case Studies
  - Web and HTTP
  - Email
  - Domain Name System (DNS)
  - Peer-to-Peer File Sharing
- Socket Programming with UDP and TCP

Pure P2P architecture

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

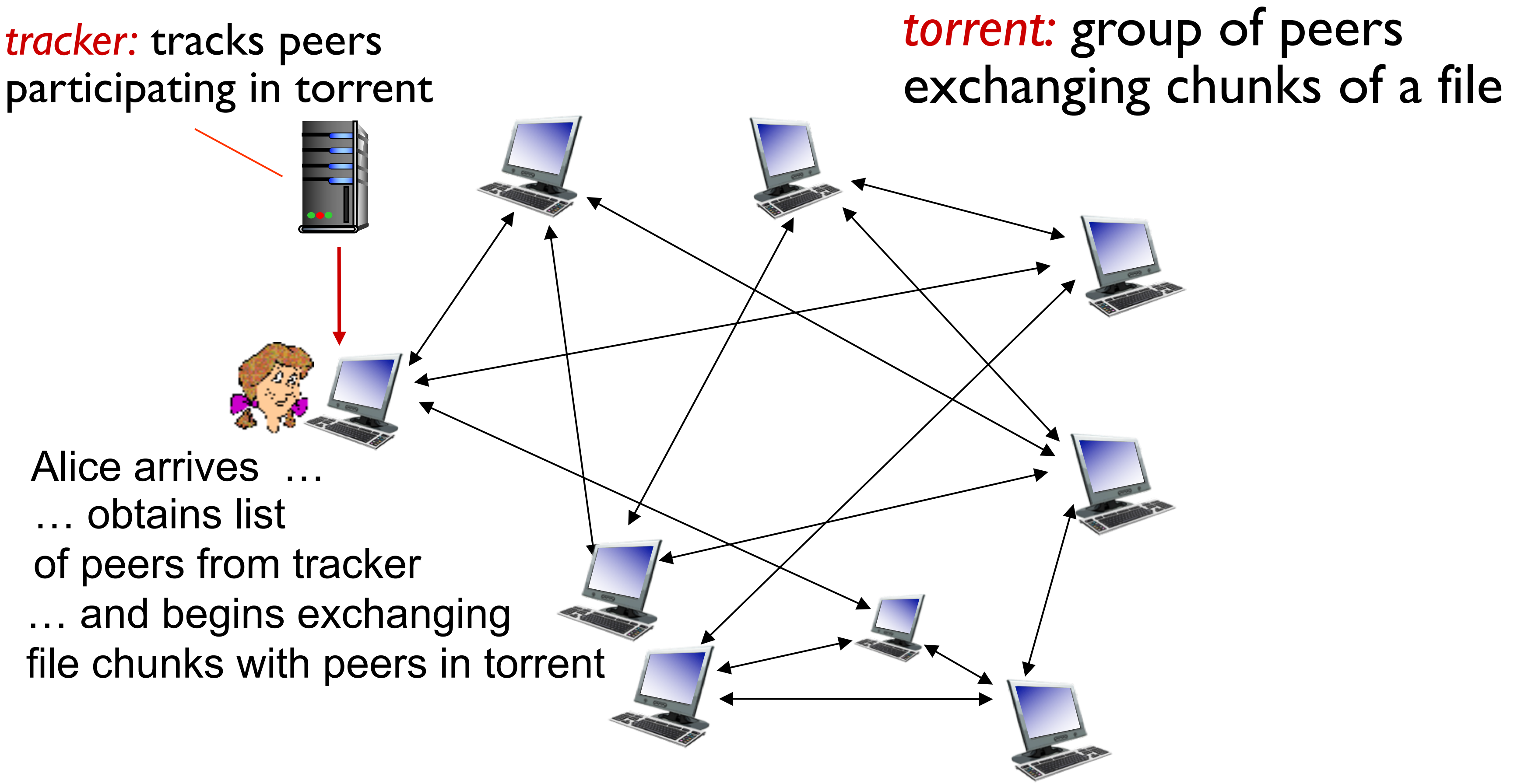
examples:

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

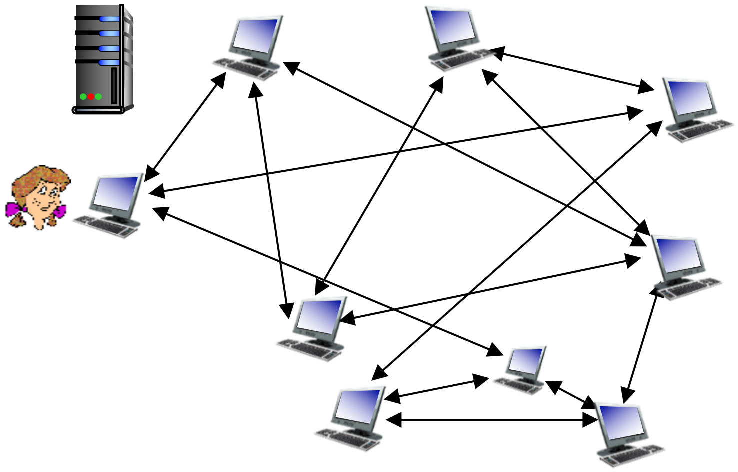


P2P file distribution: BitTorrent

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



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

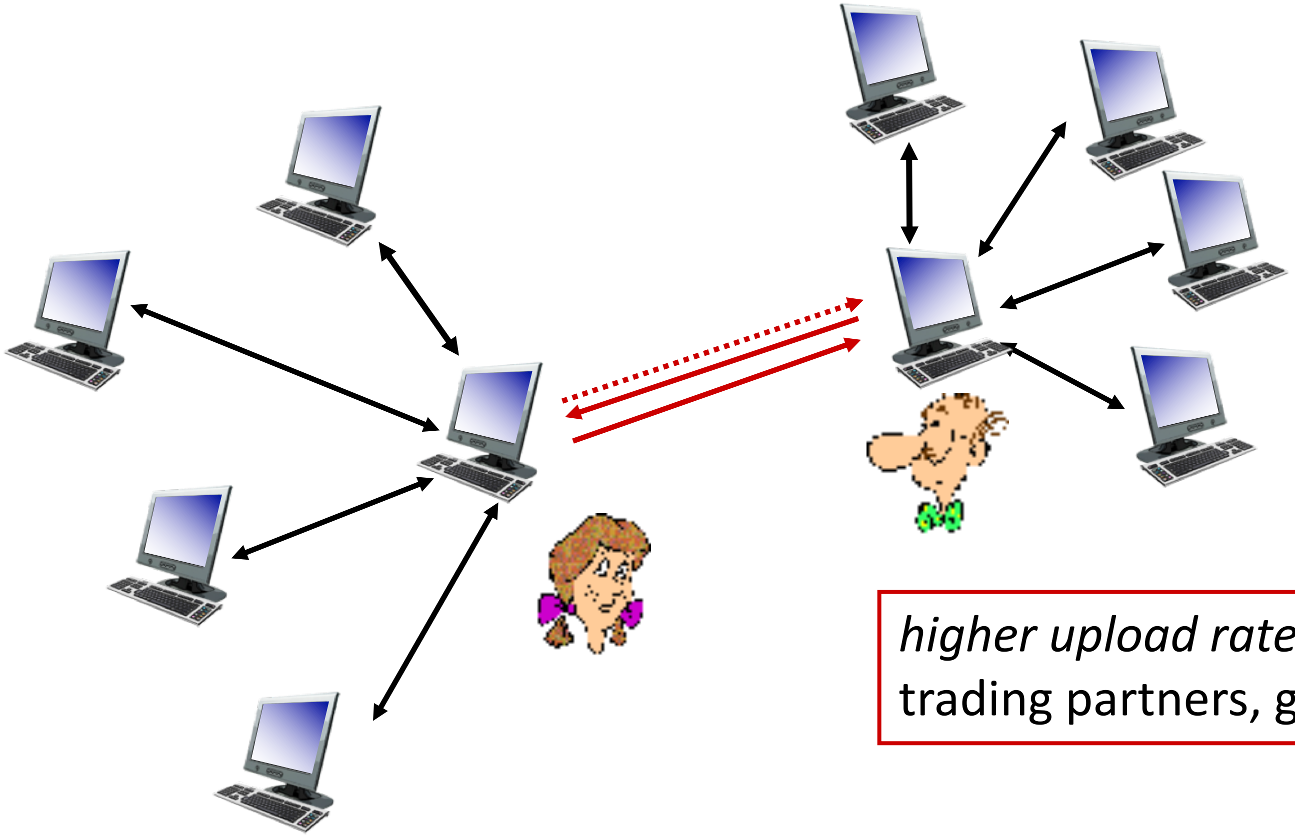
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 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

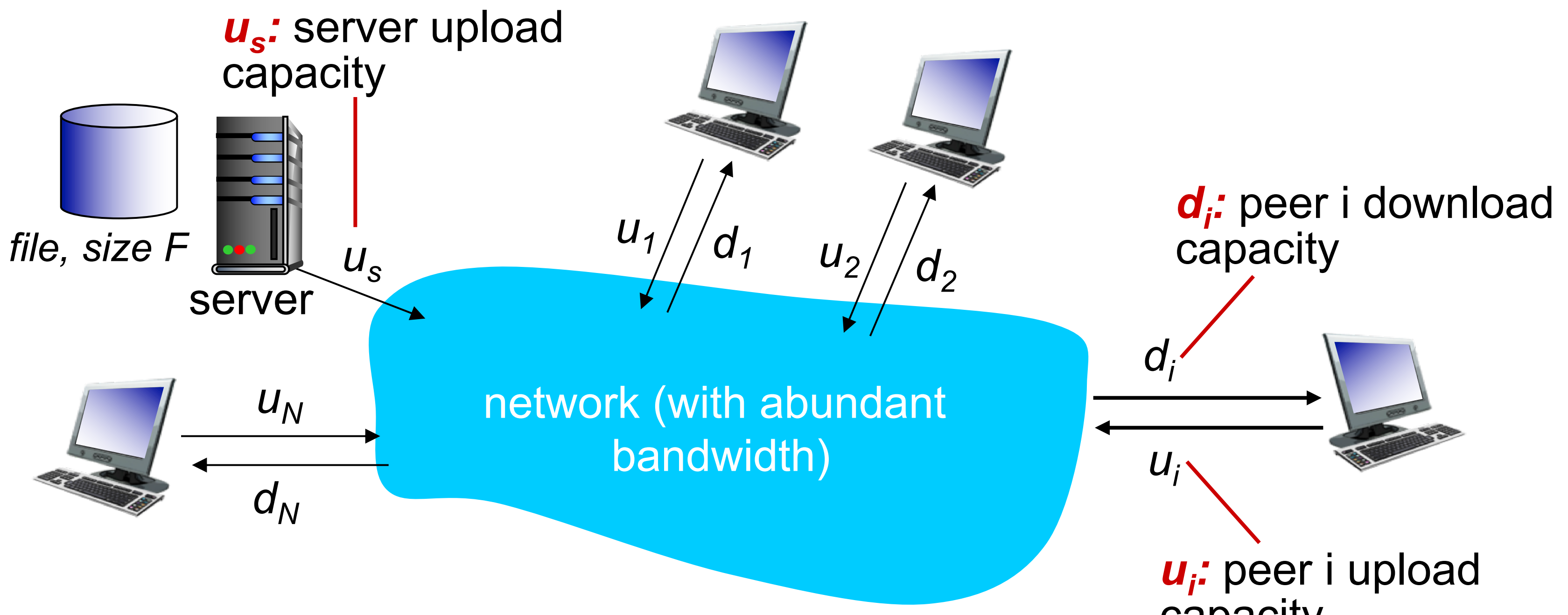


higher upload rate: find better trading partners, get file faster !

File distribution: client-server vs P2P

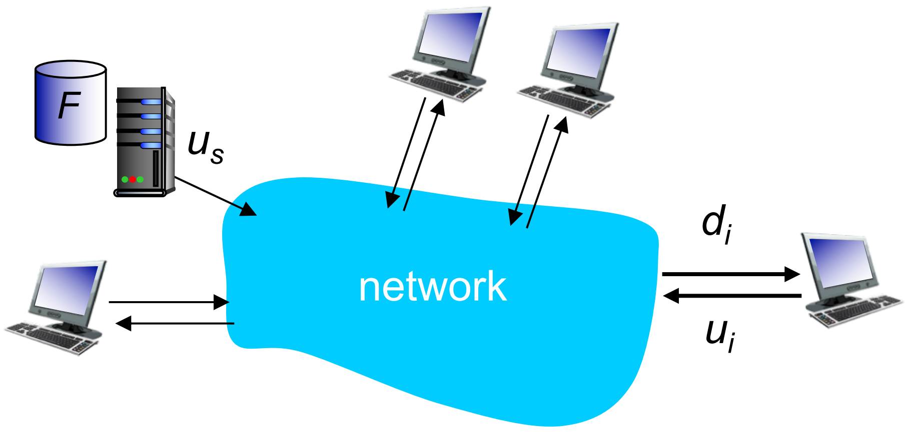
<u>Question</u>: how much time to distribute file (size *F*) from one server to *N* *peers*?

- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must sequentially send (upload) N file copies:
  - time to send one copy: F/u_s
  - time to send N copies: NF/u_s



- *client:* each client must download file copy
  - $d_{\min}$ = min client download rate
  - min client download time: $F/d_{\min}$

time to distribute F to N clients using client-server approach

$$D_{cs} \geq \max\left\{\frac{NF}{u_s}, \frac{F}{d_{\min}}\right\}$$

➢ The lower bound is achievable assuming a fluid model

File distribution time: client-server

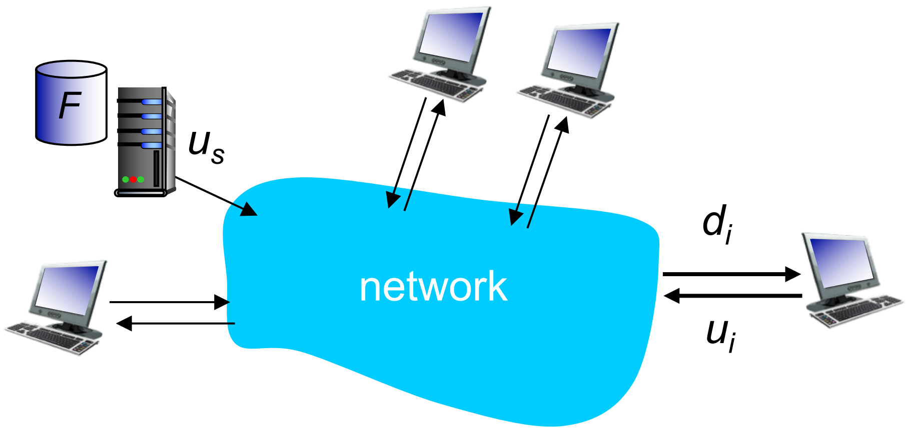
Proof (lower bound is achievable)

Case 1: $\frac{u_s}{N} \leq d_{\min}$. The server sends the file to each client, in parallel, at a rate of a rate of $\frac{u_s}{N}$

$\Rightarrow D = NF/u_s$

Case 2: $\frac{u_s}{N} \geq d_{\min}$. The server sends the file to each client, in parallel, at a rate of $d_{\min}$

$\Rightarrow D = F/d_{\min}$



time to distribute F to N clients using client-server approach

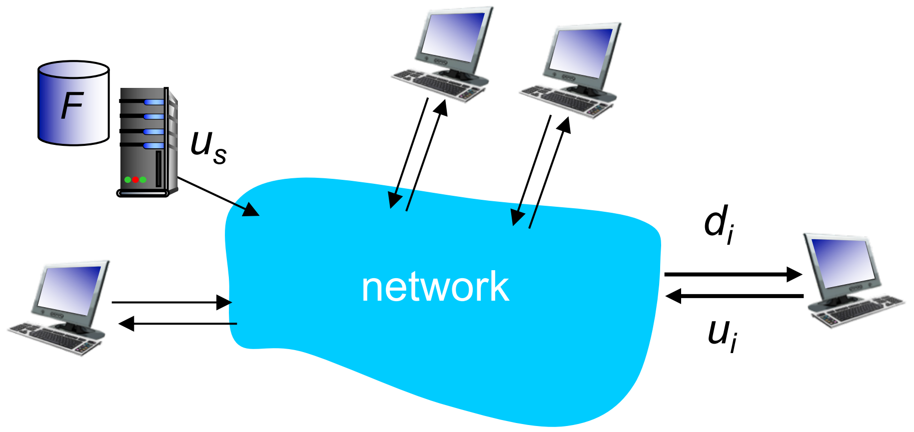
$$D_{cs} = \max\left\{\frac{NF}{u_s}, \frac{F}{d_{\min}}\right\}$$

increases linearly in N

File distribution time: P2P

- *server transmission:* must upload at least one copy
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: $F/d_{\min}$
- *clients:* as aggregate must download *NF* bits
  - max upload rate (limiting max download rate) is $u_s + \sum_{i=1}^{N} u_i$

➢ The lower bound is achievable assuming a fluid model



time to distribute F to N clients using P2P approach

$$D_{P2P} \geq \max\left\{\frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\}$$

increases linearly in *N* …

… but so does this, as each peer brings service capacity

File distribution time: P2P

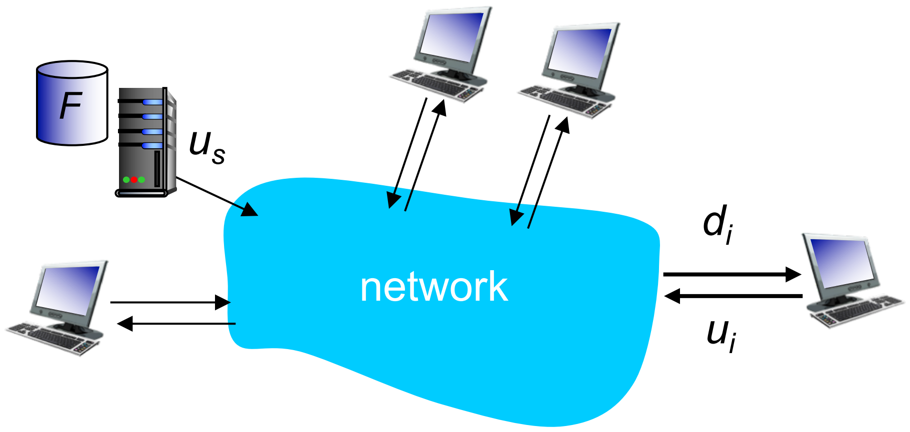
Exercise: prove that the lower bound is achievable. For simplicity, assume that $d_{\min}$ is very large so that peer download bandwidth is never a bottleneck.

Hint: design distribution schemes for the following two cases:

Case 1: $u_s \le \frac{u_s + \sum_{i=1}^{N} u_i}{N}$

Case 2: $u_s \ge \frac{u_s + \sum_{i=1}^{N} u_i}{N}$

It may help to start with the simple setting with one server and two identical peers.



time to distribute F to N clients using P2P approach

$$D_{P2P} = \max\left\{\frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\}$$

Structured P2P File Sharing

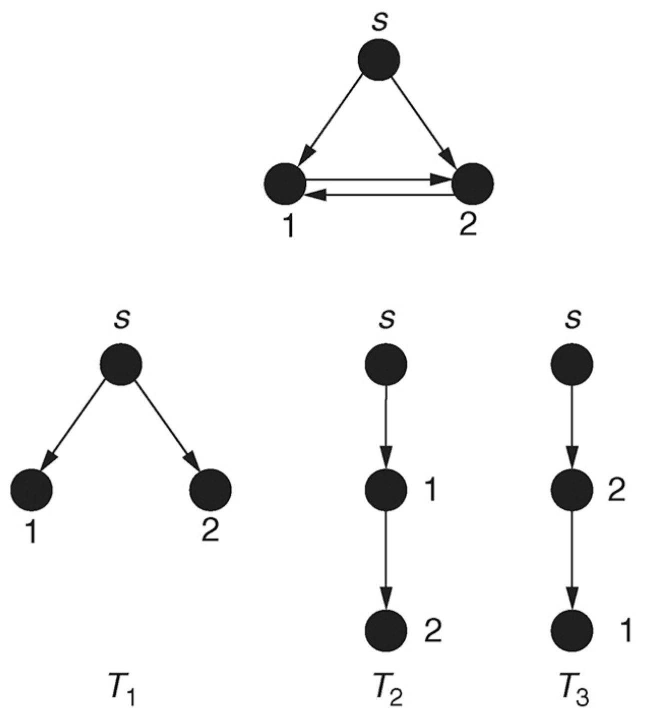
- One source node s, N other nodes.
- File sharing using a set of spanning trees rooted at s
  - spanning trees: $T_1, T_2, \dots, T_k$
  - r_t: rate transmitted over tree T_t
- Optimization formulation

$$\max \sum_t r_t$$

$$\text{s.t.} \quad \sum_t r_t \le d_i \qquad \forall i \in \{1,2,\dots,N\}$$

$$\sum_t c_t(j)\, r_t \le u_j \quad \forall j \in \{s\} \cup \{1,2,\dots N\}$$

$c_t(j)$: number of children of node j in tree t



The three spanning trees in a three-node network

Structured P2P File Sharing

Theorem: $D_{P2P} = \max\left\{\frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\}$

Proof :

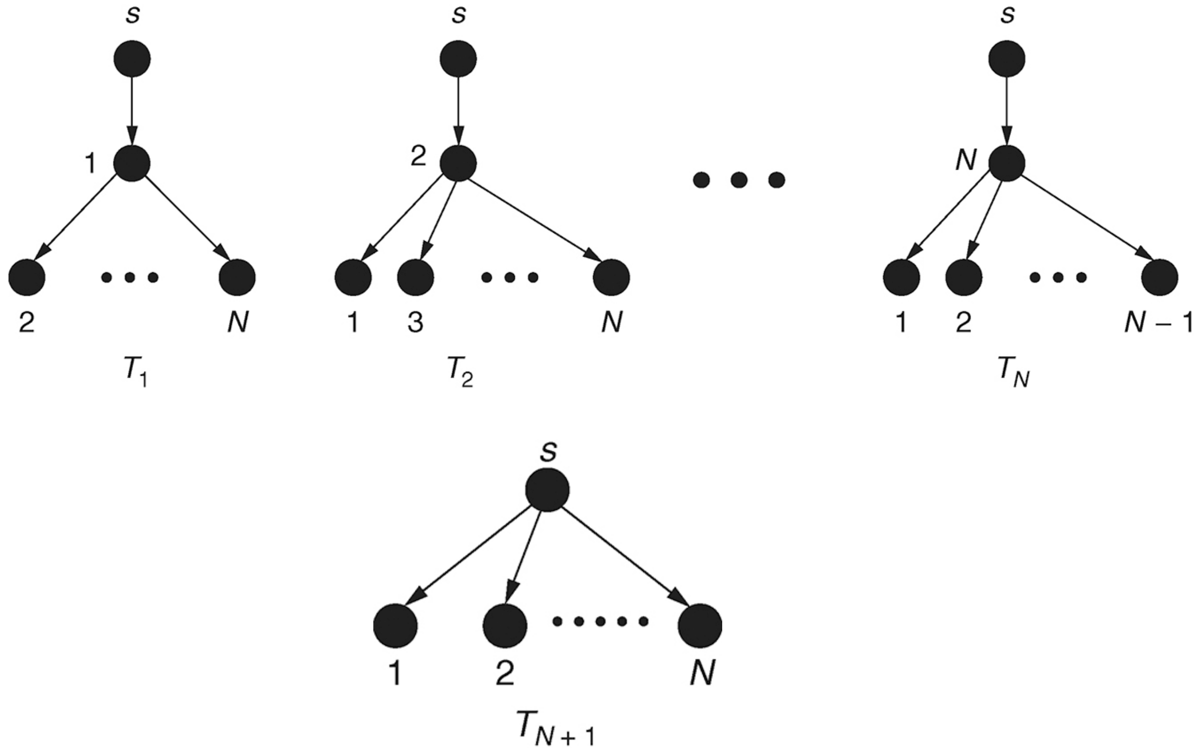
Case 1: $u_s \le \min\left(d_{\min}, \frac{u_s + \sum_{i=1}^{N} u_i}{N}\right)$

Consider N spanning trees with $r_i = \frac{u_i}{\sum_{j=1}^{N} u_j} u_s$

Case 2: $\frac{u_s + \sum_{i=1}^{N} u_i}{N} \le \min(d_{\min}, u_s)$

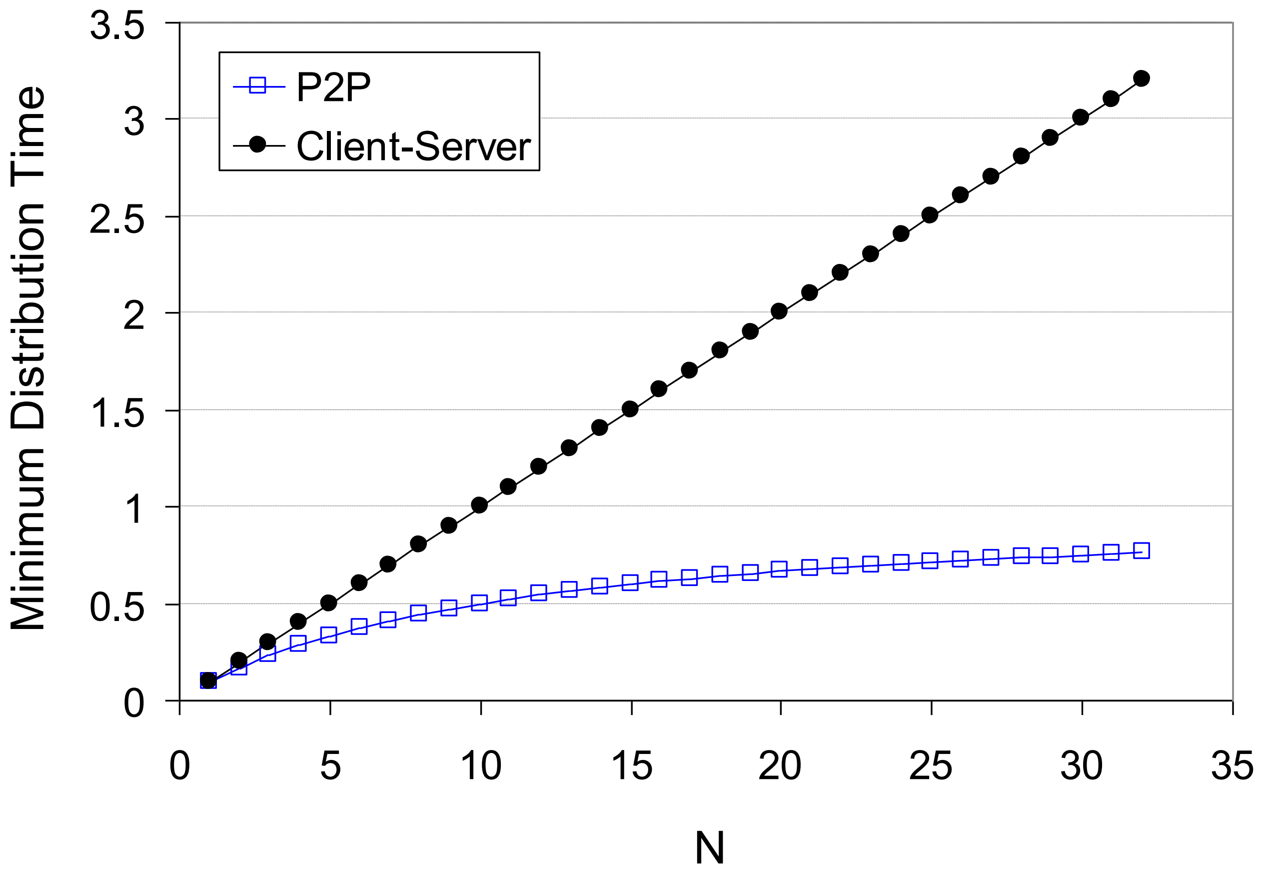
Consider $N+1$ spanning trees with $r_i = \frac{u_i}{N-1}$ for $i = 1, \ldots, N$, $r_{N+1} = \frac{1}{N}\left(u_s - \frac{\sum_{j=1}^{N} u_j}{N-1}\right)$

Case 3: $d_{\min} \le \min\left(u_s, \frac{u_s + \sum_{i=1}^{N} u_i}{N}\right)$ (see [Srikant and Ying 8.3])



Client-server vs. P2P: example

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



Chapter 2: summary

- 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
  - DNS
  - P2P
- socket programming:

  TCP, UDP sockets