CHAPTER 2 - APPLICATION LAYER

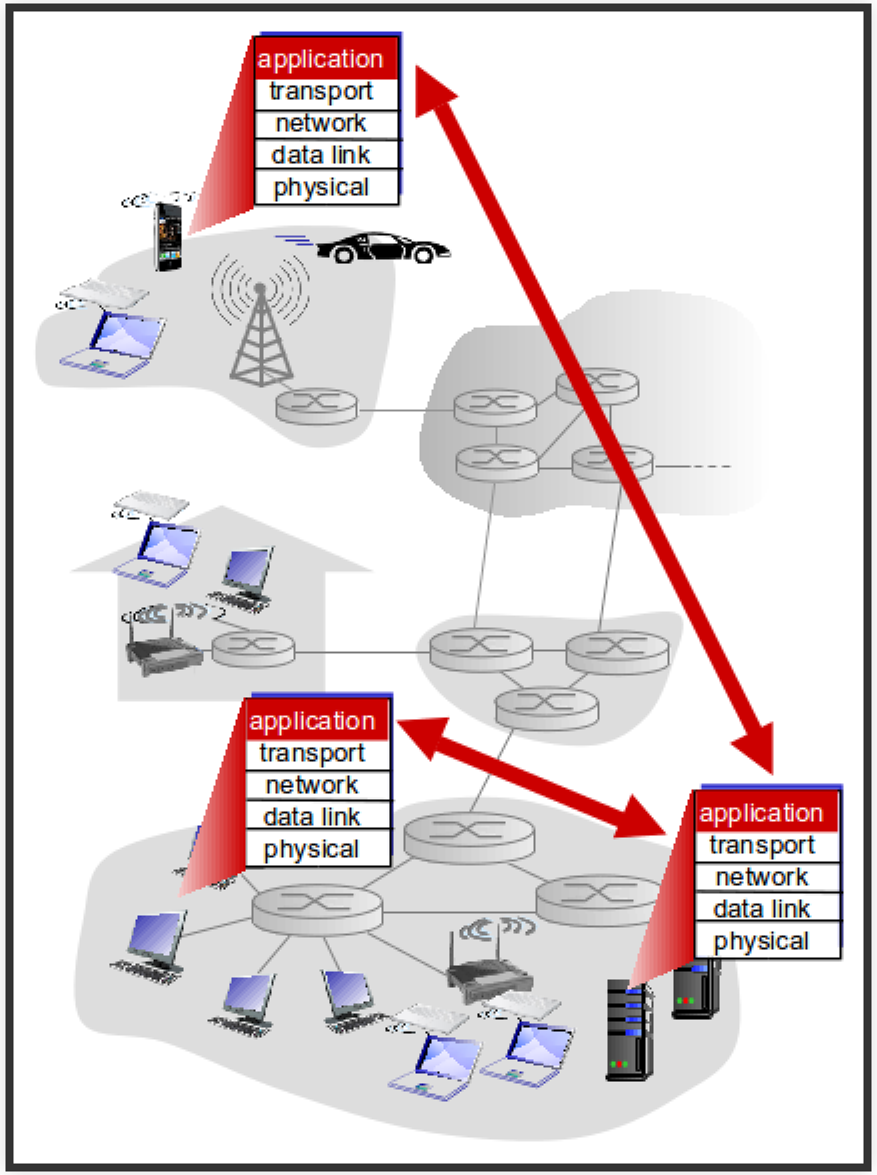
GOALS

PRINCIPLES OF NETWORK APPLICATIONS

SOME NETWORK APPLICATIONS

- e-mail & web
- 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
- search
- …

CREATING A NETWORK APP



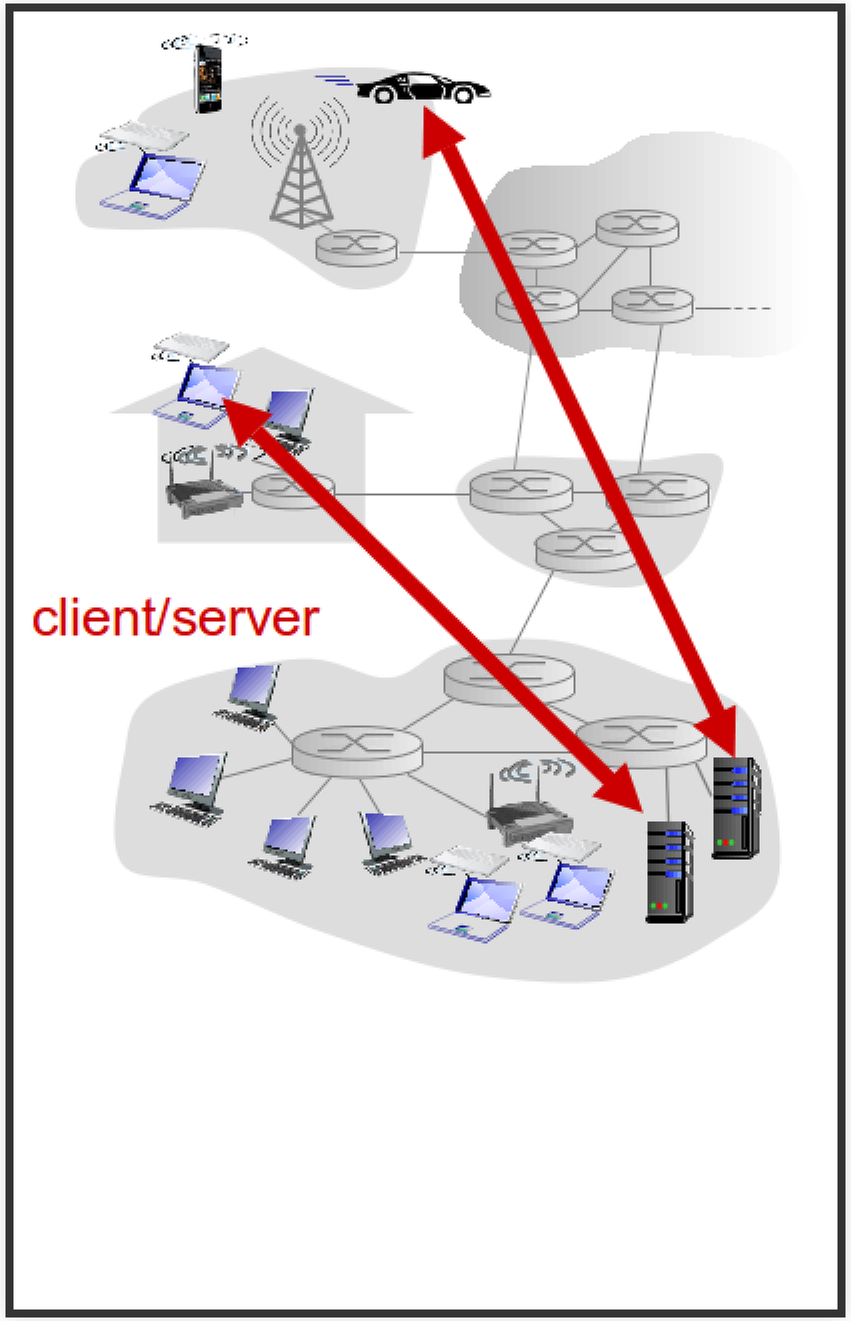
CREATING A NETWORK APP

- 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

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

CLIENT-SERVER ARCHITECTURE



CLIENT-SERVER ARCHITECTURE

server

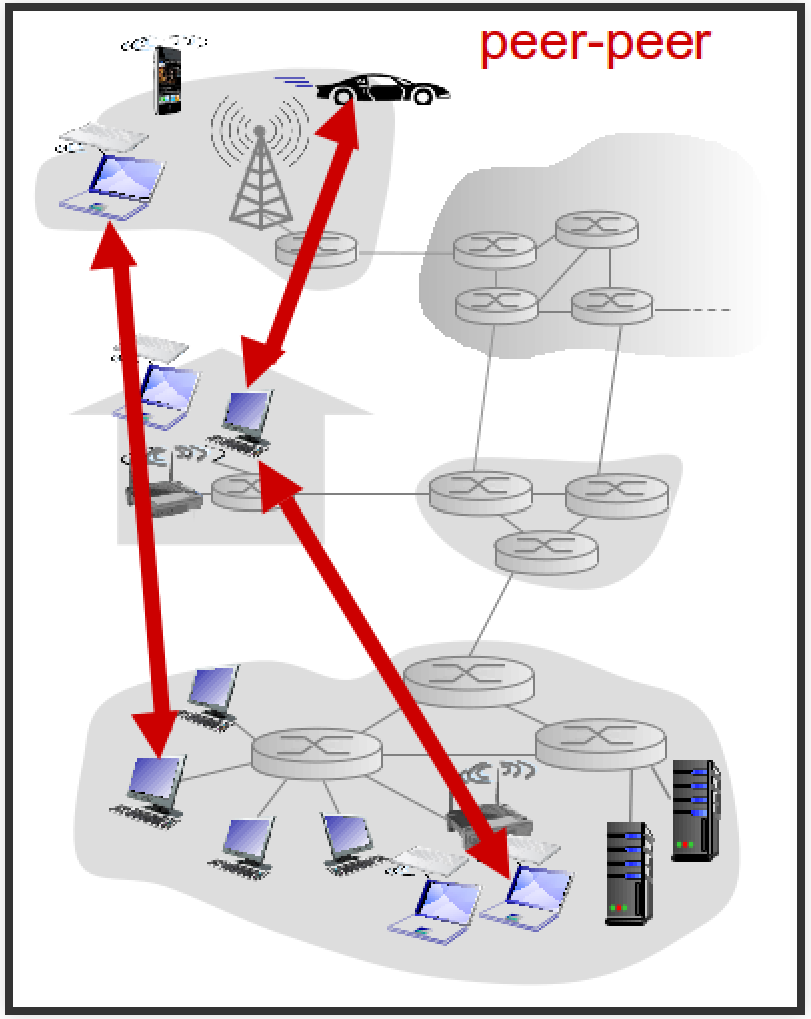
- always-on host
- permanent IP address
- data centers for scaling

CLIENT-SERVER ARCHITECTURE

clients

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

P2P ARCHITECTURE



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

PROCESSES COMMUNICATING

client process

process that initiates communication

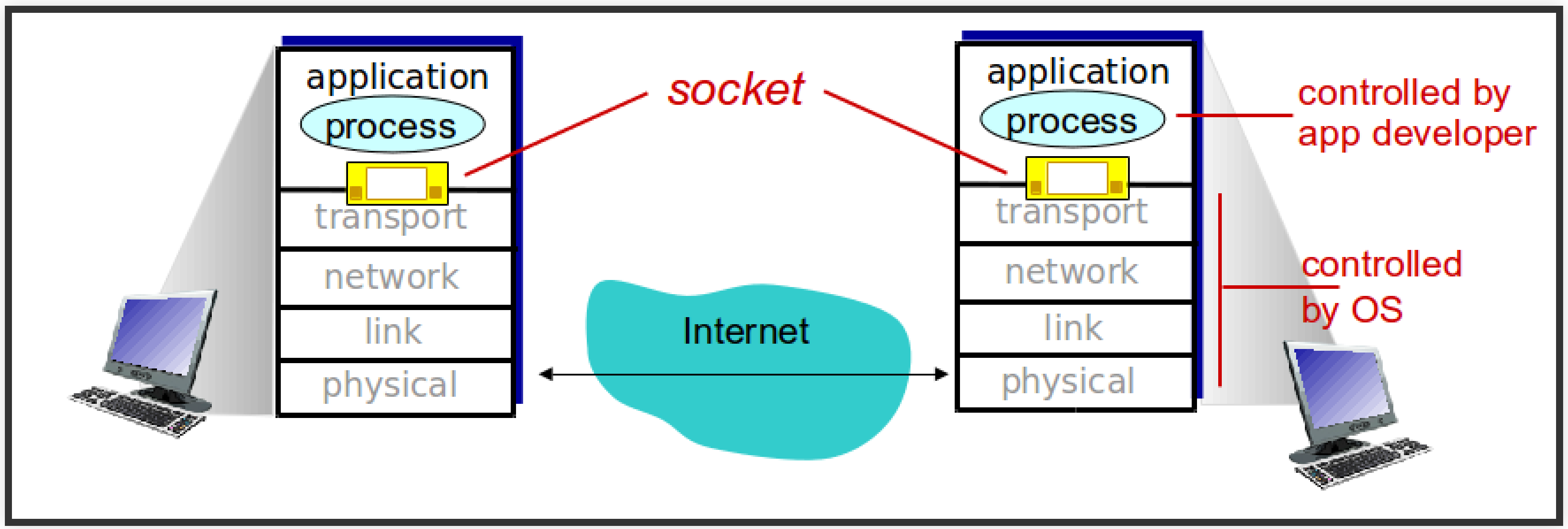
server process

process that waits to be contacted

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

SOCKETS

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



ADDRESSING PROCESSES

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

EXAMPLE PORT NUMBERS:

- HTTP server: 80
- mail server: 25
  - to send HTTP message to gaia.cs.umass.edu web server:
- IP address: 128.119.245.12
- port number: 80
  - more shortly…

APP-LAYER PROTOCOL DEFINES

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

PROTOCOL TYPES

- Open protocols:
  - defined in RFCs
  - allows for interoperability
  - e.g., HTTP, SMTP
- Proprietary protocols:
  - e.g., Skype

TRANSPORT SERVICE AN APP NEEDS?

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

TRANSPORT SERVICE AN APP NEEDS?

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

TRANSPORT SERVICE AN APP NEEDS?

- **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

TRANSPORT SERVICE AN APP NEEDS?

- **Security**
  - encryption, data integrity, …

TRANSPORT SERVICE REQUIREMENTS

Application	Data loss	Throughput	Time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no

TRANSPORT SERVICE REQUIREMENTS

Application	Data loss	Throughput	Time sensitive
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps, video:10kbps-5Mbps	yes, 100’s msec
stored audio/video	loss-tolerant	same as above	yes, few secs

TRANSPORT SERVICE REQUIREMENTS

Application	Data loss	Throughput	Time sensitive
interactive games	loss-tolerant	few kbps up	yes, 100's msec
text messaging	no loss	elastic	yes and no

INTERNET TRANSPORT PROTOCOLS SERVICES

TCP SERVICE

- **Reliable transport** between sending and receiving process
- **Flow control**: sender won't overwhelm receiver
- **Congestion control**: throttle sender when network overloaded
- Does not provide: timing, minimum throughput guarantee, security
- **Connection-oriented**: setup required between client and server processes

UDP SERVICE

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

> Q: why bother? - Why is there a UDP?

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

APPLICATION, TRANSPORT PROTOCOLS

Application	Application layer protocol	underlying transport protocol
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

SECURING TCP

- TCP & UDP
  - no encryption
  - cleartext passwords sent into socket traverse Internet in cleartext

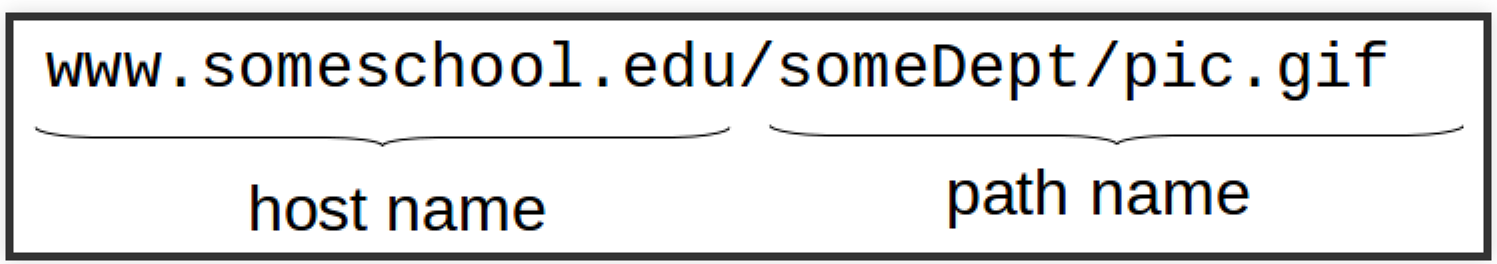
SECURING TCP

- SSL (TLS)
  - provides encrypted TCP connection
  - data integrity
  - end-point authentication
- SSL is at app layer
  - Apps use SSL libraries, which "talk" to TCP
- SSL socket API
  - cleartext passwds sent into socket traverse Internet encrypted
  - We cover this in chapter 8

WEB AND HTTP

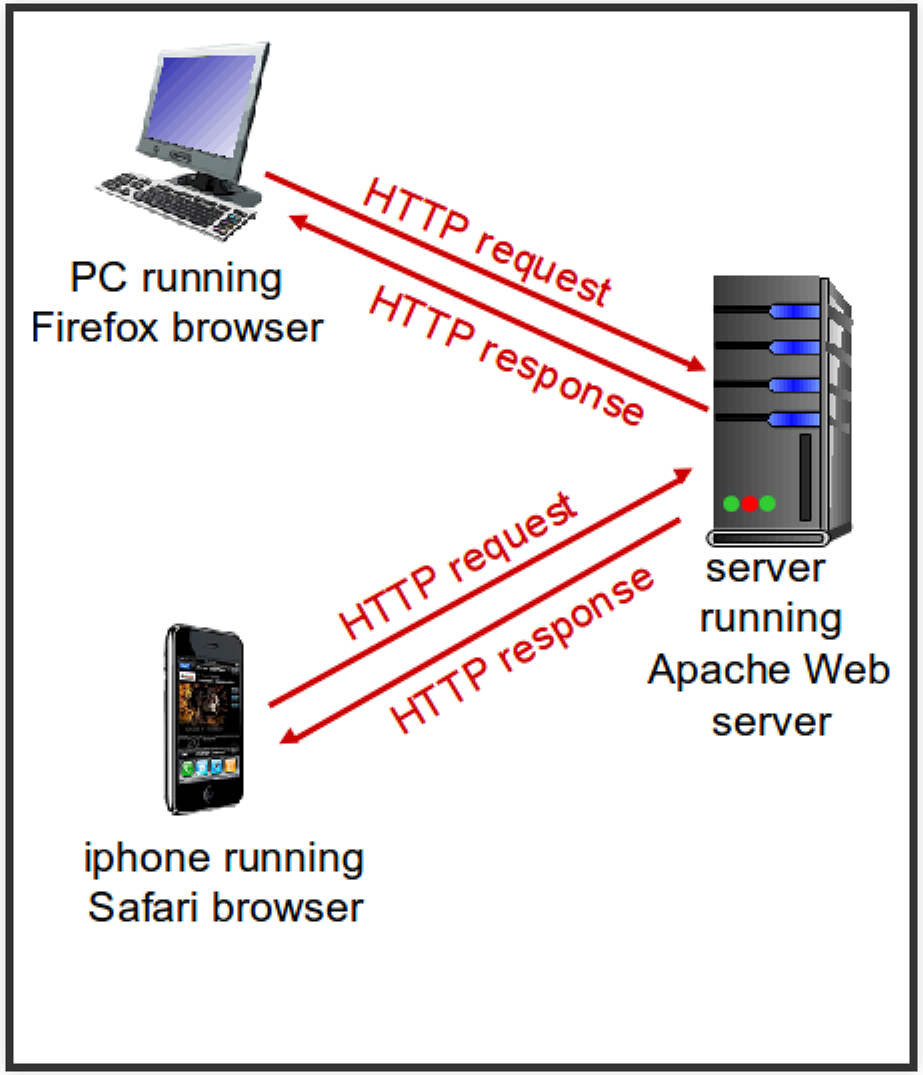
First, a quick intro…

- 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.,



HTTP OVERVIEW

HTTP: hypertext transfer protocol



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

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

HTTP OVERVIEW

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

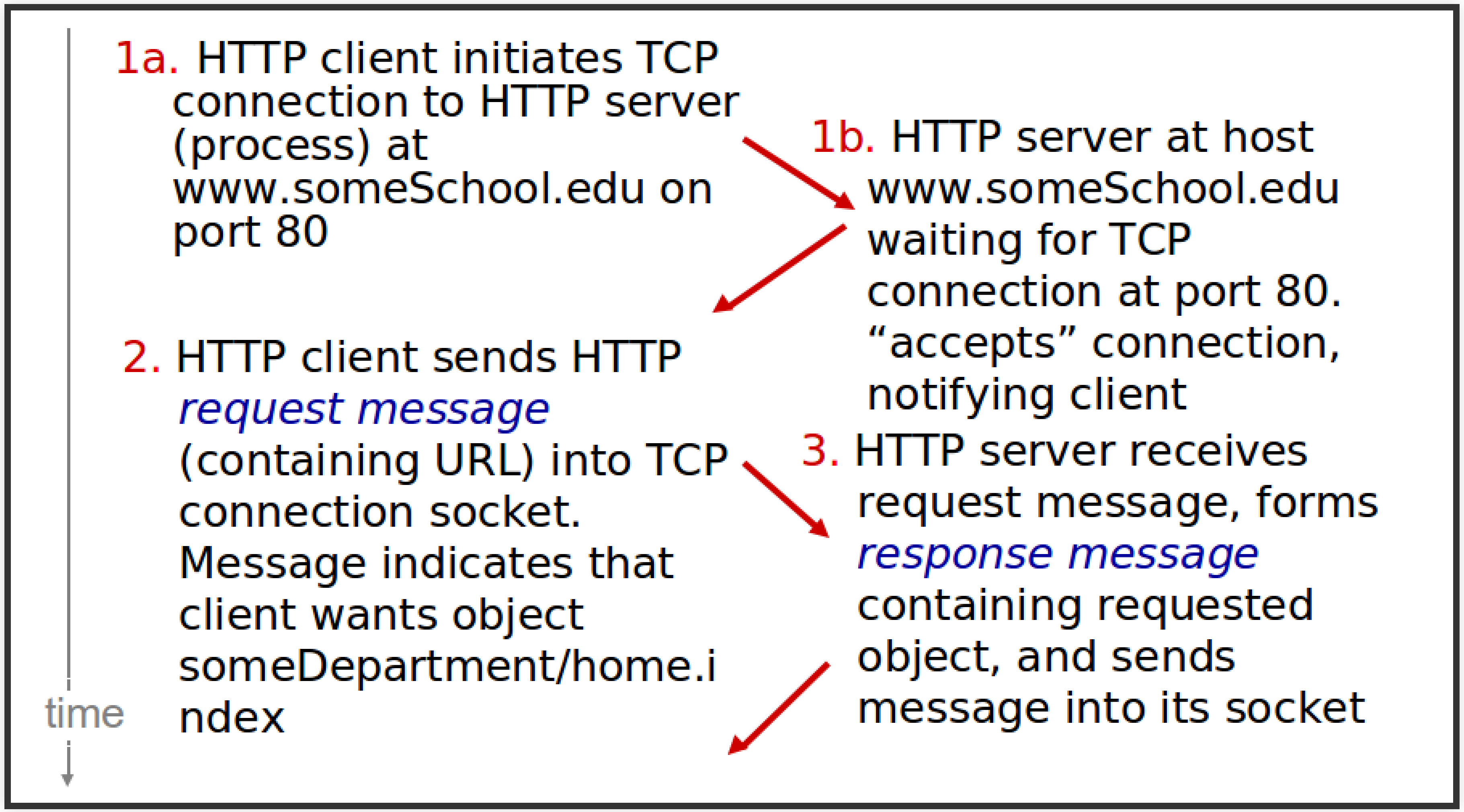
- multiple objects can be sent over single TCP connection between client, server

NON-PERSISTENT HTTP

suppose user enters URL:

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

(contains text, references to 10 jpeg images)



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

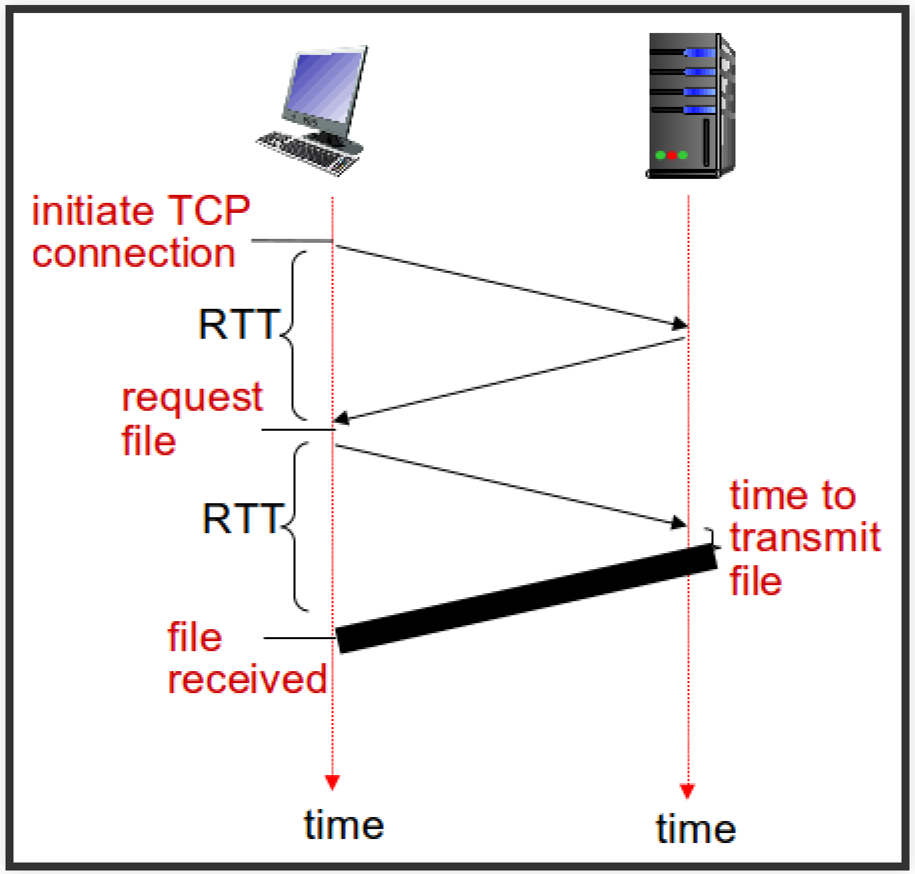
time

NON-PERSISTENT HTTP: RESPONSE TIME

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

- HTTP response time
  - one RTT to initiate TCP connection
  - one RTT for HTTP request and first few bytes of HTTP response to return
  - file transmission time
  - non-persistent HTTP response time = 2RTT + file transmission time

NON-PERSISTENT HTTP: RESPONSE TIME



NON-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

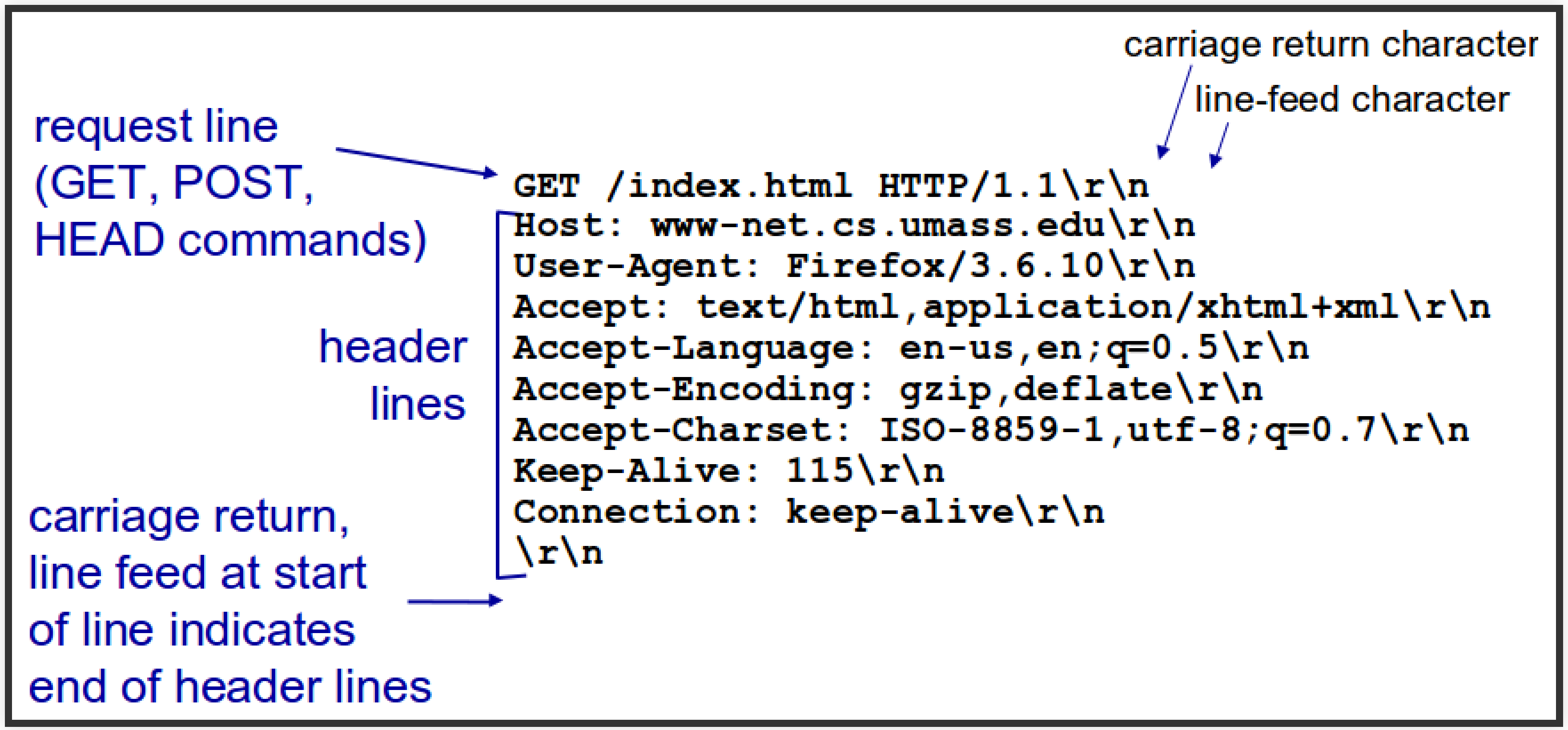
PERSISTENT HTTP

Persistent HTTP:

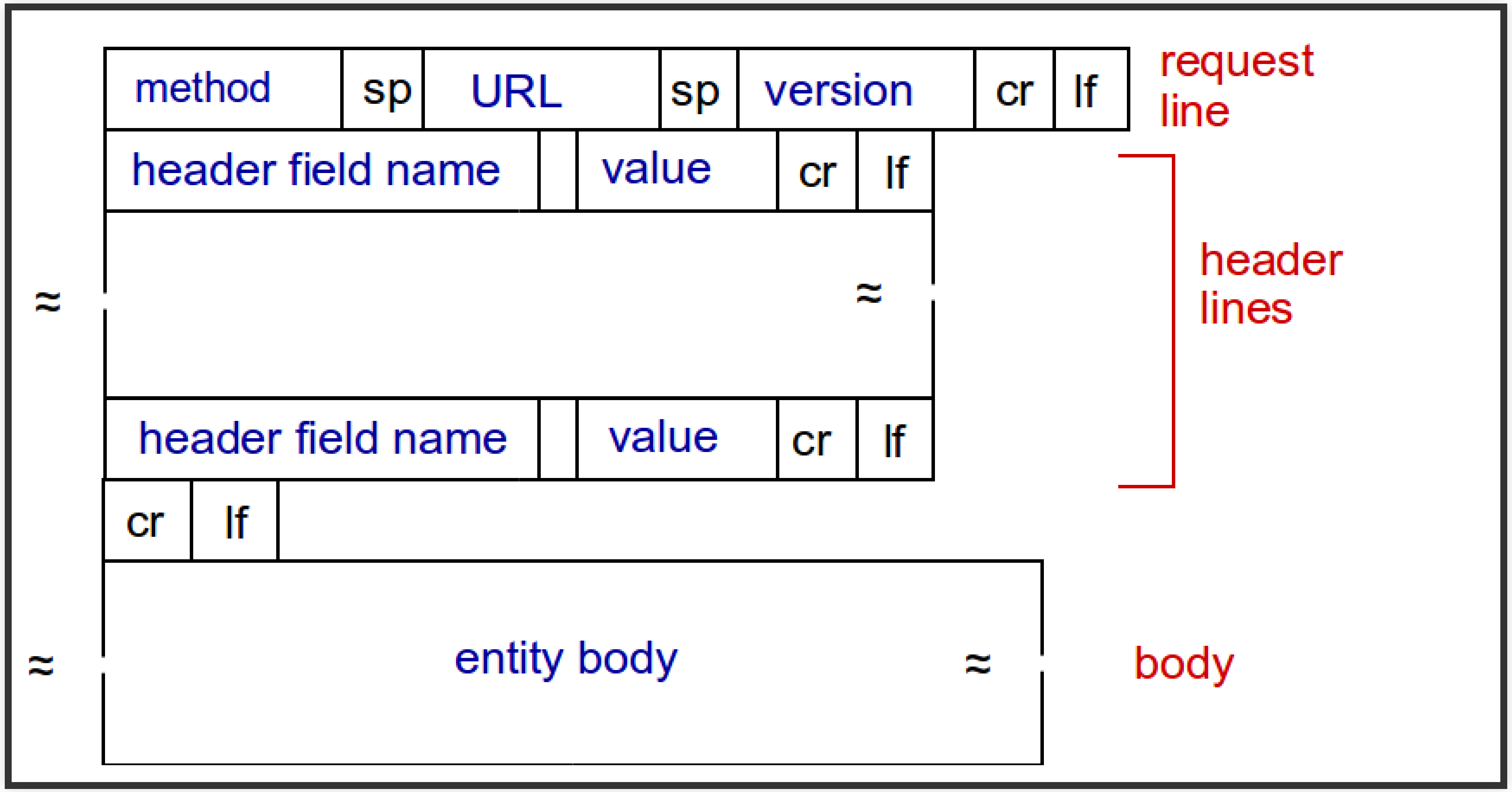
- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP REQUEST MESSAGE

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



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=4&banana=2`

METHOD TYPES

HTTP/1.0:

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

METHOD TYPES

HTTP/1.1:

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

HTTP RESPONSE MESSAGE

status line (protocol status code status phrase)

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

header lines

data, e.g., requested HTML file

HTTP RESPONSE STATUS CODES

- 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 RESPONSE STATUS CODES

400’s: You fucked up

500’s: We fucked up

Take a look at code 418

TRYING OUT HTTP (CLIENT SIDE) FOR YOURSELF

- Telnet to your favorite Web server: `telnet imada.sdu.dk 80`
  opens TCP connection to port 80 (default HTTP server port) at imada.sdu.dk.
  anything typed in sent to port 80 at imada.sdu.dk
- type in a GET HTTP request:
  ```
  GET /~jamik/ HTTP/1.1
  Host: imada.sdu.dk
  ```
  by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server
- look at response message sent by HTTP server!

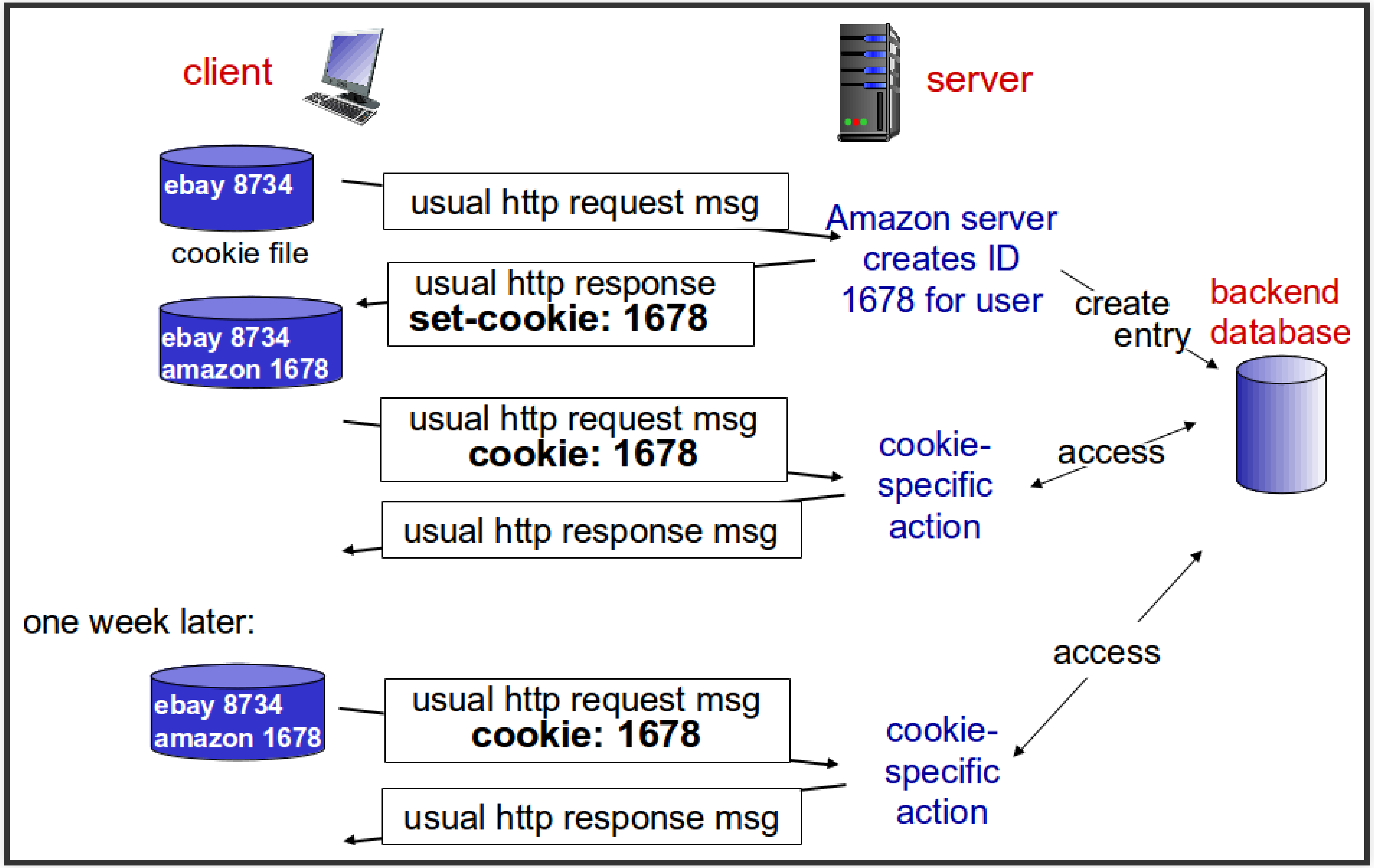
USER-SERVER STATE: COOKIES

many Web sites use cookies

four components:

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

COOKIES: KEEPING "STATE"



COOKIES (CONTINUED)

What cookies can be used for:

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

HOW TO KEEP "STATE"

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

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

HTTP/2

- The HTTP/2 specification was published as RFC 7540 in May 2015
- Most major browsers added HTTP/2 support by the end of 2015
- Last update: HTTP 1.1 was in 1997
- Material from RFC 7540 and https://en.wikipedia.org/wiki/HTTP/2

GOALS FOR HTTP/2

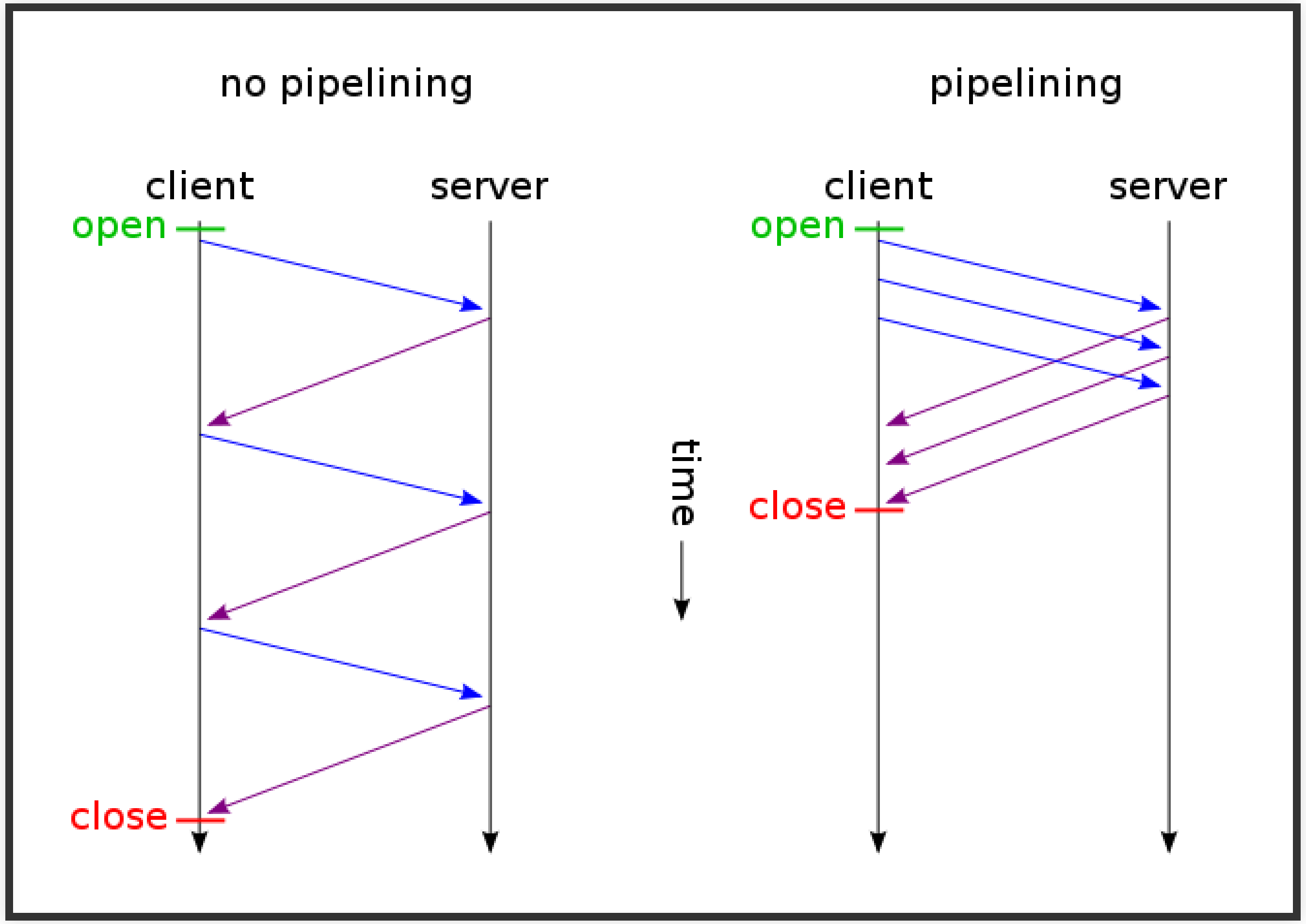
- Negotiation mechanism that allows clients and servers to elect to use HTTP 1.1, 2.0, or potentially other non-HTTP protocols.
- Maintain high-level compatibility with HTTP 1.1 (for example with methods, status codes, and URIs, and most header fields)
- Decrease latency to improve page load speed in web browsers by considering:
  - Data compression of HTTP headers
  - Pipelining of requests
  - Fixing the head-of-line blocking problem in HTTP 1.x
  - Multiplexing multiple requests over a single TCP connection

HEAD-OF-LINE BLOCKING

An issue for HTTP 1.0 and HTTP 1.1

- Requests must be treated in order
- Means that requests can wait behind a slow or large request
- HTTP 1.1 introduced pipelining to partially fix the issue
- Many clients also tried to create multiple connections to increase speed (unfair to other better behaved applications)

PIPELINING



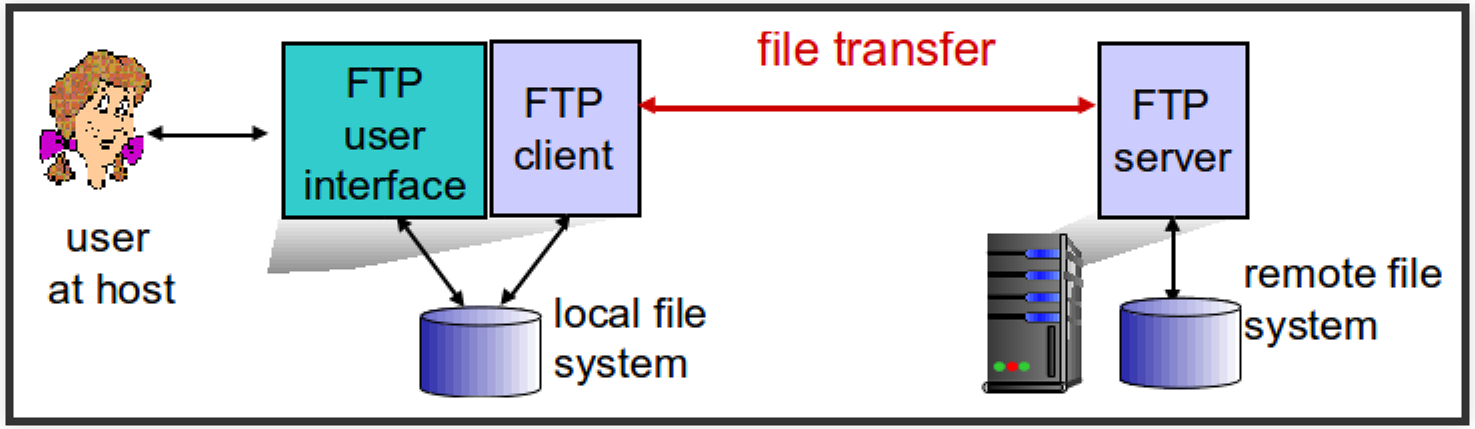
no pipelining
client
server
open
close
time
pipelining
client
server
open
close

STREAMS AND MULTIPLEXING

- All communication in HTTP/2 is done through streams
- The client can create streams with odd numbers 1, 3, 5 and so on
- The server can create streams with even numbers 2, 4, 6 and so on
- 0 is used for connection control messages
- Even if one stream is blocked waiting for a slow or large request the others can still carry on

FTP

FILE TRANSFER PROTOCOL

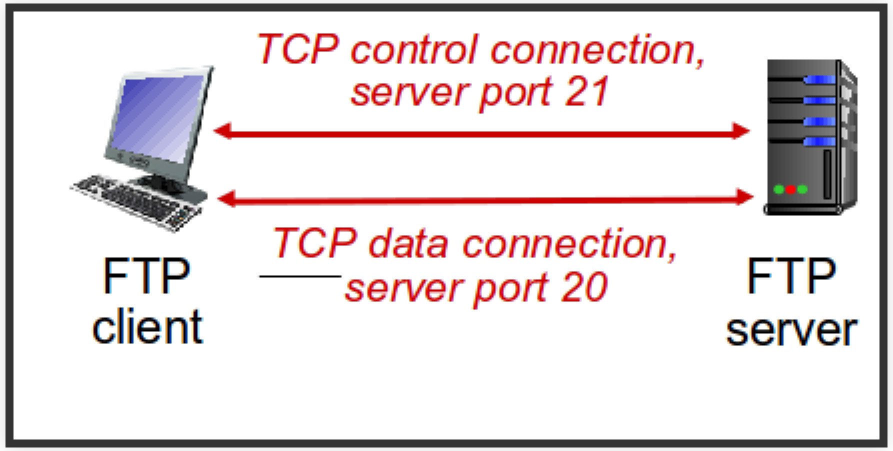


- transfer file to/from remote host
- client/server model
  - **client:** side that initiates transfer (either to/from remote)
  - **server:** remote host
- ftp: RFC 959
- ftp server: port 21

SEPARATE CONTROL, DATA CONNECTIONS

- FTP client contacts FTP server at port 21, using TCP
- client authorized over control connection
- client browses remote directory, sends commands over control connection
- when server receives file transfer command, server opens 2nd TCP data connection (for file) to client
- after transferring one file, server closes data connection

SEPARATE CONTROL, DATA CONNECTIONS



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

FTP COMMANDS, RESPONSES

sample commands:

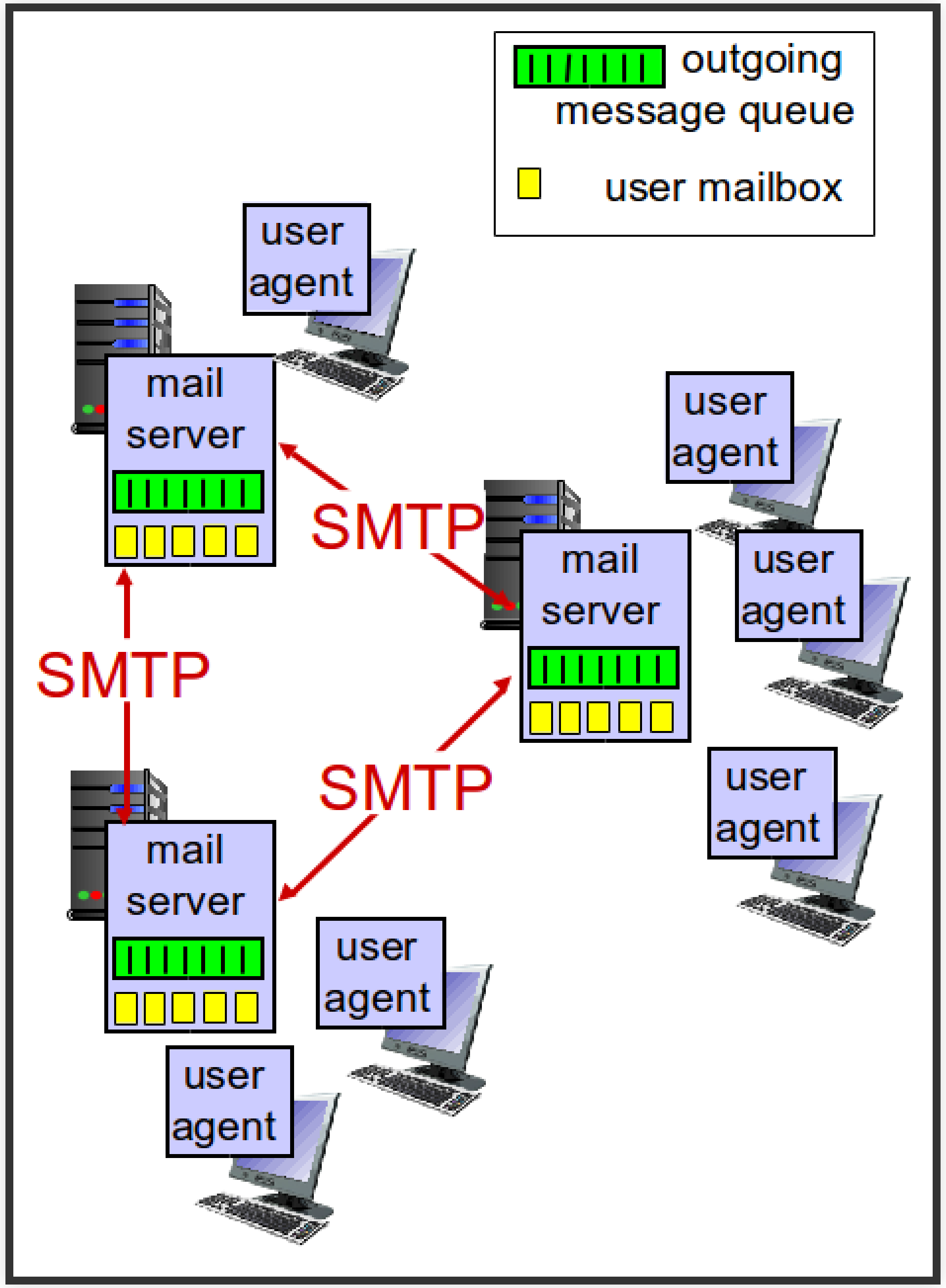
- sent as ASCII text over control channel
- `USER username`
- `PASS password`
- `LIST` return list of file in current directory
- `RETR filename` retrieves (gets) file
- `STOR filename` stores (puts) file onto remote host

FTP COMMANDS, RESPONSES

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`

ELECTRONIC MAIL



ELECTRONIC MAIL

Three major components:

1. user agents
2. mail servers
3. simple mail transfer protocol: SMTP

USER AGENT

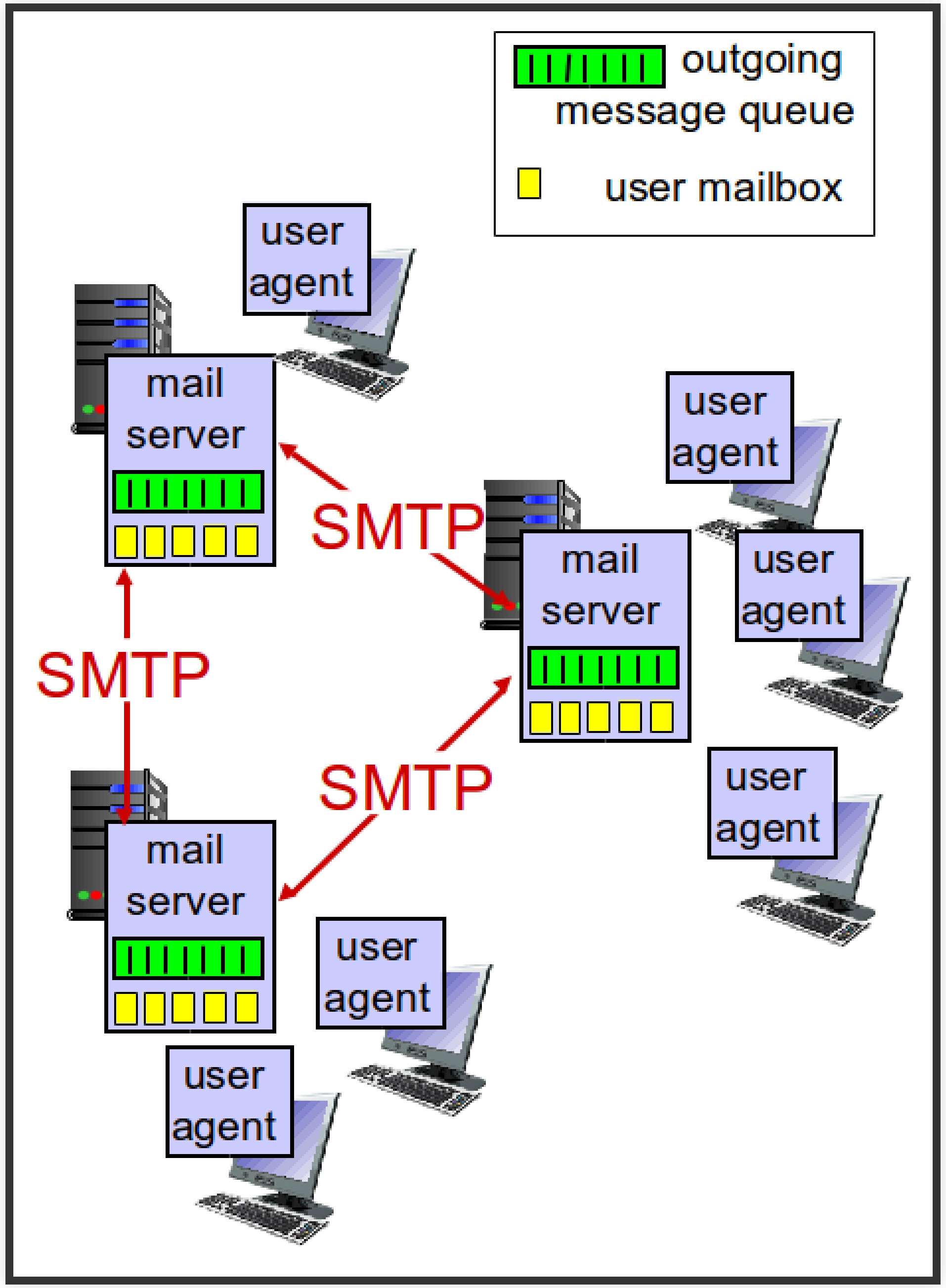
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server

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

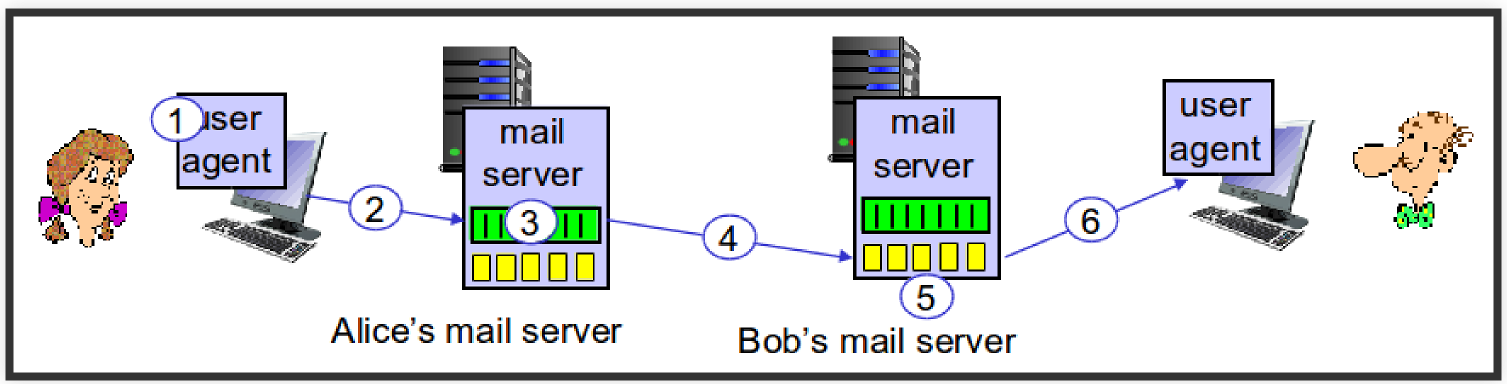
MAIL SERVERS



SMTP [RFC 2821]

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

```
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

SMTP: FINAL WORDS

comparison with HTTP:

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

MAIL MESSAGE FORMAT

SMTP: protocol for exchanging email msgs

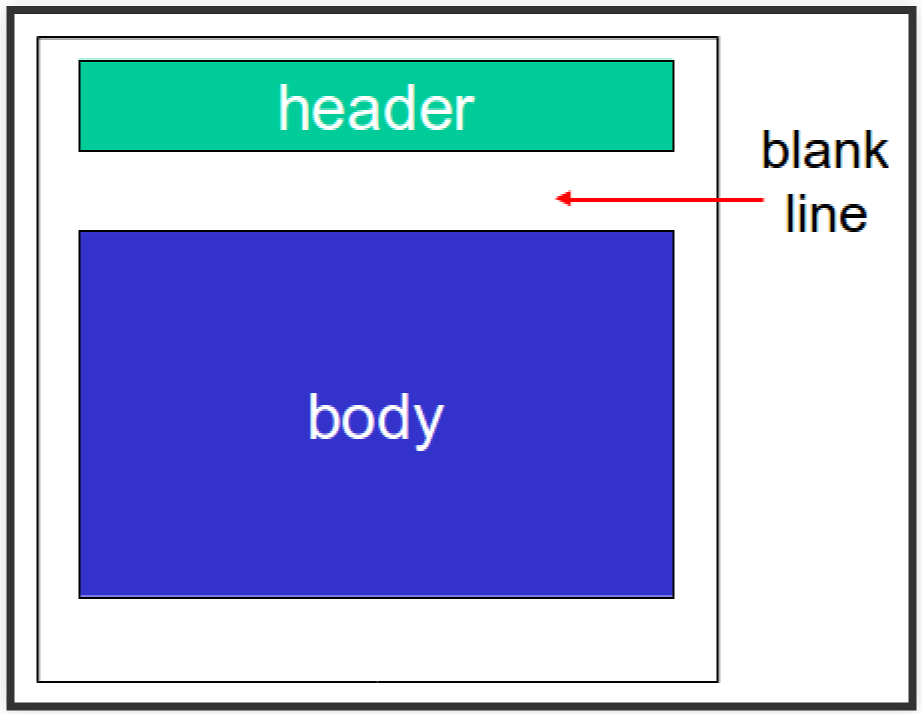
RFC 822: standard for text message format:

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

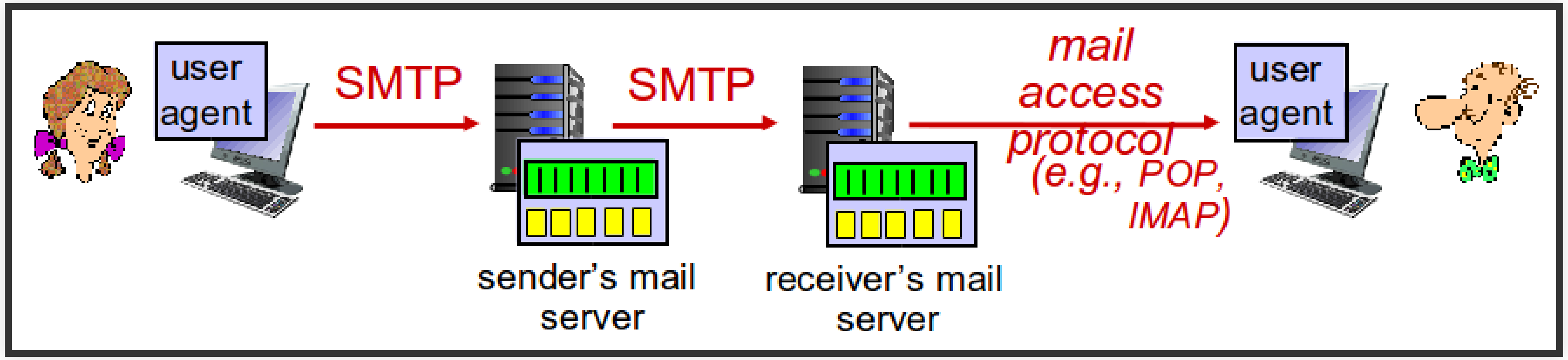
different from SMTP MAIL FROM, RCPT TO: commands!

- Body: the “message”
  - ASCII characters only

MAIL MESSAGE FORMAT



MAIL ACCESS PROTOCOLS



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

POP3 PROTOCOL

Authorization phase

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

POP3 PROTOCOL

Transaction phase, client:

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

POP3 PROTOCOL

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 (MORE)

More about POP3

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

IMAP

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

DNS

DOMAIN NAME SYSTEM

- **people:** many identifiers:
  SSN, name, passport number
- **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

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

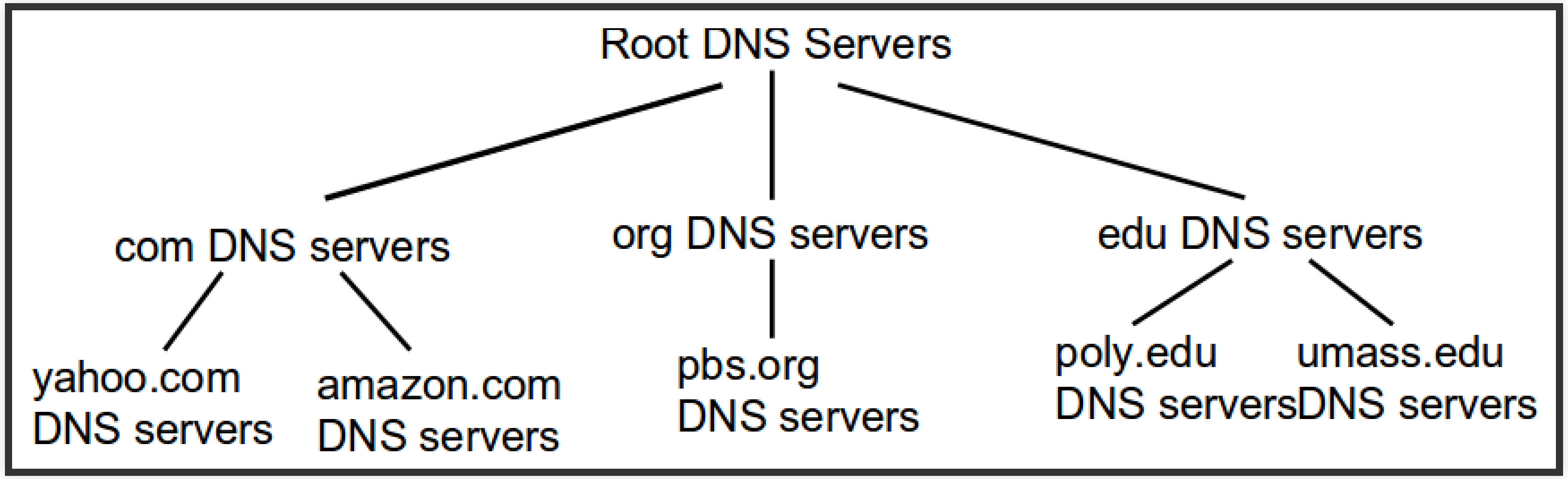
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

DNS: SERVICES, STRUCTURE

Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance
- A: **doesn’t scale!**

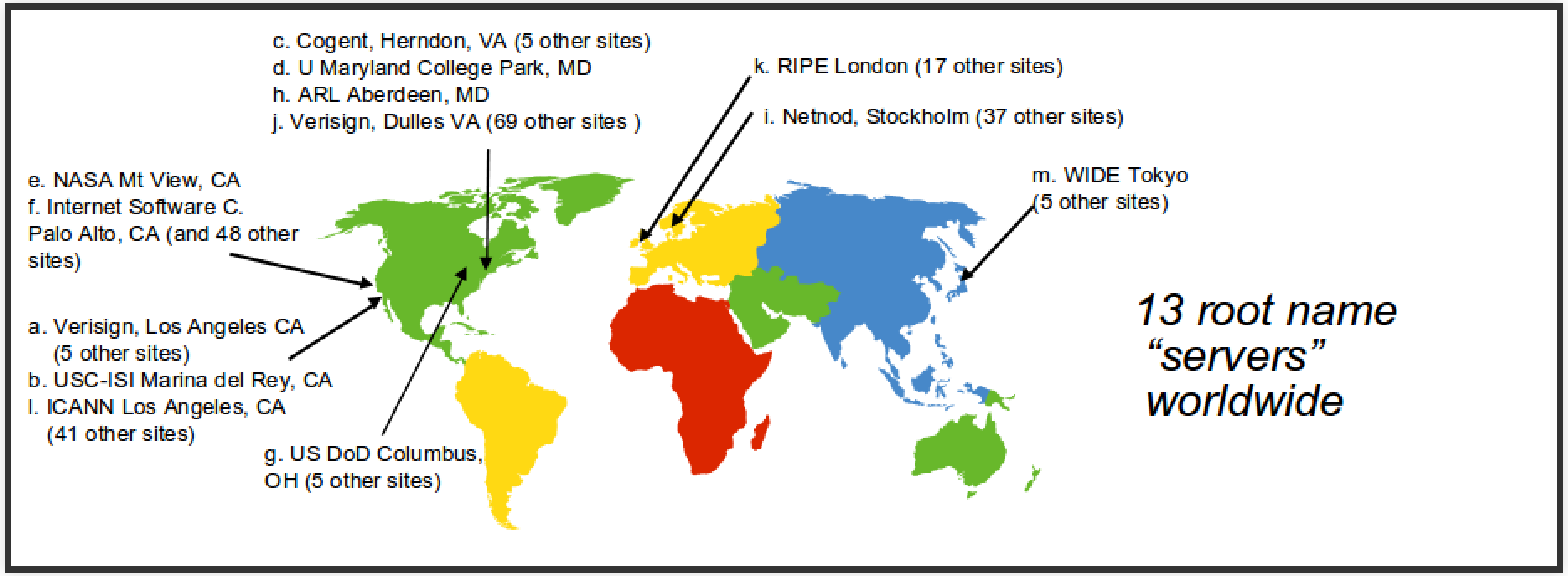
DNS: A DISTRIBUTED, HIERARCHICAL DATABASE



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

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

DNS: ROOT NAME SERVERS



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

TLD 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 SERVERS

❗ 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

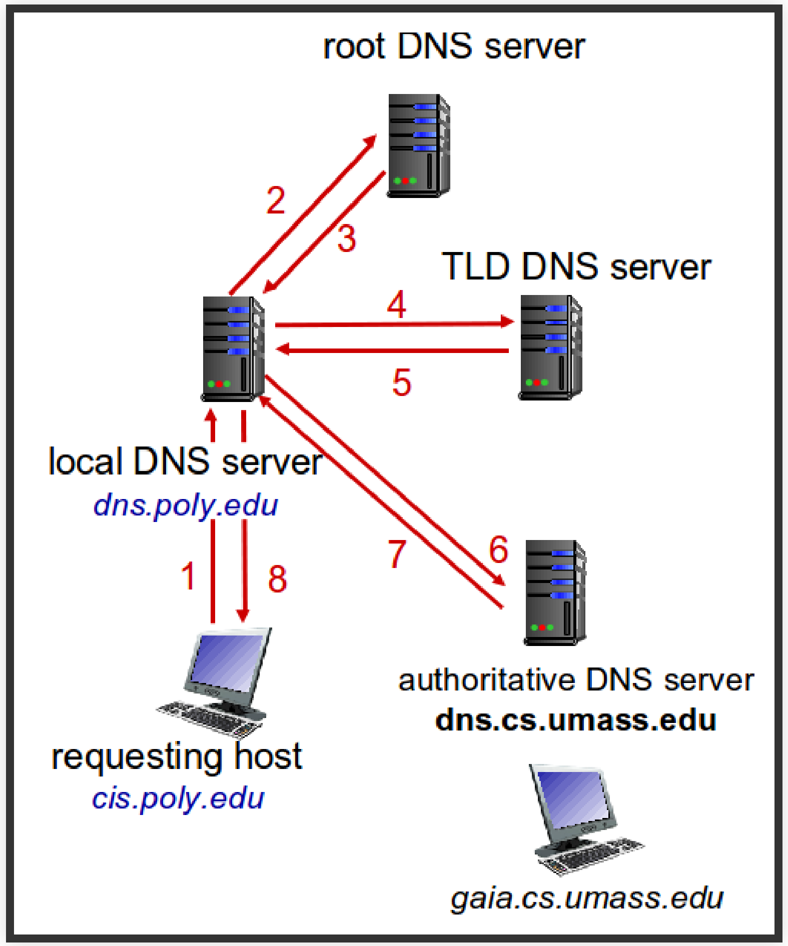
> Is there security considerations with a local DNS server

DNS NAME RESOLUTION EXAMPLE

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

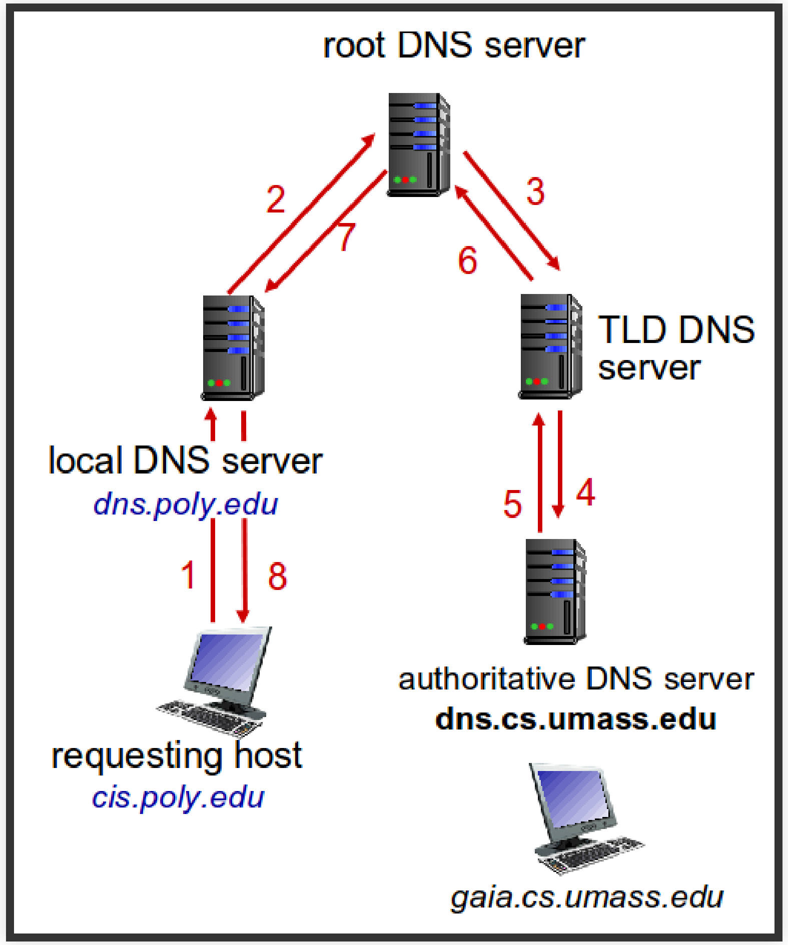
ITERATED QUERY:

- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”



RECURSIVE QUERY

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



DNS NAME RESOLUTION EXAMPLE

Demo:

nslookup

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 db storing **resource records** (RR)

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

DNS RECORDS

- [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

DNS RECORDS

- [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 NAME RESOLUTION EXAMPLE

Demo:

dig

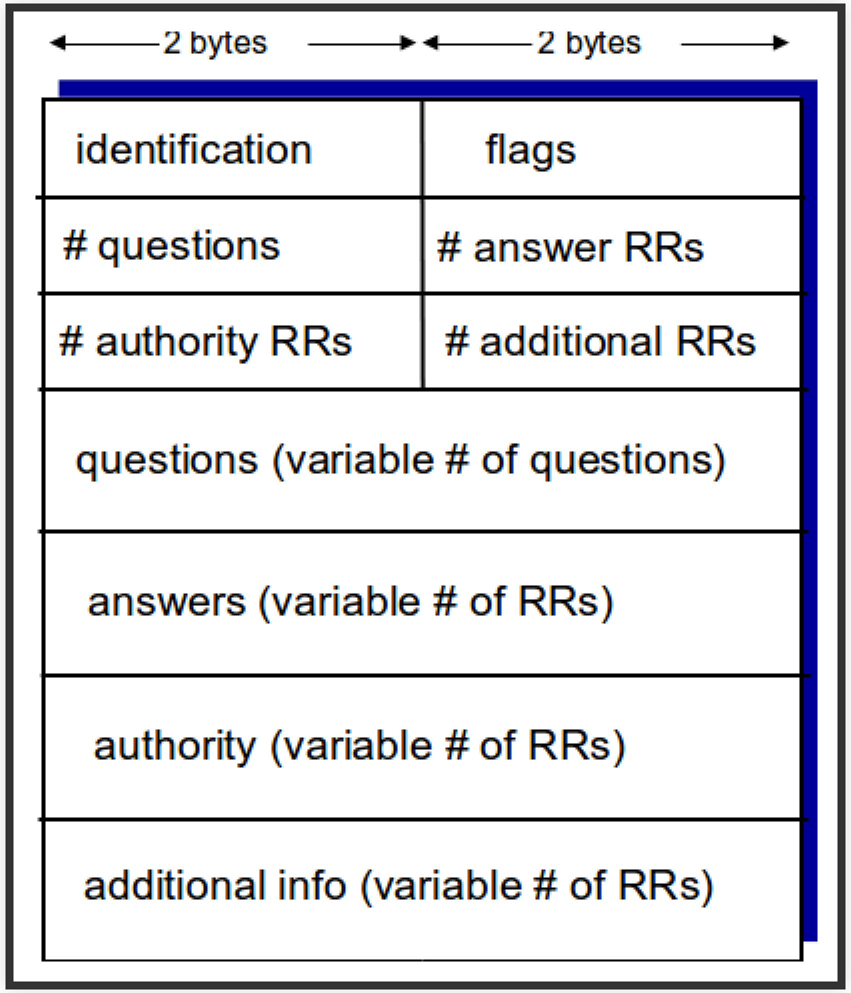
DNS PROTOCOL, MESSAGES

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



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 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

ATTACKING DNS

Redirect attacks

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

ATTACKING DNS

Exploit DNS for DDoS

- Send queries with spoofed source address: target IP
- Requires amplification

P2P APPLICATIONS

PURE P2P ARCHITECTURE

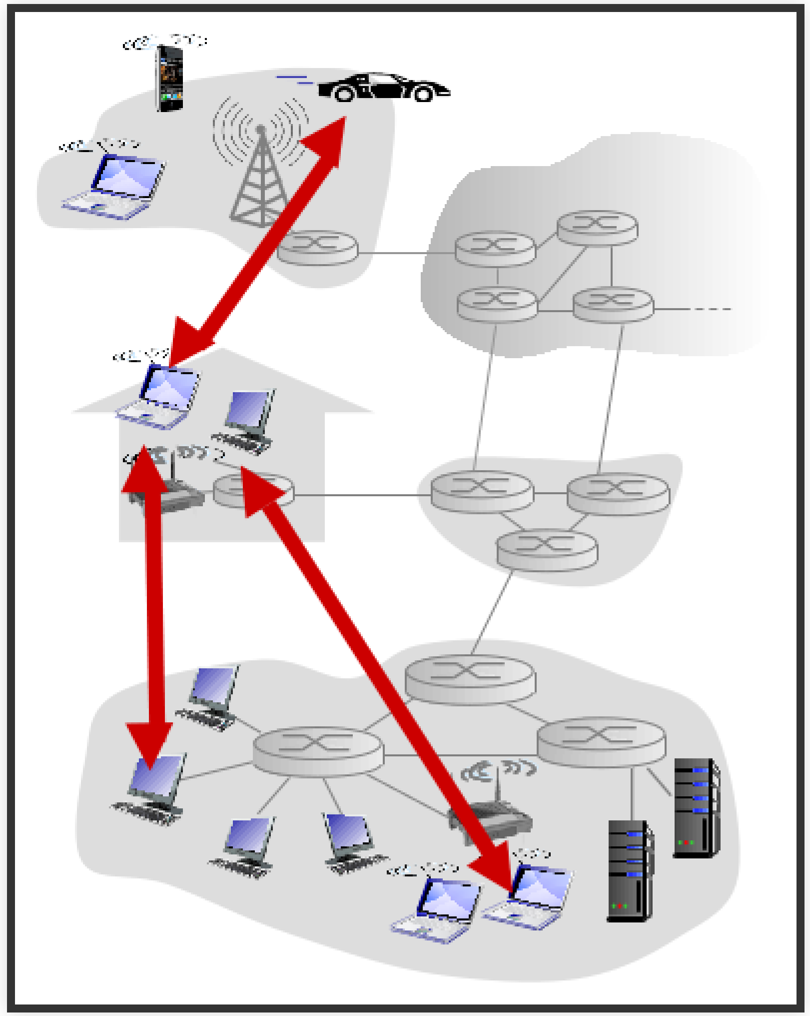
- no always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

PURE P2P ARCHITECTURE

examples:

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

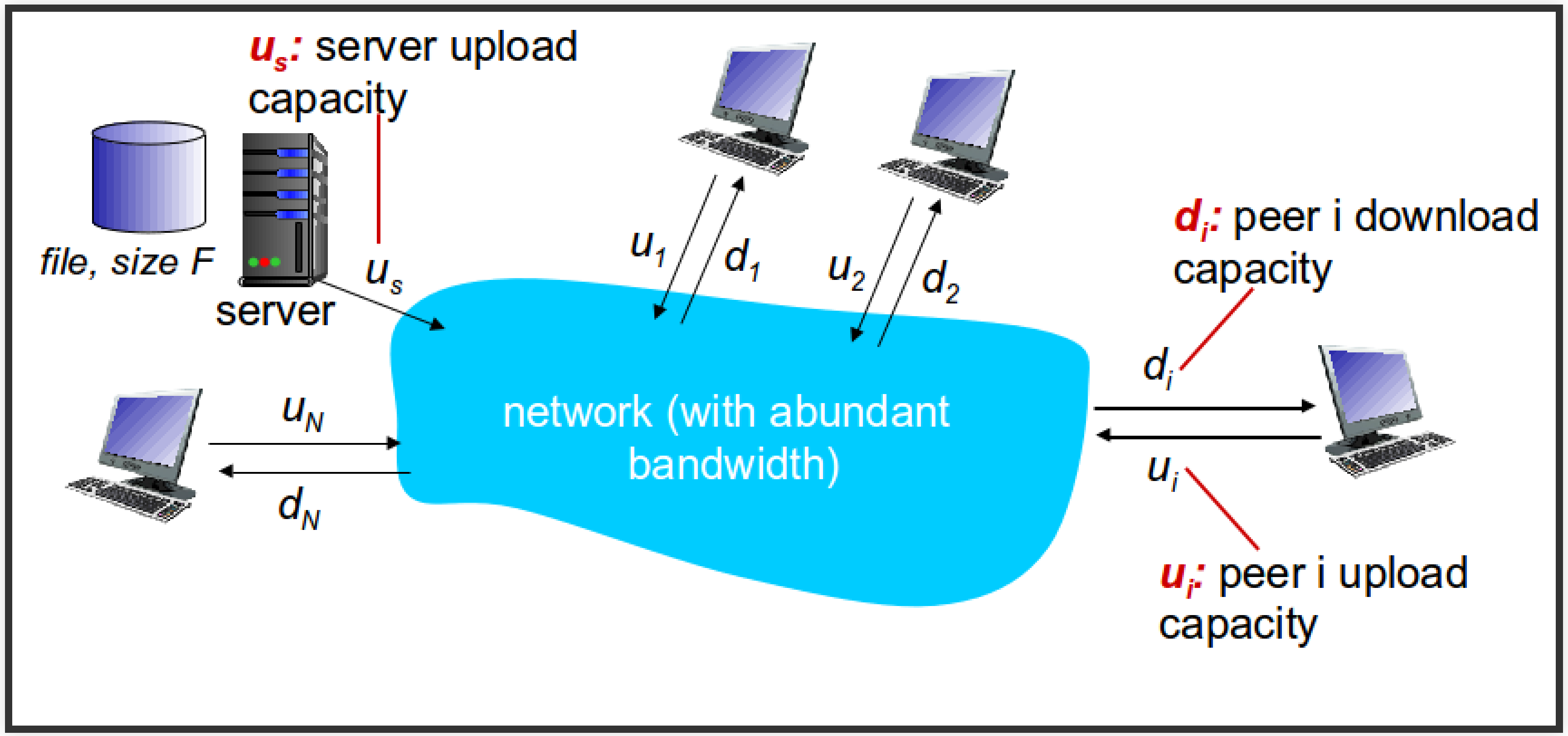
PURE P2P ARCHITECTURE



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



FILE DISTRIBUTION TIME: CLIENT-SERVER

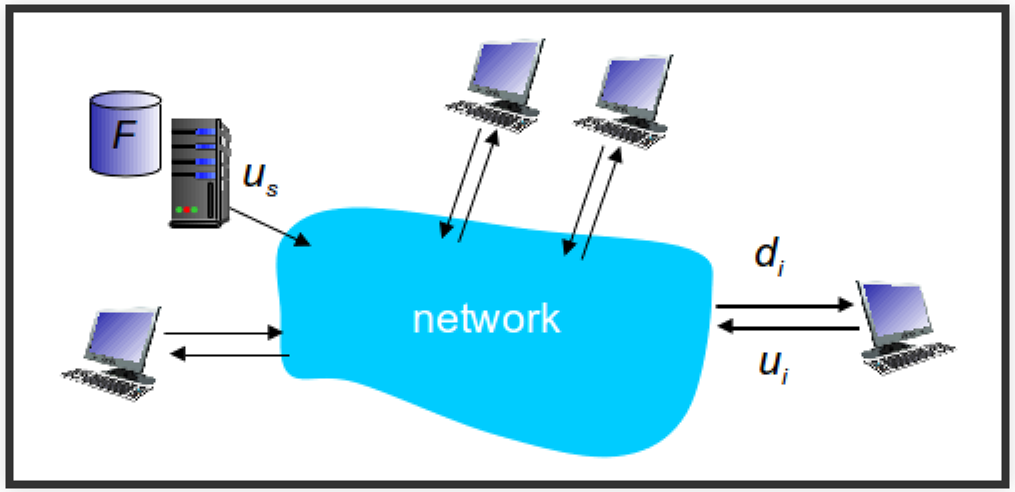
❗ **Server transmission:** must sequentially send (upload) N file copies:

- 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}

FILE DISTRIBUTION TIME: CLIENT-SERVER



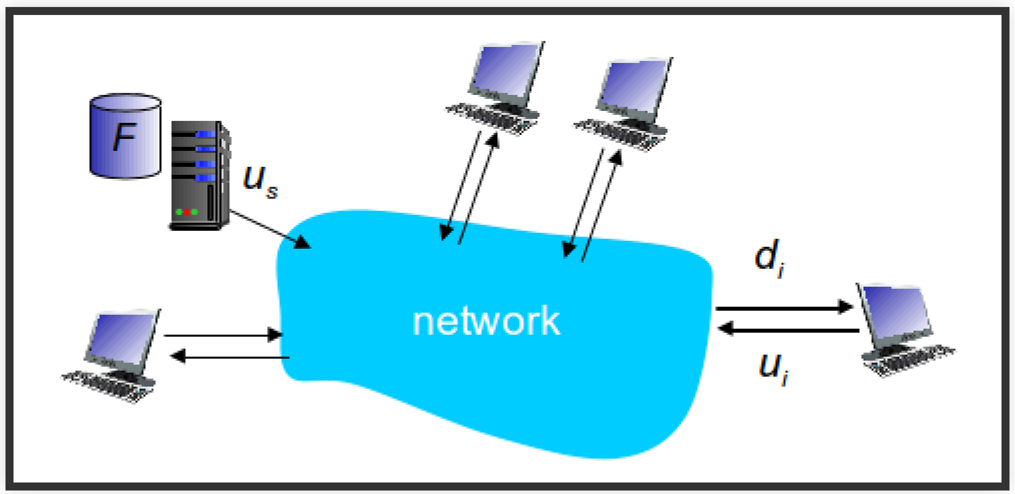
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} > \max(NF/u_s, F/d_{min})$$

Notice it 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

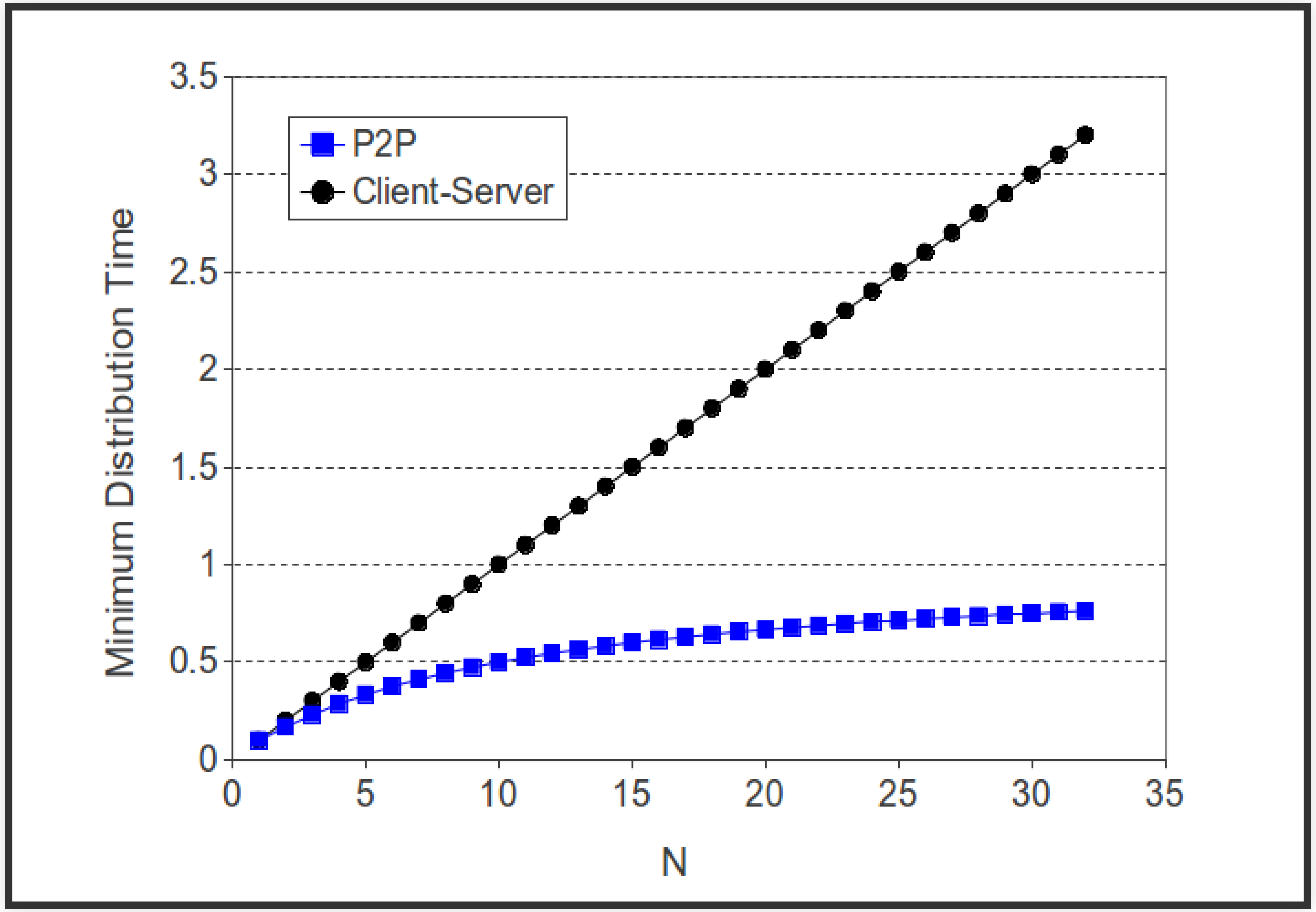
- max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

> time to distribute F to N clients using P2P approach

$$D_{P2P} > \max(F/u_s , F/d_{min} , NF/(u_s + \Sigma u_i))$$

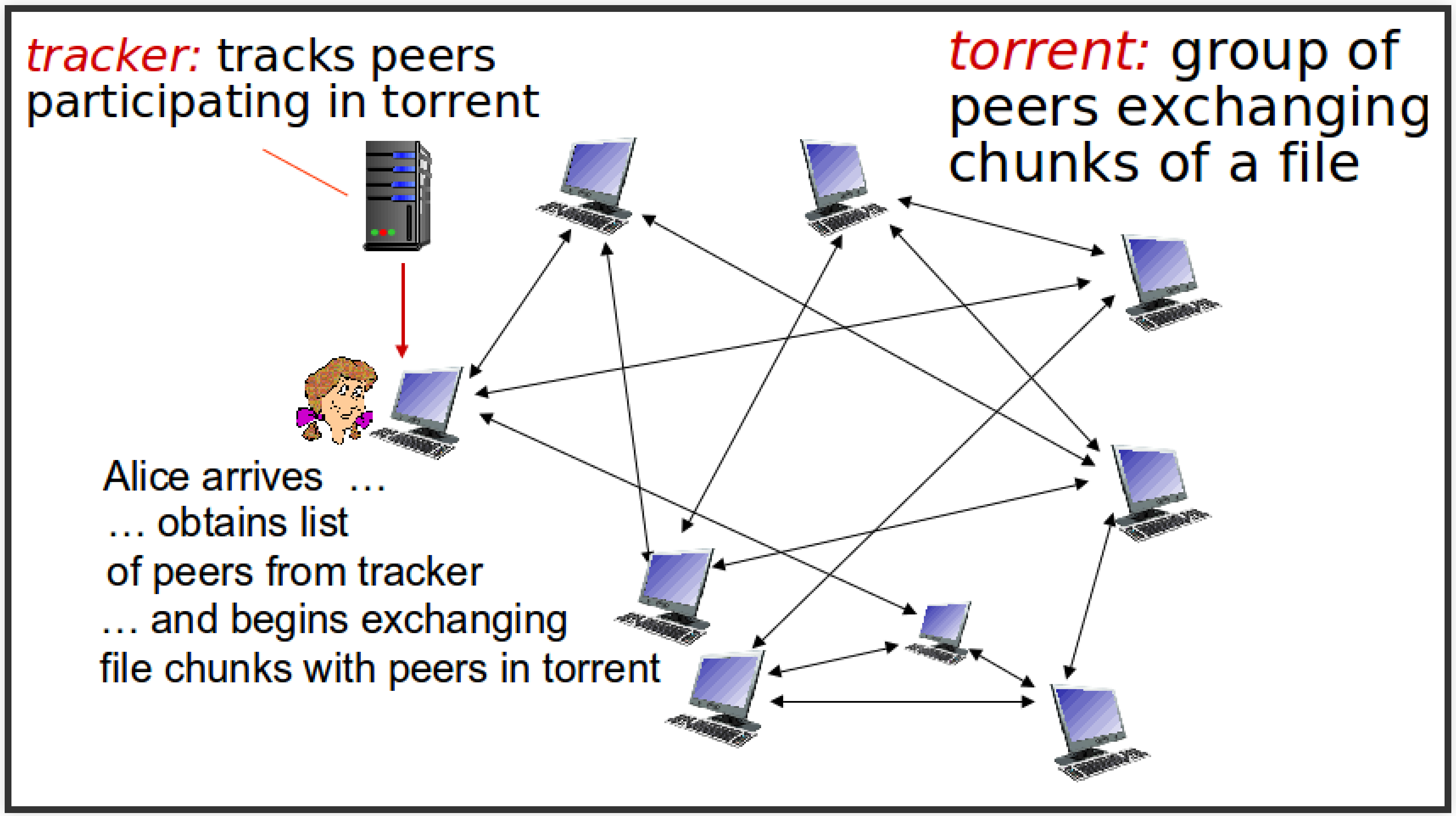
CLIENT-SERVER VS. P2P: EXAMPLE

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



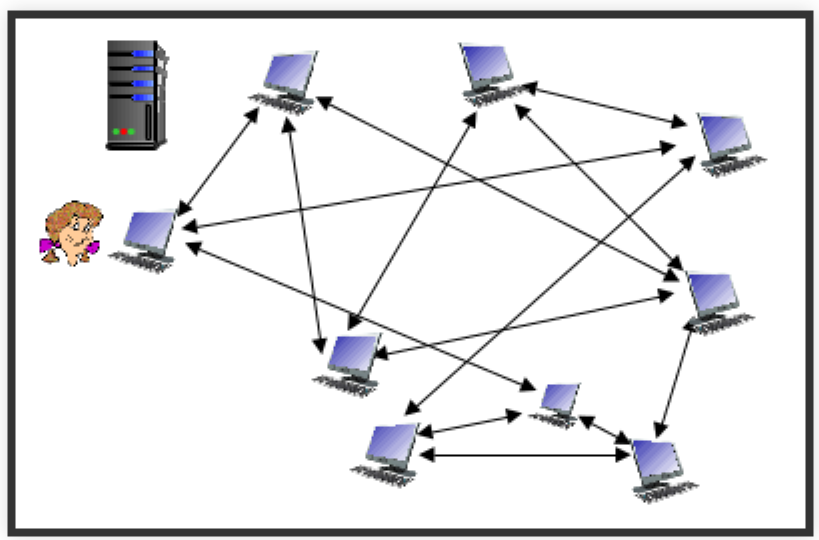
P2P FILE DISTRIBUTION: BITTORRENT

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



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”)



P2P FILE DISTRIBUTION: BITTORRENT

- 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

BITTORRENT: REQUESTING, SENDING FILE CHUNKS

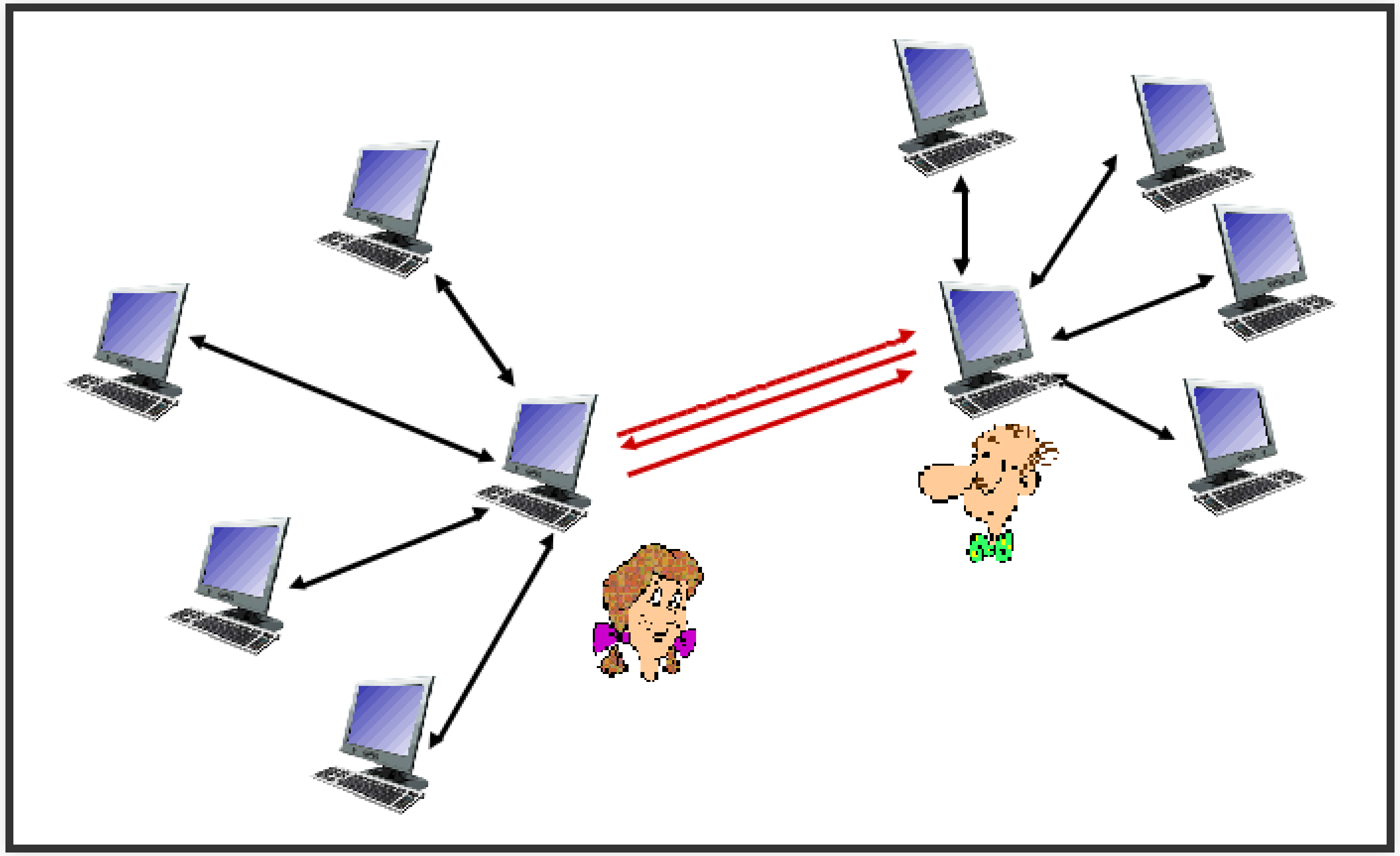
sending chunks: tit-for-tat

- 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 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

- Alice “optimistically unchokes” Bob
- Alice becomes one of Bob’s top-four providers; Bob reciprocates
- Bob becomes one of Alice’s top-four providers

BITTORRENT: TIT-FOR-TAT

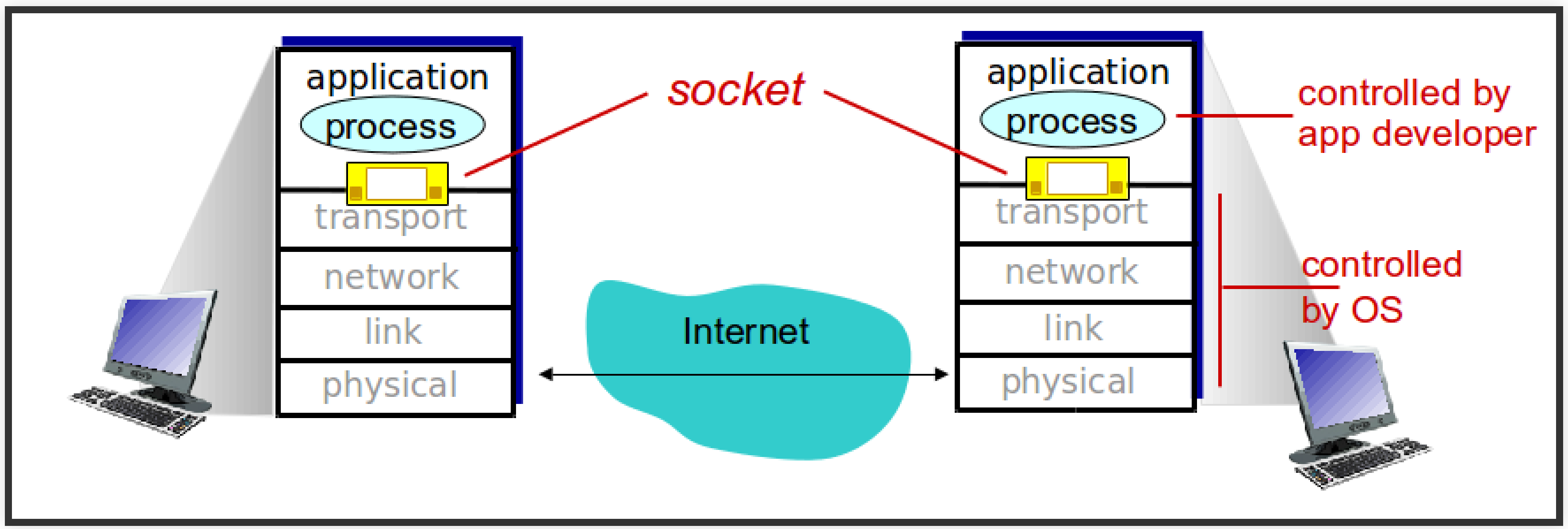


higher upload rate: find better trading partners, get file faster !

SOCKET PROGRAMMING WITH UDP AND TCP

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:

SOCKET PROGRAMMING WITH UDP

UDP: no “connection” between client and server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port number to each packet
- rcvr 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

SOCKETS IN JAVA

UDP Server

```
DatagramSocket socket = new DatagramSocket(12000);

System.out.println("Waiting for packets");

byte[] buf = new byte[1024];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);

String payload = new String(packet.getData(), 0, packet.getLength());
String responsePayload = payload.toUpperCase();

InetAddress address = packet.getAddress();
int port = packet.getPort();
buf = responsePayload.getBytes();
packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);
```

SOCKETS IN JAVA

UDP Client

```
InetAddress address = InetAddress.getLoopbackAddress();
Integer port = 12000;

DatagramSocket socket = new DatagramSocket();

System.out.println("Input lowercase sentence:\n");
Scanner scanner = new Scanner(System.in);
String message = scanner.nextLine();

byte[] buf = message.getBytes();
DatagramPacket packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);

packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);

String received = new String(packet.getData(), 0, packet.getLength());
System.out.println("Received:" + received);
```

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

SOCKET PROGRAMMING WITH TCP

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)

SOCKET PROGRAMMING WITH TCP

❗ Application viewpoint: TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

SOCKETS IN JAVA

TCPServer

```
ServerSocket serverSocket = new ServerSocket(12000);
Socket socket = serverSocket.accept();

boolean autoflush = true;
PrintWriter out = new PrintWriter(socket.getOutputStream(), autoflush);
BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream())
);
// read the response
boolean loop = true;
StringBuilder sb = new StringBuilder(8096);
while (loop) {
    if (in.ready()) {
        int i = 0;
        while (i != '\n') {
            i = in.read(); sb.append((char) i);
        }
        loop = false;
    }
}
String payload = sb.toString();
out.println(payload.toUpperCase() );
out.flush();
out.close();
```

```
socket.close();
serverSocket.close();
```

SOCKETS IN JAVA

TCPClient

```
Socket socket = new Socket("127.0.0.1", 12000);

boolean autoflush = true;
PrintWriter out = new PrintWriter(socket.getOutputStream(), autoflush);
BufferedReader in = new BufferedReader(
    new InputStreamReader(socket.getInputStream())
);

System.out.println("Input lowercase sentence:\n");
Scanner scanner = new Scanner(System.in);
String message = scanner.nextLine();

out.println(message);
out.println();
out.flush();

// read the response
String response = in.readLine();
System.out.println("Received:" + response);
out.close();
socket.close();
```

VIDEO STREAMING AND CONTENT DELIVERY NETWORKS

INTERNET VIDEO - CONTEXT

- **Video traffic:** major consumer of Internet bandwidth
  - Netflix: 37% of downstream residential ISP traffic
  - YouTube: 16% of downstream residential ISP traffic
  - ~1B YouTube users, ~75M Netflix users

INTERNET VIDEO - CONTEXT

- **Challenge:** scale - how to reach ~1B users?
    - Single mega-video server won't work (why?)
- **Challenge:** heterogeneity
- different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- **Solution:** distributed, application-level infrastructure

MULTIMEDIA: VIDEO

- **Video:** sequence of images displayed at constant rate
  - e.g., 24 images/sec
- **Digital image:** array of pixels
  - each pixel represented by bits
- **Coding:** use redundancy *within* and *between* images to decrease number of bits used to encode image
  - *spatial* (within image)
  - *temporal* (from one image to next)

MULTIMEDIA: VIDEO

spatial coding example: instead of sending *N* values of same color (all purple), send only two values: color value (*purple*) *and number of repeated values* (N)

frame *i* →

frame *i+1*

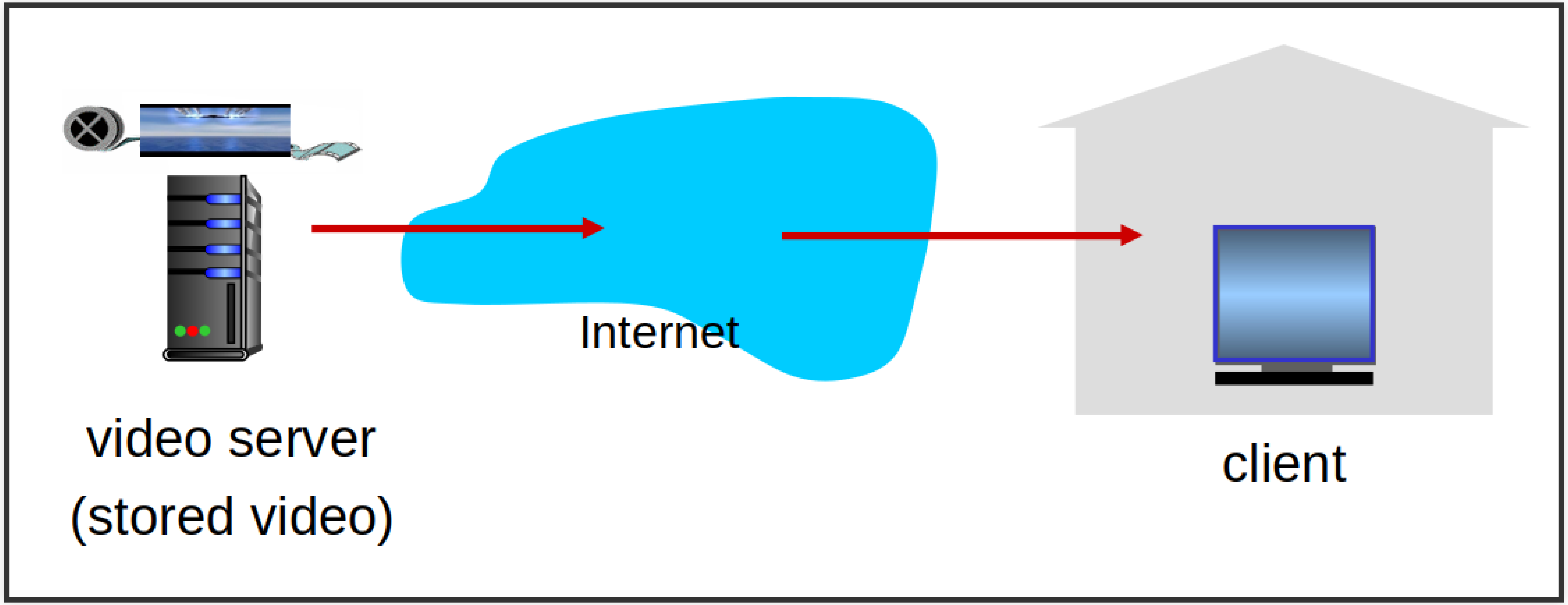
temporal coding example: instead of sending complete frame at i+1, send only differences from frame i

MULTIMEDIA: VIDEO

- **CBR (constant bit rate):** video encoding rate fixed
- **VBR (variable bit rate):** video encoding rate changes as amount of spatial, temporal coding changes
- **Examples:**
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (often used in Internet, < 1 Mbps)
  - 4K quality (> 10Mbps)

STREAMING STORED VIDEO

Simple scenario



MULTIMEDIA: VIDEO

Single 2Mbps video with 67 min duration ⇒ 1 GB storage and traffic

> Most important: Average throughput >= bit rate of compressed video

HTTP STREAMING AND DASH

DASH: **D**ynamic, **A**daptive **S**treaming over **H**TTP

DASH - SERVER

- Divides video file into multiple chunks
- Each chunk stored, encoded at different rates
- **Manifest file:** provides URLs for different chunks

DASH - CLIENT

- Periodically measures server-to-client bandwidth
- Consulting manifest, requests one chunk at a time
  - Chooses maximum coding rate sustainable given current bandwidth
  - Can choose different coding rates at different points in time (depending on available bandwidth at time)

DASH

- **"Intelligence" at client:** client determines
  - **When** to request chunk (so that buffer starvation, or overflow does not occur)
  - **What encoding rate** to request (higher quality when more bandwidth available)
  - **Where** to request chunk (can request from URL server that is "close" to client or has high available bandwidth)

CONTENT DISTRIBUTION NETWORKS

- **Challenge:** how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?

CONTENT DISTRIBUTION NETWORKS

Option 1: single, large “mega-server”

- single point of failure
- point of network congestion
- long path to distant clients
- multiple copies of video sent over outgoing link

1. quite simply: this solution **doesn’t scale**

CONTENT DISTRIBUTION NETWORKS

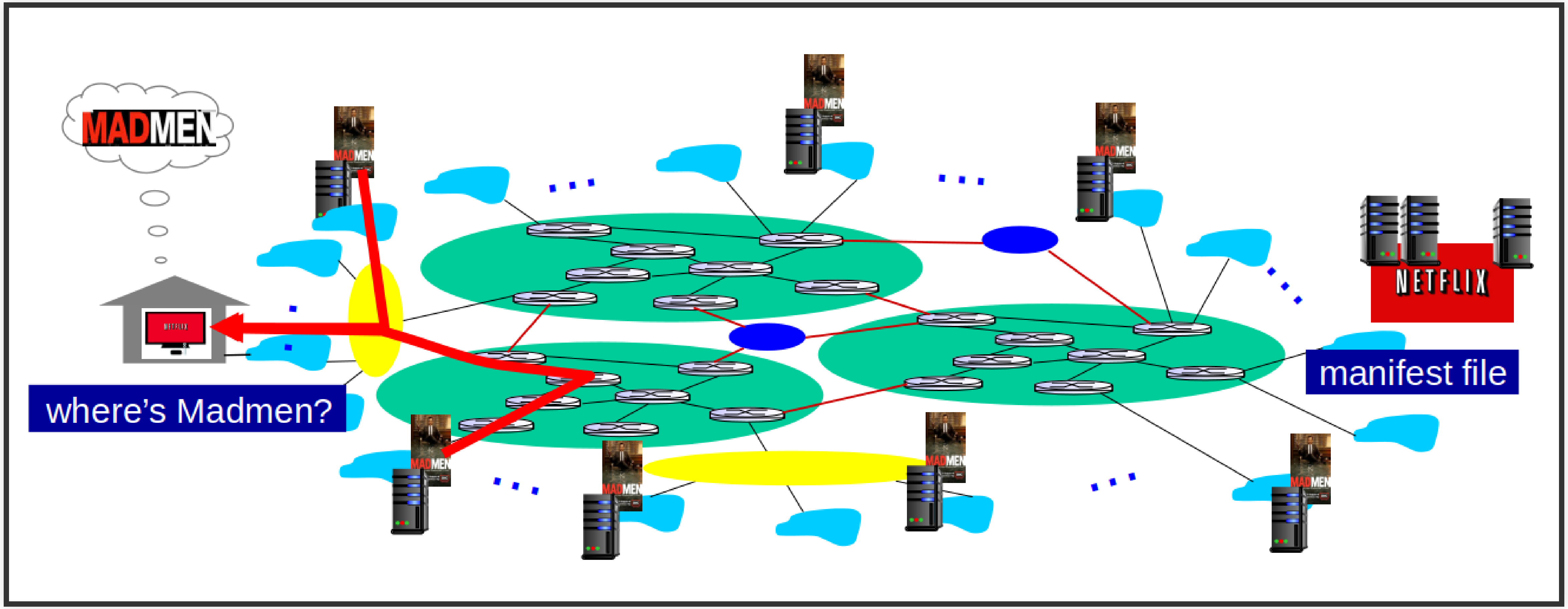
Option 2: store/serve multiple copies of videos at multiple geographically distributed sites (CDN)

- **Enter deep:** push CDN servers deep into many access networks
  - close to users
  - used by Akamai, 1700 locations
- **Bring home:** smaller number (10's) of larger clusters in POPs near (but not within) access networks
  - used by Limelight

CDN OPERATION

- **CDN:** stores copies of content at CDN nodes
  - e.g. Netflix stores copies of MadMen
- subscriber requests content from CDN
  - directed to nearby copy, retrieves content
  - may choose different copy if network path congested

CDN OPERATION



GOOGLE NETWORK INFRASTRUCTURE

- 14 Mega data centers (2016)
  - Each ~100.000 servers
- Estimated 50 clusters in IXP
  - Each 100-500 servers
- Many hundreds of "Enter-deep" clusters located in access ISP
  - Typically 10 servers in rack

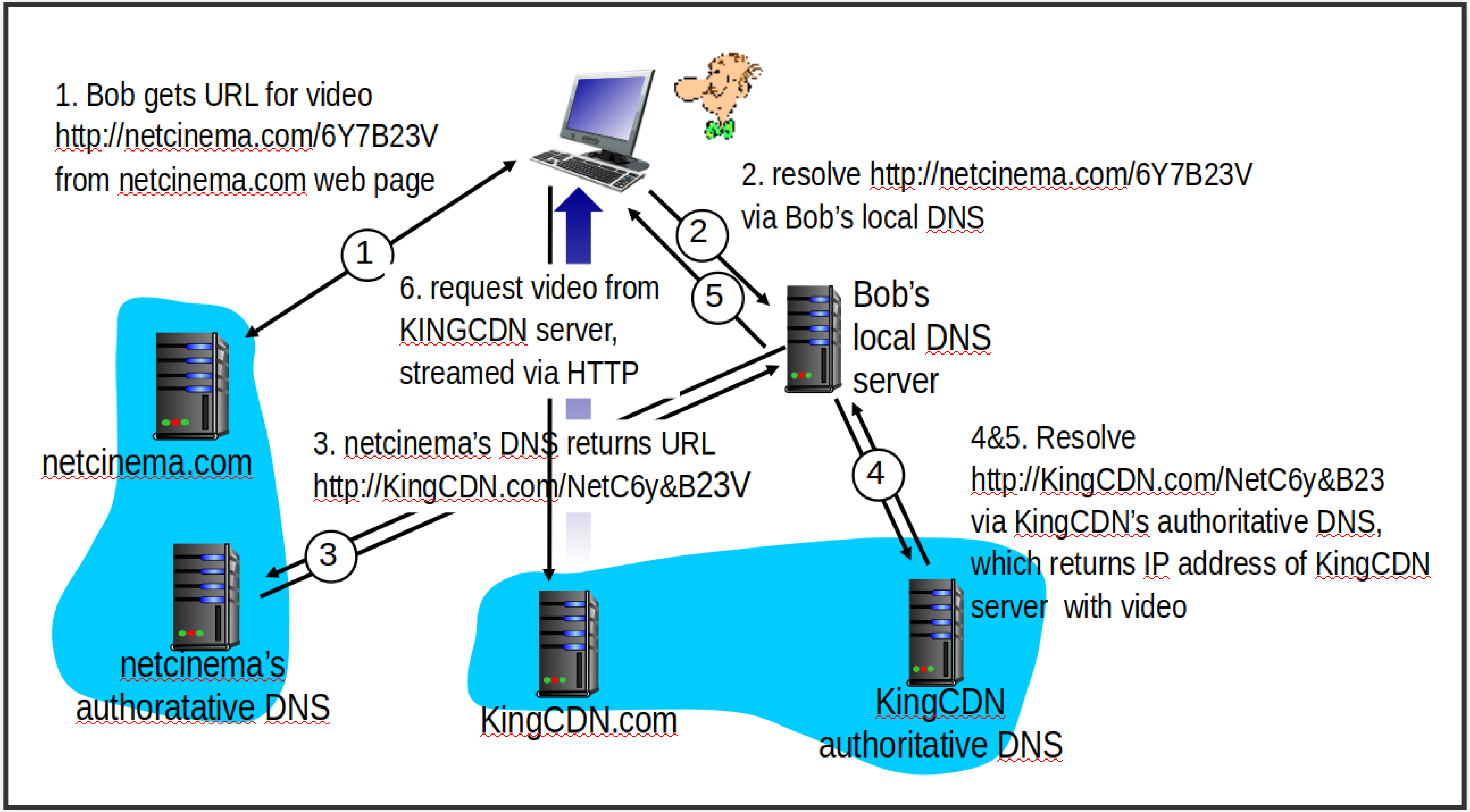
All networked with Googles private network - largely independent of public internet

CDN CONTENT ACCESS

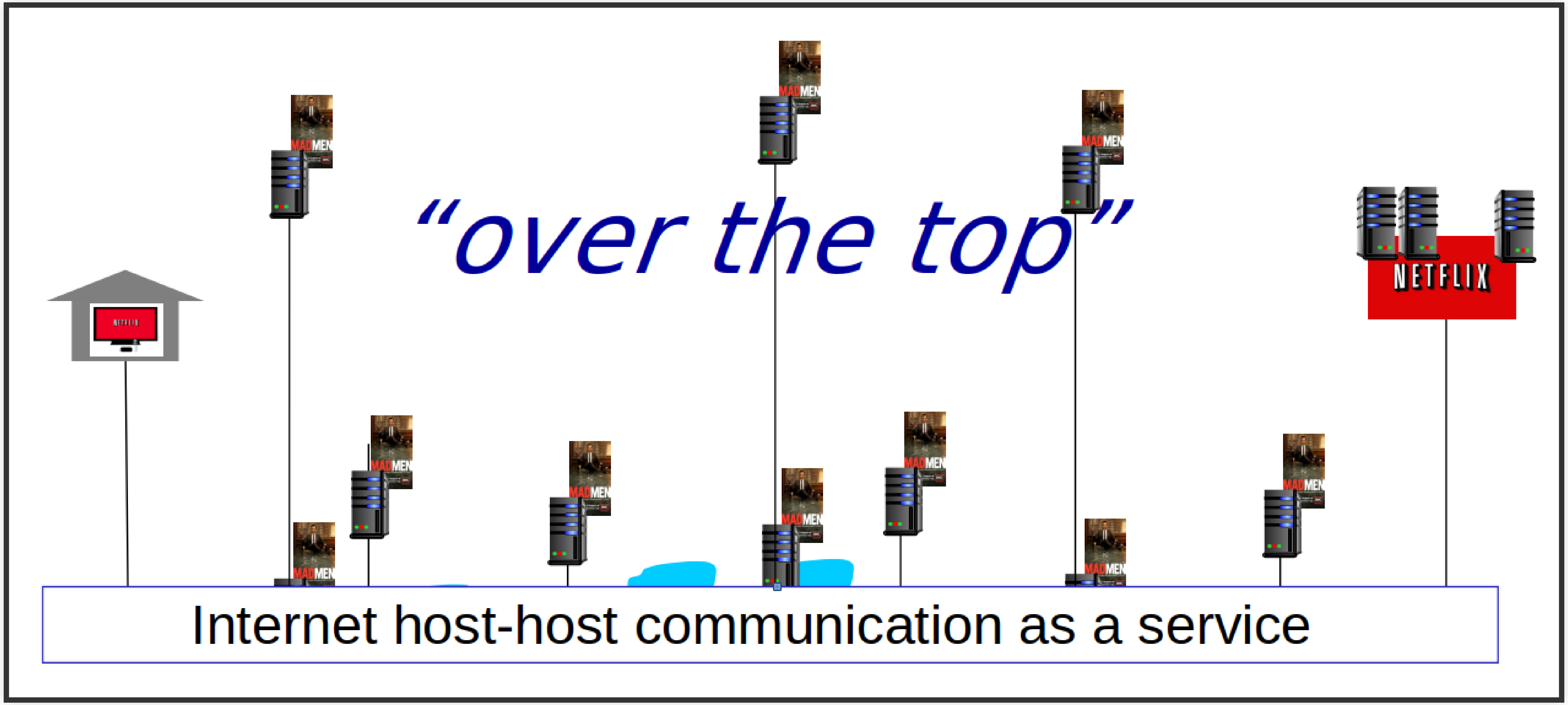
Bob (client) requests video *http://netcinema.com/6Y7B23V*

video stored in CDN at *http://KingCDN.com/NetC6y&B23V*

CDN CONTENT ACCESS



CDN - OVER THE TOP



CDN - OVER THE TOP

- **OTT challenges:** coping with a congested Internet
  - from which CDN node to retrieve content?
  - viewer behavior in presence of congestion?
  - what content to place in which CDN node?

CLUSTER SELECTION STRATEGIES

Geographically closest: Using geo-location database → Map Local DNS to location

- Ok for many users
- But if local DNS is not really local ⇒ poor performance
- Does not take into account network hops or network latency/conguestion

Real-time measurements: Performed periodically by the CDN

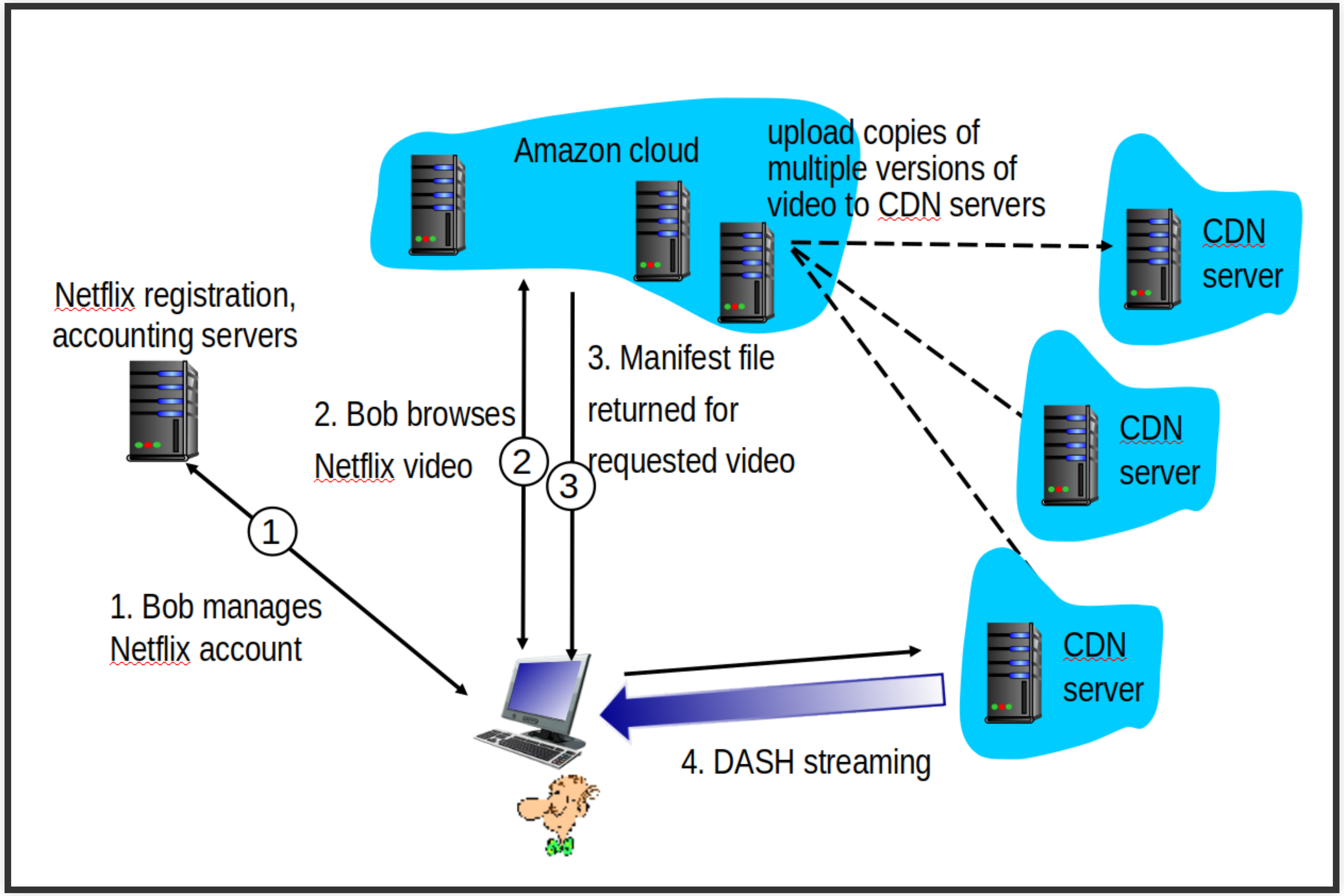
- Takes into account current traffic conditions
- Drawback: DNS might not reply to such probes

CASE STUDIES - NETFLIX, YOUBE AND KANKAN

CDN EXAMPLE - NETFLIX

- Runs website and more on Amazon Cloud
  - **Content ingestion:** Receive master movie and upload
  - **Content processing:** For DASH
  - **Uploading versions to CDN:** Has their own CDN (Akamai for website)

CDN EXAMPLE - NETFLIX



CDN EXAMPLE - NETFLIX

- Server racks at
  - 50 IXP locations
  - Hundreds of ISPs
- Pushes to racks during off-peak periods
- Netflix software tells which CDN server to use

CDN EXAMPLE - YOUTUBE

- 300 hours of video uploaded every minute
- Several billion video views a day
- Uses pull-caching
- Directs user to server where RTT is lowest
- Requieres user to select version/quality (Not DASH)
- Processes every video uploaded (making different versions)

CDN EXAMPLE - KANKAN

- Netflix and Google setup costly (servers, bandwith)
- Kankan uses P2P delivery (along with client-server)
- Few 100's servers within China - pushes video to these
- Start videos from client-server, gradually use P2P when downloaded

QUESTIONS