Application Layer

Computer Networks, Winter 2020/2021

Lecturer: Prof. Xiaoming Fu
Assistants: Yachao Shao (MSc),
Fabian Wölk (MSc)

Chapter 2: The Application Layer

Our goals:

- Conceptual aspects of network application protocols
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - SMTP, IMAP
  - DNS

Application Layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 E-mail, SMTP, IMAP
- 2.4 The Domain Name System DNS
- 2.5 P2P applications

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
- Internet search
- …
- …

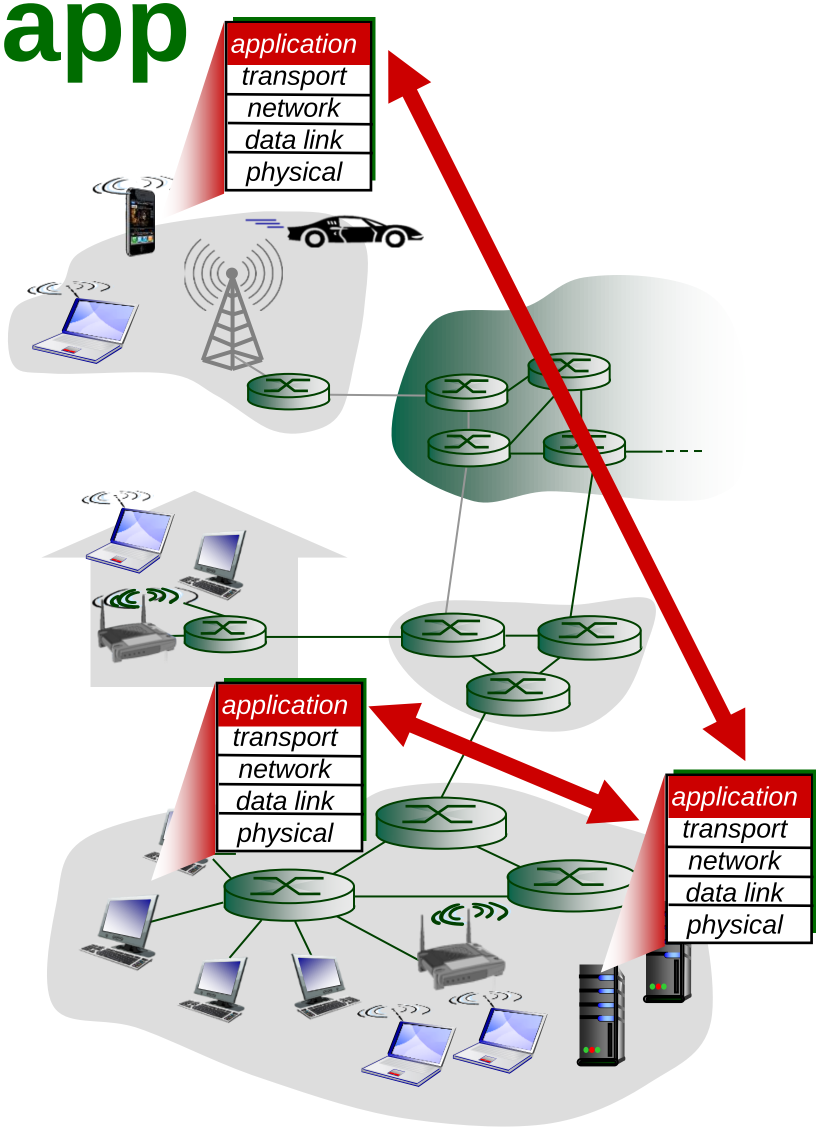
Creating a network app

write programs that:

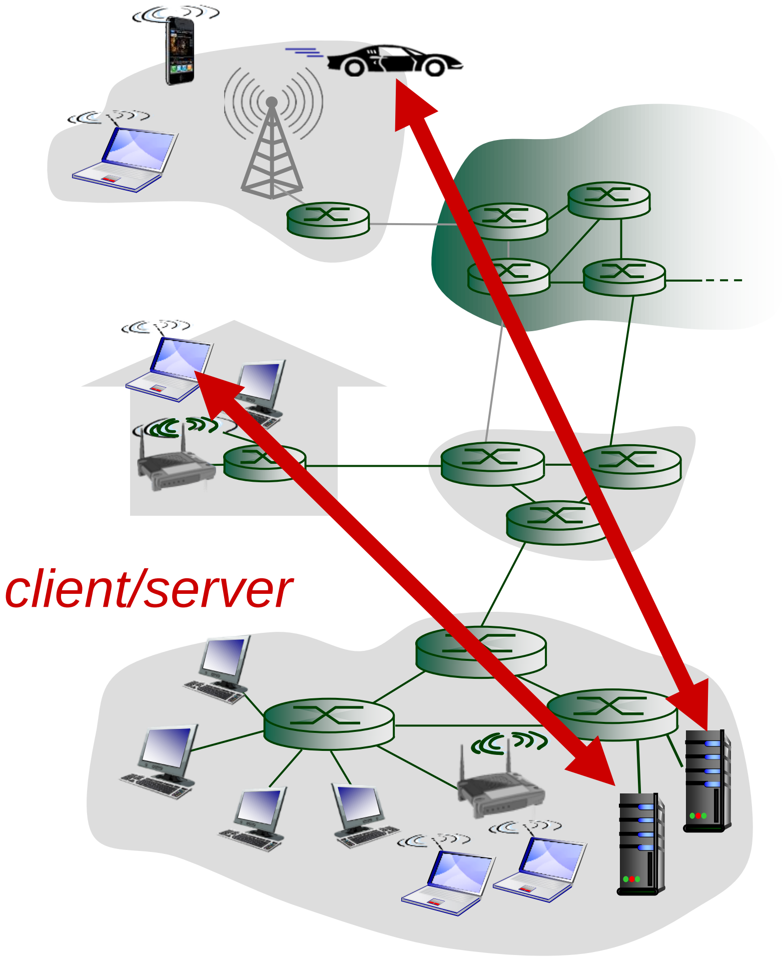
- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

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



Client-server architecture



server:
- always-on host
- permanent IP address
- often in data centers for scaling

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

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 via sockets (later in transport layer)

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*

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 & 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

Application Layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 E-mail, SMTP, IMAP
- 2.4 The Domain Name System DNS
- 2.5 P2P applications

Web and HTTP

First, a review…

- *web page* consists of *objects,* each of which can be stored on different Web servers
- 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.,

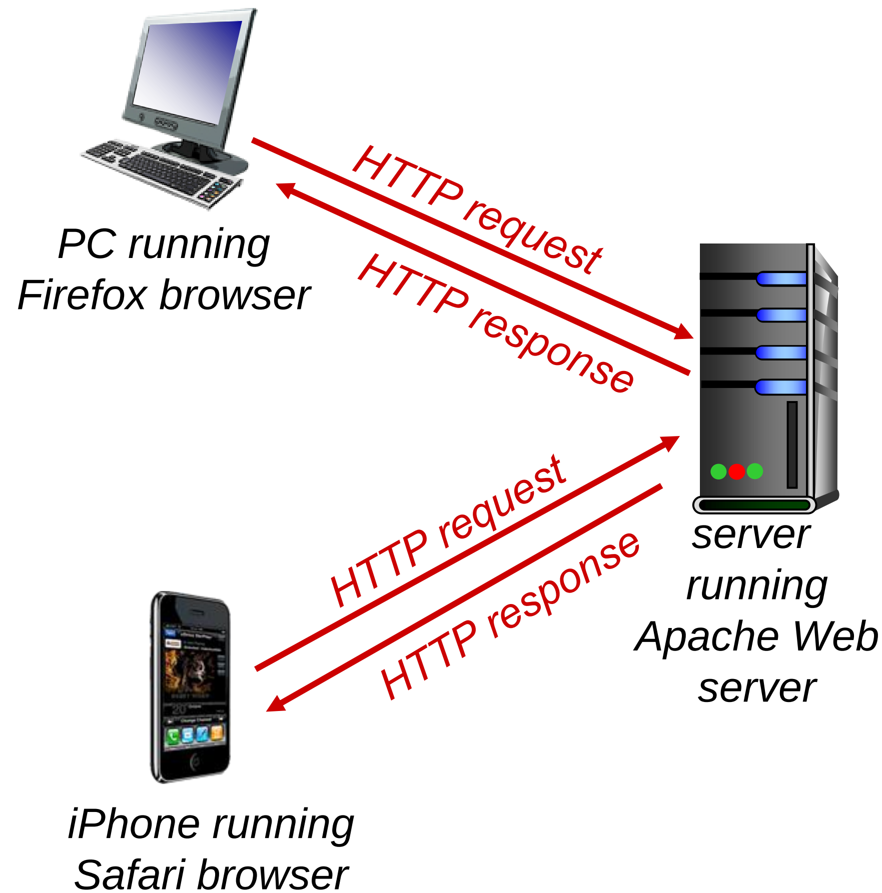
www.someschool.edu/someDept/pic.gif

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: two types

non-persistent HTTP

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

persistent HTTP

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

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

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

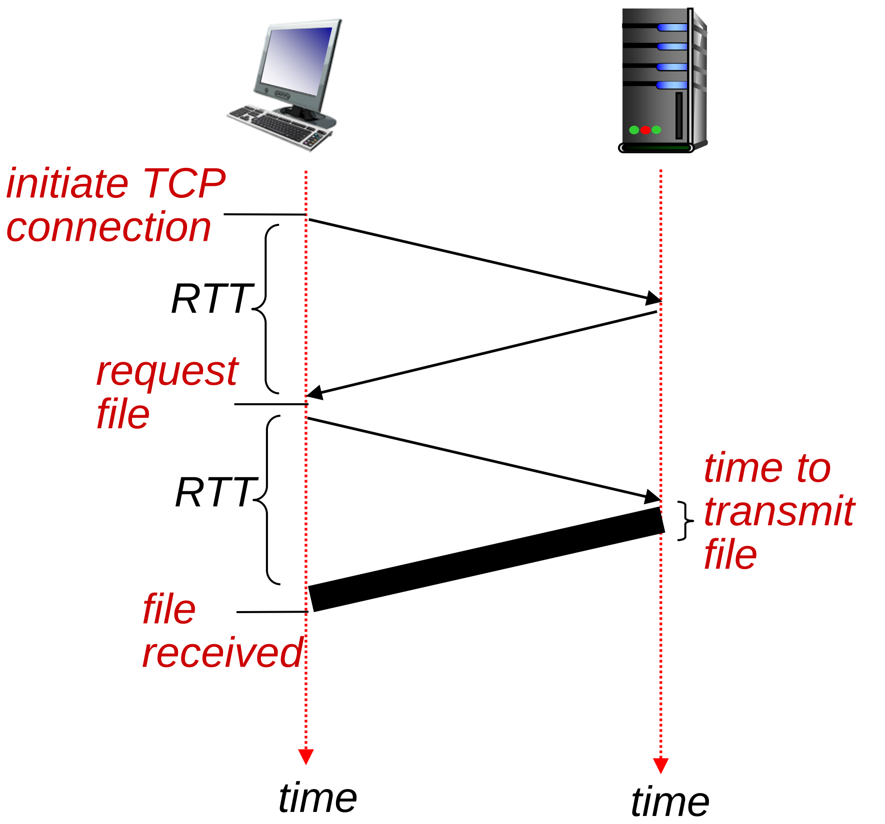
Non-persistent HTTP: response time

RTT (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 multiple parallel TCP connections to fetch referenced objects in parallel

persistent HTTP (HTTP1.1):

- 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 (cutting response time in half)

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)

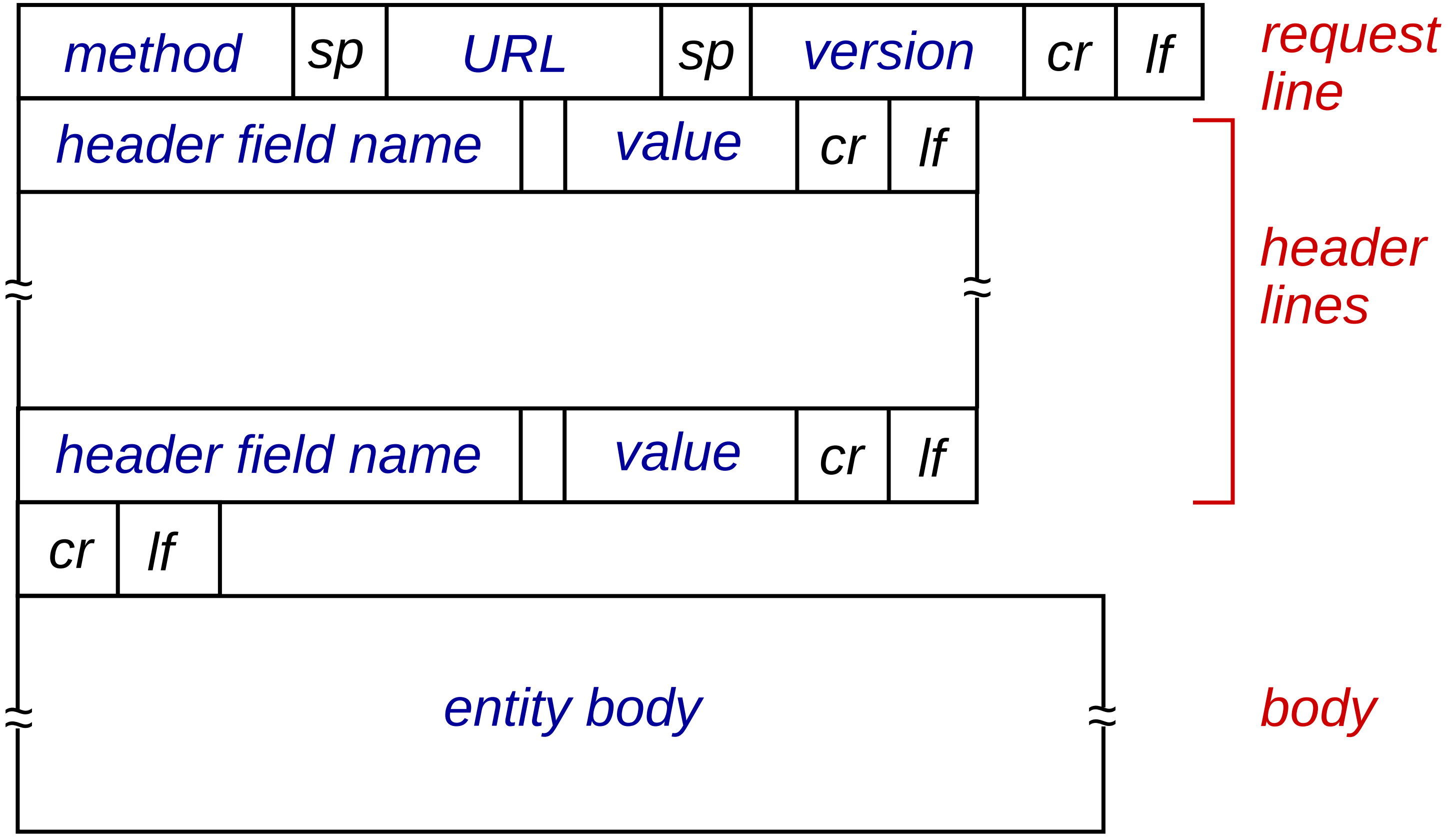
header lines

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: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

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

HTTP request message: general format



Request messages

POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET 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
  - requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads new file (object) to server
  - completely replaces file that exists at specified URL with content in entity body of POST HTTP request message
- 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

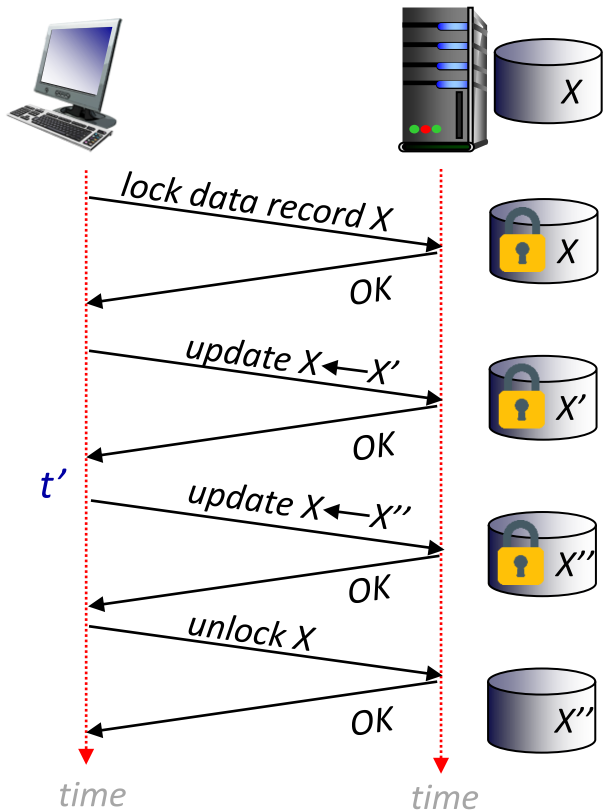
505 HTTP Version Not Supported

User-server state: cookies

Recall: HTTP GET/response interaction is *stateless*

- no notion of multi-step exchanges of HTTP messages to complete a Web "transaction"
  - no need for client/server to track "state" of multi-step exchange
  - all HTTP requests are independent of each other
  - no need for client/server to "recover" from a partially-completed-but-never-completely-completed transaction

a stateful protocol: client makes two changes to X, or none at all



Q: what happens if network connection or client crashes at t' ?

User-server state: cookies

many Web sites use cookies

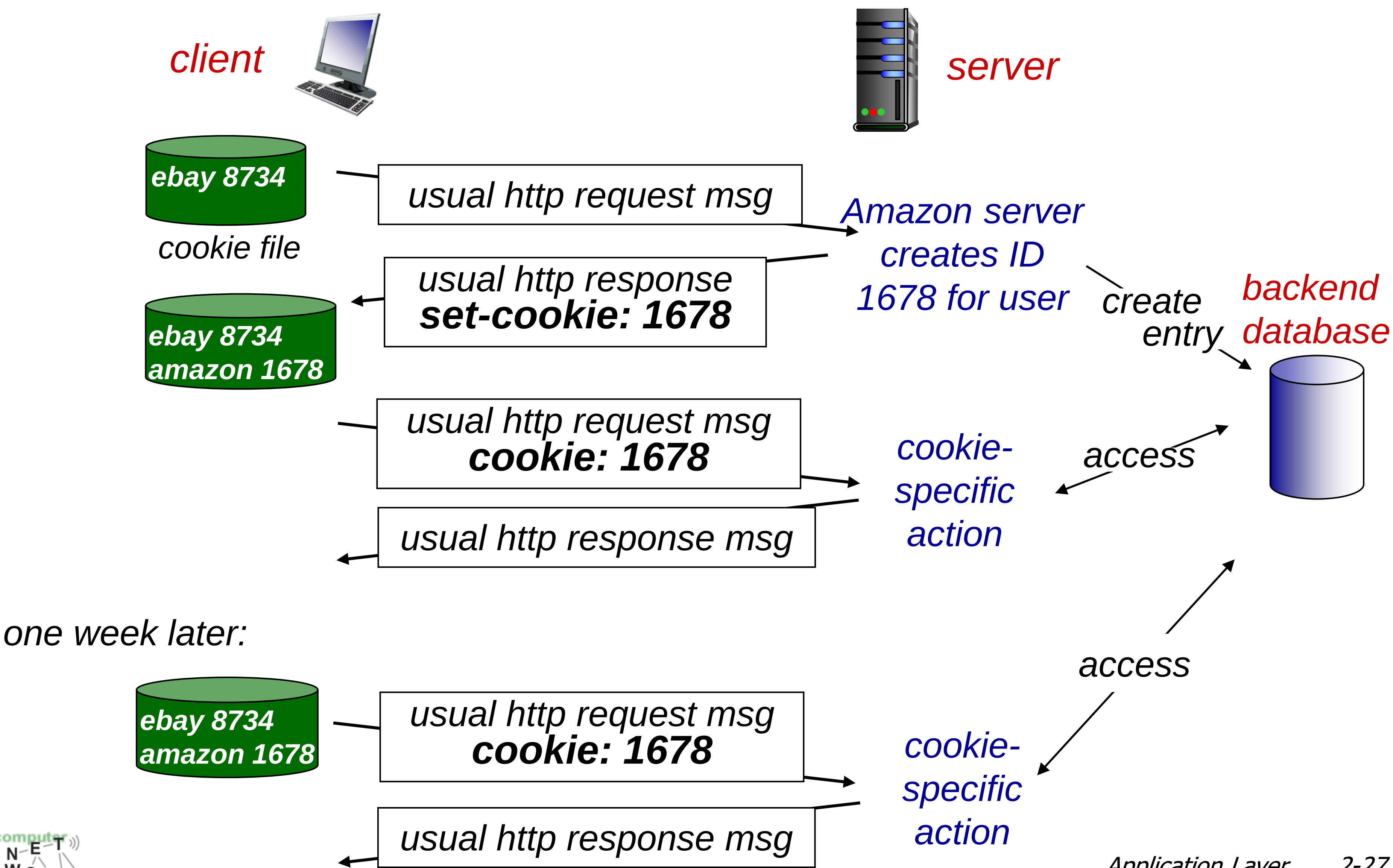
four components:

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 uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
- unique ID (aka "cookie")
- entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

Cookies: keeping "state" (cont.)



Cookies (continued)

what cookies can be used for:

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

how to keep "state":

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

aside

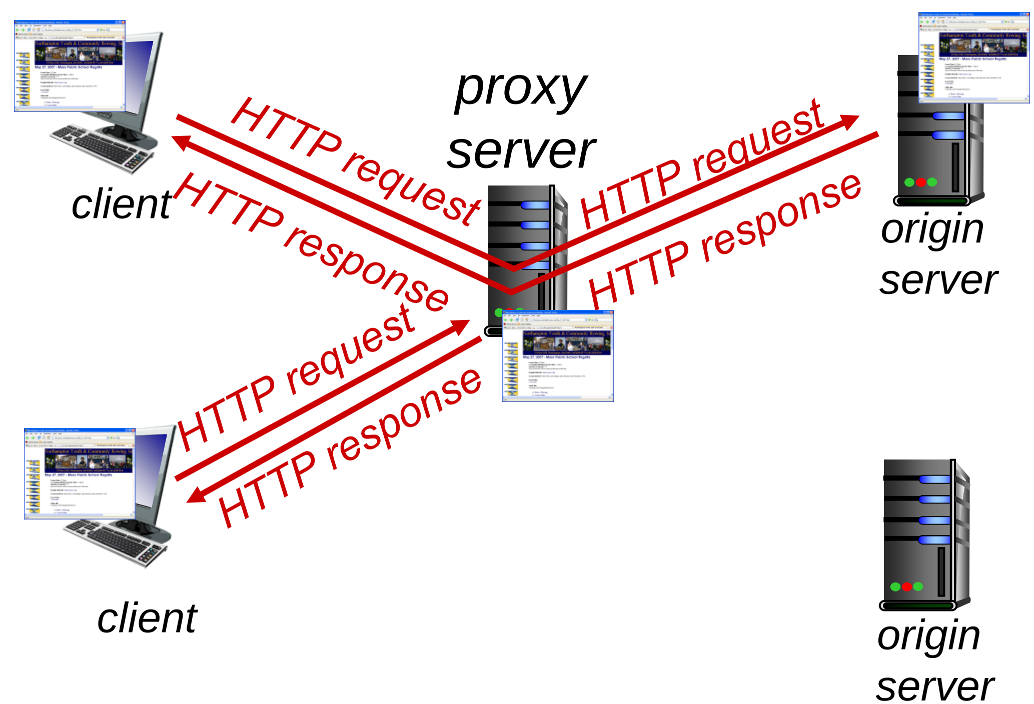
cookies and privacy:

- *cookies permit sites to learn a lot about you on their site.*
- *third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites*

Web caches (proxy server)

goal: satisfy client request without involving origin server

- user configures browser to point to a (local) *Web cache*
- browser sends all HTTP requests to cache
  - *if* object in cache: cache returns object to client
  - *else* cache requests object from origin server, caches received object, 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)
- *server tells cache about object's allowable caching in response header:*

```
Cache-Control: max-age=<seconds>
```

```
Cache-Control: no-cache
```

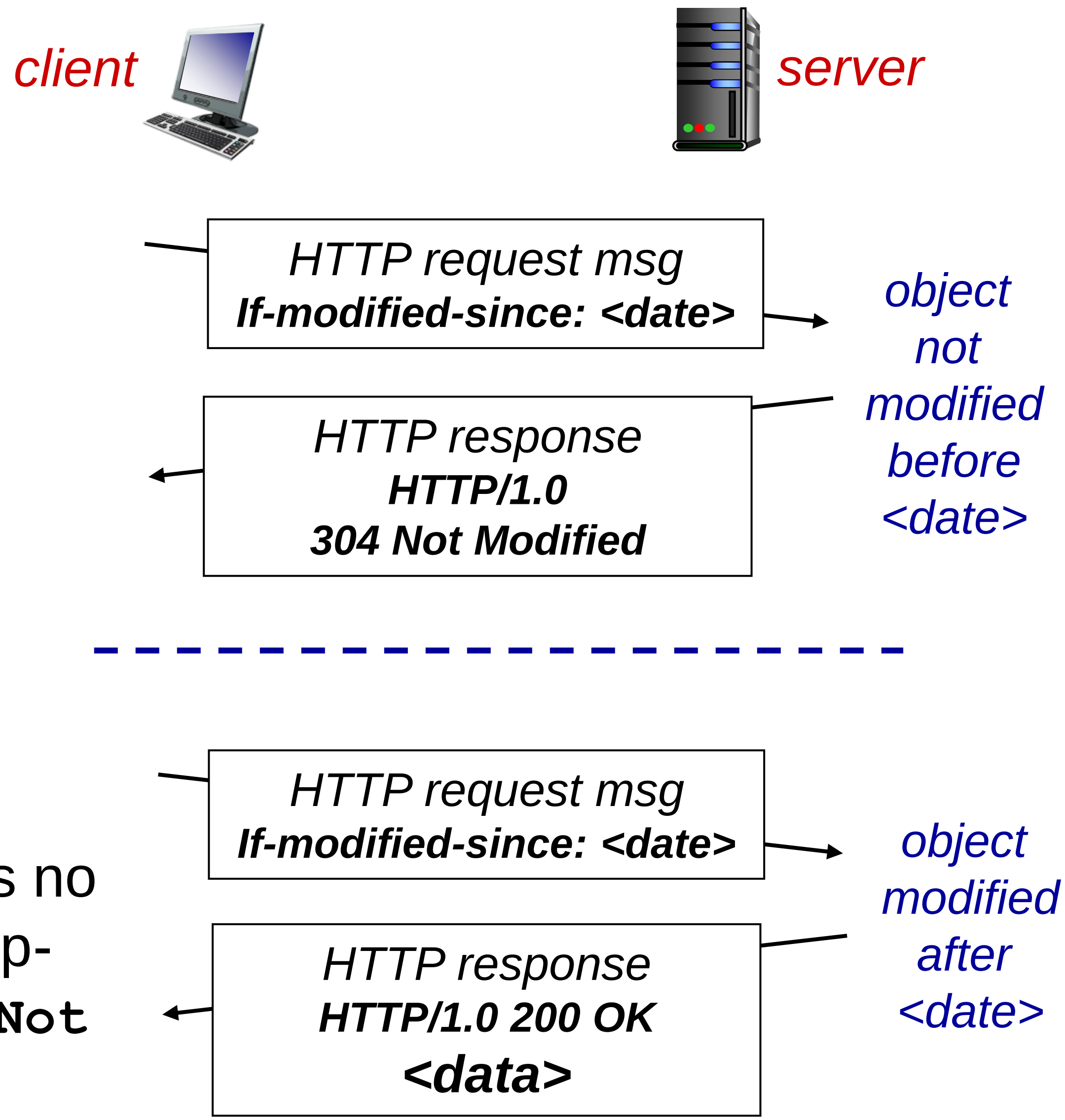
why Web caching?

- reduce response time for client request
  - cache is closer to client
- 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)
- *Faster and cheaper as buying faster access links!*

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

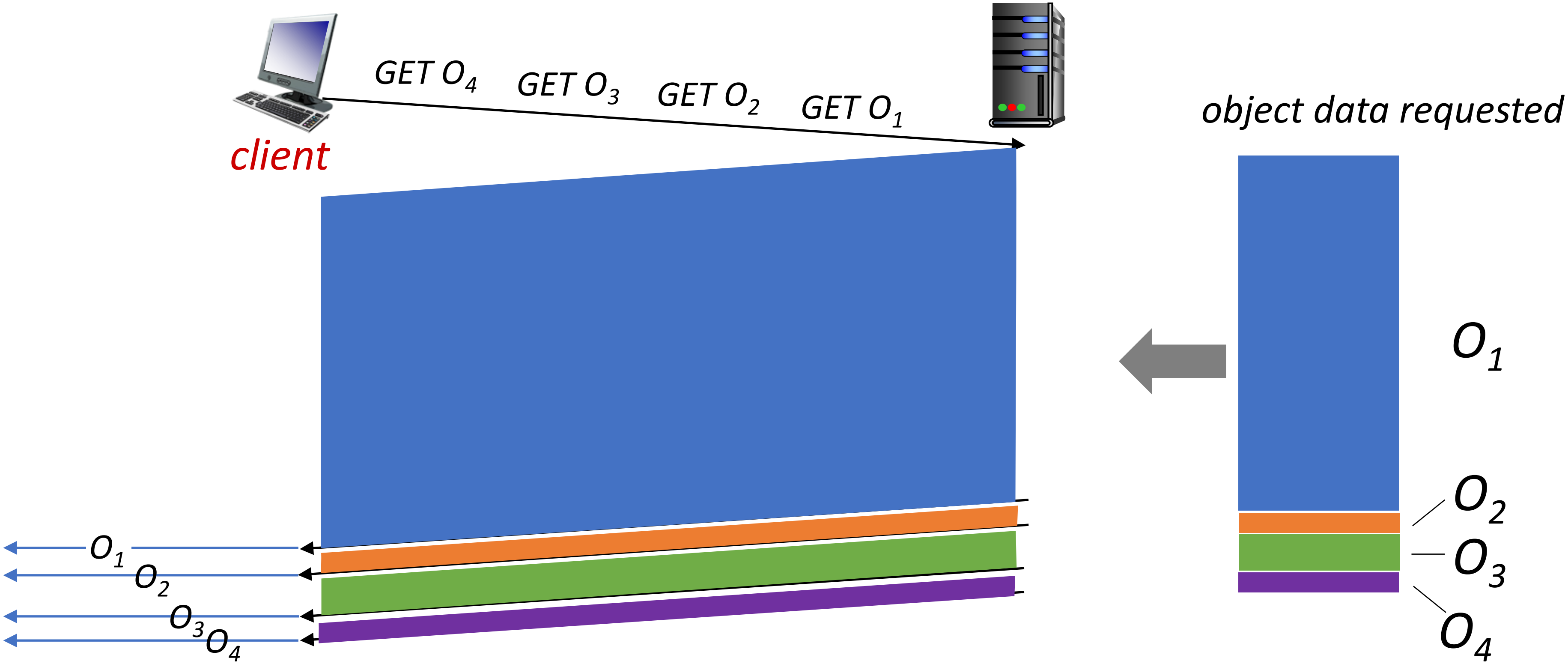
Key goal: decreased delay in multi-object HTTP requests

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

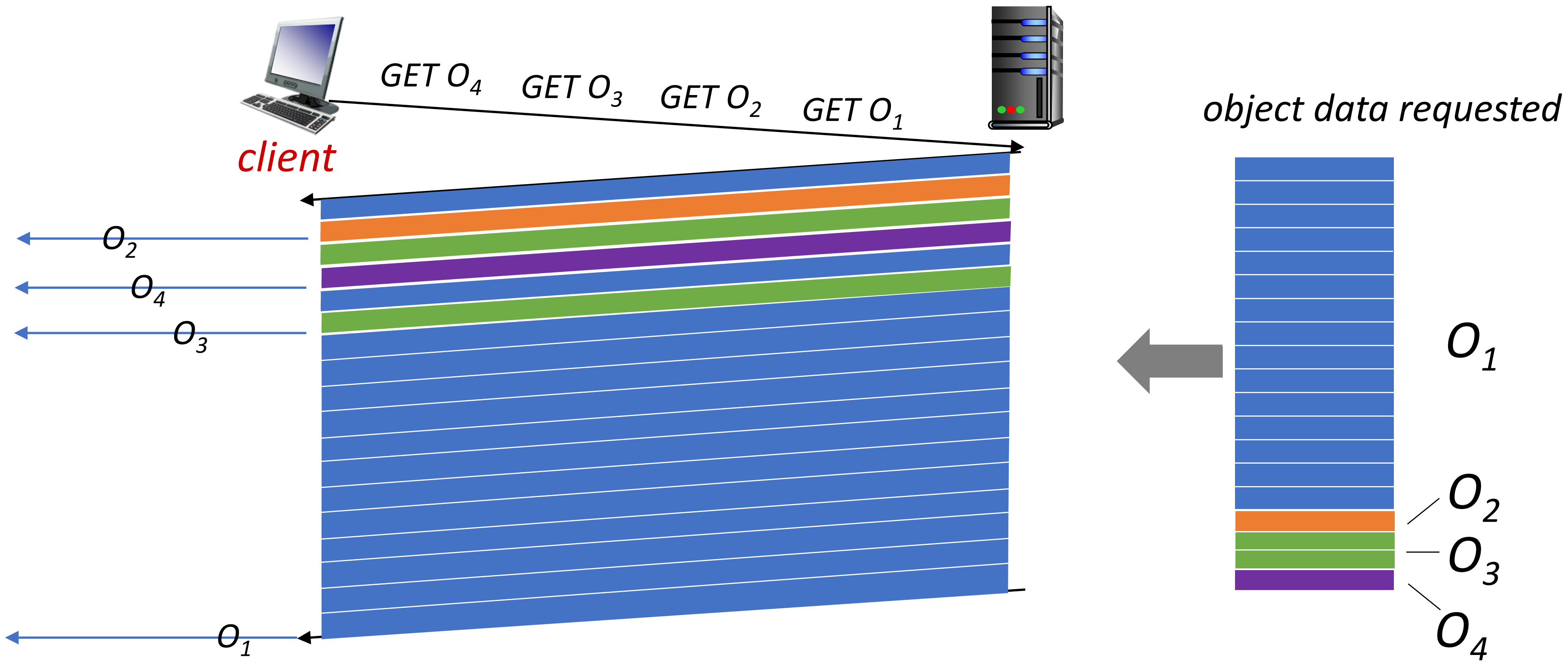
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



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

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*

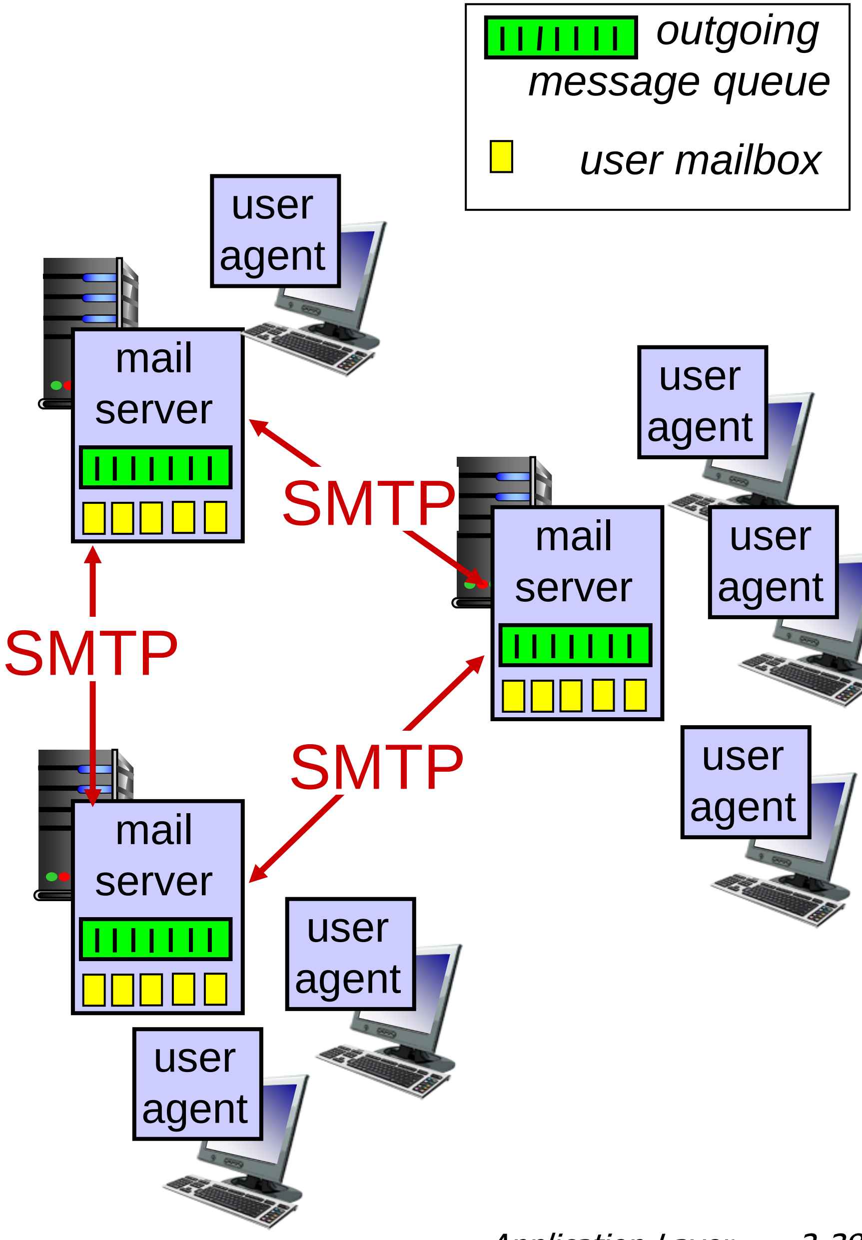
Application Layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 E-mail, SMTP, IMAP
- 2.4 The Domain Name System DNS
- 2.5 P2P applications

Electronic mail

Three major components:
- user agents
- mail servers
- simple mail transfer protocol: SMTP

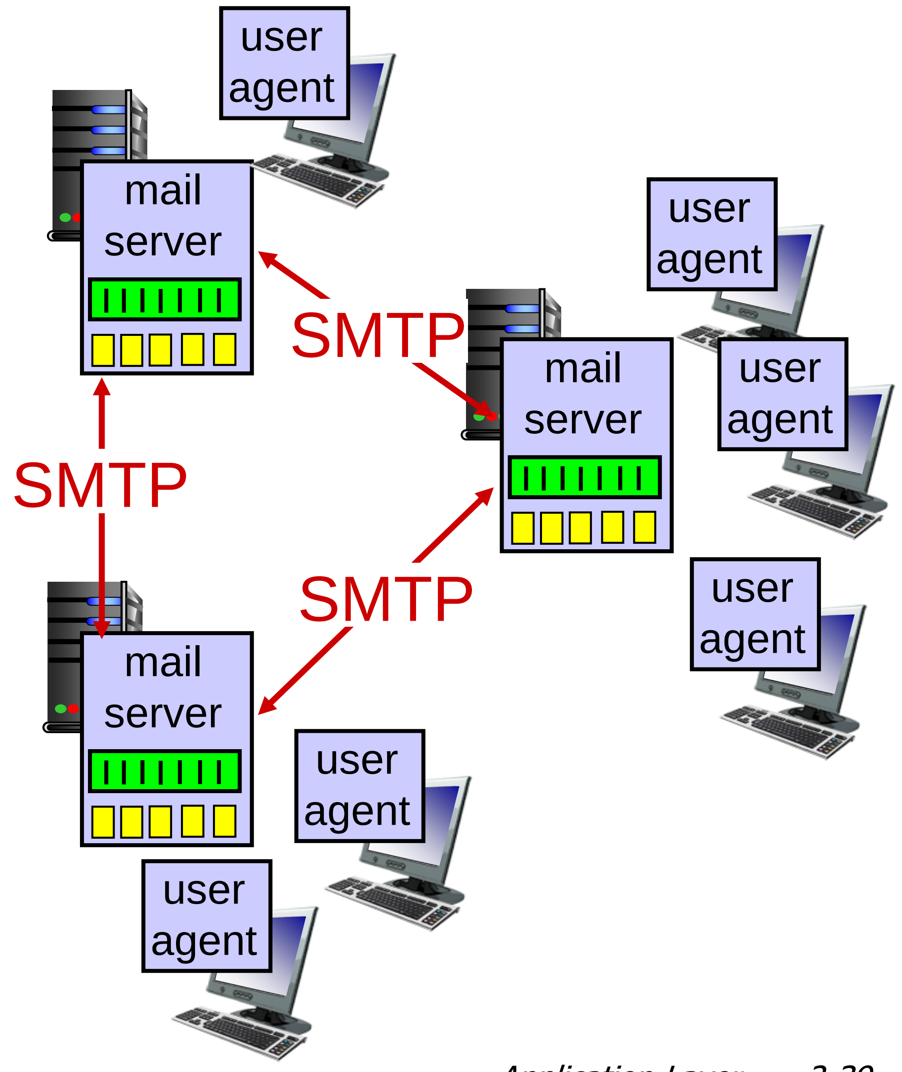
User Agent
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Outlook, iPhone mail client
- outgoing, incoming messages stored on server



Electronic mail: mail servers

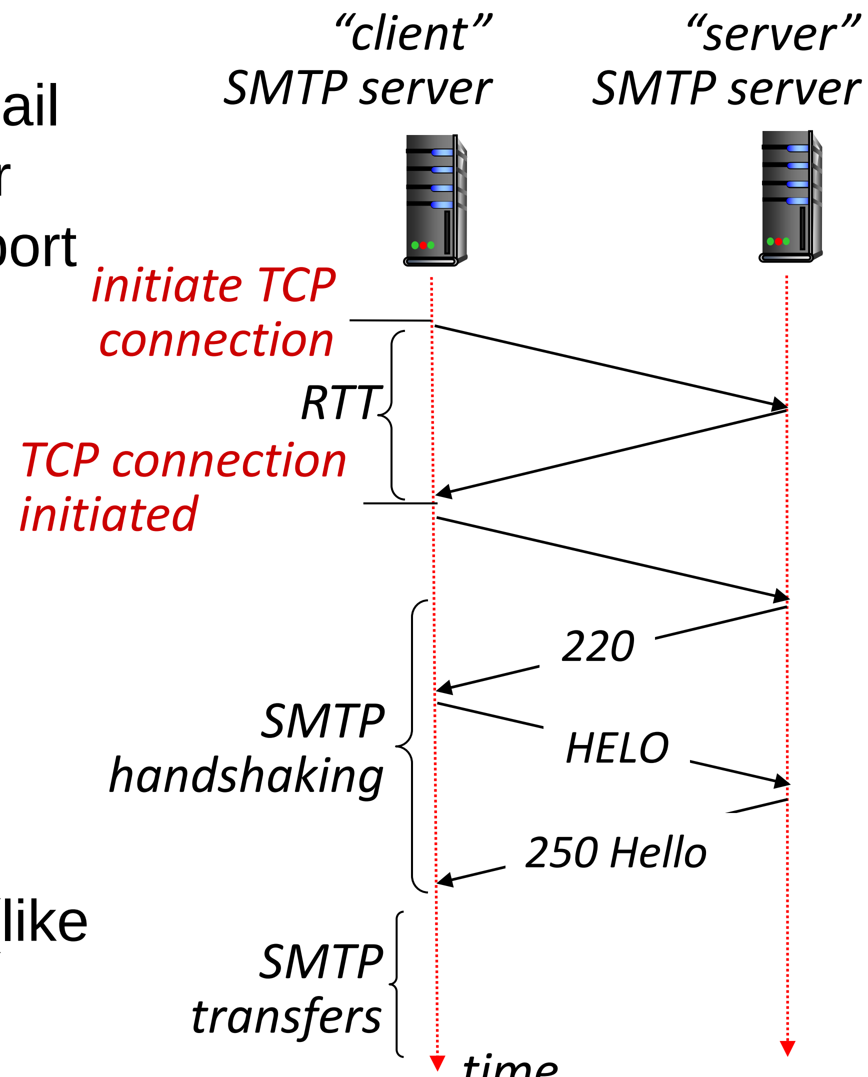
mail servers:

- *mailbox* contains incoming messages for user
- *message queue* of outgoing (to be sent) mail messages
- *SMTP protocol* between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server



Electronic Mail: SMTP [RFC 5321]

- uses TCP to reliably transfer email message from client (mail server initiating connection) to server, port 25
  - direct transfer: sending server (acting like client) to receiving server
- three phases of transfer
  - SMTP handshaking (greeting)
  - SMTP transfer of messages
  - SMTP closure
- command/response interaction (like HTTP)
  - commands: ASCII text
  - response: status code and phrase



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: client pull
- SMTP: client 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

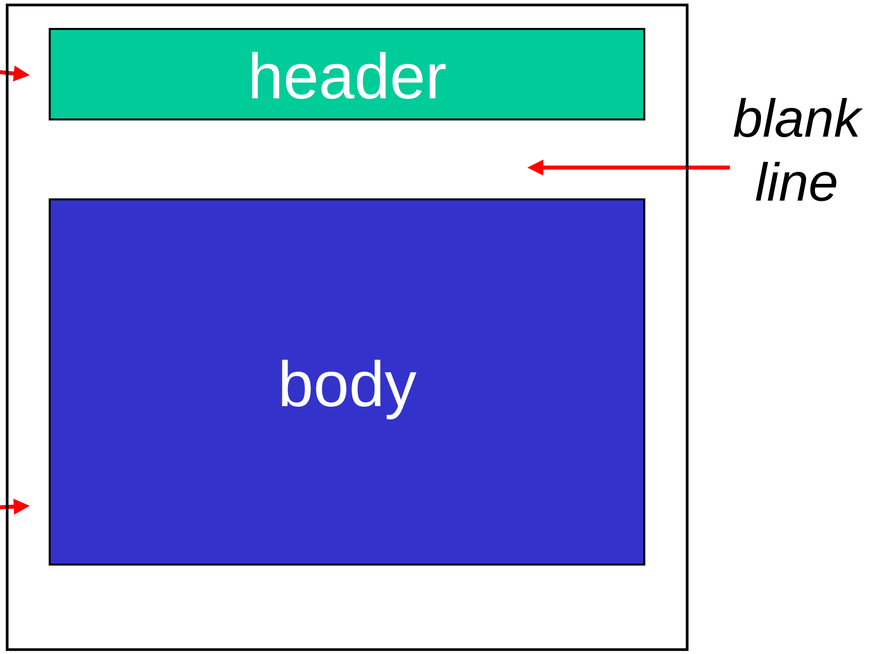
Mail message format

SMTP: protocol for exchanging e-mail messages, defined in RFC 5321 (like RFC 7231 defines HTTP)

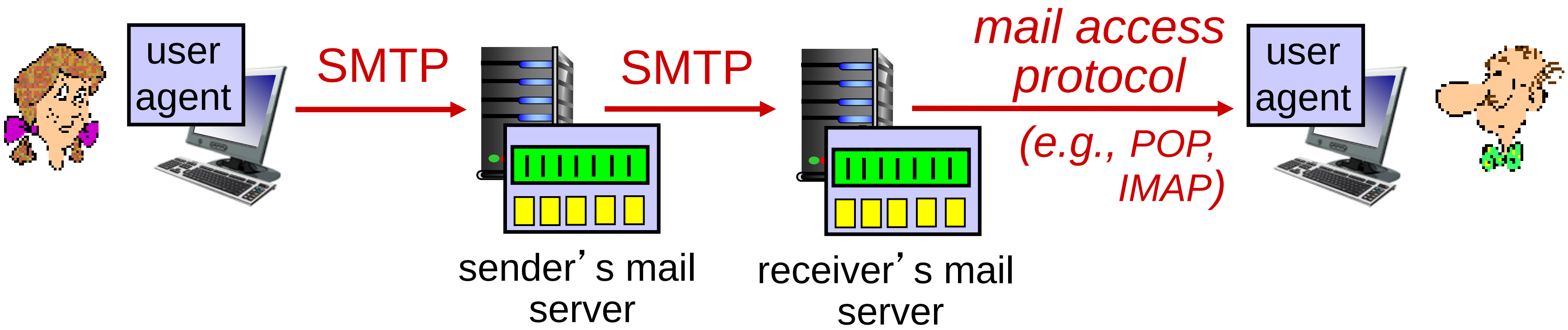
RFC 2822 defines syntax for e-mail message itself (like HTML defines syntax for web documents)

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

  these lines, within the body of the email message area different from SMTP MAIL FROM:, RCPT TO: commands!
- Body: the “message”: ASCII characters only



Mail access protocols



- SMTP: delivery/storage of e-mails to receiver's server
- mail access protocol: retrieval from server
  - IMAP: Internet Mail Access Protocol [RFC 3501]: etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages
  - HTTP: gmail, Hotmail, Yahoo! Mail, etc. provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages

Application Layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 E-mail, SMTP, IMAP
- 2.4 The Domain Name System DNS
- 2.5 P2P applications

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

Q: why not centralize DNS?

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

A: doesn't scale!

- *Comcast DNS servers alone: 600B DNS queries/day*
- *Akamai DNS servers alone: 2.2T DNS queries/day*

Thinking about the DNS

humongous distributed database:

- *~ billion records, each simple*

handles many trillions of queries/day:

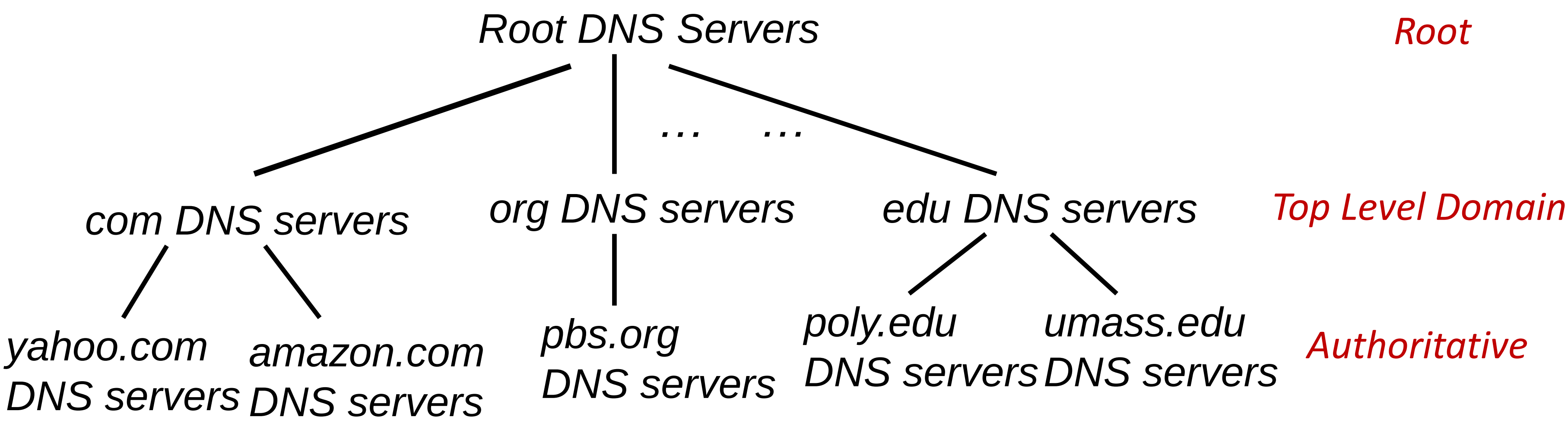
- *many more reads than writes*
- *performance matters: almost every Internet transaction interacts with DNS - msecs count!*

organizationally, physically decentralized:

- *millions of different organizations responsible for their records*

"bulletproof": reliability, security

DNS: a distributed, hierarchical database



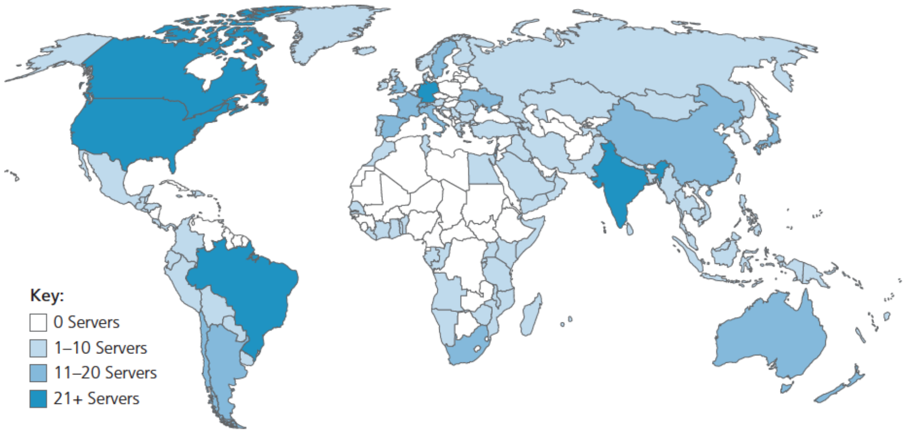
client wants IP for www.amazon.com; 1st approximation:

- 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: root name servers

- official, contact-of-last-resort by name servers that can not resolve name
- *incredibly important* Internet function
  - Internet couldn't function without it!
  - DNSSEC – provides security (authentication, message integrity)
- ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain

13 logical root name "servers" worldwide each "server" replicated many times (~200 servers in US)



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 authoritative registry for .com, .net TLD
- Educause for .edu TLD

authoritative DNS servers:

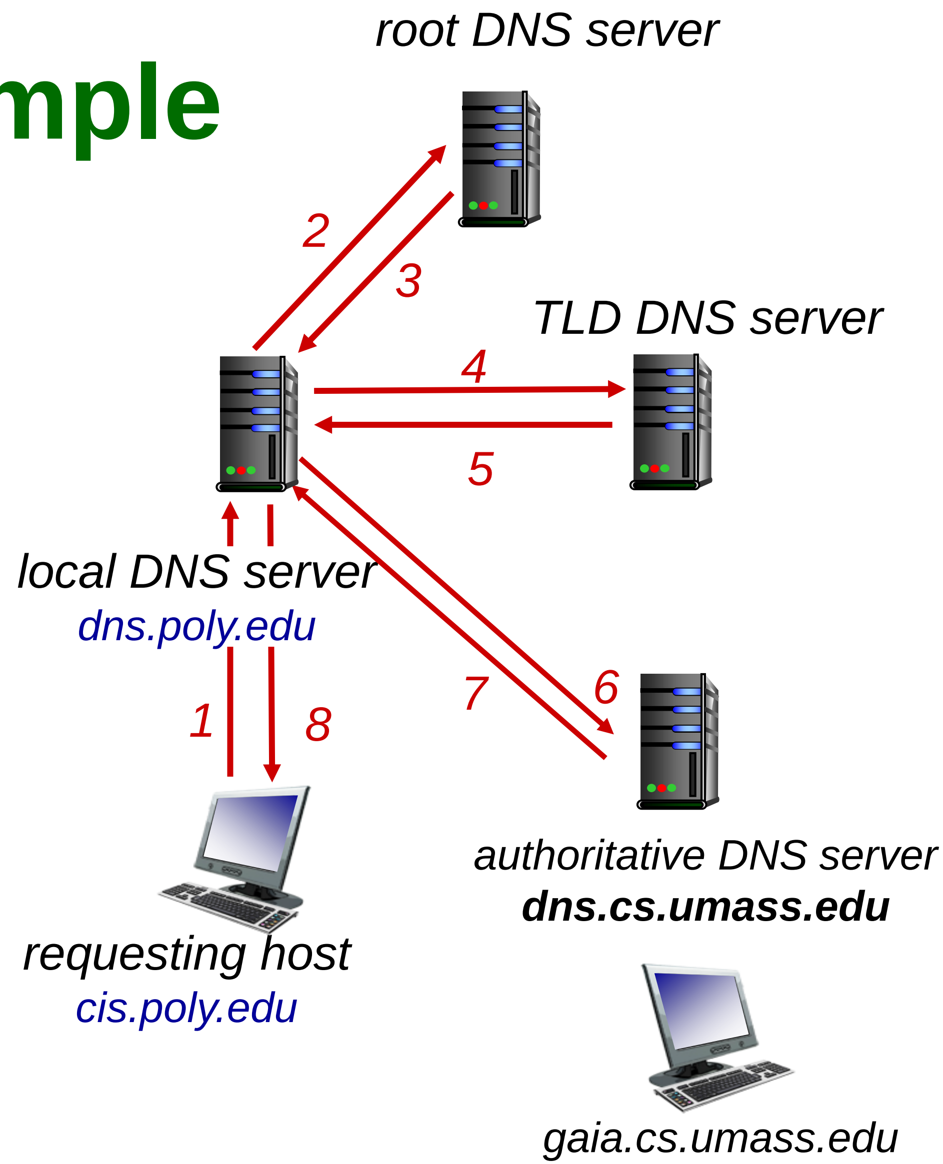
- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

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
  - Local DNS server returns reply, answering:
    - from its local cache of recent name-to-address translation pairs (but may be out of date!)
    - acts as proxy, forwards query into DNS hierarchy for resolution

DNS name resolution example

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



iterated query:

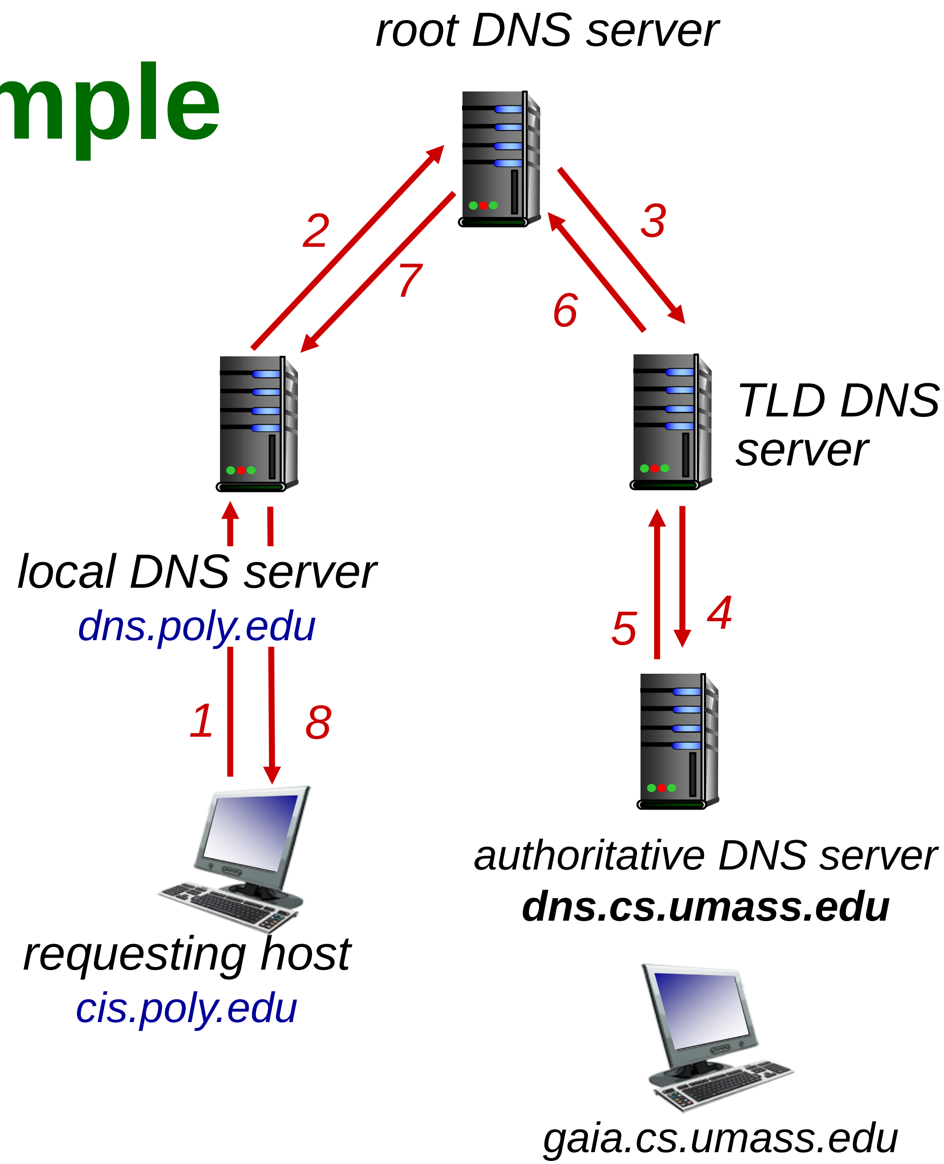
- *contacted server replies with name of server to contact*
- *"I don't know this name, but ask this server"*

DNS name resolution example

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

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 *immediately* returns a cached mapping in response to a query
  - caching improves response time
  - 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*
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
  - *best effort name-to-address translation!*

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

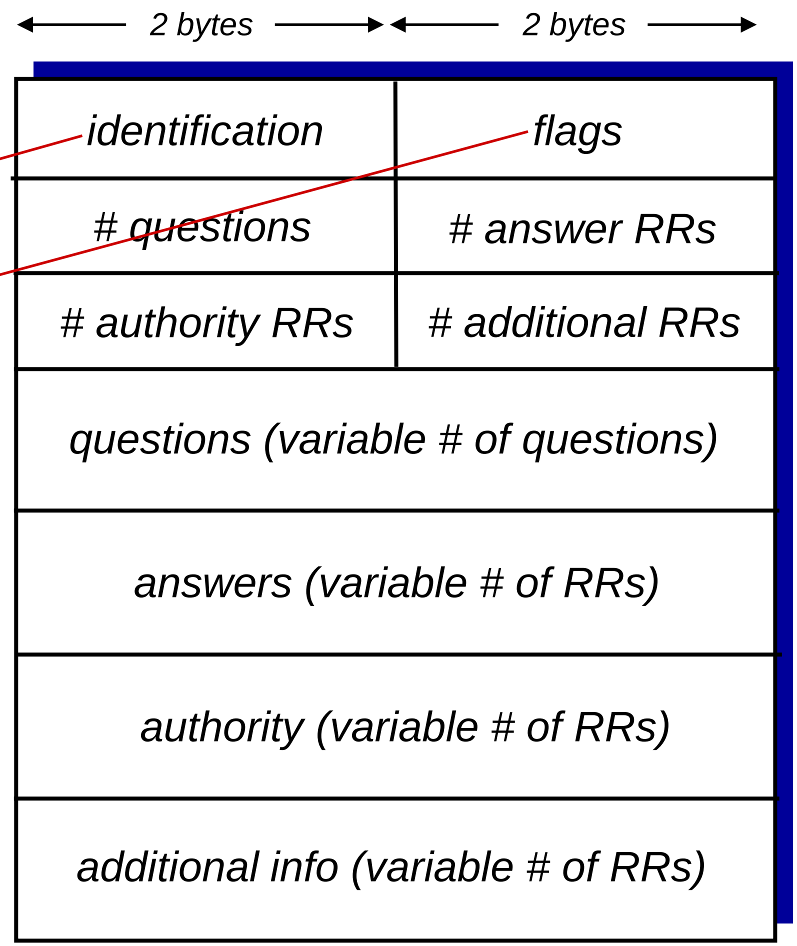
- ***`name`*** *is alias name for some "canonical" (the real) name*
- ***`www.ibm.com`*** *is really* ***`servereast.backup2.ibm.com`***
- ***`value`*** *is canonical name*

type=MX

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

DNS protocol, messages

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

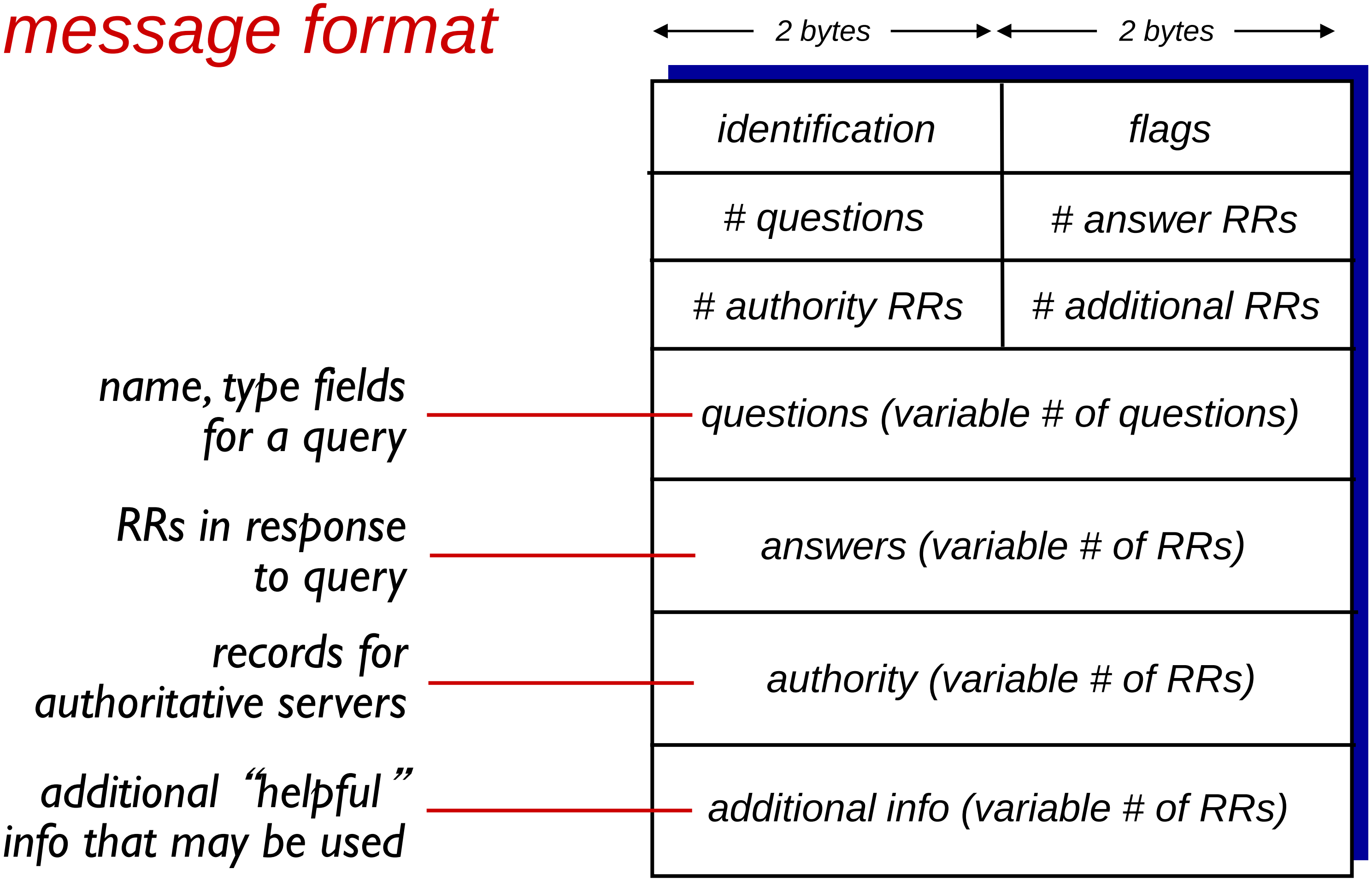


message header

- *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 NS, A RRs into .com TLD server:

    (networkutopia.com, dns1.networkutopia.com, NS)

    (dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server locally with IP address 212.212.212.1
  - type A record for www.networkuptopia.com
  - type MX 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

Spoofing attacks

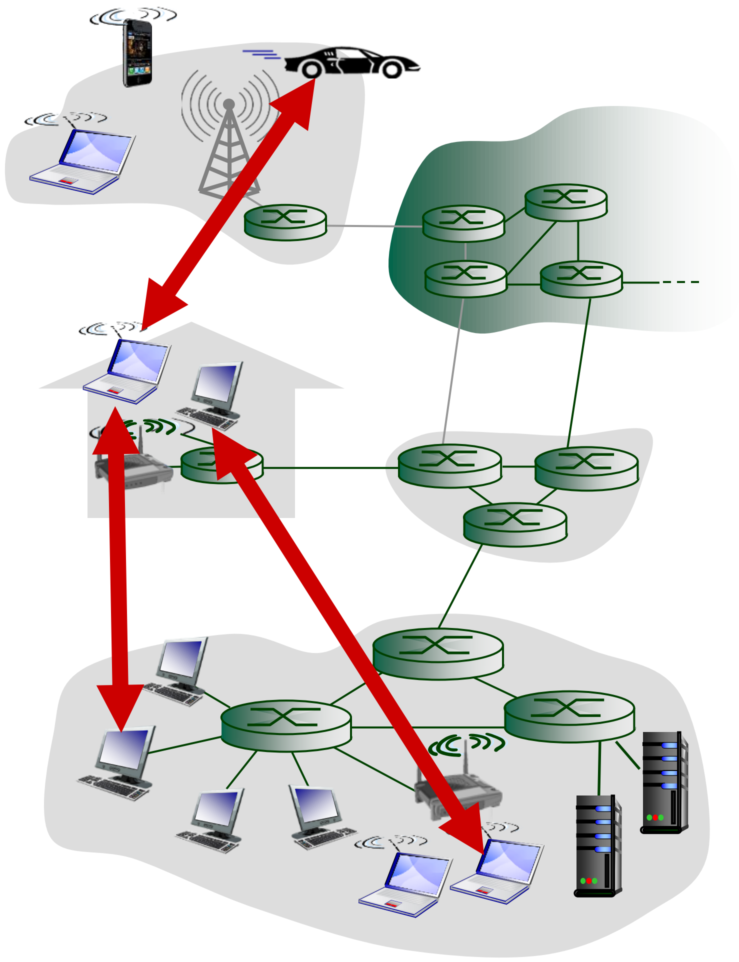
- intercept DNS queries, returning bogus replies
  - DNS cache poisoning
  - RFC 4033: DNSSEC authentication services

Application Layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 E-mail, SMTP, IMAP
- 2.4 The Domain Name System DNS
- 2.5 P2P applications

Pure P2P architecture

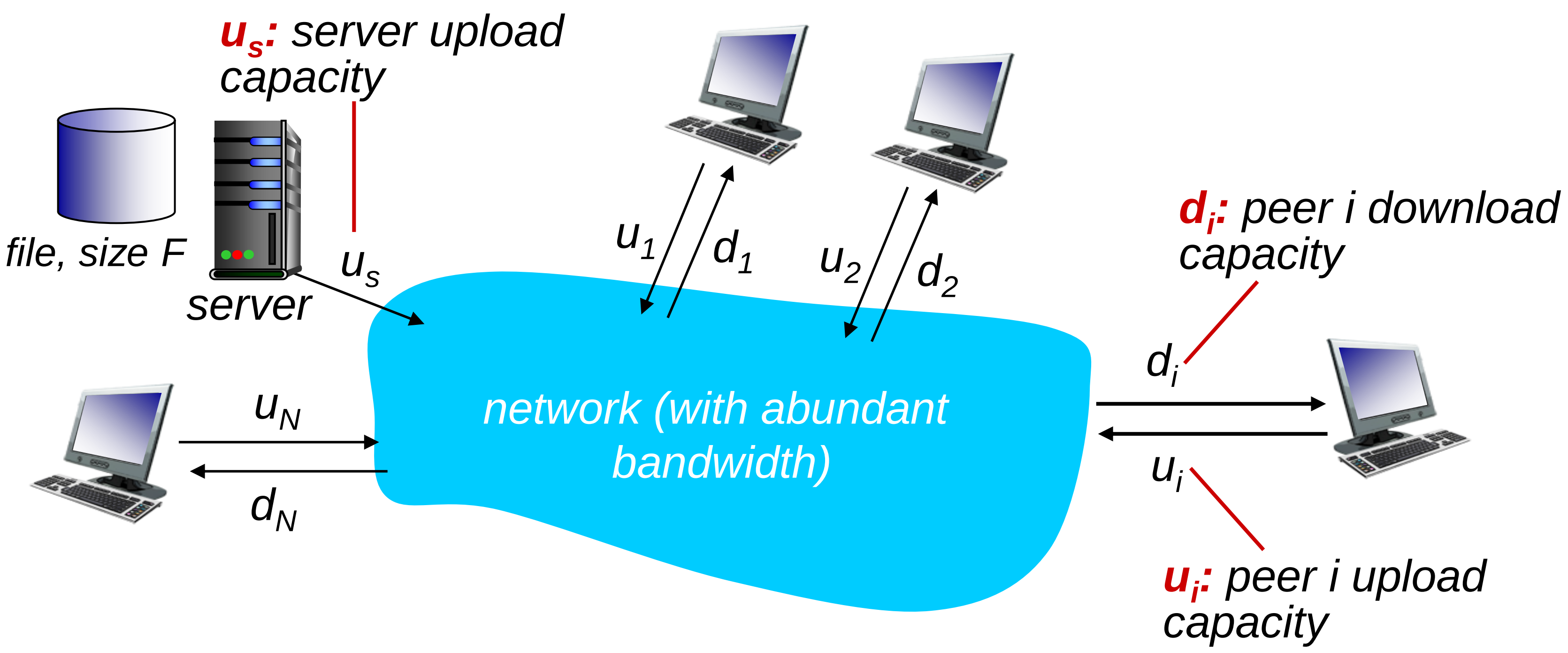
- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
  - *examples:* file distribution (BitTorrent), Streaming (KanKan), VoIP (Skype)



File distribution: client-server vs P2P

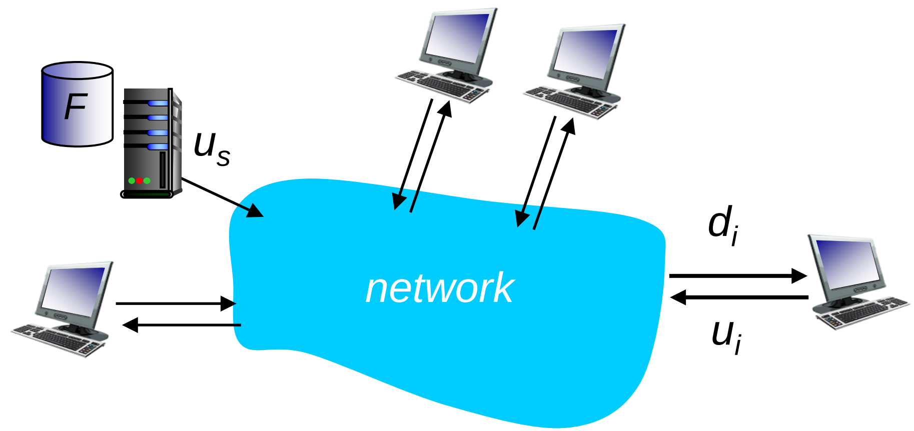
Question: how much time to distribute file (size *F*) from one server to *N* *peers*?

- peer upload/download capacity is limited resource



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}



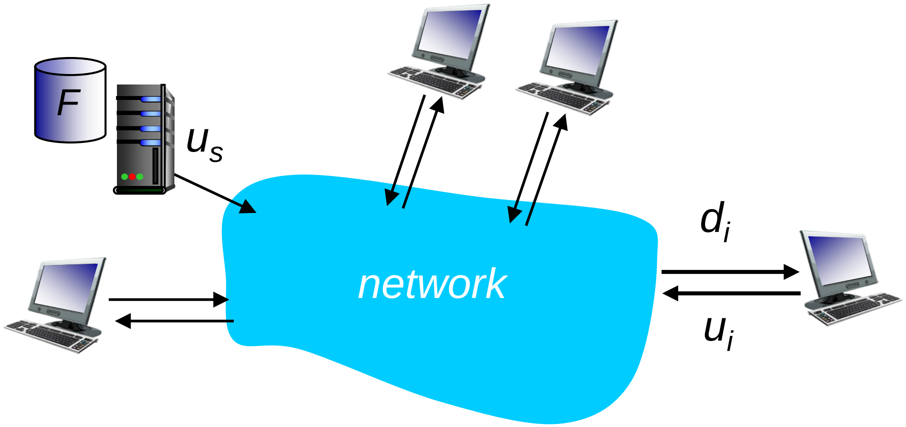
time to distribute F to N clients using client-server approach

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

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

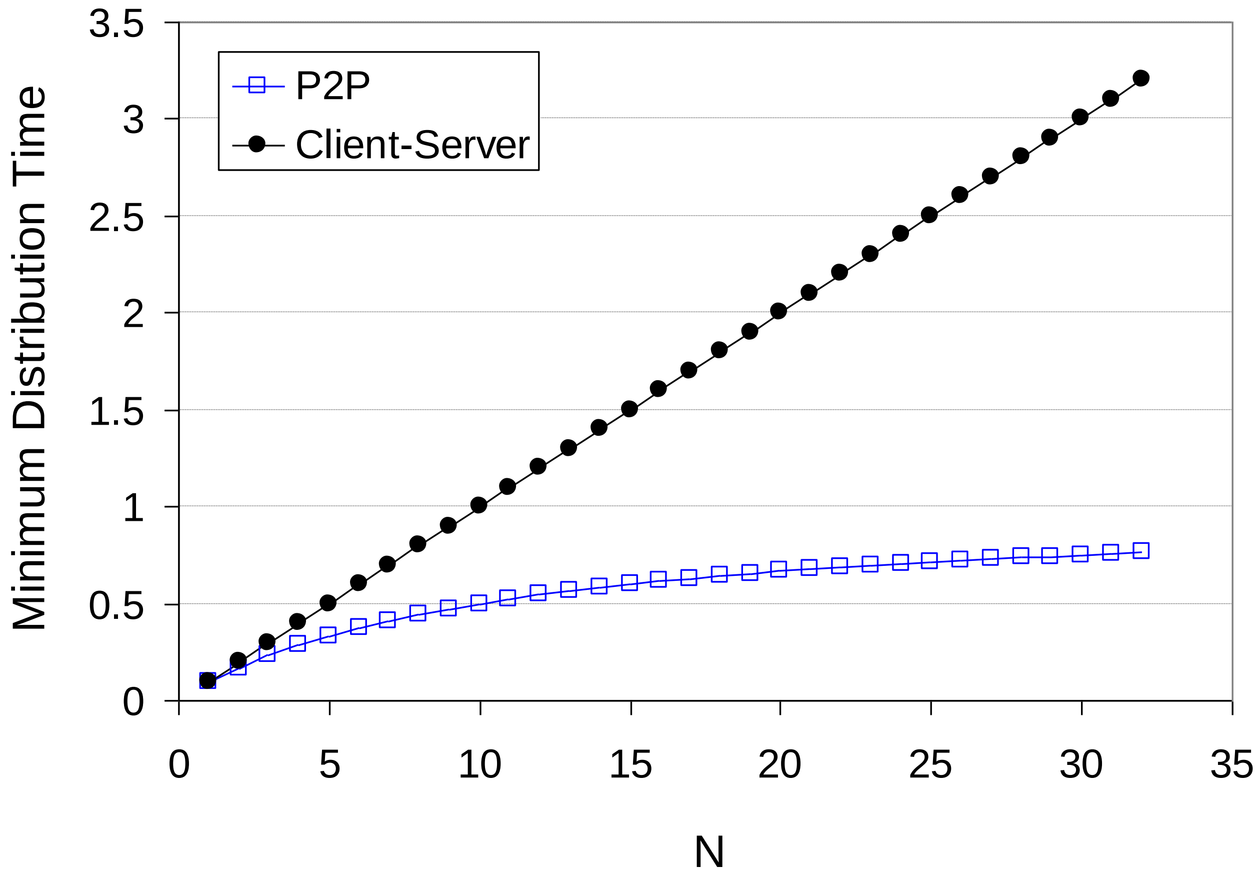
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in N …

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

Client-server vs. P2P: example

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



Chapter 2: summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS

Chapter 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:

- *centralized vs. decentralized*
- *stateless vs. stateful*

Additional slides

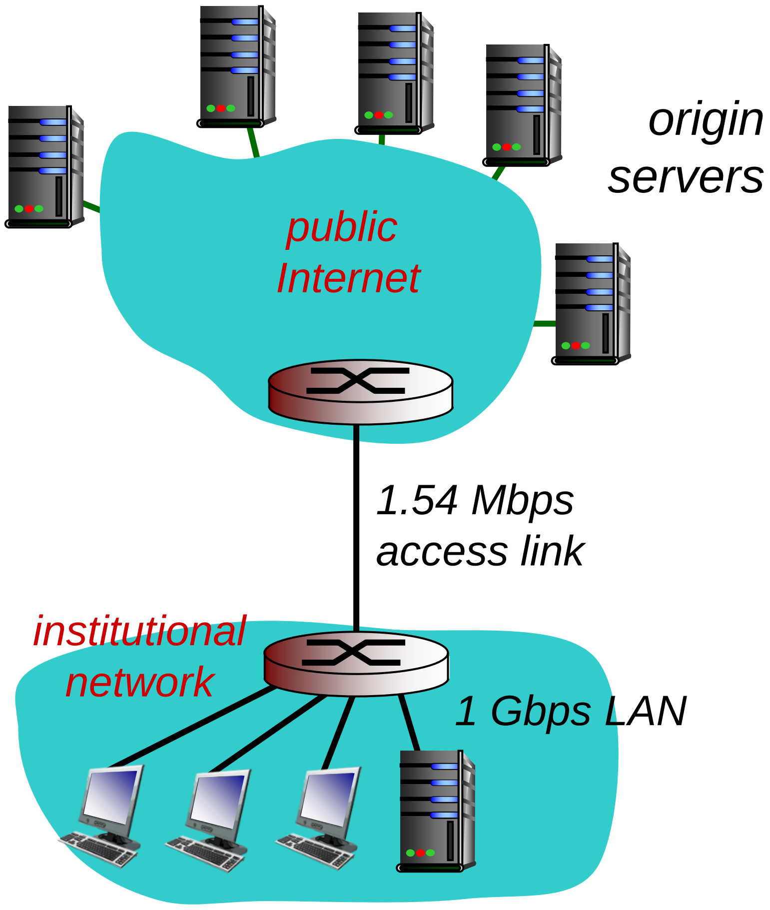
Caching example:

assumptions:

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

consequences:

- LAN utilization: 15%
- access link utilization = 99% problem!
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + usecs



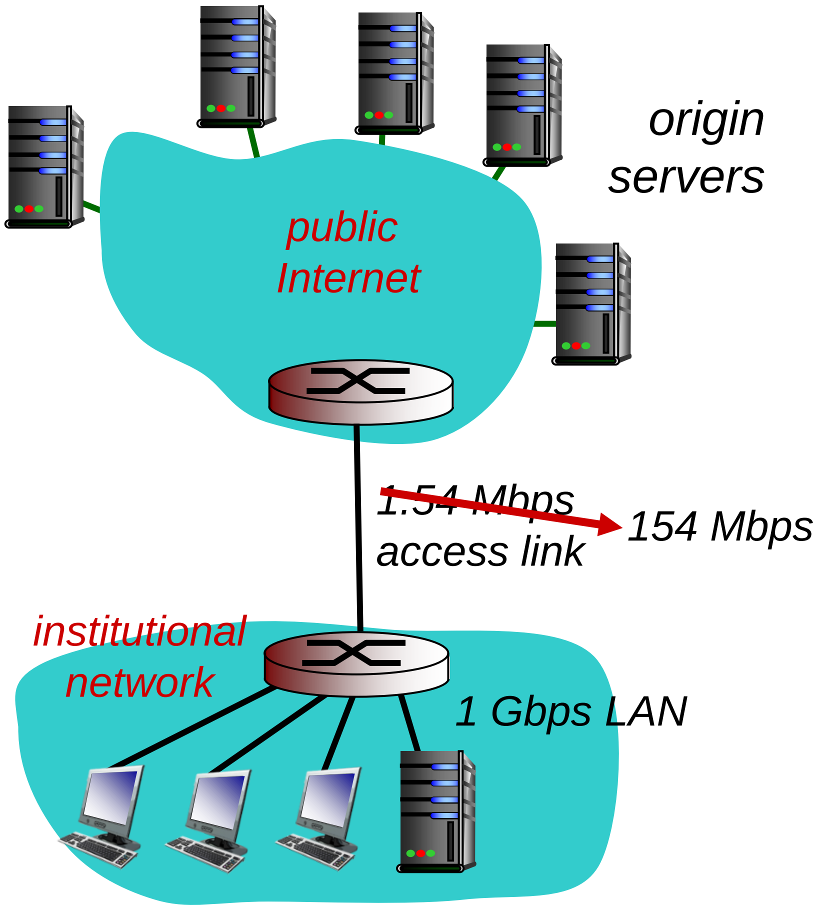
Caching example: fatter access link

assumptions:

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

consequences:

- LAN utilization: 15%
- access link utilization = ~~99%~~ → 9.9%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + ~~minutes~~ + usecs → msecs



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

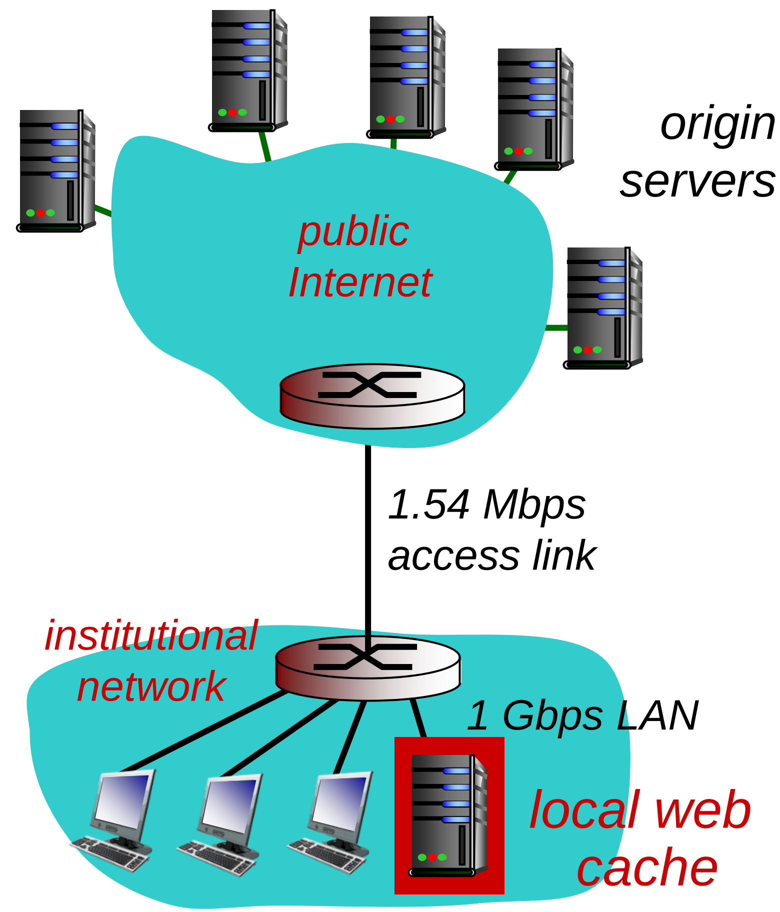
- *avg object size: 100K bits*
- *avg request rate from browsers to origin servers:15/sec*
- *avg data rate to browsers: 1.50 Mbps*
- *RTT from institutional router to any origin server: 2 sec*
- *access link rate: 1.54 Mbps*

consequences:

- *LAN utilization: 15%*
- *access link utilization = ?*
- *total delay = ?*

How to compute link utilization, delay?

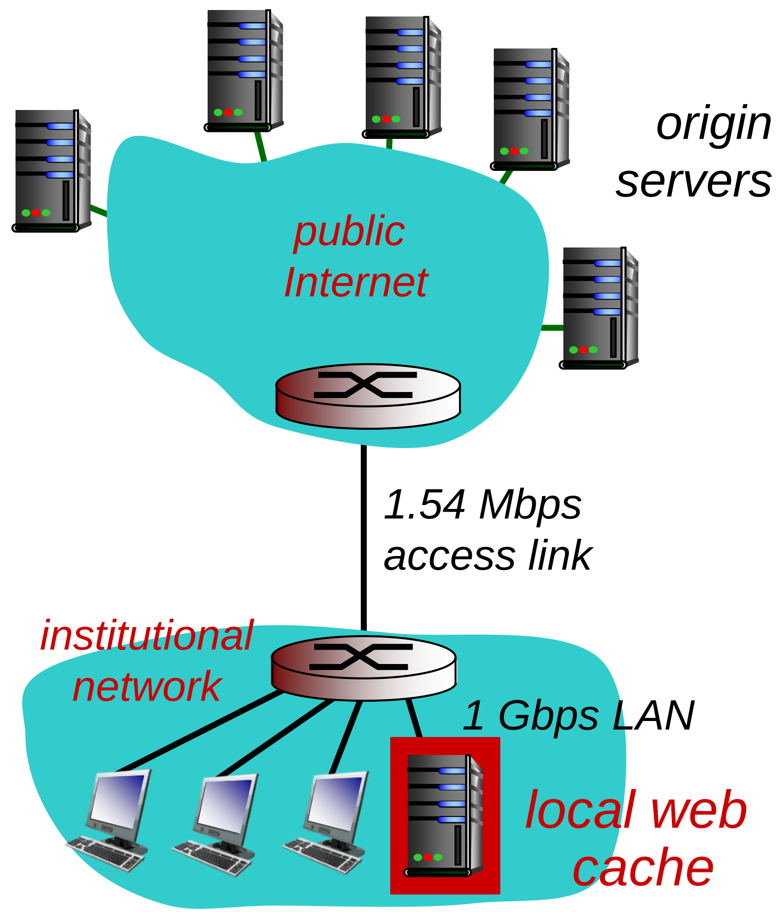
Cost: web cache (cheap!)



Caching example: install local cache

Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin
- *access link utilization:*
  - *60% of requests use access link*
- *data rate to browsers over access link*
  - *= 0.6*1.50 Mbps = .9 Mbps*
  - *utilization = 0.9/1.54 = .58*
- *total delay*
  - *= 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)*
  - *= 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs*
  - *less than with 154 Mbps link (and cheaper too!)*



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
```

Scenario: Alice sends message to Bob

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

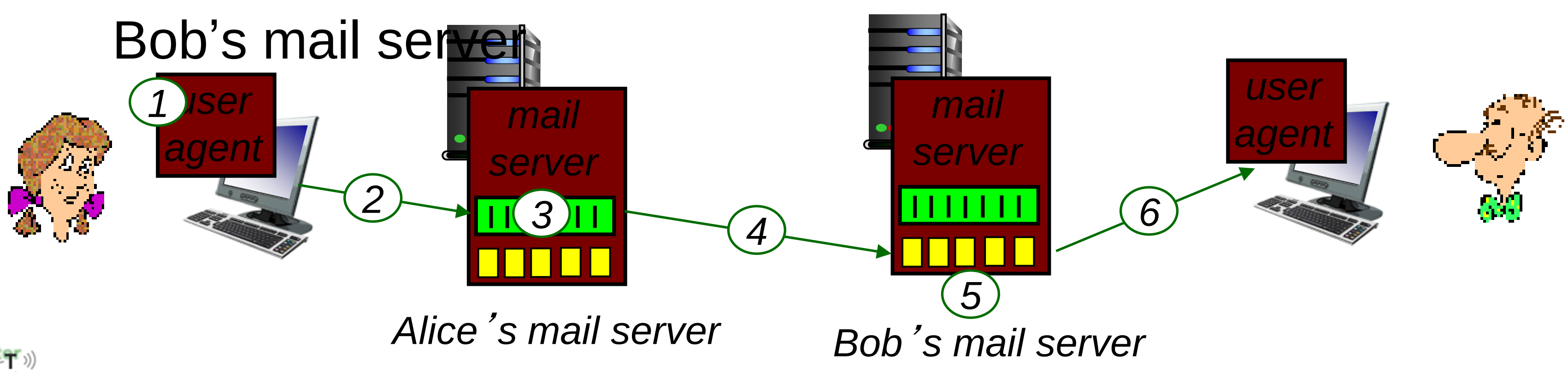
2) Alice's UA sends message to her mail server; message placed in message queue

3) client side of SMTP opens TCP connection with Bob's mail server

4) SMTP client sends Alice's message over the TCP connection

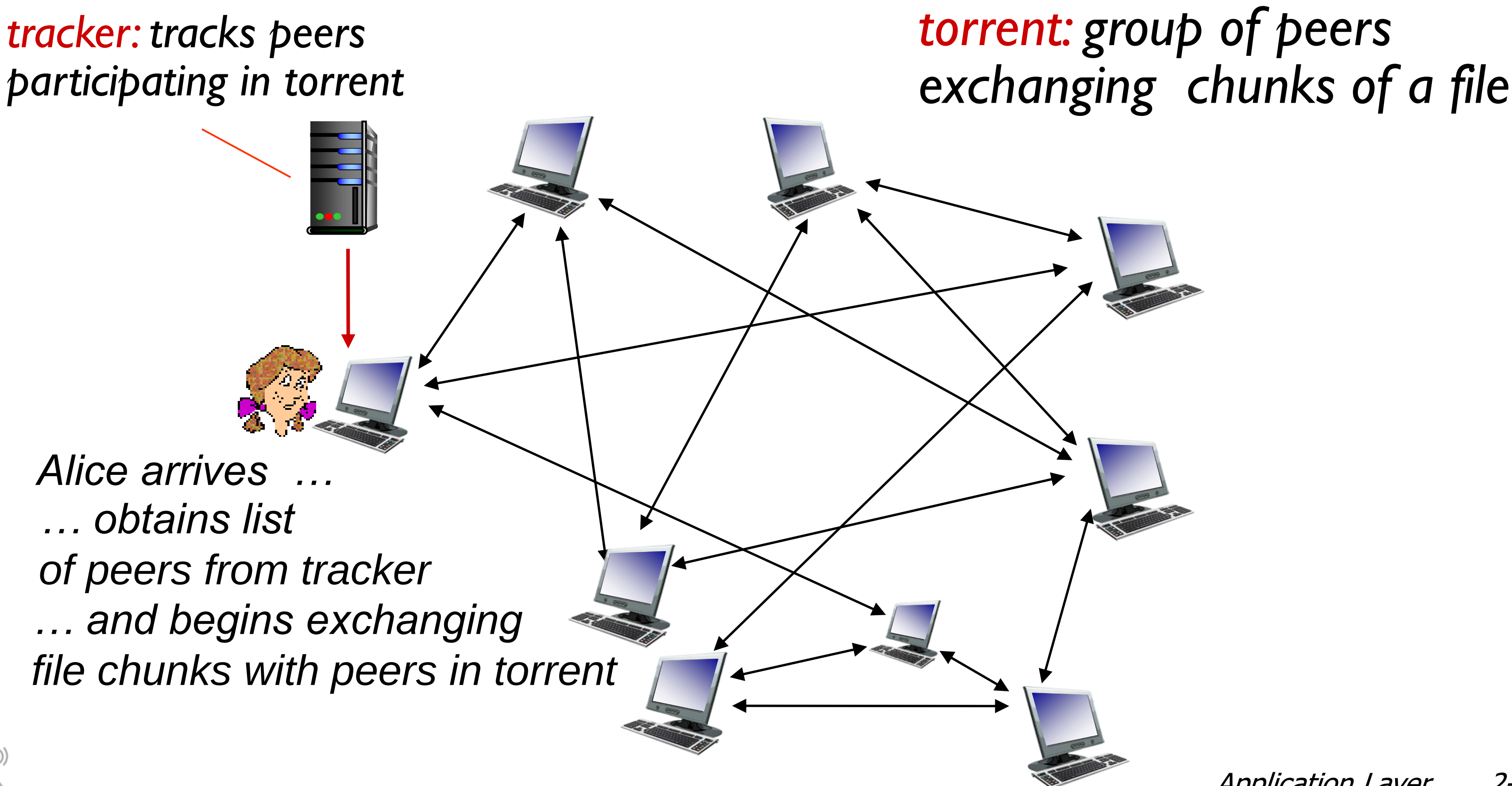
5) Bob's mail server places the message in Bob's mailbox

6) Bob invokes his user agent to read message



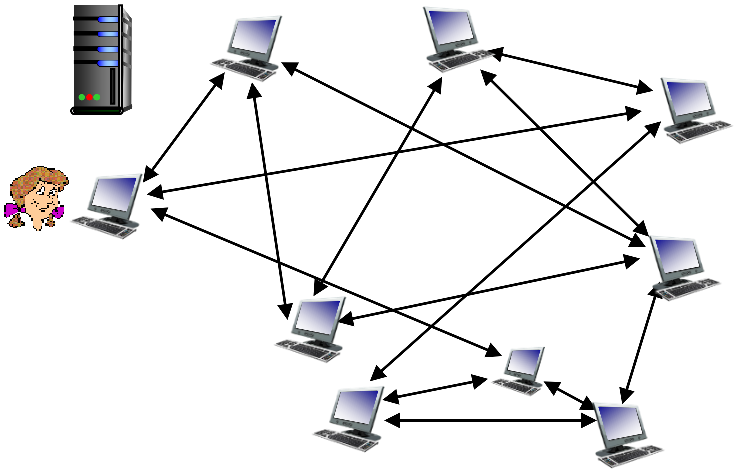
P2P file distribution: BitTorrent

- 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

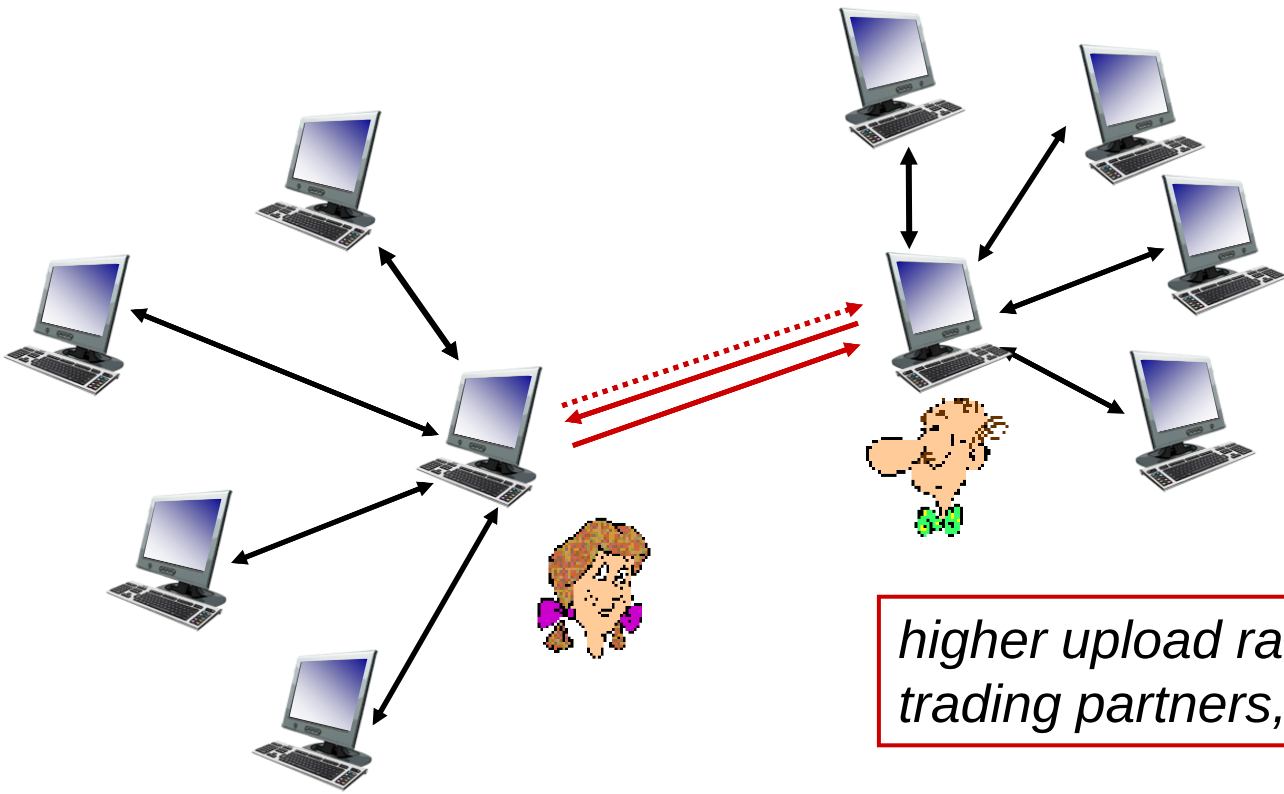
- Alice sends chunks to those four peers currently sending her chunks at highest rate
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  -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