CS509

Computer Networks

Lecture Two

Lecture Two Outline

Chapter 2: Application Layer

2.1 Principles of Network Applications

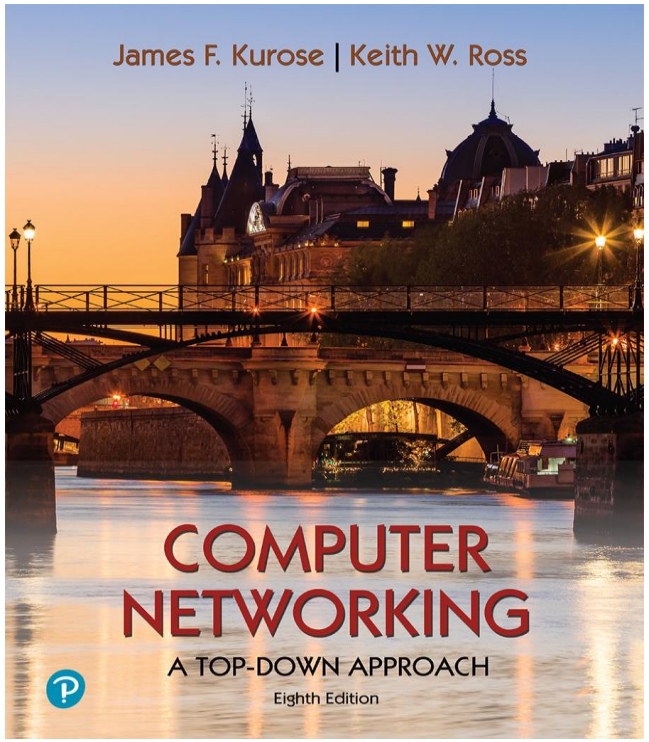
2.2 The Web and HTTP

2.3 Electronic Mail in the Internet

2.4 DNS-The Internet's Directory Service

2.5 Peer-to-Peer File Distribution

2.6 Video Streaming and Content Distribution Networks



Computer Networking: A Top-Down Approach

8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS

- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Some network apps

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

Q: your favorites?

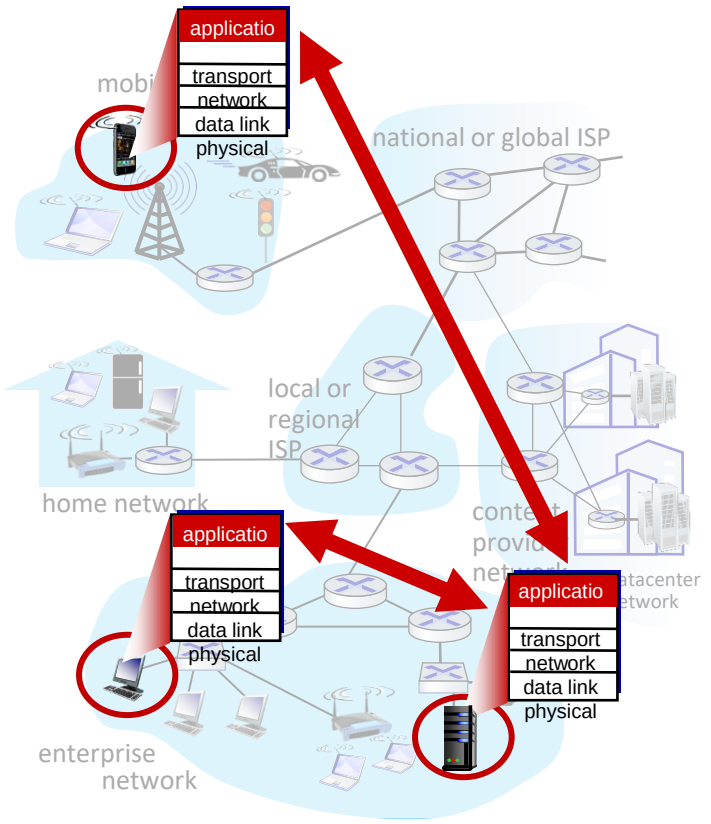
Creating a network app

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 paradigm

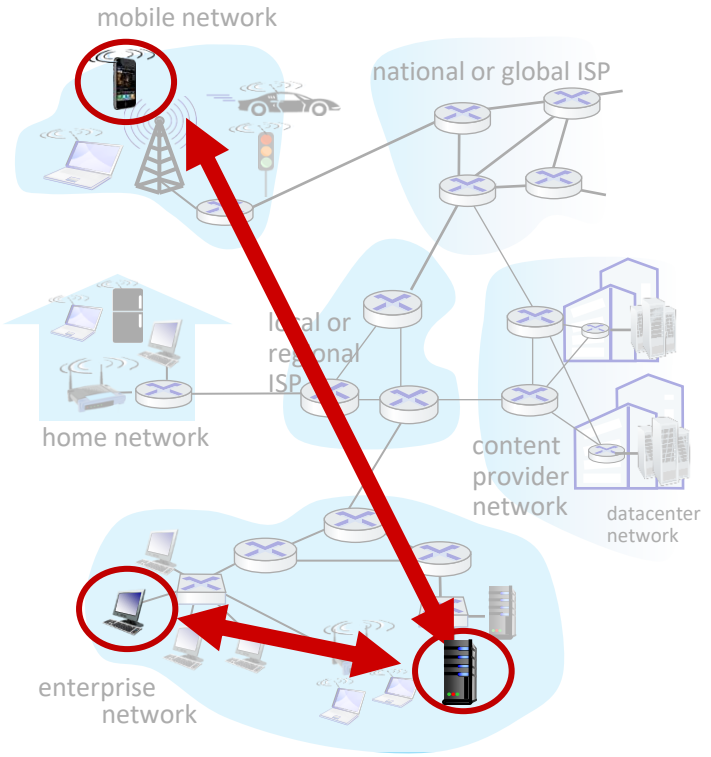
server:

- always-on host
- serves requests from many other hosts, called clients
- has permanent IP address
- often in data centers, for scaling

clients:

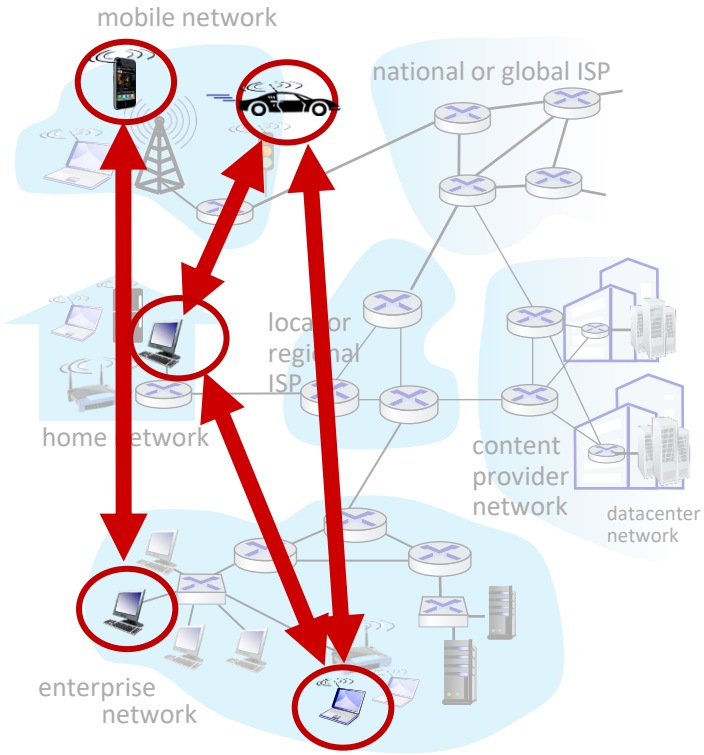
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



Peer-peer (P2P) architecture

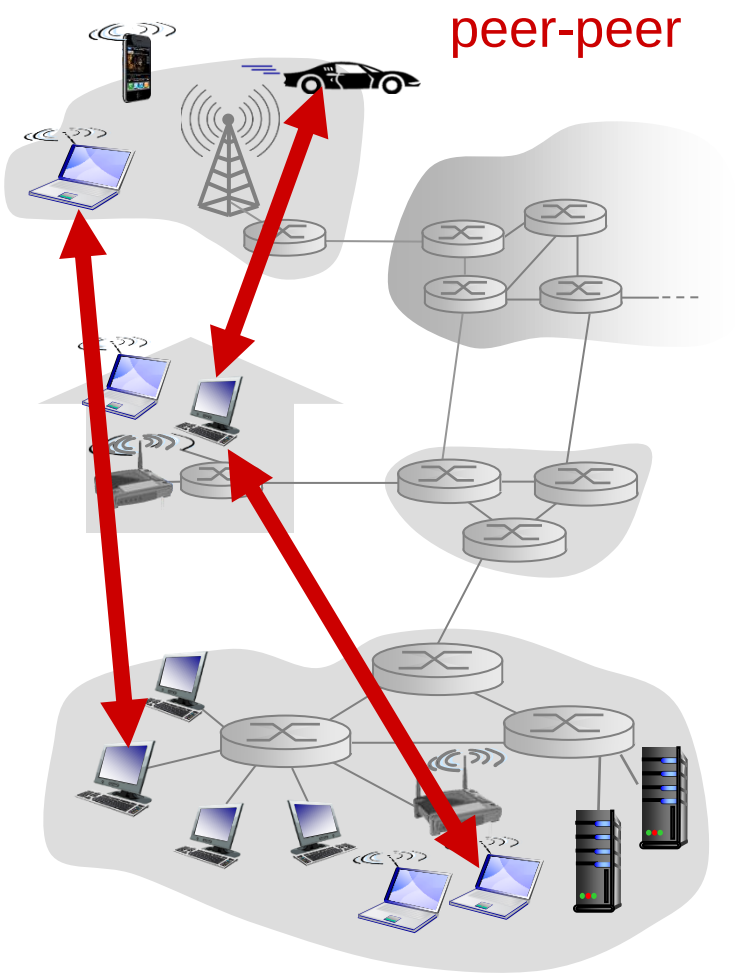
- *no* always-on dedicated server
- peers are not owned by the service provider, but controlled by users
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- example: P2P file sharing (BitTorrent)



P2P architecture-2

- advantage:
  - P2P architecture is more cost effective than client-server architecture, since it normally don't require significant server infrastructure and server bandwidth allocation in contrast with clients-server designs with datacenters

- disadvantages:
  - complex management: peers are intermittently connected and change their IP addresses
  - security: because of their highly distributed and open nature, P2P applications can be a challenge to secure



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

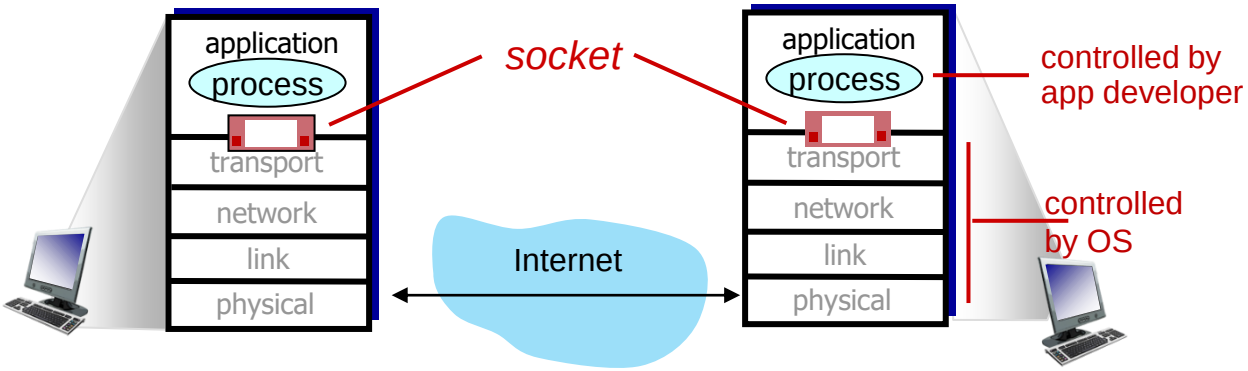
client process: process that initiates communication

server process: process that waits to be contacted

- note: applications with P2P architectures have client processes (download) & server processes (upload)

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
  - two sockets involved: one on each side



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…

An application-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, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype, Zoom

Application-layer protocol-2

- RFCs started out as general requests for comments (hence the name) to resolve network and protocol design problems that faced the precursor to the internet.

- RFCs tend to be quite technical and detailed and there are currently more than 6,000 RFCs.

- some application-layer protocols (e.g., web HTPP, email SMTP) are specified in RFCs and are therefore in the public domain. For example, If a browser developer follows the rules of the HTTP RFC, the browser will be able to retrieve web pages from any web server that has also followed the rules of the HTTP RFC

- many other application-layer protocols are proprietary and intentionally not available in the public domain. For example, Skype uses proprietary application-layer protocols.

Application-layer protocol-3

- it is important to distinguish between network applications and application-layer protocols because an application-layer protocol is only one piece of a network application

- for example, the web application consists of many components, including a standard for document formats (HTML), web browsers (e.g., Firefox and Chrome), web servers (e.g., Apache and Microsoft Internet Information Server), and an application-layer protocol (HTTP) that only defines the format and sequence of messages exchanged between browser and web server.

- another example, the email application also has many components, including mail servers (e.g., Hotmail and Gmail); mail clients (e.g., Microsoft Outlook), and application-layer protocols (e.g., SMTP and POP3) that are standards for defining the structure of an email message, how the contents of an email message headers are to be interpreted and how messages are passed between mail servers and mail clients.

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web banking transactions) require 100% reliable data transfer (no error & no loss)
- other apps (e.g., audio) can tolerate some loss

throughput

- some apps, “bandwidth-sensitive apps”, (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

security

- confidentiality (encryption & decryption), end-point authentication, …

Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	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, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10'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
- *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, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP?

Internet transport protocols services-2

TCP services:

- *connection-oriented service:* connection setup required between client and server processes.

A. TCP has the client and server exchange transport layer control information with each other *before* the application-level messages begin to flow (handshaking procedure)

B. After the handshaking phase, a TCP connection is said to exist between the sockets of the two processes.

C. The connection is a full-duplex connection in that the two processes can send messages to each other over the connection at the same time.

D. When the application finishes sending messages, it must tear down the connection.

Internet applications, and transport protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP [RFC 5321]	TCP
Remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP 1.1 [RFC 7230]	TCP
File transfer	FTP [RFC 959]	TCP
Streaming multimedia	HTTP (e.g., YouTube), DASH	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	UDP or TCP

Figure 2.5 ♦ Popular Internet applications, their application-layer protocols, and their underlying transport protocols

- Because Internet telephony applications (such as Skype) can often tolerate some loss but require a minimal rate to be effective, their developers usually prefer to run their applications faster over UDP.

- But because many firewalls are configured to block most of the UDP traffic, Internet telephony applications often are designed to use TCP as a backup if UDP communication fails.

Securing TCP

TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn
- cleartext sent into "socket" traverse Internet *encrypted*
- more: Chapter 8

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Web and HTTP

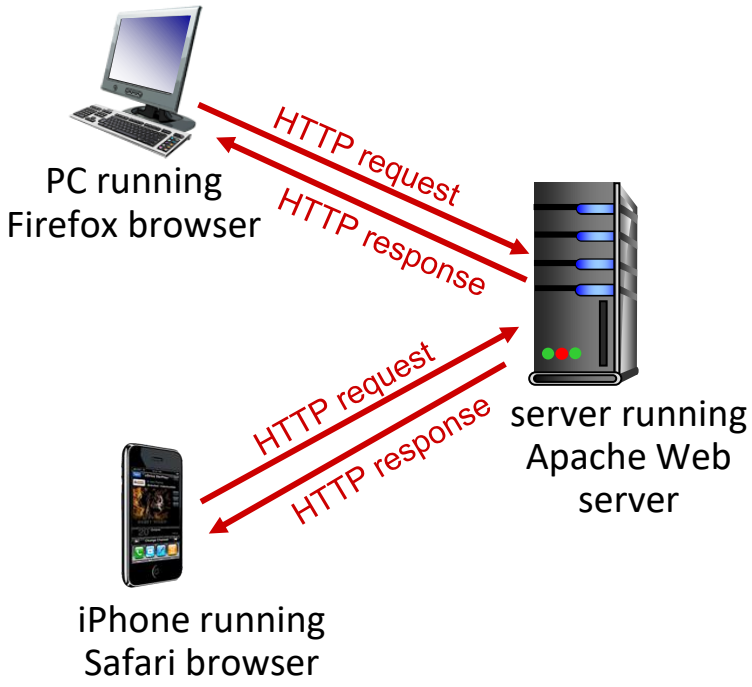
First, a quick review…

- 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* addressable by a *URL,* e.g.,

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)

HTTP 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 (if a particular client requests the same object twice in a period of a few second, the server resends the object)

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: example

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing 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: example (cont.)

User enters URL: **`www.someSchool.edu/someDepartment/home.index`**
(containing text, references to 10 jpeg images)

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

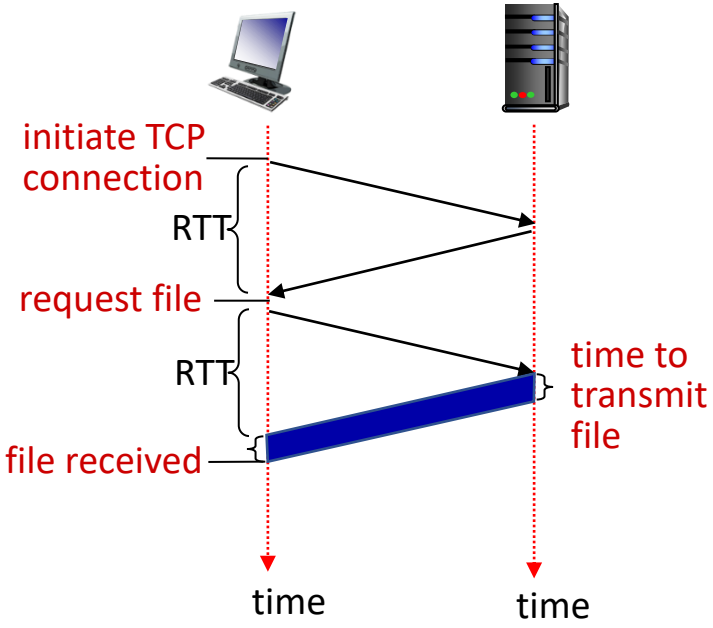
time

Non-persistent HTTP: response time

Round-Trip Time (RTT): time for a small packet to travel from client to server and back

HTTP response time (per object):

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time

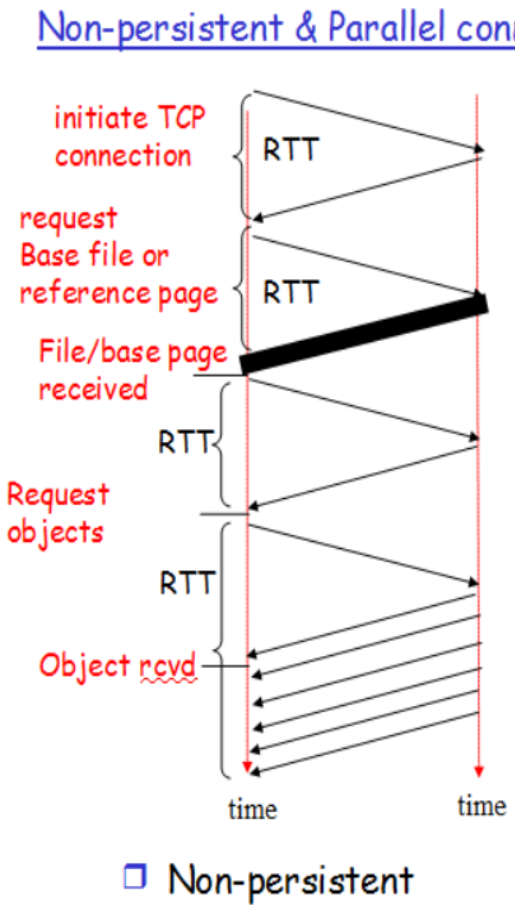


Non-persistent HTTP response time = 2RTT+ file transmission time

Non-persistent HTTP (HTTP 1.1)

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

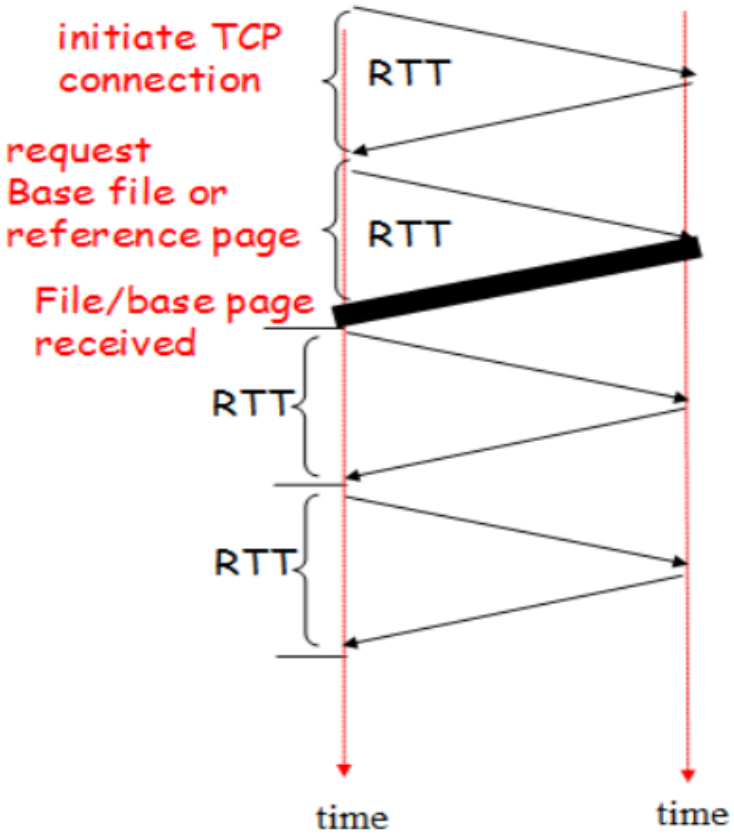


Persistent HTTP

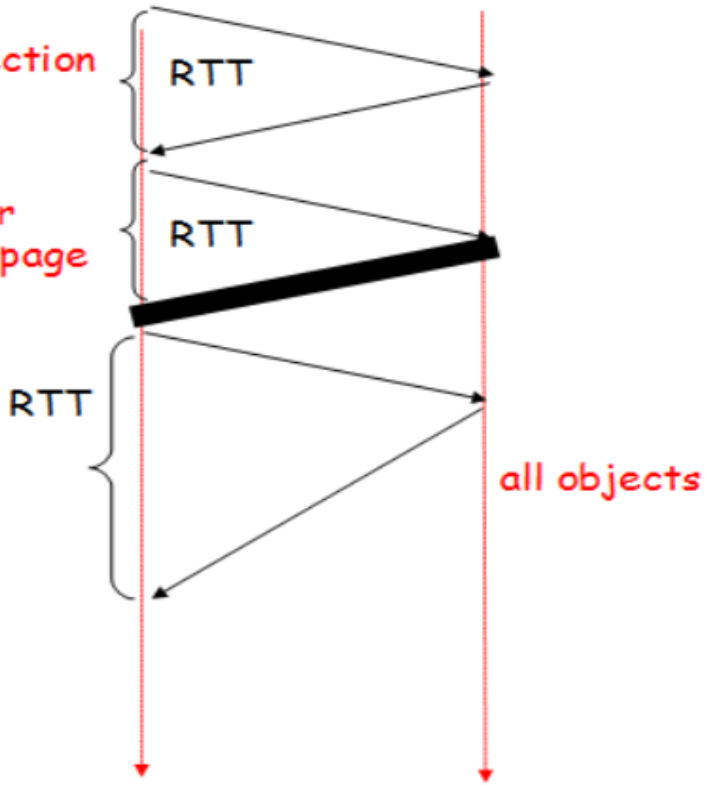
- server leaves TCP connection open after sending response
- client sends request as soon as it encounters a referenced object
- subsequent HTTP messages between same client/server are sent over the same open TCP connection
  - multiple web pages residing on the same server can be sent back-to-back from the server to the same client, because the server receives back-to-back requests for their. back-to-back means sending messages without waiting for replies to pending requests (pipelining)
- HTTP server closes a connection when it isn't used for a certain time
- The main advantage of persistent HTTP is that, it requires as little as one RTT for all the referenced objects

Persistent HTTP - 2

Persistent & Pipelined/non-pipelined connections



- Persistent without pipelining



- Persistent with pipelining

HTTP request message

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

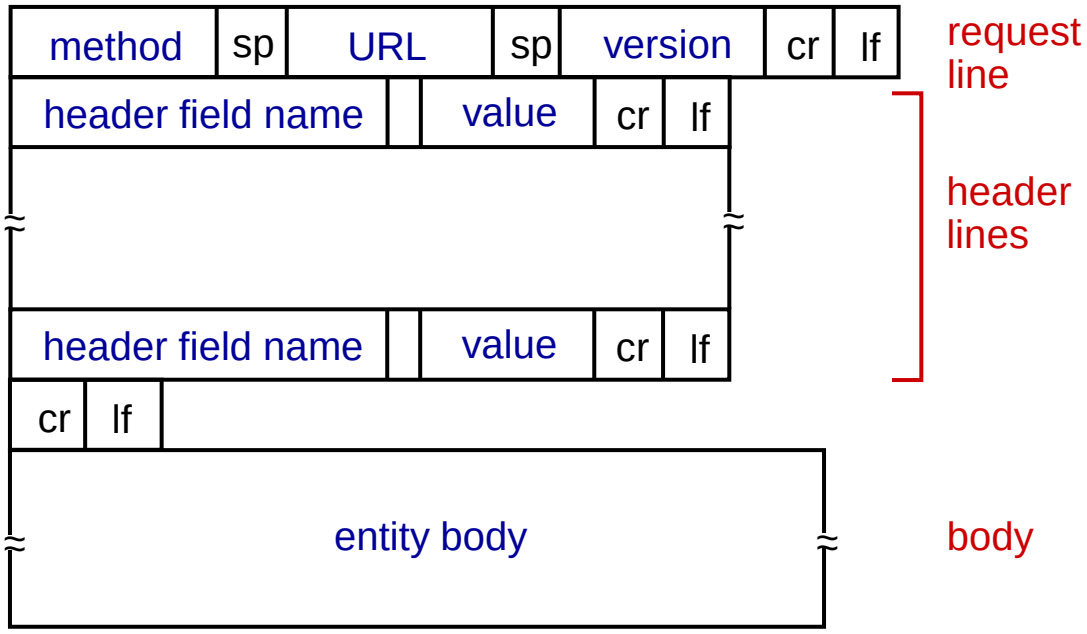
carriage return character
line-feed character

request line (GET,
POST,
HEAD commands)

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

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

HTTP request message: general format



HTTP request message: method types

HTTP/1.0:

- GET: for sending data to server
  - include user data in the URL field of the request line of HTTP GET request message (following a '?'):

```
www.somesite.com/animalsearch?monkeys&banana
```

- POST: web page often includes form inputs
  - user inputs sent from client to server in entity body of HTTP POST request message
- HEAD: asks server to leave requested object out of response
  - Used by developer for debugging

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads object (file) in entity body to path specified in URL field (replace if exists)
- DELETE
  - deletes existent object (file) specified in the URL field

HTTP response message

status line (protocol status code status phrase) → `HTTP/1.1 200 OK`

HTTP response message: general format

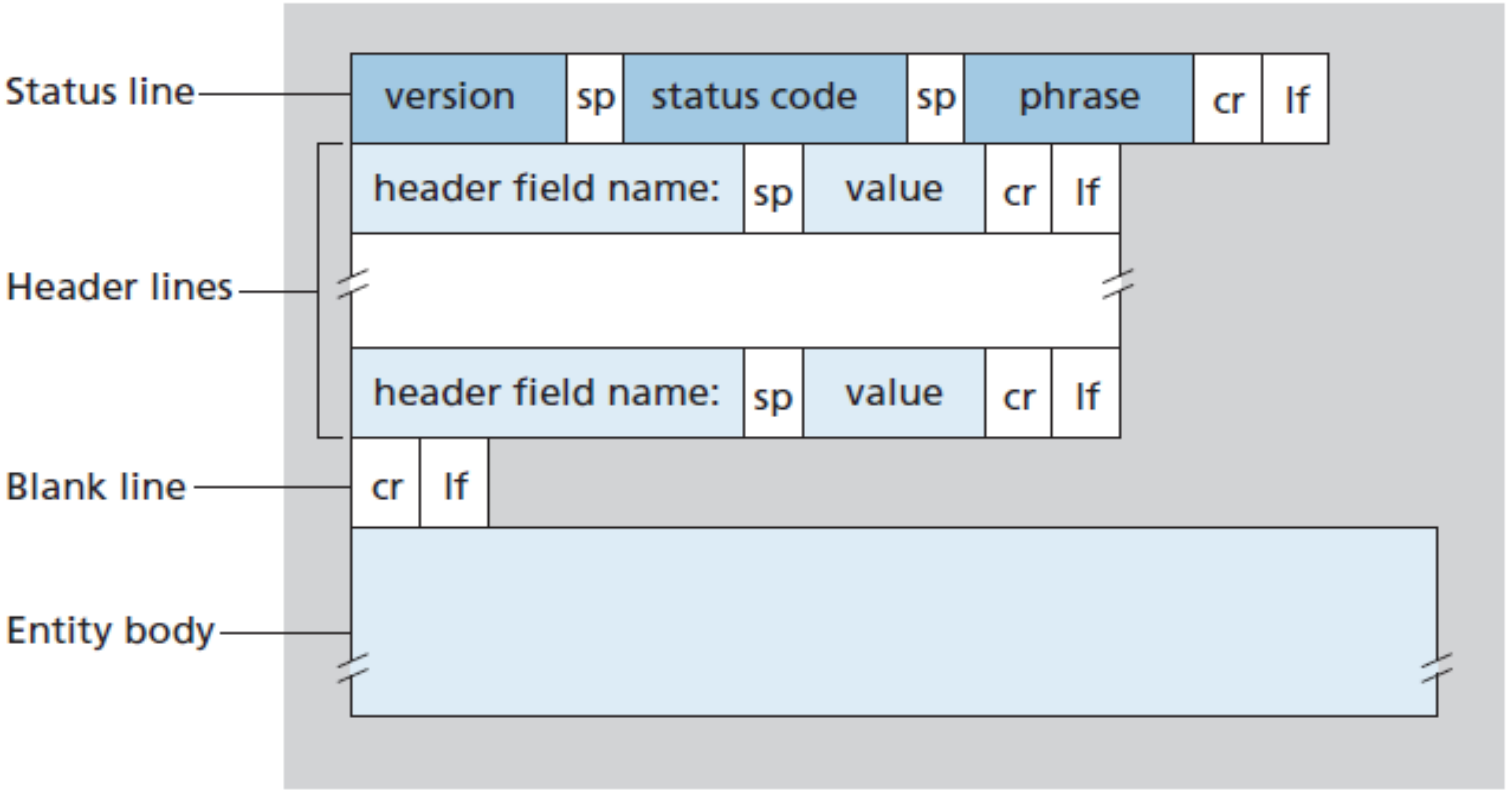


Figure 2.9 ♦ General format of an HTTP response message

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 message

  301 Moved Permanently
  - requested object moved, new location specified later in this message (in Location: field)

  400 Bad Request
  - request msg not understood by server

  404 Not Found
  - requested document not found on this server

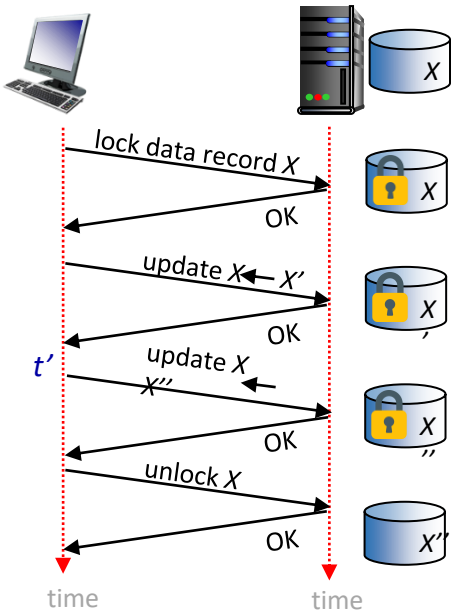
  505 HTTP Version Not Supported

Maintaining 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'* ?

Maintaining user/server state: cookies

Web sites and client browser use *cookies* to maintain some state between transactions

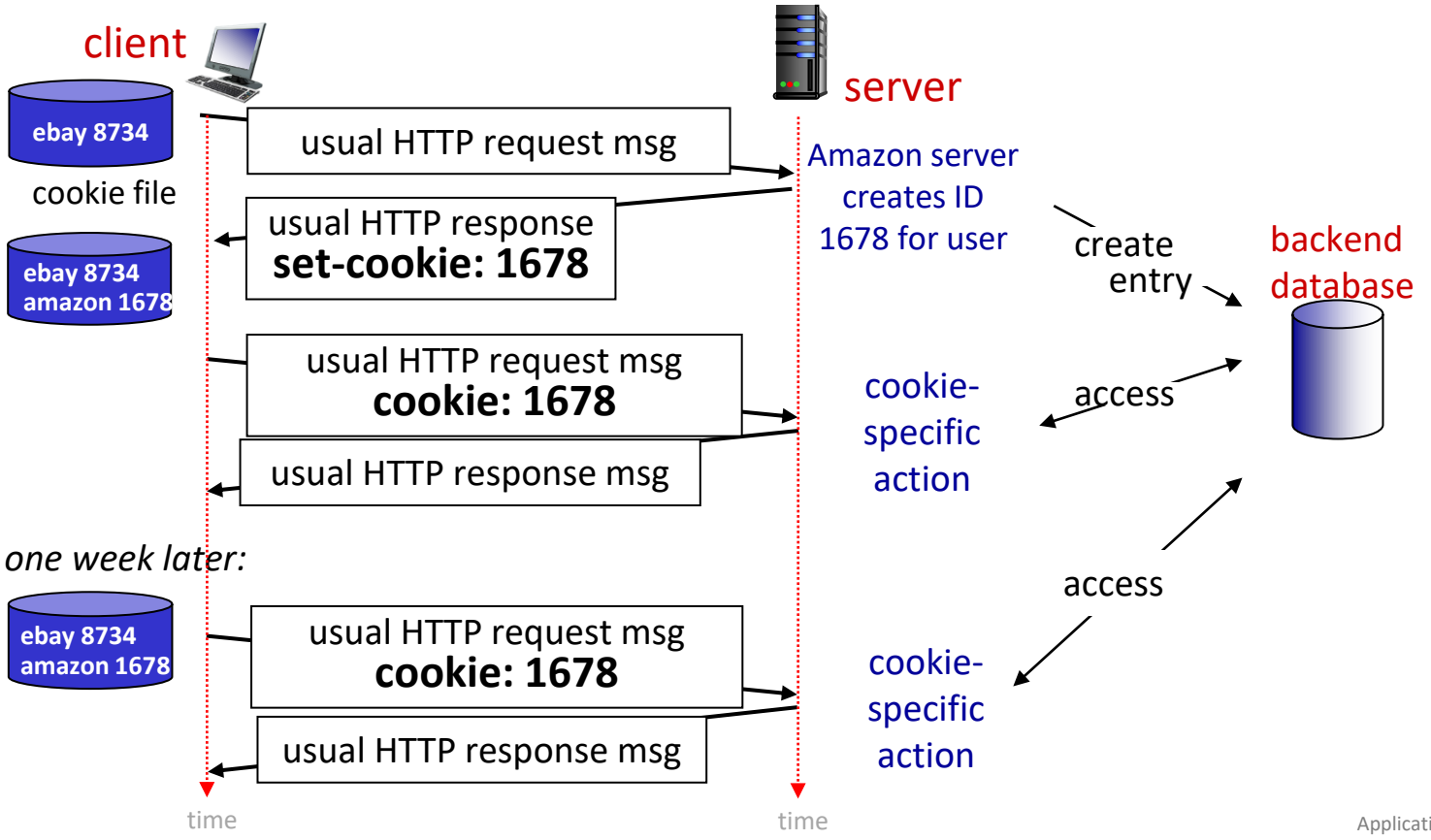
four components:

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

Maintaining user/server state: cookies



HTTP cookies: comments

What cookies can be used for:

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

Challenge: How to keep state?

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

aside

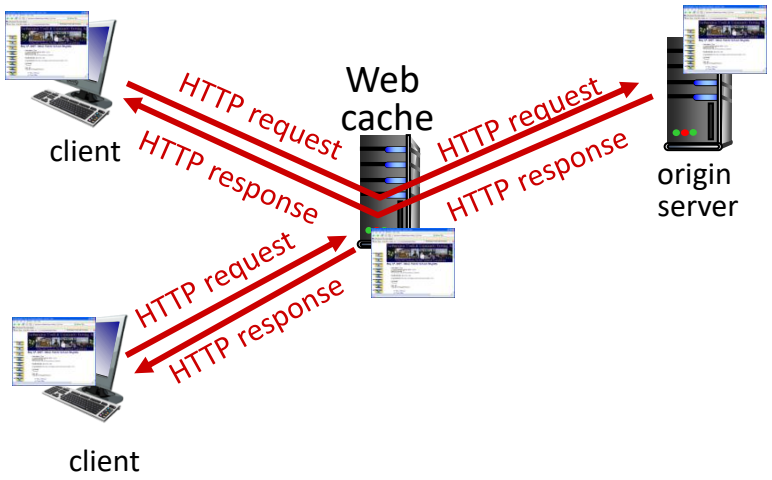
cookies and privacy:

- 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

Goal: satisfy client requests 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
- web cache is purchased and installed by an ISP, i.e., a university might install a web cache on its campus network and configure all of the campus browsers to point to the web cache



Web caches (aka proxy servers)

- Web cache acts as both client and server
  - server for original requesting client
  - client to origin server
- server tells cache about object's allowable caching in response header:

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

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

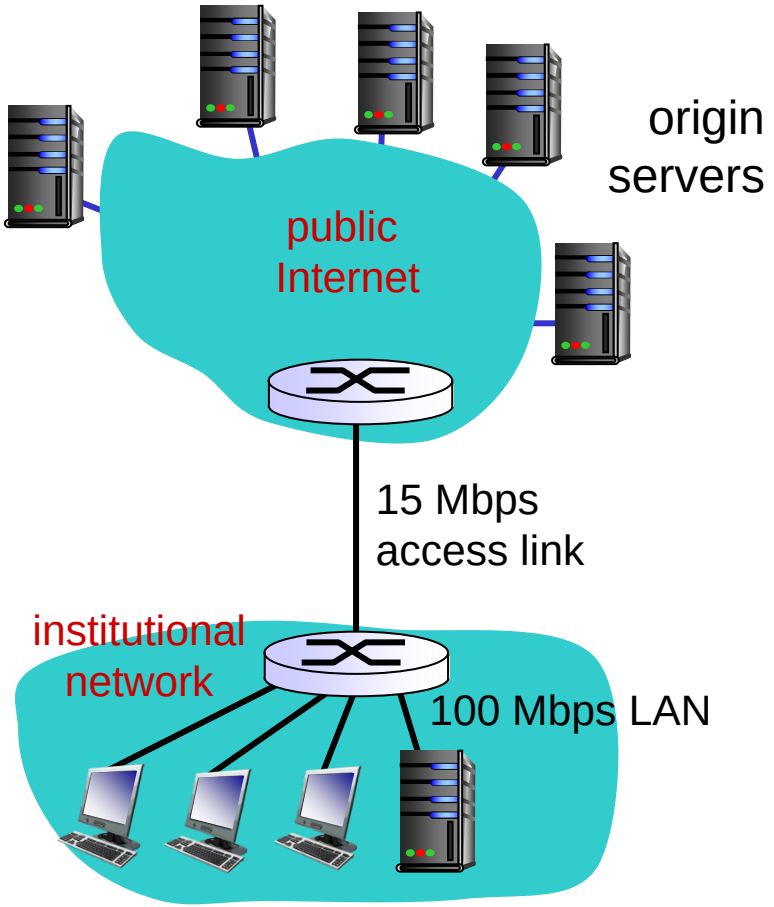
Why Web caching?

- reduce response time for client request
  - cache is closer to client
- reduce traffic on an institution's access link
- Internet is dense with caches
  - enables "poor" content providers to more effectively deliver too old content

Caching example

assumptions:

- avg object(file) size: 1Mbits
- access link rate: 15Mbps
- avg request rate from browsers to origin servers: 15 requests/sec
- HTTP request messages sizes are negligibly small
- avg Internet delay (the amount of time it takes when the router on the Internet side of the access link forwards an HTTP request until it receives the response): 2 secs



Caching example - 2

consequences:

- total response time (the time from the browser's request of an object until its receipt of the object) = LAN delay + access link delay (the delay between the two routers) + Internet delay

- traffic intensity on the LAN= (15 requests/sec) * (1Mbits/request) / (100 Mbps) = 0.15

- traffic intensity on the access link (from the Internet router to institution router)=(15 requests/sec)*(1Mbits/request) / (15Mbps) = 1

- traffic intensity of 0.15 on a LAN typically results in, at most, tens of milliseconds of delay; hence, we can neglect the LAN delay

- but in the case of the access link, as its traffic intensity approaches 1, the delay on the link becomes very large and grows without bound

Caching example - 3

performance:

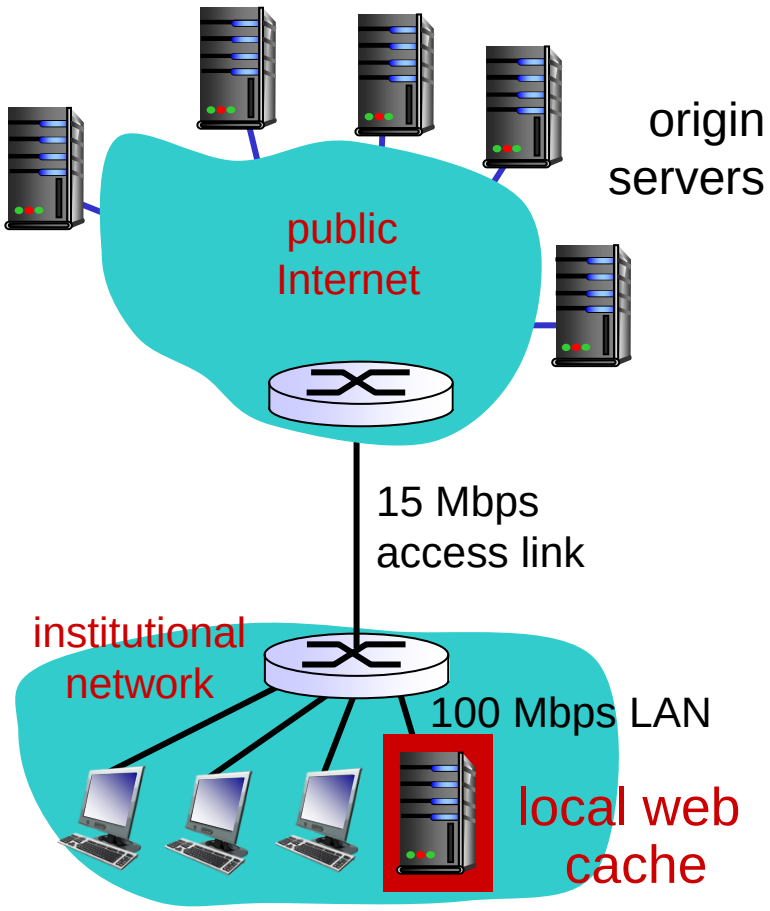
- LAN utilization: 15% *problem!*
- access link utilization = 99%
- total response time = negligible secs + unbounded minutes +2 secs
- thus, the total response time to satisfy requests is going to be on the order of minutes, if not more, which is unacceptable for the institution's users

- one possible solution is to increase the access rate from 15 Mbps to 100Mbps.
- this will lower the traffic intensity on the access link to 0.15, which translates to negligible delays between the two routers.
- in this case, the total response time will be 2 seconds, that is, the Internet delay.
- but this solution also means that the institution must upgrade its access link from 15 Mbps to 100 Mbps, a costly proposition.

Caching example: install local web cache

assumptions:

- Web cache provides a hit rate of 0.4 for institution's requests (40% requests satisfied at cache, 60% requests satisfied at origin)
- avg object size: 1Mbits
- access link rate: 15Mbps
- avg request rate from browsers to origin servers:15 requests/sec
- HTTP request messages sizes are negligibly small



Caching example: install local web cache - 2

consequences:

- 40 percent of the browsers' requests will be satisfied almost immediately, say, within 10 milliseconds (0.01 sec), by the cache.

- with only 60 percent of the requested objects passing through the access link, the traffic intensity on the access link is reduced from 1.0 to 0.6

- traffic intensity less than 0.8 corresponds to a negligibly small delay, say, 10's of milliseconds (0.01's of seconds), on a 15 Mbps link.

- average response time = 0.6 * (delay from origin servers) +
  0.4 * (delay when satisfied at cache)
  = 0.6 * (2 + 0.01) + 0.4 * (0.01 sec) ≈ 1.2 sec

- with this second solution, an institution doesn't require the to upgrade its link to the Internet

Conditional GET

Goal: don't send object if cache has up-to-date cached version

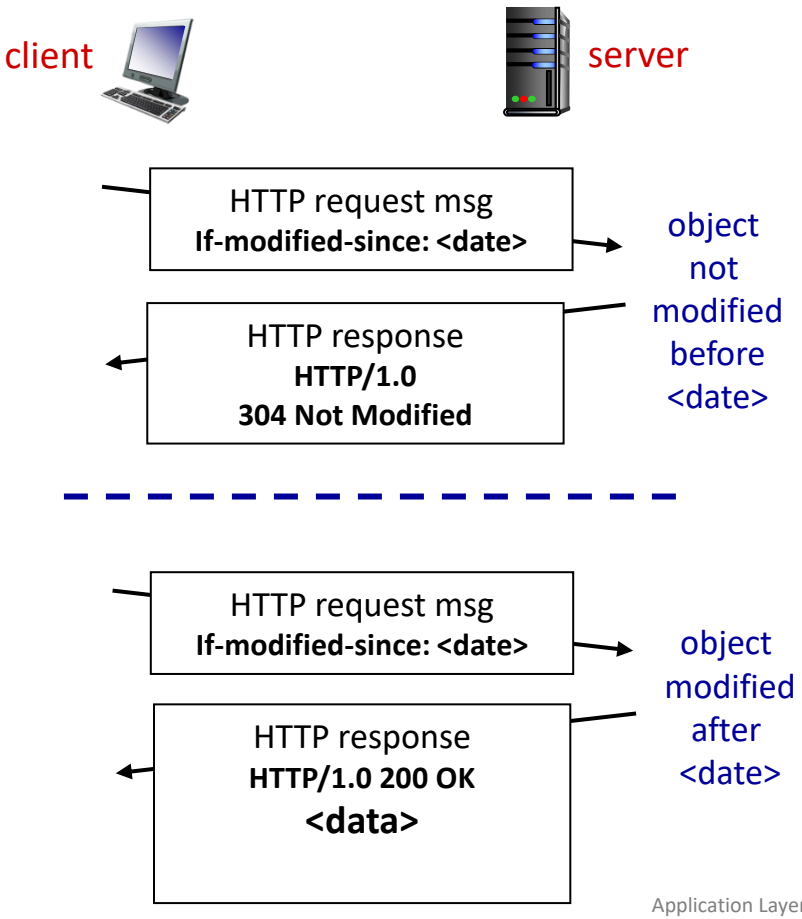
- no object transmission delay (or use of network resources)

- *client:* 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

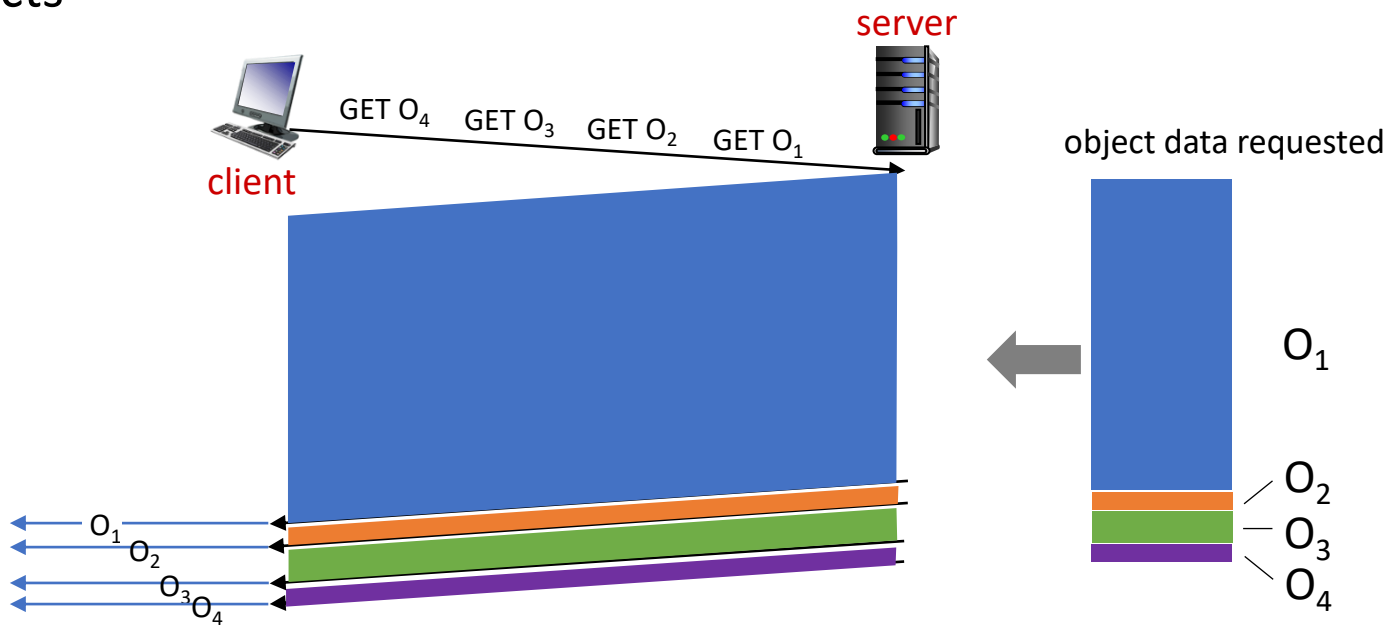
Key goal: decreased delay in multi-object HTTP requests

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

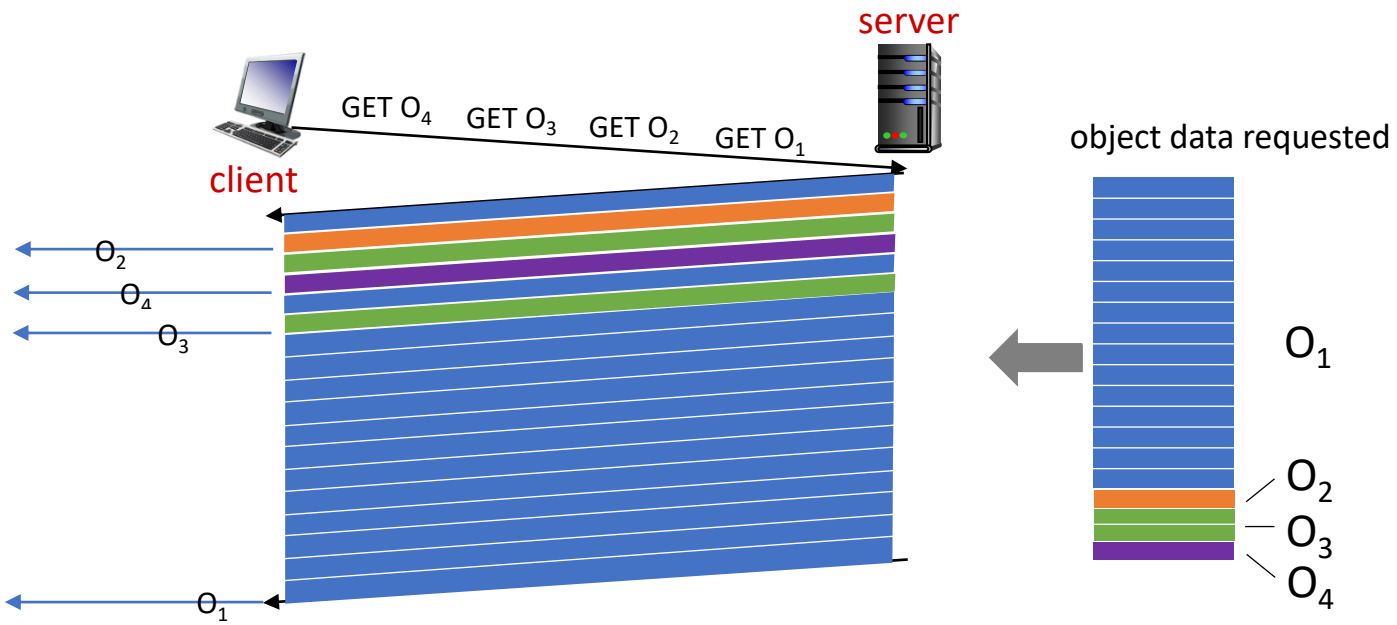
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



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 TCP connection
- HTTP/3: adds security, per object error- and congestion-control (more pipelining) over UDP
  - more on HTTP/3 in transport layer

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

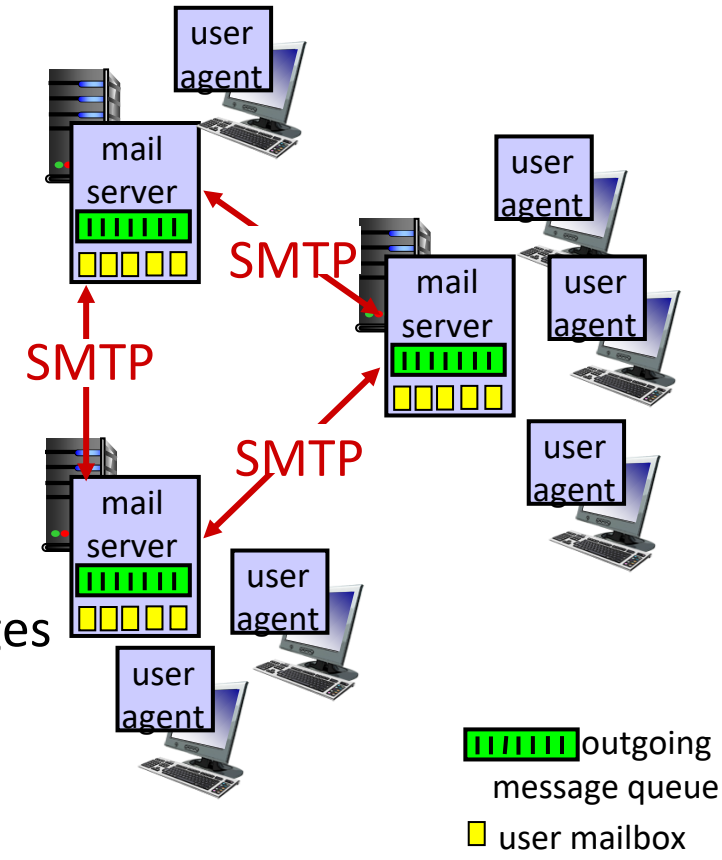
E-mail

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



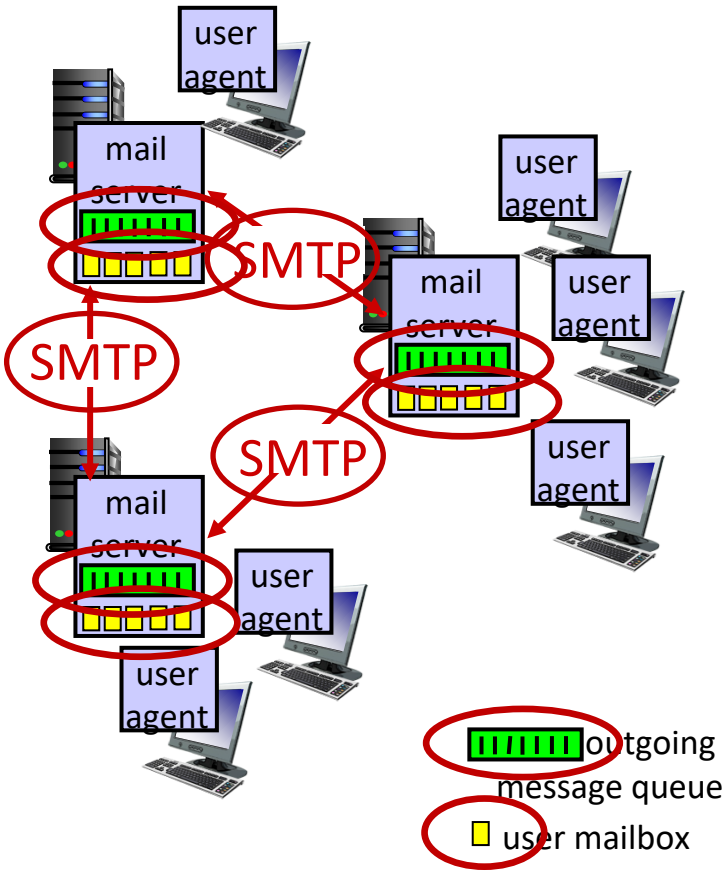
E-mail: mail servers

mail servers:

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

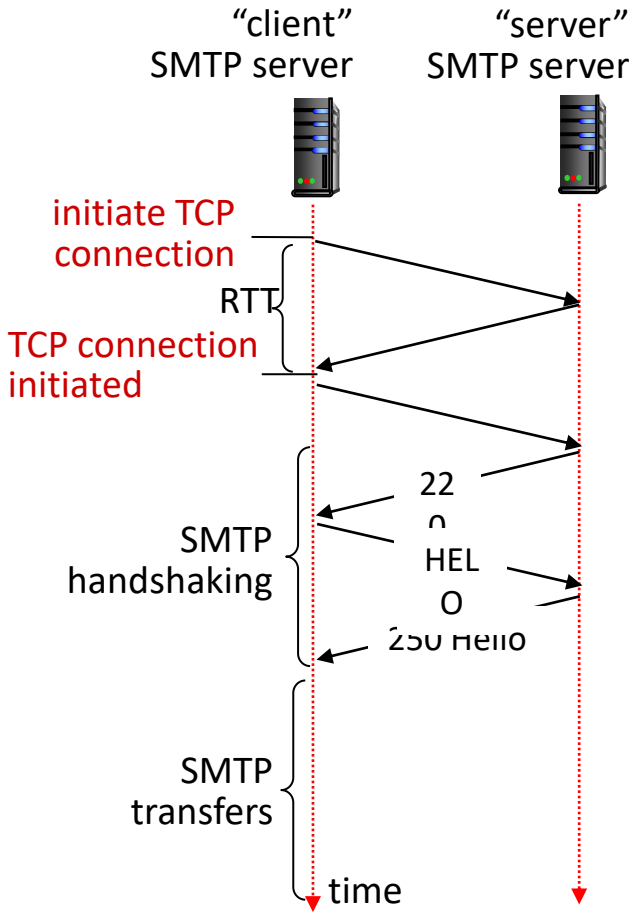
SMTP protocol between mail servers to send email messages

- client: sending mail server
- "server": receiving mail server



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



Scenario: Alice sends e-mail to Bob

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

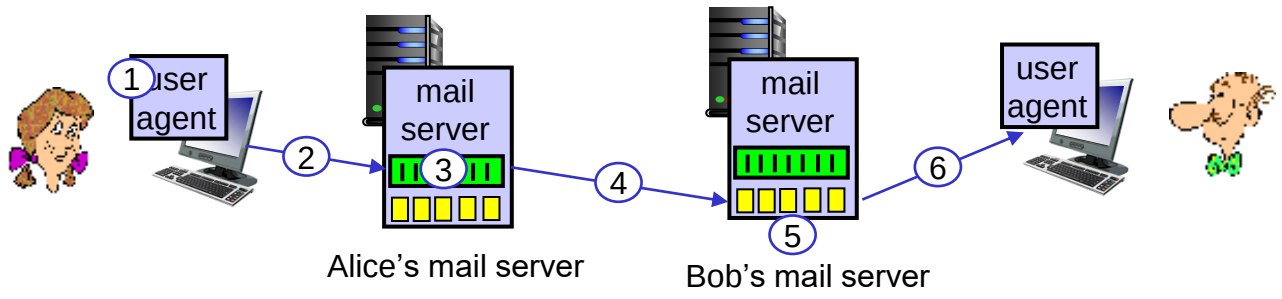
2) Alice's UA sends message to her mail server using SMTP; message placed in message queue

3) client side of SMTP at mail server 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



SMTP: final words

- SMTP uses persistent connections (if the sending mail server has several messages to send to the same receiving mail server, it can send all of the messages over the same TCP connection)

- SMTP requires message (header & body) to be in 7-bit ASCII (restriction), so it requires binary multimedia data (image, audio or video) to be encoded to ASCII before being sent over SMTP; and it requires the corresponding ASCII message to be decoded back to binary after SMTP transport

SMTP: final words-2

comparison with HTTP:

similarities:

- both protocols are used to transfer files from one host to another
- both persistent HTTP and SMTP use persistent connections
- both have ASCII command/response interaction, status codes

differences:

- SMTP is push protocol (TCP connection is initiated by the machine that wants to send the file), while HTTP is pull protocol (TCP connection is initiated by the machine that wants to receive the file)
- SMTP requires each message, including the body of each message, to be in 7-bit ASCII format, while HTTP data does not impose this restriction
- SMTP sends multiple objects in multipart message, while in HTTP each sent object encapsulated in its own response message

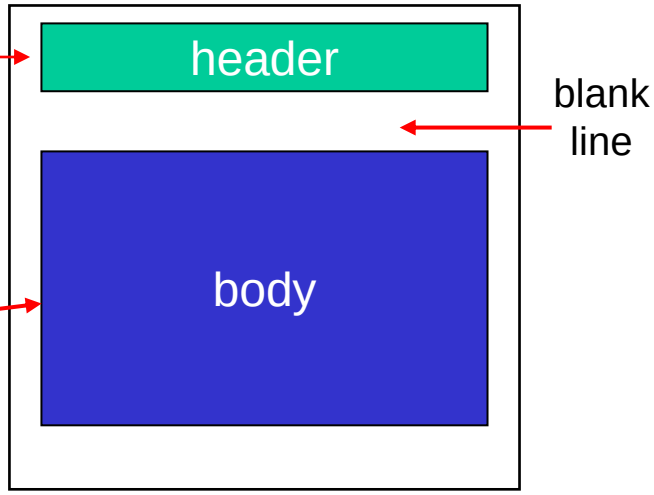
Mail message format

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

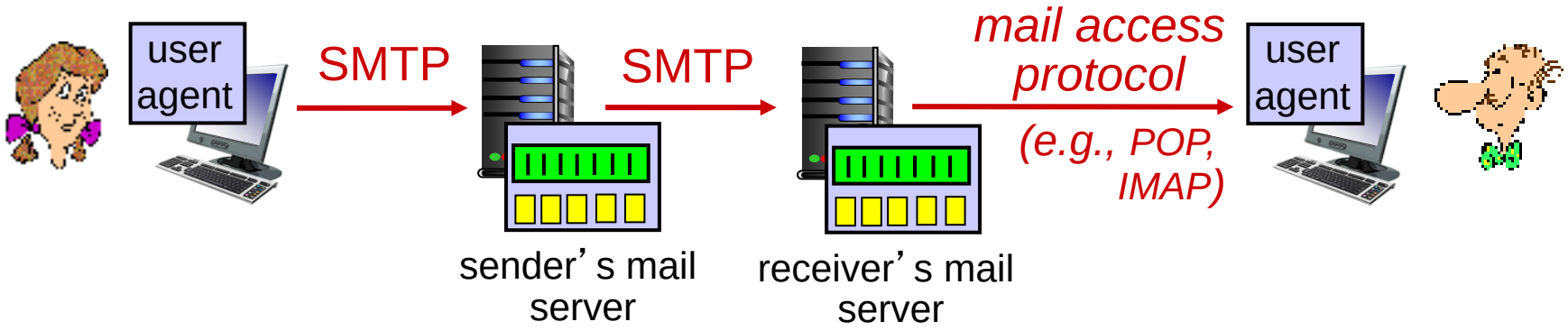


Mail access protocols

Why using a user agent (mail client) to send and receive emails?

- a mail server manages mailboxes and runs the client and server sides of SMTP.
- if the recipient's mail server was to reside on his/her local PC, then this PC would have to remain always on, and connected to the Internet, in order to receive new mails, which can arrive at any time.
- This is impractical for many Internet users.
- Instead, a typical user runs a user agent on the local PC but accesses his/her mailbox stored on an always-on shared mail server.
- This mail server is shared with other users and is typically maintained by the user's ISP (for example, university or company).

Mail access protocols-2



- SMTP: delivery/storage to receiver's mail server

- mail access protocol: retrieval from mail 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.

- Q: Why SMTP is not used by user agent to retrieve emails?
- A: because obtaining the messages is a pull operation whereas SMTP is a push protocol.

Web-based email

- more and more users today are sending and accessing their e-mail through their Web browsers (Hotmail, Gmail, Yahoo,…etc.)
- the user agent is an ordinary Web browser, and the user communicates with its remote mailbox via HTTP
- When a sender, such as Alice, wants to send an e-mail message, the e-mail message is sent from her browser to her mail server over HTTP rather than over SMTP
- Alice's mail server, however, still sends messages to, and receives messages from, other mail servers using SMTP
- When a recipient, such as Bob, wants to access a message in his mailbox, the e-mail message is sent from Bob's mail server to Bob's browser using the HTTP protocol rather than the POP3 or IMAP.

Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- **The Domain Name System DNS**
- P2P applications
- video streaming and content distribution networks
- 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 domain *name servers*
- *application-layer protocol:* hosts, domain name servers communicate over UDP using port 53 to *resolve* names (IP address/name translation) by *query* and *reply* messages

- note: core Internet function, implemented as application-layer protocol because of more complexity is at network's "edge" than at networks "core"

DNS: domain name system-2

- in order for the user's host to be able to send an HTTP request message to the Web server www.someschool.edu, the user's host must first obtain the IP address of www.someschool.edu.

- this is done as follows:

1. the same user machine runs the client side of the DNS application.
2. the browser extracts the hostname, www.someschool.edu, from the URL and passes the hostname to the client side of the DNS application.
3. the DNS client sends a query containing the hostname to a DNS server.
4. the DNS client eventually receives a reply, which includes the IP address for the hostname.
5. once the browser receives the IP address from DNS, it can initiate a TCP connection to the HTTP server process located at port 80 at that IP address.

DNS: services, structure

DNS services

- hostname to IP address translation
- host aliasing
  - map alias name of a web server (e.g., www.enterprise.com) to canonical name (e.g., relay1.west-coast. enterprise.com)+IP address.
- mail server aliasing
  - map alias name of a mail server (e.g., hotmail.com) to its canonical name (e.g., relay1.west-coast.hotmail.com)+IP address
- load distribution
  - replicated Web servers: many IP addresses correspond to one canonical host name
  - when clients make a DNS query for a name mapped to a set of addresses, the server responds with the entire set of IP addresses, but rotates the ordering of the addresses within each reply to distribute the traffic among the replicated servers.
  - DNS rotation is also used for mail servers or content distribution companies that have same alias name

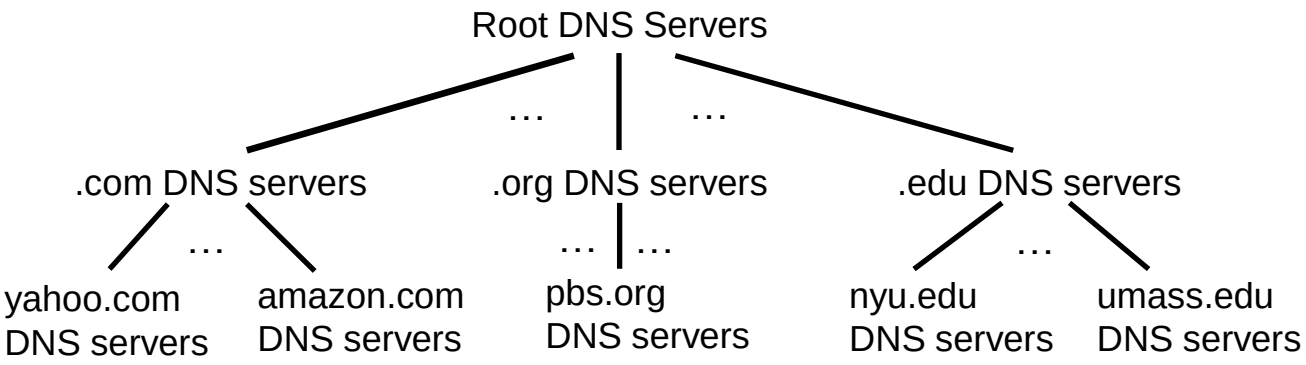
DNS: services, structure-2

Q: why not centralize DNS?

A: doesn't scale!

- single point of failure
  - if the DNS server crashes, so does the entire Internet!
- traffic volume:
  - A single DNS server would have to handle all DNS queries (for all the HTTP requests and e-mail messages generated from hundreds of millions of hosts).
- distant centralized database
  - A single DNS server cannot be "close to" all the querying clients which can lead to significant delays
- maintenance
  - the single DNS server would have to keep records for all Internet hosts, but it would have to be updated frequently to account for every new host

DNS: a distributed, hierarchical database

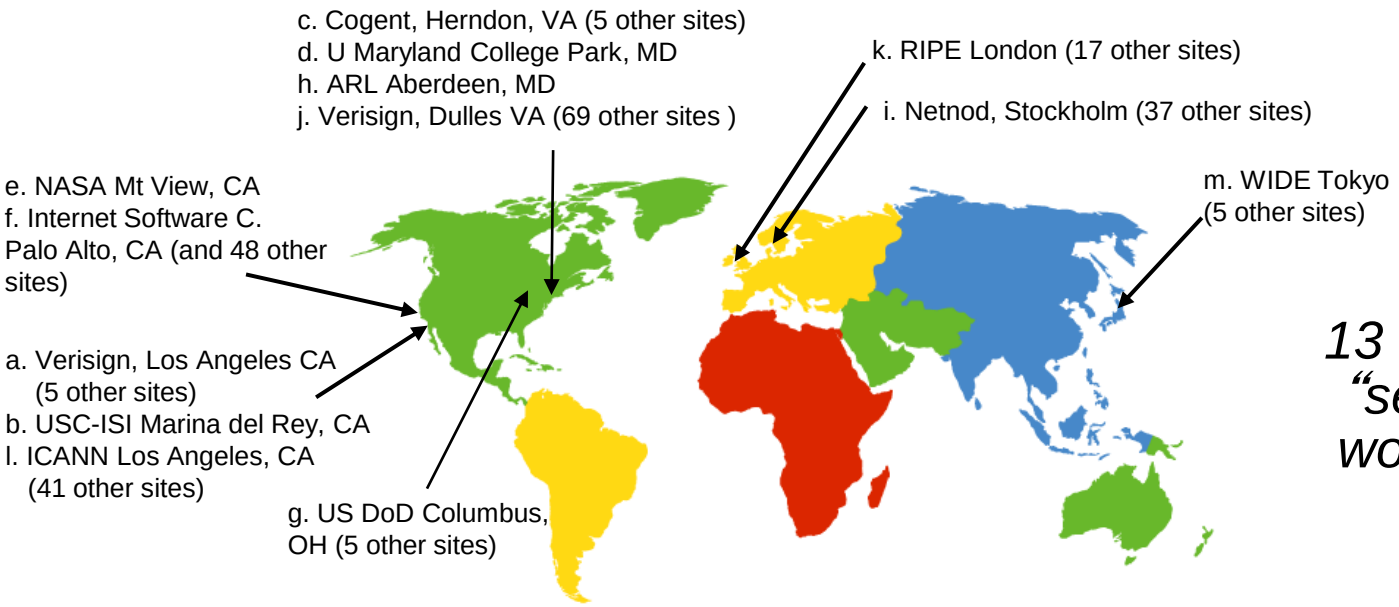


Client wants IP address for www.amazon.com; 1st approximation:

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

DNS: root name servers

- > 1000 root DNS servers scattered all over the world.
- these servers are copies of 13 different root DNS servers, for both security and reliability purposes,
- are managed by 12 different organizations and most of them are located in north America.
- provide the IP addresses of the TLD DNS servers.

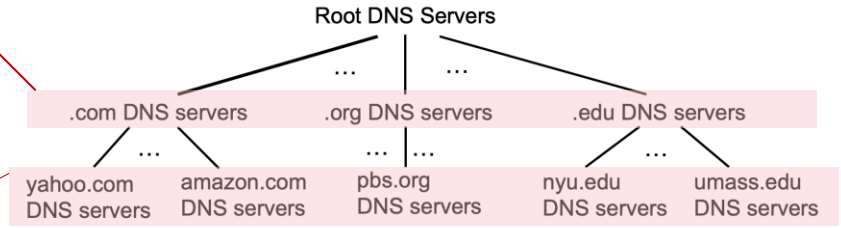


13 root name "servers" worldwide

Top-Level Domain, and authoritative servers

Top-Level Domain (TLD) servers:

- responsible for .com, .org, .net, .edu, .aero, .jobs, .museums, and all top-level country domains, e.g.: .cn, .uk, .fr, .ca, .jp
- provide the IP addresses for authoritative DNS servers
- company Verisign Global Registry Services: maintaining registry for .com, .net TLD
- company Educause: maintain .edu TLD



authoritative DNS servers:

- organization's (universities & large companies)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 servers

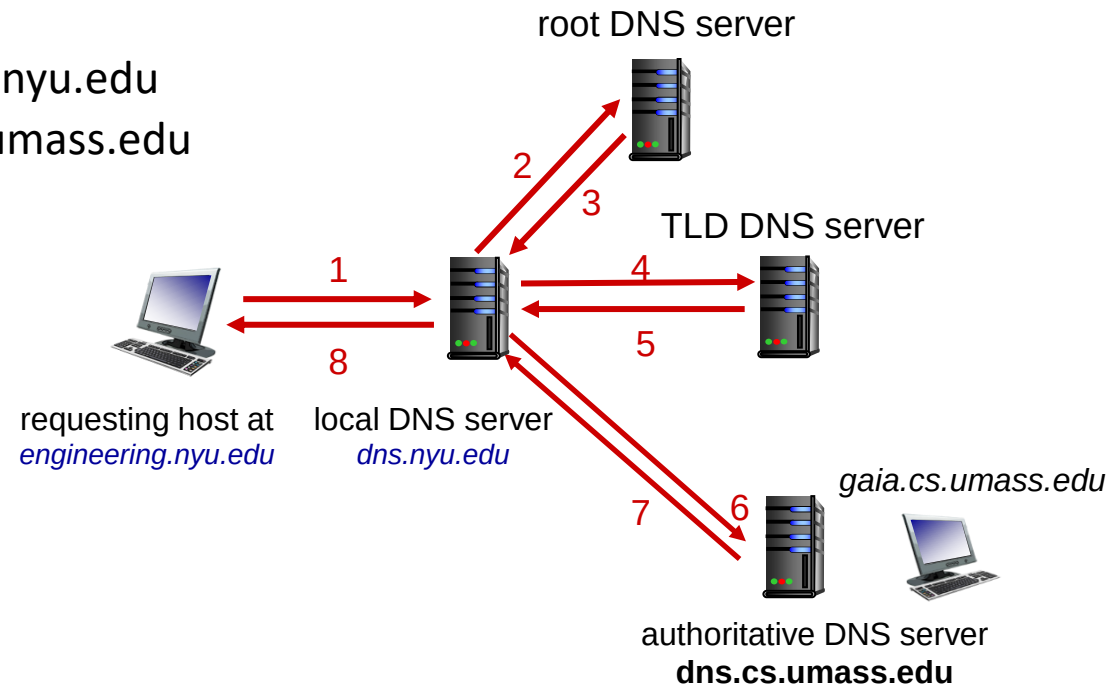
- when host makes DNS query, it is sent to its *local* DNS server
  - Local DNS server returns reply, answering:
    - from its local cache of recent name-to-address translation pairs (possibly out of date!)
    - forwarding request into DNS hierarchy for resolution
  - each ISP has local DNS name server; to find yours:
    - MacOS: `% scutil --dns`
    - Windows: `>ipconfig /all`
- local DNS server doesn't strictly belong to hierarchy

DNS name resolution: iterated query

Example: host at engineering.nyu.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"



8 DNS messages were sent:

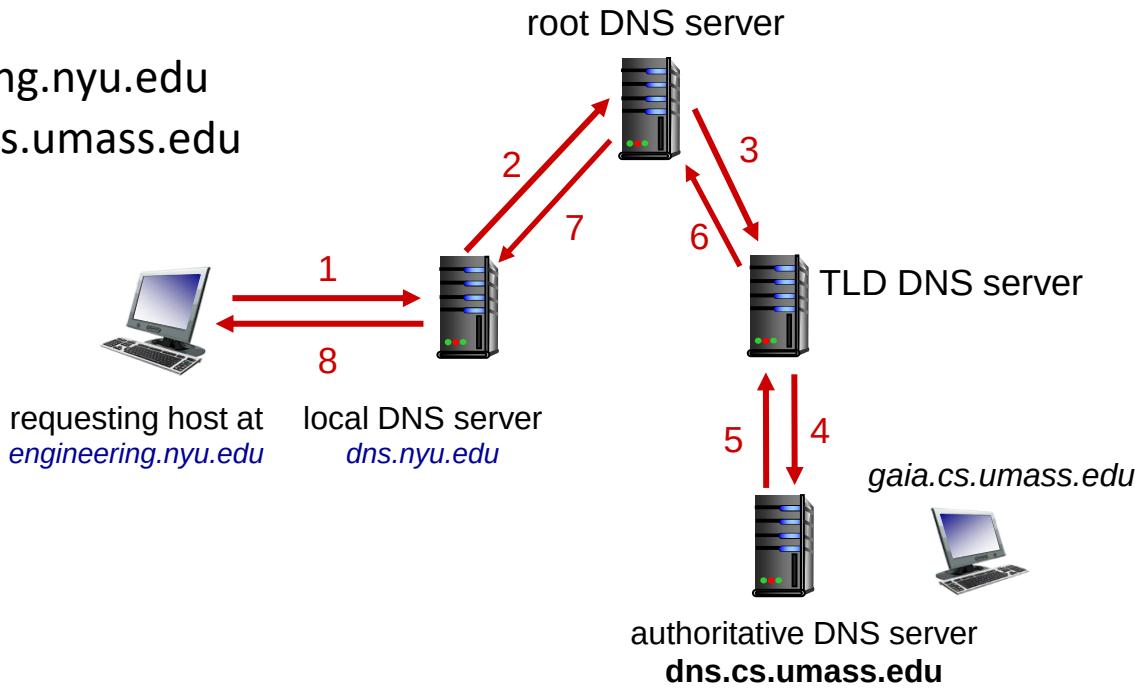
4 query messages + 4 reply messages

DNS name resolution: recursive query

Example: host at engineering.nyu.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?



In practice, the query from the requesting host to the local DNS server is recursive, and the remaining queries are iterative.

DNS: caching, updating records

- DNS caching improves the delay performance and reduce the number of DNS messages in the Internet

- once (any) name server learns a mapping, it *caches* this mapping
  - cache entries timeout (disappear) after some time-to-live (TTL)
  - TTL often set to 2 days
  - TLD servers typically cached in local DNS 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 (RRs) of mappings

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

type=A

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

type=CNAME

- **`name`** is alias name for some "canonical" (the real) name
- **`value`** is canonical name
- e.g., (foo.com, relay1.bar.foo.com, CNAME)

type=NS

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

type=MX

- **`value`** is the canonical name of mail server associated with alias **`name`**
- e.g., (foo.com, mail.bar.foo.com, MX)

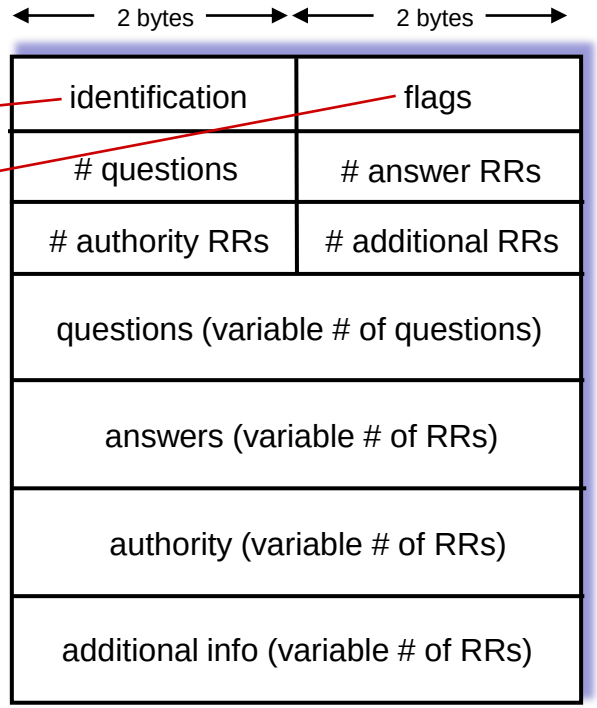
- with types CNAME and MX, a company can have the same aliased name for its mail server and for one of its Web server

DNS protocol messages

DNS *query* and *reply* messages, both have same *format:*

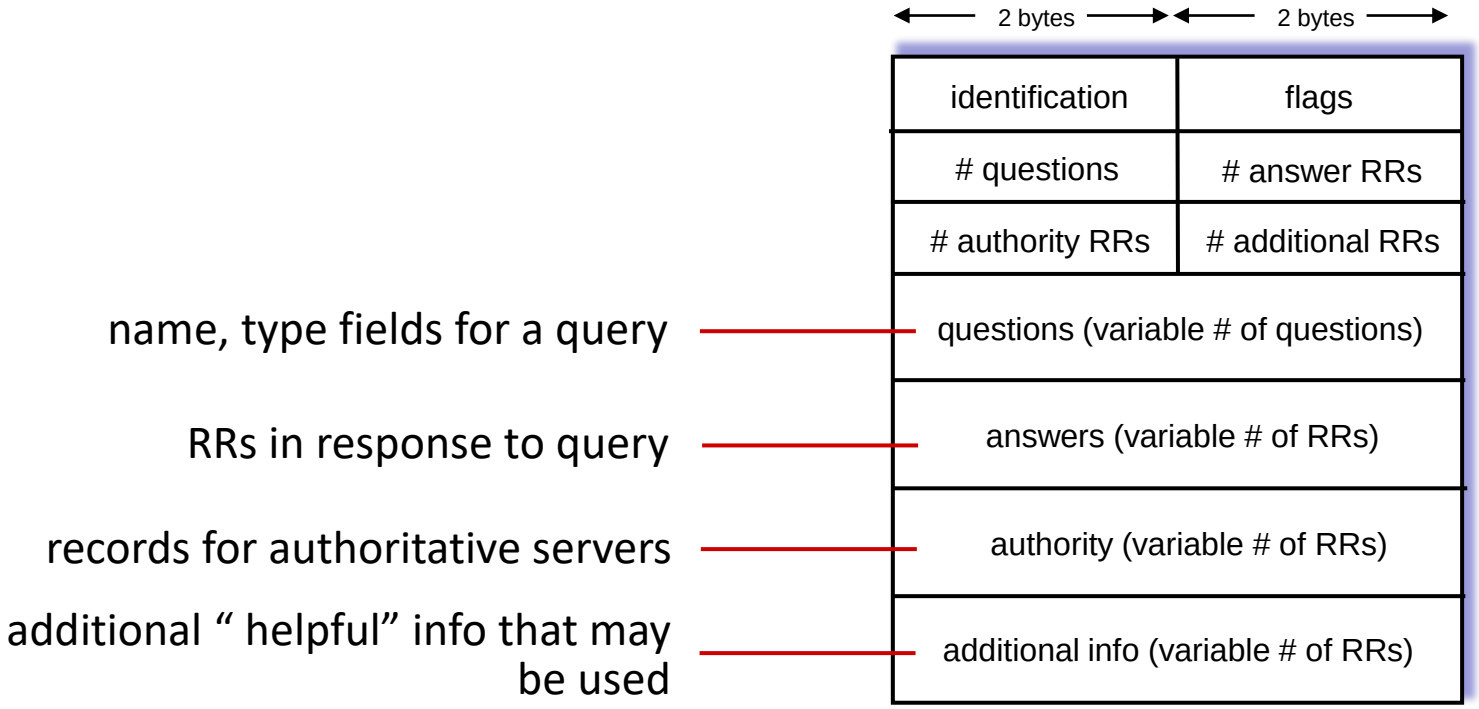
message header:

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



DNS protocol messages

DNS *query* and *reply* messages, both have same *format:*



Getting your info into the DNS

example: new startup “Network Utopia”

- register name networkuptopia.com at *DNS registrar* (commercial entity verifies the uniqueness of the domain name, enters the domain name into the DNS database, and collects a small fees from you for its services)
  - 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

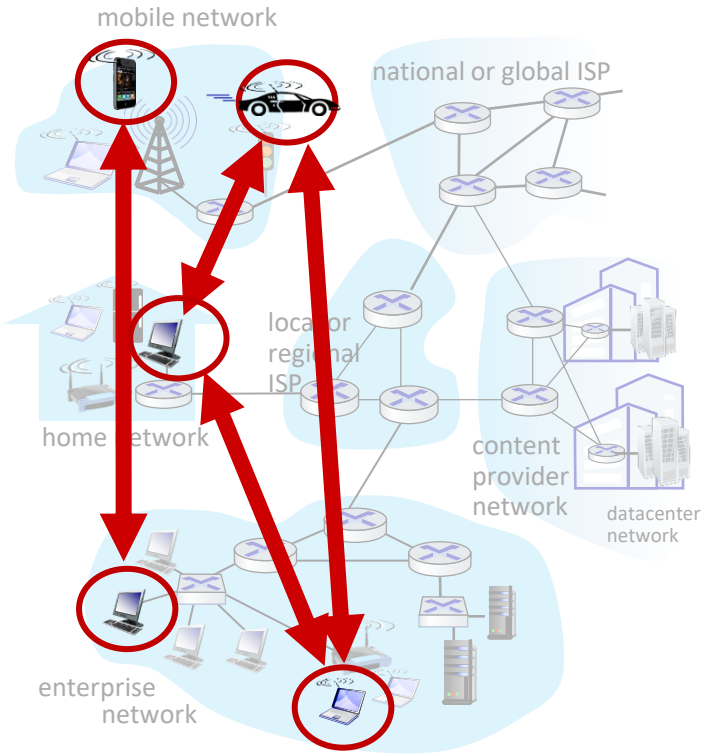
Application Layer: Overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS

- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

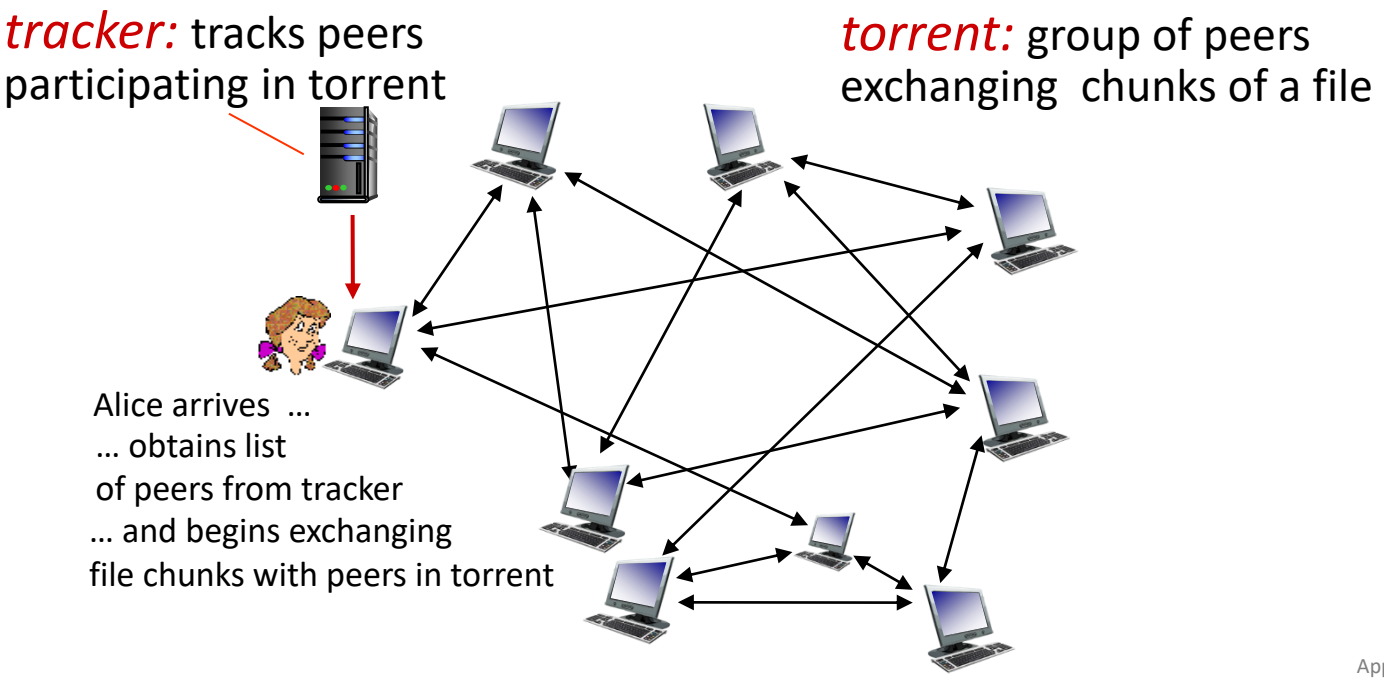
Peer-to-peer (P2P) architecture

- *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, and new service demands
- peers are intermittently connected and change IP addresses
  - complex management
- examples: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)

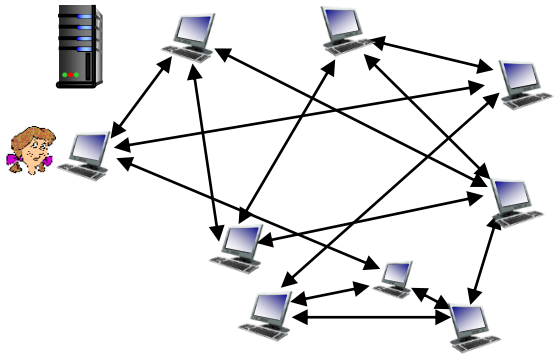


P2P file distribution: BitTorrent

- 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

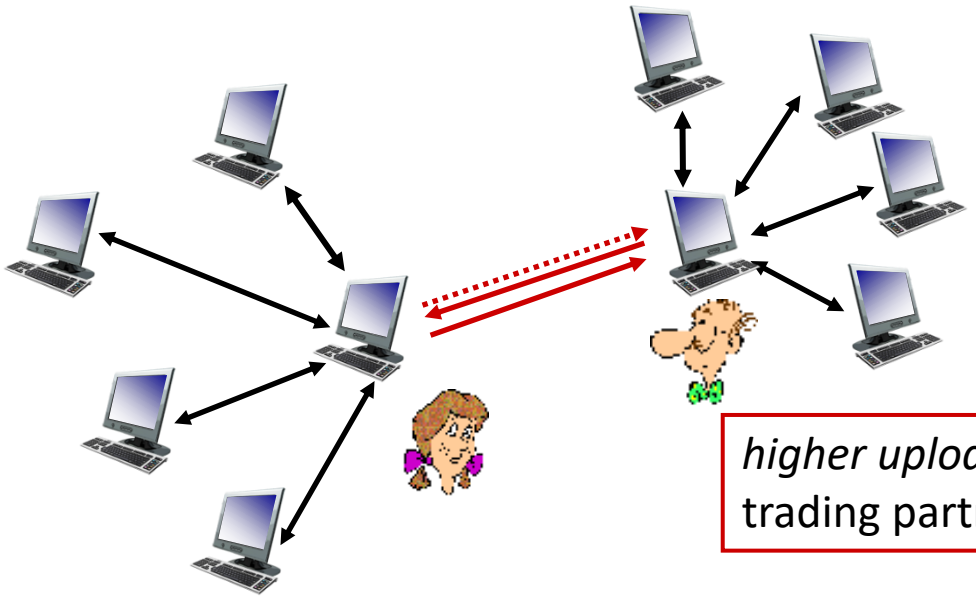
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
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

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Video Streaming and Content Distribution Networks (CDNs)

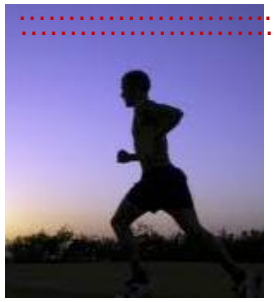
- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime: 80% of residential ISP traffic (2020)
  - ~1B YouTube users, ~75M Netflix users

- *challenge:* scale - how to reach ~1B users?
  - *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., 24-30 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy *within* and *between* images to decrease # bits used to encode image
  - spatial (within image)
  - temporal (from one image to next)

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)



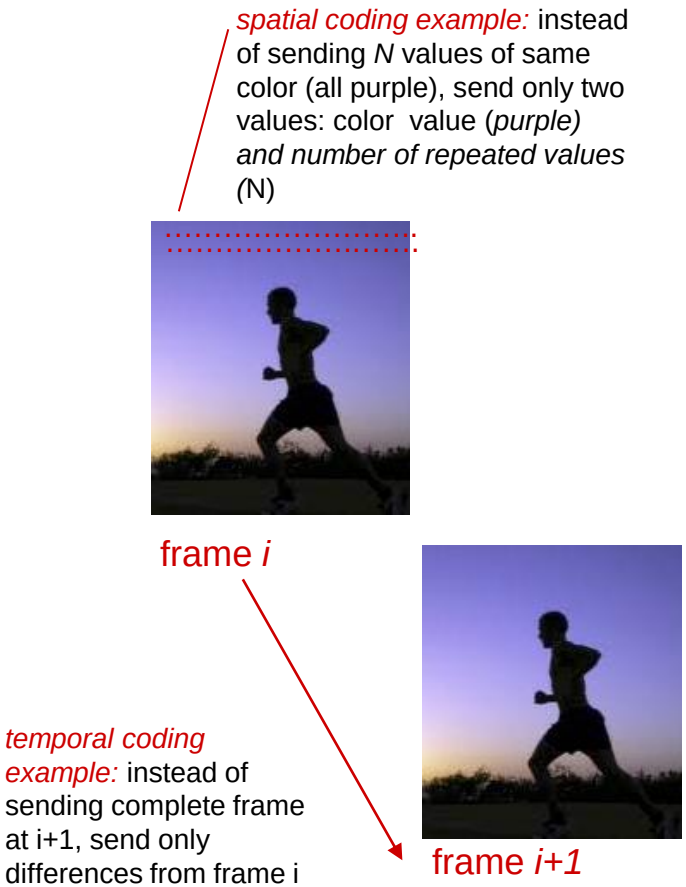
frame *i*

frame *i+1*

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

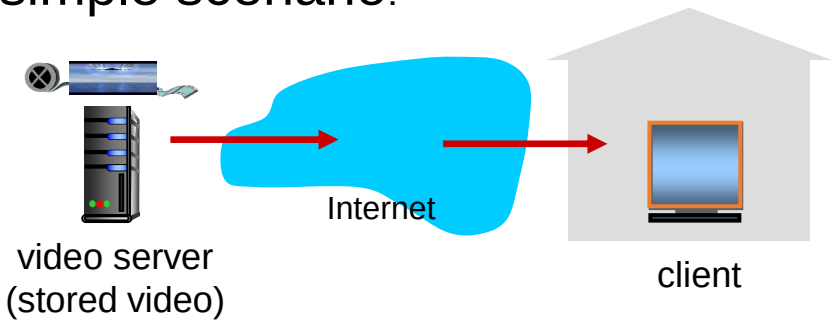
Multimedia: video

- 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, 64Kbps – 12 Mbps)



Streaming stored video

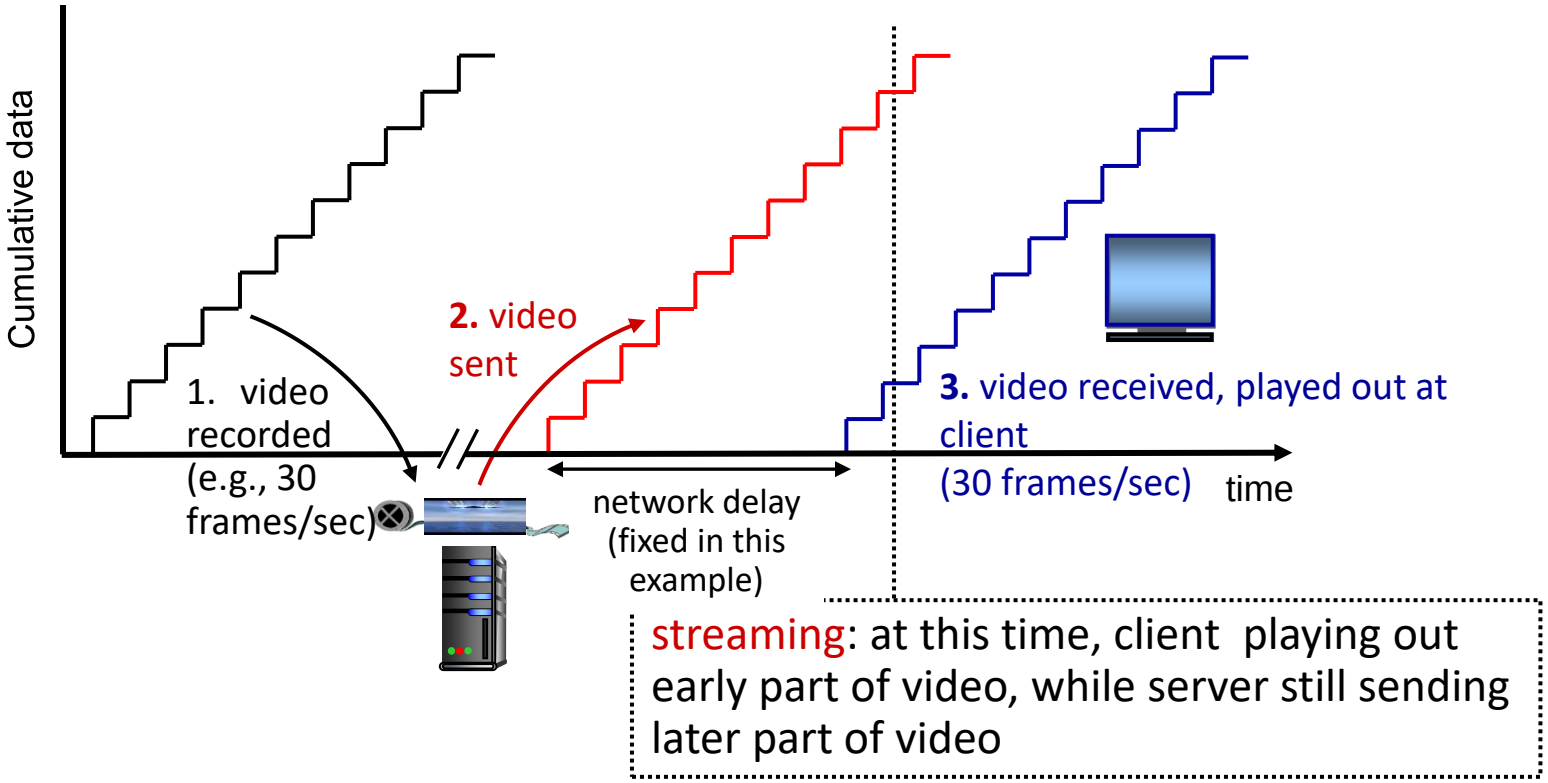
simple scenario:



Main challenges:

- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, access network, network core, video server)
- packet loss, delay due to congestion will delay playout, or result in poor video quality

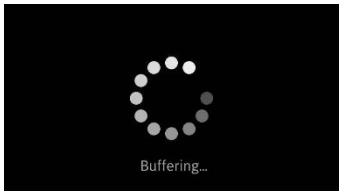
Streaming stored video



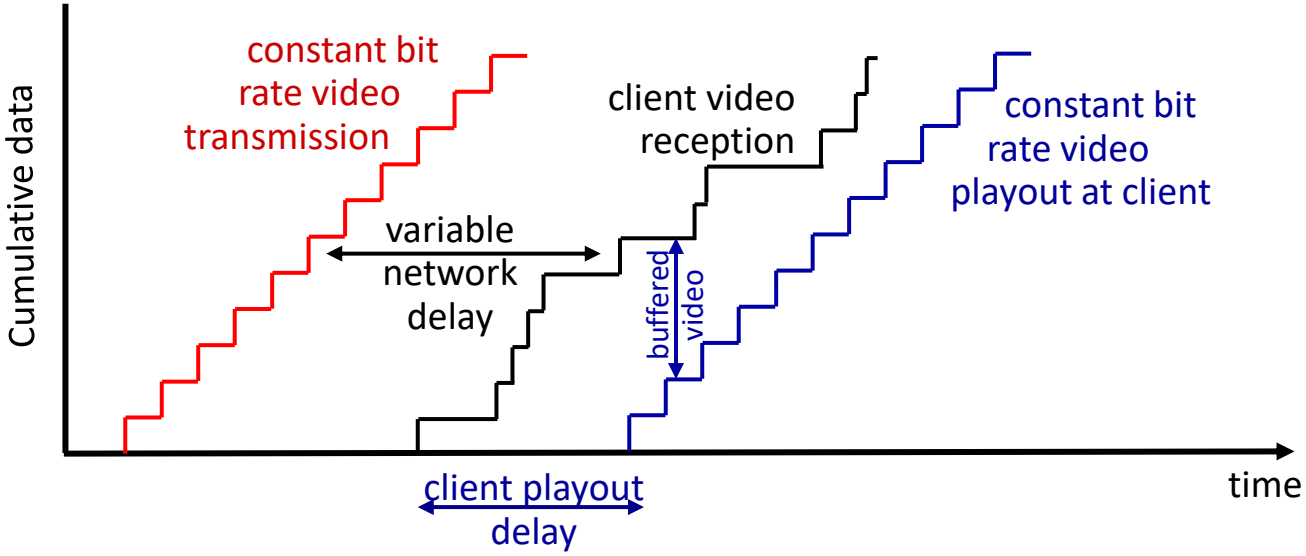
streaming: at this time, client playing out early part of video, while server still sending later part of video

Streaming stored video: challenges

- continuous playout constraint: during client video playout, playout timing must match original timing
  - … but network delays are variable (jitter), so will need client-side buffer to match continuous playout constraint
- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted



Streaming stored video: playout buffering



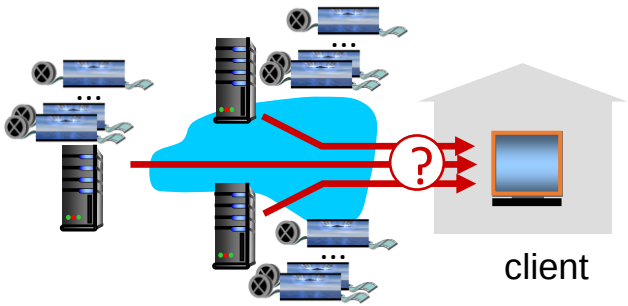
- *client-side buffering and playout delay:* compensate for network-added delay, delay jitter

Streaming multimedia: DASH

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

server:

- divides video file into multiple chunks
- each chunk encoded at multiple different rates
- different rate encodings stored in different files
- files replicated in various CDN nodes
- *manifest file:* provides URLs for different chunks

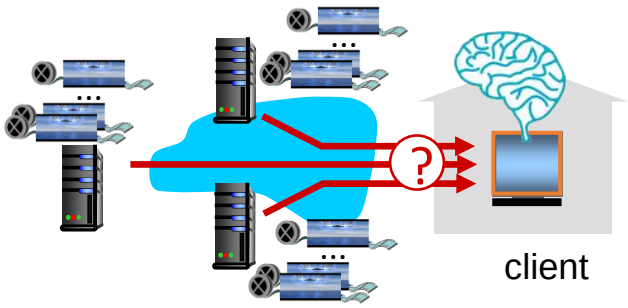


client:

- periodically estimates 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), and from different servers

Streaming multimedia: 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)



Streaming video = encoding + DASH + playout buffering

Content distribution networks (CDNs)

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

- *option 1:* single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long (and possibly congested) path to distant clients

….quite simply: this solution *doesn't scale*

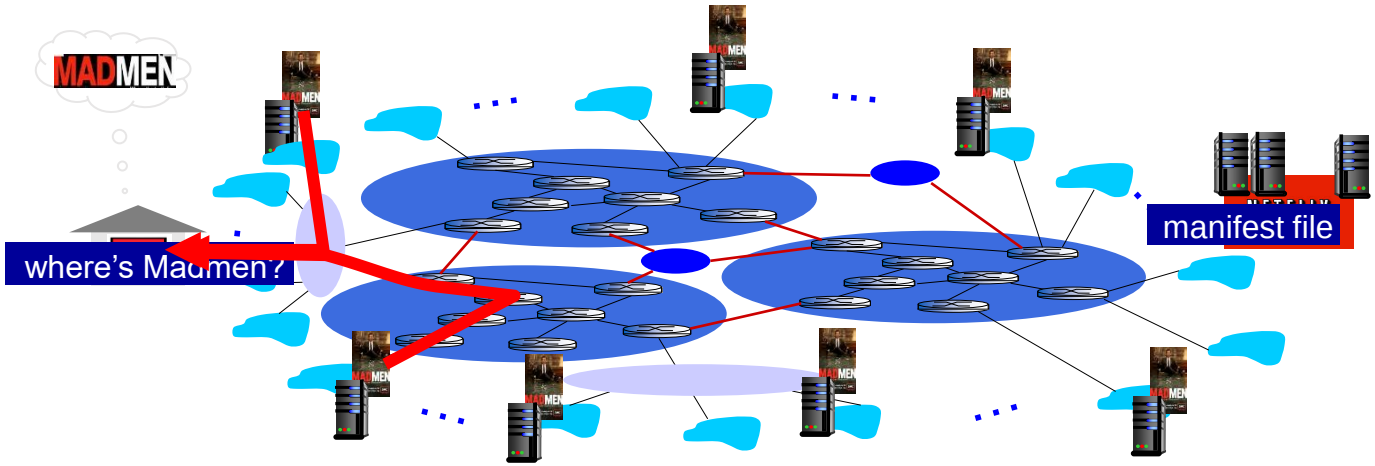
Content distribution networks (CDNs)

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

- *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
    - task of maintaining and managing becomes challenging
    - Akamai: 240,000 servers deployed in > 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in Internet Exchange Points (IXPs) near access nets
    - possibly at the expense of higher delay and lower throughput to end users
    - used by Limelight

Content distribution networks (CDNs)

- CDN: stores copies of content at CDN nodes
- subscriber requests content, service provider returns manifest
  - using manifest, client retrieves content at highest supportable rate
  - may choose different rate or copy if network path congested



Content distribution networks (CDNs)

Internet host-host communication as a service

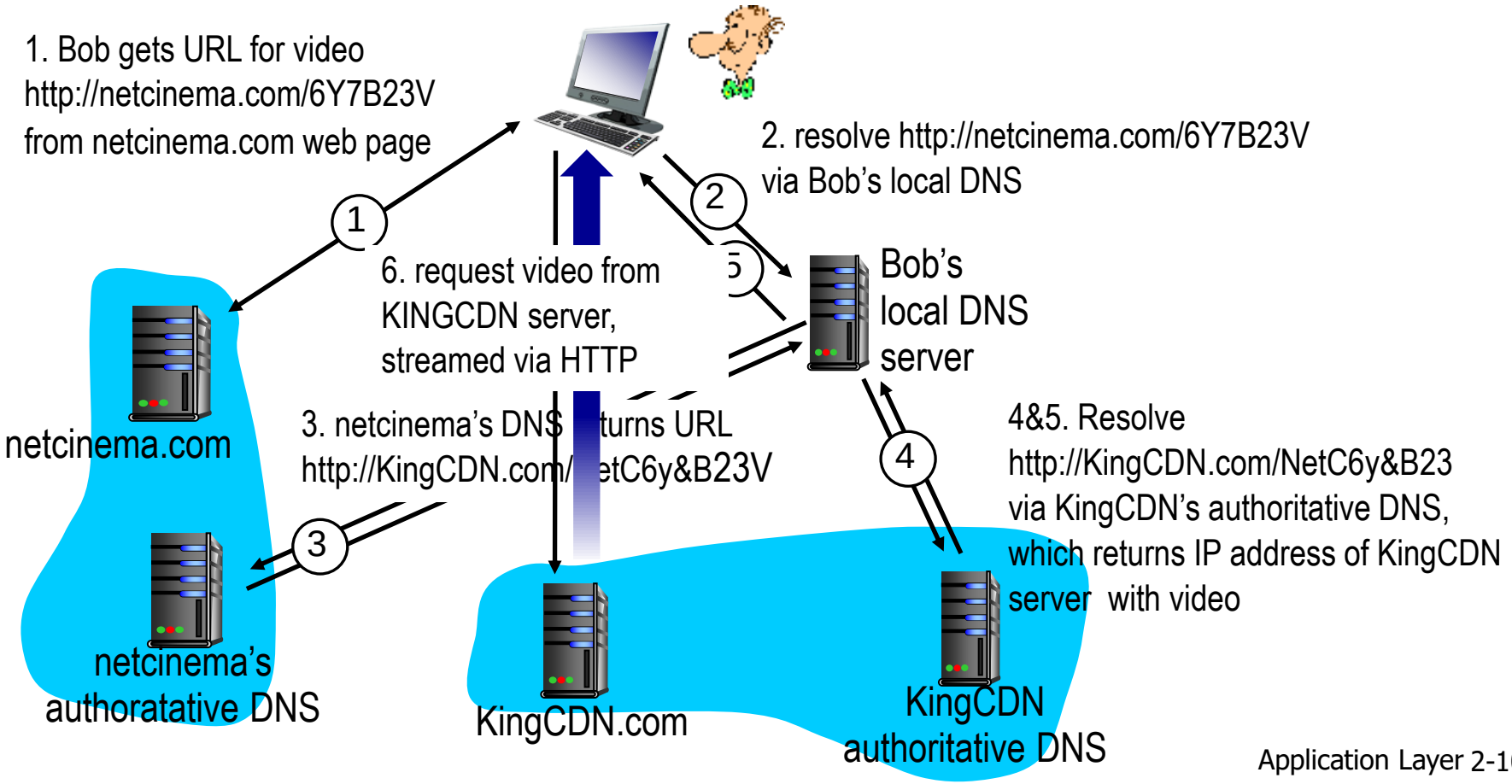
OTT challenges: coping with a congested Internet from the "edge"

- what content to place in which CDN node?
- from which CDN node to retrieve content? At which rate?

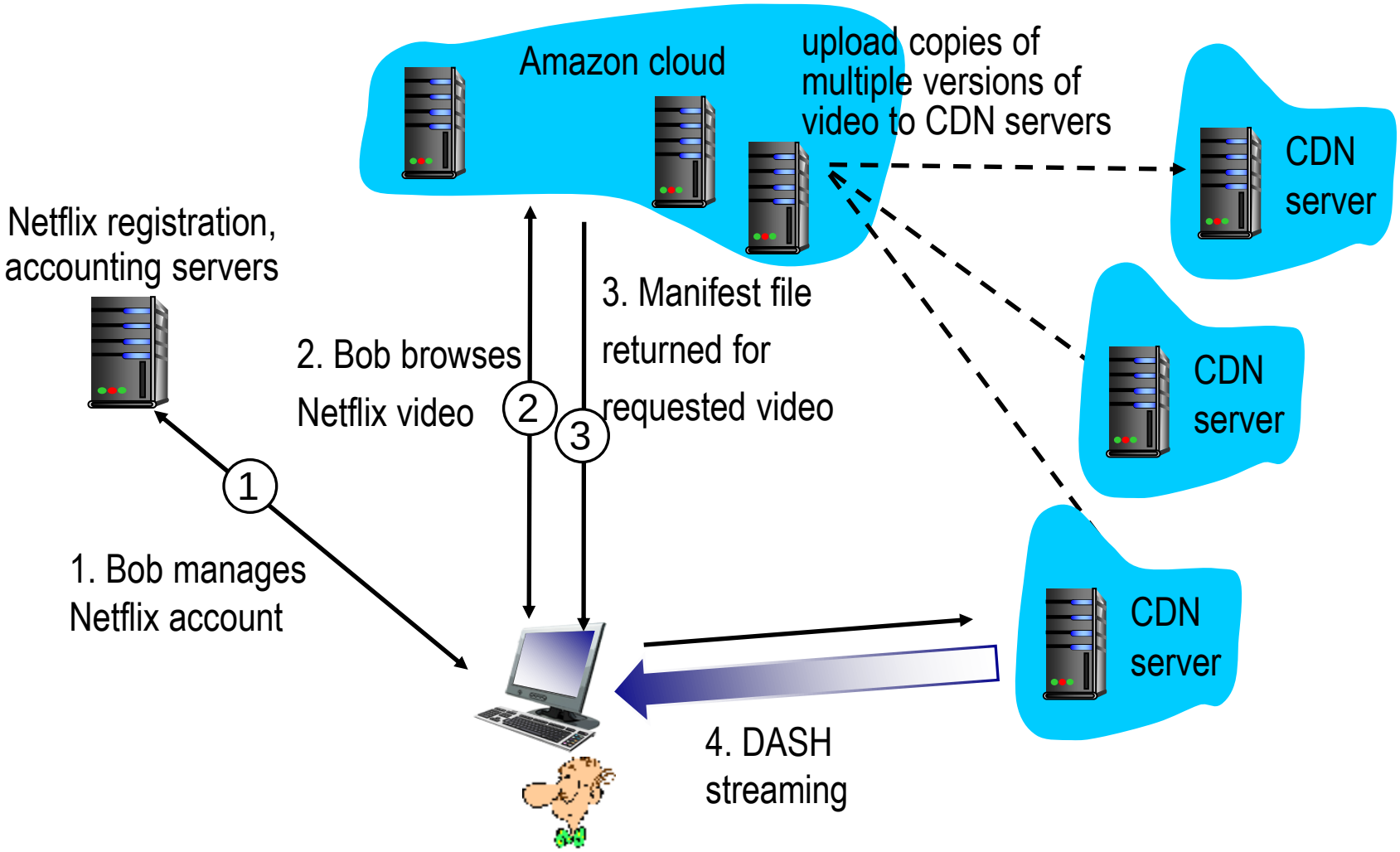
CDN content access: a closer look

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

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



Case study: Netflix



The End