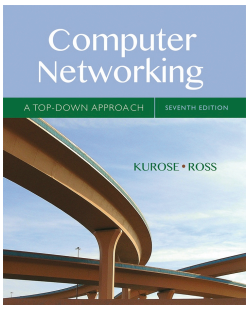
Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 Video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP



Computer Networking: A Top Down Approach 7th edition
Jim Kurose, Keith Ross
Addison-Wesley
April 2016

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 Video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP

Chapter 2: Application Layer

Our goals:

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

Some network applications

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (e.g., YouTube, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing (e.g., Facetime, Google Hangouts)
- social networking (e.g., Facebook, Instagram, Twitter, WeChat)
- location-based mobile apps (e.g., Waze)
- …

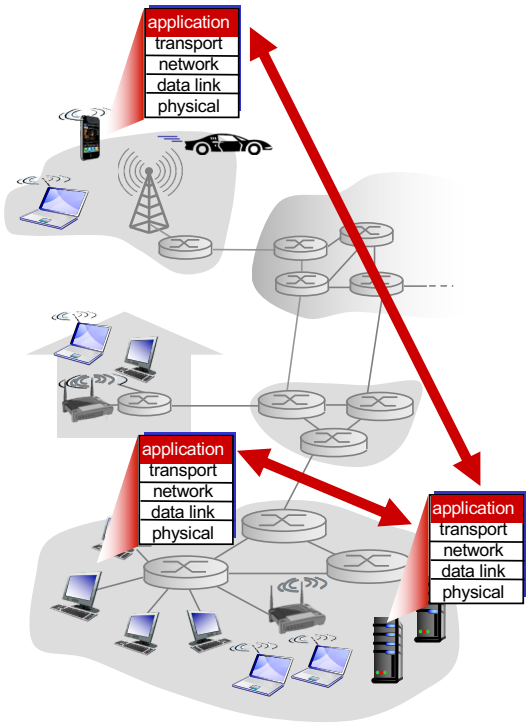
Creating a network app

Write programs that

- run on *different* end systems and communicate over a 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's applications
- applications on end systems allow for rapid application development, propagation



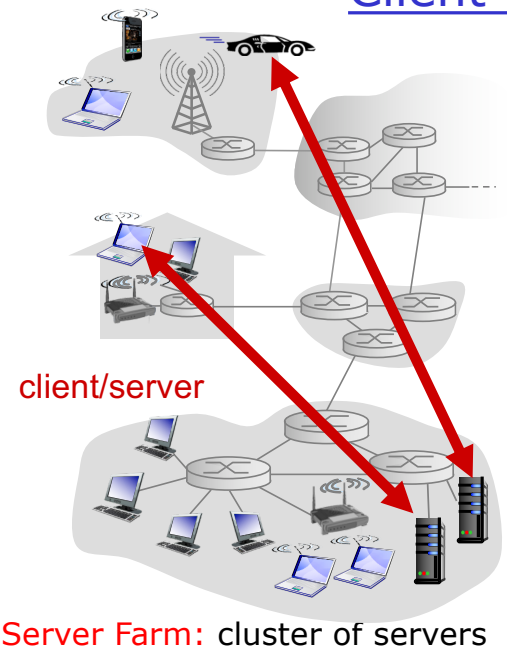
Application architectures

- Dictate how the application is organized over various end systems
- Designed by the application developer

- Possible structure of applications:

1. Client-server architecture
   - Including data centers / cloud computing
2. Peer-to-peer (P2P) architecture

Client-server architecture



Server Farm: cluster of servers

Search engine (Google, Bing), e-commerce (Amazon, eBay), webmail (Gmail), social networking (Facebook, Twitter), video (YouTube)

Server:

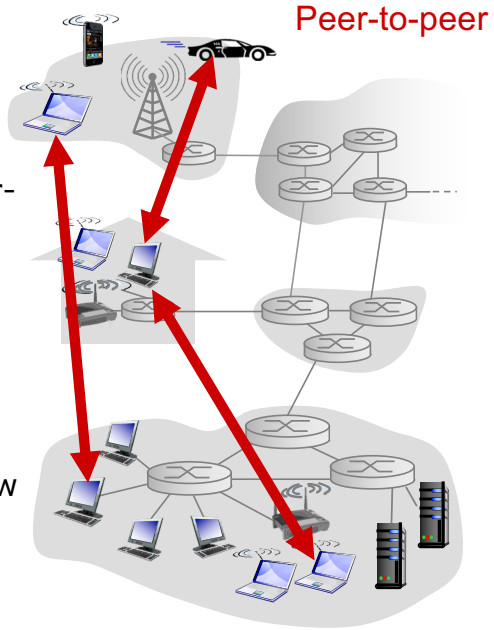
- always-on host
- permanent IP address
- server farms (data center) for scaling

Clients:

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

P2P architecture

- no always on server
- arbitrary end systems directly communicate
- e.g., file sharing (e.g., BitTorrent), peer-assisted download acceleration (e.g., Xunlei), Internet telephony and video conference (e.g., Skype)
- 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 may change IP addresses
  - complex management



Processes communicating

Process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts (with potentially different OS) communicate by exchanging messages

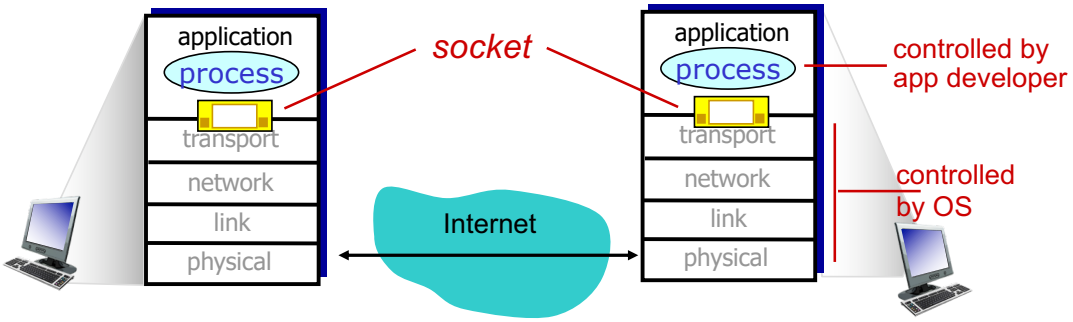
Client process: process that initiates communication

Server process: process that waits to be contacted

- Note: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its socket
- Process is analogous to a house and socket is analogous to door
  - sending process shoves message out its door
  - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process



Addressing processes

- For a process to receive messages, it must have an identifier
- A host has a unique 32-bit IP address
- Ex: use ipconfig (Windows) or ifconfig (Apple) from command prompt to get your IP address
- Q: does the IP address of the host on which the process runs suffice for identifying the process?
- Answer: No, many processes can be running on same host
- Identifier includes both the IP address and port numbers associated with the process on the host.
- Example port numbers:
  - HTTP server: 80
  - Mail server using SMTP: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - Port number: 80
- More on this later

What transport service does an app need?

Reliable Data Transfer

- Some apps (e.g., file transfer, web, financial apps) require data be delivered correctly and completely to the destination
- Loss-tolerant apps: Some apps (e.g., real-time audio /video) can tolerate some loss

Timing

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

Throughput

- Bandwidth sensitive apps: some apps (e.g., multimedia) require minimum amount of bandwidth to be effective
- Other elastic apps (e.g., e-mail, web, file transfer) make use of whatever bandwidth they can get

Security

- Confidentially (encrypt data), data integrity, end-point authentication

Transport service requirements of common apps

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic (few kbps)	no
Internet telephony video conferencing	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
Streaming stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps–10 kbps	yes, 100's msec
instant messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- *connection-oriented:* setup required between client and server processes (handshaking)
- *reliable data transfer* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, min throughput guarantees

UDP service:

- *connectionless:* no handshaking
- *unreliable data transfer* between sending and receiving process
- does not provide: flow control and congestion control
- can pump data into network layer at any rate it pleases

Q: why bother? Why is there a UDP?

Internet apps: Application, transport protocols

Application	Application layer protocol	Underlying Transport Protocol
e-mail	SMTP [RFC 5321]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube)	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	UDP or TCP

Securing TCP

TCP and UDP

- no encryption
- cleartext (i.e., unencrypted) passwords sent into socket traverse Internet in cleartext

SSL (Secure Sockets Layer)

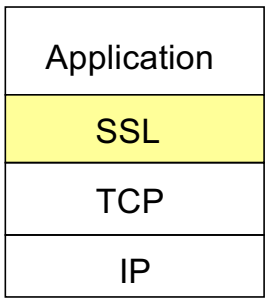
- provides encrypted TCP connection
- data integrity
- end-point authentication

SSL at application layer

- Apps use SSL libraries, which "talk" to TCP

SSL socket API

- cleartext passwords sent into socket traverse Internet encrypted
- see Chapter 8



application with SSL

Application layer protocol defines

- Types of messages exchanged, e.g., request & response messages
- Syntax of message types: what fields in messages & how fields are delineated
- Semantics of the fields, i.e., meaning of information in fields
- Rules for when and how processes send & respond to messages

Open protocols:

- defined in IETF Request for Comments (RFCs)
- allow for interoperability
- e.g., HTTP, SMTP

Proprietary protocols:

- e.g., Skype

Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 Video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP

Web and HTTP

First, a review

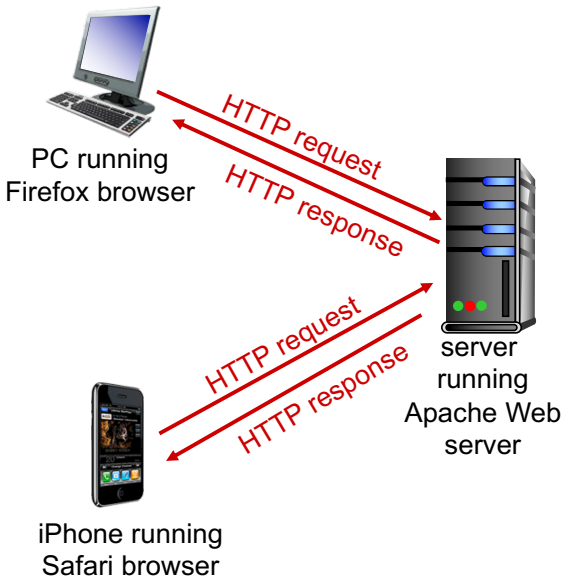
- Web page consists of objects
- Object can be HTML file, JPEG image, Java applet, video clip…
- Web page consists of base HTML file which includes several referenced objects
- Each object is addressable by a URL
- Example URL:

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

Hostname | path name

HTTP (HyperText Transfer Protocol) overview

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



HTTP overview (continued)

Uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

Protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections

Non-persistent HTTP

- At most one object is sent over a TCP connection.
- Connection is then closed.
- Downloading multiple objects required multiple connections.

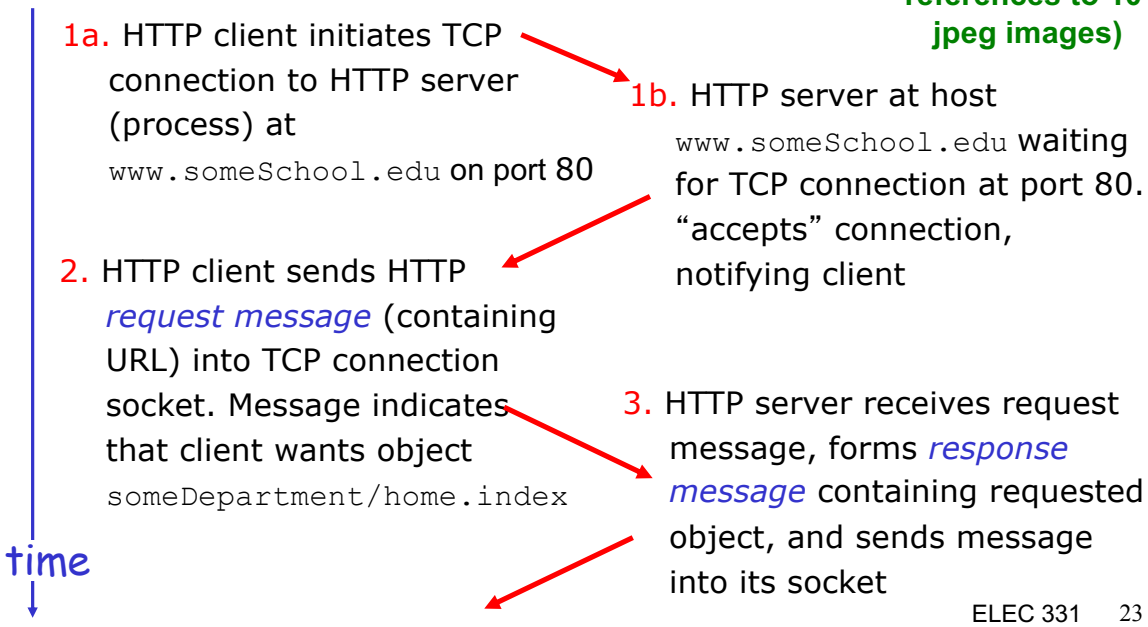
Persistent HTTP

- Multiple objects can be sent over single TCP connection between client and server.

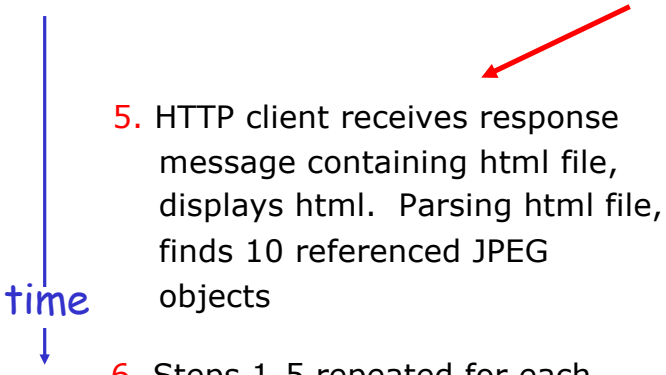
HTTP with non-persistent connection

Suppose user enters URL `www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)



Non-persistent HTTP (cont.)



6. Steps 1-5 repeated for each of 10 JPEG objects

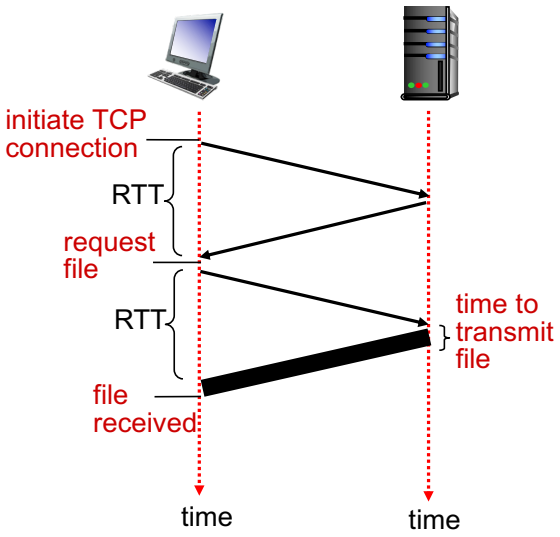
Response time modeling

Definition of RTT: time to send a small packet to travel from client to server and back.

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

total = 2RTT + transmission time



Persistent HTTP

Non-persistent HTTP issues

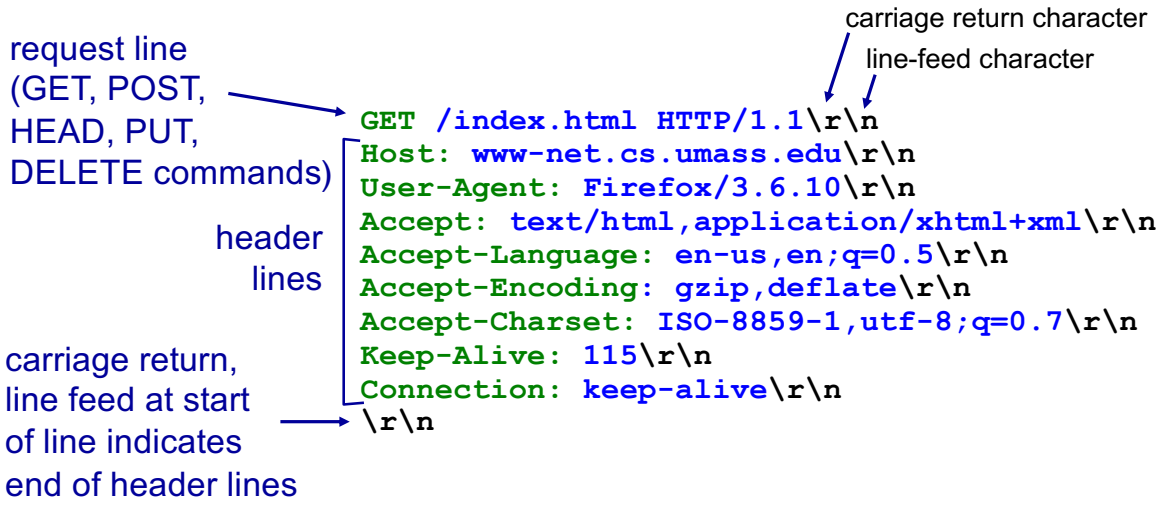
- requires 2 RTTs per object
- OS must work and allocate host resources for each TCP connection
- but browsers often open parallel TCP connections to fetch referenced objects

Persistent HTTP

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

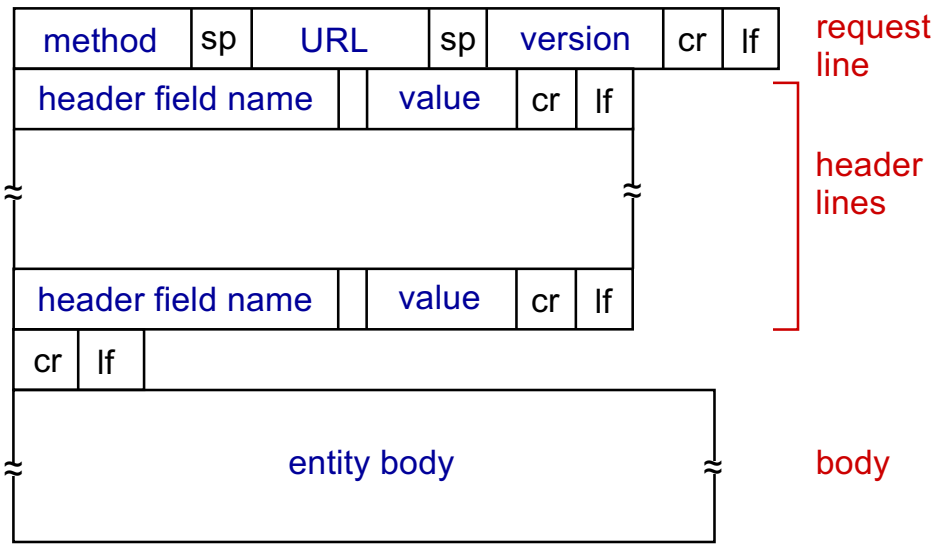
HTTP request message

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



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

HTTP request message: general format



Uploading form input

Post method:

- Web page often includes form input
- Input is uploaded to server in entity body

URL method:

- Uses GET method
- Input is uploaded in URL field of request line:

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


Method types

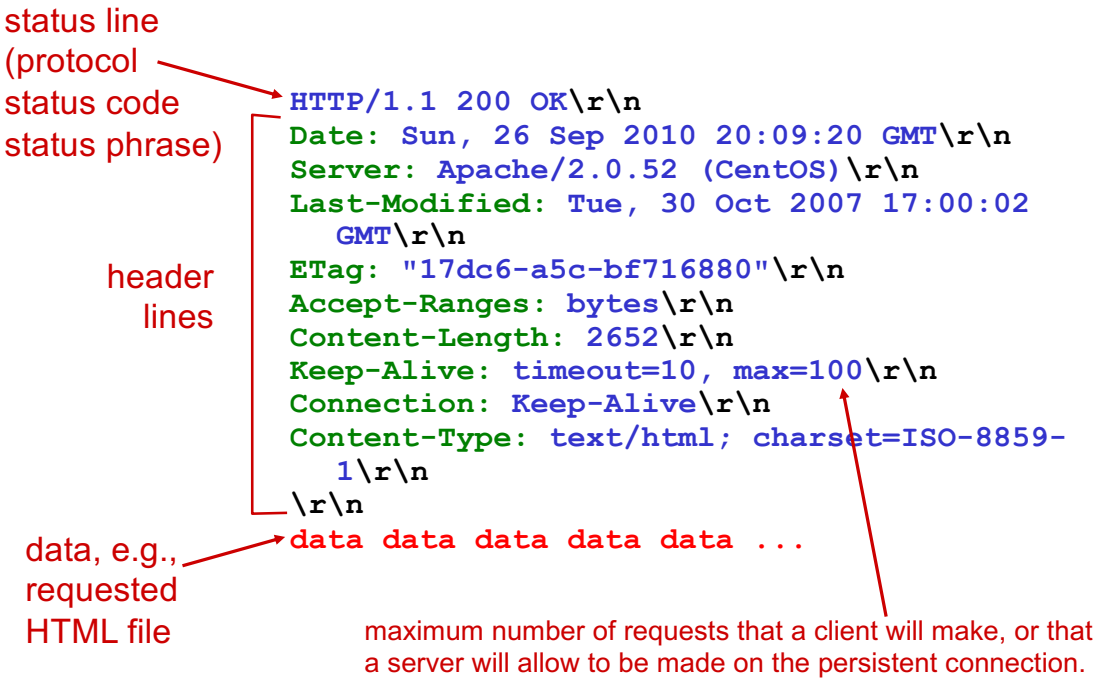
HTTP/1.0

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

HTTP/1.1

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
  - for web publishing
- DELETE
  - deletes file specified in the URL field
- … and a few others

HTTP response message



* 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.

A few sample codes:

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently or **Permanent Redirect**

- requested object moved, new location specified later in this message (Location: header)
- Can be used to support URL shortening. E.g., "http://en.wikipedia.org/wiki/URL_shortening" can be shortened to http://bit.ly/urlwiki

400 Bad Request

- request message not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at gaia.cs.umass.edu
Anything typed in sent to port 80 at gaia.cs.umass.edu

2. Type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php  HTTP/1.1
Host: gaia.cs.umass.edu
```

By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. Look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

4. Try again. This time replace GET with HEAD.

User-server state: cookies

Many major Web sites use cookies

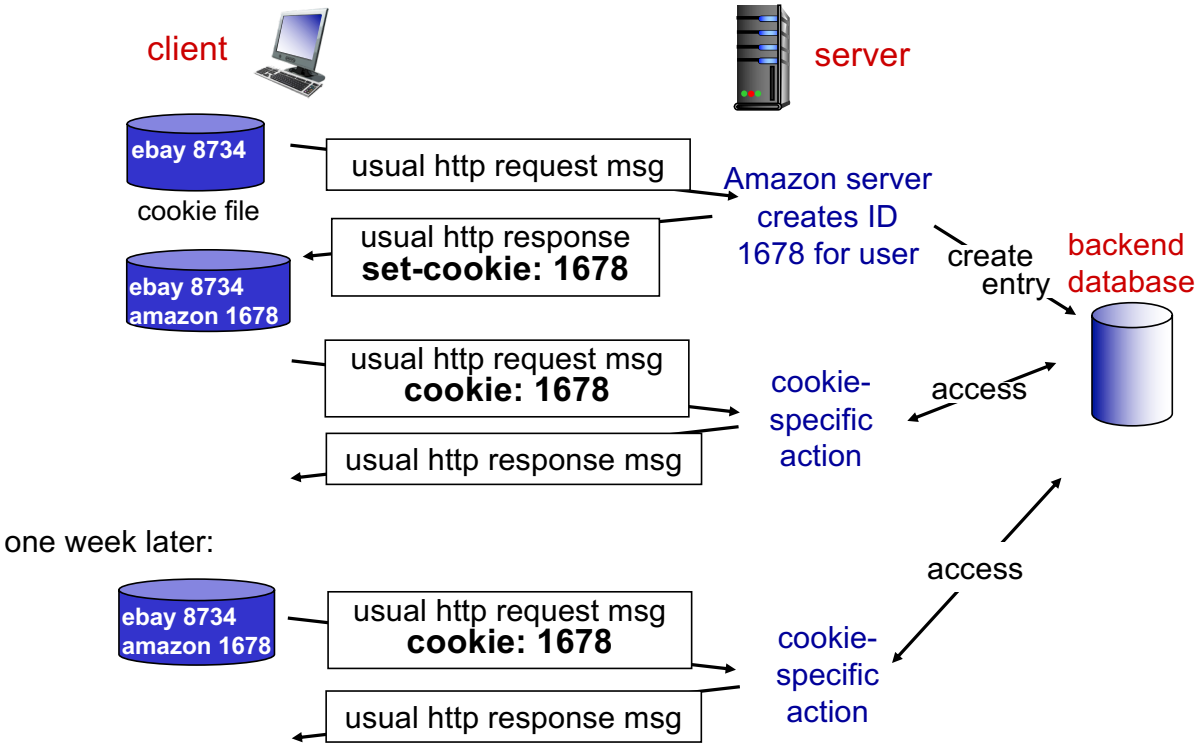
Four components:

1) cookie header line in the HTTP response message
2) cookie header line in HTTP request message
3) cookie file kept on user's host and managed by user's browser
4) backend database at Web site

Example:

- Susan accesses Internet always from same PC
- She visits a specific e-commerce site for first time
- When initial HTTP requests arrives at site, site creates a unique ID and creates an entry in backend database for ID

Cookies: keeping "state" (cont.)



Cookies (continued)

What cookies can be used for:

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

How to keep "state":

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

aside

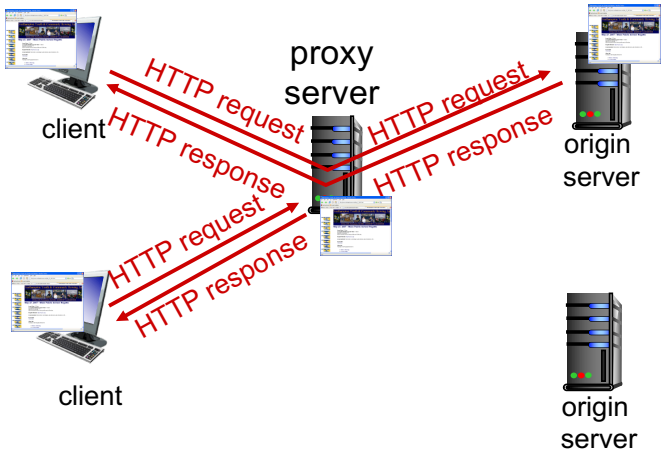
Cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites
- search engines use redirection & cookies to learn yet more
- advertising companies obtain info across sites

Web caches (proxy server)

Goal: satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



More about Web caching

- Cache acts as both client and server
  - server for original requesting client
  - client to origin server
- Typically cache is installed by ISP (university, company, residential ISP)

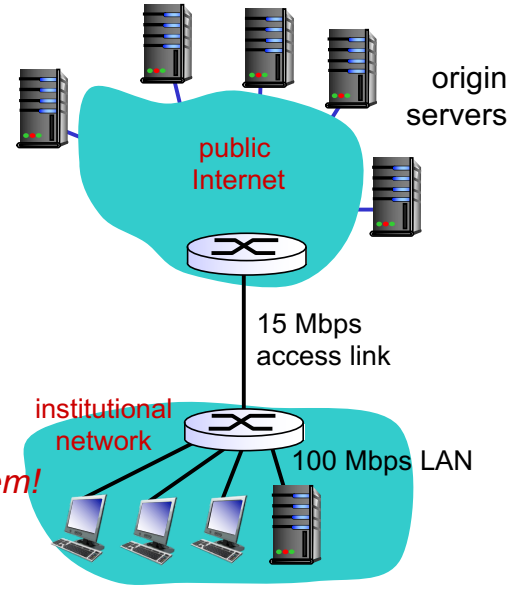
Why Web caching?

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

Caching example

Assumptions

- avg object size: 1 Mbits
- avg request rate from browsers to origin servers: 15 requests per sec
- avg arriving data rate on access link: 15 Mbps
- HTTP request msg negligibly small
- access link rate: 15 Mbps
- RTT between router on Internet side and any origin server: 2 sec



Consequences

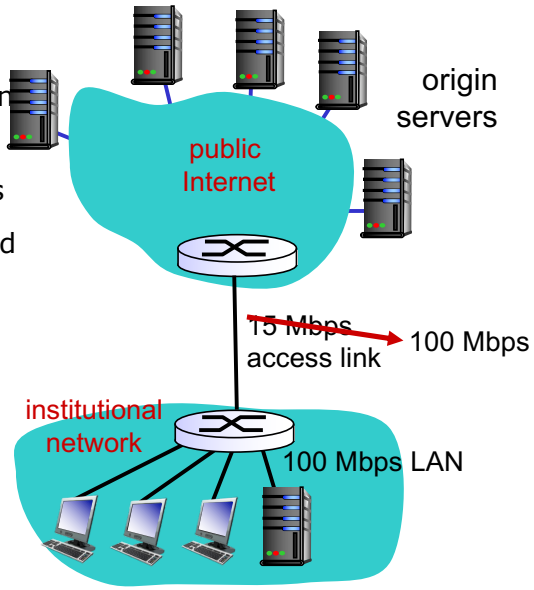
- traffic intensity on LAN: 0.15
- traffic intensity on access link = 1 *problem!*
- total delay = Internet delay + access delay + LAN delay

  = 2 sec + minutes + msecs

Caching example fatter access link

Possible solution

- avg object size: 1 Mbits
- avg request rate from browsers to origin servers: 15 requests/sec
- avg arrival rate on access link: 15 Mbps
- RTT between router on Internet side and any origin server: 2 sec
- access link rate: ~~15 Mbps~~ → 100 Mbps



Consequences

- traffic intensity on LAN: 0.15
- traffic intensity on access link = ~~1~~ → 0.15
- total delay = Internet delay + access delay + LAN delay

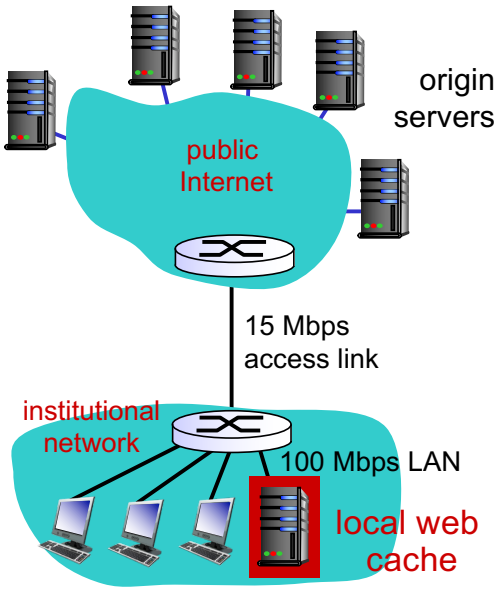
  = 2 sec + ~~minutes~~ → msecs + msecs

Cost: increased access link speed (not cheap!)

Caching example install local cache

Calculating access link utilization, delay with cache:

- suppose 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 = 0.6×15 Mbps = 9 Mbps
  - Traffic intensity on access link = 0.6
- 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 150 Mbps link (and cheaper too!)



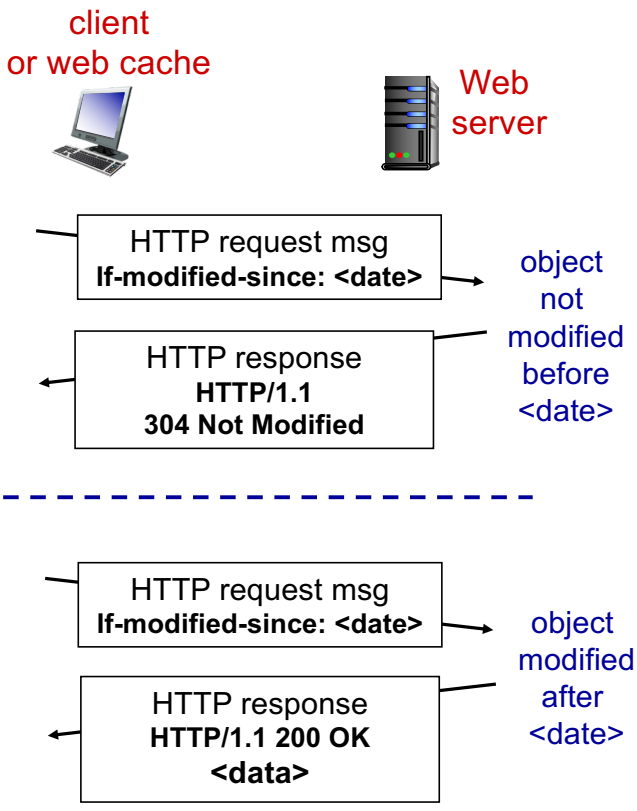
Conditional GET

- *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/1.1 304 Not
  Modified
```



Final words on caching

Cache directives: Must be obeyed by all caching mechanisms along the request/response chain. Specify whether or not an object can be cached. How long it can be cached.

Server

- Cache-Control header: Tells all caching mechanisms from server to client whether they may cache this object.. E.g., Cache-Control: : max-age=3600
- Expires header: date/time at which contents of object are considered stale. Expires: Tue, 23 Sep 2013 16:00:00 GMT
- Etag (Entity Tag) header: used to determine if a cached version of the requested object is identical to the current version of the object on the server. If object's content has changed, a new and different Etag is assigned. ETag: "1269c72-44d6-48f50ca515ed2"

Client

- Cache-Control header: Used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain. E.g., Cache-Control: no-cache
- If-none-match header. Allows a 304 Not Modified to be returned if content is unchanged. E.g., If-None-Match: "284d8af7ad3"

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 Video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP

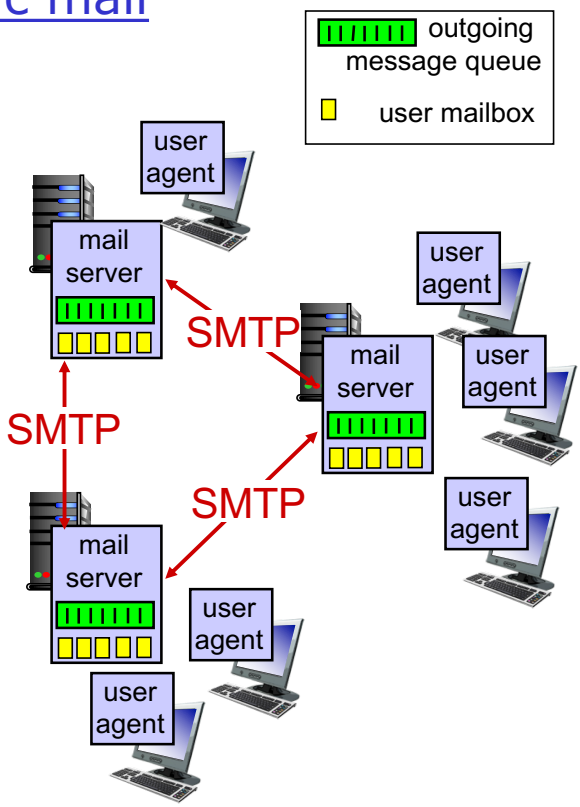
Electronic mail

Three major components:

- user agents
- mail servers
- simple mail transfer protocol (SMTP)

User Agent

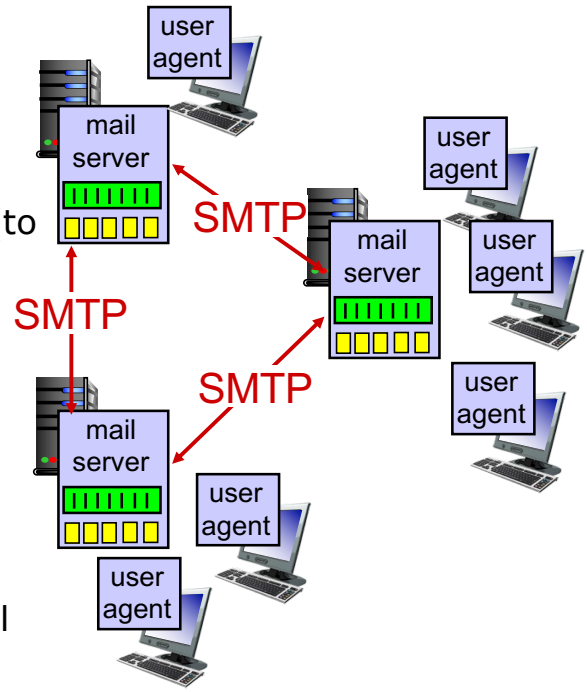
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Microsoft Outlook, Apple Mail
- outgoing, incoming messages stored on server



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 side: sending mail server
  - server side: receiving mail server

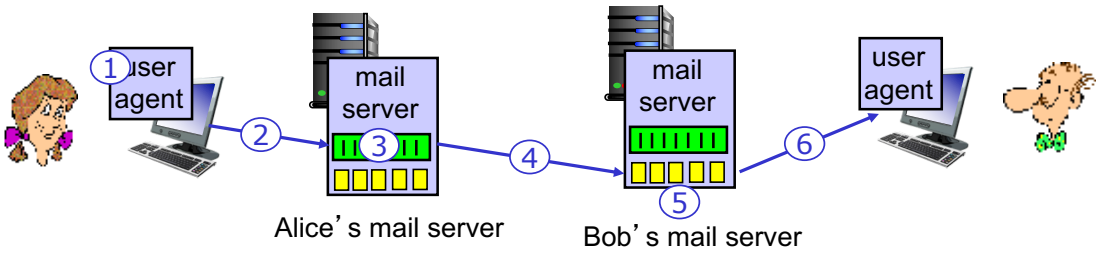


Electronic mail: SMTP [RFC 5321]

- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII

Scenario: Alice sends message to Bob

1) Alice invokes her user agent (UA) to compose message and "to" `bob@someschool.edu`
2) Alice's UA sends message to her mail server; message placed in message queue
3) Client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message



Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```


Try SMTP interaction for yourself:

- **`telnet servername 25`**

  where **`servername`** is the name of the *local* mail server.

- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client (reader)

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII. HTTP does not have this restriction.
- SMTP server uses `CRLF.CRLF` to determine end of message

Comparison with HTTP:

- Both use persistent connections.
- HTTP: pull protocol
- SMTP: push protocol
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in single multipart message

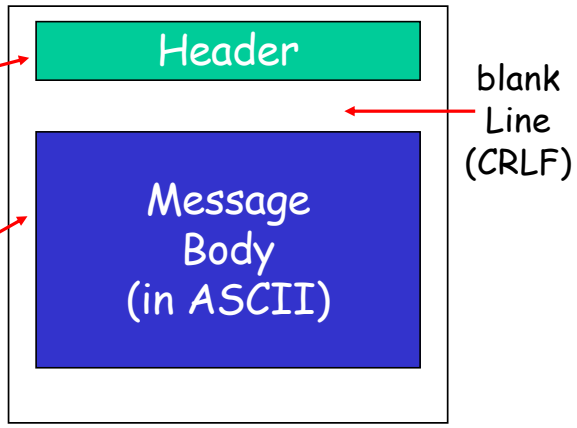
Mail message format

SMTP: protocol for exchanging email messages

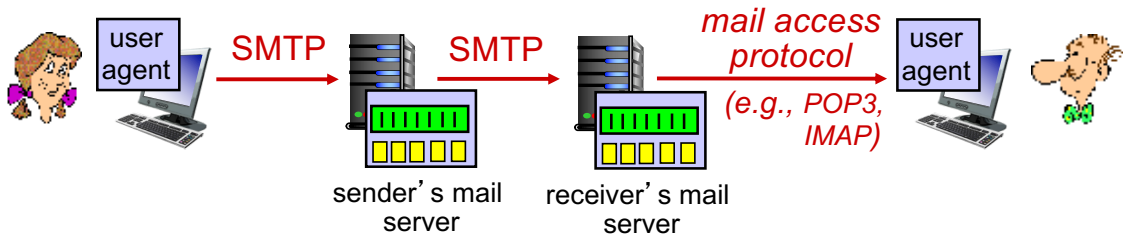
RFC 5322: standard for text message format:

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

  different from SMTP MAIL FROM, RCPT TO commands!
- body
  - the "message", ASCII characters only



Mail access protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP3: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 3501]
    - more features, including manipulation of stored msgs on server
  - HTTP: e.g., gmail, hotmail
    - UA communicates (i.e., sends/retrieves e-mail msgs) with its mail server using HTTP GET and POST.

POP3 protocol

UA opens TCP connection to server on port 110.

authorization phase

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

transaction phase, client:

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

update phase, client:

- **quit**

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

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

POP3 (more) and IMAP

More about POP3

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

IMAP

- Keep all messages in one place: the server
- Allow user to organize and maintain msgs in remote folders
- user can search remote folders for msg matching some criteria
- keep user state across sessions
  - names of folders and mappings between msg IDs and folder name
- Permit user agent obtains components of message

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP

DNS: Domain Name System (RFC 1034, 1035)

People: many identifiers:

- name, social insurance number, passport #

Internet hosts, routers:

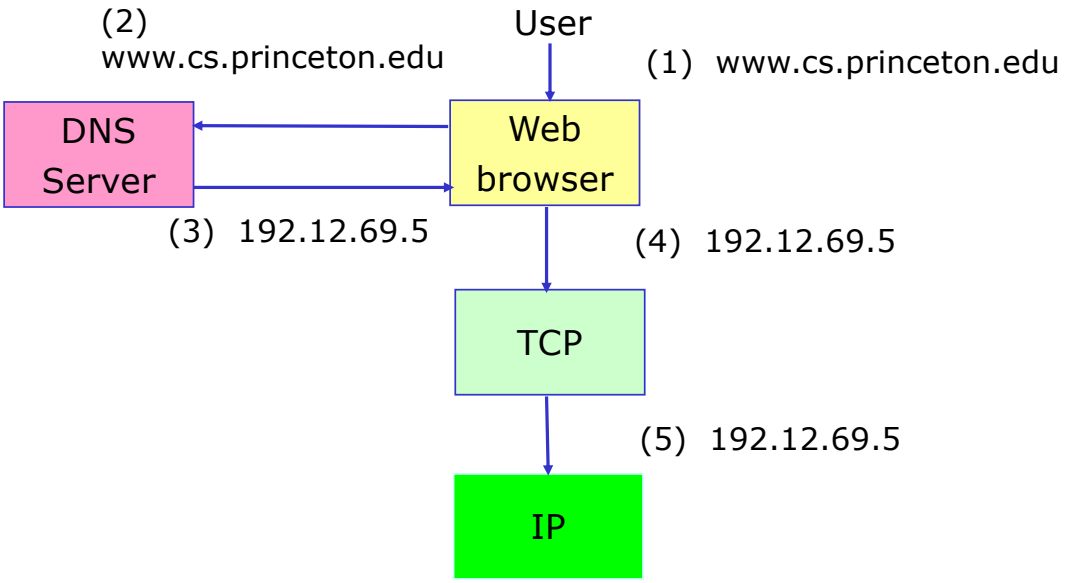
- hostname, e.g., www.yahoo.com - used by humans
- IP address (32 bit) – e.g., 121.7.106.83, used for addressing datagrams

Q: Mapping between hostname and IP addresses?

Domain Name System:

- *distributed database* implemented in hierarchy of many *DNS servers*
- *application-layer protocol* : allow host, routers, DNS servers to communicate to *resolve* names (address/name translation)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network edge
- *Runs over UDP, Uses port 53*

DNS example



- Names translated into addresses, where the numbers (1)-(5) show the sequence of steps in the process.

DNS services

1. Hostname to IP address translation
   - e.g. hostname = mach4.ece.ubc.ca, IP Addr = 137.82.57.104
2. Host aliasing
   - alias name and canonical hostname (real name) translation
   - alias name = ca.yahoo.com, canonical hostname = media-router-fp1.media.yahoo.com
3. Mail server aliasing
   - alias name = ece.ubc.ca, canonical name of mail server = esva.mail-relay.ubc.ca
4. Load distribution
   - replicated Web servers: many IP addresses correspond to one canonical name
   - hostname = www.yahoo.com, set of IP addresses: 206.190.39.43, 206.190.39.42.

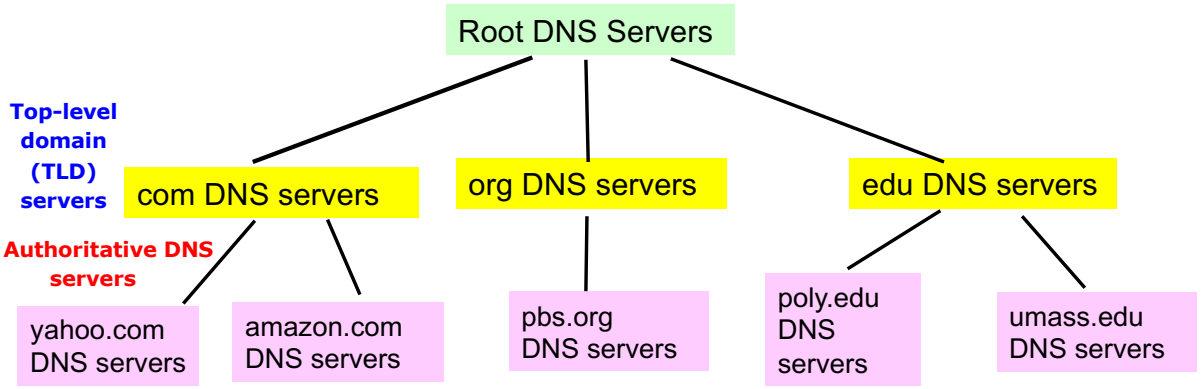
DNS

Why not using a centralized DNS?

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

doesn't *scale!*

Distributed, hierarchical database

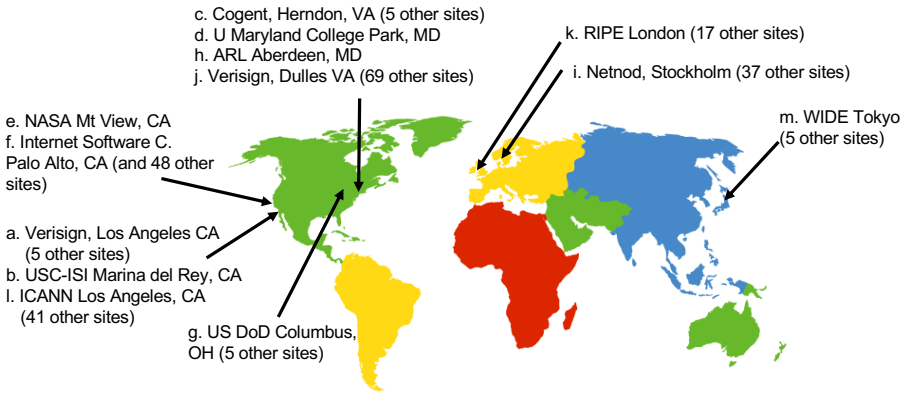


Client wants IP address for www.amazon.com; 1st approximation

- client queries a 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

Root DNS servers

- Contacted by local name server that cannot resolve name
- Provide IP addresses of the TLD servers
- Over 400 root DNS servers scattered all over the world for both security and reliability purposes
- These root DSN servers are managed by 13 different organizations
- http://www.root-servers.org/



TLD and authoritative servers

- Top-level domain (TLD) servers: responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
  - Provide IP addresses for authoritative DNS servers
  - Verisign Global Registry Services maintains TLD servers for .com top-level domain
  - Educause maintains TLD servers for .edu top-level domain
- Authoritative DNS servers: organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web and mail).
  - Provide publicly accessible DNS records
  - Can be maintained by organization or service provider
  - In our department, it is the "dns.ece.ubc.ca"

Local DNS server

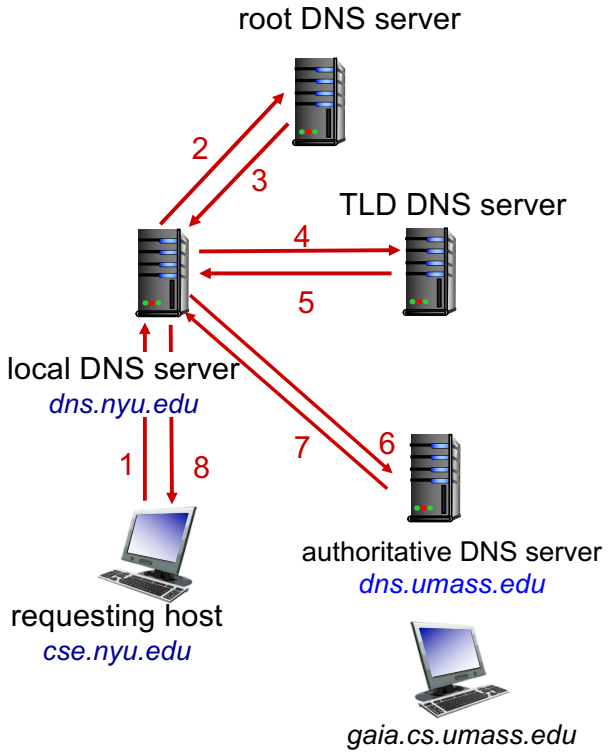
- Does not strictly belong to hierarchy
- Each ISP (residential ISP, company, university) has one.
  - Also called "default name server"
- When a host makes a DNS query, query is sent to its local DNS server
  - Acts as a proxy, forwards query into hierarchy.
  - e.g., in our department, the local name servers are: dns1.ece.ubc.ca, dns2.ece.ubc.ca

DNS name resolution example

- Host at cse.nyu.edu wants IP address for gaia.cs.umass.edu

iterated query:

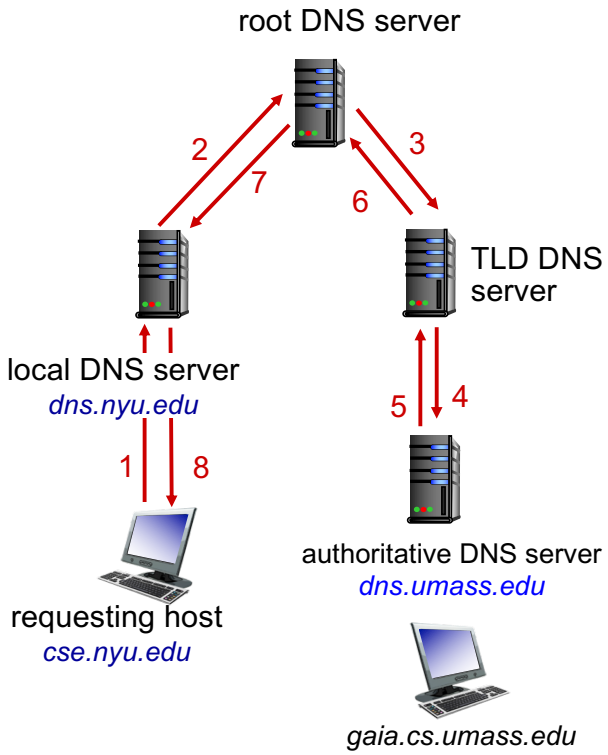
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

Recursive query:

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



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

DNS: Caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL) (e.g., two days).
  - TLD servers typically cached in local name servers.
    - Thus root name servers not often visited.
- cached entries may be out-of-date (best effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms under design by IETF
  - RFC 2136
  - http://www.ietf.org/html.charters/dnsind-charter.html

DNS records

DNS: distributed database storing resource records (RRs)

RR format: **(name, value, type, TTL)**

- Type=A
  - **name** is hostname
  - **value** is IP address
  - `(relay1.bar.foo.com, 145.37.93.2, A)`
- Type=NS
  - **name** is domain (e.g., `foo.com`)
  - **value** is hostname of authoritative DNS server for this domain
  - `(foo.com, dns.foo.com, NS)`
- Type=CNAME
  - **name** is alias name for some "canonical" (the real) name
  - **value** is canonical name
  - `(foo.com, relay1.bar.foo.com, CNAME)`
- Type=MX
  - **value** is name of mailserver associated with **name**
  - `(foo.com, mail.bar.foo.com, MX)`

DNS records example

- Sample records maintained by root server

```
<edu, a3.nstld.com, NS>
<a3.nstld.com, 192.5.6.32, A>
<com, a.gtld-servers.net, NS>
<a.gtld-servers.net, 192.5.6.30, A>
...
```

- Records maintained by a3.nstld.com server for .edu domain

```
<princeton.edu, dns.princeton.edu, NS>
<dns.princeton.edu, 128.112.125.15, A>
...
```

- Records maintained by authoritative server dns.princeton.edu

```
<cs.princeton.edu, dns1.cs.princeton.edu, NS>
<dns1.cs.princeton.edu, 128.112.136.10, A>
...
```


DNS records example (cont.)

- Sample records maintained by authoritative server dns1.cs.princeton.edu

```
<penguins.cs.princeton.edu, 128.112.155.166, A>
<www.cs.princeton.edu, coreweb.cs.princeton.edu, CNAME>
<coreweb.cs.princeton.edu, 128.112.136.35, A>
<cs.princeton.edu, mail.cs.princeton.edu, MX>
<mail.cs.princeton.edu, 128.112.136.72, A>
...
```

- Local name server maintains the cache and also records for root servers

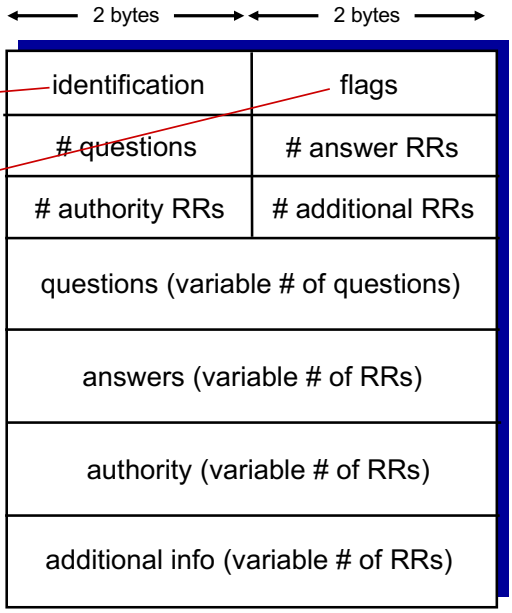
```
<'root', a.root-servers.net, NS>
<a.root-servers.net, 198.41.0.4, A>
...
```

DNS protocol, messages

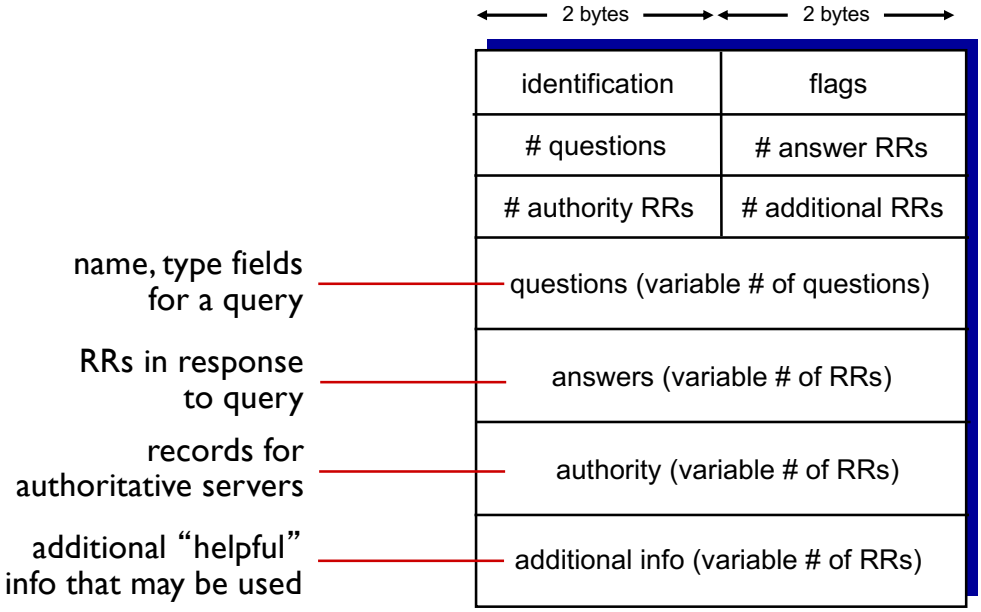
DNS protocol: *query* and *reply* messages, both with same *message format*

Message Header

- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query (0) or reply (1)
  - reply is authoritative (1)
  - recursion desired: set by client
  - recursion available: set by server



DNS protocol, messages



Inserting records into DNS

- Example: new startup “Network Utopia”
- Register name networkutopia.com at a DNS registrar (e.g., Network Solutions)
  - Provide names and IP addresses of authoritative name server (primary and secondary)
  - Registrar inserts two RRs into the com TLD server:

```
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
```

- create authoritative server Type A record for www.networkutopia.com and Type MX record for mail.networkutopia.com

```
(www.networkutopia.com, 192.10.232.2, A)
(networkutopia.com, mail.networkutopia.com, MX)
(mail.networkutopia.com, 192.10.232.3, A)
```


Attacking DNS

DDoS attacks

- Bombard root servers with traffic
  - Not successful to date
  - packet filters
  - Local DNS servers cache IP address of TLD servers, allowing to bypass root server
- Bombard TLD servers
  - Potentially more dangerous

Redirect attacks

- Man-in-middle
  - Intercept queries and returns bogus replies
- DNS poisoning
  - Send bogus relies to DNS server, which caches
- Difficult to implement

Exploit DNS for reflection attack

- Send queries with spoofed source address: target IP
- Size of reply message > > Size of query message

Chapter 2: Application layer

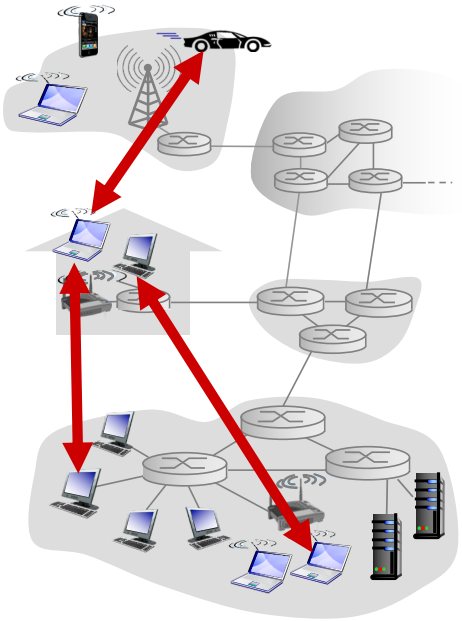
- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 Video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP

Peer-to-Peer applications

1. File Distribution
   - Distribute a file from a single source to a large number of peers
   - BitTorrent protocol
2. Searching for Information
   - Support search and update operation
   - Distributed Hash Tables (DHTs)
3. Skype: P2P Internet telephony application

P2P architecture

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



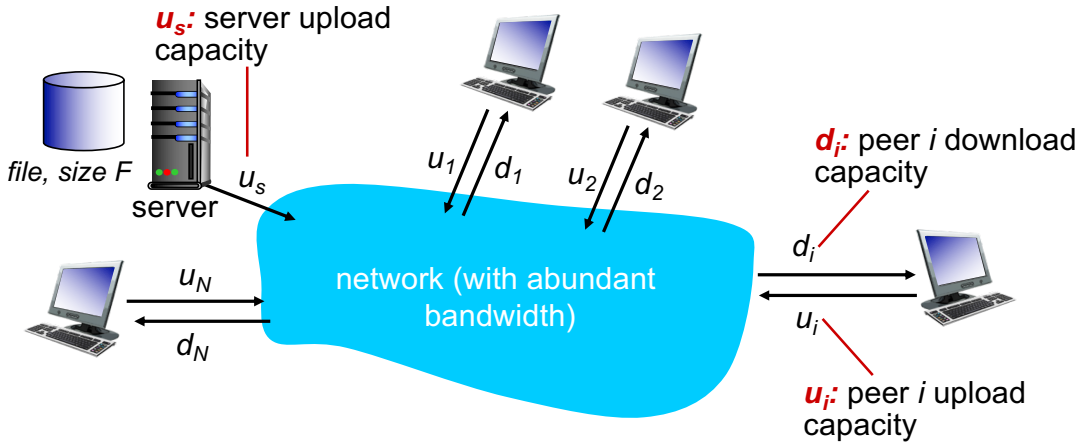
Examples:

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

File distribution: client-server vs P2P

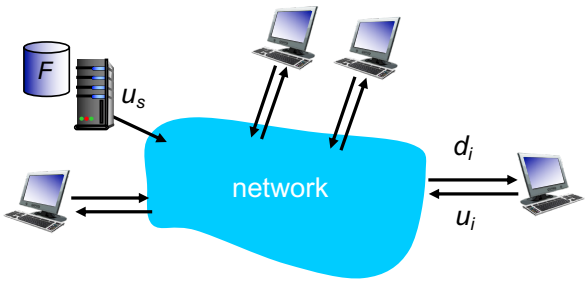
Question: how much time to distribute file (size F bits) from one server to N peers?

- *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
  - download time for client with bottleneck link d_{min} : F/d_{min}



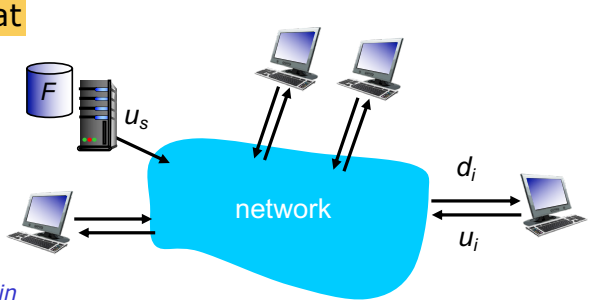
time to distribute F bits to N clients using client-server approach

$$D_{cs} \geq \max\{ NF/u_s, F/d_{min} \}$$

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 bits to N clients using P2P approach

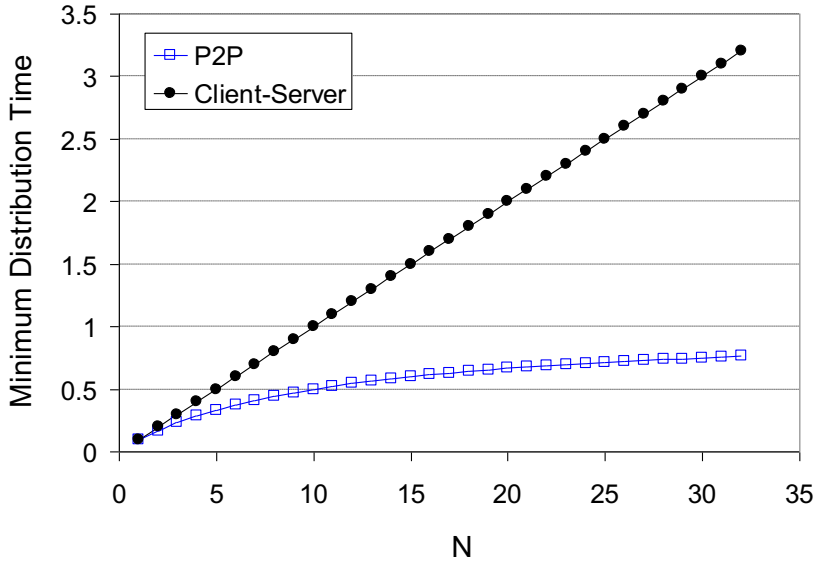
$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in *N* …

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

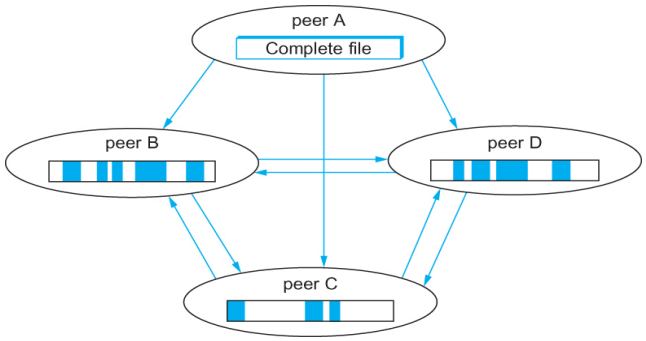
Client-server vs. P2P: example

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



- Peers being consumers as well as redistributors of bits.

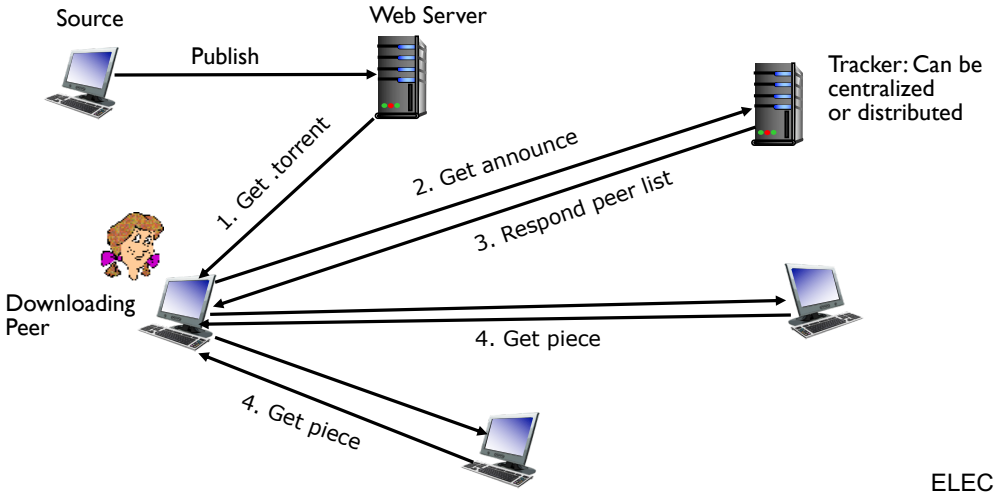
BitTorrent basics



- File is divided into fixed size (e.g., 256 Kbytes) chunks
- Peers in torrent send/receive file chunks
- Peers in a BitTorrent swarm download from other peers that may not yet have the complete file
- Replication is a natural side-effect
- As soon as a peer downloads a particular piece, it becomes another source for that piece

P2P file distribution: BitTorrent

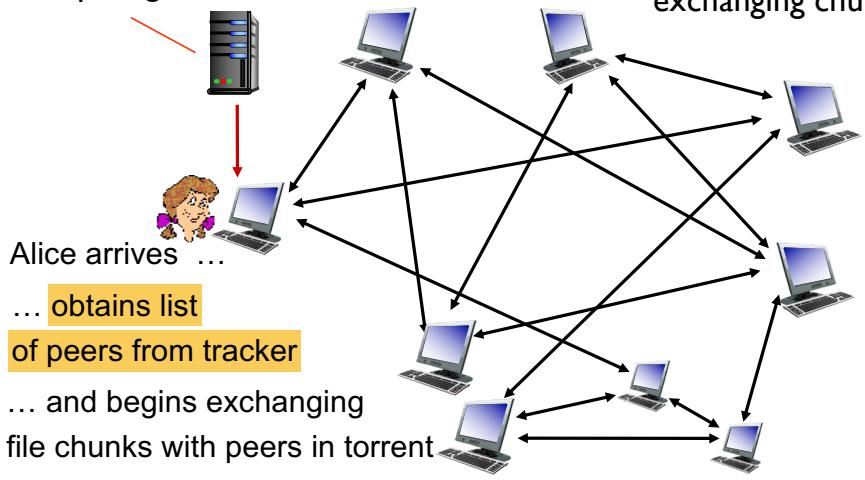
- To share a file, a peer first creates a ".torrent" file
- .torrent file: contains metadata about the file to be shared (e.g., file name, file size, size of each piece/chunk, SHA-1 hash code of each piece, URL of tracker)
- Torrent file is usually published to some well-known website.



P2P file distribution: BitTorrent

tracker: tracks peers participating in torrent

torrent: group of peers exchanging chunks of a file



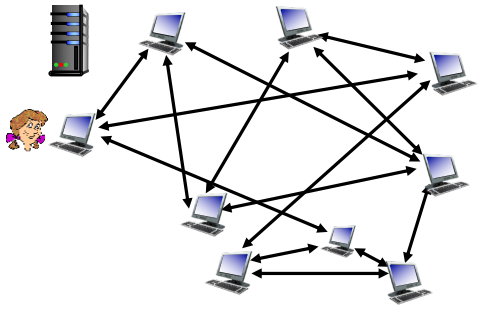
Alice arrives …

… obtains list of peers from tracker

… and begins exchanging file chunks with peers in torrent

BitTorrent Schematic

- 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: refers to the phenomenon that peers may dynamically join and leave the system at well
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

Piece Selection

- At any given time, different peers have different subsets of file chunks
- Periodically, Alice asks her neighbors for list of chunks that they have
- Use random first selection at the initial phase
  - Randomly select the first few pieces (usually around four)
- After that, use rarest first policy
  - Select those chunks which have the fewest repeated copies
- At the end, end-game mode
  - Send requests for all missing pieces to all the peers

Peer Selection

sending chunks: tit-for-tat

- Alice sends chunks to those four peers currently supplying her data chunks at the highest rate
  - these four peers are said to be unchoked
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - This peer is said to be optimistically unchoked
  - newly chosen peer may join top 4
- other peers are choked by Alice (do not receive chunks from her)

Peer Selection

Anti-snubbing algorithm

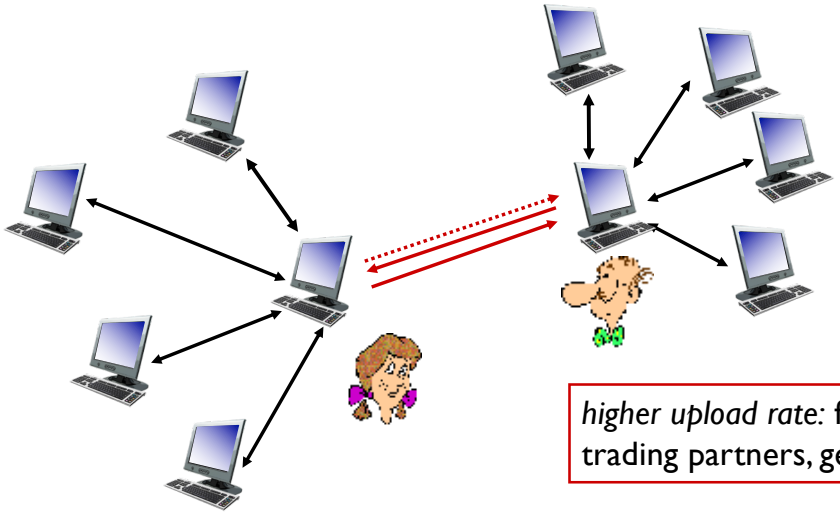
- Alice runs the anti-snubbing algorithm whenever she is choked by all of her peers
  - A peer is snubbed if it does not receive any data in 60 sec.
- Alice can do better to run optimistically unchoking more often to explore more peers who are willing to cooperate.
- Thus, the anti-snubbing algorithm stops uploading to peers selected tit-for-tat so that optimistically unchoking can be performed more often.

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 !

Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 Video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP

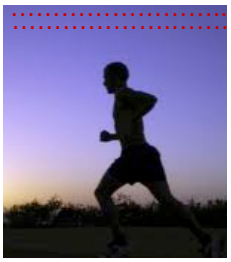
Video Streaming and CDNs: context

- video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube: 37%, 16% of downstream residential ISP traffic
  - ~1B YouTube users, ~75M Netflix users
- challenge: scale - how to reach ~1B users?
  - single mega-video server won't work (why?)
- challenge: heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- solution: distributed, application-level infrastructure

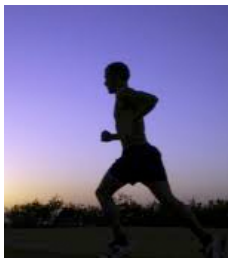
Multimedia: video

- video: sequence of images displayed at constant rate
  - e.g., 24 images/sec
- digital image: array of pixels
  - each pixel represented by bits
- coding: use redundancy within and between images to decrease # 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*



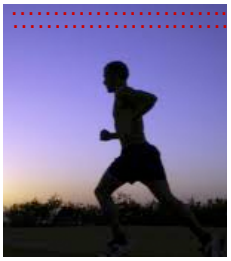
frame *i* + 1

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, < 1 Mbps)

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*)



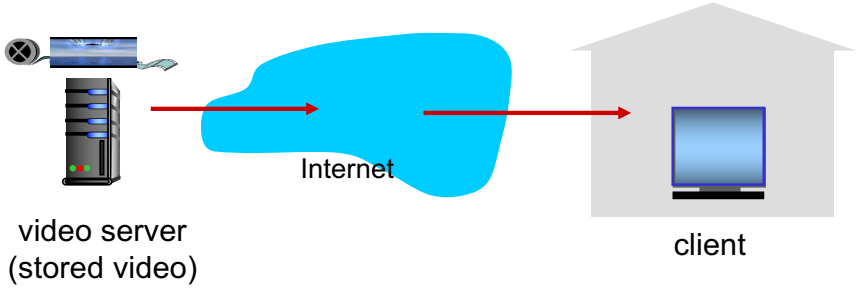
frame *i*

frame *i* + 1

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

Streaming stored video

simple scenario:



Streaming multimedia: DASH

- DASH: Dynamic Adaptive Streaming over HTTP
- A video is encoded into several different versions, each having a different bit rate (e.g., 300 kbps, 1 Mbps, 3 Mbps)
- Each video version is stored in server, each with a different URL
- Server has a *manifest file* which provides a URL for each version along with its bit rate
- client:
  - Select one chunk at a time by specifying a URL and a byte range in HTTP GET request message for each chunk.
  - periodically measures server-to-client bandwidth by rate determination algorithm
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

Streaming multimedia: DASH

- DASH: Dynamic Adaptive Streaming over HTTP
- "intelligence" at client: client determines
  - when to request chunk (so that buffer starvation, or overflow does not occur)
  - what encoding rate to request (higher quality when more bandwidth available)
  - where to request chunk (can request from URL server that is "close" to client or has high available bandwidth)

Content distribution networks

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

- option 1: single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long path to distant clients
  - multiple copies of video sent over outgoing link

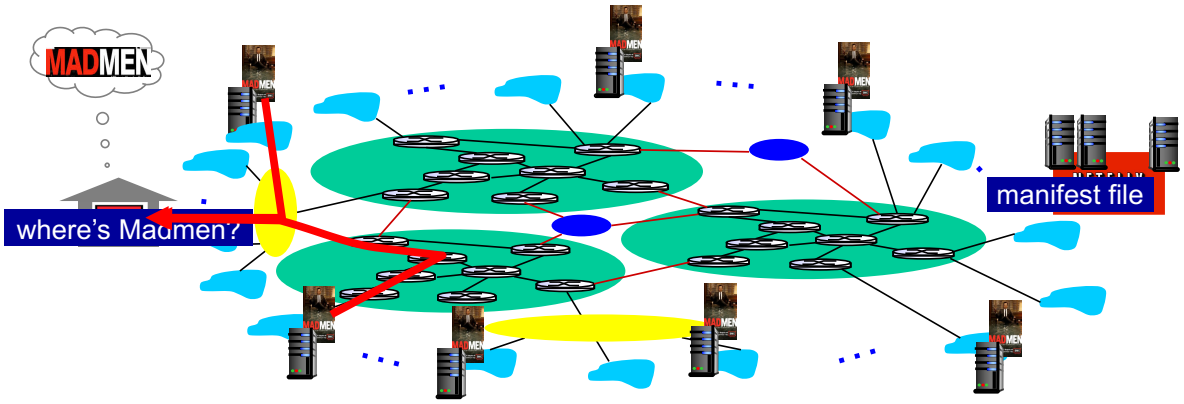
- ….quite simply: this solution doesn't scale

Content distribution networks

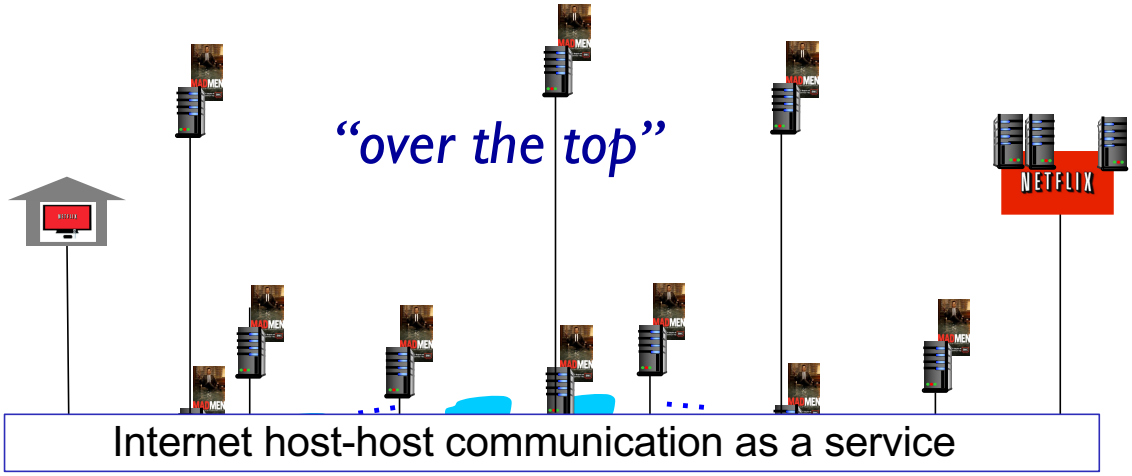
- challenge: how to stream content (selected from millions of videos) to hundreds of thousands of simultaneous users?
- option 2: store/serve multiple copies of videos at multiple geographically distributed sites (CDN)
  - *enter deep*: push CDN servers deep into many access networks
    - close to users
    - used by Akamai, 1700 locations
  - *bring home*: smaller number (10's) of larger clusters in IXPs near (but not within) access networks
    - used by Limelight
    - lower maintenance and management overhead

Content Distribution Networks (CDNs)

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



Content Distribution Networks (CDNs)



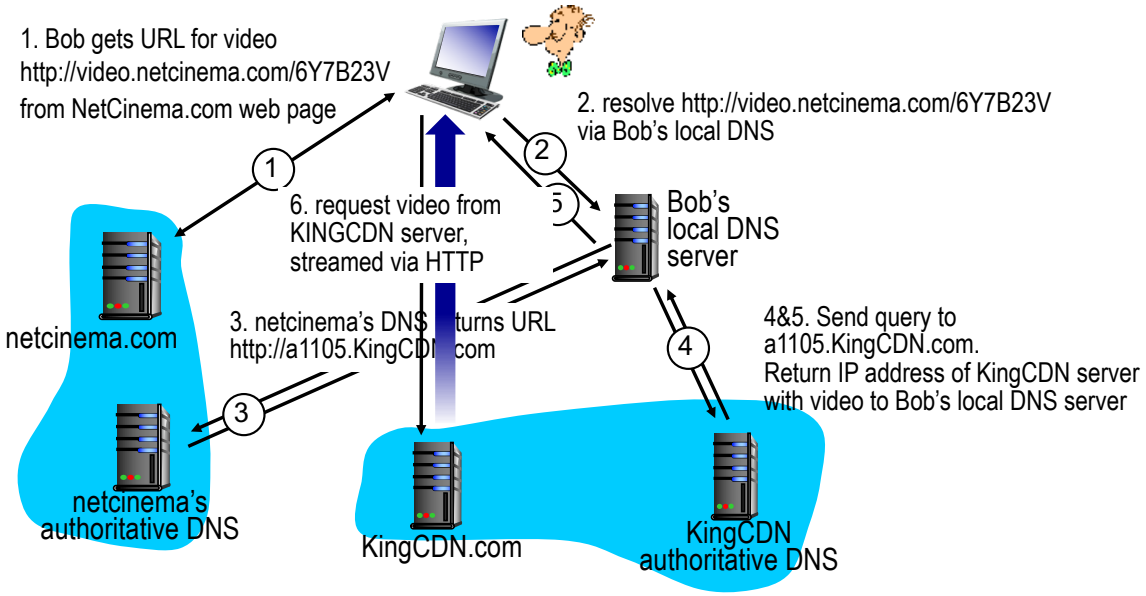
OTT challenges: coping with a congested Internet

- from which CDN node to retrieve content?
- viewer behavior in presence of congestion?
- what content to place in which CDN node?

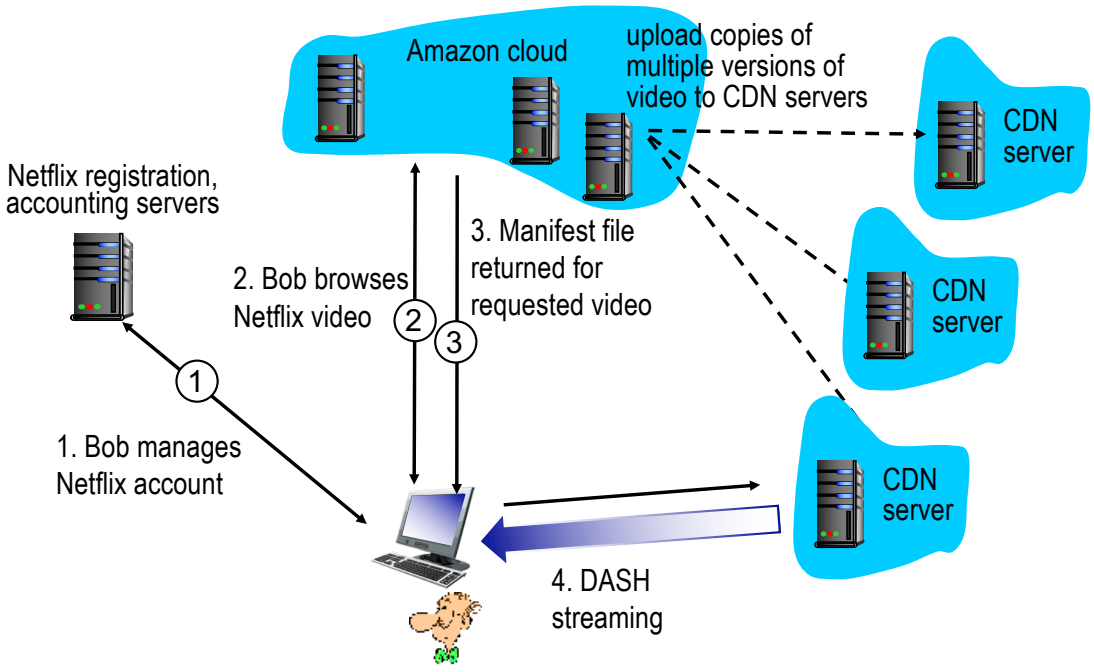
more .. in Chapter 9

CDN content access: a closer look

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



Case study: Netflix



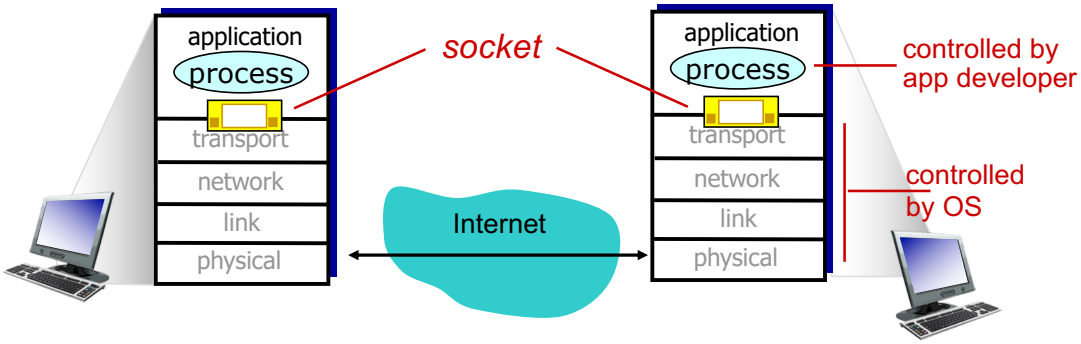
Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 Electronic Mail
  - SMTP, POP3, IMAP
- 2.4 DNS
- 2.5 P2P applications
- 2.6 Video streaming and content distribution networks
- 2.7 Socket programming with UDP and TCP

Socket programming

Goal: learn how to build client/server applications that communicate using sockets

Socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- UDP: unreliable datagram
- TCP: reliable, byte stream-oriented

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Socket programming *with UDP*

UDP: no "connection" between client and server

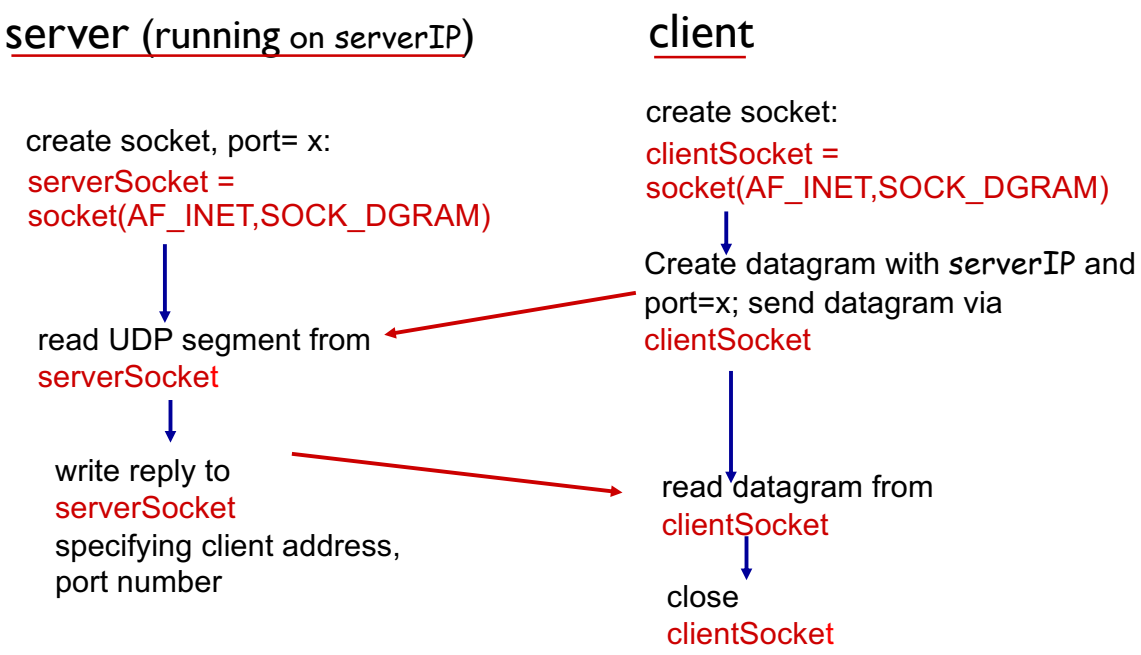
- no handshaking
- sender explicitly attaches IP destination address and destination port number to each packet
- Receiver extracts sender (i.e., source)'s IP address, source port number from received packet

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

Application viewpoint:

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

Client-server socket interaction: UDP



Example app: UDP client

Python UDPClient

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

`serverName = 'hostname'`

`serverPort = 12000`

create UDP socket for server → `clientSocket = socket(AF_INET,`
`SOCK_DGRAM)`

get user keyboard input → `message = raw_input('Input lowercase sentence: ')`

Attach server name, port to message; send into socket → `clientSocket.sendto(message.encode(),`
`(serverName, serverPort))`

read reply characters from socket into string → `modifiedMessage, serverAddress =`
`clientSocket.recvfrom(2048)`

print out received string and close socket → `print(modifiedMessage.decode() )`

`clientSocket.close()`

Example app: UDP server

Python UDPServer

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

create UDP socket

assign local port number 12000 to socket

loop forever

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

send upper case string back to this client

Socket programming with TCP

client must contact server

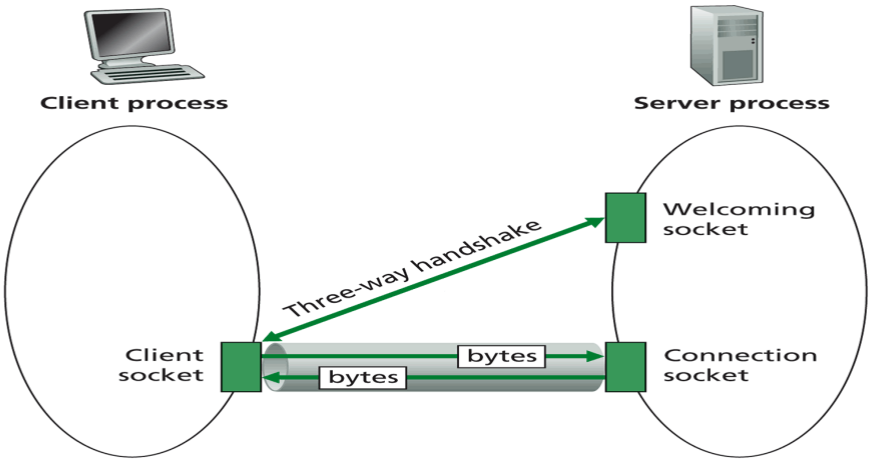
- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

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

- when contacted by client, server TCP creates new socket for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

Socket programming *with TCP*

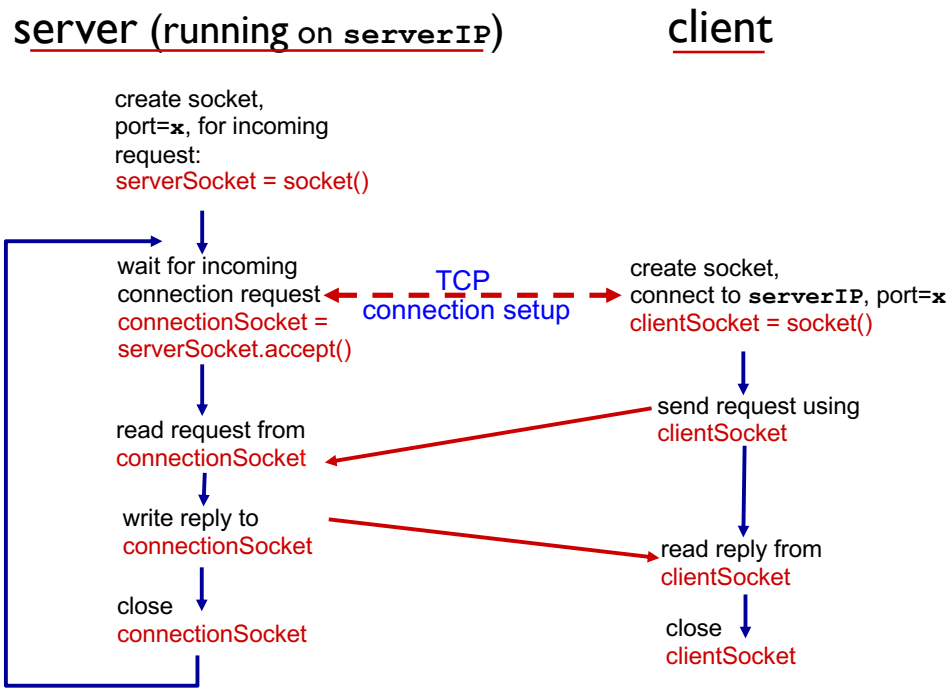


- Client socket, welcoming socket, and connection socket

application viewpoint

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

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

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

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

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

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

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

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

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

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

Chapter 2: summary

Our study of network apps now complete!

- Application architectures
  - client-server
  - P2P
- application service requirements
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP
  - POP3, IMAP
  - DNS
  - P2P: BitTorrent
  - video streaming, CDNs
  - socket programming:
  - TCP, UDP sockets

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

- control vs. data messsages
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable message transfer
- “complexity at network edge”