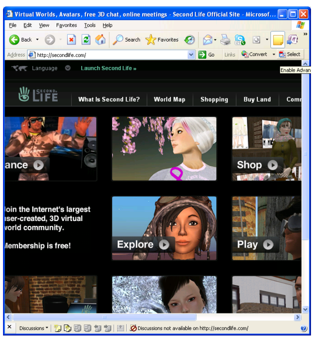
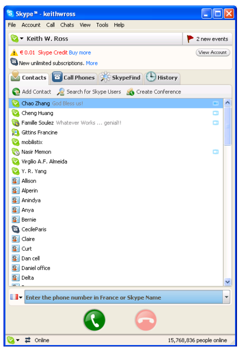
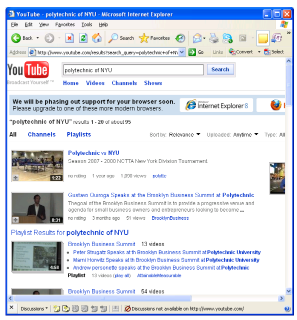
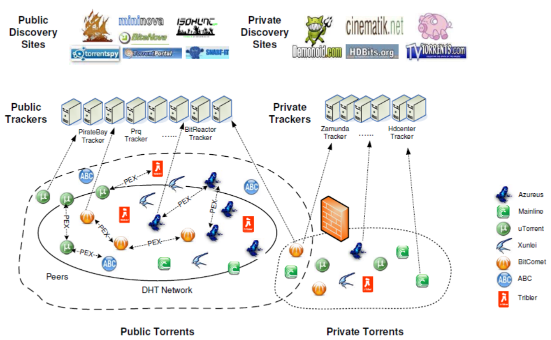
CS2505
Section 2 - Application Layer

Section 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
- 2.5 DNS
- 2.6 File Distribution

Section 2: Application Layer

Our goals:

- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS

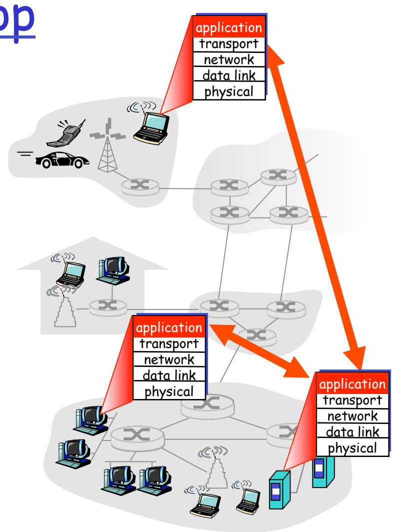
Section 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
- 2.5 DNS
- 2.6 File Distribution

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



Internet Addressing

Every computer on the Internet gets a unique "IP address"

- Assigned when it connects to the network
- Used to identify the destination when sending a packet
- Used by receivers to check who sent the packet (and to reply!)

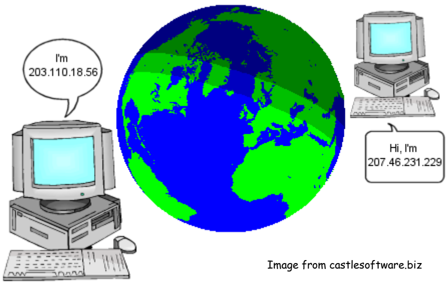


Image from castlesoftware.biz

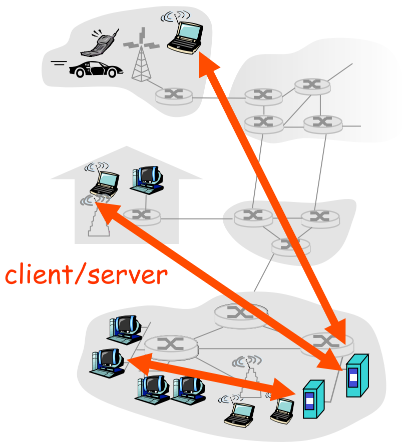
For convenience we often use hostnames

- But these are mapped to a corresponding IP address when sending a packet
- E.g. cs1.ucc.ie -> 143.239.75.218

Application architectures

- Client-server
  - Including data centers / cloud computing
- Peer-to-peer (P2P)
- Hybrid of client-server and P2P

Client-server architecture



server:
- always-on host
- permanent IP address
- server farms for scaling

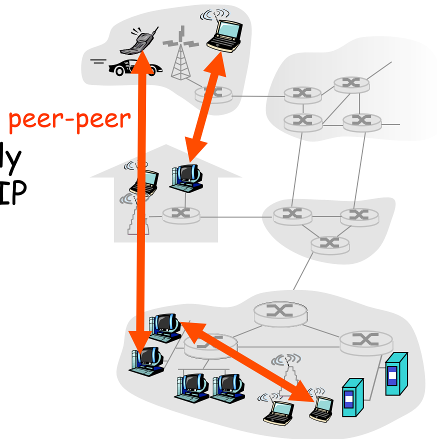
clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

Google Data Centers

- Estimated cost of data center: $600M
- Large Internet companies such as Google spend many billions of Euro on data centers

Pure P2P architecture

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



Highly scalable but difficult to manage

Hybrid of client-server and P2P

Skype

- voice-over-IP P2P application
- centralized server: finding address of remote party:
- client-client connection: direct (not through server)

Instant messaging

- chatting between two users is P2P
- centralized service: client presence detection/ location
  - user registers its IP address with central server when it comes online
  - user contacts central server to find IP addresses of buddies

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

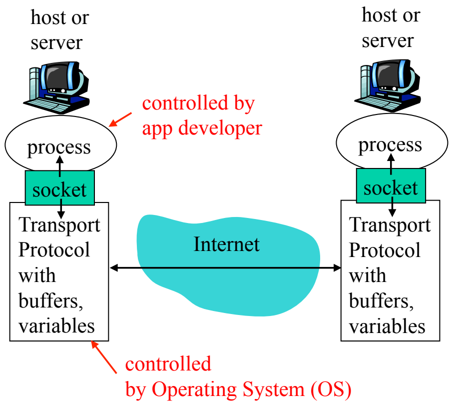
Client process: process that initiates communication

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
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process



API: (1) choice of transport protocol; (2) ability to fix a few parameters (lots more on this later)

Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Exercise: use `ipconfig` from command prompt to get your IP address (Windows)
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* No, *many* processes can be running on same
- *Identifier* includes both IP address and port numbers associated with process on host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25

App-layer protocol defines

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

Public-domain protocols:
- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP, BitTorrent

Proprietary protocols:
- e.g., Skype, ppstream

What transport service does an app need?

Data loss

- some apps (e.g., audio) can tolerate some loss
- other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Timing

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

Throughput

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

Security

- Encryption, data integrity, …

Transport service requirements of common apps

Application	Data loss	Throughput	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
instant messaging	no loss	elastic	yes and no

Popular Internet transport protocols

Transmission Control Protocol (TCP)

- Offers reliable end-to-end delivery between the sending and receiving processes
- Uses sequence numbers to detect lost or out-of-order packets; these are retransmitted (NB: extra delay)
- Additional features to control the sending rate so as to match the available network and receiver buffer capacity
- does not provide: timing, minimum throughput guarantees, security

Unreliable Datagram Protocol (UDP)

- A very basic delivery service that offers no assurance about delivery between sending and receiving process
- does not provide: timing, minimum throughput guarantees, security, rate control

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

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

Section 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
- 2.5 DNS
- 2.6 File Distribution

Web and HTTP

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

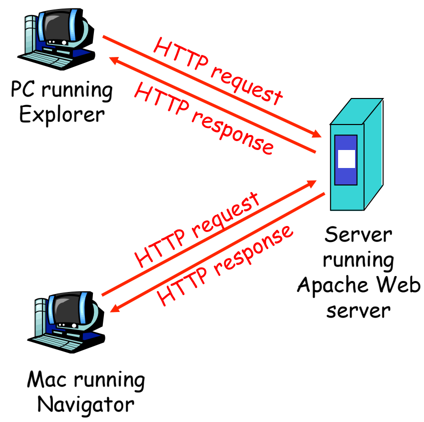
`http://www.cs.ucc.ie/pic.gif`

protocol host name path name

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- Uses TCP as the Transport Protocol



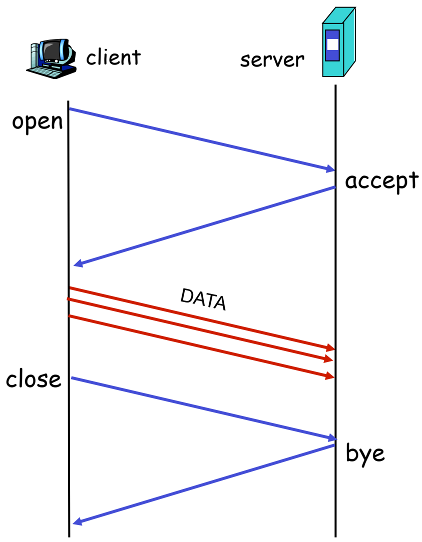
TCP Basics

TCP requires the sender and receiver to setup a connection

- *Before* they transfer any user data
- And to close it when they are finished

This incurs delays

- Shown in the time-sequence diagram opposite as *Round-Trip Time (RTT)* delays



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

Nonpersistent HTTP
- At most one object is sent over a TCP connection.

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

Nonpersistent HTTP

Suppose user enters URL www.ucc.ie/index.html

(contains text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at www.ucc.ie on port 80

1b. HTTP server at host www.ucc.iewaiting 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 index.html

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Nonpersistent HTTP (cont.)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

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

time

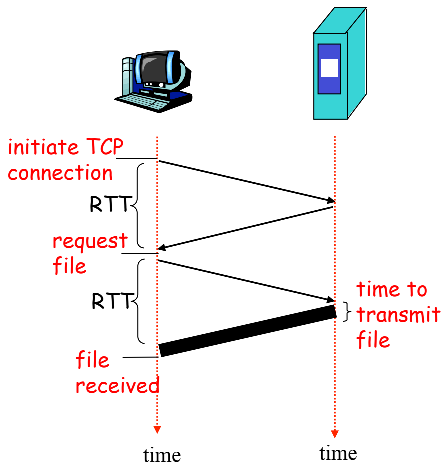
Non-Persistent HTTP: Response time

Definition of RTT: time for 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+transmit time



Persistent HTTP

Nonpersistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

HTTP/1.1 also supports pipelining of requests

Persistent HTTP

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

HTTP request message

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

request line (GET, POST, HEAD commands)

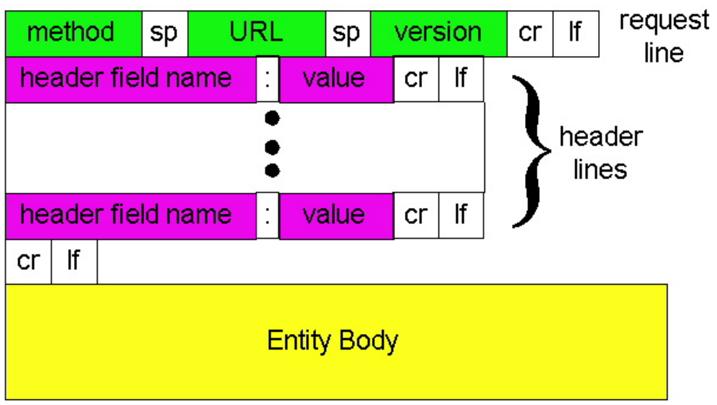
header lines

```
GET /index.html HTTP/1.1
Host: www.ucc.ie
User-agent: Mozilla/4.0
Connection: close
Accept-language:en
```

Carriage return, line feed indicates end of message

(extra carriage return, line feed)

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

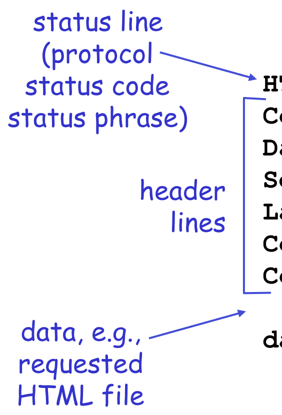
HTTP/1.0

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

HTTP/1.1

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

HTTP response message




```
HTTP/1.1 200 OK
Connection close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```


HTTP response status codes

In first line in server->client response message.
A few 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 (Location:)

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 www.cs.ucc.ie 80
```

Opens TCP connection to port 80 (default HTTP server port) at www.cs.ucc.ie
Anything typed in sent to port 80 at www.cs.ucc.ie

2. Type in a GET HTTP request:

```
GET /~cjs/ HTTP/1.1
Host: www.cs.ucc.ie
```

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!

User-server state: cookies

Many major Web sites use cookies

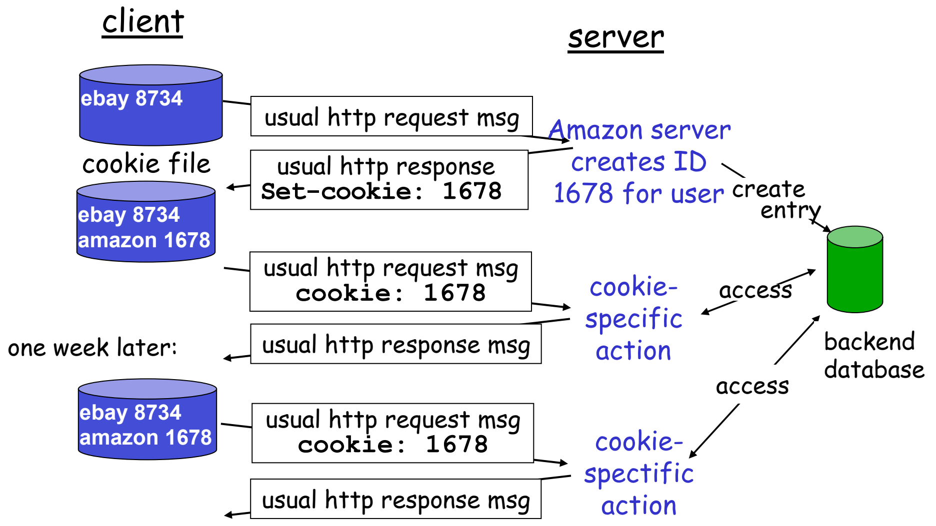
Four components:

1) cookie header line of HTTP *response* message
2) cookie header line in 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 always access Internet always from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping "state" (cont.)



CS2505: Application Layer 39

Cookies (continued)

What cookies can bring:

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

aside

Cookies and privacy:

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

How to keep "state":

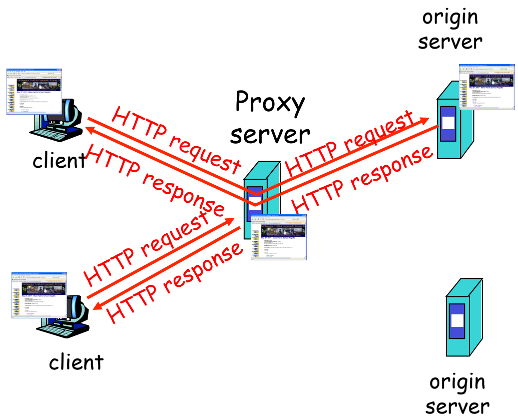
- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

CS2505: Application Layer 40

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
- typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

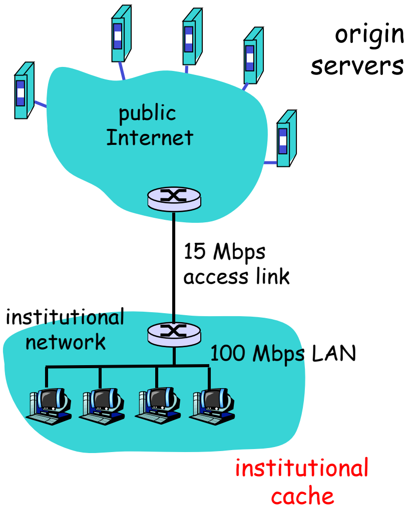
- reduce response time for client request
- reduce traffic on an institution's access link.
- Internet dense with caches: enables "poor" content providers to effectively deliver content (but so does P2P file sharing)

Caching example

Assumptions
- average object size = 1,000,000 bits
- avg. request rate from institution's browsers to origin servers = 15/sec
- delay from institutional router to any origin server and back to router = 2 sec

Consequences
- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay

= 2 sec + minutes + milliseconds

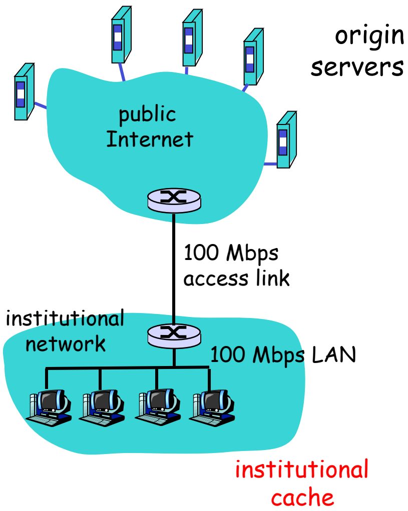


Caching example (cont)

possible solution
- increase bandwidth of access link to, say, 100 Mbps

consequence
- utilization on LAN = 15%
- utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay

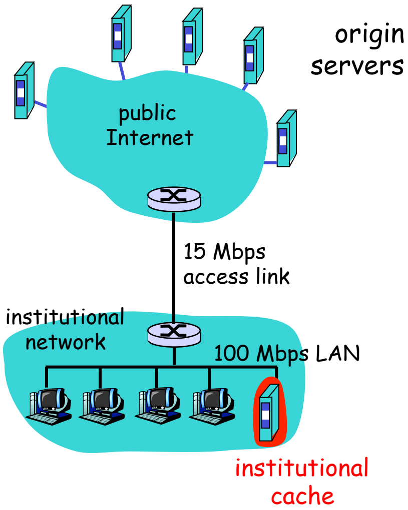
= 2 sec + msecs + msecs
- often a costly upgrade



Caching example (cont)

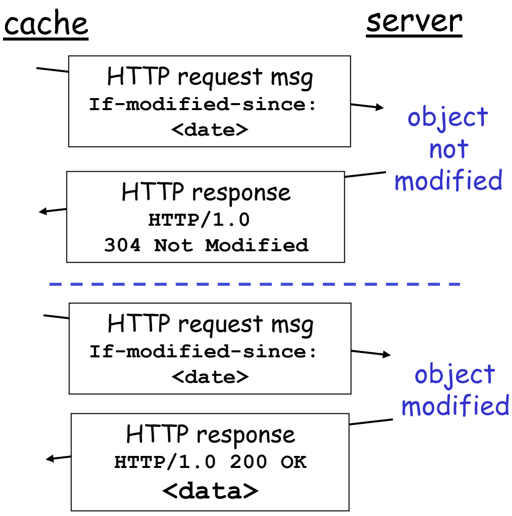
possible solution: install cache
- suppose hit rate is 0.4

consequence
- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
- total avg delay = Internet delay + access delay + LAN delay = .6*(2.01) secs + .4*milliseconds < 1.4 secs



Conditional GET

- Goal: don't send object if cache has up-to-date cached version
- cache: specify date of cached copy in HTTP request
  `If-modified-since: <date>`
- server: response contains no object if cached copy is up-to-date:
  `HTTP/1.0 304 Not Modified`



HTTP/2

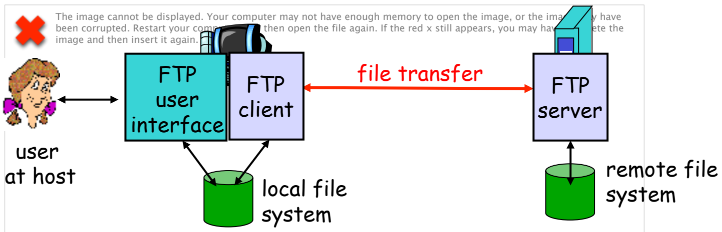
Image from akamai.com

- Latest version of HTTP released in 2015:
  - Available in most browsers; server use growing
  - Derived from Google's SPDY
  - Backwards compatible with HTTP/1.1
- Main goal is to reduce page load times:
  - Binary encoding and header compression
  - Server can "push" content that it believes the client will need, e.g. inter-dependent objects
  - Eliminates head-of-the-line blocking from HTTP/1.1 for pipelining by allowing asynchronous request/response
- Concept of multiplexing prioritised "streams" of requests/responses over a HTTP/2 session

Section 2: Application layer

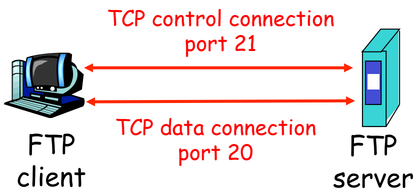
- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
- 2.5 DNS
- 2.6 File Distribution

FTP: the file transfer protocol



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

FTP: separate control, data connections



- FTP client contacts FTP server at port 21, TCP is transport protocol
- client authorized over control connection
- client browses remote directory by sending commands over control connection.
- when server receives file transfer command, server opens 2*nd* TCP connection (for file) to client
- after transferring one file, server closes data connection.

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

FTP commands, responses

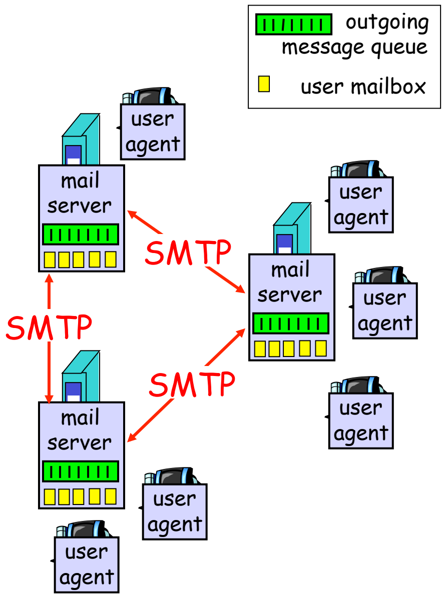
Sample commands:
- sent as ASCII text over control channel
- `USER` *`username`*
- `PASS` *`password`*
- `LIST` return list of file in current directory
- `RETR filename` retrieves (gets) file
- `STOR filename` stores (puts) file onto remote host

Sample return codes
- status code and phrase (as in HTTP)
- `331 Username OK, password required`
- `125 data connection already open; transfer starting`
- `425 Can't open data connection`
- `452 Error writing file`

Section 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 File Distribution

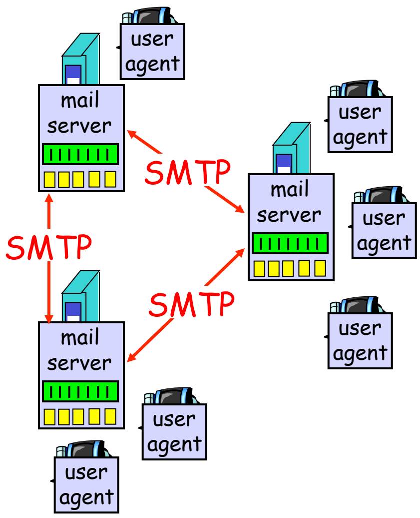
Electronic Mail



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., Eudora, Outlook, elm, Mozilla Thunderbird
- outgoing, incoming messages stored on server

Electronic Mail: mail servers



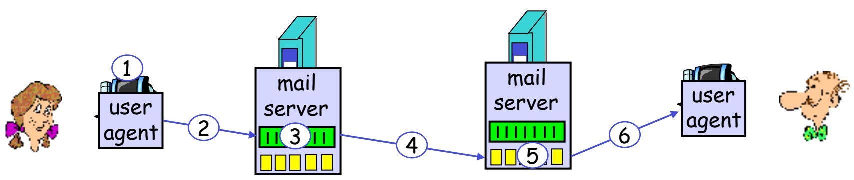
Mail Servers
- mailbox contains incoming messages for user
- message queue of outgoing (to be sent) mail messages
- SMTP protocol between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server

Electronic Mail: SMTP [RFC 2821]

- 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 by default (MIME extension to handle other data types)

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message and "to" `bob@somesuni.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 someuni.edu
C: HELO yahoo.com
S: 250  Hello yahoo.com, pleased to meet you
C: MAIL FROM: <alice@yahoo.com>
S: 250 alice@yahoo.com... Sender ok
C: RCPT TO: <bob@someuni.edu>
S: 250 bob@someuni.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: How is the weather there?
C: It's like a hurricane here today.
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 someuni.edu closing connection
```


Try SMTP interaction for yourself:

- **`telnet servername 25`**
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

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

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

Comparison with HTTP:

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

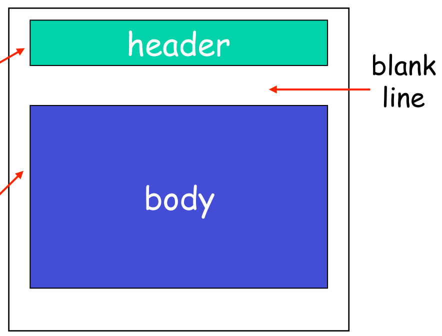
Mail message format

SMTP: protocol for exchanging email msgs

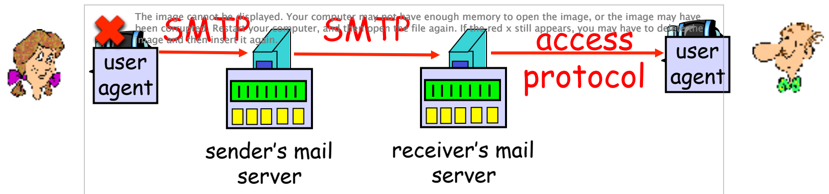
RFC 822: standard for text message format:

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

  different from SMTP commands!
- body
  - the "message"



Mail access protocols



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

POP3 protocol

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

transaction phase, client:
- **list**: list message numbers
- **retr**: retrieve message by number
- **dele**: delete
- **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 1 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
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name

Section 2: Application layer

DNS: Domain Name System

People: many identifiers:
- name, passport #

Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- Host "name", e.g., ww.yahoo.com - used by humans

Q: map between host name and its IP address?

Domain Name System:
- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network's "edge"

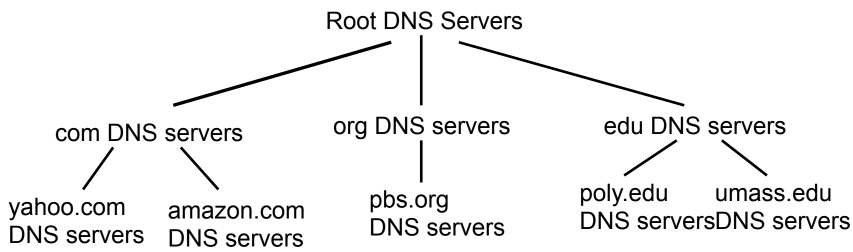
DNS

DNS services
- hostname to IP address translation
- host aliasing
  - Canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance

doesn't *scale!*

Distributed, Hierarchical Database

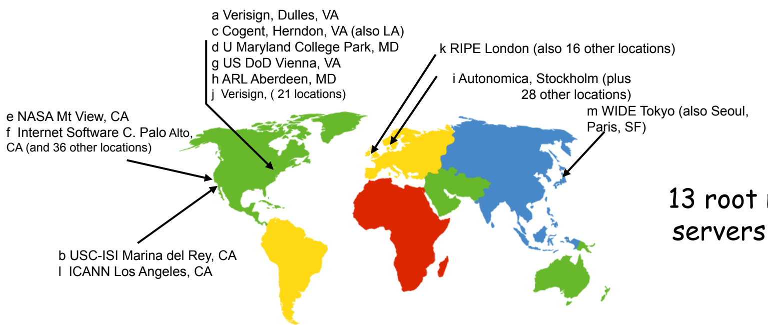


Client wants IP for www.amazon.com; 1st approx:

- 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

DNS: Root name servers

- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server



13 root name servers worldwide

TLD and Authoritative Servers

- Top-level domain (TLD) servers:
  - responsible for com, org, net, edu, etc, and all top-level country domains ie, uk, fr, ca, jp.
  - Network Solutions maintains servers for com TLD
  - Educause for edu TLD
- Authoritative DNS servers:
  - organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web, mail).
  - can be maintained by organization or service provider

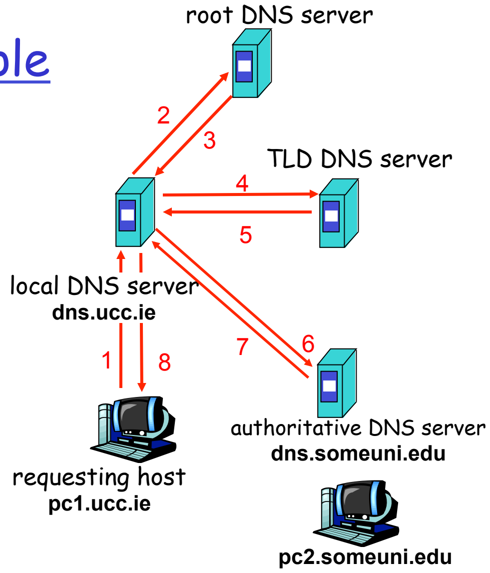
Local Name Server

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one.
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - acts as proxy, forwards query into hierarchy

DNS name resolution example

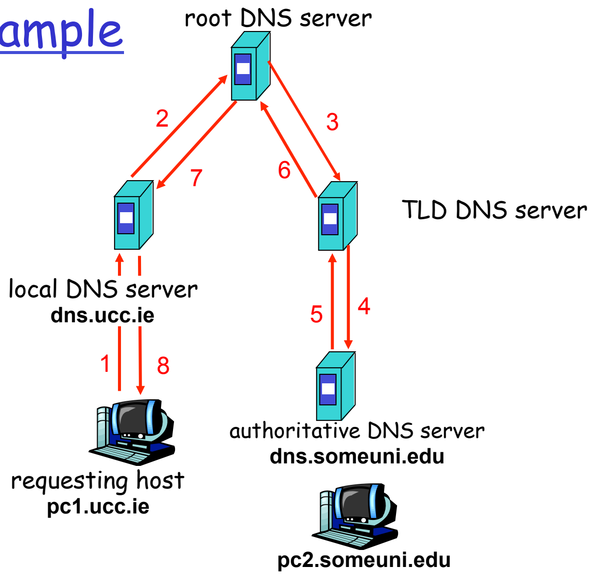
- Host at pc1.ucc.ie wants IP address for pc2.someuni.edu

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?



DNS: caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time
  - TLD servers typically cached in local name servers
    - Thus root name servers not often visited
- update/notify mechanisms under design by IETF
  - RFC 2136
  - http://www.ietf.org/html.charters/dnsind-charter.html

DNS records

DNS: distributed db storing resource records (RR)

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

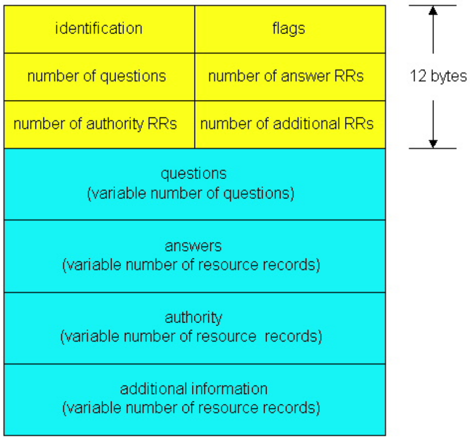
- Type=A
  - **name** is hostname
  - **value** is IP address
- Type=NS
  - **name** is domain (e.g. foo.com)
  - **value** is hostname of authoritative name server for this domain
- Type=CNAME
  - **name** is alias name for some "canonical" (the real) name
    `www.ibm.com` is really `servereast.backup2.ibm.com`
  - **value** is canonical name
- Type=MX
  - **value** is name of mailserver associated with **name**

DNS protocol, messages

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

msg header

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



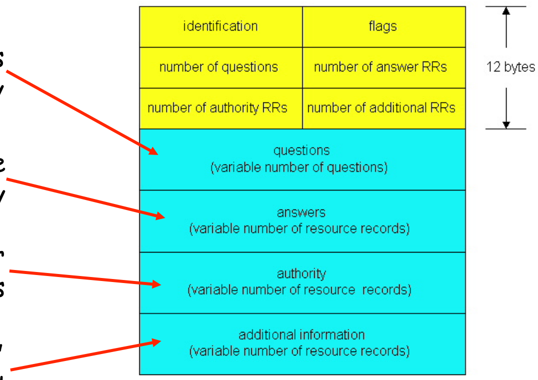
DNS protocol, messages

Name, type fields for a query

RRs in response to query

records for authoritative servers

additional "helpful" info that may be used



Inserting records into DNS

- example: new startup "Network Utopia"
- register name networkuptopia.com at *DNS* *registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts two RRs into com TLD server:

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

- create authoritative server Type A record for www.networkuptopia.com; Type MX record for networkutopia.com
- How do people get IP address of your Web site?

Section 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 File Distribution
  - Peer-to-Peer (P2P), Content Distribution Networks (CDN), Streaming Video

Problem of File Distribution

- A common requirement is to use the Internet to distribute files to end-users
  - music, video, movies, binaries, software patches, web objects, etc
- We need mechanisms that can do this in a scalable manner
  - Scalable in the number of requesting users
  - Scalable in the number of requested files
- Issues are
  - Response time to end-user for delivery
  - Network congestion
  - Server load

File Distribution: Mirror Sites

- Client-Server model but using *mirror* sites
  - The file is replicated across a set of servers (each known as a mirror)
  - User selects a mirror, typically based on geographical proximity and/or an indication of how busy the mirror is
- Issues
  - Need to ensure consistency of file replicas on each mirror
  - Users may converge on just a few mirrors, eg especially in highly populated areas

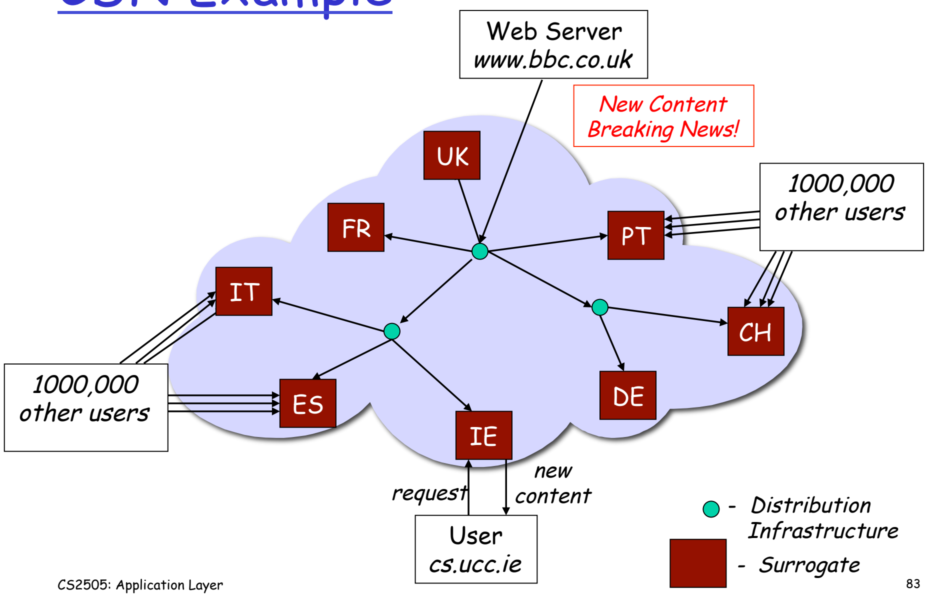
File Distribution: CDNs

- CDN = Content Distribution Network
  - An *automated* approach to the use of mirrors
  - Use of CDN is *transparent* to end-users
- CDN composed of large number of servers
  - Organised as an *overlay* network
  - Distributed globally, strategically placed, often at edge of backbone network
  - Each server keeps replicas of popular files
  - CDN management system responsible for deciding how many replicas and in which locations (based on expected/actual usage)

File Distribution: CDNs

- Overlay network to distribute content from origin servers to users
- Avoids large amounts of same data repeatedly traversing potentially congested links on the Internet
- Reduces Web server load
- Reduces user perceived latency
- Tries to route around congested networks

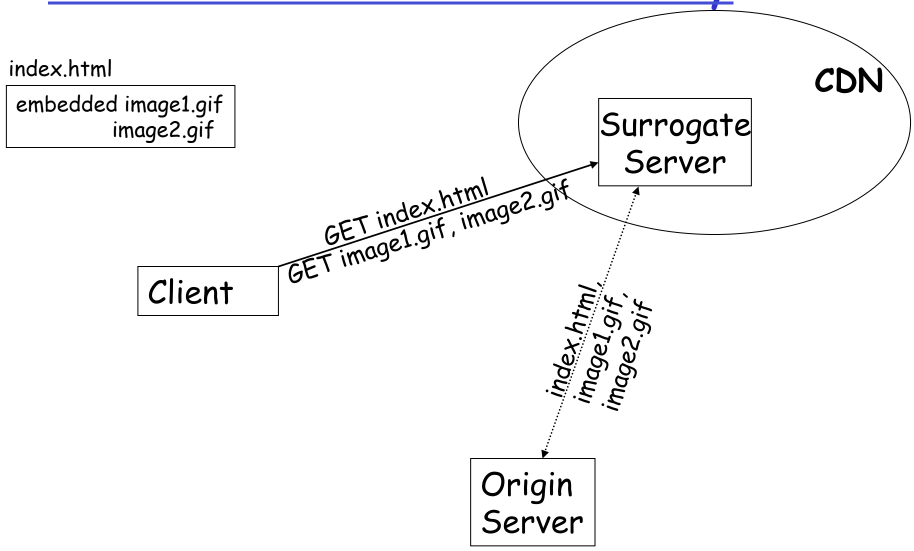
CDN Example



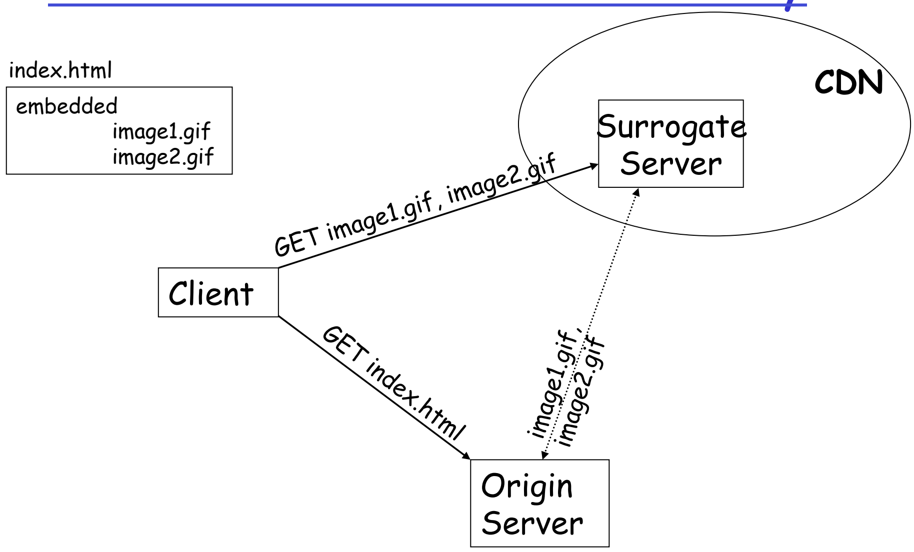
File Distribution: CDNs

- Issues
  - High costs for having many servers
  - How to direct clients to best server?
    - How to define best? Answer:__________
  - Redirection mechanism
    - Often using DNS, eg send request to *www.xyz.com* but response is to *www.xyz.ie*
    - Or HTML rewriting, where the embedded URLs are changed dynamically to include the target server
- Example is Akamai, with tens of thousands of servers worldwide

CDN with Full Site Delivery



CDN with Partial Site Delivery

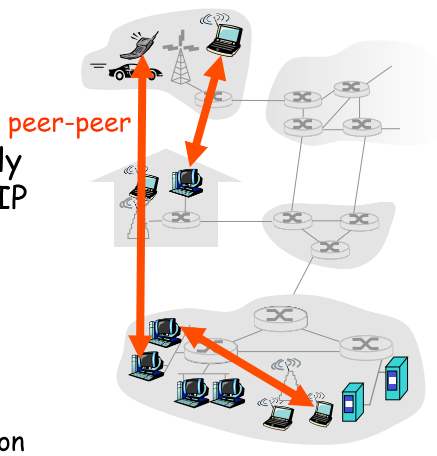


CDN -versus- Caches

- CDN is proactive while a cache is reactive
- Different purposes
  - Caches to reduce network traffic
  - CDN to improve delivery of content
- Different "customer"
  - CDN caters to content providers (web servers)
  - Cache caters to web users

Pure P2P architecture

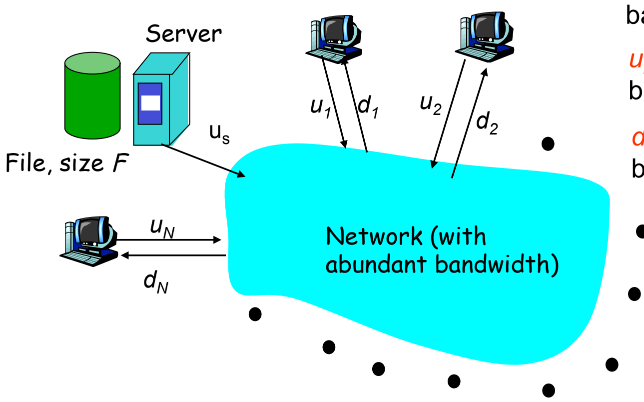
- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses



- Three topics:
  - File distribution performance
  - Searching for information
  - Case Study: Skype

File Distribution: Server-Client vs P2P

Question : How much time to distribute file from one server to N *peers*?



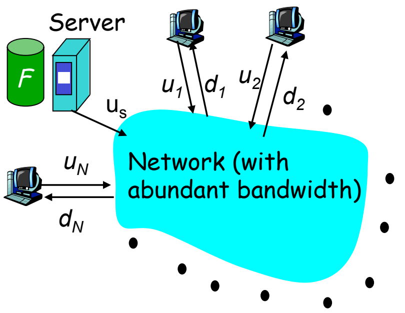
u_s: server upload bandwidth

u_i: peer i upload bandwidth

d_i: peer i download bandwidth

File distribution time: server-client

- server sequentially sends N copies:
  - NF/u_s time
- client i takes F/d_i time to download

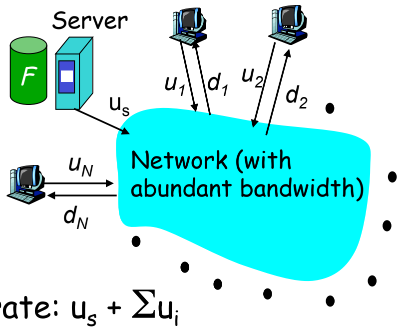


Time to distribute F to N clients using client/server approach $= d_{cs} = \max \{ NF/u_s, F/\min_i(d_i) \}$

increases linearly in N (for large N)

File distribution time: P2P

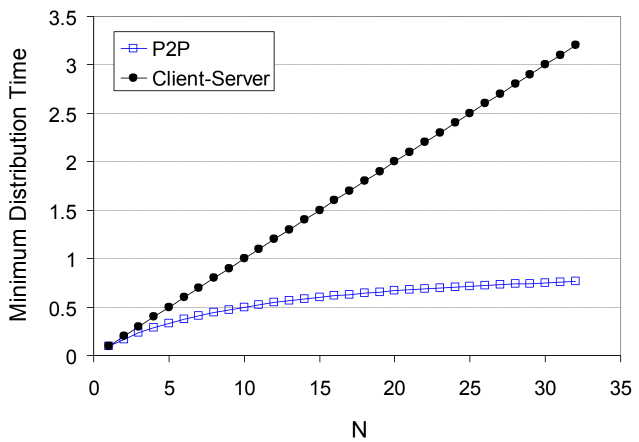
- server must send one copy: F/u_s time
- client i takes F/d_i time to download
- NF bits must be downloaded (aggregate)
  - fastest possible upload rate: $u_s + \Sigma u_i$



$$d_{P2P} = \max \{ F/u_s, F/\min_i(d_i), NF/(u_s + \Sigma u_i) \}$$

Server-client vs. P2P: example

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



BitTorrent

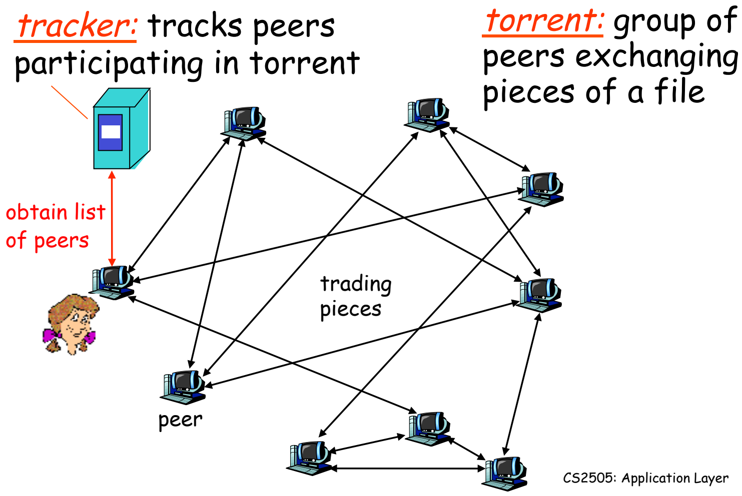
- Very popular P2P protocol written by Bram Cohen (in Python) in 2001
  - Lots of clients available for all platforms
- Nodes organise as an overlay network and volunteer to participate in distributing a file
- "Pull-based" "swarming" approach
  - Each file split into smaller pieces
  - Nodes request desired pieces from neighbours (peers)
  - Pieces not downloaded in sequential order
- Encourages contribution by all nodes

BitTorrent

- A user wishing to distribute a file X creates a small torrent descriptor file Xt and distributes it via web, email, etc
- Then make actual file X available on node known as the *seed*
- Users wanting file X pass the torrent descriptor Xt to their client which then contacts the seed and/or peers
- Over time there is less dependence on the seed, thus providing scalability

BitTorrent

P2P file distribution



BitTorrent

- file divided into fixed size *pieces*, typically 256KB
- peer joining torrent:
  - has no pieces, but will accumulate them over time
  - registers with tracker to get list of peers, connects to subset of peers ("neighbours")
- while downloading, peer uploads pieces to other peers
- peers may come and go (known as *churn*)
- once peer has entire file, it may (selfishly) leave, or (altruistically) remain (as an additional seed)

BitTorrent

Pulling Pieces

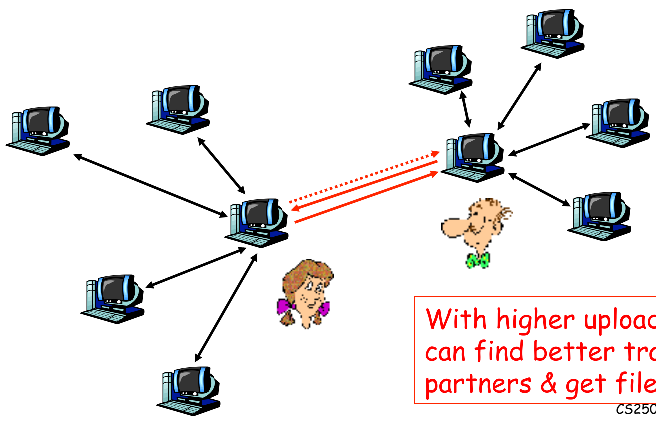
- at any given time, different peers have different subsets of file pieces
- periodically, a peer (Alice) asks each neighbour for list of pieces that they have
- Alice sends requests for her missing pieces
  - rarest first

Sending Pieces: tit-for-tat

- Alice sends pieces to small subset of neighbours currently sending her pieces *at the highest rate*
  - re-evaluate periodically, eg every 10 secs
- every 30 secs: randomly select another peer, starts sending pieces
  - newly chosen peer may join top subset
  - "optimistically unchoke"
  - Why bother?

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



With higher upload rate, can find better trading partners & get file faster!

Distributed Tracking

- Problem with use of a Tracker
  - Vulnerability to single node
  - Privacy concerns (Tracker knows all peers in torrent)
- Use a distributed tracker – a distributed P2P database
  - Database has (key, value) pairs, e.g. (content name, IP address)
  - Implemented as a Distributed Hash Table (DHT) to make queries scalable when the DB is spread over large numbers of Peers

Distributed Hash Table (DHT)

- Distribute (key, value) pairs over millions of peers
  - Ideally spread evenly
- Any peer can query database with a key
  - database returns values matching that key
- Each peer may only knows about a small number of other peers
- Robust to peers coming and going (churn)
- **Peers can also insert (key, value) pairs**

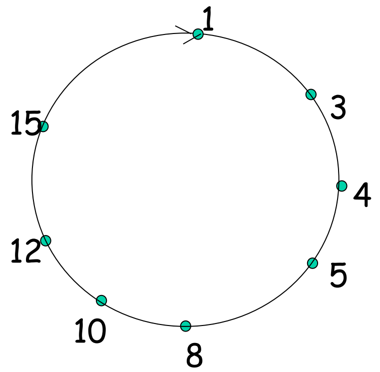
DHT Identifiers

- More convenient to store and search on numerical representation of key
- To get integer keys, hash original key.
  - eg, key = h("Led Zeppelin IV")
- Assign integer identifier to each peer in range $[0,2^n-1]$.
  - Each identifier can be represented by n bits.
- Require each key to be an integer in same range.
- Use hash to map a key to a Peer which will store the details of where to find the content

How to assign keys to peers?

- Central issue:
  - Assigning (key, value) pairs to peers.
- Rule: assign key to the peer that has the closest ID.
- Convention in lecture: closest is the immediate successor of the key.
- Ex: n=4; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

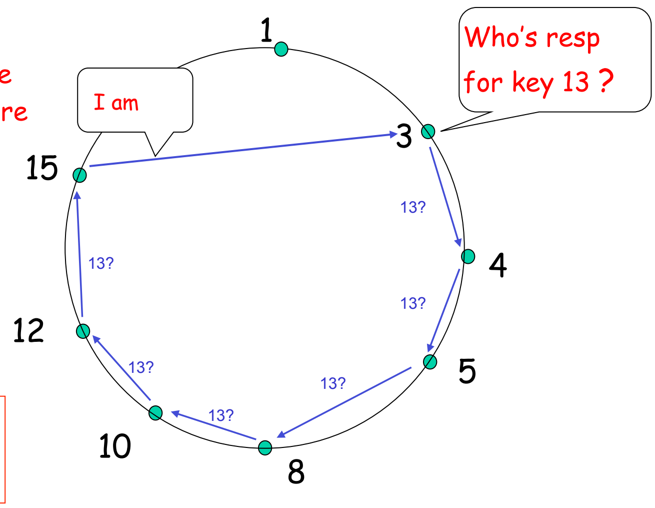
Circular DHT (1)



- Each peer *only* aware of immediate successor and predecessor.
- "Overlay network"

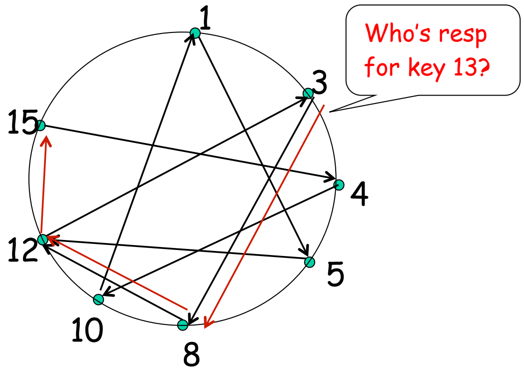
Circular DHT (2)

O(N) messages on avg to resolve query, when there are N peers



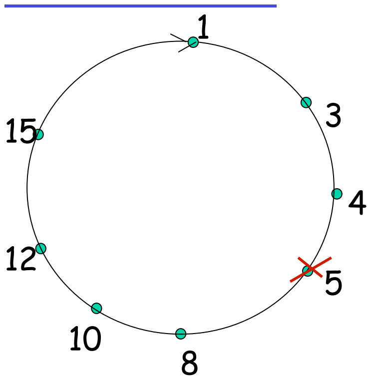
Define closest as closest successor

Circular DHT with Shortcuts



- Each peer keeps track of IP addresses of predecessor, successor, short cuts.
- Reduced from 6 to 3 messages.
- Possible to design shortcuts so O(log N) neighbours, O(log N) messages in query

Peer Churn

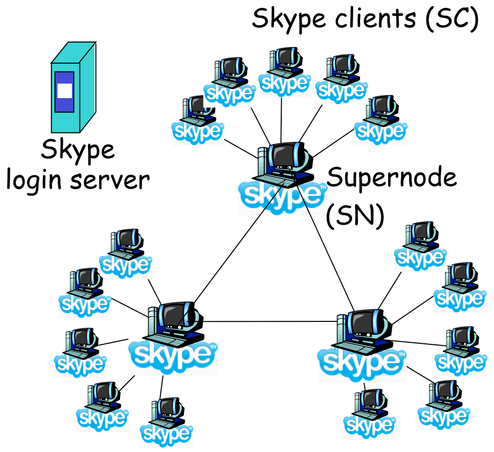


- To handle peer churn, require each peer to know the IP address of its two successors.
- Each peer periodically pings its two successors to see if they are still alive.

- Peer 5 abruptly leaves
- Peer 4 detects; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8's immediate successor its second successor.
- What if peer 13 wants to join?

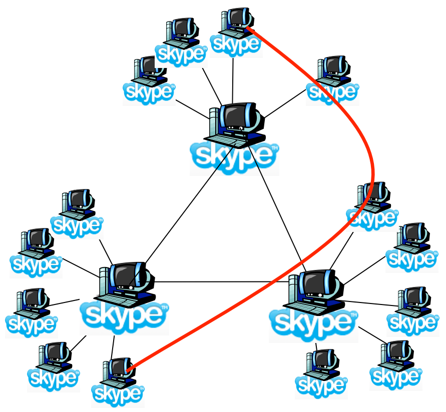
P2P Case study: Skype

- inherently P2P: pairs of users communicate.
- proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay with SNs
- Index maps usernames to IP addresses; distributed over SNs



Peers as relays

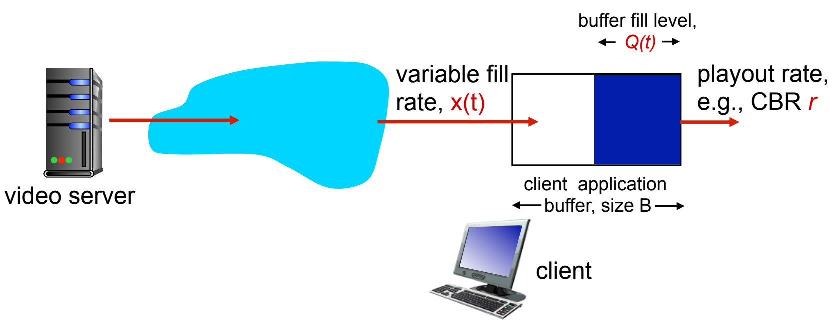
- Problem when both Alice and Bob are behind "NATs".
  - NAT prevents an outside peer from initiating a call to insider peer
- Solution:
  - Using Alice's and Bob's SNs, Relay is chosen
  - Each peer initiates session with relay.
  - Peers can now communicate through NATs via relay



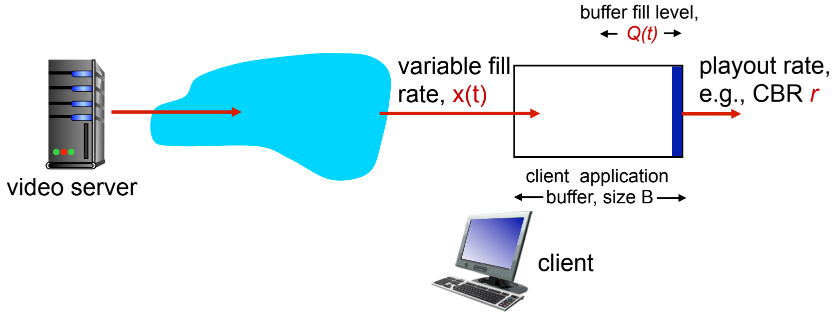
Streaming stored video

- Video is the dominant traffic type on today's fixed and mobile networks
- Video files usually too large to download fully, so instead they are streamed at a rate that matches the playback rate
- *continuous playout constraint*: once client playout begins, playback must match original timing
  - … but *network delays are variable* (jitter), so will need *client-side buffer* to match playout requirements

Client-side buffering, playout

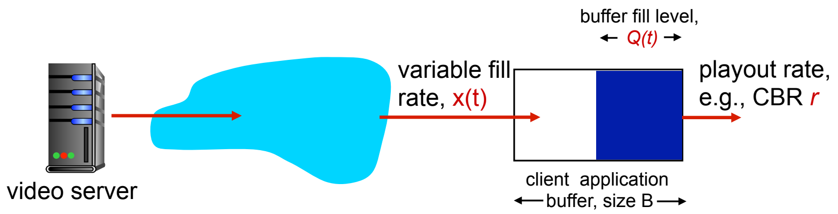


Client-side buffering, playout



1. Initial fill of buffer until playout begins at t_p
2. playout begins at t_p,
3. buffer fill level varies over time as fill rate *x(t)* varies and playout rate *r* is constant

Client-side buffering, playout

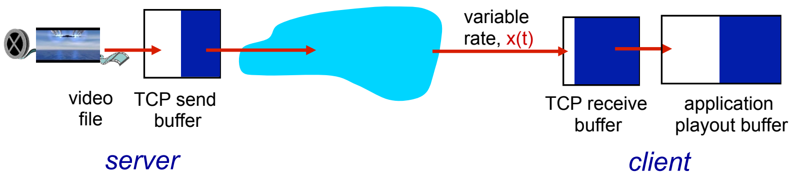


playout buffering: average fill rate ($\bar{x}$), playout rate (r):

- $\bar{x} < r$: buffer eventually empties (causing freezing of video playout until buffer again fills)
- $\bar{x} > r$: buffer will not empty, provided initial playout delay is large enough to absorb variability in x(t)
  - *initial playout delay tradeoff:* buffer starvation less likely with larger delay, but larger delay until user begins watching

Streaming multimedia: HTTP

- multimedia file retrieved via HTTP GET
- send at maximum possible rate under TCP



- fill rate fluctuates due to variable network delivery delays
- Use larger playout delay: to smooth delivery rate
- HTTP/TCP passes more easily through firewalls

Streaming multimedia: DASH

- *DASH: D*ynamic, *A*daptive *S*treaming over *H*TTP
- *server:*
  - divides video file into multiple chunks
  - each chunk stored, encoded at different rates
  - *manifest file:* provides URLs for different chunks
- *client:*
  - periodically measures server-to-client bandwidth
  - consulting manifest, requests one chunk at a time
    - chooses maximum coding rate sustainable given current bandwidth
    - can choose different coding rates at different points in time (depending on available bandwidth at time)

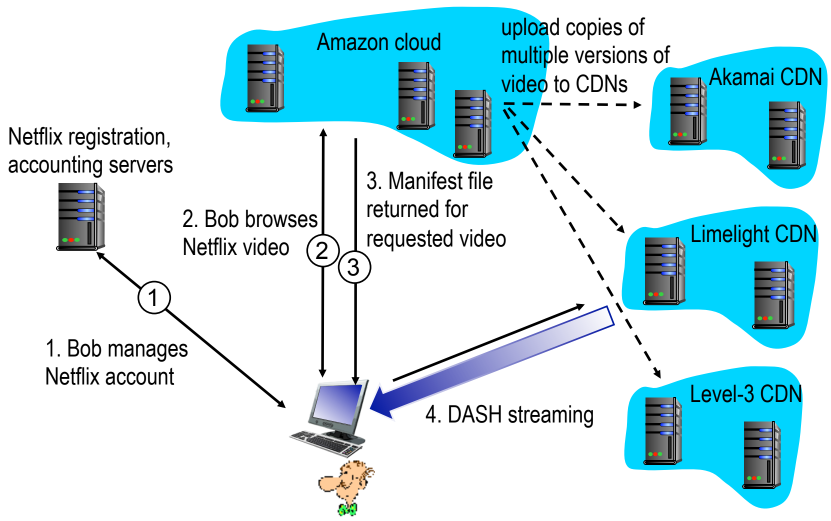
Streaming multimedia: DASH

- *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)

Case study: Netflix

- 35% downstream peak-hours US traffic (2014)
- owns very little infrastructure, uses 3rd party services:
  - own registration, payment servers
  - Amazon (3rd party) cloud services:
    - Netflix uploads studio master to Amazon cloud
    - create multiple version of movie (different endodings) in cloud
    - upload versions from cloud to CDNs
    - Cloud hosts Netflix web pages for user browsing
  - *three* 3rd party CDNs host/stream Netflix content: Akamai, Limelight, Level-3

Case study: Netflix



Section 2: Summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
  - hybrid
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, Skype,DASH

Section 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 being communicated

Important themes:

- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"