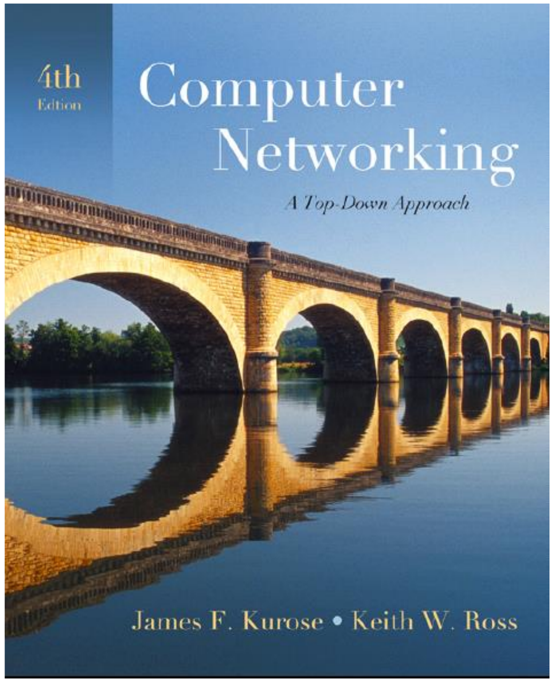
Chapter 2
Application Layer



Computer Networking: A Top Down Approach,
Jim Kurose, Keith Ross
Addison-Wesley.

Chapter 2: Application layer

Chapter 2: Application Layer

Our goals:

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

Some network apps

- e-mail
- web
- instant messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video clips
- voice over IP
- real-time video conferencing
- grid computing
- …
-
-

Note: different applications may have different

- Requirements (delay, loss, Tput, jitter bounds, security)
- Number of participants (unicast, multicast, broadcast, manycast, profilecast)
- Architecture (client-server, p2p, flat, hierarchical, hybrid, self-configuring)

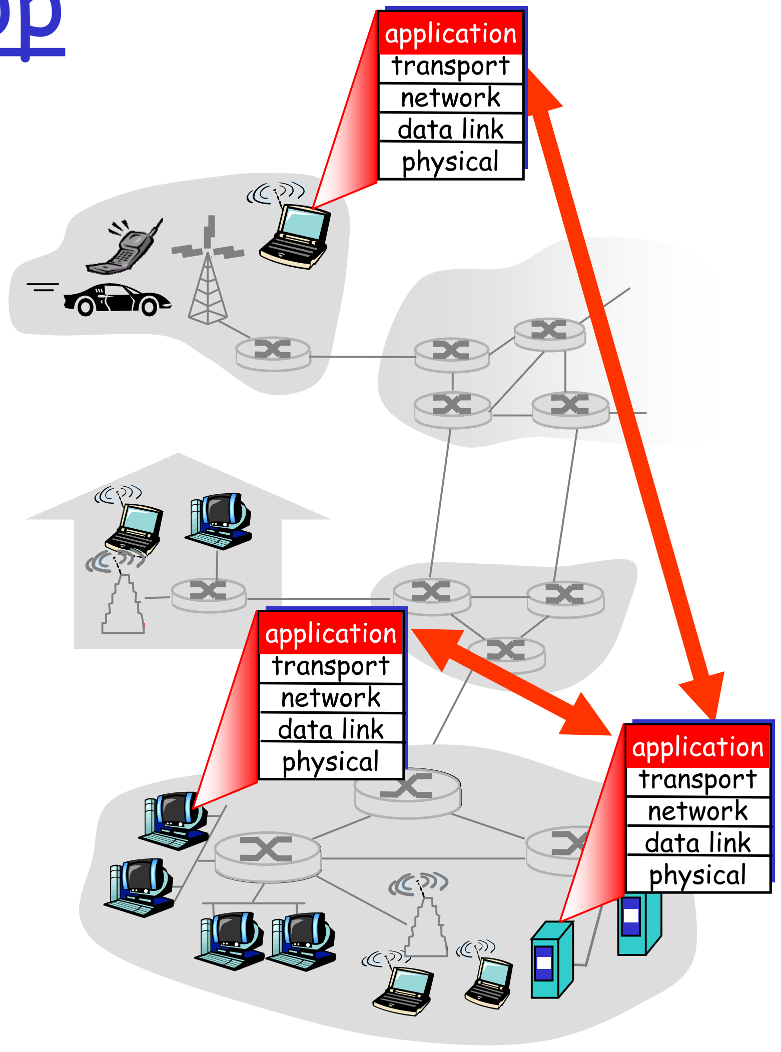
Creating a network app

Application programs

- run on *end systems*
- communicate over network

little software written for devices in network core

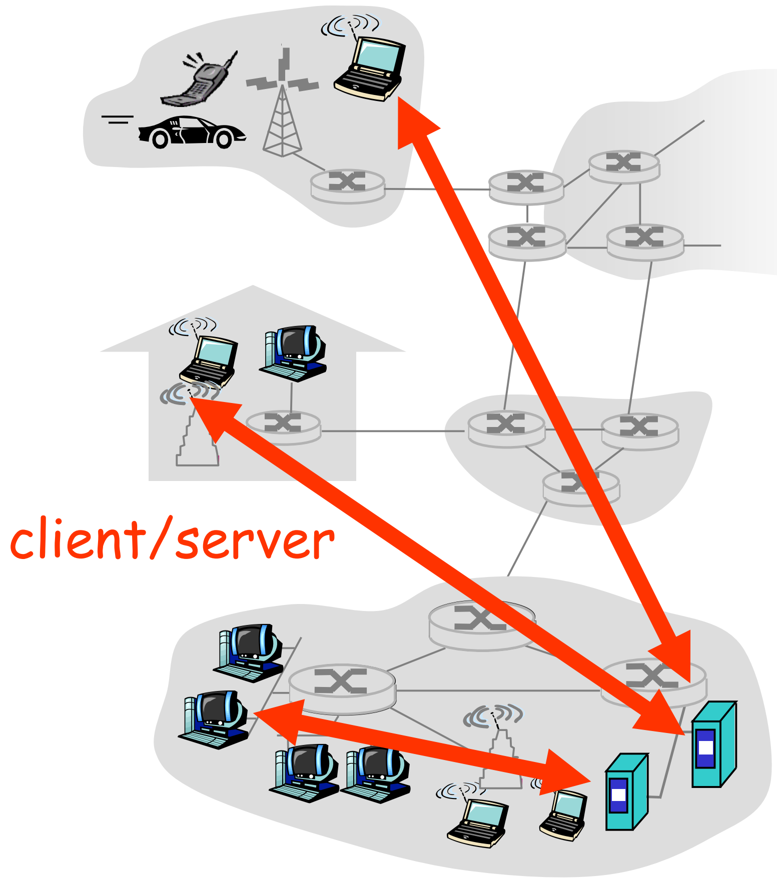
- network core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



Application architectures

- Client-server
- Peer-to-peer (P2P)
- Hybrid of client-server and P2P

Client-server architecture



server:

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

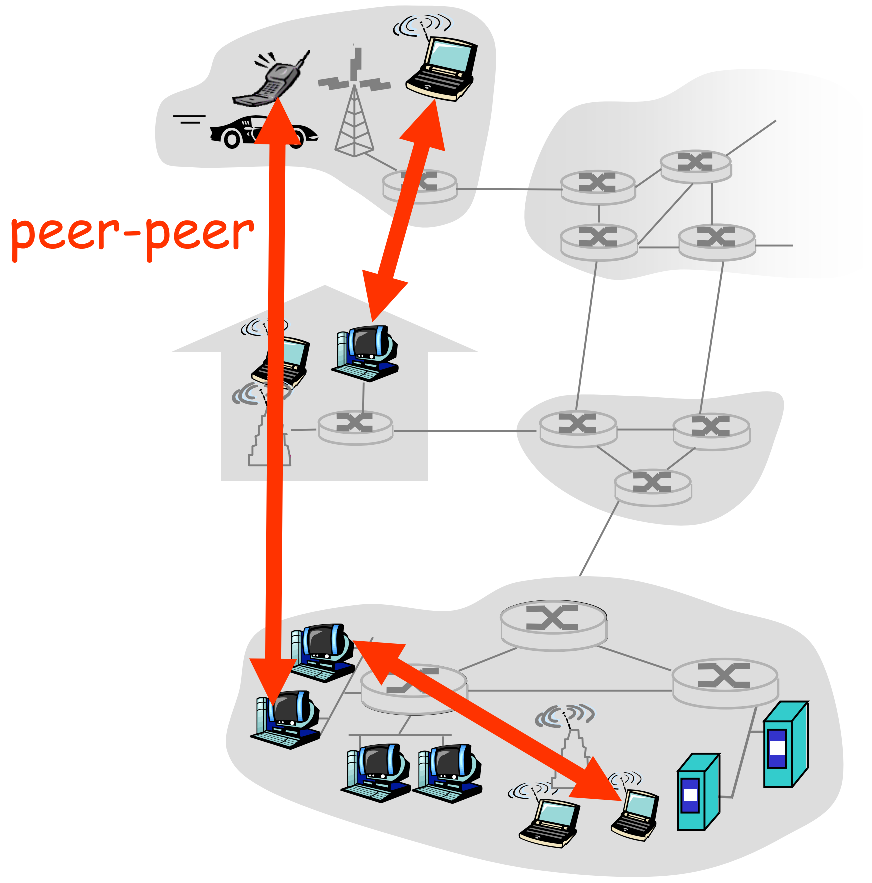
Clients (in general):

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

Pure P2P architecture

- No 'always-on' server
- arbitrary end systems directly communicate
- peers intermittently connected & change IP addresses
- example: Gnutella

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 detection & location
    - user registers IP address with central server
    - uses central server to find 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

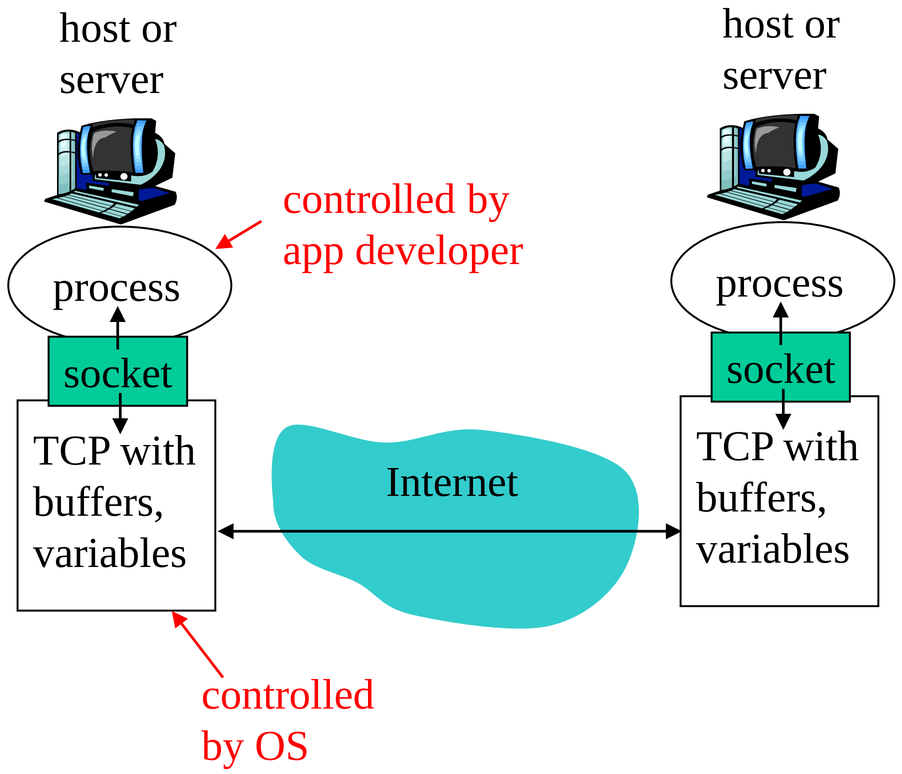
Client process: process that initiates communication

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


host or server
host or server
controlled by app developer
process
process
socket
socket
TCP with buffers, variables
TCP with buffers, variables
Internet
controlled by OS

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

Addressing processes

- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: No, *many* processes can runn on same host
- *identifier* includes IP address & port numbers associated with process on host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - Port number: 80
- more later…

What transport service does an app need?

Data loss

- some apps (e.g., audio) can tolerate some loss (~10%)
- others (e.g., FTP, telnet) require 100% delivery

Timing

- some apps (e.g., VoIP, interactive games) require low (bounded) delay (and/or jitter) to be "effective"
  - Example: VoIP jitter or handoff delay bound is ~200ms
- others (e.g., FTP) are tolerant to some delay/jitter
  - Some multimedia apps use buffering & playback point adjustment

Bandwidth

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

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	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

Internet transport protocols services

TCP service:

- *connection-oriented:* setup required between client and server processes
- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum bandwidth guarantees, multicast support

UDP service:

- -

Internet apps: application, transport protocols

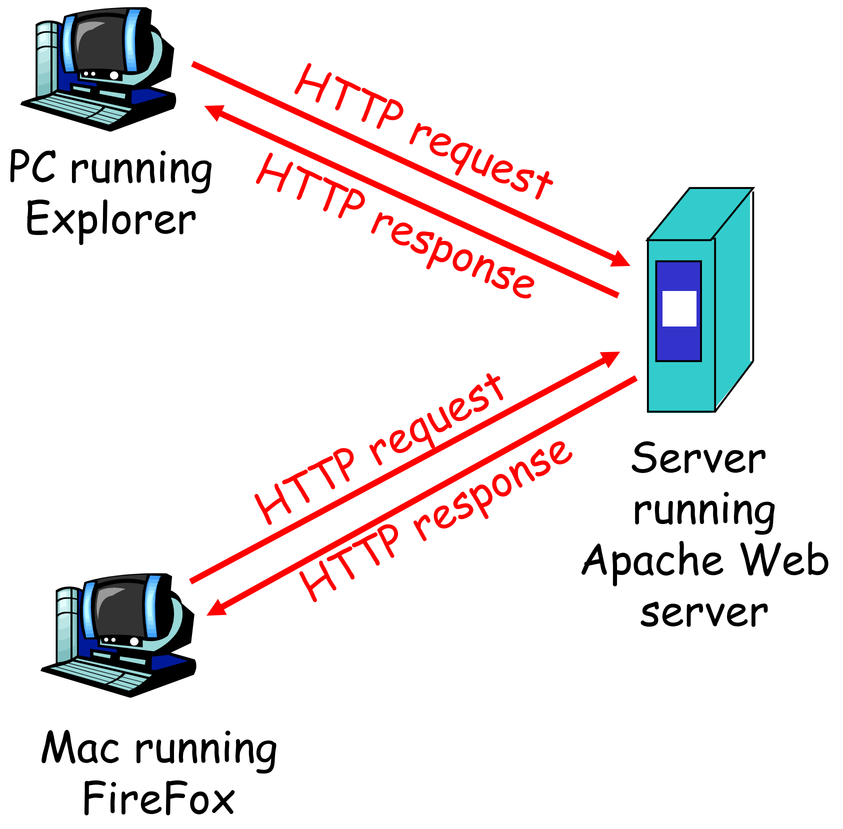
Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Vonage,Dialpad)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

Chapter 2: Application layer

HTTP overview

HTTP: hypertext transfer protocol

- client/server model
  - *client:* browser to request & receive Web objects
  - *server:* Web server sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068 (persistent TCP)



HTTP overview (continued)

Uses TCP:

- 1. client initiates TCP connection to server,port 80
- 2. server accepts TCP connection from client
- 3. HTTP (application-layer) messages exchanged between HTTP client and HTTP server
- 4. TCP connection closed

A 'state' is information kept in memory of a host, server or router to reflect past events: such as routing tables, data structures or database entries

HTTP is "stateless"

- server maintains no information about past client requests

Protocols that maintain "state" are complex!

- history (state) is maintained
- if server/client crashes, views of "state" may be inconsistent, must be reconciled
- state is added via 'cookies'

Design Issues:

- Stateful vs Stateless
- Hard State vs Soft State

HTTP connections

I. Nonpersistent HTTP

- At most one object is sent over a TCP connection. Used in HTTP/1.0

II. Persistent HTTP

- Multiple objects can be sent over single TCP connection. Used in HTTP/1.1 by default:
  - A. persistent with pipelining
  - B. persistent without pipelining

I. Nonpersistent HTTP

User enters URL

`someSchool.edu/someDepartment/home.index`

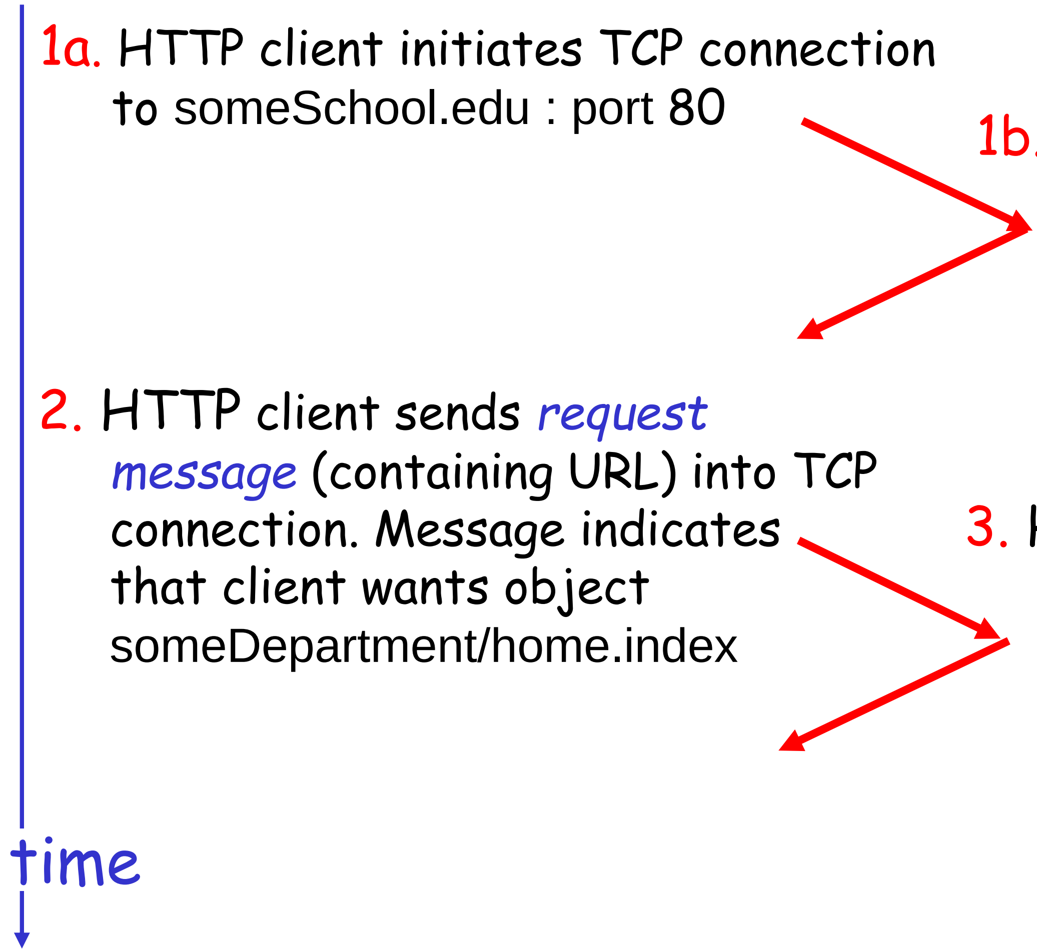
(contains text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to someSchool.edu : port 80

1b. HTTP server @ someSchool.edu port 80. "accepts" connection, notifying client

2. HTTP client sends *request message* (containing URL) into TCP connection. Message indicates that client wants object someDepartment/home.index

3. HTTP server receives request message, sends *response message* containing requested object



I. 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

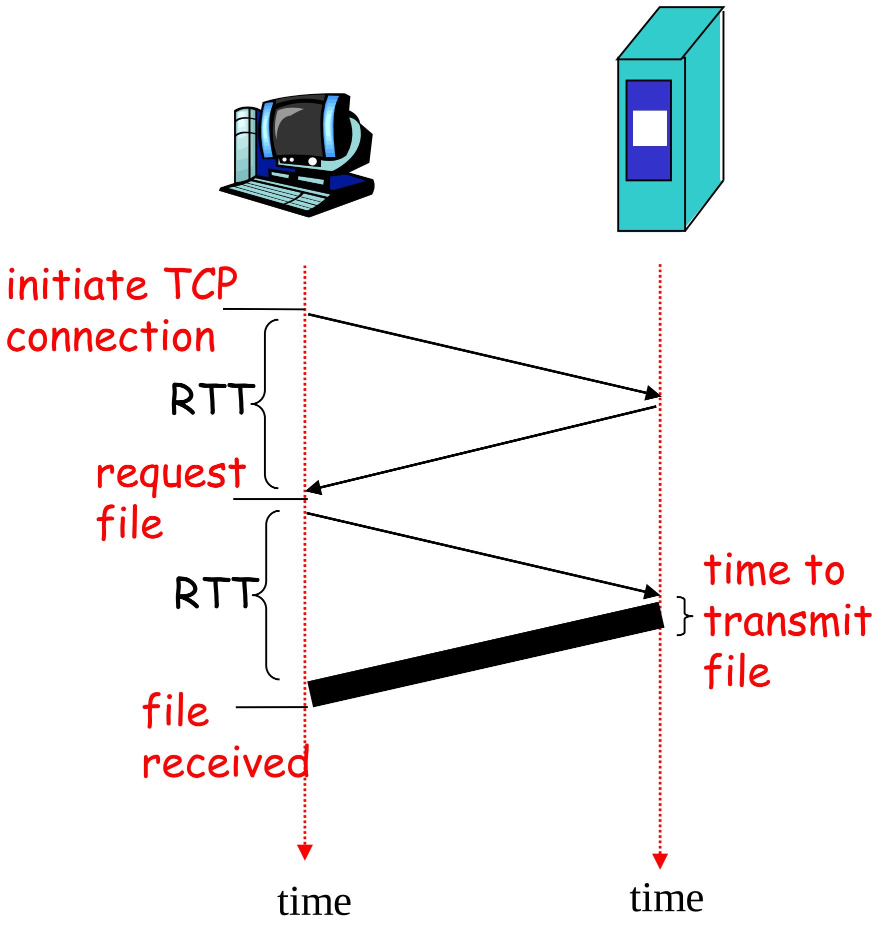
I. Non-Persistent HTTP: Response time

Definition of RTT: time to send request 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

I. Nonpersistent HTTP issues:

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

II. Persistent HTTP

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection

A. Persistent ***without*** pipelining:

- client issues new request only when previous response has been received
- one RTT for each referenced object

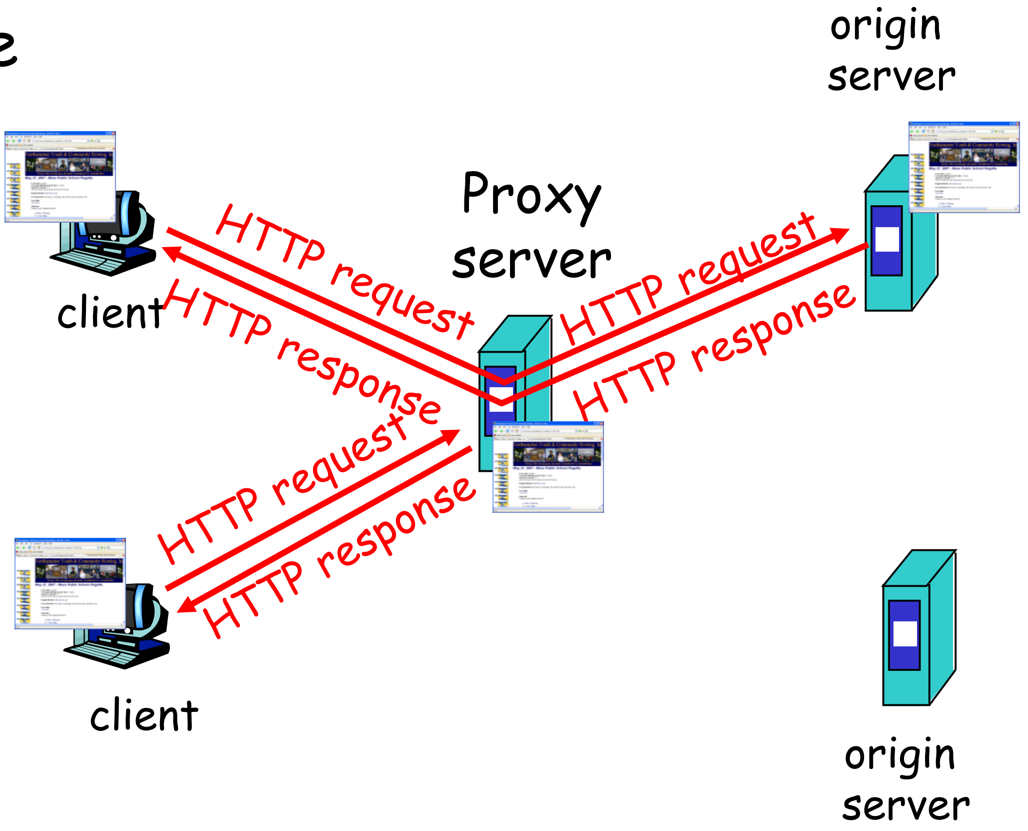
B. Persistent ***with*** pipelining:

- default in HTTP/1.1
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

Web caches (proxy servers)

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
  - Cache keeps copy of object for future use



- Can all objects be cached?
- Proxy vs. local browser cache

More about Web caching

- cache acts as both client and server
- typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

- 1. reduce response time for client request
- 2. reduce traffic on an institution's access link.

Caching example

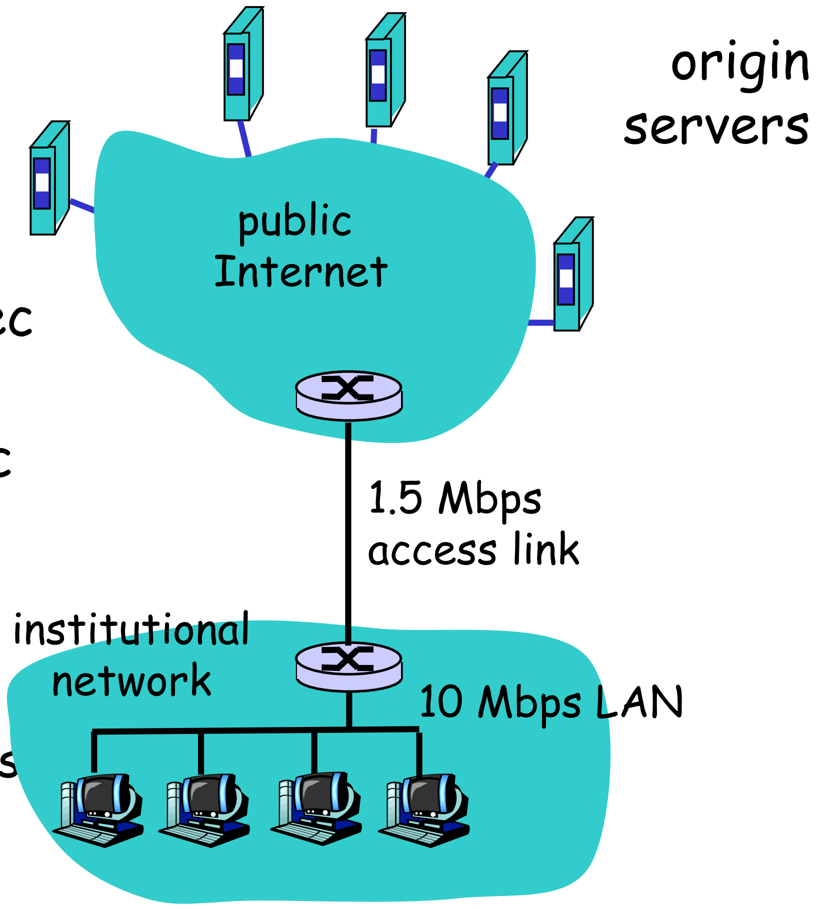
Assumptions

- average object size = 100k bits
- avg. request rate from institution's browsers = 15 req/sec
- delay from institutional router to any origin server and back = 2 sec

Consequences

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

= 2 sec + minutes + milliseconds



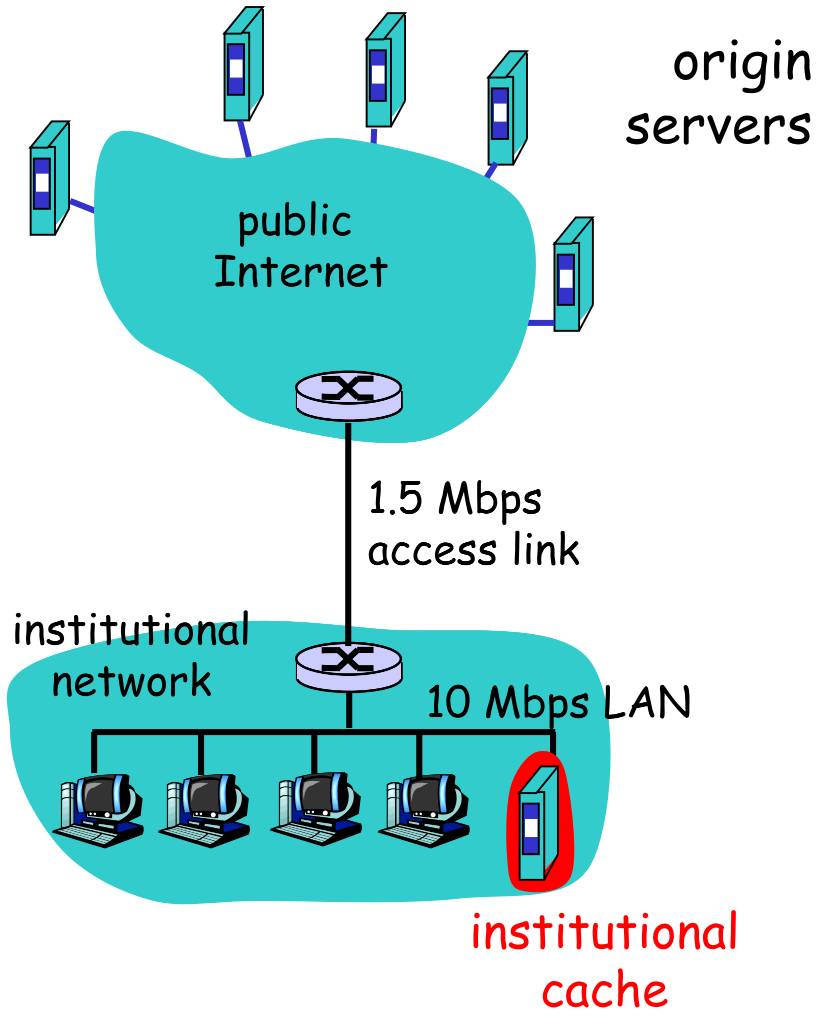
Caching example (cont)

one solution: install cache

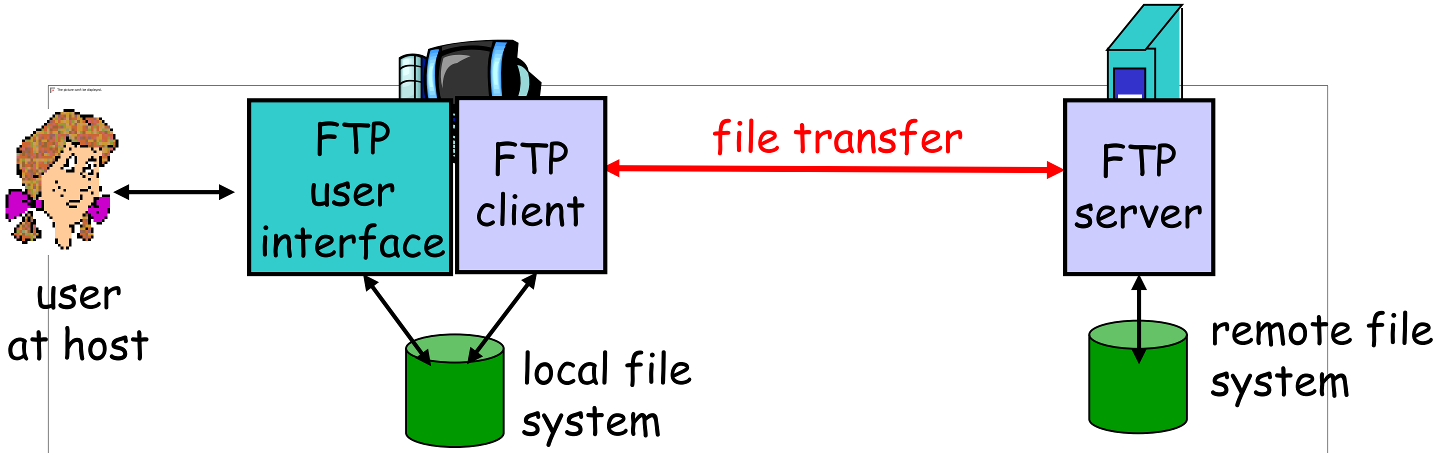
- suppose hit rate is 0.4

consequence

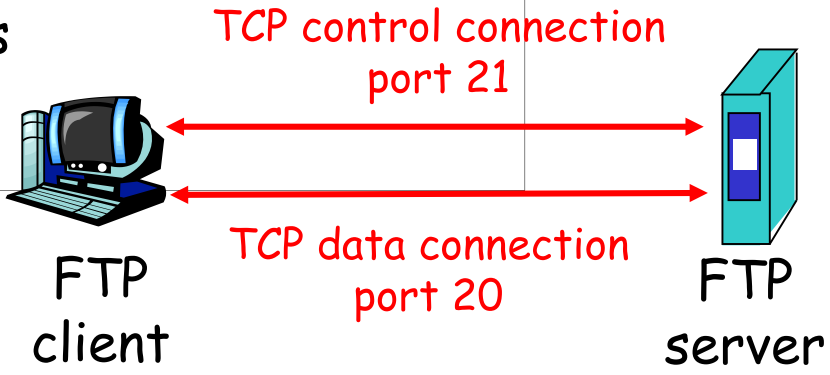
- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
- total avg delay = Internet delay + access delay + LAN delay = .6*(2.01) secs + .4*milliseconds < 1.4 secs



FTP: the file transfer protocol

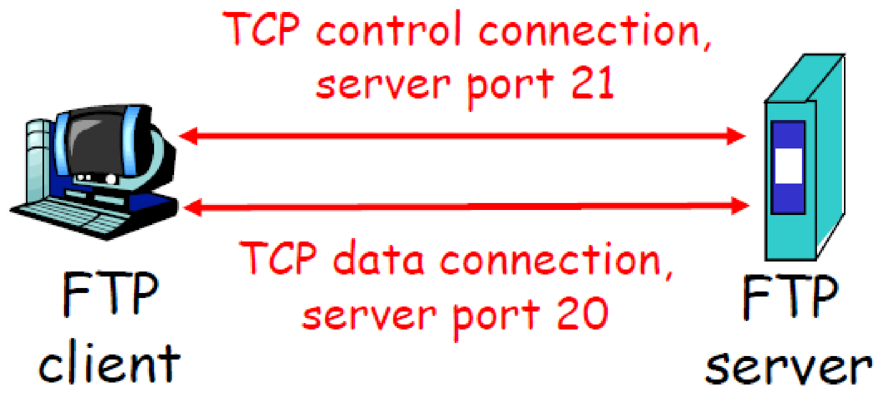


- client/server model
  - *client:* side initiating transfer, *server:* remote host
- ftp: RFC 959, ftp server: port 21
- Separate data and control connections
- FTP server maintains "state":
  - current directory, earlier authentication



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 2nd TCP connection (for file) to client
- after transferring one file, server closes data connection.


TCP control connection, server port 21
TCP data connection, server port 20
FTP client
FTP server

- 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`

Electronic Mail

Three components:

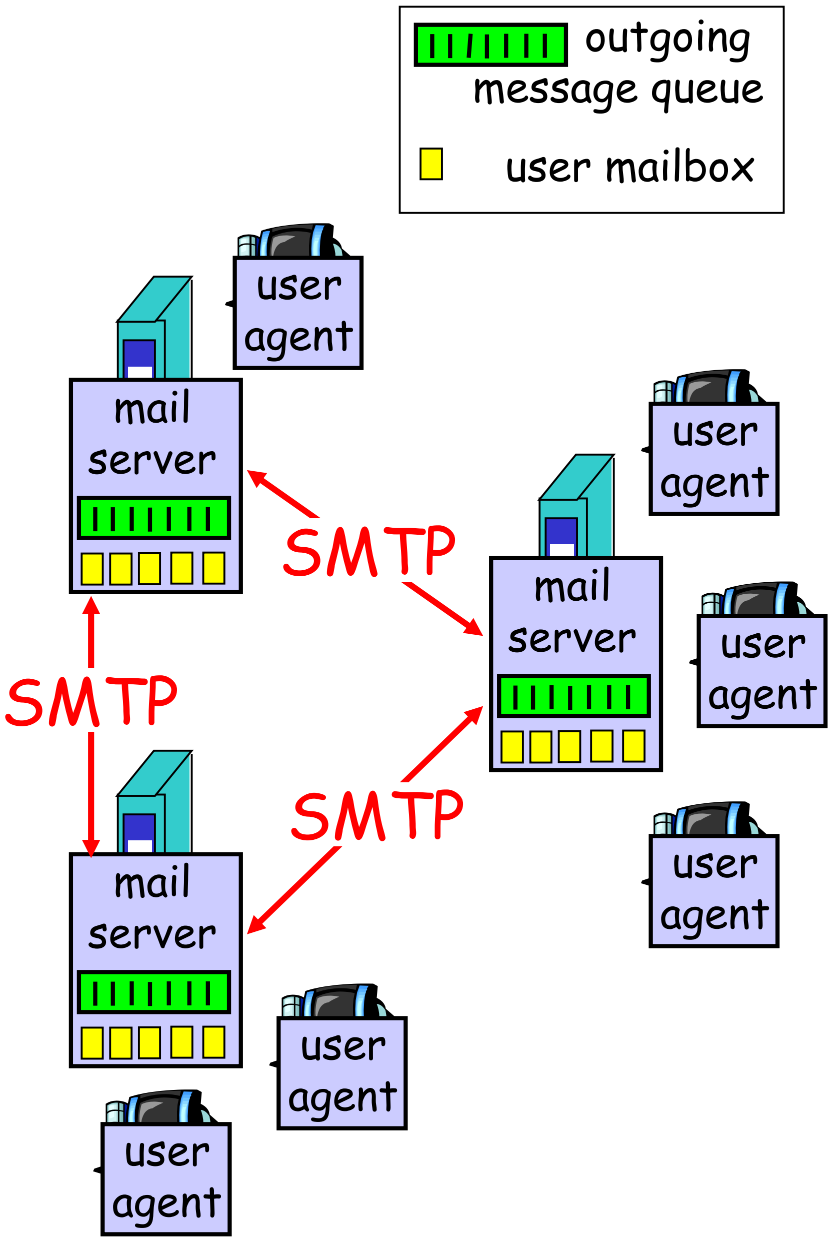
- 1. user agents, 2. mail servers
- 3. SMTP (simple mail transfer protocol)

User Agent

- "mail reader": editing, reading mail
- e.g., Outlook, Mozilla Thunderbird
- Out/incoming msgs stored on server

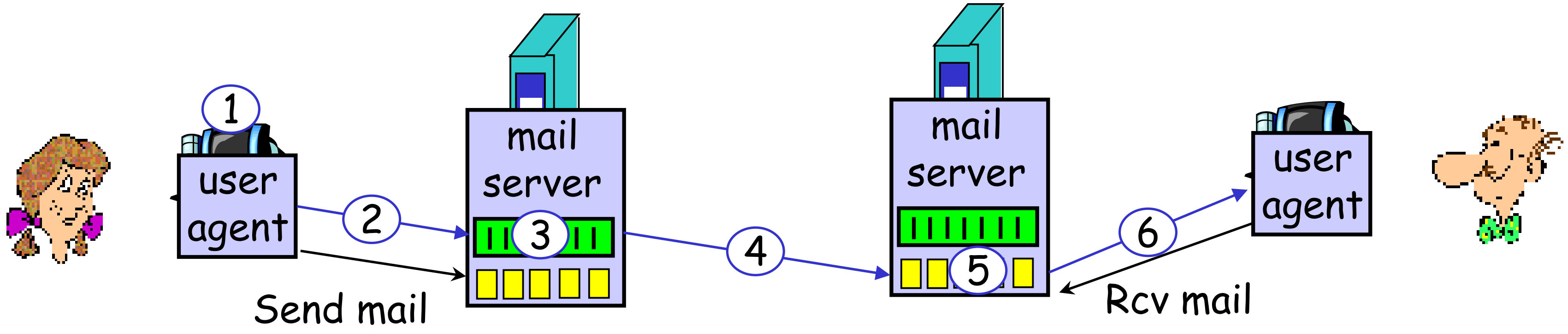
Mail Servers

- Mailbox: incoming messages
- message queue outgoing msgs
- SMTP protocol between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server



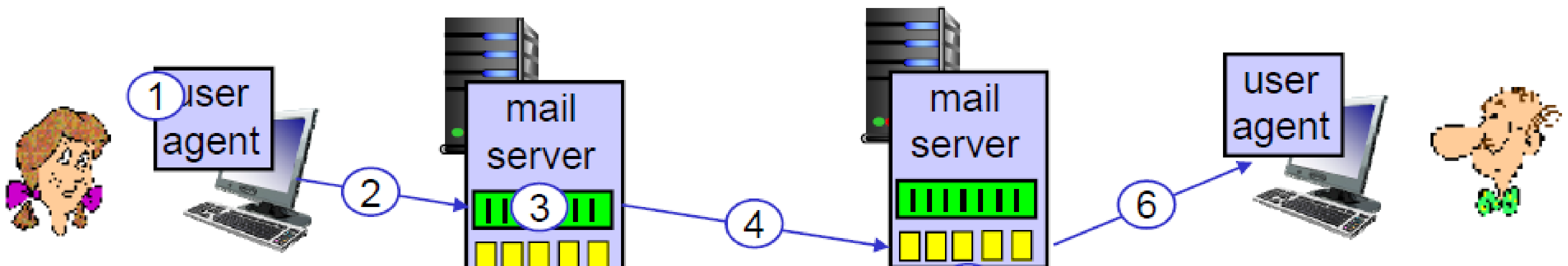
Electronic Mail: SMTP [RFC 2821]

- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - 1. handshake, 2. transfer of messages, 3. closure
- SMTP uses persistent connections: sending mail server sends all its messages to the receiving mail server over one TCP connection
- Email Scenario:



Scenario: Alice sends message to Bob

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



Alice's mail server

Bob's mail server

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 message
- SMTP: multiple objects sent in multipart message

Chapter 2: Application layer

DNS: Domain Name System

Internet identifiers for hosts, routers:

- IP address used for addressing datagrams
- "name", e.g., ww.yahoo.com - used by humans

Q: map between IP addresses and name ?

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

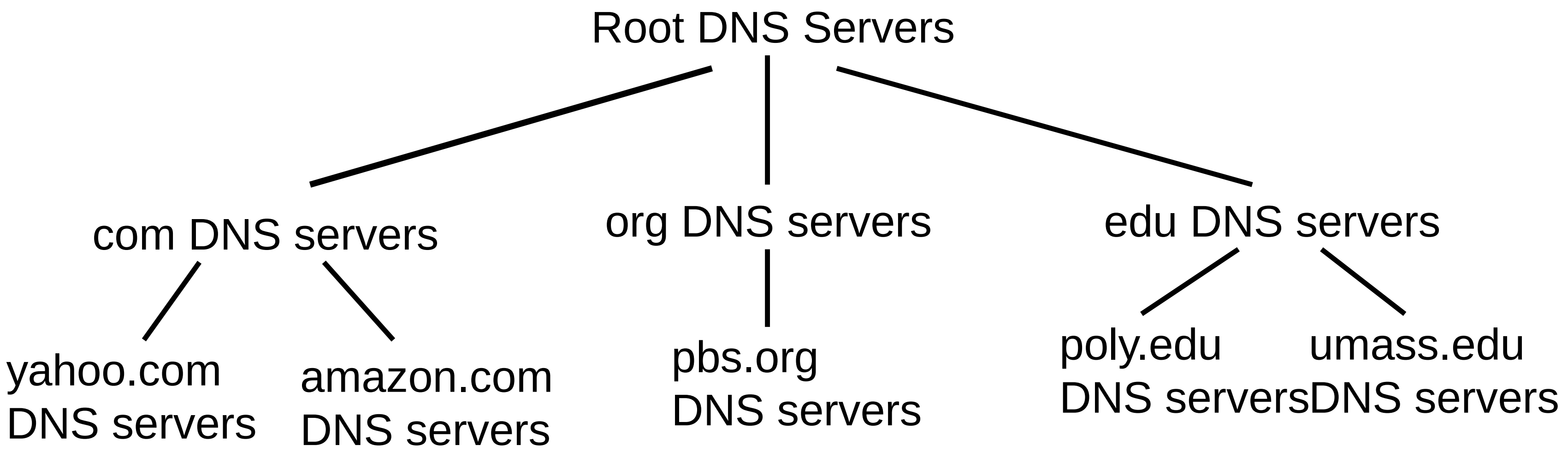
- hostname to IP address translation
- host aliasing
  - Canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?

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

doesn't *scale!*

Distributed, Hierarchical Database

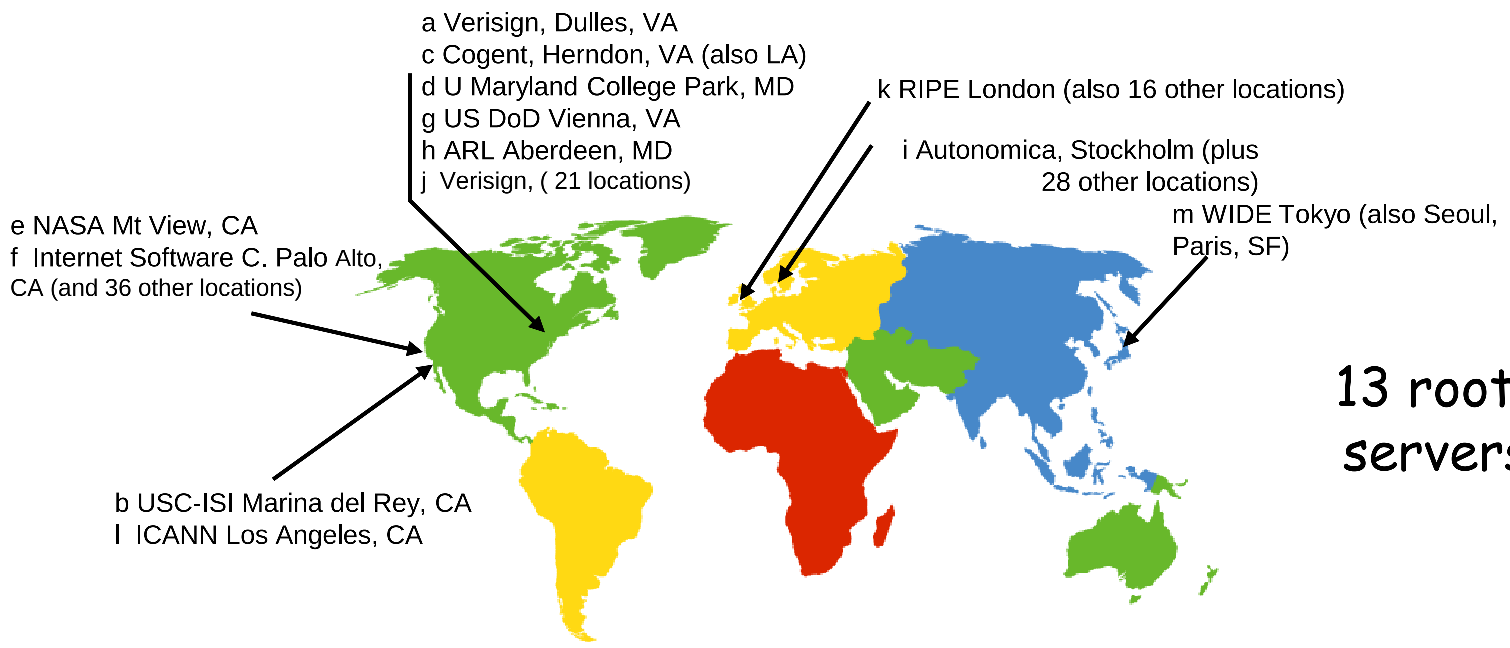


Client wants IP for www.amazon.com; 1st approx:

- 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


a Verisign, Dulles, VA
c Cogent, Herndon, VA (also LA)
d U Maryland College Park, MD
g US DoD Vienna, VA
h ARL Aberdeen, MD
j Verisign, (21 locations)
k RIPE London (also 16 other locations)
i Autonomica, Stockholm (plus 28 other locations)
m WIDE Tokyo (also Seoul, Paris, SF)
e NASA Mt View, CA
f Internet Software C. Palo Alto, CA (and 36 other locations)
b USC-ISI Marina del Rey, CA
l ICANN Los Angeles, CA

13 root name servers worldwide

TLD and Authoritative Servers

- I. Top-level domain (TLD) servers:
  - responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
  - Network Solutions maintains servers for com TLD
  - Educause for edu TLD
- II. 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

III. Local Name Server

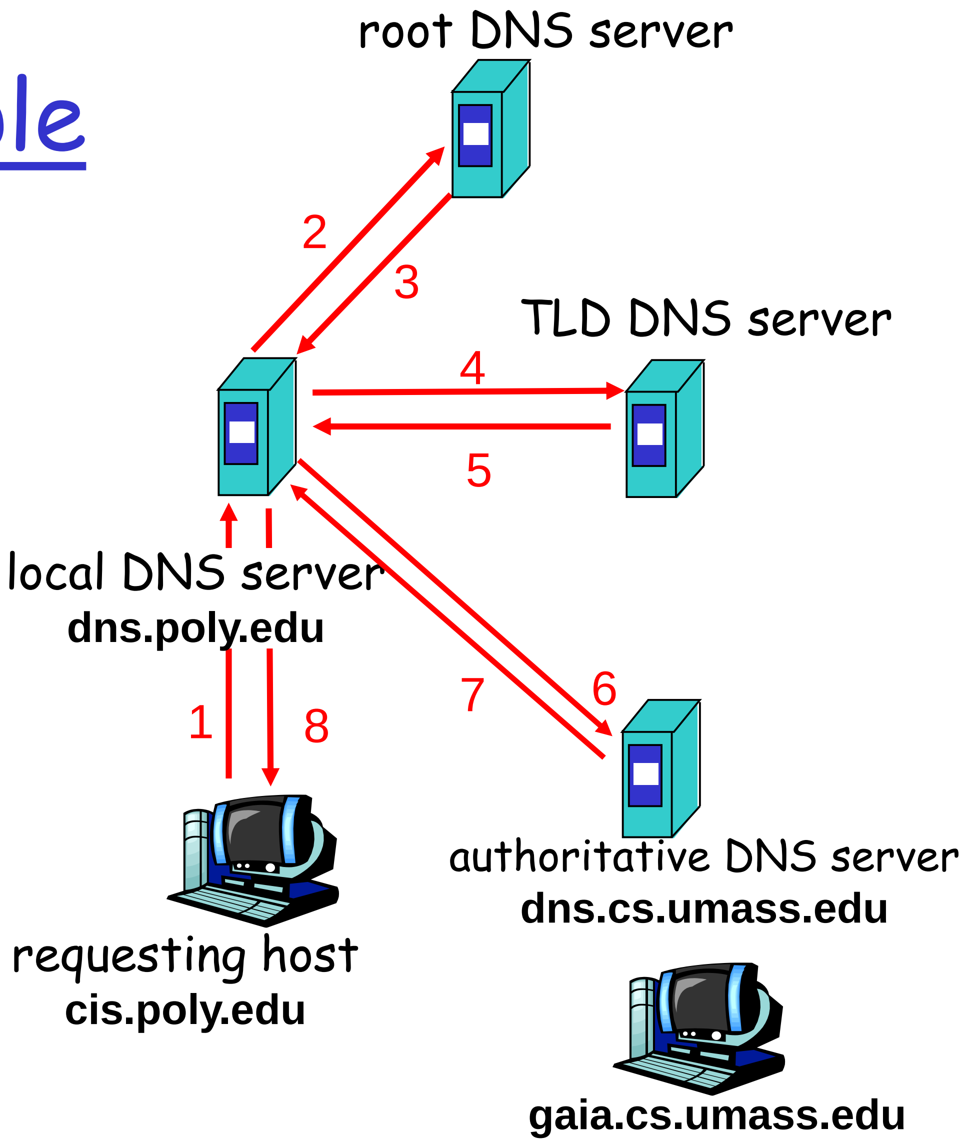
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one.
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - acts as proxy, forwards query into hierarchy

DNS name resolution example

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

A. iterative query:

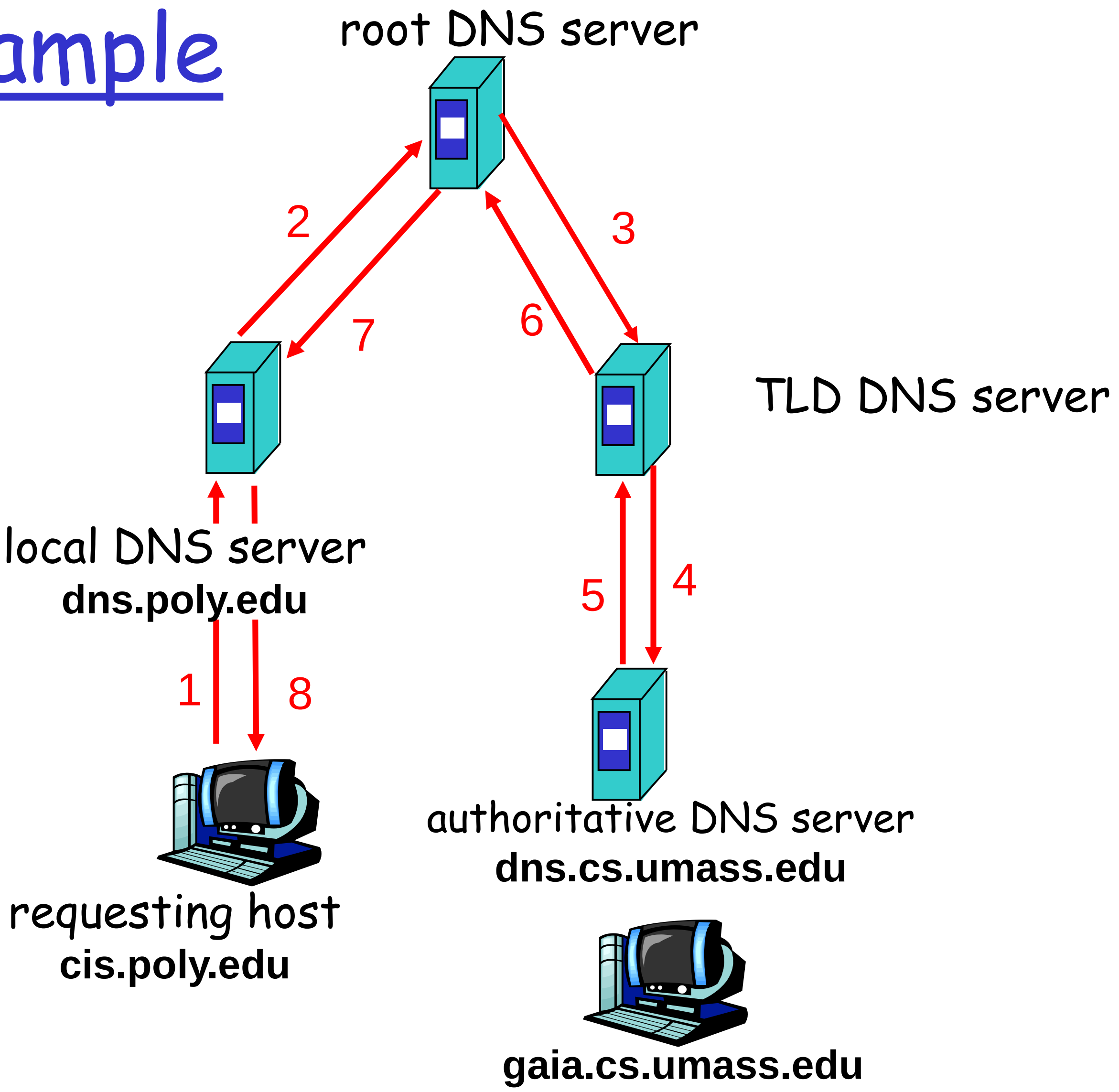
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

B. recursive query:

- puts burden of name resolution on contacted name server
- heavy load?



Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

P2P file sharing

Example

- Alice runs P2P client application on her notebook computer
- intermittently connects to Internet; gets new IP address for each connection
- asks for "Hey Jude"
- application displays other peers that have copy of Hey Jude.
- Alice chooses one of the peers, Bob.
- file is copied from Bob's PC to Alice's notebook: HTTP
- while Alice downloads, other users uploading from Alice.
- Alice's peer is both a Web client and a transient Web server.

All peers are servers = highly scalable!

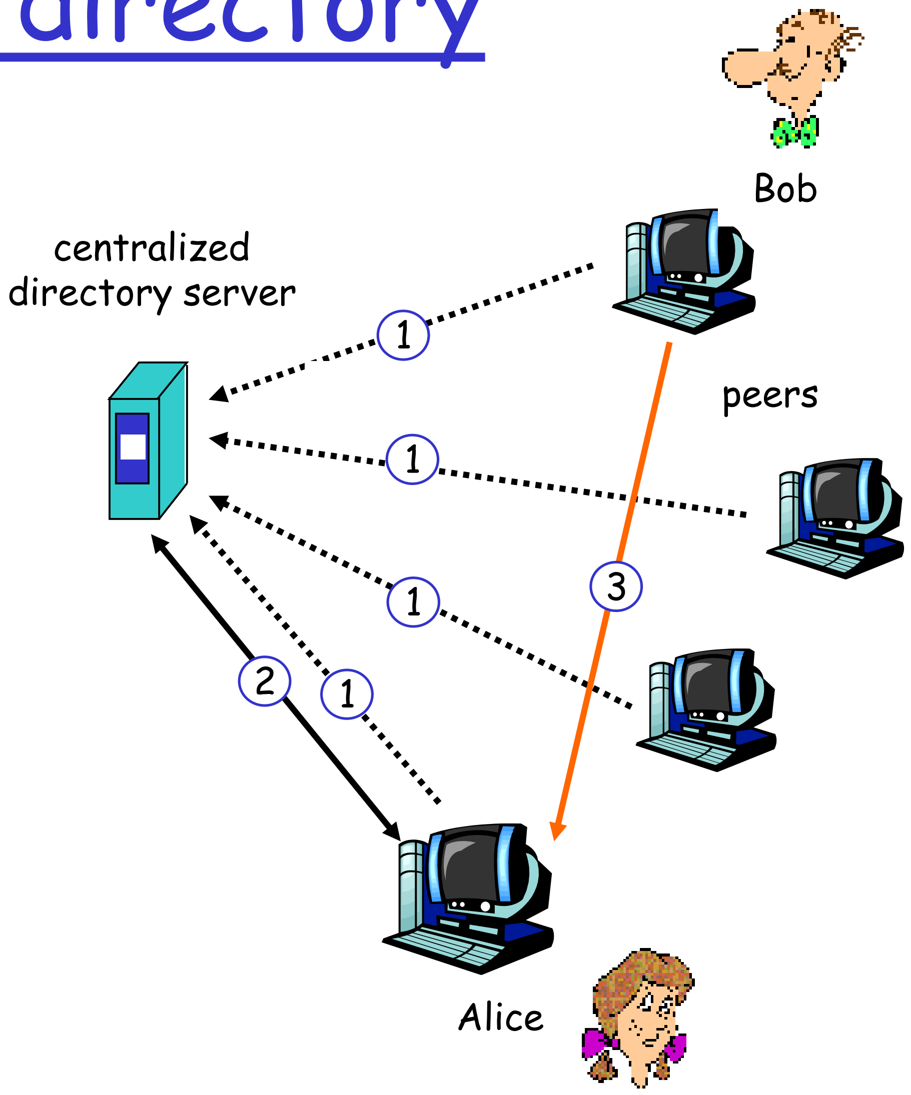
P2P: centralized directory

original "Napster" design

1) when peer connects, it informs central server:
   - IP address
   - content

2) Alice queries for "Hey Jude"

3) Alice requests file from Bob



P2P: problems with centralized directory

- single point of failure
- performance bottleneck
- copyright infringement: "target" of lawsuit is obvious

file transfer is decentralized, but locating content is highly centralized

Advantages vs. disadvantages
Search time and overhead?