Chapter 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
- programming network applications
  - socket API

Some network apps

- E-mail
- Web
- Instant messaging
- Remote login
- P2P file sharing
- Multi-user network games
- Streaming stored video clips
- Internet telephone
- Real-time video conference
- Massive parallel computing

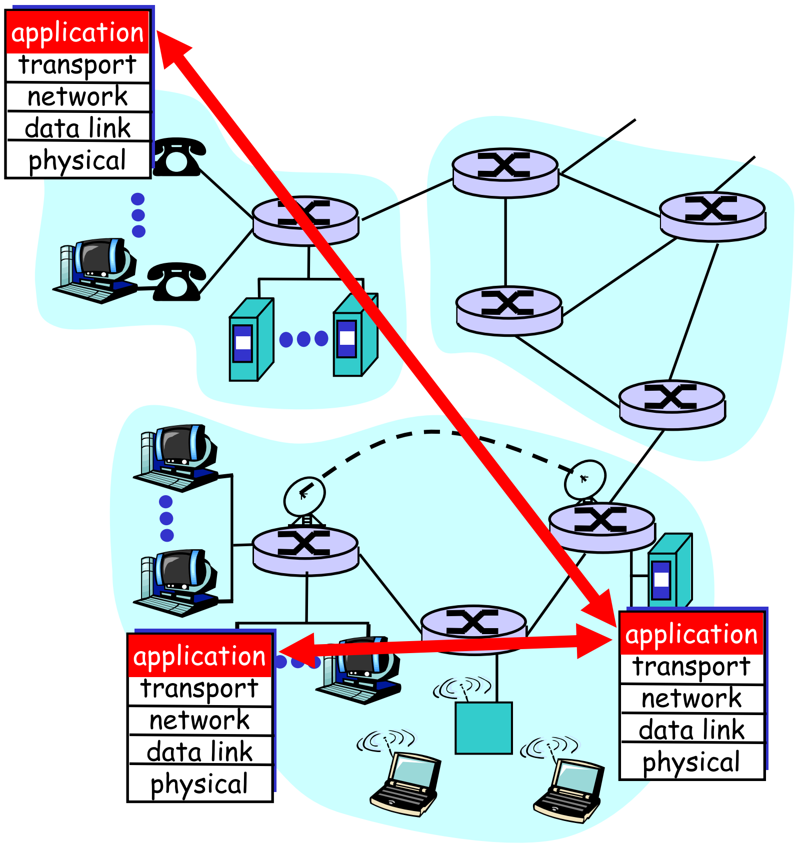
Creating a network app

Write programs that

- run on different end systems and
- communicate over a network.
- e.g., Web: Web server software communicates with browser software

No software written for devices in network core

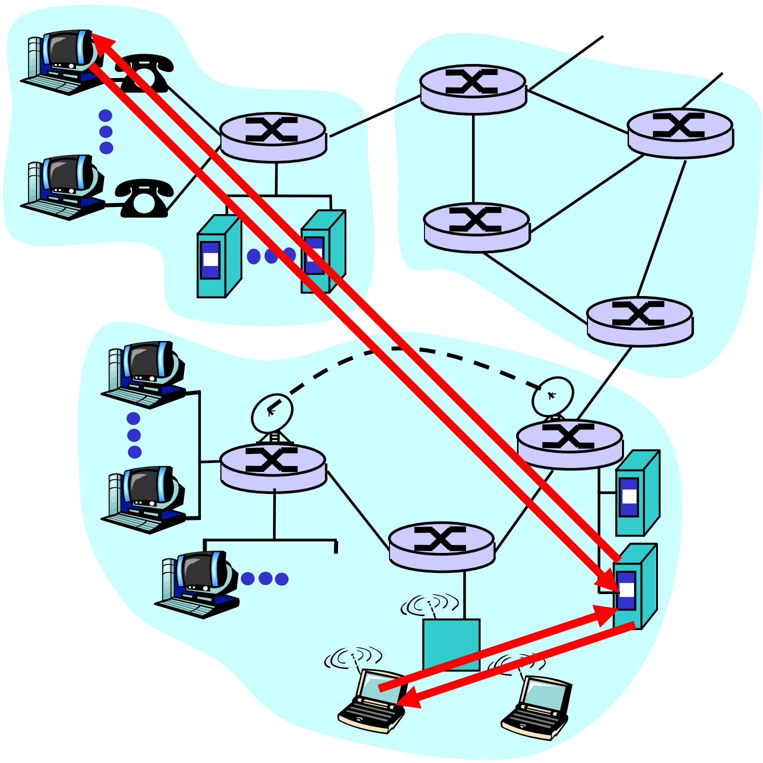
- Network core devices do not function at app layer
- This design allows for rapid app development



Application architectures

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

Client-server architecture

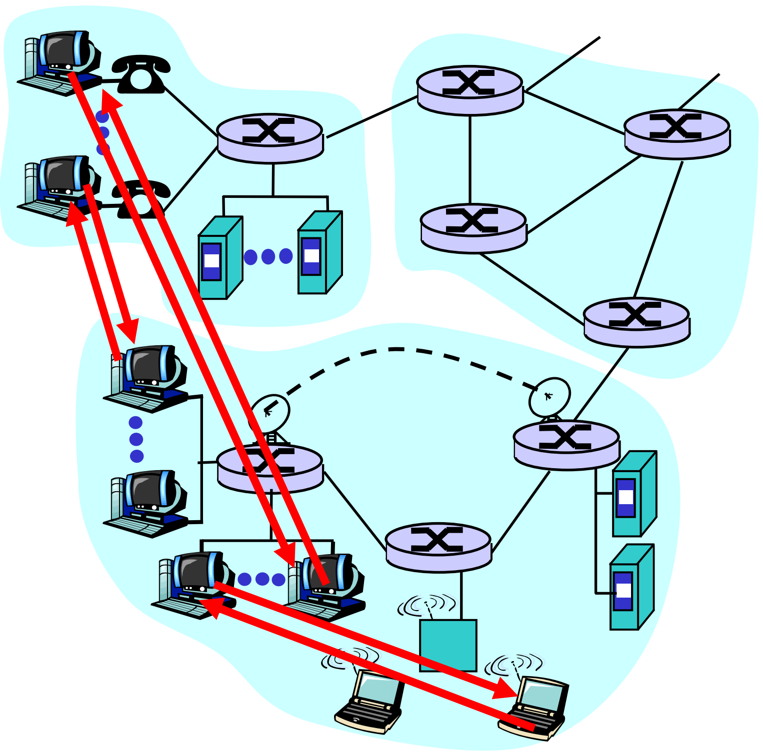


server:
- always-on host
- permanent IP address
- server farms for scaling

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

Pure P2P architecture

- no always on server
- arbitrary end systems directly communicate
- peers are intermittently connected and 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
- Presence detection/location centralized:
  - User registers its IP address with central server when it comes online
  - User contacts central server to find IP addresses of friends

Network applications: some jargon

Process: program running within a host.

- within same host, two processes communicate using interprocess communication (defined by OS).
- processes running in different hosts communicate with an application-layer protocol

user agent: interfaces with user "above" and network "below".

- implements user interface & application-level protocol
  - Web: browser
  - E-mail: mail reader
  - streaming audio/video: media player

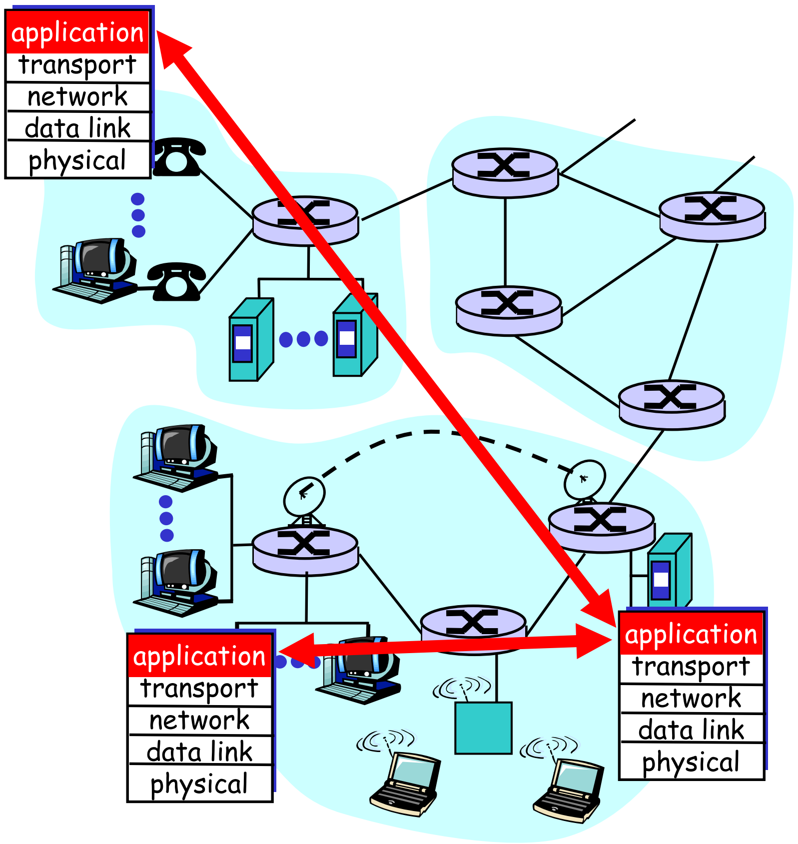
Applications and application-layer protocols

Application: communicating, distributed processes

- e.g., e-mail, Web, P2P file sharing, instant messaging
- running in end systems (hosts)
- exchange messages to implement application

Application-layer protocols

- one "piece" of an app
- define messages exchanged by apps and actions taken
- use communication services provided by lower layer protocols (TCP, UDP)



App-layer protocol defines

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

Public-domain protocols:

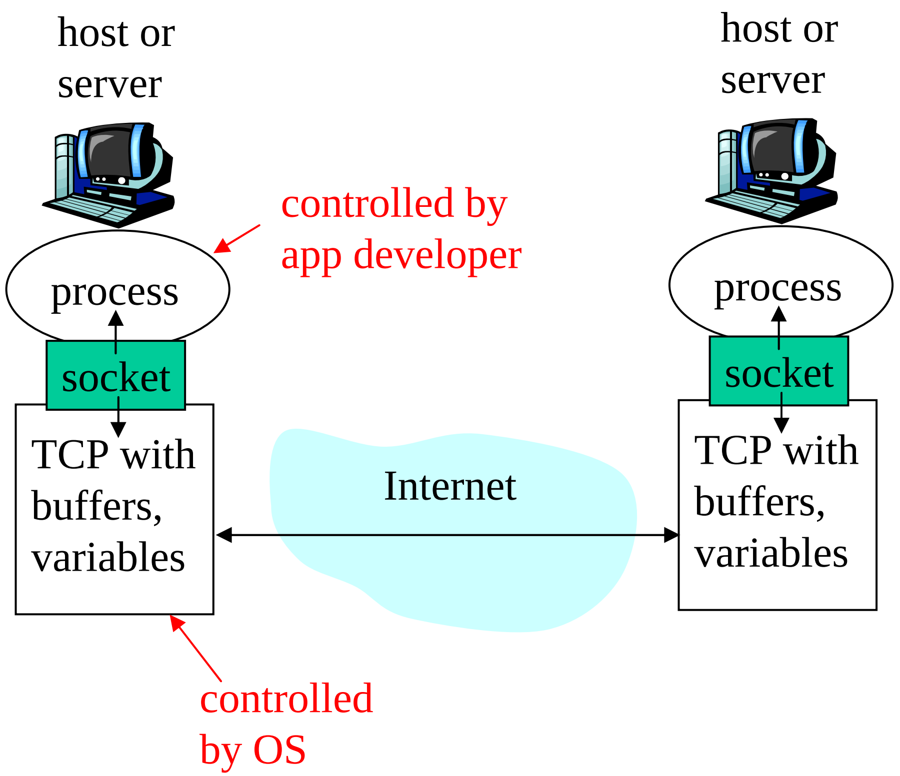
- defined in RFCs
- allows for interoperability
- eg, HTTP, SMTP

Proprietary protocols:

- eg, KaZaA, Skype

Processes communicating across network

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process asssumes 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
- Every host has a unique 32-bit 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: 25

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"

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 providing:* timing, minimum bandwidth guarantees

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

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	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Dialpad)	typically UDP

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:

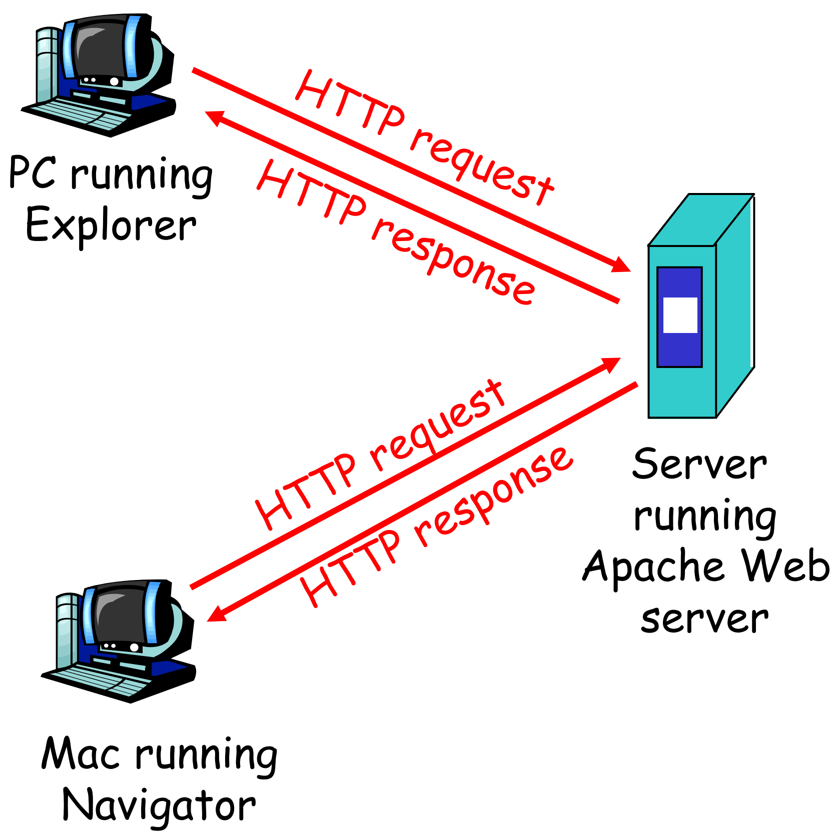
```
www.cs.bilkent.edu.tr/bilkent/academic/main_logo.gif
```

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
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068



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

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.
- HTTP/1.0 uses nonpersistent HTTP

Persistent HTTP

- Multiple objects can be sent over single TCP connection between client and server.
- HTTP/1.1 uses persistent connections in default mode

Nonpersistent HTTP

Suppose user enters URL
`www.bilkent.edu.tr/someDepartment/`

(contains text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at www.bilkent.edu.tr on port 80

1b. HTTP server at host www.bilkent.edu.tr waiting for TCP connection at port 80. "accepts" connection, notifying client

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

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

time

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

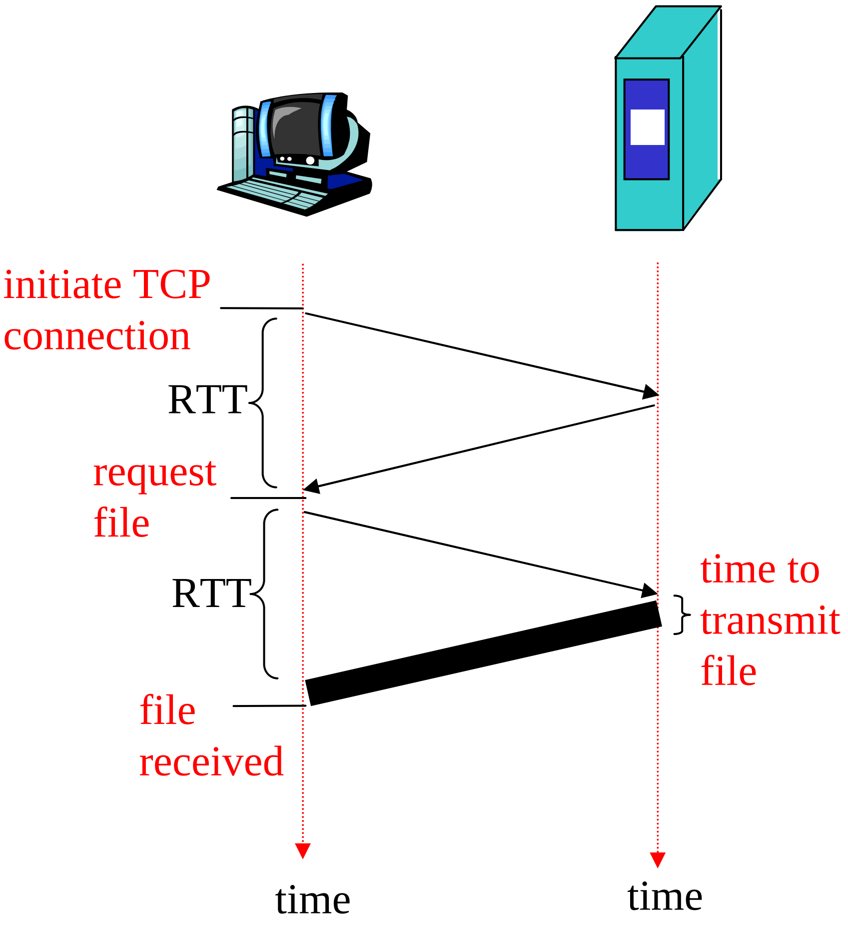
Response time modeling

Definition of RRT: 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+transmit time



Persistent HTTP

Nonpersistent 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

Persistent without pipelining:

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

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

HTTP request message

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

request line (GET, POST, HEAD commands)

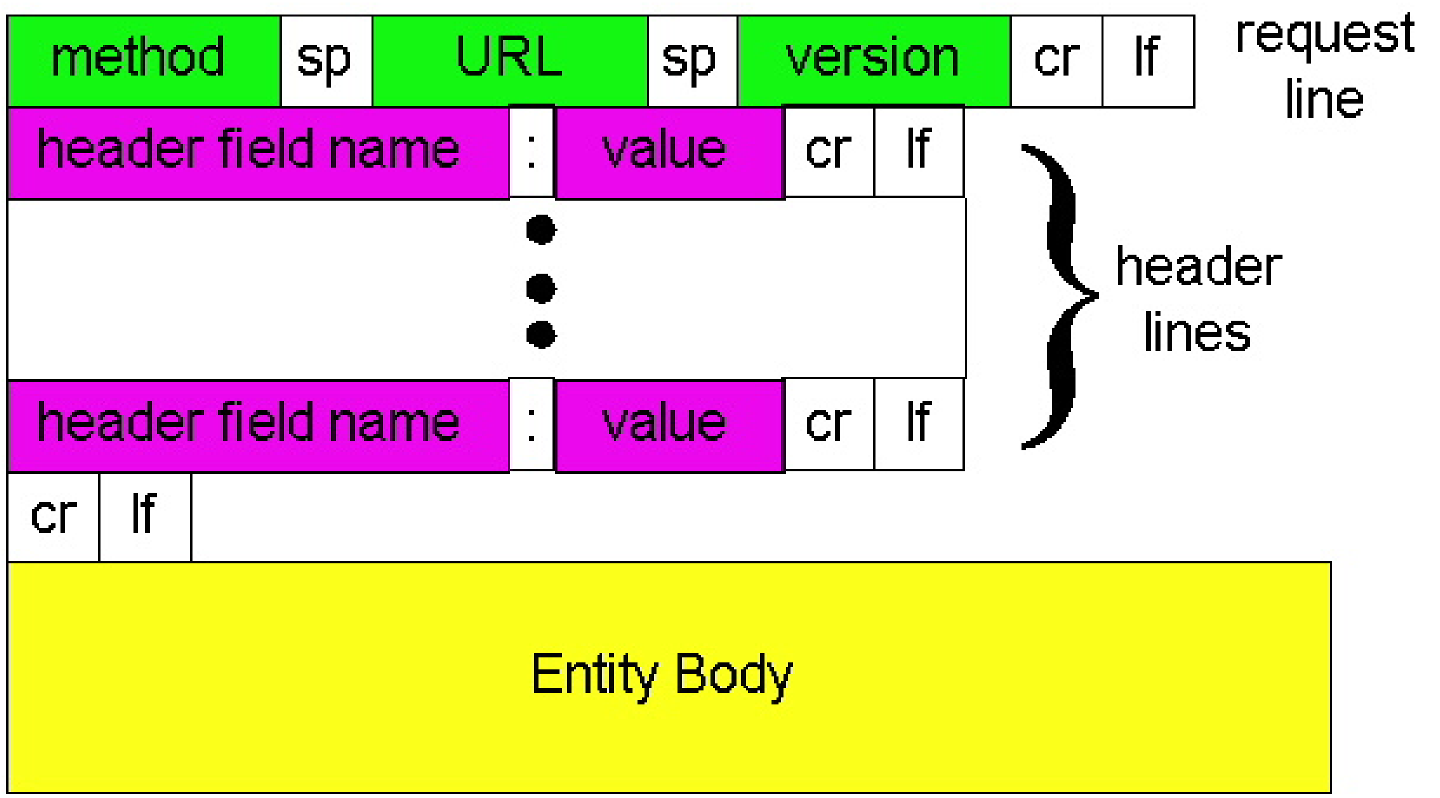
header lines

Carriage return, line feed indicates end of message

```
GET /somedir/page.html HTTP/1.1
Host: www.bilkent.edu.tr
User-agent: Mozilla/4.0
Connection: close
Accept-language:fr
```

(extra carriage return, line feed)

HTTP request message: general format



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

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
```

HTTP response message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
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.ee.bilkent.edu.tr 80
```

Opens TCP connection to port 80 (default HTTP server port) at www.ee.bilkent.edu.tr. Anything typed in sent to port 80 at www.ee.bilkent.edu.tr

2. Type in a GET HTTP request:

```
GET /~ezhan/index.html HTTP/1.0
```

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!

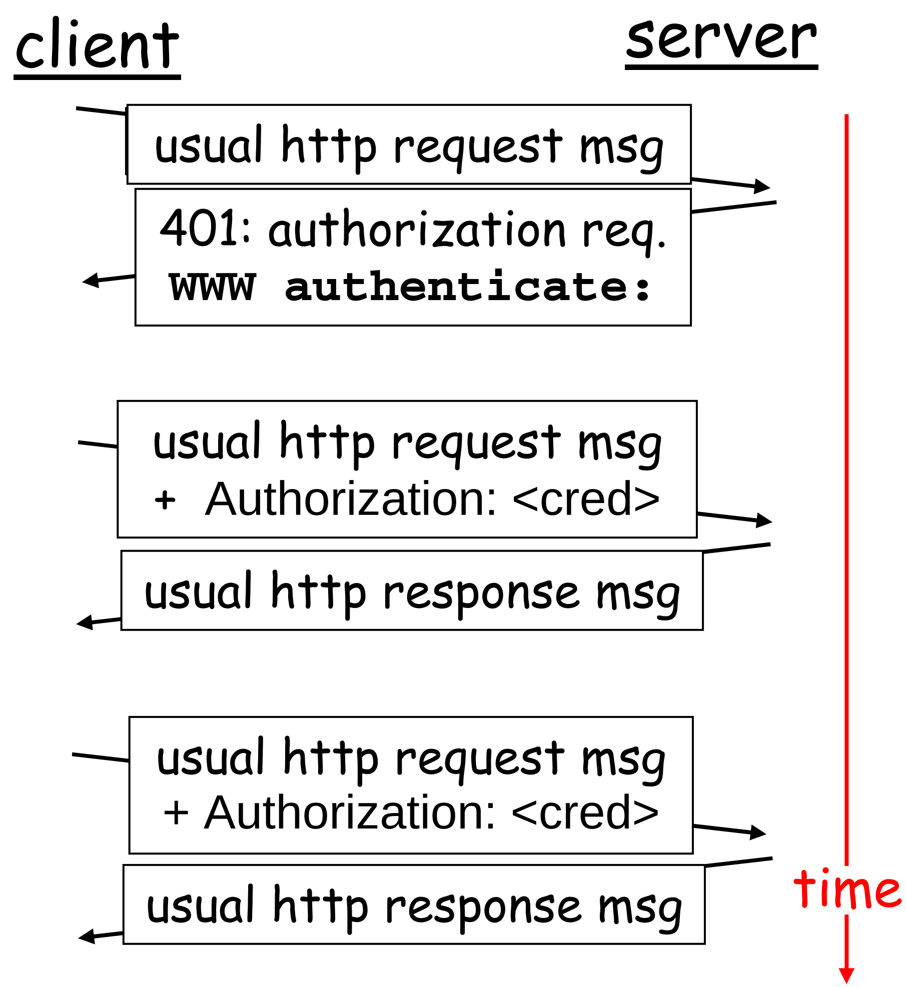
User-server interaction: authorization

Authorization : control access to server content

- authorization credentials: typically name, password
- stateless: client must present authorization in *each* request
  - authorization: header line in each request
  - if no authorization: header, server refuses access, sends

    WWW authenticate:

    header line in response



Cookies: keeping "state"

Many major Web sites use cookies

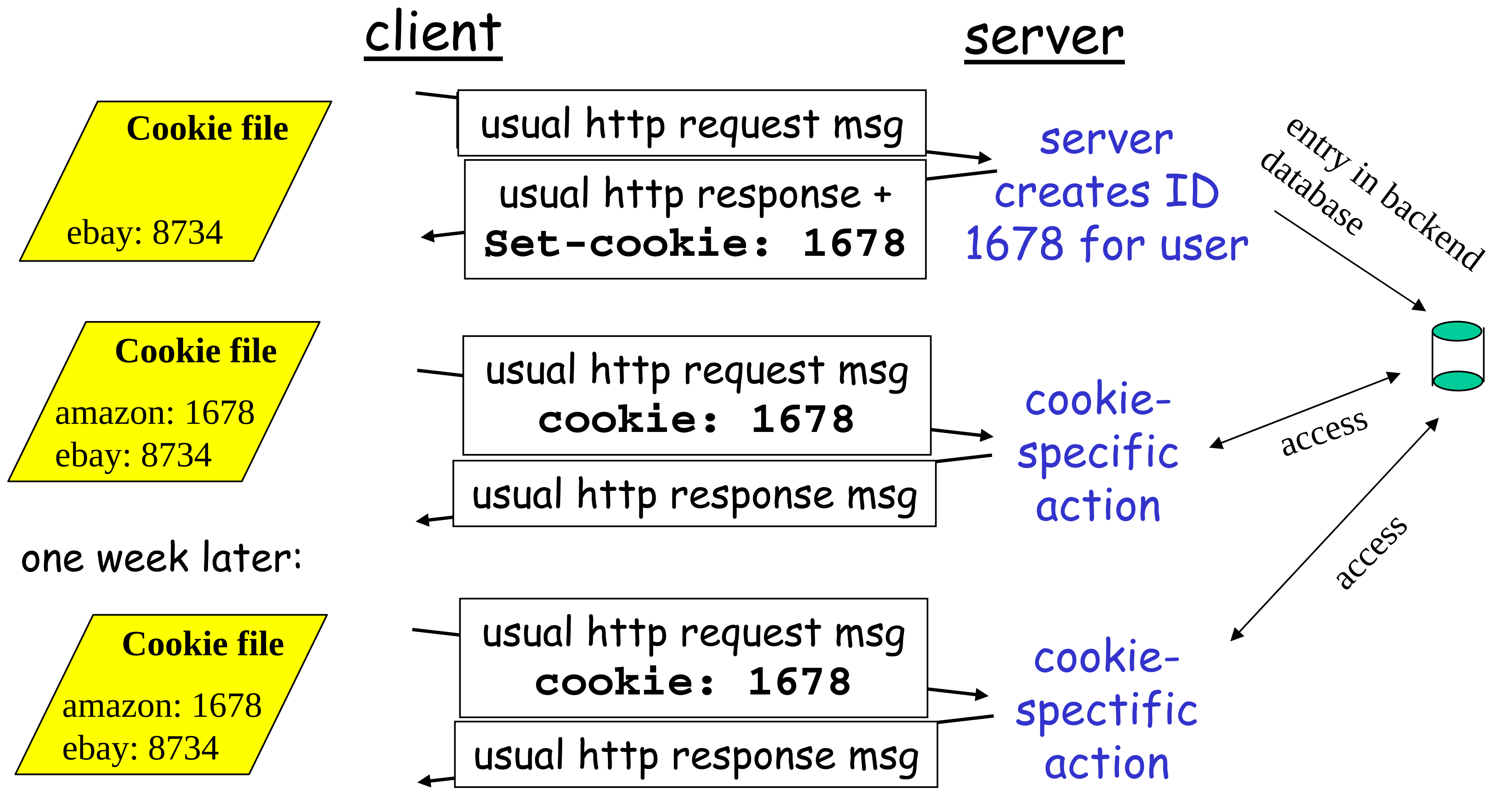
Four components:

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) back-end database at Web site

Example:

- Susan access 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 bring:

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

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

Set-Cookie HTTP Response Header

Set-Cookie: *NAME*=*VALUE*; expires=*DATE*; path=*PATH*; domain=*DOMAIN_NAME*; secure

- ***NAME=VALUE***
  - sequence of characters excluding semi-colon, comma and white space (the only required field)
- **expires**=*DATE*
  Format: Wdy, DD-Mon-YYYY HH:MM:SS GMT
- **domain**=*DOMAIN_NAME*
  - Browser performs "tail matching" searching through cookies file
  - Default **domain** is the host name of the server which generated the cookie response
- **path**=*PATH*
  - the subset of URLs in a domain for which the cookie is valid
- **Secure**: if secure cookie will only be transmitted if the communications channel with the host is secure, e.g., HTTPS

Cookies File

- Netscape keeps all cookies in a single file ~username/.netscape/cookies whereas IE keeps each cookie in separate files in the folder C:\Documents and Settings\user\Cookies

```
# Netscape HTTP Cookie File
# http://www.netscape.com/newsref/std/cookie_spec.html
# This is a generated file!  Do not edit.

.netscape.com   TRUE   /   FALSE   1128258721   sampler 1097500321
.edge.ru4.com   TRUE   /   FALSE   2074142135   ru4.uid 2|3|0#1274030263208642l#1917818738
.edge.ru4.com   TRUE   /   FALSE   1133246135   ru4.1188.gts   :2
.netscape.com   TRUE   /   FALSE   1128065747   RWHAT   set|1128065747300
.nytimes.com   TRUE   /   FALSE   1159598159   RMID   833ff0b33a03433cdccf603e
.netscape.com   TRUE   /   FALSE   1128148560   adsNetPopup0   1128062159725
servedby.advertising.com   TRUE   /   FALSE   1130654161   1812261973   _433cdcd1,,695214^76559_
.advertising.com   TRUE   /   FALSE   1285742161   ACID   bb640011280621610000!
.bluestreak.com TRUE   /   FALSE   1443407766   id   3316728525856612O bb=141A11twQw_"4totrKoAA| adv=
.mediaplex.com  TRUE   /   FALSE   1245628800   svid   80016269101
.nytdigital.com TRUE   /   FALSE   1625726176   TID   0e0pcsb11jpn70
.nytdigital.com TRUE   /   FALSE   1625726176   TData
.nytimes.com   TRUE   /   FALSE   1625726176   TID   0e0pcsb11jpn70
.nytimes.com   TRUE   /   FALSE   1625726176   TData
.doubleclick.net   TRUE   /   FALSE   1222670215   id   8000006195fbc8b
servedby.advertising.com   TRUE   /   FALSE   1130654216   5907528 _433cdd08,,707769^243007_
www.yahoo.com   TRUE   /   FALSE   1149188401   FPB   fc1hmqbqc11jpnci
```

Cookies File Format

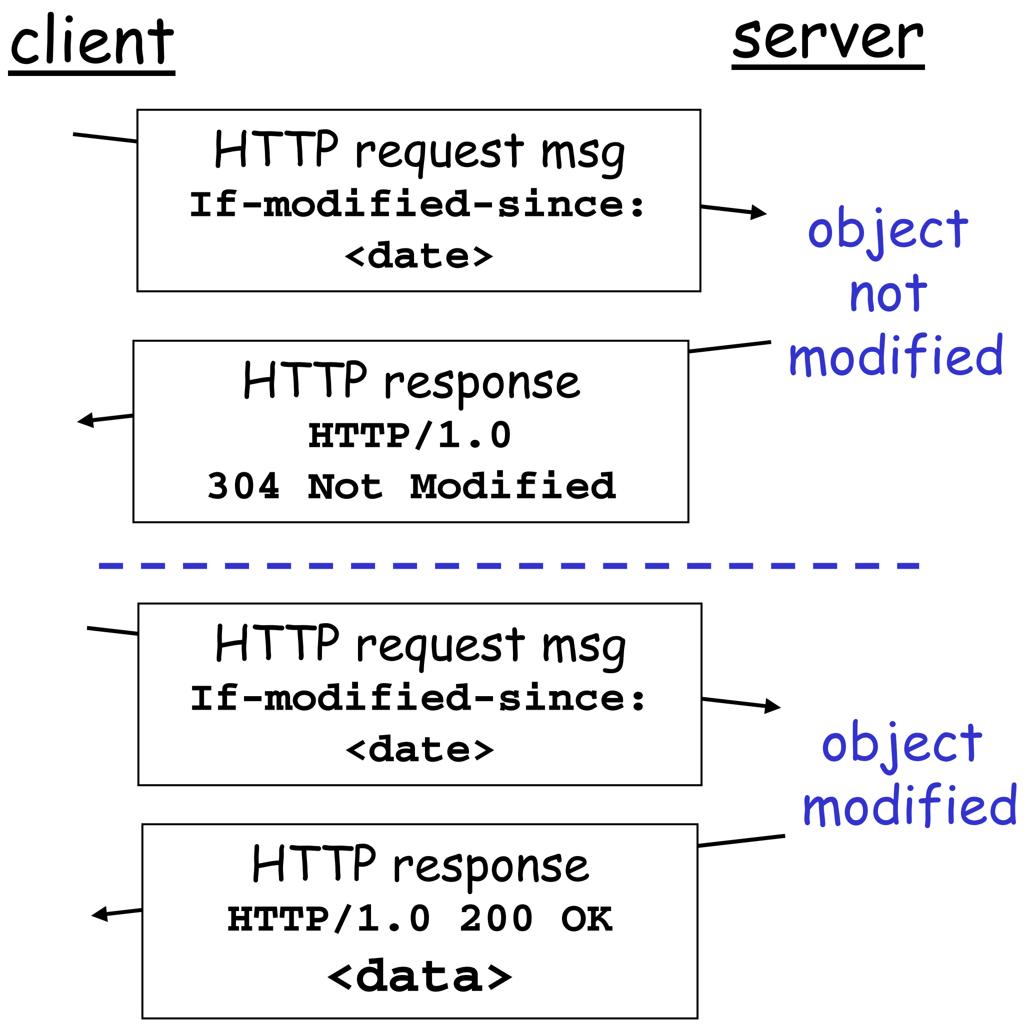
Domain	Accessible by all hosts	Path	Secure	Expiration (Unix time)	Name	Value
edge.ru4.com	TRUE	/	FALSE	2074142135	ru4.uid	2|3|0#1274…
nytimes.com	TRUE	/	FALSE	1625726176	TID	0e0pcsb11jpn70

Sun, 23 Sep 2035 06:35:35 UTC

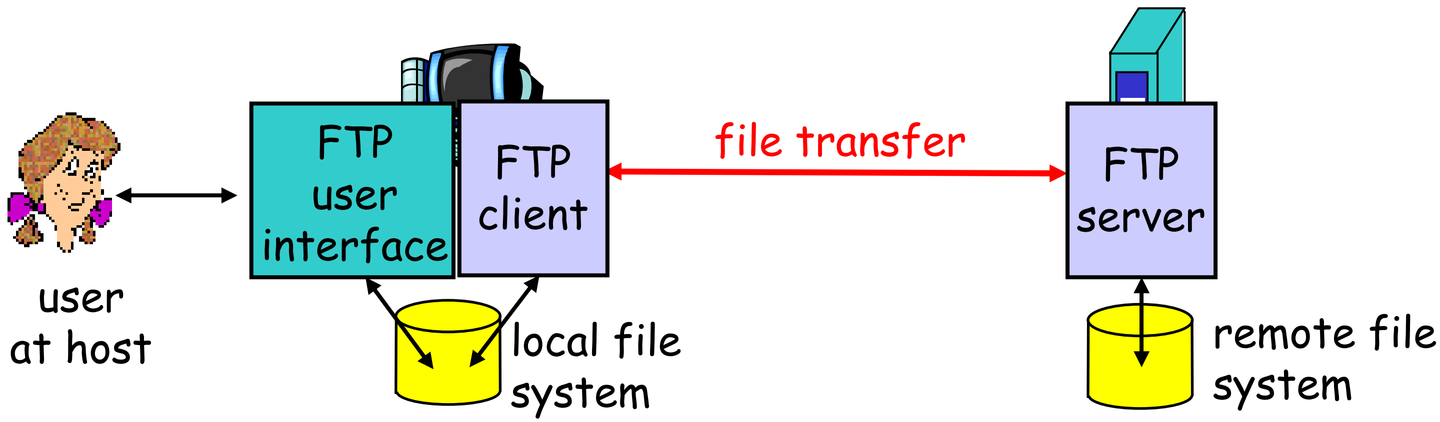
Thu, 8 Jul 2021 06:36:16 UTC

Conditional GET: client-side caching

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



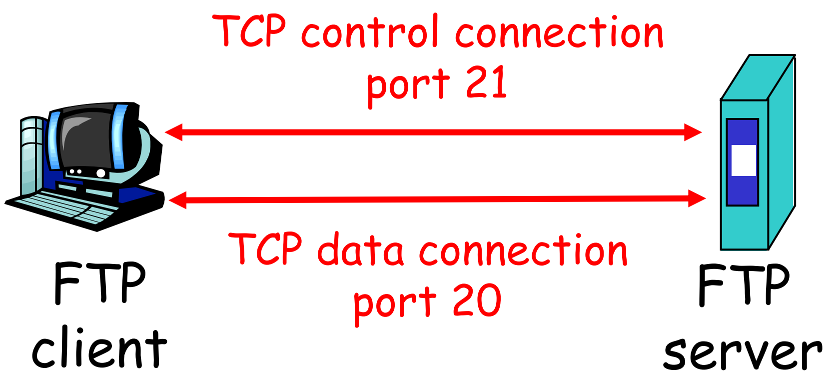
FTP: the file transfer protocol



- 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, specifying TCP as transport protocol
- Client obtains authorization over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives a command for a file transfer, the server opens a TCP data connection to client
- After transferring one file, server closes connection.



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

FTP commands, responses

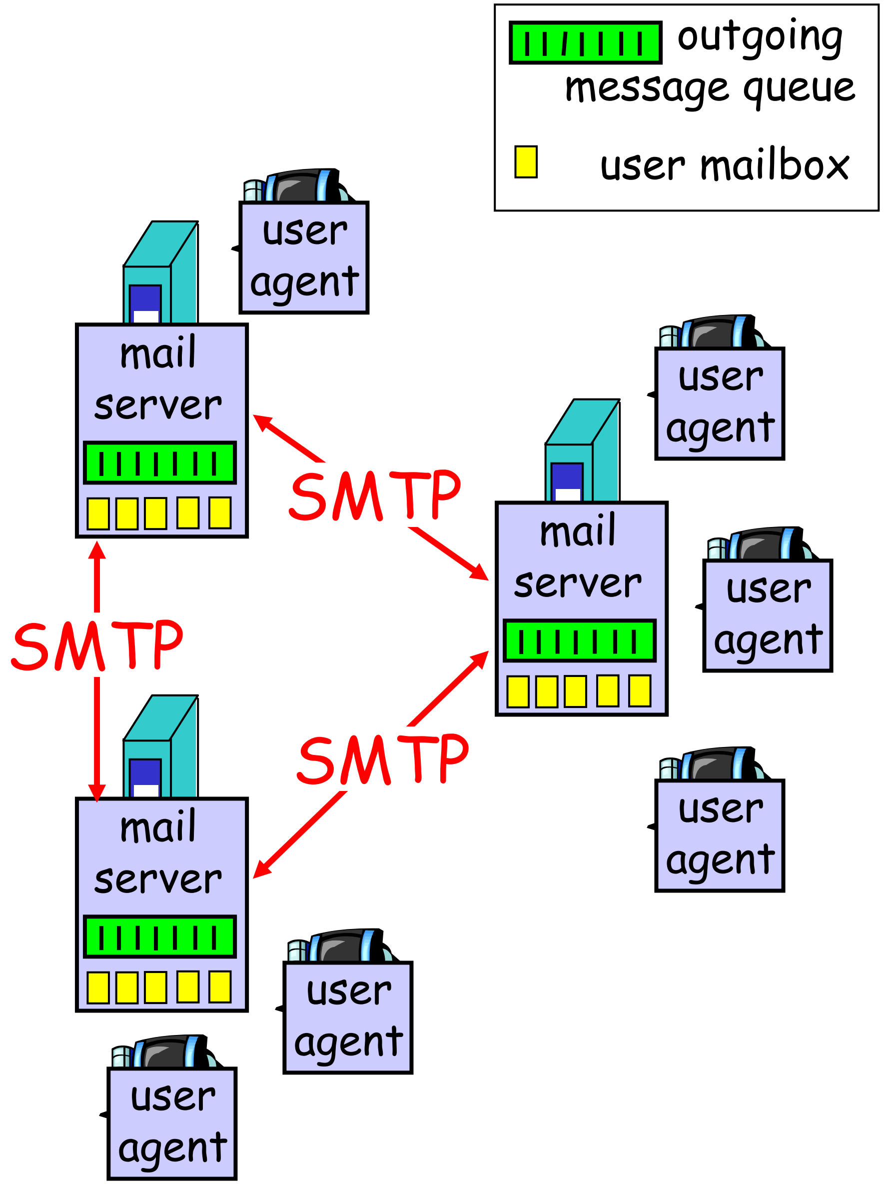
Sample commands:

- 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 major components:

- user agents
- mail servers
- simple mail transfer protocol: SMTP

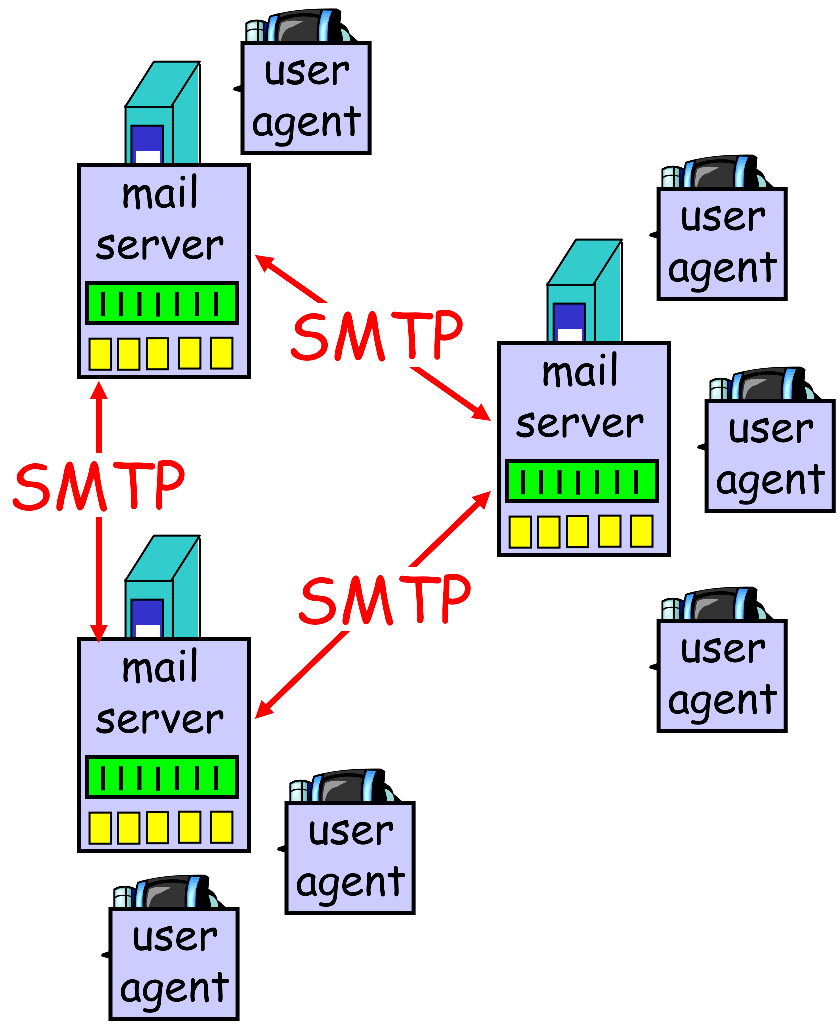
User Agent

- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- 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

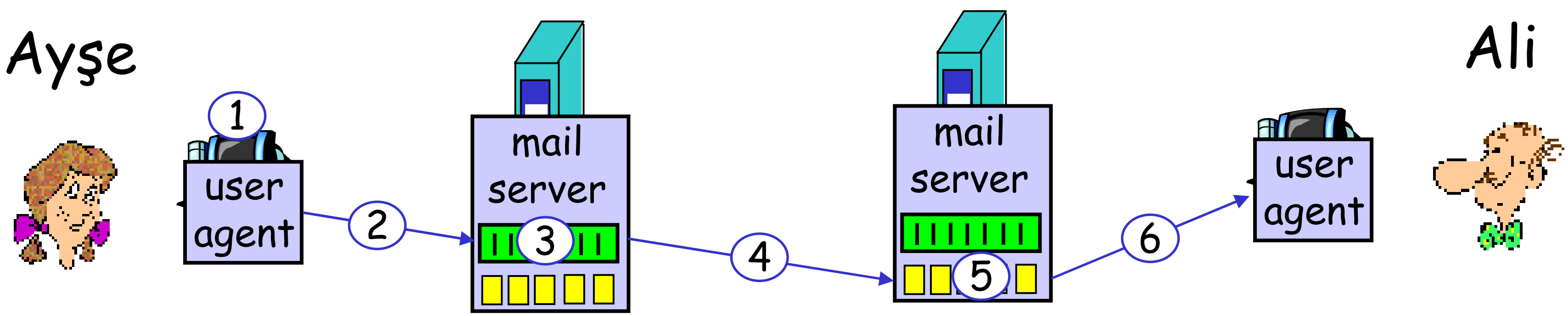


Electronic Mail: SMTP [RFC 2821]

- 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

Scenario: Ayşe sends message to Ali

1) Ayşe uses UA to compose message and "to" ali@bilkent.edu.tr
2) Ayşe's UA sends message to her mail server; message placed in message queue
3) Client side of SMTP opens TCP connection with Ali's mail server
4) SMTP client sends Ayşe's message over the TCP connection
5) Ali's mail server places the message in Ali's mailbox
6) Ali invokes his user agent to read message



SMTP interaction for yourself

- telnet cs.bilkent.edu.tr 25

  220 gordion.cs.bilkent.edu.tr ESMTP Sendmail 8.12.9/8.12.9; Wed, 3 Mar 2004 11:17:52 +0200 (EET)
- HELO cs.bilkent.edu.tr

  250 gordion.cs.bilkent.edu.tr Hello nemrut.ee.bilkent.edu.tr [139.179.12.28], pleased to meet you
- MAIL FROM: <somebody@somewhere.net>

  250 2.1.0 <somebody@somewhere.net>... Sender ok
- RCPT TO: <ezhan@ee.bilkent.edu.tr>

  250 2.1.5 <ezhan@ee.bilkent.edu.tr>... Recipient ok
- DATA

  354 Enter mail, end with "." on a line by itself
- hello

  .

  250 2.0.0 Message accepted for delivery
- QUIT

  221 2.0.0 gordion.cs.bilkent.edu.tr closing connection

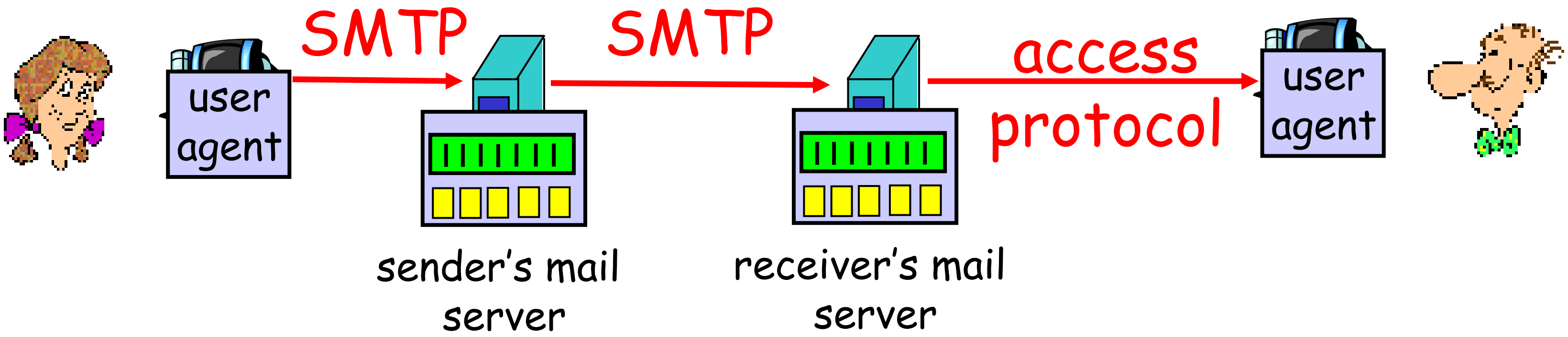
SMTP: final words

- 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 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: Hotmail , Yahoo! Mail, etc.

DNS: Domain Name System

People: many identifiers:
- SSN, name, passport #

Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., www.cs.bilkent.edu.tr - 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

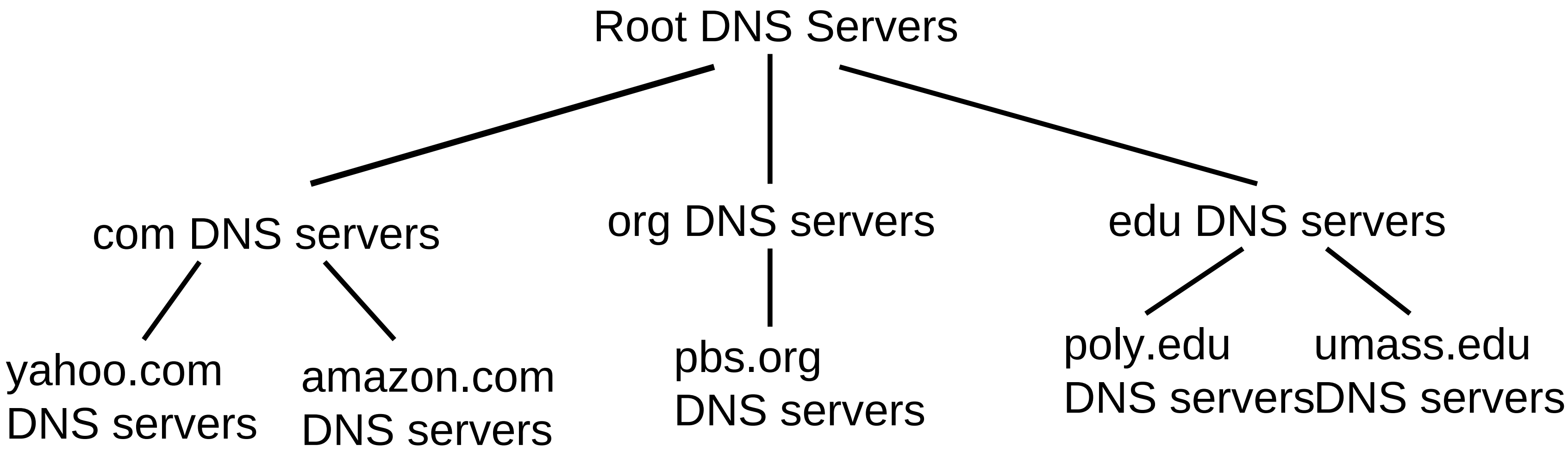
- Hostname to IP address translation
- Host aliasing
  - Canonical and alias names
- Mail server aliasing
- Load distribution
  - Replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?

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

doesn't *scale!*

Distributed, Hierarchical Database

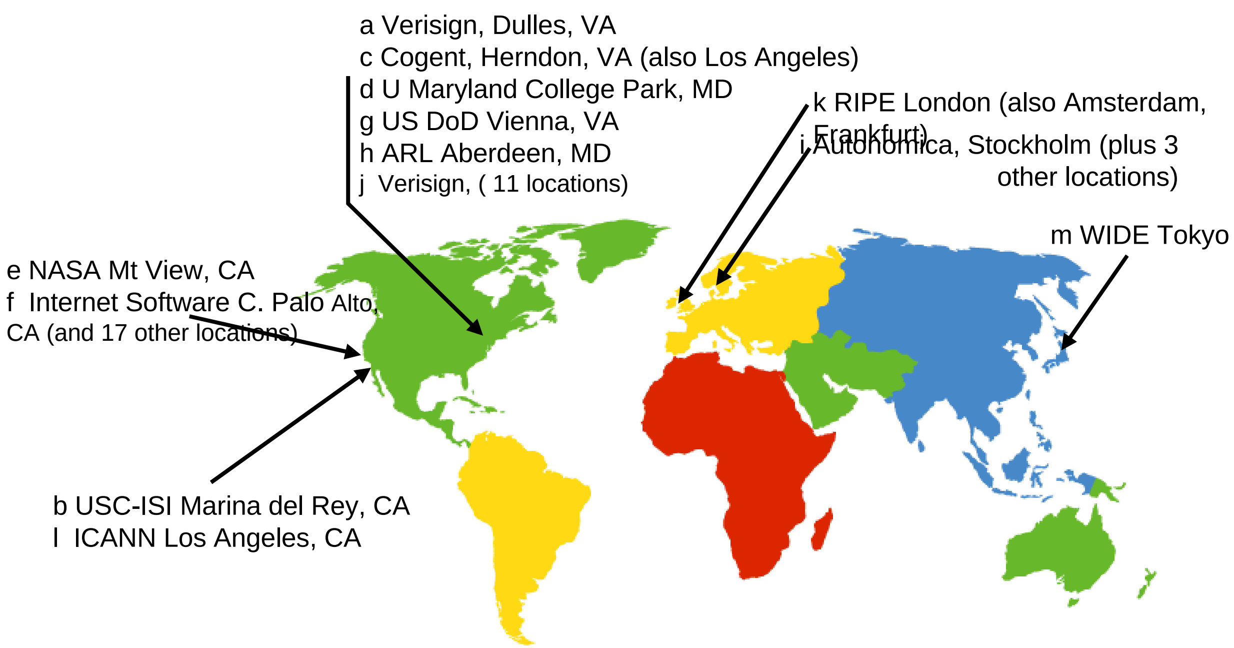


Client wants IP for www.amazon.com; 1st approx:

- 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

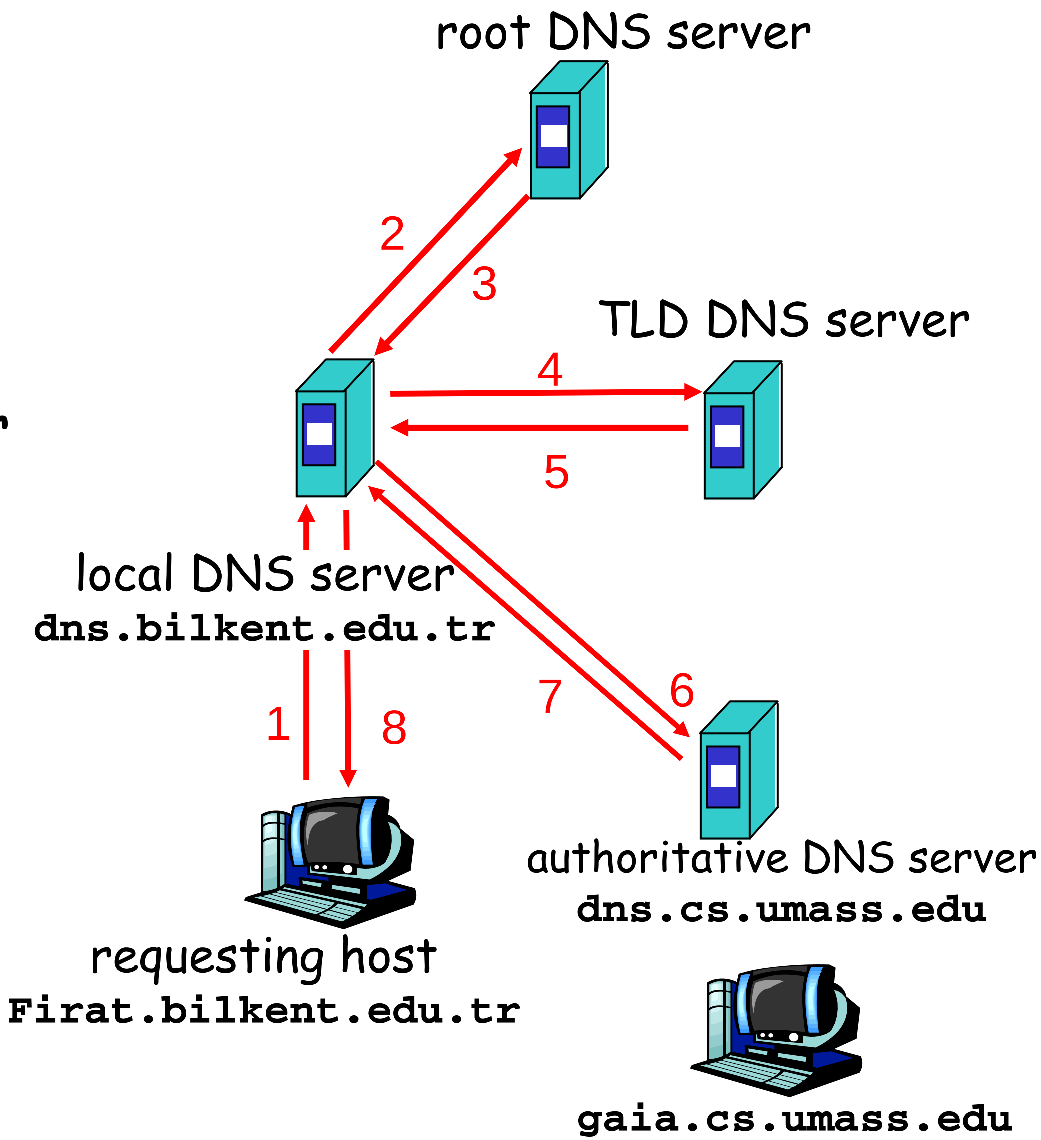
- Top-level domain (TLD) servers: responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
  - Network solutions maintains servers for com TLD
  - Educause for edu TLD
- Authoritative DNS servers: organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web and 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 a host makes a DNS query, query is sent to its local DNS server
  - Acts as a proxy, forwards query into hierarchy.

Example

- Host at firat.bilkent.edu.tr wants IP address for gaia.cs.umass.edu



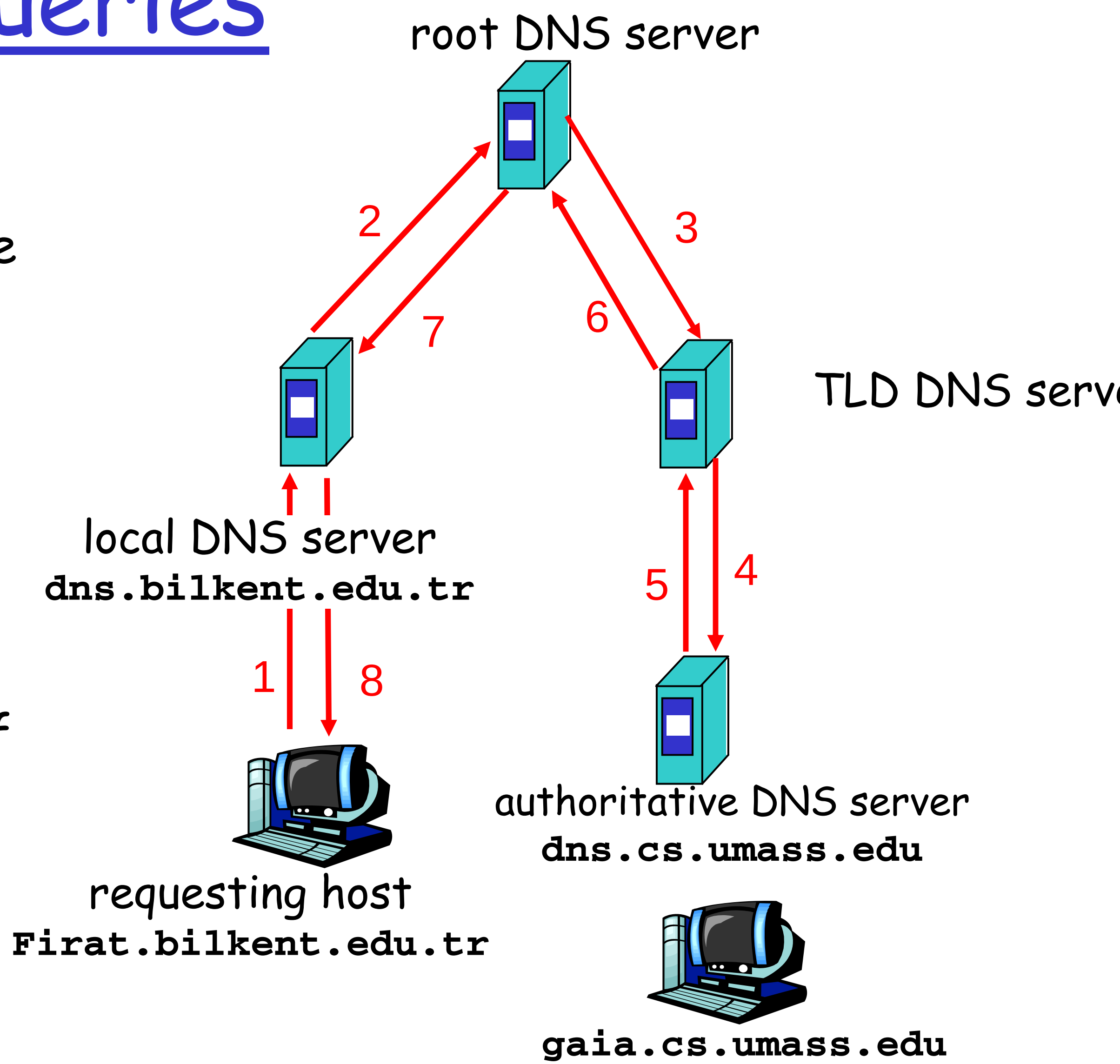
Recursive queries

recursive query:

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

iterated query:

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



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)`

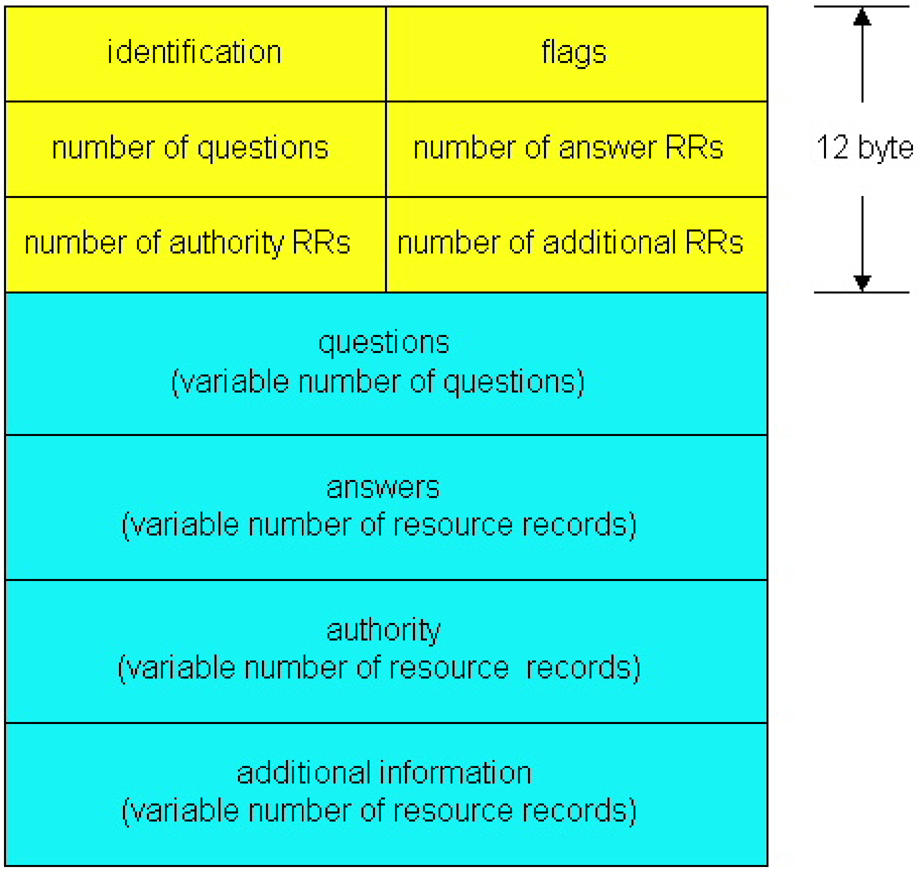
- Type=A
  - **name** is hostname
  - **value** is IP address
- Type=NS
  - **name** is domain (e.g. foo.com)
  - **value** is IP address of authoritative name server for this domain
- Type=CNAME
  - **name** is alias name for some "cannonical" (the real) name
    `www.ibm.com` is really
    `servereast.backup2.ibm.com`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

DNS protocol, messages

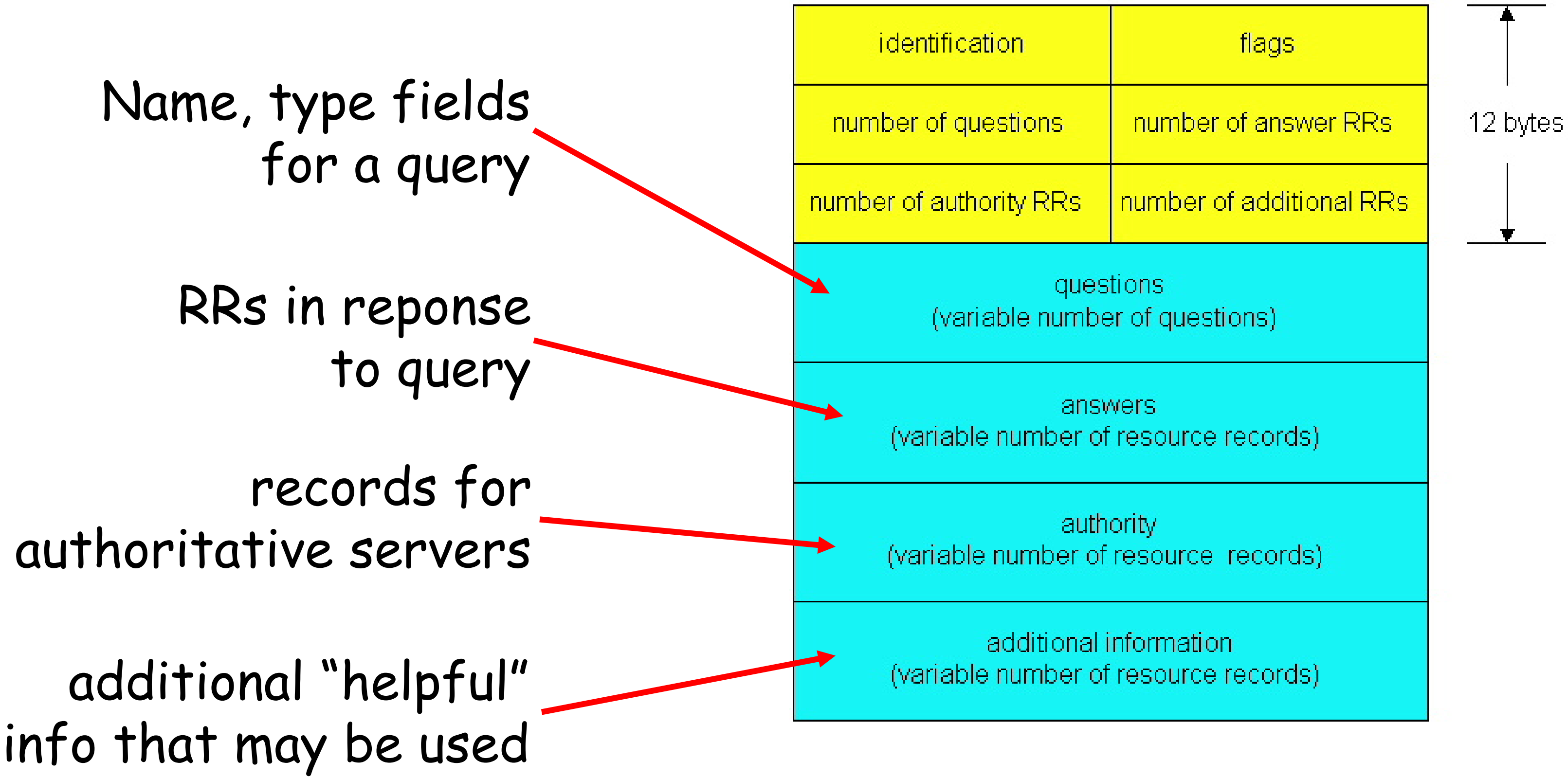
DNS protocol : *query* and *reply* messages, both with same *message format*

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



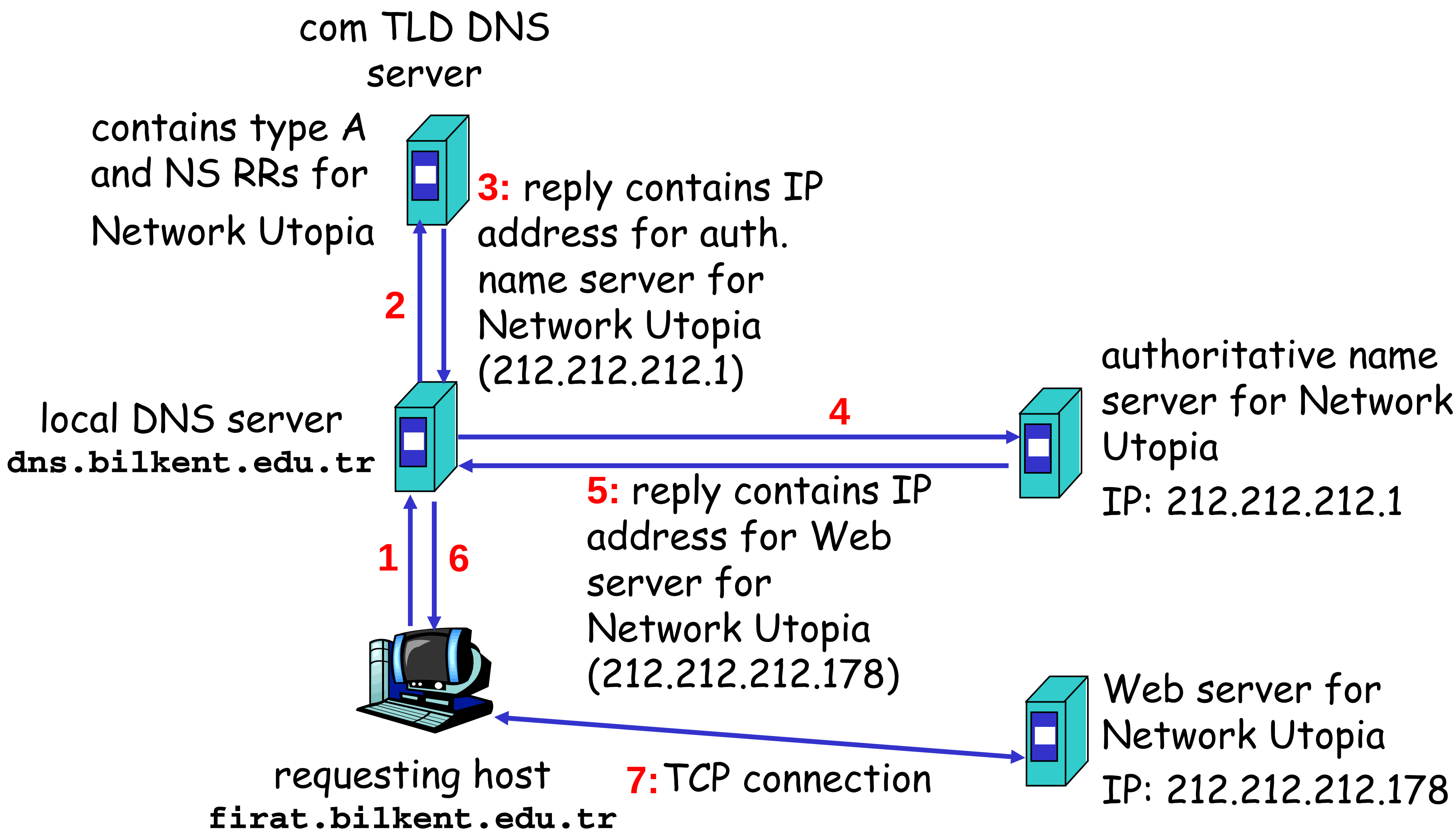
Inserting records into DNS

- Example: just created startup “Network Utopia”
- Register name networkuptopia.com at a registrar (e.g., Network Solutions)
  - Need to provide registrar with names and IP addresses of your 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)
```

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

How do people connect to Web server?



Socket programming

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

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

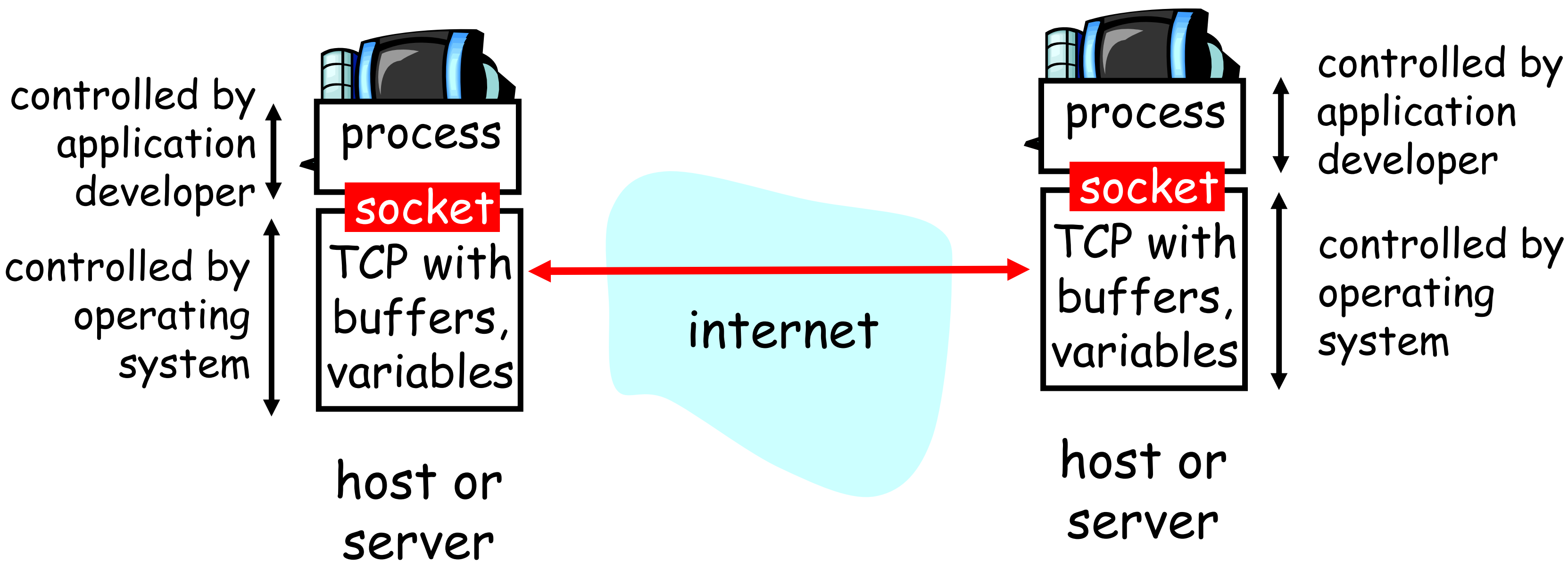
socket

a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of **bytes** from one process to another



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 client-local 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 client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint

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

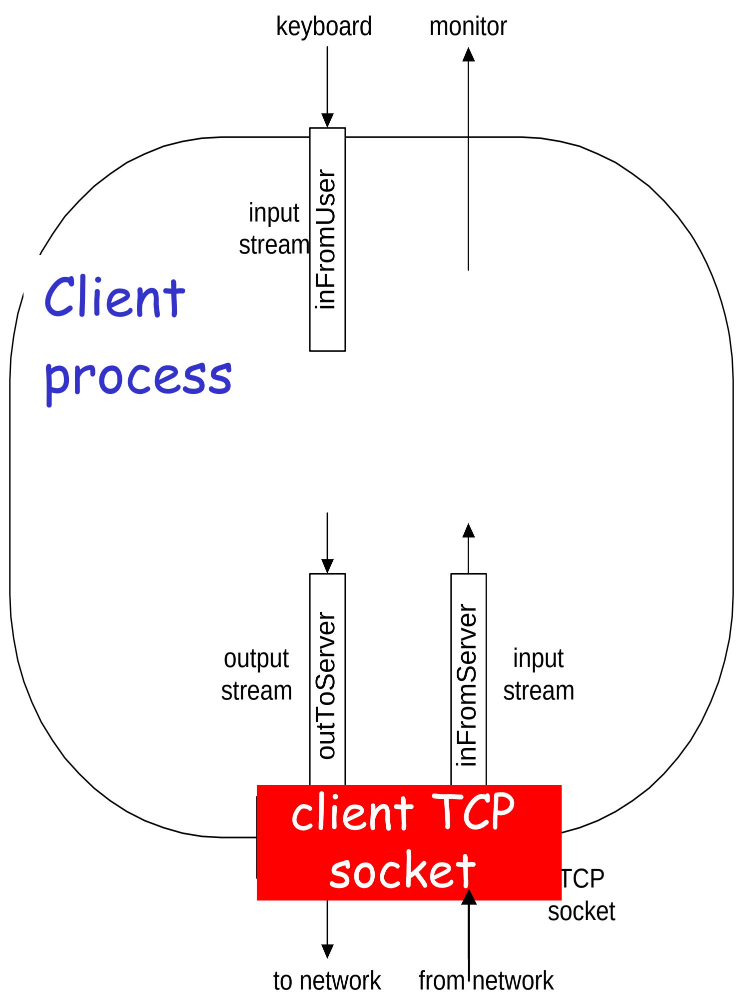
Stream jargon

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, eg, keyboard or socket.
- An output stream is attached to an output source, eg, monitor or socket.

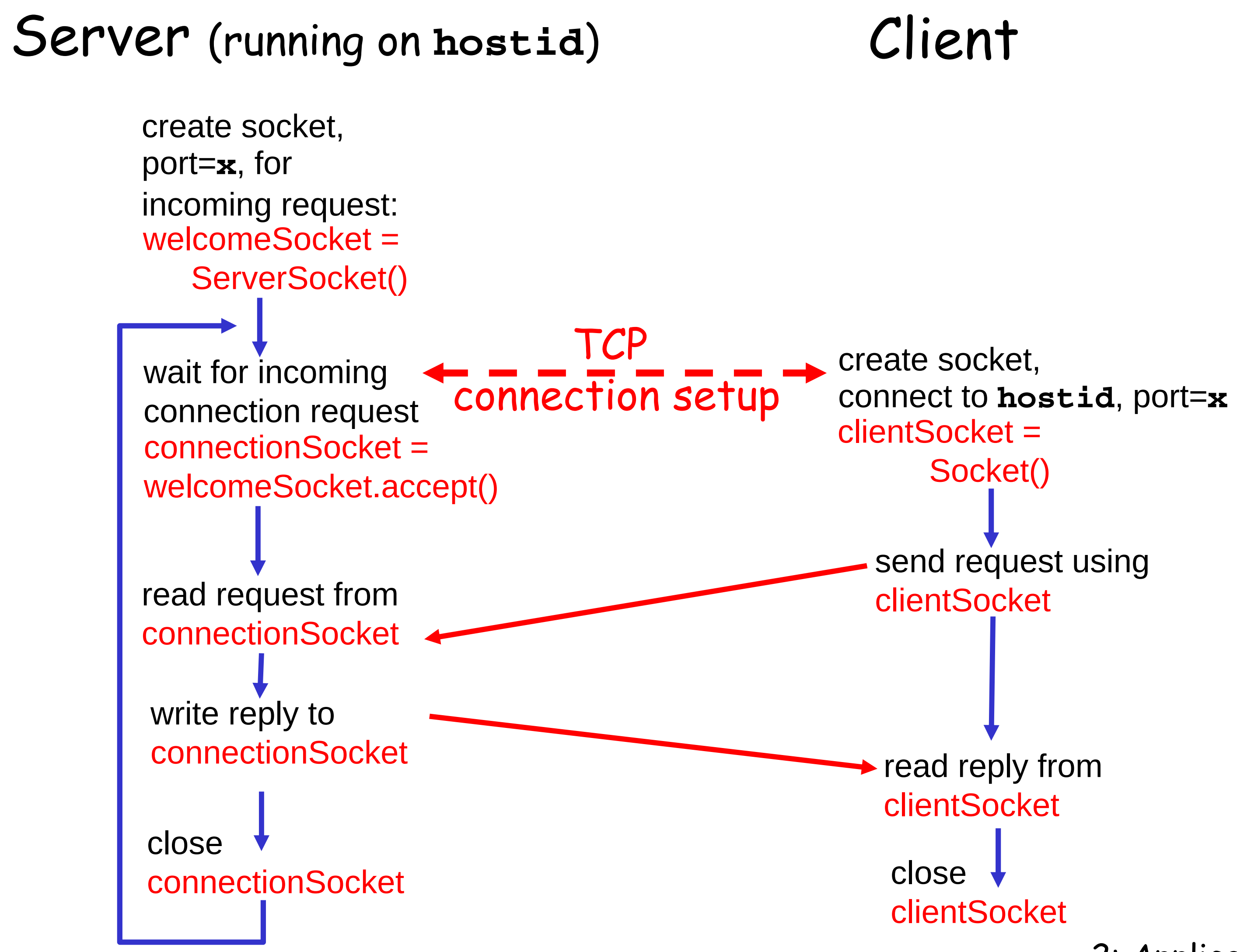
Socket programming with TCP

Example client-server app:

1) client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket (**inFromServer** stream)



Client/server socket interaction: TCP



Example: Java client (TCP)

```java
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream → BufferedReader inFromUser

Create client socket, connect to server → Socket clientSocket

Create output stream attached to socket → DataOutputStream outToServer

Example: Java client (TCP), cont.

Create input stream attached to socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
```

Send line to server

```
outToServer.writeBytes(sentence + '\n');
```

Read line from server

```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

   }
}
```

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

          Socket connectionSocket = welcomeSocket.accept();

         BufferedReader inFromClient =
            new BufferedReader(new
            InputStreamReader(connectionSocket.getInputStream()));
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

Create output stream, attached to socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

Read in line from socket

```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Write out line to socket

```
outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

End of while loop, loop back and wait for another client connection

Socket programming *with UDP*

UDP: no "connection" between client and server

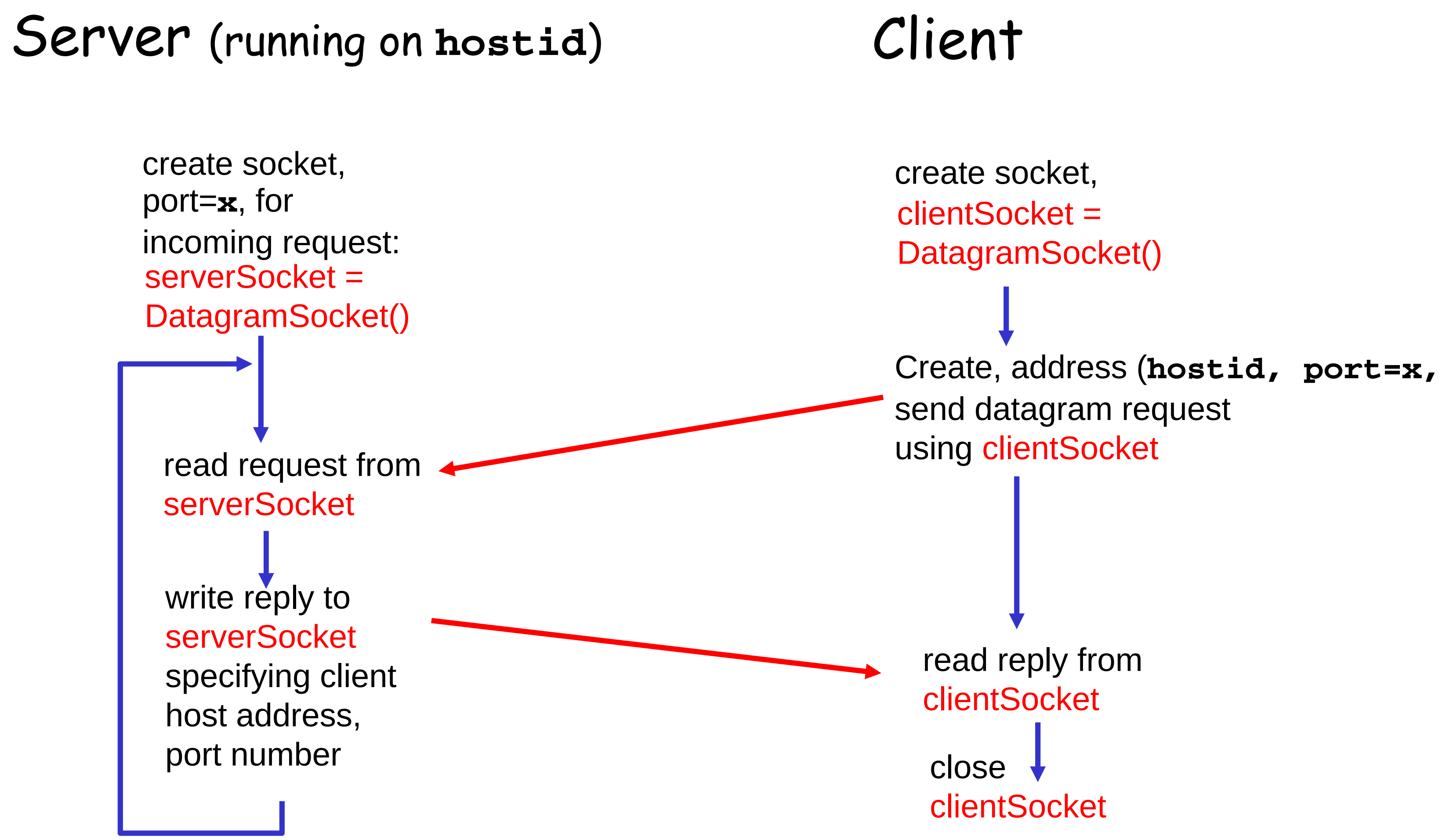
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

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

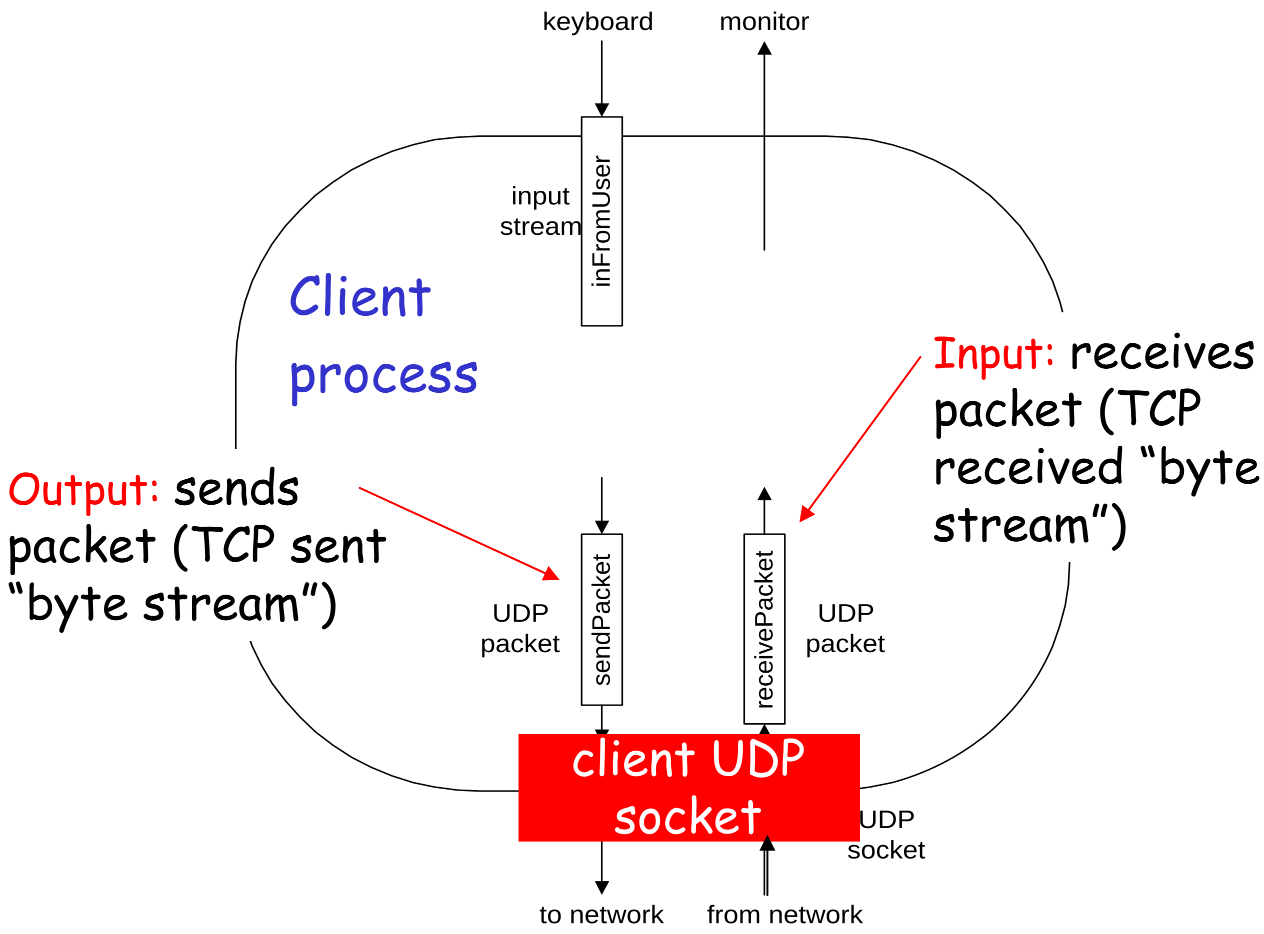
application viewpoint

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

Client/server socket interaction: UDP



Example: Java client (UDP)



Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {

      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Create input stream

Create client socket

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Send datagram to server

```
clientSocket.send(sendPacket);

DatagramPacket receivePacket =
  new DatagramPacket(receiveData, receiveData.length);
```

Read datagram from server

```
clientSocket.receive(receivePacket);

String modifiedSentence =
   new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}
}
```

Example: Java server (UDP)

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {

      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {

          DatagramPacket receivePacket =
             new DatagramPacket(receiveData, receiveData.length);

           serverSocket.receive(receivePacket);
```

Example: Java server (UDP), cont

```
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

        String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress,
                port);

serverSocket.send(sendPacket);
    }
  }
}
```

Get IP addr port #, of sender

Create datagram to send to client

Write out datagram to socket

End of while loop, loop back and wait for another datagram

Socket programming: references

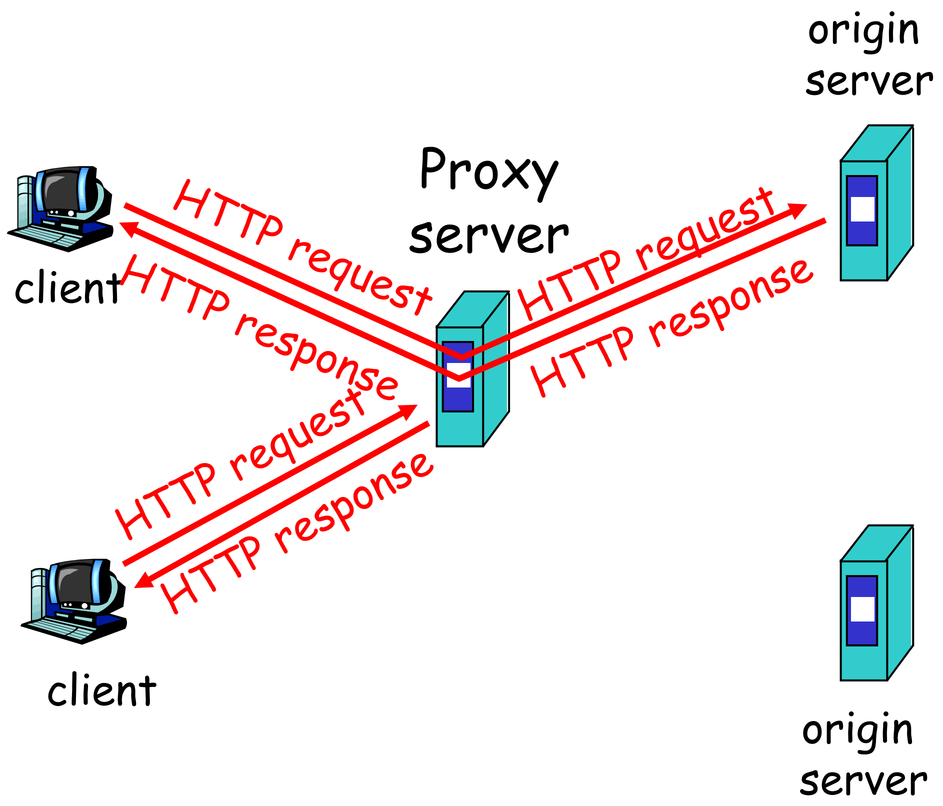
Java-tutorials:

- "All About Sockets" (Sun tutorial), http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html
- "Socket Programming in Java: a tutorial," http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html

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
- Cache can do up-to-date check using `If-modified-since` HTTP header
  - Issue: should cache take risk and deliver cached object without checking?
  - Heuristics are used.
- 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

Caching example (1)

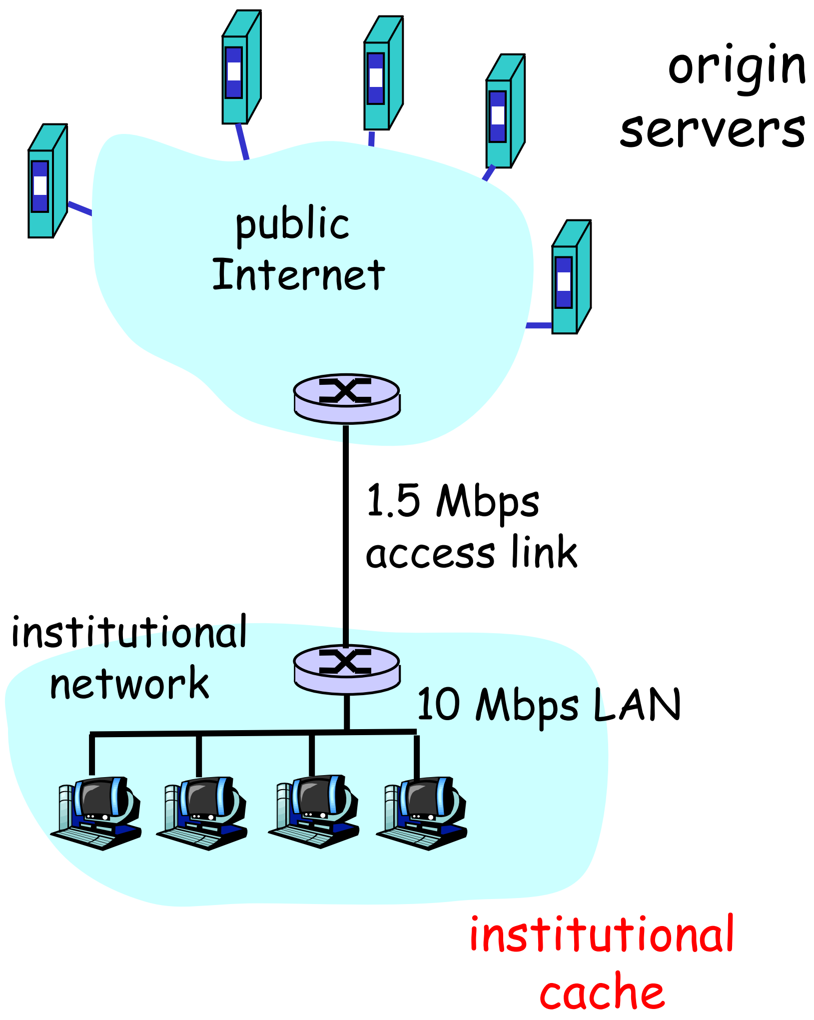
Assumptions

- average object size = 100,000 bits
- avg. request rate from institution's browser to origin serves = 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 (2)

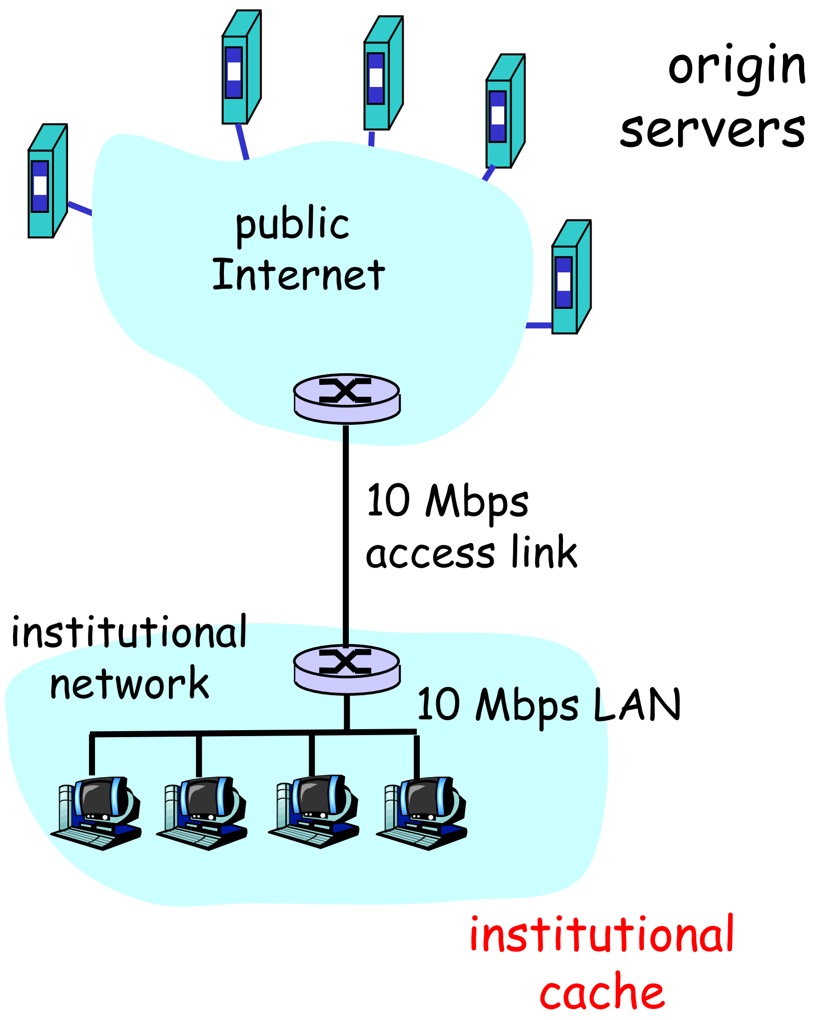
Possible solution

- increase bandwidth of access link to, say, 10 Mbps

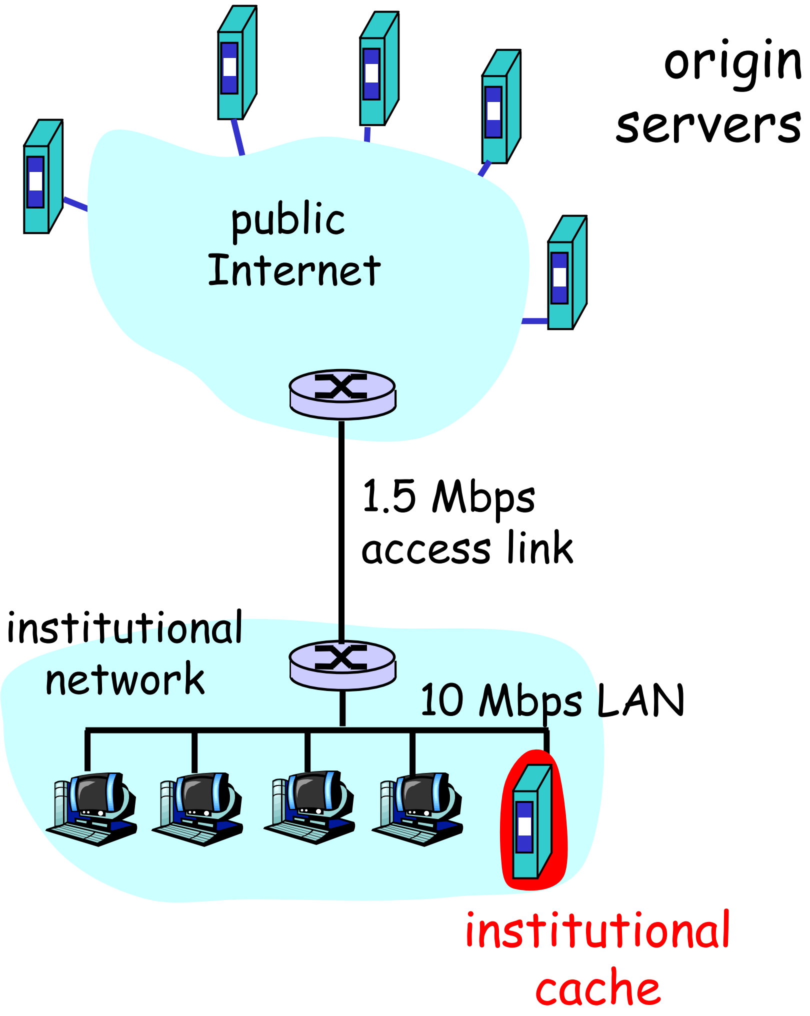
Consequences

- 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 (3)



Install cache

- suppose hit rate is .4

Consequence

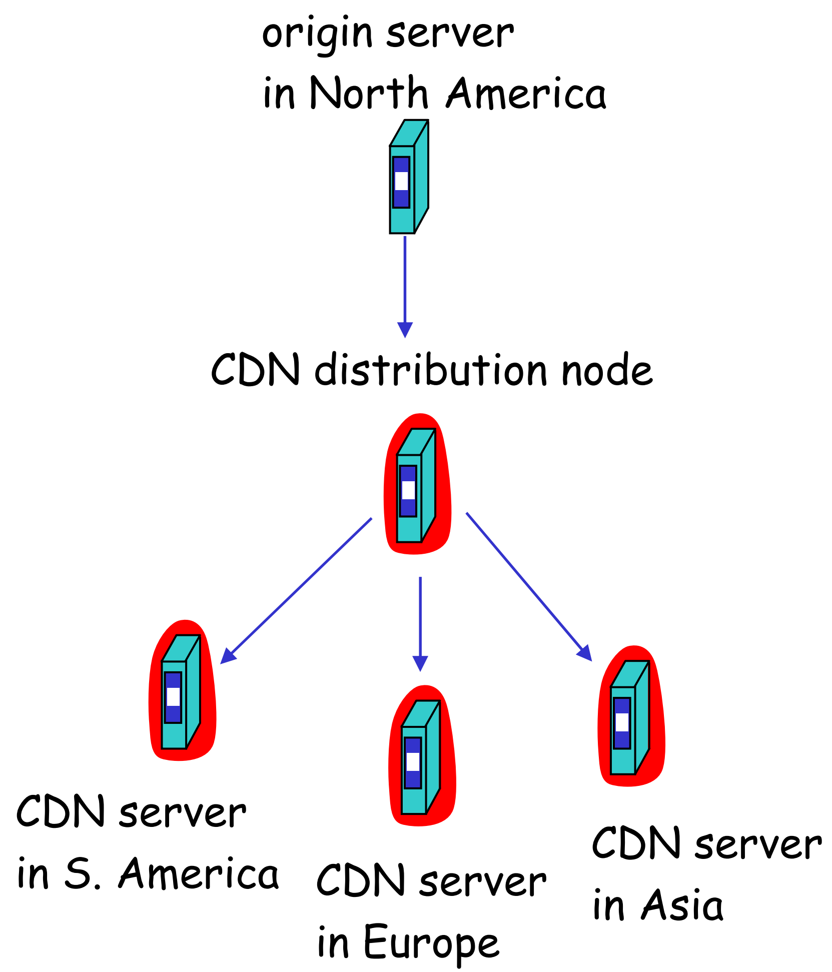
- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)

Content distribution networks (CDNs)

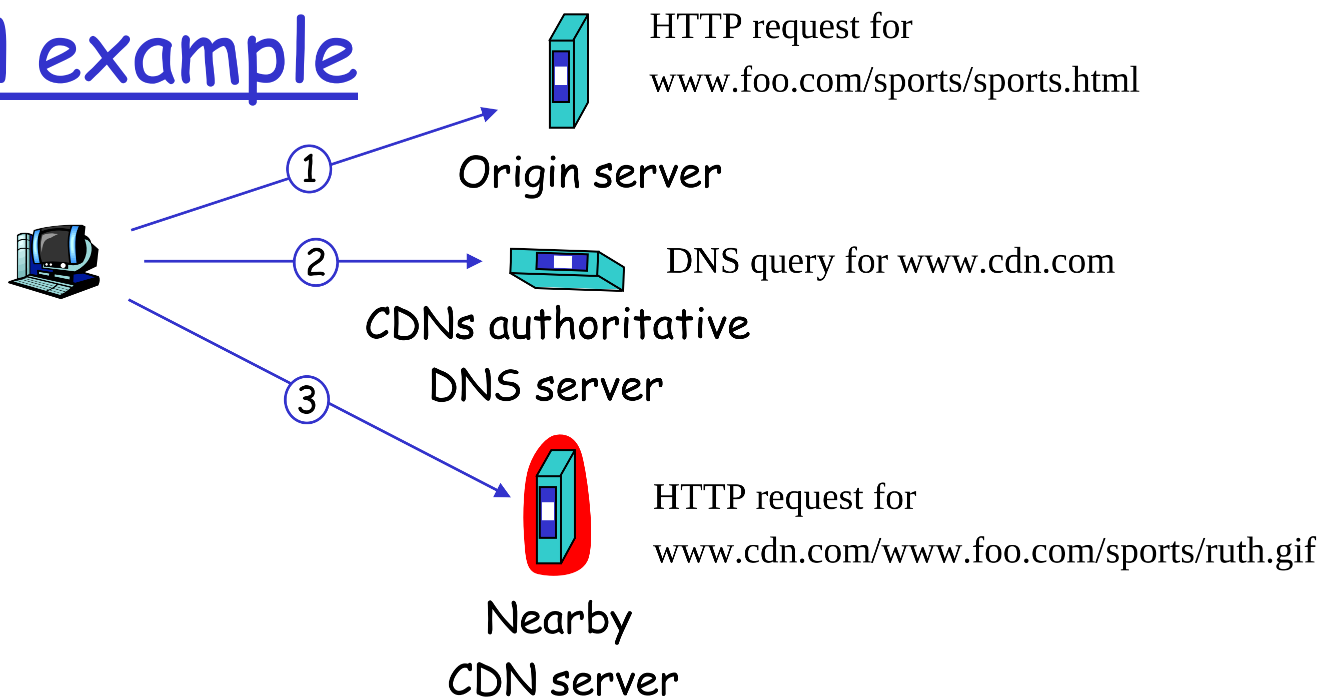
- The content providers are the CDN customers.

Content replication

- CDN company installs hundreds of CDN servers throughout Internet
  - in lower-tier ISPs, close to users
- CDN replicates its customers' content in CDN servers. When provider updates content, CDN updates servers



CDN example



origin server

- www.foo.com
- distributes HTML
- Replaces:

  http://www.foo.com/sports.ruth.gif

  with

  http://www.cdn.com/www.foo.com/sports/ruth.gif

CDN company

- cdn.com
- distributes gif files
- uses its authoritative DNS server to route redirect requests

More about CDNs

routing requests

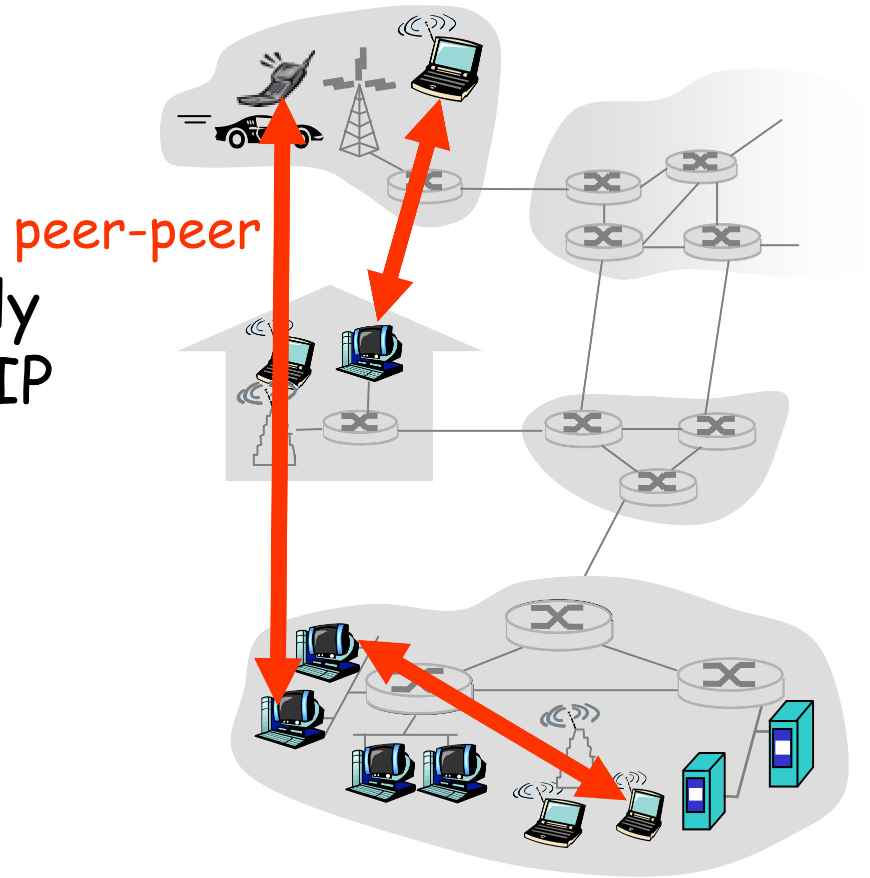
- CDN creates a "map", indicating distances from leaf ISPs and CDN nodes
- when query arrives at authoritative DNS server:
  - server determines ISP from which query originates
  - uses "map" to determine best CDN server

not just Web pages

- streaming stored audio/video
- streaming real-time audio/video
  - CDN nodes create application-layer overlay network

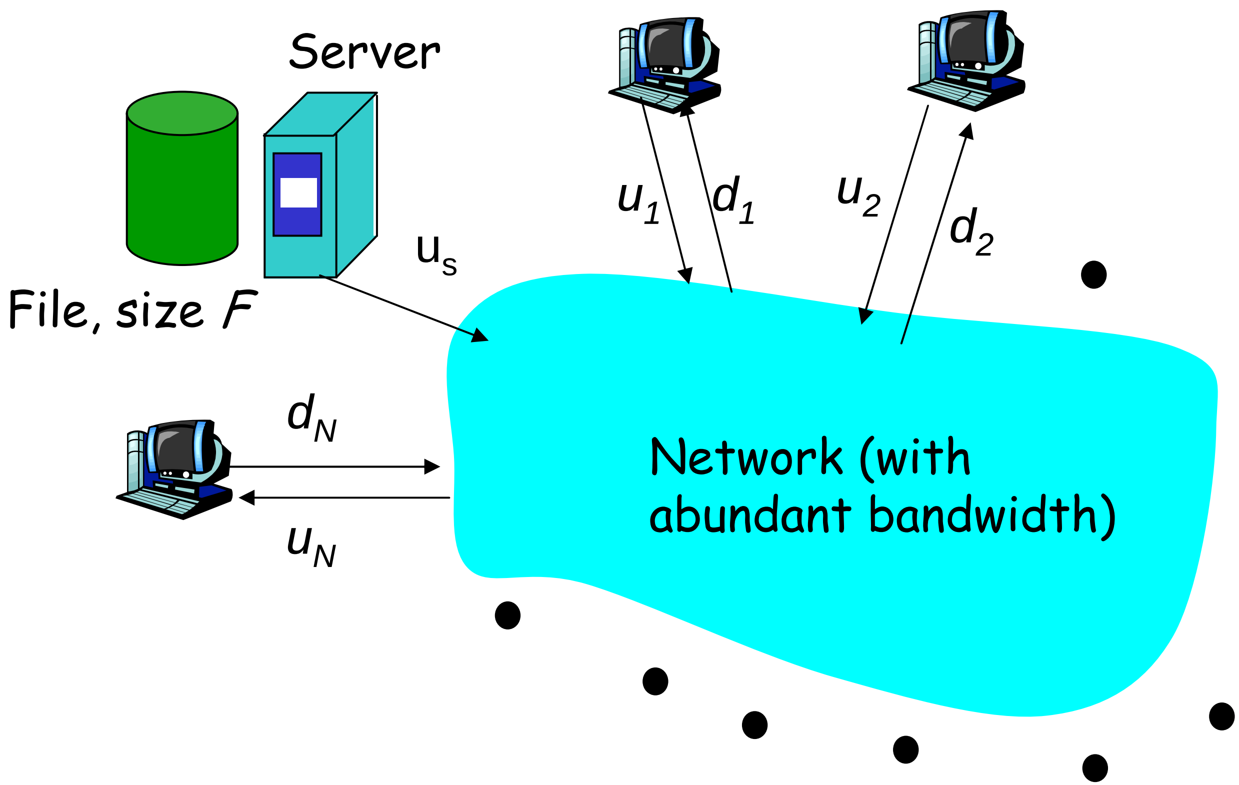
Pure P2P architecture

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



File Distribution: Server-Client vs P2P

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



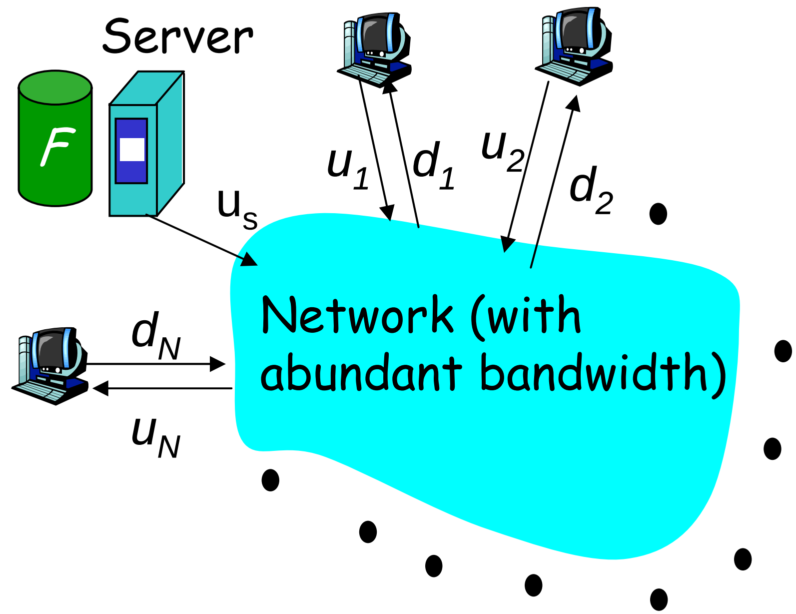
u_s: server upload bandwidth

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

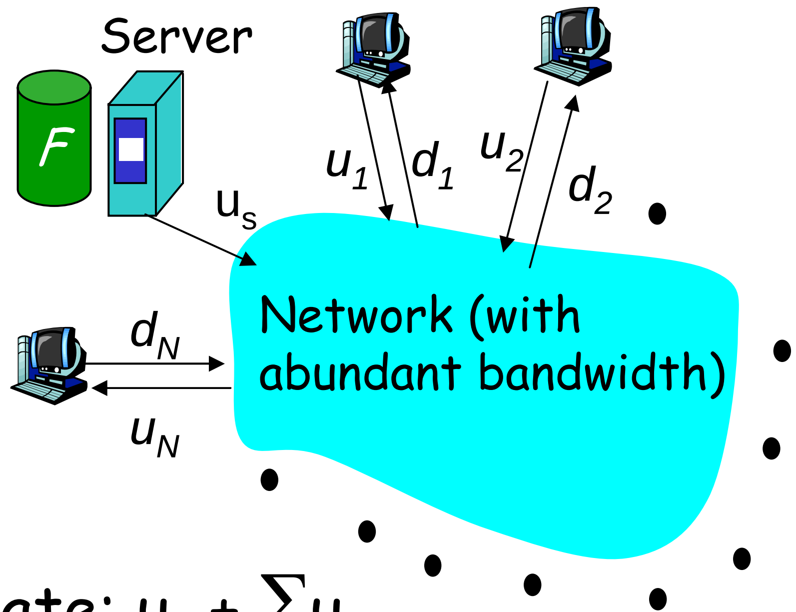


Time to distribute F to N clients using client/server approach $= d_{cs} = \max_i \{ NF/u_s, F/\min_i(d_i) \}$

increases linearly in N (for large N)

File distribution time: P2P

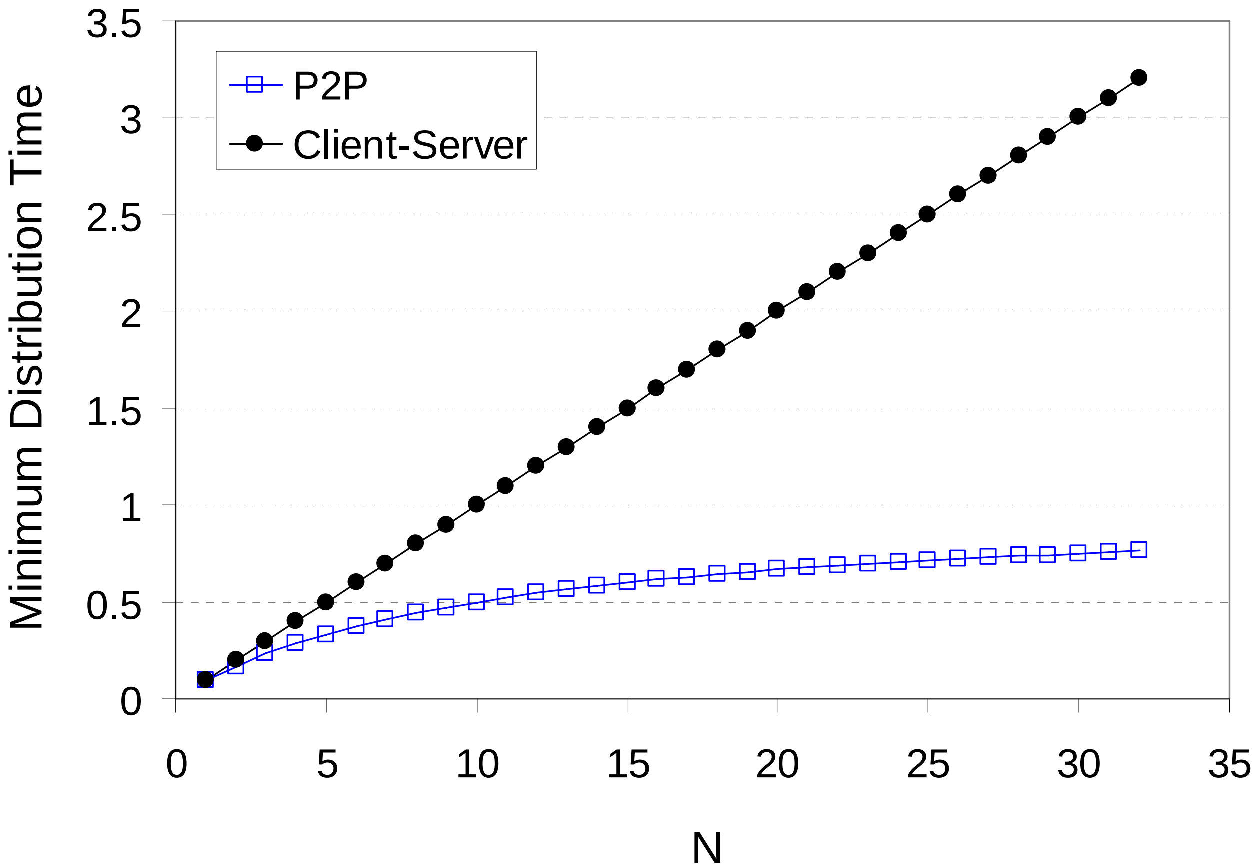
- server must send one copy: F/u_s time
- client i takes F/d_i time to download
- NF bits must be downloaded (aggregate)
  - fastest possible upload rate: $u_s + \sum u_i$



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

Server-client vs. P2P: example

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



Searching for Information- Query flooding: Gnutella

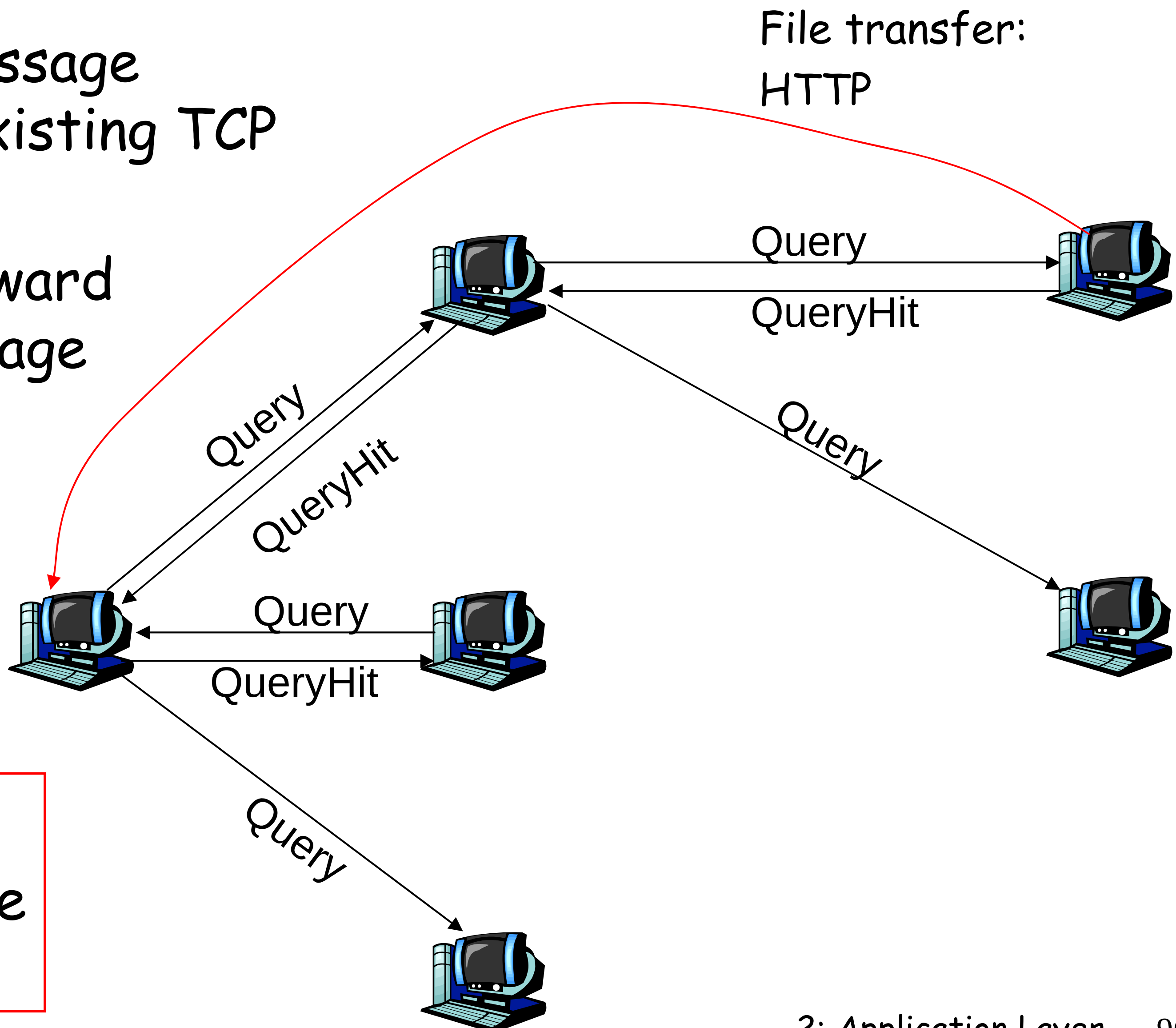
- fully distributed
  - no central server
- public domain protocol
- many Gnutella clients implementing protocol

overlay network: graph

- edge between peer X and Y if there's a TCP connection
- all active peers and edges is overlay net
- Edge is not a physical link
- Given peer will typically be connected with < 10 overlay neighbors

Gnutella: protocol

- Query message sent over existing TCP connections
- peers forward Query message
- QueryHit sent over reverse path



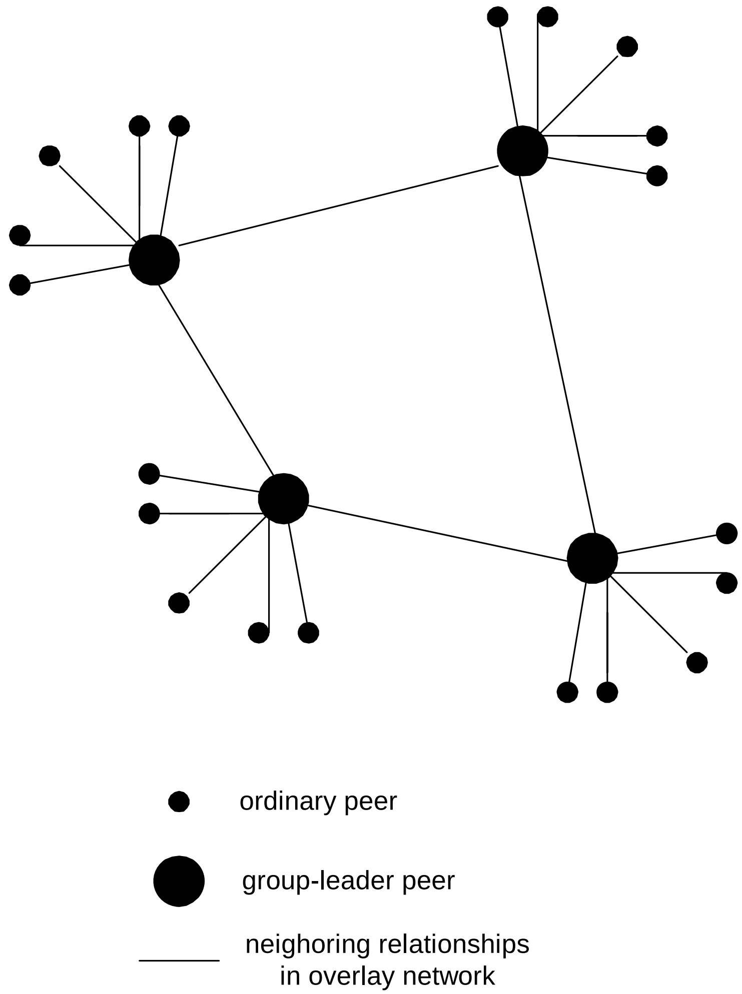
Scalability: limited scope flooding

Gnutella: Peer joining

1. Joining peer X must find some other peer in Gnutella network: use list of candidate peers
2. X sequentially attempts to make TCP with peers on list until connection setup with Y
3. X sends Ping message to Y; Y forwards Ping message.
4. All peers receiving Ping message respond with Pong message
5. X receives many Pong messages. It can then setup additional TCP connections

Exploiting heterogeneity: KaZaA

- Each peer is either a group leader or assigned to a group leader.
  - TCP connection between peer and its group leader.
  - TCP connections between some pairs of group leaders.
- Group leader tracks the content in all its children.



KaZaA: Querying

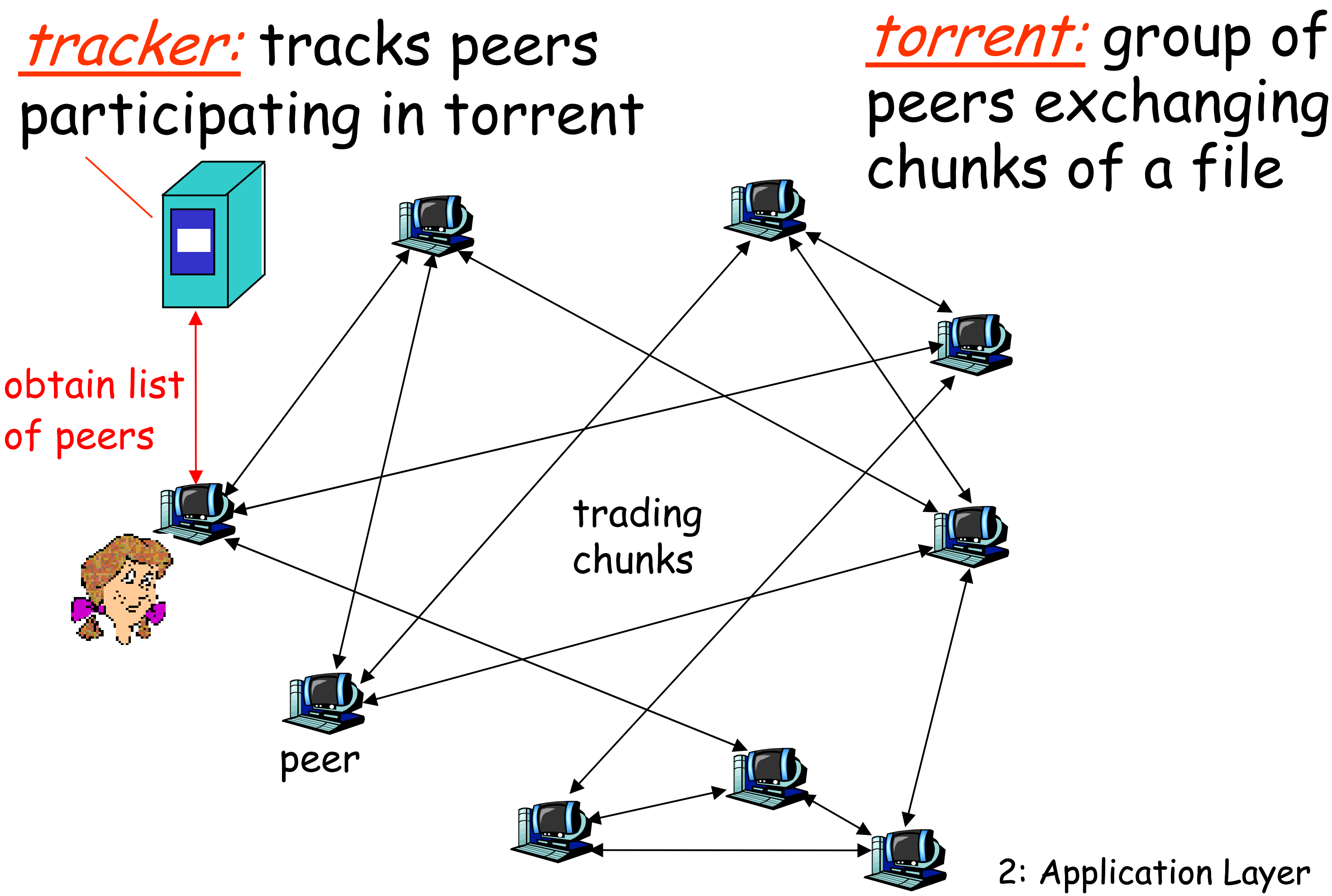
- Each file has a hash and a descriptor
- Client sends keyword query to its group leader
- Group leader responds with matches:
  - For each match: metadata, hash, IP address
- If group leader forwards query to other group leaders, they respond with matches
- Client then selects files for downloading
  - HTTP requests using hash as identifier sent to peers holding desired file

Kazaa tricks

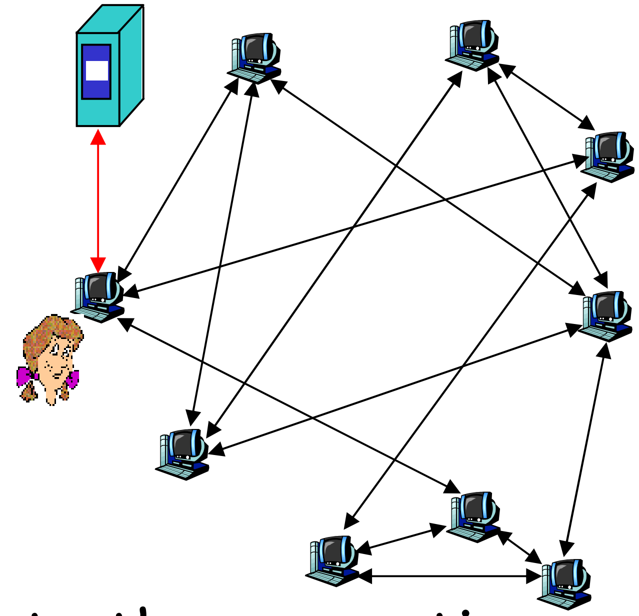
- Limitations on simultaneous uploads
- Request queuing
- Incentive priorities
- Parallel downloading

P2P Case Study: BitTorrent

P2P file distribution



BitTorrent (1)



- file divided into 256KB *chunks*.
- peer joining torrent:
  - has no chunks, but will accumulate them over time
  - registers with tracker to get list of peers, connects to subset of peers (“neighbors”)
- while downloading, peer uploads chunks to other peers.
- peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain

BitTorrent (2)

Pulling Chunks

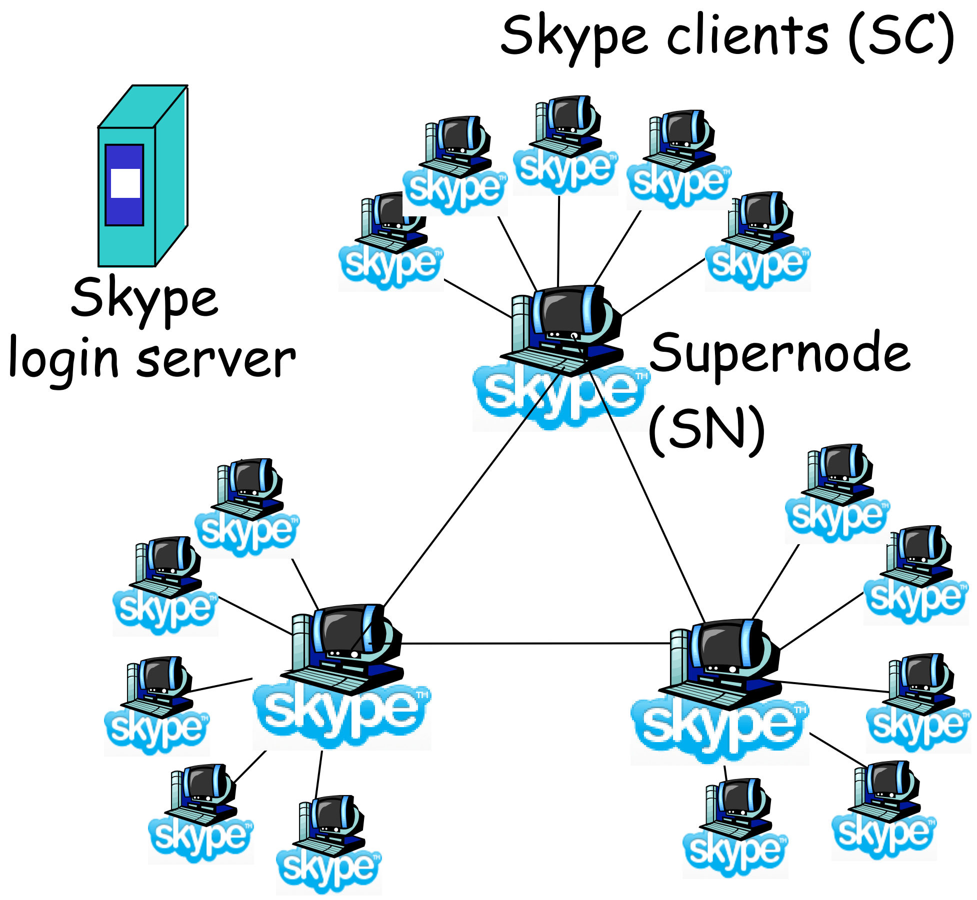
- at any given time, different peers have different subsets of file chunks
- periodically, a peer (Alice) asks each neighbor for list of chunks that they have.
- Alice sends requests for her missing chunks
  - rarest first

Sending Chunks: tit-for-tat

- Alice sends chunks to four neighbors currently sending her chunks *at the highest rate*
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - newly chosen peer may join top 4
  - "optimistically unchoke"

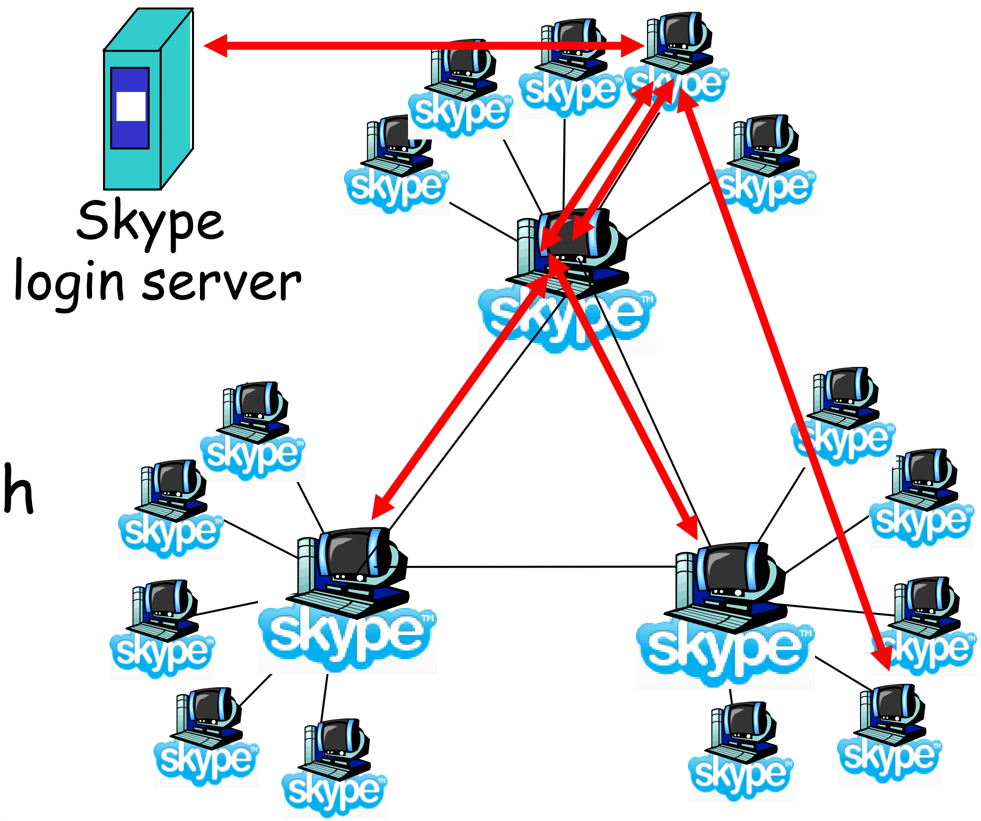
P2P Case study: Skype

- 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



Skype: making a call

- User starts Skype
- SC registers with SN
  - list of bootstrap SNs
- SC logs in (authenticate)
- Call: SC contacts SN with callee ID
  - SN contacts other SNs (unknown protocol, maybe flooding) to find addr of callee; returns addr to SC
- SC directly contacts callee



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