

Chapter 2

Application Layer

Chapter 2: Application layer

- ❑ 2.1 Principles of network applications
- ❑ 2.2 Web and HTTP
- ❑ 2.3 Electronic Mail
 - SMTP, POP3, IMAP
- ❑ 2.4 Socket programming with TCP
- ❑ 2.5 Socket programming with UDP
- ❑ 2.6 DNS
- ❑ 2.7 P2P file sharing

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

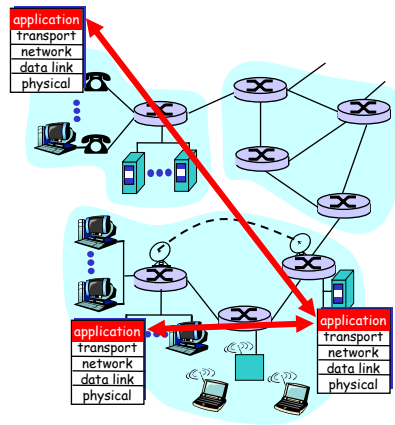
Creating a network app

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



2: Application Layer 5

Chapter 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 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

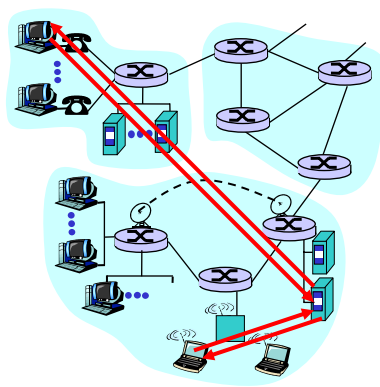
2: Application Layer 6

Application architectures

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

2: Application Layer 7

Client-server architecture



server:

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

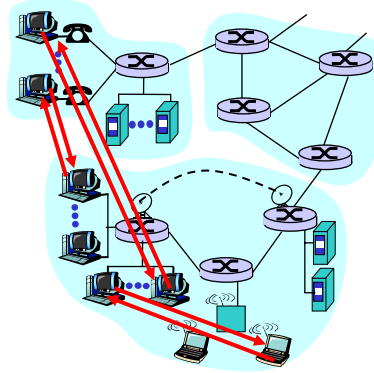
clients:

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

2: Application Layer 8

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

2: Application Layer 9

Hybrid of client-server and P2P

Napster

- File transfer P2P
- File search centralized:
 - Peers register content at central server
 - Peers query same central server to locate content

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 buddies

2: Application Layer 10

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**

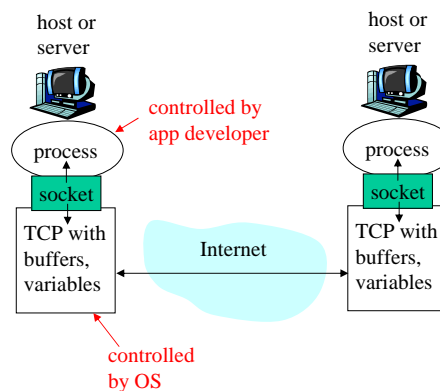
Client process: process that initiates communication

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

- ❑ For a process to receive messages, it must have an identifier
- ❑ A 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
- ❑ **More on this later**

2: Application Layer 13

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

2: Application Layer 14

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

2: Application Layer 15

Transport service requirements of common apps

<u>Application</u>	<u>Data loss</u>	<u>Bandwidth</u>	<u>Time Sensitive</u>
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

2: Application Layer 16

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

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?

2: Application Layer 17

Internet apps: application, transport protocols

<u>Application</u>	<u>Application layer protocol</u>	<u>Underlying transport protocol</u>
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

www.ietf.org

Internet engineering task force

2: Application Layer 18

Chapter 2: Application layer

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

2: Application Layer 19

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.someschool.edu/someDept/pic.gif`

host name

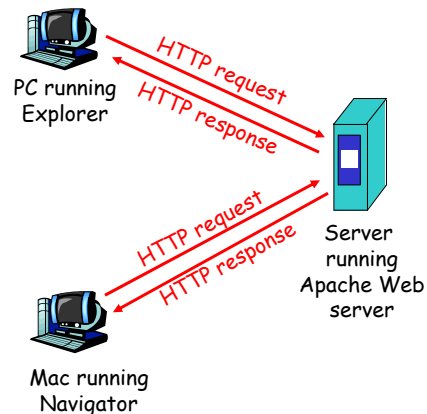
path name

2: Application Layer 20

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



2: Application Layer 21

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! aside

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

2: Application Layer 22

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

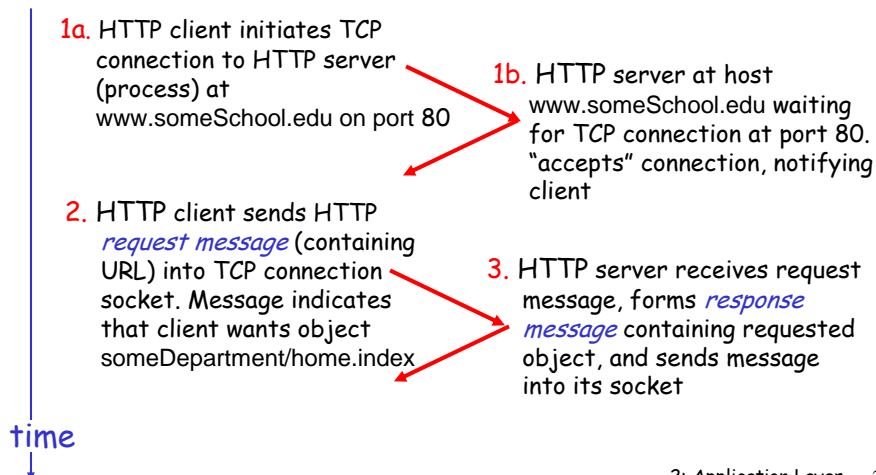
2: Application Layer 23

Nonpersistent HTTP

Suppose user enters URL

`www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)



2: Application Layer 24

Nonpersistent HTTP (cont.)

- time ↓
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

2: Application Layer 25

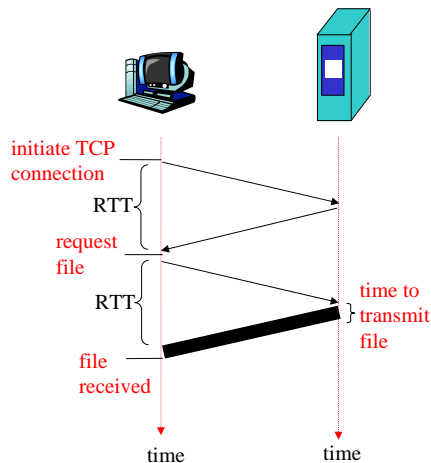
Response time modeling

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

Response time:

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

total = 2RTT + transmit time



2: Application Layer 26

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

2: Application Layer 27

HTTP request message

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

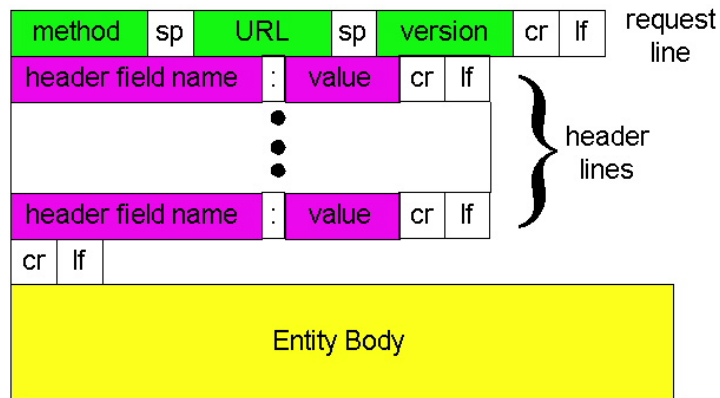
request line
(GET, POST,
HEAD commands) → GET /somedir/page.html HTTP/1.1

header
lines → Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr

Carriage return
line feed
indicates end
of message → (extra carriage return, line feed)

2: Application Layer 28

HTTP request message: general format



2: Application Layer 29

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`

2: Application Layer 30

Method types

HTTP/1.0

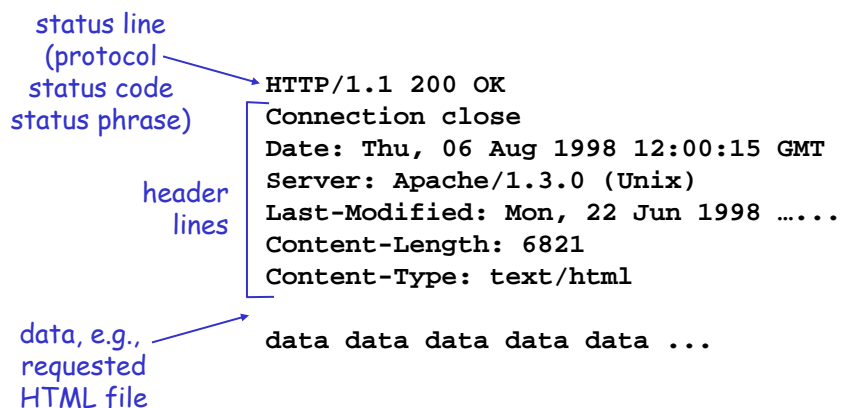
- GET
- POST
- HEAD
 - asks server to leave requested object out of response

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

2: Application Layer 31

HTTP response message



2: Application Layer 32

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

2: Application Layer 33

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet cis.poly.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at cis.poly.edu. Anything typed in sent to port 80 at cis.poly.edu

2. Type in a GET HTTP request:

```
GET /~ross/ HTTP/1.1  
Host: cis.poly.edu
```

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

3. Look at response message sent by HTTP server!

2: Application Layer 34

User-server state: cookies

Many major Web sites use cookies

Four components:

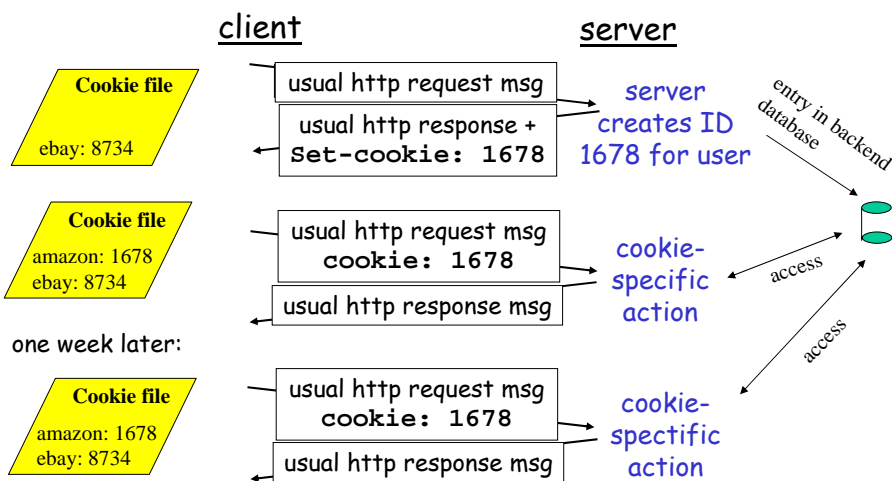
- 1) cookie header line in the HTTP response message
- 2) cookie header line in HTTP request message
- 3) cookie file kept on user's host and managed by user's browser
- 4) back-end database at Web site

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

2: Application Layer 35

Cookies: keeping "state" (cont.)



2: Application Layer 36

Cookies (continued)

What cookies can bring:

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

aside

Cookies and privacy:

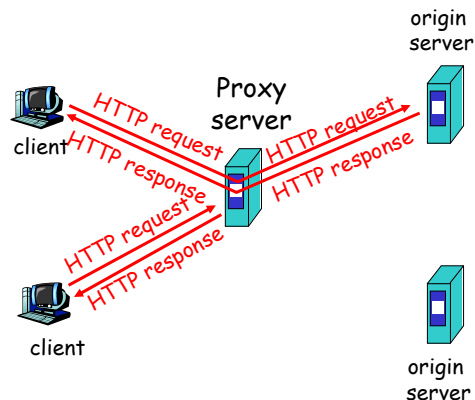
- ❑ cookies permit sites to learn a lot about you
- ❑ you may supply name and e-mail to sites
- ❑ search engines use redirection & cookies to learn yet more
- ❑ advertising companies obtain info across sites

2: Application Layer 37

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



2: Application Layer 38

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)

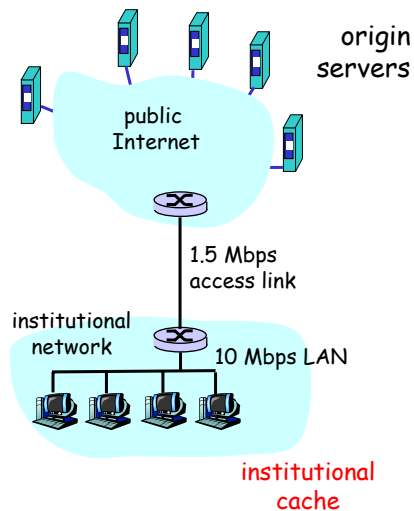
Caching example

Assumptions

- ❑ average object size = 100,000 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



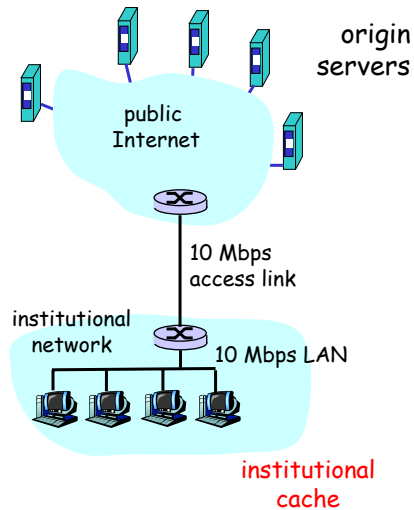
Caching example (cont)

Possible solution

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

Consequences

- utilization on LAN = 15%
- utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay
= 2 sec + msec + msec
- often a costly upgrade



2: Application Layer 41

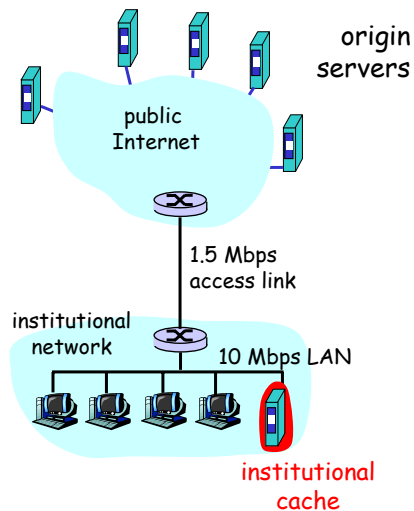
Caching example (cont)

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)
- total avg delay = Internet delay + access delay + LAN delay
= .6*(2.01) secs
.4*(.01) secs < 1.4 secs



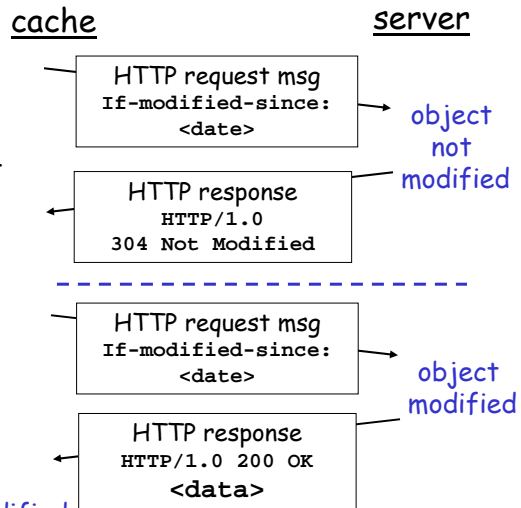
2: Application Layer 42

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

Improvement???

Push (webservice pushes the modified content to the web cache)



2: Application Layer 43

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 **Electronic Mail**
 - SMTP, POP3, IMAP
- 2.4 Socket programming with TCP
- 2.5 Socket programming with UDP
- 2.6 DNS
- 2.7 P2P file sharing

2: Application Layer 44

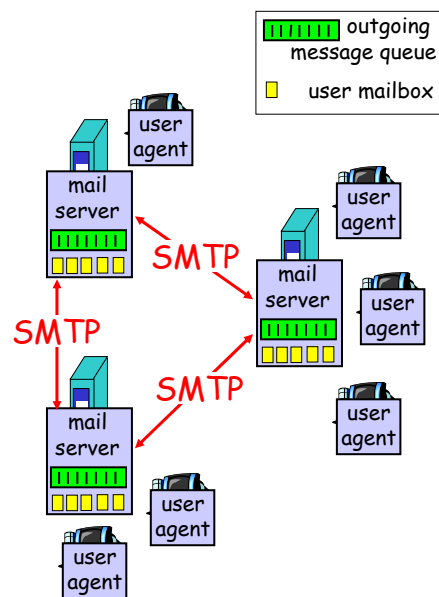
Electronic Mail

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

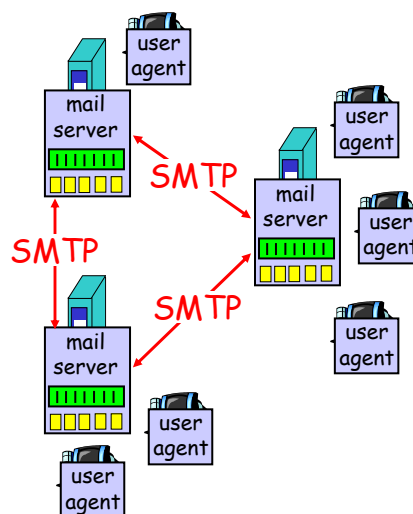


2: Application Layer 45

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



2: Application Layer 46

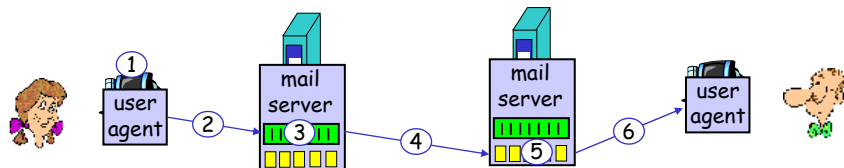
Electronic Mail: SMTP [RFC 2821]

- ❑ 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

2: Application Layer 47

Scenario: Alice sends message to Bob

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



2: Application Layer 48

Sample SMTP interaction

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

2: Application Layer 49

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)

2: Application Layer 50

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:

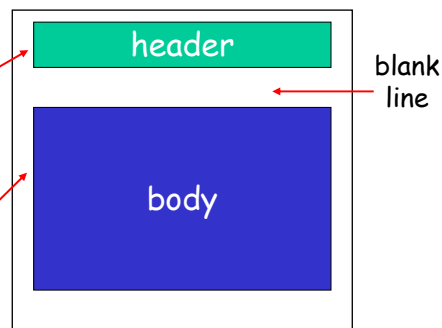
- ❑ HTTP: pull
- ❑ SMTP: push
- ❑ both have ASCII command/response interaction, status codes
- ❑ HTTP: each object encapsulated in its own response msg
- ❑ SMTP: multiple objects sent in multipart msg

2: Application Layer 51

Mail message format (not covered in lecture - read by yourself)

SMTP: protocol for exchanging email msgs
RFC 822: standard for text message format:

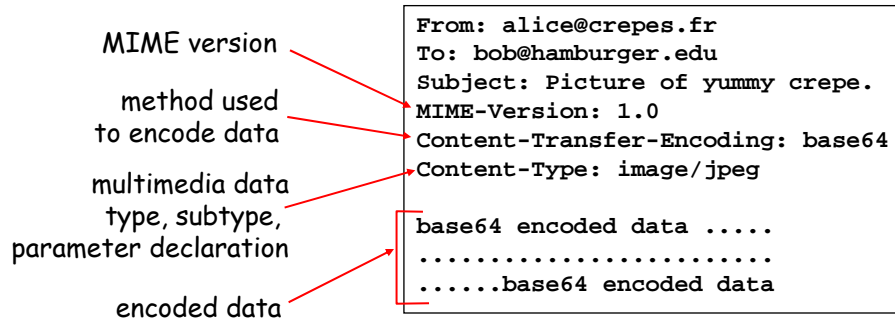
- ❑ header lines, e.g.,
 - To:
 - From:
 - Subject:*different from SMTP commands!*
- ❑ body
 - the "message", ASCII characters only



2: Application Layer 52

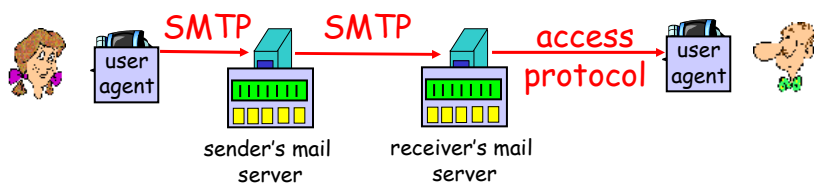
Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type



2: Application Layer 53

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.

2: Application Layer 54

POP3 protocol (not covered in lecture - read by yourself)

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
```

2: Application Layer 55

POP3 (more) and IMAP (not covered in lecture - read by yourself)

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

2: Application Layer 56

Chapter 2: Application layer

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

2: Application Layer 57

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

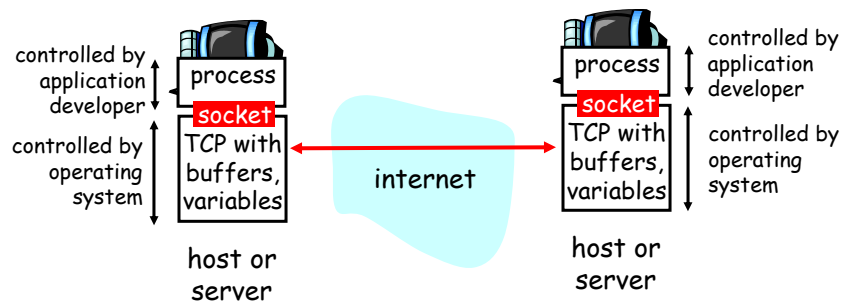
a *host-local, application-created, OS-controlled* interface (a "door") into which application process can **both send and receive** messages to/from another application process

2: Application Layer 58

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



2: Application Layer 59

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

2: Application Layer 60

Stream jargon

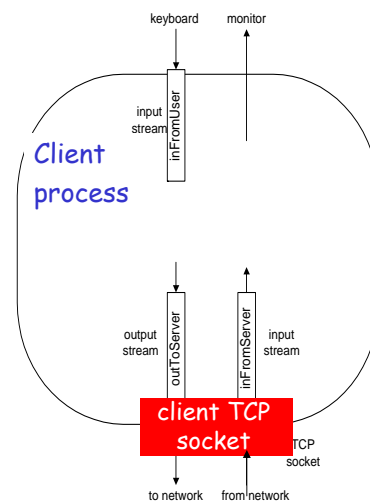
- A **stream** is a sequence of characters that flow into or out of a process.
- An **input stream** is attached to some input source for the process, eg, keyboard or socket.
- An **output stream** is attached to an output source, eg, monitor or socket.

2: Application Layer 61

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)

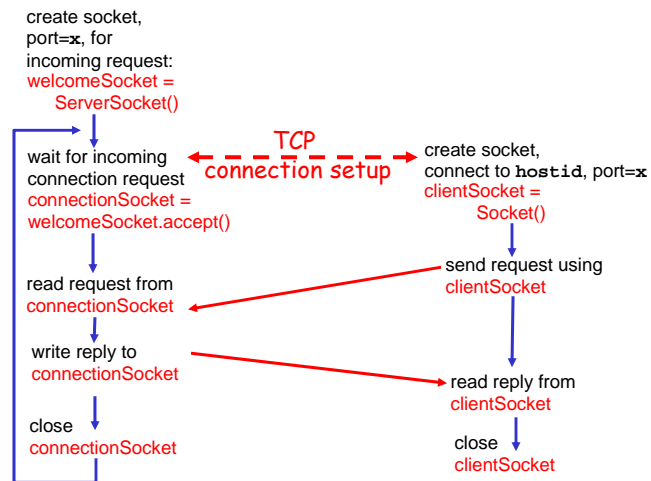


2: Application Layer 62

Client/server socket interaction: TCP

Server (running on `hostid`)

Client



2: Application Layer 63

Example: Java client (TCP)

```

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

```

    public static void main(String argv[]) throws Exception
    {
  
```

```

        String sentence;
        String modifiedSentence;
  
```

```

        Create input stream → BufferedReader inFromUser =
                               new BufferedReader(new InputStreamReader(System.in));
        Create client socket, connect to server → Socket clientSocket = new Socket("hostname", 6789);
        Create output stream attached to socket → DataOutputStream outToServer =
                                                  new DataOutputStream(clientSocket.getOutputStream());
  
```

2: Application Layer 64

Example: Java client (TCP), cont.

```
        Create  
input stream ] → BufferedReader inFromServer =  
attached to socket ] new BufferedReader(new  
                        InputStreamReader(clientSocket.getInputStream()));  
  
        sentence = inFromUser.readLine();  
  
        Send line ] → outToServer.writeBytes(sentence + '\n');  
to server ]  
  
        Read line ] → modifiedSentence = inFromServer.readLine();  
from server ]  
  
        System.out.println("FROM SERVER: " + modifiedSentence);  
  
        clientSocket.close();  
  
    }  
}
```

2: Application Layer 65

Example: Java server (TCP)

```
import java.io.*;  
import java.net.*;  
  
class TCPServer {  
  
    public static void main(String argv[]) throws Exception  
    {  
        String clientSentence;  
        String capitalizedSentence;  
  
        Create ]  
welcoming socket ] → ServerSocket welcomeSocket = new ServerSocket(6789);  
at port 6789 ]  
  
        Wait, on welcoming ]  
socket for contact ] → Socket connectionSocket = welcomeSocket.accept();  
by client ]  
  
        Create input ]  
stream, attached ] → BufferedReader inFromClient =  
to socket ] new BufferedReader(new  
                InputStreamReader(connectionSocket.getInputStream()));  
  
    }  
}
```

2: Application Layer 66

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

application viewpoint

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

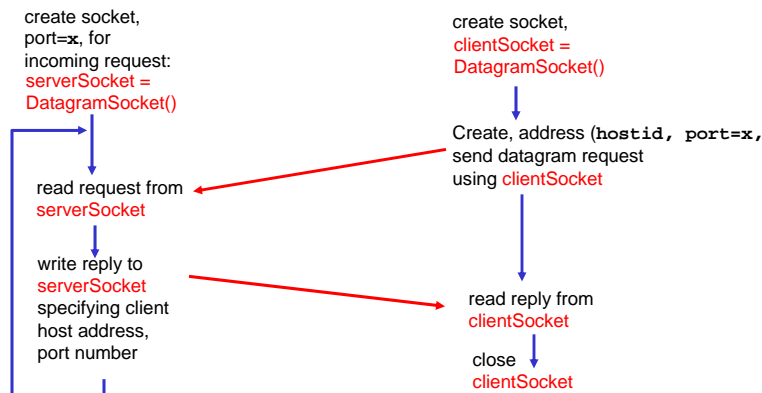
UDP: transmitted data may be received out of order, or lost

2: Application Layer 69

Client/server socket interaction: UDP

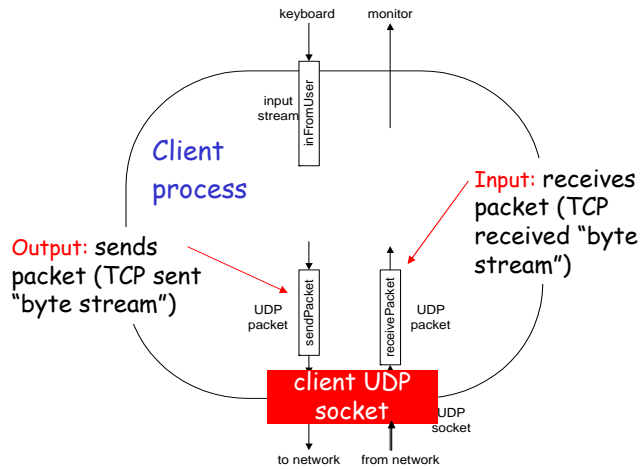
Server (running on `hostid`)

Client



2: Application Layer 70

Example: Java client (UDP)



2: Application Layer 71

Example: Java client (UDP)

```

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

class UDPClient {
    public static void main(String args[]) throws Exception
    {
        Create input stream → BufferedReader inFromUser =
                               new BufferedReader(new InputStreamReader(System.in));
        Create client socket → DatagramSocket clientSocket = new DatagramSocket();
        Translate hostname to IP address using DNS → InetAddress IPAddress = InetAddress.getByName("hostname");

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

        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
    }
}

```

2: Application Layer 72

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();  
                               }  
                               }  
                               }
```

2: Application Layer 73

Example: Java server (UDP)

```
import java.io.*;  
import java.net.*;  
  
class UDPServer {  
    public static void main(String args[]) throws Exception  
    {  
        Create  
        datagram socket  
        at port 9876 ] → DatagramSocket serverSocket = new DatagramSocket(9876);  
  
        byte[] receiveData = new byte[1024];  
        byte[] sendData = new byte[1024];  
  
        while(true)  
        {  
            Create space for  
            received datagram ] → DatagramPacket receivePacket =  
                                   new DatagramPacket(receiveData, receiveData.length);  
            Receive  
            datagram ] → serverSocket.receive(receivePacket);  
        }  
    }  
}
```

2: Application Layer 74

Example: Java server (UDP), cont

```
String sentence = new String(receivePacket.getData());  
Get IP addr  
port #, of  
sender → InetAddress IPAddress = receivePacket.getAddress();  
→ int port = receivePacket.getPort();  
  
String capitalizedSentence = sentence.toUpperCase();  
  
sendData = capitalizedSentence.getBytes();  
Create datagram  
to send to client → DatagramPacket sendPacket =  
new DatagramPacket(sendData, sendData.length, IPAddress,  
port);  
Write out  
datagram  
to socket → serverSocket.send(sendPacket);  
}  
}  
} ← End of while loop,  
loop back and wait for  
another datagram
```

2: Application Layer 75

Chapter 2: Application layer

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

2: Application Layer 76

DNS: Domain Name System

People: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - 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"

2: Application Layer 77

DNS

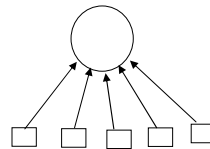
DNS services

- 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?

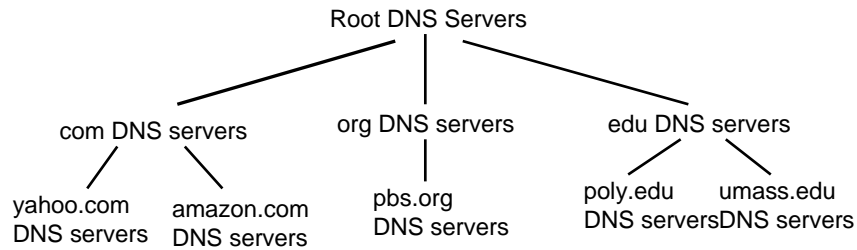
- single point of failure
- traffic volume
- distant centralized database
- maintenance

doesn't scale!



2: Application Layer 78

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

2: Application Layer 79

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

2: Application Layer 80

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

2: Application Layer 81

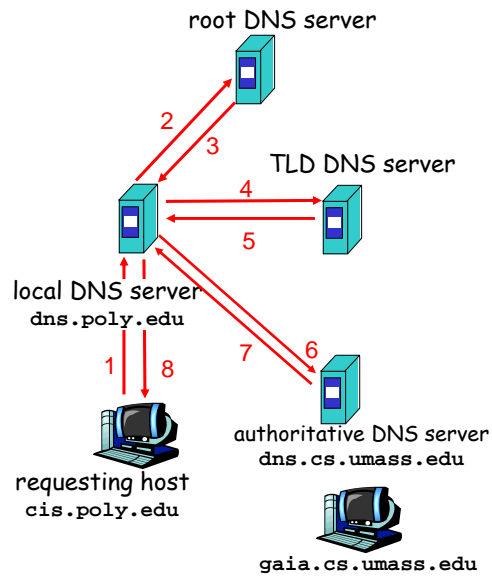
Local Name Server

- ❑ 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.

2: Application Layer 82

Example

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



2: Application Layer 83

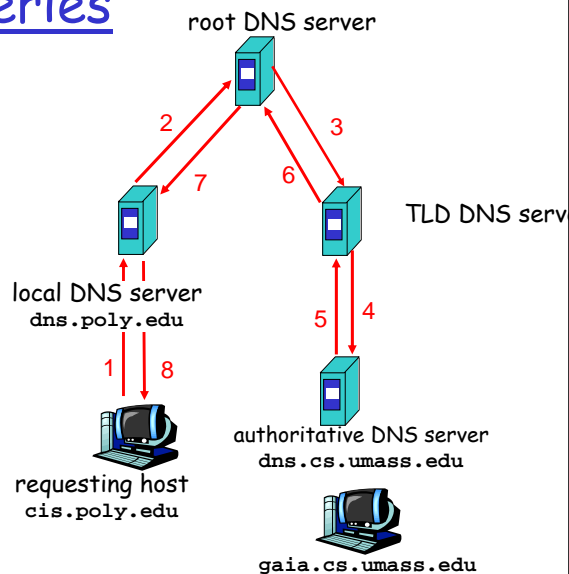
Recursive queries

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"



2: Application Layer 84

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>

2: Application Layer 85

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 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

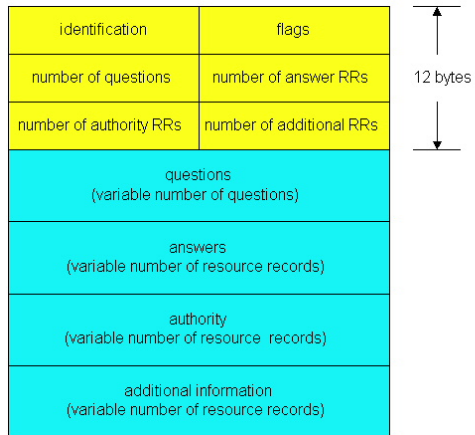
2: Application Layer 86

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



2: Application Layer 87

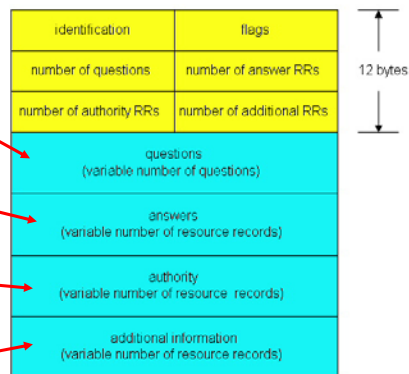
DNS protocol, messages

Name, type fields
for a query

RRs in reponse
to query

records for
authoritative servers

additional "helpful"
info that may be used



2: Application Layer 88

Inserting records into DNS

- Example: just created startup "Utopia"
- Register name networkutopia.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:

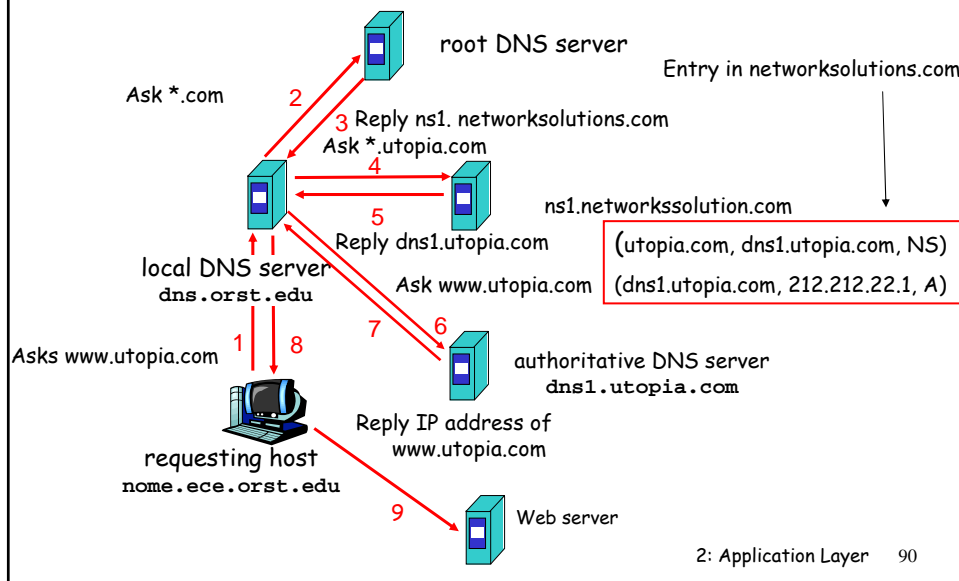
(utopia.com, dns1.utopia.com, NS)

(dns1.utopia.com, 212.212.22.1, A)

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

2: Application Layer 89

How do people get the IP address of your Web site?



Chapter 2: Application layer

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

2: Application Layer 91

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!

2: Application Layer 92

P2P: centralized directory

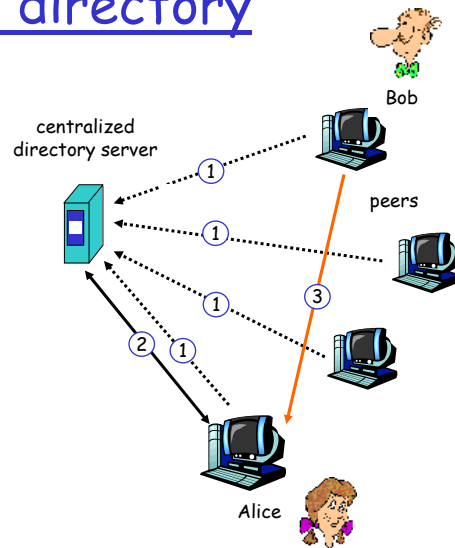
original "Napster" design

1) when peer connects, it informs central server:

- IP address
- content

2) Alice queries for "Hey Jude"

3) Alice requests file from Bob



2: Application Layer 93

P2P: problems with centralized directory

- ❑ Single point of failure
- ❑ Performance bottleneck
- ❑ Copyright infringement
 - (Kazaa located in Pacific island nation of Vanuatu)

file transfer is decentralized, but locating content is highly centralized!

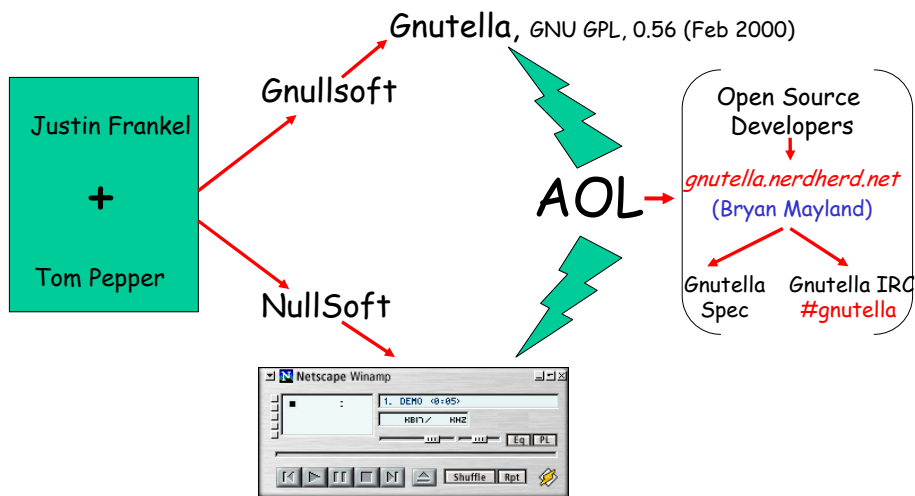
2: Application Layer 94

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

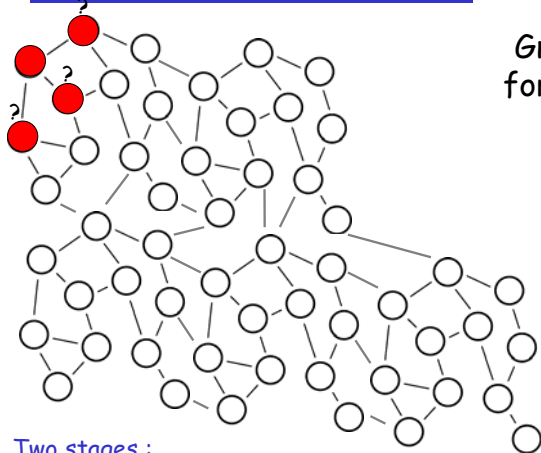
2: Application Layer 95

History of Gnutella



2: Application Layer 96

What is Gnutella?



Gnutella is a protocol for distributed search

- peer-to-peer comms
- decentralized model
- No third party lookup

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

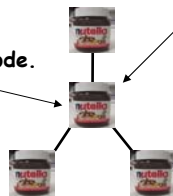
Two stages :

1. Join Network ... later
2. Use Network
 1. Discover other peers
 2. Search other peers

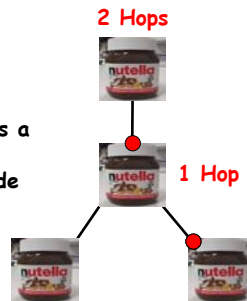
The Jargon

Servent: A Gnutella node.

Each servent is both a client and a server



Hops: a hop is a pass through an intermediate node



Horizon: how many hops a packet can go before it dies (default setting is 7 in Gnutella)



Gnutella Descriptor

- Gnutella messages that are passed around the Gnutella network

5 Descriptor Types

• **Ping**: used to actively discover hosts on the network. A *servent* receiving a *Ping* descriptor is expected to respond with one or more *Pong* descriptors.

• **Pong**: the response to a *Ping*.

(Each Pong packet contains a Globally Unique Identifier (GUID) plus address of *servent* and information regarding the amount of data it is making available to the network)

• **Query**: the primary mechanism for searching the distributed network. A *servent* receiving a *Query* descriptor will respond with a *QueryHit* if a match is found against its local data set.

• **QueryHit**: the response to a *Query*: contains IP address, GUID and search results

• **Push**: allows downloading from *firewalled servents*

2: Application Layer 99

Gnutella Scenario

Step 0: Join the network

Step 1: Determining who is on the network

- "Ping" packet is used to announce your presence on the network.
- Other peers respond with a "Pong" packet.
- Also forwards your Ping to other connected peers
- A Pong packet also contains:
 - an IP address
 - port number
 - amount of data that peers is sharing
- Pong packets come back via same route

Step 2: Searching

- Gnutella is a protocol for distributed search.
- Gnutella "Query" ask other peers if they have the file you desire (and have an acceptably fast network connection).
- A Query packet might ask, "Do you have any content that matches the string 'Homer'?"
- Peers check to see if they have matches & respond (if they have any matches) & send packet to connected peers
- Continues for TTL

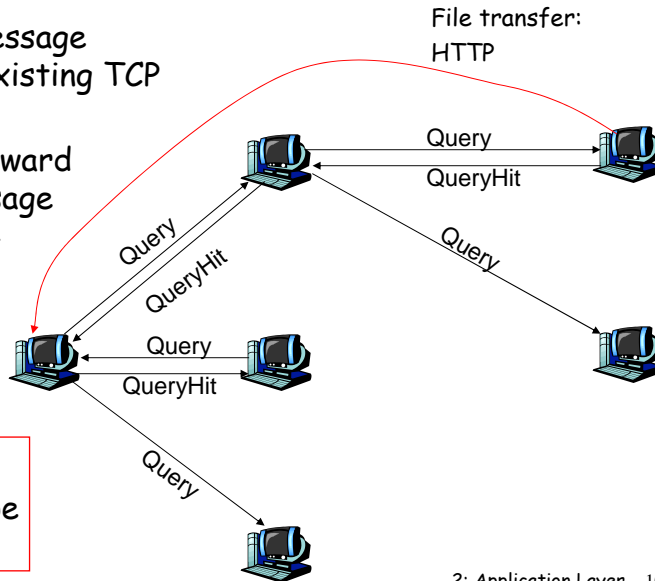
Step 3: Downloading

- Peers respond with a "QueryHit" (contains contact info)
- File transfers use direct connection using HTTP protocol's GET method
- When there is a firewall a "Push" packet is used - reroutes via Push path

2: Application Layer 100

Gnutella: protocol

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



Scalability:
limited scope
flooding

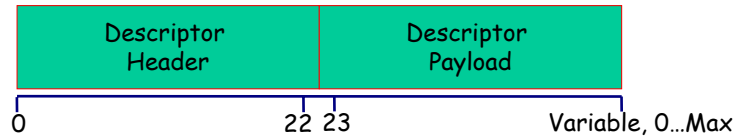
2: Application Layer 101

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

2: Application Layer 102

Gnutella Descriptors

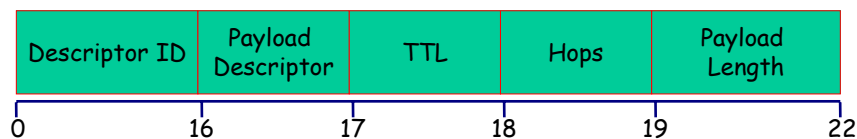


Descriptor Types

- **Ping**: to actively discover hosts on the network.
- **Pong**: the response to a *Ping* (includes the *GUID* address of a connected *servent* and information regarding the amount of data it is making available to the network)
- **Query**: search mechanism
- **QueryHit**: the response to a *Query* (containing *GUID* and file info)
- **Push**: mechanism for *firewalled servents*

2: Application Layer 105

Gnutella Descriptor Header



- **Descriptor ID**: a unique identifier for the descriptor on the network (16-byte string)
- **Payload Descriptor**: *0x00* = *Ping*; *0x01* = *Pong*; *0x40* = *Push*; *0x80* = *Query*; *0x81* = *QueryHit*
- **TTL**: *Time To Live or Horizon*. Each *servent* decrements the TTL before passing it on - when TTL = 0, it is no longer forwarded.
- **Hops**: counts the number of hops the descriptor has traveled i.e. hops = TTL(0) when TTL expires
- **Payload Length**: next descriptor header is located exactly *Payload Length* bytes from end descriptor header

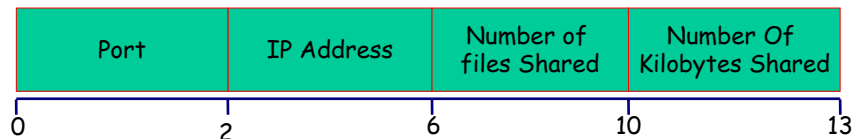
2: Application Layer 106

Gnutella Payload 1 - Ping Descriptor

- **Ping descriptors:**
 - no associated payload
 - = zero length
- A Ping is simply represented by a *Descriptor Header* whose:
 - **Payload_Length** field is 0x00000000.
 - **Payload_Descriptor** field = 0x00

2: Application Layer 107

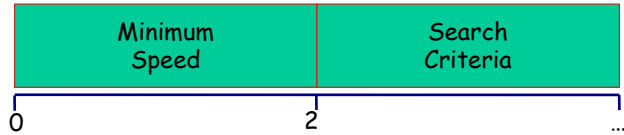
Gnutella Payload 2 - Pong



- **Port:** port which **responding host** can accept *incoming* connections.
- **IP Address:** IP address of the **responding host** (big-endian)
- **Number of Files Shared:** number of files **responding host** is sharing on the network
- **Number of Kilobytes Shared:** kilobytes of data **responding host** is sharing on the network.

2: Application Layer 108

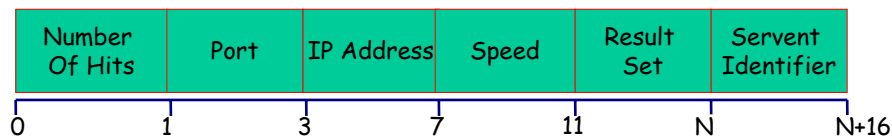
Gnutella Payload 3 - Query



- **Minimum Speed:** minimum speed (in kb/second) of *servents* that should respond to this message.
 - A *Servent* receiving a *Query* descriptor with a minimum speed field of *n* kb/s should only respond with a *QueryHit* if it is able to communicate at a speed $\geq n$ kb/s
- **Search Criteria:** A nul (i.e. 0x00) terminated search string - maximum length is bound by *Payload_Length* field of the descriptor header.
 - e.g. "myFavouriteSong.mp3"

2: Application Layer 109

Gnutella Payload 4 - QueryHit



- **Number of Hits:** number of *query* hits in the *result set*
- **Port:** port which the *responding host* can accept incoming connections
- **IP Address:** IP address of the *responding host* (big-endian)
- **Speed:** speed (in kb/second) of the *responding host*
- **Result Set:** set of *Number_of_Hits* responses to the corresponding Query with the following structure:

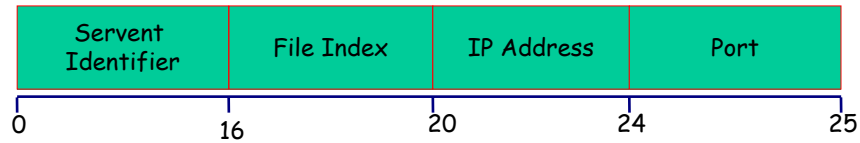


- **File Index:** ID of file matching the corresponding query - assigned by the *responding host*
- **File Size:** size (bytes) of this file
- **File Name:** name of the file (double-nul (i.e. 0x0000) terminated)

- **Servent Identifier:** servent network ID (16-byte string), typically function of servent's network address - instrumental in the operation of the *Push Descriptor* ...

2: Application Layer 110

Gnutella Payload 5 - Push

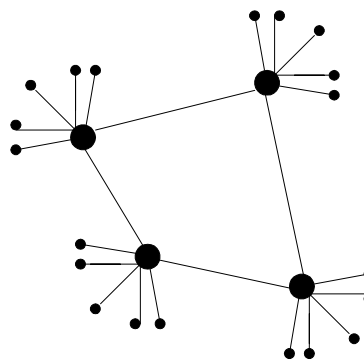


- **Servent Identifier:** *target servent* network ID (16-byte string) requested to **push** file (with given index *File_Index*)
- **File Index:** ID of the file to be pushed from the *target servent*
- **IP Address:** IP address of *target* host which file should be **pushed** (big-endian forma)
- **Port:** port on *target* host which file should be pushed

2: Application Layer 111

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.



● ordinary peer
● group-leader peer
— neighboring relationships in overlay network

2: Application Layer 112

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

2: Application Layer 113

Kazaa tricks

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

2: Application Layer 114

Chapter 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
 - SMTP, POP, IMAP
 - DNS
- ❑ socket programming

2: Application Layer 115

Chapter 2: Summary

Most importantly: learned about *protocols*

- ❑ typical request/reply message exchange:
 - client requests info or service
 - server responds with data, status code
- ❑ message formats:
 - headers: fields giving info about data
 - data: info being communicated
- ❑ control vs. data msgs
 - in-band, out-of-band
- ❑ centralized vs. decentralized
- ❑ stateless vs. stateful
- ❑ reliable vs. unreliable msg transfer
- ❑ "complexity at network edge"

2: Application Layer 116