Chapter 2
Application Layer

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers).
They're in PowerPoint form so you can add, modify, and delete slides
(including this one) and slide content to suit your needs. They obviously
represent a lot of work on our part. In return for use, we only ask the
following:
☑ If you use these slides (e.g., in a class) in substantially unaltered form,
that you mention their source (after all, we’d like people to use our book!)
☑ If you post any slides in substantially unaltered form on a www site, that
you note that they are adapted from (or perhaps identical to) our slides, and
note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2007
J.F Kurose and K.W. Ross, All Rights Reserved

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
- 2.5 DNS
- 2.6 P2P Applications
  Gnutella, Kazaa, BitTorrent
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

Creating a network app

write programs that
- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

little software written for devices in network core
- network core devices do not run user applications
- applications on end systems allows for rapid app development, propagation
Application architectures

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

Client-server architecture

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

clients:
- 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
- centralized service: client presence detection /location
  - user registers its IP address with central server when it comes online
  - user contacts central server to find IP addresses of buddies
Processes communicating

Process: program running within a host.
- within same host, two processes communicate using inter-process communication (defined by OS).
- processes in different hosts communicate by exchanging messages

Client process: process that initiates communication
Server process: process that waits to be contacted

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

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving proc.

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

- to receive messages, process must have identifier
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
  - A: No, many processes can be running on same host

identifier includes both IP address and port numbers associated with process on host.

Example port numbers:
- HTTP server: 80
- Mail server: 25

To send HTTP message to gaia.cs.umass.edu web server:
- IP address: 128.119.245.12
- Port number: 80

more shortly...
App-layer protocol defines

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

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

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Data loss</th>
<th>Bandwidth</th>
<th>Time Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5kbps-1Mbps</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>video:10kbps-5Mbps</td>
<td></td>
</tr>
<tr>
<td>stored audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few kbps up</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>instant messaging</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>

### 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?
Internet apps: application, transport protocols

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

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
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.someSchool.edu/someDepartment/home.index

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

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

3. HTTP server receives request message, forms response message containing requested object, and sends message into its socket
Nonpersistent HTTP (cont.)

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

Non-Persistent HTTP: Response time

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
**Persistent HTTP**

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

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

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

  (extra carriage return, line feed)
HTTP request message: general format

<table>
<thead>
<tr>
<th>method</th>
<th>sp</th>
<th>URL</th>
<th>sp</th>
<th>version</th>
<th>cr</th>
<th>If</th>
</tr>
</thead>
<tbody>
<tr>
<td>header field name</td>
<td>:</td>
<td>value</td>
<td>cr</td>
<td>If</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>header field name</td>
<td>:</td>
<td>value</td>
<td>cr</td>
<td>If</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cr</td>
<td>If</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entity Body

Uploading form input

**Post method:**
- Web page often includes form input
- Input is uploaded to server in entity body

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

www.somesite.com/animalsearch?monkeys&banana
Method types

HTTP/1.0
- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1
- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP response message

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

data data data data data ...

2: Application Layer  29
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 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!
**User-server state: cookies**

Many major Web sites use cookies

Four components:
1) cookie header line of HTTP response message
2) cookie header line in HTTP request message
3) cookie file kept on user’s host, managed by user’s browser
4) back-end database at Web site

**Example:**
- Susan always access Internet always from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

**Cookies: keeping “state” (cont.)**

[Diagram showing the interaction between client and server with examples of cookie usage.]

- Usual HTTP request message
- Usual HTTP response message
- Amazon server creates ID 1678 for user
- cookie-specific action
- backend database

[Diagram showing the flow of data between eBay, Amazon, and the server with specific cookie values.]
Cookies (continued)

**What cookies can bring:**
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

**How to keep “state”:**
- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

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

Web caches (proxy server)

**Goal:** satisfy client request without involving origin server

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

2: Application Layer 35

2: Application Layer 36
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
Caching example (cont)

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

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

Caching example (cont)

possible solution: install cache
- suppose hit rate is 0.4

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

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

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, TCP is transport protocol
- Client authorized over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives file transfer command, server opens 2nd TCP connection (for file) to client
- After transferring one file, server closes data connection.
- Server opens another TCP data connection to transfer another file.
- Control connection: "out of band"
- FTP server maintains "state": current directory, earlier authentication

FTP commands, responses

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

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

Three major components:
- user agents
- mail servers
- simple mail transfer protocol: SMTP

User Agent
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Mozilla Thunderbird
- outgoing, incoming messages stored on server

Electronic Mail: mail servers

Mail Servers
- mailbox contains incoming messages for user
- message queue of outgoing (to be sent) mail messages
- SMTP protocol between mail servers to send email messages
  - client: sending mail server
  - “server”: receiving mail server
Electronic Mail: SMTP [RFC 2821]

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

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
Sample SMTP interaction

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

Try SMTP interaction for yourself:

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

above lets you send email without using email client (reader)
SMTP: final words

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

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

Mail message format

SMTP: protocol for exchanging email msgs
RFC 822: standard for text message format:
- header lines, e.g.,
  - To:
  - From:
  - Subject: different from SMTP commands!
- body
  - the "message". ASCII characters only
Message format: multimedia extensions

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

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
```

- base64 encoded data ....
- .........................
- ........base64 encoded data

Mail access protocols

- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-> server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - HTTP: gmail, Hotmail, Yahoo! Mail, etc.
POP3 protocol

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

transaction phase, client:
- list: list message numbers
- retr: retrieve message by number
- dele: delete
- quit

S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off

POP3 (more) and IMAP

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

IMAP
- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name
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.yahoo.com - used by humans

Q: map between IP addresses and name?

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

DNS

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

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

doesn’t scale!
Distributed, Hierarchical Database

Client wants IP for www.amazon.com:
- 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 cannot 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, mail).
  - can be maintained by organization or service provider

Local Name Server

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

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

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

**Recursive query:**
- puts burden of name resolution on contacted name server
- heavy load?
DNS: caching and updating records

- Once (any) name server learns mapping, it caches mapping
  - Cache entries timeout (disappear) after some time
  - TLD servers typically cached in local name servers
    - Thus root name servers not often visited
- Update/notify mechanisms under design by IETF
  - RFC 2136

DNS records

DNS: distributed db storing resource records (RR)

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

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

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

### msg header
- **ident**ication: 16 bit # for query, reply to query uses same #
- **flags**:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of questions</td>
<td>number of answer RRs</td>
</tr>
<tr>
<td>number of authority RRs</td>
<td>number of additional RRs</td>
</tr>
</tbody>
</table>

```
questions (variable number of questions)
answers (variable number of resource records)
authority (variable number of resource records)
additional information (variable number of resource records)
```

---

## DNS protocol, messages

**Name, type fields** for a query

**RRs in response to query**

**records for authoritative servers**

**additional “helpful” info that may be used**

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of questions</td>
<td>number of answer RRs</td>
</tr>
<tr>
<td>number of authority RRs</td>
<td>number of additional RRs</td>
</tr>
</tbody>
</table>

```
questions (variable number of questions)
answers (variable number of resource records)
authority (variable number of resource records)
additional information (variable number of resource records)
```
Inserting records into DNS

- example: new startup “Network Utopia”
- register name networkuptopia.com at DNS registrar (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts two RRs into com TLD server:
    (networkutopia.com, dns1.networkutopia.com, NS)
    (dns1.networkutopia.com, 212.212.212.1, A)
- create authoritative server Type A record for www.networkuptopia.com; Type MX record for networkuptopia.com
- How do people get IP address of your Web site?

P2P file sharing

Example

- Alice chooses one of the peers, Bob.
- file is copied from Bob’s PC to Alice’s notebook: HTTP
- while Alice downloads, other users uploading from Alice.
- Alice’s peer is both a Web client and a transient Web server.

All peers are servers = highly scalable!
P2P: centralized directory

original "Napster" design

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

2) Alice queries for “Hey Jude”

3) Alice requests file from Bob

P2P: problems with centralized directory

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

file transfer is decentralized, but locating content is highly centralized
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 form overlay net
- edge: virtual (not physical) link
- given peer typically 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

File transfer: HTTP

Query message sent over existing TCP connections
Gnutella: Peer joining

1. Joining peer Alice must find another peer in Gnutella network: use list of candidate peers
2. Alice sequentially attempts TCP connections with candidate peers until connection setup with Bob
3. Flooding: Alice sends Ping message to Bob; Bob forwards Ping message to his overlay neighbors (who then forward to their neighbors...)
   - peers receiving Ping message respond to Alice with Pong message
4. Alice receives many Pong messages, and can then setup additional TCP connections

Hierarchical Overlay

- between centralized index, query flooding approaches
- 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 content in its children
Comparing Client-server, P2P architectures

**Question**: How much time distribute file initially at one server to \(N\) other computers?

![Diagram](image)

- \(u_s\): server upload bandwidth
- \(u_i\): client/peer i upload bandwidth
- \(d_i\): client/peer i download bandwidth

**Client-server: file distribution time**

- 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 increases linearly in \(N\) (for large \(N\))

\[ t_{cs} = \max \left\{ \frac{NF}{u_s}, \frac{F}{\min(d_i)} \right\} \]
**P2P: file distribution time**

- server must send one copy: $F/u_s$ time
- client $i$ takes $F/d_i$ time to download
- NF bits must be downloaded
- fastest possible upload rate (assuming all nodes sending file chunks to same peer): $u_s + \sum_{i=1}^{N} u_i$

$$d_{P2P} = \max \left\{ \frac{F}{u_s}, \frac{F}{\min(d_i)}, \frac{NF}{(u_s + \sum_{i=1}^{N} u_i)} \right\}$$

**Comparing Client-server, P2P architectures**

![Graph comparing P2P and Client-Server architectures](image-url)
P2P Case Study: BitTorrent

- P2P file distribution

  **tracker:** tracks peers participating in torrent  
  **torrent:** group of peers exchanging chunks of a file

  - obtain list of peers
  
  - trading chunks
  
  - peer

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 issues 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

P2P Case study: Skype

- P2P (pc-to-pc, pc-to-phone, phone-to-pc) Voice-Over-IP (VoIP) application
  ❖ also IM
- proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay
Skype: making a call

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

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-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

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server
Client/server socket interaction: TCP

Server (running on hostid)

- create socket, port=x, for incoming request:
  - welcomeSocket = ServerSocket()
- wait for incoming connection request: connectionSocket = welcomeSocket.accept()
- read request from connectionSocket
- write reply to connectionSocket
- close connectionSocket

Client

- create socket, connect to hostid, port=x
- clientSocket = Socket()
- send request using clientSocket
- read reply from clientSocket
- close clientSocket

TCP connection setup

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, e.g., keyboard or socket.
- An **output stream** is attached to an output source, e.g., 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)

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());
        BufferedReader inFromServer =
            new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    }
}
```
Example: Java client (TCP), cont.

```java
BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + \n);
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();
}
}
```

Example: Java server (TCP)

```java
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()));
```

```
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

Server (running on hostid)

1. Create socket, `port=x`, for incoming request:
   ```java
   serverSocket = DatagramSocket();
   ```
2. Read request from `serverSocket`
3. Write reply to `serverSocket` specifying client host address, port number

Client

1. Create socket, `clientSocket = DatagramSocket();`
2. Create, address `(hostid, port=x)`, send datagram request using `clientSocket`
3. Read reply from `clientSocket`
4. Close `clientSocket`

Example: Java client (UDP)

- **Input**: receives packet (recall that TCP received "byte stream")
- **Output**: sends packet (recall that TCP sent "byte stream")

Client process

Client UDP socket

Input: receives packet (recall that TCP received "byte stream")

Output: sends packet (recall that TCP sent "byte stream")
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();
        DatagramPacket sendPacket =
            new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
        clientSocket.send(sendPacket);
        DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence =
            new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}

Example: Java client (UDP), cont.

DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
clientSocket.send(sendPacket);
DatagramPacket receivePacket =
    new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
String modifiedSentence =
    new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}
Example: Java server (UDP)

```
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);

            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);
        }
    }
}
```
Chapter 2: Summary

- application architectures
  - client-server
  - P2P
  - hybrid
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP
  - DNS
  - P2P: BitTorrent, Skype
- socket programming

Most importantly: learned about protocols

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated

Important themes:
- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- “complexity at network edge”