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
Chapter 2 outline

- 2.1 Principles of application layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing

Network applications: some jargon

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

**User agent**: interfaces with user “above” and network “below”.
- implements user interface & application-level protocol
  - Web: browser
  - E-mail: mail reader
  - streaming audio/video: media player
Applications and application-layer protocols

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

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

App-layer protocol defines

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

Public-domain protocols:
- defined in RFCs
- allows for interoperability
- eg, HTTP, SMTP

Proprietary protocols:
- eg, KaZaA
**Client-server paradigm**

Typical network app has two pieces: *client* and *server*

**Client:**
- initiates contact with server ("speaks first")
- typically requests service from server,
- Web: client implemented in browser; e-mail: in mail reader

**Server:**
- provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail

**Processes communicating across network**

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process assumes 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.
- Every host has a unique 32-bit IP address.
- Q: does the IP address of the host on which the process runs suffice for identifying the process?
- Answer: No, many processes can be running on same host.
- Identifier includes both the IP address and port numbers associated with the process on the host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- More on this later.

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 video:10kbps-5Mbps</td>
<td>yes, 100's msec</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 providing: timing, minimum bandwidth guarantees

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

Q: why bother? Why is there a UDP?
### Internet apps: application, transport protocols

<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</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>proprietary (e.g. RealNetworks)</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>proprietary (e.g., Dialpad)</td>
<td>typically UDP</td>
</tr>
</tbody>
</table>

### Chapter 2 outline

- 2.1 Principles of application protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing
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

PC running Explorer
HTTP request
HTTP response

Server running Apache Web server
HTTP request
HTTP response

Mac running Navigator
HTTP overview (continued)

Uses TCP:
- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"
- server maintains no information about past client requests

Aside:
Protocols that maintain "state" are complex!
- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections

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

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

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

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

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

Nonpersistent HTTP (cont.)
Response time modeling

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

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

total = 2RTT + transmit time

**Persistent HTTP**

**Nonpersistent HTTP issues:**
- requires 2 RTTs per object
- OS must work and allocate host resources for each TCP connection
- but browsers often open parallel TCP connections to fetch referenced objects

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

**Persistent without pipelining:**
- client issues new request only when previous response has been received
- one RTT for each referenced object

**Persistent with pipelining:**
- default in HTTP/1.1
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects
HTTP request message

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

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

Carriage return, line feed indicates end of message

HTTP request message: general format

```
method  sp  URL  sp  version  cr  if  
header field name : value  cr  if

header field name : value

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

status line
(protocol
status code
status phrase)

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, e.g., requested HTML file

data data data data data ...

HTTP response status codes

In first line in server->client response message.
A few sample codes:

200 OK
  • request succeeded, requested object later in this message

301 Moved Permanently
  • requested object moved, new location specified later in this message (Location:)

400 Bad Request
  • request message not understood by server

404 Not Found
  • requested document not found on this server

505 HTTP Version Not Supported
Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

   telnet www.eurecom.fr 80

   Opens TCP connection to port 80
   (default HTTP server port) at www.eurecom.fr.
   Anything typed in sent
   to port 80 at www.eurecom.fr

2. Type in a GET HTTP request:

   GET /~ross/index.html HTTP/1.0

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

3. Look at response message sent by HTTP server!

User-server interaction: authorization

Authorization : control access to server content

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

<table>
<thead>
<tr>
<th>client</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>usual http request msg</td>
<td>WWW authenticate:</td>
</tr>
<tr>
<td>401: authorization req.</td>
<td>usual http request msg</td>
</tr>
<tr>
<td>+ Authorization: &lt;cred&gt;</td>
<td>usual http response msg</td>
</tr>
<tr>
<td>usual http response msg</td>
<td>WWW authenticate:</td>
</tr>
<tr>
<td>+ Authorization: &lt;cred&gt;</td>
<td>usual http response msg</td>
</tr>
</tbody>
</table>

2: Application Layer 29
Cookies: keeping “state”

Many major Web sites use cookies

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

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

---

Cookies: keeping “state” (cont.)

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Cookie file" /></td>
<td><img src="image2" alt="Cookie file" /></td>
</tr>
<tr>
<td><img src="image3" alt="Cookie file" /></td>
<td><img src="image4" alt="Cookie file" /></td>
</tr>
</tbody>
</table>

**one week later:**
- ![Cookie file](image5)
- ![Cookie file](image6)

- usual http request msg
- usual http response + Set-cookie: 1678
- server creates ID 1678 for user
- cookie-specific action
- cookie-specific action

2: Application Layer 31

2: Application Layer 32
Cookies (continued)

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

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

Conditional GET: client-side caching

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

client
- HTTP request msg
- If-modified-since: <date>

server
- object not modified
- HTTP response
  HTTP/1.0 304 Not Modified

client
- HTTP request msg
- If-modified-since: <date>

server
- object modified
- HTTP response
  HTTP/1.0 200 OK
  <data>
Chapter 2 outline

- 2.1 Principles of application layer protocols
  - Clients and servers
  - App requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing

FTP: the file transfer protocol

- Transfer file to/from remote host
- Client/server model
  - Client: side that initiates transfer (either to/from remote)
  - Server: remote host
- FTP: RFC 959
- FTP server: port 21
FTP: separate control, data connections

- FTP client contacts FTP server at port 21, specifying TCP as transport protocol
- Client obtains authorization over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives a command for a file transfer, the server opens a TCP data connection to client
- After transferring one file, server closes connection.

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

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

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

User Agent
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- outgoing, incoming messages stored on server
**Electronic Mail: mail servers**

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

**Electronic Mail: SMTP [RFC 2821]**
- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - **commands**: ASCII text
  - **response**: status code and phrase
- messages must be in 7-bit ASCII
Scenario: 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

MIME version
- method used to encode data
- multimedia data type, subtype, parameter declaration
- encoded data

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
**MIME types**

**Content-Type:** type/subtype; parameters

- **Text**
  - example subtypes: `plain`, `html`

- **Image**
  - example subtypes: `jpeg`, `gif`

- **Audio**
  - example subtypes: `basic` (8-bit mu-law encoded), `32kadpcm` (32 kbps coding)

- **Video**
  - example subtypes: `mpeg`, `quicktime`

- **Application**
  - other data that must be processed by reader before "viewable"
  - example subtypes: `msword`, `octet-stream`

---

**Multipart Type**

From: alice@crepes.fr  
To: bob@hamburger.edu  
Subject: Picture of yummy crepe.  
MIME-Version: 1.0  
Content-Type: multipart/mixed; boundary=StartOfNextPart

```
--StartOfNextPart
Dear Bob, Please find a picture of a crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64  
Content-Type: image/jpeg  
base64 encoded data .....  
............base64 encoded data  
--StartOfNextPart
Do you want the recipe?
```

---
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.

---

**POP3 protocol**

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

**transaction phase (client):**
- **list**: list message numbers
- **retr**: retrieve message by number
- **dele**: delete
- **quit**

**Example transaction**:

```
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

Chapter 2 outline
- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing
DNS: Domain Name System

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

Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., gaia.cs.umass.edu - 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 name servers

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

- no server has all name-to-IP address mappings

local name servers:
- each ISP, company has local (default) name server
- host DNS query first goes to local name server

authoritative name server:
- for a host: stores that host’s IP address, name
- can perform name/address translation for that host’s name

doesn’t scale!
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

Simple DNS example

host `surf.eurecom.fr` wants IP address of `gaia.cs.umass.edu`

1. contacts its local DNS server, `dns.eurecom.fr`
2. `dns.eurecom.fr` contacts root name server, if necessary
3. root name server contacts authoritative name server, `dns.umass.edu`, if necessary
**DNS example**

Root name server:
- may not know authoritative name server
- may know *intermediate name server*: who to contact to find authoritative name server

**DNS: iterated queries**

**recursive query:**
- puts burden of name resolution on contacted name server
- heavy load?

**iterated query:**
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"
DNS: caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time
- update/notify mechanisms under design by IETF
  - RFC 2136

DNS records

DNS: distributed db storing resource records (RR)

<table>
<thead>
<tr>
<th>RR format: (name, value, type, ttl)</th>
</tr>
</thead>
</table>

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

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

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

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of questions</td>
<td>number of answers</td>
</tr>
<tr>
<td>number of authority RRs</td>
<td>number of additional RRs</td>
</tr>
<tr>
<td>questions (variable number of questions)</td>
<td>answers (variable number of resource records)</td>
</tr>
<tr>
<td>answers (variable number of resource records)</td>
<td>authority (variable number of resource records)</td>
</tr>
<tr>
<td>authority (variable number of resource records)</td>
<td>additional information (variable number of resource records)</td>
</tr>
</tbody>
</table>

### 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
Chapter 2 outline

- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing

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

- **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)

---

*TCP provides reliable, in-order transfer of bytes (“pipe”) between client and server*
Stream jargon

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, 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)
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

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());
    }
}
```
**Example: Java client (TCP), cont.**

```java
Example: Java client (TCP), cont.

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();
```

2: Application Layer

**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()));
        }
    }
}
```

2: Application Layer
Example: Java server (TCP), cont

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

clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '
';

outToClient.writeBytes(capitalizedSentence);
```

Chapter 2 outline

- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing
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

---

Client/server socket interaction: UDP

Server (running on hostid)

- create socket, port=x, for incoming request:
  - serverSocket = DatagramSocket()
- read request from serverSocket
- write reply to serverSocket specifying client host address, port number

Client

- create socket, clientSocket = DatagramSocket()
- Create, address (hostid, port=x), send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket

---
Example: Java client (UDP)

```java
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();

        ...
Example: Java client (UDP), cont.

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

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

Send datagram to server

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

Read datagram from server

```java
String modifiedSentence = new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
```

Example: Java server (UDP)

```java
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);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram
Example: Java server (UDP), cont

```java
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
```

Get IP addr port #, of sender
Create datagram to send to client
Write out datagram to socket
End of while loop, loop back and wait for another datagram

Building a simple Web server

- handles one HTTP request
- accepts the request
- parses header
- obtains requested file from server's file system
- creates HTTP response message:
  - header lines + file
- sends response to client
- after creating server, you can request file using a browser (eg IE explorer)
- see text for details
Socket programming: references

C-language tutorial (audio/slides):
- “Unix Network Programming” (J. Kurose),

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

Chapter 2 outline

- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing
Web caches (proxy server)

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

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

More about Web caching

- Cache acts as both client and server
- Cache can do up-to-date check using If-modified-since HTTP header
  - Issue: should cache take risk and deliver cached object without checking?
    - Heuristics are used.
- Typically cache is installed by ISP (university, company, residential ISP)

**Why Web caching?**

- Reduce response time for client request.
- Reduce traffic on an institution's access link.
- Internet dense with caches enables "poor" content providers to effectively deliver content
Caching example (1)

Assumptions
- average object size = 100,000 bits
- avg. request rate from institution's browser to origin serves = 15/sec
- delay from institutional router to any origin server and back to router = 2 sec

Consequences
- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + milliseconds

Caching example (2)

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

Consequences
- utilization on LAN = 15%
- utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + msecs + msecs
- often a costly upgrade
**Caching example (3)**

- **Install cache**
  - suppose hit rate is .4

- **Consequence**
  - 40% requests will be satisfied almost immediately
  - 60% requests satisfied by origin server
  - utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)
  - total delay = Internet delay + access delay + LAN delay
    \[= .6 \times 2 \text{ sec} + .6 \times .01 \text{ secs} + \text{milliseconds} + 1.3 \text{ secs}\]

---

**Content distribution networks (CDNs)**

- The content providers are the CDN customers.

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

1. Origin server
2. CDNs authoritative DNS server
3. Nearby CDN server

HTTP request for www.foo.com/sports/sports.html

DNS query for www.cdn.com

HTTP request for www.cdn.com/www.foo.com/sports/ruth.gif

**Origin server**
- www.foo.com
- Distributes HTML
- Replaces:
  - http://www.foo.com/sports/ruth.gif
  - with

**CDN company**
- cdn.com
- Distributes gif files
- Uses its authoritative DNS server to route redirect requests

**More about CDNs**

**Routing requests**
- CDN creates a "map", indicating distances from leaf ISPs and CDN nodes
- When query arrives at authoritative DNS server:
  - Server determines ISP from which query originates
  - Uses "map" to determine best CDN server

**Not just Web pages**
- Streaming stored audio/video
- Streaming real-time audio/video
  - CDN nodes create application-layer overlay network
P2P file sharing

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

All peers are servers = highly scalable!

P2P: centralized directory

original "Napster" design
1) when peer connects, it informs central server:
   - IP address
   - content
2) Alice queries for "Hey Jude"
3) Alice requests file from Bob.
**P2P: problems with centralized directory**

- Single point of failure
- Performance bottleneck
- Copyright infringement

File transfer is decentralized, but locating content is highly decentralized.

---

**P2P: decentralized directory**

- Each peer is either a group leader or assigned to a group leader.
- Group leader tracks the content in all its children.
- Peer queries group leader; group leader may query other group leaders.
More about decentralized directory

**Overlay network**
- peers are nodes
- edges between peers and their group leaders
- edges between some pairs of group leaders
- virtual neighbors
- bootstrap node
  - connecting peer is either assigned to a group leader or designated as leader

**Advantages of approach**
- no centralized directory server
  - location service distributed over peers
  - more difficult to shut down

**Disadvantages of approach**
- bootstrap node needed
- group leaders can get overloaded

---

**P2P: Query flooding**

- Gnutella
- no hierarchy
- use bootstrap node to learn about others
- join message

- Send query to neighbors
- Neighbors forward query
- If queried peer has object, it sends message back to querying peer
P2P: more on query flooding

Pros
= peers have similar responsibilities: no group leaders
= highly decentralized
= no peer maintains directory info

Cons
= excessive query traffic
= query radius: may not have content when present
= bootstrap node
= maintenance of overlay network

Chapter 2: Summary

Our study of network apps now complete!

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

specific protocols:
= HTTP
= FTP
= SMTP, POP, IMAP
= DNS
socket programming
content distribution
= caches, CDNs
= P2P
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"
- security: authentication