Chapter 2: Application Layer

Chapter goals:
- conceptual + implementation aspects of network application protocols
  - client server paradigm
  - service models
- learn about protocols by examining popular application-level protocols

More chapter goals
- specific protocols:
  - http
  - ftp
  - smtp
  - pop
  - dns
- programming network applications
  - socket programming
Chapter 2 outline

- 2.1 Principles of application layer protocols
- 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
  - Content distribution networks vs. Web Caching
Applications and application-layer protocols

Applications: communicating, distributed processes

- running the “user space” of network hosts
- which exchange messages among themselves

Network Applications are applications which involves interactions of processes implemented in multiple hosts connected by a network. Examples: the web, email, file transfer

- Within the same host, processes communicate with interprocess communication defined by the OS (Operating System).
- Processes running in different hosts communicate with an application-layer protocol

Application-layer protocols

- are a “piece” of Application (apps)
- define messages exchanged by apps and actions taken
- use services provided by lower layer protocols
Client-server paradigm

Typical network app has two pieces: client and server

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

Server:
- provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail
Application-layer protocols (cont).

API: application programming interface
- defines interface between application and transport layer
- socket: Internet API
  - two processes communicate by sending data into socket, reading data out of socket

Q: how does a process “identify” the other process with which it wants to communicate?
- IP address of host running other process
- “port number” - allows receiving host to determine to which local process the message should be delivered

... lots 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

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

Timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”
## 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: 5Kb-1Mb</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>video: 10Kb-5Mb</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>financial apps</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>
Services provided by Internet transport protocols

TCP service:
- connection-oriented: setup required between client, server
- 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, reliable transport, flow control, congestion control, timing, or bandwidth guarantee
## Internet apps: their protocols and 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 821]</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 2068]</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>remote file server</td>
<td>NFS</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>proprietary (e.g., Vocaltec)</td>
<td>typically UDP</td>
</tr>
</tbody>
</table>
Chapter 2 outline

- 2.1 Principles of application layer protocols
- 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
  - Content distribution networks vs. Web Caching
The Web: some jargon

- **Web page:**
  - consists of “objects”
  - addressed by a URL

- **Most Web pages consist of:**
  - base HTML page, and
  - several referenced objects.

- **URL has two components:**
  - host name and path
  - name:

  www.someSchool.edu/someDept/pic.gif

- **User agent for Web is called a browser:**
  - MS Internet Explorer
  - Netscape Communicator

- **Server for Web is called Web server:**
  - Apache (public domain)
  - MS Internet Information Server
The Web: the http protocol

**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
- http1.0: RFC 1945
- http1.1: RFC 2068
The http protocol: more

http: TCP transport service:
- 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
Suppose user enters URL
www.someSchool.edu/someDepartment/home.index

1a. http client initiates TCP connection to http server (process) at www.someSchool.edu. Port 80 is default for http server.

2. http client sends http request message (containing URL) into TCP connection socket

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

3. http server receives request message, forms response message containing requested object (someDepartment/home.index), sends message into socket
http example (cont.)


6. Steps 1-5 repeated for each of 10 jpeg objects
Non-persistent and persistent connections

Non-persistent
- HTTP/1.0
- server parses request, responds, and closes TCP connection
- At least 2 RTTs (Round Trip Time) to fetch each object
- Repeated 10 times for 10 objects. Each object transfer suffers from slow start

But most 1.0 browsers use parallel TCP connections.

Persistent
- default for HTTP/1.1
- on same TCP connection: server, parses request, responds, parses new request...
- Client sends requests for all referenced objects as soon as it receives base HTML.
- Fewer RTTs and less slow start.
http message format: request

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

  request line
  (GET, POST, HEAD commands)

  header lines

  GET /somedir/page.html HTTP/1.0
  User-agent: Mozilla/4.0
  Accept: text/html, image/gif, image/jpeg
  Accept-language: fr

  (extra carriage return, line feed)

  Carriage return, line feed indicates end of message
http request message: general format

```
method  sp  URL  sp  version  cr  lf
```

```
header field name : value  cr  lf
```

```
header field name : value  cr  lf
```

```
request line
```

```
header lines
```

```
Entity Body
```

http request message: more info

- http/1.0 has only three request methods
  - **GET:**
  - **POST:** for forms. Uses *Entity Body* to transfer form info
  - **HEAD:** Like GET but response does not actually return any info. This is used for debugging/test purposes

- http/1.1 has two additional request methods
  - **PUT:** Allows uploading object to web server
  - **DELETE:** Allows deleting object from web server
http message format: response

status line
(protocol
status code
status phrase)

HTTP/1.0 200 OK
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 data ...

data, e.g., requested html file

header lines

status code
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:

   \texttt{telnet cis.poly.edu 80} \quad \text{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:

   \texttt{GET /~ross/index.html HTTP/1.0} \quad \text{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!

   \textbf{Try telnet www.cs.ust.hk 80}
User-server interaction: authentication

**Authentication goal:** control access to server documents

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

Browser caches name & password so that user does not have to repeatedly enter it.
User-server interaction: cookies

- Server sends “cookie” to client in response msg
  - Set-cookie: 1678453
- Client stores & presents cookie in later requests
  - Cookie: 1678453
- Server matches presented-cookie with server-stored info
  - Authentication
  - Remembering user preferences, previous choices

Diagram:

- Client
  - Usual HTTP request msg
  - Cookie: #
  - Usual HTTP response msg

- Server
  - Usual HTTP request msg
  - Set-cookie: #
  - Usual HTTP response msg
  - Cookie-specific action
  - Cookie-specific action
Cookie example

telnet www.google.com 80

Trying 216.239.33.99...
Connected to www.google.com.
Escape character is '^[].

GET /index.html HTTP/1.0

HTTP/1.0 200 OK
Date: Wed, 10 Sep 2003 08:58:55 GMT
Set-Cookie:
PREF=ID=43bd8b0f34818b58:TM=1063184203:LM=1063184203:
S=DDqPgTb56Za88O2y; expires=Sun, 17-Jan-2038 19:14:07 GMT;
path=/; domain=.google.com

.
User-server interaction: conditional GET

- **Goal:** don't send object if client has up-to-date stored (cached) version
- **Client:** specify date of cached copy in http request
  
  ```
  If-modified-since: <date>
  ```

- **Server:** response contains no object if cached copy up-to-date:
  
  ```
  HTTP/1.0 304 Not Modified
  ```

![Diagram showing the interaction between client and server](image-url)
**Web Caches (proxy server)**

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

- user sets browser: Web accesses via web cache
- client sends all HTTP requests to web cache
  - if object at web cache, web cache immediately returns object in HTTP response
  - else requests object from origin server, then returns HTTP response 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*2 sec + .6*.01 secs + milliseconds < 1.3 secs
Chapter 2: Application Layer

Chapter 2 outline

- 2.1 Principles of application layer protocols
- 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
  - Content distribution networks vs. Web Caching
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
- two parallel TCP connections opened:
  - control: exchange commands, responses between client, server.
    - “out of band control”
  - data: file data to/from server
- ftp server maintains “state”: current directory, earlier authentication
Ftp: separate control, data connections

- When server receives request for file transfer it opens a TCP data connection to client on port 20.
- After transferring one file, server closes connection.
- When next request for file transfer arrives server opens new TCP data connection on port 20.
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
Chapter 2 outline

- 2.1 Principles of app layer protocols
- 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
  - Content distribution networks vs. Web Caching
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 (yet to be read) 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 821]

- uses tcp to reliably transfer email msg 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 that message (header & body) be in 7-bit ascii
- certain character strings are not permitted in message (e.g., CRLF.CRLF). Thus message has to be encoded (usually into either base-64 or quoted printable)
- smtp server uses CRLF.CRLF to determine end of message

Comparison with http

- http: pull
- email: push

- both have ASCII command/response interaction, status codes
- http: each object is encapsulated in its own response message
- smtp: multiple objects message sent in a multipart message
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**: (Multipurpose Internet Mail Extensions)
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
```

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

Content-Type: type/subtype; parameters

Text
▷ example subtypes: plain, html

Image
▷ example subtypes: jpeg, gif

Audio
▷ exampe 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: application/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
  - -ERR

transaction phase, client:

- list: list message numbers
- retr: retrieve message by number
- dele: delete
- quit

S: +OK POP3 server ready
C: user alice
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 application layer protocols
- 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
  - Content distribution networks vs. Web Caching
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 doesn’t scale!

- 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
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
- ~ dozen root name servers worldwide
2. DNS

- Defined in RFCs 1034 and 1035.
- Hierarchical, domain-based naming scheme, and uses distributed database system.

Illustration from Tanenbaum
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)

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

- **Type=A**
  - name is hostname
  - value is IP address

- **Type=NS**
  - name is domain (e.g. foo.com)
  - value is IP address of authoritative name server for this domain

- **Type=CNAME**
  - name is an alias name for some “cannonical” (the real) name
  - value is cannonical name

- **Type=MX**
  - value is hostname of mailserver associated with name
2. Resource Record

From Tanenbaum

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>Start of Authority</td>
<td>Parameters for this zone</td>
</tr>
<tr>
<td>A</td>
<td>IP address of a host</td>
<td>32-Bit integer</td>
</tr>
<tr>
<td>MX</td>
<td>Mail exchange</td>
<td>Priority, domain willing to accept email</td>
</tr>
<tr>
<td>NS</td>
<td>Name Server</td>
<td>Name of a server for this domain</td>
</tr>
<tr>
<td>CNAME</td>
<td>Canonical name</td>
<td>Domain name</td>
</tr>
<tr>
<td>PTR</td>
<td>Pointer</td>
<td>Alias for an IP address</td>
</tr>
<tr>
<td>HINFO</td>
<td>Host description</td>
<td>CPU and OS in ASCII</td>
</tr>
<tr>
<td>TXT</td>
<td>Text</td>
<td>Uninterpreted ASCII text</td>
</tr>
</tbody>
</table>

; Authoritative data for cs.vu.nl

<table>
<thead>
<tr>
<th>Host</th>
<th>Type</th>
<th>Class</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs.vu.nl</td>
<td>SOA</td>
<td>IN</td>
<td>star boss (952771,7200,7200,2419200,66400)</td>
</tr>
<tr>
<td>flits.cs.vu.nl</td>
<td>A</td>
<td>IN</td>
<td>130.37.16.112</td>
</tr>
<tr>
<td>flits.cs.vu.nl</td>
<td>MX</td>
<td>IN</td>
<td>1 flits.cs.vu.nl</td>
</tr>
<tr>
<td>ftp.cs.vu.nl</td>
<td>CNAME</td>
<td>IN</td>
<td>zephyr.cs.vu.nl</td>
</tr>
<tr>
<td>rowboat</td>
<td>A</td>
<td>IN</td>
<td>130.37.56.201</td>
</tr>
<tr>
<td></td>
<td>MX</td>
<td>IN</td>
<td>1 rowboat</td>
</tr>
<tr>
<td></td>
<td>MX</td>
<td>IN</td>
<td>2 zephyr</td>
</tr>
<tr>
<td></td>
<td>HINFO</td>
<td>IN</td>
<td>Sun Unix</td>
</tr>
<tr>
<td>little-sister</td>
<td>A</td>
<td>IN</td>
<td>130.37.62.23</td>
</tr>
<tr>
<td></td>
<td>HINFO</td>
<td>IN</td>
<td>Mac MacOS</td>
</tr>
<tr>
<td>laserjet</td>
<td>A</td>
<td>IN</td>
<td>192.31.231.218</td>
</tr>
<tr>
<td></td>
<td>HINFO</td>
<td>IN</td>
<td>&quot;HP Laserjet IIIi&quot; Proprietary</td>
</tr>
</tbody>
</table>
**DNS protocol, messages**

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

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

- 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
- 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
  - Content distribution networks vs. Web Caching
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

A host-local, application-created, OS-controlled interface (a “door”) into which application process can both send and receive messages to/from another application process.
Socket-programming using TCP

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

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

```
controlled by operating system
process
socket
TCP with buffers, variables
host or server

internet

controlled by application developer
process
socket
TCP with buffers, variables
host or server
```

controlled by operating system

controlled by operating system

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, eg, keyboard or socket.
- An output stream is attached to an output source, eg, monitor or socket.
Socket programming with TCP

Example client-server app:
1) client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket (inFromServer stream)
Client/server socket interaction: TCP

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

TCP connection setup

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());

        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
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();
```

Create input stream attached to socket

Send line to server

Read line from server
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()));

            String clientSentence = inFromClient.readLine();
            String capitalizedSentence = clientSentence.toUpperCase();
        }
    }
}
Example: Java server (TCP), cont

Create output stream, attached to socket

Read in line from socket

Write out line to socket

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

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

outToClient.writeBytes(capitalizedSentence);

End of while loop, loop back and wait for another client connection
Chapter 2 outline

- 2.1 Principles of application layer protocols
- 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
  - Content distribution networks vs. Web Caching
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
TCP vs. UDP

**TCP**

1. **Socket()**
   - Connection steam established: Data goes in one end of pipe and out the other. Pipe stays open until it is closed.

2. **ServerSocket()**
   - A special type of socket that sits waiting for a knock from a client to open connection. Leads to **handshaking**.

**UDP**

1. **DatagramSocket()**
   - Data sent as individual packets of bytes. Each packet contains all addressing info. No concept of open “pipe”.

2. **No handshaking!**
   - A DatagramSocket waits to receive each packet.
Example: Java client (UDP)

Client process

Output: sends packet (TCP sent "byte stream")

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

client UDP socket

Comp361 Spring 2004
Example: Java client (UDP)

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

class UDPCClient {
    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);
```

Send datagram to server

```java
clientSocket.send(sendPacket);
```

Read datagram from server

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

clientSocket.receive(receivePacket);

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

System.out.println("FROM SERVER:" + modifiedSentence);
}
```
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);
        }
    }
}
```
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
Chapter 2 outline

- 2.1 Principles of app layer protocols
- 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
  - Content distribution networks vs. Web Caching
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 (e.g. IE explorer)
- see text for details
Chapter 2 outline

- 2.1 Principles of application layer protocols
- 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
  - Content distribution networks vs. Web Caching
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
   - HTTP request for www.foo.com/sports/sports.html
2. DNS query for www.cdn.com

**origin server**
- www.foo.com
- distributes HTML
- Replaces:
  - http://www.foo.com/sports.ruth.gif

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

- Nearby CDN server

---

Chapter 2: Application Layer

Comp361 Spring 2004
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
Web Caching vs. CDN

Both Web Caching and CDN replicate content

- **Web Caching**: Content replicated on demand as function of user requests

- **CDN**: Content replicated by content provider
P2P

As well as retrieving objects from content providers/proxy caches/CDNs it is also possible for edge-machines to retrieve content from other edge-machines. This approach is known as Peer-To-Peer (P2P).

For more on P2P see textbook.
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
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-based, out-of-band

- centralized vs. decentralized

- stateless vs. stateful

- reliable vs. unreliable msg transfer

- “complexity at network edge”

- security: authentication