## Application layer

Reading: RFC 3117  
Kurose-Ross chapter 2

## Goals

- Be able to program a network application using the socket API.  
- Understand design issues for application protocols.  
- Understand design decisions in particular application protocols.  
  - Understand how those protocols operate.

## Revision

- Revise at home:  
  - COMP247 week 2: application layer; Fitzgerald and Dennis chapter 2.
Socket programming (KR 2.6)

- Socket API
  - Introduced in BSD4.1 UNIX, 1981
  - Explicitly created, used, released by apps
  - Client/server paradigm
    - Can also be used for peer-peer protocols
  - Two types of transport service via socket API:
    - Unreliable datagram
    - Reliable, byte stream-oriented

A socket is...

- An end-point for communication;
- An interface (service access point)
  - Local to a particular host
  - Owned and created by an application
  - Controlled by the Operating System
  - Into which an application process can
    - Send and receive messages
    - To/from another application process
    - That may be remote or local

Socket-programming using TCP

Socket: an interface between application process
and end-to-end transport protocol (UDP 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 (interface) that welcomes client’s contact
- Client contacts server:
  - create client-local TCP socket
  - specify IP address, port number of server process
  - Client TCP establishes contact to server TCP

Socket programming with TCP

- When server is contacted by client
  - server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients

APPLICATION VIEWPOINT

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

Socket, Port and IP address

Example client-server app:
- client reads line from standard input (Stdin), sends to server via socket (ServOut stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (ServIn stream; Stdout)
### Socket programming with TCP

1. **Client process**
   - Keyboard
   - Screen
   - Copy

2. **Server process**
   - Keyboard
   - Screen
   - Copy

#### Case conversion

### Socket interaction: TCP

#### Server (running on `hostid`)
1. Create socket, `port=x`, for incoming request: `socket(); bind()`
2. Wait for incoming connection request: `ConnSock = accept()`
3. Read request from `ConnSock`
4. Write reply to `ConnSock`
5. Close `ConnSock`

#### Client
1. Create socket, connect to `hostid`, `port=x`: `ClientSock = socket(); connect()`
2. Write request to `ClientSock`
3. Read reply from `ClientSock`
4. Close `ClientSock`

### Socket programming with UDP

- **UDP**: no “connection” between client & server
  - No handshaking
  - Sender explicitly attaches IP address and port of destination
  - Server must extract IP address, port of sender from received datagram
  - **UDP**: transmitted data may be received out of order, or lost

#### APPLICATION VIEWPOINT

**UDP provides unreliable transfer of blocks of bytes (“datagrams”) between client and server**

### Socket interaction: UDP

#### Server (running on `hostid`)
1. Create datagram socket, `port=x`, for incoming request: `ServSock = socket(); bind()`
2. Read request from `ServSock`
3. Write reply to `ServSock`, specifying host and port number
4. Close `ServSock`

#### Client
1. Create datagram socket: `ClientSock = socket()`
2. Create, address (`hostid`, `port=x`) datagram request, send using `ClientSock`
3. Read reply from `ClientSock`
4. Close `ClientSock`
Socket programming: more

- Unix man pages
- Assignment 1
- http://www.uwo.ca/its/doc/courses/notes/socket/
  - C-language tutorial, includes inetd 1997
  - Very distinct style of tutorial 2005
- http:// manic.cs.umass.edu/~amldemo/courseware/intro.html
  - Does not seem to work

App Protocol Design Issues

- Dialog control – whose turn to “talk” (session layer issue); asynchrony; parallelism
- Data representation – network standard encoding (presentation-layer issue)
- Security – authentication, privacy
- Transport-layer – connection/connectionless
- Framing of messages
- Error/status reporting
- Syntax and semantics of message
- State maintenance – client, server, both

Reference: RFC 3117

Application Protocols

Design and operation

Dialog control: whose “turn”

- Take turns to send a single message
  - Application-level “stop and wait”
  - Client-server
  - RPC
- Pipelined command/response
- Segmented messages
- Multiple messages depending on protocol state
- Asynchronous; parallel
Data representation

- ASCII Text
  - telnet end-of-line: CR-LF
- Byte ordering for binary data
  - Big-endian / little-endian
- ASN.1
- MIME

Security

- Authentication
- Privacy
- SSL / TLS

Transport service required?

- Data loss
  - Does app require 100% reliable data transfer?
- Timing
  - Does app require low delays?
- Bandwidth
  - Does app require a minimum amount of bandwidth to be “effective”?}

Transport service requirements

<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>loss-tolerant</td>
<td>audio: 5Kb-1Mb</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>video:10Kb-5Mb</td>
<td>yes and no</td>
</tr>
<tr>
<td>streamed 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>
Internet 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

Internet UDP service

- unreliable datagram 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?

Transport protocols used

<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>H.323 / SIP RTP etc</td>
<td>TCP and UDP</td>
</tr>
</tbody>
</table>

Framing

- Stuffing
  - Delimiter to mark end of frame
  - Stuff literal delimiters in data
  - Compare Data Link Layer – review COMP247 week 4

- Counting
  - Frame header contains length of frame/data
  - More efficient
  - Must know data (block) in advance
  - Segmented frames may have individual lengths and continuation flag

- Connection “blasting”
  - Separate connection for each message
Error / Status Reporting

- Error response messages
- ASCII reply messages
  - +OK -ERR (POP3)
    - No information about the precise nature of the error.
  - 3-digit reply codes
    - Different ways of interpreting

Telnet: remote login protocol

- RFC 854
  - CR LF is new line
  - IAC (255) escapes commands (stuffed for literal byte)
    - Similar to "\n \t \" in C

Telnet Design

- Dialog: Asynchronous
- Representation: raw bytes; IAC byte-stuffed; CRLF
- Security: Nil
- Transport-layer: TCP
- Framing: Byte-by-byte
- Error reporting: Minimal
- Syntax: IAC-escaped commands
- State: Server: Logged in “shell”

The Web: http protocol (KR 2.2)

- 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 to server, port 80
  - server accepts TCP connection from client
  - http messages exchanged between browser (client) and Web server (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 example

- Suppose user enters URL
  http://www.someSchool.edu/someDepartment/index.html

  - 2. http client sends http request message (containing URL) into TCP connection socket
  - 3. http server receives request message, forms response message containing requested object (someDepartment/index.html), sends message into socket

http example (cont.)

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


Persistence of connections

- **Non-persistent**
  - http/1.0: server parses request, responds, closes TCP connection
  - 2 RTTs to fetch object
  - TCP connection
  - object request/transfer transfers suffer from TCP’s initially slow sending rate
  - many browsers open multiple parallel 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, less slow start
http message: request

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

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

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

http request message: format

<table>
<thead>
<tr>
<th>Request line</th>
<th>Header lines</th>
<th>Entity body</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>sp</td>
<td>URL</td>
</tr>
<tr>
<td>header field name</td>
<td>value</td>
<td>cr</td>
</tr>
<tr>
<td>header field name</td>
<td>value</td>
<td>cr</td>
</tr>
<tr>
<td>cr</td>
<td>If</td>
<td></td>
</tr>
</tbody>
</table>

HTTP Reply Codes

- 1xx: Informational
- 2xx: Successful. 200 OK
- 3xx: Redirection. 301 Moved permanently 307 Temporary redirect
- 4xx: Client error. 401 Unauthorized 404 Not found
- 5xx: Server error. 503 Service unavailable
Pretending to be http client

1. Telnet to your favorite Web server:
   - telnet www.ics.mq.edu.au 80
     - Opens TCP connection to port 80 (default http server port) at www.ics.mq.edu.au. Anything typed in is sent there
2. Type in a GET http request:
   - GET /~len/index.html HTTP/1.0
     - Hit carriage return twice, to send this GET request to http server
3. Look at response message sent by http server!

User-server interaction: authentication

- Authentication: control access to server content
- Authorization credentials: typically name, password
- Stateless: client must present authorization in each request
- Authorization: header line in each request
  - If no authorization: header, server refuses access, sends WWW authenticate: header line in response

Cookies: keeping "state"

- Server-generated #, client-remembered #, later used for:
  - Authentication
  - Remembering user preferences / choices
- Server sends "cookie" to client in response msg
  - Set-cookie: 1678453
- Client presents cookie in later requests
  - Cookie: 1678453

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
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 in web cache: web cache returns object
  - Else web cache requests object from origin server, then returns object to client

Why Web Caching?

- Assume: cache is “close” to client (e.g., in same network)
- Smaller response time: cache “closer” to client
- Decrease traffic to distant servers
  - Link out of institutional/local ISP network often bottleneck

HTTP Design

- Dialog: Command-reply; pipelined commands (v1.1)
- Representation: MIME objects
- Security: HTTPS provided by SSL
- Transport-layer: TCP
- Framing: HTTP/1.0: connection; HTTP/1.1: length header in MIME object
- Error reporting: 3-digit error codes
- Syntax: ASCII commands and parameters; CRLF; MIME objects (headers and data)
- State: Client maintains state; stateless server (cookies)

Electronic Mail

- Three major components:
  - User agents
  - Mail servers
  - Simple mail transfer protocol: smtp
Electronic Mail: SMTP [RFC 821]

- 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

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 ...
C Do you like ketchup?
C How about pickles?
C .
S 250 Message accepted for delivery
C QUIT
S 221 hamburger.edu closing connection

Theory of reply codes

- RFC821 Appendix E
- 3 digits: csn
- C: Class of response: success/fail
  temporary/permanent
- S: Section of protocol: syntax,
  communication channel, mail system
- D: Detail: specific condition

Reply code examples (SMTP)

- 2yz: S: 220 server.com ready
- 3yz: C: DATA
  S: 354 Enter the message...
  C: Blah blah blah...
- 4yz: C: MAIL FROM:<stranger@un.com>
  S: 421 Service not available – too busy
- 5yz: C: RCPT TO:<missing@site.com>
  S: 550 No such user
SMTP Design

- Dialog: Take turns
- Representation: ASCII text, CRLF
- Security: Minimal
- Transport-layer: TCP
- Framing: CRLF; CRLF “.” CRLF
- Error reporting: theory of error codes; human-readable text message
- Syntax: four-letter commands; ASCII text parameters; CRLF
- State: Both: short-term state (e.g. recipient list); long-term (e-mail queues)

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

SMTP: final words

- Persistent connections
- Message (header & body) must be in 7-bit ASCII
- Certain character strings not permitted in msg (e.g., CRLF,CRLF). Thus msg has to be encoded (e.g. base-64)
- Server uses CRLF.CRLF to determine end of message

Message format: multimedia

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

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

base64 encoded data ....
.........................
......base64 encoded data
**MIME types**

- **Text**
  - example subtypes: plain, html
- **Image**
  - example subtypes: jpeg, gif
- **Audio**
  - example subtypes: basic (8-bit 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

**MIME**

- Is not a protocol but is used in SMTP and other protocols to address certain issues:
- Data typing: MIME types
- Representation: ASCII text or binary data
- Security: nil
- Framing: external to MIME objects; some protocols add a length header
- Error reporting: not applicable
- Syntax: headers in ASCII text (mail format); blank line; data object encoded according to header

**Multipart Type**

- `Content-Type: type/subtype; parameters`
- `--98766789`
- `Content-Transfer-Encoding: quoted-printable`
- `Content-Type: text/plain`

  Bob, Please find a picture of a crepe.

- `--98766789`
- `Content-Transfer-Encoding: base64`
- `Content-Type: image/jpeg`

  base64 encoded data ....
  
  base64 encoded data

- `--98766789--`

**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 (RFC1939)**

- TCP Port 110
- Command/response interaction
  - ASCII commands terminated by CR LF
  - Response +OK or –ERR
- Multiline response terminated by CR LF. CR LF
  - Line beginning with “.” has an additional “.” inserted at the front. Client removes leading “.” from response lines.

**POP3 stages**

- Greeting
  - +OK POP3 server ready
- Authorisation
- Transaction
- Update

**POP3 authorisation**

- USER and PASS commands
  - Password as clear text. **Security risk.**
- APOP username digest
  - Timestamp in greeting
    - +OK POP3 server ready <186.697170952@x.com>
  - timestamp + secret → MD5 digest (16 bytes)

**POP3 transaction stage**

- **STAT** +OK 2 1328 *(count, total size)*
- **LIST** +OK 1 456 *(msg number, size)* 3 872 .
- **RETR 3** +OK message follows *(message contents)* .
- **DELE 3** +OK
### POP3 transaction stage

- **RSET** +OK (unmark deletions)
- **NOOP** +OK
- **QUIT** +OK
- **TOP 1 12** +OK *(first 12 lines of message 1)*
- **UIDL** +OK
  1. whqtswO00WBw418f5wZ
  2. QhdPYR:00WBw1Ph7x7

### POP3 Design

- Dialog: Take turns
- Representation: ASCII text (email)
- Security: Secure authorisation option
- Transport-layer: TCP
- Framing: CRLF; CRLF \"\" CRLF
- Error reporting: +OK -ERR
- Syntax: ASCII text commands and parameters
- State: Both (per session: protocol stage; authorised user; items marked for deletion)

### IMAP protocol (RFC 2060)

- Every message in the mailbox is assigned a unique identifier.
- Each command is uniquely numbered by client – server uses same number in response.
- Capability to fetch parts of messages: header, body, MIME parts.
  - C: A654 FETCH 2:4 (FLAGS BODY[HEADER.FIELDS (DATE FROM)])
  - S: * 2 FETCH ....
  - S: * 3 FETCH ....
  - S: * 4 FETCH ....
  - S: A654 OK FETCH completed

- Message flags: seen, answered, flagged, deleted, draft, recent.
- Multiple mail folders – select folder, create and delete folders, move or copy messages between folders.
- Search messages on server.
  - C: A282 SEARCH FLAGGED SINCE 1-Feb-1994 NOT FROM “Smith”
  - S: * SEARCH 2 84 882
  - S: A282 OK SEARCH completed
**IMAP example**

- C: A81 CREATE foo
- S: A81 OK CREATE completed
- C: A82 LIST "" ""
- S: "" LIST () "." art
- S: "" LIST () "." foo
- S: "" LIST () "." foo.bar
- S: A82 OK LIST completed
- C: A83 DELETE art
- S: A83 OK DELETE completed

**IMAP Results codes**

- OK – Operation successful
- NO – Operation failed
- BAD – Command unknown or invalid parameters

**IMAP Design**

- Dialog: Pipelined commands
- Representation: ASCII text
- Security: Authentication option; protection option
- Transport-layer: TCP
- Framing: CRLF; continuation flag
- Error reporting: OK NO BAD
- Syntax: ASCII commands and parameters
- State: Both: Per session (authenticated user; selected folder); Server: folders and items status maintained between sessions

**POP vs IMAP**

- Simple protocol
  - RFC 1939 is 5800 words
  - One mailbox
  - Download and delete
  - E-mail only
- Complex protocol
  - RFC 2060 is 21000 words
  - Multiple mail folders
  - Manage e-mail repository
  - Subscribe to net news groups
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 transfer
- 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 files 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

FTP Data Transfer

- STREAM mode: close connection to indicate end of file.
- BLOCK, COMPRESSED modes: EOF indication in data

Data connection setup

- Client listen on port used by control connection, server connect from 20
- Client send PORT p command, listen on port p
- Client send PASV command, server send port number s in response, server listen on port s
FTP Design

- Dialog: Take turns; out-of-band data
- Representation: Text files CRLF; binary files
- Security: Nil: Passwords in plain text
- Transport-layer: TCP
- Framing: CRLF: connection “blasting” for files
- Error reporting: 3-digit codes; human readable text
- Syntax: ASCII commands and parameters
- State: Both: per session (authorised user)

DNS: Domain Name System

- People: many identifiers:
  - TFN, name, passport number
- Internet hosts, routers:
  - IP address (32 bit) - used for addressing datagrams
  - “name”, e.g., pompeii.ics.mq.edu.au
  - used by humans
- Core Internet function, implemented as application-layer protocol
- Domain Name System:
  - distributed database implemented in hierarchy of many name servers
  - administration protocol
  - complexity at network’s “edge”

RFC1034, 1035, 882, 883

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 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 mapping not known
  - gets name mapping
  - returns mapping to local name server

13 root name servers worldwide
Simple DNS example

- **host** `surf.eurecom.fr` wants IP address of `gaia.cs.umass.edu`
  - 1. Host contacts 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
  - 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 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
  - identification: 16 bit # for query, reply to query uses same # (RFC1035: MSB first)
  - flags:
    - query or reply
    - recursion desired
    - recursion available
    - reply is authoritative

- **Identification**
  - # questions
  - #answer RRs
  - #authority RRs
  - #additional RRs

- **Flags**
  - Questions (variable number)
  - Answers (variable # RRs)
  - Authority (variable # RRs)
  - Additional info (variable # RRs)

**DNS Design**

- **Dialog**: Query-response
- **Representation**: RRs; 16-bit MSB first
- **Security**: Nil
- **Transport-layer**: UDP or TCP
- **Framing**: Datagram; RR counts
- **Error reporting**: Error flag bits
- **Syntax**: Binary data
- **State**: Stateless protocol (query-response)
**BOOTP RFC951**

- Allows diskless clients to bootstrap themselves using IP/UDP.
  - 1. BOOTP configuration parameters (IP, file)
  - 2. TFTP (or FTP etc) download boot file from server
- Client broadcasts request to reserved port:
  - Hardware address identifies client
- Response broadcast to second reserved port contains boot information:
  - IP address (based on hardware address)
  - File name to boot
- Optional vendor-specific area at end of packet

---

**BOOTP message format**

<table>
<thead>
<tr>
<th>Op (1)</th>
<th>Htype (1)</th>
<th>Hlen (1)</th>
<th>Hops (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xid (4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Secs (2)</td>
<td>Unused (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ciaddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yiaddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Siaddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Giaddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chaddr (16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sname (64)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>File (128)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vend (64)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**BOOTP message fields**

- **Op**: 1=BOOTREQUEST, 2=BOOTREPLY
- **Htype**, **Hlen**: hardware address type and length
- **Xid**: transaction id (client random number)
- **Secs**: time elapsed since client began booting
- **Ciaddr**: client IP address (if known)
- **Yiaddr**: 'your' (client) IP address (from server)
- **Chaddr**: Client hardware address
- **Sname**: Optional server host name (ASCII, NUL terminated)
- **Vend**: Vendor-specific area.

---

**Tagged vend RFC1497**

- Defines tagged data fields in the vendor specific area (specified magic number).

  ![Tagged vend RFC1497](image)

- Opportunity to define additional tags through IANA.
**BOOTP Design**

- Dialog: Query-response
- Representation: Binary/text data; MSB first
- Security: Nil
- Transport-layer: UDP
- Framing: Fixed-size Datagram
- Error reporting: Nil – discard packet
- Syntax: Fixed fields (RFC1497: tagged fields)
- State: Stateless protocol (query-response)

**DHCP RFC2131**

- Dynamic Host Configuration Protocol
  - Delivers host configuration parameters from server
    - DHCP server looks up client hardware address (or identifier option)
    - Client requests configuration parameters and receives responses
  - Method to allocate network addresses to hosts
  - Client-server model
  - Based on BOOTP protocol and RFC1497

**DHCP address allocation**

- Client requests address allocation for a lease time.
- Server(s) guarantee not to reallocate address during lease.
- Server(s) attempt to return same address each time a client requests its address.
- Client can extend lease and/or release address.

**DHCP IP address allocation**

- Automatic: server allocates permanent IP address to host
- Dynamic: server allocates IP address for limited time or until client releases it
  - Useful for limited pool of IP addresses
  - IP address can be reused by another client later
- Manual: administrator configures IP addresses to be allocated by the server
### DHCP message format

<table>
<thead>
<tr>
<th>Op (1)</th>
<th>Htype (1)</th>
<th>Hlen (1)</th>
<th>Hops (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xid (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Secs (2)</td>
<td>Flags (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Claddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yiaddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Siaddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Giaddr (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chaddr (16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sname (64)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>File (128)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Options (variable) includes DHCP message type</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### DHCP Design

- Dialog: Query-response
- Representation: Binary/text data; MSB first
- Security: Nil
- Transport-layer: UDP
- Framing: Datagram
- Error reporting: DHCPNACK message
- Syntax: Fixed fields; tagged fields (RFC1497)
- State: Server maintains IP lease data

### P2P File Transfer

- Peer acts as client and as server.
  - To fetch file: act as client to another peer.
  - To ‘publish’ files: act as server for other peers.
- How to find file on peers?
  - Napster: central database of peers and files offered (could be shut down).
  - Gnutella: connect to several known peers; requests propagate through connections.
  - KaZaA: some peers designated as group leaders – index their group; requests propagate between leaders.
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

- Socket programming
  - Client/server implementation
    - Using TCP, UDP sockets

- Specific protocols:
  - Telnet
  - HTTP, FTP, P2P
  - SMTP, POP3, IMAP
  - DNS
  - DHCP

---

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