Application Layer

Jim KUROSE
www-aml.cs.umass.edu

Derived from Chapter 2 of

http://occawlonline.pearsoned.com/bookbind/pubbooks/kurose-ross1/chapter0/deluxe.html
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
Applications and application-layer protocols

Application: communicating, distributed processes
- running in network hosts in "user space"
- exchange messages to implement app
- e.g., email, file transfer, the Web

Application-layer protocols
- one "piece" of an app
- define messages exchanged by apps and actions taken
- user 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,
- e.g.: request WWW page, send email

**Server:**
- provides requested service to client
- e.g., sends requested WWW page, receives/stores received email
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>loss-tolerant</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>
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 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: 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>NSF</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>
WWW: the http protocol

http: hypertext transfer protocol

- WWW’s application layer protocol
- client/server model
  - **client**: browser that requests, receives, “displays” WWW objects
  - **server**: WWW 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 WWW server (http server)
- TCP connection closed

http is “stateless”
- server maintains no information about past client requests

Protocols that maintain “state” are complex!
- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled
http example

Suppose user enters URL

www.someSchool.edu/someDepartment/home.index

(contains text, references to 10 jpeg images)

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

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

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 connection: one object in each TCP connection
  - some browsers create multiple TCP connections simultaneously - one per object
- persistent connection: multiple objects transferred within one TCP connection
http message format: request

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

```
GET /somedir/page.html HTTP/1.1
Connection: close
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```
(extra carriage return, line feed)
http request message: general format

request line

header lines

Entity Body
http message format: reply

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 data data data data data ...

header lines

data, e.g., requested html file
http reply 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 WWW server:

   \texttt{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:

   \texttt{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: 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

```
client    server
usual http request msg
401: authorization req. WWW authenticate:
usual http request msg + Authorization:line
usual http response msg
usual http request msg + Authorization:line
usual http response msg
```
User-server interaction: cookies

- server sends “cookie” to client in response
  
  ```
  Set-cookie: #
  ```

- client present cookie in later requests
  
  ```
  cookie: #
  ```

- server matches presented-cookie with server-stored cookies
  - authentication
  - remembering user preferences, previous choices

```
usual http request msg

usual http response

Set-cookie: #

usual http response msg

usual http request msg

cookie: #

usual http response msg

usual http request msg

cookie: #

usual http response msg

cookie-specific action

cookie-specific action
```
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

- server: response contains object if cached copy modified:
  - HTTP/1.0 200 OK
    - <data>
Web Caches (proxy server)

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

- user sets browser: WWW 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
Why WWW 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
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 commands, responses

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

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

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

User Agent
- a.k.a. “mail reader”
- composing, editing, reading mail messages
- e.g., Eudora, pine, 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 server 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
  - closure
- command/response interaction
  - commands: ASCI text
  - response: status code and phrase
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
smtp: final words

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)

Comparison with http
- http: pull
- email: push
- both have ASCII command/response interaction, status codes
- http: multiple objects in file sent in separate connections
- smtp: multiple message parts sent in one connection
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

- line containing only `.`
Message format: multimedia extensions

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

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

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

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
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
**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
```

```