Chapter 2: outline

2.1 principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 electronic mail
   - SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP

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
- creating network applications
  - socket API
Some network apps

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- Search
- Location- and context-sensitive apps
- ...
- ...

Creating a network app

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

no need to write software for network-core devices
- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation
**Application architectures**

possible structure of applications:

- client-server
- peer-to-peer (P2P)

---

**Client-server architecture**

- **server:**
  - always-on host
  - permanent IP address
  - data centers for scaling

- **clients:**
  - communicate with server
  - may be intermittently connected
  - may have dynamic IP addresses
  - do not communicate directly with each other

---

Application Layer 2-5

Application Layer 2-6
**P2P architecture**

- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - **self scalability** – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management

**Processes communicating**

**process:** program running within a host

- within same host, two processes communicate using **inter-process communication** (defined by OS)
- processes in different hosts communicate by exchanging **messages**

**clients, servers**

- **client process:** process that initiates communication
- **server process:** process that waits to be contacted

**aside:** applications with P2P architectures have client processes & server processes
**Sockets**

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

**Addressing processes**

- to receive messages, process must have **identifier**
- host device has unique 32-bit IP address
- **Q:** does IP address of host on which process runs suffice for identifying the process?
  - **A:** no, many processes can be running on same host
- **identifier** includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…
App-layer protocol defines

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

open protocols:
- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:
- e.g., Skype

What transport service does an app need?

data integrity
- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

throughput
- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

security
- encryption, data integrity, ...

Application Layer 2-11
### Transport service requirements: common apps

<table>
<thead>
<tr>
<th>Application</th>
<th>Data Loss</th>
<th>Throughput</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</td>
</tr>
<tr>
<td>Stored audio/video</td>
<td>Loss-tolerant</td>
<td>Same as above</td>
<td></td>
</tr>
<tr>
<td>Interactive games</td>
<td>Loss-tolerant</td>
<td>Few kbps up</td>
<td>Yes, few secs</td>
</tr>
<tr>
<td>Text messaging</td>
<td>No loss</td>
<td>Elastic</td>
<td>Yes, 100’s</td>
</tr>
</tbody>
</table>

### Internet transport protocols services

**TCP service:**
- **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 throughput guarantee, security
- **Connection-oriented**: setup required between client and server processes

**UDP service:**
- **Unreliable data transfer** between sending and receiving process
- **Does not provide**: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup

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>HTTP (e.g., YouTube), RTP [RFC 1889]</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>SIP, RTP, proprietary</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>(e.g., Skype)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Securing TCP

**TCP & UDP**
- no encryption
- cleartext passwds sent into socket traverse Internet in cleartext

**SSL**
- provides encrypted TCP connection
- data integrity
- end-point authentication

**SSL is at app layer**
- Apps use SSL libraries, which “talk” to TCP

**SSL socket API**
- cleartext passwds sent into socket traverse Internet encrypted
- See Chapter 7
Chapter 2: outline

2.1 principles of network applications
   • app architectures
   • app requirements
2.2 Web and HTTP
2.3 FTP
2.4 electronic mail
   • SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP

Web and HTTP

First, a review…

❖ 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, e.g.,

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, (using HTTP protocol) and displays Web objects
  - server: Web server sends (using HTTP protocol) objects in response to requests

HTTP request
HTTP response

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

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

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

aside
HTTP connections

**non-persistent HTTP**
- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

**persistent HTTP**
- multiple objects can be sent over single TCP connection between client, server

Non-persistent HTTP

suppose user enters URL:

www.someSchool.edu/someDepartment/home.index

1. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80
2. HTTP client sends HTTP request message (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index
3. HTTP server receives request message, forms response message containing requested object, and sends message into its socket
Non-persistent HTTP (cont.)

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

Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back

HTTP 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
- non-persistent HTTP response time = 2RTT + file transmission time
**Persistent HTTP**

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

**persistent HTTP:**
- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- 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 /index.html HTTP/1.1
Host: www-net.cs.umass.edu
User-Agent: Firefox/3.6.10
Accept: text/html,application/xhtml+xml
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7
Keep-Alive: 115
Connection: keep-alive

```

Application Layer 2-25
HTTP request message: general format

<table>
<thead>
<tr>
<th>method</th>
<th>sp</th>
<th>URL</th>
<th>sp</th>
<th>version</th>
<th>cr</th>
<th>lf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>header field name</td>
<td>value</td>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>header field name</td>
<td>value</td>
<td>cr</td>
<td>lf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cr</td>
<td>if</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>entity body</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

request line
header lines
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
Date: Sun, 26 Sep 2010 20:09:20 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT
ETag: "17dc6-a5c-bf716880"
Accept-Ranges: bytes
Content-Length: 2652
Keep-Alive: timeout=10, max=100
Connection: Keep-Alive
Content-Type: text/html;
   charset=ISO-8859-1

```

data, e.g., requested HTML file

```
data data data data data ...
```
HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:
  200 OK
  - request succeeded, requested object later in this msg
  301 Moved Permanently
  - requested object moved, new location specified later in this msg
    (Location:)
  400 Bad Request
  - request msg not understood by server
  404 Not Found
  - requested document not found on this server
  505 HTTP Version Not Supported

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

   telnet cis.poly.edu 80

   opens TCP connection to port 80
   (default HTTP server port) at cis.poly.edu.
   anything typed in sent
to port 80 at cis.poly.edu

2. type in a GET HTTP request:

   GET /~ross/ HTTP/1.1
   Host: cis.poly.edu

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

3. look at response message sent by HTTP server!

   (or use Wireshark to look at captured HTTP request/response)
User-server state: cookies

many Web sites use cookies

four components:

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

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

Application Layer 2-34
Cookies: keeping “state” (cont.)

Cookies (continued)

what cookies can be used for:
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

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

aside: cookies and privacy:
- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites
Aside - HTTP: cookies and advertising

This is FYI only (“not on test”):
- third-party cookies: ad network server tracking user web page accesses across multiple sites

HTTP: homepage, image, ad (v1)

Web page at ilovedisco.com

HTML file contains text, and two <IMG> tags. Both images are stored on ilovedisco.com

1. HTTP GET request to ilovedisco.com for homepage  
2. ilovedisco.com server send homepage HTML file to browser via HTTP reply  
3. Browser reads homepage HTML file, sees <img> tag (John Travolta), requests first image via HTTP GET  
4. ilovedisco.com server sends first image to browser via HTTP reply  
5. Browser sees second <img> tag (the ad) and requests ad image via HTTP GET  
6. ilovedisco.com server sends ad image to browser via HTTP reply
HTTP: homepage, image, ad (v1): observations

- all web page content at ilovedisco.com
- HTML file, Travolta image, ad are separate files on server - composed into webpage at client
- same content would be served to all browsers
- ilovedisco.com would sell ad space directly to Disco Hits

HTTP: homepage, image, ad (v2)

1-4. As before, home page, and first image (J. Travolta) downloaded from ilovedisco.com
5. Browser reads second <img> tag (ad) requests ad image from MegaAd.com via HTTP GET, with referer field: ilovedisco.com
6. MegaAd.com server sends ad image to browser via HTTP reply, knowing image is to be embedded in page from ilovedisco.com
HTTP: homepage, image, ad (v2): observations

- ad content *not* served by ilovedisco.com
- ilovedisco.com could sell ad space directly to Disco Hits who provides content
- ilovedisco.com could sell ad space to *ad network*, who serves content
  - ad network serves as aggregator for *many* products/Companies,
  - knows “referer”
  - ilovedisco wouldn’t even know what ad is going to be displayed in its page!

HTTP: homepage, image, ad (v3): cookies

1. HTTP GET request to ilovedisco.com for homepage
2. ilovedisco.com server send homepage HTML file to browser via HTTP reply, with ilovedisco cookie: 82. *ilovedisco cookie: 82 Cookie stored at client.*
3. HTTP GET for image, GET message contains with ilovedisco cookie: 82
4. ilovedisco.com server sends first image to browser via HTTP reply
5. Browser sees requests ad image via HTTP GET message with ilovedisco cookie: 82
6. ilovedisco.com server sends ad image to browser via HTTP reply
A short history of the best music ever.

Disco is a genre of music from the 1970's...

HTTP: homepage, image, ad (v3): cookies

7. HTTP GET request to ilovedisco.com for homepage with ilovedisco cookie: 82 from last week
8. ilovedisco.com server sees cookie in GET msg, sends homepage HTML file to browser via HTTP reply containing DIFFERENT AD IMAGE from last time
9. HTTP GET for Travolta image, GET contains with ilovedisco cookie: 82
10. ilovedisco.com server sends Travolta image
11. Browser requests new ad image via HTTP GET with ilovedisco cookie: 82
12. ilovedisco.com server sends new ad image to browser via HTTP reply

HTTP: homepage, image, ad (v3): observations

- cookies can be used to personalize (target) content (e.g., ads) to client based on past interaction with this server
- web server can dynamically generate content depending on what client has done/seen in past
HTTP: Third party cookies

1-5. As before, home page, and first image (J. Travolta) downloaded from ilovedisco.com, request made for ad image from MegaAd.com via HTTP GET, with referer field: ilovedisco.com

6. MegaAd.com server sends ad image to browser via HTTP reply, knowing image is to be embedded in page from ilovedisco.com, adds its own cookie MegaAd: 814. Remembers that cookie #814 owner had visited ilovedisco.com

Third party cookie: when you visit a web page, a third website is able to put a cookie on your browser (as shown here).

HTTP: Targeted advertising (v4)

1-6. client visits ilovedisco.com, disco ad served by MegaAd.com

7-10. client visits iloveNY.com, HTML text and image served by iloveNY.com

11. client contacts MegaAd.com to get ad to display, includes MegaAd cookie # 814

12. MegaAd.com sees referred request from iloveNY.com, sees cookie 814, knows client visited disco site earlier, serves targeted content ad: disco + NY
HTTP: Targeted advertising - observations

- **third party cookies** allow third party (e.g., MegaAds.com) to track user access over multiple web sites (any site with MegaAd link)
- MegaAd uses past user activity to **micro-target specific ads** to specific users
  - MegaAd can charge ad creators more to place their ads in micro-targeted manner (since user is more likely to be interested in ad)
- users not aware of third party cookies and tracking
  - invasion of privacy ????

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
  - server for original requesting client
  - client to origin server
- typically cache is installed by ISP (university, company, residential ISP)

why Web caching?
- reduce response time for client request
- reduce traffic on an institution’s access link
- Internet dense with caches: enables “poor” content providers to effectively deliver content (so too does P2P file sharing)

Caching example:

assumptions:
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:
- LAN utilization: 15%
- access link utilization ≤ 99%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + usecs

Application Layer 2-49

Application Layer 2-50
Caching example: fatter access link

**assumptions:**
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

**consequences:**
- LAN utilization: 15%
- access link utilization: 99%
- total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + usecs

**Cost:** increased access link speed (not cheap!)

---

Caching example: install local cache

**assumptions:**
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

**consequences:**
- LAN utilization: 15%
- access link utilization: ?
- total delay: ?

*How to compute link utilization, delay?*

**Cost:** web cache (cheap!)
Caching example: install local cache

Calculating access link utilization, delay with cache:

- Suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin
- Access link utilization:
  - 60% of requests use access link
- Data rate to browsers over access link
  \[ 0.6 \times 1.50 \text{ Mbps} = 0.9 \text{ Mbps} \]
  - Utilization = 0.9/1.54 = .58
- Total delay
  - \[ 0.6 \times (\text{delay from origin servers}) + 0.4 \times (\text{delay when satisfied at cache}) \]
  - \[ = 0.6 \times 2.01 + 0.4 \times (\text{~msecs}) \]
  - \[ = \sim 1.2 \text{ secs} \]
  - Less than with 154 Mbps link (and cheaper too!)

Conditional GET

- **Goal:** don’t send object if cache has up-to-date cached version
  - No object transmission delay
  - Lower link utilization
- **Cache:** 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
- **Request:** HTTP request msg
  - If-modified-since: <date>
  - HTTP response
    - HTTP/1.0 304 Not Modified
  - Object not modified before <date>
- **Response:** HTTP response
  - If-modified-since: <date>
  - HTTP request msg
    - HTTP/1.0 200 OK
    - <data>
  - Object modified after <date>
Chapter 2: outline

2.1 principles of network applications
   ▪ app architectures
   ▪ app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
   ▪ SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

2.7 socket programming with UDP and TCP

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, using TCP
- Client authorized over control connection
- Client browses remote directory, sends commands over control connection
- When server receives file transfer command, server opens 2nd TCP data connection (for file) to client
- After transferring one file, server closes data connection

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 network applications
   • app architectures
   • app requirements
2.2 Web and HTTP
2.3 FTP
2.4 electronic mail
   • SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP

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., Outlook, Thunderbird, iPhone mail client
• 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 (like HTTP, FTP)
  - **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 “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 MAIL FROM, RCPT TO: commands!
- Body: the “message”
  - ASCII characters only

Mail access protocols

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

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

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

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

POP3 (more) and IMAP

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

**IMAP**
- keeps all messages in one place: at server
- allows user to organize messages in folders
- keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name
Chapter 2: outline

2.1 principles of network applications
   - app architectures
   - app requirements
2.2 Web and HTTP
2.3 FTP
2.4 electronic mail
   - SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP

DNS: domain name system

**people:** many identifiers:
- SSN, name, passport #

**Internet hosts, routers:**
- IP address (32 bit) - used for addressing datagrams
- “name”, e.g., www.yahoo.com - used by humans

**Q:** how to map between IP address and name, and vice versa?

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

**DNS services**
- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

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

*A: doesn’t scale!*

---

**DNS: a distributed, hierarchical database**

```
Root DNS Servers
- ... com DNS servers
  - yahoo.com DNS servers
  - amazon.com DNS servers
- ... org DNS servers
  - pbs.org DNS servers
- ... edu DNS servers
  - poly.edu DNS servers
  - umass.edu DNS servers
```

*client wants IP for www.amazon.com; 1st approx:*
- client queries root server to find com DNS server
- client queries .com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com*
DNS: root name servers

- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server

13 root name "servers" worldwide

TLD, authoritative servers

top-level domain (TLD) servers:
- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TLD

authoritative DNS servers:
- organization’s own DNS server(s), providing authoritative hostname to IP mappings for organization’s named hosts
- can be maintained by organization or service provider
**Local DNS name server**

- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called “default name server”
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

**DNS name resolution example**

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

*iterated query:*
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”
**DNS name resolution example**

recursive query:
- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy!

**DNS: caching, updating records**

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard
  - RFC 2136
DNS records

DNS: distributed db storing resource records (RR)

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

type=A
- name is hostname
- value is IP address

type=NS
- name is domain (e.g., foo.com)
- value is hostname of authoritative name server for this domain

type=CNAME
- name is alias name for some “canonical” (the real) name
- www.ibm.com is really servereast.backup2.ibm.com
- value is canonical name

type=MX
- value is name of mailserver associated with name

DNS protocol, messages

- 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

<table>
<thead>
<tr>
<th>identification</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td># questions</td>
<td># answer RRs</td>
</tr>
<tr>
<td># authority RRs</td>
<td># additional RRs</td>
</tr>
<tr>
<td>questions (variable # of questions)</td>
<td>answers (variable # of RRs)</td>
</tr>
<tr>
<td></td>
<td>authority (variable # of RRs)</td>
</tr>
<tr>
<td></td>
<td>additional info (variable # of RRs)</td>
</tr>
</tbody>
</table>

**Inserting records into DNS**

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

DDoS attacks
- Bombard root servers with traffic
  - Not successful to date
  - Traffic Filtering
  - Local DNS servers cache IPs of TLD servers, allowing root server bypass
- Bombard TLD servers
  - Potentially more dangerous

Redirect attacks
- Man-in-middle
  - Intercept queries
- DNS poisoning
  - Send bogus relies to DNS server, which caches

Exploit DNS for DDoS
- Send queries with spoofed source address: target IP
- Requires amplification

Chapter 2: outline

2.1 principles of network applications
  - app architectures
  - app requirements
2.2 Web and HTTP
2.3 FTP
2.4 electronic mail
  - SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP
Pure P2P architecture

- no always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

eXamples:
  - file distribution (BitTorrent)
  - Streaming (KanKan)
  - VoIP (Skype)

File distribution: client-server vs P2P

Question: how much time to distribute file (size $F$) from one server to $N$ peers?

- peer upload/download capacity is limited resource

\[ u_i: \text{peer upload capacity} \]
\[ d_i: \text{peer i download capacity} \]

\[ u_s: \text{server upload capacity} \]
\[ u_N: \text{server upload capacity} \]
\[ d_N: \text{server upload capacity} \]
File distribution time: client-server

- **server transmission**: must sequentially send (upload) $N$ file copies:
  - time to send one copy: $F/u_s$
  - time to send $N$ copies: $NF/u_s$
- **client**: each client must download file copy
  - $d_{\text{min}} = \text{min client download rate}$
  - min client download time: $F/d_{\text{min}}$

The total distribution time for the client-server approach is:

$$D_{\text{c-s}} \geq \max\{NF/u_s, F/d_{\text{min}}\}$$

which increases linearly in $N$.

---

File distribution time: P2P

- **server transmission**: must upload at least one copy
  - time to send one copy: $F/u_s$
- **client**: each client must download file copy
  - min client download time: $F/d_{\text{min}}$
- **clients**: as aggregate must download $NF$ bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

The total distribution time for the P2P approach is:

$$D_{\text{P2P}} \geq \max\{F/u_s, F/d_{\text{min}}, NF/(u_s + \Sigma u_i)\}$$

which increases linearly in $N$ ... but so does this, as each peer brings service capacity.
**Client-server vs. P2P: example**

client upload rate = \( u \), \( \frac{F}{u} = 1 \) hour, \( u_s = 10u \), \( d_{\text{min}} \geq u_s \)

![Graph showing minimum distribution time for P2P and Client-Server comparison.](image)

**P2P file distribution: BitTorrent**

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

**tracker**: tracks peers participating in torrent

**torrent**: group of peers exchanging chunks of a file

Alice arrives …
… obtains list of peers from tracker
… and begins exchanging file chunks with peers in torrent
P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- churn: peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

requesting chunks:
- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

sending chunks: tit-for-tat
- Alice sends chunks to those four peers currently sending her chunks at highest rate
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4
BitTorrent: tit-for-tat

(1) Alice “optimistically unchokes” Bob
(2) Alice becomes one of Bob’s top-four providers; Bob reciprocates
(3) Bob becomes one of Alice’s top-four providers

higher upload rate: find better trading partners, get file faster!

Chapter 2: outline

2.1 principles of network applications
   ▪ app architectures
   ▪ app requirements
2.2 Web and HTTP
2.3 FTP
2.4 electronic mail
   ▪ SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP
Socket programming

**goal:** learn how to build client/server applications that communicate using sockets

**socket:** door between application process and end-end-transport protocol

---

**Socket programming**

*Two socket types for two transport services:*
- **UDP:** unreliable datagram
- **TCP:** reliable, byte stream-oriented

**Application Example:**

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.
Socket programming with **UDP**

**UDP: no “connection” between client & server**
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

**UDP: transmitted data may be lost or received out-of-order**

**Application viewpoint:**
- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

---

**Client/server socket interaction: UDP**

**server (running on serverIP)**
- create socket, port=x:
  - `serverSocket = socket(AF_INET,SOCK_DGRAM)`
- read datagram from `serverSocket`
- write reply to `serverSocket` specifying client address, port number

**client**
- create socket:
  - `clientSocket = socket(AF_INET,SOCK_DGRAM)`
- Create datagram with server IP and port=x; send datagram via `clientSocket`
- read datagram from `clientSocket`
- close `clientSocket`
**Example app: UDP client**

*Python UDPClient*

- include Python's socket library
- create UDP socket for server
- get user keyboard input
- Attach server name, port to message; send into socket
- read reply characters from socket into string
- print out received string and close socket

```python
from socket import *

serverName = 'hostname'
serverPort = 12000
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message, (serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

**Example app: UDP server**

*Python UDPServer*

- create UDP socket
- bind socket to local port number 12000
- loop forever
- read from UDP socket into message, getting client's address (client IP and port)
- send upper case string back to this client

```python
from socket import *

serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"

while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```
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 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 that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

**Application viewpoint:**
TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

---

**Client/server socket interaction: TCP**

**server (running on hostid)**
- create socket, port=x, for incoming request:
  - serverSocket = socket()

  wait for incoming connection request:
  - connectionSocket = serverSocket.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

---

Application Layer 2-103

Application Layer 2-104
**Example app: TCP client**

**Python TCPClient**

```python
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence: ')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

**Example app: TCP server**

**Python TCPServer**

```python
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024)
capitalizedSentence = sentence.upper()
connectionSocket.send(capitalizedSentence)
connectionSocket.close()
```
Chapter 2: summary

our study of network apps now complete!

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

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

Important themes:

- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- “complexity at network edge”
Chapter 1
Additional Slides

Introduction 1-109