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 \texttt{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
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 2 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

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
- 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**

---

**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:
  a. Verisign, Los Angeles CA (5 other sites)
  b. USC-ISI Marina del Rey, CA
  c. Cogent, Herndon, VA (5 other sites)
  d. U Maryland College Park, MD
  e. NASA Mt View, CA
  f. Internet Software C. Palo Alto, CA (and 48 other sites)
  g. US DoD Columbus, OH (5 other sites)
  h. ARL Aberdeen, MD
  i. Netnod, Stockholm (37 other sites)
  j. Verisign, Dulles VA (69 other sites)
  k. RIPE London (17 other sites)
  l. ICANN Los Angeles, CA (41 other sites)
  m. WIDE Tokyo (5 other sites)
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”

Diagram:
- Requesting host at cis.poly.edu
- Contact local DNS server dns.poly.edu
- Contact TLD DNS server
- Contact authoritative DNS server dns.cs.umass.edu
- Return IP address for gaia.cs.umass.edu
DNS name resolution example

recursive query:
- puts burden of name resolution on contacted name server
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:** \( (\text{name}, \text{value}, \text{type}, \text{ttl}) \)

- **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
  - **value** is canonical name

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

---

Application Layer 2-23
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
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-to-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 specifying client address, port number

close clientSocket
Example app: UDP client

Python UDPClien

- 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

```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)
```

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
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`
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()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

Application Layer 2-34
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()
```

create TCP welcoming socket
server begins listening for incoming TCP requests
loop forever
server waits on accept() for incoming requests, new socket created on return
read bytes from socket (but not address as in UDP)
close connection to this client (but not welcoming socket)
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”