Chapter 2 Network Applications Contd

Application Layer Protocols
- http
- ftp, tftp
- telnet, rlogin
- snmp, dns
- smtp, pop3, imap

Unix Socket Interface

Chapter 2 outline
- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.3.1 FTP
- 2.3.2 telnet, rlogin, snmp
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P File sharing

Unix Socket Interfaces
### Sockets Domain and Socket Types

- **UNIX domain sockets** are named with UN*X paths. **UNIX domain** sockets communicate only between processes on a single host.
- **Internet domain** communication uses the **TCP/IP** suite.
  - **Stream** sockets enable processes to communicate using **TCP**.
    - `SOCK_STREAM`
  - **Datagram** sockets enable processes to use **UDP** to communicate.
    - `SOCK_DGRAM`
  - **Raw sockets**
    - `SOCK_RAW`

### Socket Libraries

- The socket interface routines are in a library that must be linked with the application.
  - `libsocket.so`, `libsocket.a`
- Compile and Link
  - e.g. `gcc main.c -o main -lsocket -lnsl`
- Mechanism for System Calls
  - UNIX system calls and library routines
    ```c
    if ((code = syscall()) < 0) {
        perror("syscall");
        exit(1);
    }
    ```

### Interprocess Communication (IPC)

- **Issues**
  - Creating sockets
  - Naming (identifying) sockets
  - Sending/receiving data over sockets
**Connection-Oriented Communication w/ Stream Sockets**

**Socket Creation**

```c
int s = socket(domain, type, protocol);
```

- **domain**: AF_INET, AF_UNIX
- **type**: SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW
- **protocol**: 0

**Example:**

```c
s = socket(AF_INET, SOCK_STREAM, 0);
```

**Naming Sockets**

```c
int bind(s, name, namelen)
```

- **int s**;
- **struct sockaddr *name**;
- **int namelen**;

  - Used to associate a port to socket s on the local m/c
  - Needed at server side (typically)
  - Not needed when doing only `connect()`
**Socket Address**

- Several types of socket addresses
- You will use sockaddr_in, because you use AF_INET
- sockaddr_in is a C structure with 3 important fields:
  - sin_family
  - sin_addr
  - sin_port

---

**Internet Address and Ports**

- sin_addr values
  - Four bytes
  - e.g. csx01 is 141.218.143.190 (see /etc/hosts)
  - If you specify INADDR_ANY, local host IP address gets filled in
- sin_port values
  - 0–1024 reserved for system
  - well known ports
    - http is port 80
    - telnet is port 23
  - if you specify 0, the system picks a port

---

**Understanding Ports**

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer
Data Transfer

Send or receive a message with the normal read and write interfaces:
- write(s, buf, sizeof buf);
- read(s, buf, sizeof buf);
You can also use send and recv:
- send(s, buf, sizeof buf, flags);
- recv(s, buf, sizeof buf, flags);

Other System Calls

int connect(sd, name, namelen)
- specifies peer with which sd is to be associated
int listen(sd, backlog)
- specifies maximum backlog of connections a server will allow
int accept (sd, addr, addrlen) : new_sd is returned
- extracts first connection off queue of pending connections
int close (sd)
- deletes descriptor from system tables;
**select()—Synchronous I/O Multiplexing**

select() gives you the power to monitor several sockets at the same time. It'll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions, if you really want to know that.

```c
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

**htonl(), htons(), ntohs(), ntohl()**
- convert between network byte order and host byte order
- Required since some machines are big endian and others are little endian (network order is big endian)
- Convert to network order before sending, and back to host order after receiving

```c
my_addr.sin_port = htons(MYPORT); /* short, network byte order */
bzero(&my_addr.sin_zero, 8); /* zero the rest of the struct */
```

**Connection-Oriented Communication w/ Stream Sockets**
**Chapter 2 outline**

- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing

**Socket programming with UDP**

UDP: no "connection" between client and server
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

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

Application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

**Client/server socket interaction: UDP**

**Server (running on hostid)**

create socket, port=x
create serverSocket = DatagramSocket()
read request from serverSocket
write reply to serverSocket specifying client host address, port number

**Client**

create socket, certificate = DatagramSocket()
create, address=(hostid, port=x)
read datagram request using certificate
read reply from serverSocket
close clientSocket
Client/server socket interaction: TCP

Server (running on hostid)
- Create socket, port=x, for incoming request:
  welcomeSocket = ServerSocket()
- Wait for incoming connection request:
  connectionSocket = welcomeSocket.accept()
- Read request from connectionSocket
- Write reply to connectionSocket
- Close connectionSocket

Client
- Create socket, port=x, for incoming request:
  connectionSocket = Socket()
- Connect to hostid, port=x
  clientSocket = Socket()
- Close connectionSocket
- Read reply from clientSocket
- Close clientSocket

TCP connection setup

Client/server socket interaction: UDP

Server (running on hostid)
- Create socket, clientSocket = DatagramSocket()
- Read request from clientSocket
- Write reply to clientSocket specifying client host address, port number
- Close clientSocket

Client
- Create address (hostid, port=x, send datagram request using clientSocket
- Connect to hostid, port=x
- Create socket:
  serverSocket = DatagramSocket()
- Read request from serverSocket
- Write reply to serverSocket

Example: Java client (UDP)

Client process
- Receive packet (TCP sent "byte stream")
- Send packet (TCP sent "byte stream")

Client
- Input: receives packet (TCP received "byte stream")
- Output: sends packet (TCP "byte stream")

Client socket
- Input: from network
- Output: to network
Example: Java client (UDP)

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

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];

        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();

        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
        clientSocket.send(sendPacket);

        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}
```

Example: Java server (UDP)

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

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        while(true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
            String sentence = new String(receivePacket.getData());
            System.out.print("FROM CLIENT:" + sentence);
            sendData = sentence.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, receivePacket.getAddress(), receivePacket.getPort());
            serverSocket.send(sendPacket);
        }
    }
}
```
Example: Java server (UDP), cont

```java
String sentence = new String(receivePacket.getData());
String IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
String capitalizedSentence = sentence.toUpperCase();
byte[] sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
```

End of while loop, loop back and wait for another datagram

Building a simple Web server

- handles one HTTP request
- accepts the request
- parses header
- obtains requested file from server’s file system
- creates HTTP response message:
  - header lines + file
- sends response to client
- after creating server, you can request file using a browser (e.g. IE explorer)

WebServer.java

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

class WebServer {
    public static void main(String argv[]) throws Exception {
        String requestMessageLine;
        String fileName;
        ServerSocket listenSocket = new ServerSocket(6789);
        Socket connectionSocket = listenSocket.accept();
        BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
        DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

        BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
        DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());
```
WebServer.java (contd.)

```java
requestMessageLine = inFromClient.readLine();

StringTokenizer tokenizedLine =
    new StringTokenizer(requestMessageLine);

if (tokenizedLine.nextToken().equals("GET") &&
    fileName.startsWith("/") == true )
    fileName = fileName.substring(1);

File file = new File(fileName);
inFile = new FileInputStream (fileName);
byte[] fileInBytes = new byte[numOfBytes];
inFile.read(fileInBytes);
```

WebServer.java (contd.)

```java
outToClient.writeBytes("HTTP/1.0 200 Document Follows\n\n");

if (fileName.endsWith(".jpg"))
    outToClient.writeBytes("Content-Type: image/jpeg\n\n");
if (fileName.endsWith(".gif"))
    outToClient.writeBytes("Content-Type: image/gif\n\n");

outToClient.writeBytes("Content-Length: " + numOfBytes + "\n\n");
outToClient.writeBytes("\n\n");
outToClient.writeBytes(fileInBytes, 0, numOfBytes);
connectionSocket.close();
else System.out.println("Bad Request Message");
```

Concurrent server

- Servers need to handle a new connection request while processing previous requests.
  - Most TCP servers are designed to be concurrent
- When a new connection request arrives at a server, the server accepts and invokes a new process to handle the new client.
Handling port numbers

```
cosmos% netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address     (state)     
tcp    0      0         *:23           *:*            LISTEN

cosmos% netstat -a -n -f inet
Proto Recv-Q Send-Q Local Address Foreign Address     (state)     
tcp    0      0      192.249.24.23  192.249.24.31.1029   ESTABLISHED
```

Socket programming: references

C-language tutorial (audio/slides):
- "Unix Network Programming" (J. Kurose),
  http://manic.cs.umass.edu/~amldemo/courseware/

Java-tutorials:
- "All About Sockets" (Sun tutorial),
  http://docs.sun.com/app/docs/doc/806-4125/6jd7pe6c1?awview
- "Socket Programming in Java: a tutorial,"
  http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html

Chapter 2 outline

- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3.1 TFTP, FTP
- 2.3.2 telnet, rlogin, snmp
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing
Protocols (recap...)

- Established ways of transferring information
- Provides
  - Formulas for passing messages
  - Specify the details of message formats
  - Describe how to handle error conditions
- T S S R: Type, Syntax, Semantics, Rules

TFTP

- TFTP (Trivial File Transfer Protocol) is a very simple protocol for transferring files
- Lacks most features of FTP
- Allows users to read and write files from remote host to local machine and vice versa
- uses UDP as its underlying protocol and operates on a well-known port 69

TFTP Messages

- Five different types of TFTP messages
  - RRQ: Read Request
  - WRQ: Write Request
  - DATA
  - ACK: Acknowledgement
  - ERROR
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: ports 20 (data) and 21 (control)

FTP contd..

- FTP uses two parallel TCP connections
  - control connection and data connection to transfer a file
  - The client side consists of a client user interface, a control process, and a data transfer process
  - The server side consists of a server control process and a server data transfer process

FTP: separate control, data connections

- FTP client contacts FTP server at port 21, specifying TCP as transport protocol
- Client obtains authorization over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives a command for a file transfer, the server opens a TCP data connection to client
- After transferring one file, server closes 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

FTP contd..

The three basic files
- ASCII (American Standard Code for Information Interchange) file
- EBCDIC (Extended Binary Coded Decimal Interchange Code) file
- Image File

An FTP address looks like a HTTP address, or website address except it uses the prefix ftp:// instead of http://

FTP Client software
- WS_FTP Pro is Windows-based FTP client software
- Fetch is the FTP client software used in the Macintosh systems

Anonymous FTP
- Anonymous FTP allows users to have access on files of a remote host without having an account on that machine.
- The username for anonymous FTP is "anonymous" and the password is the user's email address.

Chapter 2 outline

2.1 Principles of app layer protocols
- clients and servers
- app requirements
2.2 Web and HTTP
2.3.1 FTP
2.3.2 telnet, rlogin, snmp
2.4 Electronic Mail
- SMTP, POP3, IMAP
2.5 DNS
2.6 Socket programming with TCP
2.7 Socket programming with UDP
2.8 Building a Web server
2.9 Content distribution
- Network Web caching
- Content distribution networks
- P2P file sharing
TELNET

- TELNET is an acronym for Terminal Network
- A text-only protocol that allows users to connect to a remote computer and use programs and data as if the users were sitting in front of the computer
- TELNET uses the TCP connection and the server uses a well-known port 23 for communication

Architecture of TELNET

- A user sends a bunch of keystrokes to a terminal driver in his/her local operating system
- The characters are sent to a telnet client which transforms the characters to a universal character set called Network Virtual Terminal (NVT) characters and delivers them to the local TCP/IP stack
- These NVT characters travel through the Internet and arrive at the TCP/IP stack of the remote machine and given to telnet server which converts the character to an understandable form by the remote computer

TELNET contd...

- Modes of operations
  - Telnet implementations operate in three different modes
    - Default mode
    - Character mode
    - Line mode
- Client software
  - Typical GUI applications for Telnet include: TeraTerm and EWAN for Windows and NCSA Telnet for the Macintosh
**Rlogin**

- Rlogin is a remote login protocol provided by the BSD UNIX.
- The "rlogin" protocol requires the use of TCP.
- The contact port is 513.
- Rlogin performs remote connections and terminal emulations between two UNIX machines.
- Rlogin uses only one TCP connection – special character FF base 16 is used to distinguish between the data and command.

**SNMP**

- Simple Network Management Protocol (SNMP) is a protocol that facilitates the exchange of management information between network devices.
- SNMP uses UDP (User Datagram Protocol) as its underlying protocol on two well-known ports 161 and 162.
- Versions of SNMP are:
  - SNMPv1
  - SNMPv2c
  - SNMPv3

**Architecture of SNMP**

- Network device: A network device or the "Managed Object" is a network component that requires some form of monitoring and management.
- Agent: An agent is a mediator between the manager and the device.
- Manager: A manager or management system is a separate entity that manages the agents from a remote place.
Components of SNMP

- **The protocol** - defines the functioning of the basic operations of SNMP
- **Structure of Management Information (SMI)** - a set of rules used to specify the format for defining managed objects or the devices that are accessed using SNMP
- **Management Information Base (MIB)** - a collection of definitions, which define the properties of the managed object or the device

SNMP contd...

- **Basic operations**
  - Retrieving data
  - Altering variables
  - Receiving unsolicited messages
- **SNMP tools**
  - "WhatsUP"
  - CiscoWorks Simple network management solution (SNMS)

Chapter 2 outline

- 2.1 Principles of app layer protocols
  - clients and servers
  - app requirements
- 2.2 Web and HTTP
- 2.3.1 FTP
- 2.3.2 telnet, rlogin, snmp
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 Socket programming with TCP
- 2.7 Socket programming with UDP
- 2.8 Building a Web server
- 2.9 Content distribution
  - Network Web caching
  - Content distribution networks
  - P2P file sharing
Electronic Mail

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

User Agent
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- outgoing, incoming messages stored on server

Electronic Mail: mail servers

Mail Servers
- mailbox contains incoming messages 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
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII
Scenario: Alice sends message to Bob
1) Alice uses UA to compose message and "to" bob@someschool.edu
2) Alice's UA sends message to her mail server; message placed in message queue
3) Client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message

Sample SMTP interaction
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection

Try SMTP interaction for yourself:
- telnet servername 25
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands
- above lets you send email without using email client (reader)
**SMTP: final words**
- SMTP uses persistent connections
- SMTP requires 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:
    - different from SMTP command!
  - body
    - the "message", ASCII characters only

**Message format: multimedia extensions**
- MIME: multimedia mail extension, RFC 2045, 2056
  - additional lines in msg header declare MIME content type
  - MIME version
    - method used to encode data
      - multimedia data type, subtype, parameter declaration
    - encoded data
      - 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

Content-Type: type/subtype; parameters

Text
- example subtypes: plain, html

Video
- example subtypes: mpeg, quicktime

Image
- example subtypes: jpeg, gif

Audio
- example subtypes: basic (8-bit mu-law encoded), 32kadpcm (32 kbps coding)

Application
- other data that must be processed by reader before "viewable"
- example subtypes: msword, octet-stream

Multipart Type

From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.

MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=StartOfNextPart

--StartOfNextPart
Dear Bob, Please find a picture of a crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
base64 encoded data ........
.........................
......base64 encoded data
--StartOfNextPart
Do you want the recipe?

Mail access protocols

SMTP: delivery/storage to receiver's server
Mail access protocol: retrieval from server
- POP: Post Office Protocol (RFC 1939)
  - authorization (agent <--> server) and download
- IMAP: Internet Mail Access Protocol (RFC 1730)
  - more features (more complex)
  - manipulation of stored msgs on server
- HTTP: Hotmail, Yahoo! Mail, etc.
POP3 protocol

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

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

S: +OK POP3 server ready
C: user 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 "download and delete" mode.
- Bob cannot re-read e-mail if he changes client
- "Download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

IMAP
- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name