Network Programming using sockets

TCP/IP layers

<table>
<thead>
<tr>
<th>Layers</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Messages (UDP) or Streams (TCP)</td>
</tr>
<tr>
<td>Transport</td>
<td>UDP or TCP packets</td>
</tr>
<tr>
<td>Internet</td>
<td>IP datagrams</td>
</tr>
<tr>
<td>Network interface</td>
<td>Network-specific frames</td>
</tr>
<tr>
<td>Underlying network</td>
<td></td>
</tr>
</tbody>
</table>
The programmer's conceptual view of a TCP/IP Internet

A Programmer’s View of the Internet

1. Hosts are mapped to a set of 32-bit IP addresses.
   - 128.2.203.179

2. The set of IP addresses is mapped to a set of identifiers called Internet domain names.
   - 128.2.203.179 is mapped to www.cs.cmu.edu

3. A process on one Internet host can communicate with a process on another Internet host over a connection.
IP Addresses

32-bit IP addresses are stored in an *IP address struct*

- IP addresses are always stored in memory in network byte order (big-endian byte order)
- True in general for any integer transferred in a packet header from one machine to another.
  - E.g., the port number used to identify an Internet connection.

```c
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

Handy network byte-order conversion functions:
- htonl: convert uint32_t from host to network byte order.
- htons: convert uint16_t from host to network byte order.
- ntohl: convert uint32_t from network to host byte order.
- ntohs: convert uint16_t from network to host byte order.

Dotted Decimal Notation

By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period

- IP address 0x8002C2F2 = 192.168.243.226

Functions for converting between binary IP addresses and dotted decimal strings:

- inet_aton: converts a dotted decimal string to an IP address in network byte order.
- inet_ntoa: converts an IP address in network byte order to its corresponding dotted decimal string.
- “n” denotes network representation. “a” denotes application representation.
IP Address Structure

IP (V4) Address space divided into classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Net ID</th>
<th>Host ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A</td>
<td>0</td>
<td>8-31</td>
</tr>
<tr>
<td>Class B</td>
<td>1-3</td>
<td>8-31</td>
</tr>
<tr>
<td>Class C</td>
<td>1-3</td>
<td>8-31</td>
</tr>
<tr>
<td>Class D</td>
<td>1-127</td>
<td></td>
</tr>
<tr>
<td>Class E</td>
<td>1-255</td>
<td>Reserved for experiments</td>
</tr>
</tbody>
</table>

Network ID Written in form w.x.y.z/n
- \( n \) = number of bits in host address
- E.g., GMU written as 129.174.0.0/16
  - Class B address

Unrouted (private) IP addresses:
- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

Internet Domain Names

```
unnamed root
  .net .edu .gov .com
  mit gmu berkeley amazon
  cs ece www
  markov 129.174.88.235
  tigger 129.174.88.167
```

First-level domain names

Second-level domain names

Third-level domain names
Domain Naming System (DNS)

The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called DNS.

- Conceptually, programmers can view the DNS database as a collection of millions of host entry structures:

```c
/* DNS host entry structure */
struct hostent {
    char    *h_name;       /* official domain name of host */
    char    **h_aliases;   /* null-terminated array of domain names */
    int     h_addrtype;    /* host address type (AF_INET) */
    int     h_length;      /* length of an address, in bytes */
    char    **h_addr_list; /* null-terminated array of in_addr structs */
};
```

Functions for retrieving host entries from DNS:

- `gethostbyname`: query key is a DNS domain name.
- `gethostbyaddr`: query key is an IP address.

Properties of DNS Host Entries

Each host entry is an equivalence class of domain names and IP addresses.

Each host has a locally defined domain name `localhost` which always maps to the loopback address `127.0.0.1`

Different kinds of mappings are possible:

- Simple case: 1-1 mapping between domain name and IP addr:
  - `kittyhawk.cmcl.cs.cmu.edu` maps to `128.2.194.242`
- Multiple domain names mapped to the same IP address:
  - `eecs.mit.edu` and `cs.mit.edu` both map to `18.62.1.6`
- Multiple domain names mapped to multiple IP addresses:
  - `aol.com` and `www.aol.com` map to multiple IP addr.
- Some valid domain names don't map to any IP address:
  - for example: `cmcl.cs.cmu.edu`
A Program That Queries DNS

```c
int main(int argc, char **argv) { /* argv[1] is a domain name */
    char **pp;                    /* or dotted decimal IP addr */
    struct in_addr addr;         /* or bock of IP address */
    struct hostent *hostp;

    if (inet_aton(argv[1], &addr) != 0)
        hostp = Gethostbyaddr((const char *)&addr, sizeof(addr),
                               AF_INET);
    else
        hostp = Gethostbyname(argv[1]);
    printf("official hostname: %s\n", hostp->h_name);
    for (pp = hostp->h_aliases; *pp != NULL; pp++)
        printf("alias: %s\n", *pp);
    for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
        addr.s_addr = ((struct in_addr *)*pp)->s_addr;
        printf("address: %s\n", inet_ntoa(addr));
    }
}
```

Querying DNS from the Command Line

Domain Information Groper (dig) provides a scriptable command line interface to DNS.

```
linux> dig +short kittyhawk.cmcl.cs.cmu.edu
128.2.194.242

linux> dig +short -x 128.2.194.242
KITTYHAWK.CMCL.CS.CMU.EDU.

linux> dig +short aol.com
205.188.145.215
205.188.160.121
64.12.149.24
64.12.187.25

linux> dig +short -x 64.12.187.25
aol-v5.websys.aol.com.
```
Internet Connections

Clients and servers communicate by sending streams of bytes over connections:
- Point-to-point, full-duplex (2-way communication), and reliable.

A socket is an endpoint of a connection
- Socket address is an IPAddress:port pair

A port is a 16-bit integer that identifies a process:
- Ephemeral port: Assigned automatically on client when client makes a connection request
- Well-known port: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

A connection is uniquely identified by the socket addresses of its endpoints (socket pair)
- (cliaddr:cliport, servaddr:servport)

Putting it all Together: Anatomy of an Internet Connection

Client socket address
128.2.194.242:51213

Connection socket pair
(128.2.194.242:51213, 208.216.181.15:80)

Server socket address
208.216.181.15:80

Client host address
128.2.194.242

Server host address
208.216.181.15
Clients

Examples of client programs
- Web browsers, ftp, telnet, ssh

How does a client find the server?
- The IP address in the server socket address identifies the host (more precisely, an adapter on the host)
- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
- Examples of well known ports
  - Port 7: Echo server
  - Port 23: Telnet server
  - Port 25: Mail server
  - Port 80: Web server

Using Ports to Identify Services
Servers

Servers are long-running processes (daemons).
- Created at boot-time (typically) by the init process (process 1)
- Run continuously until the machine is turned off.

Each server waits for requests to arrive on a well-known port associated with a particular service.
- Port 7: echo server
- Port 23: telnet server
- Port 25: mail server
- Port 80: HTTP server

A machine that runs a server process is also often referred to as a “server.”

Server Examples

Web server (port 80)
- Resource: files/compute cycles (CGI programs)
- Service: retrieves files and runs CGI programs on behalf of the client

FTP server (20, 21)
- Resource: files
- Service: stores and retrieve files

Telnet server (23)
- Resource: terminal
- Service: proxies a terminal on the server machine

Mail server (25)
- Resource: email “spool” file
- Service: stores mail messages in spool file

See /etc/services for a comprehensive list of the services available on a Linux machine.
Sockets Interface

Created in the early 80’s as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

Provides a user-level interface to the network.

Underlying basis for all Internet applications.

Based on client/server programming model.

Sockets

What is a socket?

- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
  - All Unix I/O devices, including networks, are modeled as files.

Clients and servers communicate with each other by reading from and writing to socket descriptors.

The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.
Socket programming

**Goal:** learn how to build client/server application that communicate using sockets

**Socket API**
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

- **socket**
  - a host-local, application-created/owned, OS-controlled interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process

Sockets and ports

Internet address = 138.37.94.248

Internet address = 138.37.88.249
**Berkeley Sockets (1)**

Socket primitives for TCP/IP.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication endpoint</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Receive</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

---

**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 client-local 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 client
  - allows server to talk with multiple clients

**application viewpoint**

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server
Socket programming with TCP

Example client-server app:
- Client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)
- Server reads line from socket
- Server converts line to uppercase, sends back to client
- Client reads, prints modified line from socket (inFromServer stream)

Input stream: sequence of bytes into process
Output stream: sequence of bytes out of process

Client/server socket interaction: TCP

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

Client
- Create socket, connect to hostid, port=x:
  \[\text{clientSocket = Socket()}\]
- Send request using clientSocket
- Read reply from clientSocket
- Close clientSocket
Berkeley Sockets (2)

Connection-oriented communication pattern using sockets.

Sockets used for streams

Requesting a connection

s = socket(AF_INET, SOCK_STREAM, 0)
connect(s, ServerAddress)
write(s, "message", length)

Listening and accepting a connection

s = socket(AF_INET, SOCK_STREAM, 0)
bind(s, ServerAddress)
listen(s, 5)
sNew = accept(s, ClientAddress)
n = read(sNew, buffer, amount)

ServerAddress and ClientAddress are socket addresses
Example: Java client (TCP)

```java
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        BufferedReader inFromServer =
            new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + '
');
        modifiedSentence = inFromServer.readLine();
        System.out.println("FROM SERVER: " + modifiedSentence);
        clientSocket.close();
    }
}
```

Example: Java client (TCP), cont.

```java
BufferedReader inFromServer =
    new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + "\n");
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();
```
**Example: Java server (TCP)**

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

class TCPServer {

    public static void main(String argv[]) throws Exception {
        String clientSentence;  
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {
            Socket connectionSocket = welcomeSocket.accept();

            BufferedReader inFromClient = 
                new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));

            DataOutputStream outToClient = 
                new DataOutputStream(connectionSocket.getOutputStream());

            clientSentence = inFromClient.readLine();

            capitalizedSentence = clientSentence.toUpperCase() + '
';

            outToClient.writeBytes(capitalizedSentence);

            BufferedReader inFromClient = 
                new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));

            DataOutputStream outToClient = 
                new DataOutputStream(connectionSocket.getOutputStream());

            clientSentence = inFromClient.readLine();

            capitalizedSentence = clientSentence.toUpperCase() + '
';

            outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

---

**Example: Java server (TCP), cont**

```java
Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Create output stream, attached to socket

Read in line from socket

Write out line to socket

End of while loop, loop back and wait for another client connection
```
Socket programming with UDP

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

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

Client/server socket interaction: UDP

Server (running on hostid)
- create socket, port=x, for incoming request:
  serverSocket = DatagramSocket()
- read request from serverSocket
- write reply to serverSocket specifying client host address, port number

Client
- create socket, clientSocket = DatagramSocket()
- Create, address (hostid, port=x), send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket
**Sockets used for datagrams**

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
    • bind(s, ClientAddress)
    • sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
    • bind(s, ServerAddress)
    • amount = recvfrom(s, buffer, from)
```

*ServerAddress* and *ClientAddress* are socket addresses

---

**Example: Java client (UDP)**

```java
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();
    }
}
```
**Example: Java client (UDP), cont.**

Create datagram with data-to-send length, IP addr, port

Send datagram to server

Receive datagram

---

**Example: Java server (UDP)**

```java
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);
        }
    }
}
```
Example: Java server (UDP), cont

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