5: Socket Programming

Last Modified:
2/10/2003 2:38:37 PM

Socket programming

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

Socket API
- introduced in BSD 4.1 UNIX, 1981
- Sockets are explicitly created, used, released by applications
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

Sockets

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

Languages and Platforms

Socket API is available for many languages on many platforms:
- C, Java, Perl, Python,...
- *nix, Windows,...

Socket Programs written in any language and running on any platform can communicate with each other!
Writing communicating programs in different languages is a good exercise

Socket Programming is Easy

- Create socket much like you open a file
- Once open, you can read from it and write to it
- Operating System hides most of the details

Decisions

- Before you go to write socket code, decide
  - Do you want a TCP-style reliable, full duplex, connection oriented channel? Or do you want a UDP-style, unreliable, message oriented channel?
  - Will the code you are writing be the client or the server?
    - Client: you assume that there is a process already running on another machines that you need to connect to.
    - Server: you will just start up and wait to be contacted
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 created socket, client TCP establishes connection to server TCP
When contacted by client, server TCP creates new socket for server process to communicate with client
- Frees up incoming port
- Allows server to talk with multiple clients

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

Pseudo code TCP client

Create socket, connectSocket
Do an active connect specifying the IP address and port number of server
Read and Write Data Into connectSocket to Communicate with server
Close connectSocket

Pseudo code TCP server

Create socket (doorbellSocket)
Bind socket to a specific port where clients can contact you
Register with the kernel your willingness to listen that on socket for client to contact you
Loop
- Accept new connection (connectSocket)
- Read and Write Data Into connectSocket to Communicate with client
- Close connectSocket
End Loop
Close doorbellSocket

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 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 inFromServer = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
            String clientSentence = inFromServer.readLine();
            String capitalizedSentence = clientSentence.toUpperCase();
            System.out.println("FROM SERVER: " + capitalizedSentence);
            connectionSocket.close();
        }
    }
}
```
Example: Java server (TCP), cont

Example: C client (TCP)

Example: C server (TCP)

Example: C server (TCP), cont.
**TCP Server vs Client**
- Server waits to accept connection on well-known port
- Client initiates contact with the server
- Accept call returns a new socket for this client connection, freeing welcoming socket for other incoming connections
- Read and write only (addresses implied by the connection)

**Concurrent TCP Servers**
- What good is the doorbell socket? Can’t accept new connections until call accept again anyway?
- Benefit comes in ability to hand off processing to another process
  - Parent process creates the “door bell” or “welcome” socket on well-known port and waits for clients to request connection
  - When a client does connect, fork off a child process to handle that connection so that parent process can return to waiting for connections as soon as possible
- Multithreaded server: same idea, just spawn off another thread rather than a full process
  - Threadpools?

**Pseudo code concurrent TCP server**

```
Create socket doorbellSocket
Bind
Listen
Loop
  Accept the connection, connectSocket
  Fork
  If I am the child
    Read/Write connectSocket
    Close connectSocket
    exit
EndLoop
Close doorbellSocket
```

**Backlog**
- Many implementations do allow a small fixed number (~5) of unaccepted connections to be pending, commonly called the backlog
- This helps avoid missing connections while process not sitting in the accept call

**Socket programming with UDP**
- UDP: very different mindset than TCP
  - no connection just independent messages sent
  - no handshaking
  - sender explicitly attaches IP address and port of destination
  - server must extract IP address, port of sender from received datagram to know who to respond to
- UDP: transmitted data may be received out of order, or lost

**Pseudo code UDP server**

```
Create socket
Bind socket to a specific port where clients can contact you
Loop
  (Receive UDP Message from client x)*
  (Send UDP Reply to client x)*
Close Socket
```
**Pseudo code UDP client**

Create socket

Loop
  (Send Message To Well-known port of server)+
  (Receive Message From Server)

Close Socket

---

**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();
        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)**

```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);
            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);
        }
    }
}
```

---

**Client/server socket interaction: UDP**

**Server** (running on hostid)
- Create, address (hostid, port=x)
- send datagram request using clientSocket

**Client**
- Create socket, clientSocket = new DatagramSocket()
- Create, address (hostid, port=x)
- send datagram request using clientSocket
Example: C client (UDP)

```c
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char **argv) {
    int socket;
    char sentence[MAX_LINE];
    char modifiedSentence[MAX_LINE];
    struct hostent *hp;
    struct sockaddr_in cliAddr, remoteServAddr;

    socket = socket(AF_INET, SOCK_DGRAM, 0);
    /* translate host name into peer's IP address */
    hp = gethostbyname("hostname");
    cliAddr.sin_family = AF_INET;
    cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    cliAddr.sin_port = htons(0);
    bind(socket, (struct sockaddr *) &cliAddr, sizeof(cliAddr));

    remoteServAddr.sin_family = h->h_addrtype;
    memcpy((char *) &remoteServAddr.sin_addr.s_addr, h->h_addr_list[0], h->h_length);
    remoteServAddr.sin_port = htons(9876);

    sendto(socket, sentence, MAX_LINE, 0, (struct sockaddr *) &remoteServAddr, sizeof(remoteServAddr));
    recvfrom(socket, modifiedSentence, MAX_LINE, 0, (struct sockaddr *) &remoteServAddr, &remoteServLen);
    fprintf(stderr, "FROM SERVER: %s \n", modifiedSentence);
    close(socket);
}
```

Example: C server (UDP), cont.

```c
register to receive datagrams on this socket

bind(socket, (struct sockaddr *) &svrAddr, sizeof(svrAddr));
for (;;) {
    recvfrom(socket, clientSentence, MAX_LINE, 0, (struct sockaddr *) &cliAddr, sizeof(cliAddr));
    // would have to write the capitalize procedure
    capitalize(clientSentence);
    sendto(socket, clientSentence, MAX_LINE, 0, (struct sockaddr *) &svrAddr, sizeof(svrAddr));
}
```

UDP Server vs Client

- Server has a well-known port number
- Client initiates contact with the server
- Less difference between server and client code than in TCP
  - Both client and server bind to a UDP socket
  - Not accept for server and connect for client
- Client send to the well-known server port; server extracts the client’s address from the datagram it receives

TCP vs UDP

- TCP can use read/write (or recv/send) and source and destination are implied by the connection; UDP must specify destination for each datagram
- Sendto, recvfrom include address of other party
- TCP server and client code look quite different; UDP server and client code vary mostly in who sends first
Java vs C

- Java hides more of the details
  - `new ServerSocket` of Java = socket, bind and listen of C
  - `new Socket` hides the `getByName` (or `gethostbyname`) of C. Unable to hide this in the UDP case though
  - Socket API first in C for BSD; more options and choices exposed by the interface than in Java?

Note

- Examples were simple code snippets
- To fit on a slide, I omitted important things like:
  - Testing each connect, sendto and recvfrom for errors
  - In UDP case, handling the case of packet loss
  - The behavior of many of these functions can be "customized" with various socket options
  - In C, use `setsockopt/getsockopt`
  - In Java, use `setOption/getOption`

Socket Programming in the Real World

- Download some open source implementations of network applications
  - Web browsers (Mosaic, Jazilla)
  - DNS Servers and resolvers (BIND)
  - Email clients/servers (sendmail, qmail, pine)
  - telnet
- Can you find the socket code? The protocol processing? What percentage of the code is it? What does the rest of the code do?

On to the transport layer...

- Important to remember that we build transport services to support applications
- Transport services are a means to an end

Outtakes

Real Internet Traffic Analysis

Credit: CAIDA (1999)
### Transport service requirements of common apps

<table>
<thead>
<tr>
<th>Application</th>
<th>Data loss</th>
<th>Bandwidth</th>
<th>Time Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer</td>
<td>no loss</td>
<td>elastic</td>
<td>no?</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>loss-tolerant</td>
<td>elastic</td>
<td>yes, 100's msec</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5Kb-1Mb, video: 10Kb-5Mb</td>
<td>yes, 100's msec</td>
</tr>
<tr>
<td>stored audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few Kbps up</td>
<td>yes, 100's msec</td>
</tr>
<tr>
<td>news</td>
<td>No loss?</td>
<td>elastic</td>
<td>no</td>
</tr>
</tbody>
</table>

---

### Internet apps: their protocols and transport protocols

<table>
<thead>
<tr>
<th>Application</th>
<th>Application layer protocol</th>
<th>Underlying transport protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-mail</td>
<td>smtp (RFC 821)</td>
<td>TCP</td>
</tr>
<tr>
<td>remote terminal access</td>
<td>telnet (RFC 854)</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>http (RFC 2068)</td>
<td>TCP</td>
</tr>
<tr>
<td>file transfer</td>
<td>ftp (RFC 959)</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>RTP, proprietary (e.g., RealNetworks)</td>
<td>UDP</td>
</tr>
<tr>
<td>remote file server</td>
<td>NFS</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>proprietary (e.g., Vocaltec)</td>
<td>typically UDP</td>
</tr>
<tr>
<td>DNS</td>
<td>DNS</td>
<td>typically UDP, TCP</td>
</tr>
</tbody>
</table>