CSE 3112
Lab 01
Tutorial: All about Sockets

In this Lab you will be introduced to socket programming at a very elementary level. Specifically, we will focus on TCP socket connections which are a fundamental part of socket programming since they provide a connection oriented service with both flow and congestion control. TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other.

Introduction

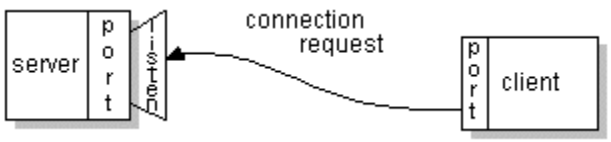
In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

What Is a Socket?

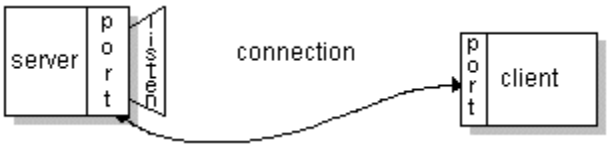
Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and

port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

*The **java.net** package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The **Socket** class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the **java.net.Socket** class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.*

*Additionally, **java.net** includes the **ServerSocket** class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the Socket and ServerSocket classes.*

Overview of IP4 addresses:

IP4 addresses are 32 bits long. They are expressed commonly in what is known as dotted decimal notation. Each of the four bytes which makes up the 32 address are expressed as an integer value (0 – 255) and separated by a dot. For example, 138.23.44.2 is an example of an IP4 address in dotted decimal notation.

Ports:

Sockets are UNIQUELY identified by Internet address, end-to-end protocol, and port number. That is why when a socket is first created it is vital to match it with a valid IP address and a port number.

So when a client first tries to contact a server, which port number should the client specify? For many common services, standard port numbers are defined.

Port	*Service Name, Alias*	*Description*
1	`tcpmux`	TCP port service multiplexer
7	`echo`	Echo server
9	`discard`	Like `/dev/null`
13	`daytime`	System's date/time
20	`ftp-data`	FTP data port
21	`ftp`	Main FTP connection
23	`telnet`	Telnet connection
25	`smtp, mail`	UNIX mail
37	`time, timeserver`	Time server
42	`nameserver`	Name resolution (DNS)
70	`gopher`	Text/menu information
79	`finger`	Current users

Ports 0 – 1023, are reserved and servers or clients that you create will not be able to bind to these ports unless you have root privilege.

Ports 1024 – 65535 are available for use by your programs, but beware other network applications maybe running and using these port numbers as well so do not make assumptions about the availability of specific port numbers.

Reading from and Writing to a Socket

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the Socket class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client, **EchoClient**, that connects to an echo server. The echo server receives data from its client and echoes it back. The example **EchoServer** implements an echo server. (Alternatively, the client can connect to any host that supports the Echo Protocol.)

The **EchoClient** example creates a socket, thereby getting a connection to the echo server. It reads input from the user on the standard input stream, and then forwards that text to the echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server.

Note that the **EchoClient** example both writes to and reads from its socket, thereby sending data to and receiving data from the echo server.

Let's walk through the program and investigate the interesting parts. The following statements in the try-with-resources statement in the **EchoClient** example are critical. These lines establish the socket connection between the client and the server and open a **PrintWriter** and a **BufferedReader** on the socket:

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
)
```

The first statement in the try-with resources statement creates a new **Socket** object and names it **echoSocket**. The Socket constructor used here requires the name of the computer and the port number to which you want to connect. The example program uses the first command-line argument as the name of the computer (the host name) and the second command line argument as the port number. When you run this program on your computer, make sure that the host name you use is the fully qualified IP name of the computer to which you want to connect. For example, if your echo server is running on the “172.16.3.88” and it is listening on port number “2200”, first run the following command from the computer echoserver.example.com if you want to use the **EchoServer** example as your echo server:

```
java EchoServer 2200
```

Afterward, run the EchoClient example with the following command:

```
java EchoClient 172.16.3.88 2200
```

The second statement in the try-with resources statement gets the socket's output stream and opens a **PrintWriter** on it. Similarly, the third statement gets the socket's input stream and opens a **BufferedReader** on it. The example uses readers and writers so that it can write Unicode characters over the socket.

To send data through the socket to the server, the **EchoClient** example needs to write to the **PrintWriter**. To get the server's response, **EchoClient** reads from the **BufferedReader** object stdIn, which is created in the fourth statement in the try-with resources statement. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the while loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the PrintWriter connected to the socket:

```
String userinput;
while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

The last statement in the while loop reads a line of information from the **BufferedReader** connected to the socket. The **readLine** method waits until the server echoes the information back to **EchoClient**. When readline returns, EchoClient prints the information to the standard output.

The while loop continues until the user types an end-of-input character. That is, the EchoClient example reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of-input.

Writing the Server Side of a Socket

The **EchoServer** example runs on a specific computer and has a socket that is bound to a specific port number. The server waits, listening to the socket for a client to make a connection request. The server echoes the input back through the socket to the client.

Let's walk through the program and investigate the interesting parts. The following statements in the try-with-resources statement in the **EchoServer** example are critical.

```
int portNumber = Integer.parseInt(args[0]);

try (
    ServerSocket serverSocket =
        new ServerSocket(Integer.parseInt(args[0]));
    Socket clientSocket = serverSocket.accept();
    PrintWriter out =
        new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(clientSocket.getInputStream()));

)
```

The example program uses the first command-line argument as the port number. When you run this program on your computer, make sure that the client and server works on the same port number. First run the following command from the computer if you want to use the **EchoServer** example as your echo server:

```
java EchoServer 2200
```

The second statement in the try-with resources statement gets the socket's output stream and opens a **PrintWriter** on it. Similarly, the third statement gets the socket's input stream and opens a **BufferedReader** on it. The example uses readers and writers so that it can write Unicode characters over the socket.

To send data through the socket to the server, the **EchoClient** example needs to write to the **PrintWriter**. To get the server's response, **EchoClient** reads from the **BufferedReader** object stdIn, which is created in the fourth statement in the try-with resources statement. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the while loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the PrintWriter connected to the socket:

```
String inputLine;
 while ((inputLine = in.readLine()) != null) {
 out.println(inputLine);}
```

Exception

You have to catch the exception `IOException,` and print appropriate messages.

Reference

- Java: The Complete Reference, Ninth Edition 9th Edition, **by Herbert Schildt**
- https://docs.oracle.com/javase/tutorial/
- http://alumni.cs.ucr.edu/~ecegelal/TAw/