Al-Azhar UNIVERSITY

Faculty of Engineering

Computers and Systems Engineering Department

- -

EXPERIMENT Socket Programming

OBJECTIVES

(A) To understand Client-Server communication via sockets
(B) To gain exposure to the basic operations of a Web Server and Client
(C) To explore basic structures of HTTP messages

MATERIALS/EQUIPMENT NEEDED

1. Java and any suitable IDE

INTRODUCTION

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols −

- **TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects −

- **Socket Programming** − This is the most widely used concept in Networking and it has been explained in very detail.
- **URL Processing** − This would be covered separately. Click here to learn about URL Processing in Java language.

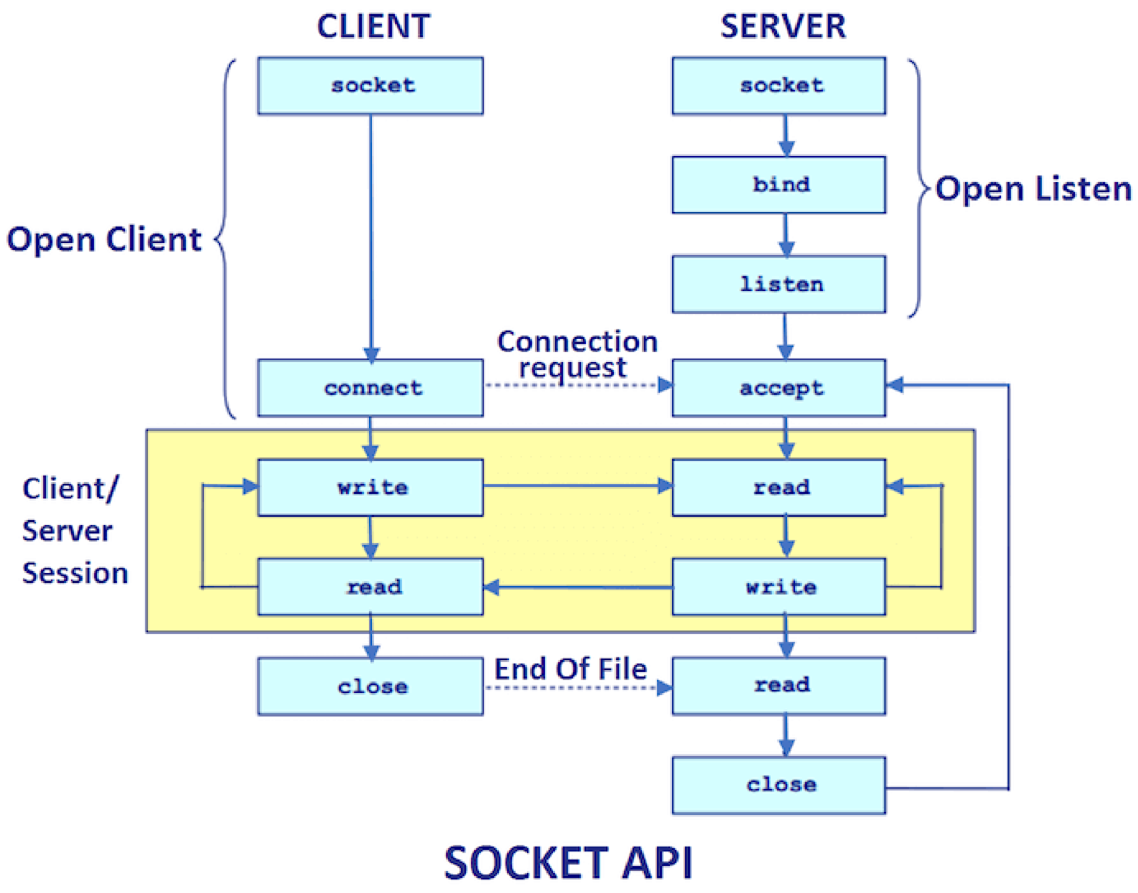
Socket Programming - Life Cycle

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets –



- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.

- The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

Socket Programming Example 1

Client-Side Programming

In the case of client-side programming, the client will first wait for the server to start. Once the server is up and running, it will send the requests to the server. After that, the client will wait for the response from the server. So, this is the whole logic of client and server communication. Now let's understand the client side and server-side programming in detail.

In order to initiate a client's request, you need to follow the below-mentioned steps:

1. Establish a Connection

The very first step is to establish a socket connection. A socket connection implies that the two machines have information about each other's network location (IP Address) and TCP port.

You can create a Socket with the help of a below statement:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

- Here, the first argument represents the **IP address of Server**.
- The second argument represents the **TCP Port**. (It is a number that represents which application should run on a server.)

2. Communication

In order to communicate over a socket connection, streams are used for both input and output the data. After establishing a connection and sending the requests, you need to close the connection.

3. Closing the connection

The socket connection is closed explicitly once the message to the server is sent.

Now let's see how to write a Java program to implement socket connection at client side.

```
// A Java program for a Client
import java.net. * ;
```

```java
import java.io. * ;

public class Client {
  // initialize socket and input output streams
  private Socket socket = null;
  private DataInputStream input = null;
  private DataOutputStream out = null;

  // constructor to put ip address and port
  public Client(String address, int port) {
    // establish a connection
    try {
      socket = new Socket(address, port);
      System.out.println("Connected");

      // takes input from terminal
      input = new DataInputStream(System. in );

      // sends output to the socket
      out = new DataOutputStream(socket.getOutputStream());
    } catch(UnknownHostException u) {
      System.out.println(u);
    } catch(IOException i) {
      System.out.println(i);
    }

    // string to read message from input
    String line = "";

    // keep reading until "Over" is input
    while (!line.equals("Over")) {
      try {
        line = input.readLine();
        out.writeUTF(line);
      } catch(IOException i) {
        System.out.println(i);
      }
    }

    // close the connection
    try {
      input.close();
      out.close();
      socket.close();
    } catch(IOException i) {
      System.out.println(i);
    }
  }

  public static void main(String args[]) {
    Client client = new Client("127.0.0.1", 5000);
  }
}
```

Now, let's implement server-side programming and then arrive at the output.

Server-Side Programming

Basically, the server will instantiate its object and wait for the client request. Once the client sends the request, the server will communicate back with the response.

In order to code the server-side application, you need two sockets and they are as follows:

- A **ServerSocket** which waits for the client requests (when a client makes a new Socket())
- A plain old **socket** for communication with the client.

After this, you need to communicate with the client with the response.

Communication

getOutputStream() method is used to send the output through the socket.

Close the Connection

It is important to close the connection by closing the socket as well as input/output streams once everything is done.

Now let's see how to write a Java program to implement socket connection at server side.

```java
// A Java program for a Server
import java.net.*;
import java.io.*;

public class Server
{
        //initialize socket and input stream
        private Socket          socket = null;
        private ServerSocket server = null;
        private DataInputStream in      = null;

        // constructor with port
        public Server(int port)
        {
                // starts server and waits for a connection
                try
                {
                        server = new ServerSocket(port);
                        System.out.println("Server started");
                        System.out.println("Waiting for a client ...");

                        socket = server.accept();
                        System.out.println("Client accepted");

                        // takes input from the client socket
                        in = new DataInputStream(
                                new 
BufferedInputStream(socket.getInputStream()));

                        String line = "";

                        // reads message from client until "Over" is sent
                        while (!line.equals("Over"))
                        {
                                try
                                {
                                        line = in.readUTF();
                                        System.out.println(line);

                                }
                                catch(IOException i)
                                {
                                        System.out.println(i);
                                }
                        }
                        System.out.println("Closing connection");

                        // close connection
                        socket.close();
                        in.close();
                }
                catch(IOException i)
                {
                        System.out.println(i);
                }
        }

        public static void main(String args[])
        {
                Server server = new Server(5000);
        }
}
```

After configuring both client and server end, you can execute the server-side program first. After that, you need to run client-side program and send the request. As soon as the request is sent from the client end, server will respond back.

Socket Programming Example 2(Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

File: MyServer.java

```
import java.net. * ;
import java.io. * ;
class MyServer {
  public static void main(String args[]) throws Exception {
    ServerSocket ss = new ServerSocket(3333);
    Socket s = ss.accept();
    DataInputStream din = new DataInputStream(s.getInputStream());
    DataOutputStream dout = new DataOutputStream(s.getOutputStream());
    BufferedReader br = new BufferedReader(new
InputStreamReader(System. in ));

    String str = "",
    str2 = "";
    while (!str.equals("stop")) {
      str = din.readUTF();
      System.out.println("client says: " + str);
      str2 = br.readLine();
      dout.writeUTF(str2);
      dout.flush();
    }
    din.close();
    s.close();
    ss.close();
  }
}
```

File: MyClient.java

```
import java.net. * ;
import java.io. * ;
class MyClient {
  public static void main(String args[]) throws Exception {
    Socket s = new Socket("localhost", 3333);
    DataInputStream din = new DataInputStream(s.getInputStream());
    DataOutputStream dout = new DataOutputStream(s.getOutputStream());
    BufferedReader br = new BufferedReader(new
InputStreamReader(System. in ));

    String str = "",
    str2 = "";
    while (!str.equals("stop")) {
      str = br.readLine();
      dout.writeUTF(str);
      dout.flush();
      str2 = din.readUTF();
      System.out.println("Server says: " + str2);
    }

    dout.close();
    s.close();
  }
}
```

POST-LAB

- Make tow clients send messages to each other?
- Make a GUI for the Server and the Client?