St. Francis Institute of Technology
<u>Department of Computer Engineering</u>

Academic Year: 2021-2022 **Semester: VIII**
Subject: Distributed Computing **Class / Branch / Division: BE/CMPN/A**
Name: Rebecca Dias Roll Number:18

Experiment No: 01

Aim: Client Server Implementation Using Java Socket

Pre-requisites: Networking

Theory:
The endpoint in an interprocess communication is called a socket, or a network socket for disambiguation. Since most communication between computers is based on the Internet Protocol, an almost equivalent term is Internet socket. The data transmission between two sockets is organized by communications protocols, usually implemented in the operating system of the participating computers. Application programs write to and read from these sockets. Therefore, network programming is essentially socket programming.

Point-to-Point Communication
In a nutshell, a socket represents a single connection between exactly two pieces of software. More than two pieces of software can communicate in client/server or distributed systems (for example, many Web browsers can simultaneously communicate with a single Web server) but multiple sockets are required to do this. Socket-based software usually runs on two separate computers on the network, but sockets can also be used to communicate locally (interprocess) on a single computer.

Sockets are bidirectional, meaning that either side of the connection is capable of both sending and receiving data. Sometimes the one application that initiates communication is termed the client and the other application the server, but this terminology leads to confusion in non-client/server systems and should generally be avoided.

Interface Types
Socket interfaces can be divided into three categories. Perhaps the most commonly-used type, the stream socket, implements "connection-oriented" semantics. Essentially, a "stream" requires that the two communicating parties first establish a socket connection, after which any data passed through that connection will be guaranteed to arrive in the same order in which it was sent.
Datagram sockets offer "connection-less" semantics. With datagrams, connections are implicit rather than explicit as with streams. Either party simply sends datagrams as needed and waits for the other to respond; messages can be lost in transmission or received out of order, but it is the application's responsibility and not the socket's to deal with these problems. Implementing datagram sockets can give some applications a performance boost and additional flexibility compared to using stream sockets, justifying their use in some situations.
The third type of socket -- the so-called raw socket -- bypasses the library's built-in support for standard protocols like TCP and UDP. Raw sockets are used for custom low-level protocol

development.

Addresses and Ports
Today, sockets are typically used in conjunction with the Internet protocols -- Internet Protocol, Transmission Control Protocol, and User Datagram Protocol (UDP). Libraries implementing sockets for Internet Protocol use TCP for streams, UDP for datagrams, and IP itself for raw sockets.
To communicate over the Internet, IP socket libraries use the IP address to identify specific computers. Many parts of the Internet work with naming services, so that the users and socket programmers can work with computers by name (e.g., "thiscomputer.compnetworking.about.com") instead of by address. Stream and datagram sockets also use IP port numbers to distinguish multiple applications from each other. For example, Web browsers on the Internet know to use port 80 as the default for socket communications with Web servers.

Implementation:

Server:

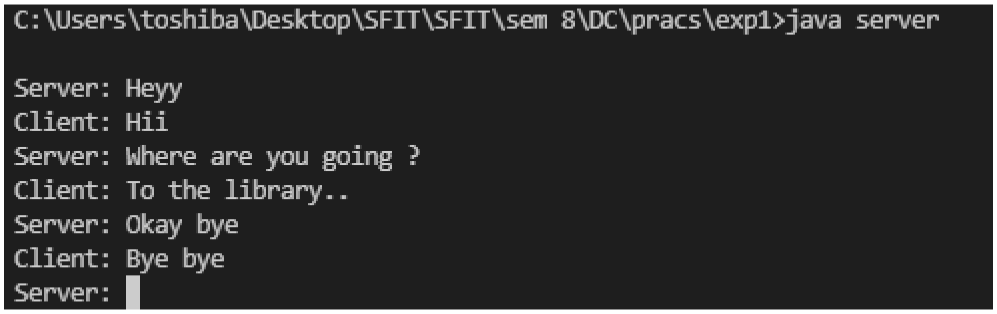
```
import java.io.*;
import java.net.*;

public class server {
    public static void main(String args[]) throws Exception {
        server serv = new server();
        serv.run();
    }

    public void run() throws Exception {
        ServerSocket servsock = new ServerSocket(9999);
        Socket sock = servsock.accept();
        String clside = new String();
        String srside = new String();

        BufferedReader br1 = new BufferedReader(new
InputStreamReader(System.in));
        BufferedReader br = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
        PrintStream ps = new PrintStream(sock.getOutputStream());
        while (true) {
            System.out.println();
            System.out.print("Server: ");
            srside = br1.readLine();
            ps.println(srside);
            clside = br.readLine();

            System.out.print("Client: " + clside);
        }

    }
}
```

Client:

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

public class client {
    public static void main(String args[]) throws Exception {
        client clt = new client();
        clt.run();
    }

    public void run() throws Exception {
        Socket sock = new Socket("localhost", 9999);
        String clside = new String();
        String srside = new String();

        BufferedReader br1 = new BufferedReader(new
InputStreamReader(System.in));

        BufferedReader br = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
        PrintStream ps = new PrintStream(sock.getOutputStream());
        while (true) {
            srside = br.readLine();
            System.out.print("Server: " + srside);
            System.out.println();
            System.out.print("Client: ");
            clside = br1.readLine();
            ps.println(clside);
        }
    }
}
```

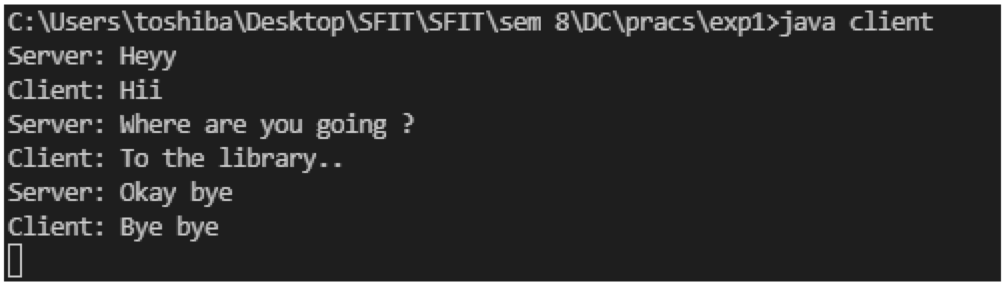
Output:

Server



```
C:\Users\toshiba\Desktop\SFIT\SFIT\sem 8\DC\pracs\exp1>java server

Server: Heyy
Client: Hii
Server: Where are you going ?
Client: To the library..
Server: Okay bye
Client: Bye bye
Server:
```

Client



```
C:\Users\toshiba\Desktop\SFIT\SFIT\sem 8\DC\pracs\exp1>java client
Server: Heyy
Client: Hii
Server: Where are you going ?
Client: To the library..
Server: Okay bye
Client: Bye bye
```

Conclusion:

In the given experiment, we implemented server side and client side communication using java socket programming. getInputStream (), getOutputStream () and void close functions were used to return the InputStream and similarly OutputStream attached with the socket is used for returning and closing the socket respectively. The program was successfully executed and the desired output was obtained.