

Lab Report No: 03

Lab Report Name: Socket Programming Implementation

Name: Md.Rasel

ID: IT-17049

Socket Programming:

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are the only two protocols supported by Java that run on top of IP (Internet Protocol), and application layer protocols are built on top of these two. The basic difference between the protocols is that TCP is connection oriented and UDP is connectionless. Besides these two, there is another commonly used protocol, named ICMP (Internet Control Message Protocol). The common use of ICMP is the *ping* program. ICMP is not supported by Java because it uses raw IP datagrams to relay messages between hosts. Java, however, has a provision to implement them with the help of native code. Socket programmers should at least skim over the conceptual bases of the TCP/IP protocol stack before delving into network programming. The article tries to provide some key information with a focus on building an UDP client/server application in Java.

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.

Time Protocol implementation: 2.2.2

A java program where the following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;

public class GreetingClient {

    public static void main(String [] args) {
        String serverName = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            System.out.println("Connecting to " + serverName + " on port " + port);
            Socket client = new Socket(serverName, port);

            System.out.println("Just connected to " + client.getRemoteSocketAddress());
            OutputStream outToServer = client.getOutputStream();
            DataOutputStream out = new DataOutputStream(outToServer);

            out.writeUTF("Hello from " + client.getLocalSocketAddress());
            InputStream inFromServer = client.getInputStream();
            DataInputStream in = new DataInputStream(inFromServer);

            System.out.println("Server says " + in.readUTF());
            client.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument –

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
    private ServerSocket serverSocket;

    public GreetingServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
    }
}
```

```

serverSocket.setSoTimeout(10000);
}

public void run() {
while(true) {
try {
System.out.println("Waiting for client on port " +
serverSocket.getLocalPort() + "...");
Socket server = serverSocket.accept();

System.out.println("Just connected to " + server.getRemoteSocketAddress());
DataInputStream in = new DataInputStream(server.getInputStream());

System.out.println(in.readUTF());
DataOutputStream out = new DataOutputStream(server.getOutputStream());
out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress()
+ "\nGoodbye!");
server.close();

} catch (SocketTimeoutException s) {
System.out.println("Socket timed out!");
break;
} catch (IOException e) {
e.printStackTrace();
break;
}
}
}

public static void main(String [] args) {
int port = Integer.parseInt(args[0]);
try {
Thread t = new GreetingServer(port);
t.start();
} catch (IOException e) {
e.printStackTrace();
}
}
}

```

Output:

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

Echo Protocol implementation: 2.2.3

Client Program:

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

public class EchoClient {
    public static void main(String[] args) throws IOException {

        Scanner s = new Scanner(System.in);
        String serverHostname;
        System.out.println("Enter an IP value: ");
        serverHostname = s.next();

        //String serverHostname = new String ("127.0.0.1");

        if (args.length > 0)
            serverHostname = args[0];
        System.out.println ("Attempting to connect to host " +
            serverHostname + " on port 10007.");

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket(serverHostname, 10007);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: " + serverHostname);
```

```

System.exit(1);
} catch (IOException e) {
System.err.println("Couldn't get I/O for "
+ "the connection to: " + serverHostname);
System.exit(1);
}

```

```

BufferedReader stdIn = new BufferedReader(
new InputStreamReader(System.in));
String userInput;

```

```

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

```

```

out.close();
in.close();
stdIn.close();
echoSocket.close();
}
}

```

Server Program:

```

public class EchoServer
{
public static void main(String[] args) throws IOException
{
ServerSocket serverSocket = null;

try{
serverSocket = new ServerSocket(10007);
}
catch (IOException e)
{
System.err.println("Could not listen on port: 10007.");
System.exit(1);
}

Socket clientSocket = null;
System.out.println ("Waiting for connection.....");

try {
clientSocket = serverSocket.accept();
}

```

```

catch (IOException e)
{
    System.err.println("Accept failed.");
    System.exit(1);
}

System.out.println ("Connection successful");

PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
true);

BufferedReader in = new BufferedReader(
    new InputStreamReader( clientSocket.getInputStream()));

String cAddress = "";
String inputLine;

cAddress = clientSocket.getInetAddress().toString();

while ((inputLine = in.readLine()) != null)
{
    System.out.println ("Server: " + inputLine + " " + cAddress + " ");
    out.println(inputLine + " " + cAddress);

    if (inputLine.equals("bye"))
        break;
}

out.close();
in.close();
clientSocket.close();
serverSocket.close();
}
}

```

Trivial File Transfer Protocol implementation: 2.3.1

Client Program:

```

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

class FTPC
{

```

```

public static void main(String[] args) throws Exception
{
    String option;
    DataInputStream in=new DataInputStream(System.in);
    Socket s=new Socket("localhost",Integer.parseInt(args[0]));
    System.out.println("MENU");
    System.out.println("1.SEND");
    System.out.println("2.RECEIVE");
    FTPC ftp=new FTPC();
    while(true)
    {
        option=in.readLine();
        if(option.equals("1")){
            System.out.println("SEND Command Received..");
            ftp.sendfile(s);
        }

        else if(option.equals("2")){
            System.out.println("RECEIVE Command Received..");
            ftp.receivefile(s);
        }

    }
}

public void sendfile(Socket s) throws Exception
{
    Socket ssock=s;

    DataInputStream in=new DataInputStream(System.in);

    DataInputStream cin=new DataInputStream(ssock.getInputStream());
    DataOutputStream cout=new DataOutputStream(ssock.getOutputStream());

    cout.writeUTF("RECEIVE");

    String filename=in.readLine();
    System.out.println("Reading File "+filename);
    cout.writeUTF(filename);
    File f=new File(filename);
    FileInputStream fin=new FileInputStream(f);
    int ch;
    do
    {
        ch=fin.read();
        cout.writeUTF(String.valueOf(ch));
    }while(ch!=-1);
    fin.close();
}

```

```

System.out.println("File Sent");
}

public void receivefile(Socket s) throws Exception
{
    Socket ssock=s;
    DataInputStream in=new DataInputStream(System.in);
    DataInputStream cin=new DataInputStream(ssock.getInputStream());
    DataOutputStream cout=new DataOutputStream(ssock.getOutputStream());

    cout.writeUTF("SEND");

    String filename=in.readLine();
    cout.writeUTF(filename);
    System.out.println("Receiving File "+filename);
    File f=new File(filename);
    FileOutputStream fout=new FileOutputStream(f);
    int ch;
    do
    {
        ch=Integer.parseInt(cin.readUTF());
        if(ch!=-1) fout.write(ch);
    }while(ch!=-1);
    System.out.println("Received File...");
    fout.close();
}
}

```

Server Program:

```

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

class FTPS
{
    public static void main(String[] args) throws Exception
    {
        ServerSocket Sock=new ServerSocket(Integer.parseInt(args[0]));
        Socket s=Sock.accept();
        DataInputStream cin=new DataInputStream(s.getInputStream());
        DataOutputStream cout=new DataOutputStream(s.getOutputStream());

        FTPS ftp=new FTPS();
        while(true)
        {

```



```

String option=cin.readUTF();
if(option.equals("SEND")){
System.out.println("SEND Command Received..");
ftp.sendFile(s);
}

else if(option.equals("RECEIVE")){
System.out.println("RECEIVE Command Received..");
ftp.receivefile(s);
}
}

```

```

public void sendfile(Socket s) throws Exception
{
Socket ssock=s;

```

```

DataInputStream cin=new DataInputStream(ssock.getInputStream());
DataOutputStream cout=new DataOutputStream(ssock.getOutputStream());
String filename=cin.readUTF();
System.out.println("Reading File "+filename);
File f=new File(filename);
FileInputStream fin=new FileInputStream(f);
int ch;
do
{
ch=fin.read();
cout.writeUTF(Integer.toString(ch));
}while(ch!=-1);
fin.close();
System.out.println("File Sent");
}

```

```

public void receivefile(Socket s) throws Exception
{
Socket ssock=s;

```

```

DataInputStream cin=new DataInputStream(ssock.getInputStream());
DataOutputStream cout=new DataOutputStream(ssock.getOutputStream());

String filename=cin.readUTF();
System.out.println("Receiving File "+filename);
File f=new File(filename);
FileOutputStream fout=new FileOutputStream(f);
int ch;
while((ch=Integer.parseInt(cin.readUTF()))!=-1)
{

```

```
fout.write(ch);  
}  
System.out.println("Received File...");  
fout.close();  
}  
}
```

Conclusion: The Time Protocol may be implemented over the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). A host connects to a server that supports the Time Protocol on port 37. The server then sends the time as a 32-bit unsigned integer in binary format and in network byte order, representing the number of seconds since 00:00 (midnight) 1 January, 1900 GMT, and closes the connection. Operation over UDP requires the sending of any datagram to the server port, as there is no connection setup for UDP. The **Echo Protocol** is a service in the Internet Protocol Suite defined in RFC 862. It was originally proposed for testing and measurement in IP networks.

A host may connect to a server that supports the Echo Protocol using the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) on the well-known port number 7. The server sends back an identical copy of the data it received. **Trivial File Transfer Protocol (TFTP)** is a simple lockstep File Transfer Protocol which allows a client to get a file from or put a file onto a remote host. One of its primary uses is in the early stages of nodes booting from a local area network. TFTP has been used for this application because it is very simple to implement.

TFTP was first standardized in 1981 and the current specification for the protocol can be found in RFC 1350