Distributed Systems Programming
ICT2107

Lab 1 – Distributed Network Programming (TCP/UDP)

Objectives

In this lab, you will:

- Appreciate the use of TCP/UDP Socket programming abstraction to develop distributed applications.
- Practice on Socket programming

Setting up IDE

Before you start programming, you will need to install an IDE that supports Java programming. In this subject, we will be making us of the Eclipse IDE. Take the following steps:

Step 1: Download Java SDK. If you have not already installed the Java SDK, you can proceed to the following website to download the Java SDK: https://www.oracle.com/technetwork/java/javase/downloads/index.html

If you are running Windows 8 or above, you may choose to install Windows x64.

Step 2: You can download and install the Eclipse IDE from the following site: https://www.eclipse.org/downloads/packages/

Using Eclipse for Java Programming

If you are not familiar with Eclipse, you may want to watch a video that has been uploaded to the xSite LMS on "Using Eclipse for Java Programming". In this video, it is briefly introduced on how we can use Eclipse to organize program files into packages, writing and running your first "Hello World" program.

Exercise 1. Transmission Control Protocol (TCP)

In this section, you will learn the basics of creating a distributed application. This is done by writing distributed processes (namely client and server) to be executed on different machines, and using TCP to provide a channel for communication between the processes.

E1.1 TCP Programming Illustration

Read the TCP Programming illustration slide (TCPIllustration.pptx)

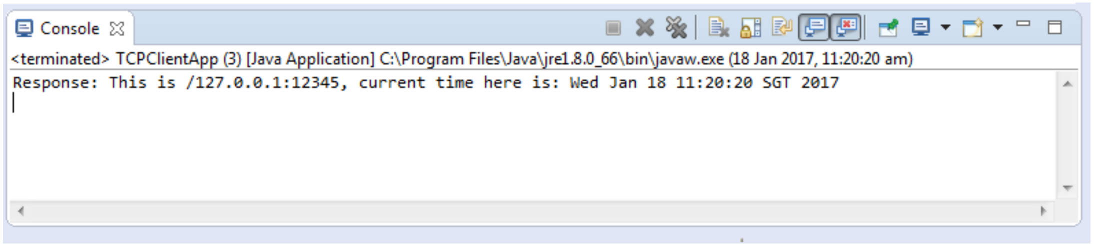
E1.2 TCP Server Application

Create a new Java project named TCPServerApp, followed by a package called tcpserverapp. Within the package, create a class called TCPServerApp.java with the following codes:

```
package tcpserverapp;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;
public class TCPServerApp {
    public static void main(String[] args) {
        //The port, at which, the server is listening for
requests
        int port = 12345;
        try {
            //ServerSocket object is used to listen for
requests
            ServerSocket ss = new ServerSocket(port);
            System.out.println("Server is ready to receive
command!");
            while(true){
                //accepting connection requests
                Socket socket = ss.accept();
                //get the input stream to read data
                InputStream is = socket.getInputStream();
                //Read data as character
                InputStreamReader isr = new
InputStreamReader(is);
                //Read data as lines
                BufferedReader br = new BufferedReader(isr);
                //Read the string command from the user
                String command = br.readLine();
                String response = "";
                if(command.equals("GET TIME")){
                    response = "This is " +
socket.getLocalAddress() +
                            ":" + socket.getLocalPort() +
                            ", current time here is: " + new
Date();
                }else{
                    response = "Invalid command, should use
'GET TIME'";
                }
                //Access to the output stream, to write data
back
                OutputStream os = socket.getOutputStream();
                //Write lines
                PrintWriter pw = new PrintWriter(os);
                pw.println(response);
                pw.flush();
                pw.close();
                socket.close();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

E1.3 TCP Client Application

Create a new Java application named TCPClientApp.
Write the following code to the TCPClientApp.java file inside this application. You are required to fill-in the blanks indicated by the boxes. The output from the console should look like this:

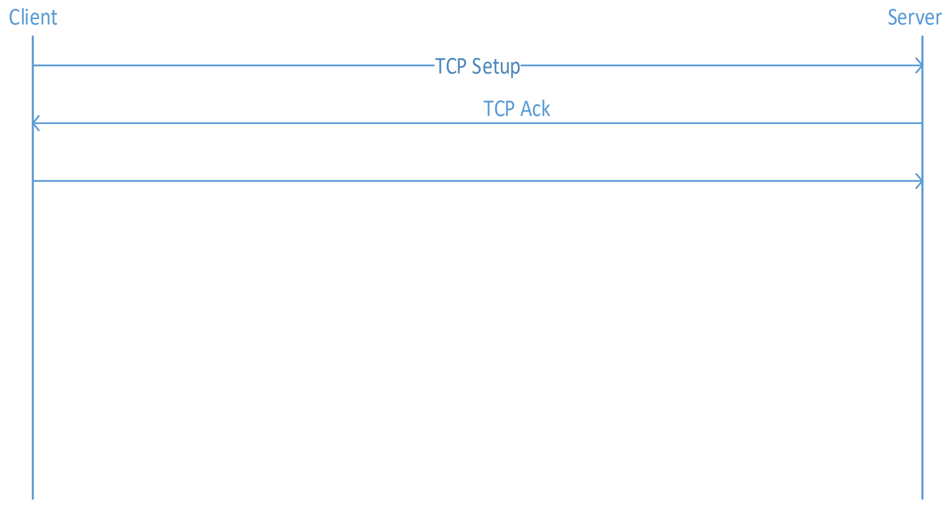


```
<terminated> TCPClientApp (3) [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (18 Jan 2017, 11:20:20 am)
Response: This is /127.0.0.1:12345, current time here is: Wed Jan 18 11:20:20 SGT 2017
```

```java
package tcpclientapp;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;

public class TCPClientApp {

    public static void main(String[] args) {
        //Name of the host that we are going to conenct to
        String hostName = "☐";
        //Make sure that this port is the same as
        //the serrver listenign port
        int port = ☐;
        try {
            //Use socket to connect to the server
            Socket socket = new Socket(hostName, port);
            String request = "☐";
            //Access to the output stream
            OutputStream os = socket.getOutputStream();
            //Write lie
            PrintWriter pw = new PrintWriter(os);
            pw.println(☐);
            pw.flush();
            //Read response.
            InputStream is = socket.getInputStream();
            //Read characters
            InputStreamReader isr = new InputStreamReader(is);
            //Read lines
            BufferedReader br = new BufferedReader(isr);
            String header = br.readLine();
            System.out.println("Response: " + ☐);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Answer the following questions:

1) What is the purpose of the port number?
2) Is TCP/IP socket programming location transparent? Explain.
3) Study the programming codes carefully and describe the protocol interactions between the client and server by completing the message sequence chart as shown.



Assignment 1

Based on your understanding of the previous exercise, develop a client-server application with the following requirements:

The server application

You are required to build a TCP Server application which is used to receive a command from a client application, process it, and send the result back accordingly

The command should be in the form "*cmd x y*"
- *cmd* will represent either: add, subtract, multiply, divide
- *x, y* represent two numbers as two arguments for the *cmd* operation to work

The server should perform the following validations and return the corresponding messages in case there is a validation error:

- Invalid command
- Invalid number of arguments
- x, y must be numbers
- If the command is a division, then y should not be zero
- *Any other validations that you feel appropriate to add*

The client application

You are required also to build a TCP Client application which is used to send a command to the server to process, then receive the result and display it.

Sample output 1:
Please input command: add 10 20
The add result is: 30

Sample output 2:
Please input command: nothing 10 20
Error: Invalid command "nothing"

Sample output 3:
Please input command: divide 20 0
Error: Divided by zero exception

Sample output 4:
Please input command: add abc 10
Error: "abc" is not a number

Sample output 5:
Please input command: add 10
Result: Invalid number of arguments

Assignment 2

A Simple Web Server

Riding on what we have learned so far, we are going to build a simple web server that is able to communicate with a commercial web client such as Chrome, Fire Fox or IE explorer. The server should be able to seamlessly communicate with commercial web browsers if it is implemented correctly, that is, the HTTP.

A summarized flow of your web server should behave something like this:

1. Open a ServerSocket.
2. Wait for client connection request (accept).
3. Open I/O streams from the socket.
4. Read the request line of the client's http request message.
   e.g. GET index.html HTTP/1.0
5. Check the command (e.g. GET) and react accordingly.
6. Suppose it is a GET. Extract the path and filename from the request line.
7. Open the html file requested (what if it doesn't exist? Check the specification of HTTP)
8. Send the response line to the client first.
   e.g. HTTP/1.0 200 OK
9. Read in each line of the html file, and send it back to the client (until end-of-file).
10. Close the I/O streams, file and finally the socket connection.
11. Go back to step 2 to wait for another http request.

You may want to create a simple HTML file to test your server. A sample is shown below:

```
<HTML>
<TABLE border="1"
          summary="This table gives some statistics about fruit
                   flies: average height and weight, and percentage
                   with red eyes (for both males and females).">
<CAPTION><EM>A test table with merged cells</EM></CAPTION>
<TR><TH rowspan="2"><TH colspan="2">Average
    <TH rowspan="2">Red<BR>eyes
<TR><TH>height<TH>weight
<TR><TH>Males<TD>1.9<TD>0.003<TD>40%
<TR><TH>Females<TD>1.7<TD>0.002<TD>43%
</TABLE>
</HTML>
```

Using your favorite browser, test out the web server.