Lab Exercise 2: Socket Programming

OBJECTIVE:

The objective of this lab is to learn basics of client/server programming with sockets and gain a deeper understanding of the transport layer protocols such as UDP and TCP. A simple UDP application will be developed to demonstrate the communications between a client process and a server process.

BACKGROUND:

Socket opens a “door” between the application process and the transport layer protocol. By using sockets, two (or more) processes running on different machines can communicate and exchange data. The original Berkeley sockets application programming interface (API) was included in the BSD Unix operating and comprised a library for developing applications in the C programming language that perform inter-process communication, most commonly for communications across a computer network. Nowadays, many programming languages such as Java and Python all provide socket programming interfaces.

LAB ACTIVITIES:

1. Read carefully the attached source code UDPClient.java and UDPServer.java for a client and a server based on user datagram protocol (UDP). These programs illustrate how two processes can communicate using **datagram sockets**. The source code TCPClient.java and TCPServer.java are a client program and a server problem based on transport control protocol (TCP). These programs illustrate how two processes can communicate using **TCP sockets**.

2. Test and run the server and client programs as follows. Fig. 1 shows the architecture of the client side and the server side.

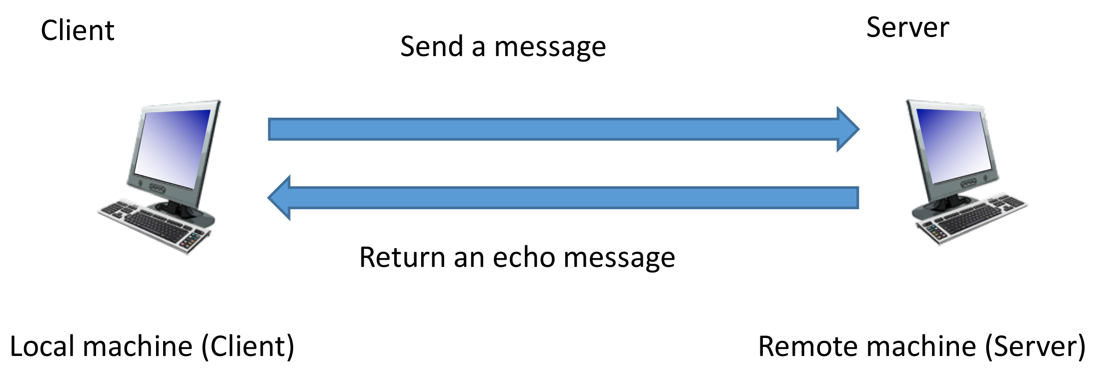

Fig. 1. Architecture of the client-server echo application.

a. Start your UNB VPN client.

b. Open a terminal to start an ssh session for remote login to any of the lab machines (e.g., in GWC112, ITD414, or ITD415), following the instructions posted at https://www.cs.unb.ca/help/ssh-help.shtml.

c. Use the ssh session to upload your Java source files UDPClient.java and UDPServer.java to the remote machine using scp (secure copy protocol):

```
scp /local_path/<java_file> <username>@<hostname_svr>:
<java_file>
```

Here, `<java_file>` is the Java source file to upload, `<username>` is your UNB ID, `<hostname_svr>` (in the format of machine_name.cs.unb.ca) is the hostname of the above remote machine to where you have logged in.

To upload multiple Java files, use the following command:

```
scp /local_path/*.java <username>@<hostname_svr>:~
```

d. Compile the above uploaded Java source files using the command:

```
javac UDP*.java
```

e. Then use the ssh session to run the sever class on the server machine:

```
java UDPServer
```

f. Open a new terminal to start an ssh session for remote login to a lab machine different from the remote machine you used to run the server. Then run the client class using the following command, where `<hostname_svr>` is the hostname of the above server:

```
java UDPClient <hostname_svr>
```

g. Follow the prompt on the client side to send a message; check the server side and see if the server receives the message; and then check the client side and again and see if the client receives an echo message sent back from the server to the client.

h. Once you finish the testing, make sure you kill the UDP server process (Ctrl+C). Otherwise, the server process will occupy the server's port number and interfere your further testing.

i. Follow the same procedure to try the example client and server based on TCP socket.

3. Next, write a new socket client and a socket server that can **transfer a text file over TCP.** Use the attached text file PoemShakespeare.txt (a line by line poem by Williams Shakespeare, titled *Blow, Blow, Thou Winter Wind*).

   a. Your TCP client should read the text line by line from the file and send it to the server, which displays each line received at the console.

   b. Once all the lines in the poem are sent to the TCP server, the client should further send a **"Done"** message to signal the end of the file transfer to the server, and the server responds to the client a **"Bye"** message. The above two messages exchanged between the client and server should be case-insensitive. Note: the server can only send out the "Bye" message after the server checks that the "Done" message from the client has been received.

c. Only after receiving the “Bye” message, should the client close the connection with the server, and display the total number of data sent in bytes and the total time in milliseconds spent in file transfer. This total time is counted from the moment before the client sends the first line to the moment after the “Bye” message is received.

NOTE: When testing your client and server, you should follow the above procedure to upload your Java files to two remote machines, compile them, and run the class files. The text file PoemShakespeare.txt should also be uploaded to the client machine.

For simplicity, you can hardcode the text filename and only take the server’s hostname as the argument to the client. That means, your client is supposed to take the following command format for running the client:

```
java <your_client> <hostname_svr>
```

LAB SUBMISSION:

Go through the above lab activities. Then, write your own client and server programs **in Java** according to the requirements in item 3. Please add comments in your code and employ good programming practices.

- For this lab, **write a lab report using the posted Word template, including your source code as appendices.** Convert the lab report to a PDF file and submit it to the dropbox on D2L by the due time.
- Also, you need to submit your **source code in Java (the original “.java” files)** to the dropbox on D2L by the due time.
- You don’t need to submit the provided “PoemShakespeare.txt”.