Lab Exercise 4: Basic Socket Programming[1]

(Total: 25 points)

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 TCP-based Q&A 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. 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.

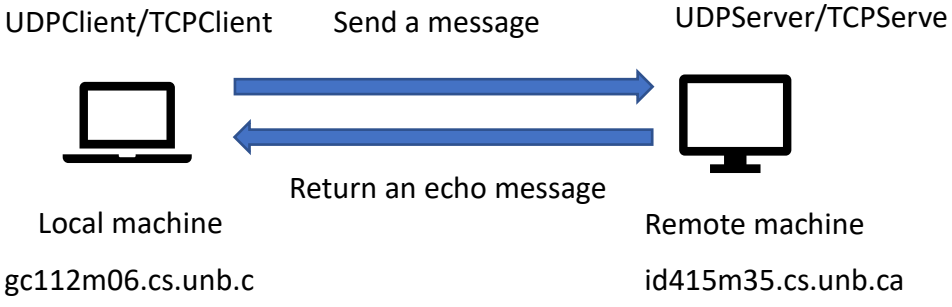

Fig. 1. Architecture of the client-server echo application.

3. If you are working off campus, do the following steps.
   a. Start your UNB VPN client.
   b. Open a terminal and upload your Java source files to the remote machine using scp (secure copy protocol):

[1] The lab materials are adapted from the examples in the textbooks "Computer Networking – A Top-Down Approach" by James Kurose and Keith Ross. They can only be used by students who registered for this course. Reproduction outside of this course use is prohibited.

```
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>:~
```

c. Within the terminal, start an ssh session and remote login to any of the lab machines (e.g., in GWC112, ITD414, or ITD415) as your server machine, following the instructions posted at https://www.cs.unb.ca/help/ssh-help.shtml.

4. If you are working in the lab on campus, do the following steps.

   a. Download, unzip the package to get the Java source files, and save them into the home folder of your lab machine.

   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) as your server machine, following the instructions posted at https://www.cs.unb.ca/help/ssh-help.shtml.

5. Compile and run the TCP example.

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

   ```
   javac TCP*.java
   ```

   b. Go to the terminal on the remote machine as the server, run the sever class using the following command:

   ```
   java TCPServer
   ```

   c. Go to the terminal on your local machine as the client, run the client class using the following command, where `<hostname_svr>` is the hostname of the above server:

   ```
   java TCPClient <hostname_svr>
   ```

   d. 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.

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

<u>You can follow the same procedure to try the example client and server based on UDP socket.</u>

6. Next, write a new socket client and a socket server that implement the following functions.

   a. The client reads out the calculation questions from a text file and sends them line by line to the server **by using TCP**.

   b. Once the server receives a question, it converts the received text into a mathematic expression and evaluates it. After that, the server sends the answer back to the client **by using TCP.**

c. Once the client sends over all questions to the server, it sends the last message **"DONE\n"** to the server. Once the server receives this last message, it replies **"CLOSE\n"** and then close the connection with the client.

d. The server should remain open to accept new clients after it is done with one client. To be efficient, the client should only set up a connection with the server once for all questions.

e. For debugging purposes, the client and server both display their sent and received messages on the screen. The client also records the turn-around time it takes to send each question and receive the corresponding answer. Once the client receives the "CLOSE" message from the server, it displays the number of characters it has sent and received **(including characters in messages "DONE\n" and "CLOSE\n")** and the average turn-around time over all questions. After that, the client terminates its connection with the server.
The following is an example of the final display at the client side:

```
#20--------------------------
Question from client: 89.03 - 86.453
Answer from server: 2.5769958
Q&A END******************************
Total characters sent: 331
Total characters received: 183
Avg turn-around delay per question-answer: 0.6ms.
```

NOTE: When testing your client and server, you should make sure your Java files are properly stored on two different machines as shown above and these files are compiled properly to Java classes for running.

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>
```

Last but not the least, I strongly recommend you use Git or other version control tools in your coding and testing. Most Java IDEs such as IntelliJ IDEA and VS Code can easily integrate these plug-ins.

LAB SUBMISSION:

Go through the above lab activities. Then, write your own client and server programs **in Java** according to the requirements in item 6. Please add comments in your code and employ good programming practices.

- For this lab, **write a lab report using the posted Word template.** Convert the lab report to a PDF file and submit it to the dropbox on D2L by the due time.

- Also, you need to zip your **source code in Java (the original ".java" files)** and the provided text file and submit the zip package to the dropbox on D2L by the due time. These source code are important to run and test your programs.

APPENDIX[2]:

- **`ServerSocket`**

  This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

  `public Socket accept()`

  Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

- **`Socket`**

  This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

  `public OutputStream getOutputStream()`

  Returns an output stream for this socket.

  `public InputStream getInputStream()`

  Returns an input stream for this socket.

- **`DataOutputStream`**

  A data output stream lets an application write primitive Java data types to an output stream in a portable way.

  `public final void writeBytes(String s)`

  Writes out the string to the underlying output stream as a sequence of bytes. Each character in the string is written out, in sequence, by discarding its high eight bits. If no exception is thrown, the counter `written` in the `DataOutputStream` is incremented by the length of string **s**.

- **`DataInputStream`**

  A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.

- **`InputStreamReader`**

  An `InputStreamReader` is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

- **`BufferedReader`**

[2] References: Java online documentation for packages `java.io and java.net`.

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

`public String readLine()`

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a line feed. Returns a `String` containing the contents of the line, not including any line-termination characters, or `null` if the end of the stream has been reached