SENG3400/6400 Network and Distributed Computing
Assignment 1 – 20%

Due Date: 2nd September 2018 (via Blackboard)

Note: Please read the important submission information at the end of this specification!

This assignment will require you to research and implement an assignment using the Java Socket API. You will be implementing a pair of client and server applications which allow two users to chat via a simple chat protocol (SCP).

SENG3400 students have the option to complete this assignment either individually or in pairs. For individuals, only part 1 of this specification needs to be implemented. For pairs, parts 1 & 2 need to be implemented. Pairs submitting only part 1 will have their marks divided equally between the two. Individuals submitting parts 1 and 2 will not receive extra marks. SENG6400 students must complete both parts 1 & 2 individually.

Getting Started

Tutorial on Java Sockets:
http://download.oracle.com/javase/tutorial/networking/sockets/

Socket Javadoc:
https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html

ServerSocket Javadoc:
https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html

Half Duplex – Allows transmission in both directions, not simultaneously
Full Duplex – Allows transmission in both directions simultaneously

Part 1 – Half-Duplex Chat with SCP

SCP is a very simple chat protocol. It functions similarly to a 'walkie talkie' where communication is one-way, with only one user capable of sending messages at a time. Users take turns in sending messages. A user must wait for a response before they can send another message.

A SCP server only supports one connection at a time. If the server is busy with another client, the request will be ignored or eventually timed-out by the Socket API. The SCP protocol is aware of 6 message types which the server and client must be aware of:

- CONNECT
- REJECT
- ACCEPT
- ACKNOWLEDGE
- CHAT
- DISCONNECT

The operation of the protocol is as followed:

1. The client will send a SCP connection request structured as followed.

```
SCP CONNECT
SERVERADDRESS <server hostname>
SERVERPORT <server port>
REQUESTCREATED <time since epoch>
USERNAME <username as String with quotes>
SCP END
```

For example

```
SCP CONNECT
SERVERADDRESS 127.0.0.1
SERVERPORT 3400
REQUESTCREATED 1533018800
USERNAME “Hayden”
SCP END
```

2. The server will read and process this request.
If the passed epoch time is more than 5 seconds different to the current epoch time, then the server will send a response message rejecting the connection, and then proceed to close the accepted client socket. The rejection message is structured as followed.

```
SCP REJECT
TIMEDIFFERENTIAL <time difference>
REMOTEADDRESS <requesting clients hostname>
SCP END
```

Otherwise the server will accept the connection and send message:

```
SCP ACCEPT
USERNAME <username as String with quotes>
CLIENTADDRESS <client hostname>
CLIENTPORT <client port>
SCP END
```

3. The client will acknowledge the acceptance of the request by sending message:

```
SCP ACKNOWLEDGE
USERNAME <username as String with quotes>
SERVERADDRESS <server hostname>
SERVERPORT <server port>
SCP END
```

4. The server will register the acknowledgement and initiate a chat. The server will send the first message of the chat (the welcome message passed upon start-up – see section Submission). The client and server will then each take turns sending CHAT messages.

```
SCP CHAT
REMOTEADDRESS <remote hostname>
REMOTEPORT <remote port>
MESSAGECONTENT
<line feed>     ← This will cause 2 ‘\n’ characters to be sent!
<message contents>
SCP END
```

The serialised message contents may be over many lines. You do not need to escape any existing newline characters in the message contents, **but you need to account for them when receiving**.

5. When either the server or the client receives user input “DISCONNECT”, the application will send the following message.

```
SCP DISCONNECT
SCP END
```

The receiver will then send a modified ACKNOWLEDGE message back to the client

```
SCP ACKNOWLEDGE
SCP END
```

When the DISCONNECT sequence has finished, the client will close its socket. The server will close the connection to the client and resume waiting for a new client.

Notes:
In all transactions between the client and the server each line is terminated by a linefeed ‘\n’ character. **In a CHAT message the indicated <line feed> is distinct to this linefeed character.**

Each request is **terminated by the sequence ‘SCP END’**. This is to indicate there is no more content to be read for the request. **You may assume that the user will never enter the message ‘SCP END’ when chatting. You many assume that every transmitted message will be terminated by ‘SCP END’.**

If at any point the server or the client receives a message which is malformed (e.g. missing data) or invalid, the application is to close the socket connection and terminate. You will need to print the exact reason to why the termination occurred (e.g. received message X but expecting message Y, expected value Z)

Part 2 – Full-Duplex Chat with SCP

After identifying the limitations of the SCP client and server, the developers have decided to extend SCP to allow for full-duplex chat between clients. A server will still be locked to a single client at a time, however both the client and server will be capable of sending and receiving messages simultaneously. i.e. Each user will not have to wait for the other to send a message.

The SCP protocol will not be modified. Only the functionality of the client and server. You will have to identify how to enable reading and writing **in parallel** to a socket in a command-line application. You will need to identify how to manage reading and writing to a console in parallel. You *may* use a simple GUI to ease this process, however this is at your own discretion.

Hint: You will need to use threads!

Usage

The SCP protocol specifies how a SCP client and server communicate. The SCP protocol is modelled around message passing, where instructions are sent between the client and server in well-formatted messages. A SCP client and server allow two users to communicate via command line. The usage of the applications (conforming to part 1 only) from a user perspective is as followed.

User A will start the ChatServer application.

```
java ChatServer "localhost" 3400 "Welcome to my Chat!"
```

When the ChatServer starts up it will bind a Java ServerSocket on the specified hostname & port and begin listening for new connections. The ChatServer only needs to support 1 chat. There is no need for a ChatServer to handle multiple clients simultaneously. A ChatServer is essentially a client that hosts the SCP chat session. It does not act as a broker for multiple clients chatting in parallel.

User B will start the ChatClient application.

```
java ChatClient "localhost" 3400
```

The ChatClient will initiate a connection to the ChatServer. The client will open a Socket connection to the passed hostname & port. It will begin by sending a valid CONNECT message by the socket. This will require asking the user their screen name, entered by console input. The user does not have to type the SCP protocol CONNECT message manually, it will be automatically formatted by the client and transmitted to the server.

The ChatServer will then read the transmission and send an appropriate response (i.e. a REJECT or ACCEPT message). If an ACCEPT response is received, the ChatClient will send an ACKNOWLEDGE message. This is to inform the server that the client is aware and ready to start a chat. The ChatServer will then send the first CHAT message, containing the welcome message specified at startup.

The ChatClient and ChatServer will begin sending CHAT messages in a half-duplex manner. The ChatClient will start sending messages, followed by the ChatServer, and repeating in this loop until either User A or User B enters "DISCONNECT" into the console to terminate the chat. When the user has their turn to type, they will be prompted to enter a message. When the user is awaiting a response, they will be informed that the other user is typing.

When the special "DISCONNECT" message is sent, the recipient application will send an acknowledge message, and close their Socket connection. The ChatServer will begin waiting for a new client to connect in order to begin a new SCP chat session.

At no point is a user expected to type a message of the SCP protocol. For example, when connecting they WILL NOT have to print the entire 'SCP CONNECT …' sequence. The most a user will type is their user name and the actual text to be sent as a chat message.

Submission

The entry point to the server application MUST be named *ChatServer*. The entry point to the client application MUST be named *ChatClient*.

Both the *ChatServer* and *ChatClient* applications will accept command-line parameters upon startup. A description of the parameters is as followed:

For *ChatServer*:

- Parameter 1 will be the hostname (or ip) the server will bind to (a String)
- Parameter 2 will be the post number the server will bind to (an int)
- Parameter 3 will be a welcome message (a single String)

For *ChatClient*:

- Parameter 1 is the hostname (or ip) the client will connect to (a String)
- Parameter 2 is the port number the client will connect to (an int)

If any of these values are missing you will use a default value of "localhost" for the hostname, **3400 for the port number**, and "Welcome to SCP" for the welcome message. If either of these values are invalid (e.g. invalid hostname or port number) you will terminate the program.

It is imperative that you use a server port number bigger than 1023 so that you do not interfere with any system-provided ports.

When your submission is marked, we will use a sample client to interact with your server, and a sample server to interact with your client. It is therefore non-optional that you comply with the specified interaction protocol (which is, of course, consistent with the 'real world' of computer networking.

Compiling the assignment should be as simple as executing javac *.java. If there are any extra steps to compilation, please document this in your submission. You are not to use any third-party libraries. Java Sockets **MUST** be used.

Submissions will be tested against a correct implementation of a SCP client and server. Marks will be allocated for the correctness of the implementation of the SCP protocol.

All assignments are to be submitted via Blackboard. Assignments are to be implemented using Java 8. You are to submit all .java files and any other dependencies required to compile and run your submission. Both individual and pair submissions must be accompanied by an assignment cover sheet. Only one submission is required for pairs.

You will be expected to print appropriate user feedback. Remember the applications should be user friendly. But also keep the marker in mind to give them sufficient feedback such that they know what is occurring in your application. At the very least you should print user feedback when:

- **Waiting for a new client to connect;**
- **Waiting for a message to be sent;**
- **Tell the user they need to send a new message;**
- **And on every message sent/received (say what message type is sent/received).**

Assignments which do not follow this specification will not be marked.

Marking Rubric

SENG3400 Individuals

Part 1 – 20 Marks

Marks	Description
2	Server is correctly defined, and can listen on correct ip & port
2	Server can communicate correctly with student SCP client
6	Server can communicate correctly with correct SCP client
2	Client is correctly defined, and can open a socket to the server
2	Client can communicate correctly with student SCP server
6	Client can communicate correctly with correct SCP server
Total 20	

SENG3400 Pairs & SENG6400

Part 1 – 10 Marks

Marks	Description
1	Server is correctly defined, and can listen on correct ip & port
1	Server can communicate correctly with student SCP client
3	Server can communicate correctly with correct SCP client
1	Client is correctly defined, and can open a socket to the server
1	Client can communicate correctly with student SCP server
3	Client can communicate correctly with correct SCP server
Total 10	

Part 2 – 10 Marks

Marks	Description
7	Implementation of full-duplex communication
3	Management of user input & output
Total 10	

Note:
The 'correct' SCP client and server refers to the reference implementation provided by the lecturer or marker.