CM10228 Coursework 1
Simple Chatting System

8th February 2021

1. Introduction

In your first programming assignment for this semester, you will create a simple chat system. You will have to code both a *client* and a *server* component for the system.

You can use any Integrated Development Environment (IDE) for the development of your code, but your code must run on **Java 11** (the default version used on the University's linux.bath server) without requiring the installation of additional libraries, modules or other programs.

Please check Moodle for the submission deadline.

2. Submission

By the date/time specified on Moodle, you should upload a single zip file. The name of the zip file must be in the form **<*your id*>-CW1.zip,** e.g. **cjc234-CW1.zip**.

The following should be in the root of the zip:

1. Your source (.java) files. Do not include packages or other subfolders. Do not include compiled (.class) files. If you are using an IDE to develop your code, you must extract the source files – do not include the entire project structure or version control files.
2. A Readme.txt file (or alternatively, Javadoc files).

The main file for your server should be *ChatServer.java*, and the main file for your client should be *ChatClient.java* (please see 3.1 and 3.2 for more details).

Failure to follow the submission specifications, by providing unneeded files or not following the .zip structure specified, will result in a penalty being applied to the final mark.

Your .zip file must be submitted to the Moodle assignment page by the submission deadline shown. Submissions received after this deadline will be capped at 40% if received within 5 working days. Any submissions received after 5 working days will marked at 0%. If you have a valid reason for an extension, you must submit an extension request through your Director of Studies – unit leaders cannot grant extensions.

3. Specifications

Below are the specifications for the basic chat server and client. A total of 50% can be achieved by providing working code that satisfies all the requirements. In addition, 30% of the marks are allocated to the quality of your code (i.e. good coding practices, formatting and commenting). The final 20% can be achieved by implementing both advanced features outlined below in section 3.3.

3.1 Server

The first component you have to code is a *multi-threaded server*. You will have to spawn threads to handle incoming requests in parallel. In addition to the list of requirements below, we will mark your approach on networking and concurrency. Make sure you use locks or implement concurrent data structures, as required, to make the chat server thread safe.

1. The server can handle connections by multiple clients.
2. The server accepts clients connecting through a TCP/IP connection, using Java's *Socket* API.
3. The server can receive messages from clients.
4. The server broadcasts all messages received to all connected clients.
5. The server continues running if one or more clients disconnect from it.
6. The server shuts down cleanly if its user enters *EXIT* on the terminal (hint: toLowerCase() may help here).
7. The server uses 14001 as a default port.
8. The server's implementation contains a class called *ChatServer*, allowing the software to start by running the following on *linux.bath*:

   ```
   java ChatServer
   ```

9. You can pass an optional parameter, called *csp*, to ChatServer to request that it bind to another port. For example:

   ```
   java ChatServer -csp 14005
   ```

3.2 Client

The second component you are to create is a *client*. The client is responsible for sending and receiving messages from the server. Think of it as any Instant Messaging (IM) application, for example *WhatsApp* or *MSN Messenger*. The following specifications are the minimum requirements for your client:

1. Your client must be able to read messages from the console.
2. The client can send messages to a *server*.
3. The client must be able to receive messages sent from the server.
4. The client should output all messages received by the server.

5. The client connects to the server through a TCP/IP connection, established using Java's *Socket* API.
6. The client sends and receives any messages through the socket connection established in the step above.
7. The client's implementation contains a class called *ChatClient*, allowing the execution of the software by running the following on *linux.bath*:

   java ChatClient
8. The client should use *localhost* as a default IP address to try to connect to.
9. The client should use 14001 as a default port.
10. You can pass an optional parameter, called *cca*, to ChatClient to request that it bind to another IP address. For example:

    java ChatClient -cca 192.168.10.250
11. You can pass an optional parameter, called *ccp*, to ChatClient to request that it bind to another port. For example:

    java ChatClient -ccp 14005
12. You can pass both the IP address and port of the server. For example:

    java ChatClient -cca 192.168.10.250 -ccp 14005

If your client supports reading input from the console and displaying messages at the same time, by using a multi-threaded solution, you will be awarded additional marks.

3.3 Advanced Features

3.3.1 Advanced Feature 1: Simple Chat Bot

Allow user(s) to talk to a chat bot - note we do not expect you to implement AI and NLP algorithms, rather you can achieve full marks with a chat bot that provides simple (e.g., pre-scripted) responses to user interactions with the chat bot. See for example https://rubberduckdebugging.com/.

The chat bot must be run as a client (i.e., run with an IP address and port provided as command line arguments, then connect to the specified server).

3.3.2 Advanced Feature 2: DoD Integration

Allow your user(s) to play Dungeons of Doom! Adapt your DoD implementation so that it can connect to the server as a DoDClient. i.e., The IP address and port are provided as command line arguments, then connects to the specified server. The DoDClient should then allow other users to join and control a player in the Dungeon of Doom. For example:

> [Alan] JOIN

> [DoDClient] Player Alan has been spawned.

> [Ben] JOIN

> [DoDClient] Player Ben has been spawned.

> [Alan] MOVE W

> [DoDClient] Player Alan: SUCCESS.

> [Alan] EXIT

> [DoDClient] Player Alan has exited.

You will get additional marks for the DoDClient only sending responses to individual players. E.g. if player Alan calls LOOK, only Alan should see the map in response.

Note - we will not mark your implementation of DoD, just how you have implemented networked client functionality into the game.

4. Marking Scheme

4.1 Marks Distribution

The table below outlines the high-level distribution of marks.

Criteria	*Max score*	*Description*
Client Specifications - 20		
Basic Client	10	Client satisfies core specifications.
Multi-threaded Client	10	Client can read/write from console simultaneously.
Server Specifications - 30		
Basic Server	15	Server allows at least one client to be connected.
Concurrency	15	Server allows more than one client to be connected and uses thread safety techniques.
Code Quality - 30		
Good-code practices	20	Object-oriented programming techniques, including but not limited to encapsulation and single-responsibility principle, have been used. Code is sensible, with optimisation taken into account.
Formatting & Comments	10	Indentation, whitespace, sensible comments throughout the submission
Advanced Features - 20		
Chat Bot	10	Allows user(s) to talk to a chat bot - note we do not expect AI and NLP to be used, rather you can achieve the full 10 marks with a chat bot that provides simple responses to user interactions.
DoD Integration	10	Allow user(s) to play Dungeons of Doom - use your DoD implementation and turn it into a client. Note – we will not mark your implementation of DoD, just how you have implemented networked client functionality into the game.

4.2 Marking Guidelines

The following are guidelines for what is required for the respective marks:

4.2.1 Threshold for Pass (40%):

1. Basic networking functions: able to demo on linux.bath connecting one user client to server.
2. Code is good i.e., well formatted and commented.

4.2.2 Good Pass (≈60%):

1. Meets all the criteria of the pass threshold above.
2. Provides networking support for multiple clients.
3. Code follows clean-code practices, i.e., very good use of OO design etc.

4.2.3 Distinction (70% - 80%):

1. Meets all the criteria of a good pass.
2. The client is also multi-threaded.
3. Concurrency principles, such as threads safety and atomic access, were taken into account.

4.2.4 Distinction (80-90%):

1. Meets all the criteria of a distinction.
2. Implements one advanced feature.

4.2.5 Distinction (90-100%):

1. Meets all the criteria of a distinction.
2. Implements both advanced features.