Komodo - Client-Server Chat Program

Group Members:

Gareth MacBeth: 20129882@sun.ac.za
Michelle Mitchell: 19976755@sun.ac.za

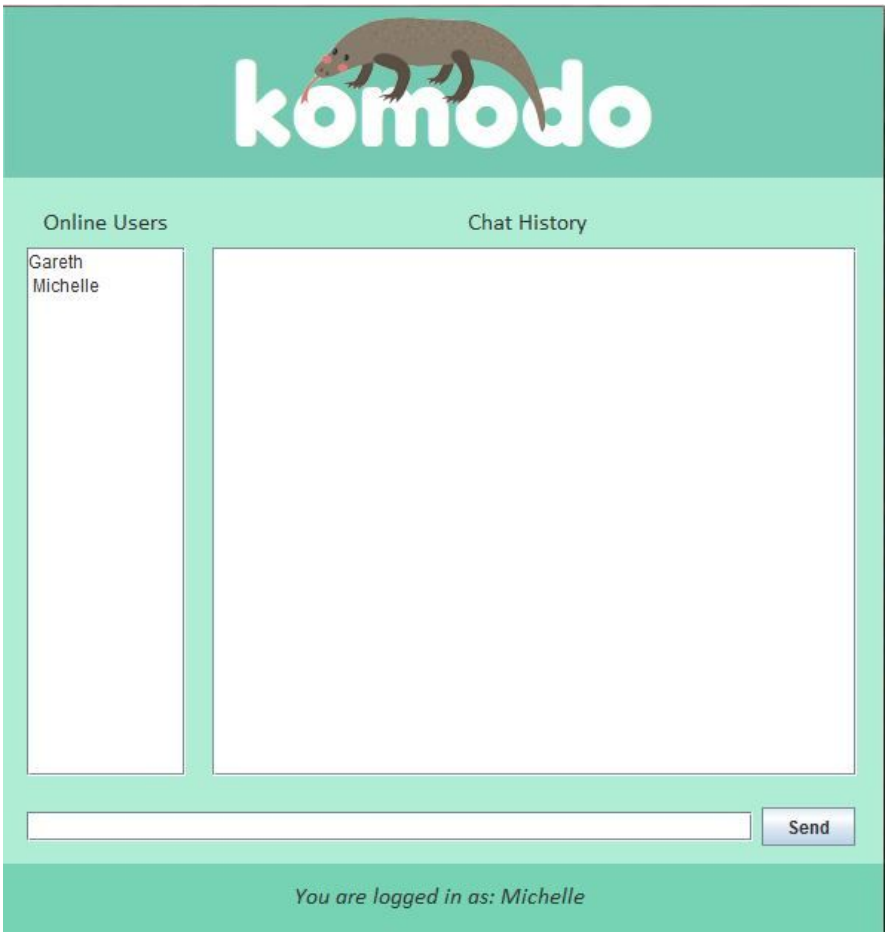
Introduction:

The purpose of this project was to develop a chat program based on a client-server model, where multiple clients communicate with a single server using multithreading. The implemented, as well as unimplemented features will be discussed and an explanation of the source code provided. The project was coded using Java, and the Graphical User Interface (GUI), was specifically made using Swing.

All algorithms and data structures were carefully chosen for the chat model to provide an efficient, stable program that is also scalable.

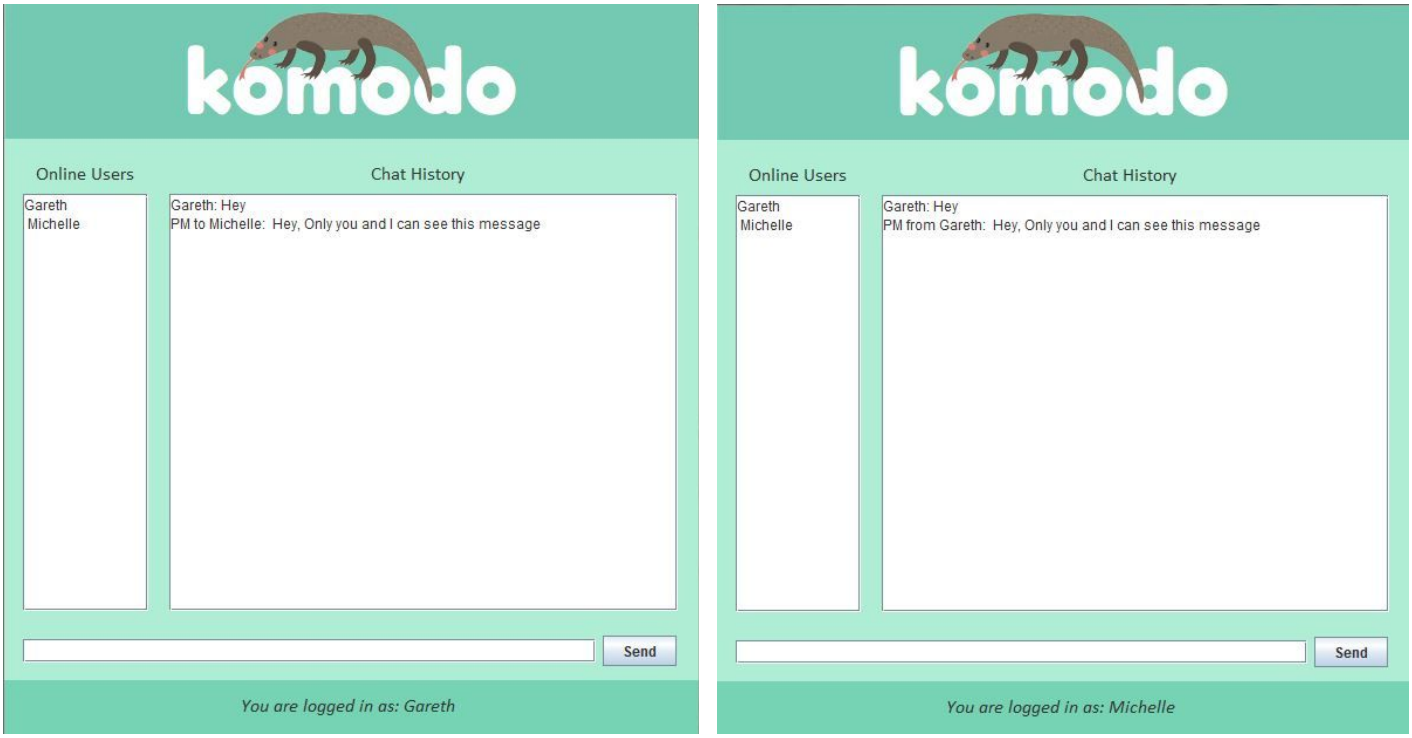
Implemented Features:

- ***GUI:***



  - A client GUI was implemented that provided the Client (End-user) with a text input field in which they could enter text.
  - A text field is appended to the GUI upon entering a unique username, as can be seen above, identifying your GUI as yours.
  - The GUI contained a JTextPanel that displayed the online users who were connected to the server at that time, and was updated according to user connect/disconnect requests.
  - The Chat History field displays the messages that all users have sent as broadcast messages(messages are broadcast by default).
  - Whispering(Private Messaging) was implemented, but is explained under the section Whispering.

- ***Whispering Messages:***



  - Users can send and receive messages privately by specifying the user which they want to send the message to. This is done by prepending the message with "@" and specifying the username directly after, and escaping the username with a colon ":"
    - The sequence is as follows: @Gareth: Hello

- ***Prevention of Duplicate usernames:***
  - Upon entering a user name, that name is added to an ArrayList that stores all the Users currently connected to the server. When another user attempts to connect to the server and enters a username, that new username is checked against all usernames currently in the username Arraylist, if a match is found an Error dialog is output as follows:

- ***Concurrency:***
    - Both the server and the client are run concurrently, thus allowing multiple users to connect/disconnect, send broadcasts as well as private messages without interrupting the connections of other users.
    - Running multiple threads (one for each client) allows the program to provide stability both to the Client after a server termination has been detected and the Server after a client termination has been detected.

Unimplemented features:

- ***All required features have been implemented***

Additional Features:

- **Message Logs:**
    - While the server is still active and there are clients connected to the server, every time a client sends a message to another client, that message is output to a plain text file called “ChatServer-logs.txt”.
      This allows clients to look at all the messages sent during the period in which they were active, thus it acts as an archive for all user interactions. This provides users with a backup of there’s and others texts in the event of a server-side/client-side program crash.

Experiments:

- ***The use of PrintWriter:***
    - In the first iteration of the Client-Server Chat Program, a PrintWriter was used in place of a OutputStream such as a BufferedWriter and/or DataOutputStream. This was done in order to gain a basic understanding of how to send data over sockets, before implementing the more complex OutputStreams.
    - **Conclusion:**
        - Much was learnt about the general implementation of sockets and the methods with which data can be sent across sockets between server and client and vice versa.
- ***Synchronising Threads:***
    - Before sending the data across a socket, the data to be sent was printed out. Once the data had been sent it was printed out at the point at which it was received, the two data sets were then visually compared to see if the

data had been transferred correctly, and thus in synchronisation with the program.

- **Conclusion:**
  - The code was correctly implemented to begin with, thus the code did not need any further altering to ensure the correctness of the code.

Issues Encountered:

- ***newLine() and flush() statement for BufferedWriter:***
  - Initially the newLine() and flush() statements required for a BufferedWriter to correctly send data via an OutputStream was overlooked. This lead to problems in code, that required debugging until it became clear that these statements were required.

Description of Files:

- **ChatServer.java** listens for incoming connections on a specified port, and upon accepting them - opens a thread for them, which is handled by the class ConversationHandler. Each thread then validates the username send to it from ChatClient.java, and waits for incoming messages to send to each client's GUI. Each thread has an arrayList of userBuffer objects (containing BufferedWriters of each thread, as well as the name of the user they are associated with) which send the desired messages to the GUI.
- **ChatClient.java** requests a user to input the IP address required to connect to the server. Once the connection has been successfully established, the GUI is launched and the user is able to enter their name. A BufferedReader and BufferedWriter is then assigned to the chat client to add the user to an ArrayList of User Objects (containing the user and their IP Address), read messages from the GUI, and send them to each thread.
- **Display.java** contains all the code used to build the GUI using the Swing toolkit, as well as handle Action and Window events.

Program Description:

In order for the program to start successfully, the Server-side program must be started first and then the Client-side program can be started. Once both programs have been started, the Client-side program will prompt the user for input in the form an IP address, which is used to establish a connection between the client and the server, once this connection has been authenticated without error, the Client-side program will once again prompt the client to input a unique username, thus one that is not already taken by another active user, if the client inputs a username that is taken they will alerted to the existence of that username and be required to enter a new username.

The client will now be free to send messages to other connected users at will, without any interruption. If a client disconnects from the server, all other clients currently connected will be notified of that clients disconnection, however neither the server nor the client will be interrupted.

If for some reason the server were to terminate before all the clients disconnected, all remaining clients will be notified by a JOptionPane message dialog that the server has terminated and that they no longer can send messages, however, there GUI will not be closed automatically, this is done to ensure that clients have time to read all the messages leading up to the servers termination.

Significant Data Structures:

- The ArrayList of User Objects, called User Array which is held in ChatServer is arguably the most important data structure used in the program. User objects have two members of type String, to store the user's name as well as their IP Address.
- There is also an ArrayList of userBuffer Objects named userBuffers which is used by each thread to keep track of the user associated with each BufferedWriter. This was very useful in being able to implement private messages - as the sender and receiver's BufferedWriters could be easily isolated using a simple for loop that iterates through each userBuffer Object in the array.
- ArrayLists were used wherever possible due to the fact that their size is not fixed, and they are able to simply grow as needed.
- Although PrintWriters were used in our first attempt at the project, we replaced them with BufferedWriters because BufferedWriters can

efficiently to write to files (or DataStreams) as it buffers the characters in Java memory, preventing loss of data.

Compilation:

- A Makefile is included within the project directory, thus in order to compile all the java source files, the command line must be in the clients working directory and the client must run the following command:

make - f Makefile

Execution:

- To execute the code successfully, the client must have two terminals open in the programs working directory. In the first terminal, the client must enter the following command:

java ChatServer

- In the second terminal, the client must enter the following command:

java ChatClient

- The program should now be running and the client must now follow the prompts as mentioned in the Program Description section.