The University of Melbourne
School of Computing and Information Systems
COMP90015 Distributed Systems
Semester 2, 2020

Assignment 2 – Distributed Whiteboard

James Barnes, 820946

1 Problem Context

The problem posed for this assignment is to create a system that acts as a distributed whiteboard. This system must handle various features typical of distributed systems, such as concurrency and failure tolerance.

This system must support multiple users with a single acting as the manager. All users can draw using various tools on the whiteboard, with these drawings shown to all connected users. The manager has extra responsibilities, such as managing connected users and has the ability to save and reload the state of a previous whiteboard instance.

2 System Design

This system can broadly be decomposed into two parts. These parts are the server and client side. There is overlap between these two for various aspects of the system, allowing for good programming practices, such as code re-use.

There is an addictional class named `Fields` which acts as a common source for the various constants used in communcations. This class is absent from diagrams as the inclusion of this class does not add useful information.

Further, various fields and methods, such as getters, setters, and various helper methods have been ommitted from diagrams, as they also do not add useful information.

2.1 Server Side

The server side is composed mostly in three parts, namely the *server*, the *client users* and the *manager* and it's GUI.

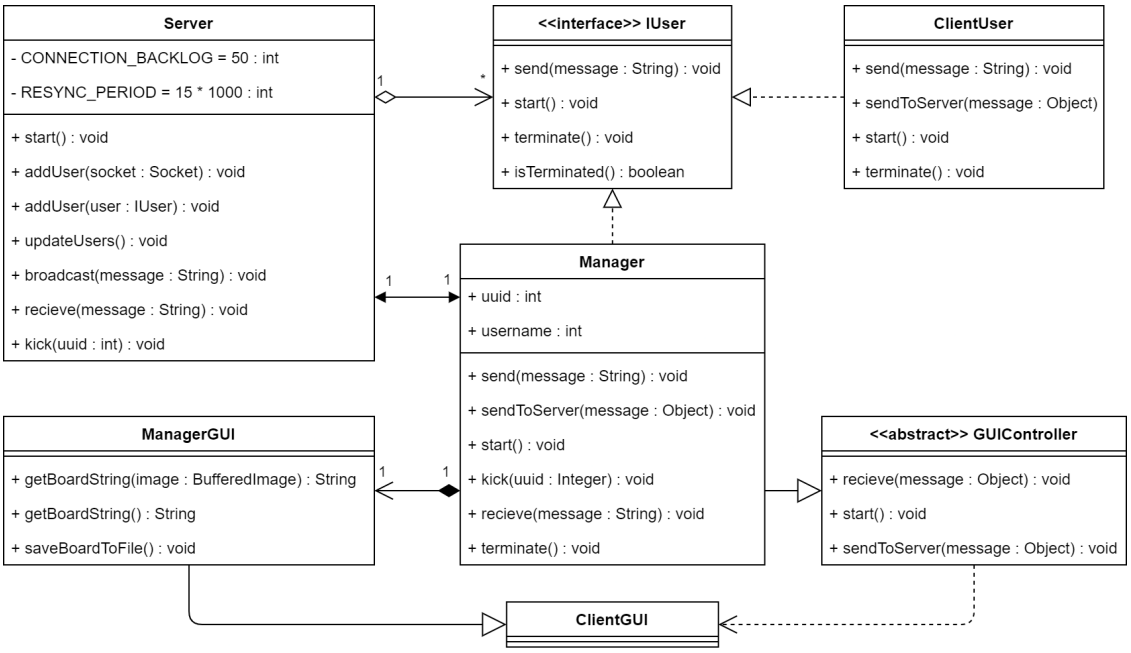

Figure 1: A Class Design Diagram representing the Server portion of this System.

The server's primary purpose is to handle the communication bewtween user's. When a user recieves a message from a client or the manager, the server typically will rebroadcast this to all connected clients and the manager. Further, the server handles new incoming connections by first confirming with the manager to allow the new connection and sending them a message with the current state of the board.

The client user's purpose is to handle the communcation with a client instance. This communcation is performed using TCP sockets, allowing for guaranteed in-order transmission of messages. Any message recieved from the socket is sent to the server to broadcast to all users, and any message from the server is sent to a thread pool (of size 1) that is then sent by the sending thread once idle. The client user also gracefully shuts down the connection with the client when kicked.

The manager is a special type of user. It's communcation is performed directly via method calls with the server as the manager is a local object tied directly to a whiteboard server instance. The manager has it's own GUI, which is an extension of the client GUI (defined in the Client Side section) with extra functions to allow the manager to kick users and save or load the state of the whiteboard. Further, the state of the whiteboard as seen by the manager is considered canonical, and is used by the server if the state of the whiteboard is needed in any instance.

2.2 Client Side

The client side is composed of two components, namely the *client* and the *GUI*.

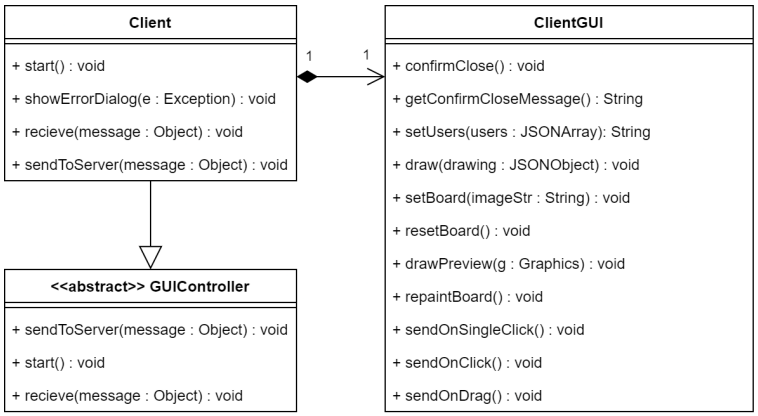

Figure 2: A Class Design Diagram representing the Client portion of this project.

The client portion of the client side is responsible for handling communication with a connected server. As previously mentioned, this communcation is performed via TCP sockets. All messages recieved by the client are forwarded to the GUI and all messages from the GUI are sent to the server via the socket.

The GUI behaves as the manager GUI does, minus the extra functionality included from the manager. This behaviour includes interpretting messages from the server and updating the state of the whiteboard, managing the state of the whiteboard, and packaging messages encapsulating the client user's input to be sent to the server.

2.3 Communication

The client and server sides communcate together using JSON via sockets. Any messages not conforming to the JSON syntax are discarded.

The JSON messages fall into four types, indicated by their `command` field's value.

`board`-type messages denote a message relating to the entire state of the board. If the message contains a `board` field, this field contains a base 64 encoded string containing the state of the whiteboard. If there is no `board` field, this is interpretted as a board reset and the whiteboard should return to a default blank state.

Messages that have the value of `drawing` in the `command` field contain information about a new object to be drawn to the whiteboard, including the drawing's location, type, and colour. When recieved by a client, the information should be passed to the GUI to allow the object to be drawn on the whiteboard. When the server recieves such a message, this should be broadcast to all users, propogating the state change.

Messages with a value of `users` for the `command` field contain a list of user information in the form of an array. This array contained in the `users` field is composed of numerous JSON objects containing each user's corresponding `uuid` and `username`. This message is broadcast whenever there is a change in the list of the currently connected users.

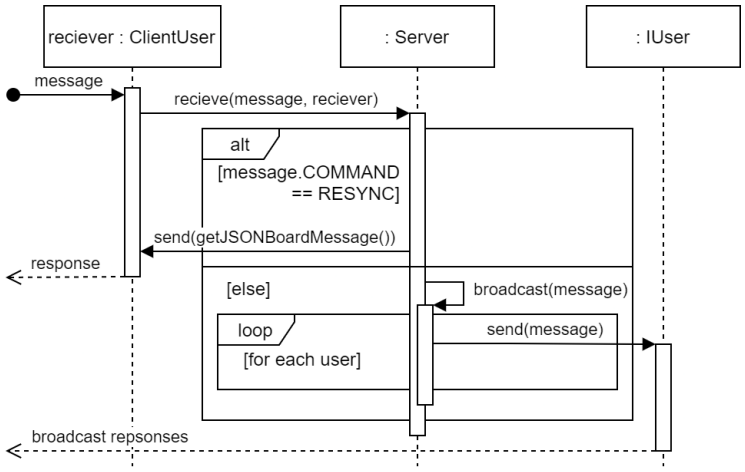

Figure 3: A Sequence Diagram showing the behaviour of the server when recieving a message from a connected client.

When first connecting to a server, the client must send a message with a `username` command. This message should contain a `username` field containing the client's specified username. Failure to do this will terminate the client's user on the server. When this message is recieved, a pop-up in the Manager's GUI shows a message confirming that the new connection should be allowed. If the manager accepts the new connection, the client is then added to a list of clients which causes a rebroadcast of the list of users to all connected users (including the manager) and sends the new user a copy of the state of the board.

Messages marked by the `resync` command denote a request from a client to recieve a copy of the whiteboard's current state. When recieved by a client, these messages shuld be ignored. When recieved by the server, the server should respond with a `board` message containing the current state of the whiteboard and a `users`-message with a list of all connected users.

```
{
  "type":       "object",
  "required":   [ "command" ],
  "properties": {
    "command":  {
      "type": "string",
      "enum": [ "board", "drawing", "users", "username", "resync" ]
    },
    "board":    { "type": "string" },
    "drawing":  { "type": "object" },
    "users":    { "type": "array" },
    "username": { "type": "string" },
  }
}
```

Listing 1: A simplified JSON Schema for the messages.

3 Critical Analysis

There were a few key design choices that had to be made when designing this system.

Firstly, there was a choice of which communication method to use. The chosen method is to send messages encoded in JSON via sockets. This choice was made over alternatives such as Java's RMI. The reason this was chosen over RMI was to allow for more homogeneity in the client-side. As RMI is exclusive to Java, the use of such a technology limits the availability of this system to Java only. With the use of JSON over sockets, clients can be built using any language/platform, provided the message protocols are followed. I believe this is a major drawback of RMI in this use-case as RMI is limited to Java, and as such the use of JSON and sockets is an adavantage.

The choice to use TCP was also made. This insures that messages are sent in in-order and are guaranteed to be delivered if possible. If we choose to use UDP sockets to send messages, extra assurances would be required to ensure that the order of messages is correct (via the implementation of some form of sequence number) and to ensure that messages are sent to all users or to the server. These two properties are necessary, as the order in which messages are recieves matters for drawing on the board, and their guaranteed delivery is necessary as lost messages can lead to inconsistent state amongst users. RMI also makes use of TCP, further justifying the usage of TCP.

Further, to ensure that race conditions are dealt with adequately that arise from multiple users trying to draw on the whiteboard at the same time, the server utilises a single-thread pool per client to write each message to respective clients. This ensures that the order of broadcasts to all users is consistent, however this does lend preference to users with a faster connection to the server as their messges arrive first. There is no way to remove this preference and I feel that this choice is reasonable as it simplifies the broadcast process. The use of the thread pool also reduces thread-churn, as a thread-per-request architecture in this case would cause such behavious as messages are frequently sent to the server by all clients.

The state of the whiteboard is saved in a special `.wb` file. This file is a PNG unde the hood. A raw PNG could be used instead to save the state of the whtieboard, however the choice was made to utilise a different file format. This choice was made as it abstracts the true format of the file, hiding unnecessary details from the user.

Error messages are generated for the user in two forms. Depending on when the error occurs or on the severity of the error, the error message will display as either a pop-up or is printed to the console. The errors that display as a pop-up are related to runtime errors, such as a `.wb` file being incorrectly read, and are displayed this way because they are directly relevant to the user's experience. Other errors printed to the console are printed to the console for two reasons. Firstly, there are errors printed before the GUI is showing, thus no pop-up could be displayed. Also, there are some messages that are not relevant to the user, such as parse errors with JSON. This second kind of message is not directly relevant to the user, and thus showing a pop-up is not useful to the user.

4 Additional Features

All additional features outlined in the specification have been implemented. These features are the manager's ability to save or load the whiteboard's state and the ability for the manager to kick other users.

Following the original specification, extra tools such as the freehand tool and the eraser have been implemented. An extra tool, a color picker, which allows the user to choose a colour on the whiteboard is also implemented. Finally, controls to set the size of the text, freehand, eraser, and line tools have been implemented.

Aditionally, there is a Resync item in the File menu. If a client is to click this item, the server will send a copy of the board to the client. If the manager clicks this button, the server will broadcast a copy of the board to all connected users. This allows any user at any time to ensure that they (or in the case of the manager, that all) share consistent state, however this use is optional and not required in my testing to allow for consistent state.