Contents

[Project Report 1](#_Toc34568979)

[System Design 1](#_Toc34568980)

[Client 2](#_Toc34568981)

[Server 4](#_Toc34568982)

[Communication Protocol 5](#_Toc34568983)

[Database 6](#_Toc34568984)

[Test Plan 7](#_Toc34568985)

# Project Report

## System Design

When designing our project, thereb were two mainmodels we considered using. The first was a peer-to-peer model in which clients directly communicate with one another.

Suitable for multiplayer game

Less latency between players

No reliant on a single point of failure(server)

Cons

Unknown connections

No central database for login

Potential user bandwidth issues.

Due to the issues presented by using this model, we instead decided to opt for a client-server model instead. This had a number of benefits such as providing a single point of connection: Every client only needs to know the address of the server for multiple users to communicate with one another.

Secondly is provides a centralization of data such as login credentials. Even with a peer-to-peer service, login details must be stored in a centralized location, as a result some degree of server would be required regardless. By having the server access the database rather than letting the clients access it themselves, there is also some degree of security from rogue clients making unauthorised entries and access. Additionally, it makes the clients more lightweight as they only need to provide simpler UI tasks rather than containing all the game logic. Additionally, the bandwidth requirements for the clients may be drastically reduces, as now rather than sending all the data pertaining to images and text to every single other player in a game, it only has to be sent to a single point, the server, which handles passing out the information to everyone else.

This model does have a number of downsides for our project as well: Having a central server means that there is a single point of failure, if the server was to crash or be otherwise offline, no clients would be able to connect and play a game. Additionally, a server will have to potentially handle a large volume of incoming and outgoing data at the same time and could potentially become a bottleneck alongside the overhead of first having to send the information to the server before it reaches other clients.

Ultimately we decided that, despite the downsides, it would be better for us to use the client-server model as it would be quicker to implement in the limited time we had to complete the project due to the complexities that would be required in building a peer-to-peer project.

### Client

The client in our system is built out of multiple classes using the Model-View-Controller design paradigm. The Client class is the entry point into the program for the client and acts as the model. This class starts the JavaFX application thread to initiate the user interface components and contains the methods required to open a connection to and send communications to the servers. The Client class also has a limited amount of capability to receive incoming communications from the server, largely when a response is expected from the server after sending information such as confirmation after sending a log-in request.

Prior to joining a game room, a client acts in a purely call and response manner, only reading from its sockets input when it is expecting something. After the client joins a game, messages from the server at any point in time, such as chat messages from other players and drawing information. As the client needs to perform its own functionality at the same time, it cannot continuously attempt to read from its inputs as it will enter a blocked state and become unresponsive.

Due to this, the Client class is accompanied by the ClientListener class. When a game is entered, a ClientListener object is instantiated. This listener will continuously attempt to read from the input stream and, depending on what message is received, enter a switch case using the header of the message to decide what action to take, usually calling a method within the Client class. This ClientListener is run in its own thread so that while it is in a blocked state, the Client class can still function itself without being blocked. If a user exits a game room, the ClientListener is terminated and the Client goes back to using a call and response methodology to communicate with a server.

The View components of the client consist of FXML files generated by Scene Builder and additionally edited by hand. Scene Builder was used because it allowed simple GUI components to be put together to create a functional program quickly and efficiently. In order to increase the aesthetic quality of the program, CSS was used to expand upon the basic GUI to improve the look and feel.

The Controller component of the program is made up of multiple Controller classes. Each GUI scene as generated by an FXML file has its own controller that deals with user interactions with all the nodes and scene objects and contains methods that allow these components interact with the model and update the view when required.

The LoginController and CreateAccountController both take text field inputs and send them via the Client model to the server when the user has finished filling them out and when a confirmation or rejection response is received, will transition to a new scene or display a message to the user warning of errors.

After a successful login the scene transitions to the Home scene. This scene is very simplistic and only displays the rooms available on the server and a brief text description of the game and its rules.

When a user clicks a room, it transitions into the GameRoom scene if a positive confirmation is received by the server.

At this point the ClientListener thread is started which when receiving messages, will largely trigger methods in this controller. In order to exit a room, the ClientListener must be stopped or the call and response methodology will be disturbed as the ClientListener will intercept incoming messages from the server. As the ClientListener exists in a blocked state, using an interrupt on the thread does not work when it is blocked on a Socket. As a result, the ClientListener can only be ended in one of two ways: Firstly: The client completely closes the program, and the socket is closed, ending the ClientListener through handing the exception that is thrown. Secondly, if the client just wants to go back to the home screen, a message is sent to the server confirming the client wants to leave the room. The server then sends a message back to the ClientListener which triggers a flag to be set. When the next read loop starts, this flag is first checked, and if it has been triggered, the ClientListener breaks out of the reading loop and ends gracefully. A new ClientListener is then created in the Client class to be started when next time a game room is joined. This is done so that the input does not have to be closed, resulting in the socket being closed to the server, ultimately resulting in the client having to re-log in.

The GameRoom displays a list of all users present in the current game instance and is updated by the server every time a player joins or leaves a game without being requested by the client.

The GameRoom also contains a text area and field which acts as both the input in which to make a guess of the drawing, and to doubly act as a chat window so players can talk with one-another. When a client enters a message in the TextField, it is sent to the server. The server then distributes this to every player in the room which is then shows on their screen via the controller including the original sender. This way all players receive text messages in the same order, whereas if the sending player displays text client side for themselves, it may appear earlier in their own chat than for other players. This also allows the parsing of all text inputs by the server for guesses before being siplayed.

The largest feature of the GameRoom is the canvas. The canvas allows users to draw an image freehand with the mouse or devices like graphics tablets. This is achieved through several methods such as setting three mouse events for the mouse being pressed, dragged and released.

When the mouse is pressed, a new ArrayList of coordinates is created, and the first coordinate, the location of the mouse at the time of the mouse being clicked is added.

When the mouse is then dragged, the coordinates of the mouse are continuously added to the ArrayList. In addition to this the end of the path so far is drawn onto the canvas via the GraphicsController for the canvas with a set colour and width and additionally sent to the server to display for other users. The finally when the mouse is released, the path is again drawn and sent to the server so that single clicks without the mouse being dragged are also displayed on the canvas.

The colour and width of the line to be drawn from the list of coordinates is prescribed by the user setting the colour in the ColourPicker and the size via a slider which is provided by JavaFX which is also packaged in the message sent to the server. Because the GraphicsController can draw lines between a list of coordinates, that means that incoming lists of coordinates from the server can also be drawn onto the user’s screen. Unlike chat messages which are first sent to the server before being displayed, for the drawer, the image is instantly displayed on their own screen. This is because many drawing messages are being sent to and from the server resulting in a slight latency between when the user draws on their own screen and the image appearing for the other users, making accurate drawing difficult if the image was only drawn from server messages.

Because the game requires the users to take turns drawing, the user is only allowed to draw after the server has sent them a message that unlocks the drawing features. Likewise, after their turn is over, this must be again disabled again from a signal from the server. This is achieved by having methods which can disable and enable the mouse events associated with drawing and by hiding the drawing elements like the colour picker and size slider. These methods also disable the chat input for the drawer so they can not make guesses on their own drawing and stop them from giving hints in chat.

### Server

The server is also made up of multiple components used to effectively deal with multiple clients simultaneously.

The Server class itself has very simple functionality: Firstly, it creates a ServerSocket to accept connections from clients continuously. When a client connection is accepted, that client is given its own thread so the Server class can go back to accepting more clients.

Secondly it simply tracks the users that are currently connected to the server and the game rooms running on the server that are created when the server begins.

Finally, it allows clients to request information about the rooms that are running on the server.

When a client connects to the Server class, a new ServerThread class is created. This class acts as the client’s interface with the server. Because this class extends thread, multiple ServerThreads can run concurrently allowing multiple clients to make requests simultaneously. This ServerThread acts somewhat similarly to the ClientListener class in the client. It contains a loop that attempts to read from the input stream coming from its designated client. When a message is received, a decision is made based on the header of the message using a large switch case statement and, if applicable, sends responses back to the client. Its methods can also be called by the server itself so information can be sent to the client without specifically being requested such as in the case of receiving a new chat message that needs to be displayed.

The server also has another concurrency requirement in the form of game rooms. In our game, we had the requirement that multiple instances of a game can exist while being played by different players at the same time. As a result, the Room class was created to run on its own thread. The room class has several key responsibilities. Firstly, it contains all the logic required for the game to functions. Secondly it tracks users currently attached to it, so that messages the server receives from a user can be dispersed to all other users in the same room. Because these run on their own threads, each room can communicate with different users simultaneously.

The Room class exists from when the server is started. When a client connects to the server it is given its own ServerThread. From here a client can request to join any of the rooms available on the server. If a Room accepts a client, the room is passed the reference to the clients ServerThread, and the clients ServerThread is passed the room reference. From this point until each dereference the other, a message from a client can be passed to the room they are attached to. This message can then be passed from the room to every other client also attached to the room, effectively allowing communication between the clients.

### Communication Protocol

When deciding our communication protocol, we considered using a standard existing format such as XML or JSON to send information between client and server. Due to the requirement of sending drawing data over the network, we decided that these were unsuitable because of the difficulty of representing our drawing data using these formats which would require arrays of indeterminate size and more complicated objects than just primitive data types. As a result, we created our own messaging protocol.

Our protocol involves sending a serializable message object between the client and server using the ObjectInputStream and ObjectOutputStream classes.

The basic Message class is made up of two parts: First is the Command header. Command is an Enum type that consists of a list of commands that the server or client can carry out. Each message contains a single Command, and when the incoming object is read by either the ServerThread or ClientListener, the first part to be checked is this Command. Using this, the reading class enters a switch case and decides what action to take based on the Command in the Message.

Secondly, the Message constructor allows for a variable number of string parameters to be given. The result is that a Message may contains just a simple command with no other information, or additional information that is required for the command to act on, for instance when the LOGIN command is given, the ServerThread expects the login details associated with the login request to also be present in the message. The ultimate result is that the ServerThread and ClientListener know when to expect data associated with a certain command through their switch case statements, when data is not required, it is ignored if it is even present at all.

When sending part of a drawing between server and client, an extension of this Message class is used: the MessagePath class. Like Message, MessagePath contains a command header to identify the contents, but instead of taking strings as parameters, it instead takes a single Path object. A Path object represents a line that has been drawn by a user and is made up of a colour string, a brush size int and an ArrayList of Coordinates that makes up the location and shape of the line on the canvas. After this path object is sent to the server, it can be sent to other clients in the game room and also be temporarily saved so that joining users in the middle of a drawing can be sent the entirety of the drawing rather than just what is sent to the room after they have joined. When a client receives a MessagePath, the Path object can be read, and its variables used to redraw the image on their canvas.

Beyond this, minimal other communication is used between server and client. The only other communication is the occasional primitive Boolean type using the DataInput/OutputStream which is only used as a confirmation for things such as confirming logins and room joins while the client is still in the call and response mode of communication where a full Message is not required.

### Database

The database acts as persistent storage of user’s login credentials. The database is accessed through the DatabaseManager class which is instantiated by the Server class when it starts. This class contains methods that allows the server to check if a given login is valid and the addition of new accounts.

## Test Plan

Our testing is made up of two methodologies. Where possible we have used JUnit testing to test the functionality of methods under a variety of different scenarios to ensure they are functioning as expected automatically.

Due to a significant portion of our project being based on client-server communications and GUI functionality, it is not possible to conduct such tests. As a result we decided to conduct manual user testing to identify bugs in the programs functionality as well as GUI errors by trying to perform as many possible scenarios that a user could encounter while using our program and documenting the expected and actual results.

Evaluation

The good

The bad

What future improvements could be made

That we should have done differently

Path drawing versions