Team ACES

COMP90015

Distributed Shared White Board

Xu Yang 961717

Yizhou Zhu 1034676

Haonan Chen 930614

Chen Xu 945756

Table of Contents

1. Introduction

This project designs and implements a distributed shared whiteboard. Multiple users can draw lines or shapes on the whiteboard and chat with each other via the "chatroom" feature. It allows concurrent users to select shapes, draw a picture and send message to other users. It also assigns some privileges to the owner of the shared whiteboard, who is the first user entering the room and can kick other users out of the room.

In the implementation, there are some challenges on the distributed system, including handling concurrency and maintaining data integrity, dealing with the network communication, and implementing the user interface for the system. We solved these problems and deepened our understanding of the distributed system. The whole system is developed with Java programming language. The protocol of networking communication is Transmission Control Protocol (TCP). The main architecture of the system is multi-threaded client-server architecture, while a middleware using Publish-Subscribe system is implemented to provide a layer of indirection, decoupling the server end and client end.

This report will give an introduction of the rules of shared whiteboard and explain the implementation details such as system architecture, technology stack and messaging protocol. Finally, a conclusion will be drawn regarding the challenges of the distributed system.

2. Technology

This section explains what technologies constitute the shared whiteboard system.

2.1 Java

The shared whiteboard distributed system codes are written with Java programming language, one of the most popular programming languages for the client-server application. Java programming language that introduces the idea of Object-Oriented Programming (OOP), a paradigm that makes use of the encapsulation, polymorphism, and inheritance in
writing programming codes. In this project, the system is developed based on the Java Development Kit version 1.8.

2.2 Java Swing and Java AWT

Swing is a set of Graphical User Interface (GUI) toolkit for Java. It was developed to provide sophisticated GUI components. In this project, the client UI and server UI are developed by Java Swing. To control the events on the UI, such as drawing a shape using mouse, Java Abstract Window Toolkit (AWT) is used. Java AWT can bind the events of users to corresponding GUI. Also, the shapes used in this system, such as rectangle or circle, are defined in Java AWT.

2.3 Socket

The shared whiteboard distributed system is developed with socket programming which refers to writing an application that can be executed across multiple computers that connected each other. Each client has its own socket when connecting to the server. The IP address and port number of client sockets should match with those of the server socket. The server assigns one thread for each client connection. When the client is connected with the server, it can continue to send data to the server, as long as the connection is not closed. For each client, it can send multiple requests through the connection.

2.4 Exchange Protocol

While there are two communication protocols that can be used for socket programming,
which are Transmission Control Protocol TCP and User Datagram Protocol (UDP), The shared whiteboard distributed system uses TCP as the communication protocol because it provides more reliable, ordered and error-checked delivery of a stream of data between clients and the server via the internet protocol (IP) network. TCP provides sequence number in its header to maintain the order of data delivered. It also provides mechanisms such as slow start, congestion avoidance, fast retransmit, fast recovery and flow control to guarantee the stability of data transmission. These advantages are

what the system needs, the clients need to know whether their operations are received by the server in time and in correct order, while the server needs to update the shared graph in real time.

2.5 Thread Pool

Since each client has a connection to the server and should be treated as an individual thread, the server needs a mechanism to deal with the multi-thread situation. Moreover, when a large number of clients connect to the server, the CPU resources of the server might not be able to sustain these client threads (for each client, a thread is assigned to deal with the request of client). Also, the creation and destroy of threads waste a lot of computational resources, the reuse of threads must be considered. To equip this system with stability to handle the client requests, we decide to use FixedThreadPool in java to limit the number of connections to the single server. We defined an integer variable "poolLimited", which can be set by the Server UI. When the number of connections exceeds this limit, the new connection will be blocked and put into a LinkedBlockingQueue of the thread pool. Once the number of connections is below this limit, a new connection is retrieved from LinkedBlockingQueue and assigned a thread from the thread pool. The threads in thread pool can be reused, saving the time of creation and destroy of threads.

2.6 Message Queue

We define a variable "maxNum" to indicate the max number of clients that can use the same shared whiteboard. When the number of clients in the whiteboard reaches the limit, the new clients who want to enter the shared whiteboard will be put into a waiting list, which is implemented by a LinkedBlockingQueue. When the client is put into the waiting list, a message will be shown in the client UI and the thread of client UI is blocked, from here, the client can choose to wait until there is space available or close the application. When there is some space available, a client socket will be put into the concurrent HashMap of the whiteboard to indicate the client has entered the room. Then, user can draw shapes from the client UI. One disadvantage of this implementation is that, the fairness of clients cannot be guaranteed. The client that enters the waiting list late may be allowed to enter the room first. This deficit comes from the features of LinkedBlockingQueue. We can use ArrayBlockingQueue to guarantee the fairness of clients, but ArrayBlockingQueue must have a limited boundary. In this case, we assume that there are infinitely many clients wanting to join into the same whiteboard. Therefore, LinkedBlockingQueue is more suitable since it is unbounded.

2.7 Pessimistic offline lock

In the situation of concurrency, we use Pessimistic offline lock to allow only one transaction at a time to operate on the data. In this way, conflicts can be avoided. Pessimistic offline lock is implemented by "synchronized" keyword in Java. For example, we use "synchronized" keyword when checking whether the whiteboard has a manager already. The "synchronized" keyword only allows one thread to check or set the manager at a time. The synchronization process guarantees the data integrity and avoids conflicts. However, since the "synchronized" keyword requires context switch between kernel mode and user mode, it demands a lot of CPU resources and may increase the load of the server. Under this circumstance, we try our best to narrow down the scope of "synchronized" keyword in our codes.

2.8 Encryption

The transmission of data between server and clients should be safe, otherwise, the tamper on data may happen and the consistency of data may be compromised. To maintain the safety of data in transmission, we add an EncryptDecrypt class to encrypt or decrypt the data. We use AES symmetric encryption algorithm to encrypt the data. The shared secret key is generated by a shared secret message defined in the system. Although the encryption and decryption process may slow down the speed of data transmission, it prevents the data from tampering and only allows entities within the system to communicate with each other.

2.9 Serialization

To transfer the shapes objects drawled by clients to the server, we firstly serialized the shape object into a byte array (Base64), then encode the bytes array into a string, which is put into the JSON for transmission. To recovery the shape object, the server needs to decode the string to a byte array and then deserialize it to a shape object. Actually, the serialized stream can be encrypted, authenticated and compressed, supporting the needs of secure Java computing.

2.10 Concurrent HashMap

In multi-thread situation, it is important to maintain the consistency of data. In PublishSubscribeSystem, we use Concurrent HashMap to store the information of clients sharing the whiteboard and the information of applicants applying to enter the room. Concurrent HashMap uses "Node Lock" to enforce the synchronization of data. When one thread is updating the data in Concurrent HashMap, another thread cannot update the data. Concurrent HashMap is thread safe, while HashMap is not thread safe. Moreover, when one thread is iterating over the Concurrent HashMap and another thread makes some changes on the data, there will be no exception to throw, since Concurrent HashMap is Fail Safe. However, HashMap will throw an exception and crash the system since it is Fail Fast.

2.11 Message Format

The message format between the client-server is using simple JSON. The data is stored in JSON as key-value pair. The JSON object is encrypted and serialized for transmission between client and server.

3. System Architecture

The system architecture for the shared whiteboard distributed system is described in this section.

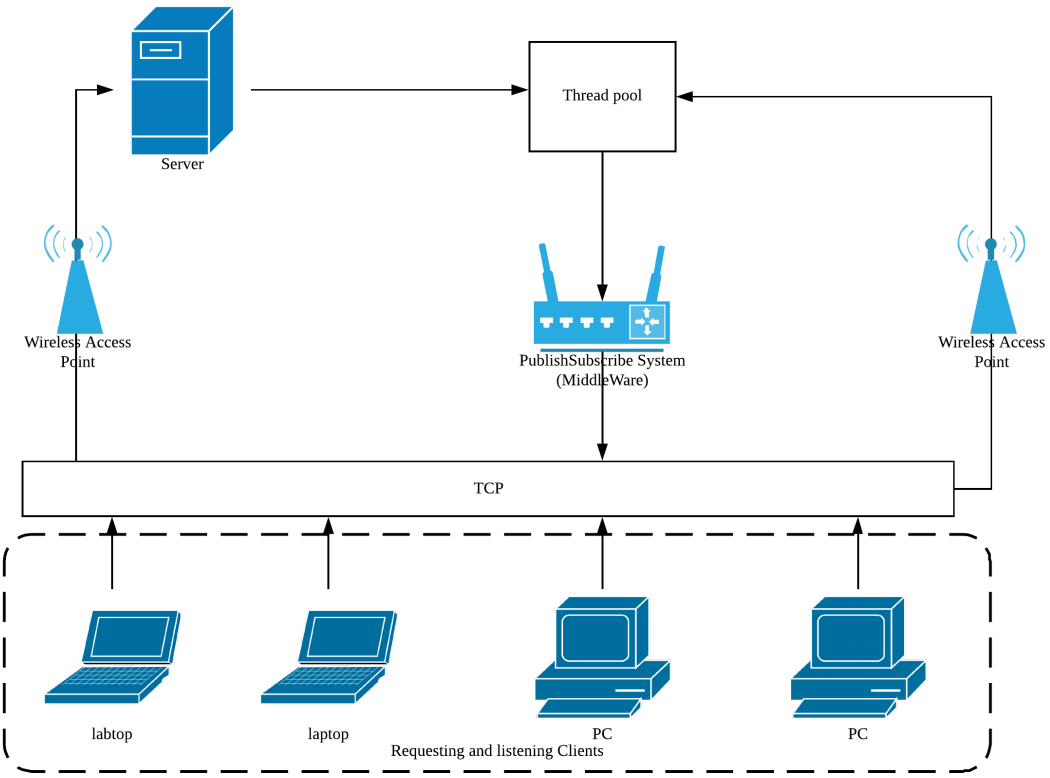

Figure 1 General Architecture Diagram of system

3.1 Client-Server Architecture

The distributed shared whiteboard uses a client-server architecture which means that there is a server that provides the multi-threaded concurrent connection from the unlimited numbers of computer clients. However, the maximum number of clients is restricted by the server capacity and controlled by the FixedThreadPool. Having a centralized server lowers the difficulty of programming. We don't need to consider the issue of load balancing, which is important to decentralized server situation. Also, centralized server makes it convenient to add or remove any client from the system. In this way, the whole system is scalable.

3.2 Publish-Subscribe System

Publish–Subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead uses middleware to encapsulate published messages without knowledge of which subscribers. This pattern provides a layer of indirection between the single server and a large number of clients. In this way, server is loosely coupled to clients and need not know of their existence. The middleware keeps track of the existence of clients and their requests. Publish–Subscribe pattern also provides the opportunity for better scalability than traditional client-server architecture.

For the implementation, we establish a middleware class named PublishSubscribeSystem and we use **Singleton Pattern** in this class. In this way, both server and clients can call the methods of middleware to realize the message passing functionalities. For the getInstance() method of the middleware, we use **Double-Checked Locking Pattern** to make sure the operation of getting singleton instance of middleware is thread safe and cache coherent.

4. Implementation

4.1 The Shared White Board

4.1.1 Share the shapes

The distributed Shared White Board is a live-stream collaborative drawing tool that several participants, at least one person to make the system runnable, can draw and chat under the same channel. The latest change to the shared white board can be viewed to all the current users.

4.1.2 Draw

The system allows the free line drawing as well as the rectangle, straight line, circle, ovals. A text-based images will be put on the canvas if the user wants to display text instead of some shapes in the selected area. All the changes should be merged to all the other canvas belonged to other users instantly. Colour can be selected, and the size of the shapes are also variable.

4.1.3 Chat room

A chat room is also available for the participants to exchange information. A list of the chat history is displayed with all the messages attached with the time sent and its username.

4.1.4 Create the board or enter the board

The first one to connect to a running server can only create a white board and become the manager of the board. All the other requester will be blocked and only have the access to enter the shared board. If the user is not the first one who also created the white board on the server, he will receive all the drawings from the current session cached on the server, but he cannot access the chat history. We also developed a waiting-list mechanism, please refer to the innovation part.

4.1.5 The manager

The manager is not only an ordinary participant but also the administrator of the board who can have more functions than the ordinary user. He can remove anyone other than himself in the user list while the removed one will lose the contact with the shared white board. He can also refresh the white board to an all-clear state or withdraw the drawing from last step or upload the saved picture from the local

directory and broadcast the loaded one to all the users. As long as the manager left the board, all the other users will be forced to quit, and the server will wipe out all the records.

4.1.6 The user

The user who entered the board can have all the basic functions like drawing and chatting. The user can also save the picture on the board to the local directory but no access to upload it. When they leave the board, all the user will receive a message notification. Namely, all the online user will receive a notification about the new user on the board.

4.2 Class Diagram

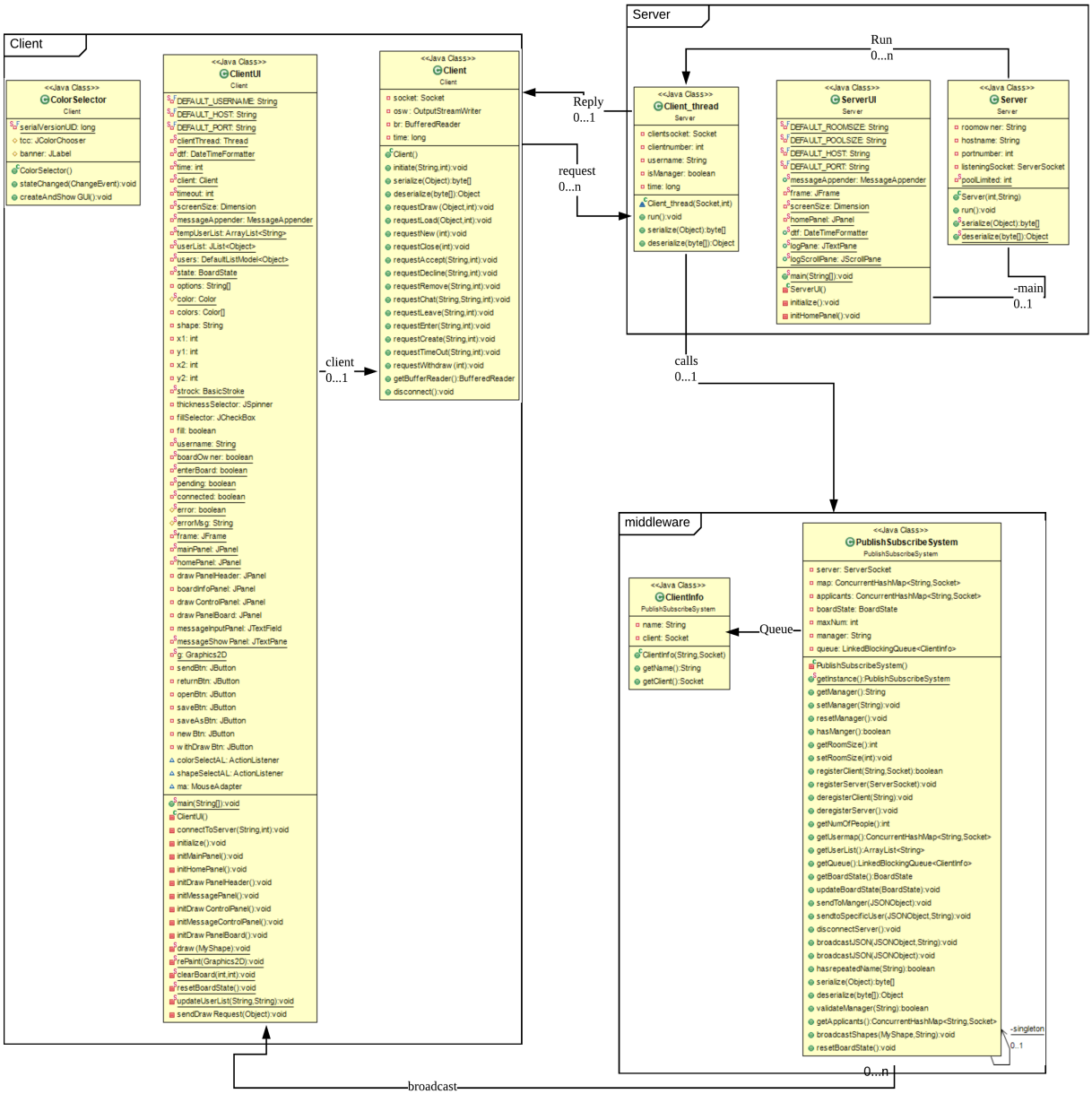

Figure 2 Design Class Diagram for the Client and Server

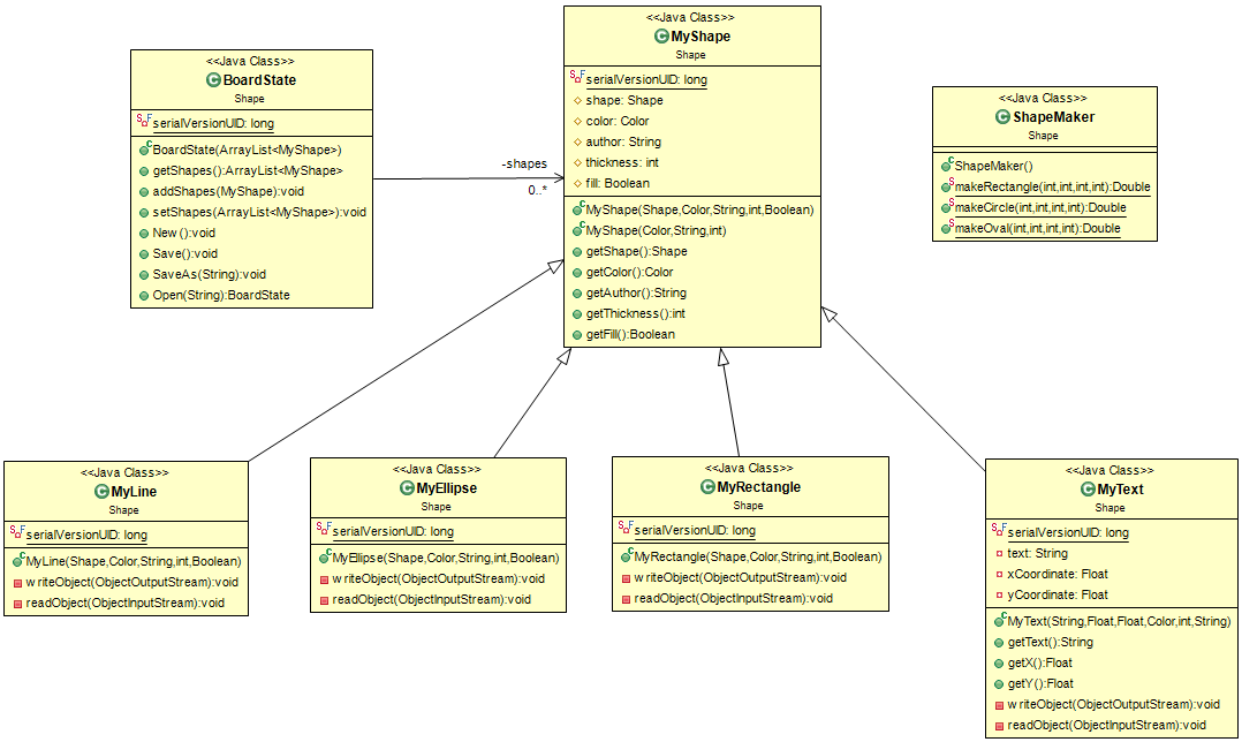

Figure 3 Design Class Diagram for self-designed Class

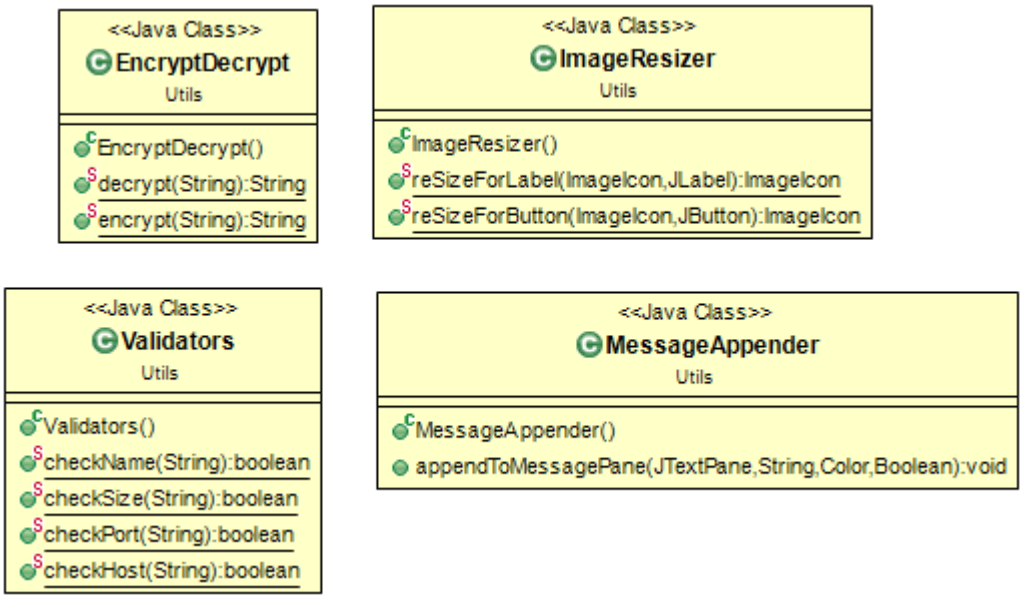

Figure 4 Design Class Diagram for self-designed helper classes

Number	Class	Description
1	Client	The main skeleton for the clientUI. It defines multiple methods to send request to the Server
2	ClientUI	The runnable class that will create the GUI for the user. It can complete most of the task described in the requirement and listening the incoming message for change.
3	ColorSelector	Class to enable the colour selection in drawing
4	PublishSubscribeSystem	The middle ware to manage the client and the connected socket. Using this class to control the client number and broadcast the necessary information to the client terminal
5	ClientInfo	A class for the participant in the queue. Record the username and the attached socket
6	Server	The main skeleton for the server. Only worked as the listening thread to accept the incoming connection request and start a thread for a new client
7	Client_thread	The main frame to accept the request from the client and perform the correspondent work. One client only has one thread to handle it and close when the client is gone.
8	ServerUI	Runnable class to create a Server UI. It can display the Server received request and exception handling information. The room size and server parameter such as IP and Port number is also settable.
9	Shape	Contains Several child classes. It is the main class to describe what is drawn on the board and the transferable to the Server for broadcasting.
10	BoardState	The class contains a list of shapes. The clientUI relies on this class to draw and demonstrate.
11	EncryptDecrypt	Method to encrypt and decrypt the exchanging request and acknowledgement
12	Validators	Check whether the input is valid syntax or not

Table 1 Class Description of Shared Whiteboard System

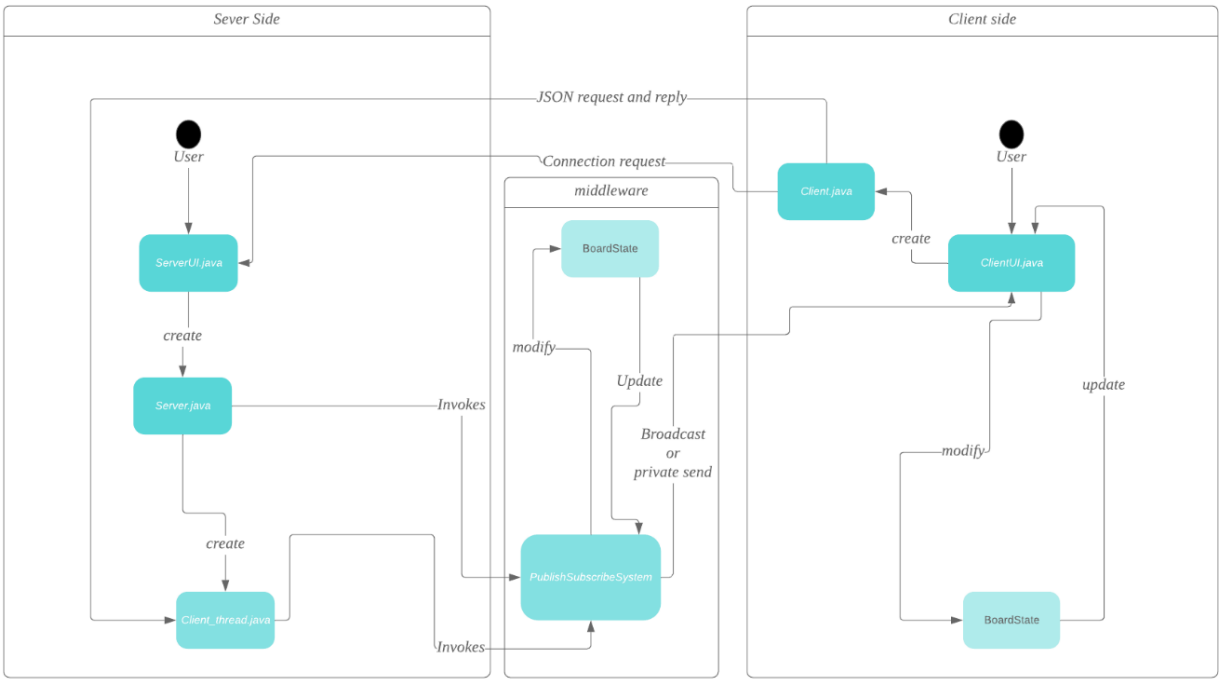

Figure 5 Interaction between some important classes

Brief on the basic operation for Server and Client:

Server:

1. Start the ServerUI using command line
2. Set the room size, IP address and Port number
3. Start the Server or adjust the parameters in step2
4. Listening for the incoming connection
5. (optional) disconnect the Server

Client:

1. Start the ClientUI using the command line
2. Set the IP address and Port number
3. Create the Board or enter the room if board already existed
4. Conduct some operations or disconnect.

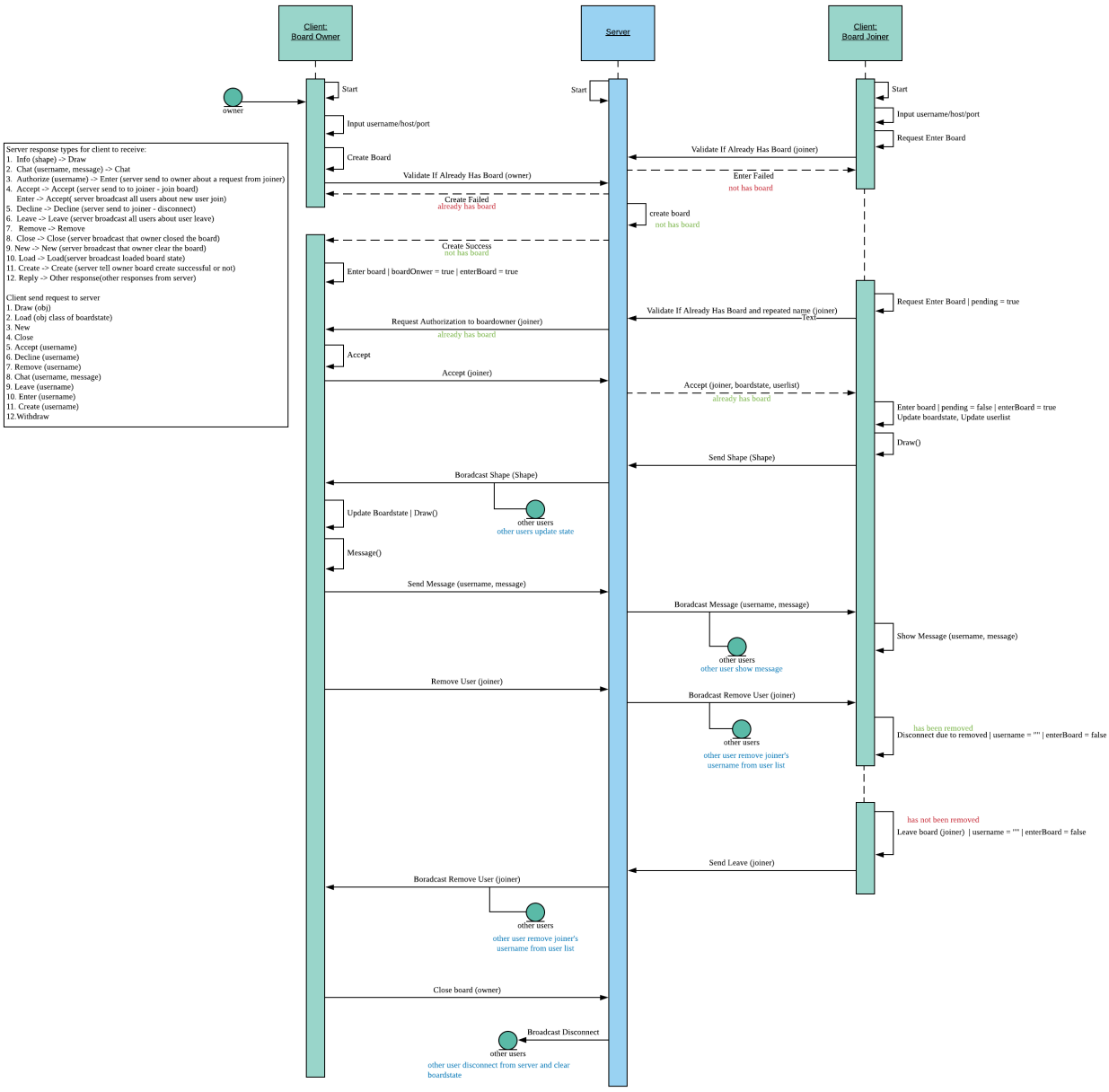

Figure 6 Sequence Diagram of the interaction between the client and server

4.3 JSON object introduction

Here are the JSON objects transferred from client to server with the follow result that will occur.

The majority of the JSON transferred will be:
{Source: Client, Goal: Draw, ObjectString: (String of shape drawled), username: XXX}

The server will add the new shape into the local board state maintained on middleware (PublishSubscribeSystem) and Broadcast the new shape from XXX to all the users using JSON:
{Source: Server, Goal: Draw, ObjectString: (String of shape drawled)}

The clientUI of each receiver will read the JSON and extract the shape and add to the boardstate it maintained by itself to demonstrate it on the canvas. Meanwhile, the Serve will generate an acknowledgement using JSON:

{Source: Server, Goal: Reply, ObjectString: Successfully received!}
The following discussion is based on the JSON sent from client to server, the generic type will be **{Source: Client, Goal: XXX, ……}**

The discussion will be based on different value from the key Goal:

4.3.1. Create

After clicking the create button, the client will send a {create} JSON to the Server
{Source: Client, Goal: Create, Username: XXX}
The server will receive the request and start a thread, the thread will get named after the username extracted.
The server will check whether the middleware has a registered manager, if there is one, the Server will reply the request with **{Source: Server, Goal: Failure}**
Otherwise, it will be **{Source: Server, Goal: Success}**
The Server will register the user as the manager. After receiving the {success} JSON, the clientUI will replace the login interface with the drawable interface.

4.3.2. Enter

After clicking the "Enter" button, the applicant will send a JSON to the Server
{Source: Client, Goal: Enter, Username: XXX}
The server will check whether there is a manager on server or there is a repeated name in all of the name list.
If there is a repeated name, the replay will be **{Source: Server, Goal: Reply, ObjectString, repeated name}**
If there is no manager, it will be **{Source: Server, Goal: Reply, ObjectString, No board yet, try to create one}**
Otherwise the server will send a request to the manager for "Authorize"
{Source: Server, Goal: Authorize, ObjectString: need to authorize the applicant, Username: XXX}. XXX is the name of the applicant.

The manager will get a popped window to choose to accept or decline the request. It will generate two similar but different JSON; **{Source: Client, Goal: Accept, Username: XXX}**
Or **{Source: Client, Goal: Decline: Username: XXX}**

The Server will then forward the result to the client. If the client gets declined, he will receive the result and fail to load the canvas. Otherwise, the Serve will check if there is a vacancy in the room. If there is, the server will update the local user list, the forwarded JSON will be
{Source: Server, Goal: Accept, BoardState: (BoardState cached on Server), UserList: (UserList cached on Server), Status: In_room}

After receiving the result, the clientUI will load the BoardState and user list from the JSON and enable the Canvas. A JSON will be broadcasted to all of the users **{Source: Server, Goal: Enter, username: XXX}** The user receives the "Enter" from the Server will update the userList with XXX.

If there is no vacancy in the room, the forwarded JSON will be **{Source: Server, Goal: Accept, Status: In_queue}**. The clientUI will pop a window to show the client is in a queue and the timeout counter will start.

If there is a vacancy emerged before the timeout, the server will broadcast the "Enter" JSON to update the clients' user list and send the JSON
{Source: Server, Goal: Accept, BoardState: (BoardState cached on Server), UserList: (UserList cached on Server), Status: In_room}
to the applicant in queue. It will enter the canvas as usual.

Otherwise, the applicant will send a JSON to mark the timeout **{Source: Client, Goal: Timeout, username, XXX}.** The server will automatically remove the applicants name in the queue.

4.3.3. Remove, Leave and Close

When the server received "Remove" **{Source: Client, Goal, Remove, Username, XXX},**
the server will retrieve the socket of the XXX and forward the JSON to the client privately. It will disable all the functions of the canvas of XXX. The Server will also broadcast to all
{Source: Server, Goal: Leave, Username: XXX}.
It will wipe the name of XXX on all the clients' user list.

The same logic works on "Leave". If the server received **{Source: Client, Goal: Leave, Username, XXX}**, the Server will broadcast it to all the users and the clientUI will wipe out the name of XXX.

Both conditions mentioned above will make the Server check the vacancies in the list and send Enter to all the users while a "Accept" to the applicant in the queue.

If the server received a **{Source: Client, Goal: Close}.** It represents the leave of a manger; it will forward the JSON to all the users and reset the server. All the canvas connected will get disabled.

4.3.4. Load and New

If the Server received a **{Source: Client, Goal: New},** the server will first clear the local boardstate and forward this (with Source changed to Server) to all the users with a
{Source: Server, Goal: Chat, messge: the boardowner cleared the board!, username: BoardOwner}.
The receiver will refresh the canvas and get a message append in the chat room called "**the boardowner cleared the board!**"

The similar logic applies on **{Source: Client, Goal: Load, Objectstring: (boardState)}**
The receiver of the forwarded message will load the boardstate into the canvas to refresh and send a message notification.

4.3.5. Withdraw

If the server received **{Source: Client, Goal: Withdraw},** the server will extract the local boardstate and remove the last added shapes in the boardState and forward JSON
{Source: Client, Goal: Load, Objectstring:(newboardState)}
The receiver will replace the boardstate with the newboardState, which is kindly a rollback of the last pictures.

5.Innovation implemented

5.1 The Double Encryption

There is double encryption to protect the secret of the messaging in the flow. The first encryption occurs in the drawing request. The shapes transferred in the process is encrypted with the basic base64 encoding process to generate an equivalent string. It is not an understandable string. All the acknowledgement and request will be encrypted for the second time with a pre-shared private key embedded in both the client and Server. Without the embedded key string, the intruder can hardly decode the string to get the correct JSON object, while a minor tamper on the string will fail the authentication process on each side.

5.2 The Waiting Queue

To improve the performance of the board on the communication and decrease the load of the server, the team proposed a blocking queue in the Publish Subscribe system. When the Manager accepts the entering request from the client, the middleware will check the capacity of the room. If the acceptance over-crowds the capacity of room, the accepted client will temporarily be contained in a waiting list. If anyone in the room leaves or get removed by the manager, the user in the waiting list will get the authority to enter the room and draw on the whiteboard.

5.3 The Withdraw Feature

It is the unique feature for the manager. As long as some unacceptable events occur on the board, the manager can withdraw what is last drawled on the board. For example, if someone spits some really coarse word on the board, the manager can instantly withdraw the shape or word on the board at his discretion.

6.Team Contribution Outline

In our team, we use GitHub as the shared development platform to work together. We had regular team meetings per week to discuss the design and implementation. Here is the outline of team contribution:

Name	Contribution	Percentage
Xu Yang	Implement the Client UI and Server UI.	25%
Yizhou Zhu	Implement the Server Backend.	25%
Haonan Chen	Implement the Client Backend and the middleware.	25%
Chen Xu	Write the report; design the format of Client UI and Server UI	25%

Table 2 Team Contribution Outline

7.Conclusion

The interactive and reliable communication is the most crucial factor in the distributed system. Therefore, robust technology stacks and architecture design are needed to support it. This distribute shared whiteboard system has utilizes such technology such as thread pool and message queue to achieve the goal and conquer the concurrency and message passing challenges of distributed system.

Thanks for every team member worked on this project. For a detailed review of source code, please refer to our GitHub link: https://github.com/EggTronic/COMP90015-Shared-White-Board