COMP90015 Distributed Systems
Assignment 2: Distributed Shared Whiteboard

Renjie Meng 877396
The University of Melbourne

May 27, 2020

1 Problem Context

In this project, a shared white board system which allows multiple user to draw shapes and input text together in real-time is designed and implemented. This system follows a client-server architecture. The communication as well as the interaction between client and server is based the RMI provided by JAVA. Based on this design, users are allowed to draw lines, rectangles, circles, free-hand line and type text onto the whiteboard, and the changes would been seen by all the users in the whiteboard in the real time concurrently.

For the failure handling of the system, invalid parameters for starting the programs (i.e. illegal port number, unreachable server address, duplicate username), illegal operation (e.g. create a canvas while there is already one) and communication errors can be detected, handled properly as well as notify the user with proper message on the GUI. Furthermore, the connectivity between the client and server is shown on both client and server side's GUI.

2 System Architecture

In this project, the shared white board system follows the traditional **client-server architecture**. There is one **centralized** server which is responsible for the *user management* as well as *broadcasting* user session messages to the users. During broadcasting the message to each user, this system utilize the Java *thread* to *speed up* the broadcasting to achieve better user experience. With a single central server, the *concurrency* and user management is much easier to implement.

This system uses two different RMI to enable the communication between client and server. This would be discussed in detail in section 2.1.3.

2.1 Design diagrams

Below are the class diagrams as well as the interaction diagrams of this system.

2.1.1 class diagram

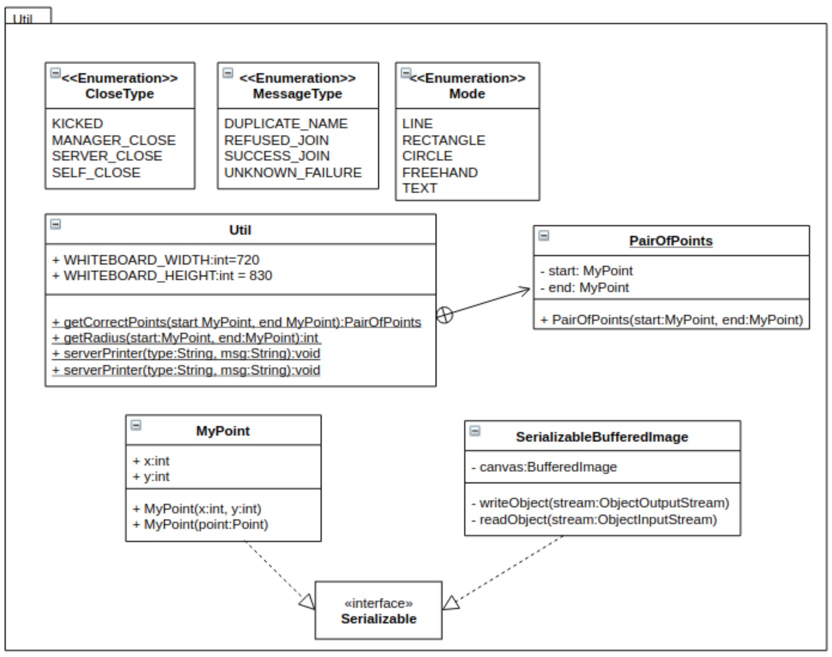

Util
<<Enumeration>> CloseType
KICKED
MANAGER_CLOSE
SERVER_CLOSE
SELF_CLOSE
<<Enumeration>> MessageType
DUPLICATE_NAME
REFUSED_JOIN
SUCCESS_JOIN
UNKNOWN_FAILURE
<<Enumeration>> Mode
LINE
RECTANGLE
CIRCLE
FREEHAND
TEXT
Util
+ WHITEBOARD_WIDTH:int=720
+ WHITEBOARD_HEIGHT:int = 830
+ getCorrectPoints(start:MyPoint, end:MyPoint):PairOfPoints
+ getRadius(start:MyPoint, end:MyPoint):int
+ serverPrinter(type:String, msg:String):void
+ serverPrinter(type:String, msg:String):void
PairOfPoints
- start: MyPoint
- end: MyPoint
+ PairOfPoints(start:MyPoint, end:MyPoint)
MyPoint
+ x:int
+ y:int
+ MyPoint(x:int, y:int)
+ MyPoint(point:Point)
SerializableBufferedImage
- canvas:BufferedImage
- writeObject(stream:ObjectOutputStream)
- readObject(stream:ObjectInputStream)
«interface» Serializable

Figure 1: Class Diagram of Util Package

Figure 1 shows the class diagram for the Util package. This package includes common classes used by other packages. There are 3 enum classes, which are **CloseType**, **MessageType** and **Mode**. Mode refers to the drawing mode, it is used for **interaction** between the client and server for message exchanging. The **MessageType** and **CloseType** defines the information that the server would send to each client based on the semantics of the user operations. Then, there are two classes **MyPoint**, **SerializeableBufferedImage** who implement the **Serializable** interface, to make sure they are able to be passed between RMI calls. The **Util** class contains an inner class which is **PairOfPoints**, this is used for calculating correct points for drawing. And in the Util class, there are some static constants and methods used by other classes.

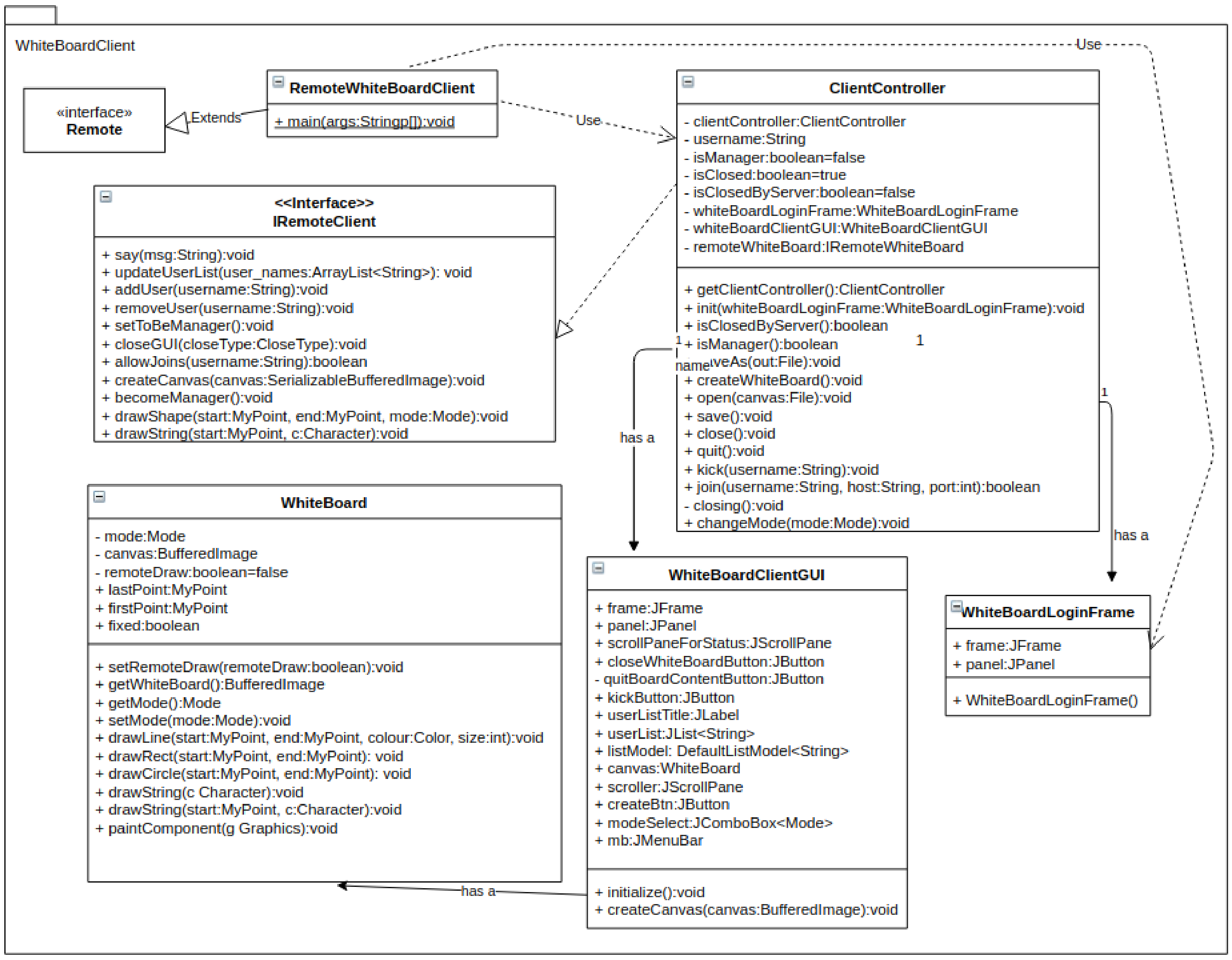
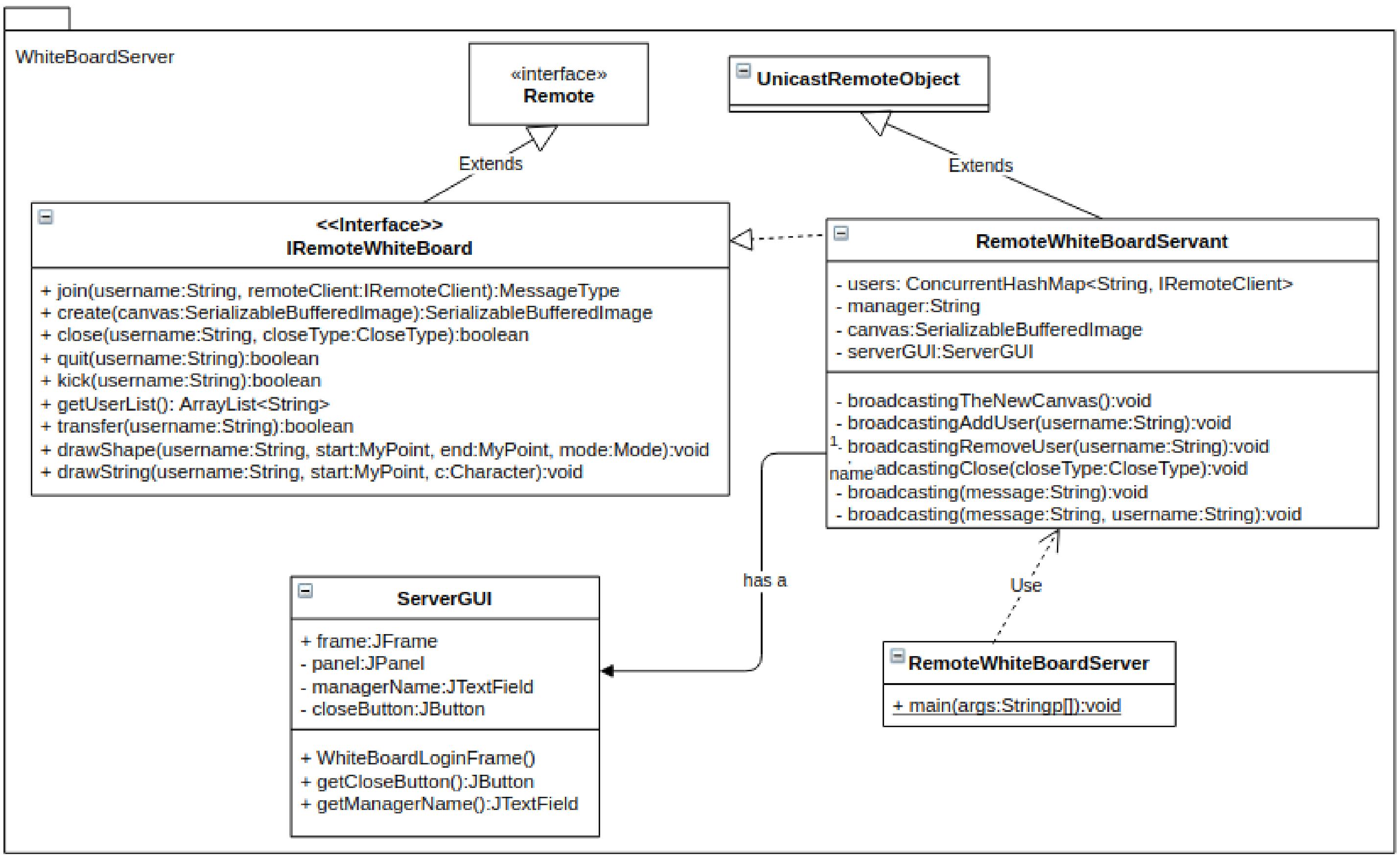

WhiteBoardClient
«interface» Remote
Extends
RemoteWhiteBoardClient
+ main(args:Stringp[]):void
Use
ClientController
- clientController:ClientController
- username:String
- isManager:boolean=false
- isClosed:boolean=true
- isClosedByServer:boolean=false
- whiteBoardLoginFrame:WhiteBoardLoginFrame
- whiteBoardClientGUI:WhiteBoardClientGUI
- remoteWhiteBoard:IRemoteWhiteBoard
+ getClientController():ClientController
+ init(whiteBoardLoginFrame:WhiteBoardLoginFrame):void
+ isClosedByServer():boolean
+ isManager():boolean
+ saveAs(out:File):void
+ createWhiteBoard():void
+ open(canvas:File):void
+ save():void
+ close():void
+ quit():void
+ kick(username:String):void
+ join(username:String, host:String, port:int):boolean
- closing():void
+ changeMode(mode:Mode):void
Use
<<Interface>> IRemoteClient
+ say(msg:String):void
+ updateUserList(user_names:ArrayList<String>): void
+ addUser(username:String):void
+ removeUser(username:String):void
+ setToBeManager():void
+ closeGUI(closeType:CloseType):void
+ allowJoins(username:String):boolean
+ createCanvas(canvas:SerializableBufferedImage):void
+ becomeManager():void
+ drawShape(start:MyPoint, end:MyPoint, mode:Mode):void
+ drawString(start:MyPoint, c:Character):void
1
name
1
1
has a
has a
WhiteBoard
- mode:Mode
- canvas:BufferedImage
- remoteDraw:boolean=false
+ lastPoint:MyPoint
+ firstPoint:MyPoint
+ fixed:boolean
+ setRemoteDraw(remoteDraw:boolean):void
+ getWhiteBoard():BufferedImage
+ getMode():Mode
+ setMode(mode:Mode):void
+ drawLine(start:MyPoint, end:MyPoint, colour:Color, size:int):void
+ drawRect(start:MyPoint, end:MyPoint): void
+ drawCircle(start:MyPoint, end:MyPoint): void
+ drawString(c Character):void
+ drawString(start:MyPoint, c:Character):void
+ paintComponent(g Graphics):void
WhiteBoardClientGUI
+ frame:JFrame
+ panel:JPanel
+ scrollPaneForStatus:JScrollPane
+ closeWhiteBoardButton:JButton
- quitBoardContentButton:JButton
+ kickButton:JButton
+ userListTitle:JLabel
+ userList:JList<String>
+ listModel: DefaultListModel<String>
+ canvas:WhiteBoard
+ scroller:JScrollPane
+ createBtn:JButton
+ modeSelect:JComboBox<Mode>
+ mb:JMenuBar
+ initialize():void
+ createCanvas(canvas:BufferedImage):void
has a
WhiteBoardLoginFrame
+ frame:JFrame
+ panel:JPanel
+ WhiteBoardLoginFrame()

Figure 2: Class Diagram of ClientWhiteBoard Package

Figure 2 shows the class diagram of the ClientWhiteBoard package and this package contains the client site code.

There is one interface and 5 classes in this package. The **IRemoteClient** is an interface inherits from the Remote interface, this is the RIM interface of the client side. The **WhiteBoard**, **WhiteBoardClientGUI** and **WhiteBoardLoginFrame** are the GUI classes on the client side. WhiteBoardLoginFrame define the UI elements of the login interface, then the WhiteBoardClientGUI defines the UI elements of the whiteboard used by the user after it login. Then, the WhiteBoard class defines the whiteboard that users draw on, it provides basic shape drawing as well as text typing.

Then, the **ClientController** class receives the input from the GUI, then it communicates with the remote server. It would handling the client drawing input as well as the user management request, then it calls the RMI of the server to perform appropriate operation. Also, this class is responsible for the error handling of the remote server, since the server might crashed at any time.



WhiteBoardServer
«interface»
Remote
UnicastRemoteObject
Extends
Extends
<<Interface>>
IRemoteWhiteBoard
+ join(username:String, remoteClient:IRemoteClient):MessageType
+ create(canvas:SerializableBufferedImage):SerializableBufferedImage
+ close(username:String, closeType:CloseType):boolean
+ quit(username:String):boolean
+ kick(username:String):boolean
+ getUserList(): ArrayList<String>
+ transfer(username:String):boolean
+ drawShape(username:String, start:MyPoint, end:MyPoint, mode:Mode):void
+ drawString(username:String, start:MyPoint, c:Character):void
RemoteWhiteBoardServant
- users: ConcurrentHashMap<String, IRemoteClient>
- manager:String
- canvas:SerializableBufferedImage
- serverGUI:ServerGUI
- broadcastingTheNewCanvas():void
- broadcastingAddUser(username:String):void
- broadcastingRemoveUser(username:String):void
- broadcastingClose(closeType:CloseType):void
- broadcasting(message:String):void
- broadcasting(message:String, username:String):void
1
name
has a
Use
ServerGUI
+ frame:JFrame
- panel:JPanel
- managerName:JTextField
- closeButton:JButton
+ WhiteBoardLoginFrame()
+ getCloseButton():JButton
+ getManagerName():JTextField
RemoteWhiteBoardServer
+ main(args:Stringp[]):void

Figure 3: Class Diagram of WhiteBoardServer Package

Figure 3 shows the classes from the **WhiteBoardServer** package and this package includes class of the server side. There are 1 interface and 3 classes in this package. The IRemoteWhiteBoard extends the Remote interface and this defines the RMI of the server side, it provides drawing and user management APIs. Then, the **ServerGUI** class defines the server GUI, it includes the manager name as well as the close button to close the whiteboard. Then, the **RemoteWhiteBoardServer** class implements the IRemoteWhiteBoard, it controls all of the user management as well as the user drawing activity. Finally, the **RemoteWhiteBoardServer** class creates the RemoteWhiteBoardServant and starts the server.

2.1.2 interaction diagram

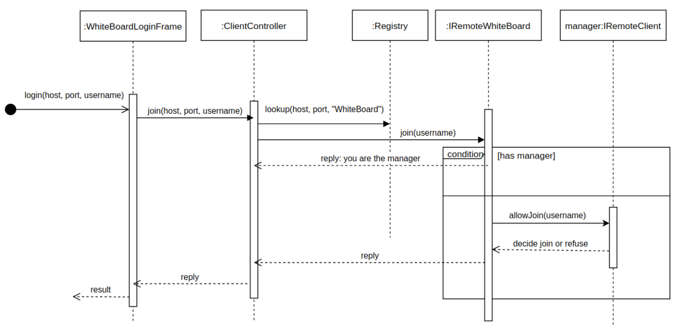


Figure 4: Sequence Diagram of user join

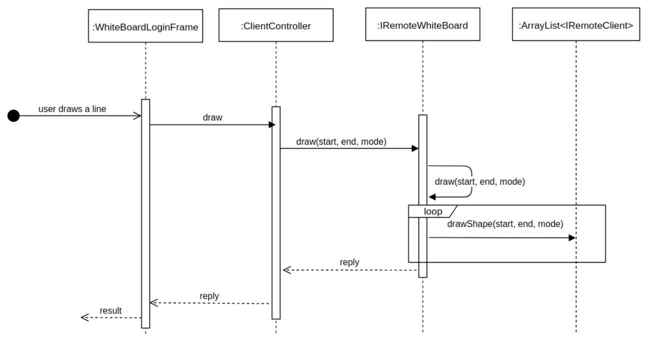


Figure 5: Sequence Diagram of user drawing a line

As Figure 4 and 5 show the sequence diagram of user joining and user drawing a line.

From figure 4, we can see that, when the user clicked on join, it provides the host, port and its username to the **ClientController**, then the ClientController tries to lookup the RMI from the registry with given host and port. Then, after having the IRemoteWhiteBoard object, the ClientController asks the **IRemoteWhiteBoard** to join the user with the given username. Then, if there is a manager in the IRemoteWhiteBoard, it would ask the manager whether the manager allow the user to join, otherwise it would promote this user to be the manager directly.

The figure 5 shows how the user draws a line. When the user draw a line on the whiteboard, it would ask the **ClientController** to broadcast the draw line operation to other users in the whiteboard. Then, the ClientController asks the **IRemoteWhiteBoard** to broadcast the draw line operation. After this, the IRemoteWhiteBoard would send this draw line operation to all other users concurrently. So, all the user could see same draw-line operation after this.

2.1.3 overall interaction diagram

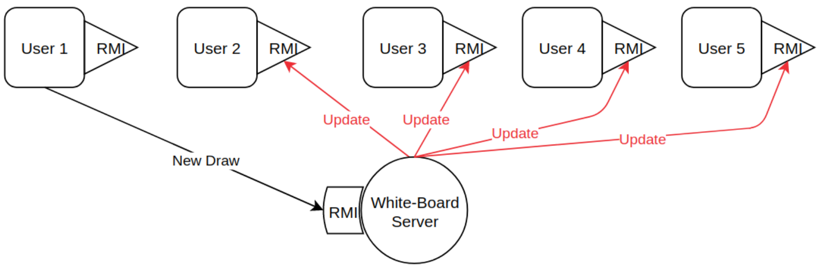


Figure 6: fig:Overall Interaction Diagram

Figure 6 shows the overall interaction of the shared whiteboard system. Basically, the message flow in the system follow the same workflow. The user 1 sends a new draw to the server by calling the RMI of server, then the server broadcasts this operation to all other users to update their whiteboard concurrently by calling their RMI.

2.2 The UI Design

Beblow are the UI deisgn for this project, figure 7 is the UI of the server, figure 8 is the client login GUI and the figure 9 is the main place for client to draw colleborately.

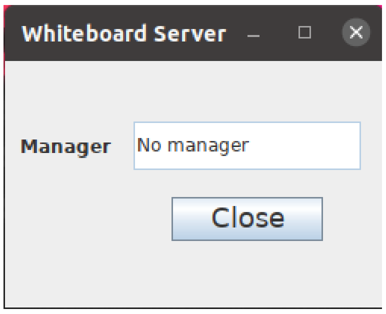


Figure 7: Server GUI

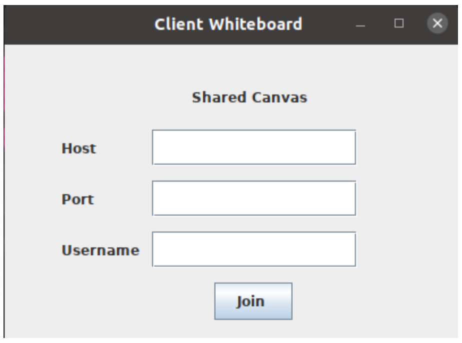


Figure 8: Client Login GUI

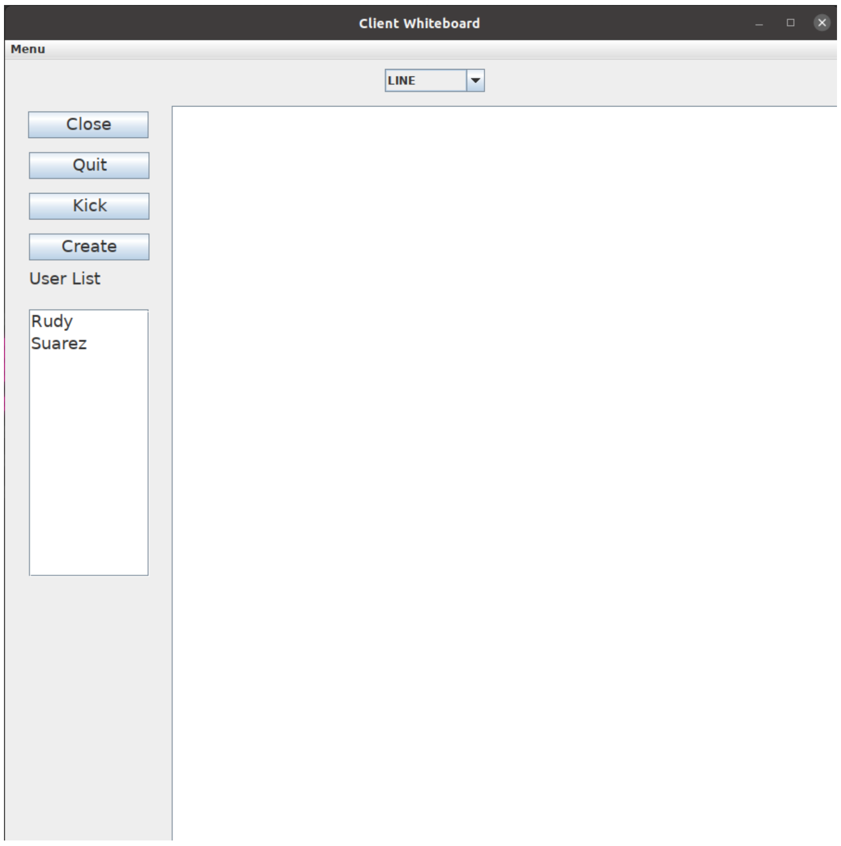


Figure 9: Client Canvas GUI

3 Communication protocols and message formats

The communication of the client and server are based on the double-way RMI call. The table below shows the RMI of the client and server. Table 1 shows the RMI of the server which is the IRemoteWhiteBoard Interface. Then, table 2 shows the RMI of the server side which is the IRemoteClient Interface. The first column shows the method while the second column shows the semantics of the method.

Table 1: IRemoteWhiteBoard RMI Interfaces

Method	Semantics
MessageType join(String username, IRemoteClient remoteClient) throws RemoteException;	join canvas
SerializableBufferedImage create(SerializableBufferedImage canvas) throws RemoteException;	create canvas
boolean close(String username, CloseType closeType) throws RemoteException;	close canvas
boolean quit(String username) throws RemoteException;	quit canvas
boolean kick(String username) throws RemoteException;	kick user
ArrayList<String> getUserList() throws RemoteException;	get users
boolean transfer(String username) throws RemoteException;	transfer
void drawShape(String username, MyPoint start, MyPoint end, Mode mode) throws RemoteException;	drawShape
void drawString(String username, MyPoint start, Character c) throws RemoteException;	drawText

There are 10 methods defined in this remote interface in total in table 1. And these methods can be divided into 2 categories of functionalities. The first one is about the user management, it includes **join**, **quit**, **close**, **create**, **kick**, **transfer** and **getUserList**. Then, the second one is about the drawing functionality, it consists of **drawShape** and **drawString**. Clients would call the corresponding method from these method to interact with the server.

Table 2: IRemoteClient RMI Interaction

Method	Semantics
void say(String msg) throws RemoteException;	print on client
void updateUserList(ArrayList<String> user_names) throws RemoteException;	update the users info
void addUser(String username) throws RemoteException;	add user
void removeUser(String username) throws RemoteException;	remove user
void setToBeManager() throws RemoteException;	delegate manager
void closeGUI(CloseType closeType) throws RemoteException;	close
boolean allowJoins(String username) throws RemoteException;	decision on join
void createCanvas(SerializableBufferedImage canvas) throws RemoteException;	create new canvas
void becomeManager() throws RemoteException;	turn to manager
void drawShape(MyPoint start, MyPoint end, Mode mode) throws RemoteException;	draw shape
void drawString(MyPoint start, Character c) throws RemoteException;	draw text

In table 2, there are 11 methods in total, which includes the management and drawing operations on the client side. The management methods include **say**, **updateUserList**, **addUser**, **removeUser**, **setToBeManager**, **closeGUI**, **allowJoins**, **createCanvas** and **becomeManager**. Then, the drawing functionality includes **drawShape** as well as **drawString**. The server would call the proper method to interact with the whiteboard user.

Table 3: Message Type Enum reference

KICKED	been kicked
MANAGER_CLOSE	manager closes canvas
SERVER_CLOSE	server closes canvas
SELF_CLOSE	closed by self
LINE	draw line
RECTANGLE	draw rectangle
CIRCLE	draw circle
FREEHAND	free hand draw
TEXT	input text

Table 3 shows the Enum used during the message exchange in the system, these are all metionded in the Util package diagram in section 2.1.1.

4 Implementation Details

This section would include several implementation details and they are about communication, user management and drawing.

In order to exchange message with the JAVA RMI, the parameters and the return type of methods should be serialized. So, this system implements the serializable Point and BufferedImage classes which are called MyPoint and SeriablizablrBufferedImage.

Then, each client has a state variable called remoteDraw, it indicates whether the draw operation is from the local client or the remote server, this enables the user and server to draw on the whiteboard independently.

Thirdly, the close operation in this system might have 3 different trigger types, the manager, server and client may close the application. Thus, there is a closetype passed into the close method to handle different kinds of close operations.

Then, in this application, there is one place that the transfer of manager-ownership could be realized. That is, when the manager tries to kick it self out of the whiteboard, then it would be asked to select a new manager first. After the selection, the manager ownership would be transfer to the selected user. In addition, the previous manager would not leave the whiteboard, but only become a normal user.

5 New Innovation

There several innovations in this system. Firstly, the server could close the whiteboard, it would notify all the client that the whiteboad has been closed by the server. Furthermore, the server could see who is the current manager of the whiteboard.

Then, when the manager kick itself, it would be asked to select a new host, then the manager would be transfer to the selected new manager. And the previous manager would be a normal user in the whiteboard.

6 How to run the system

6.1 Server

java -jar Distributed-Whiteboard-Server.jar -p <port number>

6.2 Client

java -jar Distributed-Whiteboard-Client.jar

7 Conclusion

In conclusion, this shared whiteboard application are based on the client-sever architecture, then the message exchange in the system are based on the RMI of both client and server. Overall, the shared whiteboard would connects user together, it is extremely useful when there is a real-world meeting difficulty, such as the COVID-19.