COMP90015 Distributed Systems

Assignment 2 - Report

Yixin SHEN 1336242

Problem Context

Assignment 2 aims to implement a shared whiteboard system that allows multiple users to draw simultaneously on a canvas. The changes by any user should be displayed to other users in real-time. There are some main challenges in this project, including dealing with concurrency, structuring your application, handling the system state, dealing with networked communication, and implementing GUI. If the system is fully developed, it should have one manager and multiple common users at the same time. All the illegal operations should be handled properly and the application should be able to prompt warnings to the users. The system must be implemented in Java, and the distribution feature should be implemented by one appropriate technique, in my implementation, it is Java RMI. And a variety of auxiliary functions should be implemented, including basic features and advanced features. All the features itemized in the specification are implemented in my project.

System Architecture

System Overview

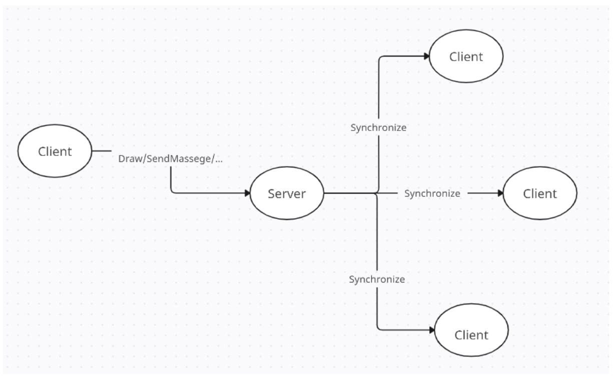


Figure 1: The synchronize process of the system

Overall, my system follows the Client-Server architecture, which contains one server and multiple clients. When a client takes some actions, for example sending a message or drawing something on the whiteboard, the changes/requests will be sent to the server first. Afterward, the server will synchronize the changes to all the other clients.

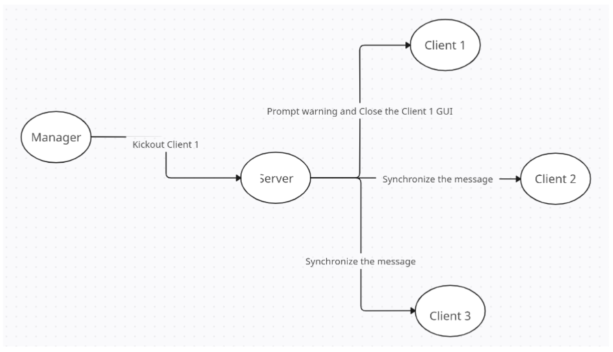


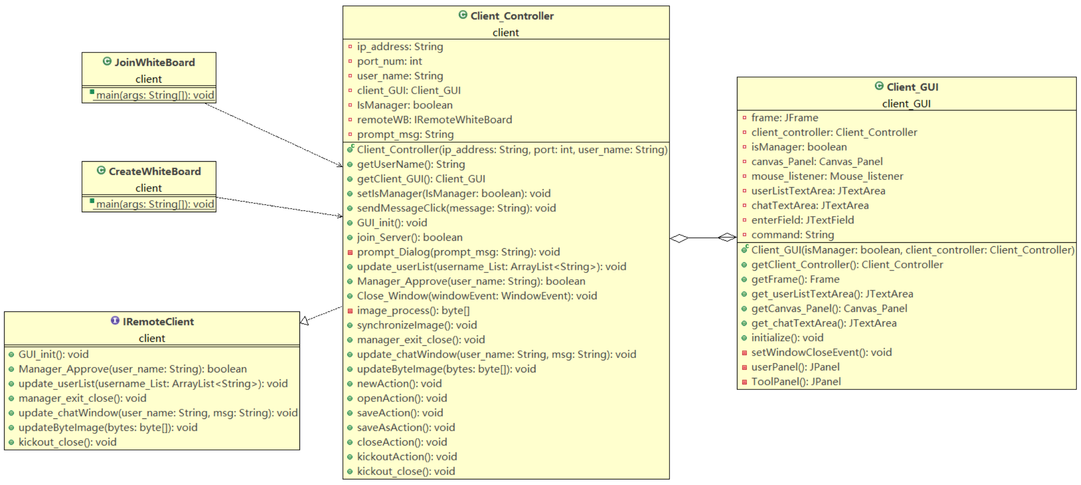
Figure 2: The kick-out process of the system

Take the kick-out process as an example, it is a special synchronize process as the request can only be made by the manager. However, the principle is the same. If the manager sends a kick-out client 1 request, this request will be sent to the server. Then through a **kickout_user** function on the server side, this certain client 1 will be removed from the client list stored on the server and the GUI of client 1 will be closed at the same time, which indicates it has been kicked out from the whiteboard game.

Client Side

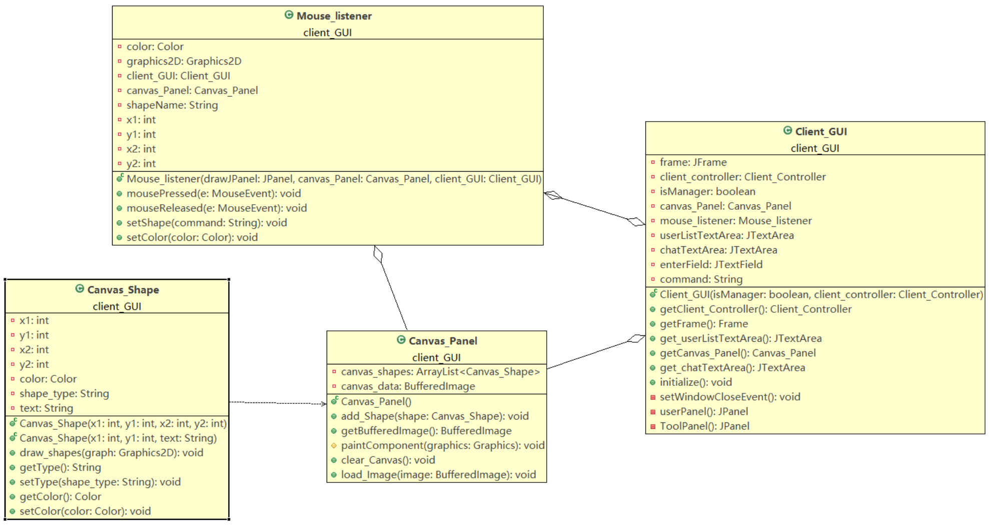
Figure 3 and Figure 4 present the UML class diagram of the client side and client GUI accordingly. From Figure 3, we can find that the client-side module consists of several components. The **CreateWhiteBoard** class and **JoinWhiteBoard** class are the manager entry and common user entry respectively. In addition, both of them can invoke the **Client_Controller** class, which implements the **IRemoteClient** RMI remote interface. Hence, the **Client_Controller** object is used to not only control the GUI actions but also take the responsibility of communicating with the server side.

Referring to the GUI part, the **Canvas_Panel** class is used to store the shapes and entire canvas data as *BufferedImage*. It is used by the **Client_Controller** to call the contents of the whiteboard and pass it to the server for synchronization. The **Mouse_listener** is designed to listen to the mouse events, for example, when the mouse is pressed and the current shape is text, we can add a string of text into the canvas, and when the mouse is released, we can add one of the several basic shapes into the canvas. The **Canvas_Shape** contains the drawing method of the required shapes in the project, and each object contains information about the type, coordinates, and color of the drawing.



JoinWhiteBoard
client
main(args: String[]): void
CreateWhiteBoard
client
main(args: String[]): void
IRemoteClient
client
GUI_init(): void
Manager_Approve(user_name: String): boolean
update_userList(username_List: ArrayList<String>): void
manager_exit_close(): void
update_chatWindow(user_name: String, msg: String): void
updateByteImage(bytes: byte[]): void
kickout_close(): void
Client_Controller
client
ip_address: String
port_num: int
user_name: String
client_GUI: Client_GUI
IsManager: boolean
remoteWB: IRemoteWhiteBoard
prompt_msg: String
Client_Controller(ip_address: String, port: int, user_name: String)
getUserName(): String
getClient_GUI(): Client_GUI
setIsManager(IsManager: boolean): void
sendMessageClick(message: String): void
GUI_init(): void
join_Server(): boolean
prompt_Dialog(prompt_msg: String): void
update_userList(username_List: ArrayList<String>): void
Manager_Approve(user_name: String): boolean
Close_Window(windowEvent: WindowEvent): void
image_process(): byte[]
synchronizeImage(): void
manager_exit_close(): void
update_chatWindow(user_name: String, msg: String): void
updateByteImage(bytes: byte[]): void
newAction(): void
openAction(): void
saveAction(): void
saveAsAction(): void
closeAction(): void
kickoutAction(): void
kickout_close(): void
Client_GUI
client_GUI
frame: JFrame
client_controller: Client_Controller
isManager: boolean
canvas_Panel: Canvas_Panel
mouse_listener: Mouse_listener
userListTextArea: JTextArea
chatTextArea: JTextArea
enterField: JTextField
command: String
Client_GUI(isManager: boolean, client_controller: Client_Controller)
getClient_Controller(): Client_Controller
getFrame(): Frame
get_userListTextArea(): JTextArea
getCanvas_Panel(): Canvas_Panel
get_chatTextArea(): JTextArea
initialize(): void
setWindowCloseEvent(): void
userPanel(): JPanel
ToolPanel(): JPanel

Figure 3: UML class diagram of the client side



Mouse_listener
client_GUI
color: Color
graphics2D: Graphics2D
client_GUI: Client_GUI
canvas_Panel: Canvas_Panel
shapeName: String
x1: int
y1: int
x2: int
y2: int
Mouse_listener(drawJPanel: JPanel, canvas_Panel: Canvas_Panel, client_GUI: Client_GUI)
mousePressed(e: MouseEvent): void
mouseReleased(e: MouseEvent): void
setShape(command: String): void
setColor(color: Color): void
Canvas_Shape
client_GUI
x1: int
y1: int
x2: int
y2: int
color: Color
shape_type: String
text: String
Canvas_Shape(x1: int, y1: int, x2: int, y2: int)
Canvas_Shape(x1: int, y1: int, text: String)
draw_shapes(graph: Graphics2D): void
getType(): String
setType(shape_type: String): void
getColor(): Color
setColor(color: Color): void
Canvas_Panel
client_GUI
canvas_shapes: ArrayList<Canvas_Shape>
canvas_data: BufferedImage
Canvas_Panel()
add_Shape(shape: Canvas_Shape): void
getBufferedImage(): BufferedImage
paintComponent(graphics: Graphics): void
clear_Canvas(): void
load_Image(image: BufferedImage): void
Client_GUI
client_GUI
frame: JFrame
client_controller: Client_Controller
isManager: boolean
canvas_Panel: Canvas_Panel
mouse_listener: Mouse_listener
userListTextArea: JTextArea
chatTextArea: JTextArea
enterField: JTextField
command: String
Client_GUI(isManager: boolean, client_controller: Client_Controller)
getClient_Controller(): Client_Controller
getFrame(): Frame
get_userListTextArea(): JTextArea
getCanvas_Panel(): Canvas_Panel
get_chatTextArea(): JTextArea
initialize(): void
setWindowCloseEvent(): void
userPanel(): JPanel
ToolPanel(): JPanel

Figure 4: UML class diagram of the client GUI

Server Side

When it comes to the server side, shown in Figure 5, the composition will be relatively simple. The **WhiteBoardServer** class is the launcher of the whiteboard server, which means it creates the *Registry* of RMI and we should run it before running the client entries. The **RemoteWhiteBoard** is the servant of the **IRemoteWhiteBoard** interface, which implements the methods in the RMI interface, allowing remote communication with the clients. Lastly, the **JoinType** enumeration is used to indicate the return values of different cases after the client has requested to join. Different return values will be sent back to the client side and the **Client_Controller** will prompt different warning messages to the user based on the return value.

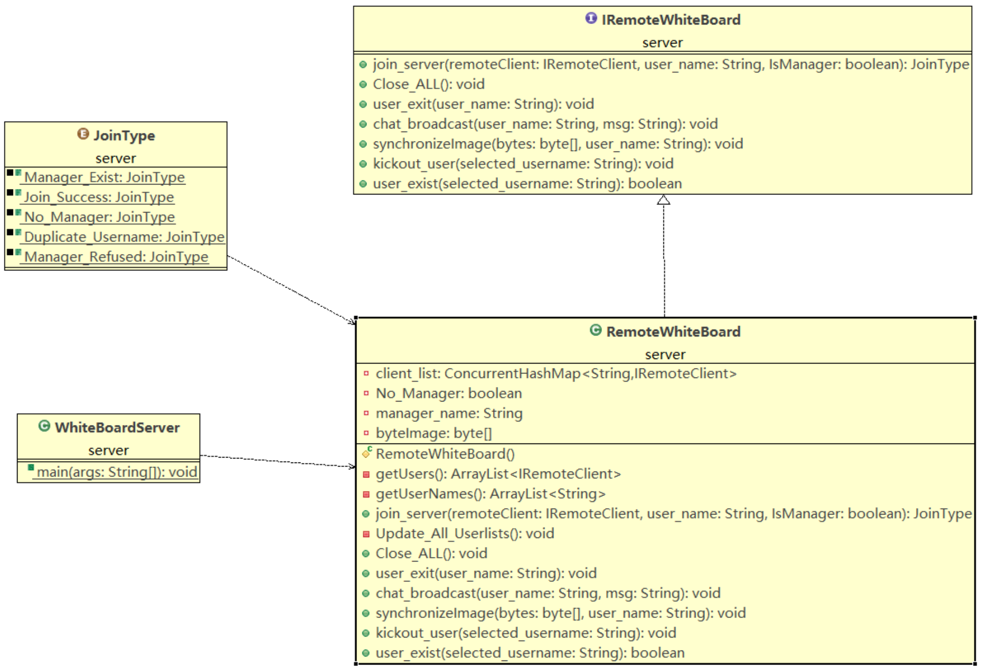

IRemoteWhiteBoard
server
join_server(remoteClient: IRemoteClient, user_name: String, IsManager: boolean): JoinType
Close_ALL(): void
user_exit(user_name: String): void
chat_broadcast(user_name: String, msg: String): void
synchronizeImage(bytes: byte[], user_name: String): void
kickout_user(selected_username: String): void
user_exist(selected_username: String): boolean
JoinType
server
Manager_Exist: JoinType
Join_Success: JoinType
No_Manager: JoinType
Duplicate_Username: JoinType
Manager_Refused: JoinType
WhiteBoardServer
server
main(args: String[]): void
RemoteWhiteBoard
server
client_list: ConcurrentHashMap<String,IRemoteClient>
No_Manager: boolean
manager_name: String
byteImage: byte[]
RemoteWhiteBoard()
getUsers(): ArrayList<IRemoteClient>
getUserNames(): ArrayList<String>
join_server(remoteClient: IRemoteClient, user_name: String, IsManager: boolean): JoinType
Update_All_Userlists(): void
Close_ALL(): void
user_exit(user_name: String): void
chat_broadcast(user_name: String, msg: String): void
synchronizeImage(bytes: byte[], user_name: String): void
kickout_user(selected_username: String): void
user_exist(selected_username: String): boolean

Figure 5: UML class diagram of the server side

Communication Protocols and Message Formats

The communication protocol used in the project is Java RMI. RMI stands for Remote Method Invocation, allowing an object residing in one system to access/invoke an object running on another. It is an extension of the object-oriented programming model.

In this project, two RMI interfaces are implemented. One is for the server side, while the other is for the client side, achieving duplex communication between the server and the client. The overview of the RMI implementation is shown in Figure 6. From the Figure, we can see the **Client_Controller** class is the servant of the **IRemoteClient** interface, and the **RemoteWhiteBoard** class is the servant of the **IRemoteWhiteBoard** interface. The implementation details of the communication part are included in the next section.

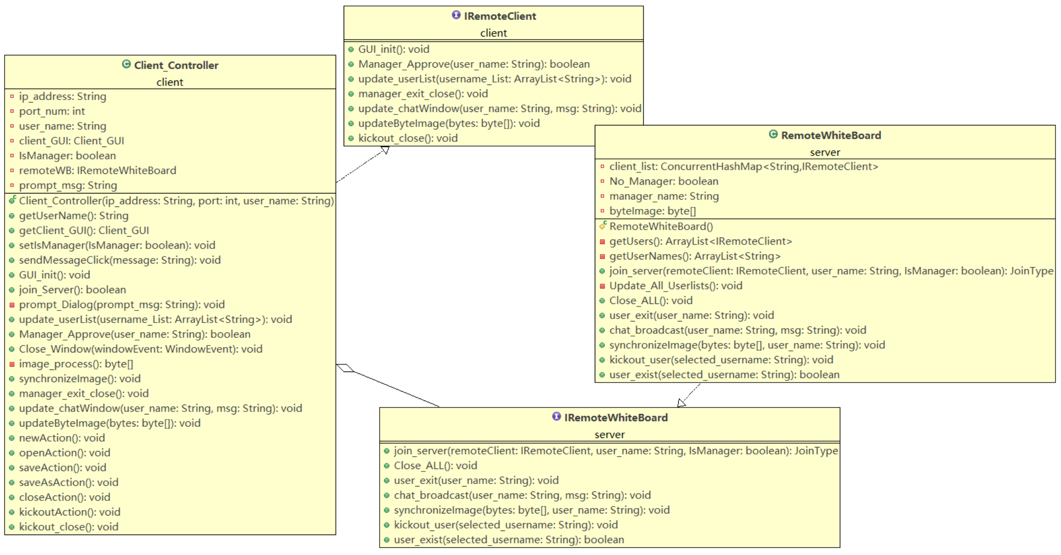

IRemoteClient
client
GUI_init(): void
Manager_Approve(user_name: String): boolean
update_userList(username_List: ArrayList<String>): void
manager_exit_close(): void
update_chatWindow(user_name: String, msg: String): void
updateByteImage(bytes: byte[]): void
kickout_close(): void
Client_Controller
client
ip_address: String
port_num: int
user_name: String
client_GUI: Client_GUI
IsManager: boolean
remoteWB: IRemoteWhiteBoard
prompt_msg: String
Client_Controller(ip_address: String, port: int, user_name: String)
getUserName(): String
getClient_GUI(): Client_GUI
setIsManager(IsManager: boolean): void
sendMessageClick(message: String): void
GUI_init(): void
join_Server(): boolean
prompt_Dialog(prompt_msg: String): void
update_userList(username_List: ArrayList<String>): void
Manager_Approve(user_name: String): boolean
Close_Window(windowEvent: WindowEvent): void
image_process(): byte[]
synchronizeImage(): void
manager_exit_close(): void
update_chatWindow(user_name: String, msg: String): void
updateByteImage(bytes: byte[]): void
newAction(): void
openAction(): void
saveAction(): void
saveAsAction(): void
closeAction(): void
kickoutAction(): void
kickout_close(): void
RemoteWhiteBoard
server
client_list: ConcurrentHashMap<String,IRemoteClient>
No_Manager: boolean
manager_name: String
byteImage: byte[]
RemoteWhiteBoard()
getUsers(): ArrayList<IRemoteClient>
getUserNames(): ArrayList<String>
join_server(remoteClient: IRemoteClient, user_name: String, IsManager: boolean): JoinType
Update_All_Userlists(): void
Close_ALL(): void
user_exit(user_name: String): void
chat_broadcast(user_name: String, msg: String): void
synchronizeImage(bytes: byte[], user_name: String): void
kickout_user(selected_username: String): void
user_exist(selected_username: String): boolean
IRemoteWhiteBoard
server
join_server(remoteClient: IRemoteClient, user_name: String, IsManager: boolean): JoinType
Close_ALL(): void
user_exit(user_name: String): void
chat_broadcast(user_name: String, msg: String): void
synchronizeImage(bytes: byte[], user_name: String): void
kickout_user(selected_username: String): void
user_exist(selected_username: String): boolean

Figure 6: The RMI structure overview of the system

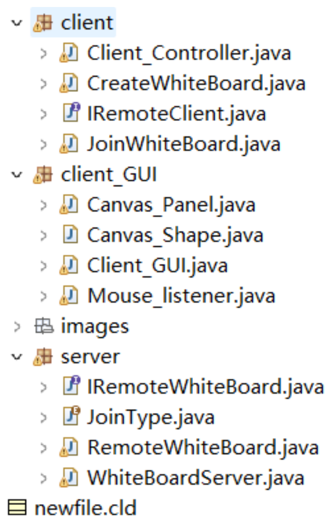
Referring to the message formats, the main feature of this application is to synchronize the changes in the canvas for every client. In this project, the image data is synchronized by using **BufferedImage**. When any client draws a new shape on the canvas, the mouse listener functions will be activated and send the new image data to the server, and the server will synchronize/update all the other clients' canvases. Some other messages, like chat messages, and join return values, are various. You can find the exact message formats in Figure 6.

Implementation Details

Code Structure

Figure 7 shows the code structure of the project. Overall, there are three packages, including **client** package, **client_GUI** package, and **server** package. Among them, the

client package consists of two client program entry classes, one client RMI interface, and one client_controller class. Regarding the **client_GUI** package, it focuses on the GUI initialization and whiteboard drawing feature's implementation and data storage. When it comes to the **server** package, it contains one server launcher/object, one server RMI interface, and one server RMI servant, and a join type return value enumeration.



```
client
    Client_Controller.java
    CreateWhiteBoard.java
    IRemoteClient.java
    JoinWhiteBoard.java
client_GUI
    Canvas_Panel.java
    Canvas_Shape.java
    Client_GUI.java
    Mouse_listener.java
images
server
    IRemoteWhiteBoard.java
    JoinType.java
    RemoteWhiteBoard.java
    WhiteBoardServer.java
newfile.cld
```

Figure 7: The code structure of the project

Sequence Diagram

Since client joining is a crucial but complex feature, it will be used as an example to introduce the implementation details. Figure 8 presents the sequence diagram of the common user joining procedure. When a common user starts the join procedure by running **JoinWhiteBoard** with appropriate arguments (i.e., IPAddress, Port, username), it will create a Client_Controller object and invoke the join_Server() function. In this function, the client object will look up the remote reference bound in the registry based on the ip address and port provided. Afterward, it will call the join_server function implemented by the server servant, checking whether the username is duplicated or not. If not, then try to get approval from the manager. If successful, the common user will enter the stage where the GUI is initialized and the image of the current whiteboard is loaded. If failed, the common user joining procedure will be suspended and call the System.exit(0).

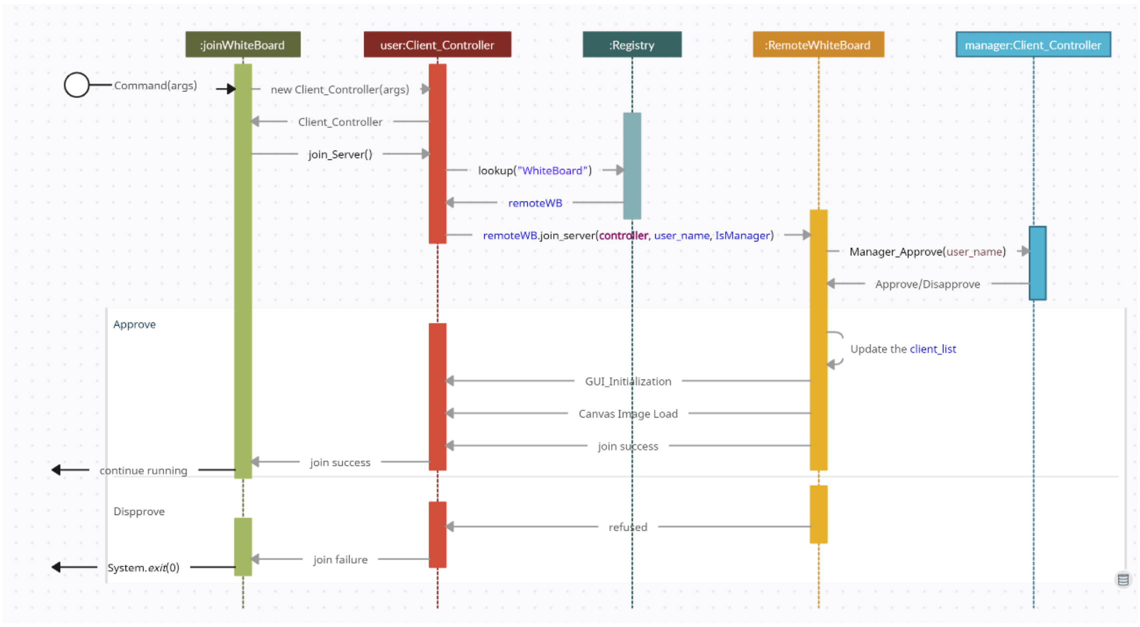
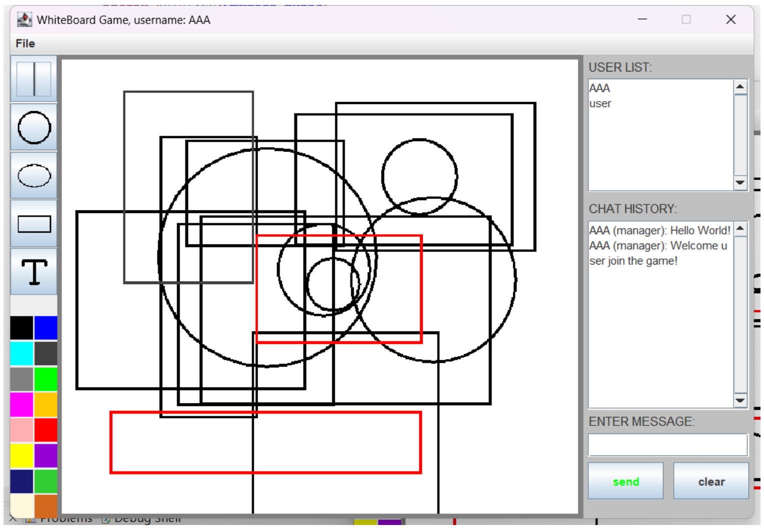


Figure 8: The sequence diagram of join process

New Innovations

All the basic and advanced features itemized in the specification are implemented in my system. All functions in the guidelines on usage/operation are implemented exactly as described. Some trivial innovations that can make the system more user-friendly are also implemented. For example, in Figure 9, we can find that if a new user joins the whiteboard successfully, the manager will send one message and broadcast it to all the other users.



AAA (manager): Hello World!
AAA (manager): Welcome user join the game!

Figure 9: GUI and trivial improvements

Lastly, the operations in the file menu are implemented as follows: 1. **new** is for clearing the canvas. 2. **open** is for loading the archive of the saved canvas image. 3. **save** is for saving the current canvas image as a byte array file to a **archive** folder, the file suffix is .out. 4. **saveAs** is for saving the current canvas image to a png image. 5. **close** is for exiting the whiteboard system. 6. **kickout** is for kicking out a specific user.

Critical Analysis

Advantages

1. The system adopted a centralized server architecture, using one server to operate all the requests from the clients. Such architecture is easy to understand and maintain.
2. The GUI is intuitive, and the exceptions are handled comprehensively. If an exception occurs, it will be prompted by a Java swing dialogue.

Disadvantages

1. Since there is only one server, if the number of clients increases, the server response might be slowed. And if the server crashes, all the clients will be suspended.
2. If we need to extend the drawing function, it is hard to implement some features like resize since the image is stored as BufferedImage. In other words, it is hard to use a mouse to select a specific shape when the shapes are overlapping with each other.