COMP90015 Distributed Systems

Assignment 2 - Report

Hanyi Gao

1236476

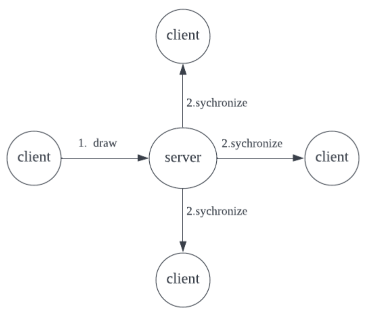
1. Problem Context

In this assignment, a distributed shared whiteboard should be designed, which allows one manager and multiple users to draw different shapes on the whiteboard simultaneously. Any change by a user would be displayed on the other users' whiteboards in real-time. The system is designed with a client-server architecture and RMI is used to establish the connection and communication between clients and the server. Remote interfaces should be well designed so that both clients and the server can handle the exchanging data more easily. Besides, the illegal operations, communication failure, and any other exceptions should be detected and handled properly. Users need to be notified with a proper message of these exceptions.

2. System Components and Design

2.1 System Architecture

The system follows the Client-Server model. There is one server and multiple clients. The centralized server is used to process the requests from clients and send the response to a certain client or all the clients. The following two figures show how the centralized server process a client's request in two different situations.



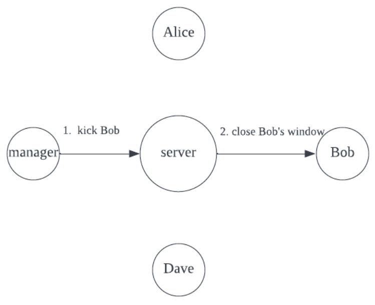


Figure 1. The user sending the draw request

Figure 2. The manager kicking a user

The communication between clients and the server is implemented by JAVA's RMI. The RMI interfaces are provided for both the server and the client sides to implement the duplex communication between the server and the client. The two interfaces are shown in the figure below. The details of the remote interfaces will be introduced in section 3.

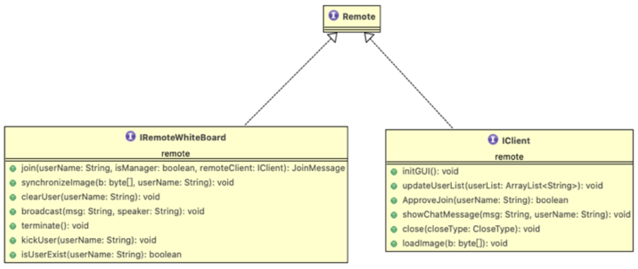

Remote
IRemoteWhiteBoard
remote
join(userName: String, isManager: boolean, remoteClient: IClient): JoinMessage
synchronizeImage(b: byte[], userName: String): void
clearUser(userName: String): void
broadcast(msg: String, speaker: String): void
terminate(): void
kickUser(userName: String): void
isUserExist(userName: String): boolean
IClient
remote
initGUI(): void
updateUserList(userList: ArrayList<String>): void
ApproveJoin(userName: String): boolean
showChatMessage(msg: String, userName: String): void
close(closeType: CloseType): void
loadImage(b: byte[]): void

Figure 3. Remote Interfaces

2.2 Client Side

The UML class diagram of the client side is shown in the figure below.

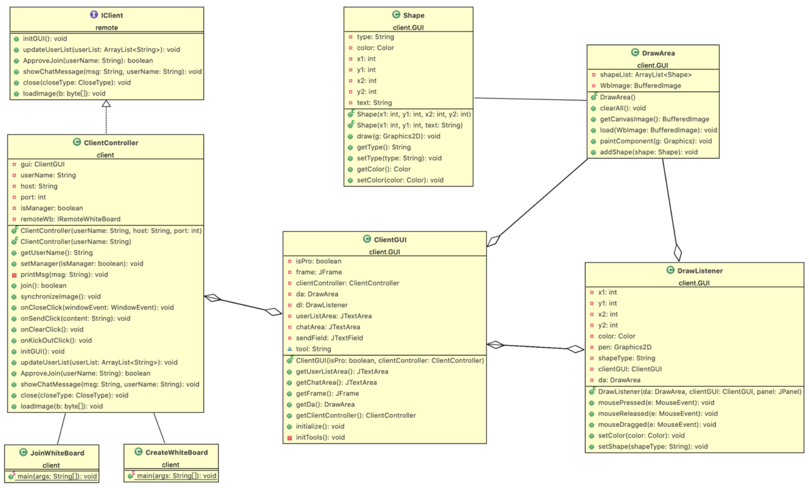

IClient
remote
initGUI(): void
updateUserList(userList: ArrayList<String>): void
ApproveJoin(userName: String): boolean
showChatMessage(msg: String, userName: String): void
close(closeType: CloseType): void
loadImage(b: byte[]): void
ClientController
client
gui: ClientGUI
userName: String
host: String
port: int
isManager: boolean
remoteWb: IRemoteWhiteBoard
ClientController(userName: String, host: String, port: int)
ClientController(userName: String)
getUserName(): String
setManager(isManager: boolean): void
printMsg(msg: String): void
join(): boolean
synchronizeImage(): void
onCloseClick(windowEvent: WindowEvent): void
onSendClick(content: String): void
onClearClick(): void
onKickOutClick(): void
initGUI(): void
updateUserList(userList: ArrayList<String>): void
ApproveJoin(userName: String): boolean
showChatMessage(msg: String, userName: String): void
close(closeType: CloseType): void
loadImage(b: byte[]): void
JoinWhiteBoard
client
main(args: String[]): void
CreateWhiteBoard
client
main(args: String[]): void
Shape
client.GUI
type: String
color: Color
x1: int
y1: int
x2: int
y2: int
text: String
Shape(x1: int, y1: int, x2: int, y2: int)
Shape(x1: int, y1: int, text: String)
draw(g: Graphics2D): void
getType(): String
setType(type: String): void
getColor(): Color
setColor(color: Color): void
DrawArea
client.GUI
shapeList: ArrayList<Shape>
WbImage: BufferedImage
DrawArea()
clearAll(): void
getCanvasImage(): BufferedImage
load(WbImage: BufferedImage): void
paintComponent(g: Graphics): void
addShape(shape: Shape): void
ClientGUI
client.GUI
isPro: boolean
frame: JFrame
clientController: ClientController
da: DrawArea
dl: DrawListener
userListArea: JTextArea
chatArea: JTextArea
sendField: JTextField
tool: String
ClientGUI(isPro: boolean, clientController: ClientController)
getUserListArea(): JTextArea
getChatArea(): JTextArea
getFrame(): JFrame
getDa(): DrawArea
getClientController(): ClientController
initialize(): void
initTools(): void
DrawListener
client.GUI
x1: int
y1: int
x2: int
y2: int
color: Color
pen: Graphics2D
shapeType: String
clientGUI: ClientGUI
da: DrawArea
DrawListener(da: DrawArea, clientGUI: ClientGUI, panel: JPanel)
mousePressed(e: MouseEvent): void
mouseReleased(e: MouseEvent): void
mouseDragged(e: MouseEvent): void
setColor(color: Color): void
setShape(shapeType: String): void

Figure 4. UML class diagram on the client side

The client module includes several components. The **JoinWhiteBoard** class and the **CreateWhiteBoard** class are the entrances of the program for users and the manager respectively. The **ClientController** class receives the users or the manager's operations and handles the operations by calling the RMI of the server. It also implements the methods defined in the remote interface so that the server can call the RMI of the client as well. The **ClientGUI** class displays the UI elements and the **DrawArea** class is one part of the GUI component, which represents the whiteboard on the window. The **DrawListener** class binds the whiteboard and is responsible for listening to the mouse event on the whiteboard so that further drawing can be done. The **Shape** class is the abstraction of the shape drawn on the whiteboard. The DrawArea class needs to store the list of shapes drawn on the whiteboard to repaint them when the whiteboard is refreshed.

2.3 Server Side

The UML class diagram of the client side is shown in the figure below.

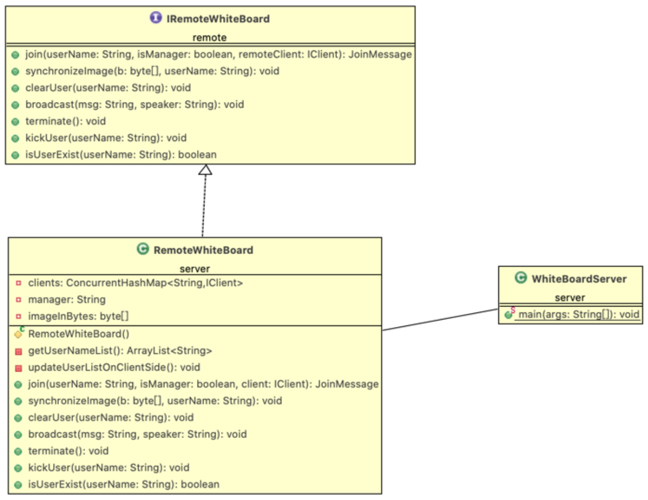

IRemoteWhiteBoard
remote
join(userName: String, isManager: boolean, remoteClient: IClient): JoinMessage
synchronizeImage(b: byte[], userName: String): void
clearUser(userName: String): void
broadcast(msg: String, speaker: String): void
terminate(): void
kickUser(userName: String): void
isUserExist(userName: String): boolean
RemoteWhiteBoard
server
clients: ConcurrentHashMap<String,IClient>
manager: String
imageInBytes: byte[]
RemoteWhiteBoard()
getUserNameList(): ArrayList<String>
updateUserListOnClientSide(): void
join(userName: String, isManager: boolean, client: IClient): JoinMessage
synchronizeImage(b: byte[], userName: String): void
clearUser(userName: String): void
broadcast(msg: String, speaker: String): void
terminate(): void
kickUser(userName: String): void
isUserExist(userName: String): boolean
WhiteBoardServer
server
main(args: String[]): void

Figure 5. UML class diagram on the server side

The server module consists of three parts. The **WhiteBoardServer** class is the entrance of the server program. The **IRemoteWhiteBoard** is the remote interface for the server and the **RemoteWhiteBoard** implements this interface so that the clients can call the RMI of the server. The RemoteWhiteBoard class also stores all the client instances in a **ConcurrentHashMap** to achieve high concurrency for updates.

3. Communication Protocols and Message Formats

Since the system uses the RMI to deal with networked communication, the remote interfaces should be designed well. The following two tables show the RMI interfaces of the server and the client.

Table 1. RMI interface of the server

Method	Description
JoinMessage join(String username, Boolean isManager, IClient remoteClient)	Join the whiteboad and return whether the join is succeessful
void synchronizeImage(byte[] b, String userName)	Synchronize one client's whiteboard image to all the other clients
void clearUser(String userName)	Remove a user from the user list
void broadcast(String msg, String speaker)	Broadcast the message to all the clients
void terminate()	Terminate the application when the manager leaves
void kickUser(String userName)	Kick a user out of the white board application
boolean isUserExist()	Check if a user is currently editing the whiteboard

Table 2. RMI interface of the client

Method	Description
void initGUI()	Initialize the GUI window
void updateUserList(ArrayList<String> userList)	Update the user list on client's GUI
boolean approveJoin(String userName)	Notify the manager that someone wants to join and let the manager approve or disapprove the join request
void showChatMessage(String msg, String userName)	Display the message on chat window
void close(CloseType closeType)	Close the GUI window
void loadImage(byte[] b)	Load the image from the server

In the operation of joining the whiteboard, there are many types of results. The types of application close also differ. The two tables below show the enumeration values in the two situations.

Table 3. Enumeration values of join operation's result

Enumeration value	Description
SUCCESS	Successful join
REFUSED	Refused by the manager
NO_MANAGER	The manager has not created the whiteboard yet
MANAGER_EXIST	Join as the manager while the manager has already existed
INVALID_NAME	The user name is duplicated

Table 4. Enumeration values of the types of closing the window

Enumeration value	Description
SELF_CLOSE	The user closes its window
KICKED_OUT	Window close due to being kicked out by the manager
MANAGER_LEAVE	Window close due to manager's leave
SERVER_CRASH	Window close due to server's crash

4. Implementation Details

Details of the process of a user joining and synchronizing the whiteboard's image on other users' canvas are introduced in this section. The figure below shows the sequence diagram of how a user joins into the whiteboard program.

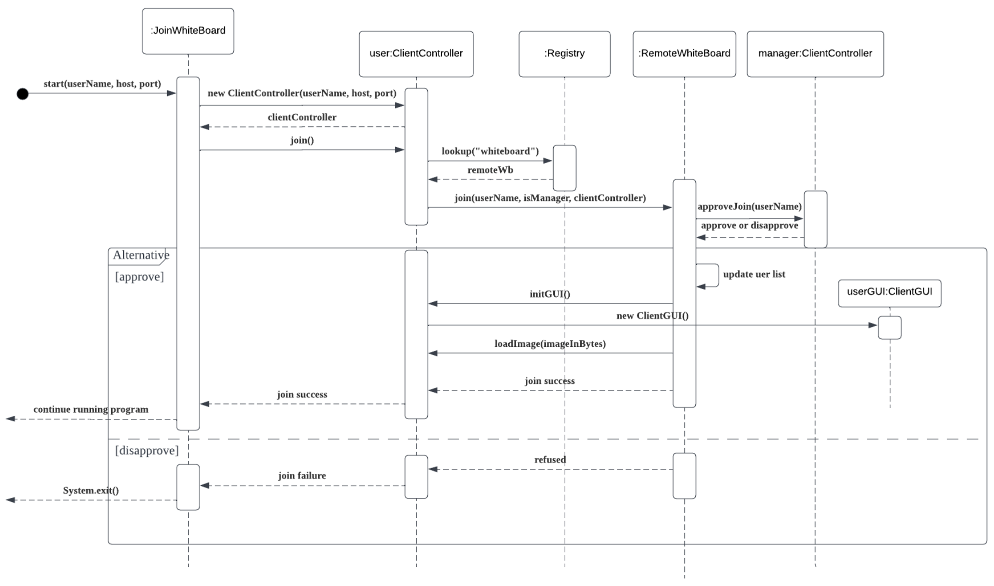


Figure 6. Sequence diagram of the user join

First, the client controller will be created it will look up the remote reference bound in the registry based on the host and port provided. Then, it will call the remote join method on the server side. The server will then ask the manager to approve the join request by calling the client's RMI method **approveJoin.** If the manager approves the request, the server will add this new user to the user list and help initialize the user's GUI and load the current image of the whiteboard on the user's GUI as well. Then the server will return the successful join message to this joined user. If the manager disapproves the request, the server will return the message of joining failure, and the program on the client side will terminate.
The figure below shows how one user's whiteboard image is shared with peers in real-time.

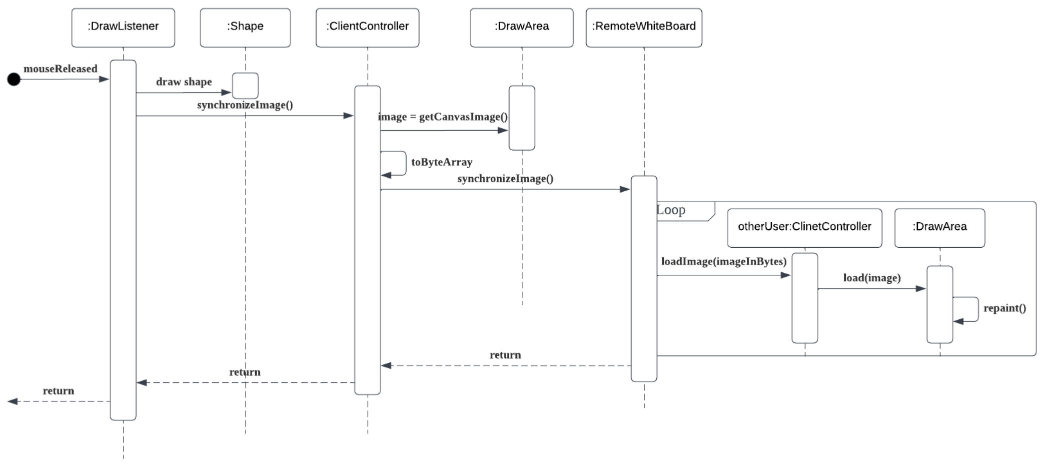


Figure 7. Sequence diagram of synchronizing the whiteboard image to peers

Once the DrawListener class detects that the mouse is released on the whiteboard, the shape will be first drawn locally. Then, the client controller will get the real-time image of the whiteboard and convert the buffered image to a byte array in order to convey the image data to the server. After the server receives the image data, the server will call all the other clients to load the latest image.

5. Advanced Features

Apart from the basic features of the whiteboard, some advanced features are designed and implemented. First, there is a chatting area and users can text with others. All users' texting information will be displayed in a chatting box. Second, the manager can kick out a user by entering the user's name. The application on that user's side will terminate then. Finally, the manager can clean the whiteboard and create a new one.

6. Critical Analysis

5.1 Advantages of the system

1. The system uses a centralized server to handle requests from clients. One server means that operations can be handled more easily.
2. Java's RMI is used in the system. We can pass full objects as arguments through RMI, not just predefined data types. Therefore, the implementation of the system becomes much easier. Besides, RMI is multi-threaded. This means the server can process client requests concurrently.

5.2 Disadvantages of the system

1. The whiteboard updates its status by loading a BuffedImage from the server. Therefore, it is hard to undo the painting and the painting area has to be fixed in size.
2. Despite the easy request handle, the centralized server will make all clients fail to communicate with each other once the server crashes. Besides, it may make slow response if the number of clients increases.

5.3 Conclusion

The distributed shared whiteboard is implemented based on the Client-Server architecture. One centralized server handles clients' requests. The data exchange is implemented by using Java's RMI of both client side and server side.