THE UNIVERSITY OF MELBOURNE

COMP90015

Distributed Whiteboard Application

October 25, 2019

Che-Hao Chang 1040445
Hai Ho Dac 1050369
Eldar Kurmakaev 1028326
Isaac Pedroza 1004268
Maxim Priymak 216213

Contents

1 Introduction

In this project, we are tasked with implementing a collaborative whiteboard application that can be shared between clients. A collaborative whiteboard allows multiple users to draw simultaneously on a shared canvas, as well as broadcast messages to each other in order to facilitate this collaborative creative effort.

The whiteboard will be shared by several users, each of which is going to be connected to a central server via the Java RMI (Remote Method Invocation) framework. The implemented network topology is that of a star schema: several remote clients are connected to a centralized messaging server with which they synchronize the state of their User Interfaces (UI). Two-way communication between server and client is enabled, with all messages being sent via RMI.

The first user to connect to the whiteboard server is granted the Manager role for that particular session. This role authorizes the user to receive join requests of subsequent users who wish to connect to the same session. The manager may choose to either accept and grant this connection request or to decline (bouncing the requesting user back to the start screen). Additionally, the Managers role allows access to administrative tools which allows the manager to kick a user, clear a whiteboard, save/save as the current session as an image file, and load a previously-saved session or any supported image format.

Any of the authorized and connected users may be 'kicked' from the whiteboard by the manager at any time. Additionally, all connected users will receive a notification when the manager of the current session quits the application. In the event that the manager of the current session quits, the Managerial authority is transferred to the next user based on seniority (i.e., the user next in the connection queue is picked). When the last user of the current session logs out, the server terminates the current Whiteboard session. After termination of the session, the server remains running and waiting for further incoming connections in order to initialize a new whiteboard session and assign its manager. The manager may choose to transfer administrative privileges to another user at any time. Only one Manager may exist per single Whiteboard session.

Each of the connected users has free access to modify the whiteboard by adding shapes/lines/text, broadcasting chat messages to the shared chat room, and sending private messages to any user. However, all administrative tools are disabled for non-Manager users.

2 Architecture

2.1 Client-Server Model

The application is designed along with the client-server architecture – the server hosting the RMI registry for the whiteboard acts as a centralized server, and the board users represent the clients. Clients communicate all canvas events and administrative messages via the centralized server. Essentially, the server acts as a Message-Oriented Middleware (MOM), insulating the clients from the heterogeneous nature of the underlying distributed system and network interfaces, while allowing to access an API that provides a distributed communications layer. The server is accessed by clients to add objects to the canvas, and the server commits all of these updates to the canvas that is stored in the main memory as a buffered image. Therefore, the server keeps the most current copy of the canvas in its main memory. When a client adds an object to the local canvas, this object is immediately sent to the centralized server to be broadcast to all connected users such that their local whiteboards can be kept in a state that is consistent with the server. The server synchronizes all clients by disseminating all canvas updates (as Shape objects). Every time a new user connects to an ongoing whiteboard session, the most current state of the whiteboard is polled from the server.

Communication via a centralized server while avoiding direct inter-client (i.e., P2P) communication yields an application design that is generally simpler to implement. A client-server architecture has the additional benefit of avoiding many of the consistency, timing, and concurrency issues (which would be unavoidable if clients had the ability to disseminate messages/objects b/w each other directly).

2.2 RMI Distribution Framework

RMI is the technology chosen to facilitate two-way communication b/w the clients and the server. Remote method calls allow the client to submit whiteboard updates and chat messages to the server for synchronization with other users. Exporting stubs of the remote client interface to the server during connection allows the server to callback the client via RMI calls at any time in order to update the whiteboard.

RMI was picked over Java sockets for the client-server communication for the following reasons:

- In comparison to socket programming, RMI provides unparalleled transparency and simplicity since RMI abstracts many of the messaging and communications details like error handling, object marshaling and unmarshaling, etc. RMI operates via the TCP/IP protocol, thereby providing all of the benefits of this protocol (i.e., guaranteed message delivery, error handling, in-order delivery, etc.). This enables RMI communication b/w remote resources to become transparent to the application programmer and ensures that remote services are accessed as easily as local method calls.
- Compared to Enterprise Java Beans (EJB), Web Services, and RESTful, RMI offers a protocol with a much lighter overhead and higher performance. This ensures that RMI introduces a much lighter overhead and results in better real-time interactivity for applications that rely on such interactivity for their Quality of Service (QoS) (such as a distributed whiteboard).
- RMI library is already incorporated within the Java Development Kit distribution and is supported by the Java virtual machine (JSE). This obviates the necessity for separate installation of third-party packages on either the server or the client. This important advantage emphasizes application portability – the server and client may be launched on any Java virtual machine.
- RMI allows us to abstract away much of the multi-threading (both client and server-side) that would typically be required in order to meet the application's QoS. The RMI itself decides whether to spawn separate threads for method invocations to remote objects. The only task of the application programmer is to ensure that all remote objects are thread-safe and can provide for the requisite concurrency.
- RMI easily supports the direct transfer of serialized Java objects and has the inbuilt capability for Distributed Garbage Collection (DGC).
- Stub methods behave exactly like local methods, except that they also marshal and unmarshal incoming method arguments. This makes interaction b/w server and client more intuitive.

3 Server Design

The server is responsible for managing the clients currently connected to it and for the distribution of the whiteboard drawing and chat messages. The server also listens for incoming client connections.

3.1 Server Class Diagram

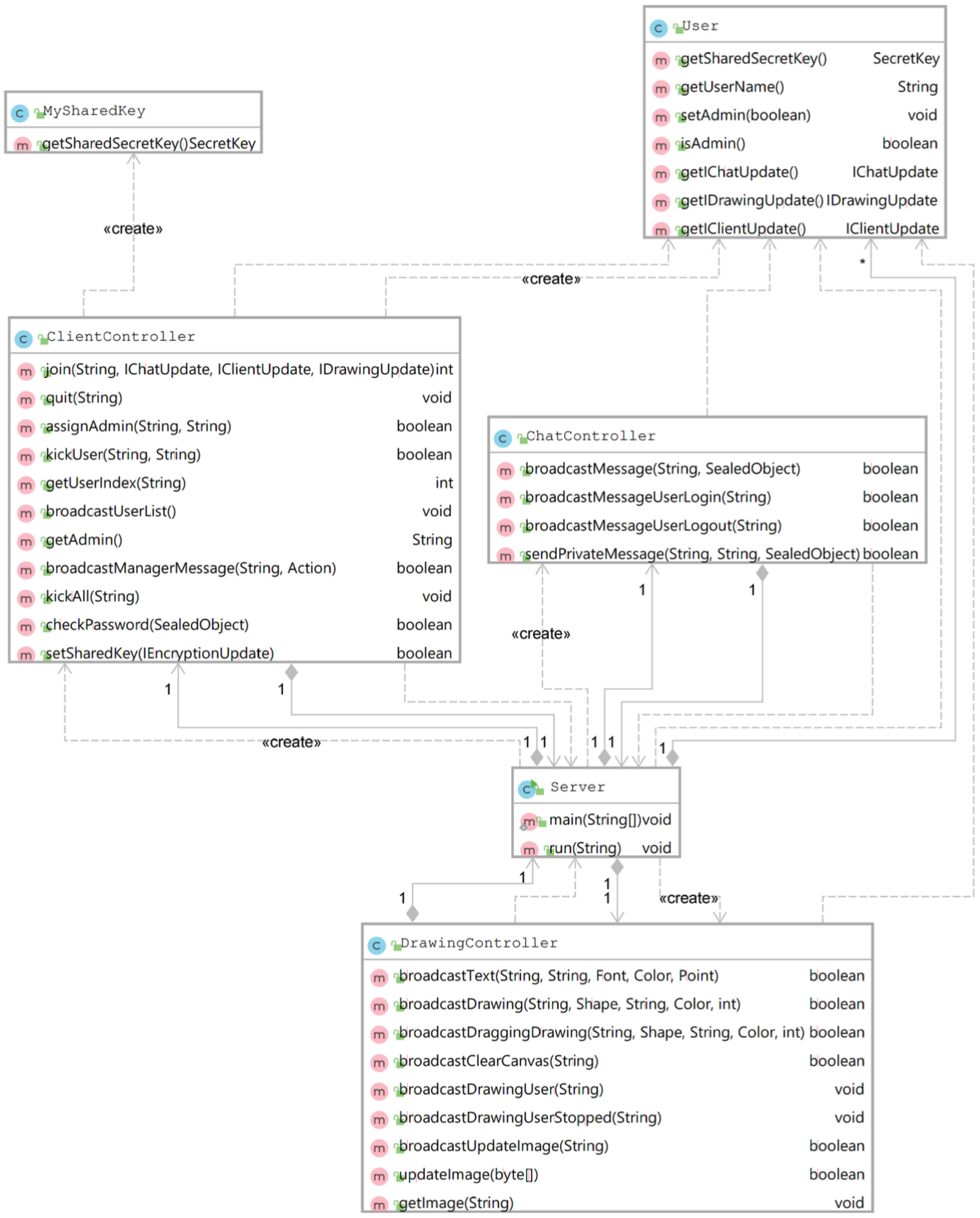

User
getSharedSecretKey() SecretKey
getUserName() String
setAdmin(boolean) void
isAdmin() boolean
getIChatUpdate() IChatUpdate
getIDrawingUpdate() IDrawingUpdate
getIClientUpdate() IClientUpdate
MySharedKey
getSharedSecretKey() SecretKey
«create»
«create»
*
ClientController
join(String, IChatUpdate, IClientUpdate, IDrawingUpdate) int
quit(String) void
assignAdmin(String, String) boolean
kickUser(String, String) boolean
getUserIndex(String) int
broadcastUserList() void
getAdmin() String
broadcastManagerMessage(String, Action) boolean
kickAll(String) void
checkPassword(SealedObject) boolean
setSharedKey(IEncryptionUpdate) boolean
ChatController
broadcastMessage(String, SealedObject) boolean
broadcastMessageUserLogin(String) boolean
broadcastMessageUserLogout(String) boolean
sendPrivateMessage(String, String, SealedObject) boolean
«create»
«create»
1
1
1
1
1 1
1 1
1
Server
main(String[]) void
run(String) void
1
1
1
1
«create»
DrawingController
broadcastText(String, String, Font, Color, Point) boolean
broadcastDrawing(String, Shape, String, Color, int) boolean
broadcastDraggingDrawing(String, Shape, String, Color, int) boolean
broadcastClearCanvas(String) boolean
broadcastDrawingUser(String) void
broadcastDrawingUserStopped(String) void
broadcastUpdateImage(String) boolean
updateImage(byte[]) boolean
getImage(String) void

Figure 1: The class diagram of the Server package.

3.2 Server Classes

3.2.1 Server

Starts a server and instantiates all of the requisite controllers. Additionally, it stores a collection of all the currently connected users as well as references to ClientController, ChatController, and DrawingController objects. This class also creates an RMI registry that listens on the default port 1099 and binds the remote objects to the RMI registry (so that the clients are able to obtain the stubs of the remote objects). The server class also allows getting/setting the server password for a particular session. All of the controllers extend UnicastRemoteObject classes (to allow publishing to the RMI registry) and implement Serializable interfaces (to allow marshaling and unmarshalling of interfaces during RMI calls). Additionally, all controllers implement their respective interfaces, which outline the public methods that should be accessed by the clients during client-server communication. All interfaces extend the Remote interface to enable the generation of interface stubs.

3.2.2 ClientController

Implements basic functionality for client management. It allows clients to connect to the server, to quit the server, for Manager role to be assigned to any clients (only allowed to be accessed by a manager), to kick a particular user (also limited to the manager), to check whether entered password is valid, to broadcast manager message, and to retrieve the name of the current manager. Additionally, helper methods allow retrieval of user index based on username and broadcasting the user list of currently connected users to everyone.

3.2.3 ChatController

Allows all connected users to broadcast chat messages to all connected users, as well as send private chat messages to a specified user.

3.2.4 DrawingController

Contains the method to broadcast a new shape that has been drawn on a canvas to all currently connected users.

3.2.5 User

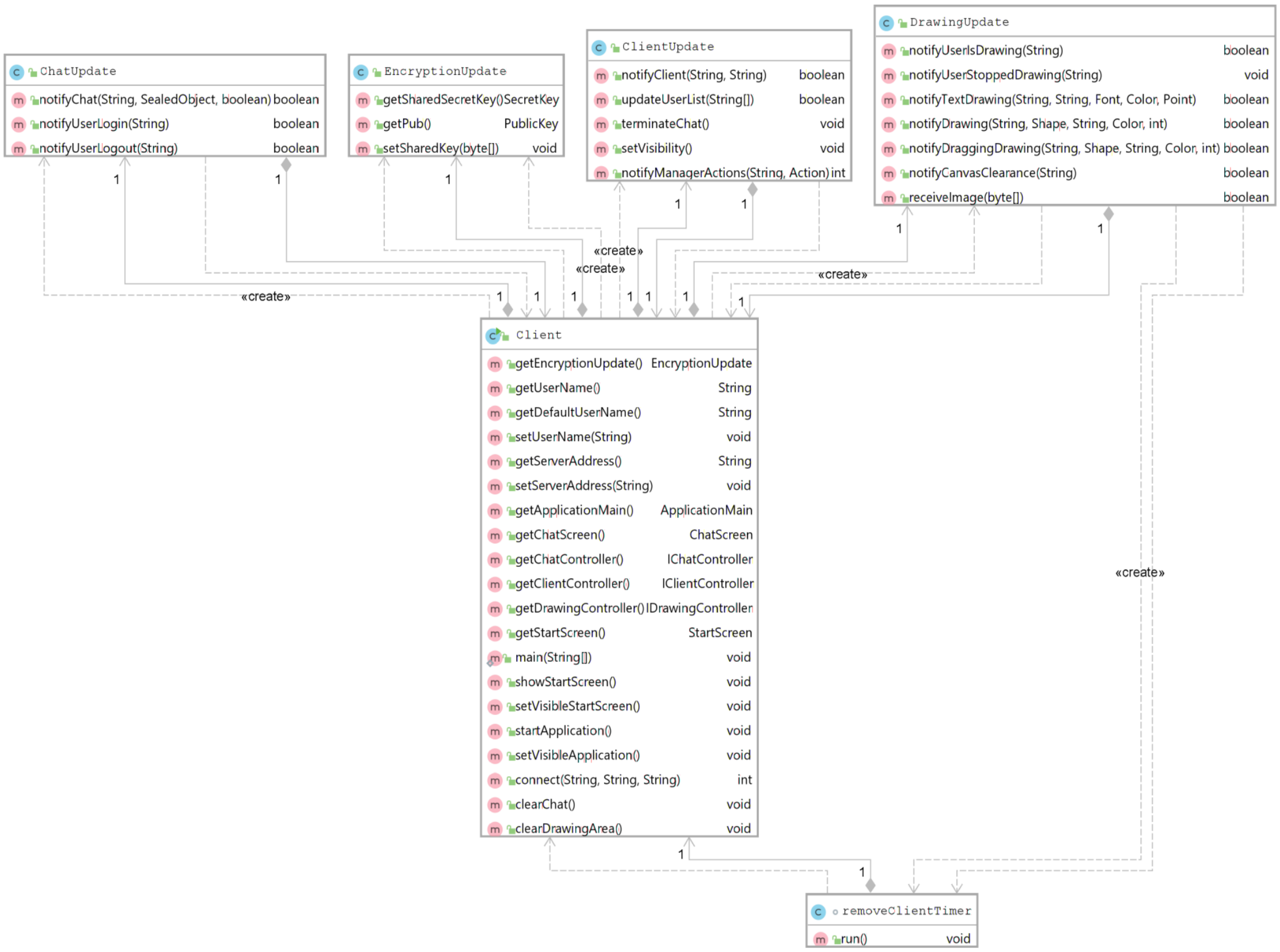
User class is used to store the relevant information of all connected users, such as username, a flag signifying whether a user is a session Manager, and references to the IChatUpdate, IClientUpdate, and IDrawingUpdate stubs for that particular client.

4 Client Design

In our implementation, the first user to create a new whiteboard session assigns the password for the session in order to prevent unauthorized users from accessing the whiteboard. This user automatically becomes the manager for the session and may create a New Whiteboard file, Save (and Save As) the current WhiteBoard as either a PNG or JPG image, Open a previously created image, and Clear the whiteboard content. In addition to full access to edit the whiteboard, the manager accepts/rejects connection requests by other clients, may kick out connected users, and transfer administrative privileges to any other connected user.

All non-manager clients can place a request to the manager in order to join an existing whiteboard session and use most whiteboard features (including public/private messaging, drawing). However, non-manager users are restricted from accessing the menu toolbar and the administrative tools.

4.1 Client Class Diagram



ChatUpdate
notifyChat(String, SealedObject, boolean) boolean
notifyUserLogin(String) boolean
notifyUserLogout(String) boolean
EncryptionUpdate
getSharedSecretKey() SecretKey
getPub() PublicKey
setSharedKey(byte[]) void
ClientUpdate
notifyClient(String, String) boolean
updateUserList(String[]) boolean
terminateChat() void
setVisibility() void
notifyManagerActions(String, Action) int
DrawingUpdate
notifyUserIsDrawing(String) boolean
notifyUserStoppedDrawing(String) void
notifyTextDrawing(String, String, Font, Color, Point) boolean
notifyDrawing(String, Shape, String, Color, int) boolean
notifyDraggingDrawing(String, Shape, String, Color, int) boolean
notifyCanvasClearance(String) boolean
receiveImage(byte[]) boolean
«create»
«create»
«create»
«create»
«create»
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
Client
getEncryptionUpdate() EncryptionUpdate
getUserName() String
getDefaultUserName() String
setUserName(String) void
getServerAddress() String
setServerAddress(String) void
getApplicationMain() ApplicationMain
getChatScreen() ChatScreen
getChatController() IChatController
getClientController() IClientController
getDrawingController() IDrawingController
getStartScreen() StartScreen
main(String[]) void
showStartScreen() void
setVisibleStartScreen() void
startApplication() void
setVisibleApplication() void
connect(String, String, String) int
clearChat() void
clearDrawingArea() void
removeClientTimer
run() void

Figure 2: The class diagram of the Client package.

4.2 Client Classes

4.2.1 Client

Starts an instance of a client, instantiates all of the contained classes and controllers, such as ClientUpdate, ChatUpdate, DrawingUpdate, StartScreen, and ApplicationMain. The class contains helper methods to display the startscreen, connect to the server, and to start the main application if the connection is successful.

4.2.2 ClientUpdate

Contains methods that are invoked remotely by the server to update the client with the list of the currently connected users, as well as to inform the client if an Administrative action (such as kicking a user or promoting a user) has taken place.

4.2.3 ChatUpdate

Contains all of the methods that are called by the server in order to update the chat window of the client with either public/private messages from other clients, or to inform the client whether someone has joined or left the current session.

4.2.4 DrawingUpdate

Receives notification from the server regarding any new updates that have been committed to the whiteboard.

5 GUI Design

The GUI (Graphical User Interface) was implemented by utilizing AWT, Swing, and Java2D components.

AWT (Abstract Window Toolkit) is a set of APIs that are used to create GUIs for Java applications. AWT is part of Java Foundation Classes (JFC). The drawback of AWT is platform-dependence (i.e., GUI objects are dependent on the specific windowing system of a particular OS – also referred to as ‘heavyweight components’).

Swing is a GUI widget toolkit for Java and is also part of JFC. Swing provides a more sophisticated set of GUI components than AWT and allows the applications to have a look and feel that is unrelated to the underlying platform (i.e., ‘lightweight components’).

Java2D is an API for drawing 2D graphics using the Java programming language (also part of the JFC). Every Java2D drawing operation can ultimately be treated as ‘filling’ a ‘shape’ using a ‘paint’ and ‘compositing’ the result onto a screen. Additionally, the core rendering functionality used by Swing to draw its ‘lightweight’ components is provided by Java2D.

The usage of Java2D for the whiteboard necessitates the usage of the Shape class to store and disseminate updates to the canvas.

5.1 GUI Class Diagram

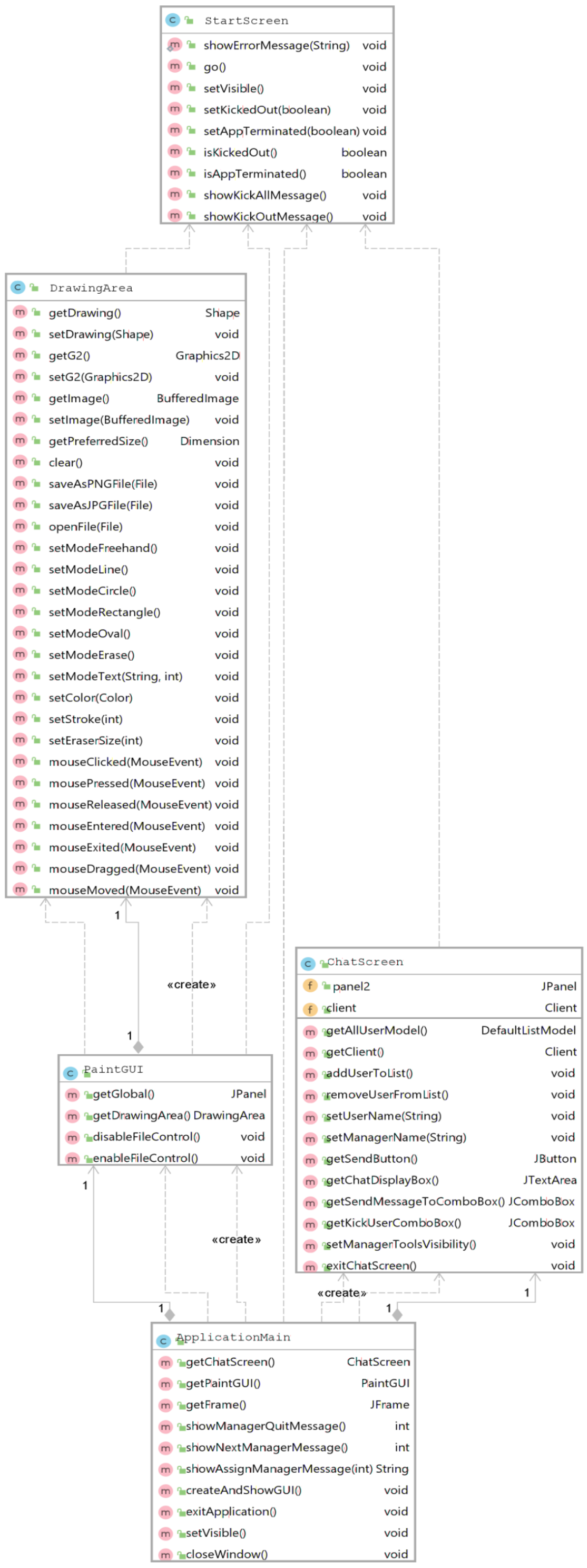

StartScreen
showErrorMessage(String) void
go() void
setVisible() void
setKickedOut(boolean) void
setAppTerminated(boolean) void
isKickedOut() boolean
isAppTerminated() boolean
showKickAllMessage() void
showKickOutMessage() void
DrawingArea
getDrawing() Shape
setDrawing(Shape) void
getG2() Graphics2D
setG2(Graphics2D) void
getImage() BufferedImage
setImage(BufferedImage) void
getPreferredSize() Dimension
clear() void
saveAsPNGFile(File) void
saveAsJPGFile(File) void
openFile(File) void
setModeFreehand() void
setModeLine() void
setModeCircle() void
setModeRectangle() void
setModeOval() void
setModeErase() void
setModeText(String, int) void
setColor(Color) void
setStroke(int) void
setEraserSize(int) void
mouseClicked(MouseEvent) void
mousePressed(MouseEvent) void
mouseReleased(MouseEvent) void
mouseEntered(MouseEvent) void
mouseExited(MouseEvent) void
mouseDragged(MouseEvent) void
mouseMoved(MouseEvent) void
1
«create»
1
PaintGUI
getGlobal() JPanel
getDrawingArea() DrawingArea
disableFileControl() void
enableFileControl() void
1
«create»
ChatScreen
panel2 JPanel
client Client
getAllUserModel() DefaultListModel
getClient() Client
addUserToList() void
removeUserFromList() void
setUserName(String) void
setManagerName(String) void
getSendButton() JButton
getChatDisplayBox() JTextArea
getSendMessageToComboBox() JComboBox
getKickUserComboBox() JComboBox
setManagerToolsVisibility() void
exitChatScreen() void
«create»
1
1
1
ApplicationMain
getChatScreen() ChatScreen
getPaintGUI() PaintGUI
getFrame() JFrame
showManagerQuitMessage() int
showNextManagerMessage() int
showAssignManagerMessage(int) String
createAndShowGUI() void
exitApplication() void
setVisible() void
closeWindow() void

Figure 3: The class diagram of the GUI package.

5.2 GUI Classes

5.2.1 StartScreen

The welcome screen that is displayed to any client that starts the client application for the first time. It contains all of the necessary action listeners. It also contains a public static method to display a custom error notification. The function of the start screen is to prompt the user for a username, password, server IP address, and to call the connect method in the main client with the relevant information. StartScreen is initialized within the Client instance using a constructor.

5.2.2 ApplicationMain

Combines the ChatScreen and PaintGUI objects into one such that they can be displayed together as one GUI. All of the relevant action listeners are implemented in the ChatScreen and PaintGUI classes.

5.2.3 ChatScreen

Defines how the chat screen appears and functions. It contains a method to restrict the visibility of Manager tools to unauthorized clients, as well as all of the relevant action listeners. Users are able to send and receive messages from other users via this object.

5.2.4 DrawingArea

This is the canvas that users can draw on and share their drawings. All the drawing actions will be recorded on a BufferedImage that will then be shared with every user in the room. The DrawingArea class realise all the drawing functionalities with a MouseEventListener and a drawing tool of Graphic2D class. All the drawing related elements (i.e., colour, font, stroke) are defined in this class as well. This class also implemented the image export function for saving the current canvas into .png file or .jpg file.

5.2.5 PaintGUI

Contains the drawing canvas with all of the relevant whiteboard functionality and event listeners. Users may draw, erase, and type text into the canvas. The shapes are drawn using the Java2D shapes method, with each shape being a separate Shape object with properties such as x- and y-coordinates, Height, and Width. When the canvas is updated, the Shape object and its properties are disseminated to all users via the server. PaintGUI generates a tool selection toolbar that presents the color choices, stroke choices, drawing options, and an eraser to the user. The toolbar allows the user to invoke a colour palette to choose among a multitude of colours. The drawing options include shapes like Line, Circle, Oval, Rectangle, Square, as well as text. An eraser is also available. Eraser and Shape stroke sizes may be selected. Also, PaintGUI generates a canvas menu that is accessible only to the manager. The menu allows the manager to Save/Save as the current canvas as either a PNG or JPG image files, to clear the entire canvas of all objects, and to open a previously saved canvas image to continue editing.

6 RMI Implementation

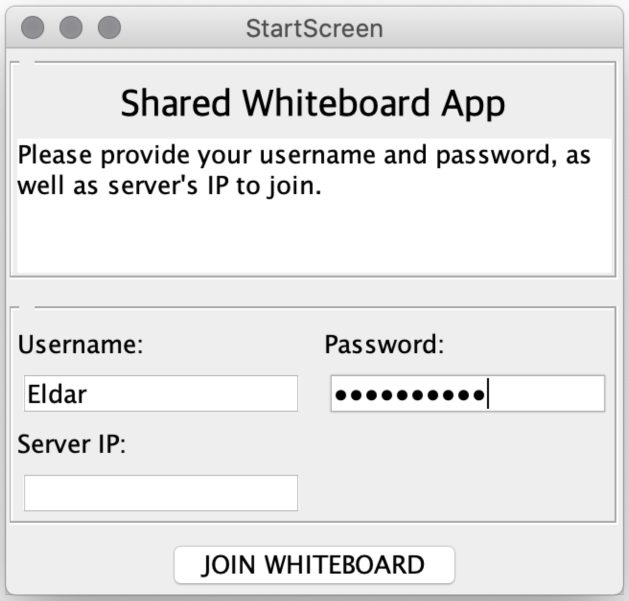
RMI is implemented by placing all of the client and server interfaces into a single package. This package is accessed by both the client and the server in order to obtain interface stubs. All of the two-way communication b/w the server, and the client utilizes these interfaces.

Java RMI supports at-most-once invocation, with several fault tolerance measures implemented, such as retransmission of request message, duplicate filtering, and retransmission of reply. For the distributed whiteboard application, the RMI is implemented as follows:

1. The server binds a reference to a remote object in its RMI registry.
2. A client looks up this remote object and stores it as a client-side stub/proxy.
3. The client connects to the server by invoking a join() method on the ClientController stub – this passes relevant user information to the server together with exported stubs for all client-side controllers.
4. The server stores the stubs and the user information in an ArrayList of custom objects – this allows the server to callback the clients via RMI at any time to update chat/client/drawing controllers.
5. Subsequently, each client can message the server via the stubs retrieved from the RMI registry on the server, whereas the server can callback each individual client by accessing the exported client-side controller stubs stored for each user.

7 Screenshots

The following subsection will present some of the snapshots of the applications. All snapshots will assume that the connected client is a session Manager and therefore has full access to tools related to Whiteboard administration tasks.


StartScreen

Shared Whiteboard App

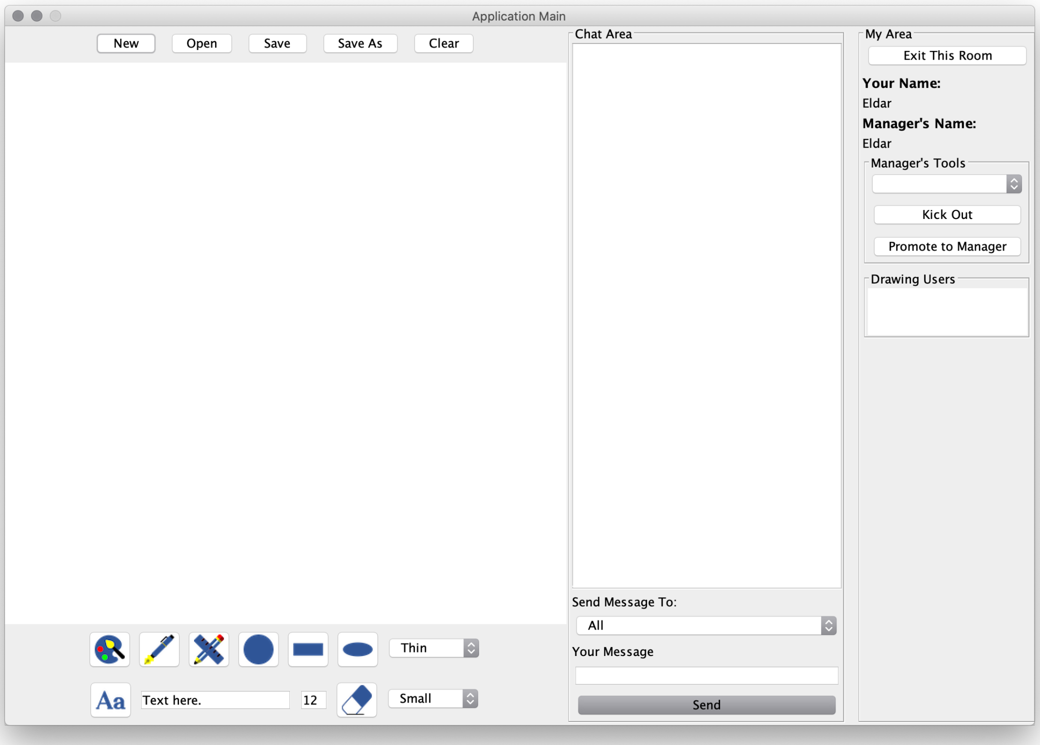
Please provide your username and password, as well as server's IP to join.

Username: Eldar

Password: ●●●●●●●●●●

Server IP:

Figure 4: The login screen that is visible to any new user.



Your Name:
Eldar
Manager's Name:
Eldar

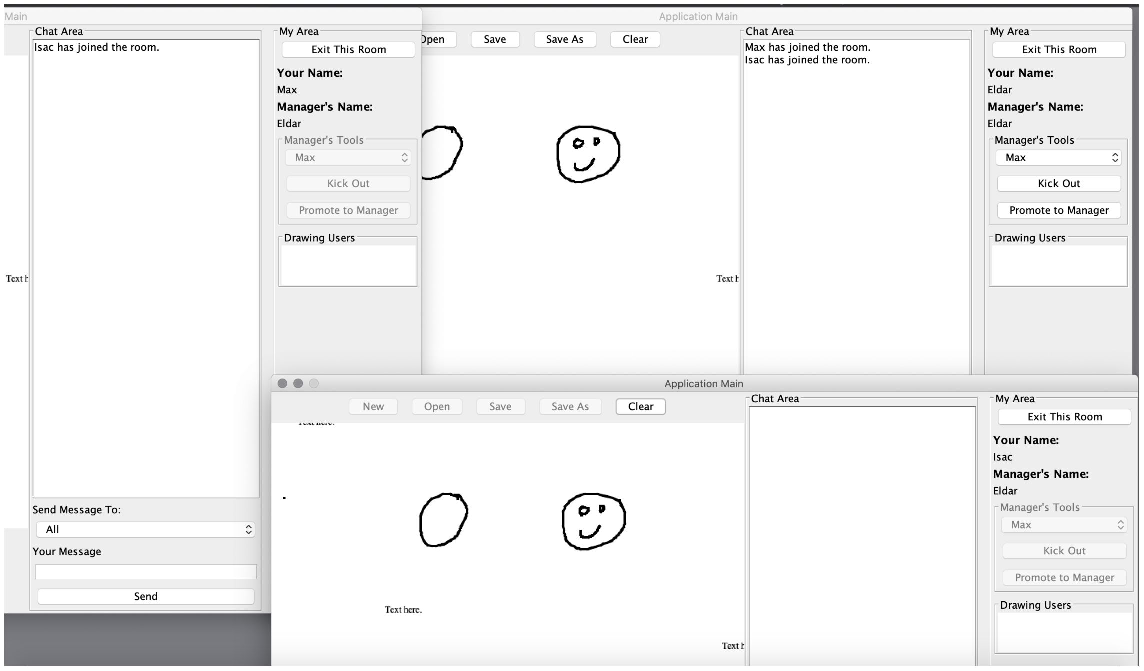
Send Message To: All

Thin

Text here. 12

Small

Figure 5: The main application GUI.



Isac has joined the room.

Your Name:
Max
Manager's Name:
Eldar

Max

Send Message To: All

Max has joined the room.
Isac has joined the room.

Your Name:
Eldar
Manager's Name:
Eldar

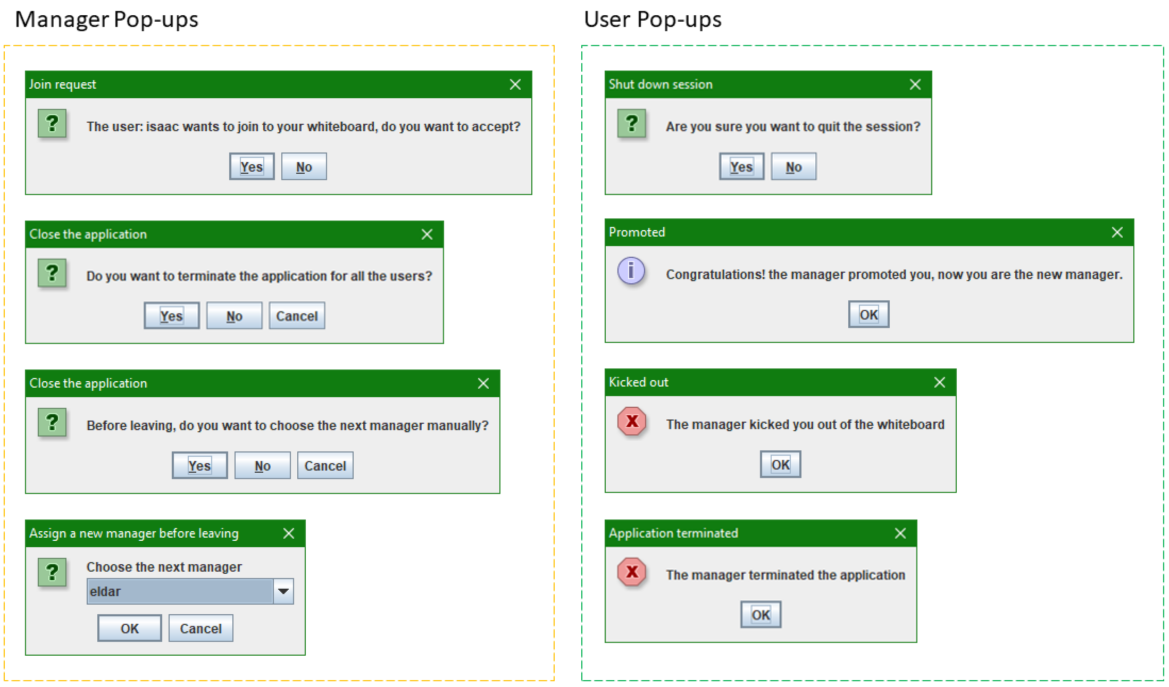
Max

Your Name:
Isac
Manager's Name:
Eldar

Max

Text here.

Figure 6: Screenshot of several collaborating users.



Join request

The user: isaac wants to join to your whiteboard, do you want to accept?

Close the application

Do you want to terminate the application for all the users?

Close the application

Before leaving, do you want to choose the next manager manually?

Assign a new manager before leaving

Choose the next manager

eldar

Shut down session

Are you sure you want to quit the session?

Promoted

Congratulations! the manager promoted you, now you are the new manager.

Kicked out

The manager kicked you out of the whiteboard

Application terminated

The manager terminated the application

Figure 7: Notification pop-ups for clients and managers.

8 Sequence Diagrams

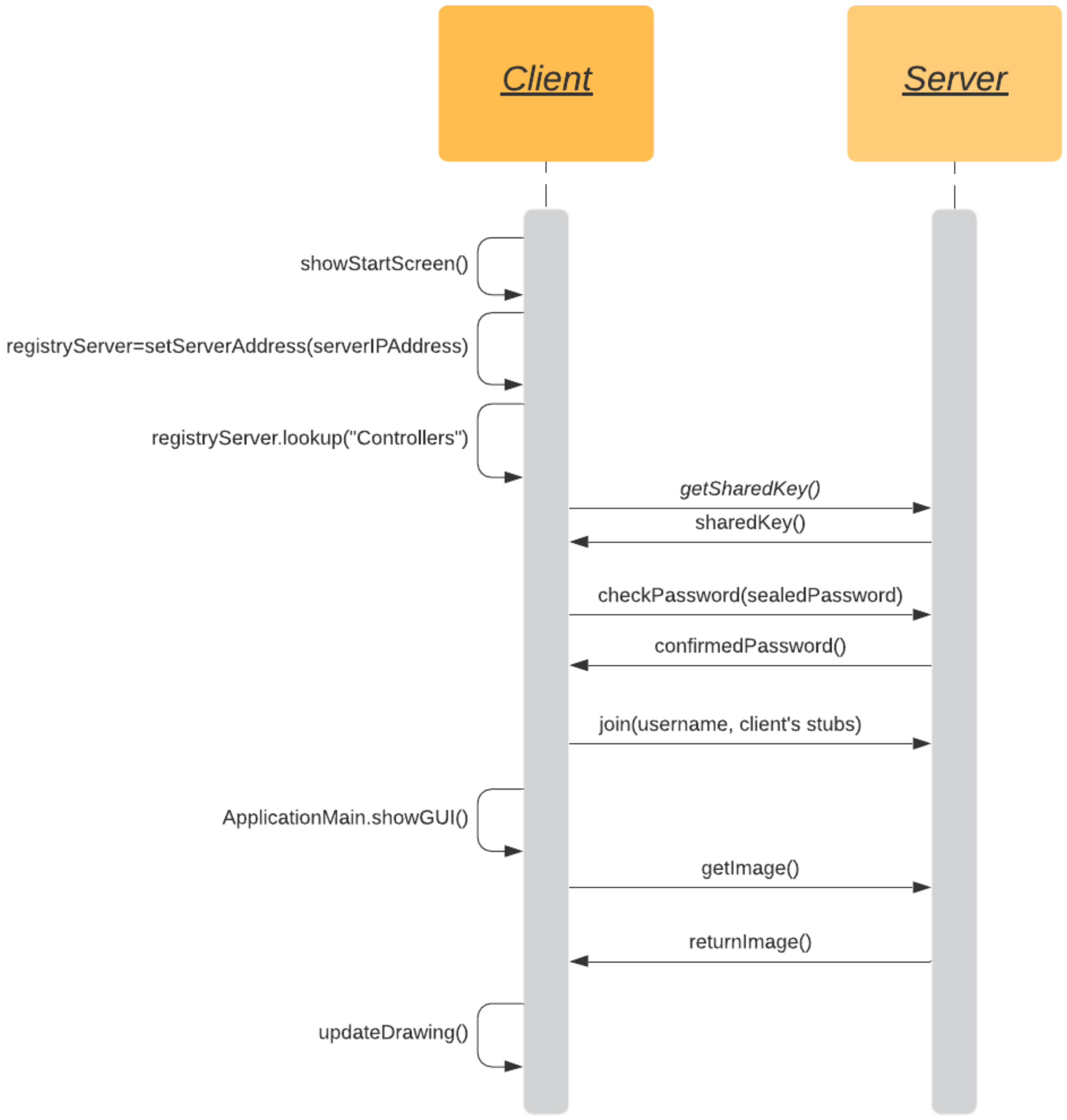

Client
Server
showStartScreen()
registryServer=setServerAddress(serverIPAddress)
registryServer.lookup("Controllers")
getSharedKey()
sharedKey()
checkPassword(sealedPassword)
confirmedPassword()
join(username, client's stubs)
ApplicationMain.showGUI()
getImage()
returnImage()
updateDrawing()

Figure 8: Sequence diagram for connecting a client to the server.

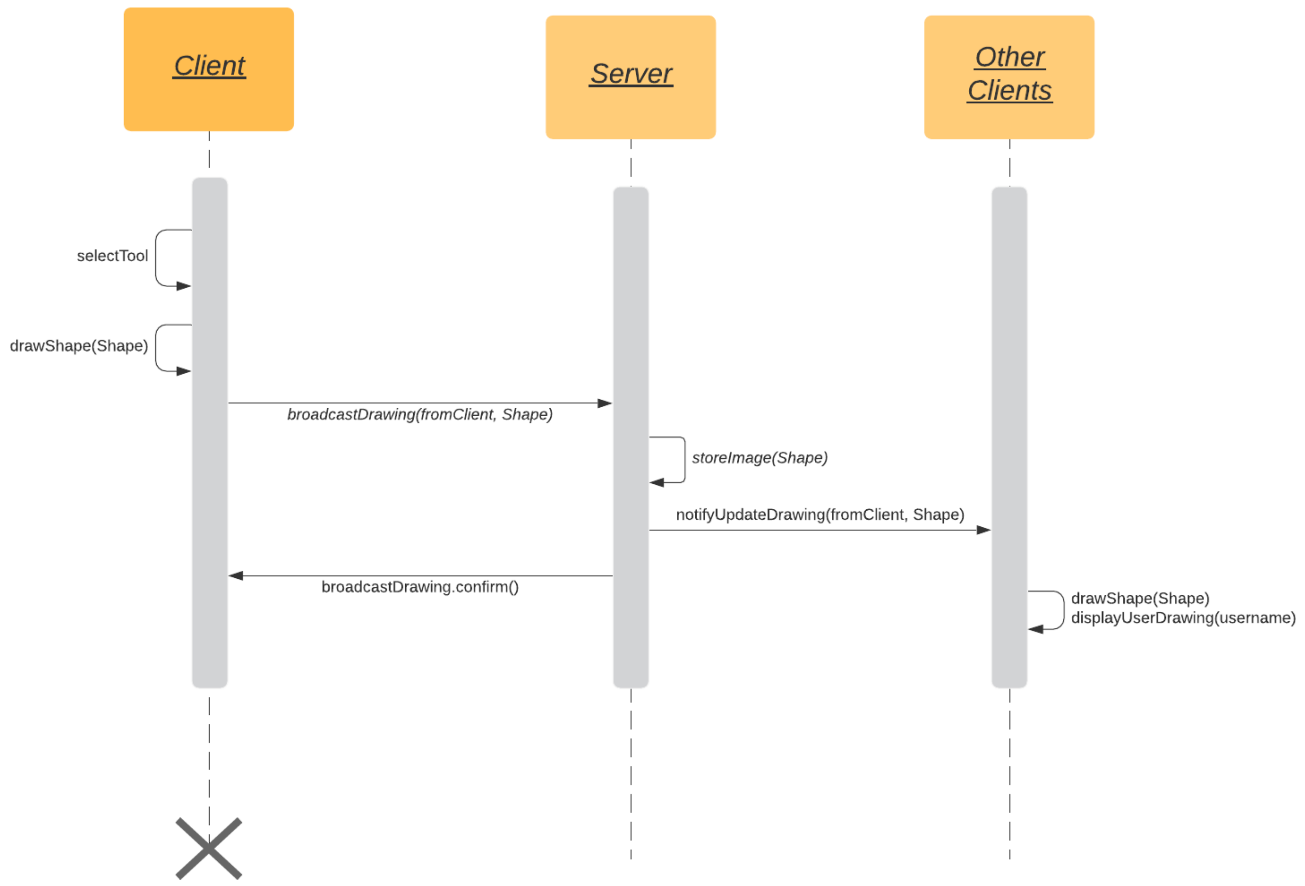

Client
Server
Other Clients
selectTool
drawShape(Shape)
broadcastDrawing(fromClient, Shape)
storeImage(Shape)
notifyUpdateDrawing(fromClient, Shape)
broadcastDrawing.confirm()
drawShape(Shape)
displayUserDrawing(username)

Figure 9: Sequence diagram for disseminating canvas updates.

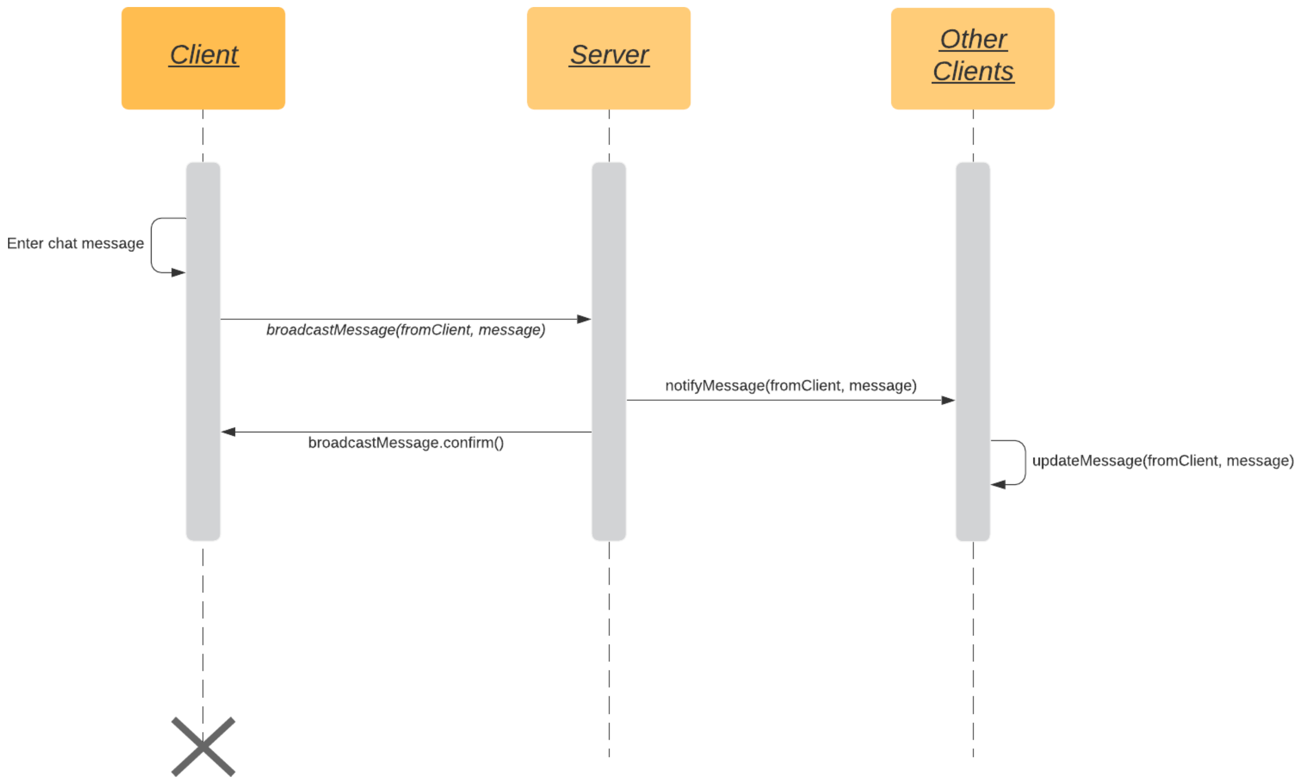

Client
Server
Other Clients
Enter chat message
broadcastMessage(fromClient, message)
notifyMessage(fromClient, message)
broadcastMessage.confirm()
updateMessage(fromClient, message)

Figure 10: Sequence diagram for broadcasting chat messages.

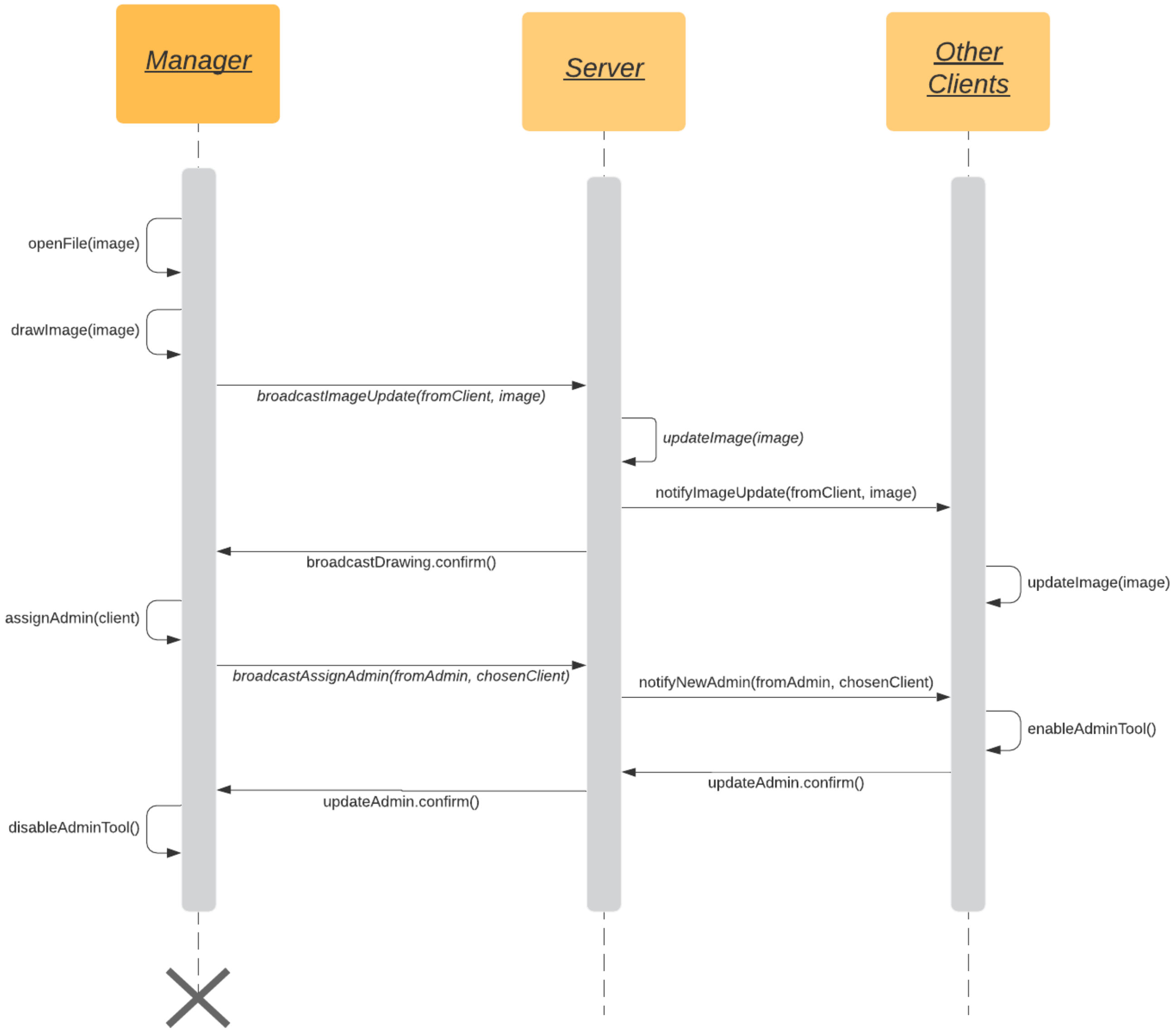


Figure 11: Sequence diagram for administrative actions.

9 Creativity and Innovation

Beyond the basic requirements of the project, our group has implemented the following additional functionality:

1. The application has a log in screen for the user to insert the username, password and IP address of the server.
2. Authorization of clients via passwords. The user who first initiates a whiteboard session becomes the Manager and sets up the password for the session. All subsequent users are required to provide a valid password to be admitted into the session.
3. Implementation of pop-ups to notify the manager and the users about errors and special events such as:
   - When a user is promoted to be the next manager.
   - When the manager kicks out a user.
   - When a user wants to quit.
4. The manager can promote another user to take the manager role at anytime the application is active.
5. The manager is allowed to do 3 different actions when leaving from the application:
   - Select the next manager manually.
   - Let the application select the next manager randomly.
   - Close the application for all users.
6. A user is allowed to send a private message to a specific user.
7. Exchange of shared key using public private key pair. EncryptionUpdate class is used to generate public private key pair on the client side. The public key is sent to the server. The server generates a shared key using SecretKeySpec class, stores it, wraps shared key using public key and transmits to client. Clients use the private key to unwrap the shared key and stores it.
8. Using Sealed Object class to wrap and using a shared key to encrypt the password and chat communication. Server and clients use custom EncryptDecrypt class that provides method to wrap and unwrap strings to encrypt and decrypt String for password exchange and chat private and public communication.
9. Displaying the username of the user who is currently actively editing the whiteboard canvas.
10. Enable every collaborator to access a wide colour gamut for the shapes that are added to the canvas.
11. The ability for the Whiteboard Manager to save the whiteboard as either a PNG or JPG image.
12. A time-stamped running log on the server that records every vital interaction b/w clients and server (I.e. user connections, Managers actions, etc).

10 Contributions

All members of the group have worked tirelessly to implement this distributed Whiteboard application. Every group member was tasked with several non-overlapping responsibilities in order to avoid redundancy. Ultimately, however, due to the large scope of the project, every member of the group cooperated with and aided each and every other member of the group, and therefore contributed to each other's work.

The following list briefly outlines the primary tasks that each group member was occupied with during the implementation of the distributed Whiteboard application:

- **Eldar (20%):** Designed much of the Graphical User Interface, implemented public/private key encryption protocol for symmetric key exchange and encryption of messages b/w the client and server, implemented the interface to display the users who are actively updating the canvas.
- **Isaac (20%):** Implemented most of the functionality behind user management. Implemented the function for the manager to kick out other users, to join by getting approval from the manager, the function that allows the manager to choose the next manager before leaving the application, all the functionality of quitting and closing the application, and the notifications by different pop-ups.
- **Hai (20%):** Designed the communication skeleton for server-client drawing via RMI, worked on optimizing the Menu bar (New, Open, Save/Save as, Clear) functionalities, helped implement saving and loading of images by manager and synchronization of canvases during new user login, optimized the concurrency of drawing and synchronization of canvas update among all clients.
- **Max (20%):** Contributed to the design of the RMI communication framework and interfaces, implemented some of the controllers and their respective interfaces, worked on the back-end logic, start screen and password authorization, designed the general application skeleton, conducted debugging of the application during the design phase, designed icon images that are used in the GUI, and composed the final report.
- **Che-Hao (20%):** Designed the drawing mechanism, implemented the Whiteboard's drawing functionalities (freehand, shapes, dragging, text drawing, eraser), whiteboard image input/output system with different formats (.png/.jpg), GUI implementation for drawing area and file I/O system, optimization of RMI communication of Whiteboard drawing updates, troubleshooting RMI registry setup and RMI communication, general debugging.

Acknowledgements

The group would like to thank their friends and family for understanding (and tolerating) the prolonged coding/debugging sessions that each group member had to undertake for this project.

Additionally, the group would like to recognize the help of Haowen Tang in graciously finding the time to provide invaluable aid to the group during a particularly stressful session of wrangling with RMI.