Distributed Whiteboard (Documentation)

Vincent Kurniawan
The University of Melbourne

Problem Context

1.1 Introduction

This report addresses the choice of design, implementation, and analysis of a client-server architecture model of a distributed whiteboard among shared users, allowing them to draw shapes, send text messages, as well as view active users concurrently interacting with the whiteboard.

1.2 Functional Requirements

For this project, two types of users exist, namely managers and users. All managers and users should be able to perform the following operations:

1. Draw re-sizable basic shapes for line, circle, oval and rectangle on the whiteboard
2. Input text on the whiteboard
3. For a chosen shape, a minimum of 16 color options can be chosen to draw the shape with
4. Use a chat window to send text messages to all other user's connected to the whiteboard
5. View a list of currently active users on the whiteboard

On the other hand, a manager has an additional capability to:

1. Use a file menu to clear a whiteboard, open an existing file, save the file by default, save the file based on location, and close the whiteboard
2. Kick out a particular user from the whiteboard

1.3 System Architecture Overview

The system adopted a server-client architecture, where a single server exists and interacts with clients (i.e., users and managers) to handle data access and storage. Here, the server acts as a storage hub for managers and users to access the contents of the whiteboard. In particular, this system utilized Remote Method Invocation (RMI) from Java to handle invocations of methods remotely belonging to the whiteboard objects between user-applications, allowing users to perform concurrent operations on them. RMI servants of the whiteboard includes the whiteboard canvas, a group chat, and an active user list.

On top of RMI, the system implemented Transmission Control Protocol (TCP) via sockets to establish inter-communications among the users and managers through the server. To execute this, a thread-per-request architecture was implemented, where a new thread is created to execute the requests. Furthermore, a graphical User Interface (GUI) has also been implemented for users and managers to interact with the whiteboard. In addition, any errors and responses associated with a particular request is visually displayed on each respective client GUIs. An implementation to display and handle failures or errors that may occur throughout server-client interactions is provided by the system.

Implementation Details

2.1 Project Structure

Figure 1. shows the overview structure of the packages and its associated Java classes that are used in this project. The source folder "java" is divided into 6 separate packages: Server, RMI, WhiteBoard, Response, Manager, and User. The use of packages provides better encapsulation, management, and reusability of classes. A resource folder was used to store icons of the file menu for Manager's GUI.

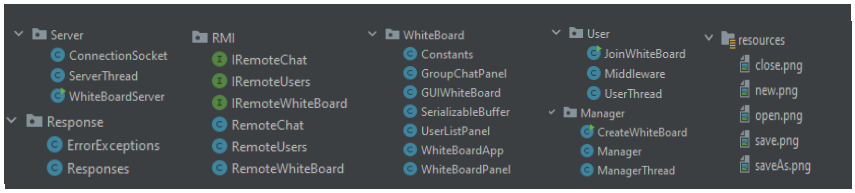

Figure 1: Project structure of system components, including packages and their associated classes

2.2 Server

The Server package contains classes responsible for instantiating and binding RMI servants. Once the "WhiteBoardServer.java" class is instantiated, it binds a socket to the specified <port number> and <server address>, creates an RMI registry with default port 1099, and binds the RMI servants to the registry. In addition, the server is initiated with a manager, assuming that a server can only have a single manager at a time. Furthermore, it uses a thread-per-request loop to handle incoming requests. In particular, it is used to handle authorized access and creation of the whiteboard. A manager is able to create, while a user needs to request access to the manager to join only if a manager already exists beforehand. The whiteboard can also be closed by the manager. For each incoming request, the "ServerThread.java" thread is instantiated and is ready to process the request. Once the server has finished processing the request, it sends a response back to the designated client.

Additionally, a custom TCP socket implementation is used through the "ConnectionSocket.java" class. This class is used by threads to handle data flow using input and output stream of the socket. This adopts a Pure Fabrication pattern to increase cohesion and low coupling among the classes.

2.3 RMI

The RMI package contains the RMI interfaces and servants of the whiteboard objects instantiated by the server, namely "RemoteWhiteboard.java", "RemoteChat.java", and "RemoteUsers.java", each implementing their respective interfaces. Furthermore, each method of these servants contains the logic of the operation at which the users are capable of and throws a remote exception in the event of remote communication errors. The whiteboard canvas implements methods to allow drawing operations, group chat implement methods to allow message exchange, and user list implement methods to handle users coming in and out.

2.4 WhiteBoard

The WhiteBoard package contains classes and panels for the visual and logic of the whiteboard application. Whenever a user/manager attempts to join/create a whiteboard on the server, the "WhiteBoardApp.java" class is instantiated. For unsuccessful attempts, the application sends notification of failure to the client and closes the system. On the other hand, in successful attempts, this class instantiates the "GUIWhiteBoard.java" class which contains panels that handle logic of GUI format display for the whiteboard's RMI objects among clients. These are implemented using Java's Swing for its convenience in

customizing layouts of the GUI such as adding buttons, labels, and text fields and update them accordingly based on client's inputs and events. In particular, there are three main panels, namely "WhiteBoardPanel.java", "GroupChatPanel.java", and "UserListPanel". In addition, the "Constants.java" class contains the information of dimensions for the components within the panels above, as well as existing user types and shapes of the whiteboard.

As shown in the left-hand side of Figures 2 and 3, for both users and managers, "WhiteBoardPanel.java" panel contains the (1) shape selection row, the (2) color grid, and the (3) whiteboard canvas. The highlighted shape and color indicate the client's selected choice. The canvas is implemented with a buffered image that's serialized with the "SerializableBuffer.java" class. This is required as it allows the image data to be transmitted as a byte stream to transfer image from server during RMI access. In addition, the manager application contains the (4) file menu within same panel, allowing users to perform file operations on the whiteboard.

Furthermore, the upper right-hand side of Figures 2 and 3 show a (5) group chat box. Both users and their manager can see the chat history of the current whiteboard session, as well as use a (6) button to send their own messages by inputting the text into a (7) text box. The chat box is a JTextArea component which updates itself with incoming chats using a separate thread. Additionally, the lower right-hand side of Figures 2 and 3 show a user list, displaying the (8) list of active users in a JList on the current whiteboard session and also uses a separate thread. The (9) manager of the whiteboard has an additional (10) kickout button, allowing connected users to be kicked out at the manager's discretion.

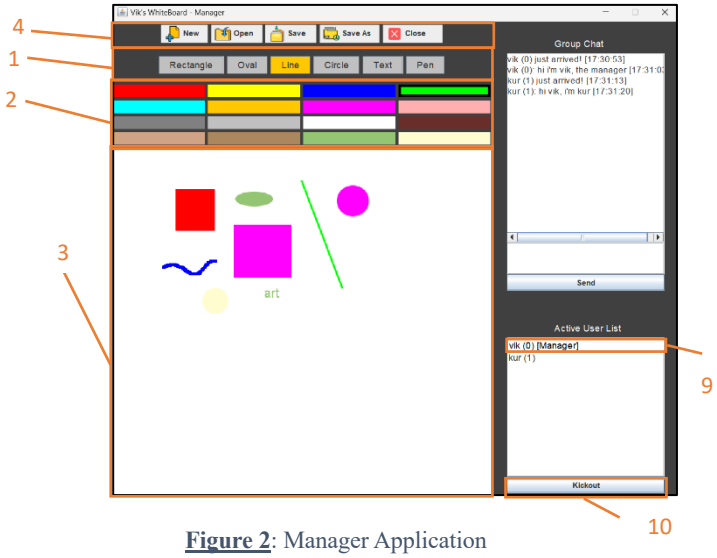

Figure 2: Manager Application

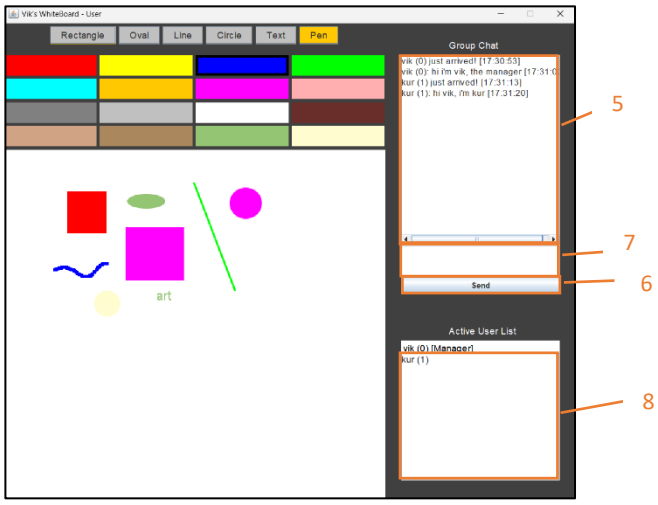

Figure 3: User Application

2.5 Response

The Response package contains the "ErrorExceptions.java" class that holds custom exceptions that can be thrown by the user, manager, and server accordingly depending on the errors caught. An exception may either cause the application to quit with the "showError" method or display a notification to with the "notifyGUI" method. Each exception is supplied with the error code, description, and message; information of all errors that can be thrown by the exceptions provided is shown in Table 1. Furthermore, the package also contains the "Response.java" class that holds the constants that specify the query type of requests sent to the server and the responses sent.

Error Code	*Error Constant*	*Error Description*
400	ARGUMENTS_LENGTH_ERROR	There should only be 2 arguments inputted: <server address> <server port>
401	SERVER_PORT_ERROR	Inputted server port has an invalid format
402	ALREADY_BOUND_ERROR	RMI port is already bounded
403	ACCESS_ERROR	Error in accessing RMI objects
404	BIND_PORT_ERROR	Port provided is currently in used or bind by another application
405	REMOTE_EXCEPTION_ERROR	Error occurred in invoking methods RMI objects
406	PARSE_REQUEST_ERROR	Error in parsing client request
407	CONNECT_REFUSED_ERROR	Server connection is refused
408	UNKNOWN_HOST_ERROR	Unable to connect to specified host/domain name
409	RMI_UNKNOWN_HOST_ERROR	Unable to connect to specified RMI host
410	SERVER_DISCONNECTED	Server connection to user/manager is interrupted
411	EMPTY_INPUT	Inputted keyword cannot be empty
412	RMI_NOT_BOUND	Unable to bind RMI objects to the RMI registry
413	ARGUMENTS_LENGTH_ERROR_USER	There should only be 3 arguments for user/manager inputted: <server address> <server port> <username>
414	NO_MANAGER	User can't join whiteboard without a manager existing
415	REJECT_USER	User's join request is refused by manager
416	DUPLICATE_MANAGER	There can't be 2 managers concurrently on the server
417	KICKOUT	User is kicked-out by manager
418	NO_KICK	Manager can't kick-out a non-existing user

Table 1: List of error codes and its description that can be thrown by an exception at user, manager, and server side

2.6 User

The User package contains the classes associated with the users of the application. Once the "JoinWhiteBoard.java" class is instantiated, a join request to server is sent via sockets, in which this request is relayed to the manager (if one already exists) as shown in Figure 3 below. While waiting, a pop-up shown in Figure 4 is displayed on the user screen. If the manager accepts, the user then instantiates the whiteboard application. Otherwise, the user is sent a declined-to-join pop-up as shown Figure 5. To handle kick-out requests and closing responses from the manager, a separate thread is created to handle such events. Furthermore, a "Middleware.java" class instantiated to establish connection between the application and server.

2.7 Manager

The Manager package contains the classes associated with the manager of the application. Once the "JoinWhiteBoard.java" class is instantiated, a create request to server is sent via sockets. If successful, the manager has access to the whiteboard and can handle join requests from other users wanting to share the whiteboard as shown in Figure 3. Furthermore, a manager can decide to kick-out a user as shown in Figure 6, and a kick-out prompt will be shown to the kicked-out user as shown in Figure 7. Furthermore, the package contains "Manager.java" class which handles the RMI objects of the group chat and user list.

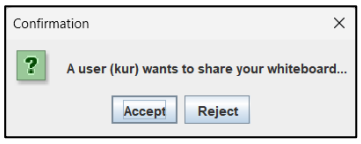

Figure 3: User join request

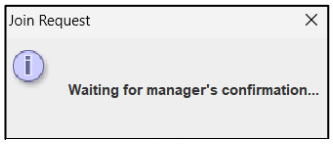

Figure 4: User join request

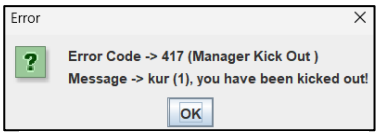

Figure 7: User join request pop-up

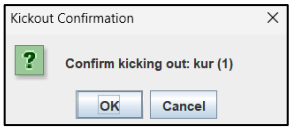

Figure 6: User kick out request

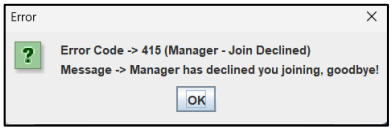

Figure 5: User join request pop-up

2.8 JAR Files

These packages are compiled as three separate JAR executables, namely "WhiteBoardServer.jar" for server, "CreateWhiteBoard.jar" for manager, and "JoinWhiteBoard.jar" for user. To run these JAR files, Figure 8 below shows the required arguments to be inputted in the command line for each JAR file.

```
>java -jar WhiteBoardServer.java <server address> <server port>
>java -jar CreateWhiteBoard.java <server address> <server port> <username>
>java -jar JoinWhiteBoard.java <server address> <server port> <username>
```

Figure 8: Command to run the JAR files

System Diagrams

3.1 Use Case Diagram

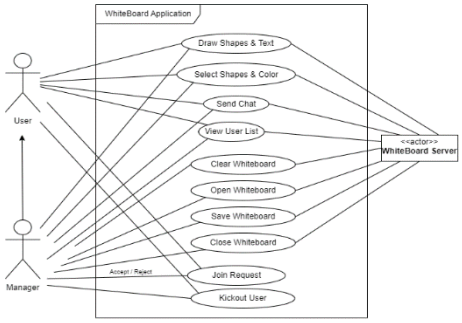

Figure 9 shows a high-level overview of interactions between the user, manager, and server. These interactions include users and managers drawing shapes and text on the whiteboard with selected preferences, sending text to the group chat, and viewing user list. A user must request the manager to join the whiteboard, and the manager can kick out connected users. Furthermore, the manager can perform basic file operations on the shared whiteboard.

Figure 9: Use case diagram of the whiteboard application

3.2 UML Class Diagram

The overall class diagram of the system is shown in Figure 10 below and it is represented using a Unified Modelling Language (UML) model. The model encompasses the instantiation of classes that was previously discussed under the systems components section above.

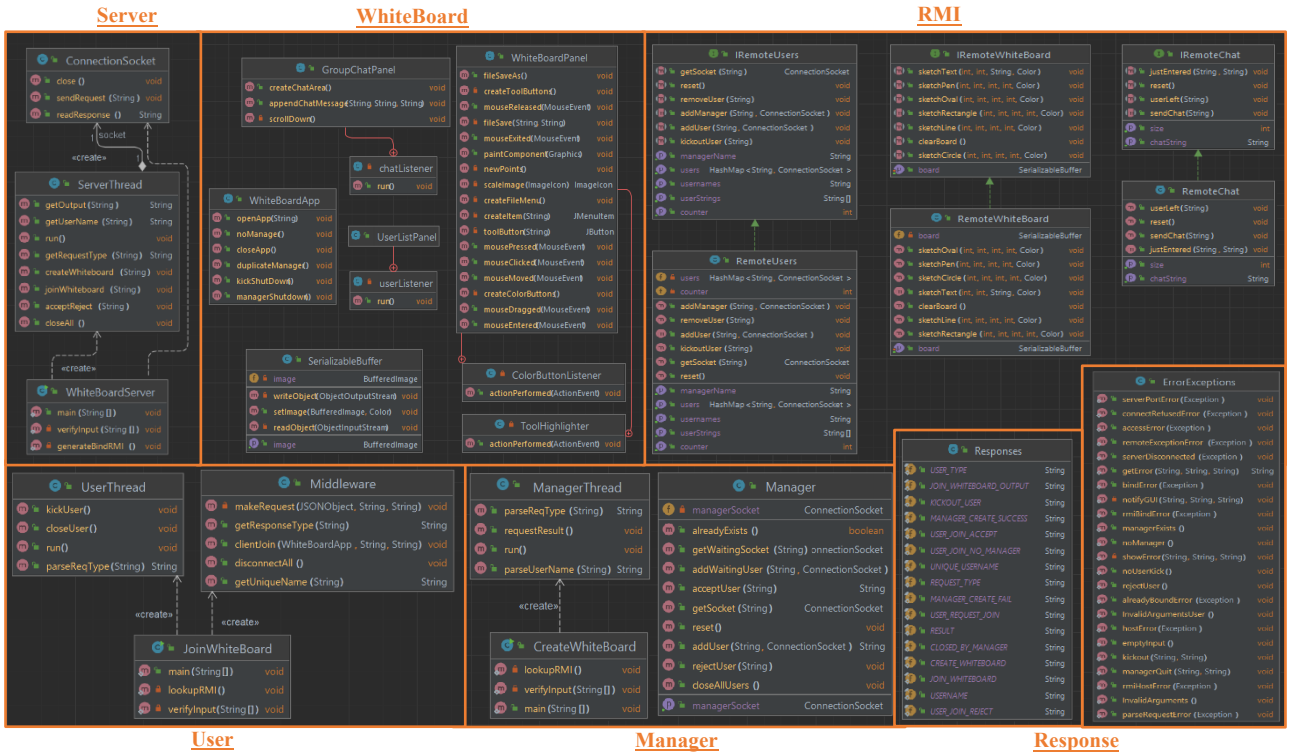

Figure 10: Overall UML class diagram of the system based on the packages organization

3.3 User-to-Manager Request Interaction Diagram

Figure 11 below shows an interaction sequence diagram of the between the manager and a user when the user sends a join request to the manager as shown in Figure 3 above. The response from the manager is sent back to the user through the user's socket.

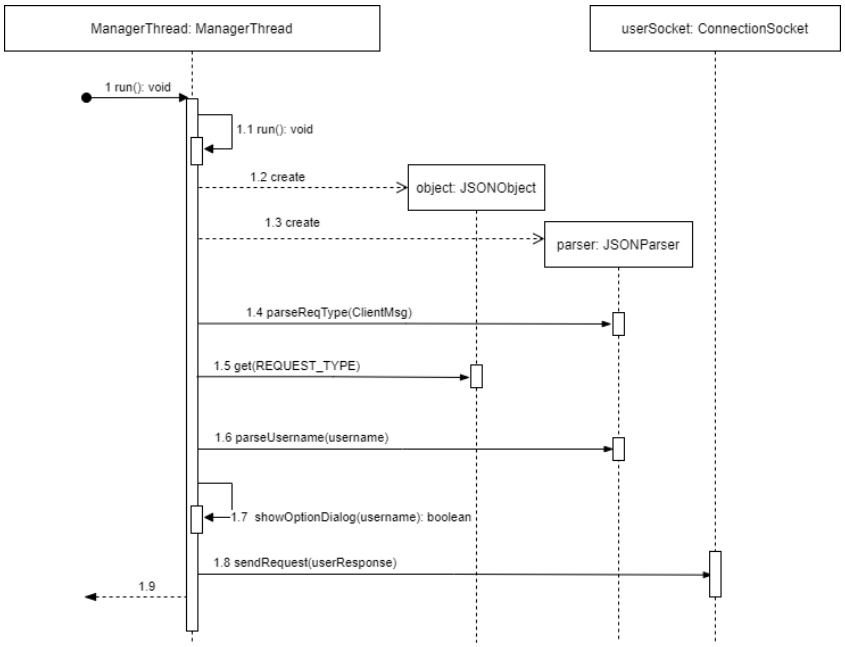

Figure 11: Interaction sequence diagram of user's join request to and response by manager.

3.4 Client-Server Architecture Diagram

Figure 12 below shows the client-server architecture diagram of the system, consisting of a central server that manages system states (i.e., whiteboard canvas, chats, and users). In particular, the server handles permissions between user types (i.e., manager and user) through parsing request types, as well as the method invocations towards the RMI objects instantiated by the server.

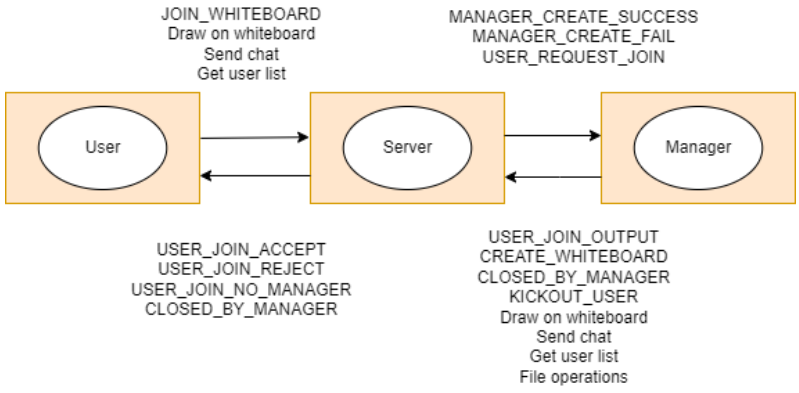

Figure 12: Client-Server architecture of the system

System Architecture Analysis

4.1 TCP & RMI

Communication of the implemented client-server architecture is established via sockets using TCP, a connection-oriented communication protocol that ensures reliable communication and ordered flow of data. On the other hand, RMI is used to allow remote operations to be invoked by user and/or manager to the whiteboard objects. Since the system must handle concurrent invocations, RMI is suitable because it uses multi-threading which allows multiple clients to conduct operations to the whiteboard, as well as propagate modifications on the objects from one peer to another. However, it was noted that the use of RMI induced slight overheads from having to marshal and un-marshalling the RMI objects and having to serialize the buffered image as it is stored remotely instead of being a stand-alone canvas.

In this system, TCP is mainly used to parse incoming requests from managers and users, and send responses according to such requests, while RMI is mainly used to handle concurrent invocations of whiteboard objects. Although either one of these technologies suffices to achieve concurrency, using both utilizes a "Separation of Concerns" principle, which promotes better understandability, maintenance, and extensibility of the system. Hence, the logic to handle permission access and whiteboard operations are separated, resulting in lesser code-bloated classes and interfaces.

4.2 Thread-Per-Request

To execute incoming client requests, the server uses a thread-per-request architecture where a new thread is created to execute each request. In the event where multiple concurrent requests are being sent to the server, creation of more threads is necessary to accommodate the supply and demand for these requests. However, it was noted that these may lead to degrading performance when an overload of requests from multiple clients is received by the server in a short amount of time. In spite of this, due to the system's separated model as discussed in section 4.1, the creation of such threads is mostly affected by the number of clients requesting to join the whiteboard. On the other hand, threads that are responsible for whiteboard operations are handled by RMI, with an additional two threads for each client to constantly oversee incoming chats and users joining.

4.3 Message Exchange Protocol

The system uses JSON as the chosen message exchange data format. This is due to the readily available and easy-to-use JSON library named "json-simple" which provides the fundamental operations that the system requires to parse incoming requests from client, as well as outgoing responses from server to users and the manager.

Conclusion

Overall, the system discussed in this report provides the fundamental functionalities to design and implement a distributed shared whiteboard, allowing a single manager to create a whiteboard and accept or reject incoming users. Users and managers themselves may draw, send messages, and view active users on the whiteboard through the interactive GUIs of the application. This system adopted a client-server architecture that uses TCP to establish reliable client-server communication, as well as RMI to handle concurrent remote invocations on whiteboard objects. Server requests and responses are parsed and sent in JSON format, and the system also adopts a custom failure model that handles display and handling of errors and exceptions.

Excellence Elements

5.1 Error Notifications & Exceptions in GUI

Clear notifications of errors through pop up dialogs in server and client GUIs. Each error is accompanied by a custom error code, description, and message to emphasize further clarity.

5.2 Comprehensive Analysis and Report Structure

Analysis in report highlights the pros and cons of the architecture of the implemented system .Justifications of design and implementation choices are also provided within the analysis. Report is well structured in headings and sections for better readability and understandability.

5.3 Comprehensive Graphs & Figures

Report highlights the overall use case diagram, UML class diagrams, an interaction sequence diagram of a manager handling a user join request, and the adopted client-server architecture. Each graph and figure throughout the report is referenced and provided with relevant explanations.

New Innovations

6.1 Implementation of TCP & RMI

The system adopted both TCP and RMI to handle concurrent transactions, allowing better understanding and maintenance of the system's workings. TCP is more intuitive to handle join/decline requests, while RMI is more intuitive in performing operations on objects.

6.2 Custom "Pen" Drawing Feature

An additional "Pen" shape is implemented to allow free-hand drawing on the whiteboard. Clients can also select a color choice from the color grid to color the pen with.

6.3 Custom Chat Arrivals

In the chat group, clients are notified by the arrival and departure of other clients connected to the whiteboard through the chat box. The time at which such an event occurs is also within the notification.

6.4 Unique User Naming

The system enforces a unique naming scheme for each connected client to the whiteboard. This is done by embedding the current total number of connected users to the end of their usernames.

6.5 Custom Handling of User/Manager Disconnections

The user GUIs are capable of displaying notifications in the event when their manager closes the whiteboard or disrupts their application abruptly. Furthermore, both user and manager are notified whenever the connection to the server is disrupted.

6.6 Server-Manager Reset

When a manager of a user quits, the server is able to reset the manager's group chat and user list, allowing other managers to attempt create a new whiteboard without having to close down the server first.