COMP90015 DISTRIBUTED SYSTEM ASSIGNMENT 2 REPORT

GROUP NUMBER: 77
MEMBER NAMES: Taylor Tang (1323782), Qingyun Wu (1370606)

Introduction

This group assignment aimed to build a collaborative whiteboard application that multiple user could draw (such as line, shape etc.) and chat simultaneously. Each user's interface will be synced. And we also made privilege distributions, including the Manager account and The normal User account. This application is implemented by using Java along with socket connection. The high-level architecture is shown below (diagram 1).

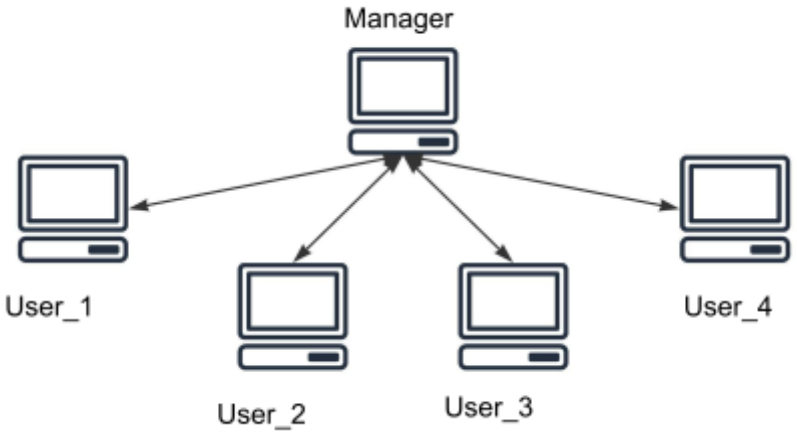

Diagram 1: High-Level Architecture

As shown in the above Figure, the main architecture of this system is that we have multiple users, of which one of them can choose to be the host (manager). While other users can then connect to the Manager's service as clients. Manager's action will be broadcasted to all users directly. All the actions done by a User will be sent to the Manager. The Manager then broadcasts this action to all other users to achieve real-time functionality.

Communication Protocols

We use Socket for chatting communication and other general communication separately, which is an encapsulation of the TCP/IP protocol and acts as an API for communication itself. It represents a communication process between clients. In Socket, we create input and output stream objects to help us communicate.

The advantage of using Socket to communicate is that the transferred data is byte-level, the data is customizable, and the operability is high. And it is efficient and enables better real-time presentation between WhiteBoards. Since we are only passing simple text messages, compared to RMI which is object-oriented, it defines its application protocol on top of the TCP protocol, resulting in some loss of flexibility and performance degradation due to increased transfer packet size. Combining these, we decided to use Socket for message delivery.

Message Formats

Currently, we deliver formatted messages in the form of plain text transmission.

Manager ←→ Users

For this part, every action a user takes will be sent to the manager and the manager will synchronize every action with all other users.

1) For drawing line, circle, triangle, and rectangle:

   Format: **Draw,Type,startX,startY,endX,endY,R,G,B**

   - The **Type** can be [Line, Circle, Triangle, Rectangle];
   - The **startX** and **startY** represent the place where the click starts;
   - The **endX** and **endY** represent the place where the click ends;
   - The **R**, **B**, and **G** are three integers from 0 to 255 to indicate the values of Red, Blue, and Green.

2) For free-drawing:

Format: **Draw,Type,startX,startY,startX,startY,R,G,B**

- The **Type** can be [Line];
- The **startX** and **startY** represent the place where the click starts;
- The **startX** and **startY** represent the place where the click starts;
- The **R**, **B**, and **G** are three integers from 0 to 255 to indicate the values of Red, Blue, and Green.

3) For text input

Format: **Draw,Type,endX,endY,R,G,B,:),Text**

- The **Type** can be [text];
- The **endX** and **endY** represent the place where the click ends;
- The **R**, **B**, and **G** are three integers from 0 to 255 to indicate the values of Red, Blue, and Green;
- The **Text** is the text itself showing on the WhiteBoard.

4) For chat messages:

Format: **Chat,userName,aMessage**

- The **Chat** is the type of message;
- The **userName** is the user who sent this message;
- The **aMessage** is the text message itself.

Manager → Users

For this part, these actions can only be taken by the manager and the manager will send them to relative users.

1) For kicking users, a message is sent to the user who is been kicked.

   Format: **Kick,userName**

   - The **Kick** is the type of action;
   - The **userName** is the user who is kicked by the Manager;

2) For kicking users or users leaving on their own, a message is sent to all other users.

   Format: **RemoveUser,userName**

   - The **RemoveUser** is the type of action;
   - The **userName** is the user who is kicked by the Manager;

3) After a user joins the WhiteBoard, a message is sent to synchronise the current Whiteboard state.

   Format: **Draw,drawRecord**

   - The **Draw** is the type of action;
   - The **drawRecord** is the current drawing record;

4) After the current user list is changed, a message is sent to synchronise.

   Format: **AllUserNames,userList**

   - The **AllUserNames** is the type of action;
   - The **userList** is the current user list;

5) When a user joins the Whiteboard, a message is sent to all users.

Join,userName

- The **Join** is the type of action;
- The **userName** is the user who have joined the WhiteBoard;

Users → Manager

1) When a user requests to join the WhiteBoard.

Format: **NewUser,userName**

- The **NewUser** is the type of action;
- The **userName** is the user who wants to join the WhiteBoard;

2) When a user requests to sync the WhiteBoard.

Format: **RequestUpdate,userName**

- The **RequestUpdate** is the type of action;
- The **userName** is the user who wants to join the WhiteBoard;

Design Diagram

Manager

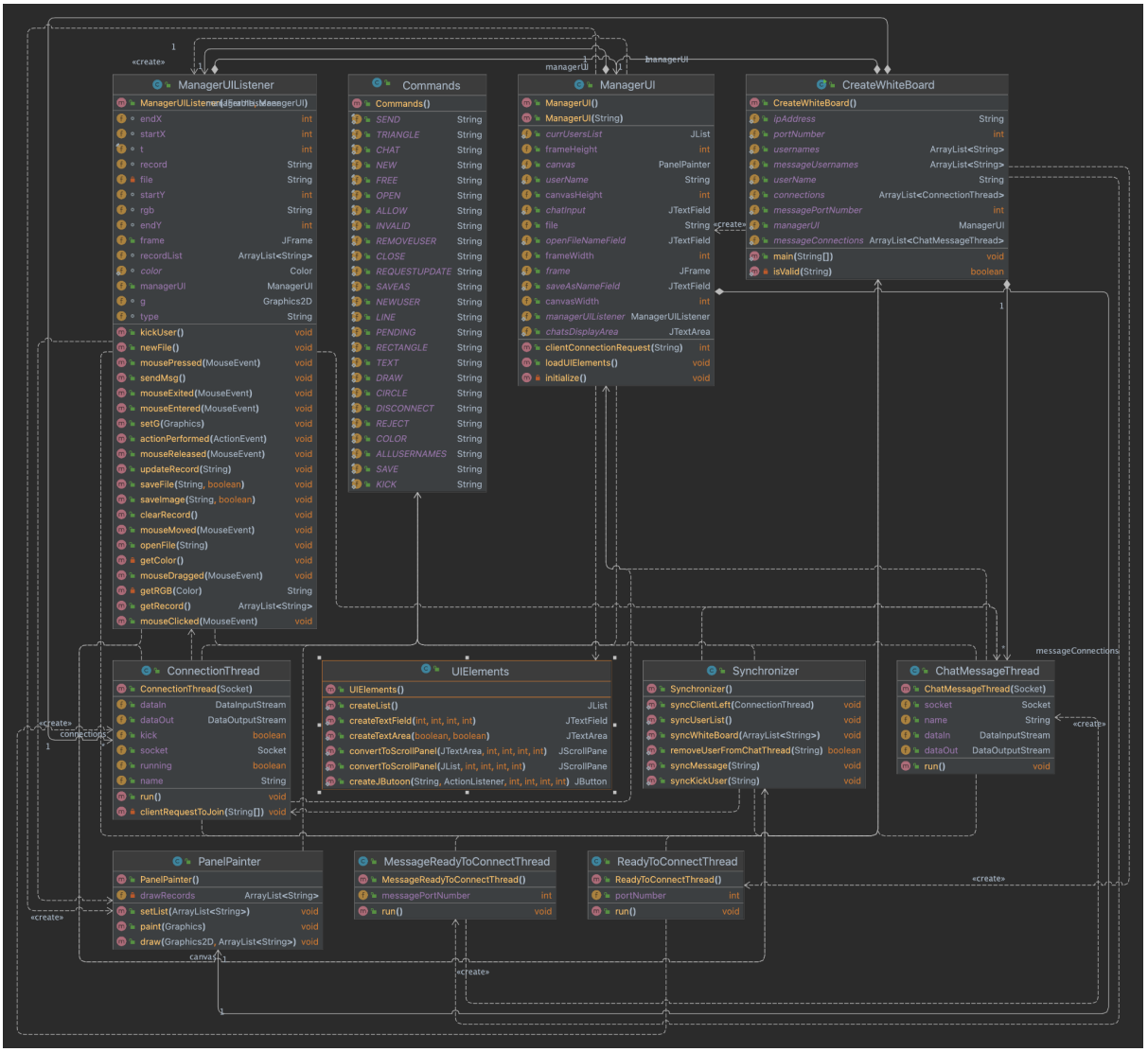

Diagram 2: UML - Manager

User

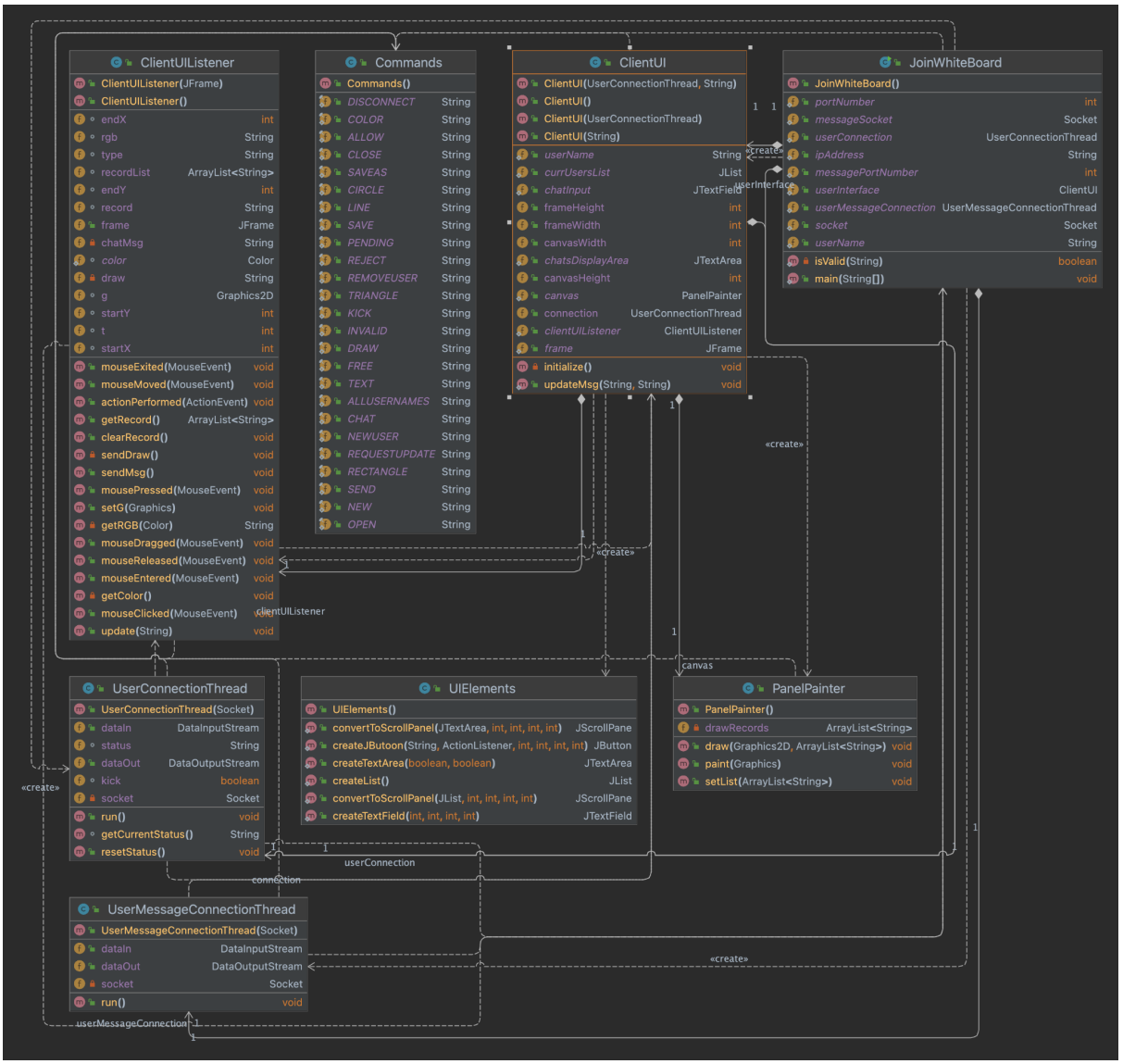

Diagram 3: UML - User

Implementation Details

For the application implementation, it can be roughly divided into three parts (while some parts may overlay with each other in some way): UI Design, Actions Response, and Communication Design.

UI Design

This application utilizes **Java.swing.JFrame** along with other sub-APIs to implement the user interface.

- All buttons are implemented by using JButton with designed parameters.
- All user text input areas are implemented by using JTextArea with some parameters.
- All text-displaying areas are implemented by using JList with some parameters.
- The canvas is implemented by using an inherited JPanel class with Graphics2D as the drawing tool.
- All the above UI elements are then added to the main JFrame Content panel.

Action Response

A inherited class “Listener” from Java.awt.ActionListener / MouseListener / MouseMotionListner is then created to monitor the User’s actions on UI and then act accordingly.

- For buttons, it listens to buttons’ corresponding actions that were designed in button creation. Then pass the action to the corresponding functions.
- For drawing canvas, it listens to the mouse’s actions such as the position when pressed, released, and continuous position when dragged. Then pass the position and actions to Graphics2D for drawing.

Communication Design

1) **Synchronization**
   To achieve synchronization between the manager and users, this application on the manager side, keeps monitoring the communication channel from users and synchronizes other users. The manager acts as the main server of this system.

For drawing, whenever a user (including the manager) draws on the whiteboard, the drawing record is first passed to the manager, then broadcasted to every user (including the drawer) for drawing execution.

For chat messages, whenever a user (including the manager) sends a chat message , the chat record is first passed to the manager, then broadcasted to every user (including the sender) for chat record displaying.

For manager-only actions, such as New, Open, the manager broadcasts this action to all users.

For manager-only actions targeting a specific user, such as Kick, the manager sends the kick command to the specific user and broadcasts this action to other users to synchronize their current user list.

2) **Threads**

Several threads are invoked to work collaboratively for this application.

When calling the command in the terminal to create a manager, one thread is for initialization and UI interface, one thread is for monitoring the user's message communication request, one thread is for monitoring the user's general communication request.

When the manager accepts the user’s requests, one thread is created for the user’s chat message communication, one thread is created for the user’s general communication.

When calling the command to create a client, one thread is created for UI, one thread is created for the user’s chat message communication, one thread is created for the user’s general communication.

3) **IP-address, Port and Usernames**

The manager decides the ipAddress for hosting this application. Two port numbers from the manager side are required for this application. The first one is for chat messages, the other is for general communications. By default the port numbers are 3000 & 3001. The manager can manually configure the port number during initialization by providing one pointed port number; the other port number will be the

first portNumber +1. Manager and user can choose their username when initialization. Certain rules applied to username creation such as no duplicated names, no symbols etc. to prevent miscommunication.

4) **New user request to connect**
   When a user is trying to approach the manager throughout the above two channels, these two threads create two new threads (one for message & one for general information) for connections based on the socket setting. These two kinds of threads (chatConnectionThread and generalConnectionThread) along with the user's name are saved in three ArrayLists.

5) **User to manager communication**
   Whenever a user tries to communicate with the manager, the user writeUTF message in a certain format, and passes it the dataOutputStream to be delivered to the other side of the stream. The manager receives the message, checks the message format in event-based action, then acts accordingly.

6) **Manager to one user communication**
   When the manager needs to send a message to one specific user, the manager identifies the user's communication channel by its username and sends the message through the dataStream. When a user receives the message from the user, the user checks the message format in event-based action, and then acts accordingly.

7) **Manager to multiple user communication**
   If the message needs to be synchronized to other users, the manager then sends this message to each of the saved connections (stored in the ArrayLists).

Conclusion

This assignment helped us better understand distributed systems. In the back-end architecture, we tackled concurrency by using several threads, picked TCP/IP-based Sockets for client connection, and created communication protocols (including message formats, etc.). We utilized Java Swing in the front-end design to interface with the back-end. We established separate permissions systems for the Manager and User. To increase user interaction, we built chat boxes and user lists. From this assignment, we have gained practical experience in building a distributed shared whiteboard.

Appendix A

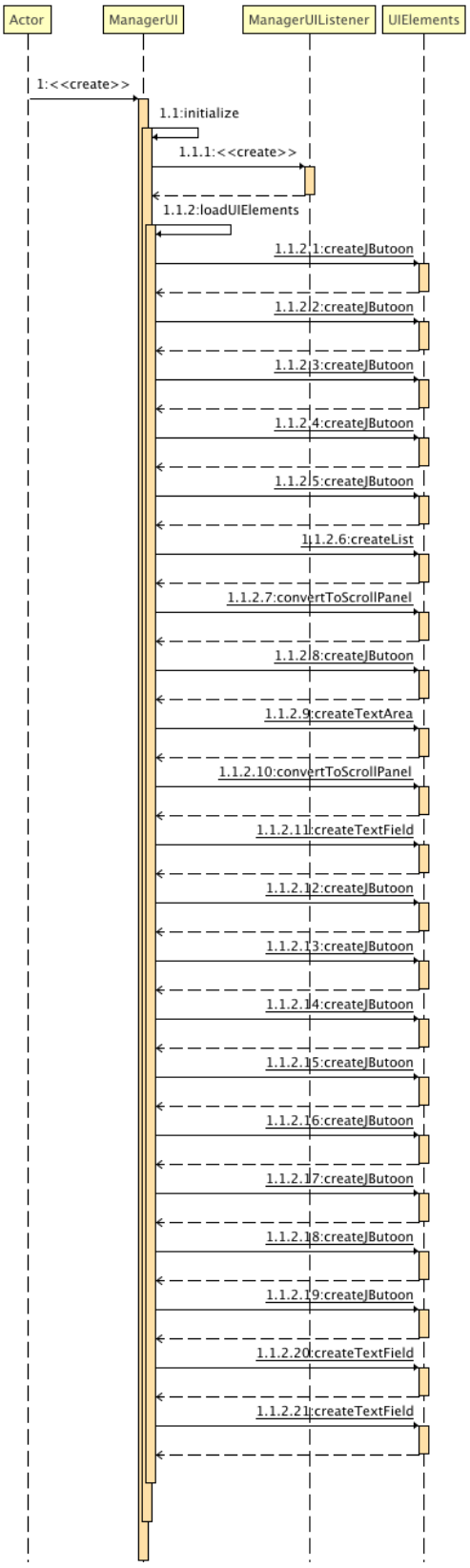

Diagram 4: Sequence Diagram - ManagerUI_new

Appendix B

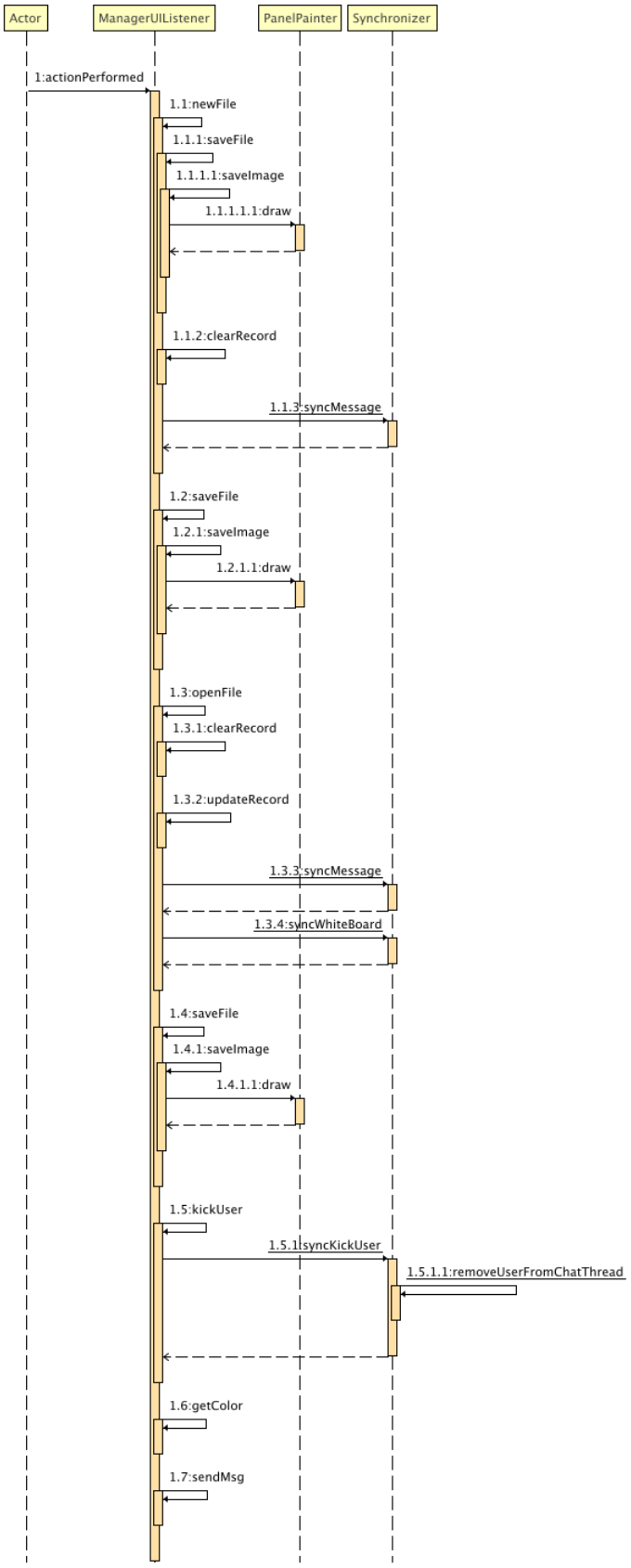

Diagram 5: Sequence Diagram - ManagerUIListener_actionPerformed

Appendix C

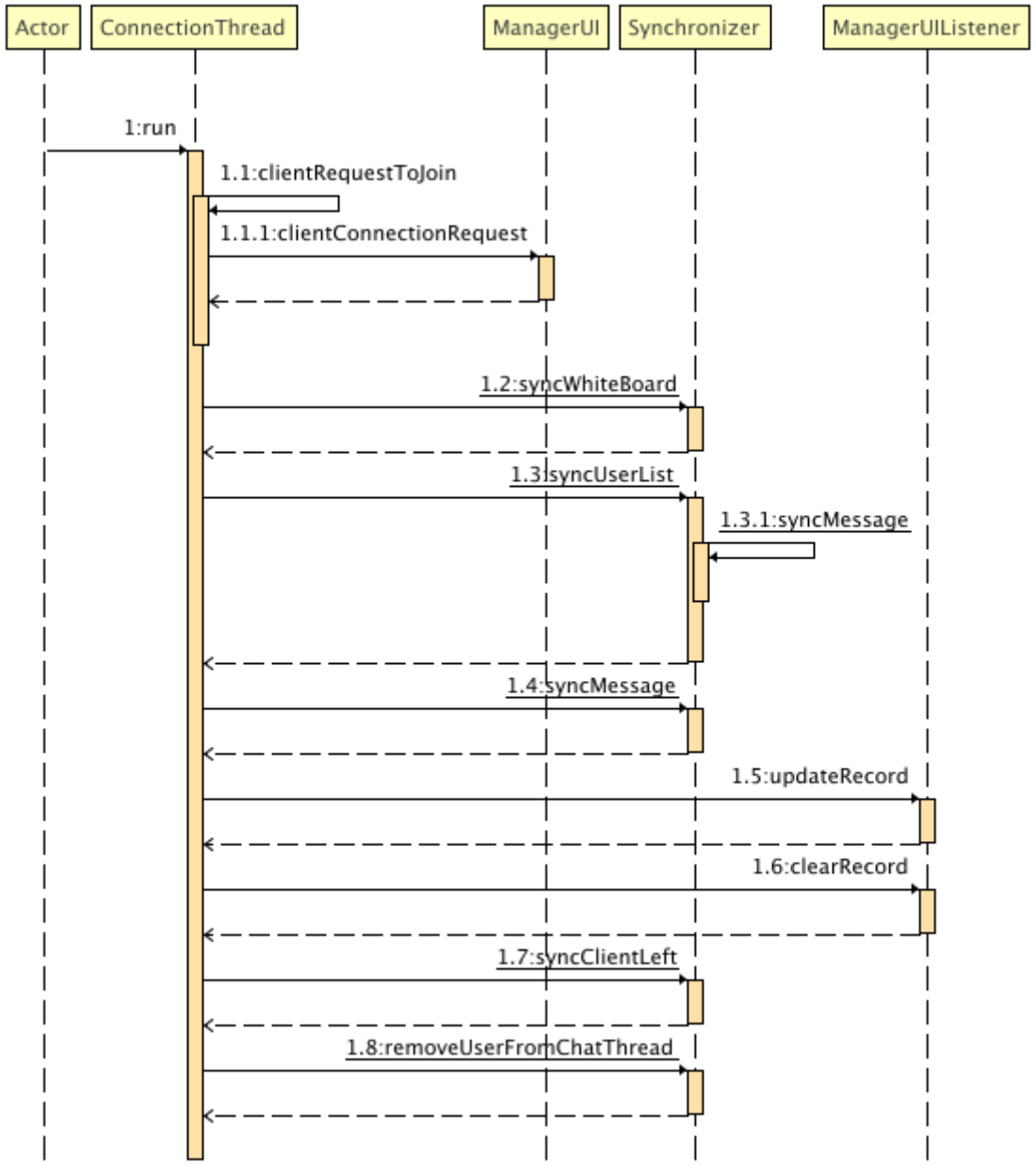

Diagram 6: Sequence Diagram - ConnectionThread_run

Appendix D

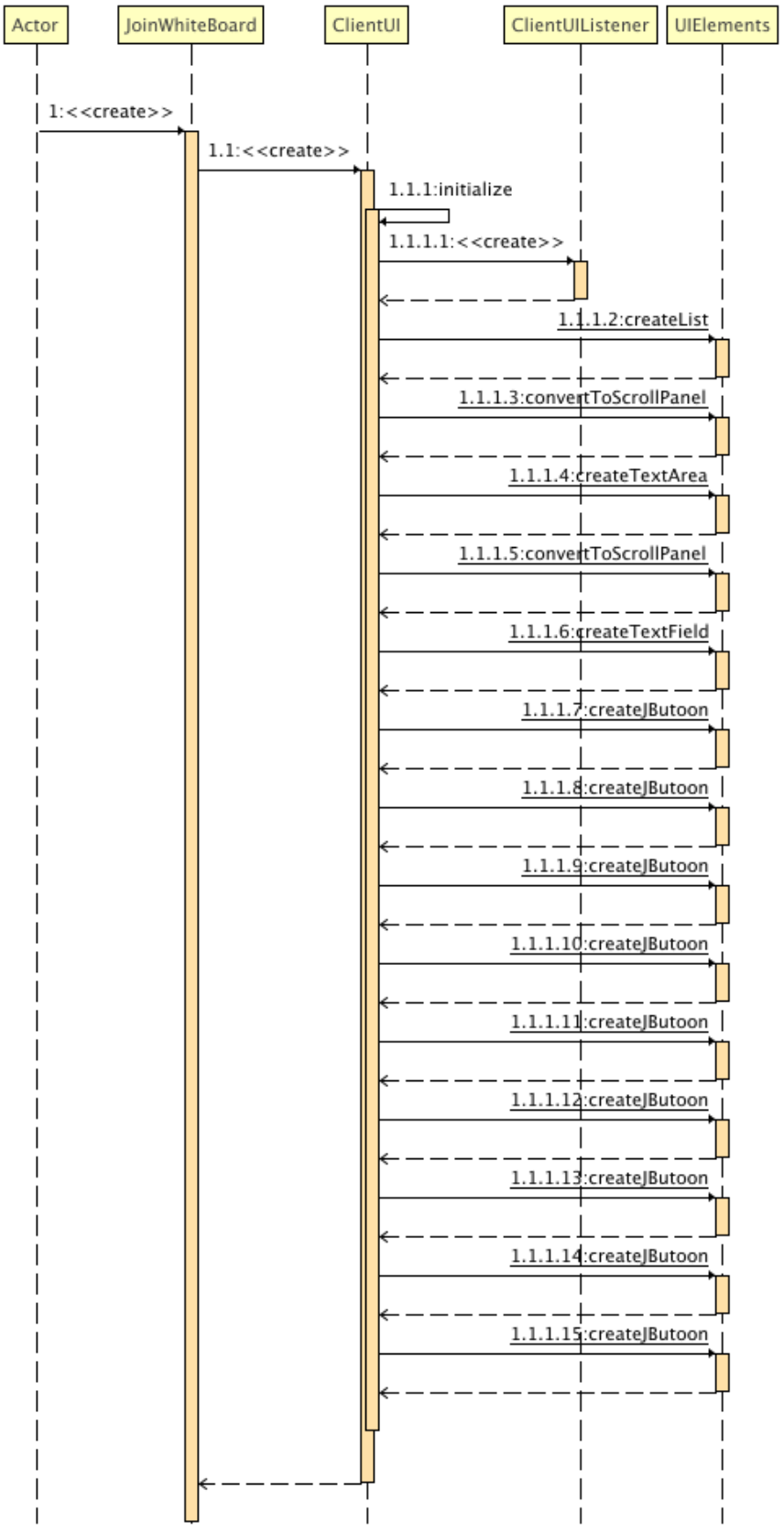

Diagram 7: Sequence Diagram - JoinWhiteBoard_new