Design and Implementation Report for Shared Whiteboard Application

COMP90015: Distributed Systems

Timothy Holland
Xiaojiang Zheng

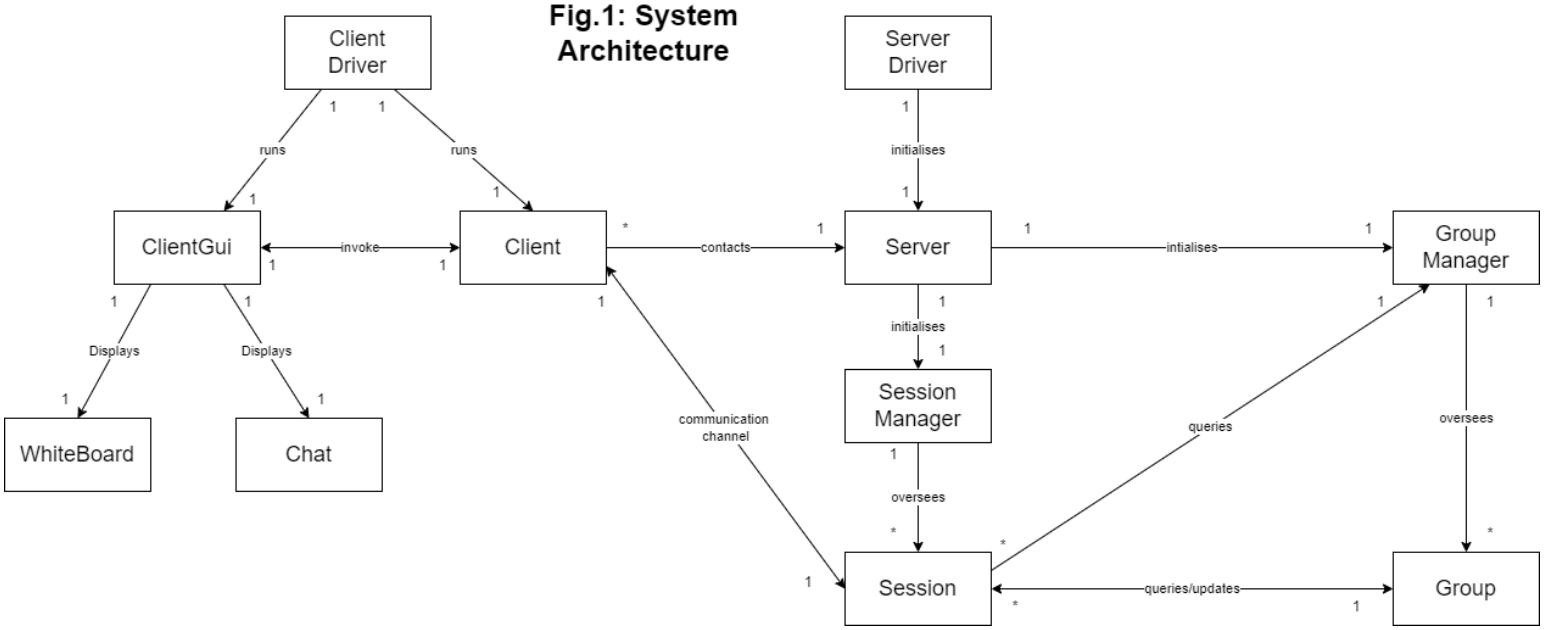
1. Introduction

To describe the execution of our distributed shared whiteboard, we will proceed from higher levels of abstraction to the particularities of implementation over four sections. To begin, we define and justify our centralised system architecture. Whilst sufficient for the application of a small number of shared whiteboards, we acknowledge it lacks scalability and propose changes for the future. Second, the communication paradigm and message formats are defined. Our system has both a synchronous and asynchronous protocol. The first handles all *validation* messages, whereby access must be checked on the server's end, thus requiring both a request and reply. The second handles the rest, all *updates* or *notifications* which can merely be propagated throughout the system. Third, we provide visualisations of the details of our main modules through UML diagrams. And lastly, we explain the implementation of the above architecture, justifying the APIs and libraries used, and the particularities of the construction of our modules.

2. System Architecture

The central aim of this project requires "developing *a whiteboard* that can be *shared between multiple users over a network*." To achieve this goal, we developed our application using a centralized system architecture, with an independent Server securing and validating the connection between multiple Clients. Of course, multiple Servers may be active, but they remain independent of one another, cannot communicate, and thus constitute different systems. Our overall architecture can be visualised as follows:

Fig.1: System Architecture

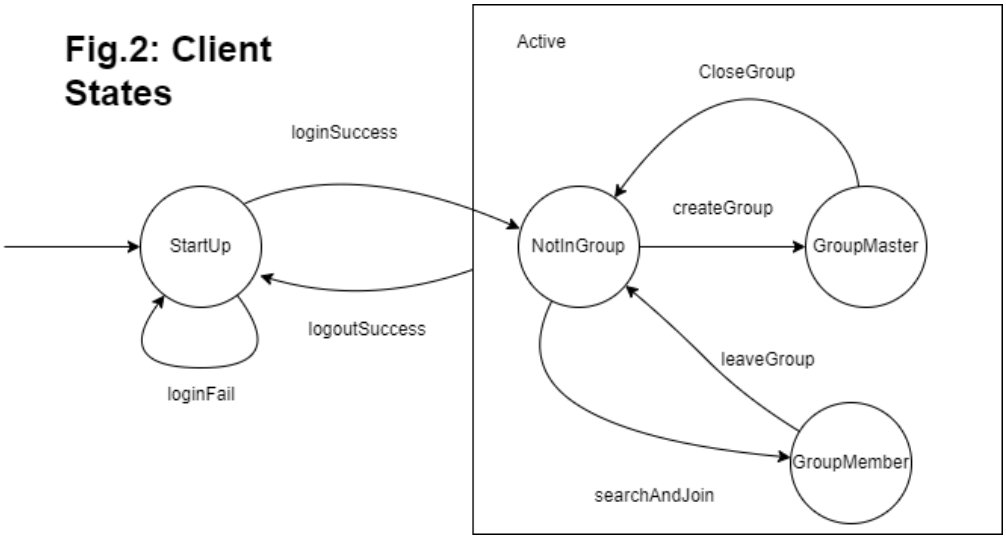


The best way to understand this architecture is by stepping through how a shared whiteboard is created/joined in our application. First, the Client and Server drivers respectively initiate a Client and Server. Upon creation, a Server awaits connections on a single socket, whilst a Client is prompted for login details via the GUI. For a connection to begin, a Client must input a unique name and the Server's secret. The GUI invokes a thread within the Client which marshalls and sends the login request, the Client then awaits the reply. After receiving this login request, the Server invokes the Session Manager to spawn a Session thread, which handles all further communication with the client, thus constituting a thread-per-connection paradigm for our protocol. After authenticating the request, the Server replies to the Client with an outcome, if successful, the communication channel is established. From here, the Client can either create or search/join a group. In either case, the Client marshals this request (again, through the GUI which invokes a thread in the Client) and sends it to its respective Session. Upon validation (depending on correct creation/search/join semantics) the Session sends the Client a reply with an outcome. If creation, the Session queries the Group Manager to ensure such a group with the defined name doesn't already exist. If unique, the Group is created with the invoking Client established as having "Manager" status. If, on the other hand, the Client joins a preexisting group, the Session instead queries the Group Manager to find such a group. This begins another protocol, whereby the Session, through the Group, sends a join request to the group's

manager.[1] If successful, the Session then invokes the Group to update all other Group Members (Clients) of the changes.

This example illustrates the fundamentals of our system architecture. In essence, the Server forms a centralised repository of all information regarding shared whiteboards, and mediates the communication of different Clients via different sessions which correspond to group members within groups. The Client merely has to marshall requests, and the Server will validate their correctness, or pass them along if they're application updates. Moving beyond the specification, our system allows for multiple groups to exist simultaneously. The Group Manager and Session objects ensure that each member in the system is unique and that no Client can be in two groups at once. As an insurance, aspects of the ClientGUI are disabled depending on the state of the Client (see figure 2).

Fig.2: Client States



For example, once a Client moves from NotInGroup to the GroupMember state (having a successful outcome from JoinGroupRequest), the buttons to join another group are no longer available. Instead, the Client must first leave the group before joining another. These measures likewise secure the integrity of the actions unique to the group's manager. Whilst our system can handle multiple groups, this is unlikely to be scalable using a single server. To expand our application, we would need to introduce intra-server communication, and create proxies to handle the load balancing of sessions and groups. Sessions that join a group would need to be handed over to the Server which stores that group's state.

[1] Note, this refers to the specific client who is a manager of the group. Not to be confused with the Server's Group Manager which manages all groups.

3. Communication Protocols, Message Formats and Interaction Diagrams

Our system utilises two different communication protocols, each corresponding to a different aspect of the system. First, as already mentioned, we implement a Request-Reply protocol, but this is unique to validation tasks. In other words, who can get access to specific resources, which in our instance, are Shared Whiteboards. For each of the following we define both a request and reply form of message, which thus constitutes the protocol:

- Login: establish session with Server
- Logout: end session with Server
- CreateGroup: initialise/manage shared whiteboard
- JoinGroup: participate in shared whiteboard
- LeaveGroup: end participation in shared whiteboard
- SearchGroup: retrieve possible whiteboards to join
- CloseGroup: manager invokes to close shared whiteboard
- SyncPull: get updated state of whiteboard (in case of failure)

In each instance, information is needed for the Client to proceed, which usually pertains to whether they have been validated in the system and/or group. The formatting of these messages are straightforward. Within each message is one of a series of reply specific strings which is propagated to the GUI to give information to the user about the outcome of their request. For example, for LoginRequest, if the server secret provided within the message is incorrect, the Server will send back a string informing such within the LoginReply message, along with a false boolean value. Other than these two fields, any request specific information is provided, like group members when joining a group, or group entries when searching for a group.

Second, we also implement an asynchronous update protocol (just request), for all messages that don't require validation. The rationale behind this is to maximise the efficiency of Client interactions. Thus, once the Client has validated their entry into a given Shared Whiteboard, we remove the request protocol for their interactions. For each of the following, we just define a request, which the Server then hands over to the Group which propagates the message:

- CanvasUpdate: member invokes when makes change to shared whiteboard

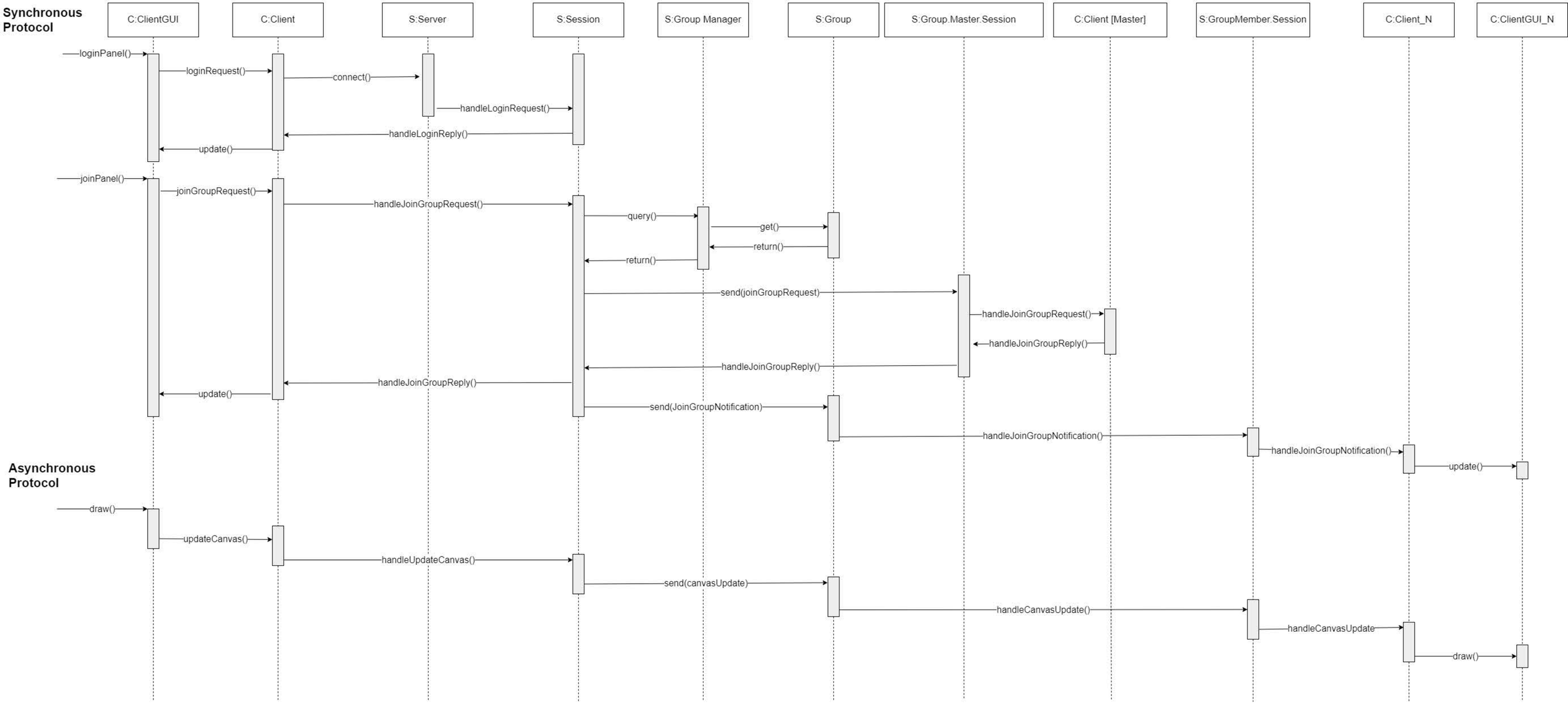
- GroupMessage: member invokes when sends message in group chat
- KickMember: manager invokes to remove member from group
- SyncPush: manager invokes when wiped or opened new canvas (to propagate to group)
- JoinNotification: session invokes when its user joins group
- LeaveNotification: session invokes when its user leaves group
- CloseNotification: session invokes when its user (also the manager) closes group

In each of these instances, the server isn't required to validate the message, and thus can hand it over to the group controller which propagates it to all other members. To avoid faults, on the client side, the GUI is encoded with different states (StartUp, NotInGroup, GroupMaster, GroupMember) which prevents a Client from sending messages or drawing on the board until they're in a group (figure 2). Moreover, network failure threatens to upend this asynchronous protocol, and thus we've included the above SyncPull synchronous messaging for a client to ensure they possess a consistent view. Formatting of these messages are specific to the update. For sharing new shapes, we simply wrap the shape in a CanvasUpdate message and propagate it through the system (likewise for chat with GroupMessage). Notably, there is no outcome or string attached, because no validation is required.

In Figure 3, these protocols can be visualised. Three events are expressed, in order to capture the complexity of our messaging system. First, the synchronous protocol follows Login then Join Group request/replies. We've chosen to model Join over Create as it provides an illustration of how messages are propagated to group asynchronously after the initial validation phase with Server's session. Also, it is the most complex request-reply message as it includes another request-reply protocol within, from the Server to the group's manager and back again. In essence, all other synchronous messages follow the schema of request, then reply/notify.

Second, the asynchronous protocol follows the propagation of drawing a shape on the shared canvas. No validation takes place, messages are merely passed on to the next object, propagating from one Client to all the others via Server Sessions. Moreover, the notification extension of certain request-reply messages can be visualised from the session onwards. The schema is simply defined by updates propagating throughout the system.

Fig.3: Interaction Diagrams



4. Design Diagrams

The execution details of the above architecture and protocols are represented in the following four UML diagrams: Figure 4 represents the Shape class; Figure 5 represents the Message class; Figure 5 represents the Client side (interactions between Client and ClientGui); Figure 6 represents the Server side (interactions between Server, Session, Group, GroupManger and SessionManager). We have divided our representation in order to improve readability. Both Client and Server sides utilise Message and Shape, thus too many dependencies arise when a system wide visualisation is generated. The mechanics of the entire system can be understood by imagining a dependency between the Client and Session in the latter two diagrams.

Fig.4: Shape UML

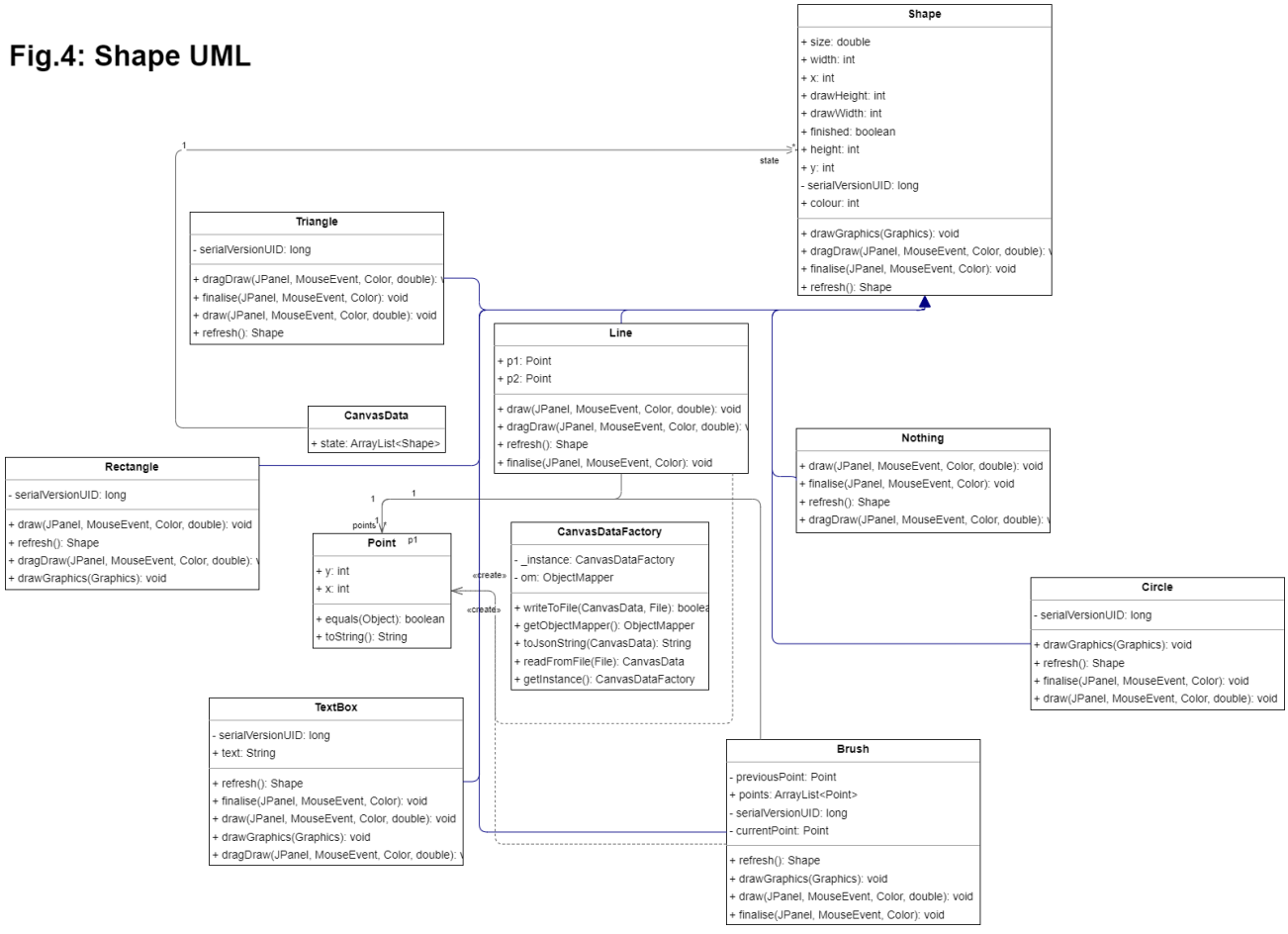


Fig.5: Message UML

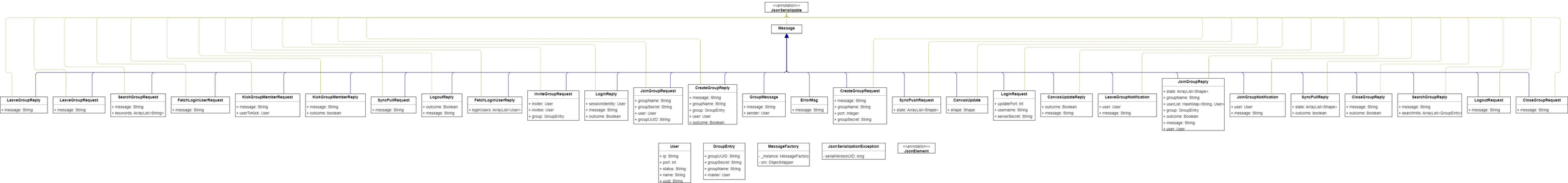


Fig 6: Client UML

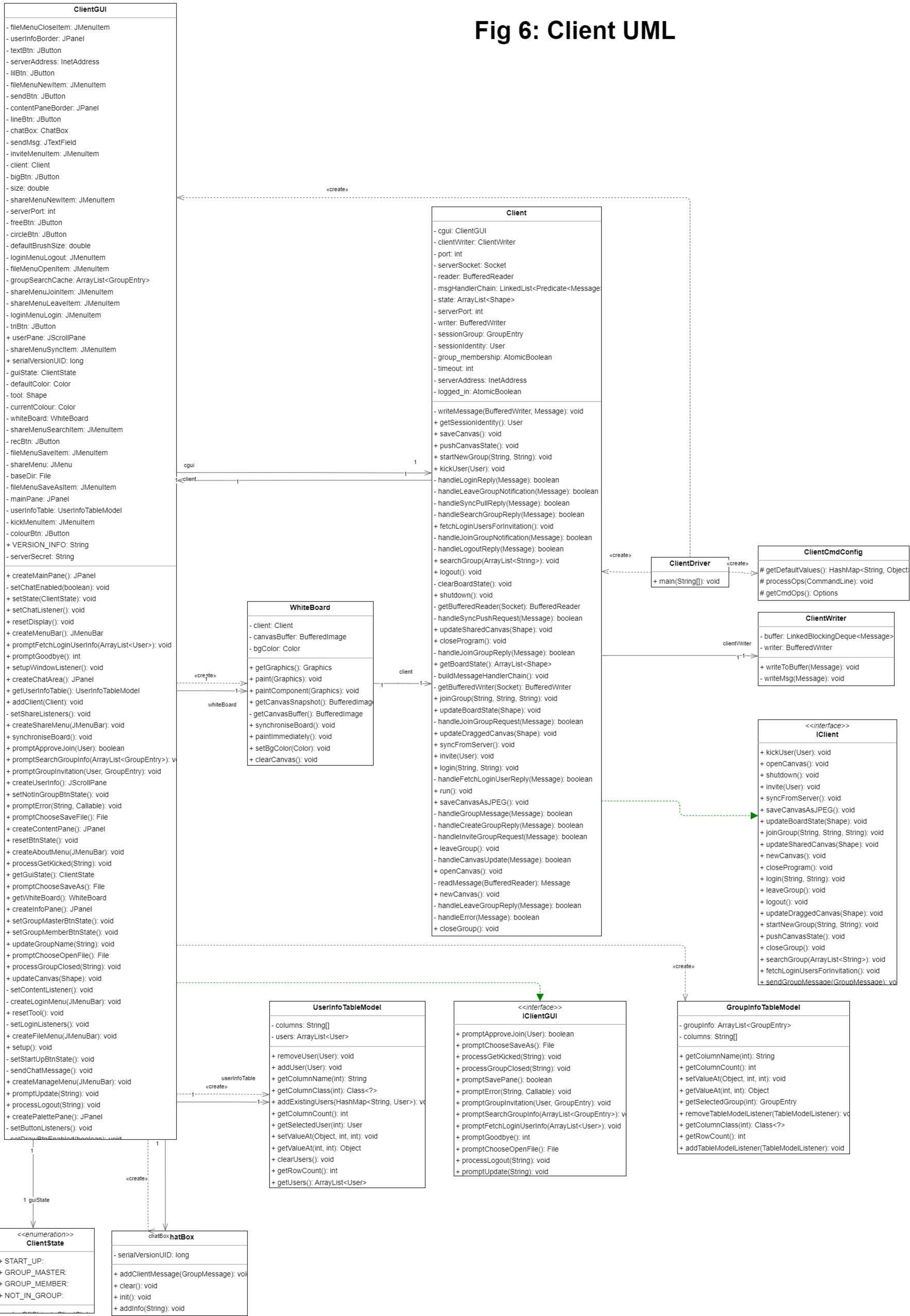
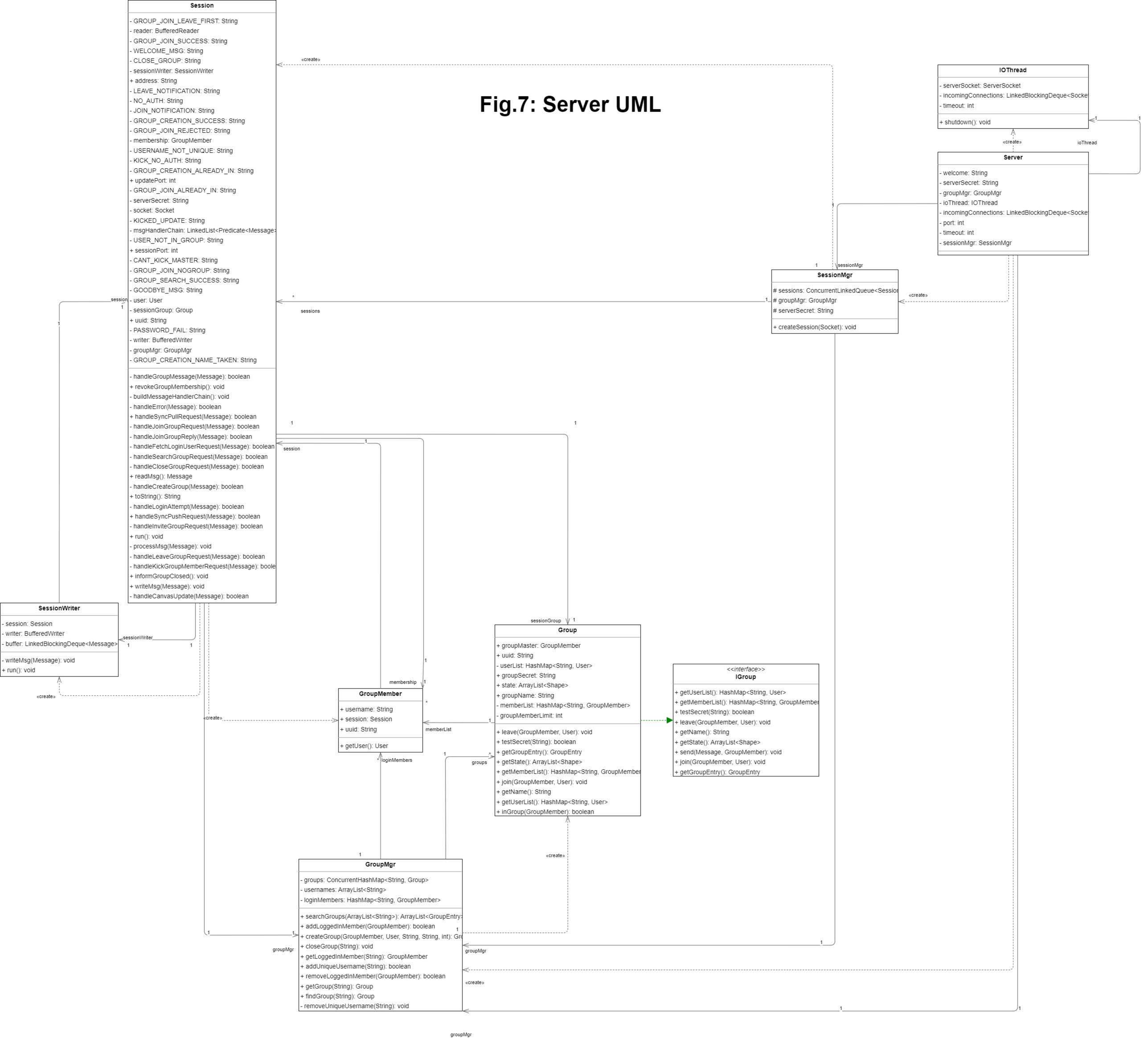


Fig.7: Server UML



5. Implementation Details

For the implementation of our design we kept things simple by building our application around socket programming and utilizing pre-existing libraries where possible. To discuss this further, this section will walk through our implementation from higher abstraction discussed so far to the finer-grained: the programming tools; and the main modules utilised and how they relate. The latter will cover the use of sockets and how we handled parallelisation.

In terms of programming tools, our application has been written with Java Developer Kit 11 (JDK11) assisted by the help of three APIs: Java Swing; JacksonXML; and Maven. We chose this environment because of our mutual experience, and the use of these specific APIs allowed us to focus on maintaining the integrity of our system instead of becoming bogged down in front-end or back-end programming. As stated, the central module employed in JDK11 to deliver our system is the TCP Socket class. As an alternative, RMI was a potential option, especially with the invocations of our asynchronous protocol for all updates. However, we figured it best to rely on our pre-existing knowledge with standard socket programming. Moreover, due to specific Client-Server design which mediates all interactions between users, the separation of roles benefited from the lack of access transparency, so each actor could handle their invocations independently. At the front-end, we employed Java Swing to display and facilitate user interactions with the whiteboard. This provided a simple means of geometrically generating and displaying all shapes through event listeners. Notably, we developed our own Shape class in place of Java's. This allowed us to have greater control over how Shape was defined (i.e. as also a TextBox, FreeHand, or even Nothing), and was easier to marshal. To handle messaging, all marshaling was implemented using JacksonXML. Again, this benefited from the experience of our first project, and the use of ObjectMapper made it simple to transform all our messages into a Json String. Lastly, at the back-end, we employed Maven to configure the project build, specifically to generate Client and Server drivers for our application to run.

The main modules we implemented are as follows. ClientDriver initiates the client side of the program, booting both a ClientGUI and Client which are mutually constituent, with the former gathering input from the user, and the latter distributing input to the system and gathering messages for the former to display. Specifically, the Client module consists of a base class, which handles all communication, both requests and replies. Client utilises ClientWriter to dispatch these messages, and otherwise awaits incoming messages on a socket read call. In each case a new thread is created in order to handle either request, reply, or update. The

Server side is more complicated. ServerDriver initiates a Server class which in turns initiates an IOThread. Group Manager, and Session Manager. The IOThread waits on incoming socket connections, and then passes them back up to the Server. The Server then initializes a Session class to handle all communication on that socket, passing in the Group Manager for queries on group validation. This connection will have come from a client login request. If the login succeeds, then the Session remains open, otherwise the socket is closed.

At its core, our system parrallises on a thread-per-connection basis. However, it is more realistic to say it is *threads*-per-connection. Whilst two central threads, Client and Session await messages from one another, all new communication, whether a request, a reply, or an update, both incoming and outgoing, are handled by these main threads parallelising into further threads. The advantage of this approach is it expedites all communication, ensuring both sides of the channel are waiting on incoming messages on a socket, thus propagating updates as fast as possible through the system. Thus, update speeds are high in our implementation, an integral feature for an interactive and live environment like a shared whiteboard.

6. Conclusion

In reporting on the execution of our distributed shared whiteboard we have moved from higher levels of abstraction to the particularities of implementation over four sections. To begin, we described our centralised Server-Client system architecture and justified it on the basis of requiring a single whiteboard only. Whilst our system can handle multiple whiteboards, we acknowledge it lacks scalability. To expand this application, we propose introducing server-to-server communication and load-balancing proxies. Second, we defined our communication paradigm as consisting of both synchronous and asynchronous protocols. The first handles all validation messages, whereby access must be checked on the server's end, thus requiring both a request and reply. The latter handles the rest, all update messages like canvas drawings or chat messages which require no validation and can merely be propagated throughout the system. We also described the constitutive features of our Message type, as defined by outcomes in the former, and updates in the latter. Third, we provided visualisations of the details of our main modules through UML diagrams. Last, we explained our implementation of the above architecture, justifying the APIs used and the construction of the modules displayed in the third section.