Margara AM25 – Communication Protocol

Introduction

This document serves as an explanation of the communication protocol employed by client and server as part of the group's implementation of the digital version of the Masters of Renaissance game by Cranio Games, which was the subject of the 2021 Software Engineering Final Exam at Politecnico of Milan.

Contents

Table of Figures

Connection Phase

At first, the client attempts to connect to the server and create a socket. On the client's side, this is handled either by the *CLI* or *GUI* classes, whereas on the server side it is handled by the *ServerMain* class, which will then run a *ServerPlayerHandler* object on a separate thread for each client socket: any messages received from the client will pass through this object.

On the client's side, the *CLI* or *GUI* class will run a *ClientReader* object on a separate thread, which will take care of processing the server's messages. The player's inputs will be handled by the specific JavaFx scene in the case of the *GUI*, or by the *CLIWriter* class, itself run on a separate thread, in the case of the *CLI*.

The *CLI* or *GUI* classes will also create a *ClientView* object which will be used to store the game's information that needs to be shown to the player.

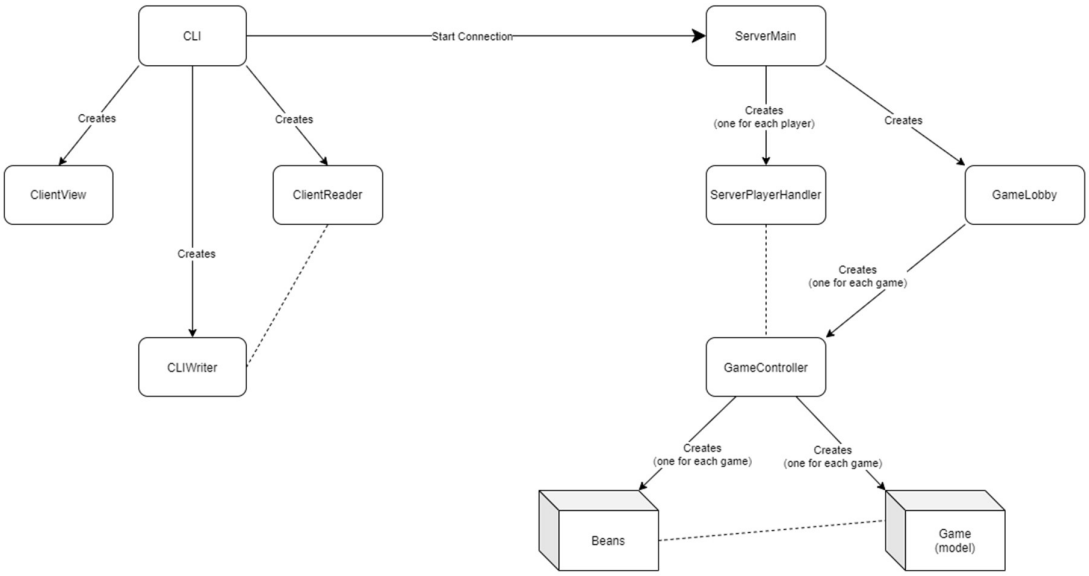

Figure 1 - Network objects (CLI)

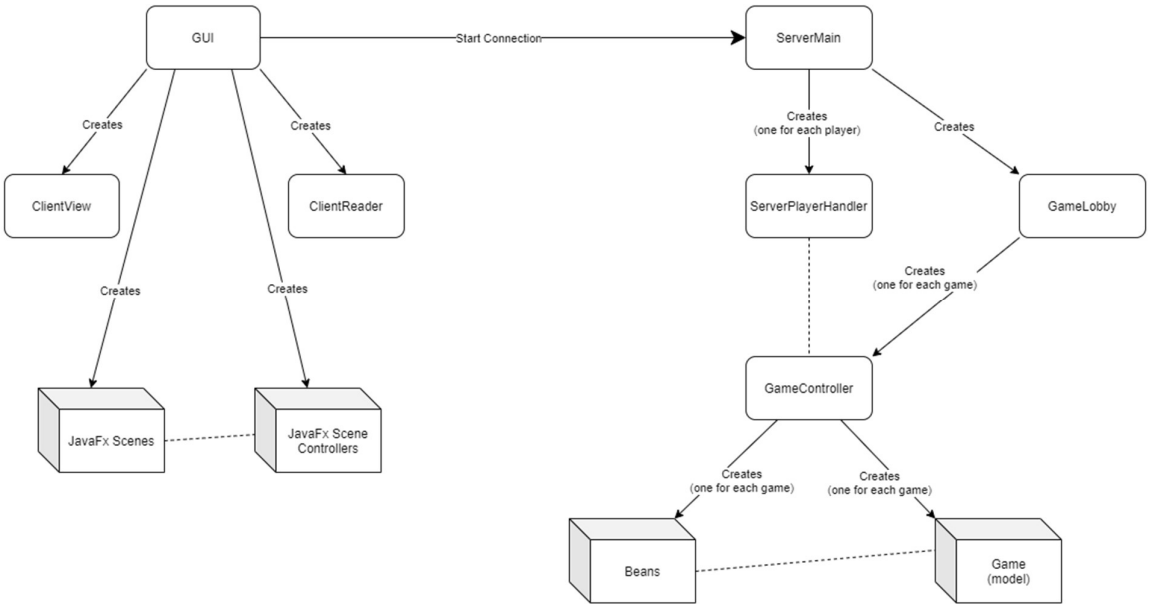

Figure 2 - Network objects (GUI)

Login Phase

Any further messages exchanged between the client and the server are sent through a *MessageWrapper* object serialized into a JSON file. This object has two attributes: a *MessageType* object, which determines what kind of information is being sent, and a *String*, which is the content of the message and may itself be simply a *String* or a different kind of object that was serialized.

During the login phase, the server sends a message of type *INFO* to the player asking for their username. The client then answers with a message of type USERNAME which contains the String that will be the player's username.

The *ServerPlayerHandler* then validates this username through the *ServerLobby*, which returns to it the *GameController* class for the game the player will take part in (either a new game, a game waiting for players or an already running game from which the player previously disconnected). At this point, the server informs the client that their username has been accepted through a message of type *CONFIRM_USERNAME*. The *SeverPlayerHandler* will then identify its player with the given username until the socket is closed.

At this point, if a new game was created, the user is prompted through an *INFO* message to send a *NUM_OF_PLAYERS* message specifying the game's size. The *ServerPlayerHandler* will then attempt to set this number of players on the *GameController*.

If the game has yet to reach its full capacity, the player (and any further players joining until it is full) is sent a message of type *WAIT_PLAYERS*, signaling the necessity for more players to arrive.

Once the proper number has been reached, all players receive a *GAME_START* message signaling the beginning of the game (the same message is sent to players reconnecting to a game they have left).

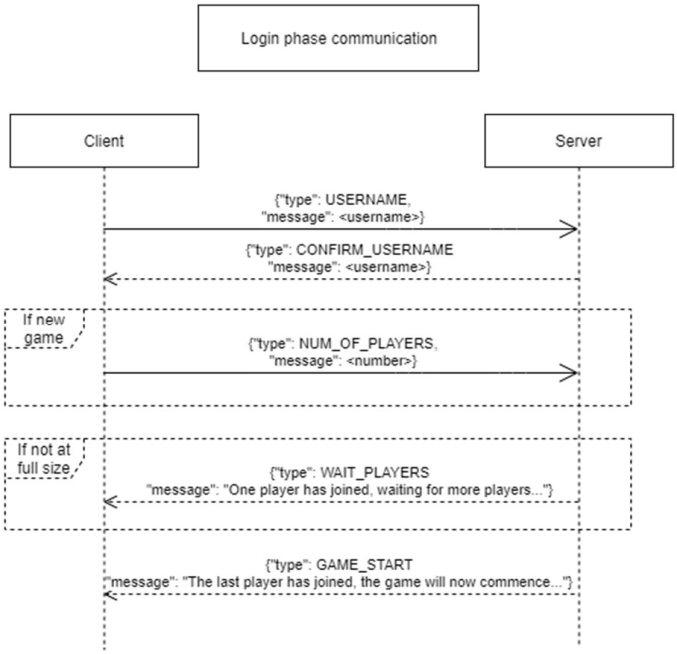

Figure 3 - Login phase main messages

If at any point something goes wrong with the process (such as the username being already taken by a connected player), the client is sent a message of type *ERROR* with a *String* detailing the problem as content.

Game Phase

At the start of the game, the server will send the client(s) several messages of type *_BEAN* (*MARKET_BEAN*, *PLAYERBOARD_BEAN*, etc.), each the serialized version of a *Bean* object on the server.

These objects are each an *Observer* to one of the model's main classes (*Market, PlayerBoard,* etc.), and serve the purpose of storing a serializable copy of those attributes that need to be sent to the client, so that it may display the game's interface to the player.

Whenever there is a change in one of those attributes in the *Observable* object, the *Bean* is updated with the new information and sends itself, serialized as a JSON, to the client through the *GameController* in a *MessageWrapper* of the corresponding type.

The bean is then be caught by the *ClientReader,* which updates the *ClientView* accordingly and triggers an update of the player's interface.

On the client's side, the player can utilize the interface to execute game actions. These are interpreted by either the *CLIWriter* class or the *JavaFx Scenes* and their *Controllers* depending on the interface being used and are then sent to the server in the form of a message of type *COMMAND*, which holds an appropriate serialized *Command* object as content.

The *Command* class has a *runCommand* method which is called by the *GameController* and executes the appropriate model-modifying action on the *Game* object passed as argument.

If this action fails for some reason, the client is once again warned through a message of type *ERROR.*

Once the game ends, the players are informed through a message of type *GAME_END.*

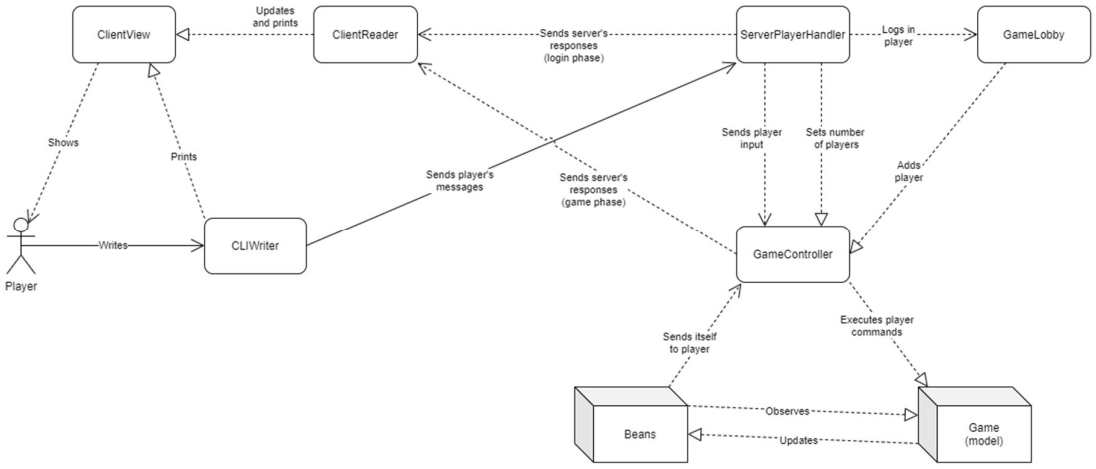

Figure 4 - Network messages (CLI)

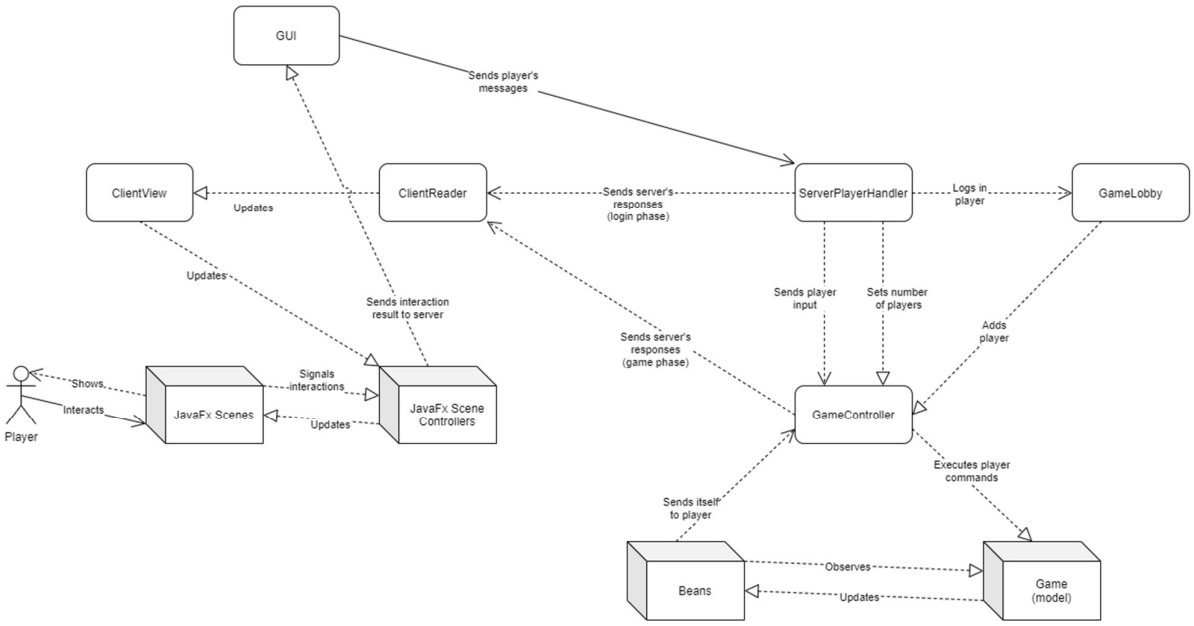

Figure 5 - Network messages (GUI)

Connection Loss

Upon creating the connection with the client, the *ServerClientHandler* runs a *TimerTask* object on a separate thread, which sends a message of type *PING* to the client every five seconds. Upon receiving this message, the *ClientReader* responds by sending another message of type *PING* to the server through either the *CLIWriter* or *GUI* objects.

Since both the client and server-side sockets are set with a timeout of ten seconds, this ensures that if the connection is lost and the *PING* messages stop being received, each part of the communication is quickly warned.

When a client leaves a game all other players are warned through a message of type *PLAYER_DISCONNECTED* bearing the username of the player as content.

Should the player join the game again by using the same username during login, all other players are informed through a similar *PLAYER_CONNECTED* message.