Communication protocol

When the client starts, it asks the user if he wants to connect to the server with socket or RMI.
After that, the user needs to login with a username.
Client and server communicate with serialized Json objects, so, technically, they send each other strings that are in this format:

{“method”: “*”, “value”: “*”}

“Method” is the first word the user needs to type, and we use it to choose which method will be called either in RMI and socket, and “value” is a String that contains the arguments that will be given to the specified method.

Our implementation consists in a lightweight client that has only the View of the Model – Passive View – Presenter (a variation of the Model – View – Controller pattern).

RMI

Server:
The class “Presenter”, is an abstract class that implements the “VirtualPresenter”.
Two classes, “RmiPresenter” and “SocketPresenter”, extend “Presenter” and implements its abstract methods, the ones that are devoted to the network.
Client:
The class “ConsoleView” implements the methods of the interface “VirtualView”, which are the methods that we expose to the server using a “callback” method that not only gives the client server the possibility to use client’s methods, but also sets a one to one connection between the ConsoleView of each client and its Presenter.
Thus, every client has a personal presenter that accesses the model which is unique (within the same match).

Socket

Server:
The class “SocketPresenter” extends “Presenter” but, as said before, also has methods that put on the output buffer the serialized JsonObject for the client.
Simmetrically the client sends a String that is actually a Json object that can be used after being deserialized.
Client:
The class SocketConnection deserializes the Json object as soon as it is processed from the input stream to make it usable.

Initialization

Login phase

- User: “login *username*”;
- Client: sends the proper json object to server;
- Server: sends a confirm message to client;
- Client: prints to user login effettuato come *username*”.

Every time a new user connects to the server, everyone connected receives a notification.
In every moment any user can ask the server to start a new game, but the request is accepted only if there are at least three people logged in, the minimum number of players necessary to play.
If there are too many people waiting and no one asks to start a new game, the game starts automatically.
Since for every query from the user, the client and the server share each other Json objects, to make the description less verbose we will omit that part and pretend the client is the user that "speaks" with the server.

New game phase

- User: "login nomeUtente";
- Server: "login effettuato come nomeUtente";

The User can here ask different things:

- User: "mostrapartite"
- Server (on ok): "nomePartita, numeroTeschi, frenesiaFinale, giocatoriConnessi"
- Server (on error): "Non è stata create nessuna partita".

- User: "creapartita nomePartita numeroTeschi, frenesiaFinale";
- Server: "partita creata con nome nomePartita";
- Server (on error): "Error message".

- User "selezionapartita nomePartita nomePersonaggio";
- Server: "sei stato connesso alla partita nomePartita con successo."
- Server (on error): "Error message".

Once the game starts, the server will set the player who created the game as the first player.

Everyone can make requests to the server at any time, but some requests will be accepted only if it is the turn of who made the request.

Game

Spawn phase

The board will show up
- Server: gives two power up cards to the client;
- User: types where he wants to spawn based on the power up cards he received (for example): "stanza rossa, quadrato 2"

This process will occur not only in the first round, but every time someone dies.

Turn phase

The user has a set of requests that will always be answered like:
- "mostra azioni disponibili";
- "mostra carte";

- “mostra info *nome personaggio**”;
- “mostra quadrato 0 stanza rossa”; // shows the cards in that spawn square
- “”

The board will always be visible (also on the cli) and will be updated every time it changes.
It must be chosen the type of action the user wants to perform depending on what he can do in that moment of the game.

Collect:
- User: “raccogli - munizioni/*nome arma*”;
- (Or) Client “corri - *quadrato di destinazione*, raccogli - munizioni/*nome arma”;
- Server (on ok): performs the action and notifies everyone of what happened;
- Server (on error): notifies only the user who made the invalid request “azione non valida, inserisci un’azione valida”.

Run:
- User: “corri - *quadrato di destinazione*”;
- Server (on ok): performs the action and notifies everyone of what happened;
- Server (on error): notifies only the user who made the invalid request “azione non valida, inserisci un’azione valida”.

Shoot:
- User: “spara”;
- User: “usacarta nomeArma, tipoEffetto, bersaglio/listaDiBersagli”;
- Server (on ok): performs the action and notifies everyone of what happened;
- Server (on error): notifies only the user who made the invalid request “azione non valida, inserisci un’azione valida”.

If the user wants to add a power up can do so by adding this:

- User: “usacarta nomePowerUp, bersaglio(opzionale)”;
- Server (on ok): performs the action and notifies everyone of what happened;
- Server (on error): notifies only the user who made the invalid request “azione non valida, inserisci un’azione valida”.
-

Reload:
- User: “ricarica - *nome arma, power up (opzionale)*”
- Server (on ok): performs the action and notifies everyone of what happened;
- Server (on error): “non hai abbastanza risorse per ricaricare, riprova”.