# Communication Protocol

*Introduction*

The protocol used in the program for the communication between client(s) and server is based on two streaming channels taken from each socket by the methods *socket.getInputStream()* and *socket.getOutputStream()*: the input stream of each one of the server's sockets is connected (and only communicates with) the corresponding client's output stream, and vice versa.
So, the server extends an Observable class which is responsible for the update of each listening Observer (a socket on the server linked to another socket on client's side) whenever anything happens in the model, for example each time the model updates the board or asks a client to make a choice.
When the server has to send a message, it behaves in two different ways: it sends the message to all the connected clients or to a specific client.
Instead, when the client receives the message, it can decide to answer back (that is what happens most of the time) or just update the view's state.

*Messages*

A message can fulfil a certain role generally based on the phase in which the server is found at that moment, that is the "lobby" phase or the "game" phase (see paragraphs below). In these two phases the message will be, respectively, a **LobbyMessage**, containing an id for the printing on the client side of a specific textual message (based on the *NetworkMessage* enum) and/or some relevant data, and a **GameMessage**, that changes its content interpretation based on which side is sending it.
In fact, a GameMessage has got an *id*, a *flag* and some *data*:

- id: it can be the value of the *MethodHeading* enum which helps the search of the method to call on the view (server side) or just the id of the sender (client side);
- flag: distinguish whether a certain message has to be sent to a single client or to all the clients;
- data: this field contains all the relevant Objects that a certain client should find useful and then calling the method described by the id field (same on server side).

Two other types of message are exchanged in order to handle a normal disconnection (**QuitMessage**) and a forced disconnection (**KickMessage**). The latter shouldn't be confused with the disconnection caused by a forced closing of the terminal both on client and server side, but it is to be intended as a disconnection imposed directly from the server (for example when a certain player loses the match but the game isn't over yet).
When a player wins a game, a special **WinMessage** is sent in order to handle correctly the disconnection on every client, that is making them disconnect in an orderly way. Instead, when a player interrupts the current game with his disconnection, the server sends the last connected client a new **InterruptionMessage** to bring it back to the lobby phase.
The last type of message is the **TextMessage**, which is basically used to exchange simple text and differs from the LobbyMessage because it's not intended to call other methods on the client's view, but just to print the text contained in the message.

N.B.: Each type of message extends an *abstract* class named **SerializableMessage**, which simply implements the Serializable interface.

*Disconnections*

All the forced disconnections, in the real sense of the word, are handled by catch clauses both on client and server side, but they are handled differently by the server based on the phase the game is in.
When the server is in "lobby" phase, the disconnected client doesn't create any side effects except for the host rotation, that is the "host" title reassignment if the client who just disconnected was the host.
In "game" phase, on the other side, the disconnection of a single client generates the disconnection of all the other connected clients and then a state transition of the server from "game" to "lobby" phase (which can be seen as a restart).

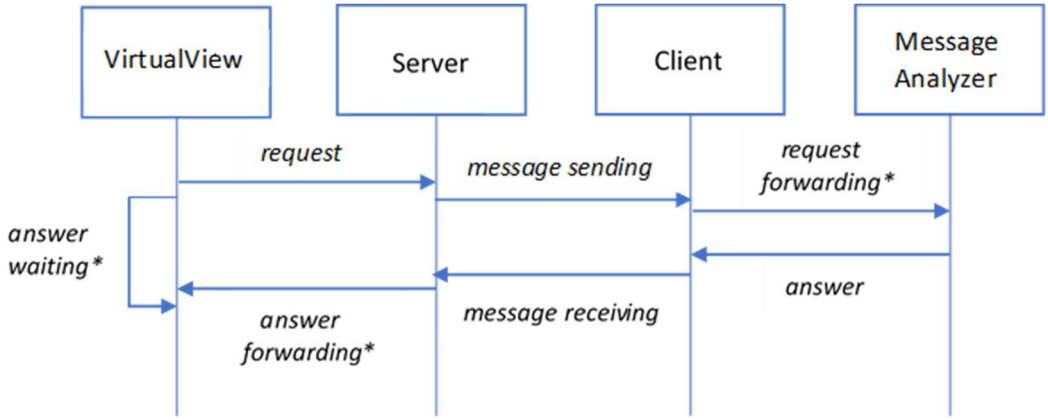
Every disconnection could happen at any moment, regardless of server and clients' state.
N.B.: the very first "host" title is assigned to the first client who connects to the server.

*Read and Write*

On both sides the writing and reading processes work asynchronously in separate threads, with the only difference that the reading process is the main one, so basically the two sides are always listening to each other.
Furthermore, both sides make use of a **MessageReceiver** object which contains a sort of buffer to store the messages exactly in the order they arrive in, so that the two sockets can instantly return to listen to the input stream.

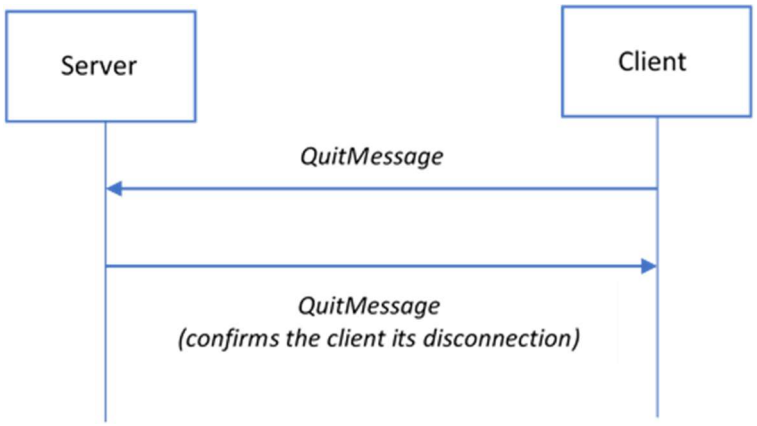
*Sequence diagram*

Example of a typical server-to-client interaction:




**operations made through the MessageReceiver class for the storing and acquisition of the message*

Example of a typical client-to-server interaction, which usually happens for a disconnection request:




*N.B.: In this example there's no forwarding to the Receivers because the client immediately closes every connection as well as the server*

In both of the examples above, the server includes the classes **ServerNetworkHandler** and **ClientConnectionSocket**, which are respectively the Observable and the Observer(s) (more ClientConnectionSocket classes are handled by an *ExecutorService* on the ServerNetworkHandler); the client just features the **ClientNetworkHandler** class.

# Lobby Phase

*Introduction*

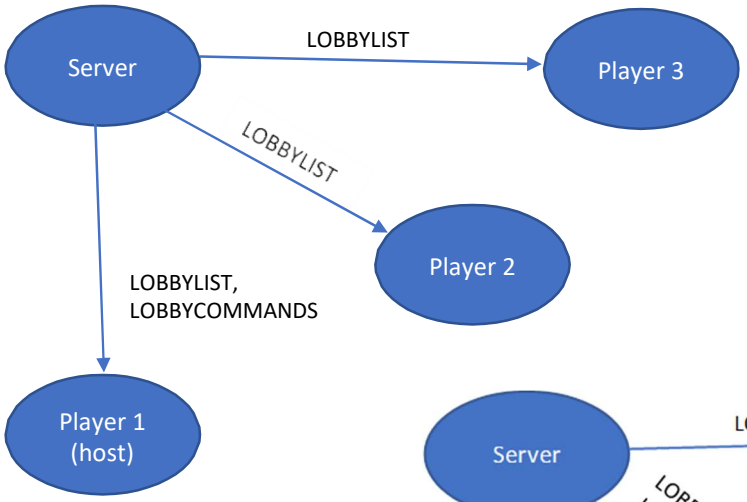
The lobby phase is characterized by an exchange of a specific type of message, that is the **LobbyMessage**. This one can be used both to display textual messages sent by the server and to let the host interact with the server to decide what to do during this phase. When the host disconnects during this phase, the player who connected to the lobby right after him will be elected as the new host and will be opportunely notified about that.

*Login*

When a player connects to the server, he is asked to choose a unique username to be identified in the game. The choice is then forwarded to the server, which can ask the player to choose another username if the previous one has already been chosen.
After that the player will have no more interactions with the server until the game starts, but only if he is not the host (see next paragraph).

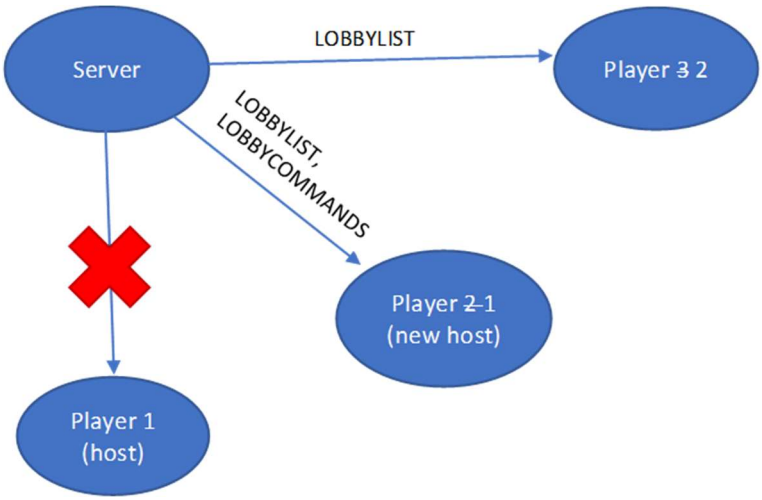
*Host-Server interaction*

The first player who connects to the lobby is marked as the "host" of the game. With such a role, a player can decide to change the size of the lobby (within certain defined limits) or to start a new game. In order to allow the host to make a decision, the server sends to him exclusively a list of available commands.
The following diagrams show what the server sends to each connected client during this phase and how the host disconnection is handled.




LOBBYLIST and LOBBYCOMMANDS are two MethodHeading's values, respectively used to print the list of players in the lobby and the list of available commands (for the host).

When someone disconnects, the id assigned to each player changes to make the host have the id "1".



# Game phase

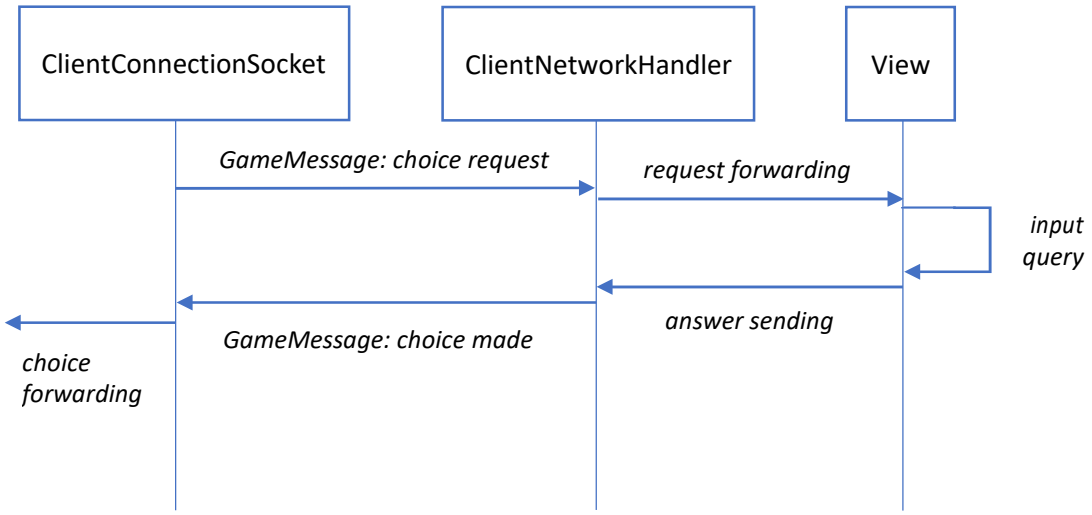
*Introduction*
In this phase the wrong inputs are handled locally by each client. Thus, if a player chooses an invalid position to move/build/place a worker to/in etc., the client can already distinguish which are the correct inputs for that specific choice and which are not.

*Setup*
Right after the lobby phase, the server starts the game setup and, from that moment until the game ends or someone disconnects, clients and server communicate through **GameMessage** objects.
There are three game modes but only two types of setup: one mode is “basic”, that is it includes only the workers’ color selection; the other ones are more or less the same, with the only difference being that the “advanced” one includes more cards than the “complete” one. In the advanced and complete modes, the server sends the player the list of cards in game (in particular they are **ImmutableCard**) in order to print them on the screen; also, each client enables a new command that shows the power of every card in game when explicitly requested by a certain user (this type of command is locally handled by the client as well as every other helping command).

*Start*
When the setup ends the game officially starts, and the server becomes responsible for the update of the board on each client. In fact, every time a move or a build is required the server sends the board to every client, then it asks for the needed information.
Basically, this phase can be divided in two different parts:

- the **worker selection**: each player chooses the wanted worker, then sends the choice to the server;
- the **move**/**build**: both move and build require the choice of a position (**ImmutablePosition**) which is then forwarded to the server.

The following diagram shows a typical interaction for a generic selection request:


ClientConnectionSocket
ClientNetworkHandler
View
GameMessage: choice request
request forwarding
input
query
answer sending
GameMessage: choice made
choice
forwarding