Sequence diagram

Group AM09

Contents

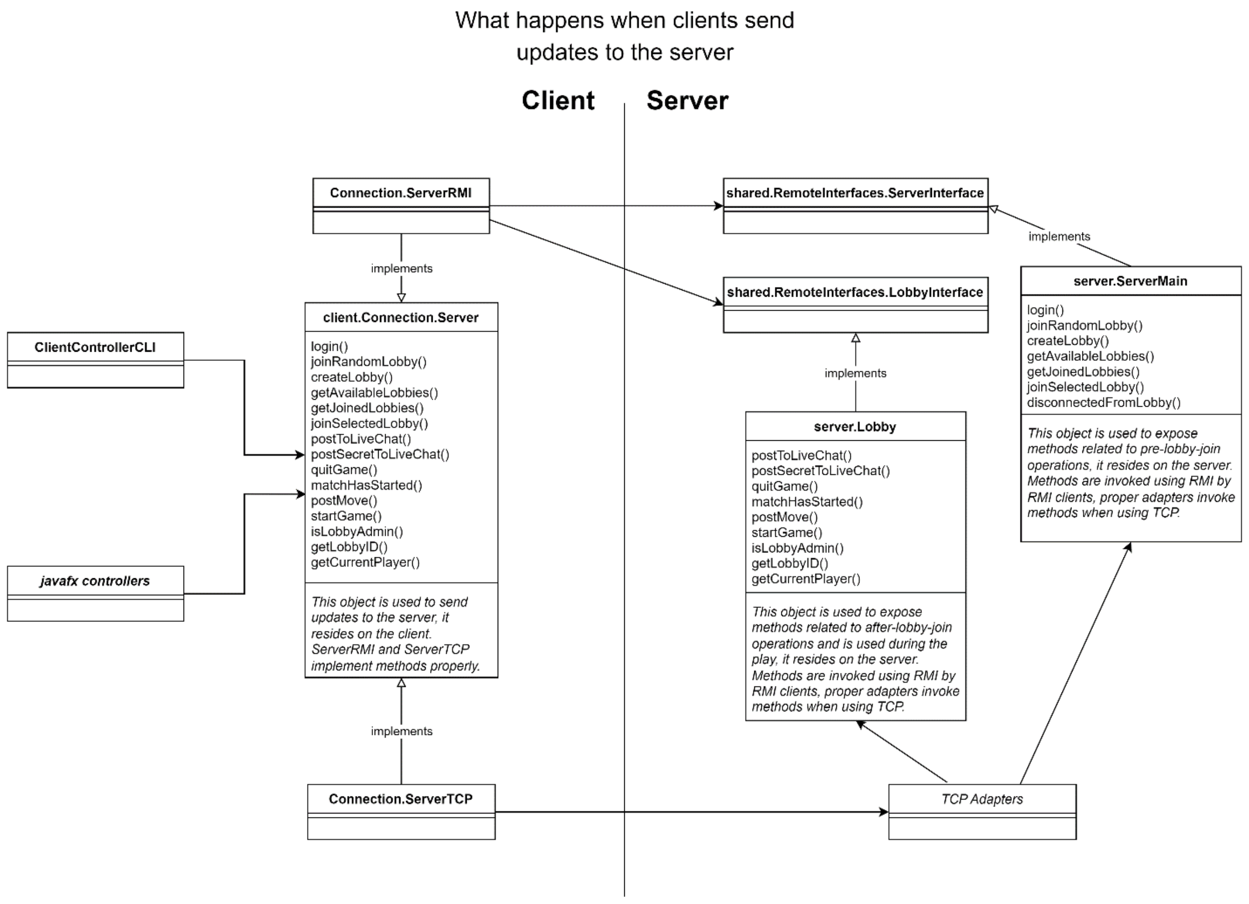
Classes responsible for connection

The project is structured in a way that allows bi-directional communication between clients and server. Clients can update the server status and the server can force updates on clients.

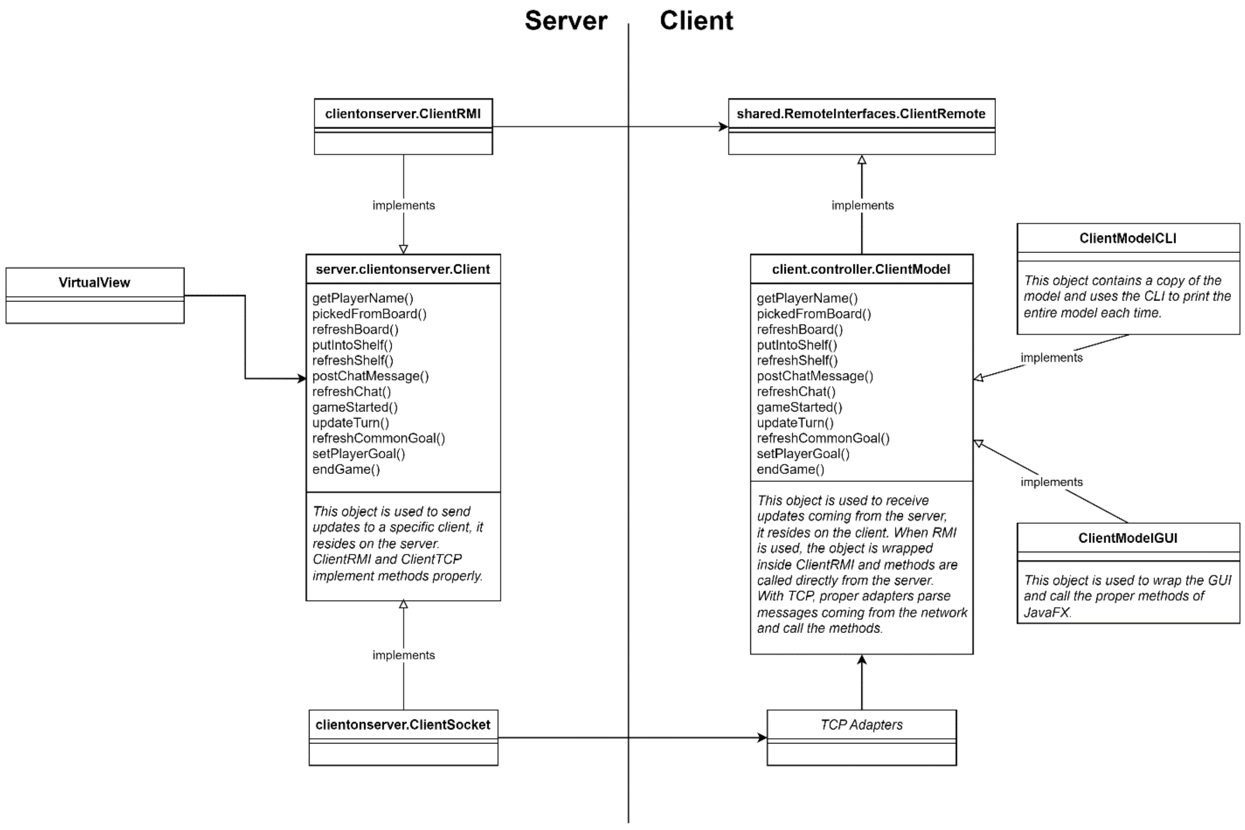
As requested, the connection can be implemented via TCP sockets or via RMI. The server supports both types of connection at the same time while a client selects a type of connection before launching the application and sticks with it until shutdown of the application.

We decided to implement an **adapter pattern** in order to achieve complete transparency of the connection type for objects that handle client-server communication.
The main classes of Client and Server do not need to know the underlying implementation, so the sequence diagrams are not different for TCP or RMI clients.

Below are reported the UML class diagrams of the classes that will be used in the following sequence diagrams.


What happens when clients send updates to the server
Client
Server
Connection.ServerRMI
shared.RemoteInterfaces.ServerInterface
implements
shared.RemoteInterfaces.LobbyInterface
server.ServerMain
login()
joinRandomLobby()
createLobby()
getAvailableLobbies()
getJoinedLobbies()
joinSelectedLobby()
disconnectedFromLobby()
This object is used to expose methods related to pre-lobby-join operations, it resides on the server. Methods are invoked using RMI by RMI clients, proper adapters invoke methods when using TCP.
implements
client.Connection.Server
login()
joinRandomLobby()
createLobby()
getAvailableLobbies()
getJoinedLobbies()
joinSelectedLobby()
postToLiveChat()
postSecretToLiveChat()
quitGame()
matchHasStarted()
postMove()
startGame()
isLobbyAdmin()
getLobbyID()
getCurrentPlayer()
This object is used to send updates to the server, it resides on the client. ServerRMI and ServerTCP implement methods properly.
ClientControllerCLI
javafx controllers
implements
server.Lobby
postToLiveChat()
postSecretToLiveChat()
quitGame()
matchHasStarted()
postMove()
startGame()
isLobbyAdmin()
getLobbyID()
getCurrentPlayer()
This object is used to expose methods related to after-lobby-join operations and is used during the play, it resides on the server. Methods are invoked using RMI by RMI clients, proper adapters invoke methods when using TCP.
implements
Connection.ServerTCP
TCP Adapters

What happens when the model needs to update clients


Server
Client
clientonserver.ClientRMI
shared.RemoteInterfaces.ClientRemote
implements
implements
VirtualView
server.clientonserver.Client
getPlayerName()
pickedFromBoard()
refreshBoard()
putIntoShelf()
refreshShelf()
postChatMessage()
refreshChat()
gameStarted()
updateTurn()
refreshCommonGoal()
setPlayerGoal()
endGame()
This object is used to send updates to a specific client, it resides on the server. ClientRMI and ClientTCP implement methods properly.
client.controller.ClientModel
getPlayerName()
pickedFromBoard()
refreshBoard()
putIntoShelf()
refreshShelf()
postChatMessage()
refreshChat()
gameStarted()
updateTurn()
refreshCommonGoal()
setPlayerGoal()
endGame()
This object is used to receive updates coming from the server, it resides on the client. When RMI is used, the object is wrapped inside ClientRMI and methods are called directly from the server. With TCP, proper adapters parse messages coming from the network and call the methods.
ClientModelCLI
This object contains a copy of the model and uses the CLI to print the entire model each time.
implements
implements
ClientModelGUI
This object is used to wrap the GUI and call the proper methods of JavaFX.
implements
clientonserver.ClientSocket
TCP Adapters

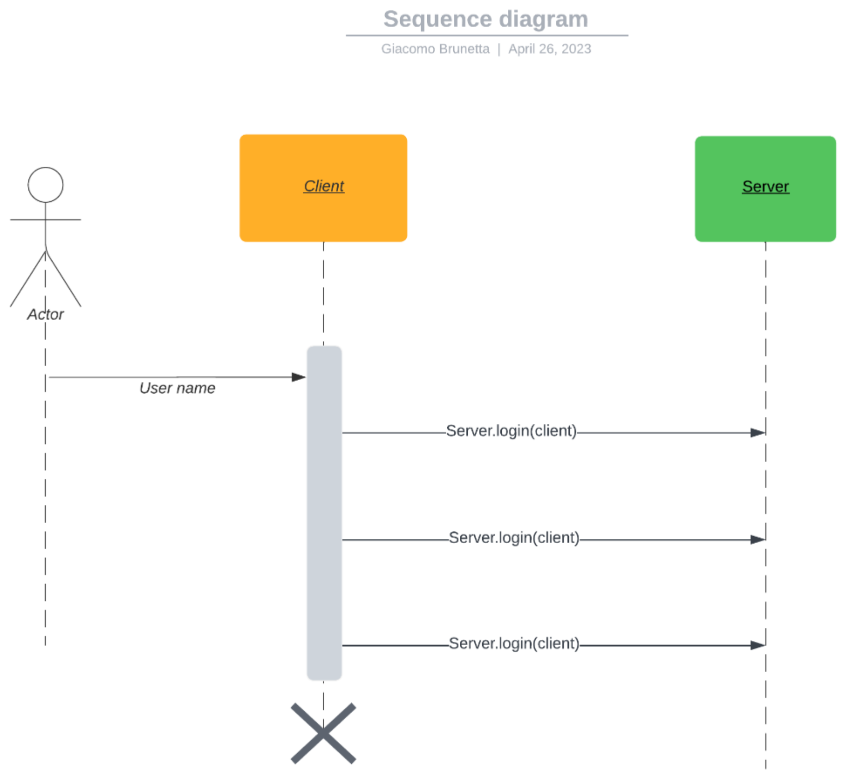
Sequence diagrams: unsuccessful client login

The first step for connecting the client to the server is to login.

The client application asks the user for a username and then creates the client connection object that will allow the server to update the client-side model and then tries to connect to server passing the client connection object as parameter.

The *tryLogin(int number, int seconds)* method is invoked with number = 3 and seconds = 2 (according to the *ConnectionSettings* util class parameters) so that the client will try to connect at maximum 3 times and will wait for 2 seconds between every try.

The sequence diagram below shows a client that tries three times to connect to server before the automatic shut down of the application that notifies the user that server is unavailable and it was impossible to login.



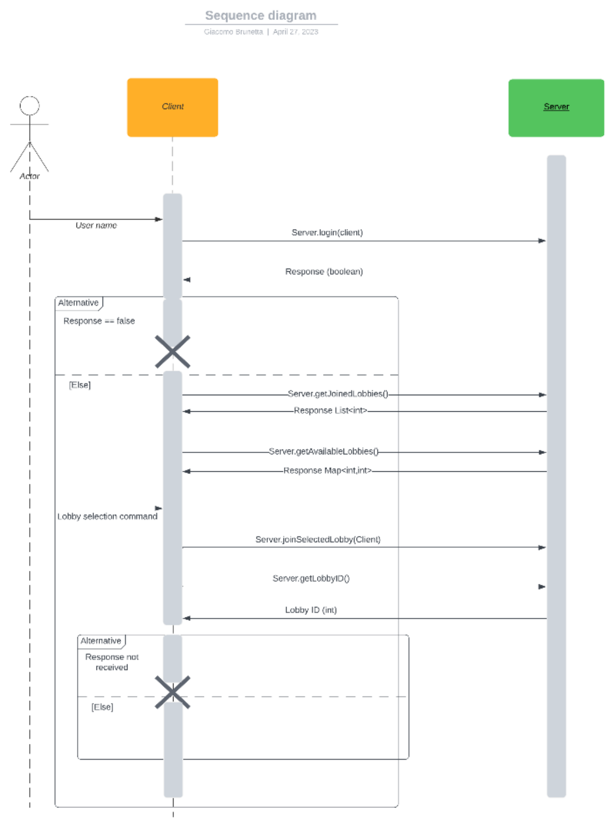
Sequence diagram: successful client login

When the client receives a response from server it will interpret the Boolean value received as response: if it is negative the client will shut down, if it is positive the client will continue ask the player what lobby he wants to join.

The process of joining lobbies follows this sequence of events:

- Ask for all the lobbies in which the player is already present as a disconnected user.
- Obtain a response from the server.
- Ask for all the available lobbies on the server.
- Obtain a response from the server.
- Receive from the view a command that specifies the action to perform that can be:
    - Join a specific lobby.
    - Join the first available lobby or create one if none is available.
    - Create a new lobby.
- Obtain the lobby id from the server.

If at any time a response is not received or throws an exception the client will consider the server unreachable and shut down notifying the user about the issue.



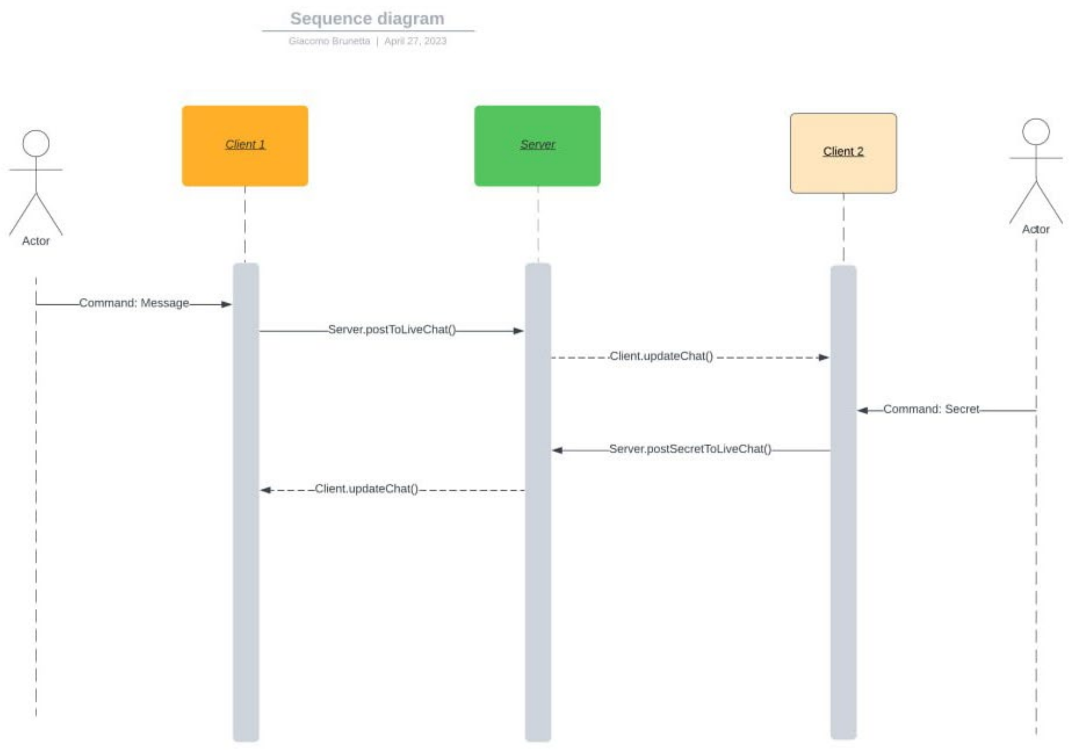
Sequence diagram: chat

Once in a lobby a client can receive status updates from the server and can update the status of the server that will push the update to all the clients.

Chat is always available, even before and after the actual match. Anyone can post messages at any time and others will receive asynchronous updates on their local copy of the chat.

Messages can be either private or public. A private message will be seen only by the receiver decided by the sender, while the public message can be seen by everyone.

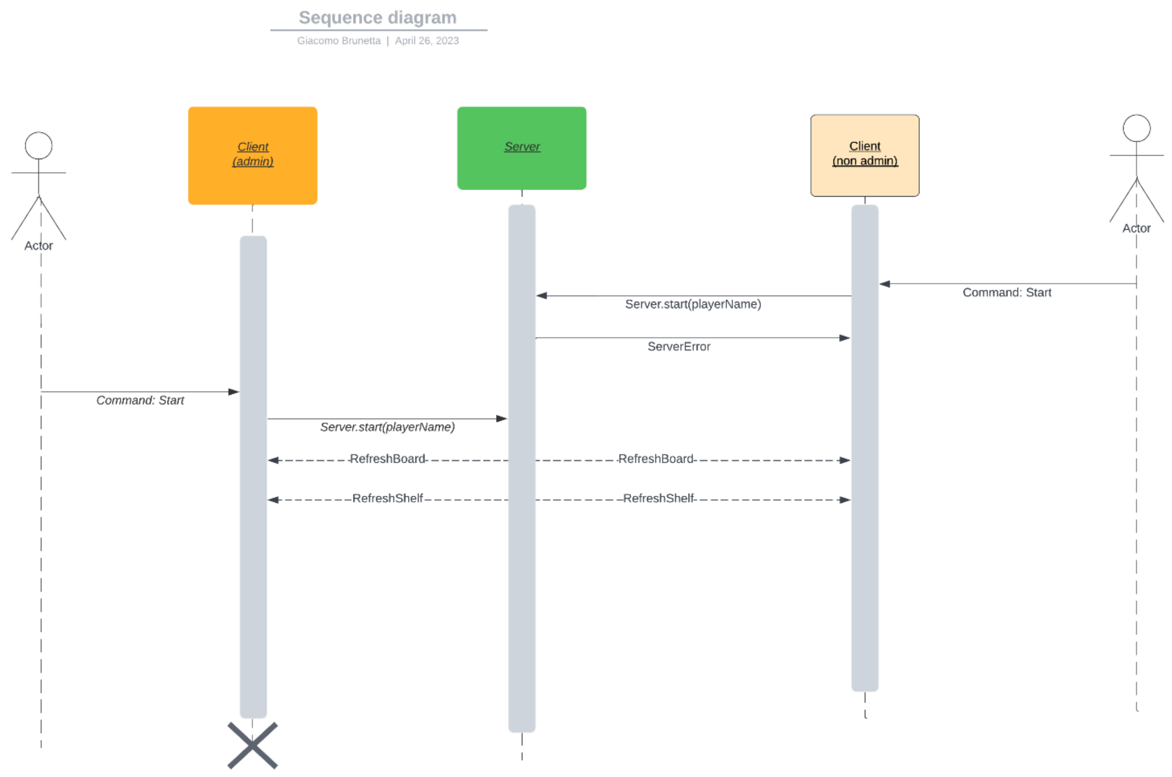
The view sends two different commands "secret" and "message" that trigger the *Server.postToLiveChat()* and *Server.postSecretToLiveChat()* methods.



Sequence diagram: start of match

A match can start in a lobby only if there is the minimum required number of players.
The *GameSettings* utils class sets the minimum number of players to 2 and the maximum to 4, so a lobby is ready when at least 2 players are in.

A lobby also has a lobby admin that is the creator of the lobby or an other player that took the admin place when they disconnected. As shown on the sequence diagram only the admin will receive a positive response when he tries to start the match.
The start of the match causes an asynchronous notification that updates the model of all the clients and notifies them that the game has started.



Sequence diagram: two players making moves

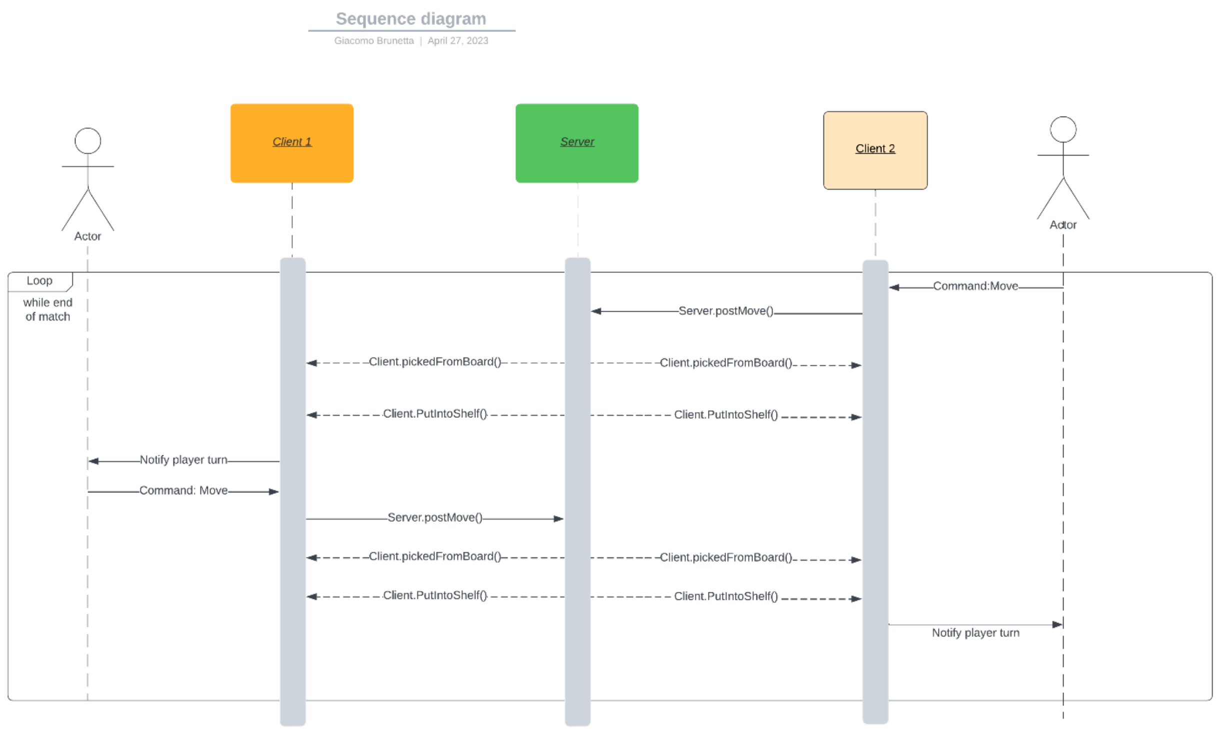
Once the match has started, players can update the status of the match by posting moves.

Posting a move will result in an actual change of the server model only if the client posts a move when it is their turn, otherwise the server will notify the user that they cannot post a move at that time.

After the move has been posted all the players receive a model update and the player who has to post the next move is notified that it's their turn so that players always know if it's their turn or not.

The update on the server is done by the client calling the *Server.postMove()* method and the udate of the clients is done by the server calling *Client.putIntoShelf()*.

Client.refreshBoard() and *Client.refreshShelf()* are called only when the game starts or when a disconnected player re-connects to the lobby and reload the Board and Shelf objects from scratch, while *Client.putIntoShelf()* triggers an update of the client by passing a Move object that will update the copy of Board and Shelves without reloading them.


Sequence diagram
Giacomo Brunetta | April 27, 2023
Actor
Client 1
Server
Client 2
Actor
Loop
while end of match
Command:Move
Server.postMove()
Client.pickedFromBoard()
Client.pickedFromBoard()
Client.PutIntoShelf()
Client.PutIntoShelf()
Notify player turn
Command: Move
Server.postMove()
Client.pickedFromBoard()
Client.pickedFromBoard()
Client.PutIntoShelf()
Client.PutIntoShelf()
Notify player turn