Communication Flow

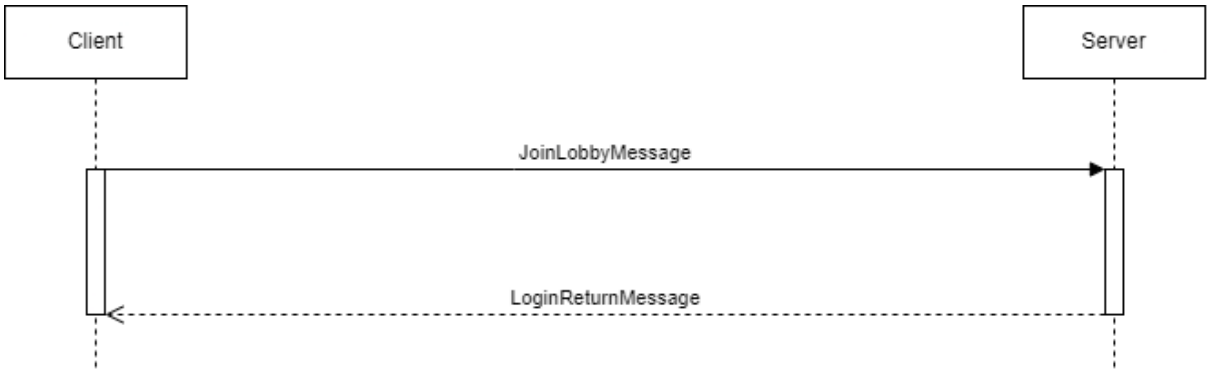
Introduction

This document presents the network interactions between the client and the server. Since the specs require that both TCP Socket and RMI must be implemented. Considering the RMI paradigm, each of the following message descriptions must be considered a method invocation and the response as a return value.

Login Phase

In this phase the user sends a *JoinLobbyMessage* to the server with the nickname and the server responds with a *LoginResultMessage* that has a false value if there is already a user with the same nickname, or with a true value if the player was created successfully.

If the *LoginResultMessage* is true, the client can proceed to the game setup phase.

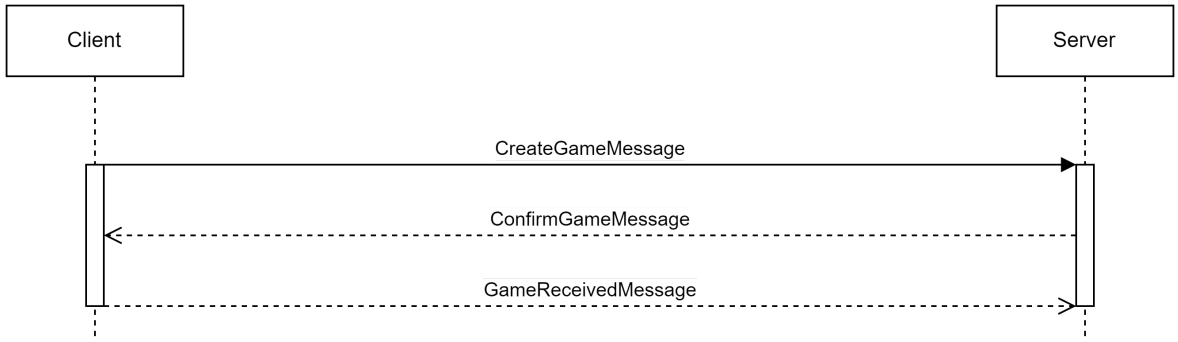


Game Setup Phase

In the Game setup phase the client can choose to either create a game or participate to an available game. If the player wants to create a new game he also needs to specify the number of players for the game.

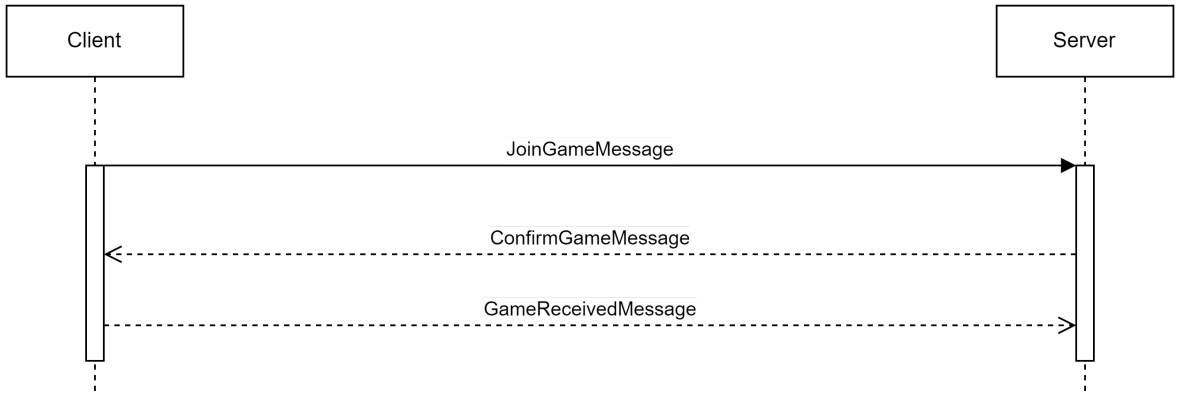
Game creation

The game creation is handled by the lobby controller's method *CreateGame(String player, int nPlayers)*. The client sends the number of players that will join the game. The server will respond with a *ConfirmGameMessage* which will tell the client if its request was processed properly or there were any errors. To finish the client will send a *GameReceivedMessage* to notify the server if it has received everything properly or not.

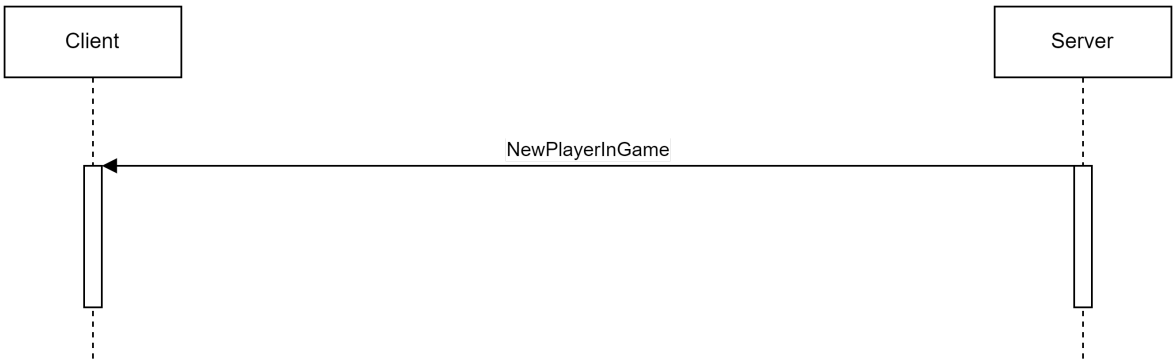


Joining a game

The joining in a game is handled by the lobby controller's method *addPlayerToGame(String player)*.



When the player joins a game, the server notifies all the connected clients to the game with the necessary informations. With the confirm message the client will be subscribed to all the necessary listeners of the new player (for example his bookshelf).



Rejoin game

Let's assume a client was in a game and for some kind of network problems the connection between the client and the server fell down. Since we decided to implement the advanced functionality ***resiliency to disconnections***, the client has the possibility to re-enter the game just by signing in with the nickname that was using during the previous game. The client asks the server to join the lobby, the server finds out that there is a game with inside a crashed player with the same nickname as the one trying to enter the lobby, so the server adds the player directly to the game, subscribes the player to all the necessary listeners and finally sends a *ConfirmGameMessage* to the client.
After the client receives the *ConfirmGameMessage* with a positive result, it sends to the server a *GameReceivedMessage* to confirm to the server that everything was received properly. From now on the client is considered to be in a game and the communication between it and the server is described in the next sections.

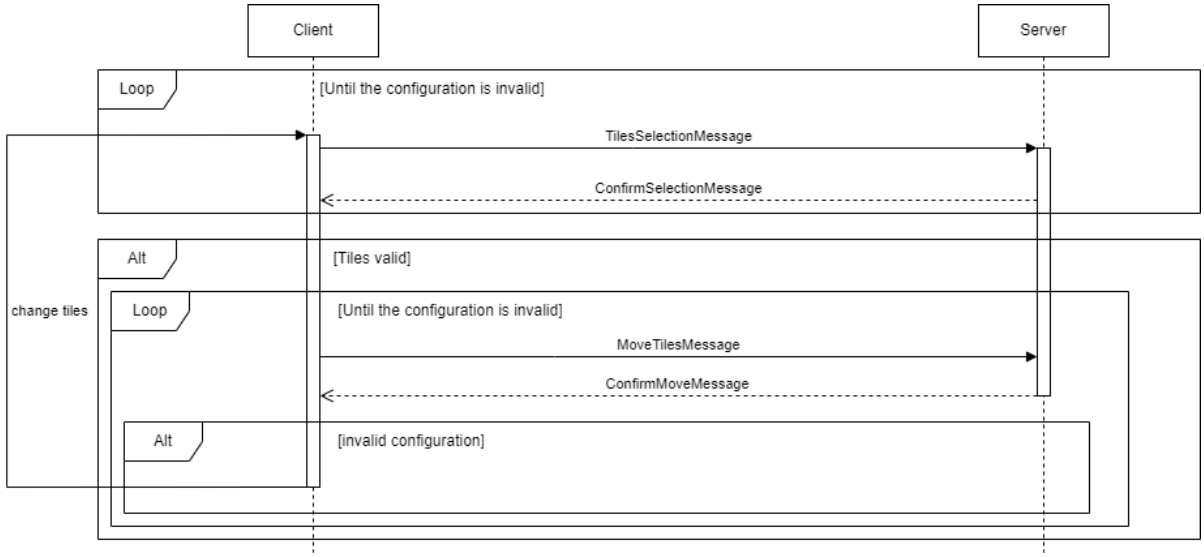
Turn Phase

In this phase the client selects the tiles from the LivingRoomBoard and the server responds with a *ConfirmSelectionMessage* to notify if the tiles are valid or not. This is done in loop until the selection is valid.

After that the client selects in which column to move the tiles selected to and the server responds with a *ConfirmMoveMessage*. As before this is done in loop until the column selected is valid, the client has also the possibility to change the initial selection if the column is invalid.
If everything is valid the server moves the tiles and notifies the client with a successfull *ConfirmMoveMessage*.

After the player has done its move all the clients will be notified via listeners about what changed in the player boards, *BookshelfUpdateMessage*, the living room board,

BoardUpdateMessage, and current points, *PointsUpdateMessage*, or new tokens, *TokenUpdateMessage*.



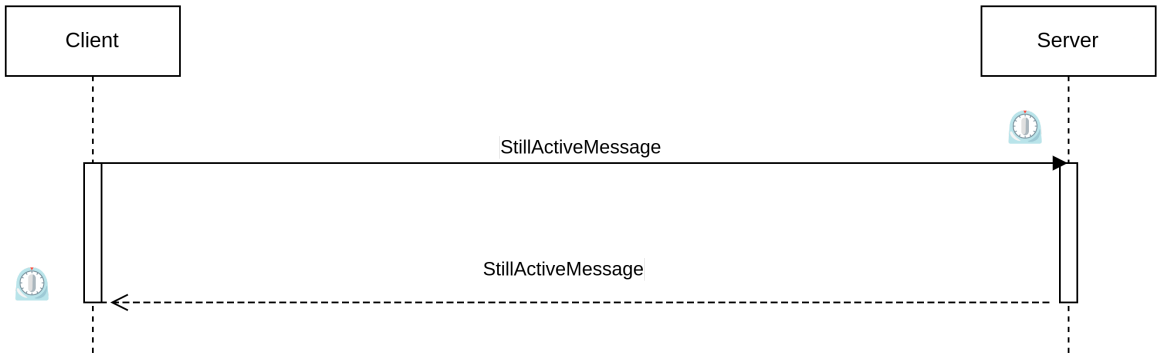
End Game Phase

In this phase the server notifies the client (via listener) that the game is ended and there is a winner. After this the client has two options, the first is to close the program, *exit*; the second option is to go back to the lobby and if the user wants it can re-enter the lobby and play another game, *Back to lobby*.

Check active client

This communication has the purpose to check if a client is still active or not. When a client connects to the server, the server will start a timer. If the server recieves a *StillActiveMessage* or any other message from the client, it resets the timer and the client is considered active, otherwise the client will be considered as crashed and the

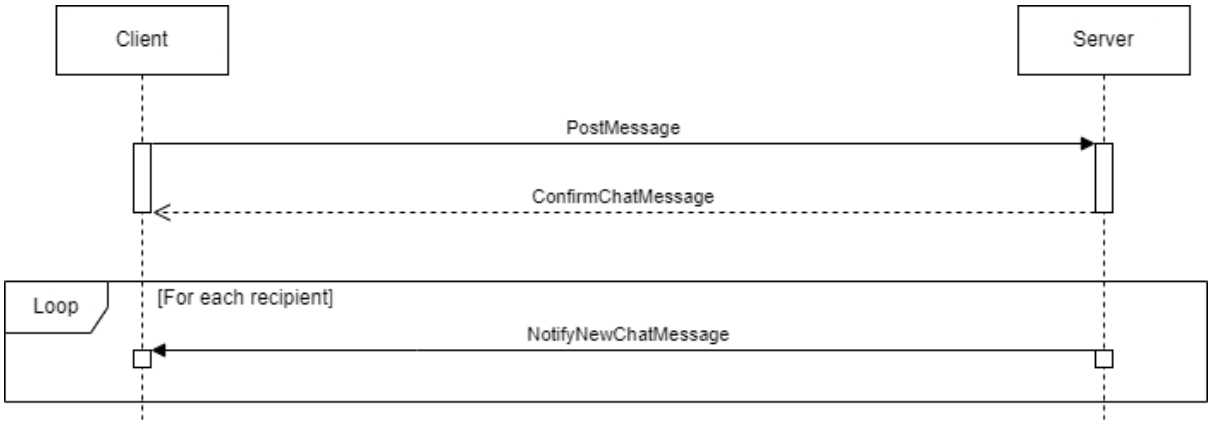
server will handle it properly. Note that the player's timer is resetted every time he interacts with the server.



Chat

This communication happens when a client wants to post a message inside the chat. The client sends a *PostMessage* with the info of the message and the server responds with a *ConfirmChatMessage* with the info if the message was posted succesfully or there was an error.

The second phase is when there is a new message for the player. In this case the server notify the client with a *NotifyNewMessage*. This is done for each recipient of the message.



Listeners notifications

In the model, some changes on the state of the classes are notified to a list of subscribers in the client, this permits the client to update the status of the view of his elements, for example the livingroom board, or the points of every player.

Here we illustrate a generalization of the listener communication, in the messages list there are illustrated all the listeners messages.

Messages list

For each type of message the way they will be serialized can change depending on the needs; in general we decided to write/read the Java object using .writeObject()/.readObject(). So the way we implement things might be a little different than what we wrote in this document; the fundamental ideas are the same though.

Each Message is sent/read as a java serialized object using *.writeObject()/.readObject()*. For every message are also provided the attributes of the corresponding java class.

Login Phase

JoinLobbyMessage

```
//JoinLobbyMessage structure
{
  //String, contains the player username
  username: "playerUsername"
}
```

LoginReturnMessage

This message tells the client if the login request to the lobby has been accepted or otherwise.

```
//LoginResultMessage structure
{
  //boolean, used to confirm the login has performed successful
  confirmLogin: true | false,
  //boolean, tells the client if it has taken the place of a previos crashed player
  confirmRejoined: true | false,
  //string, used for communicate details about the eventual error
  details: "String"
}
```

Game Setup Phase

CreateGameMessage

The client sends this message to tell the server to create a game with a certain number of players.

```
//CreateGameMessage structure
{
  //integer, defines the number of player for the game istance
  playerNumber: 3
}
```

ConfirmGameMessage

With this message the server tells the client if the request to create a game has been accepted or refused and why.

```
//ConfirmGameMessage structure
{
  //boolean, used to notify the result of the create game request
  confirmGameCreation: true | false,
  //boolean, used to notify the result of the join game request
  confirmGameJoined: true | false,
  //String, defines the type of the error, if an error occured
  type_of_error: "",
  //string, used for communicate details about the eventual error
  details: "String"
}
```

GameReceivedMessage

With this message the client tells the server that it received the *ConfirmGameMessage* and if any errors occured.

```
//GameReceivedMessage
{
  //boolean, used to notify the server if any error
  //occured during the local game creation/join
  errorOccured: true | false
}
```

JoinGameMessage

This message is used when a client wants to enter in a random game.

```
//JoinGameMessage structure
{
  //String, contains the player username
  username: "playerUsername"
}
```

NewPlayerInGame

This message contains all the information needed to the players in game about the new player, all the players already part of the game will be notified about the joining. The message contains the *player username*.

```
//NewPlayerInGame structure
{
  //String, contains the player username
  newPlayerUsername: "playerUsername"
}
```

AlreadyJoinedPlayersMessage

This message is used by the server to tell the client which player were in the game before he entered.

```
//AlreadyJoinedPlayerMessage
{
  //Set<String>, represents who was already in the game before the new user entered
  alreadyJoinedPlayers: ["x","y",...]
}
```

Turn Phase

TilesSelectionMessage

The client sends to the server the coordinates of the tiles it wants to retrieve from the living room board.

```
//TilesSelectionMessage structure
{
  //array of coordinates, minimum 1, maximum 3 coordinates
  tiles:[
    {row: 3, column:4},
    {row: 2, column:1},
    {row: 1, column:1},
  ]
}
```

ConfirmSelectionMessage

This message has the purpose to notify the client if the selection we sent him via *TilesSelectionMessage* is valid or invalid.

```
//ConfirmSelectionMessage structure
{
  //boolean, used to confirm that the selection is legal
  confirmSelection: true | false,
  //optional field, express the error type. used only if confirmSelection is false
  type_of_error: "...",
  //string, used for communicate details about the eventual error
  details: "String"
}
```

MoveTilesMessage

Send the server the move the client wants to make. The message will specify the tiles selection and the column of the user's personal board.

```
//ConfirmSelectionMessage structure
{
  //String, contains the player username
  username: "playerUsername",
  //array of coordinates, minimum 1, maximum 3 coordinates,
  //they are the same selected before, the order represent the insertion order
  tilesCoordinates:[
    {row: 3, column:4},
    {row: 1, column:1},
```

```
    {row: 2, column:1}
  ],

  //integer, the column used for the insertion
  column: 3
}
```

ConfirmMoveMessage

This message is used to tell the client if the requested column is usable, if that is the case then the move will be executed and the server sends this message to notify the client about the result of the move.

```
//ConfirmMoveMessage structure
{
  //boolean, used to confirm that the selection is legal
  confirmSelection: true | false,
  //String, used to communicate what type of error occured
  errorType: "errorType",
  //String, used for communicate details about the eventual error
  details: "String"

}
```

Connection handling

StillActiveMessage

Used to check if the client and the server are still active. Both clients and server have a timer that is rescheduled every time they receive a message. The still active message is sent every 5 seconds, in order to ensure both the client and the server are not crashed. The Message does not contain information, but it is only used for ensuring the connection.

CloseConnectionMessage

Used by a client or server to notify the other that the connection between the two is being closed from one side. Example: *Hey i'm closing the connection, consider me as crashed if i'm in a game.*

Chat

PostMessage

Sent by the client to post a new message into the game chat. The chat message can be broadcast or private to a single user.

```
//PostMessage structure
{
  sender: "user1",
  //"broadcast" -> broadcast message
  recipient: "user2" | "",
  content: "Text of the message",
}
```

ConfirmChatMessage

Used by the client to confirm that the message has been posted in the game chat.

```
//ConfirmMessage structure
{
  result: true | false //depending the result of the post procedure on the server
}
```

NotifyNewChatMessage

This message is used by the server to notify a client that a new message has been posted in the game chat. If the recipient doesn't equal the client's username (and it is not broadcast), the message must be "forgotten" by the client, because there must have been an error ,the message was not meant to be sent to this client and so the user can't see its content.

```
//NotifyNewChatMessage structure
{
  sender: "x",
  recipient: "y" | "", //"" -> broadcast message
  content: "Text of the message"
}
```

Listener messages

BoardUpdateMessage

The message that is sent from the server when the board changes his status.

```
//BoardUpdateMessage structure
{
  tilesInBoard:
              {
                {0, 1}: "BOOK",
                {2, 3}: "CAT",
                {4, 6}: "EMPTY",
                 ...
              }
}
```

BookshelfUpdateMessage

As for the living room board, any bookshelf update is notified, the notification includes the index of the column that is updated and the elements added in order (from one, to three maximum elements).

```
//BookshelfUpdateMessage structure
{
  username: "player",
  column: 4,
  insertedTiles: [ "CAT","CAT","BOOK"]

}
```

Player listeners messages

The change of states in the player triggers many actions, such as the new points of the player, the assignment of a scoring token from the common card and the assignment of the cards. Those information can be sent in three different messages:

TokenUpdateMessage

This message is sent when the player receive a token from a common goal card (the tokens can be maximum three: two from the common goal cards, one from the first player token, the one that is assigned to the player that completes the bookshelf for first).

```
//TokenUpdateMessage structure
{
  username: "player",
  //ArrayList of the tokens assigned to the player
```

```
  tokens: [ SIX, FOUR ]

}
```

PointsUpdateMessage

It is used to notify the new points assigned to the player. In this example the player had a starting point of 6, he earned 4 points, and has now an overall of 10.

```
{
  username: "username",
  totalPoints: "10",
  addedPoints: "4"
}
```

PersonalGoalCardUpdateMessagge

Used by the server to send the clients the updated personal goal card of a certain player

```
//PersonalGoalCardUpdateMessagge
{
  player: "username",
  //RemotePersonalGoalCard, updated PersonalGoalCard of the player
  card: obj
}
```

Game listener messages

The game listener sends notifications about the player that have won the game, but also notify all the players with informations about the players that join the game.

NewPlayerInGame

It is used to notify a new player that has joined the game.

```
{
  newPlayerUsername: "playerJoined"
}
```

EndGameMessage

It contains information about the player that has won the game, and the ranking

```
{
  winningPlayer: "winner"
  ranking:{ "winner": 50,
            "player2": 30,
          }
}
```

CommonGoalCardsUpdateMessage

Notify the client about what has changed in the commong goal cards

```
//CommongGoalCardsUpdateMessage
{
  //ArrayList<RemoteCommonGoalCard>, list of the updated common goal card
  //Each object in the array is a single RemoteCommonGoalCard
  commongGoalCards: [obj1, obj2,...]
}
```

GameStatusMessage

Used by the server to notify the client about the state change of the game

```
//GameStatusMessage
{
  // new GameState
  state: CREATED | STARTED | PAUSED | CRASHED | RESUMED | ENDED
}
```

NotifyPlayerCrashedMessage

Used by the server to notify the other clients that a player has crashed

```
//NotifyPlayerCrashedMessage
{
  //Username of the player that crashed
  userCrashed: "username"
}
```

NotifyPlayerInTurnMessage

Used by the server to tell each client whose the current turn.

```
//NotifyPlayerInTurnMessage
{
  //String, username of the player that is in turn
  userInTurn: "username",
  //boolean, true if the current turn is mine
  yourTurn: true | false
}
```

NotifyTurnOrder

Used by the server to declare to the clients the final turn order.

```
//NotifyTurnOrder
{
  //ArrayList<String>, list of the player ordered by turn order
  playerOrder: ["x","y", ...]
}
```

NotifyWinnerPlayerMessage

Used to tell the client that the game has a winner and what is the final scoreboard of the game.

```
//NotifyWinnerPlayerMessage
{
  //String, username of the winner
  winningUser: "x",
  //int, final points of the winner
  points: 33,
  //Map<String, Integer>, final points for each player
  scoreboard: ["x": 33, "y": 10, ...]
}
```