# Communication Protocol

Salim Salici, Lorenzo Trombini, Niccolò Benetti, Matteo Scarlino
Gruppo AM49

## Overview

The communication protocol we have developed allows the server to send exceptions to the client. This is immediate through **Java RMI**, which allows methods to be invoked on remote objects with the invoker receiving return values and exceptions.
We have designed a **socket** communication implementation (java.net.Socket and java.net.ServerSocket) that also allows the invoker to receive return values/exceptions after invoking methods on remote objects.

The various clients and the server exchange messages through their respective interfaces, one provided by the client and one provided by the server. The methods of these two interfaces are listed below.

**Server interface**

- `fetchRooms`
- `createRoom`
- `joinRoom`
- `leaveRoom`
- `readyUp`
- `readyDown`
- `executeAction`
- `reconnect`
- `ping`
- `getClientHostAddress`
- `chatMessage`

**Client interface**

- `roomUpdate`
- `receiveGameUpdate`
- `receiveChatMessage`
- `startHeartbeat`
- `stopHeartbeat`

As can be seen, these interfaces primarily provide methods whose function is to orchestrate the organization of the various clients in their respective game rooms (we chose to implement the additional feature of "multiple games"). Messages related to the actual conduct of a game are passed through the underlined methods and listed below:

`executeAction(Client client, GameAction action)` $[Client \rightarrow Server]$

`receiveGameUpdate(GameUpdate update)` $[Server \rightarrow Client]$

# Actions

The abstract class `GameAction` represents a game action that a player wants to perform. Each GameAction has a field `String username`. The different actions that extend GameAction add their specific fields. All GameActions are sent from clients to the server.

| GameAction | Description | Additional Fields |
|---|---|---|
| JoinGameAction | This action communicates the actual entry of a player (with associated username) into a game | |
| ChooseStarterSideAction | This action communicates the player's choice to play their starter card flipped or not | `boolean flipped` |
| ChooseObjectiveAction | This action communicates the objective card chosen by the player | `int objectiveId` |
| PlaceCardAction | This action communicates the placement of a card on the board by the player. The player must specify where to place the new card and whether to play it flipped or not | `int cardId`<br>`int parentRow`<br>`int parentCol`<br>`CornerPosition cornerPosition`<br>`boolean flipped` |
| DrawCardAction | This action allows the player to draw a card from a position they specify | `DrawPosition drawPosition`<br>`int idOfRevealedDrawn` |

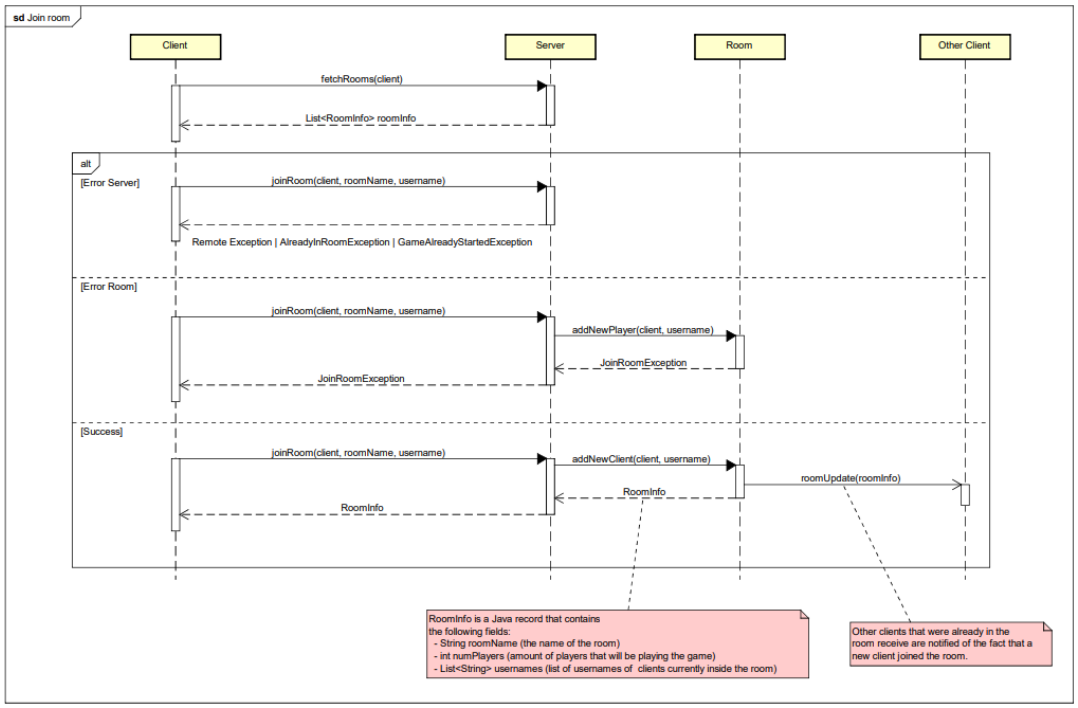
## Game Updates

The abstract class **GameUpdate** represents a game state update that the server notifies to the various clients.

| GameUpdate | Description | Additional Fields |
|---|---|---|
| PlayerOrderUpdate | This update communicates the order in which players will play in each round | `List<String> playerOrder` |
| CardPlacedUpdate | This update communicates the placement of a card in the game area by the player with the respective username, with additional information regarding points and visible symbols in the game area of the player who placed the card | `String username`<br>`int cardId`<br>`Position position`<br>`boolean flipped`<br>`Map<Symbol, Integer> activeSymbols`<br>`int points` |
| HandUpdate | This update communicates to the player who just drew a card the IDs of the cards in their hand (including the one just drawn) | `String username`<br>`List<Integer> handIds` |
| HiddenHandUpdate | This update communicates to all players the hand of a player indicating only the type of resource of each card, for the purpose of representing the back of their hand | `String username`<br>`List<Resource> hiddenHand`<br>`List<Boolean> isGold` |
| GameStateChangedUpdate | This update communicates that the game has transitioned to a different game state, indicating the current turn, round, and the name of the current player | `GameStateType gameStateType`<br>`int turn`<br>`int round`<br>`String currentPlayer` |
| StarterCardAssignedUpdate | This update communicates to each player their starter card | `String username`<br>`int starterCardId` |
| ChoosableObjectivesUpdate | This update communicates to each player the objective cards they can choose from | `String username`<br>`List<Integer> objectiveCards` |
| ChatMSG | This update represents a chat message sent between players | `String text`<br>`String sender`<br>`String recipient` |

| | | |
|---|---|---|
| DrawAreaUpdate | This update represents the state of the draw area in the game, including information about remaining resources and golds, the top card on the deck, and revealed resources and golds | `int remainingResources`<br>`int remainingGolds`<br>`Resource deckTopResource`<br>`Resource deckTopGold`<br>`List<Integer> revealedResources`<br>`List<Integer> revealedGolds` |
| EndGameUpdate | This update provides the final game standings, including points for each player, the number of completed objectives, and the personal objective ID | `Map<String, Integer[]> playerToPoints`<br>`String forfeitWinner` |
| GameStartedUpdate | This update provides information about the game that has just started, including player order, colors, and initial cards | `String username`<br>`int starterCardId`<br>`HashMap<String, Color> playersToColors`<br>`List<Integer> commonObjectivesIds`<br>`int remainingResources`<br>`int remainingGolds`<br>`Resource deckTopResource`<br>`Resource deckTopGold`<br>`List<Integer> revealedResourcesIds`<br>`List<Integer> revealedGoldsIds` |
| IsPlayingUpdate | This update is recived by players that are already in a room when a new player joins | `String username`<br>`Boolean status` |
| PersonalObjectiveChosenUpdate | This update indicates that a player has chosen their personal objective | `String username`<br>`int objective` |

# 1 Join Room




Initially, the client initiates the process by sending a `fetchRooms(client)` request to the server. In response, the server provides a list of available rooms. Following this, the client sends a `joinRoom(client, roomName, username)` request to the server to join a specific room. The server processes the join room request and can encounter several scenarios:

### Success

In the success case, where the join room request is successful, the server calls `addNewPlayer(client, username)` on the room instance to add the new client to the room. The server then sends a `RoomInfo` object back to the client. This `RoomInfo` object includes the room name (`String roomName`), the maximum number of players that will be playing the game (`int maxPlayers`), and a map that associate usernames of clients currently inside the room with their totem's color (`Map<String, Color> playerToColor`). Additionally, the server notifies other clients already in the room.

### Server Error

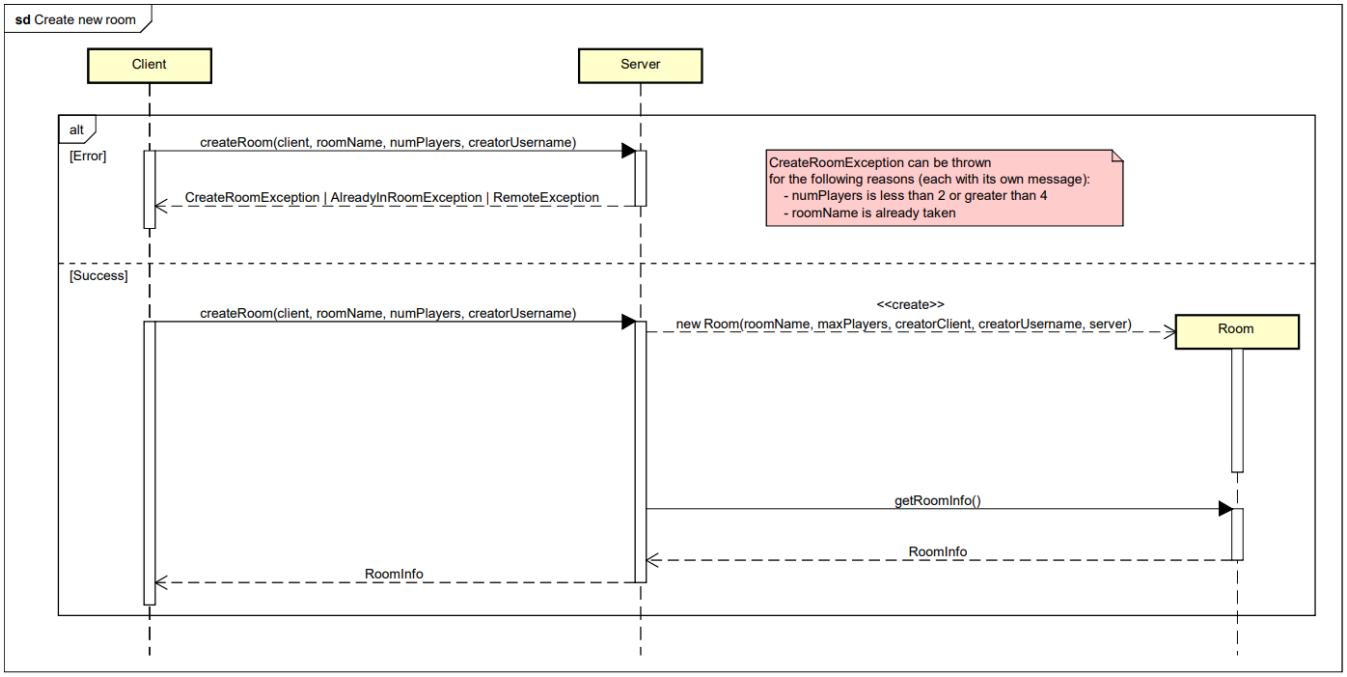
If there is an issue on the server side, the following exceptions can be thrown:

- `AlreadyInRoomException`: This exception indicates that the client is attempting to join a room that they are already part of, and therefore the request cannot be processed.
- `RemoteException`: This exception indicates that there was a communication error between the client and the server, preventing the request from being completed.
- `GameAlreadyStartedException`: This exception indicates that the client is trying to join a room where the game has already started, and therefore new clients cannot join.

### Room error

If there is an issue with the specified room, a `JoinRoomException` is thrown.

# 2 Create Room




The client sends a `createRoom(client, roomName, numPlayers, creatorUsername)` request to the server to create a new room with the specified parameters. From here there are two possible scenarios:

### Success

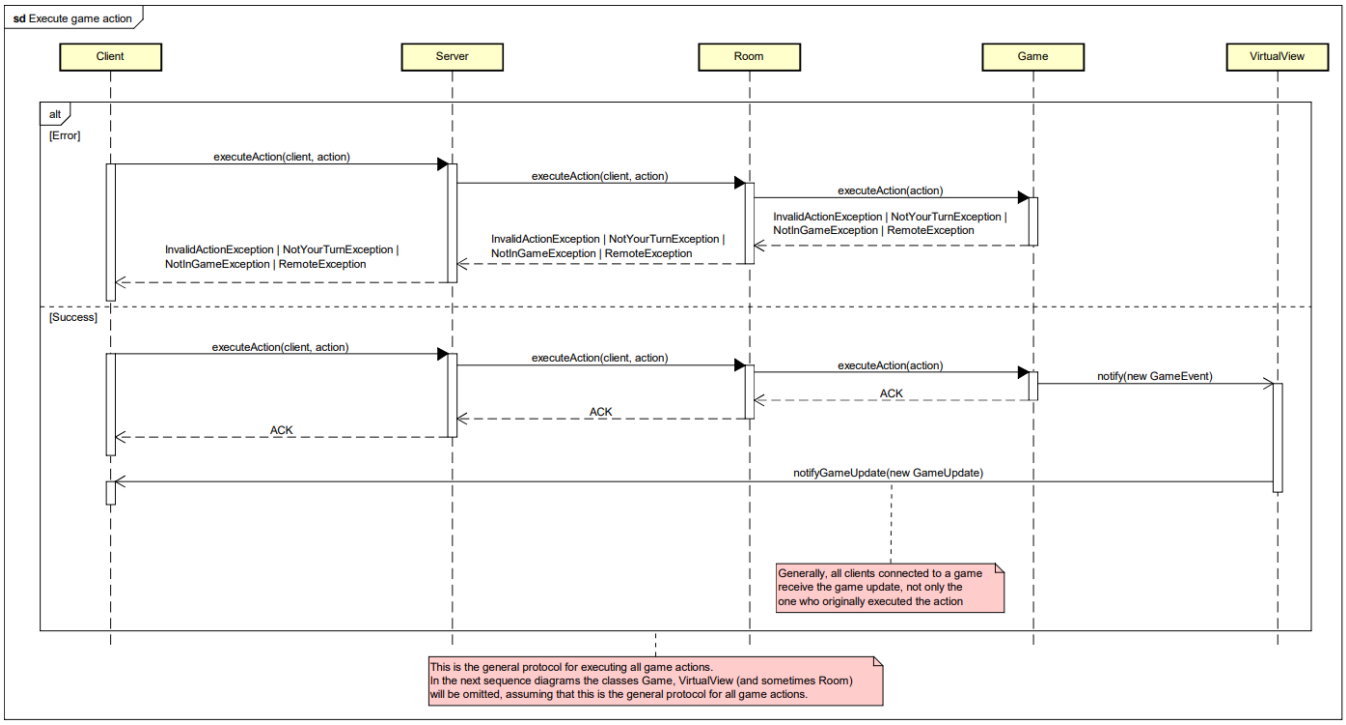
In the success case, where the create room request is successful, the server initiates the creation of a new room by calling the `new Room(roomName, maxPlayers, creatorClient, creatorUsername, server)` constructor. This involves the following steps: The `Room` object is created with the provided parameters. The server calls `getRoomInfo()` on the `Room` object to retrieve the `RoomInfo`. The server sends the `RoomInfo` object back to the client.

### Error

If there is an error in the room creation process, the following exceptions can be thrown:

- `AlreadyInRoomException`: Indicates that the client is already in another room and cannot create a new room until they leave the current one.
- `RemoteException`: This exception indicates that there was a communication error between the client and the server, preventing the request from being completed.
- `CreateRoomException`: This exception can be thrown if the number of maximum players specified for the room is less than 2 or greater than 4, or if the roomName is already taken.

# 3 Execute game action




Initially, the client sends an `executeAction(client, action)` request to the server to execute a specific game action.

### Success

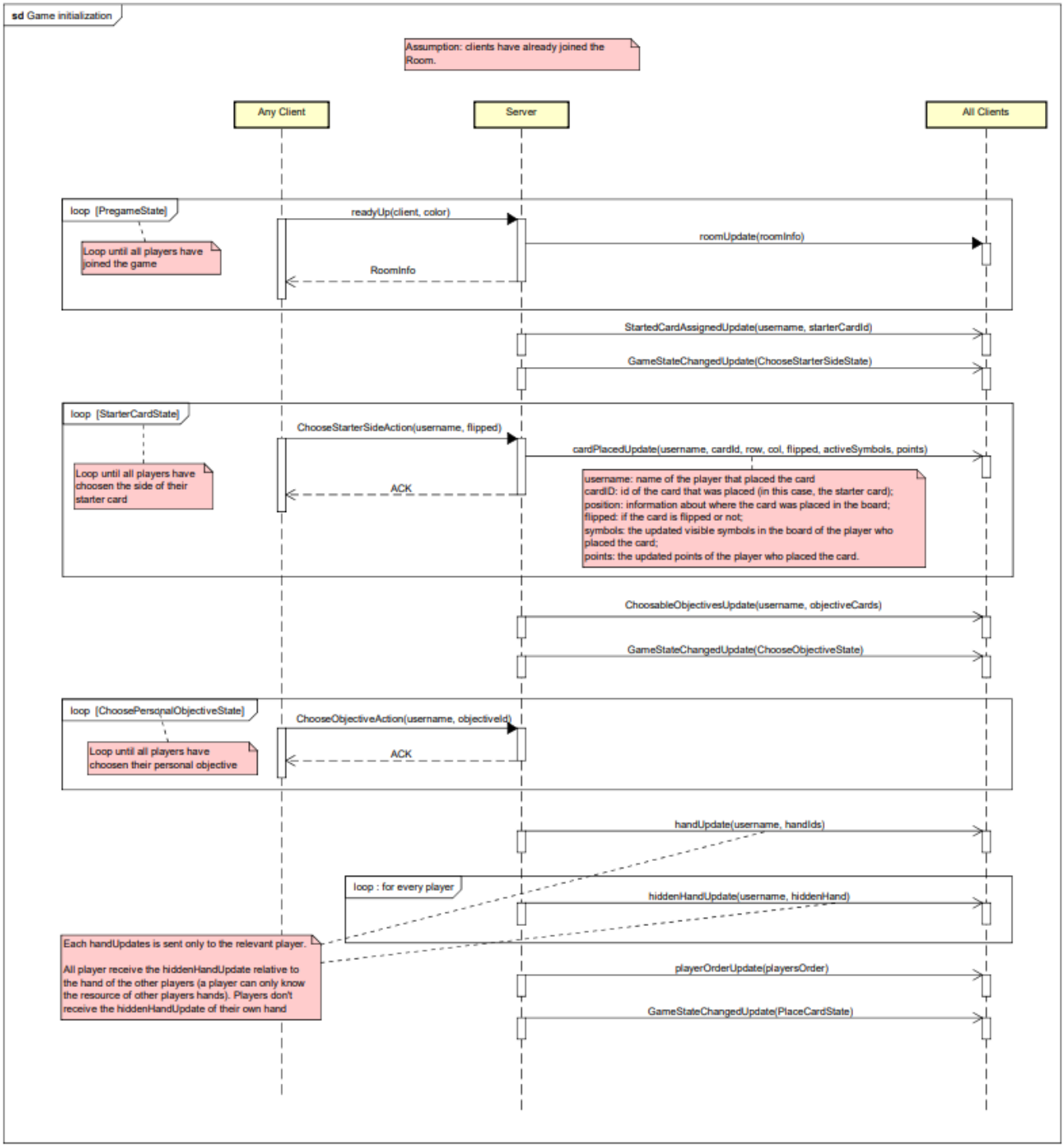
In the success case, where the execute action request is successful, the server processes the action by calling `executeAction(client, action)`. The server forwards the action to the `Room` component by calling `executeAction(client, action)`. The `Room` component then processes the action and calls `executeAction(action)` on the `Game` component. The `Game` component acknowledges the successful execution of the action by returning an `ACK`. The server sends an `ACK` back to the client to confirm the successful execution. Additionally, the server notifies all connected clients of the game update by calling `notifyGameUpdate(new GameUpdate)`.

### Error

If there is an error during the execution of the action, the following exceptions can be thrown:

- `InvalidActionException`: Indicates that the action performed is not valid in the current game state.
- `NotYourTurnException`: Indicates that the client is attempting to perform an action when it is not their turn.
- `NotInGameException`: Indicates that the client is attempting to perform an action when they are not part of the game.
- `RemoteException`: Indicates that there was a communication error between the client and the server, preventing the request from being completed.

# 4 Game initialization




It is assumed that all clients have already joined the room.

## Pregame State

In this state, any client can send a `readyUp` request to the server to indicate that they are ready. The server processes this request and responds with the `RoomInfo`, confirming the client's readiness. The server also sends a `roomUpdate` message to all clients, updating the status of the player who is ready.
Once all players have joined and are ready, the room creates a new Game (not shown in the diagram), and the server

assigns a starter card to each players by sending a `StartedCardAssignedUpdate` to each client. Finally, the server sends a `GameStateChangedUpdate(ChooseStarterSideState)` to all clients to transition to the next state, where players will choose the side of their starter cards.

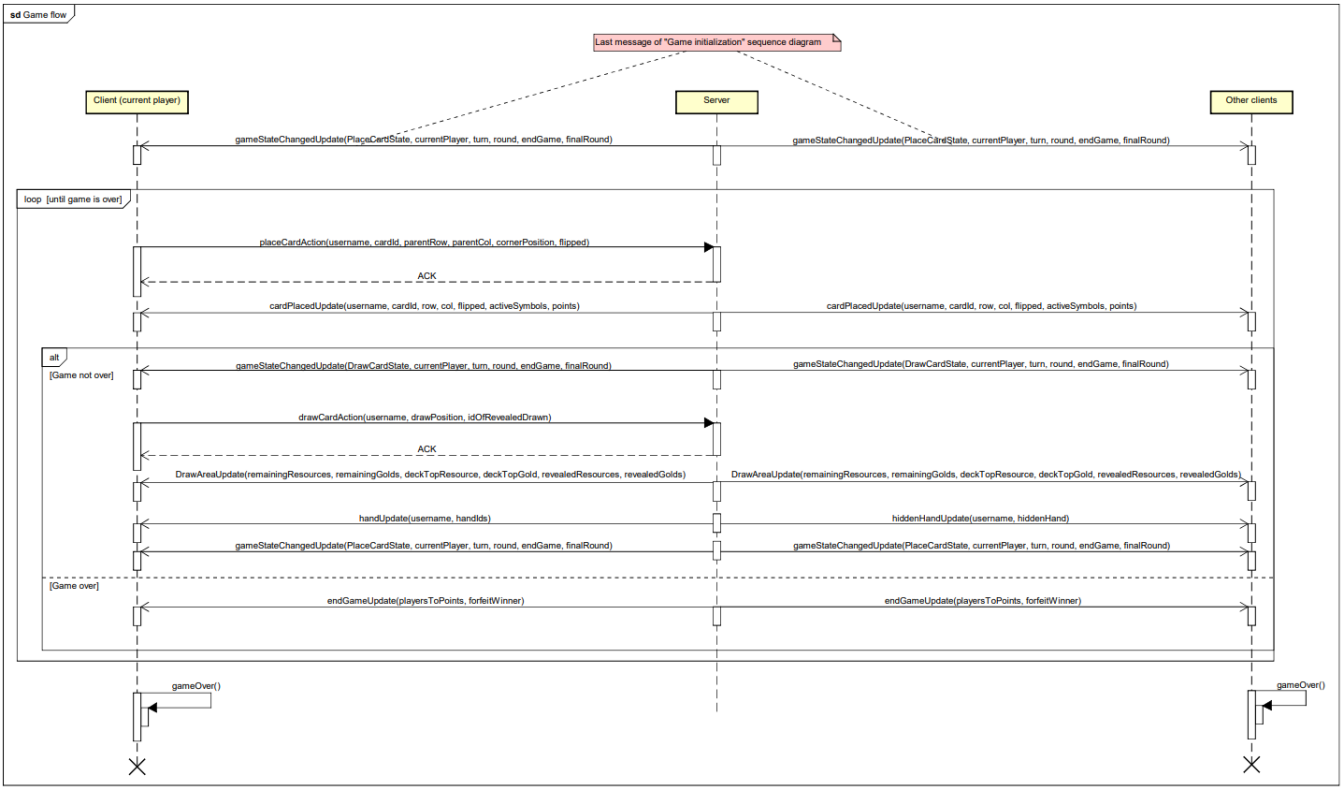
## Starter card

Players send a `ChooseStarterSideAction` request to the server. The server also sends a `cardPlacedUpdate` to all clients, providing details about the placed starter card.
After all players have chosen their starter card sides, the server sends a `ChoosableObjectivesUpdate` to each player, followed by a `GameStateChangedUpdate(ChooseObjectiveState)` to transition to the next state.

## Personal objective

Players send a `ChooseObjectiveAction` request to the server.
When all players have selected their objectives, the server sends the `handUpdate` and the `hiddenHandUpdate` to all other players. The server then sends a `playerOrderUpdate` to all clients, indicating the order in which players will take their turns.

# 5 Game flow




The "Game Flow" process describes the sequence of actions and updates that occur during the game. Initially, the game flow starts with the last message of the "Game Initialization" sequence diagram, which is the `gameStateChangedUpdate` sent to the clients.

## Game not over

The main loop continues until the game is over. Within this loop, two primary actions occur: placing a card and drawing a card.

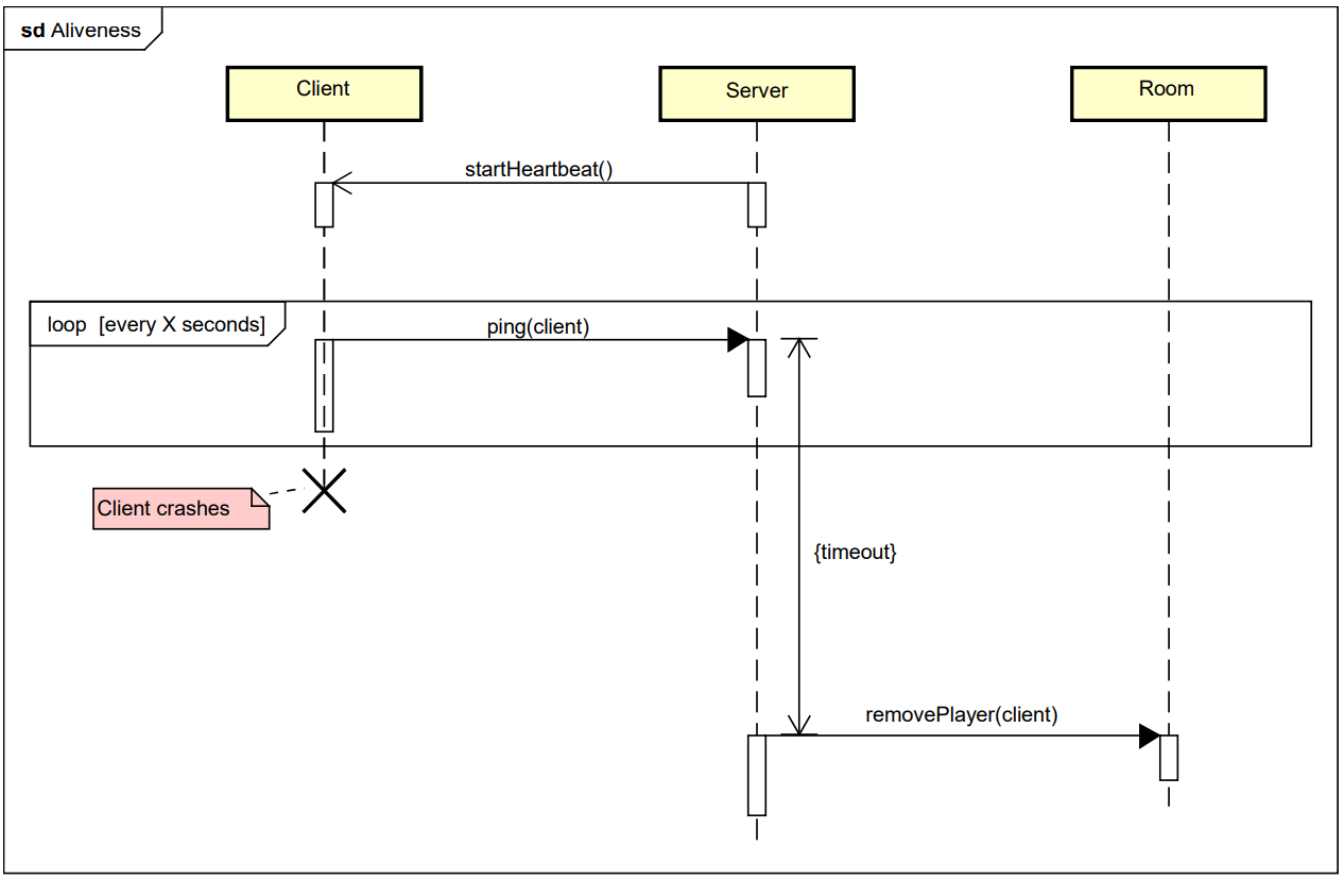
In the place card action, the client (current player) sends a `placeCardAction` request to the server. The server processes the request and responds with an `ACK`. The server sends a `cardPlacedUpdate` to all clients, including the client who performed the action. This update includes the details of the placed card, its position, whether it was flipped, the active symbols, and the updated points.
The server then updates the game state by sending a `gameStateChangedUpdate(DrawCardState, currentPlayer, turn, round, endGame, finalRound)` to all clients.

In the draw card action, the client (current player) sends a `drawCardAction` request to the server. The server processes the request and responds with an `ACK`. The server sends a `DrawAreaUpdate` to all clients. The server sends a `handUpdate` to the client who drew the card and a `hiddenHandUpdate` to all the other clients, providing the hidden hand information (rapresenting the back of the cards) of the player who drew the card.
The server updates the game state again by sending a `gameStateChangedUpdate(PlaceCardState, currentPlayer, turn, round, endGame, finalRound)` to all clients.

## Game over

If the game is over, the server sends an `endGameUpdate(playersToPoints, forfeitWinner)` to all clients, providing the final points of all players and indicating the forfeit winner if present.

# 6 Aliveness




The "Aliveness" process ensures that the client is active and responsive during the game. This involves periodic communication between the client and server to confirm the client's status. Below is a detailed description:

First of all the server sends a `startHeartbeat()` request to the client to initiate the heartbeat process.

## Ping loop

The loop runs every X seconds. Within this loop, the client sends a `ping(client)` request to the server to signal that it is still active. If the server does not recive a ping within a specified timeout period, it will detect the timeout.

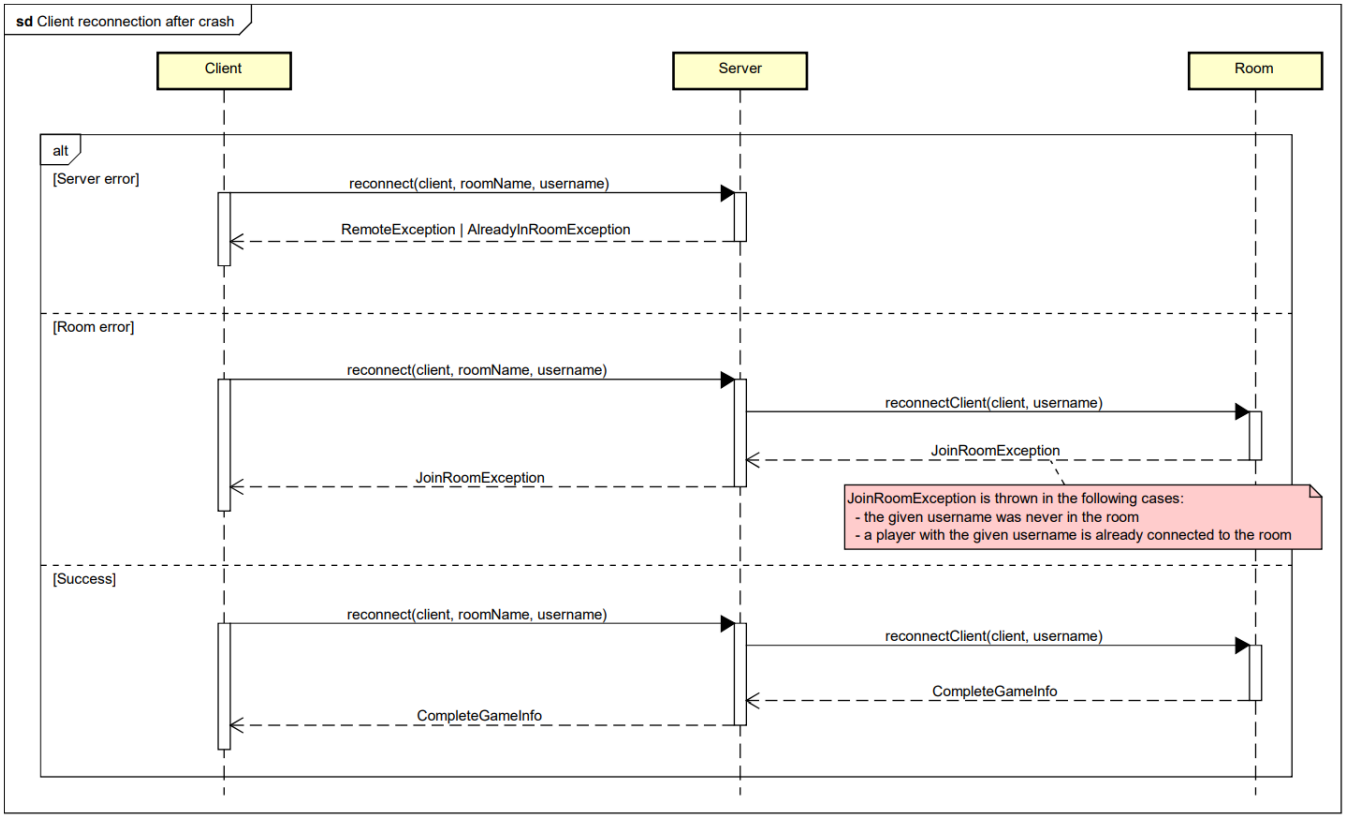
## Client crash

If the client crashes, the server will stop receiving pings. After detecting a timeout, the server will proceed to remove the player from the room by calling `removePlayer(client)`.

## Server crash

If the server crashes, the client's ping will be left hanging. After a timeout, the client will realize that the server has crashed and will go back to the server selection screen, where the user can connect to another server.

# 7 Client re-connection after crash




The client sends a `reconnect` request to the server to attempt reconnection to the specified room with the provided username.

### Server error

If there is an issue on the server side, several exceptions can be thrown:

- `RemoteException`: Indicates a communication error between the client and server.
- `AlreadyInRoomException`: Indicates that the client is already connected to the room.

### Room error

If there is an issue with the specified room, a `JoinRoomException` is thrown. This exception is raised in two cases:

- The given username was never in the room.
- A player with the given username is already connected to the room.

### Success

In the success case, where the reconnection request is successful, the server calls `reconnectClient` on the room instance to re-establish the client's presence in the room. The server then sends a `CompleteGameInfo` object back to the client, which includes all the necessary information for the client to resume the game.

## Socket comunication messages

In order to facilitate communication between clients and the server using sockets, a series of custom messages have been created. These messages enable the clients and server to perform various actions and exchange information. Each message includes a unique identifier. Below is a detailed list of the messages used for socket communication, categorized by whether they are sent by the client or the server.

### Messages Sent by the Server

| Message Name | Description | Additional fields |
|---|---|---|
| ReceiveChatMessageMTC | Message used to receive chat messages from the server. | `ChatMSG chatMSG` - The chat message content. |
| ReceiveGameUpdateMTC | Message used to receive game state updates from the server. | `GameUpdate gameUpdate` - The game state update. |
| RoomUpdateMTC | Message used to update the state of a game room. | `RoomInfo roomInfo` - Information about the room.<br>`String message` - Additional message. |
| StartHeartbeatMTC | Message used to start monitoring the heartbeat of the connection. | |
| StopHeartbeatMTC | Message used to stop monitoring the heartbeat of the connection. | |

### Messages Sent by the Client

| Message Name | Description | Additional fields |
|---|---|---|
| ChatMessageMTS | Message used to send chat messages between clients and the server. | `ChatMSG chatMSG` - The chat message content. |
| CreateRoomMTS | Message used to create a new game room. | `String roomName`<br>`int numPlayers`<br>`String creatorUsername` |
| ExecuteActionMTS | Message used to execute a game action. | `GameAction gameAction` - Action to be executed. |
| FetchRoomsMTS | Message used to fetch the list of available game rooms. | |
| GetClientHostAddressMTS | Message used to get the client's host address. | |

| | | |
|---|---|---|
| JoinRoomMTS | Message used to join an existing game room. | `String roomName`<br>`String username` |
| LeaveRoomMTS | Message used to leave a game room. | |
| PingMTS | Message used to send a ping between the client and server to check the connection. | |
| ReadyDownMTS | Message used to mark a client as not ready. | |
| ReadyUpMTS | Message used to mark a client as ready. | `Color color` - Color chosen by the client. |
| ReconnectMTS | Message used to request reconnection of a client to the server. | `String roomName`<br>`String username` |
| ReturnMessage | Message used to return a generic response from the server to the client. | `Object returnValue` - The value returned. |
| SocketMessage | Base message for socket communication. | |