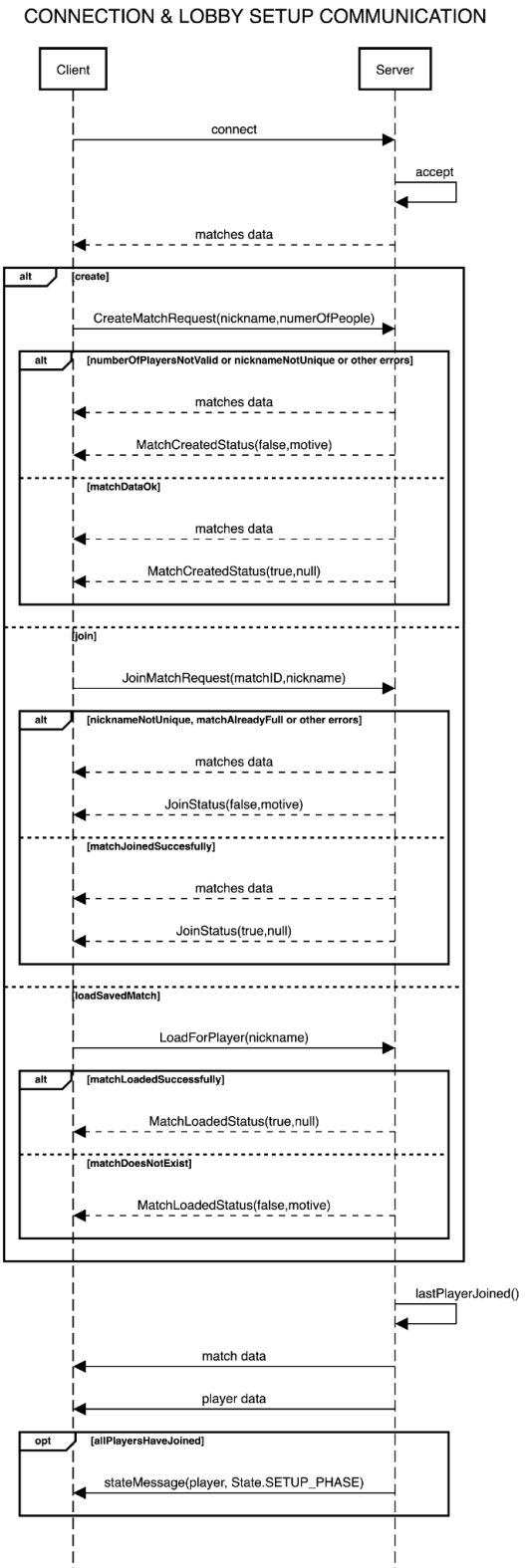
CLIENT SERVER COMMUNICATION PROTOCOL

We allow multiple matches to be created. Each player has a unique nickname, used to identify the clients in case of disconnection. The server and the client can always receive messages, but each message can only be processed if the client and the server are in the right states. In addition, when the server is waiting for a request from the client, a timer will start. If the client takes too long to send a request, the server will forcefully disconnect the client.

We use Gson for the serialization and deserialization of messages between the client and the server. Each message from the client to the server is divided into two classes. One visible to both client and server, containing attributes, and another one in the server which implements from an interface a method that performs operations in the server. In this way, the client doesn't know the methods used by the server. Using RuntimeTypeAdapterFactory.java and the interface, we reduce code duplication.

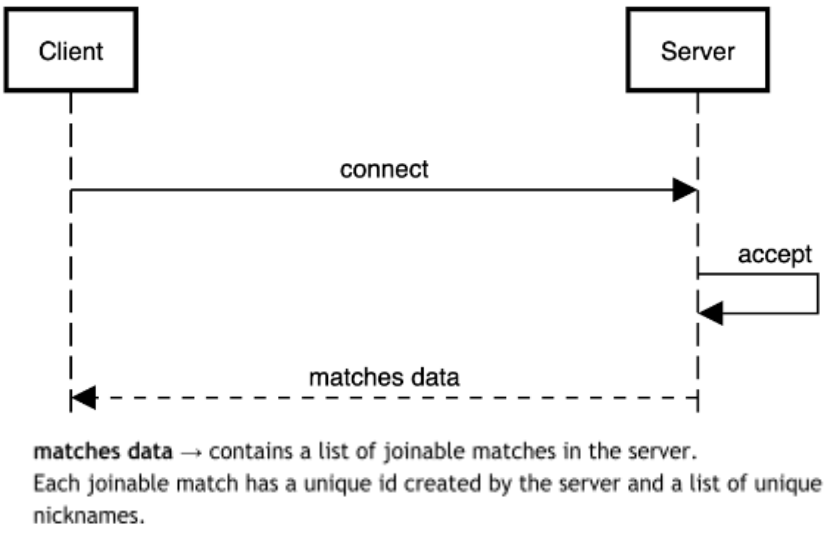
For messages from the server to the client, the logic is similar.

CONNECTION & LOBBY SETUP COMMUNICATION



1-Connection

The setup phase starts with a client connecting to the server. After logging into the server the player will receive a list of joinable matches. Each joinable match has a unique ID created by the server and a list of unique nicknames, belonging to currently online players in the lobby, waiting for the match to start.



matches data → contains a list of joinable matches in the server.
Each joinable match has a unique id created by the server and a list of unique nicknames.

Now there are 3 possibilities:

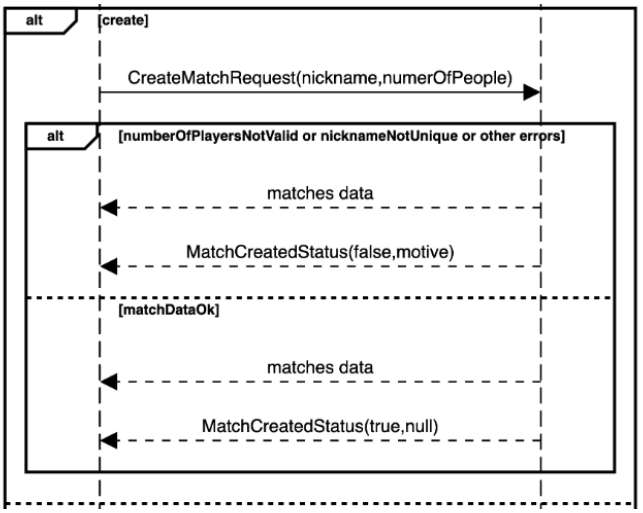
- Client requests to create a match
- Client requests to join a match
- Client requests to load a match

Each possibility has a different communication protocol, let's explore them in detail.

2-Client requests to create a match

The client requests to create a new match by sending a **CreateMatchRequest** message specifying the desired nickname and number of players. If the provided nickname has been already chosen by another player among matches and/or the requested lobby capacity is invalid, the server asks for correct parameters by sending a **MatchCreatedStatus(false, motive)** response.
When nickname and capacity are OK there are two possibilities:

- Capacity is greater than 1. After all the players enter the lobby each client will receive the match data, data of all the players and the game will start. There is a limited waiting time in the lobby, then the lobby is destroyed and the main menu view is shown.
- Capacity equals 1, the single-player match starts and the client is notified.

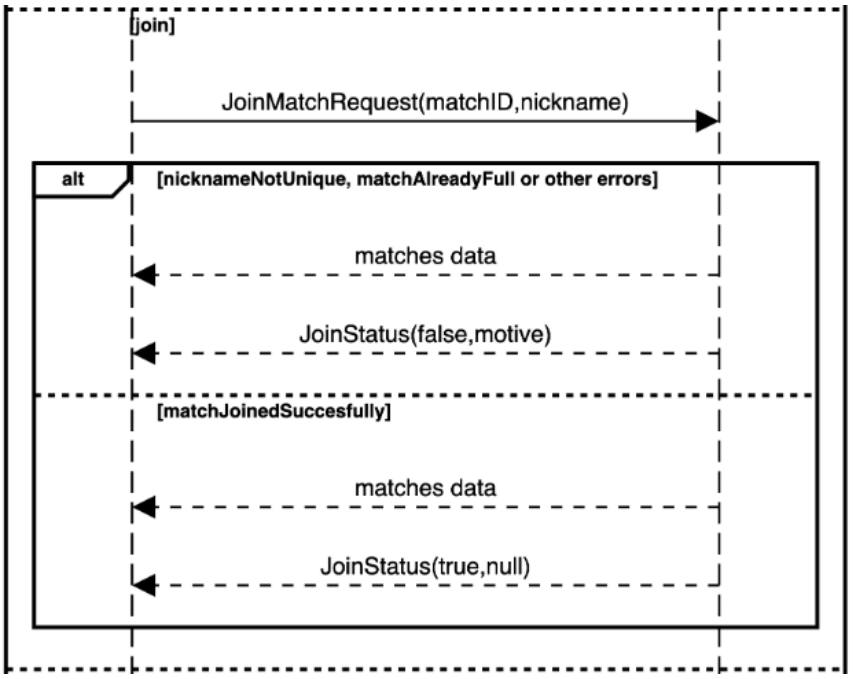


The motive in **MatchCreatedStatus(false,motive)** is an enum containing:

- numberOfPlayersNotValid
- nicknameNotUnique
- Other

3-Client requests to join a match

The client requests to join an existing match, then the server requests the desired nickname. If the provided nickname inside the message **JoinMatchRequest(matchID, nickName)** has been already chosen by another player or is invalid syntactically, the server asks for another nickname, waiting for another **JoinMatchRequest(matchID, nickName)** message. When the request is successful the match starts.

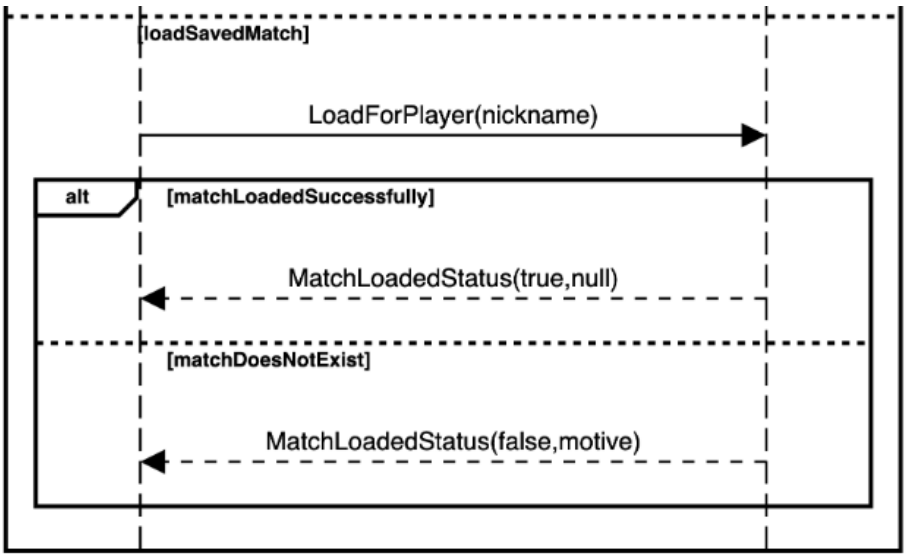


The motive in **JoinStatus(false,motive)** is an enum containing:

- numberOfPlayersNotValid
- matchAlreadyFull
- Other

4-Client requests to load a match

- The client requests to load a match, then the server requests a registered nickname among ones in saved matches, either active or inactive. If the provided nickname inside the **LoadForPlayer(nickname)** message sent to the server doesn't belong to an offline player, the server asks for another nickname, waiting for another **LoadForPlayer(nickname).** When the request is successful the client receives match data and the loaded match starts. If the loaded match is a multiplayer one, the other players are notified.

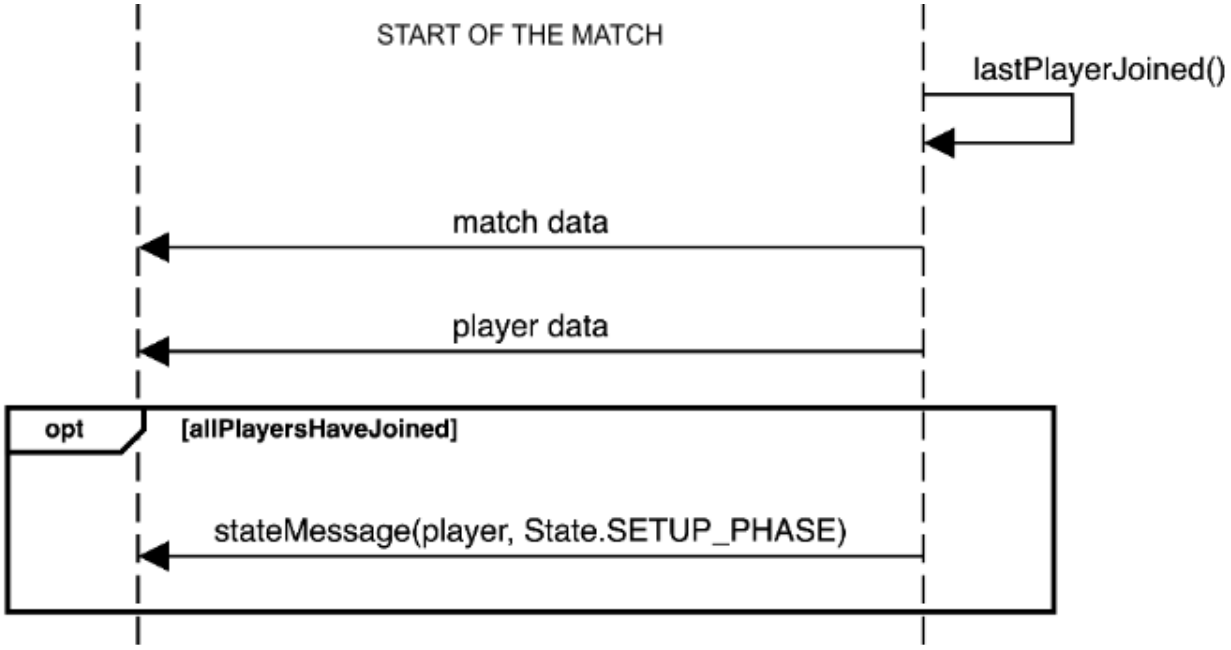


The motive in **LoadForPlayer(nickname)** is an enum containing:

- noExistingMatchesForPlayer
- playerIsAlreadyPlaying
- Other

5-Start of the match

As described in detail in each possible scenario, after a request is successfully accepted by the server, in case of multiplayer match the lobby is destroyed after all players have joined the match, otherwise the single player joins directly the match. Then the server sends the **match data** and the **player data.** Finally the client starts the match.

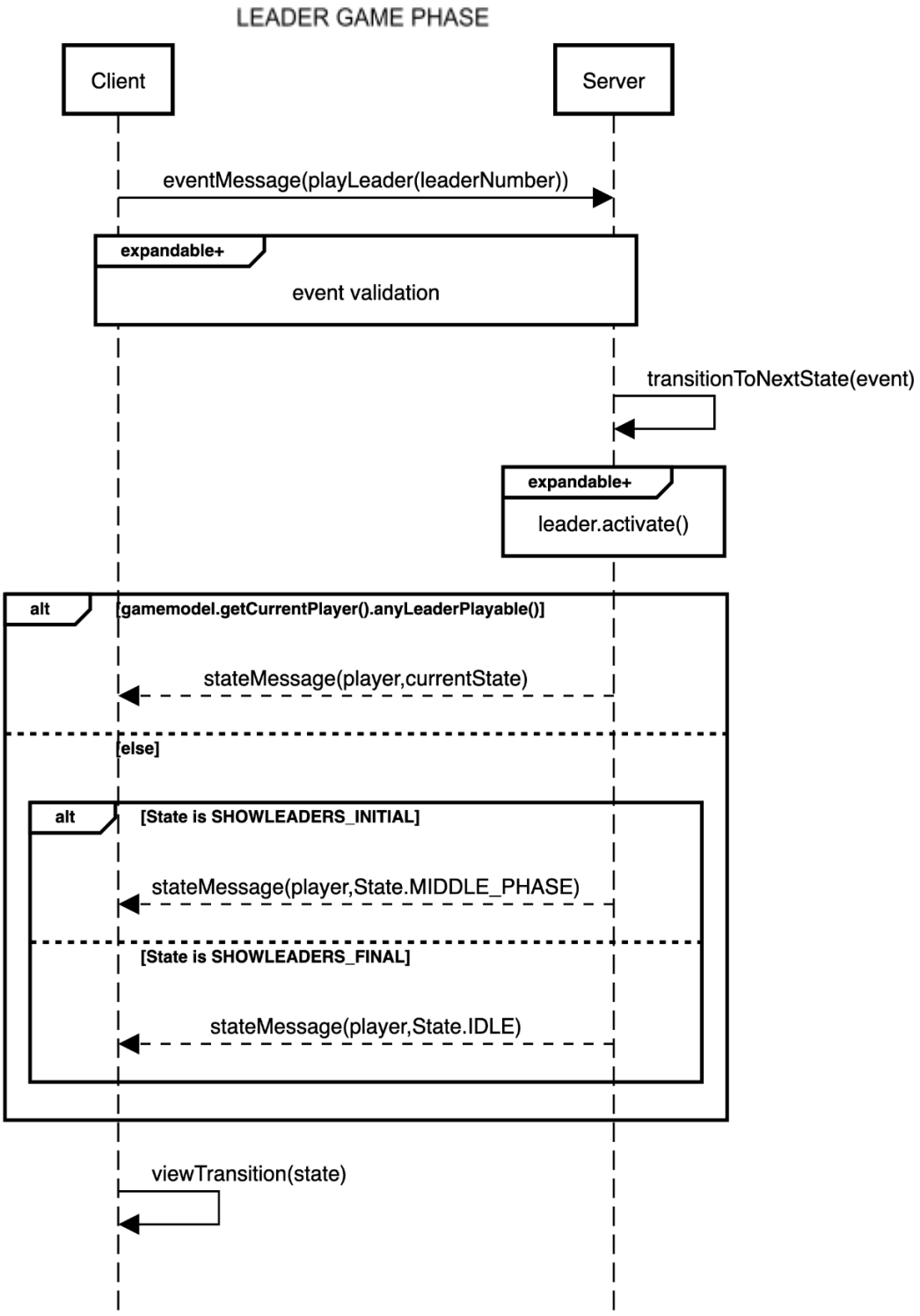


LEADER GAME PHASE

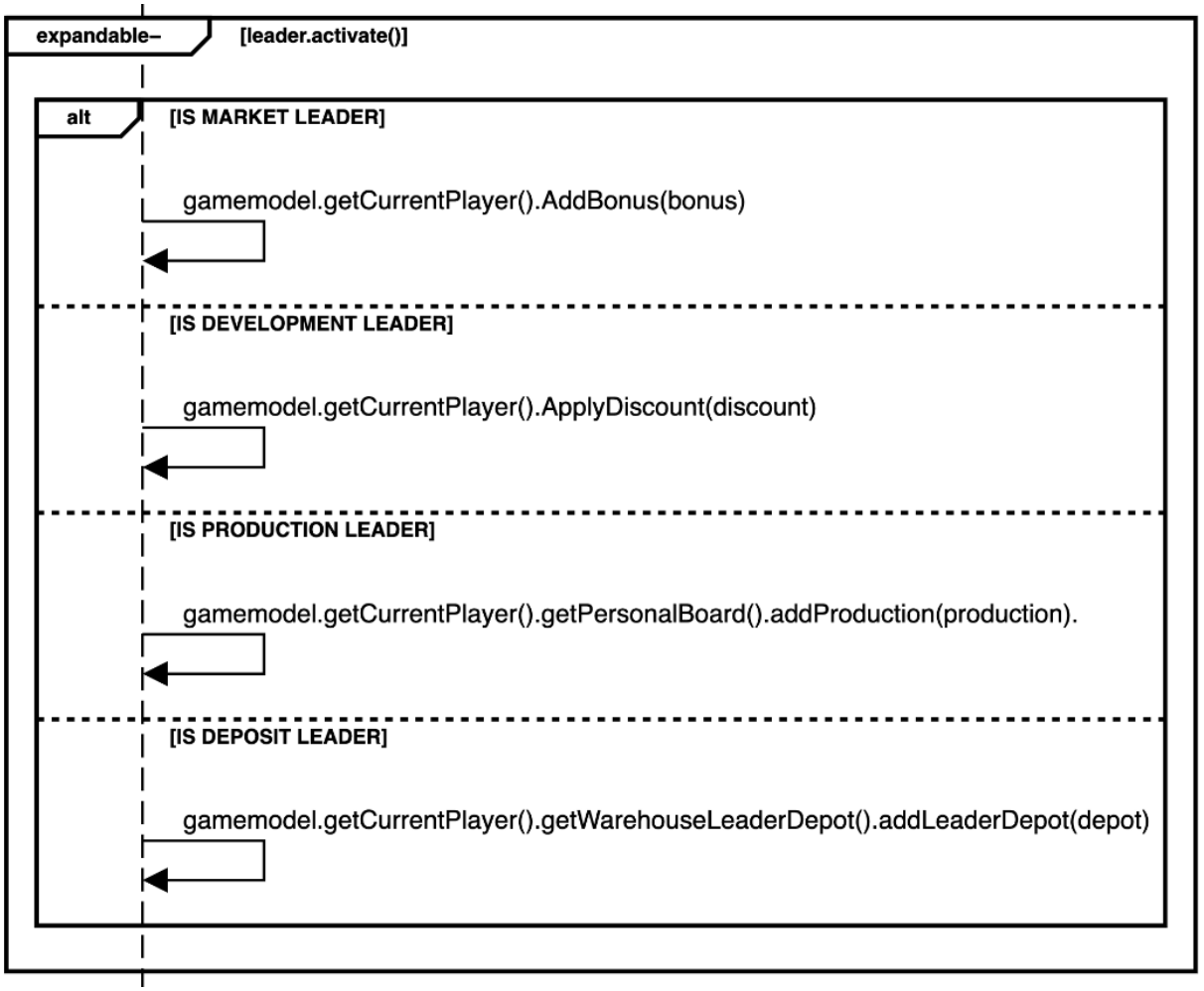
The player can select a leader during the **initial phase** or the **final phase.**
The client selects what leader to play and sends it to the server in **eventMessage(playLeader(leaderNumber)**.
The **Player** class contains two instances of the abstract class **Leader**, which has four concrete extensions, all overriding the activate() method:

- **MarketLeader** and **DevelopmentLeader** set certain values on the **Player** class
- **DepositLeader** and **ProductionLeader** add their items to the **PersonalBoard**.

The method **activate()** is executed after selecting a leader from the player class, it changes the player's PersonalBoard and their state accordingly, in the server and in the client via **stateMessage(player, state)**.
The **middle phase** is the phase where the player can choose to produce, get resources from the market, or buy a development card.

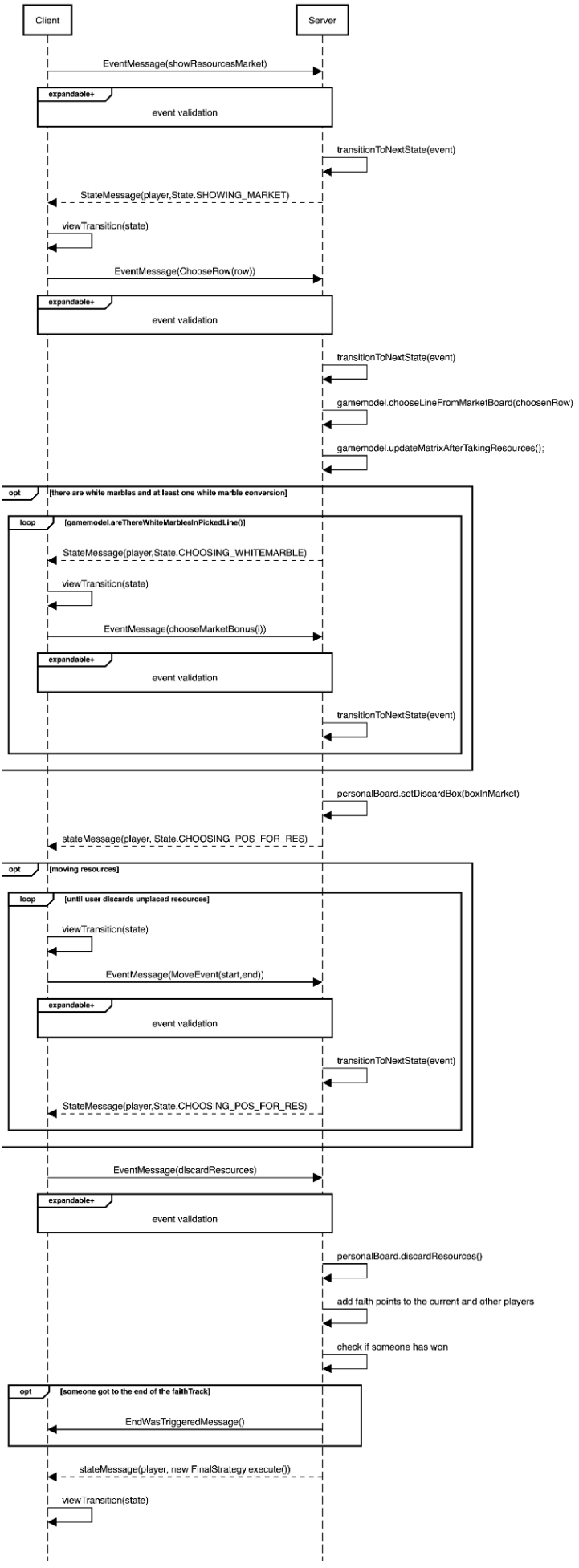


Leaders activation in server

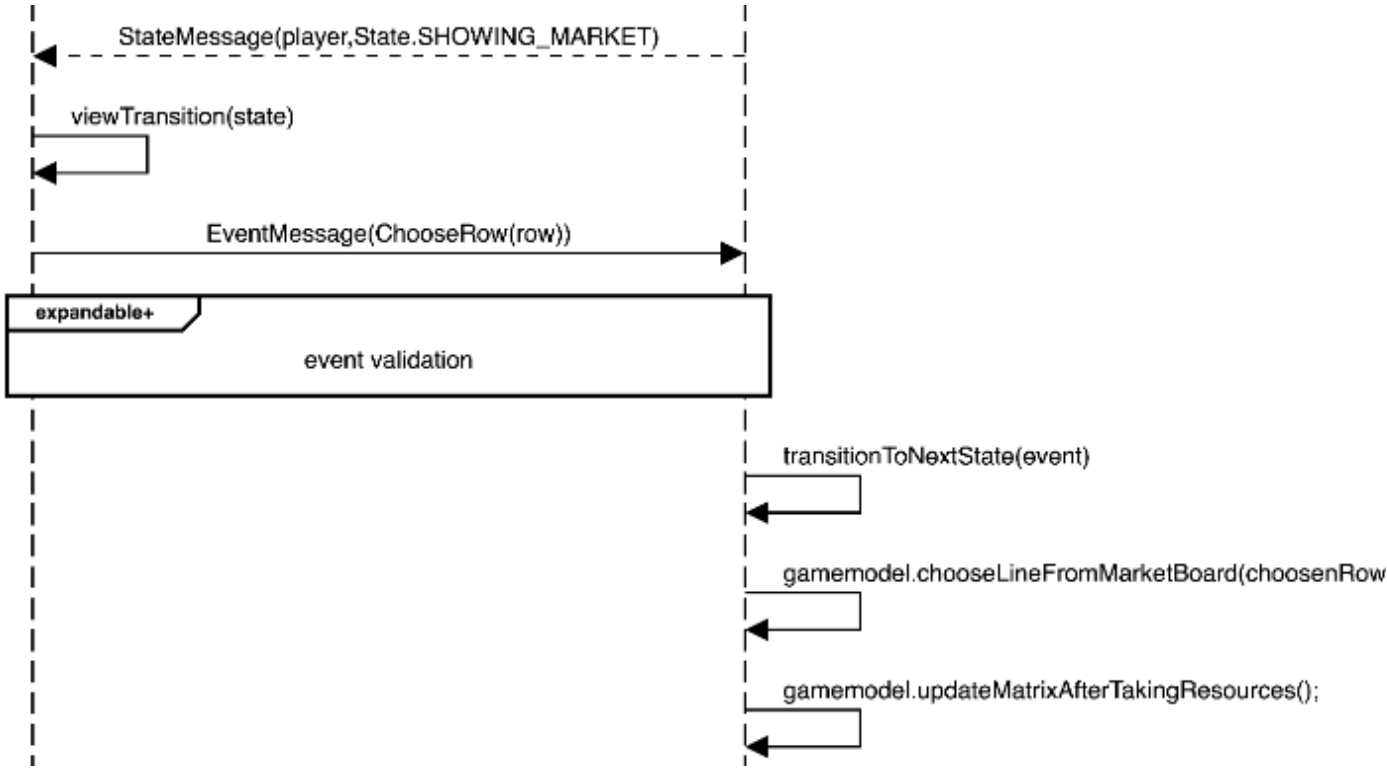


GET RESOURCES FROM MARKET

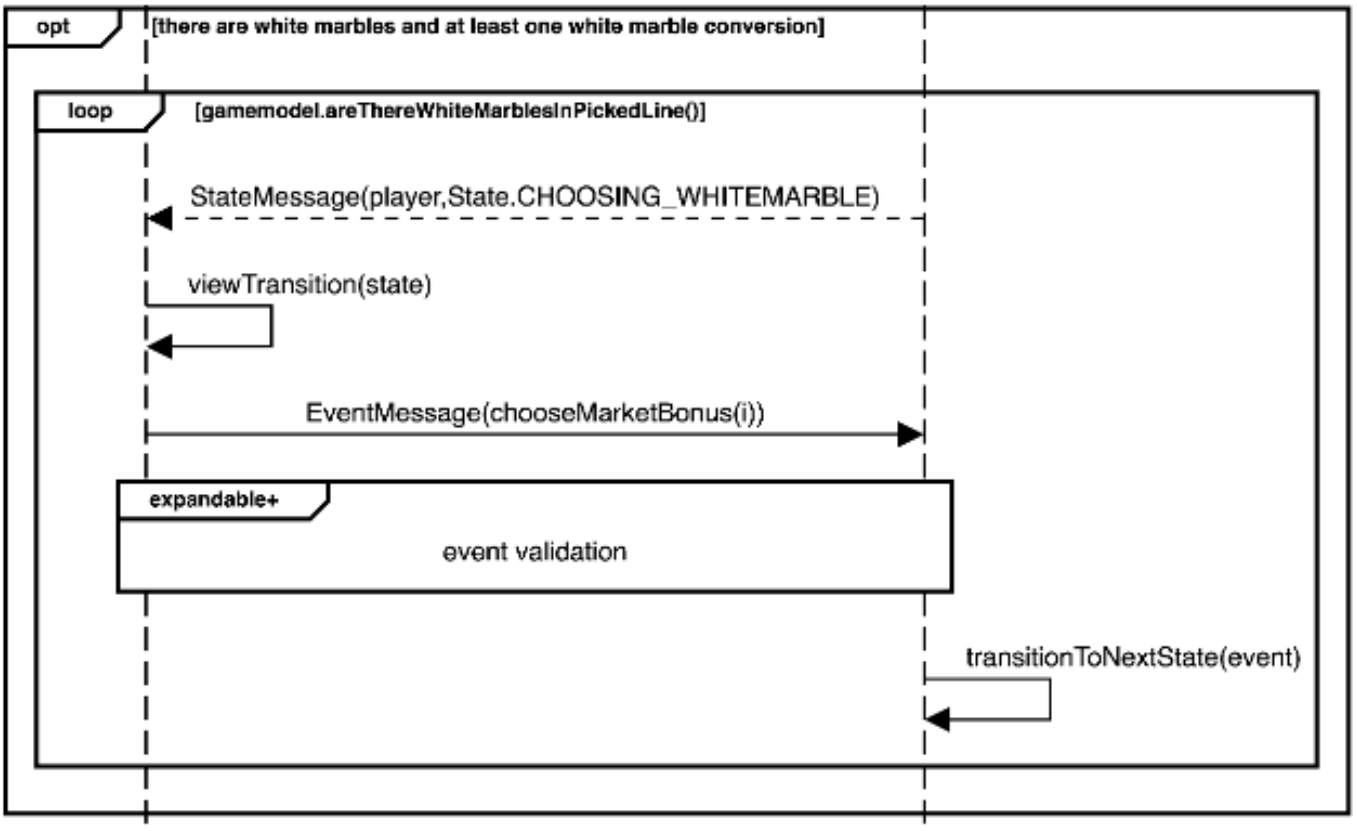
The Market is composed of a matrix of twelve marbles. After each row or column selection, it gets shuffled to ensure random placement of the marbles.



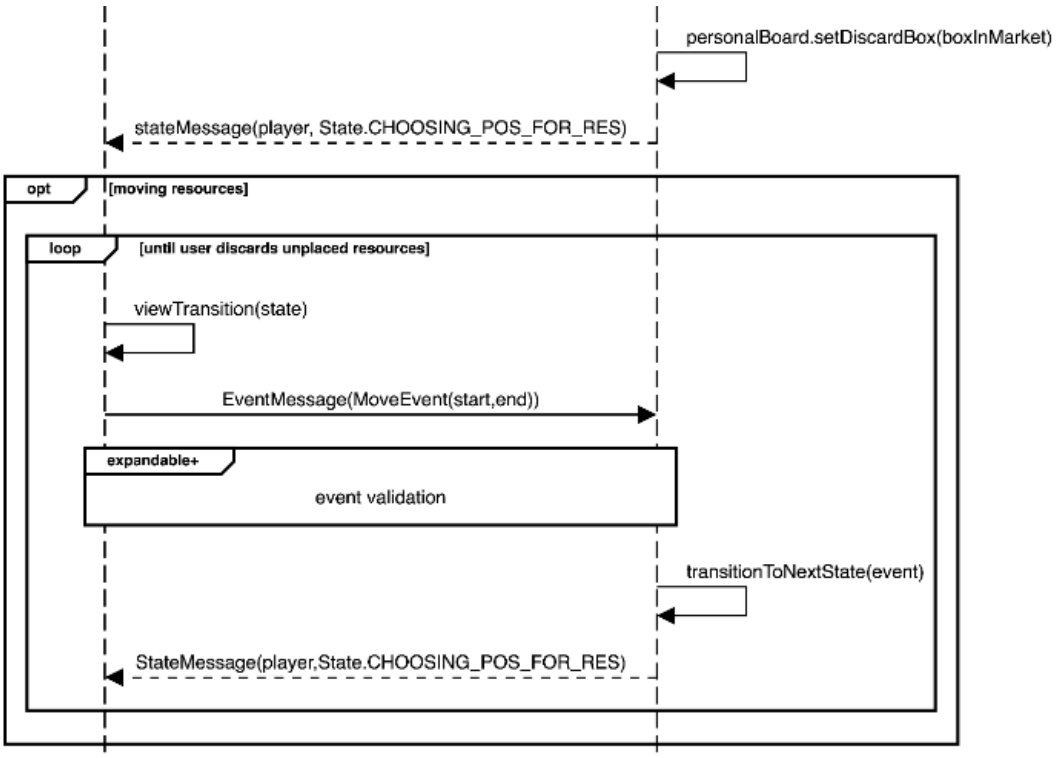
In the state **SHOWING_MARKET** the user has to choose one of the market matrix's rows or columns, with seven total possible choices (their values range from 0 to 6). The choice is memorized in the enum row. The user will receive a **discardBox**, containing the resources in the selected line.



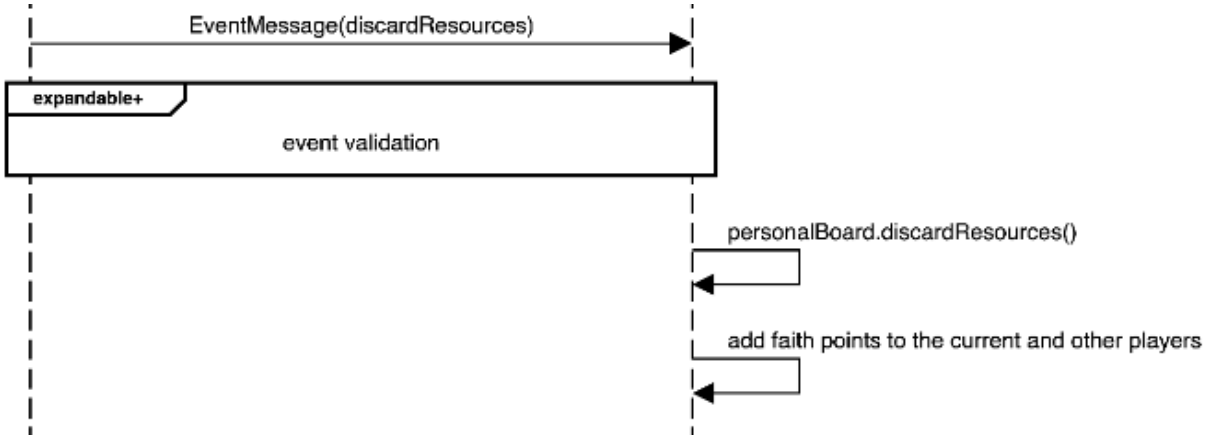
If there is more than one conversion available for white marbles the client will get the message **StateMessage(player, State.CHOOSING_WHITEMARBLE)** and the state message will be **CHOOSING_WHITEMARBLE**.
In that state, the player will be required to choose the resource for the white marble/marbles conversion from an enum, which can assume values between zero and four, each of them representing a resource. The chosen conversion will be sent to the server by **EventMessage(chooseMarketBonus(i))** where i is the chosen conversion.



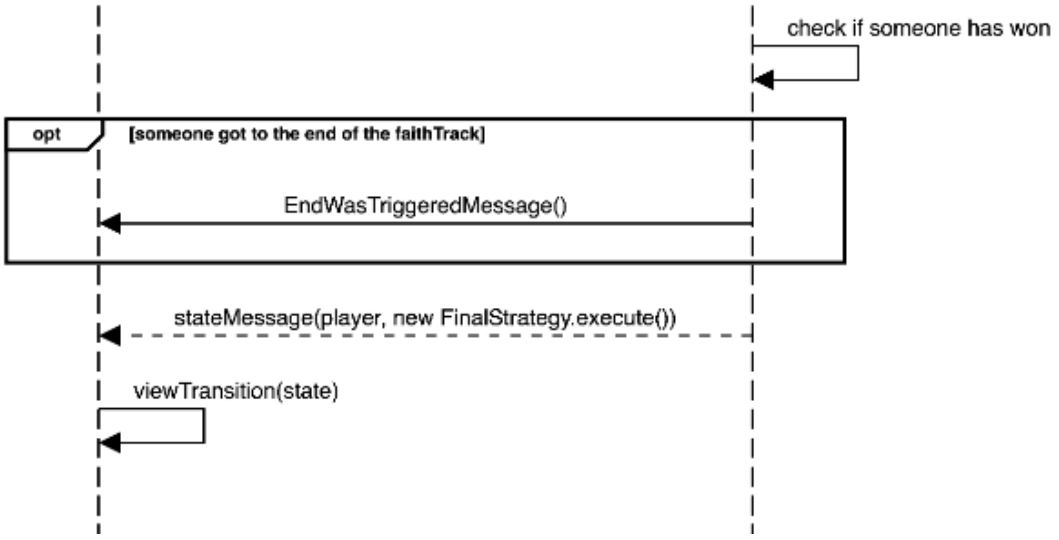
In the state **CHOOSING_POSITION_RES** the player will receive the **discardBox** containing the resources obtained from the market. Player will be able to freely place resources according to the rules, moving them from the **discardBox** to the Warehouse or vice-versa by sending the message **EventMessage(MoveEvent(start, end))**,where start and end are integers, representing a global resource position. Global meaning it can point to a location in the **discardBox**, the warehouse, or the leader depots.



After the server receives a **EventMessage(discardResources)** the remaining resources in the **discardBox** will be discarded and the faith point added to the other players. In case of one or more *Faith markers* in the selected line,
faith points are added to the current player too.

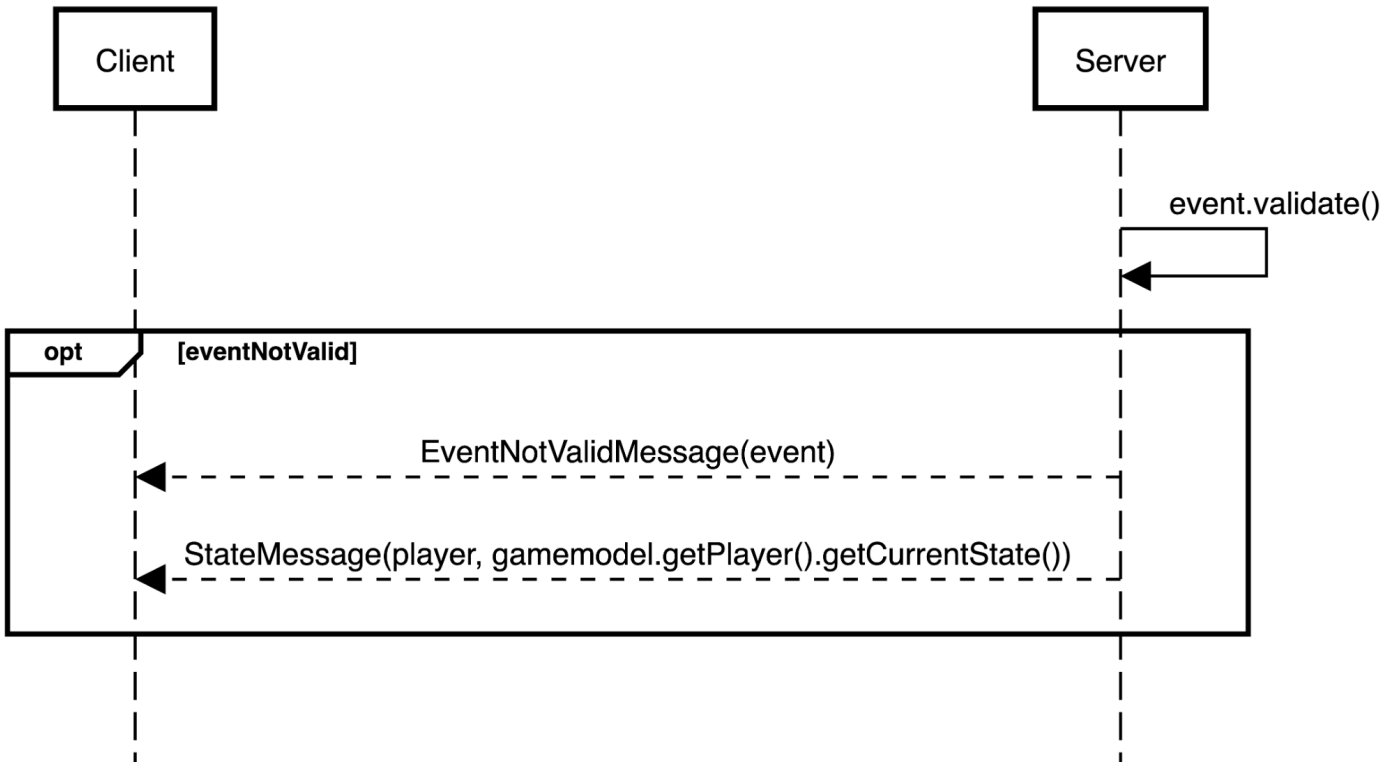


If a player has reached the end of the faithThack all the other players are notified by the **EndWasTriggeredMessage().**



EVENT NOT VALID

We validate the event in the controller and if it's not valid (for example a modified client moves a resource in a non-valid position) the event is ignored in the server and we send **two messages** to the server: the first one with the non-valid event for the client to know what went wrong, the second one with the previous state of the player.



The **state message** is composed by:

- The player of which to change the state, an int.
- The new state of the player, an enum that contains a serialized String with the objects that the view will see in the state. It will be memorized in a cache in the client to reduce network data.