Eriantys Protocol Documentation

Giovanni Manfredi, Mattia Martelli, Sebastiano Meneghin
Group 27

Introduction

The following document will describe the Client-Server communication protocol of the program Eriantys. We used GSON library for serialization and deserialization of messages.

Our group decided to opt for the use of multi threading to manage the variety of different actions the server is required to do. The server is composed of four main threads: *ConnectionHandler, Game, Info and CharacterCard*. This has determined how we explained the protocol in this document: 5 different sequence diagram are present, 4 of which represent separate threads.

Our group decided to focus on the following **advanced features**:

- Character cards implementation
- Four players game
- Persistence

The advanced features that more influenced our communication protocol are the character cards management and saving the game if the connection is lost (persistence).

1 Messages

Introduction

Our group decided to use *parametric messages* for great part of our communication protocol due to the similarities across different messages (in particular the ones regarding in-game communication).
We used five main parametric messages in our protocol, which are: *RequestAction, ResponseAction, RequestValue, ResponseValue, sendInfo*.
NB: these parametric messages are used in the whole program and not only in the section in which they are presented. Tables will be used to simplify the explanation.

1.1 Connection

Connection

MessageName	Description	Sender	Arguments
registerNewPlayer	Sends information required to connect to the server and to be identified as a player.	Client	*String* username: username of the player. *Int* magicAge: for how may years has the player known magic.
requestResumeGame	Asks the player if he wants to resume an interrupted game.	Server	
responseResumeGame	Responds to requestResumeGame received.	Client	*Boolean* resume: true if the player wants to resume a game.
signalWaitConnectionStatus	Notifies player that he will wait the connection of other players.	Server	
signalGameStart	Notifies player the start of the game	Server.	

Continues in the next page

Continues from the previous page

MessageName	**Description**	**Sender**	**Arguments**
requestRules	Requests player a new set of rule he desires to play with.	Server	
responseRules	Responds to responseRules with a new set of rules.	Client	*Int* numOfPlayers: number of players that will participate in a game. *Boolean* expertRule: true if the player wants to play a game in expertMode.
requestWizard	Asks the player which Wizard Deck he wants to play with.	Server	*Wizard[]* availableWizards: a wizard deck with the cards that haven't been selected by other players.
responseWizard	Responds to requestWizard.	Client	*Wizard* selectedWizard: wizard selected by the player from available wizards.
signalPlayer Disconnected	Notifies to the players that they will be disconnected since one player left the game and disconnects them.	Server	
signalGameInProgress	Notifies to the player that a game is already ongoing and disconnects him.	Server	

1.2 Game

Game

MessageName	Description	Sender	Arguments
requestValue	Asks for the value of a generic field. **Parametric message**.	Client	*String* fieldName: name of the field is requested the value of.
responseValue	Responds to requestValue with the value of the field requested. **Parametric message**.	Server	*<type> fieldValue:* value of the field. i.e. type = int[], Color[], ...
sendInfo	Sends all the information about the GameBoard (after the setUpPhase) or its changes any time its modified. **Parametric message**.	Server	*Int* numOfFields: number of the field/fieldArray included in the message. *<type>* fieldArray[]: various fields are needed to update local player data, i.e. professorsLocation, emptiedCloudTile
signalWaitTurnStatus	Notifies player that he has to wait another player turn.	Server	
signalGameTurn Status	Notifies player that has started his turn.	Server	

Continues in the next page

Continues from the previous page

MessageName	**Description**	**Sender**	**Arguments**
requestAction	Requests player to make an action (i.e choose an assistantCard, move students) between provided options. **Parametric message**.	Server	*<type>* fieldArray[]: options for a specific field to choose from, i.e. studentsArray, fillCloudTileArray
responseAction	Responds to requestAction with the option selected by the player. **Parametric message**.	Client	*<type>* field: value selected by the player

1.3 EndGame

EndGame

MessageName	Description	Sender	Arguments
signalGameResult	Updates player about the end of the match and the winner(s) (including draws).	Server	*GameResult<enum>* gameResult: value indicating the result of the game, depending on how game has finished.
requestReMatch	Asks to the player if he wants to play another game with same players and same rules.	Server	
responseReMatch	Responds to requestReMatch.	Client	*Boolean* rematch: true if the player wants to play another game.
signalEndGame	Updates player about other players choice about a re-match and disconnect him.	Server	

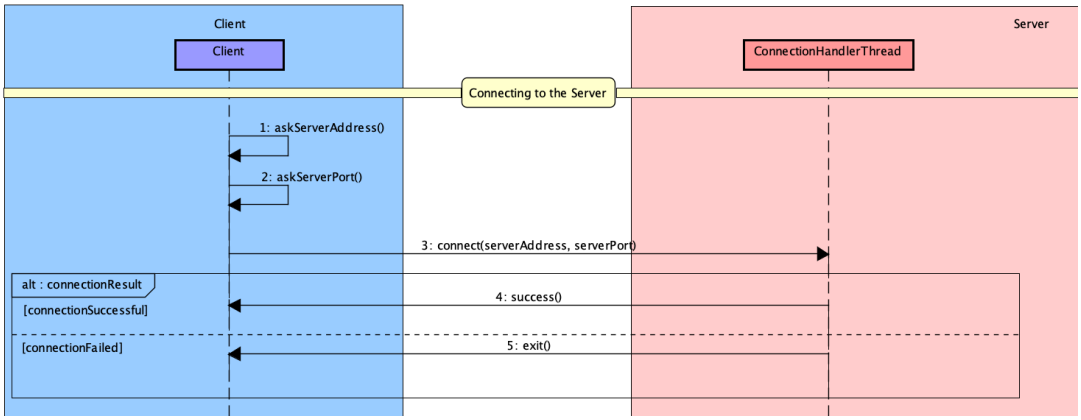
2 Scenarios

Introduction

As previously mentioned we divided the program into five sections. For each one of them we created a sequence diagram showing the different messages of the protocol we just explained above. In this documentation we will analyze the different sections in details, so it has been necessary to crop the original sequence diagrams. The unaltered versions are presented at the end of their respective sections.

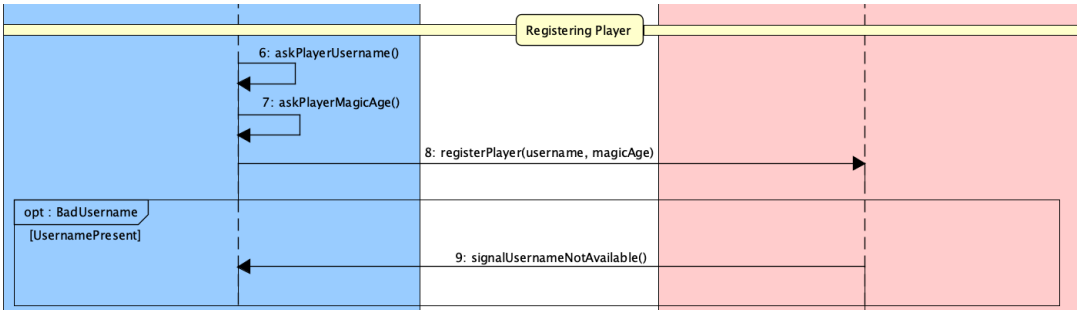
2.1 Connection

2.1.1 Connecting to the server



First, the client is asked internally for the ServerAddress and the ServerPort he wants to connect to. Then the client sends this information in the message *connect(serverAddress, serverPort)*. If the server responds with *success()*, the connection is ready to receive the registration related messages.
NB: The server handles the connection with a dedicated thread: **ConnectionHandler**.

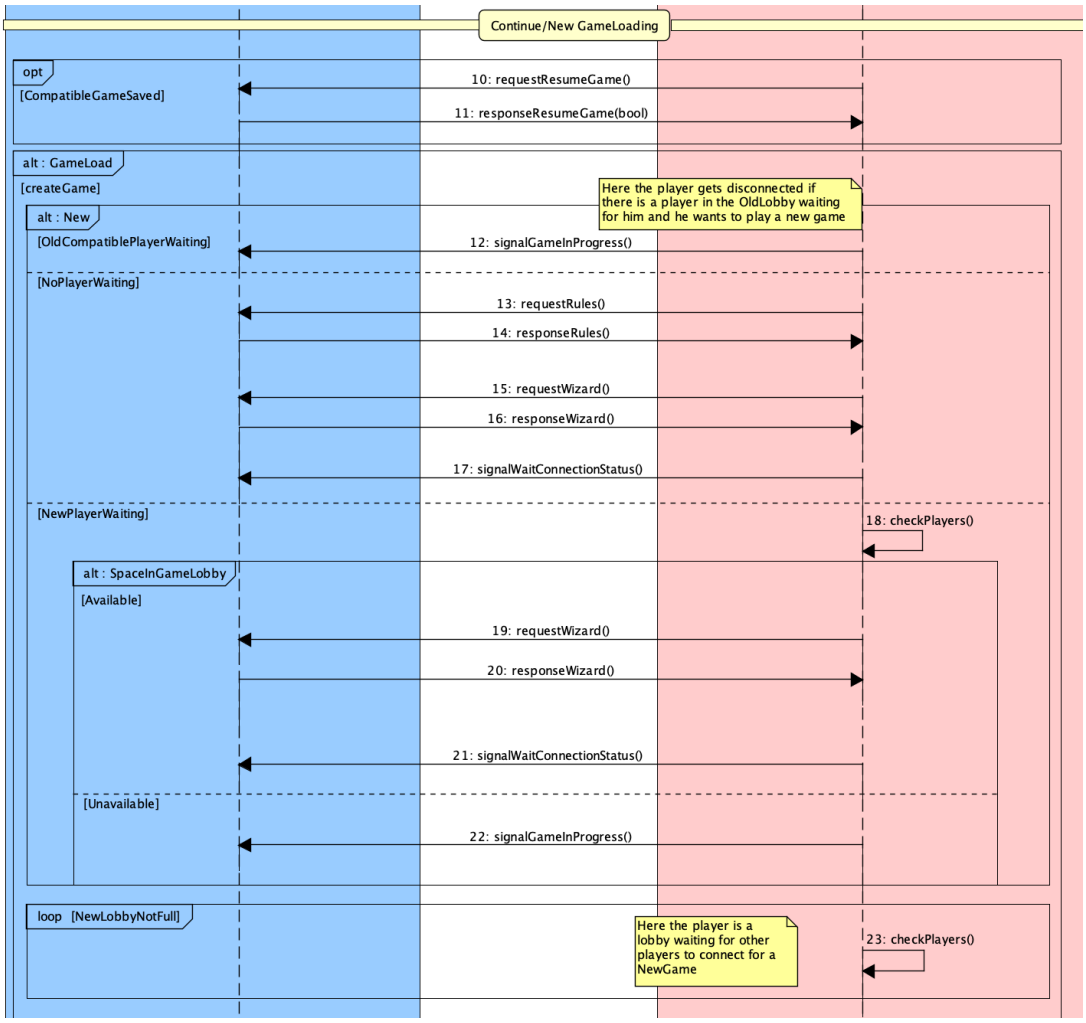
2.1.2 Registering a player



The client is asked internally for his username and magicAge (for how many years has he known magic). Then the client sends this information in the message

registerPlayer(username, magicAge). If the server responds with *signalUsernameNotAvailable*, another player with the same username is already connected to the server so the client is disconnected with a message explaining the issue.

2.1.3 Creating a new game



The server sends the client a *requestResumeGame()* if a saved game is present in the server with a username equal to the client's. The client, if it has received this message, responds with *responseResumeGame(bool)* indicating the choice to resume his game ($1 \rightarrow$ true, $0 \rightarrow$ false).
If the client didn't receive *requestResumeGame()* or chose to not resume his old game the alternative taken by the program is [createGame].
Then based on the other players already connected to the server there could be three different scenarios:

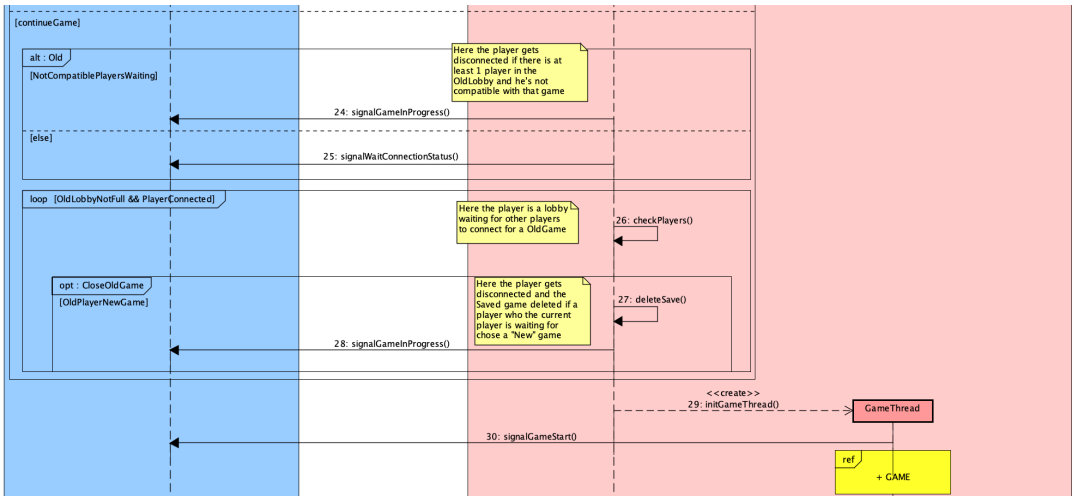
- **OldCompatiblePlayerWaiting**: one or more players compatible with a saved game are currently waiting in the OldLobby. The client gets disconnected when he receives *signalGameInProgress()* explaining the issue.

- **NoPlayerWaiting**: no player is currently waiting for a game, so the server sends *requestRules()* and *requestWizard()* which ask the rules for the new game and which wizard (nature, desert, cloud or snow) the client wants to play with respectively. The client responds with *responseRules()* and *responseWizard()* indicating which rules (int numberOfPlayers, bool expertMode) and which wizard (enum chosenWizard) he wants to play with. **NB**: The first player who connects to the newLobby is the one who decides the rules of the game, the other players connect to his lobby if space is still available.

- **NewPlayerWaiting**: a player is currently waiting for a new game (in the NewLobby) so we check if there is still space available in the lobby. If there is still space the servers asks the player which wizard he wants to play with through the request-response: *requestWizard()*, *responseWizard()*. Then the server informs the player using the message *signalWaitConnectionStatus()* that he is in the lobby waiting for new players to connect and start the game.

Finally the server waits for new players to connect checking in a loop if the NewLobby is full. Once the lobby is full the server is ready to create a new game.

2.1.4 Resuming an old game



If the player received *requestResumeGame()* and chose to resume his old game the alternative taken by the program is [resumeGame].
Then based on the other players already connected to the server there could be two different scenarios:

- **NotCompatiblePlayersWaiting**: one or more players are waiting for a game not compatible with the client (OldNotCompatible game or NewGame). The client gets disconnected when he receives *signalGameInProgress()* explaining the issue.

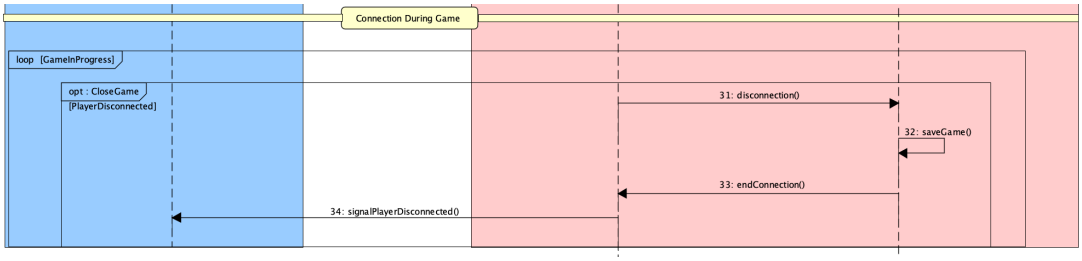
- **CompatiblePlayersWaiting or NoPlayerWaiting**: alternatively the player receives *signalWaitConnectionStatus()* informing him that he is connected to the OldLobby waiting for the oldPlayer to join.

Finally the servers waits for the old compatible players to connect checking in a loop if the OldLobby is full. If a player the server is waiting for to start the game connects but decides to begin a new game, the server deletes the save and informs the other players waiting for him of the issue and disconnects them with *signalGameInProgress()*.

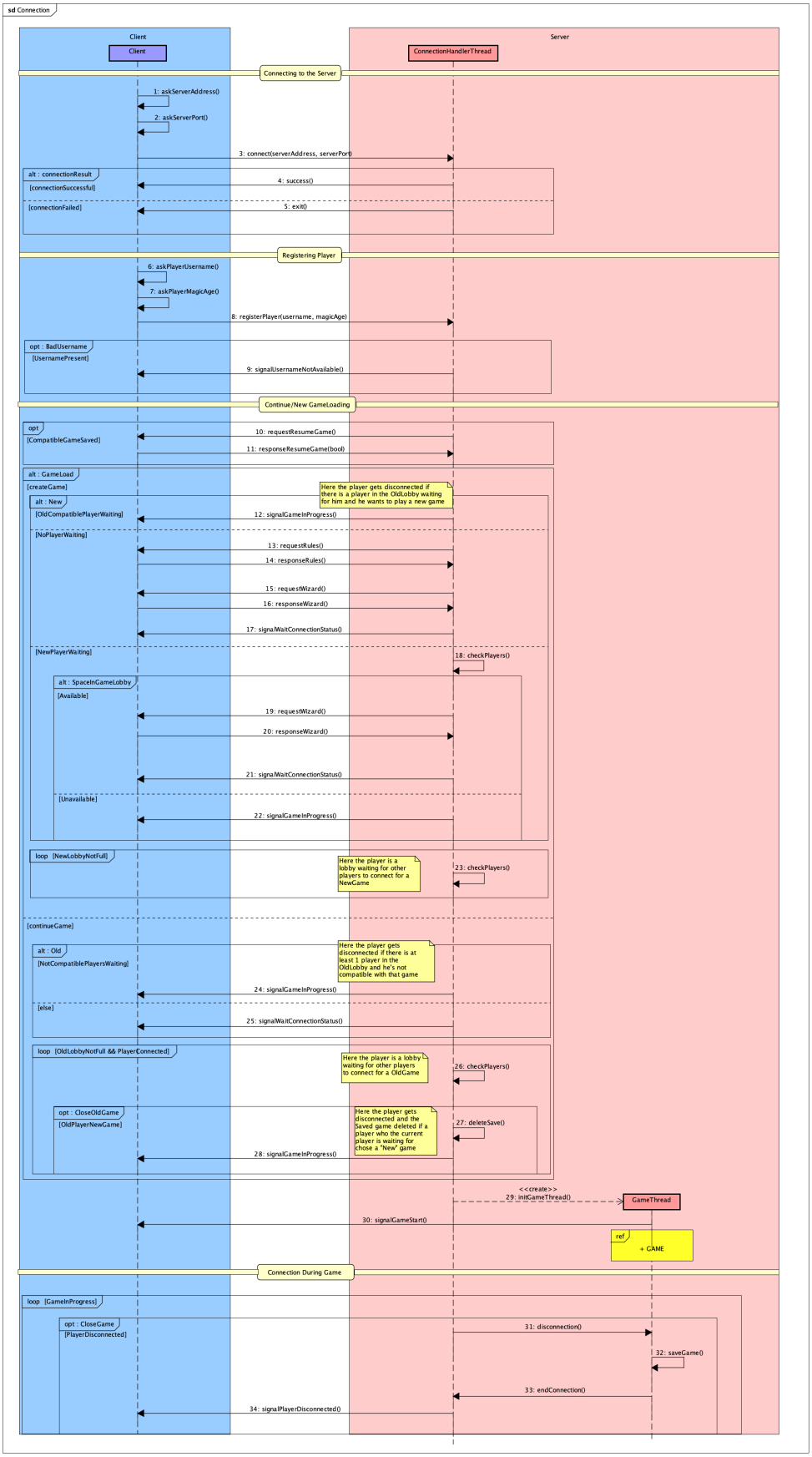
Out of the alternative block called GameLoad, the program now has a full lobby so the ConnectionHandler initializes GameThread that when created informs the client with the message *signalGameStart()*.

NB: a new game will begin with a SetupPhase and will be followed by the consecutive phases, whereas a resumed game will begin with the saved state and continue the game from there.

2.1.5 Disconnecting during a game

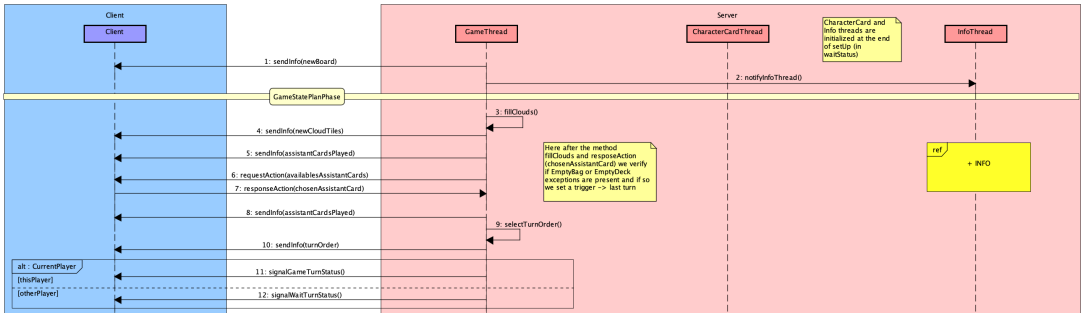


During the game the **ConnectionHandlerThread** handles the disconnections by the client. The server loops an optional condition of disconnection that if reached informs the GameThread, which saves the game and asks the ConnectionHandler to close the connection. The ConnectionHandler sends *signalPlayerDisconnection()* to inform the client of the issue and to disconnect him.


sd Connection
Client
Client
Server
ConnectionHandlerThread
Connecting to the Server
1: askServerAddress()
2: askServerPort()
3: connect(serverAddress, serverPort)
alt : connectionResult
[connectionSuccessful]
4: success()
[connectionFailed]
5: exit()
Registering Player
6: askPlayerUsername()
7: askPlayerMagicAge()
8: registerPlayer(username, magicAge)
opt : BadUsername
[UsernamePresent]
9: signalUsernameNotAvailable()
Continue/New GameLoading
opt
[CompatibleGameSaved]
10: requestResumeGame()
11: responseResumeGame(bool)
alt : GameLoad
[createGame]
Here the player gets disconnected if there is a player in the OldLobby waiting for him and he wants to play a new game
alt : New
[OldCompatiblePlayerWaiting]
12: signalGameInProgress()
[NoPlayerWaiting]
13: requestRules()
14: responseRules()
15: requestWizard()
16: responseWizard()
17: signalWaitConnectionStatus()
[NewPlayerWaiting]
18: checkPlayers()
alt : SpaceInGameLobby
[Available]
19: requestWizard()
20: responseWizard()
21: signalWaitConnectionStatus()
[Unavailable]
22: signalGameInProgress()
loop [NewLobbyNotFull]
Here the player is a lobby waiting for other players to connect for a NewGame
23: checkPlayers()
[continueGame]
alt : Old
[NotCompatiblePlayersWaiting]
Here the player gets disconnected if there is at least 1 player in the OldLobby and he's not compatible with that game
24: signalGameInProgress()
[else]
25: signalWaitConnectionStatus()
loop [OldLobbyNotFull && PlayerConnected]
Here the player is a lobby waiting for other players to connect for a OldGame
26: checkPlayers()
opt : CloseOldGame
[OldPlayerNewGame]
Here the player gets disconnected and the Saved game deleted if a player who the current player is waiting for chose a "New" game
27: deleteSave()
28: signalGameInProgress()
<<create>>
29: initGameThread()
GameThread
30: signalGameStart()
ref
+ GAME
Connection During Game
loop [GameInProgress]
opt : CloseGame
[PlayerDisconnected]
31: disconnection()
32: saveGame()
33: endConnection()
34: signalPlayerDisconnected()

2.2 Game

2.2.1 SetupPhase and PlanPhase



The first phase of the game is the Setup, which is not shown in these diagrams because it is mainly composed of methods internal to the server.
At the end of the SetupPhase the server sends the information regarding the game board to the client using a *sendInfo(newBoard)*. GameThread also notifies InfoThread waking him up. After this message InfoThread will be active simultaneously with GameThread, answering requests from the Client for values of the Model.

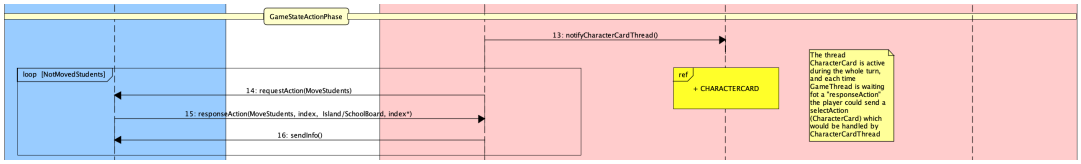
The server begins the PlanningPhase which is composed of three sections:

- **FillClouds**: the server fills the CloudTiles with students extracting them from the Bag. If the Bag is empty, it triggers the *EmptyBagException* that will set a trigger which is then checked in the last phase: if the trigger is set this will be the last turn of the game.
  The server then informs the players with the message *sendInfo(newCloudTiles)*.

- **PlayAssistantCard**: the server informs the players with what AssistantCards have already been played with *sendInfo(assistantCardsPlayed)* and then asks him what AssistantCard he wants to play from an available pool with *requestAction(availableAssistantCards)*. The client responds with the card he wants to play using the message *responseAction(chosenAssistantCard)*. The server further replies by sending *sendInfo(assistantCardsPlayed)* informing the client and the other players on what AssistantCard was just played.

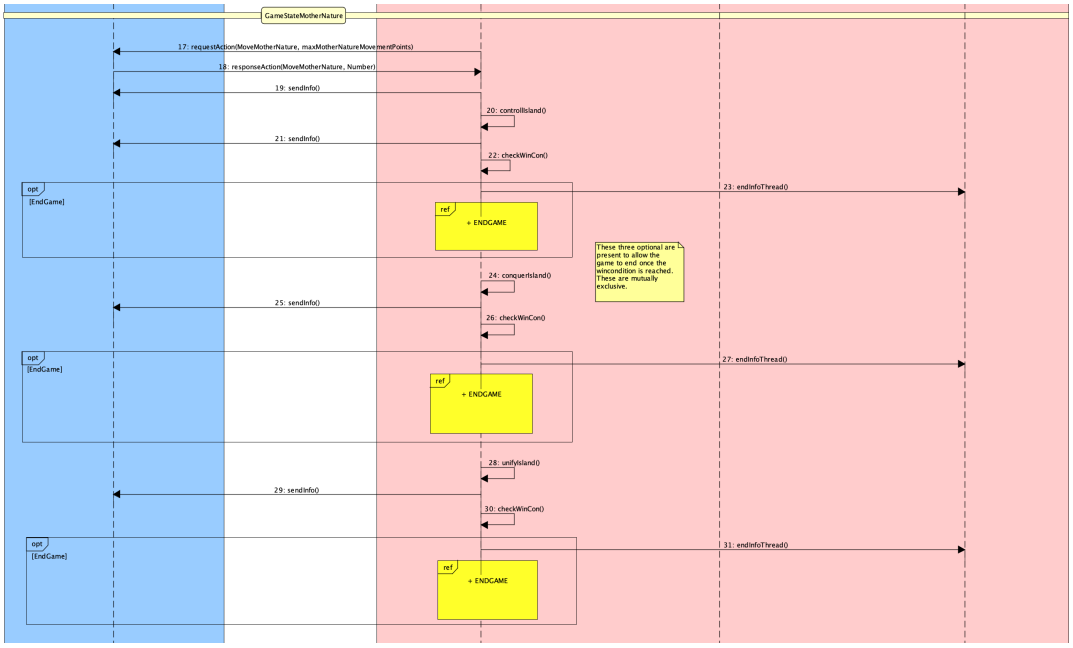
- **SelectTurnOrder**: the server selects a turn order based on the played cards and sends the client a *sendInfo(TurnOrder)* to inform him on the turn order. Then the client could be the firstPlayer or not, if he is , the server notifies him with a *signalGameTurnStatus()* indicating that he will now start his turn. Alternatively, the server informs him with *signalWaitTurnStatus()* that he will wait another player's turn.

2.2.2 ActionPhase - MoveStudents and CharacterCards



At the beginning of the ActionPhase the GameThread notifies CharacterCardThread waking him up: it will be now be possible for the client to send requests to use a CharacterCard.
The server then enters a loop (of three cycles) to allow the client to move his students. GameThread sends a *requestAction(MoveStudents)* to which the client responds with a *responseAction(MoveStudents, index, Island/SchoolBoard, index*)* specifying the student counter the client wants to move and where he wants to move it. After the movement the server updates the client with a *sendInfo()*.

2.2.3 ActionPhase - MoveMotherNature



First, the GameThread sends a *requestAction(MoveMotherNature, maxMotherNatureMovementPoints)* asking the player to move Mother Nature of a positive value smaller or equal to parameter. The client replies with a *responseAction(MoveMotherNature, number)* specifying where he wants to move Mother Nature. The server then moves it and then informs the players with a *sendInfo()*.
Subsequently GameThread applies the three effects of Mother Nature's movement which are:

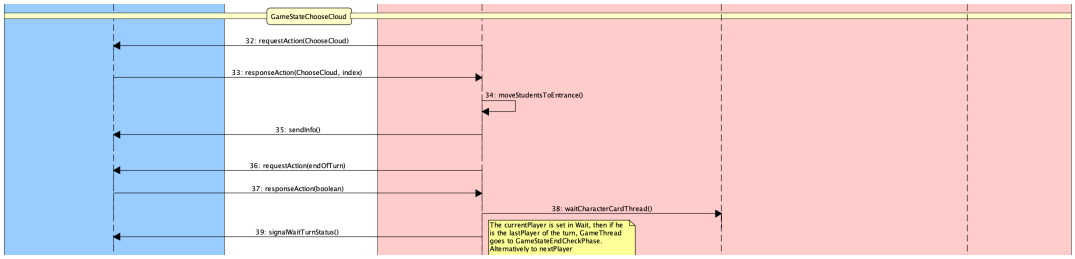
- **ControlIsland**: if no tower is present on the island (on which Mother Nature finished the turn on), the server calculates the influence on the

island and places the tower of the player with the highest influence on that island. Then, after informing the client with a *sendInfo()*, the GameThread controls if the winning condition is reached (all towers of a player played), and if so, it enters in a optional block called "EndGame".

- **ConquerIsland**: if one or more towers are present on the island the server calculates the influence on the island (on which Mother Nature finished the turn on) and replaces the tower with the one of the player with the highest influence on that island. Then, after informing the client with a *sendInfo()*, the GameThread controls if the winning condition is reached (all towers of a player played), and if so, it enters in a optional block called "EndGame".

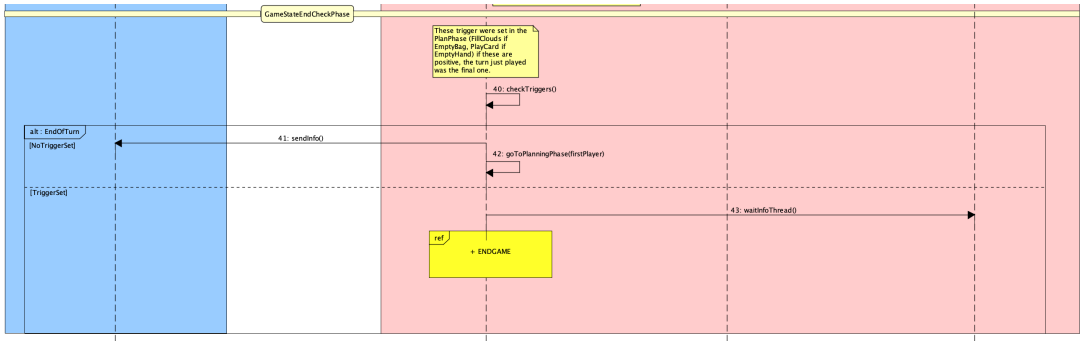
- **UnifyIsland**: if the adjacent islands (either or both) to the island (on which Mother Nature finished the turn on) have towers of the same TowerColor to the current island, the server unifies the islands (from three to one or from two to one). Then, after informing the client with a *sendInfo()*, the GameThread controls if the winning condition is reached (three or less islands on board), and if so, it enters in a optional block called "EndGame".

2.2.4 ActionPhase - ChooseCloud

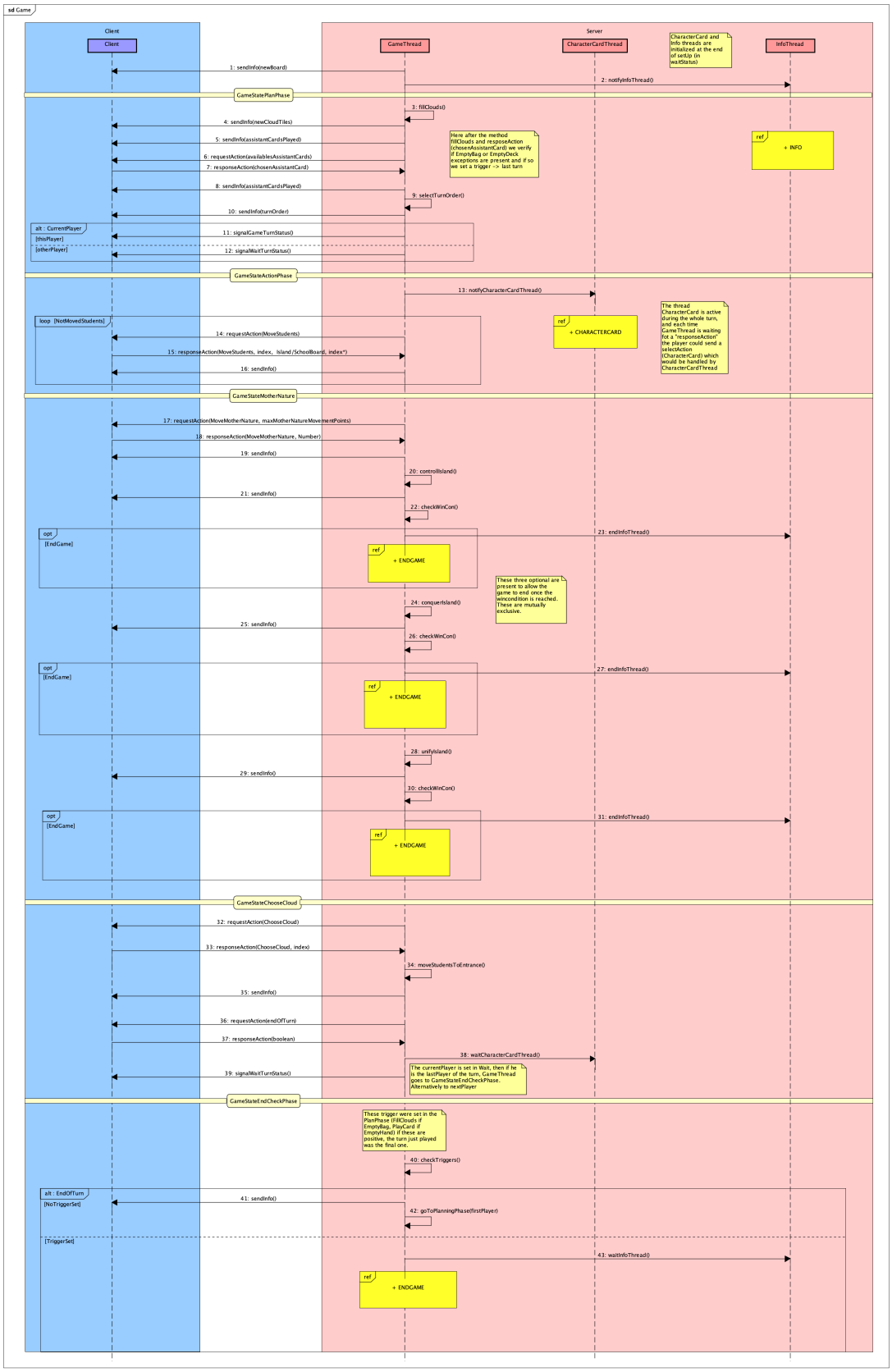


First the client is asked which CloudTile he is going to select with the message *requestAction(ChooseCloud)* to which the client responds with *responseAction(ChooseCloud, index)* specifying the CloudTile he chose. Then the GameThread after having moved the students from the chosen CloudTile to the Entrance of the player shows him the updated GameBoard with a *sendInfo()*.
Finally the server asks the client if he wants to end his turn using *requestAction(endOfTurn)* to which the client responds with *responseAction(boolean)* containing his decision (1 → true, 0 → false). The server waits until he receives a positive response (with value 1) and after receiving it, he ends the turn.
CharacterCardThread is informed and set to a waitStatus and the client receives a *signalWaitTurnStatus* which makes him wait for another player to play. After this Phase, GameThread goes to the next player, starting the turn from the ActionPhase, or if the current player was the last player GameThread continues in the EndCheckPhase.

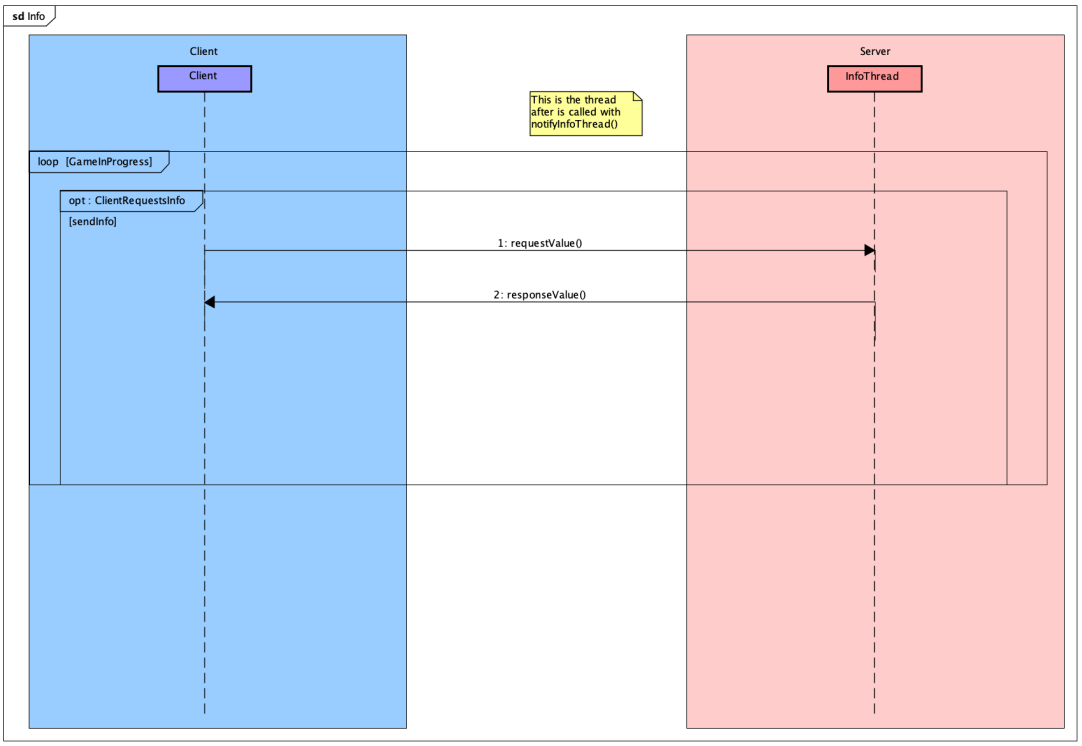
2.2.5 EndCheckPhase



First, the GameThread checks if any trigger (among EmptyBag or EmptyDeck) is set, if so the server enters the "EndGame", otherwise the server informs the players of the beginning of a new round with *sendInfo()* and the GameThread returns back to the PlanningPhase.

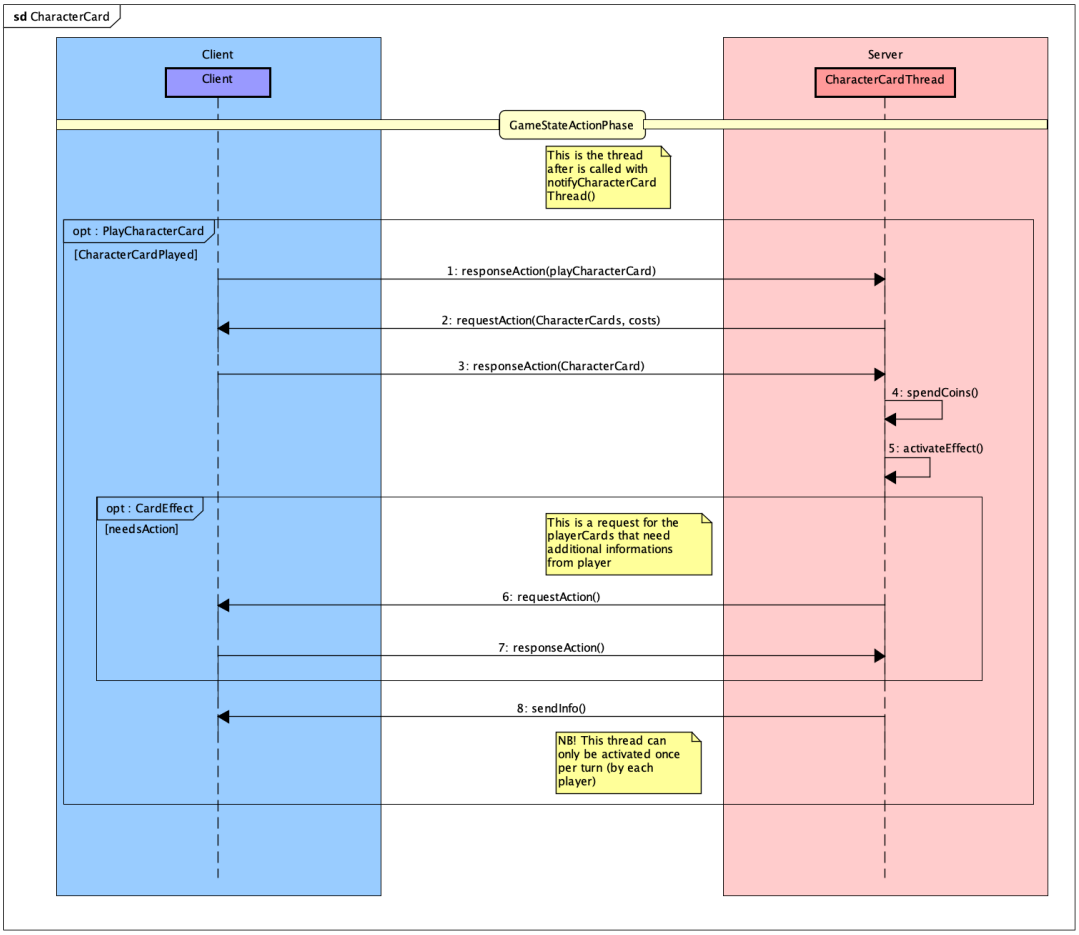

sd Game
Client
Client
Server
GameThread
CharacterCardThread
InfoThread
CharacterCard and Info threads are initialized at the end of setUp (in waitStatus)
1: sendInfo(newBoard)
2: notifyInfoThread()
GameStatePlanPhase
3: fillClouds()
4: sendInfo(newCloudTiles)
5: sendInfo(assistantCardsPlayed)
6: requestAction(availablesAssistantCards)
7: responseAction(chosenAssistantCard)
Here after the method fillClouds and responseAction (chosenAssistantCard) we verify if EmptyBag or EmptyDeck exceptions are present and if so we set a trigger -> last turn
ref
+ INFO
8: sendInfo(assistantCardsPlayed)
9: select(TurnOrder()
10: sendInfo(turnOrder)
alt : CurrentPlayer
[thisPlayer]
11: signalGameTurnStatus()
[otherPlayer]
12: signalWaitTurnStatus()
GameStateActionPhase
13: notifyCharacterCardThread()
loop [NotMovedStudents]
ref
+ CHARACTERCARD
The thread CharacterCard is active during the whole turn, and each time GameThread is waiting for a "responseAction" the player could send a selectAction (CharacterCard) which would be handled by CharacterCardThread
14: requestAction(MoveStudents)
15: responseAction(MoveStudents, index, Island/SchoolBoard, index*)
16: sendInfo()
GameStateMotherNature
17: requestAction(MoveMotherNature, maxMotherNatureMovementPoints)
18: responseAction(MoveMotherNature, Number)
19: sendInfo()
20: controlIsIsland()
21: sendInfo()
22: checkWinCon()
opt
[EndGame]
23: endInfoThread()
ref
+ ENDGAME
These three optional are present to allow the game to end once the wincondition is reached. These are mutually exclusive.
24: conquerIsland()
25: sendInfo()
26: checkWinCon()
opt
[EndGame]
27: endInfoThread()
ref
+ ENDGAME
28: unifyIsland()
29: sendInfo()
30: checkWinCon()
opt
[EndGame]
31: endInfoThread()
ref
+ ENDGAME
GameStateChooseCloud
32: requestAction(ChooseCloud)
33: responseAction(ChooseCloud, index)
34: moveStudentsToEntrance()
35: sendInfo()
36: requestAction(endOfTurn)
37: responseAction(boolean)
38: waitCharacterCardThread()
39: signalWaitTurnStatus()
The currentPlayer is set in Wait, then if he is the lastPlayer of the turn, GameThread goes to GameStateEndCheckPhase. Alternatively to nextPlayer
GameStateEndCheckPhase
These trigger were set in the PlanPhase (fillClouds if EmptyBag, PlayCard if EmptyHand) if these are positive, the turn just played was the final one.
40: checkTriggers()
alt : EndOfTurn
[NoTriggerSet]
41: sendInfo()
42: goToPlanningPhase(firstPlayer)
[TriggerSet]
43: waitInfoThread()
ref
+ ENDGAME

2.3 Info



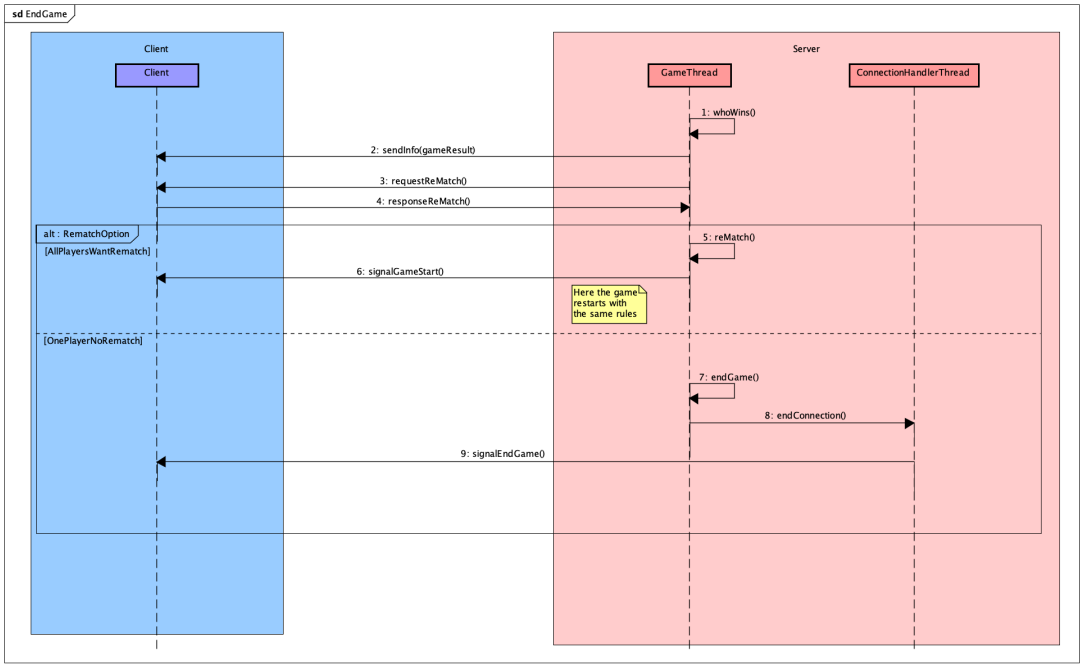
During the Game, the players could ask for information about the Game's or the Model's status. These requests are handled concurrently with GameThread using a loop in the InfoThread. This thread is initialized once at the end of SetUpPhase and will be active until EndGame. When a *requestValue()* is received by the InfoThread, it answers with a *responseValue()* contaning the value of the field/attribute the player requested.

2.4 CharacterCard



During the Game, the players could ask to use one of the CharacterCards. These requests are handled concurrently with GameThread in the CharacterCardThread. This thread is initialized once at the end of the SetUpPhase and will be active until the EndOfTurn or the EndGame. When a *responseAction(playCharacterCard)* is received by the CharacterCardThread, it answers with *requestAction(CharacterCards, costs)* containing the available Cards to choose from and their respective costs. The client further responds with his choice in the message *responseAction(CharacterCard)*.
The server then proceeds to remove the coins from the player (if he has enough to pay the cost, else he sends an error message) and then activate the effect of the card. The card may need a further communication with the player (e.g. move a student counter or a no entry tile) so a server-client request-response is present as *requestAction()* and *responseAction()*. Once the effect of the card is activated the server informs the client with a *sendInfo()*.

2.5 EndGame



This is the final stage of the Game: the client will arrive to this stage being forwarded by other stages, once reached the winning condition or end-game condition (finishing student or assistantCard). First, the EndGame determines which player(s) won the game (or if a game is a draw), then sends the game result to all the partecipants via a *sendInfo(gameResult)*. It sends also *requestReMatch()* and waits for *responseReMatch()*, where the player decides whether he wants to have another game, with the same players and the same rules, or not to do so.

Thus, depending on the players' responses to the server there could be two different scenarios:

- **AtLeastOnePlayerRefusedRematch**: the GameThread closes the current Game, then the ConnectionHandler closes the connection to all the players currently in the match.

- **AllThePlayersAcceptedRematch**: the GameThread initializes a new GameBoard with the same rules of the previous game, then sends a *signalGameStart()* to all the players, notifying them that a NewGame is starting.