COMMUNICATION PROTOCOL GC34

1. Message structure

Messages have a general structure; there are only two classes that represent messages: MessageFromClient and MessageFromServer.

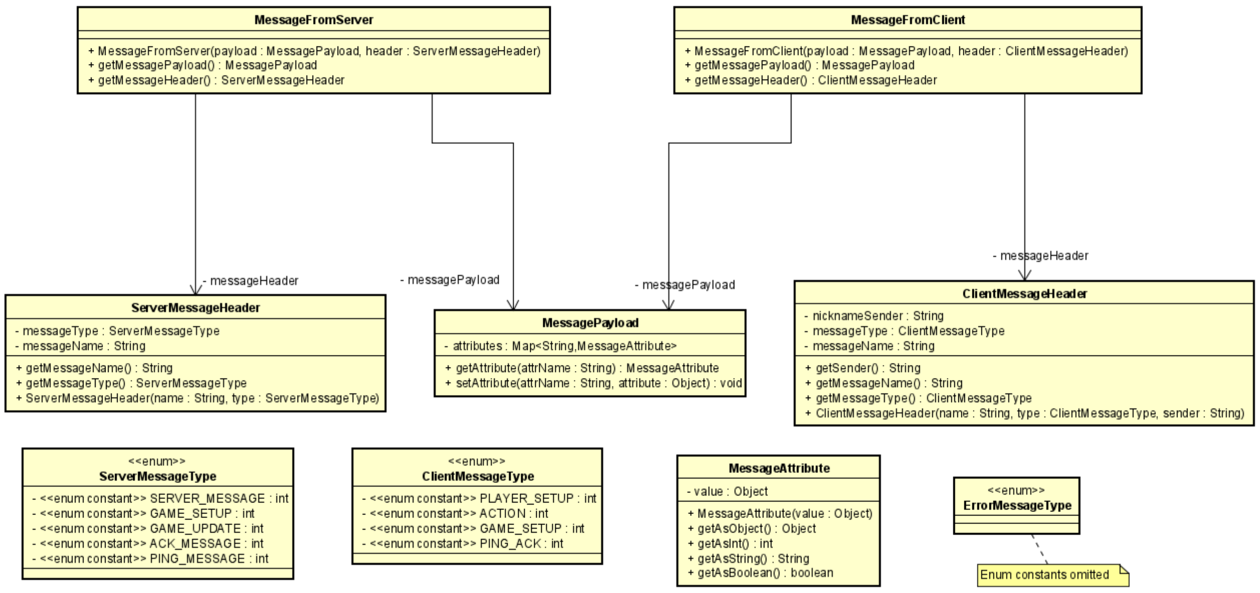
MessageFromClient have a ClientMessageHeader that contains the nickname of the message sender (to distinguish who sent the message), a ClientMessageType that will be used to determine which part of the server (including controllers) will handle the message and a message name (that will distinguish the specific message).

MessageFromServer is like MessageFromClient; indeed, it has a ServerMessageHeader that contains a ServerMessageType (client will use this to select the correct handler of the message) and a message name (which distinguish the specific message).

Both MessageFromClient and MessageFromServer have also a MessagePayload, which contains MessageAttributes. Each message attribute can be a different type of Object and is associated with a name (a String):

- Who send the message insert in the message payload an attribute by doing set(attributeName, attribute)
- Who receives the message can get the attribute by doing get(attributeName).

In base of the message name, who receives the attribute know what attributes contains the message and know attributes names (so it can easily obtain the attribute value).


MessageFromServer
+ MessageFromServer(payload : MessagePayload, header : ServerMessageHeader)
+ getMessagePayload() : MessagePayload
+ getMessageHeader() : ServerMessageHeader
MessageFromClient
+ MessageFromClient(payload : MessagePayload, header : ClientMessageHeader)
+ getMessagePayload() : MessagePayload
+ getMessageHeader() : ClientMessageHeader
- messageHeader
- messagePayload
- messagePayload
- messageHeader
ServerMessageHeader
- messageType : ServerMessageType
- messageName : String
+ getMessageName() : String
+ getMessageType() : ServerMessageType
+ ServerMessageHeader(name : String, type : ServerMessageType)
MessagePayload
- attributes : Map<String,MessageAttribute>
+ getAttribute(attrName : String) : MessageAttribute
+ setAttribute(attrName : String, attribute : Object) : void
ClientMessageHeader
- nicknameSender : String
- messageType : ClientMessageType
- messageName : String
+ getSender() : String
+ getMessageName() : String
+ getMessageType() : ClientMessageType
+ ClientMessageHeader(name : String, type : ClientMessageType, sender : String)
<<enum>>
ServerMessageType
- <<enum constant>> SERVER_MESSAGE : int
- <<enum constant>> GAME_SETUP : int
- <<enum constant>> GAME_UPDATE : int
- <<enum constant>> ACK_MESSAGE : int
- <<enum constant>> PING_MESSAGE : int
<<enum>>
ClientMessageType
- <<enum constant>> PLAYER_SETUP : int
- <<enum constant>> ACTION : int
- <<enum constant>> GAME_SETUP : int
- <<enum constant>> PING_ACK : int
MessageAttribute
- value : Object
+ MessageAttribute(value : Object)
+ getAsObject() : Object
+ getAsInt() : int
+ getAsString() : String
+ getAsBoolean() : boolean
<<enum>>
ErrorMessageType
Enum constants omitted

Enum class ErrorMessageType will be used by the Server to send error messages to the client.

2. Message Content

In this section message content will be presented in this way:

- *"messageName" //*eventual message explanation
  - *"attribute1Name": attribute1Type* //eventual attribute explanation
  - *"attribute2Name": attribute2Type*
  - *…*

Notes:

- a message can also have no attributes; for these messages only the message name will be presented
- some messages use a utility class that we created named Pair<T, V>, which associate an object of type T to an object of type V.

2.1 Message From Server

Server messages are divided into PING_MESSAGE, ACK_MESSAGE, GAME_SETUP, SERVER_MESSAGE, and GAME_UPDATE messages.

PING_MESSAGE is used to verify if a client is still connected to the server, it has a null message name and a null payload.

ACK_MESSAGE messages are used to confirm a client request. Server will always send a reply to a client message of type ACTION/PLAYER_SETUP, this reply can be an error message (which is a SERVER_MESSAGE) or an ACK_MESSAGE; if the client receives an error message his request is cancelled (this can mean that the request was malformed or that it wasn't correct by client to send such a request at that time, for example if a client request an action but it wasn't his turn). ACK_MESSAGE messages can be:

- *"ActionAck"* //acknowledge a client ACTION message
  - *"ActionName": String* //The action the server wants to acknowledge
  - *"NewPossibleAction": List<TurnPhase>* //null if turn ended
- *"SetupAck"* //acknowledge a client PLAYER_SETUP message
  - *"SetupName": String* //The setup that server wants to acknowledge (TowerChoice or WizardChoice

SERVER_MESSAGE messages are used by the server to notify client about an error or to send to the client general information during client setup. These messages are:

- *"Error"*
  - *"ErrorType": MessageErrorType*
  - *"ErrorInfo": String*
- *"NicknameRequest"* //request the client to choose a nickname
- *"GlobalLobby"* //sent after the client inserts a correct nickname
  - *"NumGamesNotStarted": Integer* //number of games that are created but not started (if 0 the client must create a game, otherwise he can choose to create a game or not)

    - *"GamesInfo": Map<Integer, Triplet<Integer,Boolean,String[]>>* //the first integer (the key of the map) is the game id, the integer in the Pair is the number of players of the game, the Boolean is they type of rules of the game (if true the game use expert rules, if false it use simple rules)
- *"GameLobby"* //sent after a client was inserted in a game
    - *"WaitingPlayers": String[]* //nicknames of the clients that are waiting in the lobby
    - *"NumPlayers": Integer*
    - *"Rules": Boolean*
- *"PlayerDisconnected"*
    - *"PlayerName": String*
- *"DeletedGame"* //informs the client that the saved game on which he was waiting has been deleted because another participant has decided to not resume it
    - *"ChoiceMaker": String* //the player who has decided to delete the game
- *"PreviousGameChoice"* //informs the client that he has a saved game on the server and ask him if he wants to resume it
    - *"NumPlayers": Integer*
    - *"Rules": Boolean*
    - *"Participants": String[]*

GAME_SETUP messages are sent by the server in the setup phase of the game and before the game has started; they are:

- *"PlayerJoined"*
    - *"Nickname": String* //nickname of the client that has joined the game lobby
- *"TowerTypeRequest"* //ask the client to choose a tower
    - *"FreeTowers": TowerType[]*
- *"WizardTypeRequest"* //ask the client to choose a wizard
    - *"FreeWizards": WizardType[]*
- *"TowerTaken"* //notify other clients that a client has chosen a tower
    - *"TowerType": TowerType* //tower chosen
    - *"PlayerName": String* //nickname of the client that have chosen the tower
- *"WizardTaken"*
    - *"WizardType": WizardType*
    - *"PlayerName": String*
- *"GameInitializations"* //message that notify that the game has started and send initializations of the field
    - *"Field": SimpleField* //islands, clouds, mother nature (, characters)
    - *"PlayersInfo": SimplePlayer[]*
- *"RestoredSetup"* //message sent if a game is restored, and it wasn't already started (this means that the setup phase was not completed)
    - *"SetupInfo": SimpleModel* //contains info about choices (for towers and wizards) already made by players
- *"GameRestoredData"* //message sent if a game is restored, and it was already started
    - *"SimpleModel": SimpleModel* //all info about the game state as it was before the server was interrupted

GAME_UPDATE messages are sent when there are changes in the game that needs to be notified to clients; these messages are:

- *"AssistantPlayed"*
  - *"AssistantId": Integer* //the assistant that is played
  - *"MotherNatureMovement": Integer* //mother nature movement of the played assistant
  - *"PlayerName": String* //player that have played this assistant
- *"ProfessorUpdate"* //a player has taken control of a professor
  - *"ProfessorType": RealmType*
  - *"PlayerName": String*
- *"MotherNatureMovement"*
  - *"InitialPosition": Integer*
  - *"FinalPosition": Integer*
- *"SchoolDiningRoomUpdate"* //students were inserted in/removed by the dining room
  - *"PlayerName": String* //school owner
  - *"Students": RealmType[]* //students inserted/removed
  - *"IsInsertion": Boolean* //true if students were inserted, false otherwise
  - *"IsFromEntrance": Boolean* //it means something only if is an insertion
- *"IslandStudentsUpdate"* //students were inserted in an island
  - *"IslandId": Integer*
  - *"Students": RealmType[]*
  - *"IsFromEntrance": Boolean*
- *"IslandUnification"* //a group of islands were merged
  - *"IslandsId": Integer[]*
  - *"NewIsland": SimpleIsland*
- *"IslandTowerUpdate"* //a player as conquered an island
  - *"Tower": TowerType*
  - *"IslandId": Integer*
- *"PickFromCloud"* //a player has picked students from cloud
  - *"CloudId": Integer*
  - *"Students": RealmType[]*
  - *"PlayerName": String*
- *"CharacterPlayed"*
  - *"CharacterId": Integer* //the id of the character
  - *"PlayerName": String*
- *"CharacterStudents"* //special (additional) message for characters 1, 7, 11 that specify the student type that is inserted in the character
  - *"Character": Integer*
  - *"Students": RealmType[]*
- *"MotherNatureMovementIncrement"* //special (additional) message for character 4
  - *"PlayerName": String*
  - *"Increment": Integer*
- *"SchoolSwap"* //special (additional) message for character 10
  - *"PlayerName": String*

    - *"ToEntrance": RealmType[]*
    - *"ToDiningRoom": RealmType[]*
- *"EntranceSwap"* //special (additional) message for character 7
    - *"PlayerName": String*
    - *"Inserted": RealmType[]*
    - *"Removed": RealmType[]*
- *"NoEntryTileUpdate"* //a no entry tile is inserted in an island
    - *"IsanldId": Integer*
- *"EndTurn"*
    - *"TurnEnder": String* //nickname of the player that has ended the turn
    - *"TurnStarter": String* //nickname of the player that will start the turn
    - *"PossibleActions": TurnPhase[]*
- *"ChangePhase"*
    - *"NewPhase": RoundPhase*
    - *"Starter": String*
    - *"PossibleActions": TurnPhase[]*
- *"EndGame"*
    - *"IsWinOrTie": Boolean* //true if someone has win, false if the game is tied
    - *"WinnersOrTiers": String[]* //nickname winner or nicknames tiers
- *"AssistantUpdate"* //update assistants that client can play, this message is sent before a new planning phase beginning; each client receives his assistants
    - *"Values": Integer[]*
    - *"MotherNatureMovements": Integer[]*
- *"CloudsRefill"*
    - *"Clouds": Integer[]*
    - *"CloudsStudents": RealmType[][]*
- *"CoinsUpdate"*
    - *"PlayerName": String*
    - *"OldCoins": Integer*
    - *"NewCoins": Integer*

2.2 Message From Client

Client messages are divided into GAME_SETUP, PLAYER_SETUP, ACTION, and PING_ACK messages.

PING_ACK message is used to keep alive the connection with the server in response to a server PING_MESSAGE; PING_ACK messages have a null MessagePayload and a null message name.

GAME_SETUP messages are used before a client is associated to game; they are:

- *"NicknameMessage"*
    - *"Nickname": String*
- *"NewGame"* //to create a new game
    - *"NumPlayers": Integer* //number of players of the game to create
    - *"GameRules": Boolean* //true if the game will be expert game, otherwise false
- *"GameToPlay"* //to join an existing game
    - *"GameId": Integer* //id of the game join
- *"RefreshGlobalLobby"* //ask the server to send an updated version of the global lobby

- *"RestoreGame"* //informs the server that the client wants to resume the previously saved game
- *"DeleteSavedGame"* // informs the server that the client doesn't want to resume the previously saved game (that will be deleted)
- *"QuitGame"* //informs the server that the client wants to abandon the game

Note: GAME_SETUP message doesn't require an acknowledgement from the server, because server always reply to these messages with another message (for example server reply to the nickname message with an error if the nickname is not valid or with a global lobby message).

PLAYER_SETUP messages are used in the setup game phase to communicate player choices to the server; these messages are:

- *"TowerChoice"* //client choose a tower in the game setup phase
    - *"Tower": TowerType*
- *"WizardChoice"* //client choose a wizard in the game setup phase
    - *"Wizard": WizardType*

ACTION messages correspond to a player action request during the game; these messages are:

- *"PlayAssistant"*
    - *"Assistant": Integer* //the assistant id that client wants to play
- *"MoveStudents"*
    - *StudentsToDR: RealmType[]* //students that the client wants to move to the dining room
    - *StudentsToIsland: List<Pair<RealmType, Integer>>* //students that client wants to move to islands, each one is associated with the island id
- *"MoveMotherNature":*
    - *"MotherNature": Integer* //mother nature movement in the island field
- *"PickFromCloud":*
    - *Cloud": Integer* //id of the cloud from which client wants to pick students
- *"EndTurn"* //client request to end the turn
- *"PlayCharacter":*
    - *"CharacterId": Integer*
    - *"Arguments": String* //character arguments represented as a string; this is used for characters that can take parameters, these characters are character 1, 3, 5, 7, 9, 10, 11, 12 (the string is used to have a general message for all characters, because all these characters take different parameters)

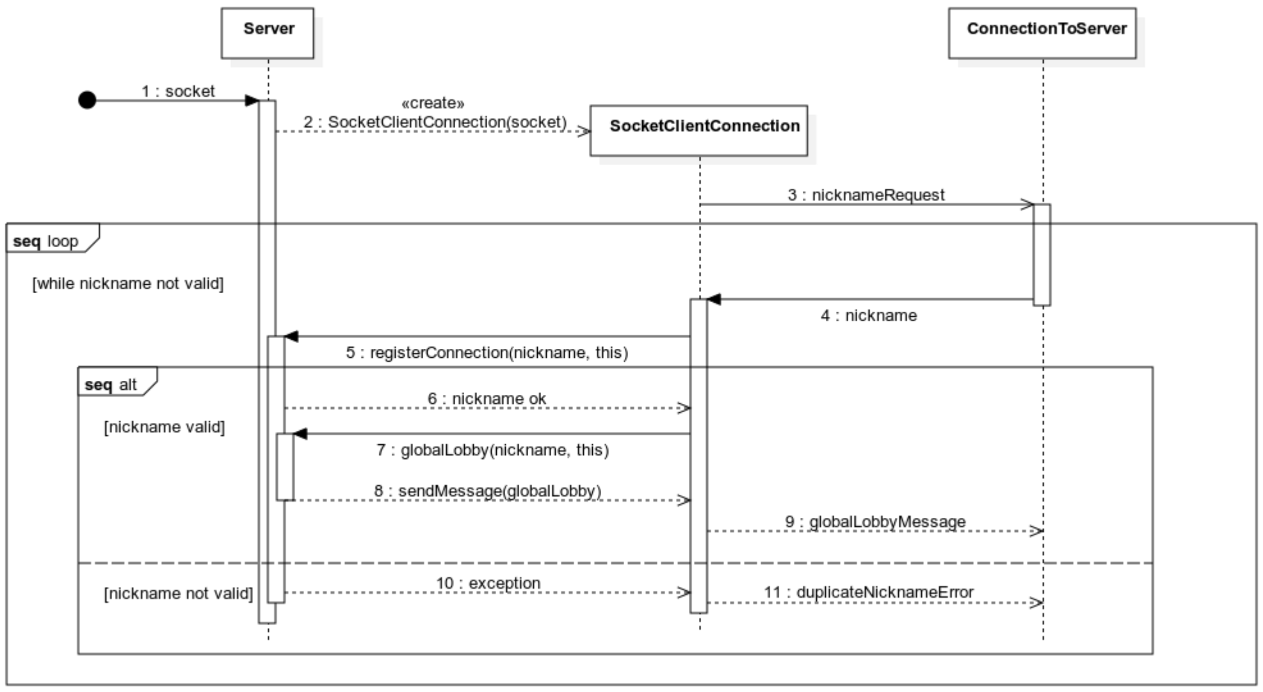
3. Sequence Diagrams

In the following sequence diagrams, SocketClientConnection is the server class that will send messages to the client and receive messages from client. In the client machine there will be the ConnectionToServer class that will send and receive messages.

Class MessageHandler is on the client machine and is used to handle messages that ConnectionToServer receives. Class TurnHandler is also on the client and will help the user (represented by a UserInterface that can be CLI or GUI) during his turn in a round.

1) Client login

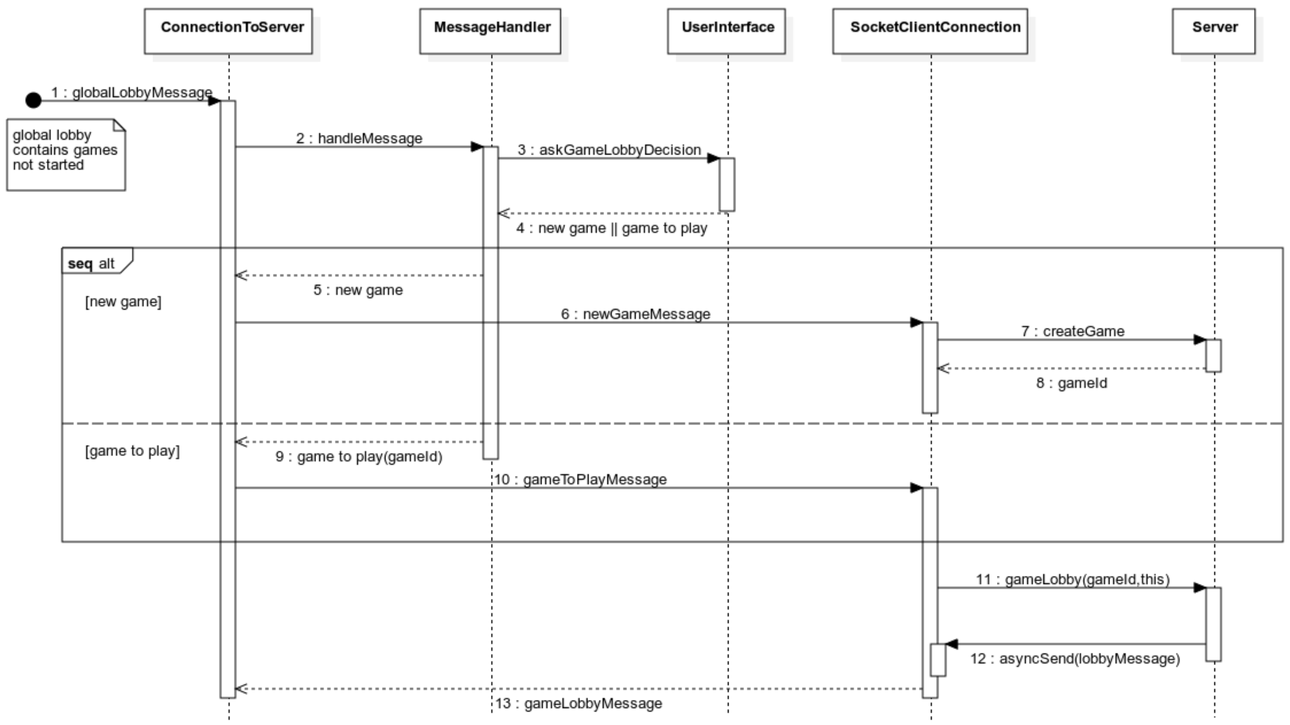
When a client tries to connect, the server creates a SocketClientConnection that will continue to ask client to insert a nickname until the client sends valid nickname.



2) Client game setup

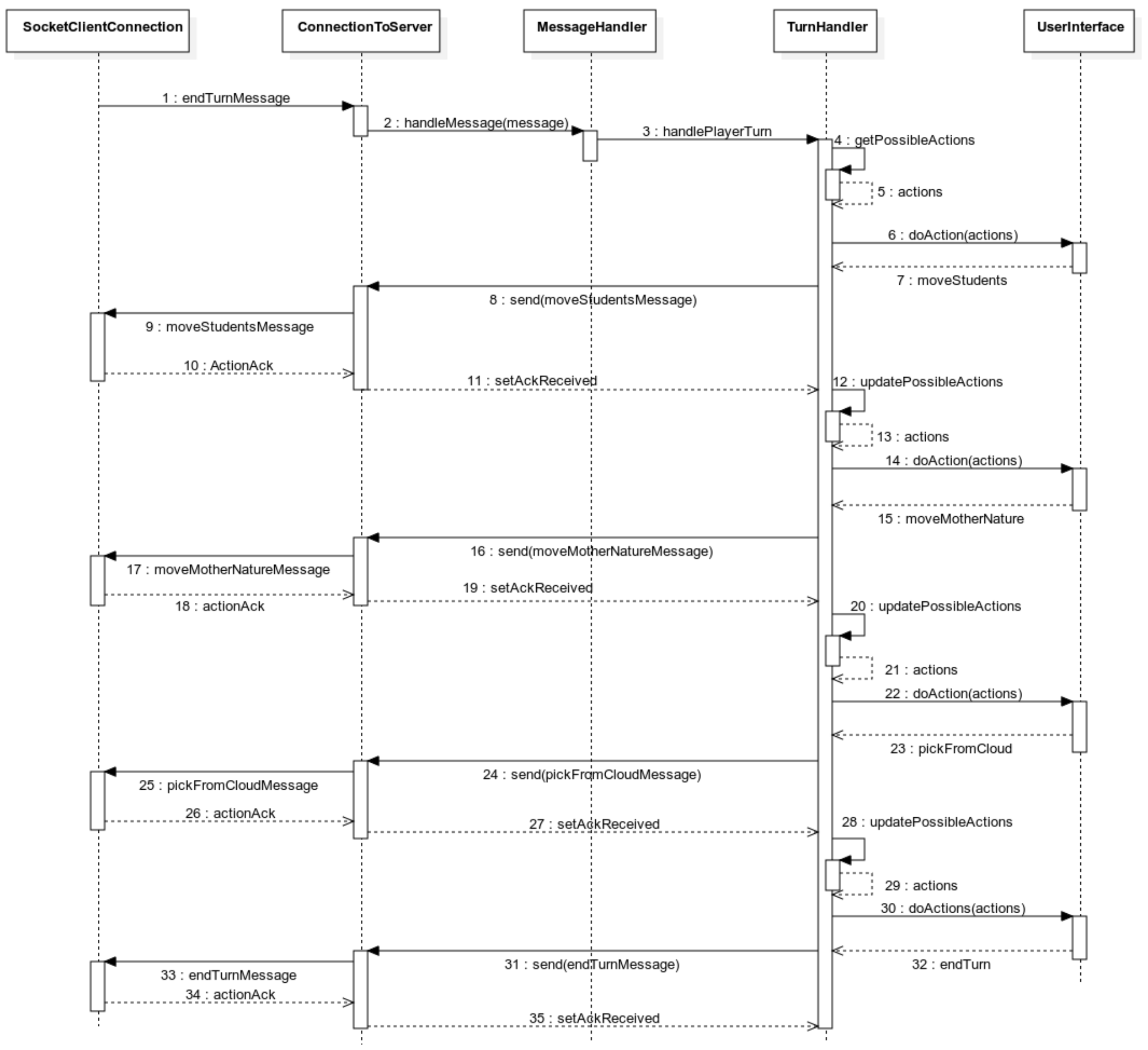
After client nickname setup is done, the server sends a *"GlobalLobby"* message to the client that contains information about all games that are currently created not started (because they haven't reached the number of players that were chosen by the player who creates the game). The client can choose either to create a new game (client sends a *"NewGame"* message) or to enter in a already created game (client sends a *"GameToPlay"* message). In both case client is inserted in a game lobby and the server respond with a *"GameLobby"* message.

If there are no games that are created but not started in the server, the client can only create a new game. In this diagram we assume that there are games not started, so the client can choose what to do.



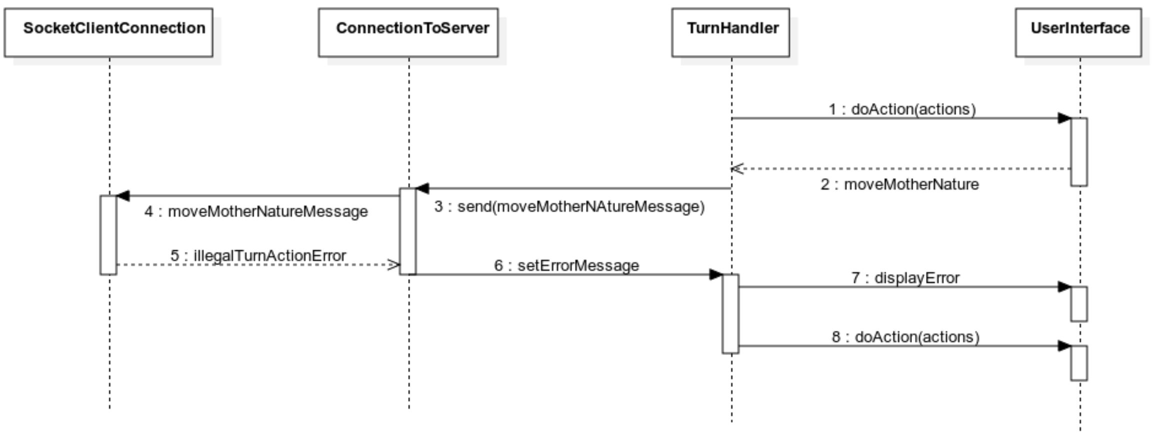
3) Client turns management

In the following diagram we assume that the game is started, and all the initializations (tower and wizard choices) are done; we assume also that the client is playing a game that is currently in the action phase of a round. When server sends a *"EndTurn"* message where the *"TurnStarter"* attribute contains the nickname of the client the TurnHandler class will be notified and will start to ask the user what action he wants to do. In this diagram we assume that the user always selects the correct action in the correct order (in this case we assume that the user don't want to play a character card), so all the actions that are sent to the server are acknowledged. After the user make an action, the TurnHandler notifies the ConnectionToServer that will send an action message to the server. The TurnHandler will wait until an acknowledgement or an error arrive before making the user choose another action to do. The user must end the turn explicitly (because a character card can be played at any time during the turn); when he does that a *"EndTurn"* message will be sent to the server. Also the *"EndTurn"* message needs to be acknowledged by the server because a client could send it before he has done all the mandatory actions (for example if he has moved mother nature but he hasn't picked students from a cloud).



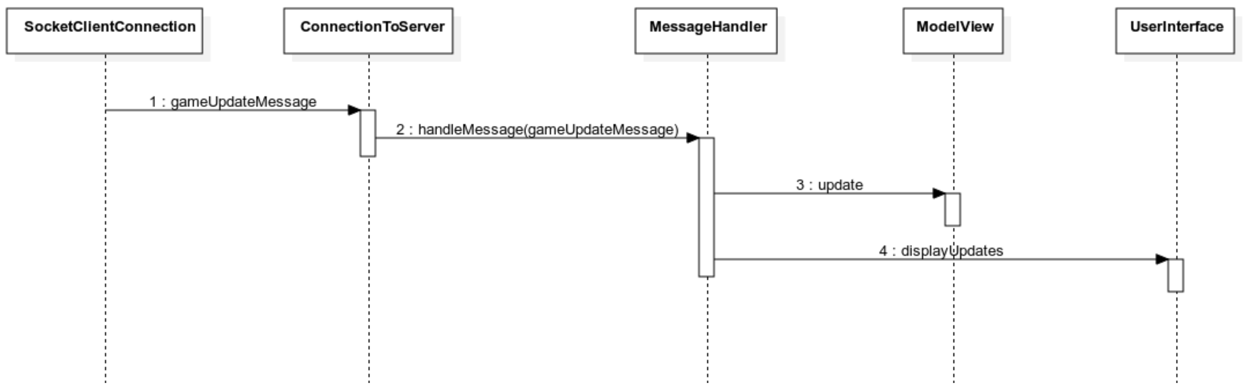
4) Client action with error

The following diagram is like the previous and assume that all the first 5 points of the previous diagram are done. So, the user have to do the first action of the turn (which have to be moving students or playing a character card). In this case the user decides to move mother nature before moving students, which cannot be done so the server sends an *"Error"* (wich has IllegalTurnAction as *"ErrorType"*), so the user have to remake the action.



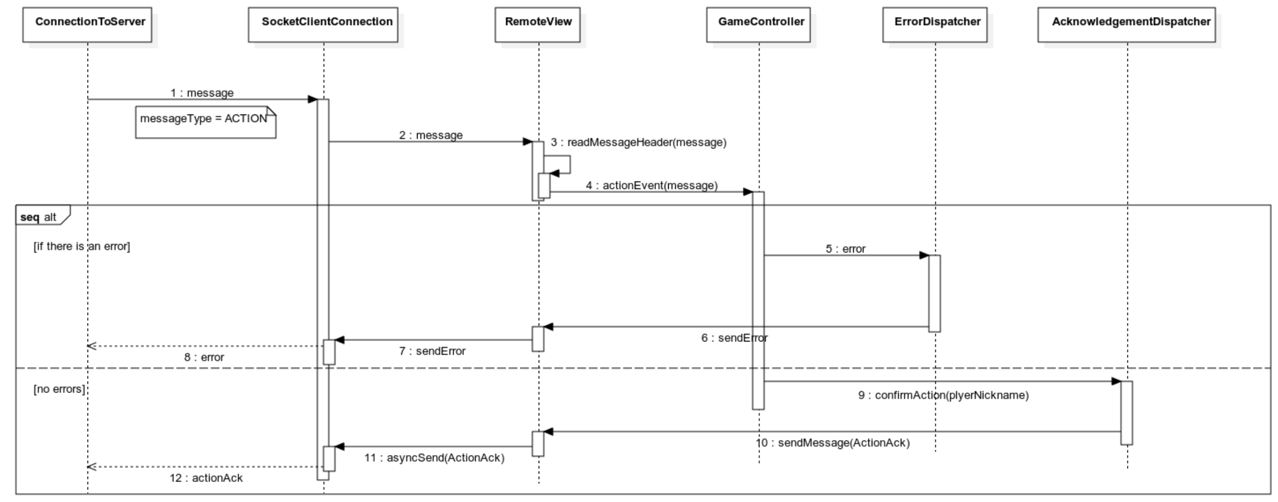
5) Client receives an update of the game

When client receives a message from the server which has messageType = GAME_UPDATE, the message will produce an update of the ModelView (a class on the client machine that is summary of the model of the game). This update will be notified also to the user interface. Server will always send the differences that players action produces in the game components and not all the information about the model every time. This message that arrives from the server doesn't need to be acknowledged, this because the TCP is a reliable protocol, so the message delivery is guaranteed. Also, this message doesn't need any reply by the client to the server.



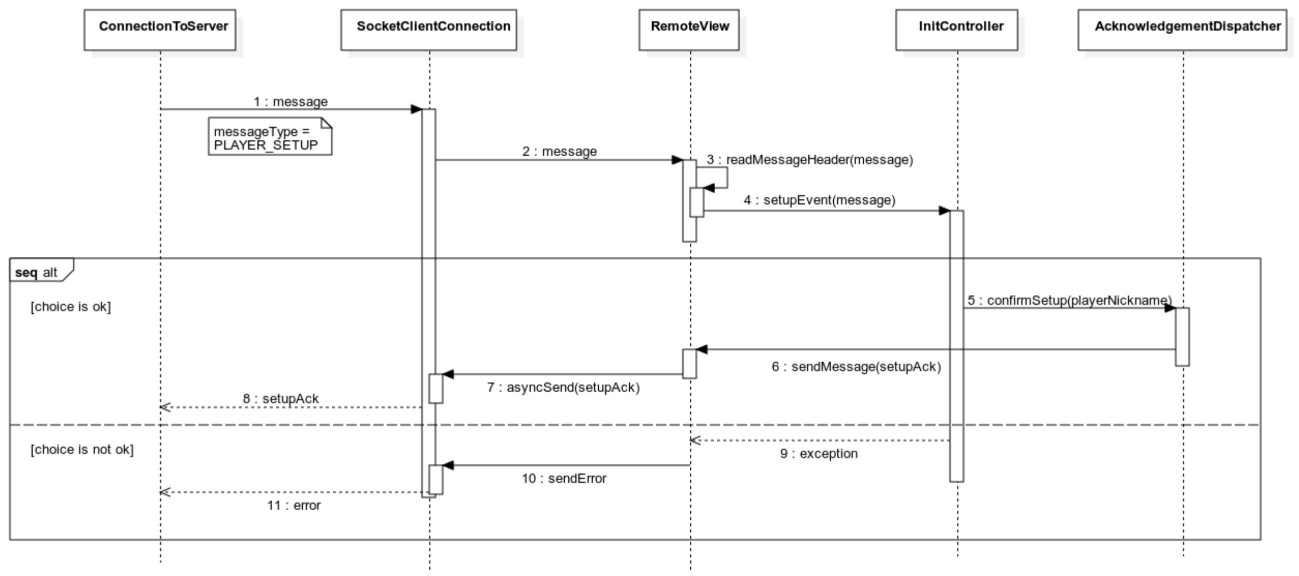
6) Server receives a client action message

This diagram shows what happens in the server-side when a client sends a message with messageType = ACTION. RemoteView's task is to read the header of the message and to forward it to the correct controller; for action messages the message is forwarded to the game controller. GameController will compute the action: if the action is malformed (for example if the one who send the message isn't the active turn player) it uses an ErrorDispatcher (which is a listener) to forward an error to the RemoteView, that will forward to the SocketClientConnection an *"Error"* message for the client; if everything is correct GameController uses an AcknowldgementDispatcher to send an ack message to the client.



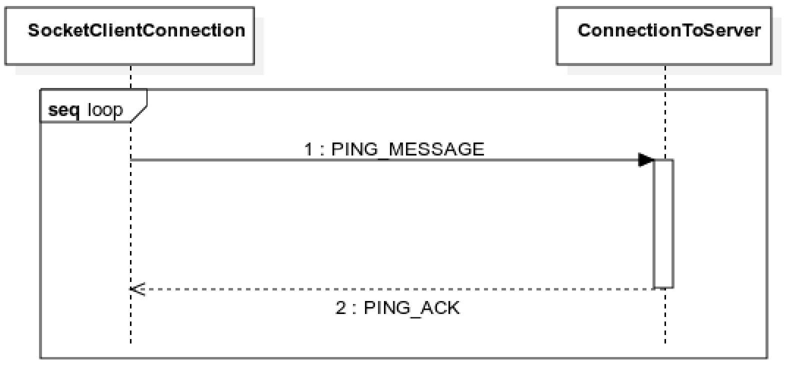
7) Server receives a player setup message

When a client sends a message with messageType = PLAYER_SETUP (it can be a tower or a wizard choice) the server process it in a similar way to the action message, with the difference that the message is forwarded to the InitController of the game.



8) Ping messages

Server will constantly send a ping message to the client, that must always reply with a ping ack message. If client doesn’t reply with a ping ack, the server will interpret it as a disconnection.


SocketClientConnection
ConnectionToServer
seq loop
1 : PING_MESSAGE
2 : PING_ACK