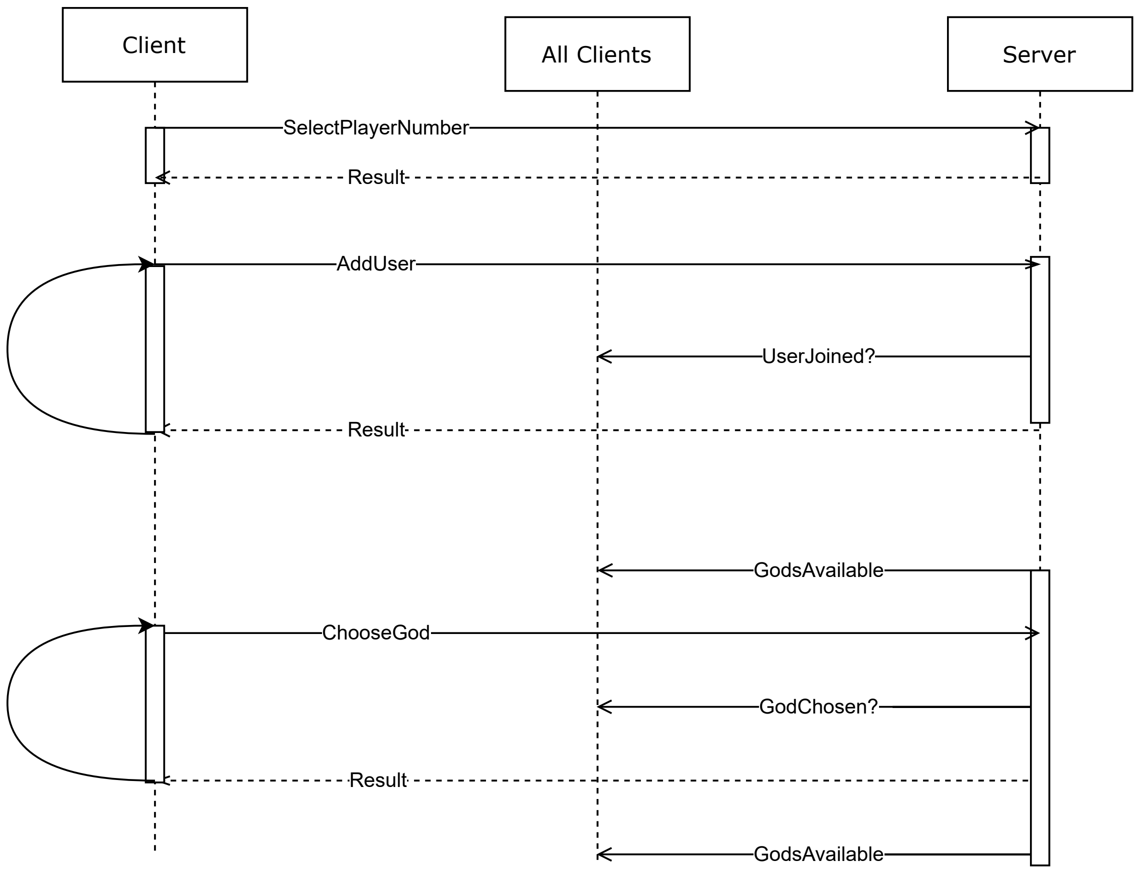
Communication Protocol

When the server is started it waits for clients to connect.
At first the server will require one client to set the lobby size, then it will allow users to login and set up their player and pawns. After this initial setup the server will enter the main game cycle and the users will be able to play the game.
The server only allows setting the lobby size once and logins with identical usernames will be forbidden, as well as logins when the lobby is full.
On client connection the server will direct the client to the lobby that is being filled. When the current lobby is full a new one will be started and successive connections will be directed to the new lobby. A client disconnecting before logging in will be ignored, although after a client has logged in and has chosen a username its disconnection will notify the users that their lobby is being terminated.
After finishing a game either because someone won or beacuse of an unrecoverable server error the server will notify the clients, close the lobby and clients may reconnect to play again.
After any unrecoverable error all clients are notified that the lobby will be terminated.

The next sections will show in detail the communication during the different phases of a match

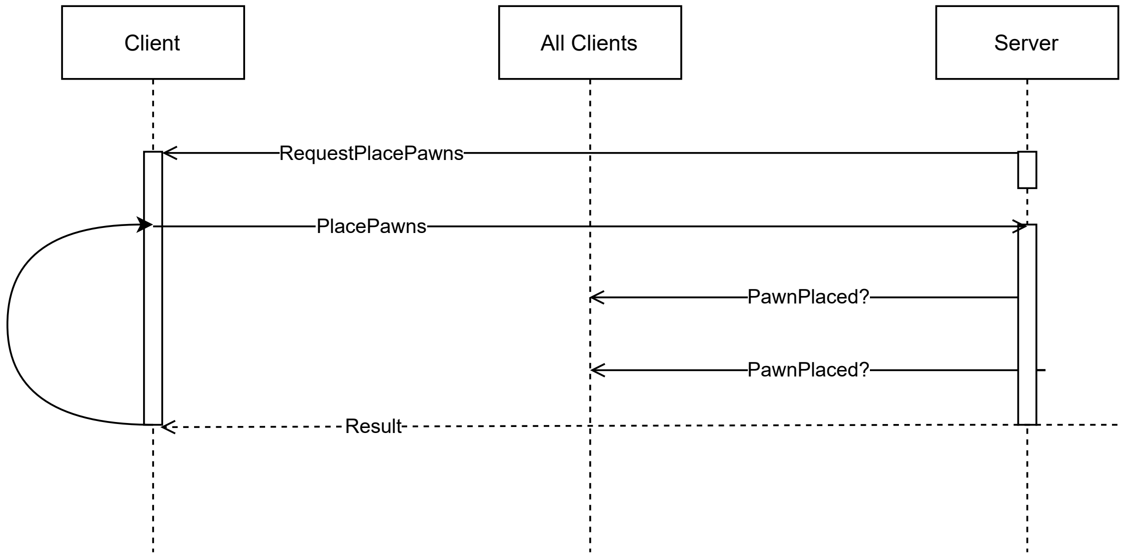
Setup: set lobby size and login

The fist client to send a SelectPlayerNumber message containing a valid size will set the lobby to the desired size and the server will now start accepting user logins.
The users can login using an AddUser message receiving a Result specifying whether the operation completed successfully and in case it was all the users will be notified of the newly created user.
When as many users as the lobby size are added the server will send a message to the users with the gods available to choose from and it will start accepting ChooseGod messages.
ChooseGod messages behave in a similar way to AddUser messages, the client will receive a result and other clients will be notified on success. When the last successful ChooseGod is processed the server will advance to the next phase.



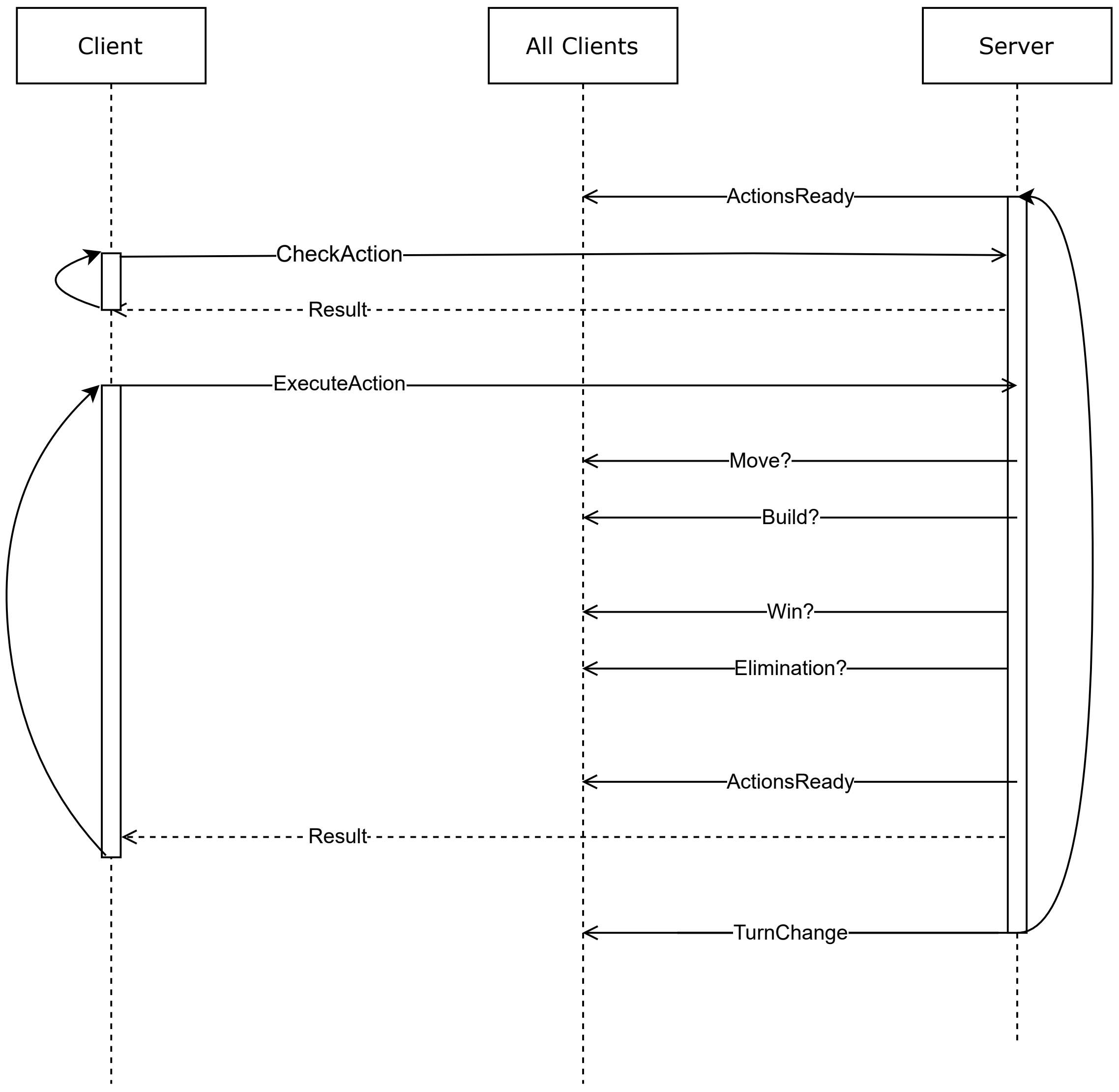
Setup: pawn placement

The server will now send a RequestPlacePawns message notifying users which user must now place their pawns.
The user will send a PlacePawn message with the desired coordinates (as usual it will receive a Result and all users will be notified of successful changes).
Trying to place pawns from another user will result in a negative Result.
When pawns for all users are placed the server will advance to the next phase.



Game cycle

During the main game cycle the server will send ActionReady messages notifying users which user should now execute an action and which actions he is allowed to choose from.
Users can send a CheckAction message containing the interested pawn, the desired action and the target coordinate, the server will answer with a result that will be positive if the user can execute the action with that specific pawn in that specific moment, negative otherwise. If any user other than the one previously chosen through the ActionReady message tries to execute an action it will receive a negative Result.
The user can send an ExecuteAction message that, after checking if the action is allowed, will be executed, the server will then notify all the users with the changes that happened to the model and if the move was a winning move it will send an Win message. Then, if the player's turn is not finished yet, another ActionsReady message will be sent containing the next allowed actions.
When the end of a user's turn has been reached a TurnChange message is sent and the cycle will restart asking for the next user to execute an action.



Message serialization

Messages are serialized in JSON form.

Example message:

```
{
    "type": "ACTION_TYPE",
    "content": {
        "user": {
            "username": "USERNAME"
        },
        "id": 0,
        "actionIdentifier": {
            "description": "MOVE"
        },
        "coordinate": {
            "x": 0,
            "y": 0
        }
    }
}
```

The *type* field is used to identify the type of message through an identifier and the *content* field will contain all the message specific parameters

Custom class serialization

This section shows how the relevant non-primitive classes are serialized

```
User: {"username": "USERNAME"}
ActionIdentifier: {"actionIdentifier": {"description": "DESCRIPTION"}}
GodIdentifier: {"name": "NAME", "description": "DESCRIPTION"}
Coordinate: {"x": 0, "y": 0}
Building: {"level":"LEVEL0", "dome":false}
```

Messages

Message reference

```
    ActionsReadyMessage(User user, List<ActionIdentifier> actionIdentifiers)
    AddUserMessage(User user)
    BuildMessage(Building building, Coordinate coordinate)
    CheckActionMessage(User user, int id, ActionIdentifier actionIdentifier, Coordinate coordinate)
    ChooseGodMessage(User user, GodIdentifier god)
    EliminationMessage(User user)
    ExecuteActionMessage(User user, int id, ActionIdentifier actionIdentifier, Coordinate coordinate)
    GodChosenMessage(User user, GodIdentifier godIdentifier)
    GodsAvailableMessage(List<GodIdentifier> gods)
    MoveMessage(Coordinate from, Coordinate to)
    PawnPlacedMessage(User owner, int pawnId, Coordinate coordinate)
    PlacePawnsMessage(User user, Coordinate c1, Coordinate c2)
    RequestPlacePawnsMessage(User user)
    ResultMessage(boolean value)
    SelectPlayerNumberMessage(int size)
    ServerErrorMessage(String type, String description)
    TurnChangeMessage(User user, int turn)
    UserJoinedMessage(User user)
    WinMessage(User user)
```