COMMUNICATION DEADLINE DOCUMENT - 14/05/2019

Prova Finale di Ingegneria del Software 2019
Gruppo AM43: Riccardo Bassani, Marco Bagatella, Davide Aldè
https://github.com/BassaniRiccardo/ing-sw-2019-alde-bagatella-bassani

Data format

The server communicates with clients by sending JsonObjects and receiving a string when requested. Every JsonObject has a "head" field, describing the type of the message, that allows the client to correctly process it. Five message types with different headers are defined:

PING
JSON: {"head" : "PNG"}
Test message to be ignored by the client, can be used for debugging purposes.

MESSAGE
JSON: {"head" : "MSG", "text" : "Hello world."}
A text message to be displayed that does not require an answer.
- "text" field, containing a string.

REQUEST
JSON: {"head" : "REQ", "text" : "Input a string.", "length", "16"}
A text message that requires a string to be sent back as an answer. The only constrain for the string is its maximum length.
- "text" field, containing the question.
- "length" field, containing an int representing the maximum length allowed.

OPTION
JSON: {"head" : "OPT", "text" : "Choose one.", "options" : ["1","2"]}
A text message and a list of options the client has to choose among. The client must send back a string containing the index of the chosen string.
- "text" field, containing the question.
- "options" field, containing a list of strings.

UPDATE
JSON: {"head" : "UPD", "mod" : JsonObject}
A ClientModel object serialized as a JsonObject, containing all the information the client needs about the model.
Sent when the client model needs to be updated.
-"mod" field, containing the Json object representing the model

Note on synchronization

As of the current state of the project, synchronization is guaranteed by the flow of execution of the program. While the server is constantly accepting messages, it is only supposed to receive a single one after a request has been sent to the client. The server must not receive messages in any other case (e.g. cannot receive messages from a client which was not asked anything) or unwanted behaviour can occur. We are considering a modification that "blocks" and discards all incoming messages when the server is not waiting for them. It can be implemented, so that an agent faking the execution of the client (e.g. connecting with another custom application) cannot exploit unwanted behaviours. However, this modification has not been implemented yet and is not a priority.

Note on input validation

As of right now, player input validation is carried out by the client, which keeps reading input until the message to the server have a suitable content. This means that message exchange is less intense, but that the server processes messages without checking if they respect any property and is vulnerable to malformed messages.
When basic functionalities have been implemented, and if this is believed to be a weakness, it is possible to move input validation to the server side.

Note on model updates

We are planning to implement the possibility to reverse a player's decisions in case he changes his mind. During a player's turn, the only way in which another player can interact is by using a grenade powerup as an answer to being damaged. It should not be possible to undo other players' actions. Moreover, it should not be able to re-attempt drawing a powerup card. This means that a player's actions are submitted and cannot be reversed once another player is given the chance to interact (basically, once another player is damaged) or once a player picks up an ammo tile.
A player's turn is therefore split into two actions and we are planning to give him the possibility to restore the state of the game to the beginning of the last action. However, once an action is completed, it can be no longer reversed. This is the reason which led us to design model updates in the following way.
The server model and the current player's model are updated in real time as a consequence of choices that have not necessarily been submitted yet. Any damage dealt is registered as soon as the current player confirms his choices, giving other player the possibility to react and sending them an updated model. When an action is reverted, all that needs to be done is to restore the server's model to the previous state (since only an handful of properties can be modified during an action this is not hard to implement) and to forward it to the current player.
This way of managing update guarantees consistency and respects the rule of the game, while also not bringing unfair advantage to any player.

Note on connection handling

The game server can handle multiple games at once and carries out matchmaking automatically. Clients can connect to the server both via Socket and RMI. Communication with clients is independent from routines that check for connection interruptions. Receiving and sending operations are designed not to be blocking.

Socket connection adopts a very standard design: each client can connect to a TCPServer (running on a dedicate thread) which initializes a PlayerController, handling player's input. This PlayerController manages the client's attempts to login and runs on a separate thread until the player disconnects or the login is successful. This allows different players to login simultaneously.

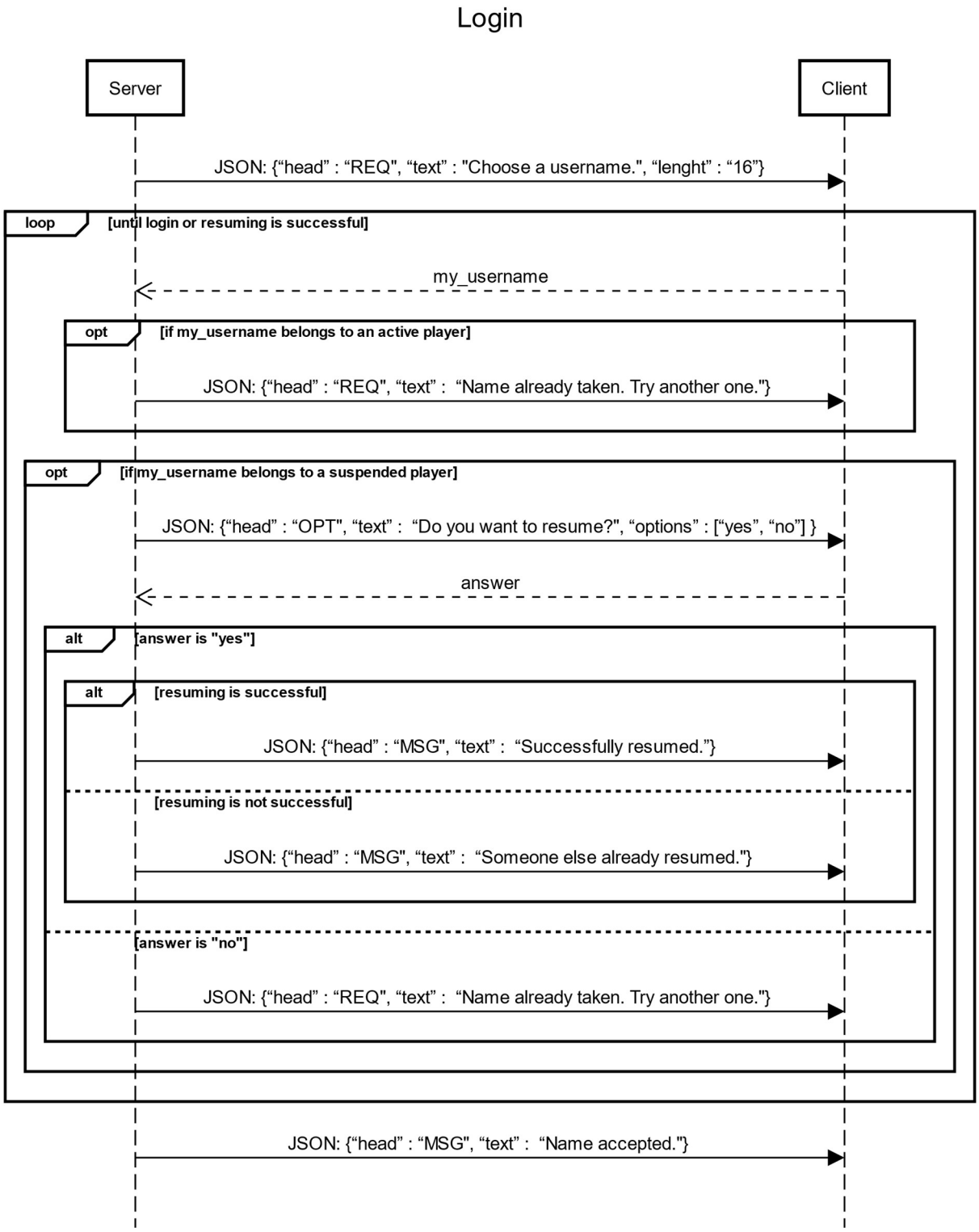
Clients that choose to use RMI can access a remote RMIServer bound to the RMIregister. A function can be called to create an RMIPlayerController, bind it and return the key to look it up to the client. The client can then send messages and periodically attempt to retrieve incoming ones by calling two remote functions of this PlayerController.

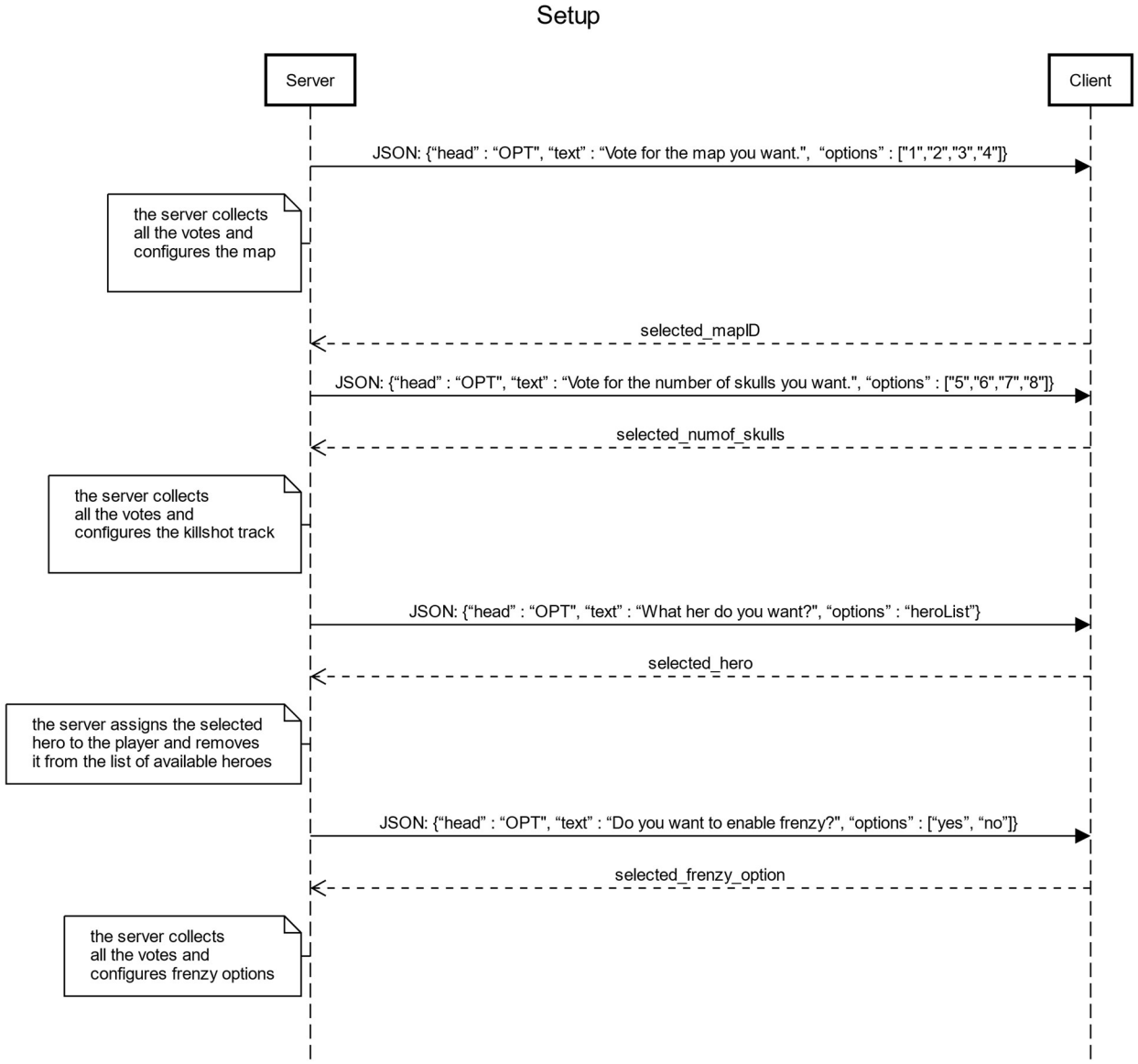
Once the client has logged in, a single thread for each client loops on retrieving messages from the connection, handling requests and sending messages back. Serverside, a single thread refreshes all TCP connections and no dedicate thread is needed for RMI.

Verifying if an agent has disconnected is a task handled separately.
An RMI client creates a thread periodically calling a remote ping() function. If this function throws a RemoteException the client can detect that the server is offline. The server can detect if the client has disconnected by checking how much time has passed since ping() was last called. TCP agents can simply attempt reading the socket's InputStream to verify if the other side is offline.
These checks are performed periodically and can spot a disconnection with short delays, the only exception being TCP clients while they are managing a server request. This can be modified by creating another thread to check the InputStream.

Message protocol

1) Communication before game start

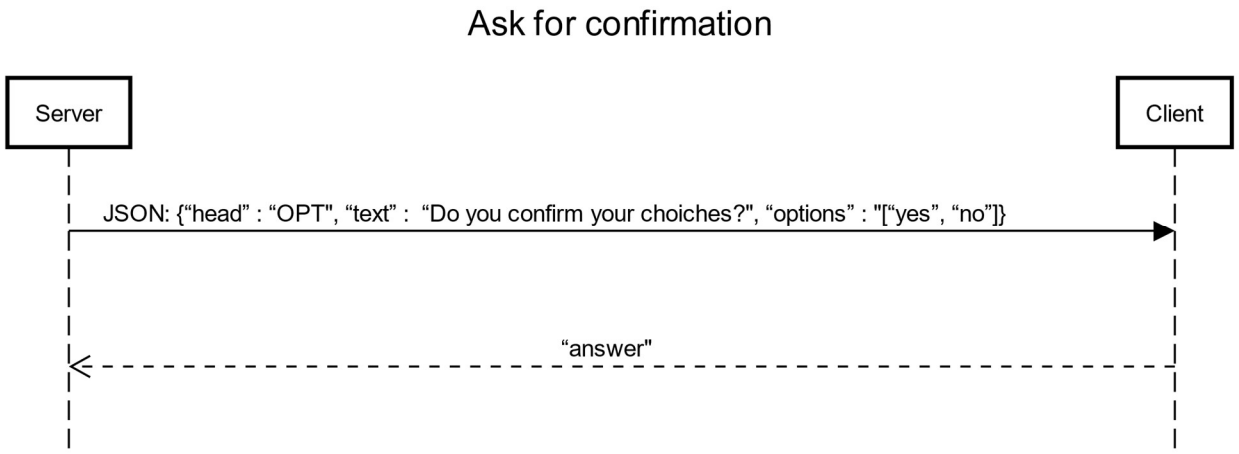




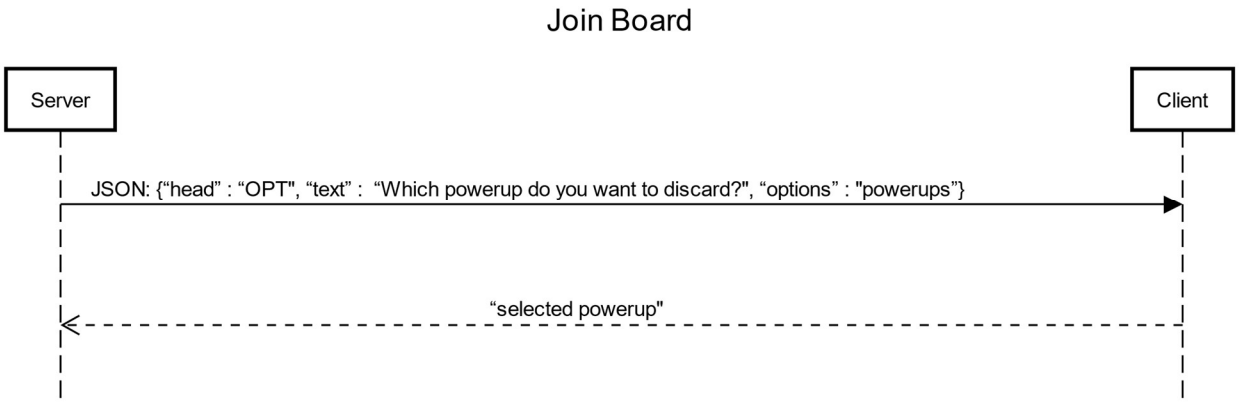
During the setup phase the server exchanges messages with the client in order to set the game options; the preferences of all players are taken into account.
In this phase the server does not allow players to change their decisions after they are taken.

2) Communication during game

The following diagrams model interactions between server and clients. Such exchanges happen in different orders and frequencies depending on the flow of the game. In several cases (when shown in the diagram) the OPT messages contain an additional “option” element: “reset”. When selected, the server cancels the player’s last moves so that he can act differently, or simply correct small mistakes or misclicks.

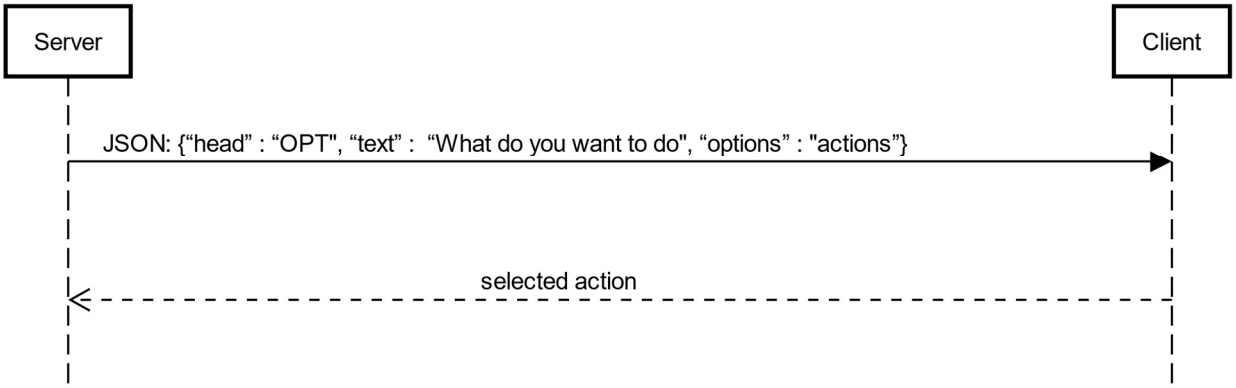


A player is generally asked for a confirmation before submitting a set of moves and applying their consequences permanently to the server. If the answer is no, the server restores the model to its last saved state.



“powerups” contains the powerups held by the player. This exchange occurs at the very beginning of the game.

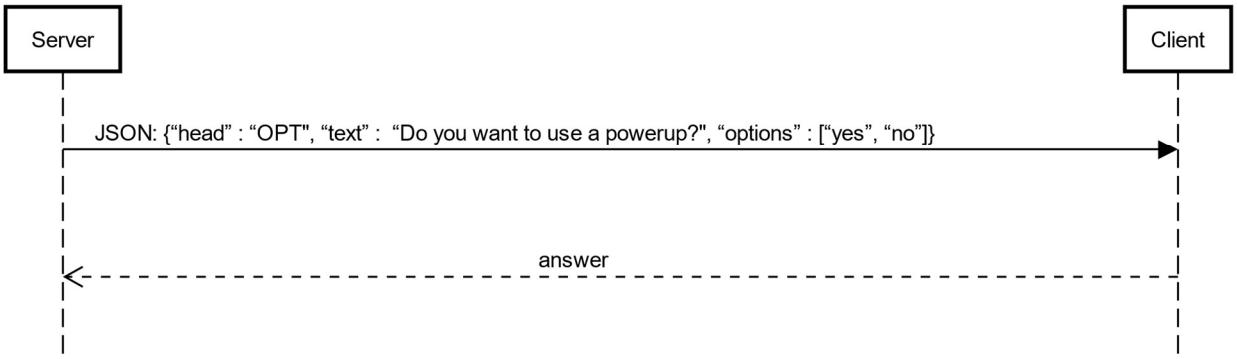
Execute action



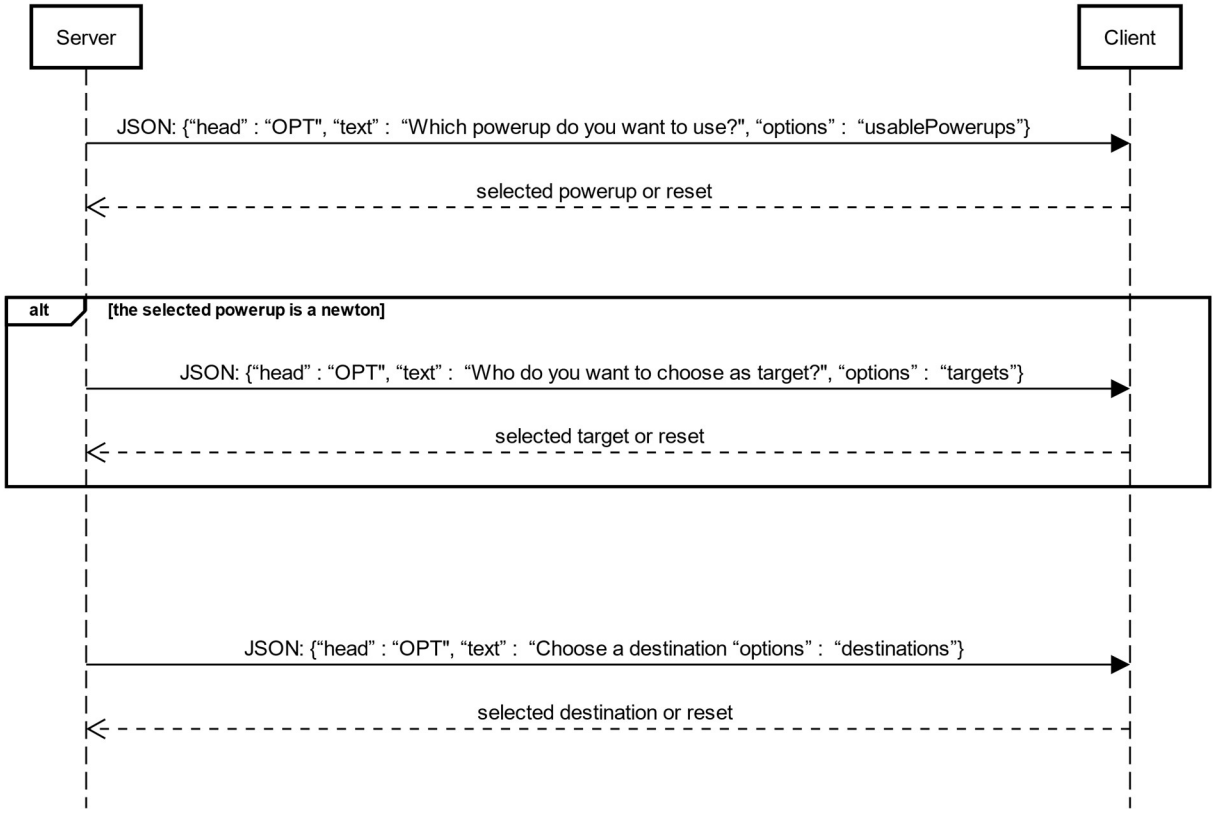
“actions” contains the actions the player can perform, including using or converting a powerup.

The server saves the selected action and then handles it through other methods.

Decide whether to use a powerup



Use powerup

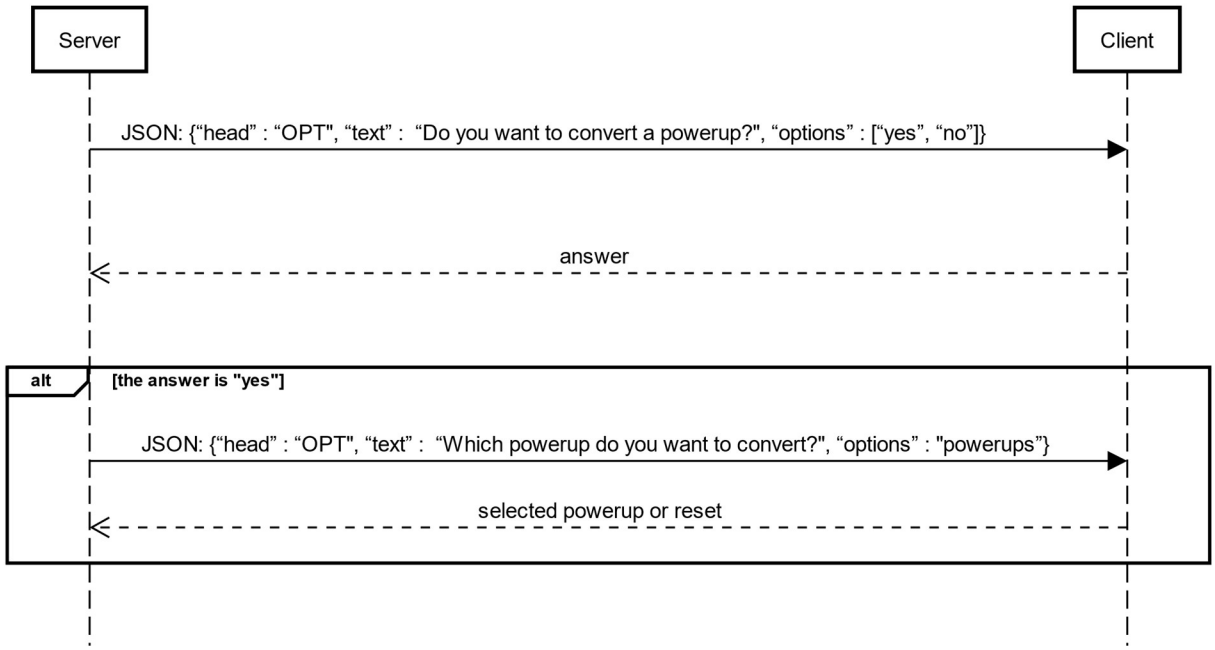


“usablePowerups” contains the player’s powerups that can be used between two actions, before or after an action.

“targets” contains the players that can be targeted by the selected powerup (the current player in some cases.

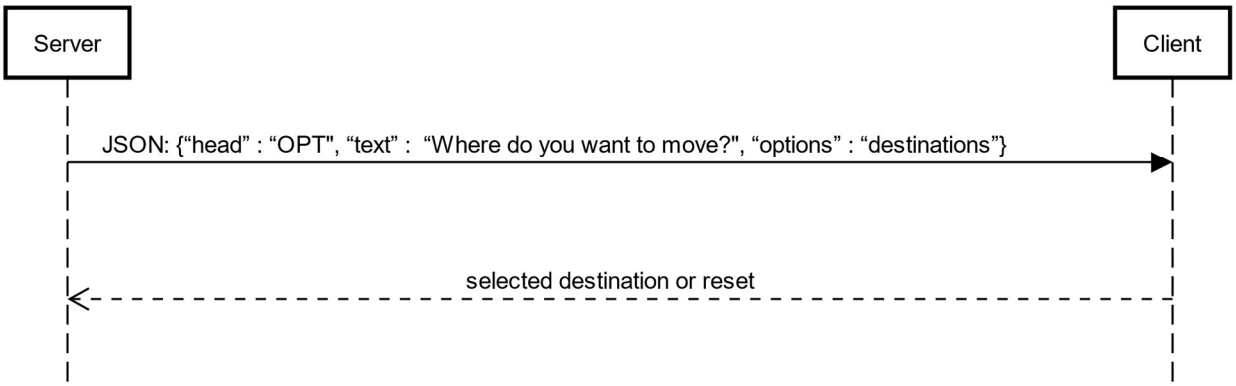
“destinations” contains the list of the squares where the player can move the target.

Convert powerup

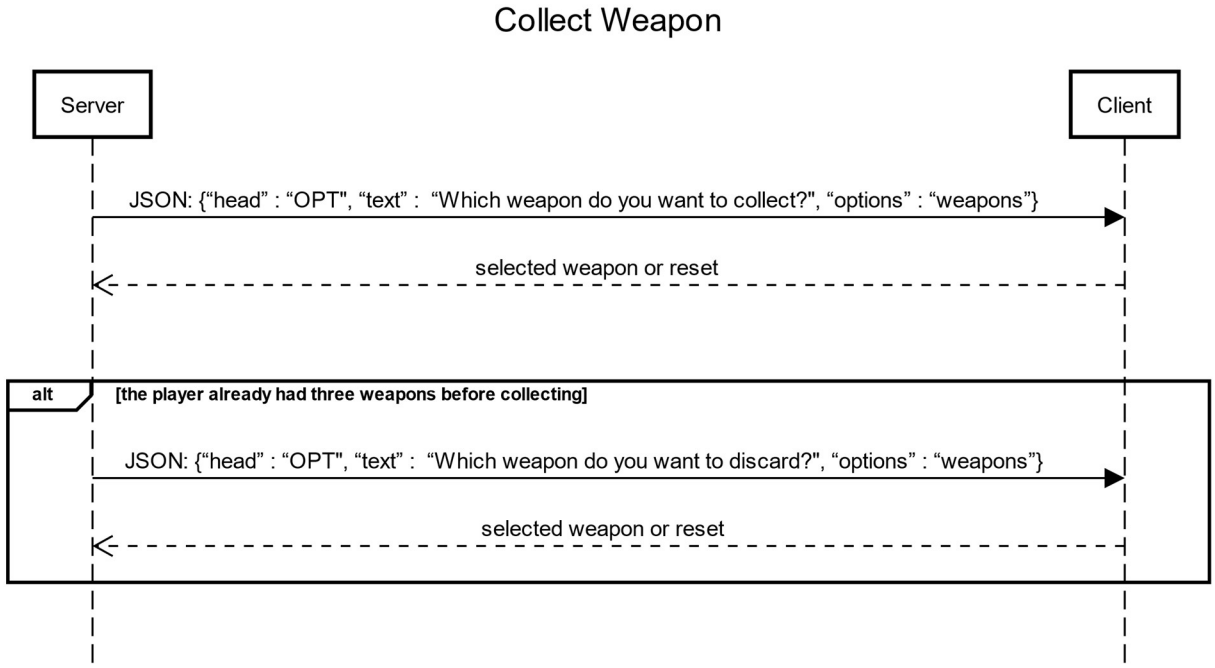


“powerups” contains the powerups held by the player.

Move



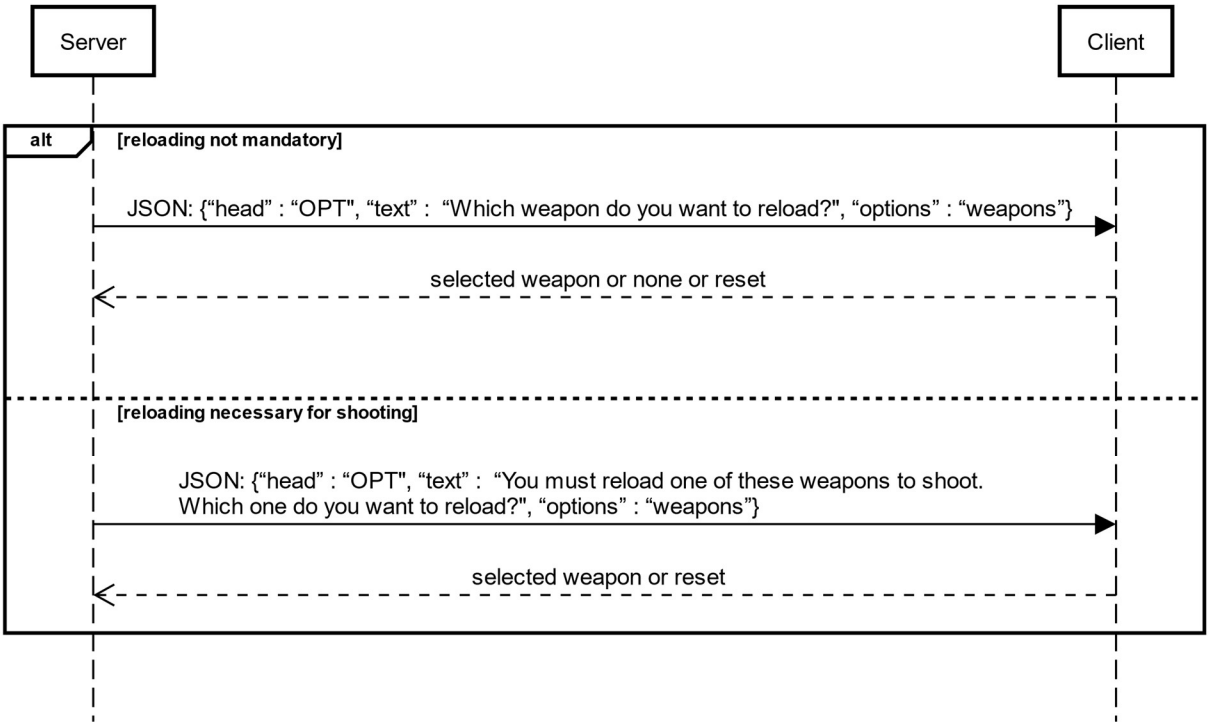
“destinations” contains the squares where the player can move, depending in the action he chose.



“weapons” in the first message contains the weapons the player can collect.

“weapons” in the second message contains the player weapons.

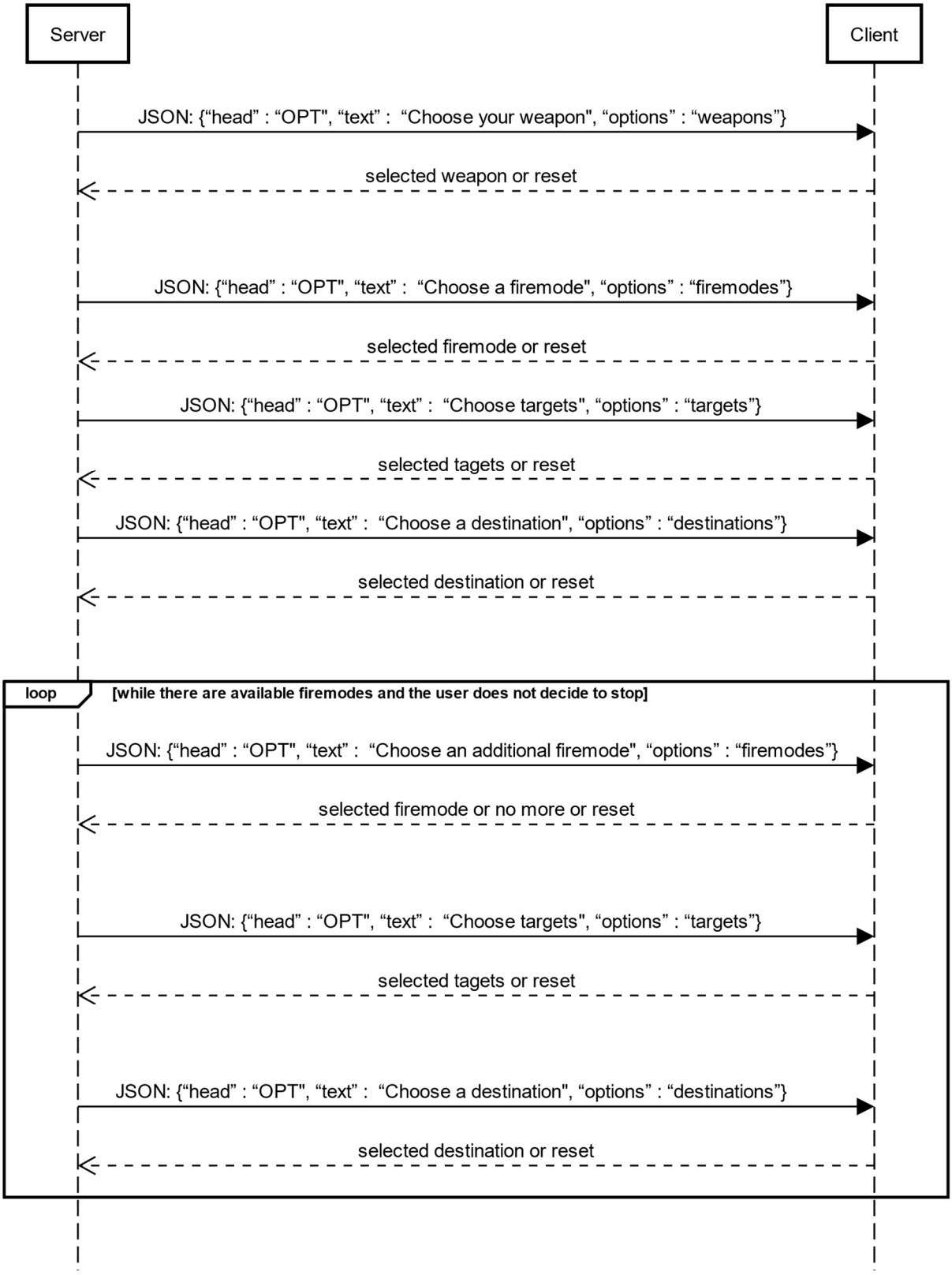
Reload



“weapons” in the first message contains the weapons the player can reload. It also contains the options "none".

every weapon in "weapons" in the second message can be reloaded and would allow the player to shoot.

Shoot


Server
Client
JSON: {“head” : “OPT", “text” : “Choose your weapon", “options” : “weapons”}
selected weapon or reset
JSON: {“head” : “OPT", “text” : “Choose a firemode", “options” : “firemodes”}
selected firemode or reset
JSON: {“head” : “OPT", “text” : “Choose targets", “options” : “targets”}
selected tagets or reset
JSON: {“head” : “OPT", “text” : “Choose a destination", “options” : “destinations”}
selected destination or reset
loop
[while there are available firemodes and the user does not decide to stop]
JSON: {“head” : “OPT", “text” : “Choose an additional firemode", “options” : “firemodes”}
selected firemode or no more or reset
JSON: {“head” : “OPT", “text” : “Choose targets", “options” : “targets”}
selected tagets or reset
JSON: {“head” : “OPT", “text” : “Choose a destination", “options” : “destinations”}
selected destination or reset

“weapons” contains the weapons the player can use.

“firemodes” contains the available firemodes of the weapon.

“targets” contains the possible targets of the firemode.

“destinations” contains the possible destination of the firemode.

“firemodes” in the loop contains the firemodes that can be added. It also contains the option “no more”.

Document redacted by Riccardo Bassani (@BassaniRiccardo) and Marco Bagatella (@marcobaga)