COMMUNICATION DEADLINE DOCUMENT - 14/05/2019

[revised 03/07/2019]
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 can communicate with multiple clients either through a socket connection or via RMI. In the second case, message exchange is limited, as for most purposes a remote client function can be called by the server (callback), passing information for the client as parameters and retrieving information as a return value. For example, when it is time for the player to choose which weapon to shoot with, the server can call the remote function choose() passing a list of strings representing weapons, which will return a value corresponding to the player's choice. If TCP connection is chosen, however, requests for the player need to be serialized (as JsonObjects) and sent through the socket. The client will then answer with a simple String.

Every JsonObject has a "head" field, describing the type of the message, that allows the client to correctly process it. Seven message types with different headers are defined:

ME**S**SA**G**E

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", "type" : "string", "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.
- "type" field, containing the type of options.
- "options" field, containing a list of strings.

SUSPENSION

JSON: {"head" : "SUSP"}

This message notifies a player that it was suspended (usually because the turn timer ran out).

END OF GAME

JSON: {“head” : “END", “msg” : “You won”}

This message notifies the player of the end of the game and contains the results.

- “msg” field, containing the results

<u>UPD</u><u>ATE</u>

JSON: {“head” : “UPD", “type” : “typeOfUpdate”, additional fields }

- *“type” field, declaring the type of change to the model*
- *other specific fields*

There are three categories of update message.

The first category contains messages of various type, that are sent when a specific element of the model is modified. Every message of this kind contains only the information regarding the modified element. The type of messages belonging to this category and the relative fields are listed below.

“type”	additional fields
“reload”	“weapon”, “loaded”
“skullRemoved”	“number”, “killer”, “overkill”
“powerUpDeckRegen”	“number”
“drawPowerUp”	“player”, “powerupname”, “powerupcolor”
“discardPowerUp”	“player”, “powerupname”, “powerupcolor”
“pickupWeapon”	“player”, “weapon”
“discardWeapon”	“player”, “weapon”
“pickupWeapon”	“square”, “weapon”
“useAmmo”	“player”, “redammo”, “blueammo, “yellowammo”
“addAmmo”	“player”, “redammo”, “blueammo, “yellowammo”
“move”	“player”, “square”
“status”	“player”, “status”, “boolean”
“addDeath”	“player”
“damage”	“player”, “list”
“mark”	“player”, “list”
“removeWeapon”	“square”, “weapon”
“setInGame”	“player”, “ingame”
“removeAmmoTile”	“square”

The second category contains only the “mod” type. The information in the message is a ClientModel object serialized as a JsonObject, containing all the information the client needs about the model.

It is sent to assure the client model has been correctly updated by the partial update messages previously sent. Since it is sent to every player every turn, it guarantees that the models of different players are coherent.

The last category (*“type”: “render”*) does not apply changes to the ClientModel, but forces it to render previous changes graphically.

Note on synchronization

In both TCP and RMI implementations, the server can only forward one request to each client at a time. To ensure synchronization, all attempts to send further requests to a busy client are ignored and not sent. Similarly, all answers coming from the client are discarded unless previously requested by the server and on time. It is important to note that in normal circumstances the server waits for its clients to answer and clients only answer when prompted to do so, but these additional filters grant the server robustness to the violation of this assumption.

Note on input validation

As of right now, player input validation is carried out by the client, which keeps reading (keyboard or mouse) 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. Moving input validation to the server side is a straightforward task which had to be discarded due to time constraints.

Note on model updates

We implemented 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 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). All changes to the server model are automatically converted in update messages and added to queues. It is possible to control the state of ClientModels simply by choosing when to forward such messages.

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. This function takes a remote client object as a parameter to implement callback. This means that the server can invoke remote methods of the client for normal game interactions, while the client only invokes server methods to check the status of the connection.

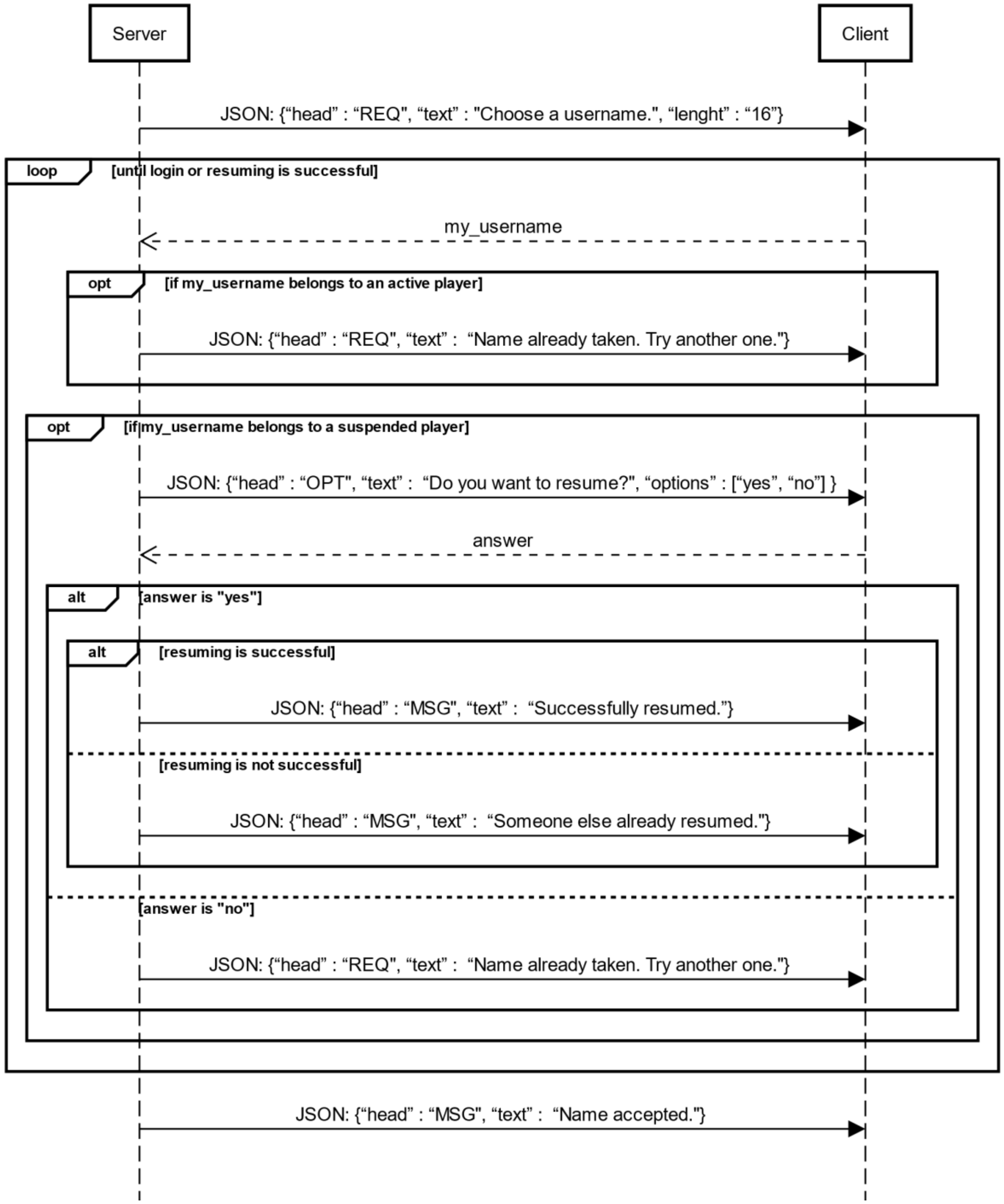
Once the client has logged in, in case of RMI connection no thread is explicitly started, while for TCP a single thread loops on retrieving messages from the connection, handling requests and sending messages back. Serverside, a single thread refreshes all TCP connections and no dedicate thread is needed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

Verifying if an agent has disconnected is a task handled separately.
For RMI, both the server and the client have classes imlementing a remote interface with a ping() method. On both sides it is possible to call this function periodically to survey the status of the connection (this will be done by the same thread refreshing TCP connection on the server and by a separate thread on clients).
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.

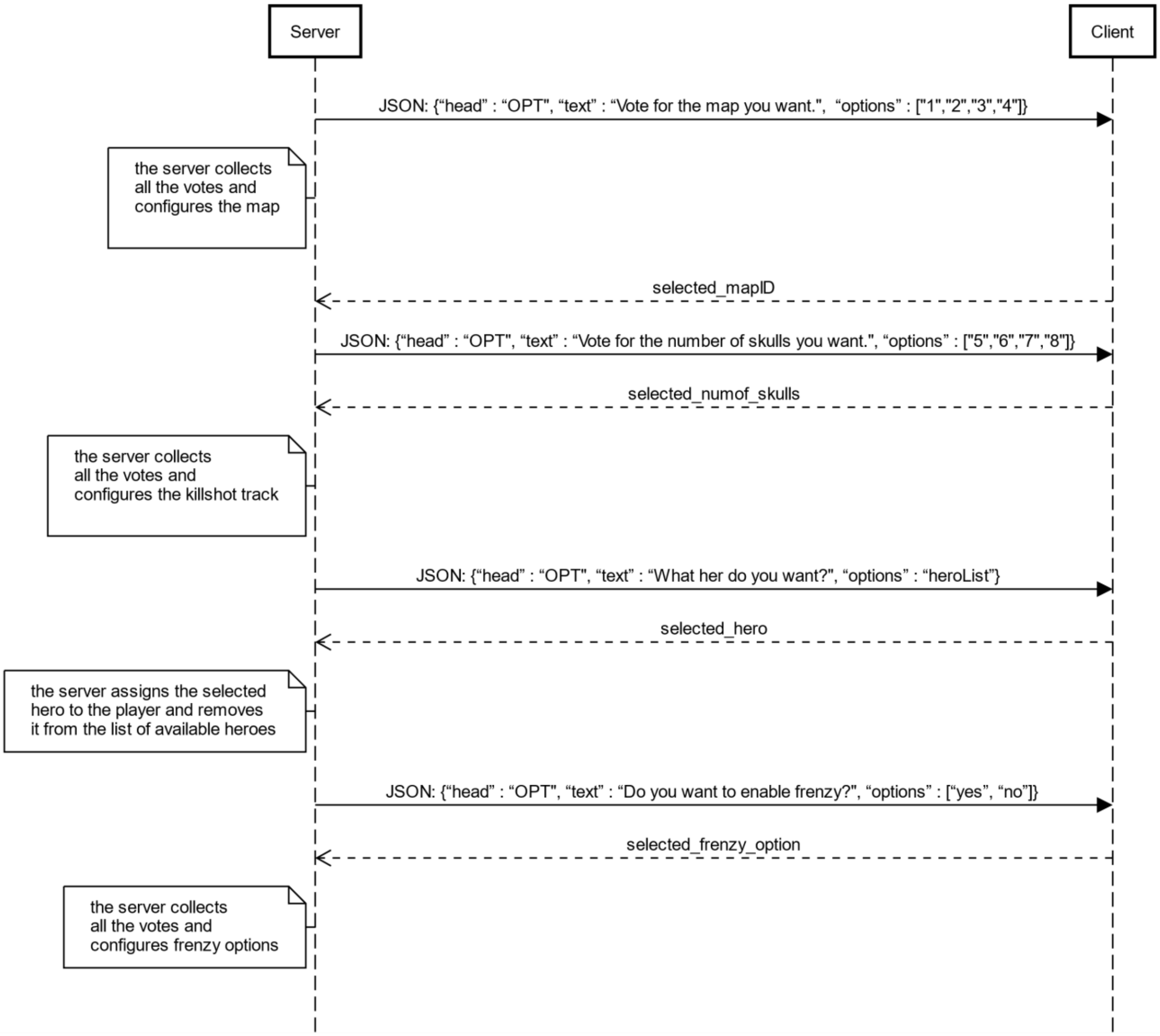
Message protocol

1) Communication before game start

Login


Server
Client
JSON: {“head” : “REQ", “text” : "Choose a username.", “lenght” : “16”}
loop
[until login or resuming is successful]
my_username
opt
[if my_username belongs to an active player]
JSON: {“head” : “REQ", “text” : “Name already taken. Try another one."}
opt
[if my_username belongs to a suspended player]
JSON: {“head” : “OPT", “text” : “Do you want to resume?", “options” : [“yes”, “no”] }
answer
alt
[answer is "yes"]
alt
[resuming is successful]
JSON: {“head” : “MSG", “text” : “Successfully resumed."}
[resuming is not successful]
JSON: {“head” : “MSG", “text” : “Someone else already resumed."}
[answer is "no"]
JSON: {“head” : “REQ", “text” : “Name already taken. Try another one."}
JSON: {“head” : “MSG", “text” : “Name accepted."}

Setup



JSON: {“head” : “OPT", “text” : “Vote for the map you want.”, “options” : ["1","2","3","4"]}

the server collects all the votes and configures the map

selected_mapID

JSON: {“head” : “OPT", “text” : “Vote for the number of skulls you want.”, “options” : ["5","6","7","8"]}

selected_numof_skulls

the server collects all the votes and configures the killshot track

JSON: {“head” : “OPT", “text” : “What her do you want?”, “options” : “heroList”}

selected_hero

the server assigns the selected hero to the player and removes it from the list of available heroes

JSON: {“head” : “OPT", “text” : “Do you want to enable frenzy?”, “options” : [“yes”, “no”]}

selected_frenzy_option

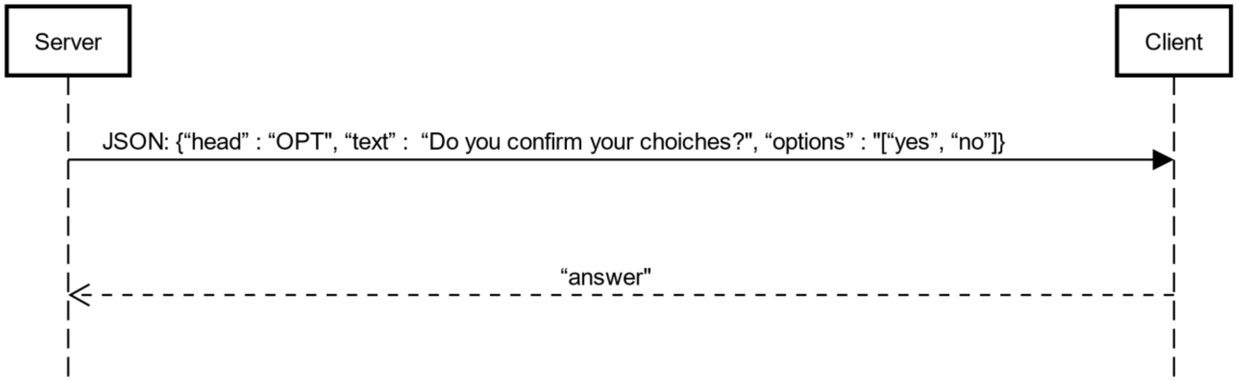
the server collects all the votes and configures frenzy options

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. All of 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.

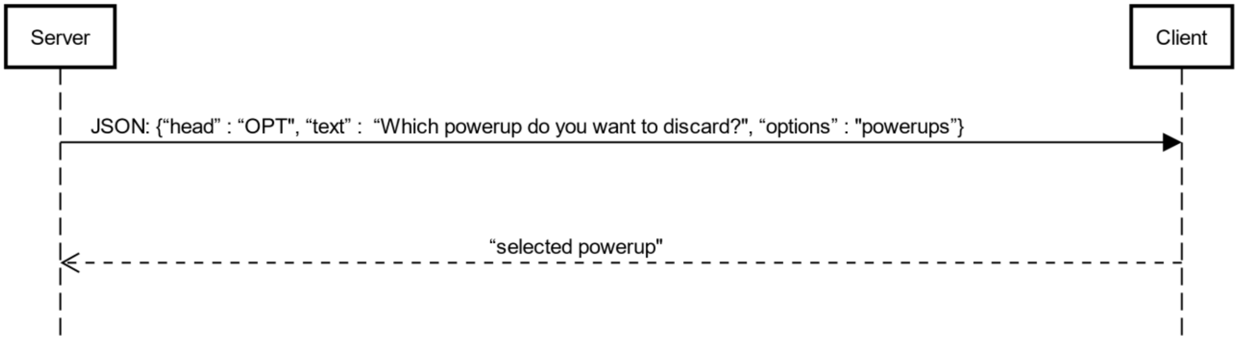
Ask for confirmation



JSON: {“head” : “OPT", “text” : “Do you confirm your choiches?", “options” : "[“yes”, “no”]}

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.

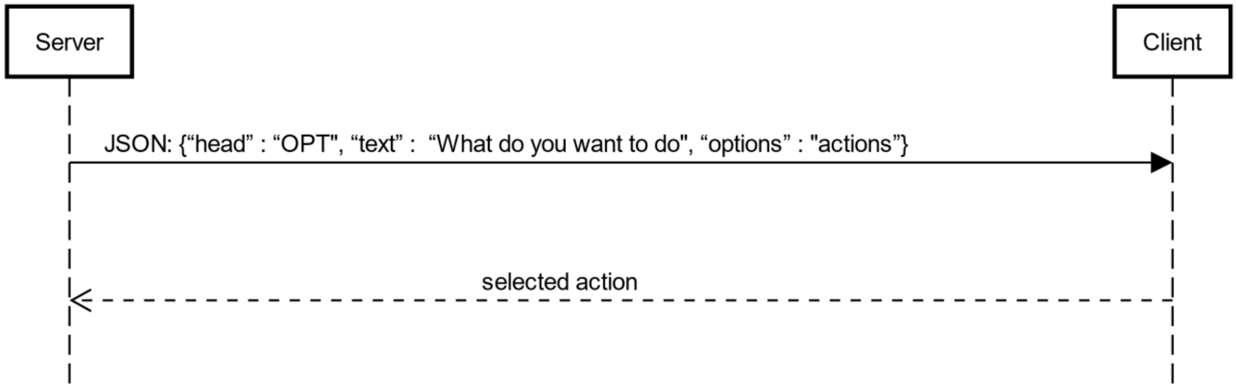
Join Board



JSON: {“head” : “OPT", “text” : “Which powerup do you want to discard?", “options” : "powerups”}

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

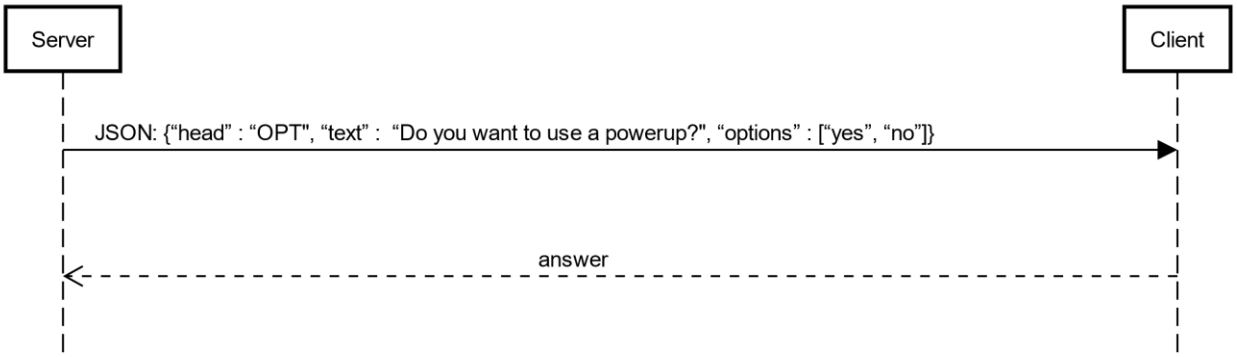
Execute action


Server
Client
JSON: {“head” : “OPT", “text” : “What do you want to do", “options” : "actions”}
selected action

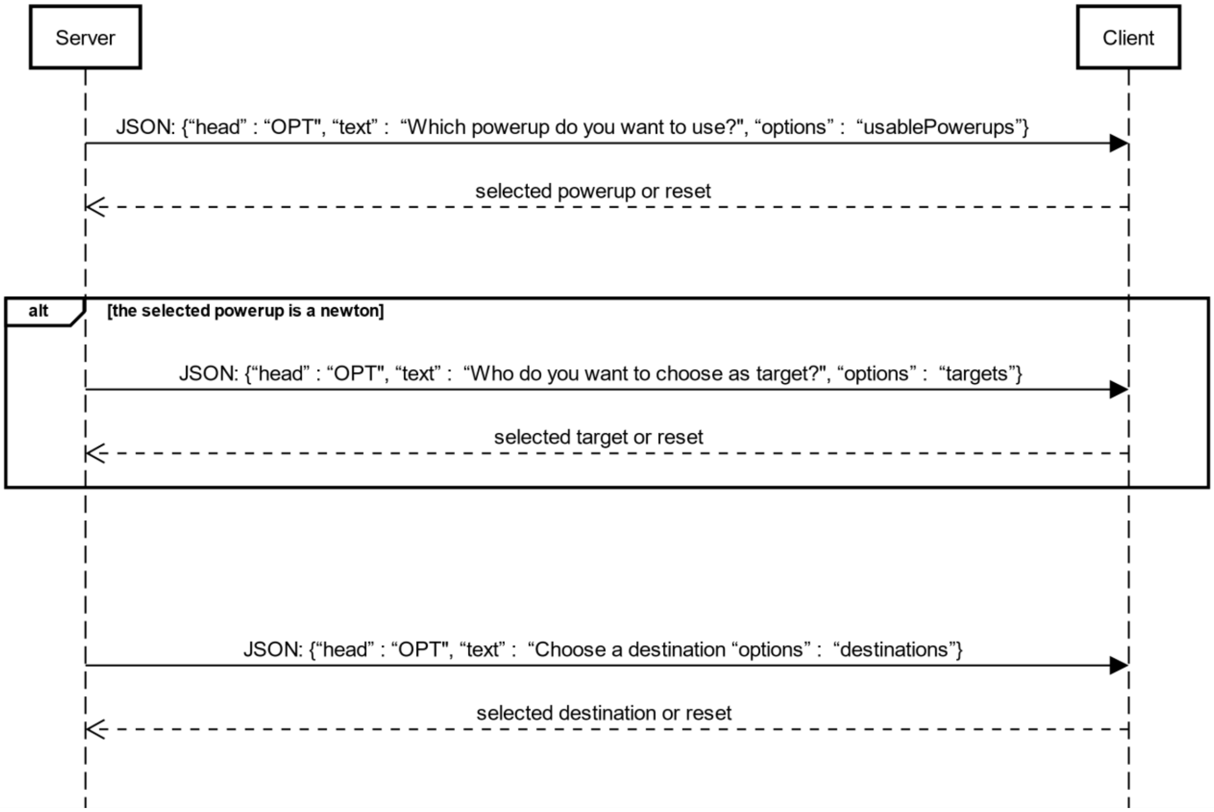
“actions” contains the actions the player can perform, including using or converting a powerup.

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

Decide whether to use a powerup


Server
Client
JSON: {“head” : “OPT", “text” : “Do you want to use a powerup?", “options” : [“yes”, “no”]}
answer

Use powerup

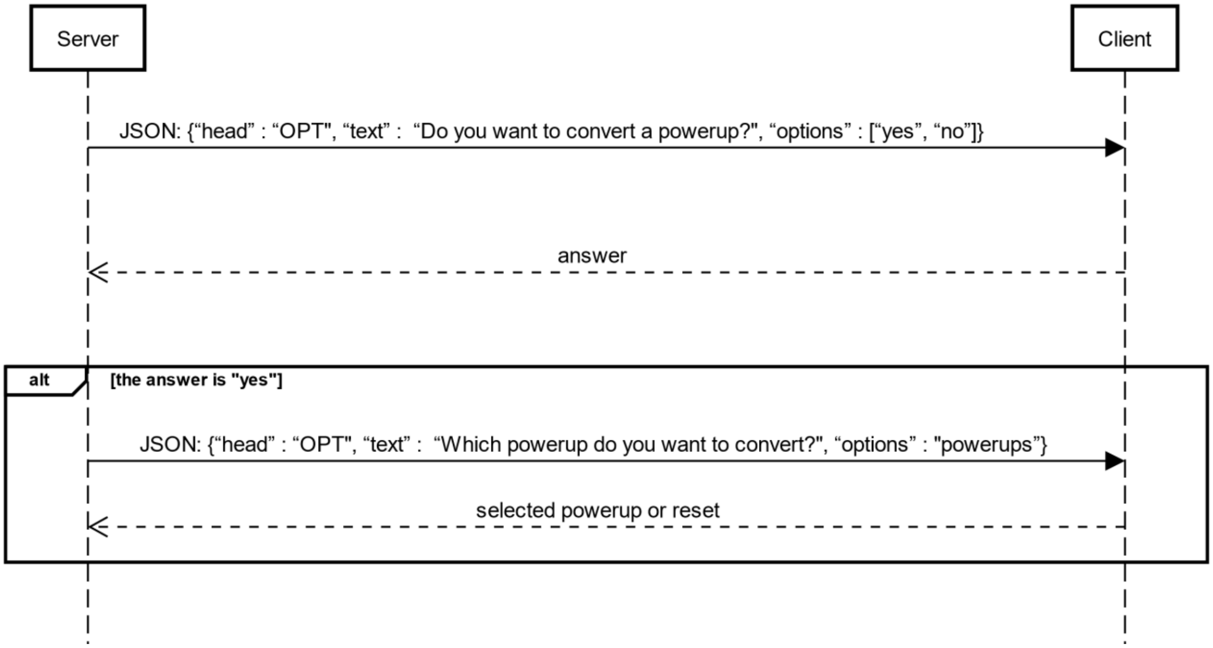

Server
Client
JSON: {“head” : “OPT", “text” : “Which powerup do you want to use?", “options” : “usablePowerups”}
selected powerup or reset
alt
[the selected powerup is a newton]
JSON: {“head” : “OPT", “text” : “Who do you want to choose as target?", “options” : “targets”}
selected target or reset
JSON: {“head” : “OPT", “text” : “Choose a destination “options” : “destinations”}
selected destination or reset

“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.

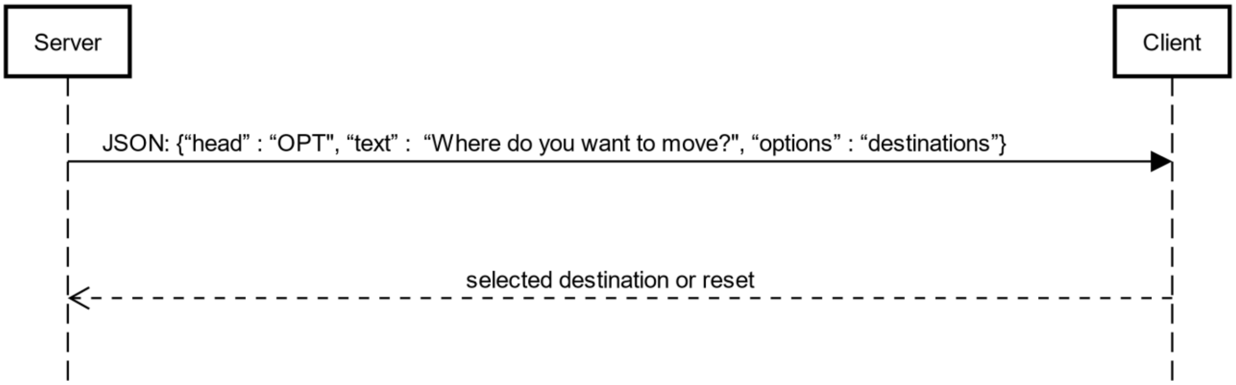
“destinations” contains the list of the squares where the player can move the target.

Convert powerup


Server
Client
JSON: {“head” : “OPT", “text” : “Do you want to convert a powerup?", “options” : [“yes”, “no”]}
answer
alt
[the answer is "yes"]
JSON: {“head” : “OPT", “text” : “Which powerup do you want to convert?", “options” : "powerups”}
selected powerup or reset

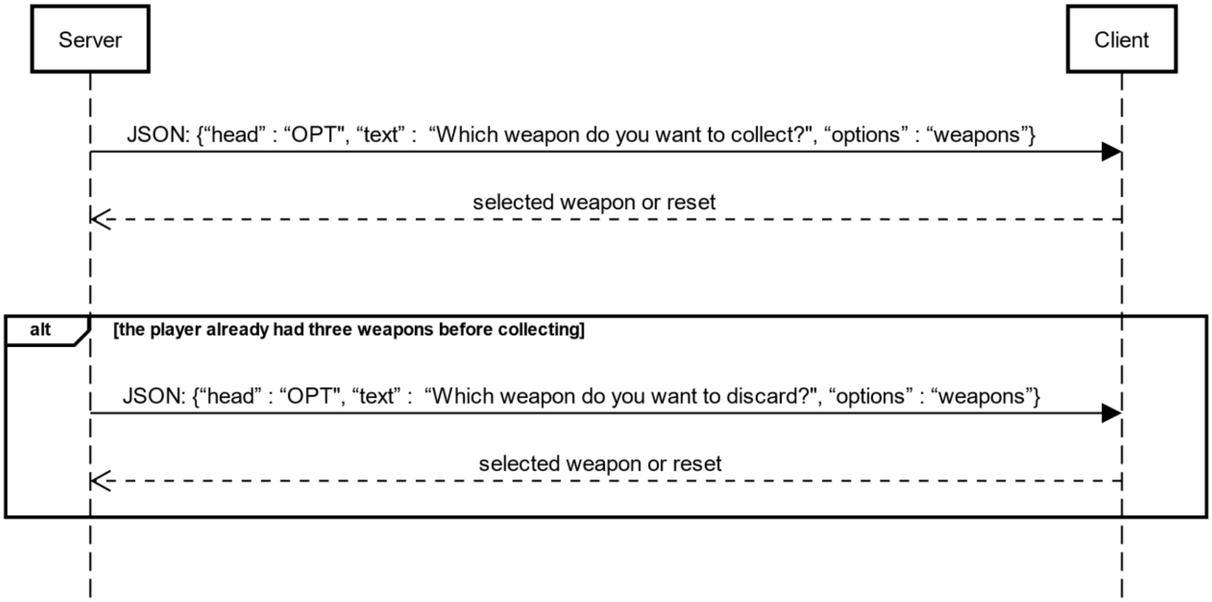
“powerups” contains the powerups held by the player.

Move


Server
Client
JSON: {“head” : “OPT", “text” : “Where do you want to move?", “options” : “destinations”}
selected destination or reset

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

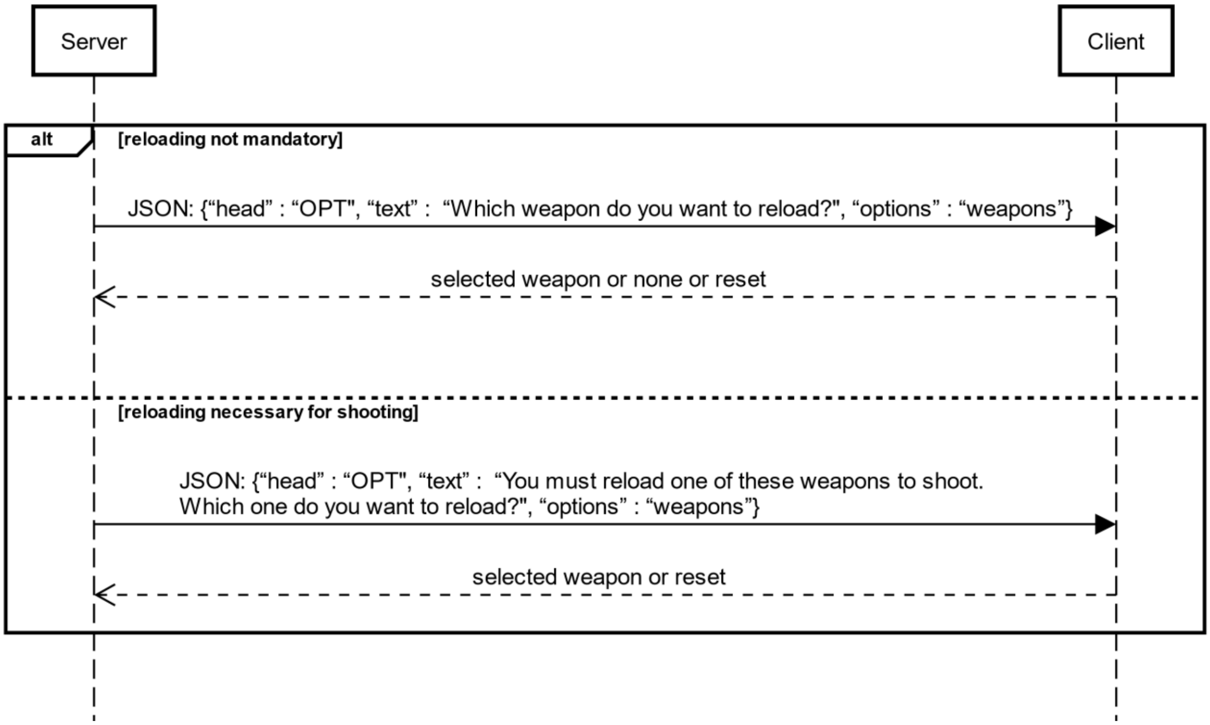
Collect Weapon


Server
Client
JSON: {“head” : “OPT", “text” : “Which weapon do you want to collect?", “options” : “weapons”}
selected weapon or reset
alt
[the player already had three weapons before collecting]
JSON: {“head” : “OPT", “text” : “Which weapon do you want to discard?", “options” : “weapons”}
selected weapon or reset

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

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

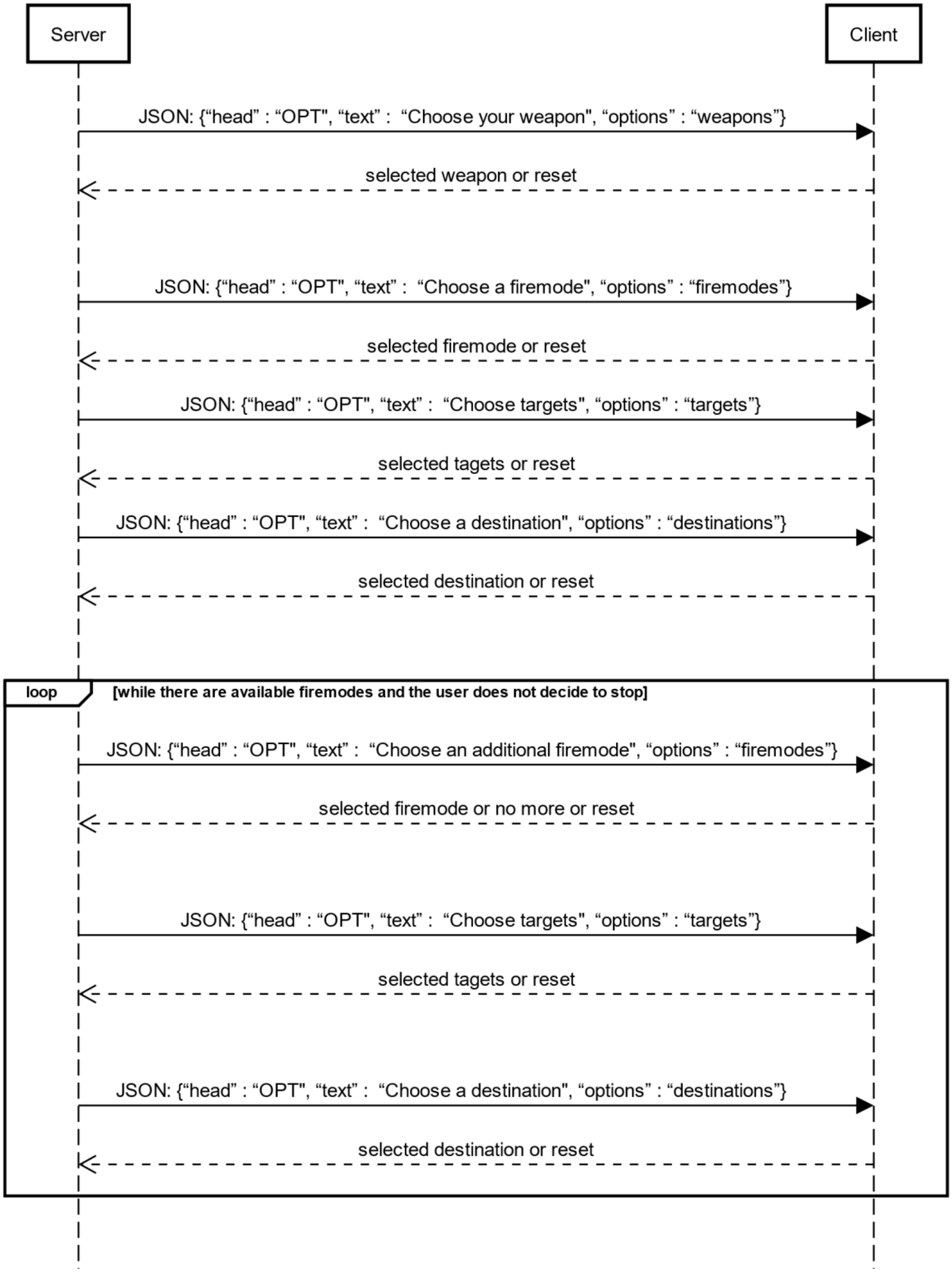
Reload


Server
Client
alt
[reloading not mandatory]
JSON: {“head” : “OPT", “text” : “Which weapon do you want to reload?", “options” : “weapons”}
selected weapon or none or reset
[reloading necessary for shooting]
JSON: {“head” : “OPT", “text” : “You must reload one of these weapons to shoot. Which one do you want to reload?", “options” : “weapons”}
selected weapon or reset

“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

Note on OPT messages

In all the OPT messages a field “type” has been added. Namely:
- “string” if the user must choose one among multiple generic strings.
- “weapon” if the user must choose one among multiple weapons.
- “powerup” if the user must choose one among multiple powerups.
- “square” if the user must choose one among multiple squares.
- “player” if the user must choose one among multiple players.

Other notes on messages above

“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)