AM38 - https://github.com/MarcoLorenzoCampo/ingswAM2021-de_luca-cristiano-campo

COMMUNICATION PROTOCOL

The communication between Client and Server works following simple concept:

ClientRequest → Server Update → Client Update

Updates are Sent when *Observers* are notified via the *NotifyAll* method. Updates are then decoded, and Messages are compiled, which are then sent to the Client.

These update messages will be called "*Model Elements*" for the rest of the protocol.

The main object exchanged between clients and Server is the abstract class **Message**, which is *Serializable* and serves as a template to be extended and customized.

There are 3 main categories of messages:

- Client Messages: Sent by the player to the Server;
- Server Messages: Sent by the Server to the Client;
- Model Messages: Controller notification messages sent by the Model.

As mentioned before, we opted for the *Serializable* interface to exchange objects between Server and Client, however, just a few Model classes implement this interface. Most of the objects are manually deserialized in the Server and re-serialized by the Client.
This allows for lighter Model elements to be sent and cached locally.

Not all objects follow the same *serialize*/*deserialize* approach, ProductionCards are parsed from JSON by the Client and the Server and serial numbers are exchanged instead of bigger objects.

(Note: Model Messages extend the Message class even if they are not exchanged via Network. We opted for this solution to be able to use the same Observer/Observable interfaces used in the Model).

Message Structure

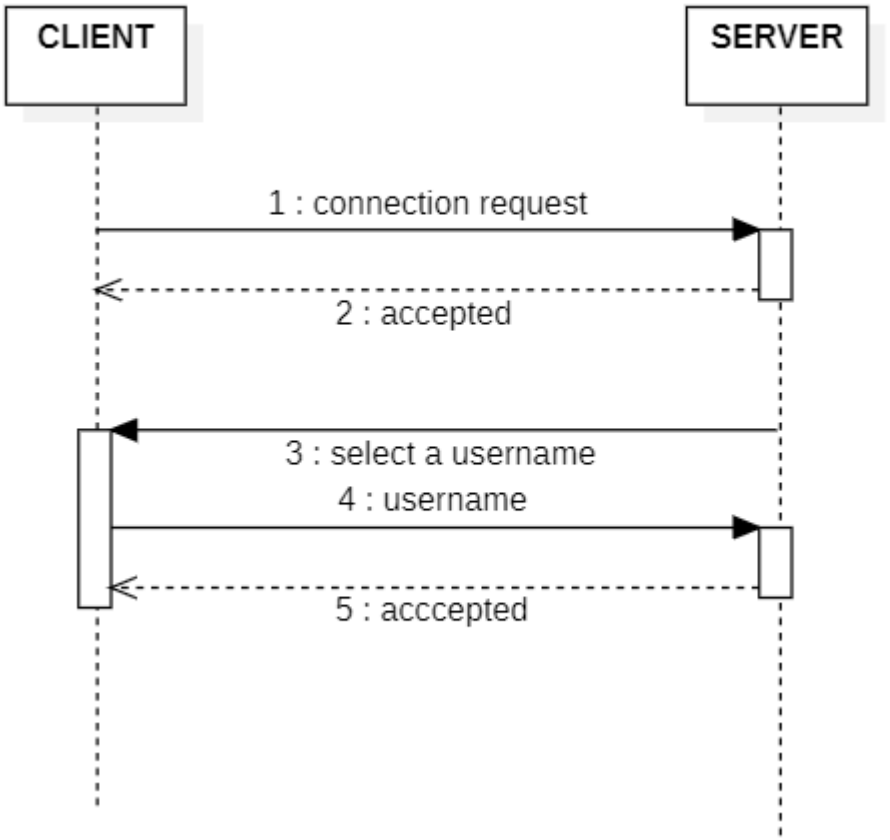
Each message has attributes and methods hereditated by the *abstract* class Message:

- SenderUsername: to identify the player who sent the message;
- SerialVersionUID: required to sent the object via network;
- MessageType: enumeration literal to identify the message.

Each sub-class then adds attributes to specify the objects that are being sent.

SETUP PHASE:

1. The client tries to connect to the server by sending a **connection request** via socket.
2. The server answers with **accepted/denied.**
3. The server then sends a **nickname request** to the Client who just got connected**.**
4. The player answers with a **nickname.**
5. The server checks if the username is valid or not
   5.1. The username is valid, the server accepts the player and sends a positive feedback.
   5.2. if the username is not valid (or already in use) the server closes the communication with the client.

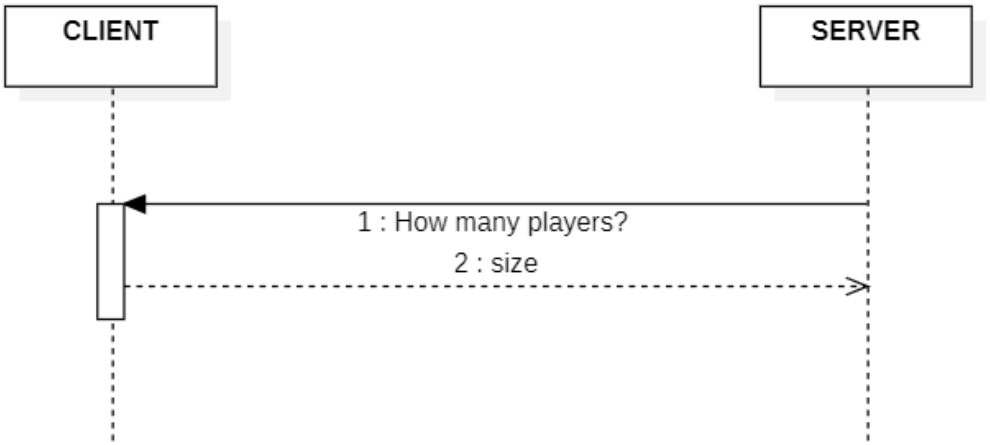


FIRST PLAYER CONNECTION:

When the first player is connected, a specific procedure starts. His name is added to the player list only after he sends a **game size** message.

1. The server asks how many players to add
2. The client responds with **size**
3. The server add the first player to the list and creates a lobby of said size

size = [size : “int of game size”]

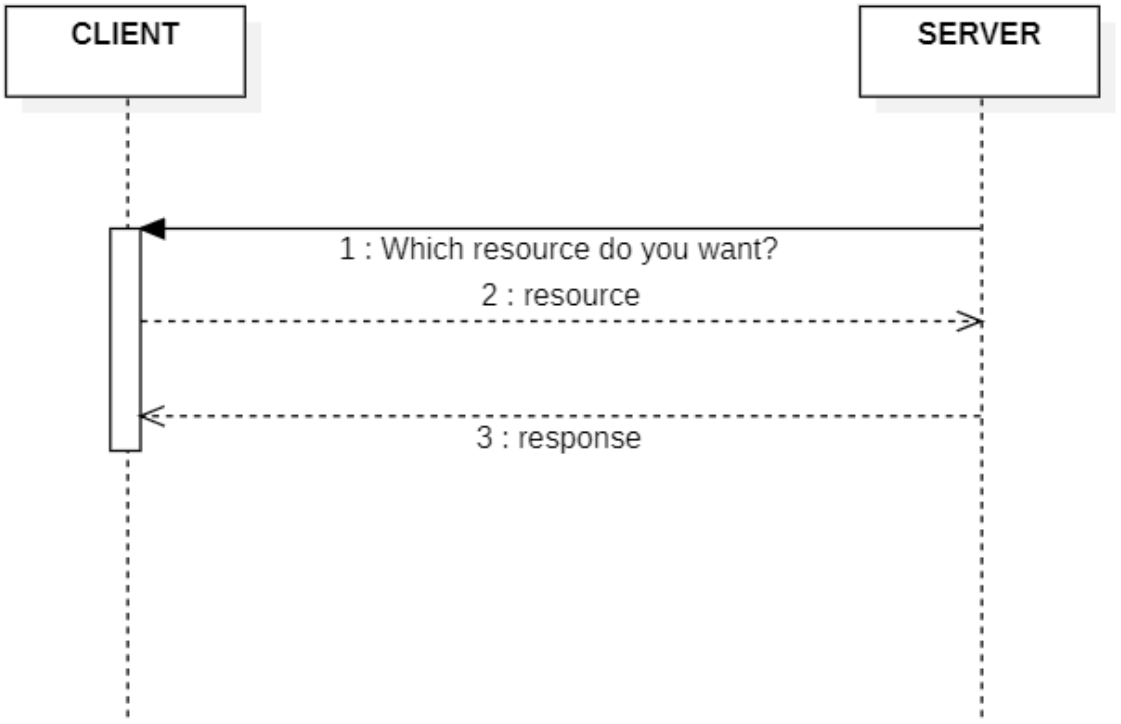


SETUP OTHER PLAYERS:

Other players are added automatically to the lobby after the login phase and, if the lobby is full,the setup phase starts by dealing the default resources.

1. For each resource the player needs to obtain the server asks **“which resource do you want?”**
2. The player responds with **resource**
3. The server either responds with **“accepted”** or **“denied”**

resource = [type: “desired resource”]

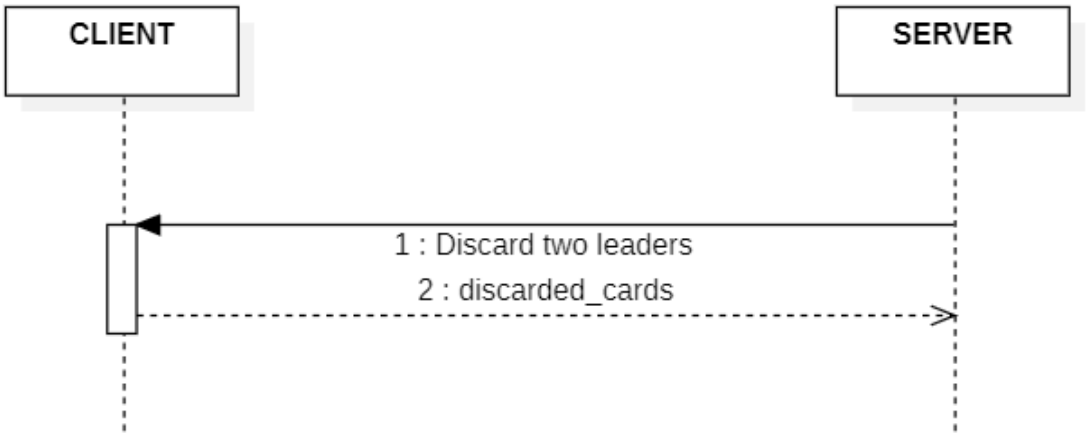


SETUP LEADER CARDS:

Each player needs to discard 2 leader cards before the game begins:

1. The server asks **“discard two leader cards”**
2. The client responds with **discarded cards**

discarded cards = [index_first: “int index of card”, index_second: “int index of card”]



PRE ACTION:

This action can be performed at any moment, it is a request to view the state of the model in order to decide what to do next.

This always happens locally. Model is cached locally and can be checked at any time.
What can be checked:

- Owned LeaderCards;
- State of the ProductionBoard;
- State of the FaithTrack;
- State of the Warehouse;
- State of the Strongbox;
- State of the MarketBoard;

An important task to manage is checking on other players. We implemented a **peek** functionality which is also performed locally by checking on cached information.

peek “name of the enemy player”
(peek Lorenzo in single player games)

Allows you to check on your enemies. What checked:

- FaithTrack position;
- State of the Inventory;
- Active LeaderCards;

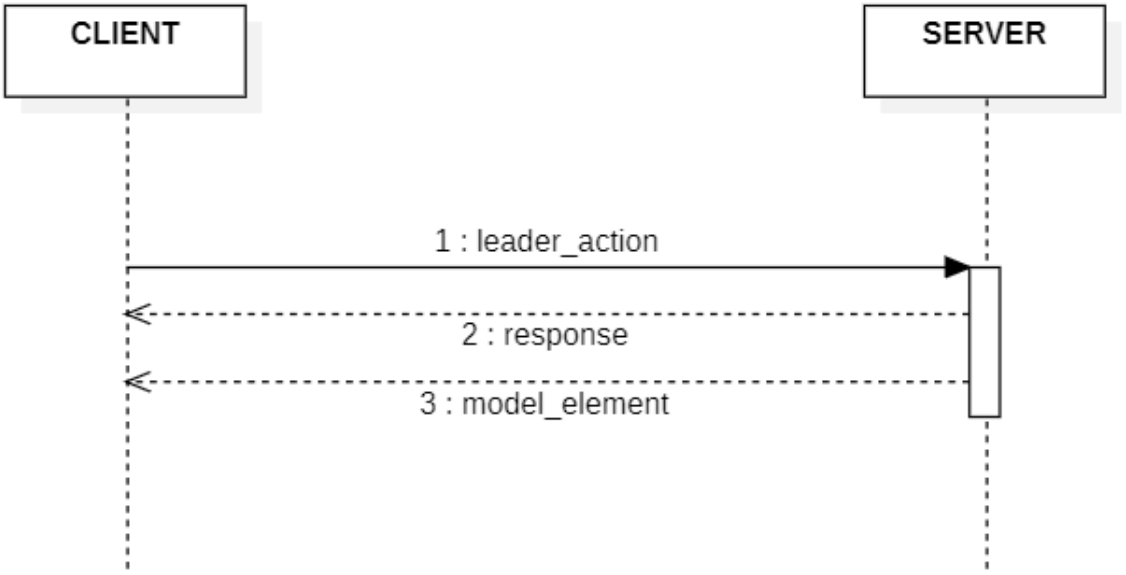
state_ request = [command:”request”, part: “model part”]

LEADER ACTION:

Leader Actions can be performed both at the beginning or end of a turn.

1. The player sends a **leader_action**
2. The server responds with **“accepted/denied”**
3. The server sends **model_element**

leader_action = [command: “activate”, index: ”int of desired card”]
leader_action = [command: “discard”, index: “int of desired card”]



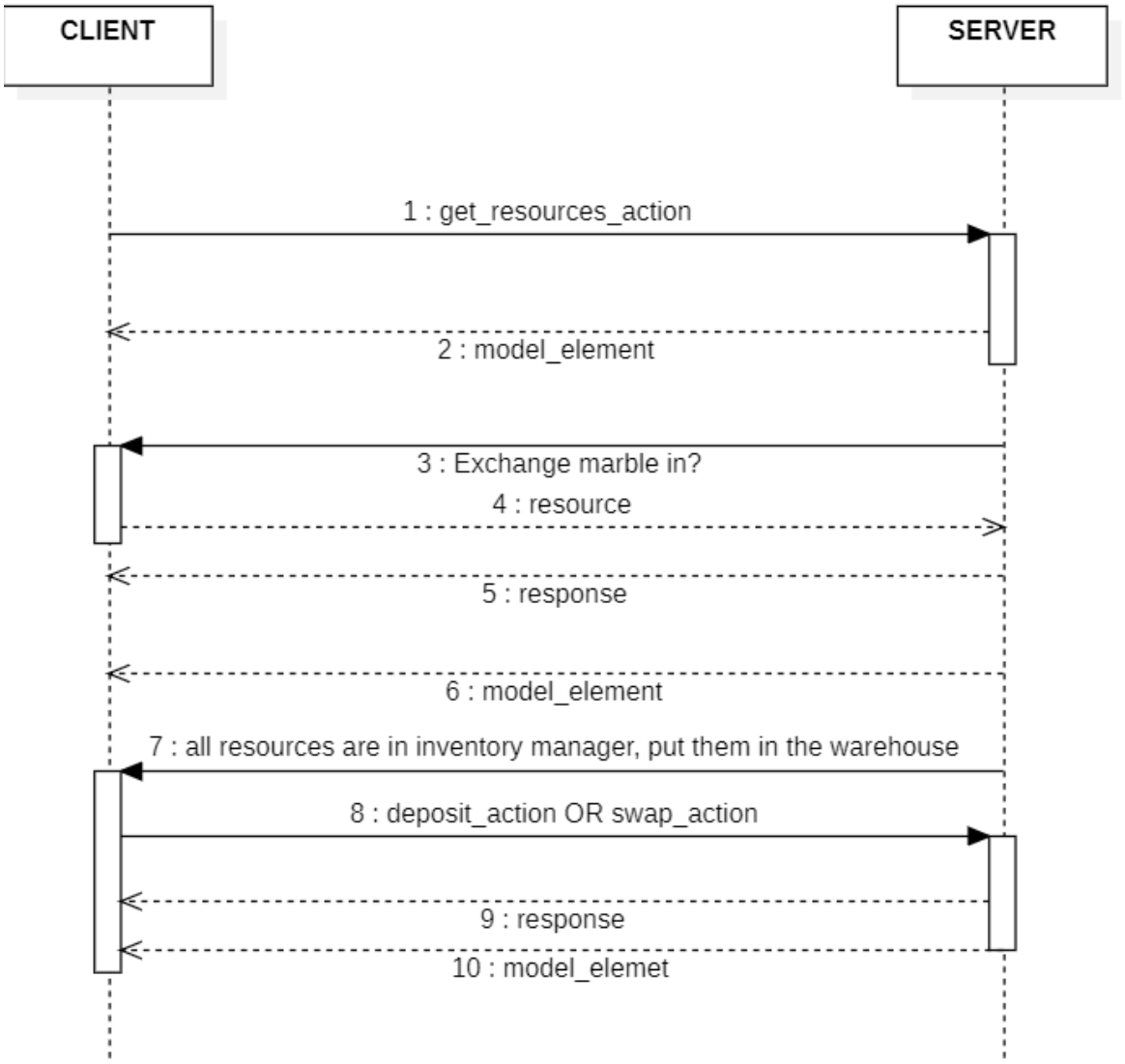
GET RESOURCES:

1. The player send a **get_resources_action** with the number of the arrow chosen
2. The server places the resources in the inventory manager buffer and then:
   2.1. if the player has no exchange power or only one it changes all the white marbles and moves on to point 6
   2.2. else it goes to point 3
3. Foreach white marble in the inventory manager the server asks **“exchange marble in?”**
4. The client sends a **“resource”**
5. The server checks if the resource is in the list of exchange powers:
   5.1. if yes, it changes the color as specified by the client and sends **“accepted”**
   5.2. if no, it sends **“denied”** and goes back to point 3
6. The server sends the changed **inventory_manager_model_element**
7. Now all the white marbles have been changed, for each resource in the inventory manager the server asks **“All resources are in inventory manager, put them in the warehouse”**
8. The client can answer with a **deposit_action**
9. The server responds:
   9.1. if swap_action sends **“accepted/denied”**
   9.2. if deposit_action sends **“placed/cannot be placed”** and if it is the latters tells all the other players to move one space forward in their faith track
10. the server sends **warehouse_model_element** and goes to point 6

get_resource_action = [command: “get resources”, index: “int of desired arrow of matrix”]

deposit_action = [command: “deposit”, index: “int of desired resource”]

resource = [command: “resource type]



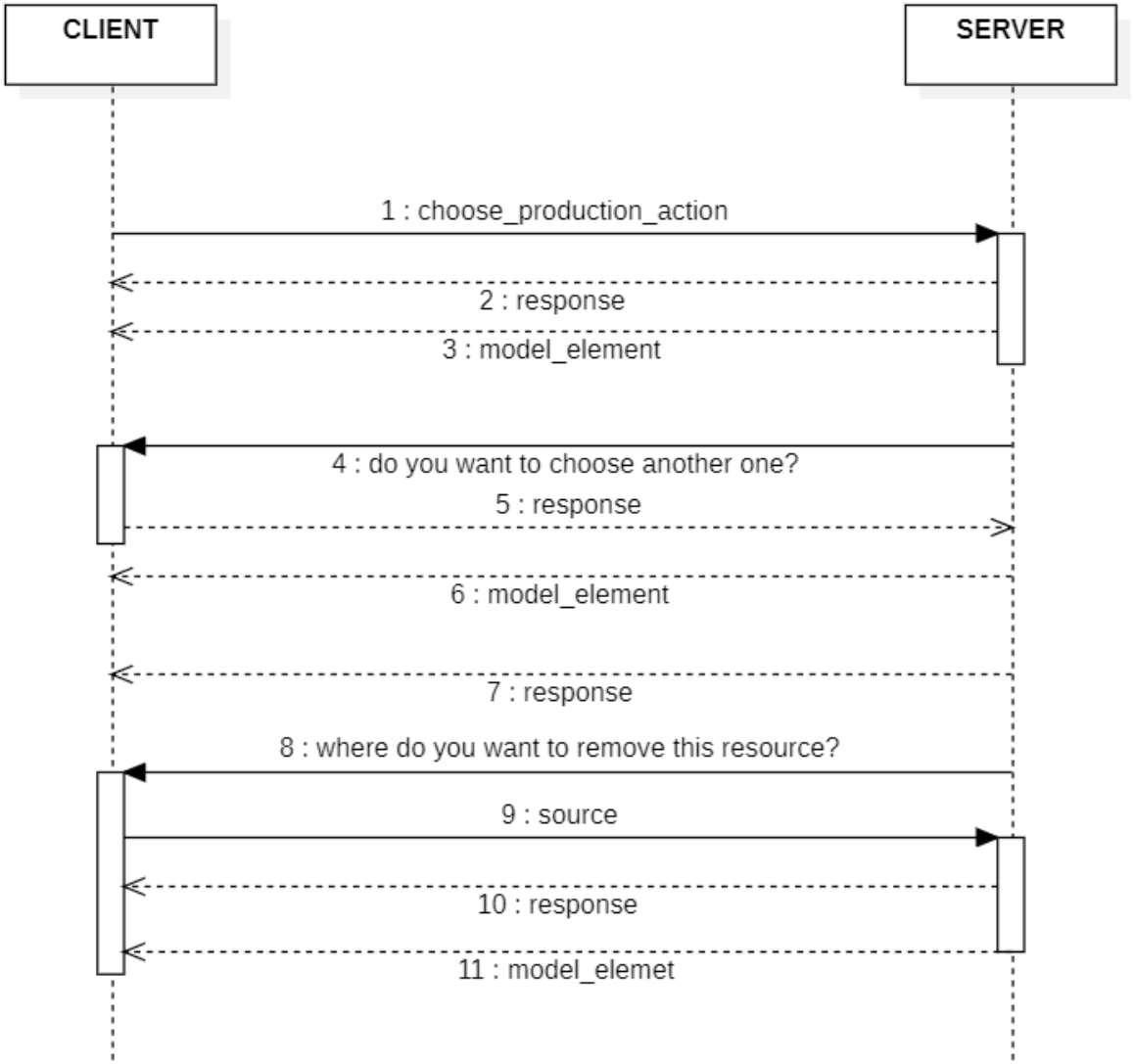
ACTIVATE PRODUCTION:

1. The player sends a **activate_production_message** with the card that they want to select
2. The server checks if the card was activated previously or not and answers:
   2.1. If it was already selected the server doesn't change the model and sends **"development card already selected"**
   2.2. if it wasn't already selected the server adds the development card to the final production and sends **"development card selected"**
3. The server sends **production_board_model_element**
4. The server then asks **"do you want to choose another one?"**
5. The player answers:
   5.1. if the answer is **YES** go back to point 1
   5.2. if the answer is **NO** the process moves one
6. The server sends **final_production_model_element**
7. The server check if the final production can be fulfilled (checks if all the requirements are met)
   7.1. if they are met sends **"accepted"** and moves on
   7.2. if not sends "denied", deletes all previous choices and goes back to point 1
8. Then, foreach resource tag in final production input, the server asks **"where do you want to remove this resource?"**
9. The client responds with **"warehouse/strongbox"**
10. The server starts removing said resource from selected source and:
    10.1. if the removal process was completely fulfilled by the specified source, the server sends **"resource removed"** and moves on to the next resource in the list
    10.2. otherwise server sends **"couldn't remove the resource, removed also from other source"** and moves on to the next resource in the list

The action ends when all inputs of the final production input list have been removed from the player's storage, then all the resources are generated, placed in the strongbox, all the selections made in the production card and the final production are erased and this phase of the player turn ends.

choose_production_action=
[command: "activate base production", resource1, resource2, resource3]
[command: "activate production", index: "int of desired production"]
[command:"activate extra production", index: "int of desired production"]

source = [source: "warehouse/strongbox"]


CLIENT
SERVER
1 : choose_production_action
2 : response
3 : model_element
4 : do you want to choose another one?
5 : response
6 : model_element
7 : response
8 : where do you want to remove this resource?
9 : source
10 : response
11 : model_elemet

BUY PRODUCTION

1. The player sends a **buy_production_message** with the card that they want to buy and the slot in which to put it
2. The server checks if the price is met and if the slot is available for said card:
   2.1. if the answer is no, the server sends **“denied”** and doesn’t fulfill the action
   2.2. if the answer is yes, server sends **“accepted”** and moves on
3. Then, foreach resource tag in the price, the server asks **“where do you want to remove this resource?”**
4. The client responds with **“warehouse/strongbox”**
5. The server starts removing said resource from selected source and:
   5.1. if the removal process was completely fulfilled by the specified source, the server sends **“resource removed”** and moves on to the next resource in the list
   5.2. otherwise server sends **“couldn’t remove the resource, removed also from other source”** and moves on to the next resource in the list

When all resources have been removed the server places the card in the designated production slot and increases the number of cards owned by the player. If the number of cards is equal to 7 then it notifies the controller that the game will end soon.

buy_production_message = [command: “buy production”, index:”int of desired card”]

source = [source = “warehouse/strongbox]