Design, Development and Deploy "Risiko!" Clone

Riccardo Squarcialupi
riccard.squarcialupi@studio.unibo.it

Filippo Benvenuti
filippo.benvenuti3@studio.unibo.it

12 December 2023

Contents

List of Figures

List of Tables

1 Introduction to "Risiko!"

1.1 Goal of the Game

Reach the Secret Goal written in your Goal Card.

1.2 Components

Game board divided in 42 part representing the entire world, each of them inside one of 6 continents.
A set of 56 Card divided in:

- 42 Territory Cards, each representing a single state in the game board with a symbol (Cavalry, Cannon, Infantry).
- 2 Jolly Card with every the 3 symbol early listed.
- 12 Goal Card with different Secret Goal(e.g., own each Card of Africa and South America), each player have one of those.

There are 6 dices:

- 3 dices used by the player who attack
- the other 3 used by the player who defend

1.3 How to play

Any match is played from 3 to 6 players.
At the start of the game Armies are distributed in function of the number of players as listed in the table below, then players put those in territories that are initially randomly owned.

Players	**Armies**
3	35
4	30
5	25
6	20

Table 1: Table Players-Armies

Every player's turn is composed of 3 phases.

1.3.1 First Phase

Players get a number of armies equal to number of territories that owns divided by 3 rounded downwards and position those where they prefer. If a player own an entire continent receive an additional army-bonus as shown in the table:

Continent	**Bonus Armies**
Oceania	2
South America	2
Africa	3
Europe	5
North America	5
Asia	7

Table 2: Table Continent-Bonus Armies

In this phase if player have tris of State Card (acquisition method are provided in next subparagraph) can get another army-bonus as shown in the table:

Card Symbols	**Bonus Armies**
3 Cannon	4
3 Infantry	6
3 Horse	8
3 State Card with different symbol	10
Jolly and 2 same symbol State Card	12

Table 3: Table Card Symbols-Bonus Armies

1.3.2 Second Phase

After placing your armies at the beginning of your turn, decide if you wish to attack at this time. The object of an attack is to capture a territory by defeating all the opposing armies already on it. The battle is fought by a roll of the dice. If you choose to attack, you must follow these rules:

- You may only attack a territory that's adjacent (touching) to one of your own, or connected to it by a dashed line.
- You must always have at least two armies in the territory you're attacking from.
- You may continue attacking one territory until you have eliminated all armies on it, or you may shift your attack from one territory to another, attacking each as often as you like and attacking as many territories as you like during one turn.

Attack First announce both the territory you're attacking and the one you're attacking from. Then roll the dice against the opponent who occupies the opposing territory.

- Before rolling, both you and your opponent must announce the number of dice you intend to roll, and you both must roll at the same time.
- You, the attacker, will roll 1,2 or 3 red dice: You must have at least one more army in your territory than the number of dice you roll.
- The defender will roll either 1 or 2 white dice: To roll 2 dice, he or she must have at least 2 armies on the territory under attack.

Decide a Battle Compare the highest dice each of you rolled. If yours (the attacker's) is higher, the defender loses one army from the territory under attack. But if the defender's die is higher than yours, you lose one army from the territory you attacked from. If each of you rolled more than one die, now compare the two next-highest dice and repeat the process. In case of a tie, the defender always wins. The attacker can never lose more than 2 armies on single roll.

1.3.3 Third Phase

In each turn players can perform a single "Strategic Move" consist of moving armies from 2 contiguous states.

1.3.4 Earning Cards

At the end of any turn in which you have captured at least one territory, you will earn one (and only one) State card.

2 Goal/Requirements

The main goal of this project is developing a distributed version of the classic game RiSiKo!.
In particular, we'll see two entities: the player (the person who actually play) referred now on as the client and the host (responsible for the match making) referred now on as the server. We'll be using both client-server and peer-to-peer connections in order to get a scalable system, minimizing load on the server.

2.1 Scenarios

Each client creates a connection with the server asking for all the lobbies available to be joined, the server retrieves information about lobbies and returns them to the client who asked for it, now the client can choose to connect to one of the available lobbies or to create a new one becoming the manager of it.
Now the client will be expected to wait other players joining the lobby until it's full, only when this condition is met the manager can decide to start the game. The interface displayed to all players will allow them to place troops in the territories they own, once done standard rounds will start and repeat until one of the players reach the goal in his secret card.
A standard round is formed by three-phase described in "Introduction to "Risiko!"".
Whenever a client take an action of any type, the other clients will be warned to maintain consistency.

2.2 Self-assessment policy

- How should the *quality* of the *produced software* be assessed?

  The quality of the produced software can be assessed through the analysis of the non-functional requirements 3.2, which can be found in 7.2.

- How should the *effectiveness* of the project outcomes be assessed?

  The effectiveness of the project outcomes can be assessed through the analysis of the functional requirements 3.1, like our case in can be done with unit tests, explained in 7.1.

3 Requirements Analysis

3.1 Functional Requirements

The System will give the user the possibility of:

- Create a lobby.
- Find suitable lobbies.
- Join a lobby.
- Exit a lobby.
- Change manager of the lobby.
- Start the match if possible.
- Make actions in game as described in "Introduction to "Risiko!"".
- Exit a running game.

3.2 Non Functional Requirements

- System must be maintained easily.
- System must be extended easily.
- System must manage client connection/disconnection in order to prevent unexpected case.
- System must be scalable.

3.3 Paradigm / Technology

For the implementation will be used both client-server and peer-to-peer connections architecture, will also be used libraries and framework such:

- **Vert.X** for peer-to-peer connection of clients.
- **Flask** REST API for client-server communication.
- **Gradle** for build automation of client.
- **Docker** for deployment and testing.
- **Jackson** for binding JSON content into Java Objects.
- **Guava** for use hash functions and to operate on HashCode objects.
- **X11/Xming** Graphic Handler used with Docker.

4 Utilized Technology

4.1 Communication and Connections

4.1.1 Flask REST

Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools.
The following code shows a simple web application that displays "Hello World!" when visited:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello() -> str:
    return "Hello World"


if __name__ == "__main__":
    app.run()
```

Flask-REST is an extension for Flask that adds support for quickly building REST APIs. It is a lightweight abstraction that works with your existing ORM/libraries. Flask-RESTful encourages best practices with minimal setup. If you are familiar with Flask, Flask-RESTful should be easy to pick up.

4.1.2 Vert.x

Vert.x is a toolkit that enables creation of reactive applications for the JVM, by definition a reactive application is responsive and keep a low latency optimizing usage of resources in the system.
Vert.x is a various ecosystem of modules such as: Web APIs, databases, messaging, event streams, cloud, registries, security and so on, our project take advantage of it's core, the Web API and the Web Client API.
Vert.x is based on concurrent and asynchronous programming, it hides the complexity related to it offering a user friendly interface based on an event-loop paradigm which can be used to fast and safely program scalable and efficient applications, the only shrewdness is to avoid slow or blocking code inside callbacks of events.
Here follows key concept of Vert.x:

Verticles
A verticle is the fundamental processing unit in Vert.x. The role of a verticle is to encapsulate a technical functional unit for processing events such as exposing an HTTP API and responding to requests, providing a repository interface on top of a database, or issuing requests to a third-party system.
Verticles have a private state that may be updated when receiving events, they can deploy other verticles, and they can communicate via message-passing.

Event Bus
A Vert.x event-bus is a light-weight distributed messaging system which allows different parts of your application, or different applications and services to communicate with each in a loosely coupled way. An event-bus supports publish-subscribe messaging, point-to-point messaging and request-response messaging. Message delivery is best-effort and messages can be lost if failure of all or part of the event bus occurs.

Router
A router receives request from an Http Server and routes it to the first matching Route that it contains. A router can contain many routes. A route is a holder for a set of criteria which determine whether an HTTP request or failure should be routed to a handler. Finally the handler is an interface to be implemented to handle specific events.

4.1.3 Jackson

In computing, Jackson is a high-performance JSON processor for Java. Its developers extol the combination of fast, correct, lightweight, and ergonomic attributes of the library. Jackson provides multiple approaches to working with JSON, including using binding annotations on POJO classes for simple use cases.
Sample code for reading and writing with POJOs may look like the following:

```
public class ReadWriteJackson {
    public static void main(String[] args) throws IOException {
        ObjectMapper mapper = new ObjectMapper();

        String jsonInput = "{\"id\":0,\"firstName\":\"Robin\",\"lastName\":\"Wilson\"}";
        Person q = mapper.readValue(jsonInput, Person.class);
        System.out.println("Read and parsed Person from JSON: " + q);

        Person p = new Person("Roger", "Rabbit");
        System.out.print("Person object " + p + " as JSON = ");
        mapper.writeValue(System.out, p);
    }
}
```

In our project, Jackson was used to serialize and deserialize classes from and to JSON, in order to be able to make clients communicate with our objects.

4.1.4 Guava

Google Guava can be roughly divided into three components: basic utilities to reduce manual labor to implement common methods and behaviors, an extension to the Java collections framework (JCF) formerly called the Google Collections Library, and other utilities which provide convenient and productive features such as functional programming, graphs, caching, range objects, and hashing.
The creation and architecture of the collection component were partly motivated by generics introduced in JDK 1.5. Although generics improve the productivity of programmers, the standard JCF does not provide sufficient functionality, and its complement Apache Commons Collections had not adopted generics in order to maintain backward compatibility.[1] This fact led two engineers Kevin Bourrillion and Jared Levy to develop an extension to JCF, which provides additional generic classes such as multisets, multimaps, bitmaps, and immutable collections.
The library's design and code were advised and reviewed by Joshua Bloch, the original lead designer of the Java Collections framework, and Doug Lea, one of the lead designers of concurrency utilities in JDK.
In our project, Guava was used for his hash functions, to easily implement one with a private key, used for dice throwing.

4.1.5 X11/Xming

The X Window System (X11 for Linx Xming for Windows) is a windowing system for bitmap displays, common on Unix-like operating systems.
X provides the basic framework for a GUI environment: drawing and moving windows on the display device and interacting with a mouse and keyboard. X does not mandate the user interface – this is handled by individual programs. As such, the visual styling of X-based environments varies greatly; different programs may present radically different interfaces.
X is an architecture-independent system for remote graphical user interfaces and input device capabilities. Each person using a networked terminal has the ability to interact with the display with any type of user input device.
X provides the basic framework, or primitives, for building such GUI environments: drawing and moving windows on the display and interacting with a mouse, keyboard or touchscreen. X does not mandate the user interface; individual client programs handle this. Programs may use X's graphical abilities with no user interface. As such, the visual styling of X-based environments varies greatly; different programs may present radically different interfaces.
X's network protocol is based on X command primitives. This approach allows both 2D and (through extensions like GLX) 3D operations by an X client application which might be running on a different computer to still be fully accelerated on the X server's display. For example, in classic OpenGL (before version 3.0), display lists containing large numbers of objects could be constructed and stored entirely in the X server by a remote X client program, and each then rendered by sending a single glCallList(which)

across the network. In our project, X11 was used to be able to see a gui through the usage of docker, which natively doesn't support gui for windows.

4.2 Build Automation and Deployment

4.2.1 Gradle

Gradle is a open-source build automation tool for multi-language software development. It controls the development process in the tasks of compilation and packaging to testing, deployment, and publishing.
Gradle is designed for multi-project builds, which can grow to be large. It operates based on a series of build tasks that can run serially or in parallel. Incremental builds are supported by determining the parts of the build tree that are already up to date; any task dependent only on those parts does not need to be re-executed.
A Gradle project is based on a specific structure of the folders, a script named gradlew (gradle wrapper) rappresent the way to standardize the gradle version of the project with the purpose of having a more reliable and strong build, and from a build.gradle.task file determines the dipendency and the task of the application.

4.2.2 Docker

Today Docker is the standard reference of containers for creating and sharing deliverable app.
Containers are a type of software that allow to isolate an app and his environment, make easier to install on a PC with a Container Engine installed.
There are 4 main concept which Docker is based:

- Dockerfile: configuration file describe every needed command for create a new image.
- Docker Images: executable file representing the static description of a container, running the image instantiate the container itself.
- Docker Container: a 'lightweight' virtual machine that share the kernel with the host machine, granted the isolation from the environment and from the other container.
- Docker Compose: tool allow the definition of applications with more services and, on run-time, manage the totality of the life-cycle of the multi-container app created. The definition of this type of application is possible by a YAML file, also named compose-file.

5 Design

One of the first choice we had gone through was the design architecture, evaluating which communication system was the most suitable for our project. Two idea came out, a full Monolithic Client-Server architecture or a full Peer-to-Peer. The first one present some problems such as an excessive overload of computations in the server, making the solution not scalable nor deliverable. The second one has problems relative on communications initialization and difficulties to program the match making, so why not merging the two solutions and get the benefits of both?
We use Client-Server paradigm only for the matchmaking part, in this way it's possible to have multiple independent servers to interrogate which makes the system scalable. Follows the communication schema:

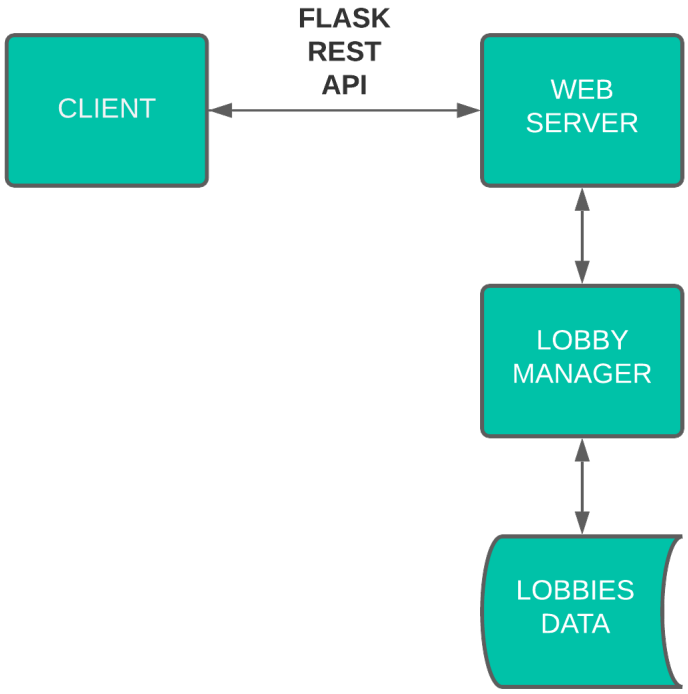


Figure 1: Client-Server High-Level Architecture

As in Figure 1 Client communicate with Server through Flask REST API and can create or join lobbies that are stored by the lobby manager (no DBs are needed).
Peer-to-Peer paradigm is used for each lobby and game management, so it can run independently of the others and from the server.

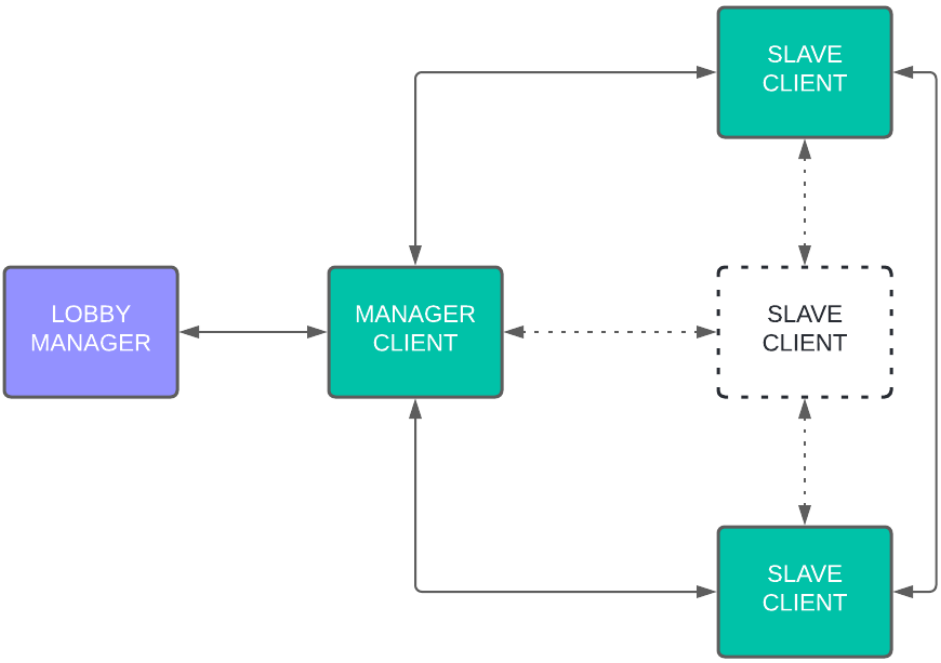


Figure 2: P2P High-Level Architecture

In Figure 2 we have a view of a lobby where clients transmit information to the other for each action they take. The Manager Client has a lobby manager to direct the lobby (Start the game, accept new client into lobby, close the lobby...), a lobby always has at least the manager and two or more slave (maximum 5), in particular for this lobby we can see 3 clients waiting for the fourth.

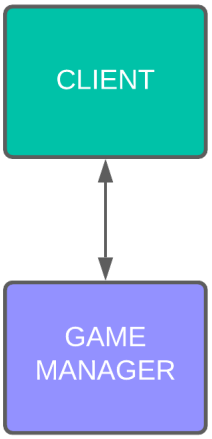


Figure 3: Client High-Level Architecture

Figure 3 is the client subsection of Figure 2 after game is started, in particular each client(manager or slave) has a game manager that process information in input from the other players, update the interface of the user keeping him up-to-date and broadcast every action of him to the others in the same lobby.

5.1 Structure

5.1.1 Class Diagram

In this section are described the main entities used to develop Server, Client in Lobby and in Game.

Figure 4 report the diagram class of the lobbies temporary saved on the server. A server communicate with multiple Clients (the Risiko player, both LobbyClient and Manager-Client) which can enter in different Server-side lobbies(ServerLobby class). The Server methods corresponds to the API of the server; server stores all the lobbies' information in ServerLobby objects.

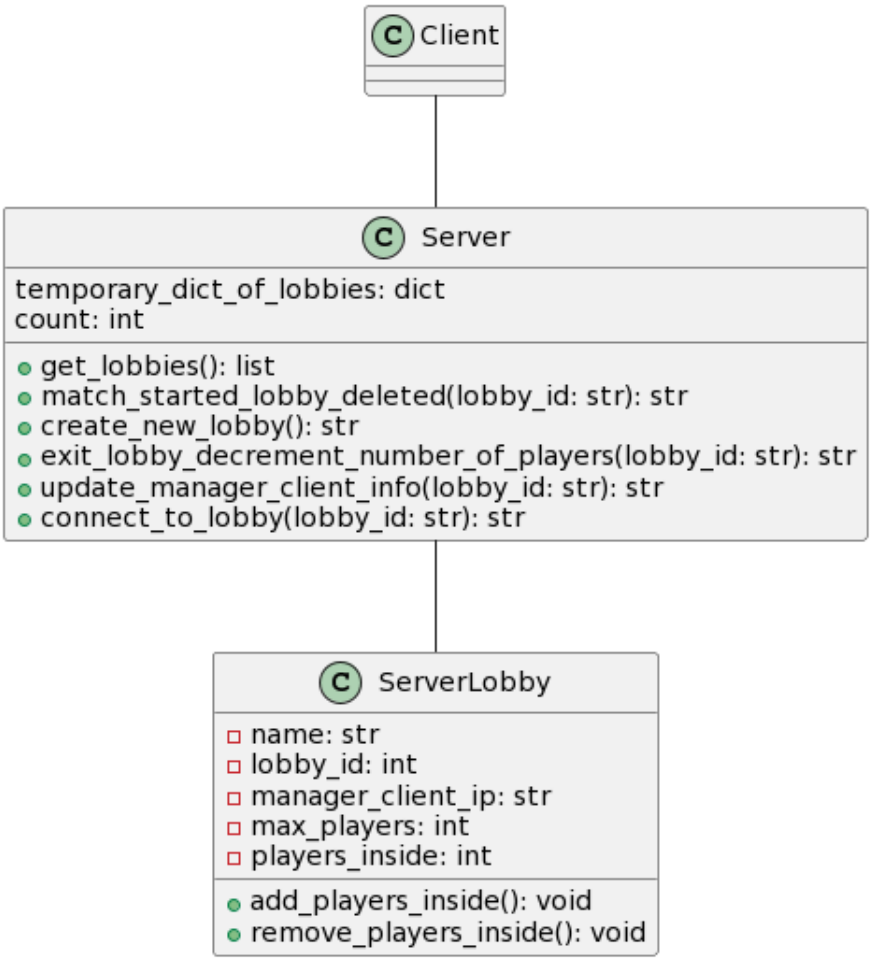


Figure 4: Server Part Class Diagram

Figure 5 report the class diagram of client inside a lobby to describe its entities, each LobbyClient(a Risiko player) is a Client who is part of a specific lobby, one of them must

be a ManagerClient(i.e., the Client who create the lobby) in charge of update server for every event happened in the lobby and welcomes new entries until game is started.

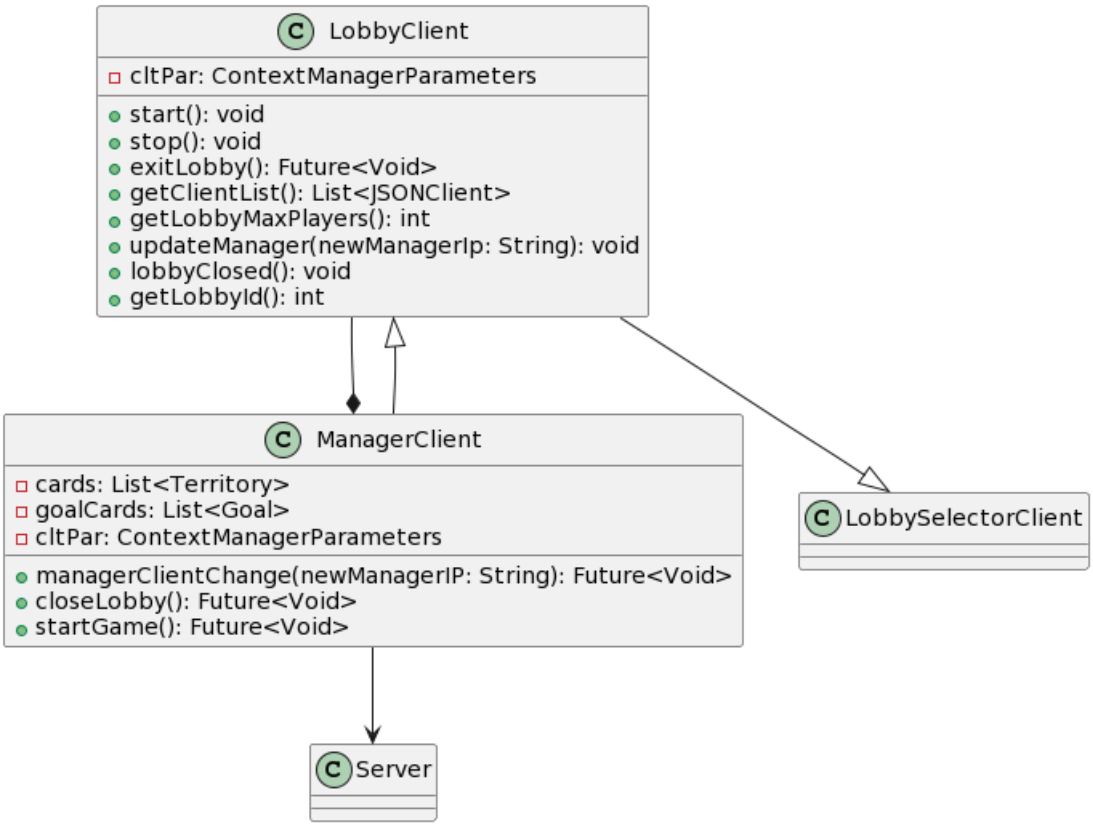


Figure 5: Client in Lobby Part Class Diagram

Figure 6 diagram illustrates the relationships and interactions between different classes and enums in the GameClient class during a match.

- **GameClient**: The client typology during a game, it stores his information inside ContextManagerParameters.
  - **Attributes**:
    - **cltPar**: An instance of the ContextManagerParameters class.
    - **myTurn**: A boolean indicating whether it is the client's turn.
  - **Methods**: Various methods for game actions such as checking for the client's turn, updating enemy territories, handling bonuses, managing connections, placing armies, throwing dice, sending attack and defend messages, handling player leaving events, and using bonus cards.

- **ContextManagerParameters**: Stores all the information needed during a game.
  - **Attributes**:
    * **ip**: The IP address of the client.
    * **clientList**: A list of clients represented by JSONClient instances.
    * **nickname**: The client's nickname.
    * **idLobby**: An identifier for the game lobby.
    * **maxPlayer**: The maximum number of players allowed.
    * **allTerritories**: A mapping of pairs of clients and territories to integer values.
    * **goalCard**: An instance of the Goal enum representing the client's goal.
    * **myBonusCards**: A list of bonus cards represented by instances of the CardType enum.
- **CardType Enum**: Enumerates types of cards:
  - **INFANTRY**.
  - **CAVALRY**.
  - **ARTILLERY**.
  - **JOLLY**.
- **Goal Enum**: Enumerates various game goals, each represented by a specific string description. Includes goals such as conquering territories, destroying players, and continent-specific goals.
- **Territory Enum**: Enumerates game territories, each represented by a specific name. Includes methods to retrieve neighbors, continent, name, and type (represented by the CardType enum).
- **Continent Enum**: Enumerates continents, including NORTH_AMERICA, SOUTH_AMERICA, EUROPE, AFRICA, ASIA, and OCEANIA.

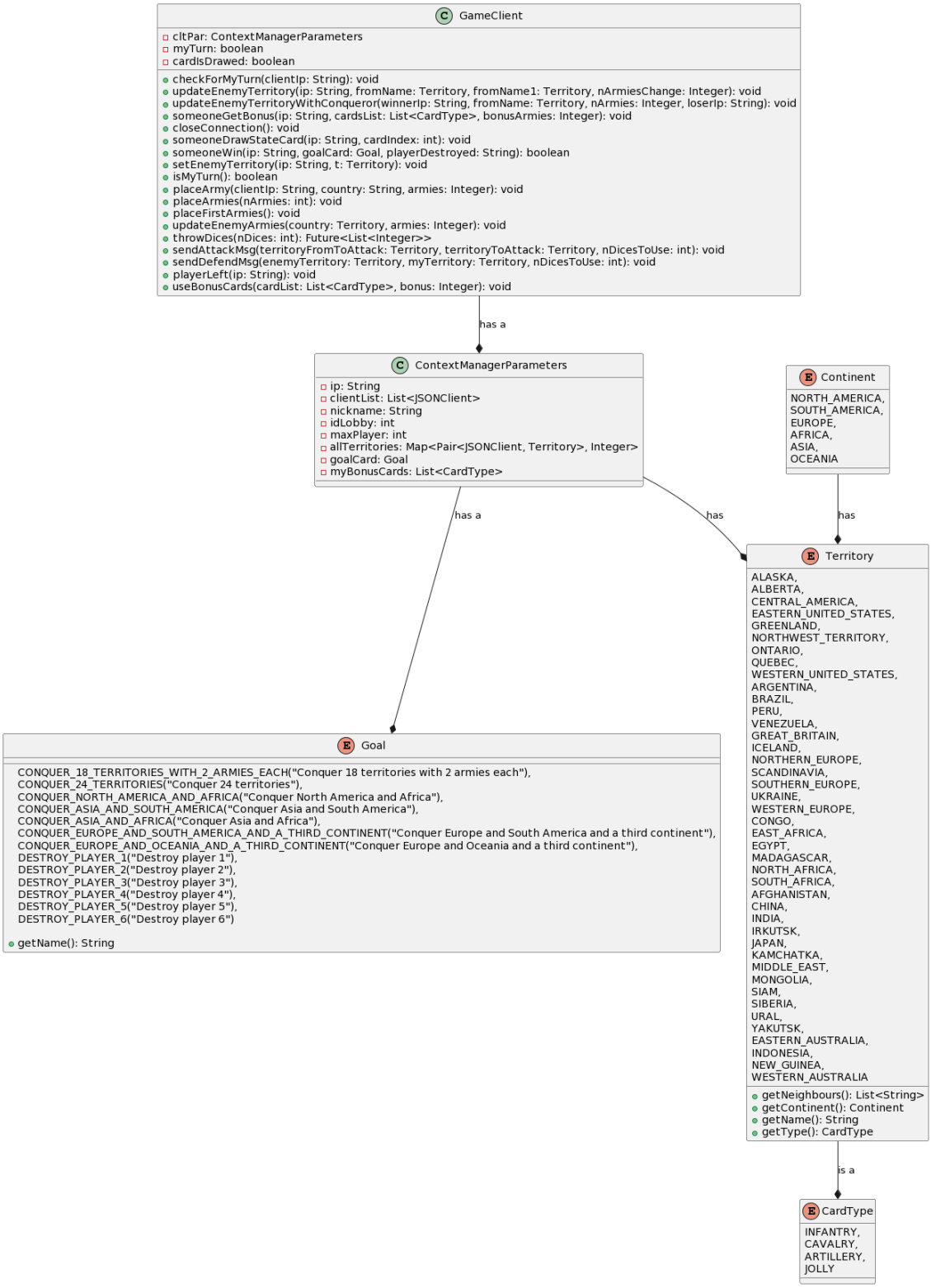


Figure 6: Client in Game Class Diagram

5.2 API Specifications

We can divide API in 3 sub-part each one linked to the previous class diagram.

- Client-Server API Server-side Lobbies
- Peer-to-Peer API Client-side Lobby
- Peer-to-Peer API Client-side Game

Each working independently of the other.

5.2.1 Client-Server API Server-side Lobbies

This API is used by the Server and expose:

Get Lobbies:

- Method: GET
- URL: server/lobbies
- Expected Handle Error:
  - 500: Server Down.

Create a new Lobby:

- Method: POST
- URL: server/lobbies
- Body: {name, max_players}
- Expected Handle Error:
  - 500: Server Down.
  - 400: Incongruous Lobby provided.

Tell the server the match started/lobby closed (remove the lobby):

- Method: DELETE
- URL: server/lobby/{id}
- Expected Handle Error:
  - 500: Server Down.
  - 404: Lobby not found.
  - 403: Lobby id is not a number.

Exit Lobby(decrement number of players):

- Method: DELETE
- URL: server/lobby/{id}/numberOfPlayer
- Expected Handle Error:
  - 500: Server Down.
  - 404: Lobby not found.
  - 403: Lobby id is not a number.

Update manager client Information from the selected Lobby:

- Method: PUT
- URL: server/lobby/{id}/managerClientIp
- Body: {new_manager_ip}
- Expected Handle Error:
  - 500: Server Down.
  - 404: Lobby not found.
  - 400: New manager not set.

Connect to a Lobby:

- Method: PUT
- URL: server/lobby/{id}
- Expected Handle Error:
  - 500: Server Down.
  - 404: Lobby not found.
  - 403: Lobby id is not a number.

5.2.2 Peer-to-Peer API Client-side Lobby

This API is used by the Client for the lobby part and expose:

Client join a lobby:

- Method: POST
- URL: client/lobby/clients
- Body: {Client}
- Expected Handle Error:
  - 500: Manager client is down.
  - 405: Lobby is full.

Client exit a lobby:

- Method: DELETE
- URL: client/lobby/clients
- Expected Handle Error:
  - 500: Client Down.

ManagerClient change:

- Method: PUT
- URL: client/lobby/manager
- Body: {new_manager_ip}
- Expected Handle Error:
  - 500: LobbyClient Down.

Game has started:

- Method: PUT
- URL: client/lobby/game
- Body: "{list[territory], goal, list[card]}"
- Expected Handle Error:
  - 500: LobbyClient Down.

Receive broadcast ip on new client join:

- Method: POST
- URL: client/lobby/clients
- Body: {Client}
- Expected Handle Error:
  - 500: Client Down.

5.2.3 Peer-to-Peer API Client-side Game

This API is used by the Client for the game part and expose:

Receive broadcast territories of another client:

- Method: PUT
- URL: client/game/territories
- Body: {client_ip, list[territory]}
- Expected Handle Error:
  - 500: Client is down.

Get notify when a territory is conquered:

- Method: PUT
- URL: client/game/territory/armiesWithConqueror
- Body: {client_ip, territory, armies, client_loser}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify a client has finished his turn.

- Method: PUT
- URL: client/game/turn/finish
- Body: {client_ip}
- Expected Handle Error:
  - 500: Client is down.

Get notify when bonus from State Card have been used.

- Method: PUT
- URL: client/game/card
- Body: {client_ip, list_card, nArmiesBonus}
- Expected Handle Error:
  - 500: LobbyClient Down.

Lobby close:

- Method: PUT
- URL: client/lobby
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify if an offense move is starting:

- Method: PUT
- URL: client/game/battle/offense
- Body: {ip_client_attack, id_client_defend, list[diceATKResult], offensive_terriotry, defensive_territory}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify of the defense response:

- Method: PUT
- URL: client/game/battle/defense
- Body: {ip_client_attack, id_client_defend, list[diceATKResult], enemy_terriotry, my_territory}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify when someone draws a State Card:

- Method: PUT
- URL: client/game/stateCard
- Body: {ip_client, card_index}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify if a Client has won the game.

- Method: PUT
- URL: client/game/win
- Body: {ip_client, goal_card, opt(nickname)}
- Expected Handle Error:
  - 500: LobbyClient Down.
  - 403: Client didn't win.

Send Territories at the start of the game (from Manager Client to other clients).

- Method: PUT
- URL: client/game/territories
- Body: {idClient}{listTerritories}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get a state card from the Manager (Mandatory: client must have conquer a territory in his round during an attack).

- Method: GET
- URL: manager/game/territory
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify a client wants to throw a dice:

- Method: POST
- URL: client/game/dice/throw
- Body: {ip_client, dice_hash}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get dice share from other clients:

- Method: PUT
- URL: client/game/dice/share
- Body: {ip_client, random_dice}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get random turns order:

- Method: PUT
- URL: client/game/order
- Body: {ip_client, list[client]}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify when armies number change:

- Method: PUT
- URL: client/game/territory/armies
- Body: {client, territory, receiving_territory, armies}
- Expected Handle Error:
  - 500: LobbyClient Down.

Receive dice confirm intention:

- Method: PUT
- URL: client/game/dice/confirm
- Body: {client_ip, random_dice, private_key, sum_dice}
- Expected Handle Error:
  - 500: LobbyClient Down.
  - 403: Dice throw is not eligible.

Get broadcast when army is placed:

- Method: PUT
- URL: client/game/armies/update
- Body: {territory, armies}
- Expected Handle Error:
  - 500: LobbyClient Down.

Get notify when a player has left:

- Method: PUT
- URL: client/game/player
- Body: {ip_client}
- Expected Handle Error:
  - 500: LobbyClient Down.

5.3 Behaviour

5.3.1 WebServer Behaviour

When the WebServer start, it remains in waiting mode until a Client want some information or decide to create/join a lobby. Using Flask we create a web service listening on 5000 port, then we create callbacks for routes that he can serve (the same defined in 5.2.1. We decide to use API approach to provide information for the Client.

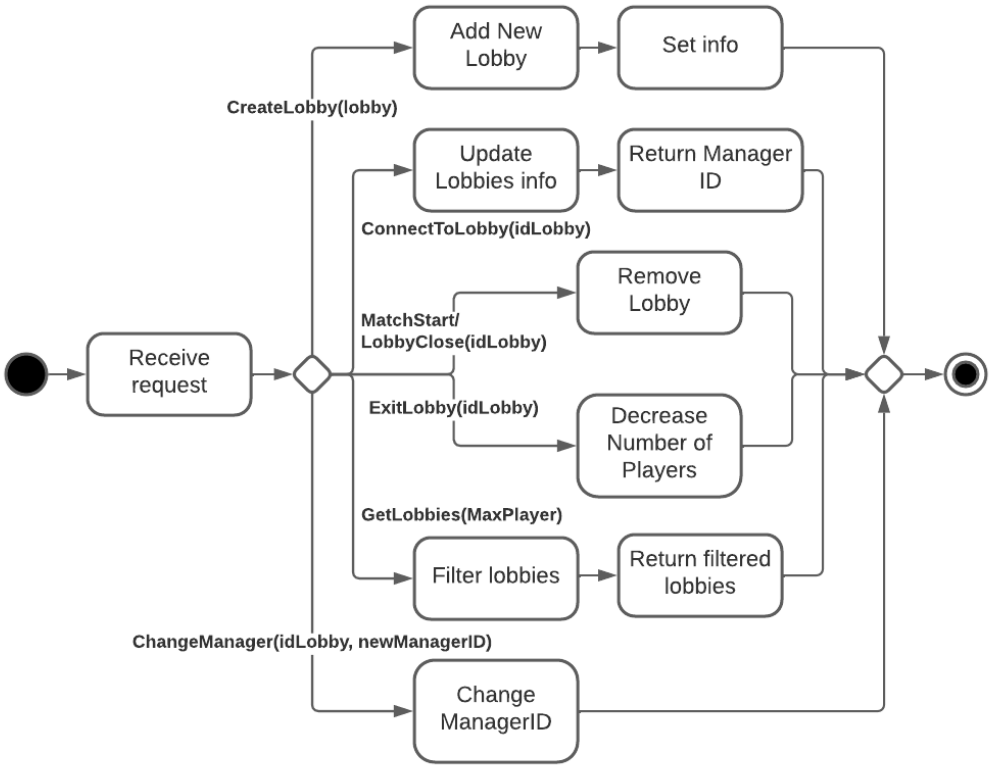


Figure 7: Activity Diagram of Server

Figure 7 shows how the server handle his requests:

- **GetLobbies(MaxPlayer)**: Server prepare a JSON array with the matching MaxPlayer number lobbies he's currently storing and return it to the requester.
- **ConnectToLobby(idLobby)**: Update information relative to current number of player inside the lobby and return to the requester a JSON object with the managerID to connect to.
- **CreateLobby(lobby)**: Server create a new lobby with info received, setting the current manager of that lobby as the requester who asked to make it.

- **ExitLobby(idLobby)**: Server decrease the number of players inside the provided lobby.
- **ChangeManager(idLobby, newManagerID)**: Server change the manager ID of the provided lobby with newManagerID.
- **MatchStart/LobbyClose(idLobby)**: Server delete the lobby from his storage, at this point he doesn't really care if the lobby was closed or the match started.

5.3.2 Client Lobby Behaviour

When lobby is created we need to manage the Clients behaviour inside it. Using P2P Vert.X API model each client expose the same API for communication (which are always broadcast to all the other clients), we divide this part in two diagram:

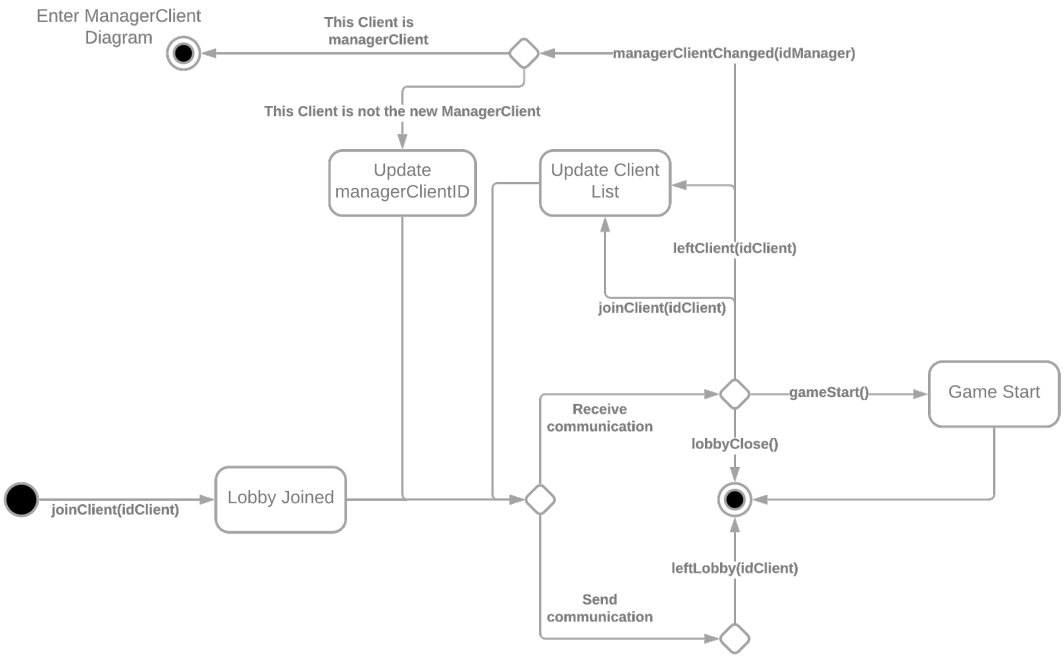


Figure 8: Activity Diagram of Client Lobby

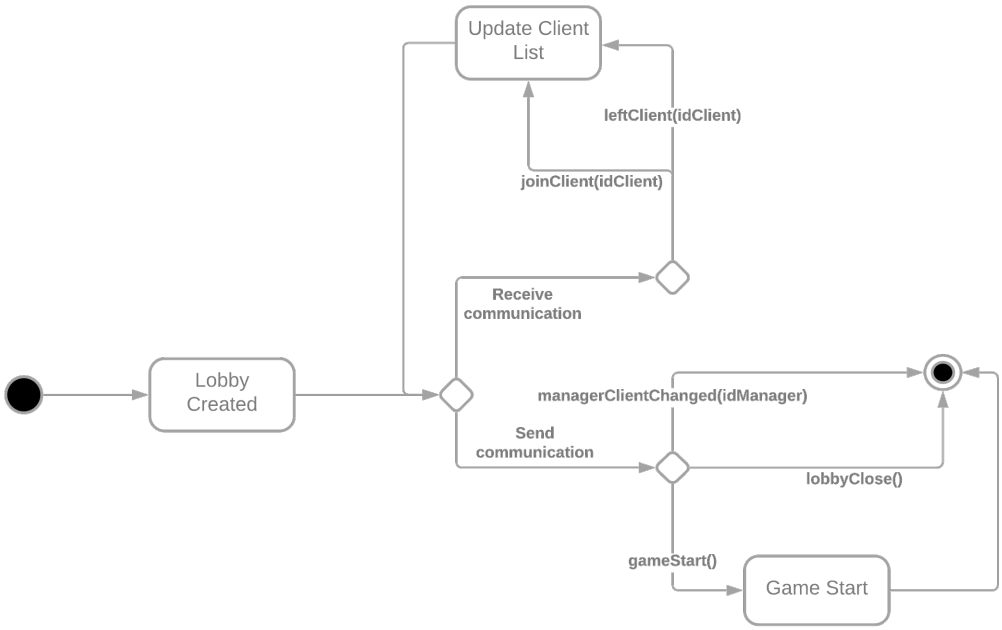


Figure 9: Activity Diagram of Manager

Figure 8 and Figure 9 show how internal activities relatives to requests and responses they have to handle:

- **joinClient(idClient)**: Client gets notified a new client has joined the lobby, locally update his list of clients adding the client retrieved by JSON object received. In this way he can now send data to him.
- **leftClient(idClient)**: Client gets notified a client has left his lobby, locally update his list of clients removing the client retrieved by JSON object received. In this way he stops to send data to the leaver client.
- **managerClientChanged(idManager)**: Client gets notified by the manager that he assigned his Manager role to another client, by the JSON object he retrieves the new idManager and store it locally. If the new idManager corresponds to his idClient it means he got promoted and as a consequence have to start working as expected.
- **gameStart()**: Client gets notified by the manager that the game has started, lobby phase is over and game can be initialize, we here pass to the Game Client Behaviour[5.3.3].
- **lobbyClose()**: Client gets notified by the manager that the lobby has been closed, he can consider his self as out of any lobby and may back to communicate with server through his API [5.2.1].

5.3.3 P2P Game Client Behaviour

The same method applied for the Client lobby it's applied here but regarding the client behaviour in game. Each Client communicate to each other always in broadcast, in this way everyone can maintain a consistent status of the game and observe game changing in real time even when it's not their turn.

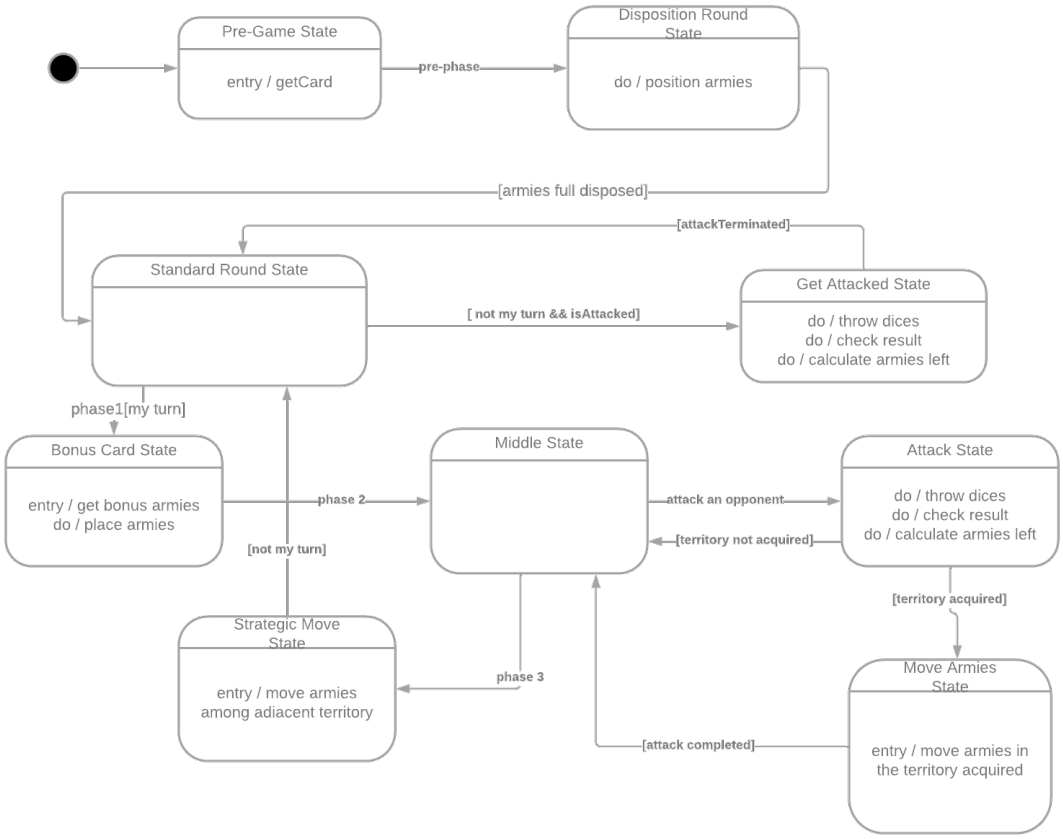


Figure 10: States Diagram of P2P Game

Figure 10 shows how client game part perform during phases of a match:

- **Pre-Game State**: Get cards and move to Disposition Round State in pre-state.
- **Disposition Round State**: Place armies in our territory and when over move to Standard Round State.
- **Standard Round State**: Waiting for our turn and move to Bonus Card State in phase 1 or get attacked by another player and move to Get Attacked State.

- **Get Attacked State**: Throw dice, check result and calculate number of armies left, move back to Standard Round State when attack is over.
- **Bonus Card State**: Get Bonus Armies if player has an entire continent or a set of card as describe in Table Continent-Bonus Armies and Table Card Symbols-Bonus Armies, then move to Middle State in phase 2
- **Middle State**: Can decide to attack an opponent moving to Attack State or finally move to Strategic Move State in phase 3.
- **Attack State**: Throw dice, check result and calculate number of armies left, if territory is acquired move to Move Armies State else move to Middle State.
- **Move Armies State**: Forced move at least an army in new territory, next move to Middle State.
- **Strategic Move State**: Move armies from an owned territory to an adjacent one.

5.4 Interaction

In this section will be described how different component of the system will interact with each other, every scenario is going to be linked with his sequence diagram.

5.4.1 Client join/create lobby Interactions

As shown inFigure 11 client decide to connect or join a lobby, in the first scenario the Manager Client tell the server with a PUT he wants to create a new lobby. In the second scenario it use a GET to the server retrieving information about lobby and then using a POST inform the Manager that it will join the lobby.

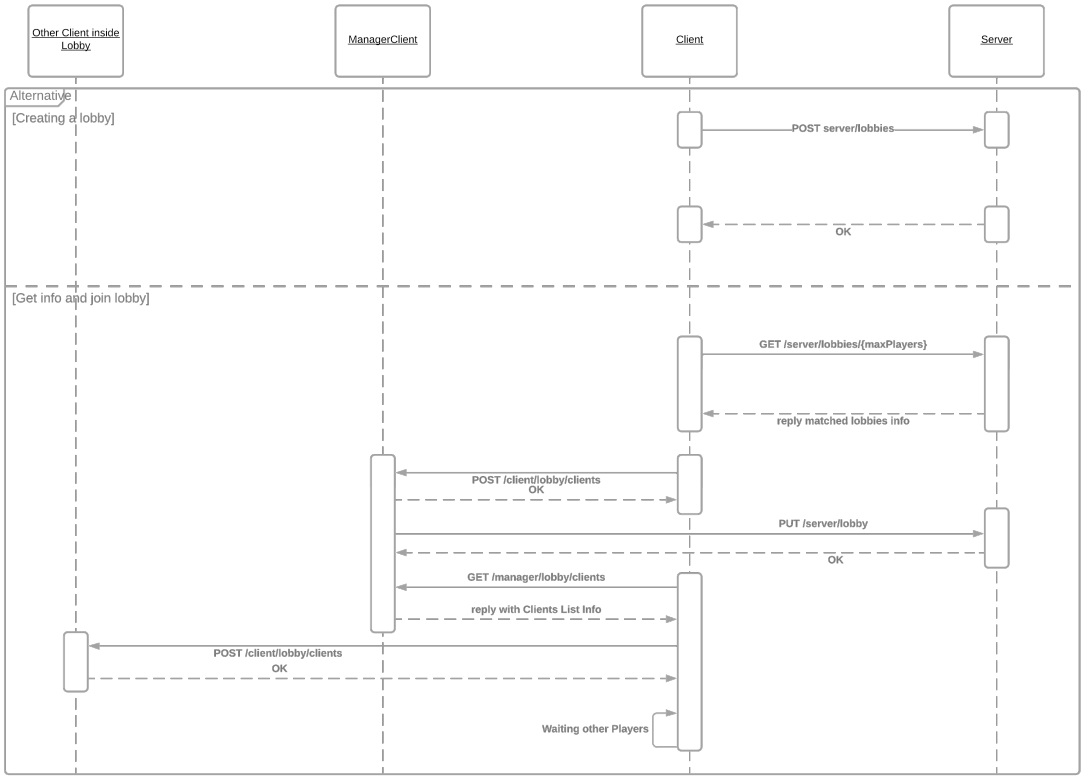


Figure 11: Sequence Diagram for the Join or Create Lobby Part

5.4.2 Client/Manager exit lobby or Manager changed Interactions

As shown inFigure 12 we describe the situation where Client or Manager Client wants to leave the lobby, the normal client just need to send a simple DELETE to the other clients and the manager will update the Server info. For the Manager Client part we have two alternative:

- If it wants to exit and close the lobby, it perform a DELETE from both other Clients and Server kill the lobby.
- Before exit, it elects a new Manager Client and then exit like a normal Client, informing Clients and Server with a PUT.

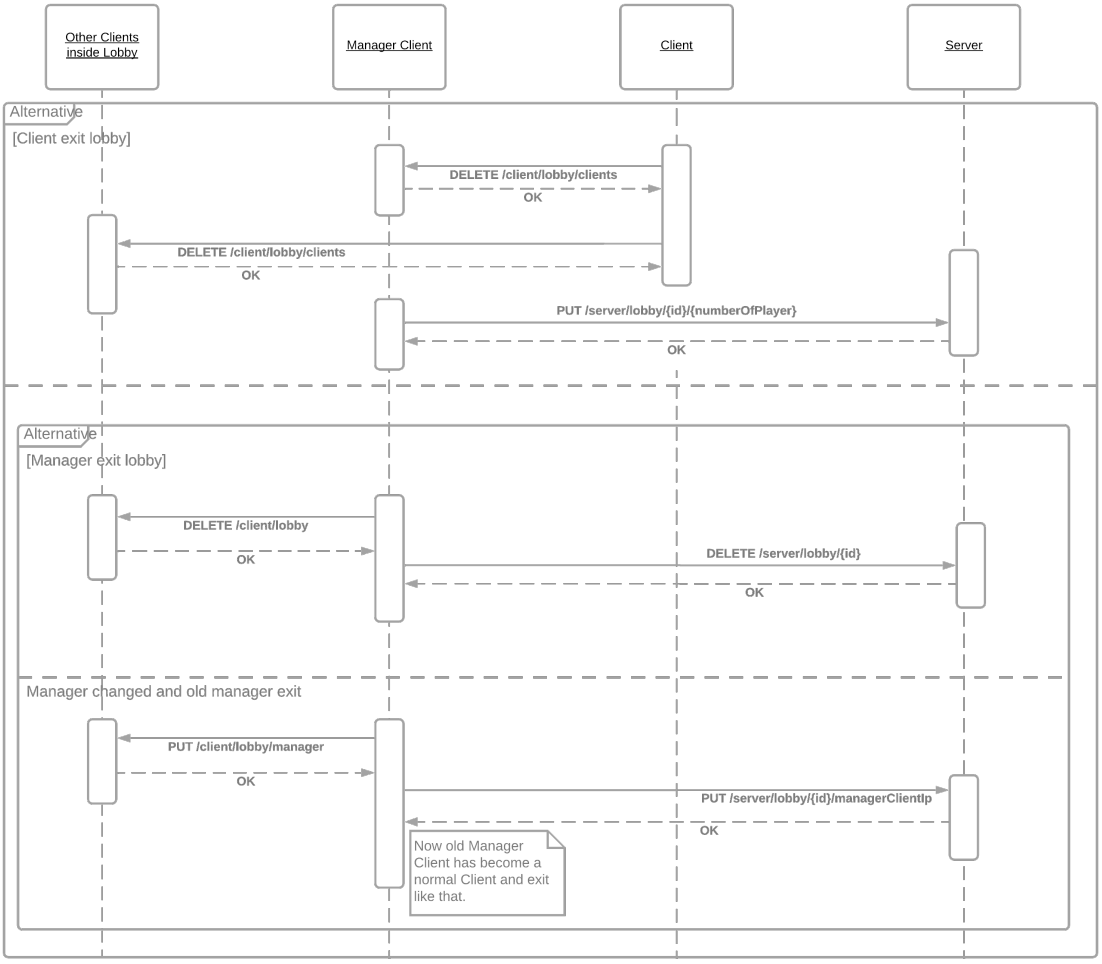


Figure 12: Sequence Diagram of Client or Manager Exit

5.4.3 Match has start/Lobby removed Interactions

As shown in Figure 13 ManagerClient is the guy who can decide when the game start or when the lobby should be closed, in both scenarios after sending a DELETE or a PUT to the other clients in lobby the Manager Client inform the server to keep him updated.

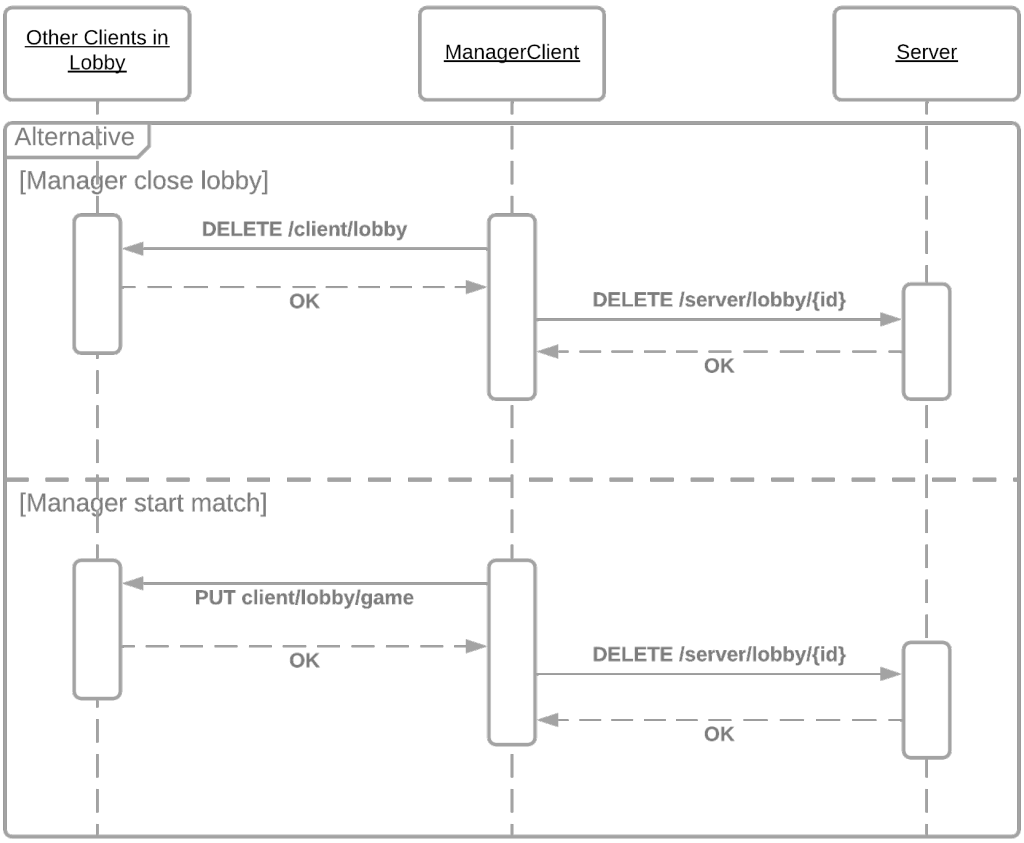


Figure 13: Sequence Diagram of Match starting or lobby closed

5.4.4 Start and Pre-Phase Game Interactions

Figure 14 describe the interaction between clients(including ManagerClient) at the start of the game in particular in the pre-phase round until the first turn (the Manager-Client one). First of all the ManagerClient tells other clients that the match is started, then distribute to anyone (even himself) the starting territories cards. After the distribution every client arrange his armies into his territories, notify each other and send when they finish a PUT to the ManagerClient. ManagerClient expect to receive exactly nClient(number of players in the match) - 1 PUTs to discover when he can start his first turn, from now on the interactions are looped as shown in 5.4.5.

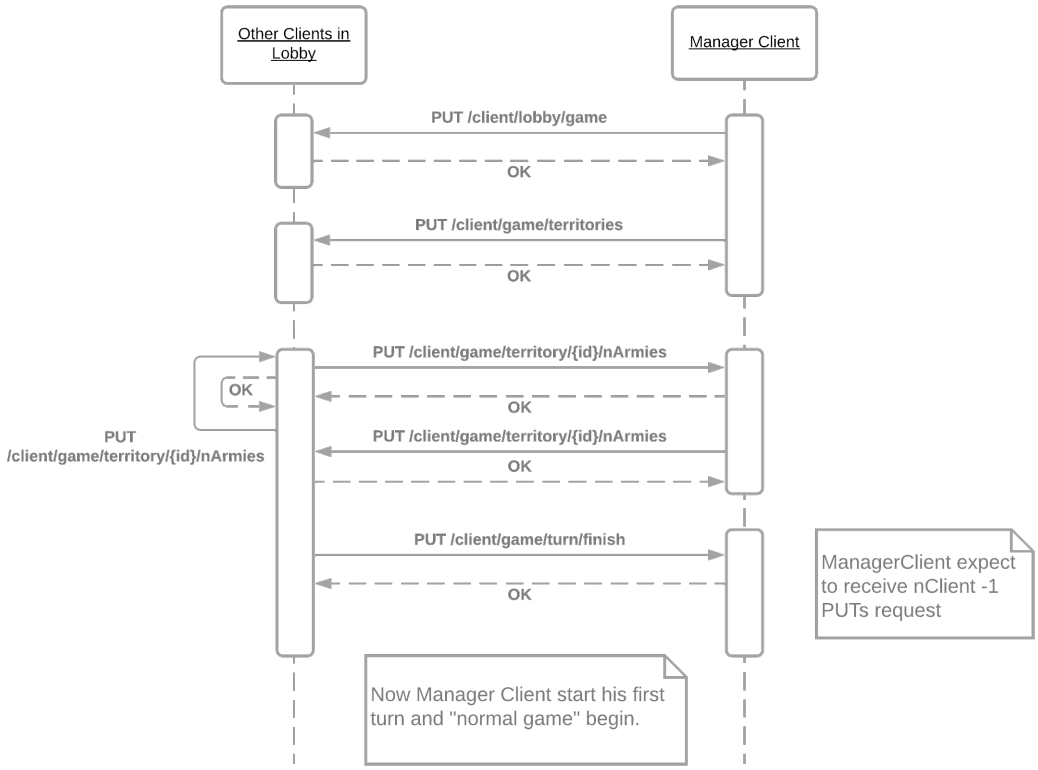


Figure 14: Sequence Diagram of pre-phase and start of the game

5.4.5 Game Round Interactions

Figure 15 shows the loop of a normal round game.

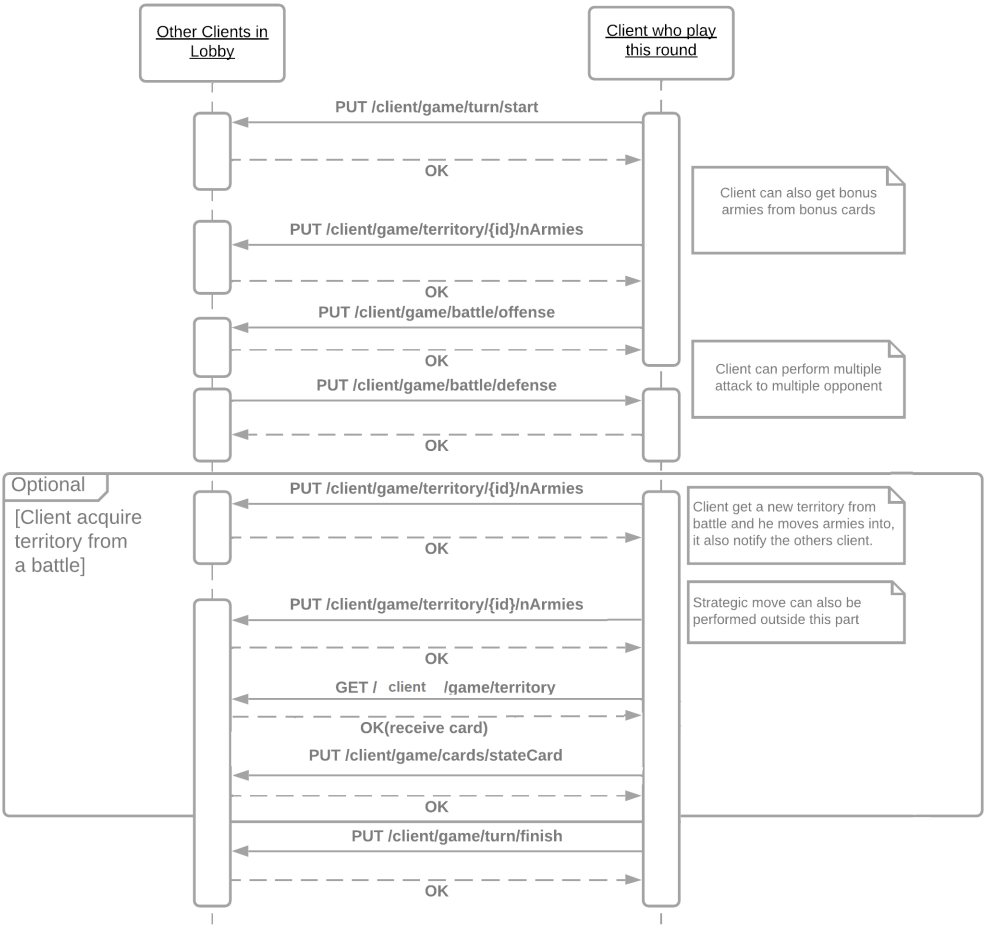


Figure 15: Sequence Diagram of Game Round Interactions

The client who plays this round inform other he's starting his turn, then get new armies including continent bonus and eventually state card bonus armies always notifying other clients(Phase 1). In Phase 2 client could start multiple attack(executed sequentially one by one) and if at least one territory was subtracted to an opponent at end of the round he will draw a stateCard. Phase 3 is focused on strategic move as explained in 1.3.3. Now the client notify the finish of his round to other clients, in this way the next client's turn can start and the loop is repeated.'

6 Implementation Details

In this section, we are explaining in details some non-easy understandable choices or algorithms we have made.

6.1 Static launcher

Figure 16 class diagram illustrates the organization and relationships between the main components of the application.
Launcher includes a ContextManager class, which manages the context parameters and handles window changes. It interacts with a ContextParameters class, which holds specific parameters related to the application context.
The GUIManager class is responsible for managing the graphical user interface (GUI) and handles window changes.
The Window class enumerates various window types, including BASE, LOGIN, GAME, LOBBY, and MANAGER. These window types define the different states of the application.
The methods in the Launcher class, such as userLogged(), gameStarted(), lobbyClosed(), etc., indicate the different states and events in the application, triggering changes in both the context and GUI.

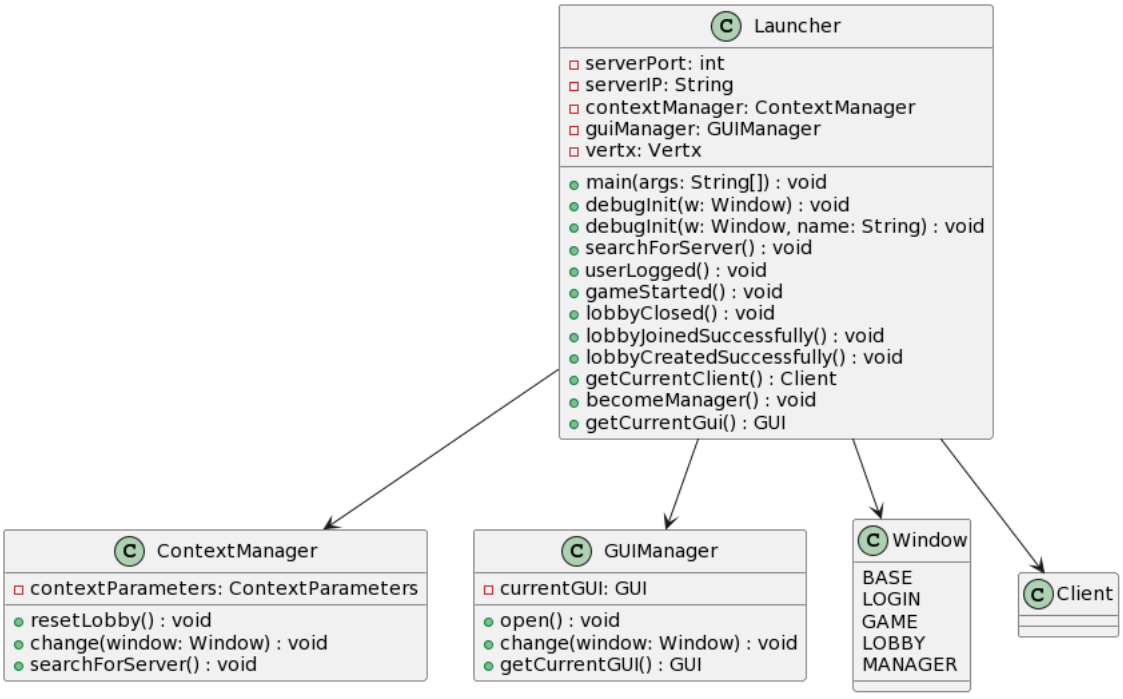


Figure 16: Game Launcher Class Diagram

The static launcher makes it easy to obtain anywhere in the code information needed, or to call methods to change state (from login to lobby selector, etc...) easily. Also, the decouple of the GUI to the backend is a great quality for the code itself.

6.2 Game order for players

In this part, multiple clients participate in a game, generating and transmitting random turn orders to synchronize game progression. On the receiving end, each client processes incoming orders, maintains order consistency, and manages the overall game flow. The sender side initiates the process by generating and transmitting a random order, while the receiver side handles the reception, synchronization, and optional features such as resending orders. This description provides a comprehensive overview of the interaction between sender and receiver components, showcasing the asynchronous nature of the communication and the error-handling mechanisms in place.

6.2.1 Sender Side (Java Function `sendRandomOrderForTurning`)

1. **Random Order Generation:**
   The sender generates a random order (`shuffledList`) representing a specific turn sequence in the game.

2. **Storing Random Order Locally:**
   The sender maintains a local map (`randomOrderList`) associating each client's IP with its corresponding random order.

3. **Asynchronous Order Transmission:**
   For each client, the sender initiates an asynchronous communication process, making an HTTP PUT request to transmit the random order to the "/client/game/order" endpoint.

4. **Handling Responses:**
   Asynchronously handles responses from each client:
   - If response status is 200, prints a success message.
   - If response status is 500, prints an error message indicating client downtime.
   - For other response statuses, prints a generic error message.

5. **Completion and Return:**
   Returns a `Future<Void>` after all promises are either completed or failed.

6.2.2 Receiver Side (Router Handling `/client/game/order`)

1. **HTTP PUT Request Handling:**
   Receives and handles incoming HTTP PUT requests to the "/client/game/order" endpoint.

2. **Extracting IP and Order:**
   Extracts the IP address and random order from the request body.

3. **Calling `receiveRandomOrder`:**
   Invokes the `receiveRandomOrder` method on the game client, passing the IP address and received random order.

6.2.3 Receiver Side (Java Function `receiveRandomOrder`)

1. **Order Processing:**
   Processes the received random order:
   - If `orderFound` is false, updates the local `randomOrderList` with the received order associated with the sender's IP.
   - Checks if random orders from all clients are received and if they are all the same.
   - If not the same, triggers a process to clean up and repeats the order transmission with a randomly picked order.
   - If the orders are the same, marks `orderFound` as true.

2. **Handling Order Found:**
   If `orderFound` is true:
   - Prints an order found message.
   - Sends the order again to all clients for synchronization.
   - Initiates the turn if it's the first player's turn; otherwise, enters a waiting phase.

3. **Simulating Resending (Optional):**
   If `orderFound` is true, there's a chance to randomly send the order again. This optional step it's used for redundancy to ensure that all clients have the correct order.

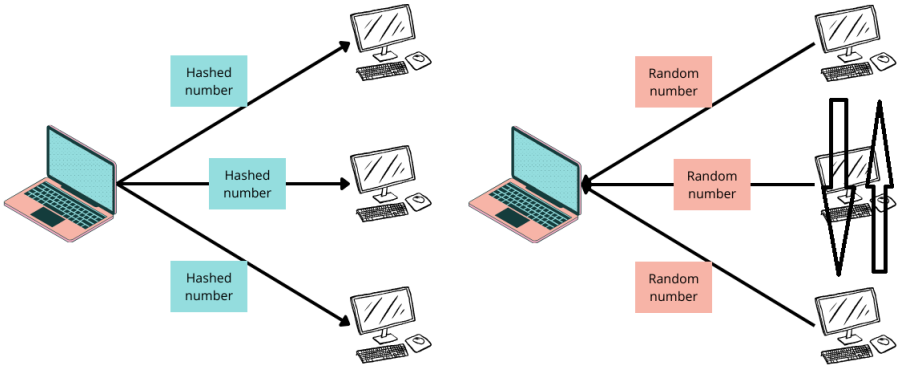
6.3 Dice throw

The game is based on dice throwing, making it so important that it can't be a simple random number generated from one of the peer, simply because we don't want anybody to cheat! In order to preserve the fun of the game, we had to come up with a solution to this problem:

> How can we throw dice in a peer-to-peer system, making everybody reach the consensus that the dice are actually random?

We came up with a relative simple solution, which make use of hashing, divided in two phases:

Throw init



The "pink-client", the one who wants to throw dice, start the communication following those steps:

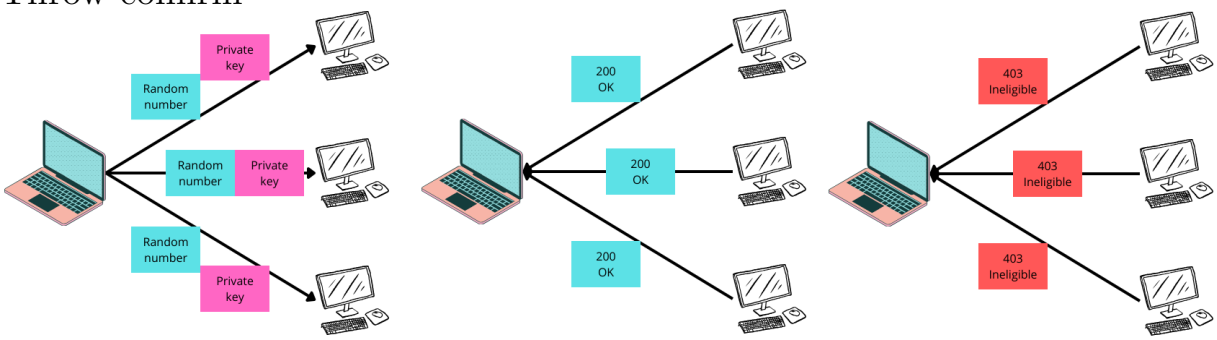
1. Generate a random number R.
2. Generate a random private key K.
3. Generate the hash of R with key K, producing H.
4. Send H to every else.

Each other client, when receive the throw init, acts like this:

1. Generate a random number CR.
2. Share CR with all other clients.
3. Send CR to pink-client.

Everyone should sum up their own random number with the random numbers received.

Throw confirm



The pink-client now has all the random numbers collected, so he can compute his dice throw, but still need to be confirmed from other clients, so he acts like this:

1. Sum all random number into S.

2. Compute S % 6 + 1 (to clip from 1 to 6, a dice) into D.

3. Share with other clients R and K.

4. D is dice threw, but need confirmation response.

Now all clients know all random number, and can confirm that the pink-client random number correspond to the one he generated before knowing the other clients random numbers:

1. All other clients sums all random number.

2. Compute the actual dice threw, so they know.

3. Compute hash CH of R with key K.

4. If CH equals H, respond with 200, else 403 ineligible.

In this way, no one has the possibility to cheat on dice throw, even if all other clients organize to cheat, they don't know the random number of the pink-client and can't calculate it due to preimage resistance of hashing functions. In this way, everybody contribute to the randomness of the dice throw, but nobody can manipulate it.

7 Self-assessment / Validation

In this section is described the way we validated our project, how functional requirements has been tested and how non-functional requirements has been asserted.

7.1 Functional requirements tests

Functional requirements simply provide questions that can be answered in order to verify the correct functionality of the system. Those answers can be given manually, but real advantages comes with automatic testing, like in our project. In this project, the functional requirements tests can be translated to API tests; it's easy to see that "if all requests give the expected results, we have a correct functioning system", but problems are not all solved, tests like this are not and can't be exhaustive, to be sure of the correct functioning we should write an infinite amount of tests and use an infinite amount of time to test the system; to avoid the usage of infinite time, but to reach a more secure way to test our project, we didn't only focus on the responses given by API requests, we also checked that the state (information stored, variable values...) were correct all along during each phase of tests, before and after each API call. This, once again, doesn't provide a perfect way to test (still impossible), but give us more trustfully and controlled tests, which are way more reliable.

7.2 Non-functional requirements analysis

Non-functional requirements do not really have a standard way to be tested, each of them has its own way to be verified. In this fragment, we will describe how this project respect the non-functional requirements described in 3.2:

- **System must be maintained easily**: "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.", the written code is divided in packages, each package represents a different screen of the project and its inner structure is always the same: the GUI part, the backend part, the communication package divided in receiver (the server part of a p2p client) and sender (the client part of a p2p client). Every screen is managed by the Launcher, which uses managers of GUI and backend to keep code readable and short. Thanks to this role separated structure, the code can be considered easily understandable (what code is responsible for what behavior, how the code works), easy to find what needs to be changed in order to fix bugs.
- **System must be extended easily**: thanks to the modular structure described above, summed with the test of functional-requirements, drive it easy to make changes and check that the changes have not introduced any bugs, making the project naturally extendible.
- **System must manage client connection/disconnection in order to prevent unexpected case**: in the phase of match selection if a client disconnects,

there are no problems; when in a lobby, if a client disconnects, other clients notice it by a message or timeout and simply remove him from the lobby; when in game, if a client disconnects, when other clients notice it, they simply set all his territories as neutral. If a client reconnects before the timeout, nobody notice and nothing change.

- **System must be scalable**: in this case, the scalability is naturally intrinsic in the project, that because it does not really matter how many players are gaming together, due to the fact that the game is peer-to-peer, meaning that even if billions of players are gaming, nothing will prevent you from starting a new game (the only limit here is the actual bandwidth internet is capable of, but this project won't generate such a throughput of data to block internet). The only observation to take notice is about the server which storage lobbies of non-started games, of course this server can be overloaded if a lot of people tries to join/create lobbies all together; that problem can be easily solved in the same way every online game solve it, thanks to the usage of multiple geographical separated servers, not originally an auto scalable system in terms of power, usage, and consumptions, but can be considered completely scalable if server-less functions are used.

8 Deployment Instructions

In this section, we explain how to retrieve and successfully launch the project and the tests.

8.1 What is needed

Before you can run this project, you need to have installed and correctly working those following software programs.

8.2 Docker

For any environment, we will need a working and running docker instance. Simply follow the online guide: Docker install.

8.2.1 X11 (Windows)

If you are using Windows, you will need an X server to be able to see GUI through Docker, in this case we'll be using Xming, a X11 server for Windows OS. You can download and install it from `https://sourceforge.net/projects/xming/`. Once installed, open it up and start the server, default configs are good for this project, so you just need to press the "ok" button until it correctly starts.

8.2.2 X11 (Linux)

If you are using Linux, check if X11 (xorg, xauth, x11-apps). Then launch Docker Desktop, navigate to the "Volumes" section and add the shared volumes /tmp/.X11-unix.
[1] In Linux distributions we need the X server on the host to display our GUI applications. In this case a little commitment from the user is required by customize the LinuxClient/LinuxClientTest Dockerfile at line 12 with the user ID (UID) and the group ID (GID) that we can obtain from the id command:

```
id \$USER
```

After that we need to provide to docker the access to our display via shell command:

```
sudo xhost +local:docker
```

8.2.3 Git

To easily retrieve the project.
Once correctly installed from GIT, you need to clone the repository and init the submodules, to do so run the following shell commands:

```
git clone https://gitlab.com/pika-lab/courses/ds/projects/
ds-project-benvenuti-squarcialupi-ay2122.git

git submodule --init --recursive

git pull --recurse-submodules

git pull
```

If you want to avoid using the –recurse-submodules option, you can configure git as follow:

```
git config --global submodule.recurse true
```

From now on you just need to git pull to update the entire project. Now the project is correctly downloaded and ready to go.

8.3 Launch the project

Once you had all software needed for the project and the project itself, you can launch this project.
To run the project, open a command shell and type:

```
docker-compose -f (WindowsCompose.yaml/LinuxCompose.yaml) up --build
```

To run tests of the project, run the shell command:

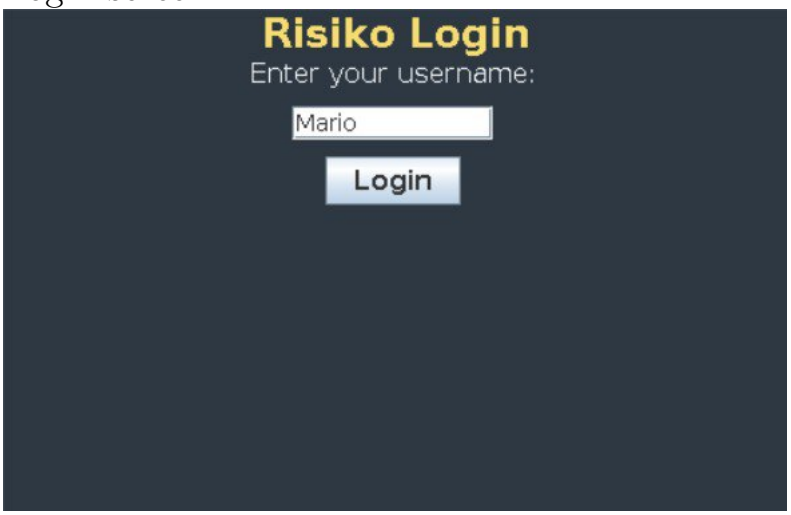
```
docker-compose -f (WindowsComposeTest.yaml/LinuxComposeTest.yaml) up --build
```

In the end, if you want to clean up storage occupied by Docker containers, you can run the following shell command:

```
docker-compose down
```

9 Usage Examples

Login screen



- User enters their username and password.
- Clicking "Login" button authenticates the user.

Lobby Creator

- Allows a user to create a new game lobby.
- Assigns a unique lobby code for friends to join.
- Max players and Name should be set from the user.
- By clicking the Switch button you can move to the Lobby Selector page.

Lobby Selector

- Displays a list of available game lobbies.
- Enables users to join a selected lobby.
- By clicking the Switch button you can move to the Lobby Creator page.

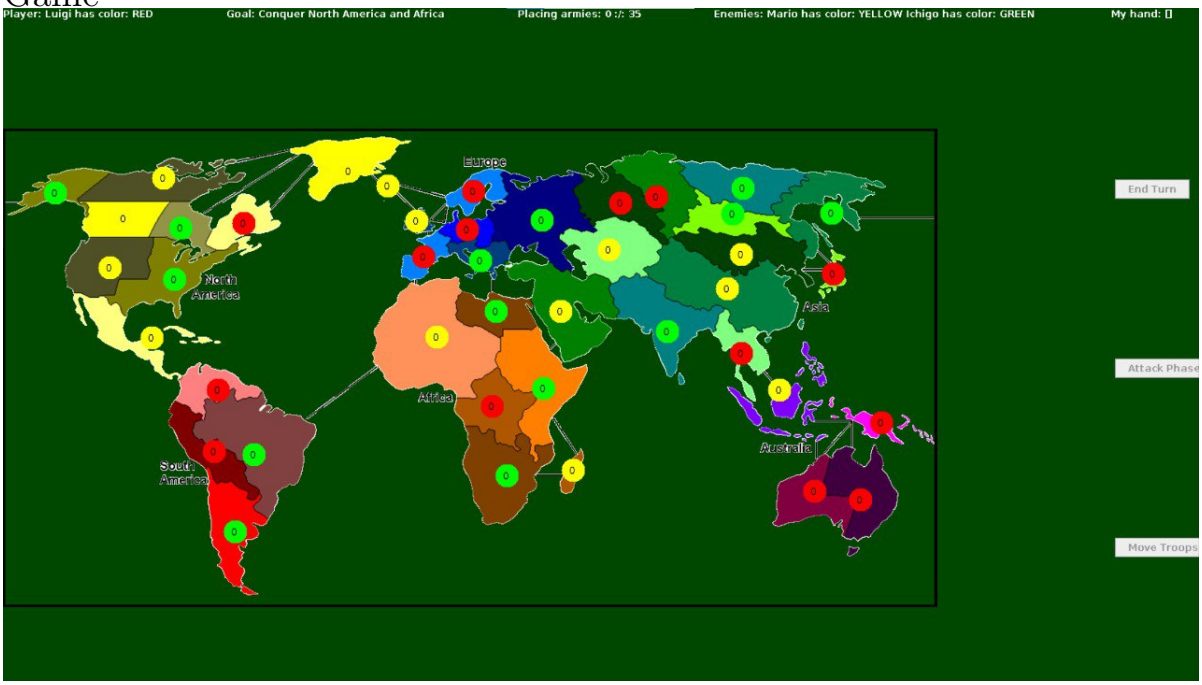
Waiting Lobby for Manager

- Appears for the player who created the lobby.
- Displays the list of players who have joined.
- When lobby is full the manager can start the game.
- If Exit Lobby is clicked, one of the players connected become the new manager.

Waiting Lobby for Players

- Appears for the players who joined the lobby.
- Displays the list of players who have joined.
- By clicking the Exit Lobby button players can exit the lobby and return to Lobby Selector scene.

Game



- Displays the game board with territories, armies, and borders.
- Allows players to take turns placing their initial armies.

- Facilitates the actual gameplay such as: troop movements, battles, placing armies/extra armies and using cards.
- Displays notifications and updates on game events.

10 Conclusions

In the course of this project, we undertook the development of Risiko!. Particularly noteworthy in the context of distributed systems is the peer-to-peer architecture employed in our implementation. In the context of traditional tabletop gaming, each player is afforded visibility into the prevailing game state, with a built-in safeguard against cheating, as fellow players scrutinize one another's moves. In a conventional server-client paradigm, ensuring integrity is relatively straightforward, as the server functions as a trusted authority, and consensus is effortlessly attained by adhering to the server's directives. Conversely, in a peer-to-peer framework, establishing consensus becomes a formidable challenge due to the absence of a universally trusted entity. Consequently, each action necessitates meticulous examination and validation by all participating clients. The adoption of a peer-to-peer model in tabletop gaming presents a myriad of opportunities for the implementation of sophisticated algorithms, which we actively pursued and integrated into our project.

10.1 Future Works

- Managing the drawing of cards involves treating it as an undisclosed source, preventing the player cheating in the draw. Simultaneously, the other players possess the capability to authenticate the draw without knowing the specific card chosen.
- Handling crashes of other clients in the same lobby/game, our project correctly behave in the case one client voluntarily deice to exit, while in a lobby other clients simply remove him from the list of players and the player himself communicate with the lobby server his exit; in the case of a running game, when other clients receive the exit information they simply assign all his territory as neutral and remove him from the order of playing turns. To handle a crash of a client, other clients should be able to detect it, to do this we can either decide to: have a keep alive packet that is sent periodically, use a timeout on requests; in both cases when someone detects a client failure he should communicate with other clients to reach the consensus that a client has failed and take actions to properly remove him from the lobby or game as described above (obviously in the lobby, the manager should notify the lobby server, because the failed one will not do it).

10.2 What did we learned

In this project, we gained valuable insights into various aspects of Docker and network programming. Noteworthy lessons include:

- Service Scaling: We discovered the importance of creating larger services in the 'compose.yaml' file, optimizing resource utilization and enhancing overall system performance.

- Environment Variable Management: We learned effective strategies for handling environment variables, ensuring seamless configuration and adaptability in different deployment scenarios.
- Shared Volumes: Our experience involved implementing shared volumes, facilitating data persistence and efficient communication between different components of our system.
- Dockerfile Expertise: The process of Dockerfile creation provided us with valuable insights into optimizing container builds, minimizing image sizes, and streamlining application deployment.
- Network Programming Challenges: We encountered the inherent difficulties of network programming, grappling with issues like Heisenbugs that underscored the complexity of this domain.
- Vertx Considerations: While Vertx proved to be a comprehensive framework with a myriad of features, we discerned its potential drawbacks for simpler projects. Its monolithic nature necessitates a significant learning curve, demanding a substantial investment of knowledge for tasks that may not require extensive functionality.

References

[1] Javier Lobato Perez. Running gui applications in a linux docker container, 11 November 2023.