Galaxy Trucker

Boglioli Alessandro, Galimberti Matteo, Kumar Tanish

30 May 2025

1 Introduction

The aim of this project is to implement a multiplayer version of the board game *Galaxy Trucker* using a client-server architecture in Java. The application supports multiple players, real-time interaction, and automated game state management. The system is designed with modularity and scalability in mind, utilizing established software engineering design patterns. This document provides an overview of the system architecture, communication protocol, and the main design choices made during development.

2 Communication Protocol

2.1 Overview

This document describes the communication protocol between client and server in the implementation of the Galaxy Trucker game. The protocol is based on object serialization over TCP sockets. The architecture leverages several established design patterns to ensure modularity, scalability, and maintainability.

2.2 Architectural Patterns Used

Command Pattern. The **Command Pattern** is applied to encapsulate every client request as an object that implements a common interface. Each request sent from the client (such as drawing a card, placing a tile, or validating a ship) is represented by a concrete `Request` subclass. The server receives these request objects, which contain all necessary information and expose an `execute(Controller controller)` method. This allows the server to process each command polymorphically, decoupling the request handling from the communication layer.

Chain of Responsibility Pattern. The **Chain of Responsibility Pattern** is reflected in the server's controller dispatch mechanism. When a request arrives, it is delegated to the appropriate controller (either the `MenuController` during the pre-game phase or the `GameController` during the match). Each

controller is responsible for handling only those requests it understands, passing unrelated requests onward or discarding them. This design makes it easy to manage multiple phases of the game and extend the protocol with new request types as the project evolves.

Singleton Pattern. The **Singleton Pattern** ensures that only one instance of key controllers (`MenuController` and `GameController`) exists at any time. This guarantees a centralized coordination of the game state and communication flow, preventing inconsistencies and synchronization issues. Singleton access methods (such as `getInstance()`) are used to retrieve controller instances throughout the server codebase.

Observer Pattern. The **Observer Pattern** is used to manage the asynchronous delivery of responses from the server to the clients. The server maintains a list of `Observer` objects (one per connected client), which are notified whenever there is a change in game state or a response needs to be sent. When a controller processes a request and generates a response, it notifies all relevant observers using the `notifyAll` or `notify` methods, which in turn forward the response to the clients. This ensures loose coupling between the core game logic and the network communication layer.

2.3 Protocol Flow

The communication protocol is based on the following flow:

- When a client connects, the server immediately sends a `SenderIdResponse` to communicate the client's unique session identifier.
- **Client requests** are serialized and sent to the server as concrete `Request` objects.
- The `ClientHandler` on the server deserializes the request and forwards it to the current controller.
- The controller processes the request (using the Command and Chain of Responsibility patterns) and updates the game state as needed.
- The controller then generates a `Response` object, which is sent to the appropriate client(s) using the Observer pattern.
- Every request sent by the client has as **first parameter** the unique identifier given by the socket

3 Heartbeat Protocol

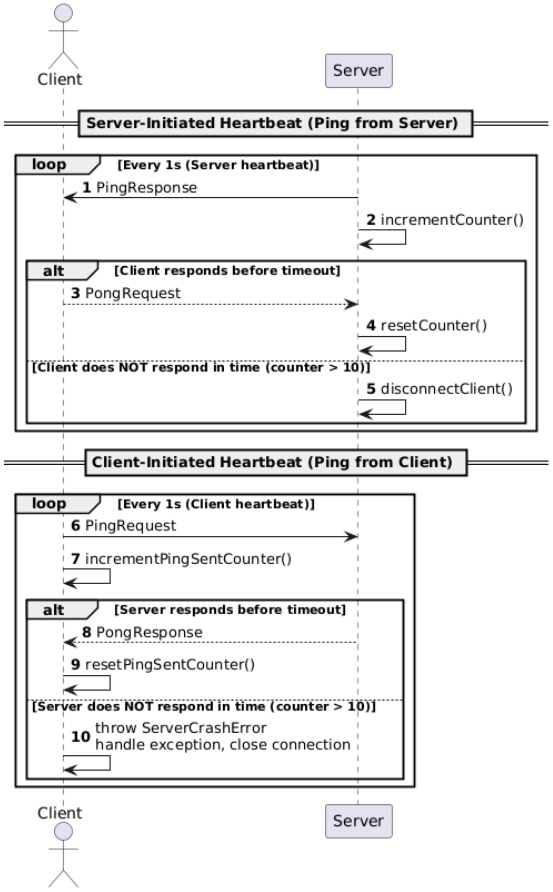

Figure 1: Heartbeat communication between Client and Server.

4 Game Phases

4.1 Menu Phase

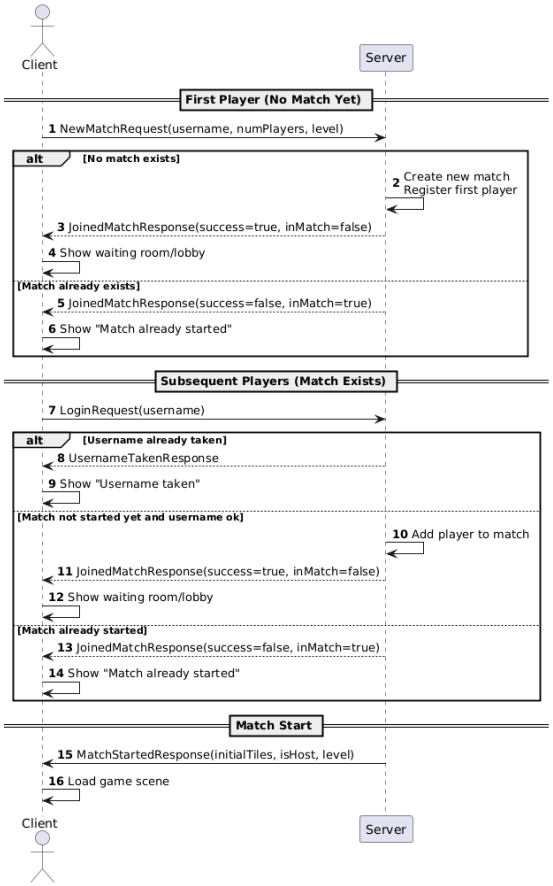

Figure 2: Lobby setUp for the game

4.2 Construction Phase

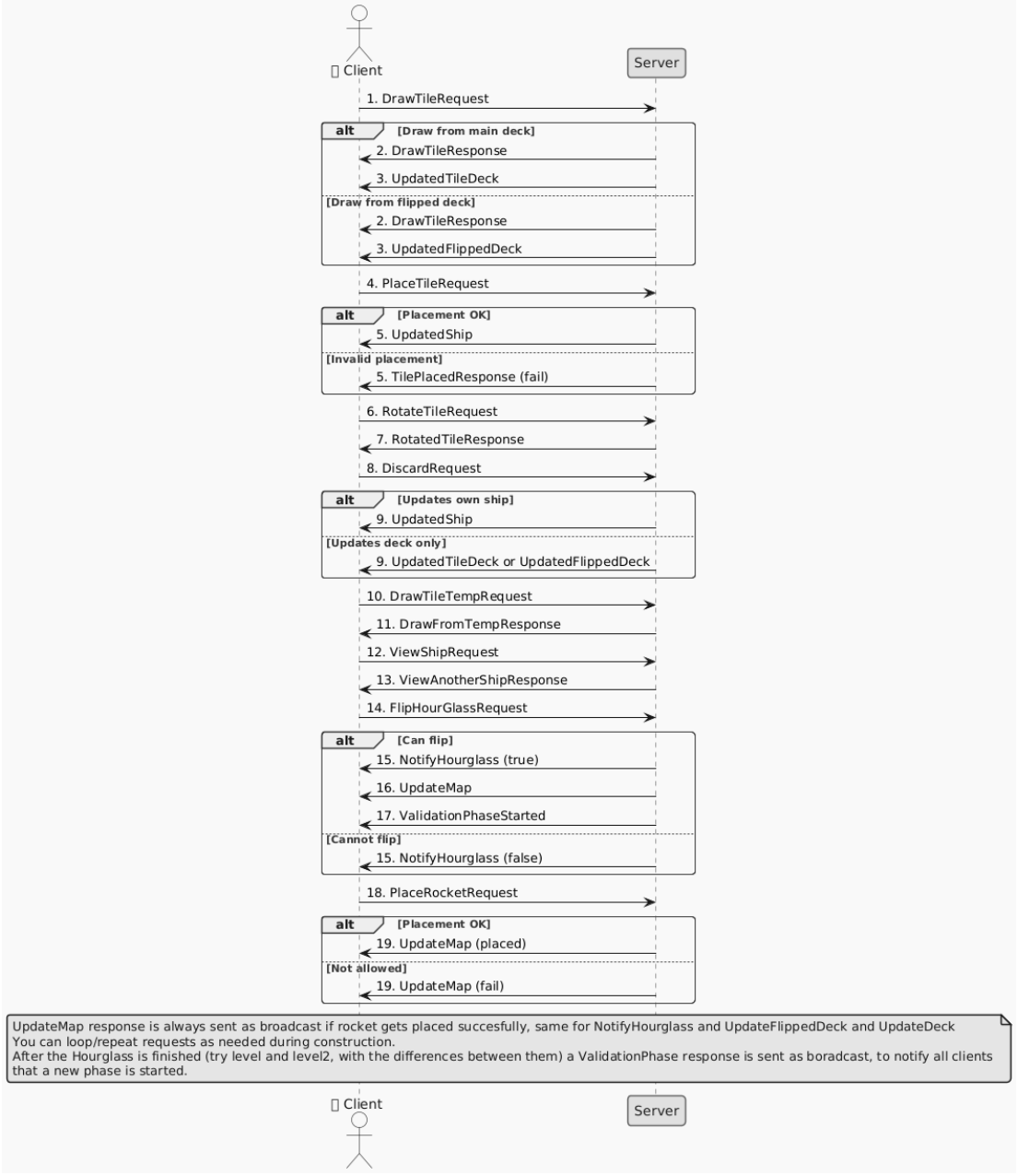

Figure 3: Construction phase of the game

4.3 Adventure Phase

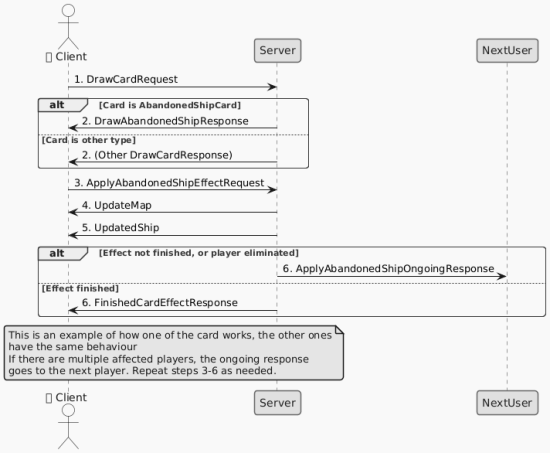

Figure 4: Adventure phase of the game