# Client-Server Communication Protocol

[CG04] - Russica, Staffoni, Stanghellini, Tisato

June 2025

## Contents



## 1 Introduction

This document describes the interaction between the client and server, explaining how player actions are handled by the server to update the game state and how these updates are then propagated to all connected players.

Given the large number of possible actions a player can perform during the game, and the fact that each action is only valid in specific game states under certain conditions, we chose to adopt a combination of the state pattern and the command pattern. This design allows the game to exist in a particular state, where only the actions valid for that state are implemented and executable.

In short, the core idea of the communication protocol is as follows:

1. The client creates and sends an `Action` to the server
2. The server receives the `Action` and forwards it to the controller
3. The controller identifies the game the `Action` refers to and retrieves the current state of that game
4. The controller executes the `Action` based on the implementation defined for that specific state

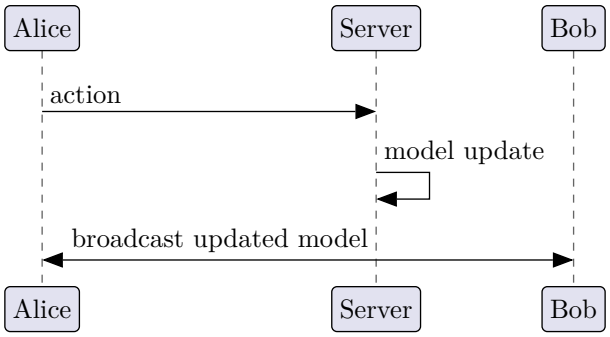


**Figure 1:** Diagram that represent the basic idea of the

### 1.1 `Message` structure

To exchange information we relied on a record `Message` comprised of a `messageType` field and an payload field.

The key message types we used were:

- `ACTION`: where the payload is a player generated action to change the game state
- `GAME`: where the payload is the updated version of the game
- `LOG`: where the payload is a list of strings

On top of them we used other messages to perform specific tasks like setting the nickname and broadcasting information about joinable games.

## 2 Command/State pattern for the actions

The `Action` interfaces has the following methods:

- `Action.checkAction()`
- `Action.execute()`

### 2.1 `Action.checkAction()`

The `Action.checkAction()` is in charge of checking the integrity of the parameters passed when the Action was created.

To give some brief examples:

- in `createGameAction` the method checks if:
  - game level is correct
  - number of players is correct
  - color is allowed and available
- in `chooseBatteryAction` the method checks if:
  - coordinate of the tile is in bound
  - the coordinate host a battery tile
  - the battery tile contains batteries

### 2.2 `Action.execute()`

The `Action.execute()` method is executed by the controller and it is in charge of applying changes in the server-side model, if the action received is allowed in a given state by a given player.

All `Action.execute()` are implemented as follows

```
public void execute(Player player) throws InvalidStateException {
        GameState state = player.getGame().getGameState();
        state.exampleAction(player, x, y);
        this.addLogs(state.getLogs());
}
```

The current `GameState` of the `Game` that the `Player` is in is retrieved, and the state-specific implementation of `Action` is called.

An `InvalidStateException` is thrown when an action has correct parameters but cannot be executed by the player in that state.

# 3 Sequence diagrams

## 3.1 Game creation and joining

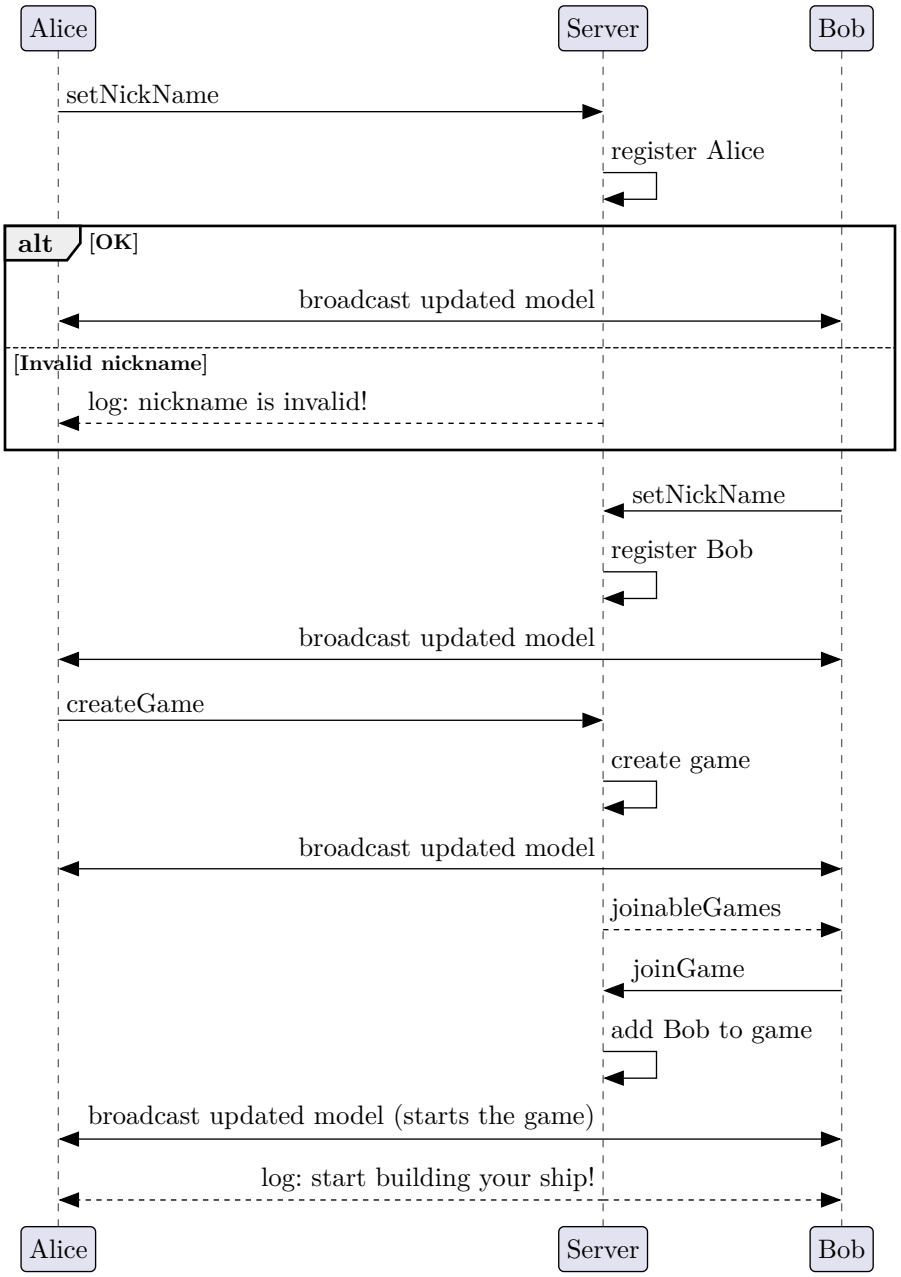


**Figure 2:** Diagram representing the typical game initialization and joining phase.

### 3.2 Building phase

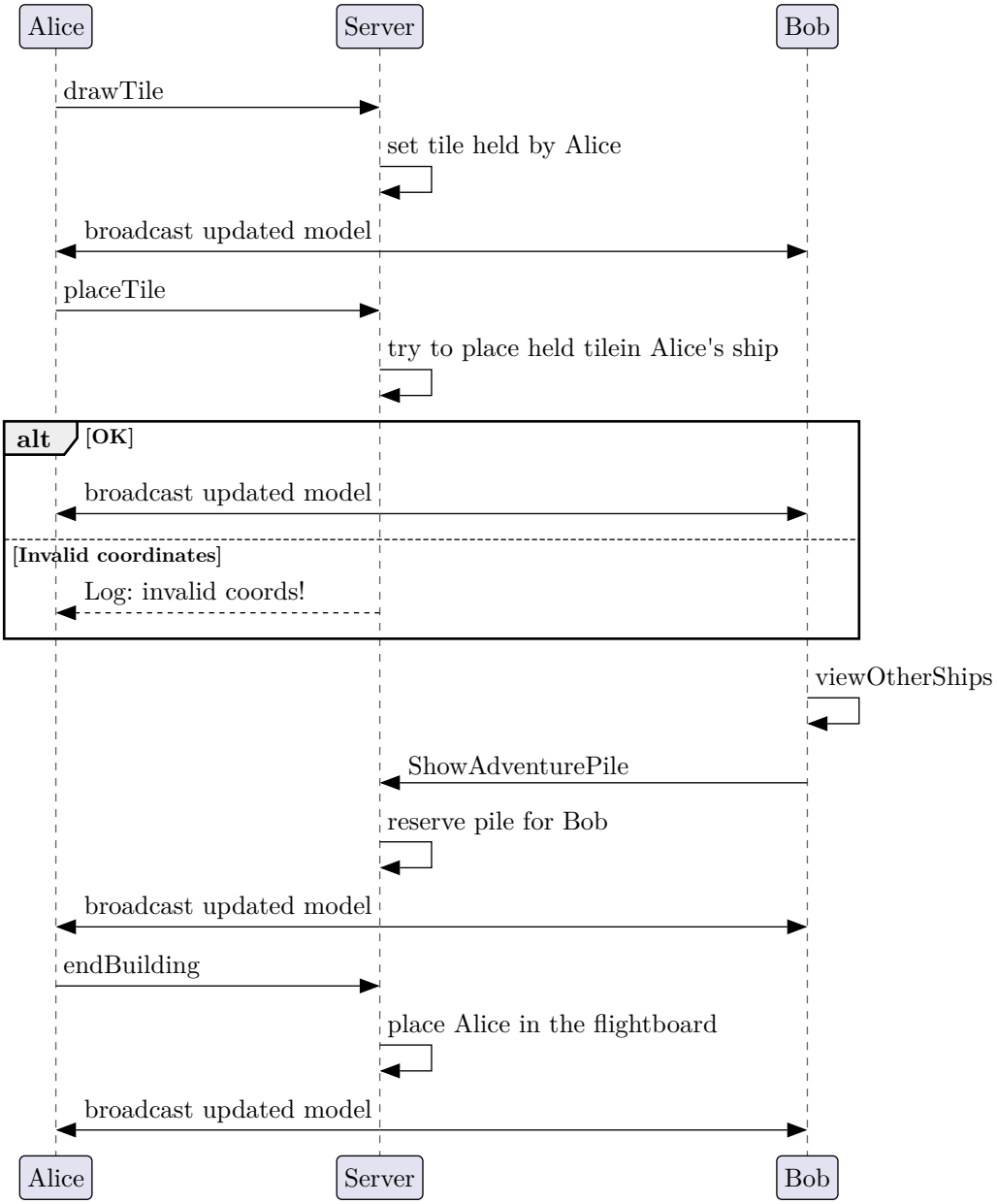


**Figure 3:** Diagram representing some action sequence that can happen in the build state.

Several other actions can also be executed by the players, such as: `startTimer`, `chooseFaceUpTile`, `placeInBuffer`, `chooseFromBuffer`.

Thanks to the pattern we implemented all actions are handled in the exact same way, regardless of the change they produce on the `Game` or the `GameState`

### 3.3 Adventure state

Let's analyze how the players state and the GameState evolves for adventure cards like `MeteorsRain`. Considering Alice as the leader of the pack.

### 3.3.1 `MeteorsRain` State actions sequence

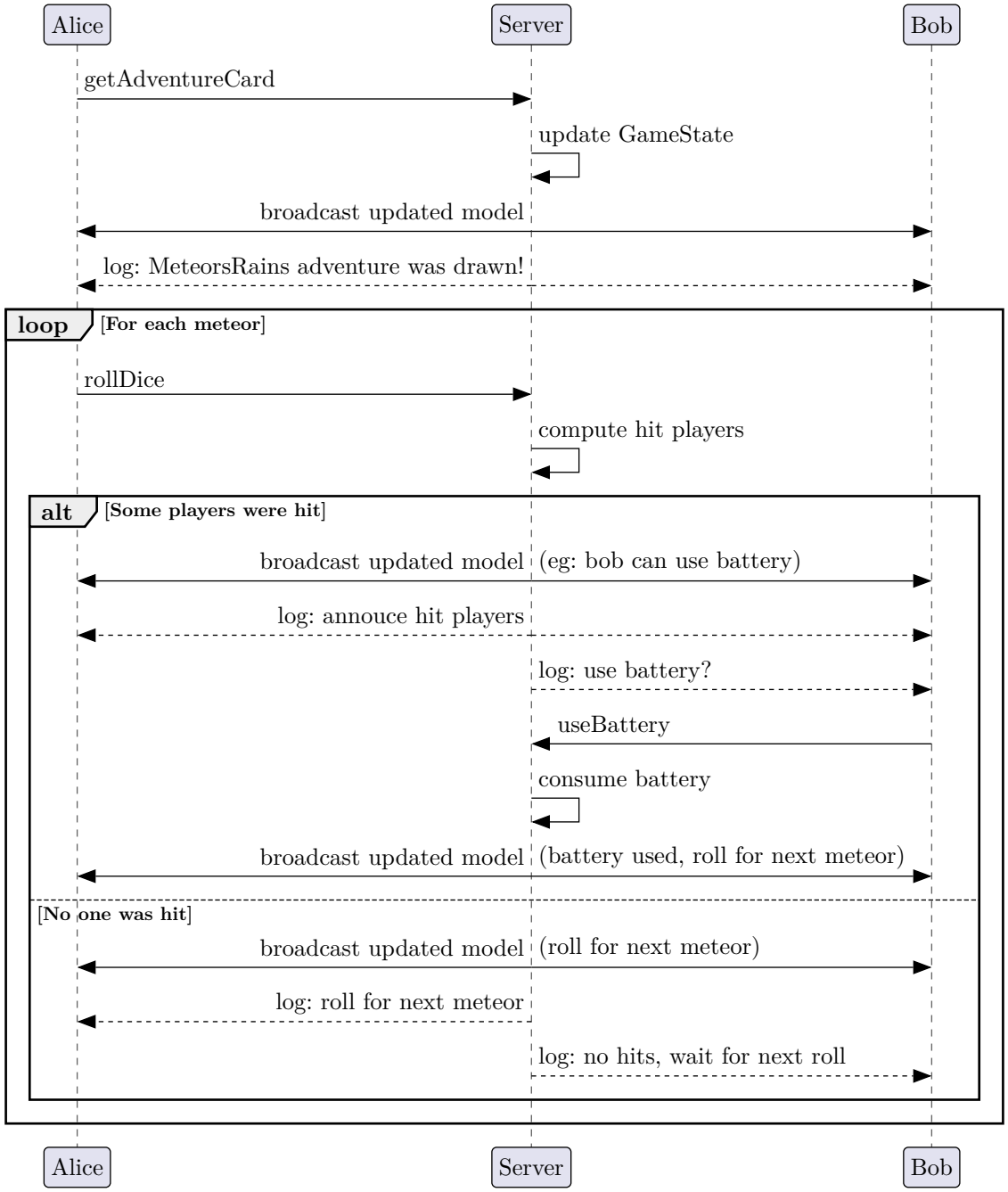


**Figure 4:** Diagram representing typical actions sequence for the `MeteorsRain` state.

It should be noted that, in case of multiple hit ships, each players state within the `GameState` is managed independently therefore all players can concurrently `useBatteries`, `fixShip` and `viewOtherShips`.

### 3.3.2 `OpenSpace` State actions sequence

The state pattern allowed us to manage different states in substantially different ways like in `OpenSpace` state.

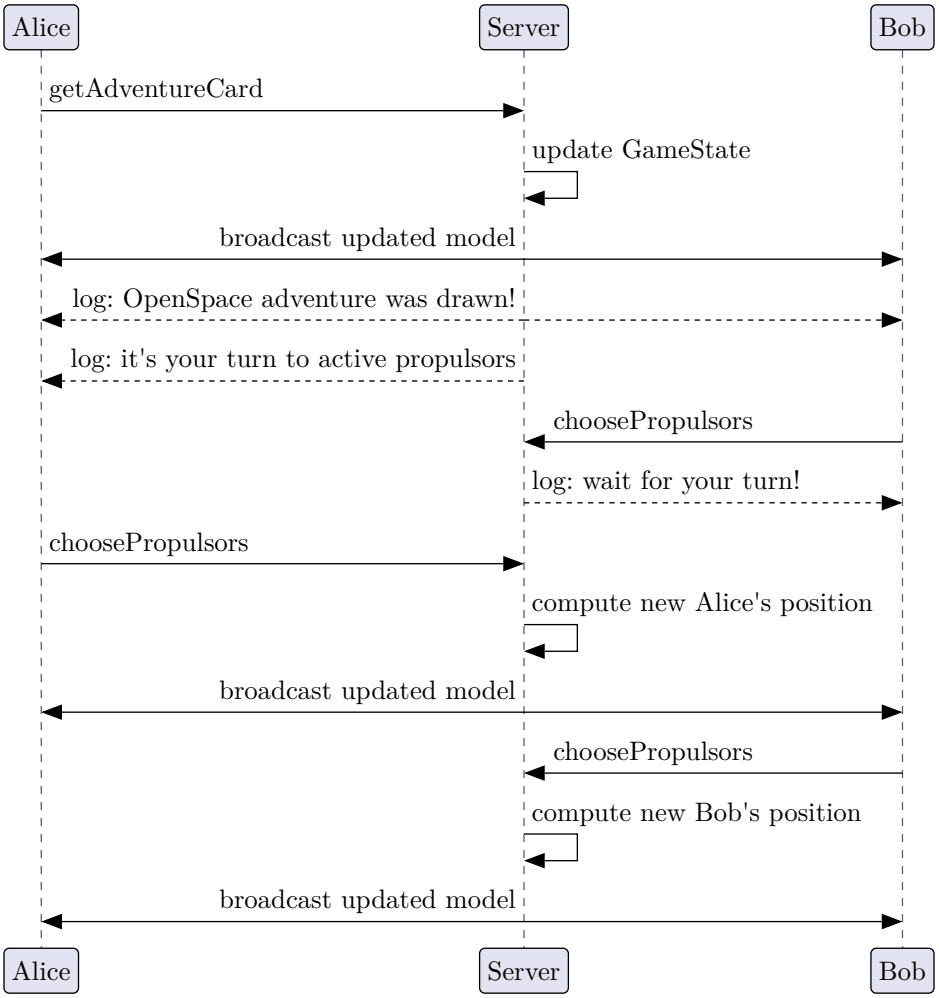


**Figure 5:** Diagram representing typical actions sequence for the `MeteorsRain` state.

Since actions must be taken following flight order, actions for this state are only executed if it's the player turn to act.