Pixformer

Multiplayer Super Mario

- Giorgio Garofalo

Abstract

This report presents the Distributed Pixformer project, made for the 2024-2025 Distributed Systems course at Unibo. This project builds upon the Pixformer project, developed for the 2022-2023 Object-Oriented Programming course at Unibo, crafted in Java with an MVC architecture combined with ECS (Entity Component System), which was selected as a good foundation due to its modularity and maintainability. In this enhanced version it becomes possible to compete against players over the network in a fun Super Mario-like adventure.
Credits for the original project go to: Giorgio Garofalo, Giacomo Antonelli, Nicolò Ghignatti, Luca Patrignani.

AI Disclaimer

```
"During the preparation of this work, the author(s) used GitHub Copilot
to assist with code completion and documentation generation.
After using this tool/service, the author(s) reviewed and edited the
content as needed and take(s) full responsibility for the content of the
final report/artifact."
```

1 Concept

Pixformer is a GUI game designed as a clone of the classic 1985 *Super Mario Bros.*.

Players take on the role of Mario, competing against each other to reach the end flag in horizontal levels, while collecting power-ups and defeating or avoiding enemies along the way.

Use case collection

The users will be interacting with the system while sitting in front of their own computer, connected to the Internet from different locations. Although games will be started sporadically, interactions during a level are highly frequent and happen via keyboard commands, which trigger events, the results of which are displayed on screen.

No data is stored persistently, but a massive amount of information needs to be exchanged.

2 Requirements

Functional requirements

- While in the main menu, the user will be able to select a level to play.
  If a game does not exist yet, it will be created and said user becomes its leader.
  The system must be scalable enough to allow players to join a game at any time.
- Fault tolerance: if a player goes offline he will be kicked out of the game, and will be able to join again. However, if the leader goes offline, the game will end.
- During the game, a player will encounter different enemies, each with their own behavior:
  - Goomba: a small, brown, walking mushroom that can be defeated by jumping on it;
  - Koopa: a walking turtle that retreats into in its shell when jumped on. If a player hops on the shell, it will slide on the ground and damage any encountered entity in its path, including the player itself.
- A player can grab and use different power-ups:
  - Mushroom: increases Mario's height;
  - Fire Flower: allows Mario to shoot fireballs.

  If the player is hit by an enemy and has active power-ups, the most significant one is lost.
- The game world is composed of different block types:
  - Surprise: when hit from below, it produces a power-up;
  - Destructible: when hit from below, it is destroyed;
  - Indestructible: cannot be destroyed.

- Each player has a score, which is earned by collecting coins and defeating enemies. The leaderboard can be seen both during and after a game.
- When the user finishes its game, either by winning or losing, he will be able to spectate the other players, or leave at any time.
- A player's game ends if he:
    - is damaged while in the smallest state;
    - falls into a pit;
    - reaches the end flag.

  A level finishes when all players have ended their game.

Non-functional requirements

- The game must run on the three main operative systems without any major difference;
- The codebase must be maintainable and scalable;
- Latency must be minimal to ensure a smooth experience, while ensuring a strong consistency between players.

3 Design

3.1 Architecture

The project follows the MVC and ECS (Entity Component System) architecture.

Zooming in on the distributed part, the adopted architecture is a slight variation of client-server, the ***leader-follower*** pattern:

- The architecture is centralized by means of a server. However, it becomes an invisible detail for the user.
- When the client attempts a connection to a server and it fails, it sets the server up and connects to it again. By doing so, this client becomes the leader;
- Other clients that join are followers.

This architecture was chosen because of simplicity, efficacy, speed and complexity hiding from the user. An alternative that was thoroughly considered was a peer-to-peer architecture which consisted of one server per user, where all the servers could communicate.
This strategy was discarded because of high complexity and potential latency issues, with the only benefit being a better fault tolerance in case the leader goes offline.

3.2 Infrastructure

- Each player is a client. N players can join a game;
- There is one server per game, located on the same machine as the leader client;
- For development and simplicity purposes, all clients were run on the same machine. However, it is possible for clients to be distributed across the world as long as the leader's firewall accepts connections from outside the network;
- In order to allow different machines in the network to find an open game, an additional server is introduced: the *game finder*, which acts like a dynamic DNS server by internally storing the IP address of the leader clients, associated to the game level they

are hosting. This makes it possible for clients to dynamically discover game servers, as long as they know in advance the address of the game finder server;

- Data isn't stored persistently, and the state of the game is replicated on all N clients. The game finder server stores an IP address only while the corresponding game is running, in a volatile way;
- No authentication is performed, mutual trust is assumed;
- Communication is performed via a *publish-subscribe* pattern: since the game is heavily event-based, clients will mainly announce the trigger of an event to the server, which may process it and eventually forward it to the other clients, hence clients are both publishers and subscribers;
- Entity mapping:
    - The game loop is performed on each client. Since the leader's client also overlaps with the server, it is the source of truth which periodically sends alignment updates to clients (see *Interaction*).
    - The rendering process is performed on the client side.

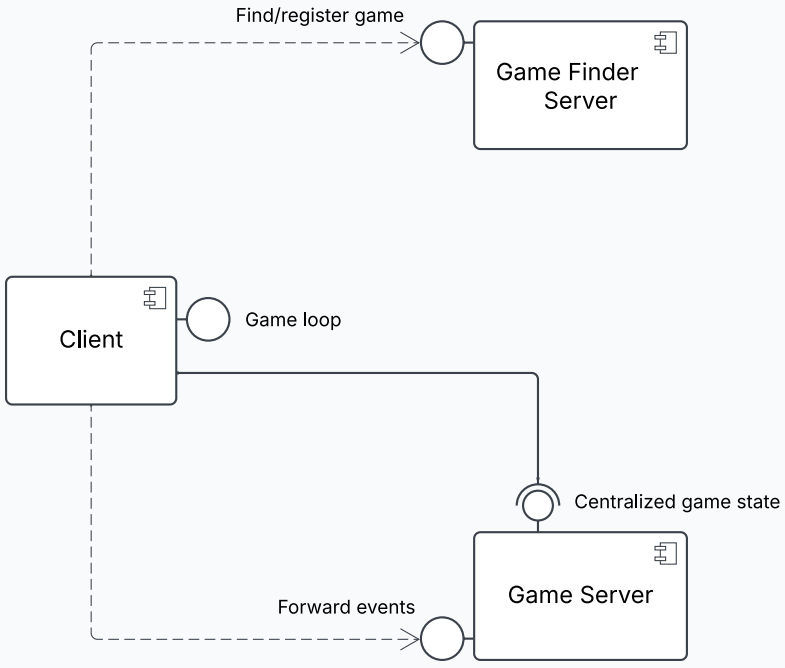


Figure 3.1: Component diagram of the infrastructure.

3.3 Modelling

The entry point of the distributed enhancement of the project lies in the `ServerManager`, which acts as a bridge between client and server which keeps track of the following information:

- The current WebSocket session with the server;
- Online players;
- Player index assigned to the client;
- Reconciliation routine;

and performs the following operations:

- Connects to the server, and also starts it if this is the leader client;
- Initializes the reconciliation routine;
- Gracefully disconnects from the server;
- Dispatches commands on specific players;

This lets the architecture of the original project stay untouched, aside for the new `ServerManager` component provided by the MVC `Controller`:

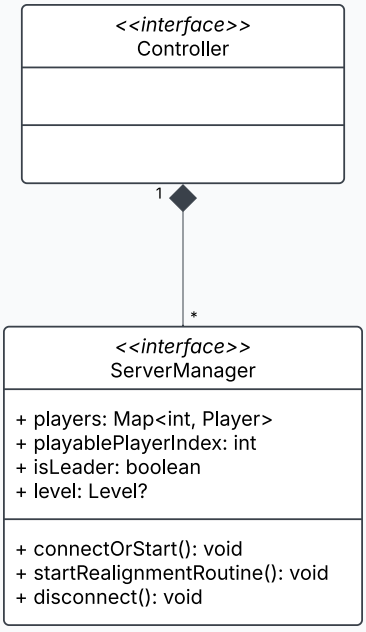


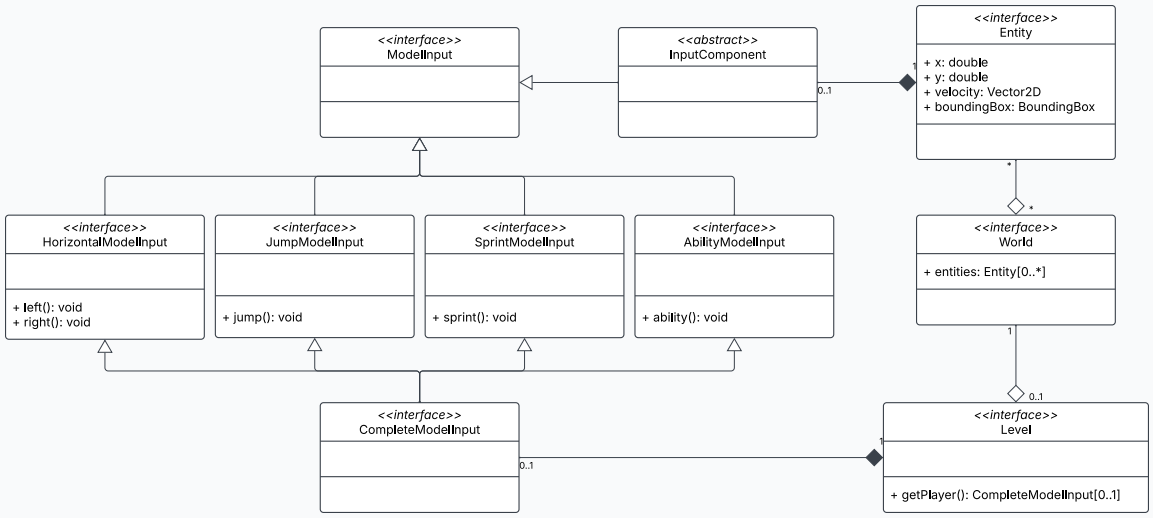
Figure 3.2: Position of ServerManager in the MVC architecture.

3.3.1 Entities

The domain entities are:

- Players;
- Enemies (Goomba, Koopa);
- Power-ups (Mushroom, Fire Flower);
- End flag;

As defined by the Entity Component System (ECS) architecture, each entity is defined by a set of components: input, physics, collision and graphics. For this report, only the input component is relevant, as it defines the reaction of the entity from a user input.



For instance, a Goomba's `InputComponent` may just be a `HorizontalModelInput`, as no complex movement is needed. A `Player`, instead, requires a full `CompleteModelInput` to use all his features.

Why is the user input particularly important here? Let's say our client has to send information to the server each time our player moves, jumps or does any other action. The most maintainable, invisible and elegant solution would be to take the already existing `InputComponent` of the player, wrap it into a *decorator* and inject it as the entity's new input component.

The decorator performs the exact same operations as the wrapped input (*proxy* pattern), while also sending messages to the server at each performed action:

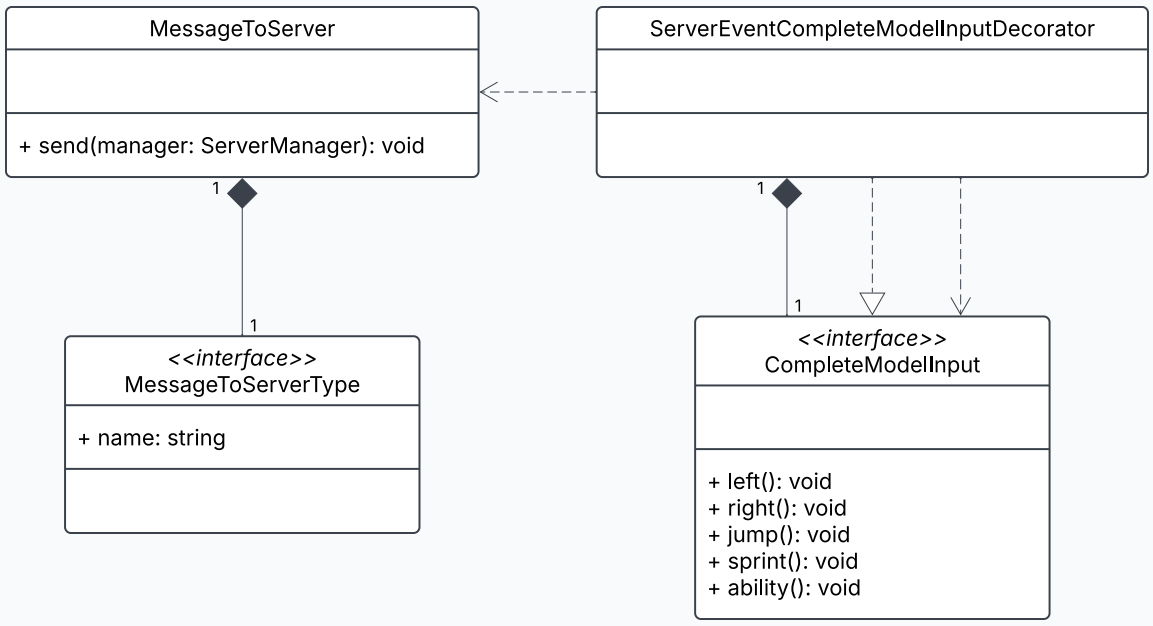


Figure 3.3: Class diagram of the input decorator.

3.3.2 Events

Domain events are:

- Player:
    - joins;
    - quits;
    - moves horizontally;
    - jumps;
    - defeats enemy;
    - collects coin;
    - gets power-up;
    - shoots a fireball;
    - hits surprise/destructible block from below;
    - ends his game (by either enemy damage or falling);
- Enemy:
    - damages player;
    - changes state (e.g. Koopa retreats in its shell);

Commands

The subset of events that are directly triggered by user inputs are modeled by a `Command` class:

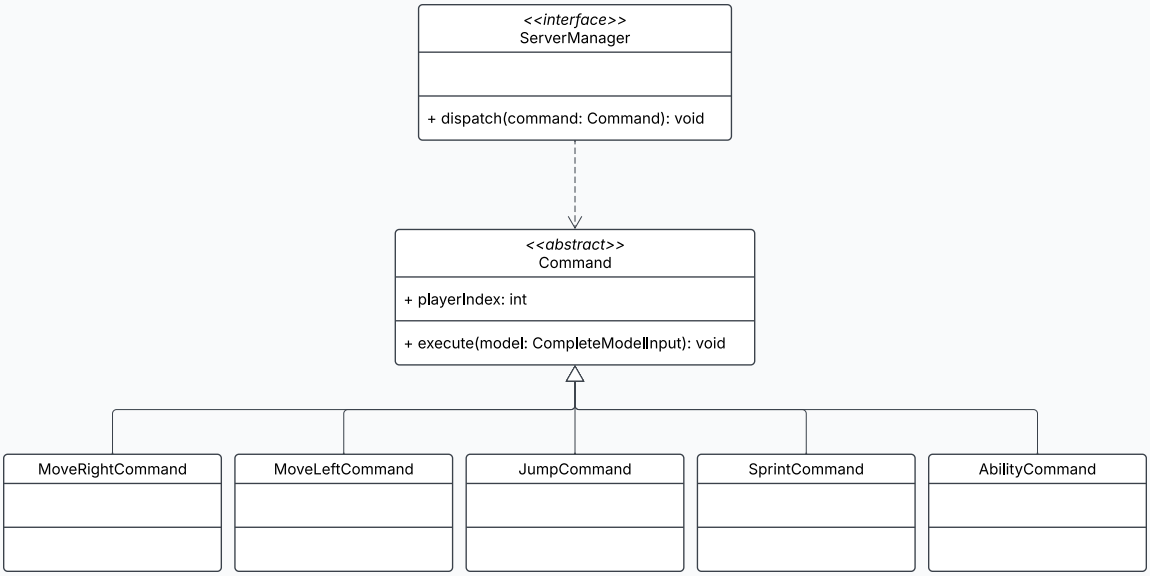


Figure 3.4: Class diagram of the command hierarchy.

Commands must be serialized and deserialized in order to be exchanged:

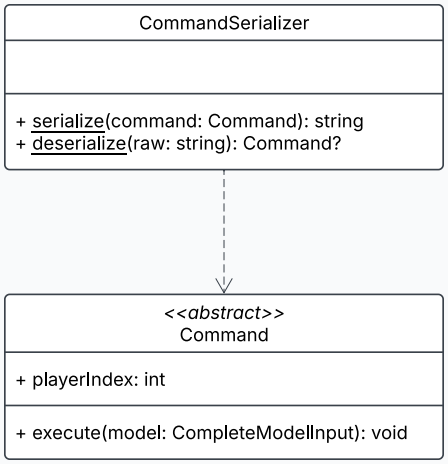


Figure 3.5: Class diagram of the command (de)serializer.

Aside for alignment messages (see *Interaction*), these events represent the only kind of message exchanged between server and clients.

3.4 Interaction

The system adopts a ***client-side prediction and server reconciliation*** pattern. The following subsections will describe its components in detail.

Game finding

When a client chooses a level to play, the game finder server is reached to look for any active games on the same level, or to create one otherwise.

The game finder server is not strictly needed: in case a connection cannot be established, the system assumes to join or create the game on the local machine.

1. Assuming the game finder server is up and running, the client attempts at retrieving the IP associated with the selected level;
2. If found, the server at said address is reached to join the game. If not found, the client, which is now the leader, asks the game finder to register a new game associated to its IP address;
3. When the game server is shut down, it is removed from the game finder.

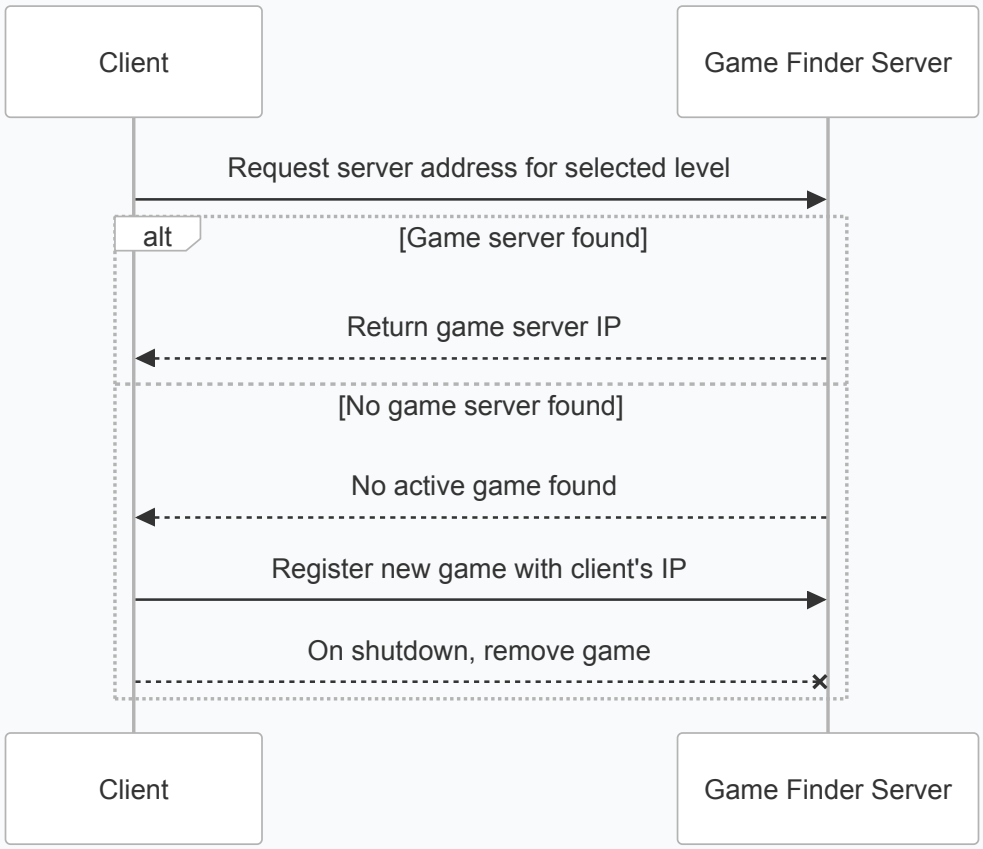


Figure 3.6: Sequence diagram of game finding.

Handshake

As soon as a new client connects to the game server, a handshake is required in order to assign the player index to the new client. This index is incremental and acts as an identifier for the client. When the handshake is complete, the client can start playing.

1. At connection time, a *hello* message is sent from the client to the server;
2. The server decides the client's index and sends it as a response.

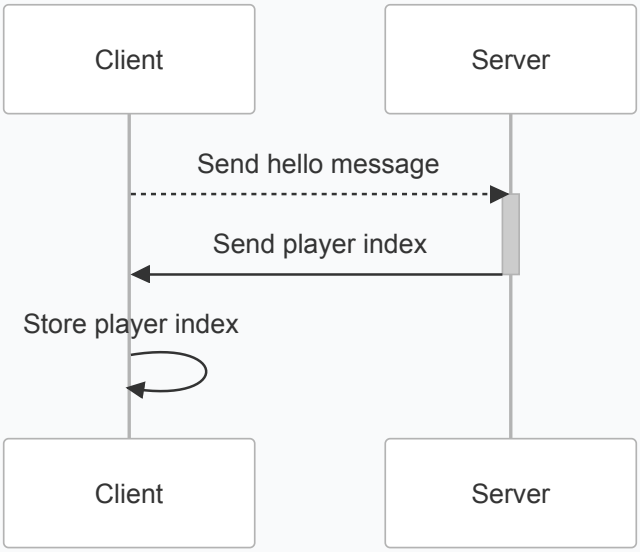


Figure 3.7: Sequence diagram of client-server handshake.

User input events

1. The client processes user inputs (player movement, jump, sprint and ability) immediately in order to provide a responsive and smooth experience, updating its local game state independently. The event message, along with the player identifier, is sent to the server;
2. The server receives the event, updates its internal game state and forwards, via broadcast, the event message to the other clients, which update their game state as well;

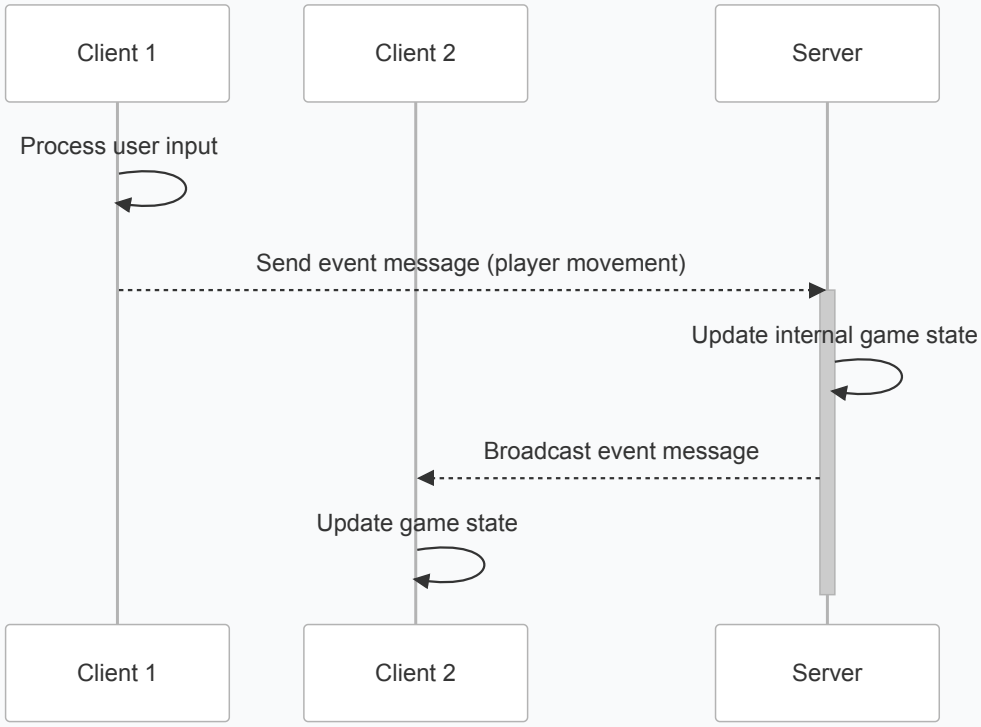


Figure 3.8: Sequence diagram of input events.

Game state reconciliation

Each client periodically sends a reconciliation request to the server, the authoritative source of truth, which sends its internal game state as a response. This ensures consistency is kept across clients, which may otherwise be compromised by latency or packet loss;

Since the server matches with the leader's client and they share the same game state, reconciliation is not performed on the leader's client.

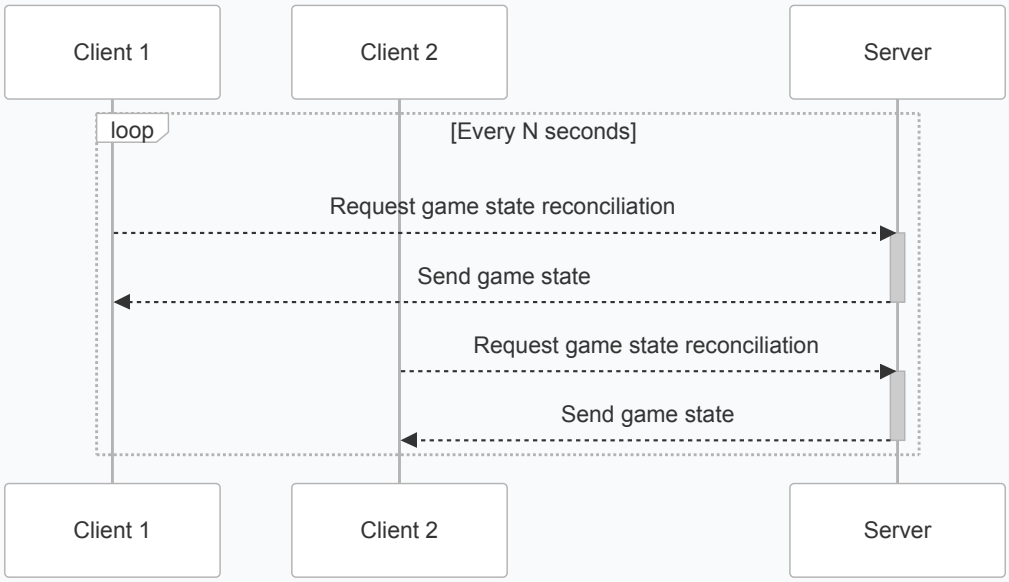


Figure 3.9: Sequence diagram of the game state reconciliation.

3.5 Behaviour

The client-side of the system is heavily stateful:

- The main menu is displayed at the beginning. Choosing a level starts the game;
- The player can hit a power-up item, mushroom or fire flower, to absorb that specific power and obtain points;
- When a surprise block is bumped from below, a power-up item is spawned once. After that, the block is deactivated;
- The player can hit coins to collect them;
- The player can jump onto enemies to obtain points and alter the enemy state:
    - A Goomba disappears;
    - A walking Koopa transforms into a shell Koopa;
- When the player is touched by an enemy:
    - If his current power-up is fire flower, it is brought down to mushroom;
    - If his current power-up is mushroom, it is brought down to the smallest state;
    - If he is in the smallest state, his game is over;
- If the player falls into a pit, his game is over regardless of any active power-up;
- If the player hits the end pole, he is awarded 1000 points and his game is over;
- When a player's game is over, the user will be able to spectate the other players as long as there are any;
- At any time the user can exit the game and go to the main menu.

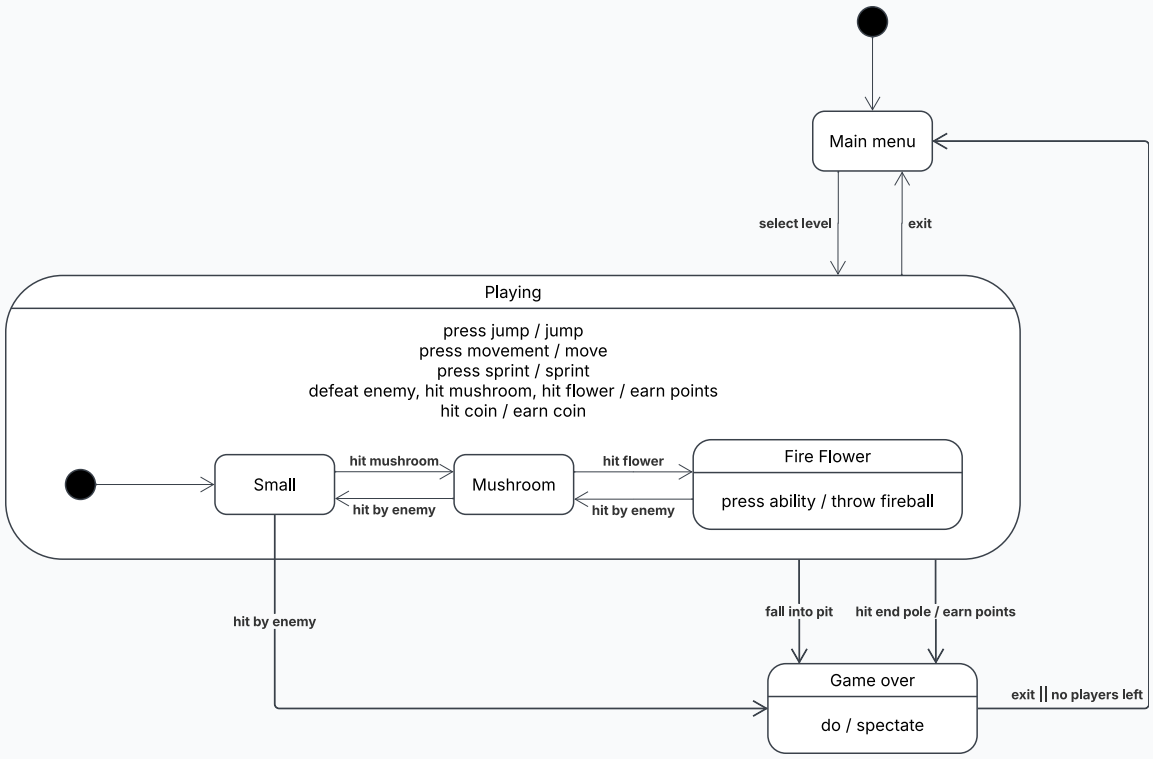


Figure 3.10: State diagram of the player lifecycle.

The server, instead, is stricly stateless. However, if more generically seen as the union of the server and the leader client, it acts as the authorative, stateful, source of truth being in charge of the updates.

The game finder server is stateless, as it serves as a volatile storage.

3.6 Data and Consistency Issues

Although no peristent data is stored, volatile game data is shared and replicated between clients. Consistency is guaranteed through periodic reconciliation with the server, as already mentioned.

3.6.1 Fault-Tolerance

- The game state is replicated on each client with its own game loop, prioritizing availability over consistency in order to provide a smooth experience;

- If a client fails the reconciliation request twice in a row, meaning the server cannot be reached possibly because the leader player has quit the game, it is kicked out to the main menu;
- Heart-beating and timeout are used, as provided by the framework (see *Implementation*);
- If a client quits the game or crashes, its corresponding player entity is removed from the server and clients;
- In order to avoid relying heavily on yet another component, the game finder server is optional — although encouraged to use. If the server is not accessible, the system will look for games on the local machine.

3.6.2 Availability

In case of network partitioning, the system prioritizes availability over consistency. In case a client goes offline for a short time, its user input is still processed immediately to provide a fast response and smooth experience. As soon as it comes back online (assuming it wasn't already kicked out due to timeout), the reconciliation will align its game state to the server's.

4 Implementation

- Almost every interaction happens in a persistent WebStocket session between the server and each client.

    - In-transit data is represented using a simple text-based format: `n|event`, with `n` being the index of the player triggering the event, and `event` the event type. For example `2|jump` signals player 2 jumping.
    - These are by far the most common exchanged messages, thus having a simple-to-deserialize format is crucial for performance reasons.

- At handshake time, the new client sends a `hello` text frame, and then waits for the next incoming frame. The server then sends the player index as a text response, which the client reads and stores as an integer. At this point, the client begins the persistent session.
  This is handled by the `PlayerConnectMessage` class.

- Upon receiving an event message, the server does nothing but broadcasting it to all the clients.

    - When a client receives the event, it deserializes the message and performs the action on the player with the given index;
    - However, because of the leader-follower pattern, one of the clients shares the same game loop and state with the server, hence updating the centralized game state, acting as the single source of truth.

- Ping (heart-beating) and timeout parameters match to 10 seconds: if a ping to the server fails, the session is closed;

- Client-to-server event messages are triggered by the `ServerEventCompleteModelInputDecorator` that wraps the original player's input component, while also sending the corresponding message, on the active WebSocket session, for each action:

```
class ServerEventCompleteModelInputDecorator(
    private val modelInput: CompleteModelInput,
    private val serverManager: ServerManager,
) : CompleteModelInput {
    override fun jump() {
        modelInput.jump() // Invokes the original action
        MessageToServer(PlayerJumpMessage).send(serverManager)
    }
    ...
}
```

- Periodic reconciliation of a client to the server's game state happens via an HTTP GET request every 3 seconds;

    - The level state is represented using JSON, adopting the same structure used for loading levels from files:

```
{
  "name": "Level 1",
  "spawnPointX": 5,
  "spawnPointY": 2,
  "scores": {
    "1": {
      "points": 2,
      "coinsNumber": 10
    },
    "2": {
        "points": 5,
        "coinsNumber": 20
      }
  },
  "entities": [
    {
      "type": "player",
      "x": 5.0,
      "y": 2.0,
      "velocity": {
```

```
  "x": 0.1,
          "y": 0.0
        },
        "playerIndex": 1,
        "powerup": "Mushroom"
      },
      ...
    ]
}
```

- The game server runs by default on port `8082` and features two endpoints:
  - `/ws`: endpoint for WebSocket exchanges;
  - `/realign`: HTTP GET endpoint for fetching the centralized game state.

- The only significant change brought to the original Java project is the `ControllerServerManager` link. This is handled via a convenient application of the *observable* pattern on the current level, set up during the initialization of the controller:

```java
// ControllerImpl.java

getLevelManager().addOnLevelStart((level, playerAmount) -> {
    this.getServerManager().connectOrStart();
    level.init();
    this.getGameLoopManager().start();
});

getLevelManager().addOnLevelEnd((level, leaderboard) -> {
    this.getGameLoopManager().stop();
    this.getServerManager().disconnect();
});
```

Game finder

The game finder server is a web server, running on port `8083`, with the following endpoints:

- `/check` (GET): used to check the availability of the server;
- `/get?name=<name>` (GET): returns the IP of the server associated with the given level name, if it exists;
- `/add?name=<name>&ip=<ip>` (POST): registers a name-address mapping, if an address is not already present for the given name;
- `/remove?name=<name>` (POST): removes the entry associated with the given name.

To communicate with the server, the client uses an agent that models said operations. In order to tolerate the unavailability of the game finder server, two distinct implementations of the agent are provided:

- One that communicates with the server via HTTP;
- A "mock" implementation that does not perform any add/remove operation, and always returns `localhost` as the game address. This implementation is used if the server is unreachable.

This implementation switching mechanism offers the caller an opaque representation of the underlying process.

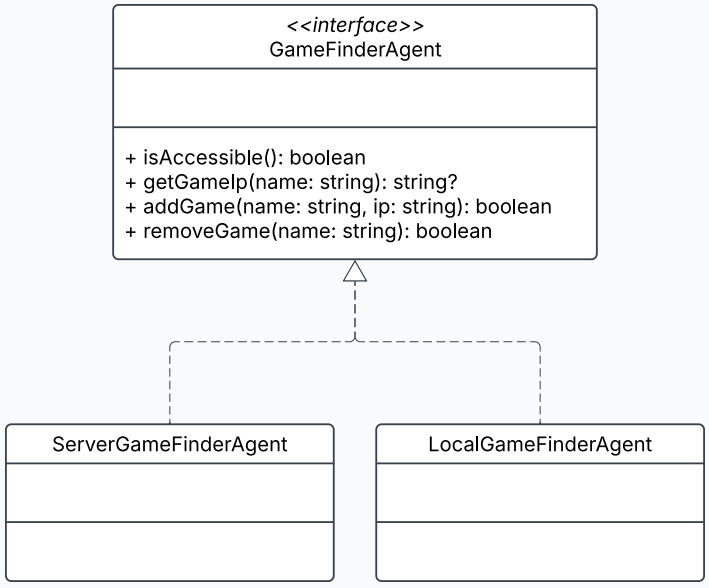


Figure 4.1: Class diagram of the game finder agent hierarchy.

4.1 Technological details

- The base game was developed in Java;
- The distributed system was developed in Kotlin, with the Ktor framework, used in combination with Netty;
- The WebSocket protocol does not natively provide a broadcast feature, which was instead made possible by using Kotlin Coroutine's shared message flows, which collect and store each session;
- Ktlint was used to ensure a consistent code style across the Kotlin codebase.

5 Validation

5.1 Automatic Testing

Components were unit-tested by creating a mock, simplified environment. Tests can be run via `./gradlew test`.

- *Game* tests (unimportant for this report's purposes)
    - Tested the correct execution and dynamics of the game loop and interaction between game entities.
- *Client-server* tests
    - The purpose is testing the communication and interaction between two independent clients and the game server;
    - All three components are represented by three `ServerManager` fields, hence testing the opaqueness that allows for the leader-follower pattern;
    - Connections are tested by ensuring the server has the correct amount of active players after both clients connected;
    - Handshake is tested by checking for an incremental assignment of the player index for each client;
    - Event broadcasting and dispatching of their corresponding command are tested by letting one client send an event message to server, to make sure the other client receives the corresponding command (e.g. a `PlayerJumpMessage` corresponds to a `JumpCommand`);
    - Disconnection is tested by disconnecting one client and making sure the number of active players decreased.
- *Command* tests
    - The purpose is testing the serialization and deserialization of commands to be sent across components;
    - Correctness is tested by serializing and deserializing a command, ensuring the before and after represent the same information.

- *Game state* tests
    - The purpose is testing the JSON serialization and deserialization of the game state that is sent in response to reconciliation requests;
    - Correctness is tested by serializing and deserializing a level, ensuring the before and after represent the same information;
    - Critical aspects include de-serialization of entities, including both common information (position and velocity) and entity-specific (e.g. player's current power-up, whether a surprise block was deactivated, etc.), along with players' score.
- *Game finder* tests
    - The purpose is testing for the correct interaction between a client and the game finder server;
    - The game finder server is started, and a `ServerGameFinderAgent` is used to make requests to the server;
    - Retrieval and updating is tested by adding games, retrieving their address, and removing them, including the corner case of duplicate entries.

5.2 Acceptance test

Manual testing was performed for:

- Update of the game world after receiving the centralized game state: the update of the game world is strictly bound to the game loop and its command queue, making it difficult (although not impossible) to test automatically;
- Triggering of event messages on player actions: this is tightly related to user input and interaction, which is more intuitive when tested manually.

Please note that the system was manually tested, on a single machine, on macOS 15 and, minimally, on Windows 10. The original game was also tested on Linux.

6 Release

The project is structured into two modules:

- The game (root module), divided into two distinct parts: the `java` source root houses the original base project, containing the front-end and the entire game logic, while the `kotlin` source root contains the distributed features. These two parts are connected by the `Controller` (present in the Java part).
  Since the leader-follower pattern is used and the concepts of client and server overlap, splitting the two components into two different modules was not needed.
- The game finder server (module `gamefinder`), extremely small and compact.

The JAR executables are available for download on GitHub Releases, or can be built from source (see *Deployment*).

7 Deployment

1. Build:
   `./gradlew clean build`
2. Start game finder server (optional, but suggested):
   `java -jar build/libs/gamefinder-1.0-SNAPSHOT-all.jar`
3. Launch game:
   `java -jar build/libs/pixformer-1.0-SNAPSHOT-all.jar`

A game window will open from the main menu. Games can be joined from there.

8 User Guide

Step 0 (optional): launching the game finder server enables finding games from the network. If the game finder server is active on another machine, you can pass its address (only IP, not port, which is fixed to 8083) as the only command-line argument.

After launching the game, the main menu is displayed. From there, the user can choose a level to play (only one is provided: *Level 1*).

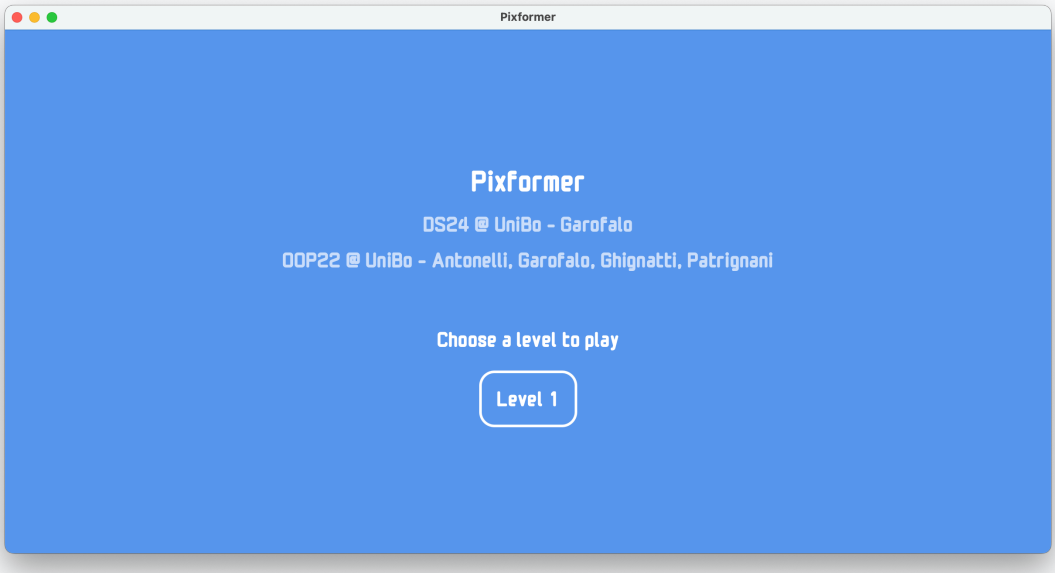


Figure 8.1: The main menu

Jump	Move right	Move left	Ability (fireball)	Sprint	Go to menu
W/Space	D	A	Shift	CTRL	ESC

Table 8.1: In-game key bindings

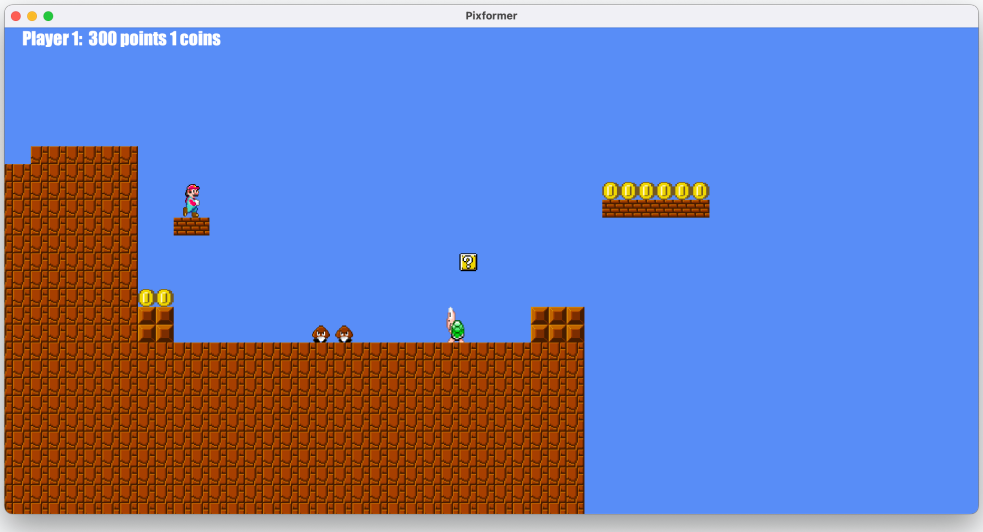
Other clients can subsequently join the same level.

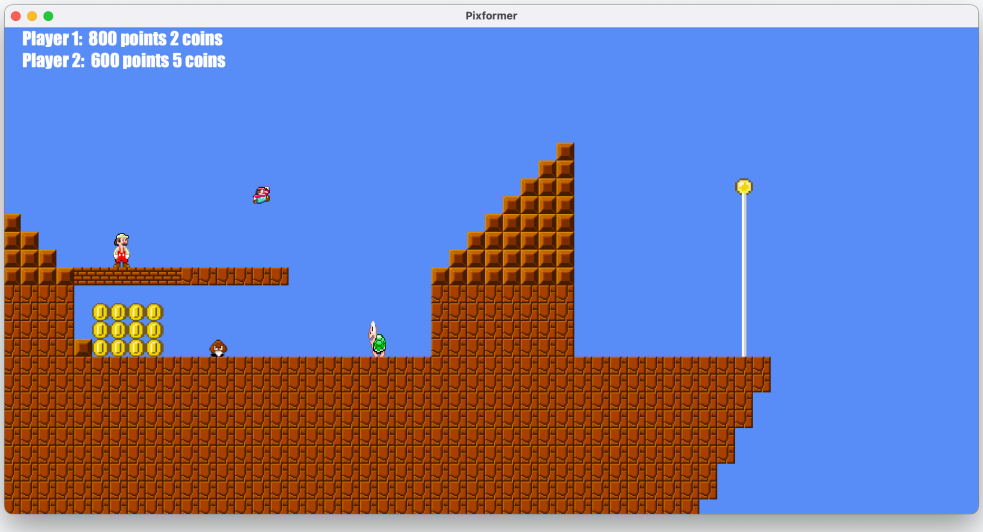
The game follows the same behavior as the original *Super Mario Bros.*: long-pressing the *Jump* key increases the jump height, and horizontal movements are affected by inertia.

In order to get to the end of the level (the end pole), you can:

- Hit, jumping, surprise blocks from below to spawn power-ups. Collect power-ups to gain new powers;
- Jump on enemies' heads to defeat them.

> **Note.** Remember that going to menu as the leader client shuts the server down.





9 Self-evaluation

Garofalo Giorgio

I enjoyed working on this project, as it involved a lot of pre-planning to give life to a maintainable, modular product. I put a considerable effort into this project, and I feel satisfied with the final product in its integrity.
I find its strengths in the ease of connecting with other clients and smoothness of the gameplay, while I find its biggest weakness in the sometimes-big difference between what the client sees and the game state received from the reconciliation, meaning there is some discrepancy between the direct user input and what the server evaluates. I struggled, unfortunately, to find a solution to this issue.

Taking on an already big project (credits go to my original team), an intriguing challenge was finding out how to connect the 'old' with the 'new' while altering as few elements as possible from the legacy codebase, while teaching me a lot about distributed systems.