Alma Mater Studiorum - Università di Bologna

Campus di Cesena

Raga.io

Grow, eat and survive in a distributed chaos.

Distributed System Course

A.A. 2024/2025

Authors:
Elvis Perlika
Eleonora Falconi

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Disclaimer

During the preparation of this work, the authors used Chat-GPT to refine the report by improving sentence structure and correcting grammatical errors. After using these tools, the authors reviewed and edited the content as needed and takes full responsibility for the content of the final report.

1.2 Abstract

Agar.io is an online, *massively multiplayer* action game. Players take on the role of a small, circular cell inside a map that resembles a Petri dish. The primary goal is to grow as large as possible by consuming smaller cells, both those controlled by other players and those that are scattered randomly throughout the game world as food. This simple premise leads to a dynamic and competitive environment where a player's size directly dictates their power and vulnerability. Players can join a randomly assigned game session, create a new session, or join an existing one using a unique session ID. The session ends only when the player is eaten or decides to quit.

The goal of this project is to design and develop a clone of Agar.io with a robust client–server architecture, ensuring scalability, performance, and smooth multiplayer interaction.

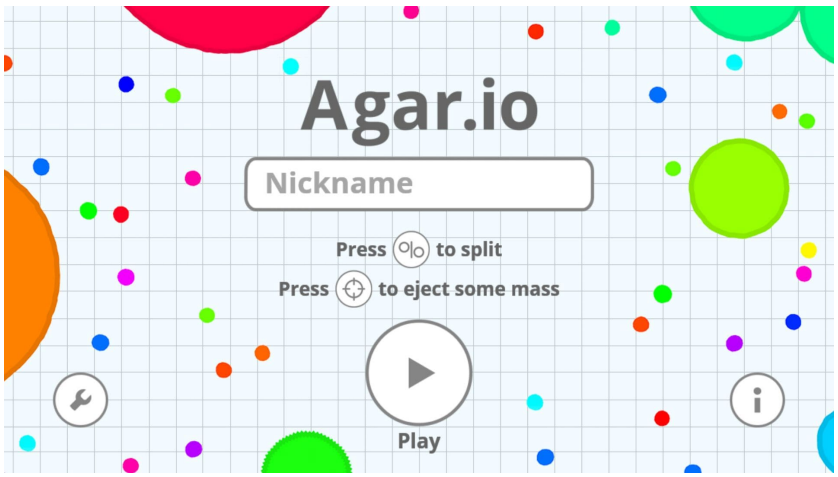

Figure 1.1: Agar.io login page

Achievements

- Developed a real-time client–server architecture supporting multiple concurrent players.
- Implemented smooth synchronization and communication protocols to minimize lag.
- Designed an efficient game loop for rendering, collision detection, and player interaction.
- Created scalable server logic capable of handling large numbers of simultaneous connections.
- Implement robust backup mechanisms to reliably save and restore game sessions in case of server failure.
- Implemented core game mechanics: cell growth, collision handling, food spawning, and player elimination.
- Delivered a competitive multiplayer experience close to the original Agar.io.

Chapter 2

Requirements

2.1 Concept

This project is a desktop application with a graphical user interface (GUI) packaged as a Java Archive (JAR) file. It can be executed on any system with a Java Runtime Environment (JRE) installed.

- **Primary Users**: The game is designed for casual gamers who enjoy competitive multiplayer experiences. Players can be located anywhere worldwide as it is built for online play.
- **Usage Patterns**: Sessions are typically short and it is common to play this game during leisure time, ranging from just a few minutes up to an hour.
- **Interaction Method**: Players interact through a graphical user interface (GUI) on desktop computers, using a mouse to control their in-game cell.
- **Data Handling**: No personal data are stored. The system only maintains temporary session data (e.g., player positions, scores), which is discarded once the session ends.
- **User Roles**: There is only one role: **Player**. All users share the same capabilities, with no role-based differences or special permissions. It is possible to integrate bots to fill rooms with few players.

2.2 Functional Requirements

1. The player can join a randomly generated game session.
2. The player can control the movement of their cell by directing it toward the mouse cursor.
3. The player can create a new game session.
4. The player can join an existing session by providing its unique session ID.
5. The system must support multiple concurrent players within the same session.
    - **Acceptance criteria:** support for at least 5 concurrent players per session.
6. The system must provide real-time visibility and interaction between players.
7. The system must ensure low-latency handling of movements and gameplay actions.
    - **Acceptance criteria:** maintain a minimum frame rate of 30 FPS under typical load.
8. A mechanism must exist for consuming smaller cells (players or food) to increase size.
    - **Acceptance criteria:** larger cells can consume smaller ones, and growth is visually observable.
9. On cells size increase the speed must be reduced by a percentage.
10. The system spawns food items randomly within the map.
11. The system performs collision detection and resolves outcomes based on relative cell sizes.
12. A game session ends when a player is eliminated or exits by closing the game window.
13. Players need to have different colors.

14. The graphical user interface must provide intuitive session-management controls, including:
    - A button to join a random session.
    - A nickname input field.
    - A button to create and join a new session.
    - An input field to enter a session ID for joining an existing session.
    - A button to join the specified session.
15. Visual representation must be provided for player cells, food items, and other players.
16. Player nicknames must be displayed above corresponding cells.
17. The session ID must be displayed so players can share it with others.
18. It's possible to see the number of players in the current session.
19. The system must rely on a scalable server architecture capable of supporting increasing numbers of sessions and players, including:
    - A master server that coordinates multiple game servers, each managing an individual session.
    - Dynamic allocation of players across game servers based on load and availability.
    - Horizontal scalability to support growth in both active sessions and concurrent players.
    - Backup and failover mechanisms ensuring session continuity if a game server fails.

Chapter 3

Design

3.1 Architecture

This project employs a **client-server architecture** with a distributed system design.

3.2 Infrastructure

3.2.1 Deployment

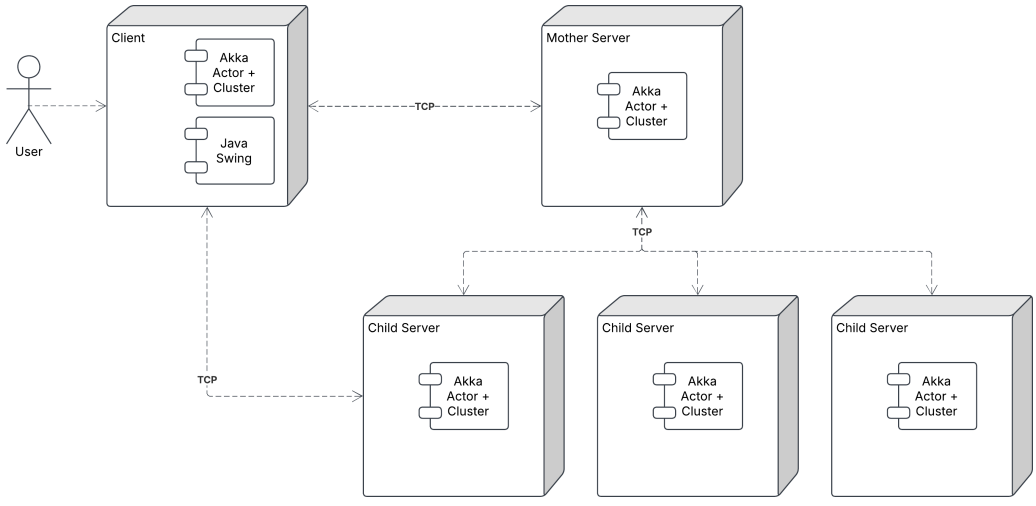

Figure 3.1: Deployment architecture

The system architecture consists of three main components:

- **Client**: The client application is a desktop GUI that players use to interact with the game. It handles user input, renders the game state, and communicates with the servers.
- **Mother Server**: The Mother Server acts as a central coordinator. It manages game sessions, handles player matchmaking, and distributes players to Child Servers based on load and availability. It store Child Server's backups to recover a failed one.
- **Child Servers**: A Child Server manage one game session. It handles real-time game logic, player interactions, and communicates with the Mother Server for session management.

3.2.2 Components

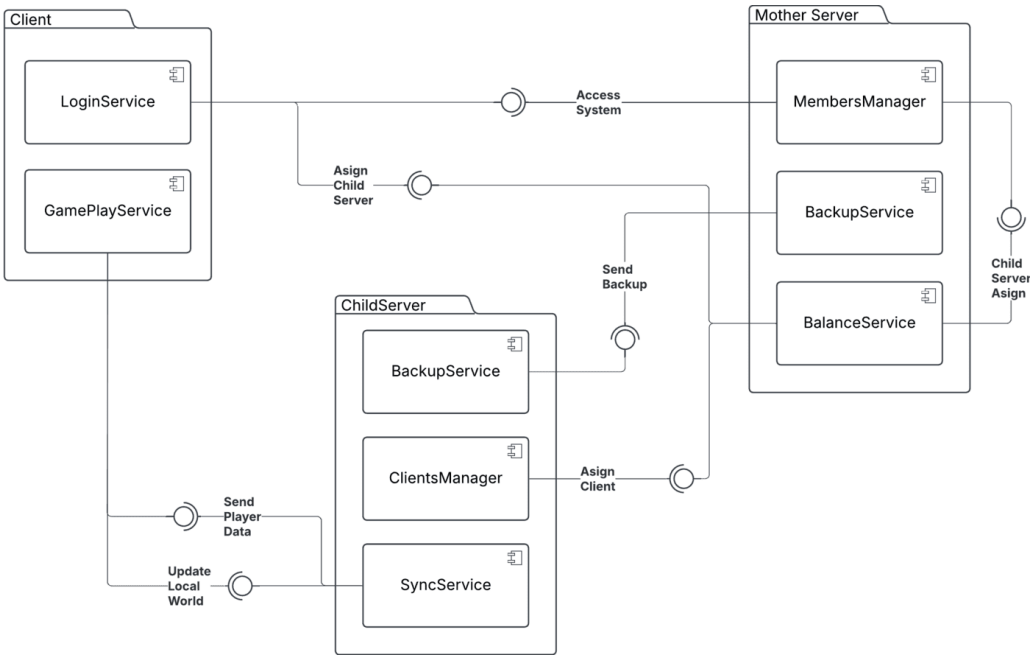

Figure 3.2: System components overview

Client

- **LoginService**: Handles discovery and connection to the Mother Server. It manages connection status, matchmaking requests, and error conditions such as service unavailability.

- **GamePlayService**: Processes all gameplay-related interactions. It sends player inputs to the assigned Child Server, receives authoritative world updates, and coordinates local rendering and state synchronization.

Mother Server

- **MembersManager**: Monitors the cluster, detecting Client and Child Server availability. It forwards connection and disconnection events to the Mother Server.

- **BackupService**: Receives periodic backups from Child Servers. These snapshots allow the Mother Server to restore a Child Server in case of failure, ensuring continuity of a game session.

- **BalanceService**: Distributes clients across Child Servers according to their current load. This enables horizontal scaling and prevents performance bottlenecks.

Child Server

- **BackupService**: Periodically saves the authoritative world state and active players, sending snapshots to the Mother Server to enable fast recovery after failures.

- **ClientsManager**: Maintains the mapping between player IDs and client actors, handling joins, leaves, eliminations, and routing updates to the correct players.

- **SyncService**: Processes player inputs, updates the world at each tick, and broadcasts the new authoritative state to all connected clients to keep gameplay consistent.

3.3 Modelling

3.3.1 Client

The client is structured according to the *Model-View-Controller (MVC)* architectural pattern, which separates the application into three interconnected components: the Model, the View, and the Controller.

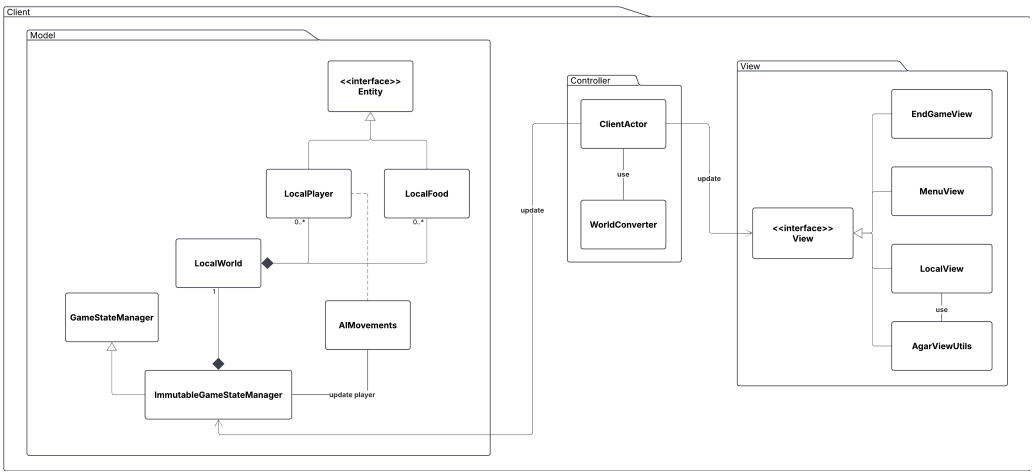

Figure 3.3: Client class diagram

Client Model

The Model encapsulates the game logic and state management:

- **Entity (interface)**: A common abstraction implemented by `LocalPlayer` and `LocalFood`.
- **LocalPlayer and LocalFood**: Entities within the game world.
- **LocalWorld**: Represents the game environment, containing multiple players and food objects.
- **GameStateManager** and **ImmutableGameStateManager**: Handle the game state. The immutable version provides read-only access to the state.
- **AIMovements**: Updates the behavior and movement of AI players, if present.

Client Controller

The Controller handles interactions and updates between the model and the view:

- **ClientActor**: Central controller component managing the game flow and communication with servers.
- **WorldConverter**: Used by `ClientActor` to transform model data into a format suitable for local computation.

Client View

The View presents the game to the user:

- **View (interface)**: General abstraction for different visual components.
- **MenuView, EndGameView, LocalView**: Different visual representations of the game's state (menu, in-game, end screen).
- **AgarViewUtils**: Utility functions used by `LocalView` to draw game elements.

3.3.2 Mother Server

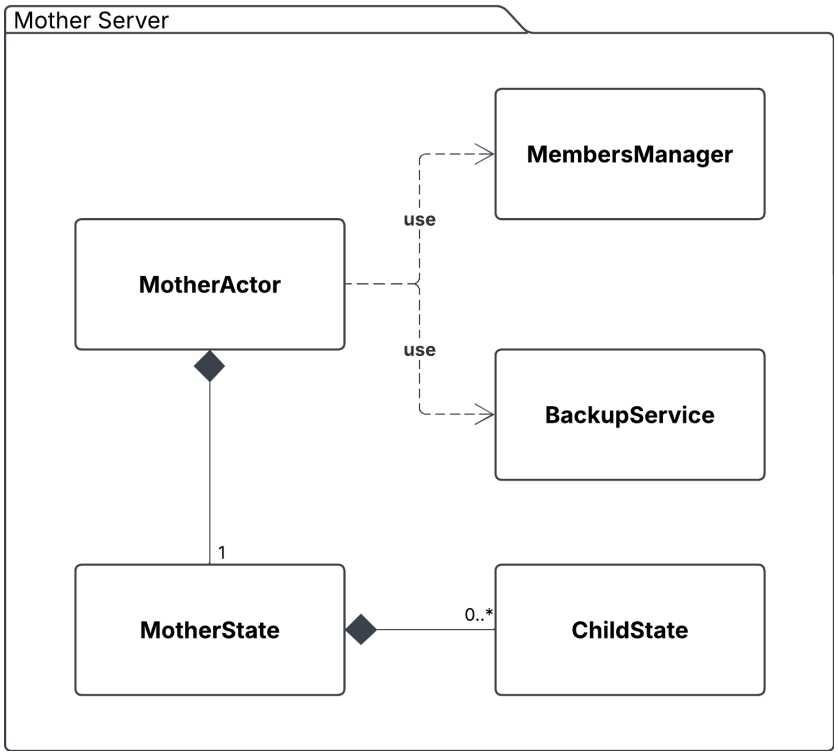

Figure 3.4: Mother class diagram

The Mother Server coordinates the main management logic:

- **MotherActor**: The main actor of the system. It's the Single Point of Failure.

- **MembersManager**: Discovers Clients and Child Servers that access the system.
- **BackupService**: Manages data backup operations.
- **MotherState**: Stores references to Child Servers and pending Clients.
- **ChildState**: Represents the state of a Child Server, including its load and the world's ID it manages.

3.3.3 Child Server

Child Servers manage individual game sessions and receive players from the Mother Server.

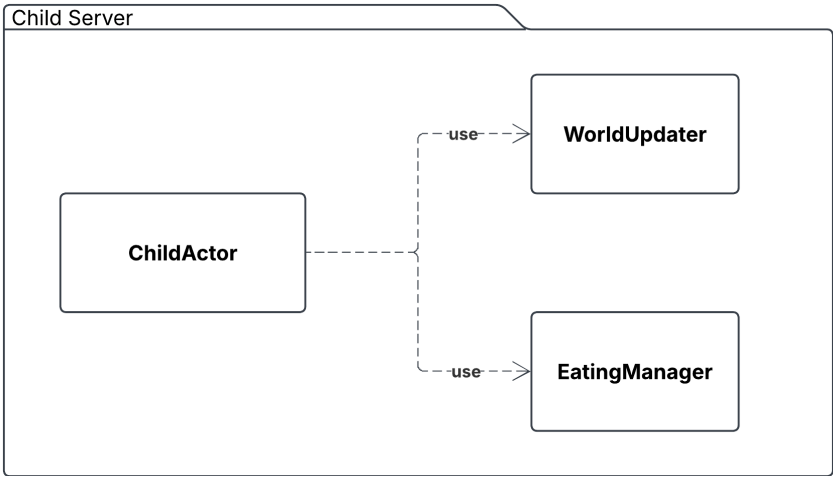

Figure 3.5: Child class diagram

It's composed by:

- **Child Actor**: Receive Player's input, update the world state and sent it to all players.
- **World Updater**: Compute all world transformation based on the Player's input.
- **EatingManager**: Utility class used to compute collisions and determine if a Player can eat another one or a food.

3.3.4 Protocol

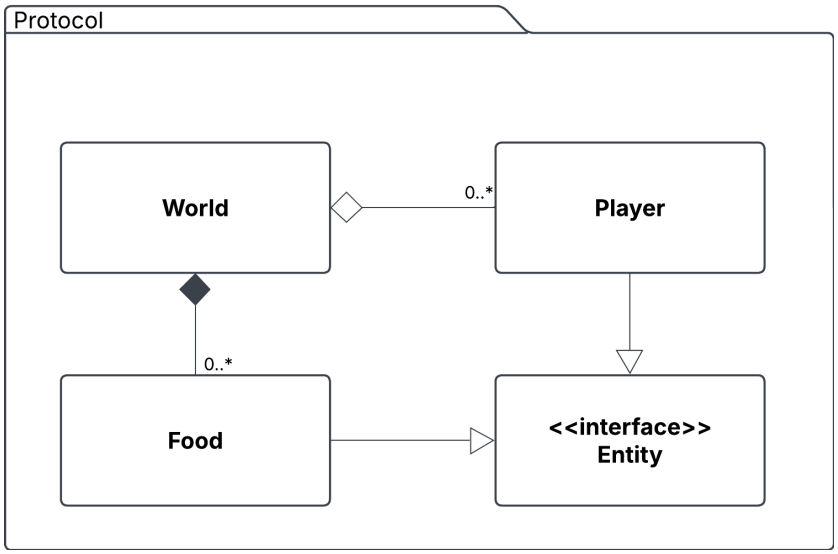

Figure 3.6: Protocol class diagram

As seen in the class diagrams above, there are a set of classes shared between the Mother Server and Child Server components. These classes represent the world state of a game session and are used as the body of messages exchanged to synchronize the game state between servers and clients, and also between the Mother Server and Child Servers. They include:

- **Entity (interface)**: A common abstraction implemented by `Player` and `Food`.
- **Player**: Represents a player in the game, including attributes like ID, nickname, position, and size.
- **Food**: Represents a food item in the game, including attributes like ID, position, and size.
- **World**: Represents the game world, containing multiple players and food objects.

3.3.5 Full Diagram

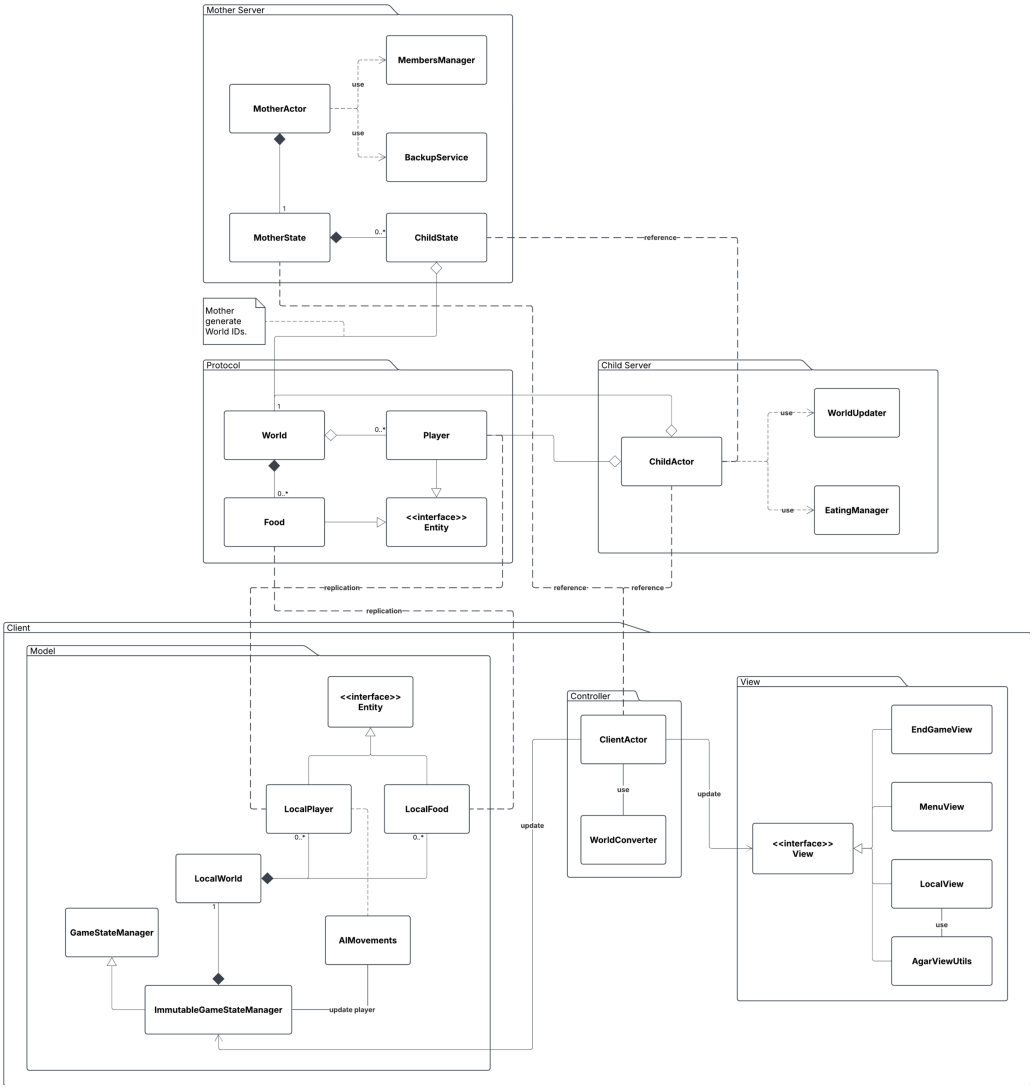

Figure 3.7: Full class diagram

- `Child State` contains reference to `ChildActor`
- `ChildState` contains the Child's managed `World` ID.
- `ChildActor` contains the `World`'s datas.
- `ClientActor` reference is available inside the `MotherState` and the `ChildActor`

3.4 Interaction

This section describes how the main components of the system – `MotherActor`, `ChildActor`, and `ClientActor` – communicate, when these interactions occur, and which interaction patterns they employ. The architecture is fully actor-based, relying on asynchronous message passing to coordinate distributed gameplay, room management, and fault recovery.

3.4.1 Client – Mother

When a client node starts, it registers itself in the cluster and waits for a game manager (`ChildActor`) to become available.

The `MembersManager` is a Mother Server agent that acts as the entry point for cluster membership events. Whenever a new client or child server joins or leaves the system, the `Receptionist` emits a `Listing` message that is wrapped by a message and forwarded to the `MotherActor`.

The `MembersManager` sends message to the `MotherActor` when a Client get in the cluster. Mother Server maintains a global view of all active child servers. Upon receiving this message, the `MotherActor` performs load balancing by selecting the least loaded `ChildActor`. The client is then assigned to that child and receives the reference through a message.

The interaction also covers room management. When a client requests to join a private room or create a new one, it sends the relative message to the currently assigned child, which forwards these requests to the mother. The `MotherActor` checks room availability or child capacity and responds via relative success or failure messages.

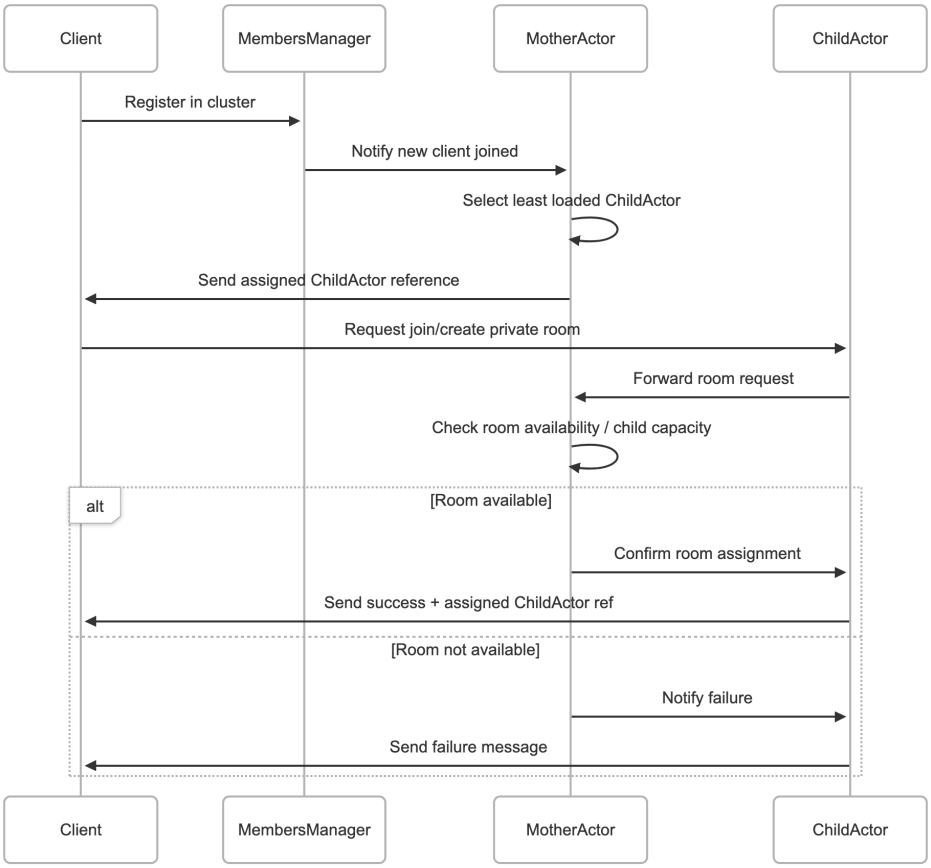

Figure 3.8: Sequence diagram illustrating the interaction between a client, the Mother server, and Child actors.

3.4.2 Mother – Child

The `MotherActor` manages global orchestration. When a new child server joins, the mother assigns it a unique world identifier through a set up message and starts tracking the clients connected to that child. From that moment on, the mother and child interact through several coordination messages:

The mother also supervises failure handling. When a child server disconnects, the mother requests the latest backup from the `BackupActor`. If another free child is available, the mother reassigns the recovered world and clients sending a 'new set up' message to the free child..

This supervision pattern ensures that failure of a child server does not result in data loss or abandoned players.

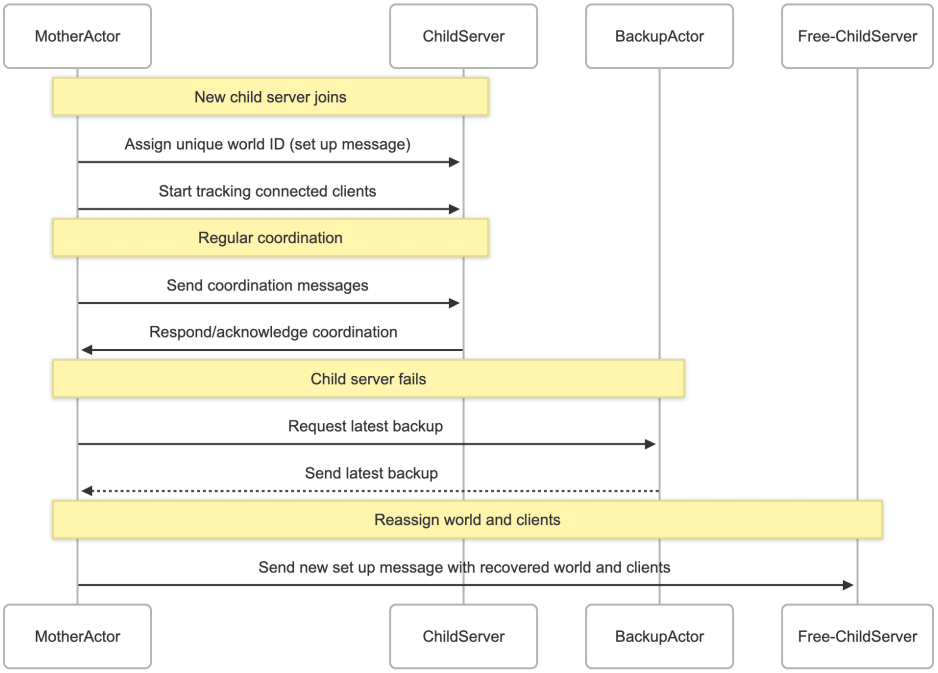

Figure 3.9: Sequence diagram showing MotherActor orchestrating child servers, handling client tracking, and supervising failure recovery through BackupActor.

3.4.3 Child – Client

The `ChildActor` handles gameplay logic and all direct communication with clients. When a client wants to join a world, it sends a request message to the child. The child creates a new player inside the managed world, updates the world state, and send it to the requestor client. Clients also initiate room-based interactions, such as joining a specific room or entering a private room, which the child forwards to the mother for approval.

The `ChildActor` manages the real-time game loop:

- Clients send move.
- The child aggregates all movements.
- At each timer tick, it updates the world using internal logic.
- Updated worlds are sent back.

If a player is eaten or leaves the room, the child updates the world, notifies involved clients, and may send and end game message to the affected player. The child also informs the backup system after each update to mantain games-snapshot stored.

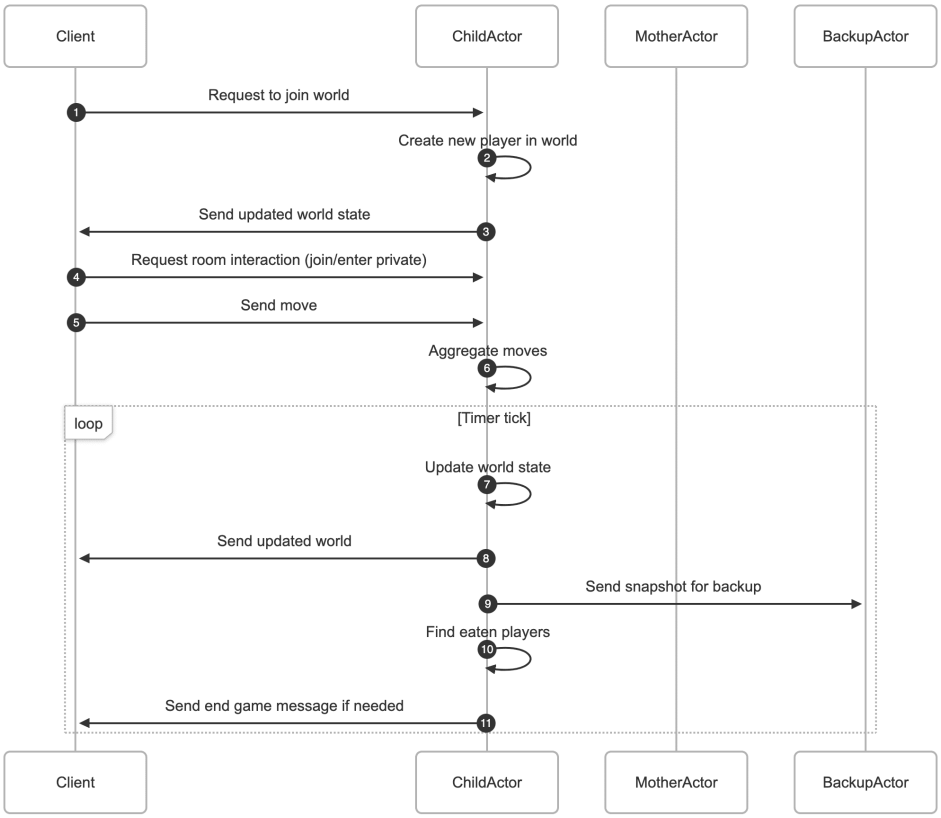

Figure 3.10: Interactions between clients, the ChildActor, MotherActor, and BackupActor during real-time gameplay.

3.4.4 Interaction Patterns

Across all actors and components, the system employs several recurring interaction patterns:

- **Asynchronous request–response**: Used in world requests, backup retrieval, and room creation.
- **Publish–subscribe via Receptionist**: Enables dynamic discovery of clients and child servers.

- **Periodic broadcasting**: Child actors push world updates at fixed intervals.
- **Supervision and failover**: The mother orchestrates recovery after child failures.
- **Load balancing**: The mother assigns new clients to the least loaded child server.
- **State replication**: The backup actor stores snapshots for recovery.

3.5 Behaviour

The system is composed of multiple distributed actors, each with a specific behavioural role and its own message-driven logic. Stateful actors such as the `MotherActor`, `ChildActor`, and `BackupActor` maintain essential information about connected clients, game worlds, and replicated states across the cluster. Stateless or lightweight components like `MembersManager`, react to events without storing long-term data, instead serving as intermediaries for discovery and data transformation. This separation allows the architecture to process high volumes of asynchronous events while keeping critical state centralized in dedicated actors.

State transitions in the system occur exclusively through message exchanges. The system behaviour emerges from the coordinated reaction of autonomous actors that evolve their state only when relevant messages or cluster events occur.

3.6 Data and Consistency Issues

This section analyzes potential data and consistency issues in the current project. It identifies risks associated with state replication, message handling, concurrency and failure scenarios.

- **Serialization Failures:** Nodes with different versions may fail to deserialise messages, causing crashes or silent message loss.
- **Inconsistent State:** The World produced from the Child Server depend on moves sent from the Client so it's not granted that a certain Client move is computed before another, it's non deterministic.
- **Duplicate Processing:** At-least-once message delivery may cause commands or events to be applied multiple times.

- **Volatile Data:** The game is structured around temporary sessions, and persistent data storage is not required. All game state information, including player positions, entity sizes, and in-game items, is maintained exclusively in memory throughout the duration of each session. Upon session termination, whether due to player elimination or voluntary exit, all associated state data is discarded.

3.7 Fault-Tolerance

3.7.1 Failure Detection via Akka Cluster / Receptionist

- The system relies on Akka Cluster's membership and discovery mechanisms to detect when nodes join, leave, or become unreachable.
- Components receive automatic notifications about changes in network availability and node status.
- These mechanisms are based on periodic heart-beating and reachability checks, allowing the system to promptly identify failures.
- A coordinating entity maintains the list of active nodes and updates it in real time to ensure consistent system-wide coordination.

3.7.2 Graceful degradation when capacity is unavailable

- The system maintains replicated snapshots of session state to provide recovery points in case of node failures.
- A dedicated component periodically collects and stores the information required to restore the state of ongoing sessions.
- In the event of a failure, the coordinator retrieves the most recent snapshot and initializes a new node with the recovered data.
- This enables session continuity with minimal loss of recent actions, depending on the snapshot frequency.

3.7.3 Cleanup on Disconnections

- When a client disconnects, the system updates the session by removing the participant and synchronizing the state with the remaining clients.
- This ensures that the simulation remains consistent and avoids references to entities that are no longer active.

3.7.4 Client-Side Back-Pressure

- The client uses a controlled communication pattern, sending new updates only after receiving a response from the system.
- This prevents overloading the processing nodes and mitigates the risk of network congestion.
- The mechanism also acts as an implicit retry strategy: each update is confirmed before the next one is sent.

3.7.5 Gaps and Risks

- The central coordinator still represents a potential single point of failure.
- No explicit application-level retry mechanisms are implemented to handle unstable network conditions.
- The recovery process relies on periodic snapshots: if a failure occurs just before a new snapshot, the most recent actions may be lost.

3.8 Availability

The system is designed to ensure high availability through a distributed server architecture capable of handling multiple game sessions concurrently. The *Mother Server* operates as a central coordinator, managing the assignment of players to the various *Child Servers* in order to avoid overloading any single server.

This load-balancing mechanism increases overall availability by dynamically redirecting new players to the least loaded servers. In the event of a game server failure, the Mother Server detects the loss of connectivity and can reassign players to another available server, thereby minimizing service disruption.

The distributed structure also prevents a single point of failure: even if one *Child Server* becomes unresponsive, other servers continue to operate normally, ensuring that ongoing matches remain accessible and that new sessions can still be created. This architectural choice significantly enhances the robustness and continuity of the system.

3.9 Security

The system's architecture incorporates security principles through isolation and controlled exposure of components. A key design choice is that the central coordinating component (the Mother Server) is never directly accessible by clients. All external communication passes exclusively through the Child Servers, which act as a *Protection Layer*. This ensures that clients do not know the identity, address, or internal structure of the Mother Server, effectively shielding it from direct attacks, probing, or unauthorized requests.

By exposing only the Child Servers at the network boundary, the architecture reduces the attack surface and compartmentalizes potential damage: even if a Child Server were compromised, the attacker would not gain direct access to the coordination logic or global state management. Internal communication within the cluster occurs through trusted mechanisms provided by Akka, preventing unauthorized nodes from joining or intercepting coordination messages.

Clients operate under a limited trust model: they only interact with their assigned Child Server, and all system-wide operations are validated internally within the cluster. This design protects the integrity of the authoritative session state, as clients cannot directly manipulate global management processes.

Some risks remain, such as the need to secure communication channels, validate client inputs robustly, and protect exposed Child Servers from abusive traffic. Nonetheless, the architecture provides a strong foundational security layer by isolating the core logic and ensuring that the most critical component remains completely hidden and protected.

Chapter 4

Implementation

4.1 Technological details

- Scala: main programming language used for both client and server components.
- Akka: utilized for building the actor-based concurrency model, facilitating communication between different components.
  - Akka Actors: used to implement the actor model for concurrent processing.
  - Akka Cluster: employed to create a distributed system where multiple nodes can work together.
- Java Swing: used for creating the graphical user interface (GUI) of the client application.

The adoption of Akka was driven by its robust actor-based concurrency model, which significantly reduces the complexity of developing distributed, fault-tolerant systems. By relying on asynchronous message passing, components operate independently, effectively decoupling the system logic. Furthermore, Akka's supervision hierarchy provides a structured approach to fault recovery, ensuring resilience against partial failures, while native support for location transparency and clustering facilitates horizontal scalability. These capabilities are essential for a multiplayer game architecture, meeting the rigorous demands of continuous world updates, real-time client–server coordination, and reliable session management.

4.2 Configuration

Default Akka communication use **Transmission Control Protocol** (TCP) and we chose to mantain it for several key reasons, with reliability and order being the most critical. Unlike UDP, TCP is a "connection-oriented" protocol that ensures data packets are delivered, and if a packet is lost, it's automatically resent. This is crucial for maintaining a consistent game state, especially for core mechanics like player movements, collisions, and cell consumption, where a lost packet could lead to players appearing to be in different locations or a desynchronized game world. Furthermore, TCP guarantees that packets arrive in the correct order, which is essential for deterministic game logic. UDP is favored for fast-paced shooters where low latency is paramount, the nature of an Agar.io clone tolerates slightly higher latency in exchange for the absolute reliability and synchronization that TCP provides.

Below is a snippet from the *application.conf* file showing the configuration for using TCP as the transport protocol with Akka's remote artery:

Listing 4.1: Network protocol configuration

```
remote {
    artery {
      transport = tcp
    }
  }
```

To represent in-transit data, we opted for **JSON (JavaScript Object Notation)** produced by Akka's Jackson serializer. The converse from entity to JSON and vice versa is handled automatically by Akka, which simplifies the serialization process.

Listing 4.2: Transit data configuration

```
serializers {
    jackson-json = "akka.serialization.jackson.
       JacksonJsonSerializer"
}
serialization-bindings {
    "akka.actor.typed.ActorRef" = jackson-json
    "akka.actor.typed.internal.adapter.ActorRefAdapter"
       = jackson-json
    "it.unibo.protocol.Message" = jackson-json
}
```

Chapter 5

Validation

The system is designed as a CP (Consistency–Partition Tolerance) distributed architecture. Running on Akka Cluster and relying on asynchronous message passing, it is built to tolerate network partitions ('P') while strictly enforcing strong data consistency ('C').

The architecture is server-authoritative, establishing the Child Server as the single source of truth for the world state. This actor merges all position updates received from clients and broadcasts the resulting unified world snapshot. Crucially, clients wait to receive this authoritative state before submitting subsequent moves. This synchronization ensures that all participants always perceive an identical, coherent view of the world, preventing state divergence.

In the event of a node failure or partition—managed by components such as the MembersManager, MotherActor, and BackupActor—the system prioritizes correctness over absolute availability. When a Child Server becomes unreachable, the MotherActor detects the failure and restores the latest state from the BackupActor. During this brief recovery window, the specific game world becomes temporarily unavailable to ensure no conflicting states are generated. Once the state is restored, operations resume with full consistency, effectively handling the trade-off between uptime and data integrity inherent in distributed systems.

5.1 Acceptance tests

We did not implement automatic tests because the system relies heavily on distributed actors, UI interactions, and asynchronous event sequences that are difficult to reproduce deterministically. Many behaviors depend on timing, network conditions, and user-driven actions, making them hard to

simulate reliably with standard automated testing tools. As a result, manual and scenario-based testing was more suitable for verifying the system's correctness.

Our tests were computed in a "MacBook Pro 1,4 GHz Quad-Core Intel Core i5 - 8 GB 2133 MHz LPDDR3 - Intel Iris Plus Graphics 645 1536 MB" and all acceptance criteria 2.2 were empirically verified.

1. Support for at least 5 concurrent players per session.
2. End-to-end latency below 200 ms.
3. Maintain a minimum frame rate of 30 FPS under typical load (5 players).
4. Larger cells can consume smaller ones, and growth is visually observable.
5. Cells speed is visible decreased on size growing.

Chapter 6

Release

Each release is packaged into three separate JAR files: one for the Mother Server, one for the Child Server, and one for the Client. This modular structure enables independent deployment and scaling of each component according to demand.

All JAR files follow semantic versioning (e.g., 1.0.0, 1.1.0, 2.0.0), making it easier to track changes and maintain compatibility across components.

The JARs are distributed through a public GitHub repository, where users can download the latest versions. Each release is also tagged in the repository for convenient access.

Chapter 7

Deployment

The following instructions guide you through deploying the Raga.io system from scratch on your local machine - for remote deployment, additional configuration may be required.

To deploy, follow these steps:

- **Prerequisites:**
  - Ensure that you have a compatible Java Runtime Environment (JRE) installed on your system.
- **Download the JAR files:**
  - Mother Server JAR
  - Child Server JAR
  - Client JAR
- **Start the Mother Server:**
  - Open a terminal and execute the following command:

```
java -jar mother-server-assembly-X.X.X-SNAPSHOT.jar
```

  - The Child Servers must be started next, using a similar command:

```
java -jar child-server-assembly-X.X.X-SNAPSHOT.jar
```

  - Finally, the Client application can be launched:

```
java -jar client-assembly-X.X.X-SNAPSHOT.jar
```

Chapter 8

User Guide

8.1 Menu View

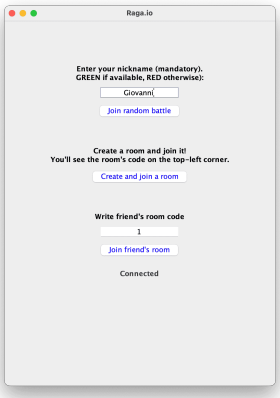

Figure 8.1: User is correctly connected with the Server (ready to play).

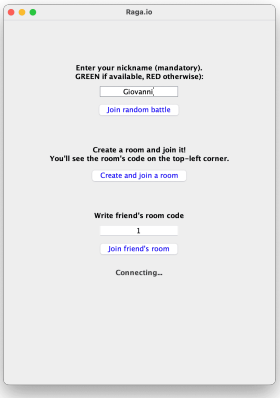

Figure 8.2: User is connect with Main Server but there isn't a Child Server available.

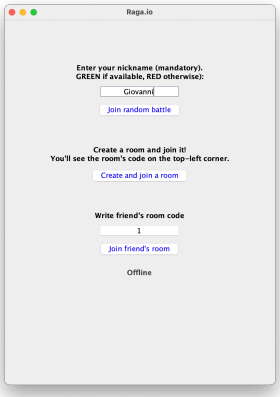

Figure 8.3: User is offline, Mother Server is not reachable for some reasons.

User actions on the menu page:

- Enter a nickname in the "Nickname" text field.
- Join a random session by clicking the "Join Random Battle" button.
- Create a new session by clicking the "Create and Join a room" button.

- Write a session ID in the "Session ID" text field.
- Join a specific session by clicking the "Join friend's room" button (the session ID must be valid).
- At the bottom of the page, the user can see network status information.

8.2 Game View

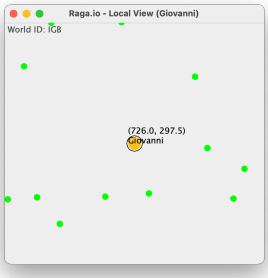

Figure 8.4: Game play (1/3).

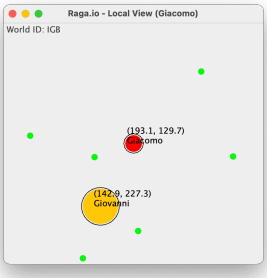

Figure 8.5: Game play (2/3).

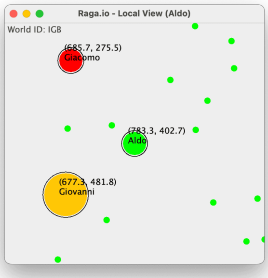

Figure 8.6: Game play (3/3).

In the game page, the user can:

- View the Room Code on the top left corner, which can be shared with friends to join the same session.
- Move the cell using the mouse cursor.
- See their nickname displayed above their cell.
- View other players' nicknames above their respective cells.
- See food items as small circles scattered around the map.

Chapter 9

Conclusions

9.1 Limitations

- Single Point of Coordination

  The Mother Server acts as the global orchestrator and remains a potential single point of failure.

- Snapshot-Based Recovery

  Fault recovery relies on periodic snapshots rather than continuous replication. If a Child Server fails immediately before a new snapshot is produced, the system may lose recent game actions, causing partial de-synchronization or state rollback.

- Limited Load Adaptation

  Load balancing is based on simple metrics (e.g., number of connected players). This approach does not consider more complex factors such as computational load, network latency, or heterogeneous server capacities.

- Consistency Under High Latency

  The authoritative tick-based model ensures coherence, but under unstable network conditions some client inputs may be dropped. This is acceptable for casual gameplay, but limits precision and responsiveness at scale or under adverse network conditions.

- Security Exposure at Child Servers

  Child Servers represent the only externally exposed surface. Although the Mother Server is isolated, Child nodes remain vulnerable to denial-

of-service attempts, malformed client inputs, or abusive traffic without additional hardening.

9.2 Future improvements

- **Transition to UDP for Gameplay Updates** – Currently, the system uses TCP to ensure message ordering and reliability. However, for real-time movement TCP's retransmission mechanism can introduce latency. A hybrid approach using UDP for high-frequency player movement and TCP for critical game state changes (like spawning or game over) would significantly reduce latency.
- **Client-Side Prediction and Reconciliation** – Currently, the system is server-authoritative, and clients wait for the server's state update before confirming moves, which can lead to ignored inputs or visual stutter. Implementing client-side prediction would allow the client to render movement immediately while reconciling the state when the authoritative server response arrives, creating a smoother experience for players with higher latency.
- **Persistent Data Storage** – Session data is currently volatile and discarded when a game ends. Integrating a persistent database would allow for long-term player statistics, global leader boards, and user customization (skins) that persist across different game sessions.

Chapter 10

Self-evaluation

During the development of this project, we had the opportunity to expand our knowledge in the field of Distributed Systems, particularly in the real-time coordination and distributed management required by the Agar.io game. We also deepened our understanding of how to maintain data consistency, handle concurrent interactions, and mitigate the effects of network latency without compromising system performance.

Elvis took the lead in laying the foundations of the project by defining the initial system structure, establishing the distributed communication model, and setting up the core architectural principles on which the rest of the system was built. Together we worked on how nodes discover each other, how messages circulate across components, and how the system can scale by distributing the workload efficiently under varying conditions.

Eleonora complemented this effort by focusing on distributed reliability and system correctness. She implemented the failure-handling strategies and real-time detection mechanisms needed to maintain system stability, and contributed to the logic governing distributed gameplay behaviour to ensure coherent state evolution under concurrent conditions.

Together, these combined efforts allowed us to tackle the full spectrum of challenges typical of distributed systems. The final result aligns with our expectations, and we consider the project outcome highly satisfactory.