Distributed Multiplayer Pacman

Final Report for the Distributed Systems Course 2025/2026

Simone Carpi
simone.carpi@studio.unibo.it

Alessandro Ronchi
alessandro.ronchi7@studio.unibo.it

Jiekai Sun
jiekai.sun@studio.unibo.it

September 10, 2026

This project presents the design and implementation of a multiplayer adaptation of the classic arcade game Pac-Man. At its core, the system adopts a client-server architecture to manage real-time game state synchronization and player inputs. Around this foundation, additional services such as authentication, matchmaking and match results persistence were integrated. Beyond these aspects, the project addresses common distributed systems issues. Specifically, it implements horizontal game server scaling to handle fluctuating player demand, periodic game state checkpointing for match recovery and robust disconnection and reconnection handling.

1 Concept

1.1 Product Type

The product developed in this project is a distributed real-time multiplayer game system based on the classic Pac-Man arcade game. Overall, the system is envisioned to support multiple concurrent users, each operating a desktop GUI client application that interacts with a distributed backend of RESTful web services and dynamically provisioned game servers:

- **Desktop GUI Client:** A Java-based application installed on player devices that serves as the entry point to the overall system. It handles registration and login, matchmaking requests, real-time gameplay rendering and statistics display.
- **Backend Infrastructure:**
  - Authentication Service: A REST service responsible for user registration, authentication and issuance of cryptographic authentication tokens.
  - Matchmaking Service: A REST service responsible for aggregating players waiting to play into four-player matches, requesting game servers to host such matches and routing clients to their assigned server.
  - Query Service: A REST service providing endpoints to query individual player statistics.
  - Game Server Manager: A REST service that listens for requests from the Matchmaking Service to dynamically spin up, monitor, and tear down game server instances.
  - Game Servers: Dedicated servers spawned on demand to host live matches, executing the game logic authoritatively and synchronizing the connected clients.

1.2 Use Case Description

1.2.1 What the Software Does

The project adapts classic arcade Pac-Man rules into a four-player format:

- A match takes place on a shared 2D map with four players, each controlling a distinct Pacman character.
- Players compete to achieve the highest score by eating dots while avoiding bot controlled Ghosts that roam the map. Consuming a special dot grants players the ability to temporarily disable ghosts on impact.
- Each player starts with three lives. Colliding with a Ghost deducts one life and triggers a brief period of temporary invincibility during which no life can be lost. Depleting all three lives results in permanent elimination.

- A match ends when the three-minute timer expires or only one player with remaining lives is left standing. The winner is the player with the highest number of remaining lives and the highest score.

To take part in a match, players launch the client application, login, select a map and queue for a match. Once four players willing to play the same map are successfully matched, the game session starts and players control their Pacman character in real time, competing for the highest score. Outside of matches, players can navigate to a dedicated section to review their personal statistics.

Behind the scenes, the platform manages user authentication, creates matches out of queueing players and dynamically provisions game servers. During a live match, the game server owns the authoritative game state, synchronizing it across all connected clients to ensure real-time consistency while providing fault tolerance through disconnection and reconnection protocols as well as game state recovery measures.

1.2.2 Where the Users Are

In the target deployment model, users are assumed to be geographically distributed across the Internet, connecting to backend services from remote desktop devices.

1.2.3 When and How Frequently Users Interact

User interaction with the system operates across two distinct frequency models:

- Outside of live matches, user interactions are discrete and request-driven. Users perform occasional actions such as registering accounts, logging in, reviewing personal statistics and interacting with the matchmaking system.
- Once a match starts, interaction shifts to a continuous high-frequency model. During gameplay, users transmit commands to move their Pacman character, while the game server continuously updates and broadcasts the authoritative game state to all connected clients.

1.2.4 How Users Interact and Devices Used

Users interact with the platform by using the client application on PCs running Windows, Linux or macOS. All the actions on the client are performed using standard mouse and keyboard controls.

1.2.5 User Data Storage

To support core platform features, user credentials (usernames and passwords) are persisted for authentication, while long-term player statistics (matches played, wins, and best scores) are stored to enable historical performance tracking.

1.3 Why Distribution is Needed

- **The system is intrinsically distributed:** Distribution is an intrinsic aspect of an online multiplayer game. In fact, multiple players play the same match on different devices, and the centralized backend infrastructure needs to collect user input and send updates about the current authoritative game state.
- **Performance Consistency:** Real-time game logic requires consistent, high-frequency execution with as little performance overhead as possible. Separating CPU-bound game execution from I/O-bound administrative operations, such as authentication and database operations, prevents background workloads from competing for shared system resources.
- **Horizontal Scalability:** Isolating non-gameplay related services from game execution allows individual system components to scale independently. As concurrent player demand fluctuates, the Game Server Manager can dynamically provision dedicated Game Server nodes.
- **Fault Tolerance:** In a monolithic architecture, a single process crash terminates all active matches simultaneously. Distributing live matches across independent Game Server nodes isolates potential failures. Coupled with periodic game state checkpointing, this enables match recovery on replacement Game Servers without disrupting other ongoing matches or overall platform operations.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

1. **User Account Management**
   - *Description*: The system shall allow new users to register an account with a unique username and password and allow existing users to login.
   - *Acceptance Criterion*: A user can successfully register and login. Invalid credentials must result in an appropriate error message.
2. **Player Statistics Tracking**
   - *Description*: The system shall track, update and display individual statistics for registered users, including total matches played, best scores and win count.
   - *Acceptance Criterion*: Upon completion of a match, a player's metrics are updated and visible in the statistics section.
3. **Matchmaking**
   - *Description*: The system shall allow users to select a game map and enter a queue to be matched with other players.
   - *Acceptance Criterion*: Selecting a map transitions the user into a queuing state alongside three other players who selected the same map. Once a match is found, the players engage in gameplay on the selected map.
4. **Real-Time Game Execution**
   - *Description*: The system shall manage and execute 3-minute multiplayer matches where 4 players simultaneously control Pacman characters in a 2D map, eating dots to score points and avoiding bot controlled Ghosts.
   - *Acceptance Criterion*: Player inputs update their character positions on screen in real time, scores change dynamically as dots are eaten and the match terminates when the 3-minute timer expires or win conditions are met.
5. **Player Disconnection**
   - *Description*: If a player loses connection during a match, the system shall replace the disconnected player with a bot to ensure an uninterrupted game experience for remaining participants.
   - *Acceptance Criterion*: When a player disconnects mid-match, their character continues moving under bot control without halting the match for other players.
6. **Player Reconnection**
   - *Description*: The system shall allow a player to rejoin an ongoing match and resume character control following either a client-side disconnection (e.g. loss of connectivity, client crash) or a server-side failure.

- *Acceptance Criterion*: Re-launching or reconnecting the client while the match is still active allows the player to rejoin it, whether on the original server or a new recovery server.

2.2 Non-Functional Requirements

1. **Cross-Platform Compatibility**
   - *Description*: The client application shall execute on desktop operating systems including Windows, Linux, and macOS.
   - *Acceptance Criterion*: The client application runs successfully on the supported operating systems, allowing players across different operating systems to play together.
2. **Game State Consistency**
   - *Description*: The system shall maintain consistent game state across all connected clients by enforcing authoritative Game Server execution of the game logic.
   - *Acceptance Criterion*: Under optimal network conditions, every client displays an identical game state (e.g. entity positions and scores). In the event of a game state divergence, the client eventually reconciles its local game state upon receiving an authoritative game state snapshot.
3. **Speculative Game Execution**
   - *Description*: To maintain a responsive UI, the client shall speculatively execute local player movement inputs prior to server confirmation, while continuously reconciling local state against the authoritative game state.
   - *Acceptance Criterion*: Character movement renders immediately upon player input, even during temporary disruptions to the authoritative game state stream.
4. **Game Server Scalability**
   - *Description*: The system shall dynamically instantiate Game Server instances to play on as the matchmaking system creates matches.
   - *Acceptance Criterion*: Upon match creation, the Game Server Manager automatically provisions a Game Server to play on.
5. **Match Recovery**
   - *Description*: In the event of a Game Server crash, the system shall resume the match on a newly spawned Game Server instance with the latest checkpointed game state.
   - *Acceptance Criterion*: Simulating a Game Server crash in the middle of a match causes the platform to automatically restore the match on a recovery server without losing significant progress.

2.3 Relevant Distributed System Features

- **Transparency**: The system masks underlying distributed system complexities from end-users to deliver a smooth game experience:
  - Players do not need to know the whereabouts of the Game Servers. The Matchmaking Service and the Client negotiate Game Server connection details entirely behind the scenes, routing players directly to matches and reconnecting them after a failure.
  - Minor network-related issues during gameplay, such as latency spikes or packet drops, are mitigated through client-side speculative execution. Conversely, significant failures are promptly made visible to the user with explicit error notifications and actionable resolution steps (e.g., offering an option to reconnect) rather than allowing the application to hang in a failed state.
- **Fault Tolerance and Dependability**: Since the platform manages active real-time multiplayer sessions, system reliability and data integrity are fundamental constraints:
  - If any Game Server process fails, the system automatically detects the fault and triggers recovery procedures. This ensures uninterrupted platform operations and a smooth user experience.
  - Data loss and game state corruption is not considered acceptable. To guarantee integrity, the live game state of each match is periodically checkpointed and backend database storage is replicated across different nodes.
  - While no strict recovery deadline is enforced, the Game Server recovery process must complete in a timely manner to maintain user engagement.
- **Scalability**: Given the potential for rapid growth and sharp peaks in concurrent players, the system is designed to handle increasing user volume, request traffic, and active match instances:
  - Game Server count scales horizontally based on active player demand. They are dynamically spawned for new matches and terminated upon completion.
  - Core entry points such as Authentication and Matchmaking services are designed to be stateless. While the baseline target deployment runs limited replicated instances for basic availability, the stateless architecture allows them to be scaled horizontally and absorb sudden spikes during peak play times.
- **Security and Trust**: The system enforces role-based access control by locking all core features such as matchmaking behind authentication. Users are required to login and present signed tokens that backend services validate before executing their requests. As for backend services, we assume there to be mutual trust between the different backend components. So, no authentication or authorization is needed during internal interactions.

- **Resource Sharing, Coordination, and Synchronization**: As a real-time multiplayer system, the platform requires continuous coordination among clients and backend services to manage shared game state and infrastructure:
  - The game state represents a shared resource which all players interact simultaneously with. The authoritative Game Server performs game state updates and broadcasts periodic snapshots to ensure state synchronization and consistency across all clients.
  - The Matchmaking Service serves as a shared point of coordination where incoming player requests are handled to create four-player matches.
- **Openness, Interoperability, and Heterogeneity**: The system integrates open communication standards and dynamic server management to handle game sessions:
  - The system prioritizes client-side heterogeneity, utilizing platform-agnostic communication protocols and standardized data serialization formats so that players on Windows, Linux and macOS can play with each other.
  - The platform relies on a container orchestration layer to handle dynamic Game Server allocation and monitoring.
- **Performance, Concurrency and Efficiency**: As a real-time multiplayer platform, the system requires low-latency game logic execution and optimized network utilization to deliver a responsive user experience:
  - The game execution logic algorithm must be computationally lightweight because the client executes the exact same simulation as the Game Server in order to implement speculative execution.
  - The overall system architecture is naturally concurrent, as independent services, including Authentication, Matchmaking and Game Server instances, must operate in parallel to serve multiple users simultaneously.
  - The primary source of network bandwidth overhead is the real-time communication between clients and Game Servers during active matches. Because the authoritative server frequently broadcasts game state snapshots to maintain synchronization, payload sizes must be optimized to minimize bandwidth consumption.
- **Economy and Costs**: Although not a primary concern, basic resource management is applied to Game Servers by enforcing an on-demand lifecycle. Unlike continuously running administrative services, dedicated Game Server instances are instantiated only upon match formation and torn down upon termination, eliminating idle processes and optimizing resource consumption.

3 Design

3.1 Architecture

The platform combines a system-wide Service-Oriented Architecture with component-specific patterns to best match each service's purpose:

- **Service-Oriented Architecture (SOA)**: The platform is split into independent services (`Front-End Service`, `Auth Service`, `Matchmaker`, `Query Manager`, `GameServer Manager`, and `GameServer`). To avoid the complexity of distributed communication, some backend services share database instances while maintaining clear logical boundaries.
- **Controller-Service-Repository (CSR)**: The administrative services use a Controller-Service-Repository pattern to process requests. The Controller layer handles external API requests, the Service layer executes core business logic and the Repository layer manages database operations.
- **Model-View-Controller (MVC)**: Both the `Game Client` and `Game Server` share a common core codebase structured around the Model-View-Controller pattern:
  - *Model*: Encapsulates pure game domain logic, rules and entities, separated from networking and infrastructure concerns.
  - *View*: Renders UI menus and gameplay graphics on the client, while operating in a headless mode on the server.
  - *Controller*: Manages system interactions and network communication for both applications:
    - *Client Controller*: Handles user input, game state synchronization with the Game Server and requests to backend services.
    - *Server Controller*: Handles server-side networking, authoritative game state updates, client synchronization and interaction with backend services.

3.2 Infrastructure

To provide a complete view of the system architecture, the infrastructure is described at two levels of abstraction: a conceptual component view (fig. 1) focusing on domain interactions and a detailed Kubernetes infrastructure view (fig. 2) detailing service abstractions and orchestration.

- **Infrastructural Components**:
  - *Game Client*: The application running on the user device, interfacing with administrative services and `Game Servers`.
  - *Administrative Services*: Independent services (`Auth Service`, `Query Manager`, and `Matchmaker`), each exposing their own API interface to clients.

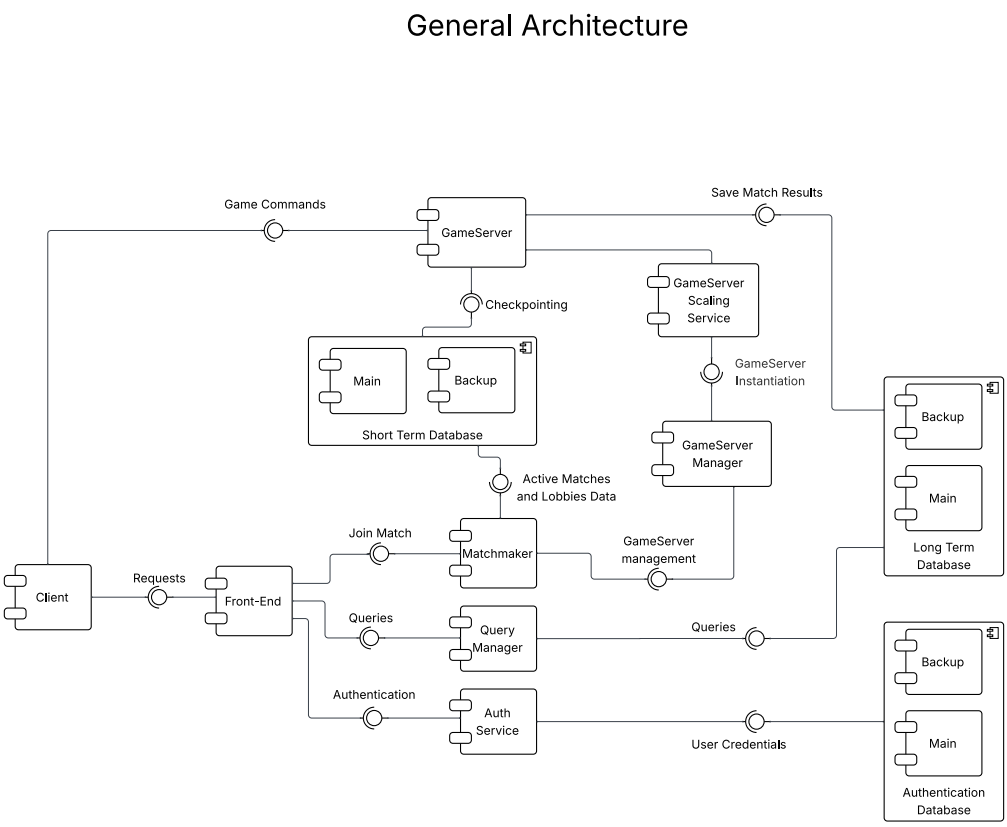

Figure 1: Conceptual System Component Diagram

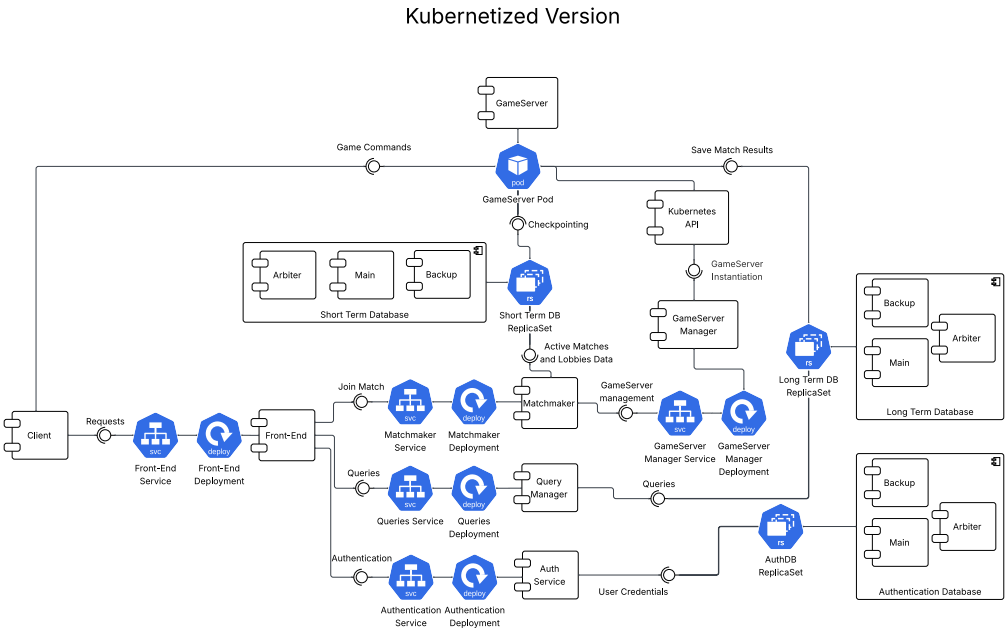

Figure 2: Kubernetes Infrastructure Component Diagram

- *Game Server Manager*: A service responsible for managing the lifecycle of `Game Server` instances.
- *Front-End*: The client's entrypoint for interactions with administrative services. Acts as a Gateway responsible for routing each client request to the correct administrative service.
- *Game Servers*: Dynamic headless server instances that host game sessions and process client commands during gameplay.
- *Replicated Databases*: Three database clusters configured with primary and backup instances for redundancy and fault tolerance:
  - *Authentication Database*: Stores credentials and other authentication related data.
  - *Short Term Database*: Stores matchmaking data, match checkpoints and Game Server metadata.
  - *Long Term Database*: Stores player statistics.

- **Network Topology and Distribution**:
  - *Game Clients*: Run locally on user devices anywhere in the world, sending requests over the public internet to communicate with the backend.
  - *Backend Services*: Run within an internal private network. While specific REST API endpoints are exposed to clients for administrative actions (like

authentication and matchmaking), internal service-to-service communication and database access remain isolated from direct public access.

  - *Game Servers*: Run within the backend server environment, but they do expose TCP and UDP ports to the outside so that clients can establish direct connections during active game sessions.

- **Component Discovery and Routing**:
  - *Administrative Services*: Clients locate and communicate with administrative backend services via standard DNS hostnames and REST API endpoints.
  - *Internal Services*: Backend services discover each other using internal network hostnames, which are resolved using an internal DNS.
  - *Game Servers*: Dedicated game servers are instantiated dynamically per match. The Game Server Manager assigns host IP addresses and TCP/UDP ports, which the Matchmaker then communicates to clients.

3.3 Modelling

3.3.1 Domain Entities

Game Domain Entities A detailed UML interface diagram illustrating the game domain entities and their relationships is provided in Figure 3. The game domain is designed around the `Game` interface, which acts as the top-level entry point upon which operations are performed on every simulation update. The `Game` advances the simulation by updating the `GameContext`, a container holding all active game entities and computed collisions required for logic evaluation. Within the context is the `GameState`, which maintains match lifecycle information such as the current score leaderboard (mapping player identifiers to their scores), remaining match duration (starting at 3 minutes), whether the match is over, and the name of the winner if the match has concluded.

The active domain entities operate within a 2D world defined by a `GameMap`, which structures the environment into a grid of `Tile` elements. Different `GameMap` instances feature distinct `Tile` arrangements, yielding unique maze layouts. Each tile can be either `EMPTY`, contain a dot (`DOT` or `SPECIAL_DOT`), act as an obstacle (`WALL`) through which entities cannot pass, or serve as a spawn point (`PACMAN_SPAWN` or `GHOST_SPAWN`).

Every active entity interacting within the `GameMap` extends `GameEntity`, inheriting 2D spatial coordinates (`Vector2D`), custom game logic (`update`) and a `BoundingBox` for spatial collision checking.

Below are the entities that interact with each other in the `GameMap`:

- **Dot**: Immovable `GameEntity` instances tied to `Tile`s. When a `Pacman` collides with a normal `Dot`, the `Dot` is consumed and awards a point to that player. Consuming a special `Dot` grants `Pacman` the ability to eat `Ghost`s, temporarily disabling any of them on impact. Once consumed, `Dot`s automatically respawn on their respective `Tile`s after a brief delay.
- **MovableEntity**: Entities capable of navigating the `GameMap` along valid directions (`UP`, `DOWN`, `LEFT`, `RIGHT`, or `NONE`):
  - **Pacman**: Movable entities driven by player inputs able to traverse the `GameMap` and score points by eating `Dot`s. Each `Pacman` starts with a maximum of 3 lives. Upon colliding with a `Ghost` under normal conditions, it loses a life and is temporarily granted invincibility to prevent quick consecutive life losses. Once all lives are depleted, the entity is permanently eliminated and will not respawn for the remainder of the match. Each match features exactly 4 `Pacman` instances.
  - **Ghost**: Movable entities controlled by bots, navigating the map to hunt player controlled avatars. While a `Ghost` can be temporarily eliminated when eaten by a `Pacman` who has consumed a special `Dot`, it is never permanently removed from the match and will always respawn on its designated spawn `Tile` after a short while.

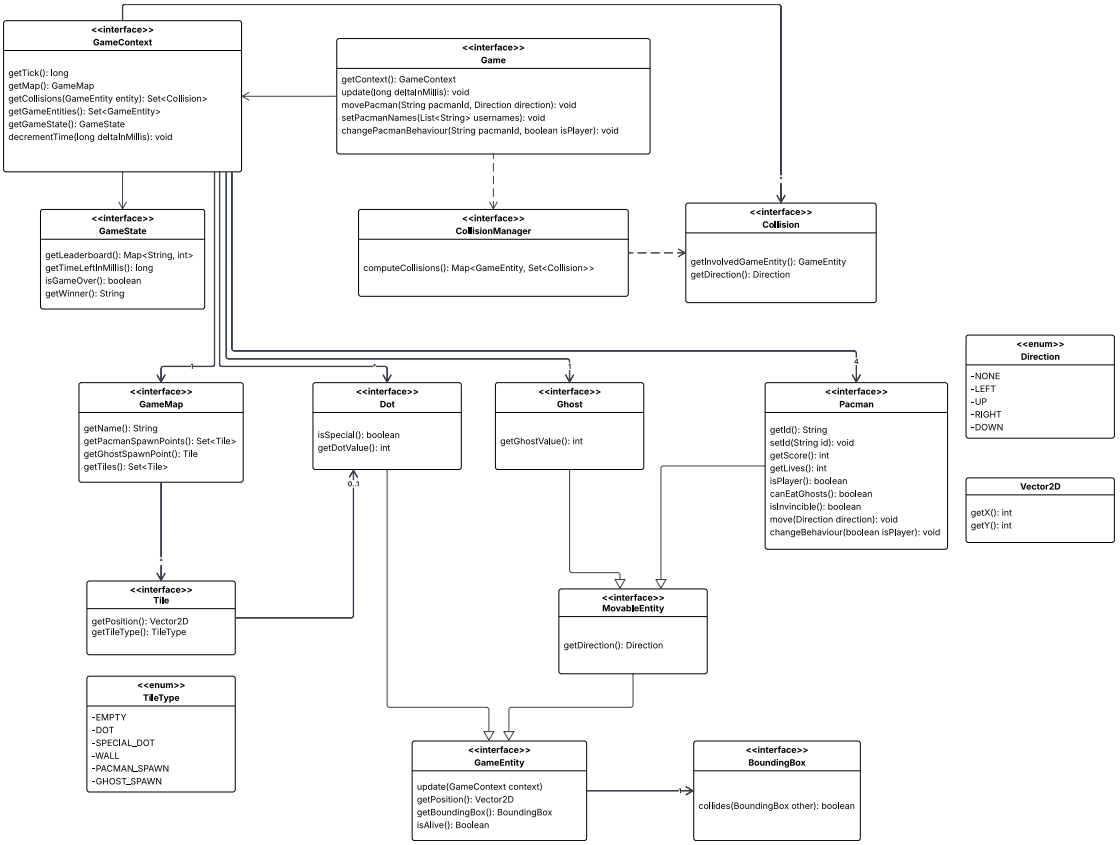

Figure 3: Game Domain Model Interfaces.

High-Level Domain Entities While the game domain handles the match execution, the platform layer relies on higher-level entities to manage users, matchmaking, server provisioning and statistics persistence:

- *User*: Represented by a record identified by a unique `username`. It is the primary actor who interacts with the matchmaker, plays matches and fetches statistics.
- *User Statistics*: Tracks aggregate performance metrics associated with a user (total matches played, total matches won and highest score). Such statistics are updated upon match completion.
- *Lobby*: Represents a temporary room utilized by the matchmaker to gather players intending to play on the same map. Identified by a unique `lobbyID`, it tracks assigned users and enforces a strict 4-player capacity limit, triggering match creation once full.
- *Match*: Serves as a shared record to keep the Matchmaker, Game Server Manager and the dedicated Game Server synchronized throughout the match. Identified by a unique `matchID`, it holds references to the expected players, the Game Server network coordinates, current match status and the periodic game state checkpoints saved throughout the match.

3.3.2 Infrastructural Mapping

- **Game State**: The authoritative `GameContext` resides in-memory on the Game Server hosting the match, where real-time physics and game logic are evaluated. Clients do not perform authoritative updates; instead, their rendering logic relies on game state snapshots received from the server. In turn, clients capture player input commands (movement directions for their respective `Pacman`) and transmit them to the Game Server for execution. Although game state and commands exchanged between client and server are ephemeral, the game server periodically persists game state checkpoints for backup purposes, retaining them until the match is completed.
- **Lobby and Match**: Matches are driven by a short-lived state shared across services. The matchmaker manages active `Lobby` instances within a database, tracking the usernames of players queueing for a specific map. Once the 4-player threshold is met, a match is found and the corresponding lobby is removed from the database. A `Match` is then instantiated as a shared coordination record across the Matchmaker, Game Server Manager and the dedicated Game Server. This entity is used to coordinate server allocation and player connections, remaining active until the match officially terminates, after which it is removed from the database.
- **User and User Statistics**: Long-term platform state, including user accounts and user statistics, is stored permanently in the database layer. The `User` entity is

managed by the Authentication Service to encapsulate core authentication credentials, such as the username and password hash, while the client application maintains generated authorization tokens exclusively in memory. `Player Statistics` are stored permanently to track aggregate metrics, such as total matches played, total wins, and highest achieved scores. These metrics are directly updated by Game Servers upon match completion and subsequently retrieved by clients through the Query Service.

3.3.3 Domain Events

- **Authentication**:
  - *UserRegistered*: A new user account is successfully created.
  - *UserAuthenticated*: User credentials are verified and an authorization token is issued.
- **Matchmaking**:
  - *PlayerQueued*: A user requests to join matchmaking for a specific map.
  - *LobbyUpdated*: A user joins or leaves a lobby.
  - *MatchFound*: A lobby reaches its 4-player capacity threshold and a match is created.
- **Match Lifecycle and Persistence**:
  - *GameSessionBound*: A client establishes a direct connection with the assigned Game Server.
  - *MatchStarted*: The Game Server starts the game and listens for player inputs.
  - *CheckpointPersisted*: The current game state has been backed up for recovery purposes.
  - *MatchEnded*: The match has terminated either because the 3-minute timer expired or win conditions are met.
  - *UserStatisticsUpdated*: User statistics have been updated upon match completion.

3.3.4 Exchanged Messages

Game Client ↔ Administrative Services

- *User Registration Request*: Sent to the Authentication Service to create a new user account.
- *User Authentication Request*: Sent to the Authentication Service to verify credentials and acquire an authorization token.
- *User Statistics Query*: Sent to the Query Service to fetch user statistics.

- *Join Matchmaking Request*: Sent to the Matchmaker to join matchmaking for a specified map.
- *Quit Matchmaking Request*: Sent to the Matchmaker to withdraw from the matchmaking process.
- *Matchmaking Status Request*: Periodically sent to the Matchmaker to determine whether a match has been found.
- *Server Parameters Query*: Sent to the Matchmaker to get the connection details of the Game Server where the match is supposed to be played.
- *Match Quit Request*: Sent to the Matchmaker forfeit a live match.

Game Client ↔ Game Server

- *Join Server*: Sent by the client upon connecting to the server.
- *Join Server Acknowledgment*: Sent by the server to confirm connection acceptance.
- *Game Start*: Broadcast by the server to all connected clients to signal the start of the match.
- *Move Pacman*: Sent by the client to request movement of their Pacman character to the server.
- *Game Context*: Broadcast by the server to deliver authoritative game state snapshots for client synchronization.
- *Game End*: Broadcast by the server to signal match completion.
- *Heartbeat*: Exchanged bidirectionally to monitor connection liveness and detect unexpected disconnections.
- *Explicit Disconnect*: Sent by the client to signal intentional disconnection.

Game Server ↔ Administrative Services

- *Expected Players Query*: Sent by the server to retrieve the expected group of players that will play the match it is about to host.
- *Game State Snapshot Write*: Sent by the server to persist a game state snapshot for the match it is currently hosting.
- *Game State Snapshot Query*: Sent by the server to fetch the latest snapshot of a given match for game state recovery.
- *Player Statistics Write*: Sent by the server upon match termination to update user statistics.

GameServer ↔ GameServer Manager

- *Match Ended*: Sent by the GameServer to signal that the match it was hosting has ended.

Matchmaker ↔ GameServer Manager

- *Create GameServer*: Sent by the Matchmaker to request the instantiation of a GameServer and obtain its connection details.
- *Check GameServer*: Sent by the Matchmaker to check the current status of a specific GameServer, and instantiate a recovery GameServer if needed.
- *Match Ended*: Sent by the GameServer Manager in response to the `Match Ended` signal received from a GameServer. Signals that a match hosted by a previously active GameServer has ended, and informations about the match can be deleted.

3.4 Interaction

How Components Interact System interactions follow two distinct communication paradigms: a synchronous Request-Response model for operations requiring immediate confirmation (such as user authentication, token verification and matchmaking queue entries) and an asynchronous bi-directional messaging model used during active game session to stream inputs and game state snapshots between clients and game servers.

When Messages Are Sent Interactions occur across four main lifecycle phases: during authentication (login, registration or token validation), matchmaking (entering or leaving queues), server provisioning (orchestrating and spawning new game server instances via the Game Server Manager) and active gameplay (continuous client synchronization throughout a game session).

What Data Is Shared The exchanged payloads fall into three primary categories: authentication data (user credentials, public keys and authorization tokens), matchmaking and provisioning data (lobby information, match information and server coordinates), and game data (match lifecycle events, player inputs and authoritative game state snapshots).

3.4.1 Authentication

The Authenticator manages user identities, secures credential exchanges and issuance of cryptographic tokens required for platform-wide access.

User Registration Prior to accessing platform services, new users register an identity with the system, as shown in fig. 4. The client and authenticator initiate a public-key exchange via a SYN and ACK mechanism to set up a secure communication channel. The client then submits encrypted registration credentials. The authenticator decrypts the message, hashes the password and persists the new user in the Long-Term Database. If the username already exists, an error response is returned to the client, otherwise confirmation is sent.

User Login Registered users login to authenticate themselves and retrieve an authorization token, as shown in fig. 5. The client performs an initial public key exchange with the authenticator using login SYN and ACK messages. Encrypted credentials are submitted, decrypted by the server, hashed and validated against the database. Upon successful authentication, the authenticator returns a signed token to the client, which is used to authenticate requests to other platform services, such as matchmaking.

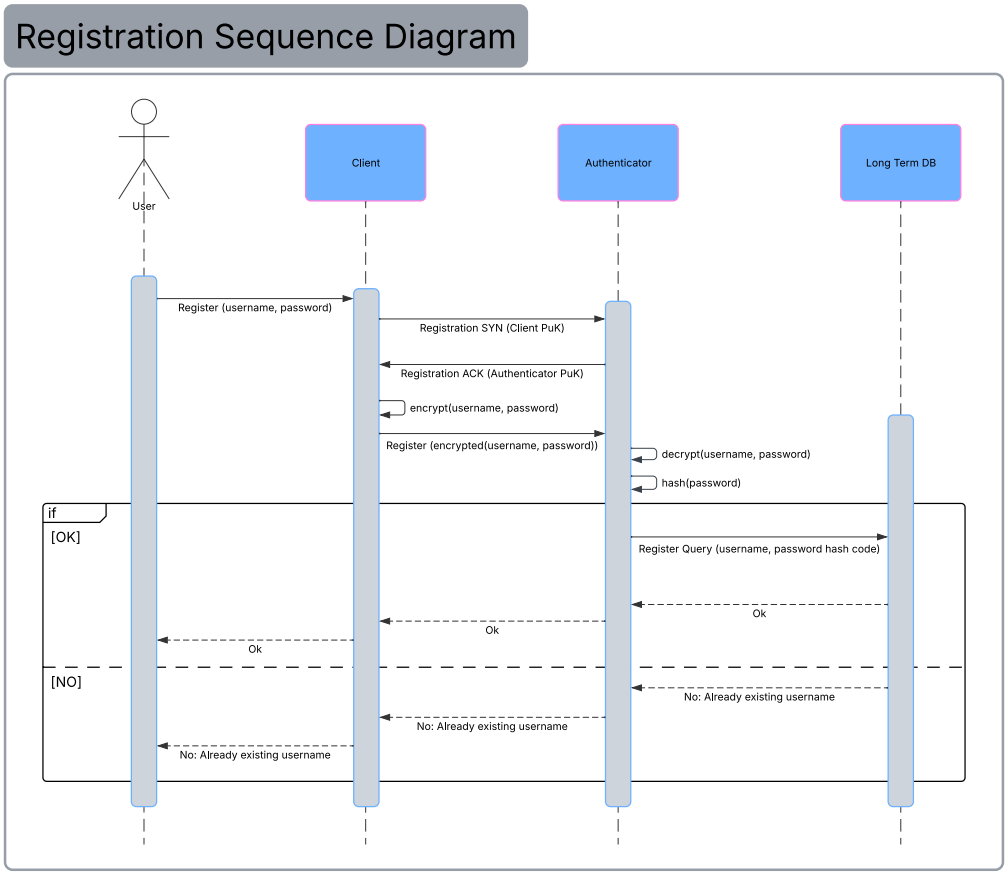

Figure 4: User Registration Sequence Diagram

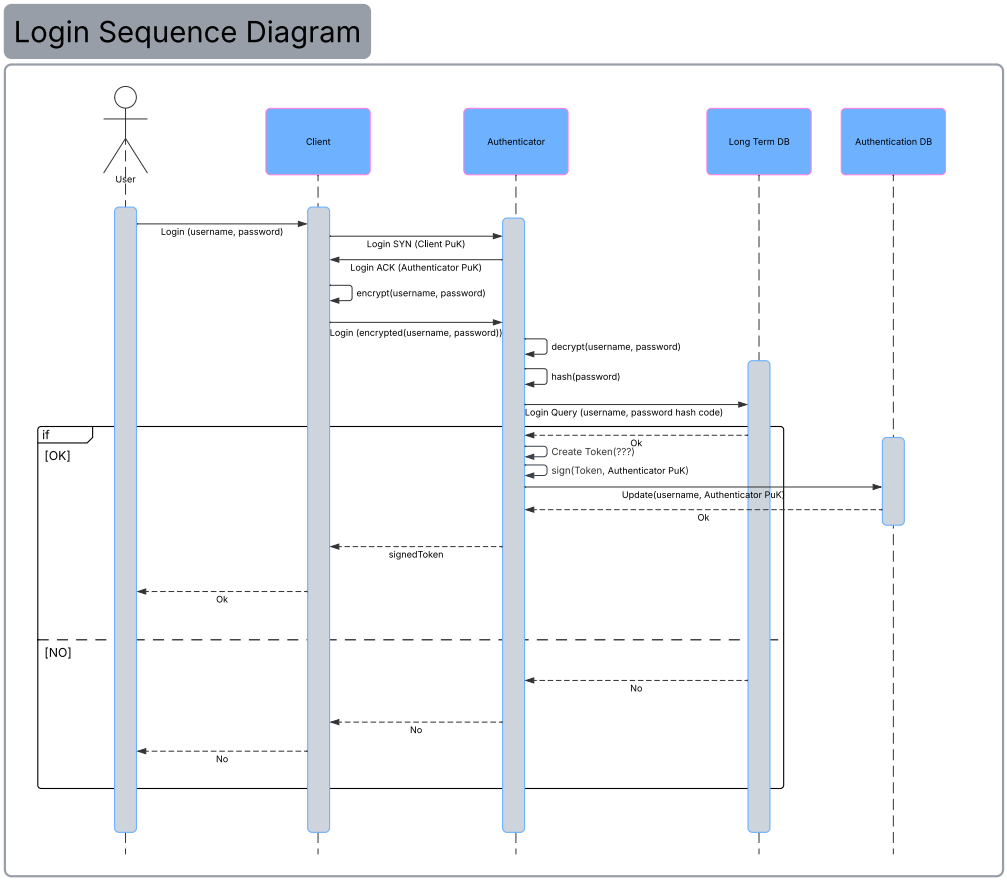

Figure 5: User Login Sequence Diagram

3.4.2 Matchmaking

The Matchmaker Service manages player queuing, game server provisioning, and client routing across three key interactions:

Matchmaker Join As shown in fig. 6, the join interaction initiates when a player issues a request specifying the map they want to play alongside their authentication token. The Matchmaker delegates token verification to the Authenticator to ensure that the player is authorized before proceeding. Once access is granted, the Matchmaker queries the Short-Term Database for existing lobbies matching the requested map. If no available lobby exists, a new one is created. The Matchmaker then evaluates the lobby list and registers the player under one of them. If adding the player causes a lobby to reach its maximum capacity, the Matchmaker turns the lobby into a match.

Matchmaker Quit As shown in fig. 7, a player may voluntarily withdraw from matchmaking prior to match creation. The client issues a quit request to the Matchmaker, which in turn invokes its removal from the lobby in the Short-Term Database. The database updates the lobby state and returns the updated count of remaining players. Upon receiving confirmation, the Matchmaker acknowledges the request back to the client. If the returned count indicates that the lobby has become completely vacant, the Matchmaker performs an operation on the Short-term Database to delete it.

Matchmaker Match Start As shown in fig. 8, once a lobby reaches its maximum capacity, the Matchmaker turns it into a match. The Matchmaker first instantiates a new match record in the Short-Term Database with the map to be played and the usernames of players from the lobby. It then requests server provisioning from the Game Server Manager, which calls the Cluster API to spawn a dedicated Game Server instance. Upon startup, the newly created Game Server queries the Short-Term Database to fetch its expected list of players. Once set up, the Game Server Manager retrieves the assigned network connection parameters from the Cluster API and returns them to the Matchmaker. The Matchmaker writes these server details into the match record in the Short-Term Database, deletes the lobby and delivers the connection parameters to the clients, allowing them to establish a direct connection with the dedicated Game Server.

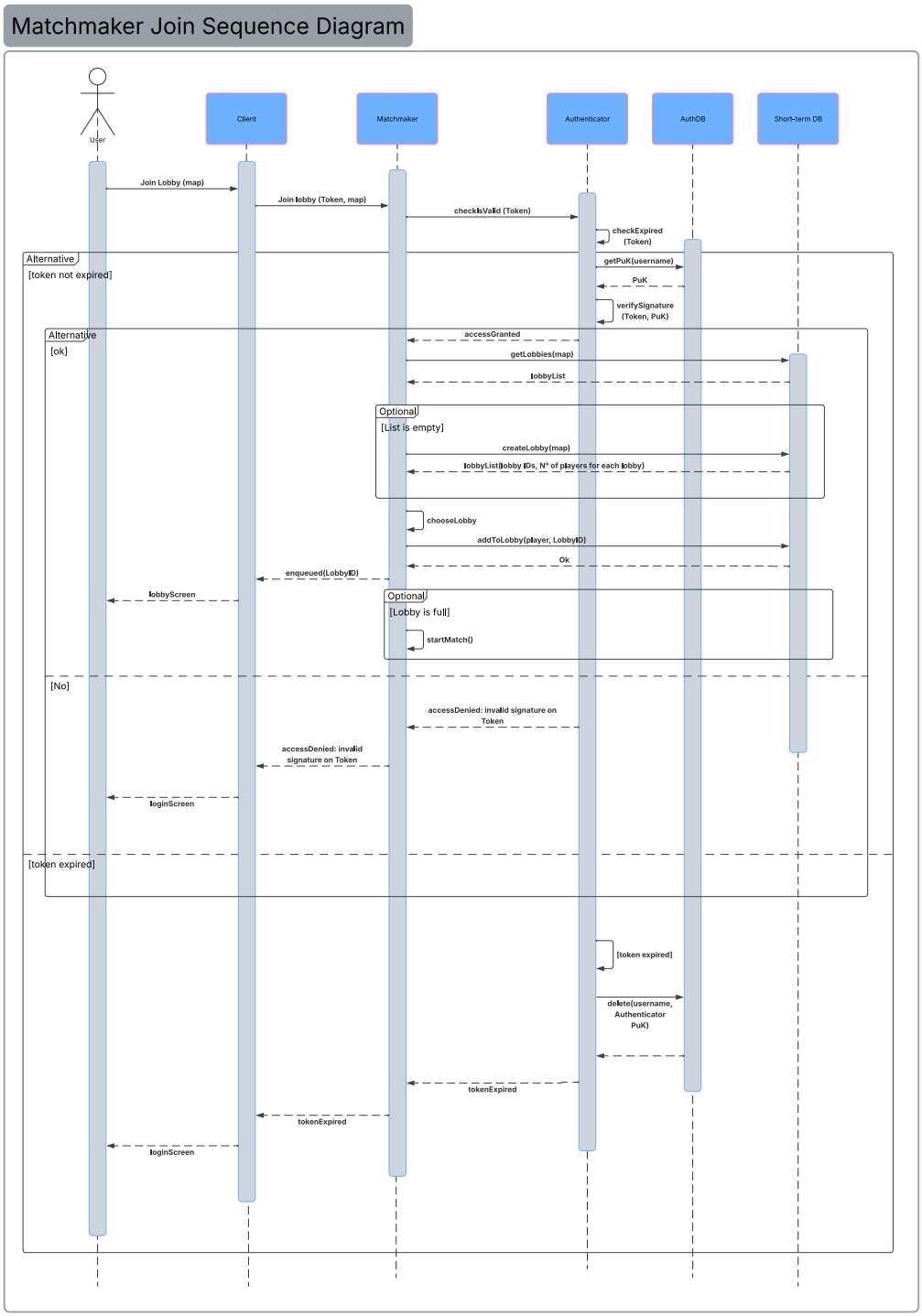

Figure 6: Matchmaker Join Sequence Diagram

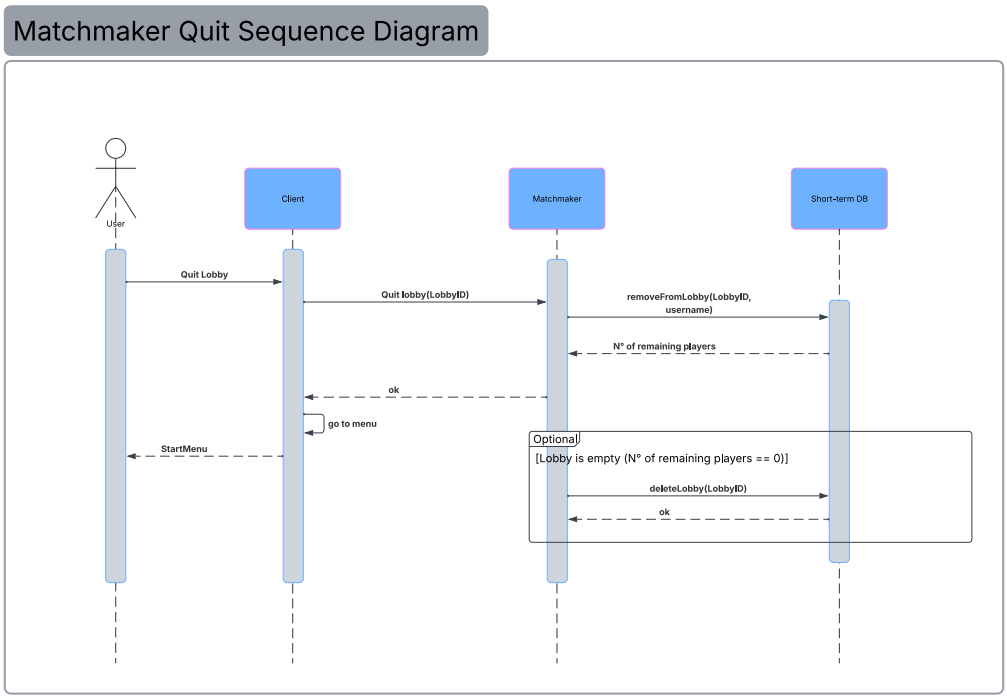

Figure 7: Matchmaker Quit Sequence Diagram

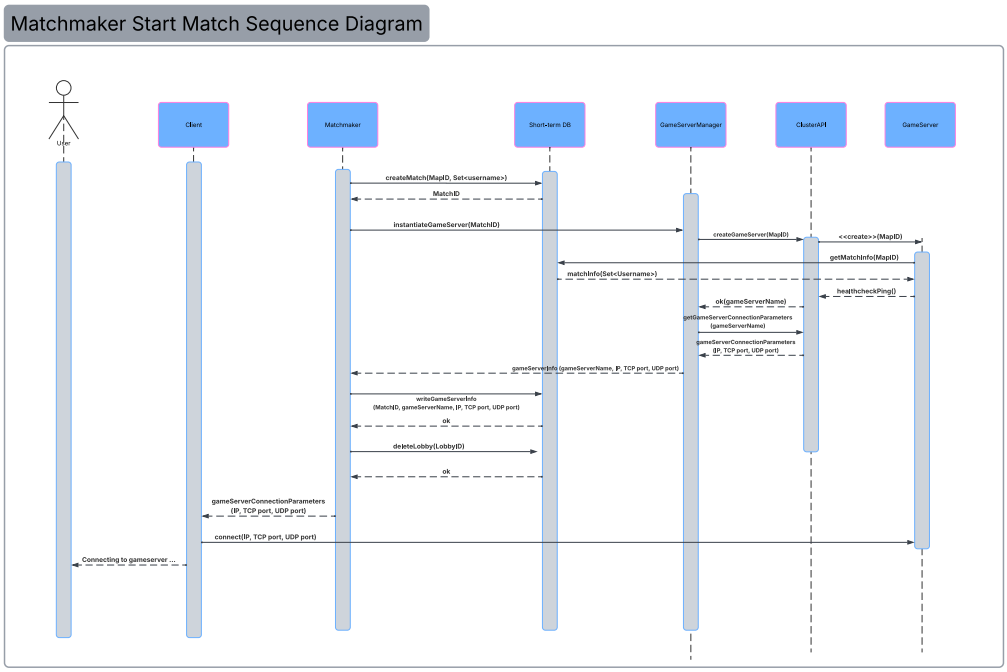

Figure 8: Matchmaker Match Start Sequence Diagram

3.4.3 Match Execution

Once a match is found, clients establish direct communication with the designated Game Server in order to let the user play the game. As shown in fig. 9, the match lifecycle consists of three main phases:

Player Connection The clients initiates a connection to the Game Server by providing the player's username. If not all players have joined yet, the server acknowledges the connection and places the client in a waiting state. Once all expected players are connected or a timer expires, the server sends the initial game state to each connected client and signals the start of the match.

Game Loop During active gameplay, the Game Server continuously executes the authoritative game simulation logic, regularly sending game state snapshots to synchronize the clients. In turn, the clients streams player input commands to the server to move the assigned Pacman character. To support game state recovery, the server asynchronously saves game state checkpoints to the Short-Term Database at fixed intervals.

Match Termination When the match finishes, the Game Server notifies the connected clients. Results are saved to the Long-Term Database and delivered to the clients. The Game Server sends a termination message to the Game Server Manager to signal readiness for deallocation. Finally, the Manager notifies the Matchmaker that the match has ended, and the Matchmaker clears the match data from the Short-Term Database.

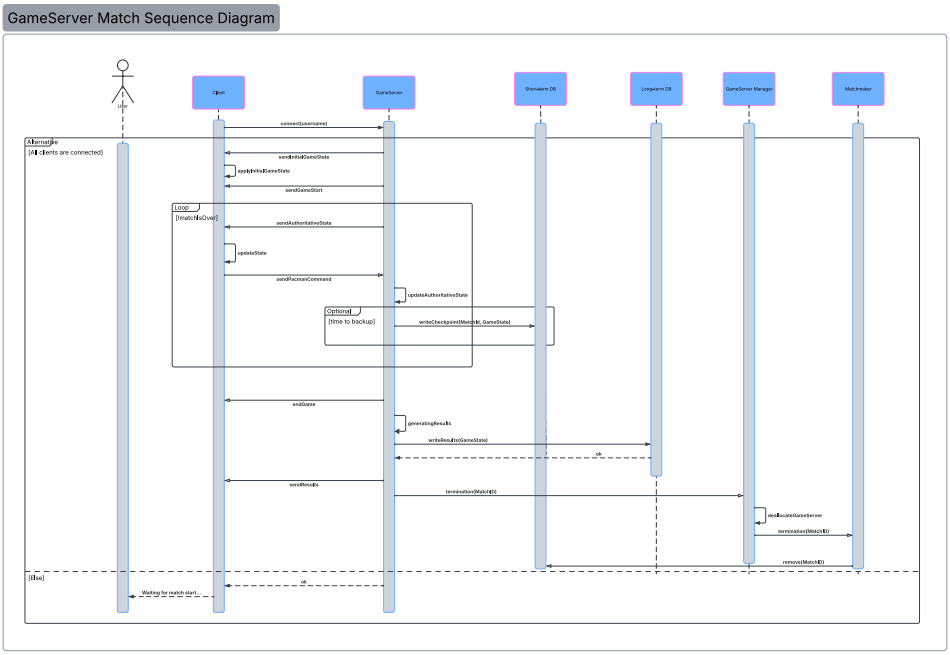

Figure 9: Game Server Match Execution Sequence Diagram

3.5 Behaviour

Game Client The Game Client is a stateful GUI-based component that allows the user to interact with the overall system by interfacing with backend services to perform authentication operations, matchmaking operations, player statistics retrieval, and real-time gameplay simulation and visualization. It essentially coordinates these operational behaviors across two main operational phases:

- At a global level the client does the following in response to user actions and backend responses:
  - Upon successful login or registration, it grants access to matchmaking, player statistics inspection and logout actions.
  - Performs an ongoing match check as long as the user is authenticated. If an active session is detected, it prompts the user to either rejoin the ongoing game or explicitly abandon it.
  - Handles matchmaking requests, allowing the player to queue for a match on a selected map or cancel the queue process before the match is found.
  - Connects to the assigned Game Server once a match is found and transitions into active gameplay.
- During an ongoing match, the client runs a game simulation loop mirroring the server's one:
  1. It applies authoritative game state snapshots received from the Game Server to synchronize its local state.
  2. It executes local player commands while also transmitting them to the server.
  3. It renders the updated local game state to the user.

Game Server The Game Server is a stateful component responsible for hosting individual match instance, enforcing authoritative game logic execution and managing player sessions through four key behaviors:

- It runs the game simulation loop during active gameplay where it:
  1. Processes all player commands received from connected clients.
  2. Advances game physics, evaluates entity interactions, updating the overall authoritative state.
  3. Checks win conditions and broadcasts game state snapshots to clients for synchronization.
- It manages player connections depending on three states:
  - **Waiting**: The server initializes all Pacman characters with the usernames of the expected players. Once all players connect, or after a timeout elapses, any missing player character is placed under bot control and the match begins.

  - **Playing**: Handles mid-game player disconnections by temporarily handing control of the player's `Pacman` over to a bot. When a player reconnects, the server sends the latest game state snapshot so the client can realign its local state and regain control of their character.
  - **Finished**: Terminates the simulation loop, saves the results and rejects any subsequent connection request.

- It saves intermediate game state checkpoints to the Short-Term Database during gameplay and writes aggregate match results to the Long-Term Database upon completion.

- It coordinates with the Game Server Manager once the match ends, signaling completion and readiness for deallocation.

Authenticator The Authenticator is a stateless component dealing with the login/register phase where the user inserts some credentials that will be checked. Also, it manages the token's creation and the sharing with the user who logged in. Finally, it is responsible for checking the validity and authenticity of the token when requested by the other services to understand wheter the user who provided the token is authorized or not to perform some operations or requests.

- **Login-Register**: before *login/register* phase the user will send a message to the authenticator to perform a SYN phase necessary to exchange the public keys of the two endpoints. This approach will guarantee a secure communication between the user and the authenticator for user credentials that will be trasmitted. Suddenly, once the user receives the public key of the authenticator, the user will send a different message if it is interested in loggin in or register its account:
  - `Login` message. Firstly, it will decode the incoming message with its own private key and then it will perform some checks to understand whether the user has inserted the correct credentials. *Positive outcome* returns the encrypted token while a *negative outcome* will give a negative response.
  - `Register` message. Just like the login phase, it decodes the message received. In this case, the authenticator will need to check on the database whether the user with the inserted credentials is already saved. *Positive outcome* will return a message indicating that the user is already registered. Instead, with a *negative outcome* the username and password (previously encrypted) will be stored and the user will be notified of the correct adding.

- **Token**: this message is received to the Authenticator from the Matchmaker and the Queries services when they want to perform checks on the token that some user sent to them.
  1. Check on the expiration date and the correctness of the issuer in the token.
  2. The public key of the Authenticator service which signed the token is obtained from the authentication database.

3. Check on the validity of the sign by means of Digital Signature Algorithm.
4. If the check return no errors then an `OK` message is sent to the sender, otherwise a `Bad Request` error.

Matchmaker The Matchmaker is a stateless component in the system that manages the *lobbies* of the users waiting for a match and it is responsible of notifying the `GameServerManager` for the creation of a `Game Server` to host the new match. Also, it can remove a user who is not more interested staying in the queue lobby. Finally, once a match is over and the GameServerManager send a specific request it can delete all the informations about a specific match stored on the short-term database and, also, a player can be removed in the match informations if he quits the ongoing match in case he wants to play a different ones.

- **Join\Quit Lobby Message**: when the Matchmaker receives a Join\Quit request from a client, firstly it will perform checks on the token to be certain that the user is authorized (the Matchmaker will make a request to the Authenticator service). Suddenly, with a `join request` the Matchmaker will try to find an acceptable lobby for the player and otherwise a new one will be created, returning the lobby's information to the client. With a `quit request` the Matchmaker will delete the user from the lobby's information on the database.
- **Game Server Message**: in this case the Matchmaker will return the informations about the Game Server instantiated by the GameServerManager, like `IP` address, `UPD` and `TCP` ports to connect with.
- **Quit\Delete Match Message**: the `quit message` is sent from the client when the player quit the ongoing match. While the `delete request`, received by the Matchmaker from the GameServerManager, apprise the Matchmaker that the specific match can be removed from the database.

Queries The Queries service is a stateless component that allows the players to receive their personal statistics over the played matches, like:

- The total number of matches;
- The number of wins achieved by the player;
- The win-rate;
- The best personal score between all the played matches.

When the player send a **Info message** to the service, the Queries needs to check the existence of the specified user and retreive all the informations stored on the *long-term* database. All the communications between the `Client` and the `Queries` are encrypted, included for future cases, whether sensible informations will be provided.

GameServer Manager The GameServer Manager is a stateless component responsible for dynamically provisioning GameServers on demand, monitoring their status over the course of the match, and intercepting GameServers' match ending signals, in order to allow the proper deallocation of terminated matches. In case of a GameServer crash, this component is also responsible for instantiating a recovery GameServer, so that the corresponding match can be recovered from the latest checkpoint. The GameServer Manager reacts to received messages as follows:

- **Create GameServer:** Using the resource creation API provided by the cluster, the GameServer Manager instantiates a new GameServer with the specified input parameters (Match ID and Map). Once the newly created GameServer is live and ready to accept incoming connections, the Manager fetches its connection parameters (IP, TCP port, UDP port) and returns them to the Matchmaker.
- **Check GameServer:** The Manager checks the current status of the GameServer with the specified name. The GameServer's status can be:
  - `HEALTHY`: The GameServer is live and functioning properly.
  - `UNHEALTHY`: The GameServer is in an error state, and cannot continue hosting the match.
  - `NOT_FOUND`: The GameServer with the specified name does not exist anymore.

  If the status is `HEALTHY`, the GameServer Manager does nothing. Otherwise, it decides whether to start a recovery GameServer based on the time left for the corresponding match. If the time left is higher than a pre-defined threshold, then a recovery GameServer is created, and the Manager returns its connection parameters.
- **Match Ended:** After receiving this message from the GameServer for a given match, the Manager sends a similar message to the Matchmaker, signaling the possibility to safely delete the data about the match.

The deletion of a GameServer is not explicitly handled by the Manager, since it happens automatically at the end of the GameServer's lifecycle.

Front-End The Front-End service is a stateless component that serves as the client's entrypoint for interactions with backend services. This component is responsible for routing incoming client traffic to the correct administrative service, based on the URL path of the requests received. Upon receiving a request, the Front-End service routes it to the corresponding backend service, waits for the response and sends it back to the client.

3.6 Data and Consistency Issues

This section will introduce the data that are stored on the different databases and the ways these data are managed.

3.6.1 What Data? Where? Why?

- **Credentials** The credentials consist of all the `sensitive` information, stored on the database managed by the Authenticator, required to perform identity verification durign the login or registration. They are composed by: *username*, *password* and *public-key* of the Authenticator instance which signed the token.
- **Statistics** The statistics refer to all the data relating to the matches played by users. In this case, statistics are stored on a separate database, the `long-term db`, and represent data that users wish to consult to get an overview of their match history, like: *Total Matches*, *Total Wins*, *Win-rate* and *Best Score*.
- **Lobby Information** In the `short-term database`, the Matchmaker service keeps track of the lobby queues such that it can understand if a sufficient number of users are interested in playing a match with a specific map. Each document has different fields: *matchId* (the identifier of the match), *map* (the map's name), *players* (a waiting player list).
- **Match Information** Also in this case, the Match Information are stored on the `short-term database` and contain all the data related to the ongoing matches. They are useful when a player wants to rejoin the previous match after a client crash, or in case a GameServer crash forces the recovery of the match to the last checkpoint saved on the database. To this end, the stored information is represented as: *GameServerName*, *list of users*, *Server parameters* (IP address and UPD-TCP ports), *Time of Creation* (it refers to the moment the match is created) and *list of checkpoints*.

3.6.2 How should data be stored?

Since all the databases are based on `MongoDB`, the data storage is *document-based*. This approach was chosen for its **flexibility**, as data stored in the same database can have a different structure time to time, thereby avoiding the creation and definition of a rigid schema like in the case of relational databases.

3.6.3 Who performs queries on dbs?

Essentially, the Authenticator service, the Matchmaker service, the GameServer and the Queries service perform queries on the different databases. Generally speaking, since MongoDB supports concurrent read and writes, various components of the system can take advantage of this situation to perform operations concurrently.

- **Authenticator**: the service performs queries both for *storing* and *extracting* data on the authentication db. Specifically, during the login, the Authenticator needs to check on the db the existence of the player by comparing the entered credentials with those stored. Furthermore, when a user registers its account in the system, the Authenticator performs some checks and subsequently stores the credentials.
- **Matchmaker**: The Matchmaker performs reads and writes on short-term db in different situations:
  - Lobbies creation, update and removal;
  - Matches creation, update and removal;
  - Recovery of data related to GameServers;
  - User wants to quit a lobby, a match;
  - User wants to reconnect to the previous match (during crash).
- **GameServer**: The GameServer has access to both *short-term* and *long-term* databases. In the former case, the GameServer typically writes checkpoints of the ongoing matches, and retreives data only after a crash has occured, with the intent of restoring the last playable state. In the latter case, the GameServer interacts with the long-term db to store the statistics computed for each player, only at the end of the match.
- **Queries**: The Queries service handles retrieval of `statistics` for a specific user. The query is executed only when the player accesses the *"My Stat"* page and making them visible.

3.6.4 Data shared between components?

There are two cases that imply data sharing between components: the match information and the statistics.

- **Match Information**: `Matchmaker` and `GameServer` share the Match Information needed when a *crash* or a *disconnection* happens. In this situation, the user that wants to reconnect to the previous game will need the GameServer parameters, required to connect to the Server. These parameters are inside the Match Information, managed by the Matchmaker and updated by the GameServer.
- **Statistics**: The Statistics are information stored by the `GameServer` and retrieved by the `Queries` service to satisfy the request of the user to view his match history.

3.7 Fault-Tolerance

This section outlines the fault-tolerance strategies implemented across the system components to maintain service availability, handle network disconnections and preserve game state during component failures.

3.7.1 Data Replication

- A form of data replication is used to achieve data redundancy and high availability. Since all the databases in the project (*authentication db*, *long-term db* and *short-term db*) are based on MongoDB, NoSQL and document-based, this goal was done by means of replica sets.
  The **Replica Set** consist in multiple copies of the data on different members of the set, acting as `PRIMARY` or `SECONDARY` nodes. All the writes are performed only by the primary node, while the secondary nodes make a copy of the new data modified by the primary. Each database has its own replica set and each set is composed by three members: **1** primary node and **2** secondary nodes by default.

3.7.2 Heartbeating, Timeouts, and Retries

- During an active game session, the liveness of the GameServer and the connection between GameServer and Client are monitored by means of two dinstict heartbeat mechanisms:
  - Connection health between Client and Server is monitored via a ping-pong heartbeat mechanism where the Client sends periodic pings and the Game Server responds with pong acknowledgments. Both sides enforce read timeouts so that a lack of inbound network activity within a configured threshold automatically triggers disconnection procedures.
  - The second heartbeat mechanism is used to make sure that backend services (GameServer Manager in particular) are able to monitor the liveness of the GameServer. During its whole lifecycle, the GameServer sends periodic health pings to an API exposed by a sidecar component strictly coupled with the GameServer itself. This sidecar component is in charge of updating the status of the GameServer, based on whether or not it receives the periodic health pings. Using the cluster's resource API, the GameServer Manager can then query this status during GameServer creation and checking.

- When a major network issue occurs, the system provides tailored reconnection paths based on the client state:
  - If the client application is still operational, a reconnect button allows the user to rejoin, redirecting them to the original Game Server or to a recovery one if a backend failure occurred. If the recovery server is currently in the process of starting, the client receives an error message informing the user to try reconnecting again shortly.
  - If the client application crashed completely, the client detects the ongoing match upon user re-authentication and prompts them to either rejoin or abandon the match entirely.

  Upon successful reconnection, the Game Server transmits the latest authoritative game state snapshot to synchronize the client and returns character control to the player.

- Since both standard and recovery Game Servers bind a match to a specific set of expected players, a brief grace timer is started in the *Waiting* state to prevent players who have not connected from stalling the match. Once the timer expires, the match automatically starts, placing any missing player Pacman character under bot control so that connected players can proceed.

3.7.3 Error Handling

- When a player loses connection during a match, a bot temporarily takes control of their character. This allows the remaining players to continue the session uninterrupted until the match completes.
- During an active match, the Game Server periodically writes game state snapshots to the Short-Term Database (e.g., every 10 seconds). In the event of a Game Server crash, the match state can be restored from the latest snapshot on a recovery Game Server with minimal progress loss.
- All stateless backend services are replicated across multiple cluster nodes. This way, a failure in a single replica or even a specific node in the cluster does not result in a complete loss of availability of the corresponding service. In case of a crash, the system is also able to dynamically instantiate new working copies of services, in order to reestablish the amount of computational resources available before the failure.

3.8 Availability

To maintain high service availability the system employs the following load distribution and network partition mitigation strategies:

3.8.1 Load Balancing

- Since backend services are replicated, a load balancing mechanism is put in place for each of them, so as to evenly distribute the workload between all available replicas.

3.8.2 Network Partitioning Mitigation

- During an active game session, client availability and GUI responsiveness under adverse network conditions are prioritized by sacrificing strict real-time game state consistency in favor of lightweight **Speculative Execution**. By running local prediction, the client application ensures its GUI does not freeze even if the connection with the Game Server is hampered while waiting for authoritative game states. Local prediction is restricted to inertial movement along last-known directions and local timer updates (e.g., invincibility duration), while non-deterministic or complex logic such as collision handling, score updates, AI pathfinding and entity state mutations are deferred entirely to the authoritative Game Server.

Game state reconciliation is achieved with the following strategy: while the Game Server periodically broadcasts game state snapshots, the client maintains a single-slot buffer that holds a pending snapshot before applying it. A newly received snapshot is placed into the buffer only if its sequence counter is strictly higher than that of the previous one; if the buffer is empty, the incoming valid snapshot is placed in the slot without overriding anything. Once the client applies the buffered snapshot during its update loop, it also empties the buffer.

However, the client will not predict the game state indefinitely. If a network partition causes Game Server snapshots to stop arriving and no inbound traffic is detected whatsoever for too long, the client recognizes the disconnection and initiates the reconnection procedure.

3.9 Security

Authentication: The project contains a form of *authentication* needed to guarantee the access of the system **only by** the authenticated users. In fact, when a client starts the desktop client application, a login form is shown and it requires the user to compile all the fields to get access. So, in short, to play the game the user **must** have an account or create a new one in the register form. Otherwise, the system denies the access to the user.

Authorization: Given that when the player starts the desktop application he is not authenticated, the requests of login are made by an **ANONYMOUS USER**. The system accepts these types of requests and does its checks as if the requests came from a **ROLE USER**. Once the user gets accepted by the authenticator system, it will receive a token that will be used for all the next requests to `matchmaker` and `queries`.
In fact, the token will be used to **authorize** the user to make a specific request both to matchmaker and queries service. The token needs to be approved by the authenticator, proving its authenticity with the informations that were being stored during the login phase.

Cryptographic schemas:

- *Token verification*: this process is done by the authenticator itself when it receives a validation request from the matchmaker or the queries service. The token can be verified by the means of the **RSA pair key** and the **Digital Signature Algorithm**.
  In fact, at the time of the token's creation, the authenticator signs it with its own `private key` (generating an hash attached to the token) while the `public key` of the specific authenticator is stored together with the credentials of the user. When the authenticator verifies the token, it needs to extract the public key stored on the *authentication database*, indexed at the specific username of token's user and apply that key to verify it.

– *Data encryption*: Data exchanged between the `authenticator` and the `client` during the *login/register* phase are encrypted by the means of **RSA** encryption algorithm. The keys needed to perform this operation are shared during a preliminary `synchronization` phase. Also, **SHA-512** hashing algorithm is used to check the integrity of the shared keys. Everytime the client and the authenticator communicate togheter they perform this *"handshake"* to encrypt future comunications between eachothers.
In particular, guaranteeing security for sensitive data, like credentials. All the passwords stored on the database are encrypted using **Bcrypt** hashing function to make them more resistent against `brute-force` attacks.

4 Implementation

- **Network Protocols**:
  - **HTTP** serves as the default protocol for all non-real-time interactions and administrative operations. RESTful services leverage HTTP to handle user authentication, matchmaking and game server orchestration.
  - The active gameplay phase requires both the Client and the Game Server to communicate bidirectionally using both **TCP** and **UDP** transport protocols to leverage their distinct advantages: TCP guarantees reliability for critical messages, while UDP minimizes latency for high-frequency operations like broadcasting authoritative game state snapshots and transmitting player commands.

    However, establishing bidirectional communication across both protocols simultaneously introduces a key challenge: discovering the client's remote UDP socket address without requiring the client to manually configure local listening ports. To solve this, a custom server-driven handshake protocol was implemented. Once the TCP connection is established between both parties, the server generates a temporary secret token and transmits it to the client over TCP. The client then sends a UDP packet containing the token back to the server, allowing the server to discover and bind the client's remote UDP endpoint for two-way communication. To handle UDP's inherent unreliability, the client periodically retransmits the token over UDP until the server acknowledges that the dual-protocol connection is fully bound.

- **In-Transit Data Representation**:
  - Standard **JSON** is used over HTTP for all primary services and administrative operations.
  - During the active gameplay phase, Concise Binary Object Representation (**CBOR**) is utilized over TCP/UDP sockets since they natively transport raw binary data, aligning with low-level network operations while maximizing bandwidth efficiency. Because the Game Server broadcasts the entire authoritative game state snapshot in a single package, default key-value serialization formats incur in severe overhead. To solve this, the payload schema is optimized to an array-shape structure that strips repeated field names. Benchmarks on a typical game state snapshot payload demonstrate a reduction from 6779 bytes in JSON (and 5081 bytes in standard key-value CBOR) down to just 753 bytes in compact CBOR. Furthermore, as the game simulation executes at 64 Hz, the snapshot broadcast rate is intentionally throttled to 16 Hz. As a result, transmitting a 753-byte payload at 16 Hz consumes only $\sim$12 KB/s of download bandwidth per client, resulting in significant bandwidth savings.

- **Database Querying**:

  - To interact with the databases, NoSQL queries are executed to interact with MongoDB databases.

- **Component Authentication**:
  - Regarding Authentication, a traditional approach based on *credentials* and *passwords* was used to authenticate the users. This is done by comparing the username and the salted-hashed password stored on database with the ones entered by the user.

- **Component Authorization**:
  - The Authorization is `Token-based` and determines whether a user is authorized to perform operations. Every time a components receives a token from a user, this object will be analyzed by the Authenticator to determine the authenticity and the permissions.

4.1 Technological Details

- **Game Client & Game Server**: Both components are built using standard Java features, but leverage **Netty** [4] specifically for the active gameplay phase to take advantage of its asynchronous networking performance over TCP and UDP. For HTTP interactions with RESTful backend services, both rely on Java's built-in `HttpClient`, which was chosen because it fit the system's requirements while remaining simple to use. Additionally, the client utilizes **Java Swing** for its user interface and input handling. Furthermore, the Game Client and Game Server share a substantial portion of the codebase, re-using domain models, logic and network DTOs to minimize code duplication. The Game Server makes use of the Java APIs for MongoDB to execute the queries on the Short-Term database by means of *Documents* based on BSON.

- **Authenticator, Matchmaker & Queries**: All three components were developed using the Java `Spring Boot` framework [3], which is recommended for creating self-contained microservices. The incoming HTTP connections and requests are directly processed by each component, as they act RESTful web controllers, marked with `@RestController` Annotation, being able to handle REST API requests. Regarding database interactions, `Spring Data` provides integration with MongoDB representing the repositories as *interfaces* and the domain entities as *POJOs* (Plain Old Java Objects). For the key management based on the *RSA* encryption scheme, `Java Security` and `Javax Crypto` were used to transmit data securely, both in the Authenticator and Queries services. Furthermore, since the Authenticator is responsible for *creating* and *validating* tokens, `JWT` was used to generate tokens and verify their signatures.

- **Game Server Manager**: Similarly to the administrative services illustrated above, this component exploits the Java `Spring Boot` framework to implement

its REST API. Dynamic instantiation and monitoring of GameServers are based on two technologies:

  - At the cluster level, GameServers are managed using `Agones` [1], a Kubernetes library specifically designed for deploying and scaling game servers on a cluster. The library introduces a custom `GameServer` Kubernetes resource, which represents a stateful `Pod` bound to a specific node and which listens on a given TCP and/or UDP port. Upon creation of a `GameServer` resource, `Agones` automatically selects a TCP and a UDP port inside a specific range (7000 to 7050 in our case) and opens them on the `Pod`. The ports chosen by the library are then mapped to the default ports used by the GameServer, and are thus completely transparent to the component itself. `Agones` also allows to implement the second heartbeat mechanism discussed in section 3.7.2, since it automatically provides the sidecar component responsible for updating the GameServer's status. This component consists of a REST server running inside a secondary container in the GameServer's `Pod`. The GameServer sends health pings to its sidecar by means of Java's `HttpClient`.
  - GameServer creation and monitoring are implemented in the Manager by means of Java's `Fabric8 Kubernetes Client` [2], which allows a component in the cluster to perform CRUD (create, read, update, delete) operations on other cluster resources. This library enables the Manager to create new GameServer instances, query their status, and obtain their connection parameters. The deletion of GameServers is instead managed automatically by `Agones`, upon successful completion of their execution.

- **Front-End**: The routing functions performed by this Gateway are implemented using `Spring Boot`'s `Route Locator`, since it allows to define routing rules in a concise and declarative way, without needing to worry about manual handling of incoming requests.

5 Validation

5.1 Automatic Testing

Since the codebase is written entirely in Java, the **JUnit 5** testing framework was leveraged. The automatic test suite focuses on:

- Game domain testing, encompassing validation of game map parsing, game entity behavior, collision detection and movement rules.
- Verification of game engine startup and shutdown and that the game simulation rate remains properly clamped at or below its target threshold on both Game Client and Game Server.
- Verification of the custom TCP/UDP connection establishment protocol during the gameplay phase, ensuring correct player session initialization and reconnection handling.
- Verification of match lifecycle management, ensuring that from the Game Server point of view, the match starts once player capacity threshold is reached or a timer expires.
- Verification of data preservation as game state is encoded and decoded, whilst also benchmarking CBOR's efficiency against JSON.

Due to the modular repository layout, automated tests for each individual component, where present, can be executed from their respective root directory using the Gradle wrapper:

```
./gradlew test
```

5.2 Acceptance Testing

Due to the inherent complexity of design and implementation of a distributed video game, the majority of testing was conducted manually within a live operational environment. While automated unit tests validated isolated component logic, manual testing provided the primary means of verifying overall system behavior. These manual tests include:

- Running a Game Server process alongside multiple Game Client instances on `localhost` and across different physical machines on a Local Area Network (LAN) to validate whether the overall system works as expected.
- Using a lightweight, standalone Game Client stripped of production services such as authentication, matchmaking and statistics fetching, to isolate and test core gameplay mechanics and Game Server behavior. By allowing manual input of server connection parameters, the simplified client facilitated early-stage development and enabled verification of input responsiveness, game logic correctness and graphical output coherence without requiring the full backend infrastructure.

- Focusing exclusively on the gameplay phase, the open-source tool `clumsy` was leveraged to simulate transient network partitions by injecting artificial latency, jitter and packet drops. Under simulated adverse conditions (e.g., 80 ms latency and a 10% packet drop rate), the fault-tolerance measures proved effective. Despite minor degradation in player experience, the game remained entirely playable.

- The Authenticator's behaviour was tested by means of `curl`, a command line tool for transferring data with URLs. Some example of the tested behaviour are: verification of the correct receipt of sent messages on Authenticator by simulating the Client (*login*, *register* requests), checks on the responses of the service, correct storaging of data on the Authentication database and checks on the validity of the requests (like denying the access or return a message error).

- The functionalities of GameServer Manager and Front-End were tested using `Postman`, a GUI-based API testing tool. Tests with `Postman` allowed to validate the behaviour of each API endpoint of the two services, both during normal operational conditions and during edge cases such as client error or internal exceptions caused by failure situations. Since both Manager and Front-End need to interact with other backend services in the system, interactions were simulated during test scenarios by using `Postman` to create *mock servers* that reproduce the normal activity of other components through pre-defined responses. Both components (Manager in particular) were tested incrementally throughout their development, first by validating the API Controller's behaviour and later by verifying the functionality of the component's business logic.

6 Deployment

6.1 Engineering choices

Due to its great support for replication, scalability, load balancing and fault tolerance, Kubernetes is the technology of choice for the cluster deployment of our system's backend services. As shown in fig. 2, the components of the system are deployed in the following way:

- Each *Internal Service* (Authenticator, Queries, Matchmaker, GameServer Manager, Front-End) is deployed in the form of a `Deployment` with multiple replicas, put behind a `Service` that acts both as a load balancer and as a service discovery mechanism, thanks to Kubernetes' internal DNS.
- *Databases* are deployed through `Replica Sets`, containing 3 replicas for each database: a **main** replica, a **backup** replica and an **arbiter**.
- *GameServers* consist of single, independent `Pods`.
- Since the *GameServer Manager* needs to read and create cluster resources (GameServers), a corresponding `Service Account` bound to a `Cluster Role` was created to grant it the necessary permissions.

6.2 Deployment instructions

This section contains the detailed instructions for deploying the system on a Minikube local cluster.

6.2.1 Installing Minikube with necessary dependencies

The prerequisite for deploying the system is to have Minikube version v1.38.1 installed in our host machine.

Creating a Minikube cluster compatible with Agones:

- **Linux and Mac**:

```
minikube start --kubernetes-version v1.35 -p agones \
  --memory 6000 --cpus 4;
```

- **Windows**: It's necessary to publish the desired Agones port range from Minikube to the host system. So, the command becomes:

```
minikube start --kubernetes-version v1.35 -p agones \
  --ports 7000-7050:7000-7050/tcp --ports 7000-7050:7000-7050/udp \
  --memory 6000 --cpus 4;
```

We will publish both the TCP and UDP port range, since our GameServer communicates using both protocols.

After the first creation of Minikube's custom profile, we can simply start it with:

```
minikube start -p agones;
```

Installing necessary cluster dependencies: Once the Minikube cluster is running, we can install the necessary cluster extensions by using Helm (installed in our host machine). First of all, we must add and update the Helm repositories that contain the extensions. This can be done by running the following commands in a terminal on our host:

```
helm repo add mongodb https://mongodb.github.io/helm-charts;
helm repo add agones https://agones.dev/chart/stable;
helm repo update;
```

The first requirement is Agones. We can install it on our cluster with the following command:

```
helm install agones-release --version 1.59.0 --namespace agones-system \
  --create-namespace --set gameservers.minPort=7000,gameservers.maxPort=7050 \
  agones/agones;
```

It's imperative to make sure that the port range specified in this command corresponds to the one specified during the creation of Minikube's profile.

Another requirement is MongoDB. We can install the necessary operators with the following commands:

```
kubectl create namespace authdb;
kubectl create namespace longdb;
kubectl create namespace shortdb;
helm install mongodb-operator mongodb/mongodb-kubernetes \
  --namespace mongo-operator --create-namespace \
  --set operator.watchNamespace="authdb\,longdb\,shortdb";
```

6.2.2 Deploying the system in the cluster

Cloning the project's repository: The first step is to clone the project's repository in a local directory:

```
git clone https://github.com/MrDorby/distributed-multiplayer-pacman.git
```

Building the projects: In the root directory of the project, execute:

```
./gradlew build
```

This will build the jar files for every subproject.

Building Docker images: After building the jars, we will need to add the corresponding Docker images to Minikube's Docker registry. This can be done by running the provided script for the specific platform from the root directory of the project. The command line arguments to execute the script are as follows:

- **Linux:**

```
./scripts/build-images.sh authenticator-service game-server-manager \
  matchmaker pacman-game queries-service front-end
```

- **Windows (PowerShell):**

```
.\scripts\build-images.ps1 authenticator-service game-server-manager \
  matchmaker pacman-game queries-service front-end
```

Deploying the cluster: The deployment of the cluster can be performed by running the provided script for the specific platform from the root directory of the project.

IMPORTANT: If the clients must be run on different devices as the cluster, then, **before running the script**, it's necessary to modify the file `kubernetes/matchmaker/matchmaker-deployment.yaml` so that the variable `LOCAL_CLUSTER` is set to `false` instead of `true`.

The command lines to execute the script are as follows:

- **Linux:**

```
./scripts/deploy-cluster.sh kubernetes/
```

- **Windows (PowerShell):**

```
.\scripts\deploy-cluster.ps1 .\kubernetes\
```

NOTE: After running this script, wait ***at least 5 to 10 minutes*** *in order to allow the cluster to correctly complete the setup operations. The status of the deployment can be monitored via the command:*

```
kubectl get pods -A
```

Once all pods in the `default` *namespace and all database replicas have the* `Running` *status, then the system has successfully completed the deployment.*

Exposing the front end service: Once the cluster is running, we can expose the front end service via the following command:

```
minikube tunnel -p agones
```

The terminal in which we run this command must be kept open, so that the tunnel stays active. Also, to make sure that the client is able to access the front end service, we will need to add the following line to our host machine's `/etc/hosts` file:

```
<cluster ip> multiplayer-pacman.unibo.it
```

If we are testing the client in the same host as the cluster, the line will be:

```
127.0.0.1 multiplayer-pacman.unibo.it
```

Once that is done, we can finally play the game (see section 7).

Cluster cleanup: To stop the cluster, we will first have to stop Minikube's tunnel, by stopping the corresponding process in the active terminal.

After that, we can remove all resources from our Minikube cluster. This can be done in a similar way as the deployment operation:

- **Linux:**

  ```
  ./scripts/cleanup-cluster.sh kubernetes/
  ```

- **Windows (PowerShell):**

  ```
  .\scripts\cleanup-cluster.ps1 .\kubernetes\
  ```

Finally, we can stop the cluster with the following command:

```
minikube stop -p agones
```

We can also remove every trace of the cluster's container with the following command:

```
minikube delete -p agones
```

7 User Guide

The following instructions describe how an end user interacts with the client application to join and play a match once the backend infrastructure is set up and running.

Requirements

To run the application, the system must have a Java 25 JDK or JRE installed.

The executable client application can be downloaded from the **Releases** section of the GitHub repository. Once downloaded, the client can be launched either by double-clicking the executable or by executing the following command in the terminal:

```
java -jar pacman-game-1.0-full-client.jar
```

7.1 Login

Upon opening the application, the landing page is presented to handle authentication. Users can either enter existing credentials (username and password) to transition to the Main Menu upon successful authentication or go to the registration page.

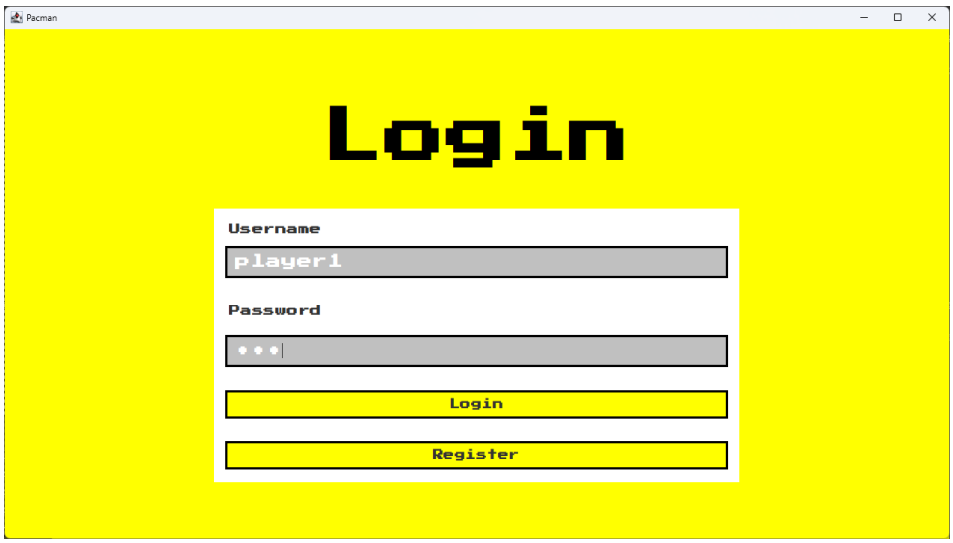

Figure 10: Login Page

7.2 Registration

New users can create an account through the registration page. Submitting the required fields creates the account and redirects to the login page, where the credentials must be entered again to authenticate.

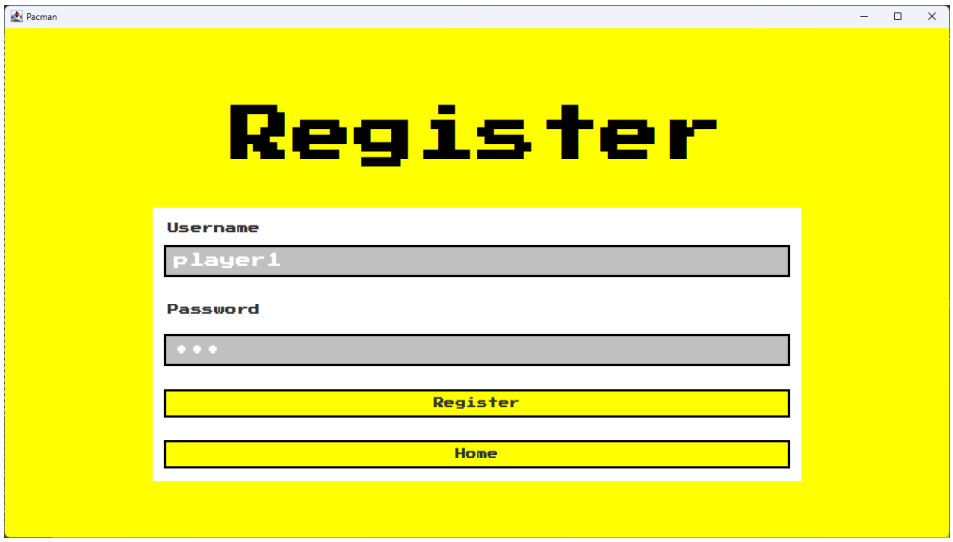

Figure 11: Registration Page

7.3 Main Menu

This page serves as the central navigation hub upon logging in. From here, options to queue for a match, view own statistics and logout are provided.

7.4 Player Statistics

This page displays profile metrics fetched from the backend, including total matches played, win count, win-rate and best score. A button to return to the Main Menu is provided.

7.5 Queueing Page

This page allows selecting the desired map the player wishes to play. Choosing one and pressing **Queue** places the player into a matchmaking state. For a match to be found, at least 4 concurrent players queuing on the same map are required.

Once a match is found, the client transitions into a connecting state with the dedicated Game Server. It remains in this state until the match officially starts, either when all players have joined or the connection window for the other players runs out.

Figure 12: Main Menu Page

Figure 13: Player Statistics Page

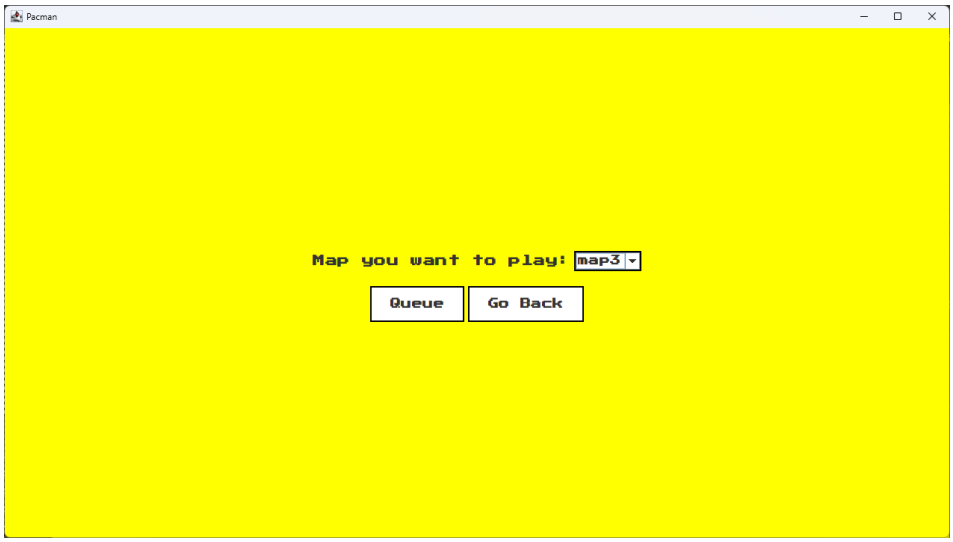

Figure 14: Queue Page

Figure 15: Connection to the Game Server

7.6 Gameplay Page

This page presents the real-time match UI where the player controls a blue Pacman character competing against three yellow Pacman opponents.

Movement across the map is performed using the WASD or arrow keys, allowing players to consume green dots to increase their score, which is tracked alongside remaining lives on the leaderboard to the right.

A **[B]** prefix before a username in the leaderboard indicates that a bot has taken temporary control of their character, either because the player failed to join or disconnected.

A red Ghost navigates the maze and colliding with it results in losing a life and gaining temporary invincibility, during which the player's Pacman color is temporarily dimmed, whereas consuming an orange dot allows temporarily disabling the Ghost on impact.

The match concludes either when the match timer displayed in the top-left corner reaches zero or when only one player with remaining lives is left. The match can be exited at any time by pressing the close button (**X**) in the top-right corner.

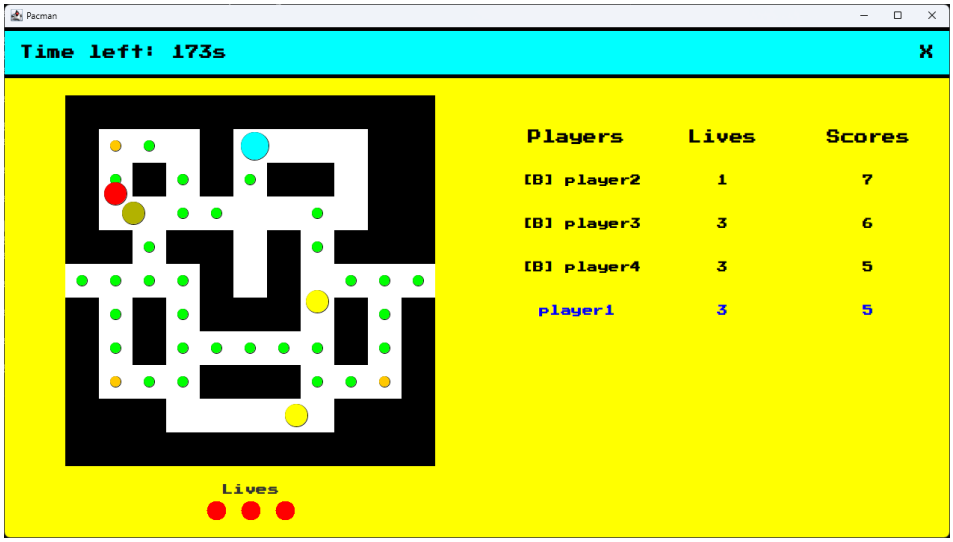

Figure 16: Real-Time Gameplay

7.7 Game Over Page

This page displays the final results upon the completion of a match, declaring the winner alongside the scores of all players. Pressing **Go Back** returns the user to the Main Menu.

7.8 Soft Reconnection Prompt

The Soft Reconnection prompt is triggered when a transient network failure occurs while the client application remains functional during an active game session.

Pressing **Reconnect** instructs the client to attempt to reconnect to the Game Server and resume gameplay, whereas pressing **Go Back** returns the player to the Main Menu.

Figure 17: Game Over Page

Figure 18: Soft Reconnection Prompt

7.9 Hard Reconnection Prompt

The Hard Reconnection prompt is shown after a match is quit abruptly (e.g. application crash). Upon entering the Main Menu, the client detects any ongoing game session, offering options to either reconnect and reinstate the player directly into the match, or completely abandon it.

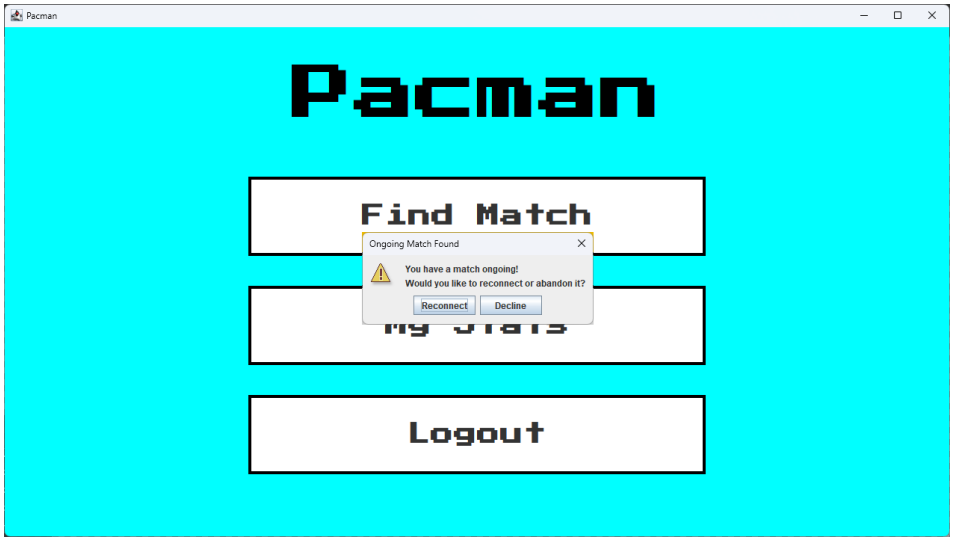

Figure 19: Hard Reconnection Prompt

8 Self-evaluation

8.1 Simone Carpi

Regarding the prokect's code development, I focused on the definition and implementation of the different Game Entities (*Pacmans*, *Ghosts* and *Dots*), in the design of most of the GUIs visible in the game and in the handling of some of their interactions. I developed the `Authenticator` service entirely, ensuring a secure access to the system by focusing on the encrypted communication between the Client and the Server and storing the users' credentials in a sepatate database managed by the Authenticator itself. Furthermore, I implemented the `Queries` service as a point of reference for the Client to get access of all the long-term data stored on db, such as match history. Finally, I worked on the `Matchmaker` service, which is essential for managing lobbies and matches, as well as connecting Clients to a new GameServer. On the `GameServer` side, I handled the interactions with the short-term databases, defining the queries both for inserting and retreiving data, while on the `Game Client` I partially focused on the interfaces for the different services and on the key management to encrypt data when communicating with some them. In conclusion, I defined containers for the various `Replica Sets` present in the project and I partially contributed to the definition of some of the `manifests` for Kubernetes. Looking at the final product, I feel confident on the Authentication part where it assures confidentiality of the stored information (credentials like *username* and *password*) and aims to prevent brute-force attacks through the use of *salt values*, guaranteeing that only registered users can access the system. On the other hand, a main weakness is the definition and the managing of the MongoDB Replica Sets within the Kubernetes environment. Lack of tutorials and my own inexperience in this area made more difficult to achieve the desired result.

8.2 Alessandro Ronchi

Throughout the development of the project, I actively took part in the definition of the game's domain, as well as in the design of the architectural and infrastructural aspects of the system and of the interactions between the different components, both during normal operation and during failures. For what concerns implementation, I programmed part of the domain entities that constitute the game's core, including the GameMap and the movement system used by MovableEntities. As for the other system components, I designed and fully implemented the GameServer Manager and the Front-End service, while also taking care of all the aspects regarding the usage of Agones for the management of GameServers. During deployment, I defined most of the Kubernetes resources necessary for the proper cluster instantiation of system components, specifically those related to stateless internal services and GameServers. Overall, I believe I contributed to creating a solid system design, and the components I implemented have been shown to correctly behave as expected, thanks to incremental testing. One thing I would improve is related to the design of the GameServer Manager's API: separating the CheckGameServer endpoint into two different endpoints, one for querying the GameServer's status and one for instantiating a recovery GameServer, would have probably been a more robust design,

since it would have made it easier for other components (especially the Matchmaker) to properly handle specific edge cases.

8.3 Jiekai Sun

During the development of this project, I partially contributed to the definition of the game domain and helped with the design of the client GUI, while taking primary responsibility for the implementation of both Game Server and Game Client logic, focusing extensively on the network communication layer and game loop. Specifically, I designed and implemented the custom TCP/UDP connection establishment protocol, player session management and game state synchronization mechanism. Looking at the product, key strengths include the effectiveness of its fault-tolerance measures, such as speculative execution under adverse network conditions, which keep the game playable for the user, as well as a Game Engine implementation that is robust enough to meet the project's requirements. Conversely, a primary weakness is that the overall netcode implementation remains somewhat rudimentary, stemming from my inexperience in the field and the considerable difficulties encountered while working with Netty.

9 Future works

The current version of the system deploys internal services using a fixed number of replicas for each `Deployment`. While this still allows some level of scalability, thanks to `Spring Boot`'s natively concurrent architecture, it might not be enough to sustain sudden and considerable spikes in player activity. To fix this issue, it would be necessary to introduce a form of horizontal autoscaling, able to dynamically increase the number of replicas during workload spikes and decrease it once player activity diminishes. This can be done by means of Kubernetes' `Horizontal Pod Autoscaler`.

References

[1] Agones documentation. `https://agones.dev/site/docs/`.

[2] Fabric8 GitHub repository. `https://github.com/fabric8io/kubernetes-client`.

[3] Java Spring Boot documentation. `https://spring.io/projects/spring-boot`.

[4] Netty framework. `https://netty.io/`.