Nomi, Cose, Città

Final Report for the Distributed Systems Course [A.Y. 2024/25]

Chiara Giangiulli
chiara.giangiulli@studio.unibo.it

Giovanni Pisoni
giovanni.pisoni@studio.unibo.it

Dilaver Shtini
dilaver.shtini@studio.unibo.it

March 18, 2026

This project presents a real-time, distributed implementation of the classic italian word game Nomi, Cose, Città. It transforms the traditional pen-and-paper format into a modern multiplayer experience, where players compete in real time to find words that match given categories and starting letters. At the end of each round, participants collaboratively validate each other's answers through an integrated voting and chat system. The game features a distributed architecture that combines client–server and peer-to-peer communication models, ensuring consistent gameplay with fault tolerance and synchronized state across all nodes.

1 Concept

- The application involves creating a simple and intuitive GUI designed to help players easily interact with the system and compete with each other to achieve the highest score at the end of each round.

- Use case description

  The game works as follows: multiple users can join a shared game room, where they are presented with a set of categories and a starting letter. Players must quickly come up with words that fit the categories and begin with the given letter. At the end of each round, participants review and vote on one another's answers to verify their validity. Points are awarded based on the uniqueness and correctness of the answers, and the first player to reach a predetermined score wins the match.

  A game session can start at any time and consist of multiple rounds. Once the session has begun, no new players can join, as doing so would compromise score fairness.

  In each round, players submit their answers and vote on others' responses in real time, under a time constraint that encourages speed and quick thinking.

  Players interact with the system through a graphical interface that provides input fields for submitting answers and offers a chat feature to communicate with others during the game. Throughout the session, the system temporarily maintains data such as usernames and scores, which are required to manage the game state and ensure proper functionality. No persistent storage is needed beyond the duration of the session.

  The game mainly involves two roles: the players and the server. Players actively participate by submitting answers and voting, while the game server manages the overall game state, enforces the rules and aggregates the results.

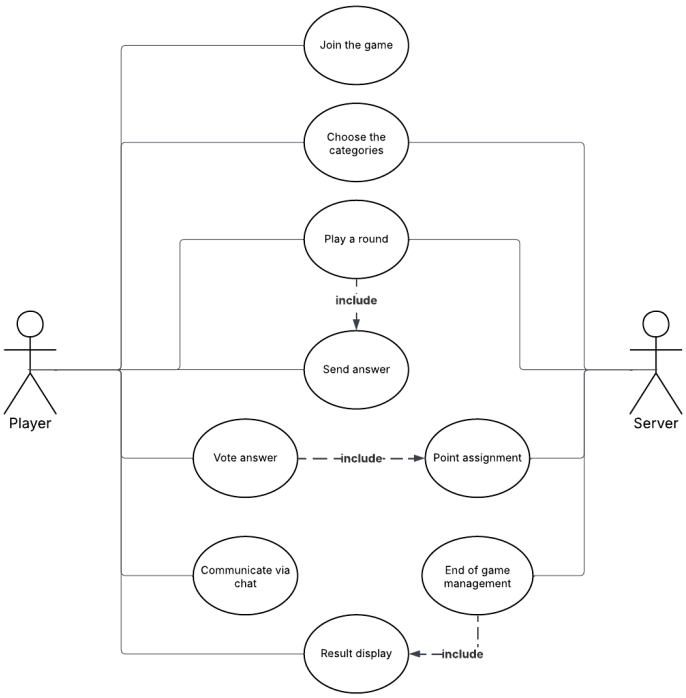

Figure 1: Use case diagram

2 Requirements Elicitation and Analysis

This section details the functional and non-functional requirements for the "Nomi, Cose, Città" distributed multiplayer game.

- **Functional Requirements**:
  1. Players must be able to join a shared game room.
  2. The game must present players with categories and an initial letter.
  3. Players can choose from several game modes.
     - Classic: present only with the default categories (Names, Things, Cities).
     - Custom: the admin chooses the number of categories they want to add (up to 10) to the default ones (Names, Things, Cities) from a predefined list and all the players can vote what categories to include. The server will then start the game using the most voted categories and a random letter. Alternatively, if the admin chooses to exclude the default categories (Names, Things, Cities), players can select only the chosen ones.
  4. Players must be able to insert their answers before a timer runs out.

5. At the end of each round, the system must evaluate answers (check that the first letter is the same as the given one)
6. After the evaluation, players must be able to vote to accept or reject other players' answers.
7. The system must provide a live chat feature for players to communicate during the game.
8. The system must include a point system. Points are awarded for validated answers, and the first player to reach a target score wins. The point system is as follows:
   - 15 points for a valid answer in a category in which no other player has answered;
   - 10 points for a valid answer not used by anyone else in the same category;
   - 5 points for a valid answer but used by at least one other player in the same category;
   - 0 points for an invalid answer.

   The round points are the sum of each category points.

- **Non-Functional Requirements**:
  1. Fault Tolerance: The system must ensure automatic recovery from disconnections and crashes.
     - Server Recovery: must preserve and restore the in-session game state in case of temporary failures.
     - Client Recovery: must support automatic reconnection to the server, re-establishment of P2P connections, and local state caching for seamless recovery.
  2. State Consistency: A synchronized game state must be maintained across all nodes.
  3. Performance: The system must support real-time multiplayer gameplay and chat, with low latency and high concurrency.
  4. Usability: The system must provide a lightweight, user-friendly interface for simple interaction.
  5. Maintainability: Code must be modular and documented to make extensions and changes easy.

- **Implementation**:
  1. System Architecture: The system will use a hybrid architecture with a central server for the authoritative game state and P2P communication between clients, chosen to ensure consistent gameplay and efficient performance.

2. Programming Language: The project will use Python (version 3.0 or higher), selected for its fast development, extensive libraries, portability, and alignment with the course material.
3. Software Tools: The project will use Poetry for dependency management and packaging, as it simplifies setup and ensures reliable, reproducible environments.

2.1 Relevant Distributed System Features

- Transparency

  The system should hide network details from players: they simply join a match and play. However, full transparency is not required. Users may notice some effects of failures (e.g., reconnection messages or delays).

- Fault tolerance and dependability

  Since the system is distributed, components can fail:

  – if the server stops, clients must detect it and try to reconnect;
  – if a client disconnects temporarily, it should be able to rejoin the match;
  – the game state (answers, votes, round results) must remain consistent.

  We aim for good availability and integrity, but the game does not require strict safety guarantees.

- Scalability

  The system must support several simultaneous users in the same room. However, the expected user base is small, so large-scale scalability is not necessary. The design remains scalable in principle but optimized for lightweight usage.

- Security and trust

  The game does not process sensitive personal data, so no advanced security mechanisms are required for the scope of this project. However, a minimal level of security is required:

  – some form of identity (usernames) is needed;
  – input must be validated to avoid malicious or malformed messages;
  – voting integrity is important to avoid cheating.

- Resource sharing

  Multiple clients interact with shared resources:

  – the shared game state (letters, rounds, timers)
  – the shared answer list
  – the shared voting results

  - the shared chat

  This requires coordination and synchronization, especially to ensure that votes are counted consistently, rounds proceed only when all players have submitted and no two players modify the same shared state inconsistently. The server then acts as the synchronizing authority.

- Openness, interoperability, heterogeneity of components

  The system is primarily implemented in Python and does not need to integrate with external services. Cross-platform interoperability is relevant for testing and deployment.

- Evolvability and maintainability

  The system will evolve during development (additional rules, improvements to fault tolerance, UI updates). A modular architecture will help maintain and extend the project.

- Performance and concurrency
  - the server must handle multiple clients concurrently;
  - chat and votes should be delivered timely;
  - network efficiency should avoid flooding or redundant messages.

  High performance is not a strict requirement, but basic efficiency is important for a smooth experience.

- Economy and costs

  The project runs on standard personal computers and has no hosting or deployment costs.

3 Design

This chapter explains the design strategies used to meet the requirements identified in the analysis phase, particularly emphasizing fault tolerance, state consistency, and real-time interaction.

3.1 Architecture

The system adopts a **hybrid architecture** combining client-server and peer-to-peer (P2P) communication models.

- **Client-Server:** The server acts as the central authority. It manages the game session, enforces the timer, and performs the initial check on answers (verifying the starting letter). Finally, it collects the results to update the global score.

- **Peer-to-Peer:** Clients communicate directly during the voting phase to validate the content of the answers and to exchange chat messages. This allows players to reach a consensus without passing every message through the server.

This division of labor improves efficiency. The server is treated as a statekeeper and does not need to process high-frequency traffic like chat or voting debates. Instead, it only receives the final outcome of the round. Additionally, this approach increases resilience: since voting happens between peers, the interaction remains smooth even if the connection to the central server experiences temporary latency.

Alternative Architectures Considered

- Pure Client-Server: A pure client-server architecture was evaluated, where all communication flows through the central server. This was rejected to avoid a total dependency on the central node. In a pure C/S model, a temporary server failure would interrupt all interactions, including chat. The hybrid approach allows social interaction to persist during server glitches.

- Pure Peer-to-Peer (P2P): A pure P2P architecture with distributed consensus was also considered. However, this was rejected due to the complexity of maintaining a consistent global state (e.g., scores and timers) without a central authority. A fully decentralized model would make it significantly harder to synchronize rounds and prevent cheating.

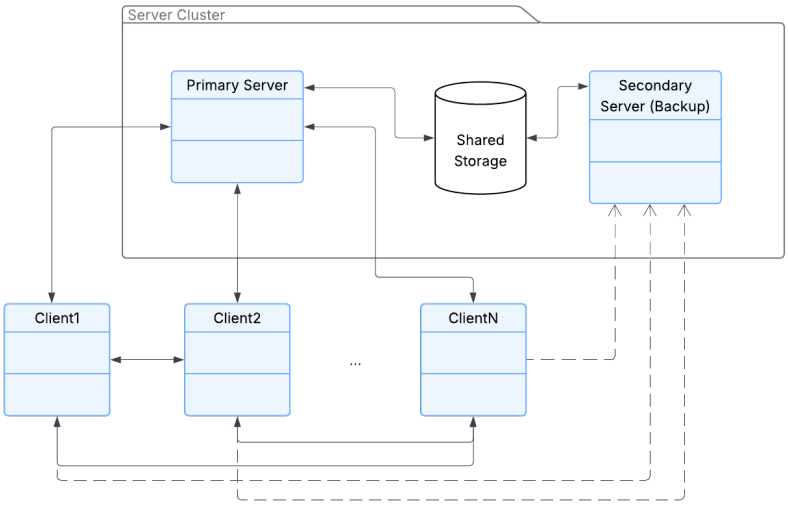

Figure 2: Architecture diagram

3.2 Infrastructure

Given the project scope and the hybrid architecture, the infrastructure is designed to be lightweight.

Infrastructural Components

The system relies on a minimal set of components to reduce deployment complexity:

- **Server Cluster (N instances):** Python processes configured in an Active-Passive architecture. Only the primary node actively hosts the game logic and in-memory session. Instead of a heavy external database, persistence is managed via a shared file-system mechanism, allowing the backup nodes to access the latest game state and recover the session immediately in case of a crash.
- **Client Nodes (N instances):** The players' machines running the client application. Each client acts as a peer in the network, capable of acting as both a sender and receiver for chat and voting messages.

Network Distribution

The components are distributed over a logical LAN.

- The Server and Clients communicate via standard TCP/IP sockets.
- To bypass complex NAT traversal and firewall configuration issues, typical of P2P systems over the public Internet, the system assumes that all nodes are reachable within the same network segment or connected via a VPN.

Service Discovery

Since there is no static registry for the dynamic client IPs, the system implements a centralized discovery mechanism:

- **Server Discovery (UDP Broadcast):** The system implements a LAN discovery mechanism. Rather than relying on static IP addresses in configuration files, the system uses UDP broadcasts. When a client application starts, it listens on a predefined UDP port. The active server announces its presence (IP and TCP port) over the local network, allowing clients to dynamically discover and connect to the authoritative node.
- **Peer Discovery:** The Central Server acts as a dynamic registry. Upon joining the lobby, each client registers its listening socket (IP:Port) with the server. When the round starts, the server broadcasts the *peer map* (a list mapping usernames to their respective addresses) to all participants, enabling them to establish direct P2P connections.

3.3 Modelling

The system domain is modeled around the concept of a synchronized game session where state ownership is distributed according to the hybrid architecture.

Domain Entities and Infrastructure Mapping

- **Game Session:** The core entity representing the match. It resides in the Server's memory and contains the list of connected Users, the configuration (target score, chosen categories), and the global status.
- **Round:** Represents the current turn.
  - *Server:* stores the authoritative data: the chosen letter, the start/end timestamps, and the collection of submitted answers.
  - *Client:* maintains a local replica of the round data to drive the UI.
- **Vote:** Represents a judgment (valid/invalid) cast by Peer A on Peer B's answer. These are exchanged P2P. The Server only receives the aggregated outcome to compute the score.
- **Chat Message:** Entities existing only in the clients' volatile memory and UI buffers. The server does not store chat history.

System State

The state of the system is divided into:

- **Global Authoritative State (Server):** Includes the registry of active players $(ID, IP, Port)$, current scores and the canonical state of the game flow. This state is periodically checkpointed to a local file for crash recovery.
- **Local View State (Client):** Includes the player's own inputs, the cached answers of other players (received for voting), and the chat log. This state is reconstructed from the Server upon reconnection.

Domain Events and Messages

Communication relies on serialized JSON messages, categorized by interaction type:

1. **Commands (Client → Server):** Messages requesting a state change.
   - `CMD_JOIN`: Request to enter the room.
   - `CMD_START_GAME`: Request to start the game.
   - `CMD_SUBMIT`: Sending the filled word list for the round.
   - `CMD_LOBBY_ACTION`: Request to update lobby state, used to sync game settings (admin only) or register category votes (all players).

2. **State Updates (Server → Client):** Events broadcasting authoritative changes.
    - `EVT_LOBBY_UPDATE`: Notifies clients of new players joining the lobby.
    - `EVT_ROUND_START`: Contains the letter and categories.
    - `EVT_PEER_MAP`: Updates the client with the list of other peers for P2P connection.
    - `EVT_VOTING_START`: Signals the beginning of the voting phase.
    - `EVT_SCORE_UPDATE`: Broadcasts the new leaderboard after a round.
    - `EVT_ROUND_END`: Signals the end of a round.
    - `EVT_GAME_OVER`: Announces the end of the game and the final scores.
    - `EVT_ERROR`: Signals an error condition (e.g., invalid command).
3. **Peer Interactions (P2P Client ↔ Client):** Direct messages for social and validation logic.
    - `MSG_CHAT`: A text message for the room.
    - `MSG_VOTE`: A payload containing {target_user, category, approval_boolean}.

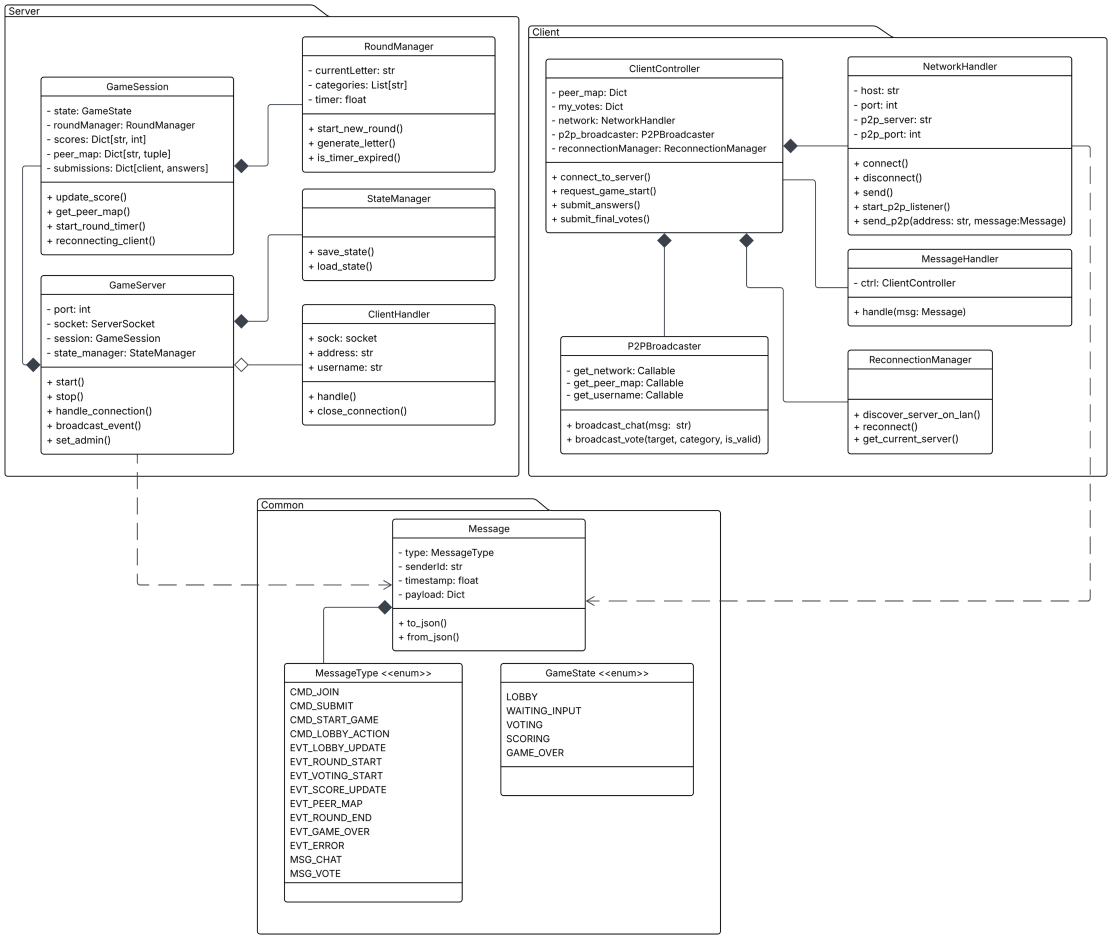

Figure 3: Class diagram

3.4 Interaction

The system employs distinct interaction patterns depending on the game phase, leveraging the hybrid architecture to optimize network traffic and latency. Communication is asynchronous and message-driven, using TCP sockets exchanging JSON-encoded payloads.

Interaction Patterns

- **Client-Server:** Used for state changes.
  - *Request-Response:* Clients send specific commands (e.g., `CMD_JOIN`) and wait for an acknowledgment.
  - *Broadcast (Publisher/Subsriber):* The server pushes state updates

(e.g., `ROUND_START`) to all connected sockets simultaneously.

- **Peer-to-Peer:** used for high-frequency data.

  When a user types a chat message or casts a vote, the client iterates through its *Peer Map* and sends the message directly to all other active clients.

  Unlike the server broadcast, this communication happens directly between nodes (1-hop latency) without passing through the central bottleneck.

Communication Flow

The interaction evolves through the game stages as follows:

1. **Handshake & Discovery (Vertical)** Clients connect to the Server. The Server registers them and, upon game start, distributes the *Peer Map* (IPs and Ports of all players) to every client. This bootstrapping phase enables the subsequent P2P layer.

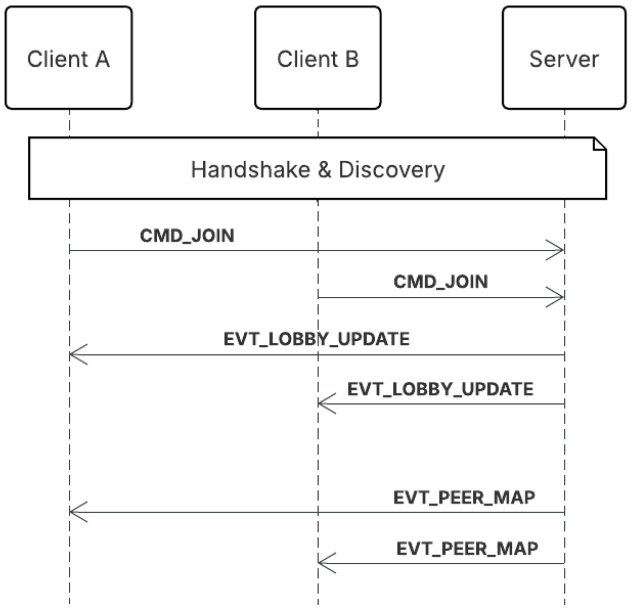

Figure 4: Handshake Sequence diagram

2. **Round (Vertical)** The Server broadcasts the letters and categories. Clients play locally. At the end of the timer, Clients send their answers to the Server via a `CMD_SUBMIT` message. The Server performs the syntactic check.

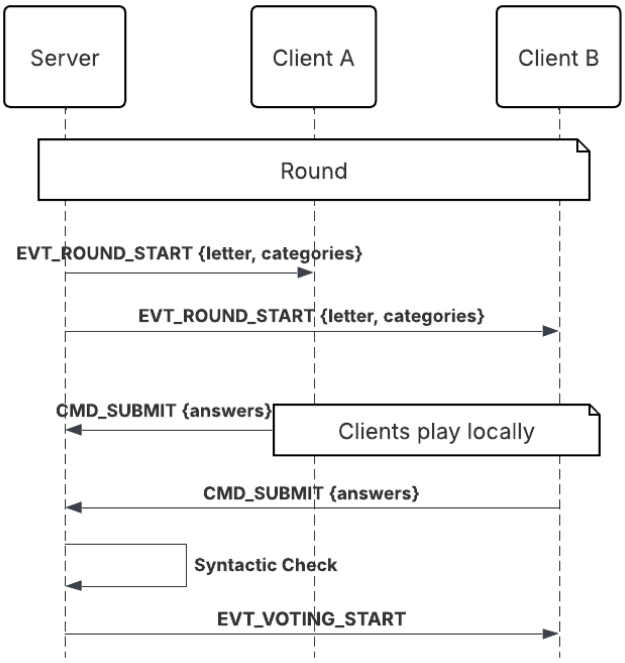

Figure 5: Round Sequence diagram

3. **Voting & Chat (Horizontal)** The Server signals the start of the voting phase. Now, clients switch to P2P mode:
   - *Chat:* a message from a Client is sent directly to others Clients.
   - *Voting:* during the evaluation phase, each client reviews the answers of all other participants. When Client A casts a judgment on an answer belonging to Client B, a `VOTE` packet is immediately dispatched to **all connected peers**. This ensures that not only does B receive the feedback, but third-party observers can also update their local UI to reflect the current voting consensus in real time.

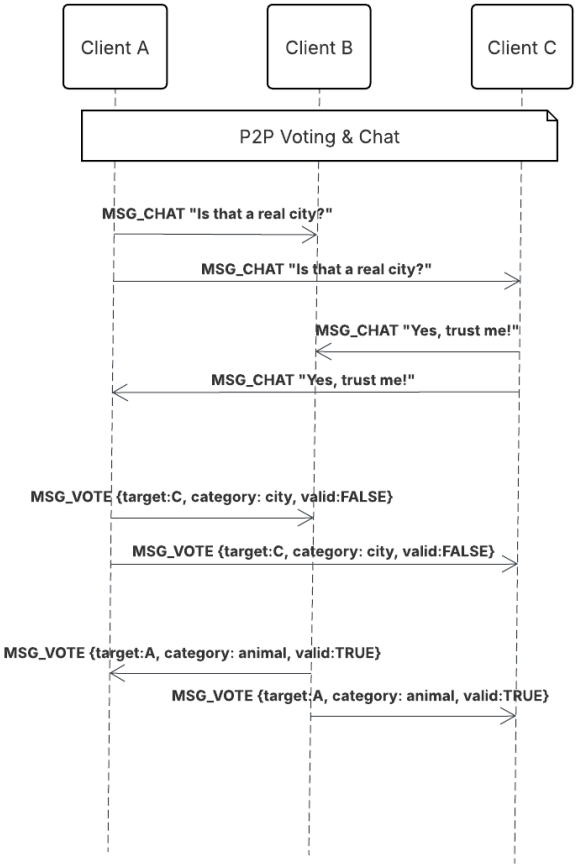

Figure 6: Chat and Voting Sequence diagram

4. **Consensus & Scoring (Vertical)** Once the voting timer expires or an agreement is reached, the Clients submit the final validation results to the Server. The Server aggregates these results, resolves any conflicts (e.g., majority rule), calculates the points, and broadcasts the updated leaderboard.

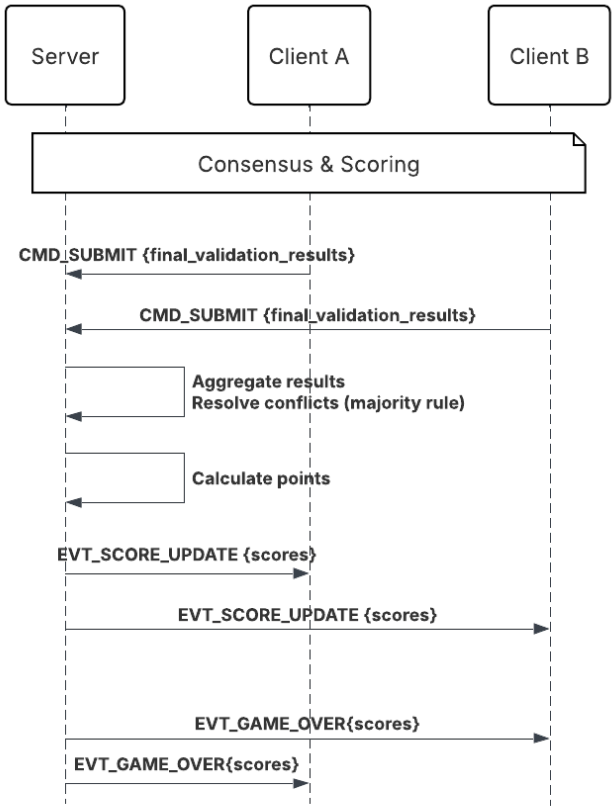

Figure 7: Consensus - Scoring Sequence diagram

3.5 Behaviour

The system behavior is modeled as an event-driven architecture where the Server acts as the central **Finite State Machine (FSM)** driving the game flow, while Clients behave as reactive consumers synchronized to the server's state transitions.

Server Finite State Machine

The Server behavior is defined by five distinct states. Transitions are triggered by timers or completion events.

1. `LOBBY`: The initial idle state. The server accepts `CMD_JOIN` requests and maintains a list of connected peers. It waits for a manual "Start Game" command.

2. `WAITING_INPUT`: The active gameplay phase. The server has broadcasted the categories/letter and started the round timer. It listens for `CMD_SUBMIT` messages

containing the players' answers. This phase ends when the timer expires.

3. `VOTING`: The distributed validation phase. The server broadcasts the aggregated answers and instructs clients to switch to P2P mode. During this state, the server is logically "waiting" for the consensus result. This phase ends when the voting timer expires or all clients signal completion.

4. `SCORING`: A transient processing state. The server aggregates the validation data received from clients, calculates points based on the rules, and updates the global leaderboard. It automatically moves to `GAME_OVER` if the target score is reached, otherwise loops back to `WAITING_INPUT` for a new round.

5. `GAME_OVER`: The terminal state. The final scoreboard is displayed. The session remains open for a defined period before resetting the lobby or allowing a clean disconnection.

Client Reactive Behaviour

The Client's behavior changes based on the current global state broadcast by the server:

- During `WAITING_INPUT`, it acts as a producer, capturing user text and sending it to the server.
- During `VOTING`, it switches role to a peer-node, opening direct connections to other clients to exchange validation messages and chat.

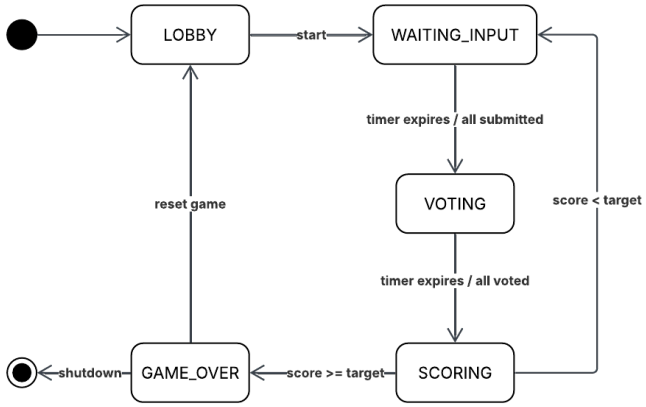

Figure 8: States Diagram

3.6 Fault-Tolerance

To ensure the continuity of the game session despite failures, the system implements a redundancy strategy covering both server availability and data persistence.

Server-Side Recovery

The system adopts an **Active-Passive** replication model based on shared storage. Instead of relying on a static configuration, server nodes are completely autonomous and auto-assign their roles upon startup. The high-availability logic relies on a "Heartbeat Lock" mechanism located in a shared folder:

1. The **primary** server continuously updates the lock's timestamp to signal that it is alive.

2. Any number of **backup** servers can be launched. They automatically enter a monitoring loop, periodically checking the timestamp of the lock file.

If the primary crashes, it stops updating the heartbeat. The backups detect the timeout, assume the primary is dead, and attempt to promote themselves to active status by binding to the game server's TCP port. To prevent race conditions when multiple backups try to take over simultaneously, only the first instance that successfully binds to the port becomes the new primary. The others revert to their backup monitoring loop, ensuring fault tolerance without conflicts.

Client-Side Recovery

Instead of relying on a static list of server addresses, the client features a dynamic reconnection manager integrated with the UDP discovery mechanism. If the connection to the current server drops, the client enters a resilient reconnection loop based on a configurable retry policy. This window provides the server-side failover mechanism enough time to complete its election and for the backup node to bind to the required port. Once the backup becomes the new primary, the client automatically re-establishes the connection.

Upon reconnecting, the client sends a handshake with its username. The new server uses this ID to link the new connection to the saved player data, allowing the user to continue playing as before.

When a client socket disconnects unexpectedly, the server does not immediately evict the player from the match, it marks the user as disconnected but keeps their object (score, username) in memory.

If the user restarts the client application and attempts to join with the same username within the period, the server detects the match.

Upon successful re-binding, the server sends the current game state to the client, allowing the user to resume playing.

3.7 Data and Consistency Issues

Given the session-based nature of the application, the data management strategy prioritizes low-latency access (in-memory) over long-term archival storage.

Data Storage Strategy

- **Transient Data:** The primary server utilizes its Random Access Memory (RAM) to store the live `GameSession` object (connected users, current scores, FSM status and active peers). This choice ensures fast access times required to update scores and handle round transitions in real-time. Standard Python data structures are sufficient and performant for these data types.
- **Persistent Data (Shared File System):** To implement Fault Tolerance described in Section 3.6, the system persists state snapshots to a JSON file located on a **shared storage volume** accessible by both server nodes. This file acts as the synchronization medium, allowing the backup node to load the latest valid game state immediately upon taking over.

Consistency Model

Data must be shared to maintain a synchronized view of the game. The system adopts a **Hybrid Consistency Model**:

- **Strong Consistency (Server-Side):** The active server acts as the "Single Source of Truth". Clients receive authoritative updates via broadcast events and cannot modify the global state directly. Furthermore, the shared state file ensures *Sequential Consistency* for the backup node, guaranteeing it always loads the most recent write performed by the primary before the failure.
- **Eventual Consistency (P2P-Side):** During the voting phase, latency in propagating votes between peers is acceptable. The system guarantees that the state will converge to a consistent result when the active server aggregates all votes at the end of the phase.
- **Voting Default Policy:** During the aggregation phase, the server applies a benefit-of-the-doubt rule for answers that received no votes at all. This can occur when a player disconnects during the voting phase or when the voting timer expires before all peers have cast their judgment. In such cases, rather than penalising the author of the answer for a network or timing issue outside their control, the system defaults to treating the answer as valid. This choice prioritises fairness over strict consistency: it is considered preferable to occasionally award points for a borderline answer than to silently invalidate a correct one due to an incomplete vote collection. Answers already marked as syntactically invalid during the initial server-side check are never subject to this rule, they remain invalid regardless.

3.8 Availability

The system availability strategy is designed to maintain responsiveness and handle load distribution without introducing heavy middleware components.

Caching Mechanisms

Every Client node maintains a local replica of the relevant game data in its own process memory. This minimizes network traffic and improves UI responsiveness. If the server connection drops temporarily, the client retains the last known state, allowing for a smooth user experience upon reconnection.

Load Balancing

The system does not employ a dedicated hardware Load Balancer because the traffic volume for a single session does not justify the added latency and complexity.

The workload is distributed between the central Server and the Client nodes via the Peer-to-Peer layer. Delegating the bandwidth-intensive tasks to the clients acts as a logical load balancer, preventing the single server instance from becoming a bottleneck during the most active game phases.

Network Partitioning Behavior

In the context of the CAP Theorem, the system operates primarily as a CA (Consistency and Availability) architecture under normal conditions. However, in the event of a network partition, the system prioritizes Availability over Consistency. Instead of blocking offline gameplay, the Active-Passive failover mechanism aggressively attempts to keep the game running: isolated backup nodes will automatically promote themselves to primaries. While this ensures that players can seamlessly reconnect and continue playing, it introduces a *split-brain* scenario where multiple servers might accept inputs simultaneously, ultimately compromising the consistency of the shared game state until the partition is resolved.

4 Implementation

This section details the concrete technological choices and protocols utilized to realize the hybrid distributed architecture of the game.

Network Protocols The system relies entirely on Transport Layer protocols to minimize overhead, as the application consists of native desktop clients rather than web browsers. **TCP (Transmission Control Protocol)** is used for all reliable communications, encompassing both the Client-Server FSM state updates and the Peer-to-Peer interactions, such as chat and voting. It is implemented via asynchronous streams to ensure ordered and lossless data delivery.

Conversely, **UDP (User Datagram Protocol)** is employed exclusively for the Server Discovery mechanism. The active server broadcasts its presence (IP and TCP port) via UDP datagrams over the local network, enabling clients to dynamically discover and connect to the authoritative node without manual IP configuration.

Data Representation In-transit data is represented using **JSON payloads** encoded in UTF-8. These payloads are encapsulated within a custom `Message` class, which provides structured serialization to and deserialization from bytes. Each message includes a specific `MessageType` enumeration to efficiently route events and commands through the system's Finite State Machine.

State Persistence and Storage The system operates without a traditional Database Management System (DBMS). Instead, persistent state is achieved via a **shared file system**. A dedicated `StateManager` serializes the FSM state, active players, and scores into a local JSON file (`state.json`). To prevent data corruption during sudden crashes, the system utilizes an atomic file replacement strategy, writing data to a temporary `.tmp` file before safely renaming it. Similarly, the Active-Passive replication mechanism relies on a shared `heartbeat.json` file to monitor server health.

Authentication and Authorization The system prioritizes ease of access and does not employ strict authentication protocols. Players are authenticated upon establishing a TCP connection by providing a unique username in their initial `CMD_JOIN` request. The server enforces uniqueness and tracks sessions based on this string identifier. Authorization is managed via a lightweight **Role-Based Access Control (RBAC)**. The first user to join an empty lobby is automatically granted the `admin` role. Only the admin is authorized to modify lobby settings and trigger the game start. If the admin disconnects, the server ensures continuity by promoting the next connected player to the admin role using a FIFO policy.

4.1 Technological details

The implementation heavily leverages the Python standard library, alongside specific development tools, to maintain a lightweight, concurrent, and robust architecture:

- `asyncio`**:** the core framework driving the system's concurrency model. It manages multiple TCP client connections concurrently on a single thread, orchestrates non-blocking FSM timers for round and voting durations, and handles the client-side resilient reconnection loop without freezing the user experience.
- `socket`**:** utilized directly for the low-level UDP broadcast implementation, allowing the server to announce its listening port and clients to asynchronously scan for discovery datagrams.
- `json` & `os`**:** standard libraries used to serialize complex state dictionaries into UTF-8 files and manage atomic file operations for crash-resilient shared storage.
- `Tkinter`**:** provides a lightweight, cross-platform graphical user interface. Because `Tkinter` requires exclusive control over the main thread, the `asyncio` event loop is offloaded to a background daemon thread. Synchronization between the asynchronous network layer and the UI is handled safely via the `root.after()` method.

- `Poetry`: adopted for modern dependency management and environment isolation, ensuring reproducible builds across different development machines.
- `pytest`: utilized as the framework for unit and integration testing, ensuring the reliability of FSM transitions and state recovery mechanics.

5 Validation

To ensure the reliability, consistency, and fault tolerance of the distributed system, validation was performed using a combination of automated unit/integration testing and manual acceptance testing.

5.1 Automatic Testing

Automated tests were developed using the Python `unittest` framework, heavily leveraging `IsolatedAsyncioTestCase` to handle the asynchronous nature of the network and game loops. Test doubles (`MagicMock` and `AsyncMock`) were extensively used to isolate components.

Unit Testing Individual components were tested in isolation to verify their core logic:

- **Game Logic:** the `RoundManager` was tested to ensure correct letter generation and timer expiration. The `GameSession` FSM was validated to ensure strict transitions (e.g., from `WAITING_INPUT` to `VOTING` only when all answers are received or the timer expires) and correct point calculation.
- **Server Management:** the `GameServer` component was tested to verify client tracking, unique username enforcement, and the FIFO election of the lobby admin.
- **State Persistence:** the `StateManager` was tested by mocking a game session and asserting that the JSON serialization and deserialization accurately reconstruct the game state (scores, active categories, remaining timers).
- **Data Representation:** the custom `Message` class was tested to ensure correct instantiation of properties (sender, message type, payload) and accurate JSON serialization for network transit.

Integration and Interaction Testing The interactions between the networking layer and the application logic were tested to ensure seamless communication:

- **Network & P2P:** the `NetworkHandler` tests validated the proper establishment of TCP sockets and the successful dispatching of JSON-encoded `Message` objects. The `ClientChat` tests verified that Peer-to-Peer messages are correctly broadcasted only to connected peers using the dynamic Peer Map, bypassing the central server.

- **Fault Tolerance:** the `ReconnectionManager` was strictly tested against various edge cases, including parsing malformed configuration files, executing circular server rotation, and respecting retry delays during connection exhaustion.
- **GUI Integration:** the Tkinter user interface was tested headlessly. Mocking the `ClientController`, the tests simulated user inputs in the `LoginScreen` and `LobbyScreen` to verify that the correct network callbacks were triggered upon button clicks.

Test Automation The entire test suite can be executed via the command line using `poetry run pytest`. This provides immediate feedback during development and ensures that new features do not break existing FSM or network constraints.

5.2 Acceptance Test

While unit tests cover the programmatic logic, the distributed nature of the system required manual End-to-End (E2E) testing, particularly to validate corner cases and real-world network conditions.

- **LAN Discovery and Multiplayer Flow:** we performed manual testing by connecting multiple physical PCs to the same Wi-Fi network. This confirmed that UDP discovery packets were correctly dispatched and caught by the clients without any hardcoded IPs, and successfully validated the entire lifecycle of the game (lobby, rounds, chat, and scoring) among real distributed nodes.
- **Active-Passive Failover:** tested by running multiple server instances (Primary and Backup). The Primary process was forcefully killed to simulate a sudden crash. We manually verified that the Backup server detected the missing heartbeat and bound to the TCP port, and that clients connected via the Wi-Fi network seamlessly reconnected without losing their session state or scores.
- **P2P Voting and Chat Latency:** manual testing with multiple computers on the same Wi-Fi network ensured that the asynchronous P2P broadcasting updated the UI of all observing peers in real-time, confirming the responsiveness of the horizontal communication layer.

6 Deployment

The system is designed as a native desktop application and a lightweight server script.

Prerequisites To run the project, target machines must have installed:

- **Python 3.10** (or higher) to support modern `asyncio` features and type hinting.
- **Poetry**, the dependency management tool used to isolate the environment.

Installation Instructions Users must clone the repository and initialize the environment. Opening a terminal in the project root, the following command installs all required dependencies from the `pyproject.toml` file:

```
poetry install
```

Configuration No complex environment variables are required. For the clients, a `config.json` file is automatically loaded at startup to define the retry policy and UDP discovery timeout. By default, the system requires zero manual configuration thanks to the UDP LAN discovery. However, to handle restrictive network environments or to support remote WAN play, the system implements a *graceful degradation* strategy. Users can optionally modify the `config.json` file specifying the authoritative server's static IP. If the dynamic UDP discovery times out, the client's reconnection manager automatically falls back to this configuration. If no file is provided, it safely defaults to the local loopback interface, ensuring immediate out-of-the-box functionality for local development and testing without requiring code modifications. For the server's fault tolerance, the path to the shared storage volume (for the `heartbeat.json` file) can be optionally modified in the `constants.py` file depending on the host OS network sharing setup.

7 User Guide

The application offers a plug-and-play experience. Below are the steps to initialize a session and play:

1. **Start the server:** open a terminal and run the central game server:

   ```
   poetry run python src/server/main.py
   ```

   Launch the exact same command on a secondary terminal or machine to instantiate a Backup server for fault tolerance.

2. **Launch the client:** on the players' machines, start the graphical client:

   ```
   poetry run python src/client/main.py
   ```

3. **Login phase:** the application will automatically scan the LAN for the active server via UDP. Once found, the user simply enters a unique username to join.

Figure 9: The Login Screen.

4. **Lobby:** players gather in the lobby. The first player to connect is designated as the Admin and is granted UI controls to select the game mode, the extra categories, and the round timer. Once everyone is ready, the Admin clicks "Start Game".

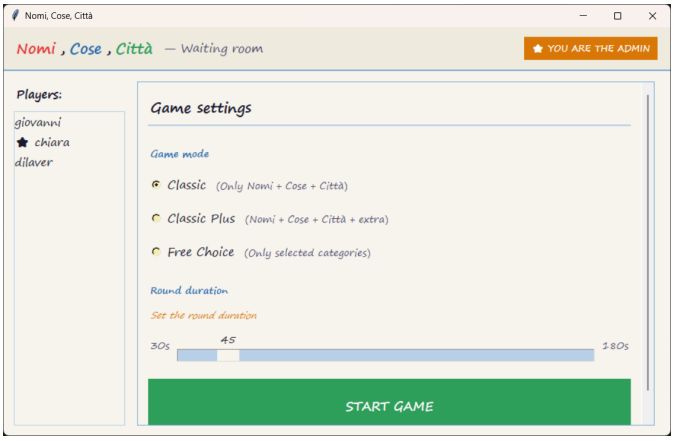

Figure 10: The Lobby Screen.

5. **Gameplay:** the server broadcasts a random letter. Players must quickly type words matching the categories starting with that letter. The UI provides a countdown timer.

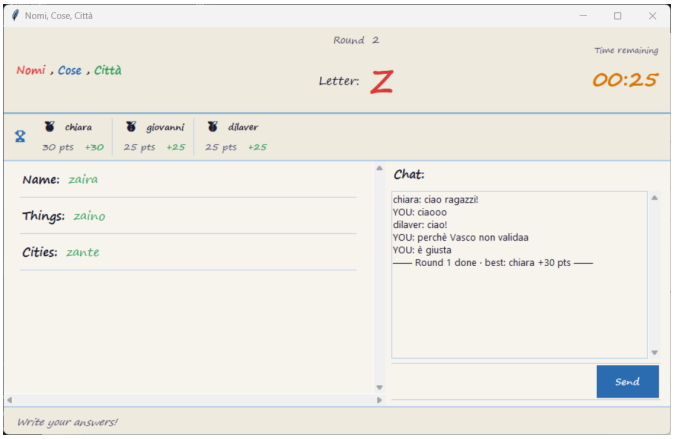

Figure 11: The Game Screen.

6. **Voting & Chat:** when the timer expires, the P2P phase begins. The UI displays all answers submitted by peers. Players interact by clicking "Valid" or "Invalid" on others' words while discussing them via the integrated live chat.

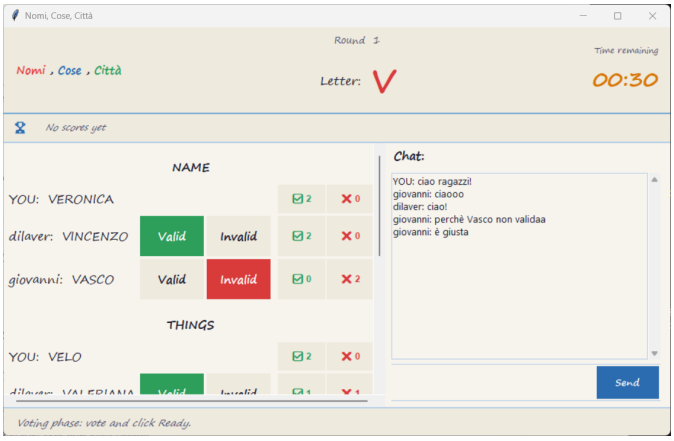

Figure 12: The Voting Phase and P2P Chat interface.

7. **Results:** the server aggregates the P2P votes, calculates the points according to the game rules, and updates the leaderboard. The cycle repeats until a player reaches the target score.

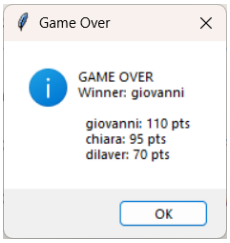

Figure 13: The Game Over Screen.

8 Self-evaluation

8.1 Chiara Giangiulli

My primary focus was on the protocol design, peer-to-peer architecture and server replication. I defined the core message structures and implemented the P2P communication layer used during the decentralized voting phase. Furthermore, I developed the primary-backup logic, based on the heartbeat mechanism, to ensure server high availability, and a UDP broadcast system for dynamic server discovery. I also contributed to the project setup, GUI refinements, and the automated testing of the networking components.

A major strength of my contribution is the decentralized voting phase, which significantly reduces the central server's bandwidth and CPU load. Additionally, the structured JSON message protocol ensures a maintainable and extensible communication layer. Another key strength is the automated failover mechanism, which guarantees high availability and seamless session recovery, masking server crashes from the end users. Furthermore, the UDP discovery mechanism was specifically implemented to allow the game to be easily hosted and played from any PC on the local network without hardcoding IPs in the codebase. To overcome the natural boundaries of UDP broadcasts, this is paired with a graceful fallback to a configuration file: if the LAN discovery times out, clients can seamlessly connect over an extended network using a static public IP. This results in a highly decoupled architecture that adapts to both dynamic LANs and remote environments.

However, a structural weakness lies in the replication mechanism: relying on a shared file system for the heartbeat introduces a single point of failure. In the context of the CAP theorem, this architecture prioritizes Consistency and Availability, lacking Partition Tolerance. If a network partition occurs or the shared storage experiences high latency, multiple servers could attempt to assume the primary role simultaneously, ultimately compromising data consistency.

8.2 Giovanni Pisoni

My primary focus was on the server-side networking infrastructure and the client-side connection management. I designed and implemented the core TCP server (`GameServer`), including the `asyncio`-based connection loop, the management of concurrent client con-

nections via `ClientHandler` instances, and the maintenance of the active client list. On the client side, I developed the `NetworkHandler` class, responsible for establishing and maintaining the TCP connection to the server and running the asynchronous receive loop without blocking the Tkinter UI thread. I also implemented the main game screen, including the category input fields and the local countdown timer. Finally, I developed the end-of-round consensus pipeline — the `VotingAggregator`, the `ScoringEngine`, and the score broadcasting logic — as well as the `ReconnectionManager`, which handles unexpected disconnections via a configurable circular retry policy across multiple fallback servers.

A major strength of my contribution is the reconnection system, which handles the full client-side failover scenario end-to-end, combining circular server rotation with graceful degradation to static IP configuration. The component is extensively tested, covering edge cases such as malformed configuration files, rotation exhaustion, and concurrent reconnection guards. The clean separation between the network layer and the application logic also made each component easy to test in isolation and straightforward to reuse across reconnection attempts. Additionally, the benefit-of-doubt policy in the voting aggregation ensures fairness when network issues prevent votes from being collected in full, prioritising the player experience over strict consistency.

However, the server-side message reading relies on a fixed-size buffer rather than a delimiter-based approach, which could lead to issues with large or fragmented messages and represents an area for improvement. Furthermore, the networking component on the client side carries two distinct responsibilities — server communication and peer-to-peer communication — which reduces clarity and would benefit from a cleaner separation in a future iteration. Finally, the reconnection retry delay is applied uniformly between all attempts; an exponential backoff strategy would be more appropriate to avoid flooding a recovering server with rapid reconnection requests.

8.3 Dilaver Shtini

In this project, I primarily focused on the client-side architecture, the core game logic using a finite state machine (FSM), and state persistence mechanisms. I partially developed the graphical user interface (GUI), integrating the lobby, chat, and real-time voting system, connecting visual events to the underlying network layer. On the server side, I designed the FSM to coordinate round phases, centralized timers, and input validation. Additionally, I implemented continuous state serialization on the shared file system and the session recovery algorithm that seamlessly reconnects disconnected clients to their previous state.

A highlight of my contribution is the session recovery system, which integrates seamlessly with the server's replication architecture. By restoring the state of the FSM, timers, and client UI, the system ensures an uninterrupted user experience during server failovers or temporary network outages. Furthermore, the FSM implementation ensures deterministic and synchronized game transitions across all distributed nodes, effectively preventing race conditions during critical phases such as reply submission and scoring.

However, a structural weakness lies in the state persistence strategy. Relying on

synchronous I/O operations to serialize the entire game state into a JSON file introduces a significant performance bottleneck. While this approach effectively supports active-passive failover for a single lobby, it severely limits the system's horizontal scalability and processing speed. If the server were to handle a large number of concurrent game sessions, this file-based approach would cause severe I/O blockages, requiring a dedicated database solution.

9 Future works

While the current implementation fully satisfies the project requirements, the architecture leaves room for several enhancements:

1. **Advanced leader election**: to solve the weakness of the shared-storage replication, a future improvement would involve replacing the Active-Passive heartbeat file with a proper consensus algorithm, such as Raft or Paxos. This would allow running an odd number of independent server nodes that actively vote for a leader, entirely removing the single point of failure represented by the shared file system.

2. **Automated word validation**: currently, the system relies entirely on the P2P phase for word validation. An unimplemented feature we considered was integrating a Dictionary API. Before broadcasting words for peer voting, the server could asynchronously query an external API to pre-validate standard words, flagging obvious errors and reducing the manual voting burden on players. We excluded this to avoid dependency on external internet services and to maintain the purely localized nature of the game.

3. **Security and encryption**: the system currently transmits JSON payloads in plaintext. For a production deployment over the public Internet, upgrading the raw TCP sockets to TLS/SSL encrypted streams is mandatory to prevent packet sniffing and ensure that malicious actors cannot inject false votes or manipulate the `CMD_SUBMIT` payloads.

4. **Database integration**: replacing the `StateManager` file-based approach with a dedicated database would significantly increase the system's throughput, enabling the server to handle multiple independent game lobbies concurrently without I/O blocking issues.