

Final Report Template

DS-Monopoly

Final Report for the Distributed Systems Course [2025/2026]

Daniele Loverre

`daniele.loverre@studio.unibo.it`

September 2, 2026

DS-Monopoly is a distributed clone of the Monopoly board game. It is a simplified version of the game where a centralized server acts as the *bank*, it coordinates the game and keeps the game state, validates actions, prevents cheating and broadcasts the updates to all the connected clients. The clients are *players* that can join a game by connecting to the central server (2 to 4 players per game). Players can buy properties or houses, roll dice and see the board and the players information from a GUI. Disconnections from both sides, clients and server, are handled in order to keep the game running. Features like checkpointing, authentication and timeouts, are used to enhance fault tolerance and ensure more reliability and availability.

Disclaimer

During the preparation of this work, the author used Gemini to assist in coding the graphical user interface with PyGame for the clients, the unit tests and to accelerate the development of some small functions and code refactoring. It was not used to design the project's architecture.

After using this tool/service, the author reviewed and edited the content as needed and takes full responsibility for the content of the final report/artifact.

1 Concept

The project is a desktop application with a GUI for clients. Players can see the board with houses built and pawns on it on the left, and the information of the players, such as their balance and their properties, on the right. The last event of the game is also written on the GUI in order to let players understand what happens during their and others player's turn. The server can be started on CLI or using docker compose, all the information and logs of the server can be read on CLI.

Players cannot play on the same device (more precisely on the same window). They connect to a server, providing the server's port and ip, provided on the server's CLI at the start, and an available nickname. Clients send messages every event they generate, created on button's pressing. The server broadcasts to all his connected clients the game state on every update on it. They exchange messages in JSON sent via TCP socket. The only data stored are the checkpoint of the game state saved in a file on the server side and the tokens saved in files for the authentication, one for each nickname used, on the client side.

When the server is started the game waits for clients to connect. Once all the players in the *lobby* are ready (at least 2 players) the game starts, 5000 coins and 5 random properties are assigned to each player. Following their turn, players can make actions that are sent to the server and updates the game state. The updated state is sent back to all the players. The game continues until only one player remains without going in *bankruptcy* or until all the players leave the game.

Distribution is needed in this project to allow geographical distribution between nodes, so that is not necessary that players are in the same physical room, but the game state is shared across the network. The state is kept only on the server that acts as a single source of truth, avoiding cheating from clients. Furthermore, the failure of a client does not interrupt the game, but it can be continued by the other players connected.

2 Requirements Elicitation and Analysis

2.1 Functional requirements

- **Server Initialization:** The server should expose and print its endpoint (IP and port) upon startup.
 - *Acceptance Criterion:* Running the server application outputs its endpoint on the CLI.
- **Client Connection:** Clients should be able to connect to the server using the provided endpoint and an available nickname, opening a GUI lobby.
 - *Acceptance Criterion:* A user entering a valid IP, port, and unique nickname successfully joins the lobby and sees the GUI.
- **Game Start Condition:** The game should start only when at least two players are in the lobby and all connected players pressed *"ready"*.
 - *Acceptance Criterion:* The game goes from lobby to playing state immediately after the last player clicks the ready button.
- **Turn Management:** The system should have a turn-based logic, each player can only do the actions they are allowed to do during their specific turn.
 - *Acceptance Criterion:* A player cannot click *"roll dice"* or *"buy"* if it is not their turn or if the action is invalid for their current state.

- **Game State Visualization:** The GUI should display the game board, player information (balance, properties), and the last game event.
 - *Acceptance Criterion:* Whenever an action is performed, the GUI immediately updates the board, the balances, and the event log for all connected players.
- **Client Disconnection Handling:** If a client disconnects, the game should pause and grant the disconnected player 30 seconds to rejoin.
 - *Acceptance Criterion:* A disconnection triggers a 30-second timer; if the player reconnects in time, the game resume. Otherwise, the player goes bankrupt.
- **Server Disconnection Handling:** If the server fails, clients should wait up to 90 seconds for it to restart before shutting down.
 - *Acceptance Criterion:* Shutting down the server triggers a 90-second waiting state on all active clients.
- **Winning Condition:** The game should end when only one player remains active and not in bankruptcy.
 - *Acceptance Criterion:* When all the players, except one, declare bankruptcy or disconnect permanently, the system declares the remaining player the winner.

2.2 Non-functional requirements

- **Consistency:** The system should ensure state consistency across all connected nodes at any given time.
 - *Acceptance Criterion:* The players GUI's will always show the exact same data.
- **Security:** The system should prevent clients from doing invalid game actions.
 - *Acceptance Criterion:* Any altered or malicious payload sent by a client is rejected by the server without affecting the game state.
- **Availability and Fault Tolerance:** The system should be able to recover its exact previous state following a critical node failure.
 - *Acceptance Criterion:* Restarting the server after a simulated crash allows clients to reconnect and find the board, balances, and turn order exactly as they were before the crash.

2.3 Implementation requirements

- **Programming Language:** The system is developed using Python.

- **Networking Protocol:** The communication between client and server is implemented using TCP sockets to ensure reliable delivery.
- **Data Format:** Nodes exchange information using the JSON format.
- **GUI Library:** The client's graphical interface is built using the PyGame library.
- **Containerization:** The server can be started using Docker Compose to facilitate automatic restarts and environment consistency.

2.4 Relevant Distributed System Features

- **Transparency:** The clients only need a single network endpoint (IP and port) to interact with the service, the internal server hosting details are hidden. However, full transparency (such as failure or replication transparency) is not a primary goal; failures and reconnection states are explicitly made visible to the players via the GUI to manage expectations during network drops.
- **Fault Tolerance and Dependability:** If a client fails, recovery systems try to reestablish all the connection and resume the game. The server needs to be active for the entire game, but, if a failure occurs, it should be able to reload the game state from the last saved checkpoint. The checkpoint must be available and not corrupted in order to be used in a server reconnection. The reconnections timeouts are 30 seconds for the clients and 90 seconds for the server. When a client fails, the reconnection timeout should restart also for the other disconnected players.
- **Scalability:** At the moment a server can handle one single game concurrently. The system could need to handle an increasing number of users when more lobbies can be created on a single server. The amount of data handled should be more or less the same even if the complexity of the game increases as the actual real game is. Each game is independent, so at the end of the game everything is reset.
- **Security and Trust:** The only data stored are the checkpoint on server's side and the token file for authentication on client's side. Cheating is prevented by having a centralized server, clients can only read the serialized game state and cannot communicate to other clients. Every player has the same permission. The authentication is needed for guarantee that a player that reconnects to a game after a failure is the exact same player that was playing before the failure occurred.
- **Resource Sharing:** The *game state* is the resource kept by the central server. The server coordinates and synchronizes the process, it handles the incoming events from clients, updates the game state and broadcasts it to the connected peers. The clients cannot access the *game state* directly, they cannot read it or modify it directly, but they can only generate events that the server elaborates in order to modify and send the *game state*.

- **Communication Efficiency and Bandwidth:** The game is turn-based, there is no need to have an instant update from the server. All the data sent on the network has almost the same size, from the server is the game state and from the clients the events they generate.

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis.

3.1 Architecture

The system adopts a **Client-Server** architecture. The central server acts as the manager of the game logic and the single source of truth for the shared game state. On the other hand the clients responsibilities are rendering the Graphical User Interface (GUI), capturing player inputs, and forwarding the events to the server.

This choice was made to ensure different requirements and features identified in Section 2 and Section 2.4:

- **Security and Anti-Cheating:** By establishing an authoritative server, clients cannot manipulate the game. The server strictly validates every requested action against the game rules before applying it and broadcasting the result.
- **Consistency:** In a turn-based board game, all players must observe the exact same state simultaneously. A centralized server resolves concurrency issues, processing incoming events sequentially and ensuring that no state divergence occurs among the connected nodes.
- **Network and Algorithmic Simplicity:** A P2P architecture would have required a fully connected mesh network and complex distributed consensus algorithms just to agree on simple game events like dice roll or a property purchase. The Client-Server model drastically reduces network overhead and algorithmic complexity.

The system's design is a mix of a **3-Tier Architecture** with an **Event-Based Architecture**.

The system is logically and physically divided into three distinct tiers:

- **Client (Presentation Tier):** Responsible exclusively for the user interface (PyGame).
- **Server (Application/Logic Tier):** It contains the game logic and acts as the authoritative coordinator.
- **Filesystem (Data Tier):** The components interact directly with the local file system to store information.

The communication between the components is inherently **event-driven**. The Server does not run a continuous computation loop, instead it waits passively for network events or system events. Once an event is received, it triggers a state mutation on the Application Tier, which generates a reactive broadcast event (the updated JSON state) sent back to the Presentation Tier.

3.2 Infrastructure

The system architecture follows a classic Client-Server model. The infrastructure is designed to be lightweight, avoiding external dependencies such as dedicated databases, message brokers, or load balancers.

3.2.1 Infrastructural Components

The system has the following infrastructural components:

- **Central Server (1 instance):** It acts as the game coordinator and the single source of truth. It manages connections, holds the authoritative game state in-memory, validates actions, and broadcasts updates to the connected peers.
- **Game Clients (2 to 4 instances):** These represent the end-users. They are responsible for rendering the graphical user interface (GUI) and forwarding user inputs to the server. Clients do not communicate with each other directly, all traffic passes through the server.
- **Storage (Local File System):** Instead of a traditional database, the server uses the host's local file system to store checkpoints (`autosave.json`) and the clients use it to store authentication tokens. This ensures fault tolerance and session recovery without the overhead of maintaining a database server.

3.2.2 Network Distribution

The components are designed to be distributed over the network. The server and clients typically run on different machines. The server binds to the machine's local IP, and clients connect over the network using standard TCP sockets.

3.2.3 Component Discovery

The system relies on explicit endpoint provisioning:

1. Upon startup, the Server inspects its network interfaces and logs its binding endpoints (IP addresses and port) to the terminal.
2. The end-users explicitly provide the target IP and port to their Client application upon launch.
3. Once connected, the server assigns a unique authentication token to the client, guaranteeing safe reconnection to the session even if the IP changes dynamically.

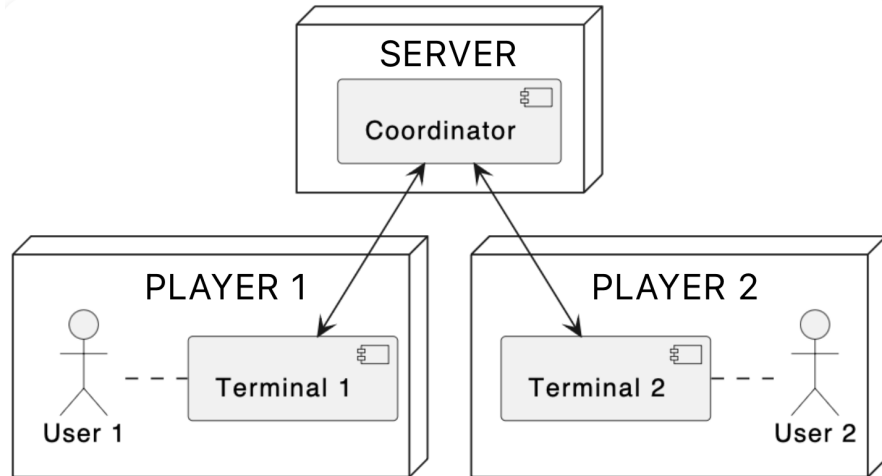


Figure 1: Component diagram of the Monopoly system with two players.

3.3 Modelling

Domain Entities

The primary domain entities are:

- **Monopoly:** The root entity representing the game session, managing the lifecycle (lobby, playing, paused, closed), turns, and the list of players.
- **Player:** Represents a user in the game. It holds personal state such as balance, board position, owned properties, and bankruptcy status.
- **Board & Space:** The physical representation of the game. The Board is an aggregate of multiple Spaces, which are divided into **PropertySpace** (can be bought, owned, and upgraded with houses) and **ActionSpace** (trigger special events like Taxes, Jail, or Chance cards).

Mapping to Infrastructural Components

- The true, mutable **state of the domain entities** resides entirely in the memory of the central server (**MonopolyBank**).
- The server periodically save the state to the local file system via the Checkpointing module.
- The clients (**MonopolyUser**) only receive a serialized representation (a JSON dictionary) of the state. Clients render the representation visually and collect user inputs.

Domain Events

In this architecture the domain events are modeled by the `PlayerEvent` enumeration:

- **Lifecycle Events:** `JOIN`, `READY`, `DISCONNECTION`, `RECONNECTION_TIMEOUT`, `QUIT`.
- **Gameplay Events:** `ROLL` (moves the player and evaluates the landing space), `BUY` (purchases a property), `BUILD` (adds a house), `END_TURN` (shifts the turn to the next player).

Exchanged Messages

Communication between components relies on asynchronous JSON messages over TCP, categorized into:

- **Events (Client → Server):** Sent by the user. They contain the event type and a payload (e.g., `{"type": "roll", "payload": {"nickname": "Alice"}}`).
- **State Updates (Server → Client):** The server responds to valid commands by broadcasting the entire updated, serialized game state to all connected peers.

System State Comprehension

The global state of the system encapsulates all the information required to resume or render the game at any exact moment. It comprehends:

- **Game Data:** Current status (`LOBBY`, `PLAYING`, `PAUSED`, `CLOSED`), active turn index, `has_rolled` flags, and a list of allowed actions for the current UI generation.
- **Players' Data:** Nicknames, board positions, balances, and boolean flags for bankruptcy.
- **Board Data:** The structural layout of the spaces, including dynamic data like the `owner` of each property and the number of `houses` built on them.
- **Network State:** Sets of ready players and sets of temporarily disconnected players waiting for reconnection.

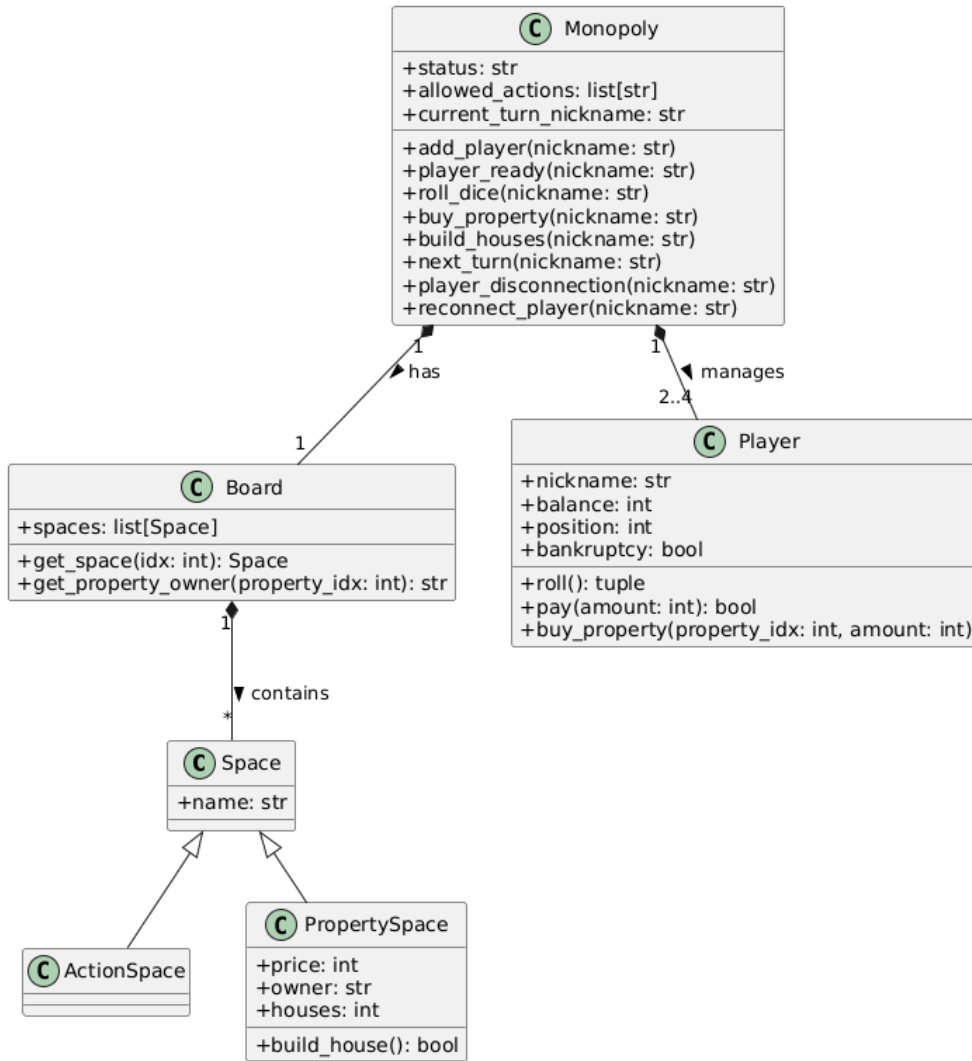


Figure 2: Class diagram of the model's main components.

3.4 Interaction

The communication between components is asynchronous and relies on exchanging serialized JSON strings over TCP sockets. The interaction architecture avoids peer-to-peer communication: all clients communicate exclusively with the central server.

Interaction Patterns

The clients never calculate game logic (e.g., they do not compute dice rolls or property costs). They only send user events and wait. The Server is the sole authority that executes the logic and determines the outcome. Instead of clients polling the server for updates, the server proactively broadcasts the entire serialized game state to all connected peers every time a mutation occurs. Upon receiving a new state, the client's GUI inspects the `current_turn_nickname` and `allowed_actions` fields to dynamically determine whether to enable the input buttons for the local user.

Communication Flow

Communication is triggered by user inputs (e.g., clicking a button) or connection lifecycle events (e.g., socket closed). Clients send lightweight JSON payloads containing an action type (e.g., `join`, `ready`, `roll`) and a payload specifying the player's nickname and token. The server responds by broadcasting the full JSON representation of the entire Monopoly object, ensuring all clients are perfectly synchronized.

Sequence Diagram

Figure 3 illustrates the initialization phase (Lobby) and the game loop. It highlights how a single client action triggers server-side validation, a local storage checkpoint, and a global state broadcast to all peers.

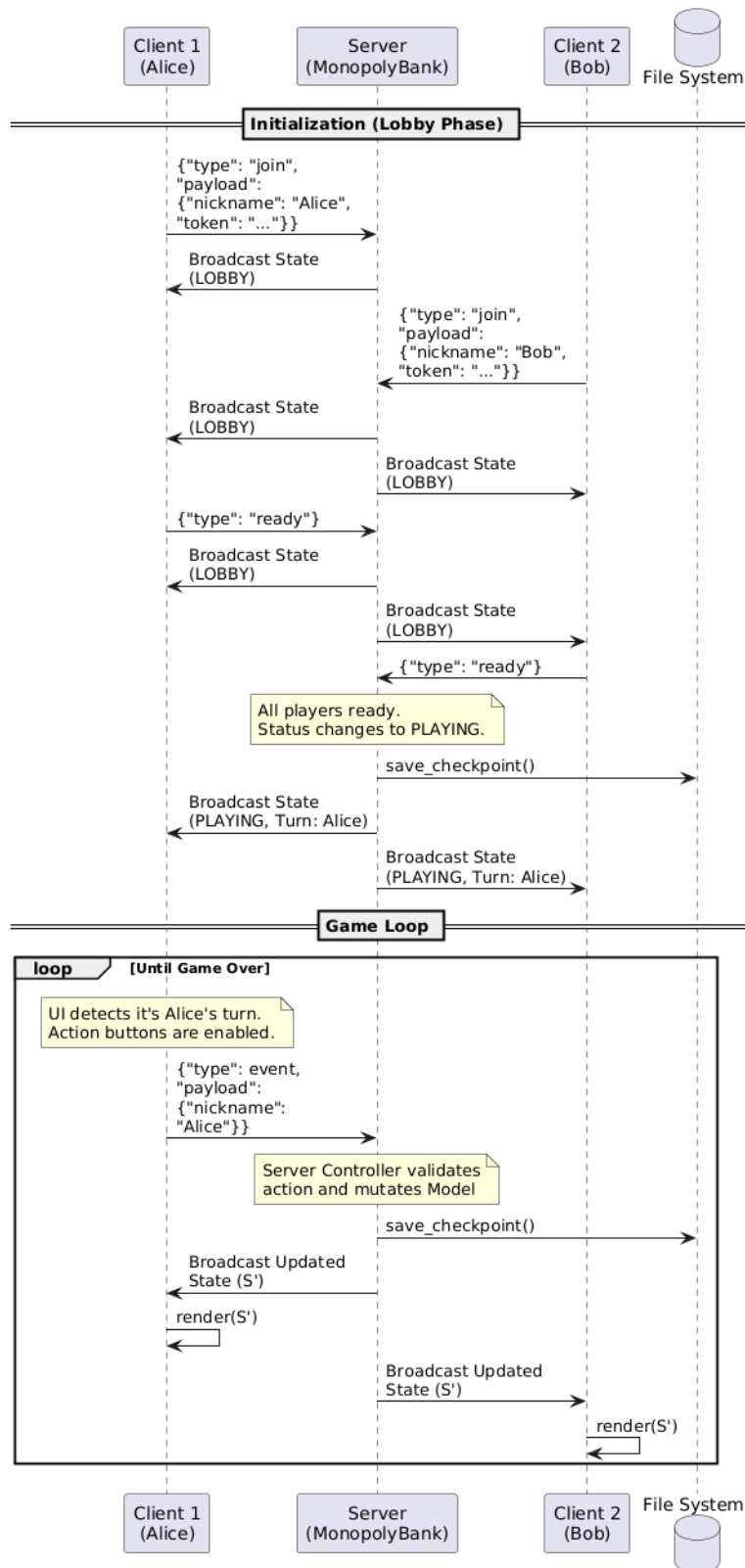


Figure 3: Sequence diagram showing the Lobby initialization and the main Game Loop interaction.

3.5 Behaviour

The behavioral flow of the system revolves around the interaction between the Model, the View, and their dedicated Controllers. Figure 4 illustrates the complete lifecycle of a single user action, from the UI input to the global state broadcast.

Individual Component Behavior

The components behave differently based on their responsibilities:

- **Monopoly (Model) – Stateful:** It is the core of the system. It maintains the current state of the game in memory. It mutates its internal variables only when invoked by the Server Controller.
- **MonopolyView (View) – Stateful:** It holds the visual state of the board but knows nothing about the game logic. It runs an infinite loop waiting for user inputs.
- **Controllers – Stateless:** Both the **ServerController** and **ClientController** act as pure functions or translators: the Client Controller maps player inputs into events, while the Server Controller maps events into Model method invocations.
- **Serializer – Stateless:** It simply transforms objects into JSON strings and vice versa, without holding any memory of past transformations.

Updating the State of the System

To guarantee strong consistency and prevent cheating, the system enforces a strict rule regarding state updates:

- Only the **Server** (specifically the **Monopoly Model**, orchestrated by the **ServerController**) is in charge of updating the state of the game.
- The state is updated only when the Server receives a valid event message from a client (e.g., a **ROLL** or **BUY** event) or when a system timeout is triggered (e.g., a disconnected player is declared bankrupt).
- When a client terminal (**MonopolyUser**) captures an input, its controller generates an event dictionary. This event is serialized and sent to the server. The **ServerController** deserializes the event, manages the input, and asks the Model to check its correctness (e.g., checking if it is actually that player's turn). If valid, the Model computes the action and updates its internal variables. Finally, the Server Controller serializes the new state and sends it back to the clients, where the local controllers update their respective Views.

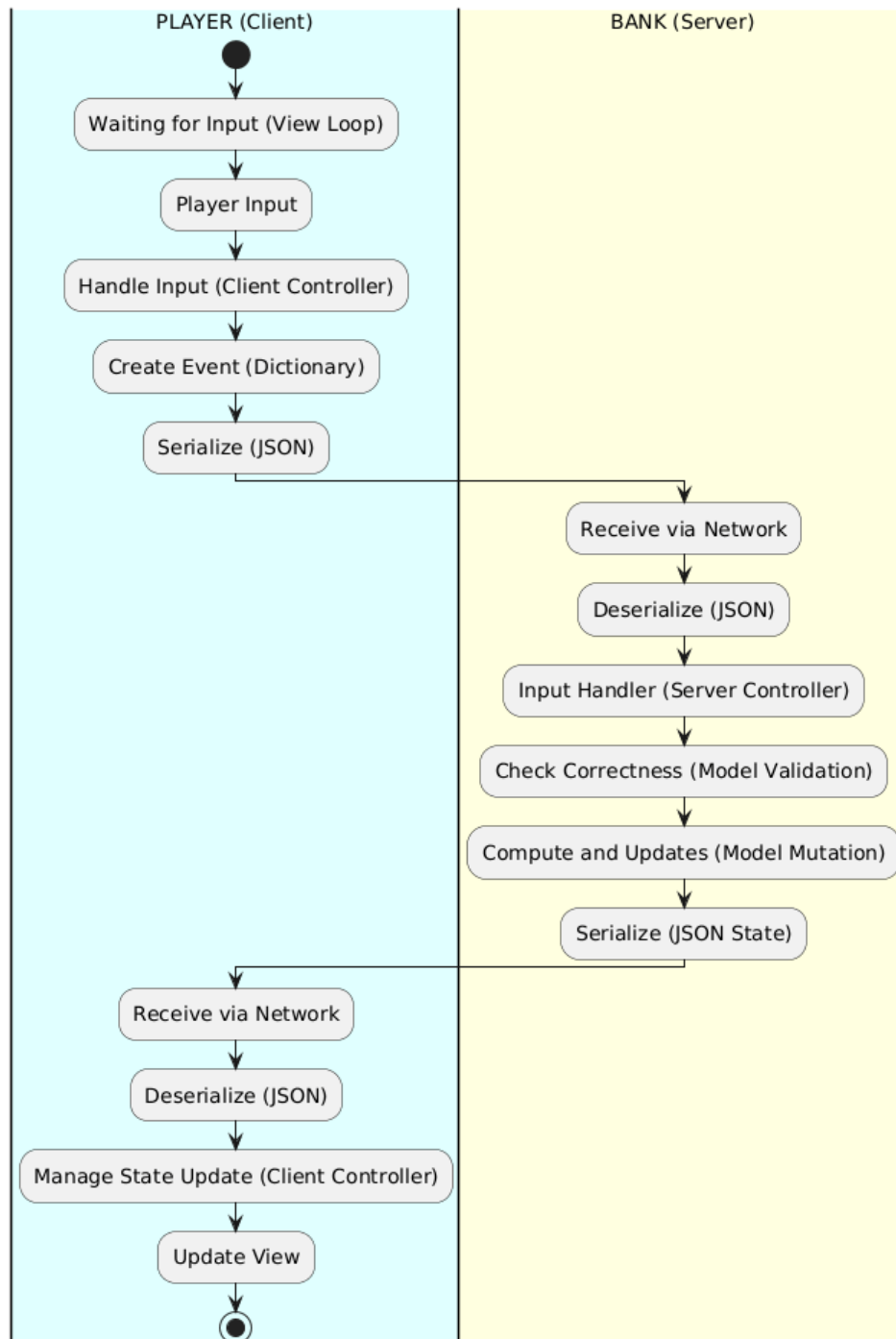


Figure 4: Activity diagram based on the MVC pattern, showing the flow between the Player (Client) and the Bank (Server).

3.6 Data and Consistency Issues

The only data that needs to be stored are the checkpoint of the model, that is the serialized JSON format of the actual state of the game including players token, and the players token.

The checkpoint is a file (`autosave.json`) where all the useful information needed to restore the game state in case of a server fail are saved. It comprehends the serialized state of the game (players, balances, board ownerships, current turn) along with the dictionary mapping player nicknames to their authentication tokens and the players token. When the server starts it checks if a valid checkpoint is available and loads it. During the game, on each event the server updates a checkpoint.

The players token are stored during the joining phase of a player (`nickname_token.txt`). It contains a unique UUID generated upon the first connection that is used after a reconnection from a player by checking if the token provided by the user is the same of the token saved on the server.

Both files are stored in the local filesystem, the checkpoint is stored on the server machine, while the tokens file are stored on the clients machines.

There is no concurrency in reading and writing these data, each file is managed by the owner's machine clients or server. There is a single copy of them, there isn't a backup, and they are not shared between nodes. By relying on single local files without backups or distributed replication, the system achieves **Strong Consistency** (there is a single source of truth). The cost of this design is reduced *Availability* and *Fault Tolerance*: the files act as a Single Point of Failure. If the server's disk fails or the file gets corrupted, the game cannot be restored.

Data must be consistent across the network to ensure the game works properly:

- **Game State Sharing:** The server broadcasts the updated JSON state to all clients after every action so they can render the view synchronously.
- **Token Consistency:** The token provided by the client during a reconnection must strictly match the token saved in the server's checkpoint. If they are inconsistent, access is denied.

3.7 Fault-Tolerance

The system is designed to handle the most common issues in multiplayer games: network drops, client disconnections, and unexpected server crashes.

3.7.1 Data replication

The only data replicated are the players token that are stored both on server and client, but it is needed only for the authorization of a player to join a started game by checking if they are the same, it cannot be used to restore a lost or corrupted data. Tokens must be consistent in order to guarantee a right form of authentication during a client reconnection, otherwise if one of the file is corrupted the authentication can fail and an allowed user cannot rejoin the game.

3.7.2 Timeouts and Retry Mechanisms

The system implements timeout and retry loops to handle temporary network failures, preventing immediate game disruption:

- **Server:** once the server lose connection a 90-second timeout starts. During this period of time if the connection is restored, the game restarts loading the last checkpoint saved. The game is set to "CLOSED" once the timeout ends.
- **Client:** once the client lose connection a 30-second timeout starts. During this period of time if the connection is restored, the game restarts. The player is set to "bankruptcy" once the timeout ends.

3.7.3 Network Partitions and Heartbeat Mechanism

On network drops, the system implements an application-level **Heartbeat** mechanism coupled with a 5-second socket timeout on the `recv()` function. Active clients send a lightweight "ping" message every 2 seconds in the background. The server intercepts these messages and immediately replies with a "pong", without interfering with the game logic. If a silent drop occurs, the 5-second timeout triggers a socket exception, immediately starting the standard disconnection protocol.

3.7.4 Error handling

The system uses TCP sockets for connections, so if a message fails to reach the peer, the exception is caught and the retry mechanisms starts. If an error occurs during the joining phase the connection gets refused and a log is displayed on client's and server's CLI. Errors that occurs in game are shown in the last event box on the GUI.

3.8 Availability

In the context of the CAP theorem, when a **network partition** occurs, the system favors *Consistency* over continuous *Availability*, pausing the game to prevent desynchronization. The behavior depends on which side of the network is isolated:

- **Server isolation:** if the server lose the connection with the clients, each of them waits 90 seconds until considering the game closed. If the connection is reestablished, then the game restarts for all the client.
- **Client isolation:** if a client lose the connection with the server, the server stops the game for 30 seconds and waits to reestablish the connection. At the end of the 30 seconds the disconnected player is set as in bankruptcy and the game continues with the other alive players.

Having a single server that orchestrate the game and the connections yields to have a single point of failure. This is handled by a set of recovery mechanism as the mechanism seen in section 3.7.

3.9 Security

The system implements a token-based authentication, introduced for handle players reconnections, avoiding that a different user could take the place of a disconnected player only by using the same nickname.

During the joining phase of a client, a UUID token is created and stored in the local filesystem. The token is then sent to the server along with his nickname. After a disconnection, while reconnecting, the clients checks if there is a saved token and sends it with the nickname to the server.

The server stores a pair of token-nickname. Once the disconnected player sends another JOIN message, the server checks if the token received is the same of the one stored during the first JOIN.

There isn't a form of authorization because every user has the same access rights, and a form of encryption, tokens and checkpoint are stored in clear.

4 Implementation

This section details the technology-dependent choices adopted to implement the architectural design.

Network Protocols

The system's communication is built on top of **TCP sockets**. TCP provides a persistent, reliable, and ordered stream of data. The server binds a dedicated socket for each connected client and maintains the connection open for the duration of the session. This is ideal for a multiplayer board game where continuous state synchronization is required and packet loss (which could occur with UDP) must be avoided.

In-Transit Data Representation

Messages exchanged between the server and clients are encoded in **JSON** format. JSON was chosen because it's easy to parse and generate Python object. A custom serializer and deserializer were implemented in order to parse the objects in a JSON format and reconstruct them from JSON to Python objects.

Database and Querying

The system does not have a traditional relational (SQL) or NoSQL database. Instead, it relies on the local filesystem. The checkpoint is saved directly as a JSON file (`autosave.json`). Consequently, there is no formal query language involved, the system performs direct I/O reads and writes to dump or load the entire game state and players token.

Authentication

The system implements a lightweight custom **Token-based Authentication** for session recovery. Upon the first connection, a standard UUID is generated. This token is saved in clear text on the client's machine and mapped alongside the checkpoint dictionary on the server. Reconnecting clients must provide this exact UUID to prove their identity and reclaim their seat.

Authorization

All users have the exact same role ("Player"). The only type of authorization implemented in the system is made using the nicknames that the server uses to check in the internal game state before authorizing any state-mutating command, ensuring that players can only interact when it is legally their turn.

4.1 Technological details

The system is entirely developed in Python 3. The framework used for the client's View is PyGame that provides the rendering engine for drawing the board, managing the game loop, and capturing hardware events (such as mouse clicks for the action buttons).

5 Validation

5.1 Automatic Testing

The core of the model has been tested automatically, along with serialization and deserialization.

Individual components that do not rely on active network sockets or GUI rendering were unit-tested using the `pytest` framework. The primary targets were:

- **Model:** main entities were tested on different scenarios for every game phase (LOBBY, PLAYING, CLOSED, PAUSED), with different number of players, and on player disconnection and reconnection.
- **Data Serialization:** `Serializer` and `Deserializer` were tested on each entity, on the full game state and on unsupported data.

5.2 Acceptance test

To validate the overall user experience, the system has been tested through an extensive manual acceptance testing.

The system has been tested on the communication, interaction, and integration among all components across a Local Area Network (LAN). The setup involved deploying the `MonopolyBank` server on one physical machine and connecting multiple `MonopolyUser` clients from different or same hosts. The standard game loop (joining, rolling, buying, ending turns) was tested focusing also on corner cases and fault-tolerance mechanisms:

- *Network Partitions:* Unexpected client disconnections (e.g., physically disabling a machine's Wi-Fi adapter) were simulated to verify the 30-second timeout and the subsequent bankruptcy logic.
- *Crash Recovery:* Server process were forcefully killed to ensure the `autosave.json` checkpoint was correctly generated and capable of restoring the paused game state upon server reboot.
- *Network Partitions:* Unexpected client and server disconnections were simulated. The system successfully detected the network partition within 5 seconds thanks to the Heartbeat mechanism, triggering the dynamic 30-second or 90-second timeouts and the subsequent recovery or bankruptcy logic.
- *Security Authentication:* Reconnections were tested verifying that dropped players could successfully rejoin using their local UUID tokens, and that unauthorized connections were correctly rejected.

These tests have been done manually because the client is tightly coupled with `PyGame`. Verifying that UI elements (tokens, action buttons, properties) are rendered at the correct coordinates and respond to hardware mouse events is particularly complex. Furthermore, simulating TCP connections, physical network drops, socket timeouts across multiple nodes would have added more complexity if tested automatically.

6 Deployment

To prepare the environment to run the system locally, there are three main steps: cloning the repository, setting up the server (via Docker or locally), and starting the client terminals.

Prerequisites and Software Versions

To run the project, the following software must be installed on the host machines:

- **Python:** Version 3.10 or higher.
- **Dependencies:** The project requires two external libraries, specified in the `requirements.txt` file: `pygame>=2.0.0` (for the client GUI) and `psutil>=5.0.0` (for server network utilities).
- **Docker & Docker Compose** (Optional for the server deployment).

Containerization (Server)

The `MonopolyBank` server is fully containerized.

- **Dockerfile:** It starts from a lightweight `python:3.10-slim` image. It copies the `requirements.txt` first to leverage Docker's layer caching during the `pip install` phase.

- **Docker Compose:** The `docker-compose.yml` maps the container's exposed port to the host's port (8080:8080). It maps a local volume (`./checkpointing:/app/checkpointing`). This choice ensures that the `autosave.json` checkpoint file persists on the host machine even if the container is stopped or destroyed, guaranteeing fault tolerance.

Step-by-Step Installation and Startup Instructions

1. **Clone the Repository:** Open a terminal and clone the project source code.

```
git clone https://github.com/22Daniele/DS-monopoly-clone
cd DS-monopoly-clone/dmonopoly
```

2. **Start the Server:** You can run the server using Docker Compose:

```
docker-compose up --build
```

Expected Outcome: Docker will build the image and the server will begin listening on port 8080.

Alternatively, to run it locally without Docker:

```
pip install -r requirements.txt
python serverMain.py
```

3. **Start the Clients:** On the players' machines, ensure Python is installed, then install the requirements and launch the client script.

```
pip install -r requirements.txt
python clientMain.py <SERVER_IP> <PORT> <NICKNAME>
```

Expected Outcome: A PyGame graphical window will open, presenting the initial Lobby screen. The CLI will log the successful connection to the server or display a rejection message if the lobby is full.

Important Note on Docker:

If the server is deployed using Docker, the IP address automatically detected and printed in the server's console logs is the Docker's internal virtual network rather than the host machine's actual IP. When connecting clients from different physical machines, users must ignore the internal Docker IP and instead provide the host machine's real IP address as the connection endpoint.

Environment Variables and Configuration

No external `.env` files or configuration files are required. The server automatically creates the `checkpointing` directory, and the clients automatically generate the `tokens` directory upon their first launch. Network configurations (like dynamic IP resolution) are handled internally by the code via the `psutil` library.

7 User Guide

This section is a guide on how to interact with the Monopoly system once the deployment steps (described in Section 6) have been completed.

7.1 LOBBY phase

During this phase, clients (up to 4) can join the game and press the ready button shown on the GUI. Once all the player in the lobby (at least 2) are ready, the game enters in the next phase.



Figure 5: The client GUI during the LOBBY phase.

7.2 PLAYING phase

In this phase the monopoly match starts, every player is initialized with a balance of 5000 coins and 5 random properties. Players should see the board on the left of the window and a dashboard showing players information and last event log on the right. Each player in turn can see and press on their screen buttons to perform actions

- **Your Turn:** When it is your turn, the GUI dynamically enables action buttons (e.g., ROLL, BUY, BUILD, END_TURN). Click these buttons to perform your allowed

moves. The action is processed by the server, and the game information on the GUI updates for everyone.

- **Others' Turn:** When it is not your turn, no actions are allowed. The action buttons are hidden or disabled. You act as a spectator, watching the real-time updates triggered by the active player.



Figure 6: The game interface during the PLAYING phase.

7.3 PAUSED phase

If a client loses the connection mid-game, the system enters the PAUESED phase, stopping immediately the game and starting the recovery mechanism 3.7.2. During this phase all the players can do nothing.

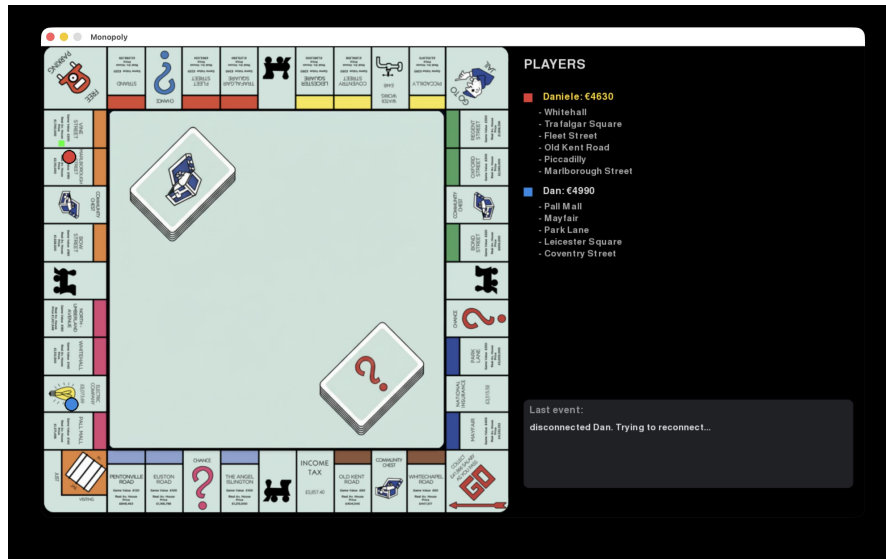


Figure 7: The game interface during the PAUSED phase.

7.4 CLOSED phase

The game ends when only one active player is left (i.e., everyone else is bankrupt or has permanently disconnected) and enters in the CLOSED phase. The winner is implicitly declared, and no further actions can be performed by the users.

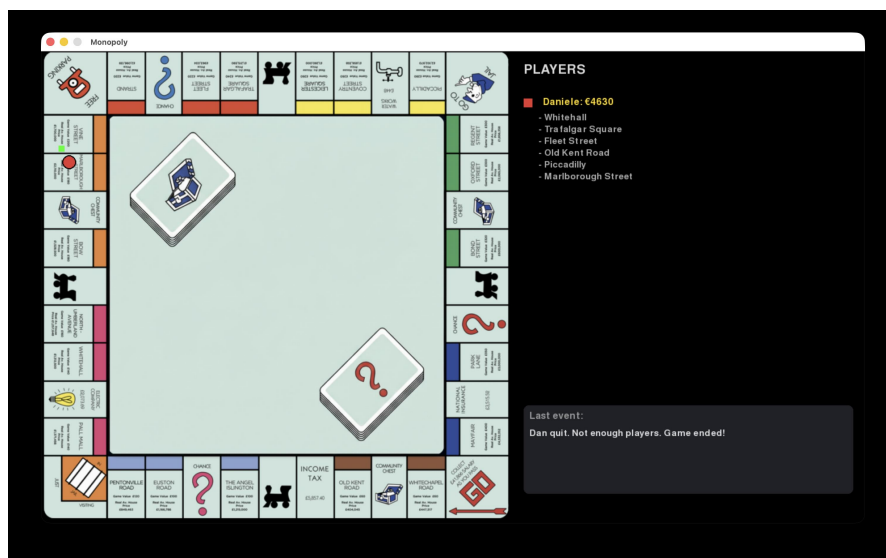


Figure 8: The game interface during the CLOSED phase.

7.5 Shutting Down the System

- **Clients:** Users can gracefully leave the game at any time simply by closing the PyGame window.
- **Server:** Once all clients have disconnected and the game is empty, the Python server process shuts down automatically. However, if the server was started via `docker-compose`, the Docker daemon will automatically restart it, immediately readying the system to host a new game session.

8 Self-evaluation

The system was completely designed and implemented individually by Daniele Loverre.

Strengths

- **Fault Tolerance:** The system handles sudden crashes and network drops effectively. The checkpointing and timeout mechanisms ensure that the game can be easily recovered without any data loss.
- **Decoupled Architecture:** The architecture was structured to be lightweight and reusable, strictly decoupling the network layer from the Model and the View.
- **Anti-Cheating Design:** Having the clients act solely as passive renderers of the View keeps the system safe from cheating, allowing players to only perform legally allowed actions.
- **Network Reliability and Efficiency:** The use of TCP sockets makes the connections more reliable and makes it easier to detect connection drops. Furthermore, the client-server architecture keeps the network lightweight for each client, which only needs a single connection with the server (avoiding the network overload of a peer-to-peer mesh). The messages exchanged are small and fast, allowing the server to instantly broadcast the game state to all connected clients.

Weaknesses of the product

- **Single Point of Failure:** Having the server as the single coordinator of the game introduces a Single Point of Failure. Even though a recovery mechanism is implemented, if the server fails and cannot be restarted to reestablish connections within the 90-second client timeout, the current game session is lost and there is no way to restore the game state.
- **Manual Network Discovery:** When connecting across different physical machines, especially if the server is hosted via Docker, users must manually look up and input the host's IP address, which makes finding the server less user-friendly.

- **Limited Security:** Security is limited as a trade-off for simplicity and performance. Authentication tokens are stored in clear text, and TCP payloads are unencrypted.
- **Static GUI:** The graphical user interface is extremely simple and lacks dynamic responsiveness to different monitor resolutions or window resizing.
- **Automated Test Coverage:** Another weakness is the limited scope of automated testing. Only the core domain logic and serializers are covered by automated unit tests. The integration and network behaviors were tested manually, potentially leaving some rare game configurations or edge cases untested.

9 Future works

Possible future work and improvements could be:

- **Web-Based Client Interface:** Replacing the current PyGame desktop GUI with a web-based frontend. This would improve the user experience by providing responsive design across different devices and removing the need for local installations, simplifying the deployment process.
- **Removing the Single Point of Failure (SPOF):** As discussed in Section 8, the current architecture relies on a single authoritative server. A more robust and highly available setup would involve deploying multiple server nodes behind a Load Balancer, and a distributed ledger so that if one server node crashes, another can seamlessly pick up the ongoing session.
- **Concurrent Game Sessions (Matchmaking):** The current system hosts a single game at a time. An extension would be upgrading the server into a central hub where users can create and join multiple concurrent public or private game rooms (lobbies), each running as an isolated game thread.
- **Account Management System:** Integrating a database to handle proper user accounts. This would allow players to register with a username and password, keeping track of their game history, win rates, and friends lists. Consequently, a more robust authentication and authorization mechanism would replace the current one.
- **Strengthened Security:** Introducing a layer of encryption on exchanged data and saved files on both server and client sides.
- **Adding game feature:** The system implements only few feature of the real board game, a lot of feature can be added to make the game complete.