Distributed Minesweeper

A Fault-Tolerant, SMR-Based Multiplayer Architecture

Final Report for the Distributed Systems Course 2025/2026

Nicolás Maire Bravo
nicolas.mairebravo@studio.unibo.it

Iago López Lamas
iago.lopezlamas@studio.unibo.it

June 13, 2026

We built a cooperative, distributed Minesweeper game. But instead of relying on an external database to keep things in sync, we designed an in-memory Replicated State Machine (SMR) from scratch. Using pure Python TCP sockets, the architecture enforces Strong Consistency across a three-node cluster through a two-phase PREPARE/COMMIT chain replication protocol. If the leader dies, the cluster recovers automatically. We fully containerized the entire stack—even the graphical Tkinter client—proving that complex fault tolerance and deterministic state can work without heavy-weight dependencies.

Development Methodology

The source code for this project was developed following a hybrid methodology. We directly conceptualized and wrote the core implementation, including the distributed architecture, the SMR logic, and the PREPARE/COMMIT chain replication protocol, according to the notions covered in class. However, in compliance with academic guidelines, we explicitly state that we used AI tools as collaborative coding assistants and to help us search errors. We primarily relied on them to refine Python syntax, generate Tkinter GUI boilerplate, structure our `pytest` cluster integration, and debug tricky runtime bottlenecks like TCP fragmentation and Docker X11 forwarding. Applying a strict "human-in-the-loop" approach, we thoroughly reviewed, validated against the CAP theorem, and adapted every block of AI-suggested code. We maintain full awareness of the codebase and take full responsibility for every implemented line in the final artifact.

1 Concept

In this project we develop the classic Minesweeper game, but adapted to a multiplayer context with the possibility of having several games running at the same time.

The game mechanics are intuitive yet logically rigorous: players reveal cells on a grid via left-click. Uncovering a safe cell displays a numeric value indicating the exact number of adjacent mines. Relying on this spatial information, players can flag suspected mine locations via right-click. The session concludes either in a loss (revealing a mine) or a win (successfully flagging all mines and revealing all safe cells). The game ends in two ways: losing (uncovering a cell with a bomb) or winning (flagging all the bombs).

- **Type of product:** A distributed multiplayer game setup. It features a desktop client with a Graphical User Interface (GUI) backed by a strictly ordered cluster of custom Python TCP servers.
- **Use case description:**
  - *What:* A cooperative Minesweeper variant. Everyone in the same room plays on the same board, and they see each other's clicks happen in real-time.
  - *Where:* Users run the client app from their machines (or Docker containers), dialing into our remote server cluster.
  - *Interaction:* Mouse-driven. Left click to dig, right click to drop a flag. Latency must stay barely noticeable.
  - *Storage:* 100% volatile. The game lives in the servers' RAM. We purposely avoided persistent databases to tackle the hard problem of maintaining state across crashing nodes using just network protocols.
  - *Roles:* All users have the same role. There is no admin, host, or privileged player, anyone in a room can reveal or flag cells equally.
- **Why is distribution needed?**

  We decided to build a distributed version of this classic game to turn it from a single-player experience into a full multiplayer one, with the possibility of running multiple games at the same time. On top of that, we wanted to satisfy:
  - *Fault Tolerance:* Hardware breaks. If the node hosting your game catches fire, the backup nodes should seamlessly take over.
  - *Resource Sharing:* We need multiple users accessing and modifying the exact same game state concurrently without creating chaos.

2 Requirements Elicitation and Analysis

- **Functional:**
  - FR1: User can create or join rooms by name.
    Acceptance: `test_multiplayer_rooms.py::test_room_isolation` — verifies that two rooms can run in parallel without interfering.
  - FR2: User identifies with a name to enter the room.
    Acceptance: Manual validation — verifies that it is not possible to enter without a username.
  - FR3: Users can reveal cells and plant flags on a shared grid.
    Acceptance: `test_multiplayer_rooms.py::test_two_clients_same_room` — verifies that reveals or flags propagate to every client connected to the room.
  - FR4: Any state-changing action must replicate to all room members instantly.
    Acceptance: `test_replication.py::test_all_replicas_same_state` — verifies that every replica ends up with the same board and `last_committed_seq` after a commit.
  - FR5: Players can hit restart once a mine detonates or the board is cleared.
    Acceptance: Manual validation — verifies that the restart button generates a new board and resets the state for every user in the room.
  - FR6: Players can see a list with the users connected to the room.
    Acceptance: Manual validation — verifies that the user list is updated when someone joins or leaves.
  - FR7: Clients automatically reconnect to a surviving node after a leader crash.
    Acceptance: `test_failover.py::test_f2_accepts_clients_after_both_crashes` — verifies that clients can keep playing after the leader is killed.
- **Non-functional:**
  - NFR1: Strong Consistency (CP). Split-brain scenarios or divergent boards destroy the game. Everyone sees the same mines, or the operation fails.
    Acceptance: `test_consistency_failures.py::test_no_confirm_if_f2_down` — verifies that with F2 unreachable, the leader rejects the operation instead of confirming a partial replication.
  - NFR2: High Availability as a fallback. The cluster must endure up to two sequential crashes.
    Acceptance: `test_failover.py::test_operation_survives_leader_and_f1_crash` — verifies that a confirmed move survives two sequential crashes.
  - NFR3: Isolation between concurrent rooms.
    Acceptance: `test_multiplayer_rooms.py::test_room_isolation` — verifies that a reveal in room A does not affect room B.
  - NFR4: All replicas must build the exact same board from a shared seed (generated by the leader), so we only replicate moves and not the whole board.

Acceptance: `test_replication.py::test_all_replicas_same_state` — verifies that all replicas show the same board after receiving the seed.

- **Implementation:**
  - IR1: Standard libraries (Python, TCP sockets, and threads), no external frameworks.
    Acceptance: Manual validation — verifies there are no external frameworks in `requirements.txt`.
  - IR2: The entire stack, client included, must be containerized via Docker.
    Acceptance: `docker compose up --build` brings up the leader, followers, and frontend with no extra action required from the user.
  - IR3: Automated integration tests with `pytest`, running the real cluster and crashing nodes to test replication and failover.
    Acceptance: `python -m pytest tests/` runs all 9 tests successfully.

2.1 Relevant Distributed System Features

- **Transparency:** Highly relevant. Failovers shouldn't involve the user. When the leader dies, the client app freezes for about a second, reconnects to a follower, and resumes. The player doesn't have to lift a finger.
- **Fault Tolerance & Dependability:** This is the system's backbone. We need to guarantee no phantom reads or data corruption when network cables get pulled.
- **Concurrency:** Highly relevant. Two players might click the same cell at the exact same millisecond. The cluster has to sequence these events perfectly.
- **Resource Sharing:** Various users access and modify the same field concurrently.
- **Security / Scalability / Economy / Evolvability:** Not our main target. We focused entirely on SMR and fault tolerance rather than load-balancing thousands of active sessions.

3 Design

3.1 Architecture

We decided to use a Replicated State Machine (SMR) with a chain replication topology and a two-phase protocol (PREPARE/COMMIT). The reason behind this architecture is to keep the system simple while having strong consistency. We know it is not the best choice for scalability, but we discuss how to deal with it in section 9. Following the CAP Theorem and the architectural principles discussed in the course lectures[**1**], we chose Consistency over Availability (CP). Playing a multiplayer puzzle game out of sync is pointless. To keep Availability as high as possible despite this, we implemented an automatic cascading failover mechanism.

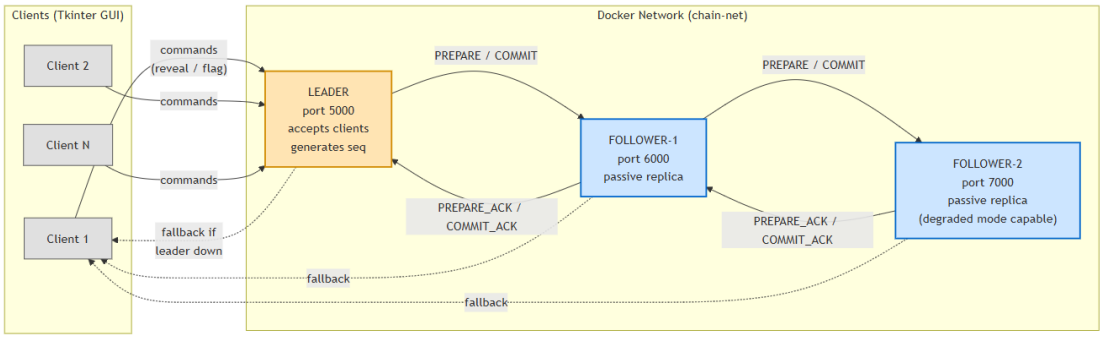


Figure 1: Components diagram

3.2 Infrastructure

The system has only 2 different types of infrastructural components: Frontend Client and Game Servers.

The Frontend Client is a Tkinter desktop application that runs on the player's machine. It handles the GUI, gets mouse clicks, serializes them into commands.

The Game Servers are the backend of the system. They live in three isolated Docker containers forming a replication chain:

- **Leader:** Binds to port 5000. Generates the seq numbers and drives the PREPARE/COMMIT protocol.
- **Follower-1:** Binds to port 6000. Passive replica. Receives upstream state from the Leader and forwards it to Follower-2.
- **Follower-2:** Binds to port 7000. Passive replica and tail of the chain.

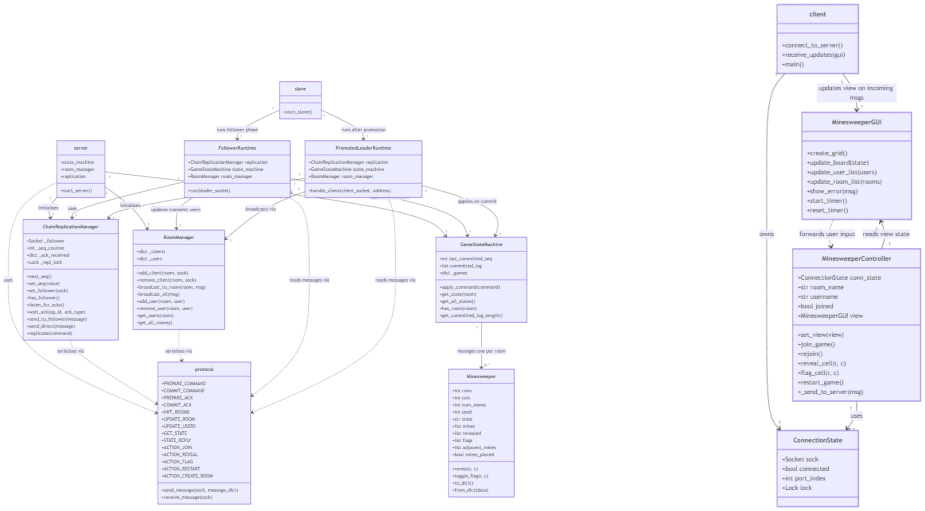
In normal operation, clients only talk to the Leader. If the Leader crashes, Follower-1 becomes the leader and starts accepting clients in its place. If both Leader and Follower-1 are down, Follower-2 takes over and can keep clients operations in degraded mode as the last survival node.

We avoided using extra infrastructure (like caches, message brokers, or external databases) to focus entirely on solving the consensus problem ourselves, holding the state in RAM and using just network sockets.

In our current deployment the three servers run on the same host machine. In a real scenario each server would sit on a different physical machine to avoid hardware failures.

The three servers find each other by name through Docker's internal DNS, which resolves the service names (`leader`, `follower1`, `follower2`) to their container IPs. On the client side, we use a static fallback list with the three endpoints in order: the client tries the Leader first, and if the connection drops or times out, it goes to the next element of the list until it finds a node that accepts it.

Figure 1 shows the full picture.



(a) Backend. (b) Frontend.

Figure 2: Class diagrams of the backend and the frontend.

3.3 Modelling

The core domain entity is the `Minesweeper` class. It models the grid, the mines, and the flags. Instead of passing the entire JSON board over the network on every click, we replicate commands. A client sends an action like:
`{'action': 'reveal', 'r': 2, 'c': 3}`.
The state machine applies this command deterministically across all nodes, tracking progress via a strictly monotonic `seq` number.

Figure 2 shows class diagrams describing the most important domain entities of the backend and the frontend.

- **Which domain entities are there?**
  - *Minesweeper:* the game logic itself. Holds the grid, the mines, the flags, the revealed cells and a deterministic seed.
  - *GameStateMachine:* keeps the totally ordered command log and applies commands to each room.
  - *ChainReplicationManager:* owns the socket to the next replica and runs the PREPARE/COMMIT protocol.
  - *RoomManager:* maps each room to its connected client sockets and usernames. Does not hold game state.
  - *FollowerRuntime* and *PromotedLeaderRuntime:* the two phases a backend node can run: passive follower or active leader after promotion.
  - *ConnectionState:* small class on the client side that holds the active socket and which fallback port is in use.

- *MinesweeperController* and *MinesweeperGUI:* controller and view of the client.

- **How do domain entities map to infrastructural components?**
  - One `Minesweeper` instance per room lives on every backend node. All three nodes hold identical copies thanks to deterministic seeding and the replicated command log.
  - `GameStateMachine`, `ChainReplicationManager` and `RoomManager` live on every backend node (one instance each).
  - `FollowerRuntime` runs on F1 and F2 while they are followers. `PromotedLeaderRuntime` runs on the node that is currently acting as leader (initial Leader, or any follower after promotion).
  - `ConnectionState`, `MinesweeperController` and `MinesweeperGUI` live only on the client.

- **Which domain events are there?**
  - User joins a room.
  - User leaves a room.
  - User reveals a cell.
  - User flags a cell.
  - User restarts the game.
  - A command is prepared across the chain.
  - A command is committed across the chain.
  - Leader crashes; next-in-chain promotes.
  - Client detects disconnection and reconnects to the next available node.
  - Game won (all safe cells revealed).
  - Game lost (a mine is revealed).

- **Which sorts of messages are exchanged?**

  Between client and leader: commands (`ACTION_REVEAL`, `ACTION_FLAG`, `ACTION_RESTART`, `ACTION_JOIN`, `ACTION_CREATE_ROOM`) and queries (`GET_STATE`). The server pushes events back to the clients (`UPDATE_ROOM`, `UPDATE_USERS`, `INIT_ROOMS`, `STATE_REPLY`). Between replicas: only replication messages (`PREPARE_COMMAND`, `PREPARE_ACK`, `COMMIT_COMMAND`, `COMMIT_ACK`). User-list updates between replicas are sent fire-and-forget, outside the two-phase protocol, since the user list is transient and reconstructed on reconnection.

- **What information does the state of the system comprehend?**
  - The set of active rooms.
  - For each room, a `Minesweeper` instance: grid layout, mine positions (derived from the seed), revealed cells, flagged cells, win/loss status.

– The committed log of commands, indexed by a monotonically increasing `seq`.
– For each room, the set of connected client sockets and their usernames.
– On the client side: the local copy of the board, the username and the active connection state.

3.4 Interaction

The interaction of our system is based on the Client–LeaderServer and the LeaderServer–FollowerServers communication.

- Client–LeaderServer communication: the client sends different types of actions (reveal, flag, restart, ...) and the LeaderServer receives them and starts the LeaderServer–FollowerServers communication. After the replication protocol described below completes, the leader updates the board of the room and broadcasts the new state to all the clients that are playing that room.
- LeaderServer–FollowerServers communication: our architecture is a chain of 3 nodes (1 leader and 2 followers initially). Here is where we have implemented the PREPARE/COMMIT: the leader sends a `PREPARE_COMMAND` to the first follower, and this one resends it to the second follower and waits for its ack to also send it to the leader. When follower2 sends its ack, follower1 does it too, making the leader know that both are prepared for the `COMMIT_COMMAND`. The leader sends it and follower1 resends it to follower2 without applying the action, waiting for its ack to do it. Follower2 receives the commit command, applies the action and sends ack to follower1. Follower1 also executes the action and sends its ack to the leader, and when all the followers have applied the action and the leader receives the ack is when the leader applies the action to its board. This sequence is shown in Figure 3.

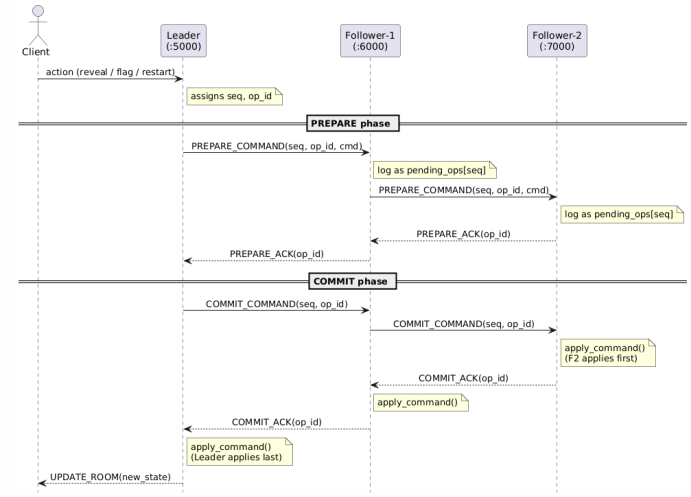


Figure 3: Normal sequence of the system

Other important issues are when the leader crashes during the prepare or commit commands.

- If the leader crashes during the prepare command, the commit phase does not start so the action is not applied. The user reconnects with follower1 and it returns the last stable state registered. We can see this sequence in Figure 4.

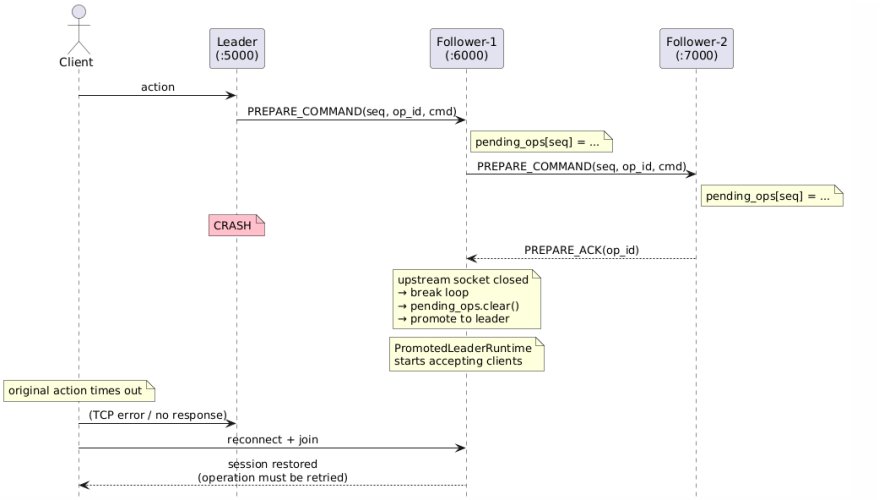


Figure 4: Normal sequence of the system but the leader crashes during the prepare

- On the other hand, if the leader crashes during the commit phase, the commit and execution of the action starts. As we see in Figure 5, the leader crashes during the commit phase (this case after receiving the F1 ack, but it is the same if it happens before this ack); F1 and F2 apply the command and when the client reconnects to F1, the board F1 returns will be with the action executed.

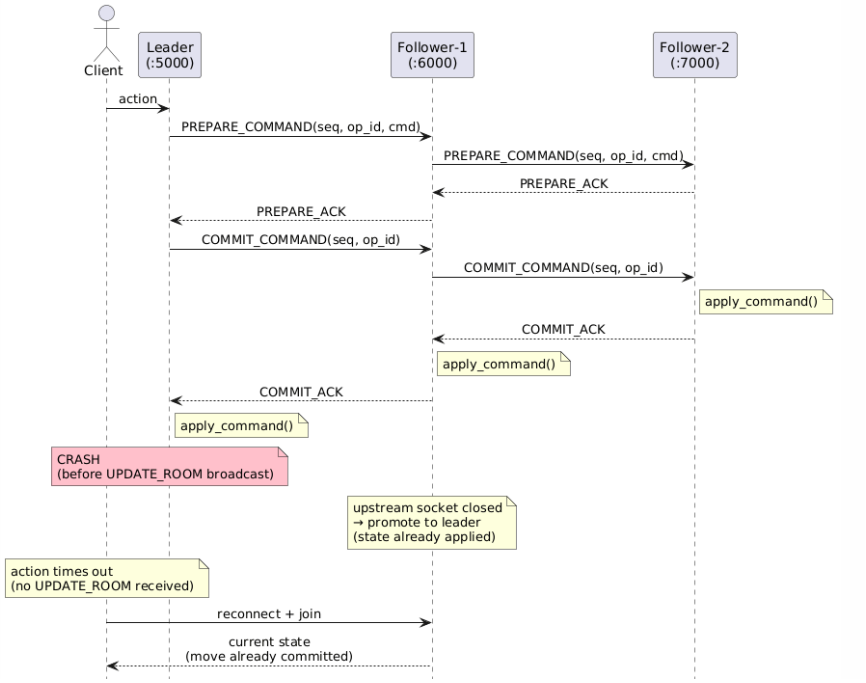


Figure 5: Normal sequence of the system but the leader crashes during the commit

When the user reconnects to F1 or F2, it becomes the new leader. These decisions were made in order to guarantee the consistency of the system over availability.

3.5 Behaviour

The most important components when we talk about the state and behaviour of the system are the servers (leader and followers). We have already described the internal behaviour of PREPARE/COMMIT in Section 3.4; here we focus on another important view: the global state of the cluster.

Currently, the system has 3 possible states:

- Healthy: Leader, F1 and F2 are alive and the system works normally. We keep strong consistency and the client connects directly with the leader.
- Reduced: exactly the same as the Healthy state, but the only alive nodes are F1 (promoted to leader) and F2, so the clients will connect with F1.
- Degraded: only F2 remains alive. The system starts behaving as a single-node server and stops working as a replicated system. We no longer have chain replication, and the system prioritises availability over consistency due to the lack of other nodes.

These state changes happen on the server side: when a follower sees that the upstream socket is closed, the `FollowerRuntime` throws away the pending operations and

`slave.py` creates a `PromotedLeaderRuntime` that starts accepting clients. The client does not know anything about this promotion: when its TCP connection drops, it just goes through the node list (leader, F1, F2) and reconnects to the first one that responds, sending a `join` to get the session back.

Degraded mode is a trade-off that we did on purpose: we prefer availability over consistency for the last node alive. If F2 also refused writes, one bad sequence of crashes would make the game unplayable even when the state machine still works fine. We assume that any other crash in this mode kills the cluster. A possible solution for this, allowing nodes to rejoin, is discussed in Section 9.

3.6 Data and Consistency Issues

According to our SMR system, each machine has its own board generated by a random seed created by the leader, who sends it to the followers inside the command that creates the room. All replicas start from the same seed so they have the same board without sharing it directly, only the commands travel through the network.

We decided not to use a database and to keep all the data in the RAM of the backend processes. The state of the game is short-lived and the replication itself already gives us durability: we have 3 copies of the state in 3 different processes, which is enough for our purposes.

But, what data do we need to store? Basically all the data related to the board: number of mines, seed, revealed cells, state of the game (playing, won, lost), placed flags, ... All of this is stored in a dictionary called `_games` inside the `GameStateMachine` class (each entry is a room). We also store the sequence counter used by the PREPARE/COMMIT protocol in the `ChainReplicationManager`, and the mapping room → clients in the `RoomManager`.

Since we have many threads running at the same time, we use a lock in each of these classes to protect their internal state. The most important one is the lock in `GameStateMachine`, which makes sure that no client reads a board in the middle of an `apply_command()`.

About data shared between components, only the commands travel through the network, never the full board. This is the main idea of SMR: if all replicas apply the same commands in the same order, they end up in the same state.

3.7 Fault-Tolerance

As we mentioned in Section 3.6, we have developed a SMR system. We don't replicate the whole board all the time. Replicas create a board with a shared random seed and all start from the same point, and by applying the commands that the client sends to the leader we make changes on the board. When all nodes have applied the command, the leader updates the board of the room and broadcasts the new state to all the clients. This is explained in Section 3.4, Figure 3. Replication is done through a chain of 3 nodes. There is one exception: the user list (`UPDATE_USERS`) is replicated fire-and-forget,

without PREPARE/COMMIT, because it is transient information that is rebuilt every time a client reconnects.

We don't follow a periodical heartbeat strategy. Instead, we have a failure detection mechanism based on three things: the close of the TCP socket, a timeout while waiting for an ack (5 seconds in our `ChainReplicationManager`), and the client reconnection. The retry mechanism lives on the client side: when its connection drops, the client iterates through the node list (leader, F1, F2) and reconnects to the first node that responds, sending a `join` to restore the session.

As we mentioned previously, the system is able to work with failures in a consistent way (Section 3.5) and these situations are explained in Figures 4 and 5 of Section 3.4. There are different cases to consider:

- When a follower crashes without the leader crashing, the operation is rejected by timeout. No promotion happens and the cluster keeps working but with no tolerance to more failures.
- When the leader crashes, F1 promotes automatically.
- If both leader and F1 crash, F2 enters the degraded mode described in Section 3.5.

We also want to explain that our system prioritises consistency over availability. The downside of this is that if F2 crashes, F2 doesn't return its ack for F1, making the leader reject the action. The same happens if F1 crashes while the leader is still alive. We do this in a conscious way to ensure the consistency of the model.

3.8 Availability

Is there any caching mechanism?
We don't use caching systems like Redis or Memcached. The game state is stored directly in the RAM of the backend servers. Since a Minesweeper game changes with every single click, caching reads wouldn't improve the performance at all.

Is there any form of load balancing?
We didn't set up a dedicated load balancer. Instead, we handled it on the client side. The client application has a fixed list of the three server addresses (Leader, F1, F2). If a connection drops, the client just tries the next one in the list until it connects to the new leader. It's a simple approach, but it keeps the game running without adding extra infrastructure.

In case of network partitioning, how does the system behave?
If the network splits, our `ChainReplicationManager` relies on socket timeouts. If F2 disconnects, F1 will notice it can't forward messages. Since we prioritize consistency (CP), the system refuses to confirm partial operations. The leader simply rejects the client's move.

However, if both the leader and F1 crash, we built a *Degraded Mode*. When F2 realizes it's the only one left, it drops the strict consistency requirement and switches to an AP (Available/Partition-tolerant) model. It accepts local writes so the remaining players can finish the match, trading fault tolerance for availability.

3.9 Security

Is there any form of authentication or authorization?
We don't have complex authentication schemas. To join a room, players just type a username. There are no passwords or admin accounts. Everyone in a room has the exact same permissions to reveal or flag cells.

Are cryptographic schemas being used?
We run the game over plain TCP without SSL/TLS encryption, mainly because the cluster runs inside a closed Docker network. Rooms are logically isolated by name, players join rooms by name, and the server strictly scopes the broadcasts so a move in "Room A" doesn't affect "Room B".

Also, we use a server-side random seed to prevent cheating. If the clients generated the mines, someone could hack their client to see where the bombs are. By creating the seed on the leader and replicating it, the server is the only source of truth.

We assume the cluster runs in a trusted environment (a private Docker network or a closed LAN). For a public deployment, TLS and authentication would be required

4 Implementation

We chose to implement everything in pure Python using only the standard library. This gave us full control over the network protocol and the replication logic, which were the core challenge of the project.

Which network protocols did you use?
Everything runs on raw TCP over IPv4. TCP is a stream-oriented protocol that does not preserve message boundaries, so several rapid clicks may arrive merged in a single `recv()` call or, the other way around, one message may be split between two `recv()` calls. To fix this, we implemented length-prefix framing. Our custom wrapper (`protocol.py`) packs a 4-byte big-endian integer header in front of every payload to specify its exact length. This guarantees we always read exactly one full command at a time.

How is in-transit data represented?
We serialize the payloads to UTF-8 JSON. We chose JSON over binary alternatives like protobuf because debugging plain text in the terminal during development is much easier, and Python parses it natively.

How should databases be queried?
Database design is not applicable to our system, as discussed in Section 3.6.

4.1 Backend technological details

The backend relies on threading to handle concurrent operations. Each server starts several threads: one for the ACK listener inside the `ChainReplicationManager`, one for each connected client, and the main loop of the `FollowerRuntime`. Because of this high concurrency, we protect our shared Python dictionaries (like the game rooms or sequence counters) using threading locks, as we previously detailed in Section 3.6.

We deploy the backend using Docker. We wrote a `Dockerfile` for the servers and set up a `docker-compose.yml` that starts the three nodes (leader, follower-1, follower-2). We use Docker's internal bridge network so the nodes can discover each other by their container names instead of hardcoded IPs.

4.2 Frontend technological details

The frontend uses the standard `tkinter` library. Since Tkinter's main loop must run in the primary thread, we organized the code using the Model-View-Controller (MVC) pattern. We have a background daemon thread handling the blocking `recv()` calls from the socket. When a new board state arrives, it uses the `gui.after()` method to safely push the visual updates to the main thread.

To containerize the client, we solved the lack of a native X display in Docker by using X11 forwarding. We set `network_mode: host` and passed the `DISPLAY` environment variable while mounting the `/tmp/.X11-unix` volume. This lets the container talk directly to the host's X11 server to render the GUI.

5 Validation

Testing concurrent crashes manually is very complex and time-consuming. Because of this, we built an automated integration suite early in the development.

5.1 Automatic Testing

We use `pytest` as our main testing framework. To run the entire test suite, you can execute `python -m pytest tests/ -v` from the root directory.

Our custom `helpers.py` script manages the cluster during the integration tests. It starts the three-node cluster using Python's `subprocess.Popen`, waits for the TCP ports to bind, and connects headless sockets to simulate clients. We organized our tests into five main areas to validate the requirements defined in Section 2:

- **Model Unit Tests:** Before testing the network, `test_model.py` checks the internal game rules. It ensures that the first click is never a mine and that adjacent counters work properly, isolating the domain logic from the distributed infrastructure.
- **SMR Ordering & Convergence:** `test_replication.py` validates **FR4** and **NFR4**. It checks that all three replicas end up with the exact same board and `last_committed_seq` after an operation is confirmed. It also verifies that the sequence advances monotonically.
- **Room Isolation:** `test_multiplayer_rooms.py` validates **FR1**, **FR3**, and **NFR3**. It proves that two clients in the same room get the same updates instantly, while ensuring a click in room A never triggers an update in room B.

- **Crash Recovery:** In `test_failover.py`, we use `proc.terminate()` to kill the leader process mid-game. We verify that F1 promotes correctly and serves the right board to a newly reconnected client. We also test cascading crashes (killing the Leader, then F1) to make sure F2 takes over and accepts new connections, fulfilling **FR7** and **NFR2**.
- **Consistency Guarantees:** `test_consistency_failures.py` validates **NFR1**. We simulate an F2 crash to ensure the PREPARE chain fails, forcing the leader to reject new operations instead of committing invalid state changes. We also check that uncommitted pending operations are correctly discarded if a failover happens.

5.2 Acceptance test

Automated tests are great, but we also ran manual Chaos Engineering sessions to cover the functional requirements that involve UI interaction (like **FR2**, **FR5**, and **FR6**). We start the Docker cluster, connect a couple of GUIs, play very fast, and abruptly run `docker stop` on the leader container. Seeing the UI freeze for a second and then recover the session on the backup node proves that the system works correctly in a real-world scenario.

6 Deployment

The whole system is shipped as a single `docker-compose.yml` at the root of the repository, which brings up the three backend replicas and, optionally, the Tkinter client. Thanks to the use of containers, the entire cluster can be installed and launched with a single command, with all the configuration centralised in environment variables.

6.1 Prerequisites

The following software must be installed on the host machine:

- **Docker.** On Linux, **Docker Engine** $\geq$ 20.10 with the Compose v2 plugin (`docker compose`). On macOS or Windows, **Docker Desktop** $\geq$ 4.0, which already bundles Compose v2.
- **Git**, to clone the repository.
- **An X11 server** running on the host. Required only on Linux when running the fully containerised stack described in Section 6.2.1, and standard on any Linux desktop distribution.
- **Python** $\geq$ 3.10 with Tkinter support. Required on macOS and Windows, where the client always runs natively (Section 6.2.2). Optional on Linux. Tkinter ships with the official Python installers on macOS and Windows, and is available as the `python3-tk` package on Debian/Ubuntu-based distributions.

Since the entire codebase relies exclusively on the Python standard library, no additional packages need to be installed.

6.2 Installation and Startup

6.2.1 Linux

On a Linux host the entire system (backend and GUI) is brought up with a single command, with both running inside containers:

1. Clone the repository:

   ```
   git clone https://github.com/nico-maire/distributed-minesweeper.git
   cd distributed-minesweeper
   ```

2. Allow Docker containers to connect to the local X11 display (once per session):

   ```
   xhost +local:docker
   ```

3. Build the images and bring up the full stack:

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

6.2.2 macOS and Windows

The fully containerised setup described above is Linux-only: the `frontend` container depends on the host's X11 server and on `network_mode: host`, neither of which is portable to Docker Desktop on macOS or Windows. On these platforms, only the three backend services are containerised, and the Tkinter client is run natively on the host.

1. Clone the repository:

   ```
   git clone https://github.com/nico-maire/distributed-minesweeper.git
   cd distributed-minesweeper
   ```

2. Bring up only the three backend services, explicitly skipping the `frontend` container:

   ```
   docker compose up --build leader follower-1 follower-2
   ```

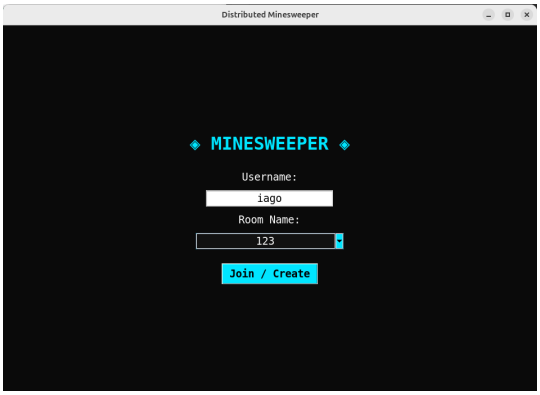
3. In a separate terminal, launch the client natively:

   ```
   python frontend/client.py
   ```

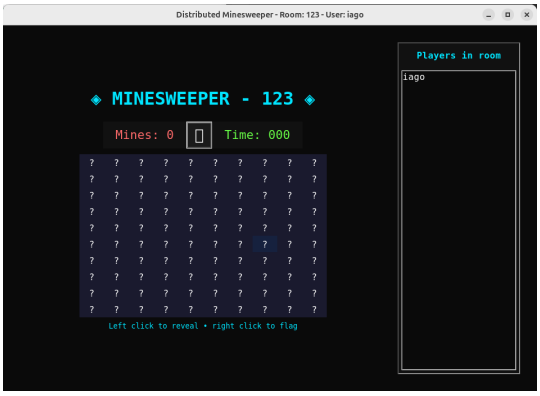
7 User Guide

After running the command described in Section 6.2, the client opens the lobby window. The user types a **username** and either selects an existing room from the dropdown or types a new one in the **Room Name** field, then clicks `Join / Create` to enter the game. If the room name is new, it is created; if it already exists, the user joins it.

First player: creating a room. The first player, `iago`, types a username and a room name (`123`) that does not yet exist on the cluster. Pressing `Join / Create` creates the room and opens the game board, as shown in Figure 6. At this point `iago` is the only user in the room, as confirmed by the *Players in room* panel on the right.



(a) Lobby: `iago` creates room `123`.

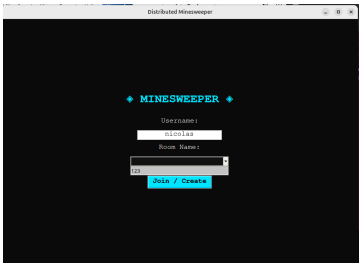


(b) Game board after joining; `iago` is alone in the room.

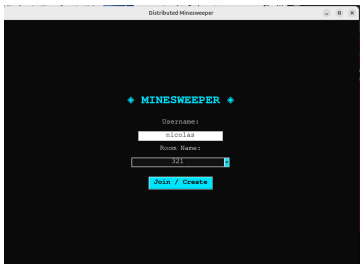
Figure 6: First player creates a new room.

Second player: creating a different room. A second player, `nicolas`, launches the client. The dropdown already lists `123` (Figure 7a), so he could join `iago`'s game, but instead he types a new room name, `321` (Figure 7b), and creates a separate game. The resulting board (Figure 7c) is independent from room `123`: a different mine layout, a distinct player list, and no shared state.

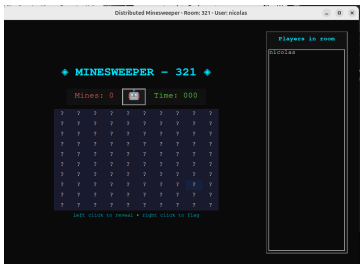
Third player: joining an existing room. A third player, `user`, opens the lobby. The dropdown now lists both active rooms, `123` and `321` (Figure 8a). The user selects `123`, joining `iago`'s game. The board in Figure 8b already shows revealed cells from `iago`'s first moves, and the *Players in room* panel lists both `iago` and `user`.



(a) Room dropdown showing `123`.

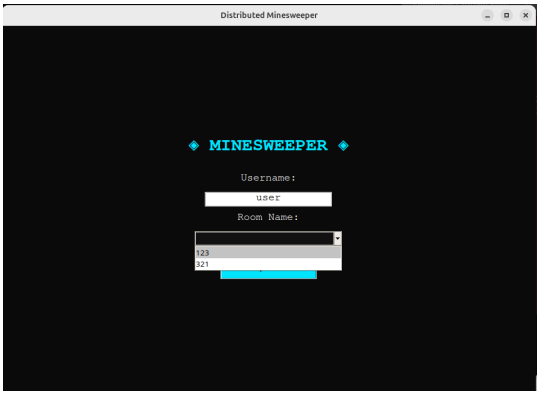


(b) Creating a new room, `321`.

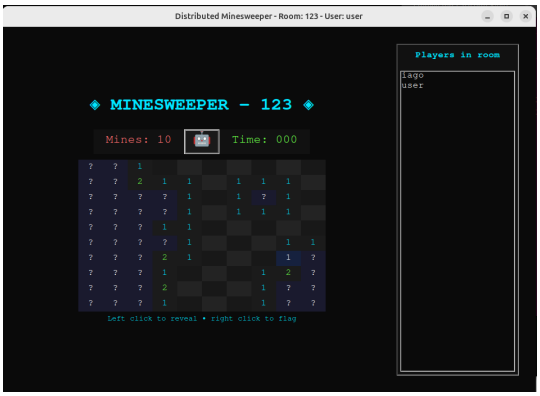


(c) Game board for room `321`.

Figure 7: Second player creates a separate, isolated room.



(a) Both rooms are visible in the dropdown.



(b) After joining, the new client sees the current board state and the updated player list.

Figure 8: Third player joins an existing room.

Playing. Once in a room, the controls are minimal and match standard Minesweeper:

- **Left click** on a cell to reveal it. The action is sent to the leader, replicated through the chain, and broadcast to every client in the same room.
- **Right click** on a cell to place or remove a flag.
- The header shows the number of `Mines` remaining, the elapsed `Time`, and a smiley face that doubles as a **restart button**: clicking it generates a new board with a fresh seed.

The game ends when either all safe cells are revealed (win) or a mine is revealed (loss). All connected clients in the room see the final state simultaneously.

8 Self-evaluation

- **Nicolás Maire Bravo:** My role was to design the foundation of the distributed backend. I built the initial Chain Replication system between the nodes and programmed the basic TCP socket communication. I also created the first Docker Compose configuration. Later, I worked on integrating the different parts of the system and coordinating the development phases.

  Strengths of the product: The leader failover mechanism is one of the strongest parts of the system. If the leader crashes, the next follower can detect the closed upstream connection, discard uncommitted pending operations, promote itself, and continue serving clients from the last committed state. This allows the game to continue after a leader failure without losing already committed moves.

  Weaknesses of the product: The degraded mode keeps the game available when only one server is left, but at that point the system loses fault tolerance because there are no remaining replicas. Also, the system does not use persistent storage: all game data is stored in volatile memory, so if the whole cluster crashes, the current games are permanently lost. Finally, since this is a prototype intended to run in a controlled environment, we did not implement real authentication, authorization, or protection against malicious clients.

- **Iago López Lamas:** My role started with the client-side development. I created the graphical interface (GUI) using Tkinter and organized the code using the MVC pattern. After that, I worked on the backend to implement the State Machine Replication (SMR) and the two-phase PREPARE/COMMIT protocol. I also wrote the automated integration tests using `pytest` and configured the X11 socket in Docker so the GUI could run inside a container.

  Strengths of the product: The SMR design guarantees that all alive replicas converge to the same board state for committed operations, avoiding inconsistent matches between players. The PREPARE/COMMIT protocol is also a strong point, since a move is not confirmed just because the leader received it, but only after the replication chain has safely processed it. This helps prevent phantom writes and unsafe partial updates. In addition, the use of sequence numbers gives a strict order to multiplayer actions, and the automated integration tests confirm the main consistency and fault-tolerance mechanisms under realistic crash scenarios.

  Weaknesses of the product: To guarantee consistency, if F1 or F2 crashes and the replication chain cannot be completed, the system may stop accepting new actions instead of risking an inconsistent state. This limitation is made worse by the fact that there is no node rejoin or catch-up mechanism: once a node crashes, it cannot automatically re-enter the chain with the updated state.

  Moreover, the system uses a fixed topology and does not implement a full consensus algorithm or dynamic membership, so adding new nodes or changing the chain at runtime is not supported. Regarding scalability, the leader can become a bottleneck

because it is the only node that receives and orders the actions of all clients from all rooms.

9 Future works

The system as it is satisfies the requirements from Section 2, but several limitations are left as future work on purpose. We split them in two stages: short-term improvements that keep the current topology and protocol, and a deeper redesign aimed at scalability.

9.1 Short-term: extending the current design

These changes preserve the chain replication topology and the PREPARE/COMMIT protocol, and they are the natural next step we would take.

Node rejoin. As mentioned in Section 3.7, a crashed node does not rejoin the cluster on restart. To fix this we would:

- Add a `REJOIN` message in the `protocol` module, sent by a restarting node to the current leader.
- Have the leader pause new operations briefly, snapshot the `_games` dictionary together with the latest sequence number from `ChainReplicationManager`, and send it to the rejoining node.
- Append the rejoining node at the tail of the chain, updating the downstream pointer of the last `FollowerRuntime`.
- Resume normal operation. From this point on the rejoined node is just another follower.

A small change in the `server` bootstrap is also needed, because right now a restarted node has no way to discover that the cluster has moved on without it.

Larger chain with configurable size. The current deployment uses one leader and two followers, so the degraded (AP) mode is reached after only two failures. A simple improvement is to make the chain length a parameter and run, for example, with four nodes. This delays degraded mode by one more failure and gives a bigger window in which rejoin can repair the cluster before consistency is lost.

Tolerating follower failures mid-transaction. Right now, if a follower crashes *during* the PREPARE phase, the operation stalls until the chain is reconfigured. We would like the system to keep accepting client commands even while a non-leader node is failing. The plan is:

- Add a timeout on PREPARE inside `ChainReplicationManager`. If the ACK does not come back within a bounded time, the leader assumes the downstream node is dead.
- Apply the same chain reconfiguration mechanism that already runs when a socket closes, but allow it to run while a transaction is still in progress.
- Retry the PREPARE on the reconfigured chain, transparently to the client.

The client should not need to know that anything happened, beyond a slightly higher latency for that single command.

Persistent state. A smaller but useful change: persist the `_games` dictionary to disk after every COMMIT applied by the leader. This way, a full cluster shutdown (or a power cut on all three machines) does not lose ongoing games, and rejoin becomes much cheaper because the rejoining node can start from its own snapshot and only ask the leader for the missing commands.

9.2 Long-term: redesign for scalability

The current architecture has an obvious bottleneck: every command, for every room, goes through one single leader. This is a constraint to scale to many concurrent rooms. We would change three things at once.

- **One leader per room.** Instead of a single global leader, there would be a coordinator node whose only job is to spawn a dedicated replication group (a small leader plus its followers) for each game room. The coordinator handles room creation and client routing, but commands inside a room never go through it. Independent rooms become fully independent, which removes the global bottleneck.
- **Parallel replication instead of chain.** Inside each room, the leader would send PREPARE in parallel to all its followers and wait for a quorum of ACKs, instead of walking down a chain.
- **Consensus-based leader election.** Without a chain, the deterministic "next-in-chain" promotion does not apply anymore, so leader election has to be done with a proper consensus algorithm such as Raft or Paxos. This also gives correct behaviour under more complex failure scenarios like network partitions, which our current design does not really handle.

As third-year undergraduate students, we believe that the features described above would be challenging for us to implement properly. Due to our limited experience in networks and distributed systems, together with time constraints, we did not develop them.

References

[1] Andrea Omicini. Distributed systems - course slides and lecture notes, 2026. Master's Degree in Computer Science and Engineering, University of Bologna, Campus of Cesena.