Distributed Bomberman

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

Giovanni Rinchiuso
giovanni.rinchiuso@studio.unibo.it

Gioele Santi
gioele.santi2@studio.unibo.it

September 11, 2026

Distributed Bomberman is a real-time, fault-tolerant, multiplayer game inspired by the classic Bomberman arcade title, developed as a project for the Distributed Systems course at the University of Bologna (A.Y. 2024/25). Up to four players compete simultaneously on a shared grid, placing bombs to eliminate opponents. An unlimited number of spectators can observe the match and may request to join as a player when a slot becomes available. The system is built on an authoritative client-server architecture fronted by a session-aware TCP proxy, which hides a primary/backup server pair from clients. State is continuously replicated from the primary to the backup; on primary failure, the backup promotes itself automatically, re-binds the game port, and client sessions are restored transparently by the proxy; a second, independent retry loop on the client covers the failure of the proxy itself. The entire system is implemented in Python 3 using the standard library for networking and Pygame for the graphical client.

Contents

1 Concept

Distributed Bomberman is a system consisting of three kinds of cooperating process: a game server (deployed as a primary and a hot-standby backup), a transparent TCP proxy, and a Pygame graphical client.

Use cases. The main actors are the **Player**, the **Spectator**, and the **Host** (the first player to join, a special role assigned automatically). Table 1 summarises the principal use cases.

Use case	Actor	Description
Join lobby	Player / Spectator	Connect, receive assigned name, enter lobby or spectator queue
Start game	Host	Press Enter in the lobby when ≥ 2 players are connected
Move	Player	Send directional command; server updates position
Place bomb	Player	Place bomb at current tile; server handles detonation
Send chat	Player / Spectator	Broadcast a text message to all participants
Join as player	Spectator	Press J in the lobby; server converts spectator if a slot is free
Observe match	Spectator	Receive live state snapshots; no game interaction

Table 1: Use cases of Distributed Bomberman.

Players launch a client on their own machine; the client connects automatically to the proxy at a preconfigured address (default: `localhost:5555`) and is assigned a random name and a slot in the lobby. Once at least two players have joined, the host can start the match. Players move on a 2D grid, place bombs, and try to eliminate opponents; a player that runs out of lives is eliminated. Spectators can join at any time and watch the match; while in the lobby, a spectator can request to become a player by pressing J, and is promoted if a slot is available. The game ends when only one player remains alive (or all die simultaneously, resulting in a draw). After a brief countdown, all players are returned to the lobby for a rematch.

Why distribution? Distribution is the core design driver:

- **Multi-player real-time gameplay** requires clients on different machines to share a consistent, up-to-date view of the game state.
- **Authoritative server** centralises game logic, preventing clients from cheating and ensuring consistency.

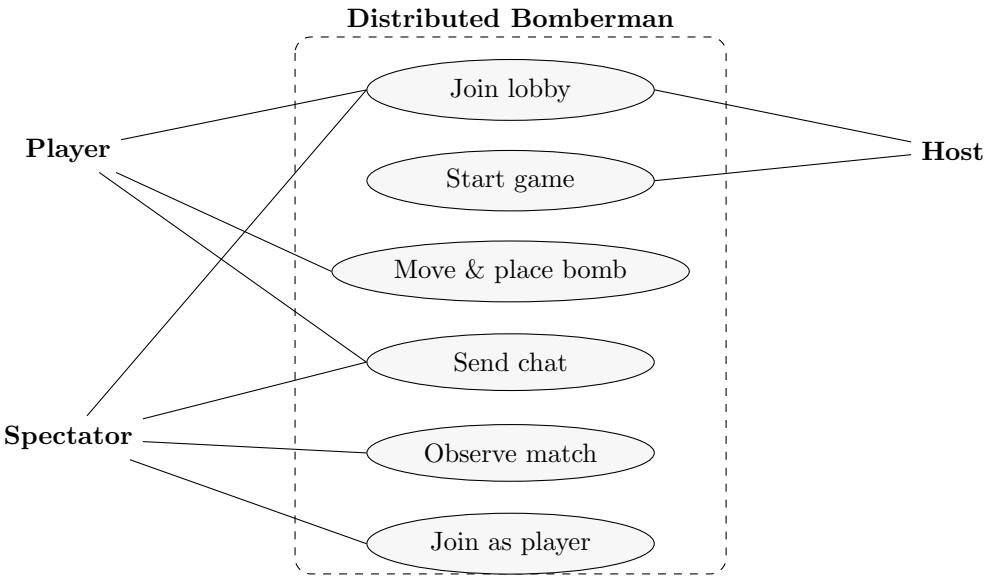

Figure 1: Use case diagram of Distributed Bomberman.

- **Fault tolerance**: if the primary server crashes, a hot standby backup takes over automatically; clients experience a brief pause but do not need to reconnect manually.

2 Requirements Identification and Analysis

2.1 Functional Requirements

F1 Up to four players shall be able to join a shared lobby and play simultaneously on the same map.

F2 An unlimited number of spectators shall be able to join and observe the match in real time.

F3 While in the lobby, spectators shall be able to request to join as a player by pressing a dedicated key; the request is accepted if a player slot is currently available.

F4 The first player to join shall become the host and shall be able to start the game.

F5 Players shall be able to move on the grid and place bombs; bombs shall explode after a timer, destroying breakable blocks and damaging players in range.

F6 A player who disconnects during a match shall be marked as disconnected and excluded from the active game; their session can be restored under the original identity — by the proxy after a server failover, or by the client itself once it re-establishes a dropped connection.

F7 Players and spectators shall be able to send chat messages visible to all participants.

F8 The system shall declare a winner (or a draw) and return all players to the lobby automatically.

2.2 Non-Functional Requirements

NF1 **Fault tolerance**: the system shall continue operating if the primary server crashes; clients shall not need to manually reconnect.

NF2 **Consistency**: all clients shall observe the same game state; the server is the single source of truth.

NF3 **Latency**: the game loop shall tick every 100 ms; the state broadcast interval shall not exceed 100 ms.

NF4 **Transparency**: clients shall be unaware of the primary/backup topology; after a failover, the proxy shall restore all active client sessions to the new primary without requiring manual reconnection.

2.3 Relevant Distributed System Features

Fault tolerance and dependability. The primary availability concern is that a server crash should not abort the ongoing match. The system addresses this with a passive backup that receives state snapshots from the primary every 100 ms and monitors it

via heartbeat probes. On failure detection (1.5 s timeout), the backup promotes itself and rebinds the game port; the proxy then reconnects automatically. The mechanism is straightforward and sufficient for the two-node, single-failure-domain scenario targeted by this project.

Transparency. Clients connect to a fixed proxy address and are unaware of the primary/backup topology (location and replication transparency). During a failover, the proxy buffers incoming client data and flushes it once the new primary is reachable, making the switchover invisible to clients. This is a limited form of failover transparency: it works well for short outages, but the proxy enforces a 30 s timeout (`TCPProxy.FAILOVER_TIMEOUT`) beyond which client connections are dropped.

Consistency. The authoritative server pattern provides a simple consistency model: only the server mutates game state, and clients are pure renderers. All client inputs are serialised through `GameService`'s single `RLock`, so there are no concurrent write conflicts. One limitation worth noting is that the backup may hold a snapshot up to 100 ms old at the time of failover, meaning a small amount of recent game state could be lost in a crash.

Performance and concurrency. The server uses one thread per client connection plus a dedicated game-loop thread. State is broadcast to all clients every 100 ms, which is adequate for a LAN setting with a small number of players.

Openness and evolvability. Communication uses TCP with line-delimited JSON, a deliberately simple protocol: command verbs are dispatched by `CommandController` and the shared parameters (ports, tick rate, map size) live in `src/common/constants.py`. State snapshots carry a `"version"` field so that incompatible schema changes can be detected at runtime, and the three-process boundary means components can be updated independently within the constraints of the current protocol.

3 Design

3.1 Architecture

The system follows a **client-server** architectural style with a **proxy** interposed between clients and the server cluster.

The server cluster consists of a **primary** server and a single hot-standby **backup**, automatically respawned after each promotion. The primary is the authoritative game engine; it replicates its state to the backup and exposes a heartbeat endpoint. The backup monitors the primary and promotes itself on failure. The proxy hides this topology from clients: it forwards traffic to a fixed backend port (the game port), buffers client data during failover, and reconnects to the new primary automatically.

This design was chosen because:

- A centralised authoritative server simplifies consistency: there is a single source of truth.
- The proxy provides location transparency and simplifies client implementation.
- The primary/backup pattern provides fault tolerance without the complexity of consensus protocols, which is appropriate for a two-node cluster with a single failure domain.

Alternative architectures considered. The following alternatives were discarded in favour of the chosen design:

- **Pure peer-to-peer**: Each client holds its own copy of the game state and propagates updates to peers. This avoids a single point of failure but introduces distributed consistency problems: concurrent inputs from multiple clients can easily produce diverging game states. Conflict resolution in a real-time game (e.g. two players moving to the same cell simultaneously) would require a consensus round for every input, adding unacceptable latency.
- **Direct client-to-server without proxy**: Clients connect directly to the primary. This would expose the primary's address to clients, making transparent failover impossible without requiring clients to implement their own discovery logic. The proxy layer is what enables location transparency and automatic reconnection.
- **Active-active replication (multi-master)**: Both primary and backup accept writes simultaneously. This would improve write availability but requires conflict resolution for concurrent game-state mutations, which is considerably more complex than the passive-backup approach and unjustified for a two-node cluster.

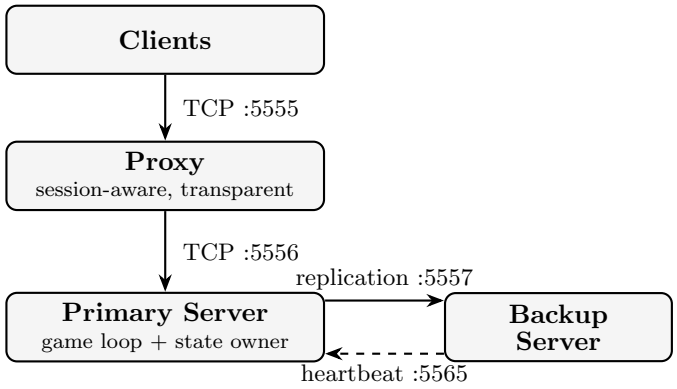

Figure 2: High-level architecture of Distributed Bomberman.

3.2 Infrastructure

Infrastructural components. Three process types are required to run the system. The client can run on a separate machine from the proxy and server cluster, which must currently be co-located: the proxy's backend address and the primary's replication target are hardcoded to `localhost`, so the proxy and the server cluster cannot yet be split across machines.

- **Server cluster** ($1 + 1$ instances): one active *primary* server (port 5556) and one *backup* server, which listens on port 5557 for incoming state snapshots (`BACKUP_STATE_PORT`) and, on promotion, re-opens the game socket on the primary's own port 5556, so that the proxy finds the new primary at the fixed backend address it already uses. Only the primary hosts the live `GameService` and accepts client connections; the backup runs in passive monitoring mode, receiving periodic state snapshots and waiting to be promoted. The primary automatically spawns the backup at startup via the `AutoSpawner` component; each newly promoted backup spawns a new one, maintaining fault tolerance continuously.
- **TCP proxy** (port 5555): a transparent relay that hides the primary/backup topology from clients. It forwards data in both directions, captures per-client `session_id`s from the server's join acknowledgements, and manages failover transparently by buffering client data and sending `RECONNECT` handshakes to the new primary.
- **Client nodes** (N instances): Pygame graphical client applications that connect to the proxy at a pre-configured address. Each client is a pure renderer: it sends input commands and renders whatever state snapshot the server broadcasts.

Network distribution. All components communicate over standard TCP/IP sockets on a local area network.

Service discovery. Components locate each other through **static port configuration** defined in `src/common/constants.py`. This is a deliberate design choice justified by the deployment scenario:

- The server cluster is a fixed two-node setup (primary + backup); there is no need to discover a dynamically changing set of server nodes.
- Clients always connect through the proxy at a single, well-known address; if the proxy address changes, clients simply update the one constant.
- A dynamic discovery mechanism (e.g. UDP broadcast) would add complexity without benefit for a LAN-local system with a fixed topology.

3.3 Modelling

Domain entities. The key domain entities are:

- **Player**: position (x, y), name, lives, alive/disconnected status, reconnect session ID.
- **Spectator**: name, join time, connected status, reconnect session ID.
- **Bomb**: position, countdown timer, owner player ID.
- **Explosion**: set of affected tiles, remaining display ticks.
- **State**: the complete, versioned snapshot of the game – game phase (`lobby / playing / victory`), map, all players, spectators, bombs, explosions, chat messages, winner.

Domain class diagram. Figure 3 shows the key domain classes and their relationships. Secondary entities (`Explosion`, `Spectator`) are omitted from the diagram for clarity but are part of the `State` snapshot.

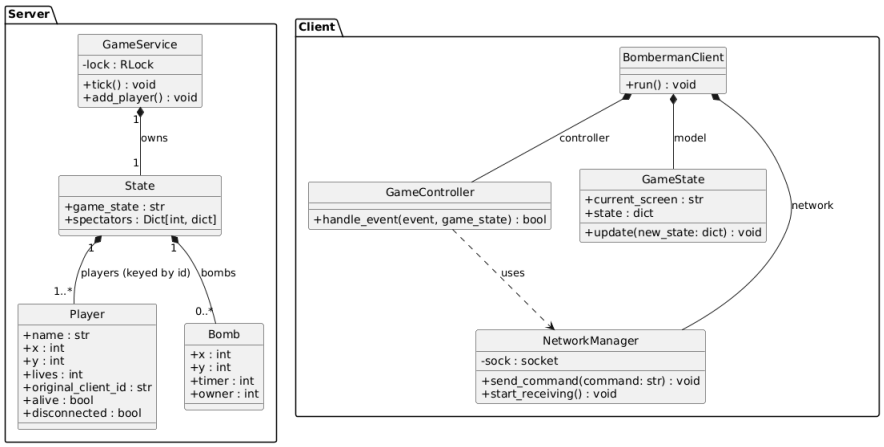

Figure 3: Domain class diagram of Distributed Bomberman.

Component overview. Figure 4 shows the main components of the four running processes — proxy, primary, backup and client — and the channels connecting them.

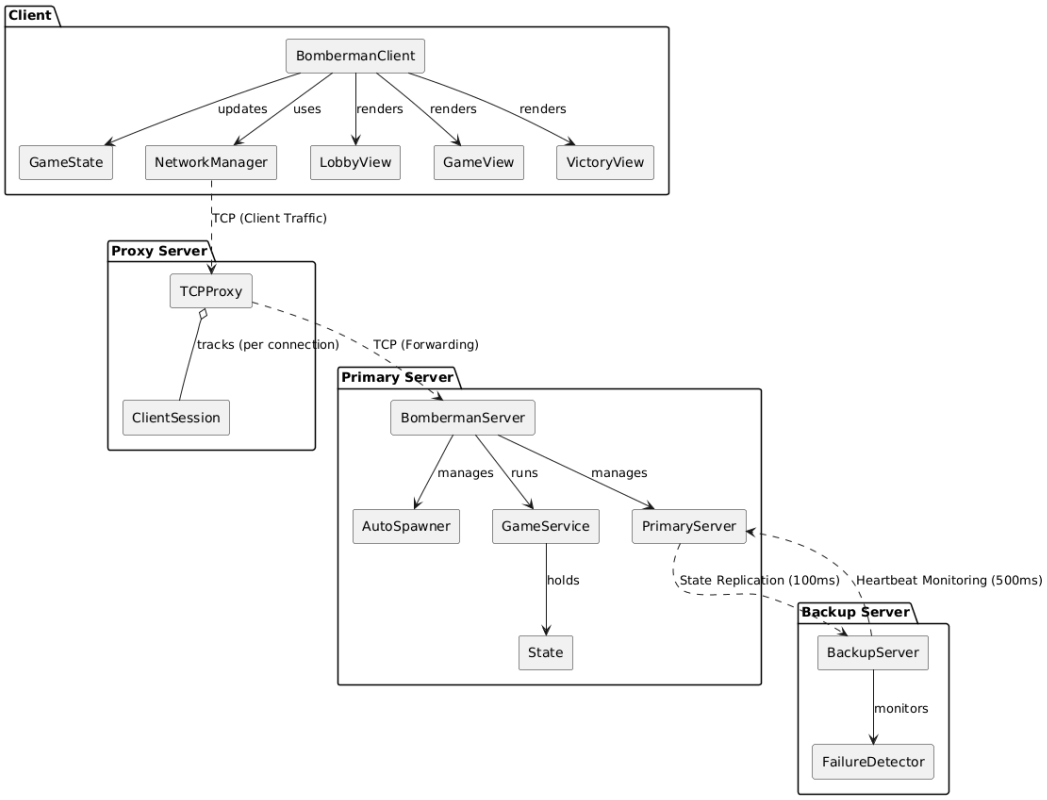

Figure 4: Component overview of Distributed Bomberman.

State ownership. The `State` object lives exclusively on the server. Clients receive a serialised projection of the state at each tick; they never mutate it directly.

Two distinct serialisations are used. Clients receive a *phase-dependent projection* produced by `GameService.get_state()`: it always carries the game phase, players, spectators, chat and host, and adds only the fields relevant to the current phase — lobby eligibility flags in `lobby`, the map, bombs and explosions in `playing`, the winner and countdown in `victory`. Replication to the backup instead uses the full versioned codec (`state_to_dict` / `state_from_dict`), which serialises every field without exception: the backup must be able to resume the match from that snapshot alone, so a partial projection would be insufficient.

Messages. Three classes of messages are exchanged:

- **Commands** (client → server): `UP`, `DOWN`, `LEFT`, `RIGHT` (movement), `BOMB`, `START_GAME`, `PLAY_AGAIN`, `JOIN_GAME` (spectator requests a player slot), `CHAT:{message}`. The server also accepts a `PING` command for manual connectivity checks (e.g. over `netcat`); the graphical client never sends it. Commands are case-insensitive and newline-terminated.

- **State snapshots** (server → client): the JSON projection of the current state described above, sent every 100 ms.
- **Control messages**: `RECONNECT:{session_id}` (sent to the server by the proxy after a failover, or by the client after it re-establishes a dropped connection), `HEARTBEAT` / `ALIVE` (backup ↔ primary).

3.4 Interaction

Interaction pattern. Distributed Bomberman uses a **pure client-server** interaction model: all communication — game commands, state updates, chat — flows through the proxy to the primary server.

This choice was deliberate:

- **Consistency**: a single serialisation point (`GameService`'s `RLock`) eliminates the possibility of concurrent writes and distributed state divergence.
- **Simplicity**: clients have no peer-discovery logic, no direct connections to other clients, and no need to synchronise local replicas.
- **Fault tolerance**: the proxy-mediated model makes failover transparent; clients never know which physical server they are talking to.

The trade-off is that the server becomes a bandwidth bottleneck for state broadcasts, which is acceptable given the small player count (max 4) and the 100 ms tick rate.

Communication model. All messages are serialised as **line-delimited JSON** over persistent TCP connections (one connection per client, kept open for the entire session). The server operates in a **publish-subscribe** style for state updates: after each game tick it pushes a state snapshot to all connected clients simultaneously, without waiting for a request. Client-to-server commands follow a **fire-and-forget** pattern: the client sends a command (e.g. `UP`) and immediately continues; the effect appears in the next state broadcast.

Join sequence. Figure 5 shows the message flow when a client connects.

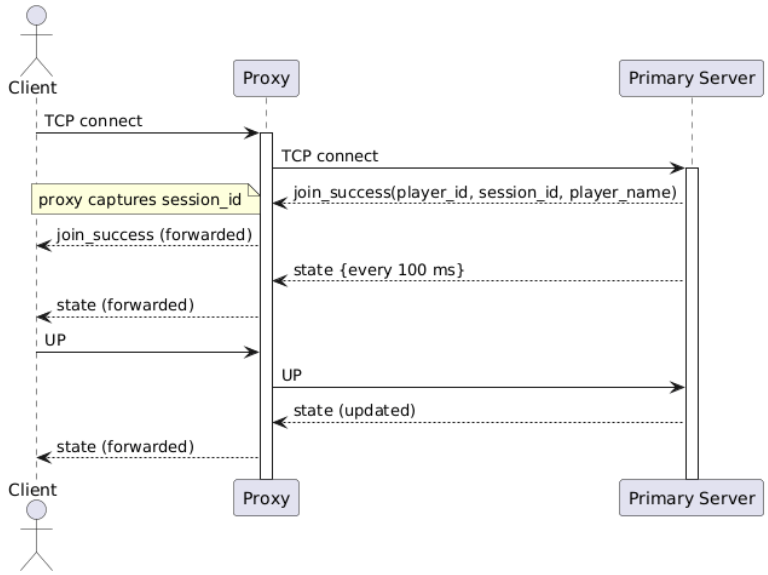

Figure 5: Join and normal operation sequence diagram.

On join, the server sends a `join_success` JSON object containing the player's assigned ID and a unique `session_id` (UUID4). The proxy captures this session ID and uses it to send a `RECONNECT:{session_id}` message to the new primary after a failover.

Game-start sequence. Figure 6 shows the message flow when the host starts the match.

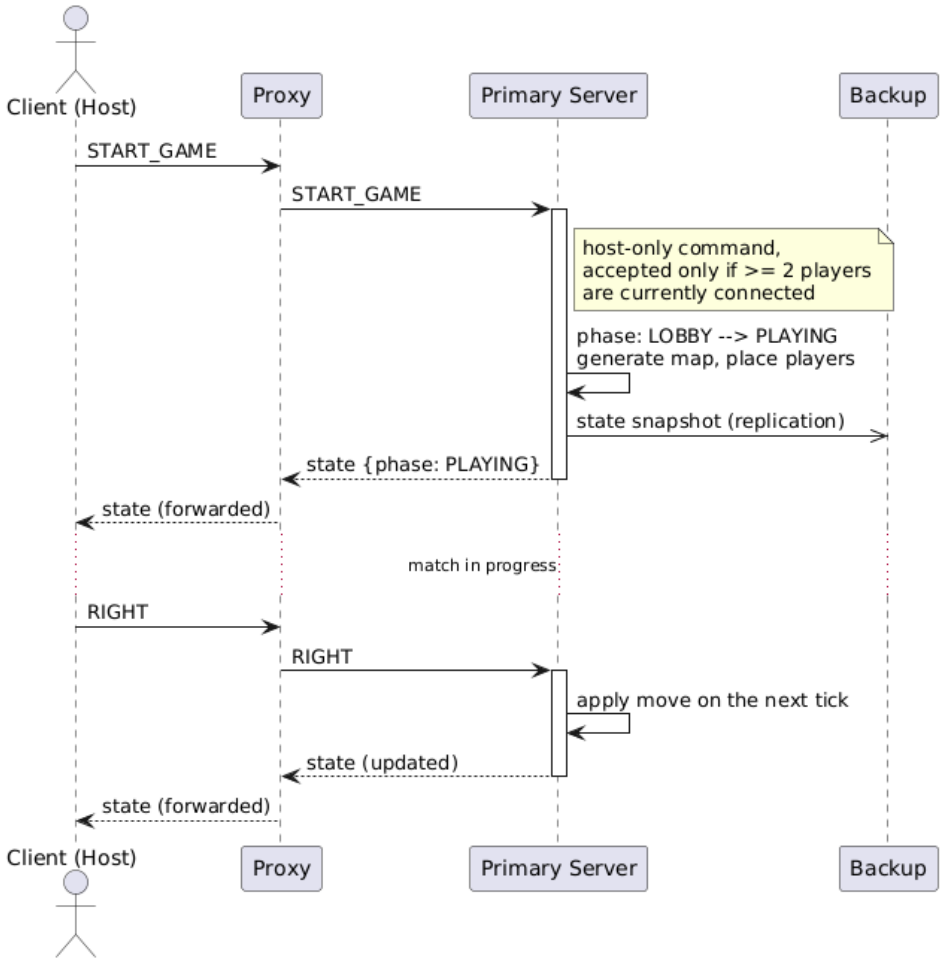

Figure 6: Game-start sequence diagram.

Failover sequence. Figure 7 shows the message flow during a primary server failure and recovery.

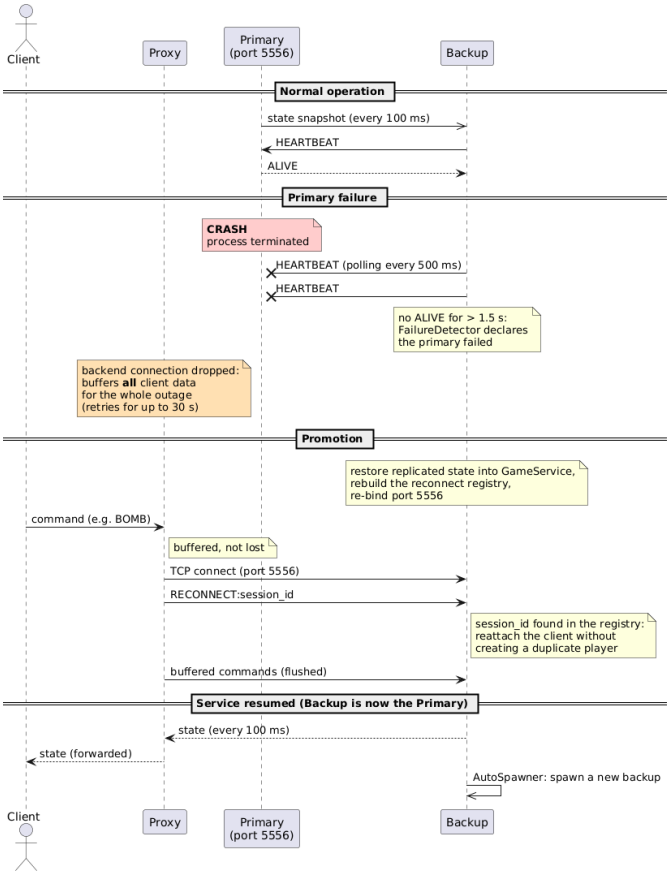

Figure 7: Failover sequence diagram.

Primary-to-backup replication uses a separate TCP connection per snapshot: the primary opens a fresh connection to the backup's state-receiver port, sends a length-prefixed JSON payload, and closes it. This approach is simple to implement but creates a new connection every 100 ms; a persistent replication channel would be more efficient, though the overhead is negligible at this scale.

3.5 Behaviour

Game phase state machine. The server `State` object progresses through three distinct phases (Figure 8):

LOBBY: The server waits for players to connect. The host can start the game when at least two players are present. Spectators may request a player slot by pressing J (lobby only; not permitted during an active match). A player who disconnects in the lobby is removed immediately.

PLAYING: The game loop is active. Players move and place bombs; the server processes inputs, detonates bombs, computes explosions, and reduces player lives. A player who disconnects mid-game is marked `disconnected=True, alive=False`, and the dis-

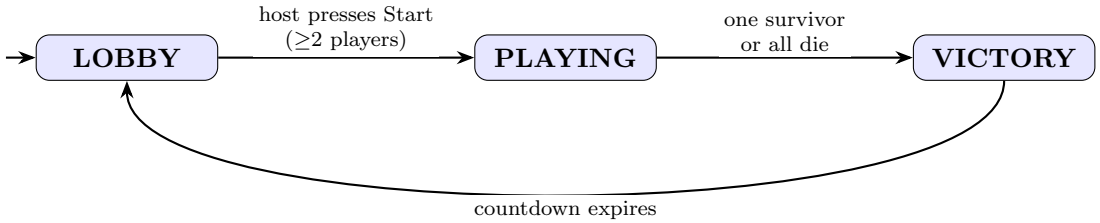

Figure 8: Server game phase state machine.

connection immediately triggers a victory check. Session restoration via the `RECONNECT` handshake is triggered either by the proxy after a primary failover, or by the client itself after it re-establishes a dropped connection — provided the client process stayed alive and still holds its `session_id`. Reattaching therefore restores the player's identity and clears the disconnected flag, but it does not undo a phase transition that the victory check has already performed.

VICTORY: One player has survived (or all died simultaneously for a draw). The winner is recorded. A countdown runs (the "return to lobby" timer); when it expires, the state is reset to LOBBY and a new match can begin.

Game loop (server). The server runs a fixed-rate game loop at 10 Hz (100 ms per tick). Each tick:

1. If the phase is `victory`, decrement the return-to-lobby timer and skip the rest of the tick.
2. Decrement bomb timers; detonate expired bombs, compute explosion cells, destroy the blocks they reach and reduce player lives.
3. Decrement explosion display timers; remove expired explosions.
4. Check the victory condition; if it is met, stop here.
5. Decrement the block-regeneration timer; when it expires, place a new destructible block on a safe tile and re-arm the timer with a random interval.
6. Broadcast the current state projection to all connected clients and spectators.

Client reactive behaviour. The client does not implement its own state machine. Instead it behaves as a **reactive consumer**: it renders whatever state the server broadcasts. The `GameState` model inspects the `game_state` field of each received snapshot and updates `current_screen` accordingly; the main loop then switches the active view:

- `lobby` ⇒ `LobbyView`: displays connected players, spectators, and the "Press ENTER" start prompt (host only).

- `playing` ⇒ `GameView`: renders the grid, players, bombs, and explosions in real time.
- `victory` ⇒ `VictoryView`: shows the winner (or draw) and the return-to-lobby countdown.

This design keeps failover handling minimal on the client: when the proxy restores the TCP stream, the next state broadcast simply resumes rendering, and no view is aware that a switchover occurred. The client is not entirely passive, however — it owns the recovery path for a failure of the proxy itself, described in section 3.8.

3.6 Data and Consistency Issues

No persistent storage is used: all game state is in memory. State is only replicated between the primary and the backup for fault tolerance purposes, not for durability. On a complete cluster failure (both primary and backup crash simultaneously), the game state is lost.

Concurrent access to the `GameService` is serialised through a single `threading.RLock`, ensuring that client handler threads and the game loop thread never mutate the state concurrently.

Consistency model. From the clients' perspective, the system provides a straightforward consistency model: every state snapshot broadcast by the server reflects all inputs processed up to that tick, and clients never modify state locally. In practice, clients observe state that is at most one tick old (100 ms), which is acceptable for the target use case.

Between the primary and the backup, consistency is **eventual** rather than strong: the backup's replicated state lags the primary by at most one replication interval (100 ms). This means that in the event of a primary crash, the backup may be missing up to 100 ms of game events. This trade-off was intentional: synchronous replication (waiting for a backup acknowledgement before processing each input) would have added per-tick latency unacceptable for a real-time game.

State versioning. Each snapshot sent to the backup includes a `"version"` field (currently `1`). The backup checks this field on receipt and discards snapshots with an incompatible version. This prevents a stale or malformed snapshot from corrupting the backup's state after a code upgrade.

Replication completeness. The state codec (`state_to_dict` / `state_from_dict`) serialises the complete game state, including all player fields (position, lives, session IDs, disconnection status), all active bombs and explosions, spectator list, chat history, and game phase. The serialisation round-trip is verified by a dedicated test module (`test_state_codec.py`) to ensure that no field is silently dropped or corrupted during replication.

3.7 Fault-Tolerance

State replication. The primary serialises the full `State` as a versioned JSON snapshot and pushes it to the backup every 100 ms. The backup deserialises the snapshot and keeps it in memory as `replicated_state`. On promotion, the backup restores this state directly into its `GameService`.

Failure detection. The `FailureDetector` class tracks the timestamp of the last successful heartbeat. It declares the primary failed if no heartbeat has been received for more than 1.5 s. The backup polls the primary's heartbeat port (TCP, short-lived connections) every 500 ms.

Post-failover session restore. When a player disconnects, the server removes the entry outright if the game is in the lobby; during a running match it keeps the entry and marks it `disconnected=True, alive=False`, so that there is still something to reattach to later.

The `RECONNECT` handshake has two senders. After a primary failover the proxy sends the `session_id` it captured from the original `join_success`; after a dropped connection the client sends the `session_id` it stored itself. In both cases the primary looks it up in the reconnect registry — rebuilt from the replicated state after a promotion — and re-attaches the client without creating a duplicate player entry, clearing the disconnected flag and restoring the player to active play. The restore is deliberately conservative: it acts only on a player marked `disconnected`, so a player eliminated in combat is never resurrected by reconnecting.

Proxy buffering. During failover the proxy keeps client-side TCP connections open and accumulates everything the client sends. Crucially, the reconnection retry loop polls the client socket non-blockingly on every attempt. An earlier implementation blocked inside the retry wait, which also stalled the relay loop: the client socket was never read for the duration of the outage, so only the chunk in flight at the moment of detection was ever captured, and every command issued during the outage was lost. Once the new primary accepts the connection, the `RECONNECT` handshake is sent and the accumulated buffer is flushed before normal forwarding resumes.

3.8 Availability

The proxy acts as a single logical entry point. It does not perform load balancing (there is only one active server at any time) but it hides the primary/backup switchover from clients, so that service continues across a primary crash.

There is no caching layer: clients always receive a fresh state broadcast every 100 ms.

CAP theorem analysis. In the CAP theorem framework, a distributed system can guarantee at most two of **Consistency**, **Availability**, and **Partition tolerance**. Par-

tition tolerance is not an optional property in any real network — partitions can always occur — so the meaningful question is how the system behaves *when* one happens.

In the absence of partitions, Distributed Bomberman provides both consistency and availability: all clients observe the same game state, since the authoritative server is the single source of truth and clients never mutate state locally, and the primary/backup pair combined with proxy buffering keeps the service running across a primary crash.

Under a partition, however, the system favours **availability** over consistency. The backup promotes itself unilaterally as soon as the heartbeat times out (1.5 s), with no quorum and no fencing mechanism to demote the old primary. If a partition separates primary and backup while both processes remain alive, both can act as primary simultaneously — a **split-brain** — and their states diverge with no reconciliation once the partition heals. The system therefore does *not* guarantee consistency under partition: in CAP terms it is **AP**-leaning. Making it genuinely CP would require replacing the timeout-based promotion with a quorum-based leader election over an odd number of nodes, for example Raft, together with a fencing token ensuring that a demoted primary cannot continue to serve clients. This was judged disproportionate for a two-node cluster in a single failure domain, but it is the principal correctness limitation of the current design.

A second structural limitation is that the proxy is a **single point of failure** for availability: while it is down no client can reach the cluster, even though both servers are healthy. Its impact is partly bounded by a second and independent recovery layer implemented on the client: when the connection drops the client retries the proxy address every two seconds and, once the proxy is reachable again, replays the `RECONNECT:{session_id}` handshake itself, so the primary reattaches the existing player instead of creating a new one.

The two layers are not equivalent, however, and the asymmetry is instructive. A primary crash is invisible to clients precisely because the dying primary never observes the disconnection: the backup resumes from a snapshot in which every player is still alive. A proxy crash is different, because the primary stays alive and does observe every client socket closing. Each disconnection marks the player `disconnected` and `alive=False` and triggers a victory check, so in a two-player match the first drop already ends the round. The client's retry loop therefore preserves the *session* — players reattach under their original identity rather than rejoining as new ones — but it does not preserve an *in-progress match*. Making proxy failure fully transparent would require the primary to distinguish a transport failure from a player quitting, for instance by suspending the victory check for a grace period after a disconnection instead of evaluating it immediately.

4 Implementation

4.1 Technology Stack

The system is implemented in **Python 3.10+**, using only the standard library for networking (`socket`, `threading`, `json`). The graphical client is built with **Pygame 2.5**, chosen for its suitability for real-time 2D rendering and its straightforward event loop.

Inter-component communication uses **JSON over TCP**, a deliberate choice over binary formats for human-readability and ease of debugging. The project is packaged with **Poetry** for reproducible dependency management, with entry points `bomberman-server`, `bomberman-proxy`, and `bomberman-client`. The test suite is written with Python's **unittest** framework and run with **pytest**. Session IDs are generated using **UUID4** from the standard library.

Network protocol. All communication uses **TCP** (Python `socket` module). TCP was chosen over UDP because the game uses an authoritative server model: every client input must be delivered reliably and in order.

Message framing. TCP is a byte stream, not a message stream: a single `recv()` may return several concatenated commands, or only part of one. All application messages are therefore newline-delimited, and both endpoints accumulate incoming bytes in a buffer which is split on `\n` rather than being treated as one message per `recv()`. The buffer is capped at 64 KB, so that a peer which never terminates a frame has its connection dropped instead of growing the buffer without bound.

Framing interacts directly with failover. When the proxy reconnects to the newly promoted primary it writes the `RECONNECT:{session_id}` handshake immediately followed by every command buffered during the outage, frequently within the same TCP segment. The server therefore preserves whatever follows the handshake line and uses it to seed the client handler's framing buffer, so that commands issued while the primary was down are executed rather than silently discarded.

Data representation. All messages are serialised as **JSON**, one object per line (`\n`-delimited). JSON was chosen for its human-readability (useful for debugging) and for native Python support via the `json` standard library module. The state snapshot uses a versioned schema (`"version": 1`) to detect incompatible snapshots during replication.

Concurrency model. The server uses **one thread per client connection** plus a dedicated game-loop thread and fault-tolerance threads (heartbeat responder, replication sender). Access to the shared `GameService` is protected by a single `threading.RLock`.

MVC on the client. The client is structured following the **Model-View-Controller** pattern:

- **Model** (`GameState`): holds the last received state snapshot.
- **View** (`LobbyView`, `GameView`, `VictoryView`, ...): renders the current screen using Pygame.
- **Controller** (`GameController`): translates keyboard input into server commands.
- **Network** (`NetworkManager`): handles the TCP connection and message framing in a background thread.

The MVC separation means that the client's rendering logic is completely decoupled from the network layer. When the `NetworkManager` thread receives a new state snapshot, it updates the `GameState` model; the Pygame main loop then re-renders whatever the model currently holds. This design also simplifies the failover scenario on the client side: since the client has no concept of "which server" it is connected to, a reconnection requires no change to the model, the views or the controller. Only the transport is replaced — `BombermanClient.reconnect()` swaps in a fresh `NetworkManager`, carries the previous `session_id` over to it and replays the `RECONNECT` handshake, leaving the Pygame window and the rendering loop untouched.

Proxy implementation. The proxy spawns one thread per client connection. Each thread manages both directions of the relay — client-to-backend and backend-to-client — using `select()` to multiplex the two sockets without blocking. During failover, the thread detects the backend drop, accumulates incoming client data in a local buffer (`client_buf`), and retries the backend connection for up to 30 s. Once reconnected, it sends the `RECONNECT:{session_id}` handshake and flushes the buffer before resuming normal forwarding.

AutoSpawner and process management. The `AutoSpawner` uses Python's `subprocess` module to launch the backup server as a child process. When the backup is promoted to primary, it invokes `AutoSpawner` again to spawn a new backup, maintaining fault tolerance continuously. If the backup process terminates unexpectedly, a background monitor thread detects the failure and automatically respawns a new backup, so the replication channel is restored without operator intervention.

5 Validation

5.1 Automatic Testing

Validation was performed through a combination of automated unit testing and manual end-to-end testing. The automated test suite consists of 100 tests across five modules, located in `src/test/` and executable with:

```
poetry run pytest src/test
```

Tests use Python's `unittest` framework. Where network or Pygame dependencies would be required, components are isolated using `unittest.mock` (`Mock`, `patch`), allowing each layer to be tested independently.

test_core.py – 18 unit tests for pure game logic. Tests cover: safe zone computation, player spawn points, connected player count, spectator join eligibility, chat message handling, map generation (boundary walls, safe spawn areas), tile walkability, player presence checks, movement, bomb placement and explosion propagation, victory detection (single winner and draw), block regeneration safety, and lobby reset. All tests are deterministic and require no network or Pygame initialisation.

One test is a regression test for spectator-slot eligibility with a disconnected player: that branch referenced a field which had been removed from the `Player` dataclass, and since no existing test exercised it the resulting `AttributeError` — reachable from inside the game loop — went unnoticed until a manual code review.

test_game.py – 31 client-side unit tests. Tests cover the `GameState` model (initial state, phase transitions on lobby/playing/victory snapshots, player and spectator data extraction) and the `GameController` (key bindings for movement, bomb placement, chat activation, spectator join request, host start command). Network dependencies are replaced with `Mock` objects, isolating the controller from the TCP layer.

test_controller.py – 28 command and network unit tests. Tests cover the server-side `CommandController` (movement, bomb, game start, spectator join, chat, case-insensitive command parsing, host-only enforcement) and the client-side `NetworkManager` (socket send, receive loop, connection error handling, thread lifecycle, JSON parsing of incoming messages). Socket objects are mocked via `Mock(spec=socket.socket)` and thread creation is patched with `@patch('threading.Thread')`.

test_game_service.py – 19 service layer tests. Tests cover the full player and spectator lifecycle (join, disconnect, removal, host reassignment on disconnect), game start and return-to-lobby transitions, and the game tick (bomb timer decrement, detonation, explosion cleanup, life reduction, victory detection under both single-winner and draw conditions).

test_state_codec.py – 4 serialisation round-trip tests. Tests verify that a fully populated `State` object (including players with all fields, bombs, explosions, spectators, and chat history) survives a complete `state_to_dict` / `state_from_dict` round-trip without data loss. These tests directly validate the correctness of the replication payload sent from primary to backup.

5.2 Acceptance Test

Manual end-to-end testing was performed to validate the fault-tolerance behaviour:

1. Start server, proxy, and two clients as described in the Deployment section.
2. Start a match; verify real-time state synchronisation across clients.
3. Identify the primary server PID via `netstat -ano | findstr 5556 | findstr LISTENING` (Windows) or `ss -tlnp | grep 5556` (Linux/macOS).
4. Kill the primary with `taskkill /PID {pid} /F` (Windows) or `kill {pid}` (Linux/macOS).
5. Observe: the backup promotes itself (new PID appears on port 5556), clients remain connected, and the match continues without returning to the lobby.

6. Repeat the kill on the newly promoted primary to verify that a second failover completes correctly.

Manual testing was necessary for the failover scenario because it requires OS-level process termination and real TCP connections, which are difficult to reproduce reliably in a unit test environment.

6 Deployment

The source code is available at:

`https://github.com/Saint138/Distributed-Bomberman`

6.1 Prerequisites

- Python 3.10 or higher.
- Poetry (install via `pip install poetry` if not available).

6.2 Installation

From the project root (`Distributed-Bomberman/`):

```
poetry install
```

This creates an isolated virtual environment and installs `pygame` and `pytest`.

6.3 Running the system

Start each component in a separate terminal, in this order:

```
# Terminal 1 - primary server (auto-spawns the backup)
poetry run bomberman-server

# Terminal 2 - proxy
poetry run bomberman-proxy

# Terminal 3..N - one per player
poetry run bomberman-client
```

A match requires at least two connected clients to start.

6.4 Port configuration

All ports are defined in `src/common/constants.py`:

Port	Purpose
5555	Proxy frontend (clients connect here)
5556	Primary game server
5557	Backup state-receiver (replication)
5558	Reserved (`BACKUP_GAME_PORT`); not bound by the current implementation
5565	Primary heartbeat responder

7 User Guide

Joining a game. Launch a client (`poetry run bomberman-client`). The client connects automatically to `localhost:5555`. A random name is assigned. If a slot is available, the player enters the lobby; otherwise they enter as a spectator.

Lobby. The lobby screen shows all connected players and spectators. The host (the first player to join) sees a "Press ENTER" prompt. The game starts when the host presses Enter with at least two players connected. The interface is driven entirely from the keyboard: the client handles no mouse input at all. A spectator can press **J** to request a player slot; the server converts them immediately if one is available.

Figure 9: The lobby screen with two connected players. The host is highlighted in green.

Gameplay. Controls: **arrow keys** to move, **Space** to place a bomb. Bombs explode after a fixed timer, destroying breakable blocks and damaging players within range. A player loses one life per explosion hit; after three hits the player is eliminated. Destroyed blocks are periodically regenerated on tiles that are safe to occupy, so the map does not grow progressively emptier over a long match.

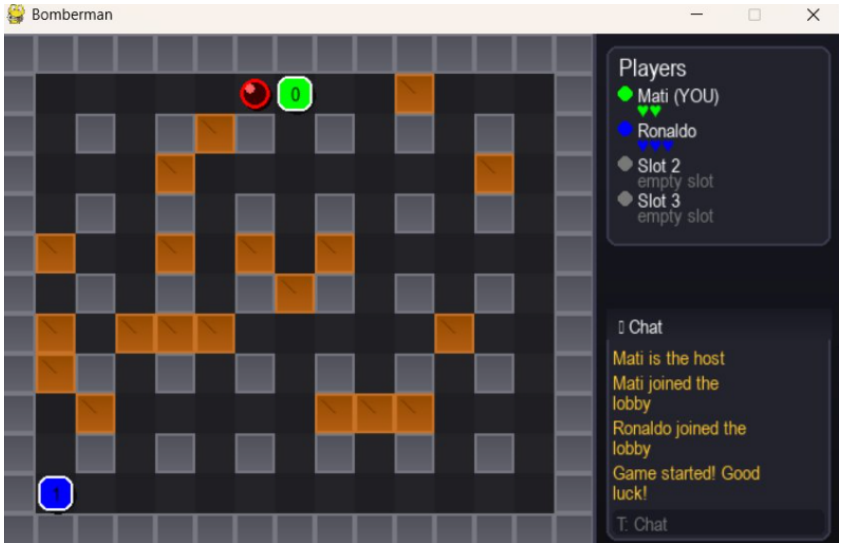

Figure 10: The game screen during an active match. Players, breakable blocks, and the chat sidebar are visible.

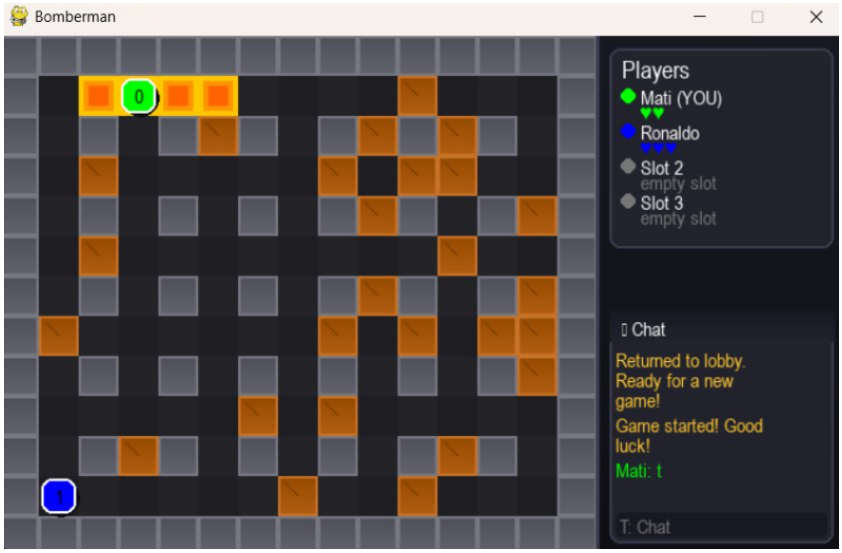

Figure 11: A bomb explosion in progress, showing the blast radius across adjacent tiles.

Chat. Press **T** to open the chat input, **Enter** to send, **Escape** to cancel. Messages are broadcast to all connected players and spectators.

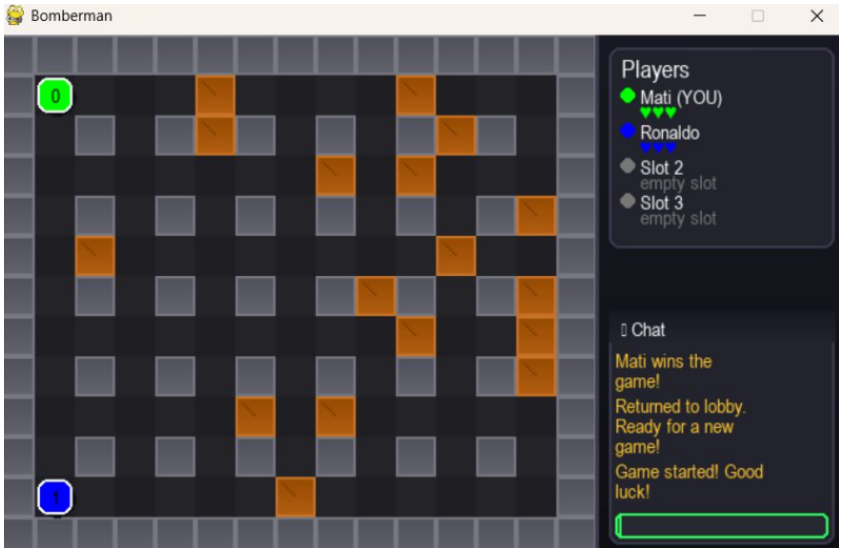

Figure 12: The in-game chat input active during a match.

End of match. The last surviving player wins; if all players die simultaneously, the match is a draw. The victory screen displays the winner and a countdown before returning all clients to the lobby automatically; a player can also press Enter to request a rematch immediately (`PLAY_AGAIN`).

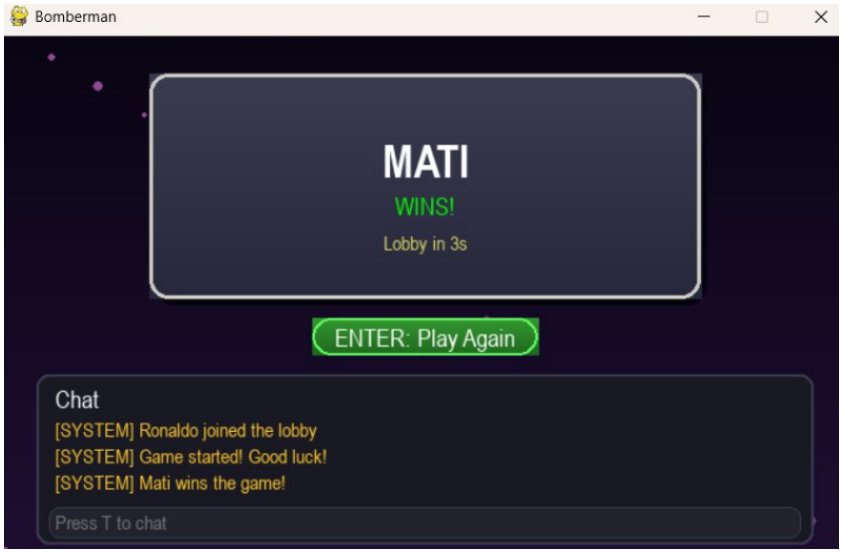

Figure 13: The victory screen showing the winner and the return-to-lobby countdown.

8 Self-evaluation

8.1 Giovanni Rinchiuso

My contributions span three areas: the **fault-tolerance infrastructure**, the **transparency layer**, and the **client** with its test suite.

For fault tolerance, I designed and implemented the full primary/backup stack from scratch: `FailureDetector` (heartbeat-based failure detection with a 1.5 s timeout), `BackupServer` (state receiver, heartbeat monitor, promotion logic), `PrimaryServer` (heartbeat responder, state replication sender), and `AutoSpawner` (subprocess-based backup lifecycle management with automatic respawn on unexpected termination). The heartbeat timeout and replication interval (100 ms) were tuned through manual testing to balance detection speed against false positives under normal load.

The transparency layer is the TCP proxy (`fault_tolerance/proxy_server.py`), which hides the primary/backup topology from clients entirely. The proxy intercepts the server's `join_success` message to extract the `session_id`, buffers outgoing client messages in `client_buf` during failover, and sends a `RECONNECT` handshake to the new primary once it comes up. A known weakness is that the proxy is itself a single point of failure: a crash severs all client connections even if the server cluster is healthy. This is a deliberate trade-off — unlike the primary, the proxy holds no game state, so a crash does not lose match data — and its impact is bounded: I also implemented a client-side retry loop that re-establishes the connection every two seconds and replays the `RECONNECT` handshake, so a restarted proxy restores each player's session instead of forcing a fresh join. It does not preserve a match in progress, for the reasons discussed in section 3.8.

On the client side, I built the entire Pygame application: the MVC structure (`GameController`, `GameState`, views for lobby, game, and victory), the `NetworkManager` (TCP connection, line-delimited JSON framing, background receive thread), and all graphical rendering. I also wrote the bulk of the automated test suite: `test_core.py`, `test_game.py` (31 tests), `test_controller.py` (28 tests), and `test_game_service.py` (19 tests). A limitation of the current replication design, discussed in section 3.4, is opening a new TCP connection for every snapshot rather than maintaining a persistent channel — functional at the current scale, but worth revisiting if the replication interval were reduced or the state size increased significantly.

8.2 Gioele Santi

My contributions are the **server game logic**, the **consistency layer**, and the **hardening of the replication path**.

I built the authoritative game engine: map generation with destructible blocks and safe spawn zones, bomb placement and explosion propagation (a cross-shaped blast of radius 2, stopped by walls and consuming the first destructible block on each arm), player lifecycle management, the lobby/playing/victory state machine, timed regeneration of destroyed blocks, and the spectator system. It was later refactored so that the pure game rules in `core.py` carry no networking or threading concern and can be unit-tested in isolation.

For consistency I implemented the `GameService` layer — the single component authorised to mutate game state — protecting every public method with a `threading.RLock` via a `@_synchronized` decorator. Previously the shared `State` was mutated by the game-loop thread, the accept thread and every client handler thread with no synchronisation at all. I chose one coarse-grained lock over per-collection locks deliberately: at 10 Hz the contention is negligible, while every public mutation becomes trivially atomic and no deadlock from inconsistent lock ordering is possible. The lock must be re-entrant because some service methods invoke others.

The replication path originally serialised state with `pickle`; since `pickle.loads` executes arbitrary code during deserialisation, any peer reaching the backup's state port could have achieved remote code execution. I replaced it with a versioned JSON codec that validates the schema version and rejects malformed snapshots instead of crashing the receiver, and covered it with round-trip tests, since a single dropped field would silently corrupt the backup at promotion time. I also replaced the session identifier generator — roughly ninety thousand brute-forceable values, enough for an attacker on the same LAN to hijack a disconnected player's session — with UUID4.

Finally I removed around 350 lines of dead code, which exposed a latent `AttributeError` reachable from inside the game loop that no test covered; I fixed it and added the missing regression test.

The main weakness in my part is that reconnection is narrower than it could be: a client whose process exits cannot rejoin, because the `session_id` lives only in the client's memory and there is no server-side grace timer bounding how long a disconnected player is retained. A transient socket drop is recovered while enough players remain alive, but `handle_player_disconnect` evaluates the victory condition immediately, so in a two-player match a single drop already ends the round. Generalising this would require persisting the session identifier client-side and driving an explicit expiry timer from the game loop, so that a brief absence suspends the victory check instead of resolving it.

9 Future Works

- Add collectable power-ups (increased bomb range, extra bomb, speed boost) to enrich gameplay.
- Support multiple simultaneous game rooms on the same server cluster.
- **Authentication**: add a simple username/password login to allow persistent scores across sessions.
- Automate the failover acceptance test using subprocess management and virtual network interfaces, removing the current need for manual steps.