

Final Report Template

Distributed SMB

Final Report for the Distributed Systems Course A.Y. 2025-2026

Author 1

ludovico.nollino@studio.unibo.it

Author 2

mattia.flamigni@studio.unibo.it

Author 3

filippo.badioli@studio.unibo.it

September 25, 2026

This report presents *Distributed SMB*, a cooperative real-time 2D platform game inspired by Super Mario Bros, developed to explore the fundamental principles of Distributed Systems within an interactive environment. Up to four players, each on their own machine connected to a local network, take part in the same session and share a single game world, cooperating to complete a level. The game mechanics are intentionally kept simple, so that the difficulty of the project lies in the coordination among the nodes rather than in the game itself.

From an architectural perspective, the system adopts a hybrid peer-to-peer model with a dynamically assigned authority. One node acts as the authoritative owner of the shared state and resolves the actions that affect it, while the other nodes anticipate the effect of their own actions locally in order to preserve responsiveness. In CAP terms this design is AP-oriented: it favours availability and low latency over immediate consistency, and relies on the convergence of the replicas over time.

The project places a strong emphasis on fault tolerance and availability. Since the authority resides on one of the participating machines, it constitutes a single point of failure: the system therefore detects its failure, elects a successor among the surviving nodes without resorting to any central coordinator, and allows the session to continue, while a node that has failed can later rejoin the session in progress.

This report describes the requirements of the system, the architectural decisions and the consistency model adopted, the implementation and deployment choices, the validation carried out, and the limitations that remain open.

Disclaimer

During the preparation of this work, the authors used LLMs (Claude, ChatGPT) to validate the architectural ideas and design decisions, write and review the codebase and improve the clarity of this document. After using this tool/service, the authors reviewed and edited the content as needed and takes full responsibility for the content of the final report/artifact.

Contents

1	Concept	5
1.1	Product Type	5
1.2	Use Case Description	5
1.3	Need for Distribution	5
2	Requirements Elicitation and Analysis	7
2.1	Requirements	7
2.2	Relevant Distributed System Features	10
2.3	Glossary	11
3	Design	13
3.1	Architecture	13
3.2	Infrastructure	15
3.3	Modelling	17
3.3.1	Domain Entities	17
3.3.2	Mapping to Infrastructural Components	18
3.3.3	Domain Events	19
3.3.4	Message Types	20
3.3.5	System State	22
3.4	Interaction	23
3.4.1	Communication Flows	23
3.4.2	Interaction Patterns	24
3.5	Behaviour	29
3.6	Data and Consistency Issues	31
3.7	Fault-Tolerance	32
3.8	Availability	34
3.9	Security	35
4	Implementation	36
4.1	Network Protocols	36
4.1.1	UDP for real-time gameplay traffic	36
4.1.2	WebSocket over TCP for coordination and critical events	36
4.1.3	HTTP for publishing events to the Game Events service	37
4.1.4	UDP broadcast for session discovery	37
4.1.5	Why not a single transport	37
4.2	Data Representation	37
4.2.1	Serialization format	37
4.2.2	Message contracts and validation	38
4.3	Data Storage and Querying	38
4.4	Authentication and Authorization	39
4.4.1	Authentication	39
4.4.2	Authorization	39

4.5	Technological Details	39
4.5.1	Game client	39
4.5.2	Concurrency model	40
4.5.3	Fixed simulation step	40
4.5.4	Timeouts and timers	41
4.5.5	Infrastructure services	41
4.5.6	Simulating adverse network conditions	42
4.5.7	Development tooling	42
5	Validation	43
5.1	Automatic Testing	43
5.2	Test Execution	45
5.3	Validation Scope and Limitations	45
6	Deployment	46
6.1	Deployment architecture	46
6.2	Prerequisites	47
6.3	Distribution package	47
6.4	Installation and execution	48
6.5	Containerization	48
6.6	Configuration and networking	50
6.7	Build and release automation	50
7	User Guide	52
7.1	Requirements and Installation	52
7.2	Starting the Application	52
7.3	Creating a Game Session	53
7.4	Joining a Game Session	53
7.5	Gameplay Controls	55
7.6	Expected Gameplay Behaviour	55
7.7	Host Failure and Session Recovery	55
7.8	Troubleshooting	56
8	Self-evaluation	58
8.1	Ludovico Nollino	58
8.2	Mattia Flamigni	58
8.3	Filippo Badioli	59
9	Future works	61
9.1	Known limits of the fault-tolerance protocol	61
9.2	Engineering improvements	61
9.3	Beyond the current scope	62

1 Concept

1.1 Product Type

The product is a real-time multiplayer desktop game with a graphical user interface: a cooperative 2D platformer inspired by Super Mario Bros. The game application is identical on every machine and runs on the players' own computers, connected to the same Local Area Network (LAN). It is accompanied by two lightweight coordination services, a lobby and a relay for game events, packaged as Docker containers and started automatically by the node acting as host. No dedicated server and no manual network configuration are required.

1.2 Use Case Description

Up to four players take part in the same game session. One player creates a room and obtains a session identifier; the others join the room by entering that identifier, and the host starts the match when everyone is ready. Each player controls a different character in a single shared level, and the team cooperates to reach its end: collecting items, breaking blocks and defeating enemies increases a shared progress, which unlocks the gates along the way. Once the level is completed, the players return to the lobby and can start a new run.

Players interact through the keyboard, sending movement commands several times per second, and observe the shared world through the game window. Some of their actions, such as breaking a block or collecting a power-up, change the world for everybody and must be propagated to all participants.

The system is ephemeral: there are no accounts, statistics or history. The state of a session exists only in memory for the duration of the match; the only exception is a small local cache that allows a crashed player to find the session again.

The system defines a single user role, the *Player*. At runtime, however, one player's node acts as the *Host*, which holds the authoritative game state, while the others act as *Clients*. This role is not fixed: it moves to another node when the host fails.

1.3 Need for Distribution

Distribution is an inherent requirement of the product rather than a design choice:

- **Distributed environment.** Players sit at different machines but must share one world in real time. Nodes share no memory and communicate only by exchanging messages over a network that can delay, lose or reorder them, with no global clock.
- **Shared state and coordination.** The world contains shared elements and exclusive resources that several players may try to use at the same moment. Conflicting actions must be resolved in the same way for everybody, which calls for a single authority. At the same time, each player expects their own character to respond immediately. The system must therefore balance a coherent view of the shared

world with responsiveness and availability, which is the central trade-off of the design.

- **Fault tolerance.** The authority runs on one of the players' machines, which makes it a single point of failure. The main goal of the project is that the session survives its loss: the remaining nodes must detect the failure, agree on a new host without any central coordinator and continue the match, and the failed node must be able to rejoin.
- **Limited scalability.** Scalability is explicitly not a goal: a session involves at most four players on a LAN, so the system favours simplicity and low latency over horizontal scaling. Several sessions may coexist on the same network, each hosted by a different machine and reachable through its own identifier, but no single session is expected to grow beyond its four participants.

These forces shape the core of the design. One node at a time holds the authoritative state and decides the outcome of every action affecting the shared world, while the others simulate their own character locally and correct it when the authoritative state arrives. In the terms of the CAP theorem the system is AP-oriented: since the network may be partitioned and nodes may fail, it cannot guarantee both consistency and availability, and it favours availability. Players keep playing when messages are delayed or lost, and a new host is elected when the current one fails, at the price of temporary inconsistencies. Consistency is pursued in a weaker form: because a single node decides, conflicting actions are resolved in the same way for everybody, and the copies held by the other nodes converge to that decision over time (eventual consistency).

2 Requirements Elicitation and Analysis

2.1 Requirements

- **Functional requirements**

FR-1 – Session creation and participation

The system should allow players to create new private game sessions and obtain a unique session identifier. Other players should be able to join an existing session by providing its identifier, up to a maximum of four participants.

Acceptance criterion: 2-4 players can successfully join the same game session using a shared identifier; any further join attempt is explicitly rejected.

FR-2 – Real-time multiplayer gameplay

The system should allow up to four players to participate simultaneously in the same game session, sharing a common game world and progression.

Acceptance criterion: Multiple players can move their characters concurrently and observe each other's movements in real time.

FR-3 – Game state synchronization

The system should continuously synchronize relevant game state information among all participants, including player positions and changes to the shared world.

Acceptance criterion: Changes in player positions or world state performed by one player become visible to all other players within a short and bounded time interval.

FR-4 – Authoritative game state management

The system should ensure that a single participant acts as the authoritative entity responsible for validating actions that affect the shared game state.

Acceptance criterion: Conflicting actions affecting shared elements are resolved consistently according to decisions made by the authoritative entity.

FR-5 – Cooperative gate mechanism

The system should include cooperative objectives that must be satisfied collectively by all players in order to unlock progression within the level. Each objective is expressed as a set of thresholds on progress shared by the whole team.

Acceptance criterion: A gate becomes accessible only after the team has reached its thresholds and remains inaccessible otherwise.

FR-6 – Exclusive shared resources

The system should manage power-ups as exclusive shared resources: each power-up can be collected only once, by exactly one player.

Acceptance criterion: When several players reach the same power-up concurrently, only one of them obtains it, and all participants agree on which one.

FR-7 – Player disconnection handling

The system should tolerate voluntary or involuntary disconnection of players without terminating the entire game session.

Acceptance criterion: If a non-authoritative player disconnects, it is removed from the session, the other participants are notified, and the remaining players continue playing without restarting the session.

FR-8 – Host failure detection and recovery

The system should detect the failure of the authoritative host and automatically assign the host role to another participant.

Acceptance criterion: If the host becomes unreachable, a new player is promoted to host and the game continues from the last known state without restarting the session.

FR-9 – Client-side state reconciliation

The system should implement a mechanism to resolve discrepancies between the client's locally predicted state and the authoritative state received from the host.

Acceptance criterion: When a client's local state diverges from the host's authoritative state (due to latency or packet loss), the client automatically corrects its state (reconciliation) to match the host's.

FR-10 – Distributed membership management

Each participating node must maintain an up-to-date local list of active session participants (Membership List) to allow for direct peer-to-peer communication and autonomous leader election in case of host failure.

Acceptance criterion: Upon a host crash, surviving clients possess sufficient local information about peers to initiate the election process without contacting the central lobby.

FR-11 – Rejoin after failure

A player whose node has failed during a match should be able to return to the ongoing session without the intervention of the other players.

Acceptance criterion: After restarting the application, the player locates the current host, even if the host role has migrated in the meantime, and rejoins the

session as a client, receiving the current state of the game world. The rejoining node never reclaims the host role.

- **Non-functional Requirements**

NFR-1 – Low interaction latency

The system should provide responsive gameplay, ensuring that user inputs are reflected in the game world with minimal perceived delay, leveraging the low-latency characteristics of the LAN environment and UDP protocol.

Acceptance criterion: Player movement appears smooth and continuous, without significant jitter or rubber-banding effects, during normal network conditions.

NFR-2 – Availability over strong consistency

The system should prioritize availability and responsiveness over immediate strong consistency for the gameplay layer, tolerating temporary divergences between nodes that self-correct over time (Eventual Consistency).

Acceptance criterion: Temporary inconsistencies (e.g., slight positional desync) may occur, but the game remains playable and the state eventually converges.

NFR-3 – Limited scalability

The system is explicitly designed to support a small, fixed number of participants ("Session-based") and is not expected to scale to massive numbers of concurrent users per session.

Acceptance criterion: The system correctly supports sessions of up to 4 concurrent players without performance degradation.

NFR-4 – Simplicity of game mechanics

The system should keep game mechanics intentionally simple in order to focus on distributed coordination aspects.

Acceptance criterion: The game logic remains limited to a single level with basic movement and interaction mechanics.

- **Implementation Requirements**

IR-1 – Programming language constraint

The system should be implemented using Python.

Justification: Python enables rapid prototyping and offers mature libraries for every part of the system: low-level sockets, WebSocket clients and servers, message validation, and 2D game development.

IR-2 – Communication middleware constraint

The system should rely on reliable, connection-oriented WebSocket channels for coordination and control messages, including room creation, player membership updates, critical gameplay events (e.g., block destruction, gate state changes), and host election signaling. WebSockets are not used for high-frequency gameplay state exchange, which is delegated to UDP as per IR-3.

Justification: WebSockets provide a reliable, ordered, bidirectional communication channel suitable for messages that require delivery guarantees, while keeping implementation complexity low compared to manual reliability layers over UDP.

IR-3 – Real-time game state communication

The system shall rely on UDP-based communication channels for the exchange of time-sensitive game information, including player inputs sent by clients to the host and world-state updates sent by the host to each client.

Justification: UDP enables low-latency, connectionless communication and allows the system to tolerate occasional packet loss without blocking execution, which is appropriate for real-time interaction and fast-paced gameplay.

2.2 Relevant Distributed System Features

- **DR-1 – Asynchronous communication**

Messages exchanged by the system may be delayed, duplicated, or lost. The system must be designed to operate correctly despite these conditions.

- **DR-2 – Absence of a global clock**

No node has a perfectly synchronized view of time. As a consequence, events may be observed in different orders by different nodes, and each node must rely only on its own local clock and on logical ordering information carried by the messages.

- **DR-3 – Leader election**

Nodes must be able to detect the absence of the current host and autonomously agree on a new host among the remaining participants, without contacting the central Lobby.

- **DR-4 – Eventual consistency**

The global game state must eventually converge across all nodes. Immediate strong consistency is not required.

- **DR-5 – Reliable delivery of critical messages**

While player movement updates may tolerate packet loss, critical events that change the shared world irreversibly (e.g., block destruction, power-up collection,

gate opening) must be delivered reliably and in the order in which the host produced them. When a failure may have caused some of them to be lost, the nodes must be realigned with the authoritative state.

- **DR-6 – Network partition handling**

If a client becomes temporarily isolated, it must be able to continue local movement in the game world to preserve availability. However, it must not be able to change the shared state observed by the other players: only the host decides on shared objects.

- **DR-7 – Small and known membership**

The participants of a session are few and known in advance: each of them is registered before the match begins and receives an identifier that establishes a total order among them. Membership changes only through departures and rejoins, never through an unpredictable arrival of new nodes during a match. The system may exploit this property in its coordination mechanisms, instead of assuming an open group of unknown size.

- **DR-8 – Security**

No authentication or authorization is needed, and there is no sensitive data processed or stored. The only user roles are Host and Client, which are handled by the system without needing tokens or authentications.

2.3 Glossary

Term	Definition
Host	The player whose node maintains the authoritative game state during a session. The Host role is dynamic and may migrate to another node in case of failure.
Client	A non-authoritative player participating in a game session.
Session	A single instance of a multiplayer game, identified by a unique Session ID, with a lifecycle spanning from room creation until the host closes the room. A session may include several runs of the level.
Lobby Service	A centralized coordination service used for session creation, membership and roster distribution before the match. It is not involved in the gameplay loop.

Term	Definition
Game Events Relay	A stateless service that forwards the critical events published by the Host to every node of the session. It runs next to the Lobby and is restarted by a newly elected Host.
Session Discovery	The mechanism by which a joining player resolves a Session ID into the address of the Host, through a broadcast query on the LAN.
Host Migration	The process by which the Host role is transferred from a failed node to a surviving peer, allowing the session to continue without interruption.
GlobalRoster	The local copy, maintained by every peer, of the full list of session participants and their network endpoints. Used to enable autonomous leader election without contacting the Lobby.
JoinIndex	A monotonically increasing integer assigned by the Lobby to each player upon joining (0 for the player who created the session), used to deterministically order candidates during leader election.
WorldStateSnapshot	The authoritative serialized representation of the game simulation at a given tick, sent by the Host to every Client via UDP.
Client-Side Prediction	A technique by which a Client simulates the effect of its own inputs locally before receiving confirmation from the Host, in order to reduce perceived latency.
Reconciliation	The process by which a Client corrects its locally predicted state to match the authoritative state received from the Host.
Shadow Copy	A visual replica of a remote player maintained by the Client, updated via interpolation from received snapshots and used only for rendering.

3 Design

3.1 Architecture

The architecture of the system is defined by two independent decisions: how the nodes are arranged with respect to one another, and how each node is organised internally.

At the level of the distributed system, the style is a **hybrid peer-to-peer architecture with a migrating authority**. One node at a time, the host, owns the authoritative state and arbitrates every action that affects the shared world. The architecture is *hybrid* because the traffic is split in two: the high-frequency state exchanged between the host and each client travels peer-to-peer, while the discrete events that must not be lost are published through a relay which forwards them to every node. Two minimal services support the session from outside the gameplay loop, the Lobby and that relay, and the authority itself is not bound to a machine: when the host fails, the surviving nodes elect a successor and the session continues.

This style suits the system for four reasons:

- **Conflicting actions receive one answer:** Two players can reach the same power-up in the same tick, each seeing itself arrive first. One node decides, and its decision is the only one published (**FR-4**, **FR-6**).
- **Each kind of traffic pays only for what it needs:** A lost state update costs nothing, since the next one supersedes it, so it takes the direct path (**NFR-1**). A lost event leaves a node permanently out of step, so it takes the reliable one (**DR-5**).
- **Responsiveness does not depend on the round trip.** A client applies its own input at once and corrects itself when the authoritative state arrives (**FR-9**). It still arbitrates nothing, so the rules of the shared world exist in one implementation only.
- **The failure of the arbiter is not the failure of the session.** The authority is a role assigned at runtime, and every node already holds the state needed to assume it: losing the host costs an election, not the match (**FR-8**). This is the property the whole project is built around.

Two alternatives were considered and set aside. A dedicated server would have been the natural home for the authority, but it presumes a machine that a group of players on a LAN may not have, and it would have removed precisely the problem the project set out to study. A pure peer-to-peer scheme with no authority, in which every node simulates and the nodes agree on each tick, requires an agreement protocol per frame: the pace of the simulation would be dictated by the slowest node, and a single stall would block everybody, which is unacceptable for a platform game. (**NFR-1**, **NFR-2**).

Inside a node, the style is a **layered architecture**, and the communication between nodes is the only event-based part of the system. The execution inside a node is imperative: a frame loop polls its inputs once per frame, advances the simulation and

renders the result. The events are exchanged between nodes, in the form of player inputs and world-state snapshots over UDP, critical game events published on a relay, and the messages of the host election.

Each node is organised in four layers plus a cross-cutting package, with dependencies allowed in one direction only, as shown in fig. 1.

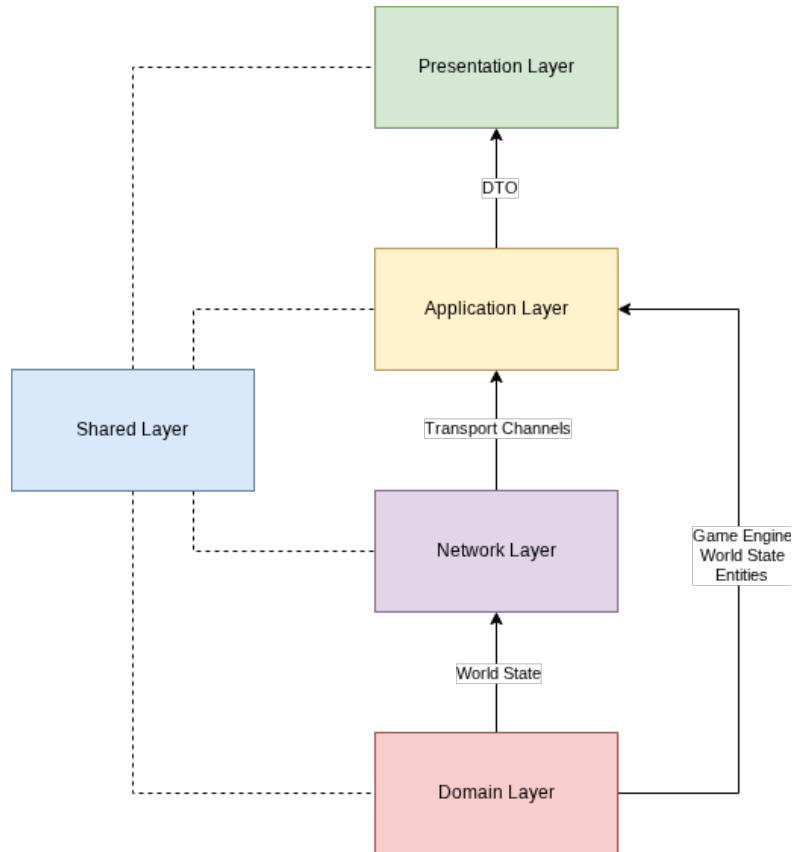


Figure 1: Layered architecture of a node. Each layer may depend only on the layers below it; **shared** is cross-cutting and depends on none.

- **Domain:** the deterministic simulation of the game: the engine, the world state, the entities, physics, collisions and level loading.
- **Network:** everything that crosses a machine boundary: the UDP transport, the WebSocket client, message serialization and validation, the Lobby service, the Game Events relay and LAN discovery.
- **Application:** the node itself: the roster, the current role, the host and client gameplay loops, client-side prediction and reconciliation, the interpolation of remote players, failure detection, election and recovery. It also builds the frame that the presentation draws.

- **Presentation:** the frame loop, the renderer, the menu and lobby screens and keyboard input. It depends only on the application layer, and only through a render-frame data transfer object: it never sees a domain entity.
- **Shared:** configuration, message definitions, roster and enumerations. Every layer uses it, and it depends on none.

The layering is not independent of the distributed style: it exists to support it. Because the domain layer is deterministic and does not know whether it is authoritative, the same simulation runs everywhere, which is the precondition of reconciliation (**FR-9**), and any node can assume the authority without swapping components, so a promotion is a change of state inside one layer and not a redeployment (**FR-8**).

3.2 Infrastructure

The infrastructure introduces no database and no external message broker: the game state is transient, and an additional intermediary would only add network hops. A session involves three kinds of runtime components.

- **Game Nodes** The desktop application, identical on every machine. At runtime a node takes one of two roles:
 - **Host:** runs the authoritative simulation and is the only node that decides the outcome of actions affecting the shared world (**FR-4**): collected power-ups (**FR-6**), defeated enemies, deaths and respawns, gate states (**FR-5**) and victory. The host also plays, so its own player experiences no network latency.
 - **Clients:** simulate their own player locally, send their inputs to the host, reconcile against the authoritative snapshots and interpolate the remote players (**FR-2**, **FR-9**).
- **Lobby Service.** An in-memory registry of sessions, reached over WebSocket, used to create a session, join it, leave it and start the match (**FR-1**). It assigns each participant its JoinIndex, enforces the limit of four participants and broadcasts the roster to every node (**FR-10**). It plays no part in the gameplay loop: once the match has started, the session can proceed without it.
- **Game Events Relay.** A stateless service that receives the critical events published by the host and forwards them over WebSocket to every node of the session (**DR-5**). Being a pure relay, it holds nothing that would have to be preserved across a failure. The messages of the host election travel through it as well.

The infrastructure explicitly excludes persistent storage and intermediate brokers: all data is held in memory for the duration of a session, and a broker between peers would contradict the latency requirement (**NFR-1**) by adding a hop to every exchange.

The two services run on the machine of the current host and are started by the game application itself when a room is created, so that the system remains self-contained: a

group of players needs no infrastructure beyond their own computers. On that machine, therefore, three components coexist: the authoritative game logic, the presentation of the host player, and the two services, which the host reaches through the loopback interface while the other nodes reach them over the LAN. The host player consequently interacts with the authoritative state without network latency, whereas the clients are always one round trip behind it. Client-side prediction exists to hide this asymmetry (**NFR-1**).

The services are deliberately kept in separate processes rather than embedded in the game application. A crash of the game process does not necessarily destroy them, and a newly elected host either restarts them on its own machine or reuses the ones already running.

Addresses are assigned dynamically by the local DHCP server, so the system cannot rely on static configuration. It identifies sessions and players by logical names rather than by physical locations: a session identifier generated at creation time, and a player identifier derived from the JoinIndex assigned by the Lobby. A joining player therefore knows only the session identifier, which the host shares out of band.

The resolution from a logical name to an address is performed by a **broadcast query on the LAN**: the joining node asks which node hosts a given session identifier, and the host of that session answers with its address.

Once a node is part of a session, it no longer needs the discovery mechanism: the roster distributed by the Lobby gives it the address of every participant, and that is the information it later needs to take part in an election without any central component (**FR-10, DR-3**).

Finally, the peer-to-peer link used during the match is **hybrid** (fig. 2): an unreliable UDP channel for the real-time state (**IR-3**), and a reliable WebSocket channel, through the relay, for the events that must not be lost (**IR-2, DR-5**). The messages of the host election are sent on both channels at once, for a reason specific to fault tolerance. The relay usually runs on the machine of the host that has just failed, so an election relying on it alone would be unable to proceed exactly when it is needed. Sending the same message over a reliable but possibly dead path and over an unreliable but direct one makes the procedure independent of the failure of either.

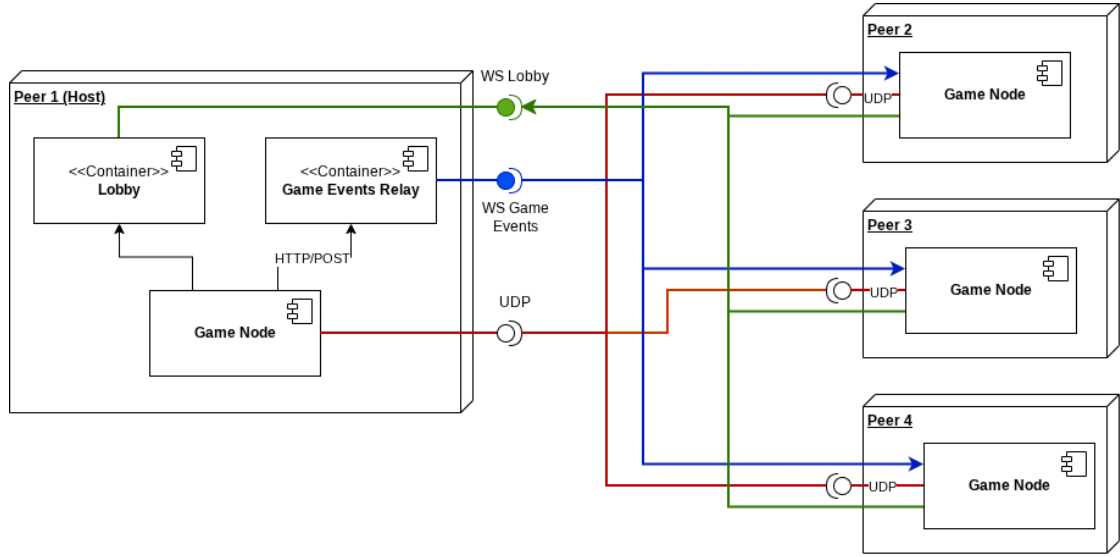


Figure 2: Runtime components of a session. The Lobby and the Game Events relay run as containers on the machine of the current host, which also runs a game node like everybody else. Gameplay state travels directly over UDP between the host and each client, session control reaches the Lobby, and critical events are forwarded by the relay to every node.

3.3 Modelling

3.3.1 Domain Entities

The model separates *who takes part* in a session from *what is simulated* in the world.

- **Session entities**

- **Session:** the record kept by the Lobby, holding the session identifier, the participants, whether the match has started, and the next JoinIndex to assign. It admits at most four participants (**FR-1**).
- **RosterEntry:** one participant, with player identifier, address, UDP port, JoinIndex, connection status and a flag marking the current host. The **GlobalRoster** is the ordered collection of entries, replicated on every node (**FR-10**).
- The **role** of a node, Host or Client, is not a separate class but an attribute of the running node, because it is reassigned at runtime by an election (**FR-8**).

- **Gameplay entities**

- **Player:** the character controlled by a participant, with position, velocity, size, ground and crouch flags, JoinIndex and the expiry of an active power-up effect.

- **Enemy:** an opponent patrolling between two bounds, defeated when a player lands on it.
 - **DestructibleBlock:** an obstacle broken by hitting it from below; once destroyed it stays destroyed.
 - **ExclusivePowerUp:** a collectible that can be obtained only once and by a single player (**FR-6**). Collecting one increases a counter shared by the team; a star additionally grants a temporary effect to its collector.
 - **CooperativeGate:** a gate that opens when the shared counters reach its own thresholds (**FR-5**). The level contains an intermediate gate, which only unblocks the path, and a final one, which completes the level.
 - **Level, Platform, SpawnPoint:** the static part of the world, identical on every node because it is loaded from the same map file.
- **State and transfer objects**
 - **WorldState:** the whole simulation at a given tick: sequence number, players, environment (blocks, power-ups, gates, enemies), shared counters, victory flag and pending respawns.
 - **WorldStateSnapshot:** a sequence number and a serialized **WorldState**, sent by the host to each client.
 - **PlayerInputPacket:** player identifier, input sequence number and the keys currently pressed.

Two modelling decisions need to be stated here:

- **No message carries a timestamp:** ordering is expressed by monotonically increasing sequence numbers. This follows directly from the absence of a global clock (**DR-2**): the sequence numbers of a single authoritative source are already totally ordered. Time is used only locally, by each node, to measure its own timeouts.
- Animations, sprites and interpolated positions are derived by each node from the state it holds; they never travel on the network and never take part in the authoritative state.

3.3.2 Mapping to Infrastructural Components

The entities of the model do not live in the same place on every node: what distinguishes a host from a client is which copy of them is authoritative.

- **Gameplay entities (players, enemies, blocks, power-ups, gates).** The authoritative instances live on the host and are advanced by its engine at every tick. Each client holds a full copy of the same world, but a predicted one: it is corrected whenever a snapshot arrives. The difference between the two is a single

flag on the engine, which decides whether the functions that resolve collections, enemy collisions, deaths, respawns and victory execute or return immediately. This flag is the concrete realisation of the authoritative model required by **FR-4**, and it is the only behavioural difference between the simulation of a host and that of a client.

- **Session entities (Session, roster).** The session record belongs to the Lobby, which creates it and assigns the JoinIndex values. The roster is replicated on every node, because each of them must be able to elect a successor without asking anybody (**FR-10**).
- **Transfer objects (snapshots, input packets).** They belong to no node: they exist only for the time needed to serialize, send, receive and apply them, and are discarded immediately afterwards.
- **Visual state.** It exists only on the node that draws it, is derived from the state that node holds, and never leaves it.

Latency compensation for the local and the remote players. On a client, no entity is drawn where the host says it is. The local player is drawn ahead of the authoritative state, because the client applies its own inputs without waiting for confirmation (**FR-9**, **NFR-1**). The remote players are drawn behind it, because their positions arrive in discrete snapshots and are smoothed by interpolating between the last two, extrapolating briefly when the next one is late; this second mechanism is the shadow copy, a visual replica of a remote player maintained by the client.

Both mechanisms belong to the application layer, and neither to the presentation one, because both compensate for properties of the network and not of the display: they reason on sequence numbers, on missing and out-of-order snapshots, and on the time elapsed since the last one arrived. The presentation layer receives only their result.

3.3.3 Domain Events

Gameplay events. Produced by the authoritative engine during a tick and collected in a list that the application layer drains at the end of the frame: **BlockDestroyed**, **PowerUpCollected**, **GateStateChanged**, **PlayerDeath** and **LevelReset**. Each of them is mapped to a message and published on the relay; every client applies it to its own copy of the world, which keeps the shared world consistent across the session (**FR-3**, **FR-4**). Only the host produces them: a client collects the events of its own predicted tick and discards them, so that no client can ever change the world of another.

Session lifecycle events.

- **SessionCreated:** a player creates a room and becomes the initial host, with JoinIndex 0.

- **PlayerJoined / PlayerLeft:** a participant enters or leaves. The roster is updated and redistributed to every node, so that each of them keeps the topology needed to elect a successor autonomously (**FR-7**, **FR-10**). A join beyond the fourth participant is refused explicitly.
- **GameStarted:** every node leaves the lobby and builds the initial world locally, from the roster and the shared level file.
- **LevelCompleted:** the victory condition is met. After a short overlay, every node returns to the lobby of the same session, and the host may start a new run.
- **SessionClosed:** the host closes the room; the clients return to the main menu.

Fault-tolerance events.

- **HostFailureSuspected:** a client has received no snapshot for the detection timeout. This is a *suspicion*, not a conclusion, and it does not start an election.
- **HostFailureConfirmed:** the suspected host did not answer a direct probe within the verification window, and is therefore declared failed.
- **SelfElected:** the staggered timer of a node expired before any claim from a lower JoinIndex arrived, so that node claims the host role.
- **FollowingHost:** a claim from a lower JoinIndex arrived first, so the node abandons its own candidacy.
- **HostPromoted:** the claimer has collected the acknowledgements of the survivors, or their deadline has expired, and takes authority over the simulation.
- **ReconnectedToNewHost:** a survivor has learned the address of the new host and redirects its traffic to it.
- **NodeRejoined:** a node that failed earlier, either a client or the host that was replaced in the meantime, has located the current host and rejoined the session as a client (**FR-11**).

The separation between the first two events is a design decision, not an implementation detail: a timeout-based failure detector cannot distinguish a crashed node from a slow one, so the system treats the expiry of a timeout as evidence to be verified, not as a fact (section 3.7).

3.3.4 Message Types

Messages fall into three categories: *commands*, sent to request an action (joining a session, moving a character); *events*, which notify a change that has already been decided; and *queries*, which ask for information and expect an answer. Every message is a self-contained object carrying a type field and its own data, and on receipt it is validated

against a schema before being turned into a typed object, so that a malformed or unknown message is rejected at the boundary of the node instead of propagating into the logic (**DR-1**).

UDP messages, exchanged directly between host and clients.

Message	Kind	Direction	Purpose
PlayerInputPacket	command	client → host	the keys pressed, with a sequence number. Loss is tolerated: the next packet supersedes the previous one.
WorldStateSnapshot	event	host → each client	the authoritative world at a tick, with a sequence number. It also acts as the implicit heartbeat of the host.
HostDiscoveryProbe	query	any node → suspected or cached peers	asks whether the addressee is the host of a session. Used both to verify a suspected failure and to locate the session when rejoining.
HostIdentityResponse	answer	host → requester	the reply to a probe, carrying the address of the current host.

Session messages, exchanged with the Lobby over WebSocket. `SessionCreate` and `SessionCreated` establish a new room; `SessionJoin` is answered by `SessionJoined` or, when the session already holds four participants, by `SessionJoinRejected`; `SessionLeave` and `SessionClosed` handle departures; `RosterUpdate` distributes the membership to every participant; `GameStart` moves them all into the match; `SessionRecreate` re-registers an existing session, under its original identifier, after a host migration; `InitialStateSync` carries the current world to a node that joins a match already in progress.

Event and election messages, forwarded by the relay over WebSocket. The gameplay events listed above (`BlockDestroyed`, `PowerUpCollected`, `GateStateChanged`, `PlayerDeath`, `LevelReset`), the membership notifications produced during a match (`PlayerLeft`, `PlayerDisconnected` and a periodic `RosterUpdate`), and the three messages of the election:

- `NewHostClaim` (candidate → survivors): the candidate announces that it is taking the host role, with its address and its `JoinIndex`.
- `ElectionAck` (survivor → candidate): the survivor accepts the claim.
- `ReconnectionAck` (new host → survivors): the promotion is complete; the message carries the address and the ports to which the survivors must now send their traffic.

Choice of channel. The division follows the nature of the data rather than its importance. Inputs and snapshots are *periodic and self-superseding*: the loss of one is repaired by the next, and retransmitting it would deliver stale information late, so an unreliable datagram service is the correct choice (**IR-3, NFR-1**). Session control and gameplay events are *discrete and irreversible*: losing one leaves a node permanently out of step, so they require an ordered and reliable channel (**IR-2, DR-5**).

The election messages are sent on both channels, for the reason given in section 3.2: they are critical, so they need reliability, but the reliable path itself may be a casualty of the very failure they respond to.

3.3.5 System State

The global state of the system is partitioned into subsets with different owners, lifetimes and consistency requirements.

- **Discovery state (Lobby scope).** For each session: its participants, the next JoinIndex to assign, and whether the match has started. It is semi-static, and it is rebuilt from the participants themselves when a new host re-registers the session after a migration.
- **Topology state (replicated on every node).** The GlobalRoster and the identity of the current host. It is first received from the Lobby; during the match it is updated by the departure notifications and by the roster that the host republishes periodically on the relay, merged additively so that a node never forgets a peer it already knows. This replication is what allows an election to run without the Lobby (**FR-10, DR-3**).
- **Simulation state (authoritative on the host).** Players and enemies, transmitted in every snapshot. It is volatile: an old value is superseded by the next snapshot and never reused.
- **Environmental state (authoritative on the host, replicated on every node).** Destroyed blocks, collected power-ups, gate states, shared counters and the victory flag. These changes are irreversible, and they are propagated as events and never overwritten by snapshots, for the reason explained in section 3.6. In addition, *every* node keeps the last snapshot it received, not only the node that will eventually be promoted: this is what allows any survivor to take the host role without a state transfer, and the others to realign with it (**FR-8**).
- **Synchronization state (local, technical).** On the host, the last sequence number and the arrival time of the inputs of each client, used to discard duplicates and stale commands. On a client, the bounded history of its unacknowledged inputs, the sequence number of the last snapshot applied, and the calibration of how far ahead of the host it is predicting (**FR-9**).

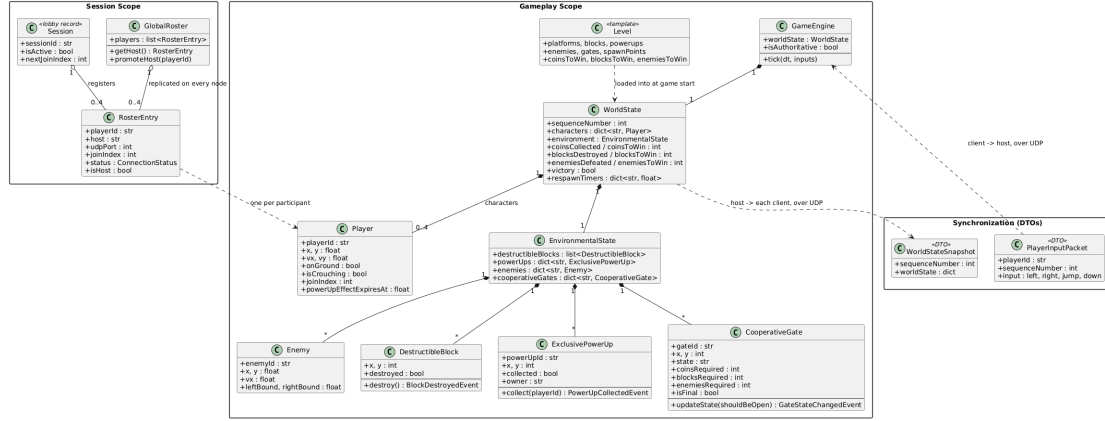


Figure 3: Domain model. The session scope (Session, RosterEntry, GlobalRoster), the gameplay scope (the engine, the world state and the entities it simulates) and the transfer objects exchanged between nodes (PlayerInputPacket, WorldStateSnapshot). The role of a node is an attribute of the node itself, since it is reassigned at runtime by an election, and the authority of an engine is a single flag on it.

- **Persistent operational state (local disk).** A small file holding the session identifier, the local player identifier and the last known peers. It is written when a match starts and deleted on a clean exit, so that finding it at start-up means exactly one thing: the previous run ended abnormally while a match was in progress. It is the only information that outlives the process, and it exists solely to make the rejoin of a failed node possible (**FR-11**).

3.4 Interaction

The interaction among components changes dynamically depending on the phase of the game session. The system shifts between a centralized coordination model (during discovery) and a distributed peer-to-peer model (during gameplay and fault recovery).

3.4.1 Communication Flows

We identify three primary communication flows that describe "how, when, and what" components communicate:

1. Discovery and Handshake (Centralized → P2P)

- **When:** At application startup or when a user wants to join a specific room.
- **How:** The interaction begins with a connection to the **Lobby Service**. Once the target session is identified, the Client switches to a direct connection with the **Host**.

- **What:** The Client exchanges control messages to request the room list or create a new session. Upon receiving the `HostEndpoint`, it performs a direct handshake with the Host to synchronize the initial map state and join the roster.

2. The Real-Time Gameplay Loop (Hybrid P2P)

- **When:** Continuously during the active game session.
- **How:** The communication utilizes a dual-channel approach: a high-frequency UDP stream for physics and websocket for critical events.
- **What:**
 - *Client → Host:* Clients send `PlayerInputPacket` (Key strokes + Timestamp).
 - *Host → Clients:* The Host broadcasts `WorldStateSnapshot` (Entity positions + Animations).
 - *Bidirectional:* Critical changes like `GateOpened` or `BlockDestroyed` are sent reliably via websocket.

3. Failure Detection and Recovery (Direct Peer-to-Peer)

- **When:** Activated solely when the Host fails to send updates for a duration exceeding $T_{timeout}$.
- **How:** Surviving peers use the cached `GlobalRoster` to communicate directly, bypassing the Lobby.
- **What:** Peers exchange consensus messages (`NewHostClaim`) to elect a new Leader and resume the game session.

3.4.2 Interaction Patterns

To address the critical requirement of **Low Interaction Latency (NFR-1)** while ensuring **Authoritative Consistency (FR-4)** and **Fault Tolerance (FR-8)**, the architecture does not rely on a single communication paradigm. Instead, it adopts a hybrid model where specific distributed patterns are selected to optimize the distinct phases of the application lifecycle:

- **Message-passing** Since the system nodes do not share physical memory, all coordination and state sharing occur exclusively through the exchange of messages over the network. The system implements this via:
 - *Unreliable Datagrams (UDP):* For the high-frequency gameplay loop where speed is prioritized over completeness.
 - *Reliable Streams (WebSocket):* For session lifecycle management and critical game events.

- **Synchronous Invocation** This pattern is strictly adopted during the interactions with the **Lobby Service**. The Client sends a request (e.g., `GetSessionList`) and blocks its flow (asynchronously) waiting for the specific response from the Lobby. This request-response cycle ensures that the client possesses valid connection data before attempting to join a peer.
- **Brokerless Publish-Subscribe (Pub-Sub)** The system adopts this pattern in two distinct contexts to decouple components:
 - *Network Level (Topic-based)*: The active game session acts as a "Topic". The **Host** acts as the Publisher, broadcasting state updates. The **Clients** act as Subscribers; by joining the session, they subscribe to the stream of updates regarding that specific world state.
 - *Internal Level (Event-Driven)*: As described in the Architecture (Section 3.1), the internal software layers utilize a local Pub/Sub mechanism. The Network Layer "publishes" incoming packets as events, and the Domain Logic Layer "subscribes" to them. This decouples the receiving logic from the processing logic.
- **Leader-based Consensus** The system relies on a strong consistency model centered around a Leader.
 - *Normal Operation*: The **Host** acts as the elected Leader. It is the sole authority responsible for ordering events, resolving conflicts (e.g., two players grabbing the same power-up), and validating the state (FR-4).
 - *Fault Tolerance*: When the Leader fails, the system transitions to an election phase to reach a new agreement on who the next Leader will be, ensuring the continuity of the authoritative model.

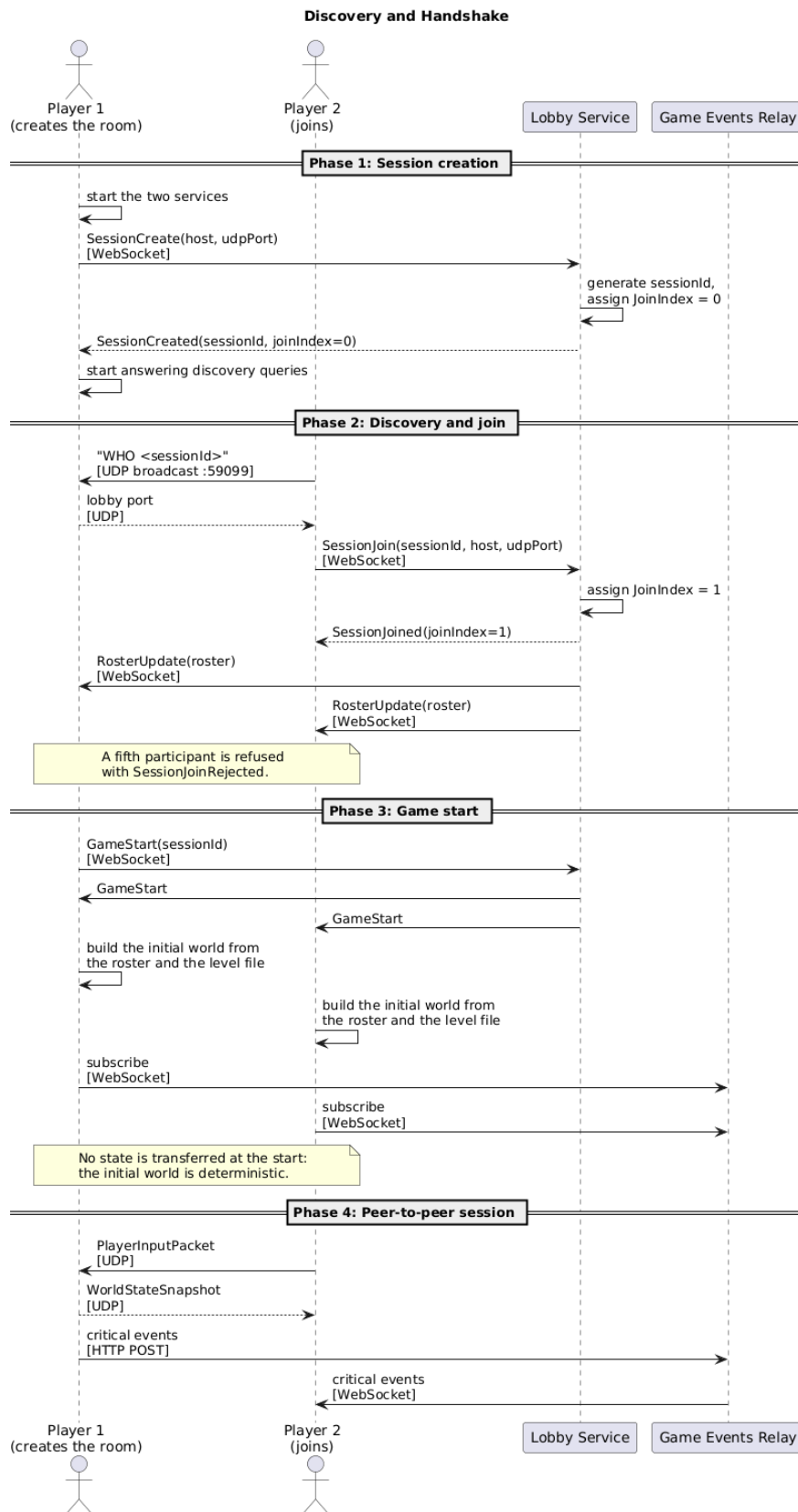


Figure 4: Sequence Diagram of Discovery and Handshake phase

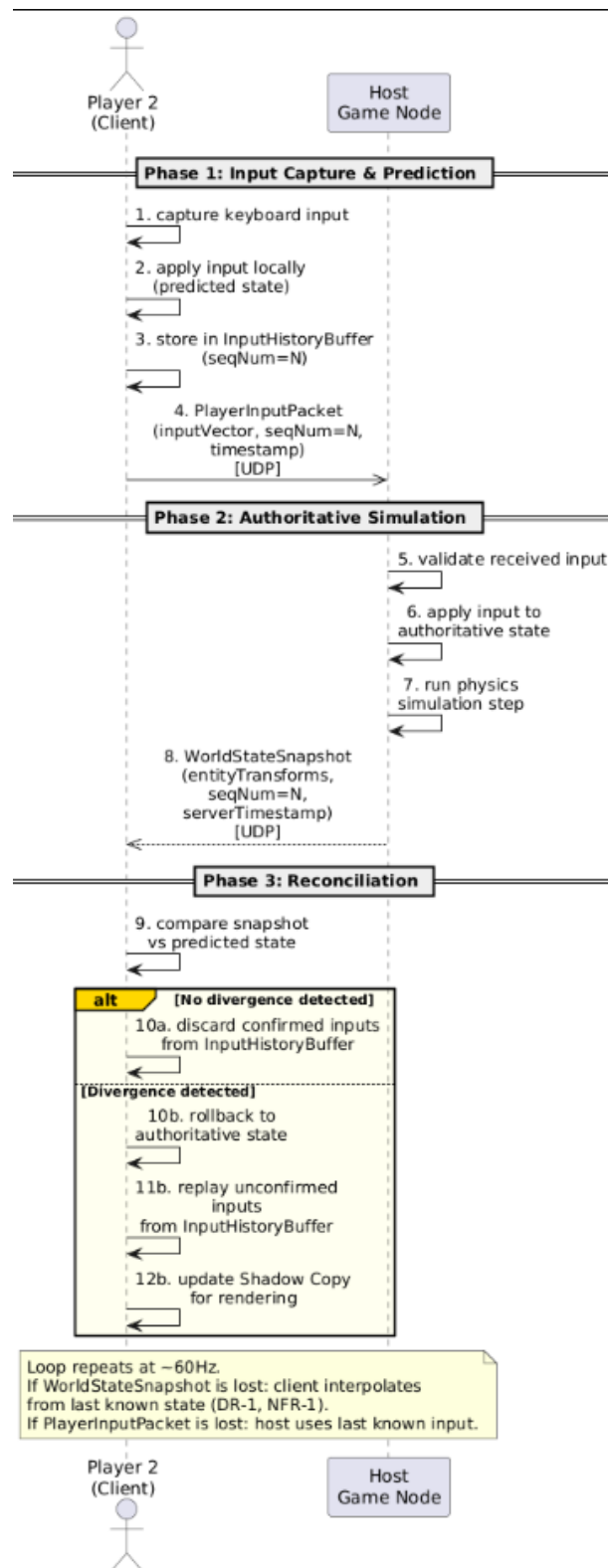


Figure 5: Sequence Diagram of Gameplay Loop

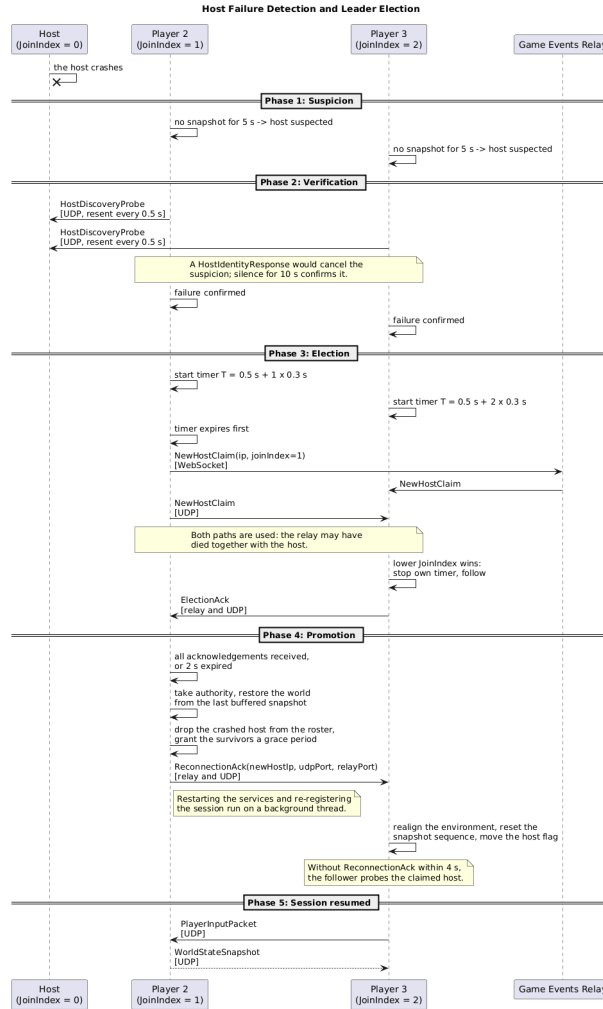


Figure 6: Sequence Diagram of Host Failure Detection and Leader Election

3.5 Behaviour

The behavior of each component is event-driven and depends on its current role within the session (Host or Client). All nodes process incoming messages and local inputs through a non-blocking event loop.

- **Host:** receives *PlayerInputPacket* messages from clients, orders them, updates the authoritative state of the game by generating *WorldStateSnapshot* at each tick, and sends it to all clients connected. It also processes critical domain events to ensure consistency with clients. The Host discards duplicate or out-of-order *PlayerInputPacket* messages based on *SequenceNumber*.
- **Client:** captures user inputs, applies them locally (Client-Side Prediction), and then sends them to the host via UDP. When it receives a *WorldStateSnapshot* from the host it will perform a reconciliation. Clients maintain a bounded input history buffer storing recent inputs; upon receiving a snapshot, the client resets its state and reapplies all inputs that have not yet been acknowledged by the Host. Clients discard stale *WorldStateSnapshot* messages using the *ServerTick* and ignore already processed events using unique event identifiers.
- **All nodes:** All nodes may trigger a *HostTimeoutDetected* if no *WorldStateSnapshot* has been received within $T_{timeout}$ and computes a new Host candidate using the lowest *JoinIndex*. If a node wants to reconnect it will send a *HostDiscoveryProbe* to known peers, receives the current Host endpoint and the latest *WorldStateSnapshot* from the current Host

The system is primarily composed of stateful components.

The Host maintains the authoritative Simulation State and Environmental State, updating them at each simulation tick. Clients maintain a predicted local state (for responsiveness) and a visual state (for rendering), both derived from the authoritative snapshots. All nodes maintain a replicated Topology State (*GlobalRoster*) required for autonomous leader election.

Stateless behavior is limited to transient message handling within the Network Layer, where incoming packets are processed and forwarded as events without persistent storage.

The Host is exclusively responsible for updating the authoritative global state at each simulation tick by applying validated client inputs and processing domain events within the physics engine.

Clients update their local state in two ways: immediately after local input, updating a temporary state, or Upon receiving authoritative snapshots, correcting discrepancies by replaying buffered inputs.

Environmental State (e.g., destroyed blocks, collected power-ups) is updated only by the Host and propagated reliably via WebSocket to ensure consistency.

The Topology State (*GlobalRoster*) is initially provided by the Lobby and subsequently maintained by the Host. In case of Host failure, nodes rely on their last known local copy to perform leader election.

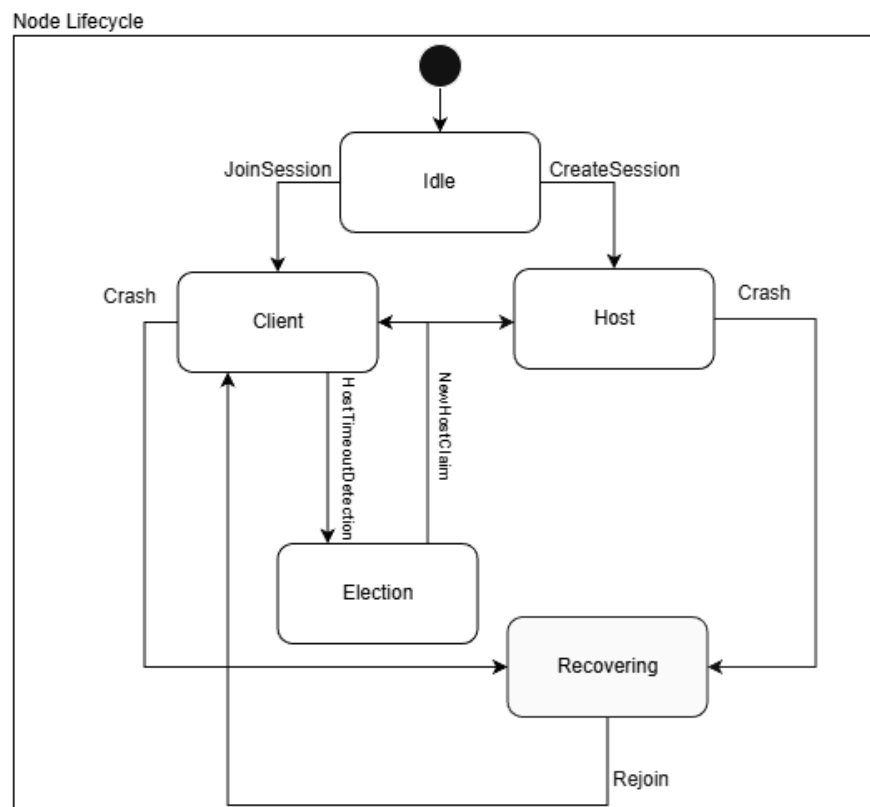


Figure 7: State diagram of a node lifecycle

3.6 Data and Consistency Issues

The system is explicitly designed around short-lived, transient game sessions, minimizing the need for persistent storage. Instead of a traditional database, the architecture relies primarily on in-memory data structures, supplemented by a minimal local cache for fault tolerance.

Data Storage and Persistence

All session-critical data is maintained in-memory due to the real-time nature of the application. The following categories of data are stored during execution:

- **Simulation State:** High-frequency data (positions, velocities, animation states). Stored in-memory as object instances on the Host and shadowed on Clients. *SequenceNumbers* and *ServerTick* values are used to enforce ordering and ensure idempotent processing of messages.
- **Environmental State:** Persistent changes to the game world (destroyed blocks, collected power-ups). Stored in-memory to prevent permanent inconsistencies.
- **Topology State (*GlobalRoster*):** A key-value dictionary mapping Player IDs to network endpoints. The *GlobalRoster* is replicated across peers and used to support autonomous host re-election in case of failure. However, it is maintained as a best-effort structure: nodes may temporarily operate on outdated or partial information. Therefore, peer liveness is validated dynamically through connection attempts during leader election and reconnection procedures.
- **Synchronization State:** Metadata (Sequence Numbers, input history buffers) stored in circular buffers to enable efficient client-side reconciliation.

Regarding **persistent data**, the system does not utilize external databases (Relational, NoSQL, or Graph). The only persistent storage is the *Operational State*, saved locally on each node's file system as a simple key-value document (e.g., `session_metadata.json`). This file temporarily caches the Session ID and the latest *GlobalRoster*. The key-value document format is chosen for its simplicity and read speed, allowing a recovering node to independently locate the session without querying the central Lobby. This file is deleted upon a graceful exit.

Database Queries and Access Patterns

Because no traditional database is deployed, there are no external queries (e.g., SQL statements) or concurrent database read/write locks to manage. All components operate directly on local in-memory data structures.

The Host performs authoritative read and write operations on the Simulation and Environmental State during each physics tick. Clients perform read-only operations on the replicated visual state, while writing to their local predictive state buffers. Concurrent access is managed naturally through the single-threaded asynchronous event loop

of the Domain Logic Layer, which processes events sequentially and prevents memory race conditions.

Shared Data Between Components

Data must be continuously shared across nodes to maintain a coherent distributed simulation. The actively shared data includes:

- **Simulation State:** Shared continuously via periodic *WorldStateSnapshot* UDP datagrams.
- **Environmental State:** Shared via reliable WebSocket messages to guarantee delivery and avoid divergence.
- **Topology State:** Shared via *RosterUpdate* WebSocket messages whenever a player joins or leaves, ensuring all nodes possess the updated peer list required for fault tolerance.

Consistency Model and Trade-offs

The system adopts a hybrid consistency model tailored to the requirements of real-time gameplay, prioritizing availability and responsiveness over strict consistency, in accordance with the CAP theorem.

- **Eventual Consistency (Simulation State):** High-frequency state updates are transmitted via UDP, tolerating packet loss. Clients may temporarily diverge from the authoritative state, but convergence is guaranteed as new snapshots are received. During normal operation, temporary inconsistencies (e.g., positional desynchronization) are tolerated to maintain smooth gameplay.
- **Strong Consistency (Critical Events):** Irreversible changes (e.g., block destruction, power-up collection) are transmitted via reliable WebSocket communication, ensuring that all nodes observe the same sequence of events.
- **Ordering Guarantees:** All state updates and events are tagged with a monotonically increasing sequence number. Clients discard stale messages and apply updates only if they are newer than the current local state, ensuring temporal consistency across different communication channels.

3.7 Fault-Tolerance

Fault tolerance is the central focus of the system's distributed architecture. The system is designed to seamlessly handle partial failures, specifically the crash or disconnection of the authoritative Host node, without terminating the active game session.

Data Replication and Sharing

The system relies on the full replication of the Topology State, specifically the *GlobalRoster*, across all participating peers. The *GlobalRoster* contains a complete map of all participants, including their Player IDs, network endpoints (IP and UDP port), and a monotonically increasing *JoinIndex*.

This replication is implemented because it guarantees that, in the event of a Host crash, every surviving node already possesses the necessary information to autonomously initiate a leader election without needing to contact the centralized Lobby Service. The roster is initially populated by the Lobby Service via *RosterUpdate* messages during the session setup phase. During active gameplay, any changes to the roster (e.g., a player voluntarily disconnecting) are reliably broadcasted by the Host to all clients via the Web-Socket channel, ensuring all local copies remain synchronized. Every peer stores the list of all the connected peers, necessary for the host re-election in case of its disconnection.

Heart-beating, Timeout, and Retry Mechanisms

To detect component failures efficiently without flooding the network, the system employs an implicit heart-beating mechanism coupled with explicit timeouts.

Instead of transmitting dedicated heartbeat packets, the system leverages the high-frequency *WorldStateSnapshot* messages broadcasted by the Host via UDP. Since these snapshots are sent continuously at every simulation tick, they act as a natural heartbeat. Every Client monitors the arrival of these packets. If a Client receives no snapshots for a predefined threshold of $T_{timeout}$ seconds, it assumes the Host has crashed and triggers a *HostTimeoutDetected* event.

A retry mechanism is implemented during the subsequent leader election phase. After a timeout is detected, surviving nodes deterministically identify the new Host candidate. They then wait for $T_{election}$ seconds to receive a *NewHostClaim* from this candidate. If the timer expires without a claim, the system assumes the newly elected candidate is also unreachable (e.g., due to a cascading network partition). The retry mechanism dictates that peers remove the unreachable candidate from their local *GlobalRoster* and immediately repeat the election process selecting the next candidate in the join order.

NewHostClaim messages also include *JoinIndex* and *Timestamp* fields. In case of more than one claim, nodes will accept the one with the lowest *JoinIndex* (in case of ties will be selected the one with the earliest *Timestamp*). If a better claim is received, nodes update their current Host accordingly.

Error Handling and Host Migration

The primary error handled by the system is the failure of the Host Game Node, which represents the authoritative entity and a single point of failure for the gameplay loop.

When a Host failure occurs and is detected via the aforementioned timeout, the system executes a Host Migration process to recover the session. The error handling procedure unfolds autonomously:

1. **Deterministic Election:** Surviving peers scan their local *GlobalRoster* (excluding the crashed Host) and independently select the node with the lowest *JoinIndex* as the new Leader. Although the election is deterministic under normal conditions, additional metadata (e.g., timestamp) is used to resolve inconsistencies caused by network delays or partitions.
2. **Role Promotion:** The elected node transitions its internal state from Client to Host, activates the authoritative Game Server Logic, and broadcasts a *NewHostClaim* containing its active endpoint via a direct WebSocket connection. Each node maintains a local replica of the latest *WorldStateSnapshot*. Upon host migration, the elected Host promotes its most recent snapshot to authoritative state, ensuring continuity without global synchronization.
3. **Session Resume:** Surviving clients update their pointer to the current Host, reply with a *ReconnectionAck*, and resume sending their *PlayerInputPacket* stream via UDP to the new Leader. The persistent Environmental State (e.g., destroyed blocks) is preserved from the last known state before the crash.

Finally, if the legacy Host recovers from its crash or network partition, it attempts to rejoin by broadcasting a *HostDiscoveryProbe* to its known peers. Upon receiving a *HostIdentityResponse* with the new Leader's endpoint, the recovered node automatically relinquishes its former authority and rejoins the ongoing session as a standard Client.

3.8 Availability

In the context of this distributed multiplayer game, availability ensures that players can continuously interact with the game world even in the presence of partial failures or network instability. The system prioritizes the responsiveness of the gameplay loop, accepting eventual consistency where appropriate.

Caching Mechanisms

The system employs caching strictly at the node level to reduce reliance on the centralized Lobby Service and to mitigate network latency.

- **Topology Cache (*GlobalRoster*):** Each peer maintains a local, in-memory cache of the *GlobalRoster*. This is crucial for availability because it allows surviving nodes to autonomously compute a new leader during a Host crash without waiting for the Lobby Service to provide routing information.
- **Persistent Operational Cache:** A local file (`session_metadata.json`) temporarily caches the Session ID and the latest roster. This allows a recovering node to quickly locate the ongoing session and attempt a rejoin.
- **Input History Buffer:** Clients cache their recent inputs locally. This ensures that the local game instance remains highly available and responsive to user commands (via Client-Side Prediction) even while waiting for the authoritative snapshot from the Host to arrive over the network.

Load Balancing

There is no form of load balancing implemented within the system.

This architectural choice is driven by the system’s explicitly limited scalability (NFR-3). The application is designed for short-lived, small-scale sessions supporting a maximum of 4 concurrent players over a Local Area Network. The Lobby Service is an extremely lightweight component used solely for initial session discovery, and the core gameplay relies on direct Peer-to-Peer communication. Consequently, the traffic volume and processing overhead are negligible for modern desktop machines, making load balancers an unnecessary infrastructural complexity that would only introduce additional network hops and increase latency (violating NFR-1).

Network Partitioning Behavior

The system’s response to a network partition depends on which node is isolated, balancing availability with the need to prevent state corruption:

- **Client Isolation:** If a Client becomes temporarily partitioned from the Host, the system favors local availability. The isolated Client can continue to move locally within the game world using its predictive engine, ensuring the presentation layer does not freeze. However, to preserve strict consistency, the isolated Client cannot interact with global shared resources (e.g., destroying blocks or opening cooperative gates) until connectivity is restored and the Host can validate the action.
- **Host Isolation:** If the Host is partitioned from the rest of the network, the Clients will stop receiving *WorldStateSnapshot* packets. After $T_{timeout}$ seconds, the Clients will perceive the partition as a Host failure and autonomously elect a new leader from the surviving partition to restore the session’s availability. When the partition resolves, the original legacy Host will discover the new leader and rejoin the session as a standard Client, losing its previous authoritative status.

3.9 Security

The system is designed as a localized, transient multiplayer game operating over a Local Area Network (LAN) and does not process or store any sensitive user information. Consequently, its security architecture is intentionally minimal. There is no form of authentication implemented; players do not create accounts or provide credentials, relying instead on ephemeral sessions joined via a simple Session ID. Furthermore, the system does not utilize formal authorization mechanisms or access control models. While there is a clear division of authority between the Host and Clients regarding game state validation, these roles are dynamically assigned at runtime for functional coordination and fault tolerance rather than security enforcement. Finally, no cryptographic schemas are employed. The system completely omits token verification and data encryption, transmitting data in plaintext over UDP and WebSockets. This deliberate lack of encryption avoids unnecessary processing overhead, helping the system meet its strict low-latency requirements for real-time gameplay.

4 Implementation

This chapter documents the technology-dependent choices made while implementing the design described in section 3. The whole system is written in **Python 3.12 (IR-1)**, and the same source tree produces the three runtime components of section 3.2: the game client, the Lobby service and the Game Events service. The two services are packaged as containers, while the client is distributed as a standalone executable, as described in section 6.

Two implementation decisions are particularly relevant to the distributed nature of the system: the use of different transports according to the semantics of the exchanged data, and the ownership of the game state by a single thread.

4.1 Network Protocols

The system uses four network mechanisms, each associated with a different type of interaction.

4.1.1 UDP for real-time gameplay traffic

The gameplay loop exchanges `PlayerInputPacket` and `WorldStateSnapshot` datagrams directly between the host and each client (**IR-3**). The same transport is also used for host-failure verification, host discovery during recovery, and as a secondary channel for election and reconnection messages.

The implementation uses the standard-library `socket` module. The UDP socket is bound once, configured as **non-blocking**, and polled by the game loop so that receiving network data can never block the simulation. The host uses a fixed UDP port, while clients use an ephemeral port assigned by the operating system.

UDP is appropriate for gameplay because new inputs and snapshots supersede older ones. Retransmitting a lost snapshot would therefore often deliver obsolete information, while head-of-line blocking would unnecessarily delay newer updates.

4.1.2 WebSocket over TCP for coordination and critical events

Session coordination with the Lobby and reliable game events are exchanged over WebSocket (**IR-2**). WebSocket was preferred to raw TCP sockets because it provides message framing and is directly supported by the same asynchronous framework used by the server components.

The WebSocket client runs its own `asyncio` event loop in a background thread and exposes a thread-safe queue to the game loop. As a result, asynchronous network operations remain confined to the networking layer and the main simulation loop does not need to await network operations.

The Game Events service maintains the active WebSocket connections and broadcasts critical gameplay events, such as player deaths, gate changes and other events that must not be lost. It does not persist game state.

4.1.3 HTTP for publishing events to the Game Events service

The host publishes critical game events to the Game Events service through an **HTTP POST** endpoint. The service subsequently broadcasts the received event to its WebSocket subscribers.

Publishing is performed by a background worker consuming an internal queue. Consequently, a slow or temporarily unreachable Game Events service does not block the 60 Hz game loop. This separation is important because blocking the host on an auxiliary service could otherwise cause clients to stop receiving gameplay snapshots and incorrectly suspect a host failure.

4.1.4 UDP broadcast for session discovery

Session discovery is implemented through UDP broadcast on a dedicated port. A joining client enables `SO_BROADCAST` and sends a query containing the session identifier to the network broadcast address. The host of the requested session replies with its IP address and Lobby port.

The discovery service can restrict responses to addresses belonging to known session participants. Once discovery succeeds, normal session coordination takes place through the Lobby WebSocket connection.

4.1.5 Why not a single transport

Using a single transport for all communication was rejected because the different interactions have different requirements. Using only TCP would make real-time gameplay sensitive to retransmission delays and head-of-line blocking. Using only UDP would require implementing reliability and ordering for session-control and critical-event messages.

Election and reconnection messages are deliberately sent through both the reliable event channel and UDP. This provides redundancy because the Game Events service may be located on the same machine as the failed host and therefore become unavailable together with it.

4.2 Data Representation

4.2.1 Serialization format

All application-level messages exchanged between nodes are represented as **JSON** encoded in UTF-8. Serialization and deserialization are implemented through the standard-library `json` module.

Messages contain a `message_type` discriminator together with the fields required by that message. For example:

```
{"message_type": "player_input", "player_id": "player2",  
"sequence_number": 812, "left": false, "right": true,  
"jump": false, "down": false}
```

JSON was preferred to a binary format such as Protocol Buffers because the application operates on a small LAN and bandwidth is not a significant constraint. Human-readable messages also simplify debugging of gameplay, election and reconciliation traffic.

World-state snapshots are the main exception to the otherwise flat structure: their payload contains the serialized world as a nested object, including players, environment state, shared counters and respawn information. Structures not directly supported by JSON are converted explicitly before serialization.

4.2.2 Message contracts and validation

Messages are represented at two levels:

- **Python dataclasses**, used by the application layer and responsible for representing typed domain messages;
- **Pydantic schemas**, used to validate the corresponding wire representation.

A central serializer maps between these two representations. On reception, the message discriminator is inspected, the raw JSON object is validated against the corresponding schema, and only then is the typed dataclass created. Unknown or malformed messages are therefore rejected at the network boundary instead of reaching the game logic (**DR-1**).

The message serialization layer is also tested through round-trip tests covering the supported message types, reducing the risk of inconsistencies between the in-memory and wire representations.

4.3 Data Storage and Querying

The system uses **no database**, relational or otherwise, and therefore does not use SQL, NoSQL queries or an ORM. The authoritative game state is transient: it is updated at every simulation tick and becomes irrelevant when a game session ends. Persisting it would therefore introduce additional complexity without satisfying a system requirement.

Runtime state is kept in ordinary Python data structures, including dictionaries for players and synchronization metadata, lists for rosters and environment elements, and a bounded double-ended queue for the input history used by reconciliation.

The only persistent data is the **session metadata** required for recovery after a process restart. It is stored as a small key-value JSON document containing the session identifier, the local player identifier and the known peers. The file is written when entering a game, read during recovery, and deleted after a clean exit or when recovery fails. Since its access pattern is limited to these operations, a plain file is sufficient and avoids introducing a database dependency.

4.4 Authentication and Authorization

4.4.1 Authentication

No authentication mechanism is implemented, consistently with **DR-8**. Participants are identified through a *player identifier* derived from their JoinIndex, such as `player1`, `player2`, and so on. The identifier is assigned by the Lobby and is included in the relevant messages.

Knowledge of the session identifier is therefore the main boundary between participants and unrelated nodes. This is suitable for the trusted-LAN scenario targeted by the project, but it does not provide protection against a malicious participant. Introducing authentication tokens issued by the Lobby would be a possible extension, as discussed in section 9.

4.4.2 Authorization

Authorization is implemented implicitly through the roles and protocol rules enforced by the application rather than through a dedicated authorization framework.

- **World authority.** The host is marked as authoritative. Operations that determine authoritative outcomes, such as resolving collisions with enemies and power-ups, respawning players, opening cooperative gates and checking victory, are disabled on non-authoritative nodes.
- **Client actions.** During normal gameplay, clients send input and protocol messages but never authoritative world-state updates. Player input represents an intent that is interpreted by the host, rather than a position that the client can directly impose on the simulation.
- **Protocol rules.** Control messages are accepted only when their sender and current state make them meaningful. For example, a host claim from a lower JoinIndex takes precedence during an election, acknowledgements are accepted only when expected, and the Lobby rejects participants once the maximum number of players has been reached.

This results in an implicit two-role model, with the *host* having authority over the world and the *client* being restricted to non-authoritative operations. These mechanisms are primarily intended to preserve consistency rather than to provide security against malicious peers.

4.5 Technological Details

4.5.1 Game client

The client uses **Pygame** for the window, rendering surface, keyboard input and frame-rate limiting. Levels are created with the Tiled editor and loaded using **PyTMX**. The

map contains the platforms, spawn points, enemies, blocks, power-ups and gates required to construct the initial world.

All nodes load the same level definition, ensuring that the initial static environment is identical without requiring the complete level to be transferred over the network (section 3.4).

4.5.2 Concurrency model

The node follows a synchronous frame loop complemented by a small number of background threads. The game state has a single owner and communication between the main loop and background components takes place through thread-safe queues.

- **Main thread.** Reads input, polls UDP and internal queues, advances the simulation, sends network messages and renders the current state. It is the only thread that directly modifies the game state.
- **WebSocket threads.** A background thread is used for each WebSocket connection, running an independent `asyncio` event loop.
- **Event-publishing thread.** Sends queued game events to the Game Events service through HTTP.
- **Promotion thread.** Performs the slower parts of host promotion and re-registration after an election.
- **Discovery thread.** Handles UDP broadcast discovery requests while the node is acting as host.

No lock protects the game state because only the main thread owns it. Background threads communicate with the simulation exclusively through queues.

4.5.3 Fixed simulation step

The simulation advances using a constant `TICK.INTERVAL` of 1/60 s. The real elapsed time measured by Pygame is used to limit the frame rate but is not passed to the simulation.

This fixed timestep ensures that two nodes processing the same sequence of inputs integrate physics using the same Δt . This is particularly important for client prediction and reconciliation, because replaying inputs with different timesteps could lead to a state different from the one computed by the authoritative host.

The trade-off is that under heavy CPU load the simulation may advance more slowly than real time rather than compensating with a larger timestep.

4.5.4 Timeouts and timers

Distributed deadlines are implemented by comparing timestamps obtained from a local monotonic or wall-clock source inside the frame loop; no distributed clock is assumed. Each node therefore makes timeout decisions using only its own notion of time.

The main parameters governing distributed behaviour are shown in table 2.

Constant	Value	Meaning
TICK_RATE	60 Hz	simulation and snapshot rate
HOST_TIMEOUT_S	5.0 s	silence after which a client suspects the host
HOST_VERIFY_TIMEOUT_S	10.0 s	maximum duration of host verification
HOST_VERIFY_RESEND_INTERVAL_S	0.5 s	resend interval of the verification probe
T_ELECTION_BASE_S	0.5 s	base delay of the staggered election timer
T_ELECTION_DELTA_S	0.3 s	additional delay per JoinIndex unit
ELECTION_CLAIM_TIMEOUT_S	2.0 s	deadline for election acknowledgements
RECONNECTION_FALLBACK_TIMEOUT_S	4.0 s	delay before direct probing of the elected host
UDP_INPUT_TIMEOUT	10.0 s	silence after which a client is evicted by the host
INPUT_HISTORY_SIZE	60	maximum number of inputs kept for replay
SNAPSHOT_TIMEOUT	0.15 s	interpolation window for remote state
MAX_EXTRAPOLATION_TIME	0.35 s	maximum extrapolation before freezing remote state
MAX_PLAYERS	4	maximum participants in a session
DISCOVERY_TIMEOUT	10 s	maximum time for connecting to the host

Table 2: Constants governing the distributed behaviour of a node.

The relatively long host-verification and input timeouts are intentional. They are designed to tolerate temporary process stalls on a loaded machine rather than immediately interpreting several seconds without packets as a permanent failure. The verification phase therefore distinguishes a temporarily unresponsive host from a host that has actually failed before starting an election.

4.5.5 Infrastructure services

Both infrastructure components are implemented as **FastAPI** applications served by **Uvicorn**. The Lobby exposes a WebSocket endpoint for session management, while the Game Events service exposes a WebSocket endpoint for subscribers and an HTTP endpoint through which the host publishes events.

The Lobby maintains active sessions and rosters in memory. The Game Events service maintains only active connections and forwards received events; neither service persists

the game state.

Both services are containerized with Docker. The client can start, inspect and stop the required containers through the Docker command-line interface. During host migration, container startup is performed in the background so that restarting the infrastructure does not block the gameplay loop.

4.5.6 Simulating adverse network conditions

The UDP transport supports configurable packet loss and artificial one-way latency. Packet loss is implemented by probabilistically discarding outgoing datagrams, while artificial latency queues outgoing packets until their configured delivery time.

Both mechanisms default to zero and are intended for local testing. They make it possible to evaluate prediction, reconciliation and failure-detection behaviour under controlled network conditions without requiring a dedicated network emulator.

4.5.7 Development tooling

Dependencies are managed with **Poetry**, with exact versions recorded in the lock file. **Ruff** is used for linting and formatting, **pytest** for automated tests, and **PyTestArch** to verify architectural constraints on the actual import graph.

The game client is packaged for Linux and Windows using **PyInstaller**, while the Lobby and Game Events services are packaged as Docker images. These artifacts are produced by the continuous-integration workflows described in section 6.

5 Validation

The system was validated through an automated test suite implemented with `pytest`. The tests cover the main functional and architectural properties of the distributed game, with particular attention to gameplay logic, multiplayer communication, state synchronization, prediction and recovery from node failures.

5.1 Automatic Testing

The current test suite contains 106 automated tests organized according to the main layers and components of the system. Tests are divided into application, domain, integration, network, and shared components, with additional tests for the application entry point and architectural dependencies.

Domain and Gameplay

The domain tests verify the deterministic behaviour of the game engine and the main cooperative game mechanics. They cover player movement and jumping, collision handling, enemy interactions, player death and respawn, temporary power-ups, destructible blocks, shared coin collection, gate activation and level completion. The tests also verify that shared objectives are updated only by the authoritative engine and that consumable entities cannot be processed more than once.

The domain layer contains 13 tests, mainly in `test_engine.py`, `test_entity.py`, `test_objectives.py`, and `test_shadow_copy.py`.

Multiplayer State and Client Prediction

The application tests verify the mechanisms used to keep clients responsive while preserving an authoritative game state. Prediction tests check that local inputs are recorded and replayed correctly, that acknowledged inputs are removed from the history, and that stale inputs are discarded after a host migration. Integration tests verify that prediction and reconciliation are performed only by clients, that fixed simulation steps are used, and that stale snapshots do not overwrite newer state.

The same group also tests the client-side shadow copies used for remote players. Remote entities are interpolated between snapshots and extrapolated when necessary, while stale or out-of-order snapshots are ignored.

Overall, these mechanisms are covered by the prediction, reconciliation, shadow-copy, and related `NodeController` tests in the application layer.

Lobby and Session Management

The lobby and session tests verify the creation of a session, assignment of `join_index` values, player membership, game start, player departure, and session closure. The tests also verify the maximum session size: a fifth player is explicitly rejected.

Additional tests cover rejoining an active session and the persistence of the session metadata required to recover after a node failure. The roster tests ensure that players remain ordered by `join_index`, that duplicate indices are rejected, and that the host is correctly identified.

These behaviours are tested through the lobby service, lobby phase, roster, session metadata, and node controller tests.

Host Failure, Election and Recovery

Host failure is one of the main distributed-system mechanisms validated by the test suite. Failure detection tests verify that a temporary lack of snapshots does not immediately trigger an election and that the host is probed before starting the recovery procedure.

The election tests verify the deterministic election based on `join_index`, including staggered election timers, host claims, acknowledgements, quorum handling, and prevention of multiple simultaneous hosts. The recovery tests additionally verify peer probing, session recovery, re-registration of the promoted host, roster updates, and synchronization of the buffered environment state.

The complete migration procedure is also exercised by integration tests, which verify that a crashed host is replaced and that the surviving players continue with the recovered world state.

Network Communication and Integration

The network and integration tests verify the actual communication mechanisms used by the game. WebSocket tests cover lobby and critical event communication, while UDP tests verify real-time gameplay input and snapshot exchange.

Serialization tests verify that all supported WebSocket and UDP messages can be encoded and decoded correctly and that world state can be transferred over both transports. Unknown message types and malformed messages are rejected.

The suite also includes tests for artificial network latency and LAN discovery. The localhost integration test exercises the complete path from a client input to the authoritative host and back through the resulting snapshot. Additional integration tests verify that game events are propagated to all connected clients and that player departures are correctly reflected in the shared state.

Architectural Validation

In addition to functional tests, the project includes architecture tests that inspect the dependency graph between software layers. These tests verify that the domain layer does not depend on other layers, that the network and application layers do not depend on presentation, and that presentation only receives the data transfer object required for rendering.

The tests also verify that the composition root correctly wires the main components and that the default implementations satisfy the interfaces used for dependency injection.

This provides an automated check that the implementation remains consistent with the layered architecture described in the design.

5.2 Test Execution

The complete test suite can be executed with:

```
poetry run pytest
```

Automated tests are also executed by the project’s GitHub Actions workflow on pushes and pull requests targeting the `main` branch. The workflow installs the project dependencies using Poetry and executes the complete `pytest` suite. The same workflow also runs Ruff for static analysis and code formatting checks.

5.3 Validation Scope and Limitations

The automated tests primarily validate functional correctness and the behaviour of the distributed mechanisms at component and integration level. In particular, host election, recovery, state synchronization, network communication, and client prediction are explicitly exercised.

Some non-functional properties are not directly measured by the test suite. In particular, the tests do not provide a quantitative benchmark of network latency, maximum scalability beyond the supported four-player session, or long-term performance under sustained network load. Similarly, complete network-partition scenarios are not reproduced as part of the automated suite. These aspects therefore remain outside the scope of the current automated validation.

6 Deployment

The deployment separates the desktop game client from the two infrastructure services of section 3.2. The client is distributed as a standalone executable, one per operating system, while the Lobby and the Game Events relay are distributed as Docker images. The separation follows from what the two kinds of component need: the client requires a display, a keyboard and a graphics library, and cannot be containerised in a portable way, whereas the services are headless and benefit from a reproducible environment that does not depend on what is installed on the machine of the player.

A property of the design makes this arrangement less obvious than it looks. Because the host role can migrate to any node (**FR-8**), *every* machine must be able to run the services, not only the one that created the session. Docker is therefore required on all participating machines, and each of them carries the same two images.

6.1 Deployment architecture

A deployed session consists of up to four machines on the same LAN, each running the same game executable, plus the two containers that run on whichever machine currently holds the host role.

- **Game client:** a platform-specific executable produced with PyInstaller. It bundles the Python interpreter, all the runtime dependencies and the game assets, so the player needs neither Python nor Poetry nor any library installed.
- **Lobby container** (image `distributed-smb-lobby`, WebSocket on port 50002): session creation, membership and roster distribution.
- **Game Events container** (image `distributed-smb-gameevents`, HTTP and WebSocket on port 50003): the stateless relay for critical events and election messages.

Both images are built from `python:3.12-slim`, install the project with its declared dependencies and start the corresponding application with Uvicorn. They contain the same source tree as the client: what differs is the entry point, not the code.

The containers are not started by the user. When a player creates a room, the client starts them through the Docker command-line interface and connects once they are reachable; when a node is promoted after a failure, it does the same on its own machine. The lifecycle is driven by the application, which has two consequences worth stating.

- **Starting is idempotent.** Before starting a container, the client asks Docker whether one with that name is genuinely running. If it is, the client reuses it; if it is stopping, the client waits until it has disappeared before creating a new one. Two nodes on the same machine, or a promoted node whose predecessor is still shutting down, therefore do not collide.

- **Stopping is not automatic.** The containers are stopped only when the application exits cleanly. If the game process dies abnormally, which is exactly what happens in the failure scenario the system is built for, the containers are left running on purpose, so that the node promoted next finds them alive and reuses them instead of paying for a restart.

6.2 Prerequisites

For a player, the requirements are minimal:

- **Operating system:** Linux or Windows (the packages are built for both).
- **Docker:** Docker Engine on Linux, Docker Desktop on Windows, installed and running.
- **Network:** all the machines connected to the same LAN, with the ports of table 3 reachable.

Nothing else has to be installed: no Python runtime, no library and no configuration file. Everything else is encapsulated either in the executable or in the two images.

For development and for building the artefacts, the project additionally requires Python 3.12 or later and Poetry, which resolves the dependencies declared in `pyproject.toml` and pins their exact versions in `poetry.lock`. The runtime dependencies are pygame, FastAPI, Uvicorn, websockets, HTTPX and PyTMX; pytest, Ruff and PyTestArch are development dependencies, and PyInstaller is installed in the build environment only.

6.3 Distribution package

The images are distributed as compressed archives shipped next to the executable, and not through Docker Hub or any other registry. The reason is the environment in which the system is meant to run: a closed LAN with no guarantee of Internet access, and machines on which nobody should have to log into a registry. A package produced once is self-sufficient.

Each platform-specific package contains four files:

- the game executable, `smb-game` on Linux or `smb-game.exe` on Windows;
- `smb-lobby.tar.gz`, the Lobby image exported with `docker save`;
- `smb-gameevents.tar.gz`, the relay image exported the same way;
- the setup script for that platform, `setup.sh` or `setup.bat`.

The two packages, `smb-game-linux.zip` and `smb-game-windows.zip`, are published on the Releases page of the repository, which gives them a stable download link. They are built by the continuous-integration workflow described in section 6.7; the same bundle can also be produced locally with the `build.bundle.sh` script, which builds both images and exports the two archives:

```
./build_bundle.sh bundle          # builds and saves both images
poetry run pyinstaller main.spec  # builds the executable into dist/
```

6.4 Installation and execution

On a machine that has never seen the project, the procedure is the following.

1. **Install Docker** and make sure the daemon is running.
2. **Download and extract** the package for the operating system, `smb-game-linux.zip` or `smb-game-windows.zip`, from the Releases page of the repository.
3. **Run the setup script once.** It loads the two images into the local Docker installation, without contacting any registry:

```
./setup.sh          # Linux
setup.bat           # Windows
```

The script runs `docker load` on both archives and then prints the images it has installed, which can be checked at any time with `docker images`.

4. **Start the executable**, `./smb-game` or `smb-game.exe`.

Expected outcome. The player who selects *Create Room* implicitly starts the two containers on their own machine and receives a session identifier; `docker ps` then shows `smb-lobby` and `smb-gameevents` running. The other players select *Join Room*, enter that identifier, and appear in the roster of every participant. From that moment the session is self-sufficient: no further command is needed, and the failure of the host is handled by the system itself (section 3.7).

The setup is performed once per machine, and it is required on every machine, clients included, because any of them may later become the host.

6.5 Containerization

Each service has its own Dockerfile, and both follow the same three steps: copy the project definition and the sources, install the project, and start the corresponding module with Uvicorn on its port.

```
FROM python:3.12-slim
WORKDIR /app
COPY pyproject.toml README.md ./
COPY src/ ./src/
RUN pip install --no-cache-dir .
ENV PYTHONPATH=/app/src
EXPOSE 50002
CMD ["uvicorn", "distributed_smb.network.lobby_server_main:app",
    "--host", "0.0.0.0", "--port", "50002"]
```

The Dockerfile of the relay is identical except for the module it starts and the port it exposes. Neither image uses a multi-stage build: the project is pure Python, so there is no compilation step to separate from the runtime, and the only saving would come from discarding the build cache of `pip`, which `--no-cache-dir` already avoids creating.

Installing the project with `pip install .` instead of listing the dependencies by hand is a decision taken after a deployment failure. In an earlier version each image installed its own explicit list of packages, which drifted from the real dependencies of the code: when the domain started reading Tiled maps, the lobby image kept working in tests but crashed at start-up in a real session, because the import chain of the lobby reaches the world state and, through it, the map loader. Installing from `pyproject.toml` removes the possibility of that drift, at the cost of a slightly larger image.

How the client starts a container. The application issues the equivalent of the following command for each service, with the name and the port of that service:

```
docker run -d --rm -p 50002:50002 --name smb-lobby distributed-smb-lobby
```

Three details of that command matter. The `-p` mapping is used instead of host networking because Docker Desktop on Windows does not support `--network=host`, and the deployment must behave identically on both platforms. The `--rm` flag removes the container when it stops, so a failed host leaves no stopped container behind to collide with the next one. The fixed `--name` is what allows a node to recognise a container started by another process on the same machine and reuse it instead of failing.

Local orchestration. The repository also contains a `docker-compose.yml` that builds and runs both services with the same port mappings:

```
services:
  lobby:
    build: {context: ., dockerfile: Dockerfile.lobby}
    image: distributed-smb-lobby
    ports: ["50002:50002"]
  gameevents:
    build: {context: ., dockerfile: Dockerfile.gameevents}
    image: distributed-smb-gameevents
    ports: ["50003:50003"]
```

It is a convenience for local development, not the way the system is normally deployed: in a real session the containers are started by the game client, on the machine that needs them at that moment. No dependency between the two services is declared, because there is none: they never talk to each other, and each of them is reached directly by the nodes.

6.6 Configuration and networking

The system has no configuration file and no environment variables to set: every port is fixed at build time, and every address is discovered at runtime. Table 3 lists the ports that must be reachable on the local network, and that a personal firewall may have to be told to allow.

Port	Protocol	Use
50002	WebSocket over TCP	Lobby service (container)
50003	HTTP and WebSocket over TCP	Game Events relay (container)
50010	UDP	gameplay socket of the host: inputs and probes
OS-assigned	UDP	gameplay socket of a client: snapshots and replies
59099	UDP broadcast	session discovery

Table 3: Ports used by a deployed session.

Fixed ports are sufficient because a session is identified by the address of its host, and every machine has its own: two sessions hosted on two different machines of the same LAN use the same port numbers without interfering. The case that fixed ports do not support is two hosts on the *same* machine, which is outside the scope of the system.

No database, message broker or external service takes part in the deployment. The only file written on a machine is the rejoin metadata described in section 3.3, created in the working directory of the executable and removed on a clean exit.

For development and testing the executable also accepts command-line options that bypass the menu and the discovery:

```
smb-game --host                    # start directly as host
smb-game --client --session-id <ID> # start directly as client
smb-game --client --host-ip <IP> ... # skip LAN discovery
smb-game --host --latency 50 --drop-rate 0.05
```

The last two options inject the artificial network conditions described in section 4.5, and `--local-ip` overrides the address a node advertises to its peers.

6.7 Build and release automation

Two GitHub Actions workflows automate the project.

Code Quality & Tests runs on every push and pull request towards `main`. It installs the dependencies with Poetry and runs two jobs in parallel: Ruff, both as a linter and as a formatting check, and the whole test suite of section 5, including the architecture tests that verify the layering.

Build Game Client runs on every push to `main` and produces the distribution packages. It first builds the two Docker images on a Linux runner and exports them as compressed archives, then builds the executable twice, on a Linux runner and on a Windows runner, and assembles each executable together with the two archives and the setup script into a platform-specific ZIP file.

Building the Windows executable on a Windows runner is a necessity rather than a convenience: PyInstaller does not cross-compile, so each platform must be built on its own. This is also what made it possible to find and fix problems that appear only on Windows, such as the slower start-up of the containers, which required the connection retries described in section 3.7.

The two ZIP files produced by the workflow are then published as assets of a release, which gives the users of the system a stable link and avoids the expiry of the workflow artefacts.

7 User Guide

7.1 Requirements and Installation

The distributed version of Distributed SMB is provided as a platform-specific package containing the game executable, the Docker images required by the infrastructure services, and a setup script.

Before running the application, Docker must be installed and running on the machine. Docker Desktop can be used on Windows and macOS, while Docker Engine can be used on Linux. Docker is required on every machine participating in a multiplayer session, including clients.

All players should be connected to the same local network. If a firewall is enabled, it may be necessary to allow the application and the network traffic required by the game.

To install the application, download the appropriate package for the operating system from the Releases page of the project repository. Two packages are provided:

- `smb-game-windows.zip` for Windows;
- `smb-game-linux.zip` for Linux.

Extract the downloaded archive. It contains:

- the game executable (`smb-game.exe` on Windows or `smb-game` on Linux);
- `smb-lobby.tar.gz`, containing the Lobby Docker image;
- `smb-gameevents.tar.gz`, containing the Game Events Docker image;
- the appropriate setup script.

The setup only needs to be performed once per machine. On Windows, run `setup.bat`. On Linux, run:

```
./setup.sh
```

The setup script loads the two Docker images into the local Docker installation. The installation can be verified by checking that the corresponding images are available with:

```
docker images
```

7.2 Starting the Application

After the one-time setup has been completed, start the game by launching `smb-game.exe` on Windows or `smb-game` on Linux.

The main menu provides two options:

- *Create Room*: creates a new game session and makes the local player the host;
- *Join Room*: joins an existing game session as a client.

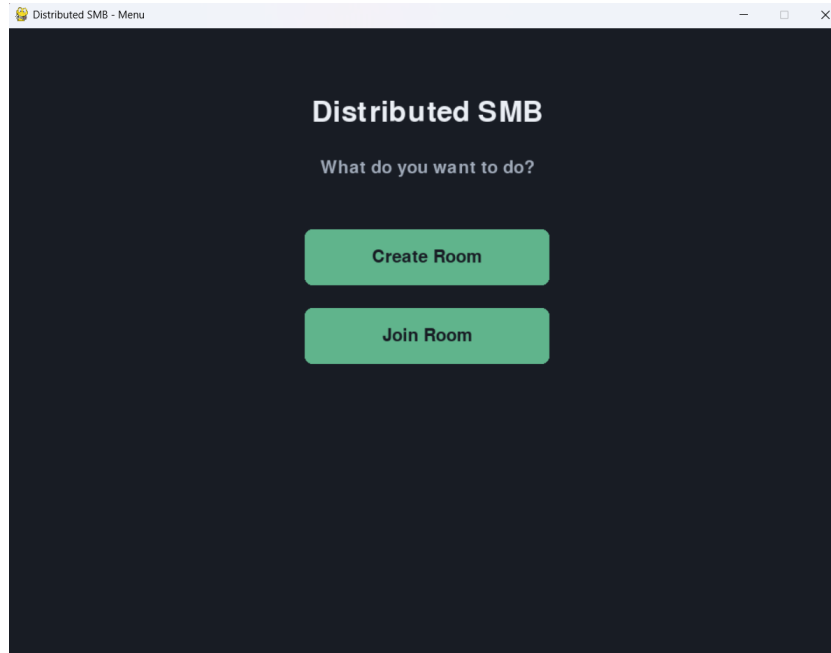


Figure 8: Main menu used to choose whether to host or join a session.

7.3 Creating a Game Session

To create a session, select *Create Room* from the main menu. The application starts the required Lobby and Game Events Docker containers and creates a new game session.

A unique session identifier is generated and displayed to the host. The host must communicate this identifier to the other players, for example through a chat application or verbally if the players are in the same room.

The host can wait for the other participants to join the session. The system supports up to four players in the same game session.

To start the match, the host can click the *Start Game* button or press the **ENTER** key.

The expected result is that all players in the lobby receive the game-start notification and automatically transition to the gameplay screen.

7.4 Joining a Game Session

To join an existing session, select *Join Room* from the main menu. The application asks the player to enter the session identifier provided by the host.

When LAN discovery is available, the host machine is detected automatically. The client then connects to the lobby and is added to the session roster.

The client remains in the lobby until the host starts the game. Once the match starts, the client automatically enters the game world and controls its assigned character.

If automatic discovery is unavailable because of network configuration restrictions, the host IP address can be specified manually together with the session identifier.

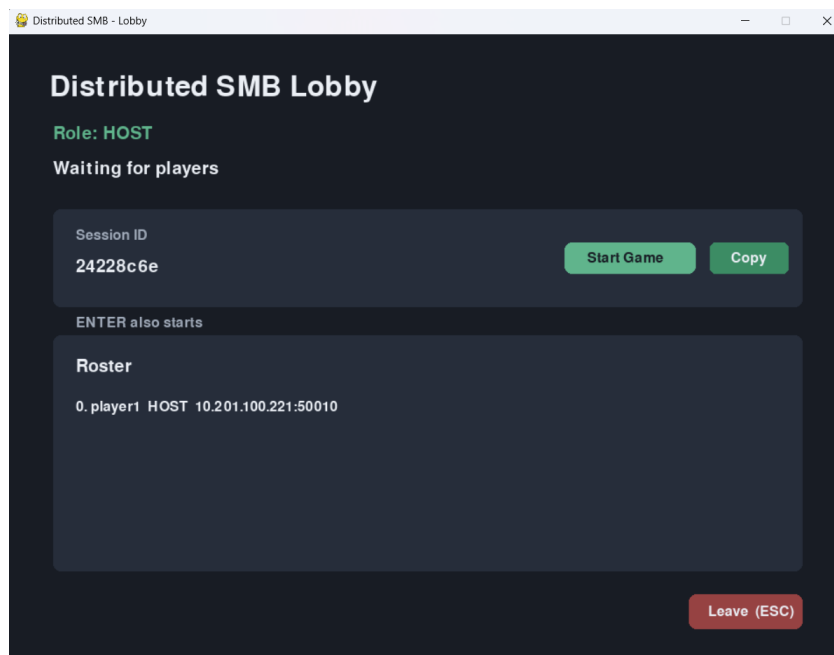


Figure 9: Host lobby with the session identifier and connected players.

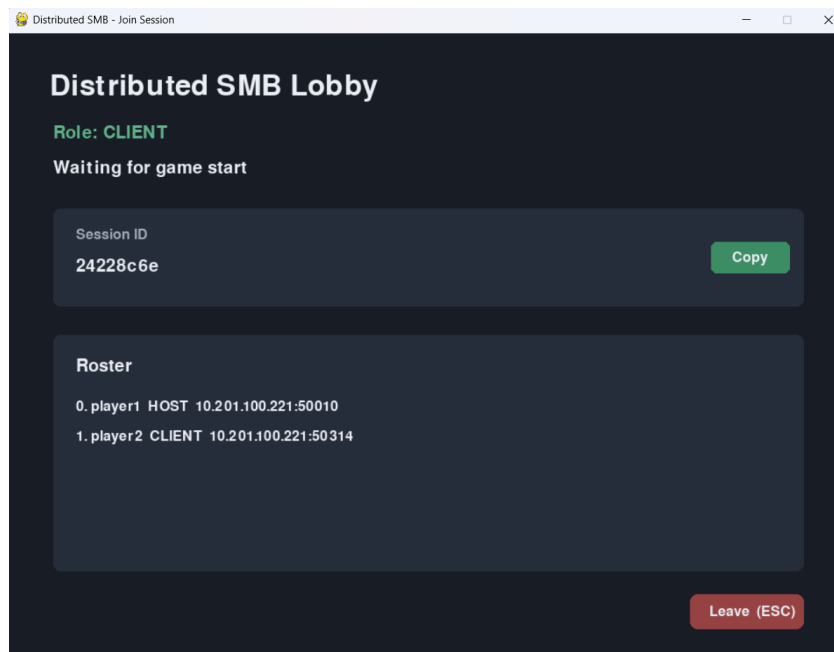


Figure 10: Client join screen.

7.5 Gameplay Controls

After the host starts the match, each player controls their own character using the keyboard. The available controls are shown in Table 4.

Table 4: Gameplay controls.

Action	Keys
Move left	A or Left Arrow
Move right	D or Right Arrow
Jump	W or Up Arrow
Crouch / move down	S or Down Arrow
Close the application	Close the game window

Players can move across platforms, jump over obstacles, collect coins and power-ups, destroy eligible blocks, defeat enemies by jumping on them, and progress through the level. The camera follows the local player while keeping the visible area within the game-world boundaries.

Changes to shared environmental elements are propagated to the other players, so that all clients observe the same cooperative game state.

7.6 Expected Gameplay Behaviour

The game is cooperative: actions performed by one player can affect the world seen by all the others. For example, when a coin is collected, an enemy is defeated, or a block is destroyed, the corresponding change is propagated to the other clients.

The client application uses local prediction and reconciliation to keep the local player's movement responsive despite network delays. Remote players are displayed using interpolation to reduce visible discontinuities caused by discrete network updates.

If a player falls outside the level bounds or is defeated, a death animation is shown. After the respawn delay, the player reappears at a valid spawn point and can continue playing.

The level is completed when the required cooperative conditions are satisfied and the players reach the final gate. When the victory condition is met, a victory overlay is displayed to the connected players.

7.7 Host Failure and Session Recovery

The system includes an automatic host-migration mechanism. If the current host becomes unreachable during a game, the remaining nodes detect the host timeout and initiate a leader-election procedure. One of the remaining nodes becomes the new host, while the other players reconnect to it.

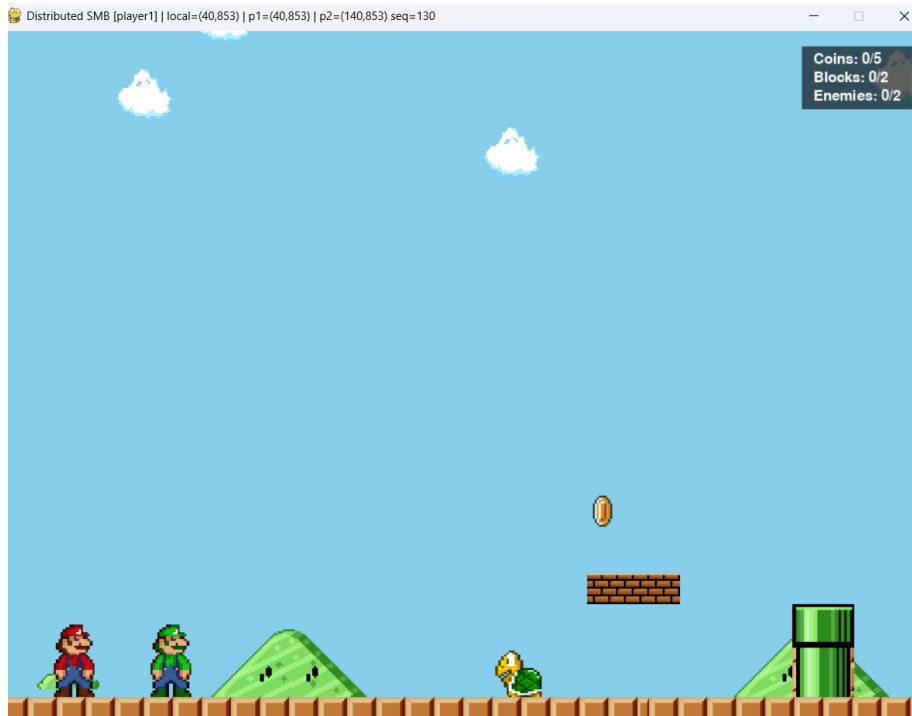


Figure 11: Gameplay screen with the shared world and progress indicators.

The selected node promotes its latest available game state and becomes the new authoritative host. From the user’s perspective, no manual intervention is normally required, although the game may briefly pause while the election and reconnection take place.

If the previous host becomes available again, it does not reclaim authority. Instead, it rejoins the session as a client.

If recovery cannot be completed, the user can close the application and create a new session following the procedure described in Section 7.3.

7.8 Troubleshooting

- **The host cannot create a session.** Verify that Docker is installed and running and that the required Docker images have been loaded using `setup.bat` or `setup.sh`.
- **A client cannot find or join the session.** Verify that all devices are connected to the same LAN, that the session ID is correct, and that the host application is still running. If automatic discovery is not available, verify the host IP address and network configuration.
- **The lobby does not start the game.** Only the host can start the match. The host must click *Start Game* or press ENTER.

- **The application cannot connect to the infrastructure services.** Verify that Docker is running and that the required Docker images are present. Also check that the ports required by the application are not blocked by the local firewall.
- **The host becomes unavailable during a match.** No manual action is normally required. The remaining nodes should detect the failure, elect a new host, and reconnect to the recovered session. If recovery does not complete, restart the application and create a new session.

8 Self-evaluation

8.1 Ludovico Nollino

My role within the group covered the distributed part of the system. I set up the project at the beginning, defining the layering, the dependency rule between the layers and the composition root, and I then developed the network layer and the application layer in their entirety: the UDP transport and the WebSocket client, the serialization and validation of every message, the Lobby service and the Game Events relay, LAN discovery, and, on top of them, the node controller with the host and client gameplay loops, the roster, client-side prediction and reconciliation. I designed and implemented the two mechanisms the project is built around, the leader election that replaces a failed host and the procedure that lets a failed node rejoin a running session, together with the failure detector they depend on. I also took care of the deployment: the containerization of the two services, the packaging of the game client, the distribution bundle and the continuous-integration workflows.

The strength I recognise in the result is that the system does what it promised in the most demanding case: the session survives the loss of the node that holds the authority, without a central coordinator and without transferring any state, and a node that crashed can come back into a game already in progress. Two decisions proved particularly sound. Confining the distributed logic to a single layer allowed the protocol to be hardened several times, adding a verification step before an election and a second transport for the election messages, without touching the rules of the game. Keeping the domain layer free of network and of real time made the simulation deterministic, which is what makes reconciliation converge at all, and made it possible to test the election and the rejoin without opening a socket.

The weaknesses are of two kinds. The first is that part of what the protocol does is calibrated on values, rather than derived from measurements: the failure detector tolerates a long silence because that is what the observed stalls required, and the prediction lead cannot tell a genuine increase in latency from a drift between two clocks, because both appear as the same number. Making the host acknowledge the last input it applied would remove that ambiguity, and is the first thing I would do next. The second is that the fault-tolerance protocol is correct in the scenarios that were validated, but it has cases that are known to be open, such as a second migration after a rejoin, and they are documented in section 9 rather than solved. Finally, the validation itself was mostly carried out with several processes on one machine and on two machines: a test on three or more physical machines would very likely have exposed more, and it is the check I regret not having been able to run.

8.2 Mattia Flamigni

My main role within the group was the development of the presentation layer and its integration with the distributed game logic. I implemented the Pygame application loop, rendering of the game world and animated character sprites, keyboard input handling, and the graphical lobby and connection screens. I also worked on the dynamic camera,

the integration of game assets and level elements, and the death and respawn animations. In addition, I contributed to improving the visual consistency of remote clients through shadow copies and interpolation of remote entities. During the final phase, I added reconciliation tests, particularly for enemy synchronization and local block state.

A key strength of my contribution is that it made the distributed features developed by the group accessible and visible to the player. Users can create or join a lobby, start a match, and receive graphical feedback about the game state. The separation between presentation, domain, and application layers also made integration with the other group members' work easier. Furthermore, interpolation of remote entities improves the perceived smoothness of gameplay when network updates are not continuous.

The main weaknesses concern the level of refinement of the graphical component. The interface is functional but still basic: it does not include advanced menus, configurable settings, audio, or support for different screen resolutions. Moreover, camera and interpolation behavior were mainly validated in controlled scenarios; more extensive tests with higher latency, packet loss, and a larger number of players could reveal additional cases to handle. Finally, some implementation choices were influenced by the progressive integration of changes to the game engine and network protocol, requiring adaptation work during the final stages of development.

8.3 Filippo Badioli

My main role within the group was the development of the domain logic and simulation core of the game, with particular attention to keeping it independent from both the presentation and networking layers. I initially implemented the basic game engine and the core game entities, including player movement, gravity, jumping and AABB collision detection. I subsequently extended the engine from a single-player model to a multi-player world state, ensuring that each player could be simulated independently while maintaining a consistent authoritative state.

I also developed the shared game entities and their associated events, including destructible blocks, exclusive power-ups and cooperative gates. This involved integrating their collision and state-transition logic into the game engine, defining deterministic conflict resolution for resources that can be accessed concurrently by multiple players, and modelling events such as block destruction, power-up collection, gate state changes and player disconnections. These entities were designed to be serializable so that the same domain state could be used by the networking layer and by the authoritative host.

Another significant part of my contribution concerned client-side prediction and reconciliation. I worked on the input history and prediction mechanisms required to execute player inputs locally while waiting for authoritative updates from the host. The deterministic behaviour of the game engine was particularly important here, since rollback and replay require the same initial state and input sequence to produce the same result. I also contributed to the state-management structures used by the distributed session, including the roster and the information required to identify players consistently across the different nodes.

A strength of my contribution is the separation between the domain simulation and

the rest of the system. The game engine does not depend on Pygame, sockets or the infrastructure services, which allowed the same simulation logic to be used by both authoritative and predictive execution and made it possible to test important behaviours independently from the network. Deterministic conflict resolution for shared entities was also useful for maintaining consistency between nodes: when multiple players interact with the same object, the authoritative host has a well-defined outcome rather than relying on the order in which network messages happen to arrive.

The main weakness is that the domain model became progressively more complex as distributed features were added. The initial engine was designed around a simpler single-player model, so extending it to support multiple players, shared environmental state, cooperative mechanics and prediction required several refactorings. Some of the resulting interfaces are therefore more complex than they would have been if all these requirements had been known from the beginning. In addition, although the core simulation and its deterministic behaviour were tested in isolation, more extensive end-to-end testing under real network conditions could have exposed additional interactions between prediction, reconciliation and the shared game state.

A further limitation is that some aspects of the game logic remain tightly connected to the assumptions of the current session model, particularly the fixed number of players and the authoritative-host architecture. Supporting more general session sizes or more complex forms of shared interaction would require additional abstraction. Nevertheless, the current design provides a sufficiently well-defined domain model for the distributed mechanisms implemented in the project and keeps the game rules independent from the underlying communication technology.

9 Future works

The system meets the requirements it set out to meet, but the validation left a number of cases open, and some of the mechanisms were kept simple on purpose. This chapter collects what would be done next, separating the limits of the fault-tolerance protocol from the engineering work around it.

9.1 Known limits of the fault-tolerance protocol

- **A second migration after a rejoin.** When the host fails, a client is promoted and the failed node rejoins, a subsequent failure of the new host can make two survivors promote themselves independently. The periodic roster broadcast reduces the window but does not close it, because the Lobby is re-registered with the new host alone and the membership known to each node can differ. The natural fix is to carry the full list of survivors in the message that re-registers the session, so that the Lobby and every node agree on the membership before the next failure.
- **A candidate that fails after claiming.** A follower stops its own timer as soon as it accepts a claim. If the claimer dies before announcing its promotion, the followers keep probing it and no new election ever starts. A deadline on the follower state, after which the election restarts among the remaining nodes, would close the gap, at the cost of a second tolerance value to calibrate.
- **An isolated host is not demoted.** When a partition heals, the node that was left alone keeps running its own copy of the session, and rejoins only after being restarted. It could notice the new host through the roster republished on the relay, or through a discovery query, and step down as a client with the same procedure used after a restart.
- **A client evicted for silence is not told explicitly.** It learns of its own eviction only through the departure event, and if that event is lost it keeps playing a game the host no longer includes it in. The host could answer an input from an unknown player with an explicit message, which would also make the condition observable in the logs.
- **Automatic rejoin and network mode.** A node that rejoins automatically from its cached metadata can start in the wrong mode, and fails to bind the ports of the services if it is later promoted. A fix was written and then set aside together with the work on the double migration, and should be redone in isolation.

9.2 Engineering improvements

- **Separating latency from clock drift.** The prediction lead is the distance between two free-running counters, so a genuine increase in latency and a drift between the two loops are indistinguishable from it. Having the host acknowledge the sequence number of the last input it applied would let a client measure its own

round-trip time locally and correct only the drift, instead of holding a calibrated value.

- **A fixed-timestep accumulator.** The simulation already advances by a fixed step, but only one step per rendered frame, so under heavy load the game slows down instead of catching up. An accumulator running zero, one or more steps per frame would keep simulated time aligned with real time, at the price of a more complex loop on both sides.
- **Interfaces for the transports.** The application layer still uses the concrete UDP and WebSocket classes. Putting them behind interfaces, as was done for the Lobby and the relay, would allow the gameplay loop to be tested without real sockets, and an alternative transport to be introduced without touching the layer above.
- **Validation on three or more physical machines.** Host migration and rejoin were validated on two machines and with several processes on one machine. The scenario with three or more machines, where a node rejoins while a third one keeps playing, is the one most likely to expose the limits listed above.

9.3 Beyond the current scope

- **Security.** If the system were to leave a trusted LAN, the Lobby could issue a signed token to each participant and every message could be authenticated with it, while TLS would protect the two WebSocket channels. Neither changes the protocol: they add a check at the boundary of each node.
- **Gameplay.** More levels, more power-up effects, audio and configurable settings. The mechanics were deliberately kept minimal, so that the difficulty of the project would remain in the coordination between the nodes.