Distributed Bomberman Game

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

Enrico Ferraiolo
enrico.ferraiolo2@studio.unibo.it

Daniele Romanella
daniele.romanella@studio.unibo.it

February 28, 2026

Abstract

We present a distributed, real-time Bomberman system designed for multiplayer gameplay with scalability and fault tolerance. The architecture is split into two layers: a hub layer responsible for matchmaking and cluster coordination, and a room layer that executes the authoritative game loop. The hub server adopts an AP-oriented design, using UDP-based gossip to disseminate room availability and peer liveness without centralized coordination.
The room server is CP-oriented, running a deterministic tick loop over a TCP and Protocol Buffers protocol to ensure consistent state across clients. The client is protocol-driven and renders the game as an ASCII interface.
The system supports transient failure handling through bounded reconnection windows and state snapshotting, keeping matches stable under brief network interruptions. A Kubernetes-based deployment is used for hub scalability and room provisioning. The result is a coherent distributed design that balances availability with strong consistency at the game engine.

Disclaimer

During the preparation of this work, the authors used LLMs, to validate the architectural ideas and design decisions related to their contribution to the project. After using these tools, the authors reviewed and edited the content as needed and takes full responsibility for the content of the final report/artifact.

1 Concept

1.1 Product Type

The product developed is a **Distributed Command-Line Interface (CLI) Game**. The application is designed as a real-time multiplayer system where users interact with the game environment through a terminal-based interface using keyboard inputs. The architecture is distributed, and designed to be executed across a cluster of distinct machines.
Every player can join from their computer without any type of network installation.

1.2 Use Case Description

The software facilitates a real-time multiplayer gaming experience.

Regarding data persistence, the system is designed to be **ephemeral**. It does not rely on long-term persistent storage for game history. Instead, the game state is transient, existing only for the lifecycle of a match.

The system defines a single user role: the **Player**, an entity responsible for establishing a connection to the server, issuing commands (e.g., movement, placement of bombs), and rendering the received game state updates on the local CLI.

1.3 Need for Distribution

The adoption of a distributed architecture is a fundamental requirement for this project, driven by the inherent nature of a real-time multiplayer environment. The system implements a **two-tier architecture**, distinguishing between a *Hub Server* and distinct *Room Servers*.

The necessity for distribution is justified by the following factors:

- **Geographically Distributed Environments**: The primary driver for distribution is the need to connect users located in different physical locations. The system must synchronize the game state across a network of heterogeneous client devices, ensuring a consistent real-time experience despite physical separation and network latency.
- **Scalability and Resource Sharing**: Distribution allows the system to decouple matchmaking from game execution. By employing a multi-server approach (Hub vs. Rooms), the computational load is spread across multiple components rather

than being concentrated on a single machine. This ensures that the system can scale horizontally, simply adding new k8s nodes to the cluster, supporting a higher number of concurrent matches without degrading performance.

- **Fault Tolerance**: The two-tier structure enhances system reliability through isolation. A failure within a specific *Room Server* is contained to that single match, preventing it from cascading to the *Hub Server* or other active games. This ensures that the matchmaking service remains available for new users even under partial failure conditions.

1.3.1 Hub Server

The Hub Server acts as the entry point for the distributed system. From a CAP theorem perspective [1, 7], it is **AP-oriented**: its primary goal is to guarantee Availability and Partition Tolerance, accepting eventual consistency across hub instances.

Multiple Hub Server instances form a peer-to-peer overlay network, where each hub maintains a local view of the cluster state (known peers, active rooms, player counts). Hubs synchronize this state through a gossip protocol, handling two categories of events: peer lifecycle (join, leave, failure detection) and room lifecycle (activation, player joins, game start, closure).

When a client requests to join a game, the Hub Server consults its local state to find a joinable room and returns the corresponding connection details. If no room is available, the hub triggers the activation of a new room instance. This design ensures that clients always receive a response, even if some hubs in the cluster are temporarily unreachable. The architectural and protocol details are discussed in sections 3.1 and 3.4.

1.3.2 Room Server

The Room Server is the execution unit responsible for managing the authoritative game state of a single match. It receives player actions and applies them deterministically within a tick-based loop, guaranteeing that all players observe the same state evolution.

From a CAP perspective, the Room Server is **CP-oriented**: it prioritizes strong consistency at the granularity of a match, even at the cost of temporarily excluding clients with unstable connections.

Clients connect via persistent TCP channels and exchange compact Protocol Buffers messages. The Room Server provides limited fault tolerance through periodic state snapshotting and bounded reconnection windows, allowing clients to rejoin after transient disconnections. The design and implementation details are discussed in sections 3.1 and 3.5.

2 Requirements Elicitation and Analysis

2.1 Hub Server

This section outlines the specific requirements for the *Hub Server* component and the underlying distributed architecture. This component is responsible for managing the cluster of hubs, coordinating peer discovery, synchronizing room state across instances, and serving as the entry point for client matchmaking.

2.1.1 Functional Requirements

1. **Peer Discovery and Membership**
   - The system must support two discovery modes: *manual* (statically configured peers - used for development) and *Kubernetes-based*.
   - Each hub must maintain a local membership list of known peers, tracking their status (alive, suspected, dead).
   - *Acceptance Criterion:* When a new hub instance starts and sends a `PEER_JOIN` message, all reachable hubs add it to their local peer list within a bounded number of gossip rounds.
2. **Gossip-Based State Synchronization**
   - Hubs must propagate cluster events (peer lifecycle changes, room state transitions) using a gossip protocol [3] over UDP, with Proto Buf as the serialization format.
   - Message forwarding must use a fanout-bounded strategy: each hub forwards a received message to at most `HUB_FANOUT`[1] peers, preventing message storms while ensuring dissemination.
   - Duplicate messages must be detected and discarded using nonce based information.
   - *Acceptance Criterion:* A `ROOM_ACTIVATED` event originating from one hub is received by all alive hubs in the cluster.
3. **Failure Detection**
   - Each hub must run a periodic failure detector that monitors peer liveness based on heartbeat timestamps.
   - A peer that has not sent any message within `SUSPECT_TIMEOUT` seconds must be marked as *suspected*. A peer silent for more than `DEAD_TIMEOUT` seconds must be marked as *dead* and removed from the active peer list.
   - *Acceptance Criterion:* If a hub process is killed, the remaining hubs detect the failure and mark the peer as dead within `DEAD_TIMEOUT` seconds. Suspected and dead events are broadcast to the cluster.

[1] `HUB_FANOUT` = 4 by default

4. **Room Lifecycle Management - Hub POV**
    - The Hub must manage the full room lifecycle: activation (`ROOM_ACTIVATED`), game start (`ROOM_STARTED`), and closure (`ROOM_CLOSED`).
    - When a client requests matchmaking, the hub must return a joinable room from its local state. If none exists, it must activate a new room instance.
    - *Acceptance Criterion:* A client calling the matchmaking endpoint always receives either an existing joinable room or a newly activated one, and the corresponding event is propagated to all hubs.
5. **Client Matchmaking Endpoint**
    - The Hub must expose an HTTP endpoint that clients use to obtain room connection details (address, port).
    - *Acceptance Criterion:* The endpoint returns a valid room reference with connection parameters. If no room is available and activation fails, the endpoint returns an appropriate error response.

2.1.2 Non-Functional Requirements

1. **Availability (AP Orientation)**
    - The system must remain available to clients even when some hub instances are unreachable. Each hub operates on its local state and does not require consensus to serve matchmaking requests.
    - *Acceptance Criterion:* If one or more hubs in the cluster crash, the remaining hubs continue to accept client requests and manage rooms without interruption.
2. **Eventual Consistency**
    - The local state of each hub (peer list, room registry) must converge to a consistent view, given that gossip messages are eventually delivered.
    - If a hub does not receive a room activation message for some short temporary network failure (i.e. collisions), the room is not returned by that hub, in addition the room will be marked as not joinable in a finite time.
    - *Acceptance Criterion:* After a room activation event, all alive hubs reflect the new room in their local state within a bounded number of gossip rounds (dependent on cluster size and fanout), if network failures does not occur.
3. **Scalability**
    - The gossip protocol must scale with the number of hubs. The fanout-bounded forwarding ensures that per-node network load grows logarithmically rather than linearly with cluster size.

- *Acceptance Criterion:* Increasing the number of hubs from 4 to 10 does not cause message storms or significant increase in per-hub message processing load.

4. **Resilience to Message Loss**
   - Since the gossip protocol operates over UDP, occasional message loss is expected. The system must tolerate lost messages through redundant forwarding paths inherent to the gossip protocol.
   - *Acceptance Criterion:* The system continues to function correctly even if individual gossip packets are dropped, as redundant paths ensure eventual delivery, in case of all knowing peers [2]

2.1.3 Implementation Constraints

- **Serialization Format**
  - All gossip messages are serialized using **Protocol Buffers** (protobuf) [8].
  - *Justification:* Protobuf provides compact binary encoding (critical for UDP datagrams with size constraints), strong typing via `.proto` schema definitions, and cross-language compatibility for future extensibility.
- **Deployment**
  - The Hub Server is designed to run as a **Kubernetes StatefulSet**, where each pod has a stable hostname (e.g., `hub-0`, `hub-1`, etc) used to derive the hub index.
  - *Justification:* StatefulSets provide stable network identities, which simplifies peer addressing and index assignment in the gossip protocol.

2.2 Room Server

This section outlines the specific requirements for the *Room Server* component and the underlying distributed architecture. This component is responsible for maintaining the authoritative game state, processing player inputs, and broadcasting updates to clients.

2.2.1 Functional Requirements

1. **Game Loop Execution**
   - The system must run a centralized game loop (tick-based) that updates the game state at a fixed frequency (e.g., 30 ticks per second).
   - *Acceptance Criterion:* The server logs show a consistent tick duration, and game physics (bomb timers, movement) progress uniformly.

2. **Input Processing**

[2] A situation where all peers know each other

- The Room Server must accept concurrent inputs (movement, bomb placement) from up to 4 connected clients.
- *Acceptance Criterion:* When multiple clients send commands simultaneously, the server processes them sequentially within the same tick without dropping data.

3. **State Broadcasting**
   - The system must broadcast the full game state to all connected clients after every tick.
   - *Acceptance Criterion:* All connected clients receive a game state packet after each tick completion.

4. **Dynamic Player Management**
   - The system must handle the sudden disconnection of a player by removing their entity from the game board or marking them as "inactive".
   - *Acceptance Criterion:* If a client process is killed, the server detects the broken pipe, logs the event, and the remaining players see the disconnected avatar disappear.

2.2.2 Non-Functional Requirements

1. **Consistency**
   - The game state must be strictly consistent across all clients. The server is the single source of truth.
   - *Acceptance Criterion:* Clients never observe conflicting states (e.g., one client sees a bomb while another does not) and all clients see the same game state after each tick, as the server broadcasts the authoritative snapshot.

2. **Responsiveness**
   - The end-to-end latency (input $\rightarrow$ server processing $\rightarrow$ state update) should be minimized to ensure playability.
   - *Acceptance Criterion:* The game feels responsive to keyboard inputs on a standard LAN connection.

3. **Fairness**
   - The order of event processing must be fair: game actions are processed in a deterministic, and consistent manner.
   - *Acceptance Criterion:* Simultaneous conflicting actions are resolved consistently by the server logic (e.g. First-In-First-Served Queue).

2.2.3 Implementation Constraints

- **Programming Language**
  - This component had been implemented in **Python**.
  - *Justification:* Python offers high-level abstractions for socket programming and threading, allowing for rapid prototyping of distributed logic.

3 Design

3.1 Architecture

3.1.1 Hub Server

The Hub Server adopts a **peer-to-peer overlay** architectural style, where each hub instance is both a client and a server within the gossip protocol. No hub acts as a coordinator or leader: every instance maintains its own local copy of the cluster state and makes autonomous decisions.

This choice is motivated by the AP orientation identified in the requirements. A leader-based architecture would provide strong consistency but would sacrifice availability during leader elections or network partitions. Since the Hub's primary role is matchmaking (returning a joinable room to clients), temporary inconsistencies between hubs are tolerable and self-correcting via gossip convergence or system evolving.

The client-facing layer uses a standard **request-response** pattern over HTTP (via FastAPI), decoupled from the internal gossip communication. This separates the concerns of client interaction (synchronous, HTTP) from cluster coordination (asynchronous, UDP gossip).

3.1.2 Room Server

The Room Server adopts a **Client-Server (Authoritative Server)** architectural style. The Room Server acts as the centralized authority for a specific game match. Clients do not communicate with each other directly; instead, they send their intentions (inputs) to the Room Server, which computes the true state of the world and broadcasts it back.

This choice is strictly motivated by the **CP-orientation** required for game logic. In a fast-paced game like Bomberman, concurrent actions (e.g., two players attempting to place a bomb in the same spot simultaneously) must be resolved deterministically. An authoritative server ensures a single source of truth, preventing desynchronization and mitigating client-side cheating.

3.2 Infrastructure

The infrastructure comprises the following components:

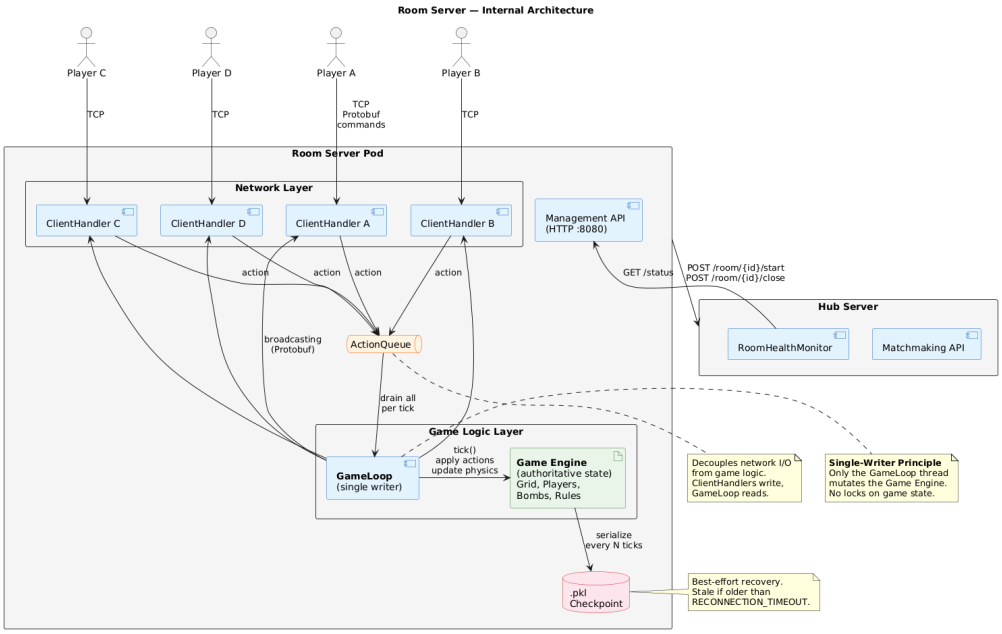

Figure 1: Room Server internal architecture.

Hub Instances. Each hub runs as a Kubernetes `StatefulSet` pod. The StatefulSet guarantees stable hostnames. Each hub pod exposes two network interfaces: HTTP API and UDP Gossip socket.

Room Instances. Each Room Server runs as an independently from HubServer. Room Servers comes from a Game Port (TCP) where the players connect to the game and a HTTP port that is used from HubServers for health checks.

An overview of room is shown in fig. 1

3.3 Modelling

3.3.1 Hub Server

Domain Entities.

- **HubPeer**: represents a known hub instance in the cluster. Each peer has an `index` (derived from hostname), a `ServerReference` (address + port), a `status` (`alive`, `suspected`, `dead`), a monotonically increasing `heartbeat` counter (used for duplicate detection), and a `last_seen` timestamp (used for failure detection).
- **Room**: represents a game session. Each room has a `room_id`, an `owner_hub_index` (the hub that created it), a `status` (following the lifecycle `DORMANT` → `ACTIVE` → `PLAYING` → `CLOSED/DORMANT`), a `player_count` with a configurable `max_players` cap, and network details (`external_port`, `internal_service`).

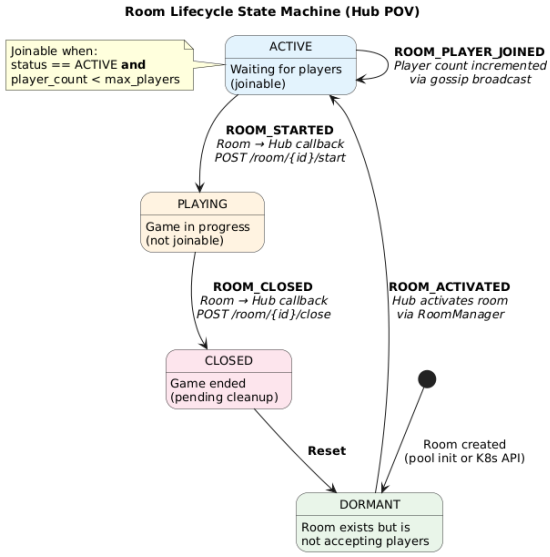

Figure 2: Room lifecycle

- **HubState**: the aggregate root that holds the local view of the cluster. It contains a list of `HubPeer` instances (indexed by peer index, with `None` gaps for unknown indices) and a dictionary of `Room` instances (keyed by `room_id`). All access is synchronized via a `threading.RLock`.
- **ServerReference**: a value object representing a network endpoint (address + port).
- **RoomStatus**: an enumeration defining the room lifecycle states: `DORMANT` (created but not accepting players), `ACTIVE` (waiting for players), `PLAYING` (game in progress), and `CLOSED` (ended, pending cleanup). This enum drives the joinability logic and state transition rules throughout the system.
- **GossipMessage**: the protocol envelope for all inter-hub communication. Each message carries a `nonce` (monotonic per origin), `origin` and `forwarded_by` indices, a `timestamp`, an `event_type` discriminator, and exactly one `oneof payload` matching the event type. Defined as a Protocol Buffers message.
- **Event Payloads**: a set of value objects, one per event type, carrying event-specific data. Examples include `RoomActivatedPayload` (room_id, owner_hub, external_port, external_address), `PeerJoinPayload` (joining_peer index), and `RoomPlayerJoined` (room_id). Each payload is a Protocol Buffers message nested within the `GossipMessage` envelope.

The Room lifecycle is shown in figure fig. 2

Domain Events. The gossip protocol defines two categories of events, encoded as `EventType` in the Protocol Buffers schema:

- **Peer lifecycle events**: `PEER_JOIN` (a hub announces itself), `PEER_LEAVE` (graceful shutdown), `PEER_ALIVE` (liveness declaration in response to suspicion), `PEER_SUSPICIOUS` (a hub suspects another is unresponsive), `PEER_DEAD` (a hub declares another dead after timeout).
- **Room lifecycle events**: `ROOM_ACTIVATED` (a room starts accepting players, includes owner hub, external port, and address), `ROOM_PLAYER_JOINED` (player count increment), `ROOM_STARTED` (game begins, room no longer joinable), `ROOM_CLOSED` (game ended, room returns to dormant).

Messages. All gossip messages share a common envelope (`GossipMessage`) containing:

- `nonce`: monotonically increasing counter per origin hub, used for duplicate detection and message ordering.
- `origin`: the index of the hub that originally created the message.
- `forwarded_by`: the index of the hub that last forwarded the message (changes at each hop).
- `timestamp`: message creation time.
- `event_type`: enum discriminator for the payload.
- One of payload field carrying the event-specific data.

System State. Each hub maintains:

- A **peer list**: all known hub instances with their status, heartbeat counter, and last seen timestamp. This is the hub's local membership view.
- A **room registry**: all known rooms (both local and remote) with their lifecycle status and player count. This is the hub's local view of available game sessions.
- A **nonce counter**: the last used nonce for outgoing messages, ensuring monotonicity.

3.3.2 Room Server

Domain Entities.

- **Game Engine**: The aggregate root representing the entire match state. It contains the grid, the list of players, and active bombs.
- **Player**: Represents an in-game player, tracking coordinates player's ID, position, alive/dead status, and if they are the owner of a bomb.

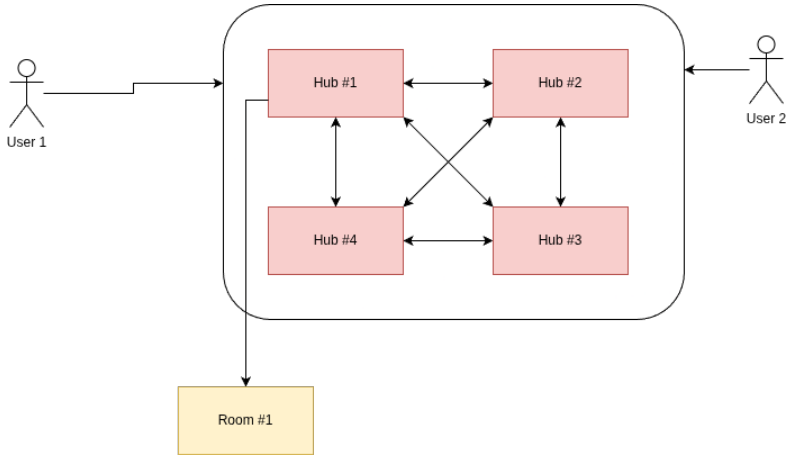

Figure 3: Client interaction with the Hub cluster. Clients connect to a single entry point without knowledge of which hub instance serves their request.

- **Bomb**: Represents a bomb placed on the grid, tracking its position, owner, explosion timer, and blast range.

Entity Mapping. The authoritative `Game Engine` resides exclusively in the Room Server's memory. Clients only hold a string-based representation of this model. When a domain event occurs (e.g., a bomb explodes), the Room Server calculates the outcome (destroyed walls, killed players) and broadcasts a new snapshot to all clients.

Domain Events and Messages. Communication between Client and Room Server relies on compact Protocol Buffers exchanged over a TCP stream.

- **Commands (Client → Server)**: Represent player intentions (actions).
- **State Updates (Server → Client)**: The server periodically broadcasts a snapshot message, ensuring clients continually synchronize with the authoritative state.

3.4 Interaction

3.4.1 Hub Server

Client → Hub (HTTP). The client sends a `POST /matchmaking` request. The hub consults its local `HubState` for a joinable room (status `ACTIVE` and `player_count < max_players`). If found, the hub increments the player count, broadcasts a `ROOM_PLAYER_JOINED` gossip message, and returns the room's address and port. If no joinable room exists, the hub asks its `RoomManager` to activate a dormant room (or create a new one in K8s mode); upon success, it broadcasts `ROOM_ACTIVATED` and returns the new room's details. If activation fails, the hub returns HTTP 503. The interaction between client and hub is shown in figure 3

Room → Hub (HTTP callbacks). The Room Server notifies the hub of lifecycle transitions:

- `POST /room/{id}/start`: triggers a `ROOM_STARTED` gossip broadcast and sets the room status to `PLAYING`.
- `POST /room/{id}/close`: triggers a `ROOM_CLOSED` gossip broadcast and sets the room status to `DORMANT`.

Hub ↔ Hub (UDP Gossip). When a hub receives a gossip message via `HubSocketHandler`, the following pipeline executes:

1. **Resolve sender**: in K8s mode, the sender reference is recalculated from the `forwarded_by` index (because UDP source addresses may differ from the pod's DNS name).
2. **Ensure peers exist**: the origin and forwarder are added to the peer list if not already known.
3. **Heartbeat check**: the message's nonce is compared against the stored heartbeat for the origin peer. If the nonce is not newer, the message is a duplicate and is discarded. The forwarder is marked as alive regardless.
4. **Process payload**: the event-specific handler is invoked (e.g., `_handle_room_activated` adds a room to local state).
5. **Forward**: the message is forwarded to a random subset of alive peers (bounded by `HUB_FANOUT`), with the `forwarded_by` field updated to the current hub's index.

Full logic is shown in figure fig. 4

Hub → Room (Health Check). The `RoomHealthMonitor` periodically sends `GET /status` HTTP requests to each `ACTIVE` room's internal service. If the room does not respond with status `WAITING_FOR_PLAYERS`, the hub marks it as unhealthy: local rooms are transitioned to `PLAYING` (with a gossip broadcast), remote rooms are removed from local state.

3.4.2 Room Server

Client ↔ Room (TCP Stream). Once a client connects to the Room Server, the interaction follows an **Asynchronous Input / Synchronous Broadcast** pattern:

1. **Input Collection**: Clients send actions messages at varying frequencies based on user input. The Room Server asynchronously receives these and places them into a thread-safe input queue.
2. **Tick-Based Broadcast**: The Room Server does not respond to inputs immediately [10]. Instead, at a fixed frequency (e.g., 10 Hz), a dedicated game loop thread processes all queued inputs, updates the game state, and broadcasts the resulting snapshot to all connected clients simultaneously.

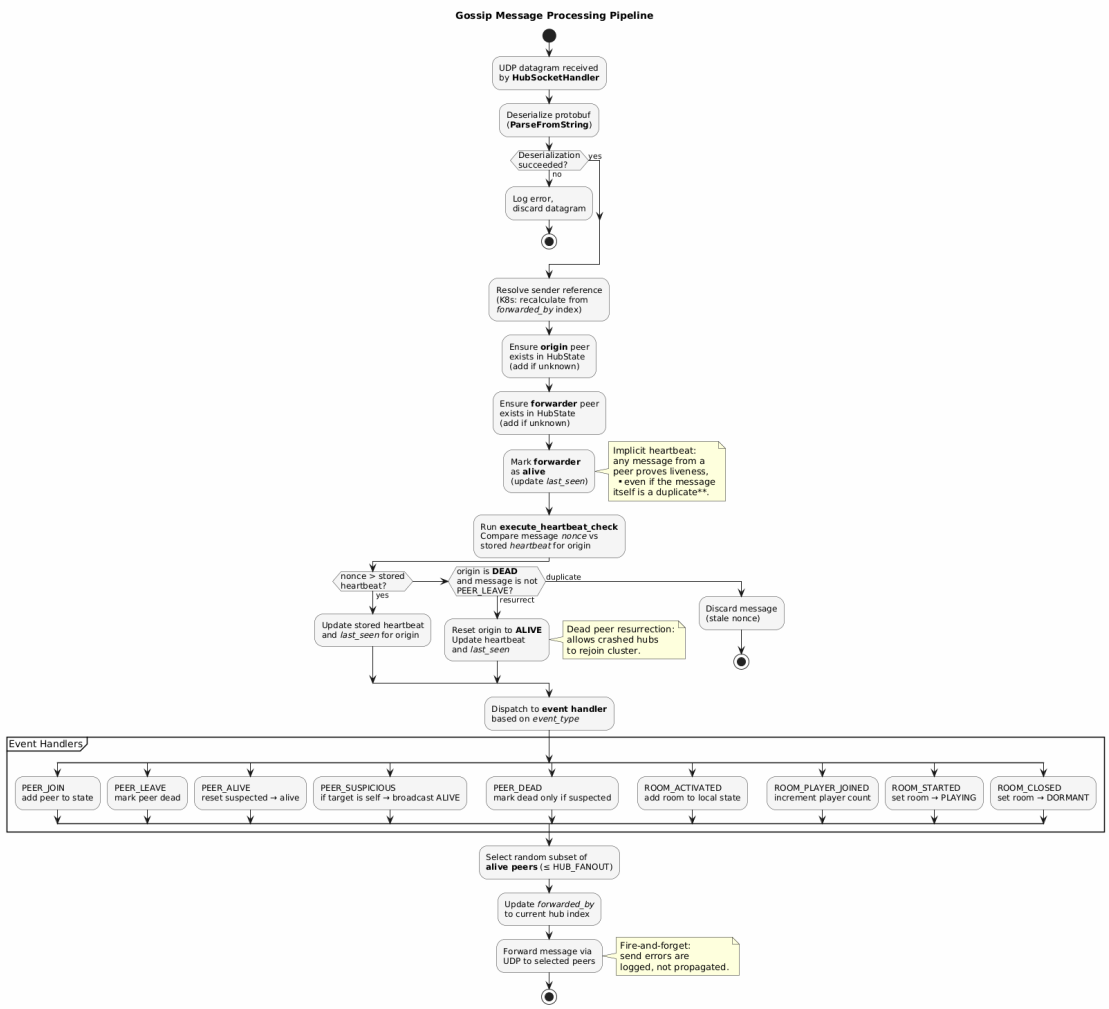

Figure 4: Gossip message processing pipeline. The forwarder is marked alive regardless of whether the message is a duplicate, providing implicit heartbeating.

Room → Hub (HTTP). The Room Server uses a **Webhook** pattern to notify the Hub of major lifecycle shifts, sending HTTP `POST` requests to the Hub when the required number of players has connected (`/start`) and when the match ends (`/close`).

3.5 Behaviour

3.5.1 Hub Server

HubServer (Stateful, Coordinator). The `HubServer` class is the central coordinator. On initialization, it creates and starts all subsystems (socket handler, failure detector, peer discovery monitor, room manager, room health monitor). It reacts to incoming gossip messages via callback dispatch (`_on_gossip_message`), to failure detector callbacks (`_on_peer_suspicious`, `_on_peer_dead`), to peer discovery triggers (`_discovery_peers`), and to room health alerts (`_on_room_unhealthy`). On shutdown (`stop()`), it broadcasts a `PEER_LEAVE` message and cleanly stops all subsystems.

HubState (Stateful, Passive). `HubState` is a thread-safe container. It does not initiate any actions; it only responds to queries and mutations from the `HubServer` and background threads. All methods acquire the internal `RLock` before accessing shared data. Notable behavior: `execute_heartbeat_check` implements the duplicate detection logic. It returns `True` only if the received nonce is strictly greater than the stored one, or if a dead peer is returning (resurrection). A special case prevents propagation of `PEER_LEAVE` messages for already-dead peers.

FailureDetector (Stateful, Active). Runs a background thread that periodically iterates over all known peers (excluding self). For each peer, it computes the silence duration (`now - last_seen`). The state machine transitions are:

- `alive` → silence > `SUSPECT_TIMEOUT` `suspected` (triggers `on_peer_suspected` callback)
- `alive` or `suspected` → silence > `DEAD_TIMEOUT` `dead` (triggers `on_peer_dead` callback)
- `dead` → no further transitions (peer is ignored)

Note: an alive peer can transition directly to dead if the silence exceeds `DEAD_TIMEOUT` without an intermediate suspected check.

PeerDiscoveryMonitor (Stateful, Active). Runs a background thread that periodically counts alive peers (excluding self). If the count is below the configured fanout, it triggers the `_discovery_peers` callback, which sends a `PEER_JOIN` message to a randomly selected peer. This ensures the hub continuously attempts to maintain a sufficient peer count for effective gossip dissemination.

RoomManager (Stateful, Passive). Manages the local pool of room instances. In K8s mode, it creates pods and NodePort services via the Kubernetes API. When `activate_room()` is called and no dormant room is available, the K8s implementation dynamically creates a new room (the local implementation cannot). On startup, K8s mode recovers existing room pods from a previous run (`_recover_existing_rooms`).

RoomHealthMonitor (Stateful, Active). Runs a background thread that periodically checks all `ACTIVE` rooms via HTTP. It only checks rooms with a known `internal_service` (skipping remote rooms without internal addressing). When a room fails the health check, the behavior depends on ownership: local rooms are transitioned to `PLAYING` (assuming the game has started), remote rooms are removed from state (assuming the remote hub will handle it).

HubSocketHandler (Stateless, Active). Manages the UDP socket. The listener thread dispatches each received datagram to a bounded thread pool (`ThreadPoolExecutor`) to avoid blocking the receive loop while bounding concurrency.

3.5.2 Room Server

GameLoop (Stateful, Active). The core behavior of the Room Server is driven by a single, stateful `GameLoop` thread. This thread dictates the passage of time in the game. It is responsible for popping inputs from the network queues, updating the `Game Engine` (e.g., decrementing bomb timers, moving players), and broadcasting the new state. It strictly enforces a `sleep` interval at the end of each cycle to maintain a consistent tick rate.

ClientHandler (Stateful, Passive). For every connected client, the Room Server spawns a `ClientHandler` thread. Its sole behavior is to block on the TCP socket, deserialize incoming Protocol Buffer messages, and append them to the main `GameLoop`'s input queue. It does not update the game state directly, preventing concurrency anomalies.

3.6 Data and Consistency Issues

3.6.1 Hub Server

Stored Data. The Hub Server does not use persistent storage. All state is held in memory within the `HubState` object. If a hub crashes and restarts, it reconstructs its peer list through gossip (receiving `PEER_JOIN`/`PEER_ALIVE` messages from other hubs) and its room list through the K8s API (`_recover_existing_rooms` queries existing pods). This design choice eliminates the need for a database and simplifies the deployment, at the cost of a brief inconsistency window during recovery.

Shared Data. The peer list and room registry are conceptually shared across all hubs, but each hub maintains its own copy. There is no shared database or distributed data

structure. Consistency is achieved through gossip propagation: when a hub modifies its local state (e.g., activating a room), it broadcasts the change to the cluster.

Consistency Model. The system provides **eventual consistency**. After a state change originates at one hub, the gossip protocol propagates it to all alive hubs through fanout-bounded forwarding. The convergence time depends on the cluster size, fanout parameter, and gossip round interval. Duplicate messages are discarded via per-origin nonce tracking, ensuring idempotent processing.

Possible temporary inconsistencies include:

- Two hubs simultaneously activating a room for the same matchmaking request (if the `ROOM_ACTIVATED` message hasn't propagated yet). This results in two active rooms, which is acceptable since it simply increases capacity.
- Player count divergence: if a `ROOM_PLAYER_JOINED` message is lost, some hubs may have a stale count. This is tolerable because the Room Server itself is the authoritative source for whether a game can start.

Concurrent Access. All access to `HubState` is serialized via a `threading.RLock` (reentrant lock). This is necessary because multiple threads concurrently access the state: the HTTP request handler thread, the gossip message handler threads (one per received datagram), the failure detector thread, the peer discovery monitor thread, and the room health monitor thread. The reentrant property is used because some methods (e.g., `mark_forward_peer_as_alive`) call other locked methods internally (e.g., `add_peer`).

3.6.2 Room Server

Stored Data. The Room Server persists the game state to disk. The `Game Engine` class, which encapsulates the entire match state, is periodically serialized and saved as a Python pickle file (`.pkl`). While active gameplay reads and writes to RAM, dumping the state to a `.pkl` file provides a persistent snapshot of the game. This file-based storage acts as a localized persistence mechanism, allowing the server to maintain a recoverable state of the match.

Concurrent Access and Consistency. The Room Server enforces **Strong Consistency**. Since multiple `ClientHandler` threads receive data concurrently, all shared access to the game state is serialized. Instead of relying on fine-grained locks which could cause deadlocks, the architecture uses a **Single-Writer principle**: the `GameLoop` thread is the *only* entity allowed to mutate the `Game Engine` and to safely execute the disk I/O operations to save the `.pkl` file. Concurrent writes (client inputs) are marshaled via thread-safe queues, ensuring the state is never pickled while partially modified.

Shared Data. The authoritative server model ensures that conflicting requests are resolved based on the order they are pulled from the queue during the tick.

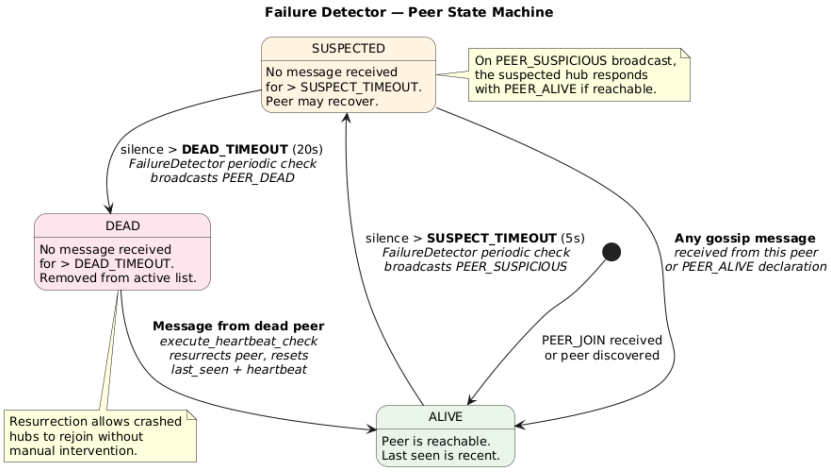

Figure 5: Failure detector state machine.

3.7 Fault-Tolerance

3.7.1 Hub Server

Data Replication. Each hub holds a full replica of the cluster state (peer list + room registry). Replication is achieved through the gossip protocol: every state-changing event is broadcast to the cluster. This is a form of **optimistic replication**, where each replica applies updates locally without coordinating with others, and conflicts are resolved by convergence (last-write-wins based on nonce ordering).

Heartbeating and Failure Detection. The `FailureDetector` implements a two-phase timeout-based scheme:

- **Suspect phase** (`SUSPECT_TIMEOUT`, default 5 seconds): if a peer has not been heard from within this interval, it is marked as `suspected`. A `PEER_SUSPICIOUS` message is broadcast, giving the suspected peer a chance to respond with a `PEER_ALIVE` declaration.
- **Dead phase** (`DEAD_TIMEOUT`, default 20 seconds): if the silence continues, the peer is marked as `dead`. A `PEER_DEAD` message is broadcast, and the peer is removed from the active peer list.

The failure detection system is shown in figure fig. 5

The two-phase design avoids premature removal: transient network delays or temporary processing stalls trigger suspicion (which is reversible) rather than immediate removal. The "alive" declaration mechanism allows a falsely suspected peer to rehabilitate itself.

Gossip messages themselves serve as implicit heartbeats: any message received from a peer (regardless of type) updates the forwarder's `last_seen` timestamp via `mark_forward_peer_as_alive`.

Peer Recovery. A dead peer can rejoin the cluster. In `execute_heartbeat_check`, if a message arrives from a peer currently marked as `dead`, the peer's status is reset to `alive` and the message is processed normally. This allows hubs that crashed and restarted to reintegrate without manual intervention.

Room Recovery. In Kubernetes mode, if a hub restarts, the `K8sRoomManager` queries the Kubernetes API for existing room pods labeled with the hub's index. Running or pending pods are recovered into the local room registry, avoiding orphaned rooms.

Error Handling.

- **Network errors**: UDP send failures (DNS resolution, socket errors) are logged but do not crash the hub. The gossip protocol's redundant forwarding paths compensate for individual message losses.
- **Malformed messages**: protobuf deserialization errors are caught in `HubSocketHandler` and logged, preventing invalid data from corrupting hub state.
- **Room health failures**: rooms that fail HTTP health checks are handled gracefully (local rooms transitioned, remote rooms removed from state).
- **Kubernetes API failures**: pod/service creation errors are caught and logged, allowing the hub to continue operating with a reduced room pool.

3.7.2 Room Server

Local Checkpointing. At the Room tier, there is no active state replication across multiple nodes. A match is entirely isolated to a single Pod. However, to mitigate complete data loss, the Room Server implements **local checkpointing**. By periodically serializing and dumping the `Game Engine` state to a `.pkl` file, the system creates a persistent snapshot of the match without the overhead of network replication.

Error Handling and Disconnections.

- **Server Failure and Recovery**: If a Room Server process crashes unexpectedly, the match state is not permanently lost. Upon restart, the server can deserialize the latest `.pkl` snapshot to restore the `Game Engine`. This allows the match to resume from the last saved checkpoint, losing only the transient inputs that occurred after the last disk write. The checkpoint is considered a best-effort recovery mechanism, as it does not guarantee zero data loss, but it significantly reduces the impact of server failures. The saved state is considered invalid if it is older than a configured threshold (e.g., 30 seconds), in which case the server starts a new match instead of attempting recovery.

- **Client Failure**: The `ClientHandler` actively monitors the TCP socket. If a socket throws an `EOFError` or `ConnectionResetError`, the client is considered disconnected. The Room Server updates the player's status within the `Game Engine` and continues the match for the remaining players.
- **Graceful Reconnection**: Because the state is preserved in the `Game Engine` (and backed up in the `.pkl`), the server retains the disconnected player's *User ID* and avatar state. If the player re-establishes the TCP connection within a bounded time window, they can reclaim their previous state (position, alive/dead status) and rejoin the match seamlessly.

3.8 Availability

3.8.1 Hub Server

No Caching. The Hub Server does not implement caching. The local `HubState` already serves as an in-memory replica of the cluster state, making additional caching unnecessary. Every matchmaking request is served from the local state without external lookups.

Load Distribution. The system does not use a traditional load balancer between hubs. Instead, a Kubernetes Service distributes incoming HTTP requests across hub pods. Since each hub holds a full replica of the room registry, any hub can serve any matchmaking request independently, making the distribution stateless and trivially parallelizable.

For gossip traffic, the fanout-bounded forwarding naturally distributes message propagation load: each hub forwards to at most `HUB_FANOUT` peers, randomly selected, preventing any single hub from becoming a bottleneck.

Network Partitioning. Under a network partition, the system behaves as follows (consistent with its AP orientation):

- Each partition continues to operate independently. Hubs in each partition serve matchmaking requests from their local state.
- Hubs in different partitions will diverge: room activations and player joins on one side will not be visible on the other.
- The failure detector will eventually mark unreachable peers as dead.
- When the partition heals, hubs rediscover each other through the `PeerDiscoveryMonitor`. Dead peers that send new messages are automatically resurrected via the heartbeat check logic. Room state converges as new gossip events propagate.
- The main consequence is that during the partition, the same room might be assigned to clients from both sides, or duplicate rooms might be activated. This

is acceptable because the Room Server itself enforces the player cap, and excess rooms simply remain unused.

3.8.2 Room Server

Network Partitioning. Because the Room Server operates on a CP (Consistency/Partition Tolerance) model, it favors consistency over availability. If a network partition isolates a client from the Room Server:

- The server will stop receiving inputs from that client, leaving their player idle.
- Once the TCP's timeout threshold is reached, the server drops the connection to maintain state integrity.

3.9 Security

3.9.1 Hub Server

The Hub Server currently operates under a **trusted environment assumption**: all components run within a private Kubernetes cluster, and no external actors have direct access to the gossip protocol or the hub's HTTP management endpoints.

Authentication and Authorization. No authentication is implemented on any endpoint. The matchmaking endpoint, room lifecycle callbacks, and debug endpoint are all unauthenticated. In a production deployment, the following measures could be added:

- API key or token-based authentication for room-to-hub callbacks (`/room/{id}/start`, `/room/{id}/close`).
- Network-level access control via Kubernetes `NetworkPolicy` to restrict gossip traffic to hub pods only.
- Disabling or restricting access to the `/debug/` endpoint, which exposes internal cluster state.

Gossip Security. Gossip messages are neither authenticated nor encrypted. A malicious actor with access to the cluster network could inject forged gossip messages (e.g., false `PEER_DEAD` declarations to destabilize the cluster). However, since the system assumes an internal Kubernetes network, this risk is accepted. Input validation is performed: malformed protobuf messages are discarded by the socket handler, and invalid peer indices or room IDs do not corrupt the hub state.

Cryptography. No cryptographic mechanisms are used. All communication (HTTP and UDP) is unencrypted. In a production deployment, TLS termination at the ingress level would protect client-to-hub communication, while mutual TLS (mTLS) via a service mesh could secure intra-cluster traffic.

3.9.2 Room Server

Authentication. There is minimal authentication at the Room tier. When a client connects, their first message must contain their *Player ID*. The Room Server verifies this ID against the list of expected players provided by the Game Engine during the game startup phase. If the ID is already taken by another connected client, or if it does not match any expected player, the server rejects the connection.

Authorization (Server-Side Validation). The most critical security mechanism in the Room Server is **State Authorization**. Clients are not authorized to dictate the game state. They cannot send "I am at coordinate (5,5)"; they can only send "I want to move UP". The Room Server validates every request against the game rules (e.g., checking for wall collisions) before applying it, neutralizing client-side manipulation (e.g., teleportation).

Cryptography. Like the Hub Server, the Room Server relies on unencrypted TCP streams. While vulnerable to packet sniffing, it is deemed acceptable for the purpose of this application.

4 Implementation

This section documents the technology-dependent choices made while implementing the design described in section 3.

4.1 Hub Server

4.1.1 Network Protocols

The Hub Server uses two distinct network protocols, chosen to match the communication semantics of each interaction:

- **UDP** for inter-hub gossip communication. UDP is chosen because gossip messages are small (single protobuf-encoded datagrams), stateless, and tolerant to occasional loss. The gossip protocol's redundant forwarding inherently compensates for dropped packets, making TCP's reliability guarantees unnecessary overhead. The socket is bound to `0.0.0.0` on the configured `GOSSIP_PORT`, and each received datagram is handled in a dedicated thread to avoid blocking the receive loop.
- **HTTP** for client-facing and room-facing APIs. HTTP is used because these interactions follow a request-response pattern where the caller needs a confirmation (e.g., the client needs room connection details, the Room Server needs acknowledgment of lifecycle events). The HTTP layer is served by **Uvicorn** [5] (ASGI server) with **FastAPI** [12] as the framework.

4.1.2 Data Representation

- **Protocol Buffers** (protobuf) for all gossip messages. The schema is defined in `messages.proto` and compiled to Python via `protoc`. Protobuf is chosen over JSON for three reasons: (1) compact binary encoding reduces UDP datagram size, (2) the `.proto` schema provides a strict contract between hub implementations, and (3) the `oneof` construct naturally models the event payload discriminated union.
- **JSON** for all HTTP endpoints, handled natively by FastAPI's `response_model` serialization via Pydantic. The matchmaking response (`MatchmakingResponse`) includes `room_address`, `room_port`, and `room_id`.

4.1.3 Database

The Hub Server does not use any database. All state is held in-memory in the `HubState` object. On restart, room state is recovered from the Kubernetes API (querying existing pods), and peer state is reconstructed through gossip protocol convergence. This choice eliminates an external dependency and is consistent with the ephemeral nature of the hub's coordination role.

4.1.4 Authentication and Authorization

No authentication or authorization is implemented. This is documented as a known limitation in section 3.9.

4.1.5 Technological Details

Python. The entire Hub Server is implemented in Python 3.11+. Python's `threading` module is used for concurrent background tasks (failure detection, peer discovery, room health monitoring, gossip message handling). The GIL limits true parallelism, but since all background tasks are I/O-bound (socket operations, HTTP requests, `time.sleep`), this is not a bottleneck.

FastAPI + Uvicorn. The HTTP API is built with FastAPI, chosen for its automatic request validation, OpenAPI documentation generation, and native async support. Uvicorn serves as the ASGI server. The `lifespan` context manager handles the `HubServer` instance lifecycle (creation on startup, graceful shutdown with `PEER_LEAVE` broadcast on termination).

Exposed endpoints:

- `POST /matchmaking`: client entry point, returns room connection details.
- `POST /room/{id}/start` and `POST /room/{id}/close`: room lifecycle callbacks.
- `GET /health` and `GET /ready`: Kubernetes liveness and readiness probes.
- `GET /debug/`: internal state inspection (peers, rooms, configuration).

Protocol Buffers. The gossip schema (`messages.proto`) defines the `GossipMessage` envelope, the `EventType` enum (9 event types across peer and room lifecycles), and the corresponding payload messages. The compiled `messages_pb2.py` module is used for serialization (`SerializeToString`) and deserialization (`ParseFromString`) in the socket handler.

Kubernetes Client. The `kubernetes` Python library (official client) is used by the `K8sRoomManager` to interact with the Kubernetes API. It uses `load_incluster_config()` when running inside a pod, with a fallback to `load_kube_config()` for local development. Operations include creating/deleting pods and services, listing pods by label selector, and reading service details (for NodePort extraction).

Threading Model. The Hub Server runs multiple concurrent threads:

- **Main thread**: runs the Uvicorn/FastAPI event loop, handling HTTP requests.
- **Listener thread**: runs the UDP receive loop in `HubSocketHandler`.
- **FailureDetector thread**: periodic peer liveness checks (configurable interval, default 1 second).
- **PeerDiscoveryMonitor thread**: periodic peer count checks (configurable interval, default 60 seconds).
- **RoomHealthMonitor thread**: periodic room HTTP health checks (every 15 seconds).

All threads access shared state through the `HubState`'s `RLock`. All background threads are started as daemon threads, ensuring they terminate automatically when the main process exits.

4.2 Room Server

4.2.1 Network Protocols

The Room Server utilizes two distinct network protocols to handle interactions:

- **TCP** for client-to-server game interactions. A persistent socket is established between each player and the Room Server. TCP was chosen over UDP for the game loop because it natively guarantees packet ordering and reliability. TCP ensures that a player's commands (e.g., planting a bomb) are never lost and arrive in the correct order.
- **HTTP** for server-to-hub management callbacks. The Room Server embeds a lightweight HTTP server to respond to health checks (`GET /status`) from the Hub, and it acts as an HTTP client to push lifecycle events (`POST /room/{id}/start`, `POST /room/{id}/close`) back to the Hub Server.

4.2.2 Data Representation

In-transit data representation depends on the direction and type of the payload, although the game loop strictly relies on Protocol Buffers:

- **Protocol Buffers (protobuf)** are used exclusively for all real-time game communication over the TCP stream, in both directions:
  - *Client-to-Server*: When a player presses a key, the client encodes the action (e.g., `MOVE_UP`, `PLACE_BOMB`) into a compact protobuf `Packet` message.
  - *Server-to-Client*: Instead of serializing the complex `Game Engine` object and sending it over the network, the Room Server pre-renders the game state into an ASCII string. This string, along with metadata, is packaged into a `GameStateSnapshot` protobuf message. This ensures cross-language interoperability, as the client can be implemented in any language with protobuf support, and it abstracts away the internal Python object structure from the network protocol.
- **JSON** is used for the HTTP management payloads exchanged with the Hub Server.

4.2.3 Authentication and Authorization

- **Authentication**: Minimal authentication is enforced. Upon establishing the TCP connection, the client must send a `JoinRequest` message containing their `player_id`. Connection is accepted if the `player_id` is unique among currently connected clients, matches the expected format, and the room is not full.
- **Authorization**: All connected users share the single "Player" role. However, **State Authorization** is strictly enforced at the logic level. The server does not trust client state; it only authorizes *intent* (e.g., "move left"). If a client attempts an illegal move (e.g., moving through a wall), the `Game Engine` rejects the input, implicitly denying authorization for that action.

4.2.4 Technological Details

Python and Concurrency. The Room Server is implemented in Python. The core concurrency model relies on the standard `threading` and `queue` modules.

- **Thread-Safe Queues**: A `queue.Queue` is used as the primary synchronization primitive between the network layer and the game logic layer. `ClientHandler` threads push protobuf-decoded input events into this queue, and the `GameLoop` thread pops them. This design avoids the need for explicit `Lock` objects around the `Game Engine`.

Threading Model. The Room Server runs the following distinct threads:

- **Main Thread**: Listens on the game port, accepting new TCP connections and spawning a `ClientHandler` for each.
- **ClientHandler Threads** (Up to 4): Dedicated daemon threads that block on `socket.recv()`, unpacking header, reading the payload, deserializing the protobuf command, and placing it in the input queue.
- **GameLoop Thread**: The authoritative heart of the server. It runs an infinite loop while the state is `PLAYING` or `WAITING_FOR_PLAYERS`. It processes the input queue, steps the engine forward, maps the state to a Protobuf snapshot, and iterates through the list of active client sockets to send the updated game state. It dynamically yields via `time.sleep()` to maintain a steady and fixed tick rate.
- **Management API Thread**: Respond to the Hub's health checks.

Docker and Kubernetes Integration. The Room Server is packaged in a Docker container. section 6.2

The overview of the implementation is shown in figure fig. 6

5 Validation

5.1 Automatic Testing

5.1.1 Hub Server - Unit test

The Hub Server unit test suite covers all source files through a one-to-one correspondence: each class under test has a dedicated test file (e.g., `test_hub_state.py` tests `HubState.py`). Every test verifies an observable behavior or state transition, not implementation details such as log output or internal method call counts.

All tests use `pytest` as the test runner, with `unittest.mock` for dependency isolation. Infrastructure components (UDP sockets, Kubernetes API, HTTP requests) are mocked at the boundary, allowing the tests to exercise the full business logic of each component without network access or external services. Within each file, tests are grouped by logical concern into test classes.

Test Categories and Rationale. The unit tests cover the following categories of business logic, each motivated by specific requirements from section 2:

1. **Hostname Parsing and Initialization** (*Req: Peer Discovery*). Validates that hub index derivation from hostnames works correctly across formats (`hub-0`, `hub-0.local`, `hub-99.svc.cluster.local`), and rejects malformed inputs (whitespace, non-numeric indices, wrong prefixes). Verifies that invalid configuration (zero/negative fanout, missing environment variables) is rejected at startup.

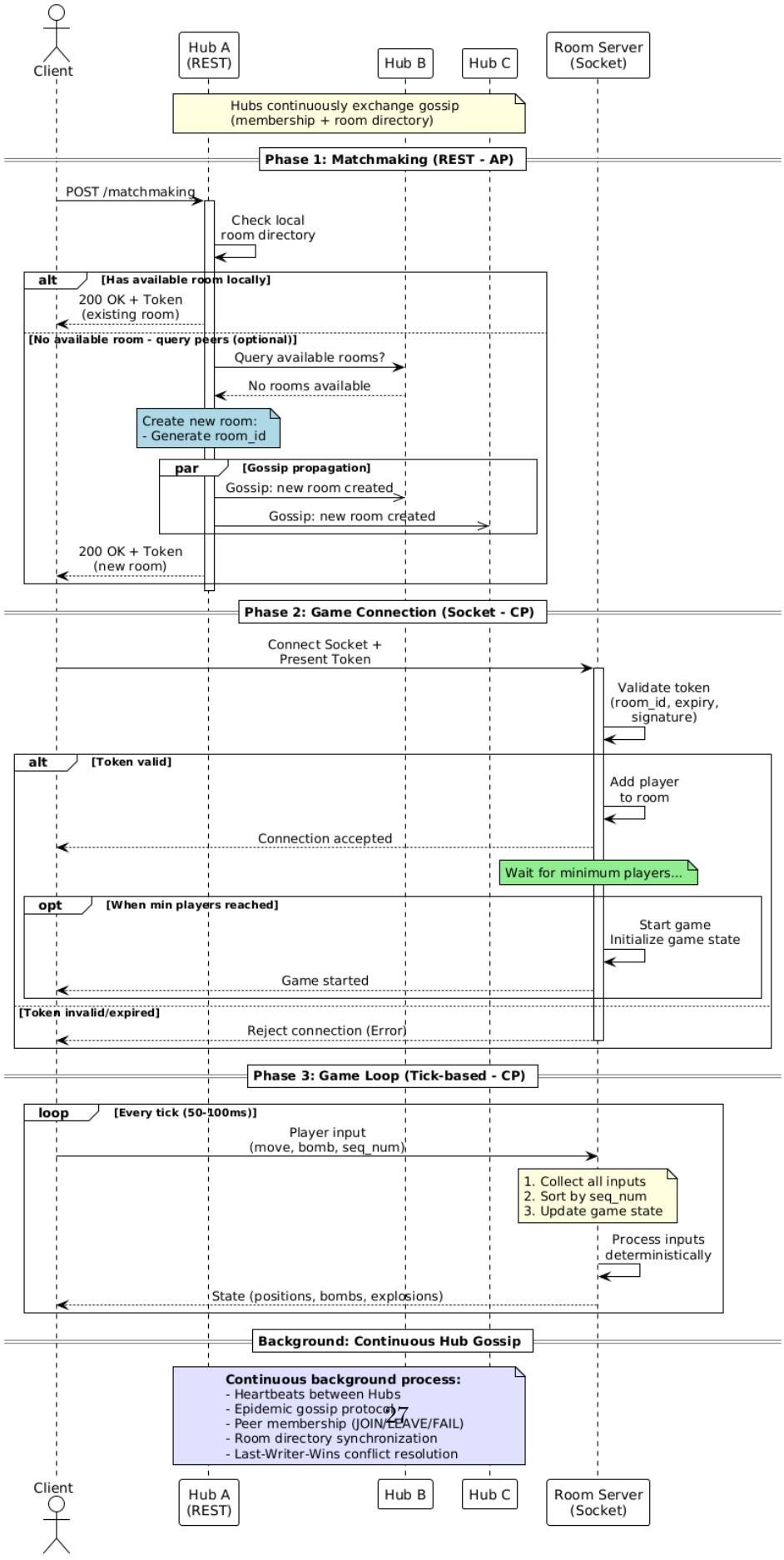

Figure 6: Matchmaking process

2. **Gossip Message Processing Pipeline** (*Req: Gossip-Based State Synchronization*). Tests the full receive pipeline: duplicate detection via nonce comparison, heartbeat update, payload dispatch to the correct handler, and message forwarding. Verifies that stale messages (nonce $\leq$ stored heartbeat) are silently discarded. Covers all 9 event types through the dispatch mechanism.

3. **Peer Lifecycle Handlers** (*Req: Peer Discovery, Failure Detection*).

4. **Failure Detection Logic** Tests the two-phase timeout state machine with controlled time mocking:

5. **Room Lifecycle Management** Verifies the full room lifecycle through gossip: Tests the matchmaking logic: returns an existing joinable room with incremented player count, activates a new room when none available, returns `None` when activation fails.

6. **Room Health Monitoring** (*Req: Failure Detection*). Tests HTTP health check logic: healthy rooms (correct status code + expected status) pass, rooms returning wrong status codes, unexpected statuses, timeouts, or connection errors are flagged as unhealthy. Verifies differentiated handling: local unhealthy rooms transition to PLAYING (with gossip broadcast), remote unhealthy rooms are removed from state. Checks that only ACTIVE rooms with known internal services are health-checked.

7. **State Management** (*Req: Gossip-Based State Synchronization, Eventual Consistency*). Covers peer list operations (add, remove, get with gaps, exclusion filters), heartbeat check logic (duplicate detection, dead peer resurrection, leave message suppression for already-dead peers), and room registry operations (add, get, status transitions, active room lookup by joinability).

8. **Input Validation and Edge Cases** (*Req: general robustness*). Validates domain entity invariants: negative indices rejected, invalid status strings rejected, negative heartbeats rejected, negative player counts rejected, player count exceeding max rejected. Socket handler validates callback signature (parameter count, callable check, None check). Room joinability is parametrized across all status/player-count combinations, including boundary conditions (exactly full, zero max players).

9. **Peer Discovery** (*Req: Peer Discovery and Membership*). Verifies discovery behavior per mode: manual mode hub-0 does not send (it is the bootstrap node), non-zero hubs in manual mode send to a random peer, K8s mode sends with DNS-based addressing. PeerDiscoveryMonitor triggers callback when alive peer count is below fanout.

10. **Message Forwarding** (*Req: Gossip-Based State Synchronization*). Verifies that forwarded messages have the `forwarded_by` field updated to the current hub's index. Tests server reference calculation for both manual (localhost + port offset) and K8s (DNS-based) modes.

Unit Test Bug Discovery. The unit testing process also identified bugs in the initial implementation:

- Peer equality comparison in `ServerReference` used `other.__class__ != ServerReference` instead of `isinstance`, breaking subclass comparisons.
- `execute_heartbeat_check` did not update `last_seen` when resurrecting a dead peer, causing the failure detector to immediately re-suspect the returned peer.
- Room status validation allowed constructing a Room with `player_count > max_players` if the count was set after construction via the setter (the check only ran in `__init__`).

5.1.2 Room Server - Unit test

The Room Server unit test suite covers all source files. Given the authoritative nature of the Room Server, the testing strategy focuses heavily on **deterministic state transitions** and **crash recovery scenarios**. Every test verifies observable state changes, fault tolerance behavior, or network protocol correctness, rather than internal implementation details.

All tests use `pytest` [9] as the test runner, with `unittest.mock` for dependency isolation. Infrastructure components (TCP sockets, file I/O, threading, HTTP clients) are mocked at the boundary, allowing tests to exercise full business logic without external dependencies. This isolation strategy enables simulation of complex failure scenarios: network partitions during client broadcasts, disk failures during autosave, corrupted persistence files, and simultaneous player disconnections.

Test Categories and Rationale.

1. **Server Initialization and Crash Recovery** (*Req: Fault Tolerance, State Persistence*). Validates server bootstrapping in both fresh-start and crash-recovery scenarios. Tests verify environment variable validation, socket configuration, and global initialization. Critical crash recovery logic is tested: loading persistent state distinguishes alive vs. dead players, setting reconnection deadline, and rejecting stale saves older than `SERVER_RECONNECTION_TIMEOUT`.
2. **Server Lifecycle and Threading Model** (*Req: Real-time Responsiveness, Concurrency*). Tests the multi-threaded architecture: main thread accepts client connections in a loop, API thread runs FastAPI/Uvicorn server for the `/status` endpoint, game loop thread executes tick-based actions at fixed frequency, and per-client handler threads process incoming actions and manage disconnections. Shutdown logic is validated: active games trigger autosave before exit, completed games delete save files to prevent resurrection, all client sockets are closed gracefully, and socket close errors are handled without crashing.
3. **Game Restart and Reset** (*Req: Availability, State Management*). Validates the post-game-over restart sequence: all connected clients receive `SERVER_RESET` notification, clients are disconnected after a brief delay, a new `GameEngine` is created

with random seed, autosave counter and timestamps are reset. Tests confirm that network errors during notification do not prevent restart completion.

4. **Game Loop Timing and Autosave** (*Req: Real-time Responsiveness, Fault Tolerance*). Tests the main game ticker with controlled time mocking: action queue is drained and passed to `engine.tick()` before physics updates, game state is persisted every 5 ticks (`AUTOSAVE_INTERVAL`) when state is `IN_PROGRESS`, no autosave occurs before interval threshold, and broadcasts occur every tick regardless of autosave status. Reconnection timeout logic is validated: if `reconnection_deadline` expires while `expected_players` is non-empty, the server triggers restart; deadline is cleared when all expected players successfully reconnect.

5. **Client Connection Management** (*Req: Client Session Management, Input Processing*). Validates the full client lifecycle: new player join during the waiting phase calls `engine.add_player()`, sends success response, and stores client socket; player reconnection during resumed game verifies player is in `expected_players` set and sends current game snapshot; new join attempts during `IN_PROGRESS` phase are rejected; QUIT action removes client from active clients and calls `engine.remove_player()`. Disconnect handling: broken pipe or empty recv marks player as dead via `engine.kill_player()`, triggers `check_game_over()`, and removes client from active set.

6. **State Broadcasting** (*Req: State Synchronization, Multiplayer Consistency*). Tests the snapshot distribution mechanism: every tick, all clients receive a Protobuf packet containing ASCII grid representation, `is_game_over` flag, and reconnection countdown message. Partial failure handling: if sending to one client fails, the broadcast continues to remaining clients.

7. **Protocol Translation and Action Mapping** (*Req: Input Processing, Protocol Correctness*). Validates the Protobuf-to-engine translation layer: maps each client input to an engine action; invalid or unknown action types return `None` instead of crashing.

8. **Hub Integration and Lifecycle Notifications** (*Req: Two-Tier Architecture, Matchmaking Coordination*). Validates the room-to-hub notification protocol: game transitions trigger corresponding HTTP requests to the hub. Network resilience: HTTP timeouts, connection refused errors, and non-200 status codes are caught and logged without crashing the game loop.

9. **Game Engine Physics and Collision Detection** (*Req: Game Engine Rules*). Movement collision detection: players cannot move into wall tiles, movement out of grid bounds is rejected, dead players cannot move. Bomb mechanics: players can place exactly one bomb at a time, bomb timer decrements each tick, explosion propagates in four cardinal directions until hitting an unbreakable wall, breakable walls stop propagation but are destroyed, players in explosion range are killed. Edge cases: players can walk over bombs, dead players cannot place bombs, attempting to place a bomb for a nonexistent player raises `ValueError`.

10. **Game State Machine and Win Conditions** (*Req: Match Lifecycle*). Validates the three-state finite state machine: `WAITING_FOR_PLAYERS` → `IN_PROGRESS` → `GAME_OVER`. Win condition logic: exactly one alive player declares that player as winner; zero alive players results in draw; two or more alive players keeps game running. Countdown system: requires at least two players to start, timer decrements each tick, game auto-starts when countdown reaches zero.

11. **State Persistence and Corruption Handling** (*Req: Fault Tolerance, Crash Recovery*). Validates serialization: save format includes timestamp and full engine object; load validates timestamp is within the reconnection timeout; stale saves are rejected. Error handling: disk full or permission errors during save return `False` without crashing; corrupted data during load and missing save files trigger fresh start.

12. **Network Protocol and Framing** (*Req: Protocol Correctness*). Validates the length-prefixed message protocol: `send_msg` prepends 4-byte big-endian length header; `recv_msg` reads header then exact message length; partial header and connection closed gracefully return `None`.

13. **Client Reconnection Logic** (*Req: Connection Resilience, Sessions*). Tests client-side reconnection behavior: successful handshake stores tick rate and sets connection state; server rejection sets connection state to `False`; `SERVER_RESET` message stops reconnection attempts; reconnection attempts occur every `RECONNECT_INTERVAL` seconds; timeout expiration sets `running=False`.

14. **Cross-Platform Input Handling** (*Req: Platform Portability, User Input*). Validates dual-platform keyboard input: Windows and Unix-based systems require different approaches to handle in-game keyboard input. Timeout logic and buffer flushing are tested to ensure the input system does not block the game loop.

15. **Edge Cases and Corner Conditions**. Player removal protection: cannot remove player during `IN_PROGRESS`. Bomb collision edge cases: multiple bombs on same position prevented. Simultaneous player deaths: all players killed by same explosion results in draw. Reconnection race conditions: partial reconnection before timeout triggers restart; all players reconnect clears deadline immediately. Network error during broadcast: one client failure does not prevent others from receiving snapshot.

Bug Discovery. The testing process identified edge cases in the initial implementation:

- Bomb placement is limited by position rather than strictly per-player: if two players occupy the same tile, only the first can place a bomb at that location.

- Player removal is only allowed during `WAITING_FOR_PLAYERS` state; attempting removal during `IN_PROGRESS` raises `ValueError` to preserve game integrity. This is useful for player reconnections.

5.1.3 Integration Testing.

While unit tests mock all external dependencies, integration tests exercise the boundaries between real components, using minimal mocking limited to infrastructure that cannot run locally (Kubernetes API).

Hub Server Integration.

1. **Peer Communication via UDP Gossip.** Two real `HubServer` instances exchange gossip messages over actual UDP sockets on localhost. The test verifies that a `PEER_JOIN` message sent by hub-0 is received, deserialized, and processed by hub-1, resulting in hub-0 appearing in hub-1's peer list. This tests the full stack: `HubSocketHandler` serialization → UDP transport → `HubSocketHandler` deserialization → `HubServer._on_gossip_message` → `HubState` mutation.

2. **HTTP API Endpoints.** A real `HubServer` is started with a real FastAPI application. The test exercises the `/matchmaking`, `/room/{id}/start`, `/room/{id}/close`, `/health`, `/ready`, and `/debug/` endpoints via HTTP requests, verifying that state changes propagate correctly between the API layer and the underlying `HubServer` business logic.

3. **Local Room Manager Lifecycle.** A real `LocalRoomManager` creates a room pool, activates rooms, and transitions them through the full lifecycle (`DORMANT` → `ACTIVE` → `PLAYING` → `DORMANT`). The test verifies that the activation callback is invoked with correct room data, that port allocation avoids collisions, and that cleanup removes all rooms.

4. **Kubernetes Room Manager.** Integration with the Kubernetes API is tested using a mocked `CoreV1Api` (since a real cluster is not available in CI), but all other components (`K8sRoomManager`, `Room`, `RoomStatus`) are real. Tests cover pool initialization, existing room recovery from pod labels, dynamic room creation when the pool is exhausted, and cleanup of pods and services.

5. **Failure Detection Flow.** A real `FailureDetector` runs against a real `HubState` with controlled timestamps. The test verifies the full detection flow: a peer whose `last_seen` is artificially aged triggers the `on_peer_suspected` callback, and further aging triggers `on_peer_dead`. This catches timing bugs that unit tests with fully mocked state cannot detect.

6. **Room Health Monitoring.** A real `RoomHealthMonitor` checks rooms against a mocked HTTP server (via `responses` library or `requests_mock`). The test verifies that ACTIVE rooms with a reachable health endpoint remain in state, while rooms returning unexpected statuses or failing to respond trigger the `on_room_unhealthy` callback.

7. **Gossip Deduplication.** A real `HubServer` receives the same gossip message twice (same origin and nonce). The test verifies that the message is processed only once:

the first reception updates state and triggers forwarding, while the second reception is silently discarded. This tests the interaction between `HubState.execute_heartbeat_check` and the message processing pipeline, which unit tests exercise separately.

8. **Matchmaking Flow.** A real `HubServer` with a real `LocalRoomManager` handles a matchmaking request end-to-end. The test verifies that when no active room exists, the server activates a dormant room, broadcasts `ROOM_ACTIVATED` via the socket handler, increments the player count, broadcasts `ROOM_PLAYER_JOINED`, and returns the room connection details.

Bugs Discovered by Integration Tests. The integration test suite for the `K8sRoomManager` uncovered four bugs that unit tests did not catch, because they only manifest at component boundaries:

- **Pod recovery state mapping**: running and pending pods were both mapped to `ACTIVE` status, but pending pods are not yet accepting connections, leading to health check failures on recovered rooms.
- **Resource leak on creation failure**: if pod creation succeeded but service creation failed, the pod was not cleaned up, leaving orphaned pods in the cluster.
- **Timeout handling in pod deletion**: `_wait_for_pod_deletion` did not propagate `ApiException` for non-404 errors, silently swallowing cluster communication failures.
- **Interface contract violation**: `K8sRoomManager._create_room` returned `int | None` instead of `bool` as declared by `RoomManagerBase._create_room`, breaking the Liskov substitution principle.

Room Server Integration. The Room Server integration suite validates the server's external behavior by treating it as a black box accessed via TCP sockets. Infrastructure dependencies (Hub API) are mocked to isolate the server.

1. **Client-Server Protocol**: Verifies the handshake sequence (Join Request $\leftrightarrow$ Server Response) and connection logic. Tests confirm that duplicate client IDs are rejected, joins are refused when the game is `IN_PROGRESS`, and the custom length-prefixed framing protocol functions correctly over real sockets.
2. **Game Loop and Physics Integration**: Sends raw Protobuf action messages (e.g., `MOVE_RIGHT`, `PLACE_BOMB`) and asserts that the resulting game state updates (position changes, bomb appearance, explosions) are correctly broadcast back to all clients in the subsequent state snapshot. This validates the integration between the Network layer, Input Queue, and Game Engine.
3. **Persistence and Recovery Flow**: Simulates a server crash by stopping the server thread during an active game, then restarts it to verify the recovery logic.

The test confirms that the server reloads the game state from the disk, waits for the specific disconnected players, accepts their reconnection, and restores their session state.

4. **Concurrency and State Consistency**: Connects multiple clients sending simultaneous actions to verify that the server's single-threaded game loop and input queue correctly serialize concurrent network events, maintaining state consistency across all connected peers.

How to Run.

```
poetry run pytest
```

5.1.4 End-to-End Testing.

An end-to-end test validates the full system against a live deployment. The test is a scripted game scenario that exercises the entire stack: client → hub matchmaking → room connection → gameplay → game over.

The scenario proceeds as follows:

1. Two players ("Daniele" and "Enrico") connect to the Hub via `/matchmaking` and receive the same room assignment.
2. Both players join the room and wait for the countdown to start.
3. Each player executes a predetermined sequence of moves. The sequence is designed so that Enrico wins the game.
4. The test verifies that the game reaches `GAME_OVER` state with Enrico as the winner, and that the room is properly closed and reported back to the hub.

The E2E test runs automatically in the CD pipeline after deployment, preceded by a 300-second stabilization period to allow all hub pods to reach ready state and complete initial gossip convergence. This is integrated into the GitHub Actions workflow as a post-deployment verification step: if the E2E test fails, the deployment is flagged as broken.

5.2 Acceptance Testing

5.2.1 Hub Server

Manual acceptance testing was performed for scenarios that are difficult or impractical to fully automate:

1. **Multi-Hub Gossip Convergence.** Three hub instances were started locally in manual mode. A room was activated on hub-0, and the `/debug/` endpoint on hub-1 and hub-2 was inspected to verify that the `ROOM_ACTIVATED` event propagated

correctly. Player joins were triggered via `/matchmaking` on different hubs and convergence of player counts was observed. *Why not automated:* requires multiple concurrent processes with real UDP networking; the integration tests cover two-hub communication but not full cluster convergence.

2. **Failure Detection and Recovery.** A hub instance was killed (SIGKILL) to simulate a crash. The remaining hubs were monitored via `/debug/` to verify that the killed hub transitioned from alive → suspected → dead within the expected timeout windows. The killed hub was restarted, and its reintegration into the cluster was verified. *Why not automated:* requires process-level control and real timing; unit tests cover the logic with mocked time, but end-to-end timing behavior needed manual verification.

3. **Kubernetes Deployment.** The full system was deployed as a StatefulSet on a MicroK8s cluster. Hub discovery via DNS, room pod creation via the Kubernetes API, NodePort service creation, and room health monitoring were verified by inspecting pod logs and the `/debug/` endpoint. *Why not automated:* requires a running Kubernetes cluster with specific configuration; the integration tests for `K8sRoomManager` mock the Kubernetes API, but actual cluster behavior needed manual verification.

4. **Client Matchmaking End-to-End.** A client sent a `POST /matchmaking` request to the hub. The returned room address and port were used to establish a TCP connection to the Room Server. Game start and close callbacks were triggered, and gossip propagation was verified on other hubs. *Why not automated:* the E2E test in CI covers a scripted version of this flow; manual testing was used during development before the E2E test existed.

5.2.2 Room Server

Manual acceptance testing was performed to validate the real-time aspects of the game:

1. **Gameplay Fluidity.** Four clients connected to a single Room Server. All players moved simultaneously while spamming actions. Confirmed that the server's input queue handled concurrent requests without dropping commands, and the authoritative state remained consistent for all clients.

2. **Room Server Crash Recovery.** During an active match, the Room Server process was manually terminated and restarted immediately. The server detected the saved checkpoint, restored the engine object, and accepted client reconnections. Players resumed the match from the exact tick where it crashed.

3. **Reconnection Timeout.** A server crashed and was left offline for 60 seconds. On restart, the server correctly discarded the stale save file and started a fresh lobby instead of resuming the old game.

4. **Client Disconnection Handling.** A player closed their terminal mid-game. The server detected the disconnection, waited for the reconnection window, and when the player did not reconnect within the timeout, marked the player as dead and continued the game.

6 Deployment

6.1 Hub Server

6.1.1 Prerequisites

The following software is required on the deployment machine:

Software	Version	Purpose
MicroK8s[2]	v1.32.9	Kubernetes distribution
Docker [4]	20.10+	Image building
Python	3.11+	Application runtime (inside containers)
Poetry	2.0.0+	Dependency management (inside containers)

MicroK8s must have the following addons enabled: `dns`, `ingress`, `storage`, and `rbac`.

6.1.2 Container Images

Both the Hub Server and the Room Server are packaged as Docker images using **multi-stage builds** to minimize image size. The build process is defined in `docker/Dockerfile.hub` and `docker/Dockerfile.room`.

Build Strategy. Each Dockerfile uses two stages:

1. **Builder stage**: installs Poetry and resolves dependencies from `pyproject.toml` and `poetry.lock`, without creating a virtual environment (`virtualenvs.create false`). Only production dependencies are installed (`--only main`).

2. **Runtime stage**: copies the installed packages from the builder, adds the application code, and configures a non-root user (`bomberman`, UID 1000) for security.

The Hub image exposes port 8000 (HTTP) and 9000/UDP (gossip). The Room image exposes port 5000 (TCP game socket). Both images include a Docker `HEALTHCHECK` that verifies socket connectivity.

6.1.3 Kubernetes Manifests

The deployment is defined by a set of Kubernetes [11] manifests in `k8s/base/hub/`:

Namespace (`namespace.yaml`). All resources are deployed in a dedicated `bomberman` namespace, isolating the application from other workloads on the cluster.

ConfigMap (`configmap.yaml`). Centralizes all Hub Server configuration:

```
HUB_DISCOVERY_MODE: "k8s"
GOSSIP_PORT: "9000"
HTTP_PORT: "8000"
FANOUT: "4"
EXPECTED_HUB_COUNT: "16"
FAILURE_DETECTOR_SUSPECT_TIMEOUT: "5"
FAILURE_DETECTOR_DEAD_TIMEOUT: "20"
FAILURE_DETECTOR_CHECK_INTERVAL: "1"
SERVICE_NAME: "hub-service"
```

Changing these values and restarting the StatefulSet reconfigures the cluster without rebuilding images.

RBAC (`rbac.yaml`). Defines a `ServiceAccount`, a `Role` granting permissions to create, delete, list, and manage pods, services, and configmaps within the `bomberman` namespace, and a `RoleBinding` linking the two. This is required because the `K8sRoomManager` dynamically creates room pods and services via the Kubernetes API.

StatefulSet (`statefulset.yaml`). The Hub Server runs as a StatefulSet with 16 replicas and `Parallel` pod management policy (all pods start simultaneously rather than sequentially). Key configuration:

- **Stable hostnames**: each pod receives a predictable hostname (e.g., `hub-0`, `hub-1`), which the application uses to derive the hub index.
- **FQDN construction**: the `HOSTNAME` environment variable is composed at deploy-time from `POD_NAME`, `SERVICE_NAME`, and `NAMESPACE` using Kubernetes variable interpolation (e.g., `hub-0.hub-service.bomberman.svc.cluster.local`).
- **Anti-affinity**: a soft pod anti-affinity rule prefers scheduling hub pods on different nodes, improving fault tolerance in multi-node clusters.
- **Health probes**: a liveness probe on `/health` (initial delay 30s, period 10s) and a readiness probe on `/ready` (initial delay 10s, period 5s) allow Kubernetes to detect and restart unhealthy hubs.
- **Resource limits**: each pod requests 128Mi memory / 100m CPU, limited to 512Mi / 500m.
- **Security context**: pods run as non-root (UID 1000), with privilege escalation disabled and all Linux capabilities dropped.

Services (`service.yaml`). Two services are defined:

- **hub-service** (headless, `clusterIP: None`): enables DNS-based peer discovery. Each pod is resolvable as `hub-{i}.hub-service.bomberman.svc.cluster.local`. Exposes both HTTP (8000/TCP) and gossip (9000/UDP) ports.
- **hub-api** (NodePort): exposes the HTTP API externally on port 80 → 8000, allowing clients outside the cluster to reach the matchmaking endpoint.

Ingress (`ingress.yaml`). Routes traffic for `hub.bomberman.local` to the `hub-api` service. The Ingress acts as a load balancer across all hub pods for incoming HTTP requests.

6.1.4 Deployment Procedure

Local Deployment. The `deploy-local.sh` script automates the full deployment:

```
./k8s/scripts/deploy-local.sh
```

The script performs the following steps:

1. Verifies that MicroK8s is installed and running.
2. Builds the Hub and Room Docker images locally.
3. Imports the images into the MicroK8s container runtime (`microk8s ctr image import`).
4. Applies all Kubernetes manifests in order: namespace, configmap, RBAC, StatefulSet, services, ingress.
5. Waits for all hub pods to become ready (timeout: 300 seconds).
6. Prints deployment status, access URLs, and useful debugging commands.

Expected outcome: 16 hub pods running in the `bomberman` namespace, reachable via the ingress at `http://hub.bomberman.local` and via NodePort at `http://<node-ip>:<node-port>`.

CI/CD Pipeline. The project uses GitHub Actions with two workflows:

- **CI** (`ci.yml`): triggered on push/PR to `main`/`master`. Sets up Python 3.11, installs dependencies via Poetry (with caching), runs the linter (`ruff`), and executes the full test suite. Test results are uploaded as artifacts.
- **CD** (`cd.yml`): triggered automatically when CI passes on `main`/`master`, or manually via `workflow_dispatch`. Builds both Docker images and pushes them to the GitHub Container Registry (GHCR) with `latest` and commit-SHA tags. A `deploy` job then runs on a **self-hosted runner** co-located with the MicroK8s cluster: it pulls the images from GHCR, imports them into MicroK8s, applies all Kubernetes manifests, and performs a rolling restart of the StatefulSet.

6.1.5 Production Network Topology

The production deployment uses a split-infrastructure architecture due to resource constraints:

- A **MicroK8s node** (private network) runs the Kubernetes cluster with all hub and room pods.
- A **public VPS** (`bomberman.romanellas.cloud`) acts as a reverse proxy, terminating TLS and forwarding traffic to the MicroK8s node.
- The two machines are connected via a **ZeroTier** [13] virtual private network, providing a secure layer-2 tunnel without requiring public IP exposure on the MicroK8s node.

HTTP Traffic (Client → Hub). An Nginx [6] reverse proxy on the public VPS forwards HTTPS traffic to the MicroK8s Ingress:

1. Client connects to `https://bomberman.romanellas.cloud`.
2. Nginx terminates TLS (Let's Encrypt certificate) and proxies to `http://hub.bomberman.local` over the ZeroTier tunnel.
3. The MicroK8s Ingress distributes the request to an available hub pod.

Game Traffic (Client → Room). Room Servers are exposed via NodePort services on ports 30000–32767. The public VPS forwards this port range to the MicroK8s node using `iptables` NAT rules:

```
iptables -t nat -A PREROUTING -p tcp --dport 30000:32767 \
    -j DNAT --to-destination <microk8s-zerotier-ip>
iptables -t nat -A POSTROUTING -p tcp -d <microk8s-zerotier-ip> \
    --dport 30000:32767 -j MASQUERADE
```

This allows clients to connect directly to room game sockets via the public VPS address and the NodePort assigned to each room.

6.2 Room Server

The Room Server does not have an independent deployment procedure. Room instances are created **on-demand** by the Hub Server's `K8sRoomManager`, which dynamically provisions Kubernetes pods and NodePort services when a room needs to be activated.

Concretely, the Room Server deployment is embedded in the Hub Server deployment:

- The Room Docker image (`bomberman-room:latest`) is built and imported into MicroK8s during the same deployment step as the Hub image (both in `deploy-local.sh` and in the CD pipeline).

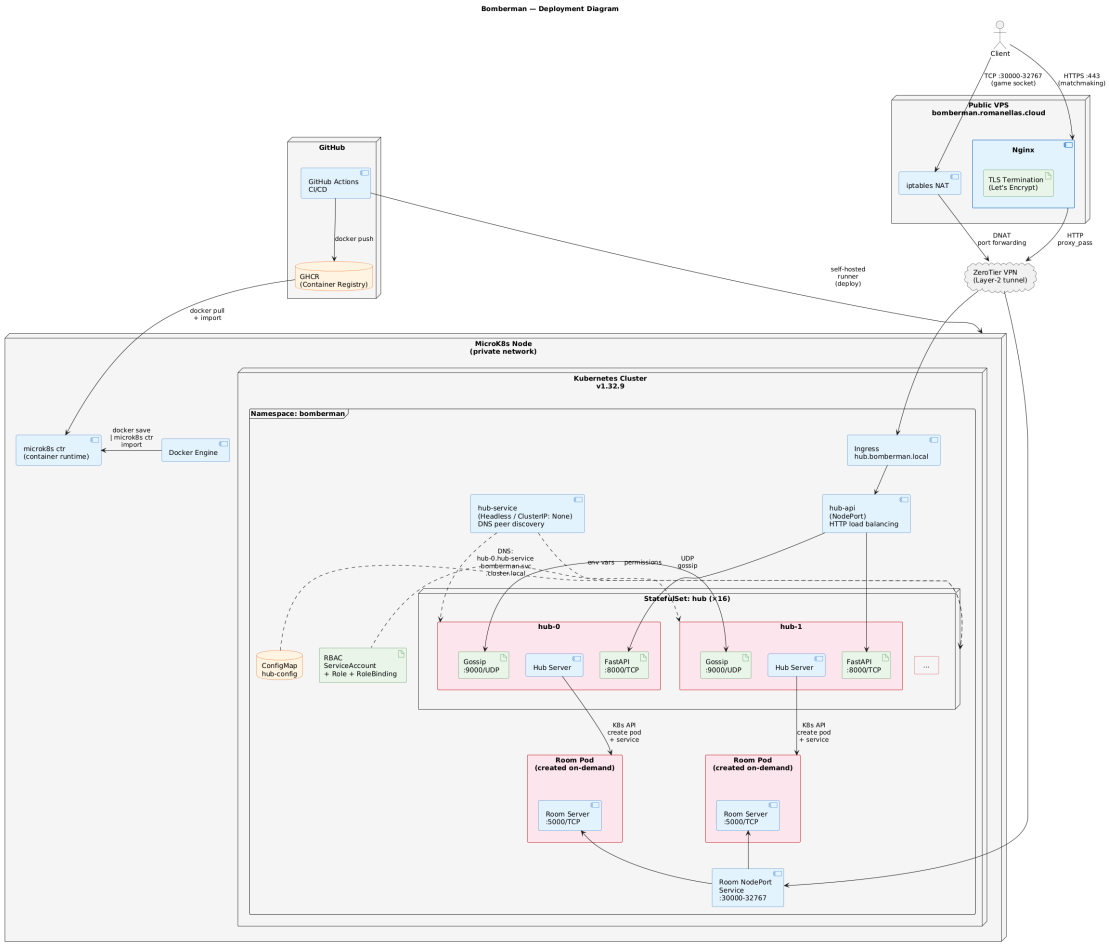

Figure 7: Deployment diagram.

Figure 8: The client prompts the user to enter a Player ID.

- When a client requests matchmaking and no joinable room exists, the Hub's `K8sRoomManager` calls the Kubernetes API to create a new room pod (using the pre-loaded image) and its associated NodePort service.
- On startup, each hub pre-creates a pool of dormant rooms (`STARTING_POOL_SIZE = 1` in K8s mode). Additional rooms are created dynamically if the pool is exhausted.
- On shutdown, the Hub cleans up all room pods and services it owns via `RoomManager.cleanup()`.

The Room image is pre-loaded into the MicroK8s container runtime with `imagePullPolicy: Never`, so pod creation does not require pulling from a remote registry. This reduces room activation latency to the time needed for pod scheduling and container startup.

No separate Kubernetes manifests (StatefulSet, Deployment) are needed for room pods: they are created imperatively by the Hub via the Kubernetes API, labeled with `app=room`, `room-id`, and `owner-hub` for identification and recovery after hub restarts.

7 User Guide

7.1 Starting the Client

Ensure the Python environment is activated, then launch the client from the project's root directory using the following command:

```
python3 Client.py
```

The client automatically contacts the Hub Server via the `/matchmaking` endpoint to obtain a room assignment. This process is transparent to the user: the client resolves the room address and establishes a TCP connection without manual configuration.

7.2 Joining a Game

Upon connection, the client prompts for a player identifier, as shown in fig. 8.

After entering a Player ID, the client connects to the assigned room. The game map is displayed in an ASCII grid, and the interface shows the current game state along with the number of players needed to start (fig. 9).

7.3 Game Start

When a second player joins the room, a countdown timer begins, as shown in fig. 10. The countdown gives additional players time to join before the game starts.

The countdown duration decreases as more players join. Once four players are connected, the countdown is shortened and the game starts quickly.

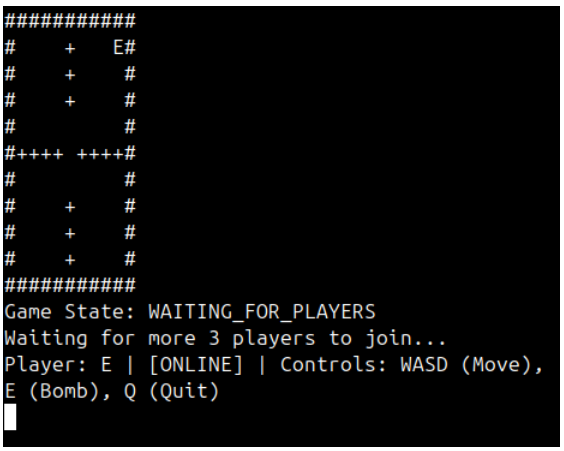

Figure 9: The player has joined the room and is waiting for other players. The map is visible, with the player's character placed on the grid.

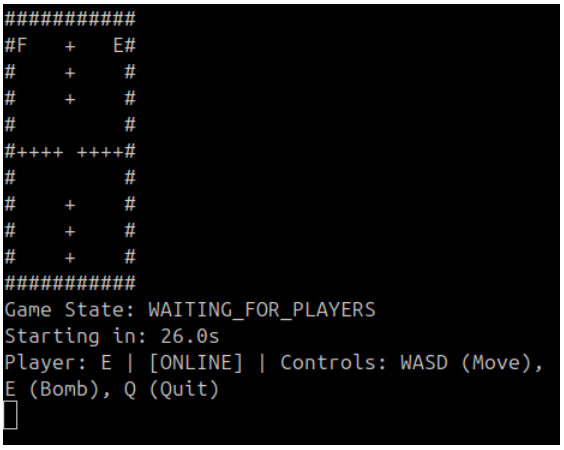

Figure 10: A second player (F) has joined. The countdown to game start is displayed.

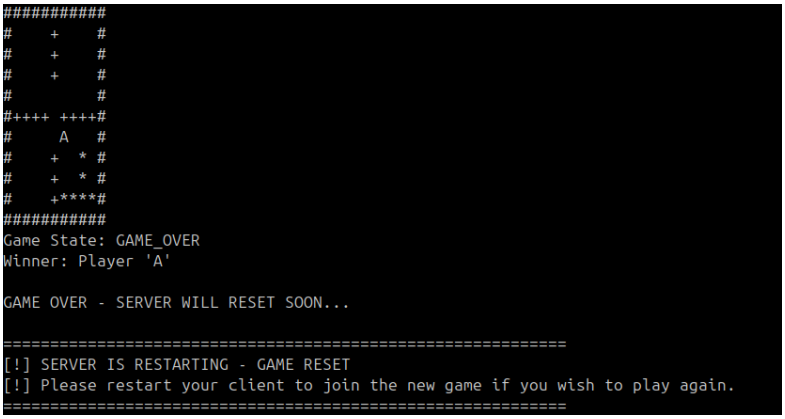

Figure 11: The game has ended. The winner is displayed, and the server announces a reset.

7.4 Controls

During gameplay, the player controls their character using the keyboard:

Key	Action
`W`	Move up
`A`	Move left
`S`	Move down
`D`	Move right
`E`	Place bomb
`Q`	Quit the game

The objective is to eliminate all other players by placing bombs. Bombs explode after a short delay, destroying breakable walls (represented by `+`) and eliminating players caught in the blast radius (represented by `*`). The `#` symbols represent indestructible walls.

7.5 Game Over

When only one player remains, the game ends and the winner is announced (fig. 11).

After the game ends, the server resets and the client disconnects. The terminal returns to the working directory. To play again, the user must restart the client with `python3 Client.py`.

8 Self-evaluation

8.1 Enrico Ferraiolo

8.1.1 Role

My primary responsibility was the design and implementation of the **Room Server**, the CP-oriented component that maintains authoritative game state and executes the deterministic tick loop. I implemented the game engine and state evolution, the TCP networking layer and protobuf protocol integration, the room server lifecycle and client handling, and the reconnection/state persistence logic.

I also implemented the **game client**, including matchmaking integration, real-time input handling, ASCII rendering, and resilience features (reconnect loop and server reset handling). The client was designed to be protocol-driven and language-agnostic so that alternative implementations can interoperate as long as they follow the same message schema.

8.1.2 Strengths

- **Robust authoritative engine.** The room server enforces a deterministic tick-based update loop with a clear separation between game logic and networking, ensuring consistent outcomes for all connected clients.
- **Resilient session handling.** State snapshotting and reconnection timeouts allow clients to recover from transient disconnects without corrupting authoritative state, preserving match continuity.
- **Protocol-first client design.** The client adheres strictly to the protobuf protocol, making the interface generic and suitable for reimplementation in other languages or UIs without server changes.
- **Clean runtime user experience.** Real-time input and terminal rendering provide low-latency feedback and a stable gameplay loop.

8.1.3 Weaknesses

- **Limited client presentation.** The terminal UI is intentionally minimal and lacks richer graphics, audio, and accessibility features that would be expected in a production-quality client.
- **Missing authentication and encryption.** The room server trusts client actions and does not implement authentication, encryption, or rate-limiting measures.

8.2 Daniele Romanella

8.2.1 Role

My primary responsibility was the design and implementation of the **Hub Server**, the AP-oriented component of the distributed system. This included the full gossip protocol (peer lifecycle, room lifecycle, fanout-based forwarding, nonce-based deduplication), the failure detection subsystem, the peer discovery mechanism, the room management layer (both local and Kubernetes-based), and the room health monitoring system.

In addition to the application code, I was responsible for the **infrastructure and DevOps** side of the project: the Docker multi-stage builds, the Kubernetes manifests (StatefulSet, headless Service, RBAC, Ingress, ConfigMap), the CI/CD pipeline (GitHub Actions with automated testing, container registry publishing, and deployment to a self-hosted runner), and the production network setup (ZeroTier VPN tunnel, Nginx TLS termination, iptables NAT forwarding for game traffic).

I also wrote the **unit test suite** for the Hub Server, designed to target business logic exclusively and to surface real bugs rather than verify trivial implementation details.

Finally, I authored the Hub Server sections of the technical report.

8.2.2 Strengths

- **Coherent distributed design.** The Hub Server consistently follows the AP model from design through implementation. Every architectural decision (gossip over UDP, in-memory state, fanout-bounded forwarding, eventual consistency) is motivated by the CAP trade-off and maps directly to a requirement. There are no ad-hoc choices disconnected from the overall design rationale.

- **Robustness of the gossip protocol.** The protocol handles a wide range of scenarios: duplicate messages (nonce-based deduplication), peer crashes and restarts (dead peer resurrection via heartbeat check), transient failures (two-phase failure detection with suspect $\rightarrow$ dead progression), network partitions (independent operation per partition with automatic reconvergence), and cluster scaling (dynamic peer discovery when alive count drops below fanout).

- **Production-grade deployment.** The system is not just a prototype: it runs on a real Kubernetes cluster with health probes, security contexts, resource limits, RBAC, and a fully automated CI/CD pipeline. The split-infrastructure setup with ZeroTier and iptables forwarding demonstrates the ability to solve real operational constraints.

- **Testing methodology.** The test suite focuses on business logic and edge cases, uncovering actual bugs (peer equality comparison, heartbeat timestamp on resurrection, room validation). Red tests that document incorrect behavior were used as a diagnostic tool during development. Coverage targets meaningful code paths rather than line-count metrics.

- **Dynamic room provisioning.** The `K8sRoomManager` creates room pods on demand via the Kubernetes API, with automatic recovery of existing rooms after hub restarts. This eliminates the need for a static room pool and allows the system to scale room capacity based on actual demand.

8.2.3 Weaknesses

- **No message acknowledgment in the gossip layer.** Gossip messages are fire-and-forget over UDP: there is no ACK mechanism to confirm that a peer has received and processed a message. This is a deliberate consequence of the AP orientation, adding ACKs would introduce synchronous coordination and move the system toward CP behavior. However, it means that in low-fanout or high-loss scenarios, some messages may fail to propagate. A lightweight protocol like *eager push with lazy pull* (where peers periodically request missing updates) would mitigate this without sacrificing the AP model.
- **No authentication or encryption.** All endpoints (HTTP and UDP) are unauthenticated and unencrypted. The system relies entirely on Kubernetes network isolation. In a production environment exposed to untrusted networks, gossip injection attacks or unauthorized matchmaking requests would be possible. Adding mutual TLS and API key validation was deferred due to time constraints. For this reason the nginx HTTP gateway is implemented.

9 Future works

9.1 Hub Server

Eager Push / Lazy Pull Gossip. The current gossip protocol relies exclusively on eager push: every message is forwarded immediately to a random subset of peers. If a message is lost (UDP drop, temporary peer unavailability), there is no recovery mechanism. A hybrid *eager push / lazy pull* approach would improve reliability without sacrificing the AP model: hubs would periodically exchange compact state digests (e.g., per-origin nonce vectors), and request missing updates from peers that advertise newer state. This adds convergence guarantees while keeping the protocol fully asynchronous and partition-tolerant.

Gossip Protocol Authentication. As discussed in section 3.9, gossip messages are currently unauthenticated. A practical improvement would be to add HMAC-based message signing using a shared secret distributed via Kubernetes Secrets. Each gossip message would include a signature computed over its serialized payload, and receiving hubs would discard messages with invalid signatures. This prevents gossip injection attacks without requiring a full PKI infrastructure.

Weighted Room Assignment. The current matchmaking logic returns the first joinable room found in the local state. A smarter strategy would consider room fill level, geographic proximity (if hubs are distributed across regions), and hub load. For example, preferring rooms that are closer to being full (to consolidate players and free up resources) or rooms owned by the local hub (to minimize cross-hub coordination).

Graceful Hub Scaling. Currently, scaling the cluster requires updating the `EXPECTED_HUB_COUNT` in the ConfigMap and restarting the StatefulSet. An improvement would be to make the expected hub count dynamic: new hubs joining the cluster would announce themselves via `PEER_JOIN`, and existing hubs would adjust their discovery behavior automatically. This would enable Kubernetes Horizontal Pod Autoscaler integration, scaling the hub cluster based on matchmaking request rate.

Gossip Protocol Metrics. Adding observability to the gossip layer would aid debugging and performance tuning in production. Useful metrics include: messages sent/received per second, duplicate message rate, average propagation latency (time from origin to last hub), peer churn rate, and fanout utilization. These could be exposed via a Prometheus-compatible `/metrics` endpoint and visualized in Grafana.

9.2 Room Server

Spectator Mode. Currently, only active players can connect to a room. Adding a spectator mode would allow additional clients to observe the game in real time without participating. Spectators would receive the same state broadcasts as players but would be excluded from the game engine's input processing.

Persistent Game History. The `GameStatePersistence` module currently handles in-memory state snapshots. Extending it with a lightweight persistent backend (e.g., SQLite per room) would enable game replay, player statistics, and leaderboards across sessions.

Dynamic Map Generation. The current game map is fixed. A procedural map generator that varies wall placement, map size, and power-up distribution based on player count would improve replayability.

9.3 General

End-to-End Encryption. Currently, all communication exception for client-hub is unencrypted within the cluster. We could provide mutual TLS between all components.

Client Reconnection. Implementing a session token mechanism, where the hub assigns a token at matchmaking time and the room could accept reconnections bearing that token within a grace period, would improve the player experience on unstable networks.

Cross-Region Deployment. The current deployment assumes a single Kubernetes cluster. Extending the gossip protocol to span multiple clusters (e.g., one per geographic region) would enable global matchmaking with region-aware room assignment. This would require gossip messages to traverse inter-cluster links (e.g., via a dedicated gateway hub per region) and a latency-aware room selection strategy.

References

[1] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2000. Keynote address.

[2] Canonical Ltd. Microk8s – low-ops, minimal production kubernetes. `https://microk8s.io`, 2024.

[3] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1):8–32, 1988.

[4] Docker, Inc. Docker: Accelerated container application development. `https://www.docker.com`, 2024.

[5] Encode. Uvicorn – an asgi web server. `https://www.uvicorn.org`, 2024.

[6] F5, Inc. nginx. `https://nginx.org`, 2024.

[7] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[8] Google. Protocol buffers. `https://github.com/protocolbuffers/protobuf`, 2026.

[9] Holger Krekel and pytest-dev team. pytest: helps you write better programs. `https://docs.pytest.org`, 2024.

[10] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[11] The Kubernetes Authors. Kubernetes: Production-grade container orchestration. `https://kubernetes.io`, 2024.

[12] Sebastián Ramírez Tiangolo. Fastapi. `https://fastapi.tiangolo.com`, 2024.

[13] ZeroTier, Inc. Zerotier – global area networking. `https://www.zerotier.com`, 2024.