Distributed Software Systems

Distributed Systems Course – A.Y. 2025/2026

Enrico Ferraiolo and Daniele Romanella

Università di Bologna - DISI

A.A. 2025/2026

Outline

Concept

What is it?

A **distributed, real-time** game playable from any terminal via CLI, inspired by **Bomerman** game.

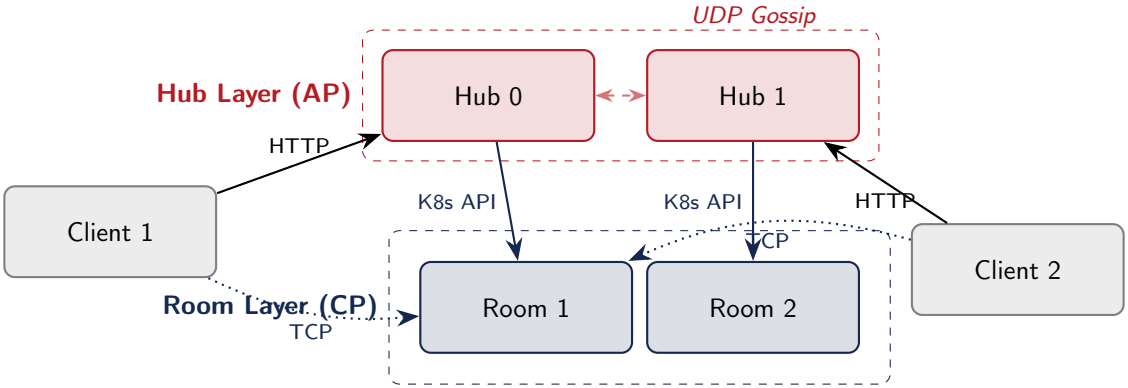
Key properties:

- Ephemeral state – no persistent DB
- Single user role: **Player**
- Up to 4 players per match
- ASCII-rendered game board

Why distributed?

- Geographically distributed players
- Scalability via horizontal scaling
- Fault isolation between matches
- Decoupled matchmaking from gameplay

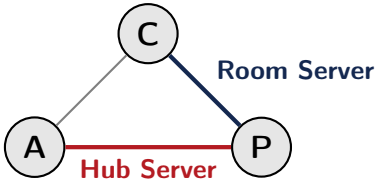
Architecture Overview – Two-Tier Design



Hub Server (AP): matchmaking, peer coordination, gossip protocol

Room Server (CP): authoritative game state, deterministic tick loop

CAP Theorem – Design Choices



Hub = AP

- Always responds to clients
- Eventual consistency via gossip
- Temporary inconsistency is tolerable

Room = CP

- Single source of truth
- Deterministic state evolution
- Disconnected clients are excluded

Hubs

Hub Server – Role and Responsibilities

Entry point for the distributed system:

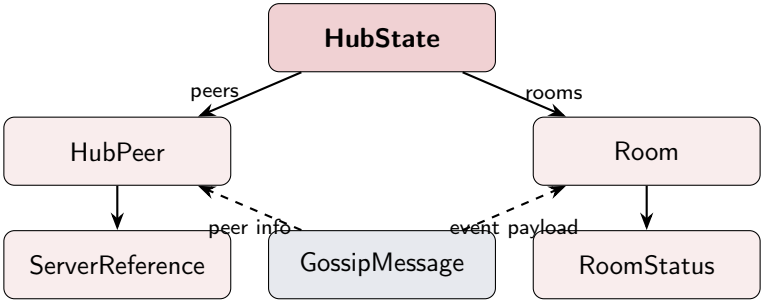
- **Peer discovery** – manual or Kubernetes-based
- **Gossip protocol** – UDP + Protobuf, fanout-bounded
- **Failure detection** – two-phase timeout (suspect $\rightarrow$ dead)
- **Room management** – local pool (local developmente) or K8s dynamic provisioning
- **Matchmaking API** – HTTP via FastAPI
- **Room health monitoring** – periodic HTTP checks

Threading Model

- Main: Uvicorn/FastAPI
- Listener: UDP receive loop
- FailureDetector: periodic checks
- PeerDiscovery: peer count monitor
- RoomHealthMonitor: HTTP probes

All threads share `HubState` via `RLock`.

Hub Server – Domain Entities



HubState: thread-safe aggregate root (RLock) holding the local cluster view.
GossipMessage: Protobuf envelope with nonce, origin, forwarded_by, timestamp, event payload.

Gossip Protocol

Two event categories:

Peer lifecycle:

- `PEER_JOIN` – hub announces itself
- `PEER_LEAVE` – graceful shutdown
- `PEER_ALIVE` – liveness declaration
- `PEER_SUSPICIOUS` – suspicion broadcast
- `PEER_DEAD` – death declaration

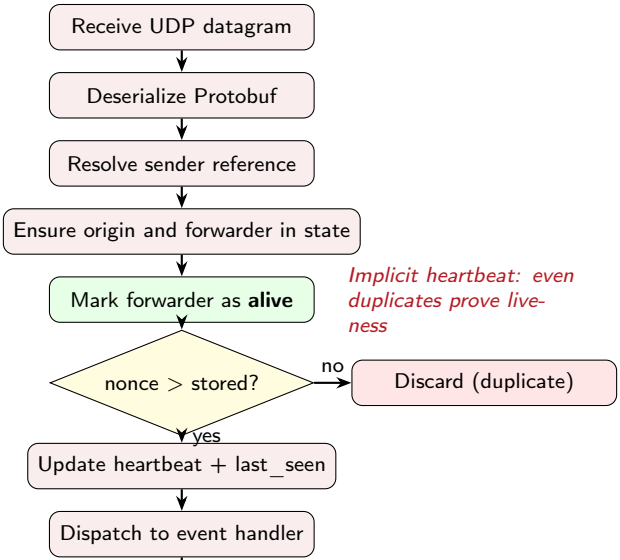
Room lifecycle:

- `ROOM_ACTIVATED`
- `ROOM_PLAYER_JOINED`
- `ROOM_STARTED`
- `ROOM_CLOSED`

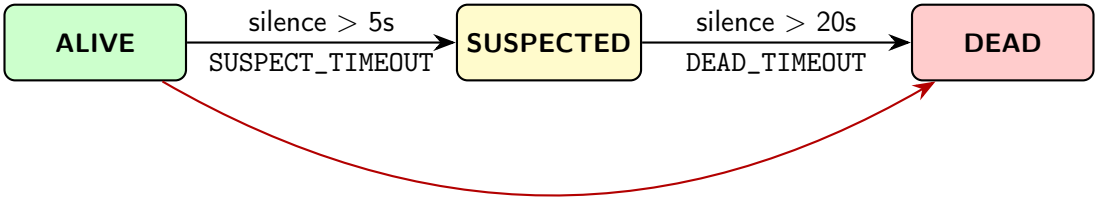
Key design choices:

- **UDP** – fire-and-forget, small datagrams
- **Protobuf** – compact binary encoding
- **Fanout-bounded** – forward to at most $F = 4$ random alive peers
- **Nonce-based dedup** – monotonic counter per origin
- **Implicit heartbeating** – any message updates `last_seen`

Gossip Message Processing Pipeline



Failure Detection – Two-Phase Timeout



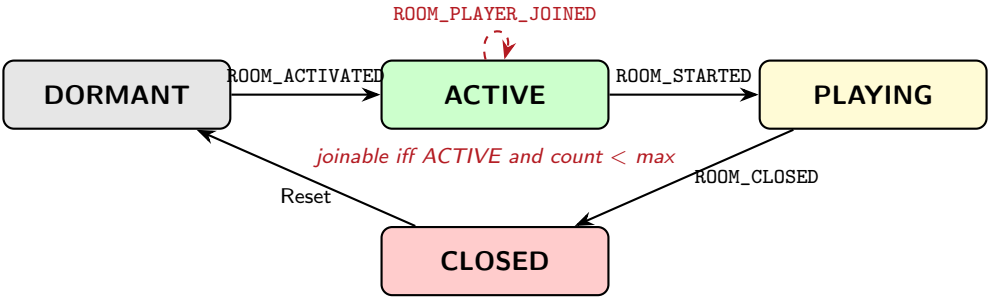
Why two phases?

- Avoids premature removal
- Transient delays → suspicion (reversible)
- Gives suspected peer time to respond

Resurrection:

- Dead peer can rejoin automatically
- `execute_heartbeat_check` resets status
- No manual intervention needed

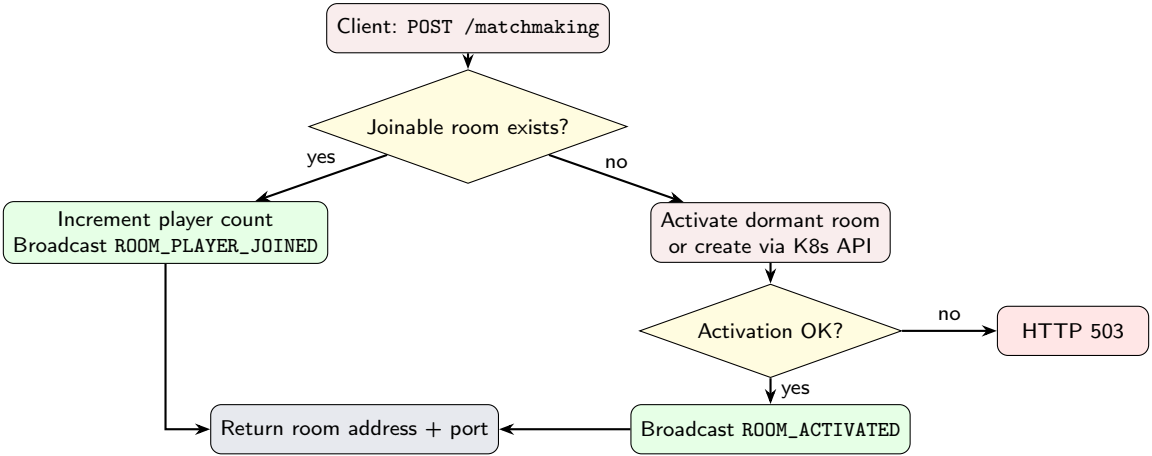
Room Lifecycle State Machine (Hub POV)



Room Management

- **Local mode**: fixed pool, rooms recycled on close
- **K8s mode**: dynamic pod + NodePort creation via Kubernetes API, with recovery on hub restart

Matchmaking Flow



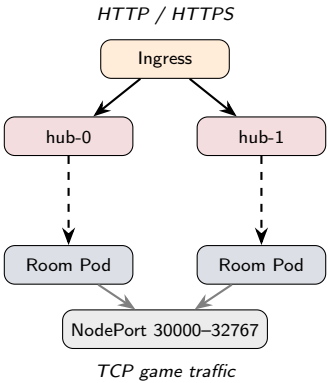
Deployment – Kubernetes

Hub Server: StatefulSet

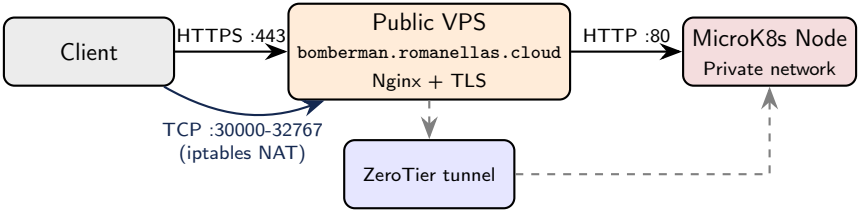
- 16 replicas, parallel pod management
- Stable hostnames: `hub-0`, `hub-1`, ...
- Headless service for DNS peer discovery
- Liveness + readiness probes
- Non-root, capabilities dropped

Room Server: dynamic pods

- Created on-demand by `K8sRoomManager`
- NodePort services for game traffic
- Labels: `app=room`, `owner-hub`
- Recovered on hub restart



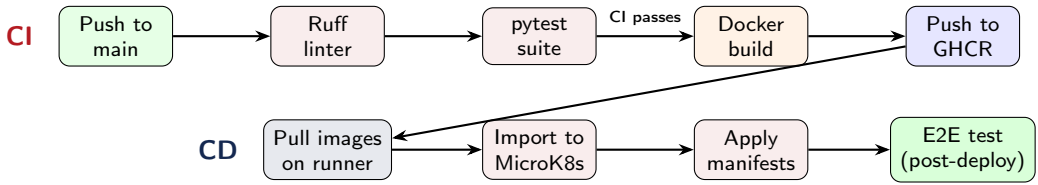
Production Network Topology



Split infrastructure due to resource constraints:

- VPS terminates TLS (Let's Encrypt), reverse-proxies to K8s
- ZeroTier provides secure L2 tunnel without public IP exposure
- `iptables` NAT forwards game port range to MicroK8s node

CI/CD Pipeline



- GitHub Actions: CI on every push/PR, CD on main after CI passes
- Self-hosted runner co-located with the MicroK8s cluster
- E2E test runs a scripted game scenario post-deployment
- In a real scenario E2E are executed in test environment. A second deployment should be executed in production environment iff E2E passes too.

Hub Server – Unit Testing

Strategy:

- 1-to-1 correspondence: class → test file
- Mock infrastructure at the boundary (UDP, K8s, HTTP)
- Focus on **observable behavior**, not call counts

Coverage areas:

- Hostname parsing and initialization
- Gossip pipeline: dedup, dispatch, forwarding
- Peer lifecycle handlers
- Failure detection state machine
- Room lifecycle and matchmaking
- Room health monitoring
- Input validation and edge cases

Rooms

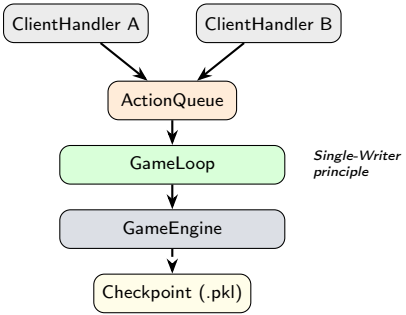
Room Server – Role and Architecture

Authoritative game server for a single match:

- Runs deterministic **tick-based game loop**
- Receives player inputs via TCP + Protobuf
- Broadcasts state snapshot every tick
- Single source of truth (CP-oriented)
- Handles up to 4 concurrent clients

Two network interfaces:

- **TCP :5000** – game socket
- **HTTP :8080** – management API



Game Engine – Entities and Rules

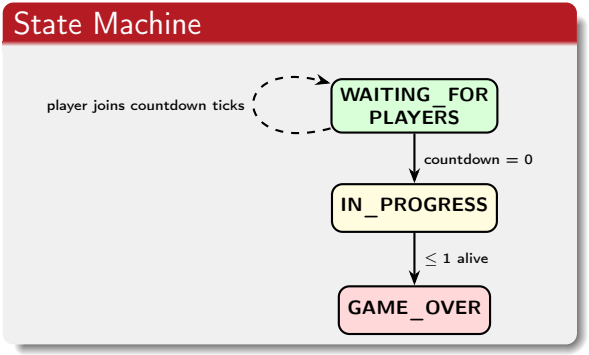
Domain Entities:

- **GameEngine**: aggregate root – grid, players, bombs
- **Player**: ID, position, alive/dead status
- **Bomb**: position, owner, timer, blast range

Game Rules:

- Movement: collision detection vs. walls and grid bounds
- Bombs: 1 per player, timer decrements each tick
- Explosion: 4 cardinal directions, stopped by walls

State Machine



Countdown: requires ≥ 2 players. Duration decreases as more players join. Auto-starts at 4 players.

Protocol – TCP + Protocol Buffers

Why TCP?

- Guarantees packet ordering
- Reliability for game commands
- No lost bomb placements

Message types:

Client → Server:

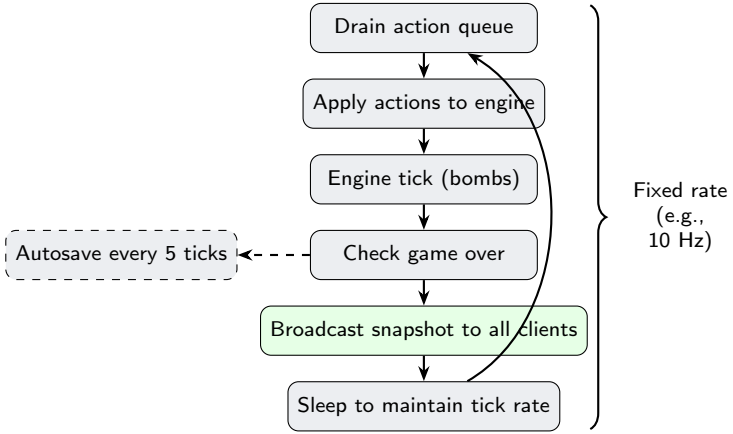
- `JoinRequest` (player_id)
- `Action` (MOVE_UP/DOWN/LEFT/RIGHT, PLACE_BOMB, QUIT)

Server → Client:

- `ServerResponse` (accept/reject)
- `GameStateSnapshot` (ASCII grid, is_game_over, message)
- `SERVER_RESET` notification

State Authorization

Game Loop – Tick-Based Processing



Single-Writer principle: only the GameLoop thread mutates the GameEngine.
Partial failure: if broadcast to one client fails, others still receive the snapshot.

Fault Tolerance – Checkpointing and Recovery

Local Checkpointing:

- GameEngine serialized to `.pkl` file
- Every `AUTOSAVE_INTERVAL` ticks (5)
- No network replication overhead
- Best-effort recovery mechanism

Recovery Flow:

1. Server restarts, finds `.pkl` file
2. Check timestamp: if < 30s old $\rightarrow$ load
3. Distinguish alive vs. dead players
4. Set reconnection deadline
5. Wait for disconnected players to rejoin
6. If timeout expires $\rightarrow$ fresh restart

Client Disconnection:

- `Error`
- Player marked dead in engine
- Match continues for remaining players

Cleanup

Completed games delete save files to prevent ghost resurrection.

Room Server – Unit Testing

Coverage areas:

- Server init and crash recovery
- Lifecycle and threading model
- Game loop timing and autosave
- Client connection management
- State broadcasting (partial failures)
- Protocol translation layer
- Hub integration notifications
- Game engine and collision
- Win conditions and state machine
- Persistence and corruption handling
- Network framing protocol
- Reconnection logic
- Cross-platform input handling

Integration Testing

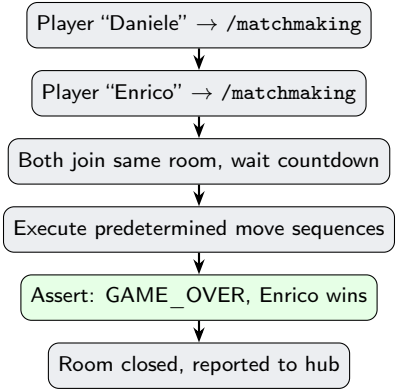
Hub Server:

- Real UDP gossip between two hubs
- HTTP API endpoints (matchmaking, lifecycle)
- LocalRoomManager full lifecycle
- K8sRoomManager (mocked K8s API)
- FailureDetector with real HubState
- Gossip deduplication end-to-end
- Matchmaking flow with room activation

Room Server:

- Client-server protocol over real sockets
- Game loop via Protobuf actions
- Persistence and recovery flow
- Concurrency: multiple clients, state consistency

End-to-End Testing



- Runs in CD pipeline after deployment (300s stabilization period)
- Tests full stack: client → hub matchmaking → room → gameplay → game over
- Failure flags the deployment as broken

Demo

Future Works

Hub Server:

- Eager push / lazy pull gossip for better reliability
- HMAC-based gossip authentication
- Weighted room assignment (fill level, proximity)
- Graceful hub scaling (dynamic `EXPECTED_HUB_COUNT`)
- Prometheus metrics for gossip layer

Room Server:

- Spectator mode
- Persistent game history (SQLite)
- Dynamic/procedural map generation

General:

- End-to-end encryption (mTLS)
- Session token-based reconnection
- Cross-region deployment with region-aware matchmaking

Conclusions

What we built

A **coherent distributed system** that balances availability (Hub, AP) with strong consistency (Room, CP) at different layers, each with its own CAP trade-off.

Strengths:

- Robust gossip protocol with failure detection and resurrection
- Deterministic authoritative game engine
- Production-grade K8s deployment with CI/CD
- Comprehensive testing: unit, integration, E2E

Known limitations:

- No gossip ACK mechanism (fire-and-forget)
- No authentication or encryption
- Minimal client UI (ASCII terminal)
- No persistent game history

Thank you

Questions?

Enrico Ferraiolo – enrico.ferraiolo2@studio.unibo.it
Daniele Romanella – daniele.romanella@studio.unibo.it

GitHub: Distributed-System-AA-2025-2026/Bomberman