DISTRIBUTED SYSTEMS COURSE — A.Y. 2025/2026

Distributed Lupus in Fabula

A Replicated Real-Time Multiplayer Game Platform

Leonardo Vorabbi leonardo.vorabbi@studio.unibo.it

Jacopo Bonifazi jacopo.bonifazi@studio.unibo.it

Alexandru Nicolescu alexandru.nicolescu@studio.unibo.it

July 20, 2026

OVERVIEW

Agenda

01

Concept & Use Case

02

Requirements

03

Architecture & Design

04

Fault Tolerance & Consistency

05

Validation & Deployment

06

Conclusions & Future Work

What is Distributed Lupus in Fabula?

A real-time, browser-based social deduction game, played by geographically dispersed groups, powered by a horizontally replicated backend.

Frontend

Vue 3 Single-Page Application, no installation required

Backend

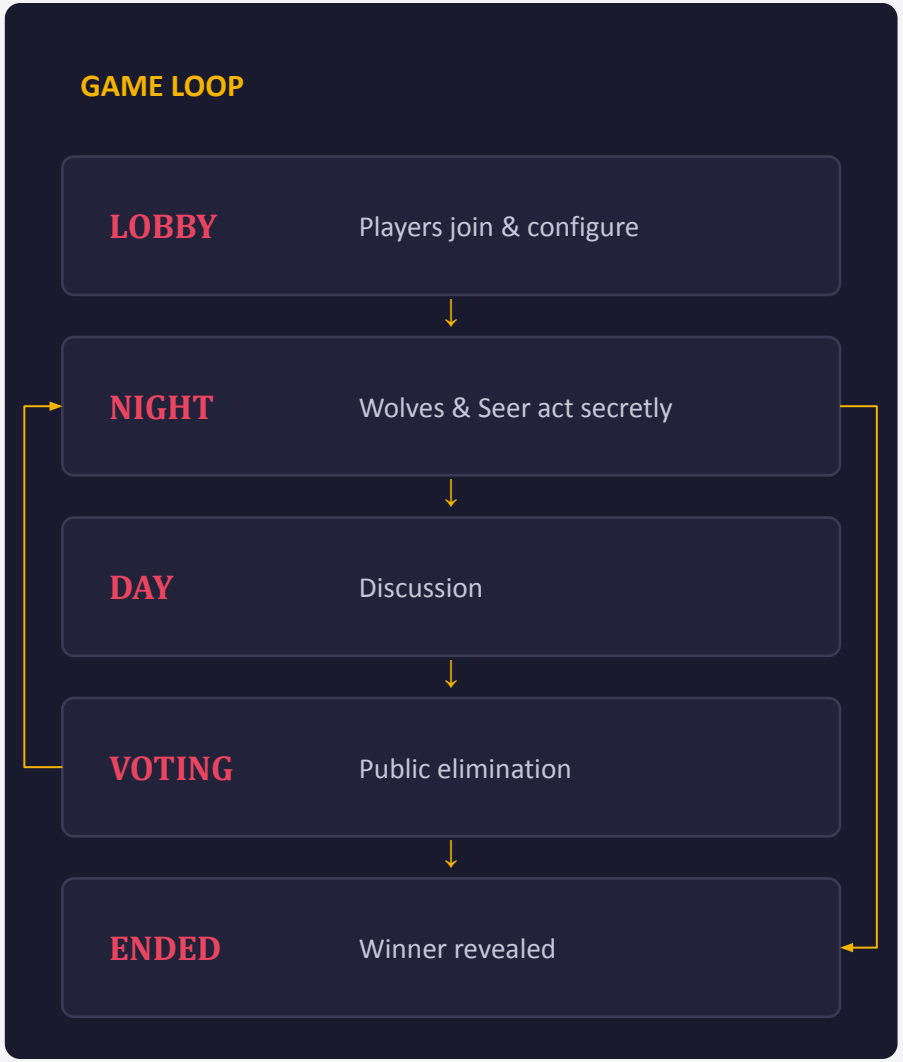
Stateless FastAPI + Socket.IO replicas, horizontally scaled

Data layer

Redis - shared state store & Pub/Sub broker

How to play Lupus in Fabula?

- Players receive a **secret role**.
- The game alternates between **Night** and **Day**.
- **Night:** Werewolves choose a victim; special roles use their abilities.
- **Day:** Players discuss, accuse, and vote to eliminate a suspect.
- **Villagers win** by eliminating all Werewolves.
- **Werewolves win** when they equal or outnumber the remaining Villagers.



Why Distribution Is Needed

Not for geographic proximity, but for scalability and fault tolerance.

Horizontal scalability

Stateless replicas serve any room in parallel; scale out by adding instances

Fault tolerance

Reconnection, retries & anti-entropy recover from crashes and drops

02

Requirements

Functional, non-functional and implementation requirements — each with an acceptance criterion (AC)

Requirements at a Glance

Functional (FR)

14 requirements

Non-Functional (NFR)

9 requirements

Implementation (IR)

3 requirements

GLOSSARY

Room / Lobby: Independent game session, hosting one match

Host: Player who configures, starts, kicks & restarts

Phase: Lobby, Night, Day, Voting, Ended

Role: Wolf, Seer or Villager - secretly assigned

Replica: One interchangeable backend server instance

Snapshot: Authoritative state sent to (re)connecting clients

Functional Requirements - Lobby & Setup

FR1 — **Room creation & joining**
First user in a room becomes host

FR2 — **Lobby browsing**
List open rooms with host & player count

FR3 — **Match configuration**
Host-only; invalid configs rejected

FR4 — **Readiness & game start**
Starts only when all players are ready

FR12 — **Host migration**
Automatic promotion if host disconnects

FR13 — **Room administration**
Kick players; restart a finished match

FR14 — **Access control**
No join to running/ended games; unique nicknames

Functional Requirements - Gameplay

FR5 — **Secret role assignment**
No player ever learns another's role early

FR6 — **Game loop & timers**
Server-driven, automatic phase transitions

FR7 — **Public vote**
Ties/skip-majority cause no elimination

FR8 — **Night actions**
Wolves & Seer act privately, cross-replica

FR9 — **Win detection**
Ends & reveals roles as soon as a faction wins

FR10 — **Chat**
Global, wolves-only & dead-player channels

FR11 — **Reconnection**
Resume with role & state, on any replica

Non-Functional Requirements

NFR1

Replication transparency

NFR2

Fault tolerance

NFR3

Convergence (~10s)

NFR4

Safety of game rules

NFR5

Scalability

NFR6

Soft real-time (<1s)

NFR7

Observability

NFR8

Zero-downtime updates

NFR9

Usability

Implementation Requirements & Key Distributed Concerns

IMPLEMENTATION REQUIREMENTS

IR1	Containerised: Docker Compose + Kubernetes
IR2	Prometheus-compatible metrics
IR3	Free / open-source components only

DISTRIBUTED-SYSTEM CONCERNS (RELEVANCE)

- **Transparency** — Central
- **Fault tolerance** — Central
- **Scalability** — Central
- **Resource sharing** — Central
- **Performance/concurrency** — Central
- **Security/trust** — Limited

WHAT MAKES IT DISTRIBUTED

- Many concurrent rooms served in parallel
- State lives in Redis, not replica memory
- Pub/Sub propagates events across replicas
- Replicas can crash / restart / scale freely
- Reconnecting players resume on any replica

03

Architecture & Design

Layered three-tier backbone with an event-based core

Architectural Style

“Stateless servers + stateful store” ⟶ the standard recipe for horizontally scalable service redundancy.

Presentation tier

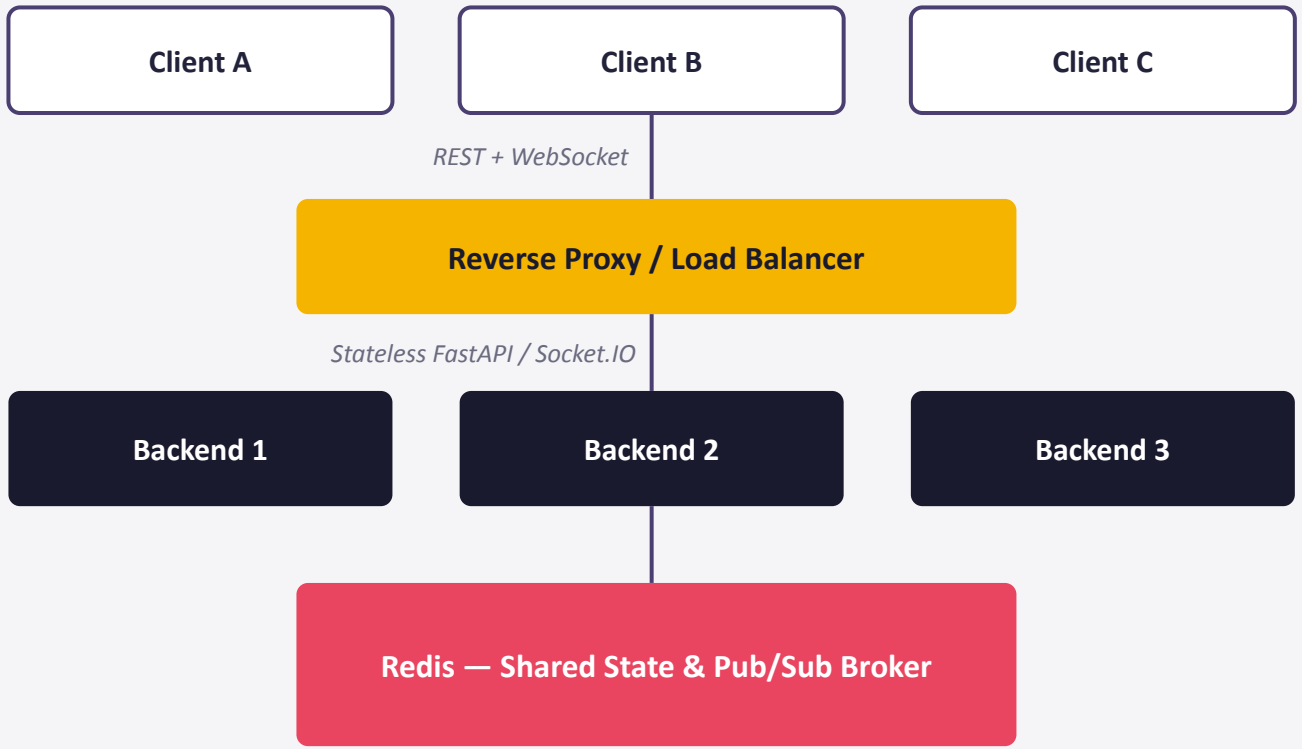
Application tier

Data tier

EVENT-BASED CORE

- Client ↔ server: each room is a Pub/Sub topic over a persistent socket
- Replica ↔ replica: one Redis channel per room propagates every event
- Replicas never address each other directly: no membership protocol
- Set of replicas can change anytime (autoscale, crash, rollout) with zero reconfiguration

Infrastructure Components



KEY PROPERTIES

- ✦ Any replica can serve any room (replication transparency)
- ✦ Segregated bridge networks confine the blast radius of a compromised edge
- ✦ Backend floor of 2 replicas (no SPOF), ceiling of 8 (autoscaler)
- ✦ Naming & discovery by name, never address — DNS resolved at request time

Domain Model

Room	Independent match container, identified by its code; owns everything else
Player	Nickname, secret role, liveness, connectivity, readiness, host flag
GameState	Phase, round, timer, config, host id, ready set, winner, state version
Votes	Public day vote + secret wolf vote + seer's nightly inspection
Chat message	Sender, text, channel — transient, relayed but never stored

Interaction Patterns

Request / Response

Queries only — lobby list, health checks. Any replica can respond.

Publish / Subscribe (×2)

Client-facing: joining a room subscribes to its stream. Replica-facing: events propagate over the Redis bus.

Cross-replica unicast

Private events (role, seer result) reach exactly one client, wherever it's connected.

Behaviour — The Match as a Replicated State Machine

Transitions are triggered by phase-deadline expiry, early completion, or a decisive disconnection. Every resolving transition runs check_winner().

Phase timers

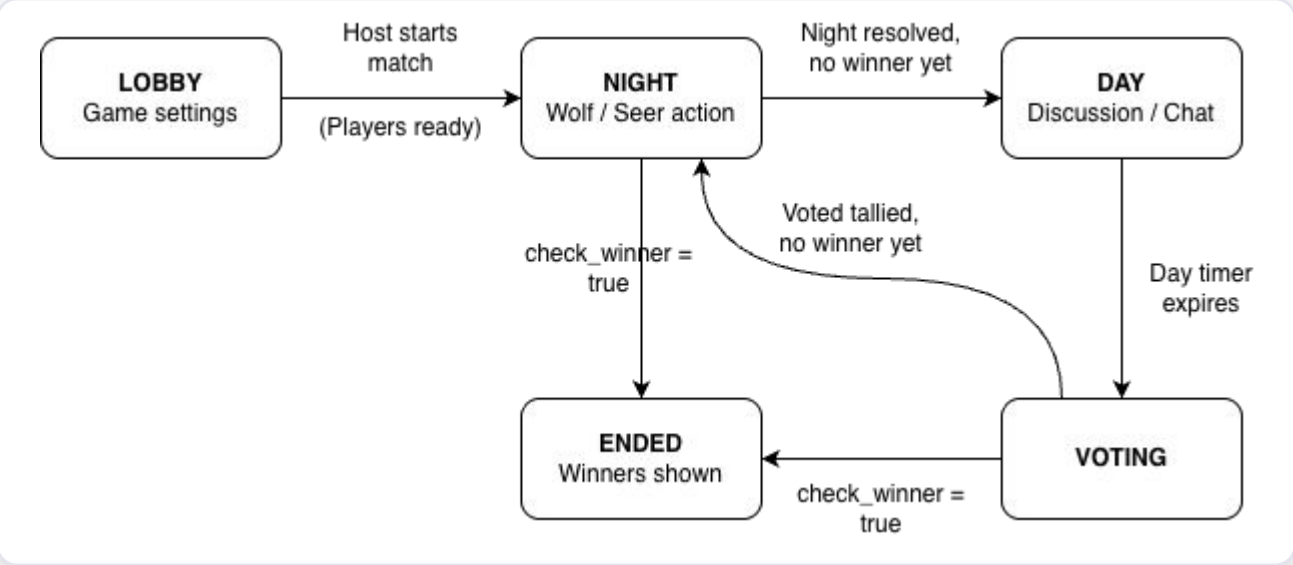
Local optimisation only — a firing timer checks the stored deadline first (stale-timer guard)

The sweeper

Every couple seconds scans for expired deadlines — phase progression survives a replica's death

Advancement lock

Short-TTL distributed lock: exactly one trigger wins per deadline



Frontend Architecture - A Layered SPA

The client is event-driven and stateless with respect to game logic: it never decides who dies, when a phase ends, or who wins.

1

Views - the screens

Present state and collect input; never talk to the network directly

2

Stores - application state

Reactive client state, server-event listeners, command-emitting actions

3

Communication layer

A single Socket.IO wrapper - the only component that touches the network

★

Three Pinia stores

Lobby · Game · Chat - each mirroring a slice of the server-held domain

WHY IT MATTERS

No authoritative state

The client can always be rebuilt from a server snapshot

Fault recovery is free

Re-sync = adopt the snapshot, no reconciliation logic

Instance-agnostic

Behaves identically whichever replica serves it

Frontend Implementation

Vue 3 SPA with reactive state, one shared socket, and configuration injected at container start.

SPA

Vue 3 · Vite · Vue Router

Composition API; four routes - home, lobby, game, results

STATE

Pinia stores

Reactive state, computed selectors, event listeners, command actions

NET

socket.io-client

One module-level connection · WebSocket-only · bounded auto-reconnect

UI

Reusable components

SVG player card, deterministic avatar, circular phase timer, chat box

NOTABLE DETAILS

Pending-listener registry

Stores register handlers before the socket exists - no mount/connect race

Message de-duplication

Absorbs the at-least-once delivery of multi-path fan-out

Runtime config

WS_URL injected at container start - one image, any domain

04

Consistency & Fault Tolerance

Strong consistency at the store, eventual consistency at the clients

Consistency Strategy: Strong at the Store, Eventual at the Clients

AUTHORITATIVE STATE (Redis)

- Single-operation atomicity for naturally atomic updates (a vote)
- Atomic create-if-absent (SET NX) — prevents the double-host race
- Optimistic transactions (WATCH/MULTI/EXEC), retried on conflict
- Mutual exclusion (advancement lock) for phase transitions

CLIENT VIEWS (eventual)

- Events may reach replicas' clients at slightly different times
- Bus offers at-most-once delivery — an event can be missed entirely
- Sub-second divergence is imperceptible in a chat-paced game
- Periodic anti-entropy snapshot overwrites any diverged view within ~10 s

What Happens When a Component Fails

 **Backend replica crashes**	Clients reconnect to a surviving replica & re-sync by snapshot; sweeper advances phase elsewhere	**Brief reconnect, match continues**
 **Store (Redis) unreachable**	Commands rejected cleanly; listener reconnects with back-off; anti-entropy re-syncs on recovery	**State-changing commands briefly refused**
 **Proxy / frontend crash**	Stateless — orchestrator restarts them automatically	**Transient connection error**
 **Client disconnect**	Match pauses; grace timer; reconnect (any replica) resumes it	**Short pause, resumes on return**

05

Validation & Deployment

Three testing layers, three deployment profiles

Three Layers of Automated Testing

Unit

Domain logic & Redis access via fakeredis — fast, deterministic

FR3, FR5–FR9, FR12

Integration

Cooperating modules on one replica; distributed-safety races encoded as tests

FR1, NFR2, NFR4

End-to-end

Real Socket.IO connections against a running multi-replica backend + load tests

NFR1–NFR6, FR10–FR11

Client-Side Testing

Vitest unit tests on the stores, manual end-to-end runs on the real UI.

AUTOMATED

Vitest — the store layer

- Unit tests on the three Pinia stores
- Reaction to incoming events (phase change, vote update)
- Chat message de-duplication
- Phase-timer countdown and router configuration
- Views deliberately not unit-tested — a pure projection of tested store state

MANUAL

End-to-end on the real UI

- Several browser windows in separate incognito sessions
- Full flow: create → join → ready/start → day/night cycle → results
- Transient disconnect and reconnect — state resync verified
- Multi-instance: players split across two replicas behave identically
- Qualitative checks a script cannot make: readability, timer smoothness

Three Deployment Profiles

1 · Development

Support services only (Redis, Prometheus, Grafana) + native hot-reload backend/frontend

```
IDE-speed edit cycle
```

2 · Full Docker Compose

Full stack, single machine — NGINX, N backend replicas, Redis, monitoring

```
docker compose up --scale backend=2
```

3 · Kubernetes (minikube)

Production-like — HPA (2–8 replicas), rolling updates, Ingress, PVC-backed Redis

```
k8s/deploy.sh
```

06

Conclusions & Future Work

A distributed, replicated real-time platform where replication is real and tested — not nominal.

High-availability store

Redis Sentinel / Cluster to remove the single stateful point of failure

Strong authentication

JWT/OAuth login & persistent player identities

At-least-once delivery

Redis Streams — replay instead of periodic snapshot repair

Connection-aware autoscaling

Scale on active WebSocket connections, not just CPU/memory