# Scopa2
# A Geo-Distributed Multiplayer Card Game

**Final Report for the Distributed Systems Course — AY 2025/2026**

Marco Coppola
marco.coppola3@studio.unibo.it

Valerio Pio De Nicola
valeriopio.denicola@studio.unibo.it

June 16, 2026

*Scopa2* is an online two-player card game: a variant of the Italian *Scopa Scientifica* extended with a card-mutation mechanic based on *Santi*, playable through a native desktop client. Behind the game there is a deliberately distributed backend: two symmetric regions (Europe and North America) run the same stack on Kubernetes, the client picks the closest region by measuring latency at startup, and every component is replicated so that the death of a pod, of a database node, or even of an entire region does not interrupt a match. The two core ideas of the design are (1) modelling each match as an event-sourced deterministic state machine, so that any server can rebuild any game by replaying its move log, and (2) delegating the hard coordination problems (consensus, leader election, failover) to infrastructure that already solves them: YugabyteDB for Raft-replicated storage, Redis Sentinel for the cache and pub/sub tier, Kubernetes for scheduling and scaling. This report explains the design, the consistency trade-offs we made (CAP/PACELC), and how we validated the system with fault-injection tests on the live deployment.

## Contents

# 1 Concept

The project is a native multiplayer game. The client is packaged for Windows, macOS and Linux (x86_64 and ARM64); a player downloads it, registers with a username, and starts looking for an opponent. Once two players are paired they play a full match of Scopa, with every move validated by the server and pushed to the opponent over a WebSocket.

The crucial part of the experience is responsiveness: a move must show up on the opponent's table within a fraction of a second. At the same time the game cannot tolerate lost or reordered moves — a card game where the two players see different tables

is simply broken. So we need low latency *and* consistency, which is exactly the kind of tension that makes this a distributed-systems problem.

A single match is small, but the system as a whole has to deal with three forces that no single machine can absorb:

- **Geography.** A player in Bologna and one in San Francisco cannot share one server and both get fast feedback. The system is therefore split into regions, each with its own compute and database, while still presenting a single global identity space.
- **Survivability.** Losing a virtual machine mid-match must not kill the game: players want to keep playing, not restart. The match state must be durable and rebuildable on any healthy node.
- **Elasticity.** Traffic is variable (evening peaks, quiet nights), so the application tier must scale horizontally — which in turn means no server can hold session state that would prevent it from being killed and replaced at any moment.

The code lives in two repositories: the *backend* (`scopa2-backend`, Laravel 12 on Octane and FrankenPHP) and the *client* (`scopa2-game`, Godot 4.6 in C#/.NET 8). The only contract between them is a small JSON-over-HTTPS API plus the Pusher-compatible WebSocket protocol, so either side can be replaced independently.

# 2 Requirements

## 2.1 Functional Requirements

F1. Players can register and authenticate; credentials survive client restarts so returning users land directly in the main menu.

F2. Players can enter a matchmaking queue that pairs people with similar ELO ratings; the tolerance widens with waiting time so that low-traffic moments still produce matches.

F3. Paired players play a complete game of Scopa extended with the *Santi* mechanic, which lets a player spend captured cards to buy consumable abilities that alter the normal flow of the game.

F4. The server validates every move against the current game state and rejects illegal actions: clients are untrusted.

F5. Every move reaches the opponent in near-real time over a pub/sub channel; both players always converge on the same state.

F6. A player whose client crashes mid-game can reconnect and resume the same match exactly where it was.

F7. At the end of a game both players' ELO ratings are updated atomically with the final outcome.

## 2.2 Non-Functional Requirements

NF1. **Latency:** the delay between a player's action and its appearance on the opponent's screen must be imperceptible when both players are in the same region.

NF2. **Single-pod resilience:** the death of any application or WebSocket pod must cost the player at most a brief visual stutter.

NF3. **Single-node database resilience:** the crash of one of the three database nodes in a region must not lose committed writes nor make writes fail.

NF4. **Region survivability:** losing an entire region must leave the other one able to serve all traffic. A bounded amount of very recent data may be lost, but a global outage is not acceptable.

NF5. **Operability:** releases must be reproducible and fully automated, with no manual steps from a developer's machine.

## 2.3 Technology Constraints

We picked Laravel, Godot and Kubernetes (in the K3s flavour) partly out of familiarity and partly because we wanted to build on components that already implement the lower layers of distributed coordination correctly. In particular we chose YugabyteDB because it hides a Raft-replicated log behind the standard PostgreSQL wire protocol (so the application code is plain SQL), and Soketi for the realtime tier because it is wire-compatible with Pusher and ships with a Redis adapter that already solves the cross-pod fan-out problem.

## 2.4 Distributed-Systems Checklist

How the classic distributed-systems concerns map onto the project, each discussed in detail later:

**Location transparency:** the client never chooses a pod, and picks a region only through an automated latency probe (Section 3.8).

**Replication transparency:** a player's data looks the same no matter which database replica answers.

**Failure transparency:** pod, Redis-master and even whole-region failures are absorbed with at most a short visual artefact (Section 3.7).

**Openness:** every external protocol is open — HTTPS, the Pusher WebSocket protocol, the PostgreSQL wire format.

**Scalability:** horizontal autoscaling on the application tier, replicated storage on the data tier, Sentinel-managed replicas on the cache tier.

**Concurrency control:** the two contended resources — the matchmaking queue and the per-game event log — are protected by an atomic Lua script and by a row lock plus uniqueness constraint respectively (Section 3.6).

**Security:** TLS at the edge, bearer-token authentication, authorised private broadcast channels.

# 3 Design

## 3.1 Architectural Style

The most important design decision in Scopa2 is about recovery rather than style: every match is an *event-sourced deterministic state machine*. The authoritative state of a game is not a snapshot in a database row but the totally ordered list of moves stored in the `game_events` table. The in-memory `ScopaEngine` is pure: given the same seed and the same event list it always produces the same `GameState`.

Almost everything else follows from this. Application pods can be stateless and interchangeable because any of them can rebuild any match by replaying its log; recovering from a crashed pod — or from a whole region going down — reduces to fetching the log from a surviving database and replaying it. Most of the resilience of the system is a consequence of determinism, not of dedicated recovery code (Section 3.7).

Two further patterns complete the picture. A publish/subscribe bus handles the realtime fan-out: validated moves are emitted as domain events on Pusher-compatible channels and delivered to whichever pod holds each player's WebSocket. A shared Redis dataspace holds the matchmaking queue, which any application pod can operate on without keeping local state. Both serve the same goal as the event-sourced engine: keep the application tier stateless so the system survives losing any of its parts.

## 3.2 Infrastructure

The deployment substrate is K3s, a lightweight Kubernetes distribution, running with two regional namespaces (`region-eu`, `region-us`) plus dedicated namespaces for the stateful tiers (`db`, `cache-eu`, `cache-us`). Everything is templated through a single Helm chart with per-region values files, so the two regions are guaranteed to be symmetric by construction.

**Ingress.** One NGINX ingress per region terminates TLS and routes by path: `/api/*` goes to the application service, `/ws/*` to the Soketi service. The same ingress exposes the `/up` health probe used by the client's region selection.

**Application tier.** The Laravel application is packaged as a multi-stage Docker image based on `dunglas/frankenphp`: a single PHP 8.3 binary in worker mode keeps the framework resident across requests, removing the per-request bootstrap cost of classic PHP-FPM setups. Inside the pod, Supervisor runs three processes side by side:

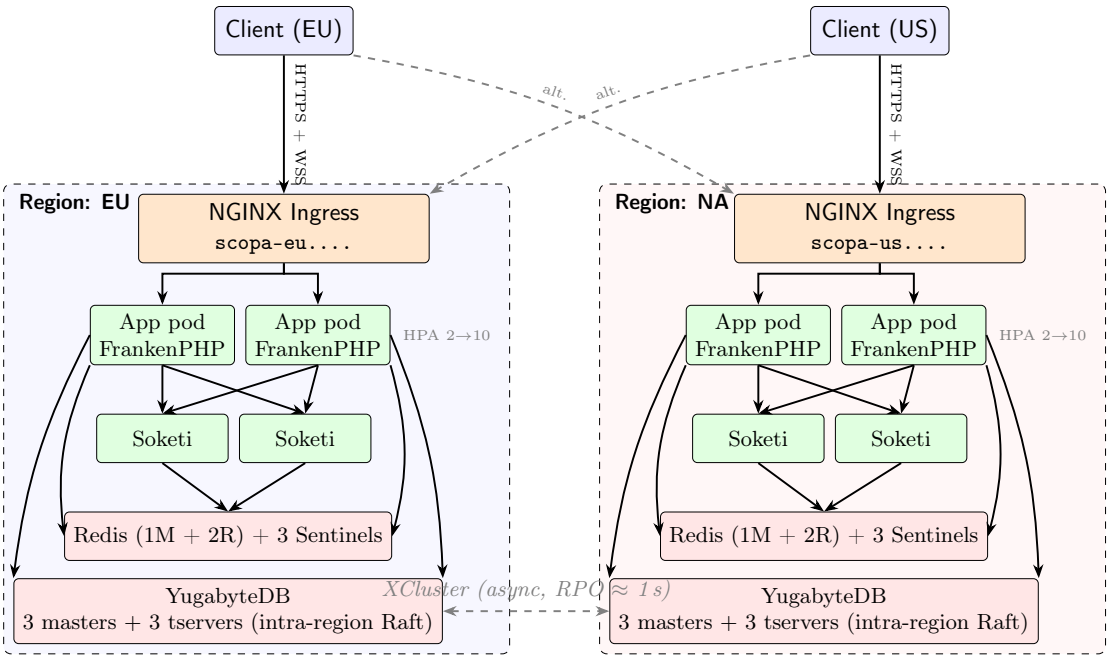


Figure 1: Global topology. Each region (EU, North America) hosts an independent stack: an NGINX ingress in front of an autoscaled fleet of FrankenPHP application pods, two Soketi WebSocket pods, a Redis group of three nodes plus three Sentinels, and a YugabyteDB cluster (three masters, three tablet servers) with intra-region Raft replication and asynchronous cross-region xCluster replication. Clients pick the closest region by probing each region's `/up` endpoint before connecting.

`octane:frankenphp` with five workers, `queue:work` for background jobs, and `schedule:work` for periodic jobs. A horizontal pod autoscaler scales the deployment between two and ten replicas on a CPU target.

**Realtime tier.** Soketi runs as a separate two-pod deployment per region. Laravel publishes to it through the in-cluster service DNS name, so broadcast traffic never leaves the cluster. The two replicas share state through a Redis adapter; without it, each pod would deliver events only to the clients attached to itself.

**Cache and coordination tier.** Each region has a Redis group of three data nodes (one master, two replicas) plus three Sentinels with quorum two. The Sentinels monitor the master and elect a replacement when it dies. Both the application (through the `phpredis` Sentinel driver) and Soketi (native Sentinel support) connect through the Sentinels rather than to a hard-coded master IP, so they inherit failover for free: when the master crashes, every client learns the new master from the Sentinels and reconnects.

**Data tier.** YugabyteDB is a distributed SQL database that speaks the PostgreSQL wire protocol but replicates each tablet through a Raft group of three peers [6]; a separate master service keeps the cluster topology. We deploy three masters and three tablet servers via the official Helm chart. The application connects with a comma-separated list of preferred hosts (the regional "local" node first), so PDO transparently fails over at connection level if the preferred node dies. Across regions, the EU and US clusters reconcile asynchronously through xCluster replication.

**Client-side discovery.** We deliberately do *not* use Geo-DNS for region selection. The Godot client ships with a static list of regional endpoints and runs an `EndpointSelector` at startup that probes each one three times and scores it as avg-ping$+100\cdot$loss%; the best reachable endpoint wins. This lets us do global routing without owning DNS authority, and — more importantly — lets the client react to a regional outage in seconds instead of waiting out a DNS TTL (Section 3.8).

## 3.3 Domain Modelling

The persistent model is small on purpose. The `users` table holds identity (id, username, `elo`). The `games` table records a match: a UUID, the two player ids (the second possibly null until the opponent joins), a sixteen-character random `seed` that fixes the deck shuffling, a status enum (`waiting_for_players` $\rightarrow$ `playing` $\rightarrow$ `finished`), and — once finished — the winner and final scores. The `game_events` table is the authoritative log: one row per move, keyed by `(game_id, sequence_number)` with a uniqueness constraint so that no two writes can ever land in the same logical slot. Each row carries the actor (`p1`/`p2`) and a textual notation inspired by chess PGN.

Seed plus event log give us *deterministic replay*: the engine always reproduces the same state from the same inputs, which is the cornerstone of our recovery story (Section 3.7).

The matchmaking state lives outside the relational database, in Redis. Two keys cooperate: a sorted set `matchmaking:queue` (members are user ids, scores are ELO ratings) and a hash `matchmaking:timestamps` mapping each queued user to the time they joined. Together they form a queue with a priority dimension: the matchmaker walks the sorted set in ELO order, while the timestamps say how patient each candidate has been.

### 3.4 Interaction Patterns

Three patterns cover all the communication in Scopa2.

**Request/response (synchronous).** The client talks to a small REST surface:

- `POST /api/auth/register`, `/login`, `/logout` — account lifecycle.
- `POST /api/matchmaking/join`, `/leave`, `GET /api/matchmaking/status` — queue management.
- `POST /api/games` and `POST /api/games/{id}/join` — manual game creation/joining.
- `GET /api/games/{id}` — fetch the canonical state of a game.
- `POST /api/games/{id}/action` — submit a move.
- `GET /api/me/active-games` — list one's unfinished games, used by the client to offer a rejoin.

All authenticated endpoints carry a Sanctum `Bearer` token, so every request is identity-bound. Responses are JSON.

**Publish/subscribe (asynchronous).** The realtime layer follows the Pusher protocol over WebSockets. The client opens one WebSocket per session, authenticates its `socket_id` against `/broadcasting/auth`, and subscribes to a few channels:

- `{user_id}_matchmaking_result` — private channel where the `match_found` event arrives.
- `game_{game_id}_player_{user_id}` — per-player channel of a match, carrying `game_state_updated` events with that player's view.
- `game_{game_id}` — shared channel for round-end and game-end events.
- `heartbeat` — public channel on which the server beats every five seconds; the client uses it as a liveness signal (Section 3.8).

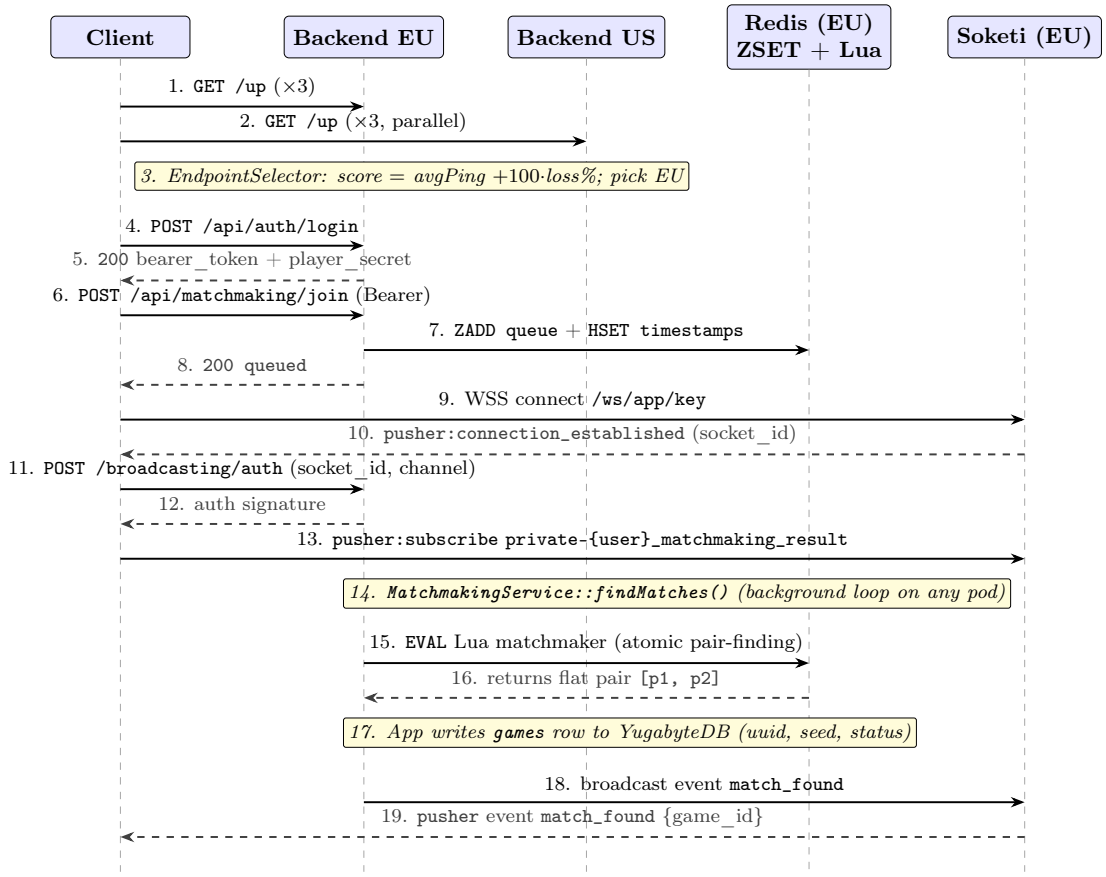


Figure 2: Discovery and matchmaking. The client runs the endpoint selector, authenticates, joins the queue, opens a WebSocket, authenticates the socket, subscribes to `{user_id}_matchmaking_result` and waits. When the Lua matchmaker pairs two compatible players it emits `match_found` with the new game id; both clients subscribe to the game channels and fetch the initial state.

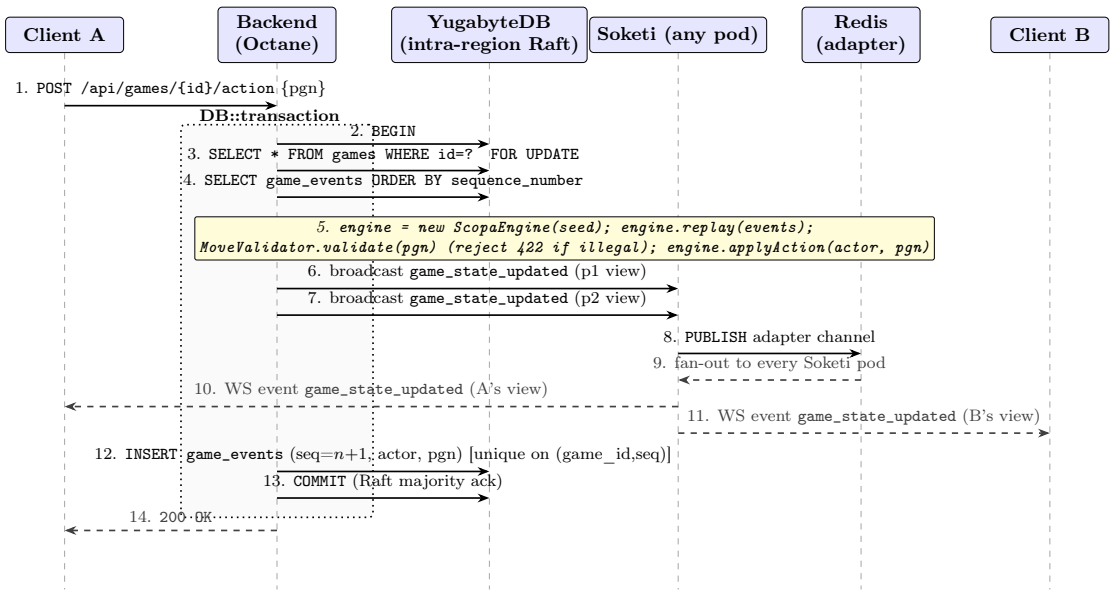


Figure 3: One in-game action. The client POSTs the move; the backend takes a row lock on the game, replays the event log, validates the move, applies it, broadcasts the two per-player views and persists the new event. Soketi receives the broadcast on one pod, fans it out through the Redis adapter, and whichever pod holds each player's WebSocket delivers the event.

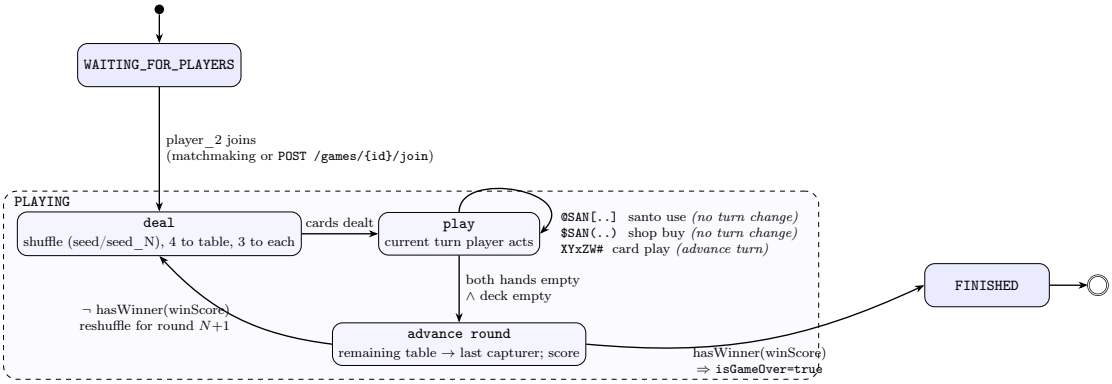


Figure 4: State machine of a match. A game starts in `WAITING_FOR_PLAYERS`, enters `PLAYING` (in the `deal` substate) when the second player joins, loops through `deal` → `play` → `advance round` until someone reaches the winning score, then ends in `FINISHED`. *Santi* effects and shop purchases do not advance the turn; only playing a card from the hand does.

**Authoritative state transfer.** On every reconnect or explicit re-sync the client calls `GET /api/games/{id}` and the server replays the full event list, returning the canonical state for that player. The protocol is therefore self-healing: a client can lose any broadcast on the wire and recover by asking for the full state.

Everything on the wire is JSON, with two pragmatic conventions: the opponent's hand is sent as a list of `"X"` placeholders (same message shape for both players, hidden cards), and card mutations are a flat map from the original card code to its currently visible one — e.g. `{"4C":"4D"}` after a *San Biagio* effect turned a four of cups into a four of coins.

## 3.5 Behaviour

The engine rebuilds its in-memory state by replaying the persisted events from the beginning. The shuffle is deterministic: the RNG is seeded with a CRC32 of the game seed for the first round and of `seed_round_N` for the following ones, so the same inputs always deal the same cards. The PGN-inspired notation has three kinds of action: a card play (`7Dx7C#`, the optional `#` marking a *scopa*), a Santo invocation (`@SAN[params]`) and a shop purchase (`$SAN(3C+4D)`). Only card plays advance the turn. When a round ends (both hands and the deck empty) the engine assigns the remaining table cards to the last capturer and computes the round score (denari, settebello, primiera, allungo, scopas); if nobody has reached the winning score it reshuffles and starts a new round, otherwise it marks the game finished. The engine exposes two callbacks (`onRoundEnded`, `onGameEnded`) which the controller wires to the corresponding broadcast events; the game-end event also triggers the ELO update (Section 4).

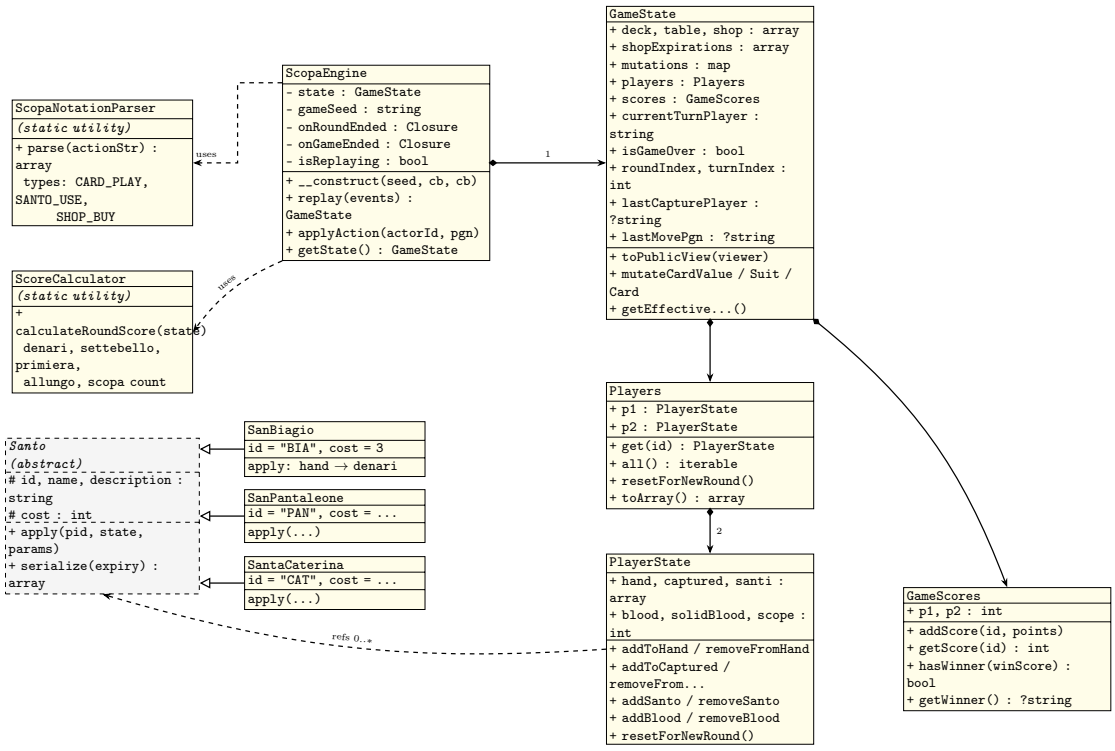


Figure 5: Classes of the in-memory game engine. `ScopaEngine` owns a `GameState` and delegates to the notation parser, the move validator, the score calculator and the `Santo` hierarchy. The engine has no dependency on Laravel or on the database: it is a pure library, which is what makes it trivially testable and replayable anywhere.

### 3.6 Data and Consistency Issues

**Atomic matchmaking.** The one place where application pods race for the same shared resource is the matchmaking queue. A naive implementation would run a sequence of Redis commands from PHP — read the ZSET, pick a pair, remove both members — but any interleaving between two pods could double-match a player, or remove one without ever matching them. We solve this by pushing the whole decision into Redis as a single Lua script: it walks the queue in ELO order, computes each candidate's tolerated ELO gap as a function of their waiting time, finds a partner acceptable to *both* windows, and removes both players atomically. Redis executes a Lua script without interleaving any other command, which makes the critical section a true atomic step no matter how many pods are running. Listing 1 shows the algorithm.

```
players  = ZSET matchmaking:queue, sorted by ELO
paired   = ∅
for each p1 in players (in ELO order):
    if p1 ∈ paired: continue
    w1 = elo_window(wait_time(p1))    // widens with patience

    for each p2 in players coming after p1:
        if p2 ∈ paired: continue
        w2 = elo_window(wait_time(p2))

        if |p1.elo - p2.elo| ≤ min(w1, w2):
            ZREM queue p1, p2
            HDEL timestamps p1, p2
            emit pair (p1, p2)
            paired ← paired ∪ {p1, p2}
            break
```

Listing 1: The matchmaking step, as pseudocode. The real version is a Lua script executed server-side by Redis, atomically.

The symmetric window check matters: a freshly-joined high-ELO player is not paired with a low-ELO player just because the latter has waited long enough to accept anyone — both must accept each other.

**Per-game serialisation.** The second contended resource is the event log of a single game. Here we use the database: the action handler runs in a transaction that takes a row lock on the game (`SELECT ...FOR UPDATE`), so concurrent moves on the same match are serialised, and the uniqueness constraint on `(game_id, sequence_number)` is the safety net that makes a double-write impossible even through a buggy code path. Within one match, `sequence_number` gives us a total order of moves — morally a Lamport clock [5] whose ticks are produced under a lock instead of being merged from concurrent processes.

**CAP and PACELC.** The system spans more than one consistency domain and treats them differently [1, 4]. Inside a region the data tier is strongly consistent: a YugabyteDB tablet leader needs a Raft majority before acknowledging a write, so we sit in the CP

corner — a write to the event log either commits cluster-wide or fails visibly; no stale reads, no split brain. Between regions, replication is asynchronous (xCluster): the two clusters reconcile in the background, so a reader on the other side may briefly see an old state, and losing a whole region may lose the tail of the last replication batch. We accept this AP behaviour because a match never spans regions — a game is always fully owned by one of them. The realtime tier, finally, is deliberately best-effort: a lost broadcast is fine because the client can always pull the canonical state with `GET /games/{id}`. In PACELC terms, the "else" branch goes the same way: intra-region we pay a few milliseconds of Raft quorum to get linearisable writes; inter-region we choose latency and never block a European write on its American mirror.

**Why no distributed snapshots?** We considered the Chandy–Lamport construction [2] during design and concluded we do not need it: the only globally consistent cut we ever care about is the per-match event log, which is already a totally ordered sequence inside a single Raft group.

## 3.7 Fault Tolerance

We designed for the following failure modes, roughly from most to least frequent.

**Application pod death.** The tier is fully stateless: every request rebuilds the game from the log. Kubernetes reschedules the pod, the ingress drops it from rotation, and the client's HTTP retry lands on a healthy peer.

**Soketi pod death.** The client loses its WebSocket, reconnects to the same region, and the ingress routes it to the surviving pod. Thanks to the shared Redis adapter, events produced during the reconnect window are delivered on whichever pod the client lands on.

**Redis master crash.** The Sentinels detect the silence, elect a replica as the new master, and both Laravel and Soketi rediscover the master through the Sentinel protocol. A few in-flight pub/sub messages may be lost in the gap; that is acceptable because `GET /games/{id}` is always available as the resync path.

**YugabyteDB tablet-leader crash.** The Raft followers stop receiving heartbeats, elect a new leader and resume serving the shard. The driver retries on the next host of its preferred-host list; the application sees a latency spike of a few seconds at worst.

**Whole-region outage.** The expensive case. Three things happen in sequence: the heartbeat channel goes silent and after fifteen seconds the client marks the region as failed and connects to the next endpoint in its priority list; the new region serves the match state as last replicated through xCluster; the client re-subscribes to the game channels and play resumes. Crucially, no operator action is involved — the global failover is entirely client-driven.

**Recovery = deterministic replay.** All of the above relies on one property: a game can always be rebuilt from its log. Conceptually this is pessimistic message logging [3]: every external input (every move) is synchronously persisted before its effect becomes visible to anyone, and since the engine is deterministic (the RNG is seeded from the game seed) replaying the inputs reproduces the state exactly. The cost is that replay grows linearly with game length; a Scopa match stays within a few hundred events, so full replay is cheap and we deliberately skipped snapshotting (it is listed as future work for longer-lived game modes).

## 3.8 Availability

What the player perceives as "the game never goes down" is the combination of a few independent mechanisms.

**Redundancy everywhere.** The autoscaler never goes below two application replicas, Soketi always has two pods, Redis and YugabyteDB keep three copies of everything. Rolling updates use `maxUnavailable: 1` / `maxSurge: 1`, so a deployment under traffic always has healthy pods serving.

**Heartbeat-driven failure detection.** The backend broadcasts a beat on the public `heartbeat` channel every five seconds. The client tracks the time since the last beat and triggers a failover after fifteen seconds of silence (three missed beats). This sidesteps the classic problem of distinguishing a slow server from a dead one: as long as beats arrive, the client assumes the server is fine even if no game events flow.

**Client-side priority list.** The endpoint selector ranks all regions at startup, producing a priority list. On failover the current endpoint is added to a failed set and the client switches to the best remaining one; a five-second cooldown between failovers prevents flapping when two regions time out back to back. If every endpoint is marked failed, the client reports a global outage to the user.

**Channel re-subscription.** The client keeps the set of channels it should be subscribed to. On every WebSocket reconnect — same pod, new pod or new region — it re-authenticates and re-subscribes each one, then re-fetches the game state. This is what makes failover seamless: the subscriptions follow the player wherever they land.

In practice, an attentive player notices a one-to-two-second freeze of the opponent's cards during a failover before the board re-syncs; a distracted one notices nothing.

# 4 Implementation

## 4.1 Technology Stack

| Layer | Component | Technology |
|---|---|---|
| Client | Game UI | Godot 4.6 (C# / .NET 8) |
| Client networking | HTTP + WebSocket | Godot `HttpRequest`, custom Pusher impl |
| Edge | Ingress | NGINX (K3s default) |
| Application | Web framework | Laravel 12 on Octane + FrankenPHP |
| Application | Broadcasting driver | `pusher/pusher-php-server` |
| Application | Redis client | `phpredis` + Sentinel adapter |
| Realtime | WebSocket broker | Soketi (Pusher-compatible) |
| Coordination | Pub/Sub + matchmaking | Redis 7 + Sentinel |
| Storage | Distributed SQL | YugabyteDB (3 master, 3 tserver) |
| Orchestration | Container runtime | K3s (lightweight Kubernetes) |
| Orchestration | Packaging | Helm chart with regional values |
| CI/CD | Build + deploy | GitHub Actions, GHCR, Tailscale, SSH |

Table 1: Technology choices by architectural layer.

## 4.2 Repository Layout

- `scopa2-backend/` — the Laravel application: `app/` (`GameEngine`, `Services`, `Http/Controllers`, `Events`, `Jobs`, `Models`), `config/` (Redis Sentinel and broadcasting settings), `database/migrations/`, `helm/` (chart plus per-region values) and `tests/` (Pest 4 unit and feature suites).
- `scopa2-game/` — the Godot project; the C# scripts live in `scripts/`. The interesting ones for this report are `NetworkManager.cs` (HTTP/WebSocket plumbing, failover, channel resubscription), `EndpointSelector.cs` (ping-based region discovery), `PusherClient.cs` (hand-rolled Pusher protocol) and `MainGame.cs` (UI flow and state synchronisation).

## 4.3 Matchmaking

The subsystem has an ingestion side (the join/leave/status endpoints in `MatchmakingController`, which write to the Redis keys of Section 3.3) and an extraction side: the Lua script of Listing 1, wrapped by `MatchmakingService::findMatches()`. The window parameters are configurable; the defaults start at ±100 ELO and widen by 50 every five seconds of waiting, capped at 500.

Extraction is driven by Laravel's scheduler: a `ProcessMatchmakingQueue` job runs every second, and the heartbeat job every five seconds. Every pod runs a scheduler process, so we mark both jobs `onOneServer()`: Laravel takes a short-lived lock in Redis before executing, which turns "every pod fires the same cron" into "exactly one pod fires

it per tick". It is a small thing, but it is a textbook distributed mutex, and we got it from the framework for free.

For each pair returned by the script, the service creates the game row (with its random seed) and broadcasts `match_found` on both players' private channels.

### 4.4 Game-Engine Replay

The backend's core property is that the state of any match is a function of the persisted log only. Listing 2 is the whole replay primitive.

```
public function replay(array $historyEvents): GameState
{
    foreach ($historyEvents as $event) {
        $this->applyAction($event->actor_id, $event->pgn_action);
    }
    return $this->state;
}
```

Listing 2: Deterministic replay. The engine is constructed with the game's seed, then applies each persisted event in order.

Replay runs on every `GET /games/{id}`, at the start of every action handling (to validate the incoming move against the current state) and throughout the unit tests. Because the engine is pure, the same event list produces the same state in any process, in any region, at any time.

### 4.5 Action Handling and Concurrency Control

Listing 3 shows the action handler, slightly trimmed. The transaction plus `lockForUpdate()` serialises concurrent moves on the same game; the unique constraint on `(game_id, sequence_number)` backs the lock up. Two details implement requirement F4 (untrusted clients): the move is checked by `MoveValidator` against the replayed state before being applied, and the *scopa* flag (`#`) is recomputed server-side — the client cannot claim a scopa that did not happen.

```
return DB::transaction(function () use ($gameId, $action) {
    $game = Game::where('id', $gameId)->lockForUpdate()->firstOrFail();
    $events = GameEvent::where('game_id', $gameId)
        ->orderBy('sequence_number')->get();

    $engine = new ScopaEngine($game->seed, $onRoundEnded, $onGameEnded);
    $engine->replay($events->all());

    $playerIndex = $this->getLoggedPlayerIndex($game);

    $validator = MoveValidator::fromState($engine->getState(),
        $playerIndex);
    if (!$validator->validate($action)) {
        return response()->json([...], 422);   // illegal move rejected
    }
    $action = $this->normalizeScopaFlag($action, $engine->getState());
```

```
16
17      $engine->applyAction($playerIndex, $action);
18
19      if ($engine->getState()->isGameOver) {
20          $game->status = GameStateEnum::FINISHED;    // + winner, final
                scores
21          $game->save();
22      } else {
23          broadcast(new GameStateUpdated($game->id,
24              $engine->getState()->toPublicView('p1'),
                    $game->player_1_id));
25          broadcast(new GameStateUpdated($game->id,
26              $engine->getState()->toPublicView('p2'),
                    $game->player_2_id));
27      }
28
29      GameEvent::create([
30          'game_id'          => $gameId,
31          'sequence_number' => $events->count() + 1,
32          'actor_id'         => $playerIndex,
33          'pgn_action'       => $action,
34      ]);
35      return response()->json(['status' => 'success']);
36  });
```

Listing 3: Action handler (trimmed). Row lock + replay + validation + append, all inside one transaction; the broadcasts carry each player's own view of the new state.

**ELO update.** When the engine reports game over, a `GameFinished` event fires and a listener updates both ratings (classic ELO, $K = 32$) in a single transaction that row-locks both user records — requirement F7's atomicity comes from the same mechanism as the event log's.

### 4.6 Realtime Wiring through Sentinel

Soketi receives, via environment variables templated by Helm, a JSON list of the regional Sentinels instead of a master IP — both for its adapter (cross-pod fan-out) and for its internal state (Listing 4). Nobody in the system ever knows the master's address statically; everyone discovers it through the Sentinels at connection time and rediscovers it after a failover.

```
1  - name: SOKETI_ADAPTER_DRIVER
2    value: "redis"
3  - name: SOKETI_ADAPTER_REDIS_INSTANCE_NAME
4    value: "mymaster"
5  - name: SOKETI_ADAPTER_REDIS_SENTINELS
6    value: '[{"host":"redis-sentinel-0...","port":26379},
7              {"host":"redis-sentinel-1...","port":26379}]'
8  - name: SOKETI_DB_DRIVER
```

```
  value: "redis"
- name: SOKETI_DB_REDIS_SENTINELS
  value: '[{"host":"redis-sentinel-0...","port":26379},
           {"host":"redis-sentinel-1...","port":26379}]'
```

Listing 4: Excerpt of the Soketi environment: Sentinel lists instead of a master IP.

## 4.7 Client-Side Failover

The client's `NetworkManager` keeps the user connected without exposing any of the machinery. Listing 5 shows the endpoint scoring used both at startup and to build the failover priority list.

```
public double Score => IsReachable
    ? AveragePingMs + (PacketLoss * 100)
    : double.MaxValue;

public async Task<ServerEndpoint> SelectBestEndpoint(
        ServerEndpoint[] endpoints) {
    var metrics = await GetEndpointMetrics(endpoints);
    var reachable = metrics.Where(m => m.IsReachable).ToArray();
    if (reachable.Length == 0) return endpoints[0]; // last-resort
        fallback
    return reachable.OrderBy(m => m.Score).First().Endpoint;
}
```

Listing 5: Endpoint scoring and selection. The $100 \cdot \text{loss}$ penalty punishes unreliable endpoints even when their average latency looks good; unreachable endpoints sort last.

A failover fires when HTTP retries are exhausted, when channel authentication keeps failing, or when the heartbeat has been silent for fifteen seconds. The routine marks the current endpoint as failed, picks the best remaining one, reconnects the WebSocket, re-authenticates and re-subscribes every active channel, and finally re-fetches the game state. If connecting to the new endpoint fails too, it recurses down the priority list until it runs out of servers.

## 4.8 Continuous Integration and Deployment

The pipeline (`.github/workflows/deploy.yml`) runs on every push to `main` as three sequential jobs. The first job runs the Pest suite against a throwaway SQLite database and a Redis service container; it is a hard gate, since both the image build and the deploy declare it as a `needs` dependency, so a single failing test stops the pipeline before anything reaches production. Only if the tests pass does the second job build the Docker image tagged with the commit SHA and push it to the GitHub Container Registry. The third job then joins the private network through an ephemeral Tailscale node, SSHes into the K3s host and runs `helm upgrade` for both regions, waiting for each rollout to complete. Rolling updates mean production never sees a zero-replica window, and a failed rollout

triggers an automatic Helm rollback to the previous release. End to end the pipeline takes five to seven minutes.

# 5 Validation

Validation happened on two levels: automated tests for fast feedback during development, and manual fault injection on the live deployment to check that the system actually behaves as designed when things die.

## 5.1 Automated Testing

The backend suite is written in Pest 4, split into unit tests (the pure game engine and its support classes) and feature tests (the full Laravel application driven over HTTP against test database and Redis instances). The coverage that matters most:

- `Unit/` — `ScopaEngineTest`, `ScoreCalculatorTest`, `ScopaNotationParserTest`, `MoveValidatorTest`, `GameStateTest`, `PlayerStateTest`, `EloServiceTest`, plus one test per Santo (`SanBiagio`, `SanPantaleone`, `SantaCaterina`).
- `Feature/MatchmakingTest` — enqueue/dequeue, the HTTP endpoints, queue-size assertions, duplicate prevention.
- `Feature/GameTest` — creation, join, state retrieval (including hand-visibility checks), captures, discards and the boundary conditions of the action handler.
- `Feature/RejoinTest` — reconnecting to an active game (requirement F6).
- `Feature/EloUpdateTest` and `Feature/AuthTest` — rating updates at game end and the authentication flows.

## 5.2 Acceptance and Chaos Testing

We exercised the deployment with manual fault injection from outside the cluster — killing pods and scaling deployments while a real match was being played. Table 2 summarises the matrix.

# 6 Deployment

Production is a single K3s cluster on a dedicated home-lab host, reachable over Tailscale by both the CI pipeline and the team. The public surface is two HTTPS endpoints, one per region:

- `https://scopa-eu.murkrowdev.org`
- `https://scopa-us.murkrowdev.org`

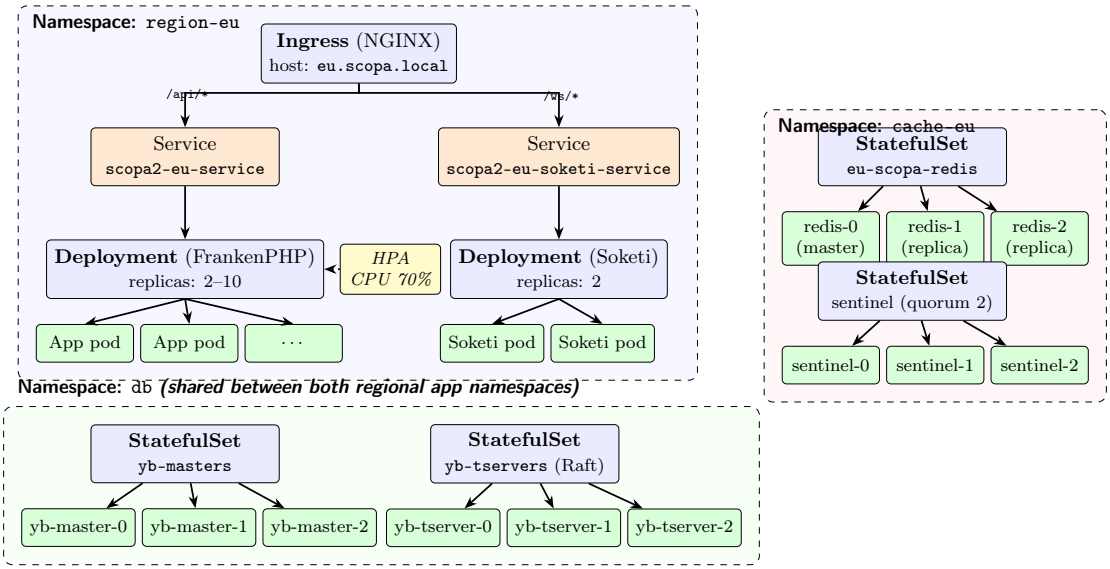


Figure 6: Kubernetes resources of one region (the other is identical by construction, since both are rendered from the same Helm chart). The compute namespace holds the ingress, the autoscaled application deployment and the Soketi deployment; the cache namespace holds the Redis nodes and Sentinels; the shared `db` namespace holds the YugabyteDB masters and tablet servers.

| Tier | Action | Expected outcome | Result |
|---|---|---|---|
| Application | Kill one app pod | LB drops failed pod, retry on peer pod returns 200 | `PASS` |
| Application | Kill all app pods in a region | Client times out, fails over to other region, state preserved | `PASS` |
| Realtime | Scale Soketi to one replica then back to two | Clients reconnect on the surviving pod | `PASS` |
| Cache | Kill a Redis follower | No visible effect | `PASS` |
| Cache | Kill Redis master | Sentinels elect a new master, traffic resumes | `PASS` |
| Storage | Kill a TServer leader | Raft re-election, query latency spike $\sim$3 s | `PASS` |
| Storage | Reduce TServer to one replica | Writes fail; reads via follower-reads, manual intervention required | `PASS` |
| Global | Delete an entire region namespace | Client failover to surviving region | `PASS` |

Table 2: Chaos test matrix, verified by hand on the live cluster while a client was actively playing.

For local development we run the same Helm charts on a single-node Minikube. Caddy on the developer's machine reverse-proxies ports `:8080`/`:8081` to the Minikube IP with the right `Host` header, so the very same NGINX ingress dispatches to either "region" just by inspecting the header. The dev environment is shared with the rest of the team over Tailscale, without opening any port to the public Internet.

# 7 User Guide

1. Download the build for your platform from the project's releases page (Windows x86_64, macOS Universal, Linux x86_64/ARM64).
2. Launch the client. On first run it probes the regional endpoints, picks the fastest reachable one and shows the registration screen.
3. Choose a username and register. Credentials are stored locally in `user://auth.cfg`, so subsequent launches go straight to the main menu.
4. Click *Find Match*. The status bar shows the chosen region and connection state; when an opponent is found the client switches to the board automatically.
5. Play. Under normal conditions every move shows up on the opponent's screen well under a second.

6. If the network or a region misbehaves, a brief *Server Switch* overlay appears while the client migrates to the fallback region; the game resumes by itself.

Logout is available from the main menu and clears the stored credentials. There is no password-recovery flow yet, since authentication is currently username-only (Section 9).

# 8 Self-Evaluation

## 8.1 Work Split

We met in person almost every time and worked from the same workstation for most of the project, so the planning, the architecture and the bulk of the implementation — backend and client alike — were done together; no controller or scene was "owned" by one of us, and this report was also written jointly.

The split shows only where the work needed hardware or context that one of us had at hand:

- **Marco** owned the production side: the K3s cluster on his home-lab server, the per-region Helm releases, the reverse-proxy and domain setup, and the Tailscale-bridged CI/CD pipeline.
- **Valerio** owned the client-side recovery manager in Godot: ping-based endpoint selection, heartbeat-driven failover, channel resubscription on reconnect, and the manual chaos sessions proving that the flow really hides regional outages from the player.

## 8.2 Reflections

The most rewarding lesson is that many canonical distributed-systems mechanisms do not need to be implemented *by us* to be used well. Raft consensus [6] lives inside YugabyteDB, leader election is delegated to Redis Sentinel, scaling and rolling updates come from Kubernetes, and even the distributed lock behind `onOneServer()` comes from the framework. What we actually own is the glue: the schema that produces a total order of moves, the Lua script that makes matchmaking atomic, the deterministic engine that turns recovery into replay.

# 9 Future Works

- **Periodic checkpointing.** For longer game modes, snapshots of the engine state would bound the replay cost and allow pruning old event-log entries.
- **Cross-region matchmaking.** The queue is per-region today. A global queue would need a causality scheme to handle concurrent joins from the two regions safely.
- **Stronger authentication and User Profiles.** Allow users to use passwords for registration and provide them the possibility to access old matches and statistics.

## References


[1] Eric A. Brewer. Towards robust distributed systems (invited talk). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2000. doi: 10.1145/343477.343502.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985. doi: 10.1145/214451.214456.

[3] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002. doi: 10.1145/568522.568525.

[4] 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. doi: 10.1145/564585.564601.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi: 10.1145/359545.359563.

[6] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 305–319. USENIX Association, 2014.