MarafoNet, Marafone on the net

Final Report for the Distributed Systems Course 2025/2026

Bagattoni Federico
federico.bagattoni@studio.unibo.it

Turchi Jacopo
jacopo.turchi@studio.unibo.it

June 6, 2026

In this report we document the concept, design and testing of MarafoNet, a digitalised version of *Marafone* a traditional card game from Italy.

The system's design is focused on distribution as its main aspect, elicitating the benefits that this approach brings and defining requirements to be satisfied both for distribution and user-related. The features of the game are explained and modeled in an architecture coherent with the requirements cited. Security and scalability issues are addressed. Implementation strategies and integration between technologies are shown to achieve requirements and the architecture planned. Finally, tests are displayed to verify the correctness of the system's operations as well as the insfrastructure components used.

A guide will help the user deploy, use the system and understand the rules of the game.

Disclaimer

During the preparation of this work, the author(s) used GitHub Copilot and Google Gemini for debugging and prototyping purposes.

After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Concept

Product Type

This project is a web application accessible via a GUI.

Use Case

The system digitalises the turn-based card game Marafone, applying the Emilia Romagna rules, to provide global accessibility. Users play via a web interface optimised for

laptops and desktops. The interaction model is turn-based, with players executing an action approximately every 30 seconds. System interaction requires account registration, but data storage is strictly limited to user credentials (username and password), with no additional information retained.

Need for Distribution

Since the user base is concentrated in central Italy, geographical distribution is a secondary concern. However, a distributed architecture is required to:

- Guarantee **data integrity** and robust state management across the system.
- Enable **parallel computation** for game logic and concurrent matches.
- Maintain **system stability and fault tolerance** under heavy load and in the case of node failures.

2 Requirements elicitation and analysis

Glossary

- **Game:** A complete playing session that concludes with the win of a team.
- **Match:** A single round within a Game, starting with the dealing of cards and ending when players run out of cards.
- **Trump suit:** The dominant suit designated for the current Match, chosen by the first player. Cards of this suit outrank all cards of the other three suits.
- **Current Player:** The player who currently holds the right to execute an action (playing a card or choosing the Trump suit).
- **Trick:** A sequence where each player plays exactly one card. The Trick is won by the player who played the highest value card of the leading suit or the highest Trump card.
- **Marafona:** A specific combination of cards (Ace, Two, and Three of the Trump suit) held by the first player in a match, which awards bonus points.

Functional Requirements

The system must allow the user to perform the following actions. Each requirement is tied to a specific acceptance criterion (AC) for validation purposes:

F1. **Account Registration:**

- Register a new account using a unique username and password.

- *AC:* The system stores the credentials and allows the user to log in immediately after.

F2. **Authentication:**

- Log in using registered credentials.
- *AC:* The system grants access upon submitting correct credentials and explicitly denies access upon incorrect ones.

F3. **Matchmaking:**

- Join a queue to participate in a Game, and leave the queue if desired.
- *AC:* The system places the user in an active Game as soon as the required number of players is reached in the queue.

F4. **Game Interaction:**

- Execute valid in-game actions (e.g. playing a card, choosing the Trump suit) during the user's turn.
- *AC:* The system accepts the action, updates the game state, and passes the turn to the next player.

F5. **State Visibility:**

- View the real-time game state, including the current turn, active Trump suit, current score, and the cards played in the last Trick.
- *AC:* The client interface accurately reflects the server-side state without requiring manual page reloads.

F6. **Move Validation:**

- Receive immediate feedback if an attempted move violates the game rules.
- *AC:* The system rejects the action, maintains the current turn, and displays an error message explaining the rule violation.

F7. **Game Resolution:**

- View the final results when a Game concludes.
- *AC:* The system halts gameplay and displays the winning team along with the final score differential.

F8. **Timeout Management:**

- Resolve the game without waiting indefinitely if an opponent disconnects.
- *AC:* The system starts a grace period timer upon a player's disconnection. If the player fails to reconnect before the timer expires, the system ends the game, registers a forfeit for the disconnected player, and declares the opposing team the winner.

F9. **Post-Game Actions:**

- Choose to queue for a new Game or quit to the main menu after a Game ends.
- *AC:* The system routes the user to the matchmaking queue or to the login page based on their explicit selection.

Non-Functional Requirements

The system must guarantee the following properties:

NF1. **Consistency:** The game state must be identical for all players at any given time. Following a network partition, reconnecting users must retrieve the exact or most recent valid game state.

NF2. **Fault Tolerance:** The system must withstand node failures without data corruption or loss. In the event of a crash, users must be able to reconnect seamlessly and resume their game without interruption.

NF3. **Availability:** The service must remain accessible; if a node fails, traffic must be routed to another available source to maintain uptime.

NF4. **Client Statelessness:** The client application must not store internal game state memory. It must depend entirely on state updates pushed by the server, ensuring the server remains the single source of truth and preventing client-side state manipulation.

Relevant Distributed System Features

The following distributed system features are explicitly relevant to the project:

- **Fault Tolerance and Availability:** Essential. The system must recover from node failures automatically to maintain uninterrupted gameplay and prevent game state corruption.
- **Transparency:** Essential. Players must perceive the application as a single, centralized entity. The underlying infrastructure, including load balancing, state replication, and failure recovery, must remain invisible to the end user.
- **Concurrency and Resource Sharing:** Essential. The system must manage parallel access to the shared game state, ensuring synchronized and consistent updates as multiple users execute actions across concurrent matches.
- **Security:** Moderately relevant. The system requires standard authentication (username and password) to verify player identity and track match participation, but it does not process or store sensitive personal data.

- **Scalability:** Moderately relevant. Although the regional nature of the game limits expected user growth, the system must accommodate potential traffic spikes. The architecture must support horizontal scaling, allowing the dynamic addition of server instances to distribute heavy loads without degrading response times.

The following features are considered not relevant for the scope of this project:

- **Openness and Interoperability:** The application is fully self-contained. It does not integrate with external legacy systems, nor does it expose public APIs for third-party consumption.
- **Economy and Costs:** The system is developed as an academic exercise. Consequently, commercial budget constraints and strict resource-usage minimization for hosting costs are not primary design drivers.

3 Design

3.1 Architecture

The base architecture of the system is client-server where the client's state is totally dependent on the server's updates. In this way the server is the only source of truth in the system and there's not way for the client to change the game's state at its own will.

The design of the server's side of the system follows the microservices pattern. This is beneficial for peformance and availability because different services may use a different amount of resources and so can be provisioned and deployed differently. It also enhances overall robustness: by decoupling the microservices, a failure in one component will not affect the others. For instance the game server microservice has to manage all the connections to the client, whereas the matchmaking microservice is not externally exposed and only watches the user's queue so it will be provisioned with far lower resources.

3.2 Infrastructure

Given the microservices architecture of the backend and the need for distributed systems features, the appropriate approach to infrastructure is to distribute the services in a cluster of nodes. The components that need to be introduced are:

- the clients, obviously, that interact with the server
- the game server's microservice, deployed with three replicas and with the possibility to be scaled up
- the matchmaking microservice, deployed with one replica due to the little work it has to do relative to the number of users
- the database, deployed with multiple instances, ideally three

- one load balancer, that would distribute incoming connections to the microservices. This component may also serve as terminator for SSL/TLS connections before entering the cluster.

The services are distributed in a cluster composed of three or more nodes. These nodes could be deployed through a cloud provider or on bare metal machines. They must not live in the same node to ensure fault tolerance. Considering it is a popular game in Emilia-Romagna and neighbor regions most of the users will come from there, with a little percentage of traffic coming from expatriates.

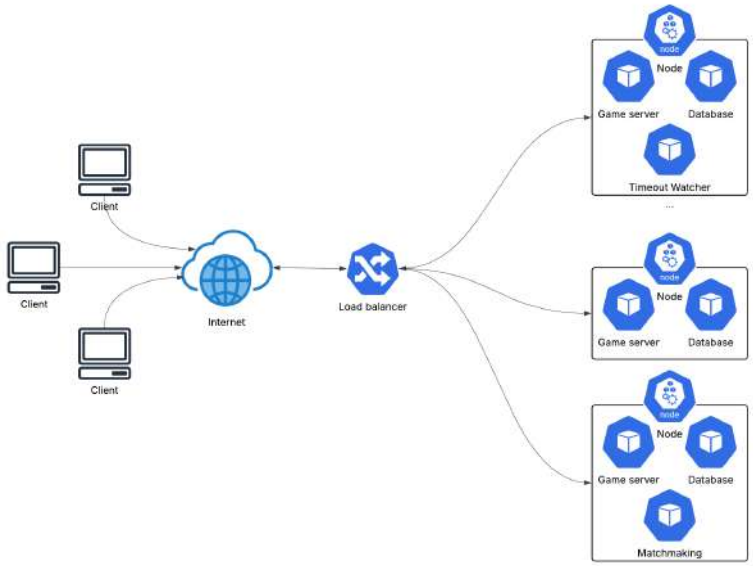


Figure 1: Deployment diagram of the system

3.3 Modelling

This section formalizes the system's structural design through UML class diagrams, focusing exclusively on the core domain entities and the service-layer dependencies.

Domain Model

The domain model encapsulates the strict ruleset and state of the Marafone game (Figure 2):

- **Game and Players:** A `Game` entity aggregates exactly four `Player` objects to form a 2v2 match.
- **Identity Decoupling:** The persistent account entity (`User`) is intentionally omitted from the core domain model. The system enforces strict decoupling between authentication and gameplay. A `Player` exists exclusively in the context of a `Game`.

It identifies the participant via a string name, completely isolating the game rules and state from the user credentials.

- **Deck and Cards:** The `Deck` is a strict composition of 40 `Card` entities. Each `Card` is defined by a numeric value and a `Suit`.
- **State Encapsulation:** The `Game` struct serves as the aggregate root. It uniquely manages the current turn logic, the active Trump suit, and the trick score, preventing invalid mutations from external services.

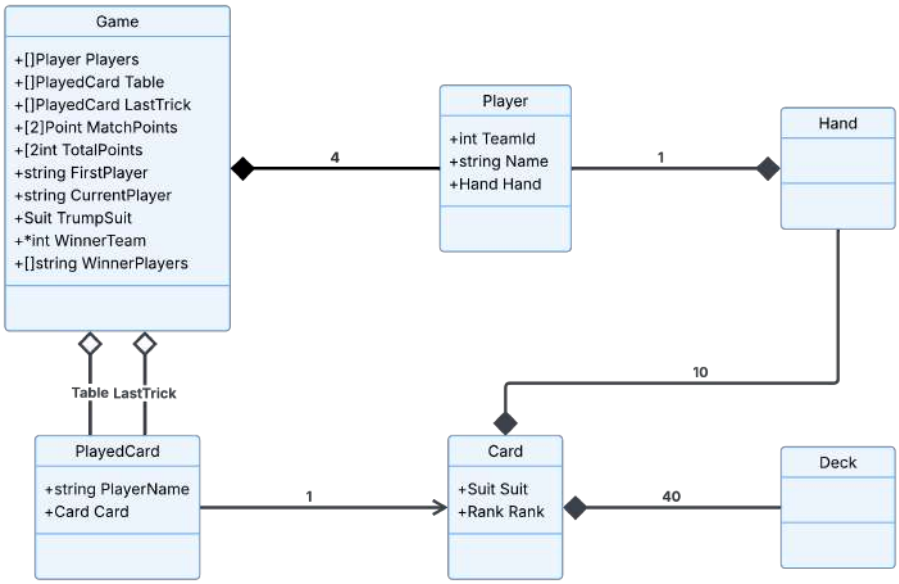


Figure 2: UML Class Diagram of the Core Domain Model

Service Architecture

The service layer orchestrates system behaviour, network interactions, and state persistence (Figure 3). The architecture enforces strict separation of concerns via distinct managers:

- **ConnectionServer:** Manages client connections. It is responsible for routing incoming payloads (actions) to the relevant services and broadcasting state updates back to connected clients.
- **MatchmakingHub:** Manages the queuing logic. It aggregates incoming user requests and instantiates a new `Game` context strictly when four players are available.
- **TimeoutWatcher:** Monitors the storage layer for expired user connection leases. Upon detecting a timeout event for a disconnected player, it commands the `GameService` to forfeit the match, ensuring games do not stall indefinitely.

- **GameService:** It receives processed actions, validates them against the domain model rules, and mutates the game state.
- **StorageService:** Abstracts the distributed key-value store interactions. `GameService` depends on this component to fetch and persist the game state to guarantee data integrity.

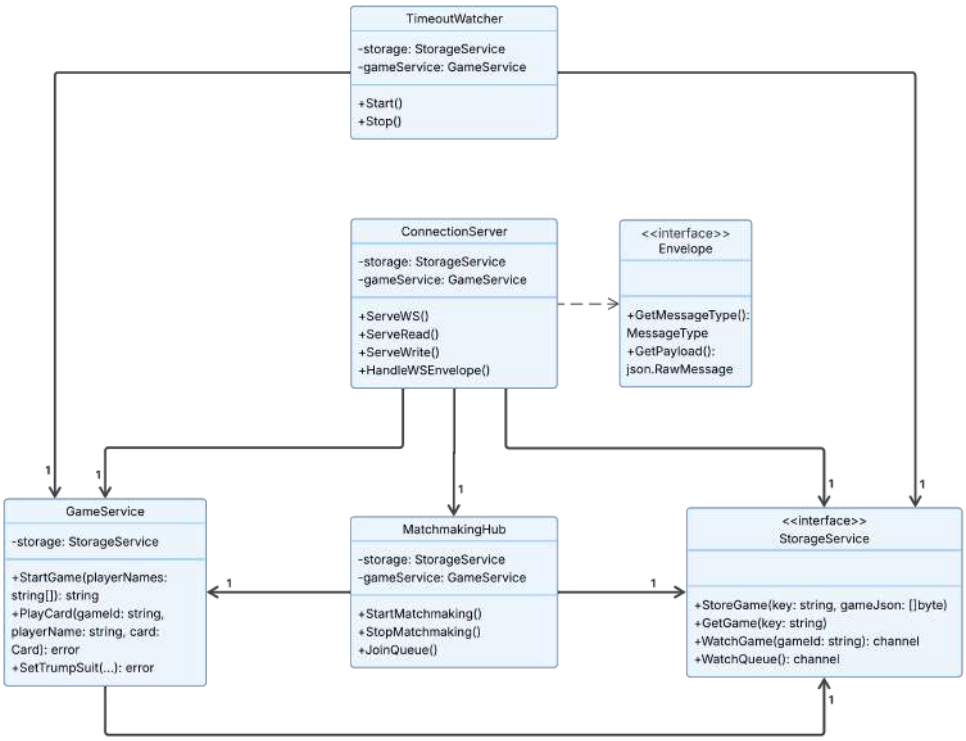


Figure 3: UML Class Diagram of the Service Layer Dependencies

State Management and Infrastructure Mapping

The architecture enforces a strict separation between transient connection data and persistent application state:

- **Client Layer:** Clients handle the visual representation and user inputs but hold no authoritative game state.
- **Application Server (Stateless):** The core services (`GameService` and `MatchmakingHub`) execute logic statelessly. The `ConnectionServer` only holds transient routing information to map active connections to users.
- **Storage Layer (Stateful):** The distributed database serves as the single source of truth. It exclusively holds the authoritative state of ongoing `Game` entities, the matchmaking queue, and user credentials.

Domain Events and Messaging

System interactions rely on a message-driven architecture to communicate state changes across the distributed components. The exchanged messages fall into three categories:

- **Commands:** Actions initiated by the client requesting a state mutation. Event examples include *RegisterUser*, *LoginUser*, *JoinQueue*, *PlayCard*, *ChooseTrumpSuit*, *PlayAgain*, *Quit*.
- **Messages:** Server-generated messages notifying clients that a move was invalid or that the game state was updated.
- **Queries:** Internal server requests directed to the `StorageService` to read or monitor data, such as fetching the current game state or checking the matchmaking queue size.

3.4 Interaction

Interactions in the system can be broken down into:

- **Client-Server Interactions:** the commuications happen through the classic Request-Response pattern between the Clients and the Game Server.
- **Server-Storage Interactions:** these interactions are based on queries to the database, the underlying implementation of the commucations is up to the technology chosen during the implementation phase
- **Server-Server Interactions:** the microservices will interact with each other to coordinate the game's dynamics but they must not interact directly, instead they will interact through the database. This is because the components are desinged to be stateless and distributed so keeping track of a state in memory would be a problem.

An example of interaction of the first point is when a client plays a card, represented in the Figure 4, from this diagram we can see that, once the play is confirmed legal, the update to the database and the notification to the client passes through the database.

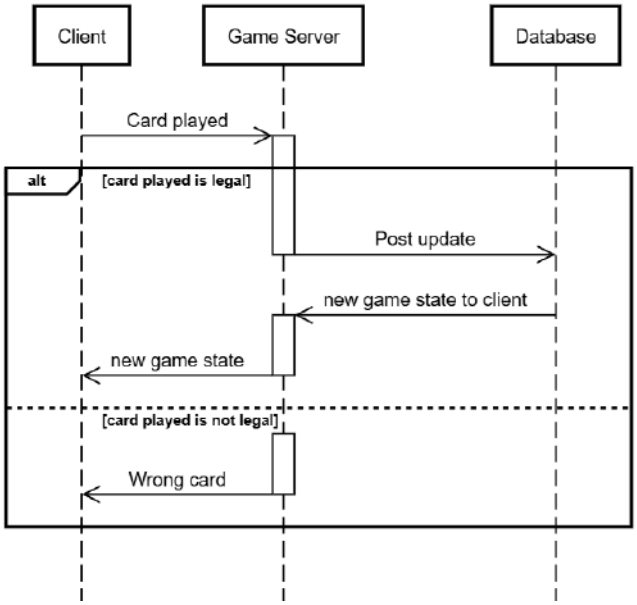


Figure 4: Interaction diagram for the 'Play Card' action

Another example of indirect interaction between microservices is in Figure 5, where the matchmaking microservice creates a new match and updates the database. Here the update is still registered by the database and then passed to the Game Server that sends it to the clients.

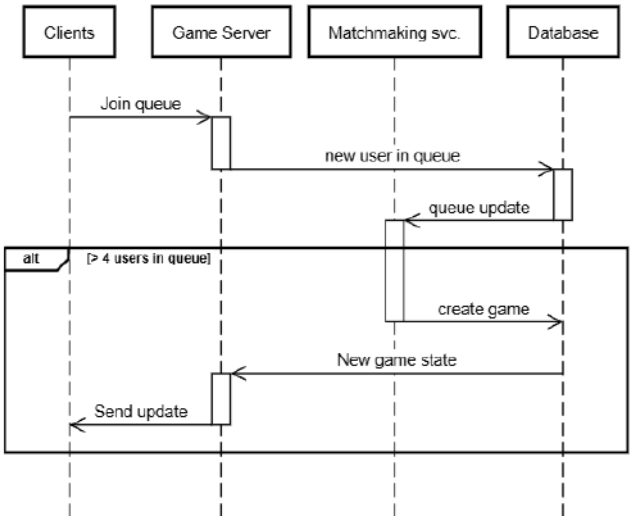


Figure 5: Interaction diagram for the 'Matchmaking' action

The same happens in the case of a timeout. Figure 6 explains how, when a user quits a game, the Game Server notifies the database, which in turn notifies the Timeout Watcher

that starts a timer and eventually forfeits the game if the user does not reconnect in time.

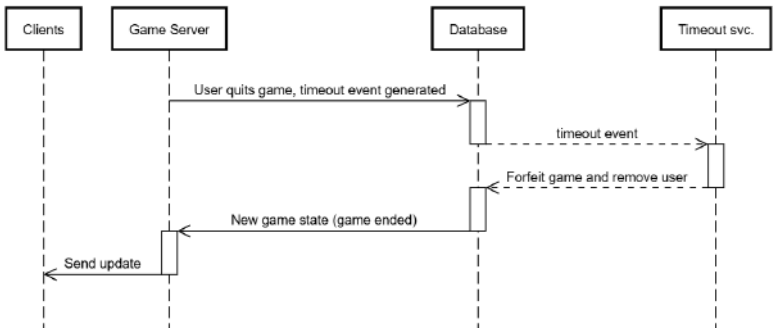


Figure 6: Interaction diagram for the 'Timeout' action

3.5 Behaviour

This section defines the dynamic behaviour and state characteristics of the core architectural components, classifying them into stateful and stateless entities, and outlining the state update lifecycle.

Component Classification

To maximize horizontal scalability and fault tolerance, the architecture enforces a strict separation between processing logic and data persistence:

- **Stateless Components:** The application layer services, GameService, MatchmakingHub, TimeoutWatcher and ConnectionServer, do not retain internal state memory across interactions. They treat each incoming network message or system event as an independent operation, executing logic based entirely on the payload and the authoritative data fetched dynamically from the storage layer.
- **Stateful Components:** The storage layer (abstracted via the `StorageService`) serves as the sole stateful entity and single source of truth in the system. It permanently holds the authoritative state of active matchmaking queues, ongoing matches, and user credentials.

State Mutation Lifecycle

System state transitions are strictly event-driven and operate under a request-validation-persistence cycle, avoiding polling.

- **Client-Driven Actions:** A client application cannot directly mutate the system state. When a player executes an action (e.g. playing a card or choosing the Trump suit), it transmits a command to the `ConnectionServer`, which dispatches it to the `GameService`. The `GameService` fetches the current game state from storage,

validates the action against the domain rules, computes the new state, and persists the updated record back to the database. A watcher of the game state will notify the update back to the clients in the lobby waiting to display it.

- **System-Driven Actions:** State mutations are also triggered automatically by internal sub-systems based on lifecycle events. The `MatchmakingHub` receives queue state updates every time a user joins the queue and atomically initialises a new `Game` context in storage as soon as four players are available. Similarly, the `TimeoutWatcher` passively monitors active connection leases; upon receiving an expiration, it forces a state transition to a forfeit status through the `GameService`, updating the persistence layer accordingly.

3.6 Data and Consistency Issues

The system relies on a distributed storage layer to maintain strict consistency across the microservices.

Stored Data and Storage Persistency

The system requires the persistence of four distinct logical data categories to satisfy operational and state requirements:

- **Authentication**: Contains registered user credentials (usernames and cryptographic password hashes). This data requires permanent persistence.
- **Matchmaking Queue**: Tracks the ordered list of active players waiting for a match. This data is highly dynamic and shared across matchmaking operations.
- **Player Session State**: Maps active players to their current game context and connection status to handle unexpected disconnections.
- **Game State**: Comprises the full structural state of ongoing matches, including player hands, trick history, points, and turn progression indicators.

To ensure loose coupling and horizontal scalability of the application services, all data is stored externally using a **Key-Value storage model**. This paradigm is optimal for this architecture because the system manages distinct, easily serialisable entities (e.g. mapping a unique Game ID to a serialised game state) and requires high-throughput, low-latency read/write operations to maintain real-time responsiveness during gameplay.

Database Interactions and Concurrency

Direct access to the database is strictly limited to the backend microservices. The interactions are structured as follows:

- **Connection Server**: Executes read and write queries during user authentication (registration and login) to verify and persist credentials. Additionally, upon match

initialisation, it instantiates watcher queries on specific game identifiers to intercept state changes and immediately stream them to the connected clients.

- **Matchmaking Hub**: Monitors the matchmaking queue registry for new player entries. When the total number of independent user records reaches the required threshold of four, it atomically removes these players from the queue and triggers the match creation sequence.
- **Game Service**: When a game is created, it queries the database to provide a unique match identifier atomically. During active gameplay, every time a player submits an action, it fetches the current game state for rule validation and saves the updated payload in the storage.
- **Timeout Watcher**: Maintains a passive, event-driven observation query on active user connection leases. If a lease expires, it intercepts the notification event and requests a write query to mutate the target game state to a forfeit status.

The distributed system architecture exhibits distinct concurrent access patterns across the shared key-value storage layer:

- **Concurrent Reads**: Read operations are executed in parallel across distinct keys. Multiple microservice instances can simultaneously retrieve independent data entities, such as separate game states or user records, without mutual blocking, maximising system throughput.
- **Event-Driven Observation**: Watcher components maintain real-time state visibility by subscribing to asynchronous event channels dispatched directly by the key-value database. Instead of polling, these components listen to dedicated streams to intercept data transitions immediately upon mutation.
- **Concurrent Writes and Atomicity**: Write operations involving match identifier generation and user session modifications are executed via atomic transactions. This transactional design guarantees isolation during the read-modify-write cycle, ensuring that no intervening modification occurs between the initial fetch of a value and its subsequent update, thereby preventing race conditions and lost updates. Game state modifications are managed through an optimistic concurrency control (OCC) approach. A read operation retrieves the state payload along with an associated revision identifier. After applying the game logic, the updated version is accepted by the storage layer only if the submitted revision identifier matches the current revision saved. If a revision conflict occurs, the operation is rejected, preserving integrity.

3.7 Fault-Tolerance

The storage layer must manage data replication across instances to ensure state consistency and availability using a Raft algorithm. This replication strategy guarantees that all nodes maintain a consistent state, even in the event of node failures, by electing a

leader to manage write operations and ensuring that changes are replicated to follower nodes before being committed.

Furthermore, mechanisms for heart-beating and graceful shutdown must be implemented to ensure liveness and to avoid data corruption in case of unexpected shutdowns or panics.

Incase of a node failure, the system must automatically detect the failure and reroute traffic to healthy nodes to maintain uninterrupted service. The stateless design of the microservices allows for seamless failover, as any instance can handle incoming requests without dependency on local state.

Nowadays, different technologies provide this kind of features out of the box, for both distributed storage and microservices orchestration. So the implementation of this features is mostly a problem of configuration and integration with the requirements of the system and will be discussed in the Section 4 about implementation.

3.8 Availability

The system must implement mechanisms to ensure high availability and minimize downtime. To do so, the architecture make use of replication, load balancing, and scaling of the microservices. This include strategies to handle node failures and network issues such as network partitioning. The statelessness of the microservices and the use of a distributed database contribute to this goal by allowing a user to connect and recover its state from another node in case of a failure.

Similarly as Section 3.7, this feature is implementable in different ways.

3.9 Security

The system design incorporates security measures focused on identity verification, game state isolation, and data protectionn.

- **Authentication**: The system implements a password-based authentication mechanism. Users must register an account and authenticate to access the matchmaking queue and participate in games. This mechanism verifies player identity, ensures accountability for game actions, and prevents unauthorized access to the application's core logic. The authentication process is fully decoupled from the game domain model and managed independently by the connection and storage services.

- **Authorization and Access Control**: All authenticated users share the same standard player privileges, eliminating the need for complex role-based access control. The `ConnectionServer` validates every incoming command to ensure it originates from the `Current Player`, rejecting out-of-turn actions or unauthorized state mutation attempts.

- **Cryptographic Schemas**:
  - **Data in transit**: All client-server communications are encrypted using secure transport protocols, ensuring that all data packets, including credentials

and real-time game updates, are protected against man-in-the-middle attacks before entering the internal network.

- **Data stored**: The system prohibits the storage of plaintext passwords. User credentials are secured using cryptographic hashing algorithms combined with unique salts before being persisted in the distributed database.

4 Implementation

This section details the technology-dependent choices made to realize the architectural design outlined in Section 3.

Network protocols and data representation

For network communications it has been chosen to use WebSockets for the persistent connection needed to push update to the clients and support for secure connections through TLS.

The data representation for the messages exchanged between the clients and the server is JSON, as it is easy to handle both in the front-end and in the back-end given that Go provides built-in support for JSON encoding and decoding of data structures.

The WebSocket connections are served throught a HTTP server that also serves the static files for the frontend. The server uses the Gin framework to handle the HTTP requests togheter with the Gorilla WebSocket library.

The web server is implemented in the `main.go` file, where the routes are defined and the connections are opened and upgraded to WebSockets. The connection management is handled by the `networking` package which defines the message types and the logic for sending and receiving messages.

The TLS protocol is not implemented in the application's logic but is instead handled by the cluster's ingress controller that terminates the TLS connection at the edge of the cluster and forwards HTTP traffic to the web server. This makes both the implementation and the certificate's management easier.

Database and authentication

Consistent with the design requirement for a distributed key-value store (Section 3.6), **Etcd** is utilized as the distributed key-value store. It implements the Raft consensus algorithm to guarantee Consistency and Partition tolerance (CP), which simplifies the deployment and synchronization of a multi-node database cluster. Etcd natively supports atomic transactions and versioning for safe concurrency control, leases for tracking active user session timeouts, and watchers to trigger asynchronous state updates across the microservices.

Authentication is implemented via standard password verification. User credentials are submitted via a secure TLS channel. The backend hashes the received password with a unique cryptographic salt before comparing it to or storing it in the Etcd database.

4.1 Technological details

React

The client's web interface is implemented in React, a JavaScript framework for designing fron-end applications that meets the requirement of real-time updates and state management. This library allows to declaratively define UI views and let the framework handle the updates when the state changes, which is perfect for the stateless nature of the client and the updates pushed by the server.

Go

The microservices are developed in Go. This language is highly effective for backend development due to its native and lightweight concurrency model. By utilizing goroutines and channels, the system efficiently manages numerous concurrent tasks, such as handling persistent WebSocket connections. Furthermore, Go provides an official and robust client library that simplifies integration with Etcd.

Kubernetes

Kubernetes fits perfectly the needs for the system's infrastructure specified in the Section 3.2. It provides an API to managing the cluster and its components and features such as load balancing, automatic scaling and recovery. Resources are defined Infrastructure-as-Code style in YAML files, which is is beneficial for versioning.

The infrastructure lightly changes with respect to the design and new components are introduced:

- Ingress: a component that redirects incoming traffic to the services and acts as TLS terminator for the connections.
- Service: a component that exposes multiple instances of a microservice as a single endpoint
- Horizontal Pod Autoscaler: automatically scales the services depending on resource usage

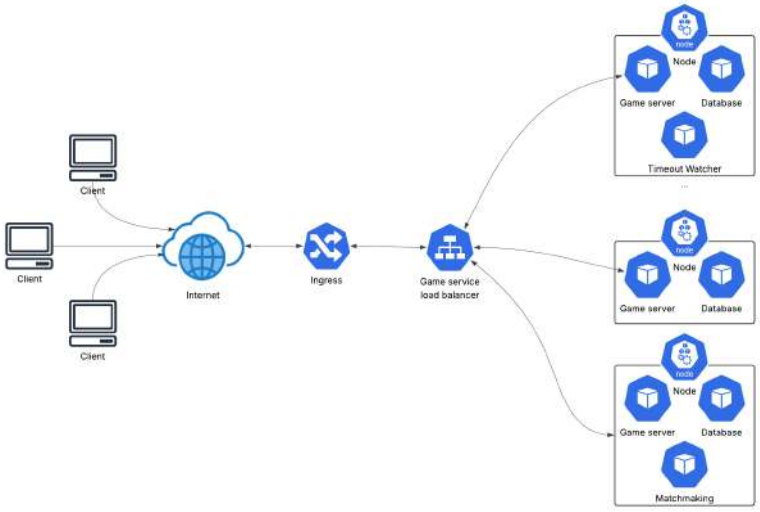


Figure 7: Deployment diagram of the Kubernetes cluster

5 Validation

5.1 Automatic Testing

Unit testing

Unit tests validate the correctness of individual components, ensuring that the game rules, state management and database interactions function as expected. They are implemented using Go's built-in `testing` package alongside the `testify` assertion library.

The test suite is executed via the `make test` command from the root of the project. This command automatically provisions a temporary Etcd instance via Docker to allow integration testing of the storage layer, runs the tests, and destroys the container upon completion.

The tests cover the following core areas:

- **Model and Game Logic:** Tests verify the strict implementation of Marafone rules, indicated by the Requirement F4, F6. This includes card ranking and power, trick winner evaluation, point calculations, and the "Marafona" bonus logic. They validate move legality, such as enforcing the leading suit, and ensure that the game view generation accurately hides opponent cards to prevent data leaks.

- **Etcd and Concurrency:** Tests verify the `EtcdService` methods to ensure robust distributed state management (for example game creation and modification). The tests verify atomic operations, user session management, optimistic concurrency control (by simulating revision mismatches) and the correct behaviour of asynchronous database watchers. Functional requirements satisfied: F1, F2,

- **Networking:** Tests ensure the proper parsing, typing, and construction of WebSocket message envelopes used for client-server communication.

End-to-end testing

End-to-end testing is performed throught Grafana K6, a framework that allows the automation of load, performance and stress tests. Testing behaviour, users, iterations and checks are defined in JavaScript files and can be run with the K6 CLI.

All the tests simulate parallel users, open WebSocket connections and send messages to the server, in particular:

- `newMatch_test.js` tests the user's registration, login and matchmaking process, verifying that the game is created. (F1, F2, F3)
- `match_test.js` tests the whole game flow, from registering to playing cards. The users are instructed to play a card coherent to the table's state, otherwise a random card. (F4, F7)
- `failedLogin_test.js` simulates users trying to login without registering first. This ensures that the systems correctly handles unauthorized access.
- `wrongMessages_test.js` simulates the users sending wrong messages without being logged in to the server such as playing a card or choosing a trump. This check thats the server correctly verifies the identity of the connection before processing a message.

Tests are performed in the production environment (the Kubernetes cluster), so to run them the user needs to:

1. install K6 from the official documentation
2. follow the instructions at the Section 6 to deploy the system

Finally, the aforementioned tests can be run with the following command:

```
k6 run test_file.js
```

For instance, running the `failedLogin_test.js` will output the test parameters and results as shown in the Figure 8.

Figure 8: Failed login test results

5.2 Acceptance test

Some tests were performed manually to rapidly check the behaviour of the application during the development phase, as it was faster than writing automated tests and the features were intuitively simple to test. The automated end-to-end tests mentioned in the Section 5.1 were instead written to verify the final infrastructure of the system.

To perform these tests, the application was containerized and deployed on a Docker container and accessed through the browser.

The features tested include:

- connection and disconnection: simply check if a page is returned when connecting to the server.
- messages exchanged: check if the format of the messages sent and received by the

client matches the ones on the server.

- user registration and login: register and log in with a new user, try to check in with an unregistered user.
- matchmaking and game flow: start a game, play correct and wrong cards to test if the logic missed any corner case. Requirements checked: F6, F7, F9
- frontend behaviour: check the correct update of the game's upon the receiving of messages and the shape and size of UI's elements. Requirement checked: F5
- timeout watcher: start a game, make a player disconnect, wait for two minutes and then check if every player sees an endgame screen. Requirement checked: F8

6 Deployment

6.1 Image building

Container images for the game server and the matchmaking service are built from source code by a GitHub Action Workflow; the same action releases the images to the project's repository.

The images are then publicly available and can be pulled directly from GHCR to be deployed on Kubernetes.

6.2 Manifests

Manifests in the `deployment/kubernetes` directory specify the deployment's characteristics such as the replica count for each pod, where the pods should be placed and which ports to open.

The `marafonet-deployment.yaml` file defines an environment variable containing the Etcd pods' addresses and a rule so that the pods from this deployment are scheduled in different nodes to ensure availability.

The `marafonet-hpa.yaml` creates an Horizonal Pod Autoscaler so that the replica count of the Game Server's pods can change dinamically according to processor and memory usage.

Other files define the Etcd cluster and the Ingress, a component that acts both as load balancer and TLS terminator.

6.3 Requirements

To deploy the application the following software must be installed on the target machine:

- Docker (depending on the operating system, it may be necessary to install Docker Desktop)
- minikube

- kubectl
- `make`: usually comes installed in Linux or Mac, or by downloading the Windows version

After successful installation the user can proceed to the next section.

6.4 Deployment instructions

First, the user needs to clone the project's repository:

```
~$ git clone https://github.com/juturbo/MarafoNet.git
```

this will pull the entire codebase and the makefiles to start both production and testing environments.

Then, the user can position at the root of the project and run the `deploy` target:

```
$ make deploy
```

This command will take some time to pull and start the node images, initialize the minikube ingress add-on, create the TLS certificates in `deployment/kubernetes/certs` and deploy the Kubernetes resources.

All the Kubernetes resources are grouped inside a `kustomize.yaml` file that is used to deploy the system throguht `kubectl apply`.

The command at Listing 6.4, in the end, should output the following:

```
kubectl apply -k deployment/kubernetes/
service/etcd-client created
service/etcd-headless created
service/marafonet created
deployment.apps/marafonet created
deployment.apps/matchmaking created
deployment.apps/timeoutwatcher created
statefulset.apps/etcd configured
horizontalpodautoscaler.autoscaling/marafonet-hpa created
ingress.networking.k8s.io/marafonet-ingress created
kubectl get pods
NAME                              READY   STATUS    RESTARTS      AGE
etcd-0                            1/1     Running   1 (14m ago)   46h
etcd-1                            1/1     Running   1 (13m ago)   46h
etcd-2                            1/1     Running   1 (15m ago)   46h
marafonet-56dff47d9d-4pxl5        1/1     Running   1 (14m ago)   46h
marafonet-56dff47d9d-g7j9m        1/1     Running   1 (15m ago)   46h
marafonet-56dff47d9d-tf65n        1/1     Running   1 (13m ago)   46h
matchmaking-846f477695-j7wjn      1/1     Running   1 (14m ago)   46h
timeoutwatcher-575b5b4d58-zpcr7   1/1     Running   1 (13m ago)   46h
kubectl port-forward -n ingress-nginx svc/ingress-nginx-controller 8080:443
Forwarding from 127.0.0.1:8080 -> 443
Forwarding from [::1]:8080 -> 443
```

Figure 9: Deployment command output

From the final lines of the command's output the user can check if the pods are deployed correctly (some may be still in `pending` or `container creation` states) and can see that the tunnel to the ingress has been opened.

Opening the tunnel is necessary in this instance due to the use of `minikube` to virtualize the Kubernetes cluser but is not done in real-world production clusters.

7 User Guide

After following the instructions at the Section 6 the user can acess the application at `https://localhost:8080`.

7.1 Logging in

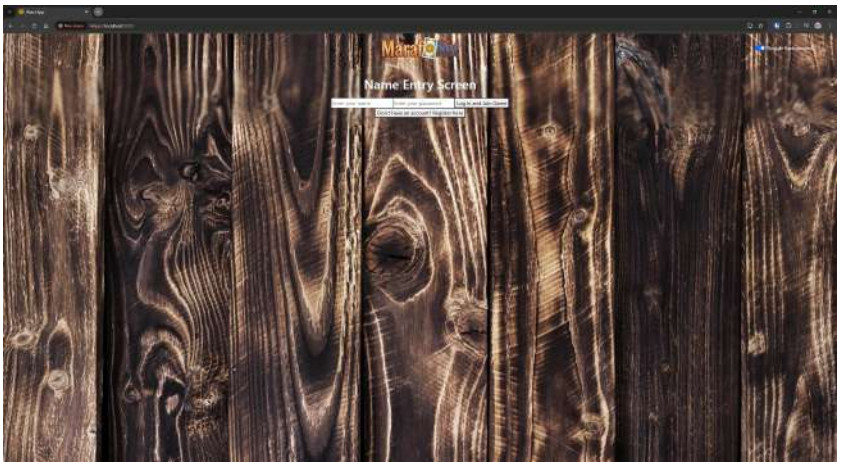
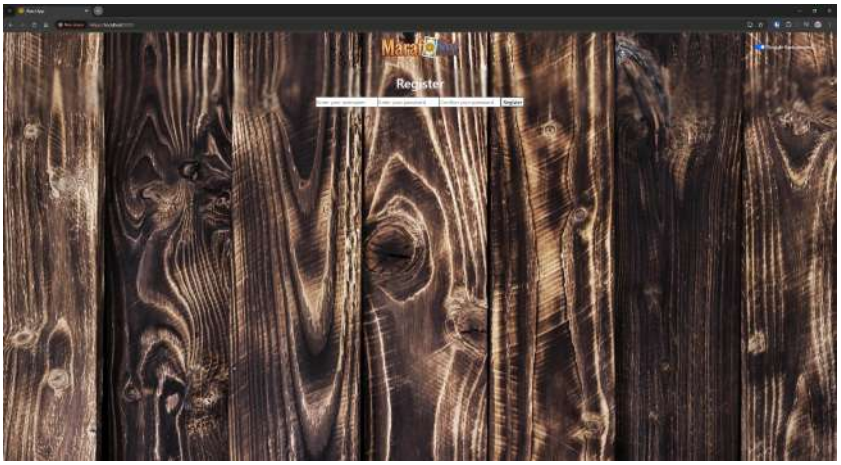


Figure 10: Login page, the initial page of the application

The user will be presented with a login page (as in Figure 10), before logging in the user needs to register an account by pressing the '`Don't have an account? Register`' button and filling the form at Figure 11a with a username and password. After successful registration, the user will be redirected to the login page where they can use the credentials to log in.



(a) Registration page

(b) Lobby page

Figure 11: The pages of the application

After logging in, the user will be automatically placed in the matchmaking queue and will be redirected to the lobby page, as at Figure 11b, where they will be waiting for other players to join the queue.

Once the required number of players is reached, the game will start and the user will be redirected to the game page where they can play the game.

UI elements

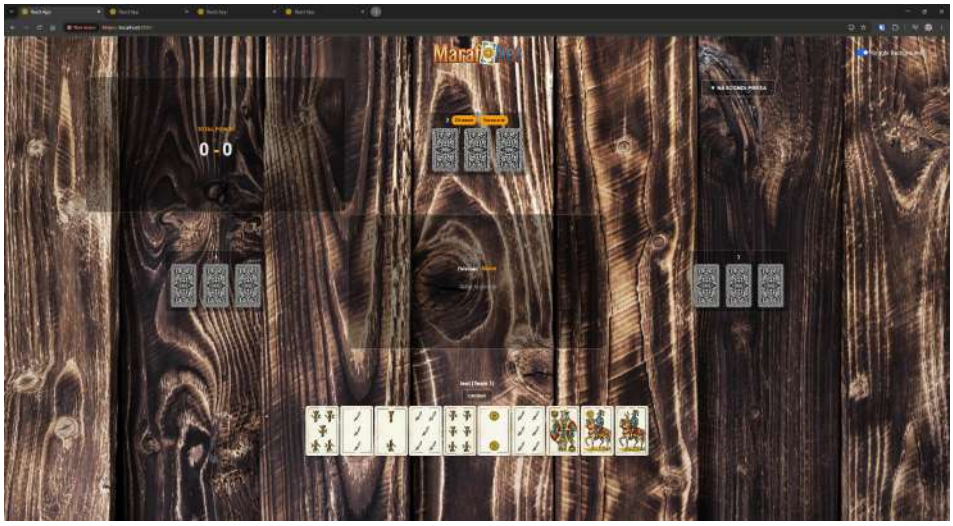


Figure 12: Game page

As shown in Figure 12, the UI presents the following elements:

- at the top left corner, the game's points, they are updated at the end of each match
- at the bottom center, the user's hand
- at top, left and right, are placed the other players' cards. The player on the top is the user's teammate
- a yellow circle with 'Di mano' written on it indicates the player that has choosen the trump suit for the current match
- a yellow circle with 'Tocca a te' written on it indicates the player that has to play a card
- at the top right corner, a container with the latest trick's cards togheter with the name of the player's who played it

7.2 Playing the game

At the beginning of each hand, the player that has the 'Di mano' circle can choose the trump suit for that match, the trump choosen will beat any card of the other suits, even if they are of higher rank. A pop-up as shown in the Figure 13 will appear to the player to let them choose the trump suit. If the player has Ace, Two and Three of the suit, chooses that suit and plays the Ace as the first card, its team is awarded with three bonus points.

Figure 13: Trump suit selection pop-up

The game is played in couples, one against the other, teammates facing each other. The game is played in a series of matches, which is composed of 10 tricks. In every trick each player plays one card. In a trick, the first player to play a card can choose whatever card they want, while the other players must play, if they have it, a card with the same suit as the first card played; otherwise they can play another card.

The game ends once a team reaches 41 or more points, wins the team that has the most points. In case of draw another match may be played. If a player disconnects for more than 2 minutes its team will automatically lose the entire game. In either scenarios, a pop-up with the winning team and the score will appear, as shown in Figure 14.

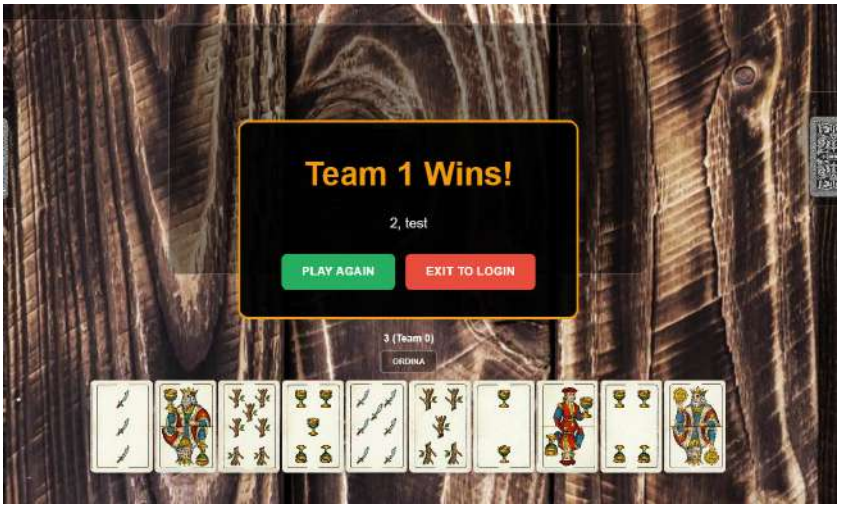


Figure 14: The end game screen

8 Self-evaluation

Federico Bagattoni

Considering that this is one of the first *big* and complete projects in which we developed from backend to frontend with a lot of different technologies and a serious amount of work, I'm really satisfied of the outcome. Although the project lacks some additional features, thinking of the integration between technologies made me appreciate the complexity in developing this kind of systems from scratch.

The roles in the project were balanced, with each member bringing experience with different technologies and ability to bring up problems that the other member had not seen coming. My role covered the frontend as well as the WebSocket connection and messages management, finally I took care of most of the Kubernetes deployment. Overall, the server side and Kubernetes was solid work. Although I am not passionate about this topic I could have done a better job at the frontend by studying recurrent patterns and best practises. Addressing and testing performances, apart from being an optimization to be mentioned in Section 9, could have been done differently by exploring resource usage and performing stress tests.

Jacopo Turchi

My role in the project focused on backend design and domain logic development. I implemented the core game logic and model, enforcing the Marafone rules through move validation, trick evaluation, and score calculation. I developed the game service responsible for processing commands and managing match state transitions. For the storage layer, I designed the interaction with the etcd distributed database to handle match creation, user credentials persistence, and session leases. I also implemented the `TimeoutWatcher` to monitor connection states and resolve disconnections via forfeits. Finally, I defined the internal communication logic and method signatures between services to orchestrate the overall game flow.

Product Strengths

- **Separation of Concerns:** The domain model is strictly encapsulated and decoupled from user identity and transport layers. Application services are completely stateless.
- **Concurrency Control:** The use of optimistic concurrency control (OCC) and atomic transactions on etcd effectively prevents race conditions during simultaneous state mutations.
- **State Integrity:** The architecture centralises the source of truth in the database. The client holds no state memory, preventing unauthorised alterations of the game flow.

Product Weaknesses

The main weaknesses of my choices concern:

- **Storage Protocol Overhead:** Using etcd introduces a performance bottleneck. Although the game is turn-based, the Raft consensus and disk I/O required for every state mutation still introduce suboptimal overhead in case of scaling.
- **Bandwidth Inefficiency:** The system currently retransmits the entire JSON state object (`GameView`) on every update. The absence of delta payloads (e.g., JSON Patch) results in redundant bandwidth usage.
- **Database-Coupled Messaging:** Using the key-value datastore to orchestrate internal notifications (e.g., triggers for the `TimeoutWatcher`) increases the database load. A dedicated Pub/Sub system or message broker would better separate the data flow from the event flow.

9 Future works

The current architecture meets core requirements, but it can be perfected by addressing the following optimisations and features.

9.1 Architectural optimizations

Perfomance testing, resource utilization and scaling

The system currently lacks in-depth studies on resource utilization, by doing these we could fine-tune the Kubernetes' CPU requests, limits and horizontal scaling behaviour to maximize concurrent connection and prevent congestion.

This process would start by first investigating the maximum concurrent connections that a single Game Server can support and then understand in depth how Kubernetes manages resources. Go provides environment variables (`GOMAXPROCS` and `GOMEMLIMIT`) to manage memory and CPU allocation so that it can regulate how much concurrent threads (and so Goroutines) it can run concurrently without being throttled by the operating system.

Datastore migration

Migrating the dynamic game storage from etcd to Redis resolves performance bottlenecks caused by etcd's consensus overhead. While etcd is ideal for static configurations, Redis is optimised for the high-frequency state mutations required by a real-time multiplayer game.

- **Write throughput and disc I/O:** etcd requires disc I/O and Raft consensus for every game move, causing latency under load. Redis operates entirely in memory (RAM), reducing write latency to sub-millisecond levels and drastically increasing system capacity.

- **Concurrency Control and Atomicity:** Instead of etcd's network-heavy Compare-And-Swap (CAS) transactions, Redis prevents race conditions naturally through its single-threaded architecture. Game state updates can be handled via atomic Lua scripts directly on the server, eliminating network round-trips.
- **Event Broadcasting and Watchers:** etcd relies on heavy gRPC watchers traversing historical logs. Redis replaces this with a lightweight, native Pub/-Sub engine. Mapping each match to a unique channel (`game:events:{gameId}`) ensures low-overhead state broadcasting.
- **Session Timeouts and Leases:** etcd manages user inactivity via continuous, network-heavy keep-alive leases. Redis simplifies this using Time-To-Live (TTL) keys. Upon client disconnection, the TTL expires and triggers a Keyspace Notification, immediately alerting the `TimeoutWatcher` to handle forfeits.

State payload reduction

The `GameService` transmits the complete `GameView` JSON object to clients on every move. The system must implement delta payloads (e.g. JSON Patch) to transmit only state mutations, reducing network bandwidth consumption.

Session management

Authentication relies on credential hashes stored in Etcd. Implementing JSON Web Tokens (JWT) will eliminate database queries for session verification, moving validation entirely to the stateless `ConnectionServer` level.

9.2 Unimplemented Features and Extensions

Custom lobbies

Implement private matchmaking by generating unique, shareable lobby codes. This requires modifying the matchmaking hub to allow players to bypass the standard public queue and join isolated game sessions.

Historical Match Database

Integrate a dedicated relational database (e.g. MongoDB) to archive completed match results and player statistics. This isolates persistent and long-term record history from the dynamic game states managed by the key-value store.

Elo-Based Matchmaking

Replace the current first-in, first-out queue with a skill-based system. Player records will store ratings to match users of similar skill levels.