MarafoNet: Marafone on the Net

Architecture and implementation of a distributed, turn-based game

Bagattoni Federico Turchi Jacopo

Alma Mater Studiorum - Università di Bologna

July 7, 2026

Table of Contents

Architectural style

The system merges the **data-centric** and **event-based** architectures. In this way the system has:

- **decoupled communications**: user interactions are asynchronous
- **transparency**: components don't know each other's existence, coordination is managed throught the dataspace. Communications and events are delivered throught the dataspace.
- **single source of truth**: the dataspace serves as single source of truths, making the system consistent

The server's architecture is divided in stateless microservices, this enables for scalability and easier lifecycle management.

Architectural choices

The stateless microservices architecture solves issues from the shared dataspace architectural style, mainly:

- **No single point of failure**: stateless microservices and distributed database nodes avoid the presence of a single point of failure
- **Scalability**: multiple stateless microservices can be scaled without data races
- **Modularity**: dividing in microservices makes debugging and lifecycle management easier

Other issues still remain:

- **Performance overhead and concurrency**: multiple services accessing the same dataspace increase overhead and slow down perfomance, making it harder to maintain a consistent state

Principles

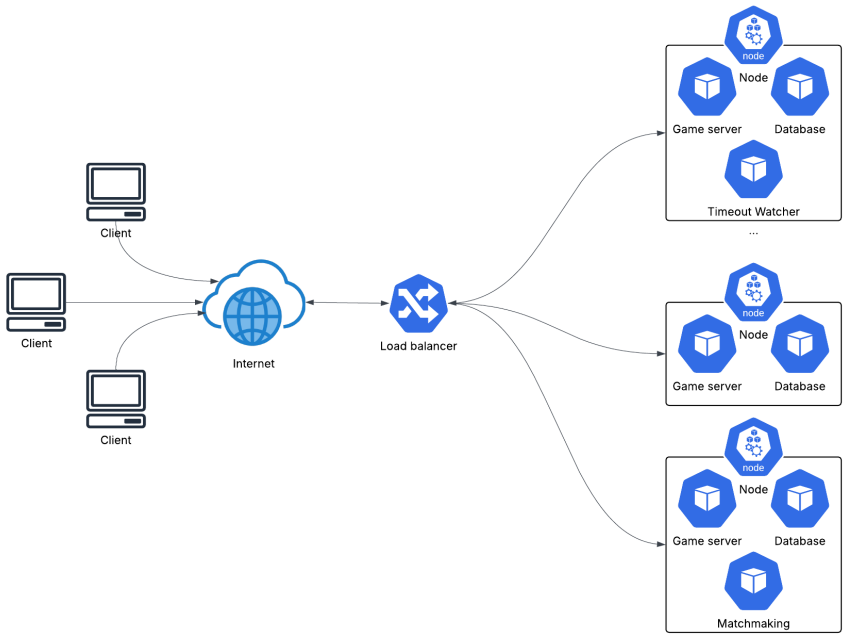
In real-world systems network partitions happen often, so choosing the P (**Partition Tolerance**) of the CAP thorem is fundamental.
Having to choose one between **Availability** and **Consistency**, we chose to have **consistent game data** at the cost of making the service unavailable in case of a fault.
Even tought it can't be totally achieved, throught *restart policies, replication and load balancing*, Availability can be increased.

System's features

Our design implements features from distributed systems:

- **Consistency and Integrity**: game's data is strongly consistent and integrity is ensured by the use of `etcd` together with `Kubernetes`
- **Availability**: even thought it's not a priority, Kubernetes employs restart policies and replication mechanisms to maximise Availability
- **Maintainability**: the stateless microservices architecture distributed on Kubernetes makes updates rollout and debugging easier

Kubernetes



Kubernetes plays a central role in the architecture. The game's stateless microservices can be easily deployed, updated and hot-swapped with K8s API. Pod allocation can also be decided.
Traffic is automatically **load-balanced** to the pods with lower resouce usage. **Scaling** rules are created to react to resource usage spikes.

Storage

Needing **strong consistency**, we needed a technology implementing a consensus algorithm such as Paxos or Raft.
These algorithms enable consistency in data by managing **consensus** between nodes, ensuring they all store the same data.
This feature is used to guarantee fault-tolerant consensus and replication.

Storage choice: etcd

Etcd implements Raft, ensuring that a cluster with $n = 2f + 1$ nodes can survive f failures.
The Go etcd library also offers notifications on data changes and leases throught Watchers, based on gRPC channels.

Remark

Etcd is not protected by Byzantine Failures: in Raft every node **assumes that all the other nodes are honest**, so a malicious behavior may result in the breaking of the vote.

Byzantine failures usually occour on components exposed to an external network, vulnerable to malicious attacks. The etcd cluster is inside the private Kubernetes network (which is trusted) and only accessed from backend service after input validation, so it's not vulernable.

Storage alternatives

Even thought geographical distribution is not a concern, etcd lacks scalability and concurrency features on large amounts of data.
Another choice could have been CockroachDB. This choice would add complexity but enable for better scalability:

- **multi-Raft**: multiple leaders can write concurrently to different **data shards**
- **linear scalability**: adding nodes actually boosts performance on data accessing
- **CDC (Change Data Capture)**: like etcd watchers, can be connected to external pub-sub brokers
- **relational database**: more complex design and access than simple key-value storage

Future works

Future works include:

- **Performance testing and scaling**: test the system's resource utilization, using Prometheus and Grafana, to tweak the scaling settings and support as much connections as possible
- **Datastore migration**: evaluate migration of the game storage from etcd to CockroachDB to scale more effectively
- **Client-side predictions** to make changes smooth and instanteneous