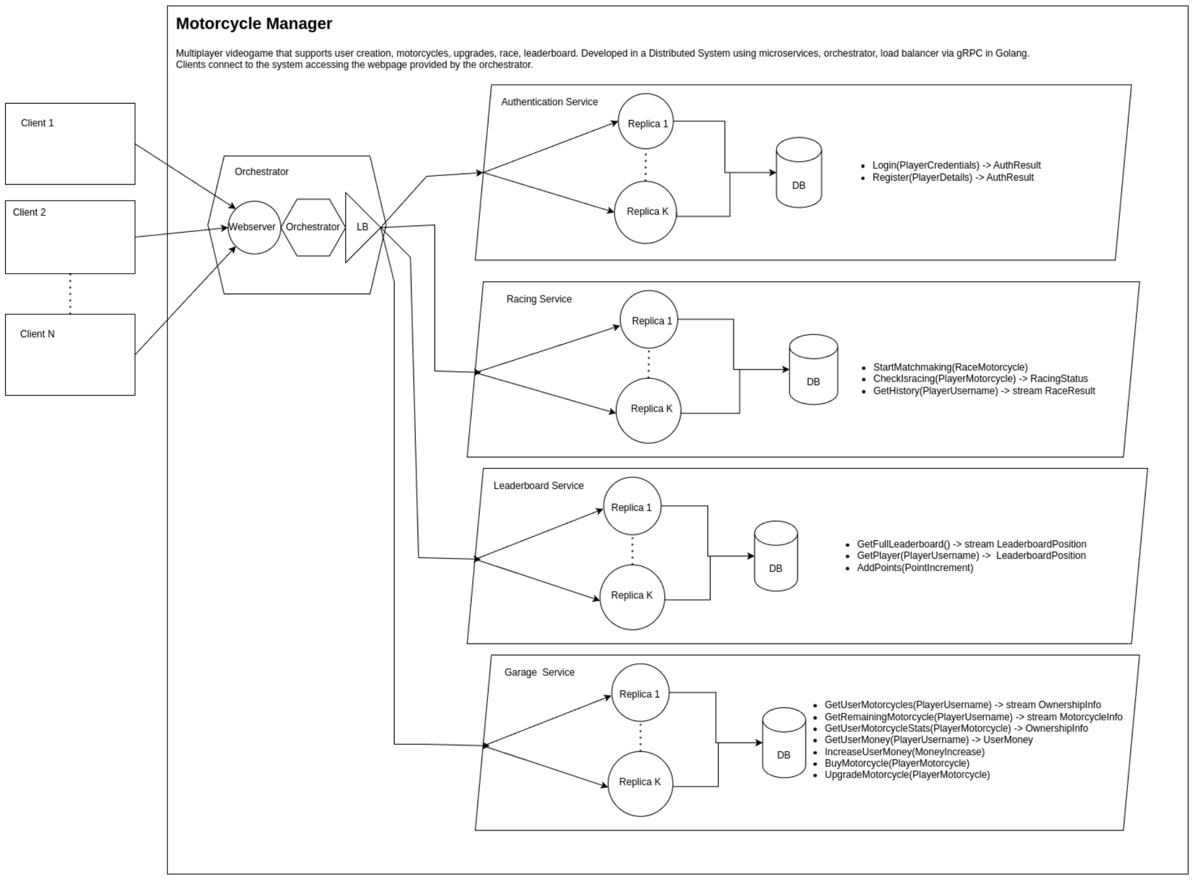
Report Motorcycle Manager

Lorenzo Bartolini lorenzo.bartolini8@edu.unifi.it


Motorcycle Manager
Multiplayer videogame that supports user creation, motorcycles, upgrades, race, leaderboard. Developed in a Distributed System using microservices, orchestrator, load balancer via gRPC in Golang.
Clients connect to the system accessing the webpage provided by the orchestrator.
Client 1
Client 2
Client N
Orchestrator
Webserver
Orchestrator
LB
Authentication Service
Replica 1
Replica K
DB
Login(PlayerCredentials) -> AuthResult
Register(PlayerDetails) -> AuthResult
Racing Service
Replica 1
Replica K
DB
StartMatchmaking(RaceMotorcycle)
CheckIsracing(PlayerMotorcycle) -> RacingStatus
GetHistory(PlayerUsername) -> stream RaceResult
Leaderboard Service
Replica 1
Replica K
DB
GetFullLeaderboard() -> stream LeaderboardPosition
GetPlayer(PlayerUsername) -> LeaderboardPosition
AddPoints(PointIncrement)
Garage Service
Replica 1
Replica K
DB
GetUserMotorcycles(PlayerUsername) -> stream OwnershipInfo
GetRemainingMotorcycle(PlayerUsername) -> stream MotorcycleInfo
GetUserMotorcycleStats(PlayerMotorcycle) -> OwnershipInfo
GetUserMoney(PlayerUsername) -> UserMoney
IncreaseUserMoney(MoneyIncrease)
BuyMotorcycle(PlayerMotorcycle)
UpgradeMotorcycle(PlayerMotorcycle)

Software Architecture

The software architecture is a Service-oriented Architecture. There are 4 different services: Auth, Racing, Leaderboard and Garage. The Auth service allows the users to register and login into the system. The Racing service allows the users to partecipate in races and stores the history of each race. The Leaderboard service is responsible for the management of the leaderboard. Lastly, the Garage service is responsible for the purchase of motorcycles, upgrades and user money.
Clients don't contact directly the services, instead, they interact with the Webserver that maps the action of the users with methods of the Orchestrator. The Orchestrator's job is to coordinate the interaction between the various services and provide a single access point. This interaction helps building an easy to use system while decoupling the clients from the services.
To further improve decoupling the number of services can vary widely. Each service registers to the Load Balancer. This component of the orchestrator stores the services and balances the load to be uniform with respect to the services instances. For simple tasks the orchestrator behaves like a Proxy offering the same interface as the service and just sending the request to the service.

System Architecture

The system architecture is very simple. It is a classic client-server architecture where the orchestrator in the middle acts both as a server (for users connecting via the webserver, and for services allowing them to register at startup and to notify the race results) and as a client (towards services when invoking their methods).

Communication Mechanisms

The communication happens in two ways depending on the nodes.
The clients communicate with the webserver using HTTP with a generic browser. The orchestrator communicates with the services inside the system using Remote Procedure Calls. This technology is optimal for this task because it is easy to standardize interfaces and helps the programmer with the serialization/deserialization of data. Moreover the data that is required is relatively simple to handle since there is no need for complex data structures like trees or graphs.

Implementation

Webserver

The Werbserver component offers a graphical interface for users to use in order to play and use the system. It is developed using Gin Web Framework. This framework offers easy to use functions to create routes, manage cookies and session.
There are some routes that are accessible without logging in and some that are private. Gin offers an easy way to manage private routes using Groups.

Load Balancer

The Load Balancer component is defined as an interface that allows the orchestrator to register and retrieve a connection of a specific service. The implementation provided is the Random Load Balancer that, as the name suggests, replies each time with a random connection to balance the load among the replicas of the same service.
The orchestrator just needs to ask the balancer for a service connection and start using it. This decoupling helps maintain easier and slimmer components in the system since the orchestrator does not know what communication paradigm each service uses. In fact services are modeled as interfaces and can use different communication techniques.
Each time the balancer finds a candidate it tests if it is still up and running by performing a Still Alive call to such service, if it is not alive it removes from the list and tries to find another one. This test is required in order to have dynamic allocation of replicas in case of high computational load.

Orchestrator

The orchestrator is the component that communicates with services to achieve a higher goal. It offers some actions to be performed inside routes, but from a more general view of the system, it can be used as an interface for the whole system. Since it is decoupled from the webserver, the orchestrator can be used to perform actions on the system by other components/interfaces.

For example, it would be easy to refactor the interface for users to create a Desktop Application that uses a REST API. In this case, the only thing to change is to create the interface because the Orchestrator is ready to perform all actions independently of the origin from which the requests come.
To enforce further decoupling, all the services (auth, …) are handled as interfaces inside the orchestrator. This is a great way in case of a change of the communication paradigm from gRPC to something else. The orchestrator uses services via their interfaces without bothering with the actual implementation and communication schema. Load balancer helps the orchestrator removing from it the management of different replicas and load balance.

Services and Databases

All the services are organized with the same structure. The Server object implements the gRPC procedures and contains a reference to the Database interface. This interface refers to the actions that can be made inside the database. The implementation of such interface is for a MySQL server but can be easily implemented for various types of databases.
The server object is responsible for calling the right methods on the database and preparing the message for the gRPC response.
The racing service has a peculiarity that other services do not have. This is the fact that when starting the matchmaking, if that motorcycle is the last one needed in order to start the race, it also performs the end of the race and replies to the orchestrator with the results for each participant.
Each service has its own database that is shared among replicas. Each service is stateless allowing for easy scaling adding more replicas without affecting the state of the service. Also having a specific database for each service is a good decision in term of not having a single point of failure for the data in the system and helps with performance since not every interaction is handled by the same database.
All the services implement the StillAlive service that is used by the Load Balancer to detect failures among replicas.

Tests

The tests were performed on individual services, more specifically on the interaction with the database. These tests can be executed individually and ensure the correct execution of services.

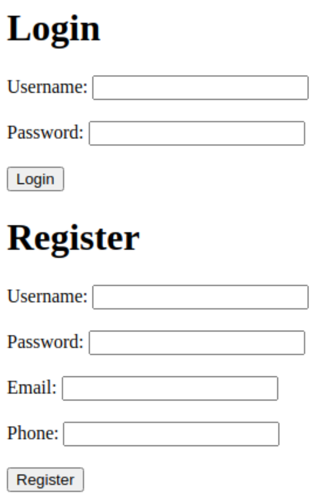
Deployment

The deployment is managed with Docker, Docker compose and Makefile.
There are two environments: one for running the system and one for testing. Thanks to Docker it is possible to separate completely these environments. In fact they live in different networks, each database has its own volume and also containers are built differently with differents targets.
The steps for running the system are: make build, that compiles the protobuf files pulling a specific image form the Hub and building the containers for each service, and then make up/upd for running the system attaching to containers or in detached mode. Using make stop and make down it is possible to stop containers and remove volumes. This is useful to reset databases since they are built using a startup script that is executed only when the volume is freshly created.
To run tests the steps are similar: make build_test (with the only difference that down_test is called in order to reset the databases) and make test, to run tests.
After running everything, calling make clean, cleans the system from every trace removing containers, volumes, networks and the compiler for protobuf.
Inside the .env file it is possible to change environment variables such as the number of replicas for each service.
Each service has its own dockerfile that is composed with three stages: build stage using golang-alpine, test stage that still uses golang alpine to run tests and the run stage that uses raw alpine copying the executable from the build stage and executing it.
The webserver is deployed by default on port 5000 accessible on localhost:5000. In case of problems with conflicting ports it is possible to change ports in the .env file.

Use Cases

Login and Registration

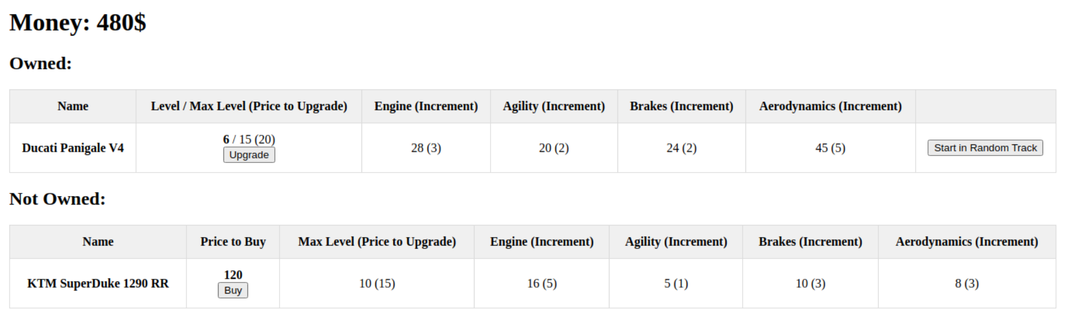
From the homepage it is possible to Login using username and password or to Register providing also the email and phone number.
Once logged in the user can logout clicking on the Logout button on the homepage.



Leaderboard

This action can be done also by non register users from the homepage clicking on the Leaderboard link.

Garage



Money: 480$

Owned:

Name	Level / Max Level (Price to Upgrade)	Engine (Increment)	Agility (Increment)	Brakes (Increment)	Aerodynamics (Increment)	
Ducati Panigale V4	**6** / 15 (20) Upgrade	28 (3)	20 (2)	24 (2)	45 (5)	Start in Random Track

Not Owned:

Name	Price to Buy	Max Level (Price to Upgrade)	Engine (Increment)	Agility (Increment)	Brakes (Increment)	Aerodynamics (Increment)
KTM SuperDuke 1290 RR	**120** Buy	10 (15)	16 (5)	5 (1)	10 (3)	8 (3)

Inside the garage page it is possible to view the user money and all motorcycles (owned and still to purchase). The table shows the name of the motorcycle, the price to buy or upgrade and the points in each category. Clicking on Upgrade it is possible to upgrade the motorcycle up to the max level paying the price specified inside the brackets. Similarly it is possible to buy motorcycles. On owned motorcycles it is possible to start racing in a random track, clicking on it it shows in which track it is racing and once the race ends, the user can see the result in the history page.
Once the race ends, each player is awarded some money and points. The number of money and points can be modified in the .env file and it is computed as a simple line connecting the values specified in the file (money for first position and money for last position, same for points).

History

This page shows the list of races in which the user took part. It is visualized the time at which the race ended, the name of the motorcycle, the trackname and the position.