ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

UNIVERSITÉ DU QUÉBEC

RAPPORT TECHNIQUE

PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

DANS LE CADRE D'UN PROJET SPÉCIAL

DU BACCALAURÉAT EN GÉNIE LOGICIEL

SYSTÈMES HAUTEMENT SCALABLES EN RUST

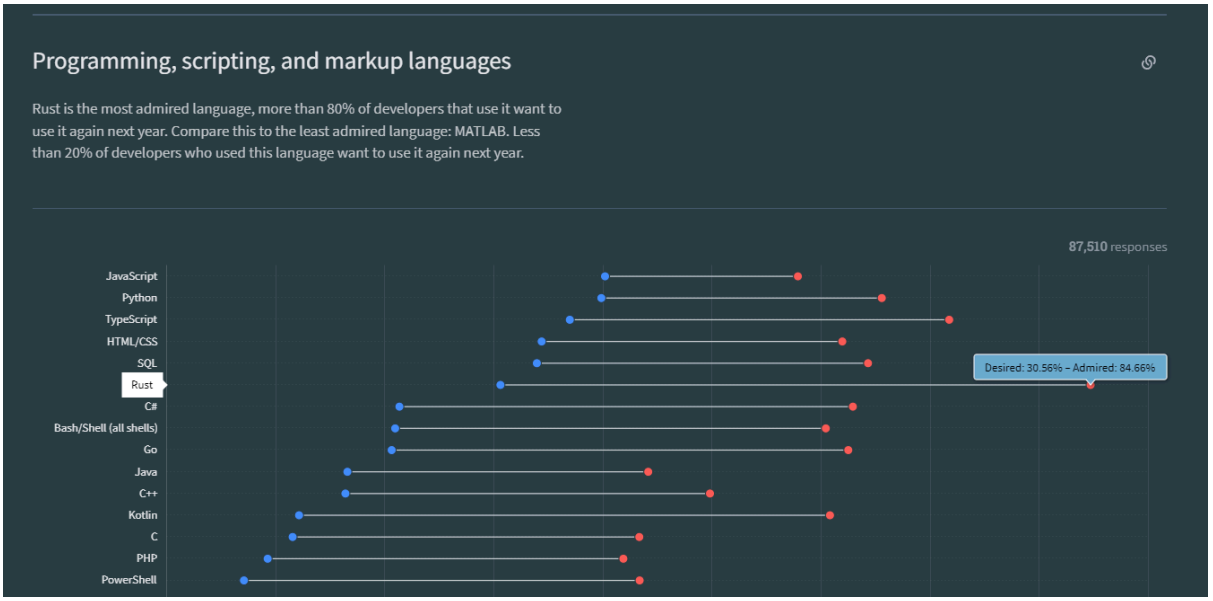
PAR

ROBYN GIRARDEAU, GIRR25029403

MONTRÉAL, LE 12 AOÛT 2024

Introduction

Rust est un langage système émergent qui gagne beaucoup en popularité d'année en année. Il se veut performant, fiable et sécure. C'est un langage de bas niveau qui a la particularité d'avoir une mémoire fiable sans avoir de collecteur de vidanges (garbage collector).
Il est le langage le plus admiré sur StackOverflow toutes les années depuis 2016[1] selon leur sondage annuel. Toutefois, il est rarement utilisé en entreprise et offre donc peu d'opportunités d'emploi, surtout en raison de sa jeunesse.



2

De nos jours, les jeux vidéos sont majoritairement programmés en C++ ou C# via l'intermédiaire d'engins comme Unreal et Unity. La performance du C++ et son écosystème mature de plusieurs décennies font qu'il reste le plus pertinent pour la programmation de rendu graphique. Avec l'arrivée de Rust, on peut se demander si celui-ci peut remplacer le C++ dans les applications où la performance est critique ou sinon quelle autre place il peut prendre.

Présentation du projet

Ce projet consiste à étudier les opportunités de construire des systèmes logiciels hautement scalables et fiables adaptant le modèle architectural traditionnel à la plateforme Rust. À la fin de ce cours, les étudiants seront capables d'architecturer des systèmes complexes en Rust.

Pour apprendre et explorer Rust, le projet cible est de développer un serveur de « matchmaking » scalable. Le matchmaking met en relation des joueurs pour créer une partie de jeu en ligne.

---

[1] https://www.reddit.com/r/rust/comments/149cu1k/2023_stack_overflow_survey_rust_is_the_most/

[2] https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages

Un des objectifs est d'offrir ce service à n'importe quel serveur de jeu. C'est donc une bonne idée de définir les messages et objets en protobuf, de façon à ce que la librairie et la communication soient compatibles avec des microservices de n'importe quel langage. Le projet implique aussi de tester l'environnement avec des joueurs, donc de la communication TCP. Une base de données sera nécessaire pour enregistrer les joueurs dans une file d'attente et créer des parties. C'est également important de tester l'expérience de déploiement. De plus, une API web serait pratique pour tester, monitorer et contrôler les applications. On regardera donc les options de librairies TCP, web, async, protobuf, bases de données, sérialisation, etc.

Le cycle de matchmaking commence par un joueur créant un lobby. Il peut alors inviter d'autres joueurs dans le lobby pour jouer en équipe. Ensuite il place le lobby dans une queue (différents types de parties, ex: normale, classée) et active la file d'attente pour trouver un match avec un autre lobby dans la même queue. Une fois le match trouvé, les lobbys sont supprimés, une partie est créée et les joueurs commencent à jouer.

Le code du projet est disponible à https://github.com/Souchy/RustProject

Chapitre 1: Rust

Cargo

Cargo, le système de gestion de dépendances et de compilation en Rust est un de ses meilleurs atouts. C'est de loin un des meilleurs systèmes tout langage confondu.

1. Il fonctionne de la même manière sur toutes les machines sans installation spéciale requise autre que Rust.
2. Il est très facile d'ajouter des dépendances avec des versions et des fonctionnalités spécifiques (« features »), dans le même format que NPM. Le téléchargement et la compilation des librairies se font automatiquement avec la même configuration sur toutes les plateformes.
3. Il permet d'ajouter des commandes.
4. Ses commandes sont très faciles (cargo build, cargo run, cargo test, etc.).

Le désavantage primaire de Cargo est le temps de compilation car il doit compiler les sources de toutes les librairies importées, mais du travail est en cours dans ce domaine pour l'optimiser et améliorer le cache.

En comparaison, NPM pour Javascript et Nuget pour C# sont les plus près. L'ajout de librairies sur NPM se fait comme avec Cargo. Nuget y ressemble également, mais offre en plus un GUI intégré à Visual Studio pour installer et gérer des dépendances.

En Java, Gradle et Maven y ressemblent, mais sont légèrement plus douloureux de par leur syntaxe. Ils demandent un peu plus de mise en place, puis fonctionnent assez bien par la suite, mais chacun impose une structure de fichiers particulière.

C++ avec sa multitude de compilateurs qui fonctionnent chacun différemment selon la plateforme est le pire. De plus, sans rentrer dans les détails, la gestion de dépendances avec CMake, Scons, et autres amène aussi son lot de frustrations.

Borrow Checker

Le « borrow checker » est la particularité principale de Rust qui le rend unique. C'est celui qui permet d'assurer la sûreté de la mémoire et des fils d'exécution (« threads ») sans avoir de collecteur de vidanges.
En C++, le programmeur est attendu d'allouer et de désallouer la mémoire lui-même. En Java ou C#, la mémoire non utilisée est désallouée automatiquement par le collecteur de vidanges. En Rust, le borrow checker impose qu'une partie de mémoire (ex. : une variable) ne soit accessible qu'à un seul endroit à la fois et est désallouée automatiquement lorsqu'on sort du contexte la contenant. On parle alors de propriétaire, de possession et d'emprunt.
Le livre de Rust[3] fourni amplement d'exemples et d'explications pour bien comprendre son fonctionnement https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.
Par exemple :

[3] https://doc.rust-lang.org/book/title-page.html

```
let s1 = String::from("hello");
let s2 = s1;
println!("{s1}, world!");
```

Ce code résulte en l'erreur suivante :

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:15
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{s1}, world!");
  |               ^^^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of
the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

La valeur de s1 a été déplacée à l'adresse de s2. Il n'y a donc plus rien à l'adresse de s1. Donc, utiliser s1 devient invalide. Pour pallier à ce problème, on pourrait écrire :

```
let s1: String = String::from("hello");
let s2: String = s1.clone();
println!("{s1}, world!");
```

En clonant la mémoire, la valeur est la même aux deux adresses et on peut continuer d'utiliser s1. Par contre, cela vient avec un coût additionnel en computation et en mémoire.

Pointeurs

Les pointeurs intelligents existent notamment pour permettre de partager des variables à plusieurs endroits.

- Box<T> pour allouer une valeur sur le heap.
- Rc<T>, un type comptant les références qui permet plusieurs propriétaires.
- Arc<T>, identique à Rc mais pour un contexte asynchrone.
- Ref<T> et RefMut<T> accédés à travers RefCell<T>, un type qui impose les règles d'emprunt à l'exécution plutôt qu'à la compilation.

Le tableau suivant résume très bien leurs différences et inclut les façons d'avoir des variables mutables selon le contexte.

Type	Storage: where is T	Lifetime mgmt	Interior mutability: &mut T from &Foo<T>	Threads Send/Sync
T [1]	itself	owner	No	Yes
Box<T> [1]	heap	owner	No	Yes
Rc<T>	heap	refcount[2]	No; T now immutable	No
Arc<T>	heap	refcount[2]	No; T now immutable	Yes
RefCell<T>	within	owner	Yes, runtime checks	Send
Mutex<T>	within	owner	Yes, runtime locking	Yes
RwLock<T>	within	owner	Yes, runtime locking	Yes
Cell<T>	within	owner	Only move/copy	Send
UnsafeCell<T>	within	owner	Up to you, unsafe	Maybe
atomic [3]	within	owner	Only some operations	Yes
Rc<RefCell<T>>	heap	refcount[2]	Yes, runtime checks	No
Arc<Mutex<T>>	heap	refcount[2]	Yes, runtime locking	Yes
Arc<RwLock<T>>	heap	refcount[2]	Yes, runtime locking	Yes

4

Box<T> est un pointeur singulier qui sert surtout à encapsuler une variable dynamique et la mettre sur le tas. Par exemple :

```
pub async fn main() -> Result<(), Box<dyn Error + Send + Sync>> { … }
```

Ici, le trait Error peut être implémenté par n'importe quelle struct et donc avoir une grandeur variable. Comme il est impossible de retourner une valeur à grandeur variable, on doit l'encapsuler dans un pointeur comme Box pour fixer sa grandeur. Sinon, on obtient l'erreur suivante :

```
the size for values of type `(dyn StdError + 'static)` cannot be
known at compilation time
the trait `Sized` is not implemented for `(dyn StdError +
'static)` rustc(Click for full compiler diagnostic)

result.rs(527, 20): required by an implicit `Sized` bound in
`Result`
```

Pour reprendre l'exemple des strings, on pourrait le réécrire de la manière suivante:

```
let s1: Rc<String> = Rc::new(String::from("hello"));
let s2: Rc<String> = Rc::clone(&s1);
println!("{}, world!", s1);
```

À ce moment, on crée un pointeur vers la string, puis on clone uniquement le pointeur en gardant la même valeur en mémoire. Cela est beaucoup plus rapide et efficient, en particulier si on travaille avec des objets plus complexes qu'une simple string.

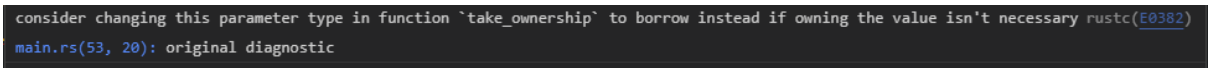
[4] https://www.chiark.greenend.org.uk/~ianmdlvl/rust-polyglot/ownership.html

Emprunter

Par défaut, une valeur passée à une fonction est déplacée en mémoire et devient inutilisable dans le contexte originel. Par exemple :

```
fn main() {
    let s1 = String::from("hello");
    take_ownership(s1);
    println!("{}", s1);
}
fn take_ownership(s: String) {
    println!("{}", s);
}
```

Ce code résulte en la même erreur de déplacement qu'au premier exemple. On peut encore cloner s1 avant de la passer à la fonction, mais c'est préférable d'éviter.
Rust nous propose plutôt ceci pour l'argument « s » :



Il est effectivement mieux de passer une référence temporaire à la fonction:

```
fn main() {
    let s1 = String::from("hello");
    borrow(&s1);
    println!("{}", s1);
}
fn borrow(s: &String) {
    println!("{}", s);
}
```

La valeur de s1 est alors déplacée à la fonction pour sa durée, puis est redonnée à s1 lorsqu'elle termine, ce qui permet de continuer de l'utiliser. C'est ce concept qui est appelé l'emprunt.

Autre exemple

Regardons un exemple simple, un écran basique contenant des rectangles à dessiner :

```
pub struct Rectangle {}
impl Rectangle {
    pub fn draw(&self) {}
}
pub struct Screen {
    pub rectangles: Vec<Rectangle>;
}
impl Screen {
    pub fn add(&mut self, rec: Rectangle) {
        self.components.push(rec);
    }
    pub fn draw(&self) {
        for rect in self.rectangles.iter() {
            rect.draw();
        }
    }
}
```

Si on veut supporter plusieurs types de formes, on doit alors utiliser un trait dynamique qui doit être encapsulé dans un pointeur :

```
pub trait Shape {
    fn draw(&self);
}
struct Rectangle {}
impl Shape for Rectangle {
   pub fn draw(&self) {}
}
pub struct Screen {
   pub components: Vec<Box<dyn Shape>>,
}
impl Screen {
   pub fn add(&mut self, sh: Box<dyn Shape>) {
      self.components.push(sh);
   }
   pub fn run(&self) {
      for component in self.components.iter() {
         component.draw();
      }
   }
}
```

Maintenant, disons qu'on veut accéder à ces formes de manière asynchrone et qu'elles sont mutables.

```
pub trait Shape : Send + Sync {
    fn draw(&self);
}
pub struct Screen {
   pub components: Vec<Arc<Mutex<dyn Shape>>>,
}
impl Screen {
   pub async fn add(&mut self, shape: &Arc<Mutex<dyn Shape>>) {
      self.components.push(Arc::clone(shape));
   }
   pub async fn run(&self) {
      for component in self.components.iter() {
         component.lock().await.draw();
      }
   }
}
```

On doit alors ajouter un mutex et le verrouiller pour assurer au compilateur la sécurité de la mémoire. On doit aussi ajouter les traits Send et Sync à Shape pour permettre de déplacer la mémoire en contexte asynchrone. Il y a quelques autres solutions à ce problème, mais ce sont des modifications forcées pour la compilation.
La syntaxe devient très lourde comparée à d'autres langages qui laissent le développeur choisir sa manière de gérer la mémoire. Évidemment cela a l'avantage d'assurer la sécurité de la mémoire, le programme ne peut pas compiler autrement. D'autres langages auraient également besoin d'ajouter des composants similaires comme des blocs synchrones ou des mutex pour au final obtenir le même résultat.

C’est globalement une énorme force de Rust que de trouver ces erreurs au moment de la compilation plutôt que de laisser le programme s’exécuter avec un comportement imprédictible. Sans cette sûreté, on peut facilement créer des fuites mémoires, obtenir des valeurs insensées dues à la concurrence, ou encore faire planter le programme entier.
On déplace donc le temps de développement d’un point à un autre et il est beaucoup plus facile de corriger les problèmes à la compilation plutôt que de déboguer l’exécution. Toutefois, cela demande beaucoup d’expérience et de compétences avec Rust pour s’adapter à son paradigme particulier et écrire du code efficacement.
Plusieurs patrons de programmation orientée objet ne sont pas recommandés en Rust. Par exemple, il est impossible d’implémenter Vec<T> sans utiliser de code unsafe. Les références circulaires (ex: LinkedList) sont découragées à cause du borrow checker qui requiert un seul propriétaire pour chaque partie de mémoire. De plus, c’est un cas qui peut causer des fuites de mémoires car il crée un cycle de références en contournant toutes les contraintes assurant la sûreté de la mémoire.

Écosystème

Éditeurs (IDE)

De ce côté, il n’y a pas encore beaucoup d’options bien développées pour Rust. L’extension rust-analyser[5] pour Visual Studio Code est probablement la plus populaire. Cette extension fonctionne plutôt bien, mais rencontre encore des bugs et des problèmes de latence. Jetbrains développe le seul IDE complet nommé RustRover[6]. Zed[7] est un nouvel éditeur de code semblable à VSCode mais spécialisé pour Rust.

Serveurs TCP

Pour Rust, Tokio[8] est le plus populaire. Il s’agit d’une librairie d’exécution asynchrone en plusieurs couches. Il faut néanmoins écrire sa propre gestion des paquets. La librairie est de très bas niveau et n’ajoute pas beaucoup de fonctionnalités pour rendre le développement plus facile. La performance est toutefois très haute et assure la sécurité mémoire.

Java et C# Netty:
Netty[9] est une librairie de communication générale organisée par pipeline. Elle est tellement populaire en Java qu’elle a également des ports comme celui en C# qui est développé par Azure[10].

Serveurs HTTP

Une API HTTP est souvent très utile pour permettre à des services de communiquer entre eux ou pour les monitorer et les contrôler. La communauté Rust développe plusieurs

[5] https://marketplace.visualstudio.com/items?itemName=rust-lang.rust-analyzer
[6] https://www.jetbrains.com/rust
[7] https://zed.dev
[8] https://tokio.rs/
[9] https://netty.io
[10] https://github.com/Azure/DotNetty

cadriciels web comme Axum[11], Actix[12], Rocket[13], Tower[14], Hyper[15], Warp[16], etc. Le travail semble très bien de ce côté-là.
Cependant, il devient plus compliqué de générer et de servir une spécification OpenApi automatiquement à partir de ses routes. Une page Swagger automatique accélère grandement la vitesse de développement et de test. C'est une fonctionnalité relativement de base dans d'autres langages, mais qui manque malheureusement dans la plupart des cadriciels Rust. Rocket est le seul à offrir une librairie adjacente pour cette fonction (Okapi[17]). Rweb est le seul cadriciel web intégrant directement la génération automatique de spécification OpenAPI, mais c'est une plus petite librairie qui n'a pas reçu de mise à jour depuis 3 ans. Oasgen[18] est nouveau de moins d'un an et est censé fonctionner avec Actix et Axum.

D'autres langages plus matures ont des librairies plus complètes et maintenues à long terme qui supportent toutes la génération de spécification. Java a Spring Boot, C# a ASP.NET, Javascript a Express, Python a Flask, etc. Bien entendu, les avantages et les désavantages restent les mêmes. Java et C# offrent une performance décente, mais ont un collecteur de mémoire et une exécution plus coûteuse. Python et Javascript sont beaucoup plus lents.

Protobuf

La communication entre plusieurs logiciels de différents langages est souvent souhaitable. Par exemple, entre un serveur de jeu en Rust et un client de jeu en C# (Godot, Unity) ou entre plusieurs microservices développés par des équipes différentes. Protobuf permet de définir un schéma pour les paquets de communication et modèles de données, puis de générer le code correspondant dans chaque langage. Les langages populaires comme C#, Java et C++ ont des librairies officielles maintenues par Google[19], mais ce n'est pas le cas de Rust.

Malheureusement il y a quelques problèmes avec les librairies offertes sur cargo.

1. Rust-protobuf
   Rust-protobuf[20] est la première librairie explorée. Elle est très complète, personnalisable et appréciable. Toutefois, elle a un problème principal qui la rend inutilisable. Elle ne supporte pas la génération de messages organisés par modules dans plusieurs dossiers. Lorsqu'on développe une application minimalement large, on veut séparer nos messages et modèles par préoccupation plutôt que tous les mettre dans le même dossier. Surtout lorsqu'il s'agit d'offrir une librairie client pour communiquer avec le service de matchmaking.

2. Tonic

---

[11] https://github.com/tokio-rs/axum
[12] https://github.com/actix/actix-web
[13] https://github.com/rwf2/Rocket
[14] https://github.com/tower-rs/tower
[15] https://crates.io/crates/hyper
[16] https://crates.io/crates/warp
[17] https://github.com/GREsau/okapi
[18] https://github.com/kurtbuilds/oasgen
[19] https://protobuf.dev/reference
[20] https://crates.io/crates/protobuf

Tonic[21] est développé par les créateurs de Hyper et est basé sur Prost. Par contre, son système de configuration ne permet pas certaines options supportées par Prost; principalement l'ajout d'attributs (traits dérivés) aux structs générées.

3. Prost
   Prost[22], initialement évité car il requiert d'installer un exécutable en plus de la librairie, supporte la majorité des fonctionnalités voulues. Il permet de séparer les messages en plusieurs modules et de dériver des traits. Le seul désavantage est qu'il ne permet pas d'utiliser, par exemple, les ObjectId de MongoDB dans les messages. Autrement dit, l'utilisation de modèles externes au projet. Cela a été contourné en utilisant seulement Redis.

L'expérience finale est plutôt positive. On utilise un fichier build.rs comme le suivant pour automatiquement générer le code associé aux protobufs lors de la compilation :

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let protos = ["src/protos/Match.proto", "src/protos/MatchResult.proto",
"src/protos/SetQueue.proto"];
    let includes = ["src/protos"];

    let mut config = prost_build::Config::new();
    config.enum_attribute(".", "#[derive(::serde::Serialize, ::serde::Deserialize,
::schemars::JsonSchema, ::rocket::FromFormField)]");
    config.message_attribute(".", "#[derive(::serde::Serialize, ::serde::Deserialize,
::schemars::JsonSchema)]");
    prost_reflect_build::Builder::new()
        .descriptor_pool("crate::DESCRIPTOR_POOL")
        .compile_protos_with_config(config, &protos, &includes)?;

    println!("Coral generated protos");
    Ok(())
}
```

Dé/sérialisation

Ici tout va bien. Serde[23] s'occupe de la dé/sérialisation en général et Bincode[24] s'occupe de la dé/sérialisation en bytes. Il suffit de dériver les traits Serialize et Deserialize de Serde pour utiliser leurs fonctions :

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, PartialEq, Debug)]
struct Entity {
    x: f32,
    y: f32,
```

[21] https://crates.io/crates/tonic
[22] https://crates.io/crates/prost
[23] https://serde.rs
[24] https://crates.io/crates/bincode/

```
}
#[derive(Serialize, Deserialize, PartialEq, Debug)]
struct World(Vec<Entity>);

fn main() {
    let world = World(vec![Entity { x: 0.0, y: 4.0 }, Entity { x: 10.0, y: 20.5 }]);
    let encoded: Vec<u8> = bincode::serialize(&world).unwrap();
    // 8 bytes for the length of the vector, 4 bytes per float.
    assert_eq!(encoded.len(), 8 + 4 * 4);
    let decoded: World = bincode::deserialize(&encoded[..]).unwrap();
    assert_eq!(world, decoded);
}
```

Bases de données

Diesel[25] est le choix le plus populaire sur Rust pour les bases de données. Toutefois, c'est un ORM pour SQL, ce qui n'est pas adapté au projet actuel. Pour un projet hautement scalable avec des données temporaire et de la communication interservices, Redis[26] fait plus de sens. Diesel aurait pu être utilisé pour garder les données permanents comme les comptes utilisateurs, mais MongoDB[27] fut exploré à la place par familiarité. Malheureusement, la librairie Mongo requiert absolument d'utiliser les ObjectId comme identifiants d'objets. Ce qui n'est pas compatible avec les librairies de génération de modèles basés sur protobuf. RedisOM[28] promet de simplifier le développement en agissant comme ORM, mais plusieurs des fonctionnalités décrites sur leur page sont absentes lors du développement (ex: support JSON). Le développement s'est donc basé sur la librairie Redis de base.

Dans d'autres langages, des ORMs sont disponibles pour toutes ces bases de données SQL, MongoDB[29] et Redis. Par exemple Redis.OM pour C#[30] et Java[31]. Même en dehors des ORM, les fonctionnalités et le support sont plus poussées et les ressources comme des tutoriels sont plus nombreuses.

Réflexion

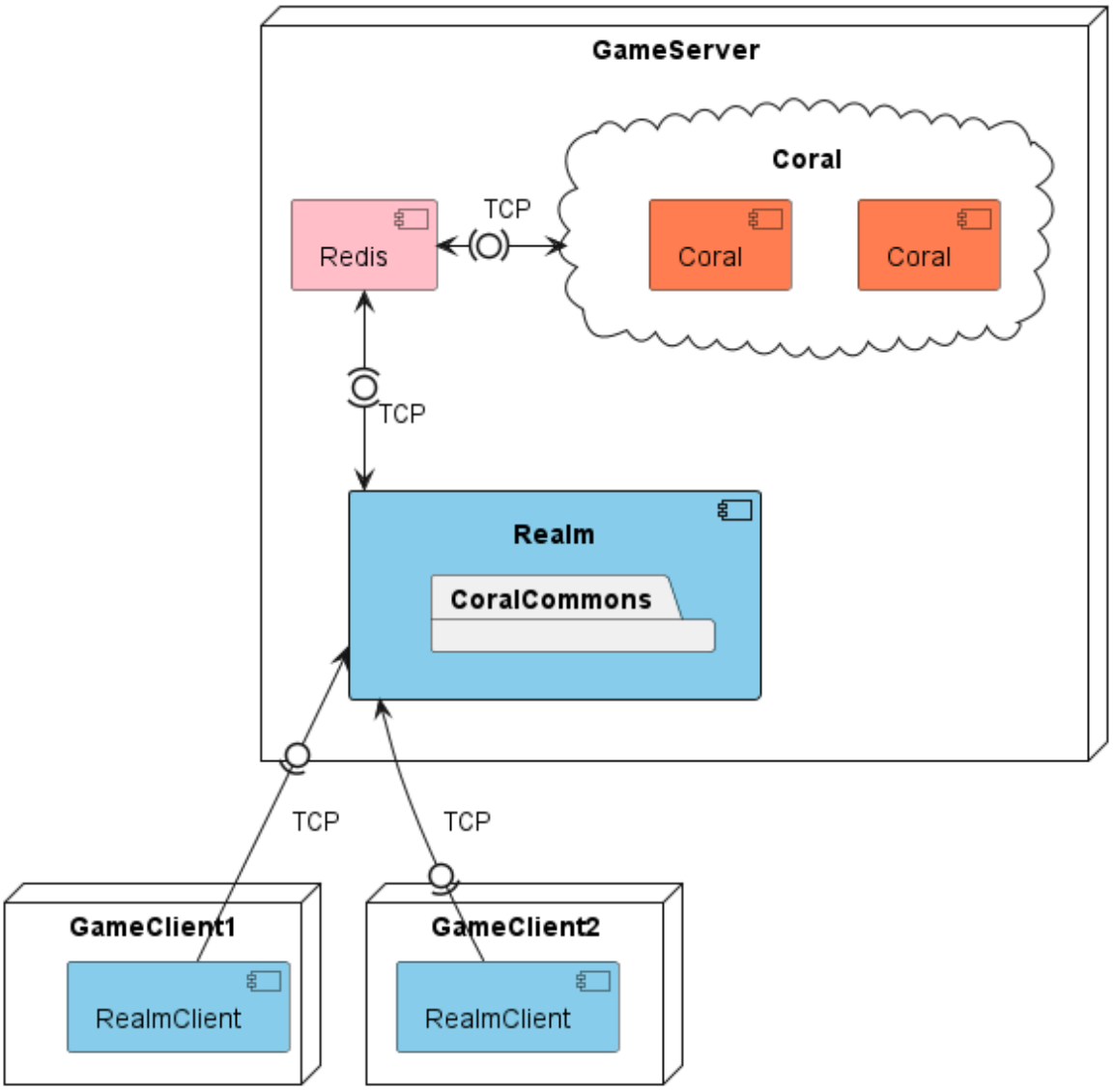
Rust ne supporte pas la réflexion à l'exécution du programme. Certaines librairies permettent de contourner ce problème en générant du code contenant les métadonnées nécessaires à décrire des types. En commençant par Syn[32] pour lire et analyser les fichiers de code. Quote[33] permet de prendre la structure de données lues et de la transformer en code source représentant les types. Quelques autres librairies comme Prost-build, Prost-reflect, Tonic-reflect utilisent des fonctions similaires pour générer le code des protobufs à la compilation et pouvoir accéder à leurs métadonnées à l'exécution.

---

[25] https://diesel.rs
[26] https://crates.io/crates/redis
[27] https://crates.io/crates/mongodb
[28] https://crates.io/crates/redis-om
[29] https://www.mongodb.com/developer/languages
[30] https://redis.io/docs/latest/integrate/redisom-for-net
[31] https://redis.io/docs/latest/integrate/redisom-for-java
[32] https://crates.io/crates/syn
[33] https://crates.io/crates/quote

Chapitre 2: Projet

Architecture proposée



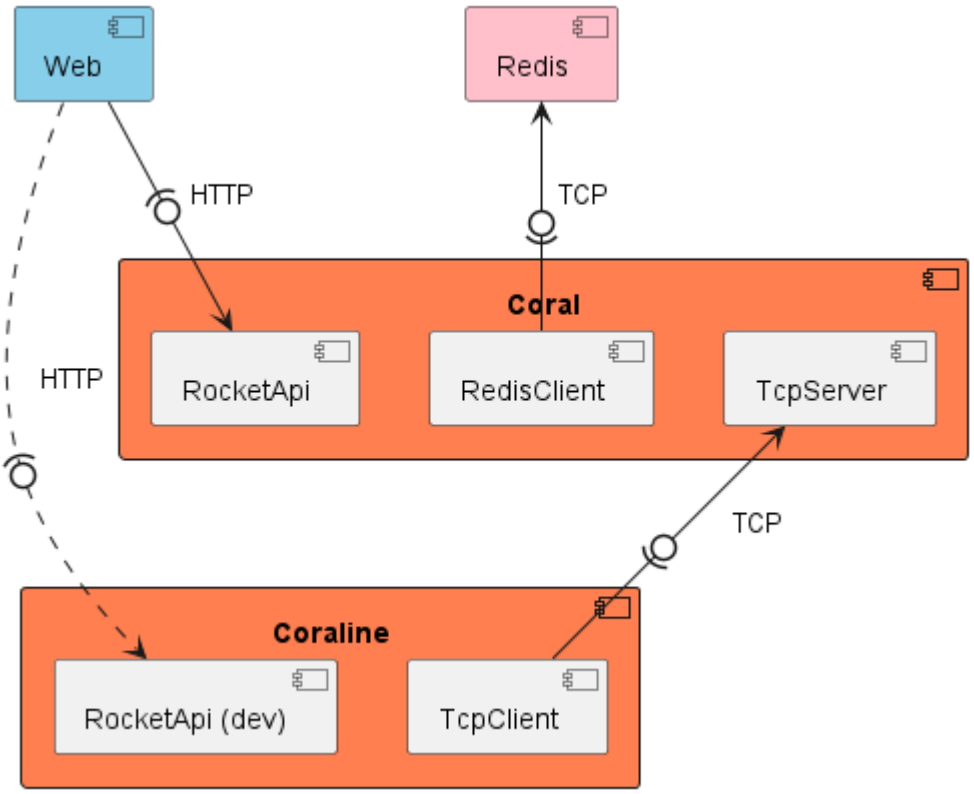
L'architecture proposée initialement incluait plusieurs instances du service de matchmaking avec le nom de code Coral. Realm représente un serveur de jeu ou de connexion quelconque utilisant la librairie coral-commons pour communiquer avec les services Coral.

Tous les services (Realm, Redis, Coral) peuvent être distribués sur des machines différentes. Via la communication par Redis, on peut établir un système d'équilibre de charge pour envoyer des requêtes à n'importe quelle instance du service Coral. Ces instances peuvent trouver un match entre le lobby de la requête et les lobbies inscrits sur la base de données Redis. Une fois la correspondance trouvée entre deux lobbies, une transaction permet de grouper les commandes pour supprimer les lobbies et créer le match. Si une autre instance de Coral exécute une transaction[34] sur les mêmes lobbies à ce

[34] https://redis.io/docs/latest/develop/interact/transactions/

moment-là, la transaction échouera car Redis est synchrone, ce qui permet d'éviter les conditions de course et de créer un seul match avec les mêmes lobbies.
Redis a été choisi car c'est un cache mémoire extrêmement rapide qui est bien adapté pour des données temporaires qui sont souvent modifiées, comme les lobbies et les matchs. Il permet aussi une communication TCP facile entre les services via son système pubsub et les notifications par espace-clé[35]. D'autres bases de données offrent une performance moindre pour ce genre d'applications, mais sont plus adaptées pour des données à long terme qui sont moins souvent lues, comme les informations utilisateur.

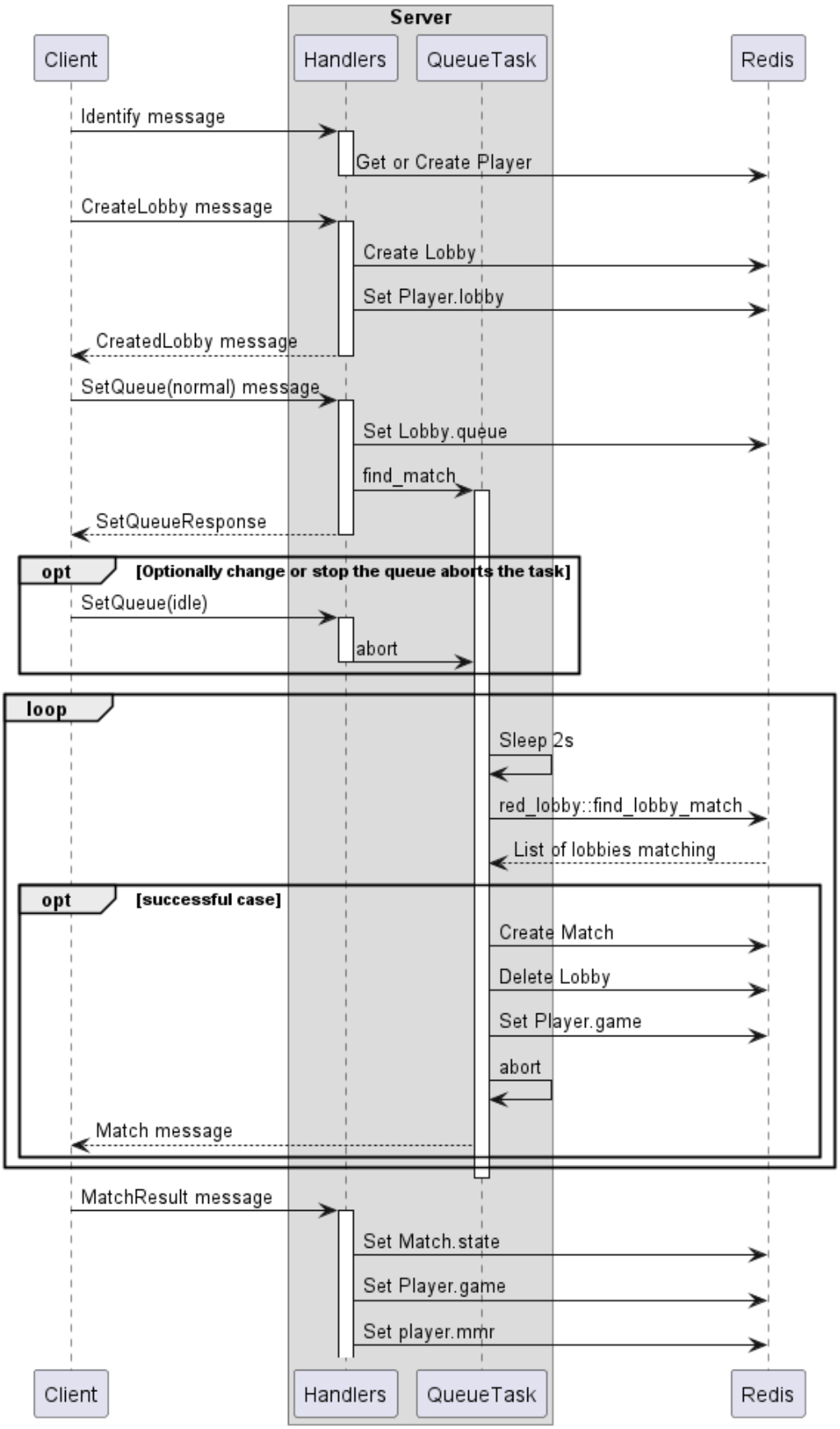
Architecture finale



Par manque de temps, beaucoup de fonctionnalités ont dû être coupées de la solution finale pour la présentation, mais les concepts de base sont tout de même démontrés. L'étape Realm a été complètement sautée. Les clients Coraline communiquent directement avec le serveur de matchmaking Coral. Le serveur et le client pourvoient une interface web afin de monitorer la base de données et d'envoyer des requêtes depuis les clients vers le serveur.

[35] https://redis.io/docs/latest/develop/use/keyspace-notifications/

Diagramme d’action

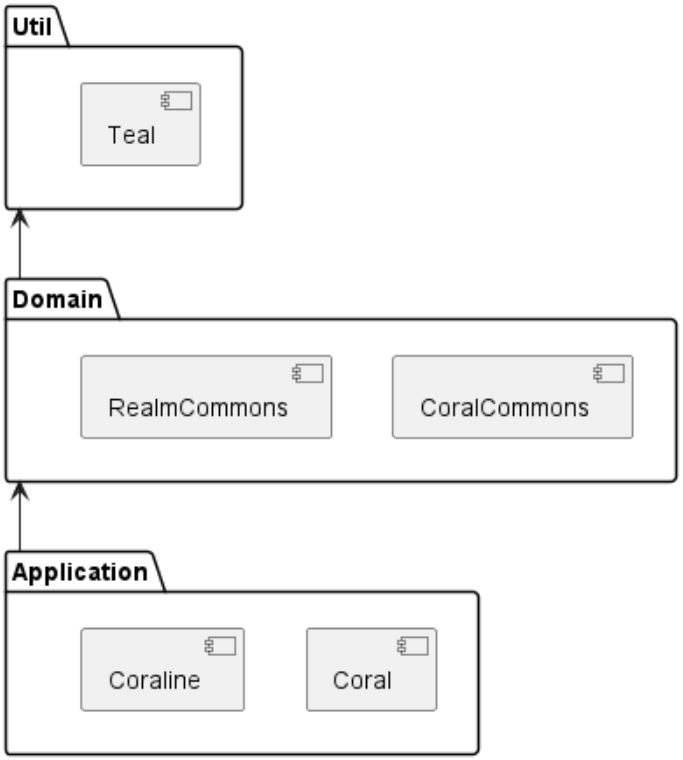


Structure de fichiers

RustProject

```
docs/
        diagrams/
projects/
        Layer - Util/
                teal/
                        src/
                        tests/
                        build.rs
                        Cargo.toml
        Layer 0 - Domain
                coral-commons/
                        src/
                        build.rs
                        Cargo.toml
                realm-commons/
                        src/
                        build.rs
                        Cargo.toml
        Layer 1 - Logic/
        Layer 2 - Application/
                coral/
                        src/
                        .env.dev
                        .env.docker
                        Cargo.toml
                        Dockerfile
                coraline/
                        src/
                        .env.dev
                        .env.docker
                        Cargo.toml
                        Dockerfile
        Layer 3 - Presentation/
        Layer 4 - Tests/
        .dockerignore
        docker-compose.yml
.gitignore
```

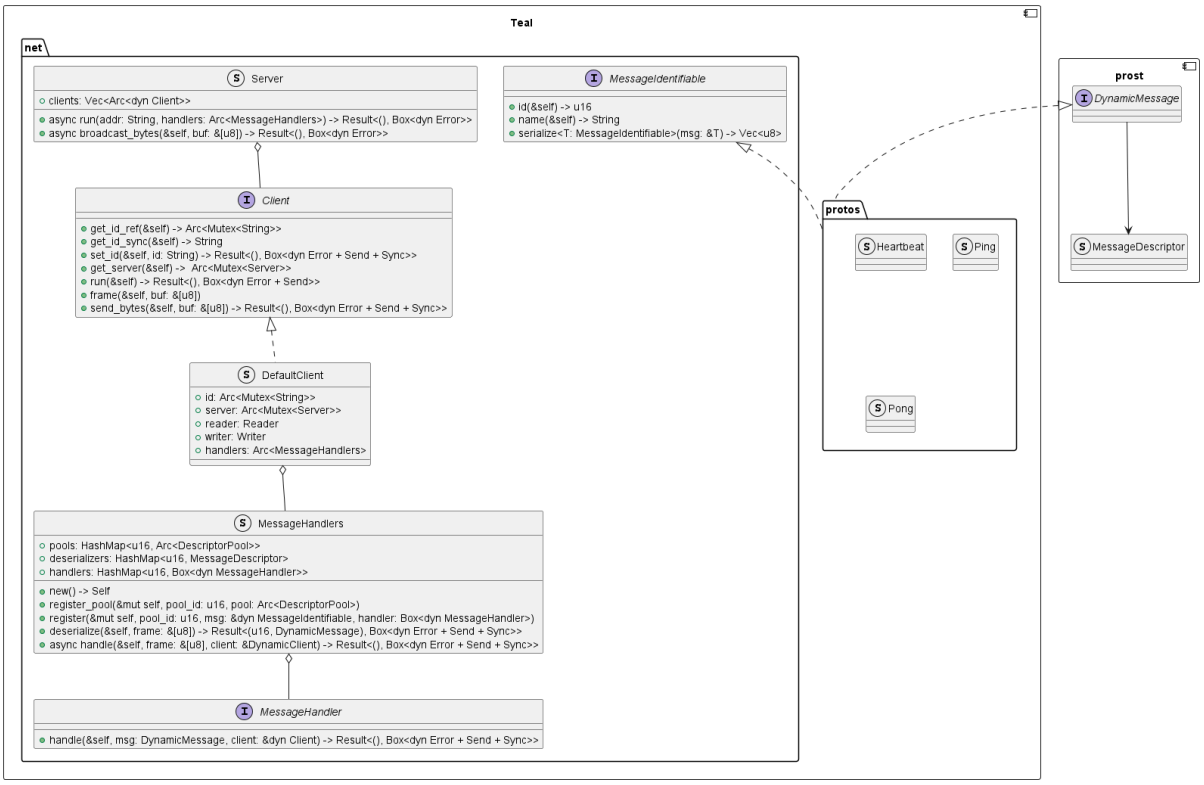
La structure par couche est respectée. Pour l'instant, il n'y a pas de projets dans les couches de logique, de présentation et de test.



Diagrammes de modules

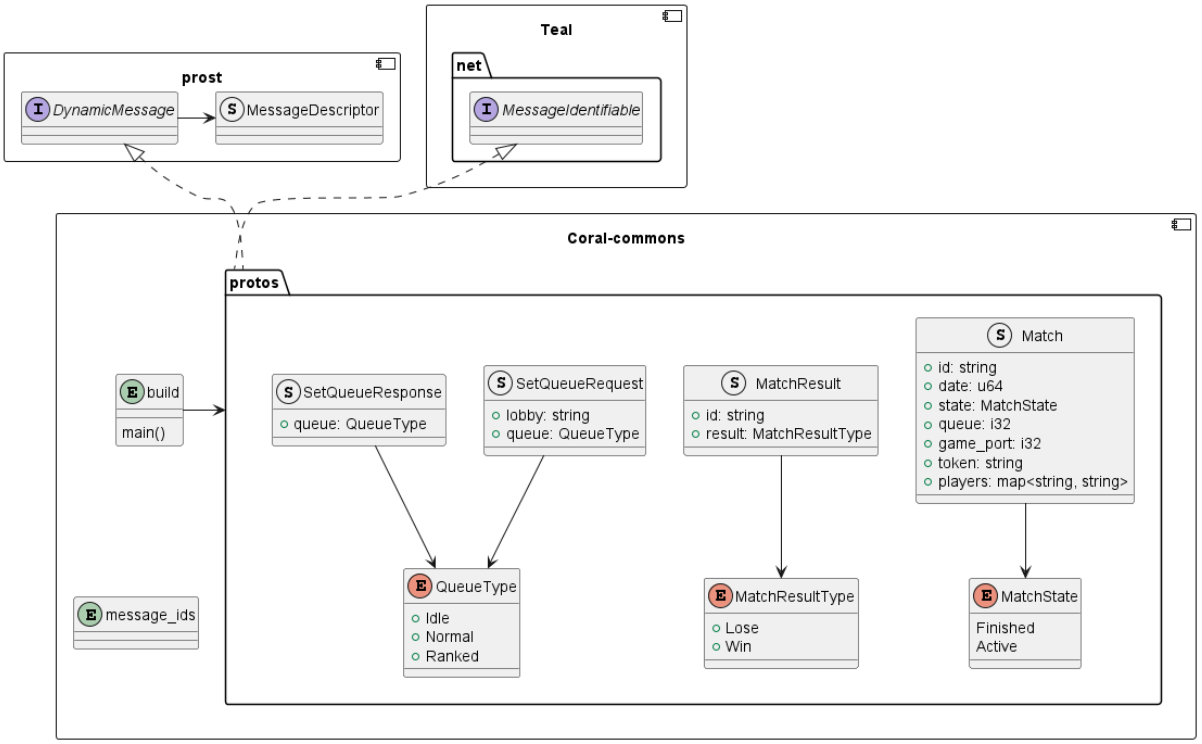
Teal

Teal sert de librairie de base pour la communication TCP et définir des messages communs.



Coral-commons

Coral-commons sert de librairie contenant les modèles et les messages spécifiques aux queues et au matchmaking. Les diagrammes sont séparés par package pour mieux voir.

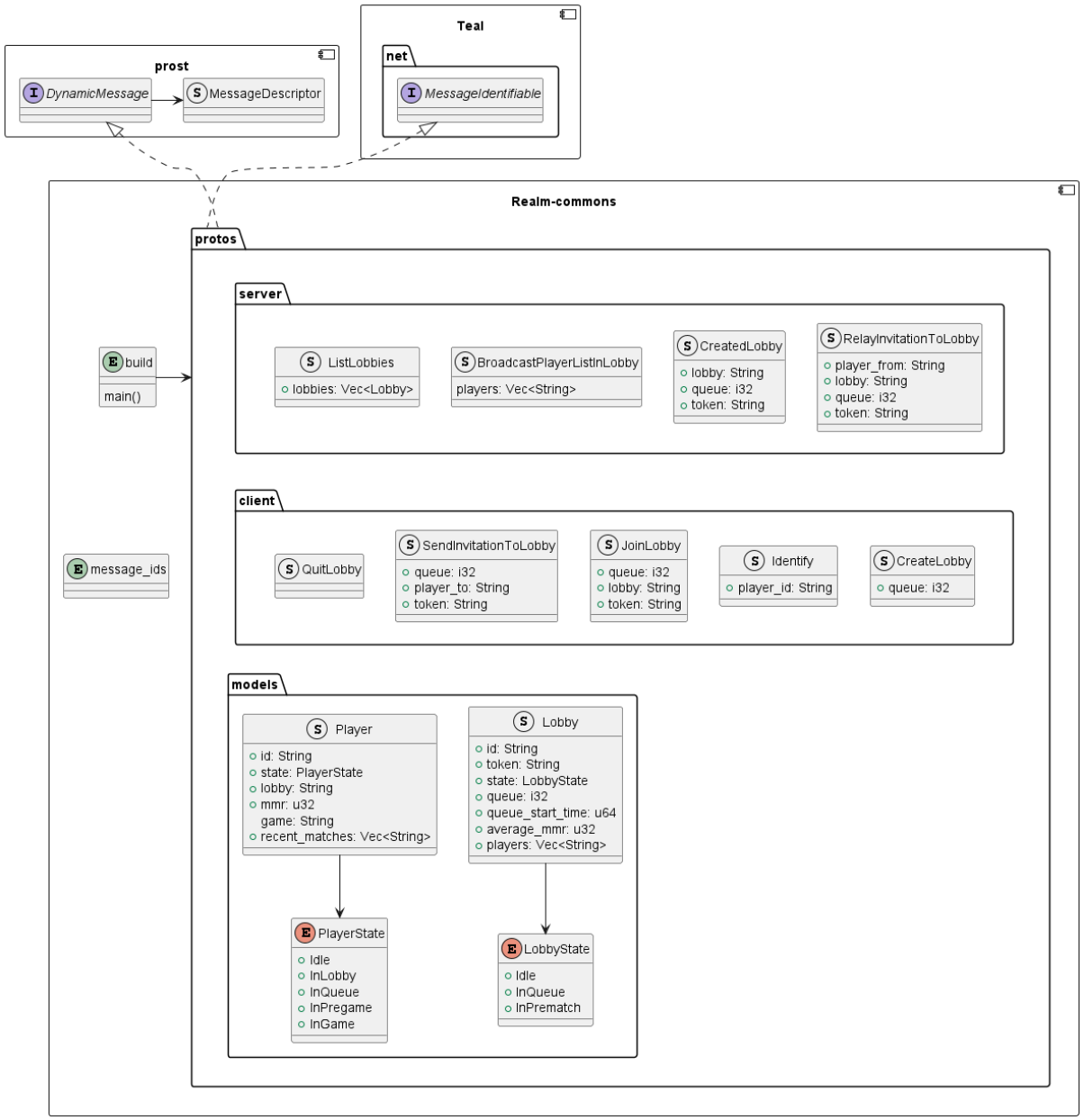


Le fichier message_ids.rs contient l'implémentation de MessageIdentifiable pour chaque message protobuf, ce qui leur donne un identifiant chacun.
Le fichier build.rs permet de générer le code Rust des messages et modèles à partir des fichiers *.proto qui sont dans le dossier protos. Les fichiers générés sont ensuite inclus dans le module protos. Chaque message implémente MessageIdentifiable et DynamicMessage, mais ce n'est pas le cas pour les enums; les flèches sont simplifiées.

Accès à Redis pour les matchs :

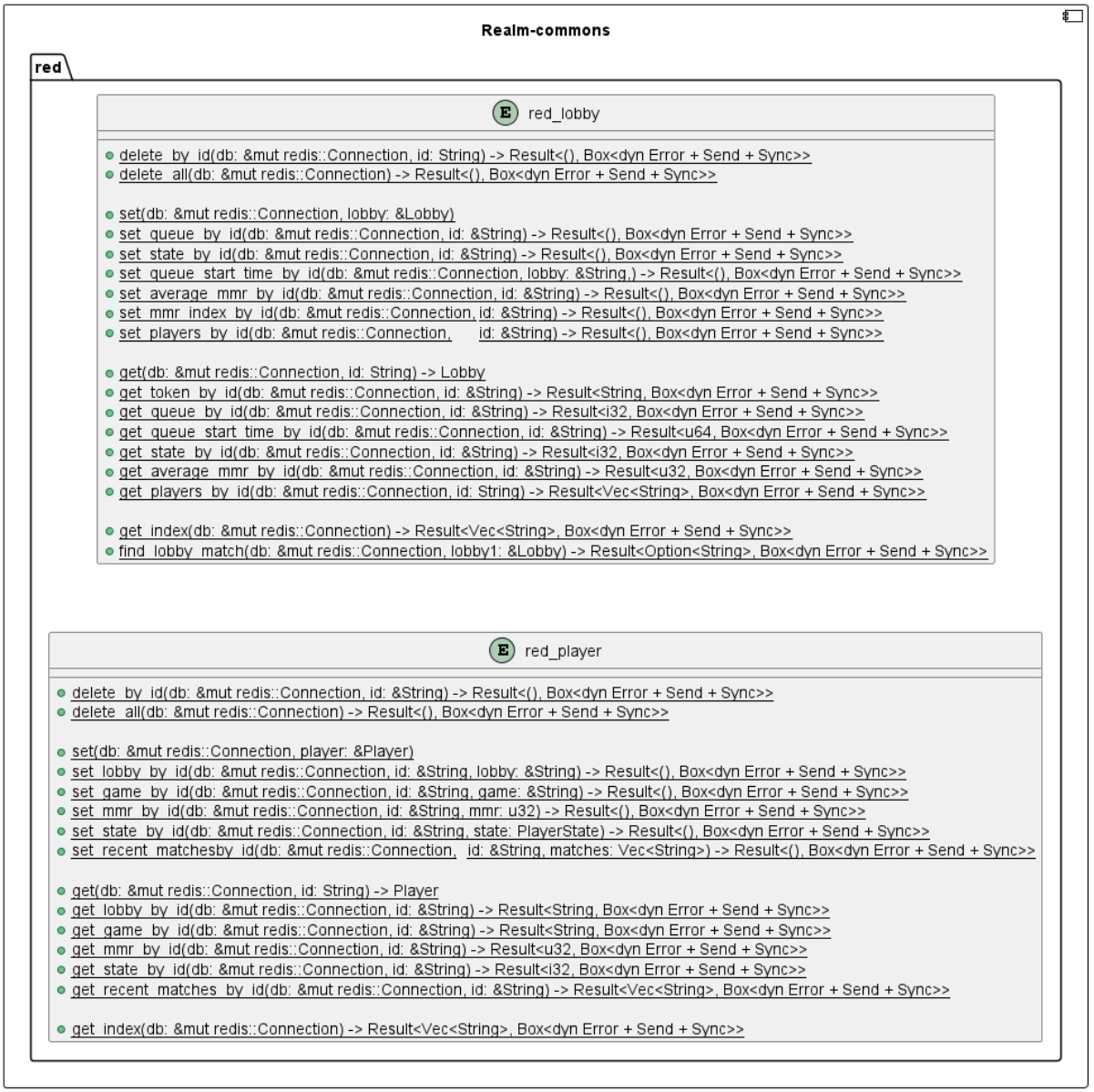
Realm-commons

Realm-commons sert de librairie contenant les modèles et les messages applicables aux lobbys et aux joueurs.
Messages et modèles protobuf :



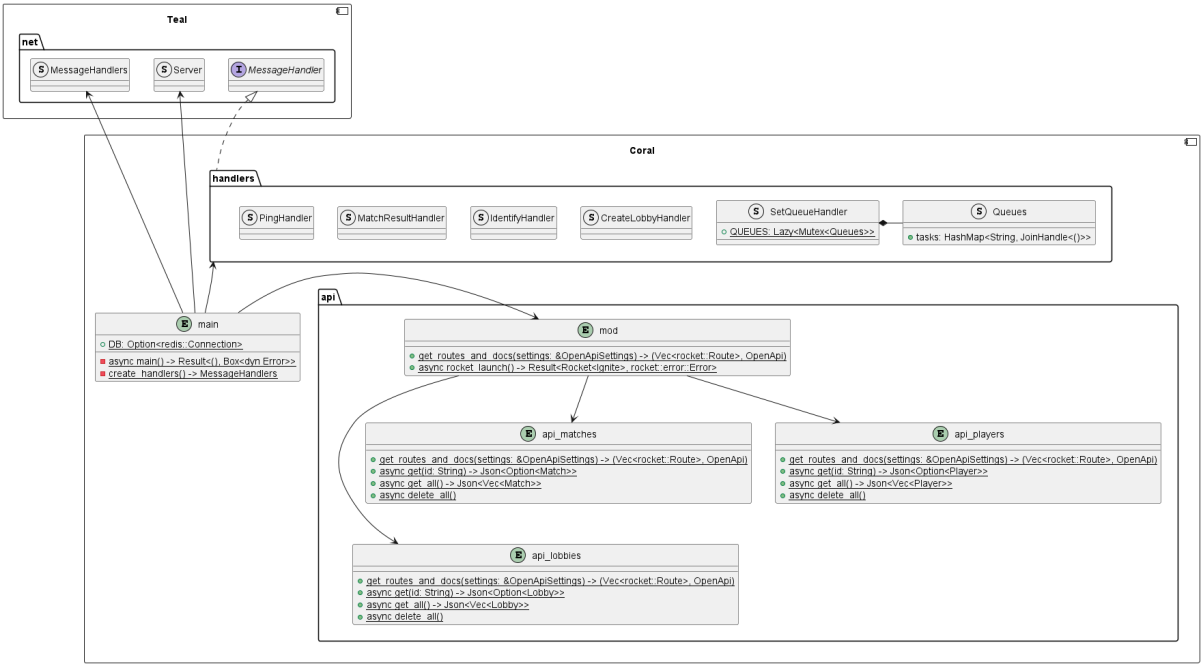
Même structure que pour coral-commons ici.

Accès à Redis pour les lobbies et les joueurs:



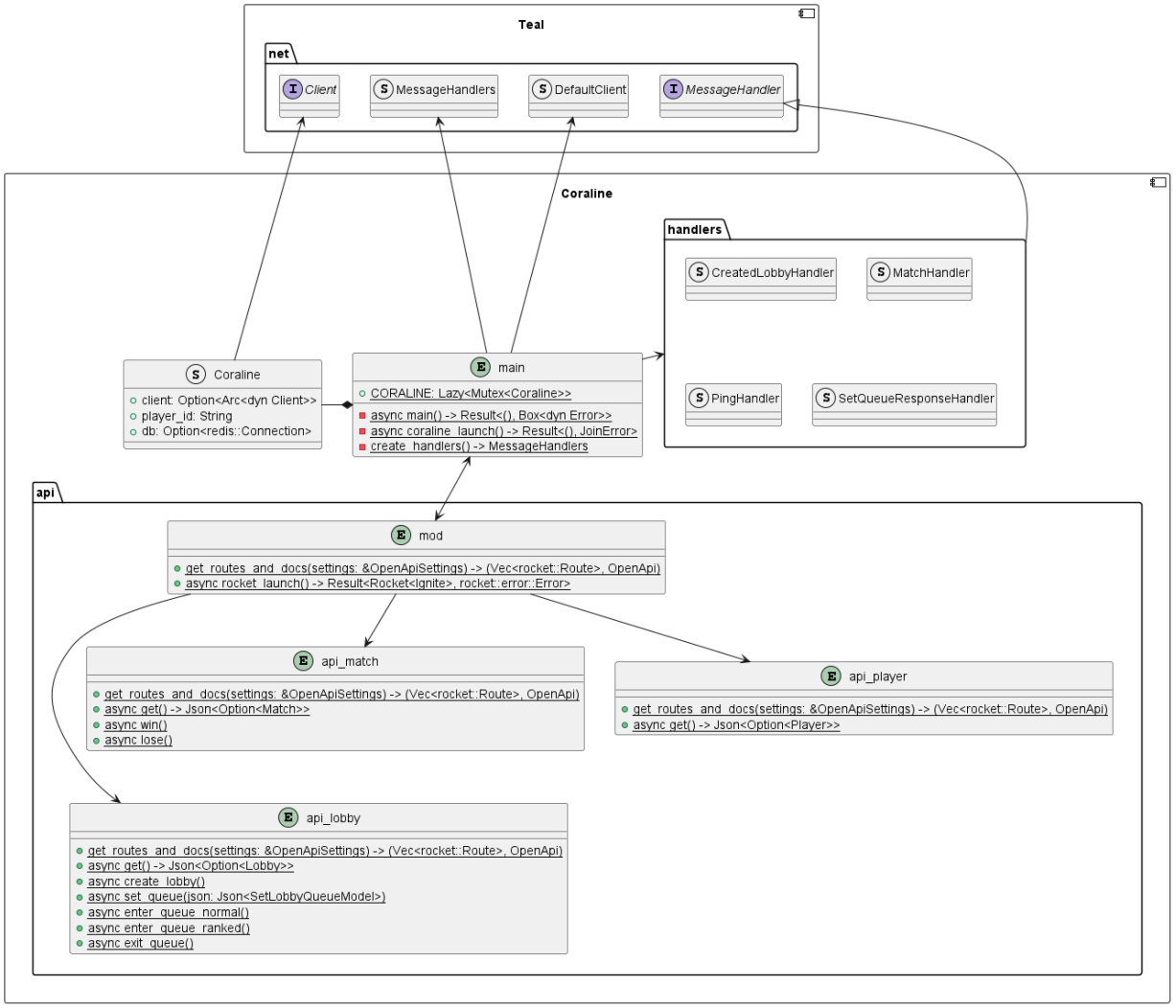
Coral

Coral est l'application servant de serveur de matchmaking. Il reçoit les requêtes des clients, crée des lobbies et trouve des matchs.



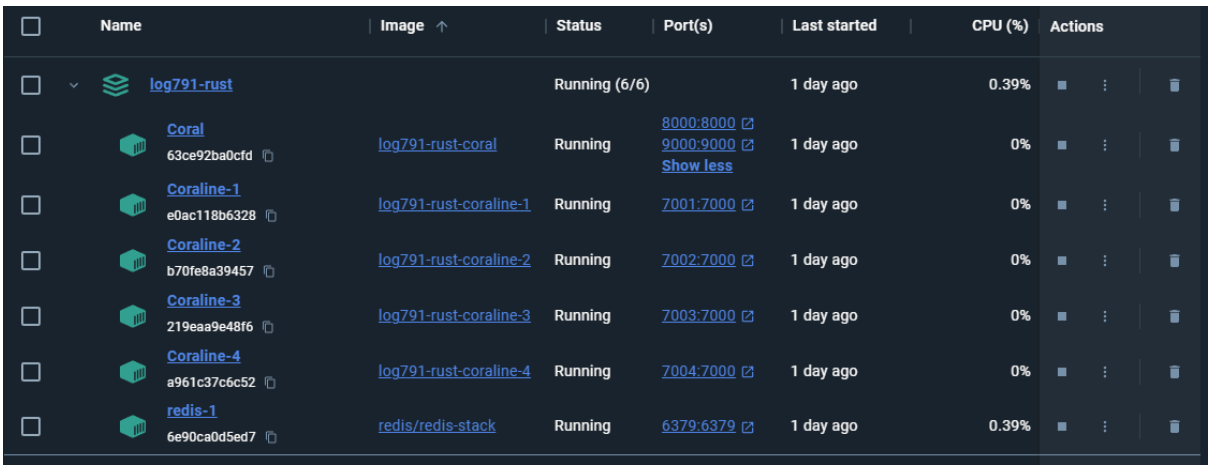
Coraline

Coraline sert d'application client représentant un joueur.



Déploiement Docker

Docker-compose permet de gérer le lancement du serveur Coral, Redis, et de quelques clients Coraline.



Name	Image	Status	Port(s)	Last started	CPU (%)	Actions
log791-rust		Running (6/6)		1 day ago	0.39%	
Coral 63ce92ba0cfd	log791-rust-coral	Running	8000:8000 9000:9000 Show less	1 day ago	0%	
Coraline-1 e0ac118b6328	log791-rust-coraline-1	Running	7001:7000	1 day ago	0%	
Coraline-2 b70fe8a39457	log791-rust-coraline-2	Running	7002:7000	1 day ago	0%	
Coraline-3 219eaa9e48f6	log791-rust-coraline-3	Running	7003:7000	1 day ago	0%	
Coraline-4 a961c37c6c52	log791-rust-coraline-4	Running	7004:7000	1 day ago	0%	
redis-1 6e90ca0d5ed7	redis/redis-stack	Running	6379:6379	1 day ago	0.39%	

```yaml
version: '3.8'
services:
  redis:
    image: redis/redis-stack
    restart: always
    ports:
      - '6379:6379'
  coral:
    container_name: Coral
    depends_on:
      - redis
    ports:
      - '8000:8000'
      - '9000:9000'
    build:
      context: ./
      dockerfile: Layer 2 - Application/coral/Dockerfile
  coraline-1:
    container_name: Coraline-1
    depends_on:
      - redis
      - coral
    ports:
      - '7001:7000'
    command: ["1"]
    build:
      context: ./
      dockerfile: Layer 2 - Application/coraline/Dockerfile
  coraline-2:
    container_name: Coraline-2
    depends_on:
      - redis
      - coral
    ports:
      - '7002:7000'
    command: ["2"]
    build:
      context: ./
      dockerfile: Layer 2 - Application/coraline/Dockerfile
```

(image écourtée)

Chaque client prend un identifiant en paramètre de commande pour créer son joueur et l'identifier auprès du serveur.

Les images Docker sont relativement optimisées pour réduire leur taille à 127mb, par exemple pour Coral:

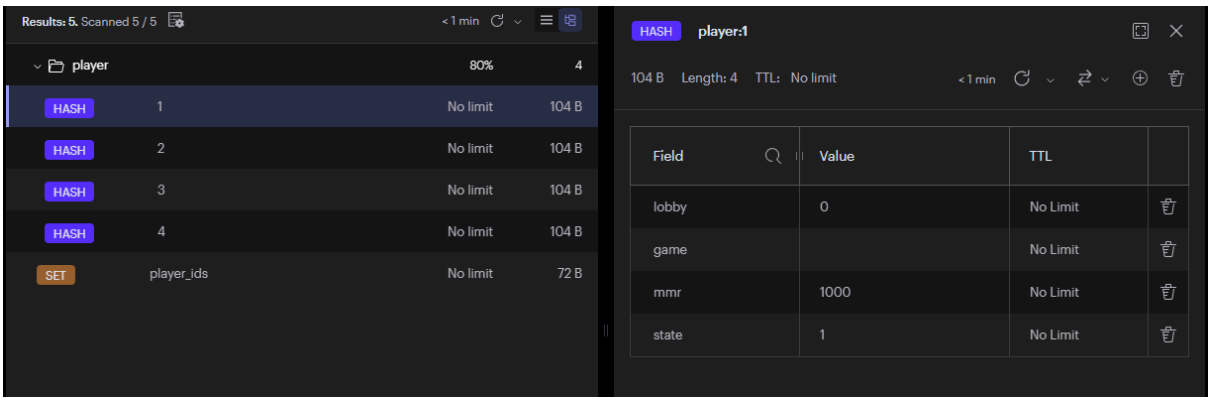
```
# Rust as the base image
FROM rust:latest AS build

# 1. Create empty shell projects
RUN USER=root cargo new --lib "./Layer - Util/teal"
RUN USER=root cargo new --lib "./Layer 0 - Domain/coral-commons"
RUN USER=root cargo new --lib "./Layer 0 - Domain/realm-commons"
RUN USER=root cargo new --bin "./Layer 2 - Application/coral"

# 2. Copy our manifests
COPY ["./Layer - Util/teal/Cargo.toml", "./Layer - Util/teal/Cargo.toml"]
COPY ["./Layer 0 - Domain/coral-commons/Cargo.toml", "./Layer 0 - Domain/coral-commons/Cargo.toml"]
COPY ["./Layer 0 - Domain/realm-commons/Cargo.toml", "./Layer 0 - Domain/realm-commons/Cargo.toml"]
COPY ["./Layer 2 - Application/coral/Cargo.toml", "./Layer 2 - Application/coral/Cargo.toml"]

# 3. Build only the dependencies to cache them
WORKDIR "/Layer 2 - Application/coral"
RUN cargo build --release
# 3.1 Install Protoc
RUN apt-get update
RUN apt install -y protobuf-compiler

# 4. Remove default sources from shell projects
WORKDIR "/"
RUN rm "/Layer - Util/teal/src/"*.rs
RUN rm "/Layer 0 - Domain/coral-commons/src/"*.rs
RUN rm "/Layer 0 - Domain/realm-commons/src/"*.rs
RUN rm "/Layer 2 - Application/coral/src/"*.rs

# 5. Copy source code
COPY ["./Layer - Util/teal/src", "./Layer - Util/teal/src"]
COPY ["./Layer - Util/teal/build.rs", "./Layer - Util/teal/build.rs"]
COPY ["./Layer 0 - Domain/coral-commons/src", "./Layer 0 - Domain/coral-commons/src"]
COPY ["./Layer 0 - Domain/coral-commons/build.rs", "./Layer 0 - Domain/coral-commons/build.rs"]
COPY ["./Layer 0 - Domain/realm-commons/src", "./Layer 0 - Domain/realm-commons/src"]
COPY ["./Layer 0 - Domain/realm-commons/build.rs", "./Layer 0 - Domain/realm-commons/build.rs"]
COPY ["./Layer 2 - Application/coral/src", "./Layer 2 - Application/coral/src"]

# 6. Build for release.
WORKDIR "/Layer 2 - Application/coral/"
RUN cargo build --release

WORKDIR "/Layer 2 - Application/coral/target/release"
RUN ls

# Use a slim Dockerfile with just our app to publish
FROM debian:latest AS app
COPY --from=build ["/Layer 2 - Application/coral/target/release/coral", "/coral"]
COPY ["/Layer 2 - Application/coral/.env.docker", "/.env.docker"]

ENV ENV_FILE=.env.docker
EXPOSE 8000
EXPOSE 9000

ENTRYPOINT ["/coral", ".env.docker"]
```

Les fichiers d'environnement permettent de configurer les adresses des services en développement et en production :

```
REDIS_URL=host.docker.internal:6379
CORAL_URL=0.0.0.0:8000
ROCKET_ADDRESS=0.0.0.0
ROCKET_PORT=9000
```

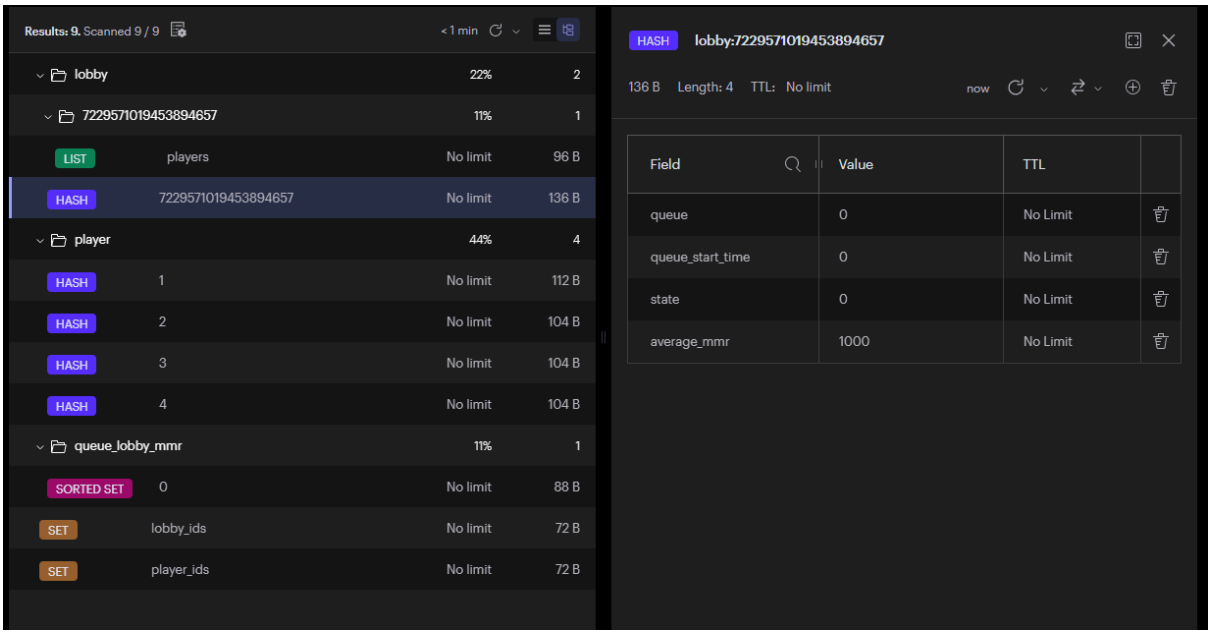
Schéma de base de données

Lorsque les clients s'identifient au démarrage, le serveur génère alors un joueur pour chacun:

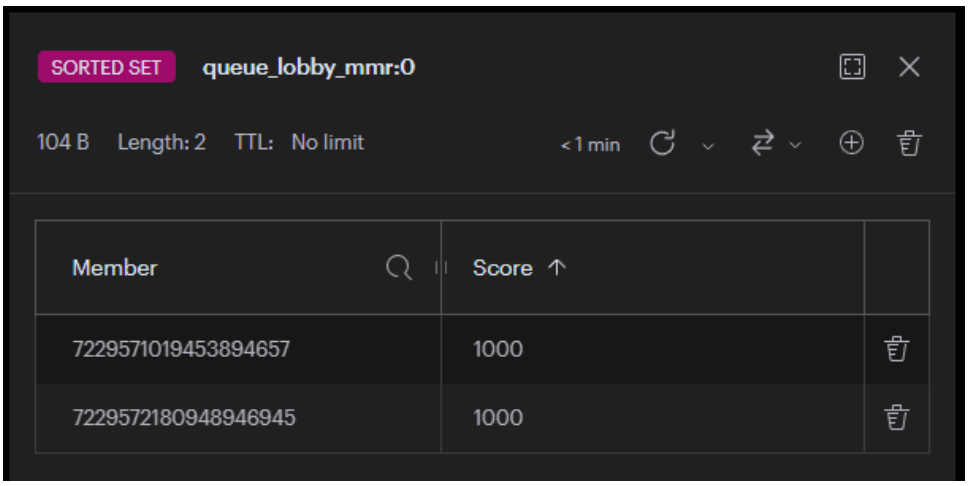


La clé player_ids contient l'ensemble des identifiants des joueurs servant d'index.

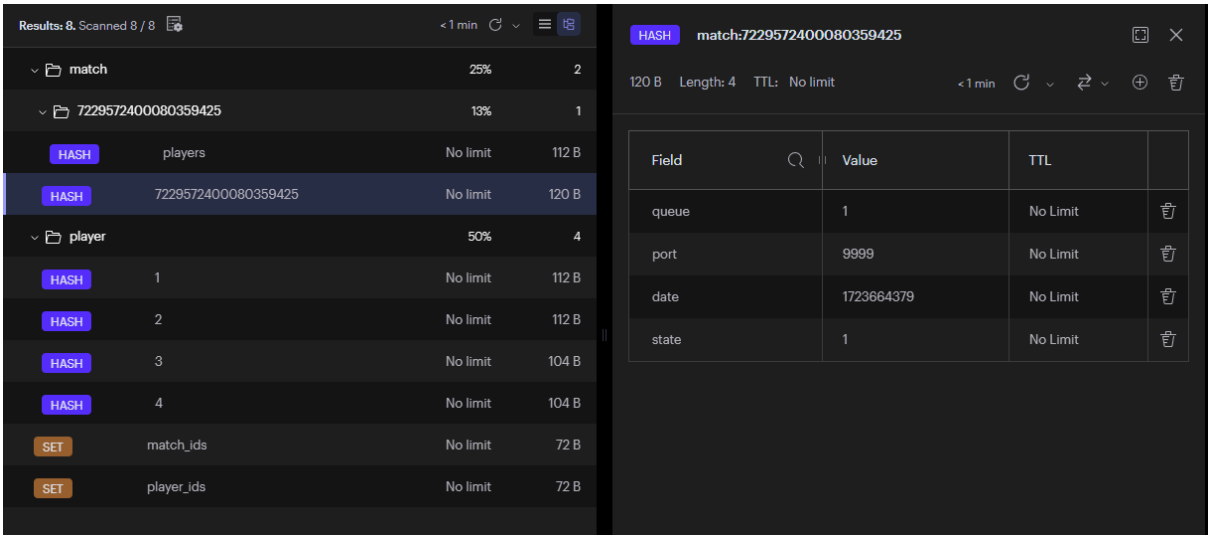
Un client peut ensuite envoyer une requête pour créer un Lobby. Le serveur crée le lobby et le retourne.



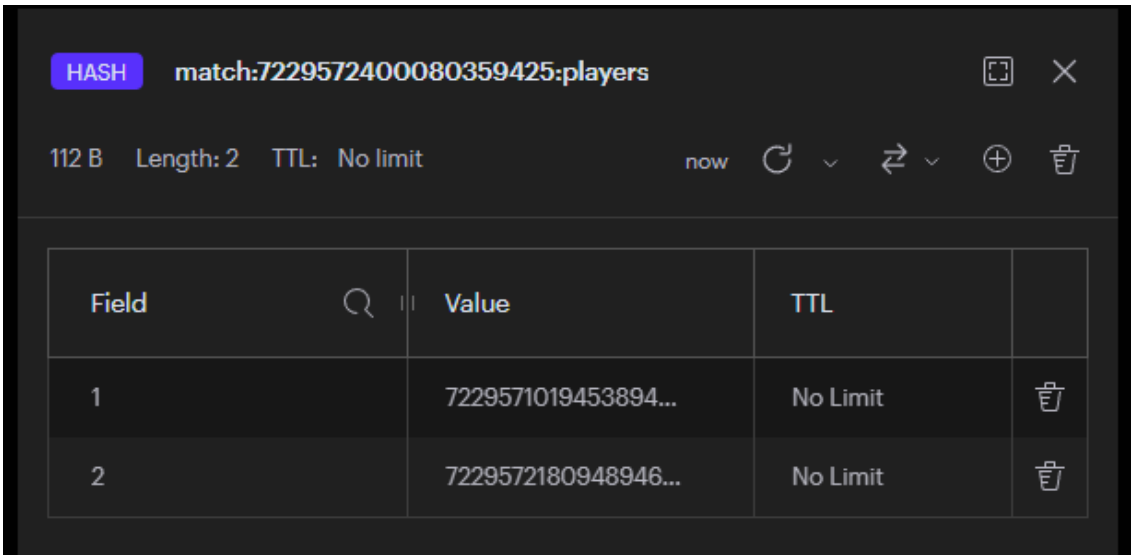
Les lobbies ont aussi un index dans lobby_ids. Ils ont aussi la liste des joueurs contenus dans *lobby:{lobby_id}:players*. Enfin, une liste ordonnée contient la moyenne d'MMR de chaque lobby dans une certaine queue: *queue_lobby_mmr:{queue_id}*. Cela nous permet de trouver une portée de lobbys dans une même queue ayant un MMR contenu entre un minimum et un maximum proche de celui qui envoie la requête.



Une fois un match trouvé, il est créé et les lobbies correspondants sont supprimés :



Comme pour le lobby, le match contient une liste de joueurs à la clé *match:{match_id}:players*. Cette fois elle est sous forme de hash pour faire correspondre chaque joueur (à gauche) à une équipe (à droite). L'identifiant d'équipe est l'ancien identifiant du lobby dans lequel ils étaient.

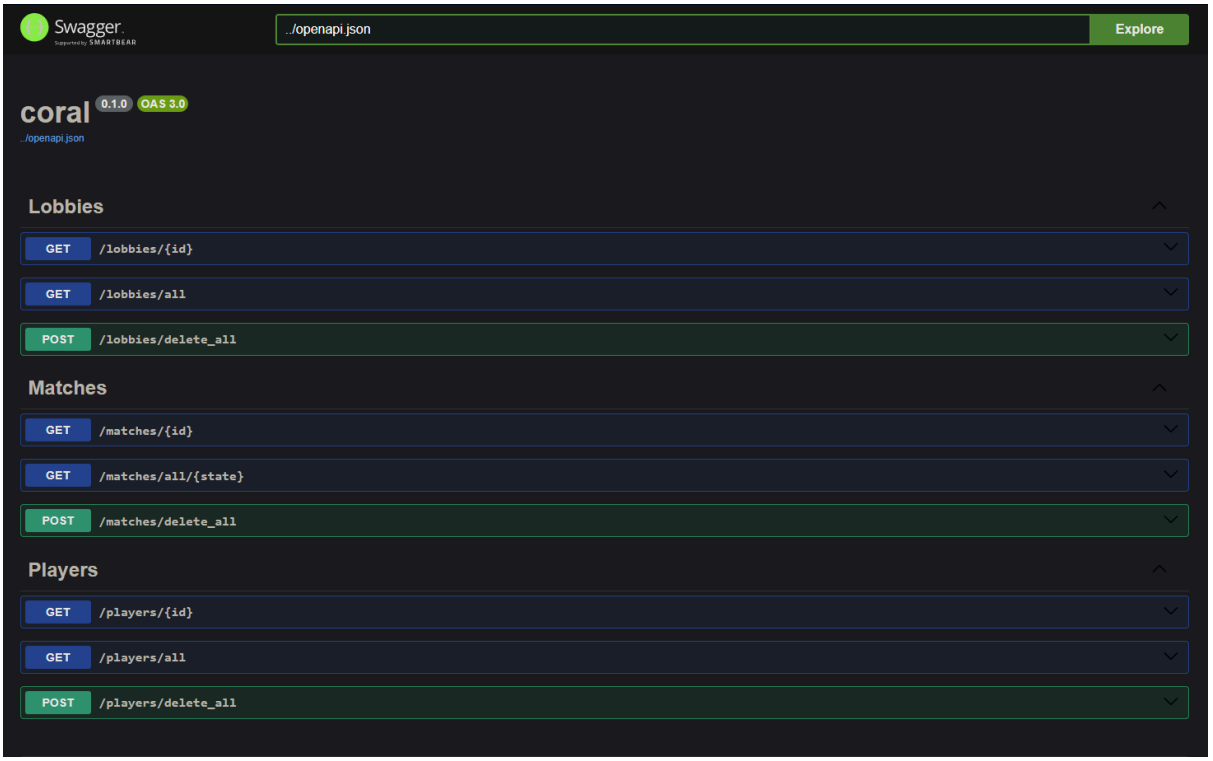


API Web

Une API web permet de monitorer le serveur et de contrôler les clients pour envoyer des commandes.

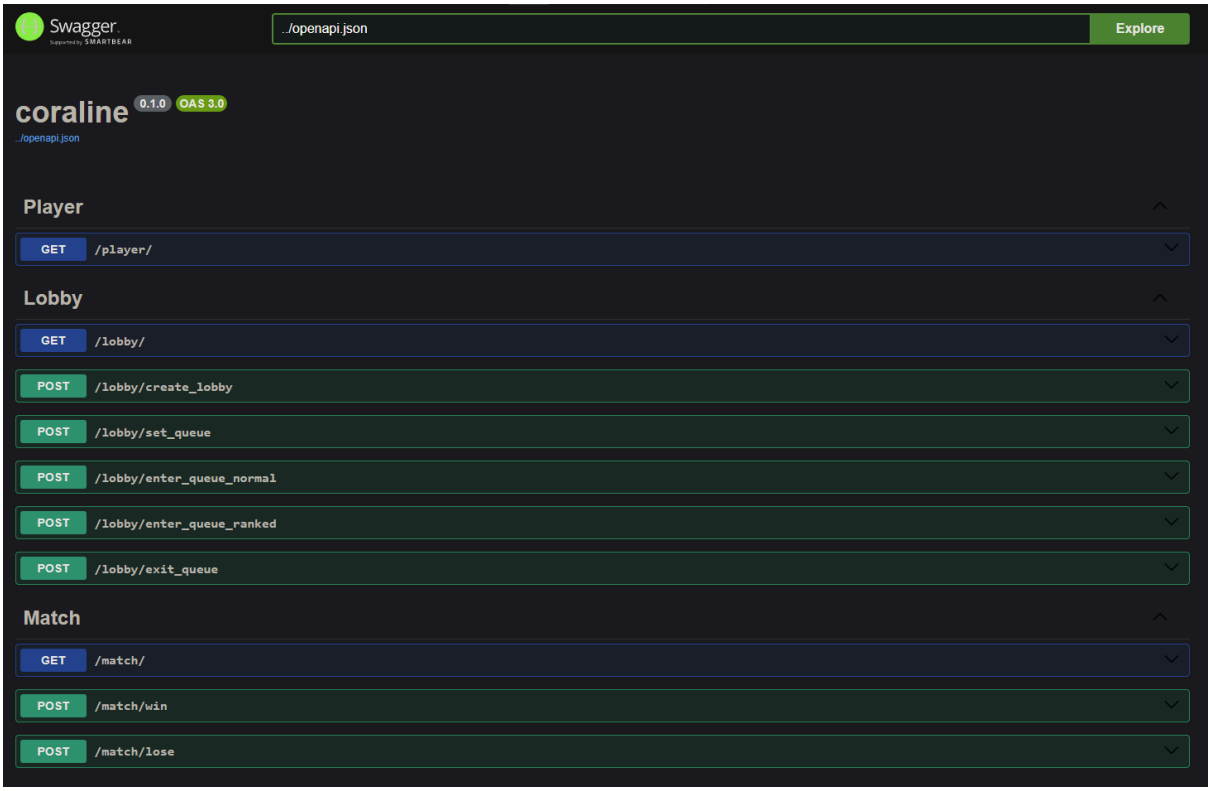
Interface serveur

Celle-ci permet d'observer les données de Lobby, Match et Player qui sont sur la base de données.



Interface client

Celle-ci permet d'observer le joueur et son lobby; créer un lobby, le mettre dans une queue et activer la file d'attente; observer le match en cours et le gagner ou le perdre.



Chapitre 3: Difficultés éprouvées

1. Choisir une seule architecture.

Beaucoup de solutions étaient possibles selon les consignes données. Par exemple, lorsqu'on pense à un système hautement scalable, on pense à des microservices réplicables entre lesquels on peut partager la charge. Comment alors supporter le partage de charge et la gestion des pannes? Avec un proxy? Un gestionnaire de microservices? Un algorithme auto-suffisant comme Raft? Est-ce que le serveur Realm permet aux clients de se connecter directement à une des instances de matchmaking? Est-ce que Realm est lui-même instancié aussi? Une fois cela élucidé, comment envoyer l'information d'un match à tous ses joueurs s'ils sont connectés sur des serveurs différents? Comment éviter les problèmes de concurrence si plusieurs instances de Coral peuvent trouver des matchs en même temps? Une architecture de plus en plus compliquée demande plus de travail et la limite de temps fait qu'on doit couper des fonctionnalités pour se concentrer sur ce qui est important.

2. Trouver des librairies complètes

Pour la génération de messages protobuf, trois librairies ont été testées extensivement avant d'en trouver une qui supporte tous les requis.

Pour le serveur web, une multitude de librairies furent explorées sans en trouver une supportant la génération automatique de spécification OpenApi et de page Swagger. Rocket[36] et Okapi ont été trouvés à la dernière minute, ce qui a permis d'avancer le projet en débloquant beaucoup de fonctionnalités.

3. Programmer la lecture de paquets manuellement.

Tokio offre un TcpListener et TcpStream. Tout le reste a été implémenté manuellement dans la librairie Teal. La gestion de threads serveur et la dé/fragmentation d'octets en paquets particulièrement sont des tâches peu adaptées à un projet débutant en Rust. Il aurait été mieux d'utiliser une librairie complète comme Netty[37] en Java, mais il était difficile de trouver une telle solution en Rust. Netty-rs[38] existe, mais n'a pas été mis à jour depuis 3 ans. Retty[39] est inactif depuis 5 mois et leur site web est mort. Encore une fois, l'écosystème laisse à désirer.

[36] https://rocket.rs
[37] https://netty.io
[38] https://crates.io/crates/netty-rs
[39] https://crates.io/crates/retty

4. Les patrons OOP en Rust.

Singleton

Il existe quelques manières de créer un singleton en Rust[40], mais elles ne sont pas encouragées ou encourageantes dépendamment du contexte.
Par exemple, l'API de base de données Redis fournie par la librairie[41] utilisée dans le serveur Coral nécessite que l'objet *Connection* soit mutable pour lire et écrire des données. Si on ne veut pas recréer la connexion plusieurs fois, cela requiert un singleton mutable. Puisque nous sommes dans un contexte multithread, Rust nous oblige à soit utiliser le mot clé unsafe, soit avoir un verrou sur notre variable pour y accéder à partir de plusieurs threads à la fois. Une autre manière serait d'utiliser de l'injection de dépendance, mais le patron n'est pas encouragé en Rust puisqu'il ne supporte pas la réflexion à l'exécution; des librairies permettent de générer du code correspondant à la compilation. Utiliser un verrou sur la base de données n'est théoriquement pas nécessaire, étant donné que Redis est synchrone. De plus, le verrou ralentirait chaque thread qui essaie d'y accéder en même temps. Il fut donc choisi d'utiliser le mot-clé unsafe pour y accéder :

Coral/src/main.rs

```
pub static mut DB: Option<redis::Connection> = None;

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
…
    let redis_url = env::var("REDIS_URL").unwrap();
    let redis_client = redis::Client::open("redis://".to_string() + &redis_url)?;
    let conn = redis_client.get_connection()?;
    unsafe {
        DB = Some(conn);
    }
…
}
```

Coral/src/api/api_players.rs

```
#[openapi(tag = "Players")]
#[get("/<id>")]
async fn get(id: String) -> Json<Option<Player>> {
    unsafe {
        if let Some(db) = &mut crate::DB {
            let opt_player = red_player::get(db, &id).ok();
            return Json(opt_player);
        }
    }
    return Json(None);
}
```

[40] https://refactoring.guru/design-patterns/singleton/rust/example
[41] https://crates.io/crates/redis

À l'inverse, le client Coraline n'exécute qu'une chose à la fois et n'a pas besoin d'une grosse performance. Donc un verrou est utilisé :

Coraline/src/main.rs

```
#[derive(Default)]
pub struct Coraline {
    // TCP Client
    pub client: Option<Arc<dyn Client>>,
    // Client's player id
    pub player_id: String,
    // Client wouldn't have a DB connection in real life but do server requests
instead. Used as a dev shortcut.
    pub db: Option<redis::Connection>,
}
pub static CORALINE: Lazy<Mutex<Coraline>> = Lazy::new(||
Mutex::new(Coraline::default()));
```

Coralin/src/api/api_player.rs

```
#[openapi(tag = "Player")]
#[get("/")]
async fn get() -> Json<Option<Player>> {
    let mut coraline = crate::CORALINE.lock().await;
    let player_id = coraline.player_id.clone();
    if let Some(db) = &mut coraline.db {
        let player = red_player::get(db, &player_id).ok();
        return Json(player);
    }
    return Json(None);
}
```

Polymorphisme

Avec l'expérience des langages OOP tels que Java, C#, Typescript et même C++, le polymorphisme est un aspect qui n'était pas intuitif dans Rust et qui a posé quelques problèmes. Tout d'abord, dans les langages OOP traditionnels, on peut référencer des interfaces directement sans problème :

```
interface Animal {}
class Dog : Animal {}
public void polymorphism() {
    List<Animal> animals = new List<Animal>();
    animals.Add(new Dog());
}
```

Si on traduit littéralement la même chose en Rust, on tombe sur l'erreur suivante :

```
trait Animal {}
#[derive(Default)]
struct Dog {}
```

```
impl Animal for Dog {}
fn polymorphism() {
    let mut animals: Vec<Animal> = Vec::new();
    animals.push(Dog::default());
}
```

```
add `dyn` keyword before this trait: `dyn `
```

En ajoutant dyn tel que demandé:

```
fn polymorphism() {
    let mut animals: Vec<dyn Animal> = Vec::new();
    animals.push(Dog::default());
}
```

```
the size for values of type `dyn Animal` cannot be known at compilation time
the trait `Sized` is not implemented for `dyn Animal`
required by an implicit `Sized` bound in `std::vec::Vec`
```

Finalement, l'erreur n'est pas très claire et ne donne pas une solution directe. Après plus de recherche, on apprend qu'il faut encapsuler les traits dynamiques dans un pointeur pour leur donner une taille fixe en mémoire :

```
fn polymorphism() {
    let mut animals: Vec<Box<dyn Animal>> = Vec::new();
    animals.push(Box::new(Dog::default()));
}
```

Ensuite, pour soutirer un *Animal* et assumer que c'est une instance de *Dog*:

```
Dog dog = (Dog) animals.get(0);
```

En Rust, on ne peut pas utiliser une syntaxe similaire.

```
let animal = animals.get(0) as Box<Dog>;
```

```
non-primitive cast: `std::option::Option<&Box<dyn Animal>>` as `Box<Dog>`
an `as` expression can only be used to convert between primitive types or to
coerce to a specific trait object
```

Certaines librairies comme rust-protobuf utilisent des fonctions comme *downcast_box* avec le mot clé *unsafe*, mais cela ne fonctionne pas dans toutes les situations.
Finalement, on trouve une solution dans un article[42] :

```
trait Animal {
    fn as_any(&self) -> &dyn Any;
}
#[derive(Default)]
struct Dog {
```

[42] https://ysantos.com/blog/downcast-rust

```
    name: String
}
impl Animal for Dog {
    fn as_any(&self) -> &dyn Any {
        self
    }
}
#[test]
fn polymorphism() {
    let mut animals: Vec<Box<dyn Animal>> = Vec::new();
    let mut dog = Dog::default();
    dog.name = "Foo".to_string();
    animals.push(Box::new(dog));

    // Retrieve dog
    let animal = animals.get(0).unwrap();
    let dog2 = animal.as_any().downcast_ref::<Dog>().unwrap();
    println!("Dog: {}", &dog2.name);
    assert_eq!("Foo".to_string(), dog2.name);
}
```

C'est encore un exemple très convoluté pour un concept habituellement très simple. De plus, c'est dommage d'utiliser le trait *Any*, mais c'est la solution la plus simple et la plus universelle. En particulier si *Animal* fait partie d'une librairie et *Dog* fait partie du logiciel qu'on développe. Si on ne peut pas modifier *Animal*, c'est la seule solution fonctionnelle avant de devoir utiliser *unsafe*. Le problème est le même lorsqu'il s'agit de *upcast*.

Ce problème nous prévient aussi de cast entre deux traits. Par exemple, pour un message Ping généré par protobuf, il implémente MessageIdentifiable (Teal) et ReflectMessage (prost). Alors quel type devrait demander une fonction qui veut utiliser les deux traits sur un message passé en argument? Idéalement, un des deux traits devrait implémenter l'autre et, à ce moment, on peut l'utiliser pour accéder aux fonctions des deux. C'est pourquoi MessageIdentifiable dérive ReflectMessage dans le projet. Sinon, on peut utiliser un générique. Cette méthode nous permet de faire fonctionner plusieurs librairies ensemble :

```
trait NamedAnimal {
    fn name(&self) -> String;
}
trait LeggedAnimal {
    fn walk(&self) -> String;
}
#[derive(Default)]
struct Dog {
    name: String
}
impl NamedAnimal for Dog {
    fn name(&self) -> String {
        self.name.clone()
```

```
    }
}
impl LeggedAnimal for Dog {
    fn walk(&self) -> String {
        "Walking".to_string()
    }
}
#[test]
fn polymorphism() {
    let mut dog = Dog::default();
    dog.name = "Foo".to_string();
    multityped(&dog);
}
fn multityped<T : NamedAnimal + LeggedAnimal>(t: &T) {
    assert_eq!("Foo".to_string(), t.name());
    assert_eq!("Walking".to_string(), t.walk());
}
```

Conclusion

Le projet fut plaisant à développer. C'est une expérience très satisfaisante que d'apprendre ce nouveau langage complètement différent des autres. Les objectifs principaux ont été atteints, c'est-à-dire l'apprentissage de Rust, l'implémentation de protobuf, de la communication serveur-client, des serveurs HTTP, de la gestion de base de données, du matchmaking et du déploiement sur Docker.

Rust est un langage encore jeune, mais qui fait ses preuves. Il est extrêmement performant comme le C++. Son compilateur est extraordinaire, permettant de découvrir et d'éviter les erreurs mémoires et d'accès concurrents pendant le développement, avant même d'exécuter le programme. Cargo est un des meilleurs gestionnaires de dépendances. La courbe d'apprentissage est toutefois extrêmement abrupte et augmente exponentiellement chaque fois qu'on ajoute le polymorphisme, async et les lifetimes. De plus, les librairies de la communauté ne sont pas aussi complètes et n'ont pas autant de support financier que celles d'autres langages. Il y a également peu d'emplois disponibles pour Rust. Malgré tout, il reste un langage très intéressant pour le futur, lorsque l'écosystème sera plus mature. Un langage système quasi aussi performant que le C++ tout en assurant la sécurité mémoire a un potentiel énorme. On peut suivre notamment l'avancée de Bevy[43], un moteur de jeu FOSS en Rust; ou encore WebAssembly[44] pour la programmation de sites web en Rust plus performants et plus légers qu'en Javascript.

D'autres ressources à suivre:

- https://github.com/UgurcanAkkok/AreWeRustYet
- https://areweguiyet.com
- https://arewegameyet.rs
- https://arewewebyet.org
- https://arewelearningyet.com

Le code du projet est disponible à https://github.com/Souchy/RustProject

[43] https://bevyengine.org
[44] https://rustwasm.github.io/book/introduction.html

Références

Rust

1. Sondage stackoverflow: https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages
2. https://www.reddit.com/r/rust/comments/149cu1k/2023_stack_overflow_survey_rust_is_the_most/
3. Livre Rust: https://doc.rust-lang.org/book/title-page.html
4. Patrons OOP: https://refactoring.guru/design-patterns/rust
5. Rust pour les polyglotes: https://www.chiark.greenend.org.uk/~ianmdlvl/rust-polyglot/ownership.html

Crates:

1. https://crates.io/crates/tokio
2. https://crates.io/crates/prost
3. https://crates.io/crates/prost-reflect
4. https://crates.io/crates/prost-build
5. https://crates.io/crates/prost-reflect-build
6. https://crates.io/crates/tonic
7. https://crates.io/crates/redis
8. https://crates.io/crates/rocket
9. https://crates.io/crates/rocket_okapi
10. https://crates.io/crates/schemars
11. https://crates.io/crates/dotenv
12. https://crates.io/crates/async-trait
13. https://crates.io/crates/rs-snowflake
14. https://crates.io/crates/serde
15. https://crates.io/crates/futures
16. https://crates.io/crates/once_cell
17. https://crates.io/crates/quote
18. https://crates.io/crates/syn
19. https://crates.io/crates/mockall

Redis

1. Transactions: https://redis.io/docs/latest/develop/interact/transactions/
2. Notifications par espace-clé: https://redis.io/docs/latest/develop/use/keyspace-notifications/