École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. (33)2-47-36-14-14
Fax (33)2-47-36-14-22
www.polytech.univ-tours.fr

Rapport

Projet de Programmation et Génie Logiciel

Serveur Rust pour le projet Labyrinthe de Fourmis

Auteur(s)

Teddy Astie
[teddy.astie@etu.univ-tours.fr]

Encadrant(s)

Nicolas Monmarché
[nicolas.monmarche@univ-tours.fr]

Polytech Tours
Département Informatique

Version du 9 janvier 2023

Table des matières

Introduction

Ce projet fait partie de l'ensemble des projets du "jeu de fourmis" proposé par Nicolas Monmarché. Il en constitue notamment l'un des serveurs ici le serveur codé en Rust.

0.1 Rappel du principe de jeu Labyrinthe de Fourmis

Le jeu labyrinthe de fourmis est un jeu coopératif dont l'objectif est d'obtenir le plus de points en apportant de la nourriture jusqu'au nid au travers d'un labyrinthe. Afin de faciliter les décisions des jours, un mécanisme de phéromone est mis en place, afin de déterminer le chemin pris par un joueur qui a réussi à apporter la nourriture jusqu'au nid.

L'environnement de jeu est un labyrinthe sur une grille en 2 dimensions pour lequel chaque case peut comporter éventuellement des mur dans chaque direction ainsi qu'éventuellement de la nouriture et/ou un nid. Le joueur doit trouver un chemin du nid jusqu'à une source de nourriture en évitant les murs, et si possible en prenant le plus court chemin.

Lorsque le joueur trouve de la nourriture, il émet des phéromones sur les cases qu'il traverse, jusqu'à ce qu'il arrive au nid.

Chaque joueur reçoit une copie du labyrinthe lors de son arrivée sur la partie, ainsi que régulièrement (chaque seconde) le niveau de phéromone de chaque case.

Les différentes parties intervenant dans le jeu (client et serveur) communiquent selon un protocole de communication basé sur du JSON et par du TCP/IP.

Le serveur est capable de gérer plusieurs parties en même temps, une phase de connexion/-négociation est effectuée lorsque qu'un client se connecte au serveur, de plus, il est possible pour un joueur de se reconnecter à une partie après un éventuel problème de connexion.

0.2 Intérêt du langage Rust dans le projet

Initialement, les langages proposés furent le C et éventuellement le C++. Toutefois, ce projet sous-entends tout un ensemble de problèmes difficiles à gérer dans ces langages : synchronisation, gestion mémoire, ... ce qui peut être grande source de difficultés (bugs, plantages, ...) en C ou même C++ [1].

1. bien que les standards modernes du type C++20 aident partiellement à résoudre ces problèmes mais au prix d'un code bien plus complexe et bien en dehors du cadre de nos cours de C++

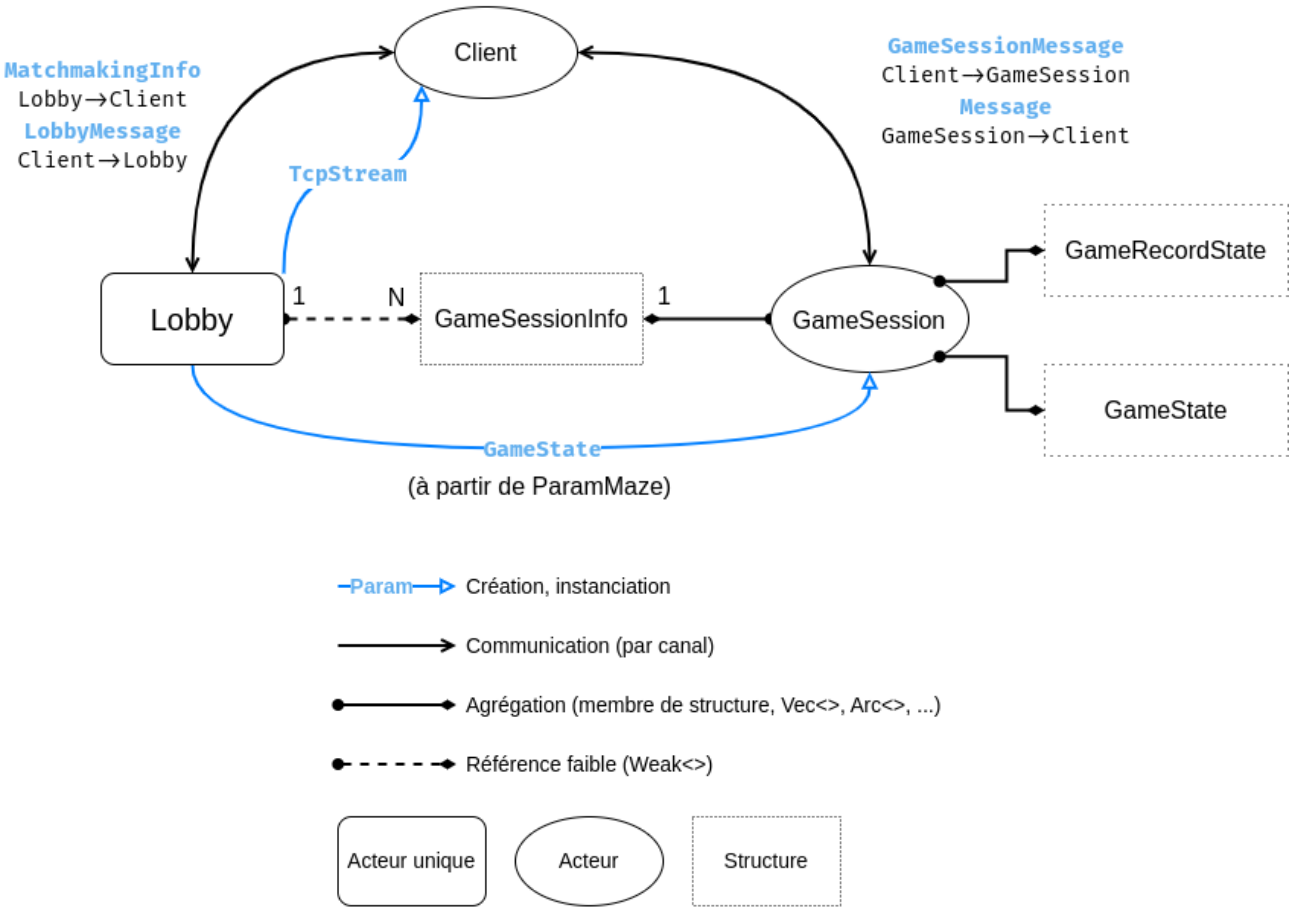
Le langage Rust qui a gagné beaucoup d'intérêt ces dernières années est décrit comme un "langage de programmation fiable, concurrent, pratique" ayant un modèle de concurrence inspiré de l'Erlang [2] et conçu pour produire des logiciels fiables et robustes tout en garantissant une très haute performance. Ce langage a été en grande partie conçu pour éviter une grande partie des problèmes cités au dessus (synchronisation, gestion mémoire) avec des mécanismes particuliers comme par exemple système d'ownership/borrowing strict, le paramètre de durée de vie (non utilisé dans le projet), divers marqueurs, clonage non implicite, etc. De plus, le directeur technique de Microsoft Azure nous recommande d'utiliser Rust à la place de C et C++ pour des nouveaux projets[1].

Pour toutes ces raisons et d'autres encore, le langage Rust a été choisi pour réaliser ce projet.

2. https://www.erlang.org/

Chapitre 1

Structure générale du projet



Organisation générale du projet (non exhaustif)

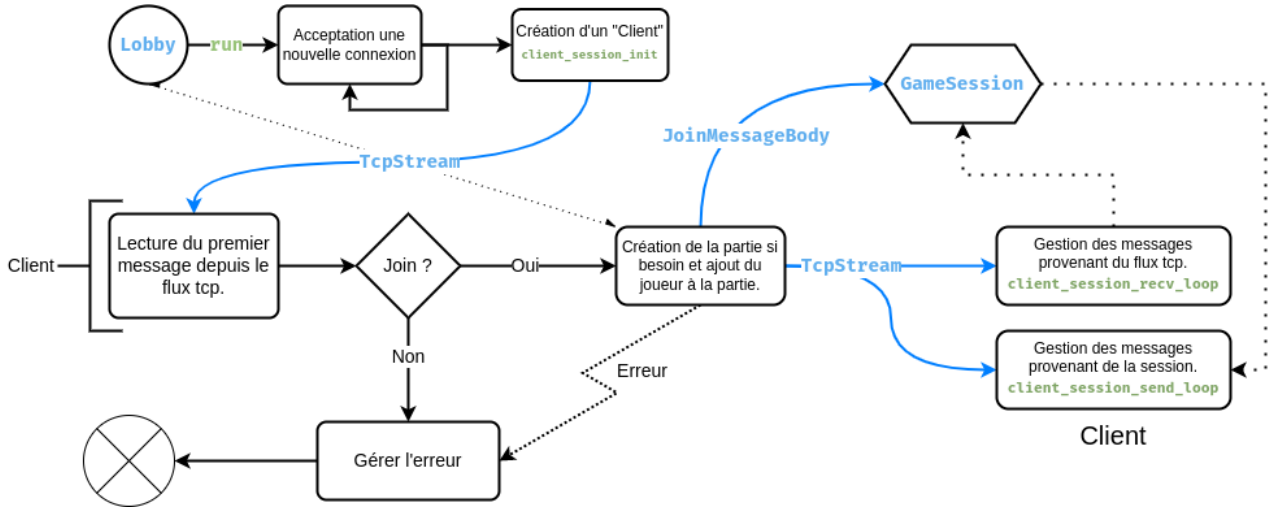
Le projet est divisé en divers composants (ou modules) intéragissants entre eux en utilisant des canaux asynchrones de type Multiple-Producer Single-Consumer disponibles dans la librairie standard Rust depuis le module `syd::sync::mpsc`[1]. Ce modèle s'apparente au modèle d'acteur[2] qui consiste à avoir plusieurs "acteurs" qui communiquent par passage de message, dans notre cas, hormis pour la gestion des `GameSessionInfo` qui se fait par partage mémoire et comptage de référence (`Arc<>` + `Weak<>`), tout le reste communique par passage de message et instanciation de nouveaux acteurs.

Le projet comporte aussi d'autres parties plutôt utilitaires qui ne sont pas affichés dans le diagramme dont l'implémentation du protocole de communication (`message`), l'intéraction avec

1. https://doc.rust-lang.org/std/sync/mpsc/index.html
2. https://fr.wikipedia.org/wiki/Modèle_d'acteur

la librairie de génération de labyrinthe (`external::generator`) ou encore la gestion d'erreur (`error`) et la structure de labyrinthe (`maze`).

Le projet est relativement modulaire, et a été pensé pour être extensible (on peut par exemple ajouter des fonctionnalitées dans les modules en ajoutant de nouveaux types de message acceptés), ainsi qu'en essayant de décupler les objets en sous-objets (en particulier pour `GameSession` découpé en objets plus pratiques pour implémenter certains traits (comme `Serializable` pour `GameState`)), ce qui peut nous permettre par exemple ici d'enregistrer l'état d'une partie dans un fichier.



Flot d'exécution général

Chapitre 2

Présentation des modules clés

2.1 Lobby

L'exécution commence tout d'abord dans le `Lobby` qui a deux responsabilitées.

Tout d'abord, le `Lobby` accepte les différents sockets, et construit des clients.

Au même temps (dans un thread à part), le `Lobby` reçoit des messages de la part des clients pour trouver une partie (*matchmaking*, en créant si besoin la partie), ainsi qu'effectuer d'autres tâches de maintenance des structures de données (supprimer les références vers des parties qui n'existent plus) et gère la liste des parties en cours, des liens entre les `UUID` des joueurs et leur serveur associé.

2.2 Client

Le `client` est construit par le `Lobby` à partir de son `TcpStream`, et d'un canal vers le `Lobby`. La première étape du `client` est de se charger de vérifier et rediriger la négociation d'une partie avec le `Lobby` afin de se connecter à une `GameSession`. Une fois le client connecté à la `GameSession`, ce client se charge de vérifier et rediriger tous les `Message` provenant du `TcpStream` jusqu'à la `GameSession` ainsi que tous les messages de la `GameSession` jusqu'au `TcpStream`.

En cas d'erreur fatale (message inattendu, JSON invalide, erreur réseau, plantage de la parties), envoie si possible une erreur au `TcpStream`, stoppe le `TcpStream` puis detruit le canal du client ce qui va à terme invalider la connexion du client pour la `GameSession`.

2.3 GameSession

L'acteur `GameSession` pilote une `GameState` à partie des messages qu'il reçoit, notamment, il gère une liste de clients connectés, gère leur éventuelle déconnection. Il gère également l'évaporation régulière des niveaux de phéromones pour chaque case ainsi l'envoi régulier des niveaux de phéromones. De plus, il détient la `GameSessionInfo`. Une fonctionnalité expérimentale gérée par le `GameSession` est l'enregistrement automatique des parties par `GameRecordState`.

2.4 GameState

La structure de `GameState` est l'état instantanné de la partie, il détient toutes les informations sur l'état de la partie : le labyrinthe considéré, les niveaux de phéromones, les `UUID` des joueurs, et leur position ainsi que si ils détiennent de la nouriture. Par conséquent, le `GameState`

n'a aucune connaissance de si un joueur concerné est actuellement connecté ou non et détient que des objets bruts ou listes, le rendant notamment compatible avec les traits `Clone`, `Send` et `Serializable`.

En plus de cela, cette structure implémente diverses fonction pour gérer la logique du jeu (déplacements des fourmis, actualisation des niveaux de phéromones, actualisation de l'évaporation, etc.).

2.5 GameRecordState et GameRecord

La structure de `GameRecordState` définit un état d'enregistrement donc l'instant du dernier message (pour calculer les délais), les différents messages depuis le début de la partie, etc., il est utilisé pour construire à terme le `GameRecord` qui est une version figée dans le marbre et `Serializable` de `GameRecordState`.

L'enregistrement d'une partie trace l'intégralité des messages envoyés par les joueurs, le labyrinthe considéré, ainsi que le moment auquel chaque message a été reçu, cela permet de rejouer une partie, un mécanisme expérimental pour rejouer une partie existe dans le module `client::record`.

Chapitre 3

Choix techniques et apports du Rust

3.1 Système concurrent par passage de message

Le choix technique le plus notable dans le projet, c'est d'utiliser une approche concurrente fonctionnant principalement par passage de message à l'inverse d'une approche plutôt orientée objet à mémoire partagée.

Cette approche est particulièrement pratique pour exploiter plusieurs processeurs tout en évitant tout un ensemble de problème de synchronisation que l'on pourrait avoir avec une autre approche. De cette manière, la modélisation obtenue s'apparente au modèle d'acteur [1].

L'approche orientée objet fonctionnant exclusivement par partage mémoire aurait été limitée car chercher à rendre le code parallèle aurait été ardu comme cela aurait impliqué l'utilisation d'un grand nombre de `Mutex` ainsi que d'autres outils de synchronisation donc aura compliqué plusieurs parties du code. Même en négligeant le parralèlisme donc utilise un unique thread, étant donné que l'on souhaite manipuler plusieurs sockets et plusieures sessions de jeu au même moment, on aurait eu d'une manière à implémenter tout un mécanisme de gestion d'I/O asynchrone ou utiliser une librairie pour le faire à notre place, ce qui est loin d'être simple. C'est pourquoi une approche évitant le couplage est plus souhaitable.

Le Rust nous offre un large panel d'outil pour implémenter les deux approches (passage de message et mémoire partagée), ce qui est pratique car on peut suivant ce que l'on souhaite effectuer, on peut choisir une approche au lieu d'une autre. Dans notre cas, l'ensemble du projet fonctionne par passage de message à l'exception des informations des parties qui est partagée entre le `Lobby` et la `GameSession` (afin notamment de pouvoir déterminer si la partie concernée est en cours ou non, à l'aide de la validité de la référence que détient le `Lobby` vers cette structure).

Il est utile de noter que la plupart des librairies asynchrones Rust [2] favorisent une approche par passage de message avec par exemple des versions adaptées de `std::sync::mpsc`.

3.2 Garanties de fiabilité

Le langage Rust met en oeuvre tout un ensemble de mécanisme ayant des implications sur le code pour éviter les fuites mémoires sans pour autant nécessiter un garbage-collector (bien qu'il soit possible d'en obtenir avec des références cycliques de `Rc<T>`), d'accéder à des références invalides, certains problèmes liés à l'aliasing, les *race conditions*, ...Cela peut impliquer l'utilisation de types particulier pour encapsuler nos objets si l'on veut permettre à l'objet d'être référencé à plusieurs endroits (`Rc<T>` et `Arc<T>` similaires à `std::shared_ptr` en C++) ou encore si l'objet peut être modifié par plusieurs thread (`Mutex<T>`), etc...

1. https://fr.wikipedia.org/wiki/Modèle_d'acteur
2. async-std, tokio

Cela prend notamment la forme de *traits* (≈ interface) spécifiques indiquant si l'objet peut être transféré dans un autre thread ou encore partagée entre plusieurs threads [3].

De cette manière, un grand nombre d'éventuels problèmes de synchronisation ou de gestion mémoire sont évités ou simplifiés. Toutefois, ce fonctionnement est assez unique et pas très habituel dans d'autres langages de programmation, ce qui peut rendre compliqué l'apprentissage du Rust aux développeurs qui ne sont pas habitués à ces concepts.

3.3 Traitement des messages à l'aide des *tagged union* et du *pattern matching*

L'une des fonctionnalitées du langage Rust les plus pratiques pour implémenter le traitement des messages du protocole utilisé dans le jeu mais aussi ceux pour le passage de message est le pattern matching couplé aux *tagged unions* [4] [5]. Cet outil nous permet de créer des *types somme* qui est une sorte d'énumération (plus précisément, une version améliorée des énumérations), mais dont chaque variant peut avoir des valeurs ou non (sous la forme d'une valeur, tuple, structure, ...). On peut par exemple définir un union taggué pouvant recevoir chaque type de message supporté par le protocole ainsi que leurs valeurs.

Cette fonctionnalité est également très utilisée dans les fonctionnalitées de base du langage, nottament `Option<T>` [6], `Result<T, E>` [7](gestion d'erreur), et bien d'autres.

```rust
pub enum Message {
    Join(JoinMessageBody),
    OkMaze(OkMazeMessageBody),
    Info(InfoMessageBody),
    Error(ServerError),
    Move(MoveMessageBody),
    Unexpected {
        expected: Vec<Box<str>>,
        received: Box<Message>,
    },
}
```

Enumeration pour les messages du protocole de communication

En utilisant `match`, il est possible de prendre une décision pour chaque variant de l'union taggué/enumération, tout en destructurant chaque variant, ce qui permet de facilement utiliser ces objets. Il est également possible de faire du "pattern matching" à l'aide de match y compris sur l'élément à l'intérieur du match.

3. https://doc.rust-lang.org/nomicon/send-and-sync.html
4. https://en.wikipedia.org/wiki/Tagged_union
5. https://doc.rust-lang.org/book/ch06-00-enums.html
6. https://doc.rust-lang.org/core/option/index.html
7. https://doc.rust-lang.org/core/result/index.html

```
match msg {
    Message::Join(body) => { /* ... */ }
    Message::OkMaze(body) => { /* ... */ }
    Message::Info(body) => { /* ... */ }
    Message::Move(body) => { /* ... */ }
    _ => { /* ... */ }
}
```

Utilisation de match sur un message suivant son type

Cette fonctionnalité est extensivement utilisée pour le traitement de message, que ce soit provenant d'un client distant (socket), ou pour le passage de messages entre les acteurs. Il existe une alternative dans le cas où un seul cas particulier nous intéresse (`if let`[8])

3.4 Sérialisation de données à l'aide de serde et serde-json

Serde[9] est un framework de sérialisation de donnéees Rust (ce framework fait parties de https://blessed.rs), il permet de rendre des types `Serialize` et `Deserialize` de manière automatiques (en utilisant `derive`) ou non ainsi que personaliser la manière dont est structuré la donnée. En complément à cette librairie est peut être implémenté une librairie pour sérialiser et désérialiser différents format de sérialisation, en particulier pour notre projet le format JSON à l'aide de `serde-json`[10].

De cette manière, en utilisant , on peut très facilement sérialiser ainsi que désérialiser l'ensemble des messages que peut envoyer ou recevoir le serveur, ce qui s'avère extrêmement pratique pour implémenter le protocole.

De plus, cela permet de facilement changer le format de sérialisation si nécessaire sans impliquer de grands changements dans le reste code.

3.5 Gestion (quasi-)systématique des erreurs

Certains langages de programmation, dont le C++ et le Java gèrent les erreurs par un mécanisme d'exception, d'autres comme le C utilisent une valeur de retour d'erreur pour indiquer d'une erreur est survenue.

Le langage Rust utilise principalement[11] une valeur de retour pour indiquer une erreur. Cela prend la forme d'un type `Result<T, E>`[12] qui indique soit une valeur valide, soit une erreur et d'un autre type `Option<T>`[13] pour des fonctions qui pourrait ne pas retourner de valeur. Un `Result<T, E>` doit être obligatoirement utilisé, par exemple en utilisant `match` pour définir le comportement en cas d'erreur ou avec une valeur, il est également possible d'utiliser l'opérateur

8. https://doc.rust-lang.org/book/ch06-03-if-let.html
9. https://serde.rs/
10. https://crates.io/crates/serde_json
11. bien qu'il existe un autre mécanisme de gestion d'erreur similaire aux exceptions pour des erreurs critiques https://doc.rust-lang.org/book/ch09-01-unrecoverable-errors-with-panic.html
12. https://doc.rust-lang.org/core/result/
13. https://doc.rust-lang.org/core/option/

`?`[14] pour remonter l'erreur comme valeur de retour à la fonction, ce qui permet d'une manière, "d'émuler des exceptions".

De plus, de la même manière que pour définir des exceptions, on peut définir nos propres types d'erreurs, et ainsi "normaliser" les erreurs que l'on peut avoir dans le projet. En effet, dans notre projet, l'objet `ServerError` qui est utilisé à cet fin implémente `Serializable`, ce qui a permis de facilement mettre en place un mécanisme de transmission des erreurs au client.

Le système d'exception du Rust est très pratique, car il nous empêche d'ignorer silencieusement les erreurs (et par conséquent déclencher des comportements indéfinis) rend les origines des erreurs plus claires (l'opérateur `?` indique que l'opération gère son erreur en la remontant), seules les fonctions retournant `Result<T, E>` peuvent échouer, permet de gérer imédiatement les erreurs, sans nécessiter d'écrire de bloc `try catch`, etc.

3.6 Riche ensemble d'outil pour développer le projet

Le Rust fournit un conséquent panel d'outil pour développer le projet[15], en allant d'une installation simplifiée[16], en passant par le confort dans l'utilisation avec un éditeur[17] ainsi que plusieurs outils liés aux pratiques de génie logiciel (compilation et gestion des librairies[18], génération de pages de documentation comme avec Doxygen[19], vérification du code[20], conventions d'écriture[21], tests unitaires[22], sans oublier l'aide fournie le compilateur[23]).

Cet ensemble d'outil est consistent et pratique sans nécessiter de lourde configuration. C'est une partie de ce qui fait le succès du langage d'après une analyse de The Overflow [StackOverflow][2][3].

Une grande partie de ces outils ont été utilisés pour réaliser ce projet. Cela participe à l'amélioration de la qualité du code, à la fiabilité ainsi qu'à la performance dans certains cas. Cela permet également de mettre en pratique certaines pratiques de génie logiciel.

14. https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html
15. https://www.rust-lang.org/fr/tools
16. https://rustup.rs/
17. https://rust-analyzer.github.io/
18. https://doc.rust-lang.org/cargo/index.html
19. https://doc.rust-lang.org/rustdoc/index.html
20. https://doc.rust-lang.org/stable/clippy/index.html
21. https://rust-lang.github.io/rustfmt/
22. https://doc.rust-lang.org/book/ch11-00-testing.html
23. https://youtu.be/CJtvnepMVAU

Chapitre 4

Organisation du projet

4.1 Gestion du projet Rust

Etant donné que le projet a été développé uniquement par moi même, il n'y a pas grand chose à noter quant à la gestion du projet. Une chose importante tout de même que j'ai pu remarquer est l'importance de régulièrement travailler sur le projet, afin d'éviter de tout faire à la fin. Dans mon cas, et comme le montre le dépôt git, le projet a été régulièrement et au fur et à mesure développé, toutefois j'ai remarqué que certains binômes de d'autres projets de génie logiciel ont eu des difficultées à gérer leur temps. Il faut donc rester sur cette approche de travail régulier pour les projets à venir.

4.2 Gestion du méta-projet Labyrinthe de Fourmis

Notre sujet, malgré que ce soit un projet effectué en binôme (ou monôme), était développé en partie en collaboration avec les autres groupes afin nos projets soient compatibles dans la mesure du possible. De cette manière, plusieurs réunions ont été organisées, afin de décider comment implémenter le projet, définir le protocole, la forme du labyrinthe, et ensuite définir de quelle manière collaborer pour ce projet, etc.

Toutefois, assez rapidement, certains groupes ont perdu le contact avec le projet, ce qui a compliqué la collaboration, et a entraîné à terme des divergences dans ce qui a été implementé [1]. De plus, cela complique l'accès à certains outils, comme par exemples les joueurs pour tester nos serveurs, et ainsi vérifier que nos implémentations du protocoles sont correctes (dans les deux sens). Pour les projets futurs [2], il faudra être très attentif sur les autres personnes du groupe quant à la cohésion et insister plus encore que ce qui a été effectué pour ce méta-projet à ce que les autres participants ne perdent pas le fil.

Pour autant, une grande partie des projets [3] était hébergée sur le Gitlab de l'université [4], afin de faciliter la collaboration entre les projets. Par exemple, j'ai pu intégrer la librairie C++ à mon projet à l'aide d'un submodule git [5] ce qui s'avère pratique pour intégrer la compilation de la librairie C++ dans mon projet (malgré d'éventuelles difficultées liés à la librairie standard C++ qui n'est pas intégrée de base à la compilation Rust).

1. en particulier la divergence dans le protocole implémenté entre les serveurs Rust et C++
2. dont le projet collectif de S8
3. notamment le code source des serveurs et des générateurs de labyrinthes utiles aux serveurs
4. https://scm.univ-tours.fr
5. https://git-scm.com/book/en/v2/Git-Tools-Submodules

Conclusion

Tout d'abord, ce projet m'a permis de mettre en perspective toute la problèmatique de gestion de projet. En effet, ce projet a cherché à faire coopérer plusieurs groupes, autour d'un même plus grand projet, ce qui peut rapidement devenir compliqué, par exemple, il peut arriver que certains groupes "partent dans leur coin" et développent leur partie du projet de leur côté donc sans trop coopérer avec les autres. Cela explique l'importance d'effectuer des réunions afin de mettre au clair ce que l'on souhaite effectuer afin d'éviter des divergences ou incompréhensions dans ce qui est effectué.

Il est entièrement possible que les autres groupes du projet aient estimé qu'il s'agissait d'un projet plutôt "individuel" et donc ont involontairement préféré ne pas coopérer avec les autres, bien que ce ne soit pas l'esprit du projet.

Ensuite, ce projet a consitué un véritable exercice concrêt de programmation en Rust, ce qui m'a permis d'apprendre plus en pronfondeur le langage, ainsi que son écosystème. Bien que ce soit pas mon tout premier projet (même petit) en Rust, cela a été le plus grand et important que j'ai effectué depuis que j'ai commencé à apprendre le langage (environ depuis 3 ans). J'ai pu expérimenter avec de nombreuses parties de la librairie standard Rust, toutes les subtilitées de l'interfaçage avec le C, la sérialisation, bien d'autres choses mais surtout la programmation concurrente par passage de message.

Depuis le début de cette année universitaire, le langage Rust a gagné énormément d'intérêt dans de nombreux domaines, par exemple, il est desormais officiellement supporté par le noyau Linux[4] avec un certain succès (driver open-source pour le GPU de l'Apple M1[5], OpenCL dans Mesa3D[6], et plus encore[7]). De plus, le Rust intéresse de plus en plus les entreprises (CEA, Thales, AtoS, ...) et intéresse aussi bien le domaine de l'embarqué, que celui des services web, serveurs, librairies système, etc.

Etant donné que mon expérience avec ce langage fut extrêmement positive, je n'hésiterais pas à effectuer un stage autour du langage Rust, ce qui est de plus en plus proposé, et que les candidats pour ce genre de stage se font rares [6].

6. hormis pour un stage promosé par AtoS explicitément prévus pour des novices du Rust qui intéresse beaucoup de candidats

Bibliographie

[1] https://www.zdnet.com/article/programming-languages-its-time-to-stop-using-c-and-c-for

[2] https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/

[3] https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/

[4] https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

[5] https://asahilinux.org/2022/11/tales-of-the-m1-gpu/

[6] https://www.phoronix.com/news/Rusticl-2022-XDC-State

[7] https://lpc.events/event/16/contributions/1180/attachments/1017/1961/deck.pdf

Annexe A

Liens utiles

Voici une petite liste d'url intéressantes au sujet de ce projet :

— Gitlab du projet https://scm.univ-tours.fr/21906867t/serveur-fourmi
— Site officiel du Rust https://www.rust-lang.org/fr/
— Référentiel de documentation Rust https://www.rust-lang.org/fr/learn
— Référence en magie noire en Rust https://doc.rust-lang.org/nomicon/
— Documentation de serde https://serde.rs

Serveur Rust pour le projet Labyrinthe de Fourmis

Rapport : Projet de Programmation et Génie Logiciel

Résumé : Le méta-projet de jeu de fourmis est un jeu de simulation de fourmis multijoueur utilisant une architecture Client-Serveur. Ce projet propose une implémentation du serveur dans le langage Rust, et permet ainsi de mesurer l'intérêt de ce langage pour ce projet. Il fait partie avec le projet C++ des 2 implémentations du serveur pour ce jeu.

Mots clé : Rust, Simulation, Fourmis, Programmation Concurrente, Programmation Fonctionelle, Multithreading, Réseau, Sérialisation, Jeu, TCP

Abstract : The meta-project of "ant simulation game" is a multiplayer ant simulation serious-game based on a Client-Server architecture. This project offer a Rust implementation for the server, and highlights the benefits of this programming language in this project. It is along with the C++ project one of the implementations of the server for this game.

Keywords : Rust, Simulation, Ant, Concurrent Programming, Functionnal Programming, Multithreading, Network, Serialization, Game, TCP

Auteur(s)

Teddy Astie
[teddy.astie@etu.univ-tours.fr**]**

Encadrant(s)

Nicolas Monmarché
[nicolas.monmarche@univ-tours.fr**]**

Polytech Tours
Département Informatique

Ce document a été formaté selon le format EPUProjetDi.cls (N. Monmarché)

École Polytechnique de l'Université de Tours
64 Avenue Jean Portalis, 37200 Tours, France
http://www.polytech.univ-tours.fr