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 6 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++

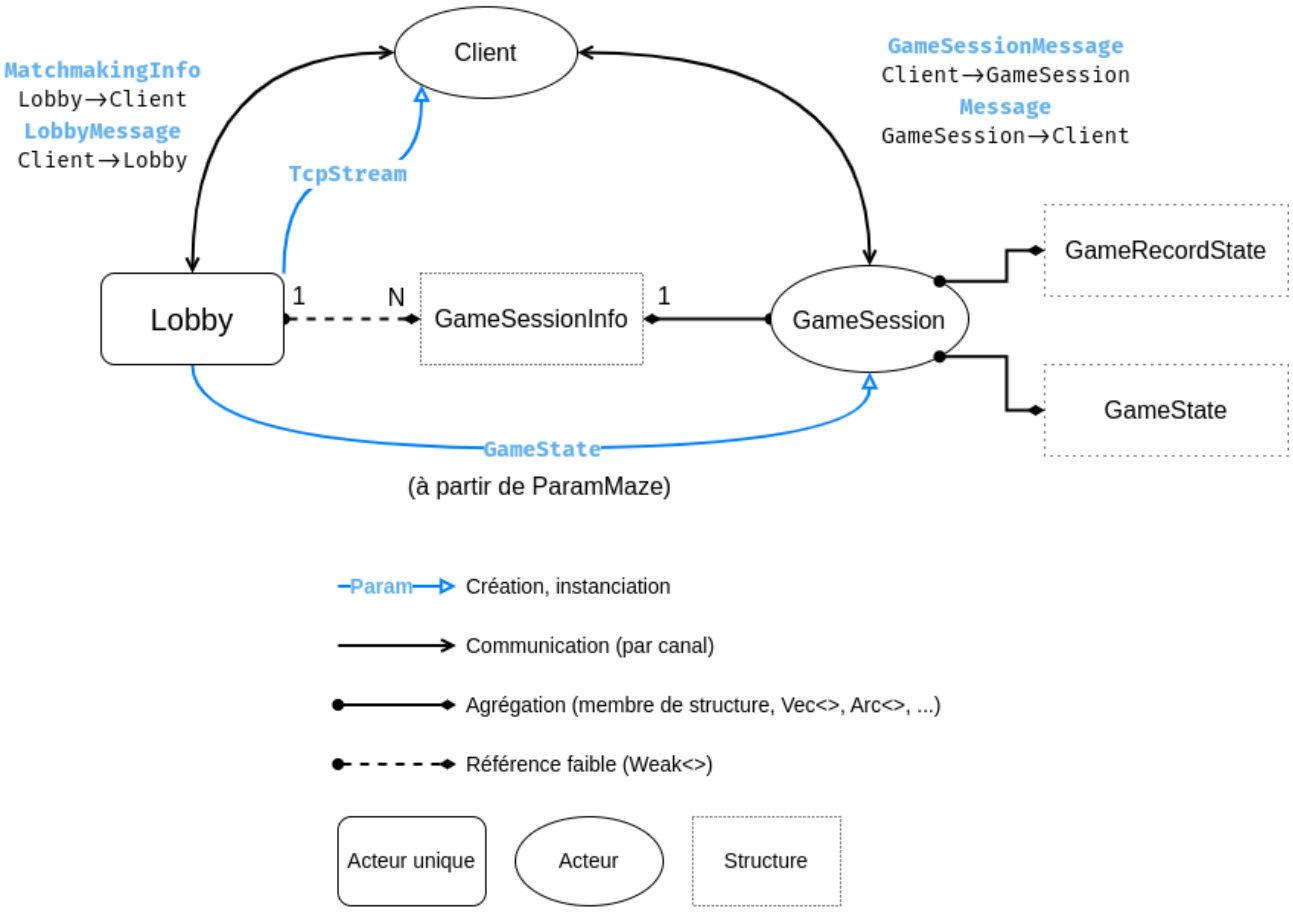
Le langage Rust qui a gagné beaucoup d'intérêt ces dernières années est décrit comme un "langage de programmation fiable, concurrent, pratique" ayant un modèle de concurrence inspiré de l'Erlang [2] et conçu pour produire des logiciels fiables et robustes tout en garantissant une très haute performance. 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)

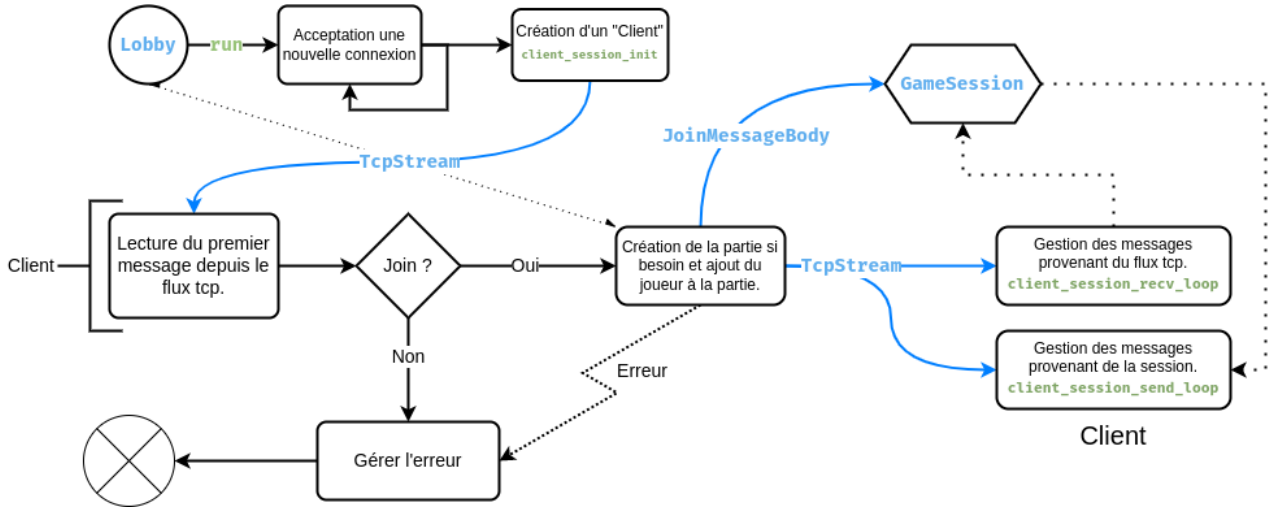
Le projet est divisé en divers composants (ou modules) intéragissants entre eux en utilisant des canaux asynchrones de type Multiple-Producer Single-Consumer disponibles dans la librairie standard Rust depuis le module `syd::sync::mpsc`[1]. Ce modèle s'apparente au modèle d'acteur[2] qui consiste à avoir 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`.

Conclusion

Bibliographie

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

[2] détail bibliographique de la ref2

[3] détail bibliographique de la ref3

[4] détail bibliographique de la ref4

[5] détail bibliographique de la ref5

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