



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](http://www.polytech.univ-tours.fr)

## Rapport

### Projet de Programmation et Génie Logiciel

# Serveur Rust pour le projet Labyrinthe de Fourmis

Auteur(s)

**Teddy Astie**

[\[teddy.astie@etu.univ-tours.fr\]](mailto:teddy.astie@etu.univ-tours.fr)

Encadrant(s)

**Nicolas Monmarché**

[\[nicolas.monmarche@univ-tours.fr\]](mailto:nicolas.monmarche@univ-tours.fr)

**Polytech Tours**  
**Département Informatique**

# Table des matières

---

<b>Introduction</b>	<b>1</b>
0.1 Rappel du principe de jeu Labyrinthe de Fourmis . . . . .	1
0.2 Intérêt du langage Rust dans le projet . . . . .	1
<b>1 Structure générale du projet</b>	<b>3</b>
<b>2 Présentation des modules clés</b>	<b>5</b>
2.1 Lobby . . . . .	5
2.2 Client . . . . .	5
2.3 GameSession . . . . .	5
2.4 GameState . . . . .	5
2.5 GameRecordState et GameRecord . . . . .	6
<b>3 Choix techniques et apports du Rust</b>	<b>7</b>
3.1 Système concurrent par passage de message . . . . .	7
3.2 Garanties de fiabilité . . . . .	7
3.3 Traitement des messages à l'aide des <i>tagged union</i> et du <i>pattern matching</i> . . . .	8
3.4 Sérialisation de données à l'aide de <i>serde</i> et <i>serde-json</i> . . . . .	9
<b>Conclusion</b>	<b>10</b>
<b>Annexes</b>	<b>11</b>
<b>A Liens utiles</b>	<b>12</b>

# 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 joueurs, 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 murs dans chaque direction ainsi qu'éventuellement de la nourriture 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-entend 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++<sup>1</sup>.

---

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<sup>2</sup> 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].

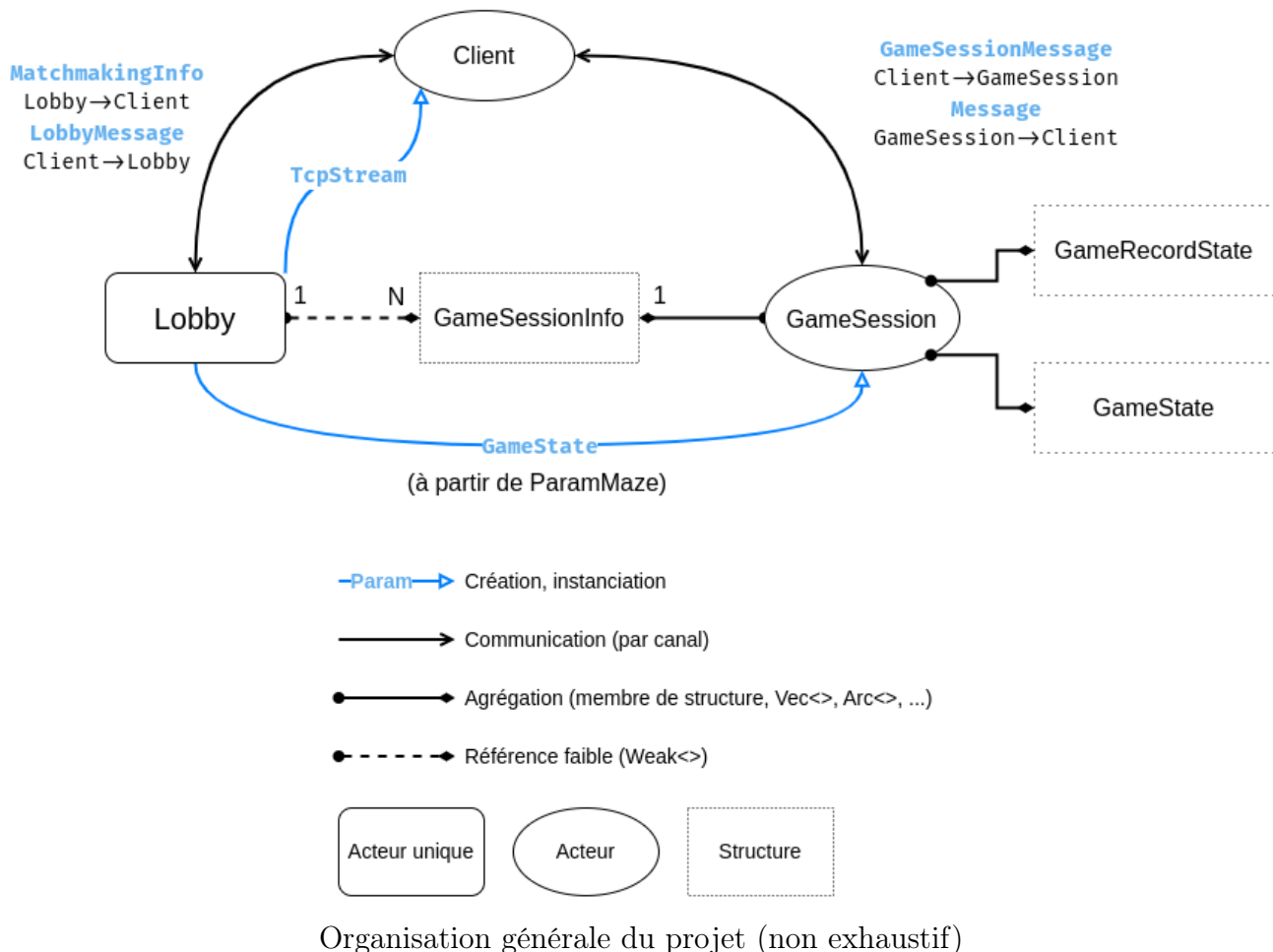
Pour toutes ces raisons et d'autres encore, le langage Rust a été choisi pour réaliser ce projet.

---

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

# Chapitre 1

## Structure générale du projet



Le projet est divisé en divers composants (ou modules) interagissant entre eux en utilisant des canaux asynchrones de type Multiple-Producer Single-Consumer disponibles dans la bibliothèque standard Rust depuis le module `std::sync::mpsc`<sup>1</sup>. Ce modèle s'apparente au modèle d'acteur<sup>2</sup> 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 instantiation de nouveaux acteurs.

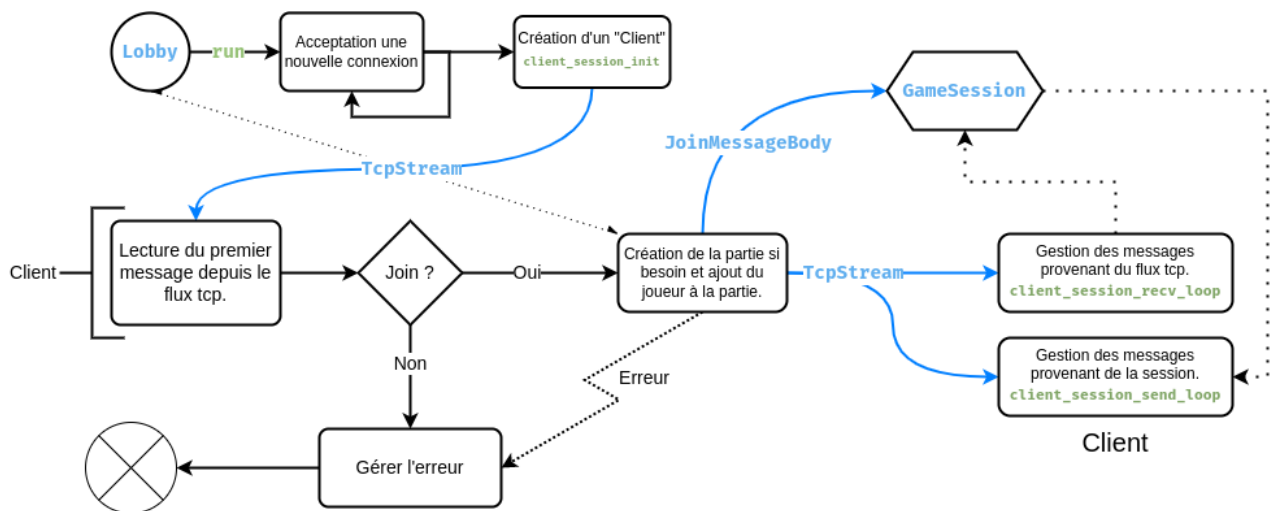
Le projet comporte aussi d'autres parties plutôt utilitaires qui ne sont pas affichés dans le diagramme dont l'implémentation du protocole de communication (`message`), l'interaction avec

1. <https://doc.rust-lang.org/std/sync/mpsc/index.html>

2. [https://fr.wikipedia.org/wiki/Modèle\\_d'acteur](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 fonctionnalités dans les modules en ajoutant de nouveaux types de message acceptés), ainsi qu'en essayant de découpler 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 responsabilités.

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 détruit 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 partir 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 instantané 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 nourriture. 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`.



# 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 <sup>1</sup>.

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 parallélisme donc utilise un unique thread, étant donné que l'on souhaite manipuler plusieurs sockets et plusieurs 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 <sup>2</sup> 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](https://fr.wikipedia.org/wiki/Modèle_d'acteur)

2. `async-std`, `tokio`

Cela prend notamment la forme de *traits* ( $\approx$  interface) spécifiques indiquant si l'objet peut être transféré dans un autre thread ou encore partagée entre plusieurs threads<sup>3</sup>.

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 fonctionnalités 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*<sup>4 5</sup>. 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 fonctionnalités de base du langage, notamment `Option<T>`<sup>6</sup>, `Result<T, E>`<sup>7</sup> (gestion d'erreur), et bien d'autres.

```
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/énumération, 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](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 fonctionnalit  est extensivement utilis e 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 o  un seul cas particulier nous int resse (`if let`<sup>8</sup>)

### 3.4 S rialisation de donn es   l'aide de serde et serde-json

Serde<sup>9</sup> est un framework de s rialisation de donn ees Rust (ce framework fait parties de <https://blessed.rs>), il permet de rendre des types `Serialize` et `Deserialize` de mani re automatiques (en utilisant `derive`) ou non ainsi que personnaliser la mani re dont est structur  la donn e. En compl ment   cette librairie est peut  tre impl ment  une librairie pour s rialiser et d s rialiser diff rents format de s rialisation, en particulier pour notre projet le format JSON   l'aide de `serde-json`<sup>10</sup>.

De cette mani re, en utilisant , on peut tr s facilement s rialiser ainsi que d s rialiser l'ensemble des messages que peut envoyer ou recevoir le serveur, ce qui s'av re extr mement pratique pour impl menter le protocole.

De plus, cela permet de facilement changer le format de s rialisation si n cessaire sans impliquer de grands changements dans le reste code.

8. <https://doc.rust-lang.org/book/ch06-03-if-let.html>

9. <https://serde.rs/>

10. [https://crates.io/crates/serde\\_json](https://crates.io/crates/serde_json)

# 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 Fonctionnelle, 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, Fonctionnal Programming, Multithreading, Network, Serialization, Game, TCP

Auteur(s)

**Teddy Astie**

[[teddy.astie@etu.univ-tours.fr](mailto:teddy.astie@etu.univ-tours.fr)]

Encadrant(s)

**Nicolas Monmarché**

[[nicolas.monmarche@univ-tours.fr](mailto: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>