Daemon de Contrôle Distribué : Rapport de Projet

Fieux Telmo, Fort Alexandre, Lagier Hadrien, Maati Mohamed-Yâ-Sîn
Université de Toulouse
Dirigé par : Poquet Millian

Mai 2025

1 Introduction

1.1 Présentation du projet

L'expérimentation réelle est une méthode essentielle pour l'étude des systèmes et applications distribuées en informatique. Elle consiste à exécuter de manière contrôlée de vraies applications distribuées sur des systèmes réels, permettant ainsi d'observer et d'analyser leur comportement afin d'en extraire des connaissances.

Bien que de nombreux outils permettent de lancer des programmes à distance (`sshd`, peu offrent la robustesse, la remontée d'erreurs efficace, et la faible interférence nécessaires pour des expériences scientifiques rigoureuses.

1.2 Objectifs

Ce projet vise à développer une application distribuée **légère**, **robuste** et **asynchrone** conçue pour **exécuter et contrôler des processus** à travers un réseau de machines. L'application sera **implémentée** en **Rust** et en **C**, avec des comparaisons de performances et de conception entre les deux langages afin de mieux répondre aux exigences du projet.

Nous appelons cette application un *Daemon de Contrôle*. Elle suit une **architecture basée sur des démons**, dans laquelle un processus de longue durée fonctionne sur chaque machine, offrant une interface de contrôle à distance. La communication entre les composants est assurée via le **protocole TCP**, et les commandes sont échangées à l'aide d'un mécanisme de **sérialisation inter-langages**.

Pour l'implémentation asynchrone en Rust, nous utilisons la bibliothèque Tokio, qui fournit des outils puissants pour la concurrence et les entrées/sorties non bloquantes.

2 Calendrier du Projet

2.1 Étude de Faisabilité

La phase initiale s'est concentrée sur la conception et la manière d'implémenter le démon. Nous avons exploré diverses API de Rust et des fonctions système telles que `clone` vs `fork`, et `poll` vs `epoll`. Nous avons ensuite commencé à isoler et tester des fonctionnalités individuelles du démon, ainsi qu'à **prototyper des méthodes de sérialisation de données** pour la **communication inter-langages**, en comparant des options comme FlatBuffers et Serde.

2.1.1 Réflexion à la fin de cette première étape

A la fin de cette première partie du projet nous nous sommes rendu compte de plusieurs choses. On a conclu que :

- `epoll` est mieux que `poll` car il permet d'ajouter et de retirer facilement des files descriptors. Par la même occasion il permet d'attacher à un file descriptor certaines données, notamment pour identifier son type avant de le lire.
- Les primitives systèmes (comme `epoll`) s'est avérées trop complexe pour la partie Rust. On a priviligié les libs comme `tokio, netstat2, inotify`.
- `Flatbuffers` seul suffit à sérialiser des messages d'erreurs de manière relativement simple, nous ne pensons pas que Serde ait une utilité pour notre utilisation.

Cependant, notre premier prototype de sérialisation présentait plusieurs erreurs ainsi que des incompréhensions de notre part. Les échanges avec notre directeur d'étude nous ont permis de clarifier cette partie du projet et d'en améliorer la conception.

2.2 Implémentation du Démon

Durant cette phase, nous nous sommes concentrés sur la **construction d'une version complète et fonctionnelle du démon**, en Rust et en C, intégrant toutes les fonctionnalités clés. L'objectif était de garantir que chaque implémentation dans chaque langage offre les mêmes capacités tout en restant compatibles.

Une étape importante a été d'établir le contrôle à distance du démon depuis un processus externe. Pour cela, nous avons implémenté une couche de communication utilisant un protocole réseau (TCP) combiné à une sérialisation inter-langages. Cela a permis aux composants en Rust et en C d'échanger des données structurées de manière fluide, en utilisant un format efficace et facile à analyser.

Nous avons également commencé à créer une **suite de tests de régression**. Cette suite permet de vérifier que toute nouvelle modification ne casse pas les fonctionnalités existantes, assurant ainsi la stabilité au fur et à mesure que la base de code évolue.

Notre démon sera capable :

- Commandes simple (Unix-like) : *echo*, *sleep* etc.
- Des surveillances sur un **socket** ou sur un **fichier** avec *inotify*.

2.2.1 Détails

Voici en pseudo-code comment fonctionne notre démon :

```
async fn main() {
    listener <- bind(adress, port)
    loop {
        socket, address <- listener.accept()
        send_on_socket(established_connection)
        async clone() {
            loop {
                buff <- socket.read()
                handle_message(buff)
            }
        }
    }
}

async fn handle_message(buff) {
    match buff.type {
        KillProcess => // example
        RunCommand => {
            if clone() != fail {
                send_on_socket(process_launched)
                execve(buff.command)
                if execve == fail {
                    send_on_socket(execve_terminated(fail))
                }
                else {
                    send_on_socket(execve_terminated(succed))
                }
            }
            else {
                send_on_socket(child_creation_error)
            }
```

```
            handle_surveillance_event(buffer.to_watch)
            send_on_socket(process_terminated)
        }
    }
}

async fn handle_surveillance_event(surveillance_event) {
    match surveillance_event.type {
        Inotify => // handle for Inotify events
        TCPSocket => // handle for TCPSocket events
    }
}
```

2.3 Analyse des Performances

À ce stade, nous avons entièrement intégré le démon dans un système. Nous avons veillé à ce que toutes les fonctionnalités majeures soient couvertes par les tests de régression.

Pour comprendre l'efficacité du démon, nous avons conçu et mis en œuvre des outils automatisés de test de performance. Ces outils simulaient des scénarios d'utilisation réels et mesuraient les ressources système consommées par le démon sous différentes charges.

Nous avons ensuite recueilli des données de performance telles que la latence, l'utilisation du processeur, et mené une analyse détaillée des résultats (cf. 3). Cela nous a aidés à comprendre les compromis entre l'utilisation de Rust et de C en termes d'efficacité à l'exécution et à choisir l'un des deux.

3 Analyse

4 Conclusion