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

Contents

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.

1.3 Répartition des tâches

On c'est répartie les tâches de la manière suivante :

2 État de l'art et veille technologique

Table 1: Comparaison des technologies pour envoyer des données sérialisées à un démon de contrôle

Technologie	**Avantages**	**Inconvénients / Problèmes**
Script Python + `netcat`	- Simple à mettre en place - Permet l'envoi de données brutes ou sérialisées (ex. JSON) - Utilisable localement ou à distance via TCP	- Pas de gestion de protocole - Risque de perte de données si connexion instable - Nécessite une gestion explicite du découpage des messages
Utilisation de `SSH`	- Communication sécurisée par chiffrement - Accès distant fiable - Permet d'exécuter des commandes à distance - Facile à intégrer dans des scripts existants	- Plus lourd à configurer (clé SSH, authentification) - Moins direct pour communiquer en temps réel - Pas conçu pour un échange de messages continu ou en temps réel
Named Pipes (FIFO)	- Très simple pour communication locale - Pas besoin de protocole complexe - Faible latence	- Limitée à la communication locale - Peut bloquer si le pipe n'est pas bien géré - Pas de sécurité intrinsèque
Unix Domain Sockets	- Communication locale performante et bidirectionnelle - Plus flexible que les pipes - Supporte les connexions multiples	- Limité à la machine locale - Nécessite un peu plus de programmation pour gérer les connexions
TCP Socket (avec protocole simple)	- Fonctionne localement et à distance - Permet d'envoyer des données sérialisées dans un protocole défini - Facile à tester avec des outils comme `netcat` ou `telnet`	- Nécessite d'implémenter un protocole de découpage et gestion d'erreurs - Pas sécurisé par défaut (besoin de TLS pour sécuriser)

On se rend compte que la technologie des **sockets** constitue une bonne solution, à condition de mettre en place une gestion d'erreurs rigoureuse ainsi qu'un protocole de découpage efficace. Les types algébriques de Rust semblent être un outil idéal pour assurer le bon fonctionnement du système. Et dans un futur (pas démandé dans ce projet) une sécurité dans le transport des données.

2.1 Outils et technologies comparés

Il existe une multitude d'outils de sérialisation. L'étude ... nous a montré que flatbuffer semble être le plus ...

2.2 Limites identifiées et opportunités

3 Organisation et conception des modules logiciels

3.1 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.

3.1.1 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.

3.2 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*.

3.2.1 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
    }
}
```

3.3 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. 4). 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.

3.3.1 Détails

Grâce au framework de tests, `pytest` nous avons pu créer des fixtures permettant de réutiliser le code dans plusieurs fonctions de tests. Par exemple pour lancer le démon une fixture s'impose pour pas réécrire le code plusieurs fois.

Prototype de tests en pseudo-code :

```
fn test_sleep(daemon) {
    connect(port)
    send(serialize_command("sleep 3"))
    wait_received_process_launched()
    start_timer()
    wait_received_process_terminated()
    stop_timer()
    assert timer +- 100 == 3
}
```

Dans ce test le paramètre `daemon` est une **fixture** de notre jeu de tests, il lancera automatiquement le démon sur un `port` précisé en amont. Ici, on vérifie si une attente de 3 secondes a été détectée avec une marge d'erreur de ε **seconde(s)** (100 dans ce cas).

4 Analyse

5 Conclusion