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 **3**
1.1 Présentation du projet . . . 3
1.2 Objectifs . . . 3

2 État de l'art et veille technologique **3**
2.1 Outils et technologies comparés . . . 3
2.1.1 Sérialisation des données . . . 3
2.1.2 Technologies réseau . . . 4

3 Conception des modules logiciels **4**
3.1 Étude de Faisabilité . . . 4
3.1.1 Réflexion à la fin de cette première étape . . . 5
3.2 Implémentation du Démon . . . 5
3.2.1 Détails . . . 6
3.2.2 Détail, utilisation de epoll dans le daemon C . . . 7
3.2.3 Implémentation de la sérialisation . . . 7
3.3 Analyse des Performances . . . 8
3.3.1 Détails . . . 9

4 Gestion du projet **9**
4.1 Partage des tâches . . . 9
4.2 Bilan personnel . . . 9

5 Analyse **9**

6 Conclusion **10**

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 État de l'art et veille technologique

2.1 Outils et technologies comparés

2.1.1 Sérialisation des données

La sérialisation des données est essentielle pour la communication réseau. Elle transforme des structures complexes en une séquence d'octets, ce qui va nous permettre de transporter facilement différents types de données, ainsi que de faire communiquer des programmes de différents langages ensemble, une fonctionnalité qui nous sera utile lors des tests de régression et de performance.

Après une étude des différentes options qui s'offraient a nous, nous avons décidé d'utiliser Flatbuffers, une bibliothèque de sérialisation *cross-platform* qui supportent des langages tels que C, Python et Rust. Cette bibliothèque ayant été initialement créée pour des applications à fort enjeu de performance, elle correspond parfaitement a notre projet.

2.1.2 Technologies réseau

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 demandé dans ce projet) une sécurité dans le transport des données.

3 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.
- Il n'existe pas de méthode asynchrone simple pour detecter la création ou le passage en listening d'un socket. Des outils comme netlink en C offre un mécanisme de notification qui malheureusement ne notifie pas les évènement précédement mentionné. La seul méthode asynchrone restante aurait été d'utiliser les eBPF en Rust et en C. Cela permet d'exécuter de manière sécuriser du code dans le kernel et d'avoir accés à des fonctionnalité particulière indisponible dans le user space. Cependant, non seulement, c'est très technique à utiliser mais en plus il faut compiler le code puis l'injecter pour qu'il fasse effet. Donc, utiliser ce type de mécanisme, pour qu'il fonctionne avec n'importe quel port demandé par l'utilisateur, aurait demandé soit de recompiler un nouveau code à chaque fois, soit de pouvoir communiquer via un système comme socket unix ou tube avec le code s'exécutant dans le kernel. Tout ça en recevant des notifications du kernel de façon asynchrone aussi. Bref c'est une méthode imparfaite qui aurait pu fonctionné si on avait eu le temps, mais n'aurait pas forcément permis de réduire l'overhead de façon significative par rapport au méthode synchrone qui on été retenu.

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.

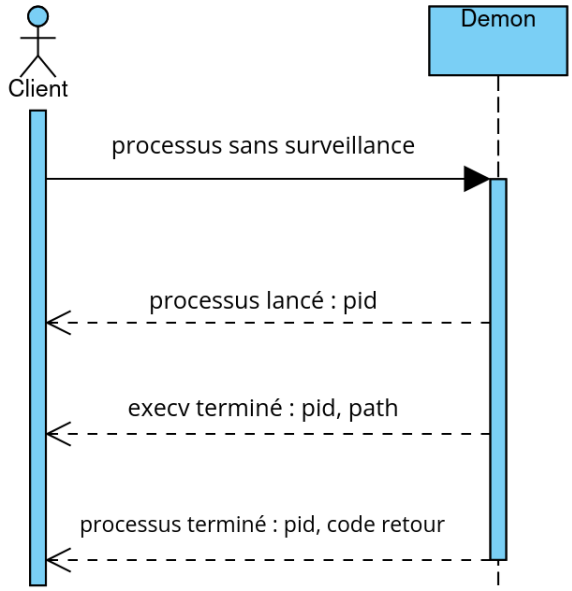
Nous avons également commencé à créer une **suite de tests de régression**. Cette suite permet de vérifier que toute nouvelle modification ne casse pas les fonctionnalités existantes, assurant ainsi la stabilité au fur et à mesure que la base de code évolue.

Notre démon sera capable :

- D'exécuter des fichier binaire exécutable. Ce qui comprend des commandes (Unix-like) : *echo*, *sleep*. Mais aussi, par exemple, n'importe quelle programme C déjà compilé qui se trouve sur la machine ou le daemon s'exécute.

- Des surveillances sur un **socket** ou sur un **fichier** avec *inotify*.
- De kill un process en cours d'exécution

Voici un exemple de diagramme de séquence d'une execution d'une commande :



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
    }
}
```

3.2.2 Détail, utilisation de epoll dans le daemon C

Un détail qui mérite une explication un peu plus poussé est l'utilisation de epoll dans le daemon C. Epoll en c permet de faire comme poll c'est à dire gérer plusieurs file descriptor en même temps mais avec plusieurs fonctionnalité supplémentaire. Celle qui s'est avéré très utile est le fait de pouvoir attacher une structure de données personnalisé à n'importe quel évènement epoll ajouté. Par exemple comme ceci :

```
struct epoll_event ev;
event_data_sock *edata = malloc(sizeof(event_data_sock));
if(edata==NULL){
    free(edata);
    perror("malloc");
    return -1;
}
edata->fd = server_socket->sockfd;
edata->type = SOCK_CONNEXION;
edata->sock_info=server_socket;
ev.events=EPOLLIN;
ev.data.ptr=edata;
if (epoll_ctl(epollfd, EPOLL_CTL_ADD, server_socket->sockfd, &ev)==-1){
    printf("error while trying to add file descriptor to epoll interest list");
    free(edata);
    return -1;
}
```

L'interet principal est de savoir à quel type de notification on a affaire. Car on ne traite pas une notification de message sur un socket de la même manière qu'une notification provoqué par un signalfd. Mais aussi cela permet de stocker des info supplémentaire nécessaire pour certaine fonctionnalité. Par exemple on devait pouvoir déclancher une notification quand un fichier atteint une certaine taille. Or, inotify ne propose pas nativement cette feature. Alors ce qu'on fait c'est qu'on créer un nouveau file descriptor de type inotify, on y ajoute le fichier, on ajoute les flags comme `IN_MODIFY`, puis quand on l'ajoute à epoll on utilise une structure de données personnalisé dans laquelle on stock la taille qui trigger la notification vers le client. Cette méthode a été utilisé de multiple fois, par exemple pour savoir quel fils vient d'exécuter execve ou pour conserver les info du socket servant de serveur pour accepter les connexions arrivantes.

3.2.3 Implémentation de la sérialisation

Pour implémenter la sérialisation nous avons commencé par établir un schéma, qui nfous permet de définir des structures de données que l'on va sérialiser. Nous avons décidé de catégoriser chaque

type de message que le client peut envoyer au démon ainsi que ceux que le démon peut envoyer au client comme des *events*. Chaque message sérialisé contient un *event* parmi tout les *events* possibles. Tous les *events* sont nommés et répertoriés dans une *table*, et chaque *event* est ensuite défini plus haut dans le schéma.

```
union Event {
    RunCommand,
    KillProcess,
    EstablishTCPConnection,
    EstablishUnixConnection,

    ExecveTerminated,
    ProcessLaunched,
    ChildCreationError,
    ProcessTerminated,
    TCPSocketListening,
    InotifyPathUpdated,
    InotifyWatchListUpdated,
    SocketWatched,
    SocketWatchTerminated
}

table Message {
    events: Event;
}
```

Exemple de la définition d'un *event* :

```
    table ExecveTerminated {
    pid: int32;
    command_name: string;
    success: bool;
}
```

Nous avons ensuite codé une fonction pour sérialiser et une fonction pour désérialiser chacune des *table* répertoriées dans l'*union* Event. Ce travail a été effectué en Rust pour le démon codé en Rust, en C++ pour le démon codé en C ainsi qu'en Python pour effectuer différents tests sur le démon.

Nous avons choisi de coder la sérialisation du démon C en C++ car si nous avions travaillé en C, il aurait fallu utiliser un projet à part, FlatCC, qui contient son propre compilateur de schéma ainsi que sa propre bibliothèque en C. Cette solution n'était pas viable car FlatCC n'est pas maintenu de la même manière que Flatbuffers, et la bibliothèque C n'est pas nécessairement mise à jour au même rythme que dans les autres langages, ce qui aurait posé problème. Nous avons préféré rester sur la même dépendance pour tout le projet.

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. 5). 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 Gestion du projet

4.1 Partage des tâches

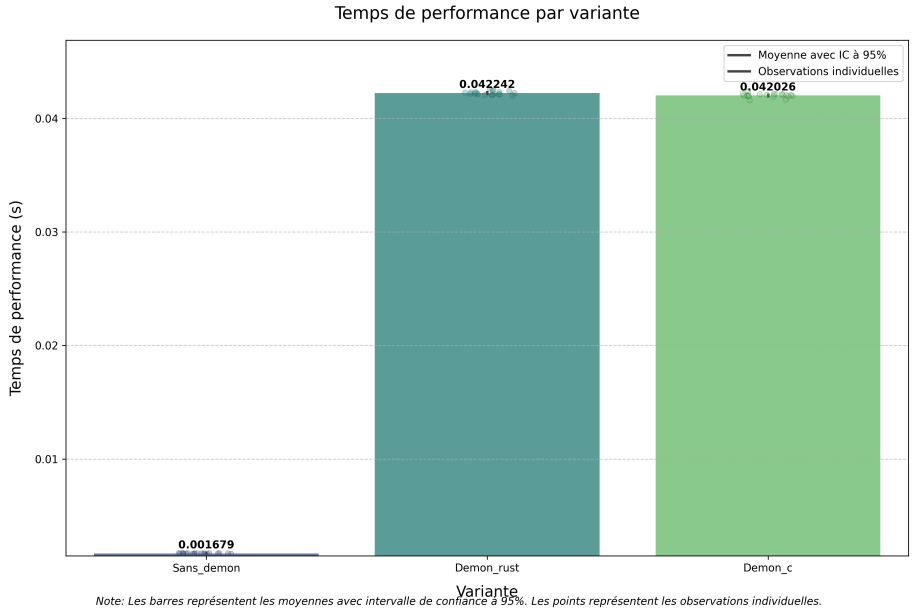
La répartition des tâches s'est faite de façon assez naturelle. Mohamed s'est concentré sur la partie sérialisation en Rust et les tests d'overheads. Alexandre s'est occupé de la sérialisation C et python pour permettre la création des tests de régression. Quant à Telmo et Hadrien, ils se sont respectivement occupé de la conception du daemon C et Rust. Enfin les tests de régression ont aussi été conçu par Telmo et Hadrien.

4.2 Bilan personnel

- Hadrien Lagier : Ce fut une experience enrichissante pour ma part. Dans mon cursus universitaire j'ai toujours eu des projet où l'on nous disait quoi faire. C'est l'inverse dans ce projet où on a du s'organiser par nous même et faire avancer le sujet de recherche. Ce projet m'a permis de me rendre compte que la recherche est un domaine qui m'intéresse tout particulièrement.

5 Analyse

Pour mesurer les performances de nos `daemons` nous comparons l'overhead généré par rapport à l'appel système d'une commande simple : **echo**. Nous mesurons d'abord 20 fois le temps moyen mis par l'appel consécutif de 30 fois **echo**. Ensuite, au début de chaque test d'un `daemon`, celui-ci est créé avant de devoir recevoir 30 appels consécutifs de la même commande **echo**. Là encore nous effectuons une moyenne sur 20 mesures. :



Note: Les barres représentent les moyennes avec intervalle de confiance à 95%. Les points représentent les observations individuelles.

L'appel système d'**echo** dure en moyenne 0.8 ms, tandis que les `daemons` en Rust et en C mettent respectivement 42.2 ms et 42.0 ms pour exécuter la commande. L'écart entre ces deux derniers est trop faible pour désigner un `daemon` plus performant que l'autre sur l'échelle temporelle, si une personne souhaite poursuivre le dévellopement d'un démon elle devra choisir celui avec le language dont elle a le plus d'affinité. De façon plus globale, les `daemons` lancent l'exécution d'une commande rapidement, l'objectif souhaité de ce projet est donc satisfait.

Pour ce qui est de l'analyse de la **mémoire vive** :

- En **Rust** : on obtient a l'exécution une utilisation de 4.1 Mo lorsque on attend une commande. Elle est de 4.4 Mo quand elle exécute une commande `sleep`.

6 Conclusion

En conclusion, nous avons réussi à produire 2 daemon de contrôle offrant les même fonctionnalités, avec un overhead léger. Le tout avec un layer de sérialisation permettant une communication interlanguage garantissant une certaine flexibilité du coté de l'utilisateur. Cependant, il reste des pistes d'amélioration possible. Par exemple, pour le moment on ne peut avoir qu'un seul client connecté par daemon. La detection de socket se fait de façon synchrone ce qui pourrait être améliorer. On pourrait aussi étendre le schéma Flatbuffers pour, par exemple, proposer plus de type d'évènement différent au moment de la reception d'une notification inotification inotify. On pourrait aussi permettre à l'utilisateur de décider du timeout pour la detection de socket. On pourrait ajouter la possibilité de créer et détecter des socket unix. Enfin à l'heure actuel, il n'est aussi pas possible de supprimer, depuis le coté client, un évènement inotify déjà ajouté. Bref, le code fourni n'est qu'une base sur laquelle étendre les fonctionnalités au grés des besoins.