Documentation réponse

Rédigé par :

Loan BELTRAN

SOMMAIRE

Table des matières

I. Présentation du sujet

1. Contexte

L'objectif de ce projet est de concevoir et de développer une architecture multi-serveurs capable de recevoir des requêtes de clients, de compiler et d'exécuter des programmes dans un langage de programmation défini, tout en gérant la répartition des charges entre plusieurs serveurs pour garantir une exécution continue. Cette architecture devra être conçue pour gérer efficacement plusieurs clients simultanés et, en cas de surcharge d'un serveur, déléguer les tâches à d'autres serveurs du cluster.

- Le projet doit être entièrement réalise avec le **langage de programmation python**.
- Le client doit pouvoir **envoyer un code source** (python, java ou C) via une **interface graphique** à un **serveur maître** en spécifiant son port ainsi que son adresse IP. Le code doit ensuite être envoyé à un **serveur esclave disponible**[1] pour que celui-ci le traite et le renvoie au serveur maître, le renvoyant lui-même au client.
- Le serveur maître doit être capable de **gérer plusieurs clients** à la fois.
- Il doit être possible de définir des **contraintes techniques** au serveur esclave comme une utilisation maximale du CPU ou un nombre maximum de programmes qu'il peut exécuter.
- La communication entre les différentes machines doit s'effectuer via des **sockets** tout en garantissant la **robustesse** de celle-ci (gestion d'erreurs).

2. Contraintes techniques

Deux contraintes techniques dominent le sujet :

- Les différents serveurs esclaves doivent être en mesure d'exécuter du code dans les langages python, java et C.
- Utilisation de mécanisme permettant la surveillance du CPU.

Il est donc primordial d'installer les outils adéquats pour parvenir au résultat attendu.

[1] Un serveur esclave dit disponible est un serveur qui est en capacité d'exécuter le code en fonction des contraintes qui lui sont appliqués (utilisation maximale du CPU, nombre de code en cours d'exécution)

II. Contenu du projet

1. Client

Du côté du client, le projet regroupe l'ensemble des fonctionnalités demandées.

Lors de son exécution, il affiche à l'utilisateur une interface graphique **intuitive**. Le client est capable de **choisir l'adresse IP ainsi que le port** du serveur maître auquel il souhaite se connecter. Il est également en mesure de **choisir un fichier** et de **l'envoyer via socket** au serveur maître choisit.

Concernant le fonctionnement, le client se connecte au serveur maître.

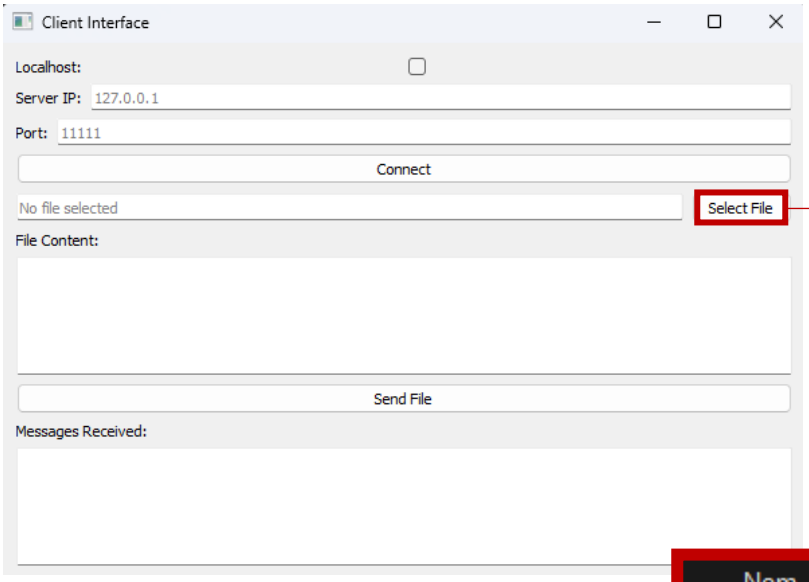


Image 2: Interface client

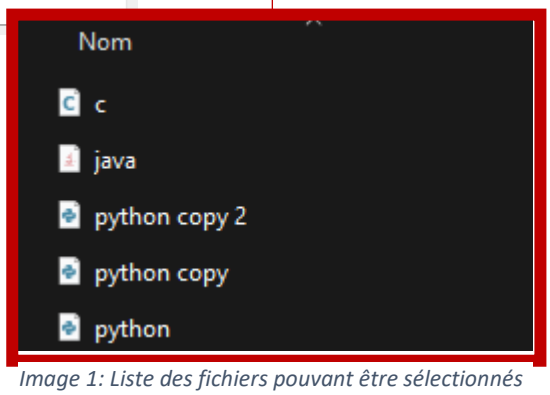


Image 1: Liste des fichiers pouvant être sélectionnés

Notez que lors du choix du fichier, l'utilisateur peut choisir des fichiers avec les extensions .py .c et .java.

Lorsque l'utilisateur importe son fichier (file content), il a la possibilité de le modifier avant de l'envoyer au serveur esclave. Néanmoins, la modification n'est pas sauvegardée dans le fichier original.

Le client reçoit le retour d'exécution de son code dans la zone « Messages Received ».

L'utilisateur a la possibilité de cocher « localhost », cela change automatiquement l'adresse IP avant qu'elle soit assignée à l'adresse IP locale de son ordinateur (127.0.0.1).

2. Serveur Maître

Concernant le serveur maître, plusieurs des différentes tâches demandées ont été mises en place.

Lors de son exécution, le serveur affiche une interface graphique au client. Celui-ci à la possibilité de définir l'adresse IP du serveur ainsi que les ports sur lesquelles vont se connecter les clients et les serveurs esclaves.

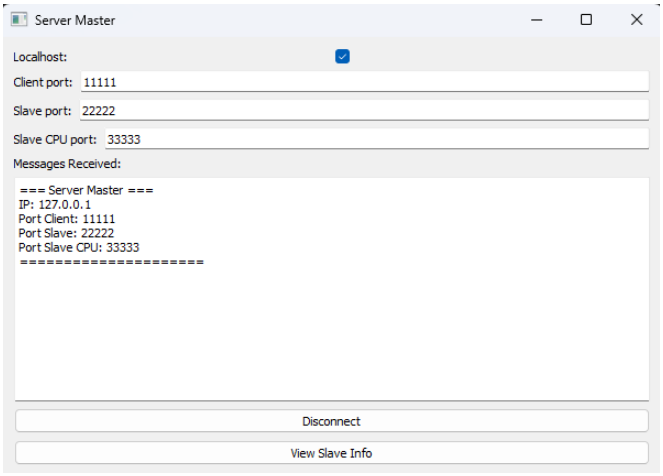


Image 3: Interface du serveur maître

Une fois les informations entrées, il initialise la connexion.

Lorsque le serveur reçoit un message du client, il vérifie si un serveur esclave est disponible, puis lui envoie le code. Dans le cas contraire, il indique au client que le serveur ne peut pas traiter sa demande pour le moment.

Enfin, lorsque le serveur esclave renvoie le code exécuté ou le retour d'erreur du code, le serveur maître renvoie le résultat au bon client.

De plus, le serveur est capable de suivre en directe l'évolution de la charge du CPU ainsi que le nombre de codes en cours d'exécution de chacun des serveurs esclaves via une interface graphique.

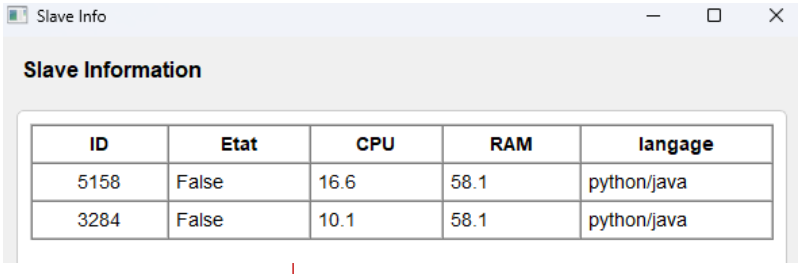
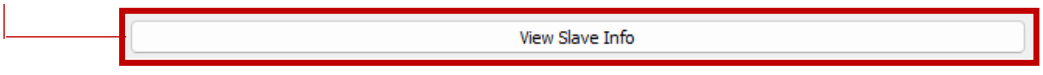


Image 4: Information des serveurs slaves



3. Serveur Esclave

Le serveur esclave peut se connecter à l'aide de socket au serveur maître. Lorsqu'il reçoit le code à exécuter du serveur, il l'exécute puis le renvoie au serveur.

Lors du lancement du serveur esclave, l'utilisateur peut choisir les différents codes que celui-ci est capable d'exécuter, il peut également choisir la charge du CPU/RAM maximale ou le nombre de codes pouvant être exécuté avant que le serveur ne devienne indisponible.

De manière permanente, il envoie au serveur maître sa charge CPU ainsi que le nombre de code qu'il est en train d'exécuter. Il compare ces valeurs aux différents critères et indique s'il est disponible ou non.

4. Technologies choisies

- Python 3.13.1 (langage de programmation du projet) (WINDOWS)
- Ubuntu 24.04 (VM hébergeant le serveur maître) + Python 3.12.3
- Debian 12 (VM hébergeant le serveur esclave) + Python 3.11.2

Notez que les différentes paquets pythons ont été installé avec la commande *apt-install* sur les différentes machines.

III. Contenu absent du projet et difficultés rencontrées

L'ensemble des fonctionnalités principales ont été implémenté. Néanmoins, après plusieurs tentatives pour implémenter des fonctionnalités supplémentaires, celles-ci ont été abandonné.

Nous avons essayé d'automatiser le lancement de serveur esclave par le serveur maître lorsque la communication n'est pas en localhost. Après avoir recherché plusieurs solutions, la plus pratique aurait été la réalisation d'un daemon, mais celle-ci demandait trop de temps et de connaissances que nous ne connaissions pas. Après un essai, cette idée a été abandonné.

Nous avons également essayé d'observer la charge du CPU des différents serveurs esclaves depuis le serveur maître grâce à des graphiques. Néanmoins, nous avons rencontré de grandes difficultés à afficher des graphiques sur PyQT5. L'idée a donc été abandonné. Nous nous sommes tournés vers un affichage simple dans un tableau.

La plus grande difficulté rencontrée a été l'émission et la réception du message par le bon client. En effet, il s'agit des difficultés ayant impacté le projet sur la plus longue période. Il a fallu une vingtaine d'heures et plusieurs dizaines d'essais afin de trouver une solution au problème.

Enfin, il a fallu insérer nos codes sur des machines virtuelles. Il nous a donc fallu les configurer tout en gérant les dépendances de chacun (PyQt, psutil). L'installation initiale sur des machines Debian s'est avérée complexe en raison de problèmes de compatibilité et de configuration. Après plusieurs tentatives infructueuses, nous avons décidé de nous orienter vers des machines Ubuntu, dont l'installation s'est déroulée plus efficacement.

Image 5: Logo Qt5

IV. Améliorations possibles du projet

Voici une liste des différentes améliorations qui auraient été intéressante sur le projet :

- Sécurisé l'échange de données entre les machines

En effet, il aurait été très intéressant de sécuriser les échanges entre les données. Un système de cryptage ou d'authentification aurait été une voie à envisager si le temps nous l'avait permis

- Gestion de langages supplémentaires

Afin d'augmenter la polyvalence de notre projet, nous aurions pu développer des fonctionnalités permettant de gérer d'autres langages de programmation tel que C++ ou javascript.

- Affichage de la charge du serveur esclave sur des graphiques

Pour améliorer l'expérience utilisateur, il aurait été intéressant (malgré des essais infructueux) d'insérer des graphiques représentant l'évolution de la charge du CPU en fonction du temps.

- Lancement dynamique des serveurs

Le lancement dynamique des serveurs lorsque le serveur en local aurait été intéressant si le serveur ne peut plus exécuter de codes.

- Ajouter une database pour créer des comptes utilisateurs

Afin d'avoir un suivi des utilisateurs, il aurait été utile d'insérer un système de création de compte afin d'avoir un suivi de l'utilisation des clients.

- **Mise en œuvre d'un système de tolérance aux pannes**

Assurer la continuité des calculs en cas de défaillance d'un ou plusieurs serveurs grâce à un système de redondance et de répartition intelligente des tâches.

- **Interface utilisateur améliorée**

Développer une interface web ou graphique pour permettre la gestion et la visualisation du cluster, y compris l'ajout ou la suppression de nœuds, le suivi des tâches et l'affichage des performances.

V. Conclusion

Ce projet de cluster de calcul a permis de poser les bases solides d'un système distribué capable de gérer des tâches de manière collaborative entre plusieurs machines. Les objectifs principaux ont été atteints, notamment la répartition des calculs, la communication efficace entre les nœuds, et l'implémentation des fonctionnalités essentielles.

Cependant, certaines améliorations pourraient encore être envisagées, telles que la sécurisation des échanges, le support de langages supplémentaires, l'affichage graphique des charges, ou encore l'ajout d'une base de données pour la gestion des utilisateurs. Ces pistes ouvrent des opportunités

intéressantes pour des travaux futurs et pourraient rendre ce cluster encore plus polyvalent, performant et adapté à des besoins réels.

En somme, ce projet a non seulement enrichi nos connaissances techniques dans des domaines variés (réseaux, programmation, optimisation), mais a également mis en évidence l'importance la gestion de projet et de la réflexion autour de solutions évolutives. C'est une étape significative dans notre parcours et un tremplin pour des développements futurs.

VI. Tables des illustrations