UHA

Rapport final SAE302

FLORIAN VISCA

Sommaire

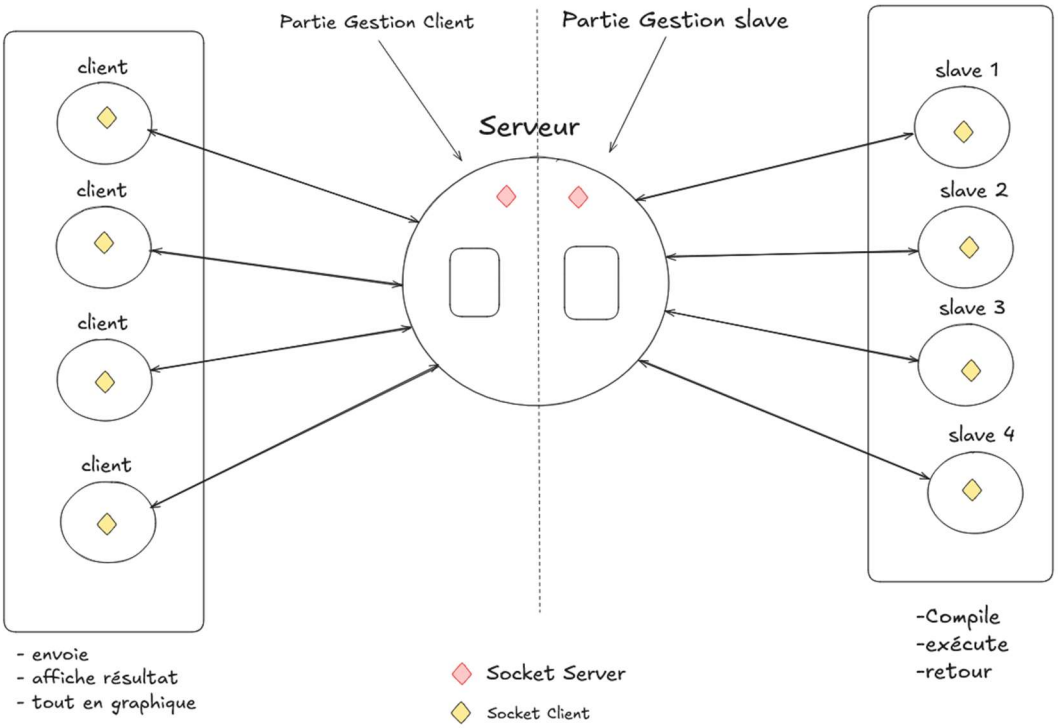
Table des matières

Introduction

L'objectif de ce projet est de concevoir et de développer une architecture multiserveur capable de recevoir des requêtes de clients, de compiler et d'exécuter des programmes dans un langage de programmation défini, tout en gérant dynamiquement la répartition des charges entre plusieurs serveurs pour garantir une exécution fluide et évolutive. 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.
Afin d'assurer une flexibilité maximale, le projet utilise des technologies telles que **PyQt6** pour l'interface graphique côté client, permettant une interaction fluide et intuitive. Les serveurs sont conçus pour exécuter des programmes écrits dans divers langages de programmation, tels que **C**, **C++** et **Java**, garantissant ainsi une large compatibilité avec les applications soumises.
Le système intègre également une répartition dynamique de la charge, permettant de déléguer les tâches entre les serveurs en fonction de leur capacité de traitement.
Le projet est conçu pour être extensible et évolutif, offrant ainsi la possibilité d'ajouter facilement de nouveaux serveurs à l'architecture pour répondre à des besoins croissants. La solution vise à offrir une exécution rapide et fiable des requêtes des clients tout en maintenant une gestion optimale des ressources du système.

Architecture du projet

L'architecture de mon projet correspond au schéma ci-dessous :



Sur ce schéma, on peut observer que le serveur établit deux connexions distinctes en utilisant des sockets : une connexion pour les clients et une autre pour les slaves. Ces connexions sont créées en tant que **sockets serveurs**, permettant au serveur de recevoir et de gérer les requêtes provenant des deux types d'entités.

- **Connexion avec les clients** : Chaque client se connecte au serveur via un **socket client**. Cette connexion permet au client de soumettre des requêtes, telles que l'envoi de fichiers ou l'exécution de programmes. Le serveur réceptionne ces requêtes et retransmet à un slave si nécessaire.
- **Connexion avec les slaves** : De la même manière, les slaves se connectent au serveur principal à l'aide de **sockets clients**. Une fois connectés, ils agissent comme des unités d'exécution, prêtes à recevoir des tâches que le serveur leur attribue, comme la compilation ou l'exécution de programmes.

Ce modèle de double connexion permet une gestion centralisée des interactions entre les clients et les slaves, tout en assurant une séparation claire des responsabilités. Grâce à cette architecture, le serveur peut agir comme un intermédiaire intelligent, distribuant efficacement les charges entre les slaves et répondant aux requêtes des clients de manière rapide et fiable.

Technologies

Les technologies utiliser pour mon programme sont :

- Python : C'est le langage principalement utilisé pour développer ce projet
- Java : Utiliser pour compiler et exécuter des programmes soumis par les clients
- C et C++ : Utiliser pour compiler et exécuter des programmes soumis par les clients
- Regex : Utiliser pour la vérification des fichiers envoyer
- Thread : Utiliser pour avoir plusieurs communications sur le serveur
- PyQt6 : Utiliser pour faire l'interface graphique du client

Outils et environnement de développement :

Visual Studio Code : Principal environnement de développement utilisé pour écrire, tester et organiser les différents fichiers du projet, grâce à ses extensions et sa prise en charge multiplateforme.

Comment déployer les différents programmes :
Je vais fournir une explication rapide du déploiement. Si cela ne suffit pas, une documentation d'installation est disponible avec ce document.
Pour le déploiement général, il faudra démarrer le serveur en premier, puis connecter le nombre de slaves souhaité, et enfin, vous pourrez lancer les différents clients.

Installation du Serveur (master) :
Il faut vérifier que les dépendances, telles que **PyQt6**, sont bien installées sur votre machine. La commande pour lancer le programme est la suivante :

py.exe <chemin_vers_master.py>\master.py --portC <port_client> --portS <port_slave>

Remplacez <chemin_vers_master.py> par le chemin d'accès au fichier **master.py** et définissez les ports d'écoute pour le socket client (--portC) et le socket slave (--portS).

Installation du Slave :
Le slave utilise des bibliothèques natives de Python, mais il est nécessaire d'installer les compilateurs **gcc** et **g++** pour l'exécution des programmes en C, ainsi que le compilateur Java **JDK**. La commande pour lancer le programme est la suivante :

py.exe <chemin_vers_slave.py>\slave.py <IP du server> <port slave>

Remplacez les éléments suivants :

- <chemin_vers_slave.py> : le chemin d'accès complet au fichier **slave.py**.
- <IP_du_server> : l'adresse IP du serveur auquel le **slave** doit se connecter.
- <port_slave> : le port utilisé pour la communication **slave** sur le socket du serveur.

Installation du Client :
Comme pour le serveur, il faut installer la dépendance **PyQt6**. La commande pour lancer le programme est la suivante :

py.exe <chemin_vers_client.py>\client.py

Les informations sur le serveur seront à rentre dans l'interface

Problèmes rencontrés

1. **Manipulation des threads**
   La gestion des threads a été un défi majeur, particulièrement pour permettre la connexion simultanée de plusieurs clients et slaves au serveur master, tout en gérant efficacement les différents messages envoyés.
   - **Problèmes rencontrés** : Difficulté à organiser et synchroniser les échanges entre les threads.
   - **Solution mise en place** : J'ai décomposé mon code en plusieurs fonctions pour le rendre plus compréhensible et organisé. J'ai également ajouté des logs pour faciliter les tests et identifier les sources d'erreurs.
2. **Gestion des fichiers sur les slaves**
   La partie la plus contrariante a été la gestion des fichiers sur un slave unique.
   - **Problème principal** : Je n'ai pas réussi à mettre en place une fonctionnalité permettant de gérer plusieurs fichiers sur un même slave.
   - **Statut actuel** : Ce problème reste non résolu et, par conséquent, cette option n'est pas encore déployée.
3. **Exécution des programmes dans différents langages**
   - **Problèmes rencontrés** :
     1. Difficulté à concevoir un code générique et robuste pour compiler, exécuter, et renvoyer les résultats des programmes soumis dans différents langages.
     2. Avec les langages C et C++, j'ai rencontré un problème spécifique lié à l'installation du compilateur sur Windows. Après avoir configuré TDM-GCC, j'avais oublié de redémarrer mon PC pour que les nouveaux chemins PATH soient pris en compte.
   - **Solution mise en place** : Une fois le redémarrage effectué, le compilateur fonctionnait correctement. Pour le reste, j'ai ajusté les étapes de compilation et d'exécution pour chaque langage.
4. **Load Balancing**
   - **Problèmes rencontrés**

     La mise en œuvre d'un mécanisme efficace de répartition des charges entre les slaves s'est avérée complexe.

     Initialement, j'avais envisagé une gestion dynamique basée sur la récupération en temps réel de l'utilisation CPU des slaves, mais cette approche n'a pas pu être mise en œuvre en raison de contraintes techniques et de complexité.

   Solution mise en place

   J'ai opté pour une gestion dynamique utilisant un compteur de fichiers. Ce mécanisme permet de répartir les tâches en tenant compte du nombre de fichiers déjà attribués à chaque slave.

Envoi et réception des programmes

- **Problèmes rencontrés** :

1. Au départ, je n'avais pas mis en place de mécanisme pour marquer le début et la fin de l'envoi des fichiers, ce qui causait des blocages côté slave.
2. Le slave restait en attente indéfiniment, pensant que l'envoi n'était pas terminé.

- **Solution mise en place** : J'ai ajouté des logs pour identifier les problèmes et mis en œuvre un protocole de contrôle pour indiquer explicitement le début et la fin de chaque fichier transmis.

Réflexion d'améliorations du projet

Pour rendre ce projet plus robuste et complet, plusieurs fonctionnalités importantes restent à implémenter. Voici les principaux points à développer :

1. Sécurité

- **Problème actuel** :
  Actuellement, le projet ne dispose pas de mécanismes de sécurité robustes. Par exemple, un client malveillant peut se faire passer pour un slave sans difficulté.
- **Solutions proposées** :
  - Mettre en place un système de chiffrement des communications (par exemple, via SSL/TLS) pour sécuriser les échanges entre les clients, les slaves et le serveur.
  - Ajouter un mécanisme d'authentification pour les clients et les slaves. Cela pourrait inclure la saisie d'un identifiant et d'un mot de passe lors de la connexion.
  - Implémenter un système de gestion des clés pour garantir que seules les entités autorisées puissent se connecter au serveur

2. Gestion des données

- **Problème actuel** :
  En cas de coupure inattendue, les données envoyées par les clients ou les tâches en cours peuvent être perdues, ce qui affecte la continuité du service.
- **Solutions proposées** :
  - Ajouter une base de données au sein du projet pour stocker les informations importantes telles que :
    - Les programmes envoyés par les clients.
    - Les tâches en cours et leur état.
    - Les logs des connexions et des actions effectuées.

- En cas de redémarrage du serveur, cette base de données permettrait de reprendre les tâches interrompues ou d'analyser les événements passés pour améliorer le système.
- Améliorer l'interface graphique du serveur pour offrir une meilleure visibilité sur les connexions actives, l'état des tâches, et les erreurs éventuelles.

3. Amélioration de l'interface utilisateur

- L'interface graphique actuelle peut être enrichie pour offrir une expérience utilisateur plus intuitive et des informations en temps réel sur :
  - Les connexions actives (clients et slaves).
  - L'état des tâches en cours.
  - Les performances du système (par exemple, la charge actuelle des slaves).