Document de réponse

1. Sommaire :

2. Fonctionnalités - Programmation :

a) Gestion des communications :

Dans un premier temps, il fallait séparer les connexions clientes des connexion serveurs du cluster, 2 sockets sont donc initialisés.

L'une des principales problématiques à se poser en premier c'est le stockage des informations de chaque client. Il fallait organiser la manière dont les données sont organisées, et qu'elles puissent être retrouvée efficacement. J'ai donc déclaré deux variables listes python, une pour les connexion serveurs et l'autre pour les connexions clientes. Là où les listes pythons sont puissantes, c'est qu'un objet d'une liste peut être également une liste. On peut donc indenter hiérarchiquement les données à l'image d'une base de donnée. J'ai alors initialisé par client, un objet de la liste correspondantes avec toutes les informations relatifs à celui-ci dans un premier échange pour éviter une surcharge de communication.

La seconde problématique à été le protocole de communication. Pour pouvoir différencier les messages communiqués dans chaque sens, j'ai établi des règles d'en-têtes de message, décrites dans la documentation des codes sources. Ainsi, seul un thread d'écoute par client est initialisé et est capable de différencier les différentes tâches à faire selon l'en-tête reçu. L'avantage est d'éviter tout conflit lié à 2 port d'écoutes simultanés qui peuvent très fortement générer des erreurs. L'inconvénient est de devoir communiquer beaucoup plus d'informations via les messages car les valeurs de variables ne peuvent pas être conservées et doivent être communiquées à chaque fois pour pouvoir traiter la suite du script quand le port d'écoute reçoit un message.

b) Gestion des threads et Events :

Afin de pouvoir gérer plusieurs tâches simultanées, on a étudier durant la ressource R3.09 l'utilisation des thread en python, qui, sans leur utilisation, ce projet aurait été beaucoup plus difficile voir impossible pour les critères demandés. Pour le script serveur maître, j'ai eu besoin de 4 threads, sans compter le thread MAIN, qui est le processus initial lors de l'exécution d'un script.

- Un premier thread pour la console, qui permet l'entrée de commande, importante pour l'interaction directe avec le client.
- 2 threads pour gérer les connexions entrantes à chacun des sockets et à la gestion de ceux-ci.
- 1 threads initialisé après l'initialisation d'une connexion entrante pour le port d'écoute.

Un Event 'stop' à été créé pour apporter un contrôle sur chacun des threads à l'arrêt du programme principal.

Un problème majeur que j'ai eu était le bloque-ment d'un thread sur une fonction d'écoute, et donc ne se terminait jamais à l'arrêt du programme principal. La solution était de configurer un time-out sur une connexion au socket et de gérer les erreurs liés à celui-ci.

c) Gestion du Load Balancing :

Dans le sujet donné, on avait plusieurs lignes directives de données pour gérer ce critère :

"Si le serveur principal (ou maître) atteint une certaine limite en termes d'utilisation du CPU ou du nombre de programmes en file d'attente"

Avec psutil , j'ai donc opté pour la solution de la charge CPU, même si la gestion avec le nombre du processus est pratiquement identique, à la différence que l'on doit comparer la longueur d'une liste contenant les PIDS associés au serveur esclave plutôt qu'à la charge CPU.

Lors de l'initialisation d'un serveur esclave, celui-ci renvoie sa charge CPU qui est alors stockée dans la liste servant de base de donnée. Par la suite, une boucle renvoyant la charge CPU au serveur maître toutes les 5 secondes met à jour cette liste pour permettre une gestion des données plus efficace.

Lors de l'exécution d'une tâche par le client, le serveur principal peut alors comparer la charge de chaque serveur esclave distinctement et sélectionner la charge la plus faible.

d) Le Monitoring :

Une un thread gérant les entrées clavier dans le terminal est initialisé et permet l'entrée de commande. Ces commandes permettent la gestion et la supervision du cluster de serveur ainsi que des clients. Comme la commande "/server list", qui renvoie tous les serveurs connectés, leur configuration (ip :port, charge CPU, langages supportés, ..).
chaque commande contient un descriptif disponible dans la console du serveur ou dans le code source.

e) La sécurité :

Peu de sécurité sont mises en place actuellement dans le projet, dû au manque de temps et aux priorités. Néanmoins, j'envisage une authentification à chaque connexion aux sockets, impliquant une clé différente à chaque version de client et serveur esclave. Permettant d'avoir une gestion de version des clients et serveurs connectés. Et une déconnexion forcée si aucune clef dans le registre de correspond. Le python, n'étant pas un langage compilé, il est difficile d'apporter une réelle sécurité car le code source est lisible directement par quiconque y aurait accès.

J'imagine également un isolement des scripts exécutés, pour éviter tout problème en cas de code malveillant ou de code qui influe directement avec le système. Dans ce cas là, une exécution des serveurs esclaves dans une virtualisation devient nécessaire.

Une confidentialité des données doit être aussi prise en compte, aucune donnée n'est ne transite entre les différents serveurs à l'exception :

- Du nom du fichier (nécessaire à la compilation JAVA)
- Du fichier en lui même.

Un cryptage et décryptage des données via un hash connu uniquement du client et du serveur peut être intéressant à mettre en place à l'entrée et sortie de la transmission d'un fichier voir d'un fichier, voir de toutes communication entre les composantes du cluster. Mais encore une fois, la problématique du Python qui est un langage non compilé et lisible de tous est encore présente.

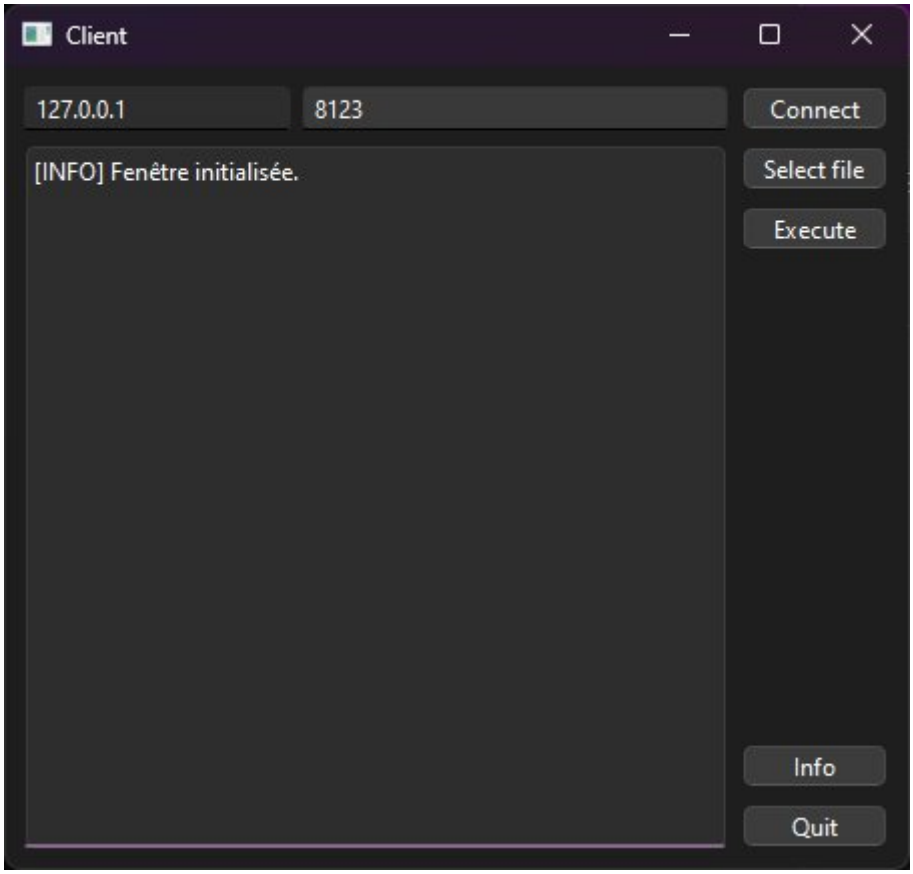
f) La configuration d'un serveur esclave

Lors de l'initialisation d'un serveur esclave, celui-ci se connecte avec les données renseignées durant la configuration, puis effectue des tests, pour connaître son environnement d'exécution. Notamment la présence de gcc pour C, gpp pour c++, jdk pour java, et python. Dans chaque cas il récupère la version du compilateur / exécuteur installé et le communique au serveur maître. J'aurai aimé avoir le temps d'approfondir cet aspect. Comme l'os, la version de l'os. Et communiquer au client plus d'information sur l'environnement d'exécution en plus du simple résultat.

3. Fonctionnalités – Interface graphique

Le client est la partie qui devait être la plus simple possible. Le critère principal était que le client devait être Intuitif, j'ai donc eu une approche d'apport en fonctionnalité limitée. Car une surcharge de fonctionnalité serait contre ce critère demandé.

a) capture du client :



Le client contient :

- 2 champs où d'on renseigne IP/PORT
- 5 boutons connexion, selection de fichier, exécution de la tâche, Info et Quit.
- Et une console

b) Arrêt du client

Une modification importante à été de gérer l'arrêt brut du serveur via la croix supérieur droit (Windows). Le module PyQT6 est très modulable et simple à utiliser, et permet de gérer les actions à effectuer quand l'utilisateur décide de fermer brutalement le client.

c) Console du client

La console est l'approche que j'ai eue pour vraiment avoir un retour sur les tâches qui sont effectuées. C'est un QtextEdit en lecture seul (qui ne permet pas l'édition de texte) que j'avais pour projet d'y intégrer l'éditeur de texte. Via un bouton 'Edit', le fichier sélectionné serait alors lu, le champ de texte en serait plus en lecture seule, et une fois l'édition terminé, les informations contenu dans le QtextEdit auraient écrasé le fichier précédent.

d) Possibles changements

Si je devais maintenant changer certaines choses sur ce client au niveau de l'interface graphique, ce serait le bouton Info, qui renvoie le statut de connexion 'connecté à l'adresse ...' où 'non connecté', que j'aurai complété en rajoutant plus d'informations comme le type de tâche que le serveur est capable de traiter

4. Conclusion

Pour conclure, j'ai beaucoup apprécié ce projet. Je pense que la conceptualisation libre de nos façon de procédé est un point que j'ai le plus apprécié. Ça m'a permis de beaucoup développer mes compétences en python pour certains modules, ainsi qu'en général dans la gestion de projet en ensemble. Même si c'est très minime, on a pas beaucoup l'occasion dans le cadre du cursus de toucher à d'autres langages que du python. J'ai pu me familiariser avec le fonctionnement de langages compilés comme le C ou le C++ principalement que je n'avais jamais manipulé.