Conception et Développement d'une Architecture Multi-Serveurs pour Compilation et Exécution de Programmes

Sommaire

Sommaire 1
Contexte 2
Description du projet 2
  Connexion client-serveur 2
  Gestion de multiples clients 2
  Load Balancing (Répartition de charge) 2
  Mécanismes de communication 3
  Scalabilité et Robustesse 3
Contraintes techniques 3
Fonctionnalités supplémentaires (facultatif) 3
Livrables 3
Critères d'évaluation 4
  Documentation 4
  Fonctionnalités du programme 4
  Fonctionnalités supplémentaires (10%) 4
Un sujet plus simple 5

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 dynamiquement la répartition des charges entre plusieurs serveurs pour garantir une exécution fluide et scalable. 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.

Quel que soit votre projet final (complexe ou simple), il faut lire la partie complexe et la partie simple.

Description du projet

Votre tâche sera de développer une application distribuée qui met en œuvre une architecture multi-serveurs capable de gérer les fonctionnalités suivantes :

Connexion client-serveur

- Le client doit pouvoir se connecter à un serveur et envoyer un programme écrit en Python. Le client sera développé sous format graphique permettant de spécifier l'IP du serveur (une valeur par défaut sera proposée) et un port (de même une valeur par défaut sera proposée).
- Le serveur doit accepter le programme, le compiler (si nécessaire) et l'exécuter.
- Le résultat de l'exécution (ou les erreurs de compilation) doit être renvoyé au client.

Gestion de multiples clients

Le serveur doit être capable de gérer simultanément plusieurs clients. Chaque client peut soumettre un programme, et le serveur doit pouvoir traiter plusieurs requêtes de façon concurrente.

Load Balancing (Répartition de charge)

- Si le serveur principal (ou maître) atteint une certaine limite en termes d'utilisation du CPU ou du nombre de programmes[1] en file d'attente, il doit être capable de déléguer des tâches à un ou plusieurs serveurs secondaires (esclaves).
- Les serveurs secondaires doivent pouvoir s'exécuter de manière autonome une fois lancés et renvoyer les résultats au serveur maître qui, à son tour, transmettra les résultats au client.
- Le nombre maximum de programmes sera définit en paramètre du programme.

[1] Ce nombre devra être un paramètre de lancement du programme

Mécanismes de communication

- La communication entre le client et le serveur, ainsi qu'entre les serveurs, doit se faire via des sockets.
- Il faudra également gérer la robustesse des connexions (gestion des erreurs, reconnexion en cas de défaillance, etc.).

Scalabilité et Robustesse

- Voir les fonctionnalités supplémentaires (facultatif)

Contraintes techniques

Compilation et Exécution des programmes clients : Vous devrez prévoir l'exécution de programmes dans un ou plusieurs langages (par exemple, un serveur qui peut compiler et exécuter des programmes Python ou C). Les outils de compilation et d'exécution (comme GCC pour C ou l'interpréteur Python) doivent être disponibles sur les serveurs.

Utilisation de ressources système : Le projet doit intégrer des mécanismes permettant de surveiller la charge CPU et mémoire de chaque serveur. Ces informations seront utilisées pour décider de la répartition des charges entre les serveurs.

Fonctionnalités supplémentaires (facultatif)

Monitoring du Cluster : mise en place d'une interface sur le serveur maître d'un outil de monitoring pour visualiser l'état du cluster (charge des serveurs, nombre de programmes en cours d'exécution, etc.).

Persistante des données : en cas de défaillance d'un serveur, les tâches non accomplies doivent pouvoir être récupérées et relancées sur un autre serveur.

Sécurité : mise en place de protocoles de sécurité (authentification, cryptage des communications) pour protéger les échanges entre les clients et les serveurs.

Scalabilité : le serveur maître doit également avoir la capacité de démarrer dynamiquement de nouveaux serveurs en fonction de la charge du système. Ces serveurs secondaires doivent être capables de prendre en charge la compilation et l'exécution des programmes des clients.

Robustesse : le système doit être capable de s'adapter à une augmentation du nombre de clients et de serveurs, tout en maintenant de bonnes performances.

Livrables

- **Code source du projet :** comprenant le code du serveur maître, des serveurs secondaires et du client sur une plateforme git que vous devrez partager avec moi (frederic.drouhin@uha.fr)

- **Documentation d'installation et d'utilisation :** détaillant l'installation du client et du serveur. Elle devra également détailler le démarrage des serveurs et du client.
- **Rapport final :** détaillant l'architecture globale, le choix des technologies, et les instructions pour déployer et tester le système. Vous devrez également expliquer les défis rencontrés, les solutions mises en œuvre, et une réflexion sur les améliorations possibles pour le système.
- **Vidéo de démonstration :** montrant le système en fonctionnement avec plusieurs clients et serveurs, en mettant en évidence la gestion des charges.

Critères d'évaluation

Documentation

- Document d'installation (20%)
- Rapport final (20%)
- Vidéo de démonstration (10%)

Fonctionnalités du programme

Un point d'attention : le client à partir des documentations devra être capable de mettre en œuvre les fonctionnalités suivantes :

- Fonctionnalités de base (20%) : le système permet de gérer des requêtes clients, compiler et exécuter les programmes, et renvoyer les résultats.
- Gestion des clients multiples (20%) : le serveur est capable de traiter plusieurs clients simultanément.
- Qualité du code et documentation (10%) : le code est bien structuré et la documentation est claire et complète.

Fonctionnalités supplémentaires (10%)

Toute amélioration ou ajout significatif au système de base sera valorisé

- Monitoring du cluster
- Persistante des données
- Sécurité
- Scalabilité
- Robustesse

Ce projet vous permettra de travailler sur des aspects clés du développement logiciel : architecture distribuée, gestion des processus, communication réseau, répartition de charge et scalabilité, tout en manipulant des concepts de bas niveau comme les sockets et la gestion des serveurs.

Un sujet plus simple

Après avoir lu le sujet plus complet, je vous propose de travailler sur les éléments suivants :

- Coder un serveur
    - Le serveur reçoit une requête d'un client
        - S'il a déjà un programme qui s'exécute d'un autre client, il renvoie un message indiquant qu'il doit se connecter à un autre serveur.
        - S'il n'a pas de programme qui s'exécute
            - Le serveur reçoit le programme
            - Le serveur compile le programme si nécessaire
            - Le serveur exécute le programme
            - Le serveur renvoie au client les résultats du programme
    - Au démarrage, le serveur doit pouvoir s'exécuter sur un port donné en argument - je peux démarrer plusieurs serveurs si je le souhaite -
- Coder un client
    - Le client doit être sous forme d'une interface graphique
    - L'utilisateur upload le programme à envoyer au serveur
    - L'utilisateur spécifie le nom ou l'IP de la machine
    - Le programme est envoyé au serveur
        - Si le serveur rejette le programme, un nouveau nom/IP/port peut être proposé sans interrompre le programme
        - Si le serveur accepte le programme, le client attends (sans bloquer l'interface qui affichera un compteur de temps) et affiche le résultat lorsque le serveur lui renvoie le résultat.

Les critères d'évaluations sont les mêmes que pour le programme plus complexe (voir page précédente).