Projet de programmation réseau

José Vander Meulen, Anthony Legrand, Grégory Seront

Ce projet consiste à construire un système distribué simulant le fonctionnement d'un système exécutant du code dans le cloud. De tels systèmes sont déjà offerts par Amazon avec la plateforme "Aws Lambda" ou par Google avec la plateforme "Google Cloud Functions".

Ce système permettra d'exécuter du code à distance et d'obtenir des statistiques sur les exécutions. Comme c'est un simulateur, il ne gérera pas les aspects de sécurité inhérents à un tel système. Il sera composé de différents programmes qui ne seront potentiellement pas tous exécutés sur une même machine.

C'est un projet à réaliser par **groupe de trois étudiants** durant les semaines 10, 11 et 12 de ce quadrimestre. Nous vous laissons le choix de former les groupes. Néanmoins, vous devez choisir des partenaires qui assistent chaque semaine à la même séance que vous. Notez que votre **présence** en séance est **requise** durant les semaines 10, 11 et 12.

Description du projet.

Le système à développer est un programme distribué qui permet d'exécuter du code C à distance. Il est composé d'un serveur central et d'applications clientes qui se connecteront au serveur central pour effectuer les différentes opérations. Dans la suite de ce document, nous avons tenté de délimiter le plus précisément possible les différents composants du simulateur, afin de vous laisser vous focaliser sur les aspects techniques du système et de vous permettre de le construire dans le temps imparti.

Le système est composé d'un serveur central qui contiendra une liste de programmes écrits en C et qui maintiendra les statistiques sur les exécutions de ces programmes. Parallèlement à ce serveur central, des applications clientes se connecteront à distance pour exécuter ces programmes et consulter les statistiques d'utilisation. La figure 1 présente une vue schématique du simulateur.

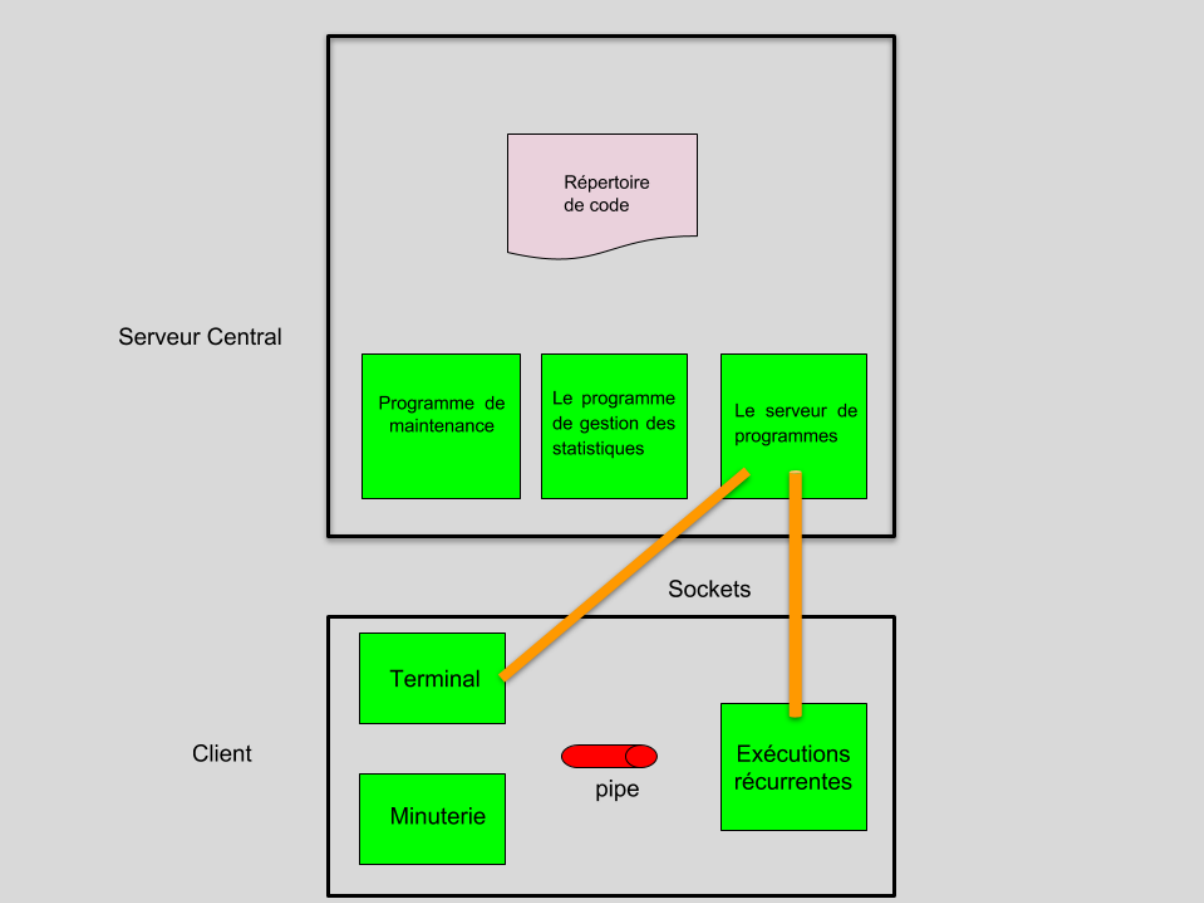

Figure 1

Le serveur central

Le serveur central simule un système similaire à "Aws Lambda". En pratique, il gère un **répertoire de code** contenant au plus 1000 programmes. Chacun de ces programmes correspond à un fichier C et ne prend pas d'argument. En pratique, votre programme ne doit pas gérer des programmes C composés de plusieurs fichiers sources ou des programmes C qui prennent un ou plusieurs arguments.

Le serveur associe à chaque programme :

1. Un numéro allant de 0 à 999.
2. Un booléen indiquant si le programme a généré une erreur ou non lors de la compilation.
3. Un entier représentant le nombre d'exécutions du programme.
4. Un entier représentant le nombre de secondes cumulées de chaque exécution du programme.

Ces informations sont représentées de manière informatique à l'aide d'une **mémoire partagée**. Cette mémoire étant partagée, vous devrez veiller à **protéger les accès concurrents** à cette mémoire à l'aide de sémaphores. Notez que la mémoire partagée est distincte du répertoire de programmes (où se trouvent les codes sources et exécutables).

Le serveur central offre **3 programmes** qui permettent de gérer les fichiers sources et les statistiques :

1. Un **programme de gestion des statiques**.
2. Un **serveur de programmes** qui offre la possibilité d'ajouter, de modifier, ou d'exécuter un programme.
3. Un **programme de maintenance** qui permet de simuler des opérations de maintenance sur le serveur central.

Le programme de gestion des statistiques : **`stat`**

Le programme de gestion des statistiques est un programme qui permet d'afficher les statistiques d'un programme. En pratique, c'est un programme dont le nom est "`stat`" qui prend en argument un numéro "`n`" associé à un programme.

Le programme de gestion des statistiques affiche les informations suivantes sur cinq lignes :

1. Le numéro du programme.
2. Le nom du fichier source.
3. Un booléen indiquant si le programme a généré une erreur ou non lors de la compilation.
4. Un entier représentant le nombre d'exécutions du programme si le programme compile (0 si le programme ne compile pas).
5. Un entier représentant le nombre de microsecondes cumulées de chaque exécution du programme si le programme compile (0 si le programme ne compile pas).

Imaginons un fichier source "`Mario.c`" syntaxiquement correct et portant le numéro 3. Imaginons également que ce programme ait été exécuté 15 fois pour une durée totale de 49 microsecondes. Le programme "`stat`" affichera les 5 lignes suivantes:

```
>> ./stat 3
3
Mario.c
true
15
49
```

Pour limiter le temps de développement et aller à l'essentiel, vous pouvez supposer que le numéro de fichier source passé en argument représente un entier valide qui correspond effectivement à un fichier source. Si ce n'est pas le cas, le comportement de votre programme est indéterminé. Pour les mêmes raisons, ce programme n'offre volontairement que des fonctionnalités basiques. Néanmoins, il n'est pas trivial dans le sens où il doit accéder de manière concurrente à une mémoire partagée. Pour ce faire, il doit protéger les accès à la mémoire partagé à l'aide de sémaphores.

Le serveur de programmes : **`server`**

Le serveur de programmes permet d'ajouter, de modifier et d'exécuter différents programmes **en parallèle**. Vous pouvez supposer qu'il n'y aura jamais plus de 50 programmes qui tournent en parallèle. C'est un serveur dans le sens où il n'offre pas de moyen direct de faire ces trois opérations. Il faut obligatoirement passer par un client informatique pour pouvoir les effectuer. Vous construirez un tel client qui réalisera les opérations suivantes lorsqu'il souhaite manipuler un programme :

1. Ouvrir une connexion TCP.
2. Envoyer les informations concernant un programme,
    a. Si le client désire ajouter ou modifier un programme, il enverra les informations suivantes :
        i. -1 si c'est un nouveau programme ou le numéro associé au programme si c'est un programme à modifier.
        ii. Le nombre de caractères du nom du fichier source. Vous pouvez supposer que ce nom comprendra au moins 1 caractère et au plus 255 caractères. Notez que plusieurs fichiers sources peuvent avoir le même nom.
        iii. Le nom du fichier source.
        iv. Le contenu du fichier source.
    b. Si le client désire exécuter un programme, il enverra les informations suivantes:
        i. -2.
        ii. Le numéro d'un programme.
3. Recevoir la réponse du serveur.
    a. Le client qui ajoute ou modifie un programme recevra les informations suivantes :
        i. Le numéro associé au programme.
        ii. 0 si le programme compile, un nombre différent de 0 sinon.
        iii. Une suite de caractères qui correspond aux messages d'erreur du compilateur.
    b. Le client qui exécute un programme recevra les informations suivantes :
        i. Le numéro "`n`" associé au programme.
        ii. Un entier indiquant l'état du programme à la fin de son traitement :
            1. -2 si le programme "`n`" n'existe pas.
            2. -1 si le programme "`n`" ne compile pas.
            3. 0 si le programme ne s'est pas terminé normalement.
            4. 1 si le programme s'est terminé normalement.
        iii. Un entier indiquant le temps d'exécution du programme.
        iv. Un entier indiquant le code de retour du programme exécuté à distance.
        v. La sortie standard du programme exécuté à distance.
4. Fermer la connection TCP.

En pratique, c'est un programme dont le nom est "`server`" qui prend comme argument uniquement le port sur lequel se connecte le serveur. Vous pouvez supposer que l'argument et les informations lues par le serveur sont valides. Si ce n'est pas le cas, le comportement de votre programme est indéterminé.

Comme pour le programme de gestion des statistiques, pour limiter le temps de développement, ce programme n'offre volontairement que des fonctionnalités basiques. Néanmoins, il n'est pas non plus trivial car il manipule des "sockets", il crée un fils pour chaque exécution d'un programme et il accède de manière concurrente à la mémoire partagée relative au livre de compte.

Le programme de maintenance : `maint`

Le programme de maintenance permet de simuler des opérations de maintenance manipulant les ressources partagées.

Concrètement, c'est un programme dont l'appel prend la forme "`maint type [opt]`". Il prend en argument un entier représentant le type de l'opération à effectuer et éventuellement un argument supplémentaire qui sera nécessaire pour une des opérations de maintenance :

1. Si "`type`" est égal à 1, il **crée les ressources partagées** relatives au répertoire de code tels que la mémoire partagée et les sémaphores.
2. Si "`type`" est égal à 2, il détruit **les ressources partagées** relatives au répertoire de code.
3. Si "`type`" est égal à 3, il **réserve de façon exclusive le répertoire de code partagé** pour une période de temps donné. La durée de cette période est fournie au programme de maintenance par le paramètre "`opt`" qui représente le nombre de secondes durant lequel le programme réserve de façon exclusive le répertoire de code. Notez que pour vous simplifier la vie, vous pouvez supposer que "`opt`" représente un entier naturel représentable en C. Si ce n'est pas le cas, le comportement de votre programme est indéterminé. Pour construire cette fonctionnalité, pensez à utiliser la fonction "`sleep`". L'intérêt essentiel de cette option est qu'elle offre un moyen effectif de tester les accès concurrents au répertoire de code. En effet, lorsque le programme de maintenance accède au répertoire de code de manière exclusive, ni le serveur, ni le programme de gestion des statistiques ne peuvent accéder à celui-ci.

Les clients

Pour pouvoir **ajouter, modifier et exécuter des programmes**, le système comprend des clients informatiques. Ceux-ci sont composés d'un programme "**père**" qui met à disposition de l'utilisateur final un "prompt" qui offre des commandes permettant de gérer les programmes, d'un "**fils minuterie**" qui génère un battement de cœur à intervalles réguliers et d'un programme "**fils**" qui exécute certains programmes de manière récurrente. Ces exécutions récurrentes se feront de manière séquentielle. Vous pouvez supposer que chaque client n'exécute que maximum 100 programmes de manière récurrente. En pratique, il exécute ces programmes à chaque battement de cœur.

Les systèmes tels "Aws Lambda" offrent généralement la possibilité d'exécuter des programmes de manière récurrente. Généralement, ces exécutions récurrentes sont gérées par un des serveurs d'Aws Lamba. Au contraire, dans votre simulateur, c'est le client qui doit gérer ces exécutions récurrentes.

Concrètement, un client est un programme dont le nom est "`client`" et dont l'appel prend la forme "`client adr port delay`" (où `por` et `delay` sont des entiers naturels). Les arguments "`adr`" et "`port`" représentent respectivement l'adresse et le port du serveur central. L'argument "`delay`" représente l'intervalle de temps (en secondes) entre deux battements de cœur.

Le père et les fils communiquent ensemble à l'aide d'un "**pipe**". Ils communiquent avec le serveur de virements à l'aide de connexions **TCP**.

Le père présente, à l'utilisateur final, un prompt permettant d'exécuter quatre types de commandes :

1. une commande "**+** `<chemin d'un fichier C>`" (où "`<chemin d'un fichier C>`" représente un chemin vers un fichier C) qui permet d'ajouter un fichier C sur le serveur.
2. une commande "**.** `num <chemin du fichier C>`" (où `num` représente un entier naturel) qui permet de remplacer le programme associé portant le numéro `num`.
3. une commande "***** `num`" (où `num` représente un numéro de programme valide) qui transmet à un fils la responsabilité d'exécuter de manière récurrente le programme ayant le numéro "`num`". Ce programme sera exécuté toutes les "`delay`" secondes par le fils.
4. une commande "`@ num`" (où `num` représente un numéro de programme valide) qui demande au serveur d'exécuter le programme ayant le numéro "`num`"**.**
5. une commande "**q**" qui déconnecte le client et libère ses ressources.

Chaque fois que le client communique avec le serveur, il affiche à l'écran les messages renvoyés par le serveur.

Notez que techniquement, nous vous imposons que, pour chaque exécution d'un programme, le client établisse une nouvelle connexion TCP avec le serveur.

Comme pour les programmes liés au serveur, les clients n'offrent volontairement que des fonctionnalités basiques. Néanmoins, ils ne sont pas non plus triviaux car ils sont constitués de plusieurs processus communiquant à l'aide de "pipes". Ils utilisent également des "sockets".

Gestion des arrêts et des situations exceptionnelles du serveur et des clients

De manière générale, ni le serveur, ni le client ne doivent gérer les arrêts brutaux. En particulier, nous supposerons que personne n'effectuera un "`kill -9`" pour "tuer" un de vos programmes. Vous ne devez donc pas gérer ces situations d'arrêts brutaux.

En revanche, il est prévu que votre **serveur de programmes** soit arrêté lorsque l'utilisateur appuie sur les touches "**Ctrl+C**". Lorsque c'est le cas, si votre serveur traite un lot de programmes récurrents, il termine son traitement courant et ensuite s'arrête ; sinon il s'arrête immédiatement.

Vous ne devez pas gérer de manière subtile les situations exceptionnelles (e.g. impossibilité de créer un socket, erreur d'écriture sur un pipe ou un socket, …). Lorsqu'un tel cas arrive, vos programmes affichent un message d'erreur avec la fonction "`perror`" et se termine de manière abrupte sans se soucier de libérer les ressources.

Lors d'arrêts brutaux d'un programme, il est possible que certains IPCs non-utilisés soient encore réservés sur la machine sur laquelle tournait le programme. Pour traiter ces cas, pensez à utiliser les commandes "`ipcs`" et "`ipcrm`" (ou la commande "`./maint 2`" si ce programme est fonctionnel).

Nous vous fournissons la dernière version du module *utils*.

Délivrables et tests de votre programme

Durant la séance de la semaine 10, afin que vous ne commenciez pas directement à coder une solution non valide, nous vous demandons de compléter la figure 1 en précisant :

- les canaux de communication entre les différents processus ;
- le sens des communications (read/write) ;
- le protocole de communication (ex : d'abord 4 bytes représentant un entier, puis 1 byte représentant un caractère, etc.) et le type de données qui transitera sur ces canaux ;
- les IPCs et les processus qui les utiliseront ;
- le contenu et la structure de la mémoire partagée.

Avant de commencer à implémenter votre solution, faites valider votre schéma par un enseignant. Ce schéma peut être fait à la main sur papier. Nous demandons uniquement qu'il soit lisible et compréhensible.

En pratique, la production de ce schéma pourra être perçue comme la phase d'analyse de votre programme.

Pour le lundi 17/05/2021 à 8h00, nous vous demandons de soumettre sur Moovin tous les fichiers sources relatifs à votre application, ainsi qu'un *makefile*. Pour information, nous utiliserons un logiciel anti-plagiat pour nous aider à détecter les éventuels plagiats entre les groupes.

Durant le courant de la semaine 13, nous organiserons une séance durant laquelle nous testerons vos simulateurs. Notez que nous testerons exclusivement l'application que vous aurez soumise sur Moovin. En plus de ces tests, nous discuterons avec vous de votre implémentation.