Projet de programmation réseau

José Vander Meulen, Olivier Choquet, Bernard Frank, Anthony Legrand

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 il s'agit d'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 deux étudiants durant les semaines 10, 11 et 12 de ce quadrimestre. Nous vous laissons le choix de former les groupes. Néanmoins, vous devez choisir un partenaire qui assiste chaque semaine aux deux mêmes séances que vous. Si une séance contient un nombre impair d'étudiants, nous accepterons un groupe de trois étudiants par séance. Notez que votre présence est requise durant toutes les séances des 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.

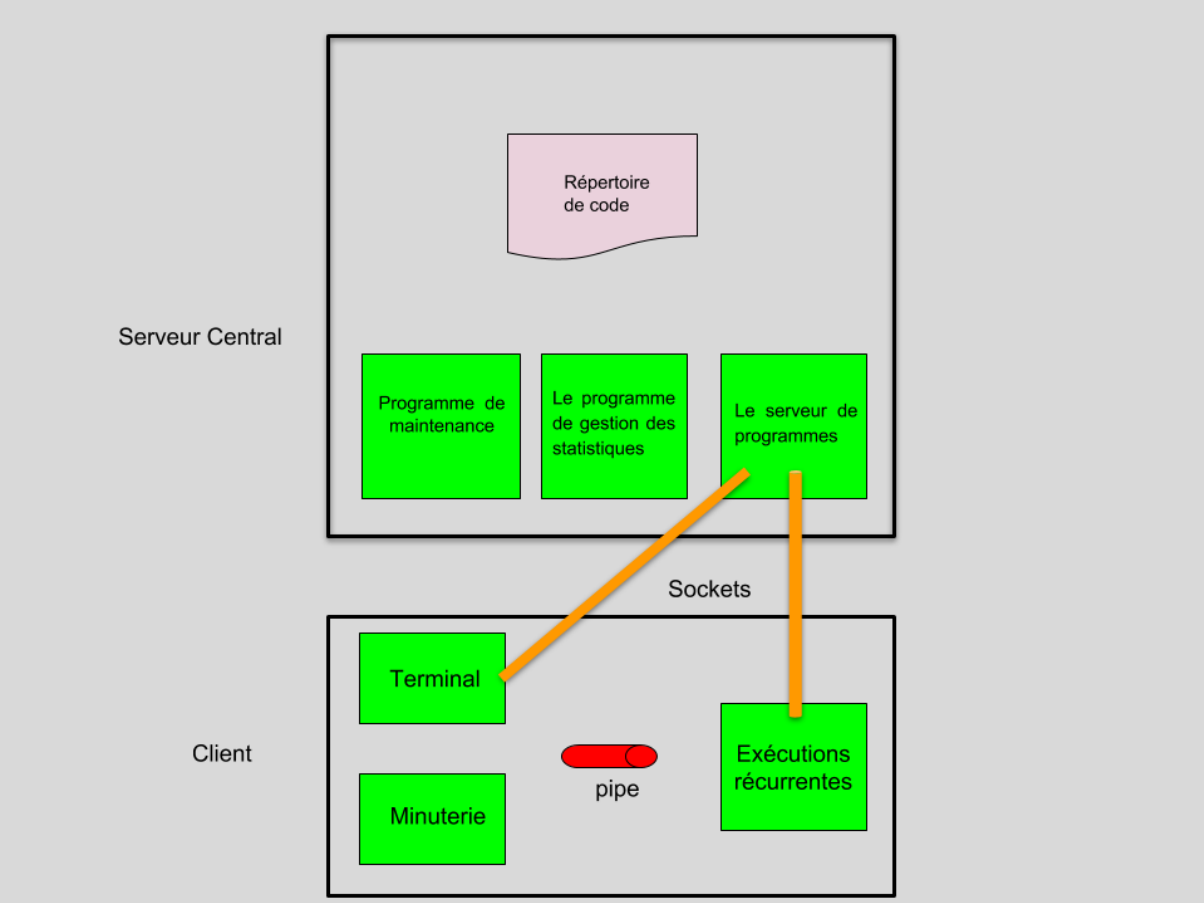

Figure 1

Notez que le fichier **compile.c** contient un programme qui compile un autre programme **hello.c,** sauve l'output de cette compilation dans un fichier **res_compile.txt** et ensuite **exécute** le programme résultant de cette compilation.

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 des plusieurs fichiers sources et 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 nom de 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
5. un entier représentant le nombre de microsecondes cumulées de chaque exécution du programme.

Ces informations sont représentées de manière informatique à l'aide d'une **mémoire partagée** que nous appellerons **répertoire des exécutions**. Cette mémoire étant partagée par différents programmes (voir ci-dessous), vous devrez veiller à **protéger les accès concurrents** à cette mémoire à l'aide de sémaphores.

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

1. Un **programme de gestion des statistiques**.
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

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 "`gstat`" qui prend un numéro "`n`" associé à un programme en argument.

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

1. le numéro du programme
2. le nom du fichier source
3. "0" si le programme a généré une erreur lors de la compilation et "1" si le programme n'a pas généré d'erreurs lors de la compilation
4. un entier représentant le nombre d'exécutions du programme si le programme compile, ou 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, ou 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 "`gstat`" affichera les 5 lignes suivantes :

```
>>./gstat 3
3
mario.c
1
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 à un **programme valide**. 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 partagée à l'aide de sémaphores.

Le serveur de programmes

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. Le serveur doit gérer les demandes d'un tel client qui réalisera les opérations suivantes lorsqu'il souhaite manipuler un programme :

1. Ouvrir une connexion TCP.
2. Recevoir les informations envoyées par le client concernant un programme,
    a. Si le client désire ajouter un programme, il enverra les informations suivantes:
        i. -1
        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. Notez que pour que le serveur détecte la fin de la lecture du fichier source, le client doit fermer sa connexion TCP en **écriture, mais pas en lecture car il devra lire le résultat de la compilation.** Pour ce faire, vous pouvez utiliser l'appel système `"shutdown"`.
    b. Si le client désire exécuter un programme, il enverra les informations suivantes :
        i. -2
        ii. Le numéro d'un programme.
3. Envoyer la réponse du serveur au client.
    a. Le client qui ajoute un programme recevra les informations suivantes :
        i. Le numéro associé au programme
        ii. 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, exprimé en microsecondes, si le programme s'est terminé normalement ; un entier non déterminé sinon.
        iv. Un entier indiquant le code de retour du programme exécuté à distance si le programme s'est terminé normalement ; un entier non déterminé sinon.
        v. Si le programme s'est terminé normalement, l'affichage à la sortie standard du programme exécuté à distance ; rien sinon.
4. Fermer la connexion TCP.

Notez qu'un même programme **peut être exécuté plusieurs fois en parallèle**.

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 répertoire des exécutions.

Le programme de maintenance

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 des exécutions tels que la mémoire partagée et les sémaphores.
2. Si "`type`" est égal à 2, il détruit **les ressources partagées** relatives au répertoire des exécutions.
3. Si "`type`" est égal à 3, il **réserve de façon exclusive le répertoire des exécutions 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 microsecondes durant lequel le programme réserve de façon exclusive le répertoire des exécutions. 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 des exécutions. En effet, lorsque le programme de maintenance accède au répertoire des exécutions 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 de:

- un programme “père” qui met à disposition de l'utilisateur final un “prompt” qui offre des commandes permettant de gérer les programmes ;
- un “fils minuterie” qui génère un battement de cœur à intervalle régulier ;
- un programme “fils” qui demande l’exécution de certains programmes de manière récurrente.

Ces programmes seront exécutés de manière séquentielle sur le serveur. Vous pouvez supposer que chaque client demande au maximum l’exécution de 100 programmes de manière récurrente. En pratique, ces demandes d’exécutions de programmes se font à chaque battement de cœur.

Les systèmes tels “Aws Lambda” offrent généralement la possibilité d'exécuter des programmes de manière récurrente. Ces exécutions récurrentes sont gérées par un des serveurs de “Aws Lambda” plutôt que par un client. Au contraire, dans votre simulateur, c’est le client qui doit gérer l’envoi de ces demandes d’exécutions récurrentes et la réception des messages du serveur liés à ces exécutions.

Concrètement, un client est un programme dont le nom est “`client`” qui prend trois arguments : “`adr`”, “`port`” et “`delay`” (où `port` 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 coeur.

Le père et les fils communiquent ensemble à l’aide d’un “pipe”. Ils communiquent tous deux avec le serveur 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`” (où `num` représente un numéro de programme valide) qui transmet au fils d’exécuter de manière récurrente le programme ayant le numéro “`num`”. Ce programme sera exécuté toutes les “`delay`” secondes par le fils ;
3. une commande “`@ num`” (où `num` représente un numéro de programme valide) qui demande au serveur d'exécuter le programme ayant le numéro “`num`” ;
4. 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.

Cependant, il est prévu que votre serveur soit arrêté lorsque l'utilisateur appuie sur les touches "Ctrl+C". Lorsque c'est le cas, le serveur s'arrête, mais ses éventuels fils terminent l'exécution de leur lot de programmes et ensuite s'arrêtent.

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 terminent de manière abrupte sans se soucier de libérer leurs 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. Pensez à utiliser les commandes "ipcs" et "ipcrm" ou votre programme `maint` pour traiter ces cas.

Délivrables et tests de votre programme

Pour le début de la deuxième séance de la semaine 10, nous vous demandons de produire et de nous remettre en main propre un document de 2 à 3 faces A4, décrivant l'architecture de votre programme et la manière dont vous allez organiser la découpe en modules. En pratique, la production de ce document pourra être perçue comme la phase d'analyse de votre programme.

Pour le samedi 11/05/2018 à 23h59, 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 organiserons une rencontre avec chaque groupe pour discuter de votre implémentation.