Un répartiteur de charge pour des machines en réseau

Encadrant: **Sens Pierre**
Toure Momar Faly et Ung Richard
MASTER SAR
UE PSAR

Sorbonne Université
Campus Pierre et Marie Curie
4 Place Jussieu, 75005 Paris

Table des matières

1 Introduction

L'équilibrage de charge ou encore "load balancing" en anglais désigne le procédé par lequel on distribue une charge entre plusieurs serveurs appartenant à un groupe de machines afin de pouvoir optimiser les performances et raccourcir les temps de réponses.

Au lieu d'avoir une seule machine extrêmement performante mais très coûteuse, vulnérable aux pannes et difficile à scaler, le répartiteur de charge permet d'avoir un réseau de machines moins coûteuses et permettant une mise à l'échelle et une maintenance plus facile.

La distribution des charges est un domaine de recherche particulièrement populaire dans la technologie des serveurs et est d'autant plus importante puisque que les données et requêtes traitées sont de plus en plus massives.

L'objectif de ce projet est de mettre en place le composant logiciel, le répartiteur de charge, qui se situera au-dessus du système d'exploitation de chaque machine et qui réalisera les enjeux précédemment décrits.

1.1 Langage

Pour réaliser ce projet, nous pouvions choisir le langage que nous voulions. Nous avons choisi d'utiliser le langage *C* avec l'API *MPI*.

1.2 Introduction à MPI

L'API MPI (Message Passing Interface) est une bibliothèque de communication par message fournie dans 3 langages : C, C++ et Fortran. Elle nous a été présenté dans l'UE d'algorithmique répartie (AR) de ce semestre par M. Frank Petit. Elle nous permet de faire communiquer par messages des processus sur des ordinateurs distants ou multiprocesseur. Dans notre cas, ces processus jouerons le rôle de serveur.

Cette interface considère un environnement totalement distribué où les processus ne partagent pas de mémoire. Chaque processus a son propre flot de contrôle et son propre espace d'adressage. Ce qui signifie qu'une variable déclarée globale dans le code source reste locale et privée par rapport au processus. Les affichages sont redirigés via le réseau sur le terminal qui a lancé le programme MPI.

1.3 Compilation et exécution

Pour compiler un programme MPI, on doit s'assurer que :

- le fichier source inclut le fichier header mpi.h (directive include <mpi.h>)
- utiliser la commande mpicc au lieu de gcc

On obtient la commande de compilation suivante : **mpicc -o nom_executable nom_prog.c**

Pour exécuter un programme MPI on doit utiliser la commande mpirun au lieu de lancer directement notre exécutable. La commande est :

mpirun –oversubscribe -np XX –map-by node –hostfile ./fichier_hostfile ./nom_executable

- **oversubscribe** : les machines sont autorisées à être sur souscrits, cela signifie qu'on peut leur assigner plus de processus que le nombre de coeurs qu'elles possèdent.

- **np** : permet d'indiquer le nombre de processus que l'on veut lancer
- **map-by** : permet d'indiquer la manière dont on veut répartir les processus dans notre hostfile
- **node** : indique que les processus sont placés à tour de rôle par emplacement (Round Robin).
- **hostfile** : permet d'indiquer la liste des machines sur laquelle on veut lancer nos processus
- **fichier_hostfile** : fichier où l'on a mis le nom des machines sur lesquelles on veut lancer nos processus

2 Consignes

Les problèmes essentiels posés par l'ordonnancement sont l'évaluation de la charge de chaque machine, la gestion de la surcharge, le choix du programme à déplacer ainsi que le choix de la machine cible qui va recevoir le programme.

Les décisions pour le placement des charges se feront à l'aide d'informations sur l'état global du système. Pour cela chaque machine participant à la répartition de charge devra échanger ses informations via au moins un serveur.

Le projet devra implémenter des stratégies dynamiques de placement. Le placement est calculé à chaque fois que le programme est lancé, en fonction de l'état global du système. Pour cela, il est nécessaire de trouver des critères d'évaluation de la charge pour chaque machine du réseau. L'équilibrage de charge nécessite le calcul de la charge globale du système. La charge globale sera définie simplement comme la moyenne des charges des machines participant au placement. Pour la détection de la surcharge, nous commencerons par utiliser des algorithmes à base de seuils fixes puis dynamiques. Au cours de ce projet, il ne sera pas nécessaire d'assurer la migration des tâches en cours d'exécution.

Afin d'y parvenir, nous serons amenés à implémenter les trois commandes suivantes :

1. gstart prog arguments

   Créer un processus exécutant " prog arguments " sur la machine la moins chargée du réseau. À ce processus sera attribué un identifiant global unique sur le réseau (gpid).

2. gps [-l]

   " Global ps " affiche la liste de tous les processus qui ont été lancés par " lance ". L'option –l permet d'afficher un format long (noms exécutable, machine, uid, éventuellement statistiques d'utilisation CPU, mémoire).

3. gkill -sig gpid

   " Global kill " envoi le signal sig au processus identifié par gpid.

Choix de l'orientation du projet

Nous avons eu la possibilité de choisir un thème sur lequel axer notre projet. Les choix étaient :

- Insertion/retrait : Ce thème gère l'insertion et le retrait de machines participant à l'équilibrage de charge. Lors du retrait, les processus locaux devront être relancés sur une autre machine.
- Tolérance aux fautes : Ce thème gère les fautes (reboot, arrêt) de machine pouvant survenir. Comme dans le cas précédent il faut relancer les processus des machines défaillantes (cela implique de détecter les fautes !). De manière optionnelle, vous pourrez mettre en œuvre des stratégies de sauvegarde et restauration de contexte de processus.

Nous avons choisi d'axer notre projet sur l'insertion/retrait des machines participant à l'équilibrage de charge.

Nous nous assurerons de réaliser une interface permettant à l'utilisateur de modifier l'ensemble des machines participant au placement et de suivre l'exécution des programmes.

3 Architecture du réseau

Dans le cadre du projet, nous avons décidé d'utiliser les machines de la PPTI, en l'occurrence celles de la salle 509 (salle de TP/TME de l'UE d'AR). Tous les ordinateurs de cette salle (à l'exception de la machine 8) possèdent la même distribution de MPI. Cela nous permet d'éviter des conflits de versions.

3.1 Structure du réseau : P2P

Pour notre réseau de machine, nous avons décidé d'utiliser une structure en Peer-2-Peer (P2P). Nous avons pour objectif d'exploiter les performances et raccourcir les temps de réponses. Avec le P2P, nous avons une structure de graphe complet permettant à chaque machine de communiquer les unes avec les autres. On évite ainsi un goulot d'étranglement au niveau du répartiteur (comme dans un modèle Client-Serveur), ici toutes les machines jouent le rôle de répartiteur.

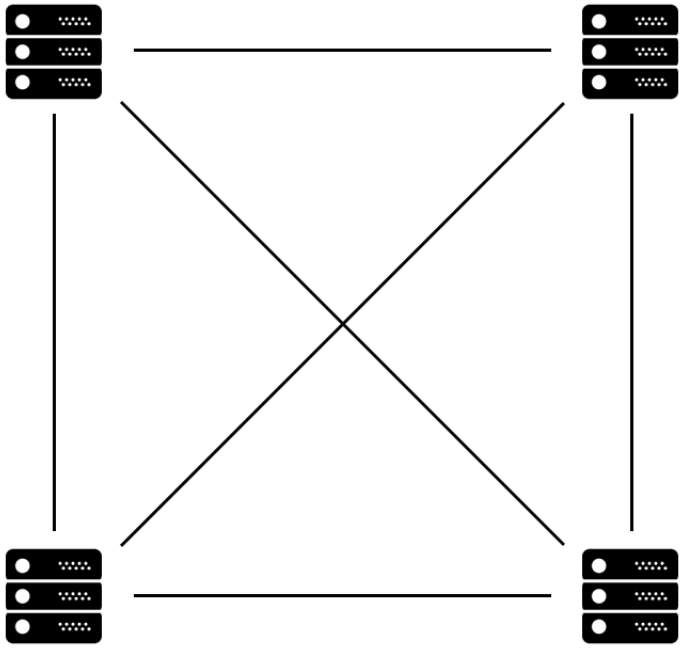

Figure 1 – Réseau P2P

3.2 Configuration du réseau P2P

L'API MPI fonctionne avec le protocole de communication SSH. Ce protocole de connexion impose un échange de clés de chiffrement en début de connexion. Pour cela nous avons dû effectuer quelques configurations sur notre compte à la PPTI.

Normalement, lorsque l'on se connecte à un hôte distant via SSH, il est nécessaire de saisir un mot de passe. Pour que MPI fonctionne correctement, il faut être en mesure de passer outre cette étape. Cette configuration nous permet ainsi de ne pas indiquer le mot de passe à chaque connexion sur machine distante.

Pour cela, nous devons commencer par générer une paire de clef RSA à l'aide de la commande :
ssh-keygen -t rsa

Copier la clé générée dans le fichier des clés autorisés :
cd $HOME/.ssh
cp id_rsa.pub authorized_keys

Lancer la commande :
eval 'ssh-agent'

Ensuite, on ajoute le mot de passe de la clé générée dans le fichier id_rsa pour ne pas a avoir à le recopier à chaque connexion de machine :
ssh-add $HOME/.ssh/id_rsa

Après ces étapes, il suffit juste de se connecter au moins une fois à chaque machine de notre réseau pour enregistrer la clé sur chaque machine.

3.3 Structure interne d'un serveur

Chaque machine du réseau possède au moins un serveur. Nos serveurs sont représentés par MPI comme étant des processus lancés sur des machines distantes. Chacun d'entre eux vont posséder des variables locales leur permettant de stocker l'état global du réseau. Pour cela, ils disposent de 3 tableaux :

- Un tableau d'entier qui indique si un serveur participe ou non à la répartition de charge
- Un tableau de flottant qui indique la charge de l'ensemble des serveurs participant à la répartition de charge
- Une matrice d'entier qui enregistre l'historique des processus lancés sur chaque machine

3.4 Communication entre serveur

La communication entre serveur se fait à l'aide de primitives MPI (MPI_Send MPI_Recv) au sein d'un communicateur. Un communicateur désigne un ensemble de processus pouvant communiquer ensemble, et deux processus ne pourront communiquer que s'ils sont dans un même communicateur. Dans notre cas, le communicateur regroupe l'ensemble des processus que l'on crée dans notre réseau P2P.

4 Gestion de la charge

4.1 Définition de la charge

Dans notre pré-rapport, nous avions défini la charge comme étant une combinaison des différentes ressources (CPU, RAM, nombre de sessions) de la machine auxquelles on affecterait des coefficients arbitraires. Depuis, nous nous sommes rendus compte que Linux dispose d'une fonctionnalité native qui a le même but. Il s'agit du **load average**.

Les load averages représentent le nombre moyen de processus dans la file d'attente des processus *ready for running* pour, respectivement, la dernière minute, les 5 dernières minutes et les 15 dernières minutes. Dans notre projet, on n'utilise que la première valeur. On peut les avoir grâce à l'utilitaire **uptime** ou bien le fichier **/proc/loadavg**.

```
$ uptime
 17:55:56 up 19:49,  1 user,  load average: 0,52, 0,50, 0,54
```

```
$ cat /proc/loadavg
    0.52 0.50 0.54 1/313 13261
```

4.2 Récupération de la charge

Pour récupérer la charge, il suffit d'ouvrir un flux de caractères basés sur un fichier en mode lecture afin de récupérer le contenu du fichier. Voici l'algorithme :

```
float getCharge():
    Ouvrir le fichier "/proc/loadavg" en lecture:
        Recuperer la premiere valeur du fichier
    Retourner le resultat
```

4.3 Mise à jour de la charge

Deux options s'offrent à nous : mettre à jour les charges de chaque machine après chaque création de tâche ou bien, selon un intervalle de temps.

Étant donné que la charge est modifiée toutes les minutes, la première solution n'est pas efficace. Le surplus d'échange de messages pour notifier les changements ne ferait que ralentir le système.

La seconde option quant à elle, paraît plus correcte. À l'aide de la fonction alarm(), nous programmons une temporisation pour qu'elle envoie un signal SIGALRM au processus appelant toutes les 30 secondes, permettant ainsi d'obtenir une mise à jour des charges en nombre de messages raisonnable.

Chaque serveur va procéder à la mise à jour des charges au sein du réseau de la manière suivante :

```
void notifyAll():
    Recupere sa charge
    Pour chaque serveur actif sur le reseau:
        Envoie sa charge
```

5 Gestion de la surcharge

5.1 Charge globale

La charge globale est définie comme la moyenne des charges des machines participant au placement.

Notons C la charge globale, n $\in \mathbb{N}$ le nombre de machines participant au placement, c_i la charge de la i^{eme} machine avec i $\leqslant$ n. La charge globale correspond donc à :

$$C = \frac{1}{n} \times \sum_{i=1}^{n} c_i$$

Définition de la surcharge

On choisit arbitrairement de fixer le seuil de surcharge à 70% de la charge globale. Comme la charge globale varie au cours du temps, le seuil est donc dynamique. Un système en surcharge aura tendance à avoir une baisse de performance. La puissance s'en trouve réduite et la capacité de calcul diminue. Il y aura donc un surcoût en mémoire et en temps de calcul.

Définition de la sous-charge

On choisit arbitrairement de fixer le seuil de surcharge à 30% de la charge globale. Comme la charge globale varie au cours du temps, le seuil est donc dynamique. Un système en sous charge fonctionnera correctement, il consommera juste des ressources inutilement. Le système est donc sur-dimensionné et il y aura une séquentialisation du traitement des tâches.

5.2 Détection de la surcharge et de la sous-charge

Après chaque mise à jour de l'ensemble des charges du réseau, chaque machine participant à la répartition de charge va comparer sa charge avec la charge globale afin de détecter une éventuelle anomalie au niveau de la charge.

Choix du programme à déplacer et de la machine cible

On choisit d'employer une politique FIFO (First In, First Out), c'est-à-dire de déplacer le programme le plus ancien vers la machine la moins occupée. Comme la charge évolue lentement, de cette manière on réduit le risque de déplacement d'une charge en cours d'exécution.

Insertion d'une machine

Lorsque l'on détecte la surcharge d'une machine. On va chercher l'identifiant de la machine active la moins chargée et lui envoyer une tâche. À la réception de cette tâche, la machine la moins chargée va vérifier si elle est en surcharge, si ce n'est pas le cas, la machine surchargée va répéter le procédé jusqu'à ce qu'elle soit en dessous du seuil de surcharge. Dans le cas contraire, cela signifie que peu importe à quelle autre machine active une tâche est envoyée, au moins une machine sera en surcharge. Dans ce cas, le système vérifie s'il est encore possible de rajouter une machine à notre réseau de machines actives. Si cela est possible, on va ajouter la machine au réseau et équilibrer les tâches. Sinon, il va bloquer la création de nouvelles tâches.

La nouvelle machine ajoutée au réseau va prévenir toutes les machines actives du réseaux de sa présence.

Retrait d'une machine

Lorsque l'on détecte la sous-charge d'une machine. Elle va envoyer des messages pour vérifier s'il reste encore au moins une machine active dans le réseau, si ce n'est pas le cas la machine va continuer de traiter les requêtes seule jusqu'à ce qu'elle soit en surcharge. Dans le cas contraire, la machine va se retirer du réseau et prévenir les machines actives restantes dans le réseau puis envoyer ses tâches sur la machine la moins chargé.

6 Implémentations

6.1 Commandes

La fonction gstart est appelée par la machine la moins chargée du réseau.

6.1.1 Gstart

```
void gstart(char** argv, int gpid, int indice_tab_process):
    pid = fork()
    if pid == 0: // processus fils
        Execute la commande argv[0] avec ses parametres
    Ajoute le gpid de la commande dans le tableau des processus a indice
   indice_tab_process
    Ajoute le pid de la commande dans le tableau des processus
    Ajoute le nom de la commande dans le tableau des processus
```

6.1.2 gps

La fonction gps est appelée par toutes les machines actives du réseau.

```
void gps(int opt):
    if opt == 0: // Pas d option
        Pour chaque processus executer sur le serveur:
            Affiche pid, uid et commande
    else:
         Pour chaque processus executer sur le serveur:
            Affichage long pid, uid, commande, CPU et memoire
```

6.1.3 gkill

La fonction gkill est appelée par la machine ayant le processus qui doit recevoir le signal sig.

```
void gkill(int sig, int pid, int gpid, int indice_tab_process):
    Envoyer le signal sig a pid
    Retirer le processus du tableau de processus
    Envoyer un message <gpid,indice_tab_process> au serveur actif du reseau
   pour retirer le processus de leur tableau
```

6.2 Gestion des réceptions de message

La fonction recv reçoit principalement des messages d'un processus fils qui récupère les commandes de l'utilisateur. Cette fonction est lancée par chaque serveur participant au réseau.

```
void recv():
    while(1):
        Reception d un message
        switch(TAG_MSG):
            case CHARGE:
                Met a jour la charge de l expediteur du message dans son
   tableau de charge
                break

            case GSTART:
                Recupere id de la machine la moins chargee
                if id == rank: // Soi meme
                    lance gstart
                else:
                    Envoie un message a la machine la moins chargee pour lui
   dire de lancer gstart
                break

            case GPS:
                Lance gps
                break

            case GKILL:
                Lance gkill
                break

            case GPID:
                Ajout d un processus dans la matrice d entier (enregistre l
   etat global du systeme)
                break

            case TAG_GKILL_GPID:
                Retire un processus dans la matrice d entier (enregistre l
   etat global du systeme)
                break

            case SURCHARGE:
                Indique la surcharge d une machine
                break

            case SOUSCHARGE:
                Indique la souscharge d une machine
```

7 Difficultés rencontrées

Dans un premier temps, nous étions partis sur l'implémentation d'un répartiteur de charge avec un modèle Client-Serveur en Java. Nous nous étions appuyé sur les cours de Systèmes Répartis Client-Serveur (SRCS). Malheureusement, le modèle Client-Serveur n'était pas adapté pour notre projet. L'un des enjeux du projet étant d'optimiser et raccourcir les temps de réponses, un goulot d'étranglement au niveau du répartiteur allait à l'encontre de l'un de nos objectifs.

Un des problèmes que nous avons rencontré à ce moment là était l'usage des serveurs sockets. En effet, nous voulions utiliser les machines de la PPTI comme cadre pour notre projet mais, malheureusement, nous nous sommes confrontés à des problèmes de permission. Que cela soit à partir de chez nous, ou par ssh à partir d'une machine de la PPTI.

Puis nous nous sommes souvenus qu'on arrivait à faire cette configuration sans problèmes en cours d'algorithmique répartie. D'où l'idée d'implémenter le répartiteur en C en utilisant MPI. Ce qui nous a permis de, non seulement régler tout ce qui était problèmes de permission, mais également d'aller vers l'architecture demandée dans le cadre du projet : un réseau Peer-to-peer.

8 Conclusion

Ce projet s'inscrit dans la continuité des cours que l'on a suivi lors de cette année d'étude. Il constitue une application directe des connaissances que l'on a pu acquérir. En effet, notre première approche nous a été inspirée par l'unité d'enseignement sur les systèmes répartis Client-Serveur avant que nous nous rendions compte que l'API MPI découverte en cours d'algorithmique répartie était mieux adaptée.

Le projet nous aura aussi permis d'avoir un petit aperçu de ce qui se fait dans l'industrie, de comprendre les enjeux de performances et les types de solutions qui sont apportées. Il a également suscité notre curiosité en matière de systems designs de manière générale : nous nous sommes interrogés sur ce qu'était un CDN, DNS, API Gateway etc.