Telecommunications

PR204

Rapport

Sujet de Projet PR 204 : Système et Réseau Implémentation d'une Mémoire Partagée Distribuée (DSM)

Etudiants :
Baudry Alexandre (abaudry10@gmail.com)
Casal Théo (tcasal@enseirb-matmeca.fr)

Professeurs :
Guillaume Mercier
Joachim Bruneau-Queyreix

17 décembre 2021

Table des matières

1 Présentation du projet

Ce projet a pour objectif de mettre en place un système de mémoire partagée entre différents processus lancés sur différentes machines de l'école. Ce projet se décompose en deux grandes phases, la première visant à mettre en place un lanceur permettant l'exécution du processus voulu sur différentes machines préalablement définies dans un fichier *machine_file*. La seconde gérant la mise en place de la zone de mémoire partagée. Le schéma de l'architecture globale à retenir est le suivant :

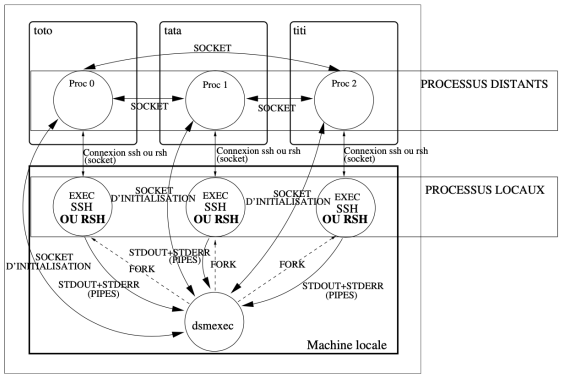

FIGURE 1 – Architecture Globale

2 Phase 1

Pour cette phase 1 on s'intéresse à la mise en place des liaisons représentées en orange sur la figure ci-dessous :

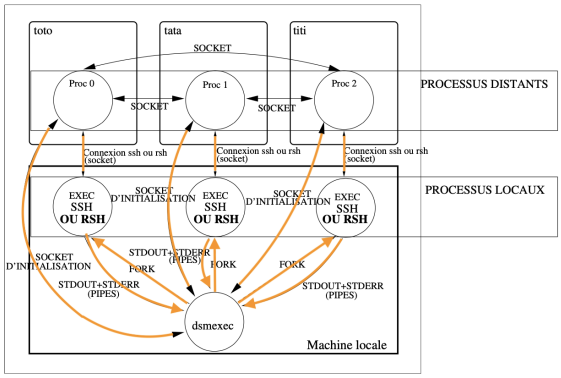

FIGURE 2 – implémentation de la phase1

2.1 Gestion du machine_file

machine_file doit contenir le nom des machines auxquelles le processus doit se connecter en ssh afin d'exécuter le processus voulu sur chacune de ces machines. Il a donc fallu préalablement définir une clef ssh afin de se connecter aux machines sans avoir à rentrer de mot de passe pour chaque connexion voulue. Le fichier *machine_file* peut contenir des lignes vides et il faut alors traiter ce cas afin que notre programme ne pense pas qu'il faille se connecter à

une machine de nom vide. Pour cela la lecture du fichier *machine_file* se fait en deux étapes. Une première étape faisant intervenir la fonction : **ligne_conter(maxstr_t fichier)** comptant le nombre de lignes du fichier différentes d'un " \n". Puis une fois le nombre de lignes comptées, **list_machin(dsm_proc_t *list_machines, maxstr_t fichier, int n_lignes)** se charge de remplir la structure *dsm_proc_t * list_machines* et en particulier son champ : **list_machines[k].connect_info.machine** avec le nom de la k^{ieme} machine.

2.2 Gestion de la commande

La ligne de commande à taper pour lancer notre programme doit être de la forme :

```
dsmexec machinefile truc arg1 arg2 arg3
```

avec *truc* le nom du programme à exécuter sur les machines distantes et (arg1,arg2,arg3) les arguments éventuels de ce programme. Il a donc fallu définir les variables **dsmexec**, **dsmwrap** et **truc** comme variables d'environnement afin que celles-ci soient accessibles peu importe le répertoire courant. Ici *dsmwrap* était le programme à lancer sur les machines distantes lues dans le fichier *machine_file* qui allait à son tour lancer le processus voulu ici *truc*.

2.3 Dsmwrap

Le programme *dsmwrap* était executé sur les machines distantes à l'aide de la fonction *execvp*() qui commençait par se connecter en ssh à la machine voulue puis lançait le programme *dswrap* avec les arguments nécessaires transmis via un tableau d'arguments rempli préalablement. Cette commande était exécutée autant de fois qu'il y avait de machines auxquelles se connecter d'où la nécessité d'un **fork()** dans une boucle for sur le nombre de processus (nombre de lignes non vides).

2.4 Sockets d'écoute

Avant le lancement du processus *dsmwrap* sur les machines distantes, une **socket d'écoute** était créée dans le programme *dsmexec* avec un port dynamique. Le port ainsi que le nom de la machine lançant le *dsmexec* étaient retenus et entrés dans le tableau d'arguments du ssh afin que les processus distants soient en mesure de se connecter. Une fois la socket d'écoute créée dans le *dsmexec* et les sockets créées dans le *dsmwrap* sur chacune des machines distantes, il fallait à présent envoyer les informations des sockets créées sur les machines distantes au *dsmexec* afin que celui-ci puisse accepter les connections. Ainsi les programmes distants pouvaient communiquer avec la socket d'écoute du *dsmexec* et s'échanger des informations.

2.5 Redirection des flux

Afin de pouvoir traiter les informations venant des processus exécutés sur les machines distantes un système de redirection de flux a dû être mis en place pour que les sorties standards ainsi que les sorties d'erreurs soient renvoyées vers le *dsmexec*. Pour cela on passait par la création de **tubes**. On créait deux tableaux ayant pour utilité de retenir les descripteurs de fichier

des extrémités des tubes (fds_out et fds_err). Puis l'on fermait STDOUT_FILENO ainsi que STDERR_FILENO et l'on dupliquait les écritures des tubes créés. Ainsi, ceux-ci prenaient leur place et les sorties standard et d'erreurs étaient redirigées et l'on conservait en mémoire les extrémités des tubes utilisés par les processus distants.

2.6 Lancement du processus voulu

Une fois la lecture du *machine_file* effectuée, la ligne de commande gérée, le *dsmwrap* mis en place et les redirections de flux effectuées, on pouvait alors lancer le processus voulu sur chacune des machines distantes. Pour cela on utilisait le même principe que pour l'exécution du *dsmwrap* via la fonction **execvp**. La fonction prenait en argument le nom du programme à exécuter ainsi que le tableau des arguments (nom_programme,arg1,arg2,arg3) transmis par dsmexec.

3 Phase 2

Dans cette phase 2 on s'intéresse dans un premier temps à la mise en place des liaisons en vert :

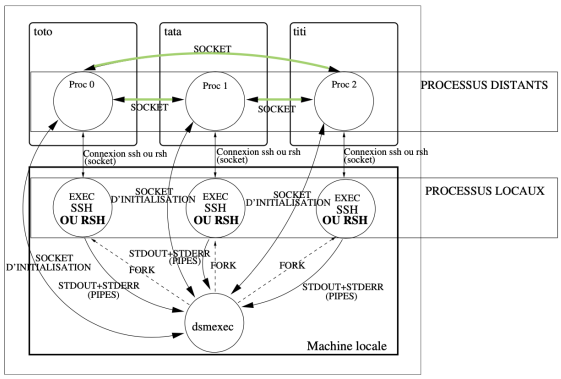

FIGURE 3 – Phase 2

puis à la gestion des zones de mémoire partagée.

3.1 Lancement du programme

Le programme est lancé à l'aide de la commande suivante :

```
dsmexec machinefile exemple
```

Avec **exemple** le programme à exécuter sur les machines données dans le fichier *machine_file*. **exemple** fait appel à des fonctions du programme **dsm.c** (**dsm_init()** et **dsm_finalise()**).

3.2 Réception des informations utiles via les sockets

Dans la fonction **dsm_init()** la première chose à faire était de récupérer les variables d'environnement DSMEXEC_FD et MASTER_FD à l'aide de la fonction **getenv()** afin de pouvoir

utiliser les sockets correspondantes. Une fois ces variables d'environnement récupérées, on pouvait alors lire dans la socket, associée à DSMEXEC_FD, afin de récupérer nombre de processus dsm envoyés par le lanceur de programmes (DSM_NODE_NUM), et de même pour le numéro de processus dsm envoyé (DSM_NODE_ID). Il fallait ensuite récupérer les informations de connexion des autres processus envoyés par le lanceur (nom de machine, numero de port, etc), pour cela on recevait, via la socket, pour chaque processus, la structure $infos\_dsm$ à l'aide de la commande suivante : $recv(atoi(\text{DSMEXEC\_FD}), \&infos\_dsm[k], sizeof(dsm\_proc\_conn\_t), 0)$. Le tableau de structure de type $dsm\_proc\_conn\_t$ (infos_dsm) était préalablement initialisé hors de toute fonction de sorte qu'il soit accessible partout dans le programme. On avait alors à disposition pour chacun des processus distants l'ensemble des informations de la structure $dsm\_proc\_conn\_t$.

3.3 Gestion des sockets inter-processus distants

Une fois ces informations récupérées il a alors fallu créer pour chaque processus une socket d'écoute à laquelle les autres processus se connecteraient. Afin d'éviter que les connexions soient faites plusieurs fois pour chaque socket, il a fallu trouver une astuce pour que chaque processus ne se connecte qu'une seule fois aux autres. Pour cela on bouclait sur l'ensemble des processus, en revanche le processus en question ne se connectait qu'aux sockets des processus d'id (DSM_NODE_ID) supérieur au sien. Ainsi le processus 0 se connectait à tous les processus excepté lui-même, en revanche le processus 1 ne se connectait qu'aux processus d'id superieur à 1. De cette façon, chaque connexion entre socket n'était effectuée qu'une seule fois. De la même façon l'acceptation de connexion ne se faisait que pour les processus d'id inférieur aux autres. Le procédé est décrit pour 3 processus distants dans le schéma ci dessous :

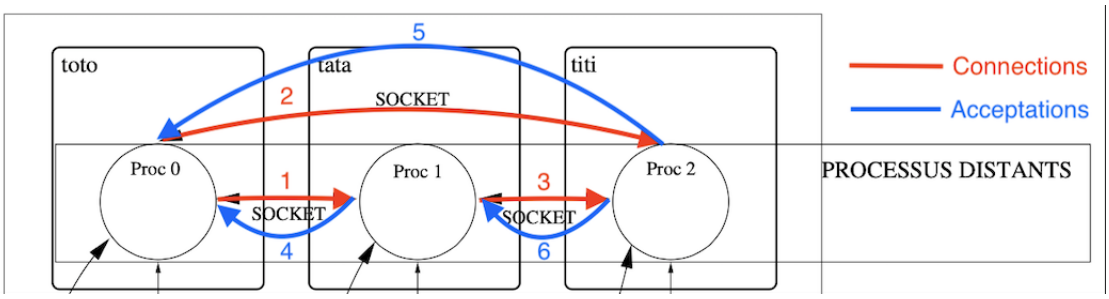

FIGURE 4 – Sockets : Connexions et Acceptations

Un tableau d'entiers était initialisé comme variable globale et rempli avec les descripteurs des sockets lors de la phase d'acceptation.

3.4 Gestion de la mémoire partagée

On crée PAGE_NUMBER zone de mémoire partagée à l'aide de la fonction **dsm_alloc_page()**, elle-même appellant la fonction **mmap()**.

3.5 Mise en place du traitant de signal

Lorsque l'un des processus cherche à avoir accès à une zone de mémoire partagée (une page) qu'il ne possède pas une *segmentationfault* (SIGSEGV) est envoyée par ce processus. Cependant pour éviter de mettre fin à ce processus nous allons nous servir de ce signal émis via l'implémentation d'un traitant de signal qui réagira à l'émission de ce dernier. En effet, une fois le signal SIGSEGV émis par le processus le traitant va entrer en action et se charger de fournir la page au processus cherchant à y accéder. Pour cela le traitant de signal va devoir déterminer via la fonction **dsm_handler()** le numéro ainsi que le propriétaire de la page à laquelle le processus cherche à accéder. Ce dernier utilise les fonctions **address2num()** et **get_owner()** et génère ainsi une requête pour demander l'accès à cette zone de mémoire. La requête est alors envoyée par l'envoie d'une structure de type : *dsm_req_t* (partie suivante). Une fois les requêtes envoyées un **mutex** est mis en place afin que la requête ne soit pas envoyée en boucle. Ce mutex est par la suite déverrouillé par le thread (partie suivante).

3.6 Gestion du daemon

Pour s'assurer de la communication entre les différents processus distants, on met en place un thread qui executera le daemon **dsm_comm_daemon()**. Ce thread aura pour fonction de **poll()** et de traiter les requêtes des autres processus. Pour cela les échanges inter-processus se font via des envoies de structures de type **dsm_req_t** possédant la source de la demande de page, le numéro de la page ainsi qu'une autre structure (**dsm_req_type_t**) spécifiant le type de requête envoyé (demande de page, changement de propriétaire, fin de processus...). Ces requêtes sont ensuite traitées selon leur type au sein d'un **switch** qui se charge pour chacun des cas de faire les changements voulus et de dévérouiller le mutex par la suite.

3.7 Dsm finalize

Pour correctement mettre fin au programme on utilise la fonction *dsm_finalise*() qui envoie, dans un premier temps, pour chaque processus une requête de type DSM_FINALIZE qui sera traitée dans le switch du daemon pour afficher la fin du processus en question. On utilise ensuite la fonction **pthread_join()** pour mettre fin aux threads non terminés. Puis l'on ferme l'ensemble de sockets.

4 Zones d'ombres du projet

Dans la version finale du rendu de ce projet, certaines erreurs persistent, notamment les connexions inter-processus qui ne se font pas systématiquement ainsi que la lecture dans les sockets qui ne fonctionne pas correctement. L'une de nos explications est la méthode de remplissage des descripteurs de fichiers des sockets qui engendrerait que l'écriture ne se fasse pas toujours dans la socket voulue.

5 Conclusion

Bien que pas totalement abouti, ce projet nous a énormément inéeressé de par sa structure dans un premier temps. En effet, le lancement de processus se connectant les uns aux autres via des sockets sur des machines distantes par la simple exécution d'un programme initial (dsmexec) est très intéressant. De même le fait qu'une fois lancé ces processus continuent de s'échanger des informations via les zones de mémoires partagées sans nécessité d'action de l'utilisateur. Ce projet très complet nous aura également permis de revoir et mettre en pratique la quasi-totalité des notions vues en cours. Enfin une partie très appréciée de ce projet pour nous aura été l'utilisation du signal de $segmentation fault$, qui à pour habitude de nous terrifier, pour activer le traitant de signal permettant au programme de traiter les requêtes et continuer de tourner sur les machines distantes.

FIGURE 5 – Autre exemple de cas où il à fallu tuer le sujet pour le relancer