Cahier des charges N°2

Création d'une *Process Management Interface* (PMI)

Description

Ce document détaille le second projet AISE. L'objectif est, basé sur les notions vues en cours, de développer un serveur *PMI* simple et son client. La PMI est un composant logiciel sur le lequel s'appuie les lanceurs d'applications pour établir les connexions entre plusieurs processus (type d'applications MPI). Avant de démarrer, l'application doit échanger des informations (le couple <IP, port> pour TCP par exemple). Une numérotation des processus existe alors (analogue au rang MPI sur MPI_COMM_WORLD). La PMI se présente souvent sous la forme d'un serveur central de **clefs-valeurs**, mis en place dès le début de l'application. Les différentes instances de programmes (=processus), lancés en parallèle, effectuent des appels vers ce serveur en utilisant une interface commune, aussi appelée l'interface PMI. Pour commencer, vous pourrez trouver dans un dépôt Github créé à cet effet, tout le nécessaire pour démarrer votre projet (https://github.com/besnardjb/aise_pmi). Surtout, vous trouverez dans libpmi/*pmi.h l'*interface client à implémenter. À vous donc de concevoir la PMI, tant sur sa partie serveur que cliente. Pour lancer votre « interface », vous pourrez utiliser le script de lancement (=lanceur) dans `helper/pmirun`.

Pour vous aider, on peut d'abord considérer les suppositions suivantes :

- Les processus lancés ont accès à l'ensemble des variables d'environnement du lanceur. Ceci veut dire que les mêmes variables sont présentes pour le serveur et tous les clients. Le transfert d'informations sera plus simple pour commencer ;
- Le script de lancement définit le nombre de processus à lancer par le biais d'une variable d'environnement qui sera accessible à l'ensemble des processus lancés. Chaque client pourra la récupérer dans son environnement ;
- Le serveur est lancé en premier, les informations de connexion (host,port) sont passées en paramètre du lanceur.
- Il est possible de considérer que le système de fichiers est partagé entre les processus lancés (ce qui est vrai sur une machine locale, et souvent vérifié sur un calculateur). Un transfert d'information peut aussi se faire par ce moyen;
- Le choix de la « couche de transport » à utiliser est libre, selon ce qu'ils jugent le plus intéressant (Système de fichiers, TCP, Socket UNIX, Segment de mémoire partagée, files de messages …);

Exemple: si l'on considère le programme C (gauche) une sortie attendue serait (droite) :

```c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char ** argv){
    fprintf(stderr, "%s / %s server @ %s\n",
            getenv("PMI_RANK"),
            getenv("PMI_PROCESS_COUNT"),
            getenv("PMI_SERVER_ADDR"));
    return 0;
}
```

```
$ ./pmirun 127.0.0.1:900 8 ./a.out

PMI Server : 127.0.0.1:900
Now Launching 8 processes
0 / 8 server @ 127.0.0.1:900
1 / 8 server @ 127.0.0.1:900
2 / 8 server @ 127.0.0.1:900
3 / 8 server @ 127.0.0.1:900
4 / 8 server @ 127.0.0.1:900
5 / 8 server @ 127.0.0.1:900
6 / 8 server @ 127.0.0.1:900
7 / 8 server @ 127.0.0.1:900
```

jbbesnard@paratools.com adamj@paratools.com

Le lanceur fourni vous permettra de vous guider pour démarrer votre projet. **Vous êtes donc encouragés à reposer sur ce qu'il définit dans l'environnement.**

Vous trouverez dans test/, deux cas-tests permettant de valider votre implémentation de l'interface :

- test_values.c **:** Après les initialisations d'usage, chaque processus enregistre des clés auprès du serveur. Une barrière attend alors que tous les processus aient complété cette étape. Ensuite, chaque processus lit les clés de son voisin de rang et vérifie leur valeur. Les valeurs enregistrées sont tirées depuis « rand » (rand() génère toujours la même séquence de nombres pour une « graine » donnée, fixée avec srand()). Ce premier cas-test permet de valider que votre implémentation fonctionne. Prenez donc le temps d'avoir un test fonctionnel, sans SEGV. Faites varier certains paramètres (nombre de clés, nombre de processus) pour vous assurer de la robustesse de votre code. Voici un exemple de sortie pour une implémentation fonctionnelle :

```
$ ./mpirun None 2 ./test_value
PMI Server : None
Now Launching 2 processes
RANK 0 SET rank_0_iter_0 = 1804289383
RANK 0 SET rank_0_iter_1 = 846930886
RANK 0 SET rank_0_iter_2 = 1681692777
0/2 ----- BARRIER -----
RANK 1 SET rank_1_iter_0 = 1804289383
RANK 1 SET rank_1_iter_1 = 846930886
RANK 1 SET rank_1_iter_2 = 1681692777
1/2 ----- BARRIER -----
RANK 1 check for 0 GET rank_0_iter_0 = 1804289383 expected 1804289383
RANK 1 check for 0 GET rank_0_iter_1 = 846930886 expected 846930886
RANK 1 check for 0 GET rank_0_iter_2 = 1681692777 expected 1681692777
RANK 0 check for 1 GET rank_1_iter_0 = 1804289383 expected 1804289383
RANK 0 check for 1 GET rank_1_iter_1 = 846930886 expected 846930886
RANK 0 check for 1 GET rank_1_iter_2 = 1681692777 expected 1681692777
```

- test_perf.c : Avoir une implémentation qui fonctionne ne fait pas tout. Dans ce second cas-test, une mesure de performance est faite sur les appels à la PMI. Ces mesures sont moyennées sur plusieurs répétitions, afin de lisser le bruit de la machine de test. Si vous prenez le temps d'avoir plusieurs couches de transport (fichiers, réseau, mémoire partagée), vous pourrez constater la différence de performances selon le mode choisi, l'objectif étant évidemment de montrer la meilleure performance.Voici un exemple de sortie pour une implémentation PMI basée sur le système de fichier:

```
$ ./pmirun lol 2 ./test_perf
PMI Server : lol
Now Launching 2 processes
Rank 0 sets 16384 keys ..0 / 16384
1000 / 16384
(…)
16000 / 16384
Writing a key takes 13.6048 usec
Rank 1 reads 16384 keys ..0 / 16384
1000 / 16384
(…)
16000 / 16384
Reading a key takes 3.80016 usec
```

Vous avez bien sûr carte blanche pour améliorer ces performances le plus possible en faisant les bons choix d'implémentation.

jbbesnard@paratools.com adamj@paratools.com

Barème

Le barème de cette première partie sera très proche de la répartition suivante et vous permettra d'atteindre facilement la moyenne :

Code	Composant	Description	Notation approximative
A1	**test_values fonctionnel**	Le test des valeurs est fonctionnel	5
A2	**test_values fonctionnel et distribuable**	Le test des valeurs est implémenté d'une manière potentiellement distribuée (plusieurs machines)	2
A3	**test_perf fonctionnel**	Le lancement de test_perf ne mène à aucune erreur	2
A4	**Utilisation d'un serveur**	Votre implémentation repose sur un modèle client serveur (et non par exemple le système de fichier)	1
—	**TOTAL**	—	**10**

Pour aller plus loin, et tenter d'améliorer votre score, voici un certain nombre de fonctionnalités que vous êtes encouragés à explorer. Vous êtes notés sur 25 **pour une note sur 20**, il devrait être facile d'avoir une très bonne note ! De plus, tout bonus vient s'ajouter à ce total de points. Un exemple à 8 points est donné dans le tableau ci-dessous, mais libre à vous d'ajouter les fonctionnalités qui vous intéressent :

Code	Composant	Description	Notation approximative
B5	**Utilisation de sockets**	Reposer sur des sockets pour implémenter le serveur	2
B6	**Utilisation de named pipe**	Reposer sur des « named pipe » pour implémenter le serveur	2
B7	**Utilisation d'un segment SHM**	Reposer sur un segment SHM pour implémenter le serveur	3
B8	**Politique dynamique**	Permettre une adaptation de la couche de transport en fonction de si le processus et local ou distant. Par exemple SHM en local et TCP en distant.	5
B9	**Optimisation de performances**	Démontrer des performances approchant voir même supérieures à l'implémentation sus sytème de fichier présentée ci-dessus. Note en fonction du gain.	3
—	**TOTAL**	—	**15**
C10	**Lancement MPI (BONUS)**	Adapter votre implémentation pour lancer un MPI supportant la PMI1 (sur MPICH 3.2 ou plus ancien car les nouvelles version utilisent une PMI plus complexe).	8
C11	**Autres idées (BONUS)**	Soyez créatifs, points bonus à discretion…	X

jbbesnard@paratools.com adamj@paratools.com

Rendu

Ce travail se fera en binôme et sera à rendre pour le **Lundi 11 Mars 2018, 23:59:59**, il vous sera possible de nous pointer vers un GitHub ou de nous envoyer la tarball à votre convenance. Nous vous fournissons un dépôt avec un squelette de projet pour vous faciliter la vie à l'adresse:

https://github.com/besnardjb/aise_pmi

Vous devrez rendre une archive compressée (lisible sous Linux) nommée selon votre binôme, qui devra contenir au minimum :

- Les **sources** du projet, en langage C ;
- Un **README** au format texte (« Markdown » si possible) à la racine du projet, expliquant dans les grandes lignes votre projet et la manière de le compiler ;
- La **liste des fonctionnalités** que vous pensez avoir implémentées, en listant l'ensemble des codes des grilles ci-dessus (dans le cas de C11, merci de préciser quelle est votre feature) ;

Chaque archive devra nous parvenir avant la date indiquée en première page, de la manière qui vous convient (email, pull request GitHub, serveur de téléchargement type WeTransfer, Torrent…). *Nous vous encourageons vivement à cloner notre dépôt et travailler sur une copie hébergée sur Github*. Une évaluation des projets sera faite le dernier jour (après le devoir sur table), directement par une démo de votre serveur PMI. Vous pourrez utiliser la machine que vous souhaitez, tant que celle-ci est sous Linux. Vous pourrez avoir à nous expliquer chaque ligne de votre programme, afin que nous puissions vérifier que vous avez bien compris leur fonctionnement. Chaque archive sera sommée via sha256sum à réception pour vérifier qu'il n'y a pas d'altération entre la date de rendu et la démo. Nous utilisons des systèmes de détection du plagiat comme celui fourni par http://theory.stanford.edu/~aiken/moss/, il ne sert donc à rien de renommer les variables en copiant le code d'un autre… N'oubliez pas que vous travaillez pour vous et que la connaissance des éléments de base de programmation (Makefile, I/O en C, réseaux) est un élément essentiel pour tout informaticien, surtout s'orientant vers le HPC.

Bon Courage !

jbbesnard@paratools.com adamj@paratools.com