Sujet du Projet Système et Réseau (ETE7-PROJ1, ex-PR204) :

Implémentation d'une bibliothèque de Mémoire Partagée Distribuée (DSM)

Joachim Bruneau-Queyreix, Guillaume Mercier, Philippe Swartvagher

joachim.bruneau-queyreix@bordeaux-inp.fr,
guillaume.mercier@bordeaux-inp.fr,
philippe.swartvagher@bordeaux-inp.fr

2025

1 Introduction

Contexte du projet

Une application *distribuée* (ou *répartie*) est un type d'application structurée avec de multiples processus s'exécutant sur des machines potentiellement différentes (physiquement parlant). Ces différents processus sont amenés à s'échanger des données et la plupart du temps, ces communications sont effectuées via un réseau de communication (par exemple le réseau sous-jacent peut être exploité avec l'interface socket dans Unix mais d'autres solutions existent, aussi bien en termes d'interface que de technologies). Cependant, il est aussi possible de mettre en place des communications qui utilisent le réseau implicitement, de façon transparente pour l'application distribuée. Ainsi, il est possible pour les processus de communiquer en utilisant une zone de leur espace d'adressage, zone qu'il se partagent tous. Bien évidemment, cette dernière n'est pas réellement partagée puisque les divers processus applicatifs s'exécutent sur des machines différentes.

Objectif du projet

Le but de ce projet est de mettre en place une bibliothèque (`libdsm`) permettant de donner l'illusion à une application distribuée l'utilisant que ses processus se partagent effectivement de la mémoire. En sous-main, ces échanges en mémoire se baseront sur des communications réseau (sockets TCP), mais ces dernières seront complètement cachées à l'application distribuées (communications implicites). Ce projet est structuré en deux phases distinctes :

- **Première phase** : le travail consiste à réaliser un programme utilitaire (appelé `dsmexec`) dont le rôle sera non seulement de lancer les processus de l'application répartie sur les différentes machines-cibles mais encore de communiquer à tous ces processus les informations nécessaires pour qu'ils puissent communiquer entre-eux via des sockets TCP (typiquement, un couple $<$ adresse IP, numéro de port $>$).
- **Seconde phase** : le travail consiste mettre en œuvre la bibliothèque elle-même (c'est-à-dire un ensemble de fonctions) qui sera utilisée par les applications distribuées afin que leurs processus puissent communiquer à l'aide d'une zone de mémoire *virtuellement* partagée.

Ces deux phases seront indépendantes l'une de l'autre, mais pour simplifier le développement vous commencerez par implémenter le programme `dsmexec` puis la bibliothèque `libdsm`. Du matériel de départ (et notamment du code) vous sera fourni pour chacune de ces deux phases.

S'il vous est possible de développer le projet sur vos machines personnelles, le déploiement et l'exécution devront être effectués sur les machines de l'école. C'est dans cet environnement uniquement qu'il sera procédé à l'évaluation de votre travail.

2 Principes de fonctionnement de la mémoire partagée distribuée

Pour que les différents processus d'une application distribuée puissent communiquer entre-eux, ils vont avoir besoin d'une plage d'adresses particulière dans leur espace d'adressage respectif. Cette plage d'adresses est *identique* pour tous les processus et sera organisée en un ensemble de morceaux de taille fixe (4ko) appelés *pages mémoire*. Les processus vont donc manipuler ces pages, c'est-à-dire y lire et/ou y écrire des données. Cependant, un *unique processus sera le propriétaire* d'une page à un instant donné. Ce processus propriétaire possède la page dans son espace d'adressage et y accède normalement, tandis que si les autres processus essayent d'y lire ou d'y écrire, cela provoquera une erreur de segmentation (i.e un `segfault`). Quand cet évènement se produit, un signal (numéro `SIGSEGV`, 11) est envoyé au processus qui a causé cette erreur. Le comportement par défaut d'un processus qui reçoit un tel signal est de normalement se terminer.

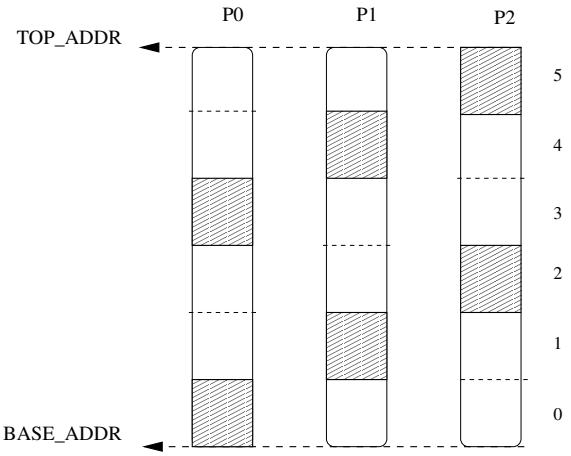

FIGURE 1 – Répartition *initiale* des pages mémoire

Cependant, notre cas est différent. Comme cette page est théoriquement accessible, on considère que cette erreur n'en est pas vraiment une : le signal est pris en compte à l'aide d'un *traitant* adapté qui va, au lieu de terminer le processus :

1. déterminer l'adresse qui a provoqué l'erreur (le *segfault*) et donc la délivrance du signal ;
2. en déduire le bloc de mémoire (ie la page) concerné, qui sera identifié par son *numéro* ;
3. en déduire le processus actuellement propriétaire, car un unique processus est le proprétaire de la page de numéro concerné ;
4. envoyer une requête à ce processus pour qu'il envoie la page au processus demandeur (celui qui a provoqué l'erreur), qui deviendra donc le *nouveau propriétaire* de la page mémoire concernée ;
5. récupérer la page envoyée par le processus propriétaire et l'installer dans l'espace d'adressage du processus demandeur.

NB : Seules les erreurs de segmentation se produisant dans la zone de mémoire partagée seront rattrapées. Les autres erreurs de segmentation ne bénéficieront pas d'un traitement particulier.

L'actuel processus propriétaire devra :

1. mettre à jour ses informations concernant cette page (en particulier l'identité du nouveau processus propriétaire) ;
2. libérer cette page de son espace d'adressage ou a minima indiquer qu'elle n'est plus accessible.

Le processus demandeur devra :

1. recevoir la page via une socket dédiée (cf plus loin) ;
2. l'allouer dans son espace d'adressage ;
3. mettre à jour ses informations concernant cette page ;
4. reprendre son exécution : en ressortant du traitant, le processus exécute à nouveau l'instruction qui a provoqué la faute. Cependant comme ce processus possède désormais la page mémoire (elle fait maintenant partie physiquement de son espace d'adressage) il peut travailler dessus sans que cela ne provoque à nouveau une erreur d'accès mémoire (*segfault*).

NB : Vous réfléchirez à la manière dont ces informations doivent également être répercutées aux autres processus, afin que tous possèdent des informations cohérentes sur l'état du système.

3 Hypothèses sur le projet

Organisation de la mémoire partagée

Les propriétés des pages de la mémoire partagée sont les suivantes :

- le nombre de pages est limité à `PAGE_NUMBER` qui est une constante *modifiable* ;
- chaque page mémoire est identifiée par un numéro. Cette numérotation est linéaire et contigüe avec des valeurs comprises entre 0 et `PAGE_NUMBER`-1) ;
- la taille d'une page est de `PAGE_SIZE` octets, qui est une constante *non modifiable* ;
- la plage des adresses considérée sera comprise entre `BASE_ADDR` et `TOP_ADDR`. **Cette plage d'adresses est identique pour tous les processus de l'application et n'est pas modifiable** ;
- l'allocation *initiale* des pages est faite cycliquement (cf figure 1) ;

La figure 1 montre un exemple d'allocation de départ pour 6 pages réparties entre 3 processus. Le processus de rang 0 possède les pages 0 et 3, le processus de rang 1 possède les pages 1 et 4 et le processus de rang 2 possède les pages 2 et 5. Cependant, chaque processus peut accéder à l'ensemble des pages.

Propriétés des processus d'une application distribuée utilisant `libdsm`

Les processus d'une application distribuée utilisant la bilbiothèque de mémoire partagée possèdent les propriétés suivantes :

- contexte **statique** : le nombre de processus de l'application répartie ne change pas en cours d'exécution. On ne rajoute pas de nouveau processus et aucun processus ne quitte l'application en cours de route ;
- dans l'application, les processus sont identifiés par un numéro unique, appelé *rang* ; ce *rang* n'a de sens qu'au sein d'une application donnée et n'a aucun rapport avec le pid du processus ;
- un processus peut connaître son *rang* à l'aide d'une **variable d'environnement** `DSMEXEC_RANKID` ;
- un processus peut connaître le nombre total de processus de l'application distribuée à l'aide d'une **variable d'environnement** `DSMEXEC_RANKNUM` ;
- les *rangs* des processus sont contigus, avec des numéros compris entre 0 et `DSMEXEC_RANKNUM`-1 ;
- tous les processus sont connectés les uns avec les autres via des sockets TCP.

— chaque processus dispose d'une table stockant les informations sur les pages mémoire (par exemple le rang du propriétaire de la page).
— tous les processus ont accès à un même système de fichier. En particulier, il ne sera pas nécessaire de se poser la question de la disponibilité du programme exécutable à lancer sur les différentes machines ;

4 Première phase : le lanceur d'applications `dsmexec`

4.1 Arguments de la commande `dsmexec`

Avant d'implémenter la bibliothèque `libdsm` proprement dite, il vous faut développer un programme annexe essentiel : `dsmexec`. Son rôle sera de déployer les processus d'une application distribuée sur les différentes machines. Ce programme va prendre en arguments :

– **Premier argument :** un fichier contenant les noms des différentes machines sur lesquelles l'application distribuée doit être déployée. L'organisation de ce fichier a une influence sur la structure de l'application à déployer. Le nombre de lignes dans le fichier correspond au nombre de processus composant l'application distribuée. Chaque ligne contient exactement un nom de machine. Également, l'ordre des machines dans le fichier est une allocation des *rangs* des processus. Le processus de rang 0 sera lancé sur la machine dont le nom apparaît sur la première ligne du fichier et ainsi de suite. Un nom de machine peut apparaître plusieurs fois dans ce fichier.
– **Deuxième argument :** le nom du programme exécutable (ou de la commande shell) à déployer.
– **Arguments suivants :** les éventuels arguments supplémentaires du programme exécutable (ou de la commande shell) à déployer.

4.1.1 Points particuliers concernant les arguments du programme `dsmexec`

1. le nom du fichier contenant les noms des machines est quelconque. Vous veillerez donc à ne pas coder « en dur » ce nom ;
2. Le programme `dsmexec` peut tout à fait s'exécuter sur une machine qui n'est pas listée dans le fichier de machines pris en premier argument ;
3. attention à gérer convenablement les éventuelles lignes vides dans le fichier de machines (c'est-à-dire ne pas les prendre en compte ...) car elles pourraient fausser le comportement de la commande `dsmexec`.

4.1.2 Restrictions concernant les noms des machines

Afin d'éviter certains problèmes au moment du déploiement d'une application répartie, nous vous demandons de respecter certaines restrictions au niveau de l'utilisation des noms de machines que vous allez écrire dans le fichier de de machines.

— le nom `localhost` ne doit pas être présent dans ce fichier ;
— il est préférable de mettre des *Fully qualified domain names (FQDN)* c'est-à-dire des noms incluant le nom de domaine. Si vous tapez la commande `hostname` dans un terminal, vous aurez le nom court de la machine sur laquelle vous vous trouvez tandis que la commande `hostname --long` vous donne le FQDN. Par exemple si je suis sur la machine `montag`, son nom court est `montag` tandis que son FQDN est `montag.pedago.ipb.fr`. NB : une configuration adéquate de ssh (cf paragraphe 4.3.1) peut vous éviter de mettre systématiquement les FQDN dans le fichier de machines.

Une liste de machines disponibles à l'ENSEIRB-MMK pour le projet vous est fournie dans la matériel de de départ (cf. section 4.4).

4.1.3 Exemple de commande

Pour fixer les idées, on prend comme exemple un programme que l'on souhaite déployer sur trois machines (`toto.pedago.ipb.fr`, `tata.pedago.ipb.fr` et `titi.pedago.ipb.fr`). Ce programme appelé `truc`, utilise la bibliothèque `libdsm` et prend trois arguments : `arg1`, `arg2` et `arg3`. Dans ce cas, la ligne de commande correspondante sera la suivante :

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

Avec le fichier `machinefile` qui contient les lignes :

```
toto.pedago.ipb.fr
tata.pedago.ipb.fr
titi.pedago.ipb.fr
```

La commande `dsmexec` procède au déploiement de l'application `truc` : le processus 0 s'exécute sur la machine `toto.pedago.ipb.fr`, le processus 1 sur `tata.pedago.ipb.fr` et le processus 2 sur `titi.pedago.ipb.fr`. Comme ces machines ont accès au même système de fichier, le programme `truc` est *accessible* (i.e visible) sur toutes.

4.2 Fonctions et capacités du lanceur `dsmexec`

Le rôle de `dsmexec` consistant à déployer des applications distribuées entraîne plusieurs conséquences essentielles :

- c'est le programme `dsmexec` qui est chargé de l'attribution des numéros de rang aux processus applicatifs. Plus exactement, cette numérotation est déduite par `dsmexec` de la structure du fichier de machines dont le nom est pris en premier argument par le programme de déploiement.
- `dsmexec` **centralise** les affichages des *sorties standard et d'erreur* de tous les processus applicatifs.
- **`dsmexec` doit être capable de lancer tout type d'applications et de programmes, c'est-à-dire pas uniquement des applications distribuées utilisant la `libdsm`.**
- `dsmexec` est chargé de récupérer les données nécessaires pour les interconnexions réseau entre les processus applicatifs et de les diffuser à tous ces processus. Ces données comprennent notamment les adresses IP/le nom de chaque machine et le numéro de port de la socket d'écoute. **Cependant, la nature de ces informations n'est connue que des processus applicatifs ; elles sont opaques pour le programme `dsmexec`. Également cet échange d'informations n'aura lieu que si l'application déployée utilise la bibliothèque `libdsm`.**

Par exemple, en supposant que l'on exécute le programme `bidule` dont voici le code :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  fprintf(stdout,"Hello World!\n");
  return 0;
}
```

avec la ligne de commande suivante :

```
dsmexec machinefile bidule
```

Alors l'affichage de dsmexec pourra (par exemple) être formaté selon le modèle suivant :

```
[Proc 0 : toto.pedago.ipb.fr : stdout] Hello World !
[Proc 1 : tata.pedago.ipb.fr : stdout] Hello World !
[Proc 2 : titi.pedago.ipb.fr : stdout] Hello World !
```

Le formatage de l'affichage centralisé se présente ici sous la forme :

[*numéro de rang* : *nom de machine* : *flux de sortie*] *affichage du programme*

Bien entendu, il ne s'agit ici que d'un exemple et vous avez toute latitude pour mettre en place le formatage que vous souhaitez.

4.3 Architecture du lanceur `dsmexec`

4.3.1 Création des processus distants avec `ssh`

— Pour créer/lancer des processus sur des machines distantes, le programme `dsmexec` va utiliser la commande `ssh`, avec un appel à une fonction de recouvrement de la famille `exec`.
— Par conséquence, `dsmexec` va devoir créer un ensemble de processus-enfants *locaux* (i.e situés sur la même machine) qui vont faire appel à `exec` pour exécuter une commande `ssh` avec des arguments judicieusement choisis, notamment le nom de la machine distante sur laquelle on veut déployer un processus applicatif (distant), le nom du programme à déployer ainsi que ses arguments.
— Également, toutes **les variables d'environnement nécessaires au bon fonctionnement du système seront transmises via la commande `ssh`** (ex `DSMEXEC_RANKID` et `DSMEXEC_RANKNUM`).

Configuration de `ssh`

— Pour mener à bien ce projet, il vous sera **indispensable** d'être en capacité de vous connecter d'**une machine de l'école à une autre machine de l'école**, avec la commande `ssh` mais sans que la commande `ssh` n'affiche quoi que ce soit. Deux situations sont possibles :

1. Vous possédez déjà une clef SSH pour vous connecter d'une machine de l'école à une autre. Dans ce cas, vous avez juste besoin de configurer ssh pour répondre aux besoins du projet (cf ci-dessous). Également, vous pouvez générer une nouvelle clef si vous le désirez. Il sera possible par la suite de choisir la clef `ssh` à utiliser via l'option `-i` de la commande `ssh`. Si votre clef ssh ne possède pas de passphrase, vous **devez** en rajouter une avec la commande `ssh-keygen -p`.
2. Si vous ne possédez pas une telle clef SSH, alors il vous faudra en générer une. Des informations sont disponibles à cet URL `https://thor.enseirb-matmeca.fr/ruby/docs/teaching/authentication`

— Une fois la clef créée, il ne faut pas oublier de rajouter la clef **publique** dans la liste des clefs autorisées. Par exemple, si je crée un couple de clefs `ma_jolie_clef` (clef privée) et `ma_jolie_clef.pub` (clef publique), cet ajout sera fait ainsi :

```
cat ma_jolie_clef.pub >> $HOME/.ssh/authorized_keys
```

— Dans tous les cas, il vous faudra configurer ssh afin que les connexions se fassent de façon silencieuse (i.e sans que la commande `ssh` affiche quoi que ce soit au moment d'une demande de connexion). Voilà ce qu'il faut rajouter dans votre fichier `$HOME/.ssh/config` sur votre compte ENSEIRB-MMK.

```
# Permet d'avoir les FQDN des machines
CanonicalDomains pedago.ipb.fr
CanonicalizeHostname yes
CanonicalizeFallbackLocal yes

AddKeysToAgent yes

Host *.pedago.ipb.fr
ForwardAgent yes
StrictHostKeyChecking no #ou accept-new
ForwardX11 yes
```

— **Nous ne vous demandons pas de créer des clefs SSH entre votre machine personnelle et les machines de l'école.**

4.3.2 Redirection des flux d'entrée/sortie

Étant donné que le programme `dsmexec` est chargé de centraliser l'affichage des processus distants (cf 4.2), il faut trouver un moyen de récupérer cet affichage. cette récupération va se faire en deux étapes :

- **Première étape :** le flux des processus distants sera récupéré par les processus-enfants locaux *automatiquement* grâce à la commande `ssh`. Vous n'avez donc rien à faire de particulier pour cette étape ;
- **Seconde étape :** il va falloir que le processus-parent `dsmexec` récupère l'affichage de tous ses processus-enfants locaux. Cette récupération sera effectuée à l'aide de tubes de communication. Plus exactement, une *paire* de tubes sera utilisée : un tube pour récupérer les informations de sortie standard (`stdout`) et un autre pour récupérer les informations de sortie standard d'erreur (`stderr`). **Les processus-enfants locaux redirigeront leur deux flux de sortie vers les extrémités en écriture de chaque tube dédié.**

4.3.3 Protocole d'échange des informations de connexion

Dans le cas où `dsmexec` déploie une application utilisant la bibliothèque `libdsm`, tous les processus applicatifs sont capables de communiquer les uns avec les autres via des sockets TCP. Il est donc indispensable – pour établir toutes ces connexions – que chaque processus dispose des informations de connexions des autres processus (par exemple, adresse IP et numéro de port). C'est `dsmexec` qui est chargé de récupérer ces informations puis de les envoyer à tous les processus distants. Cependant, ainsi qu'indiqué plus haut, la nature de ces informations est **opaque** pour `dsmexec`. Ainsi ces informations sont caractérisées par les deux champs suivants dans la structure `struct remote_proc` :

```
struct remote_proc {

/* ... */

void   *connect_info;
size_t  connect_info_size;

/* ... */

}
```

Donc, le protocole d'échange des informations de connexion sera le suivant pour le programme `dsmexec` :

1. récupération du rang du processus distant (rappel : ce rang est codé sur un `int`);
2. récupération de la taille des informations de connexion (et stockage dans le champ `connect_info_size`). Cette information est de type `size_t` ;
3. allocation du champ `connect_info` de la structure `struct remote_proc`;
4. récupération des informations proprement dites, avec la taille adéquate

Afin de tester si votre protocole est opérationnel, vous pourrez utiliser le programme de test `data_exchange`, disponible dans le répertoire `binaries` qui implémente ce protocole, mais du côté des processus applicatifs.

Attention, ce programme utilise quatre variables d'environnement nécessaires à son fonctionnement. Ces variables sont :

1. Le nombre de processus : `DSMEXEC_RANKNUM`
2. Le rang du processus distant : `DSMEXEC_RANKID`
3. Le nom de la machine d'exécution de dsmexec : `DSMEXEC_HOSTNAME`
4. Le numéro de port de dsmexec : `DSMEXEC_PORTNUM`

L'affichage de ce programme est par exemple (dans le cas de 4 processus distants, tous lancés sur la même machine appelée Palamède) :

```
[Proc 1 : Palamede : stdout]: Info from dsmexec : I'm process 1 / 4 processes total
[Proc 1 : Palamede : stdout]: Struct size is 136
[Proc 1 : Palamede : stdout]: [1] got connection info from dsmexec
[Proc 1 : Palamede : stdout]: [1] Process 0 is on machine: Palamede (port #: 36693)
[Proc 1 : Palamede : stdout]: [1] Process 1 is on machine: Palamede (port #: 55565)
[Proc 1 : Palamede : stdout]: [1] Process 2 is on machine: Palamede (port #: 60773)
[Proc 1 : Palamede : stdout]: [1] Process 3 is on machine: Palamede (port #: 35757)
[Proc 0 : Palamede : stdout]: Info from dsmexec : I'm process 0 / 4 processes total
[Proc 0 : Palamede : stdout]: Struct size is 136
[Proc 0 : Palamede : stdout]: [0] got connection info from dsmexec
[Proc 0 : Palamede : stdout]: [0] Process 0 is on machine: Palamede (port #: 36693)
[Proc 0 : Palamede : stdout]: [0] Process 1 is on machine: Palamede (port #: 55565)
[Proc 0 : Palamede : stdout]: [0] Process 2 is on machine: Palamede (port #: 60773)
[Proc 0 : Palamede : stdout]: [0] Process 3 is on machine: Palamede (port #: 35757)
[Proc 2 : Palamede : stdout]: Info from dsmexec : I'm process 2 / 4 processes total
[Proc 2 : Palamede : stdout]: Struct size is 136
[Proc 2 : Palamede : stdout]: [2] got connection info from dsmexec
[Proc 2 : Palamede : stdout]: [2] Process 0 is on machine: Palamede (port #: 36693)
[Proc 2 : Palamede : stdout]: [2] Process 1 is on machine: Palamede (port #: 55565)
[Proc 2 : Palamede : stdout]: [2] Process 2 is on machine: Palamede (port #: 60773)
[Proc 2 : Palamede : stdout]: [2] Process 3 is on machine: Palamede (port #: 35757)
[Proc 3 : Palamede : stdout]: Info from dsmexec : I'm process 3 / 4 processes total
[Proc 3 : Palamede : stdout]: Struct size is 136
[Proc 3 : Palamede : stdout]: [3] got connection info from dsmexec
[Proc 3 : Palamede : stdout]: [3] Process 0 is on machine: Palamede (port #: 36693)
[Proc 3 : Palamede : stdout]: [3] Process 1 is on machine: Palamede (port #: 55565)
[Proc 3 : Palamede : stdout]: [3] Process 2 is on machine: Palamede (port #: 60773)
[Proc 3 : Palamede : stdout]: [3] Process 3 is on machine: Palamede (port #: 35757)
```

Notez bien que chez vous, la taille de la structure pourra être différente, tous comme les numéros de port (bien entendu), ainsi que le formatage.

4.3.4 Vue d'ensemble de l'architecture du système

La figure 2 vous montre un exemple avec trois processus distants lancés sur les machines `toto`, `tata` et `titi`. Pour rappel, le processus `dsmexec` peut communiquer (cf 4.2) pendant le déploiement avec tous

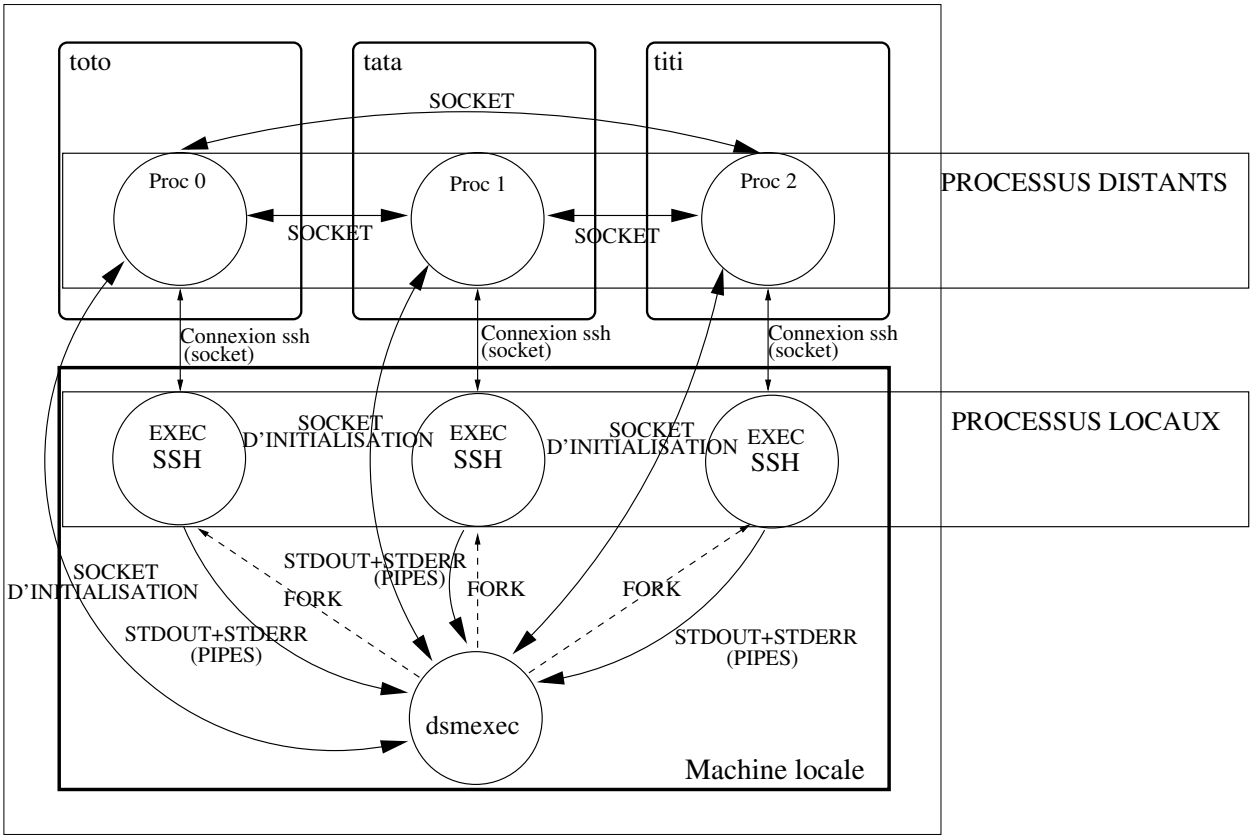

FIGURE 2 – Interactions entre processus distants, processus locaux et `dsmexec`

les processus distants via des sockets temporaires que l'on pourra fermer une fois cette phase terminée. Ces communications n'auront lieu que dans le cas où les processus distants ont besoin d'établir des interconnexions c'est-à-dire s'ils utilisent la bilbiothèque `libdsm`. Ces communications sont inutiles dans les autres cas.

Ensuite, le processus `dsmexec` attend que des données (des *caractères*) arrivent sur les tubes dédiés aux flux sur `stdout` et `stderr` et les affiche au fur et à mesure de leur disponibilité, avec un formatage correspondant à celui de l'exemple montré en Section 4.2.

4.4 Organisation du matériel de départ

Le matériel fourni pour cette première phase est localisé dans le répertoire `Phase1` dans vos dépôts Git respectifs. Ce répertoire contient les éléments suivants :

```
- binaries
  |____ libhelpers.a ------------> | bibliothèque contenant des versions
  |                                | corrigées de certaines fonctions
  |____ data_exchange -----------> | programme de test pour vérifier que
                                   | le protocole d'échange des infos de
                                   | connexion est correctement implémenté
                                   | côté dsmexec

- dsmexec.c ---------------------> | squelette du programme dsmexec (main)
                                   | organisé en étapes principales

- examples ----------------------> | programmes d'exemple utilisables
  |____ basic.c                   | (vous rajouterez vos propres
  |____ calcul.c                  | programmes d'exemple dans ce répertoire)
  |____ hello.c
  |____ Makefile
  |____ trick_hello.c

- include -----------------------> | fichiers d'en-tête avec les définitions
  |____ common_services.h         | de constantes, de structures de données et
  |____ helpers.h                 | de prototypes de fonctions

- machine_file ------------------> | fichier de machines d'exemple

- machines_list.txt -------------> | Une liste de noms de machines (par salle)

- Makefile ----------------------> | Makefile principal pour compiler la phase 1
                                   | utilisant tous les Makefiles des répertoires

- services
  |____ childproc_management.c --> | gestion des processus-enfants
  |____ cleanup_management.c ----> | liberation des ressources globales
  |____ file_management.c -------> | lecture du fichier de machines
  |____ io_management.c ---------> | gestion des tubes et des I/O
  |____ Makefile
  |____ network_management.c ----> | fonctions reseau
  |____ spawn_management.c ------> | creation des processus-enfants + ssh
```

4.4.1 Organisation du fichier `dsmexec.c`

Le fichier `dsmexec.c` contient le squelette du lanceur et est organisé en 3 étapes principales successives.

Étape 1 : lecture du fichier de machines et récupération des informations importantes : nombre total de processus à déployer et noms des machines distantes. (cf exemple 4.1.3)

Étape 2 : création des processus-enfants locaux et des processus distants avec ssh ; création des tubes de communications avec les processus-enfants locaux et redirections des flux `stdout` et `stderr` des processus-enfants. Le processus `dsmexec` — quant à lui — ne redirige rien du tout.

Étape 3 : gestion des entrées-sorties avec les tubes ; récupération de l'affichage des processus-distants et affichage centralisé formaté.

La gestion de la terminaison des processus-enfants (fonction `childprocs_mgmt_setup`) n'est pas vraiment une étape et peut être placée à plusieurs endroits en fonction de la technique choisie. De même, la création de la socket d'écoute (fonction `socket_creation`) n'est pas vraiment une étape et peut être placée *a priori* n'importe où avant l'étape 2.

4.4.2 Fonctions disponibles dans la bibliothèque `libhelpers`

Pour chacune de ces étapes principales, une fonction implémentant le service est fournie :
- Étape 1 : `file_management_correction`
- Étape 2 : `spawn_local_procs_correction`
- Étape 3 : `wait_for_all_io_correction`

Ces fonctions vous permettent d'implémenter n'importe quelle étape sans avoir besoin qu'une étape précédente soit totalement fonctionnelle. Ainsi, il sera possible de répartir au maximum le travail entre les membres du binôme.

La fonction de création de la socket d'écoute possède également sa version corrigée : `socket_creation_correction`.

Enfin, il existe dans la bibliothèque `libhelpers.a` une fonction permettant de tester si la commande à déployer avec `dsmexec` utilise la bibliothèque `libdsm.a` : `check_dsm_use_status`.

Tous les prototypes de ces fonctions se trouvent dans le fichier `include/helpers.h`. Merci de vous y référer pour comprendre leur fonctionnement.

4.4.3 Compilation des sources

Pour compiler les sources, vous **devez utiliser le fichier `Makefile` fourni**. Cette compilation s'effectue en tapant la commande `make install` qui crée un répertoire supplémentaire appelé `bin`, contenant tous vos programmes exécutables : le lanceur `dsmexec` ainsi que tous les programmes d'exemple présents dans le répertoire `examples`. Ce répertoire `bin` est recréé chaque fois qu'il est effacé (par exemple avec la commande `make distclean`).

4.5 Éléments modifiables et non-modifiables

1. Afin de garantir que votre programme `dsmexec` soit compatible avec les fonctions corrigées fournies dans la bibliothèque `libhelpers`, certains éléments ne sont pas *a priori* modifiables (cf les commentaires dans le fichier `include/common_services.h` :
   — la constante `MAX_STR` n'est pas modifiable ;
   — lé début de la structure `struct remote_proc` n'est pas modifiable.
   Si vous faites le choix de ne **pas** utiliser les fonctions corrigées alors les deux éléments listés ci-dessus deviennent modifiables.
2. Également la *structure* des sources n'est pas modifiable : merci de ne pas changer les noms des fichiers ni les répertoires présents.
3. Les fichiers `Makefile`, et en particulier les flags de compilation ne sont *pas modifiables* pour rendre la compilation plus permissive.
4. Le nom de la variable d'environnement `DSM_BIN` est imposé (cf paragraphe suivant).

Dans le doute, demandez aux encadrants si ce que vous souhaitez modifier est modifiable ou non...

4.6 Variables d'environnement et déploiement d'applications

Vous allez devoir définir une variable d'environnement appelée **`DSM_BIN` dont le nom est imposé**. Cette variable contient le nom du chemin vers le répertoire `bin` de votre projet où sont copiés tous les exécutables compilés (aussi bien pour la phase 1 que pour la phase 2). Cette définition va permettre d'éviter d'avoir à indiquer des noms absolus pour les programmes d'exemple et pour la commande `dsmexec`. Également la commande exécutée par `ssh` aura pour répertoire de travail votre `$HOME` donc ne pourra pas trouver le programme à déployer en cas d'utilisation de noms relatifs (plus courts). La définition et l'utilisation de `DSM_BIN` permet de régler cet autre problème.

Afin que tout fonctionne correctement (ie afin que le système trouve tout seul le bon chemin vers les exécutables du projet) le chemin désigné par `DSM_BIN` **doit également être copié** dans la variable d'environnement `PATH`.

Pour ce faire tout ceci, vous éditerez votre fichier `$HOME/.bashrc` et y placerez les définitions suivantes :

```
export DSM_BIN=/mon/chemin/vers/mon/projet/bin
export PATH=$DSM_BIN:$PATH
```

N'oubliez pas de sourcer votre fichier `$HOME/.bashrc` après l'avoir édité !

Vous pouvez tester ensuite si la manœuvre a fonctionné en tapant les commandes :

```
echo $DSM_BIN
```

ou

```
which dsmexec  # si vous avez déjà compilé dsmexec
```

afin de vérifier que votre définition des chemins est correcte.

5 Phase 2 : implémentation de la `libdsm`

Une fois votre lanceur `dsmexec` opérationnel, vous allez mettre en place la bibliothèque `libdsm`. Dans cette phase également, une ébauche de code est fournie pour commencer à travailler. Lisez bien les commentaires présents dans le fichier `dsm.c`, en particulier ceux dans la fonction `dsm_init`.

5.1 Interface de la bibliothèque

L'interface de la bibliothèque `libdsm` comprend des fonctions et des variables.

5.1.1 Fonctions

La bibliothèque à développer est assez réduite puisqu'elle ne va contenir que deux fonctions (*a priori*) :

1. `char *dsm_init(int argc, char *argv[]) ;`
   La fonction `dsm_init` renvoie un pointeur : c'est le pointeur marquant le début de la zone de mémoire distribuée. **Par ailleurs, la fonction `dsm_init` initialise à 0 l'ensemble des pages mémoire allouées.**
2. `void dsm_finalize( void ) ;`
   La fonction `dsm_finalize` permet de libérer les ressources éventuellement allouées par la bibliothèque au moment de l'initialisation.

5.1.2 Variables

Également, cette interface repose sur deux variables utilisables dans les applications :

1. `dsm_node_id` : il s'agit du rang du processus dans l'application distribuée ;
2. `dsm_node_num` : il s'agit du nombre de processus de l'application distribuée.

5.2 Travail à effectuer

La suite du travail va consister à :

— mettre en place le bon traitant de signal pour rattraper les erreurs de segmentation (conseil : lisez bien le manuel pour la fonction `sigaction`)

— mettre en place les envois/réceptions de requêtes. En ce qui concerne les requêtes, nous pouvons en identifier au moins quatre types :

1. les requêtes pour demander une page manquante ;
2. les requêtes pour envoyer une page à un processus demandeur ;
3. les requêtes pour indiquer qu'un processus a terminé le programme (i.e il est dans la fonction `dsm_finalize`) ;
4. les requêtes de mise à jour des informations de la table des pages.

— garantir que les informations concernant les pages (état, propriétaire, etc.) sont bien bien cohérentes d'un processus à l'autre.

5.3 Organisation du matériel de départ

Le matériel fourni pour cette seconde phase est localisé dans le répertoire `Phase2` dans vos dépôts Git respectifs. Ce répertoire contient les éléments suivants :

```
- dsm.c ---------------> | squelette des fonctions de la libdsm

- dsm.h ---------------> | interface de la libdsm (non-modifiable)

- examples ------------> | programmes d'exemple utilisables
  |____ double_touch.c   | (vous rajouterez vos propres
                         | programmes d'exemple dans ce répertoire)

- include -------------> | fichiers d'en-tête internes pour l'implémentation
  |____ dsm_impl.h       | de la libdsm. Les programmes utilisant la libdsm
                         | n'ont pas accès aux définitions dans ce fichier

- Makefile ------------> | Makefile pour compiler la bibliothèque et les exemples

- network.c -----------> | fonctions réseau utilisées en interne dans la libdsm
```

Le matériel fourni est composé des fichiers suivants :

— un fichier `dsm.h`, qui contient les prototypes des fonctions de la `libdsm` utilisées par les programmes d'exemple. C'est dans ce fichier que se trouve l'*interface* de la bibliothèque de DSM.

— un fichier `dsm.c` qui contient l'implémentation des fonctions de DSM qui peuvent être utilisées par les programmes d'exemple. Au besoin, vous pourrez y déclarer des fonctions non directement utilisables par les programmes d'exemple (i.e des fonctions `static`). Vous allez donc travailler essentiellement sur ce fichier `dsm.c` pour compléter le projet.

— Un `Makefile` permettant de compiler (commande : `make` tout court) à la fois la bibliothèque `libdsm` ainsi que les programmes d'exemple dont les sources sont localisées dans le répertoire `examples`. **Attention : les exécutables seront déplacés dans le répertoire `$DSM_BIN`, c'est-à-dire dans le répertoire `bin` de la Phase 1.**

Par ailleurs, **une version fonctionnelle de dsmexec vous est fournie** : l'exécutable se trouve dans le répertoire `Phase1/binaries` de votre dépôt Git. Cette version possède quelques options qui pourront éventuellement vous être utiles pour cette seconde phase (tapez la commande `dsmexec -h` pour obtenir la liste des options actuellement supportées).

5.4 Exemple : le programme `double_touch`

Comment tout cela fonctionne-t-il en pratique ? Examinez le programme suivant (dont on suppose qu'il s'exécute sur des machines différentes) :

```
#include "dsm.h"

int main(int argc, char **argv)
{
  char *pointer = dsm_init(argc,argv);
  char *current = pointer;
  int   value;

  fprintf(stdout,"[Proc %i] base address for shared mapping is: %p\n",
                dsm_node_id, pointer);

  if(0 == dsm_node_id){
    *((int *)current) += 4;
    value = *((int *)current);
    printf("[Proc %i] integer value: %i\n", dsm_node_id, value);
  } else if(1 == dsm_node_id){
    *((int *)current) += 8;
    value = *((int *)current);
    printf("[Proc %i] integer value: %i\n", dsm_node_id, value);
  } else {
    printf("[Proc %i] I'm not part of this \n", dsm_node_id);
  }

  fprintf(stdout,"[Proc %i] Finalizing ...", dsm_node_id);
  dsm_finalize();
  fprintf(stdout,"done\n");

  return 1;
}
```

— ❄ —