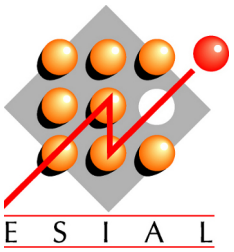
ESIAL
Université Henri Poincaré - Nancy 1
Campus Aiguillettes - 193, avenue Paul Muller
CS 90172 - 54602 Villers-lès-Nancy

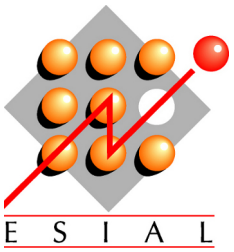


Interception système pour la capture et le rejeu de traces

Rapport de stage - 2ème année
LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications)

Marion GUTHMULLER
2009-2010

ESIAL
Université Henri Poincaré - Nancy 1
Campus Aiguillettes - 193, avenue Paul Muller
CS 90172 - 54602 Villers-lès-Nancy



Interception système pour la capture et le rejeu de traces

Rapport de stage - 2ème année
LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications)
Campus Scientifique - BP 239 - 54506 VANDOEUVRE-LES-NANCY Cedex

Effectué du 07 Juin 2010 au 03 Septembre 2010

Marion GUTHMULLER
Encadrant Industriel : Martin QUINSON
Encadrant Universitaire : Hervé PANETTO

Remerciements

Je tiens à remercier tout particulièrement Martin QUINSON et Lucas NUSSBAUM, chercheurs dans l'équipe AlGorille au LORIA, qui m'ont permis d'effectuer ce stage, d'acquérir de nouvelles compétences mais aussi de connaître le monde de la recherche informatique publique.

Je remercie également l'ensemble de l'équipe AlGorille qui a su m'accueillir dans les meilleures conditions et m'a permise de m'intégrer parfaitement durant ces 10 semaines de stage.

Je souhaite aussi remercier plus particulièrement le «mec génial» du bureau B118 qui n'a cessé de nous faire profiter de ses immenses qualités.

Enfin, je voudrais remercier plus largement le LORIA.

Table des matières

Introduction

De plus en plus populaires, les applications distribuées sont devenues plus complexes, plus grandes et ont dû faire face à davantage de contraintes et exigences. Cependant, les méthodes utilisées pour leur évaluation n'ont pas vraiment évolué. La plupart des applications sont évaluées en utilisant les plateformes expérimentales privées, comme les petits clusters dans des laboratoires ou des universités. Toutefois, ces plateformes ne fournissent qu'une exemple des conditions environnementales et même avec des plateformes à plus grande échelle, il est difficile d'évaluer correctement avec une seule plateforme sur une large gamme de conditions.

Avec le projet SimGrid, qui a débuté en 1999, les chercheurs et développeurs ont pour objectif de faciliter la recherche dans ce domaine de la programmation d'applications distribuées et parallèles sur les plate-formes de calcul distribué à partir d'un simple réseau. Il s'agit donc de fournir un outil avec les fonctionnalités de base pour la simulation de ces applications hétérogènes en environnement distribué. Au sein de ce projet, on retrouve le projet ***Simterpose*** dont l'objectif est de permettre l'émulation d'une application en utilisant un simulateur, en l'occurence, SimGrid. Mon stage a donc porté sur le développement de ce projet au sein de SimGrid.

Pour vous expliquer mon travail réalisé durant ces dix semaines, je commencerai par vous présenter le contexte dans lequel j'ai effectué mon stage, puis, je procèderai à une présentation détaillée de ma contribution dans le projet ***Simterpose***. Je terminerai par un bilan sur l'avancement actuel du projet et sur les perspectives envisagées.

1 Institution d'accueil

1.1 Le LORIA

Le LORIA[1], Laboratoire Lorrain de Recherche en Informatique et ses Applications, est une Unité Mixte de Recherche commune à plusieurs établissements :

- CNRS, Centre National de Recherche Scientifique
- INPL, Institut National Polytechnique de Lorraine
- INRIA, Institut National de Recherche en Informatique et en Automatique
- UHP, Université Henri Poincaré, Nancy 1
- Nancy 2, Université Nancy 2

L'UMR LORIA et le centre de recherche INRIA Nancy - Grand Est ont des services communs.

Le LORIA est un laboratoire de plus de 450 personnes organisé :

- en équipes de recherche , qui regroupent :
  - 150 chercheurs et enseignants-chercheurs
  - un tiers de doctorants et post-doctorants
  - des ingénieurs
- en services de support à la recherche, qui regroupent :
  - des ingénieurs
  - des techniciens
  - des personnels administratifs (gestionnaires, juristes, assistantes, ...)

C'est aussi chaque année :

- une trentaine de chercheurs étrangers invités
- des coopérations internationales avec des pays des cinq continents
- une quarantaine de contrats industriels

Les équipes de recherche du LORIA articulent leur travail autour de 4 thématiques scientifiques et une thématique transversale :

- Traitement automatique des langues et des connaissances
- Fiabilité et sécurité des systèmes informatiques
- Image et géométrie
- Perception, action et cognition
- Thématique transversale : informatique et science du vivant

Ses principaux domaines d'application sont :

- Réseaux, Internet et Web
- Réalité virtuelle
- Bioinformatique
- Sécurité des systèmes informatiques
- Robotique
- Santé

1.2 L'équipe AlGorille

Pour ce stage, j'ai été accueillie dans l'équipe AlGorille[2] (Algorithmes pour la Grille) dont le domaine d'application se situe au niveau des Réseaux, systèmes et services et calul distribué. Leur thème principal de recherche traite du calcul distribué et applications à très haute performance.

Leur travail repose essentiellement sur 3 axes de recherches :

- La gestion transparente des ressources : ordonnancement de tâches, migration de calcul, transfert, distribution et redistribution de données ;
- La structuration des applications pour le passage à l'échelle : modélisation de la localité et de la granularité ;
- La validation expérimentale : reproductibilité, possibilité d'extension et l'applicabilité des simulations, des émulations et des expériences *in situ*.

Pour cela, leur méthodologie s'appuie sur 3 points : la modélisation, la conception et l'ingénierie des algorithmes.

L'équipe AlGorille est composée de 4 chercheurs permanents et dirigée par Jens GUSTEDT. Elle accueille également actuellement 2 ingénieurs, 1 Post-Doc et 6 étudiants en thèse.

2 Travail réalisé

2.1 Présentation du sujet

Le principe de l'application distribuée est de mettre en place une architecture logicielle permettant l'exécution d'un programme sur plusieurs ordinateurs communiquant entre eux via des réseaux locaux ou par Internet en utilisant le protocole IP.

Une architecture N-Tiers (ou distribuée) typique est celle constituée d'un client, d'un serveur d'application et d'un serveur de base de données. Certains traitements s'exécutent sur le client (traitements liés à la gestion de l'interface graphique notamment), d'autres sur le serveur d'application et enfin sur le serveur de base de données (requêtes SQL).

2.1.1 Évaluation des applications distribuées et calcul scientifique

De plus en plus populaires, les applications distribuées sont devenues incontournables. En effet, grâce aux applications distribuées, on peut partager des ressources qui ne se trouvent pas au même endroit ou sur la même machine ; elles deviennent donc elles-mêmes distribuées.

Grâce à cela, on obtient tout d'abord une augmentation dans la quantité des ressources disponibles. Le modèle peer-to-peer (P2P) est un exemple de réussite des architectures distribuées où chaque ordinateur est à la fois serveur de données et client des autres. Ce modèle s'applique donc au partage des ressources. L'application la plus répandue du P2P est actuellement le partage de fichiers avec par exemple BitTorrent ou Emule. Dans ce cas, chaque client ayant téléchargé l'information devient aussitôt serveur. Ainsi, les clients eux-mêmes servent les données déjà reçues aux nouveaux destinataires. Le coût et la charge de la distribution des données sont donc considérablement réduits.

Une seconde conséquence de la distribution des ressources est une augmentation de la puissance de calcul. Les ordinateurs d'aujourd'hui sont tellement puissants que la majeure partie du temps, une grande partie de leur processeur est disponible pour effectuer des calculs. Si on exploite cette disponibilité en même temps pour chaque ordinateur, on peut obtenir une puissance de calcul plus importante que les plus gros super-ordinateurs. Ainsi, le projet BOINC[3] a saisi cette opportunité pour créer un gigantesque parc informatique réparti dans le monde afin d'utiliser cette immense puissance de calcul totale pour effectuer des calculs trop complexes pour être réalisés dans un laboratoire. Le projet BOINC demande donc au particulier de permettre l'usage de la puissance de calcul dont il n'a pas immédiatement besoin pour contribuer à la recherche sur le repliement de protéine (Folding@Home[4]) et même la recherche d'intelligence extra-terrestre par analyse de spectre électromagnétique (SETI@home[5]).

Enfin, il est possible de voir des applications telles que Wikipédia[6] comme des systèmes distribués où les connaissances jouent le rôle des ressources distribuées.

Pour exploiter ces ressources distribuées et leurs avantages, il faut donc développer des applications distribuées. Cependant, ce n'est pas toujours très simple du fait de leur complexité et hétérogénéité mais aussi à cause de contraintes et exigences plus fortes.

En effet, les applications distribuées doivent tout d'abord prendre en compte l'hétérogénéité des ressources connectées au niveau logiciel/matériel : systèmes d'exploitation différents, logiciels différents, architectures matérielles différentes (PC, processeurs ARM dans les smartphones ou les ebook readers, ...). On est donc face à une difficulté en terme d'intéropérabilité.

Il faut également prendre en compte les différences de performances possibles (connexions Internet très haut débit vs EDGE/3G sur téléphones portables, vitesses de calcul différentes, ...) tout en exploitant au mieux les ressources.

De plus, lors du développement de ces applications, il faut prendre en compte "l'attitude" des utilisateurs. Aujourd'hui, chaque utilisateur cherche à maximiser son revenu de manière "égoiste" (télécharger plus vite en uploadant moins, augmenter son score à seti@home, ...). Il faut, par conséquent, mettre au point des algorithmes qui résistent à des utilisateurs malveillants, ou à une collusion d'utilisateurs malveillants.

Enfin, il faut développer des logiciels qui fonctionnent même à une très grande échelle (un nombre important de machines (noeuds) participantes) . Ce point concerne plus particulièrement les réseaux P2P où il peut y avoir des millions de noeuds pour certains.

Pour répondre à ces contraintes, il est nécessaire de développer des méthodes spécifiques permettant l'étude des applications distribuées.

2.1.2 Différentes techniques d'études

Pour être en mesure de déterminer le comportement d'une application face à un large éventail de conditions environnementales, il est possible de l'exécuter directement sur la plateforme réelle. Cependant pour tester une application sans avoir à trouver des plates-formes fournissant ces éléments, les développeurs et les chercheurs ont utilisé 2 autres techniques à la fois différentes et complémentaires : la simulation et l'émulation.

2.1.2.1 Exécution sur plateforme réelle

La première technique pour étudier une application distribuée est de l'exécuter directement sur une plateforme réelle telle que PlanetLab [1] ou Grid'5000 [2] par exemple : on parle alors d'expérience *in situ*. Toutefois, une telle expérience peut être lourde à mettre en oeuvre du fait de son déroulement (écrire l'application, construire entièrement la plateforme et exécuter l'application sur cette dernière). De plus, il est difficile de reproduire une expérience *in situ* dans des conditions comparables car les plateformes sont partagées avec d'autres utilisateurs, qui influent directement sur les conditions expérimentales.

1. Créé en 2002, PlanetLab est un réseau d'ordinateurs comportant plus de 900 noeuds répartis sur 460 sites, utilisé en tant que plateforme d'essais pour la recherche orientée réseaux et systèmes distribués.

2. Grid'5000 est une plateforme expérimentale, répartie sur 10 sites (9 en France et 1 au Brésil), destinée à l'étude des systèmes distribués et parallèles à grande échelle.

2.1.2.2 Simulation

La simulation désigne un procédé selon lequel on exécute un programme sur un ordinateur en vue de simuler, par exemple, un phénomène physique complexe. Elle repose sur la mise en oeuvre de modèles théoriques et sert à étudier le fonctionnement et les propriétés d'un système modélisé ainsi qu'à en prédire son évolution. Au lieu d'utiliser l'application elle-même, elle est modélisée, et interagit avec un modèle de l'environnement.

La simulation a été l'objet de beaucoup de travail, et des simulateurs génériques ciblant les applications distribuées ont été développés. Il est possible de simuler des applications distribuées avec plusieurs milliers de nœuds, face à des conditions très diverses, avec une modélisation très précise de tous les paramètres. Elle apparaît donc comme une solution face à la plupart des problèmes des expérimentations *in situ* : mise en oeuvre rapide et facile, reproductibilité parfaite des expériences et possibilité d'explorer de nombreux scénarios expérimentaux en temps raisonnable.

Toutefois, la principale limitation de la simulation vient de son principe de base : un modèle de l'application est utilisée au lieu de l'application réelle. Ceci impose donc souvent de coder les applications à évaluer dans un formalisme particulier. Ainsi, la simulation ne permet pas de trouver des bugs ou problèmes de mise en œuvre dans l'application elle-même.

2.1.2.3 Émulation

L'émulation consiste à substituer un élément de l'environnement de l'application par un logiciel. L'émulateur reproduit le comportement d'un modèle dont toutes les variables sont connues.

Dans l'évaluation des applications distribuées, l'émulation est une approche intermédiaire visant à résoudre les limitations de la simulation : en émulant l'exécution de l'application sur une plate-forme virtuelle (simulée), il est possible d'exécuter l'application réelle sur une large gamme de conditions, dans un environnement entièrement contrôlé. De plus, cela évite de devoir développer 2 versions de l'application , l'une étant adaptée à l'expérimentation sur simulateur tandis que la seconde est utilisée en production. L'émulation d'applications réparties consiste à intercepter les actions de l'application et à reporter ces actions dans le simulateur. Dans le cas d'une émulation *offline*, on sauvegarde ces actions sur disque, puis on les rejoue après coup dans le simulateur. Tandis que dans une émulation *online*, on reporte immédiatement ces actions dans le simulateur, puis on retarde l'application du temps calculé par le simulateur.

2.1.3 Objectif final à long terme : émuler avec SimGrid

Le projet SimGrid[7] a débuté en 1999 pour permettre l'étude d'algorithmes d'ordonnancement sur des plateformes hétérogènes. Il s'agit d'un outil qui fournit des fonctionnalités de base pour la simulation d'applications distribuées hétérogènes en environnements distribués.

L'objectif spécifique du projet est de faciliter la recherche dans le domaine de la programmation d'applications distribuées et parallèles sur des plates-formes de calcul distribué à partir d'un simple réseau allant du poste de travail aux grilles de calcul.

SimGrid[8] se décompose actuellement en 8 environnements construits au dessus d'un

noyau de simulation unique, et déjà intégrés, plus 1 en cours de programmation.

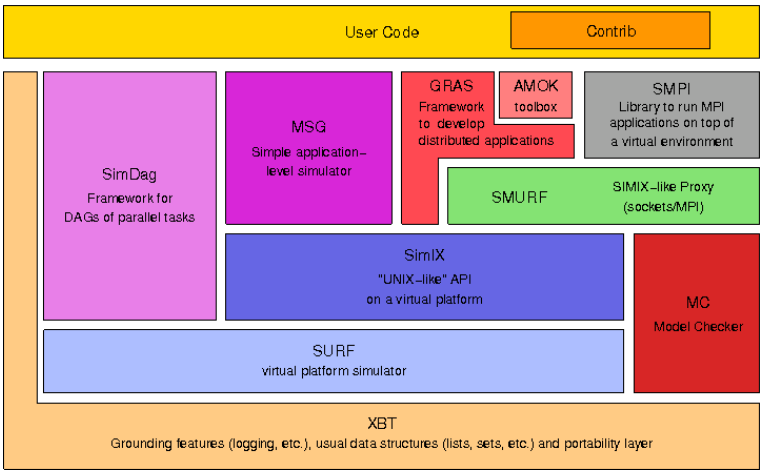


FIGURE 2.1 – Ensemble des environnements de SimGrid

On retrouve 4 interfaces utilisateurs à exploiter selon les résultats attendus et le type d'étude effectuée :

- SimDag représente la première version développée de SimGrid et est destiné à l'étude des applications structurées comme des graphes de tâches. Avec cette API, on peut créer des tâches, ajouter des dépendances entre les tâches, récupérer des informations sur la plateforme ou bien encore calculer le temps d'exécution d'un DAG (Directed Acyclic Graph) ;
- MSG est l'interface permettant l'étude des applications CSP (Concurrent Sequential Processes). Son utilisation première vise l'étude des algorithmes d'ordonnancement mais elle peut également servir pour les grilles de bureau ;
- GRAS (Grid Reality And Simulation) sert au développement d'applications distribuées au sein du simulateur mais qui sont ensuite déployées de façon transparente sur les plateformes réelles, sans modifier le code ;
- SMPI est le module destiné à la simulation d'applications MPI non modifiées en interceptant les primitives MPI.

2.1.4 Le projet *Simterpose*

Le projet ***Simterpose*** s'insère dans un axe de recherche visant à permettre l'étude sur simulateur d'applications complètes. L'objectif est d'émuler en utilisant un simulateur, dans notre cas, SimGrid. On souhaite ainsi fournir un émulateur simple et accessible à tous les utilisateurs grâce à sa facilité de déploiement que ce soit sur un simple ordinateur portable personnel ou bien sur un petit cluster. Un tel émulateur doit permettre :

- l'exécution d'un grand nombre d'instances d'une application sur un même système en vue d'un debuggage ;
- l'évaluation d'applications soumises à une large gamme de conditions telles que des caractéristiques d'un simple noeud ou d'un réseau complet différentes ;

– la collecte d'information concernant le comportement de l'application pendant son exécution.

L'aspect global du projet peut être représenté de la façon suivante : au lieu d'avoir des applications distribuées qui communiquent directement entre elles, on "intercale" notre simulateur entre les applications. Il joue ainsi le rôle de passerelle lors des échanges. Lorsqu'une application A souhaite communiquer avec une application B, sa demande est en premier lieu étudiée par SimGrid puis, selon l'analyse faite par le simulateur, ce dernier transmet l'information à l'application B en effectuant éventuellement de nouvelles actions en plus selon les cas.

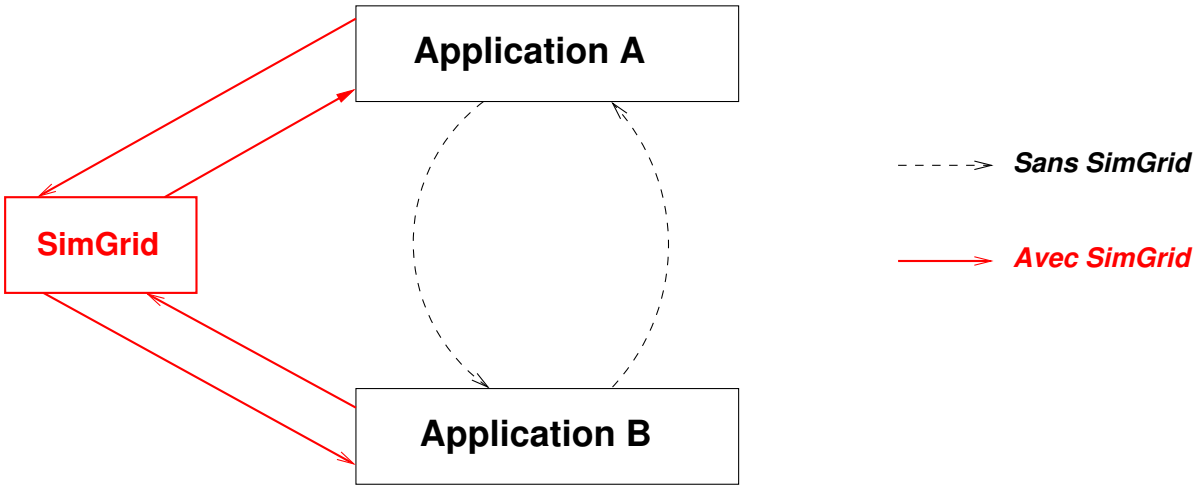


FIGURE 2.2 – Représentation schématique du projet Simterpose

Les objectifs sont d'étudier les moyens d'intercepter les actions des applications et d'implémenter une méthode d'interception en interagissant avec le simulateur SimGrid. Pour atteindre ce but, plusieurs étapes sont définies. La première consiste à déterminer combien de temps les actions d'une application mettent à se réaliser sur la plateforme logique. La seconde consiste à intercepter les actions de l'application ayant un impact sur son environnement.

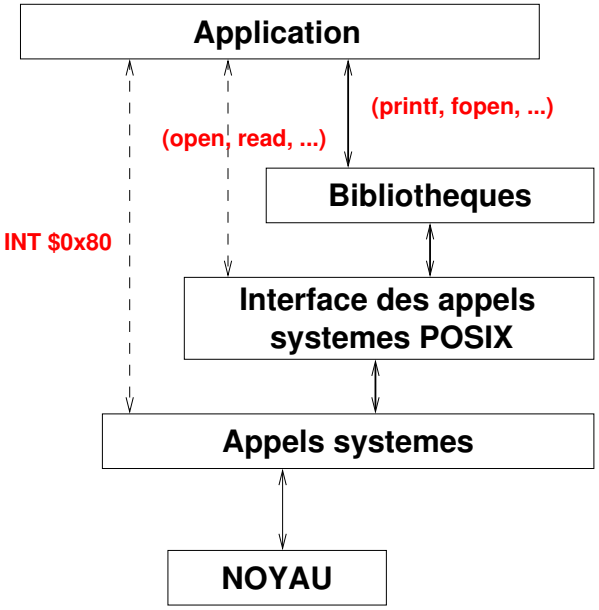
Mon travail, lors de ce stage, s'intéresse à la seconde étape, i.e, intercepter les actions de l'application (communication, calcul ou autre), tout cela pour chaque processus créé par l'application lors de son exécution. A chaque interception, je génère une trace contenant les informations nécessaires au simulateur pour le rejeu. Le but est ensuite de reporter cette trace dans le simulateur.

Pour cela, il faut commencer par étudier les moyens d'intercepter les actions de l'applications, puis implémenter une méthode d'interception en interagissant avec le simulateur SimGrid.

2.2 Interception système

2.2.1 Niveaux d'interceptions

Un système d'exploitation est composé d'un ensemble de fonctions (les appels systèmes), elles-mêmes faisant appel aux fonctions internes du noyau. Le tout est structuré de la façon suivante :



Au niveau de la couche *Bibliothèques* se situent les fonctions systèmes de la librairie standard C ainsi que les autres librairies pouvant être éventuellement utilisées par l'application. Ensuite, dans les systèmes UNIX, on retrouve les appels POSIX où les prototypes des fonctions sont normalisés ainsi que les structures de données du système auxquelles elles accèdent. Enfin, juste avant le noyau, il y a les fonctions directement appelées par le noyau via les appels systèmes et inaccessibles au programmeur. Par exemple, dans le cas d'une application C, lorsqu'on appelle la fonction `printf` pour afficher un texte sur la sortie standard, c'est la fonction `write` du noyau qui est appelée par le système.

Cette hiérarchie implique différents niveaux d'interception :

- au niveau de l'application
- au niveau des bibliothèques
- au niveau du noyau

Pour chacun de ces niveaux, il existe une ou plusieurs approches permettant d'intercepter les appels systèmes qui nous intéressent. Mon premier travail consiste à étudier chacune de ces approches afin de déterminer laquelle semble la plus adaptée face à nos attentes. Pour cette étude, j'ai commencé par coder plusieurs applications tests sous différents langages (C, Java, Ruby, Python, ...) qui me permettront d'évaluer la bonne interception des appels systèmes avec les différentes appoches.

2.2.2 Différentes approches d'interception

2.2.2.1 LD_PRELOAD

La première approche qui a été étudiée repose sur l'éditeur de liens dynamiques de Linux (ldd) qui permet d'interposer du code dans l'exécution d'un programme. Dans ce cas, l'interception se fera donc au niveau de la couche *Bibliothèques* du système.

La dernière phase de construction d'un programme est de réaliser l'édition de liens, ce qui consiste à assembler tous les morceaux du programme et de chercher ceux qui sont manquants. Les fonctions utilisées par l'application sont fournies sous forme de bibliothèques. Lorsqu'on utilise une bibliothèque statique, l'éditeur de liens cherche le code dont l'application a besoin et en effectue une copie dans le programme physique généré. Pour les bibliothèques partagées, c'est le contraire : l'éditeur de liens laisse du code qui lors du lancement du programme chargera automatiquement la bibliothèque. Linux effectue par défaut une édition de liens dynamique s'il peut trouver les bibliothèques de ce type sinon, il effectue une édition de liens statique.

Le principe de cette approche est d'utiliser la variable d'environnement LD_PRELOAD[9], qui contient la liste des bibliothèques à précharger avant les autres, pour charger notre propre bibliothèque. Pour intercepter un appel de fonction de bibliothèque, il faut tout d'abord implémenter la fonction correspondante en utilisant le même nom. Dans notre cas, nous souhaitons intercepter l'appel système mais pas forcément empêcher son exécution. Il nous faut donc appeler la fonction d'origine au sein de notre nouvelle version. Pour cela, on utilise les fonctions de la famille de `dlopen` qui permet de charger une bibliothèque dynamique dont le nom est fourni en paramètre. Ensuite, on compile notre fichier contenant toutes les fonctions wrappées pour générer notre propre bibliothèque. On obtient alors un fichier `.so`. Enfin, pour indiquer à l'application qu'il faut utiliser notre bibliothèque, il suffit d'ajouter cette dernière à la variable d'environnement LD_PRELOAD avant de l'exécuter. Ainsi, notre bibliothèque sera "prioritaire" face aux autres bibliothèques de base contenant les mêmes fonctions et ce sont nos propres fonctions qui seront utilisées.

Voici un exemple de création de bibliothèque avec une nouvelle fonction `dup` :

```
1  #define RTLD_NEXT ((void *) -1l)
2
3  char *error_msg;
4
5  /* On ecrit dans un fichier l'ensemble des fonctions wrappees */
6  void LogWrap(char *mess){
7    FILE *file_log=fopen("./LD_PRELOAD/file_log","a");
8    fprintf(file_log,"%s",mess);
9    fclose(file_log);
10 }
11
12 /* La fonction dlsym() prend un descripteur de bibliotheque et un nom de symbole et renvoie l'adresse ou ce symbole a ete
       charge */
13 void DLSym_fonctions(){
14
15   dup_orig=dlsym(RTLD_NEXT,"dup");
16   if((error_msg=dlerror())!=NULL)
17     fprintf(stderr,"Erreur dlsym dup !\n");
18
19 }
20
21 /* On reecrit la fonction dup en ajoutant des messages d'informations et en appelant l'originale */
22 int dup(int oldfd){
23   DLSym_fonctions();
24   LogWrap("appel dup()\n");
25   int ret=dup_orig(oldfd);
26   LogWrap("appel dup() terminÃŠ\n");
27   return ret;
28 }
```

Cette approche permet donc de modifier le comportement d'une application de façon indirecte, sans avoir à recompiler ou rééditer les liens à chaque fois. De plus, les applications distribuées étant généralement multi-process, il faut garantir une interception pour les éventuels nouveaux processus créés, ce que fait directement LD_PRELOAD sans ajouter d'option particulière. Cependant, cette approche s'avère vite incomplète dans notre cas puisqu'elle ne permet que de surcharger les fonctions des bibliothèques mais pas des appels systèmes. On n'obtient donc pas de réelle interception et nous n'avons aucune garantie que le système ne va pas contourner la nouvelle bibliothèque pour aller directement à la couche des appels systèmes.

2.2.2.2 Valgrind

Valgrind[10] est un outil de programmation libre pour déboguer, effectuer du profilage de code[3] et mettre en évidence des fuites mémoires. Il travaille donc directement au niveau de l'application. Il permet, lors des appels à des fonctions à intercepter, de dérouter vers une autre fournie par l'utilisateur. Ceci est généralement utilisé pour l'examen des arguments, pour compter le nombre d'appel à l'originale, et peut-être examiner le résultat.

Le wrapping de fonctions avec Valgrind consiste à créer un wrapper, i.e, une fonction de type identique à celle qui nous intéresse, mais avec un nom particulier qui permet de l'identifier par rapport à l'originale. Pour cela, il faut utiliser les macros fournies par Valgrind, en particulier les 3 suivantes :

- `I_WRAP_SONAME_FNNAME_ZU` : génère le vrai nom du wrapper grâce à un encodage spécifique à Valgrind ;
- `VALGRIND_GET_ORIG_FN` : une fois dans le wrapper, il faut accéder à la fonction originale dont l'adresse est stockée dans `OrigFn` ;
- `CALL_FN_W_WW` : permet d'appeler la fonction originale.

Voici un exemple typique de wrapper avec Valgrind dans lequel on s'intéresse à l'appel système `dup` :

```
1  #include "include/valgrind.h"
2  #include <stdio.h>
3
4
5  int I_WRAP_SONAME_FNNAME_ZZ(libcZdsoZd6, dup)(int n){
6
7    int ret;
8    OrigFn fn;
9    VALGRIND_GET_ORIG_FN(fn); // on recupere l'adresse de la fonction dup originale
10   printf("wrapper-pre : dup(%d)\n",n);
11   CALL_FN_W_W(ret,fn,n); // on execute la fonction dup avec ses parametres et on stocke le resultat dans ret
12   printf("wrapper-post : dup(%d) = %d \n",ret,n);
13   return ret;
14 }
15
16 int main(){
17
18   dup(0);
19
20 }
```

Pour la génération du vrai nom du wrapper avec la première macro, il faut préciser la bibliothèque contenant notre fonction originale et encoder selon un Z-encodage défini par Valgrind. Ceci est une opération lourde et complexe et nécessite en plus de connaître la biblio-

3. Le profilage de code consiste à analyser une application afin de connaître la liste des fonctions appelées et le temps passé dans chacune d'elles

thèque source de chaque appel système. De plus, il est important que le comportement observé sur la plateforme virtuelle soit le plus proche possible de celui obtenu sur plateforme réelle. Il faut donc faire attention à l'éventuel surcoût de chaque approche afin de garder une efficacité maximale. Dans ce cas, Valgrind s'est révélé inapproprié pour notre étude à cause d'un facteur de 7.5 lors de l'exécution de l'application avec ce dernier. Pour ces raisons, Valgrind est une approche qui a été abandonnée.

2.2.2.3 DynInst

DynInst[11] est une bibliothèque d'exécution multi-plateforme de code correctifs, utilisée dans le développement d'outils de mesure du rendement, des débogueurs, et des simulateurs. Avec son API, il est possible d'injecter directement du code dans une application en cours d'exécution. Le but de cette API est de fournir une machine interface indépendante afin de permettre la création d'outils et d'applications qui utilisent l'exécution de code de correction. Dans notre étude, cette approche travaille donc, comme Valgrind, directement au niveau de l'application.

Le fonctionnement de DynInst se décompose de la façon suivante : on commence par choisir le processus sur lequel on souhaite travailler. Puis, on crée une image de l'application sur laquelle on effectuera un éventuel travail (interceptions, injection de code, ...). Il "suffit" ensuite de tester la présence d'une fonction qui nous intéresse dans les bibliothèques chargées avec l'application.

Voici un exemple d'utilisation de DynInst pour intercepter l'appel `dup` :

```
1  #include <stdio.h>
2  #include <fcntl.h>
3  #include "BPatch.h"
4  #include "BPatch_process.h"
5  #include "BPatch_function.h"
6  #include "BPatch_Vector.h"
7  #include "BPatch_thread.h"
8
9  void usage(){
10   fprintf(stderr,"Usage : essai <filename> <args>\n");
11 }
12
13 BPatch bpatch;
14
15 int main(int argc, char *argv[]){
16
17   if(argc<2){
18     usage();
19     exit(1);
20   }
21   fprintf(stderr,"Attaching to process ... \n");
22   BPatch_process *appProc = bpatch.processCreate(argv[1],NULL);
23   if(!appProc){
24     fprintf(stderr, "echec create process\n");
25     return -1;
26   }
27
28   BPatch_image *appImage;
29   BPatch_Vector<BPatch_function *> dupFuncs;
30   fprintf(stderr, "Opening the programm image ...\n");
31   appImage = appProc->getImage();
32   appImage->findFunction("dup",dupFuncs);
33
34   if(dupFuncs.size() == 0){
35     fprintf(stderr, "ERROR : Unable to find function for dup()\n");
36     return 1;
37   }else{
38     printf("dup trouve !\n");
39   }
40 }
```

Cependant, l'API fournie par Dyninst semble très bas niveau, avec un niveau d'abstraction très élevé. On a donc un code vite complexe, long et difficile à mettre en place. Mon temps étant limité avec des objectifs à atteindre, nous avons décidé de mettre de côté cette approche sans pour autant l'écarter définitivement car elle offre un surcoût assez faible contrairement à d'autres approches telles que Valgrind travaillant au même niveau dans le système d'exploitation.

2.2.2.4 Ptrace

`ptrace()` est un appel système qui permet d'accéder en lecture/écriture à tout l'espace d'adressage d'un processus, i.e, aussi bien toutes les données que les structures de contrôle comme les registres du processeur. Toutes les fonctionnalités de `ptrace()` sont réalisées à partir d'un appel système unique, tout le travail de *dispatching* étant réalisé en espace noyau. Pour cela, l'appel système `ptrace()` fournit au processus parent un moyen de contrôler l'exécution d'un autre processus et d'éditer son image mémoire. `ptrace()` fonctionne grâce à des requêtes placées en paramètre de l'appel permettant diverses actions sur le processus tracé ou sur le processus qui trace.

L'utilisation de cet appel s'effectue de la façon suivante[12] : pour démarrer le suivi d'une application, on commence par créer un processus à tracer (le fils) et un processus traceur (le père) en appelant `fork()` . Le fils créé indique qu'il souhaite être tracé grâce à la requête `PTRACE_TRACEME` puis exécute l'application grâce à `exec`. Si le processus fils existe déjà, le père s'y attache directement en utilisant la requête `PTRACE_ATTACH`.Pendant l'exécution de l'application, le processus père peut laisser s'exécuter le processus fils et ne reprendre la main qu'à la réception d'un signal ou il peut être interrompu à chaque instruction en utilisant le mode pas à pas ou lors d'un appel système. Ainsi, le processus fils suivi s'arrêtera à chaque fois qu'un signal lui sera délivré, même si le signal est ignoré (à l'exception de `SIGKILL` qui a les effets habituels). Le processus père sera prévenu à son prochain `wait` et pourra inspecter et modifier le processus fils pendant son arrêt. Le processus parent peut également faire continuer l'exécution de son fils, éventuellement en ignorant le signal ayant déclenché l'arrêt, ou en envoyant un autre signal. Quand le processus père a fini le suivi, il peut terminer le fils avec la requête `PTRACE_KILL` ou le faire continuer normalement, i.e, non suivi, avec `PTRACE_DETACH`.

Avant de pouvoir tracer un processus, il est nécessaire de s'attacher à lui, les permissions requises sont les mêmes que celles nécessaires à l'envoi d'un signal à un processus, sauf en cas de modifications du noyau. Cela signifie qu'il n'est pas nécessaire d'être `root` et que tout utilisateur peut *ptracer* ses propres processus. Cette caractéristique fait partie des conditions à respecter dans le projet ***Simterpose***.

Voici un exemple d'utilisation de `ptrace()` pour intercepter l'appel `dup` :

```
1  #include <sys/ptrace.h>
2  #include <sys/types.h>
3  #include <sys/wait.h>
4  #include <unistd.h>
5  #include <linux/user.h>
6  #include <sys/syscall.h>
7
8  int main(){
9          pid_t child;
10   long orig_eax, eax;
11   long params[3];
12   int status;
```

```
13    int insyscall = 0;
14    struct user_regs_struct regs;
15    child = fork();
16    if(child == 0) {
17        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
18        execl("/bin/ls", "ls", NULL);
19    }else{
20      while(1) {
21        wait(&status);
22        if(WIFEXITED(status))
23            break;
24        orig_eax = ptrace(PTRACE_PEEKUSER, child, 4 * ORIG_EAX, NULL); // contient le numero de l'appel systeme intercepte
25        if(orig_eax == SYS_dup) {
26          if(insyscall == 0){ // on est en entree de l'appel
27            insyscall = 1;
28            ptrace(PTRACE_GETREGS, child, NULL, &regs); // on consulte les registres pour obtenir les parametres de l'appel
29            printf("Dup called with %ld\n",regs.ebx);
30          }else{ // on sort de l'appel
31            eax = ptrace(PTRACE_PEEKUSER,child, 4 * EAX,NULL); // contient la valeur retour de l'appel
32            printf("Dup returned with %ld\n", eax);
33            insyscall = 0;
34          }
35        }
36        ptrace(PTRACE_SYSCALL, child,NULL, NULL); // on fait repartir le fils
37      }
38    }
39    return 0;
40  }
```

Du point de vue du niveau d'interception, `ptrace()` est donc lié au noyau. Ceci implique un éventuel problème de portabilité. En effet, il n'est pas assuré de pouvoir lancer une application écrite pour une version (majeure) différente du noyau : la mémoire virtuelle vue par un processus peut avoir été complètement remaniée, les alignements, la taille d'un mot peut changer ou bien encore, la sémantique de certains signaux peut avoir été modifiée, etc ... . De même, entre différentes architectures, il est difficile d'assurer une quelconque compatibilité : les registres changent, leurs tailles, la capacité du processeur à accéder à des adresses mémoires non alignées, etc ... .

2.2.2.5 Uprobes

Avec un surcoût non négligeable et une API non portable et difficile à utiliser, notamment pour le multithreading, `ptrace()` est entrain d'être concurrencé par une nouvelle interface : Uprobes, dont l'objectif est de fournir une alternative à `ptrace()` en corrigeant ses défauts.

Uprobes permet de pénétrer dynamiquement dans une routine d'une application et de collecter des informations de débogage et les performances sans interruption. Il est possible d'intercepter à n'importe quelle adresse de code, en spécifiant un gestionnaire de routine du noyau à invoquer lorsque le point d'arrêt est atteint. Un point d'arrêt peut être inséré sur toute instruction dans l'espace virtuel de l'application. La fonction d'enregistrement `register_uprobe()` indique quel processus est à sonder, où le point d'arrêt doit être inséré et quel gestionnaire doit être appelé lorsque le point d'arrêt est touché.

Uprobes fonctionne de la façon suivante : quand un point d'arrêt est enregistré, Uprobes fait une copie de l'instruction sondée, arrête l'application, remplace le(s) premier(s) octet(s) de l'instruction sondée avec la routine à invoquer, puis permet à l'application de continuer.

Uprobes supporte l'étude des applications multithreadées et n'impose aucune limite sur le nombre de threads dans une application sondée.

Bien que prometteuse, cette approche n'a pas été approfondie car Uprobes est encore en développement avec peu d'exemple de mise en application et de documentation.

2.2.3 Comparaison des différentes approches

	LD_PRELOAD	Valgrind	DynInst	Ptrace	Uprobes
Niveau d'interception	Bibliothèques	Application	Application	Noyau	Noyau
Coût	Faible	Important	Assez faible	Moyen	Faible(?)
Facilité d'utilisation	Simple	Complexe	Complexe	Assez complexe	?

FIGURE 2.3 – Tableau comparatif des différentes approches d'interception

Avec un coût faible et une utilisation simple, LD_PRELOAD semblait une bonne approche à approfondir. Cependant, comme expliqué précédemment, avec une interception effectuée au niveau de la couche *Bibliothèques*, il existe un risque non négligeable que l'application contourne notre nouvelle bibliothèque chargée en allant directement au niveau de la couche *Appels systèmes*. C'est pourquoi nous avons décidé de travailler avec `ptrace()`. Uprobes est défini comme alternative aux problèmes rencontrés avec `ptrace()` mais, étant en cours de développement, il est envisagé pour la suite une étude de cette approche après exploration complète de `ptrace()`.

2.3 Capture de traces avec `ptrace`

2.3.1 Extraction des appels système

Mon premier objectif durant mon stage a été de réaliser une sorte de mini `strace`[4] qui repose lui-même sur la fonction `ptrace()`. Il faut donc pouvoir générer en sortie la liste de tous les appels de fonctions effectués par le programme avec les arguments et la valeur de retour, si il y a, ainsi que le numéro du processus tracé.

La première étape dans la capture de traces a été de cibler les appels système qui nous intéressent avec, en particulier, les appels orientés réseau. Parmi ces appels, on retrouve `write/read`, `open/close` ainsi que tous les appels liés aux sockets tels que `socket, bind, connect, accept, send, recv`, ... Pour intercepter ces appels avec `ptrace()`, il faut connaître les numéros affectés à ces fonctions dans le système. Pour cela, on se réfère aux fichiers */usr/include/asm/unistd_32.h*, dans le cas d'un processeur 32 bits, ou */usr/include/asm/unistd_64.h*, dans le cas d'un processeur 64 bits. Ainsi, par exemple, la fonction `write` a le numéro 4 en 32 bits mais le numéro 1 en 64 bits.

La plupart des fonctions prennent au moins un argument. Ces arguments ainsi que la valeur de retour de l'appel système intercepté sont stockés dans les registres généraux utilisés par le processeur pendant l'exécution du programme. Dans les processeurs 32 bits, on retrouve 8

4. `strace` est un outil de déboguage sous Linux pour surveiller les appels système utilisés par un programme

registres généraux tandis que pour les processeurs 64 bits, il y en a 16. Parmi ces registres, ce sont les registres de données qui sont utilisés pour stocker les arguments de fonctions et le résultat. Pour manipuler ces registres, `ptrace()` utilise la requête `PTRACE_GETREGS` qui récuprère l'ensemble des registres ainsi que leur contenu dans une structure de type `user_regs_struct`. Dans le cas des fonctions autres que celles liées aux sockets, pour récupérer les arguments, il suffit de lire immédiatement le contenu des registres. Cependant, l'opération est plus compliquée dans le cas des fonctions opérant sur les sockets avec les processeurs 32 bits[13]. En effet, dans ce cas, les fonctions `socket, bind, listen, accept, ...` sont gérées par un seul appel système nommé `socketcall`. Cet appel prend 2 arguments : le premier est le numéro de la sous-fonction à exécuter (1 pour `socket`, 2 pour `bind`, ...) et le second argument est l'adresse du fragment de la mémoire contenant les arguments de cette sous-fonction. Au niveau des registres, ceci va donc se traduire par la récupération dans le second registre de données (le premier étant réservé au résultat de l'appel système) de l'adresse du fragment mémoire contenant tous les arguments, puis il faut aller de case mémoire en case mémoire pour lire chaque argument. Pour cela, `ptrace()` propose la requête `PTRACE_PEEKDATA` qui lit un mot à une adresse donnée dans l'espace mémoire du fils. La difficulté avec cette opération est le déplacement dans la mémoire où chaque case est de taille `sizeof(long)`. Ceci nécessite de connaître chaque type des arguments des fonctions étudiées. Ce travail a été assez compliqué pour moi à cause des lectures mémoires qui nécessitent une certaine connaissance des architectures.

Par souci de clarté, il fallait également afficher le nom symbolique associé aux constantes littérales [5] régulièrement utilisées dans les paramètres de fonctions. Lors de la lecture du contenu de la mémoire, j'obtenais pour ce type de paramètre un entier qui peut correspondre à une seule constante ou bien être le résultat d'un `AND` entre toutes les constantes utilisées. J'ai donc du chercher directement dans les fichiers sources du système pour associer dans chaque fonction la valeur littérale à la valeur numérique obtenue.

De plus, lors d'un appel à une fonction, pour la majorité, il y a en fait 2 appels système qui sont effectués : un pour l'entrée dans l'appel et un pour la sortie. On ne va donc récupérer les arguments et surtout le résultat de la fonction une fois que celle-ci a fini de s'exécuter donc en sortie de l'appel. Un simple tableau à 0 ou 1 permet de gérer ce système d'entrée et sortie d'appel.

Voici un exemple de résultat obtenu lors d'une capture de trace sur un simple Client/-Serveur TCP avec `strace` :

```
1  open("/etc/ld.so.cache", O_RDONLY)      = 3
2  fstat64(3, {st_mode=S_IFREG|0644, st_size=74261, ...}) = 0
3  close(3)                                = 0
4  open("/lib/libncurses.so.5", O_RDONLY)  = 3
5  read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\240\223\0\0004\0\0\0"..., 512) = 512
6  ..
7  ..
8  clone(Process 29513 attached
9  child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0xb77ba728) = 29513
10 [pid 29512] rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
11 [pid 29513] close(255 <unfinished ...>
12 [pid 29512] rt_sigprocmask(SIG_BLOCK, [CHLD], <unfinished ...>
13 [pid 29513] <... close resumed> )       = 0
14 [pid 29513] socket(PF_INET, SOCK_STREAM, IPPROTO_IP <unfinished ...>
15 [pid 29514] <... rt_sigprocmask resumed> NULL, 8) = 0
16 [pid 29513] <... socket resumed> )      = 3
17 [pid 29514] rt_sigaction(SIGTSTP, {SIG_DFL, [], 0}, <unfinished ...>
18 [pid 29513] setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4 <unfinished ...>
```

5. Les constantes littérales représentent des valeurs numériques ou caractères entrées dans le code source du programme et réservée et initialisée lors de la phase de compilation de code source.

```
[pid 29514] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[pid 29513] <... setsockopt resumed> )  = 0
[pid 29514] rt_sigaction(SIGTTIN, {SIG_DFL, [], 0}, <unfinished ...>
[pid 29513] getsockopt(3, SOL_SOCKET, SO_REUSEADDR <unfinished ...>
[pid 29514] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[pid 29513] <... getsockopt resumed> , "\1", [1]) = 0
[pid 29513] bind(3, {sa_family=AF_INET, sin_port=htons(2226), sin_addr=inet_addr("0.0.0.0")}, 16 <unfinished ...>
[pid 29514] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[pid 29513] <... bind resumed> )        = 0
[pid 29512] rt_sigprocmask(SIG_SETMASK, [], <unfinished ...>
[pid 29514] rt_sigaction(SIGINT, {SIG_DFL, [], 0}, <unfinished ...>
[pid 29513] listen(3, 128 <unfinished ...>
[pid 29514] <... rt_sigaction resumed> {SIG_DFL, [], 0}, 8) = 0
[pid 29513] <... listen resumed> )      = 0
```

.

Voici un exemple de résultat obtenu lors d'une capture de trace sur un simple Client/Serveur TCP avec notre mini strace :

```
[24401] open("...", O_RDONLY) = 4
[24401] Unknown syscall 197 = 0
[24401] close(4) = 0
[24401] open("...", O_RDONLY) = 4
[24401] read(4, ..., 512) = 512
..
..
[24401] clone() = -38
nouveau pid avec (v)fork 24403
[24403] clone() = 0
[24401] clone() = 24403
[24402] socket( PF_INET, SOCK_STREAM, IPPROTO_IP) = 4
[24402] setsockopt(4, SOL_SOCKET, SO_REUSEADDR, 4 ) = 0
[24402] getsockopt(4, SOL_SOCKET, SO_REUSEADDR, 1 ) = 0
[24402] bind( 4, {sa_family=AF_INET, sin_port=htons(2226), sin_addr=inet_addr("0.0.0.0")}, 16 ) = 0
[24402] listen( 4, 128  ) = 0
..
..
[24403] close(1) = 0
[24403] close(2) = 0
[24403] exit_group(0) called
..
[24419] connect( 4, {sa_family=AF_INET, sin_port=htons(2226), sin_addr=inet_addr("127.0.0.1")}, 16 ) = 0
[24402] accept( 4, {sa_family=AF_INET, sin_port=htons(56842), sin_addr=inet_addr("127.0.0.1")}, 16 ) = 5
[24419] Unknown syscall 197 =? -38
[24419] send( 4, 512 ,  ) = 512
[24402] recv( 5, 512 ,  ) = 512

[24419] exit_group(0) called
```

.

2.3.2 Calcul du temps d'exécution

En plus de connaître les appels effectués par l'application pendant son exécution, il est nécessaire de connaître les périodes de calcul pour les transmettre à SimGrid afin qu'il les prenne en compte dans son analyse.

Pour cela, il fallait, à chaque interruption par `ptrace()`, déterminer le CPU time[6] et le Wall time[7]. En collaboration avec Tomasz BUCHERT, également en stage dans l'équipe AlGorille, j'ai intégré dans mon application existante une fonction permettant de calculer ces valeurs. Les résultats obtenus réprésentaient les CPU time et Wall time totaux depuis le début de l'exécution, nous avons donc ajouter un calcul permettant d'obtenir les différentes valeurs entre chaque entrée/sortie d'un appel et entre chaque sortie d'un appel et l'entrée dans un nouveau. La difficulté dans ce calcul était de différencier les différents CPU time de chaque

6. Le CPU time correspond au temps d'utilisation du processeur par un processus pendant l'exécution d'un programme.

7. Le wall time correspond au temps réellement écoulé entre le début et la fin de l'exécution d'un programme, en incluant les éventuels temps d'attente pour l'utilisation de certaine ressource.

processus dans le cas de multithreading. Ainsi, pour chaque processus tracé, on a une structure contenant les derniers CPU time et Wall time calculés, les valeurs totales n'étant pas stockées car elles sont recalculées à chaque interruption.

De plus, il existe des appels système dit "bloquants". En effet, lorsqu'un processus est en train de s'exécuter, il peut exécuter des instructions du processeur (par exemple pour effectuer des calculs), ou faire des appels système. Parfois, le noyau peut satisfaire l'appel système immédiatement : par exemple, un appel à `gettimeofday` calcule l'heure actuelle et retourne immédiatement au code utilisateur. Cependant, certains appels système ne retournent pas immédiatement et sont en attente d'un événement. Ils bloquent donc la suite de l'exécution. Parmi ces appels système, on retrouve en particulier `recv`, `recvfrom` et `recvmsg`, régulièrement utilisés dans nos applications tests. Pour différencier ces périodes d'attente des vraies périodes de calcul, une analyse des CPU time et Wall time ajoutés à la trace doit être effectuée.

2.3.3 Identification des processus communiquants

Pour faire rejouer la trace à SimGrid, en plus de connaître les appels système effectués, nous avons besoin de connaître les processus qui communiquent entre eux via les sockets lors des appels du type `send` et `recv`.

Pour chaque socket créée, le système attribue un file descriptor [8] unique. Cependant, il y a unicité par processus. Si bien que le file descriptor 4, par exemple, peut être attribué autant de fois qu'il y a de processus en cours d'exécution. Par conséquent, cette information à elle seule n'est pas suffisante pour identifier les processus communiquants entre eux. Nous avons donc découper le problème en 2 étapes : la lecture des informations sur les sockets, i.e, les adresses IP et ports locaux et distants de chaque file descriptor d'une socket, puis une mise en relation entre tous les file descriptor qui permettra de relier 2 sockets entre elles grâce à leurs informations communes.

2.3.3.1 Lecture des informations sur les sockets

La lecture des informations sur les sockets consiste à récupérer pour chaque socket l'adresse IP par laquelle elle communique (adresse locale) et sur quel port ainsi que l'adresse IP avec qui elle communique (adresse distante) sur un certain port.

Lorsqu'on crée une socket via la fonction `socket`, on commence par indiquer le domaine de comunication (INET, UNIX, NETLINK, ...) et le protocole (TCP, UDP, ICMP, IP, ...) qui doit être utilisé pour communiquer. Ensuite, dans le cas d'un serveur, on fait appel à la fonction `bind` qui fournit à une socket donnée une adresse IP locale et un port : on parle "d'assignation d'un nom à une socket". À ce moment, la socket se place donc "en attente" grâce à la fonction `listen` qui indique son désir d'accepter des connexions entrantes. Du côté du client, une fois la socket créée, on fait appel à la fonction `connect` qui permet de débuter une connexion sur une socket donnée. Par cet appel, le client indique donc au serveur qu'il souhaite entrer en communication avec lui. Le client fournit une adresse IP et un port de connexion dans les paramètres de la fonction. Il s'agit donc pour le client d'une adresse locale comme pour le serveur avec le `bind`. Enfin, pour "sceller" cette liaison et commencer à échanger, le serveur fait un `accept` qui, comme son nom l'indique, permet d'accepter une connexion entrante. Suite à cet appel, un nouveau file descriptor est créé avec pour adresse et port locaux ceux du ser-

8. Un file descriptor (descripteur de fichier) est une clé abstraite (un entier) pour accéder à un fichier. Généralement, il s'agit d'un index d'une entrée dans le noyau-résident.

veur et pour adresse et port distants ceux du client connecté. Et du côté du client, suite à la demande de connexion qui a été acceptée, son file descriptor se voit attribuer pour adresse et port distants ceux du serveur avec qui il communique. On obtient donc 2 couples (IP,port) inversement identiques.

Pour obtenir ces informations, nous avons besoin de connaître le numéro de la socket qui est unique sur l'ensemble du système contrairement au file descriptor. Pour cela, on utilise la commande `ls -l /proc/#pid/fd/#fd` dans laquelle on indique le pid du processus (#pid) auquel appartient la socket et le file descriptor (#fd) qui lui a été associé lors de sa création. Une fois ce numéro de socket obtenu, selon le protocole utilisé par la socket, nous devons parcourir un fichier : */proc/net/protocol* où *protocol* peut être tcp, udp, raw, netlink, etc ... , qui contient la table des connexions actuellement ouvertes pour le protocole donné. Pour chaque connexion, nous avons, parmi d'autres informations, les deux couples (IP,port) local et (IP,port) distant. Il suffit alors de trouver la connexion correspondant à notre socket grâce à son numéro précédemment obtenu et d'extraire les informations qui nous intéressent. Cette opération se fait par simple parcours et découpage du fichier.

Pour ne pas avoir à répéter ce travail à chaque appel système, j'ai créé pour chaque socket une structure contenant le pid, le file descriptor, le domaine et le protocol, ainsi que les 2 couples (IP,port). Ces informations sont enregistrées au fur et à mesure des appels. En effet, lors de la création de la socket, on obtient le file descriptor, le domaine et le protocol. Puis, après un `bind` on récupère le couple (IP,port) local. Coté client, après un `connect`, on récupère les 2 couples (IP,port). Enfin, suite à un `accept`, on enregistre le nouveau file descriptor avec ses adresses et ports locaux et distants.

2.3.3.2 Identification du processus destinataire

Pour identifier le processus destinataire lors de chaque communication (`send, recv, ...`), il nous suffit de parcourir le tableau contenant les structures de chaque socket et de chercher quelle socket possède les 2 couples (IP,port) inverses. On en extrait alors le pid.

Voici un exemple de trace obtenue lors d'une capture sur une application distribuée de type Client/Serveur simple avec le client qui envoie un message au serveur qui répond également avec un message.

```
1   Timestamp  pidX  wall_time  cpu_time  diff_wall  diff_cpu  type  syscall  local_addr:port
        remote_addr:port  pidY  return  param
2   23:15:18:938060  6976  19234  12000  0  12000  (v)fork
                                                  6977
3   23:15:18:944354  6976  25537  16000  6303  4000  (v)fork
                                                  6978
4   23:15:21:956346  6978  3012309  12000  0  12000  exit_group
                                                  0
5   23:15:21:957648  6976  3038838  16000  3013301  0  (v)fork
                                                  6989
6   23:15:21:969823  6977  3031988  0  0  0  in  recv  127.0.0.1: 2226
        127.0.0.1:34024  6989
7   23:15:21:970159  6989  12697  0  0  0  in  send  127.0.0.1:34024
        127.0.0.1: 2226  6977
8   23:15:21:970356  6989  12895  0  198  0  out  send  127.0.0.1:34024
        127.0.0.1: 2226  6977  512  (4, "...", 512)
9   23:15:21:970471  6977  3032640  0  652  0  out  recv  127.0.0.1: 2226
        127.0.0.1:34024  6989  512  (5, "...", 512)
10  23:15:21:970594  6989  13133  0  238  0  in  recv  127.0.0.1:34024
        127.0.0.1: 2226  6977
11  23:15:21:970791  6977  3032963  0  323  0  in  send  127.0.0.1: 2226
        127.0.0.1:34024  6989
12  23:15:21:970966  6977  3033136  0  173  0  out  send  127.0.0.1: 2226
        127.0.0.1:34024  6989  512  (5, "...", 512)
13  23:15:21:971104  6989  13643  0  510  0  out  recv  127.0.0.1:34024
        127.0.0.1: 2226  6977  512  (4, "...", 512)
```

```
14  23:15:21:971476   6977   3033648      0      512      0      exit_group
                                                                  0
15  23:15:21:971954   6989     14493      0      850      0      exit_group
                                                                  0
16  23:15:21:972737   6976   3053932  16000    15094      0      exit_group
                                                                  0
```

.

2.4 Bilan et perspectives

L'objectif principal de ce stage, qui était l'interception d'appels système et la génération de trace, a donc été parfaitement atteint. Nous obtenons donc la liste des actions qui ont un impact sur l'environnement de l'application avec les temps d'exécution et les quantités de données échangées. Nous avons commencé en testant sur les applications que j'avais moi-même écrites. Au fur et à mesure de ces tests, nous avons pu enrichir la trace et élargir le champs de cas traités. Parmi les problème soulevés lors de ces tests, il y avait la gestion des applications multithreading, qui créent donc des nouveaux processus en plus de celui de l'application elle-même. Il faut donc que notre traceur soit capable d'intercepter et différencier toutes les actions de chaque processus. Face à une documentation très succinte sur ce sujet avec `ptrace()`, cette étape du projet a été difficile à mettre correctement en place et a continué à nous occuper durant tout le développement, nous retardant parfois dans la mise en place de certaines fonctionnalités. Puis, après validation sur des cas simples, nous avons porter nos tests sur de "vraies" applications telle que BitTorrent. Le but était de tracer le téléchargement d'un fichier de 100Mo. Ceci nous a permis de mettre en avant des failles dans mon traceur, notamment avec des cas qui n'avaient jamais pu être testés avec mes applications simples, tels que l'utilisation de socket NETLINK, l'utilisation des appels système `sendmsg` et `recvmsg`. De même pour la lecture des informations sur les sockets, contrairement à un Client/Serveur simple, dans BitTorrent nous avons plusieurs clients qui deviennent eux-mêmes serveurs, ceci n'avait donc pas été géré dans un premier temps. Grâce à cette évolution dans la complexité des applications testées, nous avons aujourd'hui un traceur adapté à une large gamme de cas.

Cependant le second objectif durant ce stage était de transmettre la trace obtenue à SimGrid pour qu'il rejoue les actions interceptées. Ce dernier n'a pas pu être atteint. En effet, dans le développement actuel de SimGrid pour le rejeu de traces, ce dernier accepte des traces de la forme suivante :

```
1  # sample action file
2  tutu send toto 1e10
3  toto recv tutu
4  tutu sleep 12
5  toto compute 12
```

.

On retrouve donc les 2 processus communiquants, l'action effectuée (l'appel système) et la taille des données échangées ou calculées. Il s'agit donc d'une forme plutôt simplifiée en comparaison avec ma trace générée. Il faut donc décider si on modifie SimGrid pour qu'il rejoue ma trace dans son état actuel ou bien si on modifie ma trace pour l'adapter au format actuellement accepté par SimGrid. Le choix se porterait aujourd'hui sur le développement de SimGrid pour qu'il rejoue ma trace mais nous n'avons malheureusement pas eu le temps de travailler sur cette partie.

De plus, avec notre traceur actuel, l'émulation avec SimGrid se fera *offline*, c'est-à dire, que l'on commence par exécuter l'application que l'on trace en même temps. Ensuite, une fois l'exécution terminée et notre trace obtenue, on la transmet à SimGrid qui l'analysera et rejouera toutes les actions interceptées en ajoutant éventuellement des modifications. L'idée serait donc d'envisager une émulation *online* où on reporterait immédiatement les actions interceptées dans le simulateur, puis, on retarderait l'application du temps calculé par le simulateur.

Durant mon stage, j'ai également assisté à une réunion sur un projet d'association entre SimGrid et Glite. Glite[14] est un intergiciel de production nouvelle génération pour le grid computing. Il fournit un système pour construire des applications sur grille puisant dans la puissance de l'informatique distribuée et des ressources de stockage sur Internet. L'objectif de cette mise en relation est de définir un environnement contrôlé reproduisant fidèlement le comportement d'une grille de production pour analyser le fonctionnement du système sous charge et évaluer différentes stratégies d'optimisation. Pour mettre en place cela, les chercheurs et développeurs en charge du projet souhaitent utiliser le projet ***Simterpose*** pour la partie *Gestion de l'infrastructure matérielle* où le but est de modéliser et simuler des ressources de stockage, calcul et de communication. Il s'agit donc ici d'une nouvelle approche où émulation et simulation sont mélangées, ce qui va impliquer un travail "risqué" mais prometteur dans le partenariat Recherche/Production.

Conclusion

Le projet ***Simterpose*** a été une excellente occasion pour moi de mettre en pratique les enseignements vus à l'ESIAL, en particulier les cours de Réseaux et Système. J'ai en effet pu approfondir mes connaissances sur les applications distribuées mais aussi sur la programmation orientée Réseaux. J'ai également pu découvrir une certaine organisation dans ce type de projet avec un système de reporting hebdomadaire permettant de dresser le bilan du travail de la semaine et de discuter en avance des tâches de la semaine suivante. Ceci m'a permis à la fin de mon stage de revoir clairement toute l'évolution du projet durant ces 10 semaines.

J'ai aussi pu expérimenter le logiciel de versions décentralisées *Git* qui, une fois maîtrisé un minimum, se révèle être un outil indispensable dans le développement de projets de ce type. Le but de l'utilisation de ce type de logiciel est d'obtenir un arbre représentant tout le déroulement de développement avec la possibilité de revenir à tout moment à une étape antérieure. Cependant, pour exploiter correctement cette caractéristique, il faut une certaine rigueur dans la mise à jour des fichiers. L'idée est de mettre à jour un fichier dans l'arborescence à chaque modification de fonctions, structures ou même de variables. J'ai eu beaucoup de mal à intégrer correctement cette technique et avais tendance à attendre la fin de la journée pour transmettre en une fois toutes les modifications effectuées. Mais après plusieurs tentatives de Lucas, mon second encadrant au sein de l'équipe, pour me faire comprendre l'intérêt certain d'une telle rigueur, je pense enfin avoir compris comment utiliser correctement cet outil et compte bien m'en resservir pour mes futurs projets.

Durant ces 10 semaines, j'ai également appris à voir à plus long terme que pour les projets qu'on peut faire à l'ESIAL. Quand je suis arrivée, un travail avait déjà été commencé sur le sujet et il sera encore continué après mon départ. À chaque réunion, on discutait du prochain objectif à atteindre à court terme mais il était toujours évoqué l'objectif final même si on supposait que ce ne serait pas fait pendant mon stage. J'ai aussi du me résoudre à accepter " l'abandon" de plusieurs jours de travail suite à une voie sans issue. Bien que tout travail et toute recherche ne sont jamais inutiles, cela reste assez déstabilisant de se dire qu'on n'utilisera finalement pas cela pour la suite. Il s'agit là d'une vraie confrontation avec le monde de la recherche.

Enfin, ce stage m'a permis de comparer le monde de l'entreprise, que j'avais découvert durant mon DUT Informatique, et le monde de la recherche. Ce sont des projets de plusieurs années qui y sont développés avec de nombreuses collaborations. Il m'est apparu une réelle envie de continuer du côté de la recherche, et même de retravailler plus tard sur le projet si l'opportunité m'était offerte.

Bibliographie

[1] «LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications)». http://www.loria.fr/.

[2] «AlGorille : Algorithmes pour la grille». http://www.loria.fr/equipes/algorille.

[3] «BOINC, calculer pour la science». http://boinc.berkeley.edu/.

[4] «Folding@Home, Distributing computing». http://folding.stanford.edu/.

[5] «SETI@Home : Search for Extraterrestrial Intelligence». http://setiathome.berkeley.edu/.

[6] «Wikipédia : encyclopédie libre». http://fr.wikipedia.org/.

[7] «SimGrid». http://simgrid.gforge.inria.fr/.

[8] H. Casanova, A. Legrand et M. Quinson. «SimGrid : a Generic Framework for Large-Scale Distributed Experiments». Dans *10th IEEE International Conference on Computer Modeling and Simulation* (2008).

[9] «LD_PRELOAD : Intercepting Dynamic Library Function Calls». http://www.uberhip.com/people/godber/interception/html/slide_4.html.

[10] «Valgrind». http://valgrind.org/.

[11] «DynInst : An Application Program Interface (API) for Runtime Code Generation». http://www.dyninst.org/.

[12] P. Padala. «Playing with ptrace». *LinuxJournal* (2002).

[13] «Shellcode sous Unix - Optimisation des shellcodes sous Linux». http://www.linux-pour-lesnuls.com/.

[14] «gLite : Lightweight Middleware for Grid Computing». http://glite.web.cern.ch/glite/.