Université Pierre et MArie Curie

Master informatique

Systèmes et Applications Répartis

Loria

Equipe VERIDIS

Real-time online emulation of real applications on SimGrid with Simterpose

Encadrant
Martin Quinson

Etudiant :
Louisa Bessad

Rapporteur :
Sébastien Monnet

1er septembre 2015

Table des matières

Table des figures

Liste des tableaux

Ce stage se déroule au Loria, Laboratoire Lorrain de Recherche en Informatique et ses Applications, unité mixte de recherche commune à plusieurs établissements : le CNRS, l'INRIA, et l'Université de Lorraine. Le LORIA a pour mission la recherche fondamentale et appliquée en sciences informatiques depuis sa création en 1997.

L'encadrement est assuré par Martin Quinson au sein de l'équipe VERIDIS, dont les sujets de recherches sont la conception de méthodes pour les systèmes distribués et d'outils pour la vérification et la validation de systèmes.

TODO

1 Introduction

Dans le cadre de ce stage, nous allons nous intéresser aux applications distribuées. Il s'agit d'applications dont une partie ou la totalité des ressources n'est pas localisée sur la machine où l'application s'exécute, mais sur plusieurs machines distinctes. Ces dernières communiquent entre elles via le réseau pour s'échanger les données nécessaires à l'exécution de l'application. Les applications distribuées ont de nombreux avantages : elles permettent notamment d'augmenter la disponibilités des données en se les échangeant, comme les applications Torrent (BitTorrent, Torrent...). Grâce au projet BOINC [1] par exemple, on peut partager la puissance de calcul inutilisée de sa machine. On peut également penser aux applications de stockage de données LAFS [2] et CEPH [3]. La première apporte un stockage robuste qui préserve les données même si un site est physiquement détruit. La seconde souhaite fournir performance, fiabilité et scalabilité. Depuis une dizaine d'années, la popularité de ces applications distribuées ne cesse de croître. Elles deviennent de plus en plus complexes avec des contraintes et des exigences de plus en plus fortes, en particulier au niveau des performances et de l'hétérogénéité des plateformes et des ressources utilisées. Il est donc de plus en plus difficile de créer de telles applications (absence d'horloge commune et mémoire centrale, deadlock, race condition, famine) mais aussi de les tester. En effet, malgré l'évolution des applications distribuées, les protocoles d'évaluation de leurs performances n'ont que peu évolués.

Actuellement, il existe trois façons principales de tester le comportement d'applications distribuées [26] : l'exécution sur plate-forme réelle, la simulation et l'émulation.

La première solution consiste à exécuter réellement l'application sur un parc de machines et d'étudier son comportement en conditions réelles. Cela permet de la tester sur un grand nombre d'environnements. L'outil créé et développé en partie en France pour cela est **Grid'5000** [4][18], un autre outil développé à l'échelle mondiale est **PlanetLab** [5]. Néanmoins, afin de mettre en œuvre ces solutions complexes, il faut disposer des infrastructures nécessaires pour effectuer les tests. Il faut également écrire une application complète capable de gérer toutes ces ressources disponibles. Enfin, du fait du partage des différentes plateformes entre plusieurs utilisateurs, les expériences sont difficilement reproductibles.

La seconde solution consiste à faire de la simulation : on modélise ce que l'on souhaite étudier (application et/ou environnement) via un programme appelé simulateur. Dans ce cas, pour pouvoir tester des applications distribuées, on doit d'abord abstraire l'application ainsi que l'environnement d'exécution. Pour cela, on identifie les propriétés de l'application et de son environnement puis on les transforme à l'aide de modèles mathématiques. Ainsi, on va exécuter dans le simulateur le modèle de l'application et non l'application réelle, dans un environnement également modélisé. Cette solution est donc facilement reproductible, plus simple à mette en œuvre, et permet de prédire l'évolution du système étudié grâce à l'utilisation de modèles mathématiques. De nos jours, les simulateurs tels que **SimGrid**[23, 36] peuvent simuler des applications distribuées mettant à contribution des milliers de nœuds. Néanmoins, avec la simulation on ne

1. https://boinc.berkeley.edu/
2. https://tahoe-lafs.org/trac/tahoe-lafs
3. http://ceph.com/
4. Infrastructure de 8000 cœurs répartis dans la France entière crée en 2005. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home
5. Crée en 2002, cette infrastructure de test compte aujourd'hui 1340 cœurs. http://www.planet-lab.org

peut valider qu'un modèle et non l'application elle-même.

La troisième solution consiste à faire de l'émulation : on exécute réellement l'application mais dans un environnement virtualisé grâce à un logiciel, l'émulateur. Ce dernier joue le rôle d'intercepteur pour virtualiser l'environnement d'exécution. Cette solution représente un intermédiaire entre la simulation et l'exécution sur plateforme réelle visant à résoudre les limitations de ces deux solutions. En effet, les actions de l'application sont réellement exécutées sur la machine hôte (la machine réelle sur laquelle s'exécute l'émulation) et grâce à la virtualisation, l'application pense être dans un environnement différent de la machine réelle. De plus, cela évite d'avoir deux versions de l'application en terme de code : une pour la simulation et une pour la production. L'émulation peut-être faite *off-line* (on sauvegarde les actions de l'application sur disque et on les rejoue plus tard dans un simulateur) ou *on-line* (on bloque l'application le temps que le temps de réponse de la plateforme virtualisée soit calculé).

Au cours de ce stage, nous allons nous intéresser plus particulièrement à l'exécution d'applications distribuées arbitraires au dessus de la plateforme de simulation SimGrid. Pour cela, nous allons présenter en section 2 les différents types de virtualisations qui existent. Nous verrons en section 3 les outils qui permettent de mettre en place la virtualisation que nous aurons choisie pour ce projet. Puis en section 4 nous présenterons les projets permettant de faire de l'émulation pour tester des applications dans un environnement distribué. Ensuite, nous expliquerons en section 5, pourquoi dans le cadre du projet Simterpose c'est la virtualisation légère par interception qui a été choisie et comment elle fonctionne. En section 6, nous expliquerons le travail qui a été effectué durant ce stage . Puis, Nous présenterons nos expériences et les résultats obtenus en section **??**. Pour finir, nous conclurons avec les sections 8 et 9.

2 Approches possibles pour la virtualisation légère

Dans cette section, nous présentons les différentes approches possibles pour virtualiser des applications distribuées afin de permettre leur étude. L'objectif général étant de permettre l'étude du comportement d'une application réelle sur une plateforme différente de celle sur laquelle l'expérience a lieue. On appellera la vraie plateforme hébergeant l'expérience "machine hôte" et celle sur laquelle on veut tester l'application "machine cible".

Il existe actuellement deux méthodes permettant de faire de la virtualisation : standard et légère.

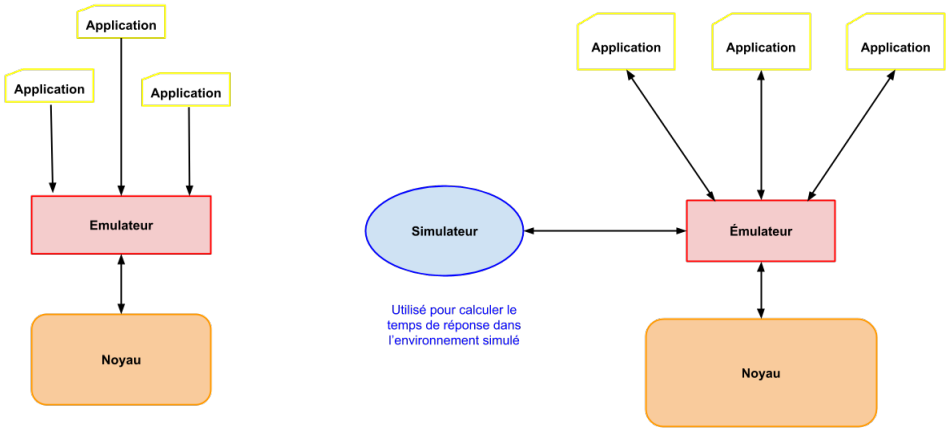
La première est une émulation complète de la machine comme celle utilisée par `vmware`. Dans cet émulateur, le système d'exploitation est hébergé sur une plateforme matérielle totalement virtualisée. Il n'a pas la possibilité de lancer toutes les instructions, l'émulateur déclenche un handler pour que l'hyperviseur les exécute. Cette solution pourrait être applicable dans le cadre de notre objectif général d'émulation d'applications mais elle ne passe pas du tout à l'échelle (plus d'une douzaine de machines).

La seconde est dite légère car elle permet de tester des applications sur une centaine d'instances. Pour que cela soit possible elle n'émule pas complètement la machine, notamment le CPU. De fait, elle ne peut exécuter que des applications ayant été compilées pour le même jeu d'instructions que celui de l'hôte. De plus, le choix de la virtualisation pour notre projet génère quatre problèmes : les threads, le temps, les communications réseaux et les échanges utilisant le protocole DNS, que notre plateforme devra résoudre. Malgré cette contrainte, c'est la virtualisation légère qui a été choisi pour notre projet car il est très important d'avoir une solution d'exécution rapide quelque soit le nombre d'instance.

La virtualisation légère peut se faire par limitation également appelée dégradation ou par interception. Nous allons donc étudier ces deux possibilités afin de choisir la plus adaptée à notre objectif et voir comment elle peut résoudre les problèmes cités précédemment.

2.1 Emulation par limitation

Avec cette première méthode, illustrée Figure 1a, on place la couche d'émulation au-dessus de la plateforme réelle (comme un hyperviseur pour une VM). De fait, la puissance de l'émulateur dépend de la puissance de la machine hôte et ne peux donc pas dépasser les capacités de cette dernière. De plus, en choisissant de placer l'émulation comme une surcouche, cela permet de limiter l'accès aux ressources pour les applications. En effet, elles ne pourront pas passer la couche d'émulation pour accéder aux ressources localisées sur la machine hôte. Les requêtes des applications seront arrêtées par l'émulateur. C'est lui qui s'occupera de récupérer les ressources demandées par les applications. Il existe différents outils permettant de mettre en place cette virtualisation, on trouve notamment **cgroups** [9] et **cpuburner** [20, 22] pour le système et **iptables** [13, 14] pour le réseau. L'émulation par limitation a l'avantage d'être simple à mettre en œuvre puisque l'on se base sur la machine hôte. Néanmoins, elle est assez contraignante du fait qu'on ne puisse pas émuler des architectures plus performantes que l'hôte.



(a) Virtualisation par limitation.

(b) Virtualisation par interception.

Figure 1 – Approches de virtualisation légère.

2.2 Emulation par interception

Dans le cas de l'émulation par interception, illustrée Figure 1b , pour mettre en place un environnement distribué émulé sur lequel les applications penseront s'exécuter, deux outils vont être utilisés ; un simulateur pour virtualiser l'environnement d'exécution, et un émulateur qui va intercepter toutes les communications de l'application avec l'hôte et qui les transmettra ensuite au simulateur.

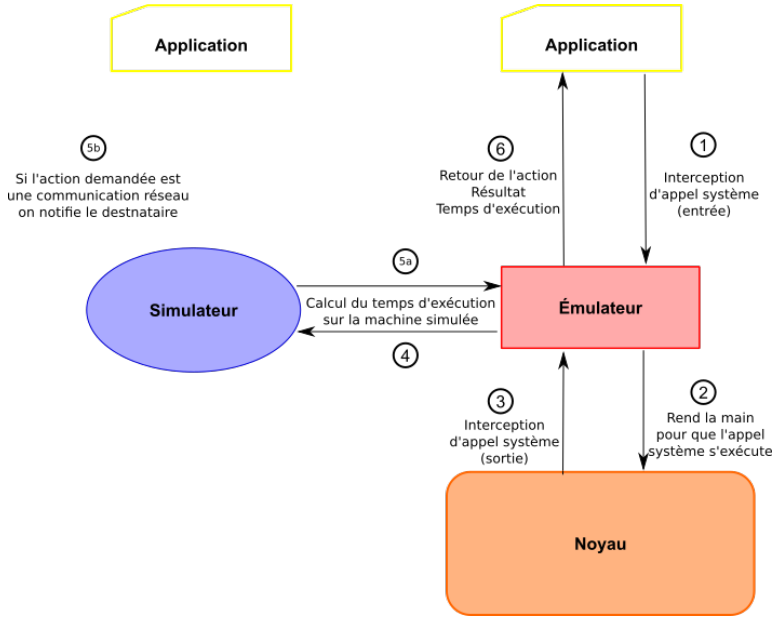


Figure 2 – Fonctionnement de l'émulation par interception.

Une application distribuée peut vouloir interagir avec son environnement soit pour effec-

tuer de simples calculs, soit pour effectuer des communications avec d'autres applications sur le réseau. Quand l'émulateur intercepte une communication venant d'un des processus d'une application, il modifie les caractéristiques de la communication pour qu'elle puisse s'exécuter sur la machine hôte. Il effectue l'opération inverse lorsque cette dernière envoie une réponse à l'application. En parallèle, l'émulateur demande au simulateur de calculer le temps d'exécution dans l'environnement virtuel de l'action demandée par l'application. L'émulateur renvoie ce temps à l'application en plus du résultat du calcul demandé pour mettre à jour son horloge. Ainsi, les calculs sont réellement exécutés sur la machine, les communications réellement émises sur le réseau géré par le simulateur et c'est le temps de réponse qu'il fourni qui va influencer l'horloge de l'application. Finalement, les applications ne communiquent plus directement entre elles.

Dans cette section, nous avons pu voir qu'il existe différents types de virtualisation. Puisque nous souhaitons pouvoir utiliser des milliers de plateformes durant nos exécutions nous ne pouvons utiliser la virtualisation complète de la machine. La virtualisation légère par dégradation est également exclue car nous devons pouvoir émuler des machines plus performantes que l'hôte. La virtualisation légère par interception semble être celle qui correspond le mieux aux besoins de notre projet. Pour pouvoir être mise en place elle nécessite d'utiliser des outils d'interception pour les différentes actions de l'application. Il existe différents outils permettant de faire cela, nous allons les présenter dans la section suivante.

3 Outils pour la virtualisation légère

Afin d'intercepter les actions d'une application il faut d'abord choisir à quel niveau se placer. En effet, une application peut communiquer avec le noyau via différentes abstractions, Figure 3. Elle peut soit utiliser les fonctions d'interaction directe avec le noyau que sont les appels systèmes, soit utiliser les différentes abstractions fournies par le système d'exploitation : bibliothèques (fonctions de la `libc` par exemple) ou les fonctions POSIX dans le cas d'un système UNIX.

Nous allons donc voir comment on peut intercepter et modifier des actions au niveau de l'application (fichier source puis binaire), des appels système et des appels de fonctions. Par la suite nous appelerons médiation l'ensemble des modifications effectuées par l'émulateur sur les actions interceptées.

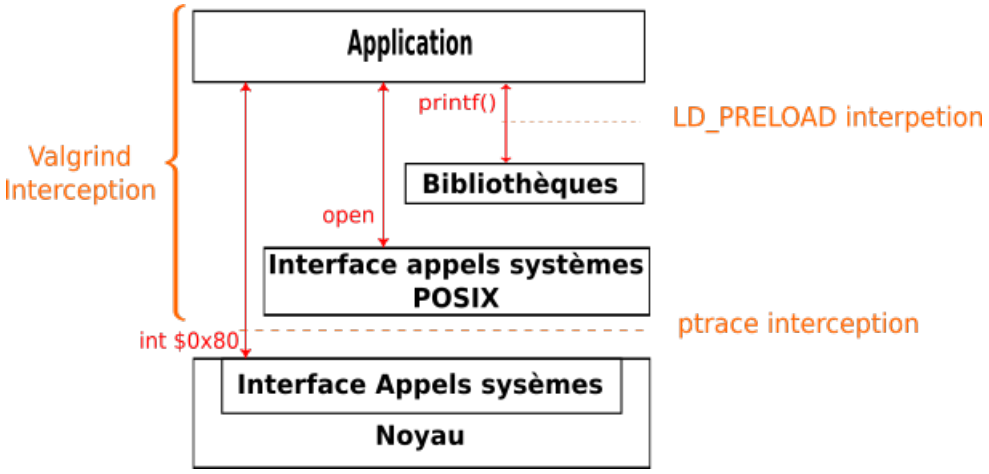


Figure 3 – Communications possibles entre le noyau et une application.

3.1 Action sur le fichier source

Le premier niveau auquel on peut se placer pour intercepter les actions est le fichier source de l'application. On peut avant de compiler le code réécrire les parties qui nous intéressent.

Un premier outil pour cela est le programme `Coccinelle` [10]. Il permet de trouver et transformer automatiquement des parties spécifiques d'un code source C. Pour cela, `Coccinelle` fournit le langage `SmPL` [6] permettant d'écrire les patchs sur lesquels il va se baser pour transformer le code. Un patch contient une suite de règles, chacune transforme le source en ajoutant ou supprimant du code. Lors de son exécution, `Coccinelle` scanne le code et cherche les lignes qui remplissent les conditions des règles spécifiées dans le patch et applique les transformations correspondantes. Dans notre cas, il s'agirait de toutes les actions de communications directes ou indirectes avec le noyau susceptibles de mettre à jour l'environnement virtuel. Néanmoins, il ne faut pas oublier de définir une règle pour chacune de ces actions sinon l'interception sera contournée. De plus, il faut pouvoir accéder au fichier source pour le modifier, or cela n'est pas toujours possible.

Une seconde solution beaucoup plus spécifique est de réimplémenter totalement la bibliothèque de communications utilisée par l'application. Cette approche est par exemple utilisée par

6. Semantic Patch Language

le projet SMPI[7] [8, 24], qui réimplémente le standard MPI au dessus du simulateur SimGrid. Cette approche manque de généricité car ce travail est à refaire pour chaque bibliothèque de communication existante.

3.2 Action sur le binaire

Pour agir sur le binaire d'une application, c'est l'outil d'instrumentation d'analyse dynamique Valgrind [16, 35] que nous allons étudier. D'autres outils suivent la même approche générale, tels que DynInst, un outil pour la modification à la volée de binaire utilisé notamment en HPC [2].

À l'origine, Valgrind était utilisé pour le débogage mémoire, puis il a évolué pour devenir l'instrument de base à la création d'outils d'analyse dynamique de code, tels que la mise en évidence de fuites mémoires ou le profilage[8]. Valgrind fonctionne à la manière d'une machine virtuelle faisant de la compilation à volée[9]. Ainsi, ce n'est pas le code initial du programme qu'il envoie au processeur de la machine hôte. Il traduit d'abord le code dans une forme simple appelée "Représentation Intermédiaire". Ensuite, un des outils d'analyse dynamique de Valgrind peut être utilisé pour faire des transformations sur cette "Représentation Intermédiaire". Pour finir, Valgrind traduit la "Représentation Intermédiaire" en langage machine et c'est ce code que le processeur de la machine hôte va exécuter. De plus, grâce à la compilation dynamique, Valgrind peut recompiler certaines parties du code d'un programme durant son exécution et donc ajouter de nouvelles fonctions au code de l'application.

Dans notre cas, on peut utiliser Valgrind pour mesurer le temps passé à faire un calcul. Ce dernier étant ensuite envoyé au simulateur pour calculer le temps de réponse dans l'environnement simulé nécessaire à l'émulateur. On peut également l'utiliser pour réécrire à la volée le code des fonctions que l'émulateur doit modifier pour maintenir la virtualisation. Pour faire cela, il faut créer un "wrapper" pour chaque fonction qui nous intéresse. Un wrapper est une fonction de type identique à celle que l'on souhaite intercepter, mais ayant un nom différent (généré par les `macro` de Valgrind) pour la différencier de l'originale. Pour générer le nom du wrapper avec les `macro` de Valgrind on doit préciser la bibliothèque qui contient la fonction originale. Cela implique donc de connaître pour chaque fonction à intercepter le nom de la librairie qui l'implémente. Cette solution est donc assez contraignante et ses performances sont assez médiocres d'après l'étude faite par M. Guthmuller lors de son stage [27] : facteur de 7.5 pour le temps d'exécution d'une application avec cet outil. Cette perte de performance est due à la compilation faite en deux phases ainsi qu'au temps nécessaire aux outils de Valgrind pour modifier ou rajouter du code à l'existant. Cela pourrait être acceptable, si Valgrind faisait de la traduction dynamique lors de la seconde phase de sa compilation, nous permettant ainsi d'avoir du code exécutable sur un autre type de processeur que celui de l'hôte, mais ce n'est pas le cas. Néanmoins, même si on résout le problème de performance, la mise en œuvre de cette approche restera difficile.

7. Simulation d'appplications MPI
8. Méthode visant à analyser le code d'une application pour connaître la liste des fonctions appelées et le temps passé dans chacune d'elles.
9. Technique basée sur la compilation de byte-code et la compilation dynamique. Elle vise à améliorer la performance de systèmes bytecode-compilés par la traduction de bytecode en code machine natif au moment de l'exécution.

3.3 Médiation des Appels Systèmes

En regardant la Figure 3 et les différents niveaux d'abstractions, le moyen le plus simple pour attraper les actions de l'application en gérant un minimum de choses semble être l'interception directe des appels systèmes. Ces derniers sont constitués de deux parties ; l'entrée initialise l'appel via les registres de l'application qui contiennent les arguments de l'appel puis donne la main au noyau. La sortie inscrit la valeur de retour de l'appel système dans le registre de retour de l'application, les registres d'arguments contenant toujours les valeurs reçues à l'entrée de l'appel système, et rend la main à l'application. Nous devons donc bloquer l'application à chaque interception d'une deux parties de l'appel système. Ainsi, on pourra récupérer et modifier les informations permettant de maintenir l'environnement simulé avant de rendre la main à l'application, pour pouvoir entrer ou sortir de l'appel système.

Dans cette section, nous allons présenter les outils existants qui permettent de faire cela.

3.3.1 L'appel système ptrace

L'appel système `ptrace`[27, 28], dont la Figure 4 illustre le fonctionnement, permet de tracer tous les événement désirés d'un processus. Il peut également lire et écrire directement dans l'espace d'adressage de ce dernier, à n'importe quel moment ou lorsque un événement particulier se produit. De cette façon, on peut contrôler l'exécution d'un processus. C'est un appel système dont chaque action à effectuer est passée sous forme de requête en paramètre de l'appel système.

Pour pouvoir contrôler un processus via `ptrace`, on va créer deux processus via un `fork` ; un processus appelé "processus espionné" qui exécutera l'application qu'on souhaite contrôler, et un autre qui contrôlera le processus espionné, appelé "processus espion". Le processus espionné indiquera au processus espion qu'il souhaite être contrôlé via un appel système `ptrace` et une requête `PTRACE_TRACEME` puis il exécutera l'application via un `exec`. À la réception de cet appel, le processus espion notifiera son attachement au processus espionné via un autre appel à `ptrace` et une requête `PTRACE_ATTACH`. Il indiquera également sur quelles actions du processus espionné il veut être notifié (chaque instruction, signal, sémaphore...), définissant ainsi les actions bloquantes pour le processus espionné. Dans notre cas, ce seront les appels systèmes que l'on considérera comme points d'arrêts (requête `PTRACE_SYSCALL`). Ainsi, le processus espion sera donc appelé deux fois : à l'entrée et à la sortie de chaque appel système.

Quand un processus de l'application voudra faire un appel système, il sera bloqué avant de l'exécuter et le processus espion qui lui est associé sera notifié via un appel système `ptrace`. Ce dernier effectue alors les modifications nécessaires dans les registres du processus espionné pour conserver la virtualisation de l'environnement. Ensuite, il rendra la main au processus espionné pour que l'appel système puisse avoir lieu. Le même fonctionnement est utilisé pour le retour de l'appel système. Le processus espion change simplement le temps d'exécution de l'appel système et l'horloge de l'application en utilisant ceux calculés par le simulateur. Quand un processus espion a fini un suivi, il peut envoyer deux types de requêtes au processus espionné : `PTRACE_KILL` qui termine le processus espionné ou `PTRACE_DETACH` qui le laisse continuer son exécution.

Néanmoins, pour contrôler un processus, `ptrace` fait de nombreux changements de contexte pour pouvoir intercepter et gérer les événements, or cela coûte plusieurs centaines de cycles CPU. De plus, il supporte mal les processus utilisant du multithreading, et ne fait pas partie de la norme POSIX. Ainsi il peut ne pas être disponible sur certaines architectures et son exécution peut varier d'une machine à une autre.

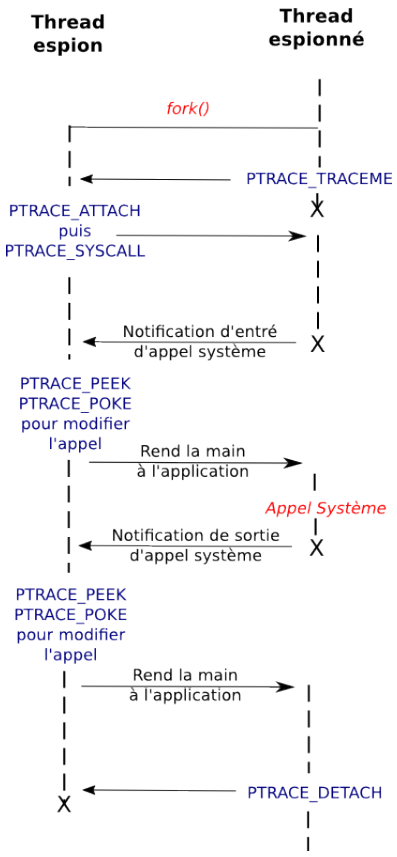


Figure 4 – Attachement d'un processus et contrôle via un espion.

Nous tenons à faire remarquer que `strace`[10] suit la même approche que `ptrace` sans avoir ses désavantages, mais qu'il a été écarté lors d'un précédent stage à cause de sa licence trop restrisctive.

3.3.2 Uprobes

Uprobes[27, 28], pour *user-space probes*, est une API noyau permettant d'insérer dynamiquement des points d'arrêts à n'importe quel endroit dans le code d'une application et à n'importe quel moment de son exécution. Nous pouvons donc l'utiliser avec les appels systèmes comme points d'arrêts.

La façon la plus classique d'utiliser cette interface se base sur Utrace, équivalent de `ptrace` en mode noyau. Ce dernier permet d'éviter les nombreux changements de contexte, qui dégradent les performances, et offre une meilleure gestion du multithreading. Dans cette version, l'utilisateur fournit pour chaque point d'arrêts un handler particulier à exécuter avant ou après l'instruction marquée. Uprobes étant un outil s'exécutant dans le noyau, les handlers doivent être placés dans un module noyau. Ce dernier contient pour chaque point d'arrêt géré par Uprobes le handler à exécuter, ainsi que le pid du processus concerné et l'adresse virtuelle du point d'arrêt. Pour gérer un point d'arrêt Uprobes utilise trois structures de données *i)* `uprobe_process` (une par processus controlé), *ii)* `uprobe_task` (autant que le processus contrôlé a de threads), *iii)* `uprobe_kimg` (une pour chaque point d'arrêt affectant un procesus). Chaque structure `uprobe_task` et `uprobe_kimg` sont propres à une structure `uprobe_process`. La fonction `init` du module va poser les points d'arrêt et la fonction `exit` les enlevera. Pour cela, on utilise respectivement la fonction `register_uprobe` et `unregister_uprobe`. Ces deux fonctions ont pour argument le pid du processus à contrôler, l'adresse virtuelle du point d'arrêt dans le code et le handler à exécuter quand le point d'arrêt est atteint. La fonction `register_uprobes` va trouver le processus passé en paramètres en parcourant la liste des structures `uprobes_process` ou la créera si cette dernière n'existe pas. Ensuite, elle crée la structure `uprobe_kimg`, puis fait appel à Utrace pour bloquer l'application, le temps de placer le point d'arrêt dans le code de celle-ci. Pour cela, on va insère avant l'instruction sondée un appel au module contenant le handler à invoquer, puis on rend la main à l'application en utilisant de nouveau Utrace. `unregister_uprobe` fait de même mais supprime la structure `uprobe_kimg` passée en paramètre au lieu de l'ajouter. De plus, s'il s'agit de la dernière structure de ce type pour un processus contrôlé, il supprimera alors la structure `uprobe_process` et toutes les `uprobe_task` associées.

Lorsqu'un point d'arrêt est atteint Uprobes prend la main et exécute le handler correspondant. Pour savoir qu'un point d'arrêt a été atteint, Uprobes utilise de nouveau Utrace, ce dernier envoyant un signal à Uprobes à chaque fois que le processus qu'il contrôle atteint un point d'arrêt.

Utrace envoie également un signal à Uprobes quand un des processus contrôlé fait un appel à `fork`, `clone`, `exec`, `exit` pour que ce dernier créé ou supprime les structures `uprobe_process` concernées. Utrace peut également être utilisé dans le handler gérant un point d'arrêt pour récupérer des informations sur l'application et les données qu'elle utilise. De plus, un handler peut également ajouter ou enlever des points d'arrêts.

Les deux avantages de cette solution sont qu'elle est rapide et qu'elle a accès à toutes les ressources sans aucune restriction. Mais ce dernier point représente aussi son plus gros défaut de

10. http://linux.die.net/man/1/strace

par sa dangerosité. De plus, dans notre cas il ne semble pas judicieux de faire de la programmation noyau via un module dont l'utilisateur devra également gérer le bon chargement.

3.3.3 Seccomp/BPF :

`seccomp` [15] est un appel système qui permet d'isoler un processus en lui donnant le droit de n'appeler et de n'exécuter qu'un certain nombre d'appels systèmes : `read`, `write`, `exit` et `sigreturn`. Si le processus fait un autre appel système, il sera arrêté avec un signal `SIGKILL`. Comme cela est assez contraignant, le nombre d'applications que l'on peut utiliser avec `seccomp` est donc limité. Pour plus de flexibilité, on peut utiliser une extension de cet appel système appelée seccomp/BPF, pour *seccomp BSD Packet Filter*, permettant de définir dans un programme BPF [33] les appels systèmes autorisés à s'exécuter, en plus de ceux cités précédemment. Cette extension fonctionne sur le même principe que le filtrage de paquet réseau où on établit une suite de règles. Pour pouvoir s'exécuter, un appel système doit pouvoir passer à travers toutes les règles. Dans le cas où les appels systèmes `fork` ou `clone` peuvent s'exécuter, l'arborescence de filtres est transmise aux enfants, de même que pour les processus faisant des appels `execve` quand ils sont autorisés. Les règles des filtres BPF portent sur le type de l'appel système et/ou ses arguments. Ainsi, à chaque entrée ou sortie d'un appel système, ne faisant pas partie des quatre autorisés par `seccomp`, l'extension utilisant BPF est appelée. Elle reçoit en entrée le numéro de l'appel système, ses arguments et le pointeur de l'instruction concernée. En fonction des règles, elle laisse l'appel système s'exécuter ou pas. De plus, seccomp/BPF possède une option qui lui permet de générer un appel système `ptrace`. Cela permet au processus espion, s'il existe, de ne plus attendre sur chaque appel système du processus espionné, mais uniquement sur les appels systèmes qu'il souhaite intercepter.

L'appel système seccomp et son extension seccomp/BPF sont disponibles uniquement si le noyau est configuré avec l'option `CONFIG_SECCOMP` pour la première et `CONFIG_SECCOMP_FILTER` pour la seconde. Pour pouvoir créer des filtres, il faut également avoir des droits particuliers, notamment l'exécution de certaines commandes administrateurs. Ainsi, l'utilisation de cet appel système et de son extension demande une certaine configuration noyau et des privilèges pour les utilisateurs.

De plus, si on l'utilise sans l'option d'appel à `ptrace`, on ne peut que lire le contenu de l'appel système et pas le modifier. On ne peut donc pas faire de médiation avec cet outil sans faire appel à `ptrace`. Néanmoins, l'utilisation de seccomp/BPF avec `ptrace` permet de réduire signifiquativement le nombre d'événements sur lequel attendra le processus espion.

Malgré ses défauts, `ptrace` semble être le meilleur outil pour intercepter des appels systèmes.

3.4 Médiation directe des appels de fonctions

Nous avons vu que l'interception des actions d'une application au plus bas niveau ne suffit pas, une autre solution est d'intercepter les actions de l'application au plus haut niveau que sont les bibliothèques. Pour cela nous allons étudier deux approches basées sur l'éditeur de liens dynamiques de Linux qui permet d'insérer du code dans l'exécution d'un programme.

3.4.1 LD_PRELOAD :

L'utilisation de la variable d'environnement `LD_PRELOAD` [5], contenant une liste de bibliothèques partagées, va nous permettre d'intercepter les appels aux fonctions qui nous intéressent et d'en modifier le comportement. Cette variable est utilisée à chaque lancement d'un programme par l'éditeur de liens pour charger les bibliothèques partagées qui doivent être chargées avant toute autre bibliothèque (même celles utilisées par le programme). Ainsi, si une fonction est définie dans plusieurs bibliothèques différentes, celle utilisée par le programme sera celle qui est contenue dans la bibliothèque partagée apparaîssant en premier dans la liste des bibliothèques préchargées. Ce ne sera pas nécessairement celle de la bibliothèque attendue par le programme. Par exemple, on créé une bibliothèque partagée qui implémente une fonction `printf` de même prototype que la fonction `printf` de la libc et on place cette bibliothèque dans la variable `LD_PRELOAD`. Quand on exécute un programme faisant un appel à `printf`, l'éditeur de lien va d'abord charger les bibliothèques contenues dans la variable d'environnement `LD_PRELOAD` puis la libc, la nouvelle bibliothèque apparaîtra donc avant la libc dans la liste des bibliothèques préchargées. Ainsi, c'est la nouvelle fonction `printf` qui sera exécutée par le programme et non l'originale. De cette façon, on peut intercepter n'importe quelle fonction.

Dans notre cas, on va donc créer notre propre bibliothèque de fonctions. Pour chaque fonction que l'on souhaite intercepter, on créera une fonction de même nom et de même type dans notre bibliothèque. Chacune de nos fonctions contiendra alors toutes les modifications nécessaires pour maintenir notre environnement simulé, suivi d'un appel à la fonction initiale. On rappelle que dans notre cas, on souhaite juste intercepter l'appel et pas l'empêcher. Notre nouvelle bibliothèque sera préchargée avant les autres en la plaçant dans la variable `LD_PRELOAD`, ainsi nos fonctions passeront avant les fonctions des bibliothèques usuelles.

Néanmoins, si l'application fait un appel système directement sans passer par la couche *Bibliothèques* (Figure 3) notre mécanisme d'interception est contourné. En effet, avec cette solution on ne peut surcharger que des fonctions définies dans les bibliothèques chargées dynamiquement, et non les appels systèmes directement. De même, si on oublie de réécrire une fonction d'une des bibliothèques utilisée par l'application. De plus, `LD_PRELOAD` étant utilisé pour les bibliothèques chargées dynamiquement, les bibliothèques statiques chargées à la compilation utilisant des fonctions à intercepter seront oubliées. Ainsi, l'interception au niveau des appels de fonctions ne permet pas une interception complète.

3.4.2 GOT Poisoning :

À la compilation, les adresses des appels de fonctions appartenant à des bibliothèques partagées ne sont pas connues. On associe alors un symbole à chaque appel d'une de ces fonctions. C'est lors de l'exécution du programme que l'éditeur de lien dynamique résoudra le symbole en trouvant l'adresse de la fonction à laquelle il correspond. Cette adresse sera ensuite stockée dans la "Global Offset Table"[3], également appelée GOT. Ce tableau, stocké dans le segment de données d'un exécutable ELF, sauvegarde pour chaque symbole résolu l'adresse de la fonction correspondante. Ainsi, lors du premier appel à la fonction, l'éditeur de lien retrouve l'adresse du symbole et pour les appels suivants il parcourt la GOT au lieu de refaire le calcul.

La technique du "GOT poisoning" [4] permet d'injecter de fausses adresses de fonctions dans la GOT lors de l'édition de lien dynamique d'un programme. Ainsi, pour chaque fonction de bibliothèque partagée que l'on souhaite intercepter, on peut remplacer l'adresse associée au symbole correspondant à la fonction par l'adresse d'une nouvelle fonction que l'on aura

implémentée. Comme avec `LD_PRELOAD` il ne faut pas oublier de fonctions sinon l'interception sera contournée.

En comparant avec l'interception via `LD_PRELOAD`, la seule différence est que la variable d'environnement `LD_PRELOAD` n'est pas lue lors de l'exécution de code `setuid` entraînant un problème d'interception. Dans notre cas, on ne s'intéresse pas aux problèmes de sécurité, nous avons donc choisi de ne pas développer cette solution et de nous concentrer sur `LD_PRELOAD`.

Dans cette section, nous avons présenté différentes approches permettant de faire de l'interception et de la médiation d'actions d'applications, résumées dans la table 1. Dans le cas d'émulateur ne souhaitant pas modifier le code source d'une application, les outils présentés en **??** sont inutiles. De plus, de par le surcoût d'utilisation de Valgrind, cette solution est à écarter dans le cas d'applications distribuées large échelle s'exécutant dans un environnement distribué. L'outils Uprobes est également à exclure car nous ne souhaitons pas écrire de code noyau. De plus, nous pouvons voir qu'il y a une certaine complémentarité entre l'appel système `ptrace` et la variable d'environnement `LD_PRELOAD`. En effet, `LD_PRELOAD` résout les lacunes de `ptrace` concernant les fonctions de temps et le multithreading. A l'inverse, `ptrace` permet d'intercepter les appels systèmes que l'on ne peut pas gérer avec `LD_PRELOAD`.

	ptrace	Uprobes	seccomp/BPF	LD_PRELOAD	Valgrind
Niveau d'interception	Appel Système	Appel Système	Appel Système	Bibliothèque	Binaire
Coût	Moyen	Faible	Moyen	Faible	Important
Mise en oeuvre	Assez complèxe	Assez Complexe	Assez complexe	Simple	Complexe
Utilisé pour	- Thread (incomplet) - Echanges réseau	- Thread (incomplet) - Echanges réseau	- Thread (incomplet) - Echanges réseau	- Thread (incomplet) - Temps - DNS	- Thread - Temps - Echanges réseau - DNS

Table 1 – Comparaison des différentes solutions d'interception entre une application et le noyau.

4 Projets de virtualisation légère

Maintenant que nous avons étudié différents outils permettant de gérer différents aspects de la virtualisation légère, nous allons présenter différents projets qui l'utilisent. Ils sont tous basés sur des architectures différentes et utilisent certains outils présentés dans la section précédente.

4.1 CWRAP

cwrap[11, 12] a pour but de tester des applications réseaux s'exécutant sur des machines UNIX ayant un accès réseau limité et sans droits administrateur. Ce projet libre a débuté en 2005 avec le test du framework "smbtorture" de Samba [11]. Pour atteindre son objectif, cwrap fait de l'émulation par interception basée sur le préchargement de quatre bibliothèques via `LD_PRELOAD`, comme nous l'avons vu en 3.4.1.

La première, `socket_wrapper` [11], gère les communications réseaux. Elle modifie toutes les fonctions liées aux sockets afin que toutes les communications soient basées sur des sockets UNIX et que le routage soit fait sur le réseau local émulé. Cela permet de pouvoir lancer plusieurs instances de serveur sur la même machine hôte. On peut également utiliser les ports privilégiés (en dessous de 1024) sans avoir les droits administrateur dans le réseau local émulé pour communiquer. Cette bibliothèque permet aussi de faire des captures réseau. La seconde, `nss_wrapper` [11], est utilisée dans le cas d'applications dont les démons doivent pouvoir gérer des utilisateurs. Pour cela, elle va modifier le contenu des variables d'environnement spécifiant les fichiers `passwd` et `group` qui vont être utilisés par l'application pendant la phase de test. Par défaut, les variables contiendraient les fichiers `passwd` et `group` du système mais dans ce cas le démon ne pourrait pas les modifier. `nss_wrapper` permet également de fournir un fichier `host` utilisé pour la résolution de noms lors de communications entre sockets. La troisième bibliothèque, appelée `uid_wrapper` [11], permet de simuler des droits utilisateurs. Autrement dit, elle fait croire aux applications qu'elles s'exécutent avec des droits qui ne sont pas les leurs, par exemple une exécution avec des droits administrateur. Pour cela, on intercepte les appels de type `setuid` et `getuid` et on réécrit le mapping fait entre l'identifiant de l'appelant et celui passé en paramètre pour le remplacer par un identifiant possédant les droits désirés. La dernière librairie, `resolv_wrapper` [11], gère les requêtes DNS. Elle intercepte ces requêtes et soit les redirige vers un serveur DNS de notre choix spécifié dans `resolv.conf`, soit utilise un fichier de résolution de noms que l'on a fourni à l'application.

Ainsi, on a un système permettant de tester des applications utilisant des réseaux complexes. Néanmoins, on utilise uniquement `LD_PRELOAD` pour intercepter les actions, or nous avons vu en section 3.4.1 que cette approche est incomplète. De plus, cette architecture ne gère pas les problèmes de virtualisation liés au CPU et à la gestion du temps.

4.2 RR

La plus grande partie de l'exécution d'une application est déterministe. Néanmoins, il reste des instructions non déterministes entraînant un exécution toujours différente de l'application (signaux, adresses de buffers...). Elles peuvent conduire à des fautes qui sont persistantes ou qui apparaissent après un certain nombre d'exécutions ou qui sont totalement aléatoires et peuvent ne pas réapparaître lors de la réexécution de l'application. Essayer de résoudre ces bugs de façon

11. https://www.samba.org/
https://wiki.samba.org/index.php/Writing_Torture_Tests

conventionnelle étant très difficile, il faut trouver de nouvelles méthodes. C'est pour cela que RR a été créé. RR [6] est outil de débogage utilisant l'émulation par interception et qui vise à surpasser gdb. Il a été créé pour déboguer Firefox, mais il peut-être utilisé sur n'importe quel type d'application. Il résout le problème des exécutions non déterministes en deux phases. La première consiste à enregistrer l'exécution de tous les événements non déterministes qui pourraient échouer. La seconde débogue l'exécution de façon déterministe en rejouant l'enregistrement aussi souvent qu'on le souhaite. On relance toujours la même exécution et les ressources restent les mêmes (espace d'adressage, contenu des registres, appels systèmes), d'où l'idée de déterminisme. Avec cette méthode, on peut même déboguer les fautes qui sont produites par des outils de fuzzing [12] ou d'injection de fautes. Néanmoins, pour des raisons d'efficacité RR ne sauvegarde pas la mémoire partagée lors d'exécutions multi-thread. Ce choix permet de n'émuler qu'une machine mono-cœur qui est plus simple à gérer même si cela empêche le parallélisme.

RR utilise différents outils selon la phase de son exécution[7]. Dans la phase d'enregistrement, pour gérer l'ordonnancement lors du rejeu, il sauvegarde les actions qu'il considère comme mécanisme d'interruption d'une application : *i)* les appels systèmes exécutés *ii)* la préemption via HPC [13] en sauvegardant le nombre d'instructions à exécuter avant une interruption *iii)* les signaux UNIX exécutés ainsi que leur handler s'il est réimplémenté. Dans la phase de rejeu, RR utilise les données non déterministes sauvegardées lors de la première phase pour mettre en place son émulation (appels systèmes, compteur d'instructions, handler de signal, valeurs des registres lors de ces actions). Quand RR va rejouer un appel système, les valeurs de retour des registres seront celles sauvegardées lors de l'exécution réelle et non celles du rejeu. RR utilise l'outil `LD_PRELOAD` que nous avons vu en section 3.4.1 pour intercepter les appels systèmes et les placer dans un buffer. Ensuite Seccomp/BPF(section 3.3.3) parcourra le buffer pour filtrer les appels système et les laisser exécuter. Pour cette partie de la gestion de l'appel on n'utilise pas `ptrace` car il est trop couteux en terme de changement de contexte, Figure 5. Il sera utilisé pour renvoyer le bon résultat à l'application (celui sauvegardé lors de la première phase) et gérer les autres événements de l'application notamment les signaux et les HPC. Pour pouvoir déboguer l'application on va utiliser les commandes gbd (placer les points d'arrêts, continuer l'exécution...). En utilisant `gdb` à l'intérieur de son outil de débogage, RR veut essayer de le surpasser en terme d'efficacité.

De par son fonctionnement, RR permet donc de diminuer le temps de débogage. De plus, il peut fonctionner avec de nombreuses applications puisqu'il arrive à gérer une grosse application telle que Firefox. Le surcoût de la phase d'enregistrement par rapport à un simple debogage avec gdb varie selon les applications et les tests effectués. Néanmoins, le fait que RR n'enregistre pas la mémoire partagée en multi-tâche est un problème pour déboguer des threads. De plus, il émule une machine simplement mono-cœur ce qui est un problème pour l'utilisation du parallélisme. Tous les appels système ne sont pas encore implémentés et en fonction de l'application à tester on risque de voir apparaître un problème d'interception de certains appels système exécutés par les processus.

12. Technique pour tester des logiciels basée sur l'injection de données aléatoires dans les entrées d'un programme. Si le programme échoue : plantage ou génération d'erreur, alors il y a des défauts à corriger.
http://fr.wikipedia.org/wiki/Fuzzing

13. Hardware Performance Counters
On compte les instructions qui s'exécutent et on arrête l'application quand le nombre d'instrucions exécutées atteint la valeur du HPC fournie par l'utilisateur.

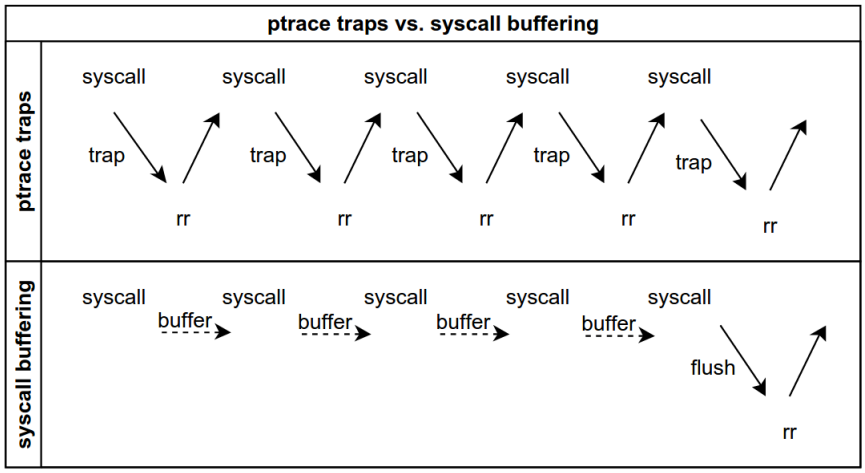


Figure 5 – Comparaison de l'exécution du rejeu de RR avec `ptrace` et seccomp/BPF pour gérer l'exécution des appels systèmes.

4.3 Distem

Distem [37] est un outil libre permettant de construire des environnements expérimentaux distribués virtuels. Pour cela il fournit un système de virtualisation de nœuds, une émulation des cœurs du processeur de la machine hôte et du réseau. À partir d'un ensemble de nœuds homogènes, il peut émuler une plateforme de nœuds hétérogènes connectés via un réseau lui-même virtuel.

Cet outil qui se veut simple d'utilisation propose différentes interfaces selon les besoins et les compétences de l'utilisateur. De plus, il supporte parfaitement le passage à l'échelle puisqu'en 2014, 40 000 nœuds ont été émulés en utilisant moins de 170 machines physiques [21]. Le prochain objectif étant de réussir à émuler 100 000 machines.

Pour construire un environnement distribué virtuel, Distem fait de la virtualisation par limitation telle que nous l'avons définie dans la section 2.1. On commence par spécifier la latence et la bande passante en entrée et en sortie de chaque lien du réseau virtuel. Ensuite, on définit les performances de chaque nœud émulé. Autrement dit, et comme représenté sur la Figure 6, on alloue à chaque nœud virtuel un certain nombre de cœurs du processeur de la machine physique dont on pourra controller la fréquence individuellement.

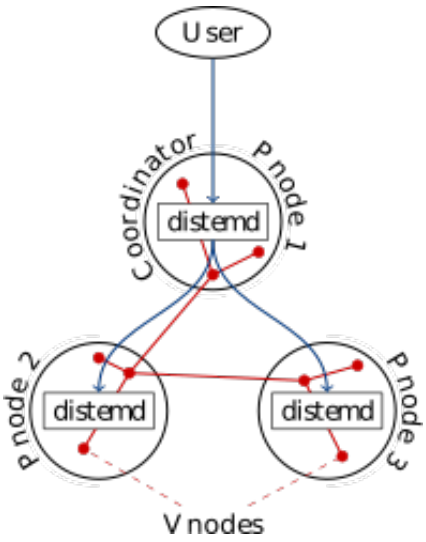


Figure 7 – Architecture de communications de Distem. Ici on a 3 nœuds physiques (“Pnodes”) contenant chacun 3 nœuds virtuels (“Vnodes”).

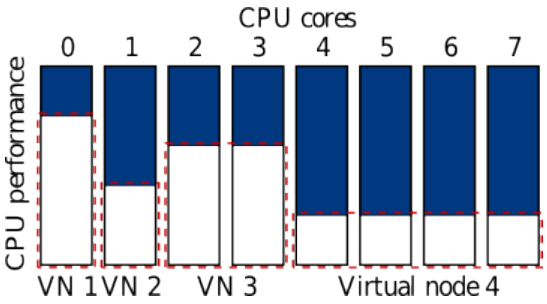


Figure 6 – Répartition des cœurs d’un processeur d’une machine hôte entre les différents nœuds virtuels qu’elle héberge et émulation de leur puissance en n’utilisant qu’une partie de leur puissance.

On construit ensuite l’environnement de test en plaçant les nœuds virtuels sur une machine physique. Pour que l’environnement de test se rapproche au plus près de la réalité, Distem peut changer à la volée les paramètres du réseau et la vitesse de chaque cœur alloué à un nœud virtuel.

Comme le présente la Figure 7, Distem repose sur une architecture simple pour construire son environnement de test distribué virtuel : les “Pnodes” et les “Vnodes”. Les premiers sont des nœuds physiques non virtualisés alors que les seconds représentent les nœuds que l’on souhaite émuler. Un des Pnodes appelé “coordinator” gère le contrôle de l’infrastructure dans sa globalité en communiquant avec l’ensemble des Pnodes. Ces derniers peuvent héberger plusieurs Vnodes, chaque Pnode possèdant son démon Distem qui contrôle les Vnodes. Les Vnodes sont séparés et n’ont pas connaissance des autres Vnodes présents sur le Pnode. Pour permettre cela, Distem utilise un conteneur LXC pour émuler un Vnode. Ainsi, chaque Vnode possède un espace d’adressage séparé pour les ressources sytème (tâches, interfaces réseau, mémoire...). Néanmoins, les conteneurs LXC partagent l’utilisation du processeur, ainsi on ne peut pas attribuer un certain nombre de cœurs de CPU à un Vnode. Pour pallier ce problème, Distem utilise en parallèle

cgroups [9]. Pour contrôler la puissance des cœurs attribués à chaque Vnode, Distem utilise l'algorithme CPU-Hogs [14] [19]. Ainsi les Vnodes ont connaissance les uns des autres uniquement via le réseau virtuel. Chaque Vnode possède une ou plusieurs interfaces réseau virtuelles reliées au réseau physique de l'hôte pour pouvoir communiquer avec l'extérieur. Du fait du grand nombre de nœuds qu'on souhaite émuler et qui vont communiquer entre eux cet accès au réseau extérieur pose problème. En effet, pour se reconnaître les nœuds vont faire des requêtes ARP et s'ils sont trop nombreux à envoyer ces requêtes en même temps on va se retrouver face à un problème d'ARP flooding. La première solution mise en place par Distem a été d'augmenter la taille des tables ARP pour les Pnodes et les Vnodes ainsi que l'augmentation du *timeout* d'une entrée dans la table. Néanmoins, le but de Distem étant de pouvoir émuler de plus en plus de nœuds cette solution finira par ne plus pouvoir s'appliquer. Une autre solution, qui est celle utilisée actuellement, est de rajouter une couche d'abstraction réseau à l'intérieur du Pnode en utilisant VXLAN[21, 32] comme le montre la Figure 8. Ainsi, les paquets seront échangés entre Pnodes sur le réseau et c'est la couche VXLAN qui s'occupera d'envoyer au bon Vnode le paquet reçu sur le Pnode. Ces derniers étant très peu nombreux on est sûrs de ne pas surcharger les tables ARP.

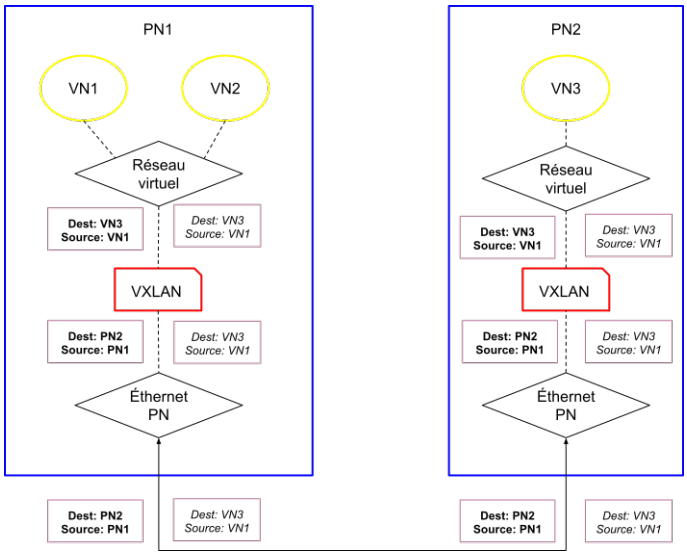


Figure 8 – Abstraction des communications réseaux de Distem via VXLAN. Les paquets en gras sont ceux envoyés en présence de VXLAN et ceux en italiques sont ceux qui seraient envoyés sur un réseau n'utilisant pas VXLAN.

On voit donc que Distem possède une infrastructure et un réseau émulé bien détaillés et assez réalistes. De plus il est capable de gérer les fautes injectées au niveau des nœuds ou sur le réseau. Son seul problème est donc d'utiliser la virtualisation par limitation empêchant ainsi l'émulation de machines plus puissante que l'hôte.

14. Méthode de dégradation des performances du CPU qui consiste à créer un processus pour qu'il occupe le CPU pendant un certain temps.

4.4 MicroGrid

Les grilles sont des environnements très hétérogènes, que ce soit en terme de configuration, de performance ou de fiabilité. Les logiciels qui les utilisent doivent donc avoir une certaine flexibilité pour pouvoir s'adapter aux différents environnements et ressources disponibles. Il existe des applications qui permettent de vérifier que l'accès aux ressources d'une grille est équitable et sécurisé. Néanmois, il n'y a pas d'outil pour étudier le comportement d'applications développées pour utiliser de tels environnements et exploiter leurs ressources. On ne peut donc pas connaître la robustesse et l'efficacité de ces applications, ainsi que l'impact qu'elles ont sur la stabilité de la grille.

MicroGrid [39, 41] est un émulateur par interception créé pour résoudre ce problème. Il fournit via l'émulation une grille virtuelle large échelle permettant d'exécuter des applications, sans qu'elles soient modifiées, selon les ressources disponibles sur la grille et les différentes topologies réseaux qui peuvent être utilisées par l'application. De plus, la virtualisation permet de gérer toutes les grilles, que leurs ressources soient homogènes ou hétérogènes. Grâce au simulateur, MicroGrid peut émuler des machines plus puissantes et donc tester les applications sur des grilles qui n'existent pas encore. En définissant les ressources et le réseau à émuler on peut prédire les performances d'applications développées pour la grille. Le but n'est pas d'avoir des prédictions parfaites mais de parvenir au moins à des estimations de performances qui soient fiables dans le cas de l'exécution d'une application utilisant une topologie inexistante. L'émulateur virtualise l'environnement et le simulateur modélise les ressources de la grille (calcul, mémoire et réseau) pour calculer le temps dans l'environnement virtuel.

Pour maintenir la grille virtuelle, l'émulateur doit gérer deux types de ressources : celles pour le réseau et celles pour le calcul. Dans le premier cas, l'émulateur intercepte toutes les actions faites par l'application qui vont utiliser les ressources simulées (`gethostname`, `bind`, `send`, `receive`). L'interception se fait au niveau des bibliothèques, via `LD_PRELOAD`. Ces appels sont ensuite transmis à l'émulateur de paquets réseau utilisé par MicroGrid, MaSSF, pour gérer les communications qui vont réellement transiter sur le réseaux. MaSSF est capable de gérer de très nombreux protocoles réseaux. Pour ce qui est des ressources de calcul, MicroGrid utilise un contrôleur de CPU qui virtualise les ressources du CPU et gère les processus des machines virtuelles via des `SIGSTOP` et `SIGCONT`. Ce contrôleur agit à trois niveaux *i)* il intercepte les appels de fonctions pour créer ou tuer des processus, toujours via `LD_PRELOAD`, pour maintenir une table des processus virtuels à jour *ii)* périodiquement il mesure le temps d'utilisation du CPU de chaque processus contenu dans sa table *iii)* il ordonne les processus de chaque hôte virtuel qui sont contenus dans sa table en fonction des mesures qu'il fait.

Dans le cas de grilles hétérogènes il faut obtenir une simulation équilibrée. Autrement dit une simulation qui ne crée pas de délais à cause des temps de réponses qui diffèrent entre les plateformes. Pour cela, il faut mettre en place un mécanisme de coordination globale. MicroGrid utilise un temps virtuel pour coordonner l'écoulement du temps sur les différentes plateformes.

L'avantage de la solution proposée par MicroGrid est qu'elle peut utiliser de nombreux protocoles réseaux complexes pour émuler un réseau réaliste et qu'elle permet d'émuler des machines avec des vitesses très variables grace au contrôleur qui gère la simulation.

Le gros problème de MicroGrid est que le temps n'est pas intercepté mais émulé par dilatation [30]. Il est très compliqué de trouver le bon facteur de dilatation et de conserver la synchronisation qu'il engendre. De plus, le réseau peut ne pas gérer parfaitement cette dilatation et prendre du retard sur le CPU.

Aujourd'hui il n'existe plus de version maintenue de MicroGrid mais l'approche utilisée par ce projet a été réutilisée dans de nombreux projet notamment Timekeeper [15] [29] et Integrated simulation and emulation using adaptative time dilation [16] [30].

4.5 DETER

Dans le domaine de la cyber-sécurité, le test de solutions de défenses proposées face aux différentes menaces n'est pas simple et se développe lentement. En effet, de nombreuses ressources sont nécessaires et il ne semble pas judicieux d'effectuer les tests en environnement réel. De plus, les innovations qui fonctionnent parfaitement dans des environnements contrôlés et prédictibles sont souvent moins efficaces et fiables dans la réalité de par la taille du réseau et des ressources qui constituent son environnement. C'est pour cela que l'USC [17] et l'UC Berkeley [18] ont lancé le projet DETER [1, 17, 34]. À sa création en 2003, DETER était un projet de recherche avancée visant à développer des méthodes expérimentales pour les innovations en matière de cyber-sécurité (contrer les cyber-attaques, trouver les failles réseaux ...). Puis en 2004, le besoin de tester ces méthodes se faisant de plus en plus sentir, le développement de DeterLab [19] a été lancé. Cette plateforme d'émulation par interception libre et partagée fournit un environnement de test large échelle et réaliste. Elle permet également d'automatiser et de reproduire des expériences pouvant être de différentes natures. En effet, DeterLab peut *i)* observer et analyser le comportement de cyber-attaques [20] et de technologies de cyber-défense, *ii)* tester et mesurer l'efficacité des solutions de défenses proposées pour contrer les menaces. Les utilisateurs accèdent aux machines dont ils ont besoins pour leurs expériences, demandent une certaine configuration du réseau, du système et des applications présentes sur les machines. Pour fonctionner, ce laboratoire virtuel a développé 7 outils complémentaires. Seuls 4 outils nous intéressent dans le cadre de notre projet [21].

Le premier, qui constitue le cœur logiciel et hardware de DeterLab, se base sur Emulab [40]. DeterLab a étendu ce dernier pour permettre de faire des tests large échelle spécialisés dans le domaine de la cyber-sécurité et dont la complexité est représentative des réseaux d'aujourd'hui (nombre de nœuds, hétérogénéité des plateformes, contrôle de la bande passante et délai). Il fournit également une interface web pour gérer à distance ses expériences, les projets en développement et accéder aux autres outils de DeterLab.

Pour gérer les ressources nécessaires à leurs expériences, les chercheurs du projet DETER ont créé "The DeterLab Containers" (Figure 9). Ces derniers permettent de virtualiser les ressources et donc de répartir la puissance de calcul là où elle est nécessaire. Ainsi, pour des ressources

15. Permet à chaque conteneur LXC d'avoir sa propre horloge virtuelle et de pouvoir faire des pauses ou des sauts dans le temps. Pour cela la fonction `gettimeofday` a été réimplémentée afin qu'elle renvoie un temps virtuel.
16. Ce projet dilate le temps pour garder une certaine synchronisation entre applications et simulateur. Quand le simulateur est surchargé il prend du retard sur le temps réel et introduit des délais quand il répond à l'émulateur. Ici la fonction `gettimeofday`() a été remplacée par une `fonction get_virtual_time` et l'émulation se fait par dégradation avec un émulateur type KVM.
17. University Southern California
18. Université de Californie à Berkeley
19. DeterLab : cyber DEfense Technology Experimental Research Laboratory
20. Attaques DDos et botnets, vers et codes malicieux, protocoles de stockage anti-intrusion (intrusion-tolerant), ainsi que le chiffrement et la détection de *pattern*.
21. Les 3 autres sont le simulateur de comportement humain DASH, "Multy-party Experiments" pour avoir une vision partielle du monde lors d'une expérience et "Risky Experiment Management Capability" pour gérer les échanges d'expériences avec le réseau extérieur.

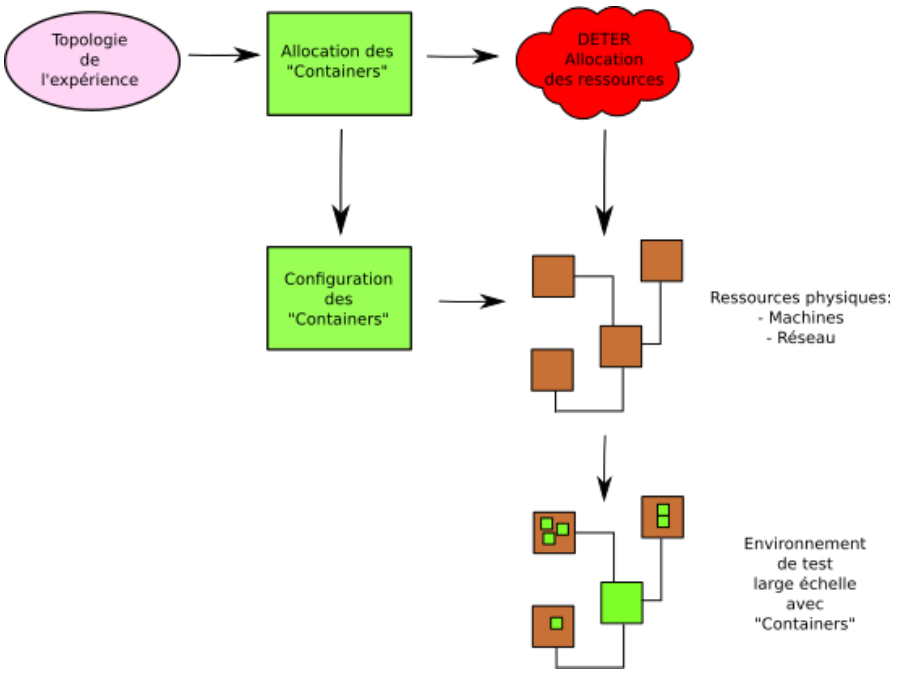


Figure 9 – Diagramme du fonctionnement d'un *Container*.

nécessitant une machine entière, le conteneur sera la machine alors que pour une ressource qui n'aura besoin que d'une partie de la machine, le conteneur sera une abstraction de cette partie de la machine contenant les ressources utilisées par l'application. Cela permet d'isoler les tests qui n'utilisent pas une machine complète et de partager ses ressources entre plusieurs tests concurrents. Ce mécanisme de virtualisation s'appelle la "Multi-resolution Virtualization".

Actuellement, il existe plusieurs plateformes de tests basées sur Emulab avec des extensions pour pouvoir être utilisées dans des domaines spécifiques comme le fait DETER. Il se peut qu'une expérience exécutée sur une de ces plateformes aie besoin de plus de machines que la plateforme ne peut en fournir, que ce soit en terme de nombre, de puissance ou d'hétérogénéité des machines. Pour pallier ce problème, le projet DETER a créé la "Federation"[25]. Elle permet de déployer une expérience sur plusieurs plateforme de tests différentes et d'avoir un plus grand facteur de passage à l'échelle. La Figure 10 montre l'exécution d'une expérience dont les 3 nœuds sont répartis sur différentes plateformes. La difficulté ici est que les plateformes sont contrôlées par des propriétaires différents ayant des règles de sécurité d'accès souvent très différentes de celles du projet DETER.

Pour que cette solution fonctionne, les plateformes vont partager un système de nommage permettant de ne pas montrer à l'application la répartition de ses ressources sur le réseau et une authentification pour contrôler l'accès d'une plateforme à une autre. Pour gérer cela on va utiliser trois types de nœuds différents (Figure 10). Le premier est le "federating", il est unique et se place sur la plateforme qui demande à exécuter une expérience. Il cherche les ressources disponibles sur les différentes plateformes puis divise l'expérience en sous-expérience qu'il assigne à chaque plateforme. Il gère l'exécution de l'expérience, récupère les données à la fin et libère les ressources utilisées. Le second type de nœud est le "federated", on en place un sur chaque plateforme. C'est lui qui fournit la liste des ressources disponibles sur sa plateforme de tests et

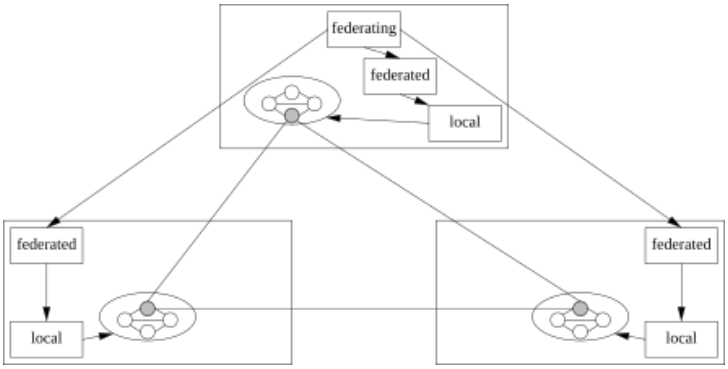


Figure 10 – Fédération d'une expérience répartie sur 3 plateformes de tests différentes.

qui configure les sous-expériences qu'il reçoit du federating. Il fait la traduction de nom entre le federating, qui utilise le système de nommage partagé, et le réseau local qui utilise son système de nommage spécifique. Il gère également la mise en place des connexions réseaux entre les entités réparties sur les différentes plateformes. Le dernier nœud appelé "local" est également présent sur chaque plateforme où l'expérience va s'exécuter. Il gère les communications entre les différentes plateformes et les sécurise pour éviter les fuites à l'extérieur du réseau ou l'espionnage par d'autres applications s'exécutant sur la même plateforme.

Pour finir, MAGI [22] fournit un système de gestion de flux entre les différentes entités d'une expérience, permettant ainsi d'avoir un certain contrôle sur les machines. En gérant le flux, on peut automatiser et reproduire les expériences. En effet, MAGI capture chaque séquence d'instructions concurrentes que l'expérience va suivre pour gérer le flux, ainsi on peut rejouer la capture plus tard avec les paramètres d'origine ou des nouveaux si un fichier de paramètres à tester existe. MAGI permet également de visualiser l'évolution d'une expérience en cours d'exécution pour s'assurer que son comportement reste correct sans avoir à attendre le résultat final. En capturant les configurations demandées par les utilisateurs, MAGI permet leur réutilisation par d'autres utilisateurs pour éviter à DETER de reconstruire la même architecture pour une prochaine expérience.

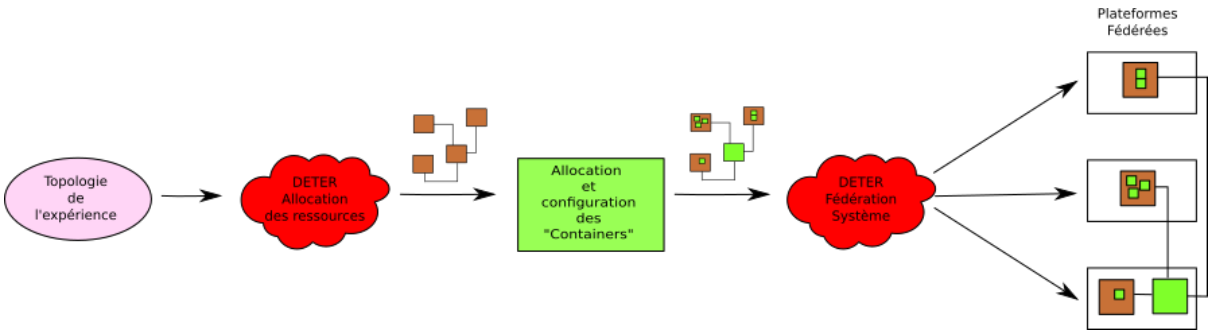


Figure 11 – Création de l'environnement d'une expérience.

Actuellement, DeterLab peut émuler des dizaines de milliers de nœuds : le projet dispose de 500 machines et 10 FPGA. Il est le seul émulateur dans le domaine de la cyber-sécurité et

22. Montage AGent Infrastructure

permet de faire des tests large échelle dont la complexité est représentative des réseaux.

5 Simterpose : la médiation

Dans la section précédente, nous avons présenté différents projets permettant de faire de la virtualisation légère. Malheureusement, ils ne pouvaient pas être utilisés dans le cadre de notre projet. Nous allons donc voir pourquoi et quel est l'émulateur qui a été choisi pour permettre d'exécuter des applications distribuées au dessus de SimGrid. Puis, nous étudierons le fonctionnement interne de cet émulateur, notamment les outils présentés en section 3 qu'il utilise.

5.1 Organisation générale

Dans le cadre du projet Simterpose de virtualisation légère et de test d'applications distribuées, c'est l'émulation par interception qui a été choisie. En effet, le but final étant de pouvoir évaluer n'importe quelle application distribuée sur n'importe quel type d'architecture, on peut se retrouver à devoir émuler des machines plus puissantes que l'hôte, ce que l'émulation par dégradation ne permet pas. Ce choix exclut donc l'utilisation de Distem dans notre projet. Les autres outils de virtualisation par interception ont été écartées pour des raisons différentes : CWRAP utilise uniquement `LD_PRELOAD` pour intercepter les actions et est spécifique aux applications réseaux. RR gère le multithread mais pas la virtualisation réseau. MicroGrid utilise une émulation par dilation pour gérer le temps. Or, dans notre cas, nous voulons faire de l'interception.

Pour satisfaire les besoins de notre projet, il a été décidé de créer un nouvel émulateur Simterpose. Ce dernier doit être simple d'utilisation et facilement déployable (simple ordinateur ou cluster). De plus, il doit permettre d'exécuter plusieurs instances d'une application sur une même machine et de proposer une large gamme de conditions d'exécutions (simple nœud ou réseau complexe). Les expériences devant être reproductibles, l'émulateur doit pouvoir générer des traces pour les rejouer dans le simulateur. La résistance aux pannes, et le fait de pouvoir fonctionner sans avoir accès au fichier source de l'application sont également des conditions à satisfaire.

Simterpose utilise la plateforme de simulation SimGrid, dont l'architecture est représentée (Figure 12), pour générer l'environnement virtuel nécessaire à l'expérience. À sa création en 1999, SimGrid [23] était une plateforme fournissant des outils pour construire un simulateur afin d'étudier les algorithmes d'ordonnancement en environnement hétérogène, puis il a évolué [36] pour devenir plus générique. Aujourd'hui, c'est devenu une plateforme de simulation permettant d'évaluer les applications distribuées large échelle s'exécutant dans des environnements hétérogènes.

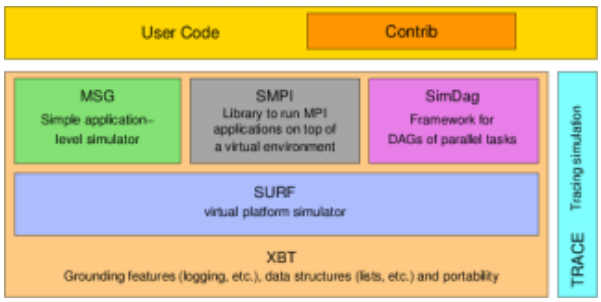


Figure 12 – Architecture de la plateforme SimGrid.

La Figure 13 présente l'organisation générale de la plateforme de simulation. SimGrid va

générer l'environnement virtuel. Simterpose intercepte les actions de l'application et les modifie pour maintenir l'émulation si nécessaire. Puis, il les laisse s'exécuter sur la machinie hôte et va interroger SimGrid pour qu'il calcule la réponse de l'environnement virtuel aux actions de l'application (temps d'exécution, gestion de communications réseaux). La Figure 14 illustre ce fonctionnement.

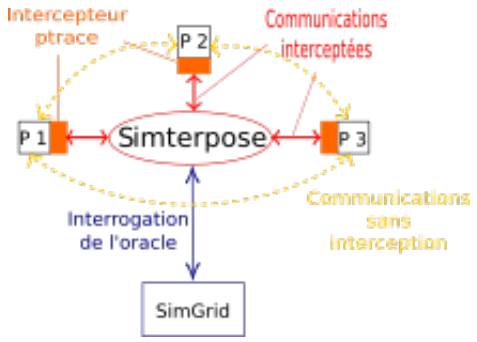


Figure 13 – Architecture de communication entre les différents acteurs.

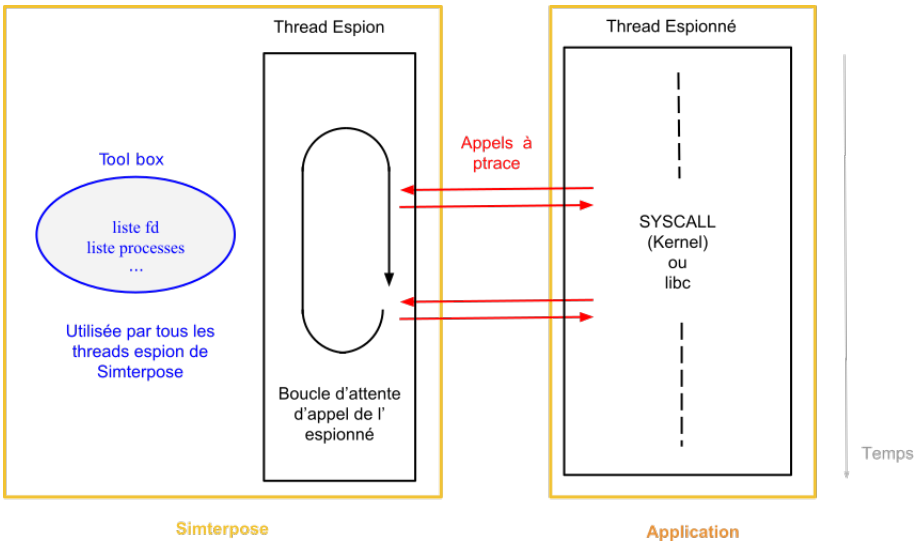


Figure 14 – Le fonctionnement de Simterpose.

Maintenant que nous avons présenté le cadre général et l'organisation globale de notre projet, nous allons nous intéresser au fonctionnement de Simterpose.

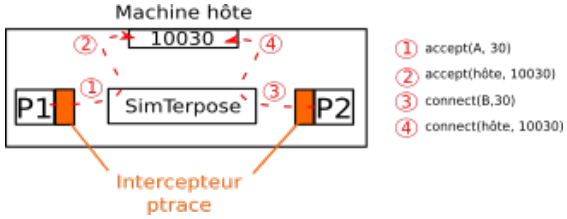
5.2 Fonctionnement interne de Simterpose

Les actions à intercepter pour maintenir la virtualisation sont de différentes natures. Il peut s'agir d'actions liées *i)* aux communications réseaux, *ii)* à la création et à l'identification des processus, *iii)* à la gestion du temps, *iv)* à l'utilisation du protocole DNS. Pour chacune, les outils utilisés ne sont pas nécessairement les mêmes. Nous allons donc voir lesquels sont employés parmi ceux cités dans la section 3.

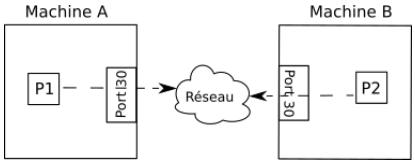
5.2.1 Les communications réseaux

Dans le cas d'une communication réseau, le but de Simterpose étant de réussir à simuler un réseau virtuel sur un réseau local, il faut gérer la transition entre réseau local et réseau simulé, comme le montre la Figure 15. En effet, l'application possède une adresse IP et des numéros de ports virtuels qui ne correspondent pas à ceux attribués dans le réseau local.

La gestion de cette transition va se faire au niveau des appels systèmes, car nous ne souhaitons pas prendre le risque d'oublier des fonctions en utilisant `LD_PRELOAD`. Pour intercepter les appels systèmes liés au réseau nous allons utiliser l'appel système `ptrace` présenté en section 3.3.1.



(a) Communications réelles.

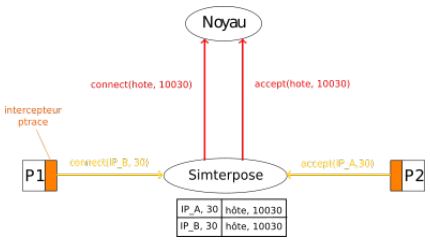


(b) Communications vues par les processus.

Figure 15 – Les communications réseau entre deux processus.

De plus on ne peut pas se baser uniquement sur le numéro de *file descriptor* associé à une socket pour identifier deux entités qui communiquent entre elles. En effet ce *file descriptor* est unique pour chaque socket d'un processus, mais plusieurs processus peuvent avoir un même numéro de *file descriptor* pour des sockets de communications différentes puisque chacune à son propre espace mémoire. Pour pallier à ce problème on va utiliser en plus du numéro de socket, les adresses IP et les ports locaux et distants des deux entités qui souhaitent communiquer comme moyen d'identification.

Afin de gérer toutes ces modifications deux solutions ont été proposées lors d'un précédent stage [38] : la *médiation par traduction d'adresse* et la *full médiation*. Néanmoins, ces deux solutions n'ont pas encore été évaluée en pratique.



(a) Médiation par traduction d'adresse.

(b) Full meditation.

Figure 16 – Les différents types de médiation.

Traduction d'adresse. Avec ce type de médiation, illustrée Figure 16a, on laisse le noyau gérer les communications. Ainsi, en entrée et sortie d'appel système, Simterpose va juste s'occuper de la transition entre le réseau virtuel simulé par SimGrid et le réseau local, en utilisant les informations de communications contenues dans la socket. Pour cela, Simterpose gère un tableau de correspondances, dans lequel pour chaque couple <IP, ports>virtuels , on a un couple <IP,

ports>réels associé. De fait, en entrée d'un appel système de type réseau (`bind`, `connect`, `accept` ...), Simterpose doit remplacer l'adresse et les ports virtuels de l'application par l'adresse et les ports réels sur le réseau local, afin que la source de l'appel système corresponde à une machine existante sur le réseau local. Au retour de l'appel système, il faudra remodifier les paramètres en remettant l'adresse et les ports virtuels. La limite de cette approche est liée au nombre de ports disponibles sur l'hôte.

Full médiation. Dans ce cas, le noyau ne va plus gérer les communications car nous allons empêcher l'application de communiquer via des sockets et même d'établir des connexions avec une autre application. Puisqu'il n'y a aucune communication, on n'a pas besoin de gérer de tableau de correspondance d'adresses et de ports et les applications peuvent conserver les adresses et les ports simulés qu'elles considèrent comme réels. Quand l'application voudra faire un appel système de type communication ou connexion vers une autre application, le processus espion de Simterpose qui sera notifié via `ptrace` neutralisera l'appel système, comme illustré sur la Figure 16b. Ensuite, ce processus récupérera, en lisant dans la mémoire du processus espionné, les données à envoyer ou récupérer et ira directement les lire ou les écrire dans la mémoire du destinataire.

5.2.2 Les threads

La gestion des threads se fera à deux niveaux dans Simterpose. On fera appel à l'interception d'appels systèmes via `ptrace` pour tout ce qui concerne la création de threads (`fork`, `clone`, `pthread_create`). Pour le reste on utilisera un autre outil, car certains mécanismes utilisés par les threads ne passent pas par des appels systèmes pour s'exécuter, par exemple les futex[23]. On utilise donc le préchargement de bibliothèques dynamiques en complément. Comme nous l'avons vu en section 3.4.1, nous allons créer une librairie contenant toutes les fonctions utilisées par les threads que nous voulons intercepter (`pthread_init`, `pthread_join`, `pthread_exit`, `pthread_yield` ...). On placera ensuite la bibliothèque dans la variable d'environnement `LD_PRELOAD` pour que nos fonctions passent avant les autres.

5.2.3 Le temps

Nous souhaitons que l'écoulement du temps perçu par les applications soit un temps virtuel, celui qui s'écoulerait dans l'environnement simulé, pour que l'émulation soit plus réaliste. Il y a deux types d'actions à différencier pour gérer le temps : les appels de fonctions liées au temps et les actions faites par l'application qui prennent du temps lors de leur exécution.

Dans le premier cas, Simterpose va intercepter les appels de fonctions temporelles et les modifier pour renvoyer l'heure virtuelle. Ces dernières étant assez nombreuses, il est moins coûteux, en terme charge de travail pour le programmeur, d'intercepter via `ptrace` les appels sytèmes temporels (`time`, `clock_gettime`, `gettimeofday`).

Néanmoins, il a été montré dans un précédent stage [31] que `ptrace` est inefficace voire inutile en ce qui concerne l'interception des appels systèmes temporels qu'une application souhaiterait exécuter car le noyau ne les exécute pas. Cela est dû à l'existence de la bibliothèque *Virtual Dynamic Shared Object* (`VDSO`). Cette dernière vise à minimiser les coûts dûs aux deux changements de contexte effectués lors de l'exécution d'un appel système. `VDSO` va retrouver

23. Fast User-space mutex http://man7.org/linux/man-pages/man7/futex.7.html

l'heure dans un segment du processus partagé avec l'espace noyau lisible par tous les processus sans changer de mode. Il est possible de désactiver cette bibliothèque lors du boot mais cela réduit les performances, augmente le nombre de changements de contexte, et oblige l'utilisateur à modifier les paramètres de son noyau.

On va donc se placer à un autre niveau pour intercepter ces fonctions. Malgré leur nombre, il a été décidé d'agir lors du préchargement de bibliothèque. On va créer une bibliothèque qui surcharge les appels de fonctions liées au temps et la placer dans la variable d'environnement `LD_PRELOAD`. Une fois l'interception temporelle effectuée, Simterpose interroge SimGrid pour obtenir l'heure virtuelle et la renvoie à l'application.

Dans le second cas, lors d'une action interceptée (calcul, communications réseau), Simterpose doit gérer l'horloge de l'application avant de lui renvoyer le résultat de l'exécution. Pour cela, Simterpose envoie à SimGrid l'heure et la durée d'exécution sur la machine hôte. Ce dernier calcule en fonction de ces deux informations, le temps qui aurait été nécessaire pour une telle exécution sur la plateforme virtuelle et l'envoie à Simterpose. Pour finir, l'émulateur rend la main à l'application en lui envoyant l'heure virtuelle en plus du résultat de son action.

5.2.4 DNS

Dans le cas d'utilisation du protocole DNS, on peut vouloir modifier le comportement de l'application afin qu'elle utilise d'autres serveurs que ceux utilisés par défaut, ou aucun afin que la résolution soit entièrement gérée par Simterpose.

Une première solution envisageable serait de faire de l'interception de communications au niveau du port 53, utilisé par défaut dans DNS. Néanmoins, cela est assez complexe à mettre en œuvre car il faut pour chaque communication faite par l'application tester le port qu'elle souhaite utiliser. De plus, il est possible que l'utilisateur définisse un autre port pour le protocole DNS que celui par défaut. Cette solution n'est donc pas suffisante.

ne autre approche serait de remplacer le fichier `resolv.conf` utilisé pour la résolution de nom par un fichier spécifié par l'utilisateur. Il pourrait également fournir un fichier de spécification de comportement en cas d'utilisation de DNS permettant à Simterpose de générer un nouveau fichier `resolv.conf`. Néanmoins, cette solution génère une surcharge de travail pour l'utilisateur et nous souhaitons avoir un émulateur qui soit simple d'utilisation.

La dernière solution envisageable est de faire de l'interception d'appel de fonctions. Dans cas, on créé une bibliothèque partagée qui réécrit les foncitons liées à la résolution de nom que l'on inclut dans la variable d'environnemnt `LD_PRELOAD`. Mais on a toujours le même problème qui est de n'oublier aucune fonction pour maintenir notre environnement virtuel. Pour l'instant, c'est la solution qui a été choisie. N'ayant pas encore été mise en place, il n'est pas exclu que nous devions trouver une autre solution pour gérer le DNS.

Dans cette section, nous avons étudié l'organisation globale de Simterpose ainsi que son fonctionnement interne. Nous avons pu voir que deux des outils présentés en section 3 sont utilisés pour implémenter notre émulateur : l'appel système `ptrace` et la variable d'environnement `LD_PRELOAD`.

6 Travail réalisé

Au cours de ce stage, deux des fonctionnalités majeures de Simterpose présentées en section 5.2 ont été implémentées : le réseau de communications et la gestion du temps. Ces fonctionalités ont parfois nécessité la mise en place de nouveaux outils. De plus, plusieurs améliorations ont été apportées à notre émulateur. Dans cette section, nous allons présenter comment les deux fonctionnalités ont été implémentées ainsi que les outils qu'elles ont nécessités et les améliorations apportées à Simterpose.

6.1 Réseau de communications

Afin d'implémenter le réseau de communications, tel qu'il est présenté en section 5.2.1, nous avons écrit pour chaque appel sysèteme affectant le réseau une version de l'appel utilisant la *médiation par traduction d'adresses* et une utilisant la *full mediation*. Une des deux versions sera exécuté par `ptrace` à chaque interception de l'appel selon le type de médiation utilisée.

Dans la version de l'appel système utilisant la *médiation par traduction d'adresses*, nous allons récupérer via `ptrace` les valeurs contenues dans les registres. Ensuite, selon le type d'appel système réseau dont il s'agit on va effectuer des actions différentes. Pour les appels qui concernent la création de sockets ainsi que l'ouverture et la fermeture de connexion (`socket`, `bind`, `connect`, `listen`, `accept`, `shutdown`, `close`) on va créer dans la table de correspondance <IP, port>$_{virtuel}$ / <IP, port>$_{\text{réel}}$ un nouvelle entrée, si elle n'existe pas déjà. Cela nous permettra de maintenir la virtualisation en traduisant les couples <IP, port>$_{virtuel}$ en <IP, port>$_{\text{réel}}$ lors d'appels systèmes effectuant des communications sur le réseau. Pour les appels concernant les échanges de messages, on récupère en utilisant les tables de traductions le couple $< IP, port >_{\text{réel}}$ correspondant aux informations passées en paramètre. Pour finir, quelque soit l'appel système, on va écrire dans ses registres les valeurs traduites grâce à `ptrace`. Puis, on laisse l'appel système s'exécuter. A la sortie, on refait la même chose à la seule différence qu'on traduit le couple <IP, port>$_{reel}$ en <IP, port>$_{virtuel}$.

Pour la version de l'appel système qui utilise la *full mediation*, on récupère comme précédement les paramètres de l'appel système contenus dans les registres. Ensuite, comme dans cette médiation il faut empêcher les appels systèmes de s'exécuter on neutralise l'appel système via `ptrace` pour empêcher son exécution quand on rendra la main à l'application. Puis, en fonction du type d'appel système réseau dont il s'agit on effectue différentes actions. Pour les appels qui concernent la gestion du réseau (création de socket, ouverture et fermeture de connexion...) on ne fait rien de particulier puisque dans ce type de médiation aucune socket n'est créée et connectée. Pour les appels systèmes qui vont effectuer des échanges de messages on va créer une tâche SimGrid qui va permettre d'écrire ou de lire les données à envoyer ou recevoir. Puis, pour tous les types d'appels systèmes réseau, on va écrire dans les registres de l'appel avec `ptrace`. On écrit d'abord la valeur de retour de l'appel et éventuellement d'autres informations dans différents registres selon l'appel. C'est le cas par exemple, quand on reçoit des données, elles sont lues durant l'exécution de l'appel système par le noyau puis il les écrit dans le bufer d'écriture dont l'adresse passée en paramètre est contenue dans un registre.

6.2 Temps

Pour pouvoir maintenir notre environnement virtuel, nous avons expliqué en section 5.2.3 que nous ne pouvons pas laisser les applications accéder aux horloges de la machine hôte, utilisant

pour cela la bibliothèque `VDSO`. Nous avons alors proposé de créer une bibliothèque réimplémentant les fonctions temporelles et d'utiliser la variable d'envieronnement `LD_PRELOAD`, présentée en section 3.4.1, pour exécuter les fonctions de notre bibliothèque en priorité. De cette façon, on bloque les appels à la bibliothèque `VDSO` et l'application ne peut plus accéder au temps de la machine hôte. Néanmoins, pour que notre virtualisation soit parfaite il faudrait que le temps que l'applicatioin voit s'écouler soit celui qui s'écoulerait sur la machine simulée par SimGrid.

Pour permettre cela, nous avons créé une bibliothèque de fonctions temporelles que nous plaçons dans la variable d'environnement `LD_PRELOAD` à chaque exécution d'une application avec Simterpose. Cette bibliothèque contient pour chaque fonction de la `libc` qui fasse appel à une des horloges de la machine (`ftime`, `time`...) une nouvelle fonction de même prototype. Maintenant, au lieu de demander l'horloge de la machine hôte, les nouvelles fonctions de temps demandent à SimGrid de fournir l'heure sur la machine qu'il est en train de simuler. Comme le montre la figure **??**, l'appel à la fonction de temps se fait dans l'application qui s'exécute sur le thread "espionné" de Simterpose. Ce dernier est controlé par le processus 'espion' via l'intercepteur `ptrace` qui est le seul processus de Simterpose à pouvoir communiquer avec le simulateur. Nous devons donc passer par le thread "espion" pour pouvoir interrroger SimGrid.

pas clair *Cependant, le seul moyen de faire intervenir le processus "espion" est d'effectuer un appel système dans l'application en train de s'exécuter. Dans notre cas, c'est la nouvelle fonction temporelle qui doit effecteur l'appel système. En effet, le processus "espion" va intercepter l'appel et exécuter le "handler" qu'on lui aura fourni si nécessaire, comme pour le reseau de communications, avant de rendre la main à l'application en plaçant dans le registre de retour de l'appel système l'horloge que lui aura donné le simulateur. Si une autre fonction effectue l'appel, la nouvelle fonction temporelle ne pourra pas récupérer la valeur de retour de l'appel système et ainsi retourner l'heure simulée à l'application. De plus, puisque nous souhaitons juste récupérer l'horloge de l'environnement virtuel et rien d'autre, il faudra que le "handler" que va exécuter `ptrace` avant l'appel contienne l'appel à la fonction `MSG_get_clock`, qui permet de récupérer l'heure dans SimGrid et neutralise l'appel système choisit avant de rendre la main à l'application.*

Néanmoins, la fonction temporelle ne doit pas utiliser n'importe quel appel système pour récupérer l'heure simulée. Par exemple, si on choisit de modifier l'appel système `send`, on va placer dans le handler de ce dernier l'appel à la fonction `MSG_get_clock` puis on le neutralise. Or, lorsqu'on voudra utiliser cet appel système pour envoyer des données sur le réseaux on ne pourra plus le faire car l'intercepteur `ptrace` exécutera le nouveau handler et l'appel ne sera plus exécuté. Il nous faut donc trouver un appel système dont on n'aura jamais besoin pour faire autre chose que récupérer l'heure simulée. Dans tous les noyaux, il existe des appels systèmes qui ne sont plus implémentés ; c'est le cas de l'appel système `tuxcall`. Quand on fait appel à ce dernier le système ne fait rien, il émet juste un avertissement pour dire que l'appel n'existe plus. Nous avons donc choisi de placer notre handler sur cet appel système.

Les Figures **??**, **??** et **??** nous montrent respectivement l'algorithme de la nouvelle fonction `time`, le handler de l'appel système `tuxcall`, et ce qui se passe dans SimGrid et Simterpose quand on veut récupérer l'heure simulée grâce à la double interception complémentaire de `LD_PRELOAD` et `ptrace`.

	Numéro de l'appel système	Valeur de retour	arg0	arg1	arg2	arg3	arg4	arg5
32bits	orig_eax	eax	edi	esi	edx	r10d	r8d	r9d
64bits	orig_rax	rax	rdi	rsi	rdx	r10	r8	r9

Table 2 – Nom des différents registres d'un appel système selon le type d'architecture.

6.3 Améliorations apportées à Simterpose

Au début de ce stage, Simterpose ne pouvait être exécuté que sur des architectures 64bits. Considérant qu'il est important qu'un tel programme puisse s'exécuter sur tous les types d'architecture nous avons voulu résoudre ce problème afin qu'il puisse s'exécuter sur des machines 32bits. Il existe plusieurs différences entre les architectures 32bits et 64bits. Celles qui nous intéressent sont celles qui sont susceptibles d'affecter Simterpose. La première différence est que les registres ne portent pas les mêmes noms en 64bits et 32bits, cf tableau 2. Or, lorsqu'on intercepte des appels systèmes il faut pouvoir récupérer les valeurs contenues dans les registre à l'entrée de l'appel système puis écrire dans ces mêmes registres à la sortie. Afin d'exécuter Simterpose sur ces deux architectures il faut donc avoir une version du code pour chacune. Chaque version utilisant les bons noms de registres pour récupérer les valeurs qu'ils contiennent lors de l'interception d'appels systèmes via `ptrace`. Pour trouver sur quelle type de plateforme Simterpose est en train de s'exécuter on teste la valeur maximale que peut avoir un pointeur en mémoire avec la `macro UINTPTR_MAX`. Pour avoir une architecture 64bits UINTPTR_MAX doit valoir 0xffffffffffffffff et pour être en 32bits elle doit valoir 0xffffffff. La seconde différences est que certaines `macro` et appels systèmes n'existent pas sur des architectures 64bits et inversement. C'est la cas de deux appels systèmes particulièrement importants car ils touchent au réseau : `send` et `recv`. Ces deux appels ne sont définis que pour des architectures 32bits, sur des architectures 64bits quand on les utilise ils sont remplacés par les appels sytèmes `sendto` et `recvfrom`.

La seconde amélioration est une mise à niveau de Simterpose pour qu'il puisse utiliser les dernières version de SimGrid. Pour cela, il a fallu remplacer d'anciennes fonctions et variables encore utilisées dans le code de Simterpose par les nouvelles utilisées dans SimGrid. Cela a également permi de mettre à jour un problème dans SimGrid dû à l'accès d'un pointeur dont on ne vérifiait pas qu'il n'était pas nul. Au début de mon stage, Simterpose utilisait une version de SimGrid datant de 2011, maintenant il utilise la version f42adf1 de git sortie le 16 Août 2015.

Troisièmememnt, nous souhaitons pouvoir utiliser un autre débugueur en plus de `gdb`. Nous avons choisi Valgrind pour les nombreux modules qu'il fourni,cité en section 3.2, notamment `memcheck` qui est pour nous le plus intéressant. Ce dernier traque les fuites mémoires et résume en fin d'exécution tout ce qui a été réservé, libérée et perdu en mémoire. Pour permettre son utilisation avec Simterpose nous avons dû implémenter l'appel système `fcntl`. Valgrind utilise cet appel système pour accéder à l'exécution de Simterpose et ainsi chercher les fuites mémoires. L'implémentation d'un handler pour cet appel n'était pas prévu à la création de Simterpose puisque nous souhaitons juste intercepter les appels systèmes réseaux, temporels et gérer les processus et leurs threads pour maintenir notre environnement virtuel et aucun débugueur autre que `gdb` ne souhaitait être utilisé à ce moment-là.

Pour finir, le but de Simterpose étant d'exécuter des applications distribuées large échelle, nous devons pouvoir exécuter des applciations de Torrent. De fait, lorsqu'on exécute des appli-

cations de ce type, le système de fichier de la machine hôte est en permanence utilisé. Simterpose dispose en parallèle de son propre système de fihier. Pour chaque socket ou fichier il dispose d'un descripteur avec des compteurs de références, les processus qui les référencent, les verrous qui peuvent être posés... Nous devons donc nous aussi maintenir à jour notre système de fichier dans le cas où nous lancerions ce genre d'applications. Ainsi, nous devons maintenant intercepter les appels systèmes touchant aux fichiers en plus de ceux affectant le réseau en utilisant toujours `ptrace`. Lorsqu'on les intercepte il faut récupérer les modifications qui seront effectués sur le système de fichiers réel une fois qu'on aura laisser passer l'appel système et les appliquer au système de fichier propre à Simterpose. Actuellement Simterpose gére les appels systèmes : `open`, `close`, `creat`, `dup`, `dup2`, `poll`, `fcntl`, `lseek`, `read`, `write`.

TODO pas d'idée

7 Évaluation des fonctionalités implémentées

Dans la section précédente, nous avons présenté le travail réalisé au cours de ce stage. Maintenant, nous souhaitons évaluer les performances des fonctionnalités que nous avons implémentées. Pour cela, nous allons d'abord présenter l'architecture utilisée. La plateforme sur laquelle nous avons effectuée nos expériences possède les caractéristiques suivantes :

— Distribution : ubuntu 3.13.0-62
— Processeur : 4 cœurs multithread à 2.60 GHz

En ce qui concerne SimGrid et Simterpose, les commits utilisés pour les expériences sont respectivement f42adf1 (16 Août 2015) et 77f7d81 (19 Août 2015).

Au niveau de la plateforme réseau simulée par SimGrid on a quates nœuds réliés les uns aux autres. Chacun a une puissance de 10^7 XXXX, une bande passante de 10^9 XXXX et une latence de $5.10^{-4}s$. Pour nos expériences on utilise seulement deux de ces nœuds, dont l'un joue le rôle de client et l'autre joue le serveur.

Maintenant que nous connaissons l'architecture utilisée, nous allons voir quelles expériences ont été faites pour chaque fonctionnalité et les résultats qu'elles ont apportés.

7.1 Réseaux

Dans les sections 5.2.1 et 6.1, nous avons présenté l'oganisation et l'implémenation du réseau de communications de Simterpose. Nous souhaitons maintenant évaluer les performances de notre implémenration à travers divers tests.

7.1.1 Protocole

L'objectif principal étant de montrer qu'il est possible de faire de la virtualisation légère, nous souhaitons d'abord mesurer l'*overhead* dû à l'utilisation de Simterpose. De cette façon, si le surcoût d'utilisation de notre émulateur est négligeable on pourra conclure que ce type de virtualisation est possible pour le réseau. Dans un second temps, nous souhaitons mesurer les performances des deux types de médiations que nous avons implémentés. Cela nous permettrait de savoir le type de médiation qu'il vaut mieux utiliser selon l'application que l'on souhaite exécuter.

Pour effectuer nos expériences, nous avons choisi deux applications réseau d'échanges de messages. La première application consiste à faire communiquer un client et un serveur en envoyant au serveur un million de messages de petite taille. La seconde va envoyer depuis un client un message de 1Mo à un serveur. Le protocole pour chaque expérience a été d'exécuter 20 fois chaque application en utilisant les mêmes appels systèmes pour communiquer les messages (`sendto`/`recvfrom` et `sendmsg`/`recvmsg`), puis une moyenne du temps d'exécution ainsi qu'une mesure des temps minimum et maximum ont été conservées.

7.1.2 Overhead conçernant le temps d'exécution

Afin de mesurer l'*overhead* produit par Simterpose, notre expérience consiste à exécuter les deux applications réseaux choisies sur une machine en utilisant uniquement le réseau local puis en utilisant Simterpose en suivant le protocole présenté plus haut. Ainsi, en comparant les temps d'exécution des applications sur les deux types d'architecture nous pourrons calculer l'*overhead*. Les résultats de ces expériences sont présentés Figures 17 et 18.

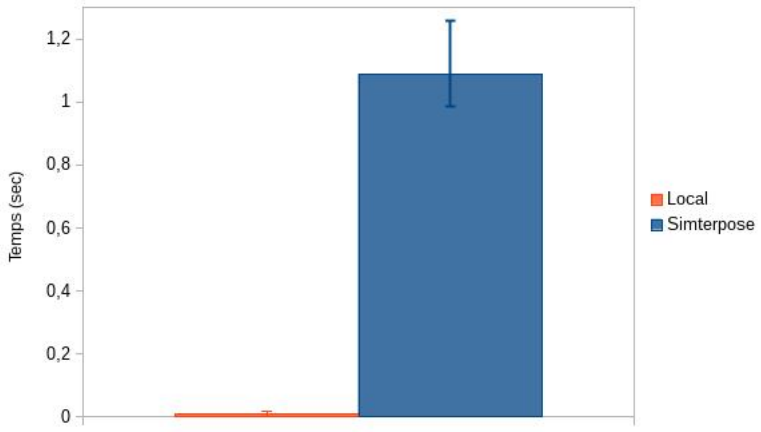


Figure 17 – Temps d'exécution lors de l'envoi d'un message de 1Mo avec et sans Simterpose.

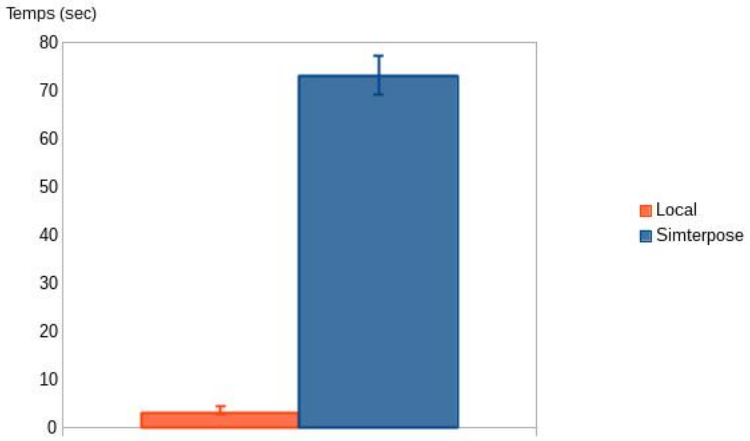


Figure 18 – Temps d'exécution lors de l'envoi d'un million de messages de 128o avec et sans Simterpose.

Dans le cas de l'envoi de plusieurs petits messages le temps moyen d'exécution en local est d'environ 3 secondes, avec Simterpose on arrive à 72 secondes en *full mediation* et à 75 secondes en *médiation par traduction d'adresse*. On a donc une exécution qui prend 25 fois plus de temps. De même, lors de l'envoi d'un gros message le système met 0,01 secondes en moyenne pour exécuter l'application alors que Simterpose met entre 1 et 1,1 secondes selon le type de médiation. Cette exécution prendra donc 100 fois plus de temps si on souhaite réellement mettre en place notre émulateur.

Néanmoins, cet écart s'explique par les nombreux appels systèmes et changements de contexte que nécessite Simterpose de par son utilisation coûteuse de `ptrace` en plus de l'exécution de l'appel système lui-même lorsqu'on utilise *médiation par traduction d'adresse*. Pour la première expérience on peut considérer que cet *overhead* représente un cas extrême car les utilisateurs en général ne testent pas leurs applications en envoyant des millions de message entre un client et un serveur. De plus, du point de vue humain une exécution de 1ms ou 1s dans le second cas ne change pas grand chose pour notre perception. Ainsi, on peut considérer que l'*overhead* produit

par Simterpose sur le temps d'exécution est acceptable.

7.1.3 Quelle médiation pour quel type d'application

Cette seconde expérience vise à comparer les deux médiations que nous avons implémentés. Pour cela, en suivant le protocole présenté plus haut on va exécuter les deux applications réseau choisi en utilisant les deux types de médiations. Les résultats de notre expérience sont présentés Figures 19 et 20.

Lors de l'envoi de nombreux petits messages, on peut constater que la *full mediation* est plus rapide que la *médiation par traduction d'adresse* avec un écart moyen de 3 secondes. Lorqu'on utilise la *médiation par traduction d'adresse* l'envoi de messages génère des appels systèmes et changements de contexte qui n'ont pas lieu en *full mediation* puisque dans ce cas, comme nous l'avons expliqué en section 5.2.1, les appels systèmes ne sont pas exécutés. Ces derniers étant coûteux cela explique pourquoi la *médiation par traduction d'adresse* est moins rapide. De plus, en *full mediation*, même si les appels systèmes sont bloqués, nous utilisons l'appel système `ptrace` pour effectuer nous-même les appels demandés par l'application que nous avons bloqués. Cette méthode même si elle reste moins coûteuse que l'exécution de l'appel système demandé consomme des cycles CPU, ce qui explique le faible écart entre les temps d'exécution moyen des deux types de médiation.

Lorsque l'on envoie un gros message, on constate que l'écart moyen entre les deux types de médiation est quasi nul, 0.03 secondes. Nous pensons que dans ce cas la *médiation par traduction d'adresse* est légèrement plus rapide car nous envoyons un seul message de 1Mo de données. En effet, même si en *full mediation* on a beaucoup moins de changement de contexte de par le blocage des appels systèmes ici on ne fait qu'un seul appel et la faible différence entre les deux médiations ne peut donc être dû à cela. Par contre, lors de l'envoi d'un tel message il faut prendre en compte la gestion de la mémoire car le message doit être stocké avant de pouvoir être envoyé par morceau sur le réseau. Ainsi, si on ne gère pas la mémoire de façon efficace on surconsomme des cycles CPU lorsqu'on accède à cette dernière. Or, nous n'avons pas encore mis en place de politique de gestion de mémoire particulière pour Simterpose. Néanmoins, si on regarde la largueur des intervalles des temps d'exécution, en *médiation par traduction d'adresse* les temps d'exécution varient bien plus qu'en *full mediation*. On peut donc supposer qu'avec une politique de gestion mémoire aussi efficace que celle qui est utilisée par le système la *full mediation* serait probablement plus rapide que la *médiation par traduction d'adresse* comme dans l'expérience précédente.

Ces expériences nous ont permis de montrer que lors de l'envoi de nombreux messages il vaut mieux privilégier la *full mediation*. Nous avons également pu voir que pour envoyer de gros messages, les deux médiations se valent. De plus, la faible durée d'exécution d'une expérience nous permet de considérer que l'*overhead* dû à la mise en place de Simterpose est acceptable. Ainsi, nous pouvons dire que notre implémentation du réseau de communications permet de faire de la virtualisation légère à ce niveau. Nous allons maintenant voir si il en est de même en ce qui concerne la gestion du temps.

7.2 Temps

Dans la section 5.2.3, nous avons pu voir qu'il est important de gérer le temps que l'application voit s'écouler. En section 6.2, nous avons présenté l'implémentation qui a été choisie pour résoudre cette problématique. Maintenant, nous souhaitons évaluer ses performances.

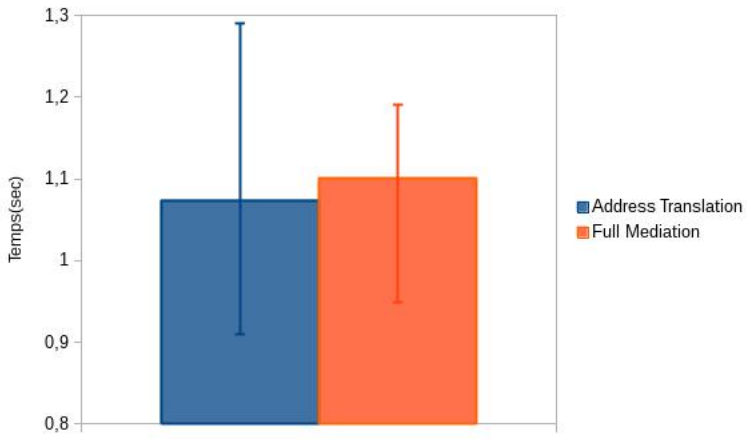


Figure 19 – Temps d'exécution lors de l'envoi d'un message de 1Mo avec et sans Simterpose.

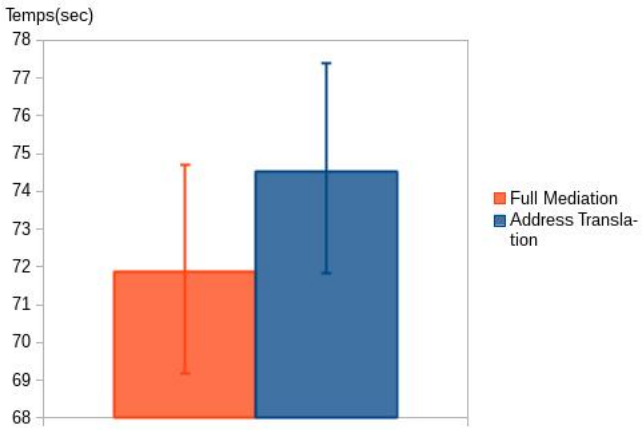


Figure 20 – Temps d'exécution de l'envoi d'un million de messages de 128o avec et sans Simterpose.

7.2.1 Protocole

Pour évaluer notre implémentation nous n'allons pas utiliser les mêmes expériences que pour le réseau de communications car le but ici est d'intercepter les fonctions temporelles.

L'objectif principal étant de montrer qu'il est possible de faire de la virtualisation légère, nous souhaitons mesurer le surcoût ajouté par cette interception lors de l'exécution de Simterpose. Si ce surcoût est acceptable, on pourra considérer qu'il est également possible de mettre en place une virtualisation légère qui gère cette fonctionalité. Nos expériences vont donc consister à comparer les temps d'exécution d'une application utilisant Simterpose avec et sans interception via `LD_PRELOAD` pour chaque type de médiation. Néanmoins, nous pensons que le changement de médiation ne devrait pas influer sur les performances car chaque médiation intervient uniquement lorsqu'on souhaite effectuer des appels systèmes réseaux que l'on fasse de l'interception avec `LD_PRELOAD` ou pas.

L'application qui a été créée pour effectuer nos expériences exécute différents appels à des fonctions temporelles (`ftime`, `time`, `gettimeofday`, `localtime`, `mktime`...). Le protocole utilisé pour exécuter l'application est le même que celui utilisé pour tester le réseau de communications de Simterpose.

7.2.2 Full mediation

Tout d'abord nous allons exécuter l'application qui effectue les appels temporels avec Simterpose en *full mediation* sans `LD_PRELOAD` et avec `LD_PRELOAD`. Les résultats de l'expérience sont présentés Figure 21. On constate que le temps d'exécution avec interception via `LD_PRELOAD` est plus ou moins constant (environ 1,03 secondes), alors que celui sans interception varie énormément, entre 0,82 et 1,13 secondes. Cela est dû au fait que lorsqu'on appelle des fonctions temporelles sans interception via `LD_PRELOAD` la bibliothèque `VDSO` est appelée pour gérer l'appel et accéde elle-même à la mémoire. Cet accès n'a pas un coût constant puisqu'il dépend de la charge du système qui varie en permanence même si on ne fait tourner que notre application et aucune autre en parallèle. Ainsi, le temps d'exécution peut varier comme c'est la cas ici alors qu'il est stable quand on utilise `LD_PRELOAD` puisqu'on empèche ces accès. Néanmoins, en moyenne les deux expériences ont environ le même temps d'exécution, 1 seconde pour la première et 1,02 secondes pour la deuxième. On peut donc considérer que l'interception via `LD_PRELOAD` a un surcoût négligeable lorsqu'on utilise Simterpose en *full mediation*.

7.2.3 Mediation par traduction d'adresse

Nous allons maintenant voir s'il en est de même lorsqu'on exécute l'application qui effectue les appels temporels avec Simterposeen *médiation par traduction d'adresse* avec et sans interception via `LD_PRELOAD`. Les résultats cette expérience sont présentés Figure 22. On constate ici aussi que le temps d'exécution avec interception via `LD_PRELOAD` est plus ou moins constant (environ 1,02 seconde), alors que celui sans interception varie beaucoup, entre 0,86 et 1,12 secondes. Cela est dû comme précédement à l'utilisation de la bibliothèque `VDSO` en l'absence d'interception via `LD_PRELOAD`. De plus, le temps d'exécution moyen des deux expériences est le même : 1,02 secondes. Dans ce cas, on peut dire que l'interceprtion via `LD_PRELOAD` en *médiation par traduction d'adresse* n'a aucun surcoût.

Ainsi, nous avons pu voir que le surcoût dû à l'interception des fonctions temporelles via `LD_PRELOAD` est inexistant en *médiation par traduction d'adresse* et qu'il est négligeable *full*

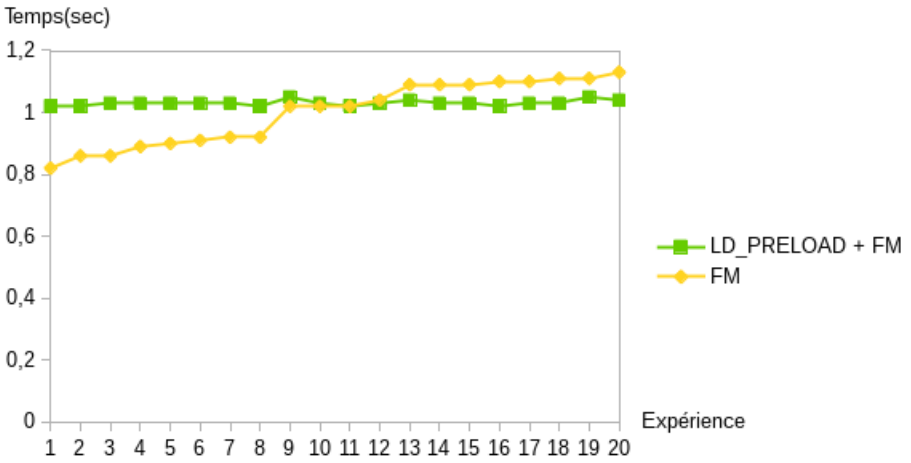


Figure 21 – Temps d'exécution d'une application temporelle en *full mediation* avec interception via `LD_PRELOAD` et sans interception

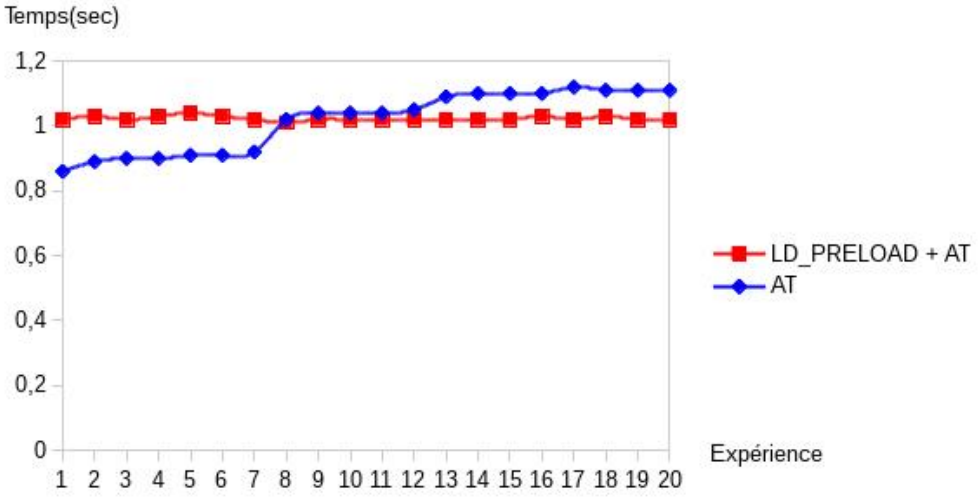


Figure 22 – Temps d'exécution d'une application temporelle en *full mediation* avec interception via `LD_PRELOAD` et sans interception

mediation (environ 2%). On peut donc considérer que nous avons réussi à mettre en place une virtualisation légère qui gère également l'écoulement du temps et qui plus est de façon particulièrement efficace. De plus, en comparant les deux graphiques présentés Figure 21 et 22, on voit bien que les courbes d'interception via `LD_PRELOAD` se superposent et qu'il en est de même pour celles sans interception quelque soit la médiation utilisée. Le changement de médiation n'influe donc par sur les performances d'interception avec `LD_PRELOAD`. Cela confirme l'hypothèse que nous avions fait.

Dans cette section nous avons analysé les performances des fonctionalités implémentées durant mon stage. Cela nous a permis de constater que malgré l'existence d'un surcoût au niveau du temps d'exécution, il est parfaitement possible de mettre en place une virtualisation légère utilisant le réseau de communications et la gestion du temps que nous avons mis en place dans Simterpose. Le tableau suivant résume les différents *overhead* de Simterpose afin d'avoir une vision plus globale.

	Réseau de communications		Temps
	Grosse données	Petites données	
Minimal	0.9s	64s	0s
Maximal	1.18s	74.5s	0.23s
Moyen	1.1s	70.1s	0.02s

Table 3 – *Overhead* du temps d'exécution d'applications avec Simterpose

8 Travaux futurs

TODO

9 Conclusion

L'objectif de ce stage était de montrer qu'il est possible de faire de la virtualisation légère afin de tester des applications distribuées large échelle quelconques. Pour cela, nous avons commencé par expliquer le concept de virtualisation légère et pouruqoi nous l'avions choisi. Puis nous avons présenté les outils permettant sa mise en place ainsi que les projets qui actuellement font de la virtualisation légère pour différents types d'applications. Les projets existants ne permettant pas de résoudre les quatre problèmes engendrés par cette virtualisation (gestion du temps, des threads, des communications réseaux et le DNS) et donc de tester n'importe quel type d'applications distribués, un nouveau projet a été lancé.

L'émulateur Simterpose dévelopé au LORIA permet d'exécuter et de tester des applications distribuées réelles, sans disposer de leur code source, dans un environnement virtuel. Il se base sur la plateforme de simulation SimGrid pour mettre en place l'environnement d'exécution dans lequel l'application pensera s'exécuter. Pour maintenir la virtualisation, les actions des applications sont interceptées et modifiées pour ensuite être exécutées. On utilise SimGrid pour calculer la réponse de l'environnement virtuel aux différentes actions. La solution proposée intercepte les actions à deux niveaux différents : appels systèmes et bibliothèques. Simterpose permet également d'injecter diverses fautes dans la simulation pour avoir une virtualisation plus réaliste.

Au début du stage, Simterpose gèrait plus ou moins bien les threads, le réseau de communications n'était implémenté qu'en parti et les deux autres fonctionnalités (temps et DNS) étaient inexistantes. Nous avons donc terminé l'implémentation du réseau de communications et avons mis en place la gestion du temps. Ces deux fonctionnalités ont nécessitées la création de nouveaux outils et des améliorations ont été apportées à Simterpose. Cependant, comme nous avons pu le voir notre émulateur n'est pas encore terminé. Néanmoins, nos expériences nous ont permis de montrer qu'il est déjà possible de mettre en place une virtualisation légère par interception pour tester des applications distribuées quelconques pour les deux fonctionnalités que nous avons implémentées.

Références

[1] The deter project. http://deter-project.org/.

[2] Dyninst. http://www.dyninst.org/.

[3] Cheating the elf, subversive dynamic linking to libraries. https://grugq.github.io/docs/subversiveld.pdf.

[4] Infection via got poisoning d'appel à des bibliothèques partagées. http://vxheaven.org/lib/vrn00.html.

[5] LD_PRELOAD. https://rafalcieslak.wordpress.com/2013/04/02/dynamic-linker-tricks-using-ld_preload-to-cheat-inject-features-and-investigate-program

[6] Rr project. http://rr-project.org/.

[7] Rr implémentation. http://rr-project.org/rr.html.

[8] Smpi project : Simulation d'applications mpi. http://www.loria.fr/~quinson/Research/SMPI/.

[9] cgroups. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[10] Coccinelle project. http://coccinelle.lip6.fr/.

[11] Cwrap website, . https://cwrap.org/.

[12] An article about cwrap and how it works, . https://lwn.net/Articles/594863/.

[13] Man iptables. http://ipset.netfilter.org/iptables.man.html.

[14] The netfilter.org project. http://www.netfilter.org/.

[15] Seccomp man. http://man7.org/linux/man-pages/man2/seccomp.2.html.

[16] Valgrind, 2000. http://valgrind.org/.

[17] Terry Benzel. The science of cyber security experimentation : the deter project. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 137–148. ACM, 2011.

[18] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. Grid'5000 : a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, 2006.

[19] Tomasz Buchert, Lucas Nussbaum, and Jens Gustedt. Methods for emulation of multi-core cpu performance. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 288–295. IEEE, 2011.

[20] Tomasz Buchert, Lucas Nussbaum, and Jens Gustedt. Methods for emulation of multi-core cpu performance. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 288–295. IEEE, 2011.

[21] Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. Emulation at very large scale with distem. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 933–936. IEEE, 2014.

[22] Louis-Claude Canon, Emmanuel Jeannot, et al. Wrekavoc : a tool for emulating heterogeneity. In *IPDPS*, 2006.

[23] Henri Casanova. Simgrid : A toolkit for the simulation of application scheduling. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 430–437. IEEE, 2001.

[24] P-N Clauss, Mark Stillwell, Stéphane Genaud, Frédéric Suter, Henri Casanova, and Martin Quinson. Single node on-line simulation of mpi applications with smpi. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 664–675. IEEE, 2011.

[25] Ted Faber, John Wroclawski, and Kevin Lahey. A deter federation architecture. In *DETER*, 2007.

[26] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. Experimental methodologies for large-scale systems : a survey. *Parallel Processing Letters*, 19(03) :399–418, 2009.

[27] Marion Guthmuller. Interception système pour la capture et le rejeu de traces, 2009–2010.

[28] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. Ptrace, utrace, uprobes : Lightweight, dynamic tracing of user apps. In *Proceedings of the 2007 Linux Symposium*, pages 215–224, 2007.

[29] Jereme Lamps, David M Nicol, and Matthew Caesar. Timekeeper : a lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation*, pages 179–186. ACM, 2014.

[30] Hee Won Lee, David Thuente, and Mihail L Sichitiu. Integrated simulation and emulation using adaptive time dilation. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation*, pages 167–178. ACM, 2014.

[31] Chloé MACUR. Rapport de stage : Émulation d'applications distribuées sur des plates-formes virtuelles simulées, 2014.

[32] M Mahalingam, D Dutt, K Duda, P Agarwal, L Kreeger, T Sridhar, M Bursell, and C Wright. Virtual extensible local area network (vxlan) : A framework for overlaying virtualized layer 2 networks over layer 3 networks. *Internet Req. Comments*, 2014.

[33] Steven McCanne and Van Jacobson. The bsd packet filter : A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2. USENIX Association, 1993.

[34] Jelena Mirkovic, Terry V Benzel, Ted Faber, Robert Braden, John T Wroclawski, and Stephen Schwab. The deter project. 2010.

[35] Nicholas Nethercote and Julian Seward. Valgrind : A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi : 10.1145/1250734.1250746. URL http://doi.acm.org/10.1145/1250734.1250746.

[36] Martin Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experiments. In *9th International conference on Peer-to-peer computing - IEEE P2P 2009*, Seattle, United States, September 2009. IEEE. URL https://hal.inria.fr/inria-00435802.

[37] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. Design and Evaluation of a Virtual Experimental Environment for Distributed Systems. In *PDP2013 - 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pages 172 – 179, Belfast, United Kingdom, February 2013. IEEE. doi : 10.1109/PDP.2013.32. URL https://hal.inria.fr/hal-00724308.

[38] Guillaume Serrière. Simulation of distributed application with usage of syscalls interception, 2012.

[39] Hyo Jung Song, Xianan Liu, Dennis Jakobsen, Ranjita Bhagwan, Xingbin Zhang, Kenjiro Taura, and Andrew Chien. The microgrid : a scientific tool for modeling computational grids. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 53–53. IEEE, 2000.

[40] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI) :255–270, December 2002. ISSN 0163-5980. doi : 10.1145/844128.844152. URL http://doi.acm.org/10.1145/844128.844152.

[41] Huaxia Xia, Holly Dail, Henri Casanova, and Andrew Chien. The microgrid : Using emulation to predict application performance in diverse grid network environments. In *Proc. the Workshop on Challenges of Large Applications in Distributed Environments (CLADE'04)*. Citeseer, 2004.