

Titre long

Étudiante : **Louisa Bessad**,

Encadrant : **Martin Quinson**

Table des matières

1	Introduction	4
2	Méthodes possibles pour la virtualisation légère	6
2.1	Virtualisation standard	6
2.2	Émulation par interception	6
2.2.1	Action sur le fichier source	8
2.2.2	Action sur le binaire	8
2.2.3	Médiation des Appels Système	9
2.2.4	Médiation directe des appels de fonctions	13
3	État de l'art	16
3.1	CWRAP	16
3.2	RR	16
3.3	Distem	16
3.4	MicroGrid	18
3.5	DETER	18
3.6	ROBOT	18
4	Simterpose : la médiation	19
4.1	Organisation générale	20
4.2	Les communications réseaux	20
4.3	Les thread	22
4.4	Le temps	22
4.5	DNS	22
5	Conclusion	23

Résumé

TODO

1 Introduction

Dans le cadre de ce stage, nous allons nous intéresser aux applications distribuées. Autrement dit aux applications dont une partie ou la totalité des ressources n'est pas stockée 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 disponibilité des données en se les échangeant, comme les applications Torrent (BitTorrent, Torrent...). Grâce au projet BOINC¹ par exemple, on peut partager la puissance de calcul inutilisée de sa machine. 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 plate-formes et des ressources utilisées. Il devient donc de plus en plus difficile de créer de telles applications 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 de tester le comportement d'applications distribuées [11] ; 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 temps-réel. Cela permet de la tester sur un grand nombre d'environnement. L'outil créé et développé en partie en France pour nous permettre de faire cela est **Grid'5000**²[7], un autre outil développé à l'échelle mondiale est **PlanetLab**³. Néanmoins pour mettre en œuvre ces solutions complexes, il faut disposer des infrastructures nécessaires pour effectuer les tests. Il faut également écrire une application capable de gérer toutes ces ressources disponibles. Enfin, du fait du partage des différentes plate-formes 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 sur un simulateur, on doit d'abord représenter de façon théorique 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 mettre en œuvre, et permet de prédire l'évolution du système étudié grâce à l'utilisation de modèles mathématiques. De nos jours, les simulateurs tel que **SIMGRID**[10, 16] peuvent simuler des

1. <https://boinc.berkeley.edu/>

2. Infrastructure de 8000 cœurs répartis dans la France entière créée en 2005.

<https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home>

3. Créée en 2002, cette infrastructure de test compte aujourd'hui 1340 noeuds.

<http://www.planet-lab.org>

applications distribuées mettant à contribution des milliers de noeuds. Néanmoins, avec la simulation on ne 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 et utilise un simulateur pour virtualiser l'environnement d'exécution. Cette solution représente un intermédiaire entre la simulation et l'exécution sur plate-forme réelle visant à résoudre les limitations de ces deux solutions. En effet, les actions de l'application sont réellement exécutées sur la machine hôte, autrement dit la machine réelle sur laquelle s'exécute l'émulation. Et grâce au simulateur 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. Dans notre cas l'émulation peut-être faite *off-line* ; on sauvegarde les actions de l'application sur disque et on les rejoue plus tard dans le simulateur ou *on-line* ; on bloque l'application le temps que les actions soient reportées dans le simulateur pour qu'il calcule le temps de réponse de la plate-forme simulée.

Actuellement, il existe deux types d'émulation pour les applications distribuées ; la virtualisation standard et la légère. On parle de virtualisation légère quand on souhaite tester des applications sur une centaine d'instances. Dans ce rapport nous allons présenter en section 2 les méthodes utilisées pour faire de la virtualisation légère : limitation et interception. Puis en section 3 nous présenterons les projets permettant de faire de l'émulation pour tester des applications dans un environnement distribué. Pour finir, nous expliquerons en section 4, pourquoi dans le cadre du projet Simterpose c'est la virtualisation légère par interception qui a été choisie et comment elle fonctionne.

2 Méthodes possibles pour la virtualisation légère

Il existe actuellement deux méthodes permettant de faire de la virtualisation légère. La première est une émulation par limitation ou dégradation également appelée virtualisation standard et la seconde est une émulation par interception.

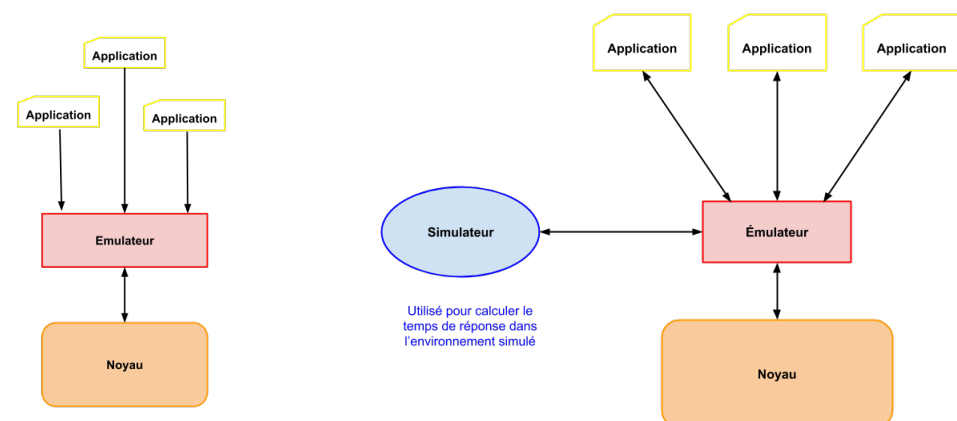


FIGURE 1 – Virtualisation par limitation (à gauche) et par interception (à droite)

2.1 Virtualisation standard

Avec cette première méthode, illustrée Fig.1, 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 peut 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 distribuées 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 **cgroup**, **netstat** et **cpuburner**. 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. De plus *à écrire deux derniers points négatifs à éclaircir.*

2.2 Émulation par interception

Dans le cas de l'émulation par interception, illustrée Fig.1, 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

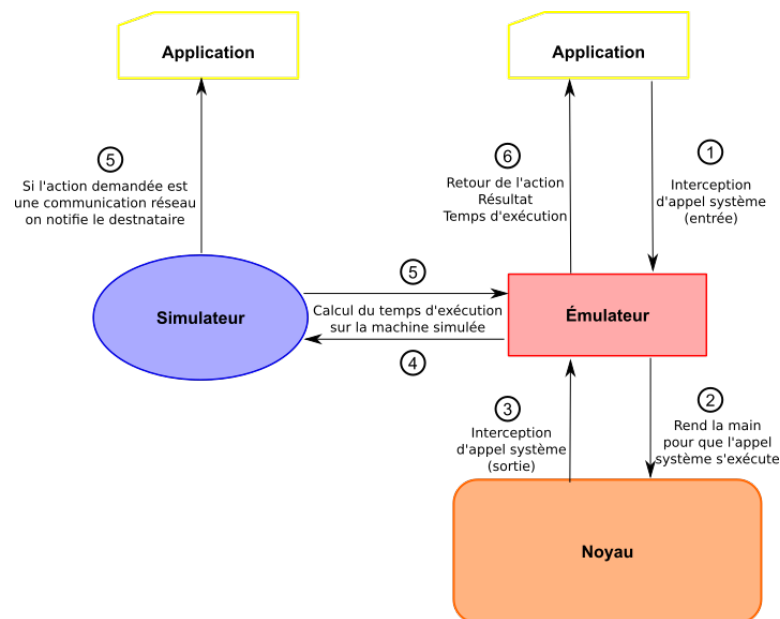


FIGURE 2 – Fonctionnement de l'émulation par interception

émulateur qui va attraper toutes les communications de l'application avec l'hôte et qui les transmettra ensuite au simulateur.

Une application distribuée peut vouloir communiquer avec l'hôte soit pour effectuer de simples calculs **TODO**(SEB), soit pour effectuer des requêtes de connexion ou de communication 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 cette dernière pour qu'elle puisse s'exécuter sur la machine hôte. Quand cette dernière renvoie une réponse à l'application, elle est également interceptée par l'émulateur pour que l'application ne voit pas le changement d'architecture. En même temps, il envoie au simulateur le temps d'exécution de l'action sur la machine hôte pour qu'il puisse calculer ce temps sur la machine simulée, en faisant un rapport entre les performances des deux machines. Les délais calculés par le simulateur sont soit des temps de calculs soit des temps de connexion ou de communication. Lorsque le simulateur a terminé le calcul du temps de réponse, il le transmet à l'émulateur qui l'envoie 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 fournit qui va influencer l'horloge de l'application. Finalement, les applications ne communiquent plus directement entre elles.

Pour intercepter ces actions, il faut d'abord choisir à quel niveau se placer. En effet, une application peut communiquer avec le noyau via différentes abstractions. 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.

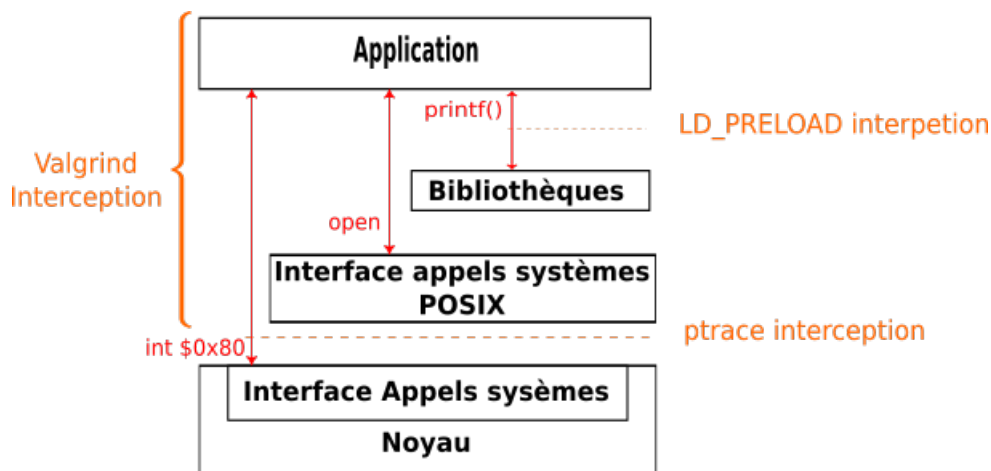


FIGURE 3 – Communications possibles entre le noyau et une application

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

2.2.1 Action sur le fichier source

2.2.2 Action sur le binaire

Pour agir sur le binaire d'une application, c'est l'outil d'instrumentation d'analyse dynamique Valgrind [6, 15] que nous allons étudier. À l'origine, il est utilisé pour le déboguage 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 profilage⁴. Valgrind fonctionne à la manière d'une machine virtuelle faisant de la compilation à volée⁵. 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

4. Méthode visant à analyser le code d'une application pour connaître la liste des fonctions appelées et le temps passé dans chacune d'elles

5. 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

dans l'environnement simulé nécessaire à l'émulateur. On pourrait 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 Valgrind 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 [12] : 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.

2.2.3 Médiation des Appels Système

En regardant la Fig.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 est d'intercepter directement les appels systèmes. Ces derniers sont constitués de deux parties ; la première, 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 seconde, 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. Nous permettant ainsi de récupérer et modifier les informations permettant de maintenir l'environnement simulé avant de lui rendre la main, pour pouvoir entrer ou sortir de l'appel système.*

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

L'appel système `ptrace` [12, 13] , dont la Fig.4 illustre le fonctionnement, permet de tracer tous les événements 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 requêtes en paramètre de l'appel système.

Pour pouvoir contrôler un processus via `ptrace`, on va créer deux processus parents via un `fork()` ; un processus appelé "processus espionné" qui exécutera l'application et 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 `PTTRACE_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 `PTTRACE_SYSCALL`). Ainsi, le processus espion sera donc appelé deux fois : à l'entrée et à la sortie de l'appel système.

Quand un des 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 fera alors les modifications nécessaires dans les registres du processus espionné pour conserver la virtualisation de l'environnement. Pour cela il pourra utiliser les requêtes `PEEK_DATA` et `POKE_DATA` passées en argument de l'appel système (à éclaircir) ou modifier directement le contenu du `/proc/id/mem`. Puis, 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 : `PTTRACE_KILL` qui termine le processus espionné ou `PTTRACE_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 cycle CPU. *De plus, il supporte mal les processus utilisant du multithreading, et ne fait pas parti 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.*

Uprobes [12, 13]

pour *user-space probes*, quant à lui est une API noyau permettant d'insérer dynamiquement des points d'arrêts à n'importe quel endroit dans le code d'une application, dans notre cas les appels systèmes, et à n'importe quel moment de son exécution.

Il existe deux versions de Uprobes la première est basée sur les "trace hook"⁶ citation. Cette solution ne sera pas développée ici car elle est très peu utilisée à vérifier.

La seconde, la plus connue, 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 est capable de gérer le multithreading. Dans cette version, l'utilisateur fournit pour chaque point d'arrêt 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 contrôlé), *ii)* `uprobe_task` (autant que le processus contrôlé a de thread), *iii)* `uprobe_kimg` (une pour chaque point d'arrêt

6. http://fr.wikipedia.org/wiki/Hook_%28informatique%29

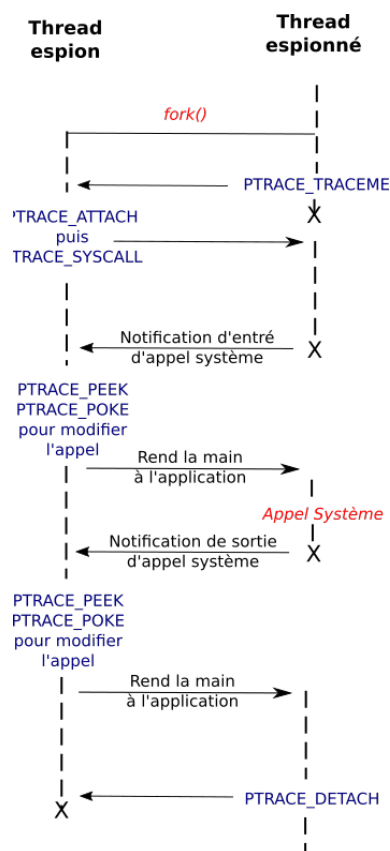


FIGURE 4 – Attachement d'un processus et contrôle via un espion

affectant un processus). 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 enlèvera. 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 placer 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 bon handler. Pour savoir qu'un point d'arrêt a été touché, 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 crée 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 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.

Seccomp/BPF : Seccomp sec [5] est un appel système qui permet d'isoler un processus en lui donnant le droit d'appeler et d'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 très limité. Pour plus de flexibilité, on peut utiliser une extension de cet appel système appelée seccomp/BPF, pour *seccomp Berkeley Packet Filter*, permettant de définir dans un programme BPF bpf [2] les appels systèmes autorisés à s'exécuter, en plus de ceux cités précédemment. Cette dernière 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.

Néanmoins, l'appel système `seccomp` et son extension `seccomp/BPF` ne sont disponibles que si le noyau est configuré avec l'option `CONFIG_SECCOMP` pour la première et `CONFIG_SECCOMP_FILTER` pour la deuxième. Pour pouvoir créer des filtres, il faut également avoir des droits particuliers, notamment l'exécution de certaines commandes root. Ainsi, l'utilisation de cet appel système et de son extension demande une certaine configuration noyau et des privilèges pour les utilisateurs, ce qui n'est pas très conseillé.

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 significativement le nombre d'événement sur lequel attendra le processus espion.

Malgré ses défauts, l'appel système `ptrace` semble être le meilleur outil pour faire ce type d'interception. Néanmoins, il a été montré dans un précédent stage [12] qu'il est inefficace voire inutile en ce qui concerne tous les appels systèmes temporels qu'une application souhaiterait exécuter (`time()`, `clock_gettime()`, `gettimeofday()`) 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 dus aux deux changements de contexte effectués lors de l'exécution d'un appel système. VDSO va retrouver l'heure *dans le contexte 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 (plus de changements de contexte) et oblige l'utilisateur à modifier les paramètres de son noyau. On peut donc dire que cette solution d'interception n'est donc pas complète.

2.2.4 Médiation directe des appels de fonctions

Puisque l'interception des actions d'une application au plus bas niveau ne suffit pas, on peut penser qu'une bonne 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.

LD_PRELOAD : L'utilisation de la variable d'environnement `LD_PRELOAD` [1], 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 apparaissant en premier dans la liste des bibliothèques préchargées. Ce ne sera pas *nécessairement* celle de la bibliothèque attendue par le programme. Par exemple, on crée une bibliothèque partagée qui implémente une fonction `open()` de même prototype que la fonction `open()` de la `libc` et on place cette bibliothèque dans la variable `LD_PRELOAD`. Quand on exécute un programme faisant un appel à `open()`, 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 `open()` 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 susceptible d'être utilisée par l'application, 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* (Fig. 3) notre mécanisme d'interception est contourné. *En effet on ne peut surcharger que des fonctions définies dans bibliothèques avec cette solution, pas les appels systèmes directement.* De même, si on oublie de réécrire une fonction d'une des bibliothèques utilisée par l'application. Cette solution n'est donc pas suffisante pour le modèle d'interception que nous souhaitons avoir.

Cependant, on peut voir que `LD_PRELOAD` résout les lacunes de `ptrace` concernant les fonctions de temps et le multithreading. À l'inverse, puisque `ptrace` permet d'intercepter les appels systèmes que le modèle d'interception avec `LD_PRELOAD` ne permet pas de gérer, on peut dire que `ptrace` résout les problèmes de `LD_PRELOAD`. Une solution choisie lors d'un précédent stage est donc d'allier les deux. On surchargera les fonctions temporelles dans notre bibliothèque préchargée avec `LD_PRELOAD` pour pallier les lacunes temporelles de `ptrace`. *Et ce dernier s'occupera de toutes les autres fonctions*, ainsi on est certain de n'oublier aucune fonction.

	Source to Source	Réimplémentation SMPI	Coccinelle	Valgrind
Fichier intercepté	Source	Source	Source	Binaire
Coût	?	?	?	Important
Utilisation	?	?	?	Complexe

TABLE 1 – Comparaison des différentes solutions d'interception au niveau de l'application

	ptrace	Uprobes	seccomp/BPF	LD_PRELOAD	Got Injection
Niveau d'interception	Appel Système	Appel Système	Appel Système	Bibliothèque	Bibliothèque
Coût	Moyen	Faible(?)	?	Faible	?
Utilisation	Assez complexe	?	?	Simple	?

TABLE 2 – Comparaison des différentes solutions d'interception entre l'application et le noyau

Got injection :

3 État de l'art

3.1 CWRAP

cwrap[3, 4] a pour but de tester des applications réseaux s'exécutant sur des machines UNIX ayant un accès réseau limité et sans droits root. Ce projet libre a débuté en 2005 avec le test du framework "smbtorture" de Samba⁷ 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 2.2.4.

La première `socket_wrapper` gère les communications réseaux. Elle modifie toutes les fonctions liées aux sockets afin que toutes les communication 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 droit root dans le réseau local émulé pour communiquer. Cette bibliothèque permet aussi de faire des captures de trace réseau. La seconde `nss_wrapper` est utilisée dans le cas d'application 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` 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 root. 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` 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 complet d'émulation permettant de tester des applications utilisant des réseaux complexes. Le seul bémol étant qu'on utilise uniquement LD_PRELOAD pour cette émulation, il ne faut donc pas oublier une seule fonction.

3.2 RR

3.3 Distem

Distem [17] 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

7. <https://www.samba.org/>
https://wiki.samba.org/index.php/Writing_Torture_Tests

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 la *???c quoi le mot que je cherche !!!* 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 [9]. Le prochain objectif étant de réussir à émuler 100 000 machines.

Pour construire un environnement distribué virtuel Distem utilise 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 c'est ce que montre la Fig.5, on va allouer à chaque nœud virtuel un certain nombre de cœurs du processeur de la machine physique dont on pourra contrôler la fréquence individuellement.

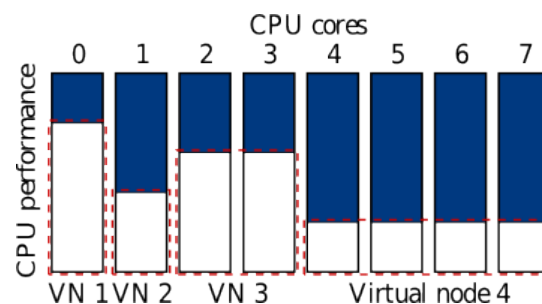


FIGURE 5 – Répartition des cœurs d'un processeur d'une machine hôte entre les différents nœuds virtuels qu'elle héberge et émulation de leur puissance en utilisant qu'une partie de leur puissance.

On finit par construire 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 est capable de changer à la volée les paramètres du réseau et la vitesse de chaque cœur alloué à un nœud virtuel.

Comme le présente la Fig.6, Distem repose sur une architecture simple pour construire un 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 possédant son démon Distem qui contrôle les Vnodes. Les Vnodes sont séparés et n'ont pas conscience 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 système (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 le *Linux Control Group* *todo définir*. Pour contrôler la puissance des cœurs attribués à chaque Vnode, Distem utilise l'algorithme *CPU-Hog* *todo définir* [8]. Ainsi les Vnodes ont conscience les uns

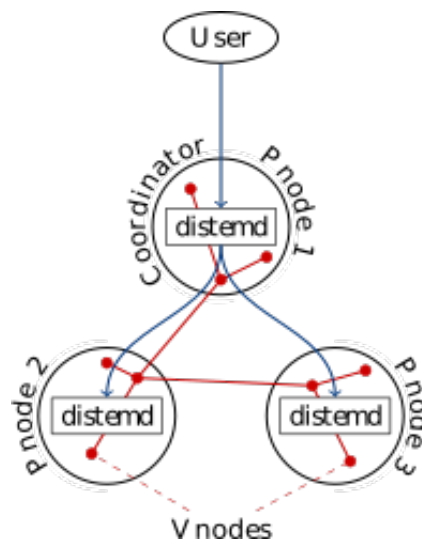


FIGURE 6 – Architecture de communication de Distem. On a 3 nœuds physique (“Pnodes”) contenant chacun 3 nœuds virtuels (“Vnodes”)

des autres uniquement via le réseau virtuel. Chaque Vnodes possède une ou plusieurs interfaces réseau virtuelles reliées au réseau physique de l’hôte afin que les Vnodes puissent 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 ne peut s’appliquer indéfiniment. Une autre solution, qui est celle utilisée actuellement, est de rajouter une couche d’abstraction réseau à l’intérieur du Pnode en utilisant VXLAN[9, 14] comme le montre la Fig.7. Ainsi, les paquets seront échangés entre Pnodes sur le réseau et c’est la couche VXLAN qui s’occupera d’envoyer au bon Vnodes le paquet reçu sur le Pnodes. Ces derniers étant très peu nombreux on est sûrs de ne pas surcharger les tables ARP.

On peut donc voir que Distem possède une infrastructure et un réseau émulsés 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 de limiter l’émulation empêchant ainsi l’émulation de machines plus rapides.

3.4 MicroGrid

3.5 DETER

3.6 ROBOT

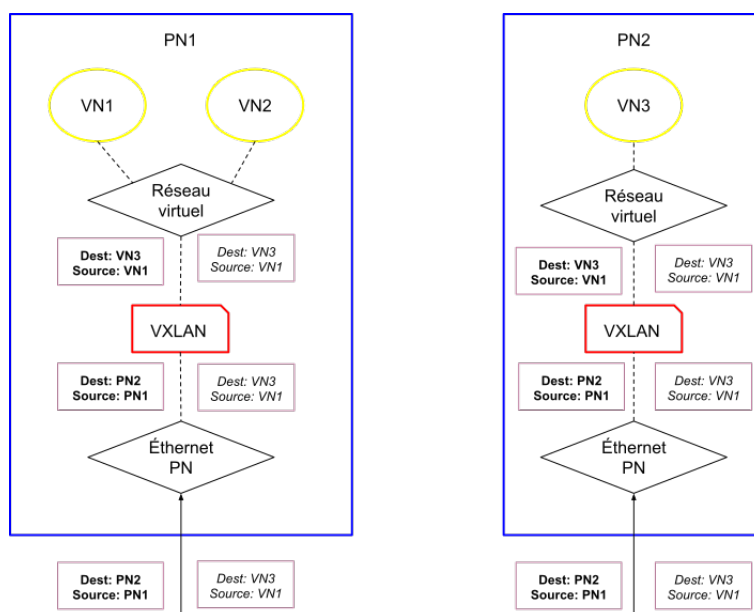


FIGURE 7 – Abstractions des communications réseaux de Distem via VXLAN. Les paquets en gras sont ceux envoyés en présence de VXLAN et ceux en italiques sont ceux qui seraient envoyés sur un réseau n'utilisant pas VXLAN

4 Simterpose : la médiation

Dans le cadre du projet Simterpose de virtualisation légère et de test d'applications distribuées, c'est l'émulation par interception qui a été choisi. En effet, le but final étant de pouvoir évaluer n'importe quelle application distribuée sur n'importe quel type d'architecture, on pourrait se retrouver à devoir émuler des machines plus puissantes que l'hôte, ce que l'émulation par dégradation ne permet pas. Pour cela on va utiliser SIMGRID comme simulateur et Simterpose comme émulateur, cette architecture est représentée Fig.8. Simterpose nous permettra donc d'utiliser le simulateur avec des applications réelles tout en leur faisant croire qu'elles s'exécutent sur des machines distinctes. Simterpose étant l'émulateur qui va nous permettre d'intercepter les communications de l'application avec la machine sur laquelle elle s'exécute et de faire de la médiation, nous allons étudier son fonctionnement et voir quels outils présentés en section 2 ont été choisis.

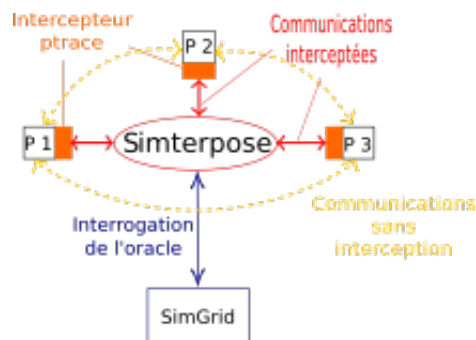


FIGURE 8 – **TODO**

4.1 Organisation générale

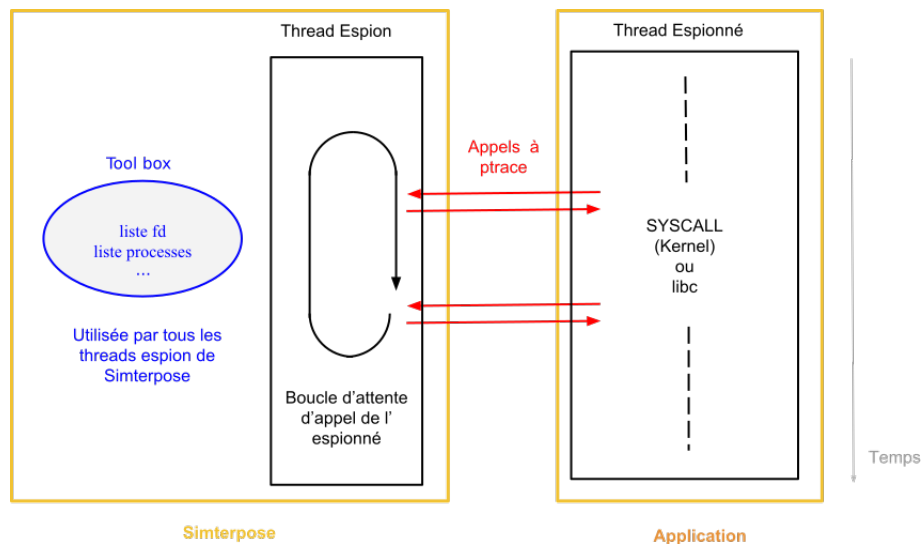


FIGURE 9 – **TODO**

TODO

4.2 Les communications réseaux

Lorsque `ptrace` est appelé en entrée ou sortie d'appel système, les modifications à apporter ne sont pas forcément les mêmes dans le cas d'une action nécessitant l'utilisation du réseau ou non. Dans le cas d'un calcul, il faut simplement maintenir une vision du temps tel qu'il s'écoule sur la machine simulée pour l'application. Ainsi en entrée d'appel système on n'a pas besoin de modifier quoique ce soit, par contre au retour il faut modifier le temps d'exécution du calcul en le remplaçant par celui calculé par le simulateur.

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. En effet l'application possède une adresse IP et des numéros de ports virtuels qui ne

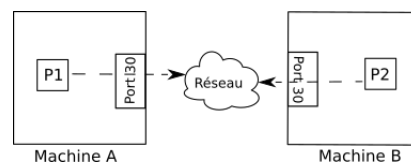
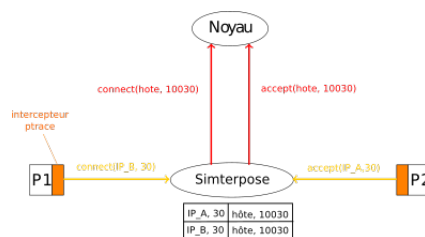


FIGURE 10 – **TODO**

correspondent pas forcément à ceux attribués dans le réseau local. 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. Pour gérer toutes ces modifications deux solutions ont été proposées lors d'un précédent stage [18] : la *médiation par traduction d'adresse* et la *full médiation*.



(a) Médiation par traduction d'adresse



(b) Full médiation

FIGURE 11 – Les différents types de médiation

Traduction d'adresse Avec ce type de médiation, illustrée Fig.11a, on considère que le noyau gère 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 $\langle IP, ports \text{ virtuels} \rangle$, on a un couple $\langle IP, ports \text{ réels} \rangle$ 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 pour que l'application pense toujours être *dans un environnement distribué*. 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'adresse et de ports et les applications peuvent conserver les adresses et les ports simulées 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 Fig.11b. *Ensuite, ce processus en utilisant **ptrace** 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.* Même si la *full médiation* permet d'éviter les communications réseaux et de conserver des tables de correspondances, elle s'avère moins efficace dans le cas d'applications qui communiquent énormément et utilisent de grosses données. En effet, les appels à la mémoire sont bien plus coûteux que les communications réseau.

4.3 Les thread

4.4 Le temps

4.5 DNS

5 Conclusion

Références

- [1] LD_PRELOAD. https://rafalcieslak.wordpress.com/2013/04/02/dynamic-linker-tricks-using-ld_preload-to-cheat-inject-features-and-investigate-programs/.
- [2] Berkeley packet filter. http://en.wikipedia.org/wiki/Berkeley_Packet_Filter.
- [3] Cwrap website, . <https://cwrap.org/>.
- [4] An article about cwrap and how it works, . <https://lwn.net/Articles/594863/>.
- [5] Seccomp man. <http://man7.org/linux/man-pages/man2/seccomp.2.html>.
- [6] Valgrind, 2000. <http://valgrind.org/>.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. Experimental methodologies for large-scale systems : a survey. *Parallel Processing Letters*, 19(03) :399–418, 2009.
- [12] Marion Guthmuller. Interception système pour la capture et le jeu de traces, 2009–2010.
- [13] 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.
- [14] 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.

- [15] 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>.
- [16] 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>.
- [17] 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>.
- [18] Guillaume Serrière. Simulation of distributed application with usage of syscalls interception, 2012.