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

Étudiante : **Louisa Bessad,**

Encadrant : **Martin Quinson**

Table des matières

Table des figures

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. 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 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** [2][18], 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 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 sur un simulateur, 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 tel que **SIMGRID**[23, 38] peuvent simuler des 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 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

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

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 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 plate-forme virtualisée soit calculé).

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 Approches 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.

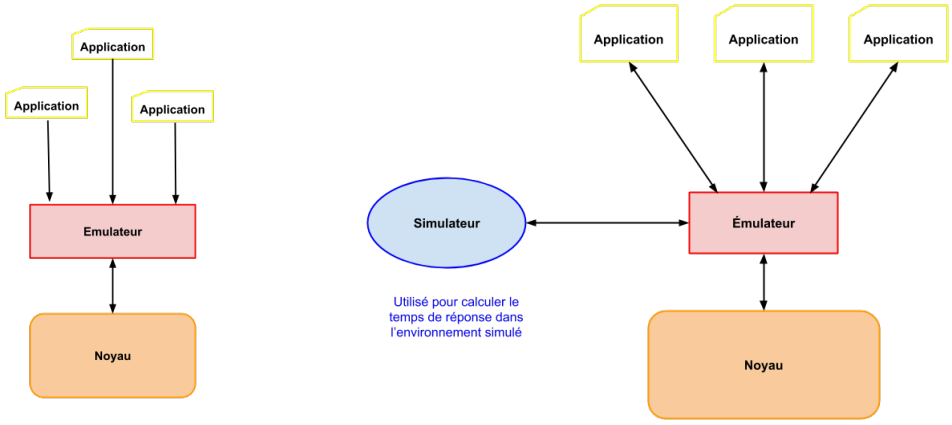


Figure 1 – Virtualisation par limitation (à gauche) et par interception (à droite)

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 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 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 **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.

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 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 , 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

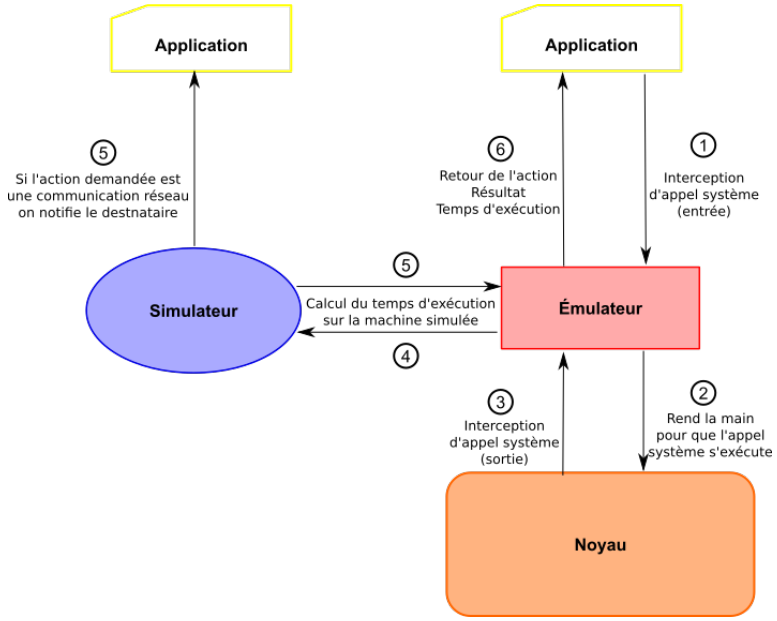


Figure 2 – Fonctionnement de l'émulation par interception

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 fourni 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.

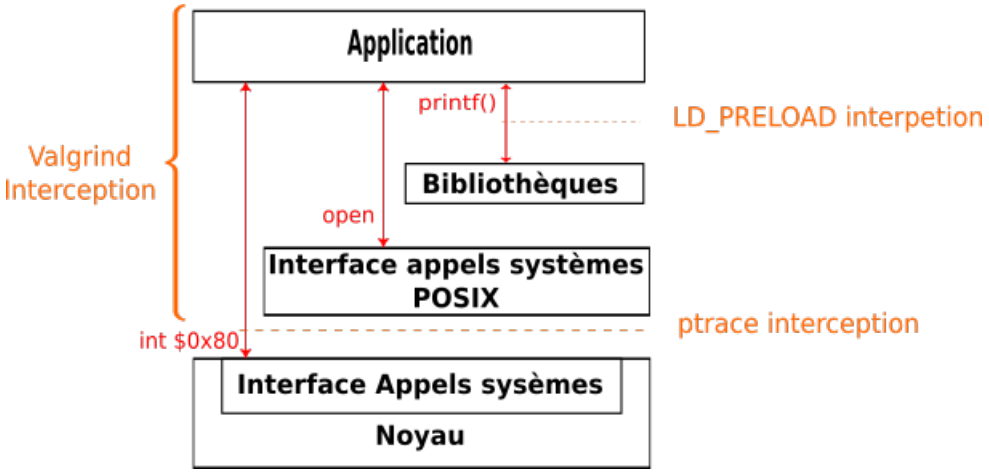


Figure 3 – Communications possibles entre le noyau et une application

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 appelerons médiation l'ensemble des modifications effectuées par l'émulateur sur les actions interceptées.

2.2.1 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 pourrait 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 transormer automatiquement des parties spécifiques d'un code source C. Pour cela, Coccinelle fournit le langage SmPL [4] 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 satisfont 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 le programme SMPI [5] [8, 24] qui permet de simuler des applications MPI. Si dans son implémentation on modifie la façon de gérer les communications on pourrait mettre en place et maintenir notre environnement virtuel. Pour cela, il devra modifier chaque communication reçue qui pourrait mettre en péril le maintien de notre virtualisation.

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 [16, 37] que nous allons étudier. À l'origine, il est utilisé pour le débuguage mémoire,

4. Semantic Patch Language
5. Simluation d'appplications MPI

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 [6]. Valgrind fonctionne à la manière d'une machine virtuelle faisant de la compilation à volée [7]. 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 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 intecepter 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.

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.

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

7. 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

L'appel système ptrace [27, 28] , dont la Fig.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 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 `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 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 : `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 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 [27, 28]

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 [8]"**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.

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

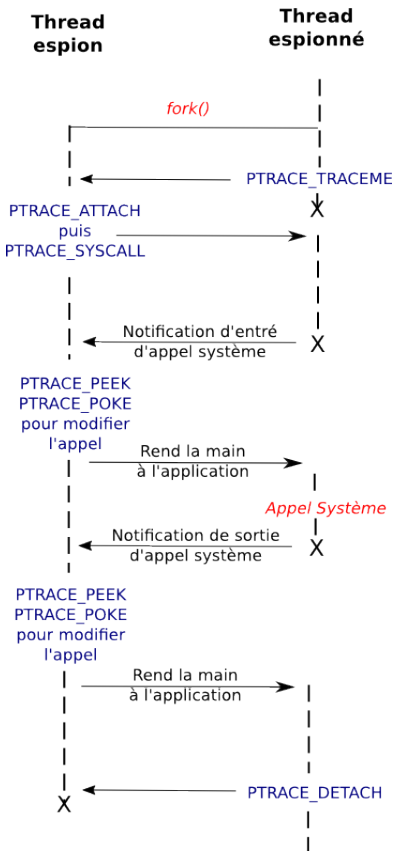


Figure 4 – Attachement d'un processus et contrôle via un espion

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 thread), *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 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 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 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 [15] 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 BSD Packet Filter*, permettant de définir dans un programme BPF [35] 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 signifiquativement 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.

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` [4], 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 bibliothèqes 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 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 chargées dynamiquement avec cette solution, et non 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.

GOT Poisoning : À la compilation, les adresses des appels de fonctions appartenant à des bibliothèques partagées ne sont pas connues. On associe alors un symbôle à chaque appel d'une de ces fonctions. C'est lors de l'exécution du programme que l'éditeur de lien dynamique résoudra le symbôle en trouvant l'adresse de la fonction à laquelle il correspond. Cette adresse sera ensuite stockée dans la "Global Offset Table"[2], 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" [3] 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 symbôle correspondant à la fonction par l'adresse d'une nouvelle fonction que l'on aura implémenté. Comme avec `LD_PRELOAD` il ne faut pas oublier de fonctions sinon l'interception sera contournée.

Cette solution étant très complexe elle ne sera pas développée en détail dans ce rapport.

Dans cette section, nous avons présenté différentes approches permettant de faire de l'interception et de la médiation d'actions d'applications, résumé dans la table 1. Dans le cas d'émulateur ne souhaitant pas modifier le code source d'une application les outils présentés en 2.2.1 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.

	`ptrace`	Uprobes	seccomp/BPF	`LD_PRELOAD`	got Poisoning	Valgrind
Niveau d'interception	Appel Système	Appel Système	Appel Système	Bibliothèque	Bibliothèque	Binaire
Coût	Moyen	Faible(?)	?	Faible	?	Important
Utilisation	Assez complèxe	?	?	Simple	?	Complèxe

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

3 Outils de virtualisaiton légère

Maintenant que nous avons étudié différents outils permettant de gérer différents aspect de la virtualisation légère, nous allons présenter différents projets. Ils sont tous basés sur des architectures différentes et utilisent certaines approches présentées dans la section précédente.

3.1 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 root. Ce projet libre a débuté en 2005 avec le test du framework "smbtorture" de Samba [9] 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 permetde 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 libraire `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

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

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

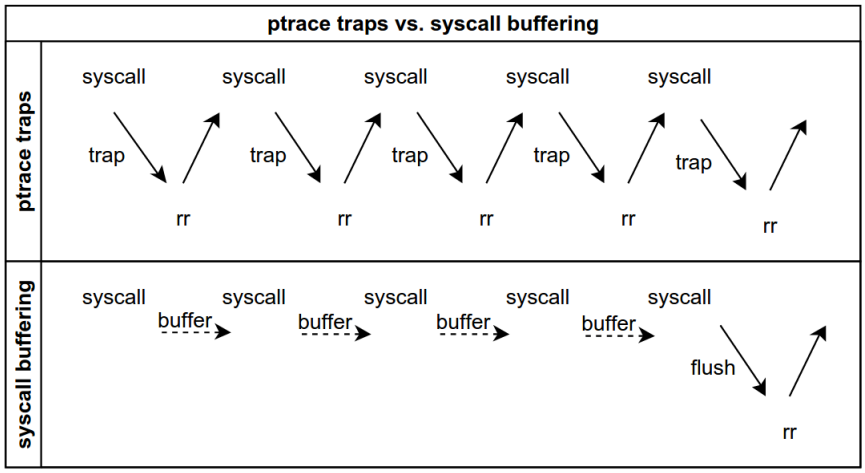


Figure 5 – Comparaison de l'exécution du rejeu de RR avec ptrace et seccomp-bpf pour gérer l'exécution des appels systèmes

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 puis à la déboguer de façon déterministe en rejouant l'enregistrement aussi souvent qu'on le souhaite. On relance toujours la même exécution et que 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**footnote ?** 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 pahse 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 [10] 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, valeur 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. Pour intercepter les appels systèmes RR utilise LD_PRELOAD (section2.2.4) qui va les placer dans un buffer. Ensuite Seccomp/BPF(section2.2.3) parcourra le buffer pour filtrer les appels systèmes 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, Fig5. Il sera utilisé pour renvoyer le bon résultat à l'application (celui sauvegardé lors de la première phase) et gérer les autres événements

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

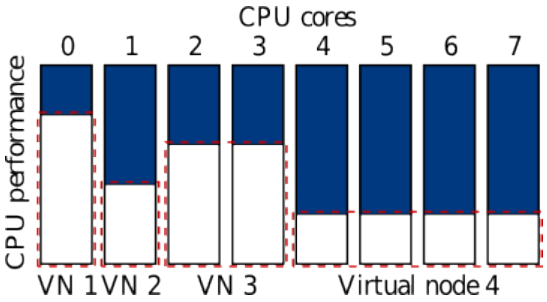


Figure 6 – Répartition des cœurs d'un processeur d'une machine hôte entre les différents noeuds virtuels qu'elle héberge et émulation de leur puissance en utilisant qu'une partie de leur puissance.

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, cotinuer 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 et d'en résoudre de nouveaux plus facilement. 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 systèmes 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 systèmes exécutés par les processus.

3.3 Distem

Distem [39] 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 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 [21]. 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.6, on va allouer à chaque nœud virtuel un certain nombre de cœurs du processeur de la machine physique dont on pourra controller la fréquence individuellement.

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

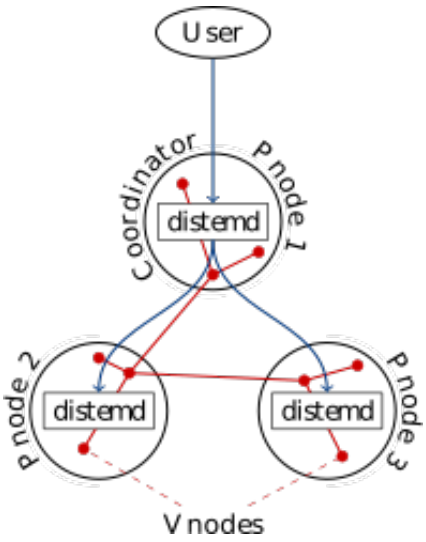


Figure 7 – Architecture de communication de Distem. On a 3 nœuds physique ("Pnodes") contenant chacun 3 nœuds virtuels ("Vnodes")

nœud virtuel.

Comme le présente la Fig.7, 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 possèdant 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 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 le *Linux Control Group*todo définir. Pour contrôler la puissance des cœurs attribués à chaque Vnode, Distem utilise l'algorithme *CPU-Hog*stodo définir [19]. Ainsi les Vnodes ont conscience les uns 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[21, 34] comme le montre la Fig.8. 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

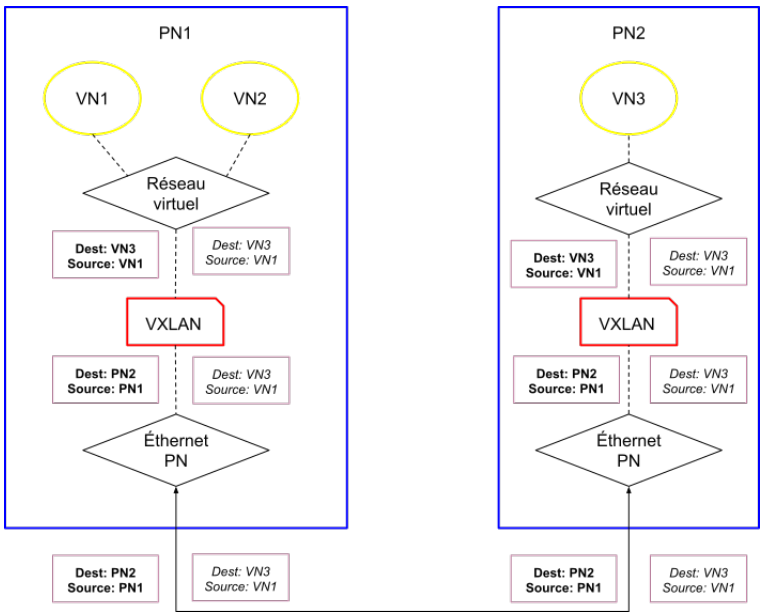


Figure 8 – Abstractions des communications réseaux de Distem via VXLAN. Les paquets en gras sont ceux envoyés en présence de VXLAN et ceux en italiques sont ceux qui seraient envoyés sur un réseau n'utilisant pas VXLAN

de ne pas surcharger les tables ARP.

On peut donc voir que Distem possède une infrastructure et un réseau émulés 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

Les grilles sont des environnements très hétérogènes, que se soit en terme de configuration, de performance ou de fiabilité. Les logiciels qui les utilisent doivent donc avoir une ceraine flexibilité pour pouvoir s'adapter aux différents environnements et ressources disponibles. Il esxiste 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, l'efficacité de ces applications ainsi que l'impact qu'elles ont sur la stabilité de la grille.

MicroGrid [41, 43] est un émulateur par interception (Section2.2) créé pout résoudre ce problème. Via l'émulation il fournit une grille virtuelle large échelle permettant d'exéuter 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'étant 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 réseaux 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` (section2.2.4). 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 contrôlleur de CPU qui virtualise les ressources du CPU et gère les processus des hôtes virtuelsvia des `SIGSTOP` ET `SIGCONT`. Ce controlleur 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 fonciton 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 coordoniation 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 sophistiqués pour émuler un réseau réaliste et qu'elle permet d'émuler des machines avec des vitesses très variables grace au contrôlleur qui gère la simulation.

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 [11] [29] et Integrated simulation and emulation using adaptative time dilation [12] [32].

3.5 DETER

Dans le domaine de la cyber-sécurité, le test des 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 controllés et prédictibles sont souvent moins efficaces et fiables dans la réalité de par la taille du réseau et des ressources qui constituent son environnement. plus, C'est pour cela que l'USC [13] et l'UC Berkeley [14] ont lancé le projet DETER [1, 17, 36]. À sa création en 2003, DETER était juste 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 ...).

11. 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 réimplémenté afin qu'elle renvoie un temps virtuel.
12. Ce projet dilate le temps pour garder une certaine synchronisation entre applciations 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 `fonciton get_virtual_time` et l'émulation se fait par dégradation avec un émulateur type KVM.
13. University Southern California
14. Université de Californie à Berkeley

Puis en 2004, le besoin de tester ces méthodes se faisant de plus en plus sentir, le développement de DeterLab [15] 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 [16] 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 besoin pour leurs expériences, demandent une certaine configuration du réseau, du système et des applications présente sur les machines. Pour fonctionner ce laboratoire viruel a développé 7 outils complémentaires.

Le premier, qui constitue le cœur logiciel et hardware de DeterLab, se base sur Emulab [42], il l'a étendu 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" (Fig.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 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'appalication. 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".

Ensuite, on trouve le simulateur DASH [17] qui permet de prédire le comportement humain en se basant sur des modèles de pensées, de réactions aux évènements et de comportements instinctifs ou délibérés.

Dans la réalité, lors de cyber-attaques ou cyber-defenses, chaque partie n'a qu'une vision partielle du monde. Par exemple, dans le cas d'un système anti-malware, le système a une vision complète de son réseau local, mais sa vision de la topologie globale du réseau est partielle. Le système doit donc prendre en compte ce facteur pour toutes ses actions. Pour intégrer cela à DeterLab, les chercheurs du projet DETER ont créé le "Multi-party Experiments". Cet outil permet de configurer pour chaque entité d'une expérience le flux d'information qu'elle diffuse et à qui, ainsi que son degré de visibilité sur le réseau.

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. La Fig.10 montre l'éxécution d'une expérience dont les 3 nœuds sont répartis sur différentes plateformes. La difficulté ici est que les plateformes sont controllées par des propriétaires différents ayant des règles de sécurité d'accès souvent très différentes de celles du projet DETER.

15. DeterLab : cyber DEfense Technology Experimental Research Laboratory
16. Attaques DDos et botnes, vers et codes mallicieux, potocoltes de stockage anti-intrusion (intrusion-tolerant), ainsi que le chiffrement et la détection de pattern.
17. Deter Agents Simulating Humans

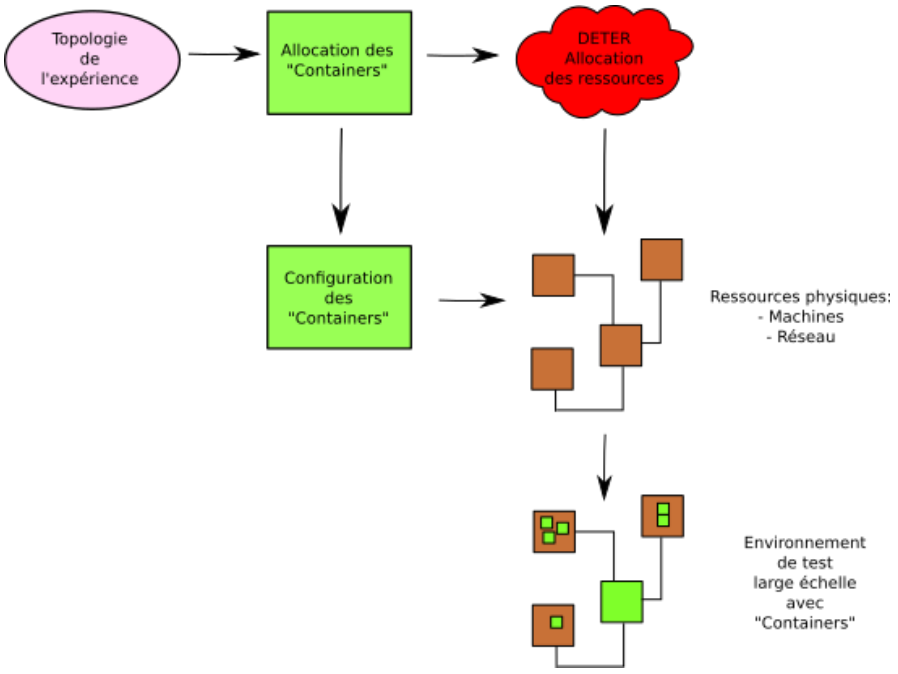


Figure 9 – Diagramme du fonctionnement d'un *Container*

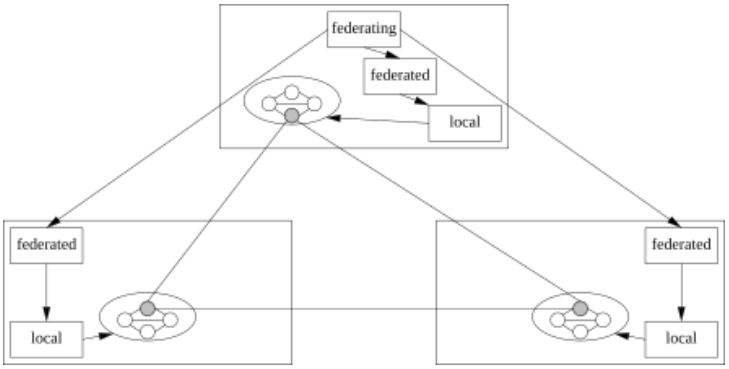


Figure 10 – Fédération d'une experience réparties sur 3 plateformes de tests différentes

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, une authentification pour contrôler l'accès d'une plateforme à une autre. Pour gérer cela on va utiliser trois types de nœuds différents (Fig10). 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 plateforme puis divise l'expérience en sous-expérience qu'il assigne à chaque plateforme. Il gère l'exécution de l'expérience et récupère les données à la fin de l'expérience et libère les ressources utilisées. Le second type de n œud 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 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 reseau 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 nœuds appelé "local" est également présent sur chaque plateforme où l'expérience va s'exécuter. Il gère les communications entre les différentes plateforme 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.

MAGI [18] 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 plus tard.

Pour finir DETER a créé un outil appelé "Risky Experiment Management Capability" qui permet de contrôler les expériences ayant besoin d'un accès à l'extérieur (Internet, ou réseau de la plateforme externe à celui d'exécution). Selon le risque des applications elles n'ont pas toutes les mêmes restrictions. Si une application ne prend aucune mesure de sécurité pendant son exécution alors l'environnement d'exécution sera contrôlé voir isolé du reste du réseau. Dans le cas contraire elle peut avoir la possibilité d'accéder à l'extérieur selon sa criticité avec plus ou moins de droits, cela permet d'avoir des tests encore plus réalistes. Pour cela l'outil va placer des portes vers l'extérieur dans l'infrastructure de l'expérience. Ces portes ne seront pas totalement libres, pour chacune on spécifie le chemin d'entrée et de sortie de l'infrastructure pour un traffic spécifique et l'adresse de la source ou de la destination.

18. Montage AGent Infrastructure

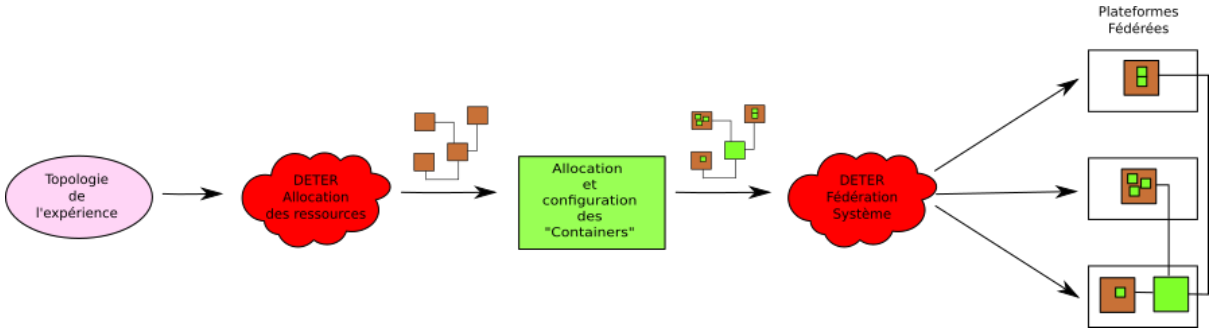


Figure 11 – Création de l'environnement d'une expérience

De par les tests que la plateforme va effectuer il faut bien la sécuriser. En effet, elle pourrait subir des attaques externes ou internes. DETER propose un système de fichier crypté pour contrer l'espionnage interne, un firewall pour éviter les fuites vers l'extérieur, une IDS pour éviter les intrusions.

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 permet de faire des tests large échelle dont la complexité est représentative des réseaux. *Dernièrement de nouvelles expériences sont apparues dans les domaines de la sécurité des réseaux avionique et la robustesse de l'anonymat sur le réseau. De plus, des exercices et des cours concernant les outils fournis par DeterLab et les méthodes expérimentales développées au sein du projet DETER sont mis à disposition des enseignants en cyber-sécurité et de leurs étudiants.***à mettre ou pas ?**

3.6 Robot

Robot [5] est une plateforme de tests automatisée pilotée par mots-clés [19] créée en 2005 par Pekka Klärck [31]. La plateforme est libre de droit et la majeure partie de ses outils et librairies le sont également. Elle permet de lancer des tests "normaux" ou du "ATDD" [20]

Robot possède une architecture modulaire extensible Fig.12. Pour créer un test on le décrit sous forme de tables et on créé une librairie associée qui les transformera en un exécutable lorsqu'elle sera appelée. Il existe quatre types de tables *i)* la table de de paramétrage pour importer des fichiers ou définir des métadonnées, *ii)* la table des variables pour déclarer des variables globales pouvant être utilisées ailleurs, *iii)* la table des *cas de test* contenant les cas réels à tester, *iv)* la table des *mots-clés* qui contient les mots-clés utilisateurs utilisés pour construire les *cas de test*. Les mots clés sont de deux types dans Robot ; les mots-clés utilisateurs définis dans les tables de mots clés et les mots clés de librairies implémentés par du code dans les librairies de test.

19. Utilisation de mot-clés dans la définition des données de test pour déterminer les actions à prendre sur les données lors de l'exécution. Voir keyword-driven testing

20. Acceptance Test-Driven Development [30] : Techniques consistant à utiliser des exemples/tests pour clarifier et documenter les exigences d'un test. Également appelée "Agile Acceptance Testing" ou "Story test-driven development"

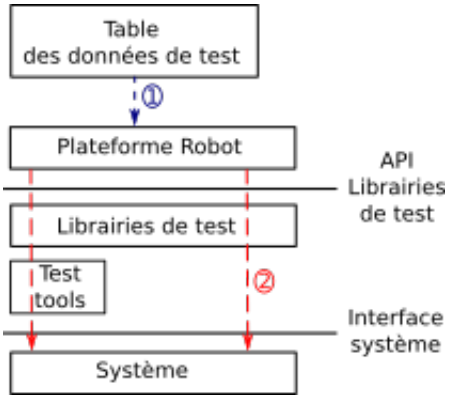


Figure 12 – Schéma de l'architecture de la plateforme Robot

Au lancement d'une exécution la plateforme Robot va analyser les tables décrivant le test en fonction des mots clés utilisés. Les librairies fournissent les mots-clés de librairie [21] que va utiliser Robot pour intéragir avec le système à tester. Les librairies peuvent communiquer directement avec le système ou utiliser d'autres outils de tests (par exemple les drivers). Les outils sont là pour faciliter la création et l'exécution des tests : "libdoc" pour générer les mots clés de librairie et fichiers sources, plug-in vim pour le développement, emacs major mode pour editer les données de test...

La plateforme dispose également d'un système de tag permettant d'ajouter des métadonnées catégorisant les tests, de collecter automatiquement les statistiques d'un test, selectionner les tests à exécuter et spécifier les tests qui sont critiques.

L'exécution rend deux fichiers le "report" et le "log" décrivant ce qui s'est passé durant l'exécution du test. Malgré le détail de ces deux fichiers on ne peut pas suivre l'exécution en temps réel du test.

21. L'architecture étant modulaire si une librairie n'existe pas on peut la créer et l'ajouter facilement a une des trois sections de librairies existants : standard (os, screendshot), externes (ssh, http, ftp) et spécifique à des projets (Android, iOS, Eclipse). On créé ainsi de nouveaux mots-clés de librairie.

4 Simterpose : la médiation

Dans la section précédente, nous avons présentés différents outils permettant de faire de la virtualisation légère. Malheureusement, ils ne pouvaient pas être utilisés dans le cadre de notre projet. Nous allons voir pourquoi et par conséquent quelle plateforme de simlulation a été choisie ainsi que l'émulateur mis en place. Nous étudierons également le fonctionnement interne de ce dernier, notamment les outils présentés en section 2.2 qu'il utilise.

4.1 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 en utilisant uniquement `LD_PRELOAD` pour intercepter les actions semble être un choix risqué de par le contournement possible de l'interception. De plus, il est spécifique aux applications réseaux. RR utilise seccomp/BPF, dans notre cas nous ne souhaitons pas avoir à faire de la programmation noyau. MicroGrid n'étant plus maintenu aujourd'hui, ce n'est pas une bonne idée de lancer un projet bassé sur cet émulateur. DETER est trop spécifique à un domaine. Pour finir, Robot est assez complexe d'utilisation (création des librairies, mots-clés...).

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'éxécuter plusieurs instances d'une application sur une même machine et proposer une large gamme de conditions d'exécutions (simple nœud ou reséau 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 utilisera la plateforme de simulation SimGrid, dont l'architecture est représentée Figure 13, 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 [38] 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.

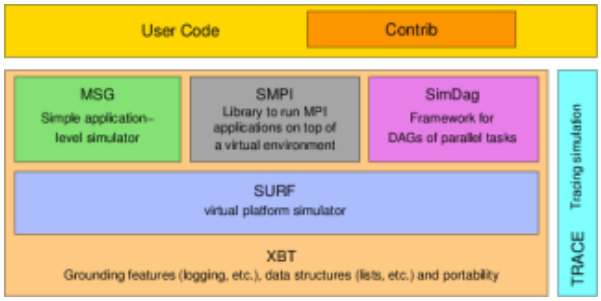


Figure 13 – Architecture de la plateforme SimGrid

La Figure 14 présente l'organisation générale du projet. Simgrid va générer l'environnement virtuel. Simterpose intercepte les actions de l'application et les modifie pour maintenir l'émulation si nécessaire, Figure 15. 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).

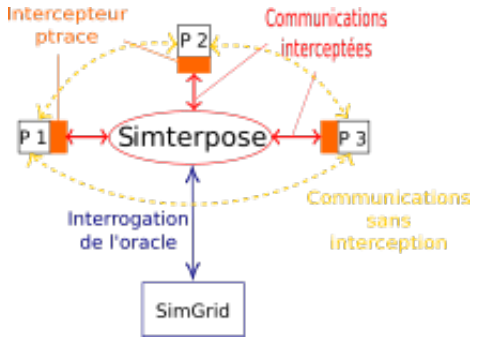


Figure 14 – Architecture de communications entre les différents acteurs

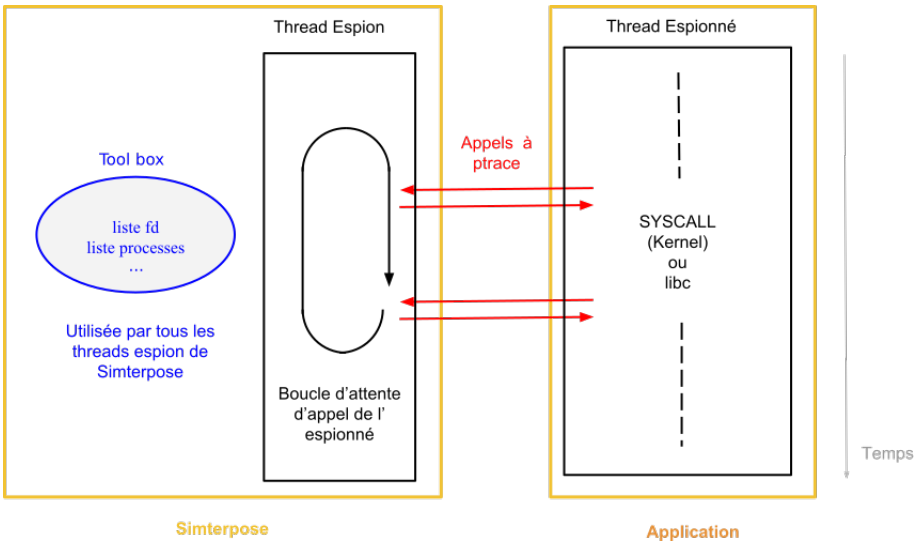


Figure 15 – Le fonctionnement de Simterpose

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

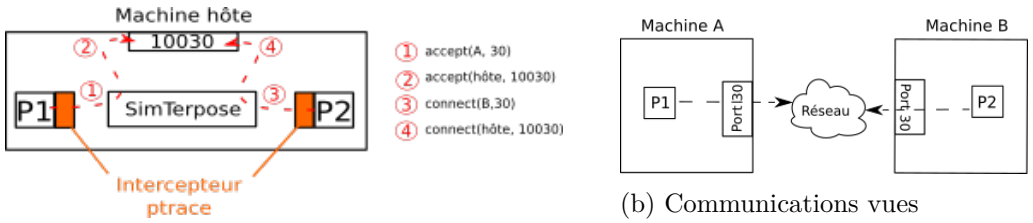
4.2 Foctionnement interne de Simterpose

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. Pour chacune les outils utilisés ne sont pas nécessairement les mêmes. Nous allons donc voir lesquels sont employés parmi ceux cités dans la section 2.2.

4.2.1 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 correspondent pas forcément à ceux attribués dans le réseau local.

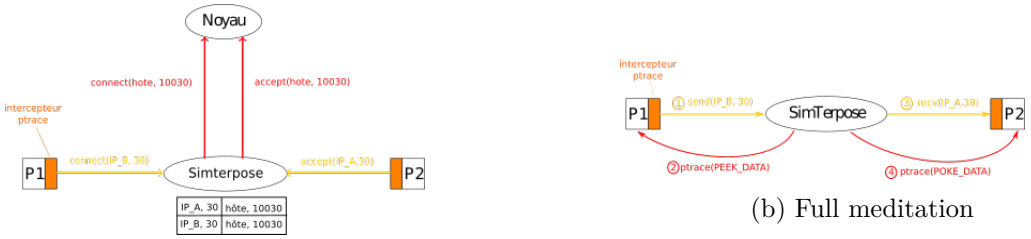


(a) Communications réelles

(b) Communications vues par les processus

Figure 16 – Les communications réseaux 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. Pour gérer toutes ces modifications deux solutions ont été proposées lors d'un précédent stage [40] : la *médiation par traduction d'adresse* et la *full médiation*.



(a) Médiation par traduction d'adresse

(b) Full meditation

Figure 17 – Les différents types de médiation

Traduction d'adresse Avec ce type de médiation, illustrée Fig.17a, 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 <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 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.17b. *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.

4.2.2 Les threads

La gestion des threads se fait à deux niveaux dans Simterpose. On utilise l'interception d'appels système via `ptrace` pour tout ce qui concerne la création de threads (`fork`(), `clone`(), `pthread_create`()). Pour le reste on doit utiliser un autre outil, car il existe des mécanismes utilisés par les threads qui ne passent pas par des appels systèmes pour s'exécuter, par exemple les futex [22]. On va donc utiliser le préchargement de bibliothèques dynamiques en complément. Comme nous l'avons vu en section 2.2.4, nous allons créer une librairie contenant toutes les fonctions utilisées par les threads que nous voulons intercepter (`pthread_init`(), `pthread_join`(), `pthread_exit`(), textttpthread_yield() ...). On placera ensuite la bibliothèque dans la variable d'environnement `LD_PRELOAD` pour que nos fonctions passent avant les autres. Mais il faudra faire attention à ne pas oublier de fonctions essentielles.

4.2.3 Le temps

Nous souhaitons que l'écoulement du temps perçu par les applications soit un temps virtuel, celui qui s'écoulerait dans l'environnement virtuel, 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 prenne 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 dernière étant assez nombreuses, il est moins coûteux, en

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

terme charge de travail pour l'utilisateur, d'intercepter via `ptrace` les appels sytèmes temporels (`time`(), `clock_gettime`(), `gettimeofday`()).

Néanmoins, il a été montré dans un précédent stage [33] 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 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, augmentation du 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. Malgrè 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`, comme nous l'avons vu en section 2.2.4. Néanmoins il faut faire attention de n'oublier aucune fonction.Une fois l'interception temporelle effectuée, Simterpose interroge SimGrid pour obtenir l'heure virtuelle et la renvoie à l'application.

Dans le cas d'une action interceptée (calcul, communications réseaux), 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.

4.2.4 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 aucuns afin que la résolution soit entièrement géré par Simterpose. Pour gérer cela plusieurs solutions sont envisageables.

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

Une autre approche serait de remplacer le fichier `resolv.conf` utilisé pour la résolution de nom. Ainsi, l'utilisateur configurerait son propre fichier, il pourrait égalemment fournir un fichier de spécifications 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 or 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 liés 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 ne pas mettre en péril 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. Le fonctionnement interne de cet émulateur montre la complémentarité de deux outils présentés en section 2.2 : 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, puisque `ptrace` permet d'intercepter les appels systèmes que `LD_PRELOAD` ne permet pas de gérer, on peut dire que `ptrace` résout les problèmes de `LD_PRELOAD`.

5 Conclusion

Références

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

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

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

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

[5] Robot project. http://robotframework.org/.

[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] Craig Larman and Bas Vodde. *Practices for scaling lean & agile development : large, multisite, and offshore product development with large-scale Scrum.* Pearson Education, 2010.

[31] Pekka Laukkanen. *Data-driven and keyword-driven test automation frameworks.* PhD thesis, HELSINKI UNIVERSITY OF TECHNOLOGY, 2006.

[32] 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.

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

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.