Titre long

Étudiante : **Louisa Bessad,**

Encadrant : **Martin Quinson**

Table des matières

Résumé

1 Introduction

Dans le cadre de ce stage, nous allons nous intéresser aux applications distribuées. C'est-à-dire les 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 sur une plate-forme. Les applications distribuées ont de nombreux avantages ; elles permettent notamment d'augmenter la disponibilités des données en se les échangeant et en les stockant lors de communication, comme les applications Torrent (BitTorrent, Torrent...). Grâce au projet BOINC [1] par exemple, on peut également 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 difficiles 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 ; 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 tester l'application 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][1], un autre outil développé à l'échelle mondiale est **PlanetLab** [3]. Néanmoins pour mettre en œuvre cette solution complexe, il faut disposer des architectures nécessaires pour effectuer les tests. Il faut également écrire une application capable de gérer toutes ces ressources disponibles. De plus, du fait du partage des différentes plate-formes entre plusieurs utilisateurs, les expériences ne sont pas forcément reproductibles.

La seconde solution consiste à faire de la simulation, c'est-à-dire à utiliser un programme appelé simulateur pour nous permettre de simuler ce que l'on souhaite étudier. Dans notre 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

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 noeuds. http ://www.planet-lab.org

le modèle de l'application dans un environnement également modélisé et non l'application réelle. Cette solution est donc facilement reproductible, simple à mettre en œuvre du moment que nous savons modéliser notre application et permet de prédire l'évolution du système étudié grâce à l'utilisation de modèles mathématiques. De nos jours, les simulateurs, tel que **SIMGRID**[2, 3], peuvent simuler des applications distribuées mettant à contribution des milliers de noeuds. Néanmoins, avec la simulation on ne peut valider qu'un modèle et pas l'application elle même puisqu'on exécute le modèle de l'application dans le simulateur.

La troisième solution consiste à faire de l'émulation, cela signifie que nous allons exécuter réellement l'application mais dans un environnement virtualisé grâce à un logiciel. On fera ainsi croire à l'application qu'elle s'exécute sur une machine autre que l'hôte. Cette solution représente un intermédiaire entre la simulation et l'exécution sur plate-forme réelle. En effet les actions de l'application sont réellement exécutées sur la machine hôte mais on lui fait croire grâce au simulateur qu'elle se trouve dans un environnement différent de la machine hôte. De plus, cette 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* ; les actions sont directement reportées dans le simulateur et on bloque l'application durant le temps nécessaire calculé par le simulateur.

Dans le cadre du projet Simterpose c'est l'émulation qui a été choisie pour tester des applications distribuées. En effet la simulation n'était pas une bonne solution puisque nous voulons valider les applications et non leur modèle. En ce qui concerne l'exécution sur plate-forme réelle il y avait trop de contraintes matérielles à satisfaire. 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 verrons les projets qui existent aujourd'hui pour ce type de virtualisation. Pour finir en section 4 nous expliquerons 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.

2.1 Virtualisation standard

Avec cette première méthode 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 pas dépasser les capacités de cette dernière. En effet, des machines plus puissantes que l'hôte répondrait plus rapidement que ce dernier à une demande d'une application. Or le délai de réponse géré par l'émulateur ne peut-être inférieur à celui de l'hôte sinon l'hôte n'aurait pas le temps de faire les calculs demandés et de répondre à l'émulateur qui répondrait à l'application. De plus, en choisissant de placer l'émulation comme une sur-couche cela permet de limiter l'accès aux ressources pour les applications. En effet les applications 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 différent outils permettant de faire de mettre en place cette virtualisation, on trouve notamment **cgroup**, **netstat** et **cpuburner**. Cette solution à l'avantage d'être simple à mettre en œuvre puisque l'on se basse 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, pour faire croire à l'application qu'elle s'exécute sur une machine autre que l'hôte on va utiliser deux outils ; un simulateur pour virtualiser l'environnement d'exécution et une API qui va attraper toutes les communications de l'application avec l'hôte et qui les transmettra ensuite au simulateur. Les calculs de l'application seront effectués sur la machine hôte mais c'est le simulateur qui calculera le temps de réponse à l'application. Pour cela il fera un rapport entre le temps d'exécution du calcul sur la machine hôte (fourni par l'API), la puissance de l'hôte et celle des machines de l'environnement que l'on simule. Le temps de l'application sera donc celui du simulateur et non le temps réel. En effet, l'application quand elle fait un calcul pense être sur une autre machine avec des performances différentes, elle est donc capable de savoir combien de temps prends une certain calcul sur son architecture. Hors sur l'hôte ce calcul ne prendra pas le même temps et l'application se retrouvera avec un temps prévu et un temps qui ne correspondent pas ce qui est problématique. Avec cette solution on ne se contente pas de faire de l'interception d'action et du rejeu par l'émulateur comme c'est le cas avec l'émulation par limitation. On va intercepter les actions des applications et faire

de la médiation, autrement dit on va modifier les actions avant de les laisser s'exécuter sous le contrôle de l'émulateur.

Une application distribuée peut vouloir communiquer avec l'hôte soit pour effectuer de simples calculs (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 la machine hôte 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 des données concernant le temps d'exécution de l'action sur la machine hôte pour qu'il puisse calculer le temps d'exécution sur la machine simulée. Les délais calculés par le simulateur sont soit des temps de calculs soit des temps de connexion. *Quand le simulateur a terminé le calcul du temps de réponse nécessaire il l'envoie à l'émulateur qui l'envoie à l'application en plus du résultat afin de mettre à jour l'horloge de l'application.* Ainsi les calculs sont réellement exécuter sur la machine, les communications réellement émises sur le réseau géré par le simulateur et c'est le temps de réponse fourni par le simulateur qui va influencer l'horloge de l'application permettant ainsi d'imiter un environnement distribué. Finalement les applications ne communiquent plus directement entre elles puisque toute communication est interceptée par l'émulateur, puis gérée par le simulateur qui s'occupe du réseau.

Mettre deux schémas de action interceptes, test modifie, renvoie, attrape réponse, simulateur, retour application, un quand simple calcul l'autre quand connexion

Pour intercepter ces actions, il faut d'abord choisir à quel niveau se place l'interception : code source ou binaire. Mais il faut également choisir sur quel type de symbole utilisée par l'application pour exécuter ses actions se fera la médiation qui suit l'interception et avec quel outil. 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 systèmes de la libc par exemple) ou les fonctions POSIX dans le cas d'un système UNIX.

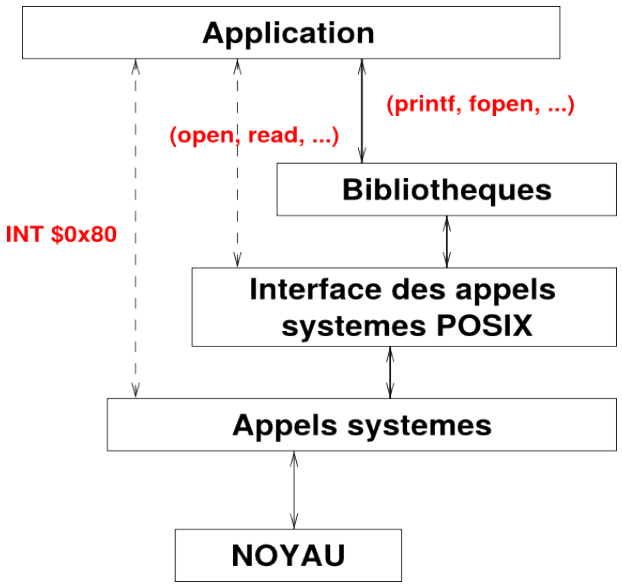

Figure 1 – Communications possibles entre le noyau et une application

Nous allons donc voir comment on peut faire de l'interception sur un fichier source puis sur un binaire, puis nous verrons comment faire de la médiation sur différents symboles : appels de fonctions et appels systèmes.

2.2.1 Action sur le fichier source

2.2.2 Action sur le binaire

2.2.3 Médiation des Appels Système

En regardant la Fig.1, et les différents niveaux d'abstractions, le moyen plus simple pour attraper les actions de l'application en gérant un minimum de choses serait d'intercepter directement les appels systèmes. Les appels systèmes 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 le 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 intercepter les deux parties de l'appel système pour maintenir notre environnement simulé et donc stopper l'application à chaque fois pour récupérer ou modifier les informations nécessaires avant de lui rendre la main pour entrer ou sortir de l'appel système.

Nous allons donc voir comment il est possible de faire cela, puisque de nombreux outils existent et si cette solution se suffit à elle même.

L'API ptrace [4, 5], qui est lui même un appel système permet de tracer tous les événement désirés d'un processus, mais aussi d'écrire et de lire directement dans l'espace d'adressage de ce dernier, à n'importe quel moment ou lorsque un événement particulier

se produit selon le choix du processus. De cette façon il peut contrôler l'exécution d'un processus. C'est un appel système unique 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 parent via un *fork()* ; un processus exécutera l'application et qu'on souhaite contrôler, appelé "processus espionné" et l'autre processus le contrôlera, il se sera 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 point d'arrêt pour le processus espionné, ainsi le processus espion sera appelé deux fois : à l'entrée et à la sortie de l'appel système. Ainsi quand un des processus de l'application voudra faire un appel système quelconque il sera bloqué avant de l'exécuter, l'appel système ptrace sera lancé et notifiera le processus espion. Ce dernier fera les modifications nécessaires dans les registres du processus espionné pour conserver la virtualisation de l'environnement grâce aux requêtes PEEK_DATA et POKE_DATA passées en argument de l'appel système. Puis il rendra la main au processus espionné bloqué pour que l'appel système puisse avoir lieu. Au retour de l'appel système le processus espionné sera de nouveau stoppé, un ptrace sera envoyé au processus espion qui remodifiera les informations nécessaires. Puis il rendra la main au processus espionné bloqué qui sortira de son appel système avec un résultat exécuté sur la machine hôte et un temps d'exécution et une horloge fournie 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.

Mettre un schéma attachement attente père attrape signal modification main fils as retour père...

Néanmoins, pour contrôler un processus ptrace fait de nombreux changements de contexte pour pouvoir intercepter et gérer les événements. De plus il supporte mal les processus utilisant du multithreading, et ne fait pas parti de la norme POSIX donc son exécution peut varier d'une machine à une autre.

Uprobes [4, 5] pour *user-space probes*, quant à lui est une API noyau, également appelé module d'instrumentation, 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. Pour chaque point d'arrêt l'utilisateur fournit un handler particulier à exécuter avant ou après l'instruction marquée. Uprobes étant un outil s'exécutant dans le noyau les handlers doivent être des modules. Pour chaque point d'arrêt géré par Uprobes, on a donc un module noyau qui contient le handler à exécuter, ainsi que le processus et l'adresse virtuelle du point d'arrêt. 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 Utrace

équivalent de ptrace en mode noyau qui permet d'éviter les nombreux changements de contexte et qui est capable de gérer le multithreading. Utrace peut également être utilisé dans le module gérant un point d'arrêt pour récupérer des informations sur l'application et les données qu'elle utilise.

Les deux avantages de l'API noyau est 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 des modules dont il faudra également gérer le bon chargement.

Seccomp/BPF [6] Seccomp est un mécanisme 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, tout le processus 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 limitée. Pour plus de flexibilité on peut utiliser avec seccomp l'outil BPF, pour *Berkeley Packet Filter*. Ce dernier permet de spécifier pour chaque appel système une règle qui définit si on laisse l'appel système s'exécuter ou non *en analysant l'appel système, ses paramètres et le risque qu'il représente*. Ainsi à chaque entrée ou sortie d'un appel système que seccomp autorise, BPF est appelé et reçoit en entrée le numéro de l'appel système, ses arguments et le pointeur de l'instruction concernée. De plus BPF possède une option qui lui permet de générer un appel système ptrace() qu'il envoie au processus espion lié au processus faisant l'appel, s'il existe. Ce qui permet au processus espion de ne plus attendre sur chaque appel système du processus espionné, il ne sera donc plus appelé sur tous les appels systèmes mais uniquement sur les appels systèmes qu'il souhaite intercepter.

2.2.4 Médiation directe des appels de fonctions

linker : LD_PRELOAD

linker got injection *petit tableau comparatif*

3 État de l'art

3.1 CWRAP

3.2 RR

3.3 Distem

3.4 MicroGrid

3.5 DETER

3.6 ROBOT

4 Simterpose : Symboles sur lesquels faire de 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 API pour ce simulateur. Il jouera donc le rôle d'émulateur en nous permettant d'utiliser SIMGRID avec des applications réelles tout en leur faisant croire qu'elles s'exécutent sur des machines distinctes. Simterpose étant l'API 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.

4.1 Organisation générale

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 selon qu'il s'agit d'une action nécessitant l'utilisation du réseau ou non. Dans le cas d'un simple calcul ce qu'il faut maintenir pour l'application, c'est une vision du temps correspondant à celle qui s'écoulerait si elle était vraiment sur la machine simulée. 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 réel 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. 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 [7] : la "médiation par traduction d'adresse" et la "full médiation".

schéma

Traduction d'adresse Avec ce type de médiation on considère que le noyau gère des communications. Ainsi en entrée et sortie d'appel système Simterpose va juste s'occu-

per de la transition entre réseau "virtuel" et 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 adresse IP et des ports "virtuels", on a un couple adresse IP et ports "réel" sur le réseau associé. De fait en entrée d'un appel système de type réseau (bind, connect, accept ...), Simterpose devra remplacer l'adresse et les ports "virtuels" de l'application par l'adresse et les ports réels sur le réseau local, ainsi l'appel système se fera avec une source qui existe réellement sur le réseau. 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 son environnement simulé. 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 des 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. Ensuite ce processus en utilisant ptrace récupérera en lisant dans la mémoire du processus espionné les informations à envoyer ou récupérer et ira directement lire ou écrire ces informations dans la mémoire du destinataire. Même si la "full médiation" permet d'éviter les communications réseaux et de conserver des tables de correspondances, dans le cas d'applications qui communiquent énormément et utilisent de grosses données elle s'avère moins efficace. En effet les appels à la mémoire sont bien plus coûteux que les communications réseaux.

4.3 Les thread

4.4 Le temps

4.5 DNS

5 Conclusion

Références

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] Seccomp. https ://code.google.com/p/seccompsandbox/wiki/overview.

[7] Guillaume SERRIERE. Simulation of distributed application with usage of syscalls interception, 2012.