RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Émulation d'applications distribuées sur des plates-formes virtuelles simulées

Marion Guthmuller, Lucas Nussbaum et Martin Quinson

LORIA / Nancy-Université
{marion.guthmuller,lucas.nussbaum,martin.quinson}@loria.fr

Résumé
L'émulation est une approche expérimentale permettant d'exécuter des applications réelles dans un environnement virtuel, permettant ainsi de se placer dans les conditions expérimentales nécessaires à l'expérience. La plupart des solutions d'émulation reposent sur une infrastructure lourde, et utilisent un cluster et une couche d'émulation pour reproduire l'environnement souhaité. Dans cet article, nous présentons une approche d'émulation basée sur l'interception des actions de l'application à étudier, et sur l'utilisation du simulateur SimGrid pour simuler l'environnement virtuel. Nous exposons les motivations, les objectifs et les apports de cette approche, ainsi que les verrous technologiques à surmonter pour l'implémenter. Puis nous nous concentrons sur le problème clé de l'interception des actions de l'application, et comparons quatre méthodes différentes à plusieurs niveaux de la pile logicielle. Nous finissons par démontrer la faisabilité de la méthode basée sur `ptrace` à l'aide d'un prototype, que nous validons par l'extraction de la trace des communications et des calculs lors de l'exécution d'une application pair-à-pair.

Mots-clés : expérimentation, émulation, interception, SimGrid

1. Introduction

La conception d'algorithmes et d'applications pour des plates-formes distribuées de grande taille – grilles, infrastructures de *Cloud Computing*, systèmes pair-à-pair – pose de nombreux challenges. Il est nécessaire de pouvoir évaluer les performances et les aptitudes à passer à l'échelle ou à résister à des pannes. Différentes approches expérimentales sont largement utilisées [8], parmi lesquelles la simulation avec un simulateur comme SimGrid [2] ou l'expérimentation sur des plates-formes expérimentales réelles comme PlanetLab [3] ou Grid'5000 [6].
Toutefois, ces approches ne sont pas pleinement satisfaisantes. L'utilisation de plates-formes réelles limite l'expérimentateur à l'environnement fourni par la plate-forme, qui n'est pas forcément représentatif de ce que l'application rencontrera une fois déployée, ou n'est pas toujours suffisant pour répondre de manière définitive à une question (il serait nécessaire de combiner des expériences sur plusieurs plates-formes). À l'opposé, la simulation permet de réaliser des expériences dans des environnements virtuels divers en les simulant à l'aide de modèles précis. Mais il n'est en général pas possible d'évaluer des applications réelles avec un simulateur : il est nécessaire d'écrire les algorithmes ou les applications en utilisant une interface spécifique au simulateur.
L'émulation est une approche intermédiaire qui consiste à exécuter l'application réelle dans un environnement virtuel pour permettre à l'expérimentateur de se placer dans les conditions expérimentales souhaitées. Dans la plupart des solutions, l'environnement virtuel est obtenu en utilisant une plate-forme réelle (un cluster) et en y ajoutant une couche d'émulation destinée à réduire les capacités de la plate-forme disponible afin de la faire ressembler à l'environnement cible. Par exemple, pour émuler un réseau longue distance, l'émulateur va rajouter des délais dans chaque communication. Cette approche, que nous nommons *émulation par dégradation*, remplit certes les objectifs de l'émulation, mais elle nécessite le déploiement d'une infrastructure complexe aux niveaux matériels (cluster) et logiciels (émulateur). De plus, il est impossible d'émuler ainsi une plate-forme plus puissante que l'existant.

Dans cet article, nous nous intéressons à une approche alternative d'émulation que nous nommons *émulation par simulation*. Au lieu de réduire les caractéristiques de la plate-forme réelle pour imiter la plate-forme cible, nous proposons de modifier la perception de l'application. Pour cela, chaque action de l'application (calcul, communication) est interceptée, et un simulateur est interrogé pour calculer la réponse de l'environnement virtuel à l'action de l'application.
La suite de cet article est organisée comme suit. Les motivations et les objectifs de ce travail sont d'abord présentés (Section 2). Nous exposons ensuite les différents verrous technologiques à surmonter par un émulateur se basant sur un simulateur (Section 3). Dans la Section 4, nous nous concentrons plus particulièrement sur l'interception des actions de l'application, et comparons quatre méthodes d'instrumentation pour la réaliser. Pour démontrer la faisabilité de l'approche basée sur l'appel système `ptrace`, nous décrivons l'implémentation d'un prototype basé sur cette dernière (Section 5), avant de présenter les travaux antérieurs et de conclure.

2. Motivations

Dans cette section, nous commençons par détailler les objectifs et les cas d'utilisation de notre travail puis nous identifions les critères de succès de l'implémentation du point de vue de la communauté scientifique.

2.1. Objectifs et cas d'utilisation

Actuellement, la majorité des recherches effectuées sur l'émulation s'est concentrée sur les plates-formes d'émulation complexes, situées entre les émulateurs et les plates-formes expérimentales telles que Emulab. Cependant, ces travaux n'ont connu qu'un impact assez faible sur l'ensemble de la communauté car les solutions résultantes restent complexes à mettre en œuvre. Notre objectif est de proposer un outil répondant au besoin croissant d'émulateurs simples. Pour cela, nous nous concentrons sur l'idée d'un émulateur simple, aussi facilement déployable sur un ordinateur personnel que sur un petit cluster. Un tel outil devra alors répondre à plusieurs utilisations :

– *Le repliement de processus :* il doit être possible d'exécuter un grand nombre d'instances d'une application sur un même système, en assistant l'utilisateur à la résolution des problèmes liés à une telle configuration tels que la gestion des identités des différents nœuds du réseau ou la gestion de la mémoire du système.
– *L'évaluation d'applications sous un large éventail de conditions :* la description de la plate-forme cible doit permettre de spécifier différentes caractéristiques pour chaque nœud (CPU plus ou moins rapide pour chaque machine de la plate-forme virtuelle) mais également les caractéristiques réseaux (en utilisant des topologies complexes). L'émulateur doit aussi permettre de tester le comportement d'une application lors d'un changement de conditions comme la défaillance simultanée de nœuds ou bien un changement de performance au niveau du réseau.
– *L'étude du comportement d'une application pendant son exécution :* grâce à la récupération de données de l'expérience sous forme de traces pour les périodes de calcul ou les communications réseaux, il sera possible d'analyser et comprendre les résultats.

2.2. Critères de succès

En nous appuyant sur notre expérience du simulateur SimGrid, nous avons identifié plusieurs critères qu'un émulateur doit satisfaire pour être largement accepté par la communauté.

– *Facilité d'utilisation :* l'émulateur doit être facilement utilisable à la fois par les développeurs des applications et par les chercheurs, sans nécessiter de connaissances expertes dans le domaine ou sans l'aide d'un administrateur système. Plus particulièrement, l'émulateur devra être totalement utilisable sans changement ou recompilation du noyau. Aussi, son infrastructure doit rester simple, et ne pas reposer sur l'exécution de services externes.
– *Performances :* l'émulateur doit présenter de bonnes performances, et ne pas utiliser trop de ressources afin de permettre l'exécution sur la même machine physique d'un grand nombre de nœuds virtuels de l'application.
– *Précision :* bien que les performances soient importantes, la précision de l'émulateur ne doit pas être négligée. Le comportement observé sur une plate-forme virtuelle doit être aussi proche que possible de celui observé sur la plate-forme réelle correspondante. Les limites des différents mo-

dèles d'émulation ou de simulation doivent être étudiées, comme cela est généralement le cas pour les simulateurs.

– *Généricité :* l'émulateur doit être le plus générique possible : il doit permettre l'émulation de différents types de plates-formes (grilles, systèmes pair-à-pair) et d'applications dans différents langages. Les mécanismes utilisés pour intercepter le comportement d'un programme ne doivent pas être spécifiques à un type de programme (des applications distribuées construites sur un intergiciel spécifique, par exemple). Si le mécanisme utilisé empêchait l'émulateur de fonctionner avec certaines applications, il serait nécessaire de le détecter pour éviter que des expériences produisent des résultats faux.
– *Stabilité, pérennité et maintenance :* même si l'émulateur est simple d'utilisation, certains utilisateurs seront plus enclins à l'utiliser si l'outil est visiblement maintenu et reste disponible sur la durée. Par conséquent, l'émulateur devra être développé dans cet esprit. Il est intéressant pour cela de baser les efforts sur des composants déjà stables afin de réduire significativement la complexité du code.

Nous avons décidé de baser nos travaux sur le simulateur SimGrid [2] pour ses performances, sa généricité, la validité bien étudiée de ses modèles et la stabilité de ses interfaces dans le temps.

3. Un émulateur basé sur le simulateur SimGrid

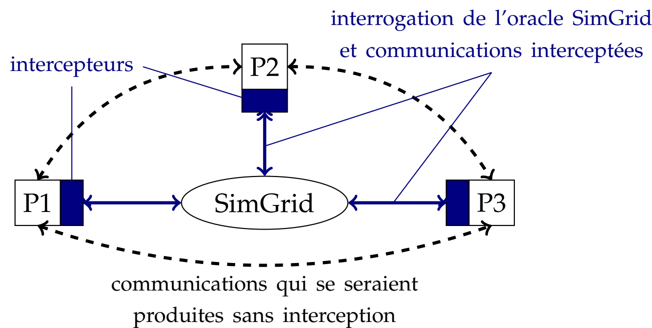

FIGURE 1 – Principe général de l'émulation basée sur le simulateur SimGrid. Trois processus (P1, P2, P3) sont exécutés, et leurs actions sont interceptées, puis redirigées vers le simulateur SimGrid.

Afin de satisfaire les objectifs et critères de succès décrits précédemment, nous proposons une solution d'émulation basée sur l'utilisation du simulateur SimGrid (figure 1). L'application réelle à étudier est exécutée, et ses actions sont interceptées puis redirigées vers le simulateur. Celui-ci est alors utilisé comme un oracle, et calcule la réaction de la plate-forme virtuelle aux actions de l'application. Les phases de calcul et/ou les communications peuvent ainsi être retardées en fonction des caractéristiques de la plate-forme virtuelle sur laquelle l'application est supposée s'exécuter. Si les fonctions liées au temps sont également interceptées, il devient possible de donner l'impression à l'application qu'elle s'exécute plus rapidement qu'elle ne pourrait le faire sur la plate-forme réelle.
Nous décrivons maintenant les différents défis à relever pour concevoir une telle solution d'émulation.

3.1. Sélection des actions à intercepter

Dans un premier temps, il est nécessaire de déterminer les points d'interception dans l'application. On distingue trois types d'actions :

1. *Actions liées à la création et à l'identité des processus* : ces actions doivent être interceptées pour permettre le repliement des processus à étudier. D'une part, il s'agit des appels systèmes qui permettent la création de processus tels que `fork`, `clone` ou `pthread_create`. D'autre part, pour parfaire l'illusion que l'application s'exécute sur la plate-forme réelle, il faut intercepter les appels systèmes liés à la résolution de nom (`gethostname`, ...).
2. *Actions liées aux entrées/sorties et aux communications* : on y retrouve tous les appels liés aux sockets tels que `socket`, `bind`, `connect`, `accept`, `select`, `poll`, `send` ou `recv`.

3. *Actions liées au temps* : de manière similaire aux communications, il est nécessaire d'intercepter les périodes de calcul pour pouvoir émuler des processeurs de vitesses différentes. Par ailleurs, il faut également prendre en compte le biais expérimental induit par le repliement. La première approche possible est de travailler en *temps réel*, en émulant sur chaque machine physique n instances de machines n fois plus lentes, en corrigeant la durée des périodes de calcul par des périodes de sommeil [1]. La deuxième approche possible consiste à virtualiser le temps perçu par l'application en interceptant les appels comme `sleep` ou `gettimeofday` [7]. Cette deuxième approche a l'avantage de permettre un repliement temporel (*time wrap*) des périodes d'attente. Ainsi, l'écoulement du temps perçu par l'application est complètement virtuel, et plus lié au temps physique. Toutefois, cette solution reste complexe du fait du grand nombre d'appels liés au temps dans POSIX. Intercepter l'intégralité de tous ces appels est indispensable pour la généricité de la solution développée.

3.2. Génération de la réaction de la plate-forme virtuelle aux actions de l'application

Deux approches différentes sont possibles pour reproduire l'impact de la plate-forme virtuelle sur l'exécution de l'application.

Il est tout d'abord possible d'utiliser un simulateur pour *calculer* combien de temps l'opération prendrait sur la plate-forme virtuelle ; l'opération n'est pas réalisée réellement. Il est donc nécessaire que le simulateur dispose d'un modèle précis des différents composants de la plate-forme (CPU, réseaux ou autre), comme c'est le cas pour le simulateur SimGrid.

L'autre alternative est de laisser l'application exécuter réellement les actions tout en mesurant leur durée, puis d'utiliser le simulateur pour calculer le temps qui aurait été nécessaire à l'exécution sur la plate-forme virtuelle. L'exécution pourra ensuite être bloquée pendant le temps correspondant à la différence entre ces deux durées s'il est nécessaire de ralentir l'action pour émuler une plate-forme virtuelle plus lente.

3.3. Interception des actions de l'application

Une fois décidés les points d'interception, il s'agit d'effectuer en pratique cette interception. Plusieurs solutions sont envisageables :

– *Interception par virtualisation complète :* cette solution consiste à exécuter les instances de l'application dans des machines virtuelles, et à réaliser l'interception autour de ces machines virtuelles. C'est une solution aussi lourde que peu extensible par rapport à d'autres.

– *Interception au niveau du middleware utilisé par l'application :* lorsque l'application utilise un middleware spécifique, il est possible de réaliser l'interception au niveau de ce middleware, comme c'est le cas dans SMPI [4] qui constitue une réimplémentation partielle de MPI au dessus de SimGrid. Bien qu'offrant un accès à une sémantique plus riche des opérations interceptées par rapport au niveau système, cette solution manque de généricité, puisque tout le travail serait à refaire pour un middleware différent.

– *Interception au travers d'outils systèmes :* une approche intermédiaire consiste à utiliser des mécanismes systèmes ou des outils n'étant pas limités à un type d'application particulier, tels que l'appel système `ptrace` ou l'outil Valgrind. Cette approche, qui constitue un bon compromis entre la légèreté et la généricité, est détaillée dans la section suivante.

4. Évaluation de quatre méthodes systèmes pour l'interception des actions de l'application

Dans cette partie, nous présentons et comparons quatre méthodes génériques et légères pour l'interception des actions de l'application. Ces solutions diffèrent par leur niveau dans la pile logicielle (*cf.* figure 2). Nous évaluons ces différentes méthodes par rapport à différents critères :

- Leur niveau d'interception, et donc leur capacité à intercepter toutes les actions d'intérêt ;
- Leur coût et leur impact sur les performances de l'application ;
- Leur facilité d'utilisation pour l'implémentation de l'émulateur.

4.1. Valgrind

Valgrind est un outil de programmation libre qui permet de déboguer ou profiler du code et mettre en évidence des éventuelles fuites de mémoire. Il permet de dérouter les appels aux fonctions à intercep-

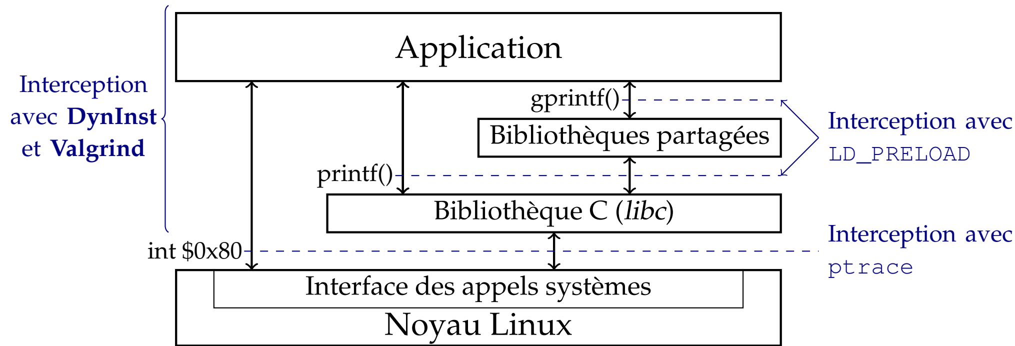

FIGURE 2 – Positionnement des méthodes évaluées dans la pile logicielle

ter vers d'autres fonctions fournies par l'utilisateur. Valgrind travaille directement au niveau du code binaire de l'application cible et des bibliothèques utilisées par cette application, et utilise une phase de décompilation/recompilation pour réaliser l'interception. Cet outil a été écarté rapidement car cette phase génère un code binaire bien moins optimisé que l'original. Lors de nos tests, nous avons mesuré des temps d'exécution multipliés par 7.5, ce qui peut être acceptable pour un outil de débogage, mais ne l'est pas pour un émulateur.

4.2. DynInst

DynInst est une bibliothèque d'exécution multiplate-forme de code correctifs, utilisée par des outils de mesure du rendement, des débogueurs et des simulateurs, comme *TAU* et *Open SpeedShop*. Grâce à son API, il est possible d'injecter du code directement dans une application en cours d'exécution. Dans notre étude, cette approche se situe donc, comme Valgrind, au niveau de la couche application. Toutefois, celle-ci semble plus prometteuse du fait d'un surcoût faible sur le temps d'exécution, contrairement à Valgrind. Cependant, sa mise en oeuvre est difficile : d'une part, l'API de DynInst est très bas niveau avec un niveau d'abstraction très élevé ce qui mène rapidement à un code très complexe, long et difficile à implémenter (même sur des exemples simples) ; d'autre part, l'utilisation de DynInst rajouterait une dépendance logicielle très importante dans le cas d'un émulateur.

4.3. Préchargement d'une bibliothèque avec LD_PRELOAD

Le principe est ici d'intercepter des appels de bibliothèque en chargeant au préalable une bibliothèque d'interception avec la variable d'environnement `LD_PRELOAD`. Cette bibliothèque d'interception contiendra alors des fonctions qui seront exécutées à la place des fonctions exécutées normalement. Cette approche permet donc de modifier le comportement d'une application de façon indirecte, sans avoir à recompiler ou rééditer les liens à chaque fois. Cependant, elle ne permet que de surcharger les fonctions des bibliothèques sans pouvoir être appliquée aux appels systèmes. Cela peut poser problème de par le grand nombre de fonctions existantes. Par exemple, pour intercepter les écritures dans un descripteur, il faut surcharger les fonctions `printf`, `fprintf` et toutes leurs variantes car l'appel système unique `write` est inaccessible. Cette multiplicité est de plus répétitive et source potentielle d'erreur. Enfin, cette approche est ineffective pour une application compilée de manière *statique*.

4.4. Interception des appels système avec `ptrace`

Introduit pour permettre l'écriture de débogueurs comme `gdb`, l'appel système `ptrace` offre un moyen de contrôler l'exécution d'un autre processus, exécuté comme le processus fils du processus *traceur*. À chaque fois que l'application tracée réalise un appel système, son exécution est arrêtée pour permettre au traceur d'examiner et/ou modifier son état avant reprise. Il s'agit donc d'un service fourni directement par le noyau. Son interface n'est cependant pas normalisée dans le cadre de POSIX, et l'API diffère entre les systèmes au détriment de la portabilité. De plus, l'API fournie par Linux a évolué pour rajouter des fonctionnalités, notamment dans le cadre du suivi des processus ou threads fils.

Par ailleurs, `ptrace` nécessite un travail particulier pour prendre en compte les différentes architectures

matérielles (x86, x86_64) en raison des différentes conventions d'appels systèmes (par exemple, le multiplexage des appels systèmes liés au réseau avec l'appel socketcall sur x86).
En raison des limitations de ptrace, une nouvelle interface est en cours de développement dans Linux : *Uprobes* [9]. Uprobes, permet de pénétrer dynamiquement dans une routine d'une application et de collecter des informations de débogage sans interruption. Pour cela, on insère des points d'arrêts à n'importe quelle adresse de code et un gestionnaire est invoqué quand un point est atteint. Il est ensuite possible de modifier l'application à chaque interruption puis de relancer son exécution du point où elle a été arrêtée. Bien que prometteuse, cette approche n'a pas été approfondie pour notre étude car Uprobes est encore en développement et n'est pas encore intégré au noyau.

4.5. Tableau récapitulatif de notre évaluation des méthodes systèmes d'interception des actions

	Valgrind	**DynInst**	**LD_PRELOAD**	**Ptrace**
Capacité d'interception	moyenne	moyenne	faible	très bonne
Coût	important	assez faible	faible	moyen
Facilité d'utilisation	complexe	très complexe	simple	assez complexe

5. Implémentation d'un prototype avec `ptrace`

Bien que l'étude précédente montre qu'il n'existe pas de solution idéale, il nous semble que ptrace est la meilleure technique existante pour implémenter l'interception nécessaire à un outil d'émulation par simulation. Nous avons donc implémenté un prototype se basant sur cette approche afin de pouvoir extraire une trace des actions applicatives qui pourra ensuite être rejouée dans SimGrid. Nous détaillons maintenant les verrous que nous avons dû lever pour établir ce prototype. Le résultat est comparable à l'utilitaire strace, mais reporte également des informations sur les temps de calcul entre les appels systèmes.

Suivi des processus fils et des threads

Depuis les noyaux Linux 2.6, il est possible de suivre correctement les créations de processus ou de threads par l'interface ptrace. Des événements supplémentaires sont générés, et le traceur doit les détecter pour démarrer le traçage des processus fils lors de leur création.

Extraction des appels systèmes et des paramètres

L'identification des appels systèmes et de leurs éventuels paramètres s'effectue à partir des registres de données au travers des primitives PTRACE_PEEKTEXT et PTRACE_PEEKDATA. Cette interface reste relativement difficile d'usage, peu efficace et pose de plus des problèmes de portabilité. En effet, des différences entre les architectures matérielles, comme l'utilisation de registres ou de la pile ou l'appel socketcall mentionné en 4.4, compliquent l'analyse puisqu'il est nécessaire de maîtriser ces spécificités pour assurer une implémentation générique, fonctionnant à la fois en 32 et en 64 bits.

Identification des processus communiquants

Dans le cas d'appels systèmes comme read ou send induisant une communication entre processus, il est nécessaire d'identifier le processus destinataire de chaque communication, afin de connaître le comportement global de l'application. Pour cela, il faut extraire des informations sur chaque socket qui permettent de connaître les deux couples *(IP, port)* locaux et distants de chaque descripteur de socket. Pour obtenir ces informations, le numéro de la socket est récupéré dans /proc à partir de son numéro de descripteur et du processus auquel elle appartient. Il faut ensuite parcourir la table des connexions ouvertes pour le protocole auquel s'applique la socket grâce au fichier /proc/net/*protocol* où *protocol* peut être tcp, udp, raw, netlink, etc. En récupérant ainsi les deux couples distants et locaux pour chaque socket, il suffit alors de comparer l'ensemble des couples de chaque socket et de trouver une socket

possédant des couples *(IP, port)* inverses à ceux de la socket utilisée par l'appel système. Le processus associé à cette socket peut alors être identifié comme étant le processus destinataire de cet appel.

Détermination des périodes de calcul

En plus d'intercepter les appels effectués par l'application pendant son exécution, il est nécessaire d'identifier les périodes de calcul pour les transmettre au simulateur, tout en les distinguant des périodes d'attente. Pour cela, à chaque appel système, les *temps processeur* (*CPU time*) et *temps horloge* (*Wall time*) sont récupérés via l'interface *NETLINK TASKSTATS*, qui est l'interface du noyau fournissant la meilleure précision.

Évaluation sur une application pair-à-pair réelle : BitTorrent

Afin de valider notre prototype, des tests ont été effectués sur des applications synthétiques, dans un premier temps. Pour vérifier l'interception des communications entre plusieurs processus, celui-ci a été testé avec un serveur communiquant avec plusieurs clients. Pour la détermination des périodes de calcul, le prototype a été évalué sur une implémentation multi-processus de la résolution du problème des n reines.

Pour se confronter à une application réelle beaucoup plus complexe, nous l'avons ensuite utilisé sur l'application pair-à-pair BitTorrent, en traçant le téléchargement d'un fichier de 100 Mo entre deux processus. Nous obtenons correctement l'ensemble des périodes de calcul et des communications pour chaque thread des processus impliqués. Notre traceur est aujourd'hui adapté à une large gamme de cas.

6. Travaux antérieurs

Des émulateurs comme Modelnet [11], DieCast [7], Emulab [12] ou Wrekavoc [1] permettent d'exécuter des applications distribuées dans des environnements virtuels pour évaluer leurs performances. Toutefois, ces solutions requièrent une infrastructure lourde en demandant l'installation d'une suite logicielle complexe sur un cluster, et ne permettent pas d'émuler une plate-forme plus puissante que celle à disposition. En revanche, les biais expérimentaux sont probablement moindres. En ce sens, ces outils sont complémentaires à notre approche : selon l'étude, l'une ou l'autre approche peut se révéler préférable.

MicroGrid [10] est le projet se rapprochant le plus de notre travail. Il utilise un ensemble de machines physiques pour exécuter l'expérience. Le réseau est simulé à l'aide d'un simulateur réseau distribué. L'interception des actions de l'application est réalisée via `LD_PRELOAD`, en interceptant les appels de bibliothèques liés aux communications. Cette interception n'étant pas parfaite, elle peut être contournée par certaines applications. Les appels liés à la résolution de noms (DNS) sont également interceptés pour prendre en compte l'identité du nœud virtuel à émuler. Enfin, MicroGrid virtualise le déroulement du temps afin de compenser les perturbations causées par le repliement de l'application. Tous ces aspects feraient de MicroGrid une solution très intéressante s'il était encore utilisable. Malheureusement, la dernière version de MicroGrid publiée date de 2004, et il n'est plus possible de le faire fonctionner sur les systèmes actuels.

Au sein du framework SimGrid, SMPI [4] permet d'exécuter des applications MPI existante en utilisant SimGrid pour simuler le réseau. L'interception étant réalisée au travers d'une réimplémentation de l'interface MPI, il est nécessaire de recompiler l'application au préalable.

Enfin, Trickle [5] est un outil permettant de limiter le débit réseau occupé par une application. Les appels réseaux de l'application sont interceptés au moyen de `LD_PRELOAD`. Il ne permet de plus que de ralentir des connexions TCP.

7. Conclusion et perspectives

Dans cet article, nous nous sommes intéressés à la conception d'un émulateur pour permettre l'évaluation d'applications distribuées sur des plates-formes virtuelles simulées, en visant une solution basée sur le simulateur SimGrid. Nous présentons les objectifs et les avantages de cette approche prometteuse, ainsi que les défis à relever. Le premier d'entre eux est l'aptitude de l'émulateur à intercepter les actions de l'application. Le simulateur est ensuite utilisé pour calculer la réponse de la plate-forme à ces actions. Nous évaluons et comparons quatre méthodes d'interception, à différents niveaux de la pile logicielle.

Valgrind et DynInst proposent une interception au niveau de l'application mais leur utilisation est complexe. De plus, l'interception par Valgrind génère un surcoût très important dans le temps d'exécution. L'utilisation du préchargement d'une bibliothèque d'interception avec `LD_PRELOAD` est plus simple à implémenter, mais dans ce cas, le travail se fait uniquement au niveau des appels de bibliothèques. Il est alors difficile d'intercepter tous les appels concernant les entrées/sorties. L'appel système `ptrace` s'est révélé être l'approche la plus intéressante grâce à une interception au niveau du noyau, et un surcoût acceptable, malgré quelques problèmes de portabilité. Un prototype de cette approche a donc été réalisé et nous a permis d'obtenir une trace complète des périodes de calcul et des communications effectuées par chaque processus lors d'un transfert de fichiers avec BitTorrent.

Ce prototype constitue une première étape en direction d'un émulateur permettant d'étudier des applications distribuées réelles sur une plate-forme virtuelle simulée avec SimGrid. L'étape suivante sera de permettre le rejeu des traces acquises par ce biais pour rendre possible une étude *offline* (ou post-mortem) des applications sur simulateur. Cette tâche sera simplifiée par l'existence d'un mécanisme générique de rejeu de traces applicatives dans SimGrid.

L'exécution d'une application sur une plate-forme différente peut induire des échanges de messages différents, par exemple si chaque instance de l'application prend des décisions en fonction des performances des autres instances. Pour étudier ce type d'applications, il sera nécessaire de réaliser des études *online*, en interfaçant directement l'intercepteur d'actions avec le simulateur pour mettre en place une boucle de feedback. Certains détails techniques devront également être résolus, comme la mise en place d'un lanceur, permettant de démarrer l'application en mettant en place la mécanique d'interception et de feedback, et en démarrant automatiquement les différents nœuds de l'application sur les différents éléments de la plate-forme simulée. Cette solution pourra être étendue vers une émulation *distribuée* afin de permettre la distribution des processus utilisateurs sur différentes machines et donc permettre une réelle exécution en parallèle. Il sera, dans ce cas, possible d'utiliser un cluster pour exécuter des instances de l'application qui seraient trop grosses pour pouvoir être exécutées sur une seule machine.

L'environnement d'exécution constitué de ces éléments offrira un *framework* intégré pour l'étude d'applications réelles inchangées sur des plates-formes simulées, combinant la plus grande simplicité d'utilisation possible avec la puissance prédictive de SimGrid.

Bibliographie

1. Louis-Claude Canon, Olivier Dubuisson, Jens Gustedt, et Emmanuel Jeannot. Defining and controlling the heterogeneity of a cluster : The Wrekavoc tool. *J. Syst. Softw.*, 83 :786–802, May 2010.
2. Henri Casanova, Arnaud Legrand, et Martin Quinson. SimGrid : a generic framework for large-scale distributed experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, 2008.
3. Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, et Mic Bowman. PlanetLab : an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3) :3–12, 2003.
4. Pierre-Nicolas Clauss, Mark Stillwell, Stéphane Genaud, Frédéric Suter, Henri Casanova, et Martin Quinson. Single node on-line simulation of MPI applications with SMPI. In *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, May 2011.
5. Marius A. Eriksen. Trickle : a userland bandwidth shaper for Unix-like systems. In *USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 2005. USENIX Association.
6. Franck Cappello et al. Grid'5000 : a large scale, reconfigurable, controlable and monitorable Grid platform. In *Grid'2005 Workshop*, Seattle, USA, 2005.
7. Diwaker Gupta, Kashi V. Vishwanath, et Amin Vahdat. DieCast : testing distributed systems with an accurate scale model. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.
8. Jens Gustedt, Emmanuel Jeannot, et Martin Quinson. Experimental validation in large-scale systems : a survey of methodologies. *Parallel Processing Letters*, 19 :399–418, 2009.
9. Jim Keniston et Srikar Dronamraju. Uprobes : User-space probes. In *Linux Foundation Collaboration Summit*, 2010. `http://events.linuxfoundation.org/slides/lfcs2010_keniston.pdf`.
10. H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, et A. Chien. The MicroGrid : a scientific tool for modeling computational grids. In *Proceedings of IEEE Supercomputing*, 2000.
11. Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, et David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI) :271–284, 2002.
12. Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, et Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI'02*, Boston, MA, 2002.