Méthode pour l'étude expérimentale par la simulation de clouds avec SCHIaaS

Luke Bertot, Julien Gossa, Stéphane Genaud

Compas'2017 : Parallélisme / Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

Méthode pour l'étude expérimentale par la simulation de clouds avec SCHIaaS.

Luke Bertot, Julien Gossa, Stéphane Genaud

Université De Strasbourg,
Laboratoire ICube - Pôle API - 300 Bd Sébastien Brant
67400 Illkirch-Graffenstaden - France
lbertot@unistra.fr gossa@unistra.fr genaud@unistra.fr

Résumé
Les clouds ont été intensivement étudiés les dernières années, que ce soit du point de vue de l'administrateur qui doit par exemple décider de l'emplacement des machines virtuelles sur l'infrastructure matérielle, ou du client qui doit décider du dimensionnement de sa plateforme virtuelle. Pour ce faire, de nombreux simulateurs ont vu le jour. Malheureusement, ces derniers se limitent souvent à mimer les fonctionnalités d'un cloud. Ils manquent d'ancrage dans la réalité, en raison d'une part d'une absence de comparaison à des exécutions réelles, et d'autre part d'un ensemble de traces réelles injectables dans les simulations. De plus, ils ne proposent pas un cadre complet d'étude des résultats obtenus pourtant indispensable à la reproductibilité des expériences et la comparaison de solutions différentes. Le projet SCHIaaS vise à combler ces deux manques. Cet article se focalise sur ce dernier point, en présentant un *framework* d'étude par la simulation de bout-en-bout, de la conception du simulateur à l'analyse des résultats.

Mots-clés : simulation ; cloud ; reproductibilité

---

1. Introduction

Les clouds sont devenus ces dernières années une brique essentielle de l'informatique, omniprésents dans les architectures de nos systèmes d'information. La compréhension de leurs comportements est donc un enjeu majeur qui a incité les chercheurs à proposer des simulateurs de cloud, essentiellement pour les clouds de type *Infrastructure-as-a-Service* (IaaS). Ces simulateurs reposent sur une modélisation que l'on peut qualifier de *bottom-up* : les ressources matérielles sont d'abord modélisées, puis le comportement des machines virtuelles sur ces ressources matérielles, puis les applications s'exécutant dans cet environnement. On peut donc distinguer dans de nombreux outils de simulation trois composants :

— une spécification de plateforme
— une spécification de l'application.
— un modèle de simulation

Le modèle de simulation forme le cœur d'un simulateur. Il décrit l'évolution de l'état de tous les composants du système simulé en fonction du temps et des évènements. Il repose sur des modèles individuels pour chacun de ces composants (réseau, processeur, ...). Etant donné la complexité du comportement des composants, les modèles actuels procèdent à des simplifications, ce qui engendre des écarts avec la réalité. La précision des modèles utilisés est le principal

élément discriminant entre simulateurs, l'utilisateur n'ayant généralement que peu de prise sur cet aspect [1].

La spécification de plateforme décrit l'environnement simulé. Elle est fournie par l'utilisateur et décrit les caractéristiques techniques de l'infrastructure, comme la topologie d'interconnexion, la puissance des processeurs, la capacité des liens réseaux, etc. Le niveau de détail de cette description, qui diffère selon les simulateurs, influence également la précision de la simulation.

La spécification de l'application, fournie par l'utilisateur, est une description de la séquence des évènements à simuler. Les simulateurs existants privilégient cette description sous la forme d'un programme qui, à l'aide des primitives de la bibliothèque de simulation, décrit le comportement des opération réelles simulées. Mais d'autres mode de description peuvent être proposés, comme l'injection d'une trace d'exécution réelle, ou une représentation abstraite de suite d'opérations.

Le tableau suivant donne un aperçu des possibilités des principaux simulateurs du domaine, en terme d'application : (a) trace d'exécution, (b) représentation abstraite, (c) simulation programmée ; de performance CPU : (d) délai mesuré, (e) délai utilisateur ; de réseau : (flow) analytique ou (packet) niveau paquet ; et enfin de disque.

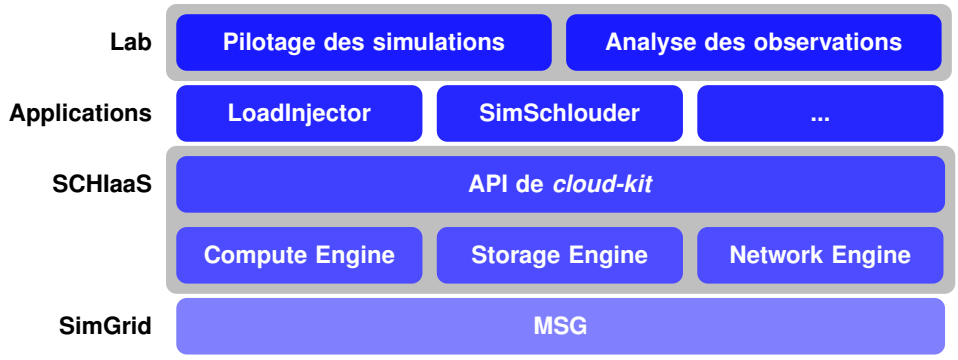
	Application			CPU		Réseau	Disque
Simulator	(a)	(b)	(c)	(d)	(e)	type	precision max
CloudSim[1]			**X**		**X**	store&forward	seek+transfert
ICanCloud[7]			**X**		**X**	packet	bloc
GroudSim[8]			**X**		**X**	flot	n/a
GreenCloud[5]		**X**			**X**	packet	n/a
SimGrid[2]	**X**	**X**	**X**	**X**	**X**	flot/packet	seek+transfert

Une fois le simulateur réalisé, l'expérimentateur peut étudier le comportement de ses composants en observant les traces de ses exécutions, parfois en très grand nombre. Les observations arrivent donc seulement en fin de processus. Or, l'étude de ces observations mènent souvent à des modifications du simulateur, ainsi que du choix des observations pertinentes. Il faudrait ensuite refaire toutes les simulations et toutes les observations, ce qui est une tâche fastidieuse souvent négligée par l'expérimentateur qui se limite aux quelques simulations et observations qui lui paraissent pertinentes sur l'instant. Ce processus artisanal est source d'erreur dans la compréhension des phénomènes simulés : n'appliquant pas les mêmes observations à toutes les simulations, certaines choses peuvent échapper à l'expérimentateur.

L'objet de cet article est de présenter le *framework* SCHIaaS, qui repose sur SimGrid, et qui grâce à un outil appelé *le lab* permet de standardiser les études par simulation afin d'éviter ces travers. Nous présentons d'abord ce *framework* dans la section 2, puis un cas d'utilisation concret en section 3 et enfin une conclusion en section 4.

1. SimGrid permet néanmoins de choisir différents modèles pour le réseau

2. Présentation du *framework*



2.1. SCHIaaS et SimGrid

SimGrid [2] est un simulateur à évènements discrets qui permet à l'utilisateur d'exprimer la modélisation d'une application distribuée à travers un programme utilisant des primitives qui décrivent les phases de communications et de calcul des différents processus applicatifs. Ces primitives sont disponibles à travers différentes interfaces, adaptées aux systèmes étudiés ; il existe par exemple des interfaces pour les applications pair-à-pair, MPI (SMPI), les workflows sous forme de DAG (SimDAG), ou les processus communicants de type CSP (MSG).

Le travail présenté dans cet article repose sur une nouvelle interface de SimGrid, nommée SCHIaaS pour *Simulation of Cloud, Hypervisor and IaaS*. Nous ne présentons ici que quelques éléments saillants de SCHIaaS mais le lecteur pourra trouver une description détaillée sur le site dédié [4]. SCHIaaS est un *framework* permettant le développement rapide de simulateurs de cloud. Il peut être utilisé pour évaluer des algorithmes au niveau du gestionnaire de cloud (appelée aussi *cloud-kit*, par exemple OpenStack), mais aussi au niveau d'applications clientes d'un cloud.

Développé en JAVA, il offre un ensemble de classes prédéfinies représentant tous les composants d'un cloud kit. L'utilisateur qui souhaite insérer sa description du comportement d'un ou plusieurs composants, peut rapidement greffer cette description sous la forme d'un programme qui vient spécialiser l'une ou l'autre de ces classes.

La simulation du fonctionnement du niveau *cloud-kit* est assurée par des moteurs, exposant les fonctionnalités des clouds au niveau administrateur :

— Compute : moteur d'instanciation et de gestion des VM ;
— Storage : moteur de gestion des stockages ;
— Network : moteur de gestion de la virtualisation du réseau.

Chacun de ces moteurs est une interface abstraite assurant l'interopérabilité entre eux ainsi qu'avec les autres niveaux du *framework*. Des implémentations de ces moteurs sont fournies, permettant d'avoir un simulateur par défaut pleinement fonctionnel. Ces implémentations imitent le comportement d'OpenStack. Un développeur de fonctionnalités administrateur n'a donc qu'à se concentrer sur la partie qui l'intéresse, par exemple différents systèmes de stockages.

Les ordonnanceurs de machines virtuelles (VM), qui décident sur quelles machines physiques (PM) seront déployées les VM demandées par l'utilisateur, ont également leur interface abstraite. Plusieurs ordonnanceurs par défaut sont fournis, sur l'exemple d'Openstack qui permet au choix d'équilibrer ou de consolider les charges des machines physiques au moyen d'un poids calculé pour chacune d'elles.

Un système abstrait d'injection de charge est également fourni. Ce dernier permet de contrôler

un nombre de VM déployées et les charges de ces VM, en particulier CPU.
Ces interfaces ne sont que celles utilisées dans la suite de l'article à titre d'illustration, mais tous les composant d'un *cloud-kit* sont présents dans SCHIaaS.

2.2. Applications

SCHIaaS ne peut s'exécuter seul. Dans la philosophie SimGrid, un simulateur est une application exploitant les fonctionnalités fournies, mais qui doit être développée et compilée. Cette philosophie est conservée avec SCHIaaS, qui ne fait qu'étendre les fonctionnalités de SimGrid avec celles disponibles dans un cloud.
Cependant, deux applications sont fournies par défaut :

— *LoadInjector*, qui permet d'injecter une charge utilisateur sans se préoccuper des applications qui s'exécutent dans les VMs. Il est utile pour mener des études au niveau administrateur. Deux injecteurs par défaut sont fournis. Le premier est un simple injecteur de charges sinusoïdales entièrement paramétrables. Le second injecte les charges observées sur l'infrastructure de Google, et rendu disponible en tant que google cluster-data[3].
— *SimSchlouder*, qui est la version simulée du système réel Schlouder[6]. Il s'agit d'un courtier de ressources virtuelles capable de piloter de concert un ordonnanceur de tâches et un *cloud-kit* pour exécuter des calculs scientifiques sur clouds.

2.3. Le Lab

Le lab est un ensemble de scripts permettant d'automatiser l'exécution des simulations, la collecte des observations, ainsi que leur analyse. Il permet donc d'exécuter de bout-en-bout l'étude par simulation, de la définition des différentes simulations, à la production des graphiques.
Au delà de l'aspect pratique et du gain de temps de mise en œuvre de l'étude, le lab a pour vocation de "standardiser" l'étude. Cette standardisation assure sa reproductibilité, ainsi qu'une comparaison équitable entre solutions à un même problème.
Mais le lab permet également une approche systématique permettant à l'expérimentateur de ne pas rater de phénomène. En effet, il est fréquent de faire un grand lot de simulations, puis d'observer plus finement celles qui présentent des particularités. Ce faisant, l'expérimentateur exclu les autres simulation de ces observations plus précises, à moins de les refaire entièrement. En rendant plus pratique la définition des observations directement au niveau du *workflow* de simulations, le lab assure que tous les cas seront observés de la même manière, ce qui évite de rater un phénomène, ou de différencier le traitement des simulations au risque d'arriver à des conclusions abusives.

3. Cas d'usage concret

Plaçons nous dans le cas du problème, parfois appelé *VM Packing*, de l'ordonnancement des VMs sur les machines physiques. Supposons que l'on dispose d'un algorithme visant à reconsolider régulièrement les VMs, c'est-à-dire à les concentrer sur le moins de PMs possibles en les migrant, et que l'on souhaite étudier l'impact du *délai* entre ces reconsolidations sur le *nombre de PMs utilisées* et sur le *nombre de migrations* nécessaires, lorsque le cloud est soumis à une charge sinusoïdale.
Pour mener cette étude, les différentes étapes sont :

1. la conception du simulateur ;
2. la description du contexte expérimental des simulations ;
3. la description et l'exécution des simulations ;
4. l'analyse des résultats.

3.1. Conception du simulateur

Notre *framework* permet de partir sur la base d'un simulateur pleinement fonctionnel en sélectionnant les modules par défaut pertinents pour l'étude. Les développements se limitent alors strictement aux parties spécifiques à l'étude.
Dans le contexte de cette étude, le développement concernera seulement l'interface `ComputeScheduler`, qu'il faudra implanter avec l'algorithme étudié. Des implantations génériques fournies par défaut facilitent ce développement. En l'occurrence, il suffira de spécialiser `SimpleReconfigurator`, qui reconfigure le placement des VMs avec un *délai* configurable.
Tous les autres modules sont ceux par défaut, et ne nécessitent même pas d'être maîtrisés.
Le problème étudié est purement administrateur, seule compte la charge en terme de nombre de VMs et de CPU de ces VMs, peu importe les applications qui y sont exécutées. Nous allons donc utiliser l'application *LoadInjector*, dont la fonction principale est `loadinjector.SimpleInjection` et qui charge simplement un injecteur dans la simulation.

3.2. Description du contexte expérimental des simulations

La description du contexte expérimental des simulations se fait au travers de fichiers xml, respectant le format standard de SimGrid :

platform.xml décrit l'infrastructure matérielle : machines physiques et réseau.

deploy.xml décrit les processus.

cloud.xml décrit la plateforme de cloud : hôtes, types d'instance, et ordonnanceurs.

injector.xml décrit les injecteurs à utiliser.

Dans notre cas, `deploy.xml` est inutile, mais il faudra produire `platform.xml` et `cloud.xml`, ou utiliser les exemples fournis.
Nous allons comparer les résultats avec des délais de reconsolidation de 0, 10 et 100. Il faudra donc décliner le fichier de configuration du cloud pour ces trois cas : `cloud-reconsolidator0.xml`, `cloud-reconsolidator10.xml` et `cloud-reconsolidator100.xml`.
De plus, il faudra configurer l'injecteur, par exemple en utilisant `SinInjector` qui injecte une charge de calcul sinusoïdale sur un nombre de VMs suivant également une sinusoïde.

3.3. Description et exécution des simulations

L'exécution d'une simulation se fait avec la commande suivante :

```
$ java  loadinjector.SimpleInjection platform.xml deploy.xml cloud.xml
injector.xml
```

Toutes les expériences du *lab* sont guidées par un fichier de configuration (voir exemple ci-après). Celui-ci contient l'ensemble des informations nécessaires aux simulations, les variables à observer, et si nécessaire les pré- ou post- traitements requis.
Les lignes `TU_ARG` permettent de définir les observations que nous souhaitons faire, en l'occurrence le nombre de machines dont le nombre de cœurs utilisés n'est pas nul, ainsi que le nombre de VM en état de migration.
Les lignes `SIM_ARG` donnent les arguments des simulations à faire. En l'occurrence, le quatrième argument doit prendre successivement les trois fichiers de configuration correspondants au trois ordonnanceurs à comparer. Le lab exécute toutes les combinaisons d'arguments possibles, ainsi il est facile de démultiplier les simulations. Par exemple, l'ajout d'une ligne `SIM_ARG 5` permettrait de faire les mêmes simulations, mais avec deux injecteurs différents.
La commande `./lab.py -p 4 cmp-scheduler.cfg` permet d'exécuter notre expérience, avec 4 simulations en parallèle pour exploiter tous les cœurs d'une machine, grâce au fichier :

```
# setup
SETUP_DIR ./setup/cmp-scheduler
NEEDED_POST template.R
POST_COMMAND_DATA R -f template.R > R.out

# observations
TU_ARG --count-if used_cores ne 0
TU_ARG --count-if vm:.*:state eq migrating

# simulations
SIM_ARG 1 loadinjector.SimpleInjection
SIM_ARG 2 platform.xml
SIM_ARG 3 deploy.xml
SIM_ARG 4:reconsolidator0 cloud-reconsolidator10.xml
SIM_ARG 4:reconsolidator10 cloud-reconsolidator10.xml
SIM_ARG 4:reconsolidator100 cloud-reconsolidator100.xml
SIM_ARG 5 injector.xml
```

3.4. Analyse des résultats

A la fin des exécutions le *lab* extrait des traces de simulation les observations demandées et les stocke dans des fichiers séparés, puis exécute les commandes stipulées par les lignes POST_COMMAND. En l'occurrence, il s'agit d'un patron R, utilisant une librairie inclue dans le lab :

```
pdf('data.pdf')
dfs <- tu_read('.', plotting=TRUE, plotting_state=FALSE)
dev.off()
```

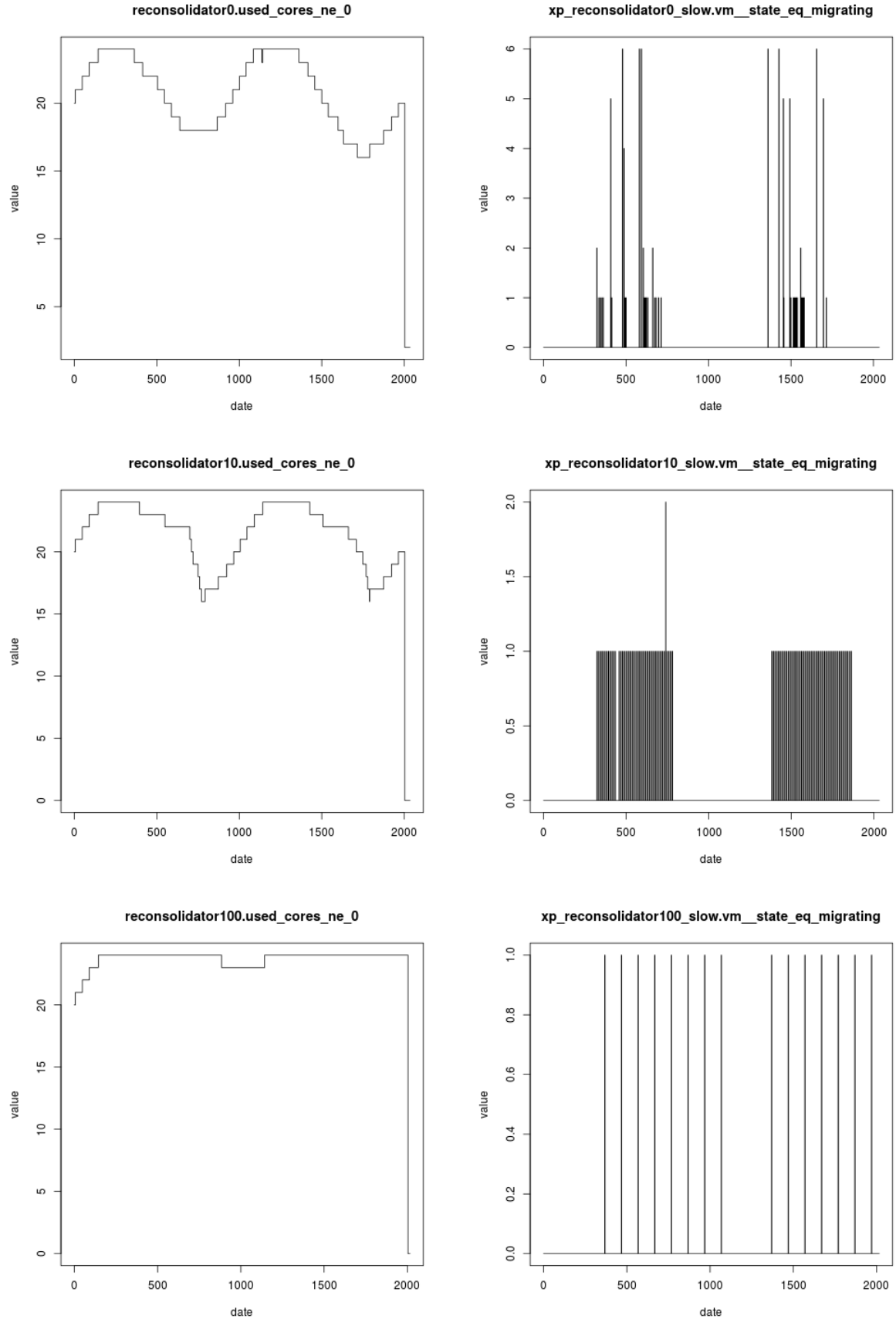
Ce dernier va automatiquement produire un graphique pour chaque observation demandée, comme montré Figure 1.

Ces six graphiques montrent le nombre de machines physiques totalement déchargées de ses VMs, puisque le nombre de cœurs utilisés est non nul, et qui peuvent donc être éteintes (à gauche) et le nombre de migrations concurrentes de VM (à droite) à chaque instant. Les trois lignes concernent des délais de reconsolidation différents : 0 (les VMs sont migrées dès que la charge de la plateforme change), 10s et 100s.

On peut ainsi observer qu'un intervalle de reconsolidation nul permet de décharger complètement des machines physiques mais au prix de nombreuses migrations concurrentes, dont il faut s'attendre à ce qu'elles perturbent grandement le fonctionnement de la plateforme. D'un autre côté, un intervalle de 100 secondes ne décharge pratiquement aucune machine, et rend donc la stratégie de migration inutile. En revanche, un intervalle de 10 secondes décharge raisonnablement les machines physiques, tout en évitant toute migration concurrente, et préservant ainsi les performances de la plateforme.

Il est important de remarquer également que ces graphiques ont les travers inévitables des données produites automatiquement : titres abscons, labels génériques, et échelles par défaut. L'utilisateur souhaitant raffiner ces données devra adapter le fichier `template.R`. En revanche, un très grand nombre de métriques est automatiquement journalisé et *le lab* dispose d'une expressivité suffisante pour tracer immédiatement une très grande game de valeurs. De plus, le système de journalisation est accessible et permet d'inclure dans le simulateur toute valeur définie par l'utilisateur, qui sera également automatiquement disponible comme toutes les autres.

FIGURE 1 – Graphiques produits automatiquement par le lab.



4. Conclusion

Le *lab* est un outil pour l'automatisation d'exécution d'expériences *in silico* qui permet d'exécuter facilement de multiples simulations. L'automatisation des simulations et des observations permet non seulement de s'assurer que rien n'échappe à l'expérimentateur, mais rend aussi l'expérience trivialement reproductible. Le *lab* est déjà utilisé au sein de notre équipe pour la validation de SCHIaaS, en simulant automatiquement l'ensemble d'une archive de 273 exécutions réelles, chacune dans 5 conditions expérimentales différentes, soit plus de 1300 simulations à observer. Nous l'utilisons également pour des simulations de Monte Carlo, en procédant à des lots de 500 simulations afin d'approcher l'évaluation des performances d'une application dont les temps d'exécution sont difficilement modélisables. Des études par la simulation de tels volumes sont pratiquement impossibles à mener sans un outil support à la méthodologie de l'expérimentation permettant une étude de bout-en-bout.

Bibliographie

1. Calheiros (R. N.), Ranjan (R.), Beloglazov (A.), Rose (C. A. F. D.) et Buyya (R.). – Cloudsim : a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw., Pract. Exper.*, vol. 41, n1, 2011, pp. 23–50.
2. Casanova (H.), Giersch (A.), Legrand (A.), Quinson (M.) et Suter (F.). – Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, vol. 74, n10, juin 2014, pp. 2899–2917.
3. Google. – Google cluster data, dec 2016.
4. Gossa (J.). – Schiaas documentation, `http://schiaas.gforge.inria.fr`, sep 2016.
5. Kliazovich (D.), Bouvry (P.) et Khan (S. U.). – Greencloud : a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, vol. 62, n3, 2012, pp. 1263–1283.
6. Michon (E.), Gossa (J.), Genaud (S.), Unbekandt (L.) et Kherbache (V.). – Schlouder : A broker for iaas clouds. *Future Generation Comp. Syst.*, vol. 69, 2017, pp. 11–23.
7. Nuñez (A.), Vázquez-Poletti (J. L.), Caminero (A. C.), Carretero (J.) et Llorente (I. M.). – Design of a new cloud computing simulation platform. – In *ICCSA (3)*, *Lecture Notes in Computer Science*, volume 6784, pp. 582–593. Springer, 2011.
8. Ostermann (S.), Prodan (R.) et Fahringer (T.). – Dynamic cloud provisioning for scientific grid workflows. – In *GRID*, pp. 97–104. IEEE Computer Society, 2010.