Exploitation des techniques de virtualisation pour l'administration autonome d'infrastructures logicielles réparties

Alain Tchana*, Suzy Temate*, Benoît Combemale**, Laurent Broto*** et Daniel Hagimont*

*IRIT, Université de Toulouse, France
prénom.nom@enseeiht.fr
**AtlanMod team – INRIA & École des Mines de Nantes, France
benoit.combemale@inria.fr
***Oak Ridge National Laboratory, United States
broto@irit.fr

Résumé. Les grappes de machines (ou clusters) sont souvent utilisées pour gérer une infrastructure de serveurs dupliqués, afin de répondre à des charges importantes et surtout variables. Un exemple est la gestion d'une architecture J2EE sur un cluster, implantant un serveur Web dynamique. Dans des travaux récents, nous avons exploré la conception d'un système autonome permettant d'ajuster dynamiquement le degré de duplication des serveurs en fonction de la charge. Cependant, la modification de l'infrastructure logicielle nécessite l'arrêt et/ou le redémarrage de serveurs, provoquant une perte de l'état d'exécution. Une alternative est d'exploiter les techniques de virtualisation de systèmes d'exploitation, permettant la migration d'un système sans perte de données. Les serveurs s'exécutant sur des systèmes virtualisés peuvent ainsi être répartis sur un ensemble de noeuds physiques variable en fonction de la charge et sans perte d'état. Nous décrivons l'implantation de cette approche et son évaluation par rapport à la précédente.

1 Introduction

Les environnements informatiques d'aujourd'hui sont de plus en plus sophistiqués. Ils intègrent de nombreux logiciels complexes qui coopèrent dans le cadre d'une infrastructure logicielle, potentiellement à grande échelle. Ces logiciels se caractérisent par une grande hétérogénéité, en particulier en ce qui concerne leurs fonctions d'administration qui sont le plus souvent propriétaires. La conséquence est que l'administration de ces infrastructures logicielles (installation, configuration, réparation, optimisation ...) est une tâche très complexe, source d'erreurs, et consommatrice en ressources humaines.

Une approche très prometteuse consiste à implanter un logiciel d'administration autonome. Ce logiciel fournit un support pour le déploiement et la configuration des applications dans un environnement réparti. Il fournit également un support pour la supervision de l'environnement administré et permet de définir des réactions face à des événements comme des pannes ou des

surcharges, afin de (re)configurer les applications administrées de façon autonome. C'est dans ce contexte qu'a été conçu et développé le système TUNe (*Toulouse University Network*), permettant de résoudre ce problème d'administration d'infrastructures logicielles réparties (Broto et al., 2008).

TUNe a été utilisé pour administrer différentes applications déployées sur des grappes de machines ou des grilles de calcul. En particulier, il a été utilisé pour la gestion d'une infrastructure J2EE en grappe. Dans ce type d'application, les serveurs déployés sont souvent dupliqués, afin de répondre à des charges importantes et surtout variables. TUNe a été utilisé pour assurer le déploiement de ces applications, ainsi que pour ajuster dynamiquement le degré de duplication des serveurs en fonction de la charge. L'ajustement se traduit par l'ajout ou le retrait automatique de serveurs.

Cependant, la modification de l'infrastructure logicielle nécessite l'arrêt et/ou le redémarrage de serveurs, provoquant une perte de l'état d'exécution. En effet, TUNe administre des logiciels patrimoniaux qui n'offrent pas forcément (et généralement pas) des fonctions permettant de capturer l'état interne du logiciel. Les machines virtuelles, e.g., Xen (Barham et al., 2003), offrent des facultés de migration et de capture d'état. Celles-ci ouvrent de nouvelles perspectives pour la résolution de ce problème de duplication.

L'objectif de cet article est de montrer comment les machines virtuelles sont utilisées dans TUNe pour résoudre le problème de duplication posé plus haut. L'article est structuré de la façon suivante. La section 2 présente le contexte et nos motivations, la section 3 présente l'approche que nous proposons et la section 4 présente les résultats des expériences réalisées. La section 5 présente les différents travaux relatifs et nous concluons enfin, dans la section 6, en présentant nos différentes perspectives.

2 Contexte et motivation

Nous introduisons tout d'abord le domaine de l'administration autonome, avec une présentation rapide de TUNe. Ensuite, nous présentons les architectures J2EE (Bodoff et al., 2002), qui constituent l'exemple applicatif auquel nous nous référons tout au long de cet article. Nous finissons par une présentation de nos motivations.

2.1 Interface d'administration de haut niveau : TUNe

L'idée de concevoir des systèmes d'auto-administration a été motivée par l'observation d'une croissance continue de la complexité des infrastructures et environnements logiciels. Ces systèmes autonomes réalisent les tâches d'administration qui auparavant étaient effectuées par des humains. Ils apportent une solution au problème que rencontrent les administrateurs dans leur tâche quotidienne à savoir : des erreurs (lors de la configuration des logiciels par exemple), l'estimation des ressources à allouer aux applications (ce qui entraine une sur-réservation et un gaspillage), la lenteur dans l'exécution des tâches.

Différentes technologies sont généralement exploitées lors de la conception d'un système d'administration autonome. L'une des technologies la plus utilisée est celle basée sur les modèles à composants. L'utilisation d'un modèle à composants permet de faciliter l'administration des logiciels en les réifiant de manière homogène. En effet, si chaque composant encapsule

un logiciel ou un morceau de logiciel et qu'il existe une logique pour administrer ces composants, il est alors aisé d'administrer n'importe quel logiciel.

Le système TUNe (Broto et al., 2008), développé au sein du laboratoire IRIT[1], est basé sur le modèle à composants Fractal (Bruneton et al., 2006). Tout élément logiciel administré par TUNe est encapsulé dans un composant Fractal. Cette approche permet de profiter des interfaces de reconfiguration qu'offre le modèle Fractal. En effet, chaque composant Fractal fournit une interface permettant son administration locale et des interfaces de contrôle qui permettent d'administrer les attributs des composants et de relier ces derniers entre eux. Dans TUNe, la description de l'architecture, du déploiement et des politiques de reconfiguration du système à administrer est effectuée à travers des profils UML (Object Management Group, Inc., 2007, §18). L'introduction de UML dans TUNe permet aux administrateurs de se passer des connaissances très spécifiques au modèle à composants Fractal. En effet, dans la plupart des systèmes d'administration autonome, la spécification des politiques de reconfiguration nécessite une programmation de l'administrateur selon l'API du modèle à composants. Dans le système TUNe, des diagrammes de classes sont utilisés pour décrire l'architecture de l'infrastructure logicielle à déployer, et des diagrammes d'état sont utilisés pour la description des politiques de reconfiguration. Une analyse de l'utilisation d'UML pour la définition de politiques d'administration autonome est donnée dans Combemale et al. (2008).

Le déclenchement d'une procédure de reconfiguration est initié par l'envoi d'une notification au système TUNe. Le monitoring est réalisé par des sondes qui scrutent périodiquement l'état de l'infrastructure logicielle et génère des notifications. Parmi les tâches d'administration qu'assure TUNe, nous nous interessons dans cet article à l'optimisation de l'usage des ressources (principalement le CPU) en modifiant le degré de duplication des serveurs dans une infrastructure de serveurs dupliqués. La section suivante présente un exemple applicatif concerné par une telle administration autonome.

2.2 Contexte applicatif : J2EE

Les spécifications de la plate-forme Java 2 Platform, Enterprise Edition (J2EE) définissent un modèle pour le développement d'applications Web selon une architecture multi-tiers. Ces applications sont composées d'un serveur Web (e.g., Apache[2]), un serveur d'application (e.g., Tomcat[3]) et un serveur de base de données (e.g., MySQL (Widenius et Axmark, 2002)). Une requête HTTP au serveur Web référence soit un contenu statique qui est retourné directement au client par le serveur Web, soit un document qui doit être généré dynamiquement, la requête étant alors transmise au serveur d'application. Quand le serveur d'application reçoit une requête, il déclenche l'exécution de traitements matérialisés par des Servlets qui peuvent interroger la base de données stockant les données persistantes. Le résultat de ces traitements est la génération d'une page Web qui est retournée au client. Dans ce contexte, la croissance exponentielle du nombre d'usagers de l'Internet a fait naître le besoin de concevoir des applications Web passant à l'échelle et hautement disponibles. Pour passer à l'échelle et assurer la disponibilité, une approche très utilisée consiste à dupliquer les serveurs composant une architecture J2EE sur une grappe de machines. Dans cette approche (cf. figure 1), un composant

[1] Institut de Recherche en Informatique de Toulouse

[2] *The Apache HTTP Server Project*, cf. `http://httpd.apache.org/`

[3] *The Apache Tomcat Project*, cf. `http://tomcat.apache.org`

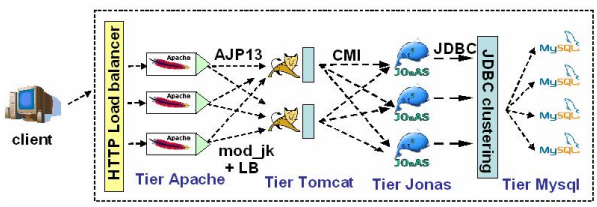

FIG. 1 – *Architecture J2EE*

particulier que nous appelons un répartiteur (ou *Load Balancer* en anglais), est installé devant chaque groupe de serveur dupliqué et répartit la charge de travail entre les duplicas. Différents algorithmes de répartition de la charge peuvent être implantés par le répartiteur, par exemple, une répartition aléatoire (*random*) ou tourniquet (*round-robin*). D'une façon très synthétisée, l'élément C-JDBC (Cecchet et al., 2004) permet la duplication de serveurs de base de données, mod_jk[4] permet la duplication de serveurs de servlet (e.g., Tomcat) à travers le protocole AJP13 (Berg, 2008) et L4 (Ok et Park, 2006) permet la duplication de serveurs web Apache.

2.3 Duplication autonome avec TUNe

En plus des opérations classiques de reconfiguration (e.g., démarrage ou arrêt), TUNe fournit à l'administrateur des opérations permettant d'augmenter ou de diminuer le degré de duplication d'un élément de l'architecture. L'ajout ou le retrait d'un duplicas met à jour les liaisons dans l'architecture à composants gérée.

Lorsqu'une sonde observe une montée en charge sur un composant logiciel qu'elle surveille, elle envoie une notification à TUNe. Celui-ci peut déclencher automatiquement l'exécution d'une procédure de reconfiguration qui consiste à ajouter une nouvelle instance de ce logiciel (dans le but d'absorber la charge observée). Inversement, lorsqu'une baisse de la charge est observée par une sonde, elle peut déclencher une reconfiguration qui consiste à retirer une instance du logiciel moins chargée.

En général, les procédures de reconfiguration qui permettent de réaliser une duplication contiennent les actions suivantes :

- Arrêter la sonde qui a déclenché la reconfiguration du logiciel qu'elle surveille. Cette action permet d'éviter que la même sonde continue d'envoyer la notification pendant le traitement de la première notification.
- Allouer un nouveau noeud (uniquement pour l'ajout) : TUNe choisit un noeud, dans l'ensemble des noeuds disponibles, pour la nouvelle instance.
- Créer la nouvelle instance (dans le cas de l'ajout) : TUNe configure, déploie et démarre le nouveau logiciel.
- Arrêter et supprimer le composant, libérer le noeud (uniquement dans le cas du retrait),
- Arrêter, reconfigurer et redémarrer les composants dépendants de l'instance ajoutée ou supprimée, pour qu'ils prennent en compte la présence d'un nouveau composant ou le retrait d'un composant. Ceci est souvent nécessaire pour la prise en compte des liaisons

[4] *The Apache Tomcat Connector*, cf. `http://tomcat.apache.org/connectors-doc/`

mises à jour par la reconfiguration. Par exemple, si on ajoute ou supprime un serveur Tomcat de notre architecture J2EE, il faut reconfigurer et redémarrer le serveur Apache qui référence ce Tomcat.

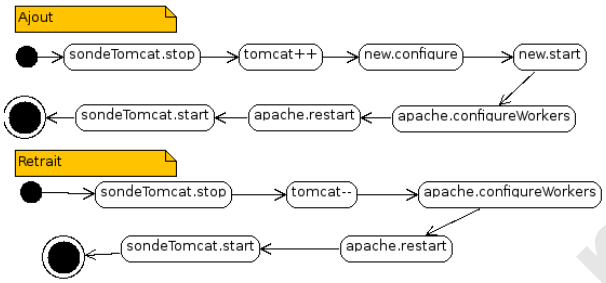

FIG. 2 – *Procédure de duplication de Tomcat*

La figure 2 présente le diagramme d'état correspondant à l'ajout et le retrait d'une instance de Tomcat. L'opération $tomcat++$ représente l'allocation d'un noeud et le déploiement de la nouvelle instance de Tomcat sur ce noeud. Le mot-clé *new* désigne la nouvelle instance créé. L'action $tomcat--$, retire une instance de Tomcat d'un noeud. Lors d'un ajout ou retrait de Tomcat, l'Apache auquel ils sont liés doit être mis à jour, ce qui est fait par les actions *apache.configureWorkers* et *apache.restart* qui respectivement provoquent la mise à jour des fichiers de configuration d'Apache et son redémarrage.

2.4 Motivation

Après cette présentation de l'approche qu'utilise TUNe pour résoudre le problème d'optimisation de ressources, nous constatons que cette approche ne prend pas en compte l'état interne du logiciel supprimé (dans le cas d'un retrait) ou de l'état des instances existantes (dans le cas d'un ajout). En effet, dans l'exemple de J2EE, le retrait d'un serveur Tomcat entraîne la perte des requêtes en cours de traitement par ce serveur. De même, lors de l'ajout d'un duplicas de Tomcat, les sessions des servlets étant dupliquées et maintenues en cohérence entre les duplicas de Tomcat[5], il faut remettre en cohérence l'instance ajoutée. Enfin, lorsque Apache est redémarré, des requêtes peuvent être également perdues. Ces différents problèmes ne sont pas traités dans l'approche de TUNe présentée précédemment. C'est pourquoi, la modification du degré de duplication d'un serveur dupliqué nécessite souvent un redémarrage des clients de ce serveur dupliqué pour prendre en compte la modification au niveau patrimonial.

Cependant, on peut se poser la question de la difficulté d'adapter cette solution pour résoudre le problème de gestion de l'état à l'exécution. Si les logiciels utilisés fournissent des interfaces permettant de capturer leur état interne, une solution est envisageable en faisant remonter ces interfaces au niveau des outils de TUNe, mais ce n'est généralement pas le cas pour la plupart des applications à administrer. Modifier une application dans ce sens est beaucoup plus difficile, et seulement possible si l'on dispose des sources de l'application.

[5] car deux appels successifs provenant d'un client (dans la même session) peuvent arriver sur un duplicas différent de Tomcat

Face à ces limites (incohérence d'état et perte de requêtes), nous proposons dans cet article une approche basée sur la virtualisation. La section suivante présente cette approche.

3 Contribution

3.1 Principe de l'utilisation des machines virtuelles

L'approche que nous exploitons dans cet article est basée sur la virtualisation (Barham et al., 2003). La virtualisation est une technologie qui permet de démarrer au dessus d'un système d'exploitation de base (système hôte), plusieurs autres systèmes d'exploitation (systèmes invités ou machines virtuelles) comme toute autre application ou serveur. Il existe plusieurs technologies de virtualisation parmi lesquelles la paravirtualisation ou systèmes à hyperviseurs. La paravirtualisation est une technique de virtualisation qui consiste à modifier à la fois le système hôte (l'hyperviseur) et le système virtualisé afin d'améliorer les performance (contrairement au technique classiques de virtualisation). C'est cette dernière technique que nous exploitons dans nos travaux à travers le système Xen qui en est une implémentation.

L'idée de notre approche est d'exploiter les possibilités qu'offrent les machines virtuelles (e.g., *chekpointing*, migration) afin de résoudre les problèmes évoqués dans la section précédente. Au lieu d'effectuer un équilibrage de charge par modification du degré de duplication (ajout et suppression), nous utilisons plutôt la possibilité de migration des machines virtuelles. Dans cette nouvelle approche, chaque instance de logiciel tourne sur une machine virtuelle. L'administrateur effectue initialement une estimation, pour chaque logiciel géré par TUNe, du nombre d'instances maximales dont aura besoin son système dans le pire des cas (c-à-d lorsqu'il est en plein régime). Toutes ces instances sont créées et déployées chacune sur une machine virtuelle différente. Les machines virtuelles sont initialement déployées sur un nombre de machines physiques pouvant supporter la charge induite par ces machines virtuelles.

Ainsi, la fonction d'ajout d'instance (en cas de saturation) dans l'approche précédente correspond ici à une migration de la machine virtuelle hébergeant une instance du serveur surchargé vers une machine physique moins chargée. Le retrait d'une instance quand à lui, correspond à un rapatriement de la machine virtuelle (initialement migrée) vers la machine physique initiale. Nous illustrons ce schéma sur la figure 3. Toutes les instances logicielles (C1 et C2 sur la figure) sont initialement déployées et démarrées sur des machines virtuelles différentes (sur la même machine physique). Ainsi, il n'est plus question, pendant le fonctionnement du système, d'ajouter ou de retirer une instance logicielle. En fonction de la variation de la charge d'une instance, celle ci est déplacée vers une autre machine physique moins chargée (par migration de la machine virtuelle sur laquelle elle se trouve).

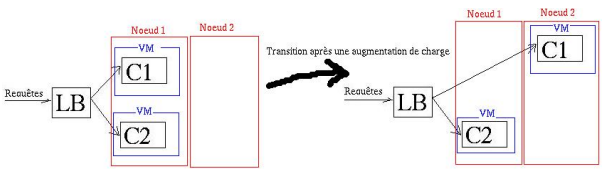

FIG. 3 – *Nouvelle approche de duplication (lors d'une augmentation de charge)*

3.2 Implantation dans TUNe

L'implantation de cette approche dans TUNe s'effectue en deux étapes. Dans un premier temps, on utilise TUNe pour déployer des machines virtuelles et ensuite, tous les logiciels à administrer par TUNe utilisent les machines virtuelles (domU dans le jargon de Xen) dans leurs politiques d'allocation de machines. Nous avons donc dans cette nouvelle approche deux niveaux d'administration : celui des machines virtuelles et celui des logiciels.

Dans ces deux niveaux, nous utilisons TUNe pour encapsuler les entités administrées (respectivement les machines virtuelles et les logiciels) dans des composants Fractal. La seule particularité de cette implantation se situe au niveau de l'allocation de noeud. Nous avons développé une nouvelle politique basée sur l'état du noeud (en terme de CPU) qui consiste à choisir le noeud le moins chargé parmi les noeuds déjà alloués (la politique précédente ne gérait que des réservations/libérations de noeuds). Nous avons également implanté une nouvelle fonction d'administration sur les composants permettant de déclencher une migration de composant.

Une sonde surveille l'état des machines physiques et virtuelles en terme de CPU. Lorsque la charge CPU d'une machine virtuelle dépasse un seuil (spécifié à TUNe par un attribut de la machine virtuelle), la sonde demande à TUNe une migration à chaud (*live migration*) de la machine virtuelle vers une machine hôte moins chargée. La figure 4 montre les différentes actions effectuées par TUNe lorsqu'il reçoit un évènement de migration. TUNe vérifie d'abord si la machine initiale de la machine virtuelle peut la recevoir (c-à-d sa charge CPU est en dessous du seuil minimal fixé). Si tel est le cas, cette machine est utilisée comme destination pour la migration. Sinon, la machine ayant la charge CPU la plus basse est choisie comme destination de la migration.

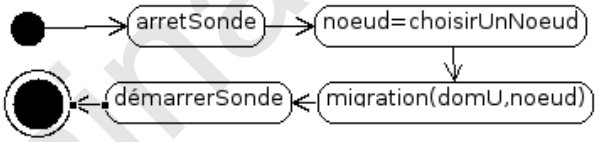

FIG. 4 – *Procédure de migration*

3.3 Impact sur l'architecture de TUNe

Dans la version précédente qui n'utilisait pas de machine virtuelle, les serveurs dupliqués étaient ajoutés sur de nouvelles machines ou retirés, en fonction de la charge. Dans la version reposant sur des machines virtuelles, les serveurs dupliqués sont en nombre constant, mais ils sont migrés en fonction de la charge afin de libérer des machines lorsque le système est sous-chargé.

Afin de comparer ces deux solutions, nous pouvons comparer l'architecture des deux solutions dans deux cas de figure : lorsque le système est en sous-charge et lorsque le système est chargé. La figure 5 présente cette comparaison.

Sans machines virtuelles, lorsque le système est sous-chargé (en haut à gauche), le répartiteur de charge (LB) envoie les requêtes à un seul serveur, qui est suffisant pour répondre à

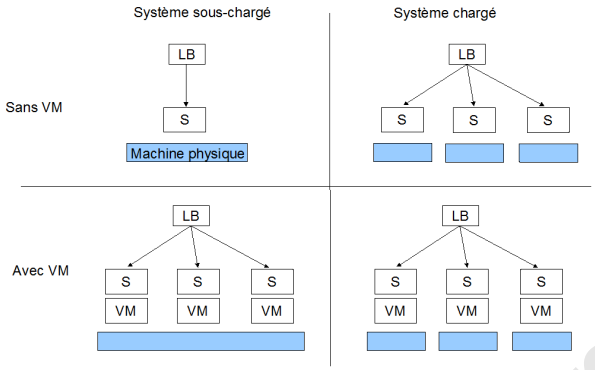

FIG. 5 – *Comparaison des deux approches*

une faible charge. Lorsque le système est chargé (en haut à droite), TUNe adapte le degré de duplication du serveur, en allouant des machines physiques supplémentaires et en déployant des serveurs sur ces machines.

En utilisant des machines virtuelles, les différents serveurs sont déployés sur des machines virtuelles (dans une quantité fixée au démarrage). Lorsque le système est sous-chargé (en bas à gauche), les machines virtuelles s'exécutent toutes sur la même machine physique (et les autres machines physiques sont libérées). Lorsque le système est chargé (en bas à droite), TUNe migre les machines virtuelles sur des machines physiques différentes allouées dynamiquement.

4 Validation expérimentale

Dans le but de valider l'approche décrite dans cet article, plusieurs expériences ont été conduites. Avant de présenter les résultats expérimentaux, nous présentons les méthodes utilisées et le contexte expérimental.

4.1 Méthodes d'expérimentation

Nous retenons deux critères pour cette évaluation : le nombre de requêtes perdues et le surcoût des machines virtuelles en terme de CPU des machines physiques. Les résultats que nous donnons à chaque fois représentent la moyenne des résultats obtenus pour le même scénario effectué cinq fois de suite. Nous comparons les résultats obtenus dans le cas des deux approches (TUNe sans et avec machines virtuelles).

Nous utilisons le benchmark RUBIS (Amza et al., 2002) qui implante une application J2EE similaire à *eBay.com*. RUBIS est fourni avec un simulateur de client Web. Ce simulateur génère une charge sur le serveur Apache, entrainant ainsi une charge sur tous les autres composants logiciels du système (i.e., Tomcat et MYSQL). Nous avons apporté des modifications au simulateur de clients RUBIS afin de pouvoir comptabiliser les requêtes non exécutées par Apache (lorsqu'il est arrêté). On se sert des fichiers de logs de Apache (notament le fichier error_log et access_log) afin de comptabiliser les requêtes non traitées par une instance de tomcat (dû

à une interruption de celui ci). Nous récupérons, pendant l'exécution de chaque scénario, la charge CPU de la machine hébergeant les serveurs Tomcat (toutes les secondes). Ces charges sont par la suite moyennées. Pour les systèmes virtualisés, nous utilisons l'outil de mesure de charge CPU *xentop*. C'est un outil fournit par la communauté Xen. Pour les systèmes Linux natif, nous récupérons la charge CPU à travers le fichier */proc/stat*. Les débits présentés ici proviennent des statistiques fournies par les clients Rubis à la fin de chaque expérience.

4.2 Contexte expérimental

Ces expériences sont réalisées dans l'environnement Grid'5000 (Bolze et al., 2006). Nous utilisons treize machines physiques et quatre machines virtuelles. Les machines physiques sont des IBM eServer 325, avec des processeurs AMD opteron 2462 GHz et 2Go de memoire. Le système hôte sur les machines est une distribution de Linux Debian « Xenifiée »(i.e., instrumenté avec le noyau Xen) avec la version 3.1.0 de Xen. C'est cette image qui est déployée par la suite sur les machines virtuelles. Toutes les machines sont connectées au serveur NFS du site et montent un répertoire partagé. Nous nous intéressons ici au serveur Tomcat. Ainsi, dans l'expérimentation de la nouvelle approche, c'est uniquement les instances du logiciel Tomcat qui tournent sur les machines virtuelles.

4.3 Résultats des expériences

4.3.1 Limite de notre solution

Cette section a pour but d'observer le comportement du système sans virtualisation (ancienne approche) et avec l'introduction de la virtualisation (nouvelle approche). Il s'agit de mesurer le surcoût dû aux machines virtuelles. En effet, lorsque le système est soumis à une charge très légère, dans l'ancienne approche (sans machine virtuelle), la configuration du système se compose d'un unique Tomcat s'exécutant sur un système natif. Par contre, dans la nouvelle approche (avec machines virtuelles), la configuration du système se compose d'un nombre de machines virtuelles co-localisées sur la même machine physique. Ce nombre de machines virtuelles est fixé par l'administrateur au déploiement du système.

Nous utilisons pour ce scénario des machines bi-processeurs. C'est la raison pour laquelle, nous avons des charges CPU allant de 0 à 200 (sur la figure 6). La charge soumise est la même pour toutes les expériences. Nous soumettons notre système à un benchmark de 600 clients Rubis d'une durée de 90 secondes. Nous effectuons trois expériences avec le nombre de machines virtuelles variant de 0 à 3. Nous nous intéressons ici à l'évolution de la charge CPU et du débit en fonction du nombre de machines virtuelles. La figure 6 montre l'évolution de la charge CPU entre un système Linux natif et le même système avec virtualisation (de 1 à 3 machines virtuelles). Chaque machine virtuelle héberge un serveur Tomcat.

Description de la courbe :

- à t=10 (le premier pic sur les courbes) : déploiement des logiciels par TUNe sur les différentes machines (machines physiques et machines virtuelles).
- de 10 à 40 : démarrage progressif des clients Rubis, ce qui explique le deuxième pic observé sur les différentes courbes. Sur la figure, l'ajout des 4 machines virtuelles entraîne un surcoût de 7% environ (de 40/200 à 55/200).

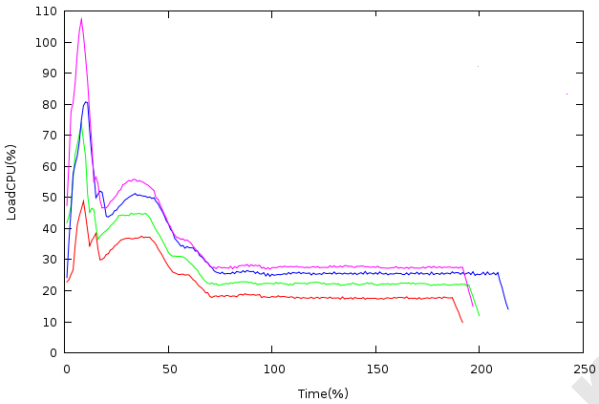

FIG. 6 – *Charge CPU en fonction du nombre de VMs*

– de 40 à 90 : arrêt progressif des clients Rubis, ce qui entraîne une baisse de la consommation CPU ;
– de 90 à la fin : la charge devient nulle et il ne reste que des tâches de fond sur le système (notamment les processus de TUNe).

Ainsi, nous constatons que le débit est pratiquement équivalent pour les deux approches.

4.3.2 Avantages de notre approche

Intéressons nous maintenant aux pertes de requêtes observées lors de la duplication dans TUNe et comparons ces résultats avec ceux obtenus avec l'approche basée sur les machines virtuelles. On constate qu'avec l'introduction des machines virtuelles, on n'a aucune perte de requêtes. Par ailleurs, sans les machines virtuelles, le retrait d'une instance entraine une perte de requêtes tant entre le client web et Apache qu'entre Apache et Tomcat. Ceci s'explique (perte entre Apache et Tomcat) par le fait que Apache n'a pas de retour des requêtes en cours d'exécution dans l'instance de Tomcat retirée, et (perte entre le client et Apache) par le redémarrage d'Apache.

Du point de vue des clients, il est inacceptable de se retrouver avec un ensemble de requêtes non traitées. C'est ici que cette approche prend toute son importance (malgré le surcoût CPU qui est introduit). Les tables 1) et 2) présente les pertes de requêtes lorsque l'application est soumise à un benchmack constant de 600 clients Rubis durant 5 minutes et que les machines virtuelles ne sont pas utilisées.

Le surcoût induit par l'exécution locale des machines virtuelles est modéré. Par contre, leur utilisation permet de perdre aucune requête lors de migration.

	CW-A	A-T	Total
exp 1	274	0	274
exp 2	267	0	267
exp 3	329	0	329
exp 4	195	0	195

TAB. 1 – *Cas de l'ajout d'un serveur Tomcat*

	CW-A	A-T	Total
exp 1	230	2	232
exp 2	257	2	257
exp 3	361	4	365
exp 4	276	3	278

TAB. 2 – *Cas du retrait d'un serveur Tomcat*

4.3.3 Dimensionnement

Le but dans ce scénario est de montrer comment TUNe réagit dans la nouvelle approche lorsque la charge CPU augmente sur la machine hébergeant les serveurs Tomcat initialement déployés. TUNe a été configuré pour réaliser le scénario suivant :

- à t=400, TUNe déploie les 2 machines virtuelles sur le même noeud physique ;
- à t=750, TUNe déploie un Tomcat sur chaque machine virtuelle ;
- de t=1100 à t=1500, augmentation du nombre de clients ;
- de t=1500 à 3500, maintient de la charge ;
- de t=3500 à 5000, diminution de la charge en diminuant le nombre de clients.

Chaque noeud physique possède une sonde (de charge CPU) déployée par TUNe qui provoque une allocation de noeud puis la migration d'une machine lorsque la charge dépasse 45% et qui provoque une migration de Tomcat puis une libération du noeud lorsque la charge est en dessous de 10%.

Comme prévu par le scénario, à t=1100, la charge commence à augmenter. La figure 7 donne la charge de la première machine, celle où sont lancés initialement les machines virtuelles et les Tomcats.

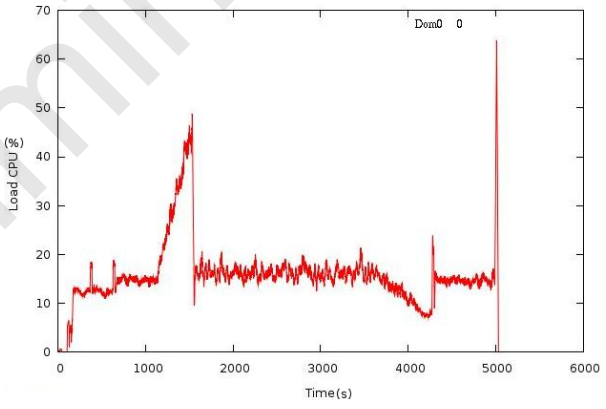

FIG. 7 – *Etat CPU du premier noeud*

Les deux pics visibles à t=400 et t=750 correspondent respectivement au lancement des machines virtuelles et des Tomcats. On peut donc constater que la charge monte à partir de

t=1100 pour chuter brutalement à t=1500. En effet, la charge a dépassé 45% et la migration d'une machine virtuelle a été réalisée. On observe cette migration sur la courbe montrant la charge de la deuxième machine physique (cf. figure 8). A t=1500, la charge de la deuxième machine physique augmente brutalement à cause de la création de la machine virtuelle qui migre (avec le serveur Tomcat). La charge se stabilise alors sur la première machine physique et la deuxième machine physique jusqu'à t=3500, instant où le nombre de clients diminue. La charge sur les deux machines physiques diminue donc jusqu'à t=4300 où la charge sur la deuxième machine physique passe en dessous de 10%. Une migration inverse (rapatriement) est alors effectuée, ce qui fait remonter la charge de la première machine. La charge de cette dernière se stabilise alors jusqu'à t=5000 qui est la fin de l'expérience. Le dernier pic sur la figure 7 (à t=5000) correspond à l'arrêt de tous les processus d'administration liés à TUNe.

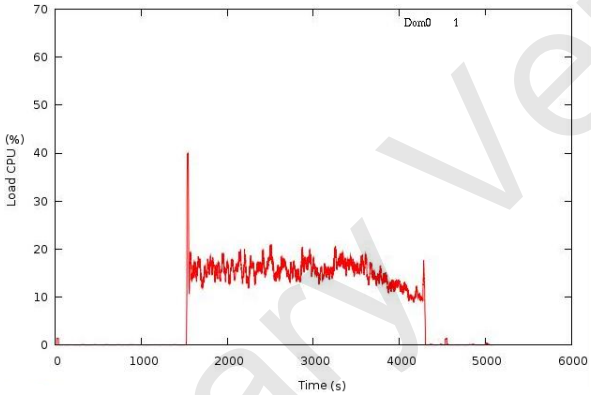

FIG. 8 – *Etat CPU du deuxième noeud*

Synthèse Au vu de ces résultats, nous constatons qu'avec l'introduction des machines virtuelles, nous réduisons à zéro le nombre de requêtes perdues pendant les opérations de duplication de logiciels. De plus, la virtualisation n'affecte pas le débit de traitement des requêtes qui reste équivalent à celui d'un système non virtualisé. Cependant, on observe un surcoût de la charge CPU d'environ 7% (lorsque le *benchmark* est au maximum) dûs à l'introduction de quatre machines virtuelles et une différence de 3% environ lorsque le *benchmark* est bas. Ce surcoût peut être négliger par rapport à une exécution sans perte de requête apportée par l'approche.

5 Travaux relatifs

Face à la croissance sans cesse de la complexité des infrastructures logicielles distribuées, plusieurs travaux ont été consacrés ces dernières années à l'aspect *administration* de ces infrastructures. Un des problèmes principaux de l'administration d'infrastructures logicielles est la gestion des ressources.

De nombreux travaux reposent sur un contrôle avancé de la répartition du temps CPU pour fournir des garanties sur les allocations de ressources (voir Cluster Reserves (Aron et al., 2000) et Sharc (Urgaonkar et Shenoy, 2004)). Dans ces systèmes, des serveurs sont dupliquées et la répartition des requêtes sur les serveurs est faite en prenant en compte la charge observée des serveurs. On remarque dans ces solutions que l'objectif principal est la gestion des ressources de calcul, mais elles entraînent un gaspillage de ressources matérielles du fait que les duplicas doivent exister tout au long du cycle de vie du système.

D'autres solutions consistent à concevoir et implanter un système d'admninistration autonome, capable d'ajouter et retirer des machines dans un ensemble de serveurs dupliqués. L'avantage est de libérer les machines lorsque le système est en sous-charge. A noter que la libération des machines permet à la fois de les réutiliser pour d'autres tâches, voir de les éteindre pour économiser l'énergie. Les projets TUNe (Broto et al., 2008), JADE (Bouchenak et al., 2005) et QuID (Ranjan et al., 2002) s'insèrent dans cette catégorie.

Cependant, l'ajout ou le retrait de serveurs pose un problème de cohérence des serveurs dupliqués. Dans de nombreux cas, l'ajout d'un duplicas de serveur nécessite d'accéder à l'état interne des serveurs. Par exemple lorsque le serveur de servlet Tomcat est dupliqué, les données de session des servlets sont dupliquées et l'ajout d'un duplicat nécessite de mettre en cohérence ces données de session dans le serveur ajouté. Un autre inconvénient est le fait d'arrêter momentanément l'application (ou une partie de l'application) afin de la reconfigurer, ce qui provoque des perturbations dans l'application. Par exemple, dans le cas d'une infrastructure J2EE, le redémarrage d'un serveur Apache pour qu'il prenne en compte la modification des serveurs Tomcat qui lui sont associés, provoque des erreurs perceptibles pour les clients de l'application Web.

L'utilisation de machines virtuelles que nous avons intégrée dans TUNe permet de résoudre ce problème. En effet, nous n'ajoutons pas de duplicas à l'infrastructure, mais nous ajoutons ou retirons des machines physiques à l'infrastructure par migration des machines virtuelles. L'inconvénient de cette approche est qu'elle impose de fixer au démarrage de nombre maximal de duplicas de serveurs que l'on lance sur des machines virtuelles.

6 Conclusion et perspectives

Face à la croissance de la complexité des environnements informatiques, la communauté scientifique a réagi en proposant des solutions innovantes pour la résolution du problème d'administration dans ces environnements. TUNe est un exemple d'implantation de système d'auto-administration de systèmes distribués. Néanmoins, TUNe connaissait certaines limites en ce qui concerne les opérations d'administration telles que la duplication, le dimensionnement ou la migration automatique de composants tout en conservant son état global. Dans cet article, nous exploitons la faculté de migration qu'offrent les machines virtuelles afin de résoudre ces problèmes. Les expérimentations ont montré que l'utilisation des machines virtuelles entraîne une baisse de performance modérée lorsque le système est en sous charge et que les machines virtuelles sont co-localisées sur une seule machine physique. TUNe arrive ainsi à maintenir le niveau de performance des applications tout en garantissant leur cohérence et aucune perte de requête.

Une des perspectives qu'ouvre ce travail porte sur la reconfiguration du répartiteur de charge. Après la migration d'une machine virtuelle, il doit répartir les charges de telle sorte

que la machine virtuelle migré reçoive plus de requêtes que celles restées sur le noeud initial. D'autre part, après le rapatriement d'une machine virtuelle vers le noeud initial (i.e., noeud de rapatriement), le répartiteur de charge doit équitablement répartir les requêtes sur les différentes machines virtuelles.

Une autre piste d'amélioration de ces travaux est d'étudier leurs applications aux systèmes sur les grilles de calcul développées avec la librairie MPI (Foster et Karonis, 1998). Ces applications parallèles prennent la forme de processus inter-opérant par échanges de messages. Elles ne suivent donc pas le schéma de répartition de charge présenté dans cet article. Il est alors intéressant d'étudier une solution consistant à co-localiser ou repartir les processus MPI en fonction de leurs utilisations de ressources CPU.

Références

Amza, C., E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, et W. Zwaenepoel (2002). Specification and Implementation of Dynamic Web Site Benchmarks. In *EEE 5th Annual Workshop on Workload Characterization (WWC-5). Austin, TX*.

Aron, M., P. Druschel, et W. Zwaenepoel (2000). Cluster reserves : a mechanism for resource management in cluster-based network servers. In *SIGMETRICS '00 : Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, pp. 90–101. ACM.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, et A. Warfield (2003). Xen and the art of virtualization. In *ACM symposium on Operating systems principles (SOSP'03)*, New York, NY, USA, pp. 164–177. ACM.

Berg, A. (2008). Separate the static from the dynamic with Tomcat and Apache. *Linux J. 2008*(165), 9.

Bodoff, S., D. Green, K. Haase, E. Jendrock, M. Pawlan, et B. Stearns (2002). *The J2EE tutorial*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Bolze, R., F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, et I. Touche (2006). Grid'5000 : A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *Int. J. High Perform. Comput. Appl. 20*(4), 481–494.

Bouchenak, S., F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, S. Jean-Bernard, N. de Palma, et V. Quema (2005). Architecture-based autonomous repair management : An application to j2ee clusters. In *SRDS '05 : Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, Washington, DC, USA, pp. 13–24. IEEE Computer Society.

Broto, L., D. Hagimont, P. Stolf, N. Depalma, et S. Temate (2008). Autonomic management policy specification in Tune. In *SAC '08 : Proceedings of the 2008 ACM symposium on Applied computing*, New York, NY, USA, pp. 1658–1663. ACM.

Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma, et J.-B. Stefani (2006). The FRACTAL component model and its support in Java : Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper. 36*(11-12), 1257–1284.

Cecchet, E., J. Marguerite, et W. Zwaenepole (2004). C-JDBC : flexible database clustering middleware. In *ATEC '04 : Proceedings of the annual conference on USENIX Annual Tech-*

nical Conference, Berkeley, CA, USA, pp. 26–26. USENIX Association.

Combemale, B., L. Broto, X. Crégut, M. Daydé, et D. Hagimont (2008). Autonomic Management Policy Specification : From UML to DSML. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, et M. Völter (Eds.), *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, Volume 5301 of *LNCS*, pp. 584–599. Springer.

Foster, I. et N. T. Karonis (1998). A grid-enabled MPI : message passing in heterogeneous distributed computing systems. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, pp. 1–11. IEEE Computer Society.

Object Management Group, Inc. (2007). *Unified Modeling Language (UML) 2.1.2 Superstructure*. Object Management Group, Inc. Final Adopted Specification.

Ok, M. et M.-S. Park (2006). Distributing Requests by (around k)-Bounded Load-Balancing in Web Server Cluster with High Scalability. *IEICE - Trans. Inf. Syst. E89-D*(2), 663–672.

Ranjan, S., J. Rolia, H. Fu, et E. Knightly (2002). QoS-Driven Server Migration for Internet Data Centers. In *10th International Workshop on Quality of Service (IWQoS 2002)*.

Urgaonkar, B. et P. Shenoy (2004). Sharc : Managing CPU and Network Bandwidth in Shared Clusters. *IEEE Trans. Parallel Distrib. Syst. 15*(1), 2–17.

Widenius, M. et D. Axmark (2002). *Mysql Reference Manual*. Sebastopol, CA, USA : O'Reilly & Associates, Inc.

Summary

Clusters are often used to manage an infrastructure of replicated servers in order to face important and variable loads. Clustered J2EE architectures are an example of such infrastructures, which implement dynamic web servers. In previous works, we explored the design and implementation of an autonomic system which allows the number of replicated servers to be dynamically adjusted according to the work load. However, the reconfiguration of the infrastructure requires to stop and/or start some of the servers, which causes loss of the servers' execution state. Virtual machines allow to elegantly deal with this problem, as they provide transparent migration of a system without any loss of state. Servers can be deployed over virtual machines which can be distributed on a variable set of physical nodes according to the work load. In this paper, we describe this approach, its implementation and evaluation.