Adaptation multi-niveaux : l'infrastructure au service des applications

Erwan Daubert, Françoise André, Olivier Barais

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

Adaptation multi-niveaux : l'infrastructure au service des applications

Erwan Daubert, Françoise André, Olivier Barais

IRISA / INRIA
Campus de Beaulieu
35042 Rennes cedex – France
Erwan.Daubert@inria.fr
{Francoise.Andre, Olivier.Barais}@irisa.fr

Résumé
De plus en plus d'applications sont construites à base de services et exécutées dans des environnements large échelle, dynamiques et distribués. La notion de "cloud" est souvent utilisée pour parler de ces environnements. Le paradigme de cloud computing correspond à un ensemble de niveaux indépendants ayant chacun leurs propres objectifs. Ces objectifs sont aussi bien de garantir la qualité de service fournie aux utilisateurs que de gérer les ressources disponibles. Mais la qualité de service tout comme les ressources disponibles peuvent évoluer au fil de l'exécution et les différents niveaux doivent s'adapter à ces évolutions. Du fait de leur indépendance, ces différents niveaux s'adaptent indépendamment les uns des autres. Pourtant si l'on considère un ensemble d'adaptations s'effectuant sur des niveaux différents, il est possible que ces adaptations soient contradictoires ou redondantes entre elles, ne produisant pas au final l'effet escompté ou bien l'atteignant de manière non performante. Nous présentons dans cet article ce que nous appelons adaptation multi-niveaux ainsi que l'intérêt qu'elle peut avoir. Nous illustrons cet intérêt sur le cas de la migration de service permettant d'adapter une application en utilisant l'infrastructure sous-jacente et ses capacités.

Mots-clés : adaptation, applications à base de services, cloud computing, adaptation multi-niveaux

---

1. Introduction

De plus en plus d'applications sont construites à base de services et exécutées dans des environnements large échelle, dynamiques et distribués. Une application à base de services correspond à une architecture d'application dans laquelle l'élément de base est le service qui fournit une fonctionnalité, comme par exemple accéder à une ressource, ouvrir une IHM ou traiter des données.
L'un des modèles de système distribué de plus en plus présent est le cloud. Celui-ci est défini comme un ensemble de niveaux que sont l'IaaS, le PaaS et le SaaS. L'IaaS (Infrastructure as a Service) offre une capacité de gestion des ressources disponibles suffisamment robuste et intelligente pour optimiser leur utilisation. On peut voir l'IaaS comme étant le système d'exploitation d'une infrastructure distribuée dans lequel on souhaite simplifier l'accès au ressources quelque soit leur localisation. Au-dessus de l'IaaS, le PaaS (Platform as a Service) a pour but de faciliter la conception d'applications en simplifiant/minimisant l'interaction avec l'IaaS. Enfin les SaaS (Software as a Service) représentent les applications composées d'un ensemble de services.
Du fait, entre autres, de la dynamicité des services (ceux-ci peuvent apparaître et disparaître à n'importe quel moment), les applications à base de service (SaaS) doivent être autonomes. De même l'infrastructure doit être capable de s'adapter afin de tenir compte de l'évolution des ressources. Ces ressources peuvent en effet, tout comme les services, apparaître et disparaitre du fait de connexions réseaux intermittentes par exemple. Enfin le PaaS doit quant à lui assurer la liaison entre l'ensemble des services formant les applications ainsi que la liaison avec l'IaaS pour le déploiement des services.
Il existe différents travaux concernant l'adaptation que ce soit pour l'infrastructure ou pour les applications. Cependant aucun d'entre eux n'a été mené sur l'adaptation de l'ensemble des différents niveaux que forme l'IaaS, le PaaS et le SaaS. Pourtant l'adaptation de l'un de ces éléments peut avoir un impact sur l'autre. C'est le cas par exemple de l'adaptation de l'algorithme d'ordonnancement des processus

applicatifs qui peut agir sur le temps d'exécution des processus sur le ou les processeurs et ainsi potentiellement réduire ou augmenter le temps de réponse d'un service en réduisant ou augmentant son temps d'exécution.
Dans cet article, nous proposons de considérer l'adaptation comme une préoccupation transverse aux différents niveaux d'un système tel que le cloud en utilisant un cadre de développement générique pour les systèmes d'adaptations nommé SAFDIS. Après un bref état de l'art en section 2 concernant les travaux sur l'adaptation que ce soit au niveau de l'infrastructure, de la plate-forme ou des applications, nous présentons en section 3, le modèle MAPE [21] comme fondement de notre système d'adaptation SAFDIS et décrivons la relation entre le système d'adaptation et les différents niveaux. Nous détaillons en section 4, notre utilisation de la plate-forme et surtout de l'infrastructure pour adapter les applications. Enfin la section 5 conlut cet article et présente des travaux futurs.

2. État de l'art

Il existent de nombreux travaux concernant l'adaptation dynamique que ce soit au niveau des infrastructures, des plates-formes ou des applications. Au niveau de la plate-forme, XtreemOS [23], Nimbus[2] ou Globus[1] proposent des solutions concernant la gestion des apparitions et disparitions des ressources et leur allocation en fonction des besoins des tâches à exécuter. On peut aussi citer Entropy [19] qui est un gestionnaire de machine virtuelle pour cluster, capable de faire migrer les machines virtuelles d'un nœud du cluster à un autre en fonction des ressources dont la machine virtuelle a besoin et des ressoruces disponibles sur les nœuds du cluster. Dans les systèmes pair à pair, des travaux sur les mécanismes d'indexation comme ceux de Aberer *et al.* [4] illustrent le besoin de reconfigurations lors de l'apparition et la disparition de nœuds. La particularité de ces travaux consiste à adapter l'organisation du réseau P2P mais aussi adapter la réplication des informations dans les DHTs selon la taille du réseau. Il existe également des travaux sur la configuration réseau utilisée sur les nœuds d'un système distribué [26]. Ces travaux adaptent des paramètres "critiques" d'un réseau pouvant entrainer, par leur seule modification, une reconfiguration importante des communications entre les nœds d'un système distribué large échelle.
De nombreux travaux portent sur l'adaptation au niveau des applications en offrant un cadre applicatif au dessus de mécanismes d'adaptation proposés par la plate-forme. On peut citer par exemple SAFRAN [15] qui est un système d'adaptation à part entière et qui permet d'adapter des applications à base de composants Fractal [9]. SAFRAN intègre entre autres WildCAT [16] qui est un moteur d'observation et FScript [14] qui est un langage de script permettant de définir des politiques d'adaptation avec un ensemble de primitives telles que l'ajout ou la suppression de composants, la modification de paramètres d'un composant. Dynaco [10], comme SAFRAN, est un système d'adaptation pour les applications Fractal. Sa particularité est d'être conçu pour adapter les applications sur grille de calcul. Il permet notamment de gérer la sauvegarde d'état des composants afin d'assurer la cohérence de l'application et d'éviter de perdre l'état d'avancement de l'application au moment de l'adaptation. Enfin FraSCAti est une mise en œuvre du standard SCA [12]. La particularité de FraSCAti est qu'elle offre un ensemble de primitives permettant d'observer et d'adapter dynamiquement les composants SCA. FraSCAti permet aussi de gérer des propriétés non-fonctionnelles au niveau des composants. Grâce aux mécanismes d'adaptation proposés, il est alors possible d'introduire ou d'enlever voir de mettre à jour de telle propriétés.
Cependant ces travaux, bien que travaillant sur l'infrastructure, la plate-forme ou les applications considèrent rarement l'adaptation comme un problème global à l'ensemble de ces niveaux et il n'existe, à notre connaissance, aucun travaux capable de gérer l'adaptation de manière transverse à ces niveaux afin d'assurer une adaptation cohérente et efficace.

3. Système d'adaptation

Un système dynamiquement adaptable, ou autonome, est une entité logicielle capable de s'adapter aux changements de son environnement lors de son exécution.
L'adaptation dynamique d'entités logicielles a été introduite au début des années 2000 [11, 25]. Le modèle MAPE (Monitoring, Analysis, Planning, Execution) [21] est une abstraction proposée par Jeffrey O. Kephart et David M. Chess. Il introduit dans l'entité à adapter un gestionnaire d'adaptation qui est chargé d'observer l'entité et ses relations afin de l'adapter.
L'adaptation y est décomposée en quatre fonctions. La première est l'observation, qui consiste à sur-

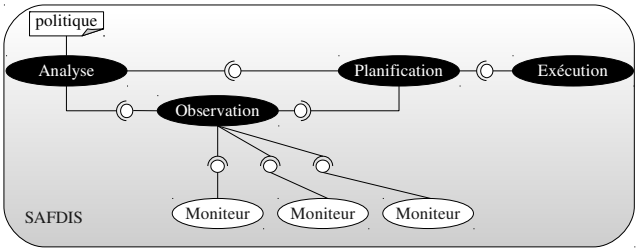

FIGURE 1 – SAFDIS

veiller le contexte de l'application afin de détecter les changements qui nécessiteraient une adaptation. Ce contexte peut aller des ressources disponibles pour le système (bande passante, processeurs...) jusqu'aux préférences des utilisateurs en passant par les propriétés de son environnement (météo, date...). La seconde fonction est l'analyse (ou décision) qui est chargée d'analyser les évènements qui sont apparus et de décider, en fonction de ces évènements, si une adaptation est judicieuse et si oui laquelle. La troisième fonction est celle de la planification, qui est chargée de décomposer la stratégie d'adaptation choisie par la phase d'analyse en un plan d'actions élémentaires. Il existe peu de travaux à ce sujet et elle est souvent confondue avec la phase d'analyse ou la phase suivante qu'est l'exécution. Celle-ci correspond à la dernière fonction et est chargée d'exécuter les actions dans l'ordre prévu par le plan.
De plus on peut identifier différents types d'actions d'adaptation selon leur impact sur l'entité à adapter. Tout d'abord l'adaptation paramétrique qui modifie un ou plusieurs paramètres de l'entité (par exemple modification du niveau de log dans une application). L'adaptation fonctionnelle permet quant à elle de remplacer le code d'une fonction de l'entité par une autre sans pour autant changer son comportement (par exemple remplacement d'une fonction de tri à bulles par une fonction de tri rapide). À l'inverse, l'adaptation comportementale consiste à changer le comportement de l'entité (par exemple ajouter une fonctionnalité à l'entité). Enfin, l'adaptation environnementale permet de modifier l'environnement d'exécution du service(par exemple migrer l'entité d'une machine à une autre).

3.1. SAFDIS

SAFDIS (Self-Adaptation For DIstributed Services) [6] est un cadre de développement générique pour les systèmes d'adaptations distribués et auto-adaptables. Il a aussi pour particularité d'être un système d'adaptation pour applications à base de services, lui-même construit à base de services. Une architecture orientée services est une architecture logicielle dont l'élément de base est le service. Un service correspond à une fonctionnalité simple comme accéder à une ressource, ouvrir une IHM ou traiter des données. C'est la composition de ces fonctionnalités qui permet la conception d'applications. Cette composition s'effectue généralement durant l'exécution de l'application, facilitant ainsi l'évolution continue des services et des applications. Basé sur le modèle MAPE, SAFDIS fournit un ensemble d'interfaces permettant d'intégrer des mises en œuvre existantes pour chaque phase du processus d'adaptation, et de les interconnecter (voir Figure 1).
Nous proposons entre autre dans notre mise en œuvre de pouvoir sélectionner dynamiquement l'algorithme de la phase de planification à utiliser ([7]). De manière générale, la phase de planification est un sujet peu abordé dans le cadre de l'adaptation dynamique et la construction des plans d'actions se fait la plupart du temps de manière statique lors de la définition des stratégies ou en utilisant un modèle fixe (par exemple les actions de type A sont toujours exécutées avant les actions de types B). Pourtant plusieurs travaux concernant la planification ont été effectués dans le domaine de l'intelligence artificielle. Ces travaux ont produit de nombreux algorithmes ayant différentes propriétés pouvant être intéressantes dans le cadre d'adaptation dynamique pour systèmes distribués comme par exemple la parallélisation des actions d'adaptation ou encore la sélection des actions en fonction de la stratégie choisie par la phase de décision et les contraintes posées sur le système d'adaptation.

3.2. L'adaptation : un problème transverse à l'ensemble du système

Chaque niveau du cloud possède son propre objectif. L'IaaS a pour objectif de gérer les ressources et de les allouer à la demande de manière efficace. Le PaaS lui doit gérer les applications qui s'exécutent audessus de lui. Enfin le SaaS, correspondant à l'ensemble des applications à base de services, doit pouvoir

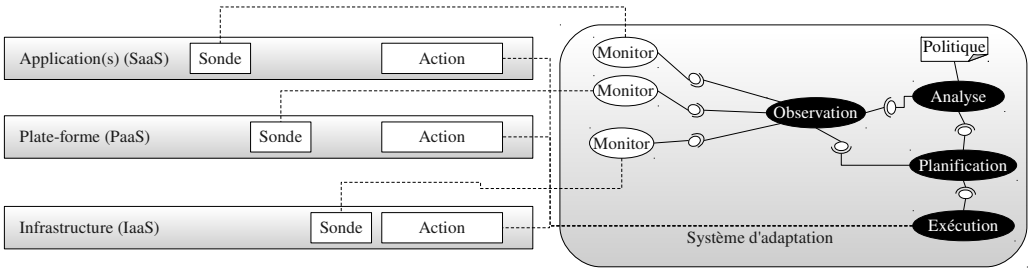

FIGURE 2 – L'adaptation comme problématique transverse

continuer à fournir les fonctionnalités de chaque application aux utilisateurs en respectant des qualités de service pouvant varier d'une application à une autre.

Afin de pouvoir atteindre leurs objectifs, chacun peut envisager d'être autonome et donc de s'adapter. Cependant ces objectifs peuvent être, dans certains cas, contradictoires ou redondants impliquant ainsi que plusieurs adaptations à différents niveaux soient incohérentes.

Par exemple, considérons deux systèmes d'adaptation indépendants, l'un au niveau du SaaS et l'autre au niveau de l'IaaS. Pour le premier, l'objectif à atteindre est la réduction du temps d'exécution d'une application A et l'action choisie est le remplacement de l'un des algorithmes de cette application. Pour le second, l'objectif est d'assurer le plus de temps processeur à l'application la plus utilisée qui est B et l'action choisie est l'augmentation de la priorité des processus de cette application au niveau de l'ordonnanceur. Dans cet exemple, les deux actions choisies indépendamment peuvent être contradictoires car l'action choisie pour l'application B peut potentiellement empêcher l'action choisie pour l'application A d'atteindre l'objectif fixé (réduire le temps d'exécution).

Si l'on considère la même situation avec cependant dans le second système d'adaptation un objectif correspondant toujours à assurer le plus de temps processeur à l'application la plus utilisée mais cette fois il s'agit de l'application A ; alors dans cet exemple, les deux actions choisies indépendamment peuvent être redondantes car seulement l'une des actions aurait pu suffire.

C'est pourquoi il est nécessaire d'envisager l'adaptation comme un problème transverse à l'ensemble des niveaux et donc de proposer un système d'adaptation multi-niveaux c'est-à-dire un système d'adaptation étant capable d'observer l'ensemble des niveaux et d'adapter aussi ces mêmes niveaux (Figure 2).

Si l'on considère maintenant les différents types d'actions d'adaptation (voir la section 3), nous constatons qu'ils ne peuvent généralement être définis qu'à un nombre restreint de niveaux en fonction de la nature de l'entité à adapter. Ainsi pour l'adaptation d'une application, l'adaptation paramétrique est dépendante de l'application et est donc implémentée au niveau de l'application alors que l'adaptation fonctionnelle peut être quant à elle implémentée aussi bien au niveau de l'application en utilisant de l'adaptation paramétrique et l'équivalent d'un "switch case" sur les algorithmes disponibles, ou au niveau de la plate-forme en intégrant de manière dynamique, dans le code, une redirection vers la version de l'algorithme que l'on souhaite utiliser. L'adaptation comportementale est implémentée au niveau de la plate-forme de la même manière que l'adaptation fonctionnelle. Enfin l'adaptation environnementale est implémentable au niveau de la plate-forme ou de l'infrastructure.

Chacun de ces types, pris indépendamment, permet d'effectuer un certain nombre d'adaptation sur les entités mais aucun n'est suffisant à lui seul. Par exemple, l'adaptation paramétrique n'est pas suffisante pour pouvoir faire de l'adaptation efficace dans le cas où toutes les ressources utilisées sont surchargées car elle ne permet pas de faire une demande de nouvelle ressource. Là encore envisager l'adaptation comme étant un problème transverse nous permet de disposer d'un plus grand panel d'actions et de stratégies.

Nous avons identifié deux possibilités pour effectuer de l'adaptation multi-niveaux. La première consiste à définir un ensemble de systèmes d'adaptation, chacun spécialisé pour un niveau particulier et d'intégrer un mécanisme de coordination entre ces différents systèmes d'adaptation afin de coordonner leur décision. L'avantage de cette solution est de complètement séparer les différents niveaux et ainsi de pouvoir définir plus facilement l'ensemble des actions d'adaptation pour chaque niveau. Cependant cette avantage est aussi un inconvénient car la complexité que l'on évite au niveau de la définition des ac-

tions et des stratégies se répercute sur la couche d'interopération entre les systèmes d'adaptation des différents niveaux. Le second avantage de cette solution est de pouvoir facilement réutiliser les différents travaux d'adaptation existants pour les différents niveaux. La deuxième solution consiste à utiliser un seul système d'adaptation qui va gérer l'ensemble des niveaux. L'avantage de cette solution est de ne pas surcharger le système avec un ensemble plus ou moins important de systèmes d'adaptation s'exécutant en parallèle. De plus cette solution offre la possibilité de définir des stratégies d'adaptation tenant compte des différents niveaux en même temps.
C'est la solution que nous avons choisi d'utiliser avec SAFDIS. Un expert en adaptation doit donc définir, pour chaque niveau, l'ensemble des sondes et les actions accessibles depuis le système d'adaptation. Ensuite l'expert peut définir l'ensemble des politiques d'adaptation orientant le choix de l'adaptation au niveau de la phase d'analyse. C'est ensuite la phase de planification qui va sélectionner les actions à utiliser afin d'effectuer de l'adaptation qui soit ni contradictoire ni redondante.

4. L'infrastructure au service des applications

Dans cette section, nous allons présenter l'utilisation de SAFDIS dans le cadre d'un environnement distribué large échelle composé de XtreemOS, de la plate-forme de services OSGi et d'applications basées sur cette plate-forme. Nous allons plus particulièrement présenter une mise en œuvre de la migration de service premièrement utilisant les capacités fournies par la plate-forme (voir Figure 3 et 4) puis une seconde utilisant certaines capacités du système XtreemOS à partir d'une plate-forme OSGi (voir Figure 7 et 8). La migration consiste à déplacer un service S1 d'une ressource R1 à une ressource R2. Cette action d'adaptation n'est pas disponible au niveau de l'application car celle-ci n'a pas la maitrise des ressources. La plate-forme quant à elle peut communiquer avec les ressources (par l'intermédiaire de l'infrastructure) et peut donc utiliser cette action. Cependant il n'existe pas de mise en œuvre dans les plates-formes actuelles. L'infrastructure quant à elle possède l'ensemble des primitives nécessaires pour effectuer la migration de tâche mais XtreemOS ne propose pas directement de primitive de migration.

4.1. Migration de service par la plate-forme

L'alliance OSGi [5], 'consortium de technologie innovante', a défini un ensemble de spécifications[3] décrivant le fonctionnement que doit avoir une plate-forme orientée services. En plus de ces spécifications, un ensemble d'interfaces de services basiques sont proposées, tel qu'un service de `journalisation` ou encore `http`, afin de les standardiser. Outre les services, on peut aussi distinguer les *bundles*. Un *bundle*, correspondant à une archive binaire Java, est l'élément de base dans la plate-forme. C'est lui qui contient un ou plusieurs services. Ces services sont le plus souvent reliés d'un point de vue logique. Un *bundle* les expose en les enregistrant sur la plate-forme. C'est aussi le *bundle* qui définit les dépendances statiques de paquetages Java (*imports* et *exports*) pour les services. Ainsi pour qu'un service fonctionne, il est nécessaire que les dépendances de son *bundle* soit résolues. OSGi propose aussi une standardisation concernant la communication entre différentes plates-formes permettant ainsi de définir une plate-forme de services distribuée comme étant un ensemble de plates-formes interconnectées entre elles. On peut donc imaginer un ensemble de plates-formes OSGi comme étant la mise en œuvre du PaaS capable de définir sur laquelle de ces plates-formes il est nécessaire de déployer tel ou tel service en fonction des besoins qui lui sont associés. Enfin OSGi propose un ensemble de primitives utilisables sur les services ou les *bundles* comme l'ajout, la suppression, l'arrêt, le démarrage et la mise à jour sans devoir redémarrer la plate-forme. Cependant il n'existe aucune primitive permettant d'effectuer de la migration de *bundle* ou de service. Pour combler cette lacune dans la cadre de l'utilisation d'OSGi comme PaaS, nous avons défini deux services, nommés BundleManagement et ServiceManagement, collaborant l'un avec l'autre afin d'effectuer ce type de migration.

4.1.1. Fonctionnement

Pour pouvoir définir un service sur une plate-forme OSGi, il est nécessaire que le *bundle* qui le gère soit installé et même démarré sur cette plate-forme (voir Figure 5 et 6). C'est pourquoi, si nous souhaitons effectuer de la migration de service, il faut aussi pouvoir effectuer de la migration de *bundles*. Pour effectuer la migration d'un *bundle*, il est nécessaire de posséder une URL donnant l'adresse du JAR représentant le *bundle*, qui doit être accessible depuis la plate-forme distante. Nous utilisons le service `HTTPService`, un service standard défini dans la spécification OSGi, qui permet de rendre disponible des ressources sur le réseau. Ainsi, le JAR est publié par l'intermédiaire de ce service, puis le *bundle* est installé depuis la

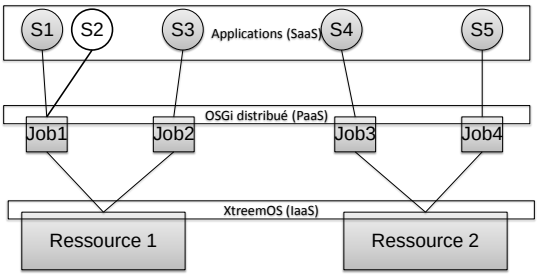

FIGURE 3 – Avant la migration du service S2

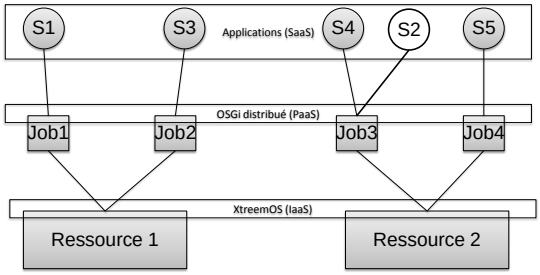

FIGURE 4 – Après la migration du service S2

plate-forme distante grâce à l'URL fournie par le HTTPService.
Ensuite, bien qu'il soit nécessaire de faire migrer le *bundle* pour pouvoir migrer un seul service, nous ne souhaitons pas nécessairement migrer l'ensemble des services du *bundle* avec celui-ci. Pour cela, nous définissons, sur la plate-forme distante, un filtre qui identifie les services à migrer. C'est ensuite au *bundle* lui-même de décider s'il enregistre seulement ces services ou s'il enregistre la totalité. Le respect de ce bon comportement est à la charge du développeur.

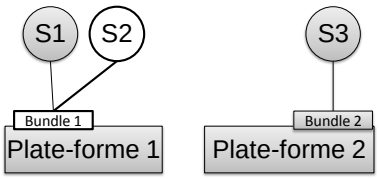

FIGURE 5 – Avant la migration du service S2

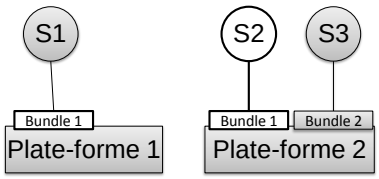

FIGURE 6 – Après la migration du service S2

En outre, quand on fait migrer un service, il peut être nécessaire de conserver son état avant la migration. Celui-ci peut correspondre aux données persistantes qu'un *bundle* peut contenir. C'est pourquoi nous définissons, en plus du filtre sur les services, les états des services avant migration. Une fois que le bundle migré enregistre son service, l'état de celui-ci est rechargé. Pour cela, il est nécessaire que le service implémente l'interface *Originator* que nous avons définie et qui permet de sauvegarder et recharger l'état du service. Cela correspond à l'utilisation du Design Pattern Memento [17]. Il appartient au développeur des services de définir les informations nécessaires à sauvegarder pour pouvoir recharger correctement l'état d'un service.
Une fois ces trois premières étapes effectuées, le *bundle* peut être démarré sur la plate-forme distante. Dans le cas de la migration, nous retirons aussi le service que nous venons de migrer. Cette adaptation est représentée sur les Figures 3 et 4. Dans le cas d'une duplication de service, le service n'est pas retiré. En plus de la migration et de la duplication, le ServiceManagement permet aussi d'enregistrer ou d'arrêter un service sur la plate-forme et de modifier les propriétés définies pour un service.

4.1.2. Contraintes et limitations

Bien qu'offrant une nouvelle action d'adaptation au niveau de la plate-forme, notre mise en œuvre de la migration de service impose un ensemble de contraintes sur la conception des applications afin de pouvoir être utilisée. Par exemple, il est nécessaire que l'enregistrement de services s'effectue par l'intermédiaire du ServiceManagement afin de tirer partie de l'ensemble des actions d'adaptation proposées par le ServiceManagement. Une autre contrainte, pouvant par contre être vue aussi comme un avantage, réside dans l'utilisation du filtre pour l'enregistrement des services. En effet, le développeur doit gérer lui-même les services qu'il souhaite ré-enregistrer après migration.
Pour utiliser cette action, il est nécessaire de gérer différentes contraintes propres à OSGi. C'est le cas des dépendances binaires des *bundles* (paquetage Java importés). En effet, si l'on migre un *bundle*, celui-ci doit pouvoir résoudre ses dépendances. Pour cela, il doit être capable de trouver, parmi les *bundles*

déjà présents sur la nouvelle plate-forme, ceux lui permettant d'accéder aux paquetages qui lui sont nécessaires. Notre service de BundleManagement ne se charge pas de contrôler ces dépendances. Ainsi, bien que la migration d'un *bundle* puisse se faire correctement, son démarrage ne pourra se faire tant que les paquetages dont il a besoin ne sont pas accessibles.
Pour résoudre cette contrainte, nous avons choisi de transférer systématiquement les bundles nécessaires depuis la plate-forme d'origine vers la nouvelle. Cela est géré par le système d'adaptation.

4.2. Migration de service par l'infrastructure

Notre deuxième solution consiste à migrer la plate-forme plutôt qu'un seul service et ainsi éviter les contraintes imposées aux développeurs et la forte dépendance à la plate-forme. En effet, la reconfiguration de la plate-forme vers laquelle le service est migré nécessite potentiellement de faire migrer un ensemble important de *bundles* afin de satisfaire les dépendances nécessaires au fonctionnement du service migré. De même pour chaque *bundle* migré, il peut être nécessaire d'en migrer d'autres et ainsi de suite. Du coup, alors que l'objectif de l'adaptation était de décharger la plate-forme A car celle-ci était surchargée, il n'est pas impossible que ce soit, après l'exécution de l'adaptation, la plate-forme vers laquelle le service est migré qui soit surchargée. Du plus il y a, au niveau de l'infrastructure, un ensemble de primitives pouvant faciliter cette migration de plate-forme telles que la création, l'arrêt de tâches, mais aussi la sauvegarde d'état pour les tâches permettant ainsi de recharger l'état d'une plate-forme sans demander aux développeurs de mettre en place cette sauvegarde au niveau de leurs services.
Une tâche, aussi appelée job, est la représentation d'un ou plusieurs processus d'une application. Chaque job possède une description permettant au gestionnaire de ressources d'identifier ses besoins (mémoire vive, capacité de disque, ...) et ainsi de faciliter la sélection des ressources nécessaires à son bon fonctionnement. Ainsi plus la description est précise et plus la sélection des ressources à attribuer sera efficace. En effet, si la tâche a besoin de 50Mo de mémoire vive, il n'est pas nécessaire d'utiliser une ressource X complètement libre qui possède 4Go de mémoire vive mais plutôt une ressource Y qui possède aussi 4Go de mémoire vive et n'est utilisée qu'à 60% et possède donc encore 500Mo de mémoire vive disponible. L'adaptation, bien que servant historiquement et encore le plus souvent, à adapter le fonctionnement des entités afin d'offrir une qualité de service suffisante aux utilisateurs, est aussi de plus en plus utilisée dans le cadre de la maîtrise de la consommation énergétique. Dans ce cadre, l'utilisation correcte des ressources est très importante, d'où l'importance d'une définition des besoins correcte et précise. Cependant les besoins d'un job sont difficiles à déterminer précisément et à l'avance. En effet ceux-ci dépendent fortement de l'utilisation des processus présents dans le job et donc de l'utilisation de l'application (ou du/des service(s)) à laquelle appartient ces processus. D'où l'intérêt de pouvoir migrer les jobs en fonction de l'évolution des besoins de celui-ci. En effet, il est possible de surveiller l'utilisation du job et plus précisément la variation dans l'utilisation des ressources. Si le système d'adaptation se rend compte d'une trop grande variation, il peut envisager de migrer un ou plusieurs jobs (voir figure 7 et 8) sur d'autres ressources correspondant plus précisément à leurs besoins.

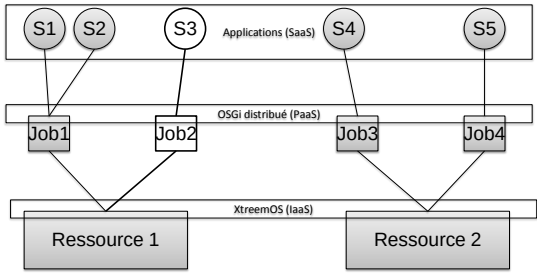

FIGURE 7 – Avant la migration du job 2

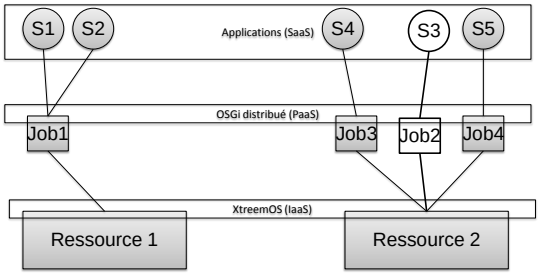

FIGURE 8 – Après la migration du job 2

XtreemOS [23] est un système d'exploitation de grille fondé sur Linux ayant pour but d'intégrer, dans une seule grille de calcul, un ensemble hétérogène de dispositifs informatiques que sont les appareils mobiles de type smartphone, les ordinateurs personnels ou encore les clusters. XtreemOS fournit un ensemble de fonctionnalités allant de la sécurité à la gestion des ressources. Il propose une gestion des

organisations virtuelles. Une organisation virtuelle est un rassemblement d'entreprises ou d'organisations qui partagent leurs ressources et leurs compétences afin de pouvoir répondre de manière efficace à des besoins aussi bien au niveau des services rendus (tels que gestion de marchés financiers, gestion de connaissances médicales) qu'au niveau de l'utilisation des capacités des ressources informatiques. Cette coopération est supportée par les réseaux informatiques dont l'interconnexion gérée par XtreemOS permet le partage efficace des ressources. XtreemOS peut être utilisé comme un gestionnaire d'infrastructure et donc faire partie de l'IaaS [24].
C'est le PaaS qui peut créer de nouvelles tâches sur les ressources et donc dans notre cas d'utilisation, c'est au niveau des plates-formes OSGi que cela doit s'effectuer. Ici, ces tâches correspondent à la création de nouvelles plates-formes OSGi qui vont étendre la plate-forme distribuée (PaaS) et dans lesquelles seront déployés un ou plusieurs services. C'est cette fonctionnalité que nous avons implémentée afin de l'utiliser pour effectuer de la migration de service sur de nouvelles ressources. En effet, la migration de tâches n'est pas une primitive implémentée dans la version 2.1 du système d'exploitation XtreemOS.

4.2.1. Création d'un Job OSGi sur XtreemOS

La création d'une tâche sous XtreemOS peut se faire de plusieurs manières. Tout d'abord les commandes xsub et xsub.sh permettent de soumettre un job au système. La première prend en paramètre la description d'un job sous la forme d'un fichier JSDL [8] alors que la seconde génère automatiquement ce fichier JSDL avant de le soumettre. L'intérêt de définir soi-même la description JSDL vient du fait qu'il est possible de préciser les besoins de la tâche afin de faciliter l'allocation de ressources qu'effectue XtreemOS. Une troisième solution consiste à effectuer la création de job de manière programmatique, c'est-à-dire en utilisant une API permettant de définir une description JSDL. C'est ensuite l'application dans laquelle l'API est utilisée qui va demander l'exécution d'un job à la grille.
Cette API s'intitule SAGA (Simple API for Grid Applications) [18]. Plusieurs mises en œuvre de cette API existent en différents langages. XOSAGA est la mise en œuvre définie et utilisée par XtreemOS. Elle supporte entre autres les langages C/C++ et Java et ajoute, en plus des fonctionnalités standard de SAGA, un ensemble d'extensions spécifiques à XtreemOS [13]. Du fait de l'utilisation de la plate-forme OSGi qui est une solution basée sur le langage Java, nous utilisons XOSAGA en version Java. De plus nous avons choisi de restreindre notre utilisation de XOSAGA aux seuls éléments présents dans l'API SAGA dans le but d'assurer la portabilité de nos travaux sur des systèmes supportant SAGA autres que XtreemOS.
La description JSDL (voir Figure 9) générée par l'intermédiaire de l'API SAGA (voir Figure 10) doit tout d'abord contenir la commande Java permettant d'exécuter la plate-forme. Elle doit aussi contenir l'ensemble des propriétés nécessaires à sa configuration. On y trouve par exemple la localisation des *JARs* définissant l'ensemble de *bundles* nécessaires au fonctionnement d'une plate-forme. Dans notre cas d'utilisation, cela comprend, entre autres, le *bundle* qui définit le fonctionnement de la plate-forme ainsi que l'ensemble des *bundles* représentant SAFDIS. Une autre propriété importante est la localisation/définition du cache de la plate-forme. Ce cache permet de conserver une trace de la configuration d'une plate-forme OSGi permettant d'arrêter et de redémarrer celle-ci sans perte d'état. Le cache est aussi utilisé comme système de fichier par la plate-forme et offre, pour chaque *bundle*, un espace personnel lui permettant de sauvegarder des informations qui lui sont propres.
Les JARs et le cache doivent être accessible de la ressource sur laquelle va s'exécuter la plate-forme. C'est pourquoi nous utilisons XtreemFS [20]. En effet, XtreemFS permet de partager des données sur l'ensemble des nœuds physiques (grilles, clusters, ordinateurs voire téléphones de type smartphone) représentant le système XtreemOS assurant l'accès aux informations nécessaires depuis n'importe laquelle de ces ressources.
Ainsi pour satisfaire notre objectif, il est nécessaire de lancer tout d'abord une plate-forme OSGi par l'intermédiaire de la commande xsub avec la description JSDL associée. Ensuite cette plate-forme, dotée de SAFDIS, est en mesure de créer de nouvelles plates-formes et de demander à XtreemOS de les exécuter sur de nouvelles ressources. L'arrêt d'une plate-forme quant à lui peut s'effectuer grâce à l'une des primitives fournies directement par la plate-forme et va entrainer l'arrêt du job.

4.2.2. Migration par XtreemOS

Durant nos expérimentations, nous avons utilisé XtreemOS 2.1. Cette version ne propose aucune primitive concernant la migration de job. Elle possède toutefois de nombreuses autres primitives permettant de concevoir la migration de job. Nous avons choisi d'utiliser la création de job, et l'arrêt de job en utilisant

```
...
<jsdl:Application>
 <jsdl:ApplicationName>OSGi Felix</jsdl:ApplicationName>
  <jsdl-posix:POSIXApplication>
  <!-- Definition de l'executable -->
  <jsdl-posix:Executable>java</jsdl-posix:Executable>
  <!-- Definition du classpath de la plate-forme pour integrer SAGA -->
  <jsdl-posix:Argument>-cp</jsdl-posix:Argument>
  <jsdl-posix:Argument>...</jsdl-posix:Argument>
  <!-- Definition des proprietes remplacant le fichier de configuration -->
  <jsdl-posix:Argument>-D...</jsdl-posix:Argument>
   ...
  <jsdl-posix:Argument>-D...</jsdl-posix:Argument>
  <!-- Definition de la location des Jars -->
  <jsdl-posix:Argument>--felix-location</jsdl-posix:Argument>
  <jsdl-posix:Argument>...</jsdl-posix:Argument>
  <!-- Definition de l'emplacement du cache -->
  <jsdl-posix:Argument>--working-directory</jsdl-posix:Argument>
  <jsdl-posix:Argument>...</jsdl-posix:Argument>
  <!-- Definition des fichiers de log -->
  <jsdl-posix:Output>...</jsdl-posix:Output>
  <jsdl-posix:Error>...</jsdl-posix:Error>
 </jsdl-posix:POSIXApplication>
</jsdl:Application>
...
```

FIGURE 9 – Description JSDL d'un job OSGi

```
if (session == null) { // initialize XtreemOS attributes
  session = SessionFactory.createSession(true);
  serverURL = URLFactory.createURL("xos:///");
  context = ContextFactory.createContext("xtreemos");
  session.addContext(context);
}
// initialize jobService used to create job
JobService js = JobFactory.createJobService(serverURL);
// initialize jd description
JobDescription jd = JobFactory.createJobDescription();

// create job description
jd.setAttribute(JobDescription.EXECUTABLE, "java");
// add OSGi properties + JARs location + cache location
// scriptProperties is an array
  // previously defined with configuration properties
jd.setVectorAttribute(JobDescription.ARGUMENTS, scriptProperties);
jd.setAttribute(JobDescription.OUTPUT, platformsLocation + File.separator
      + platformId + ".output.log");
jd.setAttribute(JobDescription.ERROR, platformsLocation + File.separator
      + platformId + ".error.log");
// Create the job according to the job description
Job job = js.createJob(jd);
// submit the job on XtreemOS
job.run();
```

FIGURE 10 – code SAGA pour créer un job OSGi

le cache des plates-formes OSGi comme étant la sauvegarde de l'état de la plate-forme avant la migration. Il est aussi envisageable d'utiliser la primitive de sauvegarde d'état [22] proposée par XtreemOS au lieu du cache de la plate-forme.

Lors de la création des plates-formes OSGi par l'intermédiaire de l'API SAGA, un cache spécifique est défini pour chaque plate-forme. Lors de la migration, le cache de la plate-forme à migrer (plate-forme A) est réutilisé lors de la création de la nouvelle plate-forme (plate-forme A'). Ainsi la plate-forme A' va recharger l'ensemble de la configuration de la plate-forme A. De cette manière la plate-forme A' correspond exactement à la plate-forme A. Une fois le démarrage de cette plate-forme A' effectuée, la plate-forme A peut être arrêtée ou pas selon que l'on souhaite faire de la migration ou de la réplication.

En plus des propriétés qui définissent l'application (ou les processus de l'application) à exécuter dans un job, la description JSDL peut contenir des informations concernant les besoins du job. Ainsi il est possible de donner des indications sur les ressources souhaitées pour exécuter le job. Ces informations peuvent être le nombre de coeurs, la quantité de mémoire vive, le nombre de processus à exécuter, le système d'exploitation nécessaire pour le bon fonctionnement du job, le nombre de processus par ressource, ... L'ensemble de ces propriétés permet de faciliter le travail du gestionnaire de ressources qui peut grâce à ses informations effectuer une sélection précise des ressources. Cependant lors de nos expérimentations, la version de XOSAGA disponible sur la version 2.1 de XtreemOS ne permettait pas d'utiliser ces propriétés. Une nouvelle version de XOSAGA a été publiée depuis mais nous n'avons pas testé cette fonctionnalité.

4.2.3. Illustration d'une adaptation multi-niveau

Nous allons présenter dans cette partie un exemple d'application sur laquelle nous pouvons utiliser les deux actions de migration présentées ci-dessus en fonction de la situation et des objectifs d'adaptation fixés par l'expert en adaptation.

Cette application est utilisée pour la gestion de crise dans le cadre par exemple d'une intervention sur un accident de la route. Dans cette situation, l'application est sensible au contexte. Le système d'observation MM1 gère un ensemble de sondes afin de collecter les données nécessaires au fonctionnement de l'application. Ce système d'observation correspond à un ensemble de services déployés dans un unique job XtreemOS sur la ressource R2. L'ensemble des sondes gérées par MM1 ne sont utilisées qu'en mode pull, i.e. MM1 demande l'information aux sondes. MM1 est aussi chargé de l'agrégation des données et utilise pour cela de nombreuses opérations mathématiques ainsi que des tables de données de taille importante. De ce fait, MM1 est fortement consommateur de ressources de calcul (en moyenne 25% de la ressource R2). Le reste de l'application n'est pas détaillée ici mais consomme en moyenne 80% des ressources disponibles sur la ressource R1 (voir Figure 11).

L'expert en adaptation chargé de ce système définit un ensemble d'objectifs non-fonctionnels aussi bien au niveau système qu'au niveau applicatif. Ces objectifs sont, par ordre d'importance, de diminuer globalement la consommation énergétique du système, de diminuer les resources consommées par chaque élément de l'application (dans notre exemple nous nous intéressons à MM1), de maximiser la précision

des données produites et/ou utilisées par MM1 et enfin de favoriser les performances de l'application.
Les mécanismes d'adaptation disponibles, en plus des deux actions de migration proposées en 4.1 et 4.2, sont l'arrêt d'un noeud XtreemOS afin de limiter la consommation énergétique, l'introduction d'un composant de cache entre le reste de l'application et MM1 afin d'éviter à MM1 de recalculer les données et ainsi de diminuer la consommation de ressources par MM1. Ce cache estampille les données et les déclare valides pendant une certaine période de temps. Passé ce délai, lorsqu'une nouvelle demande d'information est reçue par le cache, celui-ci interroge MM1 pour obtenir une valeur fraîche.
L'ensemble des objectifs et des mécanismes disponibles nous permettent d'envisager deux adaptations. La première consiste à migrer entièrement MM1 depuis R2 vers R1 tout en introduisant un composant de cache afin de limiter l'utilisation des ressources par MM1 (voir Figure 12). Cette adaptation peut être intéressante dans le cas où il ne reste plus que MM1 qui s'exécute sur R2. Sa migration permet ainsi d'éteindre R2 et donc de diminuer la consommation énergétique du système. La seconde adaptation envisageable consiste à laisser MM1 sur R2 et à migrer seulement le composant de cache sur R1 afin de diminuer la latence de l'accès aux informations par l'application et ainsi favoriser les performances (voir Figure 13).
Ces deux adaptations possibles illustrent un besoin de coordination entre les adaptations au niveau OS et au niveau applicatif dans le premier cas et un besoin d'une granularité différente dans la migration des artefacts logiciels dans le deuxième cas. La coordination entre les actions d'adaptation et le choix du type de migration s'effectue au niveau du système d'adaptation (généralement au niveau de la phase d'analyse). Dans notre mise en œuvre de SAFDIS, c'est la phase de planification qui qui va se charger de la coordination en ordonnant l'exécution des actions d'adaptation que sont la migration et l'ajout du cache et la phase d'analyse qui se charge de choisir le type de migration à effectuer. La phase de planification peut choisir éventuellement le type de migration dans le cas où, contrairement à notre exemple, un job ne contient que un et un seul service (ou composant).

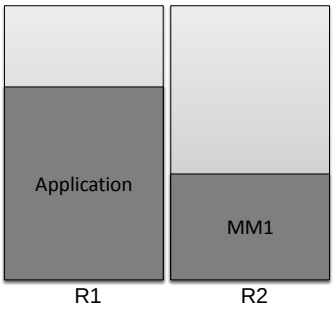

FIGURE 11 – Avant adaptation

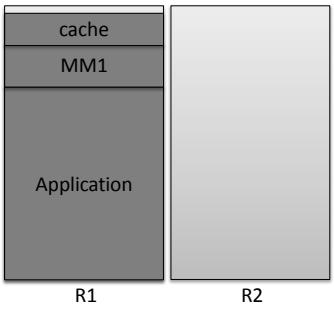

FIGURE 12 – Après adaptation 1

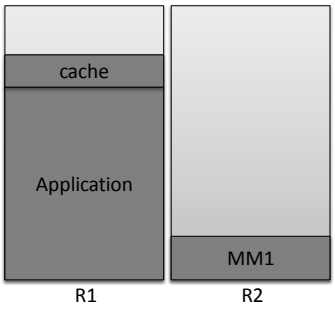

FIGURE 13 – Après adaptation 2

4.2.4. Bilan des solutions

Notre première solution concernant la migration de service permet d'effectuer de la migration de manière précise en permettant de migrer un seul service. Cependant elle peut tout de même entrainer la migration d'un nombre potentiellement conséquent de bundles OSGi afin d'assurer le bon fonctionnement du service à migrer. Notre deuxième solution quant à elle permet d'effectuer la migration d'une plate-forme et donc d'un ensemble de services tout en permettant l'allocation d'une ressource plus appropriée par l'intermédiaire de l'IaaS. Cette solution a l'avantage par rapport à la première de ne pas imposer une méthode de conception des services notamment pour la sauvegarde d'état grâce à l'utilisation des capacités intrinsèques de la plate-forme (le cache) ou de l'infrastructure (la sauvegarde d'état).
Dans notre cas d'utilisation avec OSGi en tant que plate-forme, il n'est pas envisageable de faire de la migration d'un seul service puisqu'un job est une plate-forme, avec cette solution. Cependant, si la technologie utilisée au niveau plate-forme nous permet d'identifier un job pour chaque service alors la migration d'un service est tout à fait possible. De plus, nous avons choisi une API se voulant un standard pour la définition de tâches sur les système distribués et implémentée notamment pour XtreemOS mais aussi pour d'autres gestionnaires d'infrastructures. De ce fait, notre deuxième solution peut tout à fait

être généralisée alors que notre première solution est totalement spécifique à OSGi et ne peut pas être utilisée telle quelle sur d'autres plates-formes.
Ces deux possibilités d'adaptation permettant d'effectuer de la migration de service peuvent être utilisées de différentes manières. La première est utilisable afin de migrer un service pour lui offrir plus de ressources sur une autre plate-forme (Figure 3 et 4). La seconde quant à elle peut être aussi utilisée pour offrir plus de ressources à un service, non pas en migrant celui-ci mais en migrant un job s'exécutant initialement sur la même ressource (Figure 7 et 8). Comme nous l'avons expliqué dans la partie 4.2.3, c'est le système d'adaptation qui se charge de sélectionner l'action adéquate à la situation.
Enfin nous avons montré l'intérêt de l'utilisation de l'infrastructure qui propose des mécanismes réutilisables pour effectuer certaines adaptations de l'application. Une meilleure solution serait envisageable si une primitive de l'infrastructure permettait d'effectuer l'ensemble des actions correspondant à la migration de job. En effet, l'ensemble de ces actions correspond à des fonctionnalités fournies par l'infrastructure. Cette primitive n'existe pas pour l'instant, et il est donc nécessaire, comme nous l'avons fait, d'interagir plusieurs fois avec l'infrastructure.

5. Conclusion

Les systèmes distribués large échelle sont de plus en plus utilisés de nos jours. Cette popularité vient en majorité d'un besoin croissant de puissance de calcul et du prix très important des super calculateurs. L'un des principaux intérêts de ces systèmes est l'élasticité qui consiste à pouvoir facilement augmenter la taille des ressources mise à disposition dans ces systèmes. Le paradigme de cloud computing correspond à un modèle de système distribué dans lequel est défini un ensemble de niveaux ayant chacun leur propre rôle et leur propre objectif dans le fonctionnement du système global.
Afin de pouvoir atteindre leurs objectifs, chacun peut envisager d'être autonome et donc de s'adapter et nous avons montré dans cet article que ces objectifs peuvent être contradictoires ou redondants. Ces potentielles contradictions et redondances imposent donc d'envisager l'adaptation comme un problème transverse entre les différents niveaux d'un système distribué large échelle tel qu'un cloud. L'intérêt de gérer l'adaptation pour l'ensemble des différents niveaux est d'une part d'offrir des actions équivalentes d'un point de vue de leurs objectifs (réduction du temps d'exécution d'une application par exemple) et d'autre part de posséder une gamme d'actions d'adaptation plus importante nous permettant d'envisager de nouvelles adaptations non envisageables si l'on se focalise sur un seul niveau. L'un des exemples est la migration de service qui n'est pas envisageable dans le cadre de l'adaptation d'application si l'on ne tient pas compte de la plate-forme et de l'infrastructure car l'application en elle-même ne maitrise pas les ressources disponibles.
Un autre point que nous n'avons pas mis en avant dans cet article est l'accès aux informations permettant de prendre des décisions. En effet, bien que l'accès à un nombre plus importants d'actions soit intéressant pour envisager de nouvelles adaptations, l'accès à des informations permettant d'en tirer partie est lui aussi important. En effet, si il est impossible de connaître l'utilisation des ressources par les applications, il sera difficile d'envisager la migration de tâche.
Dans la continuité des travaux que nous présentons dans ce papier concernant l'utilisation des fonctionnalités de l'IaaS pour le SaaS, nous souhaitons étudier l'intérêt de l'adaptation multi-niveaux du point de vue de l'infrastructure. En effet, certaines infrastructures (et/ou plates-formes) sont amenées à prendre des décisions d'adaptation concernant les applications afin d'assurer la qualité de service requise. Par exemple, il est envisageable que les applications soit répliquées. Dans ce cas, si une adaptation survient par exemple pour modifier le fonctionnement de l'application, il est nécessaire que l'infrastructure soit notifiée afin de mettre à jour l'ensemble des instances de l'application à partir de celle qui vient d'être adaptée.

Bibliographie

1. Globus alliance. `http://www.globus.org/`. accessed 02/02/2011.
2. Nimbus. `http://www.nimbusproject.org/`. accessed 02/02/2011.
3. OSGi alliance specifications. `http://www.osgi.org/Specifications`. accessed 01/21/2010.
4. K. Aberer, A. Datta, and M. Hauswirth. Multifaceted Simultaneous Load Balancing in DHT-based P2P systems : A new game with old balls and bins. *Self-star Properties in Complex Information Systems*, pages 373–391, 2005.
5. The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.2*. September 2009.

6. Françoise André, Erwan Daubert, and Guillaume Gauvrit. Distribution and Self-Adaptation of a Framework for Dynamic Adaptation of Services. In *The Sixth International Conference on Internet and Web Applications and Services (ICIW)*, St. Maarten Netherlands Antilles, 03 2011. IARIA.
7. Françoise André, Erwan Daubert, Grégory Nain, Brice Morin, and Olivier Barais. F4Plan : An Approach to build Efficient Adaptation Plans. In *7th International ICST Conference on Mobile and Ubiquitous Systems*, Sydney Australia, 12 2010.
8. A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. Job submission description language (jsdl) specification, version 1.0. In *Open Grid Forum, GFD*, volume 56, 2005.
9. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Software—Practice & Experience*, 36(11-12) :1257–1284, 2006.
10. Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Supporting adaptable applications in grid resource management systems. In *Proceedings of 8th IEEE/ACM International Conference on Grid Computing*, page 58–65, Austin, USA, 2007.
11. W.K. Chen, M.A. Hiltunen, and R.D. Schlichting. Constructing adaptive software in distributed systems. In *icdcs*, page 0635. Published by the IEEE Computer Society, 2001.
12. S. C. A. Consortium. *Building Systems using a Service Oriented Architecture*. 2005. Published : Whitepaper.
13. XtreemOS consortium. Final xosaga api engine implementation. Deliverable D3.1.11, September 2010.
14. P.C. David and T. Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP*, volume 2006, 2006.
15. Pierre C. David and Thomas Ledoux. An Aspect-Oriented approach for developing Self-Adaptive fractal components. In *Software Composition*, volume 4089 of *LNCS*, page 82–97, 2006.
16. Pierre-Charles David and Thomas Ledoux. WildCAT : a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing (MPAC)*, page 1–7, Grenoble, France, 2005.
17. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, January 1995.
18. T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. A simple API for Grid applications (SAGA). In *Grid Forum Document GFD*, volume 90, 2007.
19. F. Hermenier, X. Lorca, J.M. Menaud, G. Muller, and J. Lawall. Entropy : a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.
20. F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The XtreemFS architecture—a case for object-based file systems in Grids. *Concurrency and computation : Practice and experience*, 20(17) :2049–2060, 2008.
21. Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, 2003.
22. J. Mehnert-Spahn, T. Ropars, M. Schoettner, and C. Morin. The architecture of the xtreemos grid checkpointing service. *Euro-Par 2009 Parallel Processing*, pages 429–441, 2009.
23. C. Morin. Xtreemos : a grid operating system making your computer ready for participating in virtual organizations. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, pages 393–402. IEEE, 2007.
24. Christine Morin, Jérôme Gallard, Yvon Jégou, and Pierre Riteau. Clouds : a New Playground for the XtreemOS Grid Operating System. Research Report RR-6824, INRIA, 2009.
25. Paul Robertson, Howard E. Shrobe, and Robert Laddaga, editors. *Self-Adaptive Software, First International Workshop, IWSAS 2000, Oxford, UK, April 17-19, 2000, Revised Papers*, volume 1936 of *Lecture Notes in Computer Science*. Springer, 2001.
26. I. Scholtes, J. Botev, A. Hohfeld, H. Schloss, and M. Esch. Awareness-driven phase transitions in very large scale distributed systems. In *Self-Adaptive and Self-Organizing Systems, 2008. SASO'08. Second IEEE International Conference on*, pages 25–34. IEEE, 2008.