Adaptation et cloud computing : un besoin d'abstraction pour une gestion transverse

Erwan Daubert

Thèse

THÈSE INSA Rennes
sous le sceau de l'Université Européenne de Bretagne
pour obtenir le grade de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Informatique

Présentée devantpar
Erwan Daubert
ÉCOLE DOCTORALE : MATISSE
LABORATOIRE : IRISA – UMR6074

Adaptation et Cloud Computing : un besoin d'abstraction pour une gestion transverse

Thèse soutenue le 24 Mai 2013
devant le jury composé de :

Noel De Palma
Professeur, Université Joseph Fourier de Grenoble / *Président*

Françoise Baude
Professeur, Université de Nice Sophia-Antipolis / *Rapporteuse*

Marco Danelutto
Associate Professor, Université de Pise (Italie) / *Rapporteur*

Thomas Ledoux
Maître de conférence, École des Mines de Nantes / *Examinateur*

Jean-Louis Pazat
Professeur, INSA de Rennes / *Directeur de thèse*

Olivier Barais
Maître de conférence, Université de Rennes 1 / *Co-encadrant de thèse*

Remerciements

Coming Soon

Les recherches menant aux présents résultats ont bénéficié d'un financement du Programme de la Communauté européenne septième programme-cadre FP7/2007-2013 sous convention de subvention 215 483 (S-Cube).

Avant d'entrer dans le vif du sujet, j'ai une pensée pour Françoise André avec qui j'ai eu l'honneur de débuter cette thèse et qui, malheureusement, n'est plus là pour la conclusion.

En tant que directrice, elle était exigeante et passionnée. Elle était aussi difficile à convaincre, mais toujours prête à changer d'avis pour peu que vous soyez capable de vous justifier. J'ai apprécié nos discussions qui, avec nos caractères bien trempés, étaient parfois animées, mais surtout toujours intéressantes.

Merci.

J'en profite aussi pour remercier Jean-Louis Pazat qui, dans ces conditions peu communes, a accepté de prendre le relais de Françoise.

Table des matières

Première partie

Introduction

Introduction

En 1969, l'UCLA (University of California, Los Angeles) annonçait l'arrivée imminente de l'Internet (connu à l'origine sous le nom d'ARPANET). Dans cette annonce, Leonard Kleinrock, l'un des scientifiques à l'origine de ce projet disait [89] :

> *As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of 'computer utilities' which, like present electric and telephone utilities, will service individual homes and offices across the country.*

Cette vision de l'informatique utilitaire prédisait l'évolution majeure de l'industrie informatique de ce début de siècle. Basé sur le principe de fournir les ressources informatiques sous forme de services et de facturer leur utilisation en fonction de leur usage (*pay as you go* en anglais), le Cloud Computing permet d'effectuer des économies d'échelle grâce à l'externalisation de ces ressources vers des fournisseurs spécialisés. Ainsi, de nombreuses offres de Cloud ont vu le jour. C'est le cas par exemple de Google App Engine [13], Amazon EC2 [1] ou Microsoft Azure [36], les offres permettant d'utiliser les infrastructures de calcul et de communication de Google, Amazon ou Microsoft comme des services utilitaires.

Ces offres proposent différents types de service généralement spécifiques à l'usage que les utilisateurs peuvent en faire. Une architecture en couches a été proposée pour catégoriser les différents types de services et les usages correspondants. Cette architecture est aussi appelée modèle SPI [40] (*Software* / *Platform* / *Infrastructure* voir Figure 1) :

- Infrastructure (*Infrastructure as a Service* (IaaS)) : cette couche offre un niveau d'abstraction par rapport aux infrastructures disponibles. Il est alors possible de déployer des instances de systèmes virtualisés personnalisés (plate-forme) selon les besoins de l'utilisateur. L'une des solutions les plus connues est celle proposée par Amazon [1] qui permet de déployer des machines virtuelles personnalisées.
- Plate-forme (*Platform as a Service* (PaaS)) : cette couche offre un niveau d'abstraction vis-à-vis de la plate-forme d'exécution applicative. La plate-forme d'exécution applicative a pour but de simplifier la conception, mais aussi le déploiement d'applications en simplifiant voire en minimisant l'interaction avec l'infrastructure et en offrant un ensemble de fonctionnalités directement intégrées. Parmi les offres de PaaS existantes, nous pouvons citer Google App Engine [13] qui permet de déployer des applications web et qui offre un ensemble d'APIs pour simplifier l'interaction avec différents éléments tels que les bases de données ou encore le

système de fichier.
- Application (*Software as a Service* (SaaS)) : cette couche offre un niveau d'abstraction vis-à-vis des services applicatifs en masquant leur condition d'exécution. Les services applicatifs correspondent à des applications déportées le plus souvent accessibles depuis des navigateurs web ou par l'intermédiaire d'APIs dans le cadre de leur utilisation depuis d'autres applications. Nous pouvons citer par exemple, Gmail pour la gestion de courrier électronique ainsi qu'Office 365 [23] et Google Drive [14] pour l'édition de document.

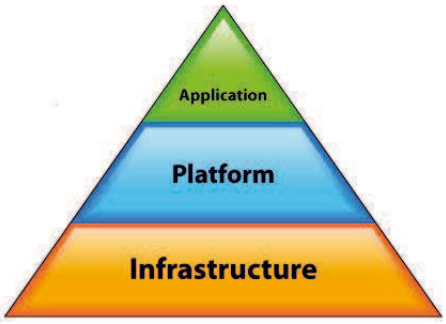


FIGURE 1 – Modèle SPI

Du fait de fournir un service à des utilisateurs qui paient pour celui-ci, les infrastructures, plates-formes et applications de Cloud Computing doivent assurer une disponibilité quasi totale de leur service tout en réduisant ou limitant les coûts pour les utilisateurs par rapport à un usage "non Cloud" des ressources informatiques. Dans ce but, elles utilisent le plus souvent des ressources spécialisées ainsi que des techniques de tolérance aux pannes tout en gérant, dynamiquement et sans intervention humaine, les ressources allouées en fonction des besoins des utilisateurs ou des applications. On parle d'élasticité des ressources. Un autre bénéfice envisagé est lié à l'informatique "écoresponsable" (*Green Computing* en anglais). Le Green Computing a pour objectif de limiter l'impact écologique des systèmes informatiques. Pour cela, le Cloud Computing et notamment la mutualisation des ressources permet d'envisager une gestion plus efficace de celles-ci et donc une gestion plus efficace de la consommation énergétique [78].

Malgré les avantages offerts par le concept de Cloud Computing, celui-ci intègre aussi quelques défauts. Par exemple, la dynamique de la tarification rend difficile l'estimation du coût pour une entreprise utilisatrice d'offres de Cloud. De plus, les solutions techniques utilisées dans les offres de Cloud sont généralement "*vendor locking*", c'est-à-dire qu'elles enferment l'utilisateur dans la solution technique choisie rendant difficile la migration d'une solution à une autre. Répondre à certains de ces défauts est l'objectif de ce que l'on appelle le Sky Computing [83] ou fédération de Cloud dont l'objectif est de pouvoir faire collaborer des solutions de même niveau.

Problématiques

De nombreuses problématiques de recherche sont encore ouvertes pour conforter ce modèle de l'informatique utilitaire. Par exemple, au niveau infrastructure, fournir des couches de virtualisation de plus en plus efficaces reste un défi en constante étude. Offrir des capacités d'observation fines des ressources consommées au niveau de l'infrastructure est là aussi une problématique scientifique non encore complètement résolue. Au niveau plate-forme, fournir une couche efficace de support de l'applicatif, prenant en charge de manière transparente ou déclarative de nombreux services techniques en limitant les hypothèses faites sur l'infrastructure est aussi un domaine de recherche toujours d'actualité. Au niveau applicatif, adapter les techniques de conception logiciel au développement d'applications pour le Cloud en bénéficiant au mieux des fonctionnalités offertes par la plate-forme ou migrer automatiquement des applications existantes sur le Cloud sont aussi des sujets toujours en cours d'étude.

Outre ces problématiques que l'on peut isoler par couche dans le modèle SPI, d'autres problématiques s'étudient de manière transverse. Pour pouvoir faire de la tolérance aux pannes, de l'élasticité de ressources et du green computing, des systèmes d'adaptation sont généralement utilisés afin de reconfigurer pendant son exécution le système qui dans notre cas correspond à une infrastructure, une plate-forme ou une application. Les mécanismes d'adaptation au niveau infrastructure utilisent les capacités de celui-ci pour le reconfigurer (migration de machine virtuelle, arrêt de serveur, ...). Pour les autres niveaux, ceux-ci utilisent des capacités qui leur sont propres, mais aussi les fonctionnalités offertes par le niveau sous-jacent. Par exemple, la plate-forme peut, par l'intermédiaire de l'infrastructure, créer de nouvelles machines virtuelles. La manière d'utiliser ces mécanismes de reconfiguration fournis par chacune des offres et chacun des niveaux du modèle SPI est souvent propre à chacune d'elles. Par exemple, l'API EC2 pour Amazon EC2 ou les APIs Google pour Google App Engine sont dédiés à ces offres. Ce sont ces spécificités qui font le " vendor locking" et qui rendent difficile la collaboration des infrastructures, des plates-formes ou des applications.

Cependant, outre ce "vendor locking", même si les différents niveaux peuvent interagir entre eux, chaque niveau est indépendant des autres et les reconfigurations de chacun des niveaux ne tiennent pas compte de l'évolution des autres alors que l'adaptation devrait être une préoccupation transverse. En effet, dans [84], les auteurs définissent plusieurs modes d'adaptation (configuration, optimisation, réparation, protection) et expliquent que ce sont des modes indépendants les uns des autres, mais que de manière optimale il serait préférable de pouvoir les intégrer les uns aux autres. De la même façon, il est nécessaire d'envisager l'adaptation comme une problématique globale à l'ensemble des niveaux. C'est ce que Google a mis en place pour son service de messagerie (Gmail) [15] avec une infrastructure, une plate-forme et une application spécialisées et optimisées pour s'exécuter ensemble.

C'est la maîtrise totale de son système informatique qui permet à Google de mettre en place ce genre de solution lui permettant de faire cohabiter l'infrastructure, la plate-forme et l'application afin d'assurer une adaptation globale cohérente et efficace. Mais la majorité des utilisateurs de Cloud n'ont pas la possibilité d'avoir une maîtrise totale

des solutions techniques. Par exemple si nous prenons CloudBees [3] qui propose de concevoir des plates-formes spécialisées selon les besoins des applications, CloudBees utilise l'infrastructure Amazon EC2 et ne peut donc pas gérer celle-ci pour effectuer de l'adaptation cohérente et efficace et est seulement capable d'utiliser l'infrastructure pour adapter ces plates-formes. C'est pourquoi il est nécessaire de disposer d'abstractions pour regrouper suffisamment d'informations sur l'ensemble des niveaux pour tirer parti des autres niveaux, mais aussi pour coordonner les adaptations entre les niveaux.

C'est la problématique principale visée par cette thèse, comment offrir aux fournisseurs de services de Cloud une vue cohérente de l'ensemble des niveaux du modèle SPI leur permettant de concevoir efficacement leurs services et de raisonner de manière efficace sur leur système, c'est-à-dire l'ensemble des niveaux du modèle SPI, lui permettant aussi de reconfigurer de manière cohérente l'infrastructure, la plate-forme et les applications en fonction des besoins de ces derniers.

Contribution

Cette thèse propose une abstraction permettant de gérer/coordonner l'adaptation entre les différents niveaux, mais aussi permettant la collaboration entre des solutions techniques de même niveau. Cette abstraction sert aussi de support à la conception de nouvelles solutions de Cloud que ce soit pour l'infrastructure, la plate-forme ou les applications.

Une abstraction regroupe un ensemble d'information correspondant à un point de vue particulier d'un système. Ici ce système correspond à l'ensemble des différents niveaux qui existent dans un Cloud et le point de vue qui nous intéresse correspond à la configuration architecturale de ces niveaux avec par exemple la représentation de l'ensemble des ressources matérielles de l'infrastructure, l'ensemble des environnements d'exécution d'une plate-forme et les différents éléments correspondants aux applications. L'intérêt d'avoir une abstraction de la configuration architecturale réside dans la possibilité de se servir de cette abstraction comme support de communication dans le cadre de la coordination de l'adaptation [61]. En effet, chacun des systèmes d'adaptation peut utiliser cette abstraction pour proposer des reconfigurations et ces reconfigurations seront comprises par n'importe quels systèmes d'adaptation si lui-même utilise cette abstraction. De la même manière, l'utilisation de cette abstraction comme support de raisonnement pour les systèmes d'adaptation permet la portabilité des systèmes de raisonnement sur différentes implémentations d'infrastructures, de plates-formes ou d'applications équivalentes dans cette abstraction.

Au sein de cette abstraction sont définis les différents concepts permettant de spécifier les services présents aux différents niveaux. Ainsi, cette abstraction permet de représenter les différents niveaux, mais est aussi capable de modéliser la distribution de chacun des niveaux, avec par exemple le nombre de machines virtuelles faisant parties d'une plate-forme. Cette abstraction est construite de manière extensible afin de faciliter l'intégration de futures caractéristiques pouvant être importantes dans la représentation d'un Cloud. C'est aussi cette extensibilité qui permet de représenter l'hétérogénéité des

implémentations que ce soit au niveau infrastructure, plate-forme ou application en permettant de représenter les caractéristiques spécifiques à chacune des implémentations. De cette façon, il est possible de représenter des Clouds hétérogènes et d'envisager des reconfigurations coordonnées entre les différentes implémentations.

Cette abstraction est fondée sur des concepts et outils issus de l'Ingénierie Dirigée par les Modèles (IDM) ou Model Driven Engineering (MDE) [110] et plus particulièrement des travaux autour des modèles à l'exécution ou Model at Runtime (M@R) [44]. L'IDM prône le développement logiciel par l'intermédiaire d'abstractions appelées modèles représentant les différentes vues d'un système et permettant de générer un système concret à partir des définitions de ces différentes vues. Le M@R quant à lui propose de porter, à l'exécution, les techniques et outils de l'IDM utilisés normalement durant la phase de conception. Ainsi, il est possible de modifier les modèles représentant le système et impacter ces changements sur le système en cours d'exécution tout en bénéficiant des outils de validation existants (l'analyse statique par exemple) autour des modèles afin d'assurer la cohérence du nouveau modèle.

J'ai utilisé l'approche M@R appelée Kevoree [1] qui proposait déjà de nombreuses propriétés pour représenter les différents niveaux de Cloud et notamment les applications et les plates-formes. J'ai étendu cette approche dans une extension appelée KevoreeKloud afin d'y intégrer les concepts nécessaires à la gestion des différents niveaux de Cloud et notamment nécessaires à la représentation du niveau infrastructure et aux relations entre les niveaux. Ces concepts gravitent autour de l'idée de nœud qui représente un conteneur d'exécution. Ces conteneurs d'exécution peuvent être, au niveau de l'Infrastructure, des conteneurs de machines virtuelles correspondant à une ou plusieurs plates-formes ou au niveau Plate-forme, des conteneurs d'applications. Ces nœuds sont les supports de l'adaptation en fournissant les primitives d'adaptation nécessaires à la reconfiguration de leur contenu. Ils fournissent aussi les outils permettant d'assurer la validité des adaptations avec notamment la planification de l'exécution des adaptations pour assurer que l'adaptation soit efficace et surtout cohérente. En plus des outils utilisés à l'exécution, Kevoree propose aussi un ensemble de framework pour la conception de nouvelles solutions logicielles que j'ai utilisé et étendu pour faciliter la conception de nouvelles implémentations de Cloud que ce soit pour le niveau infrastructure ou plate-forme.

Organisation du document

Ce document est organisé comme suit.

Tout d'abord, le chapitre 1 présente le contexte dans lequel s'inscrivent les travaux présentés dans cette thèse. Ce contexte concerne le domaine du Cloud Computing avec ses origines depuis l'informatique utilitaire en passant par les grilles de calculs, les différents types de Cloud qui existent à l'heure actuelle ainsi que les différents types d'utilisateurs visés par les solutions de Cloud Computing et les besoins qui leur sont reliés. La gestion de ces besoins se fait généralement de manière dynamique et nécessite

1. http://kevoree.org

de pouvoir reconfigurer le Cloud et ses différents éléments. Ainsi, le contexte concerne aussi le domaine de l'adaptation logicielle pour lequel est proposé une définition de ce qu'est l'adaptation et comment celle-ci doit être prise en compte.

Le chapitre 2 présente l'état de l'art des frameworks et des modèles permettant de modéliser et piloter des systèmes logiciels. L'étude de ces frameworks et modèles se fera par rapport à un ensemble de critères permettant la représentation des différents éléments d'un Cloud et offrant une abstraction permettant de gérer l'adaptation de manière transverse aux différents éléments du Cloud.

Le chapitre 3 présente en détails une abstraction appelée Kevoree et utilisant les principes de modèles à l'exécution pour la conception et l'adaptation de systèmes distribués large échelle. Les travaux de cette thèse ont été intégrés à cette abstraction afin de pouvoir modéliser un système de Cloud.

Le chapitre 4 présente les travaux de cette thèse autour d'une abstraction pour faire de l'adaptation multi-niveaux sur du Cloud. Ces travaux sont présentés sous la forme d'une extension à Kevoree permettant de modéliser l'ensemble des éléments d'un Cloud, d'un framework de développement facilitant la conception de solution de Cloud ainsi que l'intégration d'implémentations existantes et facilitant la réutilisation de différents algorithmes nécessaires à la gestion d'un Cloud.

Le chapitre 5 présente la validation de cette abstraction au travers de cas d'études. Le premier concerne l'implémentation d'un Cloud hybride afin de montrer l'intérêt de KevoreeKloud, de son extension et du framework associé pour la conception de nouvelle solution de Cloud. Le second cas d'étude permet d'observer l'impact en terme de temps de l'utilisation de KevoreeKloud dans le cadre de la reconfiguration d'un Cloud ainsi que l'impact sur l'utilisation mémoire. Enfin, le dernier cas d'étude permet de montrer comment notre abstraction permet de partager une vision globale du Cloud entre les différents niveaux de celui-ci facilitant la coopération entre les systèmes d'adaptations afin d'effectuer de l'adaptation cohérente et efficace.

Enfin, le chapitre 6 conclut ce document et présente les perspectives des travaux présentés.

Deuxième partie

État de l'art

Chapitre 1

Contexte

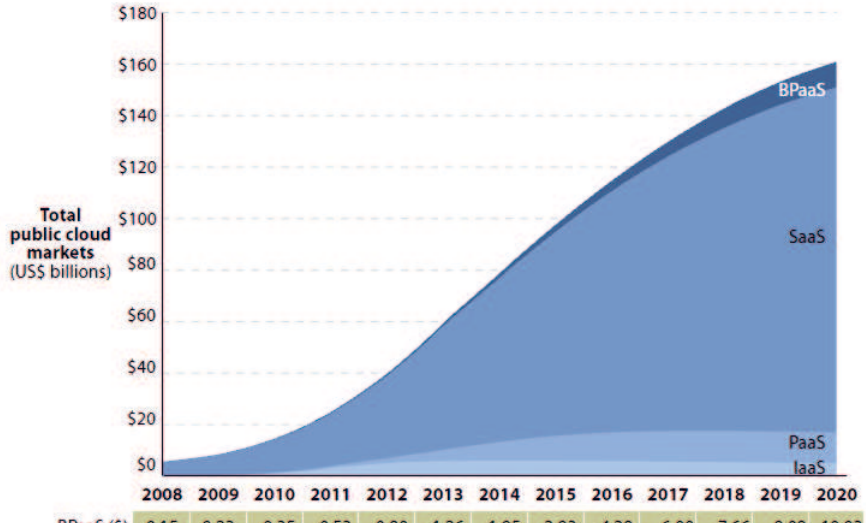
Ce chapitre présente plus en détail les deux domaines dans lesquels s'intègrent les travaux de cette thèse. Tout d'abord, en section 1.1, nous définissons ce qu'est le Cloud Computing et les nombreuses solutions existantes montrant l'importance et la diversité que l'on peut y trouver. Ensuite, la section 1.2 présente le domaine de l'adaptation devenue une caractéristique fondamentale des systèmes informatiques actuels.

1.1 Cloud Computing

Le concept de Cloud Computing ou "informatique dans les nuages" consiste à fournir les ressources informatiques sous forme de services pour lesquels l'utilisateur paie pour ce qu'il utilise. Ce concept est apparu dans les années 60 notamment avec McCarthy [69] ou encore Kleinrock [89] sous le nom d'informatique utilitaire. C'est ensuite vers la fin des années 90 que ce concept a réellement pris de l'importance avec tout d'abord le Grid Computing [66]. Ce terme est une métaphore exprimant la similarité avec le réseau électrique dans lequel l'électricité est produite dans de grandes centrales puis disséminée au travers d'un réseau jusqu'aux utilisateurs finaux. Ici les grandes centrales sont les DataCenters, le réseau est le plus souvent celui d'Internet et l'électricité correspond aux ressources informatiques. Le terme *Cloud Computing* n'est véritablement apparu qu'au cours des années 2006-2008 [116] avec l'apparition d'Amazon EC2 [1] ou encore la collaboration d'IBM et Google [11, 12] ainsi que l'annonce d'IBM concernant 'Blue Cloud' [17]. Par la suite de nombreuses solutions open source ont aussi vu le jour avec par exemple OpenShift [27] de RedHat, ou encore OpenStack [28] de RackSpace et en collaboration avec la NASA. Le marché du Cloud Computing n'en est encore qu'à ces débuts et d'après une étude menée par Forrester [106], alors que le marché du Cloud Computing s'élevait à environ 5,5 milliards de dollars en 2008, à un peu plus de 23 milliards en 2011, il devrait atteindre plus de 150 milliards d'ici 2020 (voir figure 1.1).

Il existe différents types de Cloud (voir figure 1.2) :

- les Clouds **publics** correspondent à des solutions de Cloud proposées par des prestataires de services. Ces solutions sont le plus souvent accessibles par un modèle de paiement en fonction de l'utilisation. Nous pouvons citer parmi les



	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
BPaaS ($)	0.15	0.23	0.35	0.53	0.80	1.26	1.95	2.93	4.28	6.00	7.66	9.08	10.02
SaaS ($)	5.56	8.09	13.40	21.21	33.09	47.22	63.19	78.43	92.75	105.49	116.39	125.52	132.57
PaaS ($)	0.05	0.12	0.31	0.82	2.08	4.38	7.39	9.80	11.26	11.94	12.15	12.10	11.91
IaaS ($)	0.06	0.24	1.02	2.94	4.99	5.75	5.89	5.82	5.65	5.45	5.23	5.01	4.78

FIGURE 1.1 – Évolution du marché du Cloud Computing

solutions existantes, celle d'Amazon ou encore Google App Engine (GAE) ainsi que Gmail, Microsoft SkyDrive.

- les Clouds **privés** correspondent à des solutions de Cloud mises en place au sein d'une entreprise et non accessibles aux personnes extérieures à celle-ci. Contrairement aux Clouds publics, l'utilisation d'un Cloud privé par une entreprise nécessite un investissement sur les ressources matérielles ainsi que sur la maintenance. L'un des avantages mis en avant dans l'utilisation d'un Cloud privé concerne principalement la sécurité des données, car celles-ci restent internes à l'entreprise contrairement à un Cloud public. En effet, la sécurité des données sur un Cloud public reste encore actuellement une problématique importante pour une entreprise souhaitant passer ses applications dans le Cloud.
- les Clouds **hybrides** quant à eux correspondent à une composition de plusieurs Clouds. L'utilisation de solutions hybrides permet aux entreprises d'assurer un certain niveau de tolérance aux fautes en ayant accès à des ressources extérieures lorsque les ressources internes ne sont plus suffisantes. Généralement, les Clouds hybrides sont basés sur les mêmes solutions de Cloud et la mise en place du Cloud privé est généralement effectuée par le fournisseur du Cloud public. C'est par exemple le cas avec ElasticHosts et sa solution ElasticStack [8, 7].

En plus de ces trois types énoncés ci-dessus, il existe aussi le ***Sky Computing*** [83] que l'on appelle aussi **fédération de Clouds** qui correspond à l'interconnexion de plusieurs Clouds. Le *Sky Computing* peut être vu comme la suite du Cloud Computing où il est possible de passer d'une solution de Cloud telle que celle d'Amazon à une autre

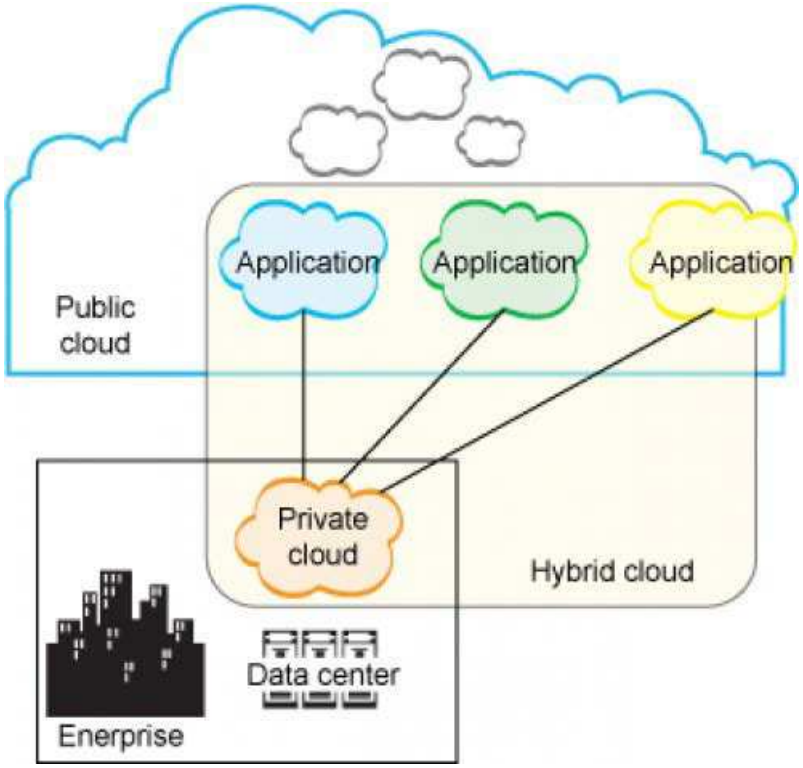


FIGURE 1.2 – Les différents types de Cloud

comme celle de Google voir d'utiliser les deux en même temps. Dans [83], les auteurs expliquent que les defis de ce genre de solution concernent entre autres l'interopérabilité des solutions de Cloud ainsi que la création de réseaux d'interconnexion sûrs et performants entre machines virtuelles.

Outre les différents types de Cloud, le Cloud Computing a entrainé une évolution des rôles des personnes interagissant avec ce nouveau type de système d'information. Plusieurs personnes de chez IBM ont proposé une définition des interactions possibles et des rôles associés [45]. Ces rôles sont catégorisés par trois ensembles. Cette approche définit tout d'abord la catégorie des **concepteurs** qui définissent et construisent des services. Ces services peuvent être de niveau Infrastructure, Plate-forme ou Application et leurs définitions correspondent non seulement à la définition des fonctionnalités, mais aussi à leurs caractéristiques non fonctionnelles comme le prix auquel ces services seront proposés. C'est aussi dans cette catégorie que l'on retrouve le rôle de composition de service pour former un service de plus haut niveau.

Cette approche introduit ensuite la catégorie des **utilisateurs** qui consomment les services proposés par un Cloud que ce soit en tant qu'application à part entière comme les services de messagerie ou en tant que services intégrés dans des applications "non-Cloud" par exemple un service de stockage de données ou un service de base de données. Cette catégorie regroupe aussi les acteurs chargés du lien entre fournisseurs et utilisa-

teurs finaux par exemple dans le cadre de l'évaluation de la sécurité des services fournis.

Enfin, cette approche introduit la catégorie des **fournisseurs** qui mettent en place les services et assurent leur disponibilité ainsi que leur bon fonctionnement. Sont regroupées dans cette catégorie, les personnes chargées de faire le lien entre les concepteurs et les utilisateurs que ce soit pour le déploiement des services, la gestion du paiement à l'usage, l'assistance aux utilisateurs dans le cadre de l'évaluation des services à utiliser selon leurs besoins, la gestion du matériel physique (machines, câble, système de refroidissement, ...), la gestion des défaillances, la surveillance des services pour évaluer leurs usages et informer des défaillances, la gestion des aspects techniques qui ne sont pas reliés aux services (le réseau par exemple), la gestion de l'énergie. De manière générale, ce sont les fournisseurs qui sont chargés de la gestion des propriétés non fonctionnelles liées aux services.

C'est la gestion de ces propriétés non fonctionnelles qui fait l'intérêt du Cloud Computing et notamment la gestion de l'élasticité. L'**élasticité** consiste à pouvoir ajouter ou supprimer dynamiquement des ressources en fonction des besoins. En effet, Armbrust *et al.* expliquent dans [40] que dans le cadre d'un système informatique standard, l'entreprise se doit d'estimer la charge possible que va subir son système afin de pouvoir estimer au plus près la taille des ressources nécessaires. Cependant, cette estimation qui tient compte de la charge maximale signifie que hormis lors des pics d'utilisation, les ressources prévues ne seront pas utilisées d'où une perte économique pour l'entreprise. De la même façon si la charge maximale a été sous-estimée, certains utilisateurs ne pourront pas accéder aux services fournis par l'entreprise ce qui peut avoir un effet néfaste sur les revenus de celle-ci non seulement à court terme, car les utilisateurs ne peuvent pas utiliser les services, mais aussi à long terme, car ils ne voudront plus utiliser ces services. Le fait de pouvoir dynamiquement faire évoluer les ressources informatiques utilisées afin de cadrer au maximum avec les besoins réels permet ainsi de limiter les coûts inutiles dus au surdimensionnement des ressources. Cela permet d'éviter des pertes de bénéfices qui pourraient être dues à un sous-dimensionnement des ressources. Cette capacité s'est révélée très utile dans le cadre de certaines *startups* ou petites entreprises souhaitant lancer un nouveau service, car il est difficile de prévoir l'engouement que peut produire un nouveau produit sur les consommateurs. Ce fut par exemple le cas avec l'entreprise Animoto [2] qui en l'espace d'un mois a vu son nombre d'utilisateurs passer de 25 000 à 250 000 et qui grâce à l'utilisation d'Amazon EC2 a pu supporter cette montée en charge sans pour autant investir dans des infrastructures grâce à la gestion de l'élasticité fournie par Amazon EC2.

Même si ces rôles sont distincts, dans la réalité, une personne endosse généralement plusieurs rôles. Par exemple, la gestion des propriétés non fonctionnelles telles que l'élasticité des services applicatifs dépend de la conception des services sous-jacents c'est-à-dire des services de la plate-forme voire des services de l'infrastructure. De ce fait, un fournisseur est souvent aussi un concepteur. Dans cette thèse ce sont les rôles de concepteur et de fournisseur qui nous intéressent et pour lesquelles nous souhaitons fournir des solutions concernant la gestion de l'adaptation dans un Cloud.

1.2 Adaptation

Dans le début des années 2000, IBM a publié un manifeste indiquant que l'obstacle à la progression de l'industrie du logiciel correspondait à la gestion de la complexité des systèmes logiciels [80]. Dans ce document, les auteurs pointaient du doigt la difficulté de gérer les systèmes informatiques en tant qu'ensemble d'applications distinctes, car celles-ci interagissaient de plus en plus entre elles. Les systèmes devenant de plus en plus interconnectés, il devenait compliquer pour les architectes logiciels de prévoir et de définir ces interactions. De la même façon, les systèmes étant de plus en plus importants et complexes, il devenait difficile de les installer, de les configurer, de les optimiser et de les maintenir. Enfin, l'évolution des besoins sur ces systèmes évoluant rapidement, il devenait difficile de les faire évoluer rapidement pour répondre à ces besoins.

Dans le but de pallier cette complexité, le concept d'adaptation d'entités logicielles a été introduit [51, 107]. L'idée de l'adaptation dynamique est de pouvoir modifié à l'exécution, un logiciel en fonction de l'évolution de l'environnement dans lequel il s'exécute. Il existe 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 de l'entité (par exemple, migrer l'entité d'une machine à une autre).

Le modèle MAPE pour *Monitoring, Analysis, Planning, Execution* [84] est une abstraction proposée par Jeffrey O. Kephart et David M. Chess qui formalise la conception d'un système d'adaptation. 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.

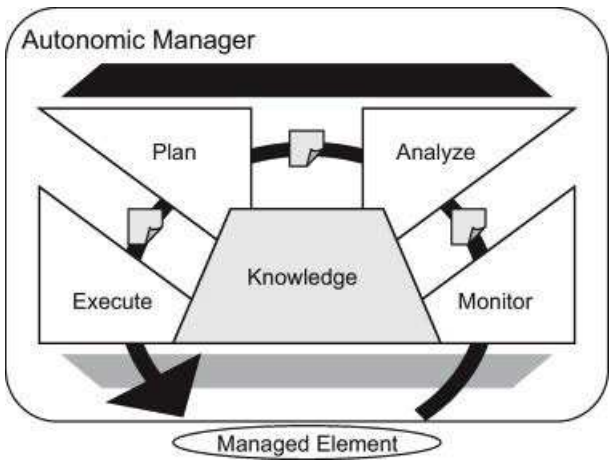


FIGURE 1.3 – Boucle autonomique correspondant au modèle MAPE

Dans leur définition de l'adaptation qui nous suivra dans cette thèse, l'adaptation y est décomposée en quatre phases (voir figure 1.3). La première est l'**observation**, qui consiste à surveiller 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...). Lorsque la phase d'observation détecte des changements, elle notifie la phase suivante : l'**analyse** (ou décision). Celle-ci 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. Une fois une stratégie d'adaptation choisie, elle est envoyée à la troisième phase qu'est la **planification**. Cette phase est chargée de décomposer la stratégie d'adaptation choisie par la phase d'analyse en un plan d'actions élémentaires. Ce plan d'actions élémentaires consiste en une liste d'action partiellement ordonnée qui permet de passer de l'état courant du système à l'état souhaité. C'est la dernière phase appelée **exécution** qui se charge d'appliquer ces actions selon le plan défini par la planification.

Le modèle MAPE est parfois appelé MAPE-K afin de mettre en évidence la partie **base de connaissances** permettant à chacune des phases d'accéder aux informations nécessaires à son bon fonctionnement.

En plus du modèle MAPE, Kephart *et al.* ont extrait quatre propriétés d'adaptation (self-*) représentant les différents besoins que peut résoudre l'adaptation :

- l'auto-configuration (Self-configuring en anglais) qui signifie que le système doit lui-même être capable de définir et composer son architecture en fonction des variations de sa plate-forme d'exécution et de son environnement.
- l'auto-réparation (Self-healing en anglais) qui signifie que le système doit être capable de diagnostiquer et résoudre un problème interne de fonctionnement.
- l'auto-optimisation (Self-optimizing en anglais) qui signifie que le système doit être capable d'adapter ses ressources pour optimiser son fonctionnement.
- l'auto-protection (Self-protecting en anglais) qui signifie que le système doit détecter et se protéger des attaques et accès extérieurs.

L'adaptation est souvent associée au principe de **réflexion** [91]. Un système est dit réflexif s'il possède la capacité de raisonner et d'agir sur lui-même par le biais d'une représentation de lui-même. Le principe de réflexion comprend deux capacités complémentaires :

- l'**introspection** qui correspond à la capacité de pouvoir s'observer soi-même. C'est cette capacité qui permet de définir une représentation du système.
- l'**intercession** qui correspond à la capacité de pouvoir se modifier soi-même permettant ainsi de changer, pendant l'exécution, la configuration du système.

Un système réflexif comporte deux niveaux que sont le système lui-même qui est en cours d'exécution et sa représentation aussi appelé méta-niveau [86]. Ces deux niveaux sont causalement liés ce qui signifie qu'une modification sur le système en cours d'exécution entraîne la modification de sa représentation et inversement.

On distingue deux aspects complémentaires dans la réflexion :

- la réflexion structurelle permet d'observer et de manipuler la façon dont le logiciel est organisé en termes de fonctionnalités et d'interactions entre ces fonctionnalités.

Dans le cas des langages à objets, cela correspond à la connaissance de la classe d'un objet, de ses variables d'instance et de ses méthodes et cela correspond aussi à la capacité de pouvoir les modifier (en changeant la valeur d'une variable d'une instance pour remplacer une fonctionnalité utilisée par l'objet).
- la réflexion comportementale permet de manipuler l'exécution du programme en ayant connaissance des méthodes en cours d'exécution et en pouvant intercepter certains appels pour les surcharger.

La plupart des solutions offrant une gestion de l'adaptation dynamique des systèmes logiciels combinent l'utilisation du modèle MAPE pour la définition du système d'adaptation avec les capacités d'un système réflexif. Ainsi, les capacités d'introspection permettent de contribuer à la connaissance du système à adapter et donc à son observation et l'intercession fournit les capacités d'adaptation pouvant être utilisées sur le système pour le reconfigurer.

Dans la suite du document, nous distinguons l'adaptation de la reconfiguration. L'adaptation définit l'ensemble du processus représenté par le modèle MAPE alors que la reconfiguration se focalise sur la phase de planification et d'exécution et qui sert à appliquer une nouvelle configuration.

1.3 Adaptation et Cloud : un besoin d'abstraction

Kephart et al ont affirmé que bien qu'il existe différents types d'adaptation (self-*), ceux-ci devraient être pris en compte de manière coordonnée [84]. En effet, chacun peut avoir un impact sur le système et donc par effet de bord, un impact sur les autres. De la même façon dans un Cloud, même si les adaptations de l'infrastructure, de la plate-forme ou de l'application visent des buts différents et manipulent des entités distinctes, chacune a un impact sur le système et cet impact peut avoir des conséquences sur les autres. Par exemple, il est possible que l'infrastructure choisisse de déconnecter l'une de ses ressources. Pour cela, elle va devoir migrer l'ensemble des plates-formes en cours d'exécution sur cette ressource. Dans ces plates-formes, des services d'une ou plusieurs applications vont donc être réalloués sur d'autres ressources et ce changement de localisation même s'il n'a pas d'impact sur le fonctionnement des applications et des plates-formes peut tout de même avoir un impact sur la qualité de service de l'application, car le délai de communication entre les services migrés et les autres peut avoir changé. C'est pourquoi il est nécessaire de voir l'adaptation comme une problématique transverse afin de pouvoir coordonner l'adaptation de chacun des niveaux et même envisager de l'adaptation entre plusieurs niveaux.

Afin d'effectuer de l'adaptation coordonnée entre les différents niveaux, deux possibilités sont envisageables. La première consiste à ne définir qu'un seul système d'adaptation capable d'adapter l'ensemble des niveaux et dans lequel sont insérées dynamiquement de nouvelles politiques d'adaptation en fonction par exemple des nouvelles applications qui s'exécutent. La deuxième solution consiste à coordonner l'ensemble des systèmes d'adaptation entre eux en faisant, par exemple, valider les choix d'adaptation de l'un des systèmes par les autres ou juste notifier l'ensemble des systèmes d'adaptation afin par exemple que ceux-ci patientent pour voir l'effet de l'adaptation avant d'eux-mêmes en définir une nouvelle si besoin est.

Pour faciliter la conception de ces systèmes d'adaptation, il est nécessaire d'avoir un support d'adaptation commun aux différents niveaux. En effet, dans le cadre d'un système d'adaptation unique, le fait d'utiliser des supports d'adaptation différents entre les niveaux va complexifier voire rendre impossible la définition des adaptations multi-niveaux. Il serait également impossible de faire collaborer plusieurs systèmes d'adaptation puisqu'ils n'utiliseraient pas le même langage. Dans d'autres domaines, ce besoin d'abstraction commune a déjà été mis en avant par exemple dans le cadre d'application pour grille de calcul [38].

Que ce soit un système d'adaptation unique ou un ensemble de systèmes d'adaptation collaborant entre eux, la conception de ces entités est à la charge des fournisseurs de services de Cloud. Ce rôle de fournisseur ressemble sensiblement au rôle d'administrateur de bases de données [43] qui a la charge d'assurer le bon fonctionnement des bases de données dans une entreprise. Pour cela, il doit être capable d'installer, de configurer, surveiller, mettre à jour et maintenir les bases de données en cours de fonctionnement. La majorité des opérations nécessaires à ces activités sont fournies au travers d'une abstraction spécifique qui correspond au langage SQL [53] pour les bases relationnelles. C'est ce genre d'abstraction qui est nécessaire pour les fournisseurs de services de Cloud.

L'une des premières solutions pour définir ce genre d'abstraction pourrait être de mettre en commun l'ensemble des APIs fournies sur chaque niveau. Cependant, les solutions d'infrastructure, de plate-forme et d'application proposent chacune un ensemble d'API le plus souvent spécifiques enfermant les utilisateurs dans une technologie particulière et rendant difficile la migration vers une autre technologie. Cet enfermement est aussi une faiblesse pour la conception d'un ou plusieurs systèmes d'adaptation multi-niveaux, car ce ou ces systèmes, bien qu'indépendant des niveaux, seraient dépendants des solutions qui les implémentent. En effet, il est par exemple possible de définir une abstraction pour l'ensemble des niveaux de Cloud proposés par Google (Google Compute, Google App Engine, Google App). Cette abstraction correspondrait à la somme de l'ensemble des APIs proposées par chacun des niveaux. De ce fait, il est envisageable de concevoir un système d'adaptation globale ou plusieurs systèmes d'adaptation interopérables. Cependant, il serait alors nécessaire de bâtir une abstraction spécifique pour chaque composition de solutions de Cloud. Par exemple, il faudrait une solution spécifique pour le Cloud composé de l'infrastructure Eucalyptus, de la plate-forme AppScale et de l'application SparkleShare et il faudrait une autre solution pour un Cloud composé de l'infrastructure ElasticStack, de la plate-forme CloudFoundry et de l'application NetSuite ERP.

C'est pourquoi il est nécessaire d'avoir une abstraction pouvant modéliser les caractéristiques d'un niveau et donc de l'ensemble des solutions existantes pour ce niveau et cela pour chacun des niveaux qui existent pour permettre aux systèmes d'adaptations de se coordonner grâce à ce support de communication commun.

Chapitre 2

État de l'art

Le précédent chapitre a mis en avant le besoin de gérer l'adaptation d'un système de Cloud de manière transverse entre les niveaux afin d'assurer que l'adaptation soit efficace et cohérente. Pour pouvoir faire cela, nous avons aussi mis en évidence le potentiel bénéfice à modéliser, de manière commune, les différents niveaux que l'on peut trouver sur un Cloud comme l'infrastructure, la plate-forme et les applications. Ce modèle vise à permettre à un fournisseurs en charge d'administrer un Cloud de définir ces politiques d'adaptations face à un contexte en constante évolution.

Nous présentons, dans ce chapitre, les différentes solutions qui proposent un support abstrait pour la définition et l'adaptation de système informatique et les comparons aux caractéristiques requises pour permettre non seulement de modéliser un Cloud, mais aussi de pouvoir utiliser cette abstraction comme support de communication entre niveaux dans le cadre de l'adaptation.

Pour cela, nous définissons en section 2.1, les différents critères nécessaires à la représentation abstraite d'un Cloud. Par rapport à ces différents critères, nous présentons un exemple simple justifiant les différents critères que nous avons mis en avant. Puis nous présentons, en section 2.3, un ensemble de solutions permettant de définir des applications. De la même façon, nous présentons, en section 2.4, des solutions pour les systèmes de Cloud ou de grilles de calcul. Enfin, nous concluons, en section 2.5, par une synthèse de ce chapitre en mettant en évidence le manque d'abstraction des propositions existantes par rapport à ces propriétés requises pour fournir une abstraction à un fournisseur de service de Cloud.

2.1 Les caractéristiques d'une abstraction pour administrer le Cloud

Pour pouvoir adapter un système de Cloud non plus comme un ensemble de niveaux indépendants, mais bien comme un système complexe, il est nécessaire de disposer d'une abstraction globale capable de modéliser l'ensemble des éléments d'un Cloud. Nous présentons dans cette section, un certain nombre de caractéristiques nécessaires pour modéliser ces éléments et pouvoir effectuer de l'adaptation sur un Cloud.

Multi-niveaux Vouloir gérer l'adaptation comme une problématique transverse entre les niveaux impose de pouvoir faire collaborer les systèmes d'adaptation de chacun des niveaux ou de pouvoir concevoir un système d'adaptation capable de manipuler les différents niveaux simultanément. C'est pourquoi il est nécessaire de pouvoir représenter ensemble les applications, les plates-formes et les infrastructures définissant un Cloud.

Hébergement Un système de Cloud Computing, ce n'est pas seulement des applications, des plates-formes et des infrastructures, mais c'est aussi un agencement de ces applications, plates-formes et infrastructures. C'est pourquoi il est nécessaire que l'abstraction puisse modéliser la notion d'hébergement que ce soit une machine physique de l'infrastructure qui héberge une machine virtuelle de la plate-forme ou une machine virtuelle de la plate-forme qui héberge une application. Disposer de ces informations permet à un système d'adaptation de réaliser des migrations de composants applicatifs entre machines virtuelles d'une même plate-forme ou des migrations de machines virtuelles d'une plate-forme entre machines physiques d'une infrastructure.

Distribution Un Cloud est généralement implanté sur des réseaux de machines (physiques ou virtuelles). De ce fait, l'une des principales informations concernant ce système réside dans la configuration de ce réseau. La configuration de ce réseau correspond à l'interconnexion des différentes machines et permet de tenir compte de la répartition des différents éléments du système dans le cadre de l'adaptation et par exemple rapprocher deux composants sur la même machine virtuelle de la plate-forme afin de limiter la consommation de bande passante ou le délai de communication entre eux. D'une manière générale, Gerraoui *et al.* affirment qu'il est nécessaire d'avoir connaissance de cette notion de distribution afin de pouvoir en tirer parti [76]. C'est pourquoi il est important que cette notion de distribution soit capturée dans le modèle représentant un Cloud.

Hétérogénéité De la même manière que les composants possèdent différentes fonctionnalités ou différentes contraintes d'exécution, une plate-forme et une infrastructure peuvent être composées d'un ensemble de machines avec différentes caractéristiques. Par exemple, une plate-forme peut être composée de plusieurs machines virtuelles ayant chacune un système d'exploitation particulier afin de pouvoir déployer des applications spécifiques sur chacune d'entre elles. Pour l'infrastructure, ces différentes caractéristiques correspondent à des ressources matérielles différentes. De la même manière qu'il est nécessaire de pouvoir représenter les informations de distribution afin d'en tirer parti, il est aussi nécessaire de pouvoir représenter cette hétérogénéité au sein même d'une solution. Cet intérêt peut se traduire par exemple par la possibilité de concevoir des Clouds hybrides dont les implantations qui lui sont associées offrent chacune des capacités de virtualisation différentes.

Extensibilité Outre le fait de vouloir représenter de la même façon différentes implantations d'un même niveau et de pouvoir en même temps modéliser leurs caractéristiques

spécifiques, nous voulons que l'abstraction nous permette facilement d'intégrer une nouvelle solution. En effet, le domaine du Cloud étant en plein essor, il est fort probable que de nouvelles implantations apparaissent avec de nouvelles caractéristiques et de nouvelles fonctionnalités que ce soit pour l'infrastructure, la plate-forme, mais aussi pour les applications.

Réflexivité désynchronisable Comme nous l'avons dit précédemment (voir la fin de la section 1.2), les systèmes d'adaptation sont généralement définis au-dessus de systèmes réflexifs qui offrent une abstraction permettant d'observer et de modifier le système pendant son exécution. Cependant, ces systèmes réflexifs définissent un lien fort entre le programme et sa représentation abstraite. Ainsi, la modification de la représentation abstraite entraîne directement la modification du programme et inversement. Pourtant, la modification d'un système complexe n'est pas une tâche anodine et il est préférable de s'assurer de la validité du changement avant de l'appliquer afin d'éviter des états incohérents du système. C'est pourquoi nous pensons que l'abstraction qui sert de modèle réflexif au système doit pouvoir par construction être désynchronisable par rapport au système en cours d'exécution afin de valider ou simuler une nouvelle configuration avant de l'appliquer sur le système en cours d'exécution.

L'expression de ces différentes caractéristiques dans une même abstraction nous permettrait de modéliser, de manière complète, un Cloud. De ce fait, cette abstraction permettrait de définir des adaptations tenant compte de l'ensemble des niveaux afin qu'elles soient efficaces et cohérente. Nous allons utiliser ces caractéristiques du modèle de Cloud dans la suite de ce chapitre comme critères d'évaluation concernant l'utilisabilité d'un ensemble d'abstractions pour la représentation d'un Cloud.

2.2 Cas d'usage

L'un des principaux bénéfices attendus par un utilisateur de Cloud est l'élasticité (voir section 1.1). En effet, l'un des intérêts de migrer ses applications sur un Cloud est de pouvoir assurer aux utilisateurs une qualité de service suffisante sans pour autant devoir se charger de la gestion des ressources. Pour les applications, c'est bien la plate-forme d'hébergement (PaaS) qui est responsable de cette qualité de service et qui est capable de reconfigurer les applications par l'intermédiaire de duplication des éléments de l'application et de l'utilisation de techniques de load-balancing. Ainsi, un serveur web distribué sera défini comme un point d'accès que l'on peut appeler le serveur et les pages du ou des sites hébergés seront définies comme un ensemble d'entités indépendantes. Au niveau de l'infrastructure (IaaS), l'hébergement de nouvelles machines virtuelles pour la plate-forme en fonction de ses besoins peut tenir compte des relations entre les différents composants applicatifs afin de placer les différentes machines virtuelles sur la même machine physique de l'infrastructure pour limiter le coût des consommations réseau entre les machines virtuelles.

Cet exemple, bien que très simple montre le besoin d'avoir une vision globale du système pour que l'infrastructure puisse tenir compte des informations concernant les applications que ce soit les interactions entre les entités de l'application, mais aussi leur hébergement sur les plates-formes. Pour montrer la notion d'hétérogénéité, nous pouvons envisager que certaines pages du ou des sites servent pour l'observation du fonctionnement de la plate-forme sur lesquelles elles s'exécutent. Si cette page dépend du système d'exploitation pour effectuer ces observations, alors il est nécessaire que l'abstraction que nous utilisons soit capable de représenter cette spécificité pour que le gestionnaire de la plate-forme puisse correctement héberger cette page sur une machine virtuelle compatible. Le fait que les machines physiques et virtuelles puissent avoir ce genre de spécificité montre le besoin de pouvoir représenter l'hétérogénéité des solutions aussi bien au niveau infrastructure que plate-forme. Enfin, le fait d'offrir de la réflexivité désynchronisable permet de faire coopérer les différents systèmes d'adaptation s'exécutant sur différents niveaux (infrastructure et plate-forme notamment) au travers de l'abstraction et avant d'appliquer les reconfigurations.

Nous allons utiliser cet exemple dans la suite de ce chapitre pour évaluer, en fonction des caractéristiques citées précédemment, les différentes abstractions existantes.

2.3 Les abstractions pour l'adaptation logicielle

Les abstractions existantes pour faire de l'adaptation d'applications intègrent en grande partie les différentes étapes proposées dans le modèle MAPE et sont basées sur le principe de réflexion (voir section 1.2).

2.3.1 Les langages réflexifs

Les interpréteurs des langages à objets ont été parmi les premiers supports d'exécution à offrir des capacités d'adaptation. En effet, du fait de leur structure (encapsulation des données) et de leur organisation (héritage et composition), ces langages sont particulièrement adaptés à la mise en œuvre d'architectures réflexives [91]. Ainsi, parmi les langages réflexifs, nous pouvons citer SmalTalk [73] apparu en 1972, ObjectiveC [54] dans les années 80, Python [114] en 1990 ou encore Java [74] et Ruby [113] au milieu des années 90. Dans ces langages, les objets du niveau méta, appelés méta-objets, décrivent la représentation du programme et offrent des capacités de contrôle sur les objets du niveau de base (le code du système). La liaison entre objets et méta-objets est définie au travers de protocoles de méta-objets (MOP) [86]. La modification des méta-objets pour introduire de nouvelles sémantiques de représentation et d'exécution des objets (concurrence, localisation des objets répartis, envoi de messages distants [95]) permet ainsi d'adapter l'exécution d'un programme.

Cependant, la réflexion étant directement intégrée dans le langage de programmation et son lien causal entre objets et méta-objets étant direct, chaque modification du niveau méta impacte directement le système en cours d'exécution, ne permettant pas de passer par une phase de validation qui pourrait éviter des reconfigurations erronées. De plus, la réflexion permet de modifier la sémantique d'un langage ce qui peut être très intéressant, mais aussi très dangereux puisque la signification du programme est directement modifiée. C'est le cas par exemple si l'on considère le changement de sémantique de l'opérateur conditionnel pour lequel, il devient possible d'inverser la sémantique.

2.3.2 Les architectures réflexives

2.3.2.1 Approche à composants

La problématique de modélisation de l'architecture d'un logiciel a mis en avant la conception à l'aide de composants logicielle. La notion de composant est apparue pour la première fois lors de la conférence NATO en 1968 avec Douglas McIlroy [112]. Ce paradigme met en avant la réutilisation des *composants logiciels* et la création d'applications par l'assemblage de ces composants. Medvidovic et Taylor classifient les approches à composants en montrant qu'elles partagent généralement les trois éléments [96] que sont les composants, les connecteurs et la configuration. Les composants sont des entités logicielles ayant un type. Ils fournissent et requièrent un ensemble de fonctionnalités accessibles au travers de connecteurs. Les connecteurs sont chargés de la communication entre les différents composants d'une application. Une application correspond donc à un assemblage particulier de composants et de connecteurs que l'on appelle configuration.

Outre la définition modulaire d'application et la réutilisabilité des composants, cette approche offre aux plates-formes d'exécution la capacité de pouvoir déployer dynamiquement de nouveaux composants qui sont des entités de déploiement pour construire ou faire évoluer une application. De plus, les approches à composants ont largement été utilisées comme paradigme de base l'architecture logicielle et pour le support d'applications adaptables.

En 2007, Crnkovic et al. ont publié une classification des approches à composants [55] en fonction de trois dimensions que sont le cycle de vie, la construction d'application à partir de composants et la définition des propriétés non fonctionnelles. Cette classification a ensuite été étendue en 2011 [56] avec une quatrième dimension qu'est le domaine d'application d'une approche. Les différentes caractéristiques que nous avons définies rentrent dans la définition de cette quatrième dimension. En effet, la représentation multi-niveaux et la notion d'hébergement sont spécifiques à la représentation d'un Cloud alors que les notions de distribution et d'hétérogénéité sont plus généralistes puisqu'ils caractérisent la représentation de systèmes distribués. Quant aux caractéristiques d'extensibilité et de réflexivité désynchonisable, ils représentent des besoins généralistes pour n'importe quel système qui se veut adaptable.

Il existe de nombreuses approches à base de composants. Parmi elles, nous avons choisi de présenter Darwin et Fractal car elles représentent le panel complet des approches existantes prenant en compte la problématique d'adaptation logicielle. Darwin est l'une des premières approches de ce genre fournissant des capacités d'adaptation et Fractal est l'une des solutions les plus complètes et est facilement extensible.

Darwin [70] Darwin est un modèle de composant proposé par Ioannis Georgiadis et al. en 2002 pour décrire l'architecture d'applications distribuées. Dans ce modèle, les composants sont définis par des types et un ensemble d'instances. Ces instances peuvent être définies lors de la conception de l'architecture ou bien pendant l'exécution. En effet, chaque instance intègre à l'exécution une vue globale de l'architecture ainsi qu'un gestionnaire de configuration chargé de maintenir la cohérence entre la vue globale et la configuration courante du système. Ainsi, chaque instance est capable de reconfigurer l'architecture du système tant au niveau des instances de composants que des connexions entre ces composants.

Bien que ce modèle offre un support pour l'adaptation dynamique, il n'offre pas suffisamment de concepts pour représenter non seulement les applications, mais aussi les plates-formes d'exécution de ces applications et encore moins l'infrastructure concrète qui héberge ces plates-formes. De ce fait, il est possible, dans notre exemple, de définir le serveur web ainsi que les pages, mais il n'est pas possible de spécifier les spécificités des plates-formes d'hébergement. De plus même si Darwin offre une représentation globale du système permettant à chacun des composants de modifier le système, il ne permet pas de désynchroniser la vue globale du système afin de permettre de préparer une adaptation et de valider celle-ci avant de l'appliquer sur le système. Pour finir, Darwin ne propose rien concernant la définition de la distribution des applications.

Fractal [48] Fractal est un modèle à composants supportant la définition de composants élémentaires ou composites. Un composant est défini par deux éléments :

- une membrane qui expose les points d'entrée du composant correspondant aux interfaces requises ou fournies par celui-ci et qui héberge aussi les capacités de réflexivité
- le contenu qui correspond au code métier du composant ou un ensemble de composants dans le cadre des composants composites.

En plus des composants, le modèle permet de définir de manière explicite les interactions entre les composants au travers des liaisons (*bindings*). Ces liaisons spécifient les liens entre les fonctionnalités fournies par un composant et les fonctionnalités requises par un autre. La composition de composant est, elle aussi, explicite et offre la possibilité de partager un même composant fils entre plusieurs composants composites. Fractal propose plusieurs implémentations de plates-formes offrant la possibilité de développer des applications avec différents langages tels que Java avec les plates-formes Julia et AOKell, C avec Cecilia, .NET avec FractNet et SmallTalk avec FracTalk.

Fractal ne propose pas de quoi modéliser plusieurs niveaux que sont l'infrastructure, la plate-forme et l'application dans un Cloud même si elle offre la possibilité de définir des composants composites c'est-à-dire des composants construits à partir d'un assemblage de composants. Dans notre exemple, il est possible de définir le serveur web et les pages, mais il n'est pas possible de définir les plates-formes d'hébergement. La notion de topologie réseau n'est pas non plus présente dans ce modèle ne permettant donc pas de localiser les différents éléments notamment dans le cadre de l'hébergement des plates-formes sur une infrastructure réelle. De plus, de la même façon que Darwin, Fractal ne propose pas de solution pour effectuer de la réflexion désynchonisable. Cependant, contrairement à Darwin, Fractal offre la possibilité d'étendre son framework grâce à l'ajout dans la membrane de contrôleurs supplémentaires. GCM est une extension de Fractal permettant de concevoir des applications pour grilles de calcul. Nous nous attarderons en profondeur sur cette solution dans la suite de ce document (en section 2.4.3) car c'est la solution qui se rapproche le plus des besoins de notre abstraction.

2.3.2.2 Architecture orientée Service

La programmation orientée services [104] définit, d'après Papazoglou, un moyen de réorganiser les applications sous forme de services qui interagissent entre eux. Les services se veulent indépendants de la plate-forme, c'est-à-dire qu'ils ne dépendent pas les uns des autres pour s'exécuter et sont utilisables au travers de leur description. Contrairement à l'approche à composants, l'interconnexion des services est dite *lâche*, c'est-à-dire qu'elle n'existe que lors de l'interaction réelle. On parle aussi de couplage faible. Ce couplage faible permet aussi aux services d'être complètement indépendants des autres pouvant ainsi s'exécuter dans un mode restreint si certains services requis ne sont pas accessibles. De plus, la sélection du service avec qui le service client va interagir peut se faire à la demande et permet ainsi la sélection dynamique de service. En plus de ce couplage faible, les approches à services proposent généralement ce que

l'on appelle un registre de service. Chaque service s'enregistre auprès de cette entité et ensuite chaque service nécessitant une interaction avec un autre service proposant une fonctionnalité spécifique peut effectuer une recherche de service au niveau de ce registre. C'est donc le registre de service qui permet de gérer le couplage entre les services.

Outre les implémentations d'architectures orientées services citées par la suite, il en existe d'autres comme les outils proposés dans le cadre de la SOA Foundation d'IBM [18]. Mais ces abstractions n'offrent pas d'outils permettant de concevoir l'adaptation selon le modèle MAPE.

iPOJO [62] iPOJO est une implémentation de l'approche à composants orientée services hébergée en tant que sous projet du projet Apache Felix qui est une implémentation open source de la spécification OSGi [103]. L'objectif d'iPOJO est de fournir une plate-forme d'exécution permettant de simplifier le développement d'applications basées sur l'approche à services. Cette plate-forme est basée sur la spécification OSGi, et comporte par conséquent les mêmes caractéristiques : centrée sur Java, centralisée et supportant un haut degré de dynamisme.

iPOJO ne permet pas de modéliser plusieurs niveaux simultanément même s'il est envisageable de modéliser une plate-forme comme une application. Là encore, il est possible de définir le serveur web et les pages, mais il n'est pas possible de définir les plates-formes d'hébergement. De plus tout comme les travaux présentés avec l'approche orientée composant, iPOJO ne permet pas de représenter la topologie réseau d'un système puisqu'il ne représente une application que dans le cadre d'une plate-forme locale qui éventuellement est connectée à des services locaux représentant des proxies de services distants. Du fait qu'il ne représente que la plate-forme locale, il ne permet pas de représenter plusieurs plates-formes et donc l'hétérogénéité de ces plates-formes.

Service Component Architecture (SCA) [33] SCA est un modèle de composants permettant la définition de service dont l'objectif est de s'abstraire des dépendances techniques que peuvent être les langages de programmation, les frameworks de conception ou les protocoles de communications. Ainsi, SCA définit quatre différents éléments :

- un modèle d'assemblage qui définit les composants et leurs services et comment ils sont assemblés pour former une application. Ce modèle permet aussi de définir les propriétés nécessaires au fonctionnement de chacun des services comme les données d'authentification.
- un modèle de conception qui définit comment fournir et utiliser un service SCA. Avec ce modèle est fourni un ensemble d'implémentations (par exemple Java, BPEL, C++, Javascript) pour permettre son utilisation dans différents langages.
- un modèle pour la définition des propriétés non fonctionnelles des services SCA
- un modèle de connecteurs qui permet de définir des moyens de communication pouvant ensuite être utilisés dans l'assemblage de l'application.

L'une des limitations de la spécification SCA réside dans le manque de définition concernant l'adaptation. De ce fait, cette gestion de l'adaptation dépend fortement de l'implantation utilisée. On peut tout de même noter que cette spécification ne fournit aucune information concernant la notion d'hébergement et qu'elle a d'ailleurs pour

objectif de s'abstraire des plates-formes d'exécution masquant ainsi la notion d'hébergement. Comme nous l'avons expliqué dans les critères importants de notre abstraction, nous ne voulons en aucun cas masquer les caractéristiques de l'environnement d'exécution ou la distribution des différents éléments du système puisque ces caractéristiques et cette notion de distribution peuvent être utilisées pour effectuer de l'adaptation. Ainsi, il est toujours possible de définir le serveur web ainsi que les pages, mais il n'est pas possible de modéliser explicitement leur distribution et il n'est pas possible de modéliser les plates-formes d'exécution.

FraSCAti [97] FraSCAti est une implémentation des spécifications SCA sur le modèle à composants Fractal. L'objectif de cette solution était de proposer l'utilisation d'un standard pour la conception d'applications orientées services tout en fournissant grâce à Fractal des facilités pour l'adaptation dynamique de ces applications.

L'une des caractéristiques de FraSCAti est de fournir une implémentation de sa plate-forme d'exécution sous la forme d'une application FraSCAti. On peut donc dire que FraSCAti peut aussi bien modéliser une application que sa plate-forme d'exécution. FraSCAti a aussi été utilisé pour la définition de plate-forme [98] telle qu'on l'entend dans le domaine du Cloud Computing. Mais FraSCAti ne permet pas de représenter la notion d'hébergement. De ce fait, il n'est pas possible de représenter la plate-forme d'exécution en même temps que l'application hébergée. De plus, FraSCAti ne permet pas de modéliser explicitement la distribution de l'application ou de la plate-forme. En effet, un modèle FraSCAti représente la configuration d'un environnement d'exécution local avec la déclaration explicite de services distants permettant aux composants locaux de les utiliser.

2.3.2.3 Modèle à l'exécution

L'ingénierie dirigée par les modèles ou MDE pour Model-Driven Engineering [110] en anglais prône l'utilisation d'abstractions de haut niveau appelées modèles dans le cadre des différentes étapes de la production d'une entité logicielle que ce soit l'implémentation, le test, l'intégration ou la maintenance. Ainsi, un modèle est un point de vue simplifié de l'entité logicielle.

L'un des langages de modélisation le plus connu est UML [47] conçue à l'origine pour l'analyse et la conception de système par objets. Est apparue ensuite la standardisation formulée par l'OMG (Object Management Group en anglais) [22] appelée MDA pour Model Driven Architecture en anglais [24] qui propose de modéliser une application indépendante de la plate-forme d'exécution (PIM pour *Platform Independent Model en anglais*). Puis ce modèle PIM est transformé en un modèle de plus bas niveau dans lequel les caractéristiques de la plate-forme d'exécution sont prises en compte. C'est le modèle PSM pour *Platform Specific Model* en anglais. En généralisant cette vision promue par l'OMG, le MDE (pour *Model Driven Engineering*) regroupe l'ensemble des approches visant à rendre les approches de modélisation productives.

Dans la continuité du MDE et de l'approche MDA bâtie sur une vision *top-down* où l'on construit l'application à partir de modèles abstraits et d'un ensemble d'étapes de

raffinement, le modèle à l'exécution ou *M@R* pour *Model at Runtime* [44, 99] en anglais souhaite effacer la frontière entre conception et exécution. L'idée est de maintenir une version courante du modèle correspondant à l'état courant du système. Cela correspond à l'application du principe de réflexion au niveau des modèles. Cependant, cette notion de réflexion est ici légèrement différente dans le sens ou l'intérêt du *M@R* est de pouvoir réutiliser l'ensemble des techniques et outils fournis par le MDE pour assurer la validité de la configuration qui doit être mise en place. Ainsi, contrairement aux plates-formes à base de composants ou de services tels que Fractal, SCA/FraSCAti ou OSGi/iPOJO, les solutions de *M@R* fournissent un niveau de réflexion désynchronisable du système en cours d'exécution. Une fois la nouvelle configuration testée, simulée, validée, le modèle est resynchronisé avec le système courant déclenchant ainsi l'adaptation.

La figure 2.1 décrit le processus du M@R. Tout d'abord un nouveau modèle (modèle cible) est soumis. Ce modèle peut être vérifié afin d'assurer sa validité. Une fois vérifié, il est comparé avec le modèle courant afin de définir le plan d'adaptation à appliquer. C'est la phase de planification du modèle MAPE (voir 1.2). Ce plan d'adaptation est transmis au moteur d'adaptation qui se charge de l'appliquer sur le système courant. C'est la phase d'exécution (voir 1.2). Une fois les modifications appliquées, le modèle est sauvegardé comme étant à présent le modèle courant.

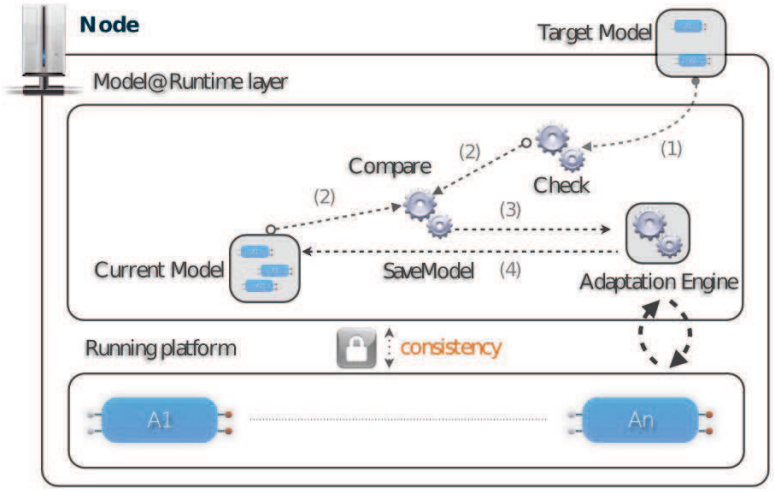


FIGURE 2.1 – Processus du Model@Runtime

PauWare [108] PauWare propose d'utiliser des diagrammes d'état UML pour modéliser le fonctionnement des composants. Ainsi, les services fournis par un composant sont modélisés par des évènements dans le diagramme d'états modélisant le composant. Les fonctionnalités de celui-ci correspondent à des actions et les services requis correspondent à la génération d'évènements. PauWare offre aussi la possibilité de définir des composants composites et définit les relations spécifiques à ce genre de relations afin d'assurer le bon fonctionnement des composants composites et composés.

Malgré l'utilisation d'un modèle à l'exécution, PauWare ne propose pas de capacités d'adaptation hormis la capacité de spécifier directement aux composants l'état dans lequel ils doivent se trouver. De plus, un diagramme d'état ne représente qu'un seul composant même si celui-ci est un composite. Ce modèle ne permet donc pas de modéliser un ensemble de niveaux comme le sont les niveaux que l'on représente dans un Cloud.

Genie [41] Genie est une approche de modèles à l'exécution proposée par Nelly Bencomo en 2009. Dans cette approche, la conception d'une application se caractérise entre autres par l'utilisation de deux langages dédiés. Le premier, appelé OpenCOM DSL, permet de définir un modèle de variabilité de la configuration d'une application. C'est de la variabilité structurelle. Le deuxième permet de définir la variabilité de l'environnement d'exécution et les impacts de cette variabilité sur la configuration de l'application. Une fois les deux modèles définis, une application et le système d'adaptation correspondant sont générés. Contrairement aux approches présentées précédemment (Fractal, SCA, ...), Genie permet de définir au moment de la conception de l'application, les adaptations potentielles qui peuvent se produire durant l'exécution de l'application.

Contrairement à PauWare, Genie permet de définir un système d'adaptation capable d'effectuer de l'adaptation de manière déconnectée. Cependant, il ne permet pas de modéliser les différents niveaux constituant un Cloud et n'explicite pas les propriétés de distribution du système. Ainsi, il est possible de modéliser le serveur web ainsi que les pages et il est possible de vérifier les adaptations avant de les appliquer. Mais il n'est pas possible de modéliser la distribution des entités.

ART [100] Le projet ART pour *model At RunTime* propose un modèle de composants exécutable sur les plates-formes OSGi et Fractal et dont la particularité est d'utiliser l'approche orientée aspects [87] pour générer les reconfigurations du système. Ainsi, une configuration du système est définie par un ensemble de points de variation représentant des préoccupations du système pouvant être définies de différentes manières et qui sont modélisées sous la forme d'aspects. Ces aspects représentent des morceaux de configuration capable de répondre à la préoccupation modélisée par le point de variation et c'est la composition de ces aspects représentant l'ensemble des points de variations qui permet de générer la configuration complète du système. Une adaptation se traduit donc par la sélection d'un ou plusieurs aspects pour un ou plusieurs points de variation présents dans le système qui sont ensuite appliqués sur le modèle courant du système.

Tout comme Genie, ART est capable de déconnecter le processus d'adaptation afin de pouvoir valider une configuration avant de l'appliquer. Et tout comme Genie, ART ne propose pas non plus de modéliser la plate-forme d'exécution de l'application et encore moins l'infrastructure d'exécution de celle-ci. ART ne permet pas non plus de modéliser la distribution d'un système.

2.4 Les abstractions pour le Cloud

Nous présentons ici des abstractions pouvant servir de support pour effectuer de l'adaptation des différents niveaux d'un Cloud. Ces abstractions peuvent être des APIs nous permettant d'une part d'obtenir des informations sur la configuration courante d'un Cloud ou tout du moins de l'un des niveaux de ce Cloud et d'autre part de pouvoir interagir avec le Cloud afin de le reconfigurer.

2.4.1 Les solutions académiques

gMDE [93] gMDE est une approche de conception d'applications pour grille basée sur l'utilisation des techniques de MDE.

Dans cette solution, quatre modèles sont utilisés pour définir l'architecture de l'application, les contraintes de qualité de service associées à cette application, la configuration logicielle requise sur la plate-forme d'exécution et l'ensemble des ressources disponibles pour la plate-forme d'exécution. Grâce à ces différents modèles, il est possible de représenter non seulement l'application, mais aussi la plate-forme d'exécution et la distribution de cette plate-forme. La plate-forme peut en plus être hétérogène du fait de la définition des nœuds qui la compose. Pour autant, ces modèles ne sont pas utilisés pour gérer de la virtualisation sur une infrastructure, ne permettant donc pas de manipuler et l'infrastructure et la plate-forme, mais seulement un seul de ces niveaux. De manière générale, cette solution ne permet donc pas d'étendre le nombre de niveaux dans le système. De plus, la hiérarchie liée au déploiement de l'application sur la plate-forme n'est pas disponible dans le modèle, mais est générée lors des multiples transformations de modèle.

Neptune [49]. Neptune est un langage dédié (DSL) permettant de configurer et déployer des applications de calcul haute-performance (HPC) sur un Cloud. L'une des difficultés dans le déploiement d'application HPC sur des solutions de Cloud réside dans la configuration de la plate-forme afin que celle-ci offre l'ensemble des dépendances dont a besoin une application HPC. Pour pallier cette difficulté, il est possible d'utiliser des machines virtuelles spécifiquement configurées. Cependant, la configuration d'une plate-forme HPC n'est pas unique.

Neptune propose grâce à son langage dédié de définir, pour chaque application à déployer, la configuration requise pour la plate-forme. Grâce à cette information, la plate-forme d'exécution construite sur l'équivalent open source de Google App Engine et appelée AppScale est capable d'installer dynamiquement, sur les nœuds alloués à l'application, l'ensemble des dépendances nécessaires.

Bien que cette solution soit fortement orientée vers les applications HPC, elle peut tout de même être utilisée pour spécialiser la plate-forme d'exécution de n'importe quelle application. Il est donc possible non seulement de modéliser le serveur web et les pages, mais aussi les plates-formes qui hébergeront ces entités. Offrant la capacité de spécialiser une plate-forme selon les besoins d'une application, elle n'intègre malheureusement pas la notion d'hébergement de ces plates-formes. Ainsi, c'est *AppScale* qui fournit la

capacité de déployer les plates-formes sur Amazon EC2 ou sur une infrastructure Eucalyptus.

CloudML [1] CloudML est un langage de modélisation dont l'objectif est de fournir une méthode et des outils pour la conception d'applications et/ou de plates-formes ainsi que pour leur déploiement en assurant l'interopérabilité, la portabilité et la réutilisation de ces éléments (application et plate-forme). Pour se faire, CloudML utilise l'API JClouds [19] qui propose une abstraction pour un certain nombre d'APIs de déploiement sur des infrastructures de Cloud telles que AmazonEC2 [1] ou Rackspace [31].

Cependant, la version courante de cette solution reste limitée. En effet, elle ne permet actuellement que la définition de plate-forme et plus exactement de machines virtuelles qu'il est ensuite nécessaire de configurer manuellement (installer les dépendances nécessaires pour l'exécution d'application). De plus, la caractérisation de ces machines virtuelles est elle-même limitée à son processeur et sa mémoire vive. Il est donc impossible de spécifier l'architecture matérielle voulue ni même le système d'exploitation requis. Enfin, cette solution ne permet actuellement que la réservation de nouvelles machines virtuelles ainsi que leur démarrage, mais ne permet pas de les arrêter, de les redémarrer ni de les détruire.

ConPaaS [105] ConPaaS, qui est issu d'un projet européen nommé Contrail [5], propose des outils pour définir des plates-formes spécifiques en fonction des besoins d'une application. Pour cela, ConPaaS permet de configurer la plate-forme en y incluant des services spécifiques tels que des bases de données ou des serveurs web. L'existence de ces services permet aux utilisateurs de migrer facilement leurs applications serveurs comme leurs sites web sur une solution de Cloud. L'intérêt étant grâce à ConPaaS de faciliter et même d'automatiser la gestion de la montée en charge. En effet, en plus de faciliter la définition de la plate-forme, ConPaaS offre aussi un support pour le monitoring aussi bien sur les VMs que sur les services afin de pouvoir identifier de manière précise la charge de chacun des éléments et de pouvoir dynamiquement installer et démarrer ou arrêter et désinstaller des VMs incluant ces services en fonction des besoins.

La configuration d'une plate-forme peut s'effectuer manuellement par l'intermédiaire de l'interface de gestion proposée par ConPaaS mais aussi par l'intermédiaire d'un "*manifest*". Ce *manifest* permet de définir l'ensemble des services à mettre en place sur la plate-forme, le code et les données à installer ainsi que toutes informations nécessaires au bon fonctionnement de chacun des services et des applications hébergées. Ce *manifest* permet aussi de définir les caractéristiques du réseau de chacun des services permettant ainsi à ConPaaS de définir un réseau privé virtuel si nécessaire.

ConPaaS est une plate-forme de Cloud qui se veut capable d'être déployée sur n'importe quelle infrastructure. Dans la version actuelle, elle intègre le déploiement sur AmazonEC2 ainsi que OpenNebula. Cependant, elle ne propose aucun support pour la gestion de l'infrastructure et ne permet de manipuler et configurer que des plates-formes. Chaque description d'une plate-forme définit les caractéristiques de chaque nœud et plus

1. http ://www.cloudml.org/

spécialement les services qui y sont hébergés permettant ainsi de modéliser l'hétérogénéité des nœuds.

Modélisation de l'infrastructure De manière générale, les solutions d'infrastructure propriétaires tels que Amazon EC2, Windows Azure ou Google Compute Engine ne fournissent aucune information sur leur système de gestion de leurs ressources. Il existe tout de même, avec les solutions open sources ou les projets de recherche, des informations sur les moyens d'effectuer cette gestion dynamique que ce soit pour Cloud mais aussi pour grille de calcul. Ainsi, nous pouvons citer les travaux sur la consolidation de serveurs [115] qui consistent à organiser l'usage des ressources afin de limiter le nombre de serveurs actifs. Nous pouvons aussi citer les travaux autour de Nimbus tels que les travaux présentés dans [94] qui proposent de combler la sous-utilisation de certaines ressources en y intégrant des machines virtuelles tampon pouvant être utilisées par des applications HTC pour *High-Throughput computing* en anglais dont l'objectif est de maximiser l'usage des ressources afin d'assurer la terminaison du plus grand nombre de jobs soumis. Nous pouvons aussi citer le projet Snooze [65, 64] qui lui aussi travaille sur de la consolidation de serveur mais qui propose en plus des algorithmes de décision spécifiques à la gestion de l'énergie.

Mais ces solutions, bien que proposant des algorithmes capables d'adapter l'utilisation des ressources, ne possèdent généralement qu'une représentation interne et spécifique de l'infrastructure. Ces représentations ne permettent en aucun cas de représenter autre chose que l'infrastructure avec des informations sur les machines virtuelles qui représentent les plates-formes et sur lesquelles ils peuvent influer sur la localisation.

2.4.2 Les solutions commerciales

2.4.2.1 Infrastructure

Chacune des solutions d'infrastructure propose une API d'interactions pouvant servir au niveau plate-forme pour pouvoir récupérer des informations sur les nœuds de la plate-forme en cours d'exécution (phase d'observation dans le modèle MAPE) mais aussi de pouvoir modifier celle-ci (phase d'exécution dans le modèle MAPE).

Ainsi, AmazonEC2 offre l'API EC2. Cette API permet de définir des instances de machines virtuelles sur l'infrastructure d'Amazon en pouvant spécifier la famille d'instance voulue qui définit les caractéristiques CPU, mémoire, architecture et disque ainsi que l'image de la machine (AMI) qui est un format spécifique à Amazon. Il est aussi possible de spécifier la localisation de cette instance même si cette localisation ne correspond pas à une machine physique, mais à une région. Il est aussi possible de spécifier pour chaque machine son adresse IP publique. L'API fournit aussi de quoi arrêter, redémarrer et supprimer des instances ainsi que des fonctionnalités permettant d'effectuer du monitoring sur les ressources utilisées.

D'une manière générale, chacune des solutions d'infrastructure propose sa propre API pour effectuer ce genre de tâches. Plusieurs projets sont tout de même compatibles avec l'API proposée par Amazon EC2 [1]. C'est le cas de Nimbus [21], de CloudStack [4]

ou encore d'OpenNebula [26]. En plus des APIs fournies par chacune des solutions, il existe d'autres APIs proposant une abstraction commune pour un bon nombre de solutions existantes. Parmi celles-ci, nous pouvons citer JClouds [19] ou deltaCloud [6]. Nous pouvons aussi citer OCCI [25], pour Open Cloud Computing Interface en anglais proposé par l'Open Grid Forum dont l'objectif était avec cette API et le protocole associé de fournir un standard pour l'interaction et la gestion d'une infrastructure de Cloud. Bien que cette API ne soit pas encore utilisée dans toutes les solutions, beaucoup d'entre elles l'ont intégrée. C'est le cas par exemple des projets open source OpenStack [28], OpenNebula [26] ou Eucalyptus [9] ainsi que l'API JClouds [19].

Pour résumer, nous pouvons dire qu'il existe autant d'APIs d'interaction qu'il existe de solutions d'infrastructure même si les APIs EC2 et OCCI se retrouvent tout de même supportées par plusieurs solutions. Mais l'ensemble de ces APIs ne permet que de la manipulation de plates-formes sur les infrastructures et en aucun cas n'offre de support pour de la manipulation d'application ni même pour de la reconfiguration d'infrastructure. De plus, même si certaines APIs fournissent des informations de localisation, celles-ci ne sont pas précises au point d'identifier les machines physiques d'hébergement. Elles permettent tout de même de gérer la notion de nœuds distribués le plus souvent par la définition d'adresse IP pour chaque machine virtuelle. L'hétérogénéité des plates-formes est quant à elle modélisée grâce aux images et aux spécificités de processeur(s), mémoire vive, etc. Enfin, Ces APIs fournissent par la même occasion des capacités d'adaptation de plates-formes, mais ne propose pas une représentation désynchronisable de la réalité pour faire de la validation de configuration.

2.4.2.2 Plate-forme

Bien qu'étant un domaine moins fourni que celui des infrastructures ou des applications, il existe tout de même de nombreuses solutions de plate-forme. Là encore, bon nombre de ces solutions sont des solutions propriétaires sur lesquelles peu d'informations techniques sur les capacités de gestion et de reconfiguration sont disponibles.

La plupart pour ne pas dire l'ensemble des solutions existantes fournissent des capacités d'adaptation par rapport à la montée en charge des applications afin d'offrir de l'élasticité. Elles permettent aussi d'observer l'exécution des applications afin de surveiller leur bon fonctionnement et sont capables de les redémarrer si nécessaire ou au moins de notifier l'administrateur pour qu'il puisse résoudre le problème. Cette observation permet aussi de connaître le taux d'utilisation des applications. Ainsi, Google App Engine est capable de dupliquer les instances d'applications en cours d'exécution pour assurer une qualité de service suffisante malgré l'augmentation du nombre d'utilisateurs. GAE permet aussi de gérer le niveau de qualité de service que l'on veut assurer à l'application sachant que le prix d'hébergement de l'application est fixé en fonction de ce niveau de qualité de service.

Outre la notion d'adaptation de la plate-forme, de plus en plus de solutions intègrent des outils pour être déployées sur différents IaaS. C'est le cas pour CloudFoundry par exemple qui embarque un composant appelé BOSH lui permettant de déployer des nœuds Sphere, vCloud, OpenStack et Amazon EC2. Cependant, ce genre de composant

n'apparait pas dans la plate-forme directement, mais est le plus souvent un outil externe nécessaire au déploiement sur une infrastructure d'une ou plusieurs applications avec la plate-forme associée. De ce fait, la connaissance de l'architecture courante du système est difficilement récupérable dynamiquement pour l'adapter par exemple pour migrer une plate-forme depuis l'infrastructure en cours d'utilisation vers une nouvelle.

Les solutions de déploiement L'activité de déploiement représente l'ensemble des opérations nécessaires à la mise en place d'un logiciel sur une plate-forme d'exécution afin que ce logiciel soit accessible ensuite aux utilisateurs [50]. En 1998, Carzaniga et al [50] ont proposé une définition de ces différentes opérations représentée par la figure 2.2.

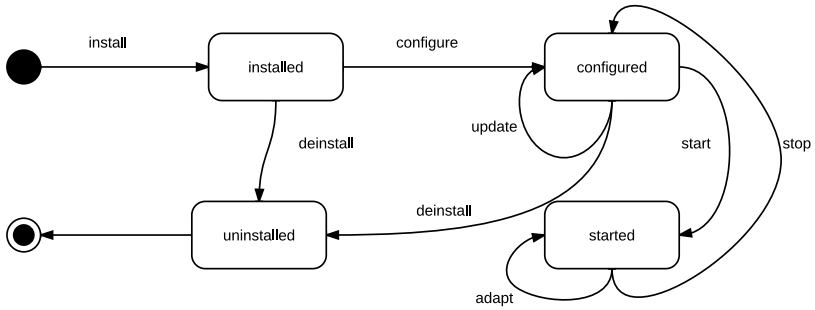


FIGURE 2.2 – Cycle de vie de l'activité de déploiement

Cette définition propose les opérations suivantes :

- l'**installation** est chargée de récupérer et de mettre en place l'ensemble des ressources nécessaires pour le bon fonctionnement de l'application. Outre les ressources spécifiques telles que les bibliothèques, les exécutables ou encore les fichiers de configuration, il y a aussi les dépendances du logiciel. Par exemple pour installer un serveur TomCat, il est nécessaire d'installer Java.
- la **désinstallation** consiste à enlever l'ensemble des ressources liées à l'application et désormais inutiles au fonctionnement du reste du système.
- la **configuration** consiste à mettre en place les différentes propriétés pouvant être nécessaires pour l'exécution de l'application. Par exemple, sous Linux, il peut être nécessaire de spécifier dans la variable d'environnement PATH, le chemin pour accéder au binaire de l'application.
- la **mise à jour** consiste à passer l'application d'une version vers une autre en mettant à jour l'ensemble des ressources correspondantes à l'application, dont les dépendances. La mise à jour de dépendances ne consiste pas simplement en un remplacement de binaire, mais consiste aussi à gérer l'évolution des versions de

ces binaires. En effet, une dépendance peut être utilisée par plusieurs applications sur la même plate-forme. Si une nouvelle version d'une dépendance est nécessaire pour l'application A mais que l'application B nécessite toujours l'ancienne version alors la mise à jour doit être capable de faire cohabiter les deux versions.
- le **démarrage** consiste à exécuter l'application sur la plate-forme
- l'**arrêt** consiste à stopper l'exécution
- la **reconfiguration** correspond à de l'adaptation dynamique de l'application en fonction des changements de l'environnement d'exécution. Un exemple d'adaptation peut être de dupliquer l'application sur plusieurs nœuds de la plate-forme afin de supporter la montée en charge dont l'application fait l'objet.

Ces différentes opérations indiquent que le déploiement correspond non seulement à la mise en place de l'application, mais aussi à la mise en place de son environnement d'exécution et donc de la plate-forme. De nombreuses solutions se sont d'ailleurs orientées vers la conception de plate-forme de Cloud. Ces solutions permettent non plus de configurer des services physiques, mais des machines virtuelles pouvant ensuite être déployées sur diverses infrastructures de Cloud. Parmi les solutions existantes, il y a par exemple Chef [29], Puppet [30], Vagrant [34] ou encore rPath [32].

2.4.2.3 Application

Les offres d'applications proposées comme SaaS sont le plus souvent des offres propriétaires et ne sont définies que sous la forme d'entités indépendantes ayant peu voire pas d'interaction avec d'autres applications. D'ailleurs, l'intégration d'un ensemble d'applications pour former un système complet est un marché important principalement développé dans le cadre des systèmes d'information et de gestion.

De ce fait, nous avons peu d'informations sur les capacités d'adaptation de ces applications et il n'y a pas de réelle abstraction de celles-ci. De plus, elles sont souvent dépendantes de la plate-forme et/ou de l'infrastructure sous-jacente. C'est le cas par exemple avec l'application de stockage Dropbox qui utilise sa propre infrastructure pour le stockage des données de ses utilisateurs ainsi qu'avec l'ensemble des applications proposées dans les systèmes de gestion de SalesForce.

Pour autant, les solutions que nous avons présentées en section 2.3 peuvent être des solutions pour la conception d'applications pour le Cloud.

Les *Rapid Business Application Development* (RBAD) [57]. Ce type d'outils permet de concevoir des applications pour Cloud utilisant un ou plusieurs langages dédiés (DSL) le plus souvent graphiques. Ces solutions sont le plus souvent spécifiques à la conception de site web ou de conception de base de données. Il existe entre autres Mendix [20], Zoho Creator [37], Force.com [10] ou WaveMaker [35].

Malheureusement, peu d'information sur leur fonctionnement interne n'est disponible. Pour autant à partir des exemples d'utilisation nous pouvons en tirer quelques informations. Bien que ce soit une abstraction de ce que l'on peut faire sur un Cloud, ces solutions se limitent dans la majorité des cas à de la conception d'applications au niveau SaaS et ne proposent aucune capacité d'adaptation hormis par le biais de leurs

outils de déploiement d'applications sur la plate-forme. Il est à noter aussi que ce genre de solution est le plus souvent associé à une plate-forme spécifique.

2.4.3 GCM : une abstraction pour grille

Grid Component Model (GCM) [92] est un modèle à composants pour applications sur grille de calcul défini dans le cadre du projet européen CoreGrid et implémenté par le projet européen de grille de calcul GridCOMP.

L'objectif de GCM était de proposer, pour la conception d'application sur grilles de calcul, une solution permettant de faciliter la conception d'application et la réutilisation de morceaux d'application. Pour cela, GCM est construit au-dessus du modèle à composants Fractal. En plus de la notion de modularité des applications, Fractal offre des capacités d'extension ainsi que des propriétés de réflexion et d'adaptation.

Du fait de son orientation pour application sur grilles de calcul, GCM fournit un ensemble d'extension à Fractal permettant de supporter les spécificités de ce genre d'application notamment des sémantiques de communication spécifiques ainsi que la notion d'hébergement de l'application. Ainsi, GCM fournit trois sémantiques de communication souvent utilisées dans le cadre d'applications sur grilles. La première permet d'effectuer des communications entre composants d'une application par l'intermédiaire de flux, la seconde offre des capacités de diffusion vers plusieurs composants et la troisième propose des capacités de regroupement de requêtes afin que le composant qui reçoit les requêtes n'en traite qu'une seule et que la réponse soit fournie à l'ensemble des composants ayant émis une requête. La notion d'hébergement est, quant à elle, définie par la notion de nœud virtuel (*virtual node*). Ces nœuds virtuels peuvent être utilisés dans le code des composants ainsi que dans le langage de description d'architecture afin d'abstraire les informations réelles de l'infrastructure d'exécution de l'application. Avec ces nœuds virtuels, GCM permet donc de définir l'architecture de l'application en spécifiant la localisation de chacun des composants tout en laissant le déploiement concret de l'application sur des ressources réelles hors des préoccupations du développeur de l'application. Le développeur est donc capable de spécifier l'architecture de son application ainsi que les contraintes des nœuds sur lesquels elle s'exécutera par la suite. Ces contraintes peuvent être utilisées par des outils de déploiement automatique qui pourront ainsi sélectionner de manière intelligente les ressources nécessaires à une application.

En plus des capacités intéressantes que peut avoir Fractal pour notre problématique, GCM offre la possibilité de spécifier une plate-forme comme un ensemble de nœuds virtuels ayant chacun des spécificités utilisées ensuite par des outils de déploiement pour sélectionner les ressources sur une infrastructure réelle. Ces nœuds virtuels servent aussi à indiquer l'hébergement de chacun des composants. Cependant, le modèle GCM ne permet pas de représenter une infrastructure concrète et donc de modéliser l'ensemble des niveaux que l'on peut retrouver dans un Cloud. Pour résumer, il est possible de modéliser le serveur web et les pages ainsi que les plates-formes d'hébergement de ces entités, mais il n'est pas possible de spécifier comment ces plates-formes seront hébergées sur l'infrastructure puisque la gestion de l'infrastructure est délégué aux outils de

déploiement utilisés sur la grille d'exécution.

GCM étant extensible, il aurait été clairement possible de partir de cette solution pour construire une abstraction visant à permettre à un fournisseur de Cloud de définir ces politiques d'adaptation multi-niveaux. Pourtant, le choix s'est porté sur l'utilisation de Kevoree (présenté au chapitre suivant) plutôt que GCM. Ce choix est clairement discutable, il a été guidé principalement par des considérations pragmatiques. Développé au sein de l'équipe dans laquelle cette thèse a été menée, Kevoree était plus facile à modeler pour les besoins de cette thèse et il était à priori plus aisé d'y intégrer les caractéristiques souhaitées. En outre, dans la lignée des approches construites autour des concepts du models@runtime, il permettait de bénéficier des outils de modélisation développés dans l'équipe.

2.5 Synthèse

Comme nous l'avons expliqué dans le chapitre 1, les fournisseurs de services de Cloud ont besoin d'une abstraction leur facilitant la conception de systèmes d'adaptation capable de tenir compte de l'ensemble des niveaux afin de gérer l'adaptation d'une manière orthogonale à ces niveaux et non plus comme une problématique locale à chacun des niveaux.

Nous avons, dans ce chapitre, énuméré un ensemble de caractéristiques que cette abstraction doit gérer. Nous avons ensuite étudié un certain nombre d'abstractions existantes et les avons évaluées par rapport à ces caractéristiques.

Malgré l'existence de nombreuses solutions, aucune ne supporte l'ensemble des caractéristiques que nous avons définies que ce soit pour la représentation des différents niveaux, la notion d'hébergement, la représentation de la distribution du système ou encore la capacité de réflexion désynchronisable (voir tableau récapitulatif 2.3).

	Multi-niveaux	**Hébergement**	**Distribution**	**Hétérogénéité**	**Extensibilité**	**Réfléxivité**
Darwin	non	non	non	non	non	en partie
Fractal	non	non	non	non	oui	en partie
iPOJO	non	non	non	non	oui	en partie
FraSCAti	possible	non	non	non	oui	en partie
PauWare	non	non	non	non	oui	limité
Genie	non	non	non	non	oui	oui
RBAD	non	non	non	non	non	non
gMDE	en partie	non	oui	oui	oui	non
Neptune	en partie	en partie	non	en partie	non	non
CloudML	non	non	non	limité	non	non
ConPaaS	non	non	oui	oui	non	limité
GCM	en partie	oui	non	oui	oui	en partie

FIGURE 2.3 – Comparatif des différentes solutions

Le manque d'une abstraction permettant de répondre à la gestion de l'adaptation comme une préoccupation transverse nous a donc poussés à définir une abstraction spécifique permettant non seulement de modéliser chacun des niveaux de manière indépendante, mais également de modéliser les dépendances entre ces niveaux.

Chapitre 3

Kevoree

Dans ce chapitre, je présente le projet sur lequel j'ai travaillé en collaboration avec François Fouquet et dont la thèse s'intitule *Kevoree : Model@Runtime pour le développement continu de systèmes adaptatifs distribués hétérogènes* [67]. C'est dans ce projet que les contributions de cette thèse ont été intégrées afin de construire une abstraction pour le Cloud. Bien que ne faisant pas partie de la contribution directe de cette thèse, ce chapitre présente la conception de différents éléments de Kevoree à laquelle j'ai activement participé notamment sur la définition de la synchronisation du modèle dans les systèmes distribués et sur la mise en place des différentes phases du modèle MAPE. Ces éléments qui sont utilisables dans un contexte plus large que la mise en place d'une abstraction pour l'architecte de Cloud lui permettant la création de politiques d'adaptation multi-niveaux ont été largement guidé par cette problématique scientifique. Ce chapitre présente en outre de nombreux concepts nécessaires pour la compréhension de la suite de ce document.

Kevoree [1] est un cadre de développement d'applications à base de composants pour la conception de systèmes distribués adaptables. Fondée sur le paradigme de modèle à l'exécution (M@R), cette approche vise à proposer un modèle abstrait au travers duquel il est possible de manipuler les différents concepts caractérisant un système distribué.

3.1 Les caractéristiques de Kevoree

Kevoree a pour objectif de fournir une abstraction pour les systèmes distribués afin de pouvoir manipuler les principaux concepts de ce genre de système et vise à faciliter la gestion de l'adaptation pour ces systèmes. Pour se faire, Kevoree propose de supporter la synchronisation et désynchronisation entre le modèle réflexif et le système en cours d'exécution, la capacité à représenter la distribution du système, la séparation entre les composants métier et leurs interactions, la dissémination des reconfigurations ainsi que l'hétérogénéité des ressources sur lesquels s'exécute le système.

1. http://kevoree.org

Séparation entre composants métier et interaction (communication) Une application distribuée est composée de code métier spécifique mais aussi de code de communication. Contrairement au code métier, le code de communication ne possède pas obligatoirement de spécificités dues à l'application. De ce fait, il est intéressant de pouvoir décorréler le code métier de celui chargé de la communication permettant ainsi la réutilisation des différentes briques logicielles. Cela permet de simplifier la conception de composant métier en masquant la problématique de communication. De plus, l'adaptation des moyens de communication entre composants selon le contexte est besoin pour une application distribuée.

Gestion de la distribution Du fait de vouloir représenter un système distribué, les caractéristiques de distribution se doivent d'être modélisées et manipulables afin par exemple de permettre des adaptations quant à la distribution des différents éléments d'une application sur l'ensemble des systèmes d'exécution.

Désynchronisation entre le modèle réflexif et le système correspondant Kevoree étant basé sur les techniques de modèle à l'exécution, la définition d'une adaptation s'effectue par la définition d'un modèle qui sera soumis au processus de modèle à l'exécution (voir section 2.3.2.3) afin d'être validé avant d'être appliqué. C'est ce processus qui permet d'avoir un modèle réflexif désynchronisé du système en cours d'exécution.

Le fait de pouvoir désynchroniser le modèle réflexif du système en cours d'exécution permet de valider une configuration avant sa mise en place. Cette validation permet notamment de s'assurer de la cohérence de la configuration afin d'éviter un état instable du système voire une panne de celui-ci. Cette caractéristique est d'autant plus importante dans le cadre de système distribué afin d'éviter l'adaptation de certains nœuds du système alors que d'autres ne peuvent pas mettre en place cette nouvelle configuration.

Dissémination des adaptations Une fois qu'une adaptation est soumise (voir section 2.3.2.3), chacun des nœuds faisant partie du système se doit d'être notifié afin qu'elle soit prise en compte sur l'ensemble du système. Cependant, un système distribué ne permet pas forcément d'assurer une communication permanente entre les différents nœuds notamment dans le cadre de réseaux sporadiques, c'est-à-dire de réseaux sujet à de fréquentes erreurs et/ou de fréquentes déconnections. La prise en compte de ces contraintes réseaux dans la dissémination des adaptations est donc nécessaire pour assurer que le système évolue de manière cohérente. Ainsi selon les différents types de communication existants entre les nœuds, la synchronisation peut se faire de différentes manières.

Hétérogénéité des systèmes d'exécution Les systèmes distribués peuvent être composés de nombreux types de plates-formes d'exécution que ce soit des plates-formes mobiles type smartphone, des PCs, des serveurs ou d'appareils embarqués tels que des Arduinos[2]. Il est donc nécessaire que le modèle Kevoree permette de différencier ces différentes plates-formes d'exécution qui ont des caractéristiques spécifiques.

2. http://arduino.cc

Comme nous pouvons le voir, plusieurs caractéristiques de ce projet correspondent aux caractéristiques nécessaires pour la représentation d'un Cloud ce qui en fait un candidat intéressant pour notre problématique.

Multi-niveaux	**non**
Hébergement	**en partie**
Distribution	**oui**
Hétérogénéité	**oui**
Extensibilité	**oui**
Réfléxivité	**oui**

3.2 Les concepts de Kevoree

Nous décrivons ici les différents paradigmes de l'approche Kevoree qui permettent l'adaptation dynamique de logiciels et de la plate-forme d'hébergement. Nous allons tout d'abord, en section 3.2.1, présenter les paradigmes utilisés pour représenter un système. Puis, en section 3.2.2, nous allons présenter les paradigmes intégrés au framework de Kevoree. Afin d'expliquer à quoi correspondent ces différents concepts, nous allons prendre l'exemple d'un système de messagerie instantanée utilisée par deux utilisateurs distants (voir figure 3.1).

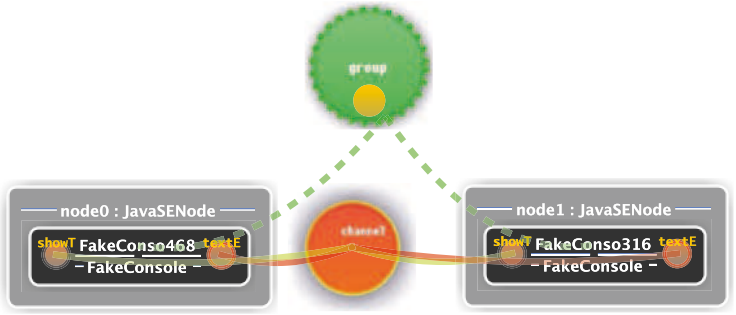


FIGURE 3.1 – Exemple d'application Kevoree

3.2.1 Paradigmes de modélisation

Composants Les composants Kevoree définissent un contrat d'interface [42]. Ce contrat définit un ensemble de fonctionnalités fournies, requises et identifiées de manière unique par un port. Un composant définit donc une ou plusieurs fonctionnalités offertes au système. Un composant A est défini comme substituable à un autre composant B si A possède au moins l'ensemble des ports que possède le composant B. Cette caractéristique offre la possibilité d'adapter le système en remplaçant un composant par un autre. Cette adaptation peut se faire en fonction de propriétés non fonctionnelles et permet de reconfigurer les applications durant la phase de conception et durant l'exécution de celles-ci tout en maintenant les fonctionnalités requises.

Dans l'exemple de l'application de messagerie instantanée, les boites noires correspondent aux composants avec un port d'entrée et un port de sortie.

Canaux de communication (*Channels*) En plus des composants, une application définit aussi les relations entre ses composants afin de les faire collaborer. Ces relations sont représentées sous forme de canaux (channels) qui encapsulent les sémantiques de communication entre composants.

Dans l'exemple, nous avons un canal de communication appelé "*channel*" qui permet de connecter l'ensemble des ports des deux composants.

Nœuds Un système distribué est caractérisé par un ensemble de nœuds de calcul. Chaque nœud peut donc héberger un ensemble de processus métier (*composants*) eux-mêmes interconnectés entre eux par des canaux de communication (*channels*) formant ainsi une application. Un nœud est donc un conteneur qui fournit un niveau d'isolation et qui est responsable localement de la synchronisation entre son modèle d'architecture et le système qui s'exécute. Cette synchronisation se caractérise par l'exécution des reconfigurations au travers des primitives d'adaptation. L'exécution de l'adaptation est le plus souvent dépendante du support d'exécution. Par exemple, l'instanciation d'un composant sur un nœud Java n'est pas la même que sur un nœud de microcontrôleur en C. C'est pourquoi les implémentations des primitives d'adaptation sont laissées à la charge du nœud. Le listing 3.1 présente le squelette de l'implémentation d'une primitive d'adaptation.

Listing 3.1 – Définition d'une primitive d'adaptation

```
public class MyConcretePrimitiveCommand extends PrimitiveCommand {
  public boolean execute() {
    // save current state to allow rollback
    // apply command
  }
  public boolean rollback() {
    // rollback to the previous state
  }
}
```

Dans l'exemple, les nœuds sont représentés par les boites grises qui contiennent les composants et sont nommés "node0" et "node1".

Groupes Un groupe est dédié à la synchronisation de la représentation par modèle d'un ensemble de nœuds. Cette synchronisation définit une portée spécifiée par un protocole de synchronisation, mais également un protocole de communication entre un ensemble de nœuds [68]. De la même façon que les *channels* sont utilisées pour définir la sémantique de communication entre les composants, les groupes sont utilisés pour définir la sémantique de communication pour la dissémination du modèle entre les nœuds. La cohérence de l'ensemble du système est donc assurée par les groupes de synchronisation.

Dans l'exemple, le groupe est nommé "*group*" et permet de synchroniser les deux nœuds entre eux. Bien que représenté comme une seule entité et tout comme les canaux de communication, un fragment du groupe est exécuté sur chacun des nœuds afin qu'il puisse communiquer et donc synchroniser la configuration des deux nœuds.

Topologie réseau La topologie réseau est représentée par un ensemble de connexion entre nœuds (NodeNetwork et NodeLinks) qui définissent les liens réseaux entre les différents nœuds. Chaque lien possède un ensemble de propriétés définissant les caractéristiques du réseau associé. Cette topologie permet ensuite aux *groupes* ou *channels* de récupérer les informations nécessaires pour établir une connexion avec les éléments distants.

Dans la figure de l'exemple, la topologie réseau n'est pas représentée, mais existe tout de même. Ici elle correspond à des adresses réseau qui peuvent être utilisées par les fragments du groupe et du canal de communication pour établir des connexions entre les fragments et ainsi envoyer des données ou synchroniser le modèle.

Patron Type/Instance Pour mettre à jour un système de façon continue, il est important de pouvoir différencier les types représentant les fonctionnalités disponibles ainsi que les paramètres utilisés pour configurer ces fonctionnalités, des instances représentant l'usage localisé de ces fonctionnalités avec un paramétrage spécifique. C'est une nécessité à la fois pour raisonner sur la substituabilité des fonctionnalités mais également pour connaître les instances concernées par une mise à jour de type. L'ensemble des concepts de Kevoree suit donc un patron de conception type/instance [118] qui, à la manière de la programmation objet, sépare les définitions (Classe) des instances (Objet).

Un type est défini par son arbre d'héritage et le type de dictionnaire qui lui est associé (voir figure 3.2). Le paramétrage d'une instance se fait au travers de son dictionnaire qui regroupe l'ensemble des attributs pouvant servir à la configuration du type correspondant. De la même façon que pour son utilisation dans les langages à objets, l'héritage de type permet de mutualiser les fonctionnalités déjà définies dans des *super types* et facilite aussi la spécialisation de ces types en permettant de définir de nouvelles fonctionnalités ou d'en redéfinir certaines.

Dans notre exemple, nous pouvons identifier quatre types pour six instances. Tout d'abord le type du groupe qui définit le type de synchronisation entre les configurations des deux nœuds. Ici ce groupe effectue de la dissémination de modèle sur l'ensemble des nœuds connectés (*broadcast*) en utilisant une couche HTTP. Les deux nœuds partagent le même type qui est le type de base dans Kevoree et qui est nommé *JavaSeNode* car il supporte des composants, canaux de communication et groupes développés dans le langage Java. Le canal de communication possède lui aussi son propre type et transmet les messages entre les composants au travers de *socket* réseau Java. Ce canal de communication possède comme propriétés de configuration, dans son dictionnaire, les ports réseau utilisés pour établir la communication entre les deux nœuds. Enfin, les deux composants possèdent le même type qui représente la console servant à émettre et

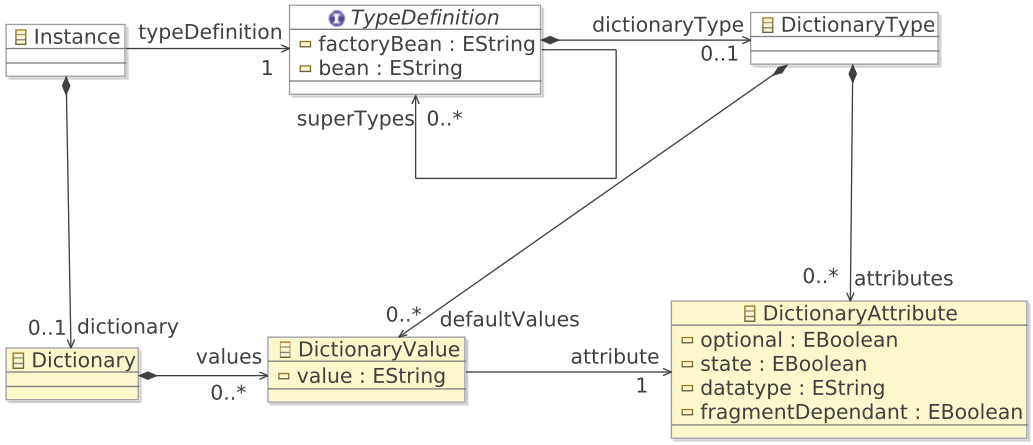


FIGURE 3.2 – Paradigme Type/Instance, dictionnaire et héritage de type

recevoir les messages entre les deux utilisateurs. Cet envoi et cette réception sont modélisés par l'intermédiaire des ports de type *message* de chaque composant. Un port de type message a pour rôle d'envoyer des données sans attendre de retour du récepteur. Il existe aussi des ports de type *service* qui eux permettent d'effectuer des communications synchrones.

3.2.2 Paradigmes du framework

Cycle de vie Afin de permettre leur usage dans un système adaptatif, les instances pouvant être définies dans Kevoree (composants, canaux de communication, nœuds et groupes) possèdent un cycle de vie similaire à celui de la figure 2.2 qui présente le cycle de vie de l'activité de déploiement. Ainsi une instance peut être installée, désinstallée, démarrée et arrêtée (correspond à l'état configuré de la figure). Le passage d'un état à un autre s'effectue par l'intermédiaire des primitives d'adaptation que possède le système. Cependant, chaque type peut définir le comportement de ses instances lors de certaines transitions. Ainsi bien que les opérations d'installation et de désinstallation restent spécifiques à la plate-forme d'exécution, les opérations de démarrage, d'arrêt et de mise à jour (correspond à la transition *adapt* dans la figure) peuvent être spécialisées par chacun des types. Dans notre exemple, le démarrage des consoles est spécialisé pour créer l'interface graphique permettant aux utilisateurs de lire et écrire des messages. Le canal de communication va, quant à lui, démarrer une *socket* réseau pour recevoir les messages provenant des composants distants. De la même façon, le groupe va lui aussi démarrer une *socket* réseau pour pouvoir recevoir les reconfigurations provenant des nœuds distants. Enfin, les nœuds vont initialiser les différents éléments pour pouvoir exécuter les primitives d'adaptation comme charger le module lui permettant de télécharger dynamiquement les librairies de composants que le modèle utilise.

Kevoree et modèle à l'exécution Kevoree est issu des travaux dans le domaine de l'Ingénierie des modèles et plus particulièrement des travaux sur les modèles à l'exécution (voir section 2.3.2.3).

Ce processus de M@R est le cœur du système Kevoree et est encapsulé dans un module appelé Kevoree Core qui est embarqué dans chacun des nœuds. Ce module offre à chaque élément du système (nœuds, composants, canaux de communication, groupes) un accès au modèle courant et leur permet aussi de soumettre de nouvelles configurations au travers de nouveaux modèles. Si le Kevoree Core reçoit un nouveau modèle, il a la charge de la validation, puis de la planification de l'adaptation à effectuer et enfin de l'exécution de cette adaptation. L'ensemble de ces étapes est effectué de manière transactionnelle.

La validation du modèle est déléguée à tout élément du système enregistré en tant que *ModelListener*. Chaque composant, canal de communication ou nœud peut déclarer cette interface et s'enregistrer au niveau du *Kevoree Core* chargé de la gestion du modèle. Une fois enregistrée, l'instance sera notifiée concernant les différentes étapes d'une reconfiguration. Pour cela, l'interface *ModelListener* définit les notifications suivantes :

- *preUpdate* permet aux *ModelListener*s d'être notifiés qu'une mise à jour a été proposée. Chaque *ModelListener* peut ainsi valider la configuration proposée.
- *preAllUpdate* permet aux *ModelListener*s d'être notifiés que la mise à jour proposée a été validée par l'ensemble des *ModelListeners*.
- *postUpdate* permet aux *ModelListeners* d'être notifiés que la mise à jour a été appliquée. Chaque *ModelListener* peut ainsi valider que la mise à jour ne pose pas de souci.
- *postAllUpdate* permet aux *ModelListener*s d'être notifiés que la mise à jour qui a été appliquée a aussi été validée par l'ensemble des ModelListeners.
- *preRollback* permet aux *ModelListener*s d'être notifiés que la mise à jour a échoué et qu'un retour à la configuration précédente va être effectué.
- *postRollback* permet au *ModelListener*s d'être notifiés que le retour à la configuration précédente a bien été effectué.

La figure 3.3 représente sous forme de diagramme états-transitions l'intégration des *ModelListeners* avec le processus de *Model@Runtime*. L'état *check* délègue la vérification à l'ensemble des *ModelListeners*. Si l'ensemble des *ModelListeners* accepte le modèle alors le système passe dans l'état *adapt* qui notifie l'ensemble des ModelListeners que le modèle a été validé et qui tente de mettre en place le modèle sur le nœud local. Si la mise en place sur le nœud local réussit alors le système passe dans l'état *validate* qui permet de sauvegarder le modèle comme étant le modèle courant du système et qui notifie l'ensemble des *ModelListeners* afin qu'ils puissent vérifier que la mise à jour a été appliquée. Si l'ensemble des *ModelListeners* valide que l'état courant correspond bien au modèle proposé alors le système passe dans l'état *terminate* qui notifie à tous les *ModelListeners* que la mise à jour a été appliquée avec succès. Outre ces transitions du cas nominal, il y a aussi des transitions pour les cas particuliers. Par exemple, dans l'état *check*, si l'un des *ModelListeners* refuse le modèle alors le système passe dans l'état *rollback* qui sert à revenir en arrière. De la même façon, si le modèle ne peut être appliqué sur le nœud local, le système passe aussi dans l'état *rollback*. Enfin, dans l'état

validate, l'un des *ModelListeners* ne valide pas que l'état courant correspond bien au modèle proposé, alors le système passe là encore dans l'état *rollback*. Cet état *rollback* permet dans ces différents cas de revenir sur l'ancien modèle, c'est-à-dire d'annuler toutes les modifications qui ont pu être mises en place afin d'appliquer la nouvelle configuration. Pour cela, les *ModelListeners* sont notifiés au début du rollback pour qu'ils puissent eux aussi annuler les opérations qu'ils ont pu commencer. À la fin du *rollback*, les *ModelListeners* sont aussi notifiés pour les informer que la configuration précédente a bien été réappliquée.

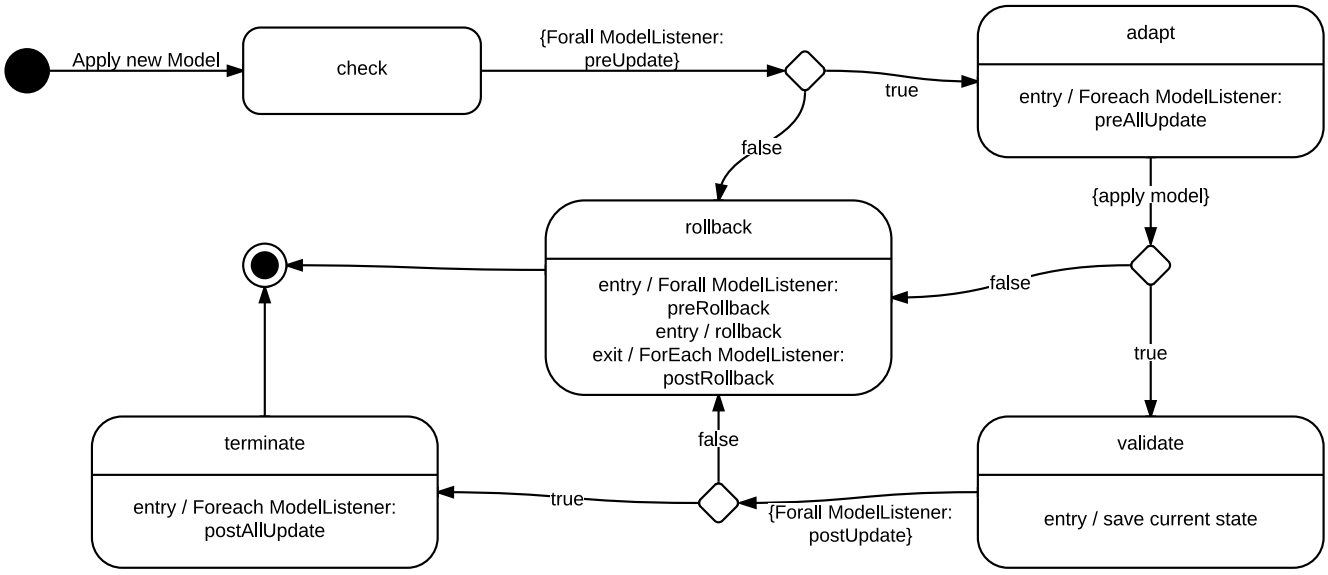


FIGURE 3.3 – Diagramme d'états-transitions montrant l'intégration des ModelListeners dans le processus de Model@Runtime

Les groupes sont par défaut des instances de *ModelListeners*. Contrairement aux composants, aux canaux de communication et aux nœuds qui peuvent implanter l'interface *ModelListener* pour être notifiés des changements locaux, les groupes implantent par défaut cette interface et sont automatiquement enregistrés au niveau du Kevoree Core. Ces instances particulières de *ModelListener* sont utilisées pour la synchronisation et la validation des reconfigurations entre plusieurs nœuds. En effet, chaque nœud héberge un module Kevoree Core lui permettant d'évoluer indépendamment des autres nœuds. Mais les éléments présents sur chacun des nœuds sont capables de modifier l'ensemble du système et si la reconfiguration impact plusieurs nœuds, il peut être nécessaire que plusieurs nœuds valident le modèle et/ou se synchronisent lors de sa mise

en place.

Outre la validation du modèle par l'intermédiaire des *PreUpdate*, l'ensemble des notifications sert aussi à la génération d'évènements dans le cadre de la phase d'observation du modèle MAPE (voir 1.2) pour pouvoir informer les différents moteurs de décision présents dans le système qu'une mise à jour soit en cours, qu'elle a été effectuée ou qu'elle a échoué.

3.3 Kevoree et ses outils

En plus du méta-modèle, le projet Kevoree fournit un ensemble d'outils permettant d'une part de concevoir et de manipuler un modèle représentant l'architecture d'un système (voir section 3.3.1 et 3.3.2) et d'autre part plusieurs *frameworks* de conception selon l'usage (voir section 3.3.3).

3.3.1 Un langage graphique

Kevoree propose un langage graphique au travers d'un éditeur pour la conception de l'architecture d'application (voir figure 3.4). Il permet de définir la liste des types disponibles dans le modèle (le 1 sur la figure 3.4) et de manipuler (ajouter, supprimer) les instances présentes dans le modèle (le 2 sur la figure 3.4). Il permet aussi de configurer chacune des instances du modèle et de déployer ou récupérer, grâce aux groupes présents dans le modèle, le modèle sur un nœud en cours d'exécution (le 3 sur la figure 3.4).

FIGURE 3.4 – Interface de l'éditeur Kevoree

3.3.2 KevScript : un langage textuel

Kevoree propose aussi un langage textuel permettant de manipuler les entités pouvant apparaître dans un modèle que ce soit les types ou les instances et leur configuration (voir listing 3.2).

Listing 3.2 – Liste des primitives de KevScript

```
merge "mvn:org.kevoree.corelibrary.javase/org.kevoree.library.javase.javasenode" // include new type(s)
updateDictionary instance1 {port="8080"} // update instance's properties
addComponent component1@node1:ComponentType1 // add a component instance
addChannel channel1 : ChannelType1 // add a channel instance
bind component1.port1@node1 →channel1 // bind a port with a channel
unbind component3.port1@node1 →channel1 // unbind a port with a channel
removeComponent component3@node1 // remove a component instance
removeChannel channel2// remove a channel instance
moveComponent component1@node1 →node2 // migrate a component from a node to another
addGroup group1 : GroupType1 // add a group instance
removeGroup group1 // remove a group instance
addToGroup node1@group1 // add a node as a child of a group
removeFromGroup node1@group1 // remove a node on the childs of a group
addNode node1 : NodeType1 // add a node instance
removeNode node1 // remove a node instance
network node1 →node2 {"ip" = "192.168.0.1"} // define a directed network link
// following primitives have been added according to the work on Cloud stuffs
addNode node1 : NodeType1
removeNode node1
addChild node2@node1
removeChild node2@node1
```

Dans la construction de système d'adaptation, il est nécessaire d'implanter des moteurs de décision chargés de la phase d'analyse du modèle MAPE. Dans notre approche, ces moteurs de raisonnement doivent construire de nouvelles configurations correspondant à de nouveaux modèles. Le langage *KevScript* permet de définir facilement de nouvelles configurations.

3.3.3 Les frameworks de conception

Kevoree fournit un ensemble de frameworks de conception pour la définition de type. Il fournit notamment un framework pour la conception de type pour microcontrôleur, un framework pour la conception de type natif (en C et C++) et un framework pour la conception Java utilisable aussi pour la conception sur Android. Nous allons ici nous intéresser à celui pour Java puisque c'est celui-ci qui sera ensuite utilisé dans la définition des types de nœuds pour le Cloud.

Ce framework est composé d'un ensemble d'annotations ainsi qu'un ensemble de classes abstraites. Les classes abstraites (voir figure 3.5) fournissent un ensemble de fonctionnalités utiles pour l'implantation des types que ce soit l'accès au dictionnaire de propriétés ou l'accès à ses ports pour les composants. Ces abstractions offrent aussi un début d'implantation pour le fonctionnement de l'instance. Par exemple, les canaux de communication possèdent des méthodes pour le transfert de messages vers les ports de composants locaux connectés au canal. Encore pour les canaux de communication, l'abstraction impose la manière dont doivent être implantées les communications en proposant deux méthodes abstraites qu'il est nécessaire d'implanter lors de la conception d'un canal de communication. De la même façon, les types de nœuds doivent implanter les méthodes servant à la mise en place des reconfigurations.

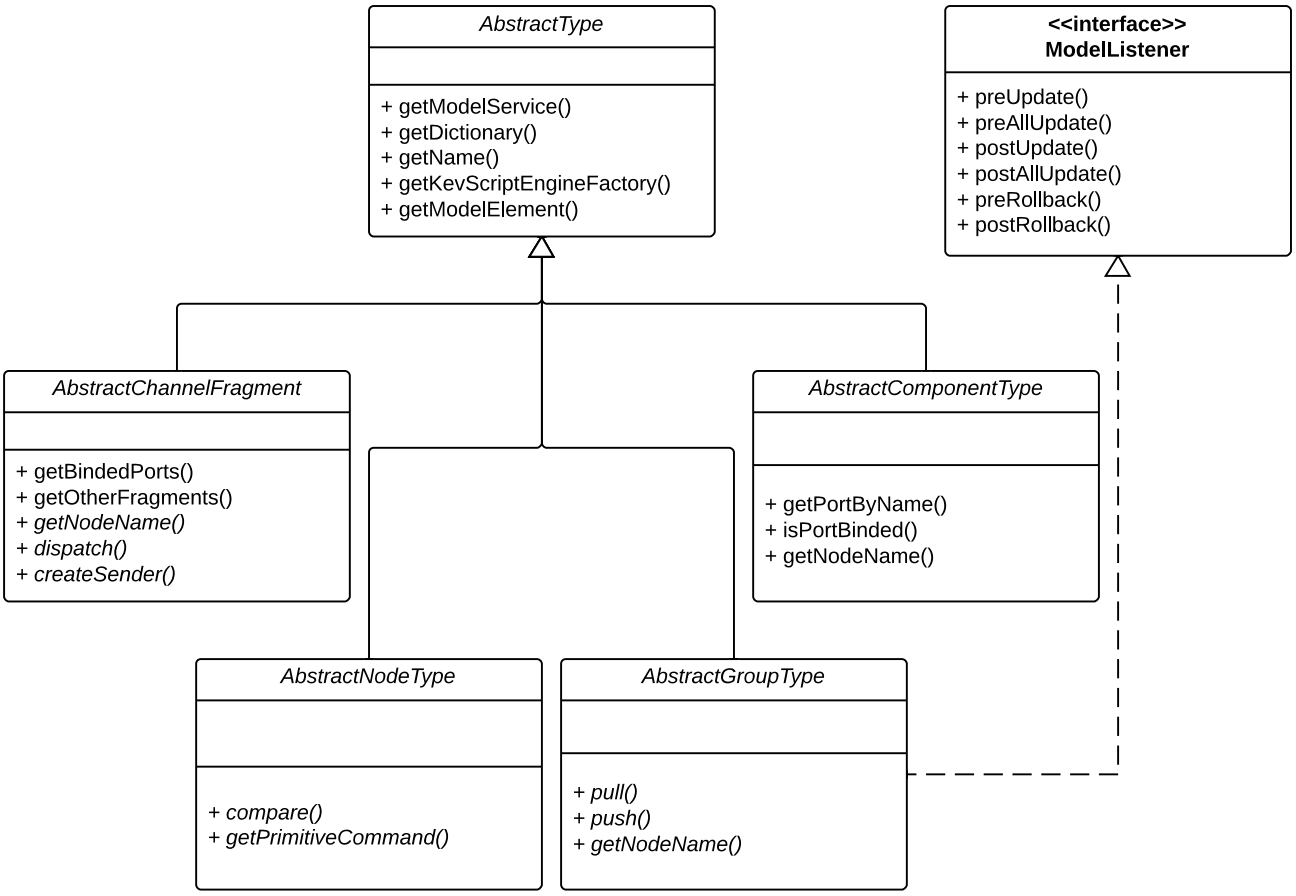


FIGURE 3.5 – Diagramme de classe des types abstraits

Les annotations quant à elles permettent de faire le lien entre le modèle abstrait et l'implantation concrète des types. Ces annotations peuvent être réparties en deux catégories. La première concerne les annotations pour la définition du type. Ces annotations permettent tout d'abord d'exprimer quel genre de type correspond à cette implantation (voir listing 3.3).

Listing 3.3 – Annotation de type

```
@ComponentType
public class MyComponentType extends AbstractComponentType {}
@NodeType
public class MyNodeType extends AbstractNodeType {}
@GroupType
public class MyGroupType extends AbstractGroupType {}
@ChannelTypeFragment
public class MyChannelType extends AbstractChannelFragment {}
```

Il existe aussi les annotations servant à définir les propriétés de type qui pourront ensuite être définies précisément pour chaque instance de ce type. Pour chacune des propriétés, il est possible de définir un nom, une liste de valeurs possibles, une valeur par défaut, la cardinalité (0 ou 1) et si cette propriété est dépendante du fragment en cours d'exécution. Cette dernière information est notamment utile sur les canaux de communication et les groupes pour par exemple pouvoir spécifier autant de ports réseau qu'il y a de nœuds sur lesquels l'instance est fragmentée.

Dans le listing 3.4, le type *MyType* définit une propriété appelée *myName*, dont la valeur par défaut est *choice2* et dont les valeurs possibles sont *choice1* et *choice2*. Cette propriété est définie comme optionnelle, c'est-à-dire qu'elle n'est pas nécessairement spécifiée pour chaque instance. Enfin, cette propriété n'est pas une propriété dépendante d'un fragment.

Listing 3.4 – Définition du dictionnaire de type

```
@DictionaryType ({
  @DictionaryAttribute (
    name = "myName", vals = {"choice1", "choice2"},
    defaultValue = "choice2", optional = true,
    fragmentDependant = false
  )
})
@ComponentType // or @NodeType or @ChannelTypeFragment or @GroupType
public class MyType {}
```

Enfin, parmi les annotations pour la définition de type, il existe aussi celles qui sont spécifiques aux composants et qui permettent de spécifier les ports disponibles. Dans le listing 3.5, le type *MyComponentType* définit deux ports : un port fourni et un port requis. Le port requis est appelé *port1* et est de type service. L'interface de ce service est définie par l'interface *MyService* et ce port est optionnel ce qui signifie que le composant pourra quand même fonctionner même si ce port n'est pas connecté vers un autre composant au travers d'un canal de communication. Le port fourni quant à lui s'appelle *port2* et est de type message.

Listing 3.5 – Définition des ports

```
@Requires ({
  @RequiredPort(
    name = "port1", type = PortType.SERVICE,
    className = MyService.class, optional = true)
})
@Provides ({
  @ProvidedPort(
    name = "port2", type = PortType.MESSAGE)
})
@ComponentType
public class MyComponentType extends AbstractComponentType {}
```

La deuxième catégorie d'annotation concerne l'exécution des instances. Les pre-

mières permettent de spécifier les opérations associées au cycle de vie d'une instance. Dans le listing 3.6, le type *MyType* définit les opérations *startInstance*, *stopInstance* et *updateInstance* comme étant les opérations atomiques correspondant respectivement au démarrage, à l'arrêt et à la mise à jour d'une instance de type *MyType*.

Listing 3.6 – Annotation du cycle de vie

```
public class MyType {
  @Start
  public void startInstance() {}
  @Stop
  public void stopInstance() {}
  @Update
  public void updateInstance() {}
}
```

La dernière annotation est utilisée pour définir le lien entre un port qui fournit une fonctionnalité et la méthode qui implante cette fonctionnalité. Dans le listing 3.7, la méthode *doSomethingAsAService* est définie comme la méthode implantant la fonctionnalité du service *myServiceMethod* du port *port1*. La méthode *doSomething* quant à elle correspond à la fonctionnalité offerte par le port de type message *port2*.

Listing 3.7 – Définition de la relation entre les ports et l'implémentation

```
public class MyType extends AbstractComponentType {
  @Port{ name = "port1", method = "myServiceMethod"}
  public void doSomethingAsAService(...) {}{}
  @Port{ name = "port2"}
  public void doSomething(Object message) {}{}
}
```

3.4 Extensibilité de Kevoree

Kevoree fournit un ensemble d'implantations pour la définition de systèmes distribués que ce soit au niveau des groupes pour la synchronisation des reconfigurations (Gossip, Paxos, broadcast), que ce soit au niveau des nœuds d'exécution (Java, Android, microcontrôleur, C/C++) ou encore au niveau des canaux de communication (Gossip, de broadcast). En plus de ces implantations, Kevoree permet de définir de nouveaux protocoles de dissémination et de communication, mais aussi la possibilité de définir de nouveaux nœuds d'exécution. C'est cette capacité à pouvoir définir de nouveaux supports d'exécution qui permettent l'extension de Kevoree notamment dans le cadre de la gestion de l'adaptation. En effet, les nœuds sont les dépositaires de deux des phases de la boucle autonomique présentée en section 1.2. Ces phases, la planification et l'exécution, permettent d'étendre les capacités du système par la définition de nouvelles primitives d'adaptation et par leur intégration dans la construction des plans d'exécution.

3.4.1 Délégation de l'exécution de l'adaptation

L'exécution de l'adaptation dans Kevoree est dépendante du support d'exécution et c'est pourquoi ce sont les types de nœuds qui définissent les implantations des primitives d'adaptation. Mais chaque support d'exécution peut avoir ces propres capacités d'adaptation. En effet, si l'on considère un nœud Java permettant d'exécuter un ensemble de composants dans une machine virtuelle Java, ces capacités d'adaptation se cantonnent à la manipulation de composants et de *channels* alors qu'un nœud capable de gérer une infrastructure de Cloud doit être capable de manipuler non seulement des composants chargés de son administration, mais aussi des nœuds représentant les machines virtuelles correspondant aux plates-formes.

C'est pourquoi le processus de Model@Runtime implanté dans Kevoree délègue aux nœuds, la déclaration de l'ensemble des capacités d'adaptation qu'ils fournissent (voir figure 3.6).

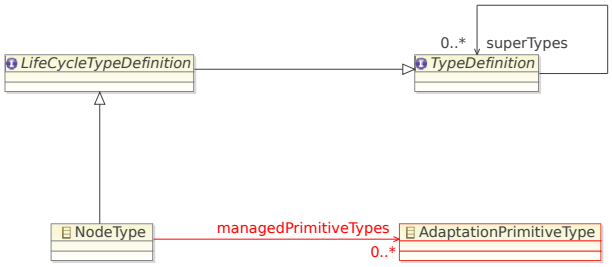


FIGURE 3.6 – Méta-modèle Kevoree avec la représentation des primitives d'adaptation

Ainsi en plus des propriétés de son dictionnaire, un type de nœud spécifie ces primitives d'adaptation. Dans Kevoree, le nœud par défaut appelé *JavaSeNode* est capable de manipuler les composants et canaux de communication ainsi que les groupes de synchronisation (voir listing 3.8). Ces primitives définissent de manière abstraite les capacités d'adaptation disponibles sur un type de nœud.

Listing 3.8 – Déclaration du type de base dans Kevoree

```
// Default node type on Kevoree
@PrimitiveCommands(
  values = {"UpdateType", "UpdateDeployUnit", "AddType", "AddDeployUnit", "AddThirdParty",
  "RemoveType", "RemoveDeployUnit", "UpdateInstance", "UpdateBinding", "UpdateDictionary",
  "AddInstance", "RemoveInstance", "AddBinding", "RemoveBinding", "AddFragmentBinding",
  "RemoveFragmentBinding", "UpdateFragmentBinding", "StartInstance", "StopInstance",
  "StartThirdParty", "RemoveThirdParty"})
@NodeType
public class JavaSENode {
}
```

3.4.2 Délégation de la planification de l'adaptation

Du fait que les actions d'adaptation sont dépendantes du support d'exécution, seul le nœud est capable d'identifier quelles actions permettent de passer d'un modèle à un autre. C'est pourquoi Kevoree délègue la comparaison de modèle au nœud. Cette comparaison fait partie de la phase de planification définie dans le modèle MAPE (voir section 1.2).

Mais la phase de planification du modèle MAPE ne se limite pas seulement aux choix des actions nécessaires pour appliquer une adaptation. Elle consiste aussi à ordonnancer les actions. En effet, à partir du modèle de la figure 3.7 dans lequel un composant a de type A envoie des données à un canal de communication y de type Y et ce canal de communication transmet ces données à un composant b de type B, la mise en place de ce modèle correspond aux actions suivantes : *installer le type A*, *installer le type B*, *installer le type Y*, *installer l'instance a*, *installer l'instance b*, *installer l'instance y*, *connecter a avec y*, *connecter y avec b*, *démarrer l'instance a*, *démarrer l'instance b*, *démarrer l'instance y*.

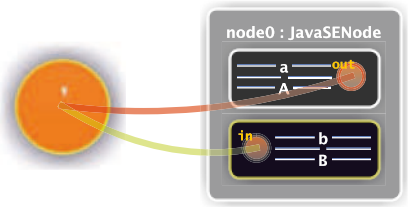


FIGURE 3.7 – Exemple de configuration nécessitant une planification

Ici, l'ordre d'exécution de cet ensemble d'actions a une importance, car si a démarre avant y et qu'il envoie des données, celles-ci ne seront peut-être pas reçues si y n'est pas démarré et de manière identique, il faut que b soit démarré avant y.

Tout comme la comparaison de modèle est spécifique au type de nœud, l'ordonnancement des actions est, elle aussi, spécifique. En effet, les dépendances entre les actions dépendent fortement des implantations des actions qui elles-mêmes sont spécifiques au type de nœud. C'est pourquoi le processus du Kevoree Core qui délègue la comparaison et l'exécution au nœud délègue aussi l'ordonnancement.

La définition d'un type de nœud se traduit donc par la spécification des primitives d'adaptation, du dictionnaire, des méthodes de cycle de vie et enfin les méthodes de planification et d'exécution (voir listing 3.9). La définition de ces différents éléments est à la charge des concepteurs de services de Cloud et plus particulièrement ici les concepteurs de services pour l'infrastructure et la plate-forme.

Listing 3.9 – Interface pour la définition d'un nouveau nœud

```
@PrimitiveCommands(
  values = {}//TODO
)
@DictionaryType({
    @DictionaryAttribute(),//TODO
})
@NodeType
public class MyNodeType extends AbstractNodeType {
  @Start
  @Override
  public void startNode() {
    // initialize internal stuffs
  }
  @Stop
  @Override
  public void stopNode() {
    // release internal stuffs
  }
  @Override
  public AdaptationModel plan(ContainerRoot current, ContainerRoot target) {
    // TODO compare models and schedule reconfiguration
  }
  @Override
  public org.kevoree.api.PrimitiveCommand getPrimitive(AdaptationPrimitive adaptationPrimitive) {
    // TODO return the concrete command which corresponds to the abstract one
  }
}
```

Troisième partie

Contribution

Chapitre 4

KevoreeKloud : Une extension de Kevoree pour le Cloud

L'objectif de cette thèse est de proposer pour un architecte de Cloud une abstraction capable de modéliser et d'administrer un Cloud c'est-à-dire l'ensemble des entités présentes sur les différents niveaux que sont l'infrastructure, la plate-forme et les applications. Cette abstraction doit permettre la manipulation de l'ensemble du système afin de pouvoir envisager l'adaptation comme une problématique transverse aux différents niveaux.

Pour pouvoir envisager l'adaptation d'un Cloud de manière transverse, il est nécessaire de pouvoir accéder aux informations des différents niveaux qui constituent un Cloud afin d'avoir une représentation globale du système et ainsi prendre en compte l'ensemble de ces informations pour effectuer de l'adaptation efficace et cohérente entre les niveaux. C'est pourquoi il est nécessaire d'avoir une abstraction capable de représenter aussi bien l'infrastructure, la plate-forme et l'application, mais aussi les interactions entre ces différents niveaux.

Dans la section 2.1, j'ai présenté un ensemble de caractéristiques que doit fournir une abstraction utilisée pour modéliser un Cloud afin de pouvoir gérer l'adaptation comme une problématique transverse aux différents niveaux. Il a ensuite été montré dans le reste du chapitre 2 qu'il existe plusieurs approches pour faire de l'abstraction de systèmes logiciels, mais aucun ne permet de modéliser à la fois les applications, les plates-formes qui hébergent ces applications et encore moins les infrastructures sur lesquelles s'exécutent ces plates-formes.

Il existe de nombreuses implantations de Cloud que ce soit pour l'infrastructure ou pour la plate-forme, mais celles-ci n'envisagent pas l'adaptation de manière transverse aux différents niveaux mais seulement pour elles-mêmes. Par exemple, Amazon EC2 fournit de quoi adapter la plate-forme, mais gère l'adaptation de l'infrastructure de manière autonome. C'est pourquoi même si l'abstraction présentée dans ces travaux permet de représenter des solutions existantes, il parait nécessaire de construire de nouvelles implantations tenant compte de l'adaptation comme une problématique transverse et fournissant des capacités d'adaptation utilisables par des systèmes externes dont la tâche

est bien de gérer l'adaptation de manière transverse.

Dans ce chapitre, une extension de Kevoree appelée KevoreeKloud est proposée afin de pouvoir modéliser un Cloud dans son ensemble. Comme nous l'avons présenté dans le chapitre 3, Kevoree répond à un certain nombre de critères (Distribution, Hétérogénéité, Extensibilité, Réflexivité désynchronisable) qui sont nécessaires pour gérer l'adaptation de manière transverse. KevoreeKloud permet de combler les différents manques par rapport aux caractéristiques précédemment définies (Multi-niveaux, Hébergement). De plus cette extension fournit un framework pour faciliter la conception de nouvelles implantations de Cloud et faciliter l'intégration de solutions déjà existantes. Enfin bien qu'il soit nécessaire d'avoir une représentation globale d'un Cloud pour pouvoir envisager son adaptation comme une problématique transverse aux différents niveaux qui le composent, tous les acteurs du domaine du Cloud Computing ne sont pas prêts à fournir toutes les informations concernant leurs implantations. C'est pourquoi il est nécessaire de pouvoir limiter les informations fournies à l'ensemble du système tout en permettant aux systèmes d'adaptations d'obtenir suffisamment d'information. C'est pourquoi ces travaux proposent une première approche permettant de gérer la précision de l'abstraction afin de masquer certaines informations à certains intervenants dans le système.

En section 4.1, KevoreeKloud est présentée avec l'ajout dans le modèle de la notion d'hébergement de nœuds par un nœud et la définition d'un framework de modélisation de Cloud contenant un ensemble de types abstraits spécifiques permettant ensuite d'avoir des notions d'équivalence entre solutions de même type (Infrastructure ou Plate-forme). La section 4.2 présente comment peuvent être définis les systèmes d'adaptation dans le cadre d'un Cloud. Cette section s'attarde notamment sur la phase de planification qui permet de faire le lien entre les reconfigurations proposées par les algorithmes de la phase d'analyse du modèle MAPE et l'exécution des primitives d'adaptation capables de reconfigurer le système en cours d'exécution. Enfin la section 4.3 présente comment Kevoree permet d'avoir une gestion du niveau de granularité du système permettant de sélectionner ce que chaque entité du système est capable de voir et propose trois types de partitionnement de modèle pour le Cloud Computing.

4.1 Définition d'un Cloud

Dans cette section, sont présentés les différents éléments de KevoreeKloud permettant de définir un support d'exécution correspondant dans le modèle à un nœud d'hébergement. Tout d'abord, la section 4.1.1 définit la notion de support d'exécution pour les nœuds dans le modèle Kevoree. Puis la section 4.1.2 présente les notions d'héritage et de composition utilisées pour définir une abstraction pour les types de nœuds utilisés dans le Cloud.

4.1.1 L'abstraction du support d'exécution : la notion d'hébergement

Bien que les niveaux Infrastructure et Plate-forme sont explicitement différents dans le domaine du Cloud et notamment au niveau des capacités d'adaptation disponibles, ils

possèdent tout de même certaines caractéristiques communes. En effet, chacun correspond à un support d'exécution que ce soit l'infrastructure qui supporte l'exécution des plates-formes ou la plate-forme qui supporte l'exécution des applications. Ce support d'exécution aussi nommé hébergement d'entités logicielles consiste donc à fournir les ressources nécessaires pour le bon fonctionnement d'une ou plusieurs entités logicielles.

La plupart des solutions décrites dans le chapitre 2.4 ne permettent pas de représenter cette notion d'hébergement nécessaire dans le cadre de système de Cloud. Cette notion d'hébergement est pourtant primordiale pour une abstraction dont l'objectif est de pouvoir modéliser l'ensemble des niveaux d'un Cloud et les dépendances entre ces niveaux.

Kevoree est capable de modéliser des applications et la plate-forme qui héberge ces applications, mais il ne propose pas de quoi modéliser la plate-forme sur l'infrastructure. Pour cela, KevoreeKloud propose une notion d'hébergement de nœuds sur un nœud (voir figure 4.1). Ainsi, un nœud est non seulement capable d'héberger des composants logiciels pouvant servir à la gestion de la plate-forme ou de l'infrastructure ainsi que des composants logiciels correspondant à des applications SaaS mais il est aussi capable d'héberger d'autres nœuds.

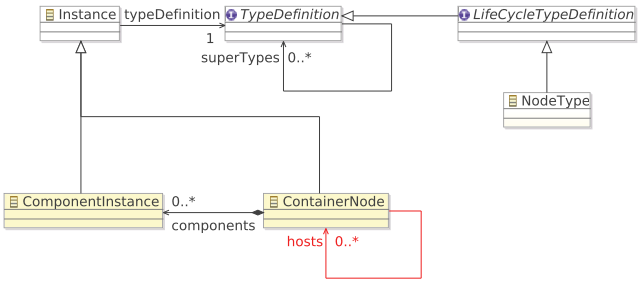


FIGURE 4.1 – Métamodèle Kevoree avec la représentation des nœuds hébergés

La figure 4.2 présente un diagramme d'instance correspondant à ce diagramme de classe. Ce diagramme d'instance présente l'exemple du serveur web décrit dans la section 3.2 lequel est déployé sur une infrastructure de Cloud (voir figure 4.3). Nous pouvons voir que le nœud *iaaSNode0* héberge les nœuds *node0* et *node1* et que ceux-ci hébergent les composants du serveur web.

4.1.2 Héritage et composition : mutualiser la conception de type

Dans la section 3.4, nous avons montré quels sont les points d'extension que propose Kevoree concernant les nœuds d'exécution. Outre le fait de pouvoir définir un type de nœud en définissant la phase de planification et l'implantation des primitives d'adaptation, Kevoree fournit aussi la notion de composition de type et d'arbre d'héritage.

Composition des caractéristiques Les propriétés d'un type de nœud permettent de définir ses capacités d'adaptation ainsi que ses propriétés fonctionnelles voire non fonc-

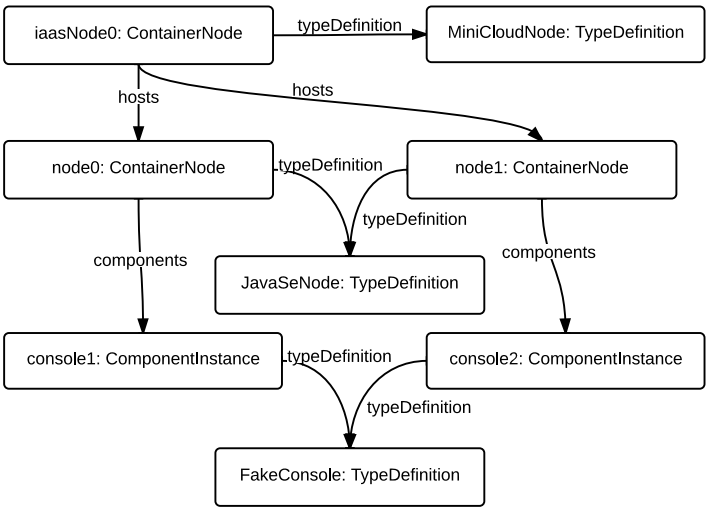


FIGURE 4.2 – Diagramme d'instance pour l'hébergement du serveur web

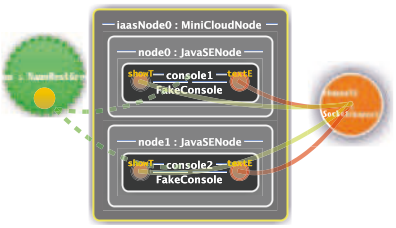


FIGURE 4.3 – Hébergement du serveur web

tionnelles. La composition de types permet donc de combiner l'ensemble des propriétés de plusieurs types dans la conception d'un nouveau type. Cette composition offre ainsi la possibilité de définir des hiérarchies de types et permet aussi de définir des niveaux d'équivalence entre types concrets par rapport à des types abstraits.

Par exemple, il est intéressant de pouvoir distinguer quels types correspondent à des types d'infrastructure. Pour cela, il est possible de regarder ces propriétés et de vérifier qu'il est bien capable de faire de la gestion de nœud (ajout et suppression de nœuds). Mais si les caractéristiques nécessaires pour identifier un type sont plus complexes alors la vérification le devient aussi. Avec la composition de types, la vérification se cantonne à la vérification des *supertypes*. Nous pouvons comparer la composition de type à la notion d'interface dans les langages orientés objets ou à la composition de trait dans les langages tels que Scala [102].

Héritage et surcharge de l'implantation Tout comme la composition de types abstraits, l'héritage de type permet de composer les propriétés. Mais l'intérêt de son utilisation réside surtout sur la mutualisation des implantations. En effet, si l'on prend l'exemple des solutions d'infrastructure, les technologies de virtualisation ne sont pas

si nombreuses et donc les actions associées sont identiques peu importe la solution d'infrastructure à partir du moment où celles-ci utilisent la même technologie. De la même façon, même si les actions ne possèdent pas la même implantation, elles peuvent avoir les mêmes dépendances et donc nécessiter la même planification.

Comme cela a déjà été dit, chaque solution d'infrastructure doit être capable de manipuler les nœuds de plate-forme. Cette manipulation de nœuds se traduit par la définition de deux primitives d'adaptation concernant l'ajout et la suppression de nœuds. Ainsi, chaque solution d'infrastructure va devoir implanter ces deux actions. Il en va de même pour n'importe quelles actions d'adaptation.

Si une solution hérite d'une autre afin de remplacer l'implantation d'une des actions, elle doit tout d'abord définir la nouvelle action concrète définie par l'interface *PrimitiveCommand* (voir listings 3.1). Une fois cette action implantée, la solution doit surcharger l'exécution de l'action abstraite en remplaçant la *PrimitiveCommand* héritée par la nouvelle (voir listings 4.1).

Listing 4.1 – Surcharge des primitives d'adaptation d'un type de nœud

```
@NodeType
public class MyNodeType extends MyParentType {
  @Override
  public org.kevoree.api.PrimitiveCommand getPrimitive(AdaptationPrimitive adaptationPrimitive) {
    if ("MYPRIMITIVE".equals(adptationPrimitive.getName())) {
      // TODO return the concrete command which corresponds to the abstract one
    } else {
      super.getPrimitive();
    }
  }
}
```

Dans la suite de cette section, nous allons présenter différents types abstraits pouvant être intéressants dans la définition de type de nœud pour le Cloud. Tout d'abord, la section 4.1.2.1 présente les abstractions de type permettant d'intégrer les solutions existantes grâce à la définition de nœud servant de proxy. Puis la section 4.1.2.2 présente les types abstraits communs aux types de nœud de Cloud que ce soit infrastructure ou plate-forme. Ensuite la section 4.1.2.3 décrit les types abstraits permettant de définir des types d'infrastructure. La section 4.1.2.4 présente les types abstraits permettant de définir des types de plate-forme. Enfin la section 4.1.2.5 présente des extensions aux types précédemment décrits permettant de définir de nouvelles propriétés pouvant être intéressantes dans le cadre d'un Cloud.

4.1.2.1 Les caractéristiques pour un type de nœud hébergeant des nœuds

La définition d'une nouvelle implantation d'infrastructure ou de plate-forme nécessite de pouvoir modéliser l'ensemble des caractéristiques des ressources et des capacités de reconfiguration afin de pouvoir en tirer parti dans le cadre de l'adaptation. Cependant, les implantations déjà existantes ne fournissent généralement pas suffisamment

d'information sur leur fonctionnement pour pouvoir les intégrer de la même manière qu'une nouvelle implantation. Pourtant, il est nécessaire de pouvoir intégrer ces implantations existantes afin de pouvoir les utiliser. Pour cela, il est possible de définir des nœuds *proxy* avec l'implantation existante même si ce *proxy* n'offre pas de quoi adapter cette solution.

La similitude entre une implantation d'infrastructure et un *proxy* vers une implantation d'infrastructure se résume à la manipulation des nœuds de plate-forme, c'est-à-dire la capacité de pouvoir ajouter, supprimer et mettre à jour des nœuds ainsi que les démarrer et les arrêter. Pour cela, un type HostNode a été défini et offre ces primitives d'adaptation (voir listing 4.2). Un proxy vers une implantation de plate-forme doit être capable de manipuler les applications c'est-à-dire les composants et les canaux de communication ainsi que les groupes de synchronisation. Ce genre de proxy existe déjà au travers de l'implantation de base proposée dans Kevoree sous la forme du nœud *JavaSeNode*.

Listing 4.2 – Type de nœud capable d'héberger d'autres nœuds

```
@PrimitiveCommands(values = {"REMOVE_NODE", "ADD_NODE", "UPDATE_NODE",
    "STOP_NODE", "START_NODE"})
// This type is not instanciable but it provides some needed information for type that inherits of it
@NodeFragment
public interface HostNode {}
```

Outre ces capacités, les implantations existantes offrent un point d'accès pour la communication le plus souvent au travers d'une API REST et utilisant des identifiants de sécurité afin de permettre l'authentification de l'utilisateur. Pour cela, une interface possédant ces caractéristiques a été définie.

Au final, un *proxy* pour une infrastructure sera composé du type *ProxyNode* et *HostNode* et un *proxy* pour une plate-forme sera composé du type *ProxyNode* et *JavaSeNode* (voir listing 4.3).

Listing 4.3 – Types de nœud proxy

```
@DictionaryType({
  @DictionaryAttribute(name = "ENDPOINT")
  @DictionaryAttribute(name = "CREDENTIALS")
@NodeFragment
public interface ProxyNode {}
@NodeFragment
public interface ProxyIaaSNode extends ProxyNode,HostNode {}
@NodeFragment
public abstract class ProxyPaaSNode extends JavaSeNode implements ProxyNode {}
```

4.1.2.2 Les caractéristiques communes des types de nœuds pour le Cloud

Même si les ressources gérées par une infrastructure et les ressources manipulées par une plate-forme ne sont pas les mêmes, elles possèdent plusieurs caractéristiques communes (voir listing 4.4). En effet, que ce soit au niveau infrastructure ou plate-forme, une ressource de calcul peut être caractérisée par l'architecture matérielle à laquelle elle correspond. Elle peut aussi être caractérisée par son processeur tant en termes de fréquence qu'en nombre de cœurs ainsi que par la quantité de mémoire vive disponible et de la même façon la capacité de stockage sur disque. Enfin, la configuration de cette ressource peut aussi être définie en termes de système d'exploitation et de configuration réseau. Ces informations sur le réseau n'apparaissent pas dans le dictionnaire du type puisqu'elles sont modélisées dans le modèle par des *NodeLinks*.

Listing 4.4 – Dictionnaire de propriété pour un type de nœud Cloud

```
@DictionaryType({
  @DictionaryAttribute(name = "ARCH", defaultValue = "x86", vals = {"x86", "x86_64", ...}),
  @DictionaryAttribute(name = "RAM", defaultValue = "N/A"),
  @DictionaryAttribute(name = "CPU_FREQUENCY", defaultValue = "N/A"),
  @DictionaryAttribute(name = "CPU_CORE", defaultValue = "N/A"),
  @DictionaryAttribute(name = "OS")
})
@NodeFragment
public interface CloudNode extends HostNode {}
```

4.1.2.3 Les caractéristiques pour un nœud d'infrastructure

Nous définissons ici une infrastructure comme à un ensemble de machines (nœuds) sur lesquelles peuvent s'exécuter des systèmes virtualisés.

Il existe plusieurs types de virtualisation. La plus simple et la plus commune est la virtualisation d'applications qui permet d'exécuter simultanément et sur la même machine une ou plusieurs applications correspondant chacune à un processus système ayant son propre système d'adressage mémoire, mais pour qui le processeur, l'adressage réseau et le système de stockage sont partagés avec l'ensemble des processus systèmes. Ici la plate-forme serait un serveur d'applications comme Apache Tomcat par exemple. Il existe aussi de la virtualisation d'espace utilisateur qui permet d'exécuter simultanément et sur la même machine un ou plusieurs espaces utilisateurs correspondant à un ensemble de processus système. La virtualisation d'espaces utilisateurs permet de séparer entre autres l'espace de stockage sur disque, mais aussi l'adressage réseau et offre généralement des capacités de limitation sur l'utilisation des ressources permettant de limiter l'impact d'un espace utilisateur sur l'ensemble de la machine et donc sur les autres espaces utilisateurs. Il existe aussi la virtualisation de système d'exploitation. La virtualisation de système d'exploitation consiste à virtualiser des ressources matérielles afin que le système d'exploitation ait l'impression de s'exécuter de manière native sur des ressources matérielles. Ce type de virtualisation peut être découpé en deux avec la virtualisation de type 2 qui virtualise le matériel dans une application du système hôte alors que la

virtualisation de type 1 gère directement les ressources matérielles sans passer par un système d'exploitation hôte.

En omettant la virtualisation d'application qui reste une virtualisation très limitée, quelques propriétés permettant de caractériser un nœud d'infrastructure (voir listing 4.5) ont été définies. Un nœud d'infrastructure peut tout d'abord spécifier le réseau sur lequel peuvent être allouées les adresses IP des machines virtuelles correspondant aux plates-formes. En effet, dans la majorité des solutions d'infrastructure, l'infrastructure alloue à chaque nœud de la plate-forme une adresse IP (publique ou privée) unique permettant de se connecter par la suite à ce nœud. Si cette adresse IP est spécifiée par l'utilisateur, la spécification de ce réseau sur les nœuds d'infrastructure permet de définir des zones pour allouer telle ou telles plages d'adresses IP.

Listing 4.5 – Type de nœud d'infrastructure

```
@DictionaryType({
    @DictionaryAttribute(name = "inet"),
    @DictionaryAttribute(name = "subnet"),
    @DictionaryAttribute(name = "mask", vals= {"0", "1",...,"31"})
})
@NodeFragment
public abstract class IaaSNode extends JavaSENode implements CloudNode {}
```

4.1.2.4 Les caractéristiques pour un nœud de plate-forme

En partant du principe qu'une plate-forme peut se résumer à un conteneur d'application, cette définition offre plusieurs types de plate-forme en relation avec les différents types de virtualisation que nous avons mis en évidence précédemment. Ainsi, une plate-forme peut correspondre à un serveur d'applications comme peut l'être un serveur Apache Tomcat et qui permet d'héberger des applications dans un même processus système. Une plate-forme peut aussi correspondre à un espace utilisateur virtualisé dans lequel est emprisonné un ensemble de processus utilisateur. Cet espace utilisateur peut éventuellement héberger son propre noyau plutôt que d'utiliser celui fournit par le système hôte. Une plate-forme peut aussi être une machine virtuelle dans laquelle s'exécute un système d'exploitation complet sur un hyperviseur de type 2. Cette machine virtuelle ne possède donc aucune caractéristique spécifique à la solution de virtualisation sous-jacente contrairement à une machine virtuelle sur un hyperviseur de type 1 qui là encore peut correspondre à un type de plate-forme.

Contrairement aux types d'infrastructure, il n'existe pas de propriété obligatoire pour la définition d'une plate-forme hormis ceux déjà cités comme étant communs entre les types de nœuds d'infrastructure et de plate-forme (voir section 4.1.2.2). C'est pourquoi le type *PaaSNode* ne fait qu'hériter des propriétés du type *CloudNode* (voir listing 4.12).

Listing 4.6 – Type de nœud de plate-forme

```
@NodeFragment
public abstract class PaaSNode extends JavaSENode implements CloudNode {
}
```

4.1.2.5 Extension possible pour les types de nœuds Cloud

Extension des types pour l'infrastructure Un nœud d'infrastructure peut spécifier les limitations qui lui correspondent concernant la définition de plate-forme. Par exemple, il peut définir les valeurs par défaut qu'il attribuera lors de la création de nœuds de plate-forme (voir listing 4.7) ainsi que les valeurs maximales (voir listing 4.8) qu'il peut attribuer à chaque nœud de plate-forme facilitant ainsi la sélection du nœud d'infrastructure qui devra héberger tel ou tel nœud de plate-forme.

Listing 4.7 – Type de nœud d'infrastructure avec valeurs par défaut

```
@DictionaryType({
    @DictionaryAttribute(name = "defaultOS"),
    @DictionaryAttribute(name = "defaultARCH", defaultValue = "x86", vals = {"x86", "x86_64"}),
    @DictionaryAttribute(name = "defaultCPU_FREQUENCY"),
    @DictionaryAttribute(name = "defaultCPU_CORE"),
    @DictionaryAttribute(name = "defaultRAM")
})
@NodeFragment
public abstract class IaaSNodeWithDefaultValue extends IaaSNode {}
```

Listing 4.8 – Type de nœud d'infrastructure avec valeur maximum

```
@DictionaryType({
    @DictionaryAttribute(name = "maxCPU_FREQUENCY"),
    @DictionaryAttribute(name = "maxCPU_CORE"),
    @DictionaryAttribute(name = "maxRAM")
})
@NodeFragment
public abstract class IaaSNodeWithMaxValue extends IaaSNode {}
```

Un nœud infrastructure peut aussi spécifier les capacités de virtualisation qu'il peut fournir (voir listing 4.9). En effet, un nœud n'est pas forcément limité à un seul type de virtualisation. Le fait de spécifier les types acceptés sur chacun des nœuds de l'infrastructure permet de sélectionner le nœud d'hébergement en fonction des caractéristiques des nœuds de la plate-forme. Par exemple, une infrastructure fondée sur FreeBSD est capable de faire de la virtualisation d'espace utilisateur avec les Jails[1] mais est aussi

1. http://www.freebsd.org/doc/en/books/handbook/jails.html

capable de supporter de la virtualisation de type 2 à base de KVM [2] ou de Virtualbox [3]. Si un utilisateur demande une plate-forme composée d'un nœud Windows et d'un autre nœud Jails, il est possible d'héberger les deux sur le même nœud d'infrastructure et donc de limiter le besoin de communication réseau entre les deux nœuds de la plate-forme.

Listing 4.9 – Type de nœud d'infrastructure avec type de virtualisation

```
@DictionaryType({
    @DictionaryAttribute(name = "virtualization",
      vals = {"userspace", "Hypervisor1", "Hypervisor2",
        "userspace/Hypervisor 1", "userspace/Hypervisor2"}),
})
@NodeFragment
public abstract class IaaSNodeWithVirtualizationType extends IaaSNode {}
```

Enfin, même si les nœuds d'infrastructure peuvent héberger des applications de gestion et d'adaptation, il n'est pas nécessaire que l'ensemble des nœuds ait cette capacité. C'est pourquoi il peut être intéressant de pouvoir configurer cette propriété en même temps que l'ensemble des propriétés déjà définies (voir listing 4.10). Ainsi, même si tous les nœuds possèdent les primitives d'adaptation adéquates pour héberger des applications, l'implantation de ces primitives ne pourra pas être utilisée si la propriété correspondante définit le nœud comme un simple fournisseur de ressources et non pas un gestionnaire. Même si un nœud ne peut donc pas toujours héberger d'applications, il pourra tout de même héberger des nœuds et doit donc posséder les primitives minimales à cet effet.

Listing 4.10 – Type de nœud d'infrastructure avec rôle

```
@DictionaryType({
    @DictionaryAttribute(name = "role", vals= {"host", "host/container"})
})
@NodeFragment
public abstract class IaaSNodeWithRole extends IaaSNode {}
```

Extension des types pour la plate-forme Dans les solutions existantes, la plate-forme est souvent associée avec la notion de stockage. En effet, l'utilisation de la plate-forme *Google App Engine* impose fortement l'utilisation de leur API de stockage *App Engine Datastore* ou *Google Cloud SQL* et *Google Cloud Storage* dont les APIs sont directement disponibles dans *Google App Engine*. De la même manière, la plate-forme *ElasticBean* tout comme l'infrastructure EC2 d'Amazon imposent fortement l'utilisation de S3 comme système de stockage afin d'éviter des coûts supplémentaires. Pour autant, afin de faciliter la portabilité d'une application entre une plate-forme et une autre, il peut être intéressant de gérer ce stockage au niveau applicatif par l'intermédiaire de

2. http://www.linux-kvm.org/
3. https://www.virtualbox.org/

composants spécifiques ayant une API (voir la notion de port dans Kevoree dans le chapitre 3) permettant d'abstraire la solution choisie. Malgré tout, si une application est migrée d'une plate-forme à une autre, il peut être nécessaire de migrer la solution de stockage et cette opération peut être effectuée par la plate-forme elle-même si elle connaît le système de stockage à mettre en place.

Listing 4.11 – Type de nœud de plate-forme avec gestion du stockage

```
@DictionaryType({
  @DictionaryAttribute(name = "STORAGE", vals = {"SparkleShare", "Dropbox", "Amazon S3"})
})
@NodeFragment
public abstract class PaaSNodeWithStorage extends PaaSNode {}
```

La modélisation de la durée d'exécution peut aussi être une propriété intéressante. En effet, un utilisateur d'infrastructure souhaiterait pouvoir maîtriser ces coûts et pour cela fixer la durée maximale d'utilisation de certaines plates-formes sur telle ou telle infrastructure et envisager des migrations de plates-formes entre infrastructures après un certain temps d'exécution. Si la durée d'exécution est spécifiée, alors le nœud d'hébergement peut être capable d'arrêter la plate-forme sans nécessiter une interaction utilisateur. Un fonctionnement similaire est disponible dans Grid'5000 et est nécessaire pour assurer le partage des ressources existantes à l'ensemble des utilisateurs.

Listing 4.12 – Type de nœud de plate-forme avec durée d'exécution

```
@DictionaryType({
  @DictionaryAttribute(name = "DURATION")
})
@NodeFragment
public abstract class PaaSNodeWithDuration extends PaaSNode {}
```

La figure 4.4 représente le diagramme de classes des différents types présentés dans cette section.

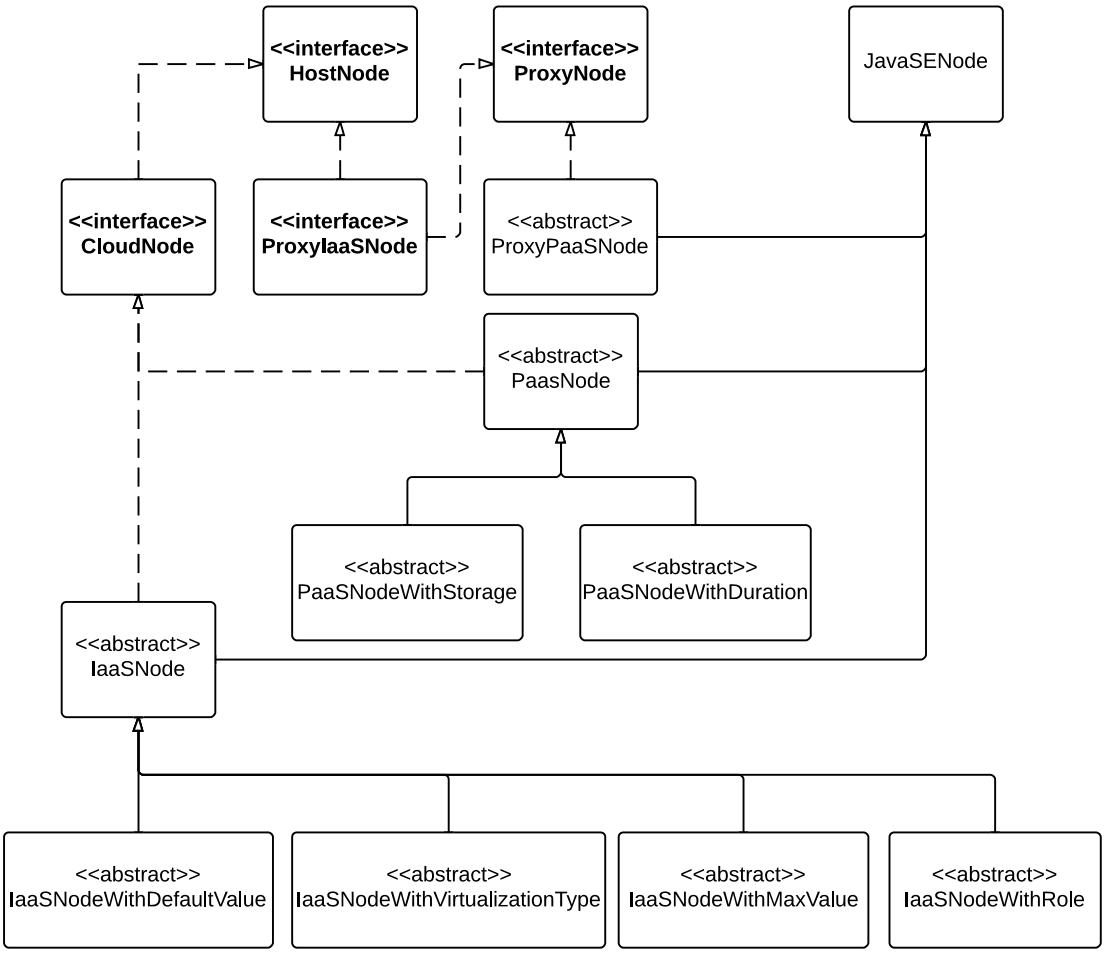


FIGURE 4.4 – Diagramme de classe des types de nœud pour le Cloud

4.2 Définition de systèmes d'adaptation

L'abstraction proposée dans la section précédente permet de représenter l'ensemble des niveaux d'un Cloud. Le framework présenté avec cette abstraction, quant à lui, définit comment les concepteurs de services de Cloud et notamment des concepteurs de services d'infrastructure et de plate-forme peuvent définir de nouvelles implantations d'infrastructure et de plate-forme. Ce sont ensuite ces nouvelles implantations dont pourra tirer parti le fournisseur de services dans sa définition des systèmes d'adaptations que ce soit au niveau infrastructure ou plate-forme. Le framework permet aussi aux fournisseurs de services de s'abstraire de ces implémentations et facilite ainsi la portabilité des systèmes d'adaptation. La notion de Proxy permet de plus d'intégrer des implémentations existantes et notamment des implémentations de Clouds publics ce qui est particulièrement utile dans le cadre de la conception de Clouds hybrides. L'ensemble de ces éléments permet ainsi pour les fournisseurs de services d'envisager de

gérer l'adaptation comme étant une problématique transverse.

Il existe deux moyens de gérer l'adaptation de manière transverse. La première consiste à utiliser un seul système d'adaptation qui se charge de l'ensemble des niveaux. La deuxième consiste à faire collaborer plusieurs systèmes d'adaptation et chacun de ces systèmes d'adaptation gère une petite partie du système globale. Le cas simple de cette deuxième possibilité est le cas où il existe un système d'adaptation par niveau. Notre abstraction sert de support de communication à cette collaboration en permettant à tout un chacun de proposer de nouvelles configurations et de valider les configurations proposées par les autres. Ainsi, ce sont les groupes qui se chargent de faire le lien entre les niveaux afin d'offrir une représentation globale et ce sont ensuite les systèmes d'adaptation qui, en implantant l'interface *ModelListener* (voir section 3.2), sont notifiés des évolutions et prennent part à la validation des nouvelles configurations.

Comme il a été présenté en section 1.2, un système d'adaptation est composé de quatre phases distinctes que sont le Monitoring, l'Analyse, la Planification et l'Exécution. Le modèle architectural de KevoreeKloud fournit un premier support à la phase de monitoring, car il permet de connaître l'évolution architecturale du système et offre, à n'importe quelle implantation de la phase d'analyse, les informations correspondant à l'ensemble des niveaux. La phase d'analyse est quant à elle à la charge des concepteurs de Cloud et c'est cette phase qui implante l'interface *ModelListener* et qui doit collaborer avec les autres pour définir des adaptations multi-niveaux. Concernant la phase de planification et d'exécution, elles sont dépendantes des types de nœuds puisqu'elles dépendent de l'implantation des actions d'adaptation.

Nous avons, dans la section précédente, expliqué, entre autres, comment étendre la phase d'exécution en implantant de nouvelles actions d'adaptation correspondant aux capacités d'hébergement des nœuds d'infrastructure ou de plate-forme. Nous allons maintenant nous intéresser à la définition de la phase de planification. Tout d'abord en section 4.2.1, nous allons présenter l'implantation par défaut qui a été mise en place dans Kevoree à partir de la définition faite dans KevoreeKloud. Puis nous présenterons, en section 4.2.2, F4Plan qui est un framework générique pour la phase de planification permettant de reconfigurer dynamiquement la phase de planification en remplaçant l'algorithme utilisé par un autre selon des contraintes spécifiques.

4.2.1 Kevoree et la planification

La phase de planification peut être divisée en deux sous-étapes que sont la comparaison de modèle afin d'identifier l'ensemble des actions nécessaires à la reconfiguration et l'ordonnancement de ces actions en fonction de leur dépendance afin d'assurer une reconfiguration efficace et cohérente. Par défaut, le type *JavaSeNode* propose une implantation de cette phase de planification.

La comparaison de modèle pour la gestion de la notion d'hébergement La comparaison ne fait pas partie de la contribution de cette thèse et plus d'informations à ce sujet sont fournies dans [67]. Cependant, elle doit être étendue notamment dans le cadre d'une infrastructure. Cette extension de la comparaison de modèle a pour objectif

d'identifier les nœuds qu'il est nécessaire de supprimer ou d'ajouter (voir listing 4.13). Pour cela le parcours de modèle se limite au parcours de la relation qui a été ajoutée dans le modèle.

Listing 4.13 – Surcharge de la comparaison de modèle

```
@NodeType
public class MyNodeType extends MyParentType {
  public AdaptationModel compareModels(ContainerRoot current, ContainerRoot target) {
    // call default comparison
    AdaptationModel adaptationModel = super.compareModels(current, target);
    // get the model element of this node on current model and target model
    ContainerNode currentNode = ...
    ContainerNode targetNode = ...
    for (ContainerNode node : currentNode.getHostForJ) {
      boolean found = false;
      for (ContainerNode node1 : targetNode.getHostForJ) {
        if (node.getName().equals(node1.getName()) {
          found = true;
          beak;
        }
      }
      if (!found) {
        // node must be removed
        adaptationModel.addAdaptations(new RemoveNodeCommand(node));
      }
    }
    for (ContainerNode node : targetNode.getHostForJ) {
      boolean found = false;
      for (ContainerNode node1 : currentNode.getHostForJ) {
        if (node.getName().equals(node1.getName()) {
          found = true;
          beak;
        }
      }
      if (!found) {
        // node must be added
        adaptationModel.addAdaptations(new AddNodeCommand(node));
      }
    }
    return adaptationModel;
  }
}
```

Définition de l'algorithme d'ordonnancement Par défaut, l'algorithme d'ordonnancement définissait un plan séquentiel d'action en ordonnant selon les types d'action. L'ordre des actions suivait cette description : les instances(composants et canaux de communication) devant être arrêtées sont arrêtées et une instance cliente d'une autre doit toujours être arrêtée avant. Puis les liens entre les instances sont supprimés, les instances sont supprimées. Puis les paramètres des instances en cours d'exécution sont mis à jour, les instances à ajouter sont ajoutées, les nouveaux liens entre les instances sont ajoutés. Enfin les instances sont (re-)démarrées et une instance cliente d'une autre doit toujours être démarrée après. Mais cette version était peu efficace, car beaucoup

d'actions peuvent être parallélisées et notamment le démarrage des instances.

Pour cela, l'implantation de l'ordonnancement intégrée dans le nœud de type *JavaSeNode* (voir listing 4.14) utilise l'ensemble de dépendances entre les actions d'adaptation pour construire un graphe orienté acyclique dont les sommets correspondent à des actions et les arêtes à leurs dépendances. Ce graphe est ensuite utilisé pour construire le plan d'adaptation partiellement ordonné. Cette implantation utilise un algorithme glouton qui recherche tout d'abord l'ensemble des sommets n'ayant pas de prédécesseurs ce qui signifie qu'il recherche les actions n'ayant pas de dépendances. Cet ensemble correspond à une première étape (Step) du plan. À partir de ces sommets, l'algorithme recherche l'ensemble de leurs successeurs et conserve parmi ceux-ci l'ensemble des sommets n'ayant pour prédécesseurs que des sommets identifiés à l'étape précédente. Ce nouvel ensemble correspond à une nouvelle étape du plan. L'algorithme continue d'effectuer cette opération tant que l'ensemble des sommets du graphe n'a pas été parcouru. À la fin de l'exécution de cet algorithme, nous obtenons un ensemble d'étapes dépendantes les unes des autres et correspondant au plan d'adaptation nécessaire pour appliquer la nouvelle reconfiguration.

L'extension spécifique au Cloud consiste à intégrer les dépendances autour des actions d'adaptation concernant la création et la suppression des nœuds. Cela se fait en étendant la définition de la méthode *buildGraph*. Le listing 4.5 résume l'ensemble des dépendances entre les actions existantes.

4.2.2 F4Plan

Outre notre implantation de la phase de planification, il existe beaucoup d'autres travaux sur ce sujet notamment dans le domaine de l'intelligence artificielle ou dans le domaine de la robotique [90]. Ces algorithmes ont généralement des spécificités les rendant intéressants dans certaines situations et notamment dans le cas où l'on souhaite contraindre l'adaptation. Par exemple, certains algorithmes (comme celui proposé dans KevoreeKloud) permettent de construire des plans d'actions partiellement ordonnés. C'est le cas de *GraphPlan* [46] ou encore SHOP2 [101]. Mais ces algorithmes peuvent entraîner des coûts importants au moment de l'exécution des actions en termes de ressources utilisées et ainsi avoir un impact sur l'exécution des applications. D'autres algorithmes permettent de calculer le temps nécessaire à l'exécution d'un plan. Ce genre d'algorithmes va ensuite pouvoir chercher un plan pour lequel le temps d'exécution est acceptable. C'est le cas par exemple avec *SGPlan* [52].

Le choix entre ces différentes caractéristiques et donc entre les algorithmes qui les supportent n'est pas quelque chose de forcément prédéfini. Par exemple, on peut souhaiter utiliser un algorithme capable de générer un plan partiellement ordonné quand les ressources disponibles sont assez importantes, mais préférer un algorithme capable de limiter l'utilisation de ces mêmes ressources lors que celles-ci sont déjà grandement utilisées. Pour cela, il faut être capable de choisir dynamiquement l'algorithme de planification à utiliser.

Afin de pouvoir dynamiquement sélectionner l'algorithme de planification à utiliser, nous avons conçu un framework appelé F4Plan [39]. Ce framework est constitué

Listing 4.14 – Algorithme d'ordonnancement

```
@NodeType
public class MyNodeType extends MyParentType {
  def schedule(adaptationModel : AdaptationModel) : AdaptationModel = {
    var commands : List[AdaptationPrimitive] = adaptationModel.getAdaptations
    var stepList = Array[List[AdaptationPrimitive]]()
    if (commands.size > 1) {
      // build graph of dependencies between commands
      val graph = buildGraph(commands)
      val index = new DirectedNeighborIndex(graph)
      var number = 0
      var alreadySeenList = List[AdaptationPrimitive]()
      var previousStep = graph.vertexSet(). filter (v → index.predecessorsOf(v). size () == 0).toList
      var step = KevoreeAdaptationFactory.eINSTANCE.createParallelStep
      var currentStep = step
      adaptionModel.setOrderedPrimitiveSet(Some(currentStep))
      // generate steps while all the commands are not set in a step
      while (number < graph.vertexSet().size()) {
        var newStep = List[AdaptationPrimitive]()
        previousStep.foreach {
          p →
            // get all commands for which all predecessors have been set on previous steps
            newStep = newStep ++ index.successorListOf(p).filter(v →index.predecessorsOf(v). forall (pred
                → alreadySeenList.contains(pred))). toList
        }
        // create a step with all the commands found
        currentStep.addAllAdaptations(newStep)
        step = KevoreeAdaptationFactory.eINSTANCE.createParallelStep
        currentStep.setNextStep(Some(step))
        currentStep = step
        alreadySeenList = alreadySeenList ++ newStep
        number = number + newStep.size
        previousStep = newStep
      }
    } else {
      step.addAllAdaptations(commands)
    }
    adaptationModel
  }
}
```

de quatre types d'entité. Le premier appelé *gestionnaire de planification* ou Planning Manager en anglais reçoit les configurations qui doivent être appliquées. Ce gestionnaire utilise un *guide de planification* ou Planning Guide en anglais qui se charge de la sélection automatique de l'*algorithme de planification*. Une fois l'algorithme choisi, le gestionnaire de planification doit traduire la stratégie fournie par le moteur de décision afin qu'elle soit comprise par l'algorithme de planification. En effet, en plus des caractéristiques concernant ces capacités de planification, un algorithme est souvent spécifique en termes de langage pour la définition du problème de planification à régler ainsi que pour le langage permettant de représenter le résultat. De la même façon, un moteur de décision peut utiliser un langage particulier n'ayant la plupart du temps aucun rapport avec les langages de planification. C'est pourquoi F4Plan intègre un mécanisme de conversion automatique permettant de passer d'un langage de décision à un langage

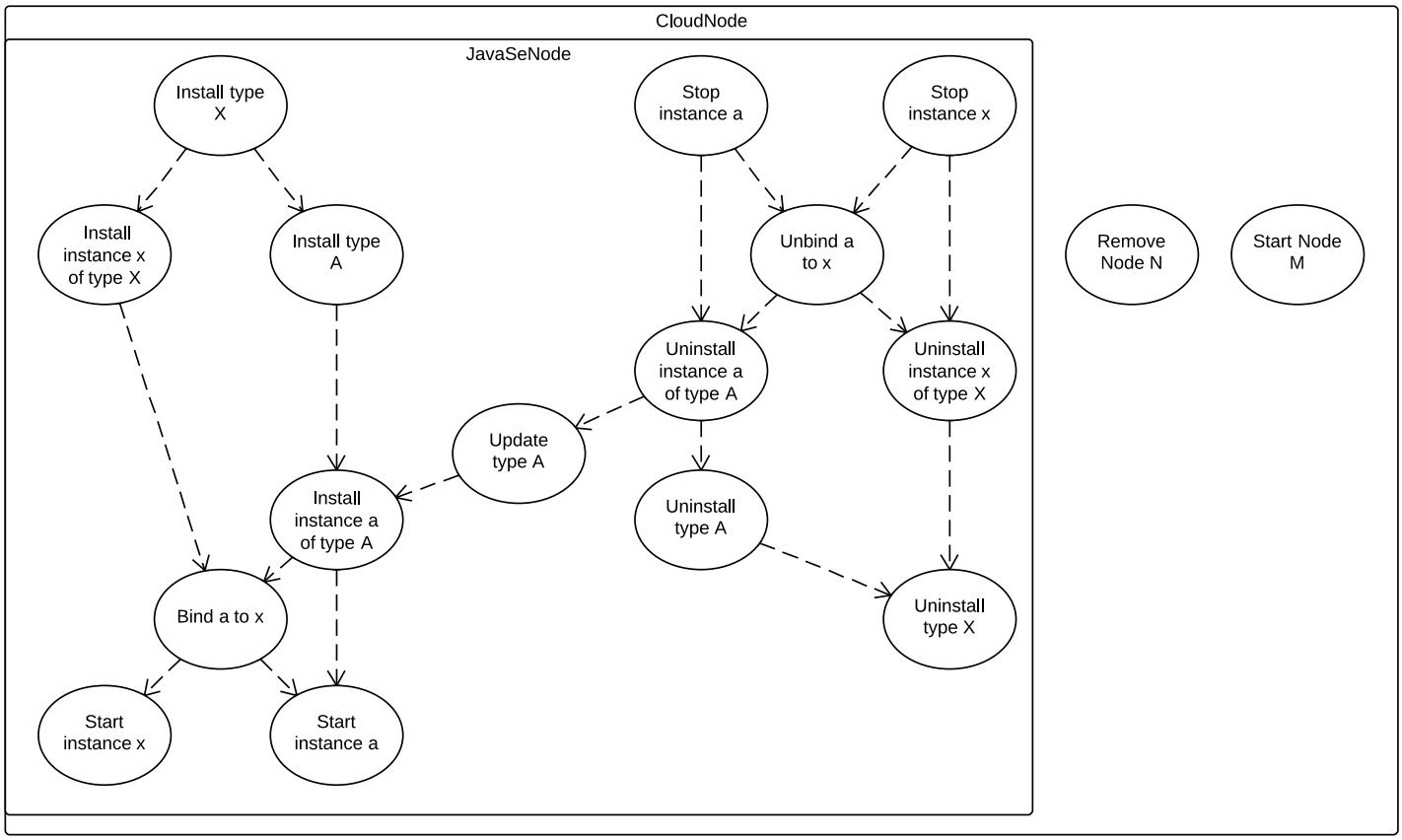


FIGURE 4.5 – Graphe de dépendance des actions

de planification (voir figure 4.6). Pour effectuer la traduction, le *gestionnaire de planification* qui va chercher une combinaison de *traducteurs* permettant en passant par un langage intermédiaire de traduire la stratégie en problème de planification. F4Plan utilise un langage intermédiaire afin de faciliter la réutilisation des traducteurs spécifiques à chaque algorithme que ce soit pour les algorithmes de planification ou de décision. Le langage intermédiaire qui a été choisi est PDDL [71] qui est une tentative de standardisation des langages de planification. En effet, même s'il existe de nombreux langages de planification, ceux-ci proposent en général de définir les mêmes propriétés dans un problème de planification. PDDL propose donc un langage à base d'extensions permettant de définir son propre langage par association d'extensions. Ces extensions représentent la majorité des spécificités des algorithmes de planification actuels. Avec ce langage intermédiaire, un traducteur générique capable de traduire la version la plus complète du langage en un sous-ensemble correspondant à celui de l'algorithme de planification a été défini. Une fois la stratégie traduite, c'est l'algorithme de planification qui cherche une solution et produit un plan. Ce plan est ensuite converti par le gestionnaire de traduction en une représentation de haut niveau utilisable par la phase d'exécution. Dans son intégration à Kevoree ou KevoreeKloud, ce framework ne nécessite plus d'avoir des

traducteurs pour les langages utilisés par les moteurs de décision. En effet, dans Kevoree et donc dans KevoreeKloud le langage utilisé est toujours le même et correspond au modèle architectural. Cependant, F4Plan conserve la traduction du langage de décision vers le langage intermédiaire afin d'anticiper de possible évolution de Kevoree et permettre aussi son utilisation hors Kevoree.

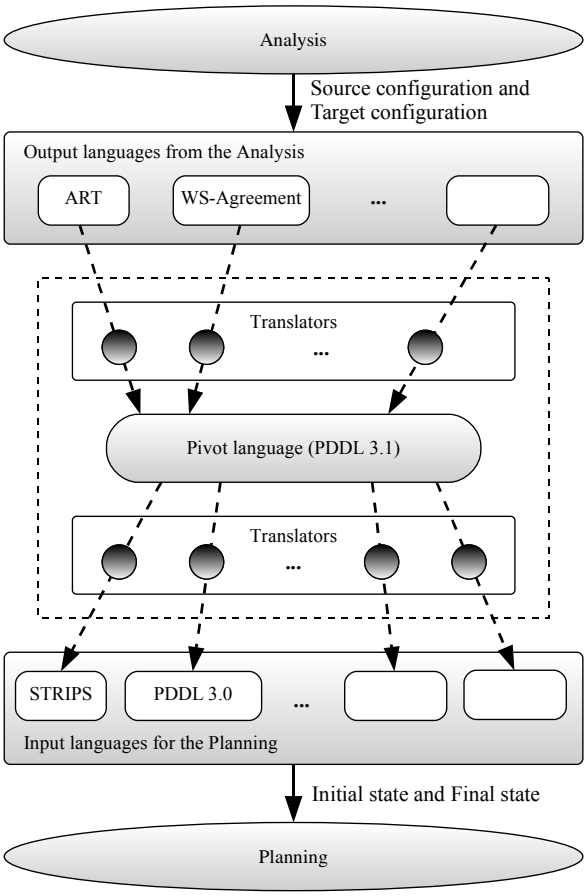


FIGURE 4.6 – F4Plan : Gestionnaire de traduction entre langages de décision et langages de planification

4.3 Vue globale ou vue partielle du système

Notre abstraction permet de modéliser, avec une même abstraction, une application, une plate-forme ou une infrastructure. Mais le fait de pouvoir faire cohabiter les architectures des applications, des plates-formes et des infrastructures ne signifie pas qu'il faut à tout prix tout modéliser.

En effet, dans un monde idéal, la vision globale du système permettrait de pouvoir gérer l'adaptation de manière globale en étant capable de tenir compte de chacun des impacts qu'une adaptation peut avoir quel que soit l'élément impacté. Cependant, dans un système composé de plusieurs milliers et peut-être dans le futur de plusieurs millions voir milliards d'entités, la gestion de ce modèle global peut entraîner de nombreuses difficultés que ce soit en termes de synchronisation entre les nœuds (e.g. : comment assurer en un temps acceptable que plusieurs milliards de nœuds soient synchronisés ?), mais aussi en termes de performance (e.g : combien de temps faut-il pour trouver la configuration d'un nœud parmi des milliards ?). De plus, toutes les informations sur le système ne sont pas forcément utiles pour pouvoir gérer l'adaptation de manière transverse.

Il pourrait donc être intéressant de limiter les informations présentes dans notre représentation globale aux informations utiles. De manière plus pragmatique, il peut aussi être intéressant de pouvoir limiter les informations présentes dans la représentation globale aux informations que les concepteurs ou fournisseurs de services de Cloud acceptent de voir partager. C'est le cas des Clouds publics qui, par exemple, ne partagent pas le nombre de ressources qu'ils ont à disposition.

Les groupes permettent de partager la configuration courante du système entre les différents nœuds d'exécution que ce soit des nœuds d'infrastructure ou de plate-forme. L'utilisation de l'interface de *ModelListener* permet ensuite aux composants, canaux de communication ou nœuds (au sens Kevoree) d'avoir la capacité de connaître les reconfigurations, mais surtout de pouvoir réagir à celles-ci.

C'est pourquoi, en plus de se charger de la synchronisation des reconfigurations ainsi que de la dissémination de celles-ci, les groupes peuvent servir au partitionnement du modèle. Le partitionnement de modèle permet de limiter les informations accessibles par les entités s'exécutant dans le système.

Ce partitionnement peut être envisagé de différentes manières. Tout d'abord, il peut correspondre aux différents niveaux. Ainsi, une infrastructure n'a pas la connaissance de la configuration des plates-formes qu'elle héberge et de même une plate-forme ne connait pas l'infrastructure qui l'héberge (voir figure 4.7). Bien entendu ce partitionnement ne permettrait en aucun cas de gérer l'adaptation de manière transverse. Ce partitionnement peut avoir un intérêt dans le cas d'infrastructure et/ou de plate-forme sur lesquelles il n'est pas possible d'influer. C'est notamment le cas pour les Clouds publics qui ne sont pas reconfigurables par les utilisateurs. Puisqu'il n'est pas possible d'envisager des reconfigurations de leur part, il n'est du fait pas nécessaire de connaître les détails les concernant.

Il est aussi envisageable de faire du partitionnement en fonction de l'hébergement. Par exemple, une application peut avoir des informations sur la plate-forme qui l'hé-

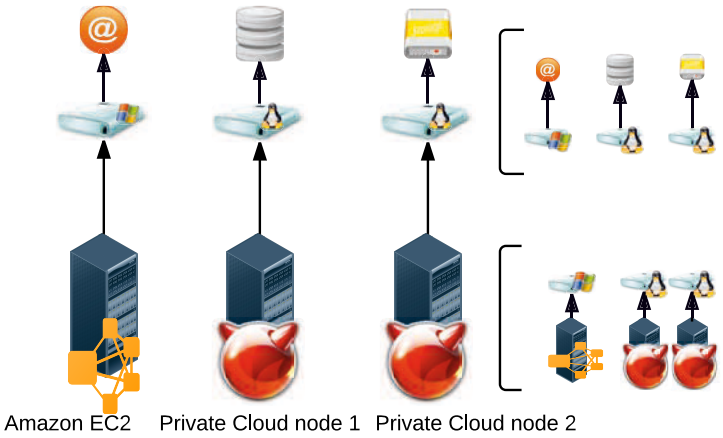


FIGURE 4.7 – Partitionnement de modèle en fonction des niveaux

berge, mais n'aura pas d'information sur les autres applications qui cohabitent avec elle sur la même plate-forme. De la même façon, une plate-forme va connaître la configuration de l'infrastructure qui l'héberge, mais n'aura pas les informations concernant les autres plates-formes qui cohabitent sur les mêmes ressources. Enfin une infrastructure aura les informations concernant l'ensemble des plates-formes qu'elle héberge mais ne connaitra rien des infrastructures qui cohabitent avec elle (voir figure 4.8). Ce mode de partitionnement permet d'envisager l'adaptation de manière transverse en tenant compte des différents niveaux. Cependant, là encore cette adaptation reste limitée. En effet, le manque de connaissance de la configuration complète d'un niveau ne permet pas d'envisager de la collaboration entre les implantations d'un même niveau. Par exemple, une infrastructure ne pourra pas déléguer, à une autre infrastructure, la gestion de plusieurs machines virtuelles si elle n'a plus la capacité de les héberger. Or cette adaptation est très importante dans le cadre de Clouds hybrides ou plus généralement dans le cadre du Sky Computing dans lesquels une infrastructure peut déléguer ponctuellement la création de machines virtuelles sur une autre infrastructure.

Enfin, il est aussi possible non plus de partitionner, mais de filtrer le modèle. Contrairement au partitionnement qui permet de ne conserver qu'un sous-ensemble d'éléments jugés intéressant, le filtrage permet d'abstraire certaines informations par d'autres. Ainsi, chaque élément peut avoir une connaissance globale de la configuration du système, mais il est possible par exemple d'abstraire le typage des composants permettant à l'infrastructure de connaître les interconnexions entre des composants distribués et donc d'inférer des communications réseau entre plusieurs plates-formes sans pour autant connaître précisément la configuration des composants (voir figure 4.9). Il est aussi possible de représenter l'un des niveaux à plus gros grains. Par exemple, en masquant l'ensemble des ressources offertes par une infrastructure comme le fait un nœud proxy (voir figure 4.10)

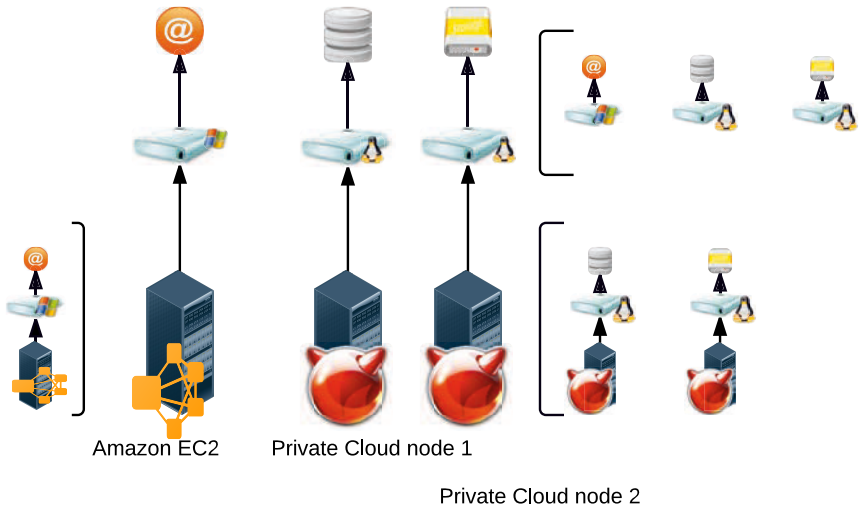


FIGURE 4.8 – Partitionnement de modèle en fonction des interactions

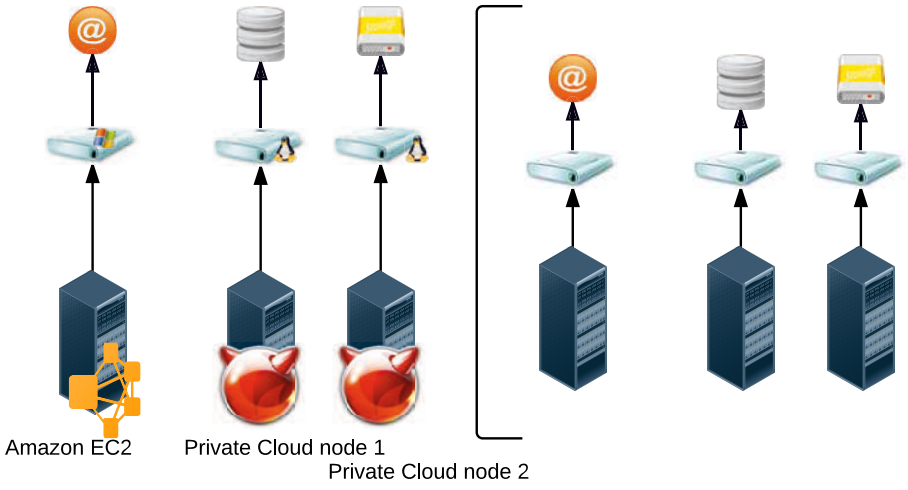


FIGURE 4.9 – Filtrage de modèle avec de l'abstraction de type

Vue du système au moment de la conception Même si notre abstraction a pour objectif de modéliser l'ensemble des niveaux d'un Cloud, elle n'interdit pas pour autant la représentation d'un seul niveau. C'est d'ailleurs cette utilisation qui permet aux concepteurs de services de Cloud, que ce soit au niveau de l'infrastructure, de la plate-forme ou des applications, de définir la configuration de leur offre. En effet, un concepteur d'application n'a pas besoin de connaître l'architecture de la plate-forme d'hébergement et encore moins l'architecture de l'infrastructure ni même le déploiement de la plate-

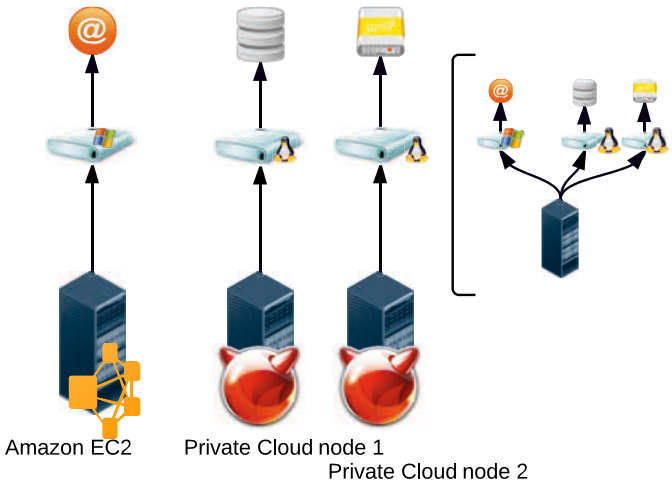


FIGURE 4.10 – Filtrage de modèle à gros grain

forme sur cette infrastructure. Les seules informations qui intéressent ce développeur sont ces composants, leurs interconnexions et leurs moyens de communication ainsi que le support d'exécution nécessaire à l'application. Il en va de même pour la définition d'un type de nœud lors de la définition d'une nouvelle implémentation de plate-forme ou d'infrastructure pour lesquels le concepteur souhaite simplement se focaliser sur la définition de l'adaptation ainsi que sur les contraintes de son support d'exécution.

Kevoree impose aussi de définir l'architecture d'une application comme un ensemble de composants et de canaux de communication hébergés par un ensemble de nœuds. De la même façon, l'architecture d'une plate-forme correspond à la définition d'un ensemble de nœuds de type plate-forme hébergés sur un ensemble de nœuds de type infrastructure.

Cette spécification des nœuds d'hébergement peut se faire à différents grains. Tout d'abord, un concepteur peut vouloir laisser à la solution d'hébergement le soin de mettre en place de manière intelligente les différents éléments de son système. Pour cela, il peut simplement définir ses propres nœuds sans spécifier l'hébergement de ceux-ci (voir figure 4.11). C'est ensuite l'infrastructure qui se chargera de déployer chaque nœud sur l'ensemble des ressources qui sont à sa disposition.

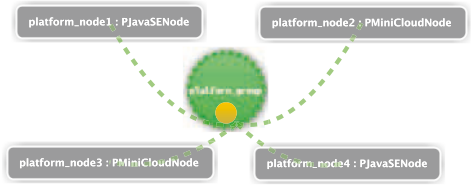


FIGURE 4.11 – Définition de l'architecture d'une plate-forme sans contraintes sur l'hébergement

Dans le cas de la définition d'une application, il est nécessaire de modéliser les nœuds d'hébergement. Si le concepteur ne souhaite pas définir l'architecture d'hébergement, il peut simplement utiliser un seul nœud (voir figure 4.12). Là encore, ce sera la plate-forme qui se chargera de déployer chaque composant sur les ressources disponibles.

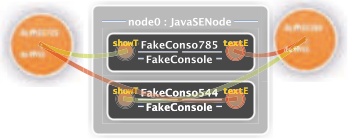


FIGURE 4.12 – Définition de l'architecture d'une application sans contraintes sur l'hébergement

Le concepteur peut aussi vouloir spécifier les différents types de nœuds dont il a besoin pour son système sans pour autant définir le nombre exact de nœuds qu'il souhaite (voir figure 4.13). Là encore, la solution d'hébergement sera chargée de répartir les différents éléments sur les bonnes ressources tout en tenant compte tout de même des caractéristiques des nœuds d'hébergement souhaités par le concepteur.

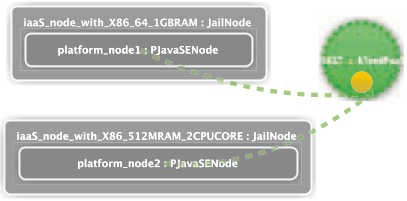


FIGURE 4.13 – Définition de l'architecture d'une plate-forme avec des contraintes sur l'hébergement

4.4 Synthèse

Ce chapitre a présenté la contribution principale de cette thèse. Cette contribution concerne KevoreeKloud qui est une extension de Kevoree afin d'intégrer les notions d'hébergement et de multi-niveaux nécessaires pour pouvoir représenter un Cloud dans son ensemble. Cette extension est accompagnée d'un framework de type facilitant la conception de nouvelles implantations d'infrastructure ainsi que de plate-forme. Utilisant les capacités de Kevoree, ce framework permet aussi d'intégrer de nouvelles capacités d'adaptation et permet d'étendre l'algorithme chargé de mettre en place les reconfigurations afin de tenir compte de ces nouvelles capacités. Pour cela, un algorithme basé sur le parcours de graphes permet d'effectuer de la planification des actions d'adaptation afin que la reconfiguration puisse s'effectuer de manière efficace sur chacun des nœuds. Nous fournissons aussi un framework appelé F4Plan qui permet d'intégrer des algorithmes de planification existants et qui permet aussi de sélectionner dynamiquement l'algorithme à utiliser afin de répondre aux contraintes imposées

sur le processus d'adaptation. Enfin, par l'intermédiaire des groupes de dissémination des reconfigurations, nous proposons différentes solutions pour la gestion de la granularité de l'abstraction permettant aux différentes implémentations de même niveau ou de niveaux différents de gérer les informations disponibles. Cette contribution a pu être intégrée à Kevoree grâce à ses capacités d'extensibilité et sa gestion de la dissémination du modèle dans les systèmes distribués. La définition de ces éléments a été effectué en collaboration avec François Fouquet dans le cadre de sa thèse [67] et a été en partie motivée par la problématique d'administration du Cloud.

Chapitre 5

Validation

L'objectif de cette thèse est de fournir une abstraction permettant de définir aussi bien une application, une plate-forme ou une infrastructure de Cloud et de permettre de gérer l'ensemble de ces solutions de manière homogène. L'utilisation d'une abstraction commune à l'ensemble de ces niveaux permet ainsi de gérer l'adaptation non plus comme une problématique présente sur chacune des solutions, mais bien comme un problème global au système.

Nous allons dans ce chapitre valider le fait que notre contribution répond bien aux objectifs fixés. Pour cela, nous allons montrer que notre solution est suffisamment générique et extensible pour pouvoir concevoir facilement de nouvelles implantations d'infrastructure ou de plate-forme de Cloud. Nous allons montrer ensuite que l'utilisation de KevoreeKloud est capable de supporter la montée en charge, c'est-à-dire qu'elle est capable de gérer des Clouds de grandes tailles. Enfin, nous allons montrer que notre abstraction permet bien de gérer l'adaptation comme une problématique transverse et qu'elle offre la possibilité de concevoir des adaptations multi-niveaux, c'est-à-dire des adaptations ayant des impacts sur plusieurs niveaux.

Afin de démontrer ces différents points, nous allons présenter trois expérimentations. La première, en section 5.1, présente l'implantation d'une infrastructure de Cloud hybride à base de virtualisation d'espaces utilisateurs pour des systèmes Unix et Linux et de virtualisation matérielles pour les systèmes Microsoft. Cette expérimentation présente aussi l'implantation d'un gestionnaire d'infrastructure dont l'objectif est d'utiliser les systèmes de virtualisation proposés pour héberger au mieux les plates-formes et de manière à équilibrer la charge sur l'ensemble des ressources concrètes offertes par l'infrastructure. Dans cette expérimentation, nous évaluons la quantité de lignes de code nécessaire pour définir un nouveau type de nœuds d'infrastructure ainsi qu'un gestionnaire d'infrastructure afin de montrer la facilité de concevoir des nouveaux types de nœuds et des nouveaux types de gestionnaire.

La seconde expérimentation, en section 5.2, présente l'implantation d'une plate-forme capable de répartir les composants d'une application sur une infrastructure en prenant en compte les contraintes concernant les ressources nécessaires au bon fonctionnement des composants. Dans cette expérimentation, nous évaluons l'impact de

KevoreeKloud au niveau du temps d'exécution des reconfigurations ainsi que de l'empreinte mémoire pour le stockage du modèle architectural représentant l'ensemble du Cloud (Infrastructure, Plate-forme et Applications).

Enfin la troisième expérimentation, en section 5.3, présente l'implantation d'une plate-forme pour la gestion de serveur web distribué. Dans cette expérimentation, nous évaluerons comment l'usage de KevoreeKloud nous permet de partager une vision globale du système permettant à l'infrastructure et la plate-forme de tirer parti des informations de l'ensemble du système. Nous montrerons aussi comment il est possible de construire des systèmes d'adaptations coordonnées en utilisant le modèle architectural comme support de communication.

Certaines expérimentations ont été menées sur Grid5000, cependant la reproductibilité des expérimentations n'étant pas assurée principalement concernant le temps d'exécution et , nous avons mené l'ensemble des expérimentations présentées dans ce chapitre sur 10 machines identiques à base de processeur Intel R, Atom™ D425 1.8GHz et avec 8 Go de mémoire vive (voir figure 5.1).

FIGURE 5.1 – Machines physiques ayant servi de Cloud

5.1 Évaluation 1 : Est-ce extensible et générique ?

L'objectif de cette section est de montrer que l'utilisation de notre abstraction permet de faciliter la définition de nouveaux types de nœud ainsi que de nouveaux gestionnaires d'infrastructure et de plate-forme.

5.1.1 Protocole expérimental

Dans cette évaluation, nous allons présenter l'implantation de plusieurs types de nœuds pour le niveau infrastructure ainsi qu'un gestionnaire d'infrastructure capable de distribuer les nœuds de plate-forme sur les nœuds d'infrastructure. Afin d'évaluer l'extensibilité de notre framework, nous avons compté le nombre de lignes de code nécessaires à l'implantation d'un nouveau type de nœud (sans compter le code généré) et comparons ces résultats aux nombres de lignes de code fournies par Kevoree ainsi que

par KevoreeKloud et qui permettent au nouveau type de nœuds de réutiliser directement une implantation des différentes fonctionnalités qui doivent être implantées. Les chiffres concernant le nombre de lignes de code ont été obtenus par l'intermédiaire de l'outil [1]

5.1.2 Implémentation du cas d'étude

Il existe plusieurs types de Cloud (voir section 1.1), nous allons ici nous intéresser aux Clouds hybrides qui permettent aux entreprises de combiner leurs propres ressources avec celles fournies par un Cloud public. L'usage du Cloud public permet d'augmenter ponctuellement le nombre de ressources utilisées lorsque les ressources locales ne sont plus suffisantes.

Dans cette section, nous allons présenter l'implantation de quelques types de nœuds ainsi que la définition du gestionnaire d'infrastructure qui permet de gérer un Cloud hybride dans lequel l'ensemble des machines virtuelles Windows sera hébergé sur une solution de virtualisation matérielle, ici sur Amazon EC2 et les machines virtuelles Linux et Unix seront hébergées sur une solution de virtualisation d'espaces utilisateurs, ici en utilisant les Jails de FreeBSD.

5.1.2.1 Infrastructure d'espace utilisateur

Nous avons choisi d'implémenter une solution d'infrastructure utilisant les Jails [2] de FreeBSD [3]. Les Jails ont été intégrées comme solution avancée de *chroot* [4]. Elles permettent d'exécuter les services sensibles dans des environnements distincts de celui du système d'exploitation. Contrairement au *chroot* qui ne peut isoler que le système de fichiers en définissant une racine spécifique pour les processus s'exécutant dans le *chroot*, les *Jails* permettent un plus haut niveau d'isolation en offrant la capacité d'isoler non seulement le système de fichiers, mais aussi les interfaces réseaux. Les *Jails* sont aussi capables de limiter la consommation des ressources que ce soit en terme d'espace disque, de quantité de mémoire vive et de temps CPU ou de nombre de cœur de CPU. Tout comme Amazon EC2 qui fournit des *AMIs* correspondant à des configurations de base pour des machines virtuelles, l'environnement Jails permet de définir ce que l'on appelle des *flavors* qui permettent de définir des patrons (*templates*) de *Jails*. Ces patrons permettent de définir le système exécuté dans les *Jails*, c'est-à-dire quels sont les services au démarrage, quels sont les applications disponibles, quels sont les utilisateurs enregistrés.

Le type *JailNode* (voir listing 5.1) permet donc de définir une *Jail* pour chaque nœud de plate-forme hébergé par l'instance du nœud d'infrastructure. En plus des primitives définies dans *IaaSNode*, un *JailNode* est aussi capable de sauvegarder la configuration d'une *Jail* et permet aussi de recharger une configuration préalablement sauvegardée.

Dans le cadre de cette expérimentation, j'ai aussi défini un type de nœud de plate-forme pouvant définir des caractéristiques aux Jails (voir listing 5.2). Ces caractéris-

1. http://cloc.sourceforge.net/
2. http://www.freebsd.org/doc/en/books/handbook/jails.html
3. http://www.freebsd.org/
4. http://en.wikipedia.org/wiki/Chroot

Listing 5.1 – Définition du JailNode

```
@PrimitiveCommands(
  values = {"SAVE_NODE", "RELOAD_NODE"})
@DictionaryType({
    @DictionaryAttribute(name = "default_mode"),
    @DictionaryAttribute(name = "default_flavor")
})
@NodeType
public class JailNode extends IaaSNode {
```

tiques sont prises en compte par le nœud d'infrastructure au moment de la création des *Jails*. Ainsi, il est possible de spécifier une *flavor* particulière ainsi qu'une archive qui correspond à une sauvegarde d'une *Jail* précédente.

Listing 5.2 – Définition du PJailNode

```
@DictionaryType({
    @DictionaryAttribute(name = "flavor", optional = true),
    @DictionaryAttribute(name = "archive", optional = true),
})
@NodeType
public class PJailNode extends MiniCloudNode implements PaaSNode {
```

Pour ces deux types de nœuds, les propriétés héritées au travers de *IaaSNode* et *PaaSNode* permettent aussi de définir les caractéristiques concernant la quantité de mémoire vive ainsi que le processeur.

En plus du type de ces types de nœud, nous avons aussi défini un composant de gestion d'infrastructure (voir listing 5.3). Son rôle est de définir le lien d'hébergement entre les nœuds de plate-forme ajoutés dans le modèle par le gestionnaire de plate-forme et les nœuds d'infrastructure présents dans le système. Nous avons défini une implantation basique dans laquelle les nœuds de plate-forme sont alloués sur les nœuds d'infrastructure en fonction du nombre de nœuds déjà hébergés. De cette manière, chaque nœud d'infrastructure héberge le même nombre de nœuds de plate-forme même si ceux-ci ne possèdent pas les mêmes limitations.

Bien entendu cette implantation est très naïve, mais cette thèse n'avait pas pour objectif de travailler sur des algorithmes de placement intelligent. Afin d'effectuer de l'hébergement intelligent, il serait nécessaire d'utiliser par exemple des algorithmes autour du green computing avec entre autres le framework *Snooze*[5] [65, 64].

5.1.2.2 Proxy pour infrastructure EC2

L'utilisation des *Jails* de FreeBSD permet d'avoir de la virtualisation d'espace utilisateur. De plus, FreeBSD propose une émulation de système linux permettant d'exécuter

5. http://snooze.inria.fr/

Listing 5.3 – Définition du gestionnaire de l'infrastructure

```
@ComponentType
class IaaSManager extends AbstractComponentType implements ModelListener {
  def modelUpdated() {
    val iaasModel = getModelService.getLastModel
    val kengine = getKevScriptEngineFactory.createKevScriptEngine
    // count current child for each parent nodes
    val parents = KloudModelHelper.countChilds(iaasModel)
    var potentialParents = List[String]()
    var doSomething = false
    var usedIps = Array[String]()
    // filter nodes that are not IaaSNode and are not hosted by an IaaSNode
    iaasModel.getNodes.filter(n → KloudModelHelper.isPaaSNode(iaasModel, n.getName)
      && iaasModel.getNodes.forall(parent →!parent.getHosts.contains(n))).foreach {
      // select a host for each user node
      node → {
        if (potentialParents.isEmpty) {
          // get a list of nodes that have less child nodes than the others
          potentialParents = lookAtPotentialParents(parents)
        }
        val parentName = selectParent(potentialParents)
        kengine.addVariable("nodeName", node.getName)
        kengine.addVariable("parentName", parentName)
        kengine append "addChild {nodeName}@{parentName}"
        potentialParents = potentialParents.filterNot(p → p == parentName)
        doSomething = true
      }
    }
    if (doSomething) {
      getModelService.unregisterModelListener(this)
      try {
        // apply the KevScript on the current model to build a new one and send it to the Kevoree Core
        updateIaaSConfiguration(kengine)
      } catch {
        case ignored : SubmissionException→
      } finally {
        getModelService.registerModelListener(this)
      }
    }
  }
  def selectParent(potentialParents : List[String]) : String = {
    // randomly select one of the potential parent nodes
    val index = (java.lang.Math.random() * potentialParents.size).asInstanceOf[Int]
    val parentName = potentialParents(index)
  }
}
```

des applications Linux. Cependant, il n'est pas possible d'héberger des nœuds dans lesquels s'exécuteraient des applications Windows. C'est pourquoi nous avons choisi de réutiliser une solution existante permettant de créer des nœuds de plate-forme avec un système d'exploitation Windows. Pour cela, nous avons choisi de définir un proxy utilisant l'API EC2 permettant de se connecter au Cloud d'Amazon mais aussi à n'importe quel Cloud compatible avec cette API (par exemple CloudStack, OpenNebula, Nimbus).

Avec ce proxy (voir listing 5.4), nous offrons la possibilité de créer de nouvelles instances, d'en arrêter ou d'en redémarrer ou encore d'en supprimer. Ce type de nœud

possède en plus des caractéristiques héritées du *ProxyIaaSNode*, des propriétés lui permettant de définir les caractéristiques par défaut des instances de machines virtuelles qu'il peut déployer.

Listing 5.4 – Définition du EC2Node

```
@DictionaryType({
  @DictionaryAttribute(name = "DEFAULT_INSTANCE_IMAGE")
  @DictionaryAttribute(name = "DEFAULT_INSTANCE_TYPE", optional = true)
})
@NodeType
public class EC2Node extends ProxyIaaSNode {
```

Nous avons aussi étendu le gestionnaire de l'infrastructure pour qu'il n'envoie les nœuds de plateforme sur le Cloud EC2 que si ces nœuds de plate-forme nécessitent un Windows en tant que système d'exploitation (voir listing 5.5).

Listing 5.5 – Algorithme pour le déploiement des système Windows

```
node →
  if (node.getDictionary().get("OS").toLowerCase().contains("microsoft") ||
      node.getDictionary().get("OS").toLowerCase().contains("windows")) {
    val iaaSNodeWithVirtualizationTypeNodes = getModelService.getLastModel.getNodes.filter(n →
        KloudModelHelper.isASubType(n.getTypeDefinition, "ProxyIaaSNode"))
    if (iaasNodeWithVirtualizationTypeNodes.size > 0) {
      val potentialParents = List[String]()
      iaaSNodeWithVirtualizationTypeNodes.foreach{n →potentialParents = potentialParents ++
          List[String](n.getName)}
      val parentName = selectParent(potentialParents)
      kengine.addVariable ("nodeName", node.getName)
      kengine.addVariable ("parentNodeName", parentName)
      kengine.append("addChild {nodeName}@{parentNodeName}")
    } else {
      logger.error("Unable to host {} because there is no IaaS able to host such kind of node.",
          node.getName)
    }
  }
```

5.1.3 Évaluation

Le tableau de la figure 5.2 récapitule les chiffres que nous avons obtenus sur les différentes implantations de type (*JailNode*, *EC2Node*, *IaaSManager*) et sur les APIs et implantations réutilisées (KevoreeKloud API, JavaSeNode, Kevoree Core). Nous constatons que la définition du type *JailNode* correspond à 529 lignes de code qui contient l'implantation nécessaire pour créer et supprimer une *Jail* mais aussi du code capable de définir et modifier les contraintes posées sur une *Jail* (CPU, RAM). Concernant le proxy vers le Cloud d'Amazon, nous avons 322 lignes de code correspondant à l'ajout et la suppression de machines virtuelles ainsi que l'observation des machines virtuelles déjà créées. Le gestionnaire d'infrastructure (*IaaSManager*) nécessite quant à lui 316

lignes de code pour effectuer le placement des nœuds de plate-forme. L'API que nous avons défini dans KevoreeKloud contient 967 lignes de codes permettant de définir un ensemble d'abstractions de type ainsi qu'une implémentation spécifique de la phase de planification incluant les primitives "ADD_NODE" et "REMOVE_NODE". Le type *JavaSENode* est quant à lui composé de 2547 lignes de code représentant l'algorithme de planification de base ainsi que l'implantation de l'ensemble des primitives de reconfiguration concernant les composants, les canaux de communication et les groupes. Enfin le Kevoree Core qui offre la gestion du modèle à l'exécution et la gestion du processus d'adaptation ainsi que le langage de script permettant de définir des reconfigurations est composé de 21768 lignes de code.

Le tableau de la figure 5.3 montre les ratios de code par rapport aux codes réutilisés (*KevoreeKloud API*, *JavaSeNode*, *Kevoree Core*). Il montre aussi les ratios de code par rapport au framework de KevoreeKloud lui-même.

Projet	**Nombre de lignes de code**
JailNode	529
EC2Node	322
IaaSManager	316
KevoreeKloud API	967
JavaSENode	2547
Kevoree Core	21768

FIGURE 5.2 – Nombre de lignes de code non générées selon le projet

Projet	**Ratio par rapport aux codes réutilisés (%)**	**Ratio par rapport au framework de cette thèse(%)**
JailNode	2,17	54,70
EC2Node	1,32	33,29
IaaSManager	1,29	32,67
KevoreeKloud API	3,97	100

FIGURE 5.3 – Ratio de code selon le projet

Nous pouvons constater que grâce à la réutilisation et l'héritage des types, la définition de nouveaux types nécessite peu de code par rapport à l'implantation de base avec *JavaSENode* ainsi que par rapport au gestionnaire de modèle (*Kevoree Core*). De même, le gestionnaire d'infrastructure nécessite lui aussi peu de code principalement grâce au gestionnaire de modèle qui offre de nombreuses fonctionnalités pour manipuler le modèle, mais aussi grâce au langage KevScript qui permet d'exprimer simplement des modifications de la configuration.

De plus, le gestionnaire d'infrastructure n'est en rien directement dépendant des types de nœuds mais bien des types abstraits proposés par l'API de KevoreeKloud. De cette manière, ce gestionnaire peut tout à fait être réutilisé sur une autre infrastructure. Par exemple, il est envisageable de remplacer le *proxy EC2* par un autre *proxy* pour Microsoft Azure par exemple et de remplacer le type *JailNode* par un type *LXCNode* (qui est une alternative au *Jails* sur Linux)

5.2 Évaluation 2 : Est-ce utilisable pour des Clouds ?

L'objectif de cette section est de montrer que l'utilisation de notre abstraction basée sur les techniques de Model@Runtime n'a pas d'impact négatif sur le temps de reconfiguration nécessaire dans le cadre d'un Cloud. Ce temps de reconfiguration correspond au temps nécessaire pour le déploiement de nouvelles machines virtuelles intégrées dans la plate-forme existante. C'est ce temps de déploiement qui définit l'instant à partir duquel la plate-forme peut utiliser ces machines virtuelles pour déployer des composants dessus.

5.2.1 Protocole expérimental

Dans cette évaluation, nous allons présenter l'impact de notre abstraction sur le temps de déploiement de nouveaux nœuds de plate-forme ainsi que l'impact sur la quantité de mémoire vive utilisée pour stocker le modèle. Pour cela, nous avons déployé des applications de grandes tailles sur notre implantation de Cloud. Le déploiement de ces applications implique le déploiement d'un certain nombre de nœuds de type plate-forme.

Le déploiement d'un nœud de type plate-forme correspond à la mise en place d'un système virtualisé sur l'un des nœuds de l'infrastructure. Dans notre cas d'étude, ce déploiement consiste à créer un espace utilisateur sur l'une des machines FreeBSD. Ce déploiement peut aussi consister à déployer une machine virtuelle sur le Cloud EC2.

Afin de montrer l'impact sur le déploiement des *Jails*, nous avons mesuré le temps nécessaire à la mise en place de la configuration d'un cas d'étude réel qui utilise 50 nœuds de plate-forme déployés sur 10 nœuds d'infrastructure. Pour mesurer le temps de déploiement, nous avons intégré, aux gestionnaires de plate-forme et d'infrastructure, un mécanisme de trace (*logger*) en temps absolu qui émet des évènements vers un serveur qui effectue une réconciliation du temps par rapport à une horloge de référence (voir Java Greg Logger [6]). Ce serveur permet à chaque client de synchroniser périodiquement le temps de latence entre l'horloge de référence et celle du client permettant ensuite au serveur de prendre en compte cette latence et la latence réseau pour calculer un temps absolu.

Les traces publiées sont émises à partir des endroits ci-dessous :

- **trace 1** : réception du modèle par le gestionnaire de plate-forme
- **trace 2** : soumission d'un nouveau modèle par le gestionnaire de plate-forme après définition des nœuds de plate-forme nécessaire
- **trace 3** : réception du modèle par le gestionnaire d'infrastructure
- **trace 4** : soumission d'un nouveau modèle par le gestionnaire d'infrastructure après définition de l'hébergement des nœuds de plate-forme non alloués
- fin de la reconfiguration sur un nœud

Ces différentes traces permettent de calculer le temps nécessaire pour chaque opération que ce soit la génération d'un modèle par le gestionnaire de la plate-forme, la génération d'un modèle par l'infrastructure et la mise en place de la reconfiguration.

6. http://code.google.com/p/greg/

Nous comparons ensuite l'ensemble des temps obtenus pour estimer le surcoût (*overhead*) introduit par KevoreeKloud et qui correspond au temps de génération des modèles.

Outre les résultats sur une application réelle, nous avons simulé des modèles de tailles variées afin de montrer l'impact de la taille du modèle sur l'utilisation mémoire de notre système de Cloud. Pour cela, nous avons généré un modèle avec seulement cinq nœuds qui correspond à un sous-ensemble du modèle de l'application réelle. Nous avons ensuite généré quatre autres modèles respectivement de 500, 5000, 50000 et 100 000 nœuds. Ces modèles étant trop important par rapport à la taille de notre infrastructure, la reconfiguration n'a pu être finalisée, mais nous avons tout de même obtenu les temps de traitement du modèle (il manque le temps de démarrage des jails qui ne correspond pas à l'overhead de KevoreeKloud).

En plus du temps de déploiement, nous avons aussi évalué l'impact de notre abstraction sur la quantité de mémoire vive nécessaire à son bon fonctionnement. Nous avons mesuré ici la taille mémoire utilisée pour représenter la configuration globale du Cloud en mémoire afin d'évaluer l'impact du modèle KevoreeKloud sur l'utilisation mémoire. Pour cela, nous avons mesuré la taille en mémoire des modèles précédemment utilisés dans l'évaluation de l'impact sur le temps de déploiement. Nous avons, en plus de ces modèles, synthétisé un modèle pouvant représenter la configuration du Cloud d'Amazon selon les données de Guy Rosen [77] pour qui Amazon EC2 comptait autour de 50 000 machines virtuelles en septembre 2009 ainsi que selon les données de Huan Liu [81] pour qui Amazon EC2 comptait autour de 7000 servers en Mars 2012.

Pour finir, nous avons aussi évalué l'extensibilité et la généricité de notre solution en regardant le nombre de lignes de code nécessaires pour concevoir cette nouvelle plateforme de Cloud. En plus des évaluations concernant l'empreinte mémoire et l'impact sur le temps de reconfiguration, nous avons de nouveau mesuré le nombre de lignes de code nécessaire pour la définition de ces nouveaux types de nœud et de gestionnaire.

5.2.2 Implémentation du cas d'étude

Afin de déployer des applications de grandes tailles, nous avons choisi de traiter un cas d'étude particulier qu'est le test logiciel dans le cadre de l'intégration continue.

L'intégration continue est un principe de génie logiciel visant à suivre l'évolution du développement d'un logiciel. En pratique, cela consiste principalement à automatiser la compilation de l'application et les phases de tests. Ainsi, l'ensemble des développeurs intègre leurs modifications sur un serveur de source (Git, SVN, Mercurial) et pour chaque commit, le serveur d'intégration effectue une vérification de ces modifications afin d'assurer que l'application compile et que les tests soient tous valides. Si quelque chose échoue, les acteurs autour du projet sont capables de savoir quelle est la modification ayant entrainé cet échec et il est ainsi plus facile de pouvoir la corriger. La figure 5.4 résume ce processus. Pour plus d'information, nous vous invitons à lire l'article de Martin Fowler et Matthew Foemmel [63].

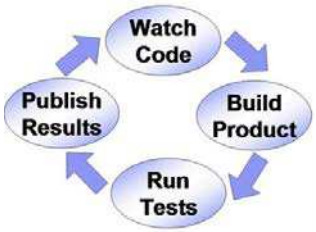


FIGURE 5.4 – Processus d'intégration continue

Du fait de l'importance prépondérante de l'informatique dans notre société actuelle, les systèmes informatiques sont de plus en plus nombreux et surtout de plus en plus complexes. De ce fait, le test de ce genre de système est quelque chose de complexe. Ainsi, de nombreux projets possèdent plusieurs centaines et même milliers de tests permettant d'assurer la stabilité du logiciel. Par exemple, Rothermel *et al.* affirme, dans leur article [109], que l'exécution des tests unitaires de certains projets de leurs collaborateurs industriels nécessitait jusqu'à 7 semaines.

Ce genre de durée ne peut correspondre à l'idée mise en avant dans l'intégration continue qui est de pouvoir détecter rapidement les modifications entraînant des régressions et des erreurs afin de pouvoir les corriger tout aussi rapidement. Plusieurs travaux ont été menés pour tenter de diminuer ce temps. Il existe par exemple des approches essayant de sélectionner un sous-ensemble des suites de tests à exécuter en fonction des modifications apportées [75, 117]. Un autre type d'approche est de définir des priorités sur l'exécution des tests en exécutant les tests les plus importants d'abord [117, 88]. Ces techniques ont pour objectifs de détecter le plus tôt possible les régressions ou erreurs dues aux modifications. Enfin il y a les techniques cherchant à distribuer l'exécution des tests [82, 60].

Mais outre la problématique de la durée d'exécution des tests, il y a aussi la problématique des contraintes d'exécution de ces tests. En effet, si un logiciel doit être multi plate-forme, il doit pouvoir s'exécuter sur plusieurs systèmes différents, que ce soit en termes de système d'exploitation, de processeur, de mémoire vive, etc. De même, certains tests ont besoin d'isolation afin de permettre la bonne exécution du reste du processus. Un exemple simple est l'utilisation de la Kinect au travers de l'API freenect [7] codée en C. Cette API propose aussi un wrapper Java qui possède trois tests unitaires. Cependant, l'exécution de chacun des tests peut tuer le processus de la machine Java à cause d'un bug de l'API qui produit une erreur de segmentation lors de la tentative de connexion à la Kinect et que celle-ci n'est pas connectée à la machine. Si la machine Java est tuée alors il n'est pas possible d'obtenir le moindre résultat concernant les tests puisque le processus d'exécution va être tué.

Le Cloud Computing est une réponse à cette problématique. En effet, il est possible de réserver autant de machines nécessaires pour pouvoir exécuter les tests sur l'ensemble des configurations requises. Les serveurs d'intégration continue actuels fournissent ce

7. http://openkinect.org/wiki/Main_Page

genre de capacité. C'est le cas par exemple de Jenkins et de plusieurs de ses *plugins* qui permettent de réserver des instances spécifiques sur des infrastructures de Cloud telles que Amazon EC2 ou encore DeltaCloud puis d'exécuter la compilation et le test des projets sur ces instances. Mais cette solution d'interaction avec une infrastructure reste une solution ad hoc au plugin qui la propose et brise l'encapsulation offerte par une plate-forme puisque c'est l'application qui directement demande à l'infrastructure de lui fournir de nouvelles ressources (nœuds de plate-forme) pour s'exécuter.

De plus, la distribution des tests n'est pas prise en compte puisque la répartition se fait par projet et non pas par suite de tests et encore moins par tests. Ainsi, pour un projet devant être testé sur différentes plates-formes, l'ensemble des tests va être exécuté pour chacune des configurations alors que seulement un sous-ensemble doit prendre en compte cet aspect (par exemple, les tests d'une interface GTK sur un système ayant la bibliothèque GTK). Là encore, une première solution consiste à modulariser le projet afin de pouvoir définir des processus d'intégration continue sur chacun des modules. C'est déjà l'approche qui est utilisée dans un grand nombre de projets. Cela permet de limiter le temps d'exécution de la compilation et du test de chaque module et permet aussi de répartir l'exécution de chaque module afin de les exécuter en parallèle et sur des plates-formes spécifiques si nécessaire. Cependant, cette modularisation est effectuée deux fois. Tout d'abord lors de la définition du projet et ensuite lors de la définition de son intégration continue. De plus, à chaque modification de la modularisation du projet, il est nécessaire de modifier les processus d'intégration continue. Enfin, si un module doit être testé dans différents environnements, il est nécessaire de définir ces environnements et de définir les processus d'intégration continue correspondants.

La définition des processus d'intégration continue selon les environnements pourrait pourtant être évitée ou au moins simplifier puisque la majorité des informations sont connues lors de la modularisation du projet. La définition d'autant de processus d'intégration continue que de plates-formes à tester peut, de plus, être automatisée en spécifiant les différents critères pour chacun des modules ou même pour chacune des suites de tests voir pour chacun des tests unitaires. La spécification de critères peut aussi être intéressante dans le cadre de l'exécution isolée des tests. C'est le cas dans le cadre du test d'application Java ayant du code natif qui s'exécute comme l'usage du *wrapper* Java de *Libfreenect*. En effet, dans ce wrapper, il y a trois tests qui vont entrainer un crash de la machine virtuelle dans lesquels ils s'exécutent si la kinect n'est pas connectée à la machine empêchant ainsi l'exécution des autres tests existants. Avoir la possibilité d'exécuter chacune des suites de tests de manière indépendante ou même chaque test unitaires de manière isolée peut assurer que l'ensemble des tests soit effectué même si certains sont en erreur ou plus simplement cela va permettre d'avoir un résultat sur l'exécution des tests montrant ceux qui ont échoué.

Nous avons donc développé une extension (*plugin*) pour le serveur d'intégration continue *Jenkins*[8] permettant à partir d'un projet de générer une architecture d'application Kevoree regroupant l'ensemble des tests à exécuter. Cette architecture d'application est soumise à une plate-forme spécifique capable de déployer l'exécution de chaque

8. http://jenkins-ci.org

test sur des nœuds de la plate-forme en fonction de leurs contraintes d'exécution. La plate-forme est capable de demander des ressources à l'infrastructure sous-jacente sans pour autant nécessiter une connaissance particulière de celle-ci. Pour cela, nous avons défini un composant de gestion de la plate-forme capable de définir de nouveaux nœuds de plate-forme n'étant pas hébergés par l'infrastructure. Cette définition de nouveaux nœuds se traduit par la proposition d'un nouveau modèle pour le système.

L'infrastructure et son gestionnaire sont donc notifiés de la mise en place de ce nouveau modèle. Le gestionnaire (voir listing 5.3) que nous avons défini suivant la philosophie du protocole map-reduce est lui capable de détecter que des nœuds de type plate-forme ne sont pas hébergés par l'infrastructure. Il peut donc décider de les placer sur les ressources de l'infrastructure.

5.2.2.1 Plate-forme de déploiement de tests unitaires

Il existe plusieurs frameworks pour la définition de tests. Nous pouvons citer par exemple JUnit [9] ou TestNG [10] pour les tests unitaires en Java.

JUnit permet de définir des tests unitaires avec la possibilité de définir une gestion de cycle de vie des tests et des paramètres d'exécution de ces tests par l'intermédiaire de plusieurs annotations spécifiques (*@Before*, *@BeforeClass*, *@After*, *@AfterClass*). TestNG propose quant à lui un peu plus de fonctionnalités dont entre autres la capacité à paralléliser l'exécution des tests sur une même machine en utilisant des threads différents. Il est indiqué aussi sur le site du projet que TestNG est capable de distribuer l'exécution des tests sur un ensemble de machines sur lesquelles s'exécuterait un esclave TestNG. Cependant, cette fonctionnalité est limitée et ne semble pas avoir été maintenue ou en tout cas n'est pas mise en avant sur les versions actuelles. De même que TestNG, GridUnit [60] propose aussi de distribuer l'exécution des tests et plus précisément de distribuer les tests sur une grille de calcul.

Bien que TestNG fournisse quelques fonctionnalités intéressantes pour effectuer de l'exécution parallèle voire distribuée des tests, cela n'est pas suffisant. En effet, outre le fait de vouloir paralléliser et distribuer les tests à exécuter, nous souhaitons aussi pouvoir spécifier les caractéristiques des plates-formes d'exécution. De plus, en aucun cas TestNG ne permet d'utiliser des infrastructures de Cloud pour déployer les plates-formes et nécessite une configuration statique de l'ensemble des machines permettant de distribuer les tests (voir le blog de l'auteur [11] pour plus de détails).

Du fait que ni JUnit, ni TestNG ne permettent de spécifier l'ensemble des informations nécessaires pour pouvoir exécuter les tests unitaires de manière distribuée, parallèle et isolée, nous avons défini un framework supplémentaire pour la définition de tests unitaires et plus particulièrement pour la définition des caractéristiques d'exécution de ces tests.

Pour cela, nous proposons un DSL interne défini à l'aide d'un ensemble d'annotations Java. Ces annotations permettent d'une part de spécifier les propriétés nécessaires de

9. http://www.junit.org/
10. http://testng.org/
11. http://beust.com/weblog2/archives/000362.html(accessibleenjanvier2013)

la plate-forme d'exécution (voir listing 5.6).

Listing 5.6 – Annotations pour la configuration de la plate-forme d'exécution

```
@OS {values = {"Windows XP", "Ubuntu 12.04", "FreeBSD 9"}}
@RAM {values = {"512MB", "1GB"}}
@CPU_CORE {values = {"2"}}
@CPU_FREQUENCY {values = {"3GHz"}}
```

Il est ainsi possible de définir le type de système d'exploitation, le nombre de cœurs du processeur, la fréquence du processeur et la quantité de mémoire disponible. Chacune de ces annotations peut prendre plusieurs valeurs afin de spécifier l'ensemble des possibilités pour chaque caractéristique. Ces caractéristiques sont ensuite composées pour définir l'ensemble des systèmes sur lesquels le test ou la suite de tests doit être exécuté. Il est aussi possible de spécifier des configurations spécifiques plutôt qu'un ensemble de propriétés qui seront composées (voir listing 5.7).

Listing 5.7 – Annotation pour la définition de configuration spécifique

```
@Configuration ({
  @OS {values = {"FreeBSD 9"}},
  @RAM {values = {"512MB"}},
  @CPU_CORE {values = {"2"}},
  @CPU_FREQUENCY {values = {"3GHz"}}
})
```

En plus de ces annotations spécifiant les contraintes de la plate-forme d'exécution, nous proposons aussi un ensemble d'annotations permettant de définir des caractéristiques non plus sur l'environnement d'exécution, mais sur l'exécution elle-même (voir listing 5.8). Ces notations permettent de spécifier si le test ou la suite de tests doit s'exécuter de manière isolée afin d'assurer que si l'exécution échoue, elle n'aura pas d'impact sur l'exécution des autres tests comme cela peut être le cas avec les tests sur la Kinect. Il est aussi possible de définir une notion de dépendance entre différents tests. Cette dépendance permet d'expliciter le fait que si le test A a échoué alors cela ne sert à rien d'exécuter le test B puisque celui-ci utilise la fonctionnalité testée par A. En plus de l'isolation des tests et de leur dépendance, il est possible de définir que des tests peuvent être exécutés en parallèle d'autres tests. Cette information signifie que hormis pour les tests, dont le test A dépend, il est possible d'exécuter le test A avec d'autres tests sur la même plate-forme. Cette propriété est intéressante notamment dans le cadre de tests unitaire ne cherchant pas à tester les performances mais simplement la fonctionnalité en elle-même. Enfin dans le cadre du test de performance, nous proposons de spécifier la durée maximale autorisée pour l'exécution d'une fonctionnalité.

Avec ce framework, nous fournissons une suite d'outils intégrée au sein d'un plugin maven capable de synthétiser, à partir d'un projet, l'architecture logicielle regroupant l'ensemble des composants correspondant à une suite de tests ou un test. L'architecture

Listing 5.8 – Annotations pour la définition de configuration spécifique

```
@Configuration ({
  @VMARGS {values = {"-Xms512m", "-Xmx1024m", "-XX:PermSize=256m",
       "-XX:MaxPermSize=512m"}},
  @ISOLATED
  @DEPENDS_ON ({
    @Test {className = org.kevoree.test.TestSuite1.class, testName = "test1"},
    @Test {className = org.kevoree.test.TestSuite2.class, testName = "test2"}
  }),
  @PARALLEL
  @TIMEOUT {values = {"10000"}}
})
```

logicielle définit au travers des canaux de communication les relations entre les composants. De cette manière, un composant ne peut exécuter le test associé que lorsqu'il reçoit un message explicite. Ce message est envoyé par les composants hébergeant les tests qui sont en dépendance du test hébergé. Une fois que le composant a reçu les messages de tous les composants dont il dépend, il peut exécuter le test qu'il héberge. En plus des composants représentant les différents tests et les canaux de communication représentant les dépendances de test, l'architecture logicielle contient un composant spécifique chargé de la gestion des tests. Cette gestion se caractérise par le démarrage des tests et l'agrégation des résultats. Ainsi, le composant de gestion est connecté à tous les composants n'ayant pas de dépendance afin de pouvoir les déclencher comme le font les composants entre eux. Les composants de tests émettent aussi les résultats des tests qu'ils envoient au gestionnaire sous la forme de données XML conformes à une grammaire communément utilisée par les outils autour de Junit. Ce format est ainsi proposé à l'origine par la tâche ant de JUnit [12] et qui est aussi proposé par le plugin maven Surefire [13] développé par Apache. C'est aussi un format souvent utilisé par les plugins de jenkins pour manipuler les résultats de tests.

L'architecture logicielle définit aussi les contraintes sur les plates-formes d'exécution. Pour cela, les composants sont hébergés par des nœuds ayant les caractéristiques associées aux tests et plusieurs tests peuvent être hébergés sur le même nœud. En plus des caractéristiques de plate-forme, l'annotation *ISOLATED* permet aussi de définir une contrainte sur l'exécution puisqu'elle spécifie que le test doit être exécuté dans un processus spécifique. Pour cela, nous avons défini un type de nœud que nous avons appelé *MiniCloudNode* qui permet de créer de nouvelles machines virtuelles Java sur le même système (voir listing 5.9 pour la définition du type).

Une fois l'architecture définie, la suite d'outil développé dans le cadre de cette expérimentation envoie le modèle à la plate-forme d'hébergement. Cette plate-forme d'hébergement définit le déploiement des composants sur la plate-forme en fonction de l'architecture fournie (voir la section 4.3), des nœuds disponibles et des nœuds qu'il peut créer (voir listing 5.10).

12. http://ant.apache.org/manual/Tasks/junit.html
13. http://maven.apache.org/plugins/maven-surefire-plugin/

Listing 5.9 – Définition du type MiniCloud

```
@DictionaryType({
    @DictionaryAttribute(name = "VMARGS", optional = true)
})
@NodeType
public class MiniCloudNode extends HostNode {
}
```

Pour tenir compte des relations entre les composants, et sachant que ce gestionnaire est un gestionnaire spécialisé dans la distribution des tests, il construit un graphe de relation entre les composants qui correspond au graphe de dépendances entre les tests ou les suites de tests. Il place ensuite les composants de façon à limiter le nombre de tests qui s'exécute en même temps sur un nœud de plate-forme. Pour cela, il choisit de positionner un composant et au moins l'un de ces successeurs sur le même nœud. De cette façon, si le composant B dépend du composant A alors le composant A devra finir de s'exécuter avant que B puisse commencer. La figure 5.5 représente un exemple de répartition de composants sur les nœuds et incluant les relations d'ordre (correspondant aux dépendances) entre ces composants. Sur cette figure, nous pouvons voir que le composant *Camel Core* ne possède aucune dépendance et que quasiment l'ensemble des autres composants dépend de lui. Ainsi, l'exécution débutera par le *Camel Core*. Une fois que celui-ci sera terminé, il notifiera les composants qui lui sont connectés afin qu'ils débutent leur exécution et ainsi de suite.

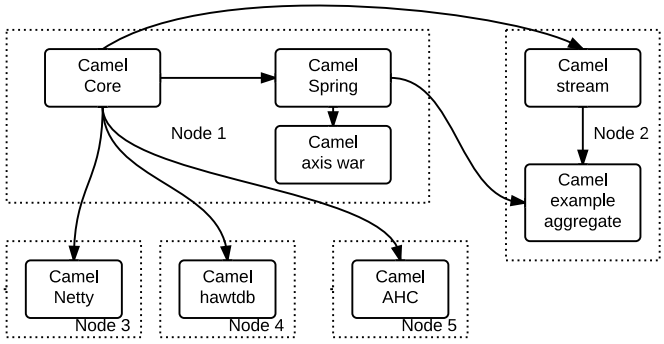


FIGURE 5.5 – Graphe de dépendances et allocation des composants sur les nœuds

5.2.2.2 Résultat sur un projet concret : Apache Camel

Nous avons effectué notre expérimentation sur le projet Apache Camel[14]. Apache Camel est un framework qui implante l'ensemble des EIPs pour Enterprise Integration Patterns [79] en anglais. Ces patterns décrivent l'ensemble des opérations couramment utilisées dans le cadre de l'intégration d'applications. Apache Camel propose un DSL

14. http://camel.apache.org/

Listing 5.10 – Définition du gestionnaire de la plate-forme

```
@ComponentType
class PaaSManager extends AbstractComponentType implements ModelListener {
  def process(newPaaSModel : ContainerRoot) {
    val graph = buildDependencyGraph(newPaaSModel)
    val index = new DirectedNeighborIndex(graph)
    val firstStep = graph.vertexSet(). filter (v → index.predecessorsOf(v). size () == 0).toList
    var number = 0
    var alreadySeenList = List[ComponentInstance]()
    var previousStep = firstStep
    var currentModel = getModelService.getLastModel
    while (number < graph.vertexSet().size()) {
      var newStep = List[ComponentInstance]()
      previousStep.foreach {
        p →
          val nodeName = findNodeForComponent(p)
          var first = true
          val tmp = index.successorListOf(p). filter (v → index.predecessorsOf(v). forall (pred →
              alreadySeenList.contains(pred))). toList
          newStep = newStep ++ tmp
          tmp.foreach {
            component →
              val kengine: KevScriptEngine =
                  getKevScriptEngineFactory.createKevScriptEngine(currentModel)
              val componentName = component.getName
              if ( first ) {
                first = false
                // add the first sucessor on the same node than its current predecessor
                addComponentOnNode(componentName, nodeName, currentModel, kengine)
              } else {
                // try to find an existing node to host the component
                if (!chooseEquivalentNode(currentModel, model, nodeName, kengine)) {
                  // create a new node to host the component
                  val tmpNodeName = findNodeForComponent(p)
                  val newNodeName = buildNode(tmpNodeName, model, kengine)
                  addComponentOnNode(componentName, newNodeName, currentModel, kengine)
                }
              }
              // apply changes (on a temporary model) to take into account into the next iteration
              currentModel = kengine.interpret()
          }
      }
      alreadySeenList = alreadySeenList ++ newStep
      number = number + newStep.size
      previousStep = newStep
    }
    try {
      getModelService.atomicCompareAndSwapModel(uuidModel, currentModel)
    } catch {
      case e : Throwable →
    }
  }
}
```

permettant d'utiliser ces patterns afin de simplifier la construction des routines de communication permettant l'intégration de deux logiciels. Ce projet comptait 8084 tests (sans compter les trois modules qui n'ont pas pu compiler) dans la version 2.8 pour un

temps d'exécution des tests d'environ une heure et quarante minutes.

Dans notre expérimentation, nous avons choisi de découper l'exécution des tests au niveau des suites de tests qui sont au nombre de 3845 pour 175 modules, sachant que certains modules n'ont pas pu être compilés et donc que leurs suites de tests ne sont pas comptées. Nous n'avons pas spécifié de dépendance sur l'isolation de l'exécution de ces suites de tests, ainsi plusieurs suites de tests peuvent être exécutées dans la même machine virtuelle Java. De même, nous n'avons pas spécifié de dépendances vis-à-vis d'un système d'exploitation afin que l'ensemble des nœuds de plate-forme soit créé en tant que *Jails*. Outre les caractéristiques que nous n'avons pas spécifiées, l'ensemble des suites de tests partage aussi trois caractéristiques communes que sont les propriétés pour la machine virtuelle Java (-Xmx1024m -XX :MaxPermSize=512m). Ces spécificités proviennent de la documentation du projet Apache Camel qui nous informe des contraintes nécessaires sur la machine virtuelle Java pour exécuter l'ensemble des tests. Ces contraintes fournissent par la même occasion une contrainte sur la quantité mémoire que doivent posséder les nœuds de plate-forme (plus de 1024Mb). La troisième contrainte identique correspond au fait que l'ensemble des tests peut être parallélisé. Enfin, la dépendance entre les suites de tests a été définie en fonction des dépendances entre les différents modules du projet Apache Camel. Cette contrainte va permettre de définir les interactions entre les différents composants.

Du fait que notre Cloud d'expérimentation soit limité en ressource, nous avons choisi de limiter le nombre de nœuds de plate-forme à 50 soit 5 par nœud d'infrastructure. Pour cela, nous avons surchargé la méthode *chooseEquivalentNode* qui permet au gestionnaire de trouver parmi les nœuds déjà existants, un nœud correspondant aux contraintes du composant à déployer. Dans notre cas d'utilisation, le gestionnaire est capable de créer 50 nœuds puis la méthode chooseEquivalentNode retourne obligatoirement un de ces nœuds pour héberger le composant. Pour cela, cette méthode utilise les informations disponibles dans le modèle lui permettant de savoir que notre infrastructure se compose de 10 machines ayant chacune 8Gb de mémoire. Bien que cette contrainte n'est pas d'intérêt tel quel dans une utilisation sur une infrastructure telle qu'Amazon, elle pourrait être remplacée par une limitation en fonction du prix que l'utilisateur serait prêt à payer.

Au final, nous avons donc 3845 composants à exécuter sur 50 Jails qui s'exécutent en 35 minutes (soit un speedup de 2.8) sachant que le test le plus long prend 20 minutes et qu'il nous faut environ 7 minutes pour démarrer les 50 Jails. Le fait qu'il faut plus de 27 minutes (test le plus long + temps d'initialisation des jails) vient du fait des dépendances entre les composants qui ne permettent pas de tous les exécuter en parallèle.

Les résultats sur ce cas concret nous montrent que la distribution et la parallélisation de l'exécution des tests unitaires est quelque chose d'intéressant. En effet, bien que ce projet ne nécessite qu'une heure et quarante minutes pour exécuter les tests, le speedup de 2.8 nous permet d'envisager un gain non négligeable sur des projets nécessitant plusieurs jours.

5.2.3 Évaluation

5.2.3.1 Impact sur le déploiement

Le tableau de la figure 5.6 montre les délais entre la réception de modèle par le gestionnaire de plate-forme et les différentes étapes de la mise en place de ce nouveau modèle.

	5	**50**	**500**	**5000**	**50000**	**100000**
trace 1	0	0	0	0	0	0
trace 2	27	176	2188	32185	337638	684650
trace 3	193	361	2571	34920	345263	693205
trace 4	222	543	4766	67115	682914	1377870
trace 6	223537	425306				

FIGURE 5.6 – Temps des réceptions des traces

La figure 5.7 représente les différentes données du tableau précédent sous forme de courbes.

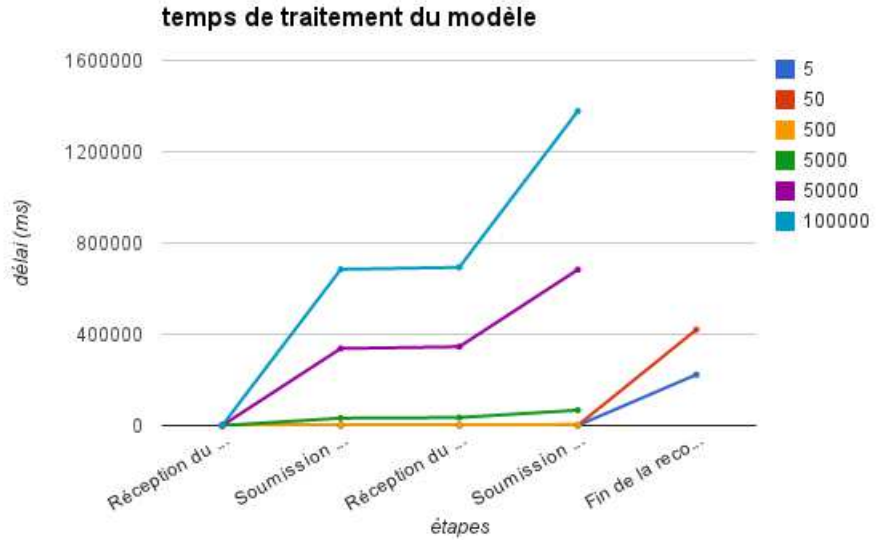


FIGURE 5.7 – Temps de traitement du modèle

Ces données nous montrent que dans le cadre du projet *Camel* (la courbe pour 50 nœuds et finissant autour des 400000 ms), le traitement du modèle utilisateur pour définir les nœuds de plate-forme utilisés pour le déploiement des tests est sensiblement le même que pour le positionnement de ces nœuds sur l'infrastructure. Nous pouvons noter aussi que le temps nécessaire pour le passage entre la trace 2 et la trace 3 reste faible en comparaison des autres délais. Cela s'explique par le fait que même si les nœuds de plate-forme sont définis, ils ne sont en aucun cas hébergés et donc le système n'a pas

besoin d'appliquer une mise à jour concrète. Ici la soumission d'un nouveau modèle par la plate-forme tire parti du fonctionnement du *Kevoree Core* et notamment de la gestion des *ModelListeners* pour notifier l'infrastructure du changement. C'est cette notification qui permet de déclencher l'exécution du gestionnaire de l'infrastructure.

Si l'on compare les temps de traitement du modèle avec la mise en place des nœuds de plate-forme (ici les jails), nous pouvons voir que le temps de traitement est faible en comparaison avec la mise en place des jails. Cependant, si l'on regarde les temps de traitement pour les modèles de l'ordre de 50 000 et 100 000 nœuds, le temps nécessaire pour définir l'hébergement des nœuds de plate-forme sur l'infrastructure devient beaucoup plus important. Cela est principalement dû au fait de devoir, pour chaque nœud de plate-forme, sélectionner une IP qui n'est pas encore utilisée dans le réseau de l'infrastructure. Si nous prenons les mêmes modèles, mais que seulement 5 nœuds ont besoin d'un hébergement (les autres sont déjà en place) alors ce temps est fortement réduit.

5.2.3.2 Impact sur l'utilisation mémoire

Le tableau de la figure 5.8 récapitule les chiffres correspondants à la taille mémoire des différents modèles que nous avons évalués.

Modèle	**taille (Mega octets)**
50 nœuds et 4000 éléments	3,72
500 nœuds et 40000 éléments	35,88
5000 nœuds et 100000 éléments	91,97
5000 nœuds et 200000 éléments	160,47
5000 nœuds et 300000 éléments	269,58
5000 nœuds et 400000 éléments	357,47
100000 nœuds et 400000 éléments	424,44
507000 nœuds et 400000 éléments	721,21

FIGURE 5.8 – Espace mémoire pour le stockage d'un modèle

Nous pouvons voir que l'utilisation d'un modèle global est quelque chose d'utilisable pour des systèmes de cette taille mais les ressources, ici la mémoire, utilisées sont tout de même importantes.

Pour autant, les chiffres qui ont été mesurés correspondent à des modèles complets ne tirant pas parti du partitionnement. En effet, l'ensemble des informations conservées dans les modèles peut tout à fait être limité (voir section 4.3). Par exemple, même si l'infrastructure peut se servir de la définition des interactions entre les composants pour positionner les nœuds hébergeant ces composants, il n'est pas nécessaire de connaître les caractéristiques de l'implémentation de ces composants. Il était donc possible de faire du partitionnement au niveau des types en utilisant des abstractions permettant de limiter les informations de types et de paramétrage.

De même, l'implémentation de notre représentation modèle n'est en aucun cas optimisée pour des systèmes de très grande taille. Il serait donc nécessaire d'envisager l'utilisation de solution de base de données par exemple pour permettre le stockage sur

disque des informations qui ne sont pas nécessaires et donc peu voire pas utilisées. Par exemple, dans le cadre du modèle permettant de représenter la configuration d'Amazon EC2, il est possible de ne conserver en mémoire que les informations concernant le datacenter courant plutôt que l'ensemble des datacenters.

5.2.3.3 Complexité de l'implémentation de nouveaux types et gestionnaire

Le tableau de la figure 5.9 présente le nombre de lignes de code pour un certain nombre de type.

Projet	**nombre de lignes de code**
PJavaSENode	12
PMiniCloudNode	12
MiniCloudNode	150
PaaSManager	309

FIGURE 5.9 – Nombre de lignes de code selon le projet

Le gestionnaire de plate-forme dans ce cas d'étude contient 309 lignes de code (sans compter le plugin maven pour la génération du modèle de l'application cliente). Là encore, nous pouvons constater que la définition d'un gestionnaire de plate-forme nécessite peu de ligne de code grâce au Kevoree Core ainsi qu'au langage *KevScript*. Concernant l'implantation des types de nœuds de plate-forme, nous utilisons le type *PJavaSENode* ainsi que *PMiniCloudNode* qui hérite respectivement de *JavaSENode* et *MiniCloudNode*. C'est deux types de nœuds de redéfinissent rien, que ce soit pour la la planification ou pour l'exécution des primitives car ils possèdent les mêmes caractéristiques que les types parents avec en plus un héritage du type PaaSNode qui permet de définir des caractéristiques spécifiques pour les nœuds de plate-forme. Ainsi, ces types contiennent chacun 12 lignes de code et le type MiniCloud contient 150 lignes de code.

5.3 Évaluation 3 : Est-ce utilisable pour de l'adaptation multi-niveaux ?

L'objectif de cette section est de montrer que l'utilisation de notre abstraction permet bien de gérer l'adaptation comme une problématique transverse en permettant la définition de politiques d'adaptation multi-niveaux, c'est-à-dire d'adaptation capable d'utiliser les informations fournies par les différents niveaux, mais aussi capable de tirer parti des capacités d'adaptation d'un niveau pour améliorer l'état d'un autre niveau. Pour pouvoir faire de l'adaptation multi-niveaux, il faut d'une part que chaque moteur de raisonnement puisse avoir accès aux informations concernant l'ensemble des niveaux. D'autre part, l'ensemble des moteurs de raisonnement doit être capable de collaborer pour définir les adaptations les plus efficaces.

5.3.1 Protocole expérimental

Dans cette évaluation, nous allons présenter comment le fait d'avoir accès un support commun pour la configuration des différents niveaux permet d'envisager des adaptations plus performantes.

Tout d'abord, nous allons montrer que l'accès aux informations de ces différents niveaux permet d'envisager des adaptations plus efficaces. Pour cela, nous allons définir un gestionnaire d'infrastructure capable de tenir compte des connexions entre composants pour placer à proximité les nœuds de plate-forme hébergeant ces composants.

Nous allons montrer ensuite que le fait d'avoir un support commun pour la définition des reconfigurations permet de faire collaborer les moteurs de raisonnement. Ici nous allons montrer que la collaboration entre les moteurs de raisonnement chargés de l'infrastructure et de la plate-forme permet de limiter les migrations de machines virtuelles en les remplaçant par des migrations de composants ou par la définition de composant de cache permettant de limiter les requêtes vers le composant qui utilise beaucoup de bande passante.

5.3.2 Implémentation du cas d'étude

L'élasticité et la scalabilité sont les deux principales caractéristiques d'un Cloud (voir section 1.1). En effet, l'un des intérêts de migrer ses applications sur un Cloud est de pouvoir assurer aux utilisateurs une qualité de service suffisante. Pour cela, la plate-forme d'hébergement est le plus souvent capable de dynamiquement assurer cette qualité de service par l'intermédiaire de duplication des composants de l'application et de l'utilisation de techniques de load-balancing. Ainsi, un serveur web distribué sera défini comme un point d'accès que l'on peut appeler le serveur et un ensemble de composants capables de générer les pages du ou des sites hébergés.

L'implantation de ce cas d'étude se découpe en deux parties. Nous allons tout d'abord présenter comment sont définis un serveur web et les pages web hébergées. Ces pages web sont des pages statiques. Nous présenterons ensuite l'implantation du gestionnaire d'élasticité permettant de gérer la duplication ou la migration de composants avec en plus la possibilité de créer de nouveaux nœuds de plate-forme en fonction des besoins.

5.3.3 Définition du serveur web distribué

Pour implémenter ce cas d'étude, nous avons défini deux types de composants.

Le premier type définit l'interface d'une page web (voir listing 5.11) qui permet de traiter les requêtes qu'elle reçoit et de retourner des réponses à ces requêtes. Cette page web définit aussi l'URL relative dont elle a la charge.

Le second type définit l'interface d'un serveur web qui écoute sur un port, qui envoie les requêtes client qu'il reçoit à ceux capables de les traiter, puis lorsqu'il reçoit les réponses correspondantes, les envoie aux clients (voir listing 5.12).

La définition d'un site web consiste donc à associer un serveur avec un ensemble plus ou moins grand de pages web (voir figure 5.10 pour un exemple).

Listing 5.11 – Définition d'une page web

```
@Provides({
    @ProvidedPort(name = "request", type = PortType.MESSAGE)
})
@Requires({
    @RequiredPort(name = "content", type = PortType.MESSAGE),
    @RequiredPort(name = "forward", type = PortType.MESSAGE, optional = true)
})
@DictionaryType({
    @DictionaryAttribute(name = "urlpattern", optional = true, defaultValue = "/")
})
@ComponentFragment
public abstract class AbstractPage extends AbstractComponentType {
```

Listing 5.12 – Définition d'un serveur web

```
@DictionaryType({
    @DictionaryAttribute(name = "port" , defaultValue = "8080"),
    @DictionaryAttribute(name = "timeout" , defaultValue = "5000", optional = true)
})
@Requires({
    @RequiredPort(name = "handler", type = PortType.MESSAGE)
})
@Provides({
    @ProvidedPort(name = "response", type = PortType.MESSAGE)
})
@ComponentFragment
public abstract class AbstractWebServer extends AbstractComponentType {
```

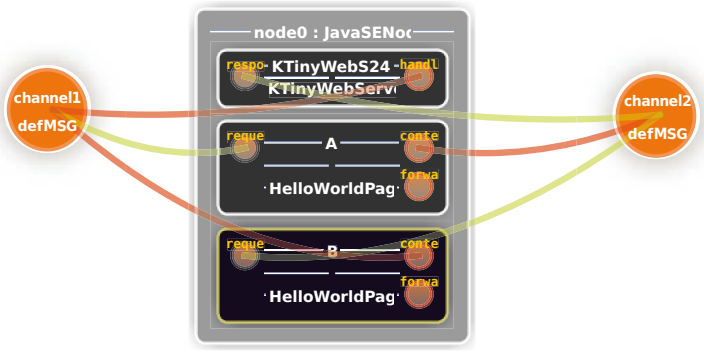


FIGURE 5.10 – Configuration de base du serveur web

C'est sur ce modèle qu'est défini le site de Kevoree [15].

Définition de la plate-forme Les types de nœuds de plate-forme restent les mêmes que dans le cas d'étude précédent. C'est le gestionnaire qui change pour prendre en compte la notion d'élasticité de l'application. Ce gestionnaire a pour objectif d'assurer

15. http://kevoree.org

la qualité de service nécessaire au serveur web. Pour cela, un nœud capable d'observer son exécution et notamment son usage de mémoire et de temps processeur est utilisé.

Ainsi, lorsque le gestionnaire détecte que la consommation de ressources d'un nœud devient critique, il est en charge de déplacer, ou dupliquer les composants de page web afin de décharger le nœud (voir listing 5.13). En plus de cela, il est nécessaire qu'il mette en place des canaux de communication spécifiques capables de distribuer les requêtes entre les composants. Pour cela, nous avons défini un *channel* de type *RoundRobin* qui permet de sélectionner, de manière aléatoire, le composant à qui la requête est soumise. La limitation du nombre de nœuds de plate-forme est actuellement fixée en tant que paramètre du gestionnaire mais ce paramètre pourrait être calculé dynamiquement en fonction du coût des nœuds de plate-forme et du coût que l'utilisateur de la plate-forme accepte de payer.

Là encore, l'implémentation de ce gestionnaire reste simple puisque nous ne surveillons que la charge du processeur et il serait nécessaire d'avoir une implantation plus complète pour être capable de gérer efficacement cette notion d'élasticité. Par exemple, il serait sans doute nécessaire d'observer le temps nécessaire entre l'envoi de la requête depuis le serveur vers les pages et la réponse d'une des pages vers le serveur web afin de pouvoir assurer une qualité de service plus précise.

5.3.4 Évaluation

Le fait d'avoir un support commun pour la communication entre les moteurs de raisonnement capables d'adapter les différents niveaux nous permet de concevoir de l'adaptation multi-niveaux.

Au travers de cette expérimentation, j'ai montré que notre abstraction permettait de faire de l'adaptation multi-niveaux que ce soit en tenant compte des informations des différents niveaux ou en faisant collaborer les moteurs de raisonnement.

Pour cela, il est bien entendu nécessaire d'utiliser notre abstraction pour la définition des reconfigurations. Grâce au langage *KevScript*, la définition de reconfiguration nécessite peu de ligne de code (173 lignes de code).

De plus, grâce à la séparation des préoccupations, un moteur de raisonnement se concentre seulement sur la définition des reconfigurations et n'a pas besoin de définir l'implantation de ces reconfigurations. Si nous souhaitons implanter le même cas d'étude avec par exemple FraSCAti sur Amazon EC2, nous sommes obligés d'implanter nous même les actions de reconfigurations en utilisant les APIs spécifiques d'Amazon alors que dans notre cas, celles-ci sont fournies par le type de nœud et grâce à l'utilisation de type abstrait, il est même envisageable d'utiliser ce moteur de raisonnement sur une autre Cloud équivalent sans devoir réimplémenter quoique ce soit.

5.4 Synthèse

Ce chapitre a montré que l'utilisation de l'abstraction proposée par KevoreeKloud permettait non seulement de définir des applications, des nœuds de plate-forme et des

nœuds d'infrastructure mais qu'elle permettait aussi de définir des gestionnaires spécialisés capables de tenir compte de l'ensemble des informations fournies par les différents types et instances de nœuds, composants, canaux de communications et groupes. Ce sont ces gestionnaires spécialisés qui permettent de définir les spécificités de la plate-forme ou de l'infrastructure correspondante et permettent une réutilisation des types de nœuds peu importe la solution de plate-forme ou d'infrastructure que nous souhaitons développer. De plus, nous avons montré que l'impact de notre abstraction sur la gestion d'une infrastructure ou d'une plate-forme est négligeable par rapport au temps nécessaire à la création ou à l'arrêt des machines virtuels ou dans notre cas d'étude d'espaces utilisateurs virtualisés. De même, nous avons montré que l'impact de l'utilisation mémoire de notre abstraction est acceptable même si elle peut être encore améliorée. Enfin nous avons montré que l'utilisation de notre abstraction permet de définir des systèmes d'adaptation capables d'utiliser les informations des différents niveaux pour effectuer de l'adaptation efficace. Ces systèmes d'adaptation sont aussi capable de collaborer grâce au modèle et ainsi assurer la cohérence de l'adaptation entre les niveaux ou encore définir des adaptations multi-niveaux.

La figure 5.11 reprend le comparatif des solutions étudiées dans le chapitre 2 et en y intégrant KevoreeKloud. Celle-ci contrairement aux autres permet donc de gérer l'ensemble des caractéristiques nécessaires pour la modélisation d'un Cloud et offre ainsi la possibilité de gérer l'adaptation comme une problématique transverse.

	Multi-niveaux	**Hébergement**	**Distribution**	**Hétérogénéité**	**Extensibilité**	**Réfléxivité**
Darwin	non	non	non	non	non	en partie
Fractal	non	non	non	non	oui	en partie
iPOJO	non	non	non	non	oui	en partie
FraSCAti	possible	non	non	non	oui	en partie
PauWare	non	non	non	non	oui	limité
Genie	non	non	non	non	oui	oui
RBAD	non	non	non	non	non	non
gMDE	en partie	non	oui	oui	oui	non
Neptune	en partie	en partie	non	en partie	non	non
CloudML	non	non	non	limité	non	non
ConPaaS	non	non	oui	oui	non	limité
GCM	en partie	oui	non	oui	oui	en partie
Kevoree Kloud	oui	oui	oui	oui	oui	oui

FIGURE 5.11 – Comparatif des différentes solutions

Listing 5.13 – Définition du gestionnaire d'élasticité

```
@ComponentType
@DictionaryType({
  @DictionaryAttribute(name = "nbNodes", defaultValue="10", optional=true},
  @DictionaryAttribute(name = "maxPercentageCPULoad", defaultValue="95", optional=true}
})
class ElasticPaaSManager extends AbstractComponentType implements ModeListener {
  // this method is periodically executed
  def cronTask() {
    val model = getModelService.getLastModel
    // get the number of PaaS node
    val paasNodes = model.getNodes.filter(n →KloudModelHelper.isPaaSNode(model, n.getName))
    // look at the CPU load and if the load is larger than maxPercentageCPULoad
    val overloadedPaaSNodes = detectOverHead(paasNodes,
        Integer.parseInt(getDictionary.get("maxPercentageCPULoad")))
    if (overloadedPaaSNodes.size > 0) {
      overloadedPaaSNodes.foreach {
        currentNode →
        val nbNodes: Int = model.getNodes.size
        val nodes: List[ContainerNode] = model.getNodes
        val nodeName = "node" + nbNodes + 1
        if (currentNode.getComponents.size > 1 && currentNode.getComponents.find(c →
            c.getTypeDefinition.getName == "WebServer").isDefined &&
          currentNode.getComponents.find(c →c.getTypeDefinition.getName == "AbstractPage").isDefined)
            {
        // if all the WebPages are hosted in the same node than the WebServer, we migrate the
            component to another node
          kengine.addVariable("nodeName", nodeName)
          kengine.addVariable("currentNodeName", nodes.get(0).getName)
          kengine append "addNode {nodeName} : PJavaSENode"
          currentNode.getComponents.filter(c →c.getTypeDefinition.getName == "AbstractPage").foreach {
            component →
            kengine.addVariable("componentName", component.getName)
            kengine append "moveComponent {componentName}@{currentNodeName} →{nodeName}"
          }
        } else if (currentNode.getComponents.size > 1 && currentNode.getComponents.find(c →
            c.getTypeDefinition.getName == "WebServer").isEmpty) {
        // if the WebPages are not hosted in the same node than the WebServer, we create a new node
            for one of the WebPages
          kengine.addVariable("nodeName", nodeName)
          kengine append "addNode {nodeName} : JavaSENode"
          kengine append "addComponent {componentName}@{nodeName} : HelloWorldPage"
          kengine.addVariable("componentName", currentNode.getComponents.filter(c →
              c.getTypeDefinition.getName == "AbstractPage").get(0).getName)
          kengine append "moveComponent {componentName}@{currentNodeName} →{nodeName}"
        } else if (currentNode.getComponents.size == 1 && currentNode.getComponents.find(c →
            c.getTypeDefinition.getName == "WebServer").isEmpty) {
        // If each WebPage is hosted in its own node, we duplicate the WebPage that is on the
            overloaded node
          kengine.addVariable("nodeName", nodeName)
          kengine append "addNode {nodeName} : JavaSENode"
          kengine append "addComponent {componentName}@{nodeName} : HelloWorldPage"
          nodes.find(n → n.getName == getNodeName) match {
            case None →
            case Some(node) →
              if (node.getComponents.size == 1) {
                // copy all component parameters on the new replica
                KloudModelHelper.cloneComponent(model, node.getComponents.get(0), nodeName)
              }
          }
        }
      }
    }
  }
}
```

Quatrième partie

Conclusion

Chapitre 6

Conclusion et Perspectives

6.1 Conclusion

Cette thèse s'est intéressé aux challenges liés au Cloud Computing et en particulier à la problématique de l'adaptation dans le Cloud. Nous avons pu voir dans le chapitre 1 que l'adaptation doit être une préoccupation transverse aux différents niveaux que représente un Cloud (Infrastructure, Plate-forme, Applications). Pour cela, il est nécessaire que les systèmes d'adaptation des différents niveaux soient capables de collaborer ou qu'un système d'adaptation soit capable de gérer l'ensemble des niveaux. C'est pourquoi il est nécessaire d'avoir une abstraction capable de représenter aussi bien l'infrastructure, la plate-forme ou les applications. Cette abstraction serait utilisée par les fournisseurs de services de Cloud. Ceux-ci pourrait ainsi définir de nouveaux systèmes d'adaptation capable d'envisager des adaptations multi-niveaux. L'adaptation multi-niveaux consiste à utiliser les informations fournies par l'ensemble des niveaux afin d'en adapter un en particulier ou d'utiliser les capacités d'adaptation offertes par les autres niveaux pour en adapter un en particulier. C'est aussi le fait de pouvoir envisager des adaptations ayant un impact sur plusieurs niveaux simultanément.

Dans le chapitre 2, nous avons présenté les caractéristiques que nous jugeons importantes pour une abstraction de Cloud. Ces caractéristiques concernent la représentation multi-niveaux, la notion d'hébergement, la notion de distribution, la notion d'hétérogénéïté, le besoin d'extensibilité et la notion de réflexivité désynchronisable. Nous avons constaté que les abstractions existantes que ce soit pour les applications, les plates-formes ou les infrastructures ne supportaient pas l'ensemble de ces caractéristiques et avons donc défini une nouvelle abstraction.

Cette thèse propose une abstraction fondée sur les concepts et outils de l'Ingénierie des modèles (IDM ou MDE pour Model Driven Engineering en anglais) et plus particulièrement sur les concepts de modèles à l'exécution (M@R pour Model at Runtime en anglais) qui propose d'utiliser le modèle servant d'abstraction comme support pour la définition des reconfigurations de celle-ci tout en utilisant les outils du MDE pour assurer la validité des reconfigurations proposées.

Fondés sur l'approche orientée composant appelée Kevoree, présentée dans le cha-

pitre 3, cette thèse en a présenté, dans le chapitre 4, une extension qui permet de représenter l'ensemble des différents niveaux que sont l'infrastructure, la plate-forme et les applications et les relations entre ces niveaux avec notamment la notion d'hébergement de nœuds nous permettant de modéliser l'hébergement de plates-formes sur des infrastructures. Grâce à l'extensibilité de Kevoree, nous avons aussi intégré des algorithmes de planification capables de tenir compte des capacités de reconfiguration offertes par une solution de Cloud. De plus ces algorithmes peuvent être étendus ou remplacés selon les besoins. L'ensemble de ces travaux permet ainsi de définir des infrastructures et des plates-formes à partir de définition de nœud modélisant les caractéristiques spécifiques du matériel ou de l'environnement d'exécution disponibles. Ce sont ensuite les gestionnaires de ces plates-formes et infrastructures représentant la phase de décision dans la boucle d'adaptation qui définissent comment sont prises en compte les reconfigurations effectuées par les autres niveaux. Ces gestionnaires ont aussi la charge de l'adaptation des différents niveaux.

Le chapitre 5 a permis de montrer l'intérêt et l'usage de cette abstraction au travers de trois cas d'étude. Le premier concernait l'implémentation d'un Cloud hybride afin de montrer l'intérêt de Kevoree, de son extension et du framework associé pour la conception de nouvelle solution de Cloud. Le second cas d'étude permettait d'observer l'impact en terme de temps de l'utilisation de notre abstraction dans le cadre de la reconfiguration d'un Cloud ainsi que l'impact sur l'utilisation mémoire. Enfin le dernier cas d'étude permettait de montrer comment notre abstraction facilite le partage d'une vision globale du Cloud entre les différents niveaux de celui-ci facilitant la coopération entre les systèmes d'adaptations afin d'effectuer de l'adaptation cohérente et efficace.

Ainsi, grâce à notre extension de Kevoree, les fournisseurs de services de Cloud ont la possibilité de concevoir de nouveaux systèmes d'adaptation. Ces systèmes d'adaptation peuvent, grâce à notre abstraction globale, tenir compte de l'ensemble des informations concernant les différents niveaux et sont capables de concevoir des reconfigurations multi-niveaux.

Ces systèmes d'adaptation tirent partie de la définition de solutions de Cloud explicitant leurs capacités de reconfiguration au travers de la définition des primitives d'adaptation et des phases de planification propres à chaque type de nœud. La définition de ces types de nœuds correspond au travail des concepteurs de services de Cloud et notamment pour les services d'infrastructure et de plate-forme.

6.2 Perspectives à court terme

6.2.1 Modèle de contexte

Kevoree fournit un modèle architectural du système servant de support à la définition de reconfiguration pour le système. Cependant la prise de décision concernant ces reconfigurations nécessite plus d'informations que celles fournies par le modèle architectural. Par exemple, afin de limiter la surcharge sur un nœud, il est nécessaire de connaître la charge dont il fait l'objet que ce soit en terme d'utilisation de processeur, de

mémoire vive ou de bande passante. Or ces informations ne sont pas représentées dans le modèle architectural. En effet, elles ne représentent pas la configuration du système. En plus d'un modèle architectural, il est donc nécessaire d'envisager l'utilisation d'un modèle de contexte permettant de connaître les évolutions du système autre que les évolutions de sa configuration.

Le modèle architectural proposé par Kevoree et l'utilisation des groupes permet de disséminer et synchroniser les configurations de l'ensemble des nœuds. Mais les informations de contexte n'ont pas la même vitesse d'évolution et donc pas les mêmes besoins en termes de dissémination et synchronisation que les informations définies dans le modèle architectural. Il serait donc intéressant de pouvoir définir de la même façon que pour le modèle architectural, des groupes chargés de disséminer et synchroniser les informations de contexte au sein de Kevoree.

6.2.2 Test d'applications distribuées

Nous avons proposé, dans notre validation, un cas d'étude permettant de distribuer dynamiquement l'exécution de tests unitaires sur un Cloud afin d'accélérer l'exécution de ceux-ci. Outre les tests unitaires, il est aussi envisageable d'effectuer du test de système distribué. En effet, les applications distribuées prenant une place de plus en plus importante, il est nécessaire de pouvoir les tester et notamment tester leurs caractéristiques en termes de scalabilité, de volatilité et de capacité d'adaptation en fonction de l'environnement d'exécution. La validation de ces caractéristiques est quelque chose de complexe du fait des nombreux prérequis comme le déploiement automatique du système, la simulation des déconnexions des nœuds du système, la définition de l'oracle de test, l'exécution de ces tests en termes de contrôlabilité et d'observabilité.

Il existe plusieurs travaux pour effectuer du test de système distribué [72, 111, 85, 58]. L'idée d'utiliser notre extension de Kevoree pour le test d'applications distribuées est de pouvoir déployer dynamiquement ces applications sur un Cloud en déployant la plate-forme nécessaire à l'exécution de l'application à tester.

6.2.3 Extension vers d'autres technologies et utilisation dans des projets de recherches

À partir de l'abstraction et du framework proposés dans cette thèse, une implantation d'infrastructure ainsi que deux implantations de plate-forme spécialisées ont été définies. Il serait intéressant de diversifier les implantations proposées notamment au niveau de l'infrastructure pour montrer que notre abstraction et notre framework facilite la portabilité des implantations que ce soit pour les plates-formes mais aussi pour les systèmes d'adaptation des infrastrucutres. Pour cela, il est envisagé de définir de nouvelles implantations avec par exemple l'utilisation de LXC [1] ainsi que Xen [2] pour la définition de nouvelles infrastructures. De plus, il est prévu d'étendre le framework proposé afin

1. http://lxc.sourceforge.net/
2. http://xen.org/

de fournir un ensemble plus fourni de types de haut niveau facilitant la portabilité des implantations de plate-formes ainsi que des systèmes d'adaptation d'infrastructure.

Cette définition de nouvelles implantations permettra une évaluation plus large que celles proposées dans cette thèse avec notamment l'usage de ces implantations et celles existantes dans la construction d'une solution de Cloud entre plusieurs laboratoires de recherche (INRIA-Rennes avec l'équipe Triskell, SNT-Luxembourg avec l'équipe Serval, l'université du Parana-Brésil avec l'équipe C3SL)

6.3 Perspectives à long terme

6.3.1 Gestion du "Big Data"

Outre les ressources pour l'exécution des applications, les Clouds s'orientent aussi vers le stockage et la manipulation de données. L'usage intensif de l'informatique dans nos activités quotidienne fond que la quantité de ces données est très importante [16]. Ces données n'ont un réel intérêt que si nous sommes capable de les utiliser, c'est-à-dire que nous devons être capable non seulement de les stocker mais aussi de les manipuler efficacement. C'est le domaine du "Big Data".

L'un des modèles de Big Data les plus connus est sans doute le Map/Reduce [59] popularisé par Google avec son intégration au sein de son moteur de recherche afin d'effectuer l'indexation du contenu disponible sur l'Internet. Le principe du modèle de *Map/Reduce* vient des opérations Map et Reduce proposées dans les langages fonctionnels tel que Lisp. La fonction *Map* se charge de diviser un problème en sous problèmes plus petits et dont la résolution est déléguée à d'autres nœuds. La fonction *Reduce* quant à elle est chargée de la collecte des résultats des sous problèmes et de leur agrégation afin de trouver une réponse au problème initial.

Il serait intéressant de voir comment l'usage de Kevoree pourrait faciliter la définition de système de Big Data. De manière générale, les travaux de cette thèse se sont principalement tournés vers la gestion des ressources de calcul et peu voir pas du tout vers la gestion des ressources de stockage même si l'abstraction proposée ne pose pas à priori de limitation à ce sujet. Si l'abstraction utilisée est capable de représenter les informations concernant le système de stockage, ces informations peuvent être prises en compte pour la distribution et la répartition des *workers* d'une exécution de *Map/Reduce* par exemple.

6.3.2 Sécurité et abstraction globale

L'une des questions qui revient souvent de la part des entreprises concerne la sécurité des solutions proposées et de l'accès aux données. Cette thèse n'a pas pour but de répondre à cette question mais pose tout de même une question de sécurité non plus du point de vue des entreprises utilisatrices mais des entreprises qui fournissent les solutions de Cloud. En effet, il existe de nombreuses solutions propriétaires dont l'implantation n'est pas connue. C'est le cas de celles d'Amazon, Google et Microsoft par exemple. L'utilisation de notre abstraction pourrait être vue comme une faille de sécurité étant

donné qu'elle peut offrir à n'importe qui les informations de n'importe quel éléments faisant partie du système.

Nous avons fourni une première solution à cette problématique de sécurité en laissant la possibilité de partitionner le modèle afin de limiter les informations partagées et d'utiliser des types abstraits pour masquer certaines informations. Cependant, cette solution n'est pas suffisante et nécessiterait une étude plus importante afin de proposer d'autres solutions.

Un autre point de sécurité que nous n'avons pas étudié concerne la dissémination des modèles ou des données au travers des groupes et canaux de communication. En effet, dans nos implantations, nous n'avons mis aucune notion de sécurité laissant ainsi la possibilité à n'importe qui d'accéder aux informations que peuvent fournir ces éléments.

Enfin, certains composants ou nœuds ont besoin de données telles que des noms d'utilisateurs ou des mots de passes. C'est par exemple le cas pour un *proxy* vers Amazon EC2 qui nécessite les données d'un utilisateur pour interagir avec la solution d'Amazon. L'une des solutions simples est de fournir ces informations dans le modèle mais cela implique que tous les éléments ayant accès aux modèles pourront avoir accès à ces informations. Il est donc nécessaire de réfléchir à comment fournir ces informations sans pour autant les offrir à l'ensemble des participants au système.

Publications liées à cette thèse

[1] Fouquet, François and Nain, Grégory and Morin, Brice and Daubert, Erwan and Barais, Olivier and Plouzeau, Noël and Jézéquel, Jean-Marc. *An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements.* MODELS, 2012.

[2] Fouquet, François and Daubert, Erwan and Plouzeau, Noël and Barais, Olivier and Bourcier, Johann and Jézéquel, Jean-Marc. *Dissemination of reconfiguration policies on mesh networks.* DAIS, 2012.

[3] Daubert, Erwan and André, Françoise and Barais, Olivier. *Adaptation multi-niveaux : l'infrastructure au service des applications.* CFSE, 2011.

[4] André, Françoise and Daubert, Erwan and Gauvrit, Guillaume. *Distribution and Self-Adaptation of a Framework for Dynamic Adaptation of Services.* ICIW, 2011.

[5] André, Françoise and Daubert, Erwan and Gauvrit, Guillaume. *Towards a Generic Context-Aware Framework for Self-Adaptation of Service-Oriented Architectures.* ICIW, 2010.

[6] Gauvrit, Guillaume and Daubert, Erwan and André, Françoise. *SAFDIS : A Framework to Bring Self-Adaptability to Service-Based Distributed Applications.* EUROMICRO, 2010.

[7] André, Françoise and Daubert, Erwan and Nain, Grégory and Morin, Brice and Barais, Olivier. *F4Plan : An Approach to build Efficient Adaptation Plans.* Mobiquitous, 2010.

Références

[1] Amazon EC2. http://aws.amazon.com/en/ec2 (last accessed on October, the 16th 2012).

[2] Amazon web services blog : Animoto - scaling through viral growth. http://aws.typepad.com/aws/2008/04/animoto---scali.html (last accessed on October, the 16th 2012).

[3] Cloud Bees. http://www.cloudbees.com/ (last accessed on October, the 16th 2012).

[4] CloudStack. http://www.cloudstack.org/ (last accessed on October, the 16th 2012).

[5] Contrail. http://contrail-project.eu/ (last accessed on October, the 16th 2012).

[6] deltaCloud. http://deltacloud.apache.org/ (last accessed on October, the 16th 2012).

[7] ElasticHosts. http://www.elastichosts.com/ (last accessed on October, the 16th 2012).

[8] ElasticStack. http://www.elasticstack.com/ (last accessed on October, the 16th 2012).

[9] Eucalyptus. http://www.eucalyptus.com/ (last accessed on October, the 16th 2012).

[10] Force.com. http://www.force.com/ (last accessed on October, the 16th 2012).

[11] Google and IBM Announced University Initiative to Address Internet-Scale Computing Challenges. http://www-03.ibm.com/press/us/en/pressrelease/22414.wss (last accessed on October, the 16th 2012).

[12] Google and I.B.M. Join in 'Cloud Computing' Research. http://www.nytimes.com/2007/10/08/technology/08cloud.html?r=1&ei=5088&en=92a8c77c354521ba&ex=1349582400&oref=slogin&partner=rssnyt&emc=rss&pagewanted=print (last accessed on October, the 16th 2012).

[13] Google app egine. https://developers.google.com/appengine (last accessed on October, the 16th 2012).

[14] Google Drive. https://drive.google.com (last accessed on October, the 16th 2012).

[15] Google's Green Computing : Efficiency at Scale. http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/fr//green/pdfs/google-green-computing.pdf (last accessed on October, the 16th 2012).

[16] IBM et sa vision Big Data. http://www-01.ibm.com/software/fr/data/bigdata/ (last accessed on October, the 16th 2012).

[17] IBM Introduces Ready-to-Use Cloud Computing. http://www-03.ibm.com/press/us/en/pressrelease/22613.wss (last accessed on October, the 16th 2012).

[18] IBM SOA Foundation. http://www-01.ibm.com/software/solutions/soa/offerings.html (last accessed on October, the 16th 2012).

[19] JClouds. http://www.jclouds.org (last accessed on October, the 16th 2012).

[20] Mendix. http://www.mendix.com/ (last accessed on October, the 16th 2012).

[21] Nimbus. http://www.nimbusproject.org/ (last accessed on October, the 16th 2012).

[22] Object Management Group's website. http://www.omg.org/ (last accessed on October, the 16th 2012).

[23] Office 365. http://www.microsoft.com/en-us/office365/ (last accessed on October, the 16th 2012).

[24] OMG's Model Driven Architecture. http://www.omg.org/mda/ (last accessed on October, the 16th 2012).

[25] Open cloud computing interface. http://www.occi-wg.org (last accessed on October, the 16th 2012).

[26] OpenNebula. http://opennebula.org/ (last accessed on October, the 16th 2012).

[27] OpenShift. https://openshift.redhat.com/app/ (last accessed on October, the 16th 2012).

[28] OpenStack. http://www.openstack.org/ (last accessed on October, the 16th 2012).

[29] Opscode Chef. http://www.opscode.com/chef/ (last accessed on October, the 16th 2012).

[30] Puppet. http://puppetlabs.com/puppet/what-is-puppet/ (last accessed on October, the 16th 2012).

[31] Rackspace. http://www.rackspace.com/ (last accessed on October, the 16th 2012).

[32] rPath. http://www.rpath.com (last accessed on October, the 16th 2012).

[33] Service Component Architecture (SCA). http://www.osoa.org/ (last accessed on October, the 16th 2012).

[34] Vagrant. http://vagrantup.com/ (last accessed on October, the 16th 2012).

[35] WaveMaker. http://wavemaker.com/ (last accessed on October, the 16th 2012).

[36] Windows Azure. http://www.windowsazure.com (last accessed on October, the 16th 2012).

[37] Zoho Creator. https://www.zoho.com/creator/ (last accessed on October, the 16th 2012).

[38] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Corrado Zoccolo, Francoise André, and Jérémy Buisson. An abstract schema modeling adaptivity management. *Integrated Research in GRID Computing*, pages 89–102, 2007.

[39] 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 (MobiQuitous)*, Sydney, Australia, December 2010. short paper.

[40] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4) :50–58, April 2010.

[41] N. Bencomo and G. Blair. Using architecture models to support the generation and operation of component-based adaptive systems. *Software engineering for self-adaptive systems*, pages 183–200, 2009.

[42] A. Beugnard, J.M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[43] Paul Beynon-Davies. *Database systems*. Citeseer, 2000.

[44] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42 :22–27, 2009.

[45] T. Bleizeffer, J. Calcaterra, D. Nair, R. Rendahl, B. Schmidt-Wesche, and P. Sohn. Description and application of core cloud user roles. In *Proceedings of the 5th ACM Symposium on Computer Human Interaction for Management of Information Technology*, page 2. ACM, 2011.

[46] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90 :281–300, 1995.

[47] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Professional, 2005.

[48] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, pages 133–147, 2002.

[49] C. Bunch, N. Chohan, C. Krintz, and K. Shams. Neptune : a domain specific language for deploying hpc software on cloud platforms. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 59–68. ACM, 2011.

[50] A. Carzaniga, A. Fuggetta, R.S. Hall, D. Heimbigner, A. Van Der Hoek, and A.L. Wolf. A characterization framework for software deployment technologies. Technical report, DTIC Document, 1998.

[51] 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.

[52] Yixin Chen, Chih wei Hsu, and Benjamin W. Wah. Sgplan : Subgoal partitioning and resolution in planning. In *In Edelkamp*, pages 30–32, 2004.

[53] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6) :377–387, 1970.

[54] Brad J Cox and Andrew Novobilski. *Object-Oriented Programming ; An Evolutionary Approach*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[55] I. Crnkovic, M. Chaudron, S. Sentilles, and A. Vulgarakis. A classification framework for component models. *Software Engineering Research and Practice in Sweden*, page 3, 2007.

[56] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5) :593–615, 2011.

[57] J. Cui, J. Xu, H. Lin, W. Li, and Z. Sun. A study of rapid business application development in the cloud. *Design, User Experience, and Usability. Theory, Methods, Tools and Practice*, pages 398–407, 2011.

[58] E.C. de Almeida, G. Sunyé, Y. Le Traon, and P. Valduriez. A framework for testing peer-to-peer systems. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 167–176. IEEE, 2008.

[59] J. Dean and S. Ghemawat. Mapreduce : simplified data processing on large clusters. *Communications of the ACM*, 51(1) :107–113, 2008.

[60] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado. Gridunit : software testing on the grid. In *Proceedings of the 28th international conference on Software engineering*, pages 779–782. ACM, 2006.

[61] B. Ensink and V. Adve. Coordinating adaptations in distributed systems. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 446–455. IEEE, 2004.

[62] C. Escoffier, R.S. Hall, and P. Lalanda. ipojo : An extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474–481. IEEE, 2007.

[63] Martin Fowler et Matthew Foemmel. Continuous Integration. http://martinfowler.com/articles/originalContinuousIntegration.html (last accessed on October, the 16th 2012).

[64] Eugen Feller, Louis Rilling, and Christine Morin. Snooze : A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds. In *12th*

IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2012), Ottawa, Canada, May 2012.

[65] Eugen Feller, Louis Rilling, Christine Morin, Renaud Lottiaux, and Daniel Leprince. Snooze : A Scalable, Fault-Tolerant and Distributed Consolidation Manager for Large-Scale Clusters. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 125–132. IEEE Computer, 2010.

[66] I. Foster and C. Kesselman. *The grid 2 : Blueprint for a new computing infrastructure*. Morgan Kaufmann, 2003.

[67] François Fouquet. *Kevoree : Model@Runtime pour le développement continu de systèmes adaptatifs distribués hétérogènes*. PhD thesis, Université de Rennes 1, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), 2013.

[68] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, and Jean-Marc Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *DAIS 2012*, Stockholm, Suède, June 2012.

[69] S. Garfinkel and H. Abelson. *Architects of the information society : 35 years of the Laboratory for Computer Science at MIT*. The MIT Press, 1999.

[70] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 33–38, New York, NY, USA, 2002. ACM.

[71] Malik Ghallab, Craig K. Isi, Scott Penberthy, David E. Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

[72] S. Ghosh, A.P. Mathur, et al. Issues in testing distributed component-based systems. In *First ICSE workshop on testing distributed component-based systems*, 1999.

[73] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[74] James Gosling and Henry McGilton. *The Java language environment*, volume 2550. Sun Microsystems Computer Company, 1995.

[75] T.L. Graves, M.J. Harrold, J.M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th international conference on Software engineering*, pages 188–197. IEEE Computer Society, 1998.

[76] R. Guerraoui and M.E. Fayad. Oo distributed programming is not distributed oo programming. *Communications of the ACM*, 42(4) :101–104, 1999.

[77] Guy Rosen. Anatomy of an Amazon EC2 Resource ID. http://www.jackofallclouds.com/2009/09/anatomy-of-an-amazon-ec2-resource-id/ (last accessed on October, the 16th 2012).

[78] 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.

[79] G. Hohpe and B. Woolf. *Enterprise integration patterns : Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[80] P. Horn. Autonomic computing : Ibm's perspective on the state of information technology. *Computing Systems*, 15(Jan) :1–40, 2001.

[81] Huan Liu. Amazon data center size. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/ (last accessed on October, the 16th 2012).

[82] G.M. Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, 2001.

[83] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes. Sky computing. *Internet Computing, IEEE*, 13(5) :43–51, 2009.

[84] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[85] A. Khoumsi. A temporal approach for testing distributed systems. *Software Engineering, IEEE Transactions on*, 28(11) :1085–1103, 2002.

[86] G. Kiczales, J. Des Rivieres, and D.G. Bobrow. *The art of the metaobject protocol*. The MIT press, 1991.

[87] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97—Object-Oriented Programming*, pages 220–242, 1997.

[88] J.M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 119–129. IEEE, 2002.

[89] L. Kleinrock. A vision for the Internet. *ST Journal for Research*, 2(1) :4–5, November 2005.

[90] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http ://planning.cs.uiuc.edu/.

[91] P. Maes. *Concepts and experiments in computational reflection*, volume 22. ACM, 1987.

[92] Elton Manias and Françoise Baude. A component-based middleware for hybrid grid/cloud computing platforms. *Concurrency and Computation : Practice and Experience*, 24(13) :1461–1477, 2012.

[93] D. Manset, H. Verjus, R. McClatchey, and F. Oquendo. A formal architecture-centric model-driven approach for the automatic generation of grid applications. *Arxiv preprint cs/0601118*, 2006.

[94] P. Marshall, K. Keahey, and T. Freeman. Improving utilization of infrastructure clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 205–214. IEEE, 2011.

[95] J. McAffer. Meta-level programming with coda. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 190–214. Springer, 1995.

[96] Medvidovic, Nenad and Taylor, Richard N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, January 2000.

[97] Rémi Mélisson, Philippe Merle, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. Reconfigurable run-time support for distributed service component architectures. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 171–172, New York, NY, USA, 2010. ACM.

[98] Philippe Merle, Romain Rouvoy, and Lionel Seinturier. A Reflective Platform for Highly Adaptive Multi-Cloud Systems. In *10th International Workshop on Adaptive and Reflective Middleware (ARM'2011) at the12th ACM/IFIP/USENIX International Middleware Conference*, pages 1–7, Lisbonne, Portugal, December 2011. http://hal.inria.fr/inria-00628643.

[99] B. Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability*. 2010.

[100] B. Morin, O. Barais, G. Nain, and J.M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009.

[101] D.S. Nau, T.C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, and F. Yaman. Shop2 : An htn planning system. *J. Artif. Intell. Res. (JAIR)*, 20 :379–404, 2003.

[102] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Incorporated, 2008.

[103] OSGi Alliance. http://www.osgi.org (last accessed on October, the 16th 2012).

[104] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40(11) :38–45, 2007.

[105] Guillaume Pierre, Ismail El Helw, Corina Stratan, Ana Oprescu, Thilo Kielmann, Thorsten Schütt, Matej Artač, and Aleş Černivec. Conpaas : an integrated runtime environment for elastic cloud applications. In *Proceedings of the Middleware conference*, December 2011.

[106] Stefan Ried, Holger Kisker, Pascal Matzke, Andrew Bartels, and Miroslaw Lisserman. Understanding and quantifying the future of cloud computing. Technical report, 04 2011. http://www.forrester.com/rb/Research/sizing_cloud/q/id/58161/t/2 (last accessed on October, the 16th 2012).

[107] P. Robertson, R. Laddaga, and H. Shrobe. Introduction : the first international workshop on self-adaptive software. *Self-Adaptive Software*, pages 1–10, 2001.

[108] F. Romeo, C. Ballagny, and F. Barbier. Pauware : a state-based component model. *Actes des Journées Composants (JC2006)*, pages 1–10, 2006.

[109] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10) :929–948, 2001.

[110] D.C. Schmidt. Guest editor's introduction : Model-driven engineering. *Computer*, 39(2) :25–31, 2006.

[111] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2) :129–157, 1994.

[112] C. Szyperski, D. Gruntz, and S. Murer. *Component software : beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[113] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby*, volume 13. Pragmatic Bookshelf, 2004.

[114] Guido Van Rossum. A tour of the python language. In *tools*, page 370. Published by the IEEE Computer Society, 1997.

[115] Werner Vogels. Beyond server consolidation. *Queue*, 6(1) :20–26, January 2008.

[116] M.A. Vouk. Cloud computing–issues, research and implementations. *Journal of Computing and Information Technology*, 16(4) :235–246, 2008.

[117] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

[118] B. Woolf and R. Johnson. The type object pattern. *Pattern Languages of Program Design*, 3, 1996.

Table des figures

Liste des algorithmes

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Adaptation et Cloud Computing: un besoin d'abstraction pour une gestion transverse

Nom Prénom de l'auteur : DAUBERT ERWAN

Membres du jury :
- Madame Baude Françoise
- Monsieur Danelutto Marco
- Monsieur De Palma Noël
- Monsieur Ledoux Thomas
- Monsieur Barais Olivier
- Monsieur PAZAT Jean-Louis

Président du jury : NOEL DE PALMA

Date de la soutenance : 24 Mai 2013

Reproduction de la these soutenue

☒ Thèse pouvant être reproduite en l'état
☐ Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 24 Mai 2013

Le Directeur,

M'hamed DRISSI

Signature du président de jury
Noel DE PALMA

Résumé

Le Cloud Computing est devenu l'un des grands paradigmes de l'informatique et propose de fournir les ressources informatiques sous forme de services accessibles au travers de l'Internet. Ces services sont généralement organisés selon trois types ou niveaux. On parle de modèle SPI pour “Software, Platform, Infrastructure” en anglais.

De la même façon que pour les applications ``standard'', les services de Cloud doivent être capables de s'adapter de manière autonome afin de tenir compte de l'évolution de leur environnement. À ce sujet, il existe de nombreux travaux tels que ceux concernant la consolidation de serveur et l'économie d'énergie. Mais ces travaux sont généralement spécifiques à l'un des niveaux et ne tiennent pas compte des autres. Pourtant, comme l'a affirmé Kephart *et al.* en 2000, même s'il existe des adaptations à priori indépendantes les unes des autres, celles-ci ont un impact sur l'ensemble du système informatique dans lequel elles sont appliquées. De ce fait, une adaptation au niveau infrastructure peut avoir un impact au niveau plate-forme ou au niveau application.

L'objectif de cette thèse est de fournir un support pour l'adaptation permettant de gérer celle-ci comme une problématique transverse au différents niveaux afin d'assurer la cohérence et l'efficacité de l'adaptation. Pour cela, nous proposons une abstraction capable de représenter l'ensemble des niveaux et servant de support pour la définition des reconfigurations. Cette abstraction repose sur les techniques de modèle à l'exécution (Model at Runtime en anglais) qui propose de porter les outils utilisés à la conception pour définir, valider et appliquer une nouvelle configuration pendant l'exécution du système lui-même. Afin de montrer l'utilisabilité de cette abstraction, nous présentons trois expérimentations permettant de montrer l'extensibilité et la généricité de notre solution, de montrer que l'impact sur les performances du système est faible, et de montrer que cette abstraction permet de faire de l'adaptation multi-niveaux.

Abstract

Cloud Computing is becoming the new paradigm for information technology to provide resources as Internet-based services. These services are basically categorized according to three layers also called SPI model (Software, Platform, Infrastructure).

The same way as ``non-Cloud'' applications, Cloud services must be able to adapt themselves according to the evolution of their environment. There are many works on dynamic adaptation such as server consolidation and green computing but these works are generally specifics to one layer and do not take the others into account. However Kephart *et al.* have explain in 2000 that even if adaptations are, in theory, independant, they have an impact on the overall system. Consequently, an adaptation at the infrastructure layer can have an impact at the platform or at the application layers.

This thesis provides an abstraction to manage adaptation as an orthogonal concern overs Cloud layers. Based on Model at Runtime (M@R) techniques which offer to use design tools to build and validate new configuration of the system at the runtime, this abstraction is able to modelize all the Cloud layers. To show the usability of this abstraction, we provide three experimentations showing the extensibility and genericity of our approach, showing that performance overhead on the system (infrastructure or platform) is weak and showing that the abstraction allows to build multi-layers adaptations.

N° d’ordre : **13ISAR 10 / D13-10**

Institut National des Sciences Appliquées de Rennes
20, Avenue des Buttes de Coësmes . CS 14315 . F-35043 Rennes Cedex
Tel : 02 23 23 82 00 – Fax : 02 23 23 83 96