HAL
open science

Déploiement d'applications patrimoniales en environnements de type informatique dans le nuage

Xavier Etchevers

To cite this version:

Xavier Etchevers. Déploiement d'applications patrimoniales en environnements de type informatique dans le nuage. Autre [cs.OH]. Université de Grenoble, 2012. Français. NNT : 2012GRENM100 . tel-00875568

HAL Id: tel-00875568

https://theses.hal.science/tel-00875568

Submitted on 22 Oct 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 aoûit 2006

Présentée par

Xavier ETCHEVERS

Thèse dirigée par **M. Noël DE PALMA**
et codirigée par **Mme Fabienne BOYER et M. Thierry COUPAYE**

préparée au sein **du département Data & Service API d'Orange Labs et du Laboratoire d'Informatique de Grenoble**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Déploiement d'applications patrimoniales en environnements de type informatique dans le nuage

Thèse soutenue publiquement le **12 décembre 2012**,
devant le jury composé de :

M. Frédéric DESPREZ
Directeur de Recherche à l'INRIA, Président
Mme Françoise BAUDE
Professeur à l'Université de Nice Sophia-Antipolis, Rapporteur
M. Daniel HAGIMONT
Professeur à l'Institut National Polytechnique de Toulouse, Rapporteur
M. Jean-Marc MENAUD
Maître Assistant à l'Ecole des Mines de Nantes, Examinateur
M. Noël DE PALMA
Professeur à l'Université Joseph Fourier, Directeur de thèse
Mme Fabienne BOYER
Maître de Conférences à l'Université Joseph Fourier, Co-Directeur de thèse
M. Thierry COUPAYE
Directeur de Recherche Cloud Platforms chez Orange Labs, Co-Directeur de thèse

Remerciements

Au terme de ces trois années de doctorat, j'adresse mes remerciements à l'ensemble des personnes qui ont contribué à l'aboutissement de ce travail.

En tout premier lieu, je souhaite remercier l'ensemble des membres du jury qui m'ont fait l'honneur de participer à ma soutenance. Merci à M. Frédéric Desprez, directeur de recherche à l'INRIA, qui en a assuré la présidence. Je remercie également Mme Françoise Baude, professeur à l'Université de Nice Sophia-Antipolis, et M. Daniel Hagimont, professeur à l'Institut National Polytechnique de Grenoble, pour avoir accepté d'être rapporteurs de mon travail et avoir apporté un jugement constructif. Merci également à M. Jean-Marc Menaud, maître assistant à l'Ecole des Mines de Nantes, pour avoir accepté d'être examinateur de cette thèse.

Je tiens également à remercier très chaleureusement mes trois encadrants de thèse pour la confiance qu'ils m'ont témoignée et pour le climat de sérénité qu'ils ont entretenu durant ces trois années. Merci à M. Noël de Palma, professeur à l'Université Joseph Fourier, à Mme Fabienne Boyer, maître de conférences à l'Université Joseph Fourier et à M. Thierry Coupaye, directeur de recherche chez Orange Labs.

J'adresse également de profonds remerciement à :

- Mme Maighread Gallagher, pour son aide précieuse sur les aspects bibliographiques ;
- M. Gwen Salaün, qui a largement contribué à la vérification des protocoles élaborés dans cette thèse ;
- MM. André Freyssinet et Nicolas Tachker, pour le support autour de la technologie AAA sur laquelle s'appuie l'implémentation proposée dans cette thèse ;
- MM. Alain Tchana et Brice Ekane, qui ont participé activement à l'interfaçage entre les systèmes *VAMP* et *TUNe* ;
- M. Alexandre Lefebvre, qui a œuvré avec persévérance au lancement de ce doctorat et qui a été un fidèle *supporter* de sa progression ;
- MM. Antonin Chazalet et Loïc Letondeur, pour leurs nombreux conseils méthodologiques toujours pertinents.

Je n'oublie pas toutes les personnes d'Orange Labs et de l'équipe Sardes de l'INRIA qui m'ont soutenu et aidé dans ce travail.

Enfin, un remerciement tout particulier pour ma famille : un immense merci à mon épouse pour avoir supporté le rythme trépidant et austère de ces trois années de thèse. Merci aussi à mes trois enfants qui ont toujours fait preuve d'une grande patience et de beaucoup de compréhension voire de compassion à l'égard d'un papa souvent trop facilement irritable. Leur humour et leur bonne humeur auront joué un rôle essentiel dans l'aboutissement rapide de ce travail.

Résumé

L'informatique dans le nuage a pour principal objectif de rationaliser les coûts d'acquisition, de mise en œuvre et d'exploitation des applications. Pour cela, une application est décomposée en un ensemble de ressources matérielles et logicielles virtualisées. De plus, elle fait l'objet d'une administration autonome visant à adapter son comportement aux changements dynamiques qui affectent son environnement d'exécution. Cette administration est qualifiée de gestion du cycle de vie de l'application (*Application Lifecycle Management* ou *ALM*). Dans le contexte de l'informatique dans le nuage, l'*ALM* est à l'origine d'un marché en pleine expansion et encore immature, qui voit se multiplier le nombre d'offres affichant d'importants gains de productivité.

Cependant, toutes ces solutions se heurtent à une limitation majeure : l'existence d'une dualité entre le niveau d'autonomie qu'elles proposent et le spectre des applications prises en charge. Fort de ce constat, ces travaux de doctorat ont visé à démontrer le caractère artificiel de cette dualité, en se focalisant sur les aspects relatifs à la gestion du déploiement initial des applications. Ainsi, les principales contributions de cette thèse sont regroupées au sein d'une plate-forme appelée VAMP (*Virtual Applications Management Platform*) capable d'assurer le déploiement autonome, générique et fiable de toute application patrimoniale répartie dans un environnement de type informatique dans le nuage. Il s'agit :

- d'un modèle à base de composants permettant de décrire les éléments qui constituent une application et leur projection sur l'infrastructure d'exécution ainsi que les dépendances qui les lient au sein de l'architecture applicative ;
- d'un protocole asynchrone, réparti et fiable d'auto-configuration et d'auto-activation de l'application ;
- de mécanismes visant à fiabiliser le système *VAMP* lui-même.

Outre l'implémentation de la solution elle-même, les parties critiques liées à la mise en œuvre de *VAMP* ont fait l'objet d'une vérification formelle. De plus, une phase de validation au travers de différents contextes applicatifs réels a visé à démontrer le caractère générique de la proposition.

Mots-clés : informatique dans le nuage, systèmes répartis, applications patrimoniales, déploiement, informatique autonome, modèle à composants.

Abstract

Cloud computing aims to cut down on the outlay and operational expenses involved in setting up and running applications. To do this, an application is split into a set of virtualized hardware and software resources. This virtualized application can be autonomously managed, making it responsive to the dynamic changes affecting its running environment. This is referred to as *Application Life-cycle Management* (*ALM*). In cloud computing, ALM is a growing but immature market, with many offers claiming to significantly improve productivity.

However, all these solutions are faced with a major restriction: the duality between the level of autonomy they offer and the type of applications they can handle. To address this, this thesis focuses on managing the initial deployment of an application to demonstrate that the duality is artificial. The main contributions of this work are presented in a platform named VAMP (*Virtual Applications Management Platform*). VAMP can deploy any legacy application distributed in the cloud, in an autonomous, generic and reliable way. It consists of:

- a component-based model to describe the elements making up an application and their projection on the running infrastructure, as well as the dependencies binding them in the applicative architecture;
- an asynchronous, distributed and reliable protocol for self-configuration and self-activation of the application;
- mechanisms ensuring the reliability of the VAMP system itself.

Beyond implementing the solution, the most critical aspects of running VAMP have been formally verified using model checking tools. A validation step was also used to demonstrate the genericity of the proposal through various real-life implementations.

Keywords: cloud computing, distributed systems, legacy applications, deployment, autonomic computing, component-based models.

Table des matières

1 Introduction **1**
1.1 Contexte . . . 2
1.1.1 Informatique répartie . . . 2
1.1.2 Informatique dans le nuage . . . 3
1.2 Motivations . . . 6
1.2.1 Caractéristiques des solutions existantes . . . 6
1.2.2 Caractérisation d'un environnement autonome d'*ALM* dans le nuage 7
1.3 Contributions . . . 8
1.4 Organisation du document . . . 10
1.4.1 Plan . . . 10
1.4.2 Charte graphique . . . 11
1.4.3 Exemple fil rouge . . . 11

I État de l'art **13**

2 Déploiement de systèmes répartis **15**
2.1 Concepts et caractérisation . . . 17
2.1.1 Concepts . . . 17
2.1.2 Caractérisation . . . 21
2.2 Solutions en environnements non virtualisés . . . 25
2.2.1 Solutions à base de langages dédiés à des domaines . . . 25
2.2.2 Solutions à base de composition de services . . . 29
2.2.3 Solutions à base de langages de description d'architecture . . . 38
2.3 Solutions en environnements virtualisés . . . 50
2.3.1 Solutions orientées infrastructure . . . 50
2.3.2 Solutions orientées service . . . 53
2.3.3 Solutions orientées application . . . 57
2.4 Conclusion . . . 69

II Contributions **73**

3 Vue d'ensemble de VAMP **77**
3.1 Contexte de l'informatique autonome . . . 78
3.2 Rappel des objectifs . . . 80
3.3 Principes de conception . . . 80
3.3.1 Modèle d'application distribuée . . . 80
3.3.2 Entités de contrôle . . . 89
3.3.3 Modèle de communication . . . 91
3.4 Architecture globale . . . 93
3.4.1 Portail . . . 93
3.4.2 Gestionnaire d'application . . . 94
3.5 Conclusion . . . 95

4 Génération d'appliances virtuelles **97**
4.1 Contexte . . . 98
4.1.1 Modélisation d'image virtuelle . . . 98
4.1.2 Génération d'image virtuelle . . . 99
4.2 Génération d'image dans VAMP . . . 101
4.2.1 Extension OVF . . . 102
4.2.2 Architecture du gestionnaire d'image . . . 105
4.3 Conclusion . . . 106

5 Protocole d'auto-configuration **109**
5.1 Contexte de l'environnement de communication AAA . . . 110
5.2 Instanciation du bus à messages . . . 112
5.2.1 Configuration initiale du bus à messages . . . 112
5.2.2 Mise à jour dynamique du bus à messages . . . 114
5.3 Protocole d'auto-configuration et d'auto-activation . . . 116
5.3.1 Description du protocole d'auto-configuration . . . 116
5.3.2 Vérification . . . 120
5.4 Conclusion . . . 121

6 Fiabilisation du déploiement **123**
6.1 Contexte de la tolérance aux pannes . . . 125
6.1.1 Détection de la panne . . . 126
6.1.2 Résolution de la panne . . . 127
6.2 Fiabilisation du déploiement des machines virtuelles applicatives . . . 130
6.2.1 Détection de pannes . . . 130
6.2.2 Fiabilisation par recouvrement . . . 131
6.3 Fiabilisation de VAMP . . . 134
6.3.1 Fiabilisation du portail . . . 134
6.3.2 Fiabilisation du gestionnaire d'application . . . 135
6.4 Conclusion . . . 142

III Expérimentations et résultats **143**

7 Evaluation **145**
7.1 Généricité de VAMP . . . 146
7.1.1 Springoo . . . 146
7.1.2 Clif . . . 148
7.1.3 CADP . . . 149
7.1.4 TUNe . . . 150
7.2 Performances et efficacité de VAMP . . . 151
7.2.1 Métriques . . . 151
7.2.2 Contexte technique . . . 152
7.2.3 Déploiements multiples . . . 153
7.2.4 Déploiement large échelle . . . 158
7.3 Conclusion . . . 160

8 Conclusion **163**
8.1 Rappel des motivations . . . 163
8.2 Rappel des contributions . . . 164
8.3 Perspectives . . . 165
8.3.1 Outillage . . . 165
8.3.2 Prise en charge d'autres phases du cycle de vie de l'application . . . 166
8.3.3 Extension de la fiabilité . . . 166
8.3.4 Support multi-nuages . . . 167
8.3.5 Dimensionnement autonome . . . 168

Bibliographie **169**

Table des figures

1.1 Déploiement d'une application répartie en environnement de type informatique dans le nuage 9
1.2 Architecture logique de l'application Springoo 12

2.1 Activités de la phase de déploiement d'une application 22

3.1 Structure d'une boucle autonomique (extrait de [81]) 79
3.2 Modèle de données de VAMP 82
3.3 Exemple de modélisation de l'architecture applicative de Springoo 83
3.4 Répartition des entités de contrôle de l'application Springoo 91
3.5 Architecture fonctionnelle de VAMP 93
3.6 Répartition des entités d'administration de VAMP 96

4.1 Architecture fonctionnelle du gestionnaire d'image 105
4.2 Algorithme mis en œuvre par le processus automatisé de génération d'images virtuelles 107

5.1 Diagramme de séquence d'ajout d'un serveur d'agents dans le bus à messages 114
5.2 Vue abstraite du flux d'exécution interne d'un configurateur 117
5.3 Illustration de l'exécution du protocole d'auto-configuration sur un exemple simple 119

6.1 Principe de duplication passive 128
6.2 Principe de duplication active 128
6.3 Principe de duplication semi-active 129
6.4 Principe de duplication coordinateur-cohorte 129
6.5 Algorithme de remplacement d'une machine applicative défectueuse . . . 132
6.6 Prise en compte de la fiabilisation des machines virtuelles applicatives dans le flux d'exécution interne d'un configurateur 134
6.7 Vue d'ensemble des techniques de fiabilisation des entités d'administration de VAMP 135
6.8 Algorithme de gestion dynamique de l'anneau 139

7.1 Mise en œuvre de Springoo à l'aide de VAMP 147

7.2 Mise en œuvre d'une instance Clif à l'aide de VAMP 149
7.3 Mise en œuvre d'une instance CADP à l'aide de VAMP 150
7.4 Mise en œuvre d'une instance TUNe à l'aide de VAMP 151
7.5 Métriques retenues pour évaluer le mécanisme de déploiement proposé par VAMP 152
7.6 Qualification du surcoût introduit par VAMP dans le cadre de déploiements multiples simultanés 155
7.7 Ressenti utilisateur dans la mise en œuvre du déploiement d'applications au moyen de VAMP 156
7.8 Estimation du degré de parallélisme introduit par le processus de déploiement de VAMP 157
7.9 Apports des mécanismes de cache d'images et de pré-approvisionnement de machines virtuelles 157
7.10 Comportement de VAMP lors du déploiement d'applications de grande taille 159
7.11 Bénéfice introduit dans VAMP par une approche *multithreads* 160

Liste des tableaux

2.1 Synthèse comparative des solutions de déploiement d'applications 71

6.1 Etats d'un duplicat participant à l'anneau 138

7.1 Caractérisation des liaisons entre les *wrappers* utilisés pour modéliser l'application Springoo 148

Listings

2.1 Exemple de ressources déclarées à l'aide du *DSL* proposé dans *Puppet* . . 26
2.2 Exemple de spécification d'un ensemble de composants à l'aide du formalisme proposé dans *FraSCAti* 33
2.3 Exemple de métadonnées contenues dans un *bundle OSGi* 35
2.4 Exemple de descripteur de configuration *SmartFrog* 42
2.5 Exemple de description d'application selon le formalisme proposé par [47] 58
2.6 Extrait d'un descripteur de configuration d'une machine virtuelle selon le *DSL* proposé par *MetaConfig* 62
3.1 Modèle d'architecture de Springoo à l'aide de Fractal ADL 84
3.2 Extrait de la description OVF étendu de l'application Springoo 88
4.1 Extrait du modèle d'image du tiers de présentation de Springoo 104
5.1 Extrait de la description statique de la topologie du bus à messages AAA 111
5.2 Configuration statique pour l'initialisation du bus à messages associé à une application 113

Chapitre 1

Introduction

Sommaire

1.1 Contexte **2**
1.1.1 Informatique répartie 2
1.1.2 Informatique dans le nuage 3
1.2 Motivations **6**
1.2.1 Caractéristiques des solutions existantes 6
1.2.2 Caractérisation d'un environnement autonome d'*ALM* dans le nuage 7
1.3 Contributions **8**
1.4 Organisation du document **10**
1.4.1 Plan 10
1.4.2 Charte graphique 11
1.4.3 Exemple fil rouge 11

Les travaux présentés dans cette thèse s'inscrivent dans le contexte de la gestion du cycle de vie des applications (*application life-cycle management* ou *ALM* en anglais) dans des environnements virtualisés de type informatique dans le nuage (*cloud computing*). Plus précisément, ils se focalisent sur la problématique du déploiement autonome et fiable d'applications réparties dans le nuage, indépendamment des domaines fonctionnels ou métier qu'elles adressent ou des choix de conception et de réalisation qui leur sont associés. Ce chapitre s'organise en quatre sections. La section 1.1 présente le contexte dans lequel s'inscrit cette thèse. Plus précisément, elle introduit les concepts d'informatique répartie et d'informatique dans le nuage et propose une définition des principes et des notions manipulés. La section 1.2 décrit les motivations qui sont à l'origine de ces travaux. Pour cela, elle s'appuie sur la mise en évidence des limitations des solutions actuelles. Par la suite, la section 1.3 offre une vue d'ensemble des contributions issues de ce doctorat. Enfin, la section 1.4 conclut ce chapitre en précisant l'organisation

du document ainsi qu'en décrivant les éléments d'aide à sa compréhension (e.g. charte graphique et application fil rouge).

1.1 Contexte

Cette section présente le contexte technique dans lequel s'inscrivent ces travaux de thèse, à savoir l'*informatique répartie* (section 1.1.1) et l'*informatique dans le nuage* (section 1.1.2).

1.1.1 Informatique répartie

Depuis l'apparition de l'informatique au milieu du XXième siècle, la performance des matériels tels que les unités de traitement ou de stockage ainsi que les équipements réseaux, n'a cessé de croître de façon continue et exponentielle. Gordon Moore, cofondateur de la société Intel®, a décrit cette évolution au moyen d'une loi [94] toujours valable aujourd'hui et selon laquelle, à coût constant, la puissance des ordinateurs double tous les dix-huit mois. Cette tendance est à l'origine du développement d'applications toujours plus sophistiquées, capables de répondre à de nouveaux besoins utilisateurs et aux exigences qui leurs sont associées en termes de qualité de service, de disponibilité, de fiabilité, de performance et de productivité.

De telles applications se composent d'un ensemble de *processus* logiciels qui s'exécutent de façon concurrente et qui interagissent entre eux pour fournir des services. Un processus est une instance d'exécution d'un programme au sein d'une machine. Il est caractérisé par un ensemble d'attributs de configuration. Désormais, l'ensemble des processus d'une application sont généralement localisés sur un ensemble de machines. On parle alors d'*application répartie*. Les interactions entre processus d'une application induisent qu'ils communiquent au travers de *canaux de communication*, basés sur une infrastructure réseau matérielle. Chaque programme est inclus dans une ou plusieurs *unité(s) de déploiement* appelée(s) *élément(s) logiciel(s)*. La coopération entre les processus d'une application repose sur un ensemble de règles ou *contraintes*, définies de façon globale à l'application. Il existe des contraintes permettant de caractériser chaque étape du cycle de vie de l'application (e.g. contraintes de configuration, de placement, d'ordre d'activation). L'*état d'un processus* correspond à l'ensemble des valeurs des attributs de configuration qui caractérisent le processus. L'*état d'une application* correspond à l'agrégation des états des processus qui la constituent.

Une *application répartie* se caractérise par :

- la répartition des entités qui la composent, au sein de plusieurs machines ou *serveurs*, potentiellement éloignées géographiquement ;
- l'hétérogénéité des serveurs, en termes de système d'exploitation et d'architecture matérielle, sur lesquels sont déployés les différents éléments logiciels qui composent l'application ;
- la prise en compte dynamique de changements dans son environnement ;

- la capacité à inter-opérer avec d'autres applications.

De ces caractéristiques découlent un certain nombre de difficultés :

Coûts d'administration : d'une part, l'exploitation et l'administration de ces applications sont devenues des tâches critiques en termes de temps et de coûts. En effet, elles nécessitent généralement l'intervention d'administrateurs dotés à la fois d'une connaissance du matériel et du middleware mais également de l'application elle-même afin de pouvoir la configurer correctement. Ainsi, selon un grand nombre d'études [60] [62] [93] [83], les coûts induits par les opérations de maintenance et d'évolution du système d'information des entreprises, représentent de 75% à plus de 90% du budget total qu'elles consacrent aux technologies de l'information.

Faible autonomie : d'autre part, le manque d'autonomie dans l'administration de ce type de solution entrave leur adaptation à des changements environnementaux. Ainsi, dans de nombreux cas, les phases d'installation ou de mise à jour de l'application nécessitent le déplacement physique d'un expert auprès des machines (éventuellement éloignées géographiquement) utilisées pour le déploiement de l'application.

Isolation contraignante : enfin, la prise en compte des contraintes de fiabilité, de sécurité et de qualité de service, nécessite que chaque application soit déployée sur un ensemble de machines qui lui est dédié, entraînant une possible sous utilisation des ressources matérielles. A titre d'illustration, à l'échelle de la planète, le taux moyen d'utilisation d'un serveur au sein d'une entreprise est de 20%. En outre, selon une enquête réalisée en septembre 2009 par *Kelton Research* pour le compte de l'entreprise *1E* et du groupement *Alliance to Save Energy* auprès de 100 administrateurs système d'entreprises de plus de 10.000 employés, un serveur sur six dans le monde est purement et simplement inutilisé.

1.1.2 Informatique dans le nuage

L'ensemble de ces difficultés est à l'origine du développement d'un nouveau paradigme appelé informatique dans le nuage (*cloud computing* en anglais). Selon les définitions les plus communément admises, à savoir celle proposée par [125] qui procède à la synthèse d'une dizaine de définitions existantes, celle issue du *National Institute of Standard and Technologies* (*NIST*) [90] et plus récemment celle élaborée dans le cadre du FP7 de l'Union Européenne [80], l'informatique dans le nuage consiste en la constitution de grands ensembles de ressources virtualisées, facilement utilisables et accessibles. Ces ressources peuvent être reconfigurées dynamiquement afin de s'ajuster à un changement survenu dans l'environnement d'exécution, offrant une utilisation optimale des ressources. Ces ensembles de ressources sont typiquement exploités selon un modèle de type paiement à l'usage dans lequel des garanties sont offertes au moyen de contrats de niveaux de service (*Service Level Agreement* ou *SLA* en anglais) personnalisables.

1.1.2.1 Organisation en couche

Ainsi, l'informatique dans le nuage correspond à la mise en relation d'un fournisseur et de clients (ou utilisateurs). Le fournisseur met à disposition un grand ensemble de ressources qui peuvent être regroupées selon leur nature, correspondant à trois niveaux d'abstraction :

Le niveau *Infrastructure as a Service* (*IaaS*) : il constitue le niveau le plus élémentaire. Il regroupe des ressources de type éléments matériels (e.g. entités de traitement, de stockage ou de réseau). Plus précisément, le niveau *IaaS* offre des ressources matérielles virtualisées s'appuyant sur un ensemble de ressources matérielles physiques. Parmi les principales solutions d'*IaaS* se trouvent *Amazon EC2/S3* [15][14] et *VMWare vCloud Director* [84] pour les offres commerciales et *Rackspace OpenStack* [106] et Open Nebula [8] dans le domaine du logiciel libre.

Le niveau *Software as a Service* (*SaaS*) : il s'agit du niveau de plus forte abstraction. Il comprend les éléments logiciels correspondant à des services applicatifs à destination d'utilisateurs finaux. Parmi les acteurs phares du *SaaS* se trouvent notamment les fournisseurs de solutions de réseaux sociaux, de messagerie, de bureautique en ligne [75] ou de gestion de la relation client (*Customer Relationship Management* ou *CRM* en anglais) [110].

Le niveau *Platform as a Service* (*PaaS*) : il constitue le niveau intermédiaire entre l'*IaaS* et le *SaaS*. Il regroupe les éléments logiciels tels que des plates-formes de développement ou d'exécution, qui constituent des outils d'aide à la mise en œuvre de services de la couche *SaaS*. Parmi les acteurs principaux de ce niveau se trouvent Google App Engine [74], Amazon Bean Talk [13], Salesforce.com [110], Microsoft Azure [92], ...

1.1.2.2 Virtualisation de ressource

Au niveau *IaaS*, les ressources mises à disposition sont dites *virtualisées*. Une ressource virtualisée désigne un artefact logiciel dont la finalité est la dématérialisation d'une ressource physique (ou réelle). Pour cela, elle en reproduit les comportements et les caractéristiques à l'identique. Ainsi, de manière comparable à une machine physique, une machine virtuelle est caractérisée par le nombre de processeurs, la quantité de mémoire, les unités de stockage et les interfaces réseaux dont elle dispose. En outre, elle fonctionne de façon identique à n'importe quelle machine réelle. La virtualisation des ressources physiques ne se limite pas aux serveurs et concerne également les entités de stockage et les équipements réseaux.

L'un des principaux intérêts de la virtualisation est de permettre la consolidation des ressources matérielles par mutualisation. Cela consiste à mettre en œuvre simultanément un ensemble de ressources matérielles virtualisées au niveau d'une infrastructure matérielle physique commune (e.g. plusieurs machines virtuelles s'exécutant sur une même machine physique). La faculté de consolidation offerte par la technique de virtualisation

s'accompagne de la garantie d'une parfaite isolation en termes de sécurité, de fiabilité et de qualité de service entre les éléments virtuels colocalisés sur un même équipement physique.

Une application répartie sera dite *virtualisable* si les éléments logiciels qui la composent peuvent être répartis au sein d'un ensemble d'*images* (ou *images virtuelles*). Chaque image peut être instanciée sous forme d'une machine virtuelle au sein d'une plate-forme d'*IaaS*. Une image est une pile logicielle complète et autonome constituée d'un système d'exploitation, d'un ensemble d'intergiciels ainsi que des binaires et des données spécifiques nécessaires à l'instanciation des éléments logiciels applicatifs qui s'exécutent sur la machine virtuelle. L'ensemble des images d'une application est appelé *appliance virtuelle* (ou *appliance*).

1.1.2.3 Intérêts de l'informatique dans les nuages

Les ressources mises à disposition par le fournisseur d'informatique dans le nuage sont accessibles au travers d'interfaces programmatiques via un réseau informatique. A tout instant, l'utilisateur peut s'adresser au fournisseur de ressources pour qu'une nouvelle ressource lui soit allouée ou pour relâcher une ressource dont il n'a plus l'utilité. L'allocation et la libération sont des opérations qui, à défaut d'être instantanées, sont réalisées dans un laps de temps relativement court.

Du point de vue de l'utilisateur, l'allocation dynamique des ressources, associée au modèle de paiement à l'usage, permet de diminuer le gaspillage de ressources inutilisées. En outre, elle offre une grande réactivité vis-à-vis des changements qui affectent l'environnement d'exécution. C'est par exemple le cas des variations de charge, celles-ci pouvant être connues (e.g. Orange réalise un quart de ses ventes en ligne annuelles sur le seul mois de décembre) ou totalement imprévisibles (e.g. Animoto qui, lors du lancement d'un service via Facebook, avait prévu 50 serveurs pour faire face aux requêtes de ses prospects, et qui a vu ce nombre doubler toutes les 12 heures pour atteindre 3500 trois jours plus tard [11]). Enfin, l'utilisateur d'une solution d'informatique dans le nuage peut se recentrer sur son cœur de métier, dans la mesure où il délègue la gestion d'une partie de son système d'information au fournisseur.

Du point de vue du fournisseur d'informatique dans le nuage, le principal intérêt réside dans le facteur d'échelle qu'il introduit en créant de grands ensembles de ressources accessibles au travers d'un réseau informatique. Comme l'illustre [20], il peut ainsi regrouper géographiquement ces ressources en fonction des coûts liés à leur hébergement (i.e. loyers), leur propre consommation énergétique ou celles des systèmes de refroidissement qu'elles induisent. En outre, le recours à la virtualisation permet de consolider le nombre de serveurs physiques nécessaires au fonctionnement d'un ensemble d'applications. La consommation de ces infrastructures peut encore être améliorée au moyen de comportements autonomiques visant à optimiser le placement des machines virtuelles afin de mettre en veille certaines machines physiques. L'action conjuguée des techniques d'informatique autonomique et de la virtualisation améliore, selon le contexte étudié, la productivité d'un administrateur système d'un facteur 5 à 50.

1.2 Motivations

Cette section esquisse un rapide état des lieux concernant les solutions existantes en matière d'ALM dans le nuage (section 1.2.1) puis définit les propriétés nécessaires à la définition d'une solution polyvalente, autonome et fiable (section 1.2.2).

1.2.1 Caractéristiques des solutions existantes

Contrairement aux niveaux *IaaS* et *SaaS* dont le rôle est parfaitement défini (i.e. mise à disposition, respectivement, de ressources matérielles virtualisées et de services logiciels applicatifs), les contours du niveau *PaaS* demeurent beaucoup plus vagues. Selon [90], cette couche tend à proposer des modèles et des outils permettant d'administrer automatiquement l'ensemble du cycle de vie des applications déployées dans le nuage. Un tel cycle de vie comprend des phases relatives à la construction de l'application (e.g. conception, développement, qualification) et des phases en lien avec son exécution (e.g. déploiement, gestion de la fiabilité, de la charge, de la sécurité, ...)[1]. Le caractère très général voire imprécis de la définition du *PaaS* a entraîné la fragmentation de ce marché à forte valeur ajoutée, sur lequel un grand nombre d'acteurs industriels essaient désormais d'entrer.

Qu'il s'agisse de fournisseurs d'*IaaS*, comme Amazon avec Elastic Beans Talk [13] ou VMWare Cloud Foundry [1], qui souhaitent proposer à leurs clients des offres de plus haut niveau permettant de valoriser leur solution de mise à disposition de ressources matérielles virtualisées, d'acteurs du *SaaS* comme Salesforce.com [110], qui veulent permettre à leurs utilisateurs de personnaliser leurs propres services métiers, ou de nouveaux entrants, qui prennent conscience des enjeux et répercussions économiques liés au marché de l'informatique dans le nuage, chacun d'eux se fixe comme priorité de conquérir, le plus rapidement possible, de nouvelles parts de ce marché encore très ouvert.

Il en résulte une profusion d'offres disparates vis-à-vis du spectre technique et/ou fonctionnel couvert, du domaine métier adressé voire de leur maturité, notamment en termes d'autonomie. Il existe même une dualité entre, d'une part, l'étendue du domaine métier ou technique couvert et, d'autre part, le degré de maturité des fonctionnalités. Ainsi, certaines solutions se focalisent sur un nombre important de fonctionnalités appliquées à un domaine métier ou technologique réduit (c'est le cas de [110] dont les services sont dédiés à la gestion de la relation cliente) alors que d'autres favorisent l'étendue du domaine technique ou métier couvert, au détriment des fonctionnalités proposées (e.g. [92] se confine à l'automatisation d'un sous-ensemble limité des opérations d'administration de n'importe quel type d'application basée sur la technologie Microsoft). Entre ces deux démarches extrêmes, il existe une multitude d'approches intermédiaires qui tentent de concilier ces deux contraintes (e.g. [13], [74] ou [1] proposent des mécanismes de déploiement et de gestion de l'élasticité d'applications web JEE d'entreprise). Cependant aucune des approches actuelles ne parvient à briser cette dualité en rendant autonome l'administration de n'importe quelle application patrimoniale virtualisable.

[1] La suite de ce chapitre se cantonne aux systèmes d'administration dédiés aux phases en lien avec l'exécution de l'application et n'aborde pas les aspects relatifs à sa construction.

1.2.2 Caractérisation d'un environnement autonome d'*ALM* dans le nuage

L'existence de cette dualité entre le niveau d'autonomie et la variété des applications gérées par les solutions de *PaaS* actuelles, souligne la nécessité d'un environnement capable d'y remédier. Un tel système doit notamment présenter les caractéristiques détaillées dans cette section.

1.2.2.1 Autonomie

L'environnement doit gérer, de manière autonome, c'est-à-dire sans qu'aucune intervention humaine ne soit nécessaire pour assurer son bon fonctionnement, l'ensemble des étapes du cycle de vie de l'application. Celles-ci comprennent notamment les phases de déploiement (i.e. le packaging, l'installation, la post-configuration et l'activation) d'exécution (i.e. la reconfiguration, l'arrêt et le redémarrage, la prise en charge de l'élasticité, de la sécurité, de la fiabilité et enfin la désinstallation).

1.2.2.2 Généricité

L'environnement doit être en mesure d'administrer n'importe quelle application répartie virtualisable. En d'autres termes, les fonctionnalités qu'il offre doivent pouvoir être appliquées à toute application patrimoniale, indépendamment du langage, du modèle ou des conventions de programmation qui lui sont associées, ainsi que de son environnement d'exécution ou de son domaine métier (e.g. format des images des machines virtuelles).

1.2.2.3 Indépendance vis-à-vis du *IaaS*

En tant que plate-forme de niveau *PaaS*, l'environnement doit être agnostique à l'égard de la solution (voire des solutions, dans le cas d'approches multi-nuages) d'*IaaS* avec laquelle il interagit pour assurer l'instanciation des machines virtuelles applicatives.

1.2.2.4 Passage à l'échelle

Il s'agit de la faculté à gérer simultanément un nombre important de machines virtuelles applicatives, qu'elles fassent partie d'applications fortement interconnectées et largement réparties (i.e. de type pair à pair) ou, dans un contexte plus industriel, d'une multitude d'applications clientes (e.g. déploiements multiples en parallèle).

1.2.2.5 Fiabilité

Il s'agit de la garantie, qu'en présence d'un nombre fini de pannes franches affectant l'environnement d'exécution de l'une des applications administrées ou le système lui-même, toute opération de gestion parvienne à son terme. Cette propriété s'avère encore plus fondamentale dans un contexte de type informatique dans le nuage que dans un contexte non virtualisé pour deux raisons. D'une part, la consolidation et la mutualisation des ressources physiques augmente sensiblement leur criticité en termes de disponibilité.

Ainsi, une défaillance d'un serveur physique affecte potentiellement plusieurs machines virtuelles applicatives. D'autre part, l'augmentation du nombre d'applications clientes administrées par un seul et même administrateur a pour conséquence qu'il consacre moins de temps à chacune d'elles.

1.3 Contributions

Fort de ce constat et en tant que première étape dans la définition d'un tel environnement, les travaux présentés dans cette thèse se sont focalisés sur les étapes d'*ALM* relatives au déploiement de l'application. Il s'agit du packaging, de la pré-configuration, de la publication, de l'installation, de la post-configuration et de l'activation. Ces travaux ont abouti à la définition, l'implémentation et l'évaluation d'une plate-forme baptisée *VAMP* [50], pour *Virtual Applications Management Platform*. La finalité de *VAMP* est d'assurer le déploiement automatique et fiable de n'importe quelle application patrimoniale virtualisable dans le nuage.

Plus précisément cette contribution s'organise autour :

1. d'un formalisme offrant la capacité de modéliser l'architecture d'applications virtualisées, à l'aide de composants. Chaque composant réifie un ensemble d'éléments logiciels applicatifs de sorte qu'il participe à la fourniture d'une abstraction uniforme des opérations de configuration de ces éléments logiciels. Cette abstraction commune est à l'origine de la polyvalence de VAMP. Ouvert et extensible, le formalisme proposé s'appuie sur les standards OVF [59] et Fractal ADL [102], permettant ainsi à un utilisateur de capturer les informations nécessaires au déploiement autonome de son application.

2. d'un environnement assurant l'interprétation des données issues du modèle de l'application, afin de réaliser l'auto-déploiement des applications réparties. Comme l'illustre la figure 1.1, cela passe par :

   (a) la génération dynamique des images virtuelles associées à l'application ;

   (b) leur publication et leur instanciation sous forme de machines virtuelles au sein d'une plate-forme d'*IaaS*. Ces opérations sont regroupées au sein de connecteurs dédiés et interchangeables, de sorte que VAMP est indépendant de l'infrastructure utilisée ;

   (c) la mise en œuvre d'un protocole distribué, asynchrone et fiable de post-configuration et d'activation des éléments logiciels. Des travaux dans le domaine de la vérification formelle ont par ailleurs permis d'attester de la correction de ce protocole. Grâce au caractère distribué et asynchrone du protocole, combiné à des techniques de répartition de charge, *VAMP* est en mesure de faire face aux enjeux relatifs au passage à l'échelle.

3. de mécanismes de fiabilisation de l'ensemble du processus de déploiement. Ces mécanismes regroupent des aspects relatifs à la tolérance aux pannes des machines

virtuelles applicatives ainsi que d'autres ayant trait à la fiabilisation du système VAMP lui-même. Certains de ces mécanismes ont également fait l'objet d'une vérification formelle afin de s'assurer de leur correction.

4. de l'évaluation de *VAMP* d'un point de vue qualitatif (i.e. le type d'applications déployées) mais également quantitatif (e.g. performance, surcoût, degré de parallélisation, tolérance aux pannes, etc.).

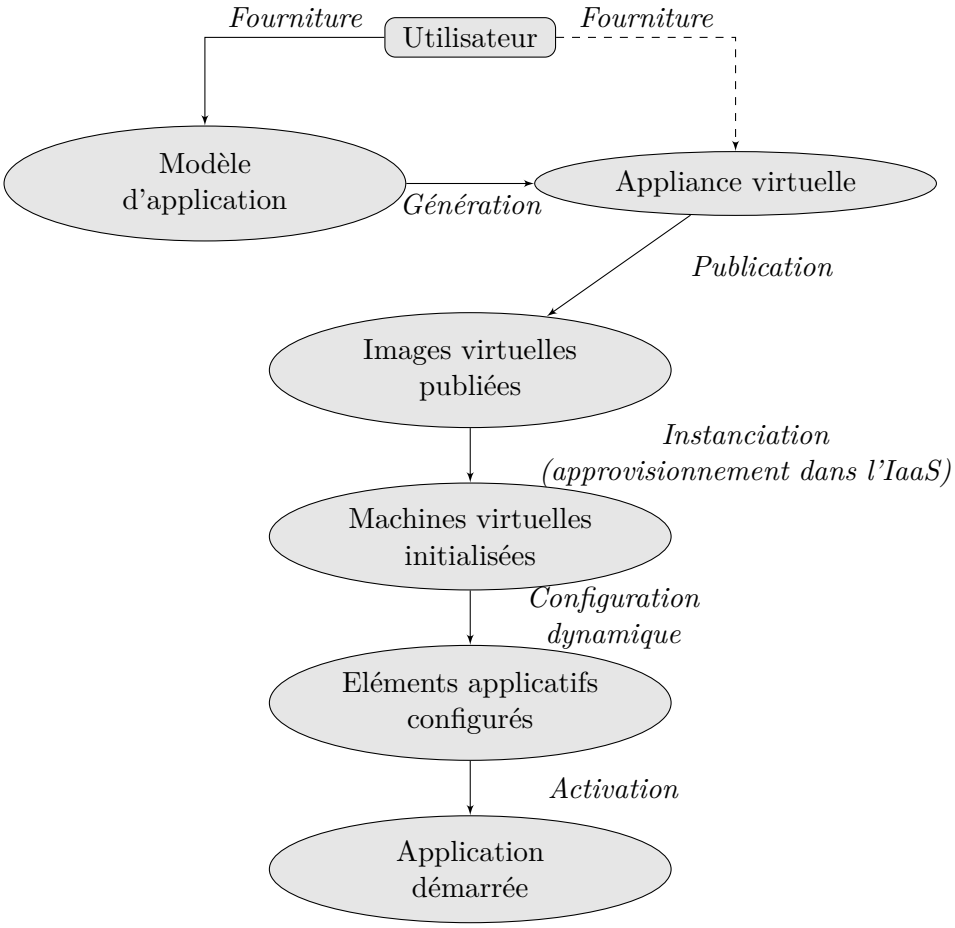

FIGURE 1.1 – Déploiement d'une application répartie en environnement de type informatique dans le nuage

Ces travaux de thèse ont été réalisés au sein de l'unité de recherche et développement *MSE* du département *Data & Service API* d'Orange Labs et du Laboratoire d'Informatique de Grenoble (*LIG*) au sein de l'équipe *Sardes* de l'INRIA Rhône-Alpes à Montbonnot. Ils ont donné lieu aux publications [64][65][108][109][63].

1.4 Organisation du document

Afin de faciliter la lecture de manuscrit et d'en améliorer la compréhension, celui-ci a été construit selon un plan structuré. En outre, les figures et schémas répondent à une charte graphique unifiée. Enfin, un exemple tenant lieu de fil directeur a été choisi et servira d'illustration à l'ensemble du propos. L'objectif de cette section est de détailler chacun de ces points.

1.4.1 Plan

La suite de ce document est organisée en quatre parties. La première correspond à un état de l'art des domaines relatifs à ces travaux. Elle est composée du chapitre 2, consacré aux techniques de déploiement d'applications réparties. Tout d'abord, il définit les notions de modélisation à base de composants et de cycle de vie applicatif, tout en précisant les spécificités relatives au contexte de l'informatique dans le nuage. Il définit alors les critères auxquels une solution doit se conformer pour assurer le déploiement automatique et fiable, dans le nuage, d'applications patrimoniales virtualisables. Enfin, ce chapitre se poursuit par la caractérisation, en fonction de ces critères, d'un panel représentatif de solutions de déploiement, certaines étant issues d'un contexte non virtualisé, d'autres correspondant à des systèmes de *PaaS*.

La deuxième partie présente les contributions relatives à ces travaux de thèse dans l'élaboration d'une solution de déploiement autonome et fiable d'applications arbitraires virtualisables dans le nuage. Le chapitre 3 offre une vue d'ensemble de VAMP en termes d'architecture fonctionnelle et technique. Il détaille également le formalisme utilisé pour modéliser une application en termes d'architecture applicative ainsi que des machines virtuelles qui la composent et des images virtuelles associées. Enfin, il présente les technologies mises en œuvre pour assurer le déploiement d'une application à l'aide de VAMP. Les chapitres suivants reprennent alors les aspects importants du déploiement autonome proposé par VAMP. Ainsi, le chapitre 4 est consacré à la génération des images virtuelles applicatives. Après un état de l'art sur les solutions existantes, il décrit la manière dont VAMP interagit avec une forge dont l'objectif est de créer des images virtuelles applicatives comprenant l'ensemble de la pile logicielle (i.e. le système d'exploitation, les intergiciels, les binaires et les données applicatives) nécessaires à l'instanciation, la configuration et l'activation des éléments logiciels déployés sur la machine virtuelle associée. Par la suite, le chapitre 5 décrit le protocole distribué et asynchrone mis en œuvre par chaque agent VAMP embarqué dans chaque machine virtuelle applicative pour réaliser les opérations de post-configuration et d'activation des composants dont il a la responsabilité. Le chapitre 6 conclut cette partie en détaillant, d'une part, la façon dont le protocole fiabilise le déploiement des machines virtuelles applicatives, d'autre part, l'approche utilisée pour fiabiliser les entités qui constituent le système VAMP lui-même.

La troisième partie, composée du chapitre 7, consiste en l'évaluation de la solution de déploiement autonome et fiable proposée par VAMP. Elle comprend un volet qualitatif qui illustre la polyvalence de VAMP, par le déploiement d'applications qui diffèrent par leur architecture, les domaines métiers qu'elles adressent ainsi que les choix technolo-

giques sur lesquelles elles reposent. Un second volet quantitatif mesure :

1. l'efficacité, en termes de temps d'exécution, de la solution de déploiement. Pour cela elle mesure notamment le surcoût qu'elle introduit et estime les gains liés à la parallélisation des traitements ;

2. la faculté de déployer simultanément un grand nombre de machines virtuelles applicatives qu'il s'agisse du déploiement simultané d'un grand nombre d'applications ou du déploiement d'une application large échelle.

La quatrième partie conclut ce document avec le chapitre 8 qui consiste en une synthèse des contributions de cette thèse et qui présente les perspectives associées.

1.4.2 Charte graphique

Certaines figures dans ce manuscrit représentent à la fois des éléments propres au système de déploiement VAMP et des éléments logiciels patrimoniaux, qui ne font pas partie de l'environnement VAMP à proprement parler. Afin de les distinguer aisément, les premiers sont colorés en orange (ou gris soutenu en équivalence de niveau de gris) alors que les seconds sont jaunes (ou gris très clair en équivalence de niveau de gris).

1.4.3 Exemple fil rouge

Afin d'améliorer la compréhension de ce document, un cas d'usage tenant lieu de fil directeur sera adopté dans la suite du document. Il s'agit d'une application appelée Springoo, dédiée à la gestion des références produits, des catalogues, des offres et des marchés d'une entreprise. Springoo est une application web qui sert de support d'illustration des bonnes pratiques dans la mise en œuvre des Enterprise Java Beans (EJB) au sein d'Orange. Elle est basée sur une architecture multi-tiers conforme au modèle de développement Java 2 Enterprise Edition Platform (JEE) [3]. Ce genre d'application comporte généralement un tiers présentation (i.e. un serveur web frontal) chargé de transmettre les requêtes émises par des clients web au tiers applicatif (i.e. un ensemble de serveurs d'applications). Celui-ci exécute la logique métier de l'application et génère dynamiquement les pages web. Pour cela il s'appuie sur un tiers base de données (i.e. un système de gestion de base de données ou SGBD) afin de stocker les informations applicatives de manière persistante. Comme l'illustre la figure 1.2, Springoo comprend :

- un SGBD MySQL ainsi que l'instance de base de données dédiée au stockage de l'ensemble des données de l'application ;

- un serveur d'applications JEE JOnAS qui met en œuvre la logique métier au travers d'une application (EAR) et d'un connecteur JDBC (RAR), permettant d'accéder à la base de données ;

- un serveur frontal HTTP Apache qui intègre un module JK, visant à répartir la charge à laquelle est soumise l'application, sur l'ensemble des instances de serveurs d'applications du tiers applicatif.

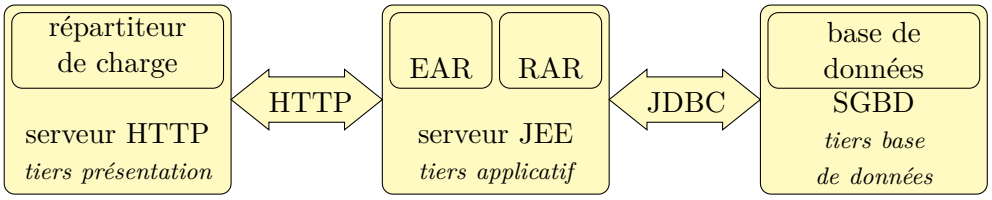

FIGURE 1.2 – Architecture logique de l'application Springoo

Chacun de ces trois sous-systèmes est installé sur une machine virtuelle qui lui est propre. Chacun d'eux expose des fonctions basiques d'administration permettant à un utilisateur de le démarrer, l'arrêter ou le redémarrer.

Première partie

État de l'art

Chapitre 2

Déploiement de systèmes répartis

Sommaire

2.1 Concepts et caractérisation 17
2.1.1 Concepts 17
2.1.2 Caractérisation 21
2.2 Solutions en environnements non virtualisés 25
2.2.1 Solutions à base de langages dédiés à des domaines 25
2.2.2 Solutions à base de composition de services 29
2.2.3 Solutions à base de langages de description d'architecture . . . 38
2.3 Solutions en environnements virtualisés 50
2.3.1 Solutions orientées infrastructure 50
2.3.2 Solutions orientées service 53
2.3.3 Solutions orientées application 57
2.4 Conclusion 69

La définition et la mise en œuvre d'une application est réalisée selon un processus appelé *cycle de vie*. Il se décompose en un ensemble d'*activités* élémentaires (e.g. développement, tests unitaires, etc.). Les activités élémentaires sont regroupées au sein de *phases*. Le contenu de chaque phase ainsi que l'organisation des phases entre elles sont décrites au travers d'une *méthodologie*. Ainsi, par exemple, les approches de type cycle en V, en cascade, en Z se focalisent sur la définition des phases relatives à la production d'un logiciel. L'ensemble des phases qui constituent le cycle de vie sont à leur tour regroupées dans des *macro-phases*.

Il existe généralement un lien assez étroit entre la répartition des phases au sein des macro-phases et la structure de l'organisation dans laquelle s'inscrit ce cycle de vie. En d'autres termes, une entité organisationnelle donnée (i.e. une équipe ou un service d'une entreprise) est généralement en charge de la mise en œuvre d'un sous-ensemble des macro-phases qui composent le cycle de vie applicatif. Néanmoins, la plupart des

cycles de vie logiciels, qu'il s'agisse d'approches incrémentales (e.g. Scrum [113]) ou non (e.g. cascade ou spirale [30]), définissent les macro-phases suivantes :

Conception : il s'agit de l'ensemble des activités qui, à partir de l'expression des besoins utilisateurs, vont aboutir à la définition de l'application à réaliser. Les éléments qui constituent les livrables de cette macro-phase sont notamment le cahier des charges, le dossier d'architecture fonctionnelle, le dossier d'architecture technique ainsi que les spécifications générales (ou fonctionnelles).

Réalisation : cette macro-phase consiste à produire une application dont les fonctionnalités couvrent l'ensemble des besoins exprimés au travers des éléments issus de la macro-phase de conception. La réalisation comprend notamment la rédaction des spécifications détaillées (ou techniques), le développement du code applicatif, la compilation, les tests unitaires, les tests d'intégration, les tests fonctionnels et le packaging.

Validation : elle consiste à vérifier que l'application répond aux exigences non-fonctionnelles (i.e. exigences techniques) exprimées par le client au travers de la maîtrise d'ouvrage. Il s'agit donc de tests de performances et de tenue en charge, de mesure de la fiabilité, etc.

Déploiement : cette macro-phase correspond à la pré-configuration, à l'installation, à la post-configuration et à l'activation de l'application en vue de son exploitation dans un contexte de production (ou de pré-production). Ces premières étapes sont parfois qualifiées de *déploiement initial*. Le déploiement comprend également les activités relatives au retrait (désinstallation) de l'application suite à son arrêt définitif. Bien que certaines définitions tendent à englober dans la définition du déploiement l'arrêt, la reconfiguration et le redémarrage de l'application au cours du temps, ces activités sont plutôt du ressort de l'étape d'administration.

Administration : cette macro-phase regroupe toutes les activités en lien avec la supervision, l'exploitation et la maintenance de l'application au cours de son exécution. Elle inclut notamment l'arrêt, la reconfiguration et le redémarrage de l'application qui peuvent être nécessaires dans le cadre de la prise en compte des variations de charge, des défaillances, des trous la sécurité, etc.

La finalité du travail présenté dans ce document étant de proposer une solution autonome et fiable assurant le déploiement initial d'une application virtualisable, arbitraire et possiblement patrimoniale dans le nuage, le but de ce chapitre est de proposer une vue d'ensemble des travaux existants dans le domaine du déploiement d'applications. Ainsi, la section 2.1 définit, d'une part, les concepts d'application répartie et de modèle à composants ainsi que le principe du déploiement, d'autre part, les facteurs et les critères qui caractérisent une solution de déploiement automatisée, fiable et polyvalente. La seconde partie du chapitre est consacrée à la présentation d'approches et de solutions existantes dans le domaine du déploiement d'application et à leur évaluation par rapport aux critères précédemment présentés. Ainsi, la section 2.2 s'intéresse aux solutions en

environnement non virtualisé alors que la section 2.3 se focalise sur les solutions issues de l'informatique dans le nuage. Enfin, la section 2.4 propose une synthèse de cet état de l'art.

2.1 Concepts et caractérisation

Cette section présente d'une part les concepts liés au domaine du déploiement d'applications réparties (section 2.1.1) puis elle propose un canevas de caractérisation (i.e. un ensemble de facteurs) des solutions de déploiements (section 2.1.2).

2.1.1 Concepts

2.1.1.1 Application répartie

Une application répartie peut être définie comme un ensemble de processus logiciels s'exécutant de façon concurrente et interopérant en vue d'offrir un ensemble de fonctionnalités. Un processus correspond à la mise en œuvre d'unités fonctionnelles appelées *composants* (cf. section 2.1.1.2). Les processus d'une application sont instanciés au sein d'un *environnement d'exécution* composé d'un ensemble de machines (également appelées *serveur* ou *hôte*) distinctes. Les interactions entre processus se traduisent par des échanges organisés de données au travers de canaux de communication.

D'autre part, une application répartie s'accompagne d'un ensemble de règles explicites (ou *contraintes*). Celles-ci définissent la manière dont les composants applicatifs doivent être instanciés avant de pouvoir être exécutés. Ainsi, la répartition des composants sur les hôtes qui forment l'environnement d'exécution est exprimée au moyen de *contraintes de placement*. De même, les *contraintes de configuration* définissent des dépendances de configuration entre composants et les *contraintes de démarrage* permettent d'exprimer l'ordre dans lequel les composants doivent être activés.

2.1.1.2 Modèle à composants

Historiquement, la modélisation, la conception et le développement d'applications a donné lieu à la définition d'approches successives. Ainsi, au cours des années 1980, l'approche objet a pris le pas sur l'approche modulaire. A l'inverse de l'approche modulaire qui manipule des regroupements d'attributs (e.g. comme les structures en C) sur des fragments de code partagés par toute une application, l'approche objet définit une application comme un ensemble d'entités logicielles autonomes : les *objets*. Un objet correspond au regroupement d'un ensemble d'attributs et de traitements. Cette approche a permis d'introduire les notions de polymorphisme, d'héritage, de redéfinition et de surcharge. Bien qu'elle ait connu un vif succès, l'approche objet présente deux restrictions majeures :

- de manière comparable à la taille des regroupements d'attributs de l'approche modulaire, la granularité des objets demeure trop fine, entravant ainsi leur potentiel de réutilisation.

- l'approche objet n'introduit pas de vue architecturale globale et explicite de l'application. Ceci induit que le code des objets mélange les aspects fonctionnels (i.e. ce qui doit être fait) et non-fonctionnels (i.e. la manière de le faire), limitant ainsi la maintenabilité et l'adaptabilité applicative au fil du temps.

Présentation

Par la suite, les années 1990 ont vu l'émergence de l'approche à composants. Elle vise à répondre aux limitations de la méthode objet en modélisant une application comme un ensemble d'entités logicielles (les *composants*) et de dépendances entre ces composants. Selon la définition proposée par [119], un composant est une entité modulaire et indépendante qui encapsule un sous-ensemble des fonctionnalités mises en œuvre au travers de l'application. Ainsi, un composant définit un ensemble de comportements, fonctionnels ou non, appelés *services*. La définition d'un service d'un composant peut soit s'appuyer sur une implémentation interne au composant lui-même ou recourir à des services proposés par d'autres composants. Ainsi, les interactions entre composants d'une application répartie s'appuient sur cette notion de service. Chaque composant expose ses services au travers de points d'accès identifiés appelés *interfaces* [35]. Les interfaces constituent les points d'accès uniques d'un composant. Elles définissent la manière d'interagir avec un service.

Un service peut être *fourni* par un composant (i.e. *interface serveur*) ou au contraire *requis* (i.e. *interface cliente*). Des interconnexions explicites entre composants sont exprimées au moyen de *liaisons*. Une liaison réifie l'interconnexion ou la *dépendance* entre une interface cliente et une interface serveur. Si elle interconnecte deux composants qui s'exécutent dans le même espace d'adressage (i.e. dans le même conteneur), elle est appelée *liaison locale*. Dans le cas contraire, il s'agit d'une *liaison distante*.

Un *assemblage* désigne un ensemble de composants interconnectés. Un assemblage est lui-même un composant qui expose des services. La notion d'assemblage introduit donc la notion de récursivité dans les modèles à composants. Une application est donc l'assemblage de tous les composants nécessaires à offrir l'ensemble des fonctionnalités de l'application.

Un composant s'exécute dans un *conteneur*. Un conteneur désigne un environnement d'exécution contrôlé qui offre un certain nombre d'opérations de gestion des composants (e.g. gestion du cycle de vie, mise en œuvre des politiques sur les composants, découverte et établissement des liaisons entre composants, etc.). Ces opérations sont généralement vues comme des services fournis par l'infrastructure.

L'*élaboration d'une liaison* (ou *résolution de dépendance*) est l'opération qui permet à un composant qui expose une interface requise d'obtenir la référence de l'interface fournie associée. Une telle référence correspond à la désignation d'un composant et d'une interface dans un espace de nommage. Selon le type de liaison, il peut s'agir d'une opération statique (e.g. indépendante de l'environnement dans lequel s'exécutent les composants) réalisée avant la phase d'installation ou dynamique (i.e. dépendante dudit environnement) réalisée une fois les composants installés.

Intérêts

L'approche à composant présente deux apports majeurs :

- grâce à la *composition* (assemblage) de composants, elle permet de définir des composants de granularité arbitraire, répondant ainsi au faible niveau de réutilisation des objets.
- grâce à la définition de services encapsulés au sein de composants (i.e. points d'accès uniques), elle permet de séparer les préoccupations fonctionnelles (i.e. le code métier de l'application) des préoccupations non-fonctionnelles, gérées au travers des services d'infrastructure exposés par les conteneurs. Associé à l'expression des relations entre composants (i.e. liaisons et assemblages) au sein d'une description architecturale globale et explicite, elle accroît la maintenabilité voire l'adaptation dynamique de l'application.

2.1.1.3 Déploiement d'application

Une première définition du déploiement logiciel a été proposée par l'*Object Management Group* (*OMG*) dans le cadre de la spécification relative au déploiement et à la configuration d'applications réparties à base de composants (OMG D&C) [100]. Selon cette proposition, le déploiement correspond à l'étape du cycle de vie applicatif qui fait suite à l'acquisition du logiciel et qui précède son exécution. En d'autres termes, après les macro-phases relatives à la production du logiciel, le déploiement est l'étape préliminaire à son exploitation. Il regroupe notamment les activités d'installation, de (post-)configuration et d'activation (i.e. démarrage initial) du système à déployer. La définition du déploiement proposée par l'OMG D&C désigne plus précisément la notion de *déploiement initial*. Par la suite, plusieurs travaux de caractérisation du déploiement [41] [52] [76] ont élargi cette première définition aux aspects de reconfiguration, de mise à jour, d'arrêt et de redémarrage de l'application au cours de son exploitation ainsi qu'à l'étape de retrait. Le canevas de caractérisation proposé par [41] défini ainsi le déploiement logiciel au moyen de huit activités :

Mise à disposition : la finalité de cette activité est de produire les paquets installables relatifs au système à déployer. Elle se décompose donc en deux sous-activités :

le *packaging* qui consiste à construire ces paquets, à partir des binaires issus de la phase de développement mais également d'un ensemble de données de configuration statique, c'est-à-dire indépendante de l'environnement d'exécution du système à déployer.

Dans un contexte virtualisé, les paquets produits sont des *images virtuelles*. Il s'agit de la représentation du contenu du disque de la machine virtuelle au sein de laquelle elle sera instanciée. Une image virtuelle est une unité logique contenant une version installée et préconfigurée de l'ensemble des logiciels nécessaires à l'exécution de l'application répartie[1].

[1] Dans le cas général, une image virtuelle peut ne contenir qu'un sous ensemble des éléments logiciels

La génération d'une image virtuelle consiste donc à créer un paquet résultat de l'installation et de la configuration statique d'un système d'exploitation, d'un ensemble d'intergiciels, et de binaires et données applicatives. Par conséquent, dans un environnement virtualisé, l'activité de packaging à l'origine de la génération des images virtuelles, reprend à son compte une partie de la phase d'installation.

la publication dont l'objectif est de rendre les paquets, issus de l'activité de packaging, accessibles aux clients en vue de leur installation. Elle peut prendre différentes formes, comme l'enregistrement des paquets au sein d'un référentiel d'installation ou encore celui des appliances dans le référentiel d'images d'une plate-forme d'*IaaS* dans le contexte de l'informatique dans le nuage.

Installation : cette activité consiste à transférer les paquets issus de l'activité de mise à disposition sur l'environnement d'exécution de l'application (cf. section 2.1.1.1), dans le respect des contraintes de placement. Chaque paquet est alors intégré au système d'exploitation sur lequel il a été déposé. Par la suite, des opérations de configuration permettent de renseigner les paramètres du système à déployer dont la valeur dépend de l'environnement d'exécution. On parle également d'opération de post-configuration ou de configuration dynamique. La mise en œuvre de ces opérations vise à résoudre les dépendances entre composants exprimées au moyen des contraintes de configuration. Une fois la post-configuration terminée, le système peut être activé.

Dans un contexte non virtualisé, le processus de déploiement d'une application suppose que l'environnement matériel d'exécution dans lequel l'application doit être instanciée est déjà disponible. Aucune étape du processus de déploiement ne prévoit donc l'approvisionnement de l'environnement d'exécution. A l'inverse, dans le contexte de l'informatique dans le nuage, l'infrastructure matérielle d'exécution (e.g. les machines virtuelles, les disques virtuels, etc.) peut être mise à disposition dynamiquement en préambule à la phase d'installation, juste avant la récupération des images virtuelles.

Démarrage : cette activité consiste à activer les éléments qui participent au système distribué en tenant compte des éventuelles contraintes de démarrage entre ces éléments. Au terme de sa première activation complète (*démarrage initial*), le système est déployé et est désormais considéré comme en cours d'exploitation. Il pourra faire l'objet de nouvelles demandes de démarrage s'il se retrouve à un moment dans un état arrêté.

Arrêt : de façon symétrique au démarrage, cette activité consiste à désactiver les éléments, précédemment activés, qui constituent le système déployé en tenant compte

lui permettant d'exécuter les entités applicatives les autres faisant l'objet d'un traitement ultérieur, lors de la phase d'installation.

d'éventuelles contraintes en matière d'ordre d'arrêt entre ces éléments[2].

Mise à jour : cette activité recouvre plusieurs niveaux de complexité. Elle inclut :

la reconfiguration qui correspond à une modification de la valeur d'un sous-ensemble de paramètres de configuration du système sans nécessiter de redémarrage : elle peut être réalisée à chaud, alors que le système s'exécute ;

l'adaptation qui correspond à une reconfiguration nécessitant un redémarrage de système (i.e. reconfiguration à froid). Elle s'accompagne éventuellement de l'installation, de la configuration et de l'activation d'un paquet existant dans une version plus récente mais elle n'implique pas de réaliser la phase de mise à disposition dans sa totalité.

la mise à jour qui est comparable à une installation dans un environnement d'exécution non vierge, c'est-à-dire sur lequel s'exécute déjà une version antérieure de l'application. Il s'agit du remplacement d'éléments applicatifs existants par des éléments mis à disposition dans une version plus récente.

Désinstallation : cette activité consiste à libérer les ressources allouées à l'exécution de l'application. Ainsi, les éléments désinstallés sont préalablement désactivés puis les ressources de calcul, de mémoire, de réseau et de stockage qui pouvaient leur être allouées sont libérées. Cela correspond, dans le cas du stockage, à un effacement des données et binaires stockés sur disque.

De manière symétrique à la phase d'installation, dans le contexte de l'informatique dans le nuage, la désinstallation peut se terminer par la destruction des machines virtuelles qui composaient l'environnement d'exécution de l'application alors que dans un contexte non virtualisé, les ressources matérielles sont seulement libérées.

Retrait : cette activité marque l'obsolescence de l'application. Le retrait correspond à la suppression des paquets publiés dans le référentiel d'installation, lors de la phase de mise à disposition. Ils deviennent alors indisponibles.

La figure 2.1 reprend la définition des huit étapes et des transitions associées, proposée dans le canevas de [41]. Seules les étapes hachurées, qui désignent celles relatives au déploiement initial, ont été étudiées au cours de ce doctorat (cf. section 1.3). Dans la suite du document, la définition retenue pour le déploiement sera celle de l'OMG et les termes *déploiement* et *déploiement initial* seront employés indistinctement.

2.1.2 Caractérisation

L'élaboration d'un système autonome et fiable, capable de déployer n'importe quelle application patrimoniale passe par la satisfaction d'un certain nombre d'exigences en lien avec les principaux facteurs présentés en section 1.2.2. L'objectif de cette section

[2]Les contraintes relatives à l'ordre d'arrêt des éléments qui constituent le système déployé ne sont pas nécessairement symétriques à celle relatives à l'ordre de démarrage.

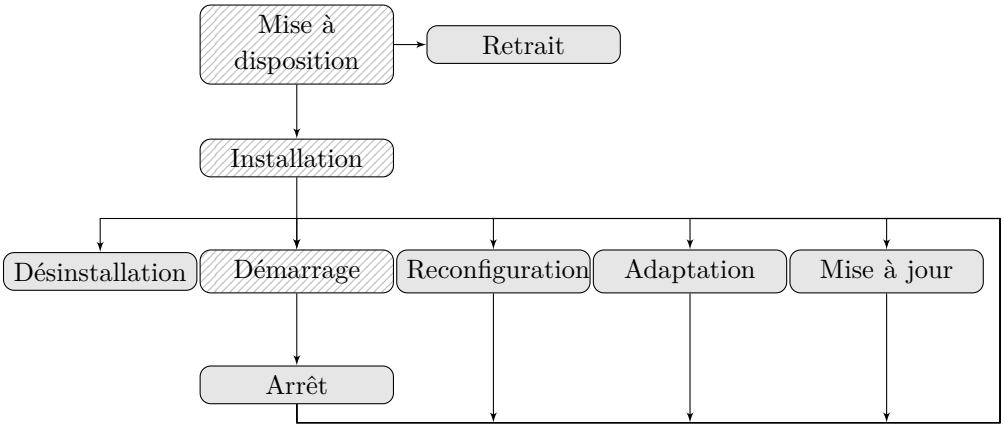

FIGURE 2.1 – Activités de la phase de déploiement d'une application

est donc de préciser la liste de ces facteurs et d'exprimer, pour chacun d'eux, le niveau d'exigence escompté.

2.1.2.1 Automatisation

La finalité de cette propriété est de mesurer la capacité du système considéré à exécuter le processus de déploiement sans avoir recours à une intervention extérieure humaine. Le niveau d'automatisation global du processus de déploiement s'apprécie donc en fonction, d'une part, du degré d'automatisation de chacune des phases qui le composent, d'autre part, de celui relatif à l'enchaînement entre ces phases.

2.1.2.2 Généricité

La généricité désigne la capacité de la solution à proposer un mécanisme applicable de manière identique pour déployer n'importe quelle application patrimoniale, sans présenter de restriction particulière vis-à-vis de l'environnement d'exécution applicatif. Cette propriété se décline selon les trois critères suivants :

Polyvalence

La polyvalence caractérise la variété des applications que le système est en mesure de déployer. Ainsi, elle s'apprécie en fonction de la capacité du système à déployer différentes architectures applicatives (e.g. architectures n-tiers, en étoile, pair à pair, etc.), différentes technologies (e.g. Java Enterprise Edition, C, etc.) et différents domaines métiers.

Indépendance vis-à-vis de l'environnement d'exécution applicatif

Ce critère caractérise la variété des environnements d'exécution applicatifs dans lesquels le système est en mesure de réaliser des déploiements. Dans le cas de l'informatique dans le nuage, par exemple, il estime le degré d'indépendance de la solution vis-à-vis

de la plate-forme d'*IaaS*. Il s'agit donc d'une appréciation du caractère agnostique du système vis-à-vis des infrastructures matérielles et logicielles utilisées dans le cadre du déploiement d'applications.

Adhérence applicative

L'adhérence applicative caractérise le niveau de couplage entre la solution de déploiement et l'application déployée considérée. Ainsi, plus le système de déploiement aura recours à des traitements spécifiques à l'application (e.g. des scripts dédiés) pour en réaliser le déploiement, plus l'adhérence sera forte. A l'inverse, l'adhérence sera minimale si le système de déploiement n'a recours qu'à des traitements pouvant s'appliquer au déploiement de n'importe quelle application.

L'adhérence applicative du système de déploiement est liée à la facilité à décrire les traitements relatifs au déploiement. Or, selon [41], cette dernière dépend du niveau d'abstraction manipulé et exposé par le système de déploiement. Ainsi, le recours à des modèles d'abstraction permet de substituer l'usage de scripts dédiés par des traitements génériques. [41] définit trois modèles d'abstraction :

le modèle de site qui décrit l'environnement de déploiement et d'exécution de l'application ;

le modèle de produit qui offre une vue logique de l'architecture applicative en termes de dépendances et de contraintes du système à déployer ;

le modèle de politique dont la finalité est d'adapter le comportement du processus de déploiement en fonction de critères environnementaux.

En résumé, la polyvalence et l'adhérence applicative correspondent à deux caractéristiques proches mais complémentaires. La première précise le panel d'applications que la solution est capable de déployer, en réponse à la question "quoi ?". La seconde caractérise la manière d'y parvenir, en réponse à la question "comment ?".

2.1.2.3 Passage à l'échelle

Le passage à l'échelle correspond à la faculté du système de déploiement à gérer simultanément un nombre important de composants applicatifs. Son estimation s'effectue au travers de deux critères :

Large échelle

Il s'agit de sa capacité à déployer une application indépendamment de sa taille ou de la complexité de son architecture, telle qu'une application constituée d'une multitude d'entités logicielles largement réparties et fortement interconnectées (e.g. architecture de type pair à pair).

Déploiements multiples

Il s'agit de sa capacité à prendre en compte, en parallèle, un grand nombre de requêtes utilisateur nécessitant de procéder au déploiement simultané d'une multitude d'instance d'applications.

2.1.2.4 Fiabilité

Il s'agit de la garantie, qu'en présence d'un nombre fini de pannes franches, toute opération de déploiement parvienne à son terme, c'est-à-dire que l'ensemble des composants applicatifs soient tous activés simultanément durant un laps de temps non nul.

La propriété de fiabilité peut se subdiviser en deux sous-propriétés selon la nature des défaillances :

- D'une part, une propriété non fonctionnelle qualifiée de *fiabilité technique* qui vise à se prémunir des défaillances de l'environnement d'exécution applicatif ou du système de déploiement lui-même :
  - Lorsqu'il s'agit d'une défaillance de l'environnement d'exécution de l'application, celui-ci peut être considéré comme générique, c'est-à-dire faisant intervenir les mêmes entités quelque soit l'application considérée. Ainsi, il est constitué d'un ensemble de serveurs interconnectés au travers de réseaux. Dès lors, le système de déploiement doit être en mesure de détecter et de corriger les défaillances de l'environnement d'exécution. Il s'agit des pannes franches de machines et des problèmes réseaux.
  - En présence de défaillances du système de déploiement, ce dernier doit être capable de les supporter, soit en adoptant une stratégie d'autoréparation ou, à défaut, par compensation des entités défaillantes.
- D'autre part, une propriété fonctionnelle qualifiée de *fiabilité fonctionnelle* qui concerne les défaillances propres (i.e. internes) à l'application. Pour que de telles défaillances puissent être détectées et traitées par un système externe à l'application (e.g. le système de déploiement), il est indispensable que cette dernière se conforme à un certain nombre d'exigences (e.g. exposition de données de monitoring au moyen de sondes, implémentation du principe d'*arrêt sur défaillance* ou *fail stop*). Cette hypothèse est en contradiction avec la polyvalence du système de déploiement, qui vise à n'imposer aucun prérequis quant aux capacités offertes par l'application pour détecter et résoudre ces dysfonctionnements au travers d'un système externe.

Ainsi, les travaux présentés ici n'ont pas traité les aspects relatifs à la fiabilité fonctionnelle. Désormais, les termes *fiabilité* et *fiabilité technique* seront indistinctement utilisés dans la suite de ce manuscrit.

La fiabilité s'apprécie en fonction de la mise en œuvre de mécanismes de remontée d'erreur, de reprises après panne, de tolérance aux fautes ainsi que par vérification formelle d'algorithmes et de protocoles.

2.1.2.5 Granularité

La granularité caractérise la finesse avec laquelle le système considéré est capable d'assurer le déploiement d'une application. Plus précisément elle représente la plus petite unité de déploiement (e.g. machine virtuelle, paquet logiciel, composant) manipulable par le système. Plus la granularité est fine, plus l'utilisateur est potentiellement capable de détailler précisément les dépendances entre les entités qui composent l'application. Ceci permet de définir des architectures applicatives plus complexes en brisant des cycles entre dépendances.

2.2 Solutions en environnements non virtualisés

L'objectif de cette section est de présenter un certain nombre de travaux existants en matière de déploiement d'applications distribuées. Il s'agit de les positionner vis-à-vis des propriétés requises (cf. section 1.2.2) en vue de la réalisation d'une solution de déploiement autonome et fiable d'applications arbitraires dans un environnement de type informatique dans le nuage.

2.2.1 Solutions à base de langages dédiés à des domaines

Selon la définition de [124], un *langage dédié à un domaine* (*Domain Specific Language* ou *DSL*)) est un langage de programmation ou de spécification exécutable dont le but est d'offrir un haut niveau d'expressivité focalisé sur un domaine spécifique. Pour cela, il propose des notations et des abstractions adaptées au domaine considéré, facilitant ainsi leur utilisation par rapport à celle de langage de programmation plus généraux [91].

La gestion automatique de la configuration des applications est un domaine qui a donné lieu à un très grand nombre de travaux. Néanmoins, la plupart d'entre eux adoptent une stratégie visant à maintenir la cohérence entre une configuration courante et une configuration de référence [55][18][105][43][101]. Pour cela, elles s'appuient sur une architecture client serveur et le contenu de la configuration est représenté, de façon explicite, au moyen d'un *DSL*.

2.2.1.1 Puppet

Parmi ces solutions, *Puppet* [104] est l'une des plus utilisées actuellement. Les données qu'il manipule sont des *ressources* relatives à un système d'exploitation. Elles sont décrites au moyen d'un formalisme déclaratif et explicite (i.e. un *DSL*) au sein d'un descripteur appelé *classe*. Une classe *Puppet* permet de définir :

la structure de l'arborescence d'une unité de disque : il s'agit de l'ensemble des répertoires [*directory*), des fichiers (*file*) et des liens (*link*) qui doivent être créés, copiés à partir d'un modèle personnalisable ou supprimés. La classe précise également les attributs (e.g. propriétaire, groupe, droits d'accès) relatif à chaque ressource.

les *packages* : il s'agit des éléments logiciels qui doivent être ajoutés, modifiés ou supprimés par le gestionnaire de paquets ainsi que la politique de mise à jour associée.

les *services* : ils correspondent aux processus *démons* à démarrer, arrêter ou redémarrer ;

les *exec* : il s'agit des commandes à lancer en précisant les arguments associés ;

les utilisateurs (*user*) , en précisant les groupes dont ils font partie et les privilèges qui leur sont attribués.

Un attribut d'une ressource *Puppet* peut contenir une valeur ou faire référence à la valeur d'une autre ressource de la même classe, mais pas nécessairement localisée sur la même machine.

Il est possible d'exprimer deux types de dépendances entre les ressources. D'une part des dépendances d'ordonnancement entre ressources, indiquant qu'une ressource ne peut être traitée par *Puppet* que si les ressources dont elle dépend l'ont été préalablement. Leur utilisation se fait à l'aide du mot clef *before* pour indiquer qu'une ressource doit être traité par *Puppet* avant une autre, ou à l'aide du mot clef *require* pour indiquer qu'une ressource doit attendre le traitement d'une ou plusieurs autres[3] avant d'être elle-même traitée. Le deuxième type de dépendance est semblable au premier à ceci près qu'il indique qu'une ressource doit faire l'objet d'un rafraîchissement à chaque modification de la ou des ressource(s) dont elle dépend. Dans ce cas, le mot clef *notify* est le pendant de *before*, et *subscribe* est le pendant de *require*.

Le listing 2.1 contient des extraits de configuration de ressources selon le formalisme proposé par *Puppet*.

```
# Copie le fichier de référence
# puppet:///sudo/files/sudoers
# dans /etc/suoders
file { "/etc/sudoers":
    ensure  => file,
    owner   => "root",
    group   => "root",
    mode    => "440",
    source  => "puppet:///sudo/files/sudoers",
    require => Package["sudo"],
}

# Installe le package kernel du système
# d'exploitation mais ne procède pas
# à sa mise à jour automatique
package { "kernel":
    ensure => installed
}

# Démarrage du service sshd
```

[3]La définition de plusieurs ressources est réalisée à l'aide d'une liste d'éléments séparés par des virgules

```
service { "sshd":
    ensure    => running,
    subscribe => [ Package["openssh-server"],
        File["/etc/ssh/sshd_config"] ],
}


# Exécution de commande
exec { "/usr/bin/commande argument1 argument2":
    cwd  => "/rep/de/travail",
    path => "/usr/bin:/usr/sbin:/bin",
}
```

Listing 2.1 – Exemple de ressources déclarées à l'aide du *DSL* proposé dans *Puppet*

Enfin, des notions d'héritages et de collection de classes existent, permettant de réutiliser au mieux le code *Puppet*.

Dans le contexte de la gestion de la configuration d'un ensemble d'hôtes interconnectés, le système *Puppet* se décompose en deux types d'entités : d'une part une entité centrale ou *serveur*, d'autre part un ensemble de composants locaux (un par hôte) appelés *agents*. Le serveur *Puppet* regroupe l'ensemble des classes de référence. A chaque hôte composant le système correspond une classe de référence stockée dans le serveur. Elle contient la configuration à mettre œuvre sur l'hôte. Le rôle d'un agent *Puppet* est de synchroniser la configuration de l'hôte sur lequel il est instancié avec celle décrite dans une classe de référence. Un agent est un processus qui peut être activé périodiquement ou à la demande. Il peut fonctionner par récupération de la recette auprès d'un serveur *Puppet* ou de façon autonome, si une classe lui est fournie. Bien qu'en mode client-serveur, le serveur joue la fonction de maître, c'est-à-dire que la configuration servant de référence soit celle qu'il détienne et non celle contenue sur l'hôte, un agent peut exporter des éléments de configuration vers le serveur. Ce dernier peut utiliser des données exportées par un hôte pour les importer dans d'autres classes, en vue de résoudre des dépendances de configuration entre hôtes.

Automatisation

Puppet s'affiche comme une solution dédiée à la configuration de systèmes. Elle adresse donc les phases de configuration et d'activation d'éléments applicatifs au moyen de scripts spécifiques. Elle couvre également la phase de mise à disposition grâce à la publication des classes au niveau du serveur. Enfin, *Puppet* assure également l'installation d'entités applicatives en recourant aux mécanismes de gestion de paquets (e.g. RPM, DEB, YUM) proposées par le système d'exploitation de l'environnement d'exécution de l'application. Le degré d'automatisation proposé dans *Puppet* peut donc être considéré comme satisfaisant.

Généricité

Proposant une solution de configuration au niveau système, *Puppet* n'impose que peu de restriction quant aux applications déployables. Ainsi, les restrictions sont liées à

l'environnement d'exécution (i.e. système d'exploitation) dans lequel les agents *Puppet* sont instanciés.

En revanche, l'adhérence applicative de *Puppet* est forte. En effet, il modélise essentiellement le niveau système d'exploitation (i.e. une sous-partie du modèle de cible) et n'adresse le niveau applicatif (i.e. modèle de produit) que de façon élémentaire. Ainsi *Puppet* n'expose pas de modèle global de l'application qui permettrait de mutualiser certaines opérations du déploiement des applications, au lieu de recourir nécessairement à des scripts fortement spécialisés, comme cela est le cas actuellement.

Enfin, dans sa version libre, *Puppet* n'est disponible que sous plusieurs distributions du système d'exploitation *Linux* (e.g. Ubuntu, Red Hat, Debian, ...). Etant donné qu'il est nécessaire d'instancier un agent *Puppet* sur chaque hôte composant l'environnement d'exécution de l'applicatif. Cette solution ne permet donc de déployer que des applications s'exécutant sur *Linux*. En outre, comme la plupart des solutions de déploiement en environnement non virtualisé, la solution ne propose pas de mécanisme automatisé d'approvisionnement de l'environnement matériel d'exécution. Elle n'offre donc pas une grande indépendance vis-à-vis du contexte d'exécution. Le portage du code de *Puppet* est néanmoins envisageable, les sources étant disponibles.

Passage à l'échelle

De par son architecture centralisée couplée à un mécanisme récurrent de ré-application des classes dont la mise en œuvre a échoué, *Puppet* est inadapté pour le déploiement d'application de grande taille (i.e. large échelle). Le problème est comparable concernant les déploiements multiples à ceci près que rien n'empêche d'utiliser un serveur *Puppet* par application à déployer.

Fiabilité

Puppet ne propose pas de mécanisme de détection de panne du serveur. De plus, en présence d'une défaillance franche de l'hôte sur lequel est instancié un agent *Puppet*, aucun mécanisme de remplacement n'est prévu. Cependant le serveur est en mesure de tolérer son absence dans l'attente d'un remplacement manuel.

Granularité

L'unité de déploiement de base proposé par *Puppet* est la classe. Or, au sein de chaque hôte, ne tourne qu'un agent *Puppet* chargé d'appliquer les éléments relatifs à une classe donnée. La granularité d'une classe correspond donc à l'ensemble de la pile logicielle d'une machine (i.e. système d'exploitation, intergiciels, éléments applicatifs). *Puppet* présente donc une granularité grossière.

Synthèse

Bien que le déploiement d'une application ne soit pas impossible au travers de *Puppet*, cette solution demeure plutôt dédiée à la configuration du système d'exploitation. Ainsi l'absence de modèle de produit et son insuffisante granularité (i.e. niveau pile logicielle) limitent le spectre des applications aisément déployables. En outre, bien que tolérant

aux défaillances des agents composant le système, sa fiabilité demeure conditionnée par le bon fonctionnement du serveur. Enfin, son architecture centralisée limite la taille et la quantité des applications déployables par un même serveur *Puppet*.

2.2.2 Solutions à base de composition de services

L'*architecture orientée services* (*Service Oriented Architecture* ou *SOA*) [103] est une approche qui définit un patron pour la mise en œuvre de fonctionnalités, regroupées au sein d'unités d'exécution appelées *services*, proposées par des *fournisseurs* ou *producteurs* à destination de *clients* ou *consommateurs*. La relation qui lie le producteur et le consommateur d'un service est décrite au moyen d'un *contrat*. Le contrat formalise, selon des standards, les fonctionnalités offertes par le service ainsi que le contexte technique dans lequel ces fonctionnalités doivent être délivrées. En revanche, le contrat ne précise pas la manière dont ces fonctionnalités sont implémentées au niveau du producteur. Un service est indépendant d'une technologie ou de langage de programmation donné. Enfin un service est publiable et découvrable : il peut être enregistré par le fournisseur au sein d'un *annuaire* ou *registre*. Le consommateur peut, quant à lui, consulter le registre et, à partir des informations contenues dans le contrat, choisir d'accéder au service. L'approche *SOA* vise donc, d'une part, à faciliter la réutilisation d'entités logicielles, d'autre part, à améliorer l'interopérabilité entre un client et un fournisseur par diminution de couplage.

Les spécifications *Service Component Architecture* (*SCA*) sont le résultat d'un travail issu d'un groupe de travail, l'*Open Service Oriented Architecture* (*OSOA*), composé notamment de grand noms de l'informatique comme IBM, Oracle, Red Hat, SAP et Siemens. Elles ont pour objet de définir un modèle capable de définir une application comme l'assemblage de composants s'insérant dans une architecture orientée services.

DeployWare, *FraSCAti* et *OSGi* constituent trois exemples de solution capable d'assurer le déploiement d'applications dans un contexte *SCA*.

2.2.2.1 DeployWare

Egalement baptisé *Fractal Deployment Framework* (*FDF*), *DeployWare* est un environnement à base de composants capable de déployer des applications patrimoniales réparties et distribuées [70]. Il se compose :

d'un modèle : Le formalisme de description d'application proposé par *DeployWare* s'appuie sur un langage dédié (i.e. *DSL*) au déploiement d'applications sur les grilles au moyen duquel, l'utilisateur définit son application en termes de *nœuds* (i.e. les machines sur lesquels l'application se répartit) et de *services* (i.e. les unités fonctionnelles qui participent à l'application). Les nœuds et les services sont des composants Fractal qui regroupent un ensemble d'attributs. Le langage permet également de décrire des dépendances de configuration et d'activation entre composants. En effet, par défaut, l'environnement de déploiement traite les machines et les services dans l'ordre d'apparition dans la description applicative, mais il est possible de spécifier que certains traitements peuvent être réalisés en parallèle.

d'une librairie de composants : L'environnement propose un registre dans lequel il est possible d'ajouter des composants en vue de leur réutilisation au sein d'autre applications. C'est ainsi le cas de composants interfaçant un serveur HTTP Apache ou des clients de communication comme FTP, SSH ou telnet. Cette librairie est comparable à l'annuaire de services du patron défini par l'approche *SOA*.

d'un moteur de déploiement : Les outils qui constituent le cœur de la solution exposent à l'utilisateur les fonctionnalités d'installation, de configuration, d'activation, d'arrêt et de désinstallation d'une application. L'un de ces outils a pour rôle de convertir la description applicative fournie par l'utilisateur dans le formalisme décrit précédemment dans un langage de description d'architecture plus riche (i.e. Fractal ADL).

L'architecture de *DeployWare* est distribuée au niveau des machines sur lesquelles est déployée une application. Ainsi, une sorte de machine virtuelle *DeployWare* exécute une partie de la logique de déploiement permettant d'instancier et de gérer les composants.

Automatisation

DeployWare automatise les phases d'installation, de configuration, d'activation, d'arrêt et de désinstallation de l'application. En outre, il offre la capacité d'assurer son propre déploiement. Néanmoins, il ne propose pas de mécanisme automatisé pour les aspects liés à la mise en œuvre de l'application et plus précisément à la phase de packaging.

Généricité

Désireux d'offrir un environnement capable de déployer n'importe quelle application patrimoniale, *DeployWare* s'appuie sur un modèle à composants (i.e. Fractal) très complet disposant de capacité d'introspection, de partage de composant et de hiérarchisation entre composant. Ceci lui confère une bonne polyvalence. Celle-ci n'est pas altérée par le recours à un *DSL* qui en reprend la notion de composants, d'attributs et de liaisons (i.e. afin d'exprimer les dépendances).

En outre, *DeployWare* est une des rares solutions en environnement non virtualisé à proposer des mécanismes d'approvisionnement de l'environnement matériel d'exécution. Couplé à la modélisation des entités manipulées à l'aide de composant, il présente un niveau élevé d'indépendance à l'environnement d'exécution de l'application.

Enfin, le formalisme de description d'application proposé par *DeployWare* permettant d'expliciter l'architecture applicative ainsi que les attributs de configuration et les dépendances entre composants, son adhérence applicative demeure limitée.

Passage à l'échelle

L'intention première de *DeployWare* étant d'assurer le déploiement d'application de grande taille dans des grilles de calcul, il présente une architecture distribuée dont la

gestion large échelle a été expérimentée[4] [70]. En revanche, aucune expérimentation ne semble avoir été faite quant à la capacité de *DeployWare* d'assurer des déploiements multiples.

Fiabilité

DeployWare propose des mécanismes de validation des déploiements avant leur mise en œuvre. Une telle approche permet de vérifier un certain nombre de propriétés (e.g. notamment la présence de services requis dans le cadre de dépendances) et d'enrichir le modèle avec les résultats obtenus. En outre, *DeployWare* propose que chaque étape du déploiement d'un composant s'accompagne d'une étape de repliement afin d'assurer d'éventuels retours arrière. Ainsi, l'environnement de déploiement proposé par *DeployWare* présente un premier niveau de fiabilité mais qui ne permet cependant pas de se prémunir des pannes franches affectant l'environnement d'exécution de l'application ou le système de déploiement lui-même.

Granularité

Le recours à un modèle à composant permettant de modéliser n'importe quelle ressource logicielle ou matérielle confère à *DeployWare* une granularité optimale.

Synthèse

Contrairement à beaucoup d'environnements de déploiement (e.g. Java EE, OSGi, OpenCCM), *DeployWare* est capable de s'auto-déployer. Néanmoins, il présente comme principale limitation de ne pas automatiser la phase de mise à disposition du logiciel. Néanmoins, son modèle à base de composants lui confère une bonne généricité et une excellente granularité même si le *DSL* permettant de décrire une application ne repose pas sur des standards. De plus, grâce à son architecture distribuée, *DeployWare* offre un bon niveau de gestion large échelle. En revanche, *DeployWare* ne présente pas de capacité particulière en matière de fiabilité vis-à-vis des pannes franches affectant l'environnement d'exécution de l'application ou le système de déploiement lui-même.

2.2.2.2 *FraSCAti*

FraSCAti est un environnement conforme aux spécifications SCA assurant la mise en œuvre d'applications réparties dans un contexte SOA. Il est le résultat de travaux de recherche réalisés conjointement par le Laboratoire d'Informatique Fondamentale de Lille (LIFL) et l'INRIA au sein du projet ADAM [115] [114]. Outre le fait de proposer un environnement répondant aux spécifications SCA, *FraSCAti* a également pour finalité d'en adresser certaines limitations identifiés par [103]. Celles-ci portent essentiellement sur des manques en termes de capacités de configuration et d'administration. Ainsi, SCA ne spécifie pas :

[4] Il faut cependant noter que ces expérimentations ne prenaient pas en compte la phase de packaging, puisque *DeployWare* ne l'automatise pas.

- de méthode de gestion de la (re-)configuration une fois l'application démarrée (i.e. lors de son l'exécution) ;
- de mécanisme d'interaction avec des services techniques mis à disposition par la plate-forme SCA (i.e. gestionnaire de transaction, d'authentification, ...) ;
- les fonctions d'administration (e.g. tolérance aux pannes, gestion des performances ou de la sécurité) des composants applicatifs voire de la plate-forme elle-même.

Pour répondre à ces besoins, *FraSCAti* propose des mécanismes d'introspection qui s'appuient sur une approche à composants grâce à laquelle il modélise l'ensemble des éléments qui constituent le système (i.e. les entités logicielles métiers, les services non fonctionnels, etc.). Concrètement, *FraSCAti* utilise le modèle à composants Fractal [36]. Il s'agit d'un modèle à composants réflexif, indépendant de tout langage de programmation, visant à mettre en œuvre des systèmes répartis complexes. Par réflexif, il faut comprendre que le modèle Fractal permet de réifier les métadonnées contenues dans les composants. Ainsi *FraSCAti* définit les notions de composants (*component*), d'interfaces clientes (*reference*) et serveurs (*service*), de liaison (*wire*) ainsi que d'implémentation. En outre, à l'instar de ce qui existe dans Fractal, *FraSCAti* permet de définir des contrôleurs qui correspondent à un aspect non-fonctionnel géré au niveau du composant (e.g. gestion des liaisons). Un contrôleur est associé à un *service* ou une *reference* à l'aide d'un intercepteur (attribut *require*). L'ensemble des contrôleurs d'un composant définissent sa *personnalité*. Une personnalité définit donc une politique d'administration. Grâce à Fractal, *FraSCAti* permet donc aux utilisateurs de définir la personnalité de leur composants. Par défaut, il en propose 6 :

***Property Controller* :** offre un accès en lecture / écriture aux attributs de configuration d'un composant ;

***Wiring Controller* :** gère la création, la modification et la suppression de liaisons entre services et références ;

***Instance Controller* :** permet de générer (i.e. créer, supprimer) l'instance du composant qu'il s'agisse d'un composant sans état (i.e. toutes les instances du composant sont équivalentes), d'un composant associé à une requête ou à une session utilisateur voire d'un singleton ;

***Life-cycle Controller* :** gère le comportement du composant lors de la mise en œuvre de son cycle de vie ;

***Hierarchy Controller* :** gère l'insertion du composant au sein d'un assemblage hiérarchique de composants ;

***Intent Controller* :** permet d'ajouter et de retirer des services non-fonctionnels correspondant à des façades de la personnalité du composant. Autrement dit, ce contrôleur a un rôle similaire au *Wiring Controller* vis-à-vis des intercepteurs.

Le listing 2.2 extrait de [115] reprend l'ensemble de ces notions au sein d'une description au format XML très proche de la syntaxe proposée par le langage de description d'architecture Fractal ADL [102].

```
<!-- Définition de l'application sous forme d'un composant
    composite -->
<composite name="MyApp">
  <!-- service fourni par l'application -->
  <!-- ce service correspond à l'exportation du service run du
      sous composant view -->
  <!-- au service est associé l'intercepteur SecurityService -->
  <service name="run" promote="View/run"
      require="SecurityService"/>
  <!-- sous composant View -->
  <component name="View">
    <!-- service fourni lié à l'intercepteur LoggingService -->
    <service name="run" require="LoggingService">
    <!-- service requis -->
    <reference name="model" require="LoggingService">
    <!-- ... -->
  </component>
  <!-- second sous-composant -->
  <component name="Model">
    <service name="model" require="TransacService">
    <!-- ... -->
  </component>
  <wire source="View/model" target="Model/mode" />
</composite>
<composite name="SecurityService">
  <service name="serviceitf">
  <!-- ... -->
</composite>
<!-- ... définition de  LoggingIntent and TransacIntent -->
```

Listing 2.2 – Exemple de spécification d'un ensemble de composants à l'aide du formalisme proposé dans *FraSCAti*

Outre la modélisation de l'application, l'environnement *FraSCAti* propose un moteur d'exécution qui assure le déploiement et l'administration d'une application SCA. Ce moteur se décompose en trois composants principaux :

***description parser* :** il vérifie la consistance de la description SCA de l'application en vue de la création d'un modèle dynamique (*modèle@runtime*) associé. Concrètement, ce composant convertit la description XML en un modèle *Eclipse Modelling Framework* (*EMF*) [117] compatible avec le méta-modèle interne au moteur *FraSCAti*.

***Personality Factory* :** ce composant permet de définir la personnalité choisie dans la mise en œuvre d'un composant ;

***Assembly Factory* :** il s'agit de l'entité en charge de construire les composites décrits dans la spécification applicative fournie par l'utilisateur.

Enfin, *FraSCAti* est un environnement auto-déployable. En effet, le moteur d'exécution est lui-même une application SCA formée d'un assemblage de services. Ces services prennent la forme de plugins de plate-forme qui peuvent donner lieu à des mécanismes de composition dynamique. Concrètement, *FraSCAti* propose un élément d'amorçage (*bootstrap*) dont le rôle est de déployer un moteur d'exécution spécifié sous forme d'une application SCA. Un utilisateur peut donc fournir sa propre spécification du moteur d'exécution ou recourir à celle disponible dans l'implémentation de référence. Par la suite, le moteur d'exécution nouvellement instancié est capable d'administrer une spécification applicative SCA. A une application est associée une instance de moteur *FraSCAti*.

Automatisation

FraSCAti offre un environnement capable de gérer l'installation, la configuration, l'activation d'une application et, au-delà, les aspects de reconfiguration. Néanmoins, *FraSCAti* ne gère pas la phase de mise à disposition. D'autre part, l'implémentation de référence ne prend pas en compte l'établissement de liaisons entre composants s'exécutant sur des machines physiques distinctes. En effet, la solution ne gère pas les éléments de configuration dynamique dans un tel contexte. Charge à l'utilisateur de développer le *Wiring Controller* adapté.

Généricité

S'appuyant sur un modèle à composant très général et complet (i.e. Fractal), *FraSCAti* offre un très bon niveau de polyvalence.

Il est également agnostique vis-à-vis de l'environnement d'exécution applicatif, l'environnement *FraSCAti* proposant une implémentation de référence en Java, capable de manipuler des infrastructures virtuelles ou matérielles.

Grâce à son modèle d'abstraction de haut niveau et à une approche ouverte et extensible (i.e. contrôleur, moteur d'exécution), *FraSCAti* présente une adhérence applicative limitée.

Passage à l'échelle

FraSCAti associe à chaque nouvelle application à déployer une instance de moteur d'exécution qui lui est spécifique. Comme cela est illustré dans [115] ceci permet de supporter les déploiements multiples en parallèle. Néanmoins, du fait de la gestion partielle des paramètres de configuration dynamique, *FraSCAti* semble moins bien armé pour procéder à des déploiements large échelle.

Fiabilité

Hormis les vérifications statiques que le *Specification Parser* applique sur une spécification d'application SCA, la version d'implémentation de *FraSCAti* ne propose pas de mécanisme d'autoréparation ou même de tolérance aux pannes.

Granularité

La granularité d'une spécification applicative au travers du formalisme *FraSCAti* est non contrainte.

Synthèse

FraSCAti est une solution de déploiement et de reconfiguration d'applications patrimoniales conformes à l'approche SCA. En ce sens, elle étend les spécifications SCA par des capacités d'administration importante. En outre, il s'agit d'une solution ouverte, très largement extensible. Cependant, *FraSCAti* présente encore des restrictions en matière de gestion de la tolérance aux pannes, d'automatisation de bout en bout du cycle de vie de l'application et de gestion des applications large échelle.

2.2.2.3 OSGi

L'alliance OSGi propose un ensemble de spécifications qui définissent une plate-forme capable d'assurer le déploiement et l'exécution d'applications Java orientées services [120]. Basé sur l'édition standard de la technologie Java et sur un modèle à composants, cet environnement a connu un véritable engouement qui a fait l'objet d'utilisation dans des domaines aussi variés que l'informatique embarquée ou l'univers des serveurs d'applications Java EE (e.g. serveur d'applications open source JOnAS [49]).

Les spécifications OSGi se divisent en deux parties : l'une dédiée au packaging et au déploiement des composants OSGi qui sera intitulée *modèle de donnée*, l'autre se focalisant sur l'administration des composants et de leur cycle de vie et intitulée *environnement d'exécution*.

Un composant OSGi est qualifié de *bundle*. Concrètement, un *bundle* est composé d'un ensemble de classes Java regroupées dans plusieurs fichiers d'archive au format jar et enrichi de métadonnées. Un *bundle* permet de décrire deux types de dépendances :

dépendances de déploiement : un *bundle* peut fournir ou requérir des paquets java (*packages*). Un *package* fourni est appelé *export*. Un *package* requis est qualifié d'*import* ;

dépendances d'exécution : un *bundle* peut fournir ou requérir des *services* qui correspondent à des interfaces fonctionnelles.

Les métadonnées contenues dans un *bundle* permettent, d'une part, de décrire l'organisation des classes qui lui sont associées, en termes de fichiers d'archives et de la définition d'un *classpath*, d'autre part, de décrire de manière explicite les dépendances de déploiement. Un exemple de métadonnées contenues dans un *bundle* est décrit dans le listing 2.3. Concernant les dépendances d'exécution, il est possible de les détailler au sein des métadonnées, mais cette description n'est pas utilisée par l'environnement OSGi. En effet, la spécification OSGi ne précise pas la manière dont l'environnement doit résoudre les dépendances d'exécution.

```
Bundle-Name: SpringooJEEBundle
Bundle-SymbolicName:
    com.orange.vamp.examples.springoo.SpringooJEEBundle
Bundle-Description: The OSGi bundle for the businness part of the
    Springoo application
Bundle-Version: 1.0.0
Bundle-Vendor: Orange
Bundle-Activator: com.orange.vamp.examples.JEEActivator
Bundle-Classpath: ., /classes, /lib/aaa.jar
Import-Package: com.orange.vamp.examples.springoo.wrp.database;
    specification-version="1.0.0"
Export-Package: com.orange.vamp.examples.springoo.wrp.application;
    specification-version="1.0.0"
```

Listing 2.3 – Exemple de métadonnées contenues dans un *bundle OSGi*

Un environnement *OSGi* est un ensemble de *threads* exécutés au sein d'une machine virtuelle Java unique. A partir des métadonnées contenues dans les *bundles*, il assure leur déploiement et leur exécution.

Au cours de l'étape de déploiement, l'environnement OSGi tente de résoudre les dépendances de déploiement à l'aide des *exports* des *bundles* déjà déployés localement. S'il y parvient, le *bundle* est considéré comme étant *résolu*. Dans le cas contraire, il demeure dans un état *non résolu* et les *exports* qu'il contient ne sont pas pris en compte par l'environnement OSGi (i.e. le solveur de dépendance de l'environnement n'utilisera pas ces packages pour procéder à la résolution de dépendances de déploiement d'autres *bundles*). L'état résolu coïncide avec la fin d'activation. Ainsi, il n'existe pas de phase d'activation des *bundle* à proprement parler dans OSGi.

Il est à noter que la spécification OSGi décrit une entité correspondant à un magasin distant de *bundles* : l'*OSGi Bundle Repository* ou *OBR*. Cette entité, instanciée au sein d'un serveur HTTP, permet de publier des *bundles*. Elle peut également être sollicitée par un environnement OSGi lors de la résolution de dépendances de déploiement d'un bundle.

Une fois le qu'un *bundle* passe dans l'état résolu, il est désormais en mesure d'interagir avec l'environnement d'exécution OSGi au travers d'une interface de gestion de cycle de vie qu'il implémente optionnellement. Un *bundle* qui n'implémente pas cette interface ne peut pas interagir avec l'environnement d'exécution OSGi. Dès lors, il est comparable à une simple librairie de classes Java, utilisable pour la résolution des dépendances de déploiement.

Suite à la résolution des dépendances de déploiement, l'environnement OSGi invoque l'opération d'activation de l'interface de gestion du cycle de vie du *bundle*, ce qui a pour effet de lui associer un contexte d'exécution. Ce contexte regroupe l'ensemble des services fournis et requis par le *bundle*. Il propose également une opération d'enregistrement, associant la référence d'un service fourni par le *bundle* à une interface et un ensemble de métadonnées, ainsi qu'une opération de recherche qui correspond à une résolution dynamique d'une dépendance d'exécution. Il faut noter que l'ajout et le retrait dynamique de services dans l'environnement OSGi a pour conséquence que des appels successifs à la méthode de recherche d'un même service peuvent renvoyer des résultats différents.

Grâce à la mise en œuvre d'un mécanisme hiérarchisé de chargement de classe, l'environnement OSGi est capable de gérer les phases du processus du cycle de vie des *bundles* au-delà de l'étape d'activation initial. Ainsi chaque *bundle* dispose de son propre chargeur de classes, ce qui permet que des *bundles* différents utilisent des versions (voire des implémentations) différentes d'une même classe. Cette propriété permet également à l'environnement de l'adapter ou de le mettre à jour par création d'un nouveau *bundle* intégrant les versions mises à jour des classes modifiées.

Automatisation

L'environnement *OSGi* dispose d'un excellent degré d'automatisation. En effet, la spécification *OSGi* définit l'automatisation de la macro-phase de déploiement dans son intégralité. Elle couvre non seulement les aspects liés au déploiement initial mais également les phases d'arrêt et de redémarrage, de reconfiguration, de mise à jour et d'adaptation ainsi que le retrait.

Généricité

En terme de polyvalence, bien que la spécification *OSGi* ait vocation à se focaliser sur les applications Java rien n'empêche de définir des *bundles* chargés d'encapsuler des entités logicielles patrimoniales afin d'en proposer une vision unifiée. Néanmoins, le caractère mono-machine virtuelle Java de l'environnement *OSGi* rend sensiblement plus complexe le déploiement et l'administration d'applications réparties sur plusieurs hôtes. En effet, il est alors nécessaire d'intégrer la gestion de la répartition au sein des *bundles* eux-mêmes, ce qui altère notablement l'indépendance entre le système de déploiement et l'application déployée (i.e. adhérence applicative). Le spectre d'applications pouvant être déployées simplement et nativement à l'aide de l'environnement *OSGi* peut donc être considéré comme relativement limité.

La finalité de la spécification *OSGi* est de proposer un environnement capable de prendre à sa charge un maximum d'opérations d'administration d'une application, regroupant un ensemble de *bundles*. Ceci étant, le modèle de données qu'elle définit est insuffisant En effet, il n'expose pas la notion de dépendances d'exécution entre *bundles*, indispensable à la modélisation de l'architecture applicative globale (i.e. modèle de produit). Sans modèle architectural, l'utilisateur doit nécessairement recourir à des abstractions de bas niveau (e.g. scripts spécifiques) pour déployer son application. L'adhérence applicative d'*OSGi* est donc forte.

Enfin, OSGi s'appuie sur l'édition standard de l'environnement Java, ce qui le rend relativement portable. Néanmoins, comme la plupart des solutions de déploiement en environnement non virtualisé, la solution ne propose pas de mécanisme automatisé d'approvisionnement de l'environnement matériel d'exécution. Il est donc assez dépendant de l'environnement d'exécution applicatif.

Passage à l'échelle

L'ensemble des fonctionnalités proposées par la spécification *OSGi* sont regroupées,

de manière centralisée, au sein de la plate-forme de déploiement. Ainsi, c'est l'environnement de déploiement, et non les *bundles* eux-mêmes, qui assure la résolution de dépendances, qu'il s'agisse de dépendances de déploiement ou d'activation. Cela induit un risque quant à la capacité d'*OSGi* de pouvoir réaliser des déploiements large échelle. Cependant, ce constat est à pondérer avec le fait qu'*OSGi* n'a pas vocation à déployer des applications multi-sites. En matière de déploiements multiples, il semble qu'*OSGi* procède séquentiellement. Néanmoins, là encore, l'approche mono-machine virtuelle Java (et donc mono-site) tend à démontrer que la spécification *OSGi* ne cherche pas à adresser ce type de problématique.

Fiabilité

Un mécanisme de remontée d'erreurs permet à l'environnement *OSGi* de notifier l'utilisateur qu'une erreur est survenue au cours du déploiement ou de l'exécution d'un *bundle*. Néanmoins, le traitement de cette erreur est laissé à la charge du client, l'environnement n'implémentant pas de mécanisme de reprise. De manière comparable, la fiabilisation de l'environnement d'exécution lui-même incombe à l'utilisateur par l'emploi de technique de duplication et de mécanisme de haute disponibilité. Le niveau de gestion de la fiabilité proposé par la spécification *OSGi* est donc jugé insuffisant.

Granularité

La granularité d'*OSGi* est le *bundle*. La structure d'un *bundle* n'étant pas contrainte, l'utilisateur peut adapter sa taille à la granularité applicative désirée. Ainsi, il peut par la suite définir des dépendances entre bundle ne comprenant qu'une classe.

Synthèse

De par son objectif (i.e. le déploiement et l'exécution d'applications Java dans un contexte mono machine virtuelle) et bien qu'il couvre de façon complète la macro-phase de déploiement, *OSGi* ne répond pas à la majorité des requis permettant de le retenir comme environnement de déploiement fiable et autonome d'applications réparties en environnement virtualisé.

2.2.3 Solutions à base de langages de description d'architecture

Un *langage de description d'architecture* (*Architecture Description Language* ou *ADL*) désigne un formalisme permettant de décrire, de façon explicite et globale, l'architecture d'une application répartie, en fonction des composants qui la constituent et des dépendances entre ces composants [89]. Le niveau d'expressivité d'un *ADL* dépend des concepts qu'il expose, eux-mêmes dépendants du modèle de composants sous-jacent. Ainsi, la plupart des *ADL*s permettent d'exprimer des dépendances fonctionnelles entre composants (i.e. liaisons entre services fonctionnels). A l'inverse peu d'entre eux permettent d'exprimer des dépendances non-fonctionnelles (i.e. vis-à-vis d'un service d'infrastructure) ou une exigence technique (e.g. la colocalisation de deux composants, l'ordre d'activation

de composants, etc.). Les ADLs qui y font référence, utilisent pour la plupart des étiquettes (i.e. définition de la location d'un composant à l'aide de la balise *virtual-node* dans [102]) parfois insérées dans le code du composant lui-même (i.e. annotations relatives à l'utilisation de services de transaction ou de sécurité dans [3]). L'un des intérêts des *ADL*s est de faciliter la dissociation entre les aspects fonctionnels et non fonctionnels de l'application. Ils peuvent donc donner lieu à une description indépendante du code métier ou au contraire y être intégré (e.g. au moyen d'annotations).

Les *ADL*s constituent donc, dans le cadre de l'administration du cycle de vie des applications en général, et de leur déploiement en particulier, un formalisme pivot entre l'utilisateur et le système d'*ALM*. Ainsi, l'utilisateur peut définir la répartition de son application au sein d'un ensemble de composants, capturer un ensemble de métadonnées (e.g. paramètre de configuration) propres à chacun d'eux ou à l'ensemble de l'application, et définir les relations qui lient les composants entre eux. Le système d'*ALM*, quant à lui, exploite cette description architecturale pour assurer les opérations de déploiement et de supervision. Il s'agit de mettre en œuvre les composants et de résoudre les dépendances. Cette dernière peut intervenir à chaque étape du cycle de vie : statiquement avant l'instanciation des composants ou dynamiquement suite à leur installation voire après leur activation (i.e. résolution tardive de dépendance ou *late binding*). A l'inverse des *DSL*s, les *ADL*s sont des langages de spécification génériques (*General Purpose Language*s ou *GPL*s).

Cette section présente donc un ensemble de solutions (i.e. *Rainbow*, *SmartFrog*, *TUNe*, *ProActive*) s'appuyant sur la notion de langage de description d'architecture pour assurer le déploiement d'applications.

2.2.3.1 Rainbow

Rainbow est une plate-forme *open source* issue du projet DASADA DARPA de l'Université de Carnegie Mellon [45]. Il s'agit d'un système d'administration automatisée du cycle de vie des applications patrimoniales. Plus exactement, *Rainbow* gère des applications instanciées et en cours d'exécution.

Afin d'assurer l'administration automatisée d'applications, *Rainbow* s'appuie sur :

- une représentation dynamique de l'architecture applicative ;
- un ensemble de gestionnaire et de services regroupant les traitements du système d'administration.

Afin de permettre le développement de services auto-adaptables, *Rainbow* met en œuvre une représentation architecturale de chaque application administrée. Une telle représentation s'appuie sur le modèle à composants Acme [72] et sur le langage de description d'architecture qui lui est associé. Elle permet de réifier l'application qu'elle modélise. Dans un premier temps, l'utilisateur fournit un modèle de l'architecture applicative permettant de l'initialiser. Par la suite, *Rainbow* la met à jour au gré des changements réalisés dans le cadre de la gestion autonome et dans le respect de politiques d'administration. Ces dernières sont quant à elles exprimées par l'utilisateur au moyen du langage Stich.

L'architecture de *Rainbow* se subdivise en :

- un ensemble de gestionnaires (e.g. gestionnaire de modèle ou *Model Manager*, de politique ou *Strategy Executor*, d'adaptation ou *Adaptation Manager*) regroupant les fonctions d'administration élémentaires et formant le cœur de la solution. Ainsi, le gestionnaire de modèle regroupe les opérations de consultation et de modification applicables à une représentation applicative. Il s'appuie pour cela sur un canevas de remontées de données formé de sondes.

- un ensemble de services personnalisables. Ils composent les entités personnalisables du système *Rainbow*. Il est donc possible d'ajouter de nouveaux services en fonction du type d'application à administrer. Les services sont modélisés sous forme de composants Acme manipulables par l'utilisateur et les gestionnaires au travers du modèle d'application.

Automatisation

Bien que *Rainbow* soit une solution dédiée à l'automatisation automatisée du cycle de vie d'application patrimoniale, il ne se focalise que sur les phases postérieures au déploiement. Ainsi, il ne couvre pas les phases de mise à disposition, d'installation, de configuration et d'activation de l'application.

Généricité

Rainbow s'appuyant sur une modélisation à base de composants (i.e. le modèle Acme) et du langage de description d'architecture associé, il demeure indépendant du domaine métier des applications qu'il administre ainsi que des choix techniques ou architecturaux qui leur sont associés. Sa polyvalence s'avère donc très satisfaisante. L'administration de quatre applications de nature différente est illustrée dans [44] : une application basique composée d'un client et d'un serveur, un système de vidéoconférence (*Libra*), une plate-forme temps réelle ubiquitaire de mise à disposition d'informations à destination d'un ensemble d'étudiants (*Grade*) et une plate-forme dédiée à la mise en œuvre de réseaux sociaux s'appuyant sur un support vocal à base de multidiffusion ou *multicast* (*TalkShoe*).

Ces diverses expérimentations laissent à penser que l'indépendance de *Rainbow* vis-à-vis de l'environnement d'exécution applicatif est également satisfaisant. Néanmoins, cette propriété resterait à vérifier dans un contexte de déploiement, en fonction de la capacité de *Rainbow* à approvisionner l'environnement matériel d'exécution de l'application.

Enfin, concernant l'adhérence applicative, *Rainbow* implémente les comportements de la plate-forme au moyen de services personnalisables. Les fonctionnalités de l'environnement d'administration peuvent donc être enrichies sans que le code développé ne soit qu'à destination d'une application donnée. Néanmoins, le processus d'exécution de *Rainbow* demeure codé de façon définitive et il n'est pas possible de le faire évoluer. Cette restriction demeure toutefois raisonnable au regard des possibilités offertes par la plupart des autres solutions présentées dans cet état de l'art.

Passage à l'échelle

L'architecture de *Rainbow* est globalement centralisée. En effet, à l'exception des sondes en charge d'assurer la remontée d'information au niveau des gestionnaires et des services, toute la logique du système d'administration demeure centralisée. Comme le souligne [44], ce choix architectural soulève de réelles difficultés quant à la gestion large échelle proposée par la solution.

Fiabilité

De même, [44] souligne la criticité de la centralisation des traitements en terme de fiabilité, aucun mécanisme de tolérance aux pannes n'étant mis en œuvre.

Granularité

Rainbow s'appuie sur le modèle à composant qui n'induit pas de contraintes particulières quant à la granularité des unités fonctionnelles manipulées. Ceci lui offre un facteur de granularité lui permettant de déployer n'importe quelle application patrimoniale.

Synthèse

Bien que *Rainbow* présente des concepts (i.e. services personnalisables) et des principes (i.e. extensibilité, généricité, autonomie) intéressants dans le cadre de l'administration autonome du cycle de vie des applications, elle n'en demeure pas moins inadaptée pour les aspects relatifs au déploiement. En outre, l'architecture centralisée des gestionnaires constituant son cœur la rendent peu adaptée à la gestion d'applications large échelle ou de déploiements multiples. Enfin, rien n'est proposé quant à la fiabilisation de l'environnement d'exécution applicatif ou de l'environnement *Rainbow* eux-mêmes.

2.2.3.2 SmartFrog

SmartFrog (pour *Smart Framework of Objects Group*) est un projet *open source* issu des laboratoires de recherche de *Hewlett Packard*, dont la finalité est d'assurer l'auto-configuration des applications patrimoniales réparties [73]. *SmartFrog* est un canevas extensible composé, d'une part, d'un modèle et, d'autre part, d'un environnement d'exécution développé en Java.

Le modèle défini dans *SmartFrog* s'appuie sur la notion de composants. Un composant *SmartFrog* correspond à l'abstraction d'un élément patrimonial. Il est constitué de trois parties :

- un ensemble de données de configuration accessibles en lecture et écriture au moyen d'interfaces programmatiques ;
- un gestionnaire de cycle de vie de l'élément applicatif encapsulé et pilotable programmatiquement ;
- un *élément administré* qui implémente les fonctionnalités du composant en lien avec l'élément applicatif encapsulé.

Données de configuration A une donnée de configuration peut être associée une valeur simple (i.e. explicite) s'il s'agit d'une donnée relative au composant considéré ou la *référence* d'une donnée d'un autre composant. Une *référence* désigne en effet le nom absolu d'une donnée de configuration d'un composant. Elle peut être utilisée au sein d'un autre composant pour accéder à la valeur de la donnée[5]. Il est possible d'associer un ensemble de contraintes à une référence. Dans ce cas, celles-ci doivent être vérifiées avant que la référence ne puisse être résolue.

A chaque donnée de configuration d'un composant est associée une phase d'initialisation permettant de définir le moment où la valeur de la donnée sera renseignée par l'environnement d'exécution de *SmartFrog* (e.g. statiquement à la définition du composant ou dynamiquement lors de son instanciation dans une approche de type *late binding*).

Le modèle défini dans *SmartFrog* introduit deux mécanismes d'assemblage des composants : la *composition* et l'*extension*. La composition procède par duplication de définition. Elle consiste à définir explicitement un composant comme étant une donnée de configuration d'un autre composant. L'extension, quant à elle, procède par référencement. Elle consiste à positionner la valeur d'une donnée de configuration du composant englobant à l'aide de la référence à un composant englobé préalablement défini.

Le listing 2.4 illustre le modèle de données de *SmartFrog* au travers d'un exemple. Il définit une application *ApplicationSample* résultant de la composition de 5 autres données (i.e. *serverPort*, *serverSample*, *serverClassSample*, *serverImplSample* et *clientSample*). *serverPort* est une donnée simple qui fait référence à la valeur du port du serveur. Celle-ci est définie en tant que donnée simple (i.e. *sPort*) dans *serverSample*. Elle est initialisée dynamiquement (i.e. gestion de liaison tardive signalée au moyen du mot clef *LAZY*) à l'aide de la variable d'environnement *port*. En outre, la donnée *serverClassSample* définit statiquement, à l'aide de son attribut *class* la classe d'implémentation du serveur. La donnée *serverImplSample* résulte de la composition d'une valeur de port et d'une classe d'implémentation. Enfin, la donnée *clientSample* dispose d'un attribut *accessPort* qui fait référence à la donnée de *ApplicationSample* faisant référence à cette valeur, à savoir *serverPort* et initialisé comme expliqué précédemment.

```
applicationSample extends {
  serverPort serverSample:sPort;
  serverSample extends {
    sPort LAZY ENV port;
  }
  serverClassSample extends {
    class "org.smartfrog.samples,ServerSample";
  }
  serverImplSample extends serverSample, serverClassSample;
  clientSample extends {
    accessPort serverPort;
  }
```

[5]Le principal intérêt du mécanisme de référence est de limiter la duplication de données et la mise en cohérence qu'elle induit

```
}
```

Listing 2.4 – Exemple de descripteur de configuration *SmartFrog*

Gestionnaire de cycle de vie Afin de gérer correctement l'enchainement dans la résolution des dépendances de configuration, *SmartFrog* offre la possibilité de définir le cycle de vie de configuration de chaque composant. Ainsi chaque composant comprend un sous-composant correspondant au gestionnaire de cycle de vie. Ce dernier comprend un attribut qui désigne une classe Java qui implémente le comportement du composant. Une telle classe répond à une interface permettant ainsi de définir les traitements à réaliser lors de la création du composant, son initialisation, son démarrage, son arrêt, la réception de notification d'échec ou de consultation du statut.

SmartFrog permet de définir des regroupements de composants et d'y associer un gestionnaire de cycle de vie. Celui-ci peut alors implémenter des comportements indépendants ou liés entre les gestionnaires de cycle de vie de chacun des composants qui font partie du regroupement.

Environnement d'exécution

Le rôle de l'environnement d'exécution de *SmartFrog* est de procéder à l'instanciation, la configuration puis l'activation automatique d'applications patrimoniales. Pour cela, à partir d'une modélisation de l'application selon le formalisme présenté précédemment, il met en œuvre le cycle de vie de chacun des composants qui la composent. Chacun d'eux est déployé au sein d'un processus léger (i.e. *thread*) dans une machine virtuelle Java. L'environnement d'exécution lui-même se compose d'une entité centrale, appelé *SFInstaller*, et d'un ensemble de démons d'administration réparties sur les machines qui constituent l'environnement d'exécution applicatif. Afin d'assurer la résolution de dépendance entre composants (i.e. la résolution des références), l'environnement d'exécution *SmartFrog* propose un système de nommage. Son implémentation peut s'appuyer sur des entités tierces (i.e. registres distribués) ou sur l'implémentation d'un protocole de type *service de nommage*.

Automatisation

Bien que *SmartFrog* présente un bon niveau d'automatisation des phases l'instanciation, l'initialisation et le démarrage des composants et les transitions entre ces étapes, il n'adresse pas la problématique de la mise à disposition de l'application à déployer.

Généricité

Concernant la polyvalence, *SmartFrog* couvre un spectre d'applications qui peut être jugé optimal dans la mesure où aucune restriction n'est faite quant au type d'application déployée. Ceci résulte notamment de la faible adhérence de *SmartFrog* vis-à-vis de l'application à déployer, grâce à l'utilisation d'un produit basé sur un modèle à composants fortement générique. Bien que, dans sa version initiale, aucun modèle ne permette de décrire l'environnement d'exécution applicatif (modèle de site), certains travaux tentent

de remédier à ce manque en couplant *SmartFrog* avec des approches par synchronisation de recette. C'est par exemple le cas de [17] qui étudie les gains apportés par l'utilisation de *LCFG* dans *SmartFrog*. Enfin, l'utilisation d'un modèle à composant générique, dans lequel un composant n'est qu'un *wrapper* de l'entité applicative qu'il encapsule, couplée à une implémentation en Java, rend *SmarFrog* relativement portable. Néanmoins, comme la plupart des solutions de déploiement en environnement non virtualisé, la solution ne propose pas de mécanisme automatisé d'approvisionnement de l'environnement matériel d'exécution. Son indépendance vis-à-vis de l'environnement d'exécution n'est donc pas optimale. De plus, elle est également altérée du fait de la limitation de l'expressivité du modèle. En effet, l'entrelacement de la notion d'attributs et de liaisons au niveau des données de configuration d'un composant rendent plus difficile la dissociation entre un composant et son implémentation.

Passage à l'échelle

SmartFrog adopte une architecture répartie sur plusieurs entités d'administration. Néanmoins, l'orchestration du déploiement est assurée de façon centrale par le *SFInstaller* ce qui limite sa capacité à assurer des déploiements multiples. De plus, pour implémenter les opérations d'élaboration tardive de liaisons (*late binding*) et de référencement de composants distants, il s'appuie sur des mécanismes de découverte et de localisation de composants basés sur des protocoles de type localisation de service, selon le patron *advertiser-locator*. Or, ce type de patron nécessite l'utilisation d'une entité tierce de référencement dont le dimensionnement n'est pas pris en charge par *SmartFrog* lui-même. Ceci induit que la gestion d'applications de grande taille demeure problématique.

Fiabilité

La fiabilité constitue l'une des principales faiblesses de *SmartFrog*. En effet le système met en œuvre un mécanisme de remontée d'erreur qui permet à un composant d'être informé de la survenue d'une erreur au cours d'une opération de configuration. Néanmoins, le traitement de cette erreur est à la charge de l'utilisateur, l'environnement n'implémentant pas de mécanisme de reprise. De manière comparable, la fiabilisation de l'environnement d'exécution lui-même incombe à l'utilisateur par l'emploi de technique de duplication et de mécanisme de haute disponibilité.

Granularité

SmartFrog définit un modèle à composants qui n'induit aucune restriction quant à la granularité avec laquelle une application doit être modélisée.

Synthèse

Bien qu'il présente un nombre important d'atouts et qu'il fasse l'objet de collaboration avec le CERN et un laboratoire de l'université d'Edimbourg, pour le coupler à une solution de création et de démarrage de machines virtuelles, en vue d'offrir des environnements de recettes, *SmartFrog* demeure trop en retrait vis-à-vis de la fiabilité pour offrir un fonctionnement totalement autonome.

2.2.3.3 TUNe

TUNe, acronyme de *Toulouse University Network*, est un système autonome d'administration (*SAA*) développé depuis 2008 par l'équipe Astre de l'Institut de Recherche en Informatique de Toulouse (IRIT) [34]. Successeur de l'environnement *Jade* [32], l'un des premiers *SAA* capable de déployer des applications patrimoniales, *TUNe* vise à en surmonter le principal défaut, à savoir que *Jade* s'adresse à des utilisateurs familiers avec la programmation à composants. En effet, *Jade* et *TUNe* s'appuient tous deux sur le modèle à composant *Fractal* pour modéliser l'application administrée. Pour faciliter son utilisation et s'abstraire ainsi des interfaces programmatiques de bas niveau, *TUNe* s'appuie donc sur la définition de langages de haut niveau basés sur les profils UML [99].

TUNe définit trois langages couvrant les notions de modèle de site, de produit et de politiques définies par [41] :

***Architecture Description Language* (*ADL*) :** s'appuyant sur le diagramme de classe défini par l'*UML*, *TUNe* permet de décrire l'architecture de l'application répartie administrée ainsi que la plate-forme matérielle sur laquelle elle sera déployée. Ainsi, un premier diagramme représente chaque type d'entité logicielle qui compose l'application au moyen d'une classe. Celle-ci regroupe un ensemble d'attributs. Certains d'entre eux sont propres au type d'élément logiciel considéré (e.g. le port d'écoute pour un serveur HTTP) alors que d'autres sont en lien direct avec le fonctionnement du *SAA* (e.g. pour indiquer le nombre d'instances du type à déployer initialement, ou d'indiquer la localisation d'une instance de l'entité logicielle). *TUNe* permet également de modéliser l'environnement matériel (i.e. environnement d'exécution de l'application) au moyen d'un second diagramme de classes. Une classe est alors vue comme un regroupement de machines (*cluster*) présentant des propriétés identiques (e.g. même distribution et même version de système d'exploitation). Un cluster ne comporte que des attributs requis et imposés par le *SAA*. Pour chacun de ces diagrammes de classes, l'*ADL* offre la possibilité de définir des interconnexions entre classes, selon la sémantique proposée par le formalisme *UML*. Dans le cas des types de logiciels, elles définissent des dépendances entre des attributs alors que dans le cas des clusters elles correspondent à des relations d'héritage, en vue de la factorisation des définitions. Le nommage des interconnexions, des composants et de leurs attributs permet au *SAA* d'identifier n'importe quel élément de l'architecture applicative.

L'*ADL TUNe* est un modèle par intention, c'est-à-dire qu'il ne propose pas une description exhaustive de l'architecture applicative, en ce sens qu'il définit des types de logiciels et non des instances. C'est donc le *SAA* qui détermine la manière de projeter l'architecture applicative sur la plate-forme matérielle. Cette opération consiste à instancier le nombre désiré de chaque type de logiciel et à mettre en œuvre les liaisons concrètes entre ces instances dans le respect du modèle proposé (i.e. des cardinalités exprimées au niveau des interconnexions entre types de logiciels). Le résultat de cette projection est une représentation concrète et exhaustive de l'application interne à *TUNe* et appelée représentation système (*system repre-*

sentation ou *SR*).

Il est à noter que toutes les instances d'un même type de logiciel sont obligatoirement déployées dans sur les machines d'un seul et même *cluster*.

***Wrapping Description Language* (*WDL*) :** ce formalisme est utilisé pour définir l'ensemble des opérations supportées par chaque type de logiciel. Ainsi, une opération est définie à l'aide d'un nom, d'une méthode java qui l'implémente, et d'un ensemble de paramètres. La valeur d'un paramètre peut être connue statiquement (i.e. constante) ou dynamiquement. Dans ce dernier cas, il s'agit d'une référence à un élément de l'architecture applicative exprimé selon le système de nommage décrit plus haut. La résolution de cette dépendance est réalisée par le *SAA* lors du déploiement.

***Resolution Description Language* (*RDL*) :** il s'agit du formalisme de définition des politiques d'administration, selon le principe des diagrammes d'états-transitions définis par *UML*. Chaque état correspond soit à la modification d'attributs des entités définies au travers de l'*ADL* ou à l'exécution d'une opération décrite dans le *WDL*. En outre, *TUNe* propose deux opérations supplémentaires, permettant d'ajouter et de retirer des instances de logiciels. *TUNe* prédéfinit deux politiques d'administration sur chaque type de logiciel : le démarrage et l'arrêt. Le principe de parallélisation de tâches est exprimée à l'aide des opérations standards *fork* (i.e. exécutions parallèles) et *join* (i.e. synchronisation de plusieurs exécutions).

D'un point de vue architectural, *TUNe* est composée d'une entité centrale, en charge d'appliquer la logique d'administration et d'un ensemble d'agents répartis sur la plate-forme matérielle. Ainsi, sur chaque machine sont instanciés deux types d'agent :

- un agent appelé *RemoteLauncher* en charge des opérations générales non spécifiques aux entités applicatives locales à la machine ;
- un ensemble d'agents appelés *RemoteWrapper*. Chacun d'eux est en charge de mettre en œuvre les opérations spécifiques à une instance de logicielle donnée.

Automatisation

De façon native, *TUNe* automatise l'ensemble des étapes du processus de déploiement de l'application. Dans un premier temps, à partir des descriptions fournies par l'utilisateur, il calcule la projection du modèle de l'application sur le modèle de plate-forme matérielle. En fonction du résultat obtenu, il crée sa représentation système interne, ce qui a pour effet de déclencher la transmission des binaires de l'application au niveau des agents *RemoteLauncher*. Le composant central contacte alors ces mêmes agents pour qu'ils procèdent à la création des agents dédiés à une entité logicielle donnée (i.e. les *RemoteWrapper*s). Le composant central peut alors interagir avec ces derniers pour orchestrer la configuration puis l'activation des logiciels.

Généricité

Basé sur une modélisation UML, *TUNe* ne présente pas de limitations particulières quant au type d'application qu'il est capable de déployer.

Néanmoins, la notion de *cluster* introduite dans sa définition de plate-forme matérielle, est contraignante. En effet, à l'inverse des types de logiciels qui peuvent être instanciés plusieurs fois, un cluster ne fait l'objet que d'une instanciation regroupant l'ensemble des machines qu'il comprend. Dans un contexte de virtualisation, dans lequel il est nécessaire de décrire une machine virtuelle tant en termes de caractéristiques matérielles que des logiciels qui la composent, ceci signifie que chaque cluster devrait être réduit à une machine.

Enfin, le mécanisme d'approvisionnement de *TUNe* demeure statique au sens où les machines d'un cluster sont disponibles avant le déploiement de l'application. Or, le packaging de l'application selon un format prédéfini, ne permet pas d'utiliser *TUNe* dans un environnement d'exécution applicatif basé sur des machines virtuelles.

Passage à l'échelle

Bien que *TUNe* soit, en partie, composé d'entités réparties sur les machines de l'environnement d'exécution de l'application, il n'en demeure pas moins que toute la logique d'orchestration du déploiement est centralisée. Ainsi la gestion d'applications large échelle demeure imparfaitement couverte par *TUNe*. Cette limitation a été identifiée au travers d'expérimentation consistant à déployer l'application Diet [40]. Diet est une application large échelle de calcul scientifique sur grilles de calcul, dont la structure hiérarchique et modulaire lui permet de supporter un grand nombre de n*oe*uds de calcul

De plus, au sein du moteur d'exécution *TUNe*, aucun mécanisme n'est mis en œuvre pour réaliser des déploiements multiples.

Fiabilité

Dans sa distribution standard, *TUNe* ne propose pas de mécanismes de gestion des pannes franches affectant la plate-forme matérielle ou le *SAA* lui-même.

Granularité

En terme de granularité, l'ADL proposé par *TUNe* demeure suffisamment général pour ne pas imposer quant à la granularité des unités de déploiement.

Synthèse

TUNe est une solution de *SAA* ouverte et extensible qui peut être enrichie de nombreuses fonctions d'administration. Les outils qu'il propose en font un outil utilisable par un large panel d'utilisateurs. Néanmoins certains choix, tels que la forme du packaging des applications ou le regroupement de machines en *cluster*, raisonnables dans un souci de facilité d'utilisation, s'avèrent contraignants, notamment vis-à-vis de la généricité de la solution.

2.2.3.4 ProActive

ProActive [23] est une solution développée au sein de l'équipe OASIS de l'INRIA - CNRS I3S - UNS. Il s'agit de l'implémentation de référence en Java du modèle à composants *Grid Component Model* (*GCM*) d'aide à la définition et la mise en œuvre d'applications réparties sur des grilles [2].

Dans l'environnement ProActive, un composant est appelé *objet actif* auquel est associé un *thread* d'exécution dédié. Un objet actif correspond à une structure transparente d'encapsulation d'un objet Java standard. Il se compose d'un *body* qui constitue le point d'accès unique à l'objet encapsulé appelé, pour sa part, *objet racine*. A l'objet *body* sont associés un certain nombre de méta-objets, en charge d'implémenter les comportements non-fonctionnels associés à l'objet actif (e.g. gestion des transactions, sécurité). L'un des premiers comportements implémentés de la sorte est la gestion de l'accès distant à l'objet depuis une autre machine virtuelle Java. Ainsi, sur le principe d'approches telles que *RMI* (*Remote Method Invocation*), lorsqu'un objet A souhaite accéder à un objet actif B distant, ProActive crée automatiquement un *stub* compatible avec B et permettant à A d'interagir avec ce dernier. Cependant, ProActive se singularise des mécanismes d'invocation à distance existants, par le fait que :

- la structure d'objet actif est totalement transparente pour le développeur et ne nécessite pas que celui-ci se conforme à un quelconque modèle de programmation, ni même que les objets qu'il développe n'implémentent une interface particulière ;
- le modèle de communication sur lequel s'appuie ProActive est asynchrone. Les objets actifs communiquent entre eux au moyen de messages. Etant donné qu'un objet actif ne dispose que d'un fil d'exécution (i.e. il traite séquentiellement les requêtes qui lui sont adressées), l'un des méta-objets d'un *body* est une file ou *queue* responsable du stockage des requêtes reçues par l'objet actif.

GCM constitue une extension du modèle à composants Fractal [25]. Il en reprend les notions de composants, d'interfaces, de liaisons entre ces interfaces, de récursivité et d'introspection. En outre il les complète par :

- la capacité à spécifier la manière de distribuer sur une infrastructure les composants qui constituent une application ;
- le support des communications de groupe ;
- une distinction claire entre les aspects fonctionnels et non-fonctionnels.

Dans ProActive le modèle *GCM* est exposé à l'utilisateur au moyen d'un *ADL*. Celui-ci se décompose en :

- un descripteur contenant l'architecture de l'application (*GCMA*). Celle-ci inclut, entre autres choses, la définition abstraite de la répartition des composants sur l'infrastructure de déploiement ;

- un descripteur de ressources qui contient la description de l'infrastructure sur laquelle l'application doit être instanciée. Cette description permet ainsi de recenser les machines qui composent l'infrastructure, leur organisation en termes de *clusters*, de *bridges* (i.e. frontaux de *clusters* ou de machines) ainsi que, dans le cas de solutions de type informatique dans le nuage ou grilles, la façon dont les ressources sont approvisionnées.

Automatisation

ProActive est une solution ouverte et extrêmement flexible qui permet d'étendre aisément les comportements non-fonctionnels associés aux *objets actifs* qu'il définit. Dans sa version courante, ProActive assure l'automatisation de l'ensemble du processus de déploiement de l'application, même si l'installation des machines virtuelles est une opération réalisée à chaud, comme dans le cas de Puppet (cf. section 2.2.1.1). Ainsi, les images ne sont pas approvisionnées avant que les machines virtuelles ne soient démarrées et l'installation est réalisée par l'intermédiaire des mécanismes standards d'installation de paquets.

Généricité

Le modèle *GCM* s'appuie sur le modèle Fractal dont il reprend les principes de base de composants, d'interfaces et de liaisons entre interfaces. Ceci lui permet de décrire n'importe quel type d'applications patrimoniales et de limiter l'adhérence applicative.

En outre, le modèle de ressources qu'il expose couplée à une implémentation Java lui confèrent une bonne indépendance vis-à-vis de l'environnement d'exécution applicatif.

Passage à l'échelle

Toute nouvelle demande de déploiement, induit la création d'un gestionnaire de ressources dédié. Ainsi ProActive est capable d'assurer convenablement des déploiements multiples.

Néanmoins, malgré l'importante utilisation de ProActive dans des infrastructures de grande taille de type grille, le caractère centralisé d'un tel gestionnaire de ressources peut laisser croire que la gestion large échelle demeure l'une des limites de l'environnement.

Fiabilité

Bien que le mécanisme d'allocation de ressource de ProActive soit en partie tolérant aux pannes (i.e. cas d'une panne de machine dans un *cluster*), la solution ne répare pas d'elles mêmes les défaillances de l'environnement d'exécution de l'application. Elle n'est pas non plus autoréparable.

Granularité

ProActive s'appuie sur le modèle *GCM* qui n'impose aucune contrainte quant à la granularité des composants manipulés.

Synthèse

ProActive est une solution ouverte et extensible dont la principale force est de permettre de modéliser n'importe quelle application virtualisable et de la déployer, de façon répartie, sur des environnements hétérogènes de type *clusters* privés, grilles voire plates-formes d'*IaaS* [16]. De part son expérience des environnements de grande taille, ProActive offre des atouts intéressants en terme de passage à l'échelle. Ses principales limitations portent sur sa gestion de la fiabilité, qu'il s'agisse de la capacité d'autoréparation de l'environnement d'exécution applicatif ou du système d'administration lui-même.

2.3 Solutions en environnements virtualisés

Cette section a pour but de présenter les principaux types de contributions relatives au déploiement d'applications réparties en environnement virtualisé. Pour cela, elle s'appuie sur un certain nombre de travaux existants jugés représentatifs. Ceux-ci font l'objet d'une caractérisation vis-à-vis des propriétés requises (cf. section 1.2.2) en vue de la réalisation d'une solution de déploiement autonome et fiable d'applications arbitraires dans un environnement de type informatique dans le nuage.

2.3.1 Solutions orientées infrastructure

Les solutions orientées infrastructure envisagent le déploiement d'une application dans le nuage au travers de la mise en œuvre (i.e. approvisionnement, installation, activation) d'un ensemble de ressources matérielles virtualisées. Elles se présentent sous forme de *cloud* publique ou privé. Un *cloud* publique désigne un environnement, mis à disposition par un fournisseur, et accessible au travers d'Internet [15] [14]. A l'inverse, un *cloud* privé désigne un environnement installé et maîtrisé par un utilisateur pour ses propres besoins [84] [106] [8] [122]. Il existe un troisième type de solution appelé *cloud* hybride ou *clouds* multiples. Il s'agit de solutions capables de proposer l'utilisation parallèle de solutions de *cloud* privé et de *cloud* privé [51].

La suite de cette section va présenter *Eucalyptus* qui est une solution représentative des fonctionnalités proposées par la plupart des solutions orientées infrastructure. Cet exemple permettra de mettre en exergue les limites de ce type de solutions vis-à-vis de la gestion automatisée du cycle de vie des applications et plus particulièrement de leur déploiement.

2.3.1.1 Eucalyptus

Eucalyptus est une plate-forme libre (*open source*) d'*IaaS* [98]. Ce projet a vu le jour en 2007 à l'Université de Californie Santa Barbara (Etats-Unis). A l'époque, la plupart des offres d'*IaaS* étaient commerciales et propriétaires. Le but d'*Eucalyptus* était donc de mettre à disposition des équipes académiques un outil permettant d'étudier les thématiques de recherche issues de l'informatique dans le nuage. Ainsi, *Eucalyptus* permet d'instancier une plate-forme d'*IaaS*. Cependant, il propose également un environnement d'évaluation publique [66].

En tant que solution d'*IaaS*, *Eucalyptus* propose à ses utilisateurs :

- des fonctions d'approvisionnement de ressources virtualisées de traitement (i.e. machines virtuelles), de stockage (e.g. pour la conservation des données utilisateurs ou de celles utilisées par les machines virtuelles elles-mêmes) et de communication (i.e. réseaux virtuels) ;
- des outils graphiques et des interfaces programmatiques de pilotage et d'administration de pilotage de la plate-forme ;
- la capacité à définir et à mettre en œuvre des *contrats de niveau de services* (*Service Level Agreement* ou *SLA*) entre la plate-forme d'*IaaS* et son environnement (e.g. un utilisateur, une autre plate-forme d'*IaaS*).

Un environnement *Eucalyptus* est déployé sur un ensemble de machines physiques. Celles sur lesquelles seront déployées des machines virtuelles sont appelées des *nœuds*. Ainsi, chaque nœud *Eucalyptus* met en œuvre un système de virtualisation (i.e. hyperviseur) ; dans sa version actuelle, la plate-forme fonctionne avec les hyperviseurs Xen, KVM et VMWare. Les nœuds sont regroupés en *clusters*. Un *cloud* correspond à un ensemble de *clusters*. De manière semblable à cette organisation, les fonctionnalités proposées par *Eucalyptus* se répartissent au sein de contrôleurs organisés selon une architecture hiérarchique, modulaire et flexible. Afin d'assurer un couplage lâche entre les contrôleurs, chacun d'eux expose ses comportements au moyen d'un service web. *Eucalyptus* propose 4 types de contrôleurs :

***Node Controller* (*NC*) :** il s'agit du contrôleur de plus bas niveau. Il existe un représentant de *NC* sur chaque nœud de l'infrastructure physique. Un *NC* effectue les opérations élémentaires d'administration (i.e. instanciation, destruction) des machines virtuelles dont il a la responsabilité (i.e. celles déployées sur le nœud considéré). Il est également en charge de collecter les informations relatives aux caractéristiques physiques des machines virtuelles. Enfin, un *NC* gère un cache d'images lui permettant d'améliorer sa rapidité à instancier une machine virtuelle, sans recourir nécessairement à un référentiel d'images distant.

***Cluster Controller* (*CC*) :** il s'agit d'une entité d'administration intermédiaire. Un *CC* est en charge d'organiser la répartition des machines virtuelles sur les *NC*s s'exécutant sur les nœuds du *cluster*. Pour cela il assure le routage des requêtes utilisateurs vers les *NC*s. D'autre part, il collecte les informations remontées par les *NC*s du *cluster*.

***Storage Controller* (*Walrus*) :** il s'agit de l'un des deux contrôleurs de plus haut niveau. Il administre un service de stockage compatible avec l'API S3 [14]. Il assure donc l'accès en lecture et en écriture à des données utilisateurs mais peut également jouer le rôle de référentiel d'images.

***Cloud Controller* (*CLC*) :** il constitue le point d'accès unique à l'environnement *Eucalyptus*, pour les utilisateurs et l'administrateur. Le *CLC* propose un ensemble de

services de manipulations de ressources virtuelles (compatible avec l'API Amazon EC2 [15]) et physiques, de définition de propriétés de configuration et d'administration de la plate-forme (e.g. définition de *SLA*, de politique de sécurité, etc.).

Automatisation

A l'exception de la phase de packaging (i.e. génération des machines virtuelles), le niveau d'automatisation proposé par *Eucalyptus* couvre l'ensemble des phases du processus de déploiement, depuis la mise à disposition des images au travers de *Walrus*, jusqu'à l'activation des machines virtuelles par les *NC*s en passant par leur installation et leur configuration préalable.

Généricité En tant que plate-forme d'IaaS, *Eucalyptus* peut être vu comme un système d'exploitation dans le nuage, s'appuyant sur une couche de virtualisation. En ce sens, les images qu'il instancie au sein de machines virtuelles, ne présentent pas de caractéristiques particulières, mis à part leur format. Ainsi, *Eucalyptus* dispose d'une bonne polyvalence vis-à-vis des applications déployées.

De plus, *Eucalyptus* supporte déjà les hyperviseurs Xen, KVM et VMWare qui font partie des environnements phares de virtualisation. Tous trois supportent un vaste choix de systèmes d'exploitations (e.g. Windows, Linux, ...). La solution offre donc une bonne indépendance vis-à-vis de l'environnement d'exécution.

En revanche, le rôle d'*Eucalyptus* étant cantonné à l'approvisionnement des éléments d'infrastructure virtualisées, la plate-forme ne propose aucun modèle de description d'une application à déployer.

Passage à l'échelle

Afin d'offrir une bonne capacité à déployer simultanément un grand nombre de machines virtuelles, l'architecture proposée par *Eucalyptus* est suffisamment modulaire et flexible. En effet, à l'exception du *CLC*, tous les autres contrôleurs peuvent être déployées au travers de plusieurs instances. En outre, les *NC*s mettent en œuvre des mécanismes de caches d'images pour diminuer les latences liées au réseau. Il n'en demeure pas moins que, lors de déploiements multiples ou d'applications de grande taille, la gestion d'un cache d'images peut s'avérer insuffisante et nécessite le recours à un mécanisme de pré-approvisionnement tel que peut en proposer Cloud Foundry BOSH [1].

Fiabilité

Lors du déploiement de toute nouvelle machine virtuelle, *Eucalyptus* interroge les *NC*s au travers des *CC*s pour définir le nœud sur lequel la machine virtuelle va être instanciée. Ainsi, le caractère dynamique de cette phase de sélection du *NC* cible permet à *Eucalyptus* de disposer d'une vue actualisée des *NC*s et des *CC*s en état de fonctionnement. Ceci lui permet de s'affranchir des pannes franches qui pourraient affecter ces contrôleurs. En revanche, il ne propose pas la fiabilisation du reste du système et ne propose pas de mécanisme d'autoréparation.

D'autre part, *Eucalyptus* ne propose pas non plus de mécanisme de remplacement d'une machine virtuelle défaillante.

Granularité

La granularité proposée par *Eucalyptus* est du niveau de la machine virtuelle. Certains systèmes [84] [51] [59] introduisent également le concept de groupe de machines virtuelles au sein duquel des interdépendances entre machines virtuelles peuvent être exprimées. Néanmoins, la granularité n'en est pas pour autant affinée.

Synthèse

Comme l'ensemble des principales plates-formes d'*IaaS*, *Eucalyptus* s'avère être très adaptée à l'approvisionnement automatisé de ressources matérielles virtualisées. De telles solutions supportent l'exécution d'applications patrimoniales dans des environnements d'exécution arbitraires. Néanmoins, une granularité insuffisante des modèles de données qu'elles manipulent les empêchent d'assurer l'administration, et plus particulièrement, le déploiement des dites applications. Globalement bien adaptée à la gestion des aspects large échelle, elles n'offrent encore que peu de capacités à résister aux pannes.

2.3.2 Solutions orientées service

Les solutions orientées service désignent des plates-formes dont l'approche consiste à envisager la mise en œuvre d'une application répartie comme l'assemblage d'un ensemble de services de haut niveau, et son administration comme l'orchestration de ces services selon des politiques définies par l'utilisateur au moyen de contrats de niveaux de services (*Service Level Agreement* ou *SLA*). Les services sont mis à disposition de l'utilisateur soit par la plate-forme elle-même, soit par des fournisseurs externes. Afin de masquer, à l'utilisateur la vue de l'environnement d'exécution de son application, les services sont capables d'adapter leur comportement aux changements dynamiques qui affectent celui-ci. Toutes les applications orientées service présentent la particularité de se focaliser sur un domaine métier (e.g. la gestion de la relation cliente [110]), sur un type d'architecture (e.g. applications web JEE [1] [74] [13]) ou sur une technologie (e.g. contexte d'exécution Microsoft [92]). En réduisant le spectre des applications mises en œuvre, elles parviennent ainsi à proposer un niveau élevé d'automatisation dans la gestion de leur cycle de vie.

La suite de cette section présente *AppScale* qui est un clone fonctionnel de l'une des premières solutions orientées services qui compte parmi les leaders du marché : *Google App Engine*.

2.3.2.1 AppScale

Tout comme *Eucalyptus*, *AppScale* est un système qui a vu le jour, en 2009, dans le département informatique de l'Université de Californie Santa Barbara (Etats-Unis) [48]. A l'instar d'*Eucalyptus*, qui dans le contexte de l'*IaaS*, tend à offrir une plate-forme ouverte et iso-fonctionnelle par rapport à l'un des acteurs clef du marché (i.e. Amazon EC2/S3),

AppScale est un système open source de *PaaS* visant à proposer les mêmes fonctionnalités que l'un des produits phare de ce secteur (i.e. *Google App Engine* ou *GAE* [74]). Outre sa volonté commune avec *Eucalyptus* d'offrir de formidables outils d'études de recherche dans le contexte de l'informatique dans le nuage, *AppScale* a également pour ambition de proposer un environnement agnostique à la couche d'infrastructure virtualisée sur laquelle il s'appuie (e.g. cluster d'hyperviseurs Xen, Amazon EC2/S3, *Eucalyptus*). En effet, en tant que solution propriétaire, *GAE* est adhérent à l'infrastructure de *Google*.

Lancé par Google en avril 2008, *GAE* est une plate-forme permettant à des développeurs de réaliser des applications web puis de les mettre à disposition des utilisateurs sur Internet. Bien que les langages proposés pour le développement de ces applications soient de haut-niveau (i.e. Python ou Java), l'environnement *GAE* présente un certain nombre de restrictions techniques visant à assurer un niveau convenable de performances, de stabilité du système et de gestion des aspects large échelle. Ainsi, parmi ces restrictions, les entités applicatives ne peuvent communiquer qu'au travers des protocoles http et https. En outre, une application doit répondre à toute requête utilisateur en moins de 30 secondes. Enfin, l'application ne peut instancier de threads Java, ne dispose que d'un accès très limité au système de fichier et ne peut stocker des données en mémoire, que sous forme de couple clef-valeur. Dans ces conditions, *GAE* offre de bonnes facultés d'administration de l'application, qu'il s'agisse de son déploiement ainsi que des aspects relatifs à l'élasticité (i.e. dimensionnement dynamique de l'application en fonction de la charge à laquelle elle est confrontée) ou à la fiabilité (i.e. tolérance aux pannes).

Le formalisme utilisé par un développeur pour décrire une application dans *GAE* est le suivant :

- dans un contexte Python, une application *GAE* correspond à la fourniture d'une page d'accueil et d'un certain nombre de scripts Python implémentant la logique métier et ses mécanismes d'accès au données.

- dans un contexte Java, une application est empaquetée dans une archive répondant au format *ear* défini dans les spécifications *JEE*.

Comme il a été dit précédemment, *AppScale* est un environnement déployé sur une infrastructure composée d'un ensemble de machines virtuelles (i.e. cluster d'hyperviseurs ou plate-forme d'*IaaS*). Son architecture s'organise autour de 5 types d'entités ou *composants* :

***Data Base Slave* (*DBS*) :** il existe un nombre arbitraire d'instances de ce composant par application hébergée dans *AppScale*. La finalité de ce composant est de réaliser l'interface avec un système de persistance de données.

***Data Base Master* (*DBM*) :** représenté sous forme d'une seule instance au sein de chaque application hébergée par *AppScale*, un *DBM* joue le rôle de répartiteur de requêtes à l'adresse des différents *DBS*. Plus exactement, il met en œuvre des mécanismes de réplication. *DBM* et *DBS* permettent d'offrir un service de stockage réparti, fiable et gérant les aspects relatifs au passage à l'échelle.

***App Server* (*AS*) :** plusieurs instances de ce composant peuvent être associées à une application donnée. Chaque *AS* exécute le code métier de l'application et, pour cela, communique éventuellement avec le *DBM* pour le stockage et la consultation des données au travers d'un lien http ou https.

***App Load Balancer* (*ALB*) :** il existe une instance de ce composant par application hébergée dans *AppScale*. Son rôle est triple. D'une part, il assure la fonction de répartition de charge entre les *AS*s sur lesquels le code métier de l'application est déployé. D'autre part, il constitue un point d'accès unique pour un administrateur désireux de déployer et de superviser une application. Enfin, il initialise l'accès des utilisateurs à l'application car il constitue un portail d'authentification pour un utilisateur souhaitant utiliser l'application. Une fois authentifié, la requête utilisateur est transmise à un *AS* qui prend désormais à son compte la suite des interactions avec l'utilisateur, sans nouvelle sollicitation de l'*ALB*.

***App Controller* (*AC*) :** il existe une instance d'*AC* par nœud d'instanciation (i.e. par machine virtuelle). Ce composant est en réalité un composant d'administration de l'environnement *AppScale*. Plus précisément les *AC*s assurent l'interconnexion entre les composants applicatifs (i.e. *DBS*, *DBM*, *AS* et *ALB*) et réalisent le déploiement, la supervision et le retrait de l'application. Un *AC* est un daemon (ou agent) s'exécutant sur un nœud. Il est démarré automatiquement, au terme de la phase d'amorçage (*boot*) de la machine virtuelle.

La mise en œuvre du déploiement d'une application au travers d'*AppScale* s'organise de la manière suivante. Elle commence par l'instanciation d'une machine virtuelle appelée *initiateur*, qui active un *AC*. Celui-ci procède alors à l'instanciation des éventuels autres nœuds, qui donne lieu à l'activation automatique des *AC*s associés. Par la suite, l'*AC* de l'initiateur initialise l'*ALB* et, parallèlement, il contacte les *AC*s des autres nœuds de l'application afin qu'ils procèdent à la mise en œuvre des autres composants (i.e. *DBS*, *DBM* et *AS*). Dans un premier temps, l'*AC* de l'initiateur demande la création du *DBM* ce qui provoque alors la création des *DBS*s. Ensuite, il ordonne la création des *AS*s qui seront configurés avec l'adresse IP du *DBM*.

L'*AC* joue également un rôle important en matière d'administration. En effet, il supervise les autres nœuds de l'application pour détecter d'éventuelles pannes (e.g. mécanisme de heart beat) ou pour collecter des informations relatives à leur consommation de ressources matérielles (e.g. cpu, mémoire, ...) en vue de proposer un redimensionnement de l'application.

Automatisation

Comme la plupart des solutions orientées services, *AppScale* est un environnement capable d'automatiser l'ensemble des phases du déploiement de l'application et même au-delà, durant la phase de supervision.

Généricité

En revanche, les applications de ce type se focalisent sur un éventail de technologies, de

domaines métiers et/ou d'architectures applicatives très réduit. Dans le cas d'*AppScale*, il ne s'agit que d'applications web développés en Python ou Java/JEE selon un modèle de programmation très restreint.

Concernant l'indépendance vis-à-vis de l'environnement d'exécution applicative, elle est également plus variable. Autant *GAE* est adhérent à l'infrastructure privée de Google, autant *AppScale* est relativement agnostique vis-à-vis de l'infrastructure virtualisée sous-jacente.

Enfin, il n'existe pas réellement de modèle d'abstraction particulière. Cependant l'adhérence applicative demeure faible, car, en réponse à un champ d'application limité, l'environnement *AppScale* est capable de mettre en œuvre des fonctions d'administration génériques.

Passage à l'échelle

Les mécanismes d'élasticité proposés par *AppScale* permettent d'adapter la taille de l'application (i.e. par ajout / suppression dynamique d'*AS*s ou de *DBS*s) en fonction de la charge qui lui est appliquée. Ceci signifie que la solution est en mesure de gérer des applications de grande taille. En effet, l'architecture qu'elle propose est faiblement centralisée en ce sens qu'il existe une instance d'*ALB* par application (i.e. le composant d'amorçage) et que celui-ci n'est sollicité que par la première requête lors de l'arrivée d'un nouvel utilisateur. En outre, les expérimentations proposées par [48] laissent penser qu'*AppScale* est capable de réaliser efficacement des déploiements multiples.

Fiabilité

Bien qu'il ne fiabilise correctement qu'une partie de ses composants (i.e. *DBS* et *AS*), *AppScale* répond partiellement à la problématique des pannes affectant l'environnement d'exécution de l'application. Le principe du *heart beat* le prémunit également des défaillances réseaux vers ces deux types de composants ainsi que des défaillances des *AC*s, à l'exception de l'*AC* situé sur l'initiateur. Toutes ces remarques ne sont cependant vraies qu'à condition d'adapter judicieusement la répartition des composants sur les machines virtuelles.

Granularité

AppScale ne propose qu'une granularité liée à la structure d'une application, c'est-à-dire du niveau d'un composant au sens *AppScale* du terme (i.e. *ALB*, *DBM*, *DBS*, *AS*).

Synthèse

AppScale est un exemple extrêmement représentatif de ce que sont les solutions orientées services. Cet environnement propose un ensemble de fonctionnalités très matures, lui permettant d'automatiser la quasi totalité du cycle de vie d'une application. Néanmoins, en contre partie, il ne présente qu'une polyvalence limitée aux applications web en Python ou Java, conçues selon un modèle de programmation contraignant.

2.3.3 Solutions orientées application

Cette troisième catégorie de solutions vise à conjuguer l'approche orientée service, dans laquelle une application répartie est définie comme un assemblage de services de haut niveau (i.e. réponse à la question "quoi ?"), et l'approche orientée infrastructure, qui explicite la façon dont une application se décompose au sein d'un ensemble de ressources virtuelles (i.e. réponse à la question "comment ?"). Les solutions orientées application proposent ainsi un fort degré de paramétrisation pour que l'utilisateur définisse l'application à mettre en œuvre et le contexte d'exécution associé ainsi que l'articulation entre eux. Pour cela, elles exposent des modèles et des formalismes de description très riches. Bien qu'un petit nombre de solutions orientées application soient déjà disponibles sur le marché [128] [12], cette approche fait encore l'objet de nombreux travaux de recherche.

Cette section présente donc successivement les solutions suivantes :

- un système d'approvisionnement de nuages proposé par IBM et qui, à défaut d'avoir reçu un nom, sera appelé *IBMDeployer* dans la suite de ce document ;
- *MetaConfig* ;
- une architecture d'approvisionnement automatisé de services dans le nuage proposée par Hewlett Packard et qui, n'ayant fait l'objet d'aucune appellation, sera nommée *HPDeployer* dans la suite de ce document ;
- *Engage*.

Toutes quatre constituent des solutions innovantes, basées sur l'approche orientée application.

2.3.3.1 Système d'approvisionnement de nuages (IBMDeployer)

Pour répondre aux exigences de réduction de coûts relatifs à l'administration des applications, IBM a défini en 2010 l'architecture d'un système capable d'automatiser l'ensemble des étapes du déploiement de services complexes, répartis sur un ensemble de machines virtuelles dans le nuage [47].

Cette architecture s'organise autour d'un ensemble de gestionnaires interagissant, directement ou indirectement, avec une infrastructure de type informatique dans le nuage. Cette infrastructure correspond à un ensemble de ressources physiques (i.e. machines, réseaux), disposant toutes d'un mécanisme de virtualisation (e.g. hyperviseurs, virtualisation réseau, etc.).

Chaque gestionnaire est responsable d'une partie des opérations d'administration comme, par exemple, le stockage et la publication des images (*Image Manager*) ou l'approvisionnement des machines virtuelles (*Resource Manager*).

L'orchestration du processus de déploiement est assurée par l'*Instance Manager* en charge également d'instancier et de configurer les machines virtuelles sur lesquelles se répartissent l'application à déployer. Ce processus comprend les étapes suivantes :

1. A partir d'une modélisation de l'application, l'*Instance Manager* s'adresse au *Resource Manager* pour obtenir les ressources matérielles nécessaires. Les ressources administrées par le *Resource Manager* sont regroupées en *pool*. Chacun d'eux contient des ressources de type identique (e.g. machine virtuelle, adresse IP) et partageant potentiellement des caractéristiques communes (e.g. taille mémoire, nombre de cpu, etc.).

2. Il effectue alors la copie des images à installer, depuis la bibliothèque d'images (*Image Library*) gérée par l'*Image Manager*, vers les machines physiques mises à disposition précédemment.

3. il procède alors à l'instanciation des images au sein de machines virtuelles et génère les éléments de configuration spécifiques à chacune d'entre elles.

4. au terme du *boot* de chaque machine virtuelle, il effectue les opérations de post-configuration et d'activation nécessaires. Plus précisément, cette étape correspond à la mise en œuvre, sur la machine virtuelle, de processus d'orchestration, à l'aide de l'outil *Tivoli Provisioning Manager* d'IBM. Concrètement il s'agit de l'exécution à distance de scripts.

5. une fois l'ensemble des machines virtuelles correctement activées, il notifie l'administrateur de la fin d'exécution du processus de déploiement.

Comme l'illustre le listing 2.5, la proposition d'IBM comprend également un formalisme, au format XML, qui permet de spécifier l'organisation de l'application selon les machines virtuelles qui la composent (*node*). Un *node* est l'agrégation d'un ensemble d'attributs relatifs aux caractéristiques matérielles de la machine virtuelle associée (i.e. section *image-requirements*), à l'image virtuelle elle-même (e.g. tag *filename*) ainsi qu'aux entités applicatives elles-mêmes (i.e. section *software attributes*). Le formalisme proposé permet également de spécifier une liste de scripts à exécuter lors de la post-configuration de la machine virtuelle (i.e. tag *runoncescript*) ainsi que de faire référence à des variables d'environnement dont la valeur n'est connue qu'au moment du déploiement et sera renseigné par la plate-forme de déploiement (e.g. paramètre *$HOST0*).

```
<appliance>
  <name>SampleApp</name>
  <description>Example of an application</description>
  <node>
    <name>ServerNode</name>
    <description>The application server side</description>
    <image-requirement>
      <req_memory>1 GB</req_memory>
      <req-cpu>1</req-cpu>
      <req-disksize>10GB</req-disksize>
      <req-network>100Mb</req-network>
    </image-requirement>
    <image-specification>
```

```
      <ostype>RHEL 5u4</ostype>
      <imageid>emi-123456</imageid>
      <filename>serverimg.tgz</filename>
      <virtualization-type>KVM</virtualization-type>
      <runoncescript>"/root/fixup.sh $HOSTNAME
          $DOMAIN;"</runoncescript>
      <userid> root </userid>
      <password> password </password>
      <software-attributes>
        <software-attribute name="serverport" value="25573"/>
        <software-attribute name="server_home"
            value="/opt/server/"/>
      </software-attributes>
    </image-specification>
  </node>
  <node>
    <name>ClientNode</name>
    <description>The application client side</description>
    <image-requirement> ... </image-requirement>
    <image-specification>
      ...
      <runoncescript>"/root/fixup.sh $HOSTNAME $DOMAIN;
          /root/setServer.sh $HOST0 $DOMAIN0"</runoncescript>
    </image-specification>
  </node>
</appliance>
```

Listing 2.5 – Exemple de description d'application selon le formalisme proposé par [47]

Automatisation

IBMDeployer offre un niveau d'automatisation satisfaisant, couvrant l'ensemble des phases du déploiement comprises entre la publication et l'activation. Néanmoins, la phase de packaging n'est pas adressée par la solution.

Généricité

Le manque de granularité du modèle proposé par *IBMDeployer* limite sa polyvalence. En effet, faute de pouvoir affiner l'architecture de l'application en vue de briser d'éventuels cycles de dépendances (cf. paragraphe traitant de la granularité), cette solution n'est pas en mesure de déployer des applications complexes.

Lors d'une demande de déploiement, l'utilisateur fournit l'ensemble des images (i.e. la totalité des logiciels et des données) qui constituent l'application. Ces images sont alors instanciées au sein de machines virtuelles. Or, pour pouvoir alors réaliser les opérations de post-configuration (i.e. lancement de scripts à distance), *IBMDeployer* doit avoir accès à ces machines (i.e. c'est le rôle du système *Tivoli Provisioning Manager*). Ceci passe par une modification préalable des machines virtuelles de l'utilisateur. Ceci signifie qu'*IBMDeployer* dépend du type d'image manipulé, donc de la couche de virtualisation utilisé au niveau de l'infrastructure de *cloud*. Actuellement, *IBMDeployer* s'interface avec les hyperviseurs (i.e. couche de virtualisation) Xen, KVM et VMWare ESX.

Enfin, le formalisme de description proposé par *IBMDeployer* est basique. Il ne permet pas de définir des applications complexes comprenant des interdépendances croisées entre entités applicatives. En outre, il recourt à des scripts dédiés pour les opérations de post-configuration et d'activation. Ceci lui confère une forte adhérence applicative.

Passage à l'échelle

L'architecture d'*IBMDeployer* se révèle être centralisée, en ce sens que les opérations d'orchestration de déploiement sont toutes orchestrées au niveau de l'*Instance Manager*. De plus, cette solution ne propose pas de mécanisme de déploiement multiples.

Fiabilité

Aucune fonction de fiabilisation de la solution n'est proposée par *IBMDeployer*.

Granularité

A l'inverse de solutions purement orientées applications qui définissent l'application et les entités logicielles qui la composent avant d'en exprimer la projection sur une infrastructure d'exécution, *IBMDeployer* est un environnement qui vise à étendre les capacités des plates-formes d'*IaaS* d'IBM, en décrivant l'infrastructure au travers d'un ensemble de machines virtuelles puis en spécifiant la manière de les peupler. La principale conséquence de cette approche infrastructure-centrique est que le modèle exposé par *IBMDeployer* n'offre qu'une granularité de niveau machine virtuelle et image associée. Une telle granularité rend impossible la définition d'architectures applicatives complexes. En effet, ces dernières se caractérisent par la présence de multiples dépendances dont l'organisation peut parfois déboucher sur la formation de cycles. Un cycle est la traduction de dépendances croisées (ou mutuelles) entre plusieurs composants. Il est donc source potentielle d'interblocages au moment du déploiement de l'application. Il convient donc de briser ce type de structure avant toute instanciation de l'application. Pour y parvenir, il est nécessaire de définir plus précisément la manière dont les éléments logiciels qui composent l'application, dépendent les uns des autres, c'est-à-dire d'affiner la définition de l'architecture applicative à l'aide de composants présentant une granularité de plus en plus fine, en fonction de la complexité des interblocages à résoudre. Le modèle d'*IBMDeployer* n'offre pas une telle granularité arbitraire.

Synthèse

IBMDeployer constitue une première étape en matière de solution de déploiement d'applications dans le nuage à base de modèle. Cependant, la faible abstraction du formalisme associée à un couplage important vis-à-vis de l'environnement d'exécution applicatif (i.e. couche de virtualisation) réduisent les situations dans lesquelles la solution peut être utilisée.

2.3.3.2 MetaConfig

MetaConfig est une solution, publiée en 2011, résultat d'une collaboration entre Cirego ApS et l'Université de Copenhague. Elle propose d'automatiser la gestion de la configuration d'une application déployée sur des machines virtuelles [96]. Plus précisément, *MetaConfig* s'organise autour de trois sous-systèmes fonctionnels. D'un point de vue architectural, chaque sous-système se compose d'un ensemble d'agents instanciés au sein d'une entité serveur :

- le sous-système de ressource assure l'instanciation et la mise en œuvre d'une politique de placement des machines virtuelles applicatives vierges (i.e. vides) sur un ensemble (ou ferme) de machines physiques. Il se compose d'un élément centralisé (*Resource Server* ou *RS*) et d'un ensemble d'agents (*Resource Client* ou *RC*) répartis sur chaque machine physique.

- le sous-système d'installation effectue l'installation, sur chaque machine virtuelle applicative, d'une pile logicielle complète. Il se compose d'un élément centralisé (*Installation Server* ou *IS*) et d'un ensemble d'agents (*Installation Client* ou *IC*) répartis sur chaque machine physique.

- le sous-système de configuration procède à la configuration d'une machine virtuelle à partir d'une configuration de référence puis s'assure du maintien de la cohérence, au fil du temps, entre la configuration courante de la machine virtuelle et sa configuration de référence, stockée dans un référentiel (e.g. une base de données). Il se compose d'un élément centralisé (*Configuration Server* ou *CS*) et d'un ensemble d'agents (*Configuration Client* ou *CC*) répartis sur chaque machine physique ou virtuelle.

L'ensemble des entités serveurs de ces sous-systèmes (i.e. *CS*, *RS* et *IS*) sont regroupés au sein d'un *Control Server* centralisé.

MetaConfig propose également un formalisme déclaratif non ambigu, s'appuyant sur un *DSL* à l'aide duquel l'administrateur définit la configuration d'une machine physique ou virtuelle. Une telle configuration comprend :

- la description des caractéristiques matérielles (e.g. nombre de cpu, quantité de mémoire, architecture matérielle). Ces valeurs sont exploitées par le sous-système de ressource qui détermine le placement optimal des machines virtuelles sur les machines physiques, comme peut le proposer un système comme [77]. L'algorithme mis en œuvre procède selon un parcours itératif de l'espace des solutions au cours duquel les caractéristiques matérielles des machines virtuelles et physiques sont comparées.

- la définition de la pile logicielle en termes de système d'exploitation (i.e. type, distribution et version), de paquets, de fichiers et de droits d'accès, d'utilisateurs et de groupes d'utilisateurs (cf. listing 2.6). Ces informations sont exploitées par le sous-système d'installation, lors de la constitution initiale de la pile logicielle. Ainsi,

lors de son instanciation, une machine virtuelle est vierge. *MetaConfig* procède à l'installation de sa pile logicielle à l'aide d'un mécanisme d'installation réseau de type PXE [9]. Par la suite, c'est au tour du sous-système de configuration de configurer la machine en fonction des informations contenues dans le modèle puis d'assurer les éventuelles mises en cohérence de la configuration courante et la configuration de référence de la machine virtuelle, comme peut le proposer un système comme *Puppet* (cf. section 2.2.1.1).

```
# Importation d'une configuraiton par défaut
import "server/default"

# Installation du paquet apache2
[apt]
  install += "apache2"

# Définition du groupe et de l'utilisateur pour l'administation du
    serveur Apache
[group.vamp]
  gid = 1000

[user.apacheadmin]
  uid = 1000
  gid = 1000
  gecos = "Apache server admin account"
  home = "/home/apacheadmin"
  shell = "/bin/sh"

# Défintion du contenu du fichier /etc/hostname
[files.def.config-hostname]
  hostname = "httphost"

# Définition des dtrois relatifs au fichier
    /etc/apache2/apache2.conf
[files]
  permission["/etc/apache2/apache2.conf"] = "0644"


# En cas de modification de /etc/apache2/apache2.conf, exécute la
    commande mycmd sur le fichier myconf.cfg
[trigger]
  files["/etc/apache2/apache2.conf"] = ["mycmd", "myconf.cfg"]
```

Listing 2.6 – Extrait d'un descripteur de configuration d'une machine virtuelle selon le *DSL* proposé par *MetaConfig*

Automatisation

MetaConfig est un système capable d'automatiser intégralement les premières phases du processus de déploiement, à savoir, la mise à disposition et l'installation (approvisionnement inclus) des éléments logiciels applicatifs. Néanmoins, *MetaConfig* n'expose

aucune notion de dépendance au travers de son modèle, qu'il s'agisse de dépendances de configuration ou d'activation. Ainsi, il n'est pas en mesure de gérer les propriétés de configuration dynamique (i.e. celles dont la valeur n'est connue qu'à l'exécution) ou la mise en œuvre d'un ordre de démarrage.

Généricité

Le modèle exposé par *MetaConfig* permet de décrire précisément un ensemble de machines virtuelles ainsi que les piles logicielles qui leur sont associées. Cela permet de procéder au déploiement de n'importe quelle application patrimoniale.

Néanmoins, le modèle ne développant pas le concept de dépendance, il est impératif que l'ensemble des propriétés de configuration de l'application soient connues à l'avance. Ceci impose, par exemple, que l'environnement d'exécution applicatif dispose d'un plan d'adressage statique.

En outre, pour cette même raison, le seul moyen de gérer les dépendances est d'en encapsuler, de manière implicite, l'administration au sein de scripts spécifiques à l'application considérée. Un tel mécanisme accroît singulièrement le couplage de *MetaConfig* vis à vis de l'application.

Passage à l'échelle

MetaConfig a fait l'objet de travaux quant à l'évaluation de sa capacité à gérer des applications large échelle. Il en est ressorti que la solution de déploiement présente des goulots d'étranglement, dont le principal se situe au niveau des référentiels de paquets, lors de l'installation simultanée d'un nombre important de machines virtuelles. En outre, aucun mécanisme particulier n'est proposé dans la gestion des déploiements multiples.

Fiabilité

A l'instar de *Puppet*, *MetaConfig* ne propose pas de mécanisme de détection de panne du *Control Server*. De plus, en présence d'une défaillance franche de l'hôte sur lequel sont instanciés des agents *MetaConfig* (i.e. *RC*, *IC*, *CC*), aucun mécanisme de remplacement n'est prévu. Cependant le serveur est en mesure de tolérer son absence dans l'attente d'un remplacement manuel.

Granularité

La granularité caractérise le degré de modularité proposé pour décomposer une application en unités d'exécution interconnectées. Ainsi, bien que le formalisme proposé par *MetaConfig* permette de manipuler des fichiers, des utilisateurs, des groupes d'utilisateurs, des instructions d'installation ou des exécuteurs de scripts, il n'en reste pas moins que l'absence de dépendances entre ces entités altère sensiblement la granularité globale de la solution. Ainsi, *MetaConfig* propose une granularité fixe de niveau machine virtuelle.

Synthèse

MetaConfig est une solution très orientée sur l'approvisionnement de machines vir-

tuelles. Ainsi ses principaux atouts sont sa capacité de mise en œuvre de politiques de placement des machines virtuelles sur un ensemble de machines physique et sa faculté à installer sur une machine l'ensemble d'une pile logicielle, système d'exploitation inclus. Néanmoins, le modèle défini par *MetaConfig* s'avère insuffisant (i.e. pas de concept de dépendances) pour déployer de manière automatisée et fiable des applications arbitraires réparties. De plus, son architecture n'offre pas de capacité particulière de gestion du passage à l'échelle.

2.3.3.3 Architecture d'approvisionnement automatisé de services dans le nuage (HPDeployer)

En 2011, Hewlett Packard propose une architecture dont l'objectif est de répondre aux enjeux associés à l'automatisation du déploiement et de l'administration de services instanciés dans le nuage [82]. Ainsi, [82] souligne que les approches existantes ne prennent pas en compte le caractère dynamique de la configuration des environnements virtualisés, tant au moment de leur configuration que de la gestion des variations de charge (i.e. élasticité).

Cette architecture identifie deux composants essentiels : d'une part, le *Service Orchestrator* responsable de la mise à disposition d'un nouveau service, d'autre part, la *Design Layer* qui propose à l'utilisateur des outils pour lui permettre de définir ses services (e.g. outils de définition, de publication et de contextualisation de modèles, interfaces programmatiques).

De manière comparable à la proposition d'*IBMDeployer* (cf. section 2.3.3.1), le *Service Orchestrator* se subdivise en deux niveaux : le premier fournit les abstractions relatives aux entités administrées (*abstraction layer*) et le second correspond à la manière d'organiser les traitements autour de ces abstractions (*orchestration layer*). L'abstraction layer regroupe un ensemble de fonctions d'administrations encapsulées dans des gestionnaires :

***Infrastructure manager* :** est en charge de l'approvisionnement des ressources matérielles virtualisées. Il procure une interface uniformisée d'accès aux plates-formes d'*IaaS* ;

***VM communication manager* :** assure la communication entre les machines virtuelles indépendamment du type (i.e. synchrone ou asynchrone) et du protocole requis par chacune d'elle (e.g. ssh, vnc, telnet, etc.).

***Package manager* :** est responsable de l'installation des logiciels au sein des machines virtuelles. Pour cela il peut s'appuyer sur des outils existants (e.g. aptitude, yum) ou implémenter ses propres mécanismes d'installation.

***Configuration manager* :** assure la configuration et la reconfiguration des entités logicielles réparties au sein des machines virtuelles.

***Application manager* :** gère l'état et le cycle de vie de l'application et prend en charge l'ordonnancement des opérations d'administration entre composants.

L'*orchestration layer* s'appuie sur les informations de monitoring collectées par un *Monitoring manager* auprès des machines virtuelles applicatives pour organiser les interactions entre les gestionnaires de l'*abstraction layer* en vue de la mise en œuvre des processus de déploiement et de reconfiguration de l'application.

Outre la définition d'une telle architecture, [82] définit un modèle regroupant l'ensemble des informations nécessaires aux opérations de déploiement et de reconfiguration de l'application (i.e. architecture à base de composants nommés, dépendances entre composants, localisation des composants sur les machines virtuelles). Ce modèle est formalisé au moyen de deux *DSL*s (*Domain Specific Languages*). Le *Component Description Language* (*CDL*) décrit les composants applicatifs au travers de leurs attributs de configuration, des logiciels qui leur sont associés et des actions à réaliser lors des différentes phases du processus de déploiement. La valeur d'un attribut de configuration peut être renseignée de façon statique (i.e. pré-configuration) ou dynamique (i.e. post-configuration) à l'aide d'une variable d'environnement, disponible dans l'environnement durant la phase de déploiement. Le *Template Design Language* (*TDL*), quant à lui, définit l'architecture de l'application en tant qu'assemblage de composants, préalablement définis à l'aide du *CDL*. Le *TDL* définit notamment les dépendances de configuration et d'activation, la localisation des composants sur les machines virtuelles ainsi que les règles relative aux politiques d'élasticité.

Automatisation

HPDeployer est une solution qui couvre la totalité du processus de déploiement. Il faut cependant souligner qu'en terme de construction d'image (i.e. packaging), *HPDeployer* propose une solution dynamique calquée sur les approches de type configuration de système, telles que *Puppet*. En effet, toute machine virtuelle applicative est préalablement instanciée et initialisée (i.e. phase de *boot*) avant que les logiciels et les données dont elle a besoin soient installés par synchronisation de sa configuration courante avec une configuration de référence. Les limites d'une telle approche sont notamment la difficulté à en fiabiliser le mécanisme et le coût de peuplement de la machine virtuelle à chaque nouvelle instanciation de l'image.

Généricité

En matière de polyvalence, *HPDeployer* ne présente pas de restriction particulière. En effet, le modèle fourni demeure suffisamment générique pour ne pas présenter de dépendance particulière vis-à-vis d'une architecture, d'une technologie ou d'un domaine métier donné.

De même, *HPDeployer* est implémenté sur une technologie Java, relativement portable sur de multiples systèmes d'exploitation. Elle s'adapte à des plates-formes d'*IaaS* à l'aide de connecteurs dédiés.

Enfin, en matière d'adhérence applicative, le modèle proposé par *HPDeployer* intègre l'essentiel des concepts identifiés pour l'automatisation du déploiement. Ainsi le *TDL* est comparable à un *ADL*. Concernant le *CDL* il formalise la composition en termes de logiciels et la configuration des composants qui forment l'application.

Passage à l'échelle

HPDeployer propose une implémentation centralisée du *Service Orchestrator*. Celle-ci automatise l'orchestration du déploiement et de la reconfiguration en se basant sur les données de monitoring remontées par des agents (i.e. Nagios ou Ganglia) instanciés sur chaque machine virtuelle applicative. Ainsi, toute la logique d'administration est centralisée, ce qui limite la capacité de la solution à gérer des applications de grande taille. En outre, la solution ne propose aucun mécanisme de prise en compte de déploiements en parallèle.

Fiabilité

Comme l'identifie [82] lui-même, *HPDeployer* ne propose pas de mécanisme particulier pour assurer la fiabilité du système d'administration lui-même ainsi que la gestion des pannes affectant l'environnement d'exécution applicatif.

Granularité

HPDeployer s'appuie sur la définition de composants dont la granularité n'est pas restreinte.

Synthèse

HPDeployer est probablement l'une des solutions les plus proches de *VAMP*. Elle présente en effet un nombre d'atouts très important, qu'il s'agisse de son degré d'automatisation, sa généricité ou sa granularité. Néanmoins, l'absence de mécanismes de fiabilité a tendance à limiter intrinsèquement le caractère autonome d'*HPDeployer*. En outre, son architecture centralisée entrave sa gestion du passage à l'échelle.

2.3.3.4 Engage

Ayant fait l'objet d'une première publication en 2012 [68], *Engage* est le fruit d'une collaboration entre la société genForma Corp. (Etats-Unis) et l'institut de recherche allemand *Max Planck Insitute for Software Systems* (*MPI-SWS*). *Engage* est un système en charge d'assurer l'installation, la configuration et l'administration d'applications réparties arbitraires sur des infrastructures physiques ou virtuelles ayant pour objectif de couvrir les limitations, identifiées par les auteurs de [68][6], des solutions actuellement disponibles :

1. Bien que les modèles à base de composants interconnectés entre eux et s'exécutant de manière concurrente, permettent de décrire la plupart des architectures applicatives réparties, le passage d'une telle modélisation à l'instance effective de l'application est une tâche complexe qui, bien souvent, s'effectue manuellement ou au moyen de scripts spécifiques à l'application.

2. Malgré le nombre important de solutions capables d'assurer une partie de l'automatisation du processus de déploiement, peu sont en mesure d'effectuer la gestion

[6]Nous ne partageons pas forcément l'ensemble des limitations identifiées dans [68]

de bout en bout de ce processus, particulièrement dans le contexte des applications réparties sur plusieurs machines virtuelles.

3. La plupart des solutions du marché ne gèrent pas de façon autonome les aspects dynamiques de la configuration, à savoir ceux relatifs à des attributs dont la valeur n'est connue qu'après la mise à disposition de l'environnement d'exécution applicatif.

4. Un grand nombre de solutions ne proposent pas la gestion d'un ordre d'activation entre services.

Pour cela, *Engage* se compose, d'un modèle à composant permettant de décrire l'architecture de l'application à déployer, d'un module de projection capable de générer une spécification d'installation exhaustive à partir d'une spécification partielle fournie par l'utilisateur ainsi que d'un moteur d'exécution de mise en œuvre d'une spécification exhaustive de configuration.

Modèle Le modèle à composants proposé par *Engage* définit des *ressources*. Une ressource est décrite selon un formalisme propre à *Engage* au moyen d'un *DSL*. Elle se subdivise en un type et un driver.

Le type décrit un ensemble de *ports* qui peuvent correspondre à des attributs de configuration (*config_port*) relatifs à la ressource ou des interfaces requises (*input_port*) ou fournies (*output_port*). La notion de *port* recouvre donc un ensemble d'attributs de configuration dont la valeur peut être soit statique ou correspondre à une référence vers un autre attribut (i.e. définition d'une liaison explicite au moyen du mot clef *map*). Le type définit également des dépendances de la ressource considérée vis-à-vis d'autres ressources. Les dépendances peuvent être de trois natures :

les dépendances internes (*inside*) : il s'agit de dépendances vis-à-vis d'un conteneur dans lequel la ressource s'exécute (e.g. dépendance entre une machine virtuelle Java et une machine physique comprenant un système d'exploitation). Seules les machines physiques et virtuelles n'ont pas de dépendance interne.

les dépendances d'environnement (*environment*) : désignent des ressources qui doivent être installées et éventuellement démarrées avant de pouvoir exécuter la ressource courante.

les dépendances de pair (*peer*) : désignent des ressources qui doivent être présentes, mais pas nécessairement colocalisées sur la même machine que la ressource considérée.

Le driver d'une ressource, décrit quant à lui une machine à états qui permet de préciser l'ensemble des opérations qui doivent être réalisées lors de la mise en œuvre du cycle de vie de la ressource (i.e. installation, démarrage, arrêt). Il s'agit donc d'un graphe dont les sommets sont les états de la ressource et les transitions sont des actions associées à des conditions. Les conditions sont des prédicats booléens correspondant à

la conjonction des dépendances de la ressource vis-à-vis d'autres ressources. Un langage formel sous-jacent permet de définir l'ensemble de ces opérations.

Il est possible de définir une relation d'héritage entre ressources, permettant ainsi de développer des composants indépendants et réutilisables.

Spécification de configuration Dans la terminologie d'*Engage*, une spécification d'installation (d'une application) est une description permettant au moteur de déploiement d'en automatiser l'installation, la configuration et le démarrage. Une spécification complète d'installation définit l'ensemble des ressources et des valeurs d'attributs de configuration nécessaires dans le cadre du déploiement de l'application. Une description partielle ne fournit qu'un sous-ensemble de ces ressources et/ou de ces attributs. Plutôt que d'imposer à l'utilisateur la fastidieuse tâche de définir une spécification exhaustive de l'application à déployer, *Engage* lui permet de n'en proposer qu'une version partielle. Un module de projection réalise alors la transformation de la spécification partielle d'installation en une spécification complète. En s'appuyant sur la modélisation de l'application en termes de ressources (éventuellement publiées au sein d'un référentiel) et la spécification partielle de son installation, ce module met en œuvre un algorithme de résolution de contraintes.

Moteur de déploiement Le principal rôle du moteur de déploiement d'*Engage* est d'automatiser l'installation, la configuration et le démarrage d'une application patrimoniale, au sein d'une plate-forme virtualisée de type *IaaS* (i.e. Amazon Web Service ou RightScale), à partir de sa spécification complète d'installation. Avant tout déploiement, *Engage* est capable d'effectuer un certains nombre de vérifications statiques quant à la cohérence de la spécification d'installation de l'application (e.g. absence de cycles entre les dépendances, toute interface cliente est liée à une interface serveur et une seule, ...)

D'un point de vue architectural, *Engage* ajoute à chaque machine virtuelle applicative un agent chargé de l'exécution locale d'actions pilotées par le moteur de déploiement et capable de remonter un certain nombre d'informations de monitoring.

Automatisation

Engage est une solution qui automatise l'ensemble du processus de déploiement et notamment les phases d'installation, de (post-)configuration et d'activation. Concernant le packaging, elle adopte une stratégie similaire à *Puppet* en s'appuyant sur les mécanismes mis à disposition par les systèmes d'exploitation eux-mêmes. Par contre, *Engage* ne procède pas à l'installation du système d'exploitation sur une machine applicative. Enfin, en terme de publication, il ne semble pas qu'*Engage* prenne l'initiative d'enregistrer lui-même de nouvelles ressources.

Généricité

Le modèle proposé par *Engage* demeure suffisamment général pour décrire tout type d'application répartie virtualisable, lui conférant ainsi une très bonne polyvalence.

Concernant son indépendance vis-à-vis de l'environnement d'exécution applicatif, *Engage* est intégré au sein des plates-formes Amazon Web Service et RightScale mais n'en est pas dépendant. Il peut également s'exécuter dans un contexte non virtualisé et il est développé en Python.

Le modèle de données exposé par *Engage* présente un fort niveau d'abstraction, comparable à celui d'un *ADL*. Ainsi, les *map* d'*Engage* reprennent la notion de liaisons entre interfaces, alors que les dépendances *inside*, *environment* et *peer* correspondent aux principes de composition et d'assemblage.

Passage à l'échelle

L'architecture centralisée d'*Engage* ne le prédispose pas à déployer simultanément un nombre important d'applications. En outre, bien que la notion de spécification partielle d'installation puisse aider l'utilisateur à définir des architectures applicatives de grande taille, il n'en demeure pas moins que le recours d'un algorithme à base de résolution de contraintes pour projeter une telle spécification partielle sur un environnement réel est une approche coûteuse.

Fiabilité

Mis à part les vérifications statiques à priori auxquelles est soumise toute spécification d'installation avant d'être déployée, *Engage* ne propose, actuellement, aucun mécanisme de tolérance aux fautes.

Granularité

Engage n'impose pas de granularité particulière quant aux ressources manipulées. Ainsi, une ressource peut se limiter à une variable d'environnement.

Synthèse

Engage est une solution de très haut niveau en matière de modélisation de l'architecture applicative et de la mise en œuvre du déploiement. Ainsi elle parvient à conjuguer facilité de modélisation et puissance d'expression, grâce à l'utilisation d'un DSL de spécifications partielles d'installation. Néanmoins, *Engage* n'adresse pas la gestion des aspects relatifs au passage à l'échelle ou à la tolérance aux fautes.

2.4 Conclusion

Après avoir défini le principe de déploiement d'application et proposé une caractérisation d'une solution fiable capable de déployer une application patrimoniale dans le nuage, ce chapitre a consisté à identifier les forces et les faiblesses d'un ensemble représentatif de solutions d'automatisation de tout ou partie du cycle de vie applicatif. Le tableau 2.1 présente une synthèse de cette étude. Il illustre notamment la diversité de ces solutions quant aux propriétés non-fonctionnelles qu'elles adressent. Ainsi, les principales conclusions qui découlent de cette comparaison sont les suivantes :

- Bien qu'une faible adhérence applicative n'implique pas nécessairement qu'une solution puisse déployer de façon fiable des applications de grande taille ou un grand nombre d'applications simultanément, un niveau d'abstraction insuffisant pour la modélisation d'une application pénalise toujours la gestion large échelle et l'autoréparation.

- A l'exception d'*Eucalyptus* et d'*HPDeployer*, il existe un antagonisme important entre automatisation et généricité. Ainsi, les solutions capables d'automatiser l'ensemble du processus de déploiement introduisent une restriction dans le spectre des applications qu'elles sont capables d'administrer.

- Seules deux solutions parviennent à gérer convenablement l'ensemble des aspects relatifs à la gestion large échelle (i.e. *Eucalyptus* et *AppScale*). Néanmoins, leur insuffisante granularité ne leur permet pas de déployer n'importe quelle application patrimoniale. La granularité, qui définit la finesse de modélisation de l'application à déployer, constitue un élément critique dans la définition d'une future solution.

- Aucune des solutions présentées ne proposent de mécanisme d'autoréparation capable de faire face à une panne franche affectant l'environment d'exécution de l'application ou du système de déploiement lui-même. En outre, seul un tiers d'entre elles implémente des mécanismes partiels de tolérance aux fautes.

[7] Comme dans la plupart des solutions en environnement non virtualisé, cette solution n'assure pas l'approvisionnement matériel de l'environnement d'exécution de l'application

Solution	Autonomie	Polyvalence	Indépendance environnement d'exécution	Adhérence applicative	Large échelle	Déploiements multiples	Fiabilité	Granularité
Puppet	✓	✓	- [7]	✗	✗	-	-	✗
DeployWare	-	✓	✓	✓	✓	✗	✗	✓
OSGi	✓	-	- [7]	✗	✗	-	-	✓
FraSCAti	-	✓	✓	✓	✗	✓	✗	✓
Rainbow	✗	✓	✓	✓	✗	✗	✗	✓
SmartFrog	-	✓	- [7]	✓	✗	✓	✗	✓
TUNe	✓	✓	- [7]	-	✗	✗	✗	✓
ProActive	✓	✓	✓	✓	-	✓	✗	✓
Eucalyptus	✓	✓	✓	✗	✓	✓	-	✗
AppScale	✓	✗	✓	✓	✓	✓	-	✗
IBMDeployer	✓	-	-	✗	✗	✗	✗	✗
MetaConfig	-	✓	-	✗	✗	✗	-	✗
HPDeployer	✓	✓	✓	✓	✗	✗	✗	✓
Engage	-	✓	✓	✓	✗	✗	-	✓

TABLE 2.1 – Synthèse comparative des solutions de déploiement d'applications

Deuxième partie

Contributions

Cette partie présente les contributions relatives à ces travaux de thèse. Elle s'organise de la façon suivante. Le chapitre 3 propose une vue d'ensemble de VAMP. Il présente, d'une part, le formalisme utilisé pour modéliser l'organisation architecturale des unités d'exécution qui composent une application et les machines virtuelles sur lesquelles ces unités sont instanciées, d'autre part, les entités de contrôle qui composent le système VAMP lui-même et la manière dont elles interagissent pour assurer le déploiement des applications.

Les chapitres suivants reprennent alors les aspects importants du déploiement automatisé proposé par VAMP :

- Le chapitre 4 décrit la manière dont VAMP interagit avec une forge en vue de créer les images de machine virtuelle comprenant l'ensemble de la pile logicielle (i.e. le système d'exploitation, les intergiciels, les binaires et les données applicatives) nécessaires à l'instanciation, la configuration et l'activation des éléments logiciels déployés sur la machine virtuelle associée.

- Le chapitre 5 présente le protocole asynchrone en charge d'assurer la post-configuration et l'activation des entités qui composent l'application. Ce protocole est réparti au sein de l'ensemble des agents VAMP s'exécutant sur les machines virtuelles de l'application.

- Le chapitre 6 détaille la façon dont le protocole fiabilise le déploiement des machines virtuelles ainsi que l'approche utilisée pour fiabiliser les entités de contrôle qui constituent le système VAMP lui-même.

Chapitre 3

Vue d'ensemble de VAMP

Sommaire

3.1 Contexte de l'informatique autonome 78
3.2 Rappel des objectifs 80
3.3 Principes de conception 80
3.3.1 Modèle d'application distribuée 80
3.3.2 Entités de contrôle 89
3.3.3 Modèle de communication 91
3.4 Architecture globale 93
3.4.1 Portail 93
3.4.2 Gestionnaire d'application 94
3.5 Conclusion 95

La finalité ce chapitre est de proposer une vue d'ensemble de la solution de déploiement VAMP. D'une part, il présente le modèle utilisé pour décrire une application à déployer. Celui-ci précise notamment les unités d'exécution applicatives qui la composent et les propriétés de configuration qui leur sont associées, leur organisation au sein de l'architecture applicative modélisée à l'aide de contraintes ou dépendances entre unités d'exécution (i.e. dépendances de configuration, de placement ou d'activation), ainsi que les caractéristiques matérielles et logicielles des machines virtuelles sur lesquelles ces unités vont être instanciées. D'autre part, ce chapitre précise les entités de contrôle qui composent le système VAMP lui-même et la manière dont elles interagissent selon les principes de l'informatique autonome. Plus précisément, le caractère autonome signifie que la solution doit être capable de :

- mettre en œuvre des différentes phases qui composent le processus de déploiement d'une application, en recourant le moins possible à une intervention humaine extérieure ;

- assurer la fiabilisation du processus de déploiement au moyen de boucles de contrôles.

Cette première contribution a fait l'objet de publications [64][63].

L'organisation de ce chapitre est la suivante. La section 3.1 introduit le contexte de l'informatique autonomique dans lequel s'inscrivent les mécanismes proposés par VAMP. La section 3.2 rappelle alors les objectifs de ces travaux et les propriétés mises en avant. Par la suite, la section 3.3 présente les principes de conception à l'origine de VAMP et précise les choix technologiques qui ont été faits dans le cadre de leur implémentation. La section 3.4 poursuit par la définition de l'architecture de la solution. Enfin, la section 3.5 conclut ce chapitre.

3.1 Contexte de l'informatique autonome

Selon la définition proposée par [79], un système autonome désigne un système d'information capable, sans requérir l'intervention d'un opérateur humain, d'assurer des tâches relatives à sa propre administration et de s'adapter dynamiquement à des changements internes ou externes dans le respect de politiques et d'objectifs métiers.

En 2001, [78] définit les principes de l'informatique autonome en s'inspirant du fonctionnement du système nerveux autonome humain. Ainsi, celui-ci est en charge d'assurer un certain nombre de fonctions physiologiques élémentaires (e.g. régulation du rythme cardiaque, de la pression sanguine, de l'émission éventuelle d'adrénaline, de la digestion en fonction du contexte) de sorte que l'individu puisse se concentrer sur des problèmes de plus haut-niveau (e.g. quelle décision prendre face à une menace imminente). De manière comparable, la finalité de l'informatique autonome est de prendre en charge un certain nombre de fonctions pour que l'administrateur humain puisse focaliser son attention sur l'objectif à atteindre et non sur la manière d'y parvenir. [78] et [81] organisent ces fonctions selon quatre propriétés principales :

auto-configuration : capacité, d'une part, à installer et (re-)configurer le système considéré de manière à l'insérer dans un environnement externe complexe composé d'autres systèmes d'information interopérants, d'autre part, à modifier la configuration courante en fonction des exigences de qualité de service.

autoréparation : capacité, d'une part, à détecter, diagnostiquer (ou qualifier) puis à compenser ou à réparer des pannes survenant dans le système, d'autre part, à anticiper et se prémunir de problèmes potentiels.

autoprotection : capacité, d'une part, à protéger le système d'attaques malveillantes et des trous de sécurité résultants d'une utilisation inappropriée ou d'erreurs en cascade, d'autre part, à anticiper et réduire les effets d'erreurs ou de défaillances.

auto-optimisation : capacité, d'une part, à optimiser les besoins en ressources du système dans l'optique de satisfaire le plus grand nombre d'utilisateurs, d'autre part, à anticiper le niveau de disponibilité ou de demande de ressources en fonction des variations de charge.

L'exécution d'une tâche d'administration dans un système autonome s'organise selon le principe de rétroaction. Ainsi il s'appuie sur les quatre étapes d'une boucle de contrôle représentées en figure 3.1 :

Monitoring : le système autonome collecte les informations en provenance des éléments qui le composent. Durant cette phase, les données peuvent faire l'objet de traitements techniques tels que de l'agrégation ou du filtrage.

Analyse : les données collectées sont alors corrélées et analysées en fonction de la politique d'administration définie par l'utilisateur du système.

Planification : selon les résultats de l'analyse, le système autonome décide si une ou plusieurs actions doivent être mises en œuvre, et le cas échéant, il élabore le plan d'actions correspondant.

Exécution : le système exécute le plan d'actions élaboré au cours de la phase de planification.

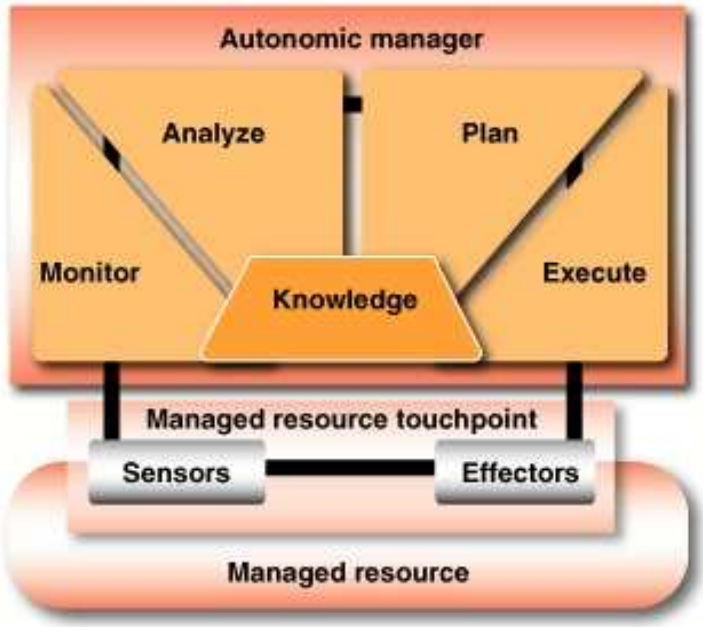

FIGURE 3.1 – Structure d'une boucle autonomique (extrait de [81])

Cette boucle de contrôle (ou boucle autonomique) est plus connue sous le nom de boucle MAPE-K pour Monitoring, Analyse, Planification, Exécution, le K faisant allusion à la connaissance (*knowledge* en anglais) nécessaire à la prise de décision. Le terme *boucle* souligne le fait que le système autonome déclenche régulièrement l'exécution des quatre étapes qui la constituent. Un système autonome peut comprendre plusieurs boucles autonomes indépendantes, en fonction de la tâche d'administration qu'elles adressent. En outre, il peut exister une hiérarchie entre boucles autonomes.

3.2 Rappel des objectifs

Le but de ce travail de thèse est de définir, d'implémenter et d'évaluer une plate-forme assurant, de manière autonome, générique et fiable, le déploiement d'applications dans le nuage.

L'aspect générique correspond, quant à lui, au fait que, la plate-forme doit non seulement permettre de déployer n'importe quelle application patrimoniale virtualisable, indépendamment du domaine métier, des choix technologiques ou architecturaux ou de l'environnement d'exécution (i.e. infrastructure) qui lui sont associés, mais également que les mécanismes mis en œuvre ne soient pas spécifiques à l'application déployée.

En outre, la propriété de fiabilité est la capacité de la solution, en présence d'un nombre fini de défaillances franches affectant l'environnement d'exécution applicatif ou le système de déploiement lui-même, à mener à bien, dans un laps de temps fini, le déploiement d'une application.

Enfin, cette plate-forme de déploiement ayant vocation à être expérimentée dans un contexte industriel, elle doit gérer les aspects relatifs au passage à l'échelle. Il s'agit, d'une part, de supporter l'exécution simultanée de plusieurs opérations de déploiement initiées par de multiples requêtes utilisateur (i.e. déploiements multiples) et, d'autre part, de permettre le déploiement d'applications de grande taille (i.e. déploiement large échelle).

3.3 Principes de conception

Les solutions actuelles dédiées à l'administration autonome des applications dans le nuage se caractérisent par une dualité importante entre leur niveau d'automatisation et la variété des applications gérées. Cette dualité est la conséquence du recours à des scripts de configuration spécifiques, en charge de réaliser les opérations de contrôle, au détriment d'une abstraction de plus haut niveau, basée sur un modèle de l'application administrée. Comme cela va être détaillé dans cette section, un tel modèle doit permettre de décrire la structure applicative afin que la solution d'administration puisse contrôler son évolution.

3.3.1 Modèle d'application distribuée

La finalité du modèle de l'application est d'assurer la capture :

- de l'architecture de l'application, en termes d'éléments logiciels applicatifs et des dépendances entre ces éléments (e.g. dépendances de configuration, d'activation, de placement) ;
- de la pile logicielle (système d'exploitation, intergiciels, binaires spécifiques et données applicatives) permettant de générer l'image de chaque machine virtuelle applicative ;
- des caractéristiques matérielles de chaque machine virtuelle utilisée dans la mise en œuvre des éléments logiciels applicatifs.

La figure 3.2 présente une vue globale des structures de données nécessaires à la modélisation d'une application. Celles-ci vont être précisées ci-après.

3.3.1.1 Modèle d'architecture applicative

L'approche utilisée pour décrire l'architecture de l'application s'appuie sur le concept de modèle à composants décrit en section 2.1.1.2.

Rappel du modèle à composants

L'approche à composants consiste à modéliser une application sous la forme d'un ensemble d'entités logicielles, appelées composants, et de dépendances entre ces entités. Un composant expose, au travers de points d'accès uniques appelés interfaces, un ensemble de services. Il s'agit de comportements, fonctionnels ou non, en lien avec l'ensemble des fonctionnalités applicatives qu'il encapsule (i.e. service fourni) ou dont il a besoin (i.e. service requis). Une liaison permet de réifier l'interconnexion (ou la dépendance) entre l'interface cliente (i.e. service requis) d'un composant et une interface serveur (i.e. service fourni) d'un autre composant. Si elle interconnecte deux composants qui s'exécutent dans le même espace d'adressage (e.g. une même machine virtuelle par exemple) elle est dite locale. Dans le cas contraire, il s'agit d'une liaison distante. La résolution de dépendance est l'opération qui permet à un composant qui expose une interface requise d'obtenir la référence de l'interface fournie associée.

La figure 3.3 illustre une manière possible de modéliser l'architecture de l'application fil rouge Springoo 1.4.3. Dans cet exemple, la modélisation proposée s'appuie sur trois composants (i.e. http-wrp, jee-wrp et db-wrp) et deux liaisons (i.e. *ajpinfo* et *dbinfo*). Le composant http-wrp encapsule le server HTTP et le répartiteur de charge. Il possède une interface cliente connectée à l'interface serveur du composant jee-wrp. Ces deux interfaces sont appelées *ajpinfo*. Le composant jee-wrp contrôle le serveur Java EE, l'instance de l'application et le connecteur JDBC. Il possède également une interface cliente connectée à l'interface serveur (toutes deux appelées *dbinfo*) du composant db-wrp. Ce dernier gère le système de gestion de base de données ainsi que l'instance de base de données.

La manière de modéliser l'application Springoo dans cet exemple, basée sur trois composants, est totalement arbitraire. Elle peut varier selon le niveau de précision désirée dans le contrôle de l'application. De cette précision découle le niveau de granularité de la modélisation. Ainsi, par la suite, la figure 3.4 propose une modélisation de Springoo à l'aide de cinq composants.

Modèle à composants Fractal

Concrètement, le modèle d'architecture de VAMP s'appuie sur le modèle à composants Fractal [36][29]. En effet, son caractère générique et extensible lui permet d'adresser indifféremment le domaine des environnements logiciels adaptables (comme les systèmes d'injection de charge Clif [56] ou de déploiement DeployWare [70]) que celui de l'administration d'applications patrimoniales (e.g. Jade [33]). Le modèle à composants Fractal s'avère donc particulièrement adapté aux besoins de VAMP.

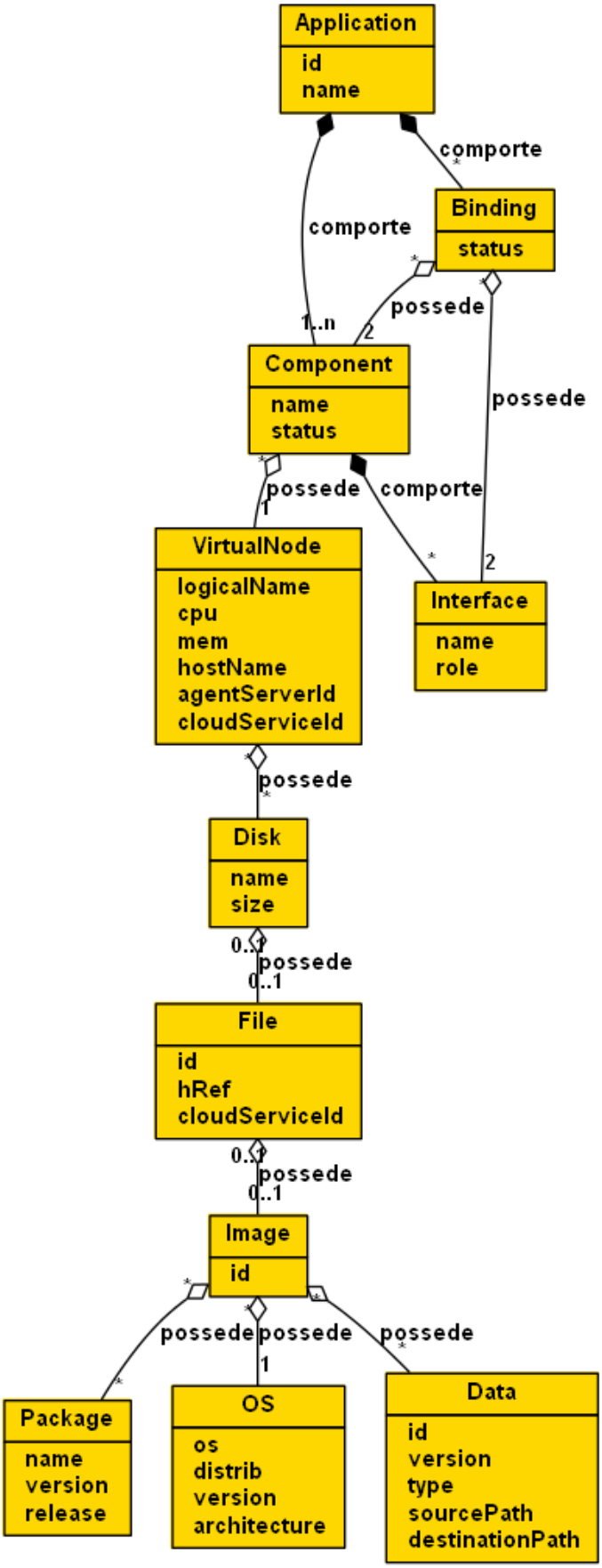

FIGURE 3.2 – Modèle de données de VAMP

Ainsi, dans le contexte de VAMP et en se basant sur les spécifications du modèle à composants Fractal [37], chaque élément logiciel applicatif est encapsulé au sein d'un composant. Celui-ci comprend :

- des attributs qui réifient des paramètres de configuration de l'élément logiciel encapsulé ;
- des interfaces typées qui constituent des points d'interconnexion potentiels. Ceux-ci correspondent au concept classique de service : une interface cliente est équivalente à un service requis alors qu'une interface serveur est assimilable à un service fourni. En outre, à une interface cliente est associée la notion de *contingence*. Elle permet d'indiquer si l'interface doit être connectée à une interface serveur avant que le composant puisse être démarré (i.e. *contingence obligatoire*) ou si au contraire, l'établissement de cette connexion n'est pas un prérequis au démarrage du composant (i.e. *contingence optionnelle*).
- des contrôleurs, qui correspondent aux interfaces non fonctionnelles du composant et qui sont utilisés dans le cadre de son administration. Ainsi, dans le contexte de VAMP, chaque composant expose les contrôleurs suivants[1] :

  NamingController : il associe au composant un espace de nommage dédié, dans lequel il permet de définir et de consulter son nom.

  AttributeController : permet de consulter et de modifier les valeurs des propriétés configurables de l'élément logiciel applicatif, c'est-à-dire celles exposées de façon externe.

  BindingController : il gère les interconnexions entre des éléments logiciels locaux ou distants. A l'aide d'un patron de type *export/bind*, il procure une

[1] De manière plus générale, un composant peut exposer tout ou partie de ces contrôleurs, voire en définir de nouveaux. Cela constitue pour partie le caractère générique et extensible du modèle Fractal

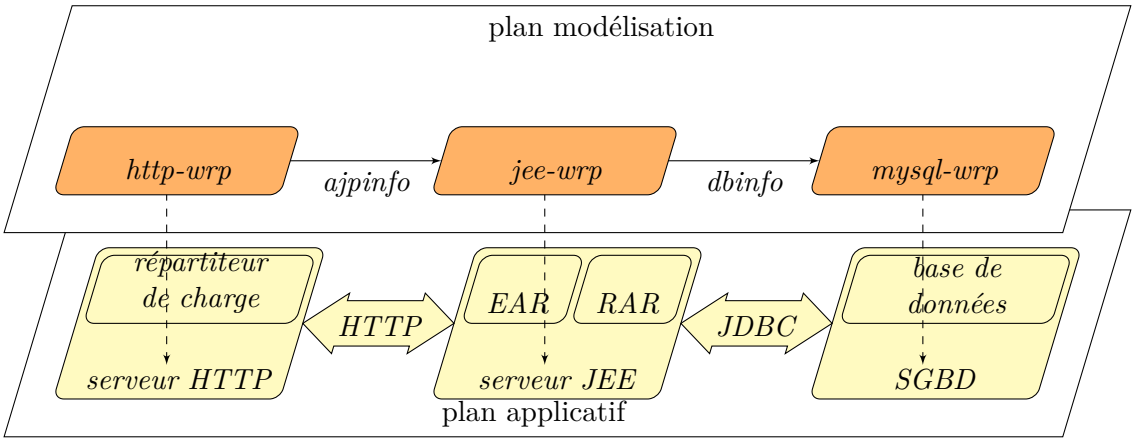

FIGURE 3.3 – Exemple de modélisation de l'architecture applicative de Springoo

vue uniforme des liaisons. Ce patron est une adaptation du *Reference Model of Open Distributed Processing* [67][53]. Il a été introduit pour définir une manière de configurer des liaisons en identifiant et en diffusant les données utilisées dans la configuration des canaux de communication patrimoniaux. Il s'organise selon deux opérations appelées *export* et *bind*. A partir du nom d'une interface serveur (qui réifie un point d'accès patrimonial tel que décrit dans le modèle), l'opération *export* renvoie un objet qui encode les informations nécessaires à un client pour qu'il puisse se connecter au point d'accès patrimonial. L'opération *bind* prend en entrée le nom d'une interface cliente à lier ainsi que l'objet exporté correspondant à l'interface serveur. Elle permet de décoder l'objet exporté qui réifie le point d'accès distant et de configurer l'élément logiciel applicatif en conséquence. Ce schéma général est déjà décliné sous différentes formes (e.g. web services, rmi, jms, etc.).

LifecycleController : il gère les transitions (e.g. démarrer, arrêter) entre les différents états du cycle de vie du composant ainsi que les opérations à appliquer sur l'élément logiciel applicatif encapsulé (sous-jacent) à chacune de ces étapes.

Outre sa capacité à décrire statiquement une application répartie en vue de son déploiement initial, le modèle à composants Fractal offre également la possibilité de découvrir l'état architectural de l'application au cours de son exécution. Cette capacité permet de gérer le déploiement comme une tâche incrémentale qui progresse en fonction de l'état dynamique de l'application répartie.

Un langage de description d'architecture (*Architecture Description Language* ou *ADL*) [89], en l'occurrence Fractal ADL [102], est utilisé pour décrire l'architecture applicative. Il adopte un format XML exploitable à la fois par les humains et les machines. Le listing 3.1 illustre l'utilisation de Fractal ADL pour décrire l'architecture applicative de Springoo. Dans cet exemple, Springoo est modélisé à l'aide de trois composants (cf. figure 3.3).

```
<component name="db-wrp">
 <interface name="dbinfo" role="server"
  signature="com.orange.vamp.MysqlConfiguration"/>
 <content class="com.orange.vamp.MysqlMBean"/>
 <attributes signature="com.orange.vamp.MysqlAttr">
  <attribute name="dbUser" value="springoo"/>
  <attribute name="dbPasswd" value="springoo"/>
  <attribute name="dbName" value="springoodb"/>
 </attributes>
 <virtual-machine name="VM3"/>
</component>
<component name="jee-wrp">
 <interface name="dbinfo" role="client"
  signature="com.orange.vamp.MysqlConfiguration"/>
 <interface name="ajpinfo" role="server"
  signature="com.orange.vamp.JonasLBConfiguration"/>
 <content class="com.orange.vamp.JonasMBean"/>
```

```
 <attributes signature="com.orange.vamp.JonasAttr">
  <attribute name="jonasName" value="jonas1"/>
  <attribute name="jonasDomain" value="domain1"/>
  <attribute name="ajpPort" value="9009"/>
  <attribute name="jvmRoute" value="jonas1"/>
 </attributes>
 <virtual-machine name="VM2"/>
</component>
<component name="http-wrp">
 <interface name="ajpinfo" role="client"
  signature="com.orange.vamp.JonasLBConfiguration"/>
 <content class="com.orange.vamp.LoadBalancerMBean"/>
 <virtual-machine name="VM1"/>
</component>
<binding client="jee-wrp.dbinfo"
         server="db-wrp.dbinfo" />
<binding client="http-wrp.ajpinfo"
         server="jee-wrp.ajpinfo" />
```

Listing 3.1 – Modèle d'architecture de Springoo à l'aide de Fractal ADL

Chaque composant (tag `component`) est défini à l'aide :

- d'un nom (attribut `name`) ;
- d'interfaces clientes et/ou serveur (tag `interface`). Chaque liaison entre une interface cliente et une interface serveur est décrite au moyen du tag `binding` ;
- d'attributs de configuration (tag `attribute`) ;
- d'une référence vers la machine virtuelle sur laquelle il devra être instancié (tag `virtual-machine`). Cette propriété permet de capturer les contraintes de placement entre composants (i.e. colocalisation ou non colocalisation). Sa valeur correspond à la valeur de la propriété `ovf:id` du tag `VirtualSystem` décrit dans la section OVF (cf. section 3.3.1.2). Un `VirtualSystem` correspond à la modélisation d'une machine virtuelle applicative ;
- d'une classe Java utilisée pour implémenter les opérations de contrôle sur l'élément logiciel encapsulé (`content` tag).

Afin d'illustrer l'utilisation des contrôleurs détaillés ci-dessus, considérons le composant http-wrp qui encapsule le serveur HTTP de l'application Springoo. Il expose des opérations de contrôle sur le fichier de configuration et les scripts d'activation du serveur Apache :

- Le NamingController est utilisé pour positionner le nom du composant à http-wrp.
- L'AttributeController permet de définir des attributs relatifs à l'exécution de l'instance locale de serveur Apache. Ainsi, une modification de l'attribut `port` du composant http-wrp est reportée dans le fichier de configuration correspondant (i.e. `httpd.conf` or `ports.conf`) dans lequel la propriété `port` est définie[2].

[2]Cette propriété définit le port TCP d'écoute du serveur Apache

- Le BindingController est utilisé pour connecter le serveur Apache à d'autres tiers. L'opération *bind* sur le composant `http-wrp` permet, par exemple, de configurer une liaison entre l'instance de serveur Apache considérée et le serveur JEE JOnAS. L'invocation de cette méthode *bind* se traduit, au niveau de la couche applicative, par la modification du contenu du fichier `worker.properties`, utilisé pour définir les connexions entre le serveur Apache et les serveurs JOnAS. De même, l'opération *export* sur le serveur JOnAS permet d'encoder l'adresse et le port de l'hôte à utiliser pour le contacter.

- Le LifecycleController est utilisé pour démarrer et pour arrêter le serveur ainsi que pour récupérer son état (i.e. *STARTED* ou *STOPPED*). Son implémentation consiste à invoquer les commandes Apache permettant de démarrer / d'arrêter le serveur.

3.3.1.2 Modèle de machine virtuelle

La modélisation d'une machine virtuelle se base sur celle proposée dans le cadre de la spécification *Open Virtualization Format* (OVF) [59]. Une machine virtuelle est donc vue comme l'agrégation d'un ensemble de processeurs, une quantité de disques, des interfaces réseaux et des disques. Chaque disque se caractérise par une taille et une image peut lui-être associée.

Concernant l'image, il n'existe pas de modélisation standard permettant de décrire son contenu. Celui-ci est donc représenté au moyen d'un système d'exploitation, d'un ensemble de packages et de données utilisateur qui peuvent aussi bien être des binaires que des données applicatives. En outre, des efforts de standardisation restent à faire concernant le format des images. Actuellement, chaque hyperviseur[3] propose son propre format (e.g. format vmdk pour VMWare [126], format qcow2 pour KVM).

Formalisme OVF

OVF est une spécification dédiée au packaging d'appliances virtuelles, en vue de leur déploiement initial. Elle fait l'objet d'une standardisation dans le cadre d'une initiative de la *Distributed Management Task Force* (DMTF) [57], dont les principaux acteurs sont VMWare et Citrix. Tous deux proposent d'ailleurs déjà des plates-formes permettant de déployer des packages OVF. La spécification OVF actuellement disponible est la version 1.0.

Selon la spécification OVF, une appliance virtuelle est un *package* qui regroupe l'ensemble des images virtuelles qui la composent[4], un descripteur au format XML (descripteur OVF), un fichier manifest et un éventuel fichier de signature. Un descripteur OVF permet, au moyen d'un langage ouvert et extensible, de décrire une application répartie en fonction des machines virtuelles qui la composent. Chacune d'elles est définie à l'aide du tag `VirtualSystem`. Elle comprend un ensemble de caractéristiques matérielles (i.e.

[3]Le terme hyperviseur désigne un environnement d'exécution capable de créer, activer, migrer, stopper et détruire des machines virtuelles

[4]Ces images sont contenues dans des fichiers fournis par l'utilisateur

nombre de processeurs, quantité de mémoire, nombre et taille des unités de stockage, interfaces réseau). A chacune de ses unités de stockage peut être associée une image virtuelle statique contenu dans le *package OVF*. Il est également possible de préciser un ordre de démarrage entre les machines virtuelles d'un même *package OVF*, de procéder au *reboot* d'une machine virtuelle lorsque l'étape de postconfiguration est terminée ainsi que de définir des variables d'environnement, communes à l'ensemble des machines virtuelles, et dont la valeur pourra n'être renseignée qu'à l'exécution.

OVF est actuellement le seul standard permettant de décrire l'organisation du déploiement d'un ensemble d'images virtuelles. Il constitue donc une brique essentielle dans la formalisation complète et cohérente d'une application répartie dans le nuage. Cependant, dans sa version standard, OVF présente deux limites importantes. D'une part, il ne permet pas de déployer, de façon autonome, des applications complexes (i.e. comportant des dépendances croisées entre éléments logiciels non colocalisés sur une même machine virtuelle). En effet, outre le fait qu'il ne propose pas de support de description de l'architecture applicative, sa granularité fixe, de niveau machine virtuelle, est insuffisante. D'autre part, OVF ne permet pas d'assurer la génération automatique des images qui constitue l'appliance à déployer. En effet, il n'en modélise pas le contenu.

Extensions d'OVF

Afin de répondre aux limitations d'OVF en termes de généricité et d'automatisation, la proposition faite dans cette thèse consiste à ajouter au formalisme actuel les deux extensions suivantes :

DynamicImage : cette extension sert à automatiser la génération des images virtuelles. Chaque occurrence de l'extension définit le contenu d'une image, en termes des entités logicielles qui la composent. La section 4.2.1 précise les éléments qu'il est possible de définir au moyen de cette extension. Dès lors, une image associée à une machine virtuelle peut être définie statiquement –grâce aux mécanismes OVF standards– ou dynamiquement –en utilisant la section `DynamicImage`–.

AppArchitectureSection : la seconde extension est une vue architecturale de l'application. Elle s'appuie sur une modélisation à base de composants au format Fractal ADL (cf. section 3.3.1.1). Lors de la création d'une image virtuelle, le système de génération d'image ajoute automatiquement un configurateur. Ce configurateur est démarré automatiquement, à la fin de la phase d'initialisation de la machine virtuelle sur laquelle il est instancié. A partir de la description de l'architecture applicative au format Fractal ADL, il procède alors à la configuration et au démarrage automatisés des composants colocalisés dans la machines virtuelle, selon le protocole décrit dans le chapitre 5.

Le listing 3.2 correspond à un extrait du descripteur OVF étendu de l'application Springoo. Il permet d'illustrer la manière d'établir le lien entre les différents éléments du modèle d'application répartie. Ainsi, le composant *http-wrp* de l'application Springoo a été décrit (cf. listing 3.1) comme devant être instancié sur la machine virtuelle dont

l'identifiant est défini au moyen du tag `virtual-node` et vaut *VM1*. Cet identifiant fait référence à l'attribut `pm:id` d'un `VirtualSystem` dans le descripteur OVF étendu. A cette entité est associé un disque (tag `ovf:Disk`) dont l'identifiant est *ovf :/disk/appDiskId2*. Le contenu de ce disque est identifié par l'attribut `ovf:fileRef` qui correspond à l'identifiant d'un fichier (tag `ovf:File`). Il peut s'agit d'un fichier statique contenant une image virtuelle fournie par l'utilisateur ou d'une image à générer dynamiquement. Le lien entre le fichier et la section `DynamicImage` est assuré par l'attribut `ovf:id` du fichier.

```
<Envelope ...>
 <Reference>
  ...
  <!-- content of the Apache appliance -->
  <DynamicImage ovf:id="appDisk2"
                pm:userDisk="appDisk2" ...>
  ...
  </DynamicImage>
  <ovf:File ... ovf:id="appDisk2" />
 </Reference>
 ...
 <!-- Applicative architecture -->
 <AppArchitectureSection name="com.orange.Springoo">
  <!-- the Fractal ADL description of the applicative
       architecture (see above) -->
 </AppArchitectureSection>
 ...
 <!-- Virtual machines configuration -->
 <ovf:VirtualSystemCollection ovf:id="springoo">
  ...
  <ovf:VirtualSystem ovf:id="VM1" ...>
   ...
   <ovf:VirtualHardwareSection>
    ...
    <ovf:Item>
     <rasd:ElementName>Harddisk 2</rasd:ElementName>
     <rasd:HostResource>
      ovf:/disk/appDiskId2
     </rasd:HostResource>
     ...
    </ovf:Item>
   </ovf:VirtualHardwareSection>
  </ovf:VirtualSystem>
  ...
 </ovf:VirtualSystemCollection>
 <ovf:DiskSection>
  ...
  <ovf:Disk ovf:fileRef="appDisk2"
   ovf:diskId="appDiskId2" ..."/>
  ...
 </ovf:DiskSection>
</Envelope>
```

Listing 3.2 – Extrait de la description OVF étendu de l'application Springoo

3.3.2 Entités de contrôle

A partir du modèle d'application présenté en section 3.3.1, le framework de déploiement autonome s'appuie sur un certain nombre d'entités de contrôle, dont l'action coordonnée permet d'assurer l'installation, la configuration et l'activation de l'application. La coopération entre ces entités implique qu'elles puissent communiquer entre elles. Afin de se conformer aux exigences de fiabilité et de passage à l'échelle (cf. section 3.2), VAMP s'appuie sur un support de communication décentralisé (i.e. communication en mode point à point), asynchrone (i.e. pas d'attente active dans l'envoi d'un message) et fiable (i.e. garantie de délivrance de chaque message envoyé).

3.3.2.1 Wrappers et configurateurs

Wrappers Dans le contexte de l'administration d'applications, le rôle des composants est de maintenir une vue sur la configuration applicative et d'exposer les opérations de contrôle qui peuvent être invoquées sur les éléments logiciels applicatifs qui composent l'application. Ils ne jouent aucun rôle fonctionnel vis-à-vis de l'application et sont donc qualifiés de *wrappers*. Dans l'implémentation de référence de VAMP développée en Java, les *wrappers* sont des *managed beans* (ou *MBeans*) qui peuvent être manipulés au travers de l'API JMX [4]. Un *wrapper* expose deux types d'interface :

interface fonctionnelle : il s'agit d'une simple réification d'un service exposé par l'élément logiciel. Ce service peut être associé à un autre service au travers d'une liaison qui traduit alors l'existence d'une contrainte (i.e. d'une dépendance) de configuration entre composants.

interface de contrôle : également appelée *contrôleur*, il s'agit d'un point d'accès à un service non fonctionnel exposé par le composant. Il permet de contrôler certains aspects de l'administration du composant, tels que la gestion de paramètres de configuration locaux (attributs) ou globaux (configuration des liaisons), le cycle de vie, etc.

L'adoption d'un modèle à composants dans ce contexte n'impose nullement que les applications modélisées se conforment elles-mêmes à ce type de modèle. De même, les interfaces et les liaisons ne servent qu'à décrire et à contrôler les interdépendances entre composants. Elles n'induisent aucune hypothèse quant au modèle de communication (e.g. synchrone, asynchrone, ...) ou protocole (e.g. rmi, http, ...) utilisés par les éléments logiciels réifiés au travers des composants, pour interopérer. Le système d'administration de l'application s'appuie donc sur un ensemble de *wrappers* pour contrôler les éléments logiciels applicatifs. Un élément logiciel et le *wrapper* associé sont ainsi colocalisés sur la même machine virtuelle.

Configurateur

Une machine virtuelle embarque également un *agent de configuration* (ou *configurateur*) qu'elle instancie automatiquement au terme de sa phase d'initialisation. A partir d'un extrait du modèle de l'architecture applicative réduit aux *wrappers* locaux (i.e. ceux colocalisés sur la même machine virtuelle que le configurateur considéré) et à leurs voisins (i.e. ceux qui se trouvent sur une machine virtuelle distante mais qui sont associés à un *wrapper* local au travers d'une liaison entre interfaces), le rôle d'un configurateur est :

1. de créer et de configurer localement les *wrappers* dont il a la responsabilité, c'est-à-dire ceux associés à des éléments logiciels applicatifs déployés dans la machine virtuelle. Lors de cette première étape, chacun des configurateurs présents sur chacune des machines virtuelles applicatives agit indépendamment des autres ;

2. d'interagir, dans un second temps, avec les configurateurs des autres machines virtuelles applicatives pour réaliser la configuration globale, c'est-à-dire l'établissement des liaisons entre *wrappers* distants, puis l'activation de l'application conformément à l'ordre de démarrage décrit par l'utilisateur.

Un configurateur est également capable de remonter des informations de monitoring. Celles-ci peuvent être relatives à l'évolution du déploiement des *wrappers* dont il a la charge mais elles peuvent également concerner son propre état de fonctionnement.

3.3.2.2 Exemple d'entités de contrôle - Springoo

La figure 3.4 illustre l'organisation de ces entités de contrôle dans le cadre de la modélisation de l'application fil rouge Springoo (cf. section 1.4.3). Comme le représente la figure, un configurateur est instancié au sein de chaque machine virtuelle applicative. Il a la charge d'un (i.e. VM1 et VM3) ou plusieurs (i.e. VM2) *wrappers* colocalisés sur la machine virtuelle.

Dans cet exemple, la modélisation proposée s'appuie sur cinq *wrappers* et cinq liaisons, proposant ainsi une granularité plus fine que l'exemple représenté sur la figure 3.3. Ainsi, le composant *jee-wrp* qui, dans la modélisation à base de trois composants, encapsule tout à la fois le serveur JEE, l'instance d'application (EAR) et le connecteur JDBC (RAR), est remplacé, dans la modélisation à base de cinq *wrappers*, par trois *composants*, chacun d'eux étant en charge de l'un des éléments logiciels précédemment énumérés. Grâce à cette granularité plus fine, il devient possible d'activer simultanément le serveur d'applications JEE et la base de données. Il est également possible d'activer le serveur HTTP sans attendre le démarrage de l'application elle-même. Enfin, cette seconde version capture les dépendances entre le serveur JEE, l'instance d'application et le connecteur JDBC, alors que dans la première modélisation, ces dépendances sont gérées de manière spécifiques par le *wrapper* jee-wrp.

3.3.3 Modèle de communication

Le déploiement d'une application au travers de VAMP met en œuvre plusieurs traitements répartis au sein d'un ensemble d'entités d'administration. Ce terme désigne des entités de contrôle génériques, c'est-à-dire non dédiés à un élément logiciel donné, comme peuvent l'être les *wrappers*. Les entités d'administration regroupent non seulement les configurateurs décrits en section 3.3.2, mais également un certain nombre de gestionnaires, présentés en section 3.4. L'orchestration des traitements passe donc par un certain nombre d'échanges entre ces entités. VAMP s'appuie sur un bus à messages [24] pour assurer ces communications. Celui-ci présente les caractéristiques suivantes :

Distribution : le bus est réparti au sein d'un ensemble d'entités fonctionnellement équivalentes, appelées serveurs d'agents. Chacun d'eux s'exécute dans un processus Java qui lui est propre. Sur chaque serveur d'agents sont déployés des objets Java appelés agents. Un agent est qualifié d'objet passif étant implémenté selon un modèle de programmation de type événement-action : il est capable de réagir à des événements provenant d'autres agents et d'émettre des événements vers d'autres agents. Dans un souci de fiabilisation du système de déploiement lui-même (cf. section 3.2), l'approche adoptée par VAMP consiste à isoler les entités d'administration (i.e. configurateur, gestionnaire d'application ou portail) les uns vis-à-vis des autres. Ainsi, chacune d'elles est un agent déployé au sein d'un serveur d'agents qui lui est propre. Chacun de ces serveurs d'agents s'exécute dans une machine virtuelle Java dédiée. Enfin chacune d'elle s'exécute dans sa propre machine virtuelle.

La structure ainsi définie est assimilable à un réseau d'*overlay*, c'est-à-dire à une structure qui vise à masquer la complexité d'un réseau de communication existant, par introduction d'une couche d'abstraction au-dessus de celui-ci. Un réseau d'*overlay* se compose d'un ensemble de *nœuds* interconnectés au travers de liens logiques (également appelés *canaux de communication*) du réseau sous-jacents. Ce type de structure est généralement découplé du mode de communication (e.g. syn-

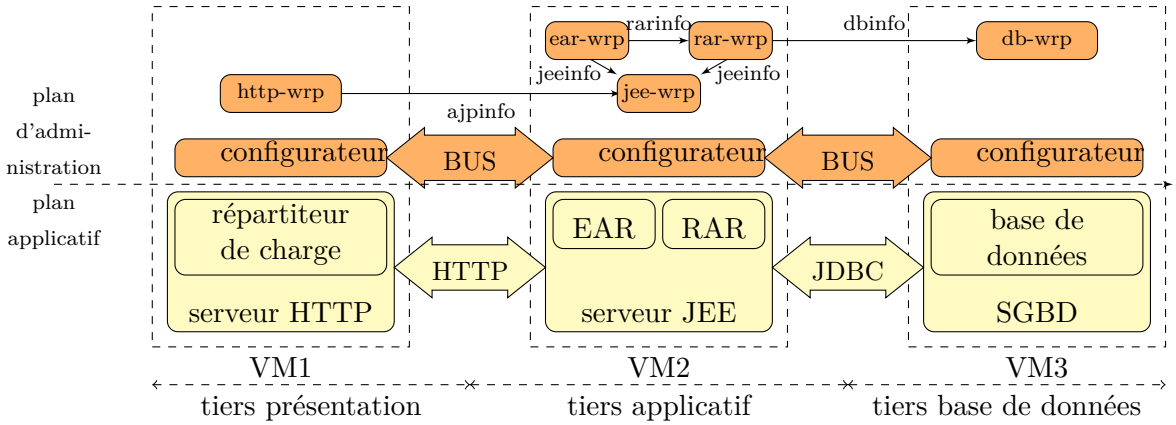

FIGURE 3.4 – Répartition des entités de contrôle de l'application Springoo

chrone vs. asynchrone, centralisé vs. décentralisé) utilisé pour assurer les échanges entre les nœuds. Dans le cas du bus à messages, les serveurs d'agents jouent le rôle de nœuds fonctionnellement identiques qui, après avoir rejoint le réseau, sont capables d'échanger directement les uns avec les autres selon un mode de communication décentralisé.

Asynchronisme : la communication entre deux agents au sein du bus s'effectue en mode déconnecté. Lorsqu'un agent émetteur A_e désire émettre un événement E à destination d'un agent A_d, il ne se connecte pas directement sur l'instance de A_d mais dépose l'événement E sur le bus en désignant le destinataire à l'aide d'un identifiant logique $Id(A_d)$. Le bus encode alors E sous forme d'un message avant de l'acheminer jusqu'à A_d. L'abstraction des agents au travers de leur identifiant permet à un agent d'émettre (respectivement de recevoir) un message à destination (respectivement en provenance) d'un agent dont l'identifiant est connu dans le bus mais dont l'instance est temporairement indisponible (e.g. non encore active, défaillante, en cours de maintenance). Le bus implémente donc un support de communication en mode déconnecté avec un couplage faible entre les agents.

Fiabilité : le bus garantit la délivrance des messages. Pour cela, chaque serveur d'agents SA dispose de deux queues persistantes pour stocker les messages : l'une, Q_out, est dédiée aux messages émis par les agents déployés sur SA, l'autre, Q_in, stocke les messages à destination de ces mêmes agents. Lorsqu'un agent A_e déployé sur un serveur d'agents SA_e émet un message, SA_e le stocke dans sa queue d'émission Q_out. Parallèlement il tente de l'envoyer au serveur d'agents SA_d sur lequel est déployé l'agent destinataire A_d. De son côté, lorsque SA_d reçoit un message de SA_e, il le stocke dans sa queue de réception Q_in. SA_d conserve le message dans Q_in jusqu'à ce qu'il ait réussi à le transmettre à A_d. Une fois que A_d a réagi au message, SA_d envoie un acquittement pour indiquer à SA_e que le message a bien été reçu par A_d. Lors de la réception de cet acquittement, SA_e retire le message stocké dans Q_out. Ce mécanisme garantit que le message émis par l'agent émetteur A_e est conservé de manière persistante dans le bus, d'abord dans la queue Q_out de SA_e puis également dans la queue Q_in de SA_d) jusqu'à ce qu'il soit effectivement remis à l'agent destinataire A_d. Outre la persistance des messages, le bus assure également la persistance de l'état des agents. L'état d'un agent est ainsi stocké après chaque réception de message.

Atomicité et cohérence : afin de garantir la cohérence de l'état des agents, le bus assure que la réaction d'un agent à un événement est atomique. Ainsi, soit l'ensemble des étapes composant la réaction sont réalisées et l'agent passe dans un nouvel état consistant, soit au contraire, une étape échoue, et le bus rétablit l'état de l'agent avant la réception (mécanisme de *rollback*).

Ordonnancement : deux messages $M1$ et $M2$ envoyés successivement par un agent émetteur A_e vers un agent destinataire A_d, seront reçus dans le même ordre (i.e. $M1$ puis $M2$) par A_d.

3.4 Architecture globale

Le déploiement d'une application dans le nuage résulte de l'exécution ordonnée d'un certain nombre de traitements (cf. section 2.1.1.3). Cette section va donc détailler la manière dont ils se répartissent au sein des entités d'administration, et plus précisément des gestionnaires, qui composent VAMP (cf. figure 3.5) ainsi que la manière dont ces gestionnaires sont répartis en termes de machines physiques ou virtuelles (cf. figure 3.6).

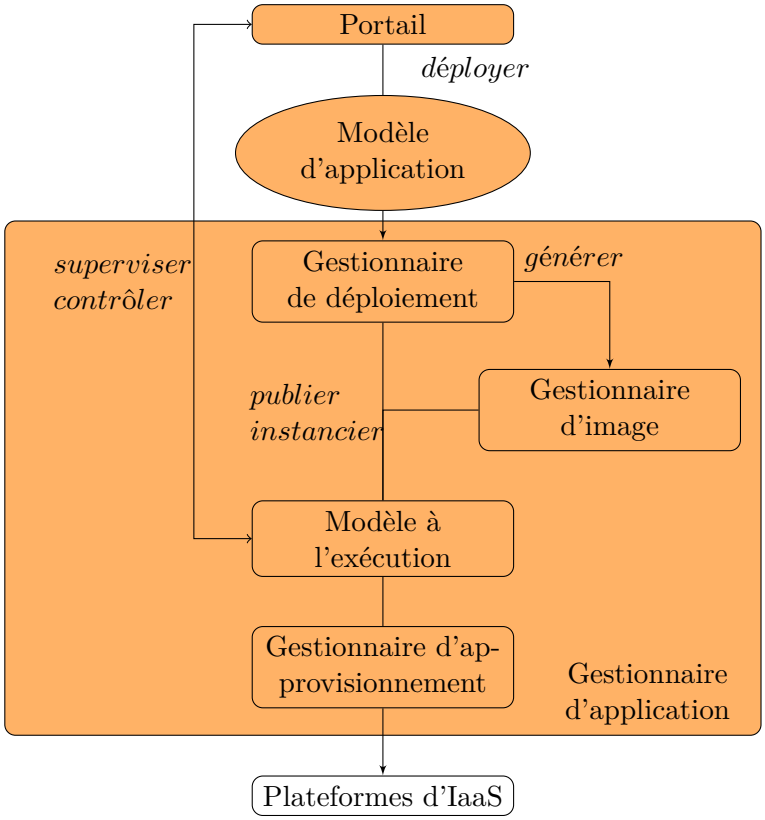

FIGURE 3.5 – Architecture fonctionnelle de VAMP

3.4.1 Portail

Le portail constitue le point d'accès unique à la plate-forme VAMP. Il offre aux utilisateurs la possibilité :

- de demander le déploiement d'une nouvelle application dans le nuage en spécifiant sa modélisation (cf. section 3.3.1) selon le formalisme décrit en section 3.3.1.2. Dès lors, le portail de déploiement crée une nouvelle instance d'un *gestionnaire d'application* (cf. section 3.4.2) au sein d'une machine virtuelle dédiée et lui transmet

la *description applicative* fournie par l'utilisateur. Le portail ne dispose donc d'aucune information applicative et ne conserve que les informations nécessaires pour se connecter au gestionnaire d'application ;

- de contacter un gestionnaire d'application auquel il a accès, afin de superviser et de contrôler le déroulement du déploiement applicatif associé ;

Bien qu'actuellement, le portail soit uniquement dédié au déploiement d'application, il pourrait être étendu pour interagir avec les gestionnaires d'applications pour superviser l'administration d'autres phases du cycle de vie (cf. section 3.4.2).

Le portail peut être exécuté indifféremment au sein d'une machine physique ou d'une machine virtuelle. Etant donné que le portail ne conserve pas d'information relative aux applications qu'il déploie, la manière préconisée pour prendre en compte sa faculté de passage à l'échelle consiste à ajouter dynamiquement de nouvelles instances de portail. En effet, ceci n'impacte nullement la capacité à interopérer entre des applications déployées entre des instances de portail différentes.

L'accès au portail de déploiement peut être interactif (i.e. au travers d'une interface graphique) ou programmatique (i.e. au travers d'un service web)

3.4.2 Gestionnaire d'application

Le gestionnaire d'application est l'entité en charge de l'administration des différentes étapes du cycle de vie d'une instance de l'application. Il se décompose en plusieurs entités qui coopèrent entre elles, comme le détaille cette section.

3.4.2.1 Modèle à l'exécution

Contrairement à ce que son appellation pourrait laisser penser, le modèle à l'exécution est plus qu'une simple structure de données. En effet, il s'agit d'un élément qui :

- réifie l'état courant de l'application au travers d'une vue (i.e. une modélisation) dont la cohérence avec l'instance applicative est maintenue dynamiquement ;

- permet d'agir sur l'état courant de l'application au travers de modifications apportées à la vue. Le modèle à l'exécution les traduit en un ensemble d'ordres de reconfiguration dynamique, transmis à l'application au moyen du gestionnaire d'approvisionnement (cf. section 3.4.2.2).

Le modèle à l'exécution est le cœur de la plate-forme VAMP. Il permet à différents gestionnaires de plus haut niveau, appelés gestionnaires de cycle de vie, d'interagir avec l'instance de l'application. Chacun d'eux est, en effet, en charge de la gestion d'une phase particulière du cycle de vie applicatif. Dans le cadre des travaux présentés dans ce manuscrit, seul un gestionnaire de déploiement a été mis en œuvre (cf. section 3.4.2.3). Néanmoins, l'architecture proposée est bâtie autour du modèle à l'exécution et permet ainsi d'envisager le développement de modules en charge de la fiabilisation de l'application (i.e. gestionnaire de défaillances applicatives), de l'adaptation de son architecture

à la charge à laquelle elle est soumise (i.e. gestionnaire d'élasticité) ou sa sécurité (i.e. gestionnaire de sécurité).

3.4.2.2 Gestionnaire d'approvisionnement

Le gestionnaire d'approvisionnement est l'élément en charge de réaliser l'interface entre la couche *PaaS* et la couche *IaaS*. Il permet de réaliser des opérations simples exposées par cette dernière. Il est ainsi capable de publier une image dans le référentiel d'images d'une plate-forme d'*IaaS* ou de l'en retirer, de créer, démarrer, redémarrer, arrêter, détruire, migrer une machine virtuelle ainsi que d'en modifier les caractéristiques matérielles. Il propose à la couche *PaaS*, une abstraction unifiée indépendante du système d'*IaaS* sous-jacent. Pour cela, il s'appuie sur des connecteurs spécialisés, chacun d'eux assurant la communication avec une plate-forme d'*IaaS* particulière.

3.4.2.3 Gestionnaire de déploiement

Il s'agit de l'un des éléments essentiels de la plate-forme VAMP. En effet, il est en charge d'assurer le déploiement d'une application. A partir d'un modèle statique initial de l'application (cf. section 3.3.1), il :

- délègue, dans un premier temps, au gestionnaire d'image la génération des images (cf. chapitre 4) puis leur publication dans un magasin d'images ;
- pilote alors le modèle à l'exécution pour procéder à l'instanciation des images sous forme de machines virtuelles ;
- transmet à chaque configurateur embarqué sur chaque machine virtuelle, l'extrait du modèle applicatif nécessaire pour finaliser le déploiement de l'application. Cet extrait contient la description architecturale de l'application réduite aux *wrappers* locaux (i.e. présents sur la machine virtuelle) ainsi qu'à leurs voisins. Les voisins, sont les *wrappers* distants (i.e. présents sur d'autres machines virtuelles) dont l'une des interfaces participe à une liaison avec une interface d'un *wrapper* local. L'extrait de modélisation est transmis à l'aide d'un message au travers du bus asynchrone.

Afin de pouvoir s'interfacer avec des systèmes externes (e.g. une console graphique d'administration, un système autonome de haut niveau), un gestionnaire de déploiement expose des interfaces programmatiques de contrôle et de supervision sous la forme d'un service web REST.

3.5 Conclusion

Après avoir rappelé les exigences d'autonomie, de généricité, de fiabilité et de passage à l'échelle nécessaire, selon nous, à l'élaboration d'une plate-forme de déploiement d'application dans le nuage, ce premier chapitre relatif aux contributions de cette thèse s'est attachée à présenter, d'une part, les principes fondamentaux à l'origine de la plate-forme

VAMP, d'autre part, les entités qui la composent et l'architecture selon laquelle ces entités s'organisent pour assurer le déploiement d'une application.

Ainsi, VAMP s'appuie sur un modèle d'application composée d'un modèle d'architecture basée sur le modèle à composant Fractal et l'ADL associé et permettant de définir l'organisation d'une application répartie selon un ensemble d'unités d'exécution, et d'un modèle de machine virtuelle, qui modélise l'infrastructure matérielle et logicielle sur laquelle l'application va être projetée et qui introduit deux extensions au formalisme défini par la spécification OVF.

En outre, VAMP définit une architecture en couche regroupant un ensemble d'entités de contrôle :

- des gestionnaires qui procèdent à la mise en œuvre des phases du processus de déploiement comprises entre la génération des images et leur instanciation au sein de machines virtuelles. Dans un souci de fiabilisation et d'isolation des problématiques de déploiement, il existe un gestionnaire dédié au déploiement de chaque application.

- des configurateurs qui coopèrent entre eux pour parvenir à la finalisation du processus de déploiement (i.e. postconfiguration et démarrage). Leur structure décentralisée (i.e. une instance de configurateur applicative par machine virtuelle) constitue une réponse aux objectifs de fiabilité et de passage à l'échelle de la solution.

- des *wrapper* qui encapsulent les éléments logiciels qui composent l'application à déployer, afin d'en proposer une abstraction uniforme de contrôle et de supervision, répondant ainsi aux objectifs de généricité ;

- un modèle de communication assurant les échanges entre ces entités de contrôle, au moyen de messages échangés au travers d'un bus décentralisé, asynchrone et fiable,

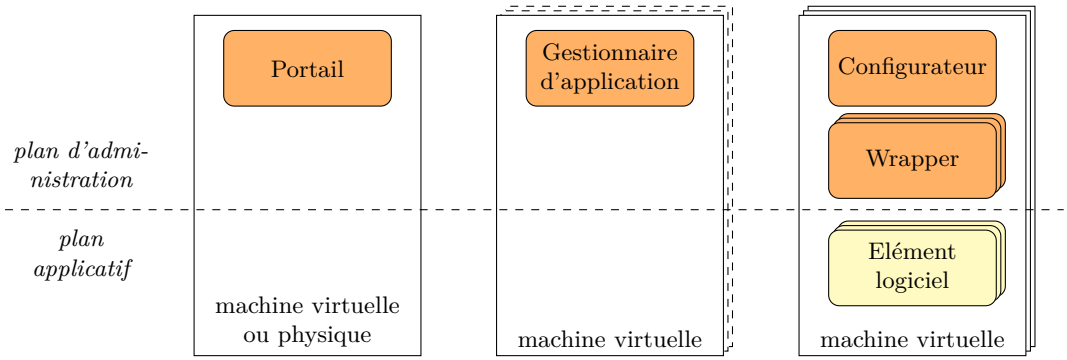

FIGURE 3.6 – Répartition des entités d'administration de VAMP

Chapitre 4

Génération d'appliances virtuelles

Sommaire

4.1 Contexte 98

4.1.1 Modélisation d'image virtuelle 98

4.1.2 Génération d'image virtuelle 99

4.2 Génération d'image dans VAMP 101

4.2.1 Extension OVF 102

4.2.2 Architecture du gestionnaire d'image 105

4.3 Conclusion 106

La première étape du déploiement d'une application dans le nuage, est la phase de packaging. Elle consiste à créer, à partir d'une modélisation de l'application, l'ensemble des images virtuelles qui la constituent. Chacune de ces images contient la totalité de la pile logicielle nécessaire à son fonctionnement. Cette pile comprend le système d'exploitation, les intergiciels ainsi que les binaires et les données relatifs à l'applicatif. En outre, elle intègre tout ou partie des propriétés statiques en lien avec la configuration applicative. Il s'agit des propriétés dont la valeur est indépendante de l'environnement d'exécution[1].

L'objectif de ce chapitre est de présenter la solution adoptée dans VAMP, pour assurer la génération automatisée des images virtuelles. La section 4.1 introduit le contexte en présentant les approches et les modèles relatifs à ce domaine. Par la suite, la section 4.2 précise l'approche adoptée par VAMP et précise les motivations qui ont guidé ce choix, avant de détailler la manière de modéliser une image virtuelle et de décrire l'architecture de la solution. Enfin, la section 4.3 conclut ce chapitre par une synthèse.

[1] A l'inverse, les paramètres dynamiques (i.e. ceux dont la valeur n'est connue qu'une fois que l'image a été instanciée sous forme de machine virtuelle), sont traités par le protocole d'auto-configuration de VAMP, présenté au chapitre 5.

4.1 Contexte

La création d'images virtuelles s'appuie d'une part sur un système de génération, d'autre part, sur une éventuelle modélisation du contenu de l'image à générer. Cette section présente donc les modèles permettant de définir des images (cf. section 4.1.1) ainsi que les solutions qui les mettent en œuvre lors de la création d'images (cf. section 4.1.2).

4.1.1 Modélisation d'image virtuelle

Le manque de maturité qui caractérise le concept d'informatique dans le nuage se traduit par l'absence, quasi totale, de standard et le faible nombre de travaux en matière de formalisation des machines virtuelles et de leur contenu, en vue de leur administration. Il en résulte que la plupart des solutions de *PaaS* n'exposent pas à l'utilisateur de véritable modèle de données. Cependant, quelques initiatives tentent désormais d'adresser cet aspect. Ainsi, certaines sont issues du niveau applicatif, d'autres proviennent du niveau infrastructure et les dernières correspondent à des travaux liés à la configuration de systèmes, antérieurs à l'apparition de l'informatique dans le nuage. L'objectif de cette section est de présenter ces trois types d'approches.

4.1.1.1 Modélisation issue du niveau applicatif

Dans le contexte de l'informatique dans le nuage, la finalité de ce premier ensemble de travaux est la mise à disposition rapide de nouveaux services à destination d'utilisateurs finaux. C'est ainsi le cas de l'*Application Packaging Standard* (*APS*) [19] proposé à l'origine par la société Parallels et désormais porté par une organisation à but non lucratif. Il s'agit de la spécification d'un formalisme ouvert et extensible qui permet de modéliser un service fourni (ou application) comme résultant de la composition d'un ensemble d'autres services requis. Chaque service est défini au travers :

- des métadonnées qui lui sont propres (e.g. son identifiant, son nom, son URL d'accès) ;
- de la description des ressources sur lesquelles il peut / doit être instancié ;
- des services qui le composent ;
- d'un ensemble de scripts liés aux différentes phases du cycle de vie du service (e.g. création, démarrage, arrêt, suppression).

APS peut être vu comme une spécialisation (e.g. limitation à la composition de services) et une extrême simplification (e.g. pas de vision architecturale réelle, pas d'introspection, pas de réflexivité) de modèles à composants tel que Fractal [37] et des langages de description d'architecture qui leurs sont associés.

4.1.1.2 Modélisation issue du niveau infrastructure

Cet ensemble regroupe des initiatives inspirées du niveau infrastructure. Il s'agit de propositions telles que celles issues de la *Distributed Management Task Force* (*DMTF*) comme les standards ouverts et extensibles Cimi (en cours de définition) [58] ou OVF [59]. Initialement, l'objectif de ce type d'approches est de décrire un ensemble cohérent de machines virtuelles (qualifié de *system* dans la terminologie Cimi) basé sur des images virtuelles préexistantes. En outre, la notion d'application demeure limitée et se résume à l'expression de dépendances à une granularité de niveau machines virtuelles.

4.1.1.3 Modélisation de la configuration de systèmes

Ce troisième type de modélisation concerne les solutions de configuration de systèmes antérieures au concept d'informatique dans le nuage. Il s'agit de solutions telles que Puppet [104], Chef [101] et celles, plus anciennes, dont elles sont les héritières comme BCFG [55]. Ces solutions consistent à installer et à configurer dynamiquement un système (i.e. ici, la machine virtuelle correspondant à l'instanciation de l'image virtuelle considérée). Pour cela, un agent responsable de l'installation et de la configuration est instancié au sein du système lui-même. Il s'appuie sur un formalisme assez riche qui lui permet notamment de gérer des notions aussi variées que le téléchargement et l'installation de paquets, la définition et la consultation de variables d'environnement, le positionnement de droits d'accès, l'arrêt et le démarrage de processus, etc. Ainsi, dans le cas de Puppet, il s'agit d'une description du système selon une syntaxe XML, qui confère au gestionnaire de configuration une certaine indépendance vis-à-vis du système d'exploitation sous-jacent. A l'heure actuelle, Puppet et Chef constituent des standards de fait en matière de configuration dynamique de système d'exploitation et font souvent l'objet d'intégration ou de perspectives d'intégration, au sein de travaux de recherche [96] [73].

4.1.2 Génération d'image virtuelle

Bien qu'une *image virtuelle* désigne un fichier contenant l'intégralité de la pile logicielle d'une machine et puisse être vue comme le contenu de son disque, le terme *génération d'image virtuelle* couvre une réalité beaucoup plus vaste et plus floue. En effet, cela inclut des solutions très disparates, allant d'une simple duplication d'un système préinstallé existant, à la création complète d'une image de manière statique en passant par l'installation dynamique de logiciels sur une machine virtuelle correspondant à l'instanciation d'une image ne comprenant qu'un système d'exploitation nu. L'objet de cette section est donc de détailler les caractéristiques ainsi que les forces et les faiblesses de ces trois types de solutions.

4.1.2.1 Génération par conversion de machine existante

Cette première approche consiste à créer une image virtuelle par conversion d'un système installé préexistant. Pour cela elle repose sur le principe de migration[2] de machine entre le monde réel et le monde virtuel. Ce principe se décline selon trois méthodes :

P2V (*physical to virtual*) : conversion d'une machine physique en une machine virtuelle ;

V2P (*virtual to physical*) : conversion d'une machine virtuelle en une machine physique ;

V2V (*virtual to virtual*) : conversion d'une machine virtuelle d'un format dans un autre.

Dans le contexte de la génération d'images virtuelles, seules les méthodes P2V et V2V présentent un intérêt. Elles font l'objet d'implémentation plus ou moins complètes au sein d'outils tels que VMWare vCenter Converter [127], Citrix XenConvert [118] ou Vizioncore vConverter [116]. Intrinsèquement, l'image cible correspondant à la reproduction de système source, cette approche n'a pas vocation à offrir à l'utilisateur des capacités de paramétrisation de la pile logicielle à générer. Elle ne propose aucune modélisation du contenu d'une image.

4.1.2.2 Génération par installation dynamique d'une machine virtuelle

Cette approche consiste à installer dynamiquement, c'est-à-dire une fois que l'image virtuelle a été instanciée sous forme de machine virtuelle, les éléments logiciels nécessaires. Elle reproduit ainsi le principe d'installation d'une machine en environnement non virtualisé. L'image virtuelle initiale se réduit à un *agent d'installation* et à son *environnement d'exécution*. Lors de l'instanciation de cette image, l'agent est automatiquement activé et peut alors procèder aux opérations d'installation et/ou de configuration nécessaires à la construction de la pile logicielle relative à l'environnement d'exécution dans lequel il se trouve.

L'image initiale, qui correspond à l'environnement d'exécution de l'agent, est généralement constituée d'une installation minimale d'un système d'exploitation. En fonction des besoins de configuration, l'agent peut alors se limiter à un gestionnaire de paquets (e.g. aptitude sur souche Linux Debian, yum sur souche Linux RedHat) ou il peut s'agir d'un agent de configuration plus sophistiqué (e.g. [104][101][10]). Cependant, de telles solutions demeurent adhérentes au système d'exploitation sous-jacent dans la mesure où l'agent ne procède pas à l'installation de cet élément de base de la pile logicielle (e.g. cas de rPath [10] vis-à-vis de Linux RedHat).

[2]Ici, le terme migration fait référence au principe de poursuite d'activité tout en faisant évoluer un système d'un environnement physique vers un environnement virtuel ou inversement. Il ne s'agit nullement de l'opération qui consiste à déplacer une machine virtuelle depuis une machine physique ver une autre machine physique

Pour contourner cette difficulté, l'agent et son environnement d'exécution peuvent également être réduits à leur plus simple expression, comme dans la proposition de [96], dans laquelle l'agent initial est vu comme la capacité qu'a la machine virtuelle nue à se connecter à un serveur d'amorçage réseau (i.e. de type PXE *Preboot eXecution Environment*). Dès lors, au moment du démarrage, la machine virtuelle se connecte au serveur qui assure l'installation du système d'exploitation et son initialisation. A partir de là, la poursuite de l'installation et de la configuration s'opèrent comme dans le cas précédent.

Ce type d'approche se caractérise donc par le fait qu'elle ne génère pas, à proprement parler, une image virtuelle instanciable, mais la possibilité de disposer d'une machine virtuelle installée. Ainsi, toute instanciation d'une nouvelle machine virtuelle, débute par une phase d'installation et de configuration des logiciels nécessaires. Cette phase est assurée par un agent exposant une modélisation du système et des capacités d'interaction.

4.1.2.3 Génération par construction statique d'image virtuelle

A l'inverse de la génération par installation dynamique d'une machine virtuelle, cette approche a pour but de mettre à disposition de l'utilisateur une image virtuelle sans procéder, préalablement, à son instanciation sous forme de machine virtuelle. L'opération de génération consiste donc à créer un fichier contenant l'ensemble des logiciels et des données que l'utilisateur veut embarquer dans l'image.

Ce type d'approche est celui adopté par des solutions telles que Kiwi [111], Novell SUSE Studio [97], Usharesoft UForge [123] ou VMBuilder[3]. Néanmoins, bien que toutes permettent à l'utilisateur de personnaliser le contenu de l'image à générer, il existe d'importantes différences entre ces solutions. Ainsi, certaines ne sont en réalité que des scripts d'orchestration basés sur les systèmes de gestion de paquets des systèmes d'exploitation sur lesquels elles s'exécutent. C'est le cas de VMBuilder qui utilise le gestionnaire de paquets aptitude et ne s'exécute que sur Linux Ubuntu. A l'inverse, les solutions qui prennent en charge la définition et l'administration de leur propre système de paquets sont qualifiées de "*forges*". Cependant, des forges comme Kiwi ou Novell SUSE Studio n'en demeurent pas moins liés à un système d'exploitation spécifique (i.e. en l'occurrence, Linux Open SUSE)[4]. Seul Usharesoft UForge est indépendant du système d'exploitation servant de base à l'image à générer.

4.2 Génération d'image dans VAMP

L'automatisation de la phase de packaging nécessite que le gestionnaire d'image de VAMP (cf. section 3.4) soit en mesure, à partir de la description des piles logicielles applicatives fournies par l'utilisateur, de piloter une solution (externe à VAMP) de génération d'image. Ceci disqualifie d'emblée l'approche par conversion de machine existante

[3] `https://launchpad.net/vmbuilder`

[4] A noter que rPath, qui est pourtant une solution à base d'installation dynamique de machine virtuelle, propose son propre gestionnaire de paquets appelé Conary [129]

(cf. section 4.1.2.1) dont l'une des principales caractéristiques est de ne pas offrir de capacité à définir le contenu de l'image à générer.

D'autre part, la volonté d'automatisation du packaging ne doit en aucun cas altérer les capacités de contrôle du système par un administrateur. Afin d'adresser les problématiques des environnements de production, le gestionnaire d'image doit donc mettre à disposition de l'administrateur système des images dont le contenu soit *certifié*. Cette *certification* correspond à une notion de garantie quant à :

- leur contenu : chaque paquet (i.e. chaque binaire) qui compose l'image doit être installé de façon correcte et certaine ;
- leur cohérence : la compatibilité de versions entre l'ensemble des paquets d'une même image doit être assurée ;
- leur déterminisme (ou pérennité) : une même description, n'ayant fait l'objet d'aucune modification, doit être regénérée de manière identique (i.e. des images identiques) au cours du temps. En d'autres termes, le résultat de la résolution des dépendances de paquets est toujours la même.

En outre, pour permettre à un administrateur de s'assurer du respect de ces propriétés par le contrôle du contenu d'une image, il est nécessaire que le processus d'auto-déploiement puisse être suspendu avant qu'elle ne soit instanciée sous forme de machine virtuelle. Dès lors, l'approche par installation dynamique de machine virtuelle (cf. section 4.1.2.2) s'avère inutilisable dans le contexte de VAMP. En effet, la pile logicielle n'étant complétée qu'au moment de l'instanciation de l'image sous forme de machine virtuelle, il n'est pas possible d'effectuer de telles opérations de contrôle.

Ainsi, seules les solutions qui adoptent l'approche par construction statique d'image virtuelle répondent à la nécessité de contrôle de l'image tout en offrant la possibilité d'en définir le contenu (cf. section 4.1.2.3). Cependant, parmi elles, seules les forges conservent la maîtrise sur le référentiel d'installation servant à générer les images. Or, sans cette maîtrise, la propriété de pérennité au cours du temps ne peut être garantie. Enfin, parmi les forges, Usharesoft UForge est la seule qui soit agnostique au système d'exploitation utilisé dans la définition d'une pile logicielle. C'est pour cela que c'est la solution qui a été retenue pour s'interfacer avec le gestionnaire d'images de VAMP.

La suite de cette section va donc décrire, d'une part, la manière de modéliser une image virtuelle dans VAMP (cf. section 4.2.1), d'autre part, l'architecture de la solution de génération automatisée d'image proposée (cf. section 4.2.2).

4.2.1 Extension OVF

Comme il a été indiqué précédemment (cf. section 4.1.1.1), les propositions de modélisation issues du niveau applicatif permettent de définir un nouveau service en décrivant de façon explicite la manière de composer des services existants. Néanmoins, ce type de formalisme ne propose pas pour l'instant d'exposer la notion de machine virtuelle ou celle d'image virtuelle nécessaire à la définition d'un service élémentaire.

De même, bien que les initiatives issues du niveau infrastructure (cf. section 4.1.1.2) prennent en considération le concept de machine virtuelle, il n'en demeure pas moins qu'elles s'appuient sur une vision statique du contenu des images virtuelles. Ainsi, celles-ci ne sont modélisées qu'à l'aide d'un fichier image préexistant.

Enfin, concernant les formalismes issus de la configuration dynamique de système (cf. section 4.1.1.3), ils offrent certes un fort niveau d'expressivité mais sont limités selon deux axes :

- d'une part, ils permettent de configurer l'ensemble de la pile logicielle d'une machine virtuelle déjà instanciée, mais ils ne disposent pas nécessairement d'une vue architecturale globale d'une application répartie sur plusieurs machines ;
- d'autre part, l'agent d'installation et de configuration sur lequel ce type de modèle repose est embarqué au sein de l'environnement d'exécution à générer. Il n'a donc aucun moyen de positionner des propriétés relatives aux caractéristiques matérielles virtualisées de la machine (e.g. nombre de cpu, taille mémoire, etc.).

L'une des contributions de ce travail a donc été de proposer un formalisme capable de faire face à ces différentes limitations. Comme indiqué en section 3.3.1.2, l'objectif était de proposer un modèle à la fois exhaustif vis-à-vis des besoins courants, ouvert et extensible et basé sur des standards (à l'inverse de ce que propose [47]). Le socle utilisé a donc été le standard OVF, qui permet de décrire l'infrastructure matérielle virtualisée. Ce standard a été étendu, d'une part, à l'aide d'un formalisme, lui-même standard (i.e. le langage de description d'architecture Fractal ADL), permettant d'offrir une vue de l'architecture applicative de la solution à déployer (i.e. balise `AppArchitectureSection`), d'autre part, d'une extension permettant de formaliser le contenu statique des images virtuelles (e.g. balise `DynamicImage`). Cette seconde extension ne s'est pas basée sur une approche de type configuration de système préexistant, tel que la syntaxe proposée par Puppet. En effet, la richesse d'expressivité d'une telle solution a été jugée excessive par rapport au besoin réel. Néanmoins une seconde implémentation de VAMP utilise ce type de formalisme.

La section `DynamicImage` modélise le contenu d'une image à l'aide des éléments suivants (cf. listing 4.1 d'exemple) :

- un système d'exploitation (tag `OperatingSystem`) avec son type (attribut `pm:os`), l'identifiant de sa distribution (attribut `pm:id`), sa version (attribut `pm:version`) et son architecture (attribut `pm:architecture`) ;
- la liste des produits qui doivent être ajoutés. Chacun d'eux est décrit à l'aide du tag `Package` qui spécifie un identifiant unique (attribut `pm:id`) et une version (attribut `pm:version`) qui font référence au paquet logiciel associé ;
- la liste des données utilisateurs, composée d'un ensemble de fichiers. Chaque fichier est décrit à l'aide du tag `Data` qui précise :
  - son identifiant (attribut `pm:id`) ;

- son type (attribut `pm:type`). Cette propriété permet d'adapter le comportement du gestionnaire en charge de la génération d'image en fonction du type de fichier manipulé. Ainsi, dans sa version courante, VAMP distingue deux types de fichiers utilisateur. D'une part ceux qui contiennent des données, d'autre part des scripts exécutables (i.e. type *sh*) ;
- l'emplacement où se trouve le fichier au moment de la création de l'image (attribut `pm:sourcePath`), en vue de sa copie dans l'image ;
- l'emplacement où le fichier doit être installé dans l'image (attribut `pm:destinationPath`). Plus exactement, cette donnée est interprétée par le gestionnaire en charge de la génération d'image en fonction du type du fichier. Dans le cadre de VAMP, un fichier de données sera copié vers l'emplacement spécifié. En revanche, s'il s'agit d'un script, ce champ spécifie son mode de mise en œuvre : exécution déclenchée lors de la première initialisation de la machine virtuelle (e.g. FIRSTBOOTSCRIPT), à chaque initialisation (e.g. EVERYBOOTSCRIPT) ou à chaque initialisation à l'exception de la première (e.g. AFTERREBOOTSCRIPT) ;
- la version du fichier (attribut `pm:version`) ;

```
...
<!-- content of the Apache appliance -->
<DynamicImage ovf:id="appDisk2"
              pm:userDisk="appDisk2" ...>
 <OperatingSystem pm:os="Linux"
                  pm:id="CentOs"
                  pm:version="5.5"
                  pm:architecture="x86_64" />
 <Packages>
  <Package pm:id="Apache HTTP server"
           pm:version="2.2.3"/>
  ...
 </Packages>
 <DataSet>
  <Data pm:id="log4j"
        pm:type="properties"
        pm:sourcePath="..."
        pm:destinationPath="/vamp/"
        pm:version="1.0"/>
  <Data pm:id="vamp-start"
        pm:type="sh"
        pm:sourcePath="..."
        pm:destinationPath="[AFTERREBOOTSCRIPT]"
        pm:version="1.0"/>
  ...
 </DataSet>
</DynamicImage>
...
```

Listing 4.1 – Extrait du modèle d'image du tiers de présentation de Springoo

4.2.2 Architecture du gestionnaire d'image

Comme l'illustre la figure 4.1, le gestionnaire d'image repose sur trois types d'entité :

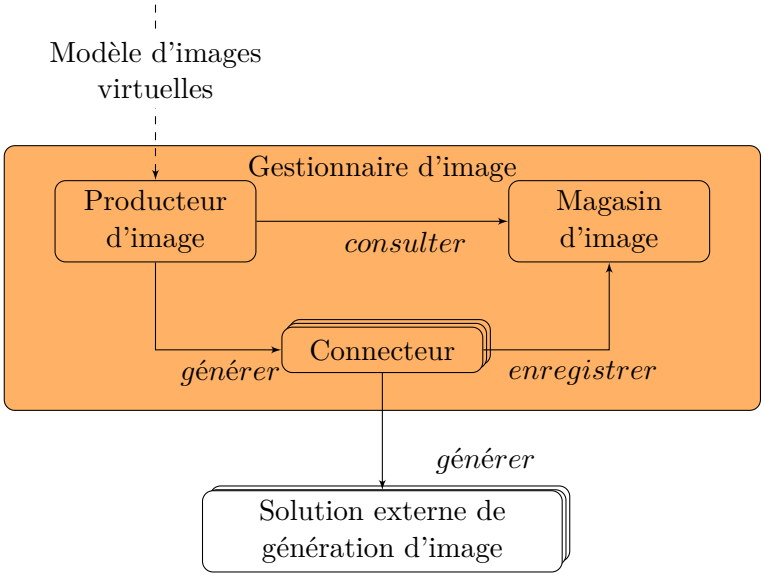

FIGURE 4.1 – Architecture fonctionnelle du gestionnaire d'image

- **un magasin d'images :** son rôle est d'assurer le stockage et la consultation d'images déjà générées. Ainsi, il permet d'indexer et d'enregistrer une image et de la retrouver à partir de sa description. Il offre également la capacité de convertir une image d'un format dans un autre, en vue de sa publication dans des infrastructures d'instanciation (i.e. plateformes d'*IaaS*) différentes. La conversion de format s'appuie sur des outils externes de type *virtual to virtual* (*V2V*).

- **des connecteurs :** chacun d'eux expose une abstraction commune, utilisée par le système VAMP, vis-à-vis d'une solution externe spécifique de génération d'image.

- **un producteur d'image :** son objectif est d'orchestrer la génération des images lors du déploiement d'une application, selon le processus suivant (cf. figure 4.2) :

  1. A partir de la liste des machines virtuelles contenues dans la description de l'application selon le formalisme décrit en section 3.3.1.2, le producteur d'images détermine l'ensemble des images qui doivent être mises à disposition du gestionnaire d'application. Celles-ci sont décrites au sein des sections `DynamicImage` de l'OVF étendu.

2. Pour chaque image, il s'adresse alors au magasin d'images afin de déterminer si celle-ci a déjà fait l'objet d'une précédente génération ou si, au contraire, il s'agit d'une première génération.
3. Dans le cas d'une première génération, le générateur d'images s'adresse, au travers de l'interface unifiée, au connecteur servant de frontal système externe de génération d'images qui doit être utilisé. Plus précisément, il interprète le formalisme de description d'image pour le convertir en commandes de pilotage de la forge. Afin de rendre chaque machine virtuelle auto-configurable, VAMP y inclut également un agent configurateur (cf. section 3.3.2.1). Une fois crée, chaque nouvelle image est stockée dans le magasin d'images et le générateur d'image est capable d'indiquer au gestionnaire d'application la manière d'y accéder (e.g. URL). Le format de l'image est déterminé en fonction de la plate-forme d'*IaaS* qui sera utilisée pour l'instancier.
4. Dans le cas d'une image déjà existante dans le magasin d'image (i.e. ayant été créée antérieurement), celle-ci peut faire l'objet d'une conversion dans le format de sortie spécifié par le générateur d'images. Le générateur d'images récupère et retransmet au générateur d'applications la manière d'y accéder pour la récupérer au sein du magasin.

Parmi les outils externes de génération d'images, certains sont capable d'assurer la publication d'images dans une infrastructure d'*IaaS* ou non. Le gestionnaire d'image et le gestionnaire d'application sont en mesure de prendre en considération ces différences afin d'en tirer une meilleure utilisation des fonctionnalités de chaque solution.

Dans le souci constant d'isolation entre applications, il existe une instance de gestionnaire d'image par gestionnaire d'application. Cette instance est créée sur une machine virtuelle qui lui est propre et sa fiabilisation est identique à celle de n'importe quelle machine virtuelle applicative (cf. section 6.2).

4.3 Conclusion

La seconde contribution de ces travaux de thèse concerne l'automatisation de la phase de mise à disposition des images virtuelles qui composent une application à déployer. Il s'agit :

- d'un modèle permettant d'en spécifier le contenu en termes de système d'exploitation, d'intergiciels ainsi que de données et de binaires utilisateur ;
- d'un gestionnaire d'image capable, à partir du modèle précédent, de piloter une forge externe en vue d'obtenir des images certifiables en termes de contenu, de cohérence entre les entités qui composent l'image et de pérennité vis-à-vis de générations successives au fil du temps.

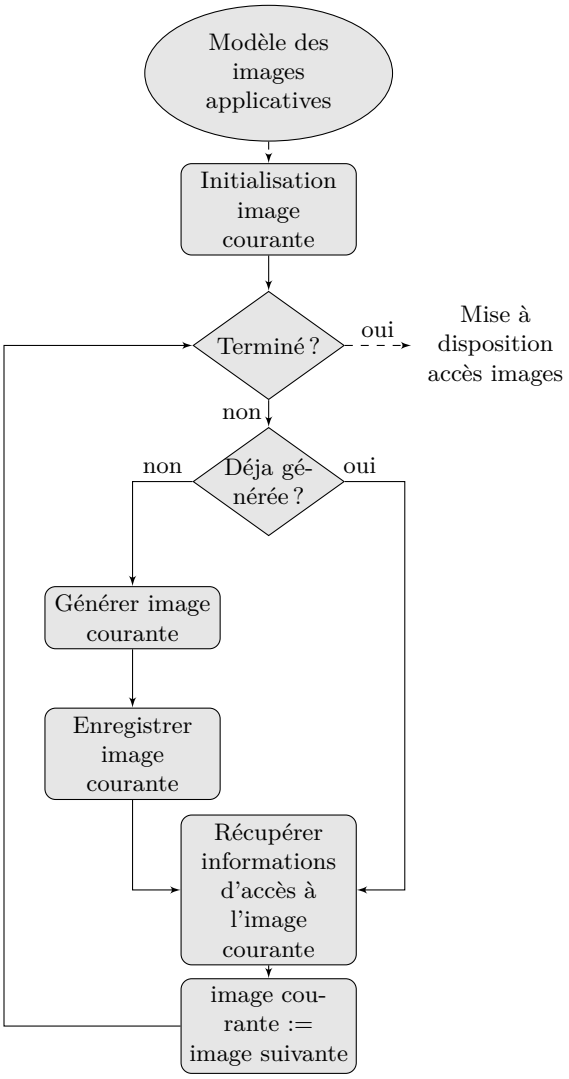

FIGURE 4.2 – Algorithme mis en œuvre par le processus automatisé de génération d'images virtuelles

Chapitre 5

Protocole d'auto-configuration

Sommaire

5.1 Contexte de l'environnement de communication AAA 110
5.2 Instanciation du bus à messages 112
5.2.1 Configuration initiale du bus à messages 112
5.2.2 Mise à jour dynamique du bus à messages 114
5.3 Protocole d'auto-configuration et d'auto-activation 116
5.3.1 Description du protocole d'auto-configuration 116
5.3.2 Vérification 120
5.4 Conclusion 121

La plateforme VAMP dispose d'un point d'entrée unique, permettant à chaque utilisateur de demander le déploiement de ses applications (cf. section 3.4). Ce point d'accès est matérialisé par le portail. A chaque nouvelle demande de déploiement, son rôle est d'instancier un gestionnaire d'application dédié à l'application considérée et de lui transmettre la description fournie par l'utilisateur, selon le formalisme défini en section 3.3.1.2. Le gestionnaire d'application inclut un gestionnaire de déploiement responsable du déploiement initial de l'application. Le gestionnaire de déploiement procède ainsi à la génération automatique des images virtuelles qui composent l'application (cf. chapitre 4) puis à leur publication dans une plateforme d'*IaaS* et à leur instanciation sous forme de machines virtuelles.

Dès lors, chaque machine virtuelle s'exécute de manière indépendante vis-à-vis des autres. Une fois la phase d'initialisation (*boot*) terminée, un processus Java est automatiquement activé. Son rôle est de finaliser le déploiement des composants applicatifs présents dans la machine virtuelle. Pour cela il implémente un protocole comportant deux étapes :

Définition du support de communication distribué, asynchrone et fiable : il instancie un serveur d'agents et procède à son intégration dans le bus à messages qui

assure les échanges entre les entités d'administration de VAMP impliquées dans les étapes de postconfiguration et d'activation. Il s'agit du gestionnaire d'application et de chaque agent de configuration inclus dans chaque machine virtuelle applicative.

Postconfiguration et activation de l'application : il crée, au sein du serveur d'agents, un agent de configuration (également appelé configurateur) dont le rôle est de finaliser le déploiement de l'application. Pour cela, il s'appuie sur un protocole d'auto-configuration et d'auto-activation des *wrappers* embarqués dans la machine virtuelle. Il s'agit d'un protocole collaboratif, réparti sur l'ensemble des configurateurs. Ce protocole constitue une des principales contributions de cette thèse.

Ce chapitre s'organise de la manière suivante. Dans un premier temps, la section 5.1 introduit le contexte du bus à messages AAA, retenu comme support de communication dans l'implémentation courante de VAMP. A une instance d'application est associée une instance de bus, répartie sur les entités d'administration (i.e. configurateur et gestionnaire d'application) en charge du déploiement de l'application. Par la suite, la section 5.2 décrit la manière dont une instance du bus AAA est mise en œuvre dans le cadre d'un déploiement applicatif. La section 5.3 détaille alors la seconde phase du protocole, qui assure la postconfiguration et l'activation automatisée de l'application. Enfin, la section 5.4 conclut le chapitre.

5.1 Contexte de l'environnement de communication AAA

Le déploiement d'une application au moyen de VAMP passe par la mise en œuvre orchestrée de plusieurs traitements répartis au sein d'un ensemble d'entités d'administration (cf. section 3.4). Elle se concrétise par un ensemble d'interactions entre les entités d'administration. L'un des principes fondamentaux de VAMP est un modèle de communication basé sur un bus à messages [24] asynchrone, distribué et fiable, capable d'assurer les échanges induits par ces interactions (cf. section 3.3.3). L'objectif de cette section est donc de présenter le bus à messages AAA (*Agent Anytime Anywhere*) [27] retenu comme support de communication dans le cadre de l'implémentation de référence de VAMP.

Le bus AAA est un intergiciel codé en Java qui constitue le cœur de Joram [6], l'implémentation *open source* de JMS (*Java Messaging Service*) [5][1] proposée par le consortium OW2.

Une instance de bus AAA est constituée d'un ensemble de serveurs d'agents, au sein desquels sont instanciés des agents. Un agent désigne un objet Java qui se conforme au modèle de programmation événement-action. Il implémente ainsi une méthode qui définit son comportement lors de la réception d'un message. Un serveur d'agents, pour sa part, a pour but d'aiguiller les messages en provenance ou à destination d'un agent dont il a la charge. Plus précisément, lorsqu'un des agents dont il a la responsabilité émet un

[1] JMS est une interface de programmation d'application (API) définissant un modèle de communications asynchrone.

message, il transmet celui-ci au serveur d'agents responsable de l'agent destinataire du message. A l'inverse, lorsqu'il reçoit d'un autre serveur d'agents un message à destination de l'un de ses agents, il le lui met à disposition. Pour cela, un serveur d'agents met en œuvre une queue de messages sortant Q_out et une queue de messages entrants Q_in. Un serveur d'agents est caractérisé par :

- un identifiant logique unique permettant de le référencer dans l'instance de bus AAA ;
- le nom ou l'adresse IP de l'hôte sur lequel il est instancié ;
- un port TCP d'écoute au travers duquel il reçoit les messages provenant des autres serveurs d'agents présents dans l'instance de bus AAA ;
- un ensemble de services optionnels activés au sein du serveur d'agents (e.g. ajout d'un nouveau serveur d'agents dans le bus, suppression d'un serveur d'agents existant).

La topologie d'une instance de bus est l'ensemble des serveurs d'agents qui constituent cette instance. Elle peut être décrite statiquement ou construite dynamiquement :

la déclaration statique consiste à initialiser un serveur d'agents à partir d'une description, totale ou partielle, de l'instance du bus AAA. Il s'agit d'un fichier au format XML (cf. listing 5.1) qui définit un ensemble de serveurs d'agents (balise `server`) en termes d'identifiant logique unique (attribut `id`), d'hôte (attribut `host`), de port d'écoute (attribut `port` de la balise `network`) et de services activés (balise(s) `service`).

la déclaration dynamique permet de modifier la définition d'une instance de bus AAA existante. Il est ainsi possible de déclarer un nouveau serveur d'agents, d'en supprimer un existant ou d'obtenir un identifiant de serveur d'agents disponible. Ces opérations sont implémentées au sein d'une API Java basée sur un protocole de communication propriétaire. Elles sont réalisées au sein de n'importe quel serveur d'agents déjà présent dans le bus et qui expose le service adéquat. Une fois la modification réalisée, le serveur d'agents sollicité se charge d'en informer l'ensemble des serveurs d'agents présents dans le bus, à l'aide d'un mécanisme de diffusion d'événements. Lors de la réception d'un événement de mise à jour, un serveur d'agents modifie la représentation interne qu'il a du bus AAA et, dans le cas de la suppression, il purge sa queue d'émission Q_out des messages à destination d'un agent déployé sur le serveur retiré.

```
<?xml version="1.0"?>
<config>
  ...
  <server id=... name=... hostname=...>
    <network domain=... port=.../>
```

```
    <service class=... args=.../>
    ...
  </server>
</config>
```

Listing 5.1 – Extrait de la description statique de la topologie du bus à messages AAA

La technologie de bus AAA se conforme aux exigences du modèle de communication de VAMP (cf. section 3.3.3) :

Distribution de la logique de fonctionnement du bus au sein de l'ensemble des serveurs d'agents ;

Asynchronisme entre l'émetteur et le récepteur d'un message ;

Fiabilité des messages échangés et de l'état des agents, qui font l'objet de mécanismes de persistance au niveau des serveurs d'agents ;

Ordonnancement des messages transmis entre une source et une destination données ;

Atomicité dans la mise en œuvre de la réaction d'un agent à un message.

5.2 Instanciation du bus à messages

Afin de proposer un modèle de communication asynchrone, décentralisé et fiable, capable de supporter les interactions entre entités d'administration, VAMP s'appuie sur la technologie de bus à messages AAA (cf. section 5.1). A chaque nouvelle instance d'application à déployer, il crée une instance de bus dédiée à cet effet. La mise en œuvre d'une telle instance se déroule en deux phases. La première correspond à la configuration statique initiale (*bootstrap*) du bus, la seconde, à sa mise à jour dynamique, au gré de l'arrivée (e.g. insertion d'une nouvelle machine virtuelle applicative lors du déploiement initial) et du départ (e.g. disparition d'une machine virtuelle applicative suite à sa défaillance) des pairs.

Cette section décrit la manière de construire l'instance de bus à messages entre les entités d'administration VAMP impliquées dans le déploiement d'une application. Elle se subdivise en une première sous-section consacrée à la phase de *bootstrap* du bus, puis en une seconde traitant de la mise à jour dynamique de sa topologie en fonction de l'arrivée progressive des serveurs d'agents[2].

5.2.1 Configuration initiale du bus à messages

Dans un souci d'isolation entre applications, le processus de déploiement proposé dans VAMP est partitionné.

L'un des principaux intérêts de la virtualisation des ressources matérielles physiques est d'en permettre l'utilisation concurrente tout en maintenant un niveau d'isolation,

[2]La gestion du retrait d'un serveur d'agents de l'instance du bus sera détaillée dans le chapitre 6 dédié à la fiabilité

en termes de performances, de fiabilité, de sécurité, comparable à celui de ressources matérielles physiques distinctes.

Afin d'étendre cette propriété au niveau applicatif, le processus de déploiement de VAMP est partitionné de sorte que deux entités d'administration impliquées dans le déploiement de deux applications distinctes ne puissent pas communiquer. Ainsi, en réponse à toute demande de déploiement d'une application, le portail de déploiement de VAMP crée une nouvelle instance de gestionnaire d'application. Celle-ci inclut un gestionnaire de déploiement dédié à cette application. Il s'agit d'un agent instancié dans un serveur d'agents au sein d'un processus Java et d'une machine virtuelle qui lui sont propres. Ce serveur d'agents constitue l'entité d'amorçage d'une instance de bus à messages dédiée à l'administration de l'application.

Il est configuré statiquement (cf. section 5.1) au moyen d'un fichier de configuration (cf. listing 5.2).

```
<?xml version="1.0"?>
<config>
  ...
  <server id="3" name="deployment-mgr1" hostname="192.168.1.25">
    <network domain="VAMP_APP_001" port="25576"/>
    <service
        class="org.objectweb.joram.mom.proxies.ConnectionManager"
        args="root root"/>
    <service
        class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
        args="29204"/>
  </server>
</config>
```

Listing 5.2 – Configuration statique pour l'initialisation du bus à messages associé à une application

Ce fichier est généré automatiquement, au terme de l'initialisation de la machine virtuelle, afin de prendre en compte certaines propriétés de configuration dont la valeur ne peut être connue avant : c'est par exemple le cas de l'adresse IP de la machine virtuelle (e.g. 192.168.1.25), qui peut être allouée dynamiquement au moyen du protocole DHCP. L'identifiant de ce serveur d'agents (e.g. id égal à 3) est choisi arbitrairement par le portail de déploiement. Dans la mesure où ce serveur d'agents est initialement seul dans une instance de bus donnée, dédiée à l'application, la valeur de l'identifiant est unique[3]. Enfin, pour permettre d'ajouter ou de supprimer dynamiquement un serveur d'agents, le service d'administration de la topologie du bus (e.g. TcpProxyService) est activé sur ce premier serveur d'agents.

Une fois cette phase d'initialisation terminée, le gestionnaire de déploiement dédié à l'application peut alors être instancié au sein de ce premier serveur d'agents. Il génère

[3]La section 6.3.2 proposera une fiabilisation du gestionnaire de déploiement s'appuyant sur sa duplication. Néanmoins, il sera démontré que l'unicité des identifiants de serveurs d'agents dans le bus n'en est nullement affectée.

alors automatiquement les images virtuelles qui composent l'application (cf. chapitre 4), les publie dans une plateforme d'*IaaS* puis les instancie sous forme de machines virtuelles. Au cours de ces étapes, elle enrichit sa représentation interne du modèle d'exécution, en positionnant pour chaque machine virtuelle, de façon unique, l'identifiant du serveur d'agents qui lui est associée (i.e. il s'agit du champ `agentServerId` de l'objet `VirtualNode` (cf. section 3.3.1)).

5.2.2 Mise à jour dynamique du bus à messages

Au cours de cette seconde phase dans la construction du bus à messages, chaque machine virtuelle s'exécute de façon autonome, indépendamment des autres. Une fois la phase d'initialisation terminée, un processus Java est automatiquement démarré. Son rôle est d'activer le configurateur embarqué dans la machine virtuelle. Ce dernier est un agent qui doit donc être créé au sein d'un serveur d'agents. Le processus Java doit donc créer ce serveur d'agents, procéder à son intégration dans le bus à messages dédié à l'application, puis créer en son sein un agent de configuration. Pour cela, il opère selon les étapes représentées dans la figure 5.1.

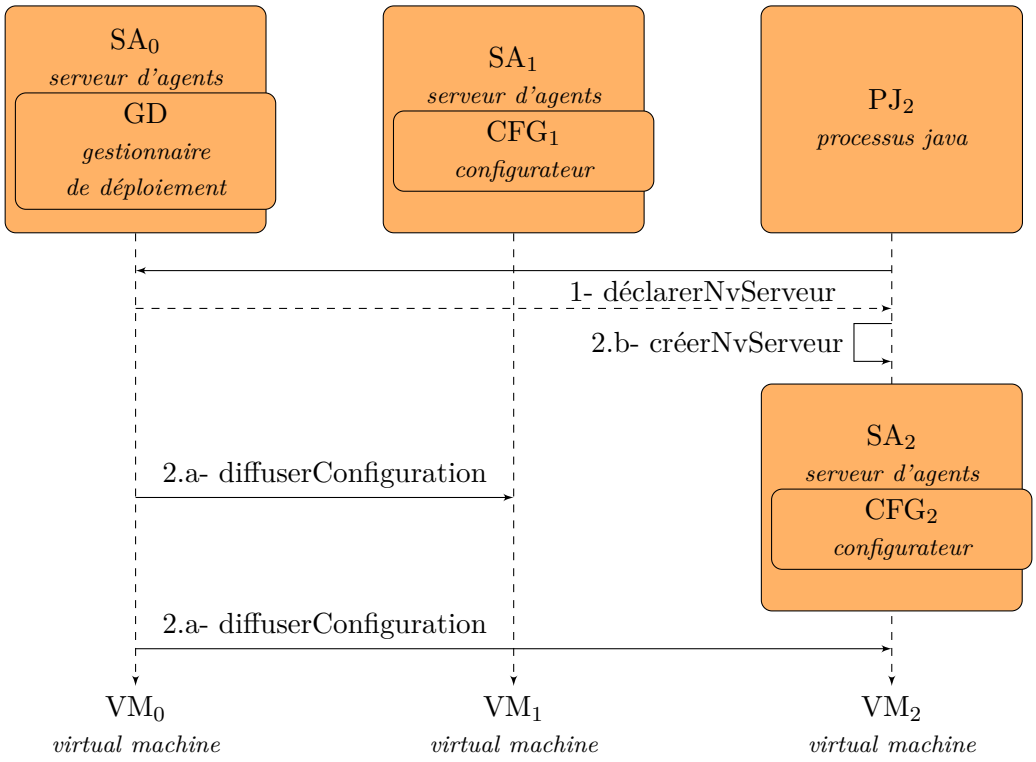

FIGURE 5.1 – Diagramme de séquence d'ajout d'un serveur d'agents dans le bus à messages

Dans cet exemple, le configurateur CFG_2 instancié dans le serveur d'agents SA_2 est supposé vouloir rejoindre le bus à messages constitué du serveur d'agents (SA_0) dans lequel est instancié le gestionnaire de déploiement (GD) ainsi que d'un autre serveur d'agents (SA_1), dans lequel est instancié un configurateur (CFG_1), arrivé dans le bus avant SA_2. Dans ce diagramme, seul le processus Java du nouvel entrant (PJ_2) est représenté, dans la mesure où celui qui exécute SA_0 et SA_1 n'est pas directement impliqué dans les échanges. Chaque processus Java a essentiellement un rôle d'amorçage (*bootstrap*). En outre, chaque serveur d'agents et l'agent qu'il crée en son sein sont regroupés dans une même boîte afin d'indiquer que, dans le cas de VAMP, leur existence est liée de leur création jusqu'à leur arrêt.

Dans un premier temps (étape 1), le processus Java accède, selon un protocole synchrone propriétaire, au service d'administration du bus à messages, exposé par SA_0. S'il ne parvient pas à le contacter dans un délai borné, il considère que son interlocuteur est défaillant et il procède à l'arrêt de la machine virtuelle sur laquelle il se trouve. Couplée au mécanisme de fiabilisation (cf. section 6.2), cela garantit la suppression de machines virtuelles applicatives orphelines, comparable aux threads zombis dans les systèmes d'exploitation. Il demande alors l'intégration d'un nouveau serveur d'agents dans le bus. Pour ce faire il invoque l'opération *declarerNvServeur* en indiquant son nom logique, l'adresse IP de l'hôte qui l'héberge ainsi que le port TCP utilisé pour échanger avec les autres serveurs d'agents du bus. En retour de cette opération, le processus Java obtient un identifiant $Id(SA_2)$ permettant de désigner, de façon unique, le nouveau serveur d'agents dans le bus. L'unicité des identifiants alloués est assurée par le fait que le service d'administration du bus n'est disponible que sur une instance de serveur d'agents (SA_0) et que le service traite la concurrence d'accès entre les requêtes d'ajout de serveur. Ainsi elles sont traitées une par une, de façon séquentielle.

Suite à l'ajout de l'identifiant $Id(SA_2)$ dans le bus, deux étapes sont exécutées en parallèle :

- étape 2.a : SA_0 et GD diffusent conjointement, à l'aide d'un message *diffuserConfiguration*, les éléments de configuration, mis à jour, dont ils disposent. Ainsi SA_0 transmet sa connaissance de la topologie du bus à tous les serveurs d'agents présents dans le bus à cet instant (i.e. SA_1 et SA_2). De façon comparable, GD transmet à l'ensemble des configurateurs présents dans le bus à cet instant (i.e. CFG_1 et CFG_2), sa représentation locale du modèle à l'exécution (cf. section 3.4) mis à jour des données de la nouvelle machine virtuelle. A réception d'une telle notification, un serveur d'agents met à jour sa connaissance interne. Simultanément, le configurateur qui lui est associé prend en compte la mise à jour du modèle à l'exécution. Il est désormais capable de communiquer, à l'aide de notifications, avec le configurateur nouvellement arrivé ($CFG2$).

- étape 2.b : le processus Java reçoit l'identifiant permettant de désigner de manière unique le serveur d'agents à créer. Il procède alors à son instanciation (*créerNvServeur*) à l'aide de cet identifiant (i.e. le nom logique). Il crée alors le configurateur CFG_2 dans le serveur ainsi créé. SA_2 reçoit alors la notification de diffusion de

la topologie courante du bus émise par SA_0 au cours de l'étape 2.a. CFG_2 reçoit quant à lui les valeurs actualisées de la table des machines virtuelles.

Au terme de l'étape 2.a vis-à-vis d'un configurateur, ce dernier est en mesure d'émettre des notifications à destination du nouvel entrant (CFG_2). Si ce dernier n'est pas encore instancié, c'est-à-dire si l'étape 2.b n'est pas entièrement terminée, le bus garantit la délivrance des notifications en cours d'envoi.

5.3 Protocole d'auto-configuration et d'auto-activation

Le déploiement d'une application dans le nuage consiste, dans un premier temps, à générer les images virtuelles sur lesquelles elle doit être distribuée puis à instancier ces images sous forme de machines virtuelles applicatives. A partir de la description applicative (ou description de l'application) fournie par l'utilisateur selon le formalisme défini en section 3.3.1.2, le gestionnaire de déploiement réalise ces deux premières étapes (cf. section 3.4.2.3). Dès lors, une fois initialisée, chaque machine virtuelle applicative démarre automatiquement le configurateur qu'elle embarque. Le rôle des configurateurs est de finaliser le déploiement de l'application comme décrit dans [65]. Ceci passe par la mise en œuvre d'un protocole d'auto-configuration et d'auto-activation des *wrappers* présents dans les machines virtuelles. Cette section décrit donc ce protocole (section 5.3.1) puis présente la vérification dont il a fait l'objet 5.3.2.

5.3.1 Description du protocole d'auto-configuration

Les configurateurs sont de simples objets Java qui s'exécutent en parallèle et qui s'appuient sur un modèle d'implémentation de type événement/réaction. Plus précisément, un configurateur est un agent qui réagit à des événements émis par d'autres configurateurs et matérialisés par des messages envoyés au travers du bus à messages (cf. section 5.1). Grâce aux propriétés d'asynchronisme, de distribution et de fiabilité de cet intergiciel, chaque configurateur s'exécute indépendamment des autres. Ceci rend le protocole d'auto-configuration et d'auto-activation indépendant de toute contrainte temporelle. La configuration globale de l'application évolue donc en fonction de la progression de l'exécution de chaque configurateur.

La séquence d'exécution d'un configurateur s'appuie sur un enchaînement précis, décrit par la figure 5.2. Dans cette figure, chaque action (e.g. `CREATEVM`, `CREATEWRP`, etc.) est représentée sous la forme d'un rectangle identifié par un entier naturel (❶, ❷, etc.) et une prise de décision est matérialisée par un losange. Une prise de décision correspond à un ensemble de choix possibles. Chaque choix identifie une action qui peut être atteinte depuis ce point de prise de décision.

Une fois démarré (❶), le configurateur crée les *wrappers* locaux[4] (❷) et les configure (❸), en fonction de la description applicative reçue du gestionnaire de déploiement. Cette

[4]Un *wrapper* local, désigne un *wrapper* qui est colocalisé sur la même machine virtuelle que le configurateur considéré. Il est donc sous la responsabilité, en termes d'administration, de celui-ci.

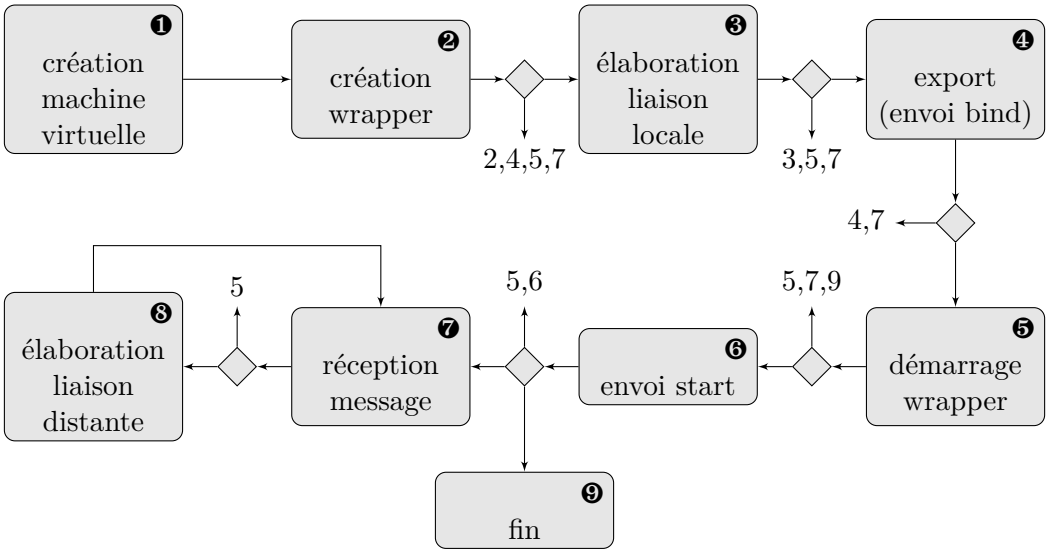

FIGURE 5.2 – Vue abstraite du flux d'exécution interne d'un configurateur

étape de configuration locale consiste notamment à établir les liaisons locales, c'est-à-dire celles qui associent deux *wrappers* locaux.

Ensuite, le configurateur participe à la configuration globale de l'application en inférant les liaisons distantes. Une liaison est qualifiée de distante, lorsqu'elle interconnecte l'interface cliente et l'interface serveur de deux *wrappers* qui ne sont pas colocalisés sur la même machine virtuelle. L'établissement d'une liaison distante nécessite que les configurateurs, responsables des *wrappers* à chacune de ses extrémités, coopèrent. Ainsi, pour chaque liaison distante dont la partie serveur correspond à une interface d'un *wrapper* local, le configurateur exporte cette interface à destination du configurateur en charge de la partie cliente de la liaison (❹). Il utilise pour cela un *message bind*.

Dans ce processus, chaque prise de décision propose un ensemble de choix dans le flot d'exécution, en fonction de l'architecture applicative considérée. Ainsi, par exemple, une machine virtuelle n'embarque pas nécessairement de liaison locale, et dans ce cas le configurateur saute directement à l'étape suivante.

Ensuite, le configurateur active les *wrappers* locaux qui peuvent être démarrés (❺), c'est-à-dire ceux dont chaque interfaces cliente avec une contingence obligatoire est liée à une interface serveur d'un *wrapper* déjà démarré. A cet instant dans l'exécution du protocole, seuls les *wrappers* qui ne possèdent aucune interface cliente obligatoire ou ceux dont toutes les interfaces clientes obligatoires sont connectées à des *wrappers* locaux, peuvent être démarrés. Pour chaque *wrapper* démarré, qui participe à une liaison distante en tant que partie serveur, le configurateur envoie un *message start* au configurateur responsable de la partie cliente de la liaison (❻). La fonction de ce message est d'informer le *wrapper* distant que le *wrapper* local a démarré.

Dès lors, le configurateur est susceptible de recevoir deux types de messages transmis

par le bus à messages et provenant d'un autre configurateur (❼) :

- un message bind, qui encapsule une interface serveur exportée, celle-ci participant à une liaison distante. A sa réception, le configurateur invoque l'opération bind sur le *wrapper* dont l'interface cliente constitue la partie cliente de la liaison distante (❽).

- un message start, qui indique qu'un *wrapper* distant a démarré. A sa réception, le configurateur conserve une trace de cette information et retourne à l'action ❺, pour déterminer si certains *wrappers* locaux peuvent être démarrés (ceux dont toutes les interfaces clientes obligatoires sont connectées et à des interfaces serveurs de *wrappers* déjà démarrés) ou non.

Note : lorsqu'un configurateur envoie un message de bind ou de start à destination d'un autre configurateur (❹ ❺), il le conserve également dans un journal, afin de pouvoir le réémettre en cas de recouvrement de l'état d'une machine virtuelle défaillante (cf. chapitre 6).

La figure 5.3 vise à illustrer, au travers d'un exemple, la manière dont les configurateurs interagissent pour déployer une application. Elle comprend, dans sa partie gauche, une vue architecturale de l'application à déployer. Il s'agit d'une instance de l'application fil rouge Springoo modélisée à l'aide de cinq *wrappers* interconnectés, distribués sur trois machines virtuelles. La partie droite de la figure détaille l'exécution du protocole d'auto-configuration qui en résulte. Seuls les messages échangés entre des configurateurs distincts sont représentés sur ce diagramme de séquence. Les communications locales (i.e. celles qui ne concernent qu'un seul configurateur) ne sont pas schématisées.

Une fois démarré, chaque configurateur met en œuvre les étapes de la séquence d'exécution détaillée ci-dessous :

1. Instanciation des *wrappers* locaux : en parallèle et de manière asynchrone, Cfg1 crée http-wrp, Cfg2 instancie jee-wrp, rar-wrp et ear-wrp, et Cfg3 crée db-wrp.

2. Etablissement des liaisons locales : jee-wrp, rar-wrp et ear-wrp étant interconnectés et colocalisés sur la même machine virtuelle, Cfg2 configure les liaisons locales correspondantes ;

3. Export des interfaces serveurs participant à une liaison distante : Cfg3 exporte à destination de Cfg2 l'interface serveur db-wrp. En effet, conformément à la description de l'application, rar-wrp (administré par Cfg2) possède une interface cliente qui doit être liée à l'interface serveur de db-wrp (administré par Cfg3). De façon identique, l'interface cliente de http-wrp et l'interface serveur de jee-wrp étant associées au travers d'une liaison distante commune, Cfg2 exporte en direction de Cfg1 l'interface serveur de jee-wrp.

4. Démarrage des *wrappers* : dès lors, Cfg3 a exporté l'ensemble des interfaces serveur dont il a la responsabilité. Etant donné que db-wrp ne possède aucune interface cliente, il peut être activé. Cfg3 envoie ensuite le message start correspondant

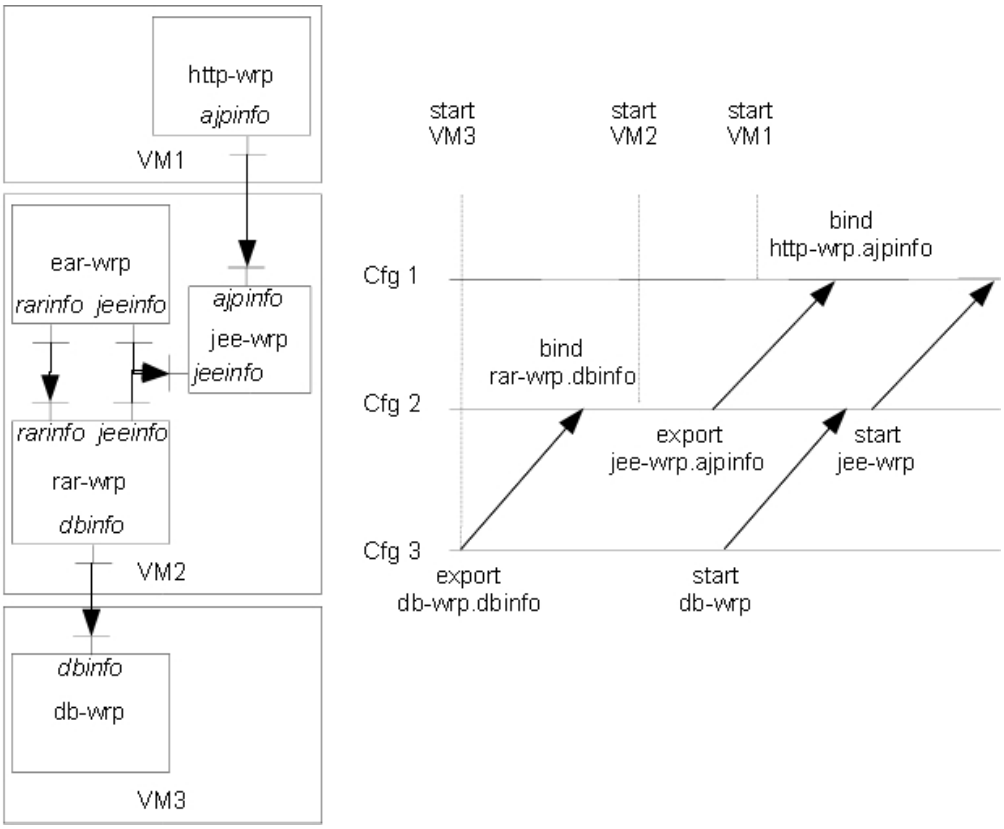

FIGURE 5.3 – Illustration de l'exécution du protocole d'auto-configuration sur un exemple simple

à rar-wrp (par le biais de Cfg2) dont une des interfaces clientes est associée à l'interface serveur de db-wrp (au sein d'une liaison distante). De manière similaire, Cfg2 démarre jee-wrp et envoie un message start à http-wrp (par l'intermédiaire de Cfg1). Il émet également un message start à l'attention de rar-wrp et de ear-wrp.

A chaque fois qu'un configurateur reçoit un message bind (respectivement d'un message start), il procède à la configuration de la liaison distante (respectivement à l'activation du *wrapper*) correspondant. Comme le montre le diagramme de séquence, les configurateurs communiquent de façon asynchrone. Ainsi, un configurateur peut émettre des informations à destination d'autres configurateurs sans disposer de connaissance de leur état. Même si certains de ces configurateurs ne sont pas disponibles (e.g. ils n'ont pas encore été instanciés), le modèle de communication asynchrone proposé par le bus à messages conserve les messages jusqu'à ce qu'ils puissent être remis à leur destinataire.

5.3.2 Vérification

La vérification formelle de la correction du protocole d'auto-configuration et d'auto-activation dans son intégralité a donné lieu à une collaboration avec l'équipe CONVECS de l'INRIA Rhône-Alpes. Celle-ci, dont un premier retour d'expérience constitue le sujet d'un article de revue en cours d'évaluation, porte sur :

1. la vérification du protocole tel qu'il est présenté dans ce chapitre. Ces travaux ont permis de mettre en évidence un dysfonctionnement au sein du protocole dans la gestion du démarrage de composants applicatifs interdépendants, colocalisés au sein d'une même machine virtuelle. Les spécifications et l'implémentation du protocole ont alors été immédiatement corrigées. Cette première phase de vérification a fait l'objet d'une publication dans une conférence [108] et d'un chapitre d'ouvrage [109], à paraître ;

2. la vérification de la version fiabilisée du protocole. Il s'agit de travaux en cours ;

3. la vérification des mécanismes de fiabilisation de VAMP lui-même. Il s'agit de travaux futurs.

La méthode de vérification formelle employée par l'équipe CONVECS pour valider un protocole de communication, consiste à le modéliser au moyen d'un graphe puis à effectuer un parcours exhaustif en s'assurant du respect d'un ensemble de contraintes quelque soit le chemin emprunté. L'ensemble des sommets du graphe correspondent aux états successifs du système. Un arc reliant deux sommets définit une transition entre deux de ces états, résultant de l'échange de message entre deux entités composant le système. Ainsi, la modélisation du protocole à vérifier consiste à définir :

le système étudié au moyen d'un ensemble d'entités fortement typées. Chacune d'elles regroupe un certain nombre d'attributs. L'état du système correspond à la valeur de l'ensemble des attributs de ses entités. Dans le cas de VAMP, les entités manipulées sont des composants, des interfaces, des liaisons, des configurateurs, des machines virtuelles, des tampons simulant le comportement du bus à messages.

les messages utilisés par les entités pour communiquer entre elles. Plus précisément, chaque entité définit le comportement qu'elle met en œuvre en cas de réception d'un message. Un tel comportement peut intégrer ou non l'émission d'un nouveau message. La définition de ces comportements est assurée au travers de processus.

les invariants qui correspondent à des contrôleurs de cohérence et dont la finalité est de formaliser les règles d'exécution du protocole. Ainsi, l'absence de cycles constitués exclusivement de liaisons obligatoires ou l'ordonnancement des messages à l'intérieur du bus constituent des exemples d'invariants dans le cas de VAMP.

L'ensemble de ces éléments de modélisation sont formalisés à l'aide du langage LOTOS NT. La compilation d'un descripteur au format LOTOS NT permet d'obtenir le graphe (*Labeled Transitions System* ou *LTS* en anglais) qu'il définit. Outre ces éléments, CADP [71] définit également un ensemble de propriétés qui peuvent être vues comme des contraintes non fonctionnelles, communes à l'ensemble des graphes. Un exemple de propriété est l'absence d'état puits (i.e. *deadlock*) dans le graphe.

Par la suite, l'environnement CADP met en œuvre un certain nombre d'outils visant à s'assurer que, quelque soit la manière de parcourir le *LTS* depuis son état initial jusqu'à son état final, les invariants soient maintenus et les propriétés vérifiées. Il s'agit donc d'un système d'analyse de graphe.

5.4 Conclusion

Ce chapitre a présenté le protocole dédié à l'automatisation des phases de postconfiguration et d'activation d'une application déployée à l'aide de VAMP. Ce protocole est réparti au sein des machines virtuelles applicatives qui participent à l'application. Il se décompose en deux phases :

- la première correspond à l'instanciation et à la configuration d'un bus à messages asynchrone et fiable entre les entités d'administration VAMP en charge du déploiement de l'application (i.e. le gestionnaire de déploiement et les différents configurateurs).
- la seconde correspond à l'automatisation des étapes d'instanciation, de configuration locale et distante ainsi qu'à l'activation des *wrappers* au sein de chaque machine virtuelle applicative.

Ce protocole est caractérisé par l'absence de contraintes temporelles entre les configurateurs qui participent à sa mise en œuvre. Cette propriété découle de l'asynchronisme et de la fiabilité des échanges assurés par le bus à messages associé à l'exécution en parallèle de chaque machine virtuelle et du configurateur associé.

Chapitre 6

Fiabilisation du déploiement

Sommaire

6.1 Contexte de la tolérance aux pannes 125
6.1.1 Détection de la panne 126
6.1.2 Résolution de la panne 127
6.2 Fiabilisation du déploiement des machines virtuelles applicatives 130
6.2.1 Détection de pannes 130
6.2.2 Fiabilisation par recouvrement 131
6.3 Fiabilisation de VAMP 134
6.3.1 Fiabilisation du portail 134
6.3.2 Fiabilisation du gestionnaire d'application 135
6.4 Conclusion 142

Dans le cadre de son déploiement ou de son exécution, une application logicielle peut être confrontée à des défaillances. Conformément aux définitions relatives à la fiabilité [87] [95] [21] [86] [130], une défaillance correspond à un comportement de l'application non conforme à ses spécifications. Elle résulte d'une erreur, qui a préalablement conduit l'application dans un état erroné.

A l'instar des causes à l'origine des erreurs [22], il existe de nombreuses classifications de ces défaillances. La plupart d'entre elles s'appuient notamment sur les critères suivants :

Localité : une défaillance peut être *interne* ou *externe*. Une défaillance sera dite *interne* si elle n'altère que les fonctionnalités de l'application sans avoir d'impact sur l'environnement d'exécution. Sa détection et sa correction requiert une connaissance approfondie de l'application. Cela induit un couplage fort entre le système de fiabilisation capable de traiter ce type de défaillance et l'application. Inversement, une défaillance sera dite *externe* si elle impacte l'environnement d'exécution

de l'application. Dans ce cas, elle peut résulter indifféremment d'une faute applicative (e.g. un débordement mémoire au sein d'une application qui entraîne une réinitialisation d'une machine dans son ensemble) ou d'une erreur indépendante de l'application (e.g. erreur matérielle qui entraîne l'interruption des communications réseau, erreur malveillante intentionnelle). Qu'il s'agisse d'une cause applicative ou non, une défaillance externe peut être détectée par un système de fiabilisation indépendant de l'application considérée.

Classe : une défaillance pourra être *franche* ou *transitoire*. Confronté à une défaillance *franche*, un système dysfonctionnera tant qu'aucune correction ne sera mise en œuvre. En présence d'une défaillance *transitoire* ou *temporaire*, le système oscillera entre un état de fonctionnement conforme à ses spécifications et un état de dysfonctionnement dont la durée sera limitée dans le temps.

L'un des objectifs de la plate-forme VAMP, qui constitue l'une des contributions majeures de ce doctorat, est de proposer un mécanisme de déploiement fiable d'applications arbitraires, virtualisables et réparties dans le nuage. Il s'agit de garantir que l'étape de déploiement sera menée à son terme, en dépit des pannes pouvant survenir à tout instant, dans la mesure où leur nombre demeure fini. Cet objectif passe par :

- la fiabilisation des machines virtuelles sur lesquelles doit être déployée l'application et des configurateurs qu'elles embarquent ;
- la fiabilisation des entités qui constituent le système de déploiement lui-même (i.e. le portail et le gestionnaire d'application) ;
- la fiabilisation des communications entre les entités constituant le système VAMP.

La fiabilisation portant sur la phase de déploiement d'une application, celle-ci ne possède pas encore d'état. En d'autres termes, jusqu'à ce qu'elle soit effectivement activée (i.e. instant qui correspond à la fin du déploiement), elle n'a encore stocké aucune information relative à la session d'un utilisateur.

Aucune supposition n'étant faite quant à l'architecture, aux technologies, aux domaines métiers ou fonctionnels relatifs aux applications déployées, VAMP ne cherche qu'à s'affranchir des défaillances franches externes et plus précisément de celles détaillées ci-après :

PANNE_01 : des défaillances franches de machine virtuelle ou physique, c'est-à-dire celles qui rendent la machine virtuelle définitivement inutilisable ;

PANNE_02 : des défaillances réseau, c'est-à-dire celles qui entraînent une indisponibilité temporaire d'une partie des ressources réseau, rendant ainsi l'accès, à une ou plusieurs machines virtuelles, impossible ;

PANNE_03 : des défaillances du système d'administration lui-même. Dans ce cas, il peut s'agir de pannes externes (et couvertes par les cas PANNE_01 et PANNE_02) ou de pannes internes au système VAMP.

L'approche adoptée fait l'hypothèse que chaque machine virtuelle applicative ne dispose pas d'espace de stockage persistant. Ainsi, à l'inverse d'une machine physique dont les disques conservent leur contenu au-delà de l'arrêt de la machine, les disques d'une machine virtuelle sont eux-mêmes considérés comme virtuels et ils n'existent que si la machine virtuelle est démarrée : en cas d'arrêt de la machine virtuelle, de manière choisie ou suite à une défaillance, l'ensemble des données qu'ils contiennent sont perdues.

Le but de ce chapitre est de présenter les techniques de fiabilisation utilisées dans VAMP et qui font l'objet de collaboration en cours avec l'équipe CONVECS de l'INRIA en vue de leur vérification formelle. Il comprend ainsi la section 6.1 consacrée aux grands principes de fiabilisation d'un système distribué. La section 6.2 définit ensuit la manière de fiabiliser les machines virtuelles applicatives ainsi que les configurateurs qu'elles embarquent, afin de proposer un protocole de déploiement tolérant aux pannes. Par la suite, la section 6.3 traite de la fiabilisation du système VAMP lui-même, c'est-à-dire du portail et des gestionnaires d'application. Enfin, la section 6.4 propose une synthèse du chapitre.

6.1 Contexte de la tolérance aux pannes

La fiabilisation d'un système quelconque est le processus qui consiste à le prémunir des pannes (ou défaillances) auxquelles il est confronté. Pour cela, il peut adopter une stratégie d'évitement ou de correction des pannes :

- Les approches par évitement consistent à détecter une faute potentielle ou une erreur latente avant qu'elle ne débouche sur une panne du système. Les techniques de détection utilisées s'appuient sur la validation (détection de causes potentielles de fautes et d'erreurs), la prévention (détection de fautes et d'erreurs avant qu'elles ne surviennent) ou l'estimation (évaluation du risque de survenue d'une faute ou d'une erreur).

- Les approches correctives ou de tolérance aux pannes visent à permettre au système de fonctionner malgré la présence de pannes. Ainsi, suite à la détection d'un état erroné du système :

  - la technique par recouvrement consiste à le ramener dans un état correct, soit par retour à un état antérieur sauvegardé, soit par détermination d'un nouvel état valide ;
  - la technique par compensation vise à introduire suffisamment de redondance dans le système pour qu'il puisse continuer à se conformer à ses spécifications malgré la survenue d'un état erroné ;

Lorsque le système considéré est une application distribuée, sa fiabilisation globale consiste à fiabiliser chacun des processus et les canaux de communication qui la composent.

Etant donné que la stratégie d'évitement des pannes est une approche fortement couplée à l'application considérée et qu'elle ne permet pas de garantir qu'aucune erreur

ne débouche finalement sur une défaillance lors de l'exécution du système, la suite de ce chapitre se focalisera uniquement sur les approches de tolérance aux pannes.

Une approche de type tolérance aux pannes se décompose en deux phases. La première consiste à détecter les défaillances. La seconde vise à y remédier.

6.1.1 Détection de la panne

La détection d'une panne constitue une première étape essentielle dans un mécanisme de fiabilisation basée sur une approche corrective. En effet, sans elle, la correction nécessaire ne serait jamais mise en œuvre. Comme indiqué dans [121], cette détection dans le contexte des approches par recouvrement, consiste à savoir déterminer qu'un état du système est erroné. La fonction de détection est implémentée au sein d'un *détecteur de pannes* (abrégé en *détecteur*). Ce détecteur peut avoir une architecture centralisée relativement simple ou se répartir au sein de plusieurs éléments logiciels. Les détecteurs de pannes se répartissent en deux catégories, en fonction de leur dépendance au temps :

les détecteurs synchrones reposent sur la réception, dans un laps de temps borné, appelé délai de garde, d'un événement les informant de la bonne santé du système à fiabiliser. Dans le cadre d'un détecteur de type *ping*, cet événement correspond à une réponse du système suite à une sollicitation du détecteur, alors que dans le cas d'un détecteur de type *heart beat*, le système à fiabiliser émet l'événement de façon régulière[1]. Si le délai de garde expire avant que le détecteur n'ait reçu l'événement, celui-ci conclut à une défaillance du système.

les détecteurs asynchrones sont indépendants d'aspects temporels. Leur principe de fonctionnement consiste à tenter de déterminer si l'état du système à fiabiliser demeure dans un espace d'états valides. Ainsi, dans le cadre d'un détecteur de type *pinpoint* [39], cette détermination consiste à observer l'ensemble des communications échangées entre les entités qui composent le système. Le résultat de ces observations est régulièrement confronté, selon un test statistique (e.g. test du χ^2), au comportement théorique normal du système. En fonction du résultat du test, une défaillance du système est détectée ou non.

Le principal intérêt des approches synchrones est de détecter les défaillances dans un temps borné (i.e. le délai de garde). Néanmoins, elles ne permettent pas de discriminer l'origine de la défaillance détectée (e.g. panne du système à fiabiliser, panne d'un canal de communication entre le détecteur et le système) ni son type (e.g. panne franche, transitoire ou intermittente). En outre, la valeur du délai de garde doit être estimée à priori. Or, un délai de garde trop court provoquera la détection de faux positifs (i.e. cas de défaillances qui n'en sont pas) alors qu'à l'inverse, un délai de garde trop long rendra la détection moins rapide donc moins efficace. A l'inverse, les approches asynchrones parviennent à mieux discriminer la nature et l'origine des pannes. Cependant, comme le

[1] Le nom de ce type de détecteur provient de la régularité de l'émission de l'événement, caractérisant l'état de fonctionnement du système. Ces événements sont semblables à des battements (ou des pulsations) cardiaques monitorés par le détecteur.

souligne [69], à l'image de pinpoint, elles reposent sur des comportements statistiques et ne proposent donc pas une détection exacte.

6.1.2 Résolution de la panne

Les techniques de résolution des défaillances dans le contexte de la tolérance aux pannes se subdivisent en deux patrons :

Approche par compensation : notamment étudiée dans [38], [26] et [107], elle vise à se prémunir des pannes par introduction de redondance. Le système à fiabiliser est donc dupliqué en plusieurs instances (ou duplicats) iso-fonctionnelles. Cette approche permet de résister à un certain nombre de pannes sans pour autant nécessiter de réparer les duplicats défaillants, c'est-à-dire d'instancier de nouveaux duplicats pour maintenir constant le nombre de duplicats valides[2]. En effet, un duplicat défaillant est automatiquement remplacé par un duplicat valide préexistant. Cela nécessite de maintenir la cohérence entre les états de chaque duplicat, afin que le système soit en mesure de délivrer un service conforme à ses spécifications. Cette mise en cohérence de l'état des duplicats s'appuie sur un mode de communication par diffusion. Ce mécanisme de diffusion doit cependant offrir des propriétés de fiabilité (i.e. pas de perte de message), d'atomicité (i.e. un message adressé à un ensemble de destinataires est délivré à tous ou à aucun) et d'ordonnancement dans la délivrance des messages (i.e. deux messages émis successivement par une même source vers un même destinataire sont reçus dans le même ordre). Il existe essentiellement deux modes de duplication de l'état du système :

duplication passive : une instance du système à fiabiliser est choisie parmi les duplicats pour jouer le rôle de *duplicat principal*. Les duplicats restants sont qualifiés de secondaires. Ce choix s'effectue au moyen d'un protocole d'élection déterministe connu de tous les duplicats. Le rôle du duplicat principal est double : d'une part il délivre le service offert par le système à fiabiliser, d'autre part, il assure la retransmission (par diffusion) de son état interne à l'ensemble des duplicats secondaires. Lorsque le duplicat principal est confronté à une défaillance, l'un des duplicats secondaires est désigné pour le remplacer (cf. figure 6.1).

duplication active : elle s'appuie sur le postulat que le comportement du système à fiabiliser est déterministe (i.e. les mêmes causes produisent les mêmes effets). A l'inverse de la duplication passive, il n'existe aucune distinction entre les duplicats. Chacun d'eux est soumis aux mêmes flux d'entrées/sorties vis-à-vis des entités externes. Le maintien de la cohérence d'état entre duplicats est garanti par les propriétés de fiabilité, d'ordonnancement et d'atomicité de la diffusion. En cas de panne d'un duplicat, les autres continuent à fonctionner

[2] A partir d'une estimation de la probabilité de survenue d'une panne, il est possible de déterminer le nombre de duplicats nécessaires pour offrir le niveau de disponibilité requis pour le système à fiabiliser.

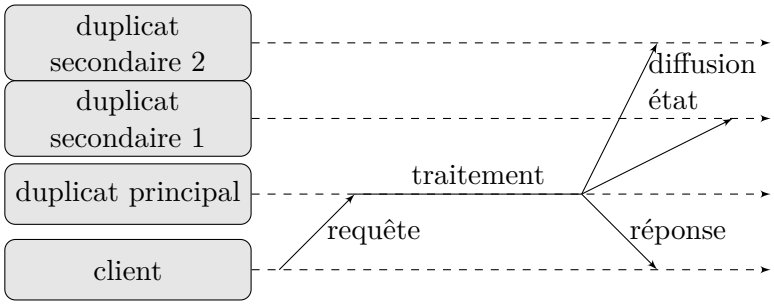

FIGURE 6.1 – Principe de duplication passive

normalement (cf. figure 6.2). La duplication active a fait l'objet de nombreux travaux [85] [112] [46] [26].

Le principal intérêt de la duplication active est sa forte réactivité. En effet, dans le cadre d'un échange entre le système à fiabiliser et un client externe, celui-ci envoie la même requête à chacun des duplicats et reçoit, en retour, autant de copies de la réponse qu'il y a de duplicats valides. Il ne traite alors que la première des réponses reçues. Cependant, cette technique nécessite la prise en compte, au niveau de chaque entité cliente du système à fiabiliser, de l'existence des multiples duplicats. En outre, l'exécution à l'identique de chaque duplicat induit un gaspillage important des ressources de calcul. A l'inverse, la duplication passive est moins gourmande en termes de traitements, dans la mesure où seul le duplicat principal réalise effectivement les traitements. Néanmoins, en cas de défaillance du duplicat principal, son remplacement peut être une opération coûteuse en temps.

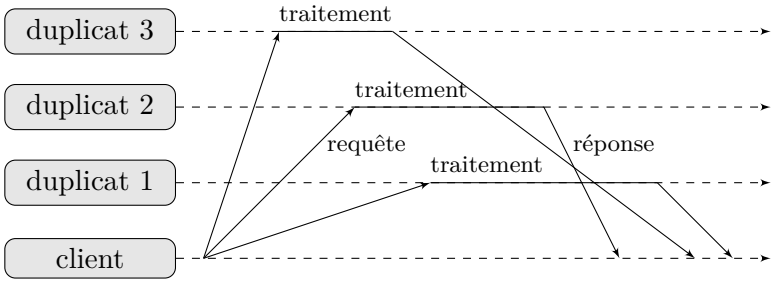

FIGURE 6.2 – Principe de duplication active

Afin de tirer partie des avantages de chacune de ces deux techniques de duplication, des propositions alliant duplication active et duplication passive ont été faites :

- Ainsi, la *duplication semi-active* [54] reprend les principes de la duplication passive à ceci prêt que le duplicat principal, appelé *meneur*, ne transmet pas son état aux duplicats secondaires, qualifiés de *suiveurs*, mais la requête

émise pas le client. Comme dans la duplication active, les suiveurs exécutent le même code que le meneur mais n'émettent pas de réponse à l'adresse du client (cf. figure 6.3).

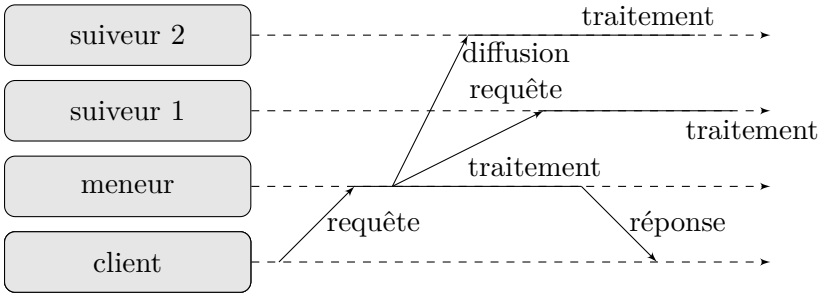

FIGURE 6.3 – Principe de duplication semi-active

- Dans la duplication coordinateur-cohorte [28], le client émet une requête à chacun des duplicats qui forment la *cohorte*, de manière similaire à la duplication active. Cependant, comme dans la duplication passive, seul un duplicat, appelé *coordinateur*, lui répond (cf. figure 6.4).

Approche par recouvrement : son objectif est de ramener le système d'un état erroné à un état valide. Une telle opération de restauration d'un état valide s'appuie généralement sur l'une des deux principales approches [61] :

point de reprise : cette technique consiste à sauvegarder régulièrement, de façon fiable et persistante, l'état du système, au cours de son exécution. A la suite d'une défaillance du système, sa restauration dans un état valide s'appuie sur le dernier point de reprise cohérent.

journalisation : basée sur l'hypothèse que le comportement du système est déterministe, cette technique consiste à enregistrer sur un support fiable et persistant l'ensemble des événements à l'origine d'un changement d'état du système.

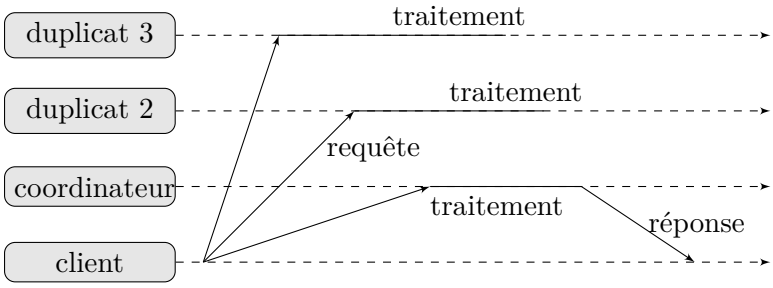

FIGURE 6.4 – Principe de duplication coordinateur-cohorte

A la suite d'une défaillance, l'opération de restauration consiste à repartir d'une instance de système dans l'état initial et à ré-exécuter l'ensemble des événements journalisés afin de revenir dans le dernier état cohérent avant la panne.

Ces deux techniques s'appuient sur de la redondance d'état, nécessitant ainsi la mise en œuvre d'un support de sauvegarde persistant et fiable. L'approche par point de reprise présente l'intérêt d'un surcoût faible lors d'une exécution nominale. Néanmoins, la restauration nécessite de ramener le système dans le dernier état cohérent, c'est-à-dire un état valide simultanément pour l'ensemble des constituants du système. L'opération peut s'avérer complexe et coûteuse. En outre elle peut nécessiter une reconfiguration à chaud du système et de son environnement d'exécution (e.g. gestion de l'allocation dynamique d'adresse IP dans le nuage). A l'inverse, la journalisation prend plus aisément en compte la reconfiguration à chaud. Cependant les opérations d'enregistrement dans le journal sont nombreuses et la restauration demeure une opération longue.

6.2 Fiabilisation du déploiement des machines virtuelles applicatives

La finalité de VAMP est de rendre autonome l'ensemble du processus de déploiement d'une application arbitraire dans le nuage, sans recourir à des interventions ou des aides extérieures (e.g. administrateur, utilisateur). Ceci nécessite donc de rendre le protocole d'autoconfiguration et d'autoactivation de VAMP tolérant aux pannes : il est nécessaire de fiabiliser l'environnement d'exécution de l'application, c'est-à-dire les machines virtuelles sur lesquelles sont déployés l'ensemble des composants applicatifs ainsi que les éléments d'administration impliqués dans ce protocole (i.e. les configurateurs).

6.2.1 Détection de pannes

L'objectif n'étant pas tant de déterminer l'origine des défaillances que de les déceler le plus rapidement possible et avec une précision optimale (vis-à-vis des faux positifs), le mécanisme de détection de pannes choisi s'appuie sur une approche synchrone (cf. section 6.1.1). Plus précisément, il s'agit d'un détecteur de type *heart beat*. En effet, une panne pouvant survenir à n'importe quel moment, il est nécessaire que le mode de détection s'appuie sur l'envoi périodique d'une preuve de l'état de fonctionnement des éléments à fiabiliser. Or, dans ce contexte, un détecteur *ping* devra recourir à deux fois plus de messages entre le détecteur et l'élément à fiabiliser, qu'un détecteur de type *heart beat*.

Le système de détection proposé adopte une architecture en étoile. En effet, il est constitué d'un ensemble d'éléments à fiabiliser (les machines virtuelles) qui s'adressent à un détecteur centralisé unique :

- Au sein de chaque machine virtuelle applicative, un thread client est responsable d'émettre à intervalles réguliers la pulsation cardiaque. Il s'exécute dans la même machine virtuelle Java que le serveur d'agents sur lequel est déployé le configurateur. Son existence est liée à celle du serveur d'agents en ce sens que, lorsque le thread associé au serveur d'agents tombe en panne, la machine virtuelle Java s'arrête automatiquement, stoppant ainsi l'exécution du thread dédié au heart beat. Ainsi, une panne franche de la machine virtuelle (PANNE_01), une panne réseau (PANNE_02) ou une panne du configurateur qui aboutit à un arrêt du serveur d'agents (PANNE_03), entraînera l'interruption de l'émission de la pulsation cardiaque par le thread.

- Le détecteur de panne est un élément centralisé colocalisé dans la machine virtuelle Java du gestionnaire d'application. Lorsque le gestionnaire d'application procède à l'instanciation d'une nouvelle machine virtuelle applicative, il en notifie le détecteur. Celui-ci crée alors un thread dédié à recevoir les requêtes émises par le thread responsable de l'émission des pulsations cardiaques, sur la machine virtuelle. Parallèlement, il initialise un *timer*. La valeur initiale de celui-ci est une estimation de la durée maximale d'instanciation et d'initialisation de la machine virtuelle[3]. En d'autres termes, il s'agit du délai maximum nécessaire pour que le détecteur reçoive la première pulsation cardiaque émise depuis la machine virtuelle considérée. Lors de la réception d'une requête correspondant à un battement de cœur en provenance de la machine virtuelle, le timer est réinitialisé avec une valeur estimant le délai de garde (i.e. la durée maximum acceptable entre deux pulsations). Si aucune requête n'est reçue avant que le temps mesuré par le timer ne soit écoulé, le détecteur indique qu'une panne est survenue. Celle-ci peut correspondre aux pannes décrites plus haut (PANNE_01, PANNE_02 ou PANNE_03).

6.2.2 Fiabilisation par recouvrement

La technique de tolérance aux pannes implémentée pour fiabiliser le protocole d'autoconfiguration s'appuie sur un mécanisme par recouvrement d'état des machines virtuelles défectueuses (cf. section 6.1.2). Sa fonction de réparation est implémenté dans un *correcteur de pannes* (abrégé en *correcteur*) centralisé, au sein du gestionnaire de déploiement de l'application.

Lorsque le détecteur détermine qu'une panne est survenue au niveau d'une machine virtuelle applicative, il en notifie le correcteur. Celui-ci exécute séquentiellement les étapes représentées en figure 6.5, pour remplacer la machine virtuelle défectueuse par une en état de fonctionnement.

Retrait du serveur d'agents défectueux : dans un premier temps, son but est de

[3]L'estimation de cette valeur initiale ne fait pas partie de la contribution présentée dans ce manuscrit. Ainsi, VAMP délègue à l'utilisateur la détermination de cette valeur qui peut ensuite être renseignée au moyen d'une propriété de configuration. Il s'agit d'une phase préliminaire de calibration du comportement de l'application à déployer sur l'infrastructure choisie et dont l'automatisation constitue un sous-ensemble du domaine de recherche de l'auto-évaluation (*self-benchmaring* en anglais).

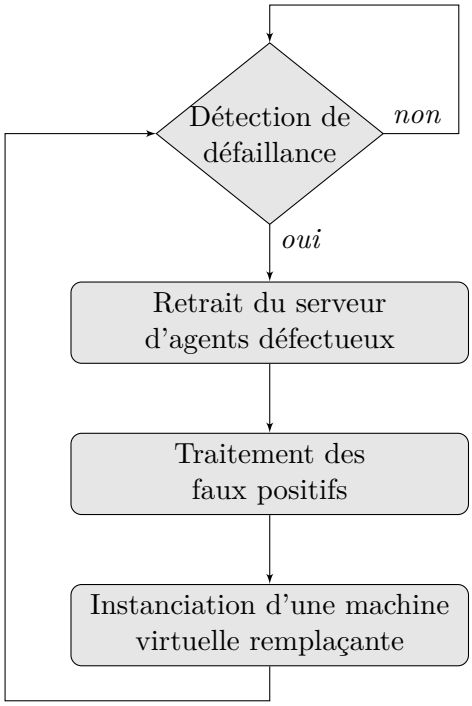

FIGURE 6.5 – Algorithme de remplacement d'une machine applicative défectueuse

mettre en cohérence la définition courante du bus à messages avec l'état effectif du système. En effet, la panne de la machine virtuelle entraîne le retrait du serveur d'agents qu'elle contenait et qui faisait partie du bus. Le correcteur s'adresse donc au serveur d'agents sur lequel est déployé le gestionnaire d'application pour qu'il mette à jour la configuration du bus à messages. Ceci consiste à :

- retirer du bus à messages, le serveur d'agents dont l'instance s'exécutait sur la machine virtuelle défectueuse ;
- purger les queues de messages entrants Q_{in} et sortants Q_{out} du serveur d'agents local (i.e. celui associé au gestionnaire d'application) en retirant les potentiels messages provenant / à destination du serveur d'agents défectueux. En effet, le bus étant asynchrone, certains messages émis ou à destination d'un agent de la machine virtuelle défaillante peuvent encore être en cours de transit dans le bus alors que la machine virtuelle est hors service. Il n'y a donc plus lieu de les traiter ;
- mettre à jour la représentation locale du modèle à l'exécution, par modification du statut de la machine virtuelle défaillante, afin qu'il soit en cohérence avec le système effectivement déployé.

Cette mise à jour de la configuration du bus à messages est automatiquement

diffusée à l'ensemble des serveurs d'agents survivants. Chacun d'eux procède à son tour au retrait du serveur d'agents défectueux et des notifications associées, éventuellement présentes dans leurs queues d'émission et de réception.

Traitement des faux positifs : dans la mesure où le mécanisme de détection n'est pas capable de faire la distinction entre les différents type de pannes détectées, il est possible qu'une machine virtuelle non défaillante soit déclarée en panne à tort. Ainsi pour les pannes de type PANNE_03, le détecteur ne reçoit plus de pulsation cardiaque du fait de la rupture de la communication réseau entre lui et la machine virtuelle incriminée. Afin de se prémunir de ce cas de faux positif, le correcteur s'adresse au gestionnaire d'approvisionnement (i.e. la plate-forme d'IaaS) pour qu'il arrête l'exécution de la machine virtuelle.

Instanciation d'une nouvelle machine virtuelle : à partir de la modélisation relative à la machine virtuelle, contenue dans la représentation locale du modèle à l'exécution, le gestionnaire d'application retrouve l'image qui a servi à instancier la machine virtuelle. Il notifie donc la plate-forme d'IaaS pour qu'il instancie une machine virtuelle de remplacement à partir de son image.

Dès lors, les protocoles d'insertion dynamique d'un nouveau serveur d'agents dans le bus à messages (cf. section 5.2.2) puis d'autoconfiguration et d'autoactivation (cf. section 5.3) sont exécutés au niveau de la machine virtuelle remplaçante, comme s'il s'agissait d'un déploiement initial.

Au cours de l'insertion dynamique du serveur d'agents de remplacement dans le bus, le gestionnaire d'application diffuse à l'ensemble des configurateurs déjà présents dans le bus, la mise à jour du modèle à l'exécution (étape 2.a). Celle-ci porte d'une part sur l'identifiant de la nouvelle machine virtuelle dans le IaaS et sur son adresse IP (dans le cas d'une allocation dynamique, via DHCP par exemple). Ainsi, l'identifiant de serveur d'agents de la machine virtuelle défaillante est réutilisé. Lorsqu'un configurateur reçoit le modèle mis à jour, il réémet l'ensemble des notifications d'export et d'activation qui ont pu être adressées à l'un des *wrappers* présents sur la machine virtuelle défectueuse. En effet, l'ensemble de ces notifications sont journalisées au niveau du configurateur, au moment de leur envoi.

D'autre part, toute nouvelle notification d'export à destination d'un *wrapper*, donne lieu à la reconfiguration, par écrasement de l'éventuelle configuration antérieure, de la liaison distante dont l'une de ses interfaces constitue la partie cliente. De même, lorsqu'un *wrapper* est destinataire d'une notification d'activation alors qu'il est déjà démarré, cela déclenche une opération de redémarrage de l'élément logiciel qu'il pilote (restart). Cette évolution dans le comportement des configurateurs est représentée par la figure 6.6 qui intègre la prise en compte du recouvrement d'une machine virtuelle défaillante. Cette représentation est à mettre en regard avec la figure 5.2.

Comme cela a déjà été mentionné en section 5.3.2, la version fiabilisée du protocole de déploiement des machines virtuelles applicatives proposé par VAMP fait l'objet de travaux en cours, dans le cadre d'une collaboration avec l'équipe CONVECS de l'INRIA-Rhône, en vue de le valider formellement.

6.3 Fiabilisation de VAMP

Outre le besoin de fiabiliser les machines virtuelles qui composent l'application et des configurateurs qu'elles embarquent, la fiabilisation du processus de déploiement autonome nécessite de fiabiliser les entités d'administration (i.e. le portail et les gestionnaires d'application) qui composent VAMP lui-même ainsi que les communications entre ces entités. La figure 6.7 illustre la manière dont la fiabilité est prise en compte au sein de ces éléments. Pour ce qui est du portail, il s'agit d'une approche par duplication (cf. section 6.3.1). Concernant les gestionnaires d'application, l'approche adoptée conjugue à la fois duplication (au sein d'une structure en anneau) et compensation, le besoin étant d'offrir une solution autonome, fiable et devant gérer un état (cf. section 6.3.2).

Enfin, la fiabilisation des échanges entre les entités d'administration s'appuie sur l'utilisation d'un bus à messages distribué, asynchrone et fiable garantissant la délivrance des messages entre ces entités (cf. section 5.1).

6.3.1 Fiabilisation du portail

La fiabilisation du portail adopte une approche par duplication basée sur un composant de répartition de charge placé en frontal d'un *cluster* d'instances du portail. Celles-ci disposent d'un état commun, partagé, synchronisé régulièrement. Le rôle du répartiteur de charge est d'assurer la haute disponibilité du portail en fonction des défaillances qui affectent les instances dans le *cluster*. Cette technique de fiabilisation est très répandue dans la mise en œuvre de capacités de haute disponibilité d'un portail web.

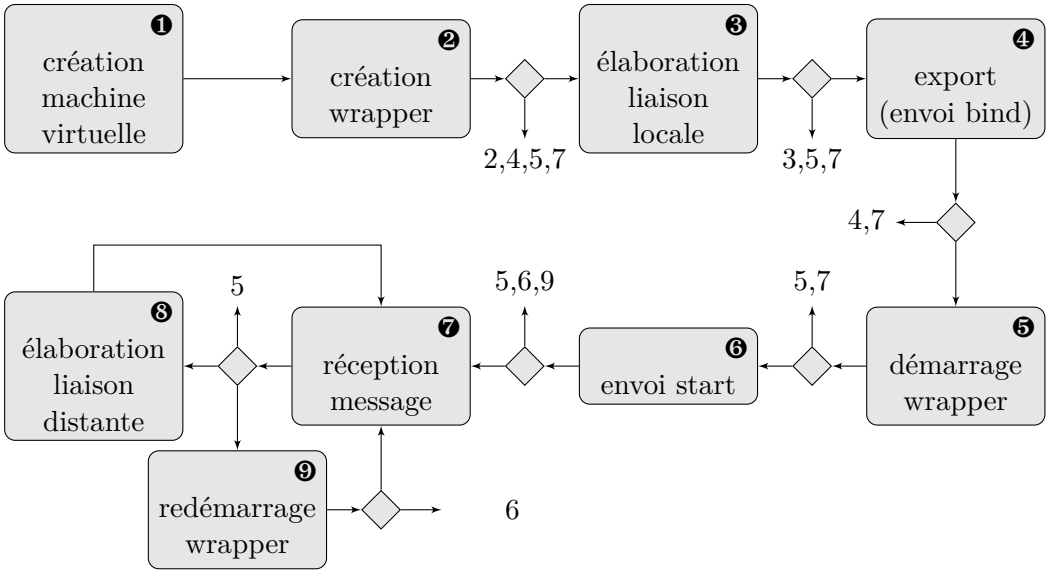

FIGURE 6.6 – Prise en compte de la fiabilisation des machines virtuelles applicatives dans le flux d'exécution interne d'un configurateur

Il est possible d'enrichir le comportement du répartiteur de charge en introduisant des règles complémentaires permettant d'améliorer le routage des requêtes. Ainsi, à l'aide d'un serveur de nom géographique, les requêtes peuvent être routées vers un sous-groupe d'instances de portail, afin de répondre à des exigences techniques (e.g. réduire les latences réseaux) ou fonctionnelles (e.g. localisation physique des données dans un pays particulier).

6.3.2 Fiabilisation du gestionnaire d'application

A l'image de la fiabilisation du protocole d'autoconfiguration – au travers des machines virtuelles applicatives et des configurateurs qu'elles embarquent –, l'approche utilisée pour garantir la fiabilité du gestionnaire d'application est également auto-réparatrice. Elle est donc capable de résister à un flux de pannes, dont les caractéristiques sont définies au travers des exigences utilisateur, en matière de qualité de service. En d'autres termes, VAMP ne dispose pas de mécanisme d'auto-calibration et il convient donc que l'utilisateur spécifie le nombre maximum de défaillances par unité de temps qu'un gestionnaire d'application doit savoir supporter. Ainsi, l'approche adoptée s'appuie sur un

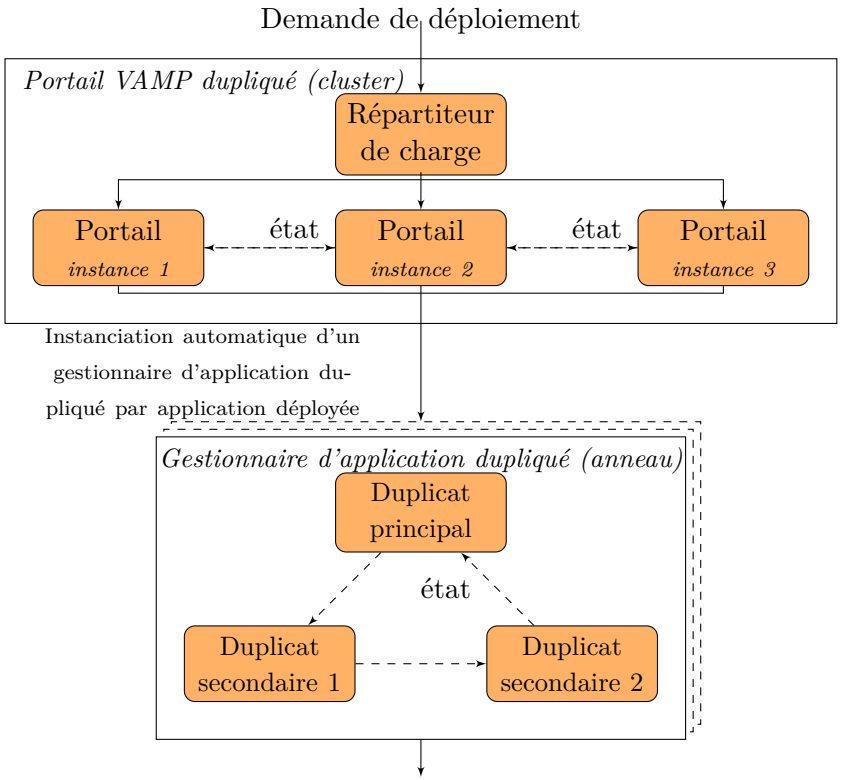

FIGURE 6.7 – Vue d'ensemble des techniques de fiabilisation des entités d'administration de VAMP

mécanisme par compensation couplé à du recouvrement d'état. La compensation vise à garantir la prise en compte de la défaillance du gestionnaire d'application, sans recourir à une couche d'administration supérieure, grâce à un ensemble de duplicats. Le recouvrement d'état permet de remplacer automatiquement les entités défectueuses.

6.3.2.1 Détecteur de pannes autoréparable

Comme dans le cas de la fiabilisation du protocole d'autoconfiguration, l'objectif n'est pas tant de déterminer la nature d'une défaillance impactant un duplicat, que de la détecter le plus efficacement possible en termes de latence et de pertinence de détection. Le choix d'un mode de détection de pannes s'est donc naturellement porté sur une approche synchrone. Là encore, il s'agit d'un détecteur de type *heart beat*. En effet, une panne pouvant survenir à n'importe quel moment, il est nécessaire que le mode de détection s'appuie sur l'envoi périodique d'une preuve de l'état de fonctionnement des éléments à fiabiliser.

Structure en anneau

Dans l'architecture hiérarchique proposée dans VAMP (cf. section 3.4), le gestionnaire d'application constitue le niveau d'administration le plus élevé, pour une application donnée. Afin de garantir son isolation et celle de l'application dont il a la charge, notamment en termes de sécurité et de performance vis-à-vis des autres gestionnaires d'application et des autres machines virtuelles applicatives, il ne doit pas s'appuyer sur une couche supérieure (i.e. le portail) pour assurer la détection des défaillances qui interviendraient à son niveau. De plus, une telle approche ne ferait que reporter le problème à la couche supérieure de sa propre fiabilisation. En outre, le gestionnaire d'application ne peut pas non plus faire appel à la couche inférieure, constituée des configurateurs, celle-ci ne disposant pas de la connaissance suffisante pour assurer sa fiabilité. Dès lors, la fiabilisation du gestionnaire d'application doit s'appuyer sur sa duplication en un ensemble d'entités iso-fonctionnelles, appelées *duplicats*. Le rôle de ces duplicats est d'assurer une surveillance mutuelle des autres duplicats. Dans cette optique auto-réparatrice, l'objectif est qu'à tout instant, au moins un duplicat soit en état de marche. En d'autres termes, il est nécessaire que le temps moyen pour réparer un duplicat défectueux (*mean time to repair* (*MTTR*) en anglais) soit inférieur ou égal au temps moyen entre deux erreurs (*mean time between failures* (*MTBF*) en anglais). Le nombre total de duplicats dans l'anneau dépend quant à lui de l'écart à la moyenne lié au MTBF. En effet, plus il sera grand, plus le nombre de duplicats nécessaires sera important.

La structure choisie pour répondre à ce besoin est la plus intuitive : une structure en anneau. Il s'agit d'une structure dans laquelle, en ce qui concerne la détection de changements qui affectent la structure de l'anneau, chaque duplicat joue un rôle identique, les uns vis-à-vis des autres. Ceci n'est cependant pas le cas pour les aspects relatifs à la réparation. Ainsi, un duplicat est en charge :

- d'envoyer régulièrement une preuve de fonctionnement (i.e. une pulsation cardiaque), à un duplicat donné, appelé *prédécesseur* ;

- de détecter la défaillance d'un autre duplicat, appelé *successeur*.

Chaque membre de l'anneau dispose d'un identifiant logique unique et il existe un ordre total entre ces identifiants, qui permet de définir les notions de prédécesseur (cf. équation 6.1) et de successeur (cf. équation 6.2) d'un élément E de l'anneau R :

$$\begin{aligned} P \text{ est le prédécesseur de } E \text{ si et seulement si} \\ \forall F \in R,\ id(E) \geq id(F) \geq id(P) \\ \text{ou } \forall F \in R \text{ tel que } id(F) > id(E),\ id(E) < id(P) \leq id(F) \end{aligned} \tag{6.1}$$

$$\begin{aligned} S \text{ est le successeur de } E \text{ si et seulement si} \\ \forall F \in R,\ id(E) \leq id(F) \leq id(S) \\ \text{ou } \forall F \in R \text{ tel que } id(F) < id(E),\ id(F) \leq id(S) < id(E) \end{aligned} \tag{6.2}$$

La mise en œuvre technique des comportements client et serveur du mécanisme de heart beat est la même que celle employée dans le cadre de la détection des pannes au niveau des configurateurs (cf. section 6.2.1).

Constitution de l'anneau

Lors de sa construction initiale, l'anneau se compose d'un seul membre (i.e. une instance de gestionnaire d'application) qui tient lieu de serveur d'agents d'amorçage (cf. section 5.2.1).

Par la suite, ce premier membre instancie l'ensemble des autres duplicats qui vont constituer l'anneau. Ceux-ci rejoignent dynamiquement le bus de la même manière que n'importe quelle machine virtuelle applicative dans le cas standard (cf. section 5.2.2). Cette opération s'accompagne d'une mise à jour dynamique de la structure de l'anneau (cf. section suivante).

Nature des duplicats

Au cours du temps, la structure de l'anneau peut évoluer. Ceci résulte du départ de l'un de ses membres (e.g. suite à défaillance) ou à l'arrivée d'un nouveau membre (e.g. insertion dynamique d'un duplicat supplémentaire ou remplacement d'un duplicat défaillant). Afin de ne pas risquer de dupliquer des opérations de suppression ou de création, ce qui pourrait entraîner des incohérences dans la structure de l'anneau[4], il n'existe, à tout instant, qu'un duplicat en mesure de demander la destruction ou la création d'une machine virtuelle auprès de son gestionnaire d'approvisionnement. Ce duplicat est appelé duplicat principal, les autres étant qualifiés de duplicats secondaires.

Pour assurer la pérennité de la structure en anneau face à ces modifications, chaque membre de l'anneau dispose, au sein d'un index, des informations qui caractérisent l'ensemble des autres membres. Ce choix provient du fait que le nombre de membres de

[4] A titre d'exemple, le remplacement non concerté d'un membre défaillant par deux membres remplaçants, supposés prendre la place vacante mais qui s'ignorent, peut empêcher la refermeture de l'anneau.

l'anneau n'a pas vocation à être important. En effet, comme il a été indiqué plus haut, il est étroitement lié au nombre maximum de pannes pouvant survenir durant un laps de temps donné. Plus précisément, plus l'anneau peut être confronté pendant longtemps à une fréquence de survenue de défaillance supérieure à la fréquence de réparation (i.e. l'inverse du temps de réparation), plus le nombre de membres dans l'anneau devra être important. De manière générale, si l'anneau n'est pas capable, en moyenne de s'auto-réparer en moins de temps que ne surviennent en moyenne les pannes (i.e. *mean time between failures* ou *MTBF*), le nombre de ses membres sera, en moyenne, perpétuellement insuffisant. L'hypothèse est donc faite dans la suite, que les défaillances qui affectent l'anneau surviennent avec une fréquence maximale et un *MTBF* suffisamment faibles.

Ainsi, la structure en anneau garantit qu'un membre de l'anneau est surveillé par un membre et un seul. Ceci permet de ne pas avoir à élaborer de mécanismes de décision majoritaires, qui peuvent s'avérer complexes à mettre en œuvre. A l'inverse, la structure totalement maillée contenue dans l'index permet de reconstruire aisément la structure de l'anneau en cas de défaillances multiples voire massives (i.e. gestion des catastrophes).

Pour chaque membre de l'anneau, l'information contenue dans cette table de hachage correspond à son identifiant logique, les paramètres qui permettent de le contacter (i.e. identifiant de serveur d'agents, adresse IP et port) ainsi que son état (cf. tableau 6.1).

Etat	**Description**
α	le duplicat fonctionne correctement et il s'agit d'un duplicat secondaire
μ	le duplicat fonctionne correctement et il s'agit du duplicat principal
δ	le duplicat est déclaré comme étant défaillant
ϕ	le duplicat est défaillant de façon certaine

TABLE 6.1 – Etats d'un duplicat participant à l'anneau

Chaque membre de l'anneau détermine quel sont son prédécesseur et son successeur à l'aide de la représentation locale dont il dispose, en ne considérant que les membres dont l'état n'est ni δ, ni ϕ. Lors de chaque modification de structure de l'anneau, un membre dont le successeur est modifié réinitialise son délai de garde à la valeur utilisée pour un déploiement initial (i.e. valeur majorée).

Modification dynamique de l'anneau

L'algorithme de gestion de l'anneau mis en œuvre au niveau d'un de ses membres est résumé dans la figure 6.8. Il se décompose en deux sous-fonctions, l'une dédiée au départ d'un membre de l'anneau, l'autre à l'arrivée d'un nouveau membre.

Départ d'un membre de l'anneau

A tout instant un membre de l'anneau peut recevoir deux types d'événements :

- la détection d'une défaillance de son successeur ;
- une mise à jour de l'état d'un membre de l'anneau émise par un membre tiers.

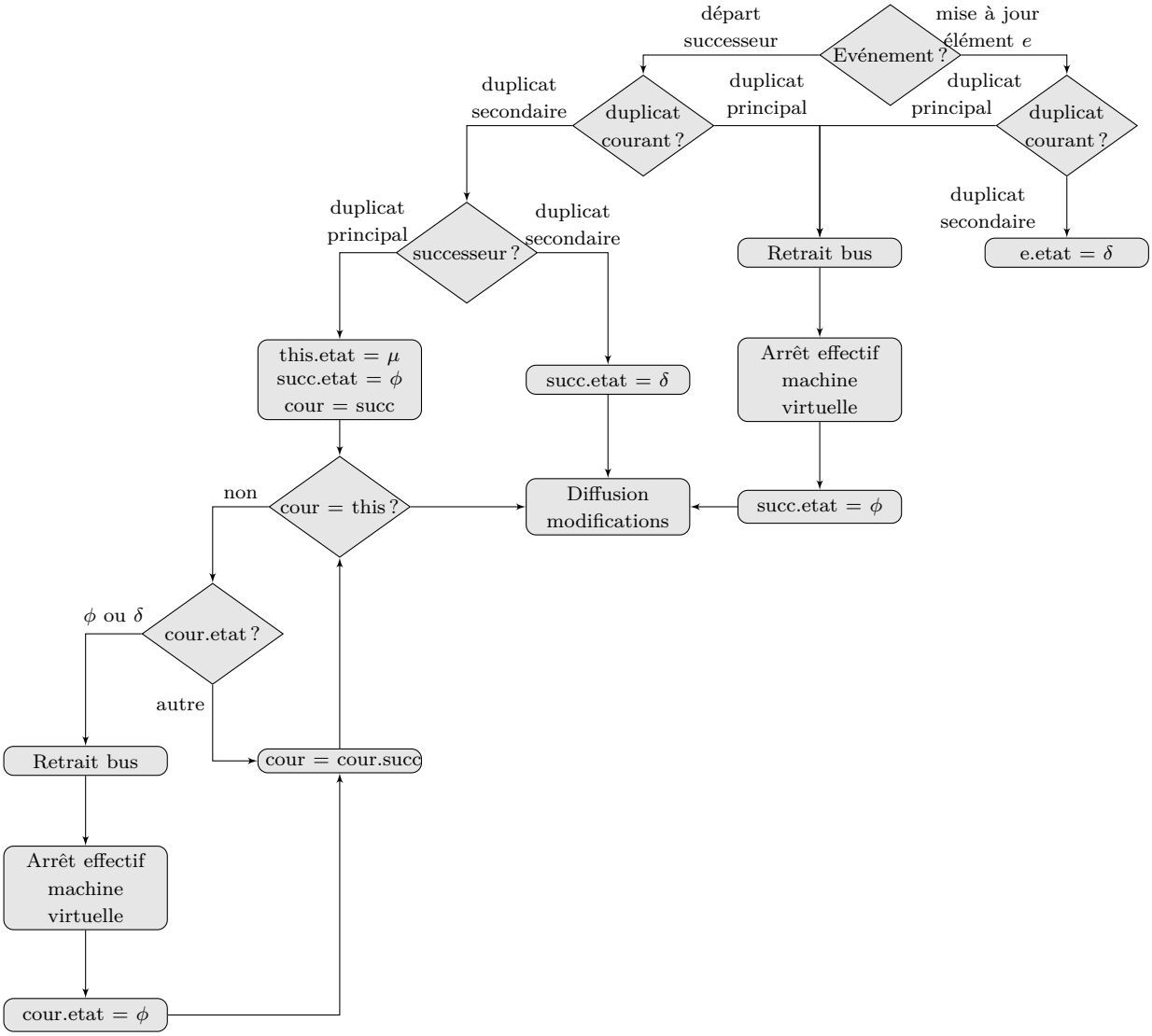

FIGURE 6.8 – Algorithme de gestion dynamique de l'anneau

Lorsque le duplicat D détecte que son successeur est défaillant :

1. si D est le duplicat principal

   (a) Afin de garantir l'élimination effective d'un éventuel faux positif, il retire du bus le serveur d'agents associé au duplicat défectueux puis il notifie la plateforme d'IaaS via le gestionnaire d'approvisionnement, pour qu'elle arrête la machine virtuelle correspondante.

   (b) Par la suite, il positionne à ϕ l'état de son successeur dans sa représentation locale des membres qui constituent l'anneau.
   (c) Il diffuse ensuite, à l'ensemble des autres duplicats, cette modification, au moyen d'un message asynchrone.

2. si D est un duplicat secondaire et que son successeur est également secondaire
   (a) Il positionne à δ l'état de son successeur dans sa représentation locale des membres qui constituent l'anneau.
   (b) Il diffuse ensuite, à l'ensemble des autres duplicats, cette modification, au moyen d'un message asynchrone.

3. si D est un duplicat secondaire et que son successeur est le duplicat principal
   (a) Il positionne à μ son propre état dans sa représentation locale des membres qui constituent l'anneau.
   (b) Afin de garantir l'élimination effective d'éventuels faux positifs, il retire du bus l'ensemble des serveurs d'agents associés à des duplicats dont l'état est déclaré comme défaillant (i.e. état égal à δ) dans sa représentation locale de l'anneau, puis il notifie la plate-forme d'IaaS via le gestionnaire d'approvisionnement, pour qu'il arrête les machines virtuelles correspondantes. Il procède de même avec son successeur.
   (c) Par la suite, il positionne à ϕ, dans sa représentation locale des membres qui constituent l'anneau, l'état de son successeur ainsi que des membres dont l'état est à δ.
   (d) Il diffuse ensuite, au moyen d'un message asynchrone, à l'ensemble des autres duplicats, les modifications ainsi apportées.

Lorsque qu'un duplicat D reçoit une notification de mise à jour de l'état d'un autre duplicat :

- si D est le duplicat principal et que l'état spécifié dans la notification est δ, D procède à son élimination de l'anneau (cf. point 1 de la description ci-dessus) ;
- sinon, D procède à la mise à jour de l'information correspondante dans sa représentation locale de l'anneau.

Arrivée d'un membre de l'anneau

Tout comme pour la gestion du départ d'un membre de l'anneau, l'arrivée dynamique d'un nouveau membre est toujours réalisée à l'initiative de l'un des duplicats, en l'occurrence, le duplicat principal. Celui-ci procède selon les étapes suivantes :

1. Dans un premier, il s'adresse à la plate-forme d'IaaS via le gestionnaire d'approvisionnement, pour qu'elle crée une machine virtuelle qui s'intègre au bus à messages selon le protocole décrit en section 5.2.2. Le nouvel arrivant se fixe un délai de garde

pour être capable de déterminer son successeur et son prédécesseur dans l'anneau, sans quoi il s'arrête automatiquement afin d'éviter le risque d'un membre ne faisant pas partie de l'anneau mais continuant à fonctionner.

2. Par la suite, le duplicat à l'origine de l'insertion ajoute ce nouveau membre dans sa représentation locale des membres qui constituent l'anneau et positionne son état à α.

3. Il diffuse ensuite, au moyen d'un message, à l'ensemble des autres duplicats, la modification ainsi apportée. Au niveau du nouvel entrant, cette mise à jour correspond au contenu intégral de la connaissance locale de la structure de l'anneau dont dispose D.

6.3.2.2 Fiabilisation par recouvrement

Le rôle principal du gestionnaire d'application est d'orchestrer la mise en œuvre dynamique du bus à messages (cf. section 5.2.2). Pour cela, il maintient de manière persistante, grâce à un mécanisme de duplication autoréparable (cf. section 6.3.2.1), la configuration courante du bus à messages au cours du temps. A partir de cette configuration ainsi que de la description applicative selon le formalisme OVF étendu, une instance de gestionnaire d'application est capable de restaurer le modèle à l'exécution associé à l'application en cours de déploiement. Ainsi, si une défaillance affecte l'instance principale du gestionnaire d'application, l'instance remplaçante procède de la manière suivante :

- A partir de la description applicative au format OVF, elle recalcule la version initiale du modèle à l'exécution ;

- En s'appuyant sur la configuration du bus à messages, elle détermine quels configurateurs ont réussi à rejoindre le bus avant la défaillance et sont donc désormais correctement configurés. En effet, au moment de son instanciation, chaque configurateur présent sur une machine virtuelle applicative, dispose d'un délai maximum pour parvenir à s'intégrer dans le bus. Ceci garantit que, si l'instance de gestionnaire d'application vient à défaillir avant qu'un configurateur ne parvienne à rejoindre le bus, ce dernier disparait automatiquement ;

- Elle contacte chaque configurateur déjà présent pour lui indiquer qu'elle est la nouvelle instance principale de gestionnaire d'application afin qu'ils s'adressent à elle, tant pour les aspects de fiabilisation (i.e. envoi de la pulsation cardiaque) que de monitoring (remontée de l'état des composants). A ce propos, chaque configurateur retransmet les informations de monitoring dont la nouvelle instance de gestionnaire d'application pourrait ne pas disposer.

- Une fois le modèle à l'exécution restauré, elle poursuit, si nécessaire, le déploiement applicatif interrompu par la défaillance, en instanciant les machines virtuelles correspondantes aux configurateurs manquants.

6.4 Conclusion

La fiabilisation de la solution de déploiement proposée dans VAMP se décompose en trois niveaux :

- Au niveau applicatif, le protocole d'autoconfiguration est fiabilisé par compensation des machines virtuelles applicatives, en s'appuyant sur un gestionnaire d'application lui-même fiable.
- La fiabilisation de ce gestionnaire d'application constitue le second niveau. Elle consiste à dupliquer le gestionnaire d'application au sein d'une structure auto-réparable en anneau, afin d'assurer la persistance de la configuration courante du bus à messages. Cette structure correspond à l'imbrication mutuelle d'un ensemble d'architecture de type duplication passive :
  - d'une part, chaque duplicat est en charge de la surveillance d'un autre duplicat appelé successeur. Il joue alors le rôle de duplicat principal dans la surveillance de son successeur vis-à-vis des autres membres de l'anneau qui sont autant de duplicats secondaires ;
  - d'autre part, l'un des duplicats, est identifié comme duplicat principal pour ce qui est du pilotage du départ d'un duplicat de l'anneau, de l'arrivée d'un nouveau duplicat ainsi que de l'insertion des configurateurs applicatifs et de la gestion de leur fiabilité.

  Concernant le reste de l'état du gestionnaire d'application, il est fiabilisé par un mécanisme de rejeu basé sur la description applicative initiale et la configuration courante du bus à messages.
- Enfin, le dernier niveau correspond à la fiabilisation du portail qui s'appuie sur les approches de duplication éprouvées avec ce genre d'éléments.

Troisième partie

Expérimentations et résultats

Chapitre 7

Evaluation

Sommaire

7.1 Généricité de VAMP 146

7.1.1 Springoo 146

7.1.2 Clif 148

7.1.3 CADP 149

7.1.4 TUNe 150

7.2 Performances et efficacité de VAMP 151

7.2.1 Métriques 151

7.2.2 Contexte technique 152

7.2.3 Déploiements multiples 153

7.2.4 Déploiement large échelle 158

7.3 Conclusion 160

En réponse aux principales limitations des systèmes de déploiement d'applications dans le nuage (cf. section 2.4), la finalité de VAMP est de proposer une solution capable d'instancier une application patrimoniale de manière générique (i.e. indépendamment de son domaine métier et des choix techniques ou architecturaux qui lui sont associés) et autonome (i.e. en limitant le recours à une éventuelle intervention humaine) (cf. chapitre 1). De plus, au-delà de cet objectif, VAMP présente un certain nombre de propriétés non-fonctionnelles :

Déploiements multiples : capacité à faire face à un nombre important de requêtes utilisateur nécessitant de procéder au déploiement simultané (i.e. en parallèle) d'un grand nombre d'instances d'applications différentes ;

Déploiement large-échelle : faculté d'assurer le déploiement d'une application répartie sur un grand nombre de machines virtuelles ;

L'objectif de ce chapitre est donc d'évaluer le degré d'autonomie, de généricité et de gestion du passage à l'échelle des mécanismes proposés par VAMP. Ainsi, il s'organise en trois sections. La section 7.1 illustre, de manière qualitative, au travers de l'étude de quatre cas d'utilisation, le caractère agnostique de VAMP vis-à-vis des applications patrimoniales déployées ainsi que de leur environnement d'exécution. La section 7.2 fournit quant à elle, une évaluation quantitative des performances de VAMP. Elle se focalise notamment sur le surcoût introduit par la solution, sa capacité à gérer des Déploiements multiples et large échelle. Enfin, la section 7.3 se conclut par une synthèse du chapitre.

7.1 Généricité de VAMP

Selon la définition proposée en section 2.1.2.2, la généricité désigne la capacité du système à déployer n'importe quelle application patrimoniale virtualisable répartie. Elle se décompose en trois composantes que sont la polyvalence, l'indépendance vis-à-vis de l'environnement d'exécution applicatif et l'adhérence applicative du système de déploiement. L'évaluation du degré de généricité proposé par VAMP a donc fait l'objet d'une évaluation qualitative consistant à l'utiliser dans le cadre du déploiement de quatre applications s'appuyant sur des domaines métiers, des architectures et des technologies spécifiques. Cette section présente donc chacun de ces cas d'utilisation et la manière dont il a été modélisé pour être déployé à l'aide de VAMP. Toutes les applications présentées ont fait l'objet de déploiements sur plusieurs distributions du système Linux (i.e. Debian, Ubuntu, CentOS) ainsi que sur de multiples infrastructures de virtualisation (i.e. IaaS OpenStack, IaaS Eucalyptus, cluster d'hyperviseurs Xen, cluster d'hyperviseurs KVM).

7.1.1 Springoo

Le premier cas d'utilisation de VAMP est l'application web JEE Springoo qui tient lieu de fil directeur de ce manuscrit (cf. section 1.4.3). La figure 7.1 illustre la manière dont l'architecture applicative de Springoo a été modélisée dans le cadre de son déploiement à l'aide de VAMP[1]. Ainsi, les éléments en orange (i.e. gris soutenu) définissent les composants ou *wrappers* utilisés pour modéliser les unités fonctionnelles (i.e. le tiers présentation, le tiers applicatif, et le tiers base de données), représentées en jaune (i.e. gris très clair), qui composent l'application elle-même. Dans le cas présent, le tiers applicatif de Springoo est constitué de deux serveurs Java EE sur chacun desquels s'exécutent un serveur d'applications JOnAS, le code applicatif (EAR) ainsi qu'un connecteur JDBC (RAR) connecté au tiers base de données. Ce nombre est totalement arbitraire et le *wrapper* en charge du tiers de présentation est capable de fonctionner avec un nombre arbitraire d'éléments du tiers applicatif.

La modélisation proposée définit cinq types de composant :

[1]Dans un souci de lisibilité de la figure 7.1, un nom est reporté au niveau de chaque liaison. Il correspond au nom que partagent, de façon arbitraire, l'interface cliente et l'interface serveur associées au moyen de la liaison.

db-wrp : un *wrapper* en charge d'encapsuler le SGBD MySQL et la base de données. Il est en charge d'exposer les informations de connexions nécessaires aux clients de la base de données (i.e. les connecteurs JDBC) et d'assurer le démarrage, l'arrêt et le redémarrage du SGBD.

jee-wrp : il s'agit du *wrapper* en charge d'un serveur d'applications Java EE JOnAS. Il expose les informations relatives à ce dernier (e.g. le nom du répertoire dans lequel il s'exécute, le nom de l'instance, etc.). Il est également en charge du démarrage, de l'arrêt et du redémarrage du serveur d'applications JOnAS.

rar-wrp : il s'agit du *wrapper* en charge d'un connecteur JDBC. Il dépend à la fois du db-wrp et du jee-wrp desquels il reçoit les informations nécessaires à son instanciation, son exécution et son accès à la base de données.

ear-wrp : encapsulant le code applicatif, ce *wrapper* dépend du jee-wrp dont il reçoit les informations nécessaires à son instanciation et son exécution mais également du rar-wrp, dont il dépend uniquement en terme d'ordre d'activation.

http-wrp : ce *wrapper* est en charge du serveur HTTP et du composant de répartition de charge qu'il comprend. Il dépend de chaque jee-wrp dont il reçoit les données nécessaires à la configuration du composant de répartition de charge. Il réalise les opérations de démarrage, d'arrêt et de redémarrage du serveur HTTP.

Le tableau 7.1 résume les dépendances entre ces cinq types de *wrapper* en fonction des interfaces qu'ils exposent.

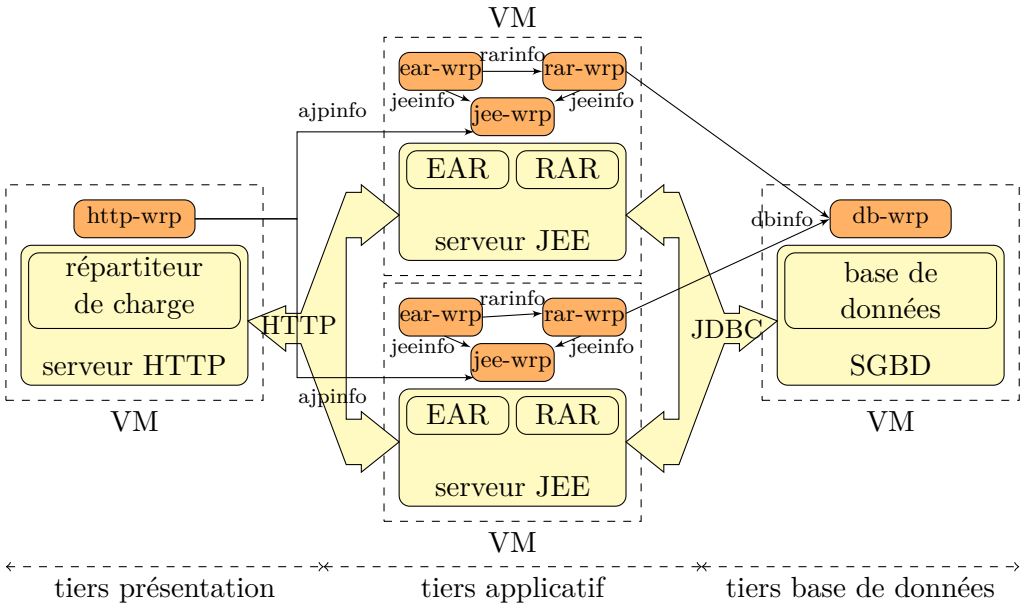

FIGURE 7.1 – Mise en œuvre de Springoo à l'aide de VAMP

Wrapper client	Interface cliente	Wrapper serveur	Interface serveur	Données de configurations partagées	Contingence
rar-wrp	dbinfo	db-wrp	dbinfo	IP address, database name, database user and password	obligatoire
rar-wrp	jeeinfo	jee-wrp	jeeinfo	IP address, JOnAS home directory, JOnAS working directory, JOnAS instance name, JOnAS domain, Java home directory	obligatoire
ear-wrp	jeeinfo	jee-wrp	jeeinfo	IP address, JOnAS home directory, JOnAS working directory, JOnAS instance name, JOnAS domain, Java home directory	obligatoire
ear-wrp	rarinfo	rar-wrp	rarinfo	IP address	obligatoire
http-wrp	ajpinfo	jee-wrp	ajpinfo	IP address, AJP port, JVM route	obligatoire

TABLE 7.1 – Caractérisation des liaisons entre les *wrappers* utilisés pour modéliser l'application Springoo

Il est important de souligner que la modélisation proposée n'est pas unique. La granularité de la représentation définit la finesse avec laquelle VAMP permet de déployer l'application. Ainsi, dans le cas présent, le serveur HTTP ne dépend que des serveurs d'applications pour démarrer alors que dans une modélisation plus grossière (i.e. comme dans le cas représenté par la figure 3.3) son démarrage est également conditionné par l'activation du code applicatif et du connecteur JDBC. En effet, dans cette seconde représentation, le composant *jee-wrp* encapsule le serveur d'applications, le code applicatif ainsi que le connecteur JDBC.

7.1.2 Clif

Clif est un système d'injection de charge open source [56], développé dans le cadre du consortium OW2. Il permet de soumettre un système sous test (*system under test* ou *SUT*) à un niveau de charge modulable et d'en observer le comportement associé. Clif est intégralement écrit en Java édition standard (i.e. Java SE). Son architecture diffère totalement de celle d'une application web multi-tiers. En effet, Clif offre une structure en étoile comprenant :

un superviseur : il s'agit d'une entité centrale chargée d'orchestrer l'injection de la charge appliquée au *SUT* et de collecter les données de monitoring permettant d'en observer le comportement ;

un ensemble de serveurs Clif : il s'agit d'éléments périphériques correspondant à des injecteurs de charge et des sondes d'observation.

Les injecteurs sont pilotés par le superviseur alors que les sondes lui remontent les données collectées. Pour cela, chaque serveur Clif doit disposer des informations nécessaires à la communication avec le superviseur (e.g. adresse IP et port). La figure 7.2 illustre donc l'architecture Clif et la modélisation qui en a été proposée. L'application utilisée en tant que *SUT* est Springoo. La dépendance entre le composant de supervision

Clif et le *SUT* correspond à la récupération du point d'accès à l'application au travers duquel l'injection va être réalisée.

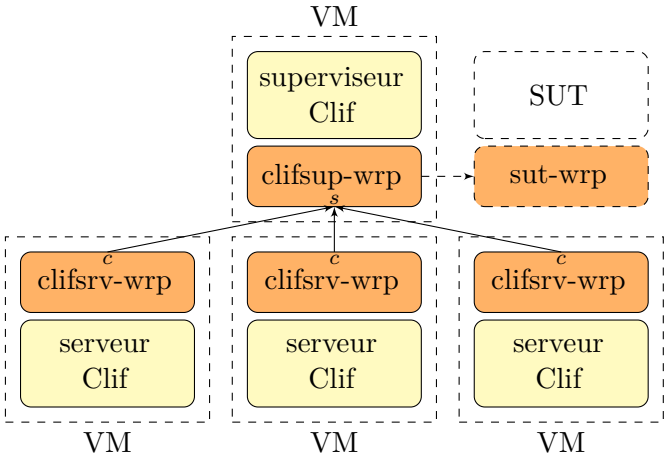

FIGURE 7.2 – Mise en œuvre d'une instance Clif à l'aide de VAMP

7.1.3 CADP

Un autre cas d'utilisation de VAMP est le système de vérification formelle de protocole CADP [71], proposé par l'équipe CONVECS de l'INRIA et qui a déjà été mentionné en section 5.3.2 dans le cadre de la vérification formelle du protocole réparti d'autoconfiguration de VAMP.

Un environnement CADP se compose d'un ensemble de N *nœuds* applicatifs identiques, codés en C. L'ensemble de ces nœuds sont interconnectés deux à deux, de façon bidirectionnelle, conférant à l'ensemble une structure de graphe complet. Ainsi, chaque nœud est une unité d'exécution qui fournit un service à l'intention des $N-1$ autres nœuds et qui requiert le service fourni par chacun d'eux. En d'autres termes, pour tout couple de nœuds (n_c, n_s) de l'environnement CADP considéré, tels que n_c soit différent de n_s, il existe un canal de communication bidirectionnel associant n_c à n_s.

La modélisation de l'environnement CADP au travers de VAMP a donc consisté à encapsuler chaque nœud CADP au sein d'un *wrapper* exposant une interface serveur s et $N-1$ interfaces clientes c_i où i désigne l'indice du nœud exposant l'interface serveur associée. La figure 7.3 illustre cette architecture dans le cas de 4 nœuds. A titre expérimental, CADP a été déployé sur un nombre de nœuds variant de 2 à 80, chacun d'eux étant instancié dans une machine virtuelle indépendante. Par ailleurs, deux modèles de contingence ont été instanciés :

pas de dépendance de démarrage : toutes les liaisons disposaient d'une contingence optionnelle ;

un maximum de dépendances de démarrage sans interblocage : la contingence

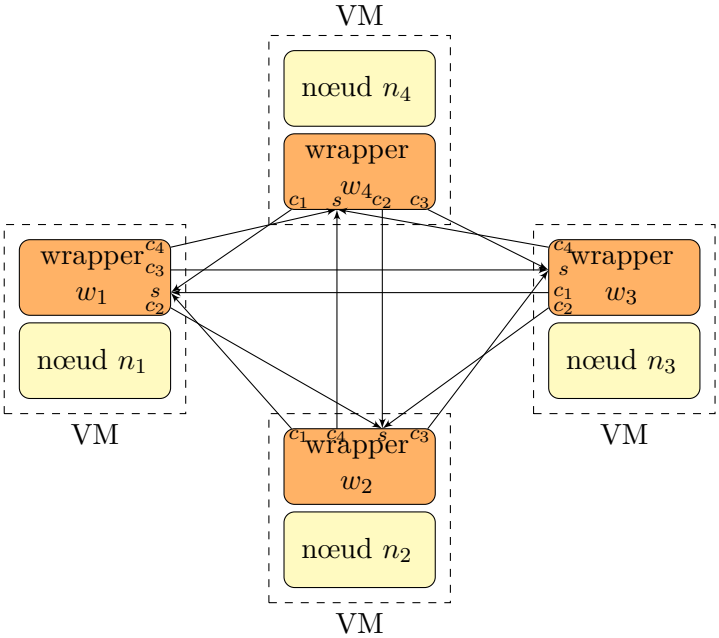

FIGURE 7.3 – Mise en œuvre d'une instance CADP à l'aide de VAMP

d'une liaison entre l'interface cliente c_j d'un *wrapper* w_i et l'interface serveur s du *wrapper* w_j est obligatoire si $i > j$. Dans le cas contraire, elle est optionnelle.

7.1.4 TUNe

Dans le cadre du projet FSN OpenCloudware[2], une collaboration avec l'équipe Astre de l'Institut de Recherche en Informatique d Toulouse (IRIT) a donné lieu au déploiement du système d'administration autonome TUNe [34], qui a fait l'objet d'une présentation en section 2.2.3.3.

La structure de TUNe s'organise autour d'un gestionnaire centralisé et d'un ensemble d'agents répartis sur chaque machine virtuelle applicative. Plus précisément, chaque machine virtuelle exécute un *RemoteLauncher* en charge d'exécuter les ordres en provenance du gestionnaire central de TUNe et de lui remonter les données de monitoring et un *RemoteWrapper* dont la fonction demeure locale à la machine virtuelle.

Pour que le gestionnaire centralisé soit en mesure de communiquer avec les *RemoteLauncher*s, il est nécessaire que ceux-ci se soient préalablement référencés auprès de lui : au cours de cette opération de référencement, le *RemoteLauncher* fournit ses informations de connexion (i.e. adresse IP et port) au gestionnaire central.

Ainsi, l'architecture de TUNe se présente sous forme d'une structure en étoile, semblable à celle de Clif, mais dans laquelle le sens des dépendances entre le composant central et composants périphériques est inversé : le gestionnaire central requiert le ser-

[2] `http://www.opencloudware.org/`

vice fourni par chaque agent périphérique (cf. figure 7.4). La modélisation proposée comporte un *wrapper* pour le gestionnaire centralisée de TUNe (*TUNeCentral*) et un *wrapper* (*TUNeAgent*) pour l'ensemble des agents (i.e. *RemoteLauncher* et *RemoteWrapper*) colocalisés sur une machine virtuelle donnée.

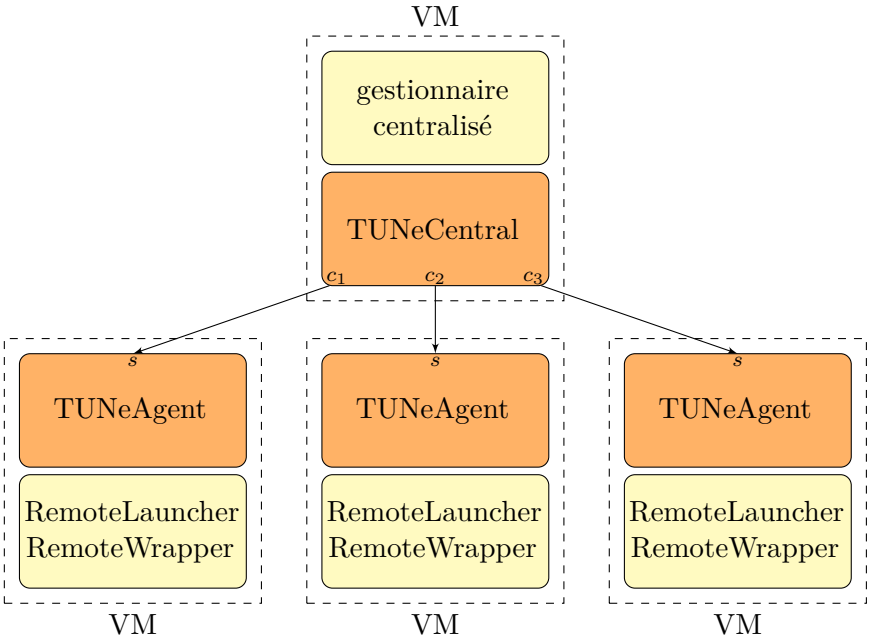

FIGURE 7.4 – Mise en œuvre d'une instance TUNe à l'aide de VAMP

7.2 Performances et efficacité de VAMP

La seconde phase de l'évaluation de VAMP consiste à estimer quantitativement son efficacité à déployer des applications dans le nuage. Cette estimation se concentre uniquement sur la mesure des performances en terme de vitesse de déploiement et ne traite pas, par exemple, d'aspects relatifs à la sécurité.

7.2.1 Métriques

Les métriques retenues dans le cadre de cette évaluation sont un ensemble de durées, représentées sur la figure 7.5. Leur valeur a été mesurée pour chaque machine virtuelle applicative mise en œuvre au travers de VAMP. Chacune de ces métriques est constituée de deux composantes. La première, qui sera qualifiée de *générique*, est indépendante de l'application déployée et représente la contribution de la plate-forme VAMP sur la valeur globale de la métrique. A l'inverse, la seconde, qualifiée de *spécifique à l'application* ou *spécifique*, quantifie l'impact de l'application elle-même sur l'estimation de la métrique. Comme cela sera illustré plus loin, la répartition entre la composante générique et la

composante spécifique varie selon la métrique considérée mais également selon l'application déployée. En effet, pour une même métrique, cette dernière peut induire plus ou moins de traitements spécifiques.

7.2.2 Contexte technique

L'environnement technique dans lequel ces tests ont été réalisés est une solution propriétaire d'*IaaS*, instanciée sur la plate-forme d'expérimentation Grid 5000 [31]. Grid 5000 est le résultat d'une initiative soutenue par le Ministère français de la Recherche et développée au sein de l'action ACI GRID qui regroupe un ensemble de partenaires parmi lesquels se trouvent l'INRIA, le CNRS et RENATER. Il s'agit d'un système ouvert capable d'exploiter et d'administrer une grille de calcul dédiée aux activités de recherche en informatique. L'infrastructure matérielle sur laquelle repose Grid 5000 est composée de *clusters* de machines interconnectés. Chacun d'eux est mis à disposition et exploité au sein d'un *site*. Grid 5000 regroupe une vingtaine de sites en France et au Luxembourg. Chaque *cluster* est constitué de machines disposant d'une configuration matérielle identique. La solution d'*IaaS* retenue pour cette évaluation a été déployée sur un seul *cluster* composé de serveurs Dell PowerEdge R410 proposant les caractéristiques suivantes :

- une architecture regroupant 16 cœurs au sein d'un processeur Intel Xeon E5620 2.40GHz supportant la virtualisation matérielle ;
- 16 Go de RAM ;
- 2 disques durs SAS RAID-0 d'une capacité de 150 Go chacun.

L'ensemble de ces machines est interconnecté au moyen d'un réseau local Ethernet 1 Gbps. En outre, la pile logicielle de chaque machine se compose d'une distribution Debian

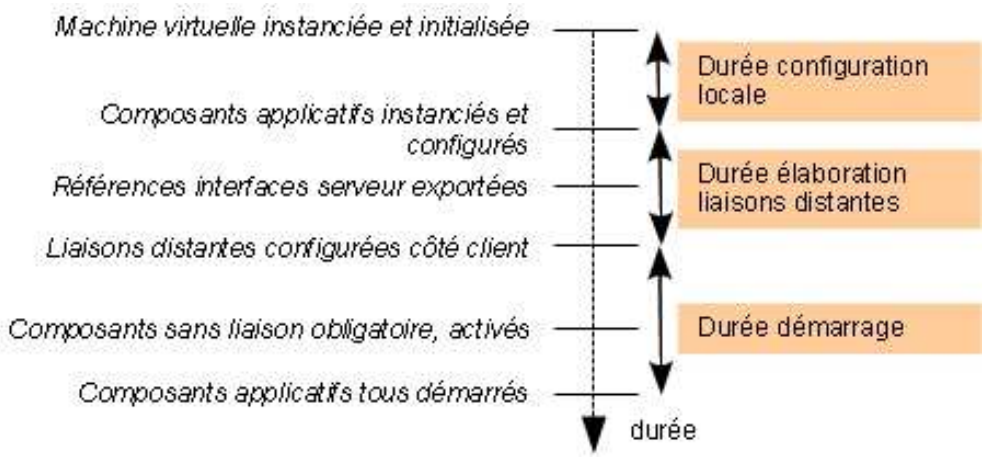

FIGURE 7.5 – Métriques retenues pour évaluer le mécanisme de déploiement proposé par VAMP

Squeeze du système d'exploitation Linux (noyau version 2.6.32-5-amd64), du système de virtualisation Xen (version 1.3) ainsi que des binaires et des données nécessaires à l'exécution de la solution privée d'*IaaS*.

L'architecture de la solution d'*IaaS* s'organise de la façon suivante :

- un frontal, instancié sur l'une des machines physique du *cluster* et constituant un point d'accès unique à la plate-forme. Il regroupe les fonctions de publications d'images, de répartition des requêtes utilisateurs et d'administration globale à la solution proposées par la plupart des plateformes d'*IaaS*, telles qu'Eucalyptus (cf. section 2.3.1.1).

- un ensemble de nœuds d'instanciation répartis sur les autres machines physiques du cluster. Chacun d'eux est en charge de la création et la suppression des machines virtuelles locales ainsi que de la remontée d'information de monitoring.

Le principal intérêt de recourir à une solution d'*IaaS* propriétaire et de pouvoir en maîtriser le comportement. Ainsi, d'une part, la politique de répartition des machines virtuelles sur les nœuds d'instanciation a pu être configurée pour être la plus équilibrée possible. Plus précisément, lors du déploiement d'une ou plusieurs applications, le nombre d'instances d'une machine virtuelle applicative d'un type donné –le type étant défini par l'ensemble de ses caractéristiques matérielles et logicielles– est le même, à un près, sur chaque n*oe*ud d'instanciation. De cette manière, chaque nœud d'instanciation est soumis à une charge à peu près identique en terme de consommation de ressources (CPU, mémoire, flux d'entrées/sorties, ...). D'autre part, cette solution d'*IaaS* a également permis d'exhiber les gains introduits par des mécanismes de cache d'images ou de pré-approvisionnement de machines virtuelles.

Concernant la génération d'appliances virtuelles, la plate-forme utilisée pour mettre en œuvre le système UForge [123] se compose d'un serveur frontal (32 bits Xeon 2x2.8 GHz, 2 Go DDR2) qui expose un service web, et d'un serveur de traitement (64 bits Xeon E5405 2x2 GHz, 4 Go de RAM) en charge de la création des images.

Enfin l'implémentation de référence de VAMP, qu'il s'agisse des gestionnaires, des agents de configuration ou des *wrappers*, repose sur Julia [7] (i.e. l'implémentation de référence Java du modèle à composants Fractal) et sur le bus à messages AAA [27], qui constitue le cœur de JORAM [6] (une implémentation JMS *open source*).

7.2.3 Déploiements multiples

Le processus d'évaluation mis en place pour étudier le comportement de VAMP dans le cadre de Déploiements multiples se décompose en deux phases :

- la première consiste à évaluer chaque métrique pour en étudier la tendance et déterminer le surcoût introduit par VAMP en terme de temps d'exécution.

- la seconde se focalise sur le ressenti utilisateur.

Les mesures présentées ont été obtenues en faisant varier de 8 à 112 le nombre d'instances d'une même application déployées simultanément. Dans le cas d'utilisation présenté ci-dessous, ceci correspondait à un nombre total de machines virtuelles déployées compris entre 24 et 336, soit 3 à 42 par nœud d'instanciation.

7.2.3.1 Cas d'utilisation

Pour réaliser ces mesures, l'application Springoo a tenu lieu de cas d'usage pour évaluer l'efficacité de VAMP. En effet, en tant qu'application web basée sur une technologie JEE, elle est représentative de plus de 80% des applications du système d'information d'Orange. Dans le cas présent, une instance de Springoo était composée d'un serveur HTTP Apache et du répartiteur de charge JK associé, d'un serveur d'applications JEE JOnAS exécutant le code métier de l'application et un mettant en œuvre un connecteur JDBC, et d'un système de gestion de base de données MySQL regroupant les données applicatives au sein d'une base de données. La répartition au sein de machines virtuelles applicatives et la modélisation des entités logicielles applicatives selon 5 types de composants a déjà fait l'objet d'une présentation approfondie en section 7.1.1.

Une instance de Springoo s'appuie donc sur trois images virtuelles distinctes, toutes faisant la même taille (i.e. 4 Go). Chacune d'elles est instanciée dans une machine virtuelle mono-cpu. La machine virtuelle exécutant la logique métier (i.e. celle qui instancie le tiers applicatif) dispose de 512 Mo de mémoire virtuelle alors que les deux autres s'appuient sur une configuration plus modeste comprenant 256 Mo de mémoire virtuelle.

Les données de configuration échangées entre les différents composants modélisant Springoo sont détaillées dans le tableau 7.1. Leur volume ne représente que quelques dizaines d'octets.

7.2.3.2 Impact et tendance

La figure 7.6 illustre l'évolution de chacune des métriques décrites en section 7.2.1 dans le cadre de déploiements multiples simultanées de l'application Springoo. Elle représente également les valeurs obtenues pour la durée d'amorçage des machines virtuelles applicatives mises en œuvre lors de ces déploiements.

Dans le contexte de l'application Springoo, l'instanciation et la configuration locale des *wrappers* VAMP n'induit aucun traitement applicatif spécifique. Ainsi, la durée moyenne de configuration locale se réduit à sa composante générique. Inversement, dans les deux autres cas (i.e. durée d'établissement des liaisons distantes et activation), le poids de la composante générique est très limité voire négligeable. En effet, il représente moins de 3.3% de la durée d'établissement des liaisons distantes et seulement 0.07% de la durée d'activation. En conséquence, une estimation raisonnable du surcoût introduit par VAMP, en terme de temps d'exécution, consiste à comparer la durée de configuration locale à la somme de l'ensemble des métriques représentées sur la figure 7.6.

Le surcoût introduit par VAMP représente donc environ 10% de la durée totale nécessaire pour obtenir une machine virtuelle opérationnelle. D'autre part, dans le cadre de Springoo, ce surcoût a tendance à diminuer en fonction du nombre d'instances de

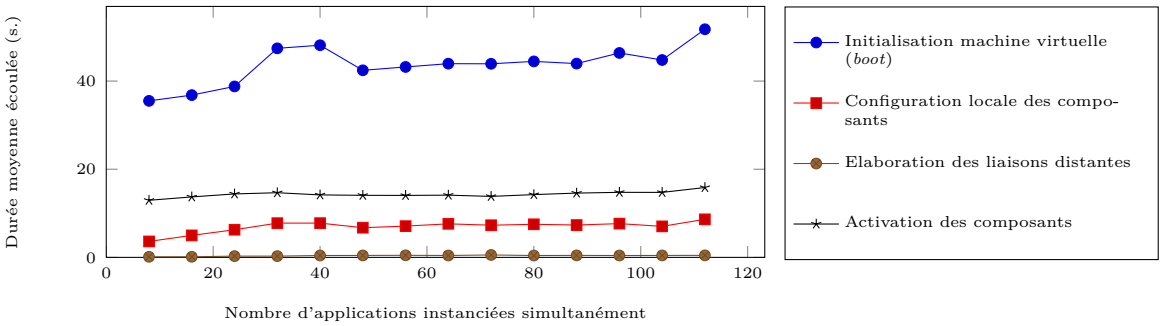

FIGURE 7.6 – Qualification du surcoût introduit par VAMP dans le cadre de déploiements multiples simultanés

Springoo déployées simultanément, la pente de la courbe représentant la durée de configuration locale étant sensiblement moins marquée que celle des durées d'activation et d'amorçage.

Concernant la génération de l'appliance virtuelle associée à Springoo, le temps nécessaire pour créer simultanément les trois images associées est d'environ 42 minutes. La génération d'une image représente à elle seule environ 22 minutes. Il est important de souligner que le surcoût introduit par VAMP est totalement insignifiant au regard du délai de génération des images. Il en résulte qu'une utilisation pertinente de VAMP passe par une phase de pré-configuration réduite à l'intégration dans les images virtuelles applicatives des données de configuration communes à l'ensemble des instances.

7.2.3.3 Ressenti utilisateur et degré de parallélisme

L'objectif de cette seconde phase d'évaluation est de mesurer un ensemble de durées reflétant le ressenti utilisateur dans le cadre du déploiement simultané de N instances d'une application. Les mesures ont concerné :

- le délai moyen de déploiement d'une machine virtuelle applicative (*Mean Time To Deploy 1 VM* ou *MTTD1V*), c'est-à-dire le temps nécessaire pour instancier et initialiser la machine virtuelle applicative puis pour instancier, configurer et activer les composants qu'elle embarque ;

- le délai moyen de déploiement d'une instance d'application (*Mean Time To Deploy 1 Application* ou *MTTD1A*), c'est-à-dire la durée qui s'écoule avant que toutes les machines virtuelles qui composent une instance d'application soient initialisées et que les composants qu'elles embarquent soient activés ;

- le temps moyen de déploiement de N instances d'application (*Mean Time To Deploy N Applications* ou *MTTDNA*), c'est-à-dire le temps nécessaire pour finaliser le déploiement de N instances de l'application.

Comme l'illustre la figure 7.7, chacun de ces métriques présente une tendance linéaire vis-à-vis du nombre d'instances applicatives déployées simultanément. Afin d'estimer le

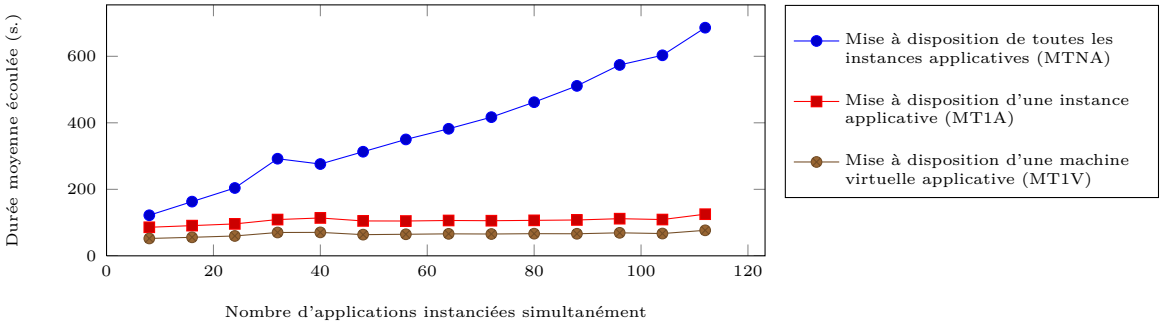

FIGURE 7.7 – Ressenti utilisateur dans la mise en œuvre du déploiement d'applications au moyen de VAMP

degré de parallélisme introduit par VAMP dans le cadre du déploiement simultané de N instances applicatives, deux ratios ont été évalués :

- le premier compare le temps nécessaire à VAMP pour déployer une instance d'application, à la durée de déploiement en séquence de chacune des trois machines virtuelles qui composent l'instance applicative. La valeur du gain introduit par VAMP est donc égale à $1 - MTTD1A/(3 * MTTD1V)$[3]. Comme le montre la figure 7.8, le nombre d'instances applicatives instanciées simultanément (N) n'a qu'un impact très limité sur ce gain dont la valeur avoisine les 94%.

- le second estime le bénéfice de déployer N instances d'application à l'aide de VAMP par rapport au même déploiement réalisé en séquence. Ce ratio est donc égal à $1 - MTTDNA/(N * MTTD1A)$. Lorsque N varie entre 8 et 112, la valeur du ratio converge vers une valeur asymptotique où le gain vaut environ 95%.

Malgré le surcoût introduit par VAMP (cf. section 7.2.3.2), l'observation de ces deux ratios démontre le bénéfice qu'il est en mesure de procurer : le caractère décentralisé et asynchrone du protocole d'autoconfiguration qu'il propose permettent à un administrateur humain de réduire singulièrement (d'un facteur 20) le temps nécessaire au déploiement simultané d'un nombre important d'applications.

7.2.3.4 Optimisation de l'*IaaS*

Le niveau de parallélisme proposé par VAMP est cependant à nuancer. En effet, les valeurs calculées s'appuient sur la mise en œuvre d'un mécanisme de pré-approvisionnement des machines virtuelles au niveau des nœuds d'instanciation de la plate-forme d'*IaaS*.

[3] 3 correspond au nombre de machines virtuelles qui composent une instance de Springoo

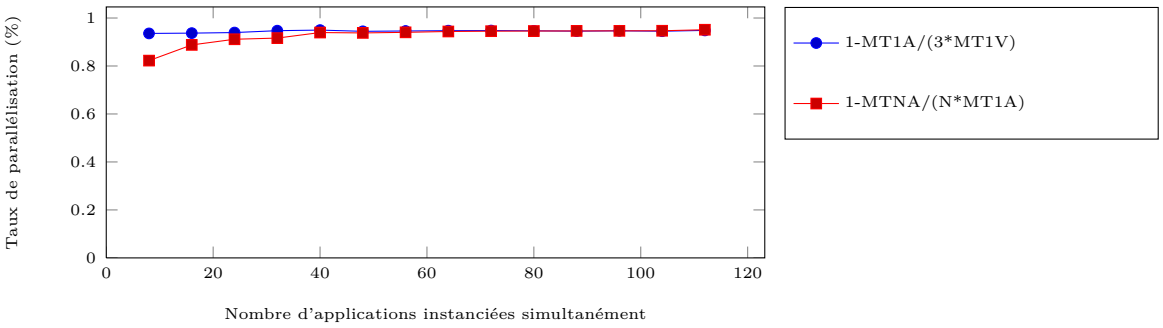

FIGURE 7.8 – Estimation du degré de parallélisme introduit par le processus de déploiement de VAMP

Ce mécanisme consiste à anticiper toute demande d'instanciation d'une nouvelle machine virtuelle applicative, en disposant à tout instant et sur chaque nœud d'instanciation, de l'image adéquate instanciable instantanément. Il s'agit là d'un fonctionnement idéal, presque utopique, mais proposé sous une forme un peu moins optimiste par Cloud Foundry BOSH [1]. Un deuxième mécanisme, plus classique, consiste à gérer un cache d'images au niveau de chaque nœud d'instanciation. Ainsi, la création d'une nouvelle machine virtuelle débute par l'enrichissement, si nécessaire, du cache, puis se poursuit par la création d'une copie de travail de l'image présente dans le cache. Enfin, en l'absence de mécanismes de pré-approvisionnement et de cache, l'instanciation d'une machine virtuelle passe par la création préalable d'une copie de travail à partir de l'image virtuelle stockée dans un référentiel centralisé, accessible via le réseau. La figure 7.9 propose une comparaison de l'impact de ces mécanismes d'optimisation de la couche *IaaS* dans le cadre du déploiement de N applications à l'aide de VAMP.

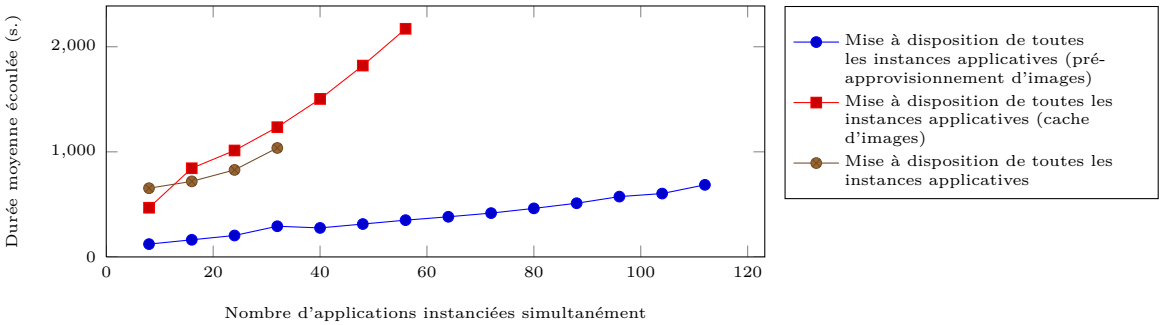

FIGURE 7.9 – Apports des mécanismes de cache d'images et de pré-approvisionnement de machines virtuelles

Ces observations exhibent deux points de contention essentiels dans une solution d'*IaaS* :

le réseau : en l'absence de tout mécanisme d'optimisation, VAMP ne parvient pas à déployer simultanément plus de 32 instances de Springoo. La cause en est l'effondrement du réseau suite à une sollicitation trop importante. En effet, le téléchargement des 96 images virtuelles associées aux 32 instances de Springoo représentent un volume de données avoisinant les 400 Go !

le contrôleur de disque des nœuds d'instanciation : ce second goulot d'étranglement est mis en exergue par la moindre efficacité du mécanisme de cache comparée à celle du téléchargement direct de l'image depuis un référentiel central, Ainsi, bien que le recours à un cache permette de limiter les flux réseaux, la saturation du contrôleur de disque est atteinte lors de l'instanciation simultanée d'environ 168 machines virtuelles (i.e. 670 Go).

7.2.3.5 Synthèse

Grâce à son degré de parallélisme et l'évolution maîtrisée du temps de déploiement (i.e. tendance linéaire), VAMP est une solution pertinente dans la réalisation de Déploiements multiples d'applications. Cependant, son efficacité dépend fortement des optimisations proposées par la plate-forme d'*IaaS* sous-jacente en matière d'instanciation de machine virtuelle.

De plus, le protocole réparti d'autoconfiguration et d'autoactivation de VAMP a fait l'objet d'une collaboration au sein même de l'équipe Sardes de l'INRIA Rhône-Alpes. Celle-ci a consisté à simuler à l'aide de l'environnement *SimGrid* [42], sur une seule machine physique (i.e. un PC de bureau), le comportement de VAMP dans le cadre du déploiement simultané d'un ensemble d'applications sur différentes infrastructures (i.e. Grid 5000, le nuage d'Amazon Web Service, ...). Le résultat de ces simulations a confirmé l'accroissement linéaire des durées de déploiement en fonction du nombre d'applications instanciées en même temps. Le nombre maximum d'applications déployées simultanément au cours de ces simulations s'est élevé à 180.

7.2.4 Déploiement large échelle

Cette deuxième sous-section se focalise sur le comportement de VAMP dans le cadre du déploiement d'une application large échelle, c'est-à-dire répartie sur un grand nombre de machines virtuelles et présentant un nombre important d'interconnexions entre les éléments logiciels qui la composent.

7.2.4.1 Cas d'utilisation

CADP est l'application qui a tenu lieu de cas d'usage dans la réalisation des mesures relatives à la gestion large échelle de VAMP. Elle se compose d'un ensemble de nœuds fonctionnellement équivalents, interconnectés deux à deux de façon bidirectionnelle, formant ainsi une structure de graphe complet (cf. section 7.1.3). Une telle structure correspond à l'architecture applicative la plus défavorable pour un processus d'autoconfiguration, du fait du nombre de références et d'ordres d'activation échangés.

Dans le cadre de cette évaluation, chaque nœud CADP a été instancié sur une machine virtuelle qui lui était propre (1 cpu virtuel, 128 Mo de mémoire virtuelle). L'image virtuelle associée faisait une taille de 800 Mo. D'autre part, les contingences des liaisons entre nœuds ont été positionnées de manière à induire un maximum de dépendances de configuration et de démarrage sans provoquer d'interblocage (i.e. sans créer de cycle composé uniquement de dépendances obligatoires (cf. section 7.1.3).

Le but de ces tests étant de caractériser le comportement du processus d'autoconfiguration d'autoactivation de VAMP dans le cas d'applications large échelle, l'étude a porté sur le déploiement d'instances de CADP de différentes tailles.

7.2.4.2 Résultats

La figure 7.10 illustre les résultats obtenus en faisant varier le nombre de nœuds CADP (i.e. le nombre de machines virtuelles applicatives) de 3 à 120.

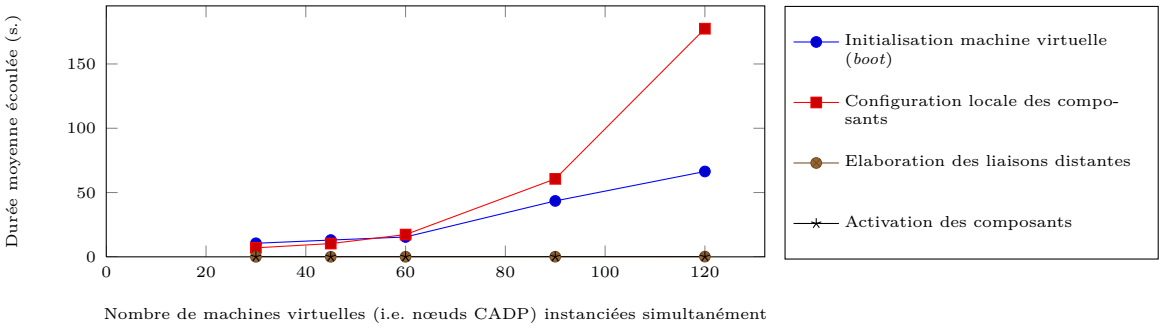

FIGURE 7.10 – Comportement de VAMP lors du déploiement d'applications de grande taille

L'observation de la figure précédente peut se résumer en deux points :

- le caractère linéaire de la progression des métriques relatives à la seconde phase du protocole réparti d'autoconfiguration et d'autoactivation (i.e. établissement des liaisons distantes et activation des composants applicatifs) ;

- la croissance polynomiale de la métrique relative à la première phase de ce même protocole (i.e. la mise en œuvre dynamique du bus à messages AAA)

En effet, bien que chacune des deux phases du protocole ait recours à un nombre de messages comparable (i.e. de l'ordre de n^2 si n désigne le nombre de nœuds CADP déployés), la première phase (i.e. celle de mise en place du bus) est centralisée autour du gestionnaire de déploiement, alors que la seconde (i.e. celle d'autoconfiguration et d'autoactivation des composants applicatifs) est répartie entre les composants.

7.2.4.3 Synthèse

Bien que le cœur du protocole réparti d'autoconfiguration et d'autoactivation proposé par VAMP soit capable d'assurer le déploiement d'applications de grande taille, il n'en demeure pas moins que la mise en œuvre dynamique et centralisée du bus à messages préalablement requis nécessite d'être améliorée. L'une des perspectives issue de cette évaluation consisterait donc à optimiser la première phase du protocole en utilisant des algorithmes plus sophistiqués, s'appuyant sur des structures arborescentes dans la diffusion des messages, offrant ainsi une progression de la durée de configuration locale de l'ordre de $N * log(N)$.

En outre, comme l'illustre la figure 7.11, cette évaluation a permis de mettre en évidence des faiblesses de VAMP dans le cadre de l'instanciation parallèle des machines virtuelles composant une même application. Des modifications ont été apportées afin d'y remédier. Ainsi, la nouvelle implémentation de VAMP (i.e. qui s'appuie sur une

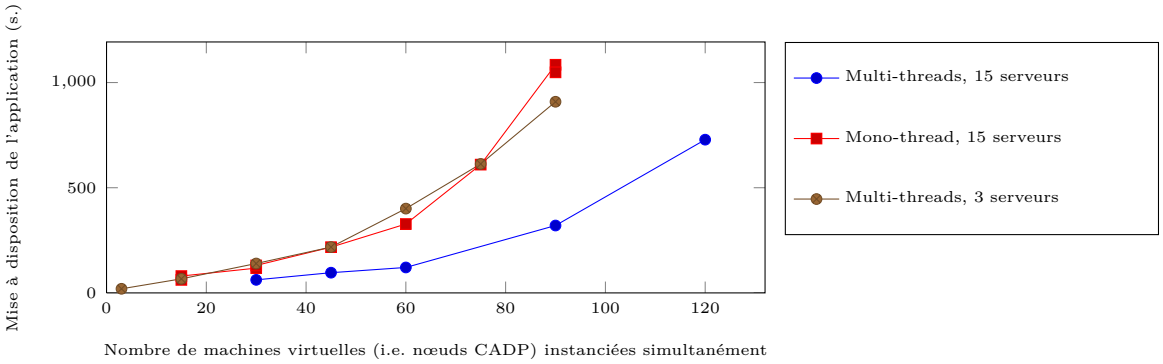

FIGURE 7.11 – Bénéfice introduit dans VAMP par une approche *multithreads*

approche à base de processus légers (i.e. *threads*) multiples, consomme environ 5 fois moins de ressources matérielles que l'implémentation initiale (i.e. comparaison du nombre de machines physiques requises pour obtenir des performances équivalentes).

7.3 Conclusion

L'évaluation expérimentale de VAMP s'est organisée selon deux axes :

- le premier a mis en évidence le caractère générique de la solution au travers du déploiement de quatre applications couvrant des domaines métiers différents et s'appuyant sur des architectures et des technologies distinctes.
- le second s'est focalisé sur l'efficacité de VAMP et sa capacité de passage à l'échelle. Ainsi, bien que le déploiement d'applications de grande taille puisse encore faire l'objet d'amélioration, notamment en perfectionnant la partie du protocole d'autoconfiguration consacrée à la construction dynamique du bus à messages, VAMP

dispose d'un comportement satisfaisant dans le cadre de Déploiements multiples. Une phase de simulation a confirmé les résultats obtenus.

Chapitre 8

Conclusion

Sommaire

8.1 Rappel des motivations 163
8.2 Rappel des contributions 164
8.3 Perspectives 165
8.3.1 Outillage 165
8.3.2 Prise en charge d'autres phases du cycle de vie de l'application 166
8.3.3 Extension de la fiabilité 166
8.3.4 Support multi-nuages 167
8.3.5 Dimensionnement autonome 168

Ce chapitre, qui finalise la présentation faite au travers de ce manuscrit de thèse, s'organise de la manière suivante. La section 8.1 rappelle le contexte dans lequel se sont inscrits les travaux réalisés et la problématique qui en est à l'origine. La section 8.2 résume les contributions de ce doctorat. Enfin la section 8.3 définit un ensemble de pistes de recherche et de développement en lien avec les limitations actuelles de ces contributions.

8.1 Rappel des motivations

Afin de conquérir, le plus rapidement possible, des parts d'un marché en cours de définition (i.e. celui du *PaaS*), les acteurs du *cloud computing* ont développé et proposé à leurs clients des systèmes d'administration autonomes disparates, tant du point de vue des fonctionnalités qu'ils offrent que des domaines métiers, des environnements techniques ou des architectures applicatives qu'ils supportent. Le manque de maturité qui les caractérise est même à l'origine d'un véritable antagonisme entre l'étendue du spectre des applications qu'ils prennent en charge et le degré d'automatisation avec lequel ils en gèrent les différentes phases du cycle de vie.

8.2 Rappel des contributions

L'un des objectifs majeurs de cette thèse a consisté à démontrer que l'antagonisme caractérisant les solutions de *PaaS* actuelles, portant sur le lien entre la largeur du domaine applicatif couvert et le niveau d'automatisation, n'avait pas lieu d'être. Ainsi, les travaux présentés proposent une solution, appelée VAMP, capable de procéder au déploiement initial de bout en bout de n'importe quelle application virtualisable dans une infrastructure de type informatique dans le nuage, de façon autonome et fiable afin de ne pas avoir recours à une intervention humaine (i.e. aide extérieure). Ainsi VAMP définit :

- un modèle permettant de capturer l'ensemble des informations de configuration (i.e. architecture applicative avec les dépendances de configuration, d'activation et de localisation, les machines virtuelles qui composent l'application considérée, le contenu des images associées) nécessaires à la mise en œuvre du déploiement. Grâce à la généricité des principes qu'il développe, ce modèle à base de composants (i.e. le modèle Fractal) permet de décrire n'importe quelle application répartie et n'impose aucune restriction quant à la granularité des entités manipulées. Ainsi chaque composant présente une abstraction unifiée et de haut niveau de(s) élément(s) logiciel(s) qu'il encapsule. Enfin, il est indépendant de toute implémentation ce qui lui confère une forte indépendance vis-à-vis de l'environnement d'exécution applicatif.

- un protocole réparti, asynchrone et fiable d'autoconfiguration et d'autoactivation des applications à déployer. Il constitue le cœur de la solution et présente les caractéristiques suivantes :

  Répartition : le protocole est réparti au sein des configurateurs instanciés sur l'ensemble des machines virtuelles applicatives qui composent l'application. Sa mise en œuvre est le fruit d'interactions entre ces configurateurs en vue de procéder à la configuration globale de l'application puis à son activation. Grâce à son caractère réparti, le protocole limite le recours à une entité centralisée synonyme de maillon faible (*Single Point Of Failure* ou *SPOF*) de la solution.

  Asynchronisme : le modèle de communication entre les configurateurs en particulier, et plus généralement entre les entités qui composent le système VAMP, s'appuie sur des échanges asynchrones et fiables à base de messages. Ainsi, chaque configurateur s'exécute indépendamment des autres et progresse dans la mise en œuvre du protocole au gré de l'arrivée de nouveaux messages en provenance d'autres configurateurs. Le caractère fiable du support de communication garantit qu'aucun message ne peut être perdu une fois qu'il a été envoyé par un émetteur (sauf, éventuellement, en cas de défaillance définitive de l'émetteur) et qu'il sera bien reçu par son destinataire. Ce modèle de communication permet de s'abstraire des mécanismes d'attente active, coûteux et difficiles à fiabiliser. En outre, la persistance des messages contribue à la fiabilisation globale de la solution.

Autonomie : chaque configurateur dispose de l'extrait du modèle d'application nécessaire à la mise en œuvre autonome du protocole. Ainsi, il connaît ses voisins, c'est-à-dire l'ensemble des configurateurs avec lesquels il devra interagir pour configurer les liaisons distantes qui impliquent des composants dont il a la charge puis pour les activer de manière ordonnée.

Fiabilité : enfin, chaque configurateur est capable d'adapter son comportement (i.e. réémission ou prise en compte de nouvelles données de configuration ou d'ordre d'activation), lors de la survenue de défaillances affectant des composants applicatifs distants.

- une approche modulaire permettant d'allouer à une application donnée un gestionnaire de déploiement fiable qui lui est dédié. Ainsi, l'isolation qui caractérise une application virtualisée est étendue à l'ensemble des entités qui interviennent dans la mise en œuvre de son déploiement. Grâce à l'indépendance de chaque environnement de déploiement (i.e. gestionnaire de déploiement et configurateurs) il est possible de procéder à des déploiements multiples simultanés.

Au-delà de ces apports, ces travaux de thèse se sont également accompagnés d'une validation de l'implémentation de référence en Java proposée pour VAMP. Elle a permis de souligner le caractère générique (i.e. polyvalence, indépendance vis-à-vis de l'environnement d'exécution et faible adhérence applicative) de VAMP. De plus, elle a caractérisée le surcoût induit par VAMP et a démontré son efficacité dans le cadre de déploiements multiples.

8.3 Perspectives

Le travail présenté dans cette thèse a permis de démontrer que l'étendue du spectre applicatif et le degré d'autonomie qui caractérisent une solution de déploiement d'applications dans le nuage, n'ont rien d'antagonistes. La plate-forme VAMP en est l'illustration, grâce à son modèle définissant un niveau d'abstraction élevé et des comportements autonomes, génériques, fiables et décentralisés.

Au terme de cette première étape, ce sont désormais de nombreuses perspectives de recherche qui surgissent, en lien avec la phase d'exécution des applications. Les plus intéressantes concernent la gestion de l'élasticité, la fiabilisation applicative et surtout la gestion d'applications réparties sur plusieurs environnements d'*IaaS*

L'objet de cette section est donc de détailler, à partir des limites actuelles de VAMP, ces futurs axes de recherche .

8.3.1 Outillage

Dans sa version courante, VAMP ne propose aucun outillage particulier. Afin d'améliorer sa diffusion et son utilisation, il serait intéressant de fournir des outils d'aide à la conception de modèles des applications ainsi que des bibliothèques de *wrappers* prédéfinis et couramment utilisés (i.e. *wrapper* d'un système de gestion de base de données MySQL,

d'un serveur HTTP Apache, etc.). De plus, afin de faire face au manque de transparence qui résulte de son comportement autonome et de faciliter son adoption, VAMP devrait proposer un utilitaire capable de dérouler le processus de déploiement en pas à pas (i.e. nécessitant une confirmation de l'administrateur au terme de chaque étape).

8.3.2 Prise en charge d'autres phases du cycle de vie de l'application

Conformément au cadre fixé durant ce doctorat, l'implémentation courante de VAMP automatise uniquement l'ensemble des phases du cycle de vie relatives au déploiement initial de l'application (i.e. mise à disposition, installation et activation). L'une des perspectives de ces travaux pourrait donc consister à étendre les principes développés dans cette thèse aux phases d'arrêt, de reconfiguration, d'adaptation, de mise à jour et de retrait[1]. Ceci passe notamment par la définition :

- de l'ordre selon lequel les entités logicielles qui composent l'application doivent être arrêtées. Ceci passe par l'expression de dépendances de désactivation entre composants applicatifs, cet ordre n'étant pas nécessairement le symétrique (i.e. l'inverse) de l'ordre de démarrage. Ainsi, dans le cas du système de tests en charge Clif (cf. section 7.1.2), l'entité de supervision, pourtant démarrée avant les serveurs Clif, est également arrêtée la première.

- du comportement du composant à arrêter vis-à-vis des requêtes utilisateurs en cours de traitement.

- du procédé de mise à jour d'un paquetage logiciel au sein d'une machine virtuelle tout en offrant la capacité de vérifier manuellement la prise en compte effective de cette modification dans l'environnement d'exécution applicatif.

8.3.3 Extension de la fiabilité

VAMP permet de fiabiliser le déploiement d'une application vis-à-vis des pannes franches qui affectent l'environnement d'exécution de cette dernière ou le système de déploiement lui-même. Néanmoins, cette fonctionnalité pourrait être améliorée selon les deux axes décrits dans cette section.

8.3.3.1 Fiabilisation de VAMP

L'implémentation courante de VAMP propose deux mécanismes en vue de la fiabilisation du système de déploiement lui-même :

- Le premier concerne l'autoréparation des configurateurs défaillants (cf. section 6.2).

[1] Le retrait englobe notamment les aspects relatifs à la désinstallation et à la réversibilité de l'installation. La réversibilité désigne la garantie pour l'utilisateur, qu'une fois son application désinstallée, aucune donnée ni aucun binaire relatif à cette dernière ne fait l'objet de conservation, intentionnelle ou non, de la part des acteurs impliqués dans la mise en œuvre et l'exploitation de l'application.

- Le second porte sur la tolérance aux pannes du gestionnaire d'application. Son principe s'appuie sur la réplication du gestionnaire au sein de multiples instances capables de se surveiller mutuellement et de faire face à la défection de l'une d'elles (cf. section 6.3.2).

Bien que le fonctionnement du premier de ces deux mécanismes ait fait l'objet d'une phase d'évaluation qualitative, le comportement de l'un et de l'autre doivent encore faire l'objet d'une validation plus poussée.

8.3.3.2 Fiabilité à l'exécution

Les contributions de ce doctorat ne portant que sur la phase de déploiement initiale, les aspects relatifs à la fiabilité ne concernaient pas l'exécution des applications. Néanmoins, une fois qu'une application a été démarrée, il devient nécessaire de rendre son comportement et celui de son environnement tolérants aux défaillances auxquels ils peuvent être confrontés.

Concernant la fiabilisation de l'environnement d'exécution, elle est déjà couverte par le mécanisme proposé par VAMP dans le cas d'une application ne gérant pas d'état partagé entre ses composants. Il serait cependant intéressant, dans un souci d'élargissement de la polyvalence de VAMP, d'étendre ce mécanisme au support des états applicatifs. Ceci pourrait passer par la sauvegarde périodique de l'état de l'application puis par sa restauration en cas de panne.

Pour ce qui est de la fiabilisation applicative, il s'agit de la capacité à faire face aux défaillances qui surviennent au sein même de l'application. Sa mise en œuvre au sein d'un système d'administration externe à l'application passe donc, dans un premier temps, par la définition des exigences auxquelles celle-ci doit se conformer : implémentation de comportements de type *arrêt sur défaillance* (*fail stop*), fourniture de sondes d'observation (monitoring), etc.

8.3.4 Support multi-nuages

Une application multi-nuages désigne une application dont les composants sont répartis au sein de machines virtuelles instanciées sur un ensemble de plateformes d'*IaaS* distinctes. Loin d'être des cas d'école, de telles applications ont vocation à se développer pour de multiples raisons :

Contraintes de performances et obligations légales : pour offrir de meilleurs performances, il peut être préférable de rapprocher géographiquement les traitements des utilisateurs finaux. A l'inverse, certaines lois imposent que les données d'une entreprise demeurent stockées sur le territoire national ;

Débordement (*cloud bursting*) : dans le cadre du redimensionnement d'une application (e.g. par mise en œuvre de mécanisme d'adaptation à la charge), plus aucune ressource n'est disponible sur la plate-forme d'*IaaS* courante. Les éléments qui composent l'application se voient donc répartis sur plusieurs nuages ;

Haute disponibilité : pour se prémunir des défaillances du niveau *IaaS*, une application intègre des mécanismes de réplication répartis sur plusieurs nuages.

Bien que des travaux préliminaires s'appuyant sur la solution d'intégration multi-*IaaS* Sirocco aient débuté en vue de déployer de telles applications, la version de VAMP présentée dans ce manuscrit ne prend en charge que des applications déployées sur un seul *cluster* d'une solution d'IaaS donnée. La notion de *cluster* désigne un regroupement de machines interconnectées au sein d'un même réseau.

Les problématiques relatives à l'administration des applications multi-nuages sont extrêmement nombreuses et vont de la gestion de formats d'images multiples, à la définition de la connectivité réseau entre machines virtuelles applicatives en passant par l'hétérogénéité des fonctionnalités et des interfaces proposées par les plates-formes d'*IaaS*.

8.3.5 Dimensionnement autonome

La finalité du dimensionnement autonome est de maintenir à tout instant un niveau de qualité de service défini comme acceptable par l'utilisateur, tout en réduisant le surcoût induit par le gaspillage de ressources sous-utilisées. Il s'agit d'adapter dynamiquement la structure de l'application à la charge à laquelle cette dernière est soumise. Cette adaptation porte donc, d'une part, sur la nature et le nombre des éléments logiciels applicatifs mis en œuvre, d'autre part, sur leur organisation au sein de l'architecture applicative.

La mise en œuvre du dimensionnement autonome de l'application passera donc par la définition d'un modèle permettant de définir l'espace des possibilités dans lequel l'architecture applicative pourra évoluer. Il conviendra alors de recourir à un module de projection en charge de transformer ce modèle par intention (i.e. modèle implicite) sur un modèle par extension (i.e. explicite) utilisable par un système d'administration autonome d'applications. D'ores et déjà, la version courante de VAMP peut servir de système de déploiement dans le cadre du dimensionnement initial. Dans le cadre de la gestion de l'élasticité (i.e. dimensionnement autonome lors de l'exécution de l'application), VAMP devra être enrichi d'opérations élémentaires (i.e. ajouts/retraits de composants ou de liaisons) afin d'assurer la transition d'une architecture applicative vers une autre.

Pour finir, de façon beaucoup plus générale à la seule gestion autonome du dimensionnement de l'application, VAMP permet grâce à son approche autonome, générique, fiable et large échelle d'envisager la définition de nouvelles fonctionnalités d'administration basées sur les quatre composantes de l'informatique autonome (i.e. auto-configuration, auto-réparation, auto-protection et auto-optimisation).

Bibliographie

[1] Cloud Foundry website. http://www.cloudfoundry.com/.

[2] Grid Component Model (GCM) specification version 1.1.1. http://www.etsi.org/deliver/etsi_ts/102800_102899/102830/01.01.01_60/ts_102830v010101p.pdf.

[3] Java 2 Enterprise Edition website. http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[4] Java Management eXtensions (JMX) specification version 1.4. http://download.oracle.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf.

[5] The Java Messaging Service (JMS) website. http://www.oracle.com/technetwork/java/jms/index.html.

[6] The Java Open Reliable Asynchronous Messaging (JORAM) website. http://joram.ow2.org/.

[7] The Julia project website. http://fractal.ow2.org/julia/index.html.

[8] Opennebula : The cloud data center management solution. http://opennebula.org.

[9] Preboot eXcution Environment (PXE) Specification version 2.1. http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf.

[10] rPath website. https://www.rpath.com/.

[11] Amazon.com CEO Jeff Bezos on Animoto, April 2008. http://blog.animoto.com/2008/04/21/amazon-ceo-jeff-bezos-on-animoto/.

[12] Amazon Web Services. AWS Cloud Formation. http://aws.amazon.com/fr/cloudformation/.

[13] Amazon Web Services. AWS Elastic Beanstalk. http://aws.amazon.com/fr/elasticbeanstalk/.

[14] Amazon Web Services. *Amazon Simple Storage Service Developer Guide*, March 2006. `http://awsdocs.s3.amazonaws.com/S3/latest/s3-dg.pdf`.

[15] Amazon Web Services. *Amazon Elastic Compute Cloud User Guide*, November 2010. `http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf`.

[16] Brian Amedro, Françoise Baude, Fabrice Huet, and Elton N. Mathias. Combining grid and cloud resources by use of middleware for spmd applications. In *CloudCom*, pages 177–184. IEEE, 2010.

[17] Paul Anderson, Patrick Goldsack, and Jim Paterson. Smartfrog meets lcfg : Autonomous reconfiguration with central policy control. In *LISA*, pages 213–222. USENIX, 2003.

[18] Paul Anderson, Alastair Scobie, and Division Of. Lcfg - the next generation. In *In UKUUG Winter Conference. UKUUG*, page 2002, 2002.

[19] APSstandard.org. APS Standard. `http://apsstandard.org/`.

[20] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[21] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. *TECHNICAL REPORT SERIESUNIVERSITY OF NEWCASTLE UPON TYNE COMPUTING SCIENCE*, 1145(010028) :7–12, 2001.

[22] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1) :11–33, 2004.

[23] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer-Verlag, 2005.

[24] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A case for message oriented middleware. In Prasad Jayanti, editor, *DISC*, volume 1693 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1999.

[25] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm : a grid extension to fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2) :5–24, 2009.

[26] Thomas Becker. Application-transparent fault tolerance in distributed systems. In *CDS*, pages 36–45, 1994.

[27] Luc Bellissard, Noel De Palma, André Freyssinet, M. Herrmann, and Serge Lacourte. An agent platform for reliable asynchronous distributed programming. In *SRDS*, pages 294–295, 1999.

[28] Kenneth P. Birman. Replication and fault-tolerance in the isis system. In *SOSP*, pages 79–86, 1985.

[29] Gordon S. Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based architecture : the fractal initiative. *Annales des Télécommunications*, 64(1-2) :1–4, 2009.

[30] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5) :61–72, 1988.

[31] Raphael Bolze, Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quétier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000 : A large scale and highly reconfigurable experimental grid testbed. *IJHPCA*, 20(4) :481–494, 2006.

[32] Sara Bouchenak, Fabienne Boyer, Sacha Krakowiak, Daniel Hagimont, Adrian Mos, Jean-Bernard Stefani, Noel De Palma, and Vivien Quéma. Architecture-based autonomous repair management : An application to j2ee clusters. In *SRDS*, pages 13–24. IEEE Computer Society, 2005.

[33] Sara Bouchenak, Noel De Palma, Daniel Hagimont, and Christophe Taton. Autonomic management of clustered applications. In *CLUSTER*. IEEE, 2006.

[34] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel De Palma, and Suzy Temate. Autonomic management policy specification in tune. In Roger L. Wainwright and Hisham Haddad, editors, *SAC*, pages 1658–1663. ACM, 2008.

[35] Alan W. Brown. An overview of components and component-based development. *Advances in Computers*, 54 :1–34, 2001.

[36] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[37] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal Component Model v2.0.3, February 2004. `http://fractal.ow2.org/specification/index.html`.

[38] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In Adrian Segall and Shmuel Zaks, editors, *WDAG*, volume 647 of *Lecture Notes in Computer Science*, pages 362–378. Springer, 1992.

[39] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. A microrebootable system – design, implementation, and evaluation. *CoRR*, cs.OS/0406005, 2004.

[40] Eddy Caron and Frédéric Desprez. Diet : A scalable toolbox to build network enabled servers on the grid. *IJHPCA*, 20(3) :335–352, 2006.

[41] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service : Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, August 1998.

[42] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society.

[43] CFEngine AS. CFEngine - Open source configuration management software. `http://www.cfengine.org/`.

[44] Shang-Wen Cheng. *Rainbow : Cost-Effective Software Achitecture-Based Self-Adaptation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 2008. `http://reports-archive.adm.cs.cmu.edu/anon/isr2008/CMU-ISR-08-113.pdf`.

[45] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. In *ICAC*, pages 276–277. IEEE Computer Society, 2004.

[46] Marc Chérèque, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. Active replication in delta-4. In *FTCS*, pages 28–37. IEEE Computer Society, 1992.

[47] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, and Alla Segal. A cloud provisioning system for deploying complex application services. *E-Business Engineering, IEEE International Conference on*, 0 :125–131, 2010.

[48] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Richard Wolski. Appscale : Scalable and open appengine application development and deployment. In Dimiter R. Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, and Eliezer Dekel, editors, *CloudComp*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 57–70. Springer, 2009.

[49] OW2 Consortium. JOnAS application server. `http://jonas.ow2.org/`.

[50] OW2 Consortium. Sirocco VAMP Deployment Manager. `http://wiki.sirocco.ow2.org/xwiki/bin/view/Components/Sirocco%2BVAMP%2Bdeployment%2Bmanager`.

[51] OW2 Consortium. Sirocco web site. `http://ow2.org/view/ActivitiesDashboard/Sirocco`.

[52] Thierry Coupaye and Jacky Estublier. Foundations of enterprise software deployment. In *CSMR*, pages 65–74, 2000.

[53] Jan de Meer. The iso reference model for open distributed processing. *Computer Networks and ISDN Systems*, 27(8) :1211–1214, 1995.

[54] Anne-Marie Déplanche, Pierre-Yves Théaudière, and Yvon Trinquet. Implementing a semi-active replication strategy in chorus/classix, a distributed real-time executive. In *SRDS*, pages 90–101, 1999.

[55] Narayan Desai, Andrew Lusk, Rick Bradshaw, and Rémy Evard. Bcfg : A configuration management tool for heterogeneous environments. In *CLUSTER*, pages 500–503. IEEE Computer Society, 2003.

[56] Bruno Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing. *Annales des Télécommunications*, 64(1-2) :101–120, 2009.

[57] Distributed Management Task Force. The DMTF home page. `http://www.dmtf.org/`.

[58] DMTF. CIMI Work-in-Progress Specifications Now Available for Public Comment. `http://dmtf.org/content/cimi-work-progress-specifications-now-available-public-comment`.

[59] DMTF. Open Virtualization Format Specification. Dmtf standard, Distributed Management Task Force, 2009.

[60] A. Eastwood. Firm fires shots at legacy systems. *Computing Canada*, 19(2) :17, 1993.

[61] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3) :375–408, 2002.

[62] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3) :17–23, 2000.

[63] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Autoconfiguration d'applications réparties dans le nuage. In *Conférence Française en Systèmes d'Exploitation (CFSE'8)*, May 2011.

[64] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Self-configuration of distributed applications in the cloud. In Liu and Parashar [88], pages 668–675.

[65] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, Noel De Palma, and Gwen Salaün. Automated configuration of legacy applications in the cloud. In *UCC*, pages 170–177. IEEE Computer Society, 2011.

[66] Inc. Eucalyptus Systems. Eucalyptus Community Cloud. `http://www.eucalyptus.com/eucalyptus-cloud/community-cloud`.

[67] Kazi Farooqui, Luigi Logrippo, and Jan de Meer. The iso reference model for open distributed processing : An introduction. *Computer Networks and ISDN Systems*, 27(8) :1215–1229, 1995.

[68] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage : a deployment management system. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *PLDI*, pages 263–274. ACM, 2012.

[69] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, 1985.

[70] Areski Flissi and Philippe Merle. A generic deployment framework for grid computing and distributed applications. *CoRR*, abs/0706.3008, 2007.

[71] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010 : A toolbox for the construction and analysis of distributed processes. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.

[72] David Garlan, Robert T. Monroe, and David Wile. Acme : an architecture description interchange language. In J. Howard Johnson, editor, *CASCON*, page 7. IBM, 1997.

[73] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43 :16–25, January 2009.

[74] Google. Google App Engine website. `http://code.google.com/appengine/`.

[75] Google. Google Docs website. `http://docs.google.com/`.

[76] Richard S. Hall. A policy-driven class loader to support deployment in extensible frameworks. In Wolfgang Emmerich and Alexander L. Wolf, editors, *Component Deployment*, volume 3083 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2004.

[77] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Entropy : a consolidation manager for clusters. In Antony L. Hosking, David F. Bacon, and Orran Krieger, editors, *VEE*, pages 41–50. ACM, 2009.

[78] Paul Horn. Autonomic computing : Ibm's perspective on the state of information technology, 2001. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`.

[79] IBM. An architectural blueprint for autonomic computing, white paper, fourth edition, June 2006. `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`.

[80] Keith Jeffery and Burkhard Neidecker-Lutz, editors. *The Future Of Cloud Computing, Opportunities for European Cloud Computing Beyond 2010*. EUROPA > CORDIS > FP7, January 2010.

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

[82] Johannes Kirschnick, Jose M. Alcaraz Calero, Patrick Goldsack, Andrew Farrell, Julio Guijarro, Steve Loughran, Nigel Edwards, and Lawrence Wilcock. Towards an architecture for deploying elastic services in the cloud. *Softw., Pract. Exper.*, 42(4) :395–408, 2012.

[83] Andreas Kluth. Make it simple. *The Economist*, 28 October 2004.

[84] Orran Krieger, Phil McGachey, and Arkady Kanevsky. Enabling a marketplace of clouds : Vmware's vcloud director. *Operating Systems Review*, 44(4) :103–114, 2010.

[85] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2 :95–114, 1978.

[86] Jean-Claude Laprie. Dependable computing : Concepts, challenges, directions. In *COMPSAC*, page 242. IEEE Computer Society, 2004.

[87] Jean-Claude Laprie, Bernard Courtois, Marie-Claude Gaudel, and David Powell. *Sûreté de fonctionnement des systèmes informatiques : matériels et logiciels*. Dunod, 1989.

[88] Ling Liu and Manish Parashar, editors. *IEEE International Conference on Cloud Computing, CLOUD 2011, Washington, DC, USA, 4-9 July, 2011*. IEEE, 2011.

[89] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. *IEEE Transactions on Software Engineering*, 26 :70–93, January 2000.

[90] Peter Mell and Tim Grance. Cloud Computing Definition, June 2009. `http://csrc.nist.gov/groups/SNS/cloud-computing/index.html`.

[91] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344, 2005.

[92] Microsoft. Windows Azure : Microsoft's cloud services platform. `http://www.microsoft.com/windowsazure/`.

[93] J. Moad. Maintaining the competitive edge. *Datamation*, 4(36) :61–62, 1990.

[94] Gordon E. Moore. *Progress in Digital Integrated Electronics*, pages 11–13. 1975.

[95] Victor P. Nelson. Fault-tolerant computing : Fundamental concepts. *IEEE Computer*, 23(7) :19–25, 1990.

[96] Thomas Damgaard Nielsen, Christian Iversen, and Philippe Bonnet. Private cloud configuration with metaconfig. In Liu and Parashar [88], pages 508–515.

[97] Novell. SUSE Studio website. `https://susestudio.com/`.

[98] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In Franck Cappello, Cho-Li Wang, and Rajkumar Buyya, editors, *CCGRID*, pages 124–131. IEEE Computer Society, 2009.

[99] OMG. Unified Modeling Language (UML) superstructure specification. `http://www.omg.org/spec/UML/`.

[100] OMG. Deployment and configuration of component-based distributed applications specification, April 2006. `http://www.omg.org/spec/DEPL/`.

[101] Opscode. Chef. `http://wiki.opscode.com/display/chef/Home`.

[102] OW2 Consortium. The Fractal ADL website. `http://fractal.ow2.org/fractaladl/`.

[103] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing : State of the art and research challenges. *IEEE Computer*, 40(11) :38–45, 2007.

[104] Puppet Labs. Puppet Labs Documentation. `http://docs.puppetlabs.com/`.

[105] Quattor.org. Quattor - fabric management for grids and cloud. `http://quattor.web.cern.ch/quattor/`.

[106] Rackspace. OpenStack : Open source software for building private and public clouds. `http://www.openstack.org`.

[107] Jennifer Ren, Paul Rubel, Mouna Seri, Michel Cukier, William H. Sanders, and Tod Courtney. Passive replication schemes in aqua. In *PRDC*, pages 125–130. IEEE Computer Society, 2002.

[108] Gwen Salaün, Xavier Etchevers, Noel De Palma, Fabienne Boyer, and Thierry Coupaye. Verification of a self-configuration protocol for distributed applications in the cloud. In Sascha Ossowski and Paola Lecca, editors, *SAC*, pages 1278–1283. ACM, 2012.

[109] Gwen Salaün, Xavier Etchevers, Noel De Palma, Fabienne Boyer, and Thierry Coupaye. *Assurances for Self-Adaptive Systems (ASAS)*, chapter Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. -, to appear.

[110] Salesforce.com. Salesforce.com website. `http://www.salesforce.com/`.

[111] Marcus Schäfer, Adrian Schröter, and Robert Schweikert. openSUSE Kiwi. `https://kiwi.berlios.de/`.

[112] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Comput. Surv.*, 22(4) :299–319, 1990.

[113] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[114] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. In *IEEE SCC*, pages 268–275. IEEE Computer Society, 2009.

[115] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Softw., Pract. Exper.*, 42(5) :559–583, 2012.

[116] Quest Software. Vizioncore vConverter v5.1.1, March 2011. `http://us-downloads.quest.com/Repository/support.quest.com/vConverter/5.1.1/Documentation/vConverter%205.1.1%20Release%20Notes.pdf`.

[117] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, and Ed Merks. Emf : Eclipse modeling framework. 2009.

[118] Citrix Systems. Citrix XenConvert Guide v2.1.1, April 2010. `http://support.citrix.com/servlet/KbServlet/download/20644-102-332133/XenConvertGuide.pdf`.

[119] Clemens A. Szyperski. Component technology - what, where, and how ? In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *ICSE*, pages 684–693. IEEE Computer Society, 2003.

[120] The OSGi Alliance. OSGi Service Platform Core Specification, Release 5, 2012. `http://www.osgi.org/Specifications/HomePage`.

[121] B. Topol, D. Ogle, D. Pierson, J. Thoensen, J. Sweitzer, M. Chow, M. A. Hoffmann, P. Durham, R. Telford, S. Sheth, and T. Studwell. Automating problem determination : A first step toward self-healing computing systems. Technical report, IBM, 2003.

[122] University of Chicago. Nimbus is cloud computing for science. `http://www.nimbusproject.org/`.

[123] UShareSoft. UForge cloud software factory and app store.

[124] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : An annotated bibliography. *SIGPLAN Notices*, 35(6) :26–36, 2000.

[125] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds : towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1) :50–55, January 2009.

[126] VMWare. Virtual machine disk (vmdk) format specifcation v5.0. `http://www.vmware.com/support/developer/vddk/vmdk_50_technote.pdf?src=vmdk`.

[127] VMWare. VMWare vCenter Converter, convert physical machines to virtual machines. `http://www.vmware.com/products/converter/`.

[128] Inc. VMWare. VMWare vFabric Suite. `https://www.vmware.com/products/application-platform/vfabric/overview.html`.

[129] Matthew S. Wilson. Constructing and managing appliances for cloud deployments from repositories of reusable components. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[130] Wojciech Zamojski and Dariusz Caban. Introduction to the dependability modeling of computer systems. In *DepCoS-RELCOMEX*, pages 100–109. IEEE Computer Society, 2006.