Système d'Administration Autonome Adaptable: application au Cloud

Alain Tchana

▶ To cite this version:

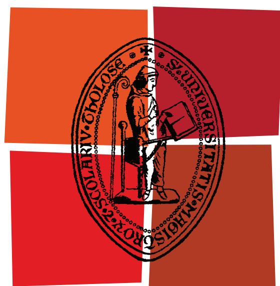
Alain Tchana. Système d'Administration Autonome Adaptable: application au Cloud. Autre [cs.OH]. Institut National Polytechnique de Toulouse - INPT, 2011. Français. NNT : . tel-04240340v1

HAL Id: tel-04240340
https://theses.hal.science/tel-04240340v1

Submitted on 6 Dec 2011 (v1), last revised 13 Oct 2023 (v2)

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.



THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Informatique

Présentée et soutenue par :
Alain-B. TCHANA

le : mardi 29 novembre 2011

Titre :

Système d'Administration Autonome Adaptable: application au Cloud

---

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :
Mr. Daniel HAGIMONT
Mr. Laurent BROTO

Rapporteurs :

Mr. Noël DE PALMA
Mr. Jean-Marc MENAUD

Membre(s) du jury :
Mr. Noël DE PALMA, Université Joseph Fourier, Rapporteur
Mr. Jean-Marc MENAUD, Ecole des Mines de Nantes, Rapporteur
Mr. Michel DAYDE, Institut National Polytechnique de Toulouse, Examinateur
Mr. Maurice TCHUENTE, Université de Yaoundé I, Examinateur
Mr. Daniel HAGIMONT, Institut National Polytechnique de Toulouse, Directeur de Thèse
Mr. Laurent BROTO, Institut National Polytechnique de Toulouse, Co-Directeur de Thèse

THÈSE

soutenue en vue de l'obtention du titre de

DOCTEUR DE L'UNIVERSITÉ DE TOULOUSE

Spécialité : **INFORMATIQUE**

délivrée par

l'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Système d'administration autonome adaptable : application au Cloud

---

Présentée et soutenue publiquement par

Alain-B. TCHANA

le 29/11/2011, devant le jury composé de :

Mr. Jean-Marc MENAUD	École des Mines de Nantes	Rapporteur
Mr. Noel DE PALMA	Université Joseph Fourier	Rapporteur
Mr. Maurice TCHUENTE	Université de Yaoundé I	Examinateur
Mr. Michel DAYDE	Institut National Polytechnique de Toulouse	Examinateur
Mr. Laurent BROTO	Institut National Polytechnique de Toulouse	Co-Directeur de thès
Mr. Daniel HAGIMONT	Institut National Polytechnique de Toulouse	Directeur de thèse

École doctorale MITT : Mathématique, Informatique, Télécommunication de Toulouse
Directeur de thèse : Daniel HAGIMONT(IRIT, ENSEEIHT)
Laboratoire : Institut de Recherche en Informatique de Toulouse - UMR 5505-IRIT/ENSEEIHT
CNRS - INP - UPS - UT1 - UTM

118 Route de Narbonne, 31062 Toulouse Cedex 09. Tel : 05.61.55.67.65

A Hélène Maréchau. Je ne trouverai jamais assez de mots, ni d'espace, ni de temps pour te dire à quel point ton aide m'a été capitale Hélène. J'ai toujours imaginé plus jeune (au secondaire) que toute thèse aboutissait sur une découverte ou une invention. Ta rencontre sera ma plus grande trouvaille de ces dernières années et j'espère quelle le restera.

Hommage au système éducatif Français pour m'avoir accueilli et financé ma thèse.
***Alain TCHANA**.*

Je tiens initialement à remercier tous les membres du jury pour le temps que vous avez consacré à l'évaluation de mon travail. Merci à mes rapporteurs Noël De PALMA et Jean-Marc MENAUD à qui a été confiée la responsabilité de critiquer la forme et le fond de mon travail. Merci à Michel DAYDE qui a accepté de présider le jury et d'examiner mon travail. Merci à Maurice TCHUENTE qui m'a fait l'honneur du déplacement du Cameroun, lieu où j'ai acquis les bases des connaissances mises à profit dans cette thèse.

Merci à mon directeur de thèse Daniel HAGIMONT et co-Directeur Laurent BROTO. Vous m'avez mis dans les conditions idéales pour réaliser cette thèse, tant sur le plan professionnel que personnel. Je dois à Daniel (le lieutenant Dan) mes modestes qualités rédactionnelles, d'esprit critique et de recul dans la réalisation d'un travail de recherche. Laurent, tu as été une des premières personnes qui m'a impressionné techniquement dans les domaines de l'informatique et de la mécanique. Tu as réussi à me transmettre ta sérénité face à tout problème et système informatique. En dehors du laboratoire, vous avez été tous les deux ma seconde famille. Je vous ai souvent présenté à la fois comme encadreurs, tuteurs et potes. Ce fut un bonheur de travailler avec vous et j'espère continuer.

Un grand merci à nos admirables secrétaires Sylvie ARMENGAUD et Sylvie EICHEN. Vous êtes exceptionnelles dans votre travail et vous avez contribué à faciliter le mien.

Je remercie la grande équipe ASTRE, multi-culturelle et très animée dans le travail. Merci au "professeur" TOURE Mohamed qui m'a accueilli et traité comme un frère durant son séjour dans cette équipe. Merci à Suzy TEMATE qui apportait toujours la lumière et la fraîcheur dans le bureau. Un merci à Aeman GADAFI avec qui j'ai passé de longues nuits de travail au laboratoire amenant son épouse a se demander qui est ce Alain. Merci à Raymond ESSOMBA, Larissa MAYAP et Tran SON avec qui j'ai terminé dans une bonne ambiance mon séjour au sein de l'équipe.

Je remercie les membres du grain : le "moussokologue" Amadou BAGAYOKO qui a toujours laissé sa porte ouverte à toute heure ; le "jeune" Bafing SAMBOU et Mekossou BAKAYOKO pour votre amitié et disponibilité ; à Cheik OUMAR AIDARA, Aboubakar DIALLO et Cheik TALL pour vos débats houleux et enrichissants les soirs de foot. Tâchez toujours de faire respecter les fondamentaux.

Je ne saurai oublier ma très grande famille qui m'a toujours soutenu dans tous mes choix. Vous constituez une grande partie de mes motivations et j'espère toujours vous faire honneur. Un merci particulier à mon père et ma mère qui ont toujours mis l'école au centre de tout. Merci à la "1759" qui représente un membre permanent de la famille auquel je peux faire appel à tout moment :).

Ces dernières années ont vu le développement du cloud computing. Le principe fondateur est de déporter la gestion des services informatiques des entreprises dans des centres d'hébergement gérés par des entreprises tiers. Ce déport a pour principal avantage une réduction des coûts pour l'entreprise cliente, les moyens nécessaires à la gestion de ces services étant mutualisés entre clients et gérés par l'entreprise hébergeant ces services. Cette évolution implique la gestion de structures d'hébergement à grande échelle, que la dimension et la complexité rendent difficiles à administrer.

Avec le développement des infrastructures de calcul de type cluster ou grilles ont émergé des systèmes fournissant un support pour l'administration automatisée de ces environnements. Ces systèmes sont désignés sous le terme Systèmes d'Administration Autonome (SAA). Ils visent à fournir des services permettant d'automatiser les tâches d'administration comme le déploiement des logiciels, la réparation en cas de panne ou leur dimensionnement dynamique en fonction de la charge.

Ainsi, il est naturel d'envisager l'utilisation des SAA pour l'administration d'une infrastructure d'hébergement de type cloud. Cependant, nous remarquons que les SAA disponibles à l'heure actuelle ont été pour la plupart conçus pour répondre aux besoins d'un domaine applicatif particulier. Un SAA doit pouvoir être adapté en fonction du domaine considéré, en particulier celui de l'administration d'un cloud. De plus, dans le domaine du cloud, différents besoins doivent être pris en compte : ceux de l'administrateur du centre d'hébergement et ceux de l'utilisateur du centre d'hébergement qui déploie ses applications dans le cloud. Ceci implique qu'un SAA doit pouvoir être adapté pour répondre à ces besoins divers.

Dans cette thèse, nous étudions la conception et l'implantation d'un SAA adaptable. Un tel SAA doit permettre d'adapter les services qu'il offre aux besoins des domaines dans lesquels il est utilisé. Nous montrons ensuite comment ce SAA adaptable peut être utilisé pour l'administration autonome d'un environnement de cloud.

Last years have seen the development of cloud computing. The main underlying principle of to externalize the management of companies' IT services in hosting centers which are managed by third party companies. This externalization allows saving costs for the client company, since the resources required to manage these services are mutualized between clients and managed by the hosting company. This orientation implies the management of large scale hosting centers, whose dimension and complexity make them difficult to manage.

With the developement of computing infrastructures such as clusters or grids, researchers investigated the design of systems which provides support of an automatized management of these environments. We refer to these system as Autonomic Management Systems (AMS). They aim at providing services which automate administration tasks such as software deployment, fault repair or dynamic dimensioning according to a load.

Therefore, in this context, it is natural to consider the use of AMS for the administration of a cloud infrastructure. However, we observe that currently available AMS have been designed to address the requirements of a particular application domain. It should be possible to adapt an AMS according to the considered domain, in particular that of the cloud. Moreover, in the cloud computing area, different requirements have to be accounted : those of the administrator of the hosting center and those of the user of the hosting center (who deploys his application in the cloud). Therefore, an AMS should be adaptable to fulfill such various needs.

In this thesis, we investigate the design and implementation of an adaptable AMS. Such an AMS must allow adaptation of all the services it provides, according to the domains where it is used. We next describe the application of this adaptable AMS for the autonomic management of a cloud environment.

Table des matières

1 Introduction 1

2 Le cloud 4

2.1 Le Cloud Computing 5

2.1.1 Généralités et Définition 5

2.1.2 Cloud vs Grilles 6

2.1.3 Principes 7

2.1.4 Bénéfices 8

2.1.5 Classification 10

2.1.5.1 Par raison de développement 10

2.1.5.2 Par niveau de service 11

2.1.6 Challenges 12

2.1.6.1 Isolation 13

2.1.6.2 Administration 14

2.1.6.3 Interopérabilité et Portabilité 14

2.2 Isolation par la virtualisation 16

2.2.1 Définition et Principes 16

2.2.2 Objectifs 17

2.2.3 Bénéfices pour les entreprises 18

2.2.4 Classification 19

2.2.5 Synthèse 20

3 Administration dans le Cloud 22

3.1 Administration niveau IaaS 23

3.1.1 Allocation de ressources 24

3.1.2 Déploiement 24

3.1.3 Configuration et Démarrage 25

3.1.4 Reconfiguration 25

3.1.5 Monitoring 26

3.2 Administration niveau Entreprise 27

3.2.1 Construction de VM et Déploiement 28

3.2.2 Allocation de ressources 29

3.2.3 Configuration et Démarrage 29

3.2.4 Reconfiguration 30

3.2.5 Monitoring . . . 30
3.3 Synthèse : système d'administration autonome pour le cloud . . . 31

4 Administration Autonome **32**
4.1 Définition . . . 32
4.2 Objectifs . . . 33
4.3 Bénéfices pour les entreprises . . . 34
4.4 Classification . . . 35
4.5 Synthèse . . . 36

5 TUNe **37**
5.1 Historique . . . 38
5.2 Principes . . . 39
5.2.1 Architecture Description Language (ADL) . . . 39
5.2.2 Wrapping Description Language (WDL) . . . 41
5.2.3 Reconfiguration Description Language (RDL) . . . 42
5.3 Choix d'implantation . . . 43
5.4 Expérimentations et Problématiques . . . 46
5.4.1 Applications cluster . . . 46
5.4.2 Applications large échelle : cas de DIET . . . 47
5.4.3 Applications virtualisées . . . 48
5.4.4 Cloud . . . 50
5.5 Synthèse et nouvelle orientation . . . 52

6 Orientation générale **54**
6.1 Caractéristiques . . . 55
6.1.1 Uniforme . . . 56
6.1.2 Adaptable . . . 57
6.1.2.1 Adaptation des comportements . . . 57
6.1.2.2 Extensibilité . . . 58
6.1.3 Interopérable et Collaboratif . . . 59
6.1.4 Synthèse . . . 60
6.2 Approche générale et Modèle Architectural . . . 60
6.2.1 Approche Générale . . . 60
6.2.2 Modèle Architectural . . . 61
6.2.3 Prise en compte des caractéristiques . . . 64
6.3 Synthèse . . . 64

7 Etat de l'art **65**
7.1 SAA . . . 66
7.1.1 Rainbow . . . 67
7.1.2 Accord . . . 69
7.1.3 Unity . . . 71

7.1.4 Synthèse 73
7.2 SAA pour le cloud 73
7.2.1 OpenNebula 74
7.2.2 Eucalyptus 77
7.2.3 OpenStack 78
7.2.4 Microsoft Azure 80
7.2.5 Autres plateformes de cloud 83
7.3 Synthèse 83

8 Contributions : TUNeEngine 85
8.1 Modèle détaillé de TUNeEngine 86
8.1.1 Soumission de commandes et Collaboration 87
8.1.2 Construction du SR 87
8.1.3 Déploiement 89
8.1.4 Configuration, Démarrage 90
8.1.5 Réconfiguration 91
8.2 Le formalisme à composants Fractal 92
8.3 Implantation de TUNeEngine 94
8.3.1 Le composant *TUNeEngine* 94
8.3.2 Soumission de commandes et Collaboration 95
8.3.3 Construction du SR 96
8.3.4 Déploiement 98
8.3.5 Exécution de programmes d'administration 100
8.4 Synthèse 102

9 Contributions : application au Cloud 104
9.1 Un IaaS simplifié : CloudEngine 105
9.1.1 Vision globale : CloudEngine-TUNeEngine, Application-TUNeEngine-CloudEngine 107
9.1.2 RepositoryController 108
9.1.3 VMController 111
9.1.4 VLANController 112
9.1.5 ResourceController 112
9.1.6 MonitoringController 113
9.1.7 Scheduler 114
9.2 Utilisation de CloudEngine 114
9.2.1 Construction et enregistrement de VM 115
9.2.2 Administration d'applications 115
9.3 Reconfiguration d'applications et de CloudEngine 116
9.3.1 Réparation de VM 116
9.3.2 Consolidation de VM et Scalabilité de serveurs 117
9.3.2.1 Exécution d'applications statique 118

9.3.2.2 Exécution d'applications variables 118
9.4 Synthèse 119

10 Expérimentations **123**
10.1 Environnements d'expérimentation 124
10.1.1 L'IaaS 124
10.1.2 Les applications entreprises : RUBIS 125
10.1.3 Les métriques 126
10.2 Évaluations 126
10.2.1 Réparation de VM 126
10.2.1.1 Réparation non collaborative 127
10.2.1.2 Réparation collaborative 128
10.2.2 Scalabilité et Consolidation 130
10.2.2.1 Scalabilité 130
10.2.2.2 Consolidation 132
10.2.2.3 Scalabilité et Consolidation simultanées 134
10.3 Synthèse 143

11 Conclusion et perspectives **144**
11.1 Conclusion 144
11.2 Perspectives 146
11.2.1 Perspectives à court terme 146
11.2.2 Perspectives à long terme 148

Chapitre 1

Introduction

Face à l'augmentation continuelle des coûts de mise en place et de maintenance des systèmes informatiques, les entreprises externalisent de plus en plus leurs services informatiques en les confiant à des entreprises spécialisées comme les fournisseurs de clouds. L'intérêt principal de cette stratégie pour les entreprises réside dans le fait qu'elles ne paient que pour les services effectivement consommés. Quant au fournisseur du cloud, son but est de répondre aux besoins des clients en dépensant le minimum de ressources possibles. Une des approches qu'utilise le fournisseur consiste à mutualiser les ressources dont il dispose afin de les partager entre plusieurs entreprises. Dans ce contexte, plusieurs défis, parmi lesquels l'isolation (des ressources, défaillances, performances, espaces utilisateur) et l'administration optimale des ressources, se posent pour permettre un réel développement du cloud.

Actuellement, la plupart des développements de plateformes de cloud utilisent la virtualisation [**?**] comme support technologique pour assurer l'isolation. Ainsi, l'unité d'allocation de ressources dans le cloud est la machine virtuelle, autrement dit une portion de machine physique. Quant à l'administration dans le cloud, nous constatons que les services fournis restent limités et sont plus ou moins semblables d'une plateforme à l'autre. Il s'agit essentiellement des services de réservation, de démarrage et d'arrêt des machines virtuelles. Les tâches d'administration des machines virtuelles et des applications des entreprises sont laissées à la charge des différents administrateurs. A titre d'exemple, les opérations de consolidation de machines virtuelles ou encore de passage à l'échelle des applications clientes doivent être implantées par les administrateurs à travers des API de bas niveau. L'objectif principal de cette thèse est la conception d'un logiciel permettant l'administration du cloud dans son ensemble.

Depuis plusieurs années, des chercheurs se sont intéressés à la conception de systèmes d'administration autonome (SAA). Ces systèmes visent à fournir des services permettant d'automatiser les tâches d'administration comme le déploiement des logiciels, la réparation en cas de panne ou leur dimensionnement dynamique en fonction de la charge. Nous observons un parfait parallèle entre les apports de ces systèmes d'administration autonome et les besoins d'administration dans le cloud.

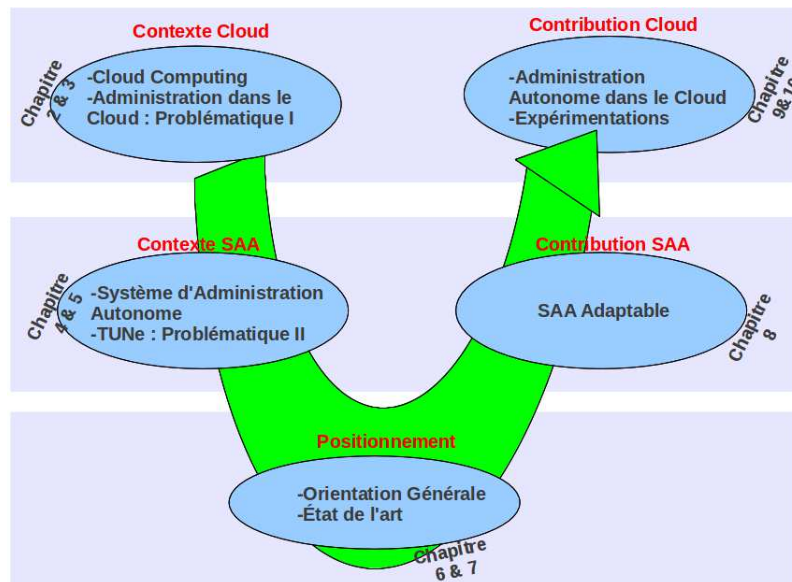


FIGURE 1.1 – Plan en U de ce document de thèse

En effet, dans les deux niveaux d'administration dans le cloud (cloud et applications qu'il exécute), les administrateurs sont confrontés aux opérations d'installation (de machines virtuelles et des logiciels), de configuration, de dépannage, d'allocation de ressources et autres tâches de reconfiguration. La principale différence entre ces deux niveaux est la nature des entités administrées : les machines virtuelles au niveau du cloud (l'hébergeur) et les logiciels applicatifs au niveau du client.

Dans le cadre de nos travaux dans le domaine de l'administration, nous avons conçu et développé un système d'administration autonome baptisé TUNe [**?**]. En plus de valider une approche de développement des systèmes d'administration autonome, TUNe a été utilisé pour l'administration de plusieurs types d'applications. Cependant, il s'est relativement inadapté pour l'administration des systèmes large échelle ou virtualisés. Ainsi, sur les bases de TUNe, nous proposons dans cette thèse la conception d'un **SAA hautement flexible et adaptable**, pouvant être notamment spécialisé pour une utilisation dans le cloud. Cette démarche de construction d'un SAA adaptable au cloud (et non spécifique au cloud) s'explique par le fait que notre activité de recherche ne s'inscrit pas (et ne s'inscrira pas dans le futur) uniquement dans la technologie du cloud, mais toujours dans l'administration autonome.

Compte tenu de la double problématique que nous adressons (construction d'un SAA adaptable et application pour l'administration dans le cloud), l'organisation de ce document de thèse suit globalement une trajectoire en U résumée par la figure 1.1 :

- **Partie I : Contexte du cloud**
  Cette partie est composée des chapitres 2 et 3. Le chapitre 2 présente le cloud et ses apports dans les entreprises (tout en clarifiant quelques définitions) tandis que le chapitre 3 présente en quoi consiste les tâches d'administration dans le cloud.
- **Partie II : Contexte SAA**
  Cette partie regroupe les chapitres 4 et 5. Le chapitre 4 présente le contexte

scientifique qu'est l'administration autonome en parcourant quelques approches de conception de SAA. Quant au chapitre 5, il présente le prototype de SAA TUNe (développé au sein de notre équipe) ainsi que ses difficultés à répondre aux besoins d'administration du cloud.

- **Partie III : Positionnement**
  Cette partie regroupe les chapitres 6 et 7. Dans le chapitre 6, nous proposons un canevas de conception d'un SAA adaptable et générique. Quant au chapitre 7, il présente une étude des travaux de recherche effectués autour des SAA et des plateformes d'administration dans le cloud.
- **Partie IV : Contribution SAA**
  Cette partie se résume au chapitre 8 dans lequel nous présentons l'implantation d'un SAA adaptable suivant le canevas que nous avons précédemment proposé.
- **Partie V : Contribution Cloud**
  Cette partie contient les chapitres 9 et 10. Le chapitre 9 présente la personnalisation du SAA adaptable que nous avons implanté, pour l'administration dans le cloud. Quant au chapitre 10, il montre les évaluations de cette personnalisation dans la réalisation des tâches d'administration dans le cloud.

Chapitre 2

Le cloud

Contents

2.1 Le Cloud Computing 5
- 2.1.1 Généralités et Définition 5
- 2.1.2 Cloud vs Grilles 6
- 2.1.3 Principes 7
- 2.1.4 Bénéfices 8
- 2.1.5 Classification 10
  - 2.1.5.1 Par raison de développement 10
  - 2.1.5.2 Par niveau de service 11
- 2.1.6 Challenges 12
  - 2.1.6.1 Isolation 13
  - 2.1.6.2 Administration 14
  - 2.1.6.3 Interopérabilité et Portabilité 14

2.2 Isolation par la virtualisation 16
- 2.2.1 Définition et Principes 16
- 2.2.2 Objectifs 17
- 2.2.3 Bénéfices pour les entreprises 18
- 2.2.4 Classification 19
- 2.2.5 Synthèse 20

2.1 Le Cloud Computing

2.1.1 Généralités et Définition

Face à l'augmentation continuelle des coûts de mise en place et de maintenance des systèmes informatiques, les entreprises externalisent de plus en plus leurs services informatiques. Elles confient leur gestion (des services informatiques) à des entreprises spécialisées (que nous appelons *fournisseurs*). L'intérêt principal de cette stratégie réside dans le fait que l'entreprise ne paie que pour les services effectivement consommés. En effet, une gestion interne de ces services par l'entreprise ne serait pas complètement amortie, en particulier lorsque les besoins d'utilisation varient. Le développement de ce mode de fonctionnement a été favorisé par plusieurs facteurs tels que l'évolution et la généralisation des accès internet, l'augmentation de la puissance des ordinateurs et des réseaux informatiques. Le *Cloud Computing* se situe dans cette orientation récente.

Devant le manque de consensus sur la définition de la notion de *Cloud Computing*, nous reprenons la définition proposée par CISCO [?] : "***le Cloud Computing est une plateforme de mutualisation informatique fournissant aux entreprises des services à la demande avec l'illusion d'une infinité des ressources***". Dans cette définition, nous retrouvons quelques similitudes avec les plateformes connues comme les grilles de calcul ou encore les centres d'hébergement. Il est présenté dans la littérature comme une évolution de ces infrastructures d'hébergement mutualisées. En guise d'exemple, prenons l'évolution proposée par [?] (figure 2.1) qui est largement citée dans la littérature. D'après [?], le cloud computing est le fruit d'une évolution pouvant être présentée en 5 phases :

- Elle débute avec les Fournisseurs d'Accès Internet (FAI 1.0). Ils ont pour but de mettre en place des moyens de télécommunication afin d'assurer le raccordement des personnes ou entreprises au réseau internet.
- La seconde phase est l'orientation des FAI vers l'hébergement de pages web (FAI 2.0). Cette phase marque un grand bond dans le développement d'internet.
- La troisième phase (FAI 3.0) est la possibilité qu'offrent les FAI à héberger des applications métiers des entreprises.
- Une connaissance des besoins applicatifs des entreprises permettent aux FAI de faire évoluer leur domaine d'intervention. Ils mettent en place des plateformes de génération d'applications à la demande. Il s'agit des ASP (Application Service Provider) dont les "Software as a Service" (section 2.1.5.2) sont des dérivés : c'est le FAI 4.0.
- La généralisation des pratiques précédentes, la prise en compte de nouvelles pratiques et l'intégration des principes que nous présentons dans les sections suivantes donnent naissance au cloud computing.

Suivant cette évolution, nous présentons dans la section suivante notre point de vue concernant la position du cloud computing par rapport aux plateformes d'hé-

bergement mutualisées comme les grilles [?] (grid5000 [?], Globus [?]).

N.B : Dans ce document, nous utilisons l'acronyme "CC" pour désigner la technologie *Cloud Computing* et le terme "cloud" pour parler d'une plateforme de Cloud Computing.

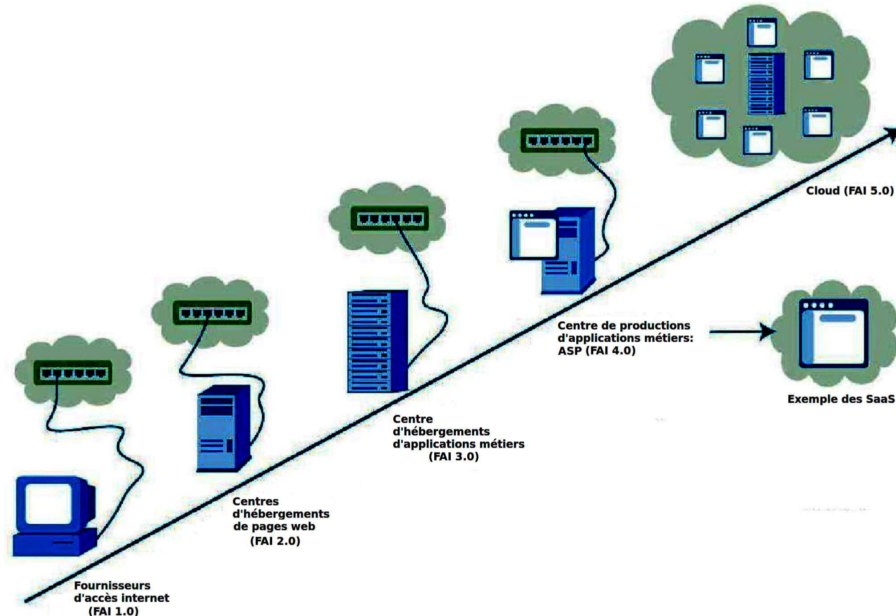


FIGURE 2.1 – Évolution vers le Cloud

2.1.2 Cloud vs Grilles

Introduites pour la première fois dans les années 90, les grilles de calcul situent la mutualisation au cœur de leur technologie. De même, la mutualisation est au cœur de la technologie du CC. La question que nous nous posons : "A quel niveau se situe la différence entre le cloud et les grilles (si elle existe) ?". Autrement dit, le terme "Cloud Computing" n'est-il pas une autre appellation de "Grid Computing" ? Après consultation de la littérature, nous ne trouvons qu'une étude sérieuse consacrée entièrement à cette comparaison ([?]). Le constat global de [?] est proche du nôtre.

D'un point de vue technologique nous n'identifions pas de réelle différence entre les plateformes de grille et de cloud. Cependant, l'ouverture du cloud aux utilisateurs de différents niveaux (pas toujours des informaticiens comme dans les grilles) et éventuellement son caractère commercial sont les potentielles différences que nous identifions. De plus, dans les environnements de grilles, les applications (appelées le plus souvent job) s'exécutent généralement sur une durée limitée (même si celle-ci peut être illimitée comme dans le cloud). Comme nous le présentons dans la section 3, ces différences rendent la gestion et l'utilisation des plateformes de cloud plus complexes que dans les grilles.

Somme toute, nous considérons que la technologie de cloud computing utilise la technologie de grille à laquelle elle associe les principes d'ouverture au public, de

facturation, et d'hébergement durable. Dans la section suivante, nous détaillons ces différents principes.

2.1.3 Principes

Nous avons énoncé dans la section précédente quelques principes fondamentaux du cloud. Ces principes lui permettent de se démarquer des plateformes d'hébergement classiques (grille, data center, centre d'hébergements). Les plus importants d'entre eux sont la **mutualisation** et la **facturation des ressources** à l'usage :

La mutualisation. C'est la pratique qui consiste à partager l'utilisation d'un ensemble de ressources par des entreprises (ou entités quelconques) n'ayant aucun lien entre elles. Les ressources peuvent être de diverses natures : logicielles ou matérielles (machines, équipements réseau, énergie électrique). Cette pratique dépend du désir des entreprises de délocaliser leurs services informatiques vers des infrastructures de cloud.

Reposant sur les technologies de grilles, le cloud doit cependant faire face aux problèmes liés à son exploitation et utilisation. Il s'agit des problèmes classiques tels que la sécurité, la disponibilité, l'intégrité, la fiabilité et l'uniformité d'accès aux données.

L'allocation et facturation à la demande. Contrairement aux centres d'hébergement web classiques dans lesquels le paiement se fait à l'avance sous forme de forfait, le cloud propose une facturation à l'usage. Cette dernière peut être implémentée de deux façons. La première consiste à facturer à l'entreprise la durée d'utilisation d'un ensemble de ressources quelque soit l'utilisation effective. Par exemple, soit r l'ensemble des ressources réservées par l'entreprise. Soient t_1 et t_2 respectivement l'instant de début et de fin d'utilisation des ressources. Soit C_u le coût d'utilisation d'une ressource durant une unité de temps. Ainsi, le coût total d'utilisation de l'ensemble des ressources r est : $C_r=(t_2-t_1)*C_u*r$. Quant à la deuxième façon, elle est beaucoup plus fine que la première. Elle consiste à facturer les véritables instants pendant lesquels les ressources ont été utilisées. En partant des mêmes paramètres que précédemment, soit $T=\{t_{u1},t_{u2},\ldots,t_{un}\}$ avec $t_{ui} \in [t_1 ;t_2]$, $1 \leq i \leq n$, l'ensemble des unités de temps pendant lesquelles les ressources ont été réellement utilisées. Dans ce cas, le coût total d'utilisation de l'ensemble des ressources r est : $C_r=C_u*r*\sum_{i=1}^{n} t_{ui}$.

A la lumière de ces pratiques, nous montrons dans quelle mesure les conséquences de l'utilisation du cloud peuvent être bénéfiques à la fois pour le fournisseur et pour les entreprises qui y ont recours.

2.1.4 Bénéfices

Les retombées des principes du cloud sont bénéfiques à la fois pour son fournisseur, les entreprises délocalisant leurs infrastructures et plus largement pour la nature (au sens écologique). Globalement, ils assurent aux deux premiers une meilleure rentabilité. De plus ils permettent à l'entreprise de se concentrer sur les tâches de production autres que la maintenance de systèmes informatiques.

Pour le fournisseur

Les bénéfices du fournisseur sont uniquement dûs au fait de la mutualisation des ressources. En effet, après son investissement dans la mise en place des infrastructures pour le cloud, il fait payer aux entreprises la marge nécessaire pour sa rentabilisation. Comme pour une entreprise disposant d'une plateforme interne, il paie pour les frais d'administration de l'ensemble. Cette dépense peut être amortie par facturation aux entreprises. En plus de cette marge, il bénéficie des coûts de réutilisation des ressources. En effet, compte tenu de la non appartenance des ressources aux entreprises, elles (les ressources) leurs sont facturées à chaque usage. La même ressource peut ainsi faire l'objet de plusieurs facturations.

Pour l'environnement

A l'ère de l'écologie et des politiques de réduction de la consommation énergétique, l'investissement dans les plateformes de cloud représente un geste envers la nature. La mutualisation de ressources (telle que pratiquée dans le cloud) accompagnée par la délocalisation des infrastructures d'entreprise vers les clouds permettent de réduire les consommations énergétiques. Pour illustrer cette affirmation, reprenons l'étude [**?**] réalisée par le groupe "WSP Environment & Energy (WSP E&E)"[1] pour le compte de Microsoft à propos de la plateforme de cloud Azure [**?**]. Cette étude est résumée dans sa conclusion : "Moving business applications to the cloud can save 30 percent or more in carbon emissions per user.". Elle montre que la délocalisation des applications Microsoft Exchange Server 2007 [**?**] des entreprises vers le cloud Microsoft Azure permet de réduire d'au moins 30% les émissions CO_2 par utilisateur de chaque entreprise. Cette réduction s'explique par deux facteurs. D'une part, la délocalisation permet de réduire le nombre et la taille des infrastructures informatiques en service. Ceci implique donc une réduction de la consommation énergétique (donc moins de rejet de CO_2). D'autre part, le fournisseur Microsoft implante dans son cloud des techniques d'utilisation efficace de ressources.

1. Groupe de Consulting en matière d'environnement, d'énergie et de climat : www.wspenvironmental.com

Pour l'entreprise

C'est elle la première gagnante de cette technologie. Elle réalise des bénéfices en argent et en flexibilité dans sa capacité à s'agrandir.

La réduction des coûts. Le recours au cloud permet à l'entreprise d'être facturée à l'usage, en fonction de ses besoins. Pour avoir une idée du gain réalisé, reprenons cette observation de Michael Crandell du groupe RightScale [**?**] à propos du cloud d'Amazon [**?**] : "*Le coût à pleine charge d'un serveur sur Amazon se situe entre 70$ et 150$ par mois alors qu'il s'élève à 400$ en moyenne par mois s'il était hébergé par l'entreprise en interne*". Plusieurs raisons expliquent cette différence de coût. En effet, une gestion interne de l'infrastructure implique l'achat des matériels, l'affectation du personnel (et donc du coût salarial qu'il induit) pour la gestion de l'infrastructure et divers moyens de production mis en place pour le fonctionnement de l'ensemble (électricité, locaux, etc). Le partage de ressources tel que pratiqué dans le cloud permet au fournisseur de répartir ces coûts entre plusieurs entreprises.

La réduction des gaspillages. Les infrastructures gérées en interne sont souvent sous-utilisées, alors que l'infrastructure d'un cloud mutualise l'ensemble de ressources pour un grand nombre d'entreprises. La mutualisation consiste à mettre à la disposition de plusieurs utilisateurs une base commune de ressources. Elle permet ainsi d'augmenter le taux d'utilisation de ces ressources. En effet, les ressources n'étant pas dédiées à un seul utilisateur, elles pourront servir à d'autres en cas de besoin.

La flexibilité et accès aux ressources large échelle. L'entreprise peut augmenter la capacité de son infrastructure sans investissement majeur. En effet, grâce à l'allocation dynamique (à la demande) des ressources qu'offre le cloud, il suffit de souscrire à de nouvelles ressources et celles-ci sont directement allouées. De plus, l'entreprise est libre de ses allées et venues car les contrats d'utilisation sont limités dans le temps (autour de l'heure). Ainsi, l'entreprise peut augmenter ou réduire son infrastructure à sa guise à moindre coût et dans un délai réduit. Rappelons que le cloud offre ainsi à l'entreprise une possibilité d'accéder à une quantité de ressources dont elle ne pourrait se l'offrir en interne. Elle peut dorénavant envisager des applications large échelle sans se soucier de l'obtention des équipements. A ce sujet, on observe quelques dérives avec des groupes qui acquièrent un grand nombre de ressources dans les clouds à des fins criminelles (décryptage de clé de sécurité ou dénis de service en sont des exemples) [2].

Notons que cette flexibilité dépend du type de cloud considéré (section suivante). Par exemple, l'augmentation de ressources dans un cloud de type entreprise ne sera pas aussi rapide que dans un cloud de type fournisseur de services. Dans le premier cas, la décision est prise de façon collégiale (entre toutes les entreprises interve-

2. Attaque pour décryptage de la clé de sécurité des Network PlayStation de Sony en mai 2011

nantes) et peut donc prendre un temps considérable. A l'opposé, dans le second cas, l'opération se réalise dans l'immédiat.

2.1.5 Classification

Avant de présenter les différents types de cloud pouvant être développés, nous établissons dans un premier temps quelques critères de classification :

- La raison de développement (*business model*). C'est la raison qui justifie la mise en place de la plateforme. Elle peut être commerciale, scientifique ou communautaire.
- Le niveau de services. C'est l'ambition du cloud à fournir aux entreprises une plateforme proche ou pas de leurs attentes.
- L'accessibilité. Le cloud peut être accessible par tous ("cloud public") ou restreint à un public particulier ("cloud privé"). Contrairement aux deux premiers, nous ne développons pas celui-ci étant donné sa simplicité de compréhension.

2.1.5.1 Par raison de développement

L'utilisation du CC ne se limite pas uniquement aux entreprises à caractère commercial. En fonction des raisons de sa mise en place, nous distinguons quatre catégories de plateformes de CC à savoir :

Cloud d'Entreprises. Dans cette catégorie, nous retrouvons des entreprises de petites et de moyennes tailles disposant chacune de peu de ressources et de moyens de maintenance de leurs infrastructures. Elles se regroupent donc autour d'un projet de cloud afin de mutualiser leurs capacités. La plateforme qui en découle est privée, c'est-à-dire accessible uniquement par les entités des différentes entreprises. Cette plateforme à l'avantage d'être de petite taille et d'accès restreint à des utilisateurs connus. Ainsi, les problèmes de sécurité sont réduits et l'administration peut être spécialisée. Les groupes Amazon EC2 [**?**] (via le "*Virtual Private Cloud*"), VMware [**?**] ou encore VeePee [**?**] offrent par exemple des solutions de clouds privés.

En comparaison avec les technologies existantes, cette catégorie est identique aux clusters privés.

Cloud Gouvernemental et Recherche Scientifique. Pour des raisons de recherche et de développement, des instituts de recherche mettent sur pied des environnements de cloud. Leur développement est encouragé et financé par des gouvernements. L'accès est exclusivement réservé aux personnes exerçant dans le même domaine de recherche, ou appartenant aux instituts de recherche associés, ou ayant une dérogation précise. Ces plateformes sont pour la plupart orientées projets. Seules les avancées scientifiques obtenues par les groupes de recherche qui l'utilisent permettent de valoriser la plateforme.

D'un point de vue technologique, d'objectif et d'utilisation, aucune différence n'est notable avec les grilles scientifiques comme grid5000 [?].

Cloud pour Réseaux Sociaux et Jeux. Le développement des réseaux sociaux et des jeux en ligne nécessite de plus en plus de grandes quantités de ressources. Cette nécessité est due à la croissance presque exponentielle d'utilisateurs. De plus, l'essence de ces environnements est la mise en commun d'un certains nombre de données et de connaissances (donc de ressources). Dans ce contexte, le développement d'une plateforme similaire au cloud devient une évidence pour optimiser l'utilisation des ressources et faciliter le partage de données. En effet, elles sont considérées comme plateforme de cloud à cause de leur recours aux principes de développement de celui-ci.

L'ouverture de ces plateformes à tous et le nombre important d'utilisateurs pourraient constituer un handicap pour leur gestion. Or n'hébergeant qu'une seule application (celle du fournisseur), leur gestion est spécialisée et moins complexe. Elles sont comparables aux clusters/griles privés. Les plateformes comme celles du réseau social facebook [?] ou des jeux en ligne zynga [?] font partie de cette catégorie.

Cloud pour Fournisseurs de Services. C'est le modèle le plus répandu. Une entreprise, appelée fournisseur, met à la disposition d'autres (appelées clients) une plateforme d'exécution d'applications et assure le service informatique inhérent. Il s'agit d'un modèle ouvert à tout public et à caractère commercial. La plateforme héberge tous types d'applications et l'accès à ces applications est ouvert aux utilisateurs externes. Les défis de sécurité et d'administration (que nous détaillons dans la section 2.1.6) sont importants dans ce modèle. La plateforme de CC Amazon Elastic Compute Cloud (EC2) fait partie de ce cette catégorie. Sachant que cette catégorie peut regrouper les autres, les contributions que nous apportons dans cette thèse s'adressent à cette catégorie de cloud. Notons que quelque soit le modèle de développement considéré, la plateforme qui en découle peut également être classée selon son niveau d'utilisation (section suivante).

Notons qu'il existe des communautés de développement logiciels autour de ces catégories de cloud. Elles fournissent des briques logicielles permettant de construire/enrichir une plateforme de cloud. Très souvent, il s'agit d'entreprises dont le but est de fidéliser ses clients (futurs propriétaires de plateformes de cloud). Les logiciels sont présentés à ces derniers comme une valeur ajoutée de l'évolution d'un produit existant (comme un système d'exploitation). Ce développement est le plus souvent pratiqué par des communautés de logiciels libres (plateforme Ubuntu Enterprise Cloud [?] ou Eucalyptus [?]) ou encore par des grands groupes de développement de logiciels spécialisés (Oracle Cloud Computing [?]).

2.1.5.2 Par niveau de service

En fonction du niveau d'abstraction qu'offre le cloud aux entreprises, nous identifions dans la littérature trois catégories de plateformes de CC (figure 2.2(a)) :

Infrastructure as a Service (IaaS). C'est le niveau de service le plus bas. Le cloud fournit des ressources matérielles aux entreprises (capacité de traitement, de stockage, etc.). L'accès à ces ressources peut être direct ou virtuel (section 2.2). On retrouve dans cette catégorie les plateformes comme Amazon Elastic Compute Cloud (EC2) [?] et IELO Cloud [?].

Platform as a Service (PaaS). Il s'agit d'un niveau d'abstraction au-dessus de l'IaaS dans lequel le cloud fournit une plateforme applicative de programmation. Elle permet à l'entreprise de programmer des applications facilement administrables dans le cloud. Elle oblige l'entreprise d'une part à maîtriser l'API du PaaS et d'autre part à ré-implémenter ses applications suivant cet API. Google App Engine [?] et Windows Azure [?] sont des exemples de PaaS.

Software as a Service (SaaS). Ici, le cloud fournit directement les applications correspondants aux attentes des entreprises. Les tâches d'administration sont assurées par le cloud ; l'entreprise n'a pas grand chose à effectuer. Très spécialisé (par l'application qu'il fournit), ce type de cloud est le moins répandu. Les clouds pour réseaux sociaux et jeux (présentés précédemment) font partie de cette catégorie. Comme exemple de plateforme SaaS, nous pouvons citer Rightscale [?] ou encore CORDYS [?].

Cette classification est la plus répandue dans la littérature. Dans le but d'éviter tout malentendu, nous proposons dans le paragraphe suivant, une vision simplifiée du cloud. Cette présentation est proche des clusters/grilles qui sont technologiquement identiques au cloud.

Notre vision. Sans entrer en contradiction avec la présentation précédente, nous considérons toute plateforme de cloud comme une plateforme à deux niveaux (figure 2.2(b)) :

- Abstraction et mutualisation des ressources. Ce niveau correspond exactement à l'IaaS dans la vision classique. Nous y retrouvons donc à la fois les ressources et les applications permettant leur abstraction et mutualisation.
- Les applications des entreprises et celles utiles à l'exploitation du cloud. Il s'agit aussi bien des applications du SaaS, des applications développées à partir d'un PaaS, que de celles issues ni du PaaS ni du SaaS. Pour aller plus loin, nous considérons le PaaS comme une application s'exécutant sur l'IaaS. De ce fait, il a le même statut que les autres applications. L'un des défis du cloud est l'isolation de ces différentes applications.

2.1.6 Challenges

Le CC n'est pas une révolution technologique en soit mais constitue une orientation vers un mode de gestion des infrastructures informatiques des entreprises. Cependant, l'idée d'héberger plusieurs applications d'utilisateurs différents pose quelques

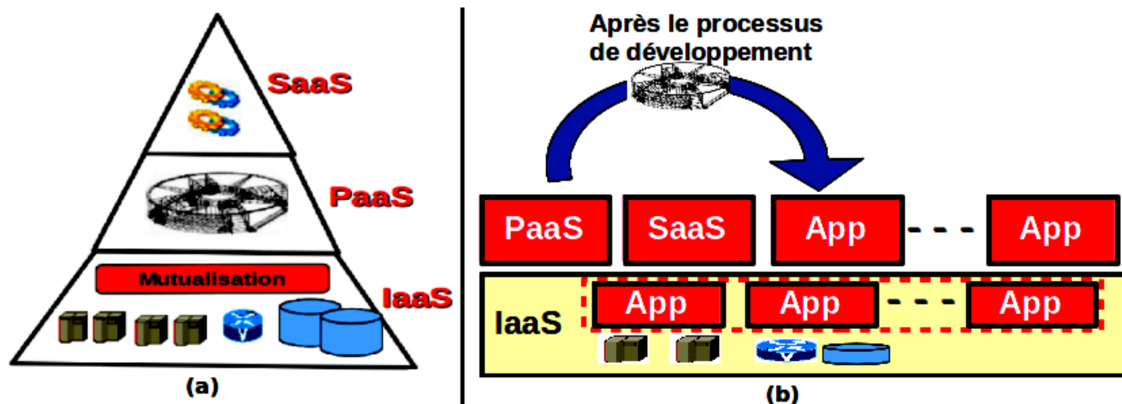


FIGURE 2.2 – (a)Vision classique. (b) Notre vision.

défis que doit surmonter le cloud. Il s'agit de l'isolation, l'administration, l'interopérabilité et la portabilité des applications entre plusieurs plateformes.

2.1.6.1 Isolation

La mutualisation de ressources dans le cloud (comme dans toutes les infrastructures) implique la mise en place de divers mécanismes (sécurité, comptage de ressources, conflit d'accès, etc). Le plus important de ces derniers est la gestion des conflits/interférences d'accès entres les utilisateurs. Une réponse idéale pour la mise en place de la mutualisation est l'isolation. Nous regroupons sous ce terme plusieurs types d'isolation à savoir : l'isolation des ressources, l'isolation d'espaces utilisateurs, l'isolation des performances et l'isolation des défaillances.

- **L'isolation des ressources** (ou encore partitionnement) garantit au client l'exclusivité d'accès à un ensemble de fractions ("morceaux") de ressources durant toute sa présence dans le cloud (malgré la mutualisation). Le client a l'illusion d'être le propriétaire et considère l'ensemble comme des machines entières. Cette isolation permet d'éviter les situations de famine aux applications du client (situation dans laquelle une application attend indéfiniment une ressource détenue par une autre). De plus, elle permet au fournisseur du cloud d'identifier et de compter les utilisations de ressources pour chaque utilisateur. Ce décompte servira par la suite à la facturation.
- **L'isolation d'espaces utilisateurs** donne à chaque client du cloud l'illusion d'être le seul utilisateur. Rien ne doit le laisser présager la présence d'autres utilisateurs ou applications. Illustrons cela à travers l'exemple d'un cloud fournissant des environnements Linux. Dans cet environnement, chaque client peut accéder en mode super utilisateur (*root* sous Linux) aux machines lui appartenant. Cependant, l'exécution de la commande *ps -aux* (affichage de la liste des processus) par exemple ne doit présenter que les processus démarrés par l'utilisateur en question.

- **L'isolation des performances** permet au cloud d'assurer le non monopole des ressources globales du cloud par un seul client. Prenons l'exemple des ressources réseaux pour illustrer cela. Une utilisation intensive de la bande passante sur le cloud par une application cliente peut affecter l'ensemble du réseau et ainsi avoir un impact sur les autres utilisateurs.
- **L'isolation des défaillances** permet d'assurer la non violation des espaces utilisateurs dans le cloud. Il comprend également le défi de sécurité. En tant que centre d'hébergement d'applications multi-utilisateurs, le cloud doit garantir l'intégrité de chaque espace utilisateur vis-à-vis des autres. Ainsi, aucune action malveillante réalisée par un client ne doit altérer ni le fonctionnement du cloud ni celui des applications appartenant à d'autres clients. Cet objectif est d'autant plus important que la plupart des utilisateurs du cloud sont non identifiables. Il s'agit des utilisateurs des services hébergés par les entreprises dans le cloud.

La prise en compte de ce défi a été effectuée grâce à l'introduction des techniques de virtualisation (section 2.2) dans l'implémentation du cloud.

2.1.6.2 Administration

Comme nous l'avons présentée précédemment, l'exploitation des plateformes de cloud implique l'intervention de trois types d'utilisateurs : l'administrateur du cloud, les entreprises et les utilisateurs des applications des entreprises. Les deux premiers (administrateur et entreprises) sont confrontés quotidiennement à plusieurs tâches d'administration (pour la plupart répétitives). L'allègement de ces tâches conditionne l'expansion du cloud dans les entreprises. En effet, afin d'éviter le même constat observé de l'utilisation des grilles de calculs (réservés aux scientifiques, donc aux utilisateurs avertis), les plateformes de cloud doivent prendre en compte et faciliter les tâches d'administration de ces deux utilisateurs. Comme nous le présentons dans la section 3, les opérations d'administration effectuées par ces deux utilisateurs sont semblables. Notons que l'objet de cette thèse porte essentiellement sur ce challenge.

2.1.6.3 Interopérabilité et Portabilité

Face à la multiplication des plateformes de cloud, les clients pourront être confrontés plus tard à deux choix : (1) la migration d'une application d'un cloud vers un autre et (2) l'utilisation de plusieurs clouds pour l'hébergement de la même application. Le choix (1) se pose par exemple lorsque la concurrence entraîne un client à partir du cloud qui héberge son application vers un autre plus attrayant. Elle peut également survenir lorsque la plateforme initiale décide de rompre ses services, ce qui oblige le client à trouver une autre plateforme pouvant accueillir ses applications. Dans ces deux situations, il se pose le problème de portabilité de l'application du cloud initial vers le cloud de destination. Quant au choix (2), il survient lorsque le cloud hébergeant l'application se retrouve à cour de ressources. Dans ce cas, le client

ou le fournisseur peut décider d'associer au cloud initial des ressources venant d'une autre plateforme. Ainsi, la même application s'exécute dans deux environnements de cloud différents appartenant à des fournisseurs distincts. L'ensemble formé par les deux plateformes constitue un "cloud hybride". Cette situation soulève le problème d'interopérabilité entre les plateformes de CC. Dans ces deux situations ((1) et (2)), la mise en place d'une API harmonisée (par standardisation) pour le développement des plateformes de CC est la bienvenue.

2.2 Isolation par la virtualisation

Comme nous l'avons présentée dans la section 2.1.6.1, l'isolation (au sens de la présentation de la section 2.1.6.1) représente l'un des défis majeurs dans l'implémentation des plateformes de cloud. Il existe plusieurs façons de la mettre en œuvre :

- La première méthode consiste à allouer la ressource matérielle entière à une entreprise même si celle-ci ne souscrit que pour une fraction de cette ressource. Cette méthode ne permet qu'une résolution partielle des problèmes d'isolation. En effet, certaines ressources comme le réseau et sa bande passante restent partagées entre les entreprises dans le cloud. A moins que le fournisseur alloue exclusivement à chaque entreprise des équipements et la bande passante réseaux (ce qui n'est pas raisonnable), il est impossible avec cette méthode d'éviter des situations de monopole de ces ressources par entreprise. Elle doit être complétée avec une solution logicielle.
- La seconde méthode consiste à laisser la responsabilité aux entreprises d'implanter les mécanismes d'isolation. Cette méthode n'est pas envisageable dans la mesure où le cloud ne dispose d'aucun moyen d'introspection des applications d'entreprise afin de s'assurer de l'implantation de ces mécanismes.
- La dernière méthode est intermédiaire (matérielle et logicielle) aux deux premières. Tout en implémentant les mécanismes d'isolation, elle donne l'illusion à l'entreprise d'avoir un accès direct et exclusif à la ressource matérielle. Parallèlement, elle garantie au cloud le non accès direct des entreprises aux ressources matérielles. C'est de l'**isolation par virtualisation**.

2.2.1 Définition et Principes

La virtualisation [**?**] se définit comme l'ensemble des techniques matérielles et/ou logicielles qui permettent de faire fonctionner sur une seule machine, plusieurs systèmes d'exploitation (appelés machines virtuelles (VM), ou encore OS invité). Elle garantie l'indépendance et l'isolation des VM (l'isolation telle que présentée dans la section 2.1.6.1). En bref, elle permet d'obtenir au niveau des VM la même isolation qu'offre les machines réelles.

L'implémentation d'un système de virtualisation repose sur une application s'exécutant entre le matériel et les machines virtuelles : c'est la "*Virtual Machine Monitor* (VMM)". C'est elle qui implante les mécanismes d'isolation et de partage des ressources matérielles. La figure 2.3 montre l'architecture globale d'un système d'une machine virtualisée (c-à-d exécutant un système de virtualisation). La VMM est capable de démarrer simultanément plusieurs machines virtuelles de différents types (Linux, Mac ou encore Windows) sur le même matériel. Comme dans un système d'exploitation normal, chaque VM conserve son fonctionnement habituel. Autrement dit, elle a l'illusion de gérer les accès mémoire, disque, réseaux, processeur et autres périphériques de ses processus. Vu de l'extérieur, l'utilisateur perçoit l'en-

semble comme un environnement constitué de plusieurs machines réelles.

Quoique les VM gèrent les accès aux ressources, aucun accès aux ressources n'est possible sans l'aval et le concours de la VMM. Elle décide notamment des attributions de temps processeurs aux machines virtuelles. Quant à la communication avec l'extérieur, la VMM peut fournir plusieurs techniques pour rendre accessible ou non les VM. Elle peut la réaliser par assignation d'adresses IP et par implantation des mécanismes d'accès réseaux aux VM (par routage, filtrage de communication, etc).

En plus de fournir un système d'isolation de système d'exploitation, l'un des premiers objectifs de la virtualisation est d'offrir des performances proches de celles des machines réelles. La section suivante présente ces objectifs.

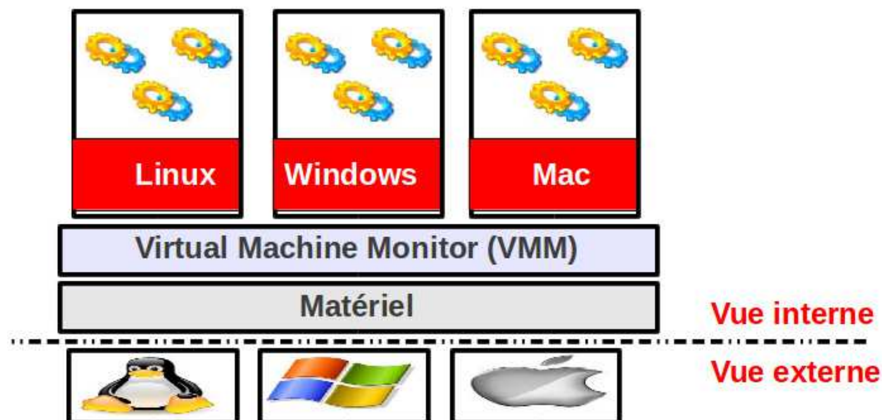


FIGURE 2.3 – Vue des systèmes virtualisés

2.2.2 Objectifs

De façon générale, l'implémentation d'un système de virtualisation doit remplir les trois objectifs suivants [**?**] :

L'équivalence : toute exécution d'application dans un système virtualisé doit être identique à une exécution sur une machine réelle ; à l'exception du temps d'exécution lié à la disponibilité des ressources (plusieurs études [**?**] montrent leur rapprochement).

L'efficacité : la majorité des instructions de la VM doit directement être exécutée par le processeur sans intervention du logiciel de virtualisation.

Le contrôle de ressources : l'ensemble des ressources est géré de façon exclusive (voir la section précédente) par le logiciel de virtualisation. Ceci permet d'assurer l'isolation de performance et de sécurité.

2.2.3 Bénéfices pour les entreprises

Malgré le surcoût d'exécution induit (3%[?]) par les systèmes de virtualisation actuels, la virtualisation offre plusieurs avantages aux entreprises qui en font l'usage. Nous remarquons dans la littérature, qu'il existe un amalgame entre les apports technologiques de la virtualisation et les conséquences de ces apports dans une entreprise. Dans cette section, nous évitons de faire cet amalgame.

Le tour est vite fait lorsqu'il s'agit de trouver les apports techniques de la virtualisation dans le domaine des systèmes (en tant que domaine de recherche) : il s'agit essentiellement de l'isolation. Présenté de cette façon, d'aucuns compareront cette isolation à celle déjà proposée par les systèmes d'exploitation pour les processus. En réalité, le véritable apport est la capacité à isoler l'exécution de plusieurs systèmes d'exploitation (et non processus) dans le même système d'exploitation. En raison du rapprochement avec certains objectifs du CC, l'isolation proposée par la virtualisation se décline également sous plusieurs formes identiques au CC (voir la section 4.1 pour leurs descriptions).

Les atouts de la virtualisation peuvent se traduire sous plusieurs formes en entreprise. Les bénéfices possibles sont les suivants :

Réduction des coûts. Au lieu d'acquérir plusieurs serveurs matériels pour l'exécution de logiciels incapables de cohabiter, l'entreprise utilise des machines virtuelles afin d'isoler chaque logiciel. Pour les mêmes raisons que le CC, cette exécution regroupée permet également de réduire les coûts en consommation électrique, ou encore en superficie des locaux qui abritent les serveurs. Un atout concerne les développeurs de systèmes d'exploitation. Au lieu de dédier une machine pour la réalisation des tests, les développeurs peuvent se servir des machines virtuelles.

Unité de facturation. Dans un centre d'hébergement, au lieu d'utiliser une machine entière comme unité d'allocation, la virtualisation permet d'allouer une fraction de machine aux clients. Cet atout est notamment exploité dans certaines plateformes de cloud.

Sauvegarde de services. Dans certaines applications, la robustesse fait partie des premiers critères d'évaluation. Elle se caractérise par la capacité du système à reprendre son activité dans un état proche de la normale (c-à-d avant la panne) en cas de panne d'un des serveurs. La virtualisation permet de sauvegarder périodiquement l'état d'exécution d'une VM et de la redémarrer en cas de nécessité à partir d'un point de sauvegarde. Cette opération est appelée *checkpointing* dans le jargon de la virtualisation.

Transfert de services. Nous entendons par transfert de services la possibilité de déplacer l'exécution d'un service d'une machine réelle à une autre sans interruption. Cette caractéristique est permise dans la virtualisation via des opérations de "migration à chaud". Elle permet d'exploiter les ressources d'une machine réelle sous-utilisée par un service en cours d'exécution sur une machine réelle sur-utilisée. Inversement, elle permet de consolider sur un nombre réduit de machines, des services en cours d'exécution sur plusieurs machines sous utilisées. Notons que tous

les systèmes de virtualisation ne fournissent pas ce service. De plus leur efficacité dépend de la technique d'implémentation utilisée.

2.2.4 Classification

Face à la difficulté de mise en œuvre du modèle de virtualisation proposé initialement (virtualisation complète du matériel), à cause de la non compatibilité des drivers matériels, plusieurs techniques de virtualisation se sont développées. Améliorées au fil des années, les techniques d'implémentation de systèmes virtuels peuvent être regroupées en différentes catégories selon le rapprochement/éloignement entre la VM et le matériel. La figure 2.4 recense les catégories majeures que nous présentons brièvement dans cette section. Le sens des flèches dans cette figure représente cette évolution chronologique. La présentation que nous donnons dans cette section suit également cet ordre. Ainsi le développement des systèmes les plus récents se justifie par les inconvénients des prédécesseurs. Notons que certains termes employés ici peuvent se retrouver dans la littérature avec des descriptions différentes.

Virtualisation du système de fichiers (figure 2.4(a)). C'est une méthode qui repose sur un système d'exploitation pré-installé (système hôte). Ce dernier permet de construire des espaces utilisateurs (désignation de la VM ici) dont les systèmes de fichiers sont complètement isolés. Chaque VM dispose d'une arborescence de système de fichiers à exclusif qui lui est propre. Les autres ressources telles que la mémoire, les cartes réseaux, le processeur sont directement accessibles par les VM. Il n'existe aucune isolation pour ces ressources. On retrouve dans cette catégorie des outils **chroot** ou UML (User Mode Linux) [**?**].

L'émulation (figure 2.4(b)). La catégorie précédente ne permettant pas l'installation d'OS, il s'est développé une technologie basée sur l'émulation. Cette technique propose une application (le virtualisateur) qui s'exécute au-dessus du système hôte, dans l'espace utilisateur. Cette application (qui correspond à la VMM) a pour mission d'émuler le matériel et permet ainsi de démarrer plusieurs systèmes d'exploitation réels dans l'espace utilisateur. Cette technique de virtualisation induit un surcoût considérable dans l'exécution des machines virtuelles. En effet, chaque opération effectuée dans la VM est interceptée et interprétée par la VMM. Le système de virtualisation VirtualBox [**?**] est basé sur cette technique.

Paravirtualisation (figure 2.4(c)). La paravirtualisation a été développée pour résoudre les problèmes de la catégorie précédente. Elle consiste à modifier les OS des VM afin qu'ils soient au courant de leur statut (de VM). Ainsi, le matériel est mappé dans la VM et donc accessible directement sous le contrôle de la VMM (hyperviseur sur la figure 2.4(c)). Cette dernière s'exécute directement au-dessus du matériel. Elle remplace l'OS hôte, qui est considéré comme une VM. La contrainte de modification d'OS des VM est une limite à cette technique. En effet, elle ne permet pas l'exécution de VM munies d'OS propriétaires (comme Windows). La plateforme

Xen [?] est le système de paravirtualisation le plus répandu grâce au niveau de performance qu'il offre [?]. Les travaux de cette thèse sont notamment basés sur ce système.

Virtualisation assistée par le matériel (figure 2.4(d)). L'évolution actuelle des drivers matériels et des processeurs (technologie Intel VT [?]) vers la prise en compte de la virtualisation permet de développer une nouvelle technique de virtualisation. Ainsi, l'intervention de la VMM sur les actions des VM est limitée et celles-ci n'ont plus besoin d'être modifiées. Les nouvelles implémentations de Xen ou VMware [?] permettent d'utiliser cette technique lorsque le matériel le supporte.

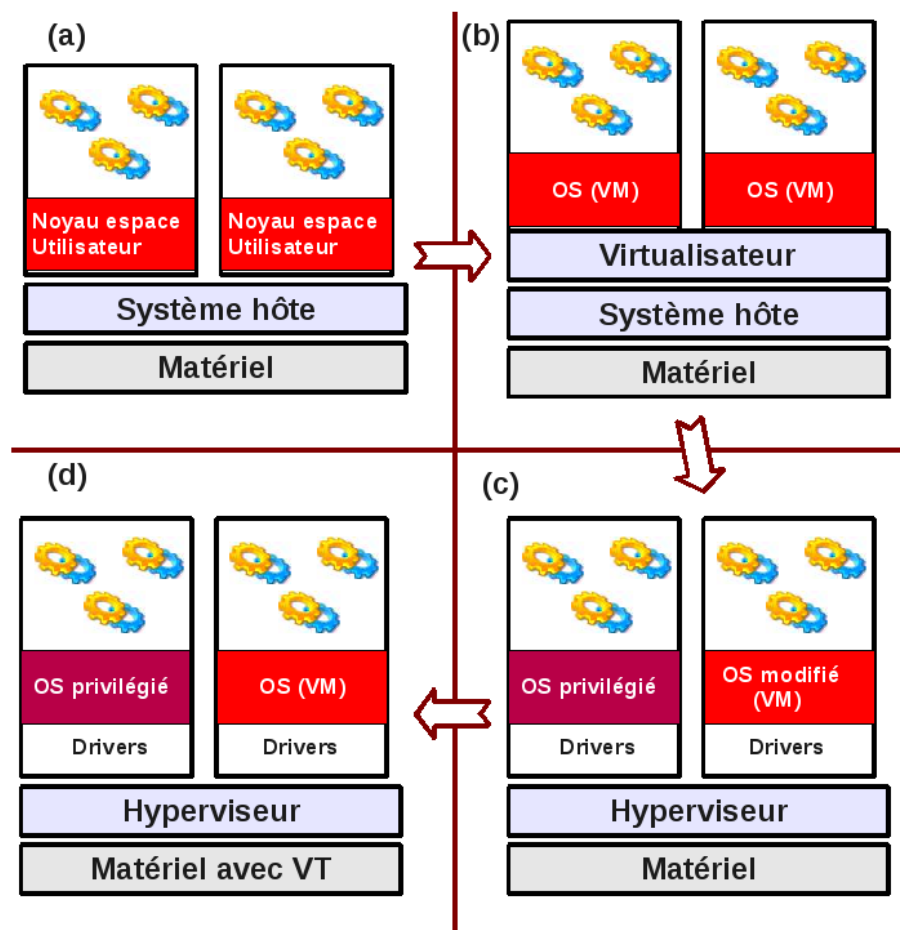


FIGURE 2.4 – Techniques de virtualisation

2.2.5 Synthèse

Après cette présentation de la virtualisation, nous constatons qu'elle répond parfaitement au challenge d'isolation auquel est confronté le cloud. De plus, nous constatons que les bénéfices de la virtualisation vis à vis des entreprises rejoignent ceux du cloud. Elle amplifie notamment la pratique de la mutualisation des ressources, qui

est au cœur du cloud. Quant aux techniques d'isolation par réservation entière de ressources, nous montrons qu'elles sont moins bénéfiques. En outre, elle ne couvrent pas tous les besoins d'isolation que requiert le cloud. Pour ces raisons, la majorité des plateformes de cloud ([**?**], [**?**], [**?**], [**?**], [**?**], etc.) adopte la virtualisation comme technique d'isolation. C'est cette catégorie de cloud qui nous intéresse dans cette thèse.

Son introduction dans le cloud implique la modification du mode de gestion des ressources et plus globalement des tâches d'administration. En ce qui concerne la gestion des ressources, l'unité d'allocation dans le cloud passe de la machine réelle à la machine virtuelle. Quant à l'administration, elle contraint les administrateurs du cloud d'acquérir des compétences en matière de virtualisation. Dans la section suivante, nous décrivons à quoi correspond l'administration dans une plateforme de CC basée sur la virtualisation.

Chapitre 3

Administration dans le Cloud

Contents

3.1 Administration niveau IaaS **23**
3.1.1 Allocation de ressources 24
3.1.2 Déploiement 24
3.1.3 Configuration et Démarrage 25
3.1.4 Reconfiguration 25
3.1.5 Monitoring 26
3.2 Administration niveau Entreprise **27**
3.2.1 Construction de VM et Déploiement 28
3.2.2 Allocation de ressources 29
3.2.3 Configuration et Démarrage 29
3.2.4 Reconfiguration 30
3.2.5 Monitoring 30
3.3 Synthèse : système d'administration autonome pour le cloud **31**

Conçues comme une évolution des plateformes partagées, les clouds nécessitent des tâches d'administration rencontrées dans les grilles et les environnements distribués en général. A celles-ci s'ajoutent celles relatives à l'introduction de la virtualisation. Elles concernent les différents protagonistes dans le cloud. Comme nous l'avons présenté dans la section 2.1.5.2, nous distinguons trois types d'utilisateurs dans le cloud (figure 3.1) : le fournisseur (administre le cloud), les entreprises clientes (utilisent les ressources du cloud et administrent ses applications) et les utilisateurs finaux (ceux qui utilisent les services fournis par les applications entreprises). Contrairement aux deux premiers utilisateurs, le dernier n'est confronté à aucune tâche d'administration. L'analyse de cette dernière laisse paraître une symétrie entre les opérations d'administration réalisées par le fournisseur du cloud (niveau IaaS) et celles réalisées par les administrateurs des entreprises clientes du cloud (niveau entreprise) (observable sur la figure 3.2). Globalement, les gestionnaires des deux niveaux d'administration assurent des tâches de :

- Déploiement : déploiement de VM au niveau IaaS et déploiement de logiciels pour l'entreprise cliente ;
- Monitoring : monitoring des ressources matérielles et VM au niveau IaaS (assuré par l'élément "*MonitoringController*" dans la figure 3.2) et monitoring des logiciels au niveau entreprise cliente (assuré par l'élément "*AppMonitoringControlle*" dans la figure 3.2) ;
- Gestion des ressources : allocation efficace des ressources matérielles au niveau IaaS (assurée par l'élément "*ResourceController*" dans la figure 3.2) et réservation efficace de VM au niveau entreprise cliente (assurée par l'élément "*AppResourceController*" dans la figure 3.2) ;
- Reconfiguration : reconfiguration des VM au niveau de l'IaaS (assurée par l'élément "*AppMonitoringControlle*" dans la figure 3.2) et reconfiguration des logiciels, pour le passage à l'échelle par exemple, au niveau entreprise cliente (assurée par l'élément "*AppScheduler*" dans la figure 3.2).

Dans cette section, nous présentons en détail et séparément d'une part l'administration telle qu'effectuée dans l'IaaS et d'autre part les opérations d'administration réalisées par l'entreprise.

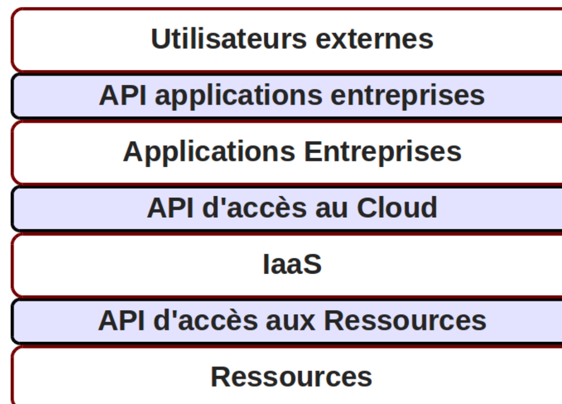


FIGURE 3.1 – Architecture simplifiée du Cloud

3.1 Administration niveau IaaS

Directement rattachée à l'environnement matériel, l'administration au niveau de l'IaaS se résume à la gestion des machines virtuelles (pour une utilisation efficace des ressources) et de ses serveurs de gestion (scheduler, serveurs de stockage, etc.). Ne pouvant être réalisées en avance, certaines tâches d'administration de l'IaaS sont provoquées par celles des entreprises (elles seront interprétées comme des opérations de reconfiguration). Parmi les opérations d'administration au niveau IaaS, les plus courantes sont : l'allocation de ressources, le déploiement de ses serveurs et des VM, la configuration et le démarrage (des VM et ses serveurs). Quant aux autres, elles sont

provoquées par des événements externes pour certaines (consolidation, réparation) et régulières pour d'autres (monitoring).

3.1.1 Allocation de ressources

C'est la première opération réalisée dans le cloud. Elle attribue à l'entreprise cliente sa portion de ressources exploitables. Par ressources, nous regroupons à la fois la mémoire, le processeur, l'espace de stockage, la bande passante et les équipements informatiques. Mutualisées entre tous les utilisateurs, les ressources représentent le point de rentabilité pour le fournisseur. Ainsi, la conception et l'implémentation des politiques d'allocation dans le cloud dépendent de la stratégie commerciale du fournisseur. Notons que l'allocation est provoquée entre autres par une réservation de l'entreprise. Le fournisseur peut dont proposer plusieurs manières de réservation :

1. Réservation pour une durée indéterminée : dans ce mode, un contrat est établi avec l'entreprise pour une durée infinie et continue (on peut également parler de forfait).
2. Réservation pour une utilisation à venir et pour une durée limitée : dans ce mode, la difficulté se situe dans la gestion des plages de réservation. On retrouve le problème très connu et complexe qui est celui de la planification.
3. Réservation pour une utilisation immédiate et limitée dans le temps : dans ce cas, les ressources requises doivent être disponibles dans l'immédiat.

La prise en compte de ces modes de réservation peut complexifier l'allocation dans le cloud. Notamment, il peut être amené à mettre en place des files d'attente, avec des notions de priorité. Ainsi, en guise d'exemple, les ressources obtenues par le dernier mode de réservation peuvent être considérées comme moins prioritaires que celles obtenues via les deux premiers. Dans ce cas, le cloud doit être capable d'identifier les ressources à libérer en cas de besoin d'une application plus prioritaire.

3.1.2 Déploiement

Le déploiement dans l'IaaS concerne essentiellement les systèmes de fichiers des machines virtuelles. Il est réalisé à deux occasions. Premièrement (flèche (2) de la figure 3.2) lors de l'enregistrement dans le cloud des systèmes de fichiers d'OS (également appelés images) et des données de l'entreprise. Pour cela, le cloud utilise un serveur de stockage (que nous appelons *RepositoryController* dans la figure 3.2) distinct des lieux d'exécution des VM. Le second déploiement (flèche (4) de la figure 3.2) intervient à l'exécution de celles-ci. En effet, l'image utilisée pour l'exécution d'une VM dans le cloud est une copie de l'image initiale. Ceci s'explique par deux raisons. La première est la possible inaccessibilité des machines hôtes (hébergeant les VM) au serveur de stockage. Très souvent, pour des raisons d'efficacité, le format de stockage des données (y compris les systèmes de fichiers) par le serveur de stockage peut être différent de celui attendu par le système de virtualisation qui exécutera la

VM. C'est le cas dans le cloud Amazon EC2 [?]. La deuxième raison est la possible utilisation de la même image pour l'exécution de plusieurs VM. De plus, l'entreprise peut souhaiter retrouver l'image originale après l'exécution de la VM (sauf demande contraire).

3.1.3 Configuration et Démarrage

L'opération de démarrage des machines virtuelles nécessite préalablement leur configuration. Cette dernière dépend d'une part des demandes de l'entreprise et d'autre part de la politique d'allocation de ressources dans le cloud. Pendant la phase de réservation, une entreprise souscrit pour des VM dont elle fournit les caractéristiques au cloud. Ces caractéristiques concernent : la quantité de mémoires, le nombre de processeurs, le lieu géométrique d'exécution de la VM et le système de fichiers représentant l'OS de la VM. Quant aux contraintes de configuration venant de l'IaaS, il s'agit des configurations réseaux. En effet, l'IaaS peut implémenter plusieurs configurations d'accès réseaux : l'attribution d'un réseau virtuel (VLAN) aux VM appartenant à la même entreprise ou encore l'utilisation d'un VLAN commun pour toutes les VM (il ne s'agit que d'exemples). Il doit également configurer les contrôles réseaux (pare feu ou encore les quotas d'utilisation réseau) en fonction des souscriptions de l'entreprise.

3.1.4 Reconfiguration

Comme nous l'avons évoqué dans le préambule de cette section, la plupart des tâches d'administration au niveau de l'IaaS peuvent être interprétées comme des opérations de reconfiguration (réalisées par le "*VMController*" sur la figure 3.2). En effet, elles interviennent pendant l'exécution de l'IaaS et modifient sa composition. Dans cette section, nous présentons quelques tâches de reconfigurations particulières, liées essentiellement à la gestion des VM : la consolidation et la réparation.

Consolidation de VM

Nous nous rappelons des plateformes d'hébergement dans lesquelles des machines entières étaient dédiées à un client. Dans ces plateformes, aucune tâche d'administration de la part du fournisseur n'était envisageable une fois la machine allouée au client (au risque de violer l'exclusivité d'utilisation de la machine par le client). A l'inverse, les clouds basés sur la virtualisation offrent une marge à l'administrateur de l'IaaS pour une intervention sur les machines physiques. Cette possibilité est offerte par le caractère isolant de la virtualisation. Elle permet entre autres au fournisseur d'implémenter différentes politiques d'allocation ou redistribution de ressources, dans le but d'effectuer des économies ou de respecter un contrat client.

La première intervention dans la gestion de ressources survient pendant la phase

d'allocation de VM (elle est également dite phase de placement). Durant cette phase, l'IaaS doit être capable d'identifier l'ensemble des machines physiques correspondant à la politique d'allocation implantée par le fournisseur. Pour illustrer cela, prenons une politique qui consiste à choisir les machines de telle sorte que le risque de gaspillage soit le plus faible possible. Cette contrainte peut entraîner le déplacement des VM en cours d'exécution afin de libérer de la place pour la VM allouée. On parle de consolidation de VM. Par exemple prenons le cas d'un IaaS constitué de trois machines physiques MP_1, MP_2 et MP_3 de puissance identique notée p. Supposons que MP_1 et MP_2 utilisées respectivement à moitié de leur puissance. Soit un client ayant besoin d'une machine virtuelle dont la puissance requise est $\frac{3}{4}p$. Dans cette situation, au lieu de faire recours à la troisième machine virtuelle, l'IaaS doit être capable de regrouper les deux machines virtuelles des deux machines MP_1 et MP_2 sur la machine MP_1 ou MP_2 et d'allouer par la suite l'une des machines libérées à la nouvelle machine virtuelle.

Notons que la consolidation peut également s'effectuer en dehors des opérations d'allocation. En effet, l'IaaS scrute régulièrement son environnement et réorganise la disposition des VM sur les machines afin de libérer certaines. Cette libération de machines permet de réduire les taux de consommation énergétique de l'IaaS.

Réparation de pannes

Compte tenu de la taille du cloud et de la multitude d'utilisateurs et du nombre de clients qu'il accueille, le risque d'apparition de pannes est important. L'apparition d'un dysfonctionnement doit être rapidement identifiée et traitée par l'administrateur afin de ne pas pénaliser les entreprises. L'une des pannes les plus critiques dans l'IaaS est le dysfonctionnement d'une machine physique ou virtuelle. En effet, elle affecte les applications des entreprises. A cause de sa non intrusivité, l'IaaS n'est pas au courant des logiciels en cours d'exécution dans les VM qu'il héberge. De ce fait, l'IaaS ne saurait résoudre efficacement une panne sans la collaboration de l'entreprise concernée. Malgré cette limitation, l'IaaS peut profiter des atouts de la virtualisation et proposer ainsi plusieurs politiques de réparation. Celles-ci peuvent aller des plus simples (redémarrage de la VM concernée) aux plus sophistiquées (redémarrage sur le dernier point de sauvegarde). L'application de ces politiques peut dépendre des termes du contrat souscrit par l'entreprise. Dans tous les cas, la mise en place de ces politiques dépend du système de surveillance implanté dans l'IaaS.

3.1.5 Monitoring

Les sections précédentes ont introduit la notion de monitoring. En effet, toutes les tâches d'administration dans le cloud dépendent des informations obtenues par monitoring des environnements matériels, virtuels et logiciels (réalisé par "*MonitoringController*" sur la figure 3.2). Parmi les informations de monitoring qui nous intéressent, nous pouvons citer le taux d'utilisation des processeurs, des disques, du réseau, de la mémoire, etc. Cependant, l'architecture particulière du cloud et des ap-

plications qui y sont exécutées constitue un facteur limitant pour le monitoring. En effet, réputé comme une tâche complexe dans les environnements distribués constitués de machines réelles, le monitoring dans le cloud est un véritable challenge.

Les ressources étant virtualisées dans le cloud, il n'est pas évident de fournir une image reflétant l'état réel des machines. En effet, nous distinguons deux niveaux d'observation et d'analyse de ressources : la machine virtuelle et la machine physique. D'une part, l'IaaS doit être capable de séparer les ressources consommées par chaque niveau afin de fournir aux clients des informations relatives uniquement à leur consommation. D'autre part, l'IaaS doit fournir à son administrateur les informations concernant à la fois les VM et les machines les hébergeant. Dans les deux cas, les informations doivent être compréhensibles et exploitables.

Jusqu'à présent, nous n'évoquons que des informations locales à chaque machine physique ou virtuelle. Or le cloud est un système réparti et hétérogène. Dans la plupart des cas, l'administrateur s'intéresse aux informations agrégées et obtenues par calculs groupés des observations locales. Par exemple, ce calcul peut se faire par combinaison des informations provenant d'un ensemble de machines appartenant à la même zone géographique ou à la même entreprise.

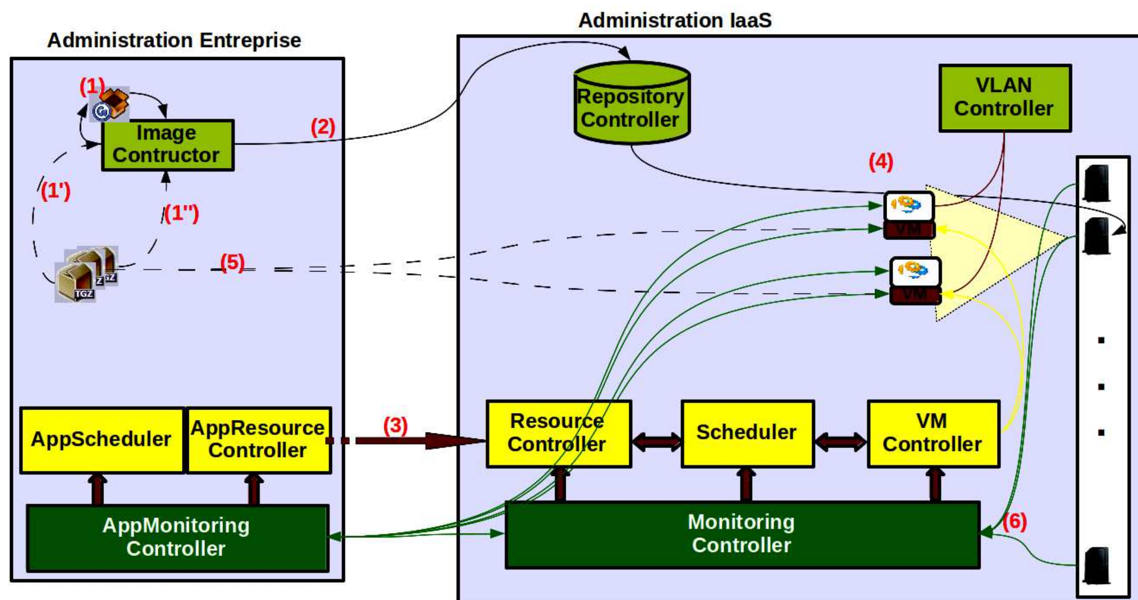


FIGURE 3.2 – Administration dans le cloud

3.2 Administration niveau Entreprise

Malgré les avantages qu'il offre, le cloud peine à séduire les entreprises depuis sa vulgarisation. Cette hésitation est d'ordre idéologique : l'idée de confier ses données (essentielles pour la concurrence) à une entreprise externe n'est pas encore complètement acceptée dans les entreprises. Face cette idée, le cloud répond avec plus de

développement de la sécurité et la confidentialité. Comme en interne, l'administration d'applications sur le cloud est une tâche fastidieuse pour l'entreprise. De plus, dans le cadre des clouds virtualisés, la gestion des VM représente une opération supplémentaire. En somme, l'administration d'applications dans un cloud virtualisé comprend les tâches suivantes :

- Construction de systèmes d'exploitation (ou VM) ;
- Réservation/Allocation de ressources sur le cloud ;
- Installation et démarrage des logiciels ;
- Suivi de l'exécution des logiciels et des VM ;
- Reconfiguration/Optimisation (scalabilité, mise à jour des logiciels, réparation, etc.).

3.2.1 Construction de VM et Déploiement

L'exécution de toute application par une entreprise dans le cloud a lieu dans une VM. L'exécution de cette dernière requiert la présence de son image dans le serveur de stockage du cloud. Pour l'entreprise, le déploiement peut donc s'effectuer à deux occasions : pendant le téléchargement du système de fichiers des VM (flèche (2) sur la figure 3.2) et pendant la copie (de l'entreprise vers les VM) des binaires des logiciels constituant son application (flèche (5) sur la figure 3.2). Comme nous le présentons dans cette section, le second déploiement peut s'appuyer sur le premier.

La première étape avant toute réservation sur le cloud est la construction des systèmes de fichiers. Elle comprend les phases suivantes : (1) installation du système d'exploitation, (2) configuration du système d'exploitation et (3) éventuellement l'installation des packages ou binaires des futurs logiciels. Dans certains cas, les phases (1) et (2) ne sont pas nécessaires lorsque le cloud fournit un système de fichiers répondant aux attentes de l'entreprise. Dans tous les cas, l'entreprise effectue la dernière étape qui est l'installation de ses logiciels. Cette tâche peut s'effectuer de différentes façons.

Méthode 1 : Elle construit un OS contenant tous les binaires et packages de tous ses logiciels (flèches (1), (1') et (1") sur la figure 3.2). La construction s'effectue chez elle et le résultat (un OS de grande taille) est ensuite transféré sur le cloud grâce aux protocoles de sauvegarde qu'il fournit. C'est par exemple le cas avec *Simple Storage Service (S3)* du cloud Amazon EC2. Le bénéfice de cette méthode est la réduction du nombre de systèmes de fichiers sauvegardés dans le cloud (donc des coûts de stockage). Par contre à l'exécution, l'entreprise paye le prix fort. En effet, l'exécution de toute VM à partir de cet OS augmente l'espace de stockage de l'entreprise et donc le coût d'exécution.

Méthode 2 : Construction d'un système de fichiers dédié pour chaque type de logiciels (flèches (1) et [(1') ou (1")] sur la figure 3.2). Les avantages de cette méthode sont les inconvénients de la précédente et inversement. Autrement dit, elle est moins

coûteuse lorsque les VM sont à l'exécution qu'à l'arrêt. En effet, soient t_s la taille d'un système d'exploitation, n le nombre de logiciels constituant l'application, t_{li} la taille du ième logiciel avec $i \in [1 ; n]$. L'espace de stockage dans cette méthode est $n * t_s + \sum_{i=1}^{n} t_{li}$ (avec n le nombre de logiciels) tandis qu'il est de $t_s + \sum_{i=1}^{n} t_{li}$ dans la première méthode.

Méthode 3 : La dernière est une solution intermédiaire aux deux premières (flèches (5) sur la figure 3.2). Le système de fichier ne contient que l'OS minimal à exécuter. Ainsi à l'exécution, l'administrateur effectue les opérations de déploiement et d'installation des logiciels sur ses instances de VM. Elle présente les avantages de la première et de la seconde approche. Par contre, elle est plus technique, fastidieuse et nécessite plusieurs connexions à distance sur les VM. Ainsi, pour une exécution multiple du même logiciel, l'administrateur réalise plusieurs installations de ce logiciel.

3.2.2 Allocation de ressources

L'allocation de ressources pour l'entreprise peut consister en la réservation des lieux de stockage ou encore des machines. La première permet de stocker les images d'OS construites ou des données utilisables par la suite par les VM. Rappelons que ces dernières ne sont pas vues par l'entreprise comme des VM. Pour elle, il s'agit de machines physiques mises à sa disposition par le cloud et dont elle est propriétaire. Cette opération est faite sous forme de contrats passés avec le cloud. L'entreprise souscrit pour plusieurs ressources de capacités identiques ou non (en terme de processeur, mémoire, bande passante, etc). Cette réservation se traduit au niveau de l'IaaS par le démarrage des VM correspondantes et par la transmission à l'entreprise des informations de connexion (flèche (3) sur la figure 3.2). La fin du contrat entraîne l'arrêt des VM et la libération des ressources qui leur étaient allouées.

3.2.3 Configuration et Démarrage

Une fois les VM démarrées et les binaires des logiciels déployés, l'administrateur de l'entreprise planifie la configuration et le démarrage des logiciels. Ces opérations nécessitent des accès multiples aux VM via les adresses IP ou nom DNS des VM. Elles dépendent des logiciels administrés. Dans certains cas, le démarrage peut nécessiter la configuration des liens de communication entre logiciels situés sur des VM différentes. A cause de la multitude des logiciels, ces opérations sont souvent sources d'erreurs. De plus, certaines applications imposent un ordre de démarrage entre les logiciels. C'est le cas par exemple des serveurs Tomcat qui doivent être démarrés avant les serveurs web Apache dans une application J2EE.

3.2.4 Reconfiguration

Comme pour l'IaaS, les opérations de reconfiguration réalisables par l'entreprise peuvent être dénombrées à l'infini (réalisées par "*Scheduler d'applications*" sur la figure 3.2). Dans cette section, nous présentons deux opérations particulières, très courantes dans l'administration des applications distribuées :

Scalabilité
Une fois l'application démarrée, l'entreprise doit être capable de suivre l'état des différents logiciels (voir la section suivante sur le monitoring) afin d'intervenir en cas de nécessité. Par exemple, l'observation d'une montée en charge au niveau d'un tiers J2EE incitera l'administrateur à réserver une nouvelle VM et effectuer tout le processus de démarrage d'une nouvelle instance du logiciel concerné afin d'absorber le surplus de charge. Inversement, il pourra retirer des instances de serveurs lorsque celles-ci sont sous utilisées. L'ensemble de ces deux opérations permettra de réaliser des économies (par diminution du nombre de VM en utilisation dans le cloud). Cette pratique est communément appelée ***scalabilité*** ou ***principe de la croissance à la demande***.

Réparation
Comme en interne, deux types de pannes peuvent survenir durant l'exécution de l'application : (1) une panne machine (VM ici) ou (2) une panne logicielle. Ne disposant d'aucune action réparatrice sur ses VM, l'entreprise peut souscrire (auprès du cloud) pour un contrat de réparation en cas de (1). Dans ce cas, l'entreprise peut être amenée à redéployer et démarrer les logiciels qui s'exécutaient sur la VM avant la panne. Si l'IaaS implante la sauvegarde régulière des VM, l'entreprise n'a aucune réparation à effectuer. Quant aux pannes de type (2), leurs réparations dépendent de l'application. Elles sont de ce fait hors de la portée du cloud. Dans certaines applications, une panne de type (1) ou (2) peut entraîner la reconfiguration entière de l'application. C'est le cas de l'application J2EE dans laquelle une panne du logiciel Tomcat nécessite le redémarrage du logiciel Apache.

3.2.5 Monitoring

La plupart des informations de monitoring recueillies à ce niveau proviennent directement des API de l'IaaS (liaison entre "*Monitoring d'applications*" et "*Monitoring de l'IaaS*" sur la figure 3.2). Cependant, dans certains cas, ces informations sont insuffisantes. Par exemple, la détermination de l'état de surcharge d'un serveur de base de données est plus significatif lorsqu'on observe son temps de réponse au lieu de la charge CPU de la machine qui l'héberge. Pour ces cas particuliers, l'administrateur introduit dans son application, des logiciels particuliers jouant le rôle de sonde (liaisons entre "*Monitoring d'applications*" et les VM sur la figure 3.2). Les informations récupérées par ces sondes sont analysées de façon individuelle ou groupée et servent à la prise de décision (pour des reconfigurations).

3.3 Synthèse : système d'administration autonome pour le cloud

En résumé, l'administration dans le cloud est essentiellement liée à son exploitation niveau IaaS et niveau entreprise. Dans les deux niveaux, nous retrouvons des opérations d'administration du même type. Elles sont proches de celles rencontrées dans les grilles de calcul et les environnements distribués. Au niveau IaaS, l'administration consiste en la gestion des ressources. De plus l'utilisation de la virtualisation implique des tâches d'administration supplémentaires qui lui sont propres. Quant à l'entreprise, ses tâches d'administration sont à l'origine des reconfigurations au niveau de l'IaaS. Elles sont pour la plupart propres aux applications hébergées.

Devant la multitude de ces tâches, fournir une solution figée serait limitée et inappropriée dans certaines situations. De plus, ni le cloud, ni l'entreprise ne peut s'offrir les services d'administrateurs humains qui assureront de façon permanente et continue (à longueur de journée) toutes ces tâches. Ainsi, comme nous l'avons effectué dans l'administration des systèmes distribués (grilles et cluster), nous proposons dans cette thèse l'utilisation d'un système d'administration autonome pour améliorer l'administration dans le cloud. Ce système doit être utilisable par les deux types d'intervenants. Pour cela, il doit globalement remplir les critères suivants :

- **Interopérable** : capacité à dialoguer avec d'autres systèmes de gestion de cloud et également des systèmes d'administration des applications qu'il héberge.
- **Auto-réparable** : capacité à prendre en compte les pannes dans l'environnement qu'il administre (machines, VM et logiciels).
- **Extensible** : capacité à prendre en compte de nouveaux éléments dans l'environnement (nouvelles machines réelles et virtuelles, nouveaux logiciels).
- **Programmable** : capacité à prendre en compte de nouvelles politiques d'administration (nouvelle méthode de consolidation, de scalabilité, etc.).
- **Adaptable** : capacité à prendre en compte le remplacement ou la modification d'un de ses modules afin de s'appliquer à d'autres plateformes de cloud.
- **Langages dédiés** : propose des facilités d'utilisation par le biais des langages d'administration proches des compétences des intervenants.

Dans le chapitre suivant, nous montrons en quoi consiste l'administration autonome et dans quelle mesure l'administration autonome peut être utilisée dans cette problématique.

Chapitre 4

Administration Autonome

Contents

4.1 Définition **32**
4.2 Objectifs **33**
4.3 Bénéfices pour les entreprises **34**
4.4 Classification **35**
4.5 Synthèse **36**

Nous constatons depuis plusieurs années la complexité croissante des infrastructures et systèmes informatiques. Cette complexité est due à plusieurs facteurs tels que la diversification des supports informatiques, la mobilité des utilisateurs, la multiplication des besoins et des logiciels, et l'augmentation du nombre d'utilisateurs d'outils informatique, etc. La multiplication de ces facteurs oriente la construction des nouveaux systèmes informatiques par combinaison de plusieurs autres systèmes de natures différentes. De ce fait, leurs administration peut devenir un véritable obstacle à leur pérennité. Elle peut devenir source d'erreurs et être très consommatrice en ressources humaines. Dans cette section, nous présentons une solution consistant à confier cette tâche à un autre système informatique (on parle d'administration autonome).

4.1 Définition

Un Système d'Administration Autonome est un système informatique conçu pour l'administration d'autres systèmes informatiques (matériels et logiciels). Dans la suite de ce document, nous utilisons l'acronyme SAA pour désigner un Système d'Administration Autonome. Le fonctionnement d'un tel système repose sur quatre modules essentiels (figure 4.1) :

- **Le système de représentation** : Il maintient une image de l'exécution réelle de l'application administrée.

- **Les sondes** : Elles représentent en quelque sorte l'œil du SAA sur les machines hébergeant les logiciels qu'il administre.
- **Un module décisionnel** : En fonction des observations faites par les sondes et de l'état courant de l'application, le SAA peut prendre des décisions conformément aux prévisions de l'administrateur humain.
- **Les effecteurs/actionneurs** : Contrairement aux sondes, ils permettent au SAA de réaliser effectivement les opérations d'administration. Ils interviennent à la demande du SAA et doivent laisser le système dans un état conforme au système de représentation. Leur intervention peut modifier le comportement de l'application administrée.

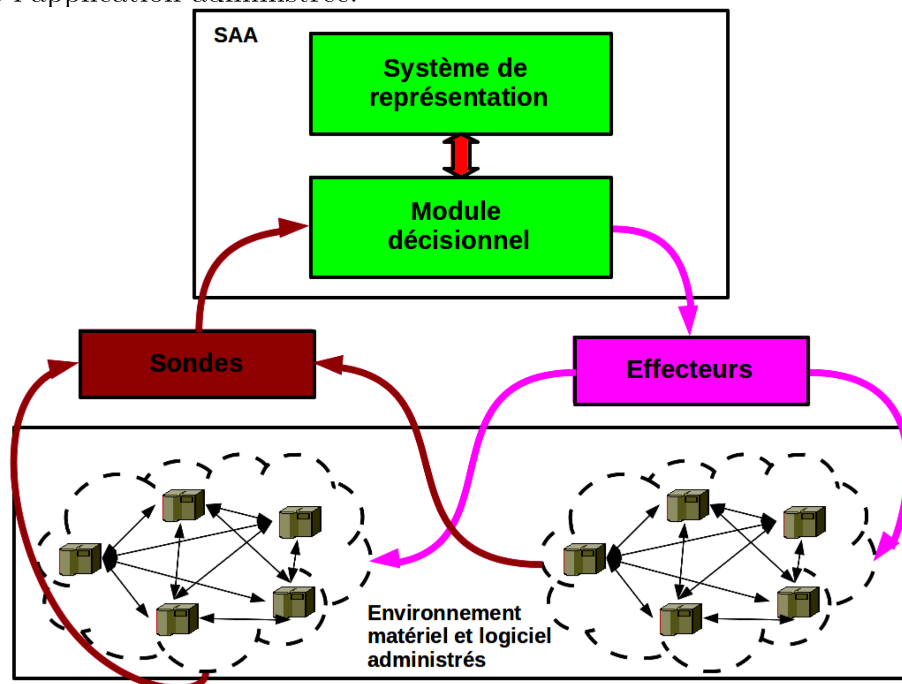


FIGURE 4.1 – Organisation d'un SAA

4.2 Objectifs

Réalisée par des humains, l'administration de systèmes complexes comporte des limites. Par systèmes complexes, nous entendons ceux faisant intervenir plusieurs types de logiciels et s'exécutant dans des environnements hétérogènes et distribués (cas d'un environnement de cloud). Parmi ces limites, nous pouvons citer :

- Le manque de réactivité dans la réalisation des tâches : par exemple en réponse à une panne ou configuration d'un système constitué de plusieurs logiciels.
- Le gaspillage de ressources. C'est une conséquence du manque de réactivité. Dans le but de prévenir des fortes demandes en ressources, l'administrateur à tendance à sur-évaluer celles-ci. Cette sur-évaluation lui permet d'avoir une marge de temps avant l'intervention.

En réponses à ces limites, le développement de systèmes informatiques jouant le rôle d'administrateur est une solution éprouvée à travers plusieurs travaux de recherche. Pour cela, le SAA prend généralement en compte tout le cycle de vie de l'application administrée à savoir [?] [?] [?] :

Le déploiement. Il consiste en l'allocation de ressources matérielles suivie de leur initialisation. Nous entendons par ressources toutes les ressources mémoire, CPU, disques, réseau et machines nécessaires au fonctionnement de l'application. L'initialisation de la machine consiste au ravitaillement de celle-ci en fichiers ou binaires constituant l'application.

La configuration. Elle consiste à définir les paramètres requis par l'application pour son fonctionnement. Elle se fait généralement par la modification des fichiers ou encore le positionnement de certaines variables.

Le démarrage et l'arrêt. Il s'agit de l'exécution des différentes commandes mettant en marche ou non l'application. Comme nous l'avons évoqué dans la section 3.2.3, le démarrage peut s'avérer quelques fois ordonné.

La reconfiguration dynamique. Il s'agit des opérations survenant durant l'exécution de l'application. Elle peut être causée par plusieurs facteurs : l'apparition d'une panne, l'essoufflement d'un logiciel faisant partie de l'application, l'apparition d'une mise à jour, le désir d'optimisation, etc. Dans la littérature, les opérations de configuration, de démarrage et d'arrêt peuvent être présentées comme des tâches de reconfiguration.

La prise en compte de ces tâches par les systèmes d'administration autonome justifie son adoption dans des entreprises comme solution d'administration. Dans la section suivante, nous présentons les bénéfices d'une telle pratique dans une entreprise.

4.3 Bénéfices pour les entreprises

Les bénéfices de l'administration autonome en entreprise sont aussi évidents que ceux de l'informatique dans une entreprise qui n'en dispose pas. En effet, conçu pour remplir les fonctions d'un administrateur humain, un SAA permet à l'entreprise de réaliser des économies :

Gain de temps. L'administration assurée par l'humain implique des interventions évaluées en minutes, heures et voie jours. Or l'utilisation d'un SAA fait passer les temps d'intervention à la seconde ou milliseconde dans certains cas. Cette différence peut être facilement observée dans les tâches de déploiement des applications grande échelle (centaines de logiciels).

Réduction des ressources. Le remplacement des administrateurs humains par un système informatique permet dans un premier temps à l'entreprise de réduire son effectif. Cette réduction implique moins de dépenses salariales. De plus, comme nous l'avons évoqué dans la section précédente, l'introduction d'une administration autonome implique moins de gaspillage en ressources matérielles. En effet, l'entreprise n'étant plus limitée par la lenteur de l'humain et donc de sa surestimation, les ressources sont allouées et utilisées par nécessité. Cette réduction de ressources implique également une économie en énergie électrique.

Adaptabilité des logiciels. L'administration autonome permet de rendre adaptable les logiciels n'ayant pas été conçus à cet effet. C'est le cas des logiciels tels que Apache [**?**] ou Tomcat [**?**]. On dit d'un système qu'il est adaptable lorsqu'il est capable de prendre en compte des modifications de son environnement ou encore de son fonctionnement. L'adoption d'un SAA permettra de prendre en compte ces modifications. Il peut s'agir d'une adaptation à une surcharge de travail d'un logiciel. Nous pouvons également citer le cas de la réparation d'une panne logicielle.

4.4 Classification

Depuis la pose des prémisses de cette technologie par IBM en 2003, plusieurs projets de construction de SAA ont vu le jour. Ces développements s'effectuent suivant différentes orientations. Dans cette section, nous proposons quelques critères permettant de les regrouper par familles.

Centralisé ou distribué. L'efficacité (en terme de temps d'exécution) d'un SAA est fortement liée à son mode de fonctionnement : centralisé ou distribué. Un SAA en fonctionnement centralisé implique une gestion en un unique point. Ce fonctionnement correspond à l'administration des systèmes de taille réduite (à cause du risque d'un goulot d'étranglement). Quant au fonctionnement distribué, l'administration peut être effectuée à n'importe quel lieu de l'environnement. Dans le cadre des systèmes à large échelle (des milliers de logiciels et de machines), un SAA centralisé atteint ses limites et laisse la place aux SAA distribués. Ces derniers incluent à la fois les SAA hiérarchisés et non hiérarchisés. Dans le premier cas, chaque point d'administration à la responsabilité d'un morceau de l'application administrée. En cas d'impossibilité d'administration, la tâche est transmise au point supérieur qui possède une vision plus étendue [**?**]. Dans le second cas par contre, les SAA sont équivalents et doivent collaborer afin d'avoir des états identiques.

Adaptable ou non. Les SAA apportent de l'adaptabilité aux infrastructures qu'ils administrent. Cela ne les empêche pas de posséder également les mêmes facultés. Un SAA adaptable donne la possibilité à l'utilisateur de remplacer des modules ou des fonctions de base de l'administration. Par exemple, les méthodes de déploiement ou d'accès à distance sur les machines administrées peuvent être remplacées par d'autres fonctions propres à l'utilisation courante.

Générique ou spécifique. La plupart des SAA existants sont conçus pour des applications particulières (exemple de Proactive [?] pour les applications MPI). Ceci implique que le moindre changement de l'application administrée nécessite la réimplantation du SAA.

4.5 Synthèse

A la lumière de cette présentation, nous constatons dans un premier temps que le cloud fait partie des infrastructures dites complexes : plusieurs utilisateurs, plusieurs types d'applications, plusieurs technologies mises en commun, environnements distribués et large échelle. Dans un second temps, nous constatons que les tâches d'administration dans le cloud (section 3) correspondent parfaitement à celles auxquelles répond l'administration autonome. A cause de sa nouveauté, les solutions d'administration autonome n'ont pas fait l'objet de plusieurs expérimentations dans le cadre du cloud. Dans la suite de ce document, nous explorons l'utilisation de SAA pour l'administration dans le cloud. Notre étude s'appuie sur le SAA TUNe [?] développé au sein de notre équipe. Le chapitre suivant présente les principes fondateurs de ce système.

Chapitre 5

TUNe

Contents

5.1 Historique **38**

5.2 Principes **39**

5.2.1 Architecture Description Language (ADL) 39

5.2.2 Wrapping Description Language (WDL) 41

5.2.3 Reconfiguration Description Language (RDL) 42

5.3 Choix d'implantation **43**

5.4 Expérimentations et Problématiques **46**

5.4.1 Applications cluster 46

5.4.2 Applications large échelle : cas de DIET 47

5.4.3 Applications virtualisées 48

5.4.4 Cloud 50

5.5 Synthèse et nouvelle orientation **52**

Exerçant dans le domaine de l'administration autonome depuis plusieurs années maintenant, l'équipe dans laquelle ces travaux ont été réalisés a développé autour du projet TUNe [?], un SAA baptisé du même nom. Avant de présenter ses principes fondamentaux, nous revenons sur son historique. La fin de ce chapitre présente les différentes expériences réalisées avec ce système, ses limitations ainsi que sa nouvelle orientation.

Pour des fins d'illustration, tout exemple dans ce chapitre se base sur l'administration d'une application J2EE. Il s'agit d'une application web dynamique organisée suivant une architecture n-tiers (figure 5.1) composée des logiciels suivants :

- Apache [?] : le serveur web qui a pour but de servir des documents statiques. Dans certains cas, un loadbalancer (HA-Proxy [?] par exemple) peut être associé à plusieurs serveurs Apache afin d'absorber une quantité importante d'utilisateurs ;
- Tomcat [?] : le serveur d'applications qui, à la demande de Apache (via son connecteur ModJK [?] par exemple), effectue des traitements pour la construction d'une page web ;

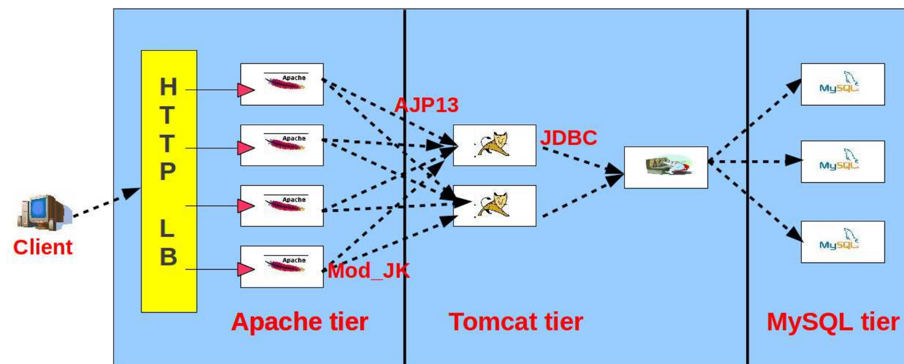


FIGURE 5.1 – Exemple d'une architecture J2EE

– MySQL [?] et MySQL-Proxy [?] : ce dernier permet de relier le serveur d'applications à plusieurs sources de données MySQL. En cas de besoin, le serveur d'applications fait appel au serveur de données MySQL pour le retrait ou la sauvegarde de données venant des clients web.

5.1 Historique

Le projet TUNe voit le jour en 2008 au sein de l'équipe Astre du laboratoire IRIT (Institut de Recherche en Informatique de Toulouse). Il s'inscrit dans la continuité des projets SELFWARE [1] et SCOrWare [2]. L'objectif principal du projet SELFWARE est le développement d'une plateforme logicielle pour la construction de systèmes informatiques répartis sous administration autonome, et son application à deux domaines particuliers : l'administration de serveurs d'applications J2EE et l'administration d'un bus d'information d'entreprise. Quant au projet SCOrWare, il a pour ambition de fournir une implantation en logiciel libre des récentes spécifications Service Component Architecture (SCA) définies par l'Open SOA Collaboration. L'objectif de TUNe dans ces projets est la proposition d'une nouvelle plateforme d'administration palliant aux limites de son prédécesseur Jade [?]. Ce dernier a été développé au sein de l'INRIA dans le projet SARDES [3] à Grenoble. Jade est l'un des premiers SAA offrant les fonctionnalités d'administration telles que décrites par [?] (déploiement, configuration et reconfiguration). Sa grande particularité est l'administration de logiciels patrimoniaux, c'est-à-dire ceux n'ayant pas été conçus avec des facilités d'administration. En plus de fournir un support pour l'administration des logiciels dans un environnement réparti, il prend également en compte la supervision de l'environnement matériel administré. Il permet entre autre de définir des enchaînements d'actions exécutables (de façon autonome) en réponse à un événement particulier dans l'environnement administré (comme des pannes ou des surcharges). Alors qu'il a permis de valider l'apport des SAA et l'approche de développement de SAA à base de composants, Jade s'adresse cependant à des uti-

1. Le projet SELFWARE : http ://sardes.inrialpes.fr/ boyer/selfware/index.php
2. Le projet SCOrWare : http ://www.scorware.org/
3. Le projet SARDES : http ://sardes.inrialpes.fr/

lisateurs avertis du domaine à composants. En effet, il suppose que les utilisateurs maîtrisent les techniques de programmation à base de composants, notamment Fractal [?]. Dans ce contexte, le projet TUNe a donc pour mission d'implanter sous les bases de Jade, un SAA dont l'utilisation est moins contraignante. Ainsi, TUNe implante des langages d'administration de haut niveau, basés sur les profils UML [?], donc plus accessibles aux utilisateurs non initiés à Fractal. Dans la section suivante, nous présentons ces langages et leur utilisation dans TUNe.

5.2 Principes

La conception et l'implantation du système TUNe repose sur une approche basée composants. Approche très prometteuse, cette dernière est également au cœur de la conception d'autres SAA tels que Rainbow [?]. Le principe général est d'encapsuler les éléments administrés dans des composants et d'administrer l'environnement logiciel comme une architecture à composants. Ainsi, le SAA bénéficie des atouts du modèle à composants utilisé tels que l'encapsulation, les outils de déploiement et les interfaces de reconfiguration pour l'implantation des procédures d'administration autonome. Grâce à cette approche, le SAA offre une vision uniforme de l'environnement logiciel et matériel à administrer. Pour ce qui est du projet TUNe (comme Jade), nous utilisons le modèle à composants Fractal [?].

Contrairement à Jade, TUNe masque la complexité liée à la maîtrise des API de programmation du modèle à composants et fournit des langages de haut niveau. Ces derniers permettent notamment la description de l'administration de systèmes large échelle tout en minimisant les difficultés rencontrées avec Jade. Basé sur les profils UML [?] (largement utilisés) et un langage de navigation architectural, TUNe fournit trois langages d'administration. Ces langages permettent à la fois la description des applications, de l'environnement matériel et des politiques de reconfiguration dynamique. Les sections suivantes n'ont pas vocation à présenter en détails le système TUNe et son implantation. Pour plus d'informations, le lecteur peu se reporter vous à la thèse [?].

5.2.1 Architecture Description Language (ADL)

Le processus d'administration avec TUNe débute par la description de l'application à administrer. Comme son ancêtre Jade, TUNe prend en compte l'administration d'applications distribuées s'exécutant dans un environnement réparti. L'ADL fournit dans TUNe permet de décrire à la fois l'application et l'environnement matériel dans lequel l'application s'exécute. Basé sur le diagramme de classe UML [?],

TUNe permet l'utilisation des outils de modélisation graphique de l'IDM [4] pour la définition des éléments administrés.

Concernant la description de l'application, chaque type de logiciel de l'application est représenté par une classe UML. L'assignation d'attributs à la classe permet d'indiquer les propriétés du type de logiciel qu'elle représente. En plus des propriétés propres au type de logiciel décrit, TUNe impose des attributs particuliers pour son utilisation interne (les attributs en rouge sur la figure 5.2). Dans la figure 5.2(a), nous décrivons 4 types de logiciels constituant une application J2EE à administrer. L'attribut "*legacyFile*" présent dans toutes les classes indique à TUNe l'archive contenant les binaires du logiciel concerné. Par contre, l'attribut "*port*" du type Apache par exemple représente une propriété propre aux serveurs web Apache.

En plus de la description des logiciels, TUNe permet la description des interconnexions entre les types de logiciels. Le nommage de ces interconnexions leur permet d'être utilisées dans les autres langages de TUNe pour désigner des logiciels d'un bout de l'application à un autre (exemple de la liaison "*workers*" entre Apache et Tomcat). Les sections suivantes présentent en détail cette utilisation. De façon générale, cette première étape de description par ADL permet de poser les bases pour les autres langages.

De façon analogue, TUNe permet l'utilisation des diagrammes de classes UML pour la description de l'environnement matériel dans lequel s'exécute l'application administrée. TUNe considère l'environnement matériel comme un ensemble constitué de groupes de machines (clusters). Chaque cluster représente un ensemble de machines homogènes (OS, systèmes de fichiers et toutes autres caractéristiques identiques). Ainsi, dans la description de l'environnement matériel, chaque classe représente un cluster. Contrairement aux logiciels, tous les attributs d'un cluster sont imposés par TUNe. En guise d'exemple, l'attribut "*nodefile*" de la classe *Cluster* (figure 5.2(b)) indique un nom de fichier contenant la liste des machines constituant le cluster. Quant à l'attribut "*allocator-policy*", il définit la politique d'allocation de machines (aux logiciels) dans ce cluster. Les interconnexions décrites ici entre les clusters sont des relations d'héritage d'attributs. Contrairement à l'utilité qu'ont les liaisons entre les éléments logiciels, les liaisons d'héritage servent uniquement à factoriser la description des attributs entre les clusters. Rappelons que tous les attributs définis par les clusters sont imposés par TUNe. L'administrateur ne peut introduire de nouveaux attributs.

Description intuitive

La première particularité de l'ADL proposée par TUNe est la description des types de logiciels. Contrairement aux SAA (comme [?], [?], [?], [?] par exemple) dans lesquels l'administrateur doit définir la totalité des instances logicielles, TUNe permet la description des types de logiciels. Un type logiciel peut être par la suite instancié en plusieurs logiciels à la demande de l'administrateur (via l'attribut "*initial*", qui représente le nombre d'instances au démarrage, et des opérations de reconfiguration). Cette particularité permettra par exemple de décrire l'instanciation de dix

4. Ingenierie Dirigée par les Modèles : exemple de Topcased [?]

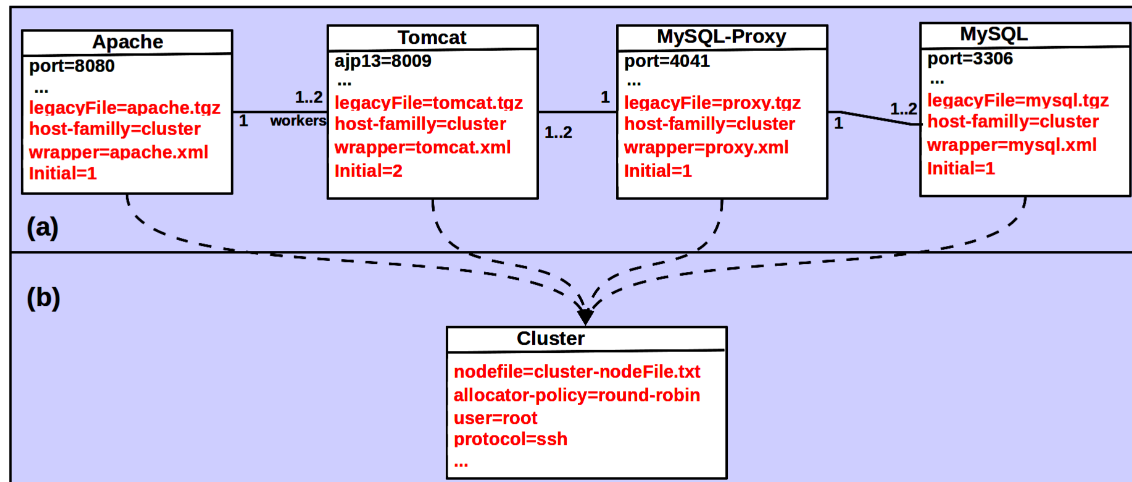


FIGURE 5.2 – Exemple d'ADL : cas d'une application J2EE

Tomcat dans une application J2EE via la description d'un seul type de logiciel Tomcat et la position de son attribut "*initial*" à dix.

Cependant, cette facilité influence la définition des interconnexions entre les types de logiciels. Il revient à TUNe la charge de réaliser les connexions entre les instances logicielles. En effet, TUNe les interprète comme un pattern régissant les liaisons entre les instances de logiciels. Il utilise la sémantique UML définissant les associations entre classes. Les liaisons entre types de logiciels sont ainsi accompagnées de cardinalités. Ces derniers permettent à TUNe de construire et de maintenir un modèle à composants conforme au pattern architectural décrit par l'administrateur. Dans la section 5.3, nous revenons sur l'interprétation de cette description dans TUNe.

Pour finir, le rapport entre la description des types de logiciels et la description de l'environnement matériel est le déploiement. Cette relation est indiquée dans la description des types de logiciels via l'attribut "*host-familly*" (qui désigne un cluster) imposé par TUNe pour chaque type de logiciel (trait en pointillé sur la figure 5.2). Ainsi, toutes les instances appartenant au même type logiciel sont déployées et exécutées sur des machines appartenant au même cluster.

5.2.2 Wrapping Description Language (WDL)

L'ADL fournit par TUNe ne permet qu'une description structurelle de l'environnement d'administration. En ce qui concerne la description des opérations d'administration, TUNe propose un langage basé sur le formalisme XML nommé : Wrapping Description Language (WDL). Dans le reste de ce document, nous utilisons le verbe "*wrapper*" pour décrire l'action de réaliser cette opération. Pour chaque type de logiciel décrit dans l'ADL, l'administrateur définit (attribut *wrapper* de chaque type logiciel dans la figure 5.2(a)) dans un fichier XML l'ensemble des fonctions pouvant être exécutées (exemple de la figure 5.3) par les instances logicielles de ce type.

```
<wrapper name='Apache'>

   <method name="start" key="J2EEWrapping" method="apacheManager" >

      <param value="start" />                    (a)
      <param value="$port" />                    (b)
      <param value="$workers.ajp13" />           (c)

   </method>
   ...
</wrapper>
```

FIGURE 5.3 – Exemple de wrapping : cas du logiciel Apache

Chaque fonction correspond à une méthode java (exemple de méthode *apacheManager* de la classe java *J2EEWrapping* sur la figure 5.3) et est spécifique au type logiciel qui lui est associé.

Le langage de wrapping de TUNe permet une description de méthodes avec passage de paramètres. Compte tenu de la non connaissance des valeurs exactes des propriétés des logiciels pendant la phase de wrapping, les paramètres définis dans le WDL peuvent être de deux types : constante ou variable. Les paramètres de type constante sont ceux dont la valeur reste la même tout au long de l'exécution de l'application et du SAA. Ils ont une sémantique comparable à celle des constantes dans les langages de programmation comme java. C'est le cas du premier paramètre de la fonction "*start*" des instances logicielles de type Apache de la figure 5.3(a). Quant aux paramètres de type variable, TUNe offre un langage de désignation de logiciels ainsi que de ses attributs. Il s'agit d'un langage de navigation architecturale. Autrement dit, il permet de parcourir, à l'aide d'une notation pointée, toute l'architecture suivant les liaisons entre les éléments administrés. Ainsi, la valeur réelle du paramètre ne sera évaluée (par résolution de nom) qu'à l'exécution de la méthode. Les deuxième et troisième paramètres de la figure 5.3 (5.3(b) et 5.3(c)) sont des exemples. Le premier (5.3(b)) fait référence à l'attribut port de l'instance d'Apache courant tandis que le second (5.3(c)) fait référence aux numéros de port des connecteurs AJP [**?**] des serveurs Tomcat liés à l'instance Apache courant (via la connexion nommée *workers*).

L'exécution des fonctions de wrapping dépend de leur présence ou non dans les politiques de reconfiguration. Dans la section suivante, nous présentons le langage permettant de mettre en application les fonctions de wrapping : le langage de reconfiguration.

5.2.3 Reconfiguration Description Language (RDL)

L'exploitation des définitions faites par ADL et WDL est la définition de politiques d'administration. TUNe fournit un langage proche des diagrammes d'état-transition d'UML : baptisé Reconfiguration Description Language (RDL) dans TUNe. Le terme "automate" est également utilisé pour désigner ces diagrammes ou politiques d'administration. Les automates décrits à l'aide de ce langage s'exécutent à

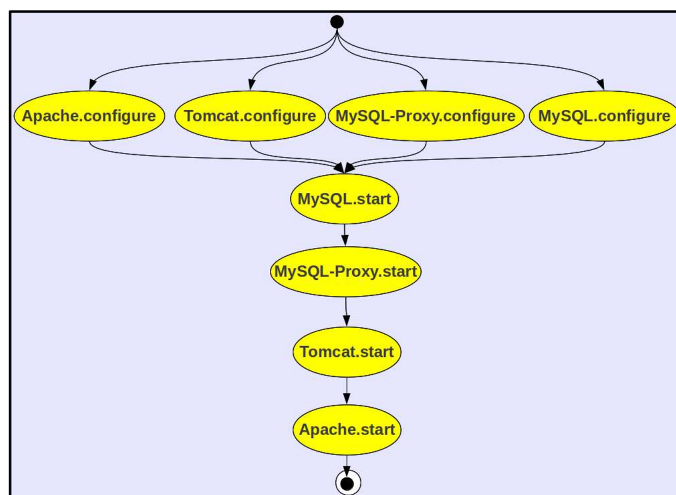


FIGURE 5.4 – Principes de TUNe

chaque apparition d'un événement dans l'environnement administré. Chaque état de l'automate définit une action à exécuter. Deux types d'actions sont possibles : la modification de paramètres et l'appel de fonctions décrites lors du wrapping. En plus des fonctions de wrapping, TUNe met à la disposition de l'administrateur deux fonctions particulières pour l'ajout et le retrait d'instances logicielles de l'architecture. Après l'exécution de ces fonctions, il assure la cohérence entre : (1) les patterns architecturaux définis auparavant par ADL, (2) la représentation interne qu'il dispose (de l'environnement) et (3) l'environnement d'exécution réelle (sur les machines). En effet, l'ajout et le retrait d'instances modifient l'architecture courante de l'application.

Comme son prédécesseur Jade, la description des programmes (cas de Jade) ou automates (cas de TUNe) de (re)configuration dépend des logiciels administrés et des événements attendus par l'administrateur. Ainsi, le nombre de ces politiques dépend des besoins de l'application administrée. Par contre, deux automates particuliers sont requis par TUNe pour l'administration de toute application : l'automate de démarrage et l'automate d'arrêt des logiciels administrés. La figure 5.4 montre un exemple d'automate décrivant à la fois la configuration et le démarrage des serveurs de l'application J2EE. On constate par exemple que les serveurs J2EE peuvent être configurés parallèlement alors que le démarrage par contre doit respecter un ordre (MySQL, MySQL-Proxy, Tomcat et enfin Apache).

De la même façon qu'avec le WDL, la navigation à travers l'architecture (par notation pointée) est utilisable dans la définition des actions.

5.3 Choix d'implantation

L'administration basée composants fournit une couche d'abstraction pour un ensemble d'éléments matériels ou logiciels. S'appuyant sur le modèle à composants Fractal, TUNe construit une architecture à composants interne représentant l'en-

vironnement administré. Cette architecture est conforme à la description par ADL faite de l'application par l'administrateur. Cette structure interne se nomme *Système de Représentation (SR)* dans TUNe. Elle contient à la fois les éléments logiciels et matériels. Chaque instance logicielle est encapsulée[5] dans un composant Fractal. Ce dernier implémente les fonctions de base de l'administration à savoir : le déploiement, le wrapping et l'exécution à distance de fonctions. Les interconnexions entre les instances sont représentées par des liaisons (binding) bidirectionnelles entre les composants Fractal correspondants. Cette bidirectionnalité facilite la navigation par notation pointée (navigation dans deux sens entre deux instances liées). Dans la figure 5.5 les flèches bleues en pointillé (de l'environnement d'édition vers le système de représentation) représentent la relation d'encapsulation entre les types de logiciels et les composants Fractal. Notons que conformément à la valeur de l'attribut "*initial*", l'élément Tomcat génère deux composants Fractal (deux instances du logiciel Tomcat).

Contrairement aux composants logiciels, les machines de chaque cluster ne sont pas représentées par un seul composant Fractal. TUNe représente le cluster par un composant Fractal. Ce composant implémente les fonctions d'allocation et de libération de machines. Les liaisons d'héritage ne sont pas maintenues dans le SR. Le déploiement d'un logiciel dans le cluster entraîne une liaison Fractal entre le composant logiciel représentant ce logiciel et le composant Fractal représentant le cluster. Cette liaison (composant logiciel-composant cluster ; flèches rouges sur la figure 5.5) permet de récupérer les propriétés des machines hébergeant un logiciel.

Afin d'accomplir les tâches d'administration qui lui sont confiées, TUNe exécute sur les machines distantes deux types de serveurs. Le premier serveur (*RemoteLauncher*) permet d'effectuer des actions générales non liées à un logiciel particulier de l'application administrée. C'est notamment lui qui initialise la machine distante avant le déploiement des logiciels. Il démarre également, à la demande de TUNe, les serveurs de second type. C'est (*RemoteLauncher*) le représentant de TUNe sur la machine distante. Contrairement au premier (qui est unique sur la machine), TUNe associe à chaque instance logicielle un représentant (serveurs de second type) sur la machine distante. Ce serveur de second type (*RemoteWrapper*) est chargé de l'exécution des opérations d'administration liées à l'instance logicielle à laquelle il est associé. Il entre en action suite à l'invocation d'une fonction de wrapping issue de l'exécution d'un automate de reconfiguration. C'est à lui que revient par exemple la charge d'exécuter la fonction "*Apache.start*" (de l'automate de démarrage de la figure 5.4) sur une instance du serveur Apache. D'autre part, ce serveur joue le rôle de communicant d'événements. C'est par son biais que les événements repérés par des logiciels de monitoring sont communiqués à TUNe (matérialisés par les flèches vertes de la figure 5.5).

Depuis sa mise en œuvre, cette conception de TUNe a fait l'objet de plusieurs

5. Contrairement à sa signification habituelle dans la littérature (implantation complète du comportement d'un logiciel), l'expression "encapsuler un logiciel" dans ce document signifie : implanter les contrôleurs permettant de manipuler un logiciel existant (considéré comme une boîte noire). Afin d'éviter l'ambiguïté avec le terme "contrôleur" déjà présent dans les modèles à composants, nous utilisons le terme "encapsuler".

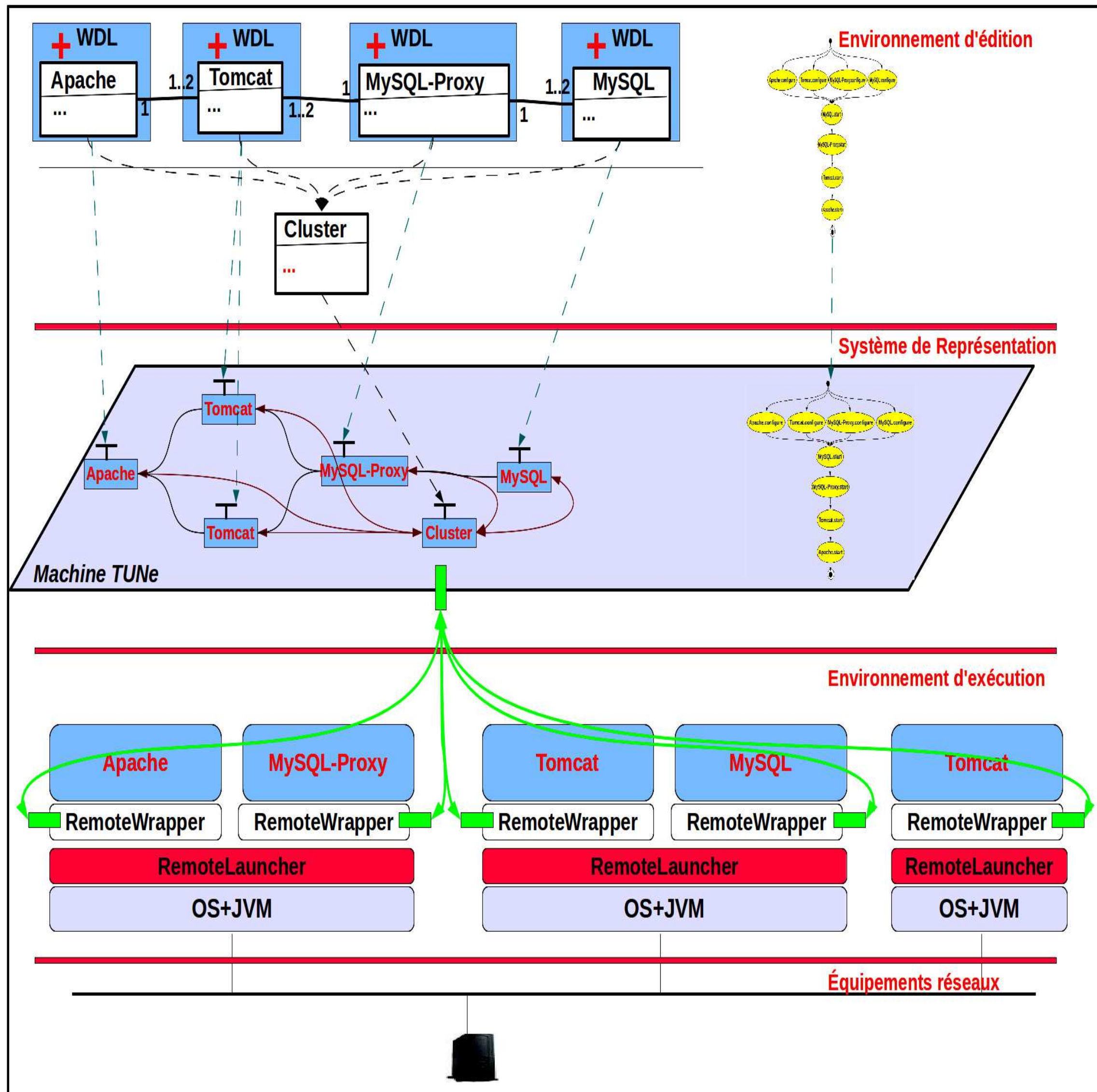


FIGURE 5.5 – Vue globale de TUNe

expérimentations dans divers domaines applicatifs. Comme nous le présentons dans la section suivante, certaines de ces expérimentations ont révélé les limites de cette conception de TUNe.

5.4 Expérimentations et Problématiques

Les expérimentations réalisées avec le système TUNe (par nos partenaires et nous) durant ces dernières années couvrent plusieurs domaines applicatifs. Sans être exhaustif dans la liste de ces expériences, nous décrivons dans cette section quelques unes d'entre elles qui ont motivé la nouvelle orientation du projet TUNe. Il s'agit de l'utilisation de TUNe pour l'administration des applications clusters de type maître-esclave(exemple de J2EE), des applications large échelle et des applications virtualisées. Ces présentations mettent en exergue d'une part l'efficacité du système TUNe et d'autre part ses limites. En plus de ces expérimentations, nous achevons cette section par la présentation des limites liées à la tentative d'utilisation de TUNe pour l'administration du cloud dans son ensemble. La complexité de ce dernier nous permet de compléter l'analyse faite des limitations de TUNe.

5.4.1 Applications cluster

Le système TUNe a été conçu pour l'administration des applications réparties. Pour des besoins de développement, il utilise des applications de type J2EE (figure 5.1) comme support de test. De ce fait, ses langages d'administration répondent parfaitement avec les besoins d'administration attendus par ce type d'applications. Parcourons à présent ces besoins afin de montrer l'efficacité de TUNe.

Concernant l'installation des serveurs J2EE sur les machines distantes, TUNe propose une méthode de déploiement consistant à copier des binaires de la machine d'administration vers les machines distantes. Ces binaires doivent être soumis à TUNe sous forme d'archives. L'attribut "*legacyFile*", imposé par TUNe pour chaque type de logiciel, permet à l'administrateur d'indiquer l'archive d'installation du logiciel. Les binaires sont par la suite désarchivés par TUNe (par l'intermédiaire de son "*RemoteLauncher*") sur les machines distantes. Cette méthode de déploiement répond parfaitement à l'installation des serveurs J2EE qui sont livrés par leur développeur sous forme d'archives.

Après l'installation, les opérations de configuration et de démarrage des serveurs J2EE sont également exprimables grâce au WDL et RDL de TUNe. En effet, la configuration de ces serveurs consiste essentiellement en la modification des fichiers de type clé-valeur. La définition d'une fonction permettant de réaliser ces modifications est facilement réalisable par WDL. Comme nous l'avons indiqué dans la section 5.2.2, le WDL permet de définir des fonctions d'administration via des méthodes de classes java. Ces dernières peuvent donc implantées la modification des fichiers. Quant au démarrage des serveurs, il s'effectue par exécution de commandes shell. De la même

façon qu'il procède pour la configuration, l'administrateur n'aura qu'à implanter les opérations de démarrage via des méthodes de classes java. La seule définition des fonctions de démarrage n'est pas suffisante. En effet, contrairement aux opérations de configuration qui peuvent s'exécuter en parallèle (sur les serveurs), le démarrage des serveurs J2EE requiert un ordre. En guise d'exemple, le serveur d'application Tomcat doit être démarré avant le serveur web Apache qui lui est connecté. Le RDL de TUNe, basé sur les diagrammes d'état-transition d'UML, permet d'exprimer ces contraintes. Ses états particuliers, que sont le "*fork*" et le "*join*", permettent respectivement de paralléliser des opérations et d'attendre l'exécution d'un flux parallèle d'opérations. Une fois le démarrage effectué, l'un des besoins les plus importants dans l'administration d'une application de ce type est la gestion des variations de charge au niveau des différents tiers.

L'administrateur souhaite ajouter des serveurs à un tiers afin d'absorber la surcharge lorsque celui-ci est surchargé. Inversement, il souhaite également réduire le nombre de serveurs d'un tiers lorsqu'il est sous utilisé. Cette dernière opération lui permet de réaliser des économies d'utilisation de machines (très bénéfique lorsque celles-ci sont payantes). L'ensemble de ces deux opérations est appelé *sizing* ou *passage à l'échelle* ou *scalabilité* de serveurs. Pour réaliser cela dans TUNe, l'administrateur associe aux serveurs J2EE des logiciels particuliers jouant le rôle de sondes. Ces dernières déclencheront (en cas de sous/sur charge d'un tiers) dans TUNe des automates de reconfiguration permettant d'ajouter ou retirer des serveurs. Les deux opérateurs d'ajout (suivis d'un déploiement) et de retrait (suivis d'un un-déploiement) d'instances logicielles proposées par TUNe permettent de réaliser ce besoin.

La généralisation de ces besoins permettent à TUNe d'adresser d'autres applications cluster de type maître-esclave comme Ganglia [**?**]. Cependant, lorsque la taille de l'application devient importante, nous observons des limites d'utilisation de TUNe. C'est le cas avec les applications large échelle comme le serveur de calculs DIET [**?**] dans l'environnement de grille grid5000 [**?**].

5.4.2 Applications large échelle : cas de DIET

L'application DIET [**?**] permet de déployer des serveurs (nommés *agents* dans DIET) de calculs dans un environnement de grille. Ces serveurs sont organisés sous forme arborescente et sont de trois types. Les serveurs de premier type se trouvent à la racine de l'arbre : ils sont nommés Master Agent (MA). Ces derniers reçoivent les demandes de calculs venant des clients et les orientent vers les serveurs de calculs (serveur de troisième type) les mieux adaptés. Ces derniers sont déterminés par les serveurs MA en fonction des informations de monitoring qu'ils obtiennent des serveurs de second type : nommés Local Agent (LA). Enfin, les serveurs de troisième type sont situés aux feuilles de l'arbre. Ils sont les véritables serveurs de calculs : nommés Server Daemon (SeD). Ils reçoivent des requêtes des clients et restituent les résultats une fois le calcul effectué. Notons qu'un MA pourrait directement être relié aux SeD si ceux-ci ne sont pas de grand nombre. L'introduction des LA permet d'éviter la saturation des MA lorsque l'application DIET utilise un grand nombre de

serveurs de calculs. Plusieurs niveaux de LA sont introduits entre le MA et les SeD en fonction de la taille de l'application. La figure 5.6 résume l'architecture de cette application. Elle peut être constituée de milliers de serveurs lorsque l'environnement matériel le permet (cas de grid5000). L'utilisation de TUNe pour l'administration (déploiement, configuration et démarrage) de cette application dans un contexte large échelle a montré des limites.

Problème 1.1
La première limite concerne la description par ADL de l'application DIET. Cette description devient fastidieuse (répétition dans la description) lorsque l'application contient des centaines de serveurs (cas sur grid5000). Dans sa version initiale, le système TUNe ne permettait pas la description par intension, c-à-d la description d'un déploiement multiple d'instances de même type via l'attribut "*initial*". En effet, l'attribut "*initial*" (section 5.2.1) qui indique à TUNe le nombre d'instances à déployer initialement pour un type de logiciel a été introduit pour prendre en compte le déploiement des serveurs DIET sur grid5000. Comme nous l'avons décrit dans la section 5.2.1, cet attribut simplifie la description de l'architecture d'une application contenant plusieurs instances du même logiciel. Notons que cette modification du langage ADL de TUNe n'a nécessité aucune modification du cœur de TUNe.

Problème 1.2
La deuxième limite concerne le déploiement des serveurs de calculs SeD. La performance de ces derniers étant liée aux caractéristiques matérielles des machines sur lesquelles ils s'exécutent, l'administrateur dispose de différentes versions d'installation pour chaque cluster de la grille. Or le processus de déploiement fourni par TUNe ne prévoit qu'une seule archive (qui correspond à une version) pour chaque type de logiciel. De plus, le choix de la version n'étant possible que pendant le déploiement (c-à-d une fois le cluster connu), l'administrateur souhaite associer à chaque cluster une version précise de SeD.

Problème 1.3
La dernière difficulté (que nous décrivons dans ce document) concernant l'administration de cette application est liée au caractère centralisé de TUNe. En effet, l'administration d'une application constituée de milliers de logiciels par une unique instance de TUNe devient impossible à cause des limites de la machine qui l'héberge. Par exemple, l'ouverture d'une connexion réseau entre la machine d'administration et tous les serveurs administrés est limitée par les capacités mémoire de la machine d'administration.

5.4.3 Applications virtualisées

Notre dernière tentative d'administration avec TUNe concerne l'utilisation des machines virtuelles (section 2.2) pour une gestion optimale des ressources dans un cluster J2EE [**?**]. En bref, il s'agit d'utiliser une instance de TUNe pour l'admi-

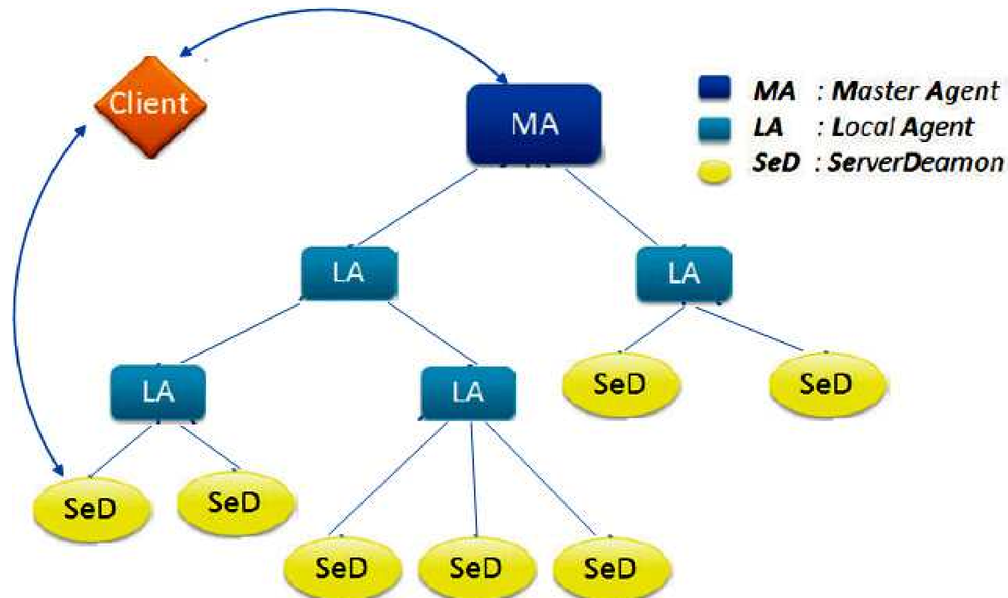


FIGURE 5.6 – Architecture de DIET

nistration à la fois des serveurs J2EE et des VM sur lesquelles ils s'exécuteront. Ainsi, grâce aux atouts des VM, nous implantons plusieurs politiques de gestions de ressources. Malgré l'obtention de résultats prometteurs concernant la gestion de ressources, le système TUNe s'est montré inadapté pour l'administration des VM.

Problème 2.1
Premièrement, l'expression de la double identité des VM (à la fois machine et logiciel, section 2.2) est irréalisable dans TUNe. En effet, les langages et le cœur de TUNe différencient les machines et les logiciels. Il impose d'une part une description sous forme de cluster pour les machines et les représente en son sein à l'aide d'un seul composant. Ceci empêche la manipulation individuelle des VM qui est un besoin important dans cette expérimentation. D'autre part, les logiciels sont décrits et représentés de façon individuelle dans le SR. Qu'elle représentation utilisera donc l'administrateur pour décrire les VM qui jouent les deux rôles ?

Problème 2.2
Après les problèmes de description par ADL et de représentation, l'installation d'une VM ne se limite pas (comme dans TUNe) à la copie de binaires sur la machine distante. En effet, installer une VM revient à installer un système d'exploitation. Ce qui implique une succession d'opérations de natures différentes parmi lesquelles : l'initialisation du système de fichiers, l'installation des pacquages, etc. Prendre en compte cette particularité d'installation revient à personnaliser la méthode de déploiement proposée par TUNe.

Problème 2.3
Pour finir, la prise en compte de l'opération de migration de VM (voir la section 2.2), essentielle dans notre expérimentation [?] est impossible dans TUNe. Il

s'agit du déplacement de l'exécution d'une VM d'une machine physique vers une autre. Elle nécessite entre autre l'exécution atomique du couple d'opérations [undéploiement de la VM ; déploiement de la VM sur la machine de destination]. Autrement dit, elle nécessite une opération du genre "*move*". Ce que ne permettent ni les langages, ni l'implantation de TUNe.

5.4.4 Cloud

Les expérimentations réalisées avec TUNe dans les domaines précédents nous ont conduit vers un domaine beaucoup plus complet et complexe qu'est le cloud. Comme nous l'avons présenté dans la section 3, le cloud dans son ensemble (IaaS et applications entreprise) représente un cas d'administration dans lequel l'utilisation d'un SAA apparaît évidente. Ainsi, nous présentons dans cette section une tentative d'utilisation du système TUNe pour l'administration du cloud dans son ensemble. Sans reprendre la présentation de l'administration dans le cloud réalisée dans la section 3, nous nous appuyons sur cette dernière pour effectuer notre étude. Plus précisément, cette étude présente les limitations du système TUNe pour l'administration dans le cloud.

Problème 3.1
Administrer le cloud dans son ensemble avec TUNe revient à exécuter plusieurs instances de TUNe par niveau d'utilisation : une instance pour l'administration de l'IaaS et plusieurs instances (à raison d'une instance par entreprise) pour l'administration des applications entreprise. Reposant sur la collaboration entre les deux niveaux, l'administration du cloud implique également la **collaboration/interopérabilité** entre les instances de TUNe de niveau entreprise et l'instance de TUNe de niveau IaaS.

Problème 3.2
Comme les environnements de grille, l'IaaS peut être constitué d'un grand nombre de ressources. Ces dernières peuvent être réparties dans plusieurs localités (cas d'Amazon EC2 avec X localités). Dans ce contexte, l'utilisation de TUNe est problématique et rejoint la situation décrite dans le **Problème 1.3** de la section 5.4.2.

Problème 3.3
L'instance TUNe de niveau IaaS administre des éléments de diverses natures : des serveurs assurant les services de base de l'IaaS (facturation, stockage de données, réseaux, etc.), des VM qui servent de support d'exécution pour les applications entreprise, et des équipements matériels. Dans le processus d'administration, le déploiement des serveurs et celui des VM n'est pas réalisable de la même manière. On retrouve le même problème que nous avons évoqué dans la section 5.4.2 (**Problème 1.2**).

Problème 3.4

Comme toute plateforme informatique, les opérations de maintenance font partie du cycle de vie du cloud. Ces opérations peuvent entraîner la mise en indisponibilité partielle ou totale d'un ensemble de machines. La prise en compte de cette activité par TUNe se traduit par **le retrait des machines** concernées du SR. Cependant, cette opération n'est pas possible dans TUNe. Si le retrait concerne un cluster tout entier, uniquement la modification des langages de TUNe permettra de prendre en compte cette opération. Dans le cas où le retrait concerne des ressources individuelles, la modification des langages et du coeur de TUNe sont nécessaires. Ceci s'explique par le fait que le système TUNe ne représente pas de façon individuelle les ressources matérielles dans le SR. Il est donc impossible de les manipuler individuellement. De façon analogue, l'extension du cloud par **ajout de nouvelles machines** ne peut être pris en compte par TUNe. Ces deux situations s'observent également au niveau entreprise. Pour illustrer cela, prenons l'exécution dans le cloud d'une application de type maître-esclave par une entreprise. La scalabilité telle que nous l'avons présentée dans la section 5.4.1) entraînera des ajout/retrait de VM (vues comme des ressources matérielles par les entreprises).

Problème 3.5

Comme nous l'avons présenté dans la section 3, la plupart des opérations d'administration dans le cloud sont déclenchées par celles du niveau entreprise. C'est ainsi que l'ajout/retrait de VM au niveau de l'IaaS survient après l'extension/réduction des ressources au niveau entreprise (les opérations décrites dans le paragraphe précédent). Cette opération n'est pas possible dans le système TUNe. En effet, il le permet uniquement pour des logiciels dont la description est connue avant son (TUNe) démarrage. De plus, les entreprises ou le fournisseur de l'IaaS peuvent souhaiter intégrer de nouveaux types de logiciels dans l'environnement après le démarrage de TUNe (ce qu'il ne permet pas). En guise d'exemple, prenons une entreprise exécutant dans le cloud une application J2EE constituée des serveurs Apache, Tomcat, MySQL-Proxy et MySQL. Initialement, l'application est constituée d'une seule instance d'Apache et aucun distributeur de charges devant Apache n'existe. Après saturation du serveur Apache, l'entreprise peut souhaiter de joindre un distributeur de charges (HA-Proxy) tout en gardant l'exécution de l'ensemble de l'application et du SAA.

Problème 3.6

Le changement ou la définition d'une stratégie commerciale (définition d'un nouveau type de contrat, d'une nouvelle méthode de tolérance aux pannes des VM, etc.) par le fournisseur du cloud se traduit par l'**ajout ou le retrait de politiques de reconfiguration** dans l'instance TUNe de niveau IaaS. Cette situation est similaire aux deux précédentes. Elle peut également concerner le niveau entreprise. De la même façon que l'administrateur de l'IaaS ajoute/retire des logiciels/matériels, il peut effectuer la même opération pour les politiques de reconfiguration.

Problème 3.7

Dans la section 5.4.3, nous avons présenté une politique de gestion de ressources

basée sur la migration des VM. Reposant également sur la virtualisation, le cloud peut mettre en place ces politiques au niveau de l'instance TUNe de l'IaaS. Comme nous l'avons fait remarqué, cette opération n'est pas réalisable dans TUNe : ni au niveau langage, ni au niveau interne.

5.5 Synthèse et nouvelle orientation

Les expérimentations réalisées avec TUNe font ressortir deux types de limites :

- Les limites dues uniquement à l'insuffisance de ses langages d'administration ;
- Les limites dues à l'inadaptation de son architecture et de sa conception.

L'analyse [?] de leurs causes souligne la variation des besoins d'administration en fonction des domaines d'applications. Nous montrons à travers ces expériences que l'implantation actuelle de TUNe ne le prédispose pas à s'adapter à ces variations. Ces dernières nécessitent un système très flexible et très générique (voir le chapitre suivant). Or la conception et l'implantation de TUNe sont réalisées autour de ses langages d'administration (on parle également de machine langages pour désigner ce type de système). En outre, la conception de ces langages a été dirigée par l'administration des applications cluster de type maître-esclave. Ainsi, cette dépendance entre le cœur de TUNe et ses langages le rend moins flexible et moins adaptable pour adresser d'autres domaines applicatifs.

Face à ce constat, nous proposons la construction d'un SAA ne reposant sur aucun langage ou domaine d'administration. L'objectif de ce SAA sera la fourniture d'un ensemble d'API permettant de le rendre le plus flexible et générique possible. Il sera sans doute complexe et donc difficile à utiliser. Pour surpasser cette difficulté d'utilisation, nous fournissons au dessus de ce SAA une pile de logiciels (couche (2) sur la figure 5.7) facilitant son utilisation. Cette pile contient des logiciels permettant aux administrateurs de définir d'eux même les langages d'administration correspondant à leurs besoins applicatifs. Il s'agit d'outils d'IDM (Ingenierie Dirigée par les Modèles) pour la réalisation de langages spécifiques (Domain Specific Language ou DSL dans le jargon IDM) à l'administration autonome (ce qui résout les problèmes d'inadaptation des langages). Ces langages masqueront la complexité du SAA.

Dans cette orientation, le but de cette thèse est de fournir sur la base de TUNe et de Jade, un SAA flexible et générique pouvant supporter plusieurs domaines d'applications (couche (1) sur la figure 5.7). Quant à la couche supérieure (couche (2) sur la figure 5.7), elle fait l'objet d'autres travaux de thèse. La démonstration de l'efficacité du SAA que nous proposons sera son application à l'administration d'un environnement complexe comme le cloud. Avant la présentation de ce SAA, nous décrivons dans le chapitre suivant les critères et l'approche qu'un tel système doit respecter.

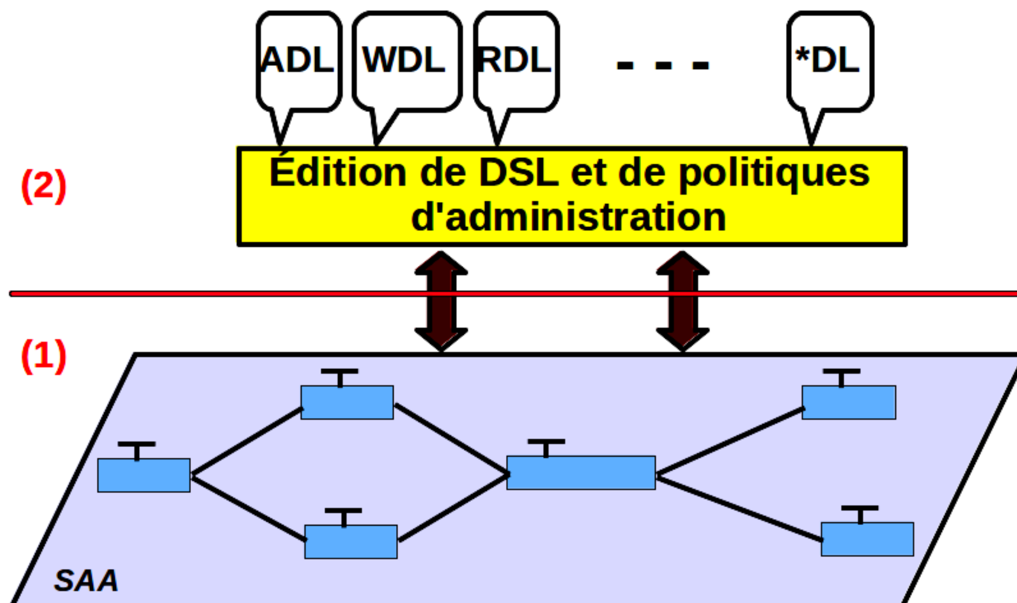


FIGURE 5.7 – Nouvelle orientation du projet TUNe

Chapitre 6

Orientation générale

Contents

6.1 Caractéristiques . . . 55
6.1.1 Uniforme . . . 56
6.1.2 Adaptable . . . 57
6.1.2.1 Adaptation des comportements . . . 57
6.1.2.2 Extensibilité . . . 58
6.1.3 Interopérable et Collaboratif . . . 59
6.1.4 Synthèse . . . 60
6.2 Approche générale et Modèle Architectural . . . 60
6.2.1 Approche Générale . . . 60
6.2.2 Modèle Architectural . . . 61
6.2.3 Prise en compte des caractéristiques . . . 64
6.3 Synthèse . . . 64

Les expériences réalisées avec le système TUNe ont permis de valider d'une part les principes de l'administration autonome et d'autre part l'approche d'administration à base de composants. En particulier, ces expériences ont montré son adéquation pour l'administration des applications cluster de type maître-esclave (application J2EE par exemple). Cependant, contrairement à son aspiration initiale (administration générique), il est difficile de l'utiliser dans plusieurs domaines applicatifs (voir la fin du chapitre précédent pour les raisons). Ainsi, la principale recommandation que nous avons tiré de ces expérimentations est : "***La conception d'un SAA à vocation générique et flexible doit suivre une architecture à deux niveaux (figure 5.7 du chapitre précédent). Le premier niveau (1) est un support pour l'administration à l'exécution (c'est la machine d'exécution ou encore le SAA générique). Le second niveau (2) fournit un support langage pour l'expression des politiques d'administration.***". La conception et le développement du premier niveau fait l'objet de cette thèse. Il s'agit en quelque sorte du développement d'un exo-SAA que nous baptisons "**TUNeEngine**".

L'objectif de ce chapitre est double. Dans un premier temps, nous identifions les

caractéristiques idéales qu'un tel SAA doit remplir afin d'être en mesure de répondre à divers besoins d'administration. Dans la présentation de ces caractéristiques, nous montrons à chaque fois dans quelles mesures elles répondent aux problèmes identifiés dans le système TUNe. Dans un second temps, nous proposons une approche de conception ainsi qu'un modèle architectural de ce SAA.

6.1 Caractéristiques

Fort de nos différentes expérimentations dans le domaine de l'administration autonome, nous proposons les caractéristiques (considérées comme critères ou directives) suivantes pour la construction d'un SAA hautement générique :

- **Uniforme** : Nous définissons un SAA uniforme comme un SAA dans lequel les différences de rôles entre éléments administrés ne sont pas câblées. Autrement dit, il s'agit d'un SAA dans lequel les éléments administrés utilisent la même représentation interne quelque soit leur nature. Ce critère se traduit dans le système TUNe par la non différenciation entre la description et la représentation (dans le SR) des machines et des logiciels (problème 2.1 de la section 5.4.3 du chapitre précédent).
- **Adaptable** : Nous définissons un SAA adaptable comme un SAA capable d'évoluer en fonction des besoins d'administration. L'adaptabilité du SAA revêt différents aspects. (1) L'adaptation de l'implantation du SAA : c'est la modification des portements de base du SAA par modification/remplacement/ajout de services sans connaissance entière de son implantation. (2) La capacité du SAA à faire évoluer dynamiquement les éléments qu'il administre. En effet, un SAA peut se borner à administrer un ensemble d'éléments fixes ou faire évoluer ces éléments par ajout de logiciels, de machines voir de programmes de reconfiguration.
  De nos caractéristiques, l'adaptabilité est la plus importante du SAA que nous envisageons. Cette caractéristique permettra par exemple d'adapter le déploiement du système TUNe pour l'administration des systèmes virtualisés (problème 2.2 de la section 5.4.3 du chapitre précédent).
- **Interopérable ou Collaboratif** : Comme nous l'avons montrée dans le cadre des applications large échelle (problème 1.2 de la section 5.4.2 du chapitre précédent) par exemple, l'administration d'une infrastructure peut nécessiter le concours de plusieurs SAA. Ce dernier est dit interopérable/collaboratif lorsqu'il est capable d'échanger des informations ou des ordres d'administration avec d'autres SAA distincts.

Après ces brèves définitions des caractéristiques du SAA que nous envisageons, le reste de ce chapitre est consacré dans sa deuxième partie à leur présentation détaillée. Pour illustrer ces présentations, nous nous appuyons sur les expériences réalisées avec le système TUNe (chapitre précédent) et plus particulièrement sur le cas de l'administration dans le domaine du cloud.

6.1.1 Uniforme

L'exécution d'un logiciel est une relation entre deux entités : le *support d'exécution (SE)* et le *logiciel*. Le *support d'exécution (SE)* héberge et exécute le *logiciel*. Il implémente tous les mécanismes nécessaires pour l'exécution du logiciel. Il est comparable à une machine munie de son système d'exploitation. D'ailleurs, dans un environnement d'administration, les éléments matériels jouent toujours le rôle de SE. Par contre, les logiciels peuvent également se comporter comme des SE. Considérons l'exemple des machines virtuelles que nous avons présentées dans la section 2.2. Dans la relation [machines physique ; VM], la VM est considérée comme logiciel à l'égard de la machine physique qui l'héberge. Par contre, dans la relation [VM ; logiciel], la VM joue le rôle de SE vis à vis du logiciel qu'elle héberge.

Un autre exemple qui illustre ce double rôle que peut jouer un logiciel concerne les serveurs d'applications (dans les applications J2EE) comme JBoss [**?**] ou Tomcat [**?**]. Pour les mêmes raisons que les VM, ces serveurs sont par définition des logiciels. Cependant, leurs fonctionnalités dans une application J2EE est l'exécution des servlets : ils sont notamment appelés conteneurs de servlets. Ils implantent tous les mécanismes permettant l'exécution et l'accès aux servlets. Ces mécanismes sont comparables à ceux que l'on retrouve dans les systèmes d'exploitation à savoir : la gestion des accès (sécurité), la gestion de la mémoire, le scheduling, la gestion des entrées/sorties, etc. Pour ces raisons, ils peuvent être considérés comme SE vis à vis des servlets.

Pour finir, nous retrouvons le problème d'uniformité dans certains SAA comme Accord [**?**] (confère chapitre 7 sur l'état de l'art). En effet, Accord fige (par implantation) la différence entre les logiciels jouant le rôle de sondes et les logiciels fournissant les services fonctionnels de l'application administrée. En conséquence, il est difficile d'intégrer dans Accord des sondes comme boites noires et de les définir comme éléments administrables.

En sommes, il résulte des deux premiers exemples que dans l'administration d'une application, les éléments administrés peuvent être soit des SE, soit des logiciels, soit les deux à la fois. Il s'agit de la situation que nous avons identifiée par le problèmes 2.1 de la section 5.4.3 du chapitre précédent. Quant au troisième exemple, il pose le problème de la différenciation des rôles des éléments administrés dans le SAA. De façon générale, la recommandation que nous proposons est la suivante : **le SAA ne doit pas figer dans sa conception les différences de rôles des éléments qu'il administre**. Cette recommandation se traduit dans le SAA par l'utilisation d'une représentation uniforme pour tous les éléments administrés qu'il gère. Ainsi, les logiciels, sondes, SE, etc. doivent utiliser la même représentation dans le SAA.

6.1.2 Adaptable

6.1.2.1 Adaptation des comportements

Comme tout logiciel informatique, le projet de développement d'un SAA met en relation deux acteurs : (1) les utilisateurs (qui sont les administrateurs dans notre contexte) et (2) l'équipe de développement. Dans la plupart des cas, ces deux acteurs sont séparés et ne possèdent pas les mêmes compétences. Le premier possède des compétences sur l'application à administrer tandis que le second maîtrise les techniques de développement des SAA. Sur cette base, nous définissons l'adaptabilité des comportements d'un SAA comme sa capacité à être modifiable par les utilisateurs de type (1) sans l'intervention des utilisateurs de type (2), dans le but d'adresser plusieurs domaines applicatifs. Autrement dit, l'adaptabilité des comportement du SAA permettra au SAA d'être générique. L'adaptabilité est une solution parmi plusieurs qui permettent de construire un SAA générique. La question qui peut être posée est celle de savoir "pourquoi notre choix s'est-il porté sur l'adaptabilité du SAA comme réponse à la généricité" ?

Une solution de généricité consiste à fournir des API de bas niveaux utilisables par les administrateurs pour l'implantation de la prise en compte de nouveaux besoins. Cette solution est fournie par le système Jade [**?**] qui propose les API Fractal. Cette solution s'adresse aux administrateurs avertis, ce qui limite son utilisation élargie.

La réponse aux limites de la solution précédente est la fourniture des outils de niveaux d'abstraction proches des administrateurs. Cette solution limite la généricité du SAA et entraîne la construction de SAA par domaine applicatif spécifique. C'est le cas du système TUNe qui se limite aux applications clusters de type maître-esclave.

La dernière solution allie la fourniture d'un niveau d'abstraction élevé à la généricité du SAA. Elle repose d'une part sur l'adaptabilité du SAA (la modification, l'ajout et le remplacement des services du SAA sans avoir une expertise complète sur son développement) et la fourniture des outils d'expressions de besoins d'administration (voir la section 5.5 du chapitre 5). Comme nous le verrons dans la section 6.2, cette dernière solution implique une architecture à composants du SAA. Elle est celle pour laquelle nous optons et dont nous justifions son utilité dans le SAA en montrant comment elle permettrait au système TUNe de résoudre une partie de ses problèmes.

Durant nos expérimentations avec le système TUNe (sections 5.4), nous nous sommes heurtés à plusieurs problèmes nécessitant son adaptation. Il s'agit des problèmes : 1.2, 2.2, 2.3, 3.3, 3.4, 3.5, 3.6 et 3.7. A présent, montrons dans quelles mesures l'adaptabilité de TUNe lui aurait permis de résoudre ces problèmes. Ces derniers requièrent deux types d'adaptation : modification/remplacement de comportements (problèmes 1.2, 2.2, 3.3) et ajout de nouveaux comportements (problèmes 2.3, 3.4, 3.5, 3.6 et 3.7).

Que ce soit le déploiement des serveurs SeD dans le cadre de l'application DIET

(problème 1.2) ou le déploiement des VM dans les systèmes virtualisés et cloud (problèmes 2.2 et 3.3), le remplacement de la fonction de déploiement de base de TUNe permet de prendre en compte les particularités de ces applications. Par exemple, le SAA peut être fourni avec une fonction basique de déploiement et permettre à chaque administrateur de fournir la méthode (par surcharge ou redéfinition de méthodes) qui lui est propre en cas de besoin. Ainsi, concernant l'administration des VM, l'administrateur pourra fournir au SAA une méthode de déploiement incluant les téléchargements de binaires via internet, des copies via des serveurs NFS, etc.

Une autre situation nécessitant une adaptation par modification comportementale s'observe dans TUNe. Elle concerne le moyen d'exécution de fonction d'administration à distance. Par exemple, le système TUNe utilise un protocole figé (RMI) ainsi que des moyens d'introspection et réflexion Java pour l'exécution de méthodes à distance. Ainsi, l'utilisation de TUNe nécessite la présence d'une machine virtuelle Java sur toutes les machines exécutant les éléments administrés. L'utilisation d'un protocole comme *ssh* en remplacement RMI dans un environnement non Java est impossible.

6.1.2.2 Extensibilité

Ne pouvant prédire toutes les opérations réalisables dans toute administration, l'adaptabilité du SAA doit le prédisposer à intégrer de nouveaux modules permettant par exemple de prendre en compte de nouveaux opérateurs (problèmes 2.3 et 3.7) ou de nouveaux besoins comme l'extension de l'environnement (problèmes 3.4, 3.5 et 3.6). La prise en compte de l'opération de migration dans les environnements virtualisés se traduit par l'intégration d'un nouvel opérateur *move*, tel que nous l'avons présenté dans les sections 5.4.3 et 5.4.4. Ce qui permettra de réaliser des politiques de réconfiguration basées sur la migration dans l'administration de l'IaaS.

Quant aux problèmes 3.4, 3.5 et 3.6, ils font ressortir trois types d'extensions de l'administration nécessitant l'adaptation du SAA :

- l'extension de l'environnement matériel (problème 3.4) : Ce problème est notamment rencontré dans les SAA tels que Jade [?], TUNe [?], ADAGE [?], SmartFrog [?], ORYA[?] ou Kadeploy [?]. En effet, ils exigent que la liste des supports d'exécution (machines) soit définie avant leur démarrage. De ce fait, les SE restent figés pendant l'exécution entière du SAA. Un module permettant leur extension dynamique permettrait de faire évoluer cet environnement.
- l'extension de l'environnement logiciel (problème 3.5) : En ce qui concerne l'extension de l'environnement logiciel, nous distinguons deux types d'extensions. La première est celle qu'offre le système TUNe, c-à-d l'ajout/retrait d'instances de logiciels dont la description était connue de TUNe au démarrage. Elle permet par exemple de réaliser la scalabilité des applications J2EE. La deuxième est l'ajout de nouveaux logiciels non existant initialement dans TUNe. La modification du comportement de TUNe aurait permis de prendre en compte ce type d'extension. Ce problème se pose une fois de plus dans les applications J2EE. Prenons par exemple une application J2EE ne disposant pas initialement de distributeur de charges devant le serveur web Apache (Ha-Proxy [?

] par exemple). Après constatation de l'incapacité d'un unique serveur web à répondre à toutes les requêtes, l'administrateur peut décider d'introduire en cours de route d'autres instances afin de partager les charges. Pour cela, il doit ajouter dans l'architecture initiale un nouveau logiciel (Ha-proxy), inexistant initialement, pour la distribution de charges aux différentes instances d'Apache.

– l'extension des politiques de reconfiguration (problème 3.4) : Comme nous l'avons présenté dans la section 5.4.4, il s'agit de la faculté du SAA à intégrer de nouvelles politiques d'administration pendant son exécution.

6.1.3 Interopérable et Collaboratif

L'interopérabilité d'un système informatique est sa faculté à dialoguer avec d'autres. Dans certaines conditions, nous utilisons le terme "collaboration" pour désigner l'interopérabilité. En effet, nous définissons la collaboration comme la mise en communication de plusieurs SAA de même type (problème 3.2) tandis que l'interopérabilité (plus généraliste) met en commun plusieurs SAA de types et de conceptions différents (problème 3.1).

La collaboration dans TUNe, telle que proposée par [**?**] pour l'administration des applications large échelle (section 5.4.2 du chapitre 5) adresse un problème particulier : le passage à l'échelle de TUNe. Elle ne s'aurait s'appliquer au cloud. En effet, [**?**] propose une collaboration entre plusieurs instances de TUNe partageant la même application et décrite par un unique administrateur. Les instances collaborent entre elles pour accomplir l'administration de l'application qui est de très grande taille ici (des milliers d'éléments). De plus, au regard de son implantation (hiérarchisation de TUNe), cette solution est propre aux applications large échelle de type DIET dont l'architecture est hiérarchisée. Les mécanismes de communication entre les instances de TUNe mises en collaboration sont câblés pour répondre à ce type d'application. Qu'à cela ne tienne, cette solution de [**?**] représente une étape préalable de la collaboration des SAA. Dans le cas du cloud par exemple, son fonctionnement n'est effectif que grâce à la collaboration entre les SAA de niveau entreprise et IaaS. Ces SAA sont de natures complètement différentes. Tout le principe de fonctionnement du cloud repose sur la collaboration entre les deux niveaux.

Quelque soit le type d'interopérabilité dans l'administration autonome, les informations échangées sont généralement de différents types. Nous retrouvons par exemple des informations de monitoring (entre le SAA de niveau IaaS et le SAA de niveau entreprise dans le cloud), des méta-informations décrivant un système de représentation répliqué entre plusieurs instances de SAA ou encore des ordres de reconfiguration (cas du cloud). Illustrons ce dernier exemple en reprenant l'administration à deux niveaux du cloud que nous décrivions dans le chapitre 3. Dans cet exemple, plusieurs formes de collaboration sont identifiables :

- La collaboration du niveau entreprise vers l'IaaS est évidente. C'est elle qui permet à l'entreprise d'accéder au cloud par réservation de ressources par exemple. Elle permet également à l'entreprise de se renseigner sur l'état des ressources (VM) qui lui sont allouées.
- La collaboration IaaS vers l'entreprise, elle survient pour la résolution de pannes par exemple. En effet, soit une entreprise ayant une réservation d'une unique VM. A cause de l'incapacité de l'entreprise à détecter une panne machine de la VM, elle est obligée de compter sur l'appel du SAA de niveau IaaS pour cette détection. Ainsi, après détection de pannes et réalisation d'opérations de réparation préliminaires (redémarrage des VM par exemple), ce SAA informe celui du niveau entreprise propriétaire de la VM.
- La collaboration entre plusieurs IaaS : elle survient lorsqu'une plateforme de cloud disposant de moins de ressources et contacte d'autres plateformes pour l'extension de ses capacités. L'ensemble constitué par ces IaaS est appelé cloud hybride. C'est le cas de la plateforme OpenNebula [**?**] qui est capable d'associer une plateforme ElasticHost [**?**] pour étendre ses ressources.

6.1.4 Synthèse

Les caractéristiques que nous avons présentées ci-dessus sont issues de nos expériences réalisées dans le domaine de l'administration autonome avec les systèmes TUNe et Jade. Ces caractéristiques sont orthogonales dans la mesure où elles se complètent et ne possèdent aucun lien entre elles. Elles doivent être envisagées lors de la conception d'un SAA à vocation générique. Dans la section suivante, nous présentons une approche de conception ainsi qu'un modèle architectural possédant les caractéristiques ci-dessus.

6.2 Approche générale et Modèle Architectural

6.2.1 Approche Générale

Les sections précédentes ont présenté les caractéristiques de base que doit fournir notre SAA. L'adaptabilité apparaît comme le critère le plus important dans la mesure où c'est lui qui permet au SAA de prendre en compte la majorité des besoins d'administration. Elle nécessite une implantation sous forme modulaire de telle sorte que chaque fonction du SAA soit identifiée par un module précis. De plus, l'implantation modulaire peut faciliter la dynamicité du SAA lorsque l'adaptation s'effectue pendant l'exécution du SAA. Dans ce contexte, nous préconisons une approche de développement du SAA suivant un modèle à composants. Contrairement à TUNe et Jade qui utilisent cette approche uniquement pour l'encapsulation des logiciels et matériels administrés, l'approche que nous proposons ici consiste à mettre la notion

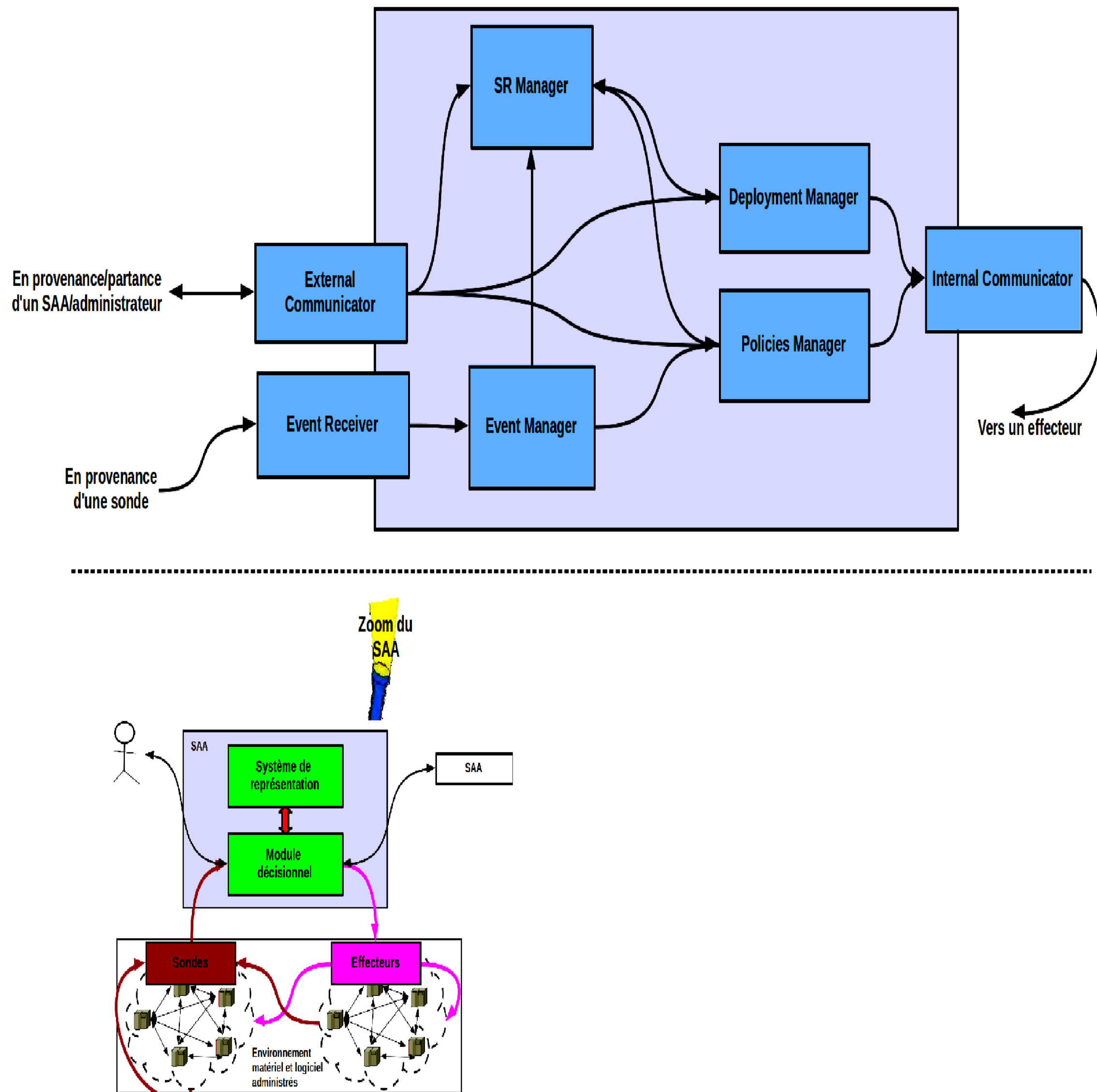


FIGURE 6.1 – Modèle architectural général

de composant au cœur de la conception et de l'implantation du SAA. Autrement dit, les fonctionnalités du SAA et les éléments administrés doivent respectivement être implantées et encapsulés dans des composants. Identifions à présent le modèle à composants que doit suivre notre SAA.

6.2.2 Modèle Architectural

Dans le chapitre 4 nous avons présenté les objectifs et le modèle (figure 4.1) de base d'un SAA. Compte tenu de sa simplicité, ce modèle est très souvent repris dans la littérature sous le terme de *MAPE-K LOOP*. Il présente uniquement l'organisation générale d'un SAA sans présenter en détails sa constitution interne. Une des

raisons qui explique cela vient du fait qu'en fonction des objectifs d'implantation, un SAA peut fournir toute ou partie des fonctionnalités de l'administration autonome (déploiement, configuration et reconfiguration). C'est le cas par exemple du système Taktuk [**?**] qui fournit uniquement des fonctionnalités de déploiement de logiciels (notamment des applications parallèles dans les clusters de grande taille). En ce qui concerne le système que nous envisageons, nous fournissons un SAA remplissant toutes les fonctions de l'administration autonome et implanté suivant une architecture ou un modèle à composants (nous utiliserons le terme "modèle"). Ce modèle fait ressortir tous les composants constituants le SAA. La figure 6.1 présente ce modèle.

Sommairement, il s'agit d'un modèle organisé autour d'une structure de données (le *SR*) contenant tous les éléments administrés (logiciels, matériels) ainsi que les politiques d'administration. Le SAA reçoit des ordres d'administration venant de l'extérieur (via l'***External Communicator***). Après construction du SR (par le ***SR Manager***), les composants gravitant autour de lui permettront au SAA de réaliser l'administration proprement dite. Il s'agit : du ***Deployment Manager*** pour le déploiement/undeploiement [1], de l'***Event Manager*** qui décide des politiques à exécuter à la réception d'événements par l'***Event Receiver***, et du ***Policies Manager*** pour l'exécution des politiques de d'administration ((re)configuration et démarrage). Présentons à présent en détails le rôle de chacun de ces composants.

External Communicator

Il permet au SAA de communiquer avec l'extérieur. Il s'agit aussi bien de la communication avec des acteurs humains (les administrateurs) que de la collaboration/interopérabilité avec d'autres SAA. Ce composant représente le point d'entrée dans le SAA. Il joue un rôle double dans le SAA. Le premier est la réception et l'interprétation des requêtes d'administration en provenance de l'extérieur. Il fera ensuite appel au composant du SAA capable d'exécuter l'ordre d'administration contenue dans la requête. Son second rôle est la restitution à l'initiateur de la requête, les résultats de son exécution. C'est ainsi que l'***External Communicator*** fera appel au :

- ***SR Manager*** lorsque la requête reçue correspondra à la soumission de l'application à administrer au SAA ou la modification du système de représentation (voir ci-dessous) ;
- ***Deployment Manager*** pour le déploiement des applications ;
- ***Event Manager*** pour l'exécution des programmes de reconfiguration.

SR Manager

Il est chargé de construire une représentation des éléments à administrer ainsi que des politiques d'administration de telle sorte que le SAA puisse facilement les manipuler. Compte tenu du fait que ces éléments sont considérés comme des boîtes noires, le ***SR Manager*** doit réaliser pour chacun d'entre eux une opération d'encapsulation. Cette dernière consiste à externaliser (avec le concours de l'administrateur) :

1. Compte tenu de la non existence des antonymes au mot "Déploiement" et au verbe "Déployer", nous introduisons respectivement le mot "Undéploiement" et le verbe "Undéployer" pour jouer ces rôles.

d'une part les fonctions métiers de l'élément considéré afin de les rendre accessibles par le SAA ; et d'autre part les propriétés de l'élément. C'est en fonction de l'implantation de l'encapsulation que le fournisseur du SAA décidera de respecter ou non le critère d'uniformité que nous avons présenté dans la section 6.1.1.

Nous nommons SR (Système de Représentation), l'ensemble constitué de ces éléments encapsulés. Il contient à un instant donné la photographie de l'état courant de l'environnement en cours d'administration. Le SR représente en quelque sorte la base de données du SAA. Il sera utilisé par tous les autres composants du SAA.

Deployment Manager

Il réalise à la fois les opérations de déploiement et undéploiement dans le SAA. Pour cela, il s'appuie d'une part sur le SR afin d'avoir les propriétés des éléments à déployer et d'autre part sur les effecteurs afin d'exécuter concrètement les opérations de déploiement sur la machine distante. La communication avec les effecteurs s'effectue par le biais de l'***Internal Communicator***. Pour finir, notons que le ***Deployment Manager*** définit également les politiques d'allocation et de libération de ressources de l'environnement administré de telle sorte qu'elles pourront être adaptables.

Internal Communicator

Il assure la communication entre le SAA et les éléments en cours d'exécution (par l'intermédiaire des effecteurs). C'est grâce à lui notamment que le SAA pourra réellement exécuter les opérations d'administration.

Event Receiver

Il gère l'arrivée des événements/messages vers le SAA en provenance des éléments qu'il administre. Il les transmettra par la suite au gestionnaire d'événements (***Event Manager***).

Event Manager

En fonction de l'état du SR et des événements qu'il reçoit, il décide de leur prise en compte ou non. Dans le cas où l'événement est pris en compte, il identifie dans le SR les programmes d'administration à exécuter par le ***Policies Manager***.

Policies Manager

Il joue le rôle d'exécution des programmes d'administration. Il doit pour cela implanter les services d'un interpréteur de programmes qui lui permettra d'identifier les séries d'actions comprises dans les programmes qu'il désire exécuter. Pour chaque action identifiée, il fera appel à l'***Internal Communicator*** pour son accomplissement réel à distance sur la machine d'exécution de l'élément concerné par l'action.

6.2.3 Prise en compte des caractéristiques

Après la présentation du modèle architectural devra suivre la conception et le développement d'un SAA générique, montrons à présent comment ce modèle répond aux caractéristiques que nous avons identifiées dans la section 6.1.

L'uniformité des éléments administrés dans le SAA est assurée dans ce modèle par le composant ***SR Manager***. En effet, c'est dans son implantation que les constructeurs du SAA décideront de figer ou non les différences entre les éléments administrés.

L'adaptabilité du SAA est évidente dans ce modèle dans la mesure où il répond entièrement sur une architecture à composants dans lequel tous les services sont identifiables par un composant.

Pour finir, la collaboration/interopérabilité est une fonctionnalité mise en avant et remplie par le composant ***External Communicator*** dans le modèle. Ce composant permet l'émission et la réception d'ordres d'administration entre plusieurs SAA de natures distinctes.

6.3 Synthèse

Dans ce chapitre, nous avons proposé des directives devant guider la conception et le développement d'un SAA hautement adaptable et flexible. Ces directives concernent à la fois les besoins que doit remplir ce SAA, l'approche générale de son implantation et pour finir le modèle architectural qu'il devra suivre. De plus, nous avons montré comment ces directives permettront au SAA de prendre en compte l'administration d'une plateforme de cloud ainsi que des applications qu'il héberge.

Dans le chapitre suivant, nous étudions les travaux de recherche existants qui s'intéressent aux mêmes problématiques que celles que nous adressons dans cette thèse. Il s'agit globalement de la construction des SAA flexibles et de l'administration autonome dans le cloud.

Chapitre 7

Etat de l'art

Contents

7.1 SAA **66**
7.1.1 Rainbow 67
7.1.2 Accord 69
7.1.3 Unity 71
7.1.4 Synthèse 73
7.2 SAA pour le cloud **73**
7.2.1 OpenNebula 74
7.2.2 Eucalyptus 77
7.2.3 OpenStack 78
7.2.4 Microsoft Azure 80
7.2.5 Autres plateformes de cloud 83
7.3 Synthèse **83**

Dans cette thèse, notre principale problématique est l'administration des plateformes de cloud et des applications qu'elles hébergent. Exerçant dans le domaine de l'administration autonome depuis ces dernières années, nous avons établi le rapprochement entre la problématique d'administration dans le cloud et celle des grilles de calculs que nous traitions dans nos recherches. C'est ainsi que nous avons entrepris d'étudier l'utilisation de notre système d'administration autonome TUNe, conçu pour l'administration des applications dans les clusters et les grilles, pour l'administration du cloud dans son ensemble (IaaS et applications des entreprises). Conçu pour être générique et utilisable dans plusieurs domaines, le système TUNe a montré cependant quelques difficultés à prendre en compte des besoins d'administration de certains domaines applicatifs parmi lesquels le Cloud (voir le chapitre 5).

L'étude des difficultés de TUNe nous a conduit à proposer plus généralement un canevas de développement d'un véritable système générique et adaptable à tous les domaines applicatifs dont le cloud. Ce canevas préconise trois recommandations à suivre dans le processus de développement d'un SAA à vocation générique :

(1) Séparer le développement du cœur du SAA (fonctionnalités de base) de celui des langages d'administration qu'utiliseront ses futurs administrateurs.

(2) Le SAA devra remplir les critères suivants :

- l'**uniformité** : il ne dispose d'aucun comportement pré-câblé lié à un type d'élément (les machines par exemple).
- l'**adaptabilité** : la modification de ses services ne doit pas nécessiter la connaissance entière de son processus de développement.
- l'**interopérabilité** : sa faculté à initier la communication avec d'autres SAA d'une part et à exporter ses interfaces d'accès distant d'autre part.

(3) Pour finir, nous recommandons l'utilisation d'une approche de développement à composants pour son implantation.

En somme, nous nous intéressons à deux problématiques : la construction de SAA génériques et l'administration du cloud dans son ensemble (niveau IaaS et entreprise). Dans ce chapitre, nous étudions les travaux de recherche qui s'intéressent à ces problématiques. Nous l'organisons de la manière suivante. Dans un premier temps, nous étudions les SAA à vocation générique existants. Dans cette première étude, nous présentons le positionnement de ces SAA par rapport aux recommandations que nous avons énumérées ci-dessus. Dans un second temps, nous étudions les systèmes d'administration spécifiques aux environnements de cloud. Comme nous l'avions précisé dans la section 3.3, ces systèmes doivent remplir les critères suivants :

- **Interopérable** : capacité à dialoguer avec d'autres systèmes de gestion de cloud et également avec des systèmes d'administration des applications d'entreprises.
- **Auto-réparable** : capacité à prendre en compte automatiquement les pannes dans l'environnement qu'il administre (machines, VM et logiciels).
- **Extensible** : capacité à prendre en compte de nouveaux éléments dans l'environnement (nouvelles machines réelles et virtuelles, nouveaux logiciels).
- **Programmable** : capacité à prendre en compte de nouvelles politiques d'administration (nouvelles méthodes de consolidation, de scalabilité, etc.).
- **Adaptable** : capacité à permettre le remplacement ou la modification de ses modules afin de prendre en compte des particularités de certaines plateformes de cloud.
- **Langages dédiés** : dispose des langages de facilitation de son utilisation. Compte tenu de sa spécialisation dans le cloud, il n'est pas absurde que ce système propose des langages facilitant son utilisation.

Remarquons que ces critères sont inclus dans ceux que nous avons identifiés pour les SAA génériques.

7.1 SAA

Parmi les travaux de développement de SAA, peu d'entre eux ont la vocation de prendre en compte plusieurs domaines d'applications à la fois. En effet, ils s'intéressent chacun pour la plupart à un domaine précis. Ainsi, le critère d'adaptabilité

que nous considérons comme le critère principal de notre modèle est très peu pris en compte dans les autres systèmes. Pour cette raison, notre revue de l'existant étudiera principalement l'extensibilité des SAA qui est l'un des critère découlant de l'adaptabilité et pris en compte par certains SAA. En rappel, l'extensibilité d'un SAA désigne la capacité du SAA à prendre en compte pendant son exécution, l'ajout/retrait de nouveaux éléments logiciels ou des instances de logiciels dont la description existait au démarrage du SAA.

Dans cette section, nous étudions les SAA : Rainbow [?], Accord [?] et Unity [?]. Le premier se définit comme SAA adaptable (au sens que nous avons défini dans le chapitre précédent) à des domaines quelconques tandis que les deux derniers sont spécifiques à des domaines particuliers.

7.1.1 Rainbow

Description générale

La plateforme Rainbow [?] est le fruit du projet DASADA DARPA [?] de l'université de Carnegie Mellon. Développée au sein de l'équipe ABLE [?], elle est disponible comme logiciel open source. A l'exception des SAA TUNe et Jade, Rainbow est l'un des rares SAA que nous avons parcourus (Pragma [?], Cascada [?], Accord [?], Unity [?], etc.) qui se réclament être adaptable à tous types d'applications. Pour cela, son architecture est organisée en deux parties : la première partie implante les fonctionnalités de base de l'administration autonome tandis que la seconde partie implante les services qui pourront être adaptable par le suite. C'est cette deuxième partie qui sera personnalisée en cas de besoin afin de prendre en compte un nouveau domaine d'applications. Le système Rainbow est conçu pour l'administration d'applications préalablement installées et en cours d'exécution. Il ne fournit pas les fonctions de déploiement de logiciels. Son exécution débute par la fourniture d'un modèle représentant l'architecture de l'application en cours d'exécution. Ce modèle sera ensuite maintenu par Rainbow tout au long de son exécution (via les composants Gauges et Model Manager). C'est l'équivalent du système de représentation (SR) dans TUNe.

Pour finir, le modèle architectural (figure 7.1) implanté par Rainbow est proche de celui que nous avons présenté dans la section 6.2 du chapitre 6. A l'exception de l'absence du service de déploiement et de collaboration, nous retrouvons plus ou moins les mêmes éléments :

- Le *Model Manager* correspond au *SR Manager* dans notre modèle.
- Le *Strategy Executor* correspond au *Policies Manager* dans notre modèle.
- L'*Adaptation Manager* et l'*Architecture Evaluator* correspondent à l'*Event Manager* dans notre modèle.
- Le *Translation Infrastructure* implante l'*Event Receiver* de notre modèle. De plus, il met en place le mécanisme d'exécution de méthodes à distance sur les machines hébergeant l'élément administré. Le service implantant ce mécanisme n'est pas clairement identifié dans l'architecture de Rainbow.
-

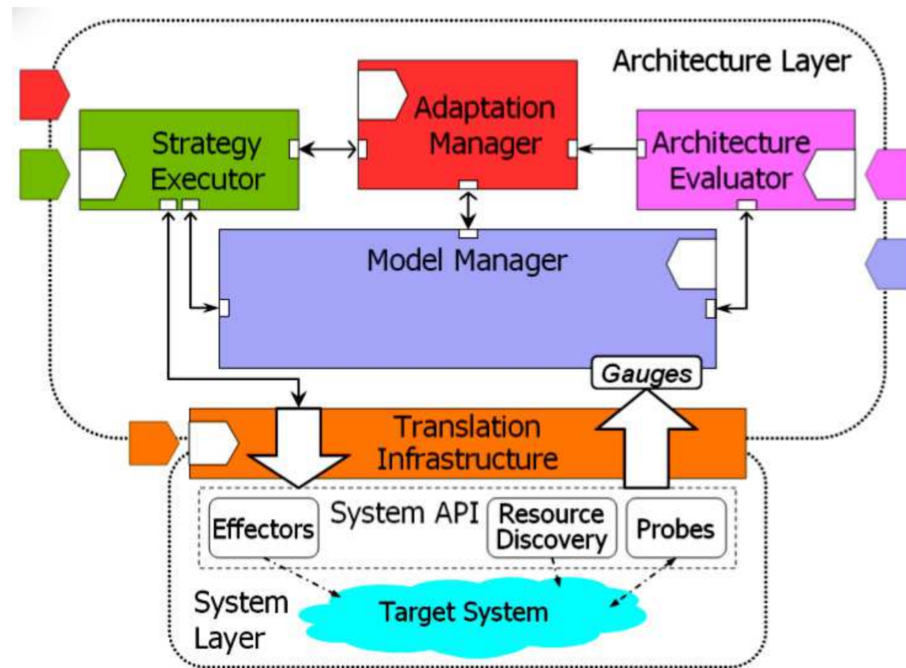


FIGURE 7.1 – La plateforme Rainbow

Ainsi, nous ne présentons pas les éléments présents dans cette architecture extraite de [?].

Machine Langage
Rainbow implante deux langages : l'un pour la description du SR (le langage Acme) et l'autre pour la définition des politiques de reconfiguration (le langage Stitch). L'interprétation de ces langages est câblée et ne peut être remplacée même par adaptation (section **Adaptable**). De plus, ces langages sont propres à l'équipe de développement de Rainbow (ABLE).

Approche de développement
Raimbow est implanté suivant une approche à composants : le modèle Acme [?] développé au sein de la même équipe. De façon similaire, la description et l'encapsulation des logiciels dans Rainbow se fait par le biais du même modèle.

Uniforme
Rainbow organise le SR en deux modèles : un modèle contenant les logiciels à administrer et un modèle contenant les éléments de l'environnement matériels (les machines). Il effectue une différence entre la représentation des machines, la représentation des logiciels et celle des sondes. En exemple : les propriétés des machines de l'environnement sont définies par Rainbow : les sondes ne sont pas considérées comme des logiciels administrables.

Adaptable
La plateforme Rainbow a été conçue afin d'être adaptable aussi bien concer-

nant ses composants de service que les applications qu'il administre. Cependant, la construction et l'intégration de nouveaux services dans cette plateforme restent réservées aux ingénieurs ayant une expertise sur la plateforme. Pour palier à cette difficulté, une plateforme graphique de facilitation de cette tâche est désormais fournie. Il s'agit de RAIDE [**?**].

Lorsque nous nous intéressons aux composants de service de Rainbow qui sont personnalisables, nous nous rendons compte qu'il ne s'agit pas effectivement des fonctions de service telles que nous les avons identifiées dans notre modèle dans la section 6.2 du chapitre 6. En effet, les composants dont parlent les constructeurs de Rainbow sont : les effecteurs, les sondes, les éléments du SR (règles de reconfiguration, logiciels, propriétés des logiciels). Par contre, les composants gérant la construction du SR par exemple, ou encore les interpréteurs de langages d'administration ne font pas partie des services personnalisables.

Interopérable
Aucune fonctionnalité d'interopérabilité n'est fournie par Rainbow.

7.1.2 Accord

Description générale
Accord [**?**] [**?**] fait partie des contributions du projet AutoMate [**?**] à l'université du New Jersey. Le but du projet AutoMate est de fournir des solutions d'administration autonome pour des systèmes complexes. Dans ce projet, Accord (via son implantation DIOS++) est une plateforme de programmation d'applications scientifiques auto-administrables. De la même façon que TUNe, Accord considère les éléments à administrer comme des boîtes noires.

L'encapsulation d'un logiciel se fait par la définition de port de communication dans Accord. Il identifie trois types de port :

- port fonctionnel : correspond aux points d'entrées des fonctions métiers de l'élément. Il est utilisé pour relier les logiciels entre eux afin de permettre leurs interactions.
- port de contrôle : permet de définir les points d'entrée/sortie des effecteurs et sondes.
- port de gestion : utilisé pour l'injection et l'exécution de règles d'adaptation du logiciel.

Il associe à chaque élément administré un Manager qui s'occupe du contrôle (monitoring de l'état, du déclenchement des événements et de l'orchestration de l'exécution des actions de reconfiguration) de son exécution. Le Manager s'exécute sur la même machine que l'élément qu'il contrôle. De cette façon, Accord décentralise l'administration. La figure 7.2 résume l'encapsulation d'un élément dans Accord. Pour finir, notons qu'Accord n'assure pas le déploiement des logiciels qu'il administre. Cette tâche est laissée à la charge des administrateurs. Par contre, il contrôle le déploiement des politiques de reconfiguration.

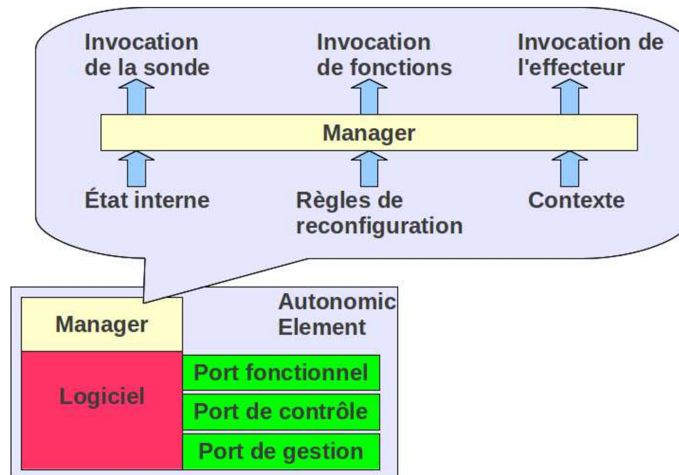


FIGURE 7.2 – Encapsulation d'un logiciel dans Accord

Machine Langage

L'administration dans Accord débute par la définition du contexte applicatif. Il s'agit de la base syntaxique et sémantique qu'utilisera plus tard l'administrateur pour la définition des éléments administrés et des règles de reconfiguration. Pour cela, Accord se base sur les langages SIDL [?] et WSDL [?] pour la définition de ce contexte. SIDL est utilisé pour la définition des composants tandis que WSDL est utilisé pour la définition des ports de contrôle. De plus, il fournit et câble le langage utilisé pour la définition des programmes de reconfiguration. Il s'agit d'un langage simpliste de type ***IF*** <condition> ***THEN*** <action>.

Approche de développement

L'implantation actuelle d'Accord (DIOS++) repose sur deux approches de développement : orientée objet (c++) et modèle à composants (CCAFFEINE CCA [?]). La première a été utilisée pour l'implantation de DIOS (un prototype d'Accord). Quant à la seconde approche, Accord l'utilise uniquement pour l'encapsulation des logiciels à administrer. C'est une approche analogue à celle utilisée par le système TUNe [?].

Uniforme

Parmi les éléments administrés, Accord ne prend pas en compte les éléments matériels constituant l'environnement d'exécution. Il fait l'hypothèse qu'ils sont installés et gérés par des systèmes externes [?] (Rudder, Meteo et Sesame/DAIS). Concernant les éléments logiciels, il effectue une différence entre les logiciels réalisant les fonctions métiers de l'application et les sondes jouant le rôle de monitoring. En effet, les sondes doivent être programmées dans la plateforme Accord. Elles ne sont pas considérées comme des logiciels administrables. Ainsi, pour une application disposant d'un véritable logiciel de monitoring, ce dernier devra être réimplanté dans Accord.

Adaptable

Le développement DIOS++ (implantation d'Accord) ne suit pas une approche à composants et n'a pas envisagé dans sa conception la possibilité de modifier/remplacer/ajouter de nouvelles fonctionnalités par un utilisateur extérieur. Qu'à cela ne tienne, intéressons nous aux facettes adaptables et non adaptables d'Acccord.

Accord permet la modification d'une part de l'architecture et d'autre part des fonctionnalités d'un logiciel en cours d'administration. En effet, étant donné qu'il utilise un modèle à composants pour l'encapsulation des logiciels, la modification de l'architecture ou encore l'enrichissement des fonctionnalités d'un logiciel deviennent possibles. La modification de l'architecture comprend les opérations suivantes : l'ajout, le remplacement et le retrait des logiciels de l'architecture. Concernant l'ajout des logiciels, il ne se limite pas uniquement aux éléments existants initialement dans la définition fournie à Accord par l'administrateur. Quant à la modification du comportement d'un logiciel, elle se réalise par la modification de liaison, la suppression ou l'ajout de ports fonctionnels.

Cependant, Accord garde le contrôle sur l'implantation du processus de remplacement des logiciels. Par exemple, c'est lui qui implante les politiques de réparation de logiciel ou de remplacement de logiciels par d'autres plus optimisés.

Interopérable

La seule interopérabilité qu'implante Accord est celle qu'il effectue avec les systèmes de gestion de l'environnement matériel et réseau (Rudder, Meteo et Sesame/-DAIS [?]) dans lesquels s'exécuteront les logiciels qu'il auto-administre. Globalement, aucune API de collaboration avec d'autres SAA n'est fournie dans Accord.

7.1.3 Unity

Description générale

Unity [?] est l'un des premiers SAA qui a vue le jour après la pose des bases de l'administration autonome par IBM en 2001 [?]. En effet, il est le prototype d'un SAA développé par IBM afin de valider les principes qu'il a énoncé auparavant. Il est conçu pour l'administration des services OGSA [?].

De la même façon que la plateforme Accord (section 7.1.2), tous les éléments à administrer dans Unity sont autonomes. Il associe à chaque élément administré, un composant *Manager* qui sera déployé et exécuté sur la même machine de cet élément. Le rôle du *Manager* est la gestion de l'élément qui lui est associé : gestion des liaisons/interactions avec les autres éléments, monitoring, déclenchement des programmes de reconfiguration (appelés règles) et maintien de l'état de l'élément (par renseignement de propriétés). Notons également qu'Unity auto-administre, de la même façon que les éléments de l'administrateur, ses propres constituants (éléments destinés à la réalisation de ses services présentés ci-dessous). De la même façon qu'Accord, l'association de Managers aux éléments permet de décentraliser

l'administration dans Unity.

Unity est construit autour de six éléments :

- Le gestionnaire de l'environnement du logiciel : il gère les ressources dont l'application a besoin pour atteindre ses objectifs. L'une de ses principales responsabilités est de prédire le comportement du logiciel en cas de de montée et baisse de charges sur les ressources qui lui sont allouées.
- Le gestionnaire de ressources : c'est lui qui implante la politique d'assignation de ressources aux logiciels. Chaque gestionnaire d'environnement de logiciel lui fournit une évaluation des ressources dont a besoin son logiciel. Ensuite, le gestionnaire de ressources décidera des quantités de ressources à allouer à chaque logiciel. Il décide également de l'instant d'assignation ou retrait des ressources.
- l'annuaire : c'est le lieu d'enregistrement des références des éléments administrés par Unity ainsi que des ressources de l'environnement. A travers cet annuaire, chaque élément d'Unity pourra découvrir les éléments dont il a besoin pour son fonctionnement.
- Le gestionnaire des règles de reconfiguration : il permet à l'administrateur de définir et de stocker les règles de reconfiguration.
- Les sentinelles : permettent à un élément d'introspecter l'état d'un autre.
- Les éléments administrés : il s'agit des logiciels (*server*) et des ressources (*OSContainer*) présents dans l'environnement.

L'architecture du système Unity n'est pas éloignée de celle du système Accord.

Machine Langage

Les logiciels à administrer par Unity doivent être développés en Java dans la plateforme "*Autonomic Manager Toolset*" fournie également par IBM. Unity fournit également un langage dit de haut niveau (sur lequel nous disposons peu d'informations dans la littérature consacrée à cette plateforme) pour la définition des règles de reconfiguration.

Approche de développement

Unity utilise un modèle à composants à la fois pour son développement et pour le développement des logiciels administrés. Il pratique le "tout composant".

Uniforme

La plateforme Unity définit un type, câblé en son sein, pour chaque type d'élément qu'il administre. A chaque type est associé un comportement précis. Ainsi : les logiciels administrés et les constituants d'Unity sont dit de type *server* ; les machines hébergeant les *servers* sont de type *OSContainer*.

Adaptable

Unity n'a pas envisagé l'adaptabilité de ses composants. Aucune architecture à composants d'Unity n'est disponible dans la littérature et nous ne saurions évaluer le niveau de difficulté de l'adaptation d'un service d'Unity pour un administrateur non averti.

L'adaptabilité d'Unity par contre peut être observée sous l'angle de l'extensibilité

de l'environnement administré. Unity permet la prise en compte dynamique de : nouvelles règles de reconfiguration, nouvelles machines dans l'environnement, ou encore de nouveaux logiciels à administrer dans l'application. Cette fonctionnalité est permise grâce à l'auto-administration de chaque élément de l'environnement. Ainsi, il sera capable de s'intégrer dans l'environnement sans nécessiter l'arrêt du système.

Interopérable
Aucune fonctionnalité d'interopérabilité n'est fournie par Unity.

7.1.4 Synthèse

Nous avons étudié trois SAA dans cette section : Rainbow [?], Accord [?] et Unity [?]. Parmi ces systèmes, seule l'architecture de Rainbow se rapproche de celle que nous avons proposé dans la section 6.2.2 du chapitre précédent. Quant à Accord et Unity, ils se situent dans la même lignée que les SAA tels que Pragma [?] ou Cascada [?] que nous n'avons pas présentés ici. Malgré son caractère adaptable, Rainbow (comme les autres SAA que nous avons étudiés) n'est pas utilisable pour l'administration du cloud. Dans la section suivante, nous présentons des SAA conçus particulièrement pour l'administration du cloud.

7.2 SAA pour le cloud

Depuis ces dernières années, plusieurs projets autour du cloud computing ont vu le jour et donné naissance à autant de systèmes d'administration dans le cloud. Dans cette section, nous étudions les plus **importants et disponibles** d'entre-eux. Par disponibilité ici, nous entendons les systèmes pour lesquels la documentation existante est suffisamment développée pour faire l'objet d'une étude telle que nous la souhaitons. En effet, parmi les plateformes de cloud importantes, la plupart d'entre elles sont propriétaires et très peu documentées. Par soucis d'exhaustivité, nous identifions deux plateformes non documentées dont nous fournirons uniquement une description générale. Il s'agit des plateformes Amazone EC2 [?] et RightScale [?].

Quant aux plateformes documentées, nous nous intéressons aux systèmes : OpenNebula [?], Eucalyptus [?], OpenStack [?] et Windows Azure [?]. Les systèmes OpenNebula [?], Eucalyptus [?] et OpenStack [?] sont issus des projets open source et ne fournissent que le support logiciel (et pas matériel) de la mise en place d'une véritable plateforme de cloud. Quant à la plateforme Windows Azure [?], elle est propriétaire et commerciale.

7.2.1 OpenNebula

Description générale

La système OpenNebula [?] voit le jour en 2005 à l'université Complutense de Madrid dans le cadre du projet européen open source RESERVOIR [?]. Son objectif dans le cadre de ce projet est l'administration des IaaS virtualisés. Autrement dit, il fournit des services permettant de déployer et d'exécuter dans un environnement matériel virtualisé des VM. Notons qu'une version commerciale d'OpenNebula (OpenNebulaPro) est disponible depuis 2010.

Dans sa version actuelle, OpenNebula est capable de prendre en compte simultanément dans l'IaaS des hyperviseurs Xen [?], kvm [?] et VMware [?]. Il organise l'IaaS sous forme de clusters et de VLAN (réseaux virtuels). Un cluster contient un ensemble de machines physiques tandis qu'un VLAN est défini pour un ensemble de VM. Lors de la création d'une VM, le client choisit la machine et le VLAN dans lequel il souhaite l'exécuter. Notons que dans l'esprit du cloud, il ne revient pas au client de choisir la machine sur laquelle il souhaite exécuter sa VM.

Toutes les opérations d'administration sont coordonnées à partir d'une unique machine de l'IaaS appelée *Frontend*. Son fonctionnement est régi par cinq processus et une base de données (figure 7.3) :

- *Virtual Machine Manager (VMM) Driver* : il gère les API d'interfaçage avec les différents hyperviseurs présents dans l'IaaS. Pour cela, il déploie sur toutes les machines de l'IaaS les scripts d'interfaçage avec l'hyperviseur sur la machine.
- *Transfert Manager (TM) Driver* : il gère le transfert des images de VM dans différentes situations : avant leur exécution (du serveur de stockage vers la machine d'exécution de la VM) ; pendant leur exécution (migration de VM d'une machine à une autre) ou après leur exécution (suppression ou sauvegarde de l'image).
- *Information Manager (IM) Driver* : il fournit les sondes de monitoring et gère les informations que celles-ci obtiennent de l'exécution des VM. Il déploie sur toutes les machines de l'IaaS une sonde (en fonction de l'hyperviseur utilisé) chargée de collecter les informations liées aux VM en cours d'exécution sur la machine.
- *Scheduler* : Il gère les politiques de consolidation des VM dans l'IaaS.
- *Oned* : Il configure, démarre et orchestre l'exécution des processus précédents. C'est le point d'entrée et de ralliement de l'ensemble de ces processus.
- *DB* : C'est la base de données qui contient toutes les informations concernant l'état de l'IaaS et de ses utilisateurs. Elle correspond au SR de notre modèle de SAA.

Interopérable

OpenNebula fournit plusieurs moyens de communication avec l'extérieur. Ces moyens sont basés sur un ensemble d'API de bas niveau (utilisables par des développeurs d'extension à OpenNebula) et une interface de ligne de commandes. Cette dernière est essentiellement réservée aux acteurs humains du cloud. Les interfaces

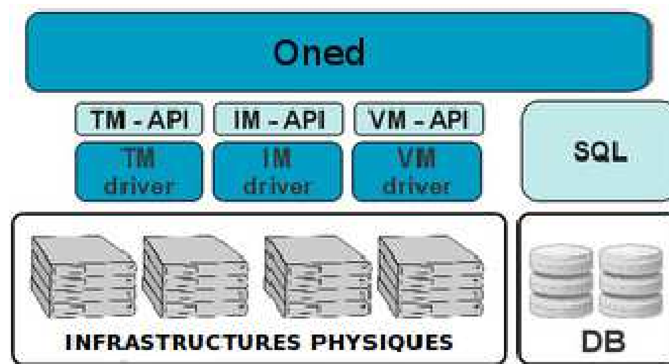


FIGURE 7.3 – Composition d'OpenNebula

de bas niveau et l'interface de ligne de commandes sont utilisées pour l'exécution des commandes d'administration de base (réservation de VM, démarrage, arrêt de VM, etc.). OpenNebula ne fournit aucun support permettant aux entreprises de facilement déployer et administrer leur applications dans l'IaaS. Cependant, ses API peuvent être exploitées par un SAA de niveau entreprise afin d'automatiser les opérations de réservation et de monitoring des VM d'une entreprise. Par contre, en dehors de la communication avec les plateformes Amazon EC2 [**?**] et ElasticHost [**?**] (deux plateformes payantes de cloud), OpenNebula ne dispose d'aucun autre moyen de l'adapter pour l'initiation de la collaboration avec d'autres plateformes de cloud.

Auto-réparable

OpenNebula gère deux types de pannes : pannes de machines et pannes de VM. Dans les deux cas, l'administrateur définit au démarrage de *Oned*, les politiques de réparation qu'il désire appliquer en cas de pannes. OpenNebula utilise le mécanisme d'événement-action (qu'il appelle *Hook*). Un événement correspond à la variation de l'état d'une machine (physique ou VM). L'état de la machine est régulièrement renseigné par les sondes de l'*IM Driver*.

L'exécution des actions associées à un événement est locale (sur la machine d'exécution de *Oned*) ou distante (sur la machine où la panne a été détectée). Cette exécution est donc limitée à une seule machine. Il revient à l'administrateur de définir le lieu d'exécution des actions de réparation. De plus, les Hook d'OpenNebula ne permettent pas de prendre en compte les pannes concernant ses propres serveurs. En effet, ne faisant pas partie des préoccupations d'OpenNebula, aucune sonde n'est prévue pour leur surveillance. La panne d'un serveur DNS par exemple ne pourra ni être détectée ni être réparée par OpenNebula.

Extensible

La gestion de l'IaaS par OpenNebula repose entièrement sur l'extension de ses éléments. En effet, il démarre l'IaaS sans aucun élément et attend leur ajout par l'administrateur. C'est ainsi que les clusters de machines, les utilisateurs et les définitions de VM lui sont rajoutés pendant son exécution. Il fournit parmi ses API le moyen d'ajouter/retirer des éléments de l'environnement. Cependant, il ne permet pas la modification des éléments préalablement ajoutés. Par exemple, la modifica-

tion des caractéristiques (nom DNS ou adresse réseau) d'un VLAN enregistré n'est pas possible. De la même façon, l'ajustement des ressources d'une VM en cours d'exécution dans l'IaaS n'est pas permis.

Programmable

Aucune interface de programmation n'est fournie pour les applications entreprises. La programmation des politiques d'administration dans OpenNebula se limite uniquement au niveau de l'IaaS. La consolidation de VM est réalisée par un processus particulier : le *Scheduler*. Ce dernier s'exécute sur le *Frontend* et est indépendant de l'exécution de *Oned*. Il s'exécute régulièrement à la recherche de VM devant être déplacées vers des machines plus adéquates selon la politique de consolidation. Dans son implantation actuelle, le *Scheduler* d'OpenNebula propose quatre politiques de consolidation : éclatement/regroupement de VM suivant leur nombre sur une machine, éclatement/regroupement de VM suivant leur consommation CPU. Toutes ces politiques sont utilisables simultanément, ce qui peut entraîner des contradictions (éclatement pour certaines VM et regroupement pour d'autres). L'intégration de nouvelles politiques à ce *Scheduler* est impossible. Le seul moyen de l'améliorer est son remplacement tout entier. Cependant, il n'est pas prévu de support logiciel pour la conception de politique de consolidation. C'est la raison pour laquelle seules des solutions issues de son écosystème (développements auxiliaires au projet OpenNebula) existent. Il s'agit des schedulers Haizea [**?**] et Claudia [**?**].

Les politiques de consolidation de VM sont associées par le propriétaire de la VM lors de sa réservation. Il semble étrange qu'OpenNebula utilise cette stratégie. En effet, la prise de décision pour la consolidation devra prendre en compte les exigences particulières de chaque VM et non de celui du cloud dans son ensemble. Rappelons que l'objectif de la consolidation dans le cloud est d'optimiser l'utilisation de ses ressources et non de celles des VM. La consolidation est réalisée dans l'intérêt du fournisseur du cloud et non de celui de la VM à qui des ressources sont allouées et garanties.

Adaptable

La construction d'OpenNebula sous forme modulaire le rend entièrement adaptable. Toutes les tâches d'administration sont identifiables et remplaçables sans aucun besoin de connaissance de l'implantation entière d'OpenNebula (à l'exception du *Scheduler*). En effet, il associe à chacun de ses services un ensemble de scripts définis dans un répertoire précis du *Frontend* de telle sorte que la modification du service s'effectue à un unique endroit. Cependant, il nécessite de la part de l'administrateur une expertise vis à vis des systèmes Linux. De plus, pour les services les plus basiques, comme le moyen de connexion sur les machines distantes (protocole SSH), il est extrêmement difficile de les adapter. En effet, son utilisation par tous les services, implique que son adaptation entraînera celui de tous les services.

En outre, on pourrait se poser la question de la dynamicité de l'adaptabilité d'OpenNebula. La modification d'un de ses modules nécessite le redémarrage de l'ensemble de ses serveurs ainsi que des VM en cours d'exécution dans l'IaaS.

Langages dédiés

OpenNebula ne fournit aucune facilité langage pour l'expression des politiques d'administration. L'administrateur doit avoir une bonne compétence en programmation système Linux et Java.

7.2.2 Eucalyptus

Description générale

Comme OpenNebula, Eucalyptus [?] est également une plateforme open source [1] de cloud computing issu en 2007 du projet VGrADS [?] à l'université de Californie, Santa Barbara. Il permet d'exécuter des VM dans un IaaS virtualisé. Actuellement, Eucalyptus prend en compte des IaaS munis des systèmes de virtualisation Xen, kvm, VMware. Il organise l'IaaS de façon hiérarchique : les machines au niveau des feuilles, les clusters (groupe de machines) au niveau intermédiaire et le cloud (ensemble de clusters) à la racine. Son fonctionnement est organisé de la même façon. Il associe à chaque niveau de l'IaaS un composant précis (figure 7.4) :

- *Instance Manager (IM)* : il s'exécute sur chaque machine de l'IaaS. Il a pour rôle de gérer le cycle de vie des VM.
- *Group Manager (GM)* : il se trouve à la tête d'un cluster. Il collecte les informations des *IM* présents sur les machines de son cluster. Il s'exécute sur une machine du cluster (pouvant également héberger un IM).
- *Cloud Manager (CM)* : C'est le point d'entrée du cloud (pour l'administrateur et les clients). Il est relié aux différents *GM* de l'IaaS.
- *Storage Controller (SC)* : il gère le stockage des images de VM.
- *Warlus* : serveur de stockage de données des utilisateurs externes au cloud. Il permet en quelque sorte au cloud d'être utilisé comme centre de données. Notons que les données qu'il stocke peuvent être utilisées par des VM en cours d'exécution (en fonction des propriétaires).

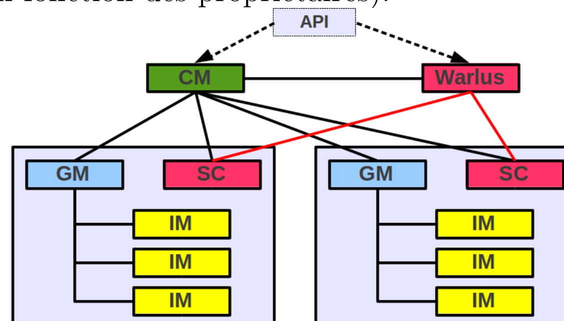


FIGURE 7.4 – Organisation d'Eucalyptus

Interopérable

Les API (de programmation) qu'offrent Eucalyptus permettent aux systèmes externes de le contacter. Par contre, un système Eucalyptus n'est pas prévu pour

1. Une version commerciale d'Eucalyptus existe depuis 2009.

initier la communication avec l'extérieur. La plupart de ces API sont compatibles avec la plateforme de cloud Amazon EC2.

Auto-Réparable

Eucalyptus ne met en place aucun moyen d'auto-réparation de VM ou de ses serveurs. Une panne de VM ne peut être constatée que sur la demande du propriétaire de la VM ou du fournisseur de l'IaaS par demande de l'état de la VM.

Extensible

Comme OpenNebula, Eucalyptus permet l'extension de l'IaaS. Par contre, l'administrateur doit initialement installer sur la machine à ajouter, tous les packages Eucalyptus nécessaires pour l'exécution de l'*IM*. En effet, contrairement à OpenNebula qui se charge de l'initialisation des machines ajoutées, Eucalyptus laisse cette charge à l'administrateur.

Programmable

Aucune politique de consolidation de VM n'est mise en place par Eucalyptus. Il se limite uniquement au placement de VM pendant leur démarrage (choix de la machine d'exécution de la VM). De plus, ce placement est très simpliste. En effet, lorsqu'un client demande l'exécution d'une VM dans un cluster, le *GM* de ce cluster choisit la première machine pouvant l'accueillir. Pour cela, il contacte ses IM séquentiellement à la recherche de celui pouvant héberger la VM. Il est pratiquement impossible pour un non développeur Eucalyptus de modifier cette politique de placement.

Adaptable

Malgré le discours autour de son adaptabilité et sa flexibilité, il est difficile de les mettre effectivement en œuvre dans Eucalyptus. Par exemple, après une installation d'Eucalyptus, la modification de la politique de copie d'images de VM nécessitera la réinstallation des IM sur toutes les machines de l'IaaS.

Langages dédiés

A l'exception des commandes et des API d'utilisation du *CM*, aucun langage dédié n'est fourni par Eucalyptus pour faciliter son administration ou son utilisation. Cependant, le pluging Hybridfox [**?**] peut être utilisé sur le navigateur firefox [**?**] pour effectuer des fonctions de base comme l'allocation des machines VM, l'authentification, etc.

7.2.3 OpenStack

Description générale

OpenStack [**?**] est une pile d'outils open source pour la mise en place et l'administration d'une plateforme de cloud. Les principaux contributeurs de ce projet sont Rackspace [**?**] et la NASA [**?**]. Rackspace fournit les outils de gestion de fichiers dans le cloud tandis que la NASA apporte les outils de gestion de l'IaaS issus du

système Nebula [?] [2]. OpenStack s'organise autour de trois composants et des API qui leur permettent de communiquer (figure 7.5) :
- *Compute Infrastructure (Nova)* : Il fournit les fonctionnalités de gestion du cycle de vie des VM (via le sous composant nova-compute), du réseau (via nova-network) et des authentifications. Il implante également les programmes de scheduling de VM à travers son composant *nova-Scheduler*. Son sous composant *Queue Server* implante le mécanisme de dispatching de requêtes aux autres sous composants en fonction des actions qu'elles requièrent
- *Storage Infrastructure (Swift)* : il gère le stockage de données dans le cloud. Les données sont stockées de façon redondante pour assurer de la tolérance aux pannes. Compte tenu de son isolation dans l'architecture, nous ne le faisons pas figuré dans la figure 7.5.
- *Imaging Service (Glance)* : gère le stockage des images VM.

Notons que ces composants peuvent s'exécuter séparément dans le cloud.

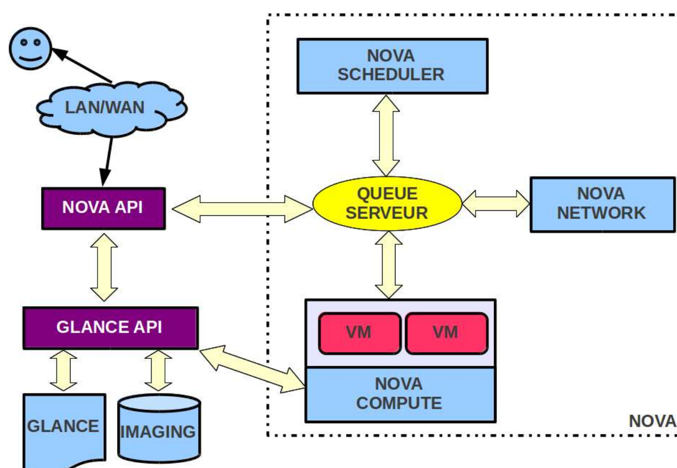


FIGURE 7.5 – Organisation d'OpenStack

Interopérable

Aucun mécanisme d'interopérabilité ou de collaboration avec d'autres plateforme n'est envisagé.

Auto-Réparable

Aucun mécanisme de réparation n'est mis en place dans OpenStack. En effet, il n'implante aucun mécanisme de monitoring et de réparation dans le cloud.

Extensible

Il est extensible dans la mesure où l'ajout d'un nouveau nœud dans l'IaaS revient à y installer un composant *nova-Compute* sur ce nœud. La configuration de ce composant lui permettra de contacter les autres composants nécessaires pour son fonctionnement (Glance, nova-Network et nova-Scheduler) et également de s'insérer

2. Ne ma confondre Nebula fourni par la NASA et OpenNebula.

dans l'environnement l'IaaS. L'ajout/retrait de nœud ne nécessite donc pas d'arrêt du cloud.

Programmable
Le Scheduler fournit uniquement un algorithme de placement (pas de consolidation) de VM. Il implante trois politiques de scheduling. la première (*chance*) choisit le nœud aléatoirement dans la liste des nœuds de l'IaaS. La seconde politique (*availability zone*) est similaire à la première à la différence qu'elle se limite à une zone/cluster de machines précisées à l'avance. La dernière politique (*simple*) choisit le nœud ayant la consommation courant la moins élevée.

Adaptable
La construction modulaire d'OpenStack lui permet d'être adaptable. Par exemple, le changement de la politique de gestion des configurations réseaux dans les VM se fait facilement en remplaçant le composant *nova-Network*.

Langages dédiés
Aucun langage particulier d'utilisation du cloud n'est fourni. Il se limite à la fourniture d'API et d'outils de ligne de commandes. Comme Eucalyptus, il supporte également l'utilisation du pluging Hybridfox [**?**] du navigateur firefox [**?**] pour effectuer des fonctions d'administration de base dans l'IaaS (comme la gestion du cycle de vie des images de VM et des VM, l'authentification, etc).

7.2.4 Microsoft Azure

Description générale
Microsoft Azure [**?**] (que nous appellerons Azure) est une plateforme commerciale de cloud développée par le groupe Microsoft. Il joue un double rôle : accompagnement des clients dans le processus d'externalisation, et gestion de l'IaaS (uniquement le système de virtualisation Hyper-V [**?**]). Il est organisé autour de quatre composants principaux :

- *AppFabric* : il réalise le premier rôle de la plateforme. C'est la plateforme de développement des applications entreprises qui seront externalisées vers le cloud. Les applications issues de cette plateforme seront facilement administrées dans l'IaaS. Il correspond au SAA de niveau entreprise. C'est grâce à l'*AppFabric* que la plateforme Azure est considérée dans la littérature comme un PaaS.
- *Windows Azure* : il réalise le second rôle de la plateforme. C'est lui qui déploie et exécute les VM dans l'IaaS (grâce à son composant *FrabricController* conçu pour le système de virtualisation Hyper-V).
- *SQL Azure* : C'est le système de gestion de base de données d'Azure.
- *Marketplace* : C'est une plateforme de vente et d'achat de composants logiciels développés sur *AppFabric*. En effet, dans le but de faciliter les développements sur *AppFabric*, les clients peuvent se procurer des briques logiciels pré-développées et mises en vente sur *Marketplace*.

Les facilités d'administration fournies par Azure se limitent aux applications (appelées "rôle[3]" dans Azure) web de type n-tiers. En effet, le composant *AppFrabric* ne permet que le développement de ce type d'applications. Cependant, il est capable d'héberger d'autres types d'applications directement fournies dans des VM (appelée rôle VM). Dans ce cas, la charge est laissée au client de construire et soumettre au cloud les VM contenant ses logiciels. Quant aux applications construites via *AppFrabric*, l'IaaS prend en charge la construction des VM devant les héberger. La figure 7.6 résume la vue générale de Microsoft Azure.

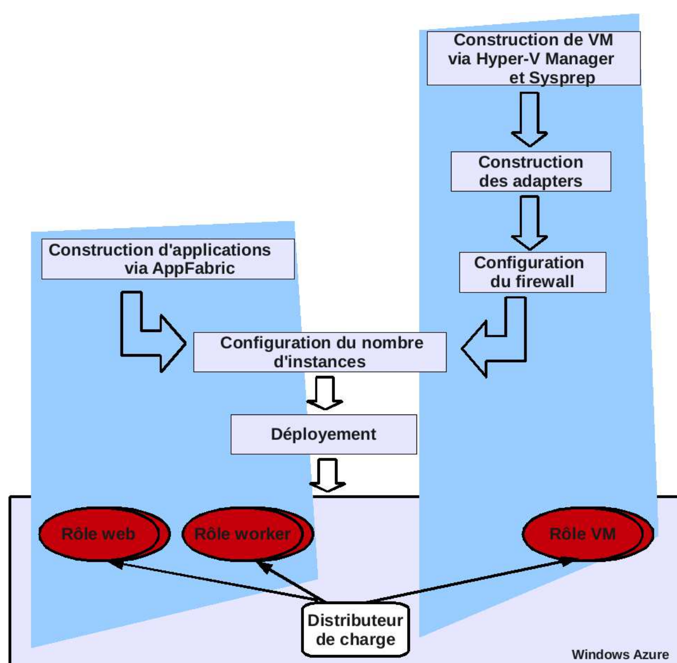


FIGURE 7.6 – Vue générale de Microsoft Azure

Interopérable

Azure ne dispose d'aucune interface de collaboration avec d'autres plateformes de cloud : ni d'Azure vers l'extérieur, ni de l'extérieur vers Azure. Il pourrait s'agir d'une stratégie de Microsoft empêchant les autres plateformes de cloud de conquérir ses clients. Cependant, Azure dispose des mécanismes de collaboration interne entre le gestionnaire de l'IaaS (Windows Azure) et le gestionnaire des applications entreprises (*AppFabric*). En effet, l'*AppFabric* s'adresse au composant *Windows Azure* (gestionnaire de l'IaaS) pour la création de VM, l'obtention des informations de monitoring, etc. Cette collaboration est câblée et maîtrisée par les deux composants *Windows Azure* et *AppFabric*.

Auto-Réparable

L'auto-réparation est uniquement fournie pour les VM construites par l'IaaS. Autrement dit, il s'agit des VM exécutants les applications issues de AppFabric. Dans ce cas, l'IaaS duplique l'exécution des logiciels en démarrant deux instances de

3. terme utilisé par Windows Azure pour désigner les éléments qu'il administrent

VM pour chacun d'eux (la VM supplémentaire prend le relais si la première tombe en panne). Toutes les opérations d'écritures sont effectuées dans un emplacement répliqué partagé ne se trouvant par sur les VM. De cette façon, lorsqu'une instance de VM est en panne, *Windows Azure* le redémarre à partir de son état d'avant la panne.

Quant aux VM construites par le client, c'est à ce dernier d'implanter les mécanismes d'auto-réparation.

Extensible

Nous n'avons aucune possibilité d'étudier cette facette d'Azure car elle est propriétaire et sa gestion interne est inconnue.

Programmable

Aucune politique de consolidation des machines virtuelles sur l'IaaS n'est mise en place dans Azure. Par contre, il propose des politiques de scalabilité pour les applications clientes sous son contrôle (c-à-d issues de *AppFabric*). Elles sont déclenchées via des API par les applications elles-mêmes ou par leur propriétaire. Cependant, il est impossible d'introduire de nouvelles politiques de reconfiguration dans une application issues de *AppFrabric*).

En ce qui concerne la configuration des applications directement construites dans les VM par le client, Azure permet l'implantation de programme (*adapters*) permettant de les réaliser. Ces programmes sont associés à la VM et s'exécutent à son démarrage.

Adaptable

Nous n'avons aucune possibilité d'étudier cette facette d'Azure car elle est propriétaire et son code source est inaccessible.

Langages dédiés

Microsoft Azure fournit à travers sa plateforme de développement des langages et API de développement d'applications pour son cloud. Ces langages sont des langages classiques de développement (java, .Net et C#) et du XML. Le développement, l'assemblage, l'interconnexion des composants logiciels ainsi que leur configuration s'effectuent par programmation dans *AppFabric*.

Pour les applications non issues de *AppFabric*, il fournit un outil (*Hyper-V Manager*) de construction et de téléchargement d'images de VM vers le cloud. Il utilise le format *Virtual Hard Disk*[4] (VHD) pour la description de l'image à construire. Ce format est basé sur XML et permet de décrire le contenu de l'image de VM à construire.

4. VHD est chez Microsoft ce qu'est OVF (Open Virtual Format) chez VMWare.

7.2.5 Autres plateformes de cloud

Amazon EC2

Amazon Elastic Compute Cloud (EC2) [?] est une plateforme commerciale de cloud multi-services. Il peut à la fois être utilisé comme espace de stockage de données, espace de calculs pour les applications scientifiques ou comme centre d'hébergements d'applications orientées web. Pour ces deux derniers services, le client doit se servir de machines virtuelles. Il exécute des VM d'OS de type Linux ou Windows. Son infrastructure matérielle est répartie sur plusieurs zones géographiques (deux zones aux USA, une zone en Europe et une zone en Asie) afin de proposer aux clients des lieux d'exécution proches. Cette pratique permet également à Amazon de minimiser les communications réseaux avec l'IaaS.

En ce qui concerne l'assistance aux clients, Amazon fournit des services web d'observation de charges des VM. Pour des applications web de type J2EE, il fournit des politiques de passage à l'échelle (scalabilité). Par contre, il ne pratique aucune politique de consolidation de VM au niveau de l'IaaS. Pour finir, Amazon n'implante pas de mécanisme de dépannage de VM. Il revient au client de l'implanter.

RightScale

RightScale [?] est présenté dans la littérature et par ses fournisseurs comme étant une plateforme de cloud. En réalité, il permet de présenter de façon uniforme l'accès à plusieurs plateforme de cloud. En effet, il est relié à plusieurs véritables plateformes de cloud (Amazon EC2 par exemple) dont il peut effectuer des réservations de VM.

Le principal apport de RightScale est sa facilitation du processus d'externalisation de l'entreprise vers le cloud. Cette facilitation se limite aux applications web de type J2EE. Il implante toutes les fonctions d'administration au niveau entreprise telles que nous les avons présentées dans la section 3.2 du chapitre 3. Il fournit des moyens de construction de VM, téléchargement d'images VM vers le cloud, déploiement de serveurs J2EE, réparation de VM et de serveurs J2EE, passage à l'échelle des tiers J2EE et monitoring des VM.

7.3 Synthèse

L'étude des travaux autour de la construction de SAA montre qu'il existe très peu d'études qui s'intéressent à l'administration d'applications appartenant à des domaines très variés. La plupart d'entre elles sont spécifiques à un domaine voir une application particulière. L'étude du système Rainbow [?] (section 7.1.1), qui se réclame de la catégorie des SAA génériques et flexibles, s'est révélée non concluante. En effet, la partie du système qu'il définit comme adaptable ne concerne que les effecteurs, les sondes et les éléments du SR. Or, ces éléments ne correspondent pas aux composants de base constituants le SAA, qui déterminent le degré d'adaptabilité d'un SAA.

Face à ce constat, il est difficile d'envisager l'application de ces systèmes dans le cadre de l'administration des plateformes de cloud, notre préoccupation initiale.

Par ailleurs, nous avons passé en revue les plateformes de cloud et étudié leur capacité d'administration autonome. Le principale constat qui découle de cette étude est l'absence de la prise en compte (par ces plateformes de cloud) de la collaboration entre les systèmes d'administration des applications entreprises et le système d'administration de l'IaaS.

Dans le chapitre suivant, nous présentons donc l'implantation d'un SAA suivant les directives que nous avons identifiées dans le chapitre précédant. Comme nous le verrons, il sera par la suite utilisé pour l'administration d'une plateforme de cloud (un prototype que nous implanterons également) ainsi que des applications entreprises qu'exécutera cette plateforme. Pour finir, la collaboration entre les deux niveaux sera favorisée par cette uniformité des SAA de niveaux IaaS et applications entreprises.

Chapitre 8

Contributions : TUNeEngine

Contents

8.1 Modèle détaillé de TUNeEngine 86
8.1.1 Soumission de commandes et Collaboration 87
8.1.2 Construction du SR 87
8.1.3 Déploiement 89
8.1.4 Configuration, Démarrage 90
8.1.5 Réconfiguration 91
8.2 Le formalisme à composants Fractal 92
8.3 Implantation de TUNeEngine 94
8.3.1 Le composant *TUNeEngine* 94
8.3.2 Soumission de commandes et Collaboration 95
8.3.3 Construction du SR 96
8.3.4 Déploiement 98
8.3.5 Exécution de programmes d'administration 100
8.4 Synthèse 102

Les systèmes Jade[**?**] et TUNe (section 5 du chapitre 5) ont été précurseurs en ce qui concerne le développement des SAA à vocation **générique**. Cependant, les difficultés observées dans l'utilisation de ces systèmes nous ont conduit à proposer dans le chapitre 6, une nouvelle approche d'implantation de SAA hautement flexible et adaptable. Cette approche identifie les besoins que doit remplir un tel système. Elle débouche ensuite sur un modèle architecturale basé sur les composants. Dans ce chapitre nous présentons une implantation de ce modèle que nous baptisons : TUNeEngine.

Reprenant la base de code de TUNe, l'implantation de TUNeEngine est basé sur le formalisme à composants[1] Fractal [**?**].

1. Afin d'éviter toute ambiguïté de compréhension avec le modèle que nous présentions dans le chapitre 6, nous n'utilisons pas l'expression "modèle à composants Fractal" mais plutôt "formalisme à composants Fractal".

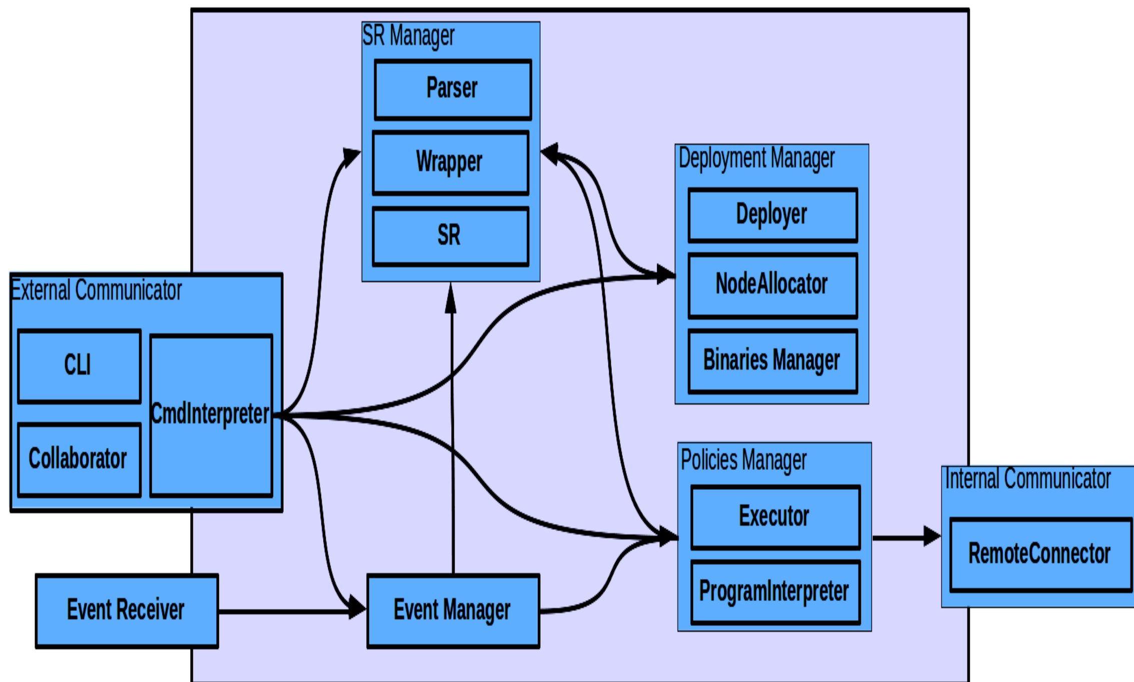


FIGURE 8.1 – Modèle détaillé de TUNeEngine

Ce chapitre est organisé de la façon suivante. Nous commençons par la présentation du modèle architectural détaillé de TUNeEngine. Ce modèle est un enrichissement de celui que nous avons présenté dans le chapitre 6. Ensuite, nous présentons de façon succincte un aperçu du formalisme Fractal sur lequel est basé l'implantation de TUNeEngine. Nous terminons ce chapitre par la présentation de l'implantation Fractal du modèle détaillé de TUNeEngine.

8.1 Modèle détaillé de TUNeEngine

Dans la section 6.2.2 du chapitre 6, nous avons présenté le modèle architectural que devra suivre l'implantation de TUNeEngine. Nous le revisitons en l'enrichissant des composants les plus basiques que doit implanter TUNeEngine. La figure 8.1 présente ce nouveau modèle enrichi. Afin de justifier les composants de ce modèle, explorons le processus d'administration d'une application quelconque par un SAA. Ce processus va de la soumission de l'application à administrer au SAA, jusqu'au suivi de son exécution en passant par son déploiement. Notons que l'exploration de ce processus nous sert uniquement de fil conducteur dans notre présentation.

8.1.1 Soumission de commandes et Collaboration

La première étape dans le processus d'administration autonome est la soumission des politiques d'administration au SAA par l'administrateur ou un système externe. Ces politiques contiennent aussi bien la description des éléments à administrer (machines et logiciels) que les politiques de reconfiguration. De plus, le SAA peut initier la communication avec l'extérieur dans certaines situations. Par exemple lorsqu'il sollicite, par collaboration, les services d'un SAA externe pour l'accomplissement d'une tâche. En conclusion, on peut parler d'un dialogue entre un acteur externe (un administrateur ou un SAA) et le SAA. Le composant ***External Communicator*** de notre modèle permet de réaliser ce dialogue.

En fonction du type de dialogue, l'***External Communicator*** s'appuie sur deux composants internes : l'***Interface de Ligne de Commandes (CLI)*** et le ***Collaborator***. Le premier s'occupe de la communication avec des acteurs humains tandis que le second s'occupe de la collaboration/interopérabilité avec d'autres SAA.

Les commandes émises/reçues sont de différents ordres. Il peut s'agir de la soumission au SAA de l'environnement à administrer, des politiques d'administration à appliquer, des ordres de déploiement/undéploiement, etc. Pour distinguer ces requêtes, le composant ***External Communicator*** est muni d'un interpréteur de commandes (***CmdInterpreter***). Ce dernier à également pour rôle de vérifier la conformité syntaxique et sémantique des commandes (autrement dit si elles seront compréhensibles par le SAA). En cas de conformité, le ***CmdInterpreter*** fait appel au composant du SAA capable de réaliser l'opération contenue dans la commande : la construction du SR par exemple.

Adaptabilité

L'adaptation du composant ***Collaborator*** peut permettre au SAA d'implanter un nouveau protocole de communication avec les SAA externes. Dans le cadre de l'implantation de TUNeEngine par exemple, nous utilisons un mécanisme basé sur RMI pour la collaboration avec des SAA externes (voir la section 8.3.2). Le remplacement du composant ***Collaborator*** dans TUNeEngine permettra d'implanter par exemple une collaboration à base du protocole REST tels que pratiquées dans les plateformes de cloud comme Eucalyptus ou OpenNebula.

L'adaptation du composant ***CmdInterpreter*** permet de modifier la syntaxe de soumission des commandes au SAA. Au lieu d'une soumission de commandes ligne pas ligne, le SAA pourra permettre l'expression de plusieurs commandes sur la même ligne et séparées par des points virgules par exemple.

8.1.2 Construction du SR

Une fois les éléments à administrer soumis au SAA, celui-ci doit construire en son sein une représentation de ces éléments : c'est le rôle du ***SR Manager*** dans notre modèle. Il s'appuie sur trois sous composants : le ***Parser***, le ***Wrapper*** et le

SR.

Le ***Parser*** a pour rôle d'identifier à partir de la soumission de l'administrateur : la liste des éléments à administrer par le SAA, leurs propriétés, ainsi que les relations entre eux. Il peut être organisé en plusieurs ***Parser***s afin de prendre également en compte les éléments comme les programmes d'administration. Pour chaque élément identifié, le ***Parser*** demande au ***Wrapper*** de construire le représentant de cet élément dans le SAA.

Construire le représentant d'un élément dans le SAA signifie encapsuler son comportement dans une structure de données qui facilitera son administration par le SAA. Nous appelons également *wrapper* ce représentant. Pour le construire, le ***Wrapper*** utilise les informations fournies par le ***Parser***. Les éléments construits peuvent être des logiciels, des machines, des liaisons, des éléments constituant un programme de reconfiguration, etc. Nous proposons dans notre modèle, l'utilisation du formalisme à composants Fractal pour la construction des *wrappers*. Comme nous le verrons ci-dessous, ce choix de Fractal ne remet pas en cause l'adaptabilité du ***Wrapper***. En effet, construire un représentant dans le SAA revient à construire à terme une architecture logiciel. Cette fonctionnalité est parfaitement remplie par Fractal. Utiliser un formalisme ou un moyen autre que Fractal revient à utiliser/implanter un formalisme équivalent. En réalité, le choix porté sur Fractal est favorisé par ses API qui répondent parfaitement à nos attentes dans la construction des représentants. Notons que l'implantation du ***Wrapper*** décidera du câblage ou non des types d'éléments dans le SAA (voir le critère d'uniformisation de la section 6.1.1 du chapitre 6). Dans le cadre du système TUNeEngine, nous utilisons (comme nous l'avons préconisée) une unique structure de données pour l'encapsulation de tout type d'éléments.

Pour finir, le ***SR*** dans notre modèle joue deux rôles. Il représente d'une part la structure de données contenant l'ensemble des représentants des éléments administrés et programmes de reconfiguration dans le SAA. Nous reprenons l'expression de "Système de Représentation" introduit par le système TUNe pour le désigner. D'autre part, il fournit le moyen d'introspection des wrappers et du système de représentation. Il sera sollicité par les autres composants du SAA lorsqu'ils voudront avoir des propriétés, des fonctions ou autres informations du représentant. Par exemple, le ***Wrapper*** fera appel à lui pour l'obtention de la référence Fractal des *wrappers* de deux éléments lors de la construction d'une liaison ces deux éléments.

Adaptabilité

L'adaptation du ***Parser*** permet au SAA de prendre en compte de nouveaux langages de description des besoins d'administration (éléments et programmes de reconfiguration). On pourra par exemple prendre en compte l'ADL du système TUNe qui est basé sur les diagrammes de classe UML.

En ce qui concerne le ***Wrapper***, son adaptation permet par exemple à l'administrateur d'exprimer un nouveau moyen d'encapsulation dans le SAA ou d'associer un comportement particulier à un représentant d'un élément administré. Pour illustrer le premier cas, prenons un exemple dans lequel l'administrateur désire encapsuler ses éléments dans une base de données. Compte tenu du fait que nous adoptons Fractal

comme la base de construction de *wrappers* dans le SAA (ce qui n'empêche l'utilisation d'autres moyens), l'adaptation du ***Wrapper*** construira donc deux *wrappers* : un composant Fractal et une entrée dans la base de données. Associé à cette adaptation du ***Wrapper***, le composant ***SR*** devra être également adapté. En effet, il devra associer aux API d'introspection des composants Fractal (ce que permet Fractal) le moyen d'accéder aux informations concernant à la fois le composant Fractal lui même (le *wrapper* de base), mais également l'entrée correspondant à ce composant dans la base de données (nouvelle structure d'encapsulation). En somme, l'utilisation de Fractal dans notre modèle sert d'interfaçage lorsque que la méthode d'encapsulation est adaptée.

Quant au second cas (l'adaptation du ***Wrapper*** pour l'association d'un comportement particulier à un élément), il permet par exemple de reproduire le choix d'implantation du système TUNe dans lequel il associe aux supports d'exécution (machines physiques), un comportement différent de celui des éléments logiciels. La seul différence ici est le non câblage de cette différence dans le SAA.

Pour finir, l'adaptation du ***SR*** permet, comme nous l'avons évoqué dans l'adaptation du ***Wrapper***, d'enrichir les API d'introspection de Fractal.

8.1.3 Déploiement

Après la représentation interne des éléments administrés, la phase de déploiement marque le début du processus d'administration proprement dit. Il est assuré par le composant ***Deployment Manager***. Ce dernier réalise le déploiement en plusieurs phases internes à savoir :

- le choix du support d'exécution (SE) : réalisé par le composant ***NodeAllocator***. Il détermine le lieu d'exécution de l'élément en cours de déploiement. Il se sert du ***SR*** afin d'avoir les informations sur les SE disponibles et retourne ensuite un ou plusieurs SE en fonction des besoins de l'élément déployé.
- initialisation du SE : à l'aide d'un moyen (ssh, rsh, etc.) et des informations (nom de connexion, mot de passe, etc.) de connexion fourni par le représentant du SE dans le SR, le composant ***Deployer*** initialise le SE. Cette initialisation permet au SAA d'avoir accès et main mise sur le SE distant.
- L'obtention et l'installation des binaires : réalisées par le composant ***Binary Manager***, il rend disponible les binaires et fichiers nécessaires pour l'exécution de l'élément déployé sur le SE distant (exemple de l'installation des packages sous Linux dans le répertoire */var/cache/apt/archives/*). Il dispose ensuite les fichiers selon l'arborescence requise par l'installation de l'élément en cours de déploiement (dans l'exemple des packages Linux, il s'agit du désarchivage des fichiers vers les répertoires requis).

A l'inverse, notons que les composants précédemment présentés réalisent également l'opération de désinstallation. Le ***NodeAllocator*** libère le SE après son nettoyage par le ***Binary Manager***.

Adaptabilité

L'adaptation du ***NodeAllocator*** permet d'implanter plusieurs politiques d'allocation ou de libération de SE dans le SAA. Par exemple, on pourra implanter l'allocation par tourniquet ou *round-robin* utilisée par le système TUNe.

Quant au ***Deployer***, prenons l'exemple de l'administration des équipements réseaux comme les imprimantes. L'adaptation du ***Deployer*** permettra de ne pas déployer le représentant du SAA sur l'équipement réseau (car impossible) comme on le fera dans le cadre de l'administration d'une machine physique. Dans ce cas, le ***Deployer*** pourra par exemple se contenter d'un déploiement de ce représentant sur la machine d'administration, qui fera par la suite un contrôle distant selon le protocole de communication fourni par l'équipement.

Pour finir, l'adaptation du ***Binary Manager*** permettra au SAA de prendre en compte différentes méthodes d'obtention de binaires : par copie distant, par téléchargement web comme c'est le cas avec l'installation des système d'exploitation, etc.

8.1.4 Configuration, Démarrage

Les phases de configuration et de démarrage suivent celle du déploiement. Étant donné qu'elles sont de même nature (exécution de programme), le SAA utilise le même composant (***Policies Manager***) pour les réaliser. Il se met en marche lorsqu'il reçoit un ordre venant de l'***External Communicator*** ou de l'***Event Manager***. L'exécution d'un programme d'administration s'effectue en deux étapes :

- l'interprétation du programme : réalisée par le composant ***ProgramInterpreter***. Ce dernier utilise une partie du ***SR*** représentant le programme à exécuter. Il identifie dans le programme la liste des actions à exécuter. Son implantation dépendra du langage d'administration choisi par l'administrateur.
- l'exécution des actions identifiées dans l'étape précédente : les actions identifiées sont exécutées par le composant ***Executor***. Pour chaque action, l'***Executor*** introspecte le ***SR*** à la recherche est éléments concernés par l'action. Il fait ensuite appel au composant ***RemoteExecutor*** qui implante le moyen d'exécution à distante des actions sur le logiciel ou le SE dans l'environnement réel.

Adaptabilité

L'adaptation du ***ProgramInterpreter*** permet de changer le langage d'expression des programmes de reconfiguration. On pourra ainsi avoir des programmes JAVA ou encore un langage tel que RDL du système TUNe.

L'adaptation de l'***Executor*** permet par exemple d'implanter une méthode d'exécution d'action dans laquelle le lieu d'exécution des actions est soit local (sur la machine d'administration), soit distant (sur le SE du logiciel concerné). En effet, des actions contenues dans les programmes de reconfiguration peuvent concernées des modifications de l'architecture du ***SR***. Dans ce cas, ces actions doivent être réalisées sur la machine d'administration (car c'est elle qui contient le ***SR***).

Pour finir, l'adaptation du ***RemoteConnector*** permet de changer le mécanisme d'exécution à distance des fonctions d'administration. Il pourra utiliser du RMI comme c'est le cas dans TUNe ou encore une exécution basique basée sur un appel de commandes à distance avec du SSH.

8.1.5 Réconfiguration

Après le démarrage des éléments, la suite de l'administration consiste en leur suivi dans le but d'avertir le SAA de leur état courant. La réalisation de cette tâche n'appartient pas au SAA. En effet, elle dépend de l'implantation de chaque élément administré (vu comme une boîte noire). Il revient donc à l'administrateur d'introduire parmi les éléments à administrer, des logiciels (observateurs) qui joueront ce rôle. Les éléments administrés (y compris les observateurs), ont la même considération de la part du SAA. Par contre, il revient au SAA de mettre en place un moyen de communication entre chaque élément administré et lui.

Dans cette situation, la communication est initiée par l'élément administré. Elle s'effectue de son support d'exécution (SE) vers la machine d'exécution du SAA. Le composant ***Event Receiver*** est chargé d'implanter ce mécanisme de communication. Il est composé de deux sous composants : l'***Event Channel*** et l'***Event Driver***. Le premier est utilisé par l'élément administré, s'exécutant sur son SE, pour émettre ses événements. Le second implante la méthode de transport d'événements des SE de l'environnement vers la machine d'administration. L'***Event Driver*** communique l'événement reçu au niveau de la machine d'administration à l'***Event Manager***. Ce dernier implante le module décisionnel du SAA. Son rôle est de choisir en fonction des événements reçus, les programmes d'administration à exécuter pour répondre aux signes émis par l'environnement encours d'administration. Pour cela, il s'appuie sur l'état courant du ***SR***. Après décision des programmes de reconfiguration à exécuter, l'***Event Manager*** fait appel au ***Policies Manager*** pour leur exécution. Le procédé d'exécution celui que nous avons décri dans la section précédente.

Adaptabilité

L'adaptation de l'***Event Channel*** permet de modifier la méthode d'obtention d'événements à partir de l'exécution d'un élément sur son SE.

Quant à l'***Event Driver***, son adaptation permet d'implanter par exemple le mécanisme de transport d'événements par le biais d'un serveur de messages tel que JMS. Ainsi, l'***Event Manager*** s'abonnera au serveur JMS afin d'être informé des événements survenus.

Pour finir la modification de l'***Event Manager*** permettra à l'administrateur de produire un mécanisme de gestion des événements avancé comme la technologie CPE [**?**] (*Complex Event Processing*). Par exemple, il pourra prendre prendre en compte des événements estampiller de l'instant d'envoie d'apparition de l'événement afin de gérer des événements temporels. Ceci lui permet par exemple d'implanter

diverses méthodes de stockage ou d'apprentissage du comportement de l'application en cours d'administration.

8.2 Le formalisme à composants Fractal

Comme nous l'avons évoqué au debut de ce chapitre, le modèle de SAA que nous utilisons repose sur le formalisme Fractal. Ce dernier est basé sur trois notions : les *composants*, les *interfaces*, les *liaisons* et les *contrôleurs* [**?**]. Un *composant* est une entité exécutable. Fractal distingue deux types de composants : les composants primitifs et les composants composites. Les composants primitifs encapsulent une unité exécutable décrite dans un langage de programmation tandis que les composants composites sont des assemblages de composants. Dans ce contexte, un composant Fractal peut appartenir à plusieurs composants composites : on parle de composant partagé.

Une *interface* est un point d'accès ou de sortie sur un composant. Fractal introduit en effet deux catégories d'interfaces :

- *Interface serveur* : il s'agit d'un point d'accès au composant. Un composant définissant une "interface serveur" implique que ce composant fournit les services implantés par cette interface. Dans le cas d'un composant primitif, le code source du composant doit contenir l'implantation de l'interface. Quant aux composites, Fractal ne permet pas de leur associer des codes sources.
- *Interface cliente* : il s'agit d'un point de sortie du composant. Un composant définissant une "interface cliente" signifie que ce composant requiert les services implantés par cette interface. Ainsi, l'exécution de ce composant nécessite au préalable sa liaison avec un autre implantant ce service.

Notons que dans le cas des composants composites, Fractal associe automatiquement à chaque interface serveur (respectivement cliente), une "interface cliente" (respectivement "interface serveur") qui n'est visible que des composants interne au composite : on parle d'interfaces complémentaires (voir la figure 8.2(a)). Ces dernières peuvent être liées aux interfaces des composants internes.

Une *liaison* est un canal de communication établi entre les composants Fractal. De la même façon que les composants, Fractal distingue deux types de liaisons : primitives et composites. Une liaison primitive est une liaison entre deux composants du même espace d'adressage (contenu dans le même composite) via leurs interfaces cliente et serveur. Ce type de liaison est implantée par des pointeurs ou des mécanismes propres au langage de programmation comme les références Java. Une liaison composite est une liaison implantée par un composant ou une chaîne de plusieurs composants. Une liaison peut être multiple dans le sens où une interface cliente peut être liée à plusieurs interfaces serveurs.

Les *contrôleurs* constituent la membrane des composants. Ils ont pour rôles d'exposer les services du composant, d'intercepter les messages en direction du compo-

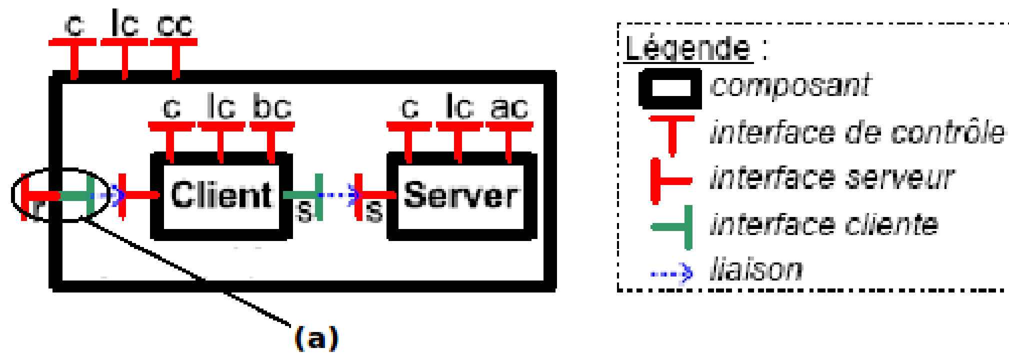


FIGURE 8.2 – Exemple d'architecture Fractal

sant et également d'introspecter le composant. Fractal n'impose pas d'implantation pour la définition des contrôleurs, ce qui permet d'exercer un contrôle adapté sur les composants. Cependant, il fournit quelques exemples de contrôleurs utiles qui peuvent être combinés et étendus :

- Le *contrôleur d'attributs* (***AttributeController***) : il permet de configurer les attributs d'un composant.
- Le *contrôleur de liaisons* (***BindingController***) : il permet d'établir ou de rompre une liaison primitive à partir du composant définissant l'interface cliente.
- Le *contrôleur de contenu* (***ContentController***) : il permet de lister, d'ajouter et de retirer des sous-composants d'un composant composite.
- Le *contrôleur de cycle de vie* (***LifeCycleController***) : il permet de contrôler le cycle de vie du composant qui possède le contrôleur. Le cycle de vie d'un composant est représenté par un automate à deux états : *started* (le composant est dans un état démarré) et *stopped* (le composant est dans un état d'arrêt). Le contrôleur permet de passer d'un état à l'autre. Il est le principal artisan dans les opération de d'ajout/retrait de composants d'une architecture déjà en cours d'exécution.

La figure 8.2 décrit les différentes entités mises en jeu dans une architecture Fractal. Le rectangle noir représente le contrôle du composant alors que l'intérieur du rectangle représente le contenu du composant. Les flèches correspondent aux liaisons tandis que les formes en "T" attachées aux rectangles correspondent aux interfaces du composant. Les interfaces de contrôle sont représentées par des lettres : "c" pour composant, "lc" pour le contrôleur de cycle de vie, "bc" pour le contrôleur de liaisons, "cc" pour le contrôleur de contenu et "ac" pour le contrôleur d'attributs. Les interfaces serveurs (respectivement cliente) sont disposées à gauche (respectivement à droite) du rectangle noir.

L'application décrite dans la figure 8.2 se compose de deux composants primitifs : **Client** et **Server**. Ces composants sont liés par une liaison entre une interface client s du composant "*Client*" et une interface serveur s du composant "*Serveur*".

8.3 Implantation de TUNeEngine

Comme nous l'avons évoquée en préambule de ce chapitre, l'implantation de TUNeEngine utilise Fractal (l'implantation Julia [**?**]). Il est implanté dans un composant composite nommé *TUNeEngine*. Ce composant contient tous les composants constituants le système, y compris les éléments administrés. De façon générale, tous les composants primitifs définissent une interface de gestion d'attributs (ou paramètres). Cette interface étend l'interface *AttributeController* prédéfinie par Fractal.

TUNeEngine s'exécute sur une machine que nous appellerons la machine d'administration. C'est à partir de cette machine qu'il coordonnera toutes les opérations d'administration.

Dans ce chapitre, nous présentons l'implantation TUNeEngine du modèle détaillé que nous avons présenté ci-dessus.Compte tenu du caractère adaptable de notre modèle, l'implantation de ses composants dans cette version de TUNeEngine permet de réaliser toutes les opérations d'administration que permettait le système TUNe. Dans un premier temps, nous présentons l'implantation du composant composite *TUNeEngine*. Ensuite, suivant le même fil conducteur que celui adopté dans la section 8.1, nous présentons l'implantation de chaque composant de TUNeEngine. Pour illustration, nous associerons à chaque section (si nécessaire) une figure résumant la présentation faite dans la dite section. Cette figure présentera le contenu du composite *TUNeEngine* conformément à la présentation, associé parfois au contenu de l'environnement réel d'exécution. Par souci de lisibilité, la figure associée à une section X ne reprendra pas obligatoirement tous les éléments des figures des sections qui lui précèdent.

8.3.1 Le composant *TUNeEngine*

L'ensemble des composants du système TUNeEngine sont contenus au sein d'un composant composite appelé *TUNeEngine*. Ce dernier est vu de l'extérieur comme le système TUNeEngine. Il définit deux interfaces serveurs. La première gère les paramètres d'exécution du SAA. Cette interface est une extension de l'interface *AttributeController* [2] que fournit Fractal. Ces paramètres seront ensuite utilisés pour l'initialisation des composants internes de TUNeEngine. En effet, compte tenu de son statut de représentant de TUNeEngine, son démarrage entraînera celui de ses sous-composants. Or ces derniers ont besoin de certains paramètres pour leur initialisation. C'est le cas par exemple du composant ***CLI*** qui requiert pour son démarrage le niveau de verbosité indiquant le degré de détails avec lequel TUNeEngine doit répondre aux requêtes des administrateurs.

La deuxième interface est le démarreur de TUNeEngine. Elle démarre tous les composants de services et ne rend la main qu'à l'arrêt de ces derniers. Compte tenu

2. En effet, l'interface de contrôle *AttributeController* n'est pas prédéfinie dans Fractal pour les composites

de son caractère de composant composite, *TUNeEngine* contient un composant primitif *TUNeEngineDelegate* qui implémente réellement ses deux interfaces serveurs. Ce composant est relié à tous les autres sous-composants de *TUNeEngine* afin de leur communiquer leurs paramètres d'initialisation.

Le composant *TUNeEngineDelegate* est implanté par une classe Java définissant une interface *AttributeController* et *Thread*.

8.3.2 Soumission de commandes et Collaboration

L'implantation du composant ***CLI*** dans TUNeEngine obtient les requêtes de l'administrateur via une socket réseau et fournit les résultats via une console locale. Les paramètres d'initialisation réseau (nom de la machine d'exécution et numéro de port) et le niveau de verbosité lui ont été fournis lors du démarrage de *TUNeEngine*. Comme le composant *TUNeEngineDelegate*, il définit deux interfaces. La première gère les paramètres d'initialisation tandis que la seconde démarre le démon d'écoute de requête et d'acheminement de résultats.

Quant au composant ***Collaborator***, nous utilisons le mécanisme de communication à distance *RMI* pour effectuer la collaboration avec d'autres SAA. Ce mécanisme nécessite la présence d'un annuaire d'enregistrement d'objets RMI dans le réseau. Ainsi, le démarrage de ***Collaborator*** requiert plusieurs paramètres d'initialisation dont : le nom de l'instance de TUNeEngine en cours d'exécution (*TUNeEngineName*), le nom DNS de la machine d'exécution, le nom DNS de l'annuaire d'enregistrement RMI et son numéro de port. Il implante une interface RMI dont l'exportation (à travers l'annuaire RMI) permet à un SAA distant de soumettre des requêtes à TUNeEngine. L'enregistrement de cette interface se fait au démarrage sous le nom *TUNeEngineName*. Quelque soit le composant de communication externe (***Collaborator*** ou ***CLI***), les natures des requêtes émises sont identiques. Toute requête reçue est soumise au ***CmdInterpreter*** avec qui ils sont liés à travers une interface cliente. **N.B.** : Dans une évolution future, nous envisageons à la place de la technologie RMI utilisée pour la collaboration, l'utilisation des web services basés sur WSDL.

Dans sa version actuelle, le composant ***CmdInterpreter*** ne prend en compte que les requêtes de la forme : une ligne équivaut à une requête. La commande est représentée par le premier mot de la ligne et le reste correspond aux paramètres de la commande. Son adaptation permettra de modifier la syntaxe de soumission de commandes au SAA. Voici la liste de quelques commandes prises en compte actuellement par le ***CmdInterpreter*** :

- L'arrêt de TUNeEngine : entraîne l'arrêt de tous les services internes.
- La soumission d'un environnement à administrer : entraîne l'appel du composant composite ***SR Manager*** ;
- Le déploiement/undéploiement : entraîne l'appel du composant ***Deployment Manager*** ;
- L'exécution d'une politique de reconfiguration : entraîne l'appel du composant ***Policies Manager*** ;

– etc.

L'enrichissement de cette liste de commandes se fait par ajout, dans le composant composite *TUNeEngine*, des composants remplissant les services requises par les commandes qui seront ajoutées. La figure 8.3 résume l'implantation Fractal de l'ensemble TUNeEngine à ce stade.

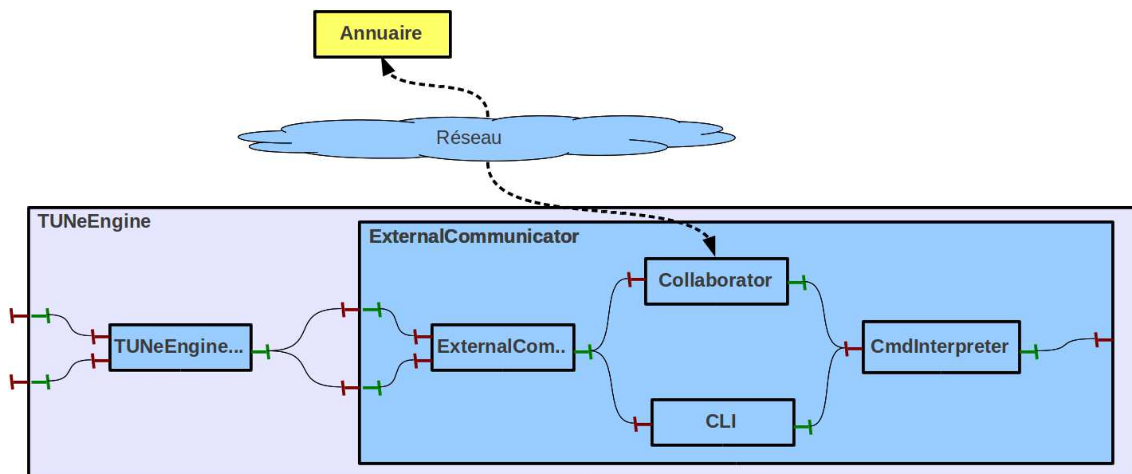


FIGURE 8.3 – Soumission de commandes et Collaboration

8.3.3 Construction du SR

Cette étape représente le cœur de TUNeEngine. En effet, elle permet de construire la structure de données contenant tous les éléments administrés et d'administration (machines, logiciels, et les programmes d'administration). Dans notre implantation, ces éléments sont fournis à TUNeEngine de deux façons : sous forme d'ADL [3] Fractal ou par des composants Fractal. Cette dernier méthode consiste de la part de l'administrateur (averti) à construire l'architecture Fractal de ses éléments et ensuite les l'insérer dans le composite *TUNeEngine*. Dans les deux cas, les API Fractal implantent tout ou partie des rôles des composants ***Parser***, ***Wrapper*** et ***SR***.

Lorsque les descriptions de l'environnement et des programmes de reconfiguration sont fournies par ADL, le ***Parser*** est entièrement implanté par appel aux API Julia de Fractal. Dans le cas où les composants sont directement fournis en Fractal, le ***Parser*** ne joue aucun rôle. En ce qui concerne le ***Wrapper***, il implante le comportement générique et adaptable (voir ci-dessous) des éléments du SR et obtient des administrateurs des comportements spécifiques. L'administrateur fournit une classe Java implantant toutes les fonctions d'administration de l'élément à encapsuler. Cette classe implante deux types de méthodes d'encapsulation : la première s'applique aux éléments administrés tandis que la seconde s'applique aux programmes d'administration. Quant au ***SR***, une partie de son implantation est fournie

3. Architecture Description Language

par les API Fractal. Son implantation est par la suite complétée par la réalisation des liaisons Fractal entre les éléments encapsulés et les composants encapsulant les services de TUNeEngine (***Deployer***, ***Policies Manager***, et ***Event Manager***).

Encapsulation des éléments administrés

Chaque élément administré est encapsulé dans un composant primitif. Dans certains cas, les éléments appartenant à une même famille (selon l'administrateur) peuvent être regroupés dans des composants composites. Par exemple, soit une application à administrer constituée de plusieurs modules. Soit un module contenant plusieurs logiciels. Dans cet exemple, TUNeEngine permet l'encapsulation du module à l'intérieur d'un composant composite contenant les composants primitifs encapsulant les logiciels de ce module. Le même raisonnement peut également être appliqué pour un cluster (dont le ***Wrapper*** fournit les procédures d'administration) contenant plusieurs SE. Dans tous les cas, les interfaces de gestion des composants sont identiques (critère d'uniformité de notre SAA).

Le ***Wrapper*** associe à chaque composant les interfaces suivantes : une interface serveur de gestion des attributs du composant (les paramètres de l'élément encapsulé), deux interfaces cliente et serveur (le nombre d'interfaces pouvant être modifié par adaptation du ***Wrapper***) lui permettant d'être relié à d'autres composants ; une interface serveur d'exécution de fonctions d'administration (qui se trouvent dans les programmes d'administration) et une d'interface serveur "*deploy*" lui permettant d'être dépoyé/undeployé. L'implantation de cette dernière est fournie par l'administrateur et sera exécuté par le composant ***Deployer*** pendant la phase de déploiement. Cependant, dans son implantation actuelle dans TUNeEngine, le Wrapper fournit une implantation basique reprenant celle du système TUNe. Nous revenons sur cette interface dans la section suivante. Notons que l'implantation de toutes ces interfaces par le ***Wrapper*** n'est utilisée qu'en comportement par défaut. En effet, l'administrateur est capable de fournir par adaptation des comportements spécifiques aux éléments en cas de besoins. Par exemple, il fournira en fonction de la nature d'un SE présent dans son architecture, différents moyens de connexion sur ce SE (utilisable pendant les opérations de déploiement).

Encapsulation des programmes d'administration

Les programmes d'administration sont regroupés dans des composites nommés *AunonomicManager*. Un *AutonomicManager* peut contenir plusieurs programmes d'administration. Le ***Wrapper*** introduit dans chaque *AunonomicManager* un composant primitif (le *Dispatcher*) chargé de choisir le programme d'administration à exécuter (voir la section 8.3.5 pour plus de détails). Dans l'implémentation actuelle de TUNeEngine, les programmes d'administration sont des automates à états. Chaque état est un composant primitif implantant l'action à effectuer. Ce composant définit une interface serveur *run* et deux interfaces clientes *sr* et *next*. L'interface *run* permet de déclencher l'exécution de l'action. Cette interface doit être fournie par l'administrateur. L'implantation de l'action a la possibilité d'accéder au ***SR*** via l'interface *sr* de l'état. En effet, certaines actions de reconfiguration peuvent entraîner la modification du SR ou encore nécessiter la récupération des paramètres de certains éléments du SR. Pour finir, l'interface *next* permet de relier l'état aux autres. Elle

donne ainsi le sens de progression de l'exécution de l'automate.

Chaque automate contient un point d'entrée (l'état initial) et un point de sortie (l'état final). L'état initial renseigne le *Dispatcher* (via un attribut) des événements pour lesquels l'automate qu'il représente peut être exécuté. Nous présentons dans la section 8.3.5, la description de l'exécution des programmes d'administration dans TUNeEngine.

La figure 8.4 résume l'organisation des composants Fractal dans le SR et dans le composant *TUNeEngine*. Les composants de service de TUNeEngine ne sont pas tous représentés dans cette figure. Ils sont symbolisés par un unique composant que nous appelons *Service*.

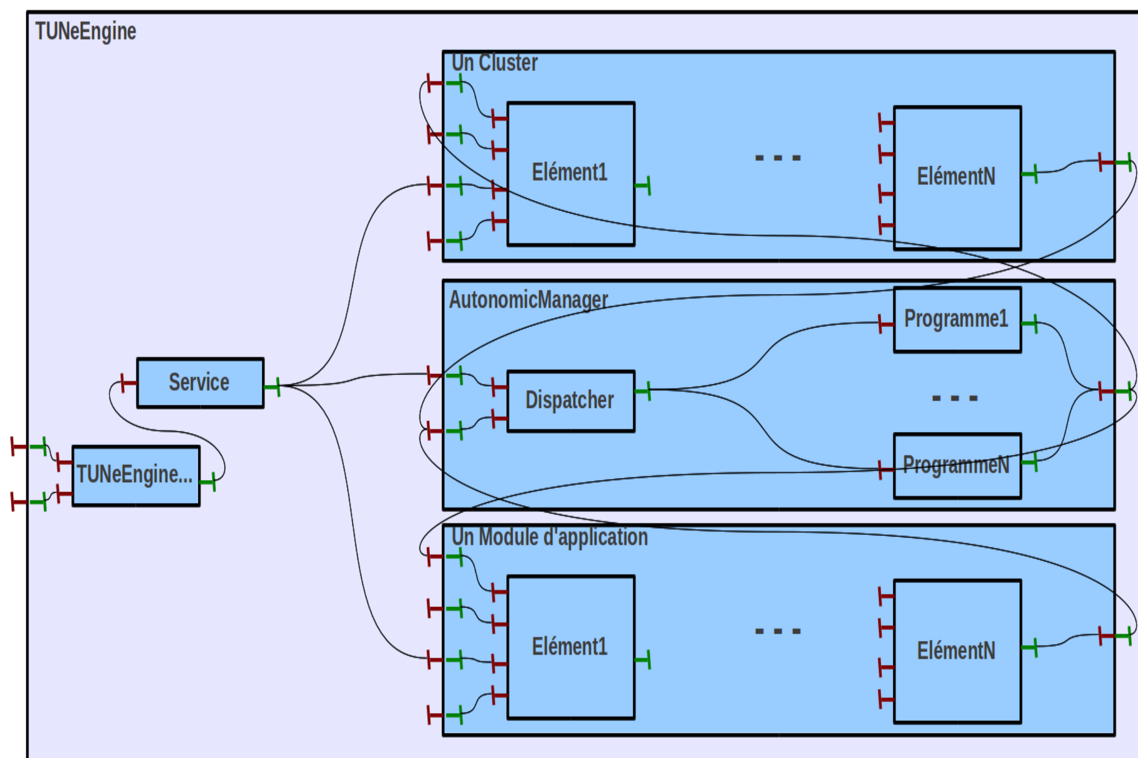


FIGURE 8.4 – Construction du SR

8.3.4 Déploiement

Le composant ***Deployment Manager*** de notre modèle est implanté par un composant composite de même nom dans TUNeEngine. Ce composant implante à la fois les fonctionnalités de déploiement et de undéploiement. Il définit deux interfaces serveurs (*deploie* et *undeploie*) et une interface cliente (*sr*). L'interface *deploie* démarre le processus de déploiement d'un ou des éléments du système de représentation. Quant à l'interface *undeploie*, elle effectue l'opération inverse. L'interface *sr* permet de relier le composant ***Deployment Manager*** au ***SR*** afin qu'il puisse y

retrouver tous les éléments administrés.

Dans notre implantation, le ***Deployer*** ne réalise pas proprement dit les opérations de déploiement et de undéploiement. Son rôle est de préparer, orchestrer, et compléter ces opérations. Soit E un élément à déployer : voici la réalisation du déploiement de E dans TUNeEngine. L'ordre de déploiement de E contient plusieurs informations : l'élément à déployer, son SE ou les caractéristiques que doivent respecter le SE choisi par ***NodeAllocator***. Lorsque le SE n'est pas fourni, le ***Deployer*** sollicite le composant ***NodeAllocator*** afin que celui-ci lui fournisse un SE selon les caractéristiques fournies dans l'ordre de déploiement. Pour cela, le composant ***NodeAllocator*** accède au SR via l'interface *sr* du ***Deployment Manager***. ***NodeAllocator*** implante deux interfaces serveurs : *allocate* (pour l'allocation de SE) et *free* (pour la libération d'un SE). L'implantation de ces deux interfaces (qui représente les politiques d'allocation et de libération de SE) est entièrement fournie par l'administrateur. Dans son implantation actuelle, TUNeEngine fournit une politique d'allocation basée sur l'algorithme de tourniquet. Le ***Deployer*** communique avec le ***NodeAllocator*** via ces deux interfaces.

Une fois le SE choisi par le ***NodeAllocator*** ou fourni dans l'ordre de déploiement, le ***Deployer*** s'assure que le SE estest préalablement déployé. Dans le cas contraire, il démarre son processus de déploiement. Concrètement, les opérations de déploiement et undéploiement proprement dites sont réalisées par les éléments administrés eux-mêmes. Comme nous l'avons présenté dans la section précédente, le ***Wrapper*** ajoute à chaque élément encapsulé deux interfaces serveurs pour réaliser ces tâches.

Concernant le SE, l'implantation particulière de son processus de déploiement consiste en :

- Déploiement (comme c'était déjà le cas avec le système TUNe) sur le SE des binaires et codes sources nécessaire pour exécuter le représentant de TUNeEngine : *RemoteLauncher*. Ce dernier est le bras de TUNeEngine sur le SE.
- Démarrage sur le SE du *RemoteLauncher* et enregistrement de sa référence RMI dans le registre RMI démarré sur la machine d'administration TUNeEngine.

Le représentant du SE dans le ***SR*** contient le moyen de copie et d'exécution à distance de commandes sur le SE lorsque celui-ci ne contient encore aucun représentant de TUNeEngine (*RemoteLauncher*).

En ce qui concerne les autres éléments faisant l'objet du déploiement, le processus de déploiement est le suivant :

- Déploiement (comme c'était déjà le cas avec le système TUNe) sur le SE de l'élément à déployer, des binaires et codes sources nécessaires pour exécuter le représentant de TUNeEngine de cet élément son SE : *RemoteWrapper*. Ce dernier est chargé d'exécuter sur le SE les futures actions de reconfiguration.
- Démarrage du *RemoteWrapper* par le *RemoteLauncher*. Ce dernier enregistre également la référence RMI du *RemoteWrapper* dans l'annuaire RMI sur la machine d'administration. Nous verrons dans la section suivante que cette référence est utilisée plus tard pour l'exécution à distance des actions de reconfiguration.

– Démarrage du démon jouant le rôle d'***Event Channel*** sur le SE. Nous verrons dans la section suivante l'implantation de ce démon.

Après le déploiement et le démarrage des représentants, le processus de déploiement des éléments à administrer se déroule comme décrit dans la section 8.1.3.

Le composants ***Binary Manager*** dans notre modèle n'est pas implanté sous forme de composant Fractal. Il est implanté comme méthodes Java dans la classe d'implantation (fournie par l'administrateur) des interfaces *deploie* et *undeploie*. Dans son comportement par défaut (fourni par TUNeEngine et adaptable), le déploiement (respectivement le undéploiement) se limite à la copie (respectivement la suppression) de fichiers de la machine TUNeEngine vers un répertoire particulier sur le SE. Il s'agit du comportement que proposait le système TUNe.

Inversement, dans le cadre du undeploiement, le ***Deployer*** s'assure que l'élément undeployé n'héberge plus d'éléments. Il défait également la liaison entre l'élément undeployé et son SE.

La figure 8.5 résume l'implantation Fractal de ces opérations ainsi qu'une vue globale de l'environnement d'environnement après cette étape. Les éléments du SR dans cette figure sont symbolisés par un élément symbolique nommé SR. On peut observer sur cette figure la relation entre chaque logiciel (pouvant jouer le rôle de sonde) et l'***Event Channel***. Remarquons également que comme les logiciels, un SE (représenté par son *RemoteLauncher*) peut se voir associer un logiciel de sonde qui communiquera son état à l'***Event Driver***. Pour finir, les références des *RemoteWrapper* et *RemoteLauncher* sont enregistrées dans l'annuaire RMI de la machine d'administration.

8.3.5 Exécution de programmes d'administration

Nous regroupons dans cette section les étapes de configuration, démarrage et de reconfiguration. Pour toutes ces étapes, ils s'agit globalement d'exécution de programmes d'administration. Comme nous l'avons évoqué dans la section 8.3.3, les programmes d'administration sont encapsulés dans des composants Fractal que nous appelons *AutonomicManager*. Dans l'implantation de TUNeEngine, ils peuvent être déclenchés soit par une commande issu du composant ***External Communicator***, soit par des éléments administrés. Dans le premier cas, l'événement est directement transmis à l'***Event Manager*** par le composant ***External Communicator***. Dans le second cas, ces événements sont acheminés du SE jusqu'à la machine d'administration par le composant ***Event Driver***. Ce dernier est implanté par un composant Fractal s'exécutant sur la machine d'administration. Ce composant dispose de deux interface : l'interface serveur "er" pouvant être exécutée par RMI et une interface cliente "em" permettant de contacter l'***Event Manager*** en cas d'événements. Le composant ***Event Driver***, démarré au démarrage de *TUNeEngine*, enregistre la référence RMI de l'interface "er" dans l'annuaire RMI située sur la machine d'administration. L'implantation de cette interface contactera le composant ***Event Manager*** via l'interface cliente "em" de l'***Event Driver***.

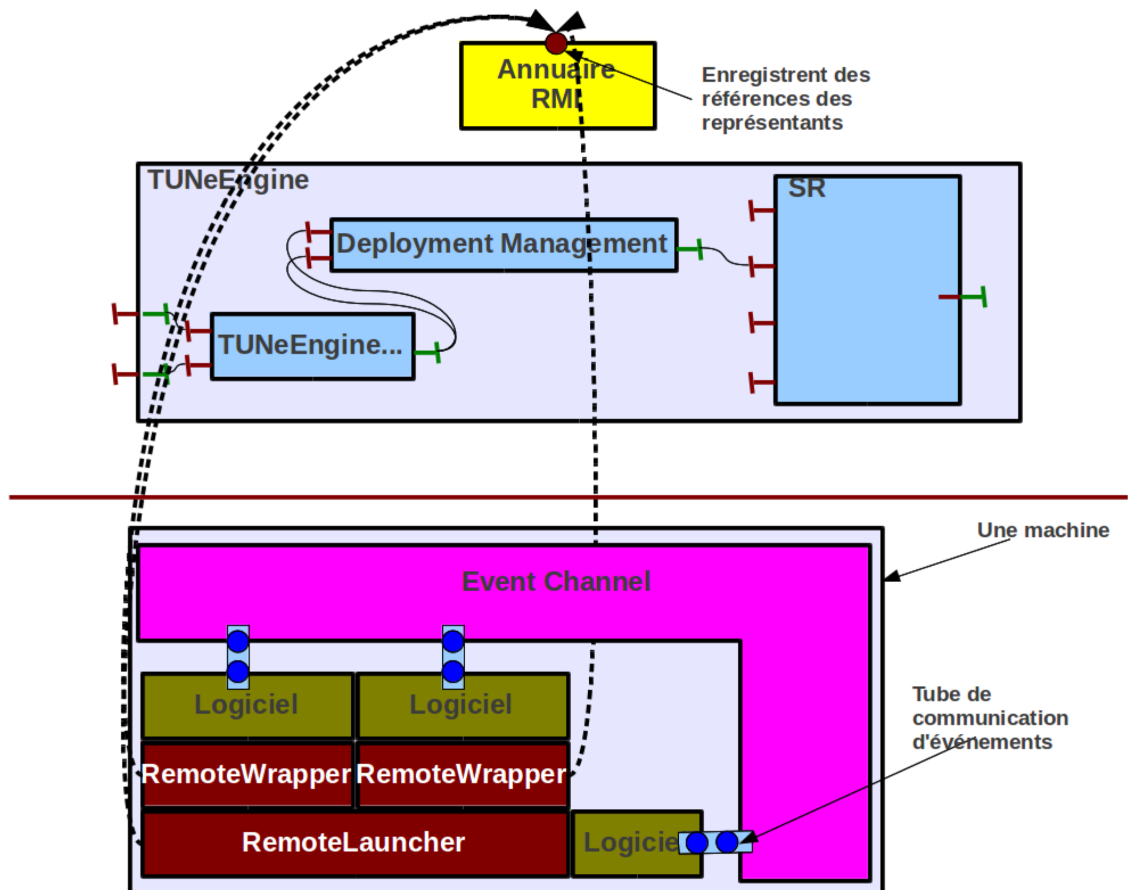


Figure 8.5 – Déploiement et Undeploiement

Comme nous l'avons évoqué dans la section précédente, le composant ***Event Channel*** est implanté sous forme de démon, associé à chaque élément administré et exécuté à distance. Pendant son démarrage, il initialise le tube dans lequel l'élément administré qui lui est associé posera ses événements. Il obtient du registre RMI de la machine d'administration, la référence de l'interface "er" du composant ***Event Driver***. Cette interface implante la méthode *notify* qui permet d'envoyer un événement à TUNeEngine. L'***Event Channel*** associe à chaque événement la reférence de l'élément administré à l'origine de l'événement. Il peut être enrichi par adaptation afin d'y associer des informations comme l'instant d'envoie de l'événement.

Le composant ***Event Manager*** reçoit les événements via son interface serveur "em". Il définit en outre d'autres interface Fractal : une interface cliente "sr" lui permettant d'avoir accès au ***SR***, et une interface cliente "pm" lui permettant d'avoir accès au composant ***Policies Manager***. L'implantation actuelle de l'interface serveur "em" de ***Event Manager*** prend en compte tous les événements reçus et ne comprend aucune logique décisionnelle. Il choisit via le *Dispatcher* de chaque *AutonomicManager*, les programmes de reconfiguration à exécuter. Ce choix est dicté par un attribut de l'état initial du programme. Cet attribut contient une liste d'événements pour lesquels le programme est apte.

Après décision des programmes à exécuter, l'***Event Manager*** fait appel au composant ***Policies Manager*** en lui passant la liste des programmes d'administration (références des composants encapsulant les états initiaux) des *AutonomicManager* à exécuter. L'implantation de l'***Event Manager*** est réalisée par un composant composite contenant deux composants primitifs : ***Executor*** et ***ProgramInterpreter***. Ce composant composite définit : une interface serveur "pm" reliée à l'interface "execute" de ***Executor*** ; une interface cliente "sr" lui permettant d'accéder au SR ; et une interface cliente "re" reliée au ***RemoteExecutor***. Le composant ***Executor***, via son interface "execute", parcourt chaque programme en partant du composant représentant l'état initial. Pour chaque état, l'action à exécuter est définit soit dans un attribut, soit par une interface serveur "run" de l'état, soit les deux. Lorsqu'elle est définie par un attribut, l'***Executor*** fait appel au ***ProgramInterpreter*** afin d'interpréter l'action à effectuer. Les deux composants sont reliés via leur interface "interprete". Dans son implantation actuelle, le ***ProgramInterpreter*** implante l'interprétation du langage RDL (section 5.2.3 du chapitre 5) du système TUNe. Cette implantation peut évidemment être modifiée afin de prendre en compte un nouveau langage. Dans tous les cas, l'interface "run" est exécutée à la fin de l'interprétation. L'utilisation de l'interface "run" est réservée aux administrateurs avertis. En effet, ils devront implanter pour chaque action (programme Java ici), le parcours du SR et l'appel du mécanisme d'exécution à distance pour l'exécution proprement dite des fonctions de reconfiguration sur la machine distante. Par contre, dans le cas d'une action interprétée, cette tâche est factorisée dans le ***ProgramInterpreter*** et est générique pour toutes les actions. De plus, une fois les actions interprétées, l'***Executor*** se charge d'exécuter réellement l'action sur le SE distant de l'élément concerné par l'action. Pour cela, il fait appel au *RemoteExecutor* via son interface "re" reliée à l'interface "re" du composite ***Policies Manager***.

L'implantation de l'interface serveur "re" du ***RemoteExecutor*** obtient du registre RMI la référence RMI du *RemoteWrapper* de l'élément sur lequel il doit exécuter une action. Le *RemoteWrapper* implante une interface "effect" qui permet d'exécuter réellement la fonction de reconfiguration sur le SE distant. L'implantation de l'interface "effect" utilise les mécanismes d'introspection et de reflexion Java pour identifier et exécuter la fonction sur le SE distant. Pour finir l'***Executor*** constate la fin de l'exécution du programme de reconfiguration lorsqu'il atteint l'état final du programme. La figure 8.6 résume l'implantation Fractal de TUNeEngine ainsi que les interactions entre les différentes éléments distants constituant l'environnement administré. Le transfert d'événement et l'exécution à distance des fonctions de reconfiguration sont réalisés par des appels RMI. Les représentants *RemoteWrapper* et *RemoteLauncher* jouent les rôles d'effecteurs sur les éléments en cours d'exécution sur les SE distants.

8.4 Synthèse

Dans ce chapitre nous avons présenté le modèle détaillé de l'architecture d'un SAA hautement adaptable, flexible et remplissant les critères que nous avons iden-

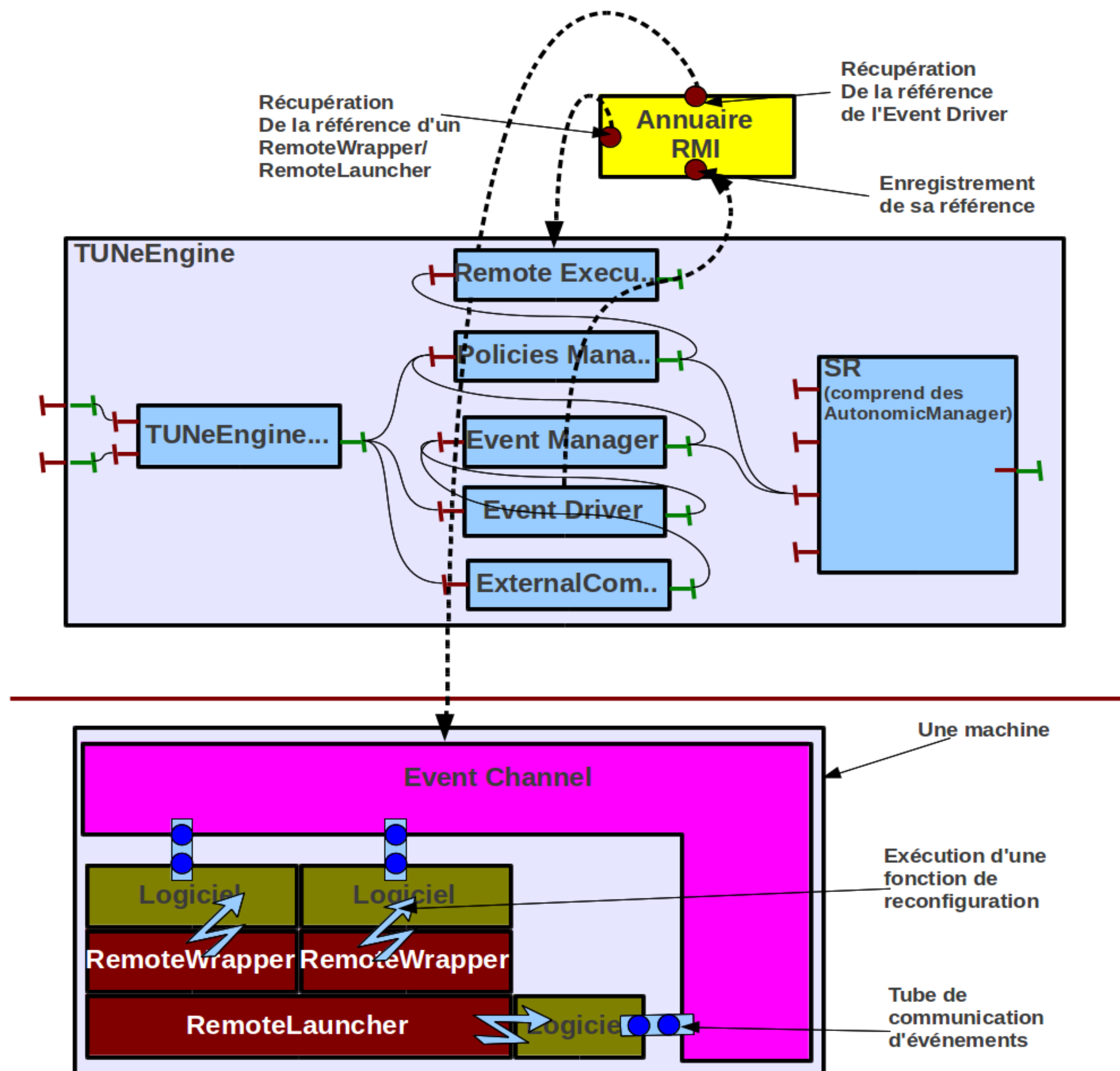


FIGURE 8.6 – Exécution de programmes d'administration

tifiés dans le chapitre 6. L'implantation de ce modèle, baptisée TUNeEngine, est une première validation dont les composants remplissent les services du système TUNe son inspirateur. Globalement, tous les éléments administrés sont gérés de la même façon. Chaque élément dispose d'un représentant (*wrapper*) dans le système de représentant de TUNeEngine et également d'un représentant (*RemoteWrapper* ou *RemoteLauncher*) sur le SE d'exécution distant. Pour réaliser aussi bien la communication des événements que l'exécution des fonctions d'administration à distance, TUNeEngine utilise des appels RMI (implantées respectivement par l'***Event Receiver*** et le ***RemoteExecutor***).

Comme nous l'avons évoqué tout au long de ce chapitre, tous les mécanismes utilisés par TUNeEngine sont adaptables. Le chapitre suivant montre notamment une adaptation et utilisation de TUNeEngine pour l'administration d'une part du Cloud et d'autre part des applications de niveau entreprise qu'il héberge.

Chapitre 9

Contributions : application au Cloud

Contents

9.1 Un IaaS simplifié : CloudEngine **105**
9.1.1 Vision globale : CloudEngine-TUNeEngine, Application-TUNeEngine-CloudEngine 107
9.1.2 RepositoryController 108
9.1.3 VMController 111
9.1.4 VLANController 112
9.1.5 ResourceController 112
9.1.6 MonitoringController 113
9.1.7 Scheduler 114
9.2 Utilisation de CloudEngine **114**
9.2.1 Construction et enregistrement de VM 115
9.2.2 Administration d'applications 115
9.3 Reconfiguration d'applications et de CloudEngine **116**
9.3.1 Réparation de VM 116
9.3.2 Consolidation de VM et Scalabilité de serveurs 117
9.3.2.1 Exécution d'applications statique 118
9.3.2.2 Exécution d'applications variables 118
9.4 Synthèse **119**

La première problématique de ce travail de thèse fut l'administration du cloud ainsi que des applications qu'il héberge. L'ors d'une tentative d'utilisation du système d'administration autonome TUNe, nous nous sommes heurtés à la difficulté d'utilisation de TUNe dans ce nouveau contexte qu'est le cloud. C'est ainsi que nous avons entrepris de concevoir et d'implanter un nouveau SAA le plus générique et flexible possible de tel sorte que son utilisation pour l'administration du cloud et ses applications soit une personnalisation de ce SAA (évitant une ré-implantation fastidieuse). Dans le chapitre précédent, nous avons présenté une implantation (baptisé

TUNeEngine) de SAA hautement adaptable. Nous validons dans le chapitre courant l'adaptabilité de ce système en montrant comment il peut être utilisé pour (1) l'administration d'une plateforme de cloud et (2) l'administration des applications entreprises s'exécutant dans le cloud.

Comme nous l'avons présenté dans l'état de l'art, plusieurs travaux autour du cloud s'intéressent à l'administration du cloud en proposant des solutions plus ou moins complètes. De plus, ces solutions sont pour la plupart intégrées et liées à une unique plateforme de cloud. Afin d'avoir un degré de liberté dans l'application de TUNeEngine à l'administration du cloud, nous proposons dans le cadre de cette thèse l'implantation d'un système simplifié de cloud, baptisé CloudEngine. Ce dernier se limite uniquement à l'IaaS et ses services de base. Les services liés à la facturation ou à l'authentification des clients ne sont pas implantés dans CloudEngine. Cependant, cette application de TUNeEngine à notre plateforme simplifiée de cloud ne limite pas son application à d'autres plateformes offrant des composants clairement identifiables. C'est le cas d'OpenNebula [?] pour lequel nous avons utilisé TUNeEngine pour automatiser son administration.

Ce chapitre est organisé de la façon suivante. Dans un premier temps, nous présentons l'implantation de CloudEngine ainsi que des adaptations de TUNeEngine pour le prendre en compte. Ensuite, nous présentons l'administration de CloudEngine par le système TUNeEngine adapté. Dans cette présentation, nous nous limitons aux tâches de déploiement et de démarrage de CloudEngine. Ensuite, nous présentons l'adaptation et l'utilisation de TUNeEngine pour l'administration des applications entreprises devant s'exécuter dans CloudEngine. La fin de ce chapitre est consacrée à la présentation de l'implantation de quelques politiques d'administration de l'ensemble "CloudEngine-applications entreprises" par des instances de TUNeEngine.

9.1 Un IaaS simplifié : CloudEngine

Comme nous l'avons annoncé à la fin du chapitre 2, nous nous intéressons aux plateformes de cloud dans lesquelles l'isolation est réalisée par les systèmes de virtualisation. Ainsi, l'implantation d'IaaS simplifié que nous proposons (CloudEngine) s'appuie sur un système de virtualisation de type hyperviseur (section 2.2.4 du chapitre 2). Le système CloudEngine s'inspire des implantations d'IaaS existants tels que OpenNebula [?] et Eucalyptus [?]. Il implante la partie IaaS du modèle de cloud que nous avons présenté dans le chapitre 3 à la figure 3.2.

La figure 9.1 montre l'organisation des services de l'IaaS de qu'implante CloudEngine. Ces services interagissent entre eux afin d'accomplir leurs taches. Les éléments en bleu sur la figure 9.1 représentent les services tandis que les éléments en jaune représentent les ressources du cloud. Une flèche pleine entre deux éléments montre qu'un élément peut acquérir les services de l'élément pointé. En bref, les services fournis par CloudEngine sont :

- *RepositoryController* : gère la soumission des images OS en provenance de l'extérieur. Il crée pour chaque image OS, un *Repository* chargé de recevoir et de stocker l'image. L'exécution d'une VM utilisant une image *Repository* pourra se servir d'une copie *ExecutedVMRepository* de cette image.
- *VMController* : gère la création, la migration, l'arrêt et la sauvegarde des VM. Il fournit les interfaces d'accès aux hyperviseurs présents sur les machines de l'IaaS. Pour son fonctionnement, le *VMController* a besoin des services des autres composants du cloud. Pour le démarrage d'une VM par exemple, il obtiendra du *RepositoryController* l'image OS qu'utilisera la VM.
- *VLANController* : gère les réseaux virtuelles dans lesquels les VM du cloud seront confinées. Il est sollicité par le *VMController* pour la configuration ou la libération des adresses réseaux des VM lors de leur création, destruction ou migration. A la fin de son intervention, les VM crées doivent être accessibles de l'extérieur par leur propriétaire.
- *ResourceController* : s'occupe de l'allocation et de la libération des ressource dans le cloud. Il fournit au *VMController* la machine la plus adéquate pour l'exécution d'une VM dans le cloud. Comme nous le verrons dans la suite (observé également sur la figure 9.1), le *ResourceController* peut également être sollicité par d'autres composants du cloud.
- *MonitoringController* : gère l'état des ressources matérielles de l'IaaS ainsi que celui des VM en cours d'exécution. Les informations qu'il dispose peuvent être utilisées par les autres composants du cloud pour la réalisation de leurs tâches. En guise d'exemple, l'allocation de ressources aux VM par le *ResourceController* nécessite de sa part l'établissement d'une cartographie de l'ensemble des ressources du cloud. Cette cartographie est réalisée à partir des informations obtenues du *MonitoringController*.
- *Scheduler* : implante les politiques de relocalisation et consolidation de VM en cours d'exécution dans le cloud. Pour cela, il utilise les services du *MonitoringController* (pour l'obtention des états des VM), du *ResourceController* (pour l'obtention des machines de relocalisation/consolidation) et du *VMController* (pour les relocalisations/consolidations proprement dites).

Dans cette section, nous présentons l'implantation de ces services ainsi que leur intégration dans TUNeEngine. Dans un premier temps, nous présentons une visions globale de cette implantation ainsi que son intégration dans TUNeEngine. Ensuite, nous présentons en détails les composants de CloudEngine et leur implantation dans TUNeEngine. Notons qu'après l'adaptation de TUNeEngine pour implanter l'équivalent du système TUNe, cette adaptation de TUNeEngine est la seconde validation de l'efficacité de notre modèle en terme de respect des caractéristiques que nous avons identifiées dans le chapitre 6.

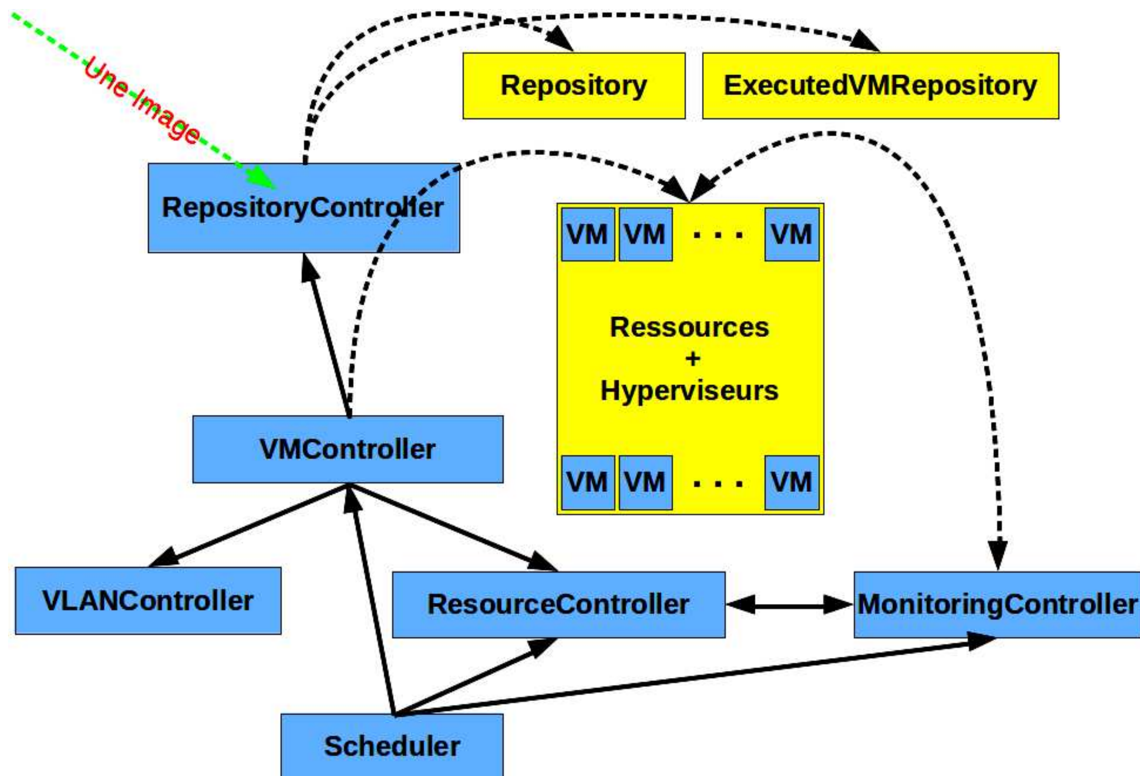


FIGURE 9.1 – CloudEngine

9.1.1 Vision globale : CloudEngine-TUNeEngine, Application-TUNeEngine-CloudEngine

Comme toute application administrée par un SAA, l'administration de notre plateforme simplifiée de cloud par TUNeEngine nécessite l'organisation de ses composants en deux catégories : les composants administrables (déployés sur les SE distants) et les politiques d'administration (déclenchées par des événements). Dans le cadre de CloudEngine, la majorité de ses éléments sont intégrés dans TUNeEngine comme programmes d'administration (plus précisément programmes de reconfiguration). Comme le montre la figure 9.2(2)(b), il s'agit du *VMController*, du *RepositoryController*, du *VLANController*, du *VMController* et du *Scheduler*. Ces composants se mettent en marche lorsqu'une entreprise cliente du cloud contacte le cloud pour soumission d'image OS ou allocation de VM. A ce sujet, les entreprises utilisent également des instances de TUNeEngine (figure 9.2(1)) (à travers son composant ***ExternalCommunicator***) pour le déploiement de leurs applications dans le cloud.

En ce qui concerne les éléments administrés, nous retrouvons le *MonitoringController*, les *Repository*, les VM et les VLAN (figure 9.2(2)(a)). A l'exception du *MonitoringController*, les éléments administrés sont crées dynamiquement par les programmes de reconfiguration présentés précédemment. Les VM par exemple sont créées par le *VMController*. Pour finir, le *ResourceController* est une adaptation

du composant ***NodeAllocator***, sous-composant du ***Deployer*** de TUNeEngine (figure 9.2(1)(d)).

L'administration de CloudEngine par TUNeEngine nécessite préalablement la description de son architecture et de ses politiques d'administration dans un langage quelconque. Dans l'attente de l'implantation des API de la couche langage de TUNeEngine (figure 5.7 de la section 5.5), nous conservons les composants de TUNeEngine permettant de réaliser cette description à savoir le (***Parser*** et ***ProgramInterpreter***). Rappelons que ces composants sont prédisposés à recevoir des descriptions en ADL de Fractal et notation pointée (RDL) du système TUNe (section 5.2.3 du chapitre 5). Le premier permet de décrire l'architecture des applications qu'administre TUNeEngine (CloudEngine et applications entreprises notamment). Quant au second langage, il simplifie la description des actions d'administration contenues dans les programmes d'administration à exécuter par TUNeEngine. Pour cette adaptation de TUNeEngine, ce second langage est une extension de celui implanté par le système TUNe.

Les éléments à administrer ainsi que les politiques d'administration de CloudEngine sont définis à l'aide de fichiers ADL Fractal. Ces derniers sont ensuite exécutés par TUNeEngine pendant son démarrage. Cette exécution de TUNeEngine démarre le cloud. L'instance d'exécution de TUNeEngine représente le point d'entrée de notre platefome de cloud : nous l'appelons *frontend* dans ce document. L'enregistrement de la référence du *frontend* dans l'annuaire de collaboration (voir la section 8.3.2 du chapitre précédent) permettra aux entreprises de réserver et d'exécuter des VM dans le cloud.

De façon générale, les instances TUNeEngine de niveau entreprise administrent les applications entreprises de la même façon que le système TUNe. A l'exception du ***NodeAllocator*** de ces instances, l'implantation des composants de TUNeEngine est identique à celle que nous avons présentée dans le chapitre précédent. En effet, le ***NodeAllocator*** n'obtient pas les composants SE (sur lesquels seront déployés les logiciels) du SR de son instance TUNeEngine. Il contacte l'instance TUNeEngine de CloudEngine (par le biais de l'***ExternalCommunicator*** des deux instances TUNeEngine du cloud et d niveau entreprise) afin d'obtenir les SE (qui sont des VM).

9.1.2 RepositoryController

Le *RepositoryController* gère les soumissions de systèmes de fichiers de VM dans le cloud. Chaque système de fichiers est contenu dans un fichier au format iso que nous appelons également "image". Pour cela, nous réservons dans l'environnement matériel du cloud une machine de stockage. Chaque image est gérée par un composant *Repository* qui s'exécute sur le serveur de stockage et s'occupe du stockage de l'image.

Le *RepositoryController* attend continuellement des requêtes sur deux canaux

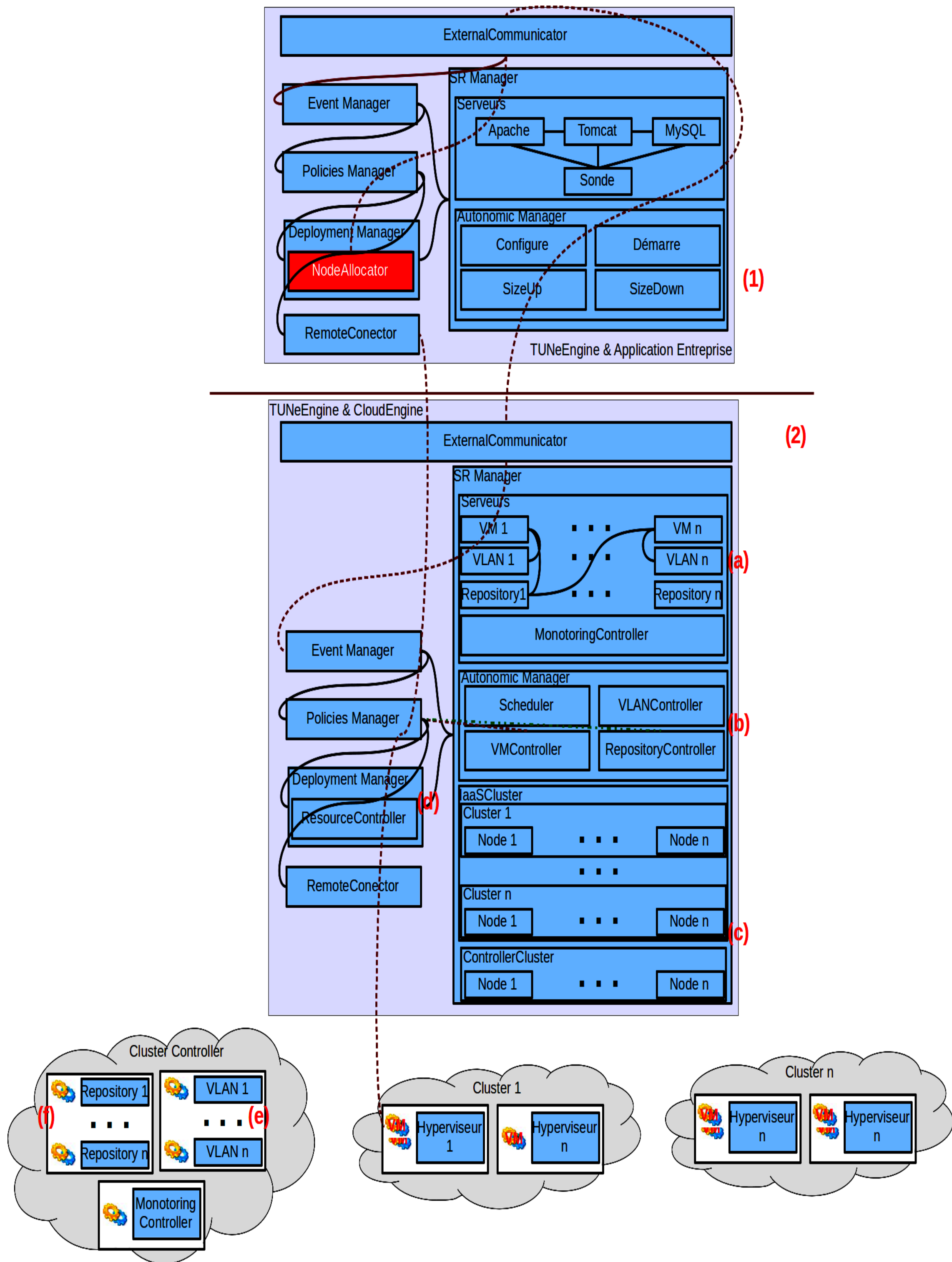


FIGURE 9.2 – Vision synthétique : intégration de CloudEngine à TUNeEngine et utilisation de TUNeEngine pour le déploiement d'applications entreprises dans le cloud.

réseaux. Les requêtes reçues de ces canaux sont de types différents. Les premières viennent des clients extérieurs du cloud. Elles correspondent au téléchargement dans le cloud d'images iso de VM. Dans ce cas, le système de fichiers est transmis bloc par bloc du client vers le *RepositoryController*. Notons que les deux intervenants négocient initialement la taille d'un bloc. A la fin du téléchargement des blocs, le *RepositoryController* reconstitue l'image iso et l'instrumentalise pour faciliter la future configuration réseau des VM qui l'utiliseront. L'instrumentalisation de l'image consiste en l'ajout d'une partition particulière dans la description des partitions disque de l'OS. Cette partition contiendra les programmes d'initialisation réseau des VM qui exécuteront cette image. Cette technique permet à notre plateforme de cloud de prendre en compte des images d'OS standard. Nous revenons sur la description de ces programmes dans la section 9.1.4.

Les deuxièmes requêtes reçues par le *RepositoryController* viennent du service *VMController* lorsque celui-ci est sur le point de démarrer une VM. En effet, comme nous l'avons évoqué dans la section 3, l'exécution d'une VM dans le cloud nécessite la duplication de son image de base (pour une réutilisation ultérieure). Dans notre implantation, c'est le *RepositoryController* qui réalise cette duplication. L'image dupliquée est ensuite transférée sur le lieu d'exécution de la VM (élément *ExecutedVMRepository* dans la figure 9.1). Ce lieu peut être la machine d'exécution de la VM ou encore un serveur de partage de fichiers accessible par la machine d'exécution de la VM. Dans notre cas, il s'agit d'un serveur NFS accessible par toutes les machines de l'IaaS.

Implantation dans TUNeEngine. Le service *RepositoryController* est implanté dans TUNeEngine par un programme de reconfiguration. Son exécution est déclenchée par une requête d'enregistrement d'images de VM. Pour chaque image, l'exécution du *RepositoryController* crée et démarre dans le SR de TUNeEngine un composant *Repository* représentant l'image. Le déploiement et l'exécution du *Repository* s'effectue sur le serveur de stockage (figure 9.2(2)(f)). C'est lui qui est chargé de recevoir du client externe, le contenu de l'image. Il obtient du *RepositoryController* les paramètres tels que : le répertoire d'enregistrement des images, les paramètres d'écoutes réseaux, la taille des blocs de réception d'images VM, le nom de l'image, la taille de l'image et le type d'OS contenu dans l'image. L'intégration des composants *Repository* dans le SR permet à CloudEngine de faciliter la tâche de création de VM du *VMController*.

Le démarrage d'une VM dans le cloud entraîne la création d'une image secondaire, replica de l'image OS originale. Cette nouvelle image est représentée dans le SR par un composant *ExecutedVMRepository* dont le déploiement et l'exécution duplique réellement l'image originale représentée par un *Repository*.

9.1.3 VMController

Il s'occupe du démarrage, de l'arrêt et de la migration des VM à la demande des clients externes ou du *Scheduler*. Pour le démarrage des VM, il requiert le nom de l'image contenant l'OS à démarrer, le nombre de VM à démarrer et les ressources de la VM (nombre de processeurs, taille mémoire, nombre d'interface réseau, etc). Le procédé qu'il suit pour le démarrage de chaque VM est le suivant :

- Il s'adresse au gestionnaire de ressources (*ResourceController*) pour l'acquisition du lieu d'exécution (SE) de la VM.
- Comme nous l'avons évoqué dans la section précédente, il sollicite le *RepositoryController* afin que celui-ci mette à la disposition du SE l'image de la VM (par la création et l'exécution de l'*ExecutedVMRepository* représentant l'image de la VM).
- En fonction du type de l'image, il obtient du *VLANController* les scripts d'initialisation réseau de la VM. Ces scripts sont enregistrés dans la partition d'initialisation réseau de telle sorte qu'ils s'exécutent au démarrage de la VM.
- Il utilise les API fournies par l'hyperviseur du SE pour démarrer la VM.
- Pour finir, les scripts d'initialisation utilisent l'hyperviseur pour communiquer les paramètres réseaux obtenus par la VM. Il transmettra ces paramètres au client propriétaire de la VM afin que celui-ci puisse accéder à ses VM.

Inversement, l'arrêt d'une VM libère les adresses réseaux qu'elle utilise. Le *VMController* sollicite également le *ResourceController* pour le nettoyage du lieu d'exécution de la VM. Ce nettoyage peut consister à supprimer l'*ExecutedVMRepository* de la VM ou remplacer le *Repository* de la VM par son *ExecutedVMRepository*.

Implantation dans TUNeEngine. Le *VMController* est implanté dans TUNeEngine sous la forme d'un programme d'administration. Il est exécuté d'une part lorsqu'un client souscrit ou met fin à l'exécution d'une VM. Comme nous le verrons dans la section 9.2, les demandes des clients se traduisent en événements de reconfiguration au niveau de l'instance TUNeEngine qui gère l'IaaS. D'autre part, son exécution peut être déclenchée par d'autres services de CloudEngine à l'instar du *Scheduler* (lorsqu'il désir relocaliser ou consolider une VM).

Comme l'enregistrement des images, il associe à chaque VM exécutée dans le cloud un composant dans le SR encapsulant les fonctions de manipulation de la VM. Il s'agit des fonctions de démarrage, d'arrêt et de migration. Ainsi, le déploiement d'une VM consistera au déploiement de son représentant du SR, comme tout autre logiciel, sur la machine fournie par le *ResourceController*. La classe d'encapsulation d'une VM implante également les fonctions de déploiement et de undéploiement telles que requise par l'IaaS. C'est notamment dans cette classe que les requêtes au *VLANController* sont effectuées pour solliciter la configuration de la VM.

9.1.4 VLANController

Il s'occupe des configurations réseaux des VM dans CloudEngine. Pour cela, il gère un fichier d'adresses MAC à allouer aux VM. Son implantation suppose la présence d'un serveur DHCP dans l'environnement du cloud.

Le *VLANController* s'exécute sous forme de démon et attend sur une socket des éventuelles requêtes du composant *VMController* lorsque celui-ci désire démarrer/arrêter une VM. Dans le cas du démarrage, il alloue une adresse MAC inutilisée et reconnue par le serveur DHCP. Il lui fournit également les scripts d'initialisation réseau de la VM en fonction du type d'image qu'elle exécute. Dans son implantation actuelle, le *VLANController* ne gère que des OS de types CentOS [**?**] et Debian [**?**]. Enfin, il configure le réseau de telle sorte que les VM appartenant au même client feront partie d'un unique VLAN.

Par contre, lorsque la requête du *VMController* concerne l'arrêt d'une VM, le *VLANController* libère l'adresse MAC de cette VM de telle sorte qu'elle puisse à nouveau être allouée à une autre VM.

Implantation dans TUNeEngine. Il est implanté dans TUNeEngine comme un programme de reconfiguration. Son exécution, déclenchée par l'administrateur du cloud, permet la création ou la suppression de composants *VLAN* dans le SR. Le rôle de ces composants est la configuration de VLAN dans l'IaaS. Chaque composant VLAN implante une procédure de configuration réseau particulière. Ils fournissent également les paramètres d'initialisation réseau des VM aux composants VM les représentants. Il s'agit des informations sur les serveurs DNS, DHCP et les adresses MAC reconnues dans le réseau. Comme les *Repository*, le déploiement et l'exécution des composants *VLAN* s'effectuent sur une unique machine de l'IaaS (figure 9.2(2)(e)). C'est à partir de cette machine qu'ils réalisent la configuration des VLAN et des VM s'exécutant dans l'IaaS.

9.1.5 ResourceController

Il est le responsable de l'allocation des machines d'exécution des VM (les machines hébergeant un hyperviseur) dans le cloud. Il intervient sur la demande du *VMController* afin de lui fournir la machine d'exécution la plus adéquate à l'exécution d'une VM. Pour cela, le *VMController* lui fournit les besoins en ressources de la VM qu'il souhaite exécuter. En fonction de ces informations et de celles fournies par le *MonitoringController*, il choisit la machine d'exécution de la VM.

Dans son implantation actuelle, le choix de la machine d'exécution s'appuie sur deux algorithmes dit de placement : éclatement et regroupement des VM. Dans le premier algorithme, le *ResourceController* choisit la machine d'exécution la moins utilisée. Dans ce cas, les machines n'hébergeant aucune VM sont privilégiées. Dans le second algorithme, il choisit la machine la plus utilisée pouvant exécuter la nouvelle VM sans s'effondrer. Les critères d'effondrement d'une machine sont configurés par

l'administrateur du cloud avant son exécution. Ils prennent en compte les capacités mémoire, processeur, réseau et disque de la machine.

Implantation dans TUNeEngine. Le *ResourceController* (figure 9.2(2)(d)) correspond dans TUNeEngine au ***NodeAllocator***. Grâce à l'adaptabilité de TUNeEngine, nous remplaçons son ***NodeAllocator*** par le *ResourceController*. Comme nous l'avons présentée dans la section 8.3.4, son exécution est déclenchée par le *VM-Controller* via le ***Deployer*** de TUNeEngine.

Nous regroupons les ressources de l'IaaS en deux catégories (figure 9.2(2)(c)) : les machines d'exécution des composants services de l'IaaS (*VLAN*, *Repository* et *MonitoringController*) et les ressources d'exécution des VM. Chaque ensemble de ressources est représenté par un cluster : *ControllerCluster* et *IaaSCluster*. Le *ResourceController* gère les ressources de l'*IaaSCluster*. Les composants Fractal VM sont configurés dans TUNeEngine de telle sorte que leur exécution se fasse dans l'*IaaSCluster*. De cette façon, le *ResourceController* sera contacter pour le choix du SE parmi les machines de ce cluster.

9.1.6 MonitoringController

Il gère le monitoring de l'IaaSCluster. Pour cela, il installe sur toutes les machines de l'IaaS des serveurs qui observent les performances de la machine et de ses VM. Ces serveurs communiquent leurs informations de monitoring à un serveur centralisé (qui est le véritable *MonitoringController*) qui s'exécute sur le *frontend*. Ces serveurs recherchent deux types d'informations : l'état des VM ; et les consommations de chaque VM et de la machine entière.

En ce qui concerne les états de VM, nous considérons qu'une VM peut être dans deux états : soit accessible soit inaccessible. Une VM est considérée comme accessible lorsqu'elle répond correctement à au moins une requête réseau de type *ping*. Quant aux informations de consommation, nous prenons en compte les consommations réseau, CPU et mémoire de chaque VM et de toute la machine.

Pour obtenir ces informations, le *Monitoringcontroller* et ses serveurs utilisent des API de l'hyperviseur et de ceux du système d'exploitation. Ces informations permettront par exemple au *ResourceController* de déterminer le niveau d'effondrement d'une machine d'exécution de VM ou encore au *MonitoringController* de déclarer la panne d'une VM.

Implantation dans TUNeEngine. Un composant Fractal encapsule le service *Monitoringcontroller* qui s'exécute sur le frondent. Quant aux serveurs de monitoring, ils s'agit de serveurs associés (par encapsulation) à chaque nœud (ce choix de conception peut être remise en cause). En effet, comme tout élément administré par TUNeEngine, nous définissons dans la classe Java d'encapsulation de tout nœud de l'*IaaSCluster* des programmes jouant les rôles suivants :

– rôle de sonde : il enregistre dans un fichier les consommations en ressources des VM qu'héberge le nœud. Ces informations seront par la suite transférées vers le

MonitoringController (sur demande de l'administrateur) ou le *Scheduler* (sur ça demande).
– role de détecteur (et réparation) de pannes : il initie dans certain cas, la réparation des VM en cas de panne détectée. C'est le cas lorsque le cloud pratique une réparation par checkpointing tel que nous le verrons dans la section 9.3.1.

9.1.7 Scheduler

Le *Scheduler* gère la réorganisation/consolidation des VM dans l'IaaS pendant leur exécution. Cette consolidation permet notamment aux ressources de l'IaaS d'être mieux utilisées. Son implantation dépend des politiques que souhaitent l'administrateur. Dans le chapitre 10 de présentation des expérimentations de nos systèmes TUNeEgine et CloudEngine, nous présenterons en détails plusieurs politiques de consolidation. Qu'à cela ne tienne, le *Scheduler* s'appuie sur les informations du *MonitoringController* afin d'établir une cartographie de l'utilisation des ressources.

Son exécution n'est pas continue durant tout le cycle de vie du cloud. Il survient régulièrement (après un temps de repos ou un changement de l'IaaS par exemple) afin de déterminer les consolidations à effectuer dans l'IaaS.

Implantation dans TUNeEngine. Comme le *VMController*, le *Scheduler* est implanté par un programme de réconfiguration dans TUNeEngine. Il est déclenché au démarrage du cloud et ne s'interrompe qu'à l'arrêt de TUNeEngine. Le programme de réconfiguration qui l'encapsule comprend un seul état exécutant une classe java implantant la politique de scheduling. Sa définition sous forme de programme de réconfiguration lui permet d'avoir accès au SR (donc de la liste des machines et des VM). A l'aide des informations du SR et des appels réguliers au *MonitoringController*, il constitue à son réveil la cartographie d'utilisation des ressources. Cette cartographie lui permet de décider de la liste des machines virtuelles à déplacer.

9.2 Utilisation de CloudEngine

Après sa mise en route par TUNeEngine, notre plateforme de cloud est prête à accueillir des applications clientes venues de l'extérieur. Comme nous l'avons annoncé en préambule de ce chapitre, nous proposons également l'adaptation du système TUNeEngine pour faciliter l'administration des applications clientes dans CloudEngine. Cette section est consacrée à la présentation de cette adaptation. Nous débutons par la présentation de l'utilisation de TUNeEngine pour l'accomplissement de la première opération d'externalisation dans le cloud à savoir la construction et le téléchargement d'images de VM (section 3 du chapitre 2). Ensuite, nous présentons l'adaptation de TUNeEngine pour le déploiement d'applications CloudEngine. A l'exception du déploiement proprement dit des application dans le cloud, nous ne présentons pas en détails l'utilisation de TUNeEngine pour la configuration et

le démarrage des applications. En effet, ces utilisations sont similaires à celles du système TUNe dont l'implantation de base de TUNeEngine reprend.

La figure 9.2 montre les deux niveaux d'utilisation de TUNeEngine pour l'administration dans le cloud. Une instance de TUNeEngine (figure 9.2(2)) sert à l'administration de CloudEngine et plusieurs autres instances sont utilisées pour l'administration des applications entreprises s'exécutant dans le cloud (figure 9.2(1)). Ces dernières instances collaborent avec la première pour remplir leurs objectifs.

9.2.1 Construction et enregistrement de VM

A l'aide de TUNeEngine, nous proposons un système d'automatisation du processus de construction et de téléchargement vers le cloud d'une image VM. Dans le reste de ce document, nous nommons *ImageConstructor*, l'ensemble constitué de TUNeEngine et cette application. Dans son implantation actuelle, *ImageConstructor* se limite à l'automatisation de l'installation des OS de type Linux. Pour cela, il s'appuie sur les outils systèmes fournis par un système Linux (Kickstart [**?**], Debootstrap [**?**], etc). En conséquence, il requiert pour son exécution que le client de CloudEngine dispose d'un système Linux.

Sans rentrer dans les détails, l'*ImageConstructor* crée dans le SR un composant Fractal pour chaque image à construire. Ce composant définit les paramètres d'installation de l'OS (distribution Linux, partitionnement, package, utilisateur, mot de passe, etc). Son exécution réalisera la construction programment dite le l'image. L'*ImageConstructor* définit également les paramètres permettant de contacter l'instance TUNeEngine du cloud (pour la suite, nous dirons tout simplement CloudEngine). Il s'agit du nom de la référence de CloudEngine dans l'annuaire ainsi que l'adresse de cette annuaire.

Après la construction de l'image, son enregistrement dans le cloud est géré par l'exécution du composant associé à l'image dans le SR. Ce composant contacte le cloud (via les informations obtenues de l'*ImageConstructor*) afin de réaliser le téléchargement de l'image dans le Cloud. Ce contact se traduit au sein de CloudEngine par l'exécution du *RepositoryController*. Après négociation de la taille des blocs d'échange avec le *RepositoryController*, le composant représentant l'image OS dans l'instance TUNeEngine de niveau entreprise transmet le contenu de l'image bloc par bloc vers le composant *Repository* le représentant au niveau de CloudEngine.

9.2.2 Administration d'applications

Comme nous l'avons évoqué ci-dessus, nous ne revenons par en détails sur le procédé d'utilisation de TUNeEngine pour la réalisation des tâches de configuration et démarrage des applications déployées dans le cloud. A l'exception du déploiement que nous présentons ci-dessous, l'administration d'une application par TUNeEngine

(au niveau applications entreprises) est identique à celle des services de CloudEngine présentée dans la section 9.1.

La description de l'application à administrer avec TUNeEngine dans le cloud requiert l'implantation d'un ***NodeAllocator*** particulier (figure 9.2(1)). En effet, les machines d'exécution des applications ne sont ni existantes, ni connues initialement par le client. De la même façon que nous définissions un composant *FrontEnd* représentant le cloud dans la construction des images, nous définissons ici un composant jouant le même rôle. Ce composant permet au ***NodeAllocator*** de souscrire aux VM dans le cloud. Cette souscription entraînera dans le cloud, l'exécution du *VMController*. Ce dernier retournera par la suite au ***NodeAllocator*** du client, les adresses de connexion aux VM lui appartenant. C'est également pendant cette souscription que le client négocie avec CloudEngine des modalités de gestion de ses VM.

En guise d'exemple, CloudEngine permet au client de négocier le moyen de réparation des VM en cas de pannes. Dans les sections suivantes, nous présentons les différentes méthodes de réparations pouvant être implantées par les deux parties. Plus généralement, nous présentons des politiques d'administration (reconfiguration) à la fois dans CloudEngine et ses applications entreprises.

9.3 Reconfiguration d'applications et de CloudEngine

Dans cette section, nous présentons l'implantation des principales opérations de reconfiguration que nous avions présentées dans la section 3 du chapitre 2.

9.3.1 Réparation de VM

Nous identifions deux méthodes de réparation de VM dans le cloud : réparation non collaborative et réparation collaborative. Avant de présenter ces deux méthodes, nous étudions deux moyens de détection de pannes de VM dans le cloud.

Dans l'implantation de CloudEngine, nous avons décrit la première méthode de détection de pannes. Elle est réalisée par les serveurs du *MonitoringController* de l'IaaS (section 9.1.6). Cependant, un client n'est pas tenu à souscrire à ce service de surveillance. Dans ce cas, il insérera parmi ses logiciels à déployer dans le cloud, des logiciels particuliers jouant le rôles de surveillance. De plus, le client doit organiser le déploiement de ces logiciels de telle sorte que l'observateur d'une VM ne s'exécute pas sur cette VM. Quelque soit le moyen de détection utilisé, nous distinguons deux méthodes de réparation de VM. Partant de la détection par IaaS, présentons ces deux méthodes de réparation.

Réparation non collaborative

Dans ce type de réparation, l'IaaS est le seul à s'occuper de la réparation des VM. Les serveurs de monitoring de CloudEngine (ceux associés aux représentants des machines dans le SR) enregistrent régulièrement l'état entier des VM qu'ils monitorent (contenu des mémoires, disque, communications réseau, etc.). Seuls les deux derniers états sont maintenus. Pour cela, ils se servent de la fonctionnalité de checkpointing que fournissent les hyperviseurs de l'IaaS (voir la section 2.2.3 du chapitre 2). Ainsi, lorsqu'une panne de VM est décelée, l'exécution d'un programme de réparation est déclenchée par le *MonitoringController* ou l'instance . Ce programme démarre (via le *VMController*) une nouvelle VM ayant non seulement les mêmes caractéristiques que la VM en panne mais aussi le dernier état correcte connu de cette VM.

Comme nous le verrons dans le chapitre suivant, l'inconvénient de cette méthode de réparation est la dégradation de l'exécution des applications clientes dans le cloud. En effet, le checkpointing entraîne l'arrêt momentané de la VM, ce qui ralentit l'exécution de ses applications.

Réparation collaborative

La seconde méthode évite le checkpointing. En cas de panne, un programme de réparation de CloudEngine démarre une nouvelle VM en remplacement de celle tombée en panne. Cette VM s'appuie uniquement sur l'image de celle en panne. Cependant, les logiciels clients précédemment en cours d'exécution dans la VM ne sont pas démarrés par ce programme. En effet, le cloud ne dispose d'aucune connaissance des logiciels s'exécutant dans les VM qu'il héberge.

Le démarrage des logiciels sera donc réalisé par l'instance TUNeEngine du client. La communication CloudEngine vers l'instance TUNeEngine du client se fait de la même façon que du client vers le cloud. En effet, le *ResourceAllocator* du client fournit au cloud pendant la souscription aux VM, les références de contact de l'instance TUNeEngine du client. Il s'agit de l'implantation d'une sorte de callback dans CloudEngine. Ainsi, le programme de réparation de VM au niveau du cloud pourra faire appel au programme de réparation de VM au niveau du client. Ce dernier programme exécutera le processus d'installation et de démarrage des logiciels.

9.3.2 Consolidation de VM et Scalabilité de serveurs

Après l'implantation des programmes de réparation dans CloudEngine, nous nous intéressons dans cette section à l'implantation des autres tâches de reconfiguration dans le cloud et ses applications. Au niveau de CloudEngine, il s'agit des opérations de consolidation que met en place l'administrateur pour optimiser la gestion des ressources. Quant aux applications clientes, compte tenu de la multitude des opérations possibles, nous nous limitons à celles ayant un impact sur le cloud. Il s'agit des reconfigurations qui entraînent l'ajout ou le retrait de VM pendant l'exécution des applications clientes. Lorsqu'elles surviennent dans la même application, plusieurs appellations sont utilisées dans la littérature pour désigner ces deux opérations : sizing, scalabilité ou encore passage à l'échelle.

Dans cette section, nous présentons plusieurs situations d'exécution de CloudEngine et des applications clientes pour lesquelles nous proposons différents programmes de reconfiguration. Pour cela, nous supposerons que les clients du cloud exécutent des applications de type J2EE.

9.3.2.1 Exécution d'applications statique

La situation la plus basique est celle dans laquelle les clients du cloud surdimensionne à chaque réservation le nombre de VM dont ils ont besoins pour l'exécution de leurs applications. De plus, nous supposons que toutes les VM d'un client sont démarrées ou arrêtées au même instant (nous considérons le cas inverse comme du sizing, section suivante). Dans cette situation, le cloud à intérêt à mettre en place un système de consolidation basé sur les consommations réelles des VM. En effet, étant donné que le client a sur-dimensionné ses allocations en termes de VM, celles-ci seront souvent sous utilisées. Si le Cloud réserve les machines en fonction des tailles des VM, tel que prévu par le contrat qui le lie au client, elles seront éclatées sur un grand nombre de machines. L'application d'un programme de consolidation basé sur le niveau d'utilisation effectif des VM permettra au cloud de réduire le nombre de machines allouées aux VM. En effet, ce programme les consolidera sur un nombre restreint de machines en cas de sous utilisation et les éclatera sur plus de machines lorsque la consommation s'accentuera.

9.3.2.2 Exécution d'applications variables

Cette seconde situation concerne les clients avertis. En effet, nous supposons ici que les clients pratiquent du sizing dans leurs applications. Contrairement à la situation précédente, ici le nombre de VM du même client est variable. L'IaaS profite de ces arrêts de VM, donc libération de ressources, pour réorganiser les VM par consolidation. Pour illustrer cela, prenons quelques exemples précis :

- Si le cloud exécute une seule application et que celle-ci pratique du sizing ordonné, alors aucune opération de consolidation n'est nécessaire. Nous parlons de sizing ordonné lorsque les VM retirées sont les dernières à être ajoutées.
- Si le cloud exécute une seule application pratiquant un sizing non ordonné, alors la consolidation est nécessaire. Cette situation est identique à celle dans laquelle le cloud exécute plusieurs applications appartenant à un ou plusieurs clients. En effet, la fin d'une application (qui entraîne la fin de ses VM), pendant l'exécution d'autres, laissera des trous sur les machines hébergeant les VM de cette application. Cette libération de ressources correspond celle que nous observons dans une application pratiquant un sizing non ordonné.
- Une situation plus fine est celle dans laquelle le cloud donne la possibilité au client de faire varier la taille d'une VM pendant son exécution. Ainsi, avant d'ajouter une nouvelle VM pour absorber une charge, le client pourra tout d'abord augmenter progressivement les ressources de la VM jusqu'à atteindre

la ressource maximale (celle de la machine qui l'héberge). Et, c'est à l'issu de ces augmentations qu'il ajoutera concrètement une nouvelle VM. Le même raisonnement est valable pour le retrait. Dans ce scénario, la consolidation au niveau de l'IaaS prend véritablement tout son sens. Les variations de tailles des VM, libéreront des ressources (avec apparition des "trous") et nécessiteront ainsi des consolidations/éclatements de VM dans l'IaaS. Notons également qu'ici, le client est dans l'obligation d'adapter également les distributeurs de charges de ses applications.

Nous évaluons dans le chapitre suivant toutes ces scénarios de reconfiguration à la fois dans le cloud et dans les applications clientes.

9.4 Synthèse

Dans ce chapitre, nous avons présenté une adaptation de TUNeEngine pour l'administration de la plateforme de cloud simplifiée CloudEngine ainsi que des applications qu'elle héberge. La figure 9.3 présente une vue globale de ces adaptations de TUNeEngine : à gauche de la figure, nous avons l'utilisation de TUNeEngine pour l'administration d'une application entreprise J2EE s'exécutant dans le cloud ; à droite nous avons d'une part l'utilisation de TUNeEngine pour l'administration de CloudEngine et d'autre part le contenu des machines de l'IaaS. Par soucis de lisibilité, toutes les liaisons Fractal entre les composants ne sont pas présentées dans cette figure.

Le système de représentation de l'instance TUNeEngine du cloud est organisé en quatre composants :

- Un composant ("IaaS" sur la figure) contenant les *wrappers* des serveurs de l'IaaS ainsi que ceux des futures images et VM qu'hébergera le cloud. Remarquons que dans cet exemple, le cloud propose une image de VM par défaut (*DefaultImage*) dont la description est montrée dans la figure 9.4(a). Plusieurs configurations de VLAN peuvent être utilisées dans l'IaaS. La figure 9.4(b) présente un exemple de configuration d'un VLAN. Cette description contient par exemple l'adresse IP réseau qu'utiliseront les VM de ce VLAN.
- Un composant ("Ressources" sur la figure) contenant les *wrappers* des machines de l'IaaS. Ces machines peuvent être regroupées en cluster.
- Un composant ("IaaS AutonomicManager" sur le figure) contenant les programmes d'administration et de reconfiguration des serveurs de l'IaaS. Ce composant est relié au composant "IaaS" de la figure. On retrouve par exemple le ***VMController*** que nous avons présenté dans les sections précédentes comme un programme d'administration.
- Un composant ("Ressources AutonomicManager" sur le figure) contenant les programmes de démarrage des logiciels de monitoring (sonde) des machines de l'IaaS. Ce composant est relié au composant "Ressources" de la figure. Nous associons une sonde par machine (voir le logiciel "Sonde" sur la partie droite basse de la figure 9.4)

Quant au système de représentation de l'instance TUNeEngine d'administration de l'application entreprise J2EE, il est organisé en deux composants :
- Un composant contenant les *wrappers* des serveurs J2EE et un *wrapper* décrivant les informations d'initiation de la collaboration avec le cloud. Chaque *wrapper* de serveur J2EE contient d'une part les informations de configuration du serveur proprement dit et d'autre part les informations décrivant la quantité de ressources VM de ce serveur. La figure 9.4(c) montre la description du *wrapper* d'un serveur Apache de l'application entreprise J2EE. Dans sa première partie, nous retrouvons les informations de configuration d'Apache comme le "ApacheUser" tandis que dans sa deuxième partie, nous retrouvons les informations de configuration de la VM qui hébergera cet Apache (par exemple "cpu" : la quantité de processeur à garantir à cette VM).
- Un composant contenant les programmes de configuration, démarrage, reconfiguration et réparation des serveurs J2EE.

Le composant ***ResourceController*** (remplaçant le ***NodeAllocator***) de l'instance TUNeEngine de niveau entreprise obtient du serveur RMI de collaboration, la référence RMI du composant ***Collaborator*** du Cloud. A partir de cette référence, le ***ResourceController*** pourra allouer ou libérer des VM pour l'exécution des serveurs de l'application entreprise. Dans cette collaboration, la plus part des ordres émis par le ***ResourceController*** à destination du cloud seront résolus par le ***VMController***. En réponse à la réservation d'une VM pour l'exécution d'un serveur Apache par exemple, le ***VMController*** construit et insère dans le système de représentation de CloudEngine un *wrapper* de VM dont une description est présentée dans la figure 9.4(d). Les informations de configuration de cette VM sont transmises par le ***ResourceController*** du niveau entreprise dans l'ordre de réservation.

Dans la même logique, l'implantation du composant ProbreMySQL utilise la référence RMI du ***Collaborator*** afin d'avoir les informations de monitoring des VM hébergeant les serveurs MySQL dont il observe. Ainsi, il pourra (à partir de ces informations) décider de la scalabilité ou non des serveurs MySQL en cas de nécessité.

Pour finir, nous pouvons observer un empilement de serveurs sur les machines de *VMCluster*. Ainsi, s'exécutent au dessus de l'hyperviseur les représentants *RemoteLauncher* et *RemoteWrapper* déployés par l'instance TUNeEngine du cloud. Chaque *RemoteWrapper* est associé à l'exécution d'une VM, qui est considérée par l'instance TUNeEngine du cloud comme un élément administrable. Ensuite, chaque VM contient les représentants *RemoteLauncher* et *RemoteWrapper* déployés par l'instance TUNeEngine de niveau entreprise. Chaque *RemoteWrapper* de chaque VM est associé à l'exécution d'un serveur J2EE. Les différents représentants des instances TUNeEngine des deux niveaux (entreprise et cloud) sont entièrement indépendantes et ne disposent d'aucun lien entre eux.

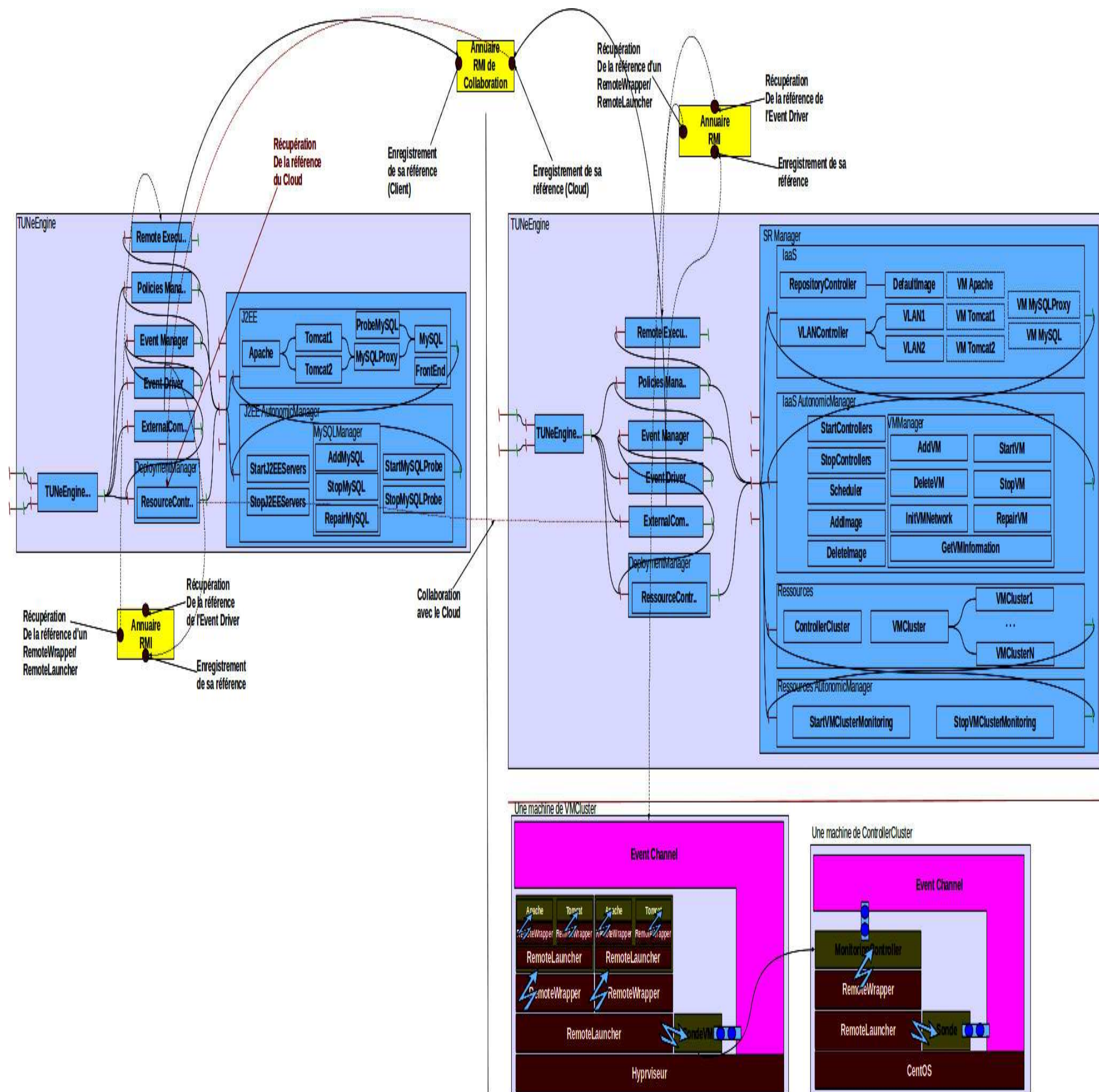


FIGURE 9.3 – Vue globale d'administration et d'utilisation de CloudEngine via TUNeEngine

(a)

```xml
<component name="DefaultImage" definition="fractalADL.LegacyComponent">
 <attributes signature="core.interfaces.ComponentAttributes">
   <attribute name="rootFS" value="xvd"/>
   <attribute name="imageType" value="centos"/>
   <attribute name="taille" value="2Go"/>
 </attributes>
</component>
```

(b)

```xml
<component name="VLAN1" definition="fractalADL.LegacyComponent">
  <attributes signature="core.interfaces.ComponentAttributes">
    <attribute name="gateway" value="192.168.0.254"/>
    <attribute name="netMask" value="255.255.0.0"/>
    <attribute name="netAdress" value="192.168.0.0"/>
    <attribute name="dns" value="192.168.0.254"/>
    <attribute name="dnsSearch" value="cluster-astre-tune.fr"/>
    <attribute name="bridge" value="xenbr0"/>
    <attribute name="initNetworkScript" value="initNetwork.sh"/>
  </attributes>
</component>
```

(c)

```xml
<component name="Apache" definition="fractalADL.LegacyComponent">
 <attributes signature="core.interfaces.ComponentAttributes">
   <attribute name="ApacheUser" value="ucluster"/>
   <attribute name="Group" value="ucluster"/>
   <attribute name="Listen" value="8080"/>
   <attribute name="RubisServlets" value="rubis_servlets"/>

   <attribute name="os" value="DefaultImage"/>
     <attribute name="memory" value="512"/>
     <attribute name="cpu" value="1"/>
 </attributes>
</component>
```

(d)

```xml
<component name="VMApache" definition="fractalADL.LegacyComponent">
  <attributes signature="core.interfaces.ComponentAttributes">
    <attribute name="memory" value="512"/>
    <attribute name="vcpus" value="1"/>
    <attribute name="image" value="DefaultImage"/>
    <attribute name="vlan" value="VLAN1"/>
    <attribute name="MAC" value="AA:00:00:00:00:10"/>
    <attribute name="IP" value="192.168.0.200"/>
  </attributes>
</component>
```

FIGURE 9.4 – (a) Description d'une image de VM dans le SR de CloudEngine, (b) Description d'une VM dans le SR de CloudEngine, (c) Description du logiciel Apache dans le SR de l'instance TUNeEngine de niveau application Entreprise, et (d) Description d'une VM dans l'IaaS générée par le VMController de CloudEngine.

Chapitre 10

Expérimentations

Contents

10.1 Environnements d'expérimentation 124
10.1.1 L'IaaS 124
10.1.2 Les applications entreprises : RUBIS 125
10.1.3 Les métriques 126
10.2 Évaluations 126
10.2.1 Réparation de VM 126
10.2.1.1 Réparation non collaborative 127
10.2.1.2 Réparation collaborative 128
10.2.2 Scalabilité et Consolidation 130
10.2.2.1 Scalabilité 130
10.2.2.2 Consolidation 132
10.2.2.3 Scalabilité et Consolidation simultanées 134
10.3 Synthèse 143

Dans ce chapitre, nous présentons une évaluation de notre SAA TUNeEngine appliqué à la plateforme de cloud CloudEngine et aux applications qu'elle administre. Compte tenu de la difficulté d'évaluation de certains aspects de l'administration (configuration, déploiement et reconfiguration par exemple), nous nous limitons dans ce chapitre aux aspects suivants : construction d'images de VM et leur enregistrement dans le cloud ; utilisation de TUNeEngine pour la réparation de VM dans le cloud ; et utilisation de TUNeEngine pour la gestion de ressources de l'IaaS et des applications entreprises.

Ce chapitre est organisé de la façon suivante : dans un premier temps, nous décrivons les environnements matériels et logiciels sur lesquels nous nous appuyons pour la réalisation de nos expérimentations. Ensuite, nous présentons les résultats d'évaluation de TUNeEngine pour la réparation de VM dans CloudEngine. Pour finir, nous présentons l'évaluation de l'utilisation de TUNeEngine pour la gestion de ressources de l'IaaS ainsi que celle des applications entreprises.

10.1 Environnements d'expérimentation

10.1.1 L'IaaS

L'environnement matériel d'expérimentation que nous avons mis en place coïncide avec notre modèle simplifié d'IaaS présenté dans la section 9.1 du chapitre précédent. Les ressources matérielles sont organisées en clusters de machines de type OPTIPLEX 755 :

- Un ensemble de machines (CloudEngine) hébergeant les différents serveurs de CloudEngine à raison de : une machine hébergeant le serveur de stockage (NFS) des images de VM (*Repository* et *ExecutedVMRepository*), les serveurs de configurations des VLAN, le *MonitoringController* ; et une machine hébergeant le *Scheduler*, le *VMController*, le *ResourceController*, autres programmes de reconfiguration de CloudEngine et le registre RMI de sauvegarde des références des instance de TUNeEngine. Cette dernière machine est le point d'entrée de CloudEngine car c'est elle qui héberge également l'instance TUNeEngine chargée de l'administration de CloudEngine. Notons que cette organisation de serveurs peut être décentralisée, par adaptation de TUNeEngine, pour un environnement large échelle. Les machines de ce cluster disposent chacune de 4Go de mémoire, de 2 processeurs de type Intel Core 2 Duo 2.66GHz et utilisent des distributions CentOS 5.6 du système Linux.
- Un cluster d'hébergement des VM (5 machines). Les machines de ce cluster sont celles qui seront utilisées pour l'exécution des VM réservées par les entreprises. Toutes les politiques de gestions de ressources dans l'IaaS s'appliquent uniquement à ce cluster. Toutes ses machines exécutent l'hyperviseur Xen dans sa version 4.1 sous un système Linux CentOS 5.6. Elles disposent de 4Go de mémoire, d'un processeur de type Intel Core 2 Duo 2.66GHz (un cœur désactivé). Les configurations liées à l'hyperviseur Xen sont les suivantes : 512Mo de mémoire pour le dom0 (le système Linux hôte) ; et l'algorithme de scheduling de VM est *sched-credit* [**?**].
- Un cluster (*ClusterExterne* dans la figure 10.1) constitué de 2 machines réservées aux entreprises et clients des applications s'exécutant dans le cloud. Ces machines sont identiques à celles du premier cluster.

Tous les clusters utilisent le même réseau IP et sont reliés entre eux via un switch 1Gb. La figure 10.1 résume l'organisation de notre environnement. Pour finir, les images de VM construites et utilisées dans nos expérimentations sont de 2Go de taille (influenceront le déploiement des VM) et utilisent une distribution CentOS 5.6 de Linux. Elles sont construites de telle sorte que les machines réservées aux entreprises aient un accès SSH ne requérant aucun mot de passe (afin de faciliter l'accès de l'instance TUNeEngine des entreprises vers leurs VM).

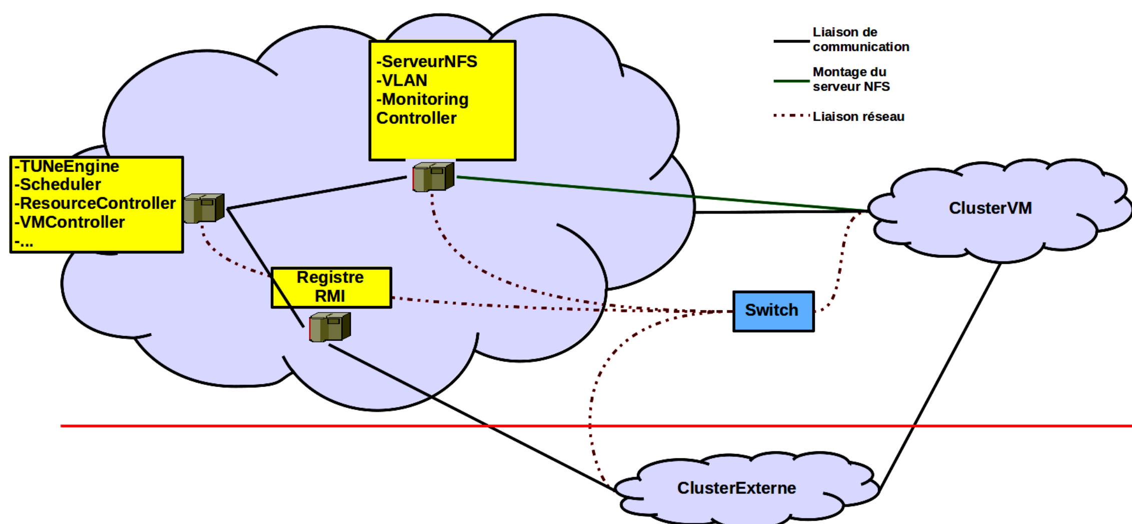


FIGURE 10.1 – Environnement matériel de notre IaaS

10.1.2 Les applications entreprises : RUBIS

RUBIS [?] (Rice University BIdding System) est une implantation d'une application J2EE de commerce électronique. RUBIS est issu du projet JMOB [?] et développé par le consortium OW2 [?]. RUBIS repose sur un serveur web (Apache par exemple), un conteneur de servlet et/ou EJB (Tomcat et JBoss par exemple) et un serveur de base de données (MySQL par exemple). En parallèle, il fournit un simulateur de clients web permettant d'effectuer des benchmarck sur le site web implanté par RUBIS.

RUBIS définit 27 types d'interactions pouvant être réalisées par un client externe. Les interactions les plus importantes sont celles de : consultation des produits par catégories/régions, passer des ordres, acheter des produits, vendre des produits, poser des commentaires ou consulter son profil. La plupart des intéractions (22) nécessitent la construction dynamique des pages.

RUBIS permet de définir deux profils de benchmarking : avec lecture de données uniquement ou avec 15% de lecture-écriture de données. Dans le cadre de nos expérimentations, nous utilisons le premier profile.

Le simulateur de clients web proposé par RUBIS implante un comportement proche de celui d'un client humain d'un site de commerce en ligne. Il permet le démarrage de plusieurs clients web, chacun s'exécutant dans une session de durée précise. A l'aide d'une table de transition définissant le profil de charge, chaque client émet régulièrement une requête à destination du site web RUBIS. Chaque client simule une durée de réflexion entre chaque requête afin de se rapprocher du comportement humain. Cette durée est choisie aléatoirement entre 7 secondes et 15 minutes suivant une distribution exponentielle négative [?]. L'algorithme de passage d'une requête web à une autre est basé sur une matrice de transition contenant des valeurs de probabilités observées dans la réalité. Enfin, chaque client génère

un ensemble de données statistiques récapitulant son exécution. Ces informations contiennent le nombre d'erreurs survenues durant son exécution, les performances des serveurs J2EE, les types de requêtes émises, etc.

10.1.3 Les métriques

Les expériences réalisées durant cette thèse ont pour but de valider l'utilisation du SAA TUNeEngine pour l'administration de la plateforme de cloud CloudEngine et des applications qu'elle héberge. Au niveau de l'IaaS, les principales métriques qui nous intéressent sont : la charge CPU et le nombre de VM par machine physique. La charge CPU comprend la consommation CPU de chaque VM, la consommation CPU du dom0 (système hôte avant l'installation de l'hyperviseur) et la charge CPU de la machine physique qui est la somme de toutes les charges CPU des VM qu'elle héberge.

Au niveau des applications entreprises, nous nous intéressons au débit et au temps de réponse des requêtes de l'application RUBIS, observés au niveau du serveur web Apache.

10.2 Évaluations

10.2.1 Réparation de VM

Dans cette expérimentation, nous évaluons les deux types de réparation que nous avons décrites dans la section 9.3.1 du chapitre précédent. Il s'agit de la réparation collaborative et non collaborative dont nous rappellerons brièvement les principes dans les sections suivantes.

De façon générale, dans les expériences que nous réalisons pour l'évaluation de la réparation, les pannes sont détectées par l'instance TUNeEngine de niveau IaaS. Nous démarrons l'IaaS de telle sorte que les serveurs du *MonitoringController*, installés sur chaque machine de l'IaaS, scrutent l'état des VM toutes les 2 secondes. Nous considérons qu'une VM est en panne lorsqu'elle ne répond plus à une requête IP (la commande "*ping*" à partir d'une autre machine) ou encore lorsque son hyperviseur Xen indique un état d'erreur.

Pour chacune des évaluations des deux méthodes de réparation, nous exécutons un benchmark RUBIS pyramidal (montée de charge, charge constante et baisse de charge). Nous simulons une panne sur une VM hébergeant un serveur MySQL de l'application RUBIS exécutée. En outre, nous associons à ces expérimentations une expérience particulière jouant le rôle de repère. En effet, cette expérience sera utilisée pour la comparaison des deux méthodes de réparation. Elle consiste en l'exécution d'un benchmack RUBIS dans lequel aucune panne n'est simulée. La comparaison

des deux méthodes de réparation est effectuée sur la base du débit en requêtes et du nombre d'erreurs observées durant chaque expérience.

Pour finir, l'application RUBIS exécutée dans cette expérience est composée de : 1 serveur Apache, 1 serveur Tomcat, 1 serveur MySQL-Proxy et 1 serveur MySQL. Les VM hébergeant ces serveurs sont configurées de telle sorte qu'elles s'exécuterons chacune sur des machines distinctes. Cette configuration n'a aucune influence particulière sur les résultats de l'expérimentation.

10.2.1.1 Réparation non collaborative

Nous définissons la réparation non collaborative comme celle dans laquelle l'instance TUNeEngine de l'IaaS est l'unique responsable du dépannage des VM. Dans notre expérimentation, TUNeEngine démarre sur chaque machine de l'IaaS un serveur dont le rôle est de sauvegarder toutes les 7 secondes l'état de chaque VM. Le choix de la fréquence de sauvegarde ne doit pas être ni trop petite, au risque de pénaliser l'exécution de la VM (comme nous le verrons), ni grande au risque d'avoir un grand écart entre la VM dépannée et son état avant la panne. Nous avons choisi une durée de 7 secondes en rapport avec la durée minimale d'attente entre chaque émission de requête de client RUBIS. Les états de VM sauvegardés sont stockés, non pas sur le serveur NFS, mais sur la machine hébergeant la VM. Ces sauvegardes seront transférées avec la VM lors des opérations de consolidation.

La figure 10.2 montre la comparaison entre l'expérimentation repère (courbe rouge) et l'expérimentation avec réparation par checkpointing (courbe verte) dans laquelle aucune panne n'a été simulée. Cette figure nous permet uniquement d'observer l'impact du checkpointing de VM sur les performances de l'application RUBIS s'exécutant dans le cloud. Nous observons une baisse de débit de requêtes traitées dans ce type de réparation : courbe rouge au dessus de la courbe verte. Nous évaluons cette baisse de débit à 46% environ. Elle est due à l'arrêt régulier des VM lors du checkpointing dans l'IaaS. En effet, le checkpointing implanté par Xen provoque l'indisponibilité de la VM pendant sa sauvegarde. Ainsi, toutes les requêtes en direction ou à l'intérieur de cette VM s'exécuteront après la sauvegarde. Ceci explique également de grandes fluctuations du débit sur la courbe verte.

La figure 10.3 présente le résultat de l'application de la méthode de réparation de VM par checkpointing dans le cloud, avec simulation de pannes sur la VM du serveur MySQL. L'instant T_p représente l'instant auquel la panne a été provoquée tandis que l'instant T_r marque la fin de la réparation. La durée de réparation dans cette expérience est d'environ 22 secondes. Cette durée comprend le temps écoulé avant la détection de la panne et également la durée de redémarrage de la VM à partir de son dernier état sauvegardé.

Nous constatons également dans cette expérimentation qu'aucune requête n'est perdue à la fois pendant les sauvegardes de VM que pendant la réparation. En effet, pour ce qui est de la sauvegarde, le protocole TCP/IP sur lequel repose notre réseau se charge de réémettre une requête lorsque celle-ci n'a pas abouti. Ainsi, durant

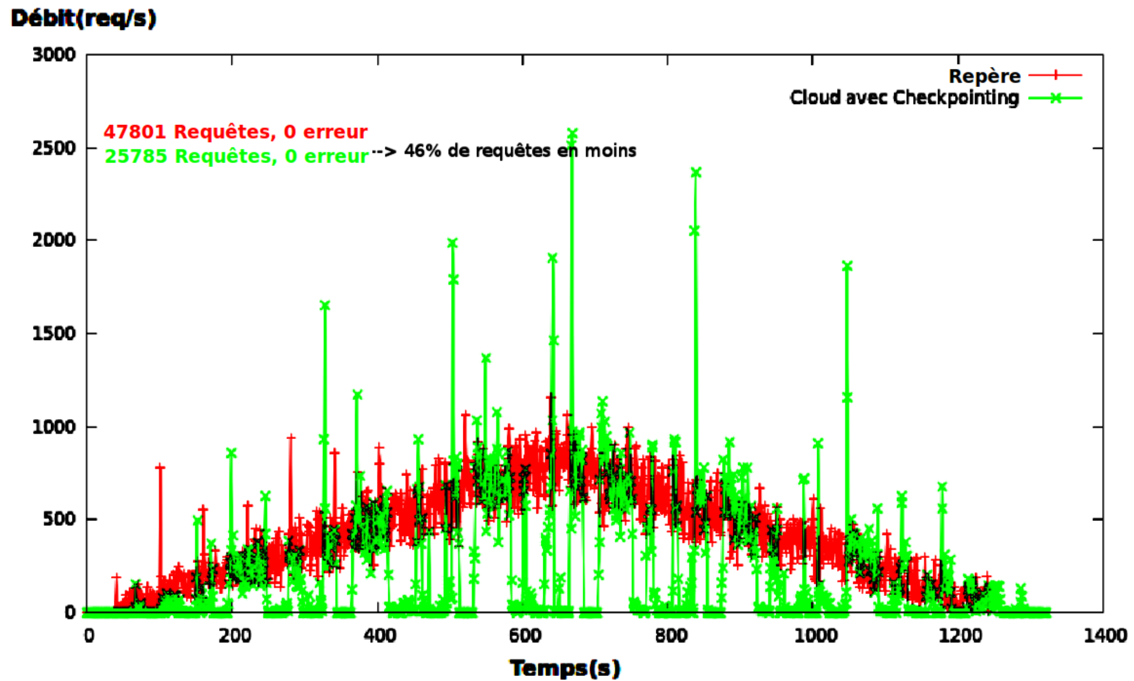


FIGURE 10.2 – Coût du checkpointing dans la réparation de VM dans l'IaaS

la sauvegarde, l'indisponibilité négligeable de la VM n'entraîne pas la rupture des communications TCP/IP et celles-ci sont reprises à la fin de la sauvegarde. Quant à la non perte de requêtes durant la réparation, elle s'explique par le comportement du client RUBIS. En effet, il réémet en cas d'erreur la même requête 6 fois toutes les secondes, ce qui laisse le temps à la réparation (démarrage de la VM avec son ancien état) de se réaliser. Notons que pour certaines applications comme MPI, l'indisponibilité de la VM (même négligeable), suivit du changement d'état de la VM ne saurait être tolérée.

10.2.1.2 Réparation collaborative

La seconde méthode de réparation est dite collaborative dans la mesure où les opérations de répartition sont réalisées à la fois par l'instance TUNeEngine de niveau IaaS et l'instance TUNeEngine de niveau application entreprise. En effet, l'instance TUNeEngine de niveau IaaS détecte la panne de VM, redémarre la VM à partir de son image initiale et fait appel à l'instance TUNeEngine propriétaire de la VM pour finaliser la réparation. L'instance TUNeEngine de niveau application est quant à elle chargée de : déployer sur la VM redémarrée les logiciels qui y s'exécutaient avant la panne ; configurer et démarrer ces logiciels. La taille des logiciels à déployer ainsi que la durée d'exécution des programmes de configuration et de démarrage des logiciels sur la VM influencent la durée de la réparation.

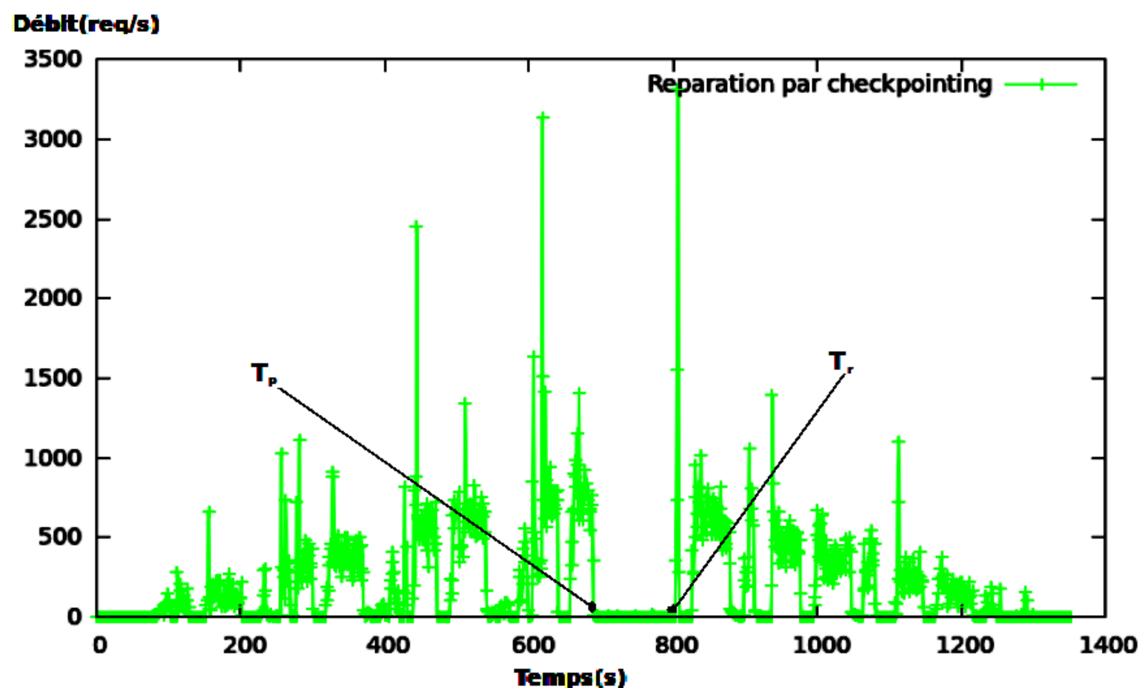


Figure 10.3 – Réparation de VM de l'IaaS par checkpointing

La figure 10.4 présente deux courbes : la courbe rouge représente l'expérimentation repère tandis que la courbe verte représente le résultat de l'expérimentation avec panne et réparation collaborative. Nous constatons que cette méthode de réparation, contrairement à la précédente, n'a aucun impact sur les performances de l'application RUBIS lorsqu'aucune panne n'intervient. Cette constatation est visible sur la figure 10.4 dans les zones (1) et (3). La zone (2) représente la durée de la réparation. Elle inclut : le temps de détection de la panne, la durée de déploiement et de redémarrage de la VM, la durée de la copie des binaires du serveur MySQL s'exécutant sur la VM et enfin la durée de redémarrage du serveur MySQL. Pour ces raisons, nous constatons dans cette expérience, une réparation en 5 minutes 30 secondes. Cette durée importante de la réparation entraîne une perte de requêtes au niveau des clients RUBIS. Cette perte est de deux ordres :

- Des pertes liées aux requêtes effectivement émises par le client RUBIS, mais heurtées à l'indisponibilité du serveur MySQL. Nous évaluons cette perte à environ 0.12% des requêtes exécutées. La valeur négligeable de cette perte vient d'une part du fait que chaque client RUBIS émet 6 fois la même requête en cas d'erreurs, avec une pause d'une seconde entre chaque tentative. D'autre part, les timeout des serveurs J2EE impliqués dans ces requêtes sont supérieurs à la minute.
- En comparaison avec l'expérimentation repère, nous observons la réduction du nombre globale de requêtes exécutées durant cette expérience. Nous estimons cette réduction à environ 23% de requêtes traitées dans l'expérimentation repère. Cette valeur s'explique par la perte de temps des clients RUBIS dans leurs tentatives de réémission de requêtes pendant la durée de la réparation.

Notons que l'utilisation d'un serveur miroir, généralement pratiquée dans les applications, permettra de palier cette latence importante. Dans certains cas, ce miroir est mis en place par l'IaaS. C'est le cas de la plateforme de cloud Windows Azure que nous avons présenté dans la section 7.2.4 du chapitre etatDeLart.

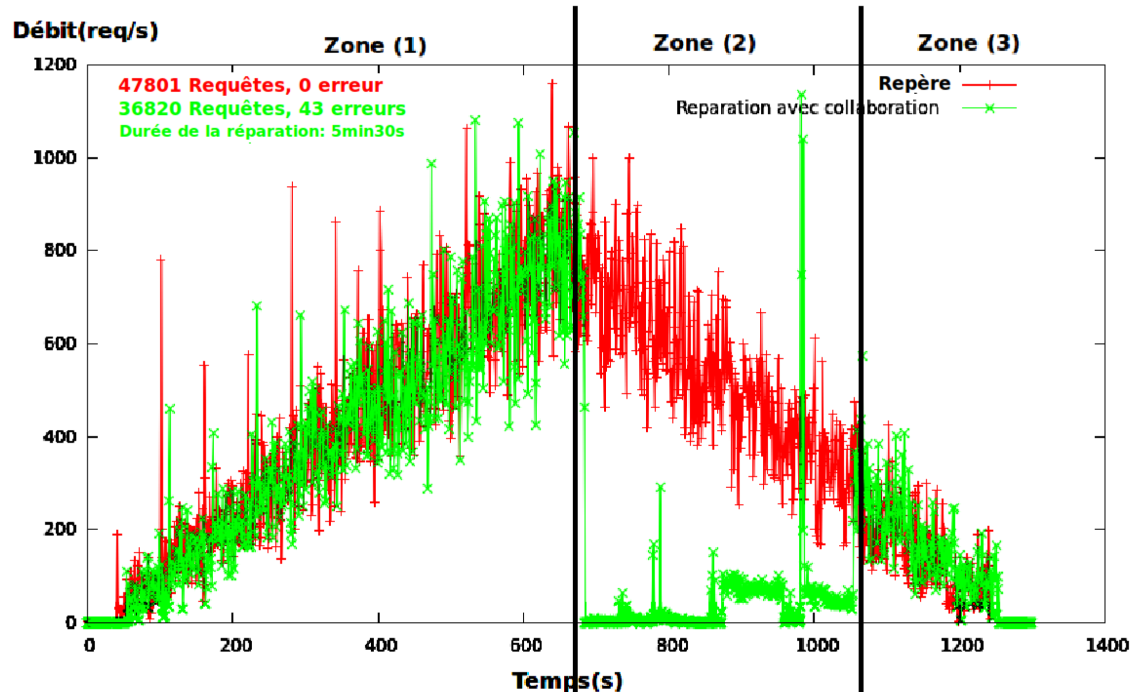


FIGURE 10.4 – Réparation collaborative de VM

10.2.2 Scalabilité et Consolidation

Pour ces évaluations, la figure 10.5 résume les différentes situations que nous expérimentons. Elle montre également pour chaque situation, les métriques sur lesquelles les prises de décision sont basées.

10.2.2.1 Scalabilité

Le but de cette expérimentation est de montrer l'utilisation de TUNeEngine pour l'allocation dynamique de ressources (VM) au niveau de l'application RUBIS dans le cloud. La consolidation au niveau CloudEngine est neutralisée. Son instance TUNeEngine s'occupe uniquement du placement des VM pendant leur démarrage.

L'application RUBIS ainsi que la sonde de monitoring des serveurs MySQL sont configurées de la façon suivante :

- 1 serveur Apache relié à 2 serveurs Tomcat. Ces derniers sont reliés à 1 serveur MySQL-Proxy qui distribue des requêtes à 1 serveur MySQL (initialement).

VM de taille fixe, Application de taille variable	**==> Scalabilité ordonnée ==>basée sur la consommation CPU des VM**
VM de taille fixe, Pas de scalabilité,	**==> Sous utilisation des VM ==> Consolidation ==>basée sur la consommation CPU des VM**
VM de taille fixe, Scalabilité non ordonnée	**==> Utilisation efficace des VM ==> Libération de ressources ==> Trou dans l'IaaS ==> Nécessite la Consolidation pour tasser les trous ==> basée sur la taille des VM (quotas CPU-mémoire)**
VM de taille variable, Scalabilité ordonnée	**==> Utilisation efficace des VM ==> L'augmentation de la taille des VM ==> Trou dans l'IaaS ==> Nécessite Consolidation pour tasser les trous ==> basée sur les la taille des VM (quotas CPU-mémoire)**

Figure 10.5 – Récapitulatif de quelques situations nécessitant scalabilité et de consolidation dans le cloud et les applications qu'il héberge

- La sonde d'observation du tiers MySQL obtient du *MonitoringController* de l'instance TUNeEngine de l'IaaS les informations sur la charge CPU des VM hébergeant les serveurs MySQL. Elle déclenchera l'ajout d'un nouveau serveur MySQL lorsque la charge CPU du tiers MySQL (moyenne des charges CPU des serveurs MySQL) sera supérieure à 60%. Inversement, elle déclenchera le retrait d'un serveur MySQL lorsque le tiers MySQL aura une charge CPU inférieure à 10%. L'ordre de retrait des serveurs MySQL est l'ordre inverse des ajouts. Autrement dit, le dernier serveur ajouté sera le premier retiré. Le seuil de charge CPU maximum est fixé à 60% parce qu'il correspond, après plusieurs tests, au seuil pour lequel le débit en requêtes traitées par l'application RUBIS ne progresse plus. Cette limitation est due à une saturation de la taille mémoire des VM hébergeant les serveurs MySQL (qui sont sollicités pour des requêtes de grande taille). Cette sonde pourrait être couplée à une sonde mémoire afin d'avoir une prise de décision plus efficace. Quant au seuil minimum de 10%, il est choisi parmi des valeurs possibles tel que le retrait d'un serveur ne provoque pas le dépassement du seuil de 60% des serveurs restants.
- Le nombre de clients RUBIS augmente/diminue de 50 toutes les minutes durant l'expérience.

La figure 10.6 montre les résultats obtenus durant cette expérimentation. La courbe verte montre l'évolution du nombre de client RUBIS soumis à l'application RUBIS. La courbe noire montre l'évolution de la charge CPU du tiers MySQl. Les autres courbes montrent l'évolution du nombre de VM sur chaque machine de l'IaaS. Voici l'interprétation de ces courbes :

- La première partie correspond à la phase de déploiement des VM. Chaque machine de l'IaaS héberge une VM.
- La seconde phase correspond au déploient des binaires et au démarrage du serveur MySQL sur sa VM.
- Le zone (a) correspond à l'ajout d'un nouveau serveur MySQL suivi de la chute de la charge CPU du tiers MySQL. Cette chute est due au redémarrage du serveur MySQL-Proxy afin de prendre en compte le nouveau serveur

MySQL. De plus, nous constatons que la décision d'ajouter un nouveau serveur MySQL n'est pas aussitôt prise lorsque la charge CPU atteint 60%. En effet, la sonde effectue plusieurs prises de charges CPU afin d'avoir confirmation du dépassement du seuil.
- La zone (b) correspond au retrait d'un serveur MySQL après que la sonde ait constaté une baisse de charge confirmée en dessous de 10%.

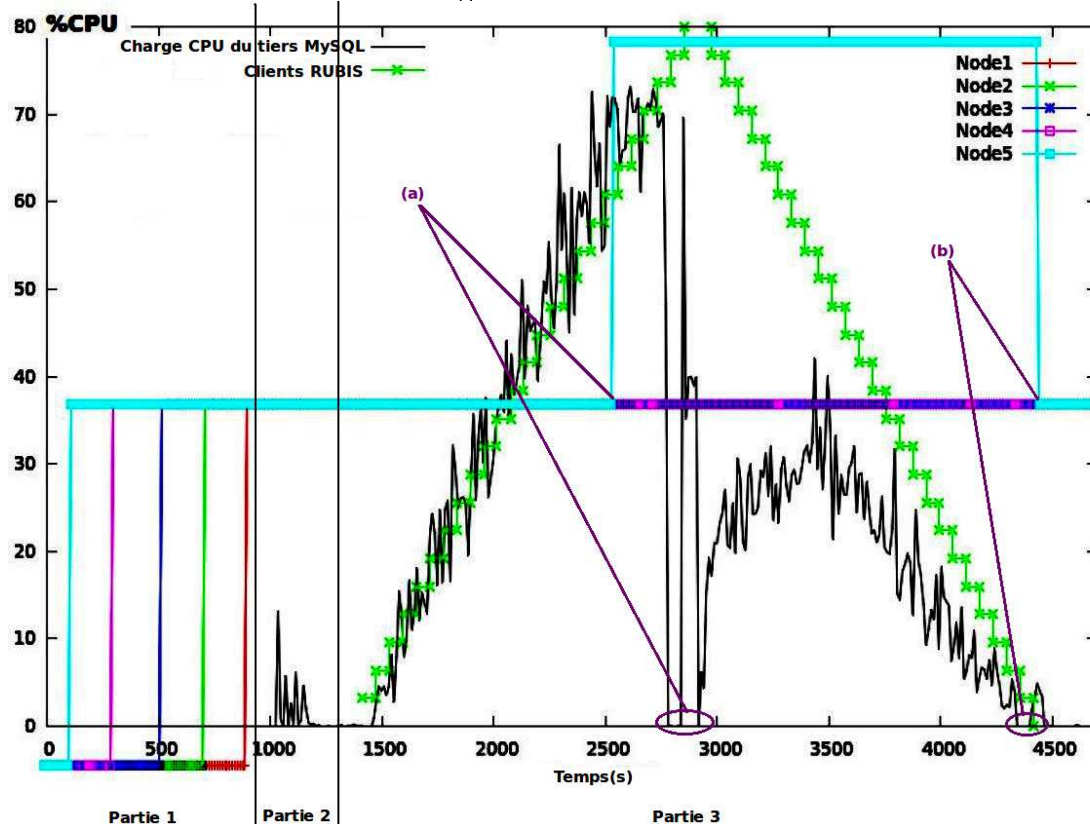


FIGURE 10.6 – Scalabilité de serveurs MySQL dans une application J2EE dans le cloud

10.2.2.2 Consolidation

Après l'évaluation de l'utilisation de TUNeEngine pour l'allocation dynamique de ressources au niveau des applications entreprises dans le cloud (ce qui correspond à ce que faisait le système TUNe, son inspirateur), montrons à présent comment TUNeEngine permet de gérer par consolidation de VM les ressources de l'IaaS. Nous montons pour cette expérimentation un scénario montrant l'utilité de la consolidation dans l'IaaS. Comme nous l'avons énoncé dans la section 9.3.2.1 du chapitre précédent, nous considérons un scénario exécutant au niveau applicatif une seule application. La version TUNeEngine d'administration de cette application n'implante aucune politique d'allocation dynamique de ressources. Quant au niveau IaaS, nous utilisons une politique de placement de VM consistant à maximiser le nombre de VM par machine. Le but de cette expérience est de valider l'existence et l'efficacité

du *Scheduler* de CloudEngine, dont le rôle est de gérer la consolidation des VM dans l'IaaS.

Pour cette expérimentation, l'application RUBIS exécutée dans le cloud ainsi que le *Scheduler* de l'IaaS sont configurés de la façon suivante :

- 1 serveur Apache, 2 serveurs Tomcat, 1 serveur MySQL-Proxy et 2 serveurs MySQL.
- Toute VM hébergent un serveur RUBIS se voit allouer 512Mo de mémoire. Cette configuration fera tenir initialement toutes les VM sur la même machine physique (512Mo * 6 serveurs + 512Mo du dom0 = 3Go512Mo < 4Go, taille d'une machine physique).
- Le *Scheduler* est configuré pour retirer une VM d'une machine physique lorsque la charge CPU de cette machine dépasse 60%. Dans ce cas, la VM la moins chargée est déplacée vers la machine de l'IaaS la moins chargée pouvant l'accueillir. A l'inverse, le *Scheduler* regroupe des VM sur des machines selon la politique suivante. Soit M_a et M_d les machines les moins chargées de l'IaaS contenant des VM telles que M_a est plus chargée que M_d. Soit VM_d, la VM la moins chargée de la machine M_d. Si la charge de M_a additionnée à la charge de VM_d est inférieure à 60% et que la taille mémoire de M_a additionnée à la taille mémoire de VM_d est inférieure à 4Go, alors le *Scheduler* déplace VM_d vers la machine M_a. Cette politique est une parmi tant d'autres. Notons que le but de cette expérience n'est pas de produire et d'évaluer différentes politiques de consolidation. Les travaux réalisés par le système Entropy [**?**] sont entièrement consacrés à cette problématique.

La figure 10.7 montre les résultats de notre expérience : la figure 10.7(a) présente la variation des charges CPU sur les machines de l'IaaS tandis que la figure 10.7(b) présente le contenu de chaque machine de l'IaaS en terme de nombre de VM par machine. Sachant que la taille et le nombre de logiciels déployés dans cette expérience sont fixes, les variations de nombre de VM par machine démontre la réalisation de la consolidation dans l'IaaS coordonnée par le *Scheduler* de CloudEngine. Ces deux figures sont interprétées de la façon suivante :

- La première partie des courbes correspond au déploiement des VM dans l'IaaS. Compte tenu de la politique de placement et de la taille des VM, toutes les VM sont déployées initialement sur la machine Node1. Ceci explique l'observation d'une charge CPU fluctuante sur la courbe CPU de la machine Node1 et l'absence de charge sur les autres machines de la figure 10.7(a). La première partie de la figure 10.7(b) montre également ce regroupement de VM sur la machine Node1.
- Pendant l'exécution du benchmark RUBIS, nous observons un premier éclatement d'une VM de la machine Node1 vers la machine Node4 lorsque la charge CPU de la machine Node1 dépasse le seuil de 60%. Ceci explique la présence d'une charge CPU sur la machine Node4 sur la figure 10.7(a)(1) et la présence d'une VM sur cette machine à la figure 10.7(b)(1). Il en est de même pour le (2) des figures 10.7(a) et 10.7(b).
- Durant l'exécution de ce benchmark, le *Scheduler* peut être amené à regrouper les VM lorsque la charge d'une machine le permet. C'est notamment le cas

de la VM de la machine Node4 qui sera déplacée vers la machine Node5 en mesure de supporter cette VM à l'instant représenté sur la figure 10.7(a)(3).
– Pour finir, les VM sont regroupées à la fin de l'exécution du benchmark sur un nombre restreint de machines. On peut remarquer que toutes les VM ne sont pas regroupées sur une unique machine physique comme initialement. Ceci s'explique par le fait que la migration pratiquée par Xen (le système de virtualisation utilisé par l'IaaS) d'une VM nécessite 2 conditions : (1) la machine de destination dispose d'assez de mémoire pour exécuter la VM à migrer ; (2) la machine de destination dispose également d'un surplus de mémoire destinée à la réalisation de l'opération de VM, consommatrice de mémoire. Cette consommation de mémoire est moins importante pendant la phase de démarrage d'une VM que pendant l'exécution. Ainsi, la machine Node5 hébergeant déjà toutes les autres VM, ne dispose pas assez de mémoire pour accueillir la dernière VM se trouvant sur la machine Node4.

10.2.2.3 Scalabilité et Consolidation simultanées

Dans cette section, nous présentons deux expérimentations dans lesquelles la consolidation au niveau de l'IaaS et l'allocation dynamique de ressources au niveau entreprise sont activées dans les deux instances de TUNeEngine. Nous avons justifié dans la section 9.3.2.2 du chapitre précédent le choix des deux scénarios que nous expérimentons.

Scénario 1 : VM de taille fixe et "Scalabilité non ordonnée"

Rappelons quelques définitions avant la description de l'expérimentation :

La "scalabilité ordonnée" est la scalabilité dans laquelle les premiers serveurs ajoutés sont les derniers serveurs retirés. Dans cette expérience, nous avons choisi d'utiliser la scalabilité non ordonnée au niveau application entreprise. Nous entendons par "scalabilité non ordonnée" ici celle dans laquelle l'ordre de retrait des serveurs est le même que l'ordre d'ajout (les premiers serveurs ajoutés sont les premiers retirés).

Une VM de taille fixe est une VM dont les quantités de ressources initialement acquises par l'entreprise n'évoluent pas durant son cycle de vie.

L'objectif de cette expérimentation est de montrer que l'exécution dans le cloud d'applications pratiquant de la "scalabilité non ordonnée" et utilisant des VM de tailles fixes nécessite à l'implantation des politiques de consolidation de VM au niveau de l'IaaS. Ainsi, l'instance TUNeEngine de niveau entreprise implante la "scalabilité non ordonnée" tandis que le *Scheduler* de CloudEngine est activé dans l'instance TUNeEngine de l'IaaS. Décrivons à présent les configurations de l'IaaS et des serveurs J2EE que nous utilisons dans le cadre de cette expérience.

Au niveau de l'IaaS :
– La politique de placement de VM (lors du démarrage) est le regroupement maximum de VM par machine physique.

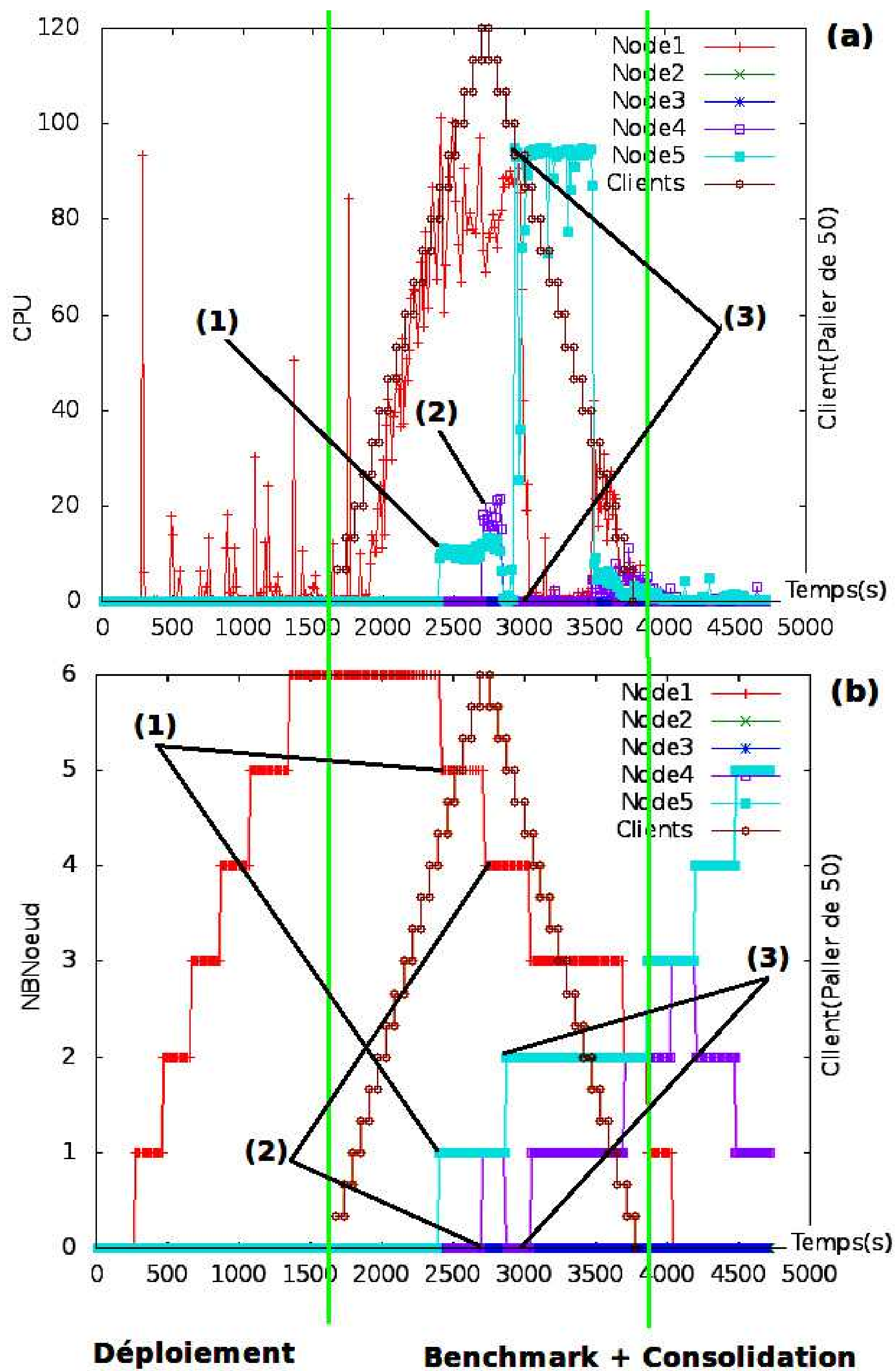


FIGURE 10.7 – Consolidation dans le cloud : (a) Évolution de la charge CPU sur les machines de l'IaaS, (b) Évolution du nombre de VM par machine de l'IaaS.

– Le *Scheduler* de l'IaaS consolide les VM en fonction de leur quota de ressources (CPU ou mémoire). Ici, nous utiliseront le quota mémoire dans la mesure où toutes les VM disposent d'un quota infini de CPU (0 ou 100).

Au niveau application entreprise, nous exécutons une seule application J2EE RUBIS munie d'une sonde associée au tiers MySQL. Les serveurs J2EE de cette application sont configurés de la façon suivante :

– 1 serveur Apache dont la VM utilise 3Go de mémoire ;
– 2 serveurs Tomcat, 1 serveur MySQL-Proxy, 1 serveur $MySQL_1$ dont chaque VM utilise 1.5Go de mémoire ;
– La sonde d'observation du tiers MySQL provoque l'ajout d'un serveur MySQL lorsque la charge CPU moyenne des serveurs MySQL est supérieure à 60%. Inversement, elle provoque le retrait lorsque cette charge est inférieure à 20%. **N.B.** Nous choisissons un seuil minimum de 20% au lieu de 10% comme dans la première expérience pour la raison suivante : un seuil de 10% (qui entraînera une scalabilité presque à la fin du benchmark) dans cette expérience ne laissera pas le temps d'observer l'effet de la consolidation (qui suit le scalabilité) pendant le benchmark. La consolidation n'étant pas évaluée dans la première expérience, ce seuil était acceptable.

Cette configuration entraînera l'instance TUNeEngine de l'IaaS à déployer les VM hébergeant les serveurs RUBIS de la façon suivante (figure 10.8(a)) :

– Compte tenu de la taille mémoire de la VM d'Apache, une machine physique entière (de taille 4Go) sera utilisée pour son hébergement ;
– Les 2 serveurs Tomcat seront hébergés sur une machine physique de telle sorte que cette machine ne pourra accueillir d'autres VM ;
– Les serveurs MySQL-Proxy et $MySQL_1$ seront hébergés sur une machine physique de telle sorte que cette machine ne pourra accueillir d'autres VM ;

La montée en charge du nombre de client web RUBIS entraînera l'ajout d'un serveur MySQL ($MySQL_2$), par l'instance de TUNeEngine du niveau application. $MySQL_2$ sera exécuté par l'instance TUNeEngine de niveau IaaS dans une nouvelle VM sur une nouvelle machine physique (figure 10.8(b)). Inversement, la baisse de charges en dessous du seuil minimal entraînera le retrait d'un serveur MySQL, donc de la VM l'hébergeant. C'est à ce niveau que nous implantons la "scalabilité non ordonnée" : l'instance TUNeEngine du niveau application exécute une politique de retrait de MySQL de telle sorte que les serveurs retirés sont prioritairement les premiers serveurs démarrés. Cette politique se traduira dans notre expérience par le retrait du serveur $MySQL_1$. En conséquence, la machine hébergeant la VM du serveur $MySQL_1$ se retrouvera avec une seule VM : celle hébergeant le serveur MySQL-Proxy (figure 10.8(c)). A cette étape, nous disposons de deux machines hébergeant deux VM dont les tailles permettent leur regroupement.

L'exécution permanente du *Scheduler* de CloudEngine dans l'instance TUNeEngine de l'IaaS regroupera les deux VM hébergeant $MySQL_2$ et MySQL-Proxy sur une seule machine physique (figure 10.8(d)). **N.B.** Rappelons que l'apparition de "trous" observée ici dans l'exécution d'une seule application dans le cloud s'observe également dans une exécution de plusieurs applications dans le cloud. La fin d'une application libérera des ressources et favorisera pour la consolidation.

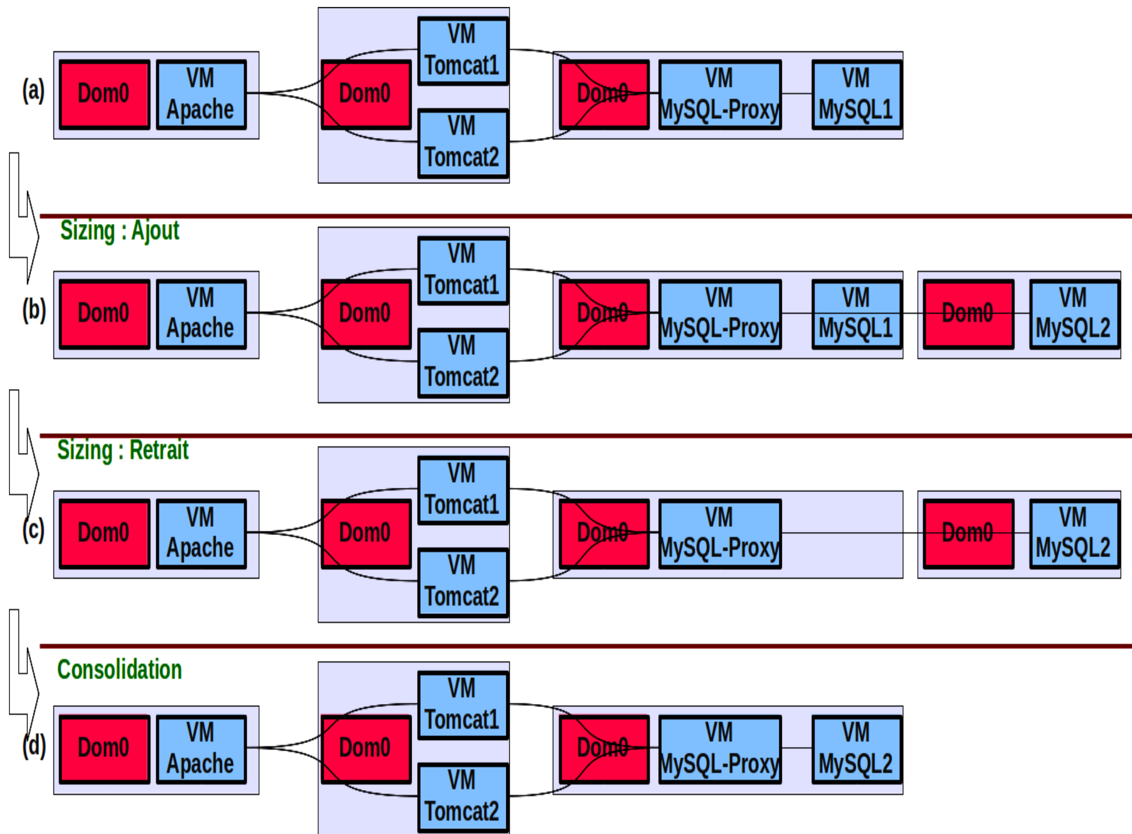


FIGURE 10.8 – Scalabilité niveau application J2EE et Consolidation de VM niveau IaaS : résumé de l'expérimentation

La figure 10.9 montre les résultats de l'expérimentation réalisée suivant le scénario ci-dessus. La courbe en rouge montre l'évolution du débit de l'application RUBIS vue de l'extérieur (à partir du serveur Apache). Le rôle de cette courbe est d'une part de prouver l'implantation de la scalabilité au niveau applicatif et d'autre part de montrer le rapport entre la scalabilité et la consolidation au niveau IaaS.

Les autres courbes montrent l'évolution du nombre de VM sur chaque machine de l'IaaS. Pour des raisons de lisibilité, nous ne présentons pas sur ces courbes l'évolution de la charge client RUBIS ainsi que la variation CPU des serveurs MySQL (elles sont similaires aux courbes observées dans les expériences précédentes). L'interprétation des résultats de cette expérience, présentés dans la figure 10.9, est la suivante :

- La première phase correspond au déploiement des VM hébergeant les serveurs J2EE. Le nombre de VM par machine correspond à la description que nous avons faite précédemment.
- La montée en charge au niveau du tiers MySQL entraîne l'ajout d'un nouveau serveur visible sur la figure 10.9 par le point (a). On constate d'une part que la VM ajoutée est démarrée sur une nouvelle machine (Node4). D'autre part, la chute du débit observée sur la courbe rouge correspond au temps de reconfiguration du serveur MySQL-Proxy pour la prise en compte du nouveau serveur MySQL. Le décalage temporel entre l'allocation d'un nouveau serveur et la reconfiguration de MySQL-Proxy s'explique par le fait que l'allocation

du nouveau MySQL comprend : la copie et le démarrage de la VM devant l'héberger (2Go), le déploiement des binaires de MySQL (35Mo) sur la VM et démarrage de MySQL.
- La baisse de la charge au niveau du tiers MySQL entraîne le retrait d'un serveur MySQL visible sur la figure 10.9 par le point (b). La chute du débit correspond au redémarrage du serveur MySQL-Proxy. Cette chute survient avant le retrait de la VM hébergeant le serveur MySQL parce que nous reconfigurons le serveur MySQL-Proxy avant de déclencher l'opération de retrait qui prend plus de temps (arrêt de la VM et undéploiement de la VM).
- Le retrait de la VM entraîne la consolidation au niveau IaaS, visible sur la figure 10.9 par le point (c). On observe une légère inflexion du débit. Cette inflexion correspond à l'instant de consolidation (par migration de VM). En effet, la migration provoque une grande activité sur les machines impliquées, ce qui réduira le temps processeur alloué aux VM qui s'y exécutent (y compris de la VM en cours de migration).

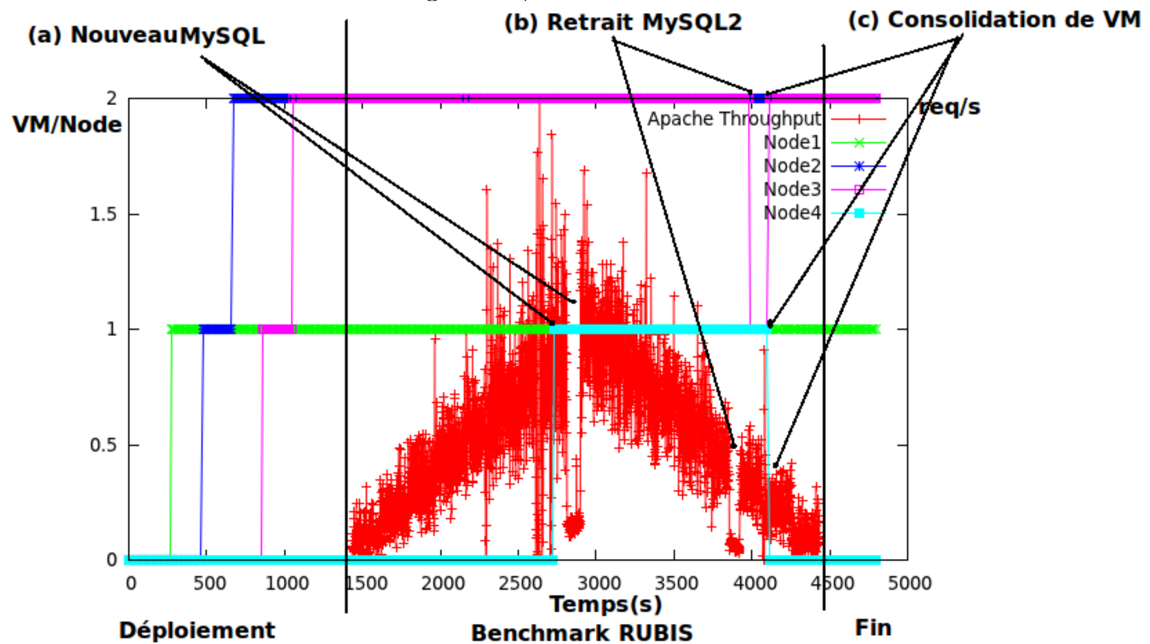


FIGURE 10.9 – Scalabilité au niveau application J2EE et Consolidation de VM au niveau IaaS

Scénario 2 : VM de taille variable et "Scalabilité ordonnée"

Dans cette expérimentation, la configuration de l'IaaS et des serveurs J2EE (à l'exception des serveurs MySQL et leur sonde) que nous utilisons est identique à celle utilisée dans l'expérience précédente. Les VM hébergeant les serveurs MySQL sont celles sur lesquelles nous appliquons la variation de taille. Nous définissons la taille d'une VM par deux valeurs : quota CPU et quota mémoire qui lui sont alloués. Représentons cette taille par le couple [CPU=xxx ;Mémoire=yyy], avec xxx le quota CPU et yyy le quota mémoire en méga octet. Ainsi, la taille maximale d'une

VM sera celle d'une machine physique de l'IaaS : [CPU=0 ;Mémoire=tailleRAM-TailleMémoireDom0]. Cette dernière correspond à :
- un quota CPU de 0 ou 100 signifie que la VM a le droit d'utiliser toutes les ressources CPU de la machine qui l'héberge.
- le quota mémoire maximal d'une VM est la taille mémoire de la machine physique l'hébergeant à laquelle est soustraite la taille mémoire du dom0.

Au niveau de l'application RUBIS, la sonde chargée d'observer le niveau d'utilisation du tiers MySQL utilise pour métrique de décision, le temps de réponse de l'application [1]. En effet, dans les expériences précédentes, les charges CPU des VM sont obtenues du *MonitoringController* de l'instance TUNeEngine de gestion du cloud. Or, ces charges ne reflètent pas réellement la consommation CPU de la VM lorsque celle-ci est configurée avec un quota CPU différent de 0/100. La charge CPU de la VM, observé par les sondes de l'IaaS à partir du dom0 des machines (sans accès à la VM) ne correspond à la charge interne de la VM lorsque le quota de cette VM est différents de 0 ou 100. Cette charge représente en réalité l'utilisation CPU de la VM vis à vis de l'utilisation globale des processeurs de la machine physique. L'utilisation des quota empêchera donc les sondes de l'IaaS d'observer une charge CPU supérieure au quota de la VM.

Au niveau IaaS, nous implantons un programme de reconfiguration (*ResizeVM*) dont le but est d'assigner à une VM de l'IaaS la taille souhaitée par son propriétaire. Ce programme pourra décider en cas de nécessité, de la migration de la VM de sa machine d'hébergement initiale, vers une autre machine pouvant supporter l'augmentation de la VM. Cette situation survient lorsque la machine hébergeant la VM contient d'autres VM telle que l'augmentation de la taille de cette VM ne soit pas possible sur sa machine d'origine (manque de ressources sur la machine).

Quant au *Scheduler*, il s'exécute régulièrement et consolide les VM lorsque des "trous" (laissés par la modification des tailles de VM ou migration) sont constatés sur les machines de l'IaaS.

Le scénario mis en place pour notre expérience est le suivant. Lorsque la sonde d'observation du tiers MySQL se rend compte du dépassement du temps de réponse d'un seuil maximal (40 secondes dans notre cas), il déclenche dans l'instance de TUNeEngine d'administration de l'application RUBIS, l'exécution d'un programme de "sizing". Ce dernier choisit la VM de plus petite taille et émet en destination de l'instance TUNeEngine du Cloud, une requête d'augmentation de la taille de cette. Le programme de sizing est informé du succès ou de l'échec de la réalisation de sa requête. L'échec peut être due à l'indisponibilité de ressources dans le cloud ou l'impossibilité d'augmenter la taille de la VM car ayant atteint la taille maximale d'une machine physique. Dans ce cas, le programme de sizing demande l'allocation d'une nouvelle VM sur laquelle il démarrera un nouveau serveur MySQL (scalabilité). Cette demande pourra également échouée si le cloud ne dispose plus de ressources.

Inversement, l'observation de la baisse du temps de réponse en dessous d'un seuil

1. Compte tenue de la configuration de nos serveurs J2EE, l'augmentation du temps de réponse de l'application dépend de la charge des serveurs MySQL. En effet les autres serveurs sont en sous charge durant toutes les expérimentations

minimal (5 secondes dans notre expérience) entraînera la diminution de la taille de la VM ayant la plus grande taille. Le choix de cette politique permet de maintenir pendant la baisse de la charge, un équilibre de taille des serveurs MySQL. Il permet de palier la non implantation de l'équilibrage de charge au niveau du répartiteur de charges MySQL-Proxy.

Pour finir, la diminution de la taille d'une VM jusqu'à l'atteinte de sa taille minimale (fixée dans notre expérience à [CPU=50 ;Mémoire=512]) entraîne son retrait de l'architecture J2EE s'il existe d'autres serveurs MySQL. Notons qu'après le retrait d'une VM, le Scheduler de niveau cloud interviendra pour réaliser d'éventuelles consolidations provoquées par les variations des tailles des VM.

Dans cette expérience, l'application RUBIS est déployé initialement avec un unique serveur MySQL de taille [CPU=50 ;Mémoire=512]. La sonde de monitoring déclenche l'augmentation (respectivement la diminution) de la taille du plus petit (respectivement du plus grand) serveur MySQL de 25 quota de CPU et 512Mo quota de mémoire. L'exécution de la VM hébergeant la première instance de serveur MySQL ($MySQL_1$) s'effectue sur la même machine que le serveur MySQL-Proxy (figure 10.10(a)) comme dans l'expérience précédente. Avec l'augmentation du temps de réponse, due à l'augmentation du nombre de clients RUBIS, la taille de $MySQL_1$ passera successivement de [CPU=50 ;Mémoire=512] à [CPU=75 ;Mémoire=1024] puis [CPU=100 ;Mémoire=1536]. Cette dernière taille de $MySQL_1$, ne pouvant tenir sur la machine l'hébergeant initialement (car nécessitant les capacités CPU d'une machine entière), la VM l'hébergeant sera migrer vers une nouvelle machine (figure 10.10(b)). Ensuite, l'augmentation de la charge entraînera l'ajout d'un nouveau serveur MySQL ($MySQL_2$) (associé à une nouvelle nouvelle VM). L'exécution de la VM de $MySQL_2$ débute avec la taille minimale. Ainsi, le *ResourceAllocator* de CloudEngine fournira comme machine d'exécution de cette VM, la machine hébergeant le serveur MySQL-Proxy (figure 10.10(c)).

Inversement, lorsque le nombre de client RUBIS baisse et entraîne une amélioration du temps de réponse, les tailles des VM des deux serveurs MySQL sont réduites selon la politique décrite ci-dessus. On constatera une consolidation de VM par le *Scheduler* au niveau du cloud (figure 10.10(d)).

La figure 10.11 montre les résultats de notre expérimentation. En rouge, nous présentons la variation du temps de réponse de l'application. Les deux autres courbes montrent l'évolution du nombre de VM sur les machines de l'IaaS hébergeant les serveurs MySQL. La courbe montrant l'évolution du nombre de clients RUBIS n'est pas présentée dans cette figure car elle est similaire à celle des expériences précédentes. L'interprétation de cette figure est la suivante :

- En (a) nous observons l'augmentation de la taille de la VM hébergeant $MySQL_1$. Cette VM passe à une taille de [CPU=75 ;Mémoire=1024]. Cette augmentation permet d'améliorer le temps de réponse (observable par la baisse du temps de réponse). Cette baisse ne provoque pas la diminution de la taille de la machine. En effet, pour déclencher la diminution de la taille de la machine, nous imposons une période de décision suffisamment long pour éviter les effets yo-yo.

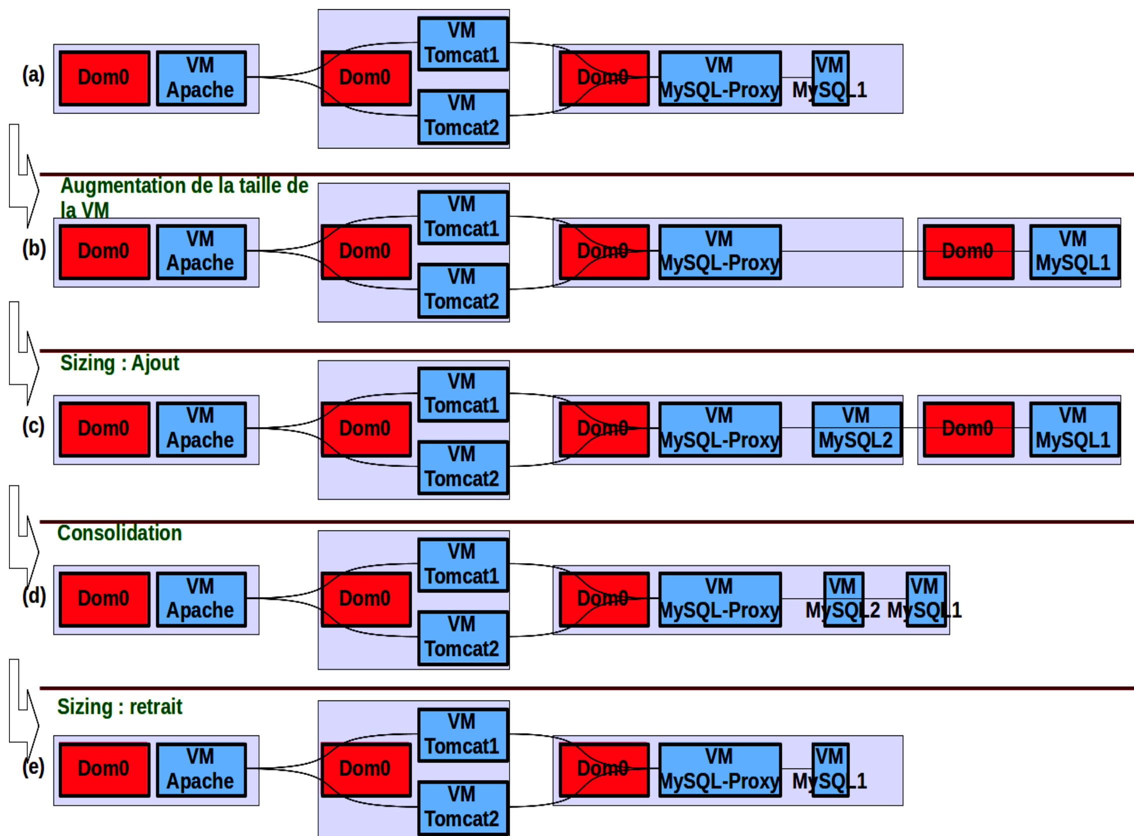


FIGURE 10.10 – Scalabilité niveau application J2EE avec VM de taille variable et Consolidation de VM niveau IaaS : résumé de l'expérimentation

- En (b) nous observons une autre augmentation de la taille de la VM de $MySQL_1$. Elle passe à [CPU=100 ;Mémoire=1536]. Cette augmentation s'effectue après la migration de cette VM de la machine Node3 vers la machine Node4 à cause de la nécessité d'obtenir un quota CPU de 100, autrement dit, une machine entière. Cette augmentation de la taille de la VM par contre n'entraîne pas une baisse significative du temps de réponse. En effet, le nombre important de requête RUBIS émises à ce stade du benchmark empêche les serveurs d'avoir du repos. C'est ce que nous observons en (c).
- En (c) nous observons une montée du temps de réponse qui entraîne l'ajout d'une nouvelle VM d'exécution d'un nouveau serveur MySQL ($MySQL_2$). Cette VM s'exécute sur la machine Node3 compte tenue de sa petite taille initiale.
- En (d), les fluctuations de temps de réponses sont dues à la présence de deux serveurs MySQL de tailles différentes, alors que le nombre de requêtes émis est important. En effet, la petite VM fournit un temps de réponse supérieure à celui de la grande VM. De plus, les requêtes ne sont pas intelligemment distribuées aux deux serveurs en fonction de leur taille. Une expérimentation plus aboutie pourra prendre en compte la reconfiguration du répartiteur de requêtes MySQL-Proxy afin que celui-ci sollicite les serveurs en fonction de leur taille.

Après cette ajout du serveur MySQL, l'IaaS n'acceptera plus d'ajout de VM tant que des ressources supplémentaires ne lui seront pas allouées (ou en cas de libération de ressources).

- En (e) nous observeront l'augmentation de la taille de la VM de $MySQL_2$ jusqu'à la taille [CPU=75 ;Mémoire=1024]. L'IaaS ne peut aller au delà de cette taille dans la mesure où aucune machine ne peut fournir une quantité de ressources pouvant supporter une VM de taille [CPU=100 ;Mémoire=1536]. Par soucis de lisibilité des courbes, nous avons limité les valeurs de temps de réponse sur la courbe rouge au seuil maximal de 40 secondes. Les valeurs supérieures à ce seuil, représentées par la zone (e), ne sont pas visibles dans la figure.
- En (f) nous observons que la diminution des tailles des VM de $MySQL_1$ et $MySQL_2$ provoquera la consolidation de la VM de $MySQL_1$ sur la machine Node3.
- Pour finir, la baisse continuelle de la charge provoquera le retrait d'un serveur MySQL ($MySQL_2$, car scalabilité ordonnée), ainsi que de sa VM.

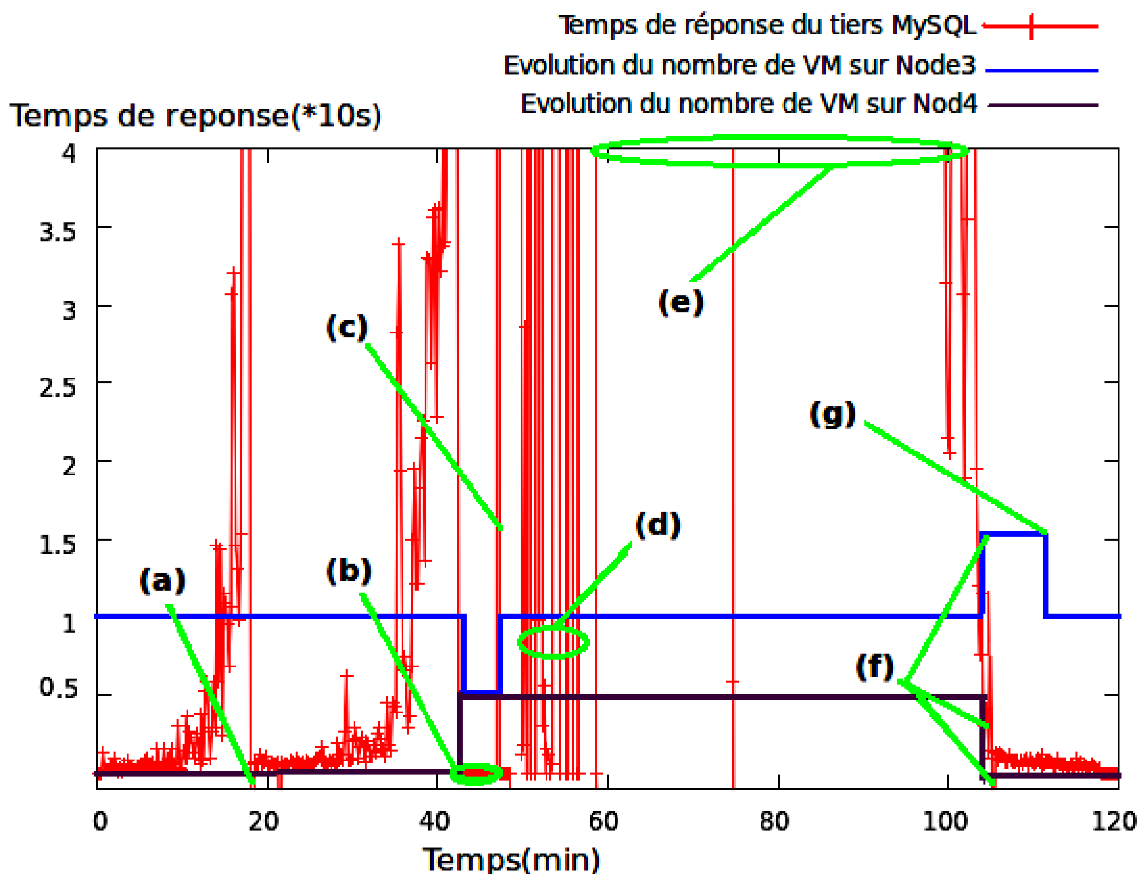


FIGURE 10.11 – Scalabilité au niveau application J2EE et Consolidation de VM de tailles variables au niveau IaaS

10.3 Synthèse

Dans ce chapitre, nous avons présenté différentes évaluations des aspects de l'administration d'une plateforme de cloud simplifiée CloudEngine avec le SAA TUNeEngine. Ces évaluations ont également pris en compte l'administration par TUNeEngine des applications s'exécutant dans le cloud. Les aspects de l'administration que nous avons évalués sont :

– Deux politiques de réparation de VM dans le cloud. La première méthode dite non collaborative réduit l'efficacité des applications s'exécutant dans le cloud. Quant à la seconde, son impact n'est visible que lorsque survient une panne de VM. Comme nous l'avons évoquée, cet impact peut être minimisé par l'introduction de serveurs miroirs.
– Une politique de scalabilité au niveau application entreprise : nous avons montré comment le système TUNeEngine peut être utilisé pour implanter la scalabilité de serveurs dans le cloud afin d'allouer efficacement des VM.
– Une politique de consolidation : nous avons montré l'utilisation de TUNeEngine dans le cloud pour l'implantation d'une politique de consolidation lorsque les VM réservées par les entreprises ne sont pas utilisées efficacement (sous utilisées).
– Pour finir, nous avons évalué deux scénarios mettant en valeur les apports d'une gestion simultanée de ressources par plusieurs les deux niveaux d'utilisation de TUNeEngine dans le cloud : consolidation au niveau de l'IaaS et scalabilité au niveau application entreprise.

Le but de ces expériences n'a pas été d'évaluer de façon exhaustive tous les scénarios d'administration avec TUNeEngine dans le cloud. Il s'agissait de prouver d'une part l'aspect fonctionnel de TUNeEngine et d'autre part son adaptation à différents scénarios d'administration.

Chapitre 11

Conclusion et perspectives

Contents

11.1 Conclusion **144**
11.2 Perspectives **146**
11.2.1 Perspectives à court terme 146
11.2.2 Perspectives à long terme 148

11.1 Conclusion

Ces deux dernières années ont vu le développement d'une nouvelle pratique d'hébergement : le Cloud Computing. Le principe fondateur de cette pratique est de déporter la gestion des services informatique des entreprises dans des centres d'hébergement gérés par des entreprise tiers. Ce déport a pour principal avantage une réduction des coûts pour l'entreprise cliente, les moyens nécessaires à la gestion de ces services étant mutualisés entre clients et gérés par l'entreprise hébergeant ces services. Cette évolution implique la gestion de structures d'hébergement à grande échelle, que la dimension et la complexité rendent difficiles à administrer. Présenté dans la littérature comme une nouvelle technologie, nous retenons de l'étude du Cloud Computing qu'il s'agit globalement d'une plateforme de grille ou centre d'hébergement fournissant à ses clients des services de plus ou moins haut niveau.

L'étude des problèmes que nous avons menée autour du Cloud Computing fait ressortir deux difficultés majeures : l'isolation (des défaillances, des ressources, des utilisateurs) et l'administration dans son ensemble (y compris les applications qu'il exécutent). Ces deux défis jouent un rôle important dans le processus de vulgarisation et d'adoption du cloud auprès des entreprises clientes. Comme nous l'avons présenté dans ce document, le défi d'isolation est résolu dans la plupart des plateformes de cloud par l'utilisation des systèmes de virtualisation comme couche intermédiaire entre les ressources du cloud et les clients. Quant à l'administration dans le cloud, l'étude des travaux de recherches effectués dans ce domaine montre que la majorité

des plateformes de cloud ne prennent pas en compte tous les aspects de l'administration. Cette étude révèle notamment des besoins d'administration à des niveaux différents (hébergeur, client).

Notre contribution dans ce contexte a été de fournir, sur les bases d'un système d'administration autonome (SAA), une plateforme permettant d'administrer à la fois des plateformes quelconques de cloud ainsi que les applications entreprises qu'elles hébergent. Notre choix porté sur l'utilisation d'un SAA a été naturellement motivé par l'aptitude qu'ont montrés les SAA à fournir des solutions d'administration dans des infrastructures de grilles, proches du cloud. C'est ainsi que nous avons entrepris d'utiliser notre SAA TUNe pour l'administration dans le cloud.

La difficulté d'adaptation du système TUNe pour le cloud nous a amené à concevoir et à implanter une nouvelle plateforme hautement adaptable et générique de telle sorte qu'elle pourra prendre en compte différents domaines applicatifs (parmi lesquels le cloud).

Nous avons proposé dans cette thèse un modèle de conception et de développement des SAA hautement adaptables et génériques. Dans un premier temps, nous avons identifié un ensemble de critères essentiels que devront respecter de tels système. Il s'agit de l'uniformité des éléments administrés (revenant à ne pas figer certains mécanismes dans le SAA), de l'adaptabilité du SAA et de sa capacité à collaborer/interagir avec d'autres SAA. Ensuite, nous avons présenté un modèle architecturale prenant en compte ces critères et remplissant les services d'un SAA. Pour valider ce modèle, nous avons développé un SAA (inspiré de TUNe) et conçu selon notre approche. Il s'agit en quelque sorte d'un *exoSAA* baptisé TUNeEngine. Ce dernier a par la suite été adapté (critère d'adaptabilité) afin de prendre en compte l'administration des plateformes de cloud.

En ce qui concerne l'administration dans le cloud, nous avons conçu une plateforme simplifiée de cloud (CloudEngine), dont les composants sont clairement identifiables (contrairement aux autres plateformes), afin de montrer la prise en compte de toutes les facettes de l'administration dans le cloud. Ainsi, nous avons adapté le système TUNeEngine pour la réalisation des tâches d'administration suivantes : construction d'images de VM et téléchargement dans le cloud ; déploiement, configuration, et démarrage des services de CloudEngine ; déploiement et démarrage des machines virtuelles dans le cloud ; réparations des machines virtuelles ; et intégration de quelques politiques d'allocation de ressources dans le clouds. Dans le même ordre d'idées, nous avons montré comment le système TUNeEngine peut être utilisé pour l'administration des applications entreprises dans le cloud. Cette utilisation va de l'interopérabilité avec le cloud pour l'allocation dynamique des machines virtuelles, au déploiement et suivi de l'exécution des applications dans le cloud.

Pour finir, nous avons identifié quelques scénarios d'évaluations de notre SAA TUNeEngine appliqué à la plateforme de cloud CloudEngine, ce qui correspond à notre problématique initiale. C'est ainsi que nous avons évalué dans un premier temps deux méthodes de réparations de VM dans le cloud. Ensuite, nous avons exploré quelques scénarios de consolidation de VM dans le cloud pour une gestion

efficace de l'utilisation des ressources. Enfin nous avons montré comment la gestion de ressources dans le cloud est liée à celle réalisée par les applications entreprises qu'il héberge. Nous constatons par exemple que certaines méthodes de scalabilité ou d'allocation de VM au niveau application entreprise sont propices à la consolidation au niveau du cloud.

11.2 Perspectives

Tout au long de la réalisation de cette thèse, nous avons identifié différents axes potentiels de prolongement de nos travaux. Ces axes concernent à la fois les travaux autour du cloud et de notre SAA adaptable TUNeEngine. Ces axes peuvent être classés en deux catégories : les travaux réalisables dans un futur proche, et ceux réalisables dans un futur plus lointain compte tenu de leur ampleur.

11.2.1 Perspectives à court terme

Clouds Existants

Administration des Clouds Existants : Une prochaine validation de l'adaptabilité de notre SAA TUNeEngine, pourra consister à l'utiliser pour l'amélioration de l'administration des plateformes de cloud existantes. A ce sujet, nous avons initié une adaptation de TUNeEngine pour l'encapsulation des services de la plateforme de cloud OpenNebula [**?**]. Ce travail, non rapporté dans ce document de thèse, est prometteur dans la mesure où cette personnalisation de TUNeEngine pour OpenNebula permet notamment (à ce stade) d'enrichir les capacités de reconfiguration de VM, jusque là difficilement intégrables dans OpenNebula.

Clouds hybrides : Administrée par TUNeEngine, notre plateforme de cloud CloudEngine se contente actuellement du composant de collaboration proposé par le système TUNeEngine pour l'interaction avec des systèmes externes. Cette collaboration a été expérimentée uniquement dans le contexte où les systèmes externes sont également administrés par TUNeEngine. Nous proposons d'étendre ces expérimentations à d'autres plateformes externes utilisant d'autres systèmes d'administration comme Eucalyptus [**?**] ou OpenNebula [**?**]. Une première approche de cette interopérabilité pourra consister à étendre le composant de collaboration de TUNeEngine, utilisé par CloudEngine, par des API WSDL compatible EC2. Dans l'attente d'une standardisation des API d'interopérabilité, EC2 est utilisé actuellement par plusieurs plateformes de cloud.

Intégration de programmes de reconfiguration

Dans sa première validation, le système TUNeEngine reprend et offre tous les services du système TUNe. En particulier, sa prise en compte des programmes de reconfiguration nécessite la description du programme sous forme d'automates. Dans le cadre d'une collaboration courante avec le Laboratoire Informatique de Grenoble

(LIG), les composants d'interprétation et d'exécution des programmes de reconfiguration ont été adaptés pour la prise en compte des programmes de reconfiguration implantés sous forme de programmes Java. En effet, nos collaborateurs du LIG se servent actuellement du système TUNeEngine pour la prise en compte dynamique de programmes de coordination de boucles de contrôles dans le système TUNeEngine, générés par un système externe.

Construction d'images

L'implantation actuelle de notre plateforme de cloud ainsi que son constructeur d'images de VM se limitent aux distributions Linux CentOS. En effet, les techniques de construction d'images et d'initialisation réseau des machines virtuelles dépendent du type d'OS contenu dans l'image. Dans ce premier prototype, nous nous sommes limités dans nos travaux à une unique distribution. Dans le futur, nous comptons étendre cette construction d'images à d'autres distributions Linux. Dans l'attente d'un langage standard d'expression de contenu d'images de VM, nous pourrons nous appuyer sur le format de description d'images OVF (Open Virtual Format) de VMWare (le plus étendu).

Prise en compte d'autres hyperviseurs

Dans notre implantation actuelle, le *VMController* (chargé de l'interfaçage avec les hyperviseurs) de CloudEngine ne prend en compte que les hyperviseurs de type Xen. Nous comptons faire évoluer l'implantation de ce composant via l'utilisation des librairies de virtualisation libvirt [**?**]. De cette façon, nous pourront étendre CloudEngine pour la prise en compte des autres systèmes de virtualisation.

Gestion des utilisateurs

Pour finir, la gestion des utilisateurs n'a pas figuré parmi nos préoccupations initiales. Dans la suite, nous pourront associer à CloudEngine un module de gestion d'utilisateurs et d'authentification. Ainsi, il pourra fournir à chaque utilisateur des modes de gestion personnalisée de VM.

Intégration d'Entropy

Malgré leur pertinences, les politiques de consolidation et de placement de VM dans notre plateforme actuelle de cloud sont simplistes. En effet, elles ne représentent pas le cœur de nous préoccupations dans ce travail de thèse. Cependant, il est envisageable d'intégrer dans la personnalisation TUNeEngine pour l'administration du cloud, un système plus évolué de gestion de ressources dans le cloud. Le système Entropy [**?**], situé dans cette catégorie, constitue une perspective intéressante.

Auto réparation de TUNeEngine

Dans son implantation actuelle, le système TUNeEngine ne peut résister à une panne d'un de ses composants. L'apparition d'une telle panne paralyserait le système. Une amélioration de TUNeEngine pour la tolérance aux pannes pourrait consister à utiliser l'implantation Fractal HA pour la construction de ses composants. En effet, Fractal HA fournit un mécanisme de réplication de composants et maintien la

cohérence entre les réplicas. De cette façon, un composant en panne sera remplacé par son réplica et ce dernier sera dupliqué de nouveau.

11.2.2 Perspectives à long terme

Édition de DSL

TUNeEngine n'est actuellement utilisable qu'au travers d'interfaces de bas niveau que sont l'ADL et les API de Fractal. Dans le cadre de notre projet de recherche, une autre thèse s'intéresse à la conception d'un environnement permettant la conception de DSL. Comme nous l'avons évoqué dans l'étude des problématiques du système TUNe (inspirateur de TUNeEngine), cette plateforme de conception de DSL générera pour chaque langage les adaptations nécessaires à leur prise en compte par TUNeEngine. Autrement dit, à chaque DSL sera associée une personnalisation de TUNeEngine. De cette façon, nous pourront notamment définir un langage d'expression de besoin d'administration dans le cadre du Cloud.

Bibliographie

Table des figures

1.1 Plan en U de ce document de thèse 2

2.1 Évolution vers le Cloud 6
2.2 (a)Vision classique. (b) Notre vision. 13
2.3 Vue des systèmes virtualisés 17
2.4 Techniques de virtualisation 20

3.1 Architecture simplifiée du Cloud 23
3.2 Administration dans le cloud 27

4.1 Organisation d'un SAA 33

5.1 Exemple d'une architecture J2EE 38
5.2 Exemple d'ADL : cas d'une application J2EE 41
5.3 Exemple de wrapping : cas du logiciel Apache 42
5.4 Principes de TUNe 43
5.5 Vue globale de TUNe 45
5.6 Architecture de DIET 49
5.7 Nouvelle orientation du projet TUNe 53

6.1 Modèle architectural général 61

7.1 La plateforme Rainbow 68
7.2 Encapsulation d'un logiciel dans Accord 70
7.3 Composition d'OpenNebula 75
7.4 Organisation d'Eucalyptus 77
7.5 Organisation d'OpenStack 79
7.6 Vue générale de Microsoft Azure 81

8.1 Modèle détaillé de TUNeEngine 86
8.2 Exemple d'architecture Fractal 93
8.3 Soumission de commandes et Collaboration 96
8.4 Construction du SR 98
8.5 Déploiement et Undeploiement 101
8.6 Exécution de programmes d'administration 103

9.1 CloudEngine 107
9.2 Vision synthétique : intégration de CloudEngine à TUNeEngine et utilisation de TUNeEngine pour le déploiement d'applications entreprises dans le cloud. 109

9.3 Vue globale d'administration et d'utilisation de CloudEngine via TUNeEngine . . . 121
9.4 (a) Description d'une image de VM dans le SR de CloudEngine, (b) Description d'une VM dans le SR de CloudEngine, (c) Description du logiciel Apache dans le SR de l'instance TUNeEngine de niveau application Entreprise, et (d) Description d'une VM dans l'IaaS générée par le VMController de CloudEngine. . . . 122

10.1 Environnement matériel de notre IaaS . . . 125
10.2 Coût du checkpointing dans la réparation de VM dans l'IaaS . . . 128
10.3 Réparation de VM de l'IaaS par checkpointing . . . 129
10.4 Réparation collaborative de VM . . . 130
10.5 Récapitulatif de quelques situations nécessitant scalabilité et de consolidation dans le cloud et les applications qu'il héberge . . . 131
10.6 Scalabilité de serveurs MySQL dans une application J2EE dans le cloud 132
10.7 Consolidation dans le cloud : (a) Évolution de la charge CPU sur les machines de l'IaaS, (b) Évolution du nombre de VM par machine de l'IaaS. . . . 135
10.8 Scalabilité niveau application J2EE et Consolidation de VM niveau IaaS : résumé de l'expérimentation . . . 137
10.9 Scalabilité au niveau application J2EE et Consolidation de VM au niveau IaaS . . . 138
10.10 Scalabilité niveau application J2EE avec VM de taille variable et Consolidation de VM niveau IaaS : résumé de l'expérimentation . . . 141
10.11 Scalabilité au niveau application J2EE et Consolidation de VM de tailles variables au niveau IaaS . . . 142