soCloud : une plateforme multi-nuages distribuée pour la conception, le déploiement et l'exécution d'applications distribuées à large échelle

Fawaz Paraiso

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

soCloud : une plateforme multi-nuages distribuée pour la conception, le déploiement et l'exécution d'applications distribuées à large échelle

THÈSE

présentée et soutenue publiquement le 18/06/2014

pour l'obtention du

Doctorat de l'Université Lille I

(spécialité informatique)

par

Fawaz PARAISO

Composition du jury

Président :

Rapporteurs : Françoise Baude, *Université Sophia Antipolis Nice - France*
Samir Tata, *Université TELECOM SudParis - France*

Examinateurs : Alain Tchana, *INP/ENSEEIHT Toulouse - France*

Directeur : Lionel Seinturier, *Université Lille I - France*

Co-encadrant : Philippe Merle, *Inria Lille - France*

Laboratoire d'Informatique Fondamentale de Lille – UMR Lille 1/CNRS 8022 – Inria Lille - Nord Europe

Mis en page avec la classe thloria.

Remerciements

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

En tout premier lieu, je souhaite remercier l'ensemble des membres du jury qui m'ont fait l'honneur de participer à ma soutenance. Je remercie Mme Françoise Baude, professeur à l'Université de Nice Sophia-Antipolis et M. Samir Tata, professeur à Université TELECOM SudParis, pour avoir accepté d'être rapporteurs de mon travail et avoir apporté un jugement constructif. Merci également à M. Alain Tchana, maître assistant à INP/ENSEEIHT Toulouse, pour avoir accepté d'être examinateur de cette thèse.

Je tiens également à remercier mes deux encadrants de thèse pour la confiance qu'ils m'ont témoignée, l'écoute et pour le climat de sérénité qu'ils ont entretenu durant ces années de thèse. Merci à M. Lionel Seinturier, professeur à l'Université Lille I, M. Philippe Merle, chargé de recherche à l'Inria Lille.

J'adresse également mes remerciements à Mme Laurence Duchien, professeur à l'Université Lille I et autres membres de l'équipe SPIRALS pour l'ambiance au travail.

Enfin, un remerciement tout particulier à ma famille : un immense merci à mon épouse pour m'avoir supporté et soutenu durant ces années de thèse. Merci à mon fils Priam Paraiso pour avoir faire preuve de grande patience et compréhension envers son papa qui est souvent occupé. Leur humour et leur bonne humeur ont joué un rôle essentiel dans l'aboutissement rapide de ce travail. Merci à ma mère qui m'a voué toutes ces années un soutien infaillible et a toujours été présente durant les périodes les plus difficiles. Je remercie mon père pour ses conseils et le goût de la recherche qu'il m'a transmis. Merci à Kara Paraiso pour ses conseils, elle m'a incité à aller toujours plus loin. Merci à mes frères et sœurs pour leurs encouragements. Je remercie Teddy Elana qui m'a toujours épaulé et soutenu.

Abstract

Multi-cloud computing has established itself as a paradigm of choice for creating very large scale world wide distributed applications. Multi-cloud computing is the usage of multiple, independent cloud environments, which assumed no priori agreement between cloud providers or third party. However, these applications, designed for multi-cloud environments, have to face real challenges in term of design, architecture, and technology. The possibility of using multi-cloud faces the heterogeneity and complexity of cloud solutions. Thus, multi-cloud computing has to face several challenges such as *portability*, *provisioning*, *elasticity*, and *high availability* we have identified in this thesis.

In this thesis, we propose *soCloud* both a model and a platform that tackle these four challenges. This model is based on the OASIS Service Component Architecture (SCA) standard to design distributed large scale applications for multi-cloud environments. A new language is proposed to effectively express the elasticity of multi-cloud applications through abstraction. The multi-cloud platform is designed to deploy and manage distributed applications across multi-clouds.

The *soCloud* model is illustrated on three distributed applications deployed in multi-cloud environments. The *soCloud* platform has been implemented, deployed and experimented on top of ten existing cloud providers : Windows Azure, DELL KACE, Amazon EC2, CloudBees, OpenShift, dotCloud, Jelastic, Heroku, Appfog, and an Eucalyptus private cloud. These experiments are used to validate the novelty of the contributed solutions.

With our contributions, we aim to provide a simple and effective way to design, deploy, run, and manage distributed applications for a multi-cloud environment by proposing a model and platform.

Keywords: Cloud computing, Multi-clouds, Distributed applications, Platform as a Service, Portability, Provisioning, Elasticity, High availability, Service Oriented Architecture, Service Component Architecture, Domain Specific Language.

Résumé

L'*informatique multi-nuages* s'est imposée comme un paradigme de choix pour créer des applications distribuées à large échelle s'exécutant à des emplacements géographiques répartis. L'*informatique multi-nuages* consiste en l'utilisation de multiples environnements de nuages indépendants qui ne nécessitent pas d'accord a priori entre les fournisseurs de nuage ou un tiers. Toutefois, ces applications conçues pour un environnement *multi-nuages* doivent faire face à de véritables défis en terme d'architecture, de modèle et de technologies. L'utilisation de l'*informatique multi-nuages* se heurte à l'hétérogénéité et à la complexité des offres de nuage. Ainsi, l'*informatique multi-nuages* doit faire face aux défis de la *portabilité*, de l'*approvisionnement*, de l'*élasticité* et de la *haute disponibilité* que nous identifions dans cette thèse.

Dans ce travail de thèse, nous proposons un modèle d'applications nommé *soCloud* qui adresse ces quatre défis. C'est un modèle basé sur le standard SCA du consortium OASIS pour concevoir de manière simple et cohérente des applications distribuées à large échelle pour un environnement *multi-nuages*. Un nouveau langage dédié d'élasticité a été proposé pour exprimer efficacement l'élasticité d'applications *multi-nuages* par l'abstraction. Nous proposons aussi une plateforme *multi-nuages soCloud* conçue pour déployer, exécuter et gérer des applications réparties à travers plusieurs nuages.

Le modèle d'applications *soCloud* a été utilisé pour la mise en œuvre de trois applications distribuées déployées dans un environnement *multi-nuages*. Quant à la plateforme *soCloud*, elle a été implantée, déployée et expérimentée sur dix nuages : Windows Azure, DELL KACE, Amazon EC2, CloudBees, OpenShift, dotCloud, Jelastic, Heroku, Appfog et Eucalyptus. Ces expériences sont utilisées pour valider la nouveauté des solutions approtées.

Grâce à notre contribution, nous visons à offrir un moyen simple et efficace pour concevoir, déployer, exécuter et gérer des applications distribuées pour des environnements *multi-nuages* en proposant un modèle et une plateforme.

Mots-clés: Informatique en nuage, Multi-nuages, Applications réparties, Plateforme en tant que Service, Portabilité, Approvisionnement, Élasticité, Haute disponibilité, Architecture orientée services, Modèle à composants, Langage dédié.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

"Une origine est toujours la fille d'une origine plus ancienne."-Erik Orsenna

Sommaire

1.1 Contexte

Au cours de ces dernières années, avec le développement rapide des technologies de l'information, de nombreuses organisations et entreprises cherchent la meilleure façon de réduire les coûts de fonctionnement, d'assurer la mise à l'échelle de leurs systèmes informatiques, de fournir la bonne performance à leurs applications tout en optimisant l'utilisation des ressources [Sou10]. Les applications complexes présentent différents patrons et comportements que les application classiques n'ont pas. De plus ces applications nécessitent de nouveaux principes de conception basés sur une meilleur compréhension de leurs composants et la manière dont les composants interagissent les uns avec les autres et avec l'environnement. Les techniques de conception et moyens de déploiement, de gestion des applications des utilisateurs et la fourniture de ressources informatiques pour faciliter et garantir la performance souhaitable sont devenus de véritables défis.

Plusieurs technologies ont évolué au fil des années, telles que les systèmes répartis, le traitement parallèle, les grilles de calcul, la virtualisation, etc. Les questions de l'approvisionnement des ressources et le déploiement des applications ont été abordées

par [Bar03, Fos01, Nur09, Roc09, Zha10d]. Avec les exigences métiers actuelles, ces technologies sont moins efficaces en raison de leur inflexibilité, leur coût parfois excessif et leur manque d'évolutivité. Les utilisateurs de ces systèmes ont découvert combien il était difficile de trouver les systèmes capables d'exécuter leurs applications de manière efficace. Il est difficile de mettre à l'échelle une application pour soutenir une charge de travail dynamique avec les défaillances des systèmes pouvant survenir à tout moment. Le redémarrage ou les procédures de reprise après un sinistre peuvent prendre des minutes voire des heures. L'informatique en nuage (Cloud computing) a émergé ces dernières années pour compléter ces technologies et ajouter de nouvelles caractéristiques à l'approvisionnement des ressources pour les applications.

L'informatique en nuage est très attractive pour les entreprises/utilisateurs pour plusieurs raisons économiques : elle nécessite un investissement faible pour l'infrastructure et des coûts de fonctionnement bas qui sont fondés sur la notion de "payez uniquement ce que vous consommez". Cependant, dans le marché de l'informatique en nuage, il n'existe pas un seul nuage homogène mais plusieurs nuages disparates.

1.2 Problématique

Bien que les nuages aient adopté des protocoles communs de communication tels que le HTTP et SOAP, l'interopérabilité, l'intégration et la coordination de tous les nuages restent des préoccupations. La manière d'utiliser le nuage par les entreprises devient de plus en plus complexe. Typiquement, les entreprises ne se satisfont plus du déploiement dans un seul nuage public ou privé. Ces nouveaux besoins changent la vision globale de la conception, du déploiement et de la gestion des applications en nuage.

En effet, il est difficile de trouver un seul nuage qui arrive à satisfaire tous les besoins du client (éviter les pannes, trouver un fournisseur de nuage qui se trouve dans toutes les zones géographiques, sélection d'un fournisseur qui offre la meilleure solution). Les entreprises sont plutôt à la recherche de solutions pour déployer une infrastructure qui s'étend sur plusieurs instances de nuage public et privé. Toutefois, cette tendance *multi-nuages* crée une complexité qui va au delà d'un seul nuage. L'approche *multi-nuages*, par sa nature, permet d'intégrer plusieurs fournisseurs de nuage disparates. Chacune des solutions de nuage a ses propres limites, ses APIs de gestion, ses propres cycles de développement qui doivent être surveillés, gérés pour fournir un ensemble cohérent. L'*informatique multi-nuages* a besoin d'une gestion prudente vue son hétérogénéité. Toutefois, l'*informatique multi-nuages* soulève un certain nombre de défis pour son adoption.

Les applications *multi-nuages* sont des applications réparties, capables de répondre aux nouveaux besoins des entreprises/utilisateurs et aux exigences qui leurs sont associées telles que le placement dynamique, la portabilité, les politiques de mise à l'échelle, la gestion automatique de la charge de travail. En outre, dans un environnement *multi-nuages*, les pannes sont inévitables. La conception d'applications réparties à large échelle qui sont extensibles

et indépendants des détails d'infrastructures requiert des exigences en terme d'architecture, de modèle et de technologies. La conception s'avère être plus complexe pour les développeurs d'applications, puisqu'ils doivent prendre en compte ces problèmes d'infrastructures supplémentaires. Le processus de conception d'applications réparties, dont les composants logiciels interdépendants sont déployés et gérés dans des environnements *multi-nuages*, est un défi. Différentes technologies et outils tentent de satisfaire les besoins des développeurs en termes de logiciels, de matériels en décrivant les environnements, l'abstraction des dépendances et l'automatisation du processus.

De nos jours, il existe un certain nombre de travaux dans la littérature [Ane09, Che08] axés sur la satisfaction des besoins des utilisateurs en terme d'ingénierie d'applications. Ces travaux utilisent l'architecture SOA et la virtualisation, en négligeant l'étude de l'environnement de nuage en tant que fournisseur de ressources. Le développement efficace d'applications de nuage hérite des défis posés par les styles d'architecture, la communication des systèmes physiques et les entrées/sorties. Ces défis sont amplifiés considérablement en raison de la nature des systèmes répartis à large échelle et le fait que pratiquement toutes les applications utilisent ou échangent des données de manière intensive.

Bien que les infrastructures de nuage essaient de distribuer des ressources et les charges de travail automatiquement, le développeur d'applications doit avoir la possibilité de placer ses applications ou ses données dans un lieu géographique donné. Il doit pouvoir exprimer la façon dont son application sera mise à l'échelle, sa localisation ou choisir le type et la quantité de ressources adéquate pour celle-ci. Naturellement, le coût joue également un rôle important dans le choix du type de ressources. Dans tout système réparti, la latence, les défaillances ou pannes de nœuds sont des facteurs à prendre en compte. Le développeur doit pouvoir spécifier le nombre de réplicats de son application ou de ses données. Certaines applications peuvent être constituées de plusieurs composants. Chaque composant peut être déployé sur une instance de machine virtuelle et communiquer avec d'autres composants dans l'environnement *multi-nuages*. L'efficacité, la cohérence, l'évolutivité deviennent des préoccupations majeures pour le développeur d'applications.

En outre, en raison de la topologie du réseau partagé qui est parfois inconnue, la latence réseau peut considérablement augmenter dans les infrastructures de nuage, ce qui affecte les performances de l'application. Un aspect essentiel à prendre en compte lors du développement d'application répartie est la latence réseau entre les composants connexes qui affecte la performance. Comment exprimer ce type de besoin ? Le processus de conception d'applications pour un environnement *multi-nuages* doit prendre en compte toutes ces exigences et offrir un cadre simple et cohérent aux développeurs pour exprimer ces besoins non-fonctionnels.

Par ailleurs, existe-il une plateforme permettant de déployer, d'exécuter et gérer des applications dans un environnement *multi-nuages* et parvenir à assurer leur disponibilité, leur performance et l'utilisation optimale des ressources ?

1.3 Questions de recherche

Dans cette thèse, nous adressons les deux questions de recherche suivantes :

Question de recherche 1

Quel modèle pour des applications multi-nuages ?

La plupart des solutions fournies pour concevoir une application sont souvent monolithiques. Elles donnent peu ou pas de possibilité de personnalisation. Ces solutions monolithiques sont susceptibles de ne pas répondre aux exigences métier de plusieurs consommateurs. Pour concevoir une application pour un environnement *multi-nuages* de manière efficace, il est essentiel d'utiliser un modèle permettant de définir de manière générale et spécifique l'architecture de l'application. Ce modèle doit utiliser les principes de l'ingénierie logicielle. Le développeur doit pouvoir utiliser un cadre simple et efficace pour exprimer les besoins non-fonctionnels et décomposer les unités de déploiement de son application répartie.

Question de recherche 2

Quelle plateforme pour des applications multi-nuages ?

Une des raisons fondamentales de l'adoption de l'informatique en nuage est la possibilité de minimiser les coûts de fonctionnement. La plupart des développeurs ne sont intéressés que par l'exécution et la performance de leurs applications réparties. Pour éviter le phénomène “captif d'un fournisseur", les applications doivent être portables. Ceci implique qu'on doit pouvoir développer l'application une fois et la déployer sur n'importe quel nuage. Les applications ont des façons différentes de consommer des ressources et par conséquent ne sont pas capables d'utiliser de manière optimale les ressources d'un seul nuage en dépit des défaillances. De par la localisation géographique des utilisateurs finaux d'une application, on doit être capable de placer les applications de telle sorte qu'elles soient proches de ces derniers. Dans le but d'utiliser de manière efficace les ressources et donc d'économiser de l'argent, il est nécessaire de mettre en place des mécanismes d'élasticité permettant de faire face efficacement à la gestion des ressources. Dans tout système réparti, les pannes sont inévitables. Il faudra mettre en place des mécanismes permettant de répliquer les applications dans différents nuages pour en assurer la disponibilité.

1.4 Contributions

En réponse à ces deux questions de recherche, nous mettons en évidence dans cette section les contributions scientifiques d'une part pour le modèle permettant de concevoir une

application orientée service à large échelle dans un environnement *multi-nuages* et d'autre part pour la mise en place d'une plateforme *multi-nuages* permettant de déployer, d'exécuter et de gérer les applications en nuages. Ces travaux ont abouti à la proposition d'un modèle d'applications et d'une plateforme nommée *soCloud* qui sont décrits respectivement dans les chapitres 4 et 5.

Contribution 1

Modèle pour concevoir des applications dans un environnement multi-nuages.

Nous proposons un modèle de haut niveau qui sert de support à la conception des applications pour un environnement *multi-nuages*. Il est basé sur un formalisme offrant la capacité de modéliser l'architecture d'applications réparties à l'aide de composants. Chaque composant réifie un ensemble d'éléments logiciels applicatifs de sorte qu'il participe à la fourniture d'une abstraction uniforme des opérations de configuration de ces éléments logiciels. Le modèle proposé est ouvert et s'appuie sur l'extension du standard SCA [Cha07]. Ce modèle capture les informations nécessaires pour exprimer des besoins non-fonctionnels des applications réparties et offre un moyen simple aux développeurs pour exprimer les besoins de leurs applications. Ce modèle constitue un cadre simple et efficace pour décrire l'architecture d'une application répartie orientée services. De plus, nous proposons un langage dédié (DSL) à base d'annotations pour exprimer l'élasticité.

Contribution 2

Mise en œuvre d'une plateforme multi-nuages permettant de déployer, d'exécuter et de gérer des applications distribuées.

Une approche architecturale est présentée pour mettre en œuvre la plateforme *soCloud* assurant l'interprétation des méta-données issues du modèle de l'application pour réaliser le déploiement, l'exécution et la gestion des applications réparties. *soCloud* est une plateforme *multi-nuages* répartie. Nous avons fait des expériences qui ont montré que notre approche permet d'assurer la *haute disponibilité* et l'*élasticité multi-nuages* en utilisant la réplication et en adaptant le système aux changements de l'environnement. Ensuite, nous avons analysé le surcoût que peut introduire la plateforme *soCloud* .

1.5 Organisation du document

Afin de faciliter la lecture du manuscrit et d'en améliorer la compréhension, celui-ci a été construit selon un plan structuré. Un exemple tenant lieu de fil rouge a été choisi et servira d'illustration à l'ensemble du propos. La figure 1.1 présente la relation entre les différents chapitres et la catégorie à laquelle ils appartiennent. Le document est organisé en quatre

parties. Les chapitres 2 et 3 font référence à l'état de l'art. Les chapitres 4 et 5 constituent la partie contribution. Le chapitre 6 porte sur la validation. Enfin, le chapitre 7 traite des conclusions et perspectives.

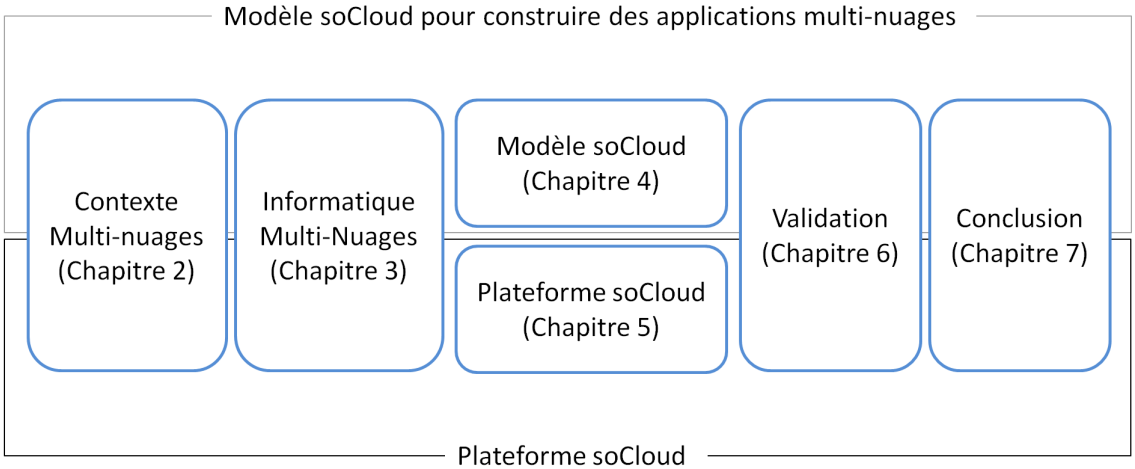

Figure 1.1 – Relation entre les différents chapitres.

Plus précisement, ce document est organisé comme suit :

— Le chapitre 2 présente les concepts liés aux *multi-nuages* que nous avons utilisés dans ce manuscrit.
— Le chapitre 3 présente et discute des travaux de la littérature qui concernent les modèles pour concevoir des applications en nuage et les architectures *multi-nuages* existantes. L'étude de ces architectures *multi-nuages* se fera par rapport à un ensemble de critères qui représentent les défis liés à l'informatique *multi-nuages*.
— Le chapitre 4 présente le modèle pour concevoir une application orientée services pour un environnement *multi-nuages*. Ce modèle offre un cadre simple et efficace pour concevoir de manière générale ou spécifique une application pour un environnement *multi-nuages*.
— Le chapitre 5 présente l'architecture et le choix d'implémentation de la plateforme *multi-nuages soCloud* permettant de déployer, d'exécuter et de gérer une application SaaS dans un environnement *multi-nuages*.
— Le chapitre 6 présente à la fois la validation du modèle d'applications pour un environnement *multi-nuages* et celle de la plateforme *multi-nuages soCloud* au travers de cas d'études.
— Enfin, le chapitre 7 conclut ce document et présente les perspectives des travaux présentés.

Les travaux de cette thèse ont été réalisé au sein de l'équipe SPIRALS commune à Inria Lille - Nord Europe et à l'Université de Lille 1. Ils ont fait l'objet de publications dans un journal, un chapitre de livre, deux conférences et un workshop [Par, Had, Par12a, Par12b, Par13].

Première partie

ÉTAT DE L'ART

Chapitre 2

Contexte Multi-nuages

"Ne vous demandez pas ce que cela signifie, mais plutôt la façon dont il est utilisé."-Ludwig Wittgenstein

Sommaire

2.1 Informatique en nuage

2.1.1 Qu'est ce que l'informatique en nuage ?

Il existe différentes définitions et concepts de l'informatique en nuage [Cha10, Vaq08a, Zho11, Buy08, Wan10, Jer08, Vaq08b, Mel11]. Dans cette section, nous interprétons quelques

définitions sur l'informatique en nuage disponibles dans la littérature (voir tableau 2.1) afin de déterminer un dénominateur commun à celles-ci.

Le document [Jer08] présente les définitions de l'informatique en nuage proposées par plusieurs experts. D'après les auteurs J. Geelan et K. Sheynkman [Jer08], le déploiement, la mise à l'échelle et la consommation des ressources à la demande sont des éléments clés pour l'informatique en nuage. D'autres auteurs tels que R. Buyya et al. [Buy08], W. Lizhe et al. [Wan10] mettent l'accent sur l'accord de prestation de service (SLA) et la qualité de service (QoS). Les auteurs L. Vaquero et al. [Vaq08b] et l'organisme NIST [Mel11] considèrent la mise à l'échelle automatique, le modèle à la demande "payez uniquement ce que vous consommez", l'utilisation ubiquitaire et la virtualisation comme éléments essentiels de l'informatique en nuage. Nous proposons une définition de ce qu'est l'informatique en nuage.

Définition proposée
En tenant compte des caractéristiques communes aux définitions précédentes, nous définissons l'informatique en nuage comme suit :

> *Une solution de virtualisation permettant aux entreprises de toute taille de se procurer et d'utiliser un large éventail des ressources informatiques à la demande, sur la base du principe payez uniquement ce que vous consommez, accessible de n'importe où, sur différents supports (ordinateurs, smartphones, tablettes, etc.) et à tout moment.*

2.1.2 Modèle de service

Il existe différents modèles de prestation et de déploiement dans l'informatique en nuage. La figure 2.1 présente les couches de base de prestations [Boj11, Lee11, Mal11, Mus11] des services de nuage. Ces couches de service sont au nombre de trois :

1. **L'Infrastructure en tant que Service (Infrastructure as a Service ou IaaS)** L'Infrastructure en tant que Service est la couche de base où les ressources informatiques sont fournies aux consommateurs de nuage. Au niveau de cette couche, les ressources (cpu, stockage, mémoire, réseau) sont fournies à la demande afin que le consommateur puisse déployer et exécuter son application. Par exemple, Amazon EC2, Windows Azure, Google Compute Engine et Rackspace sont des fournisseurs de IaaS.
2. **La Plateforme en tant que Service (Platform as a Service ou PaaS)** La Plateforme en tant que Service est une couche de service au dessus de la couche IaaS. Elle fournit aux développeurs les outils et technologies nécessaires pour construire, déployer et gérer le cycle de vie de leurs applications. Le PaaS permet aux développeurs de se concentrer essentiellement sur le développement de leurs applications, sans se soucier de la configuration des infrastructures sous-jacentes. Il existe plusieurs acteurs tels que Salesforce.com, Google App Engine, CloudBees, Heroku et CloudFoundry qui fournissent la couche PaaS.

Tableau 2.1 – Définitions de l'informatique en nuage

Auteurs-Références	Année	Définition en anglais	Interprétation
R. Buyya [Buy08]	2008	A Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.	L'informatique en nuage est une solution de virtualisation de machines interconnectées et approvisionnées à la demande comme un ensemble de ressources unifiées en se basant sur le SLA établi entre le fournisseur et le consommateur.
J. Geelan [Jer08]	2008	Cloud computing is one of those catch all buzz words that tries to encompass a variety of aspects ranging from deployment, load balancing, provisioning, business model and architecture (like Web2.0). It's the n logical step in software (software 10.0). For me the simplest explanation for Cloud Computing is describing it as, "internet centric software".	L'informatique en nuage est une solution d'approvisionnement de ressources, de déploiement, d'équilibreur de charge, de modèle économique et d'architecture Web 2.0.
K. Sheynkman [Jer08]	2008	Clouds focused on making the hardware layer consumable as on-demand compute and storage capacity. This is an important first step, but for companies to harness the power of the Cloud, complete application infrastructure needs to be easily configured, deployed, dynamically-scaled and managed in these virtualized hardware environnements.	L'informatique en nuage est une solution de virtualisation à la demande permettant à tout type d'entreprise d'approvisionner, de déployer et de mettre à l'échelle de manière dynamique son application sans se soucier des configurations des infrastructuctures sous-jacentes.
L. Vaquero [Vaq08b]	2008	Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services).	L'informatique en nuage est une solution de virtualisation de ressources matérielles et logicielles accessibles et utilisables facilement.
W. Lizhe [Wan10]	2010	A computing Cloud is a set of network enabled services, providing scalable, QoS guaranteed, normally personalized, inexpensive computing infrastructures on demand, which could be accessed in a simple and pervasive way	L'informatique en nuage est un ensemble de services accessibles de partout, qui fournit le passage à l'échelle, garantit la qualité de service. Elle est basée sur un modèle économique attractif.
The National Institute of Standards and Technology (NIST) [Mel11]	2011	Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.	L'informatique en nuage est un modèle ubiquitaire accessible à la demande qui fournit un pool de ressources configurables qui peuvent être approvisionnées de manière dynamique.

3. **Le Logiciel en tant que Service (Software as a Service ou SaaS)** Le Logiciel en tant que Service est la couche de service qui est au dessus de la couche PaaS. Cette couche fournit les applications utilisées par les consommateurs. Ces applications peuvent être directement accessibles via le navigateur Web. Nous pouvons citer par exemple, Gmail [Goo04] pour la gestion de courrier électronique, Office 365 [Lyo13] et Google Docs [Goo05] pour l'édition des documents, DropBox [Dra12] pour le stockage. Les consommateurs d'applications n'ont pas à se soucier de l'infrastructure de nuage sous-jacente comme le réseau, le serveur, le système d'exploitation et le stockage. Cependant, le développeur est responsable de la configuration de l'application. Au même moment, les développeurs bénéficient de la puissance de calcul qu'offre l'informatique en nuage.

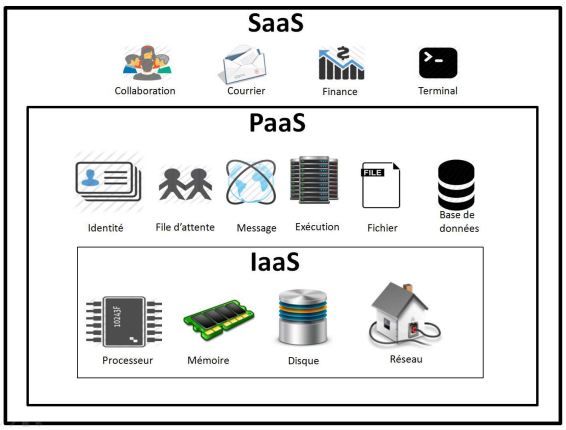

Figure 2.1 – Couche de prestation des services de nuage.

soCloud *vise à fournir un modèle pour concevoir des applications SaaS et une plateforme PaaS bâtie au dessus des IaaS et PaaS existants.*

2.1.3 Modèle de déploiement

Il existe divers modèles de déploiement de l'informatique en nuage [Pop10, Sav11, Yan11, Zha10b]. Ils diffèrent selon les besoins d'utilisation. Selon NIST [Mel11], il existe quatre modèles primaires de déploiement dans l'informatique en nuage :

1. **Nuage privé** Le nuage privé est exploité uniquement par une organisation. La plupart du temps il est hébergé en son sein. Il peut être géré par l'organisation elle-même ou par un tiers. En d'autres mots un nuage privé est une architecture informatique propriétaire ou dédiée à un nombre limité d'utilisateurs, derrière un pare-feu pour assurer le contrôle et la gestion. Par exemple, les organisations qui veulent mettre en place leur nuage privé peuvent utiliser les technologies telles que Eucalyptus [Nur09], OpenStack [Pep11], OpenNebula [Mil11] et CloudStack [Apa13].

2. **Nuage public** Le nuage public est mis à la disposition du grand public qui peut être un grand groupe ou une PME. Il est géré par un organisme appelé fournisseur de nuage qui vend des services de nuage. Le nuage public est le modèle avec lequel les services de nuage sont fournis et tout le monde est libre d'y souscrire. Par exemple, Amazon EC2, Windows Azure et Rackspace sont des fournisseurs d'infrastructure de nuages publics.
3. **Nuage hybride** Le nuage hybride, comme son nom l'indique, est la combinaison de deux ou plusieurs types de nuage. Une organisation peut créer un nuage privé pour ses applications qui sont étroitement intégrées à des systèmes patrimoniaux existants et avoir des exigences de mise à l'échelle que ne peut satisfaire le nuage privé. Ainsi, l'organisme peut exploiter le nuage public en créant un lien entre les deux nuages.
4. **Nuage communautaire** Le nuage communautaire est un modèle qui est partagé entre plusieurs organismes afin de répondre à des besoins spécifiques (par exemple, mission collaborative, sécurité, politique). Le nuage communautaire est conçu pour répondre aux exigences d'une communauté ou d'autres types d'entreprises particulières (prestataire pour l'armée, recherches, etc.) . A titre d'exemple, les entreprises qui travaillent dans des domaines sensibles comme l'armement et qui ont un réseau similaire, peuvent avoir besoin de se réunir sur un nuage communautaire.

soCloud *vise à fournir un modèle de déploiement hybride.*

2.1.4 Caractéristiques essentielles

L'informatique en nuage promeut la disponibilité [Mel11] et est composée de cinq caractéristiques essentielles :

1. **Services à la demande.** C'est la capacité de consommer automatiquement des ressources informatique telles que la machine, le stockage réseau, en fonction des besoins, sans l'intervention humaine avec le fournisseur de service.
2. **Accès au réseau.** Cette caractéristique permet la disponibilité et l'accès au réseau en utilisant un navigateur, un client lourd ou tout autre appareil (smartphone, tablette).
3. **Pool de ressources.** Les ressources informatiques sont regroupées en pool. Ces ressources sont attribuées dynamiquement et réagissent en fonction des demandes des consommateurs.
4. **Élasticité rapide.** C'est la capacité qu'a une application de passer à l'échelle de manière dynamique et rapide en fonction des besoins. Du point de vue consommateur, les ressources utilisées semble être illimitées et peuvent être louées à n'importe quel moment et en quantité illimitée.
5. **Services sur mesure.** Le système de l'informatique en nuage contrôle de manière automatique et optimise l'utilisation de ressources en s'appuyant sur le modèle "payez uniquement ce que vous consommez". Ce mécanisme s'opère à un certain niveau d'abstraction approprié pour le type de service (par exemple, stockage, traitement, bande passante) utilisé.

soCloud *vise à fournir les cinq caractéristiques décrites ci-dessus.*

2.1.5 Acteurs de l'informatique en nuage

L'architecture de référence de l'informatique en nuage de l'organisme NIST [Mel11] définit cinq acteurs majeurs : le *consommateur de nuage*, le *fournisseur de nuage*, l'*auditeur de nuage*, le *courtier de nuage* et le *transporteur de nuage*.

1. **Consommateur de nuage.** Une personne ou un organisme qui entretient une relation d'affaires avec le fournisseur de nuage et utilise le service fourni.
2. **Fournisseur de nuage.** Une personne ou un organisme ou une entité responsable de mettre à disposition des consommateurs des services en nuage.
3. **Auditeur de nuage.** Un tiers qui peut procéder à une évaluation des services de l'informatique en nuage tout en restant indépendant.
4. **Courtier de nuage.** Une entité qui gère l'utilisation, la performance et la prestation de services de nuage tout en négociant les relations entre les fournisseurs et le consommateur de nuage.
5. **Transporteur de nuage.** L'intermédiaire qui fournit la connectivité et transporte les services du fournisseur de nuage vers le consommateur.

En fonction du modèle de service (SaaS, PaaS, IaaS) utilisé, les activités et les scénarios d'utilisation peuvent être différents chez les consommateurs de nuage comme le montre le tableau 2.2.

Tableau 2.2 – Activités consommateur et fournisseur de nuage

Modèle de service	**Activités consommateur**	**Activités fournisseur**
SaaS	Utilise l'application/service pour le traitement.	Installe, gère et maintient l'application sur l'infrastucture de nuage.
PaaS	Développe, teste, déploie et gère l'hébergement dans l'environnement de nuage.	Approvisionne et gère l'infrastructure et les intergiciels pour les consommateurs de nuage. Il fournit les outils de développement, de déploiement, d'administration pour l'utilisateur de la plateforme.
IaaS	Crée, instancie, gère et surveille les services d'infrastructure.	Approvisionne et gère le cpu, la mémoire, le stockage, l'environnement d'hébergement et l'infrastructure de nuage pour les consommateurs de nuage.

soCloud *s'adresse aux consommateurs, fournisseurs, auditeurs et transporteurs de l'informatique en nuage.*

2.1.6 Elasticité dans le nuage

2.1.6.1 Passage à l'échelle

Le passage à l'échelle est la capacité d'un système à supporter une montée en charge de travail tout en conservant une performance adéquate à condition que les ressources matérielles soient ajoutées. Les ressources en question peuvent être la capacité cpu pour le calcul, la capacité mémoire pour le stockage de données et le réseau pour la bande passante. Les deux approches principales pour le passage à l'échelle sont le passage à l'échelle *verticale* et *horizontale* (voir figure 2.2). La mise à l'échelle *verticale* accroît la capacité globale de l'application en augmentant les ressources au sein d'un nœud existant. L'idée principale est d'accroître la capacité des nœuds individuels en améliorant leurs ressources matérielles (cpu, mémoire, disque, bande passante). La mise à l'échelle verticale est limitée par la capacité matérielle physique utilisable. La mise à l'échelle *horizontale* augmente la capacité globale de l'application en ajoutant de nouveaux nœuds. Chaque nœud supplémentaire ajoute typiquement une capacité équivalente en terme de cpu, mémoire. La mise à l'échelle *verticale* est l'approche la plus simple, mais elle est plus restrictive. Quant à la mise à l'échelle *horizontale*, sa mise en œuvre est plus complexe mais peut offrir des capacités qui dépassent de loin celles qui sont possibles avec la mise à l'échelle *verticale*. La mise à l'échelle *horizontale* est l'approche la plus utilisée dans le domaine de l'informatique en nuage, car tous les systèmes d'exploitation ne supportent pas la mise à l'échelle *verticale*. Le passage à l'échelle d'une application se mesure avec le nombre d'utilisateurs qu'elle peut supporter au même moment.

La mise à l'échelle atteint sa limite quand les ressources matérielles s'épuisent, ainsi la mise à l'échelle peut parfois être prolongée en fournissant des ressources matérielles supplémentaires. La manière dont nous ajoutons des ressources supplémentaires définit les deux approches de mise à l'échelle qu'on choisit. Les défis architecturaux de la mise à l'échelle *verticale* et *horizontale* diffèrent. Par exemple, la mise à l'échelle *horizontale* a tendance à influencer l'architecture de l'application, car l'application est répliquée. Tandis que la mise à l'échelle *verticale* se focalise essentiellement sur l'infrastructure matérielle.

2.1.6.2 Mécanisme de passage à l'échelle

Le passage à l'échelle d'un système est une condition préalable pour dire que ce dernier est élastique. L'élasticité est l'une des caractéristiques les plus connues de l'informatique en nuage [Par13]. Ainsi, l'élasticité est liée à la capacité d'un système à s'adapter aux changements de charge de travail et aux demandes de ressources. L'élasticité est le degré avec lequel un système est capable de s'adapter aux changements de charge de travail en approvisionnant ou en retirant des ressources de manière automatique de telle sorte que sur un intervalle de temps les ressources disponibles correspondent au plus près possible de la demande courante [Mel11].

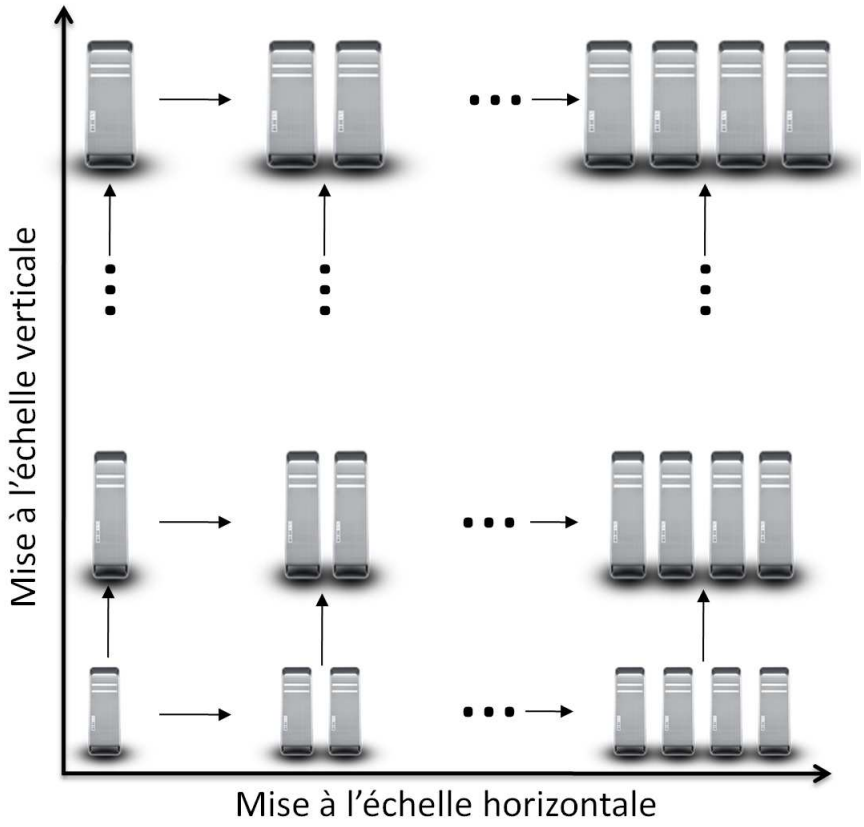

Figure 2.2 – Mise à l'échelle verticale vs horizontale.

Avec le passage à l'échelle *verticale*, la configuration du système peut être ajustée dynamiquement à l'exécution en quelques secondes ce qui est est plus rapide que les minutes que peuvent mettre le passage à l'échelle *horizontale*. Toutefois, le passage à l'échelle horizontale est très commun dans les architectures d'Internet depuis le début du réseau.

2.1.6.3 Mise en œuvre de l'élasticité

L'élasticité est la propriété que possède un système lui permettant de faire face de manière dynamique à des phénomènes (charge de travail, changement d'environnement) imprévus ou spontanés et à être restauré sous sa forme initiale. Le mécanisme d'élasticité est conçu pour être en mesure de passer à l'échelle l'application en cours d'exécution, sans l'interrompre. Il existe deux façons principales pour mettre en œuvre le contrôle d'élasticité. Le **système réactif** réagit aux changements, mais ne les prévoit pas. Le **système prédictif** essaie de prévoir les besoins futurs en terme de ressources et fournit les ressources suffisantes à l'avance.

Les méthodes à base de règles sont réactives. Elles définissent les conditions de passage

à l'échelle basées sur des seuils de métriques donnés. Certains fournisseurs de nuage tels qu'Amazon EC2, RightScale et Windows Azure utilisent les méthodes à base de règles. Par exemple, les auteurs du papier [Bre05] proposent une méthode de régression pour adapter dynamiquement les seuils de métriques en fonction de la qualité de service mais ne prédisent pas la charge de travail futur.

Les méthodes prédictives ont tendance à utiliser des séries chronologiques, la théorie du contrôle, l'apprentissage par renforcement ou la théorie des files d'attente. Une stratégie consiste à utiliser un indicateur de charge de travail combiné avec un modèle de performance pour déterminer le nombre de machines nécessaires à l'application pour prédire la demande. Il existe une variété de modèles de performance [Bod09, Cal11, Roy11]. Le contrôleur de prédiction utilise également des prévisions de charge de travail pour estimer à mi-parcours la quantité de ressources à approvisionner.

Les méthodes hybrides utilisent la combinaison des méthodes réactives et prédictives. D'une manière générale, dans la mise en œuvre des mécanismes d'élasticité, les systèmes autonomes sont beaucoup utilisés.

soCloud *vise à fournir une méthode hybride pour gérer son mécanisme d'élasticité.*

2.1.6.4 Système autonome

Un système autonome se caractérise par des fonctionnalités telles que l'auto-configuration, l'auto-diagnostic et l'auto-réparation en se focalisant sur la capacité d'un système de gérer des défaillances et de faciliter le fonctionnement continu même en présence de pannes. Le comportement autonome est un procédé dérivé de l'informatique autonome [App03, Lin05] qui vise à utiliser des moyens de gestion automatique des ressources informatiques [Has09, Liu11b]. L'informatique autonome est inspirée du fonctionnement du système nerveux humain et vise à concevoir des systèmes qui se gèrent eux-mêmes. Les systèmes autonomes sont adoptés pour l'auto-gestion des systèmes complexes répartis à large échelle qui ne sont plus gérables manuellement [Ana07, De 05, Cho10].

Dans un système autonome, les humains ne contrôlent pas le système. Ils définissent les politiques et des règles générales qui régissent le processus de gestion autonome. Ces systèmes s'adaptent en permanence à l'évolution de l'environnement comme la charge de travail, les ressources matérielles et les pannes logicielles [Kep03, Koe03]. De nos jours, des concepts de systèmes autonomes sont appliqués dans tous les domaines de la science pour fournir la prise de décision intelligente et réaliser l'automatisation pour soulager l'homme des tâches routinières ou complexes.

Une caractéristique importante d'un système autonome est la boucle de contrôle fermée intelligente où un gestionnaire autonome gère les états et les comportements des éléments. Les boucles de contrôle sont de façon générale mises en œuvre en utilisant le processus MAPE (Monitoring, Analysis, Planning et Execution) [Bra09, Mau11] (en français Sur-

veillance, Analyse, Planification et Exécution). La boucle de contrôle avec ses composants et la base de connaissances rendent le gestionnaire autonome et auto-gérable.

soCloud *vise à utiliser un système autonome pour gérer le processus du mécanisme d'élasticité.*

2.2 Raisons d'utiliser les *multi-nuages*

Le rapport du NIST [Hog11] a déclaré que le *multi-nuages* peut être utilisé en *série* lorsqu'une application ou service est déplacé d'un nuage à l'autre ou lorsque les services hébergés sur différents nuages sont utilisés *simultanément*.

Les scénarios les plus simples sont les migrations d'une application d'un nuage privé vers un nuage public (pour l'utilisation en *série*) ou une application qui utilise plusieurs services hébergés dans différents nuages publics (pour l'utilisation *simultanée*). Les raisons pour lesquelles les ressources et services provenant de plusieurs nuages sont nécessaires sont diverses [Pet13a]. Ces raisons sont mentionnées dans divers rapports de recherche et documents industriels [Goi12, Pro11, Sot13]. L'auteur du papier [Pet13a] a fait un résumé non exhaustif en se concentrant essentiellement sur celles qui apparaissent à plusieurs reprises dans différents cas d'utilisation. Les différentes raisons recensées par l'auteur [Pet13a] sont reportées dans le tableau 2.3. Ce tableau mentionne les deux cas d'utilisation de *multi-nuages* (*série* et *simultané*) utilisé dans le rapport NIST. La force motrice pour l'adoption du *multi-nuages* dépend de l'intérêt ou le besoin des acteurs. Les principaux acteurs dans les scénarios de cas d'utilisation sont : le *fournisseur de nuage*, l'*utilisateur* ou le *consommateur de nuage*, le *développeur d'applications* et le *courtier de nuage* [Pet13a]. La figure 2.3 illustre une scission possible des acteurs en fonction de leurs intérêts.

2.3 Taxonomies *multi-nuages*

Il existe plusieurs taxonomies du *multi-nuages* dans la littérature. Par exemple, les articles [Gro12] et [Pet13a] ont établi une taxonomie respectivement sur l'Inter-nuages et sur les *multi-nuages*. Dans cette section, nous allons discuter de ces taxonomies afin de mieux comprendre la classification des solutions *multi-nuages* existantes.

2.3.1 Taxonomie [Gro12]

Les auteurs du papier [Gro12] proposent une classification en fonction de l'architecture des Inter-nuages. La figure 2.4 illustre leur classification. Dans cette classification, on note que l'Inter-nuages est subdivisé en deux grandes familles que sont : la *fédération* et les *multi-nuages*. L'architecture de la famille *fédération* est basée sur la volonté de tiers, tandis que

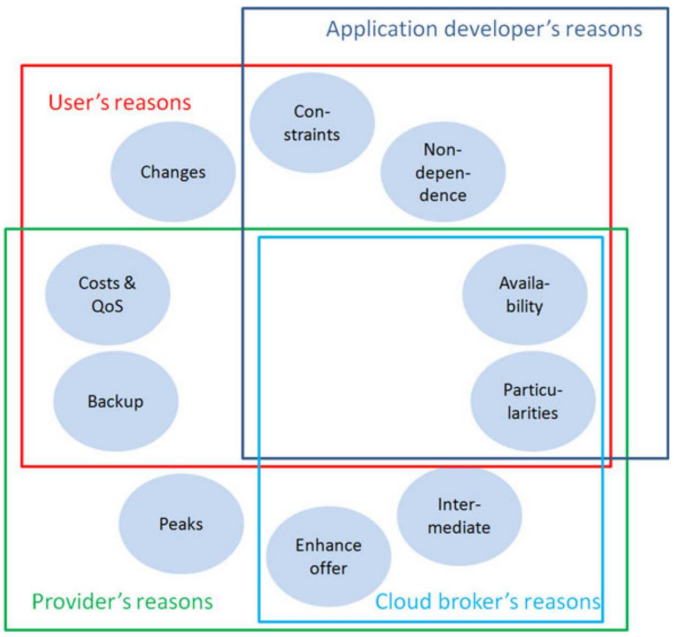

Figure 2.3 – Les raisons pour lesquelles les acteurs peuvent utiliser le *multi-nuages* [Pet13a].

Tableau 2.3 – Le top 10 des raisons pour utiliser le *multi-nuages*.

Type d'utilisation	**Raison**
Utilisation en série	Optimiser les coûts ou améliorer la qualité des services.
	Réagir à l'évolution de l'offre des fournisseurs.
	Prendre en compte des contraintes comme l'emplacement, la restriction législative.
	Éviter la dépendance à un seul fournisseur externe.
	Assurer la sauvegarde pour faire face aux catastrophes ou la planification de l'inactivité.
Utilisation simultanée	Agir comme un intermédiaire.
	Faire face aux pics de services et à la demande de ressources en utilisant des services externes à la demande.
	Réplication d'application ou service en utilisant différents nuages pour garantir leur disponibilité.
	Amélioration de sa propre infrastructure de nuage en passant des accords avec d'autres fournisseurs.
	Consommation de différents services en fonction de leur particularité qu'on ne trouve pas ailleurs.

l'architecture *multi-nuages* est constituée de manière indépendante. L'architecture de *fédération* est constituée par un groupe de fournisseurs de nuage qui veulent collaborer de manière volontaire pour échanger des ressources. Avec l'architecture *multi-nuages*, plusieurs nuages sont utilisés par une application ou un courtier. Chaque famille est elle-même subdivisée en deux sous-familles. Ainsi, dans la famille *fédération*, nous avons deux sous-familles (*centralisée* et *pair-à-pair*). L'architecture *centralisée* est une entité centrale qui agit comme un dépôt dans lequel les ressources de nuages disponibles sont enregistrées. Dans l'architecture pair-à-pair, le nuage est considéré comme un pair qui communique avec d'autres sans médiateurs. Dans la famille *multi-nuages*, nous avons aussi deux sous-familles (*service* et *librairies*). Cette approche est indépendante du fournisseur de nuage et utilise les ressources provenant de différents nuages. Dans l'architecture sous la forme de *service*, l'approvisionnement de l'application est réalisé par un service qui se charge de placer et d'exécuter l'application dans plusieurs nuages. Dans l'architecture sous forme de *librairies*, on utilise des librairies Inter-nuages qui facilitent l'utilisation de plusieurs nuages de manière uniforme.

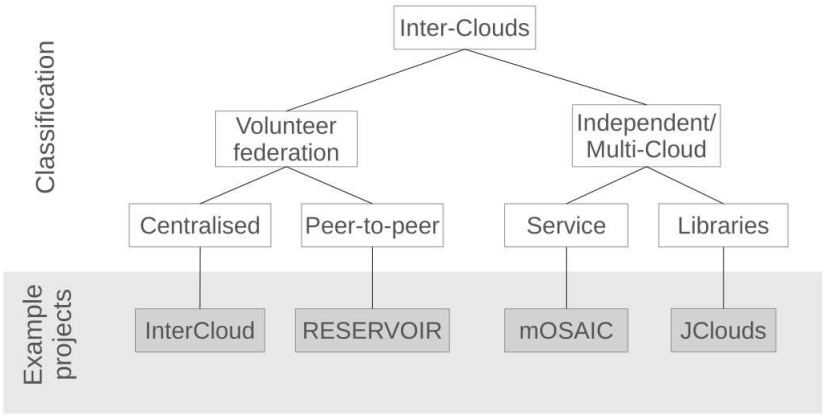

Figure 2.4 – Classification architecturale des Inter-nuages [Gro12].

2.3.2 Taxonomie [Pet13a]

L'auteur du papier [Pet13a] introduit une autre classification basée sur les différentes catégories de *multi-nuages* (voir figure 2.5). Ces catégories sont créées selon le modèle de déploiement. Au premier niveau, la catégorie *multi-nuages* est subdivisée en trois familles : *fédération de nuages*, *multi-nuages* et l'*Inter-nuages*. Le deuxième niveau est lié à l'organisation des différentes interactions entre les nuages impliqués. Les troisième et quatrième niveaux font référence à l'organisation de l'architecture. Le terme *horizontal* dans les sous-familles *fédérations horizontales* et multi-nuages *horizontales* se réfèrent au niveau du déploiement des services de nuage. Par exemple, on agrège plusieurs nuages au même niveau de déploiement SaaS ou PaaS ou IaaS. Tandis que le terme *hiérarchie* dans les sous-familles *fédérations hiérarchiques* et multi-nuages *hiérarchiques* utilise l'agrégation de plusieurs nuages indépen-

demment de leur niveau de déploiement. Par exemple, on peut agréger des ressources provenant du IaaS et PaaS.

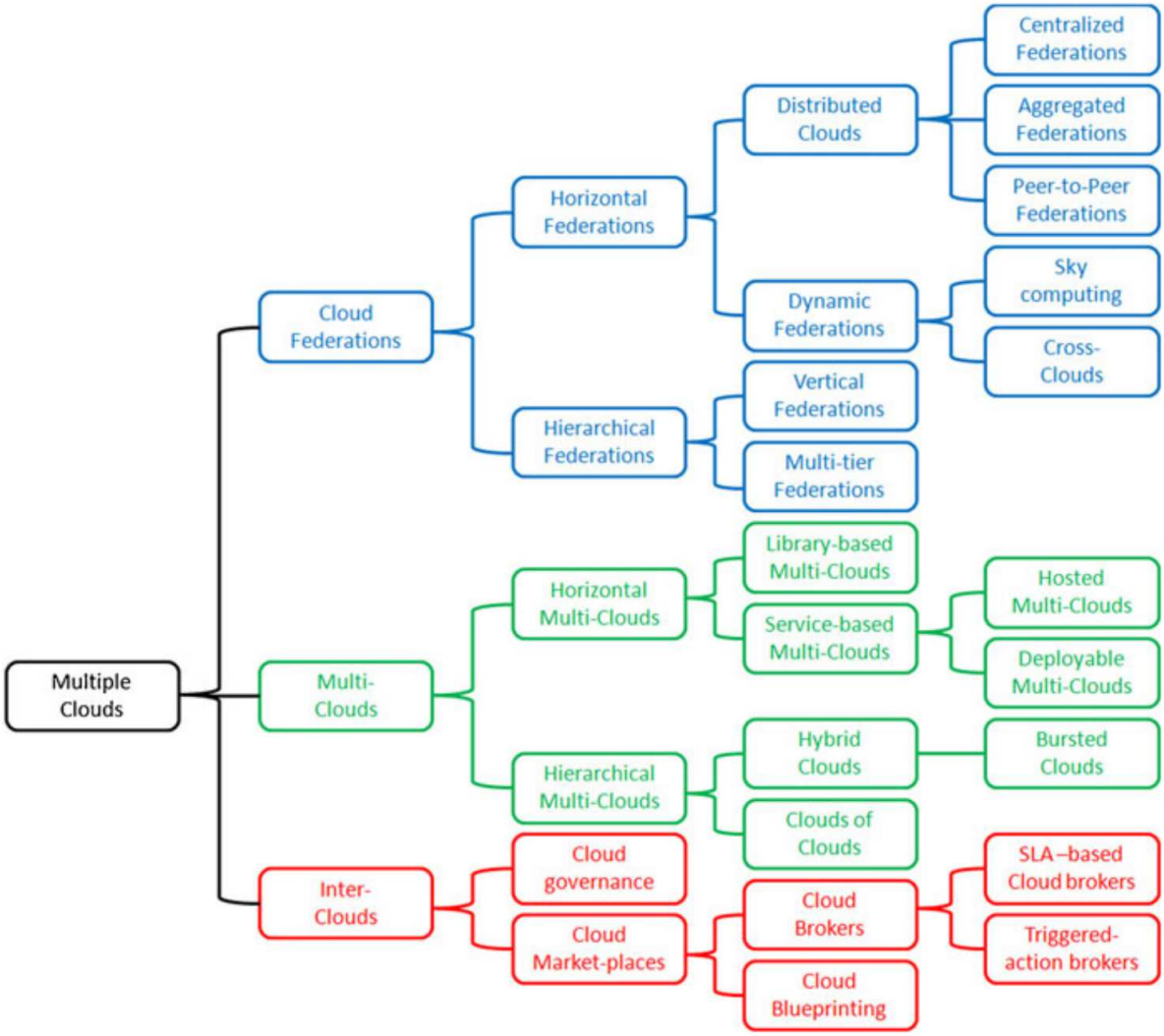

Figure 2.5 – Relations entre les catégories *multi-nuages* [Pet13a].

2.3.3 Discussion

Les deux travaux [Pet13a] et [Gro12] font tous deux une classification *multi-nuages*. Les travaux de l'auteur [Pet13a] mettent la fédération, les *multi-nuages* et les Inter-nuages au même niveau (voir figure 2.5). Tandis que les auteurs [Gro12] placent les Inter-nuages au dessus de la fédération et les *multi-nuages* qui se trouvent au même niveau (voir figure 2.4). Le papier [Gro12] souligne dans leur classification que les fournisseurs de nuage collaborent volontairement (fédération de nuage) ou pas (*multi-nuages*). Les deux catégories *fédération de*

nuage et *multi-nuages* peuvent être distinguées par le type de modèle de déploiement [Fer12]. La différence est faite par le degré de collaboration entre les nuages en question et par la façon par laquelle l'utilisateur interagit avec les nuages. Le premier modèle, appelé nuage fédéré, suppose un accord entre les fournisseurs de nuage. Dans ce modèle, le consommateur de service n'est pas au courant du fait que le fournisseur de nuage qu'il paie utilise des services d'autres fournisseurs. Le second modèle, appelé *multi-nuages*, suppose qu'il n'y a pas d'accord a priori entre les fournisseurs de nuage et un tiers (consommateur de nuage) qui est responsable des services. Dans les *multi-nuages* l'un des principaux problèmes est la portabilité des applications à travers les nuages. Tandis que le principal problème dans les nuages fédérés est l'interopérabilité [Vil12a, Pet11b].

soCloud *se positionne dans la sous-famille de* multi-nuages *orientés services dans la taxonomie [Pet13a] (voir figures 2.6) et dans la sous-famille service* multi-nuages *dans la taxonomie [Gro12] (voir figures 2.7).*

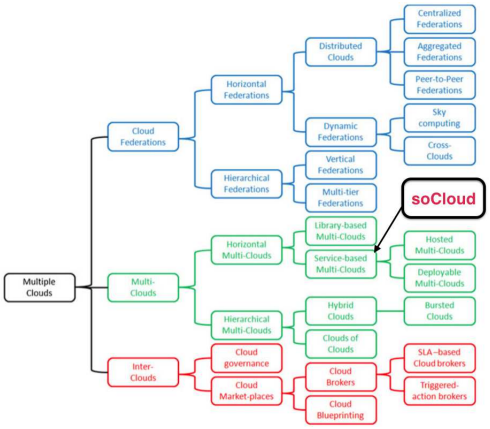

Figure 2.6 – Positionnement de *soCloud* dans la taxonomie [Pet13a].

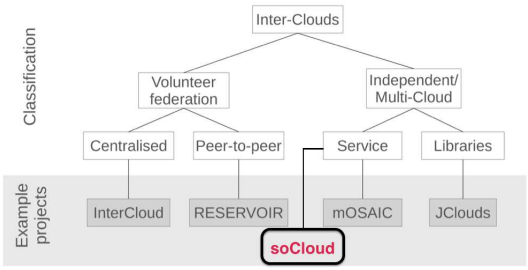

Figure 2.7 – Positionnement de *soCloud* dans la taxonomie [Gro12].

Chapitre 3

Informatique *multi-nuages*

"Dis-moi et j'oublierai, montre moi et je me souviendrai, implique moi et je comprendrai."-Confucius

Sommaire

3.1 Problématiques

L'un des défis posés par le paradigme *informatique multi-nuages* du point de vue génie logiciel est lié aux méthodes, outils et techniques pour aider les développeurs à concevoir, développer, assembler et déployer des applications réparties orientées services à large échelle qui utilisent les technologies *multi-nuages*. En fait, ce sont des tâches difficiles car elles sont beaucoup plus complexes en raison des problématiques liées à l'utilisation des plateformes de nuages hétérogènes. Dans le cadre de notre travail de recherche, nous avons identifié quatre problématiques : la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité* des applications dans un environnement *multi-nuages*.

3.1.1 Portabilité *multi-nuages*

La portabilité en général est la capacité d'utiliser des composants ou des systèmes se trouvant sur de multiples environnements matériels ou logiciels [Pet13b]. En *informatique en nuage*, la portabilité est la capacité à déplacer une application ou des données d'un fournisseur de nuage à un autre sans nécessiter de modifications significatives de cette dernière. Dans l'*informatique multi-nuages*, la portabilité est la capacité à assurer qu'une application ou une donnée fonctionnera de la même façon quels que soient les fournisseurs de nuage sans aucune modification de l'application ou de la donnée. Dans l'*informatique multi-nuages*, la portabilité des données est aussi importante que celle des applications. Dans le cadre de ce travail de thèse, nous nous focalisons uniquement sur la portabilité des applications dans un environnement *multi-nuages*.

Comme nous l'avons souligné dans la section 2.3, l'interopérabilité est un besoin crucial pour la fédération de plusieurs nuages, de même que la portabilité est un besoin crucial pour le *multi-nuages*. Dans un marché émergent et en rapide évolution comme l'informatique en nuage, il est facile de créer des applications qui deviennent captives du fournisseur de nuage en raison de l'utilisation d'API et de formats propriétaires. Pour éviter ce syndrome de dépendance à l'égard du fournisseur, les applications doivent être portables sur diverses Plateformes en tant que Service et Infrastructures en tant que Service de l'informatique en nuage. Ainsi, cette portabilité *multi-nuages* permet la migration d'applications d'un fournisseur de nuage à un autre afin de profiter de meilleur prix ou qualité de service. Cependant, la portabilité d'applications nécessite que le support d'exécution fournisse un modèle commun pour cacher la diversité des PaaS et IaaS sous-jacents. Par exemple, Google App Engine et Salesforce.com soulèvent des problèmes tels que la hausse des coûts et de dépendance exclusive à l'égard de fournisseurs [Kur11]. En outre, le nombre de modèles de programmation dominants aujourd'hui augmente et ils sont de plus en plus complexes. Par exemple, écrire une application Java EE ou une application OSGi qui expose des services Web, effectuant un traitement et qui s'interface avec un système de messages qu'on veut intégrer avec d'autres applications, requiert la connaissance de JAX-WS [Sun03b], JMS [Sun01a] et des APIs EJB [Sun03a]. Cette complexité n'est pas limitée à Java EE [Sun07]. En fait, l'intergiciel .NET [Mic06b] est soumis aux mêmes problèmes [Mar10a]. Développer une application identique en utilisant .NET 2.0 nécessite une connaissance des APIs ASP .NET [Mic06a] tels que : les services Web, les services d'entreprise (Enterprise Services) et .NET Messaging.

SCA est un standard du consortium OASIS [OAS07] (discuté plus en détails en section 4.4.1 page 73). SCA, en revanche, fournit un modèle de programmation simplifié et uniforme aux applications qui communiquent en utilisant une variété de protocoles réseau [Mar10a]. Contrairement aux technologies Java EE et .NET, SCA n'est pas destiné à être une plateforme technologique à tout faire. SCA ne précise pas les mécanismes de persistance des données ou une couche de présentation pour la création d'interfaces utilisateur. Au contraire, SCA intègre d'autres technologies d'entreprise telles que JDBC et Java Persistence Architecture (JPA) pour stocker des données dans une base de données et la myriade d'intergiciels Web qui existent aujourd'hui comme : servlets, JSP, Struts, Java Server Faces et Spring WebFlow, pour n'en nommer que quelques-uns. SCA offre un environnement beaucoup plus productif pour les développeurs qui n'ont pas besoin ou envie de faire face à la complexité des APIs de bas niveau de Java EE.

3.1.2 Approvisionnement *multi-nuages*

L'approvisionnement est un terme qui désigne l'allocation des ressources. Dans l'*informatique en nuage*, l'approvisionnement est à la fois l'allocation de ressources et le déploiement d'applications dans le nuage. Dans l'*informatique multi-nuages*, l'approvisionnement se réfère à la capacité d'allouer des ressources dans plusieurs nuages et de définir le processus de déploiement des applications sur ces ressources.

L'approvisionnement d'applications dans un environnement *multi-nuages* nécessite l'allocation de ressources dans plusieurs nuages, le placement et le déploiement de ces dernières. Il est important de fournir une méthodologie cohérente et un processus pour modéliser la façon dont les applications sont déployées. Ceux-ci offrent la flexibilité et le choix aux développeurs d'utiliser n'importe quel fournisseur de nuage. Ce mécanisme d'approvisionnement d'applications doit offrir une agilité efficace qui se base sur un modèle d'abstraction de haut niveau et entièrement automatisé pour faciliter l'approvisionnement des applications à travers plusieurs fournisseurs de nuage.

Les auteurs du papier [Zha10a] préconisent que les petits centres de données qui consomment moins d'énergie peuvent être plus avantageux que les grands et que la géo-diversité a tendance à mieux répondre aux attentes des utilisateurs. La géo-diversité diminue la latence pour les utilisateurs et augmente la fiabilité en présence de pannes qui atteint l'ensemble des centres de données du fournisseur de nuage [Gre08]. Dans un contexte juridique, la loi sur la protection des données et la confidentialité peuvent amener des utilisateurs à placer leurs données dans une zone géographique donnée. En effet, l'emplacement des données peut faciliter ou restreindre ces dernières sous des juridictions particulières.

Il est difficile de répondre à toutes ces exigences en utilisant un seul fournisseur de nuage. En fait, nous ne pouvons pas compter sur un seul fournisseur pour desservir toutes les zones géographiques.

3.1.3 Élasticité *multi-nuages*

En *informatique en nuage*, l'élasticité est le degré avec lequel le système est capable de s'adapter à la charge de travail en allouant ou en retirant des ressources de manière automatique. Dans l'*informatique multi-nuages*, l'élasticité est le degré avec lequel le système est capable de s'adapter à la charge de travail en allouant ou en retirant des ressources provenant de différents nuages de manière automatique.

La gestion de l'élasticité *multi-nuages* peut être subdivisée en deux approches comme le montre la figure 3.1. La première approche (c.à.d. élasticité à grain fin) qui permet soit le passage à l'échelle horizontale des ressources en changeant le nombre de machines virtuelles qui correspond à l'élasticité horizontale, soit le passage à l'échelle verticale en changeant la capacité des machines virtuelles en fonction des besoins de l'application en terme de mémoire, de stockage , de bande passante et de CPU. La combinaison des deux dimensions (passage à l'échelle horizontale et verticale) est utilisée pour implanter l'élasticité à grain fin. L'*informatique multi-nuages* pousse les deux dimensions (passage à l'échelle horizontale et verticale) au delà des frontières des centres de données et de fournisseurs de nuage. La deuxième approche (c.à.d. élasticité à gros grain) consiste au passage à l'échelle structurelle qui gère l'évolutivité des ressources en changeant les fournisseurs de nuage. L'élasticité à gros grain permet de pallier aux problèmes de pannes qui peuvent faire tomber tout le système d'un fournisseur de nuage ou le manque de ressources disponibles chez le fournisseur.

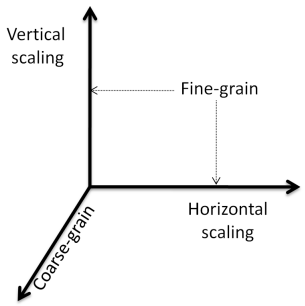

Figure 3.1 – Mise à l'échelle à grain fin et gros grain.

Les applications déployées dans le nuage peuvent créer des charges de travail imprévisibles et très variables dans le système. La prise de décision de l'élasticité devrait être en mesure de réagir aux changements de charge inattendus dans un laps de temps court. Dans le domaine du *multi-nuages*, l'automatisation de la gestion de l'élasticité est une condition obligatoire et par conséquent un principe fondamental lors de sa conception [Isa07].

3.1.4 Haute disponibilité *multi-nuages*

L'article "Above the clouds : a Berkeley view of cloud computing" [Arm10] a souligné plusieurs obstacles et opportunités dans l'informatique en nuage et considère les *multi-nuages* comme un moyen pour assurer la haute disponibilité. Les auteurs de cet article considèrent la disponibilité comme l'un des obstacles principaux et ont analysé les mécanismes de la haute disponibilité utilisés par certains fournisseurs de nuage.

La haute disponibilité est la capacité d'un système qui assure un niveau pré-établi de fonctionnement durant un intervalle de temps. En *informatique en nuage*, la haute disponibilité se réfère au fonctionnement continu d'un système malgré les pannes ou défaillances qui peuvent survenir. Dans l'*informatique multi-nuages*, la haute disponibilité se réfère à l'habileté de fonctionnement continu qu'a un système malgré les pannes en utilisant la réplication dans plusieurs nuages.

Une série de nouvelles [Inf, Zdn12, Sea] et de papiers [Bri09, Cho09, Lea09, Jan11] ont souligné plusieurs pannes de fournisseurs de nuage. Par exemple, le tableau 3.1 récapitule une liste de pannes récentes classées par fournisseur de cloud. Ces données proviennent d'une fouille sur Internet qui souligne les différentes pannes survenues chez les fournisseurs de nuage. Le tableau 3.1 recense seulement les pannes qui ont eu un impact élevé et dont les services ont été mis hors ligne pendant au moins une heure. Selon un rapport récent du groupe de travail international sur la résilience de l'informatique en nuage [IWG], il y a eu un total de 568 heures de temps d'arrêt pour un total de treize fournisseurs de nuage bien connus depuis 2007 et a causé un préjudice financier de plus de 71,7 millions de dollars. L'indisponibilité moyenne des services de l'informatique en nuage est de 7,5 heures par an, ce

Tableau 3.1 – Liste des pannes survenues dans les nuages

	Pannes de nuages		
Fournisseurs de nuages	**Que s'est-il passé ? et pourquoi ?**	**Quand ?**	**Quel impact ?**
Amazon	L'événement a été déclenché pendant une panne électrique à grande échelle dans toute la région du nord de la Virginie. (Voir [Set09]).	29 juin 2009	Inaccessible pendant plus de 5 heures et 30 minutes. Compagnies affectées : Netflix, Instagram, Pinterest, Heroku.
	La perte de puissance a été causée par un défaut de masse électrique et de court-circuit dans un panneau principal de distribution d'électricité qui a interrompu le courant de certaines instances de VMs dans une zone particulière. (Voir [Ric10])	4 et 8 mai 2010	Inaccessible pendant plus de 7 heures. Perte de données qui a effecté un petit nombre d'utilisateurs.
	Le problème a eu lieu sur le bloc de stockage élastique (EBS), qui fournit des volumes de stockage pour les instances Amazon EC2. (Voir [Ric11])	21 avril 2011	Inaccessible pendant plus de 10 heures. Compagnies affectées : reddit, Quora, HootSuite.
	Une panne de courant dans un centre de données d'Amazon Web Services en Virginie du Nord a affecté certains clients. (voir [Bar12])	30 juin 2012	Inaccessible pendant plus de 5 heures. Compagnies affectées : Netflix, Instagram, Pinterest, HotSuite, Heroku.
	Le service d'équilibreur de charge (ELB) d'Amazon EC2 dans la région-est des États Unis est tombé en panne et a affecté les applications qui l'utilisent. (Voir [Ama12b])	25 décembre 2012	Inaccessible pendant plus de 23 heures. Compagnie affectée : Netflix.
Windows Azure	Un problème de réseau au cours d'une mise à jour logicielle de routine a interféré avec le déploiement d'applications hébergées. (Voir [Mic09])	13 mars 2009	Inaccessible pendant 22 heures.
	Elle est causée par un bug logiciel. Le calcul du temps était incorrect pour l'année bissextile. (Voir [Mic12])	29 février 2012	Inaccessible pendant plus d'une heure.
Rackspace	Une panne de courant dans un centre de données. (Voir [MG 09])	29 juin 2009	Inaccessible pendant une heure.
Heroku	La panne d'Amazon a affectée Heroku. (Voir [Car09])	29 juin 2009	-
Salesforce.com	Aucune explication de ce qui s'est passé. (Voir [Ric09])	4 janvier 2010	Inaccessible pendant environ une heure et 15 minutes.

qui représente un taux de disponibilité de 99,9 % selon les résultats préliminaires du groupe. Ces résultats sont loin d'être à la hauteur de la fiabilité requise par les systèmes missions critiques qui exigent une disponibilité de 99,999 %. A titre de comparaison, l'indisponibilité moyenne d'électricité dans une capitale moderne est inférieure à 15 minutes par an [Mau12]. Outre l'impact économique, le temps d'arrêt affecte aussi des millions d'utilisateurs finaux. Bien sûr, le temps d'arrêt coûte de l'argent et cause des dommages. Toutefois, la protection des systèmes contre les interruptions avec un taux de 99.999 % de disponibilité n'est pas gratuite. Il existe trois topologies principales pour gérer la haute disponibilité :

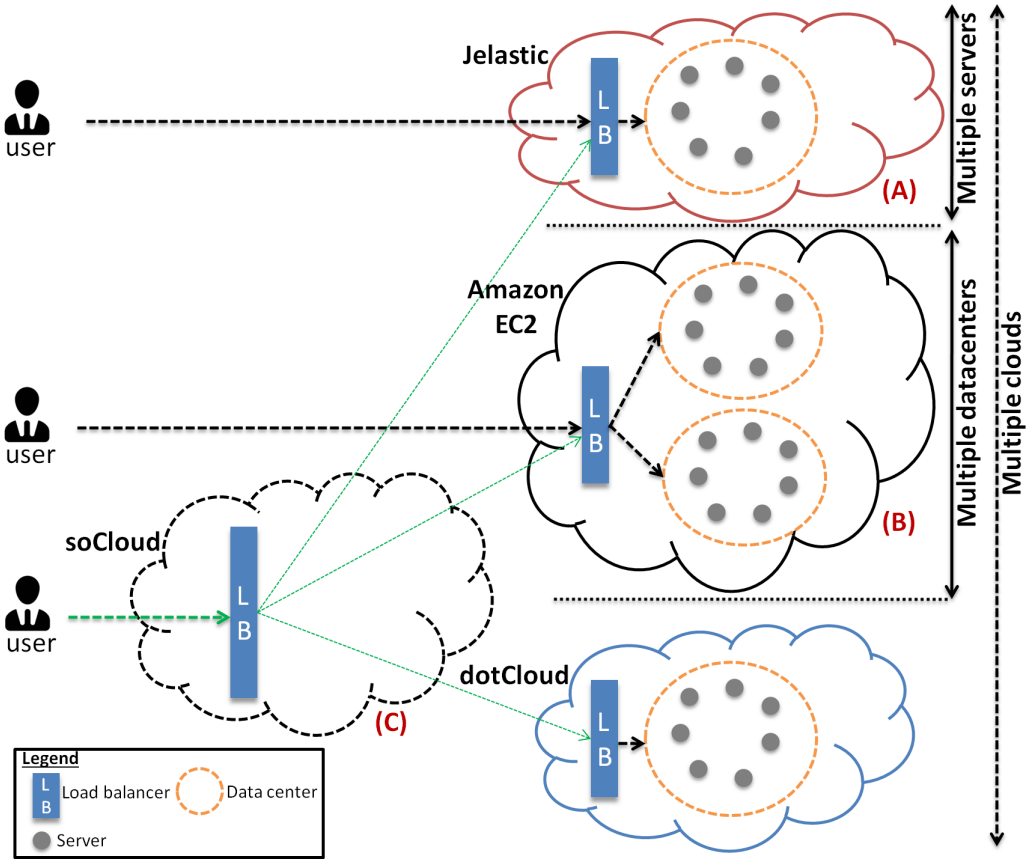

Figure 3.2 – Vue d'ensemble de différentes topologies d'architecture de nuage. (A) multiples serveurs. (B) multiples centres de données. (C) multiples nuages.

1. *Multiples machines :* comme le montre la figure 3.2(A), chaque instance d'une application est répliquée sur différentes machines chez le même fournisseur de nuage dans le même centre de données. Lorsqu'une machine devient indisponible, l'équilibreur de charge commute automatiquement sur une autre machine. Cette topologie ne représente que la disponibilité au sein d'un seul centre de données et n'aborde pas la façon dont les trafics réseaux et les applications sont gérés entre deux ou plusieurs centres de données géographiquement distincts. Le problème de cette approche est que la panne de l'équilibreur de charge met en échec l'ensemble du système. Jelastic et dotCloud utilisent la topologie multiples machines.

2. *Multiples centres de données :* dans la figure 3.2(B), au lieu d'avoir un seul centre de données avec plusieurs serveurs, nous avons plusieurs centres de données. Chaque instance d'une application est déployée dans différents centres de données. Cette topologie gère le basculement entre plusieurs centres de données. Par exemple, quand un centre de données échoue complètement, l'équilibreur de charge bascule automatiquement vers un autre. Toutefois, lorsque l'équilibreur de charge tombe en panne,

tous les services deviennent indisponibles. Amazon utilise la topologie multiples centres de données.

3. *Multiples nuages :* la figure 3.2(C) montre une approche *multi-nuages* où l'équilibreur de charge répartit les requêtes entre plusieurs fournisseurs de nuage tels que Jelastic, Amazon EC2 et dotCloud. Quand Jelastic devient indisponible, l'équilibreur de charge commute automatiquement sur Amazon EC2 ou dotCloud. Le service d'équilibrage de charge dans cette approche est répliqué dans différents nuages afin de garantir la disponibilité lorsqu'une panne se produit. Avec cette approche, la haute disponibilité est gérée à travers plusieurs nuages.

3.1.5 Synthèse

En outre, le développement et le déploiement d'applications orientées services dans un environnement *multi-nuages* ont des exigences particulières et spécifiques (la proximité en terme de latence réseau entre deux composants connexes, la possibilité d'exprimer des contraintes sur un composant, un composant constitue une unité d'exécution, le changement dans un composant n'oblige pas à redéployer toute l'application) en terme d'architecture, de méthodologies, de techniques, d'implantation et d'évolution des logiciels [Ghe02, Chh10].

3.2 Solutions *multi-nuages*

Afin de recueillir les exigences qui vont conduire à la conception des applications pour un environnement *multi-nuages* d'une part et d'autre part, la conception et le développement de l'architecture de la plateforme *soCloud*, les étapes suivantes ont été réalisées :

— Collecte et description des travaux pertinents dans la littérature.
— Analyse structurée de ces travaux pertinents.

Dans un premier temps, nous allons présenter de manière succincte les travaux de la littérature qui sont connexes à *soCloud*. Ensuite nous allons effectuer une étude comparative de ces travaux afin d'identifier les lacunes, les insuffisances, les besoins et les problèmes et donc de susciter des exigences.

La concurrence croissante entre les fournisseurs du marché de nuages, comme Amazon, Microsoft, Google et Salesforce où chacun défend ses propres standards et formats incompatibles [Mac09] accroît le nombre de solutions propriétaires. Ces fournisseurs de nuage sont incapables de prédire la répartition géographique des utilisateurs qui consomment les services. Il en est de même pour la gestion des pannes qui surviennent. Ainsi, la coordination de la charge et la gestion des pannes doivent se faire automatiquement et la distribution des services doit s'adapter à des changements dans la charge de travail. Selon la littérature existante, une approche pour contrer ces problèmes est la création d'un environnement *multi-nuages*, le nuage de nuages [K07, Buy10, Ber09] qui se réfère à des nuages interconnectés à l'échelle mondiale.

3.2.1 Les standards

Il existe quelques efforts de standardisation dans le domaine de l'informatique en nuage. Nous allons regrouper ces efforts en fonction des trois principaux modèles de services (SaaS, PaaS et IaaS).

3.2.1.1 SaaS

Lorsqu'on développe une application en nuage [1], l'absence d'adoption de standard entre les différents fournisseurs de nuage crée de la complexité qui nuit à la portabilité. Ainsi, la dépendance vis-à-vis d'un seul fournisseur de nuage devient inévitablement de plus en plus grande. Actuellement, les fournisseurs de nuage possèdent des modèles d'applications différents, dont beaucoup sont des modèles propriétaires qui limitent la personnalisation des ressources de la plateforme et de l'infrastructure sous-jacentes. Peu d'efforts sont fournis dans le soutien des outils, des techniques, des procédures ou des formats de données standards ou d'interfaces de services susceptibles de garantir la portabilité des données et des applications. En général, dans la situation actuelle, il est difficile pour les développeurs d'applications en nuage de déplacer leurs données et composants d'un fournisseur de nuage à un autre ou vers un environnement informatique en interne.

Le modèle Solution Deployment Descriptor (SDD) [OAS08] est un standard proposé par OASIS. Il définit un schéma XML pour décrire une unité d'installation de logiciel pertinente pour les aspects essentiels de son déploiement, la configuration et la maintenance. Les avantages de ce standard sont :

— La capacité à décrire des packages de solutions logicielles pour des environnements hétérogènes, à la fois simples et multiples plateformes.
— La capacité à décrire des packages de solutions logicielles indépendants de la technologie de l'installation du logiciel ou du fournisseur.
— La capacité à fournir les informations nécessaires pour permettre l'entretien du cycle de vie complet de solutions logicielles.

Par ailleurs, l'organisme IEEE propose Guide for Cloud Portability and Interoperability Profiles (CPIP) [IEE11] un méta standard avec des profils pour les nuages existants pour faciliter la portabilité et la gestion des applications. Le but de ce guide dans le développement des applications en nuage est d'assurer des solutions logicielles uniformes, cohérentes et de haute qualité.

3.2.1.2 PaaS

CAMP : Le projet OASIS Cloud Application Management for Platforms (CAMP) [OAS12a] développe des standards industriels pour gérer des applications des nuages privés et publics. CAMP définit des artefacts et des APIs qui doivent être offerts

1. Il s'agit d'une application déployée et qui s'exécute dans le nuage.

par un nuage PaaS pour gérer la construction, l'exploitation, l'administration, la surveillance et des correctifs des applications dans le nuage. Basé sur le style d'architecture REST, CAMP devrait favoriser un écosystème d'outils communs, des plugins, des bibliothèques et intergiciels qui permettra aux fournisseurs d'offrir une plus grande valeur ajoutée.

TOSCA : La topologie de l'orchestration des spécifications de OASIS pour les applications en nuages (Topology and Orchestration Specification for Cloud Applications) TOSCA [OAS12b] est un langage XML pour décrire des applications en nuages et leur régime de fonctionnement et de gestion. TOSCA standardise la description de la structure et les aspects de gestion (c'est-à-dire déploiement, opération et terminaison) [Bin13]. La portabilité est réalisée en utilisant le même plan ou en implémentant les artefacts. La portabilité des plans de TOSCA est héritée du langage de workflow et les moteurs utilisés.

3.2.1.3 IaaS

VMAN : Le standard de la gestion de la virtualisation (Virtualization Management) (VMAN) [DMT11] du Distributed Management Task Force (DMTF) a effectué une extension du standard DMTF pour la gestion de ressources réelles pour couvrir la gestion de ressources virtuelles. Ils comprennent le Open Virtualization Format (OVF) [For09] qui est un standard pour les formats d'images qui vont être chargées dans le IaaS en gérant le système, OVF représente la clé du standard de la portabilité.

CIMI : Cloud Infrastructure Management Interface (CIMI) [Dav12] est un groupe de travail au sein du consortium DMTF qui vise à gérer des ressources IaaS. Il implémente une interface REST et définit des APIs rendus aux formats XML et JSON. CIMI vise à fournir un support au standard OVF.

OCCI : De même, Open Grid Forum (OGF) [Edm12] est une autre initiative ouverte qui vise à la création d'une solution pratique pour interfacer les nuages de type IaaS existants. OGF a défini Open Cloud Computing Interface (OCCI) [Edm12] pour fournir un accès uniforme aux ressources IaaS existantes. L'objectif principal de OCCI est la création de nuage hybride exploitant les environnements de nuages indépendants des fournisseurs et des intergiciels de plateformes. Le principal formalisme utilisé pour définir des modèles de nuage est UML et le travail est encore à un stade préliminaire. OCCI définit des spécifications pour les éléments de base de nuage et des interfaces y compris un modèle utilisant l'API REST pour accéder, utiliser et gérer les nuages.

CDMI : Le Cloud Data Management Interface (CDMI) [Sli11] est un standard proposé par l'organisme Storage Networking Industry Association (SNIA). CDMI définit une API REST pour effectuer des opérations CRUD (Create Read Update Delete) sur des données à partir d'un environnement de stockage en nuage. Il définit également une API pour la découverte des stockages en nuage et la gestion des conteneurs de données.

Si nous regardons l'histoire de l'informatique, nous nous rappelons de l'importance de l'ouverture par rapport au coût élevé des plateformes propriétaires. Dans un monde parfait,

lorsqu'on développe une application en nuage, la meilleure façon pour lutter contre le coût élevé et la dépendance vis-à-vis d'un seul fournisseur est d'utiliser les standards ouverts. La plupart des efforts de standardisation effectués dans le domaine de l'informatique en nuage sont destinés aux niveaux IaaS et PaaS. En résumé, un tel idéal n'est pas encore adopté dans le jeune marché de nuages. Le problème est que la plupart des vendeurs rendent le processus d'adoption de standard difficile, frustrant et tellement long que certaines organisations abandonnent.

3.2.2 Vue d'ensemble des solutions existantes

Un très bon départ pour comprendre les grands enjeux dans l'informatique en nuage est l'article "Above the clouds : a Berkeley view of cloud computing" [Arm10]. Cet article a énuméré dix obstacles (*Disponibilité de service*, *Verrouillage de données*, *Confidentialité de données*, *Goulot d'étranglement lors du transfert de données*, *Performance imprédictible*, *Passage à l'échelle du stockage*, *Bugs dans le système distribué*, *Passage à l'échelle rapide*, *Partage de réputation*, *Licences des logiciels*) qui rendent l'adoption de l'informatique en nuage difficile. Dans cet article, les auteurs préconisent l'adoption de l'*informatique multi-nuages* pour adresser certains de ces obstacles (*Disponibilité de service*, *Verrouillage de données*, *Passage à l'échelle rapide*). Cette section présente et analyse certains travaux qui ont été effectués dans le domaine du *multi-nuages*. Cette analyse est faite sur la base de quatre critères : la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité*.

3.2.2.1 mOSAIC

Le projet européen[2] de recherche mOSAIC [Mos11] propose une API et une plateforme PaaS libre permettant de développer et déployer des applications pour un environnement *multi-nuages*. La construction d'une application déployée avec mOSAIC exige que cette dernière soit constituée de composants avec des dépendances explicites pour communiquer et échanger des données entre eux. En outre, l'architecture de l'application doit être de type SOA et doit utiliser uniquement l'API mOSAIC pour la communication. mOSAIC utilise uniquement la communication asynchrone. La figure 3.3 présente l'architecture de la plateforme mOSAIC [Pet11b]. Pour chaque composant de l'application, le développeur doit préciser les exigences en matière de ressources pour le traitement, le stockage et la communication. On peut spécifier certaines ressources pour l'ensemble de l'application. La plateforme mOSAIC déploie automatiquement les applications à travers les nuages sans l'intervention directe du développeur d'applications. Ainsi, la seule façon de contrôler l'application est d'utiliser les accords au niveau service (SLA) prédéfinis sur les indicateurs de performance au niveau composant ou application. L'approvisionnement des ressources dans la plateforme mOSAIC est faite par un courtier appelé courtier de ressources (*Resource Broker*) [Pet11b]. Il est chargé de la médiation entre les fournisseurs de nuage et le client. Le

2. http ://www.mosaic-cloud.eu

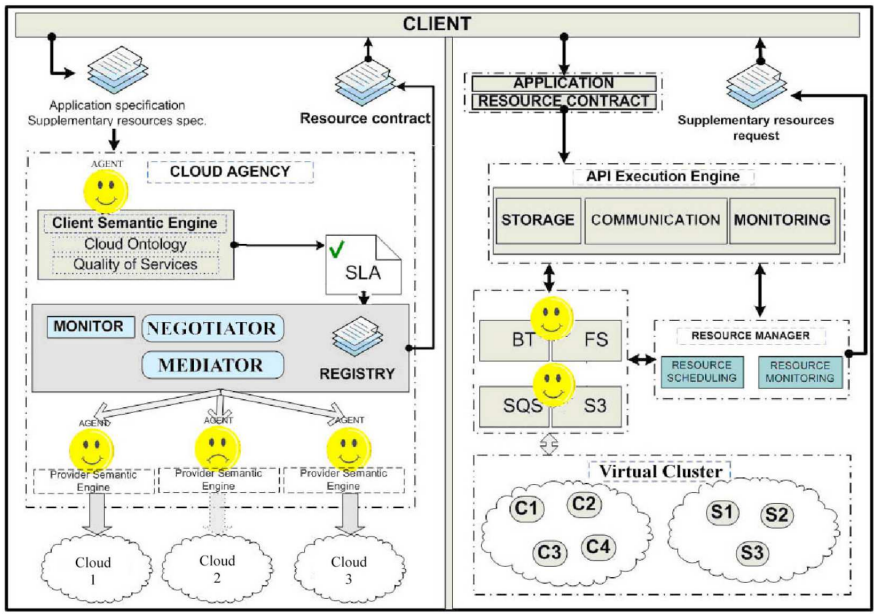

Figure 3.3 – Architecture de la plateforme mOSAIC [Mos11].

composant “Resource broker” est en outre composé d’une agence de nuage (*Cloud Agency*) qui est responsable de la découverte et de la négociation avec les fournisseurs de nuage. Le courtier de ressources est aussi composé d’une interface client (*Client Interface*) qui est responsable de demander des ressources supplémentaires en utilisant le composant exécuteur d’applications *Application Executor*. Le composant exécuteur d’applications gère le déploiement, l’exécution et la surveillance des applications.

```
1 {
2  "mHTTPgw":{
3          "type":"#mosaic-tests:exec",
4          "configuration":["/opt/mosaic-component-mhttpgw-0.2.1
5          /bin/mosaic-component-mhttpgw--run-mhttpgw",[]],
6          "annotation":null,
7          "count":1,
8          "order":4,
9          "delay":4000
10         },
11
12 "hello" : {
13         "type" : "#mosaic-components:java-cloudlet-container",
14         "configuration" : ["hello-cloudlet.properties",
15         "http://ip/cloudlets/hello.jar"],
16         "annotation":null,
17         "count" : 1,
18         "order" : 1,
19         "delay" : 0
20    }
21 }
```

Listing 3.1 – Le descripteur d'une application mOSAIC.

Le listing 3.1 présente un exemple de descripteur d'une application mOSAIC qui affiche hello world au format JSON. Dans ce fichier de description, nous avons deux composants mHTTPgw en lignes 2-10 et hello en lignes 12-20. Les lignes 4-5 et 15-16 décrivent les implémentations des deux composants. Le composant mHTTPgw permet d'instancier un serveur web et le composant hello correspond à la logique métier de l'application. Les fichiers de configuration en lignes 4 et 14 contiennent les classes de Context et de gestion de cycle de vie de l'application.

3.2.2.2 RESERVOIR

Le projet européen[3] de recherche RESERVOIR [Roc09] vise à développer une infrastructure orientée services permettant l'interopérabilité dynamique entre les fournisseurs de nuage. RESERVOIR sépare le rôle du fournisseur de service de celui du fournisseur d'infrastructure. En effet, le fournisseur de service interagit avec les utilisateurs finaux et répond à leur besoins. Les fournisseurs de service ne possèdent pas les unités de calcul mais les louent auprès des fournisseurs d'infrastructures qui interagissent les uns avec les autres de manière transparente pour créer des pools de ressources. La figure 3.4 illustre une architecture de gestion des sites RESERVOIR. Le site RESERVOIR est constitué de trois composants :

3. http ://www.reservoir-fp7.eu/

1. **Service Manager** (SM) est responsable d'instancier l'application en demandant la création et la configuration d'environnements d'exécution virtuels (virtual execution environments) ou VEEs pour chaque composant dans le Manifest. En outre, le SM est responsable d'assurer la conformité des SLAs en suivant l'exécution des applications et exécute les règles d'élasticité.
2. **Virtual Execution Environment Manager** (VEEM) est responsable du placement des VEEs dans les hôtes. Il permet de fédérer les sites RESERVOIR distants. De plus, il gère le pool de ressources.
3. **Virtual Execution Environment Host** (VEEH) représente une ressource virtualisée qui peut accueillir un certains type de VEEs. Par exemple, un VEE peut être une machine physique avec un hyperviseur pour le contrôler.

Les fournisseurs d'infrastructures exploitent des sites RESERVOIR qui possèdent et gèrent l'infrastructure physique sur laquelle les applications sont exécutées. La fédération des sites RESERVOIR qui collaborent forme un nuage de RESERVOIR. En outre, les fournisseurs de service utilisent les Manifests pour décrire leurs besoins.

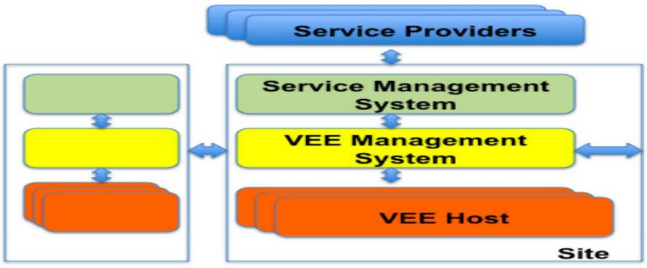

Figure 3.4 – Architecture de gestion des sites RESERVOIR [Roc09].

3.2.2.3 Cloud4SOA

Le projet européen [4] de recherche Cloud4SOA propose une plateforme PaaS qui fournit une solution pour développer et gérer des applications pour un environnement *multi-nuages*. Il introduit l'interopérabilité sémantique à trois dimensions qui vise à capturer n'importe quel type de conflit d'interopérabilité survenant au niveau de la couche PaaS. Il permet d'interconnecter plusieurs PaaS hétérogènes en utilisant le concept de l'adaptateur. La figure 3.5 illustre l'architecture de Cloud4SOA qui est constituée de 5 couches :

1. **Service Front-end layer.** Cette couche représente le portail d'accès à l'ensemble des fonctionnalités de Cloud4SOA.

4. http ://www.cloud4soa.eu/

2. **Semantic layer.** Cette couche met en œuvre trois principaux composants qui couvrent l'ensemble de l'architecture de l'interopérabilité sémantique. Cette couche se place entre les différents PaaS pour assurer l'interopérabilité.
3. **SOA layer.** Elle est constituée d'une boîte à outil accessible à travers le service front-end. On utilise les annotations pour exprimer des fonctionnalités comme la découverte, la publication de services et la migration.
4. **Gouvernance layer.** Elle permet la gestion du cycle de vie de l'application en nuage en prenant en compte les informations de surveillance, les SLAs et les problèmes de passage à l'échelle.
5. **Distributed repository layer.** Cette couche fournit des adaptateurs pour interagir avec les plateformes sous-jacentes. Elle est l'intermédiaire entre la plateforme Cloud4SOA et les autres plateformes sous-jacentes.

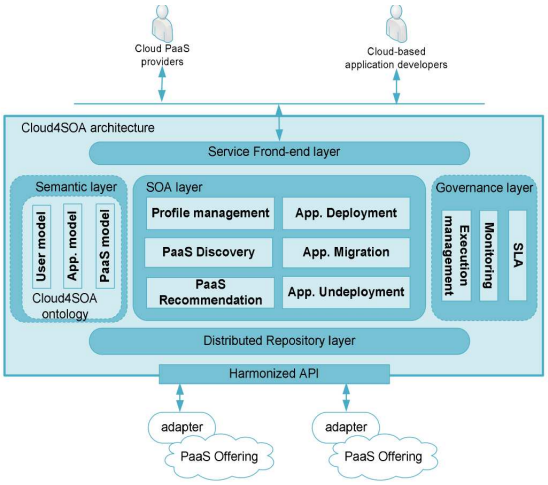

Figure 3.5 – Architecture de référence Cloud4SOA [Kam13].

De plus, la plateforme Cloud4SOA fournit quatre fonctionnalités principales :

1. **Semantic Matchmaking** permet de rechercher parmi les offres de PaaS exitantes, celles qui correspondent le mieux aux besoins du développeur.
2. **Cross-platform application management** gére les applications à travers plusieurs nuages et prend en charge le déploiement et la gouvernance de celles-ci dans l'environnement PaaS de manière indépendante.
3. **Application migration** fournit un certain nombre de fonctionnalités permettant la migration d'une application d'un PaaS à l'autre.
4. **Monitoring** offre un mécanisme unifié pour surveiller la performance et la santé des applications hébergées dans un environnement *multi-nuages*.

3.2.2.4 REMICS

Le projet européen [5] de recherche REMICS (REuse and Migration of legacy applications to Interoperable Cloud Services) [Moh11] a pour objectif principal de développer un ensemble de méthodes dirigées par les modèles et outils qui supportent l'organisation des systèmes patrimoniaux afin de les moderniser selon le paradigme "service de nuage". La figure 3.6 illustre l'approche utilisée par REMICS pour faire la migration. REMICS utilise le concept de base de l'architecture dirigée par la modernisation (ADM) [Bab11] de l'organisme OMG. Dans ce concept, la modernisation commence par l'extraction de l'architecture de l'application patrimoniale. Le modèle architectural permet d'analyser le système existant, d'identifier les meilleurs moyens pour moderniser et profiter des technologies d'ingénieries dirigés par les modèles (MDE) pour produire un nouveau système. Cette information est ensuite traduite dans le modèle qui couvre les différents aspects de l'architecture.

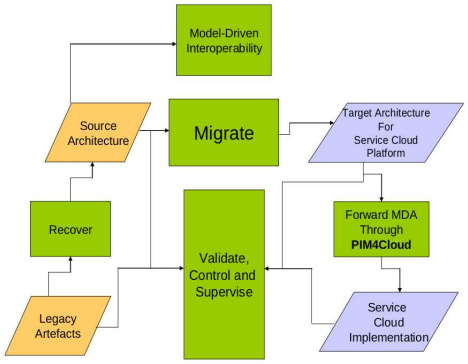

Figure 3.6 – REMICS : approche pour la migration [Moh11].

La migration des applications patrimoniaux se déroule en 7 phases :

1. **Etablissement du contexte.** Cette première étape de la migration permet de décider s'il est judicieux de migrer une application patrimoniale vers les nuages.
2. **Modernisation de l'architecture logiciel.** Dans cette partie, REMICS utilise l'architecture orientée services pour intégrer l'application.
3. **Modernisation des données.** Les applications patrimoniales dans REMICS utilisent des données et la sécurité de celles-ci est un aspect important.
4. **Gestion des besoins non-fonctionnels dans le nuage.** Les besoins en terme de ressources matérielles, sécurité, latence devraient être étudiés en amont.
5. **Vérification et validation dans le nuage.** L'intégration, le développement de nouvelles méthodes pour prédire les performances et la qualité des applications dans le nuage sont abordés.

5. http ://www.remics.eu/

6. **Introduction de l'agilité dans le processus de migration.** Dans cette étape, REMICS fournit une méthodologie agile d'ingénierie logicielle dirigée par les modèles pour supporter la migration vers le nuage.
7. **Nouveau modèle économique** L'informatique en nuage impose un nouveau modèle économique basé sur "payez uniquement ce que vous consommez".

3.2.2.5 PaaSage

Le projet européen [6] de recherche PaaSage [Paa13] Model-based Cloud Platform Upperware vise à créer une plateforme pour développer et déployer des applications avec une méthodologie appropriée afin d'aider les développeurs à implanter de nouvelles applications pour un environnement *multi-nuages*. Le projet a aussi pour but de migrer des applications existantes vers l'environnement *multi-nuages*. La figure 3.7 présente une vue d'ensemble de l'intégration et de la conception des différents composants du projet. L'architecture globale de PaaSage peut être regroupée en trois principaux composants [Paa13] :

1. **Integrated Development Environment** (IDE). Ce composant est le portail de la plateforme avec lequel les utilisateurs interagissent. Il est basé sur Eclipse et prend en charge la modélisation Application Modelling Execution Language (CAMEL) et CLOUDML [Fer13b, Fer13a].
2. **Upperware**. Ce composant est intégré avec l'IDE et expose via ce dernier un ensemble d'outils pour capturer les besoins lors de la modélisation et la conception. Le cloud Reasoner et le cloud Adapter constituent les principaux éléments de cette couche.
3. **Executionware**. Ce composant fournit des mappings et techniques d'intégration spécifiques pour intégrer la plateforme PaaSage aux APIs des infrastructures de nuages.

La méthodologie du modèle PaaSage est basée sur le cycle de vie de l'application en nuage. Ces phases utilisent le modèle en cascade (Water Fall model) [Boe88] pour le développement logiciel avec le mapping suivant : *phase de configuration* (besoins, conception), *phase de déploiement* (implémentation) et la phase d'exécution (vérification, maintenance). Le modèle PaaSage peut être scindé en trois parties :

1. **Modèle de configuration**. Ce modèle peut être créé en utilisant UML, mais les exigences du modèle sont traduites par le composant en des langages que le cloud Reasoner peut comprendre.
2. **Modèle de déploiement**. Ce modèle utilise les exigences et les données provenant des sources telles que les bases de méta-données. Ces exigences sont ensuite utilisées pour faire le déploiement.
3. **Modèle d'exécution**. Ce modèle utilise les données de surveillance provenant des infrastructures et services comme support pour l'exécution.

6. http ://www.paasage.eu/

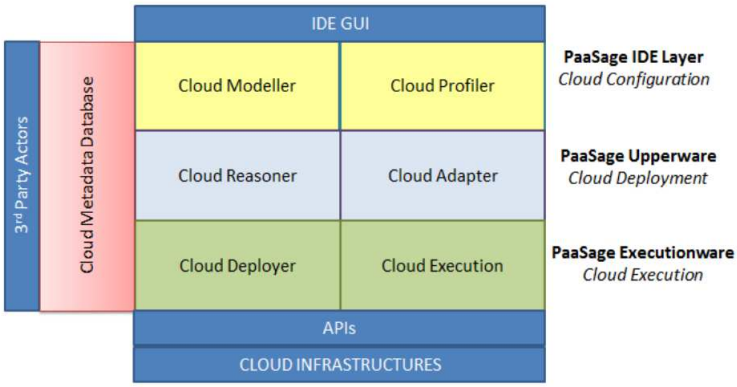

Figure 3.7 – Architecture de PaaSage [Paa13].

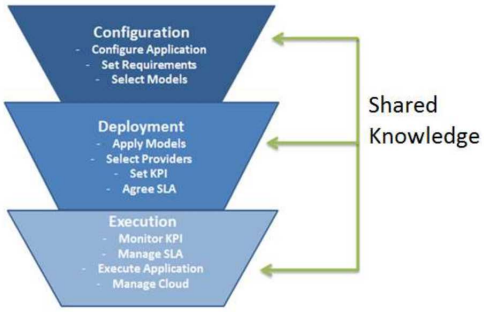

Figure 3.8 – Gestion de cycle de vie de l'application (PaaSage) [Paa13].

3.2.2.6 MODAClouds

Le projet européen [7] de recherche MODAClouds [MOD13] a pour objectif de fournir des méthodes, un système d'aide à la décision, un IDE libre et un environnement d'exécution pour la conception de haut niveau et le déploiement automatique d'applications dans un environnement *multi-nuages* tout en garantissant la qualité de service. En outre, MODAClouds fournit des techniques pour la synchronisation des données à travers plusieurs nuages.

La figure 3.9 montre les principaux composants de la plateforme MODAClouds :

1. **MODAClouds IDE.** L'IDE offre toutes les fonctionnalités pour gérer le cycle de vie de développement d'une application en nuage à partir de l'évaluation des risques et des coûts pour effectuer le déploiement.

7. http ://www.modaclouds.eu/

2. **Monitoring platform.** Ce composant est responsable de la collecte de données des métriques de l'application en cours d'exécution et l'infrastructure sous-jacente, afin d'analyser et de définir la qualité de service, la performance, etc.
3. **Self-adaptation platform.** Ce composant est responsable du suivi de l'état de l'application déployée et gère des changements dans la configuration de l'application en nuage pour satisfaire les contraintes de performance.
4. **Execution platform.** Ce composant approvisionne tous les services nécessaires pour le déploiement à la fois des applications et de la plateforme de surveillance correspondante.

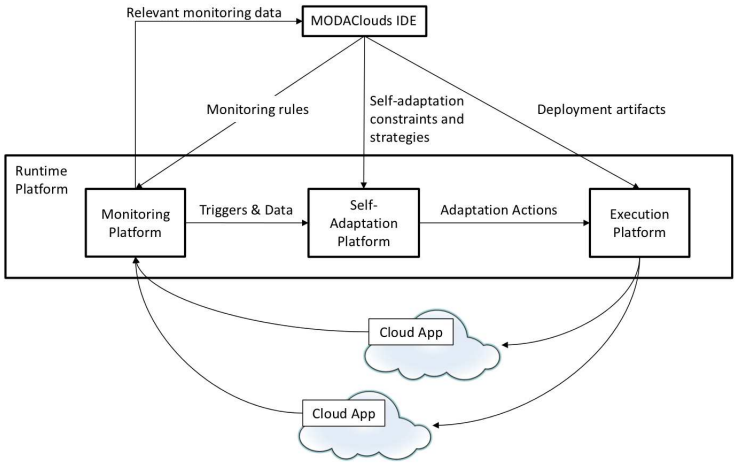

Figure 3.9 – MODAClouds : architecture de haut-niveau [MOD13].

3.2.2.7 4CaaSt

Le projet européen[8] de recherche 4Casst [Gar12] vise à créer une plateforme PaaS et prendre en charge l'hébergement optimisé et l'élasticité d'applications n-tiers. 4CaaSt ajoute des supports pour la génération automatique et l'exécution de différents mécanismes d'élasticité et de déploiement (voir figure 3.10).

Le processus de fonctionnement de l'approche 4CaaSt se déroule comme suit :

1. **Blueprint creation and resolution.** Les composants 4CaaSt doivent contenir toutes les informations (contraintes d'élasticité, indicateur de performances, configuration) pour effectuer la génération et le déploiement à l'exécution.
2. **Deployment Design Generation.** L'étape de la génération du plan de déploiement est effectuée en tenant compte des exigences spécifiques de clients qui sont gérées par ce composant.

8. http ://www.4caast.eu/

3. **Resource Provisioning.** La mise en œuvre du déploiement est réalisée en utilisant une extension du standard OVF [For09].
4. **Service Operation and Adaptation.** Une fois le déploiement effectué, les services s'enregistrent sur l'infrastructure d'exécution 4CaaSt qui prend en charge la surveillance, la compatibilité, ainsi que le passage à l'échelle en utilisant les règles d'élasticité prédéfinies.

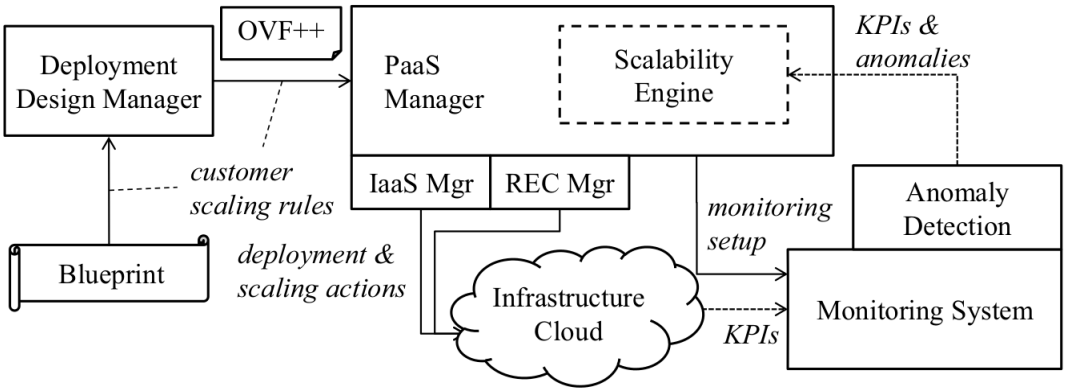

Figure 3.10 – 4CaaSt : Architecture de déploiement de PaaS et de la gestion d'élasticité [Gar12].

3.2.2.8 CONTRAIL

Le projet européen [9] de recherche CONTRAIL propose une plateforme qui vise à mettre en œuvre une infrastructure IaaS et une plateforme PaaS. Il permet aux fournisseurs de nuages d'intégrer de manière transparente les ressources d'autres nuages avec leur propre infrastructure pour fédérer plusieurs nuages en favorisant la portabilité des applications d'un nuage à l'autre.

Dans l'architecture CONTRAIL, on peut distinger trois couches (voir figure 3.11) :

1. **Federation.** Cette couche fédère plusieurs nuages. Elle dispose d'une API REST pour gérer les données, les applications, les utilisateurs et les fournisseurs. Elle utilise XtreemFS comme système de stockage à travers plusieurs nuages.
2. **Provider.** Cette couche gère le déploiement d'applications à travers plusieurs nuages. Le mécanisme de déploiement est basé sur le standard OVF [For09] utilisé pour décrire les applications réparties.
3. **Resource.** Cette couche agrège les informations sur les ressources de nuage.

9. http ://contrail-project.eu/

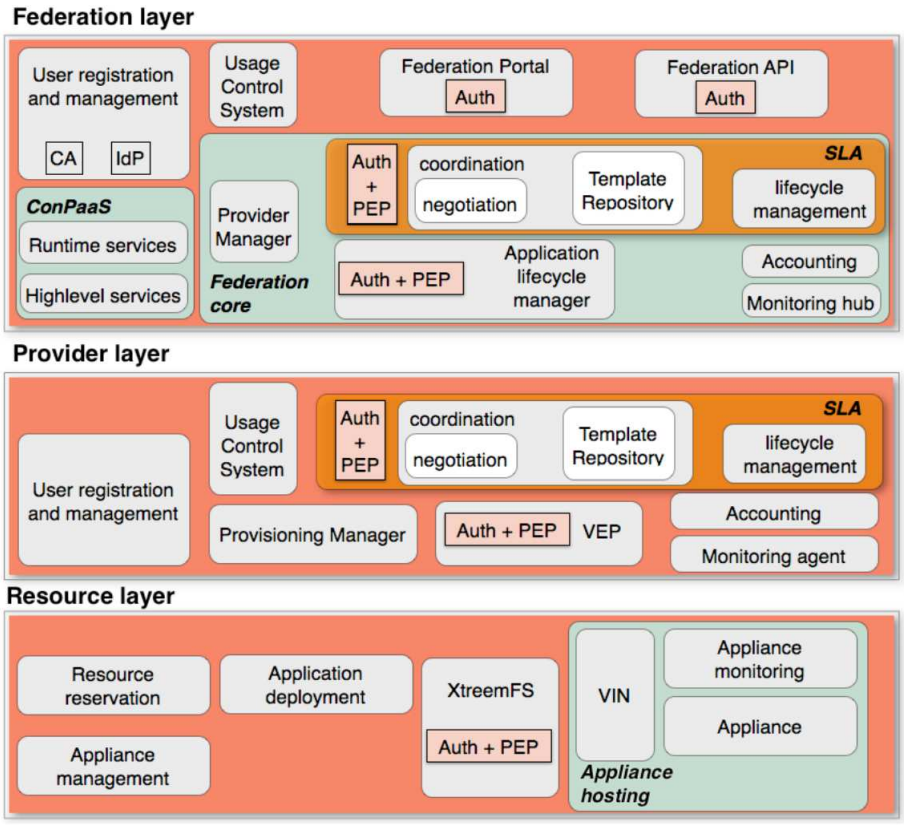

Figure 3.11 – CONTRAIL : Architecture [Car12].

ConPaaS [Pie12] est la plateforme *multi-nuages* du projet CONTRAIL. Elle fournit un environnement d'exécution et le mécanisme d'élasticité aux applications. La figure 3.12 illustre l'architecture la plateforme ConPaaS. On peut distinguer deux types de services (PHP et MYSQL) qui hébergent les applications. Les front-ends représentent le portail d'accès à la plateforme pour faire de l'administration.

3.2.2.9 Aneka

Aneka [Vec09] est une plateforme pour développer et déployer des applications réparties. Elle fournit un environnement d'exécution et un ensemble d'APIs qui permettent aux développeurs de créer des applications .NET pour un nuage hybride. Aneka est un projet industriel de Manjrasoft [10].

La figure 3.13 illustre l'architecture de la plateforme Aneka. Celle-ci est scindée en trois parties :

10. http ://www.manjrasoft.com

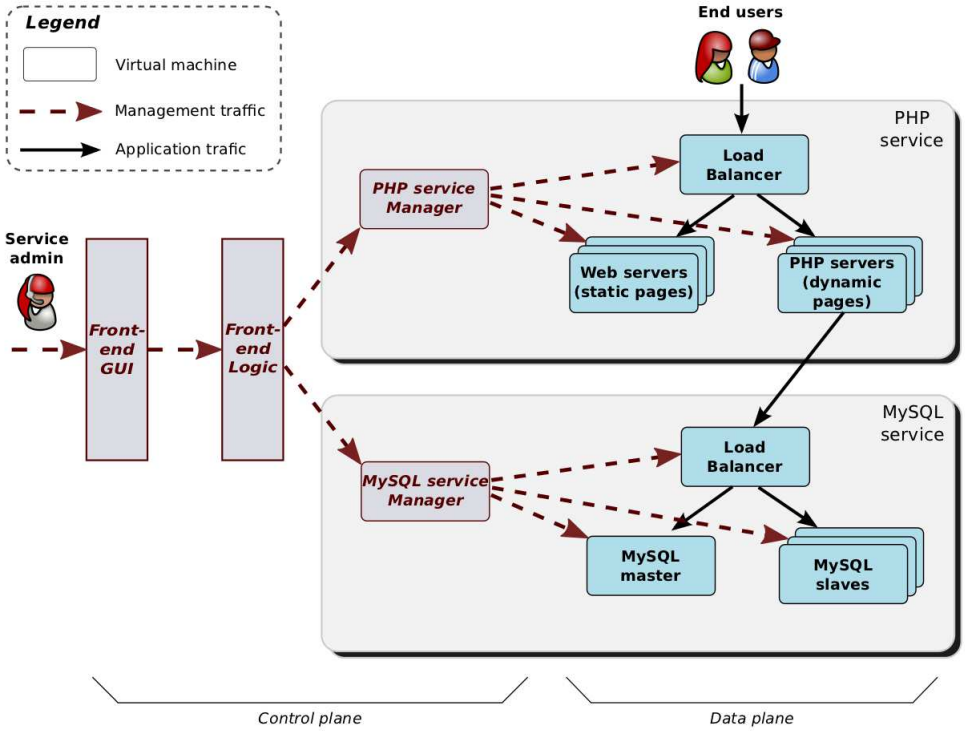

Figure 3.12 – Architecture de la plateforme ConPaaS [Pie12].

1. **Infrastructure.** C'est une collection de ressources physiques et virtuelles connectées à travers le réseau. Chaque ressource héberge une instance du conteneur Aneka.
2. **Middleware : container.** Le conteneur représente l'environnement d'exécution sur lequel les applications réparties sont exécutées. Il fournit une fonction basique de gestion d'un seul nœud sur lequel sont hébergées les applications.
3. **Application : développement and management.** Cette partie fournit différents types de composants et outils. Par exemple, elle permet de faire le développement et la gestion d'applications de nuage et la migration d'applications existantes vers le nuage.

Aneka fournit un modèle pour développer les applications en nuage. Ce modèle définit les propriétés et exigences d'une application répartie déployée sur Aneka. La figure 3.14 illustre les éléments clés du modèle d'applications Aneka. On peut noter que ce modèle d'application gère une application comme un ensemble d'unités d'exécution. Chaque modèle de programmation a une implémentation spécifique à l'unité d'exécution et le type de gestionnaire (manager) pour l'application. Le modèle de programmation représente la façon de définir une application répartie avec Aneka. Aneka utilise les modèles de programmation à base de *tâche*, *thread* et *MapReduce*. L'unité d'exécution représente le composant de base pour l'exécution d'une application distribuée, tandis que le gestionnaire d'applications est un composant interne qui est utilisé pour soumettre l'unité d'exécution à la plateforme

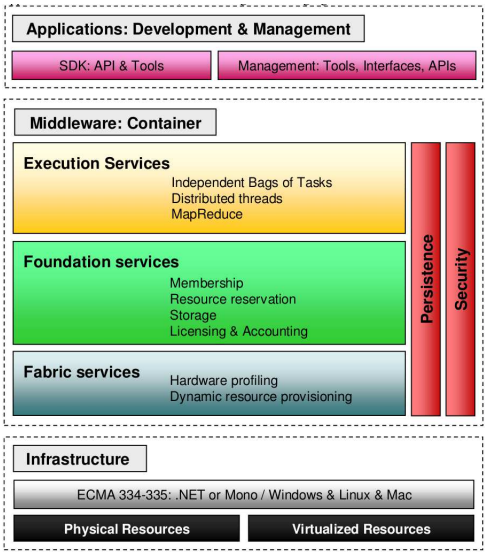

Figure 3.13 – Architecture de la plateforme Aneka [Vec09].

Aneka.

3.2.2.10 STRATOS

STRATOS [Paw12] est un projet de recherche supporté par le Engineering Research Council of Canada (NSERC), Ontario Centre of Excellence (OCE) et Amazon Web Service. Le but de ce projet est de fournir un courtier de nuage (Cloud Broker) au niveau IaaS permettant d'approvisionner des ressources provenant de plusieurs nuages. Il supporte le déploiement et la gestion des applications à l'exécution. STRATOS permet de déployer des applications, tout en définissant sur ces dernières des exigences qui sont importantes en terme d'indicateurs de performance (Key Performance Indicators) ou KPI. Ces exigences sont transformées en un ensemble de requêtes de demande de ressources qui sont évaluées en fonction de l'offre des fournisseurs de nuage, afin de sélectionner la meilleure offre. La figure 3.15 présente l'architecture de STRATOS. Dans cette architecture, le document de topologie nommé Topology Descriptor File (TDF) est utilisé pour définir la topologie de l'application qui doit être déployée dans le nuage. Le module de déploiement spécifie tous les détails sur l'environnement de l'application dans ce document. Lors de la réception du document de la topologie, le gestionnaire de nuage (Cloud Manager) contacte le courtier (Broker) pour l'instanciation de la topologie. Le courtier effectue des calculs préliminaires afin de déterminer la façon la plus efficace pour allouer des ressources à travers plusieurs fournisseurs de nuage. Le gestionnaire de nuage (Cloud Manager) et le courtier utilisent les informations de surveillances pour prendre la décision d'élasticité.

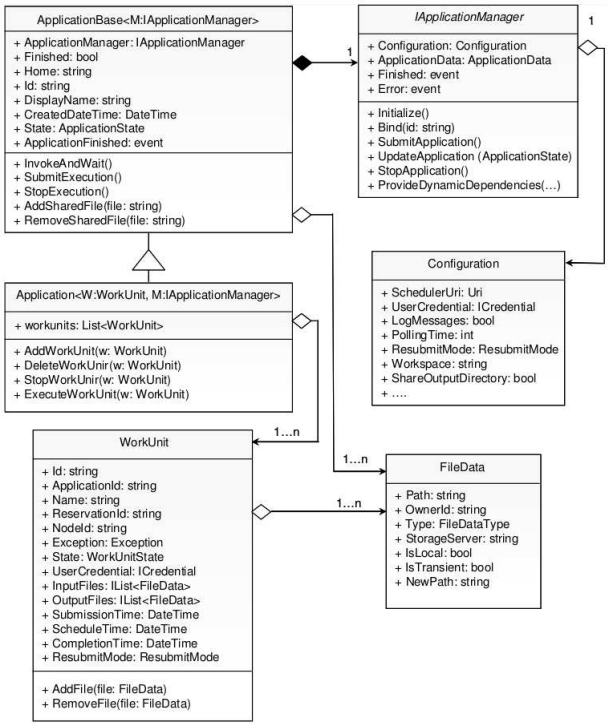

Figure 3.14 – Modèle d'applications Aneka [Vec09].

3.2.2.11 CompatibleOne

CompatibleOne[11] est un projet libre initié entre l'industrie et la recherche. CompatibleOne est un courtier de nuage (Cloud Broker) qui fournit un intergiciel interopérable permettant de décrire et de fédérer des ressources provenant de nuages hétérogènes. Il permet aux développeurs de combiner différents services provenant de plusieurs fournisseurs. Il supporte différents types de ressources qui peuvent être fournies aux niveaux IaaS et PaaS. CompatibleOne [Yan13] est une plateforme d'exécution qui fournit un modèle permettant de décrire les ressources. Ce modèle qui est nommé CompatibleOne Resource Description System (CORDS), permet de décrire une application à base d'objets. La plateforme d'exécution nommée Advanced Capabilities for CORDS (ACCORDS) est un système pour approvisionner et déployer des applications en nuage. La plateforme CompatibleOne est basée sur deux Standards CDMI et OCCI. Elle utilise OCCI comme modèle de communication via REST et CDMI pour l'accès aux données de stockage. La figure 3.16 illustre l'architecture de la plateforme CompatibleOne qui est scindée selon les quatre étapes suivantes :

1. **Gestion des besoins utilisateurs.** A cette étape, l'utilisateur spécifie ses besoins en ressources au niveau IaaS ou/et PaaS. Les besoins exprimés sont utilisés pour créer

11. http ://www.compatibleone.org

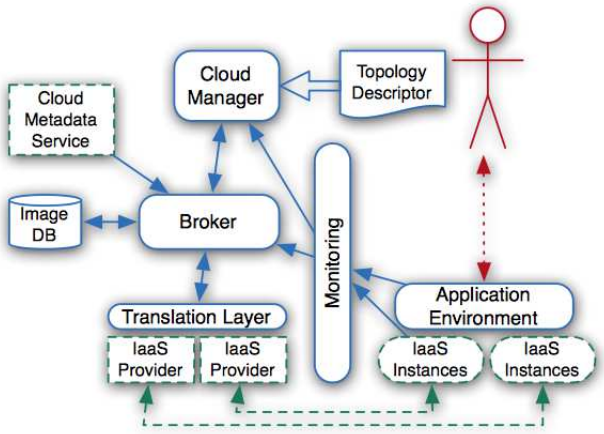

Figure 3.15 – Architecture de STRATOS [Paw12].

des manifests. Le gestionnaire des accords au niveau service (Service Level Agreement Manager ou SLAM) utilise le modèle OCCI pour gérer la négociation des accords.

2. **Validation et plan d'approvisionnement.** A la deuxième étape, le manifest créé est transféré au parseur de l'ACCORDS qui analyse et valide le document. Le parseur construit un plan d'approvisionnement qui correspond aux besoins exprimés. Le parseur interagit aussi avec le publisher de l'ACCORDS, afin de déterminer s'il y a un fournisseur de nuage qui satisfait les exigences requises.
3. **Exécution et approvisionnement de plan.** Le courtier fournit les services nécessaires pour sélectionner les fournisseurs de nuage appropriés dans le but de satisfaire les contraintes du plan d'approvisionnement.
4. **Fourniture de services de nuage.** A la quatrième étape, le courtier ACCORDS interagit avec les proxies afin d'obtenir les contrats d'approvisionnement.

Le modèle CORDS est utilisé pour décrire des ressources aux niveaux IaaS et PaaS. CORDS est un modèle à base d'objets qui utilise le standard OCCI. Les figures 3.17 et 3.18 représentent respectivement les descriptions des modèles de données logiques au niveau IaaS et PaaS.

3.2.2.12 OPTIMIS

Le projet européen [12] de recherche OPTIMIS [Fer12] a pour objectif de fournir une plateforme ouverte, évolutive et fiable pour la fourniture de services flexibles. Il fournit des services auto-adaptatifs et d'infrastructures non seulement basés sur la performance, mais

12. http ://www.optimis-project.eu/

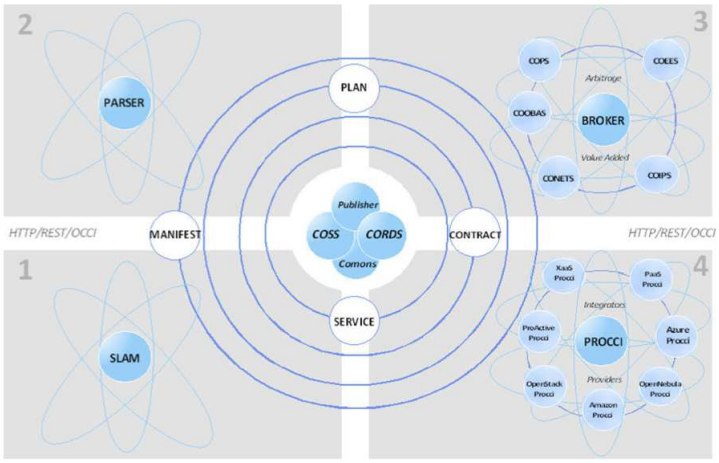

Figure 3.16 – Architecture CompatibleOne [Yan13].

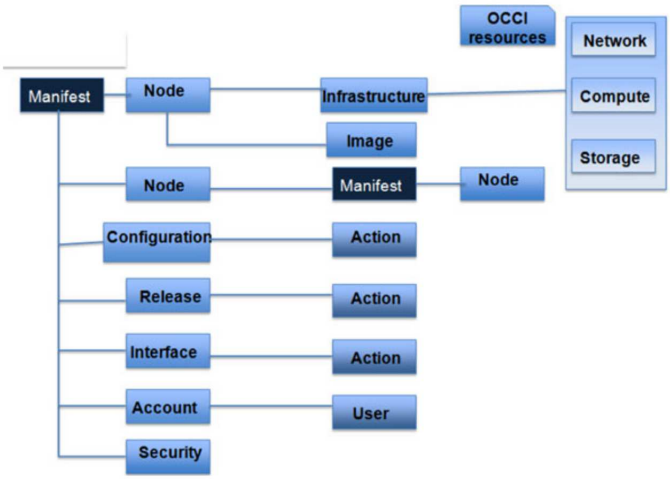

Figure 3.17 – CORDS : modèle de données logique au niveau IaaS [Yan13].

aussi sur les aspects tels que la confiance, le risque et le coût. Un des principaux résultats du projet OPTIMIS est sa boîte à outils qui est constituée d'un ensemble de logiciels qui peuvent être assemblés de manière arbitraire et donc permettre la construction de diverses architectures de nuage. Il s'agit notamment des architectures pour fédérer plusieurs nuages et les architectures *multi-nuages*. La boîte à outil OPTIMIS peut être regroupée en trois ensembles principaux :

1. **Boîte à outil basique.** C'est un ensemble d'outils fondamentaux pour la surveillance et l'évaluation des services de nuages et infrastructures, ainsi que leurs interconnexions sécurisées.

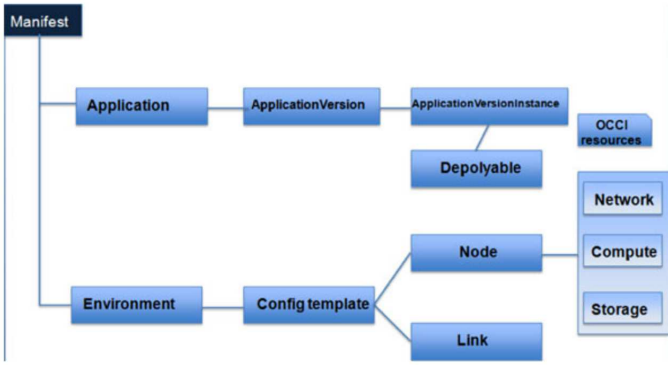

Figure 3.18 – CORDS : modèle de données logique au niveau PaaS [Yan13].

2. **Outils des fournisseurs d'infrastructures.** Il permet à un fournisseur d'infrastructure OPTIMIS d'optimiser la fourniture des services en fonction des accords sur les objectifs métiers (Business Level Objectives).
3. **Outils de fournisseurs de services.** Cet outil permet aux fournisseurs de service OPTIMIS de créer, de déployer et d'exécuter des services.

Le modèle de programmation défini dans OPTIMIS a pour objectif principal de permettre l'orchestration des compositions de services et méthodes à distance et de fournir ces compositions comme un nouveau service. Le modèle de programmation d'OPTIMIS est basé sur COMPS [Tej08] qui implémente un modèle de programmation pour développer des applications en utilisant le mécanisme de parallélisme de manière implicite pour des environnements répartis.

3.2.2.13 Cloud Foundry

Cloud Foundry [Fou11] est un projet libre développé par VMWARE qui vise à fournir les fonctionnalités de PaaS. Il permet d'orchestrer plusieurs insfrastructures de nuage. Il assure le déploiement et la gestion des applications dans les infrastructures de nuage sous-jacentes. La figure 3.19 illustre les cinq principaux composants de l'architecture de Cloud Foundry :

1. **Router.** Toutes les requêtes qui entrent dans la plateforme passent par ce composant. Il redirige les requêtes vers les applications.
2. **CloudController.** Ce composant gère les interactions avec les utilisateurs, les applications et les services. Il est responsable de la gestion des états de transitions.
3. **DEA.** Il est responsable d'exécuter toutes les applications déployées sur la plateforme. Il surveille les métriques telles que celles associées au CPU, à la mémoire, au disque, au réseau.

4. **HealthManager.** Il surveille les états de la plateforme. Il analyse chaque métrique surveillée afin de déterminer des incohérences.
5. **Messaging System.** Tous les messages qui transitent dans la plateforme passent par le système de message. Ce composant utilise le mécanisme de découverte automatique.

En plus, la plateforme Cloud Foundry permet de développer des applications en utilisant plusieurs langages de programmation et supporte plusieurs intergiciels et services.

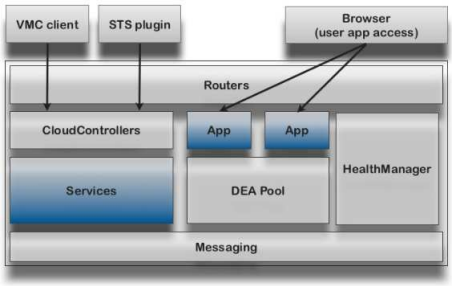

Figure 3.19 – Architecture de la plateforme Cloud Foundry [Fou11].

3.2.2.14 Cloudify

Cloudify [Gig12] est une plateforme libre développée par GigaSpace qui permet de déployer, de gérer et de mettre à l'échelle non seulement des applications existantes, mais aussi de nouvelles applications pour un environnement de nuage sans changer, ni l'architecture, ni le code source de celles-ci. L'architecture de la plateforme Cloudify (voir figure 3.20) est composée de trois couches :

1. **Recipe-based configuration.** Au cœur de Cloudify, elle est une sorte de manuel qui définit tous les détails nécessaires pour exécuter une application.
2. **Universal Service Adapter.** Cette couche s'exécute sur chaque nœud et traduit les recettes en actions sur le nœud local (par exemple, l'installation, l'initialisation du service, la surveillance du service).
3. **Pluggable cluster-aware monitoring.** Cet adaptateur active le plug-in localement sur chaque nœud de service, ainsi, le plug-in se connecte au service local et commence la collecte des métriques de ce dernier.

Cloudify fournit un modèle pour décrire une application. Le modèle définit un descripteur d'application qui décrit le nom de l'application, les services dont il a besoin et leur dépendances. Le listing 3.2 montre un exemple de fichier de description d'une application Cloudify. Le nom de l'application en ligne 1 est utilisé pour identifier cette dernière par les différents outils de gestion et de surveillance. Les lignes 14 et 19 permettent d'exprimer les

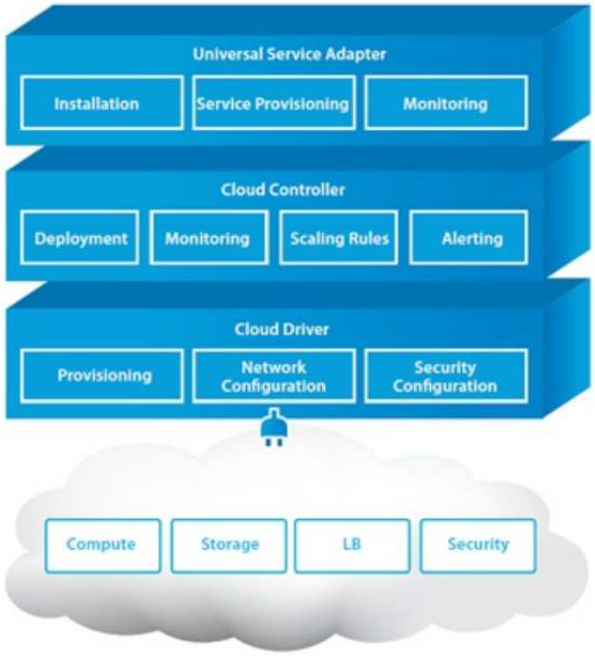

Figure 3.20 – Architecture de la plateforme Cloudify [Gig12].

dépendances entre les services. Par exemple, à la ligne 19, le service *Tomcat* dépend du service *mongos*.

```
1          application {
2          name="petclinic-mongo"
3
4          service {
5                  name = "mongod"
6          }
7
8          service {
9                  name = "mongoConfig"
10         }
11
12         service {
13                 name = "mongos"
14                 dependsOn = ["mongoConfig", "mongod"]
15         }
16
17         service {
18                 name = "tomcat"
19                 dependsOn = ["mongos"]
20         }
21 }
```

Listing 3.2 – Le descripteur d'une application Cloudify.

Le listing 3.3 est un exemple de règle d'élasticité qui ajoute une instance de Tomcat, lorsque cette dernière reçoit une requête par seconde.

```
service {
        name "tomcat"
        ...
        elastic true

        numInstances 1

        minAllowedInstances 1

        maxAllowedInstances 2

        scalingRules ([
        scalingRule {
                serviceStatistics {
                        metric "Total Requests Count"
                        movingTimeRangeInSeconds 20
                        statistics Statistics.maximumThroughput
                }

        highThreshold {
                value 1
                instancesIncrease 1
        }
        lowThreshold {
                value 0.2
                instancesDecrease 1
        }
   }
  ])
}
```

Listing 3.3 – Le descripteur d'une application Cloudify.

3.2.2.15 EASI-CLOUDS

Le nouveau projet européen [13] de recherche EASI-CLOUDS [Ema13] a pour objectif de fournir une infrastructure *multi-nuages*. Cette infrastructure comprend les trois modèles de services IaaS, PaaS et SaaS en plus de la fiabilité, l'élasticité et la sécurité.

Cette infrastructure comprendra un module pour :

13. http ://easi-clouds.eu

1. La standardisation d'interface permettant la portabilité d'applications à travers les nuages.
2. La composition et l'orchestration de services.
3. L'interopérabilité et la fédération de nuage.
4. La gestion des accords au niveau des services fournis (SLA) qui sera facilitée.
5. La gestion de l'approvisionnement, politique de prix de service utilisés.

3.2.3 Discussion

Dans cette section, nous présentons et analysons les travaux de l'état de l'art en tenant compte uniquement de quatre critères à savoir : la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité multi-nuages*.

3.2.3.1 Portabilité *multi-nuages*

L'approche de la portabilité peut être classée en trois catégories [Obe10] : *portabilité fonctionnelle (functional portability)*, *portabilité de données (data portability)* et la *portabilité d'applications (service enhancement)*. mOSAIC [Pet13b] fournit une solution pour la portabilité d'applications et de données au niveau IaaS et PaaS. Le projet Cloud4SOA [Dan12] a travaillé sur la portabilité des applications et données au niveau PaaS en utilisant une approche sémantique. MODAClouds [Ard12] assure la portabilité en implémentant le standard TOSCA de OASIS [OAS12b]. Cloud Foundry [Fou11], Cloudify [Gig12] assurent la portabilité des applications à travers plusieurs nuages. CompatibleOne [Yan13] assure la portabilité des données en utilisant le standard CDMI. Comparées aux deux solutions mOSAIC et Cloud4SOA, les autres solutions à savoir Cloud Foundry, Cloudify, MODAClouds et CompatibleOne assurent uniquement la portabilité des applications. MODAClouds et CompatibleOne utilisent des standards pour assurer la portabilité respectivement des applications et des données. Effectivement, l'utilisation du standard est la meilleure façon d'adresser le problème de la portabilité. Toutefois, l'adoption de standards est quasi inexistante et rendue très difficile voire même contraignante par les fournisseurs de nuages.

Les auteurs de l'article [Ame10, Man12] focalisent leurs travaux sur l'intégration de différents types d'environnements informatiques. Ils proposent un intergiciel de haut niveau à base de composants légers destinés à simplifier la transition des grilles vers les nuages. Ainsi, les applications sont facilement portables de la grille vers le nuage. Toutefois, leur approche se focalise essentiellement sur la portabilité d'applications.

3.2.3.2 Approvisionnement *multi-nuages*

Il existe un certain nombre de travaux sur l'approvisionnement dynamique de ressources matérielles (machines physiques ou machines virtuelles) et les grappes (cluster) de

machines [Ane10]. Le travail sur l'approvisionnement dynamique de ressources dans l'informatique en nuage peut être classifié en deux catégories. Les auteurs de l'article [Mie08] ont adressé le problème d'approvisionnement de ressources à la granularité des machines virtuelles. D'autres auteurs [Cha01] ont considéré l'approvisionnement de ressources à une granularité plus fine (cpu, mémoire, dique, réseau). Dans le contexte de notre travail, nous considérons l'approvisionnement à la granularité des machines virtuelles et à la granularité fine. RESERVOIR [Roc09] supporte deux modes d'approvisionnement : *explicite* et *implicite*. Le mode *explicite* exige qu'au moment du déploiement, le fournisseur de service spécifie précisément les besoins en capacité de l'application dans des conditions de charge de travail spécifiques. Le mode *implicite* couvre les besoins en capacité pour les applications de services non calibrés. Dans ce mode, le fournisseur de services ne peut avoir que des estimations de dimensions initiales de son service ou ne pas avoir d'estimation de dimensionnement du tout. En mode *explicite*, en utilisant les méta-données, le développeur a la possibilité d'exprimer le type de ressources, le nom du fournisseur de ressources, l'emplacement de cette dernière. Cette méta-donnée peut être exprimée à différentes granularités (application ou composant). Cloud4SOA [Kam13] utilise une API de haut niveau pour approvisionner les ressources et déployer les applications. Aneka [Vec09] fournit un modèle d'abstraction nommé Platform Abstraction Layer (PAL) pour approvisionner dynamiquement des ressources et déployer les applications. Il faudrait fournir un modèle de haut niveau permettant non seulement d'approvisionner des ressources de manière dynamique mais permettant aussi aux développeurs d'exprimer le type de ressources dont ils ont besoin. 4CaaSt [Gar12] utilise une extension du modèle OVF [For09] contenant toutes les informations pour "exécuter" le déploiement, en particulier les spécifications VM, la liste des outils et environnements qui doivent être installés sur la machine virtuelle, y compris les scripts de déploiement. Cloud4SOA [Dan12], PaaSage [Paa13], Aneka [Vec09], Cloud Foundry [Fou11] et Cloudify [Gig12] fournissent des mécanismes pour approvisionnner des ressources au niveau IaaS afin de déployer des applications dessus. Les auteurs de l'article [Sel13] proposent une solution appelée PaaS application provisioning and management API (COAPS) pour approvisionner des ressources au niveau PaaS. Cependant, CompatibleOne [Yan13] fournit deux modèles différents pour approvisionner des ressources aux niveaux IaaS et PaaS. Ainsi, CompatibleOne [Yan13] offre la possibilité de déployer une application sur un IaaS et PaaS. Toutefois, CompatibleOne n'a pas un modèle uniforme pour approvisionner des ressources à la fois au niveau IaaS et PaaS. TUNe [Tch10] est une plateforme basée sur un système autonome permettant d'approvisionner et de retirer des ressources dynamiquement en fonction de la charge de travail de l'application.

Les auteurs de l'article [Cha10] proposent une approche permettant de déployer les composants des applications réparties en spécifiant pour chaque composant son emplacement. Leur approche prend aussi en compte l'ordre dans lequel les composants doivent être déployés. Toutefois, dans le système de placement, ils tiennent uniquement compte des noms de fournisseur de nuage. Le développeur doit pouvoir définir un lieu géographique donné pour ces composants. EnaCloud [Li 09] et les articles [Bel10b, Bel10a] proposent le placement dynamique des applications de nuage en tenant compte de la consommation de l'énergie.

Dans leur approche, les applications migrées sont encapsulées dans des machines virtuelles qui sont déplacées. Cette machine virtuelle intègre un planificateur d'applications qui décide de migrer celles-ci. Toutefois, cette approche est limitée à un seul nuage et au fait qu'on ait besoin de déplacer une machine virtuelle.

Certaines bibliothèques *multi-nuages* telles que DeltaCloud [Apaa], jClouds [Apab], daseing [Das], LibCloud [Fou] et CompatibleOne [Lef11] permettent d'approvisionner des ressources à travers plusieurs fournisseurs de nuage. La plupart d'entre elles agissent comme des enveloppes (wrapper) d'autres technologies (par exemple, LibCloud, DeltaCloud ou jClouds). Elles proposent une API uniforme pour les différents fournisseurs de nuage. CompatibleOne implante le standard Open Cloud Computing Interface (OCCI) [Edm12]. Chaque bibliothèque est implantée dans un langage de programmation différent (par exemple, LibCloud utilise Python, jClouds utilise Java et DeltaCloud utilise Ruby). Parmi les solutions existantes, il n'existe pas une couche d'abstraction de haut niveau qui cache la complexité d'approvisionnement de ressources sur plusieurs nuages.

Les auteurs de l'article [Fos06] ont abordé le problème de déploiement d'une grappe de machines virtuelles avec des configurations de ressources données à travers un ensemble de machines physiques. Tandis que d'autres auteurs de l'article [Cza05] définissent une API Java permettant aux développeurs de contrôler et de gérer une grappe de machines virtuelles Java et de définir les politiques d'allocation de ressources pour ces grappes. Contrairement aux auteurs [Fos06, Cza05], on pourrait envisager une solution qui utilise à la fois les approches centrées sur l'application et la machine virtuelle. En utilisant les connaissances sur la charge de travail de l'application et les objectifs de performance associés à l'utilisation du serveur, la solution utilise un ensemble plus polyvalent de mécanismes d'automatisation. En outre, les ressources approvisionnées doivent être gérées et consolidées comme un ensemble cohérent et logique afin de faciliter leurs utilisations et permettre une gestion efficace et optimale de celles-ci.

3.2.3.3 Elasticité *multi-nuages*

La gestion de l'élasticité entre plusieurs nuages est un défi difficile. Cependant, bien que l'élasticité à travers plusieurs nuages offre un avantage lorsqu'une panne se produit, peu de solutions utilisent l'élasticité *multi-nuages*. Par exemple, dans l'article [Buy10], les auteurs présentent une approche pour fédérer des infrastructures de nuages pour fournir l'élasticité aux applications. Toutefois, ils ne prennent pas en compte la gestion de l'élasticité lorsqu'une panne survient. Une autre approche a été proposée par les auteurs [Vaq11]. Elle gère l'élasticité avec l'utilisation d'un contrôleur et de l'équilibreur de charge. Mais, leur solution n'adresse pas l'élasticité à travers plusieurs nuages. Les auteurs de l'article [Mar10c] proposent un gestionnaire de ressources pour gérer l'élasticité des applications. Toutefois, leur approche est spécifique à un seul fournisseur de nuage.

Dans certains environnements de nuage tel que Amazon Elastic Load balancing [Gui10], les métriques de la qualité de service (c'est-à-dire nombre de requêtes, latence d'une requête)

sont observées par Amazon Cloudwatch. Le mécanisme de mise à l'échelle d'Amazon dépend de l'instanciation d'une machine virtuelle comme équilibreur de charge qui distribue les trafics d'autres machines virtuelles similaires. Cette approche a deux limitations. Premièrement, elle est limitée à une application spécifique comme les serveurs Web et ne sont pas applicables à d'autres applications comme les bases de données. Deuxièmement, il n'est pas possible de supporter une application complexe qui a besoin des règles spécifiques d'élasticité. mOSAIC [Pet13b] gère le mécanisme d'élasticité de manière implicite et automatique. Les solutions RESERVOIR [Roc09], Aneka [Vec09], 4CaaSt [Gar12] et CONTRAIL [Car12] gèrent l'élasticité des applications en tenant compte uniquement des ressources. Cependant, l'élasticité pourrait être exprimée non seulement en fonction des ressources, mais aussi en fonction du coût et de la qualité de service. Comme nous l'avons remarqué avec Cloudify (voir section 3.2.2.14), le langage d'élasticité proposé par Cloudify est non seulement complexe mais pas assez expressif. En effet, il nous est impossible de définir des règles d'élasticité qui peuvent se déclencher à une heure ou une date précise en utilisant le langage d'élasticité Cloudify. Même si les langages d'élasticité proposés ne sont pas complets (en terme d'expression), ils doivent pouvoir offrir la possibilité aux développeurs de l'étendre pour l'enrichir.

Les auteurs de l'article [Cha10] proposent un langage de définition d'élasticité. Leur langage est basé sur Open Virtual Format (OVF) [For09], une spécification du standard DMTF pour conditionner et déployer des machines virtuelles. Leur langage d'élasticité adopte l'approche événement-condition-action pour définir les exigences. Cette approche est une bonne pratique. Cependant, en terme de granularité, on doit être capable d'exprimer le langage d'élasticité sur un composant de l'application déployée.

Les auteurs de l'article [Pie12] proposent une solution pour déployer une application complexe composée de plusieurs services qui utilisent un fichier manifest. Ils ont conçu un langage propriétaire du manifest pour spécifier la structure entière de l'application déployée. L'utilisation du standard permet de limiter l'utilisation d'un nouveau langage propriétaire.

Les auteurs de l'article [Elm11] se sont focalisés sur la description des problèmes de gestion dans des architectures *multi-nuages*. Les auteurs des articles [Rod10, Lim10] ont proposé un intergiciel qui permet aux utilisateurs de gérer la mise à l'échelle des applications en définissant une règle d'élasticité basée sur l'automatisation. Cependant, tout système de gestion de passage à l'échelle est lié à l'API de nuage sous-jacent (le problème de "discrete actuators" nommé par [Lim10]). Une tâche essentielle pour tout contrôleur d'élasticité au niveau de l'application est celle qui fait un mapping des politiques de mise à l'échelle de l'utilisateur à partir du niveau d'abstraction approprié vers les mécanismes originaux fournis par les infrastructures de nuage. Les auteurs de l'article [Lim09] proposent une solution qui se focalise sur le problème de construction de contrôleurs externes pour les applications dynamiques hébergées dans le nuage en utilisant la boucle de contrôle. Avec cette solution, dans un environnement *multi-nuages*, on doit définir autant de contrôleurs que de fournisseurs de nuage. Toutefois, il est préférable d'utiliser un moyen uniforme pour gérer l'élasticité à travers plusieurs nuages basé sur la boucle de contrôle. Les auteurs [Bar03] ont développé une solution

basée sur un agent pour automatiser le réglage du système. Leur agent fait le contrôleur et la boucle de contrôle à la fois. Toutefois, la lenteur du système (c'est-à-dire 10 minutes au maximum pour un client) rend ce dernier inapproprié à des changements brusques de charge de travail.

3.2.3.4 Haute disponibilité *multi-nuages*

Les fournisseurs de nuage tels que Amazon EC2, Windows Azure, Jelastic fournissent déjà un service d'équilibreur de charge destiné à un seul nuage pour distribuer des charges à travers plusieurs machines virtuelles. Toutefois, ces services d'équilibreur de charge ne font pas de distribution dans un contexte *multi-nuages*. Différentes approches de distributeur de charge dynamique existent dans la littérature [Car99, Har97, Lin92], mais elles ne fournissent pas de mécanisme permettant de mettre à l'échelle l'équilibreur de charge lui-même. Les auteurs des articles [Zha10c] et [Qia07] ont exploré la façon agile de réaffecter rapidement des ressources en utilisant le protocole de groupe d'appartenance (membership protocols). Cependant, leur approche ne tient pas compte d'un environnement *multi-nuages*. La plupart des protocoles de groupe d'appartenance [Bir94, Mos96, Ami92] existant emploient un algorithme de consensus pour parvenir à un accord dans le groupe d'appartenance. Parvenir à un consensus dans un système réparti asynchrone est impossible sans l'utilisation du délai d'attente permettant d'indiquer en combien de temps une action doit avoir lieu. Même avec l'utilisation de temps morts, la réalisation du consensus peut être relativement coûteuse en nombre de message émis et les retards qui peuvent survenir. Pour éviter de tels coûts, une approche consiste à utiliser l'élection de leader dans le protocole de groupe d'appartenance qui n'implique pas l'utilisation d'algorithme de consensus. RESERVOIR [Roc09], Cloud Foundry [Fou11], Cloudify [Gig12], CompatibleOne [Yan13], MODAClouds [Ard12] et CONTRAIL [Pie12] assurent essentiellement la haute disponibilité au niveau de l'application avec l'utilisation respective de Business Service Management (BSM) et de l'équilibreur de charge. Toutefois, la haute disponibilité devrait être prise en compte à la fois au niveau de l'application et au niveau de la plateforme elle-même.

Nous avons discuté des différentes solutions *multi-nuages* en fonction des quatre défis que nous avons identifiés dans notre travail de thèse. Dans la section suivante, nous allons présenter un récapitulatif de ces travaux.

3.2.3.5 Synthèse

Afin de mieux comprendre ce qui est fait dans la littérature, le tableau 3.2 fait le récapitulatif des solutions pertinentes avec les obstacles qu'elles adressent. Dans notre analyse, nous avons choisi les quatre problématiques la *portabilité*, l'*approvisionnement*, l'*élasticité*, la *haute disponibilité* comme critère d'évaluations.

Signification des symboles (+ et -) utilisés dans le tableau 3.2.
— **+** : le plus indique que le critère a été adressé.

Tableau 3.2 – Vue d'ensemble des solutions connexes.

Solutions	Couche	Portabilité	Approvisionnement	Élasticité	Haute disponibilité
mOSAIC [Mos11]	PaaS	+	+	+	-
RESERVOIR [Roc09]	IaaS	-	+	+	+
Cloud4SOA [Dan12]	PaaS	+	+	-	-
REMICS [Moh11]	PaaS	-	-	-	-
PaaSage [Paa13]	PaaS	-	+	+	-
MODAClouds [Ard12]	PaaS	+	-	-	+
Aneka [Vec09]	PaaS	-	+	+	-
4CaaSt [Gar12]	PaaS	-	+	+	-
CONTRAIL [Har11, Pie12]	IaaS/PaaS	-	+	+	+
STRATOS [Paw12]	IaaS	-	+	+	-
CompatibleOne [Yan13]	IaaS	+	+	-	+
OPTIMIS [Fer12]	IaaS	-	+	+	-
Cloud Foundry [Fou11]	PaaS	+	-	+	+
Cloudify [Gig12]	PaaS	+	-	+	+

— **-** : le moins indique que le critère n'a pas été adressé.

Ce tableau récapitulatif montre que les quatre problématiques discutées dans les sections 3.2.3.1, 3.2.3.2, 3.2.3.3 et 3.2.3.4 restent des défis de l'*informatique multi-nuages*.

Dans la section suivante, nous présentons et discutons des travaux qui sont basés sur l'ingénierie d'applications en nuage.

3.2.4 Approches et méthodologies pour développer des applications en nuage

Dans cette section, nous présentons certains travaux qui utilisent les technologies de nuages pour concevoir des applications. La section est divisée en quatre parties :

1. **Modèle d'applications** Les solutions qui fournissent des modèles pour concevoir des applications en nuage.
2. **Ingénierie dirigée par les modèles** Les approches qui visent à présenter les projets qui utilisent des modèles.
3. **Langage d'architecture** Les approches qui présentent des langages d'architecture d'applications.
4. **Langage d'élasticité** Les solutions qui proposent des langages dédiés pour exprimer l'élasticité.

3.2.4.1 Modèle d'applications

Les auteurs de l'article [Sel13] proposent un modèle basé sur les templates des standards CAMP et TOSCA. Leur modèle permet de décrire à la fois l'application et l'environnement dans lequel celle-ci est déployée. Ce modèle fournit un moyen simple permettant de décrire l'environnement d'exécution de l'application en utilisant des templates. Toutefois, leur modèle d'application ne permet pas de décrire une application répartie. Aneka [Vec09] fournit un modèle pour développer des applications réparties. Ce modèle permet de définir les propriétés et exigences d'une application répartie. Ce modèle considère une application répartie comme un ensemble unité d'exécution qui dépend de la spécifité du modèle de programmation utilisé. Le modèle Aneka utilise trois modèles de programmation qui sont à base de *tâche*, *thread* et *MapReduce*. Le concept de gestion d'une application répartie comme un ensemble d'unités d'exécution est une bonne pratique, car il permet d'isoler l'exécution de chaque partie de l'application répartie de manière indépendante. Le modèle d'application mOSAIC [Mos11] gère les applications réparties de la même façon. Toutefois, ces deux modèles d'applications sont restrictifs au niveau du modèle de programmation qu'ils proposent, car ils n'offrent pas de moyens permettant d'exprimer des besoins non-fonctionnels qu'exigent les applications réparties pour un environnement *multi-nuages*. Par ailleurs, les deux solutions Aneka et moSAIC fournissent des modèles propriétaires et spécifiques à leurs plateformes. En effet, le développeur d'applications doit pouvoir développer ses applications avec le modèle de programmation qui répond le mieux à ces besoins et aussi utiliser un standard existant plutôt que d'utiliser un modèle propriétaire spécifique à une plateforme.

Le descripteur de fichier de mOSAIC [Mos11] (section 3.2.2.1 listing 3.1) ne définit pas la manière dont les dépendances des composants d'une application sont exprimées. mOSAIC fournit un modèle de programmation à base de composants et utilise uniquement la communication asynchrone. Toutefois, le descripteur de fichier de Cloudify [Gig12] (section 3.2.2.14 listing 3.2) permet d'exprimer les dépendances entre les différents composants d'une application répartie. mOSAIC [Mos11] n'offre pas la possibilité d'exprimer l'élasticité en utilisant un langage, ce qui n'est pas le cas avec Cloudify [Gig12]. Toutefois, la manière dont Cloudify [Gig12] permet d'exprimer la règle d'élasticité est rudimentaire et très complexe. Par exemple, le listing 3.3 montre comment on exprime une règle basique qui "ajoute une instance de service tomcat lorsqu'il y a une requête par seconde". On doit être capable d'exprimer une règle d'élasticité de manière plus simple et intuitive. En outre, le langage défini par Cloudify pour gérer l'élasticité ne nous permet pas d'exprimer des règles qui sont capables de se déclencher à une période de temps prédéfinie par exemple. En plus, ce langage ne tient pas compte des coûts et des ressources utilisées. En effet, la gestion de l'élasticité d'une application qui prend en compte les ressources et le coût offre plus d'expressivité. Un langage de règles d'élasticité doit pouvoir permettre d'exprimer tous ces besoins de manière simple et claire.

Les modèles mOSAIC [Mos11] et Cloudify [Gig12] ne nous permettent pas d'exprimer le type ou la quantité de ressources requis par une application. Toutefois, le modèle

STRATOS [Paw12] fournit un modèle permettant d'approvisionner des ressources au niveau IaaS. Par ailleurs, le modèle CompatibleOne [Yan13] fournit des manifests permettant non seulement d'approvisionner des ressources au niveau IaaS mais aussi au niveau PaaS. Pour l'approvisionnement des ressources au niveau IaaS ou PaaS, le modèle compatibleOne définit deux fichiers manifests différents pour exprimer le besoin d'approvisionnement de ressources. Afin de faciliter la tâche des développeurs, une approche consisterait à fournir une manière uniforme d'exprimer l'approvisionnement de ressources à la fois au niveau IaaS et PaaS.

Les nouvelles technologies, les standards et les intergiciels liés à l'informatique en nuage peuvent faciliter le développement d'applications, l'évolution du logiciel actuel et rendre les entreprises plus compétitives. La littérature sur l'ingénierie logicielle pour le nuage est récente [Sil12]. L'ingénierie des applications de l'*informatique multi-nuages* est l'une des solutions alternatives trouvées dans le monde industriel et académique pour éviter les problèmes liés à l'utilisation d'un seul nuage [Par13]. L'utilisation d'un standard peut aider à améliorer ce modèle, mais il n'existe toujours pas de standard adopté pour le développement d'application pour le nuage (comme discuté dans la section 3.2.1).

La composition de service, dans le contexte d'architecture orientée service, reste encore un sujet important de recherche dans le domaine du génie logiciel [da 11, Pap08]. Dans ce contexte, les termes "inter-cloud" ou "multi-cloud" sont utilisés pour se référer à la composition de fournisseurs de nuage comme une intégration (fédération) d'environnements de nuage [Vil12b]. Dans de tels environnements, les utilisateurs peuvent, de manière transparente ou non, utiliser des ressources provenant de différents fournisseurs de nuages publics ou privés [Mar10b, Car11, Liu11a, Luc11, Jai12, Mor11b]. Ces termes sont assez similaires et suggèrent que l'informatique en nuage ne doit pas être limitée à un seul nuage [Car11].

Du point de vue du génie logiciel, la mise en œuvre d'une application en nuage nécessite certaines préoccupations spécifiques. Ces problèmes résultent du modèle d'entreprise axé sur le service qui est adopté.

3.2.4.2 Ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (MDE) [Fra07] est une approche viable au problème de dépendance à l'égard d'un fournisseur, grâce à des modèles de haut niveau. Les transformations se font automatiquement, les développeurs sont en mesure de se concentrer sur un niveau plus conceptuel, indépendant de la plateforme de développement de l'application. Cependant, il n'y a pas de consensus sur l'ensemble des modèles, langages, les transformations et les processus logiciels qui pourraient être utilisés pour développer avec succès des applications de nuages [Bru10]. Les efforts actuels tournent principalement autour de l'identification des abstractions qui permettent de spécifier et de créer des systèmes indépendamment de la plateforme sous-jacente et la génération automatique de code pour une plateforme spécifique ou de l'infrastructure de service [Sha11, Jia10].

Amazon AWS CloudFormation [Ama12a] est un service offert par le fournisseur de nuage Amazon. Ce service donne aux développeurs la possibilité de créer des fichiers template sous la forme JavaScript Object Notation (JSON) qui est chargé dans AWS pour approvisionner des piles de ressources. Une pile est un ensemble défini de ressources de différentes tailles et quantités telles que la base de données, l'équilibreur de charge, etc. Ce système de fichier est facile à reproduire et configurable par l'utilisateur. Le format JSON utilisé par Amazon est un bon format, la syntaxe JSON est lisible et facile à utiliser. Toutefois, la sémantique du modèle lui-même est propiétaire et n'est pas utilisé par d'autres fournisseurs. Ce modèle ne peut pas être considéré comme une solution *multi-nuages*.

3.2.4.3 Langage d'architecture

Les auteurs [Ren04] ont identifié un ensemble de besoins présents dans les applications à base de service en utilisant un langage de description d'architecture (ADL). Avec les besoins exprimés, les composants qui représentent des services dans ce type d'architecture devraient contenir des informations sur les ressources qu'ils utilisent, de besoins fonctionnels (par exemple les entrées/sorties) et les besoins non-fonctionnels (qualité de service, placement), des modèles de prix. Bien que les travaux de ces auteurs donnent des directives en vue d'utilisation d'un ADL, ils ne proposent aucune spécification. La séparation des préoccupations n'a pas été illustrée entre les besoins fonctionnels et non-fonctionnels. Les auteurs [Dan06] proposent un nouveau ADL, pour décrire des architectures orientées services. Ce nouveau ADL nommé SOADL est basé sur le langage XML. Il est composé de trois éléments principaux : i) *service*, représenté par les composants, ii) *connecteur* qui définit comment les composants sont connectés et de quelle manière ils utilisent les protocoles de communication et iii) *configuration*, le graphe des composants interconnectés. Cependant, l'approche SOADL est basée sur la proposition d'un nouveau ADL spécifique plutôt que d'étendre un ADL existant. En outre, les développeurs peuvent être accablés par l'apprentissage d'un nouveau langage. De plus, les travaux basés sur le MDE et l'ADL ne fournissent pas de spécifications décrivant la séparation des préoccupations.

SOCCA [Tsa10] propose une approche qui fait le mapping entre les exigences d'une application et le modèle pré-défini dans leur modèle. Ce type d'approche ne tient pas compte des applications patrimoniales.

Il existe un certain nombre de langages de description dans le domaine de l'ingénierie logicielle qui servent de configuration d'exécution et de déploiement de systèmes logiciels à base de composants. Le CDDML Component Model [J05], par exemple, décrit les exigences pour la création d'un objet de déploiement responsable de la durée de vie des ressources déployées. Il est utilisé pour les services de grille (grid). Chaque objet de déploiement est défini en utilisant le langage CDL. Le modèle définit également des règles de gestion sur l'interaction des objets avec l'API de déploiement du CDDML. Cependant, on doit pouvoir exprimer la relation entre le domaine et les modèles syntaxiques à un niveau d'abstraction plus élevé en s'appuyant par exemple, sur les annotations pour exprimer des contraintes non fonctionnelles.

3.2.4.4 Langage d'élasticité

La mise à l'échelle automatique a été introduite par Amazon EC2 pour permettre au mécanisme d'approvisionnement de mettre à l'échelle automatiquement des services selon des conditions définies par le développeur. Ces conditions sont définies sur la base de l'utilisation des ressources observées, telles que l'utilisation du cpu, de la mémoire, du réseau ou du disque. Cette approche est limitée pour définir des règles d'élasticité plus sophistiquées. En effet, la nécessité d'augmenter ou de retirer des ressources ne peut être identifiée que grâce à ces mesures. Nous avons besoin d'une compréhension du processus de planification. La capacité de décrire et de surveiller l'état de l'application est cruciale si nous voulons anticiper correctement la demande. Les auteurs [Far12] utilisent une approche d'ordonnancement statique en définissant un algorithme d'optimisation multi-objectif et ont démontré son utilisation sur certaines applications. Le développeur doit pouvoir exprimer l'élasticité de son application à la fois en fonction des ressources, du coût et de la qualité de service.

Les auteurs [Han12] utilisent une approche de mise à l'échelle à grain fin au niveau de ressources (cpu, mémoire, réseau) et à gros grain au niveau des machines virtuelles utilisant un algorithme léger de mise à l'échelle pour améliorer l'utilisation de ressources et la réduction du coût du fournisseur de nuage. Le contrôle de l'élasticité doit être pris en compte à plusieurs niveaux. Ainsi, la gestion de ressources s'effectuera en trois niveaux (grain fin, gros grain, nuage).

Les auteurs des articles [Mor11a, Cha10] fournissent une approche à base de règles pour spécifier les exigences de l'application. En contraste avec ces deux approches, notre principal objectif est de fournir un langage de haut niveau pour décrire les exigences d'élasticité qui se basent sur les évènements, conditions et actions.

3.3 Positionnement

Comme nous l'avons expliqué dans le chapitre 1, nous n'avons pas un seul nuage homogène mais plusieurs nuages disparates. Le paradigme *multi-nuages* a besoin d'une abstraction facilitant la conception d'une application répartie pour un environnement *multi-nuages* capable de tenir compte des défis (discutés dans la section 3.1) qu'imposent l'environnement *multi-nuages*. Nous avons étudié un certain nombre de solutions *multi-nuages* existantes et les avons évaluées par rapport à ces défis.

soCloud c'est :

1. un modèle pour concevoir des applications SaaS et une plateforme PaaS bâtie au dessus des IaaS et PaaS existants.
2. un modèle de déploiement *multi-nuages*.

3. une solution qui s'adresse aux consommateurs, fournisseurs, auditeurs et transporteurs de l'informatique en nuage.
4. une méthode hybride pour gérer son mécanisme d'élasticité.
5. une solution qui se positionne dans la sous-famille de *multi-nuages* orientés services dans la taxonomie [Pet13a]. Quant à la taxonomie [Gro12], *soCloud* se positionne dans la sous-famille service *multi-nuages*.

Dans ce chapitre de l'état de l'art, nous avons montré les limites des solutions *multi-nuages* existantes du fait qu'elles n'adressent pas les quatres défis mentionnés dans ce travail de thèse. Nous avons mis l'accent sur les inconvénients des modèles d'applications pour un environnement *multi-nuages*. Bien évidemment, ceci nous amène à proposer une solution alternative pour adresser les défis que nous avons exprimés à la section 3.1. Le modèle d'applications *soCloud* que nous présentons au chapitre suivant, est nécessaire pour concevoir des applications réparties à large échelle pour un environnement *multi-nuages*. En effet, l'une des préoccupations majeures dans le domaine *multi-nuages* est d'avoir un modèle permettant de concevoir de manière simple une application répartie pour des environnements *multi-nuages*. En outre, il est nécessaire de fournir une plateforme *multi-nuages* permettant de déployer, d'exécuter et de gérer des applications *multi-nuages*.

Deuxième partie

CONTRIBUTION

Chapitre 4

Modèle d'applications *soCloud*

"On croit que le style est une façon compliquée de dire des choses simples, alors que c'est une façon simple de dire des choses compliquées."-Jean Cocteau

Sommaire

Dans ce chapitre, nous présentons notre première contribution qui consiste à fournir un modèle de conception et d'implantation d'une application orientée services dans un environnement *multi-nuages*, que nous nommons application *soCloud*.

4.1 Introduction

De nos jours, l'*informatique multi-nuages* ou *Multi-cloud Computing* apparait comme un paradigme prometteur pour supporter des applications réparties à large échelle. L'*informatique multi-nuages* consiste en l'utilisation de multiples environnements de nuages indépendants qui ne nécessitent pas d'accord a priori entre les fournisseurs de nuage ou un tiers. La conception, la portabilité et le déploiement d'applications réparties dans un environnement *multi-nuages* est une tâche très complexe et un véritable défi, car il n'existe pas une manière uniforme, simple et complète pour concevoir des applications qui vont s'exécuter dans des environnements hétérogènes de nuages (voir chapitre 3). Les développeurs d'applications en nuage passent beaucoup de temps pour préparer, installer et configurer leurs applications. En outre, après le développement et le déploiement, les applications ne peuvent être déplacées d'un fournisseur de nuage à un autre sans se confronter aux problèmes liés à la portabilité d'applications [Pet11a, Bin12]. Actuellement, l'état de l'art a montré qu'il n'existe pas de consensus sur le style et le modèle d'architecture favorisant la conception d'applications orientées services dans un environnement *multi-nuages* [Cha12, Gal09, La 09, Max09, Mie10, Tsa10] (discuté dans la section 3.2.4 du chapitre 3). Ces méthodes ont clairement montré des lacunes considérables pour fournir une solution simple et efficace permettant de faire face aux principaux défis liés à la conception d'applications pour un environnement *multi-nuages*.

Dans ce chapitre, nous proposons une vue d'ensemble du modèle de conception d'une application *soCloud*. Ensuite, nous présentons les problèmes et besoins liés à la conception d'une application *soCloud*. Ce chapitre présente le modèle utilisé comme support pour la conception, la spécification, l'implantation et l'assemblage d'applications *soCloud*. Une application *soCloud* peut être composée d'une ou plusieurs unités d'exécution. Le modèle d'applications *soCloud* permet de décrire chaque unité d'exécution et les propriétés de configuration qui leur sont associées. Ce modèle précise l'organisation et la dépendance entre unités d'exécution au sein de l'architecture d'une application *soCloud*. Des exigences non-fonctionnelles (c.à.d. contraintes de configuration, de disponibilité, de placement, d'élasticité, caractéristiques matérielles et logicielles des machines virtuelles) peuvent être exprimées sur chaque unité d'exécution via le modèle. Nous mettons en œuvre notre approche en implantant le cas d'utilisation de la section 4.2.

4.2 Cas d'utilisation

La figure 4.1 illustre l'architecture d'une application répartie constituée de trois-tiers (*Présentation*, *Métier* et *Stockage*). Le tiers *Présentation* gère les interactions avec l'utilisateur et est accessible via un navigateur Web et une API REST. Ensuite, le tiers *Métier* traite les problèmes de logique applicative relatifs au domaine d'application du système d'information. Enfin, le tiers *Stockage* est utilisé par les tiers *Présentation* et *Métier* pour conserver des informations sur un support de persistance (par exemple une base de données). Cet exemple a été choisi car il est assez représentatif de nombreuses applications en nuage. Par exemple, nous pouvons avoir plusieurs réplications du tiers *Présentation* en fonction du lieu géographique des utilisateurs finaux, mais aussi plusieurs réplications du tiers *Métier* pour les charges de travail accrues et enfin plusieurs réplications du tiers *Stockage* pour la redondance en cas de pannes. Toutefois, d'autres types d'applications réparties peuvent être envisagés avec *soCloud* (voir chapitre 6).

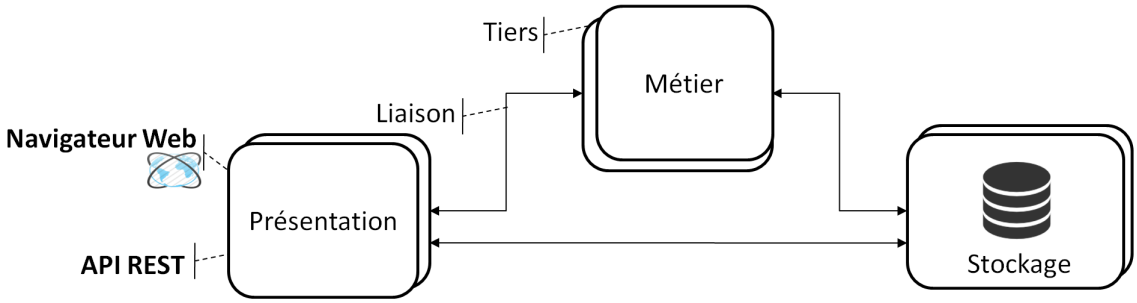

Figure 4.1 – Architecture d'une application trois-tiers : scénario de cas d'utilisation.

Cette application doit faire face à des exigences comme :

— La localisation géographique de chaque tiers de l'application. Ainsi le développeur doit pouvoir exprimer le lieu géographique ou le fournisseur de nuage chez qui il souhaite déployer chaque composante.
— Le choix de la machine virtuelle (en terme de caractéristiques matérielles) sur laquelle chaque composante de l'application répartie doit être déployé.
— La latence réseau entre des tiers connexes de l'application déployée géographiquement ou entre un tiers et les utilisateurs finaux qui sont dans une zone géographique donnée. Pour des questions de performance, le développeur doit pouvoir exprimer ces types d'exigence sur plusieurs tiers connexes.
— La croissance imprévisible du nombre d'utilisateurs qui se connectent à l'application. Le développeur doit pouvoir exprimer des règles d'élasticité spécifiques sur chacun des tiers de l'application afin de décrire la manière dont chaque tiers de l'application passera à l'échelle.

— La disponibilité de l'application malgré les pannes est indispensable. Quelle que soit la robustesse d'un système, les pannes sont inévitables. Elles doivent donc être prises en compte lors de la conception.
— Le contrôle des coûts : il est très important de tenir compte des incidences financières lors de la conception de l'architecture.

En outre, certains besoins non-fonctionnels doivent également être pris en compte comme par exemple :

— Déployer trois instances du tiers *Présentation* à Singapour.
— Redimensionner automatiquement les ressources consommées par le tiers *Présentation* lorsque la consommation sur une période de 24 heures dépasse 200 euros.
— Déployer le tiers *Métier* sur une puissante (en terme de caractéristiques matérielles) machine virtuelle et dont le système d'exploitation est Ubuntu.
— Mettre à l'échelle le tiers *Métier* lorsque le temps de réponse sur une période d'une minute est supérieur à 4 secondes et le nombre de requêtes entrantes est supérieur à $1000r/min$.
— Placer le tiers *Métier* à côté du tiers *Stockage* pour diminuer la latence réseau.
— Déployer deux instances du tiers *Stockage* en France.
— Le tiers *Stockage* utilise une base de données de type MySQL.

La question se pose alors de savoir comment exprimer de manière simple et concise ces besoins non-fonctionnels.

Pour cela, nous proposons le modèle *soCloud* comme support pour décrire, concevoir, implanter et assembler des applications *multi-nuages*. Ce modèle offre une solution effective, flexible et simple aux développeurs d'applications pour mettre en œuvre leurs applications dans un environnement *multi-nuages*. Avant de présenter la vue d'ensemble du modèle d'applications *soCloud*, voyons d'abord les principes de conception d'une application *soCloud*.

4.3 Principes de conception

Dans le domaine de l'*informatique multi-nuages*, les exigences comme la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité* sont de véritables défis comme discuté dans le chapitre 3 section 3.1. Le modèle d'une application *soCloud* doit capturer les informations nécessaires à ces exigences pour fournir une solution générique, flexible et simple. Générique, parce qu'elle doit être indépendante de toute technologie et plateforme de nuage. Flexible, pour permettre aux développeurs d'exprimer des comportements spécifiques à une application. Simple, pour diminuer la complexité des tâches de développement. Ainsi, le modèle d'une application *soCloud* a pour vocation de capturer :

— L'architecture de l'application, en terme de dépendances entre les différentes unités d'exécution pour effectuer un déploiement chronologique.

— Les contraintes de placement qui définissent l'organisation et le lieu géographique des unités d'exécution de l'application.
— La disponibilité de l'application en présence d'un nombre fini de défaillances affectant l'environnement d'exécution applicatif. Ces défaillances ne doivent pas empêcher cette dernière de mener à bien son fonctionnement.
— Les règles d'élasticité pour exprimer le comportement (c.à.d. actions) de l'application lorsque l'état de l'environnement ou de l'application elle-même change suite à une charge de travail accrue.
— Le type de machine virtuelle en termes de caractéristiques matérielles utilisé pour exécuter une unité ou l'ensemble de l'application.
— Le logiciel que requiert une unité d'exécution d'une application *soCloud* (par exemple une base de données) pour fonctionner.

Une application *soCloud* est conçue pour tirer avantages des plateformes de nuage tout en étant indépendante vis-à-vis de celles-ci. Comme illustré dans la figure 4.2, la plupart des applications en nuage ont une forte dépendance par rapport aux plateformes (PaaS) ou infrastructures (IaaS) sur lesquelles elles sont déployées, ce qui entrave la portabilité d'un fournisseur à l'autre. Le but d'une application *soCloud* est de se détacher de toutes contraintes vis-à-vis des plateformes de nuage sous-jacentes.

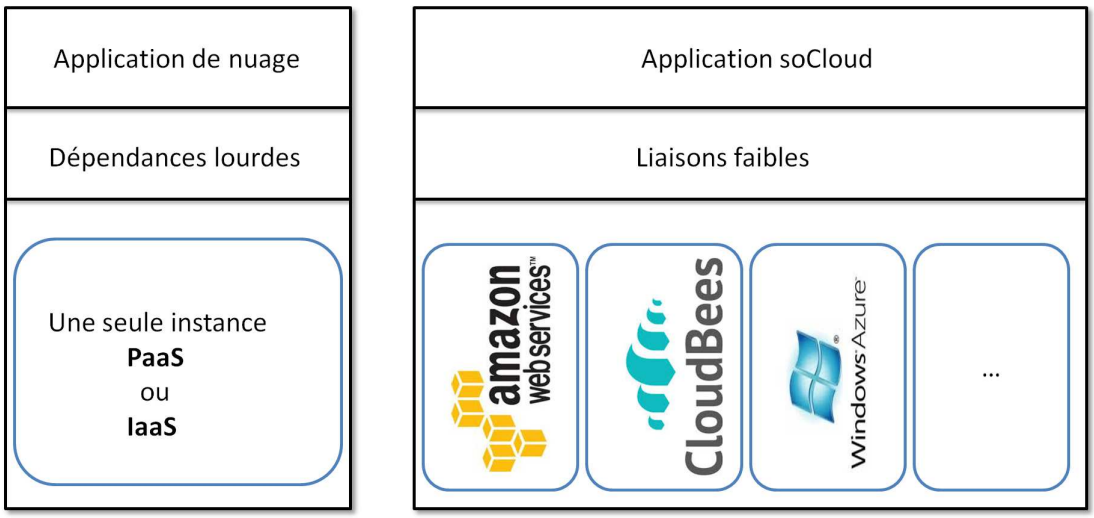

Figure 4.2 – Application *soCloud* : Indépendance par rapport aux plateformes de nuage.

En outre, les premières étapes du développement d'une application *soCloud* sont cruciales pour parvenir à une meilleure adéquation entre les besoins exprimés et la réalisation proposée. L'accent est donc mis tout d'abord sur une expression précise et non ambigüe des besoins. Ces besoins peuvent être structurés de manière globale en utilisant des styles d'architectures. Le style d'architecture modulaire [Gar93, Bas12] offre des structures de granula-

rité plus fines qui sont souvent utilisables au niveau conception. L'approche d'architecture par composants [Che01] permet de préciser comment les composants implantés indépendamment sont intégrés. Il est souhaitable de combiner ces approches pour bénéficier de leurs avantages conjugués.

Nous proposons ici le modèle que nous avons élaboré pour cette classe de problèmes qui couvre la conception, l'implantation et le déploiement d'une application répartie dans un environnement *multi-nuages*. Le modèle utilisé est basé sur les grands principes du génie logiciel [Ghe02] notamment : la *généricité*, l'*abstraction*, l'*anticipation des évolutions*, la *modularité* et la *séparation des préoccupations*.

— *Généricité* : une application *soCloud* peut être réutilisable pour d'autres contextes applicatifs. Par exemple, un composant générique qui fournit de la persistance est adaptable à d'autres contextes.
— *Abstraction* : il s'agit de promouvoir des concepts généraux regroupant un certain nombre de cas particuliers et de raisonner sur ces concepts plutôt que sur chacun des cas particuliers. Le fait de fixer la bonne granularité de détails permet de raisonner plus efficacement. Ainsi, le modèle que nous proposons pour construire une application *soCloud* peut être utilisé pour d'autres types d'applications.
— *Anticipation des évolutions* : les applications *soCloud* de par leur environnement de déploiement *multi-nuages* ont un cycle de vie plus complexe à gérer. Il est primordial de prévoir les évolutions possibles d'une application *soCloud* pour que la maintenance soit la plus efficace possible. Pour cela, il faut s'assurer que les modifications à effectuer soient le plus locales possible. Ainsi la mise à jour d'un composant n'affecte pas ou peu les autres composants.
— *Modularité* : il s'agit de partitionner une application *soCloud* en modules (composants) qui ont une cohérence interne et qui peuvent fonctionner ensemble. Les choix d'implémentation sont indépendants de l'utilisation du module. Par exemple, une application *soCloud* dont l'architecture est répartie pourra être déployée facilement dans une zone géographique donnée.
— *Séparation des préoccupations* : il s'agit de décorréler les problèmes pour n'en traiter qu'un seul à la fois. Elle permet aussi de simplifier les problèmes pour aborder leurs complexités progressivement. Par exemple, notre approche consiste en la décomposition d'une information non-fonctionnelle complexe en plusieurs sous-problèmes qui sont tous décrits séparément.

En outre, une application *soCloud* exige un degré élevé d'automatisation en ce qui concerne son déploiement, la gestion de ses ressources informatiques (cpu, mémoire, disque, bande passante), de la gestion de son évolutivité, des pannes ou de la localisation de ressources. Cette automatisation est possible via les fournisseurs de nuage qui offrent des interfaces de gestion accessibles par les applications sans interaction humaine.

Nous avons exprimé les principes de conception d'une application *soCloud*. Les exigences de la gestion des applications *soCloud* sont exprimées en utilisant des annotations

qui doivent être mises en œuvre par les applications. La section suivante présente le modèle d'applications *soCloud*.

4.4 Vue d'ensemble du modèle d'applications *soCloud*

L'approche utilisée pour décrire l'architecture des applications *soCloud* s'appuie sur le concept de modèle à composants. Le modèle d'applications *soCloud* est une extension du modèle SCA et de sa mise en œuvre dans l'intergiciel FraSCAti (décrits en section 4.4.1.2).

4.4.1 Rappel du modèle SCA

4.4.1.1 Choix du modèle SCA

Le choix du modèle **SCA** (Service Component Architecture) [Mar10a] se justifie pour les raisons suivantes :

— Le modèle SCA est un standard du consortium international OASIS qui fournit un modèle uniforme pour construire des applications orientées services (Service Oriented Architecture) ou SOA et les infrastructures qui exécutent ces applications.
— SCA en tant que technologie répond à deux questions clés : la *complexité* et la *réutilisation*. Tout d'abord, SCA fournit un modèle de programmation simplifié pour la construction de systèmes distribués. Ensuite, SCA fournit une façon d'assembler, de gérer et de contrôler les systèmes distribués.
— Le modèle SCA supporte l'hétérogénéité des langages de programmation, des protocoles de communication, des aspects non-fonctionnels. SCA répond à la manière dont on construit des systèmes composés d'une série de parties interconnectées. En SCA, ces parties interagissent en fournissant des services qui remplissent une fonction spécifique. Les services peuvent être implémentés en utilisant différents langages de programmation. Par exemple, un service peut être implémenté en Java, C++, ou un langage spécialisé comme BPEL. SCA permet aux services de communiquer en utilisant différents protocoles de communication.
— Plusieurs implantations des spécifications SCA sont disponibles. Il existe des implantations commerciales chez IBM, Oracle et des efforts Open Source comme Eclipse, Apache Tuscany, Fabric3 et OW2 FraSCAti. La liste des solutions implantant la spécification SCA sont disponibles sur le site de Open SOA [14].
— L'équipe SPIRALS a mené des recherches sur l'adaptabilité dynamique des systèmes répartis et a proposé FraSCAti un modèle SCA réflectif [Sei09, Sei12]. Cette capacité de réflexion qu'introduit FraSCAti permet de faire de l'introspection et de la reconfiguration dynamique à l'exécution.

14. http ://www.xmarks.com/site/www.osoa.org/display/Main/Implementation+Examples+and+Tools

4.4.1.2 Modèle SCA

SCA est un ensemble de spécifications du consortium international OASIS pour la construction d'applications et systèmes répartis en utilisant les principes de SOA [Pap07] et de CBSE [Bei07]. La notation graphique du modèle SCA est essentiellement constituée de six éléments : *composite*, *composant*, *service*, *référence*, *propriété* et *liaison*, comme illustrés dans la figure 4.3. L'architecture de l'application trois-tiers de notre exemple fil rouge (voir figure 4.1) se décrit naturellement en SCA : la notion d'application se traduit en *composite*, la notion de tiers se traduit en *composant* et la notion de liaison se traduit en *liaison* (voir figure 4.3).

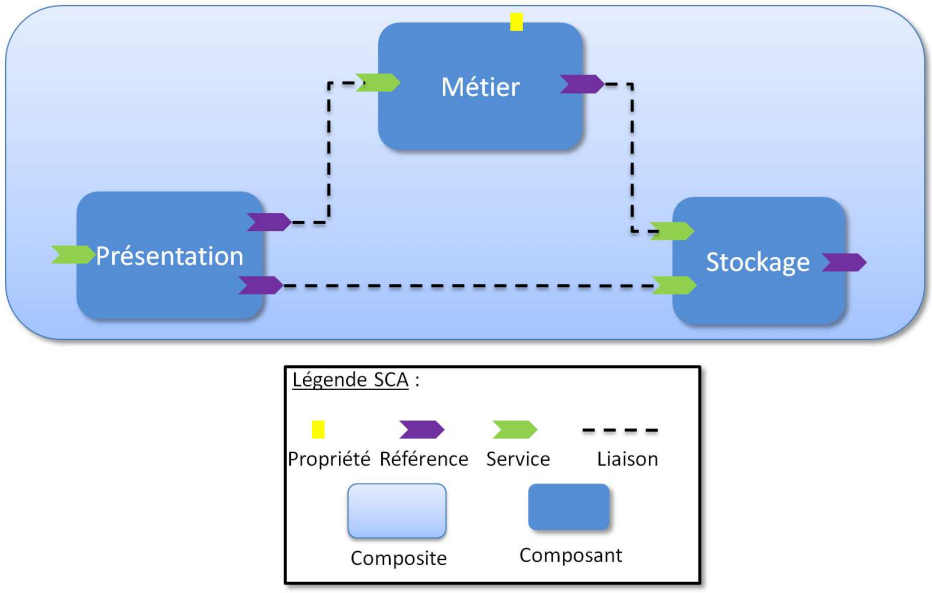

Figure 4.3 – Modèle SCA de l'application trois-tiers.

— Le *composite* est utilisé pour assembler les éléments SCA sous la forme d'un groupe logique. Il représente un moyen pour combiner des *composants* dans une structure plus large. Un composite SCA contient un ensemble de *composants*, *services*, *références* et *liaisons*.
— Le *composant* est le bloc de construction de base du modèle SCA. Lorsqu'il est combiné avec d'autres *composants*, ils peuvent interagir pour créer une solution métier complète. Essentiellement, un *composant* est une configuration d'une instance d'implémentation qui fournit et consomme des *services*. Cette implantation fournit les fonctions métiers du *composant* qui peuvent être mises en œuvre par des classes Java ou des processus BPEL ou autre langage d'implémentation.
— Le *service* représente la manière dont on accède aux fonctionnalités d'un *composant*. Il expose un catalogue d'opérations fournies par le *composant*, en utilisant une interface qui peut être accessible publiquement ou dans le même domaine. Les services d'un

composant peuvent être promus au niveau de son *composite*.
— La *référence* représente une fonctionnalité qui est requise par son *composant*. Les *références* des *composants* peuvent être promues par le *composite*.
— La *propriété* permet de configurer un *composant* avec des valeurs externes. Toutes les *propriétés* sont typées et peuvent se voir assigner une valeur par défaut via l'implémentation.
— Le *liaison* décrit une lien local entre une *référence* et un *service*. Pour supporter les interactions des applications orientées services via différents protocoles de communication, SCA fournit la notion de *binding*.
— L'*intent* est une politique déclarative qui spécifie une capacité sans identifier comment elle sera réalisée. Il permet de déclarer des propriétés non-fonctionnelles comme la sécuroté, la transaction, etc. L'itent peut être placé sur un composite, composant, service et référence.

La spécification du modèle d'assemblage SCA décrit comment des artefacts SCA et non-SCA (comme les fichiers de code) sont packagés. L'unité de déploiement en SCA est la *contribution*. Une *contribution* est un package qui contient des implémentations, des interfaces et d'autres artefacts nécessaires pour exécuter des composants. Le format du package SCA est un fichier ZIP. D'autres formats d'archives de conditionnement (WAR, EAR) sont également autorisés.

4.4.1.3 Intergiciel FraSCAti

FraSCAti [Sei09, Sei12][15] est une implémentation des spécifications SCA assurant la mise en œuvre d'applications réparties dans un contexte SOA. FraSCAti propose une approche basée sur des composants pour soutenir la composition hétérogène de différents langages de définition d'interface (WSDL, JAVA), les technologies d'implantation (Java, Spring, EJB, BPEL, OSGI, Jython, Jruby, Xquery, Groovy, Velocity, Fscript, Beanshell), les protocoles de communication réseau (Web Services, JMS, RPC, REST, RMI, UPnP). En outre, le modèle FraSCAti fournit l'adaptation et la gestion à l'exécution aux applications SCA. Ainsi, FraSCAti introduit des capacités réflexives au modèle SCA et permet l'introspection et la reconfiguration dynamique des applications SCA. Ces caractéristiques ouvrent de nouvelles perspectives pour apporter de l'agilité à SOA et la gestion des applications SCA à l'exécution.

Le modèle d'applications *soCloud* est basé sur l'extension du modèle SCA et le modèle FraSCAti que nous détaillons dans la section suivante.

4.4.2 Extensions du modèle SCA

La figure 4.4 montre le méta-modèle SCA simplifié. Nous proposons deux extensions <*implementation.contribution*> et <*annotation*> définies par le modèle d'applications *soCloud*.

15. http ://frascati.ow2.org/

L'extension <*implementation.contribution*> permet d'implémenter un composant avec une contribution et l'extension <*annotation*> permet d'exprimer des besoins non-fonctionnels. La modélisation d'applications *soCloud* est basée sur ces extensions du modèle SCA (voir Figure 4.4).

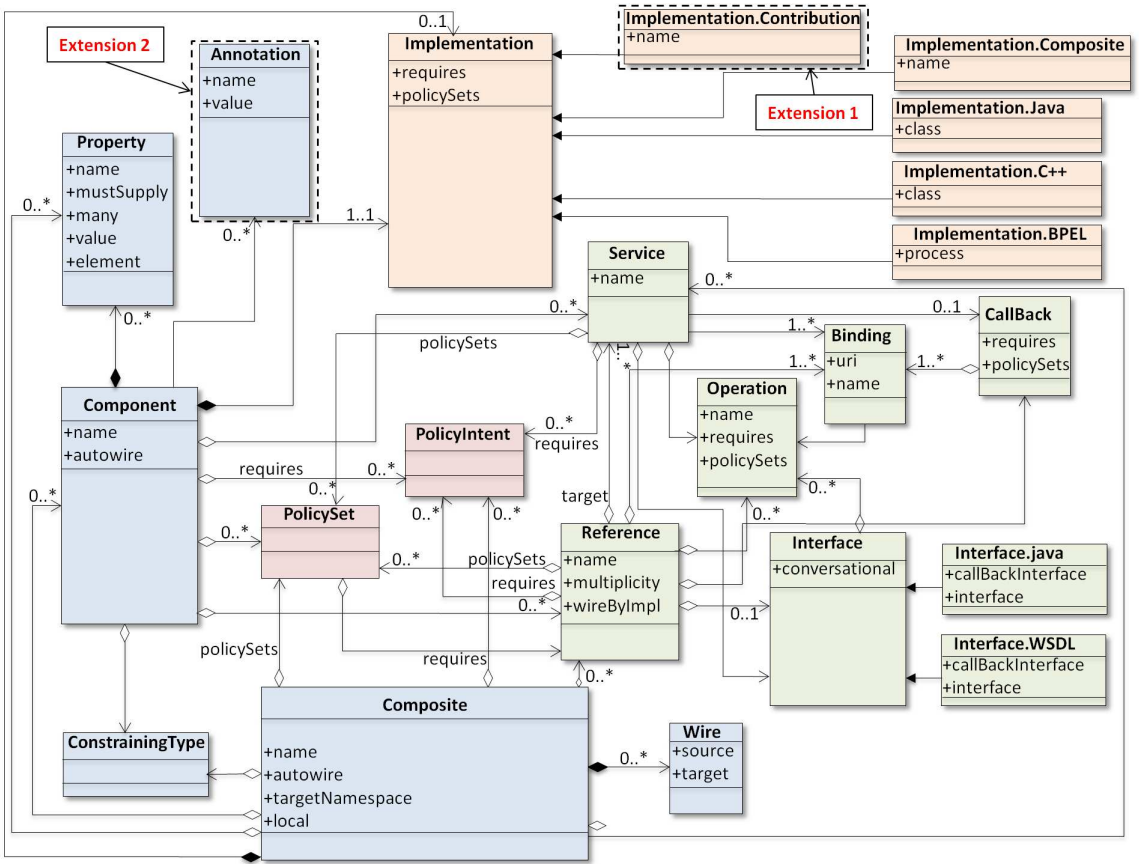

Figure 4.4 – Méta-modèle SCA et deux extensions.

4.4.2.1 Extension 1 : <implemention.contribution>

L'extension <*implementation.contribution*> permet d'implémenter un composant avec une contribution SCA. En effet, dans le modèle d'applications *soCloud*, chaque composant d'une application répartie est géré comme une unité d'exécution. Or le modèle SCA ne nous permet pas d'implémenter un composant comme unité d'exécution. Ainsi, un composant est implémenté par une contribution qui fournit un fichier binaire déployable. L'extension <*implemention.contribution*> à deux représentations : une représentation graphique 4.5 (voir figure 4.5) et une représentation XML ligne 1 du listing 4.1. Le symbole du fichier zip placé sur la représentation graphique (voir figure 4.5) permet de distinguer le composant annoté des autres composants ordinaires.

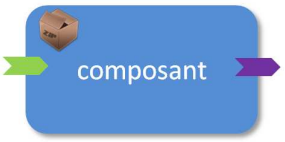

Figure 4.5 – Représentation graphique de l'extension <implemention.contribution>.

```
1 <implementation.contribution contribution="contribution.zip"/>
```

Listing 4.1 – Représentation XML de l'extension <implemention.contribution>.

4.4.2.2 Extension 2 : <annotation>

L'extension <*annotation*> permet d'associer des exigences non-fonctionnelles (par exemple, la localisation) à un composant. Nous avons choisi d'exprimer ces exigences via une annotation spécifique plutôt que de réutiliser la construction <property> qui est dédiée à la configuration d'attributs fonctionnels d'un composant. SCA fournit des *intents* qui sont des politiques pour configurer de façon déclarative des besoins non-fonctionnels tels que la sécurité, les transactions, la fiabilité. L'un des objectifs clés des intents SCA est de faire abstraction de la complexité associée à la spécification de la sécurité à des transactions à partir du code de l'application. L'utilisation des intents peut vite devenir contraignante lorsqu'on a besoin d'exprimer plusieurs besoins non-fonctionnels car ils nécessitent une définition explicite. Comparé aux intents, les annotations sont plus adaptées pour exprimer des besoins non-fonctionnels liés aux défis de l'*informatique multi-nuages* et peuvent être utilisées à des fins spécifiques. La séparation des préoccupations a été mise en évidence par [Dij82]. Le principe est de séparer différents aspects d'un programme, appelés préoccupations, lors de la conception. La séparation des préoccupations peut être mise en œuvre en utilisant la programmation orientée aspect [Kic01] ou simplement en utilisant de la modularité. Dans notre cas, il faut considérer la répartition des applications *soCloud* comme un aspect particulier et le traiter indépendemment du code métier. C'est pourquoi nous utilisons des annotations dans le modèle d'applications *soCloud*. Les annotations sont représentées sous trois formes : la représentation graphique (voir figure 4.6) avec le symbole "@" au dessus du composant, la représentation textuelle ligne 1 du listing 4.2 et la représentation XML ligne 2 du listing 4.2. Dans le listing 4.2, *@nom_annotation* et *valeur* désignent respectivement le nom et la valeur de l'annotation. L'annotation <*annotation*> est discutée plus en détail dans la section 4.4.4.1.

```
1 @nom_annotation = "valeur"
2 <annotation name="nom_annotation">valeur</annotation>
```

Listing 4.2 – Représentations textuelle et XML de l'extension <annotation>.

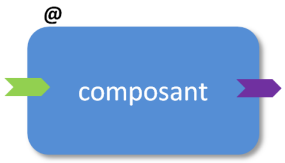

Figure 4.6 – Représentation graphique de l'extension <annotation>.

Nous avons décrit les concepts de base sur lesquels le modèle d'applications *soCloud* est fondé. Nous avons justifié le choix du modèle SCA et les extensions supplémentaires dont a besoin le modèle *soCloud*. La section suivante présente le modèle utilisé dans une application *soCloud*.

4.4.3 Modèle d'applications *soCloud*

Le méta-modèle présenté dans la figure 4.7 représente une vue globale du modèle d'applications *soCloud*. Ce méta-modèle est une extension du méta-modèle SCA. Nous nous restreignons à un méta-modèle canonique pour gérer des composants SCA sans détails techniques. La méta-classe *soCloud Application* représente l'élément racine du modèle *soCloud*. Elle regroupe en son sein, un ensemble fonctionnel cohérent, des composants (*Component*) non nécessairement liés. Une application *soCloud* peut être constituée d'un ou plusieurs *composants* qui correspondent à des unités d'exécution. Le *composant* est l'unité élémentaire de construction d'une application *soCloud*. La méta-classe *Inclusion* est utilisée pour structurer la composition de composants. Un *composant* peut offrir des fonctionnalités (Implementation) qui sont exposées en tant que services pour être utilisés généralement par d'autres composants ou clients. Chaque *composant* est implémenté par une contribution. Une contribution est un fichier binaire ou format ZIP. L'implantation est composée d'un ou plusieurs *artefacts*. Les fonctionnalités implémentées peuvent être paramétrées par des *propriétés* (*Property*). Un *composant* a zéro ou plusieurs *propriétés* (c.à.d qu'un *composant* peut ne pas avoir de propriété ou peut en avoir plusieurs). Une *propriété* est utilisée pour la configuration métier d'un composant. La méta-classe *annotation* permet de décrire des propriétés non-fonctionnelles sur un *composant*. Elle est utilisée notamment pour exprimer des contraintes sur un *composant*. Un *composant* a zéro ou plusieurs *annotations* (c.à.d qu'un *composant* peut ne pas être annoté ou avoir plusieurs annotations). La méta-classe abstraite *contrat* indique la sémantique de la relation entre deux composants connexes via un *service* ou une *référence*. Les méta-classes *service* et *référence* héritent de la méta-classe abstraite *contrat* (*Contrat*). La méta-classe connecteur (*Wire*) définit la liaison entre une *référence* et un *service* d'un composant. La méta-classe *service* est décrite au travers d'interfaces composées d'opérations (*Operation*) qui constituent le contrat de service. Un composant s'exécute sur une machine virtuelle. Une machine virtuelle peut contenir plusieurs composants qui s'exéxutent en son sein. Une machine virtuelle est localisée dans une zone géographique donnée.

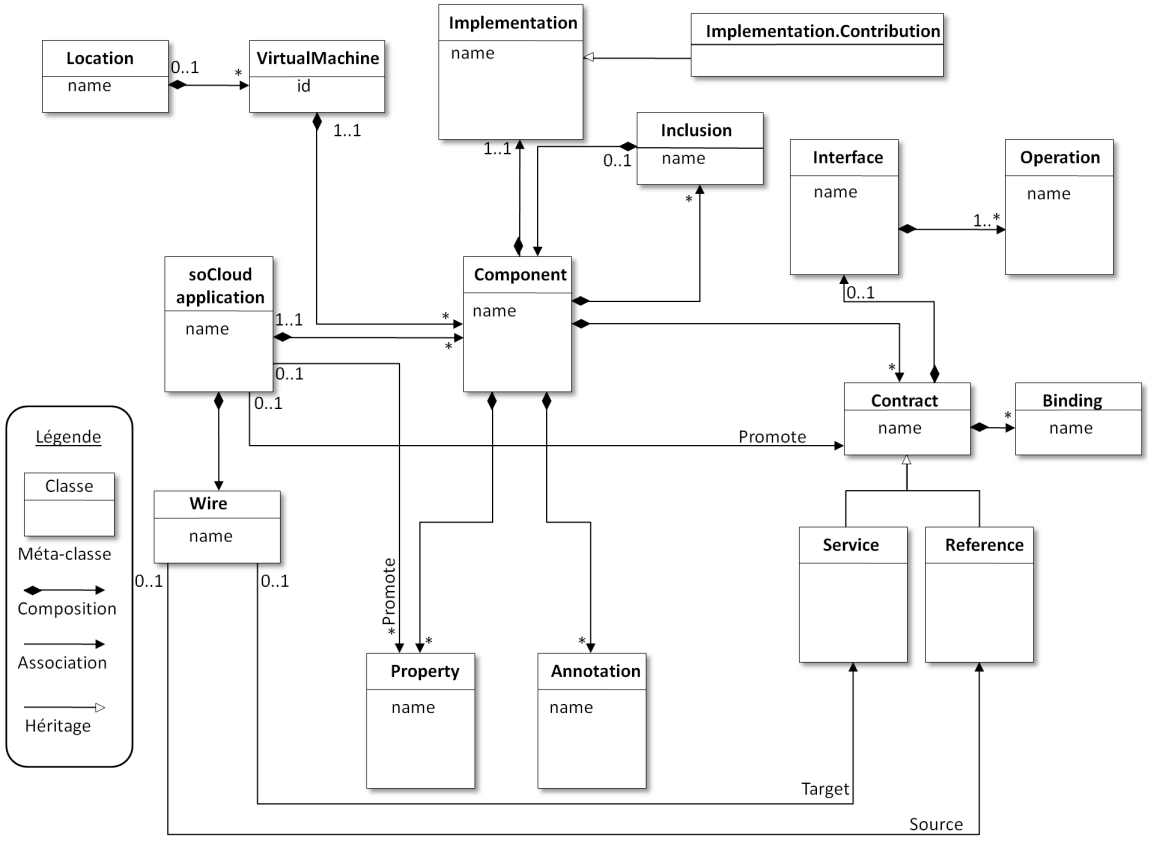

Figure 4.7 – Méta-modèle d'une application *soCloud*.

Nous avons présenté le modèle d'applications *soCloud*. La section suivante présente la description d'une application *soCloud*.

4.4.4 Description d'applications *soCloud*

4.4.4.1 Annotations d'applications *soCloud*

L'application trois-tiers décrite dans la section 4.2 peut être annotée comme le montre la figure 4.13 (page 86). A l'exécution, chaque composant peut avoir des contraintes spécifiques selon les besoins du développeur. Nous proposons une approche générique, basée sur les annotations, pour décrire les connaissances capturées dans la section 4.3. La notion d'annotation a été utilisée dans divers domaines tels que les gestionnaires de configuration logicielle [Cag95, Leb95, Was90], le web sémantique [Han03], en EMF [Lan11], les langages de programmation Java [Sun04], Python [Pyt06], C# [Mic07]. Les annotations sont considérées comme des informations complémentaires par rapport à l'information principale. Ces informations peuvent correspondre à un comportement, une mesure, une observation ou

un commentaire que l'on veut associer à une entité gérée par le système. Dans le cas d'une application *soCloud*, l'entité correspond à un composant et nous avons intégré la notion d'annotation dans le modèle. Cette intégration a pour objectifs :

— De permettre aux développeurs d'exprimer différents types d'informations. Une annotation peut contenir l'emplacement d'un composant, le type de machine virtuelle dont a besoin l'instance d'un composant, des règles d'élasticité, le nombre d'instances d'un composant à déployer, la manière dont plusieurs composants sont placés. Les informations d'une annotation doivent permettre de décrire de manière simple, complète et spécifique le modèle d'une application *soCloud*.
— D'annoter les composants dans le modèle d'une application *soCloud* (qui peut être constituée d'un ou plusieurs composants).
— D'être simple d'utilisation pour les développeurs, les annotations doivent être explicites pour ces derniers. Ils doivent être en mesure de déterminer quand, comment et sur quels composants les annotations seront placées.
— De capturer les informations automatiquement lorsqu'une situation spéciale ou un événement se produit dans le système (par exemple chaque fois que l'état d'une application est modifié, le temps prévu pour l'achèvement d'une activité est dépassé, la charge de travail est supérieure à un seuil, un composant est défaillant).

Comme décrit dans la section 4.3, le principe d'annotations est basé sur la séparation des préoccupations. Chaque type d'annotation adresse un sous-problème particulier et le développeur d'applications a la possibilité d'utiliser une combinaison appropriée de ces annotations selon l'architecture et les exigences de l'application. Cette séparation des préoccupations évite que deux annotations différentes soient incompatibles.

Les annotations d'une application *soCloud* sont similaires aux annotations Java [Sun04] dans le sens où elles sont des méta-informations, qui sont examinées par le système à l'exécution.

4.4.4.2 La représentation des annotations

Dans le modèle d'applications *soCloud*, les annotations peuvent avoir plusieurs représentations (graphique, textuelle et XML) (voir section 4.4.2). Une annotation est une instance d'un type défini par le développeur ou prédéfini dans le système dans lequel elle est exécutée. Elle est indépendante du composant sur lequel elle est créée. Une annotation existe seulement si elle annote un composant.

Dans le modèle *soCloud*, nous proposons quatre catégories d'annotations comme le décrit la figure 4.8.

1. Contraintes de placement. Elles sont exprimées sur des composants d'une application *soCloud* pour indiquer comment ces derniers doivent être placés. Elles sont exprimées avec deux types d'annotations. Le listing 4.3 montre leurs représentations graphiques. L'annotation *@location* à la ligne 1 du listing 4.3 permet de déployer le composant sur des hôtes

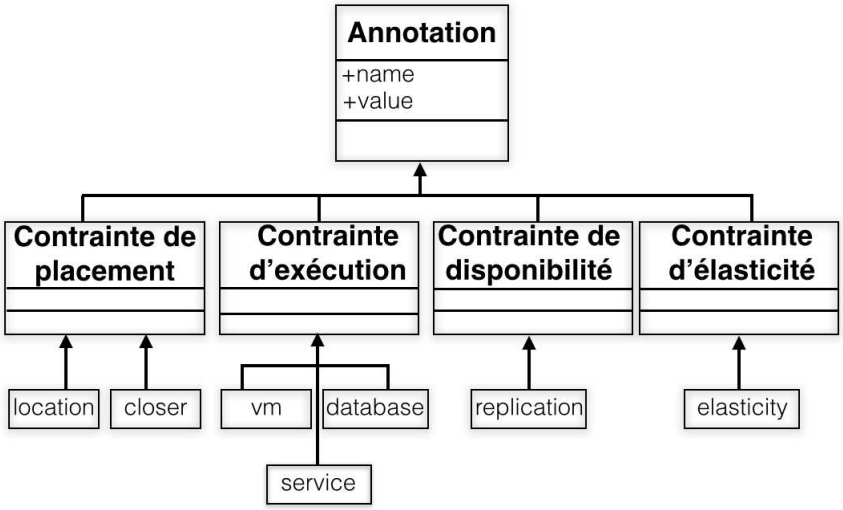

Figure 4.8 – Les annotations d'une application *soCloud*.

physiques situés dans une zone géographique donnée. L'objectif de l'annotation *location* est d'offrir la possibilité d'exprimer le lieu géographique d'un composant, par exemple le placer le plus prêt possible des utilisateurs finaux. Toutefois, lorsqu'il n'y a pas d'annotation *@location* sur un composant, ce dernier peut être placé n'importe où, chez n'importe quel fournisseur de nuage par le système *soCloud*. Les valeurs que peut prendre l'annotation *@location* sont soit de type localisation (par exemple France, Amérique) soit de type nom de fournisseur de nuage (par exemple Amazon, Windows Azure). La figure 4.10 montre le graphe acyclique dirigé (DAG) des valeurs que peut prendre l'annotation *@location* et sa granularité. Le graphe montre différents niveaux de granularité : **(1)** indique qu'on peut placer le composant annoté dans un continent, **(2)** indique que le composant annoté peut être placé dans un pays ou un état et **(3)** indique que la localisation du composant est une ville. Par exemple lorsqu'un composant porte deux annotations (*@location*="Europe" et *@location*= "France"), elles indiquent qu'il faut placer ce dernier soit en Europe soit en France. Lorsque plusieurs valeurs des annotations *@location* placées sur un même composant ont un lien d'**inclusion** (dans notre exemple précédent la France est incluse dans Europe) leur combinaison se traduit par un "ou" logique. Dans le cas où les valeurs des annotations *@location* n'ont pas de lien d'**inclusion**, par exemple les annotations *@location*="France" et *@location*="Allemagne" sur un même composant se traduisent par un placement de ce dernier dans deux zones géographiques distinctes à savoir la France et l'Allemagne. Ainsi, placer plusieurs annotations *@location* sur un même composant qui n'ont pas de lien d'**inclusion** est équivalent à un "et" logique. On peut aussi donner le nom d'un fournisseur de nuage comme valeur à l'annotation *@location* ou la région couverte par ce dernier. Dans ce dernier cas, comme le montre la figure 4.10, annoter un composant avec *@location* = "Amazon Irlande" ou *@location* = "Irlande" a la même interprétation du point de vue placement géographique. Lorsqu'un composant porte l'annotation *location*="Europe", ce dernier est déployé dans n'importe quel

pays d'Europe. De la même façon, l'annotation *@location*="France" placée sur un composant indique que ce dernier peut être déployé dans n'importe quelle ville française.

```
1@location = "valeur_location"
2@closer = "valeur_closer"
```

Listing 4.3 – Annotations des contraintes de placement.

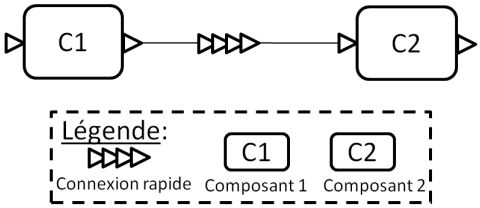

Figure 4.9 – Annotation de contraintes de placement : pour exprimer la proximité de deux composants.

L'annotation *closer* du listing 4.3 permet de placer plusieurs composants connexes le plus proche possible en termes de latence réseau dans un environnement *multi-nuages*. L'objectif de cette annotation est d'avoir une politique permettant d'exprimer la contrainte de placement (en terme de temps de réponse entre plusieurs composants) que peuvent exiger deux ou plusieurs composants d'une même application répartie à large échelle. La figure 4.9 montre deux composants connexes qui sont proches en terme de latence réseau. On peut annoter un composant en utilisant plusieurs fois *@closer* sur le même composant, par exemple, l'annoter avec *@closer="nom_composant2"* et *@closer="nom_composant3"* ont une écriture équivalente à *@closer="nom_composant2 nom_composant3"*. L'annotation *@closer="nom_composant2 nom_composant3"* sur un même composant indique qu'il faut placer ce dernier à proximité des composants *composant2* et *composant3*. Lorsqu'on veut placer un composant à côté d'autres composants qui se situent dans différentes zones géographiques, nous utilisons l'algorithme de K-proche voisin plus proches (K-nearest neighbor) [Kel85]. Cet algorithme nous permet de déterminer la zone géographique la plus proche (en terme de temps de réponse) dans laquelle le composant sera placé. D'une manière générale, l'agorithme K-proche voisin utilise la distance qui sépare chaque nœud comme valeur en entrée. Dans le cas de notre modèle la distance est remplacée par le temps de réponse. Nous avons choisir d'utiliser l'algorithme K-proche voisin, parcequ'il est plus adapté aux composants distribués dynamiques qui changent fréquemment de position géographique.

2. Contraintes d'exécution. Elles décrivent les ressources nécessaires dont a besoin un composant d'une application *soCloud* pour fonctionner. Elles sont exprimées avec deux types d'annotations. Le listing 4.4 montre leurs représentations graphiques. L'annotation *@vm* en ligne 1 du listing 4.4 décrit les ressources matérielles dont a besoin un composant pour s'exécuter. Le différentes valeurs que peuvent prendre l'annotation *@vm* sont des caractéristiques de machine virtuelle décrit dans le tableau 4.1. Ce tableau classifie le type de machine vir-

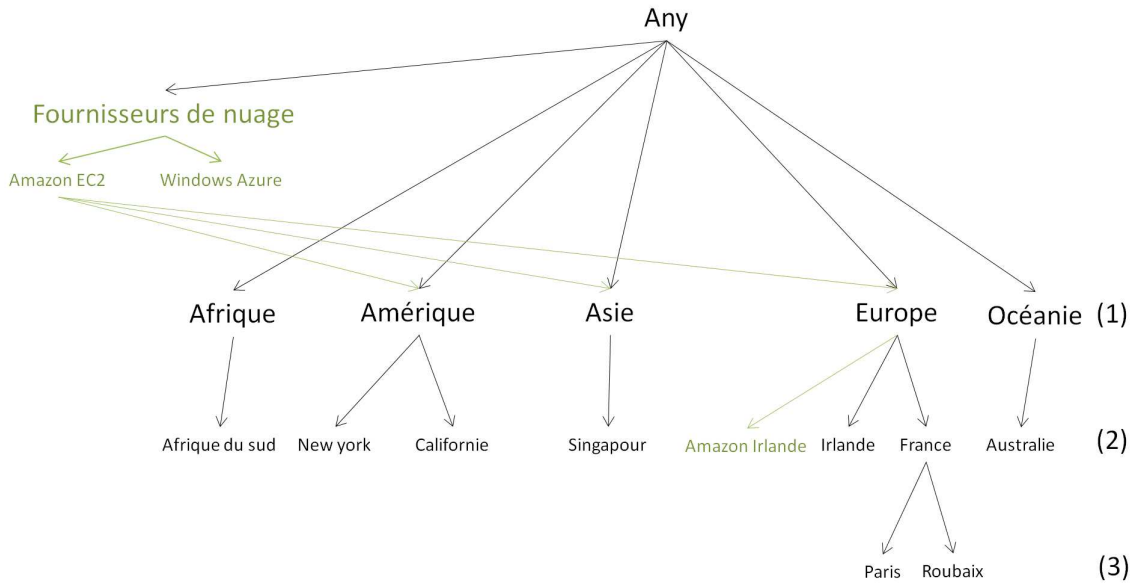

Figure 4.10 – Graphe acyclique dirigé (DAG) de l'annotation @location.

tuelle qui va de *micro* à *large* en fonction de leur puissance matérielle. Dans le tableau 4.1, nous avons repris la classification de Amazon EC2. Lorsque cette annotation est absente sur un composant, le système déploie ce dernier sur une machine virtuelle de type *micro*. L'annotation *@vm* en ligne 2 du listing 4.4 permet d'ajouter un paramètre supplémentaire *nom_OS* séparé par le symbole "->". Le paramètre *nom_OS* indique le nom du système d'exploitation que l'on veut sur la machine, il n'est pas obligatoire (voir ligne 1 du listing 4.4). Si le paramètre *nom_OS* n'est pas précisé, le système d'exploitation par défaut est de type Linux. L'annotation *@database* ligne 3 du listing 4.4 spécifie le nom de la base de données dont a besoin un composant. Les valeurs que peut prendre l'annotation *@database* sont les noms de bases de données comme MySQL, PostgreSQL, MongoDB et Cassandra. Le paramètre *version* indique la version de la base de données. Le symbole "->" délimite le nom de la base de donnée et sa version est optionnel. Si la version de la base données n'est pas précisée, le système *soCloud* fournit une version par défaut. L'annotation *@database* ne peut être placée plusieurs fois sur un même composant. L'absence de l'annotation *@database* sur un composant indique qu'il ne faut pas instancier de base de données. L'annotation *@service* ligne 4 du listing 4.4 permet de faire appel à des services externes comme la méssagerie, la base de données. De la même façon que l'annotation *@database*, l'annotation *service* (voir ligne 4 du listing 4.4) permet de spécifier son nom et sa version. L'absence de l'annotation *@service* sur un composant indique qu'il ne faut pas instancier de service.

```
1@vm = "type_vm"
2@vm = "type_vm -> nom_OS"
3@database = "nom_base_de_donnees -> version"
4@service = "nom_service -> version"
```

Listing 4.4 – Annotations des contraintes d'élasticité.

Tableau 4.1 – Caractéristiques de machines virtuelles

Resource name / Instance type	micro	small	medium	large	xlarge
cpu(GHZ)	1	1	2	2	4
cpu(# cores)	1	1	1	2	4
memory(GB)	0.615	1.7	3.5	7.5	14
disk(GB)	50	160	300	850	1024

3. Contraintes de disponibilité. Elles précisent le nombre d'instances du composant d'une application *soCloud* qui doivent être déployés dans un environnement *multi-nuages*. L'objectif de cette annotation est d'assurer la disponibilité d'un composant en le réplicant. Ainsi, lorsqu'une panne survient sur une instance de ce dernier, les autres prendront le relai. Les valeurs que peut prendre l'annotation *@replication* sont des entiers naturels, supérieurs ou égaux à 1. Par exemple, lorsque l'annotation *@replication="5"* est posée sur un composant, le système *soCloud* place automatiquement les cinq instances de ce composant derrière un équilibreur de charge comme le montre la figure 4.11. Lorsque l'annotation *@replication* n'est pas placée sur un composant, une seule instance de ce dernier est déployée. Autrement dit, l'absence de l'annotation *@replication* sur un composant est équivalent à mettre *@replication="1"* sur ce dernier.

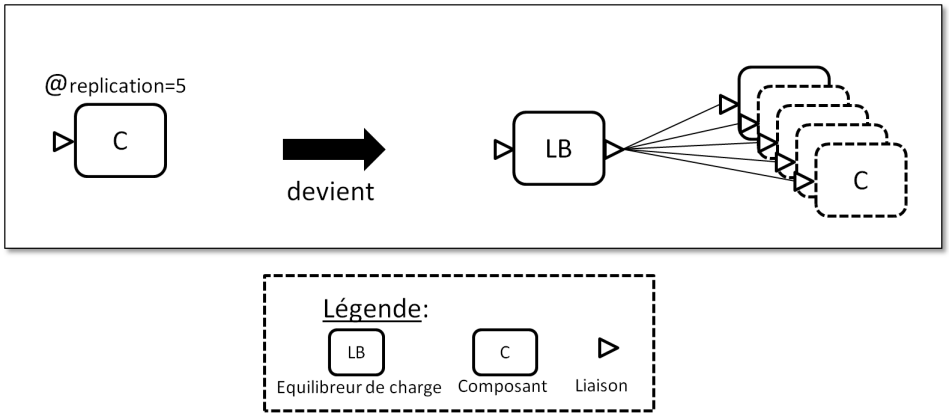

Figure 4.11 – Réplication d'un composant.

4. Contraintes d'élasticité. Elles définissent des règles d'élasticité spécifiques qui doivent être appliquées au composant déployé dans un environnement *multi-nuages*. Ainsi

on a la possibilité d'exprimer différentes contraintes d'élasticité sur différents composants d'une même application. La figure 4.12 illustre le mécanisme d'élasticité d'un composant. Elle montre le composant qui est répliqué de manière automatique pour supporter une charge de travail accrue, puis revient à l'état initial d'une seule instance lorsque la charge de travail se stabilise. Le listing 4.5 montre une représentation graphique de l'annotation *@elasticity*. L'objectif de cette annotation est de pouvoir exprimer une règle d'élasticité spécifique sur chacun des composants d'une application *soCloud*, car le besoin de passage à l'échelle d'un composant à l'autre peut varier. L'absence de l'annotation *@elasticity* sur un composant laisse la gestion de l'élasticité de ce dernier au système *soCloud*. L'annotation *@elasticity* ne peut être posée plus d'une fois sur le même composant. Toutefois, le développeur a la possibilité d'exprimer plusieurs règles d'élasticité dans la même annotation *@elasticity*. Le langage d'expression de ces règles d'élasticité est présentée en section 4.5.

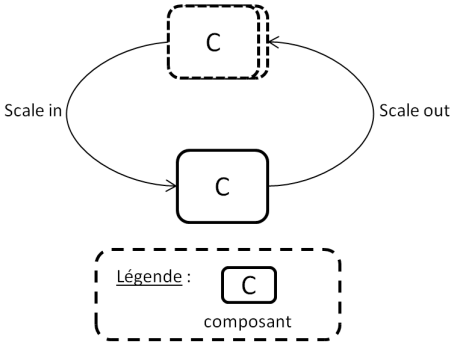

Figure 4.12 – Représentation du mécanisme d'elasticité d'une application.

```
1@elasticity = "scaling up when responseTime > 4s"
```

Listing 4.5 – Annotation de contraintes d'élasticité.

Considérons notre scénario de cas d'utilisation décrit dans la section 4.2. Selon le besoin d'expérimentation de l'application, ces composants peuvent être annotés comme le montre la figure 4.13.

La figure 4.13 illustre un exemple d'annotations de notre application trois-tiers.

Le composant *Présentation* porte trois annotations : **@location**, **@replication** et **@elasticity**. L'annotation **@location** = "Singapour" décrit que le composant Presentation doit être déployé à Singapour. L'annotation **@replication** = "3" spécifie le nombre d'instances du composant *Présentation* à déployer. L'annotation **@elasticity="scaling in when totalCost(computeCost, 24 h) > 200"** décrit que le système doit retirer de la machine virtuelle lorsque le coût total d'exploitation en 24 heures est supérieur à 200 euros. L'objectif de cette règle est de définir un seuil maximal (égal à 200 euros) du coût de fonctionnement journalier du composant.

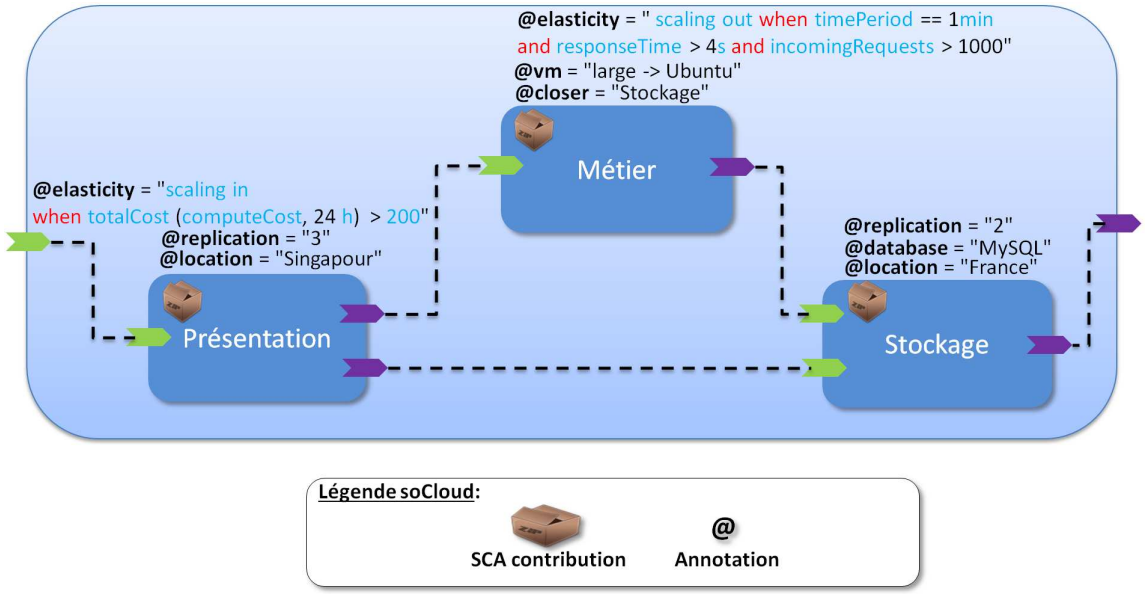

Figure 4.13 – Annotation de l'application trois-tiers.

Le composant *Métier* porte trois annotations : **@closer**, **@vm** et **@elasticity**. L'annotation **@closer** = "Stockage" spécifie que le composant *Métier* doit être proche en terme de latence réseau du composant Stockage comme décrit sur la figure 4.13. L'annotation **@vm** = "large -> Ubuntu" décrit le type de machine virtuelle et le système d'exploitation installé sur cette dernière. L'annotation **@elasticity** = " scaling out when timePeriod == 1 min and responseTime > 4s and incomingRequests > 1000" décrit la règle d'élasticité spécifique qui doit être appliquée au composant *Métier*. Cette règle stipule au système d'ajouter une nouvelle machine virtuelle lorsque le temps de réponse est supérieur à quatre secondes sur un intervalle de temps d'une minute et le nombre de requêtes entrantes est supérieur à 1000. Notons que le développeur n'a pas besoin de spécifier le *Scaling in* et *Scaling down* car en l'absence de ces deux opérations, les mécanismes de *Scaling in* et *Scaling down* sont pris en charge de manière automatique par le système *soCloud*. Toutefois, ces deux opérations peuvent être définies de manière explicite.

Le composant *Stockage* porte trois annotations : **@location** = "France", **@database** = "MySQL" et **@replication** = "2". L'annotation **@location** = "France" spécifie que le composant Stockage doit être déployé en France. L'annotation **@database** = "MySQL" décrit le type de base de données qu'utilise le composant Stockage. L'annotation **@replication** = "2" indique le nombre d'instances du composant *Stockage* à déployer.

4.4.4.3 Conditionnement d'applications *soCloud*

Les composants d'une application *soCloud* peuvent être destinés à un déploiement dans différents environnements d'exécution. Il est donc logique de les gérer comme des unités d'exécution indépendantes. Les applications réparties sont généralement divisées en un ensemble de composants qui dépendent les uns des autres et ne peuvent pas fonctionner de manière isolée. D'autre part, pour une application répartie, les composants peuvent être déployés dans plusieurs environnements d'exécution situés dans différentes zones géographiques. Dans les modèles d'assemblage qui existent comme dans la technologie Java EE, les applications sont conditionnées dans des archives (WARs, EARs) autonomes. Bien que ce modèle de déploiement fonctionne pour de nombreuses applications, pour beaucoup d'autres, il impose des limites importantes (lorsque les artefacts doivent être partagés entre les applications).

Il existe une différence importante, entre SCA et Java EE : SCA fournit un langage d'assemblage qui peut être utilisé au dessus des modèles de composants et d'exécution existants. Notre approche est basée sur l'extension du modèle SCA (voir section 4.4.2). L'assemblage d'une application *soCloud* doit offrir un moyen simple pour grouper les composants de celle-ci afin qu'ils puissent être traités de manière automatique comme un ensemble logique appartenant à une même application. Ainsi, on peut mettre à jour des composants d'une application sans le redéploiement de celle-ci. Toute application *soCloud* est assemblée sous la forme d'une contribution (voir section 4.4.2). Cette contribution contient en son sein une ou plusieurs contributions selon l'architecture (répartie ou non) de l'application (voir figure 4.14) et un fichier de description.

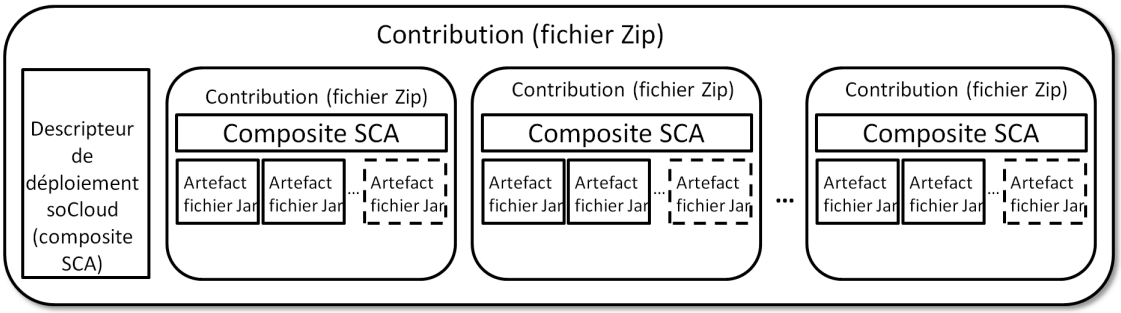

Figure 4.14 – Assemblage des composants d'une application *soCloud*.

En considérant l'application répartie trois-tiers, elle est assemblée sous la forme d'une contribution qui contient trois contributions et un fichier de description qui permet de décrire l'architecture globale. Les contributions que contient l'application répartie trois-tiers correspondent aux composants de cette dernière. Notre modèle d'assemblage permet de conditionner tout en ayant une représentation de haut-niveau des composants d'une application, ce qui facilite le déploiement de ces derniers.

Le listing 4.6 présente le fichier de description de l'application trois-tiers qui contient trois composants *Presentation* (lignes 2-12), *Metier* (lignes 13-25) et *Stockage* (lignes 26-32). Les lignes 4, 5 et 16 décrivent les liaisons qui existent entre les trois composants. Les lignes 6, 7, 8-11, 17, 18-23, 29, 30 et 31 correspondent aux représentations XML des annotations. Les lignes 3, 14 et 27 correspondent à notre extension du modèle SCA qui représentent respectivement les contributions *Presentation*, *Metier* et *Stockage*. Les contraintes de placement pour les composants *Presentation*, *Metier* et *Stockage* sont exprimées sur les lignes 6, 24 et 31. Une contrainte d'exécution est exprimée à la ligne 17 pour le composant *Metier*. Le type de base de données utilisé par le composant *Stockage* est exprimé à la ligne 30. Le nombre d'instances à déployer des composants *Presentation* et *Stockage* est exprimé respectivement à la ligne 7 et 29. Nous avons adopté une approche événementielle condition-action pour la spécification des règles d'élasticité. Ces règles s'appliquent aux événements de surveillance obtenues par le système *soCloud*, par exemple, le temps de réponse, le nombre de requêtes entrantes, etc. Comme décrit sur les lignes 8-11 et 18-23, deux règles d'élasticité spécifiques sont portées respectivement par les composants *Presentation* et *Metier*.

```
<composite name="Application-3-tier">
  <component name="Presentation">
    <implementation.contribution contribution="presentation.zip"/>
    <reference name="compute" target="Metier/compute"/>
    <reference name="storage" target="Stockage/storage"/>
    <annotation name="location">Singapour</annotation>
    <annotation name="replication">3</annotation>
    <annotation name="elasticity">
        scaling in
        when (totalCost(computeCost,24 h) > 200)
    </annotation>
  </component>
  <component name="Metier">
    <implementation.contribution contribution="metier.zip"/>
    <service name="compute"/>
    <reference name="storage" target="Stockage/storage"/>
    <annotation name="vm">large -> Ubuntu</annotation>
    <annotation name="elasticity">
        scaling out
        when (timePeriod == 1 min
                and responseTime > 4s
                and incomingRequests > 1000)
    </annotation>
    <annotation name="closer">Stockage</annotation>
  </component>
  <component name="Stockage">
    <implementation.contribution contribution="stockage.zip"/>
    <service name="storage"/>
    <annotation name="replication">2</annotation>
    <annotation name="database">MongoDB</annotation>
    <annotation name="location">France</annotation>
  </component>
</composite>
```

Listing 4.6 – Le descripteur de l'application trois-tiers *soCloud*.

4.5 Langage dédié aux règles d'élasticité

Les règles d'élasticité d'une application *soCloud* sont exprimées en utilisant un langage dédié ou Domain Specific Language [Van02]. Les langages dédiés sont des langages qui peuvent être associés à un domaine de problème spécifique. Le langage dédié fournit des

concepts de haut-niveau liés à l'annotation d'élasticité (**@elasticity**) du modèle *soCloud*. Il est exprimé à l'intérieur de l'annotation **@elasticity**. Ce langage permet d'exprimer facilement des conditions riches d'événements et de corrélation. Nous pouvons exprimer des règles sur une période. Grâce aux mécanismes et à la facilité d'expressivité, le langage réduit ainsi l'effort de développement requis pour mettre en place un système capable de réagir à des situations complexes.

4.5.1 Concepts du langage dédié

Notre langage d'élasticité est un langage guidé par les événements ce qui signifie que le système déclenche une règle basée sur ces derniers qu'on spécifie dans la règle. Chaque événement est associé à un contexte. Le contexte de l'événement est transmis aux règles qui lui sont associées. La condition et l'action des règles peuvent accéder aux informations contenues dans le contexte de l'événement. Le langage d'élasticité repose sur quatre concepts :

— *Création d'événement* : c'est la spécification d'un événement dans le langage d'élasticité, qui amène le système *soCloud* à se déclencher à chaque fois que cet événement se produit. Le listing 4.7 montre comment on crée un évènement. Le but de la création d'un nouveau événement est d'offrir la possibilité au développeur de créer ses propres événements à surveiller.
— *Événement prédéfini* : est basé sur les variables et fonctions d'environnement prédéfinies dans le système. Les tableaux 4.4 et 4.5 décrivent respectivement les variables et fonctions d'environnement prédéfinies.
— *Opérateurs* : comparent deux opérandes dans une expression. Le tableau 4.3 liste les opérateurs utilisés dans le langage.
— *Actions* : fournit des directives pour décrire comment atteindre certains objectifs et dans quelles conditions. Le tableau 4.2 montre la liste des actions.

La grammaire du langage dédié est décrite dans la section 4.5.3.

```
1  create EVENT EVENT_NAME as([property_name property_type]
2 , [...])
```

Listing 4.7 – Description d'un événement à observer.

4.5.1.1 Syntaxe du langage dédié

Le listing 4.8 illustre la syntaxe générale de notre langage dédié. Une action est déclenchée lorsqu'une ou plusieurs conditions sont remplies. Le mot clé *"when"* permet d'exprimer le conditionnel. La combinaison de plusieurs conditions se fait en utilisant les opérateurs logiques *"and"*, *"or"* et *"not"* (voir tableau 4.3). La condition est exprimée en utilisant les propriétés et les opérateurs relationnels du tableau 4.3.

Tableau 4.2 – Liste des actions.

Actions	Description
scaling up	Déclenche une élasticité de type verticale (comme décrit dans le chapitre 3).
scaling out	Déclenche une élasticité de type horizontale (comme décrit dans le chapitre 3).
scaling in	Ramène le système à l'état initial (élasticité de type horizontale) c'est-à-dire l'état dans lequel il était avant le passage à l'échelle.
scaling down	Ramène le système à l'état initial (élasticité de type verticale) c'est-à-dire l'état dans lequel il était avant le passage à l'échelle.
maximize	Maximise la disponibilité, la performance, etc.
minimize	Minimise la disponibilité, la performance, etc.

Tableau 4.3 – Liste des opérateurs.

Opérateur	Syntaxe
Opérateurs relationnels	== ; > ; < ; >= ; <= ; != ; not
Opérateurs logiques	or ; and

```
1  action when ( conditions )
```

Listing 4.8 – Expression d'une action.

```
1  scaling up when (average(cpuUsage, 120s) > 80%)
```

Listing 4.9 – Expression d'une action.

La règle du listing 4.9 augmente la capacité du processeur lorsque le pourcentage d'utilisation de ce dernier sur une période de 2 minutes est supérieur à 80%. Lorsque l'action "scaling up" n'est pas suivie du volume de ressources à ajouter, le système *soCloud* calcule ce volume de manière automatique.

Le langage offre la possibilité d'exprimer des règles qui peuvent se déclencher dans le temps. C'est utile par exemple, pour une application dont on connait les habitudes d'utilisation (ou la fréquence d'utilisation) des utilisateurs finaux. Le listing 4.10 décrit la manière dont on peut exprimer ces types de règles.

```
1  action at (time)
```

Listing 4.10 – Expression d'une action.

Dans le listing 4.10, time correspond à l'heure qui est sous la forme : *hh mm jj MMM day*.

— *hh* : représente l'heure (de 0 à 23).

— *mm* : représente les minutes (de 0 à 59).

Tableau 4.4 – Liste des variables d'environnement prédéfinies.

Variable	**Description**
responseTime	Le temps de réponse d'un composant.
timePeriod	Représente un intervalle de temps d'un événement.
incomingRequest	Le nombre de requêtes entrantes dans la file d'attente.
concurrentRequest	Le nombre de requêtes concurrentes.
cpuUsage	La quantité de cpu consommée par un composant.
memUsage	La quantité de mémoire utilisée par un composant.
diskUsage	La quantité de disque utilisée par un composant.
networkUsage	La bande passante consommée par un composant.
computeCost	Le coût du fournisseur de nuage.
numberOfUsers	Le nombre d'utilisateurs connectés.

Tableau 4.5 – Liste des fonctions d'environnement prédéfinies.

Function	**Description**
average	Calcule la moyenne sur une période de temps.
availability	Calcule la disponibilité du composant sur un intervalle de temps.
violation	Vérifie si la contrainte envoyée est en conflit avec une autre pour la même application.
enable	Vérifie si une règle d'élasticité est activée ou non.
totalCost	Le coût total du composant.

— *jj* : représente le jour du mois (de 1 à 31).
— *MMM* : représente le numéro du mois (de 1 à 12).
— *day* : représente le nom de jour (Sunday ; Monday ; Tuesday ; Wednesday ; Thursday ; Friday ; Saturday).

Pour chaque unité de temps (minute/heure/jour) les notations sont possibles :

— * : à chaque unité de temps.
— *3-5* : les unité de temps (3,4,5).
— *3,7* : les unités de temps 3 et 7.

Par exemple, une application peut être fortement utilisée ou sollicitée durant les heures de bureau, ou à des intervalles de temps précis.

```
scaling in 2 at (20:00)
```

Listing 4.11 – Expression d'une action à une heure précise.

La règle du listing 4.11 retire 2 nœuds à 20h00 tous les jours.

```
scaling in 5 at (7:00 Friday)
```

Listing 4.12 – Expression d'une action à une date et heure précise.

La règle du listing 4.12 retire 5 nœuds le vendredi à 7h00.

Notre langage dédié permet aussi d'exprimer plusieurs actions en utilisant le mot clé *"CASE"* comme illustré dans le listing 4.13. Dans ce cas, pour chaque condition exprimée, une action se déclenche. Ainsi nous avons la possibilité de déclencher plusieurs actions à la fois. Par défaut si aucune condition n'est vérifiée, une action par défaut est déclenchée.

```
CASE condition1 action1 when (conditions)
CASE condition2 action2 when (conditions)
...
CASE conditionN actionN when (conditions)
DEFAULT action x
```

Listing 4.13 – Expression de plusieurs actions.

4.5.2 Les niveaux d'expression du langage dédié

Dans ce travail, l'élasticité vise non seulement les ressources et leur capacité à passer l'échelle, mais aussi leurs relations avec différents types de facteurs comme le coût, la qualité et la capacité à obtenir le meilleur rapport qualité prix.

Pour décrire quel type de contrôle d'élasticité pourrait être nécessaire et la complexité des exigences d'élasticité, nous avons examiné les exigences d'élasticité de différents types d'applications. Considérons notre application trois-tiers décrite dans la figure 4.13. A l'exécution, chaque composant a des besoins spécifiques de passage à l'échelle selon les exigences du développeur. En fonction du type d'élasticité verticale ou horizontale (voir chapitre 3 page 15) défini par le développeur, les composants peuvent être redimensionnés individuellement, soit en créant plus d'instances du composant dans différentes machines virtuelles, soit en allouant plus ou moins de ressources informatiques. Les exigences d'élasticité contiennent des conditions suffisantes à l'application pour qu'elle soit élastique. Ainsi, l'élasticité peut être exprimée à différents niveaux par différents types d'utilisateurs (développeurs, administrateurs). Par exemple, le listing 4.14 décrit une règle d'élasticité qui permet de diminuer la disponibilité d'un composant lorsque le coût total d'exploitation est supérieur à 900 euros. Ainsi, un composant par exemple peut voir sa disponibité baissée de 99,9% à 99,7%.

```
1 minimize availability when (totalCost(computeCost,24 h) > 900)
```

Listing 4.14 – Minimiser la disponibilité d'une application.

Dans le listing 4.15, la règle d'élasticité améliore la performance du composant lorsque le temps de réponse moyen de ce dernier sur un intervalle de 60 secondes est supérieur à 3 secondes et le nombre de requêtes entrantes est supérieur à 1000.

```
1 maximize performance when (average(responseTime,60 s) > 3 s
2 and incomingResquest > 1000)
```

Listing 4.15 – Maximiser la performance d'une application.

D'autre part, le développeur a besoin d'exprimer les exigences d'élasticité à différentes granularités, dont les spécifications seront appliquées à différents niveaux, par opposition aux approches habituelles dans l'allocation et la réallocation [Cha10, Mor11a, Han12] des ressources. Pour y parvenir, les contrôles d'élasticité doivent être supportés à différents niveaux :

— Niveau application : les exigences de l'élasticité peuvent être appliquées sur la disponibilité globale de l'application.
— Niveau composant : le développeur peut spécifier des exigences différentes en fonction du type de composant. Par exemple, la nature des exigences requises pour un composant unité de traitement peut être différent de celui d'un composant frontal.

Le listing 4.16 décrit une règle qui permet de réduire le coût d'exploitation d'un composant lorsque sa disponibilité sur une période de 72 heures est supérieur à 99.9%.

```
1 minimize computeCost when (average(availability,72 h) > 99.9%)
```

Listing 4.16 – Minimiser le coût d'un composant.

La règle d'élasticité du listing 4.17 augmente ou diminue les ressources matérielles (cpu et mémoire) d'une machine virtuelle selon deux cas. Premier cas, augmentation des ressources matérielles si la disponibilité est inférieure à 97% et la consommation en cpu est supérieure à 90% , la consommation en mémoire est supérieure à 80%. Deuxième cas, diminution des ressources matérielles lorsque la disponibilité est supérieure à 99% et la consommation en cpu est inférieure à 20%, la consommation en mémoire est inférieure 30%.

```
1 CASE availability < 97% scaling up
2 when (cpuUsage > 90% and memUsage > 80%)
3 CASE availability > 99% scaling down
4 when (cpuUsage < 20% and memUsage < 30 %)
```

Listing 4.17 – Contrôle de l'élasticité sur un composant.

Le développeur doit avoir la possibilité de spécifier le comportement de l'application, plus précisément de quelle manière elle doit passer à l'échelle automatiquement en oscillant entre les trois axes : ressources, coûts et qualité.

4.5.2.1 Les ressources

L'informatique en nuage est une solution de virtualisation qui permet d'allouer et de retirer des ressources de manière automatique. Comme décrit dans la section 2.1, l'élasticité peut être exprimée en vertical (on augmente la capacité cpu, mémoire, disque dur, bande passante de la machine virtuelle sur laquelle l'application ou le composant de l'application s'exécute) ou en horizontal (on ajoute d'autres machines virtuelles sur laquelle l'application ou les composants seront déployés) au moment de la montée en charge. Ainsi, selon les besoins du développeur d'application, il peut exprimer une élasticité verticale (ligne 4 du listing 4.18) ou horizontale.

4.5.2.2 Le coût

L'informatique en nuage est basée sur le modèle "payez uniquement ce que vous consommez". Dans ce modèle, le coût est un facteur très important lorsqu'une application est déployée. Le développeur d'application doit pouvoir exprimer des contraintes sur une application ou un ou plusieurs de ses composants dans le but de contrôler sa consommation en terme de coût. Par exemple, comme le montre le listing 4.18, la condition de mise à l'échelle (la fréquence moyenne du cpu supérieur à 80% durant 60 secondes) n'est pas appliquée tant que le coût total d'exploitation sur une période de 24 heures est supérieur à 200 euros.

```
1 scaling up when (totalCost(computeCost,24 h) < 200
2 and average(cpuUsage, 60 s) > 80%)
```

Listing 4.18 – L'expression du contrôle du coût total d'une application.

4.5.2.3 La qualité

La performance d'une application peut varier à n'importe quel moment. Le développeur qui connait les limites de son application en terme de performance, peut avoir besoin de surveiller différents paramètres. Ainsi, il peut exprimer une contrainte spécifique sur la qualité qu'exige l'application ou les composants de cette dernière. La qualité peut être exprimée sur le temps de réponse de l'application ou du composant, la disponibilité, la performance.

4.5.3 La grammaire

Cette section récapitule la grammaire de notre langage dédié décrit précédemment.

```
1  expressions ::= [statement+]
2  statement ::= multiple? expression newline
3  expression ::= [action] separator LPAREN [condition+] RPAREN
4  condition ::= word operator value
5  action ::= ( scaling up | scaling out | scaling in |
6  scaling down | minimize | maximize) | value
7  operator ::= ( "<" | "==" | ">" | "<=" | ">=" | "!=" |"not" )
8  separator ::= ( "WHEN" | "when" | "AT" | "at" |  "FOR" | "for" )
9  multiple ::= ( "CASE" | "case" | "DEFAULT" | "default" )
10 key ::= ( "and" | "AND" | "or" | "OR" )
11 value ::= alphanumeric*
12 word ::= alpha*
13 LPAREN ::= "("
14 RPAREN ::= ")"
15 newline ::= "\n"
```

Listing 4.19 – Grammaire BNF du langage d'élasticité.

4.6 Conclusion

Dans ce chapitre, nous avons présenté notre première contribution qui constitue en un modèle basé sur l'extension du standard SCA pour concevoir, implémenter et assembler une application orientée services pour un environnement *multi-nuages*. L'objectif du modèle *soCloud* est de fournir une solution d'ingénierie logicielle permettant de concevoir et de décrire de manière complète et spécifique l'architecture d'une application orientée services et ces exigences non-fonctionnelles dans un contexte *multi-nuages*. Nous avons analysé les exigences posées par la conception, l'implémentation et l'assemblage d'applications *soCloud* dans un contexte *multi-nuages* afin de capturer les informations nécessaires pour modéliser et décrire ces exigences de manière simple. Ce modèle consiste en un procédé de tissage de différents composants d'une application répartie, d'expression de besoins et exigences des composants. Ce résultat est obtenu en gérant chaque composant comme une unité d'exécution et en utilisant des annotations pour exprimer des propriétés non-fonctionnelles. La séparation des préoccupations de composition (annotations, composants) en sous-problèmes différents évite des problèmes d'incompatibilité et améliore la lisibilité de l'architecture d'applications *soCloud*.

D'autre part, nous avons proposé un langage dédié de haut niveau permettant aux développeurs d'exprimer des contraintes d'élasticité de manière simple et indépendante de toute technologie sous-jacente. Le niveau d'expressivité de notre langage couvre différentes contraintes d'élasticité, de stratégies, fournit un large éventail de moyen flexible pour contrôler l'élasticité d'une application. Notre langage d'élasticité est plus expressif que le langage

d'élasticité Cloudify [Gig12] discuté dans la section 3.2.3.3. Toutefois, certaines règles d'élasticité comme par exemple l'expression de la prédiction d'une charge de travail, ne peuvent pas être exprimées. Le langage peut être facilement étendu pour tenir compte des propriétés élastiques mises au point par le développeur.

Il est impératif de mettre en œuvre une plateforme *multi-nuages* qui permet de déployer, d'exécuter et de gérer des applications *soCloud*. Dans le chapitre suivant, nous présentons la plateforme *soCloud*. Le modèle *soCloud* est validé dans le chapitre 6 sur trois applications orientées services conçues, implantées et déployées dans un environnement *multi-nuages*.

Chapitre 5

Plateforme *soCloud*

"Il suffit de nommer la chose pour qu'apparaisse le sens sous le signe."-Léopold Sédar Senghor

Sommaire

Ce chapitre présente notre deuxième contribution concernant l'architecture et l'implantation de la plateforme *multi-nuages* appelé *soCloud*, qui nous permet de déployer, d'exécuter et de gérer une application *soCloud* présentée dans le chapitre 4. La plateforme *soCloud* est une plateforme PaaS orientée services à base de composants permettant de gérer la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité* dans un environnement *multi-nuages*.

5.1 Introduction

Dans le marché de l'informatique en nuage, plusieurs opérateurs fournissent différents services de nuage allant de services d'infrastructure ou IaaS tels que Amazon, Windows Azure, Rackspace à des services Plateforme ou PaaS entièrement fonctionnels comme Google App Engine, CloudBees, OpenShift. Toutefois, l'hétérogénéité de ces plateformes et infrastructures rend le déploiement des applications orientées services d'un fournisseur de nuage à l'autre difficile. Il en est de même pour sa gestion car ces plateformes imposent l'utilisation d'APIs propriétaires. Par ailleurs, les applications peuvent avoir des exigences spécifiques telles que le prix, la qualité de service, la disponibilité, la géolocalisation, les bases de données, l'intergiciel. Il est difficile pour les développeurs d'applications de trouver une réponse à toutes leurs exigences chez un seul fournisseur de nuage. *soCloud* est une plateforme *multi-nuages* orientée services qui répond aux quatre défis identifiés dans notre travail

de recherche que sont la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité* dans un environnement *multi-nuages*. *soCloud* est une plateforme répartie à base de composants qui permet de déployer, d'exécuter et de gérer les applications orientées services dans un environnement *multi-nuages*. La plateforme *soCloud* proposée a pour objectif de fournir un support complet non seulement pour déployer et exécuter les applications orientées services mais aussi les gérer de manière dynamique et autonome dans des environnements changeants *multi-nuages*.

Dans ce chapitre, nous présentons les principes sur lesquels la plateforme *soCloud* repose. Ensuite, nous décrivons l'architecture et les détails du choix d'implémentation de chaque composant qui la constitue. Nous décrivons les interactions et le fonctionnement global de la plateforme. Enfin, nous présentons les méthodes et mécanismes d'élasticité, de déploiement et de tolérance aux fautes mis en œuvre dans la plateforme.

5.2 Exigences de la plateforme *soCloud*

L'objectif de notre travail est de proposer une solution facilitant la portabilité, le déploiement des applications en fonction des besoins, de réagir aux changements dynamiques et de s'adapter en minimisant l'impact sur la disponibilité. Les approches de l'état de l'art ne sont pas entièrement satisfaisantes puisque la portabilité, le déploiement des applications sont nécessaires pour profiter d'un environnement *multi-nuages* pour exécuter et gérer ces applications. *soCloud* est une plateforme *multi-nuages* répartie basée sur le modèle SCA. La conception d'un système réparti et robuste est difficile. Il faut être conscient de la nature de la distribution, de la sémantique des références et la gestion des exceptions. Les considérations architecturales tiennent donc une place importante dans la conception de la plateforme *soCloud*. L'architecture couvre l'organisation, la structure d'ensemble, l'expression des besoins non-fonctionnels, la gestion à la fois pour les applications métiers mais aussi pour la plateforme elle-même.

Outre les problèmes architecturaux, les principaux problèmes de la conception de la plateforme sont ceux résultant de la répartition. Cette interaction repose sur une couche de communication. Il existe différents modes d'interaction (communication synchrone, communication asynchrone). Les problèmes liés à la composition et aux composants logiciels sont des éléments centraux dans la conception de la plateforme *multi-nuages soCloud*, aussi bien pour sa structure que pour les applications métiers qui sont déployées sur cette dernière [Kra07]. Dans un environnement *multi-nuages*, la gestion en cas de panne ou défaillance de la plateforme *soCloud* et des applications SaaS hébergées soulève le problème du maintien et de la persistance de la cohérence des états. La gestion du cycle de vie des applications comprend des fonctions telles que la configuration, le déploiement, la surveillance, la réaction aux changements de l'environnement d'exécution et la reconfiguration. Les exigences spontanées, rapides et croissantes des environnements de nuages conduit à envisager d'automatiser (*autonomic computing*) les tâches qui en résulte. Dans le chapitre 3, nous avons

décrit les quatre défis (la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité*) que nous avons identifiés dans notre travail de thèse.

Bien qu'il existe quelques plateformes *multi-nuages* [Mos11, Fou11, Gig12, clo12] aujourd'hui dans la littérature (voir section 3.2.2) les caractéristiques architecturales d'applications *soCloud* vont au delà de ce qui est actuellement proposé par ces dernières. Ainsi, la plateforme *soCloud* fournit un moyen efficace et simple pour déployer, exécuter et gérer les applications *multi-nuages* déployées sur cette dernière. En outre, il n'existe pas de mécanismes permettant de capturer les aspects spécifiques qui sont présents dans le contexte *multi-nuages* tels que la **portabilité**, l'**approvisionnement**, l'**élasticité** et la **haute disponibilité**.

5.2.1 Portabilité *multi-nuages*

Les différents modèles de services de l'informatique en nuage à savoir le IaaS, le PaaS et le SaaS offrent des services dédiés. Bien que leurs granularité et complexité varient, nous pensons que la définition des principes de ces services est nécessaire pour promouvoir la portabilité et la fédération entre les environnements de l'informatique en nuage hétérogènes. Par conséquent, notre infrastructure *multi-nuages* utilise SCA tant pour la définition des services applicatifs fournis par la couche PaaS que pour les applications SaaS qui fonctionnent au dessus du PaaS.

5.2.2 Approvisionnement *multi-nuages*

La plateforme *soCloud* fournit des services basés sur FraSCAti (voir Section 4.4.1.3 du chapitre 4) pour déployer et gérer l'exécution des applications SaaS et des ressources matérielles. Elle fournit une méthodologie cohérente qui offre le choix et la flexibilité aux développeurs d'applications de fournir et de livrer des applications SaaS et des ressources à travers plusieurs nuages. La plateforme *soCloud* fournit un service *multi-nuages* permettant d'approvisionner les ressources. L'approvisionnement est basé sur des politiques définies dans la plateforme *multi-nuages soCloud*.

5.2.3 Élasticité *multi-nuages*

La gestion de l'élasticité à travers plusieurs nuages n'est pas la même lorsqu'elle est gérée avec un seul nuage. En effet, les systèmes deviennent plus interconnectés et diversifiés, la latence et des pannes peuvent survenir à tout moment. Pour qu'une architecture supporte le mécanisme d'élasticité, il faut un suivi en temps réel des charges de surveillance (monitoring) d'applications et une automatisation en temps réel de l'allocation et de retrait de ressources matérielles. Ainsi, l'architecture de la plateforme *soCloud* se focalise sur l'automatisation des interactions entre ces composants à l'exécution. Une approche appropriée est

l'utilisation d'un système autonome [IMB06]. Comme décrit à la figure 5.1, l'élasticité de la plateforme *soCloud* est gérée en utilisant le modèle de référence MAPE-K (**M**onitor, **A**nalyse, **P**lan, **E**xecute, -Knowledge) [IMB06] pour la boucle de contrôle autonome.

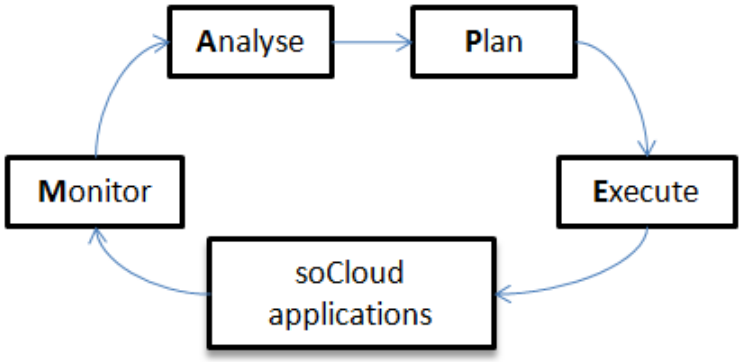

Figure 5.1 – Gestion de l'élasticité de *soCloud* avec la boucle de contrôle.

5.2.4 Haute disponibilité *multi-nuages*

Dans le contexte *multi-nuages*, les défaillances apportent de nouveaux défis parce qu'on doit changer et utiliser d'autres resources ou services qui sont dans d'autres environnements de nuage. La plateforme *soCloud* offre la haute disponibilité à deux niveaux. Premièrement, en utilisant un service d'équilibrage de charge *multi-nuages* pour faire la commutation d'un réplicat à l'autre lorsque une panne survient. Deuxièmement, l'architecture soCloud utilise une stratégie de réplication de composants. Plus précisement, soCloud utilise la redondance à tous les niveaux pour s'assurer qu'aucune défaillance d'un de ses composants déployés chez un fournisseur de nuage n'affecte la disponibilité globale du système *soCloud*.

En résumé, pour faire face au défi de la *portabilité*, *soCloud* utilise le standard SCA comme modèle. Pour faire face au défi de l'*approvisionnement*, *soCloud* offre un service pour l'approvisionnement des applications à travers de *multi-nuages*. Pour faire face au défi de l'*élasticité*, *soCloud* offre un service autonome qui fournit un mécanisme global permettant de gérer l'élasticité à travers plusieurs nuages et offre également la possibilité de définir des règles d'élasticité spécifiques à l'application. La *haute disponibilité* est assurée de deux façons. La plateforme *soCloud* est implémentée avec FraSCAti qui est un intergiciel open source pour déployer et exécuter des applications SCA. FraSCAti est l'environnement d'exécution à la fois pour la plateforme *multi-nuages soCloud* et des applications déployées sur cette dernière.

Nous avons présenté les exigences de la plateforme *soCloud*. Nous avons regroupé les caractéristiques de cette dernière en fonction des quatre défis identifiés dans notre travail de thèse en décrivant comment la plateforme *soCloud* compte les adresser. La section suivante présente l'architecture de celle-ci.

5.3 Architecture de la plateforme *soCloud*

soCloud est une plateforme *multi-nuages* répartie, conçue pour gérer la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité* des applications orientées services dans un environnement *multi-nuages*. Elle fournit des outils nécessaires pour déployer, exécuter et gérer de manière transparente des applications orientées services qui sont déployées sur cette dernière.

5.3.1 Structure de la plateforme *soCloud*

Nous présentons ici une vue globale de l'architecture de la plateforme *soCloud*. La figure 5.2 donne un aperçu de sa structure. L'architecture de la plateforme *soCloud* est constituée de deux parties : le *master* et l'*agent*. Cette décomposition a été élaborée afin de fournir une flexibilité pour le déploiement réparti de la plateforme *soCloud* dans un environnement *multi-nuages*. Cette architecture est une représentation simplifiée de la plateforme *soCloud* qui sera présentée plus en détails par la suite.

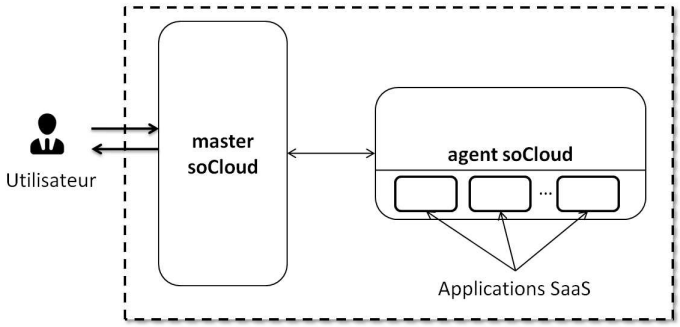

Figure 5.2 – Structure de la plateforme *soCloud*.

Le *master* se focalise sur la gestion intelligente de la plateforme *soCloud*. Ensuite, l'*agent* est utilisé pour héberger, exécuter et surveiller les applications SaaS déployées. Cette partie fournit les services nécessaires pour gérer un ensemble d'applications et de ressources matérielles. Le *master* et l'*agent* fonctionnent ensemble et sont exécutés dans différents nuages. Toutes les communications entre le soCloud *master* et les applications SaaS déployées passent par le soCloud *agent*.

5.3.2 Détails de conception de la plateforme *soCloud*

Le couplage des spécifications de déploiement, du système de gestion et l'environnement d'exécution des applications SaaS dans la conception d'une architecture fiable offre de bons potentiels [Joo05]. Par conséquent, il est nécessaire de prévoir des moyens pour

relier causalement les spécifications de l'architecture de la plateforme *soCloud* avec l'environnement d'exécution sous-jacent (FraSCAti) qui définit le déploiement, l'exécution et la reconfiguration des applications SaaS à l'exécution. Dans cette perspective, la plateforme *soCloud* fournit des composants qui permettent de spécifier des actions de déploiement, d'exécution, d'observation, de gestion et de reconfiguration à l'exécution des applications. Comme illustrée à la figure 5.3, l'architecture de la plateforme *soCloud* détaille les différents composants de ces deux parties : *master* et *agent*. En considérant les besoins des applications orientées services dans un environnement *multi-nuages*, nous avons identifié neuf composants qui doivent être fournis par la plateforme *soCloud* : *Service Deployer, Constraint Validator, Node Provisioning, PaaS Deployment, SaaS Deployment, Controller, Workload Manager, Load Balancer* et *Monitoring*.

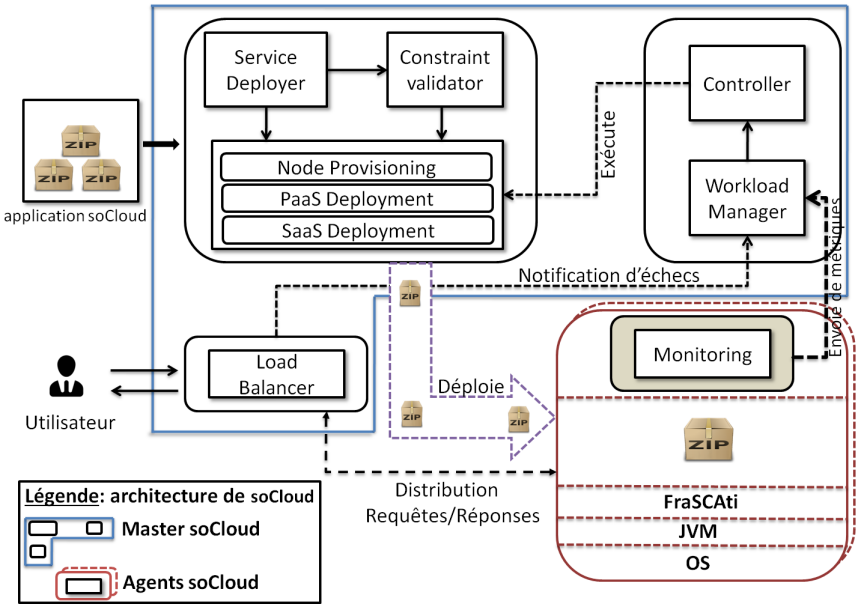

Figure 5.3 – Architecture de la plateforme *soCloud*.

Dans l'*informatique multi-nuages*, l'équilibreur de charge est utilisé pour distribuer des requêtes à travers plusieurs nuages afin de garantir la disponibilité en minimisant les défaillances de ressources et en mitigeant la limitation de ressources. Dans la plateforme *soCloud*, le développeur qui déploie son application passe par l'équilibreur qui joue le rôle de frontal au niveau de la plateforme. Avant d'être déployée, l'application est analysée par le composants *Service Deployer* et *Constraint Validator*. Le composant *Node Provisioning* approvisionne les ressources nécessaires pour l'application. Avant le déploiement de l'application par le composant *SaaS Deployment*, le composant *PaaS Deployment* déploie l'agent *soCloud* sur le nuage correspondant. L'application déployée est surveillée et contrôlée.

5.3.2.1 Load Balancer

Le composant *équilibreur de charge* (*Load Balancer*) permet un accès facile à la fois aux applications et à la plateforme *soCloud*. Il agit comme un portail pour des grappes de serveurs réparties et des requêtes de clients à travers un ensemble de serveurs. Il distribue les requêtes aux instances d'une application déployée dans plusieurs nuages. L'*équilibreur de charge* permet d'ajouter ou de supprimer des applications de sa liste des applications disponibles (accessibles) dynamiquement. Il répartit les requêtes de clients à travers les instances d'une application lorsque cette dernière est répliquée dans plusieurs nuages. Ces instances de l'application apparaissent du point de vue des utilisateurs finaux comme une seule application avec de grandes capacités de traitement. Il transmet des sessions et requêtes entrantes seulement aux instances disponibles. Ainsi, avant de distribuer les requêtes des clients, l'*équilibreur de charge* utilise un mécanisme lui permettant de déterminer les instances d'applications qui sont accessibles et disponibles pour exécuter des requêtes. Chaque application déployée sur la plateforme *soCloud* est associée à deux instances d'*équilibreur de charge*. Les deux instances d'*équilibreur de charge* appartiennent à un même groupe de gestion (group membership). Ainsi lorsqu'une instance d'*équilibreur de charge* tombe en panne, la seconde prend le relais automatiquement. Toutefois, une seule instance de l'*équilibreur de charge* est active à la fois, la seconde reste inactive. Lorsque l'équilibreur de charge détecte une défaillance sur une instance d'une application, il le notifie au composant *Workload Manager*.

5.3.2.2 Service Deployer

Le composant *Service Deployer* est responsable du traitement des applications déployées sous la forme de contributions sur la plateforme *soCloud*, de la coordination et de la gestion du placement des applications réparties dans un environnement *multi-nuages*. Ce composant décompose et prend en charge les contraintes spécifiées par le développeur via les annotations placées sur l'application à déployer. La décomposition et le traitement se font en quatre étapes comme décrites dans la figure 5.4. A la phase **1)** la contribution qui représente l'application à déployer est désassemblée en plusieurs ou une seule contribution selon l'architecture de l'application (répartie ou non). La deuxième phase **2)** valide la syntaxe et la cohérence des différentes contraintes placées sur les composants de l'application. Durant la troisième phase **3)**, le composant *Service Deployer* créé une règle de mapping entre les contraintes et les opérations de l'environnement d'exécution qui sont stockées dans le dépôt (repository). Au cours de la quatrième phase **4)**, le composant *Service Deployer* construit le graphe de dépendance des contributions correspondants aux composants afin d'établir l'ordre dans lequel ces derniers doivent être déployés et activés. Dans le cas où les contraintes placées sur les composants sont remplies par plusieurs nuages, la plateforme *soCloud* choisit le nuage qui propose le prix le moins cher. La plateforme *soCloud* effectue un choix aléatoire lorsque le prix le moins cher est proposé par plusieurs nuages. En effet, *soCloud* dispose d'une base de

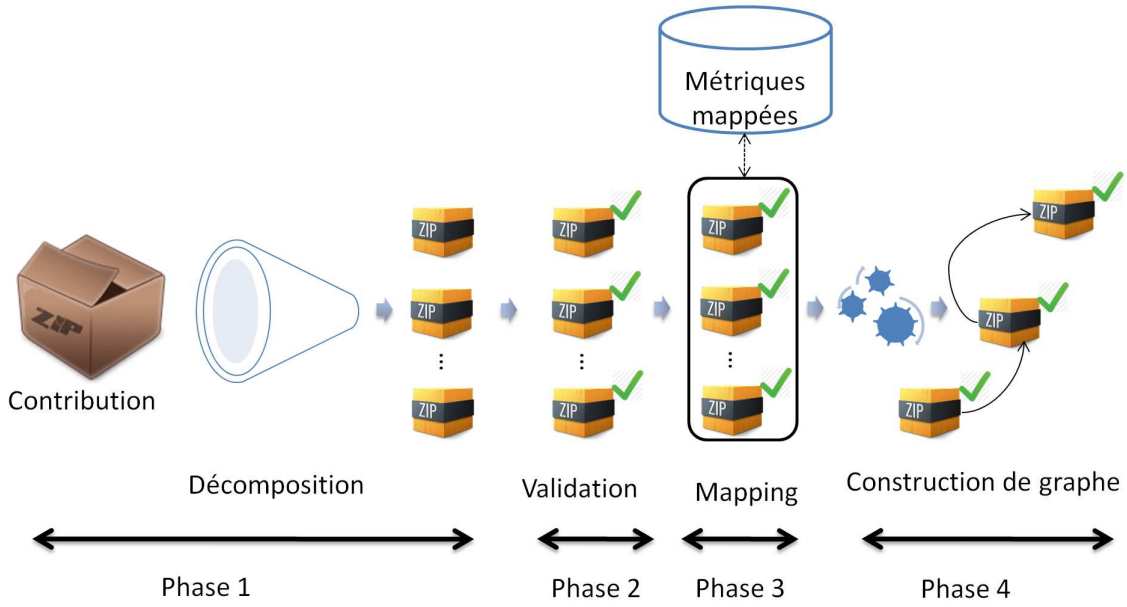

Figure 5.4 – Les différentes étapes de la décomposition d'une contribution.

données qui contient les informations sur les fournisseurs de nuages et le prix de consommation des ressources. Cette base de données est mise à jour régulièrement afin de prendre en compte les prix de consommation de ressources qui ont changés entre temps.

5.3.2.3 Constraint validator

Le composant *validateur des contraintes* (*Constraint Validator*) effectue la validation des contraintes placées sur les composants. Plus précisement, il procède à la validation des annotations (voir chapitre 4 page 75) placées sur les applications *soCloud*. Cette validation se fait en deux étapes. Une première étape consiste en la validation syntaxique des annotations. Une deuxième étape se focalise sur la cohérence des annotations.

5.3.2.4 Node Provisioning

Le composant *allocateur de ressources* (*Node Provisioning*) fournit des ressources à la fois de l'infrastructure (IaaS) et la plateforme (PaaS) en fonction des besoins. Ce composant fournit une couche d'abstraction permettant d'allouer des ressources provenant de l'infrastructure (IaaS) ou de la plateforme (PaaS) de manière uniforme. Pour faire face aux charges de travail courant, le composant *allocateur de ressources* fournit les ressources nécessaires. Lorsque les ressources ne sont pas utilisées ou sont sous-utilisées, elles sont libérées. L'allocation de

ressources peut provenir de différents fournisseurs de nuage. La capacité à fournir des ressources dynamiquement et rapidement provenant de plusieurs fournisseurs de nuage est essentielle pour la plateforme *soCloud*. Les ressources sont gérées comme des pools logiques. Fondamentalement, les pools de ressources logiques permettent au composant *allocateur de ressources* de contrôler et d'organiser des ressources matérielles provenant de différents nuages comme une seule entité uniforme. L'allocation des ressources de différents fournisseurs de nuage garantit la disponibilité des ressources aux applications qui en ont besoin en dépit de panne ou d'indisponibilité de ressources chez un fournisseur donné, tout en maximisant l'utilisation globale des ressources.

5.3.2.5 PaaS deployment

Le composant de *déploiement de Plateforme* (*PaaS Deployment*) déploie une instance de l'environnement d'exécution (FraSCAti) sur le nuage ciblé fourni par le composant *allocateur de ressources*. Une fois le déploiement de l'environnement d'exécution effectué, le *soCloud* agent est installé puis instancié sur le même nuage. Ce mécanisme est une solution générique pour le déploiement dans les environnements distribués.

5.3.2.6 SaaS deployment

Le composant de *déploiement d'applications SaaS* (*SaaS deployment*) permet de déployer ou de retirer dynamiquement des applications SaaS qui s'exécutent sur un agent *soCloud*. Au cours de la phase de déploiement, une copie de chaque application SaaS déployée est conservée dans le dépôt des applications SaaS. Ainsi, à tout moment la plateforme *soCloud* peut décider de redéployer de nouvelles instances de l'application pour fournir l'élasticité ou en cas d'indisponibilité.

5.3.2.7 Monitoring

Le composant de surveillance (*Monitoring*) se trouve dans la partie **agent *soCloud***. Il fournit une API indépendante des plateformes d'exécution qui collecte et agrége des informations de surveillance (les indicateurs de performance, métrique de disponibilité) sur des applications déployées dans un environnement *multi-nuages*. Il apporte des informations sur les processus en cours d'exécution ainsi que le système sur lequel le composant de surveillance s'exécute. Ce composant est capable de surveiller les ressources matérielles, réseaux et applicatives. Le tableau 5.1 décrit les différentes métriques que fournit le composant de *surveillance* à deux niveaux : système d'exploitation et application. Le composant de surveillance détecte les changements dans l'état de l'application. Il associe à chaque application déployée sur la plateforme *soCloud* une table temporaire appelée «responsivenessEvent» qui recueille des informations telles que le temps de réponse de l'application, le

nombre de requêtes entrantes, le nombre de requêtes traitées. Les métriques collectées périodiquement sont envoyées au composant *Workload Manager* pour l'analyse. La décision de séparer l'analyse des messages envoyés par le composant de *surveillance* vise à faciliter le passage à l'échelle de la plateforme *soCloud* et de rendre le composant de *surveillance* moins intrusif aux applications qu'il surveille.

Tableau 5.1 – Métriques observés.

Métrique	**Description**	**OS**	**Application**
CPU	La capacité cpu utilisée sur le système d'exploitation et la capacité cpu consommée par une application.	*	*
Mémoire RAM	La quantité de mémoire utilisée sur le système d'exploitation et la quantité de mémoire consommée par une application.	*	*
Disque	La quantité de disque utilisée sur le système d'exploitation.	*	
Réseau	La bande passante utilisée sur le système d'exploitation.	*	
Temps de réponse	Le temps écoulé entre le début et la fin d'une requête.		*
Nombre de requêtes	Le nombre de requêtes par seconde.		*

La figure 5.5 présente le système de monitoring de l'agent *soCloud*. Le composant de *surveillance* est conçu pour minimiser le nombre de messages qu'il envoie. En effet, il est déconseillé d'envoyer des notifications de métriques collectées à travers le réseau de manière continue. Dans un environnement réparti à large échelle comme l'*informatique multi-nuages*, plus précisement dans le cas de la plateforme *soCloud*, une application à trois composants génère trois types de messages par composant. Un message pour les métriques de ressources matérielles consommées par l'application, un message pour les indicateurs de performance de l'application et un message sur l'état de fonctionnement de l'application. Les métriques sont collectées toutes les 5 secondes. Pour une période d'une minute, on collecte 36 messages pour une application à trois composants. En fonction du nombre d'applications déployées et de leurs activités, on peut collecter des milliers de messages par minute. Une solution est d'intégrer la logique de traitement d'événements dans le composant de *surveillance* pour réduire le plus possible le nombre de notifications. Ainsi les événements seront filtrés à la source (au niveau de l'*agent soCloud*). Le mécanisme de filtrage peut être ajusté dynamiquement à n'importe quel moment. Le composant de *surveillance* recevra des instructions du composant *Contrôleur* pour arrêter la publication des notifications que le consommateur d'événements ne veut pas. Nous considérons que le fait de saturer le réseau avec l'envoi de milliers d'informations n'a pas de sens si on peut les filtrer pour n'envoyer que les notifications de haut-niveau pertinentes. Comme illustré dans la figure 5.5, le composant de *surveillance* est embarqué dans chaque machine virtuelle (VM) instanciée. Il filtre et agrège

les métriques avant de les envoyer. Le filtrage permet d'envoyer des messages qui sont pertinents et l'agrégation fait baisser le nombre de messages à envoyer.

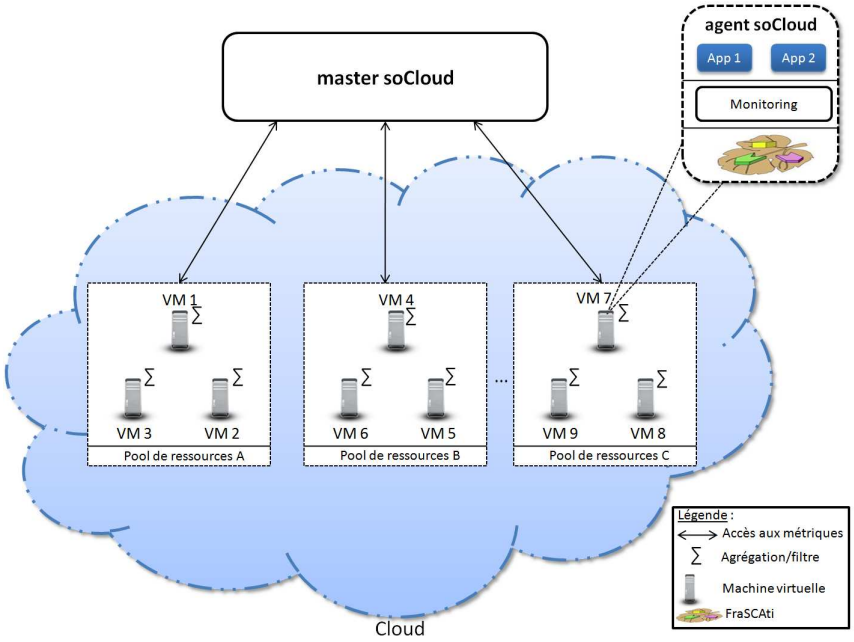

Figure 5.5 – Système de surveillance *soCloud*.

5.3.2.8 Workload Manager

Le composant *Workload Manager* fournit des fonctionnalités de traitement des événements [Etz10]. Tous les événements sont traités pour détecter une éventuelle anomalie. Par exemple, la consommation en cpu supérieure à 90% sur une période de deux minutes peut être un indicateur de dérive. Le *Workload Manager* se focalise sur le filtrage, la transformation et l'agrégation des événements. Toutes les métriques envoyées par le composant de *surveillance* sont analysées de manière continue et en temps réel pour en extraire des indicateurs de dérive. Ces analyses sont exprimées par des règles sur les événements et agissent en temps réel sur les menaces éventuelles en créant d'autres événements dérivés à transmettre. Le composant *Workload Manager* utilise une technique appelée **corrélation d'événements** [16] pour examiner des symptômes puis identifier des groupes de symptômes qui ont une cause commune. Un exemple de corrélation d'événements est : le composant *Workload Manager* prend plusieurs occurrences de la même donnée d'événement, les examine pour recenser les informations en double, puis supprime les redondantes et les signale comme un événement unique. Ainsi, bien que envoyé cinquante fois, l'événement "Consommation de mémoire supérieur à 98%" ne va être signalé qu'une seule fois. Le *Workload Manager* recherche des motifs

16. http ://tinyurl.com/qdrcpm3

particuliers parmi les événements qu'il surveille. Losrqu'il détecte une dérive, il le signale au composant *Contrôleur*.

La capacité d'extraire des informations qui peuvent être complexes à saisir de manière instantanée dans les opérations d'allocation de ressources est essentielle pour la réactivité de la plateforme *soCloud* dans un environnement *multi-nuages*. Ainsi, la capacité d'allouer et de retirer des ressources dynamiquement est un facteur important dans la construction d'une plateforme élastique. Cette caractéristique permet à la plateforme *soCloud* de gérer un volume croissant de transactions.

5.3.2.9 Controller

Le composant *contrôleur* (*Controller*) fournit des mécanismes qui construisent les actions nécessaires pour atteindre des buts et objectifs. Par exemple, il multiplexe les charges de travail dans l'infrastructure existante et permet d'allouer des ressources en fonction de la charge de travail. Il est responsable de l'interception des requêtes et délègue leurs traitements aux composants correspondants. L'état du système est géré par le *contrôleur*. Par état, nous entendons des informations conservées dans un composant qui sont significatives pour celui-ci (par exemple, une table sur la translation d'un équilibreur de charge qui associe des adresses de réseau aux noms symboliques des hôtes disponibles). Les états du système offrent la possibilité d'améliorer la cohérence du système. Un autre avantage de la gestion des états est la fiabilité. Lorsqu'une information est répliquée chez plusieurs fournisseurs de nuage et que l'un des réplicats est perdu en raison d'une panne ou défaillance, il est possible d'utiliser une autre copie pour restaurer les informations perdues. Comparé aux autres composants de la plateforme *soCloud*, c'est le *contrôleur* qui prend des décisions dans le système.

En plus des composants précédents, la plateforme *soCloud* réduit le temps de développement en fournissant d'autres services utiles tels que

1. Le service de géo-localisation : ce service permet par exemple de rediriger des utilisateurs qui se trouvent dans une zone géographique donnée vers le composant le plus proche géographiquement (voir figure 5.6).
2. Le service de persistance de données : il permet d'éclater une base de données (Database sharding) pour faciliter sa mise à l'échelle. Le processus d'éclat de base de données consiste à avoir plusieurs bases de données de manière horizontale avec les mêmes schémas. Ce service fournit un patron qui se focalise sur la mise à l'échelle *horizontale* des données à travers les éclats (sharding). Ce service est disponible uniquement pour les bases de données relationnelles.
3. Le service d'authentification : il permet de fournir des mécanismes d'authentification basés sur les mails ou comptes de réseaux sociaux.
4. Le service de création de graphiques : ce service permet de créer le graphique de consommation de ressources (cpu, mémoire, réseau) consommé par le composant de manière quotidienne, hebdomadaire, mensuelle et annuelle à partir des métriques collectées sur une application (voir figure 5.7).

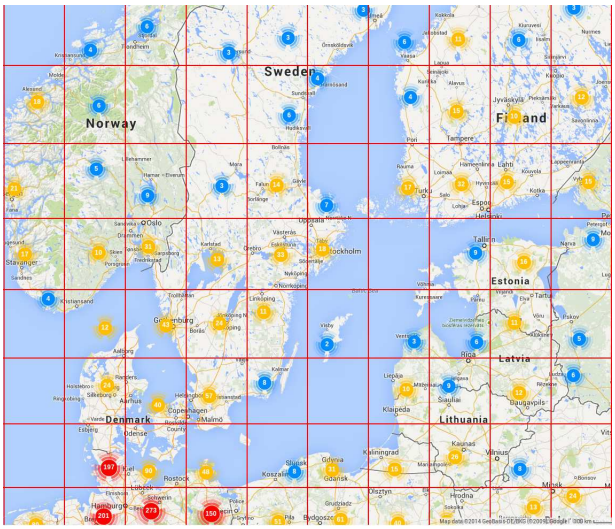

Figure 5.6 – Service de géo-localisation *soCloud* basé sur la base de données GeoIP [MAX12].

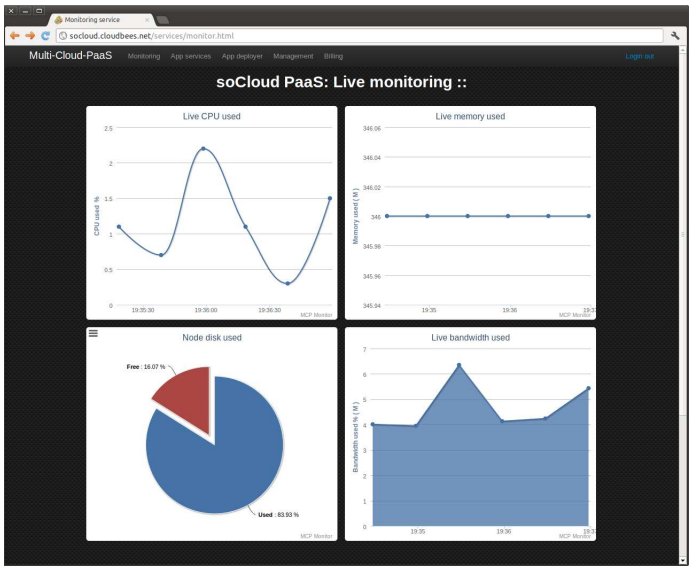

Figure 5.7 – Service de création de graphiques *soCloud*.

5.3.3 Interactions entre les composants de la plateforme *soCloud*

Nous venons de décrire dans la section précédente chaque composant de la plateforme *soCloud*. Il est maintenant nécessaire de connaitre leurs interactions. La figure 5.8 décrit le diagramme de séquence entre les différents composants de la plateforme *soCloud*. Lorsque le développeur déploie son application avec la plateforme *soCloud*, le composant *Service Deployer* déconditionne l'application pour l'analyser. Il utilise le service fourni par le compo-

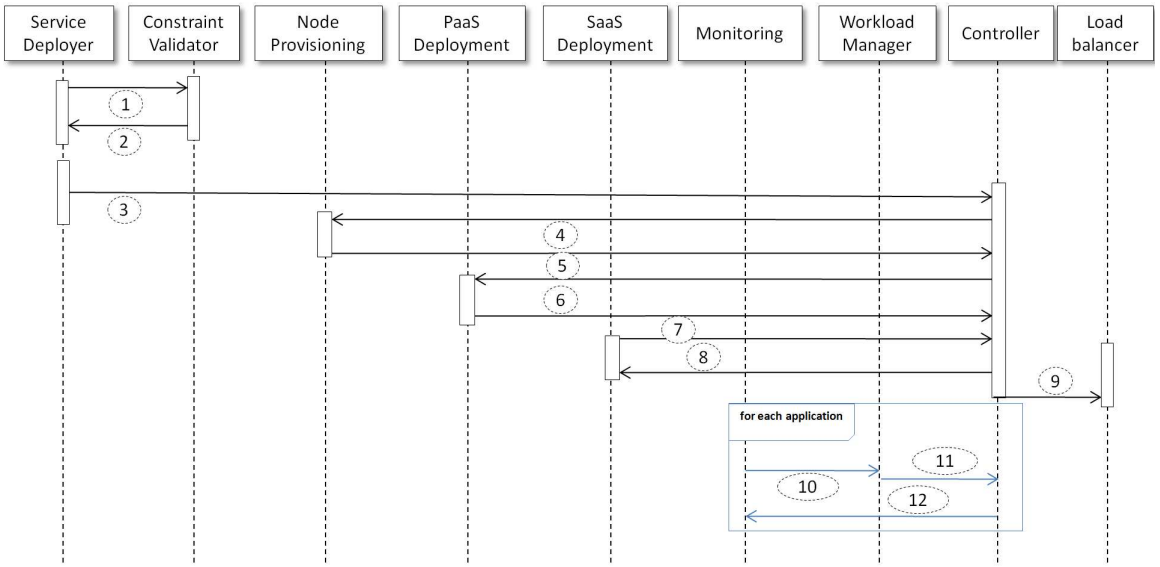

Figure 5.8 – Interactions entre les différents composants.

sant *Constraint validator* (phase **1**) pour procéder à la validation des contraintes placées sur les composants de l'application. Dans la phase (**2**), le composant *Constraint validator* fait une validation syntaxique et cohérente des annotations placées sur les composants. Les erreurs déclenchées au cours de la validation de la phase (**2**) sont notifiées au développeur. Dans le cas contraire, le composant *Service Deployer* est notifié de la bonne syntaxe et la cohérence des annotations. Le composant *Service Deployer* fait le mapping entre les annotations exprimées et les opérations (placement, élasticité, fiabilité et exécution) à effectuer et génère le graphe de dépendances des composants. Ensuite, le composant *contrôleur* est notifié de l'opération en cours. Il ordonne au composant *Node Provisioning* d'utiliser les informations de placement construites par le composant *Service Deployer* pour allouer des ressources requises pour chaque composant de l'application (phase **3**). Une fois les resources allouées, une notification est envoyée au *contrôleur* (phase **4**). Le contrôleur demande au composant *PaaS Deployer* de déployer l'environnement d'exécution (FraSCAti) sur les nœuds alloués (phase **5**). Le *PaaS Deployer* envoie une notification au contrôleur (phase **6**). Ensuite, le contrôleur demande au composant *SaaS Deployment* de déployer les différents composants de l'application en respectant les contraintes de placement spécifiées et l'ordre de déploiement établi par le graphe de dépendances (phase **7**). Une fois l'application déployée, une notification est envoyée au contrôleur (phase **8**). Ce dernier déploie une instance de l'*équilibreur de charge* qu'il place entre l'application et les utilisateurs finaux (phase **9**). Ensuite le composant *Monitoring* surveille automatiquement l'application déployée et collecte des métriques qui sont envoyées au composant *Workload Manager* pour l'analyse. Lorsqu'une dérive est détectée, le composant *Workload Manager* la notifie au composant *Contrôleur* (phase **10**) pour qu'il prenne la décision appropriée (phase **11**). En phase **12**, le contrôleur peut ajuster le comportement

du composant de *surveillance*. Les phases **10, 11 et 12** seront répétées de manière continue tout de long de l'exécution de l'application déployée.

Après les phases 1 à 8 appliquées sur notre application fil rouge (voir chapitre 4 section 4.2), la figure 5.9 illustre la phase de déploiement de cette application trois-tiers. On peut noter que l'annotation *@replication* placée sur les composants *Présentation* et *Stockage* se traduit par le déploiement de N (N est la valeur affectée à l'annotation *@replication*) contributions de ces derniers en nuage. L'absence de l'annotation *@replication* sur le composant *Métier* se traduit par le déploiement d'une seule instance de ce dernier.

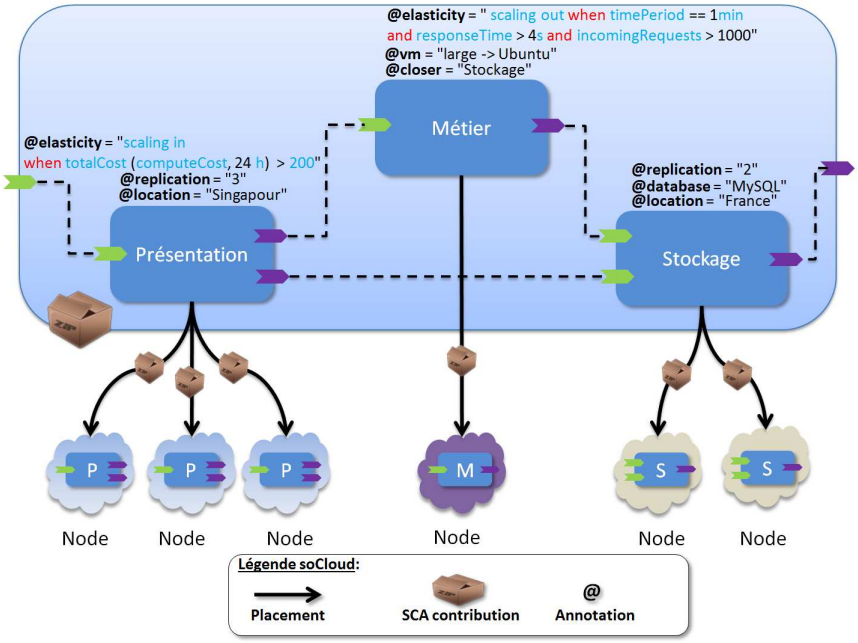

Figure 5.9 – Déploiement de l'application trois-tiers.

Nous venons de décrire l'interaction entre les différents composants de l'architecture de la plateforme *soCloud*. La section suivante présente le choix d'implémentation de chacun de ces composants.

5.4 Choix d'implantation de la plateforme *soCloud*

Dans cette section, nous décrivons les détails d'implémentation de l'architecture de la plateforme *soCloud* et les choix technologiques effectués.

5.4.1 Implémentation des composants

L'implémentation de *soCloud* vise à mettre en œuvre une plateforme *multi-nuages* qui prend en compte certaines exigences telles que la *portabilité*, l'*approvisionnement*, l'*élasticité* et

la *haute disponibilité multi-nuages*. Pour atteindre ces objectifs, l'architecture de la plateforme *soCloud* est réalisée avec le modèle SCA. La figure 5.10 illustre l'architecture SCA de la plateforme *soCloud*, avec le *master* et l'*agent soCloud* qui sont connectés via HTTP. La plateforme *soCloud* est implémentée en Java.

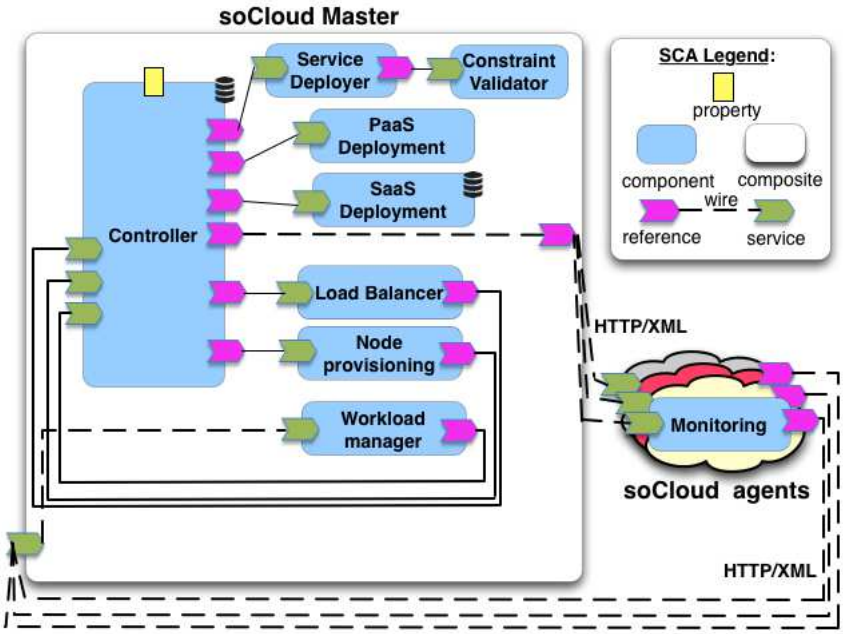

Figure 5.10 – Architecture SCA de la plateforme *soCloud*.

5.4.1.1 Monitoring

Une application *soCloud* peut être déployée en mode réparti ou non selon sa conception. La question principale est de savoir comment surveiller un environnement hautement réparti et hétérogène ? Le composant de *surveillance* (*Monitoring*) utilise un mécanisme d'identification logique et d'agrégation pour surveiller des applications réparties ou non. Par exemple, une application répartie ayant plusieurs composants se voit assigner le même identifiant afin de traiter les informations recueillies comme des informations provenant d'une seule entité. Pour prendre la décision de mise à l'échelle pour des applications déployées, il est nécessaire d'instrumenter le code des applications et les ressources sur lesquelles elles sont exécutées. Ainsi, le composant de *surveillance* agit à trois niveaux : i) Ressources, ii) Machine Virtuelle Java (JVM) et iii) Environnement d'exécution (FraSCAti). Les services du composant de *surveillance* sont exposés en REST et JMS. Le consommateur de ces services est le composant *Workload Manager* ou peut être toute autre application ou service qui s'exécute en dehors de la plateforme *soCloud*. Nous avons identifié trois niveaux pour surveiller une application *soCloud* :

1. **Ressources** : la plateforme *soCloud* a besoin de surveiller la disponibilité des applications et la charge globale du système dans lequel elles sont déployées. Le composant de *surveillance* observe les ressources matérielles (cpu, mémoire, disque, réseau) du système. Pour cela il utilise l'API de bas-niveau SIGAR développé en C. L'API SIGAR est un projet Open Source développé par HYPERIC [VMW12]. Nous avons choisi d'utiliser SIGAR parce que cette API supporte plusieurs systèmes d'exploitation [VMW12] (Linux, Solarix, Windows, AIX, FreeBSD, Mac OS X) et différentes versions d'architectures (x86, x64, amd64, ppc, ppc64, PowerPC, sparc-32, sparc-64, ia64). Le composant de *surveillance* utilise une liaison C fournie par FraSCAti pour accéder aux bibliothèques natives depuis Java.
2. **JVM** : le composant de *surveillance* utilise l'intergiciel JMX [Sun01b] pour instrumenter les applications *soCloud* déployées. L'intergiciel JMX fournit des outils pour gérer et surveiller des applications. Nous avons besoin de recueillir des statistiques importantes telles que le temps de réponse moyen, le temps de réponse maximal, le nombre de requêtes traitées par seconde.
3. **Environnement d'exécution** : avec les *intents*(voir section 4.4.1 page 74) de FraSCAti, le composant de *surveillance* a la possibilité d'observer facilement les applications *soCloud* dans le but de détecter les événements critiques. La politique des intents décrit la façon dont le service du composant de *surveillance* se comporte.

Le composant de surveillance génère un fichier XML contenant les métriques recueillies. Le formalisme du fichier XML est un ensemble de couples clé-valeur.

5.4.1.2 Workload Manager

Le composant *Workload Manager* est le principal consommateur des événements envoyés par le composant de *surveillance*. Comment ces événements sont analysés et corrélés ? Ceci est accompli grâce à un moteur de traitement d'évènements complexes (CEP). Le CEP est une technologie pour traiter des événements et découvrir des motifs complexes entre plusieurs flux d'événements. Pour le mettre en œuvre, nous avons utilisé *DiCEPE* un intergiciel réparti de moteur d'événements complexes que nous avons présenté dans [Par12b]. DiCEPE permet d'intégrer des moteurs de CEP hétérogènes et offre la possibilité de les exposer comme des services en utilisant différents protocoles de communication. DiCEPE offre la possibilité d'embarquer les moteurs ETALIS [ETA] et Esper [ESP] pour le traitement des événements. ETALIS est un moteur de traitement d'événements complexes implémenté en Prolog. ETALIS implémente deux langages pour la spécification des modèles d'événements : ETALIS Language for Events (ELE) et le traitement des événements SPARQL. Esper est un moteur de traitement d'événements complexes (CEP) et de flux d'événements ou Event Stream Processing (ESP). ESP est une technologie qui manipule les tâches de traitement de plusieurs flux de données d'événements dans le but d'identifier les événements qui ont un sens avec ces flux et en tirer des renseignements utiles. Les patrons d'événements (Event patterns) permettent de définir différentes règles de correspondance qui peuvent être

appliquées aux événements entrants. Par exemple, un opérateur temporel peut être utilisé pour comparer des événements entrants et vérifier s'ils sont arrivés dans l'ordre prévu. Nous utilisons Esper pour sa portabilité multiples plateformes et la façon dont il gère la paire clé-valeur qui est délivrée comme un événement dont les valeurs changent à chaque mesure.

Pour la persistance des événements, nous utilisons Google Fusion Table [Gon10] dans la plateforme *soCloud* pour conserver les données. Avec les données persistées, la plateforme *soCloud* a la capacité de créer des graphiques via une interface graphique web (voir figure 5.7) pour afficher des statistiques ou bien pour faire des traitements plus poussés afin d'améliorer l'analyse effectuée par le composant *Workload Manager*.

5.4.1.3 Controller

Toutes les requêtes traitées par ce composant sont gérées comme des transactions. Le moteur de transaction est mis en œuvre pour des besoins spécifiques de l'architecture de la plateforme *soCloud*. Le moteur de transaction a été implémenté en utilisant l'API JTA [Sun00]. Nous avons choisi cette API parce qu'elle permet de gérer des transactions réparties, ce qui correspond aux besoins de l'architecture de *soCloud*. Le but de la transaction est d'assurer que tous les composants gérés par le composant *contrôleur* restent dans un état cohérent lorsqu'il y a un accès concurrent et en présence de panne. Le composant *contrôleur* doit garantir que l'ensemble des transactions et les résultats sont stockés dans une base de données permanente au cas où lorsque l'une d'entre elle est défaillante, cela n'ait pas d'impact sur les autres. Chaque transaction créée est gérée par le coordinateur. Il existe deux problèmes bien connus en cas de transactions simultanées : i) la perte de mise à jour et ii) la récupération incohérente. Pour éviter ces problèmes nous utilisons l'équivalence de série (serial equivalent) des transactions [17]. L'utilisation de l'équivalence de série en tant que critère pour une exécution simultanée empêche l'apparition des mises à jour perdues et des récupérations incohérentes. Le composant *contrôleur* est au cœur de la gestion de l'élasticité de la plateforme *soCloud*. Il est conçu pour la tolérance aux pannes en utilisant la réplication.

5.4.1.4 Load Balancer

Comme mentionné dans la section 5.3.2.7, chaque application déployée avec la plateforme *soCloud* possède deux instances d'*équilibreur de charge*. Il offre la haute disponibilité, la répartition de charge et joue le mandataire (proxy) pour le protocole TCP et les applications HTTP. Pour mettre en œuvre le composant *équilibreur de charge*, nous avons implémenté un serveur non-bloquant qui offre plus de performance et facilite le passage à l'échelle. Le serveur non-bloquant consomme uniquement une poignée de threads. Il utilise une boucle Entrée/Sortie et des événements pour traiter les requêtes. Pour tirer pleinement avantage

17. équivalence de série : si chaque opération de transaction est connue pour avoir le bon effet lorsqu'elle est exécutée, alors nous pouvons en déduire que si ces opérations sont effectuées l'une après l'autre dans un ordre, la combinaison sera également correct. [Cou05].

de la technologie non-bloquante, toutes les entrées/sorties et appels réseaux doivent être non-bloquants aussi. Le composant *équilibreur de charge* implémente ce modèle à base d'événements qui utilise un seul thread et permet de supporter un nombre élevé de connexions simultanées à très grande vitesse. Les modèles multiples processus ou multiples threads peuvent rarement faire face à des milliers de connexions en raison de la limite en ressource mémoire, des limites d'ordonnanceur du système, des conflits de verrous omniprésents. Les modèles d'événements n'ont pas ces problèmes parce qu'ils implémentent toutes les tâches dans l'espace utilisateur ce qui permet une gestion de ressources plus fine [Dab02]. Le composant *équilibreur de charge* est implémenté en utilisant l'intergiciel Netty [18]. Pour la capacité dynamique de l'*équilibreur de charge*, nous avons mis en œuvre un service de gestion de groupe (group membership). Pour éviter un point de défaillance unique (single point of failure [19]), le composant doit être répliqué. Ainsi, chaque application a un nom de domaine (DNS [Gro87]). Par exemple, *appname.socloud.net* est associé à deux instances du composant *équilibreur de charge*. L'implémentation actuelle du composant *équilibreur de charge* supporte l'algorithme Round Robin [Shr95]. Une API est proposée pour étendre le comportement par défaut (c'est-à-dire le fait d'utiliser l'algorithme Round Robin) de l'*équilibreur de charge* en utilisant d'autres algorithmes sophistiqués [Adl10, Teo01]. Une application déployée avec la plateforme *soCloud* est associée à un groupe d'*équilibreur de charge* comme le montre la figure 5.11, en particulier avec l'instance active du groupe d'*équilibreur de charge*. Lorsque l'instance active du groupe d'*équilibreur de charge* tombe en panne, l'instance passive du groupe détecte l'anomalie et prend en charge les opérations en cours. L'instance passive vérifie si l'instance active précédente a effectivement échoué et si c'est le cas l'instance passive devient définitivement active. La nouvelle instance active signale l'inexistence d'instance passive au composant *Workload Manager* qui à son tour le notifie au *Contrôleur* chargé d'instancier un nouvel *équilibreur de charge* passif.

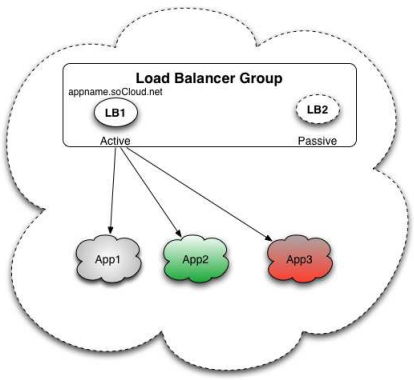

Figure 5.11 – Equilibreur de charge : groupe d'appartenance.

18. https ://netty.io
19. Un point de défaillance est un composant unique (matériel, logiciel, ou autre) dont la défaillance entraîne un certain degré d'indisponibilité.

5.4.1.5 Service Deployer

Pour effectuer le déploiement des applications *soCloud*, le composant *Service Deployer* capture les contraintes placées sur les composants de l'application et les fait valider par le composant *Constraint Validator*. Ensuite, il construit le graphe de dépendances des composants à déployer. Il fournit un outil qui permet d'analyser et de récupérer la dépendance entre les composants d'une application en analysant le fichier de description de l'application. Chaque composant est représenté par un nœud et les dépendances entre les composants représentent des liens entre nœuds. En prenant l'architecture de notre application trois-tiers du cas d'utilisation (voir Section 4.2 sur la page 69), le graphe de dépendances correspondant est illustré sur la figure 5.12. L'équation 5.1 représente l'ordre de priorité établi dans le processus de déploiement. Ainsi, le tiers *Stockage* est prioritaire par rapport au tiers *Metier* et le tiers *Metier* est prioritaire par rapport au tiers *Presentation* en terme de déploiement.

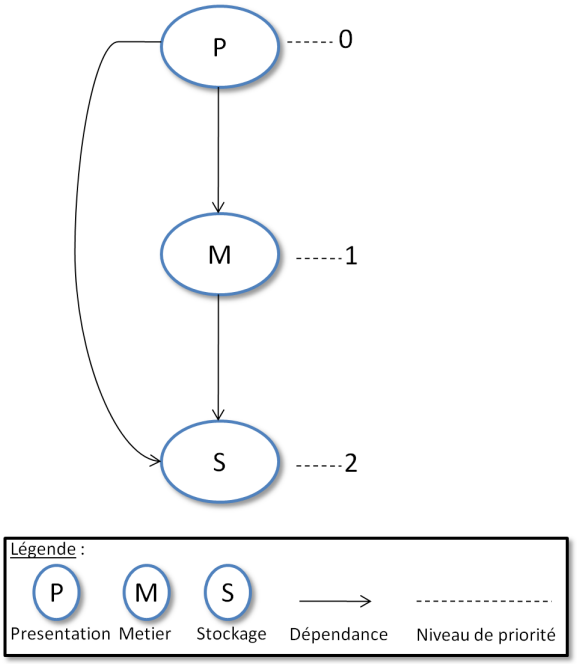

Figure 5.12 – Graphe de dépendances.

$$Pri(Stockage) > Pri(Metier) > Pri(Presentation), Pri(C) \in \mathbb{N}, C \in \{composants\} \quad (5.1)$$

Pri et *C* réprésentent respectivement la fonction de priorité et le nom des composants. La valeur de la fonction de priorité correspond au niveau du nœud dans l'arbre. Comme le montre la figure 5.12, $Pri(Stockage) = 2$, $Pri(Metier) = 1$, $Pri(Presentation) = 0$. Toutefois, il se peut qu'il y ait des composants ayant la même priorité et dans ce cas leur ordre de déploiement n'a pas d'importance.

Le mapping entre les contraintes (*placement*, *élasticité*, *exécution* et *fiabilité*) et le dépôt des règles d'opérations se font comme suit :

Contraintes de placement Grâce au composant de *surveillance*, la plateforme *soCloud* maintient une table qui collecte via des sondes des informations telles que la latence réseau entre les machines virtuelles que contrôle la plateforme *soCloud*, leur localisation, le nom de fournisseur. Pour satisfaire les *contraintes de placement*, la plateforme *soCloud* dispose d'une base de connaissances actualisée continuellement.

Contraintes d'élasticité Le langage spécifique utilisé pour exprimer les *contraintes d'élasticité* est traduit en langage EPL (Event Processing Language) du moteur de CEP Esper qui permet d'agir de manière continue et en temps réel aux changements du contexte de l'environnement. Plus précisement, nous utilisons l'outil standard ANTLR [Sun89] pour analyser notre langage d'élasticité afin de générer le langage EPL correspondant.

Contraintes d'exécution Pour satisfaire les *contraintes d'exécution*, nous réalisons le mapping entre ces dernières et la table qui contient la liste des caractéristiques des machines virtuelles utilisées. Comme le montre l'équation 5.2, une machine virtuelle VM est constituée de deux composants : Matériel (M) et *logiciel ou système d'exploitation (SE)*. Le composant *M* est constitué de cpu, mémoire, disque, réseau qui représentent les caractéristiques matérielles. Le composant *SE* est le système d'exploitation qui peut être de type Ubuntu, Debian, CentOS, Windows, etc.

$$VM = (M_{car}, SE_{type}) \tag{5.2}$$

Dans un environnement hétérogène de *multi-nuages*, les types de *M* existants fournis par les différentes infrastructures peuvent avoir de petites différences. Toutefois, pour utiliser ces nuages simultanément et cacher cette différence, la plateforme *soCloud* définit une abstraction de haut niveau où on classe les *M* similaires dans le même type. Le type de *M* qui provient de différentes infrastructures de nuages est classifié par la fonction abstraite de haut niveau f (voir Equation 5.3).

$$M_{car} = \{M : f(M_i) i \in \mathbb{N}\} \tag{5.3}$$

Cette abstraction définie par la plateforme *soCloud* n'empêche pas le développeur de choisir une machine spécifique. Pour le faire, ce dernier doit combiner les annotations *@vm* et *@placement* qui représentent respectivement le type de machine et le nom du fournisseur de nuages.

Contrainte de fiabilité Le nombre d'instances exprimé au niveau des *contraintes de fiabilité* se traduit par le nombre d'instances du composant à déployer qu'il faut placer derrière l'équilibreur de charge.

A la fin du processus de traitement des applications par le composant *Service Deployer*, on obtient un tuple App comme décrit dans l'équation 5.4. $app_{identifiant}$ représente l'identifiant de l'application, $appComponent_{installOrder}$ représente l'ordre d'installation des contributions qui correspondent aux composants de l'application et $appComponent_{contraintes}$ sont

les contraintes que les composants de l'application doivent respecter.

$$App = (app_{identifiant}, appComponent_{installOrder}, appComponent_{contraintes}) \quad (5.4)$$

5.4.1.6 Constraint Validator

La validation des contraintes placées sur les composants d'une application *soCloud* se fait au niveau du composant *Constraint Validator*. Nous avons implémenté un mini solveur de contraintes permettant de garantir la satisfaisabilité des contraintes placées sur les composants. Selon le type de contrainte (**placement**, **élasticité**, **exécution**, **fiabilité**), le composant *Constraint Validator* effectue un mapping entre les contraintes placées sur les composants et les différents modèles correspondants pour actionner un certain nombre d'opérations.

5.4.1.7 SaaS deployment

Les applications *soCloud* sont déployées comme une contribution SCA pouvant contenir en son sein une ou plusieurs contributions. Chaque archive contient une liste de composite SCA à déployer avec leurs implémentations associées comme les fichiers .class, WSDL, BPEL, fichiers de scripts. Le composant *SaaS deployment* fournit une API REST pour déployer les archives d'applications SaaS. Le composant *SaaS deployment* enregistre une copie des archives d'applications déployées dans le dépôt automatiquement.

5.4.1.8 PaaS deployment

Selon le modèle de services de nuages (IaaS ou PaaS) fourni, l'environnement d'exécution (FraSCAti) et l'agent *soCloud* sont conditionnés dans un fichier WAR pour les PaaS et un fichier Jar pour les IaaS. Le composant *PaaS deployment* fournit une API REST qui déploie FraSCAti (l'environnement d'exécution) et l'agent *soCloud*. Cette API permet également de contrôler non seulement l'étendue du déploiement sur un nœud particulier mais aussi la fédération de nœuds ou un ensemble de nœuds correspondant à une contrainte spécifique au déploiement [Par12a].

5.4.1.9 Node Provisioning

Le composant *Node Provisioning* offre un moyen uniforme à *soCloud* pour utiliser les ressources provenant de plusieurs nuages en fonction des besoins. L'efficacité de l'allocation ou du retrait dynamique des ressources dépend d'un nombre de facteurs tels que la granularité de l'allocation (VM ou cpu par exemple), l'algorithme d'allocation, les pannes, etc. [Cha03]. Pour réaliser l'approvisionnement des ressources à travers plusieurs fournisseurs de nuages, le composant Node Provisioning implémente une API qui définit des opérations telles que

addVM (ajout de VM), removeVM (suppression de VM), etc. Ces opérations sont conçues indépendamment des APIs *multi-nuages*. Du point de vue développeur, ces opérations seront les mêmes quelle que soit l'API *multi-nuages* sous-jacente utilisée. Le Node Provisioning utilise la bibliothèque *multi-nuages* jCloud [Apab]. jCloud est implémenté en Java et fournit des unités de calcul et stockages *multi-nuages*. Nous avons utilisé jCloud parce qu'il supporte plusieurs fournisseurs de nuages [20]. L'implémentation du *Node Provisioning* n'est pas restreint à jCloud. Le composant *Node Provisioning* fournit des APIs pour intégrer d'autres bibliothèques d'approvisionnement *multi-nuages*. Lorsque le *Node Provisioning* crée de nouveaux pools de ressources, les informations suivantes sont spécifiées :

— Nom : c'est le nom qui identifie le nouveau pool de ressources créé.
— CPU : c'est le cpu *partagé* et *maximal* dont dispose le pool de ressources. Cpu *partagé* se réfère à la capacité de cpu utilisé dans le pool de ressources. *Cpu maximal* indique la capacité maximale de cpu dont dispose le pool de ressources.
— Mémoire : c'est la mémoire *partagée* et *maximale* dont dispose le pool de ressources. Mémoire *partagée* se réfère à la quantité de mémoire consommée dans le pool de ressources. Mémoire *maximale* indique la quantité *maximale* de mémoire dont dispose le pool de ressources.

5.4.2 Gestion de l'élasticité comme un système autonome

Le contexte de l'environnement d'exécution des applications déployées à travers plusieurs nuages peut changer rapidement en quelques minutes voire quelques secondes. Pour gérer un système dans ce contexte changeant aussi rapidement, il est essentiel de réagir de manière dynamique et automatique pour réguler les événements qui surviennent ou les anticiper. Les concepts de systèmes autonomes et de contrôle de processus peuvent être utilisés pour mettre en œuvre un système qui connait son propre état et qui réagit à son changement. La partie essentielle d'un tel système est le *Contrôleur* qui est externe aux environnements observés. Les responsabilités d'un contrôleur d'élasticité sont de surveiller le système, d'analyser les métriques, de planifier des actions et de les exécuter. Ceci est connu sous le nom de boucle de contrôle MAPE-K (Monitoring, Analyse, Plan, Executing -Knowledge) [IMB06] introduit par IBM, traduisible en français par (Surveillance, Analyse, Planification, Exécution). Pour adresser le problème de l'élasticité *multi-nuages*, nous proposons un système de *Contrôle* hybride qui combine à la fois les mécanismes réactif et proactif.

La plateforme *soCloud* gère l'élasticité aux niveaux IaaS et PaaS de la même façon. En effet, la gestion de l'élasticité dans *soCloud* ne se focalise pas sur une couche spécifique (IaaS ou PaaS) du nuage. *soCloud* utilise les ressources en passant par la couche d'abstraction fournie par le composant *Node provisioning* qui présente une manière uniforme d'utiliser les ressources approvisionnées par les couches IaaS et PaaS. *soCloud* offre la capacité de mise à l'échelle pour des applications en allouant plus de ressources selon les besoins. Par exemple, la plateforme *soCloud* peut ajouter plus de ressources si elle détecte une dégradation de la

20. http ://www.jClouds.org/documentation/reference/supported-providers

performance de l'application. Toutefois, si les ressources sont sous-utilisées, le redimensionnement est nécessaire. Cette caractéristique est gérée comme une boucle de contrôle autonome par la plateforme *soCloud*.

En pratique, l'approvisonnement de ressources de nuages n'est pas instantanée. Déployer et démarrer une nouvelle machine virtuelle peut prendre quelques minutes [Pen12, RJ11]. Comme décrit dans le chapitre 3 à la page 23, l'élasticité *verticale* s'opère plus rapidement que l'*horizontale*. Pour réduire le temps d'approvisionnement, la plateforme *soCloud* dispose d'un pool dynamique de machines virtuelles pré-approvisionnées qui sont éteintes. Le mécanisme d'élasticité s'appuie sur le composant d'approvisionnement de *soCloud* qui est doté d'une intelligence lui permettant d'allouer des ressources de manière opportune. Par exemple, chez Amazon EC2 et Windows Azure, le prix de la location de ressources matérielles s'incrémente toutes les heures. Ainsi, louer des ressources matérielles entre 12h et 13h30 coûtera le même prix que la location entre 12h et 14h. Lorsqu'une ressource n'est plus utilisée par une application, elle est affectée au pool tampon (car heure entamée est égale à heure payée) avant d'être définitivement libérée. Ces machines virtuelles prêtes à être démarrées fournissent un tampon (buffer) en cas de hausse soudaine de l'activité, tout en garantissant une assurance supplémentaire au cas où une défaillance inattendue de ressources matérielles survient.

Comme décrit précédemment, le mécanisme d'élasticité se déroule suivant les phases de la boucle de contrôle (*surveillance*, *analyse*, *planification* et *exécution*).

5.4.2.1 Phase de surveillance (monitoring)

Les applications *soCloud* déployées sont surveillées et leurs configurations peuvent être ajustées en se basant sur les métriques collectées par le composant de *surveillance* qui est déployé dans le même environnement d'exécution que les applications. Le composant de *surveillance* est à la fois intrusif et non-intrusif aux applications. Il est intrusif car il peut instrumenter l'application à l'exécution et son environnement d'exécution. Il est non-intrusif car il surveille les phénomèmes externes à l'application comme le trafic réseau, le pourcentage de cpu utilisé, la quantité de mémoire consommée.

5.4.2.2 Phase d'analyse (analyse)

Les données surveillées sont analysées par le composant *Workload Manager* pour détecter des anomalies. L'analyse des données peut indiquer qu'un ou plusieurs critères (par exemple, l'utilisation de la fréquence CPU inférieur à 80%) acceptables par le système ne sont pas respectés.

5.4.2.3 Phase de planification (plan)

Compte tenu de la situation, le composant *Contrôleur* va entrer dans la phase de planification dans le but de créer un plan d'action afin de ramener les valeurs des métriques à l'état normal. Ce plan peut être basé sur un ensemble de règles qui régissent les opérations du composant *Contrôleur* (réactif) ou sur un modèle élaboré qui agit sur le rétablissement du comportement du système (proactif).

5.4.2.4 Phase d'exécution (executing)

Dans la phase d'exécution le composant *Contrôleur* délègue aux composants *SaaS Deployment*, *PaaS Deployment*, *Node Provisioning* et *Load Balancer* des actions décidées dans la phase de planification. Une fois exécutées, ces actions vont causer un changement dans le comportement du système qui le notifie au composant *Contrôleur* dans la séquence de boucle de contrôle.

Dans notre plateforme *soCloud*, l'élasticité du système est aussi géré dynamiquement en utilisant un mécanisme *proactif*. Toutefois, dans d'autres cas particuliers, le développeur doit pouvoir définir des règles d'élasticité associées à son application. L'élasticité est gérée dans ce cas en utilisant un mécanisme *réactif*. Ces règles sont définies dans l'architecture de l'application et surveillées par la plateforme *soCloud*. Chaque règle est constituée d'une condition ou d'un ensemble de conditions qui doivent être surveillées et d'actions qui sont déclenchées. Ces règles d'élasticité sont également gérées par la boucle de contrôle autonome de la plateforme *soCloud*. Afin de mettre en œuvre l'élasticité proactive, nous avons besoin de garder la trace de la fréquence d'utilisation des ressources gérées et des statistiques sur les applications déployées sur la plateforme *soCloud*. Ainsi, nous utilisons un système proactif qui s'appuie sur le taux de charge de travail courant pour détecter des conditions de surcharge. Le taux de charge de travail est mesuré en surveillant le nombre d'utilisateurs connectés via le composant *Load Balancer*. Pour maintenir des statistiques sur l'utilisation des applications utilisées, nous calculons dynamiquement l'exponentielle de la moyenne pondérée des requêtes sur un intervalle de temps, de la même façon que le protocole TCP calcule le temps aller-retour (round-trip time) [Kar87, Fel04] des paquets. Plus précisement, on calcule la moyenne du temps inter-arrivées (inter-arrival time) en utilisant la formule suivante :

$$f(t) = (1 - \alpha) * f(t-1) + \alpha * (\delta t(t) - \delta t\,(t-1)) \tag{5.5}$$

Le temps inter-arrivées de chaque requête (hit) est représenté par $\delta t(t)$. La constante α est un facteur de lissage qui met plus de poids sur des échantillons récents que sur des anciens. Nous avons utilisé la valeur de α = 0.12 qui est recommandée pour TCP [Gro00]. Pour détecter les surcharges ou les sous-charges, nous avons utilisé un dispositif basé sur le seuil qui est utilisé pour déclencher l'allocation de ressources. Il faut noter que le seuil obtenu en utilisant l'équation 5.5 varie d'une application à l'autre.

Nous proposons un algorithme permettant d'orchestrer l'élasticité. Cet algorithme est basé sur le modèle MAPE-K des systèmes autonomes. L'objectif de cet algorithme est de répondre à quatre questions :

— Quand faut-il passer à la mise à l'échelle ?
— Quels types d'élasticité (verticale ou horisontale) faut-il utiliser ?
— Quelle quantité de ressources est nécessaire ?
— Comment passer à la mise à l'échelle ?

L'algorithme 1 décrit un mécanisme de boucle autonome qui s'exécute tant que le système *soCloud* est actif. La ligne 1 correspond à la collecte des métriques dans le système, la ligne 2 correspond à l'analyse qui est effectuée sur les métriques collectées. Les lignes 4-7 représentent une condition qui compare les indicateurs obtenus aux seuils après l'analyse. Lorsque la condition est vérifiée, il y a un plan qui est mis en place (voir ligne 5) et une exécution de ce dernier s'effectue à la ligne 6. Les fonctions de surveillance, d'analyse, de planification et d'exécution sont décrites respectivement dans les algorithmes 2, 3, 4 et 5. L'algorithme 2 décrit une séquence de collecte et d'aggrégation de données qui sont retournées. L'algorithme 3 met à jour sa base de connaissances sur les métriques renvoyées par la fonction *monitor* (voir algorithme 2), ces métriques sont analysées puis stockées dans la base de connaissances. Une fois la mise à jour effectuée, les indicateurs sont retournés. L'algorithme 4 décrit la fontion de planification. Cette fonction récupère les indicateurs renvoyés par la fonction *plan* pour planifier la mise à l'échelle. Avant la planification, la fonction récupère le type d'élasticité (verticale ou horizontale) à mettre en œuvre (voir ligne 3 algorithme 4). Une fois la planification réalisée, le plan est renvoyé. L'algorithme 5 se charge d'exécuter le plan qui a été délégué.

Algorithm 1 CONTRÔLE DE L'ÉLASTICITÉ de la boucle de contrôle MAPE-K.

```
1: while systemIsUp do
2:     metrics ← monitor()
3:     indicators ← analyse(metrics)
4:     if indicators.values > systemThreshold then
5:         scalingPlan ← plan(indicators)
6:         execute(scalingPlan)
7:     end if
8: end while
```

Algorithm 2 FONCTION SURVEILLANCE de la boucle de contrôle MAPE-K.

```
1: procedure monitor()
2:     metrics ← collecte()
3:     agMetrics ← agregate(metrics)
4:     return agMetrics
5: end procedure
```

Algorithm 3 FONCTION ANALYSE de la boucle de contrôle MAPE-K.

procedure $analyse(metrics)$
    $Update\ Knowledge\ based\ on\ metrics$
    $Research\ pattern\ in\ metrics$
    $Set\ current\ indicators\ based\ on\ Knowledge$
    $Update\ Knowledge$
    Return $indicators$
end procedure

Algorithm 4 FONCTION PLAN de la boucle de contrôle MAPE-K.

procedure $plan(indicators)$
    $scaling \leftarrow scale(indicators)$
    $elasticity \leftarrow kindElasticity()$
    if $scaling$ **then**
        if $elasticity == velasticity$ **then**
            $scalingUp \leftarrow true$
            $addCpu \leftarrow (currentCpu * spikeMultiplier)$
            $addMem \leftarrow (currentMem * spikeMultiplier)$
            $addNet \leftarrow (currentNet * spikeMultiplier)$
            $scalingPlan \leftarrow perform(scalingUp, addCpu, addMem, addNet)$
        else
            $scalingOut \leftarrow true$
            $addVm \leftarrow CEIL(currentVm * spikeMultiplier)$
            $scalingPlan \leftarrow perform(scalingOut, addVm)$
        end if
    else
        if $elasticity == velasticity$ **then**
            $scalingDown \leftarrow true$
            $removeCpu \leftarrow (currentCpu * spikeMultiplier)$
            $removeMem \leftarrow (currentMem * spikeMultiplier)$
            $removeNet \leftarrow (currentNet * spikeMultiplier)$
            $scalingPlan \leftarrow perform(scalingDown, removeCpu, removeMem, removeNet)$
        else
            $scalingIn \leftarrow true$
            $removeVm \leftarrow CEIL(currentVm * spikeMultiplier)$
            $scalingPlan \leftarrow perform(scalingIn, removeVm)$
        end if
    end if
    Return $scalingPlan$
end procedure

Algorithm 5 FONCTION EXÉCUTION de la boucle de contrôle MAPE-K.

1: **procedure** $execute(scalingPlan)$
2: Return $perform(scalingPlan)$
3: **end procedure**

5.4.3 Déploiement distribué de *soCloud*

La plateforme *soCloud* peut être déployée de plusieurs manières selon le niveau de disponibilité que l'administrateur de la plateforme souhaite mettre en place. En effet, la plateforme *soCloud* offre la haute disponibilité en se répliquant sur différents nuages comme le montre la figure 5.13. Dans notre implémentation, nous supposons que les services de nuages peuvent tomber en panne et que le service défaillant peut être rétabli plus tard. L'administrateur système a la possibilité de définir la politique de déploiement en spécifiant le nombre de réplications ainsi que les nuages sur lesquels la plateforme *soCloud* doit être déployée. Par exemple dans la figure 5.13, il y a une réplication du *master* soCloud.

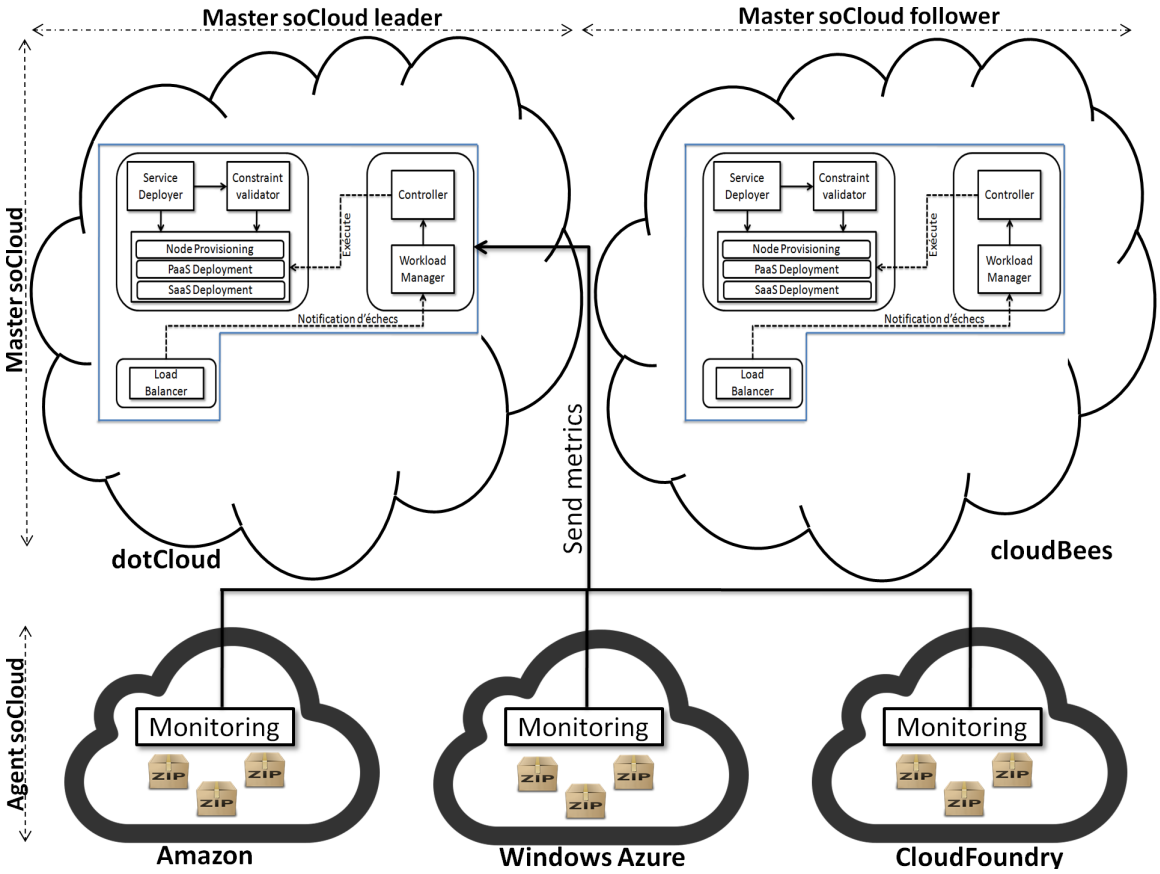

Figure 5.13 – Exemple de déploiement de la plateforme *soCloud*.

Le déploiement a été effectué en trois étapes. Dans la première étape, le *master* soCloud

est déployé dans dotCloud [dot13]. Ensuite, dans la deuxième étape, le *master soCloud*, qui est déployé dans dotCloud, déploie à son tour de manière dynamique et autonome un second *master* soCloud dans CloudBees [clo13]. Automatiquement, le premier *master* soCloud déployé dans dotCloud devient le *leader* et le second le *follower*. Le *master* soCloud *leader* est actif tandis que le *master* soCloud *follower* est passif. Par actif, nous nous référons au *master* soCloud qui traite toutes les opérations dans le système ; par passif, le *master* soCloud qui est en état de veille et qui est utilisé comme réplica. A ce stade, seul le *master* soCloud et sa réplication sont déployés. Enfin, le *master* soCloud *leader* approvisionne un nouveau nœud sur lequel il déploie à la fois l'environnement d'exécution (FraSCAti) et l'*agent* soCloud. Lorsque l'*agent* soCloud est déployé, ce dernier utilise le mécanisme de recherche de services (service lookup) pour trouver sur quel *master* soCloud il doit se connecter pour envoyer les métriques collectées. Le service lookup est chargé de tenir le registre de l'adresse du *master* soCloud sur lequel l'*agent* soCloud va se connecter. Le modèle de conception du mécanisme de recherche de service utilisé pour localiser le *master* soCloud est mis en œuvre en utilisant JNDI lookup [Ter00]. En tenant compte du coût élevé (en terme de ressources) de la recherche JNDI pour un service, le service lookup utilise la technique de cache (voir figure 5.14). La première fois que l'*agent* soCloud veut se connecter au *master* soCloud, le service lookup recherche dans JNDI et met en cache l'adresse du *master* soCloud *leader*. Ainsi, d'autres recherches du *master* soCloud via le service lookup se feront dans le cache ce qui améliorera la performance de la plateforme *soCloud* à grand échelle. A tout moment, lorsqu'un nouveau *master* soCloud est mis à jour dans le registre à l'exécution, le service lookup met le cache à jour automatiquement. Le service lookup dispose de l'adresse explicite du *master* soCloud ; ainsi l'*agent* soCloud n'a pas besoin de connaitre l'adresse explicite du *master* soCloud qui lui s'enregistre sur le service lookup. Le mécanisme du service lookup découple l'*agent* et *le master* soCloud des problèmes d'infrastructures tels que la communication, la transparence, la découverte de services. Le cache a été implémenté en utilisant Google Fusion Table [Gon10] et la référence multiple dynamique de FraSCAti. Nous utilisons Google Fusion Table pour conserver l'adresse du *master* soCloud *actif* et via la référence multiple dynamique fournie par FraSCAti nous ajoutons une référence au nouveau composant dynamiquement. Afin de garantir une plus grande sureté, *soCloud* externalise la persistance de son état de fonctionnement hors de la plateforme. Particulièrement, la persistance est effectuée en utilisant Google Fusion Table. La plateforme *soCloud* fournit un mécanisme appelé "système de vérification" (Health checking) par lequel un composant notifie son bon fonctionnement. Ce mécanisme est mis en œuvre en utilisant une notification (push) qui teste à la fois si un composant est joignable et si la pulsation (heartbeat) est perceptible. Le *master* et *agent* soCloud ont tous deux besoin du container (FraSCAti) pour s'exécuter. Lorsque le système s'agrandit (le nombre d'applications ou la quantité de charge augmente), seule la dernière étape de la phase de déploiement est répétée, c'est-à-dire le déploiement de nouveaux *agents soCloud*.

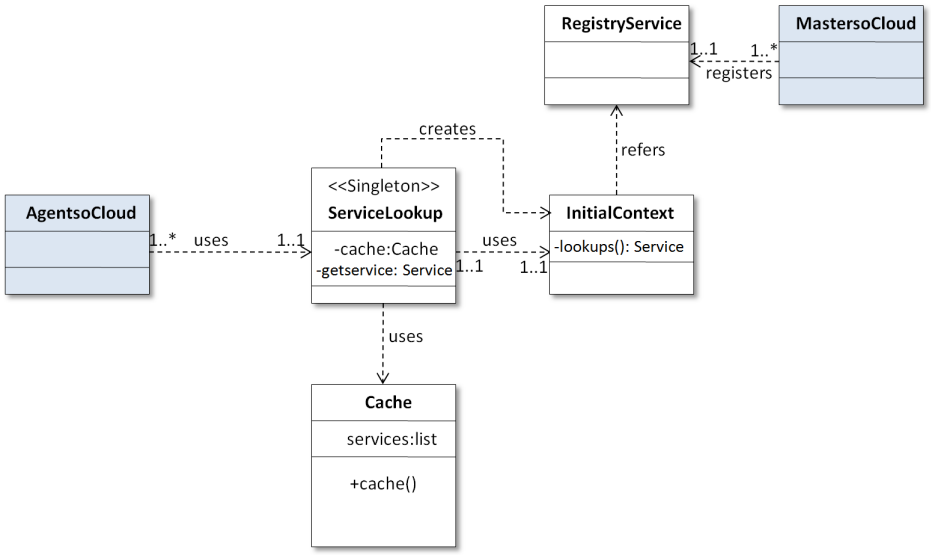

Figure 5.14 – Service lookup.

5.4.4 Tolérance de *soCloud* aux défaillances

Certains intergiciels [Cap05, Lit88] ne supportent pas la perte du coordinateur qui joue le rôle de service d'information et d'ordonnanceur. C'est aussi le cas de toutes les approches centralisées. D'une manière générale, les approches *multi-nuages* complètement réparties tolèrent mieux la défaillance ou la perte de nuage puisque les services ne sont pas centralisés dans un seul nuage.

Il existe plusieurs catégories de scénarii de défaillance à prendre en compte. Mais toutes partagent les caractéristiques de la préparation aux pannes et ensuite de la récupération. La plateforme *soCloud* assure la haute disponibilité de deux façons : i) au niveau de la plateforme soCloud et ii) au niveau des applications déployées.

5.4.4.1 Niveau plateforme *soCloud*

Le *master* soCloud *actif* est appelé le *leader* et le soCloud *master passif* est appelé le *follower*. Le processus d'élection du leader permet au système d'indiquer quel soCloud *master* aura la responsabilité de l'exécution. Les *masters* soCloud *leader* et *follower* sont synchronisés de telle sorte que lorsque le leader échoue (voir figure 5.15 (1)), une élection est organisée automatiquement pour élire un nouveau leader. Plus précisement nous utilisons la synchronisation Wait-Free qui est appropriée aux systèmes qui tolèrent les défaillances et aux applications en temps réél [Gar05]. Dans le cas où l'administrateur a défini une seule réplication du *master* soCloud, le *master* soCloud *follower* est automatiquement élu. L'élection du leader est organisée et supervisée par le composant *Contrôleur*. L'algorithme de l'élection du leader est

décrite dans le listing 6. Lorsqu'il y a plus de deux *master* soCloud instanciés, le premier remarquant qu'il n'y a pas de leader, initialise l'élection en envoyant un message qui contient son numéro de processus et le message "election" à tous les autres *masters* soCloud. Une fois le message reçu, chaque *master* soCloud compare son numéro de processus au numéro reçu. Si le numéro reçu par le *master* soCloud est inférieur à son propre numéro, ce dernier répond en envoyant à son tour un message contenant son numéro de processus au *master* soCloud qui a organisé l'élection. Si le *master* soCloud qui organise l'élection ne reçoit aucun message, il devient le leader et le notifie aux autres *master* soCloud. Dans le cas contraire, il fait une comparaison des numéros reçus et diffuse l'identité du nouveau leader. Toutefois, la plateforme *soCloud* n'est pas restreinte à cet algorithme l'administrateur système a la possibilité de définir d'autres algorithmes (par exemple, l'algorithme de Chang-Roberts [Cha79], l'algorithme de Malpini [Mal00]) selon ses besoins. Les élections ont lieu entre deux composants qui ont la même fonction (par exemple, deux composants *Workload Manager*, deux composants *Node Provisioning*, etc.). Ainsi le composant *Contrôleur* organise une élection en comparant le temps de latence. En utilisant cette stratégie, tous les composants du *master* soCloud *leader* se retrouvent dans le même nuage et les composants du *master* soCloud *follower* se retrouvent eux aussi regroupés dans un autre nuage. Lorsque le *master* soCloud *follower* tombe en panne (voir figure 5.15 (2)), le *master* soCloud *leader* déploie automatiquement un nouveau *master* soCloud *follower*. Lorsque la connexion entre les *masters* soCloud *leader et follower* est interrompue (voir figure 5.15 (3)), les deux *masters* soCloud *leader et follower* instancient deux nouveaux *masters*. Chaque nouvelle paire (leader/follower) master connectée organise une nouvelle élection pour élire un leader. Dans notre cas, les deux paires ne communiquent pas ensemble et donc chaque paire ne connait pas l'existence de l'autre. Ces deux paires essaient de s'enregistrer sur le service lookup en utilisant le Google Fusion Table. Avant de s'enregistrer sur le service lookup, le nouveau *master* soCloud *leader* vérifie si l'ancien soCloud *master* enregistré est actif. Si c'est le cas il avorte son intention d'enregistrement, sinon, il s'enregistre à la place de l'ancien soCloud *master*. Il se pourrait qu'au moment de l'enregistrement des conflits d'accès (lock contention) causés par les deux paires qui essaient de s'enregistrer simultanément (voir figure 5.16) surviennent. Pour résoudre ce problème, la plateforme *soCloud* utilise l'algorithme d'élection entre les deux paires pour en exclure une.

5.4.4.2 Niveau applications déployées

De la même manière qu'au niveau de la plateforme *soCloud*, le développeur a la possibilité de définir le nombre d'instances d'applications à déployer via les annotations. Chaque application déployée avec la plateforme *soCloud* est répliquée sur différents nuages. Le mécanisme de basculement (fail-over) est assuré par le composant *équilibreur de charge*. Lorsqu'une instance de l'application déployée est défaillante, l'équilibreur de charge bascule automatiquement sur une autre instance et le composant *Contrôleur* prend la décision d'en instancier une nouvelle pour remplacer l'instance de l'application défaillante.

D'une manière générale, l'automatisation du basculement permet à la plateforme *soCloud* d'assurer une reprise rapide en cas de défaillance avec la plupart des pannes. Mais

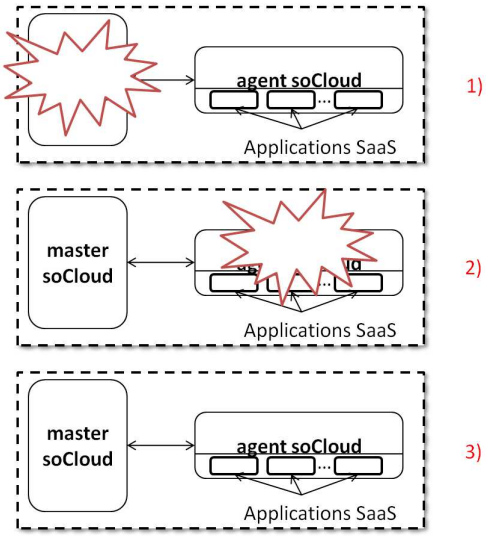

Figure 5.15 – Différents cas de figures de pannes.

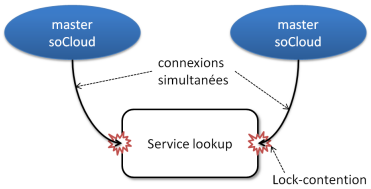

Figure 5.16 – Verrou d'accès.

cette automatisation n'est pas suffisante. Nous surveillons également notre système pour déceler toute sorte de conditions d'erreur (un exemple d'erreur est une transaction de déploiement d'applications qui échoue plus de trois fois sur la même opération). Avec la combinaison des deux niveaux (niveau *soCloud*, niveau application) de disponibilité, la plateforme *soCloud* assure la haute disponibilité *multi-nuages* discutée dans le chapitre 3.

En résumé, l'objectif des mécanismes de tolérance aux pannes mis en œuvre dans la plateforme *soCloud* garantit la disponibilité de l'application dans son ensemble.

5.4.5 Mécanisme de reprise en cas de défaillance

La méthode utilisée par *soCloud* pour assurer la tolérance aux fautes est le point de contrôle[21] (check-point). Le point de contrôle peut être *local* à un processus ou *global* à un

21. Le point de contrôle est un instantané de l'état d'un processus, enrégistré sur une mémoire non volatile pour survivre aux échecs [Wan97].

Algorithm 6 ALGORITHME D'ÉLECTION DU MASTER *soCloud* LEADER.

```
1:  procedure startElection()
2:      electionMsg ← [processNumber, election]
3:      isFound ← false
4:      responses ← None
5:      responses ← send(electionMsg)
6:      if responses is none then
7:          isFound ← true
8:          broadcast(coordinatorId, isFound)
9:      else
10:         coordinatorId ← compare(response)
11:         isFound ← true
12:         broadcast(coordinatorId, isFound)
13:     end if
14: end procedure
```

système. Avec la plateforme *soCloud*, nous utilisons le point de contrôle *global*. Nous utilisons Google Fusion Table pour enregistrer l'état global de fonctionnement du système. Ainsi lorsqu'une panne survient, l'ensemble du système peut être restauré. Pour enregistrer l'état global du système, la plateforme *soCloud* utilise la méthode de point de contrôle coordonné [Bou03]. En effet, il y a quelques désavantages pour l'utilisation du point de contrôle non coordonné comparé au coordonné [Gar05]. Tout d'abord, pour un point le contrôle coordonné, il suffit de garder les états globaux instantanés les plus récents dans une base de données stable. Tandis que pour le point de contrôle non coordonné, un schéma de traitement plus complexe est nécessaire. De plus, en cas de défaillance, le procédé de récupération du point de contrôle coordonné est plus simple à mettre en œuvre.

5.4.6 Intégration avec d'autres fournisseurs de nuages existants

La plateforme *soCloud* est actuellement déployée sur dix environnements de nuages [22]. Le déploiement a été effectué aussi bien sur des PaaS que sur des IaaS comme illustré dans la figure 5.17. Avec l'infrastructure en tant que service, les ressources sont fournies par Windows Azure, DELL KACE, Amazon EC2 et notre infrastructure privée Eucalyptus. Sur ces infrastructures, nous avons installé des intergiciels et systèmes d'exploitation dont a besoin la plateforme *soCloud* à savoir une distribution Linux, un environnement d'éxécution Java, un container Web (Tomcat ou Jetty, etc.), FraSCAti. La plateforme *soCloud* est aussi déployée sur des plateformes en tant que service telles que CloudBees [23], OpenShift [24], dot-

22. **http ://socloud.soceda.cloudbees.net**
23. http ://www.cloudbees.com/
24. https ://openshift.redhat.com

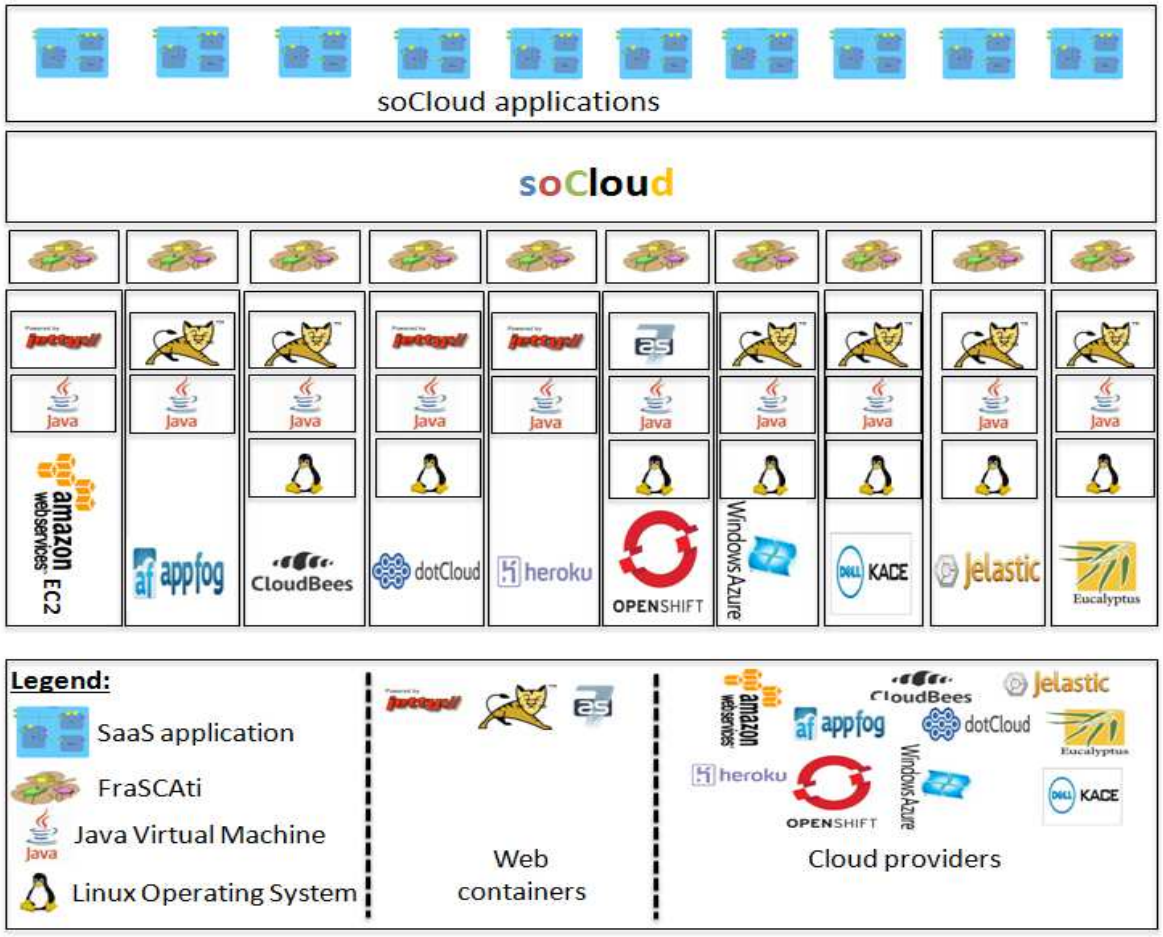

Figure 5.17 – Déploiement de la plateforme *soCloud* sur dix fournisseurs de nuages.

Cloud [25], Jelastic [26], Heroku [27] et Appfog [28]. Toutefois, il existe des plateformes de nuage qui imposent certaines restrictions. Par exemple, Google App Engine (GAE) offre un environnement d'exécution Java limité, la manipulation des threads ou JMX n'est pas possible, lorsque l'application ne peut pas être chargée dans la mémoire en une seconde, l'erreur 5000 est retournée [Car10], le protocole HTTPS ne peut pas être utilisé avec le nom de domaine préenregistré, la durée de vie d'une requête est limitée à 30 secondes, les requêtes GET ou POST de la plateforme GAE vers d'autres sites sont avortées si leur durée de vie excède 5 secondes.

25. https ://www.dotcloud.com/
26. http ://jelastic.com/
27. http ://www.heroku.com/
28. http ://www.appfog.com/

5.4.7 Prise en charge d'autres types d'applications SaaS

La plateforme *soCloud* ne se limite pas au déploiement, à l'exécution et la gestion des applications orientées service dans un environnement *multi-nuages*. Elle peut prendre en charge d'autres types d'applications. Par exemple, *soCloud* pourrait déployer, exécuter et gérer une archive Java EE (WAR ou EAR). Notre modèle décrit dans le chapitre 4 peut être appliqué à d'autres types d'applications. La différence avec une archive Java EE consiste à remplacer la contribution SCA au format Zip par l'archive Java EE WAR comme le montre le listing 5.1. Comme on peut le remarquer, la structure du descripteur d'application est inchangée, seul le fichier binaire de la contribution SCA "presentation.zip" est remplacée par l'archive Java EE "presentation.war". Le listing 5.1 montre que la plateforme *soCloud* offre une manière uniforme de déployer, d'exécuter et de gérer d'autres types d'applications.

```
1<composite name="Application-3-tier">
2  <component name="Presentation">
3    <implementation.contribution contribution="presentation.war"/>
4    <annotation name="location">Singapour</annotation>
5    <annotation name="replication">3</annotation>
6    <annotation name="elasticity">
7    scaling in
8    when (totalCost(computeCost,24 h) > 200)
9    </annotation>
10  </component>
11</composite>
```

Listing 5.1 – Gestion d'autres types d'applications SaaS.

5.4.8 Extension de la plateforme *soCloud*

La plateforme *soCloud* fournit trois APIs extensibles :

1. **Bibliothèques *multi-nuages*.** Actuellement, la Bibliothèque *multi-nuages* supportée par la plateforme *soCloud* est jCloud. Toutefois, *soCloud* fournit l'API *MultiCloudDrivers* permettant de supporter d'autres Bibliothèques *multi-nuages*.
2. **Algorithme d'équilibreur de charge.** Le développeur a la possibilité d'étendre l'API *LBAlgorithme* pour utiliser d'autres types d'algorithmes sophistiqués d'équilibreur de charge existants.
3. **Langage d'élasticité.** Notre langage d'élasticité décrit dans la section 4.5 est extensible. L'extensibilité se fait à l'aide de l'API *SelParser* où le développeur peut ajouter des fonctionnalités au parseur existant.

5.5 Conclusion

Dans ce chapitre, nous avons décrit l'architecture et la mise en œuvre de notre plateforme *soCloud* pour prendre en compte les exigences architecturales d'applications orientées services dans un environnement *multi-nuages*. Nous proposons une plateforme en tant que service *multi-nuages* permettant de déployer, d'exécuter et de gérer des applications orientées service dans un environnement *multi-nuages*. *soCloud* est une plateforme orientée services à base de composants pour gérer la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité multi-nuages*. *soCloud* est une plateforme répartie qui fournit un modèle pour construire n'importe quelle application SaaS *multi-nuages*. Non seulement, nous avons décrit chaque composant de l'architecture de *soCloud* mais nous avons discuté aussi les choix d'implémentation et les technologies utilisées pour les mettre en œuvre. Pour gérer l'élasticité, *soCloud* propose la combinaison des méthodes réactive et proactive en utilisant une boucle de contrôle autonome pour l'automatisation du mécanisme d'élasticité. La haute disponibilité dans *soCloud* s'opère à deux niveaux : au niveau plateforme où on utilise la réplication et au niveau application où on utilise l'équilibreur de charge pour passer d'une instance de l'application à une autre en cas de défaillance.

Le principal défi de tout système réparti est de garder toutes les données synchronisées et cohérentes. Pour cela, la plateforme *soCloud* utilise un mécanisme de coordination. Pour gérer la coordination, *soCloud* adopte l'approche wait-free aux problèmes de coordination des composants. L'architecture de la plateforme *soCloud* est ouverte au sens où elle ne présuppose pas un ensemble fini et figé de fonctionnalités. Bien entendu, de nouvelles fonctionnalités peuvent être ajoutées par les développeurs en fonction des besoins en utilisant les APIs fournies par *soCloud* pour l'extension. Nous avons démontré que la plateforme *soCloud* s'intègre facilement à d'autres plateformes (PaaS) existantes.

Enfin, des perspectives d'évolution peuvent être envisagées. Premièrement, comme toute approche *multi-nuages*, *soCloud* peut ne pas prendre en compte certaines fonctionnalités qui sont fournies par les nuages fédérés comme les règles d'élasticité, les propriétés d'approvisionnement, etc.). Une approche pour adresser ces problèmes consiste à développer une enveloppe (wrapper), c'est-à-dire une couche logicielle qui fera le pont entre les fonctionnalités originelles et de nouvelles fonctionnalités conforme à la norme choisie. Deuxièmement, certains développeurs ou entreprises peuvent ne pas vouloir adopter une approche basée sur SCA. Bien que notre approche soit basée sur SCA, elle pourrait offrir la possibilité de déployer tout type d'applications SaaS conditionnées en WAR ou EAR par exemple. Troisièmement, les fonctionnalités que fournissent les plateformes PaaS évoluent et sont dynamiques. Nous sommes convaincus que l'utilisation des standards est plus adéquat, mais le manque d'adoption de ces derniers les rend inefficaces. Dans le chapitre suivant, nous présentons les validations des deux contributions de ce manuscrit.

Troisième partie

VALIDATION

Chapitre 6

Validation

"Si le seul outil que vous avez est un marteau, vous avez tendance à voir chaque problème comme un clou."-Abraham Maslow

Sommaire

Dans ce chapitre, nous présentons les évaluations de nos deux contributions qui adressent les défis mentionnés dans cette thèse. Les évaluations sont décrites dans les deux sections suivantes qui correspondent au modèle d'applications *soCloud* et à l'architecture de la plateforme *soCloud* présentés dans les deux chapitres précédents.

6.1 Evaluation du modèle d'applications *soCloud*

Le but principal de cette évaluation est de fournir des prototypes pour valider le modèle d'applications proposé dans le chapitre 4. A cet égard, nous présentons dans cette section trois applications *soCloud* implantées à partir de notre modèle *soCloud* et déployées dans un environnement *multi-nuages*.

6.1.1 Application APISENSE

Afin d'évaluer le modèle d'applications *soCloud*, nous prenons un scénario applicatif de crowdsensing [29] [Gan11] mobile appelée *APISENSE* décrit dans [Had13], dans lequel on analyse les traces d'activité collectées par les scientifiques afin de mieux comprendre le comportement des populations. Depuis quelques années, la nouvelle génération de téléphones portables ou smartphones a révolutionné la collecte des activités humaines. Largement adoptés par les populations, avec plus de 472 millions vendus en 2011 (contre 297 millions en 2010) selon Gartner Institut [Egh12], les smartphones sont devenus un élément central dans la vie des populations. Au delà de leurs fonctions de communication, la capacité de faire des photos et vidéos, les appareils mobiles modernes sont maintenant équipés de plusieurs types de capteurs permettant à une nouvelle catégorie d'applications de détecter et de reconnaitre les activités des utilisateurs dans leur contexte et leurs environnements. En outre, la généralisation des magasins d'applications ou App store fournis par des opérateurs de téléphonie permettent aux scientifiques de proposer de nouvelles applications de détection, d'analyse de comportement des utilisateurs en exploitant l'inscription massive des participants à grande échelle, ce qui n'était pas possible auparavant. Toutefois, la durée de vie de la batterie, la capacité de stockage et la mobilité des smartphones imposent des limites intrinsèques sur la manière de concevoir l'architecture de l'application *APISENSE*. La Figure 4.1 illustre l'architecture de la plateforme *APISENSE* constituée de trois composants : *Sensing Node*, *APISENSE Central Node* et *Sensing Storage*. Le composant *Sensing Node* permet aux smartphones de déverser les données collectées. Le composant *APISENSE Central Node* déploie les tâches sur

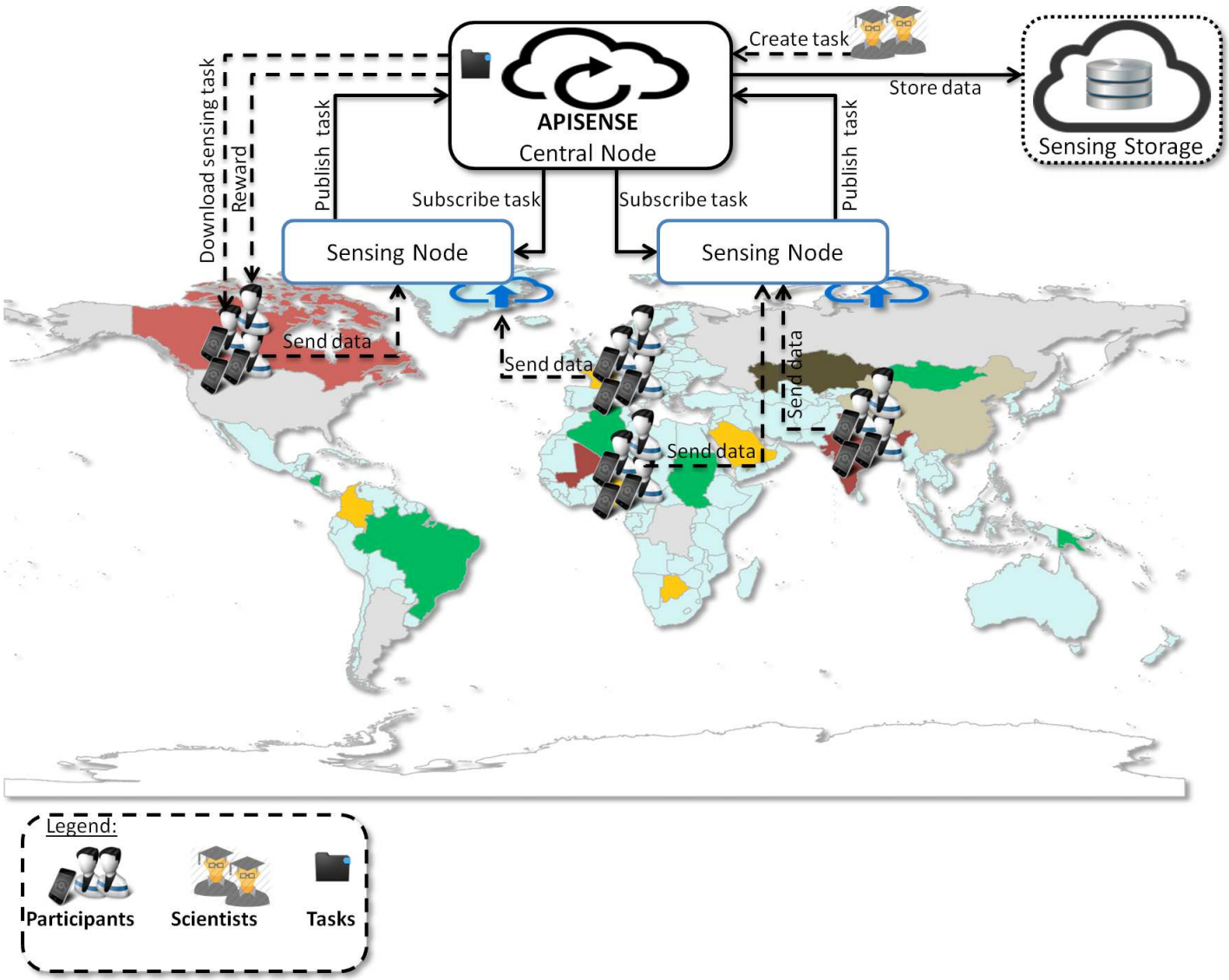

Figure 6.1 – Architecture d'une application *multi-nuages* : scénario de cas d'utilisation.

les smartphones et effectue le traitement métier sur les données. Enfin le composant *Sensing Storage* persiste les données des différentes expérimentations effectuées par des scientifiques.

Le processus d'une campagne de collecte de données se déroule comme suit :

1. Le scientifique déploie des tâches de collecte auprès du composant *APISENSE Central Node*.
2. Ce dernier propage les tâches sur les téléphones des participants préalablement sélectionnés par le scientifique.

29. Collecte de données par la foule.

3. Les données collectées sont envoyées vers le composant *Sensing Node*.
4. Le composant *Sensing Node* renvoie les données collectées depuis les téléphones vers le composant *APISENSE Central Node*.
5. Enfin les données sont stockées dans le composant *Sensing Storage*.

Dans une campagne de collecte pour une expérimentation scientifique, les scientifiques ont ciblé certains participants, par exemple les clients d'un opérateur téléphonique ayant une portée internationale. Ce scénario veut mettre à profit l'approche *multi-nuages* pour fournir une application *APISENSE* flexible capable de s'adapter aux différents cas de figures d'expérimentations.

L'application *APISENSE* doit faire face à des exigences telles que :

— La mobilité des smartphones, facteur très important à prendre en compte lors de la conception de l'architecture. L'application doit être répartie géographiquement afin d'être plus proche des utilisateurs de smartphones pour optimiser les performances.
— La croissance, imprévisible du nombre de smartphones qui envoient les données collectées. Selon les expériences et la densité de la population dans une zone géographique la charge de travail peut augmenter de manière drastique.
— La disponibilité de l'application malgré les pannes. Quelle que soit la robustesse d'un système, les pannes sont inévitables. Elles doivent donc être prises en compte lors de la conception.
— Le contrôle des coûts. Il est très important de tenir compte des incidences financières lors de la conception de l'architecture.

Le listing 6.1 décrit la structure de l'application APISENSE. Il s'agit d'une collecte de données sur les personnes qui habitent la région parisienne et utilisent leur voiture comme moyen de déplacement. Dans cet exemple, les composants *Sensing Node*, *Central Node* et *Sensing Storage* sont décrits respectivement aux lignes 2-10, lignes 12-22 et lignes 23-29. Sur le composant *Sensing Node*, nous avons trois annotations *<annotation name="location">Paris</annotation>*, *<annotation name="replication">2</annotation>* et *<annotation name="elasticity"> scaling out when (numberOfUser > 100000)</annotation>*. L'annotation *location* indique que ce dernier doit être déployé à Paris, l'annotation *replication* indique qu'il faut déployer deux instances du composant et l'annotation *elasticity* définit la règle d'élasticité permettant de passer à l'échelle le composant *Sensing Node* lorsque le nombre d'utilisateurs connectés sur cet dernier est supérieur à 100000. Le composant *Central Node* porte trois annotations *<annotation name="vm">large -> Ubuntu</annotation>*, *<annotation name="location">Lille</annotation>* et *<annotation name="elasticity">scaling out when (timePeriod == 1 min and responseTime > 2s)</annotation>*. L'annotation *vm* indique qu'il faut déployer le composant *Central Node* sur une machine virtuelle de type large sur laquelle est installée un système d'exploitation ubuntu. L'annotation *location* indique qu'il faut déployer le composant *Central Node* sur Lille. L'annotation *elasticity* définit la règle d'élasticité permettant de passer à l'échelle le composant *Central Node* lorsque le temps de réponse est supérieur à 2 secondes sur un intervalle de temps d'une minute. Le composant *Sensing*

Storage porte trois annotations *<annotation name="replication">2</annotation>*, *<annotation name="database">MySQL</annotation>* et *<annotation name="location">France</annotation>*. L'annotation *replication* indique qu'il faut déployer 2 instances du composant *Sensing Storage*. L'annotation *database* définit la base de données MySQL dont requiert le composant *Sensing Storage*. L'annotation *location* indique qu'il faut déployer le composant *Sensing Storage* n'importe où en France.

```
<composite name="Application-APISENSE">
  <component name="SensingNode">
    <implementation.contribution contribution="sensingnode.zip"/>
    <reference name="compute" target="CentralNode/compute"/>
    <reference name="storage" target="SensingStorage/storage"/>
    <annotation name="location">Paris</annotation>
    <annotation name="replication">2</annotation>
    <annotation name="elasticity">
        scaling out when (numberOfUser > 100000)
    </annotation>
  </component>
  <component name="CentralNode">
    <implementation.contribution contribution="centralnode.zip"/>
    <service name="compute"/>
    <reference name="storage" target="SensingStorage/storage"/>
    <annotation name="vm">large -> Ubuntu</annotation>
    <annotation name="location">Lille</annotation>
    <annotation name="elasticity">
      scaling out when (timePeriod == 1 min and responseTime > 2s)
    </annotation>
    <annotation name="closer">SensingNode</annotation>
  </component>
  <component name="SensingStorage">
    <implementation.contribution contribution="sensingstorage.zip"/>
    <service name="storage"/>
    <annotation name="replication">2</annotation>
    <annotation name="database">MySQL</annotation>
    <annotation name="location">France</annotation>
  </component>
</composite>
```

Listing 6.1 – Le descripteur de l'application trois-tiers en utilisant le modèle *soCloud*.

Grâce au modèle *soCloud*, nous avons pu décrire l'architecture de notre application APISENSE et exprimer des besoins non-fonctionnels tels que le placement, l'exécution, la disponibilité et l'élasticité de cette application.

6.1.2 Application pour fédérer plusieurs moteurs CEP répartis

Cette section décrit l'application DiCEPE [Par12b], une autre application implantée avec le modèle *soCloud*. DiCEPE est une application pour fédérer des moteurs de CEP hétérogènes. Elle est conçue pour faciliter l'intégration des moteurs de CEP [Luc08] qui interagissent en utilisant plusieurs protocoles de communication (HTTP, WS-NOTIFICATION et JMS). L'expérience que nous menons à travers l'application DiCEPE à pour but de valider qu'une topologie peut être exprimée facilement en utilisant le modèle d'applications *soCloud*.

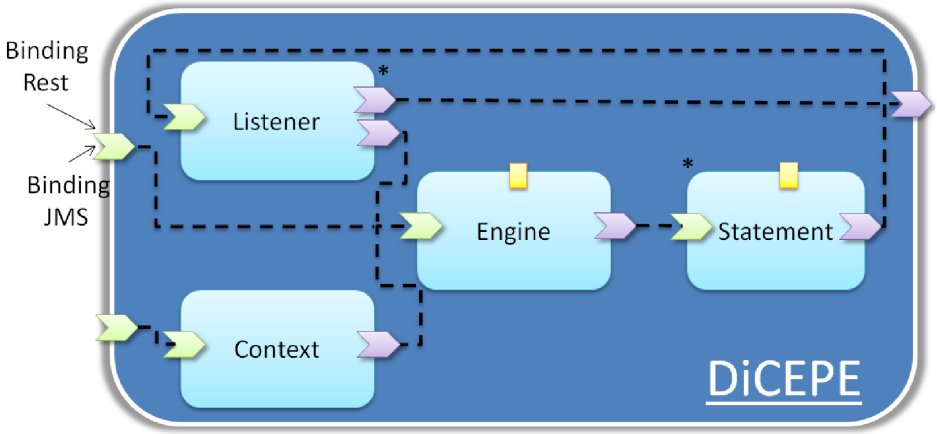

Figure 6.2 – Architecture SCA de l'application DiCEPE.

La figure 6.2 illustre l'architecture SCA de l'application DiCEPE qui est constituée de quatre composants SCA :

1. *Engine* : ce composant encapsule le moteur de CEP et exécute des opérations sur les événements telles que la création, la lecture, la transformation, l'agrégation, la corrélation et la suppression de doublons.
2. *Statement* : il permet de gérer des règles qui sont mises en œuvre en utilisant le déclencheur lorsqu'une requête sur un événement complexe est satisfait.
3. *Listener* : ce composant génère un autre événement complexe lorsqu'une action est détectée.
4. *Context* : il collecte les différentes règles exprimées et qui s'exécutent sur le moteur de CEP.

Pour valider comment le modèle d'applications *soCloud* adresse les exigences d'approvisionnement, la figure 6.3 représente une topologie de déploiement composée de 7 DiCEPEs distribués. La figure 6.3 est un exemple d'application Big Data [Zik11] de type MapReduce [Dea08] basée sur les moteurs CEP. Sur cette figure, nous avons une base de données qui contient les données météorologiques collectées sur les cinq continents. Ces données proviennent du National Climatic Data Center (NCDC web site[30]). La base de données pour

30. http://www.ncdc.noaa.gov

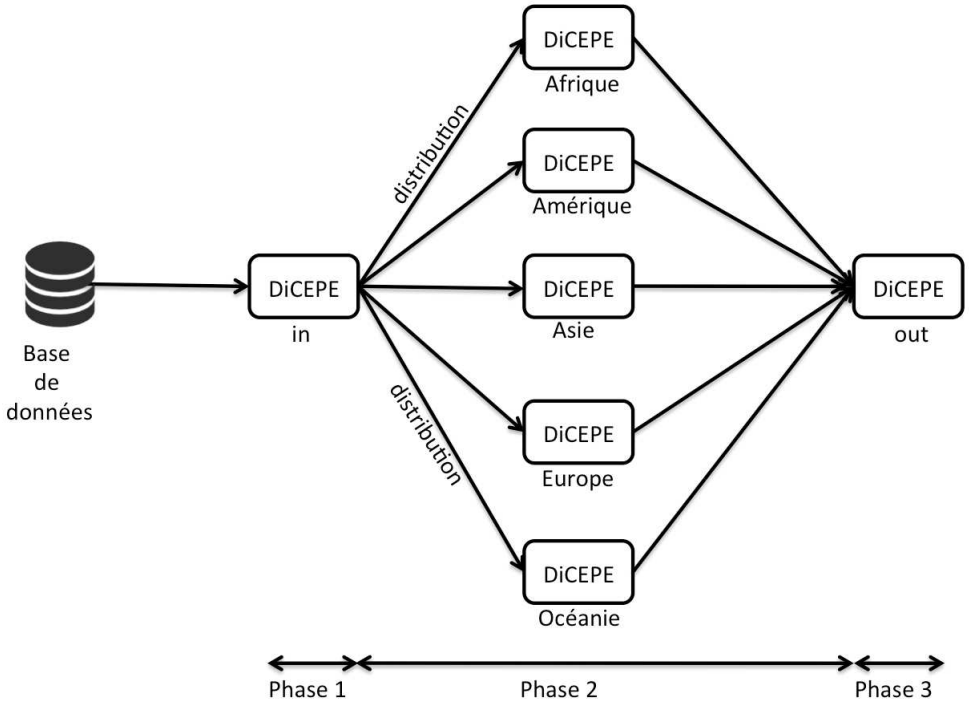

Figure 6.3 – Topologie de déploiement de l'application DiCEPE.

l'année 2010 est constituée de 6 Go d'événements météorologiques recueillis par des capteurs dans le monde entier. Le DiCEPE (in) aggrège et distribue les données par continent (Phase 1 figure 6.3). A la phase 2 figure 6.3, les moteurs DiCEPEs dédiés aux cinq continents analysent respectivement les données reçues pour le continent Africain, Américain, Asiatique, Européen et Océanie, afin de mettre en évidence la température la plus haute et basse par continent. Enfin, la phase 3 de la figure 6.3 montre le DiCEPE (out) qui analyse les données provenant des cinq DiCEPEs représentant chaque continent et affiche la plus haute et la plus basse température dans le monde.

Le listing 6.2 représente la structure de la topologie illustrée à la figure 6.3. Chaque nœud du DiCEPE représenté sur cette topologie est transformé en un composant. Ainsi, les lignes 2-9, 10-14, 15-19, 20-24, 25-29, 30-34 et 35-40 représentent les composants respectifs de *dicepe-input*, *dicepe-af*, *dicepe-am*, *dicepe-as*, *dicepe-eu*, *dicepe-oc* et *dicepe-out*. L'annotation placée en ligne 5 demande le déploiement du composant *dicepe-input* sur une machine virtuelle de type *large* qui dispose d'un système d'exploitation Ubuntu. Les lignes 6-8 représentent l'annotation *closer* qui place le composant *dicepe-input* près des composants *dicepe-af*, *dicepe-am*, *dicepe-as*, *dicepe-eu* et *dicepe-oc* en terme de latence réseau.

```
<composite name="Application-DiCEPE">
  <component name="dicepe-input">
    <implementation.contribution contribution="dicepe-input.zip"/>
    <reference multiplicity="0..n" name="service"/>
    <annotation name="vm">large -> Ubuntu</annotation>
    <annotation name="closer">
        dicepe-af dicepe-am dicepe-as dicepe-eu dicepe-oc
    </annotation>
    <annotation name="database">MySQL</annotation>
  </component>
  <component name="dicepe-af">
    <implementation.contribution contribution="dicepe-af.zip"/>
    <service name="service"/>
    <reference name="compute" target="dicepe-out/cepservice"/>
  </component>
  <component name="dicepe-am">
    <implementation.contribution contribution="dicepe-am.zip"/>
    <service name="service"/>
    <reference name="compute" target="dicepe-out/cepservice"/>
  </component>
  <component name="dicepe-as">
    <implementation.contribution contribution="dicepe-as.zip"/>
    <service name="service"/>
    <reference name="compute" target="dicepe-out/cepservice"/>
  </component>
  <component name="dicepe-eu">
    <implementation.contribution contribution="dicepe-eu.zip"/>
    <service name="service"/>
    <reference name="compute" target="dicepe-out/cepservice"/>
  </component>
  <component name="dicepe-oc">
    <implementation.contribution contribution="dicepe-oc.zip"/>
    <service name="service"/>
    <reference name="compute" target="dicepe-out/cepservice"/>
  </component>
  <component name="dicepe-out">
    <implementation.contribution contribution="dicepe-out.zip"/>
    <service name="cepservice"/>
    <annotation name="vm">large -> Ubuntu</annotation>
  </component>
</composite>
```

Listing 6.2 – Le descripteur de l'application DiCEPE en utilisant le modèle *soCloud*.

Le placement des composants peut se faire de deux façons. Premièrement, si nous supposons que les composants *dicepe-af*, *dicepe-am*, *dicepe-as*, *dicepe-eu* et *dicepe-oc* sont placés dans une même zone géographique, *soCloud* place le composant *dicepe-input* près des autres composants en tenant compte de la latence réseau qui le sépare de ces derniers. Deuxièmement, si les composants *dicepe-af*, *dicepe-am*, *dicepe-as*, *dicepe-eu* et *dicepe-oc* sont placés dans des zones géographiques différentes, *soCloud* place le composant *dicepe-input* en utilisant l'algorithme de K voisins plus proches (K-nearest neighbor) [Kel85]. Dans notre cas, K = 5. Le principe de cet algorithme est de placer le composant *dicepe-input* plus près des autres (voisins) en tenant compte de la latence réseau.

6.1.3 Surveillance de réseau pair-à-pair

Cette section décrit l'application pair-à-pair nommée P2P monitoring répartie qui permet de surveiller et détecter des pannes dans le réseau. Cette application est composée de dix pairs, un pour chaque environnement de nuage. Chaque pair est connecté aux neuf autres pairs pour construire une topologie en maille.

Chaque pair est constitué de trois composants :

1. Le *capteur* qui expose les données de surveillance locale (nom du pair, l'url, l'adresse IP, la date courante, la capacité du processeur, la mémoire libre/utilisée/maximale) comme une ressource REST.
2. L'*agrégateur* qui recueille les données de tous les pairs connexes et calcule la latence du réseau.
3. La *vue* qui produit une page WEB dynamique montrant la géolocalisation des dix pairs, la latence réseau entre les pairs et toutes les données de surveillance collectées comme le montre la figure 6.4.

Pour résumer, cette application de surveillance répartie pair-à-pair est composée de dix composites SCA, chacun constitué de trois composants et utilise des services externes de géolocalisation [31] et de carte Google Maps.

Le listing 6.3 illustre le fichier de description d'un pair déployé en France. Avec l'annotation du modèle *soCloud*, nous avons pu exprimer les différentes localisations des pairs et le nombre de réplications. La ligne 4 indique la localisation du composant, la ligne 5 indique le nombre de réplication.

31. http://freegeoip.net

Figure 6.4 – Réseau pair-à-pair.

```
1<composite name="Application-P2P">
2  <component name="pairfrance">
3    <implementation.contribution contribution="pairfrance.zip"/>
4    <annotation name="location">France</annotation>
5    <annotation name="replication">2</annotation>
6    <reference multiplicity="0..n" name="peer">
7    <service name="Peer"/>
8  </component>
9</composite>
```

Listing 6.3 – Le descripteur de l'application surveillance de réseau pair-à-pair en utilisant le modèle *soCloud*.

6.1.4 Synthèse

Le modèle *soCloud* nous a permis de construire une application de collecte de données pour les smartphones, une application pair-à-pair répartie à large échelle pour surveiller le réseau et enfin une application pour intégrer des moteurs CEP hétérogènes et faire du Big Data. Les trois applications présentées dans cette section sont toutes réparties et ont fait l'objet de déploiement dans un environnement *multi-nuages*. Nous avons montré la mise en œuvre les annotations sur des composants, le langage d'élasticité et l'assemblage des différents composants des applications.

6.2 Evaluation de la plateforme soCloud

Dans cette section, nous évaluons trois aspects de la plateforme *soCloud* : l'*élasticité*, la *haute disponibilité* et le *surcoût* introduit par *soCloud*. Nous évaluons également la performance de l'équilibreur de charge. En effet, l'équilibreur de charge distribue des requêtes et peut introduire des coûts supplémentaires en terme de performance. La section 6.2.1 mesure le temps de déploiement de la platforme *soCloud*. La section 6.2.2 évalue la réaction de la plateforme *soCloud* en présence de l'effet de foule (flash crowd), c'est-à-dire l'*élasticité* de la plateforme *soCloud*. Ensuite, la section 6.2.3 évalue le comportement de la plateforme *soCloud* lorsqu'il y des pannes (c'est-à-dire la *haute disponibilité* de la plateforme *soCloud*). La section 6.2.4 évalue le surcoût introduit par la plateforme *soCloud*. Enfin, la section 6.2.5 évalue la performance de l'équilibreur de charge.

6.2.1 Mesure du temps de déploiement

6.2.1.1 Temps de déploiement de la plateforme *soCloud*

Comme décrit dans le chapitre 5, le déploiement de *soCloud* consiste au déploiement du master *soCloud* et de plusieurs agents *soCloud*.

6.2.1.1.1 *soCloud* master Le déploiement du *soCloud* master se fait en déployant à la fois le leader et le follower sur deux nuages différents pour assurer la haute disponibilité. Le déploiement sur chaque nuage consiste à déployer l'environnement d'exécution (FraSCAti) avec les masters *soCloud*. Le déploiement des masters *soCloud* prend environ $2,1$ minutes.

6.2.1.1.2 *soCloud* agent Nous mesurons le temps pour le déploiement d'un *soCloud* agent. Le déploiement consiste à déployer l'environnement d'exécution (FraSCAti) avec le *soCloud* agent. Le déploiement d'un *soCloud* agent prend environ *0,9* minute.

D'une manière générale, la durée moyenne de déploiement du *soCloud* avec deux masters (le leader et le follower) et un agent est d'environ **3 minutes**.

6.2.1.2 Mesure du temps de déploiment d'une application *soCloud*

Dans le but de mesurer le temps de déploiement d'une application *soCloud*, nous avons déployé notre application de cas d'utilisation directement sur CloudBees et à travers la plateforme *soCloud* (voir figure 6.5). Nous avons conditionné deux différentes archives. La première archive est un fichier WAR de $50,7$ MB. Ce fichier contient l'application et l'environnement d'exécution FraSCAti. La deuxième archive est un fichier Zip (contributiuon SCA) de $2,1$ MB. Le deuxième fichier contient seulement l'application. Les fichiers WAR et Zip sont déployés respectivement sur CloudBees et *soCloud*. Le déploiement des fichiers WAR et

Zip a été effectué dix fois. Le tableau 6.1 présente le temps moyen de déploiement de chaque fichier.

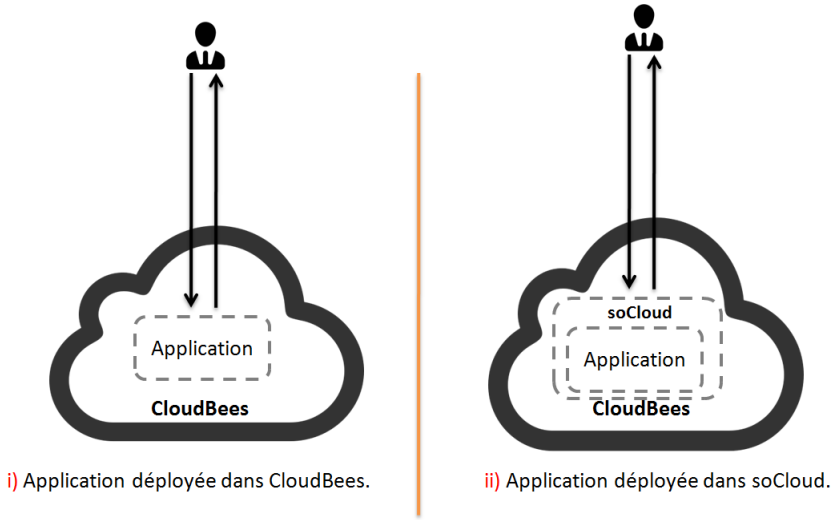

Figure 6.5 – Déploiement de l'application avec et sans *soCloud*.

Tableau 6.1 – Temps de déploiement des fichiers Zip et WAR

Implémentation	Taille du fichier	Temps moyen de déploiement	Vitesse
Zip File (Application)	2, 1 Mb	5301, 5 ms	1, 00
WAR File (Application + FraSCAti)	50, 7 Mb	80830, 8 ms	15, 24

Comme illustré dans le tableau 6.1, le temps de déploiement de l'application directement sur CloudBees est 15 fois plus lent que le temps de déploiement de l'application via *soCloud*. Ceci s'explique par la taille du fichier WAR qui est 25 fois plus grand que le fichier Zip. En effet, transférer un petit fichier dans le réseau est plus rapide qu'un gros fichier. Lorsqu'on a déployé le fichier Zip sur la plateforme *soCloud*, l'environnement d'exécution FraSCAti est déjà installé et démarré. Ce n'est pas le cas du fichier WAR qui contient FraSCAti, l'environnement d'exécution qui va être installé et démarré sur la plateforme CloudBees avant de déployer l'application dessus.

6.2.2 Mesure du temps de réaction de *soCloud* face à l'effet de foule

Nous avons implémenté et déployé avec la plateforme *soCloud* un prototype de l'application trois-tiers décrite dans la section 4.2 page 69. Notre plateforme *soCloud* est déployée chez dix fournisseurs de nuages différents. Nous avons procédé à une expérimentation de

cette application afin d'évaluer le comportement et l'efficacité de la plateforme *soCloud* face à l'effet de foule [32]. Nous avons effectué deux analyses : (a) l'application est déployée sans le mécanisme d'élasticité de la plateforme *soCloud* et (b) l'application est déployée avec le mécanisme d'élasticité de la plateforme *soCloud*.

6.2.2.1 Mesure du temps de réaction face à l'effet de foule sans l'élasticité de *soCloud*

Dans le premier cas, nous avons observé le comportement de cette application sans le mécanisme d'élasticité sous une charge accrue de requêtes. Chaque requête déclenche une opération qui consiste en l'analyse des métriques collectées et le stockage des résultats dans une base de données. Nous avons configuré l'outil httperf [Mos98] pour créer 50000 connexions avec 10 requêtes par seconde, le nombre de connexions créé par seconde varient entre 10 et 150 ce qui correspond à un total de 3020000 requêtes.

La figure 6.6(a) montre le nombre de requêtes traitées par l'application avec deux phases de l'effet de foule. La figure 6.6(b), quant à elle, représente le temps de réponse correspondant (calculé comme le nombre d'opérations effectuées). Au cours des deux phases de l'effet de foule, le temps de réponse moyen est de $65,90$ secondes. Les figures 6.6(a) et 6.6(b) montrent une charge soudaine provoquée par l'effet de foule. Nous avons remarqué que le nombre de requêtes augmente avec le temps de réponse. Ensuite, la figure 6.6(c) montre les requêtes qui ont échoué successivement durant les deux phases de l'effet de foule. Ainsi, lors de l'effet de foule, $1,13\%$ des requêtes ont échoué, plus précisément 34039 requêtes ont échoué. Ces échecs de requêtes sont dus au temps de réponse de 5 secondes maximal (time out) que nous avons configuré dans l'outil httperf pour chaque requête traitée. En fait, cet intervalle de temps signifie que l'absence de réponse du serveur sur la connexion TCP pour cette durée est considérée comme une erreur.

D'une manière générale, lorsque l'application devient saturée, elle souffre de défauts d'exécution et peut provoquer des retards de réponses longues. Nous observons également que l'application ne peut soutenir un taux de requêtes que jusqu'à une certaine limite qui dépend directement du nombre de requêtes sur un intervalle de temps.

6.2.2.2 Mesure du temps de réaction face à l'effet de foule avec l'élasticité de *soCloud*

Dans le second cas, nous avons étudié l'évolution du temps de réponse lors des deux phases de l'effet de foule lorsque l'élasticité de la plateforme soCloud est utilisée. Nous supposons que les ressources (VM) sont pré-allouées et qu'un agent *soCloud* est déployé dans ce dernier. La figure 6.6 montre les résultats de la même expérience que précédemment mais avec l'utilisation du mécanisme d'élasticité de la plateforme *soCloud*. Nous observons d'abord une certaine réaction au niveau de l'application lorsque le temps de réponse diminue.

32. L'effet de foule désigne le fait qu'une application soit submergée de requêtes provenant d'utilisateurs, le rendant ainsi momentanément indisponible.

Pendant la première phase de l'effet de foule, le temps de réponse moyen est de $37,30$ secondes. En effet la plateforme *soCloud* a détecté la montée de pic en 300 millisecondes. Après 4 secondes, la plateforme *soCloud* a répliqué l'application dans un autre agent *soCloud* qui se trouve dans un autre nuage en mettant à jour la table de routage de l'équilibreur de charge pour répartir la charge entre les différentes instances de l'application. Cette réaction apparaît clairement sur la figure 6.6(d). L'équilibreur de charge de la plateforme *soCloud* distribue les requêtes entre les deux instances de l'application et le temps de réponse reste en général faible, malgré un trafic élevé.

Au cours de la deuxième phase de l'effet de foule, l'application a déjà été déployée, la plateforme soCloud a détecté la montée du pic en *300* millisecondes. Comme le montre la figure 6.6(d), nous ne remarquons pas la montée du pic au cours de cette deuxième phase de l'effet de foule et le temps de réponse moyen est de *23,38* secondes. Le temps de réponse relativement faible lors de la deuxième phase de l'effet de foule est du au fait que la plateforme *soCloud* a déjà répliqué l'application.

D'une manière générale, lors la montée en charge de l'effet de foule, toutes les requêtes ont été traitées avec succès et le temps de réponse est relativement acceptable. La plateforme *soCloud* permet le passage à l'échelle de l'application tout en garantissant un temps de réponse acceptable pour les requêtes qui ont été traitées. Ainsi, ces résultats démontrent que la plateforme soCloud assure bien l'élasticité des applications à travers plusieurs nuages, car les applications ont été répliquées dans différents nuages.

6.2.3 Comportement de *soCloud* face aux pannes

Considérons notre application fil rouge de la section 4.2 sur laquelle nous effectuons toutes nos évaluations. *soCloud* est déployée comme décrit sur la figure 5.17. Les masters *soCloud* leader et follower sont déployés respectivement sur dotCloud et CloudBees. Les agents *soCloud* sont déployés sur Amazon EC2, Windows Azure, DELL KACE, OpenShift, Jelastic, Heroku, Appfog et notre nuage privé Eucalyptus.

6.2.3.1 Echec et récupération du master *soCloud* leader

Nous supposons que la plateforme *soCloud* fonctionne et que notre application trois-tiers est déployée sur cette dernière. Pour simuler une panne nous arrêtons le master *soCloud* leader qui se trouve dans dotCloud. Dans nos observations, *soCloud* prend en moyenne $3,5$ minutes pour la reprise et redevient opérationnel. *soCloud* prend moins de 200 millisecondes pour élire un nouveau leader. Le processus de reprise se fait comme suit : l'agent *soCloud* détecte automatiquement le nouveau *soCloud* master leader. Selon les auteurs [Mau12, Uri13], le temps moyen de récupération (Mean Time To Recovery)(MTTR) pour les nuages publics est de $7,5$ **heures**. Le MTTR des nuages publics a été observé par le groupe International IWGCR [Mau12] sur la résilience de l'informatique en nuage qui a collecté les différentes informations sur les pannes survenues pendant des années auprès de chaque fournisseur de

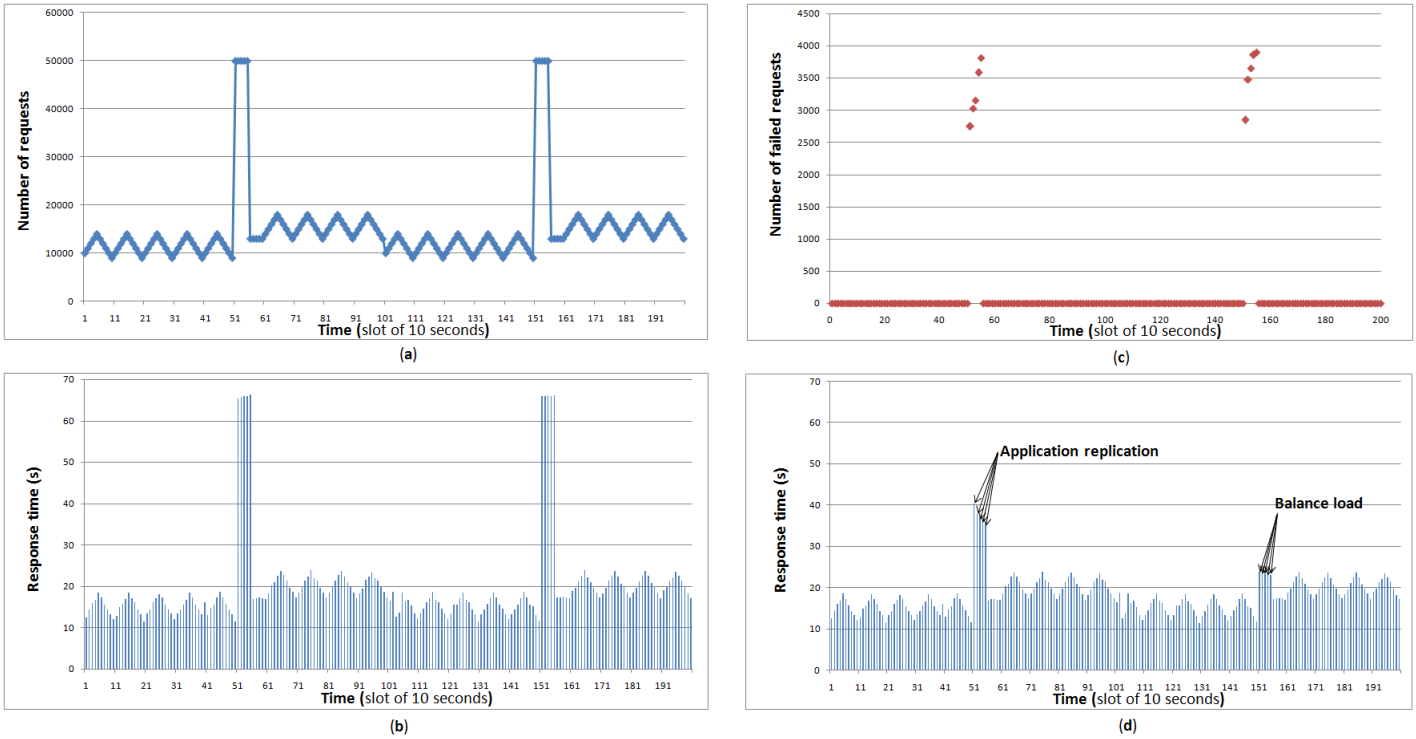

Figure 6.6 – Une série de deux effets de foule. (a) le nombre de requêtes au cours de l'exécution du scénario. (b) le temps de réponse aux clients durant l'effet de foule sans l'élasticité de la plateforme *soCloud*. (c) le nombre de requêtes ayant échouée au cours des deux phases de l'effet de foule. (d) le temps de réponse aux clients durant l'effet de foule avec l'élasticité de la plateforme *soCloud*.

nuage public. En comparaison, le temps moyen de récupération de *soCloud* est seulement de $3,5$ **minutes** comme le montre le tableau 6.2.

Tableau 6.2 – Résultats du MTTR

	MTTR(Heure)	**MTTR(Minute)**	**Ratio**
soCloud	0,06 heure	3,6 minutes	-
Nuages publics	7,5 heures	450 minutes	125

6.2.3.2 Echec et récupération du *soCloud* master follower

Dans ce cas, nous simulons la défaillance du master *soCloud* follower déployé sur cloudBees, le *soCloud* master leader détecte automatiquement la panne. Le *soCloud* master prend environ 1200 millisecondes pour élire et démarrer le nouveau master *soCloud* follower.

6.2.3.3 Temps d'arrêt d'applications déployées sur *soCloud*

La défaillance d'une application déployée sur *soCloud* n'affecte pas la disponibilité de celle-ci. En effet, en cas de panne, l'équilibreur de charge prend environ *300* millisecondes pour détecter et rediriger les requêtes automatiquement vers une autre instance de l'application déployée.

6.2.3.4 Temps d'arrêt de l'agent *soCloud*

La défaillance d'un seul agent *soCloud* n'affecte pas la disponibilité de l'application déployée sur la plateforme *soCloud*. L'équilibreur de charge de *soCloud* permet de rediriger les requêtes vers une autre instance de l'application. Le déploiement et démarrage de l'agent *soCloud* prend environ *0,9* minute. Cependant, le temps de déploiement des applications sur la plateforme dépend essentiellement de la latence réseau et la taille de l'application.

6.2.3.5 La haute disponibilité de la plateforme *soCloud*

Considérons l'équation de la disponibilité définie par [Mar03, Tor04] ci-dessous :

$$Availability = \frac{MTBF}{MTBF + MTTR} \tag{6.1}$$

Tableau 6.3 – Détails de l'équation

Variable	Description
MTBF	Mean Time Between Failure
MTTR	Mean Time To Recover

Comme l'équation 6.1 le montre, plus le MTTR augmente, moins il y a de disponibilité. La formule illustre comment le MTTR et le MTBF impactent la disponibilité globale du système. Afin de comparer la disponibilité de *soCloud* et celle des nuages publics, nous devons utiliser le même MTBF. Ainsi, sur une période d'un an, le MTBF est de *8760* heures. Le calcul de la disponibilité est effectué dans le tableau 6.4.

Tableau 6.4 – Comparaison de la disponibilité

	Disponibilité
soCloud	$\frac{8760}{8760+0,06} = 99,999\%$
Nuages publics	$\frac{8760}{8760+7,5} = 99,914\%$

En général, comme le montre le tableau 6.4, la disponibilité des nuages publics est de 99,914%. A titre de comparaison, la disponibilité de *soCloud* est de 99,999%. Cette disponibilité est proche de la fiabilité attendue des systèmes critiques [Mau12]. La plateforme *soCloud* améliore la disponibilité. Ce résultat démontre que *soCloud* assure bien la haute disponibilité à travers plusieurs nuages.

6.2.4 Le surcoût introduit par *soCloud*

Pour évaluer le surcoût introduit par la plateforme *soCloud*, 10000 requêtes ont été générées et envoyées par l'outil Httperf. Nous avons évalué deux cas de figures : i) l'application déployée sans *soCloud* et ii) l'application déployée avec *soCloud*. Les requêtes envoyées ont été exécutées dix fois dans les deux scénarios. Le tableau 6.5 présente les résultats du temps d'exécution moyen de chaque scénario, ainsi que le surcoût introduit par la plateforme *soCloud*.

Tableau 6.5 – Temps d'exécution et surcoût

Implémentation	**Temps moyen d'exécution**	**Surcoût introduit par *soCloud***
(Application + FraSCAti)	$10,85$ sec	-
(Application + FraSCAti + soCloud)	$11,10$ sec	$2,3\%$

A partir des résultats présentés dans le tableau 6.5, on peut remarquer que le surcoût introduit par la plateforme *soCloud* est seulement de $2,3\%$. Ce surcoût est généré par les composants de *surveillance* et l'*équilibreur de charge*. Le surcoût introduit par le composant de *surveillance* est du aux informations de surveillance collectées pour le mécanisme d'élasticité. Afin de garantir la *haute disponibilité*, le composant *équilibreur de charge* introduit un surcoût qui sera analysé plus en détail dans la section suivante.

En résumé, le surcoût introduit par la plateforme *soCloud* est négligeable vu les avantages que procurent le mécanisme d'élasticité et la haute disponibilité.

6.2.5 L'équilibreur de charge de *soCloud*

Dans cette section, les expériences sont focalisées sur la performance de l'équilibreur de charge.

6.2.5.1 Le coût introduit par l'équilibreur de charge

Considérons notre application trois-tiers qui traite les requêtes et persiste les résultats dans une base de données. Pour évaluer notre équilibreur de charge, nous nous focalisons sur la surcharge introduite par ce dernier. Pour analyser le surcoût introduit par l'équilibreur de charge avec le décalage d'Internet [33] (Internet Lag), toutes les expériences ont été réalisées avec deux fournisseurs de nuages différents. L'équilibreur de charge et l'application ont été déployés respectivement sur Windows Azure et DELL KACE. Pour évaluer la surcharge introduite par l'instance de l'équilibreur de charge, 100000 requêtes ont été envoyées à l'application. Nous avons évalué deux cas (voir figure 6.7) : i) accès direct à l'application et ii) accès

33. C'est une latence réseau perceptible entre l'émetteur et le récepteur.

à l'application en passant par l'équilibreur de charge. Ces tests ont été exécutés 10 fois dans chacun des deux cas. Dans le tableau 6.6, nous présentons les résultats du temps d'exécution moyen de chaque cas, ainsi que la moyenne de la surcharge introduite par l'équilibreur de charge.

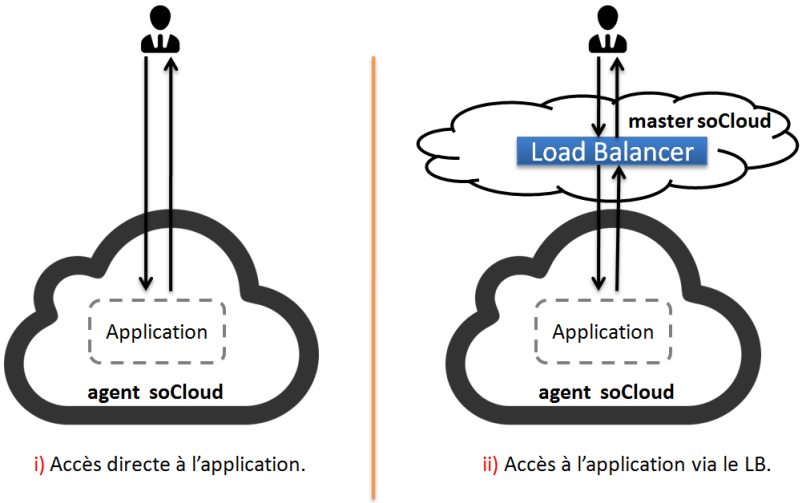

Figure 6.7 – Performance de l'équilibreur de charge.

Tableau 6.6 – Temps d'exécution et la surcharge introduite par l'équilibreur de charge.

Implémentation	Temps moyen d'exécution	Surcoût du LB
APP	16, 16 sec	-
APP + LB	16, 39 sec	1, 48%

En général, ces expériences montrent qu'il y a une petite surcharge introduite lorsqu'on ajoute l'équilibreur de charge. Toutefois, le temps d'exécution reste acceptable vu les avantages qu'offre l'équilibreur de charge à l'éxécution.

6.2.5.2 La performance de l'équilibreur de charge

Afin d'évaluer la performance de l'équilibreur de charge, nous utilisons un outil de test de performance appelé inject32 [34]. Grâce à cet outil, nous générons 134021 requêtes que nous envoyons de manière continue à une instance de l'équilibreur de charge pour une durée de 156000 millisecondes. Des données statistiques sont collectées dans un intervalle d'une seconde. Nous avons effectué 156 mesures. La bande passante est mesurée au niveau de HTTP et ne tient pas compte des acquittements TCP ni des en-têtes. Cet outil mesure la performance de l'équilibreur de charge avec trois facteurs importants :

34. http ://1wt.eu/tools/inject/

1. Le taux de session : ce facteur détermine directement quand l'équilibreur de charge ne serait plus en mesure de distribuer toutes les requêtes qu'il reçoit. C'est un facteur qui dépend la plupart du temps du CPU.
2. La concurrence de session : généralement, le taux de session diminue lorsque le nombre de sessions simultanées augmente. Plus lent est le serveur, plus grand est le nombre de sessions simultanées pour le même taux de session.
3. La vitesse des données : ce facteur est généralement l'opposé du taux de session. Elle est mesurée en Megabytes/s (MB/s) ou parfois en Megabits/s (Mbps).

La machine sur laquelle les tests de performance de l'équilibreur de charge sont réalisés a les caractéristiques suivantes : un processeur de 2,6 GH Intel (R) Xeon, une mémoire de 4GB, une carte réseau de 100 Mbits/s, un serveur Ubuntu 3.0.0-12 64 bit et un environnemnt d'exécution Oracle Java 1.6.

Tableau 6.7 – Résultats de la performance.

Taux de session	Concurrence	Taux de données	Échecs	Temps moyen
850	283	4560 kB/s	0	3 ms

Pour un total de 850 sessions, on observe 283 concurrences, aucune défaillance et le temps de réponse moyen est de *3* millisecondes. En général, compte tenu des faibles ressources utilisées par l'équilibreur de charge, les résultats obtenus dans le tableau 6.7 sont satisfaisants.

L'équilibreur de charge de *soCloud* est de type logiciel niveau 7[35](Couche du modèle OSI). On peut atteindre de meilleures performances en traitant les paquets au niveau réseau.

Une des questions les plus courantes, lorsqu'on compare un équilibreur de charge logiciel et matériel est pourquoi existe-t-il un tel écart entre le nombre de session de ces deux types d'équilibreurs de charge ? En fait, cela dépend si l'équilibreur de charge doit gérer ou non la pile TCP/IP. La pile TCP/IP nécessite qu'une fois la session terminée, elle reste dans le tableau *TIME/WAIT state* assez longtemps pour attraper les paquets en retard retransmis qui peuvent survenir plusieurs minutes plus tard après que la session soit fermée. Passé ce délai, la session est complètement supprimée [Tar06]. Les sessions qui sont dans cet état ne comportent pas de données et ne sont pas vraiment coûteuses (en terme de ressources consommées). Comme elles sont traitées de manière transparente par le système d'exploitation, l'équilibreur de charge ne les voit jamais et annonce seulement le nombre de sessions actives qu'il supporte. Mais lorsque l'équilibreur de charge gère le protocole TCP, il prend en charge de très grandes tables de session pour conserver ces dernières.

35. L'équilibreur de charge de niveau-7 implique la persistance de cookies, la commutation de l'URL et des fonctionnalités utiles (disponibilité des applications, le passage à l'échelle).

6.3 Conclusion

Pour évaluer le modèle d'applications *soCloud*, nous avons mis en œuvre trois applications réparties (*Plateforme APISENSE*, *Surveillance de réseau pair-à-pair*, *Plateforme pour fédérer plusieurs moteurs de CEP*). Ces applications ont été déployées dans un environnement *multi-nuages*.

L'évaluation expérimentale de la plateforme *soCloud* s'est organisée selon trois axes :

— *Mesure du temps de réaction de* soCloud *face à l'effet de foule*. Nous avons analysé le phénomène de l'effet de foule sur un cas d'utilisation et nous avons démontré comment la plateforme *soCloud* assure l'élasticité des applications dans un environnement *multi-nuages*.

— *Comportement de* soCloud *face aux pannes*. Les résultats observés ont montré que la plateforme *soCloud* assure la haute disponibilité en minutes, tandis que les plateformes de nuages publics assurent la disponibilité en heures en cas de pannes.

— *Le surcoût introduit par la plateforme* soCloud. Comme toute solution d'abstraction, la plateforme est susceptible d'introduire un surcoût supplémentaire. Les résultats obtenus nous permettent de mettre en évidence le faible coût que la plateforme *soCloud* a introduit. Comparativement aux bénéfices que *soCloud* apporte, le surcoût introduit est négligeable.

Quatrième partie

BILAN ET PERSPECTIVES

Chapitre 7

Bilan et perspectives

"Ce que tu fais de valeureux aujourd'hui inspire les actions des autres dans le futur."-Marcus Garvey

Sommaire

Dans ce dernier chapitre, nous faisons le bilan de nos contributions (voir section 7.1) que nous replaçons dans le contexte général de notre thèse. Dans la section 7.2, nous discutons des contraintes de nos contributions. La section 7.3 discute des perspectives pour dépasser les limites des solutions proposées dans cette thèse.

7.1 Bilan des contributions

Dans cette thèse, nous avons voulu démontrer que le modèle d'applications *soCloud* est une approche simple et efficace permettant la conception d'applications orientées services à large échelle pour un environnement *multi-nuages*. Nous avons expliqué comment ce modèle nous permet d'exprimer les contraintes de la *portabilité*, l'*approvisionnement*, l'*élasticité* et la *haute disponibilité* qui sont les quatres défis que nous avons identifiés dans notre travail de thèse. Nous avons aussi mis en œuvre la plateforme *soCloud*, comme support pour déployer, exécuter et gérer les applications *multi-nuages*. *soCloud* est une plateforme à base de composants et services qui assure le déploiement, l'exécution et la gestion des applications orientées services.

7.1.1 Contribution au modèle d'applications *multi-nuages*

Dans notre première contribution, nous avons proposé une approche basée sur un modèle pour la conception d'applications orientées services à large échelle dans un environnement *multi-nuages*. Notre modèle est une extension du modèle SCA qui est un standard du consortium OASIS. Le modèle que nous proposons implique l'utilisation systématique du modèle SCA. SCA décrit une configuration canonique des composants applicatifs basés sur le style SOA. Notre modèle simplifie la description des architectures applicatives indépendemment des plateformes techniques, des langages de programmation, des protocoles d'appel des services. Il est axé non seulement sur le métier, mais aussi sur le non-fonctionnel, qui sont formalisés. Cette formalisation nous permet de concevoir de manière complète et spécifique l'architecture des applications *multi-nuages*. Ce modèle insiste sur une séparation forte entre l'implémentation des services applicatifs, l'assemblage des services fonctionnels et la description des besoins non-fonctionnels. Ces besoins non-fonctionnels sont décrits en utilisant des annotations qui peuvent être exprimées à la granularité des composants. Notre modèle permet d'améliorer la lisibilité de l'architecture de l'application. Chaque composant constitue une unité de déploiement et les annotations aident à exprimer des besoins non-fonctionnels tissés automatiquement au composant sur lequel elles sont placées. Cette séparation des préoccupations facilite la maintenance et l'évolution des architectures par ingénierie spécifique, puisque le modèle est conceptuellement découplé. Le modèle *soCloud* fournit deux niveaux de composition pour les applications. Le premier niveau permet de construire la logique métier de l'application et de l'assembler sous forme de fichiers binaires. Le deuxième niveau encapsule le premier et permet de décrire l'architecture de l'application en définissant des propriétés non-fonctionnelles. Nous avons identifié quatre contraintes non-fonctionnelles spécifiques aux applications en nuages.

Nous avons introduit un nouveau langage pour exprimer consicément l'élasticité des applications *multi-nuages*. Ce langage nous permet de contrôler l'élasticité des applications dans un environnement *multi-nuages*. Il nous permet de contrôler l'élasticité non seulement en terme de ressources mais aussi en terme de coût et qualité de service. C'est un langage de haut niveau permettant au développeur d'exprimer de manière intuitive le comportement élastique de chaque composant de son application. La capacité qu'offre le langage d'élasticité aux développeurs, à savoir accéder aux informations de l'environnement, permet à ces derniers d'être au courant des capacités des infrastructures sous-jacentes et de l'état de l'application. Notre langage est plus expressif que certains identifiés dans la littérature (par exemple, le langage d'élasticité de Cloudify). De plus, notre langage peut être facilement étendu pour prendre en compte des exigences spécifiques en terme d'élasticité.

7.1.2 Contribution à l'architecture d'une plateforme *multi-nuages*

Dans notre deuxième contribution, nous avons mis en œuvre la plateforme *soCloud* pour déployer, exécuter et gérer les applications *multi-nuages*. La plateforme *soCloud* est un système distribué orienté services hétérogène, multiples langages et multiples protocoles. Elle

est constituée de deux parties (l'*agent* et le *master*) qui communiquent ensemble. L'agent *soCloud* permet d'héberger, de surveiller et d'exécuter les applications en tant que service. Le master *soCloud* gère le déploiement, l'approvisionnement de ressources, la coordination, la disponibilité et surveille le changement des applications déployées. La plateforme *soCloud* nous permet non seulement de déployer des applications orientées services mais aussi d'autres types d'applications. La plateforme *soCloud* nous permet de réduire les compétences nécessaires pour déployer, exécuter et gérer des applications. *soCloud* est une plateforme *multi-nuages* qui a été déployée sur dix nuages. Comparé à d'autres solutions telles que moSAIC, Aneka, CompatibleOne, STRATOS, *soCloud* supporte plus de nuages. Par ailleurs, *soCloud* supporte non seulement le mode de communication synchrone, mais aussi l'asynchrone, ce qui n'est pas de cas de mOSAIC par exemple. La plateforme *soCloud* peut être déployée non seulement sur une infrastructure en tant que service, mais aussi sur une plateforme en tant que service. L'intégration avec les PaaS existants offre la possibilité d'utiliser des fonctionalités et ressources de ces dernières. *soCloud* assure le déploiement automatique et améliore la portabilité des applications à travers plusieurs nuages. Elle est capable de fournir la haute disponibilité et l'élasticité aux applications qui sont déployées dessus. Nous utilisons l'approche *Wait-free* pour le problème de la coordination entre les différents composants de *soCloud* et un équilibreur de charge pour basculer d'une instance d'application vers une autre. Nous stockons l'historique des états de fonctionnement pour assurer la cohérence dans le système lors d'une panne. En cas de panne, il existe un mécanisme de restauration qui est mis en œuvre.

7.2 Contraintes sur les contributions de la thèse

soCloud ne tient pas compte de toutes les fonctionnalités fournies par les plateformes de nuages sous-jacentes. Plus précisement, *soCloud* n'exploite pas les caractéristiques spécifiques (c'est-à-dire règles d'élasticité, contrainte de placement, contrainte d'exécution, contrainte de fiabilité) qui sont fournies par les plateformes sous-jacentes.

Dans ce travail de thèse, nous nous sommes focalisés essentiellement sur la portabilité des applications. Les utilisateurs/entreprises doivent pouvoir aussi déplacer de manière transparente leurs données, les intégrer et les interconnecter ensemble au sein de systèmes disparates. La portabilité des données est aussi importante que celle des applications.

Les plateformes de nuages en tant que service évoluent de manière dynamique. *soCloud* comme toute solution *multi-nuages* se voit confrontée au problème de la maintenance, de sa projection vers les différents fournisseurs de nuages sous-jacents. Il en est de même pour la gestion de la mise à jour de cette évolution. Une manière commune d'aborder ces problèmes est d'envelopper (Wrapping) les caractéristiques des plateformes sous-jacentes comme celle de *soCloud*. Toutefois, l'utilisation de standard reste la meilleure approche à ces problèmes.

La plateforme *soCloud* fournit une couche d'abstraction pour cacher l'hétérogénité et la complexité des plateformes sous-jacentes. Cette solution fournie par *soCloud* peut introduire

un coût additionnel (en terme de performance, d'empreinte) aux infrastructures en tant que service existantes. Cependant, *soCloud* fournit une manière uniforme d'approvisionner des ressources, de déployer, d'exécuter et de gérer les applications dans un environnement *multi-nuages*. L'utilisation de la plateforme *soCloud* permet au développeur de se concentrer sur un seul nuage fédérateur plutôt que plusieurs implémentations hétérogènes de celui-ci. *soCloud* permet aux développeurs de bénéficier de la portabilité *multi-nuages* avec une gestion efficace de ces applications à travers plusieurs nuages.

Après avoir porté un regard critique sur les sujets que nous avons évoqués dans ce manuscrit, nous présentons les extensions et perspectives que nous envisageons pour la suite de nos travaux.

7.3 Extensions envisageables et perspectives

Nous avons montré la nécessité et les bénéfices d'utiliser un modèle pour concevoir une application pour un environnement *multi-nuages* et la plateforme *soCloud* qui sert de support pour les applications déployées. Les perspectives de cette thèse sont nombreuses car les APIs fournies par les plateformes de nuages sous-jacentes ne cessent d'évoluer et nécessitent de gérer des problèmes de plus en plus complexes tout en restant transparent pour l'utilisateur.

D'une part, il serait intéressant d'étudier plus en détail les interactions entre les entités qui peuvent évoluer rapidement dans le temps. Lors de la conception de notre platforme, nous n'avons pris en compte que les caractéristiques fournies par les plateformes que nous avons considérées comme tangibles. Il serait intéressant de prendre en considération toutes les caractéristiques fournies par les plateformes sous-jacentes afin d'offrir aux utilisateurs la possibilité d'utiliser toutes ces caractéristiques mais aussi celles fournies par la plateforme *soCloud*.

Une autre perspective de recherche reposera sur la gestion de la disponibilité malgré les défauts (bugs) logiciels. Dans notre approche, la gestion de la disponibilité se fait sans prendre en compte les défauts logiciels qui sont aussi l'une des principales causes de panne dans le domaine de l'informatique. Les défauts logiciels ne sont pas assez bien maitrisés pour fournir une méthodologie claire et simple pour éviter ou assurer la reprise après une panne. Toutefois, des travaux sont menés dans l'équipe SPIRALS autour de l'auto réparation des défauts logiciels.

Une autre perspective est le mécanisme d'élasticité par apprentissage. L'apprentissage par renforcement est un type d'approche automatique de prise de décision qui utilise l'approche proactive. L'apprentissage par renforcement est une approche de compréhension et d'automatisation dirigée par l'apprentissage et la prise de décision. Avec les différentes métriques collectées et changements qui ont lieu dans le système, nous pouvons décrire la modélisation de l'apprentissage par renforcement comme un processus de Markov en utilisant l'algorithme Q-Learning [Wat92] qui est l'une des méthodes existantes.

Une autre perspective est le partage des états entre réplicats. Le partage de l'état permettra à chaque réplicat d'être autonome. Une approche consiste à utiliser le mécanisme State Machine Replication (SMR) [Sch12]. Dans le SMR, les réplicats tombent d'accord sur une séquence de commandes d'états et les appliquent dans l'ordre de la séquence de la machine d'état (réplicat) locale. Si les séquences de commandes sont les mêmes, chaque machine d'état arrivera au même résultat.

En outre, assurer la sécurité dans le nuage est un facteur important. Les utilisateurs stockent souvent des informations sensibles chez les fournisseurs de nuages qui peuvent être non fiables. L'émergence du paradigme *multi-nuages* impose de revoir la solution de la sécurité sous un autre angle. Nous pouvons envisager la mise en œuvre d'un protocole de cryptographie pour contrôler plusieurs nuages. Ce protocole permettra à un ensemble de nuages d'assurer que les données des utilisateurs finaux sont récupérables de manière intégrale. Il permettra aussi l'intégrité et la confidentialité des données dans un environnement *multi-nuages*.

Bibliographie

[Adl10] ADLER BRIAN : Load balancing in the cloud : Tools, tips et techniques. *white paper*, pages 1–18, 2010. 118

[Ama12a] AMAZON : Amazon web services. http://aws.amazon.com, Juin 2012. 62

[Ama12b] AMAZON : Summary of Amazon ELB Service outage in the US-East Region. http://tinyurl.com/bjnxn7w, Décembre 2012. 28

[Ame10] AMEDRO BRIAN, BAUDE FRANÇOISE, CAROMEL DENIS, DELBÉ CHRISTIAN, FILALI IMEN, HUET FABRICE, MATHIAS ELTON ET SMIRNOV OLEG : An efficient framework for running applications on clusters, grids et clouds. *In Cloud Computing*, pages 163–178. Springer, 2010. 54

[Ami92] AMIR YAIR, DOLEV DANNY, KRAMER SHLOMO ET MALKI DALIA : Membership Algorithms for Multicast Communication Groups. *In Distributed Algorithms*, pages 292–312. Springer, 1992. 58

[Ana07] ANANE RACHID : Autonomic behaviour in QoS management. *In the third International Conference on Autonomic and Autonomous Systems, ICAS07.*, pages 57–57. IEEE, 2007. 17

[Ane09] ANEDDA PAOLO, GAGGERO MASSIMO ET MANCA SIMONE : A general service oriented approach for managing virtual machines allocation. *In Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 2154–2161. ACM, 2009. 3

[Ane10] ANEDDA PAOLO, LEO SIMONE, MANCA SIMONE, GAGGERO MASSIMO ET ZANETTI GIANLUIGI : Suspending, Migrating and Resuming HPC virtual clusters. *Future Generation Computer Systems*, 26(8):1063–1072, 2010. 55

[Apaa] APACHE FOUNDATION : DeltaCloud API that abstracts difference between clouds. http://deltacloud.apache.org/. 56

[Apab] APACHE FOUNDATION : The jclouds API that uses cloud-specific features. http://jclouds.incubator.apache.org/. 56, 122

[Apa13] APACHE FOUNDATION : Open source cloud computing. https://www.openstack.org/, 2013. 12

[App03] APPAVOO JONATHAN, HUI KEVIN, SOULES CRAIG AN, WISNIEWSKI ROBERT W, DA SILVA DILMA M, KRIEGER ORRAN, AUSLANDER MARC A, EDELSOHN DJ, GAMSA BENJAMIN ET GANGER GREGORY R : Enabling autonomic behavior in systems software with hot swapping. *IBM systems journal*, 42(1):60–76, 2003. 17

[Ard12] ARDAGNA DANILO, DI NITTO ELISABETTA, MOHAGHEGHI P, MOSSER S, BALLAGNY C, D'ANDRIA F, CASALE G, MATTHEWS P, NECHIFOR C-S ET PETCU D : Modaclouds : A model-driven approach for the design and execution of applications on multiple clouds. *In ICSE Workshop on Modeling in Software Engineering (MISE)*, pages 50–56. IEEE, 2012. 54, 58, 59

[Arm10] ARMBRUST M., FOX A., GRIFFITH R., JOSEPH A.D., KATZ R., KONWINSKI A., LEE G., PATTERSON D., RABKIN A., STOICA I. : A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. 27, 33

[Bab11] BABAR MUHAMMAD ALI ET CHAUHAN MUHAMMAD AUFEEF : A tale of migration to cloud computing for sharing experiences and observations. *In Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, pages 50–56. ACM, 2011. 38

[Bar03] BARHAM PAUL, DRAGOVIC BORIS, FRASER KEIR, HAND STEVEN, HARRIS TIM, HO ALEX, NEUGEBAUER ROLF, PRATT IAN ET WARFIELD ANDREW : Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003. 2, 57

[Bar12] BARB DARROW : Latest outage raises more question about Amazon cloud. http://tinyurl.com/82oqfh2, Juin 2012. 28

[Bas12] BASS LEN, CLEMENTS PAUL ET KAZMAN RICK : *Software architecture in practice*. Addison-Wesley, 2012. 71

[Bei07] BEISIEGEL MICHAEL, BOOZ DAVE, COLYER ADRIAN, HILDEBRAND HAL, MARINO JIM ET TAM KEN : SCA Service Component Architecture. http://www.osoa.org/, Mars 2007. :2007

[Bel10a] BELOGLAZOV ANTON ET BUYYA RAJKUMAR : Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. *In Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, pages 1–4. ACM, 2010. 55

[Bel10b] BELOGLAZOV ANTON ET BUYYA RAJKUMAR : Energy efficient resource management in virtualized cloud data centers. *In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831. IEEE Computer Society, 2010. 55

[Ber09] BERNSTEIN DAVID, LUDVIGSON ERIK, SANKAR KRISHNA, DIAMOND STEVE, MORROW MONIQUE : Blueprint for the intercloud-protocols and formats for cloud computing interoperability. *In the fourth International Conference on Internet and Web Applications and Services, ICIW'09.*, pages 328–336. IEEE, 2009. 30

[Bin12] BINZ TOBIAS, BREITER GERD, LEYMAN FRANK ET SPATZIER THOMAS : Portable cloud services using tosca. *Internet Computing, IEEE*, 16(3):80–85, 2012. 68

[Bin13] BINZ, TOBIAS AND BREITENBÜCHER, UWE AND HAUPT, FLORIAN AND KOPP, OLIVER AND LEYMANN, FRANK AND NOWAK, ALEXANDER AND WAGNER, SEBASTIAN : OpenTOSCA–A Runtime for TOSCA-based Cloud Applications. *In Service-Oriented Computing*, pages 692–695. Springer, 2013. 32

[Bir94] BIRMAN KENNETH P ET VAN RENESSE ROBBERT : *Reliable distributed computing with the Isis toolkit*, volume 85. IEEE Computer Society Press Los Alamitos, 1994. 58

[Bod09] BODIK PETER, GRIFFITH REAN, SUTTON CHARLES, FOX ARMANDO, JORDAN MICHAEL ET PATTERSON DAVID : Statistical machine learning makes automatic control practical for internet datacenters. *In Proceedings of the 2009 Conference on Hot topics in Cloud Computing*, Berkeley, CA, USA, 2009. USENIX Association. 17

[Boe88] BOEHM BARRY W : A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988. 39

[Boj11] BOJANOVA IRENA ET SAMBA AUGUSTINE : Analysis of cloud computing delivery architecture models. *In IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA)*, pages 453–458. IEEE, 2011. 10

[Bou03] BOUTEILLER AURÉLIEN, LEMARINIER PIERRE, KRAWEZIK K ET CAPELLO F : Coordinated checkpoint versus message log for fault tolerant mpi. *In Proceedings IEEE International Conference on Cluster Computing*, pages 242–250. IEEE, 2003. 132

[Bra09] BRANDIC IVONA : Towards self-manageable cloud services. *In Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 2, pages 128–133. IEEE, 2009. 17

[Bre05] BREITGAND DAVID, HENIS EALAN ET SHEHORY ONN : Automated and adaptive threshold setting : Enabling technology for autonomy and self-management. *In Proceedings. Second International Conference on Autonomic Computing, ICAC 2005.*, pages 204–215. IEEE, 2005. 17

[Bri09] BRISCOE GERARD ET MARINOS ALEXANDROS : Digital ecosystems in the clouds : Towards community cloud computing. *CoRR*, abs/0903.0694, 2009. http ://dblp.uni-trier.de/db/journals/corr/corr0903.html#abs-0903-0694. 27

[Bru10] BRUNELIERE HUGO, CABOT JORDI ET JOUAULT FRÉDÉRIC : Combining model-driven engineering and cloud computing. *In Modeling, Design et Analysis for the Service Cloud-MDA4ServiceCloud'10 : Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010)*, 2010. 61

[Buy08] BUYYA RAJKUMAR, YEO CHEE SHIN ET VENUGOPAL SRIKUMAR : Market-oriented cloud computing : Vision, hype et reality for delivering it services as computing utilities. *In the 10th IEEE International Conference on High Performance Computing and Communications, HPCC'08.*, pages 5–13. IEEE, 2008. 9, 10, 11

[Buy10] BUYYA RAJKUMAR, RANJAN RAJIV ET CALHEIROS RODRIGO N : Intercloud : Utility-oriented federation of cloud computing environments for scaling of application services. *In Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010. 30, 56

[Cag95] CAGAN MARTIN : Untangling configuration management. *In Software Configuration Management*, pages 35–52. Springer, 1995. 79

[Cal11] CALHEIROS RODRIGO N, RANJAN RAJIV ET BUYYA RAJKUMAR : Virtual machine provisioning based on analytical performance and QoS in cloud computing environments. *In International Conference on Parallel Processing (ICPP)*, pages 295–304. IEEE, 2011. 17

[Cap05] CAPPELLO FRANCK, DJILALI SAMIR, FEDAK GILLES, HERAULT THOMAS, MAGNIETTE FRÉDÉRIC, NÉRI VINCENT ET LODYGENSKY OLEG : Computing on large-scale distributed systems : Xtremweb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3):417–437, 2005. 129

[Car99] CARDELLINI V., COLAJANNI M. ET YU P.S. : Dynamic load balancing on web-server systems. *Internet Computing, IEEE*, 3(3):28–39, 1999. 58

[Car09] CARL BROOKS : Heroku Outage. http://tinyurl.com/cxoomya, Juin 2009. 28

[Car10] CARLOS BLE : Goodbye Google App Engine (GAE). http://www.carlosble.com/2010/11/goodbye-google-app-engine-gae/, Novembre 2010. 133

[Car11] CARLIN SEAN ET CURRAN KEVIN : Cloud computing security. *International Journal of Ambient Computing and Intelligence (IJACI)*, 3(1):14–19, 2011. 61

[Car12] CARLINI EMANUELE, COPPOLA MASSIMO, DAZZI PATRIZIO, RICCI LAURA ET RIGHETTI GIACOMO : Cloud federations in Contrail. *In Euro-Par 2011 : Parallel Processing Workshops*, pages 159–168. Springer, 2012. xiv, 43, 57

[Cha79] CHANG ERNEST ET ROBERTS ROSEMARY : An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979. 130

[Cha01] CHASE JEFFREY S ETERSON DARRELL C, THAKAR PRACHI N, VAHDAT AMIN M, DOYLE RONALD P : Managing energy and server resources in hosting centers. *In ACM SIGOPS Operating Systems Review*, pages 103–116. ACM, 2001. 55

[Cha03] CHANDRA ABHISHEK, GOYAL PAWAN ET SHENOY PRASHANT : Quantifying the benefits of resource multiplexing in on-demand data centers. *Computer Science Department Faculty Publication Series*, pages 20–26, 2003. 121

[Cha07] CHAPPELL DAVID : Introducing SCA. http://www.davidchappell.com/articles/introducing_sca.pdf, 2007. 5

[Cha10] CHAPMAN CLOVIS, EMMERICH WOLFGANG, MARQUEZ FERMIN GALÁN, CLAYMAN STUART ET GALIS ALEX : Elastic service definition in computational clouds. *In IEEE/IFIP on Network Operations and Management Symposium Workshops (NOMS Wksps)*, pages 327–334. IEEE, 2010. 9, 55, 57, 63, 94

[Cha12] CHAPMAN CLOVIS, EMMERICH WOLFGANG, MÁRQUEZ FERMÍN GALÁN, CLAYMAN STUART ET GALIS ALEX : Software architecture definition for on-demand cloud provisioning. *Cluster Computing*, 15(2):79–100, 2012. 68

[Che01] CHEESMAN JOHN ET DANIELS JOHN : *UML components*. Addison-Wesley Reading, 2001. 72

[Che08] CHENG ZHILI, DU ZHIHUI, CHEN YINONG ET WANG XIAOYING : Soavm : A service-oriented virtualization management system with automated configuration. *In IEEE International Symposium on Service-Oriented System Engineering, SOSE'08.*, pages 251–256. IEEE, 2008. 3

[Chh10] CHHABRA BHARAT, VERMA DINESH ET TANEJA BHAWNA : Software engineering issues from the cloud application perspective. *International Journal of Information Technology and Knowledge Management*, 2(2):669–673, 2010. 30

[Cho09] CHOW RICHARD, GOLLE PHILIPPE, JAKOBSSON MARKUS, SHI ELAINE, STADDON JESSICA, MASUOKA RYUSUKE ET MOLINA JESUS : Controlling data in the cloud : outsourcing computation without outsourcing control. *In Proceedings of the 2009 ACM workshop on Cloud Computing Security*, pages 85–90. ACM, 2009. 27

[Cho10] CHOPRA INDERPREET ET SINGH MANINDER : Analysing the need for autonomic behaviour in grid computing. *In the 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 1, pages 535–539. IEEE, 2010. 17

[clo12] CLOUDSOFT : cloudsoft multi-cloud PaaS. http://www.cloudsoftcorp.com, Juin 2012. 102

[clo13] CLOUDBEES : PaaS. http://cloudbees.com, Mai 2013. 128

[Cou05] COULOURIS G.F., DOLLIMORE J. ET KINDBERG T. : *Distributed systems : concepts and design*. Addison-Wesley Longman, 2005. 117

[Cza05] CZAJKOWSKI G, WEGIEL M, DAYNES, L, PALACZ K, JORDAN M, SKINNER G ET BRYCE C : Resource management for clusters of virtual machines. *In Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 1, pages 382–389. IEEE, 2005. 56

[da 11] DA SILVA EDUARDO GONÇALVES, PIRES LUÍS FERREIRA ET VAN SINDEREN MARTEN : Towards runtime discovery, selection and composition of semantic services. *Computer Communications*, 34(2):159–168, 2011. 61

[Dab02] DABEK F., ZELDOVICH N., KAASHOEK F., MAZIERES D. ET MORRIS R. : Event-driven programming for robust software. *In Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002. 118

[Dan06] DAN XIE, SHI YING, TAO ZHANG, XIANG-YANG JIA, ZAO-QING LIANG ET JUN-FENG YAO : An approach for describing SOA. *In International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2006.*, pages 1–4. IEEE, 2006. 62

[Dan12] DANDRIA FRANCESCO, BOCCONI STEFANO, CRUZ JESUS GORRONOGOITIA, AHTES JAMES, ZEGINIS DIMITRIS : Cloud4SOA : Multi-Cloud Application Management Across PaaS Offerings. *In the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 407–414. IEEE, 2012. 54, 55, 59

[Das] DASEIN : The Dasein Cloud API. http://dasein-cloud.sourceforge.net/. 56

[Dav12] DAVIS D ET PILZ G : Cloud infrastructure management interface (cimi) model and rest interface over http. *vol. DSP-0263, May*, 2012. 32

[De 05] DE WOLF TOM, SAMAEY GIOVANNI, HOLVOET TOM ET ROOSE DIRK : Decentralised autonomic computing : Analysing self-organising emergent behaviour using advanced numerical methods. *In the Second International Conference on Autonomic Computing,ICAC 2005. Proceedings.*, pages 52–63. IEEE, 2005. 17

[Dea08] DEAN JEFFREY ET GHEMAWAT SANJAY : MapReduce : simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 144

[Dij82] DIJKSTRA EDSGER W : On the role of scientific thought. *In Selected Writings on Computing : A Personal Perspective*, pages 60–66. Springer, 1982. 77

[DMT11] DMTF : DMTF to develop standards for managing a cloud computing environment. http://www.dmtf.org/standards/cloud/, Juin 2011. 32

[dot13] DOTCLOUD : PaaS. https://dotcloud.com/, Mai 2013. 128

[Dra12] DRAGO IDILIO, MELLIA MARCO, M MUNAFO MAURIZIO, SPEROTTO ANNA, SADRE RAMIN ET PRAS AIKO : Inside dropbox : understanding personal cloud storage services. *In Proceedings of the 2012 ACM Conference on Internet Measurement*, pages 481–494. ACM, 2012. 12

[Edm12] EDMONDS ANDY, METSCH THIJS, PAPASPYROU ALEXANDER ET RICHARDSON ALEXIS : Toward an open cloud standard. *Internet Computing, IEEE*, 16(4):15–25, 2012. 32, 56

[Egh12] EGHAM : Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth. http://www.gartner.com/it/page.jsp?id=1924314/, Février 2012. 140

[Elm11] ELMROTH ERIK, TORDSSON JOHAN, HERNÁNDEZ FRANCISCO, ALI-ELDIN AHMED, SVÄRD PETTER, SEDAGHAT MINA ET LI WUBIN : Self-management challenges for multi-cloud architectures. *In Towards a Service-Based Internet*, pages 38–49. Springer, 2011. 57

[Ema13] EMAN HOSSNY, KHATTAB SHERIF, FATMA OMARA ET HESHAM HASSAN : A Case Study for Deploying Applications on Heterogeneous PaaS Platforms. *In International Conference on Cloud Computing and Big Data (CloudCom-Asia)*, Fuzhou, China, Décembre 2013. 53

[ESP] ESPER : Event stream processing. http://esper.codehaus.org. 116

[ETA] ETALIS : Event-driven Transaction Logic Inference System. http://code.google.com/p/etalis/. 116

[Etz10] ETZION OPHER ET NIBLETT PETER : *Event processing in action*. Manning Publications Co., 2010. 110

[Far12] FARD HAMID MOHAMMADI, PRODAN RADU, BARRIONUEVO JUAN JOSE DURILLO, FAHRINGER THOMAS : A multi-objective approach for workflow scheduling in heterogeneous environments. *In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 300–309. IEEE Computer Society, 2012. 63

[Fel04] FELBER PASCAL, KALDEWEY TIM ET WEISS STEFAN : Proactive hot spot avoidance for web server dependability. *In Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 309–318. IEEE, Oct 2004. 124

[Fer12] FERRER ANA JUAN, HERNÁNDEZ FRANCISCO, TORDSSON JOHAN, ELMROTH ERIK, ALI-ELDIN AHMED, ZSIGRI CSILLA, SIRVENT RAÜL, GUITART JORDI, BADIA ROSA M ET DJEMAME KARIM : Optimis : A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012. 22, 47, 59

[Fer13a] FERRY NICOLAS, CHAUVEL FRANCK, ROSSINI ALESSANDRO, MORIN BRICE ET SOLBERG ARNOR : Managing multi-cloud systems with CloudMF. *In Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, pages 38–45. ACM, 2013. 39

[Fer13b] FERRY NICOLAS, ROSSINI ALESSANDRO, CHAUVEL FRANCK, MORIN BRICE ET SOLBERG ARNOR : Towards model-driven provisioning, deployment, monitoring et adaptation of multi-cloud systems. *In CLOUD 2013 : IEEE 6th International Conference on Cloud Computing*, pages 887–894, 2013. 39

[For09] Distributed Management Task FORCE : Open Virtualization Format Specification DSP0243 v1.0.0, Février 2009. 32, 42, 55, 57

[Fos01] FOSTER IAN, KESSELMAN CARL ET TUECKE STEVEN : The anatomy of the grid : Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001. 2

[Fos06] FOSTER IAN, FREEMAN THOMAS, KEAHY KATE, SCHEFTNER DOUGLAS, SOTOMAYER BORJA ET ZHANG XUEHAI : Virtual clusters for grid communities. *In Cluster Computing and the Grid, Sixth IEEE International Symposium on CCGRID 06*, volume 1, pages 513–520. IEEE, 2006. 56

[Fou] Apache FOUNDATION : Apache Libcloud a unified interface to the cloud. http://libcloud.apache.org/. 56

[Fou11] VMWARE Cloud FOUNDRY : Cloud Foundry PaaS. http://www.cloudfoundry.com, Août 2011. xiv, 49, 50, 54, 55, 58, 59, 102

[Fra07] FRANCE ROBERT ET RUMPE BERNHARD : Model-driven development of complex software : A research roadmap. *In 2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007. 61

[Gal09] GALÁN FERMÍN, SAMPAIO AMERICO, RODERO-MERINO LUIS, LOY IRIT, GIL VICTOR ET VAQUERO LUIS M : Service specification in cloud environments based on extensions to open standards. *In Proceedings of the fourth international ICST conference on communication system software and middleware*, pages 1–19, New York, NY, USA, 2009. ACM. 68

[Gan11] GANTI RAGHU K, YE FAN ET LEI HUI : Mobile crowdsensing : Current state and future challenges. *Communications Magazine, IEEE*, 49(11):32–39, 2011. 140

[Gar93] GARLAN DAVID ET SHAW MARY : An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1:1–40, 1993. 71

[Gar05] GARG, VIJAY K : *Concurrent and distributed computing in Java*. Wiley-IEEE Press, 2005. 129, 132

[Gar12] GARCIA-GOMEZ SERGIO, JIMENEZ-GANAN MIGUEL, TAHER YEHIA, MOMM CHRISTOF, JUNKER FREDERIC, BIRO JOZSEF, MENYCHTAS ANDREAS ETRIKOPOULOS VASILIOS, STRAUCH STEVE : Challenges for the comprehensive management of cloud services in a paas framework. *Scalable Computing : Practice and Experience*, 13(3), 2012. xiii, 41, 42, 55, 57, 59

[Ghe02] GHEZZI CARLO, JAZAYERI MEHDI ET MANDRIOLI DINO : *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd édition, 2002. 30, 72

[Gig12] GIGASPACE : Cloudify PaaS. http://www.cloudifysource.org, Août 2012. xiv, 50, 51, 54, 55, 58, 59, 60, 97, 102

[Goi12] GOIRI ÍÑIGO, GUITART JORDI ET TORRES JORDI : Economic model of a cloud provider operating in a federated cloud. *Information Systems Frontiers*, 14(4):827–843, 2012. 18

[Gon10] GONZALEZ HECTOR, HALEVY ALON Y, JENSEN CHRISTIAN S, LANGEN ANNO, MADHAVAN JAYANT, SHAPLEY REBECCA, SHEN WARREN ET GOLDBERG-KIDON JONATHAN : Google fusion tables : web-centered data management and collaboration. *In Proceedings of the 2010 International Conference on Management of Data*, pages 1061–1066. ACM, 2010. 117, 128

[Goo04] GOOGLE INC : Google Mail. http://mail.google.com/, Avril 2004. 12

[Goo05] GOOGLE INC : Google Docs. http://docs.google.com/, Août 2005. 12

[Gre08] GREENBERG ALBERT, HAMILTON JAMES, MALTZ DAVID A ET PATEL PARVEEN : The cost of a cloud : research problems in data center networks. *ACM SIGCOMM Computer Communication Review*, 39(1):68–73, 2008. 26

[Gro87] Network Working GROUP : Domain Names - Concepts and Facilities et related other RFCs. http://ietf.org/rfc/rfc1034.txt, Novembre 1987. 118

[Gro00] Network Working GROUP : RFC 2988. http://tools.ietf.org/html/rfc2988, Novembre 2000. 124

[Gro12] GROZEV NIKOLAY ET BUYYA RAJKUMAR : Inter-Cloud Architectures and Application Brokering : Taxonomy and Survey. *Software : Practice and Experience*, 2012. http ://dx.doi.org/10.1002/spe.2168. viii, xiii, 9, 18, 20, 21, 22, 64

[Gui10] GUIDE DEVELOPER : Amazon elastic load balancing. *Citeseer*, 2010. 56

[Had] HADERER NICOLAS, PARAISO FAWAZ, RIBEIRO CHRISTOPHE, MERLE PHILIPPE, ROUVOY ROMAIN ET SEINTURIER LIONEL : *A Cloud-based Infrastructure for Crowdsourcing Data from Mobile Devices*. Springer Review, To appear. 6

[Had13] HADERER NICOLAS, ROUVOY ROMAIN ET SEINTURIER LIONEL : A preliminary investigation of user incentives to leverage crowdsensing activities. *In 2nd International IEEE PerCom Workshop on Hot Topics in Pervasive Computing (PerHot)*, San Diego, États-Unis, Mars 2013. IEEE Computer Society. 140

[Han03] HANDSCHUH SIEGFRIED ET STAAB STEFFEN : *Annotation for the semantic web*, volume 96. IOS Press, 2003. 79

[Han12] HAN RUI, GUO LI, GHANEM MOUSTAFA M ET GUO YIKE : Lightweight resource scaling for cloud applications. *In the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 644–651. IEEE, 2012. 63, 94

[Har97] HARCHOL-BALTER M. ET DOWNEY A.B. : Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems (TOCS)*, 15(3):253–285, 1997. 58

[Har11] HARSH PIYUSH, JEGOU YVON, CASCELLA ROBERTO G ET MORIN CHRISTINE : Contrail virtual execution platform challenges in being part of a cloud federation. *In Towards a Service-Based Internet*, pages 50–61. Springer, 2011. 59

[Has09] HASSAN SHENIN, AL-JUMEILY DHIYA ET HUSSAIN ABIR JAAFAR : Autonomic computing paradigm to support system's development. *In the second International Conference on Developments in eSystems Engineering (DESE)*, pages 273–278. IEEE, 2009. 17

[Hog11] HOGAN MICHAEL, LIU FANG, SOKOL ANNIE ET TONG JIN : NIST cloud computing standards roadmap. *NIST Special Publication*, 35, 2011. 18

[IEE11] IEEE : IEEE STANDARDS ASSOCIATION. http://standards.ieee.org/develop/project/2301.html, Septembre 20011. 31

[IMB06] IMB : An architectural blueprint for autonomic computing. *IBM White Paper*, 2006. 103, 122

[Inf] INFOWORLD : The 10 worst cloud outages (and what we can learn from them). http://tinyurl.com/br9ck4a. 27

[Isa07] ISARD MICHAEL : Autopilot : automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007. 27

[IWG] IWGCR : International Working Group on Cloud Computing Resiliency. http://iwgcr.org. 27

[J05] Tatemura J : CDDLM XML Configuration Description Language Specification version 1.0. *In CDDLM XML Configuration Description Language Specification version 1.0*. GGF, 2005. 62

[Jai12] JAIN PRITESH, RANE DHEERAJ ET PATIDAR SHYAM : A novel cloud bursting brokerage and aggregation (cbba) algorithm for multi cloud environment. *In Second International Conference on Advanced Computing & Communication Technologies (ACCT)*, pages 383–387. IEEE, 2012. 61

[Jan11] JANSEN W.A. : Cloud hooks : Security and privacy issues in cloud computing. *In the 44th Hawaii International Conference on System Sciences (HICSS)*, pages 1–10. IEEE, 2011. 27

[Jer08] JEREMY GEELAN : Twenty one experts define cloud computing. Virtualization, Electronic Magazine. http://virtualization.sys-con.com/node/612375, Août 2008. 9, 10, 11

[Jia10] JIANG XIAOYAN, ZHANG YONG ET LIU SHIJUN : A Well-designed SaaS Application Platform Based on Model-driven Approach. *In the 9th International Conference on Grid and Cooperative Computing (GCC)*, pages 276–281. IEEE, 2010. 61

[Joo05] JOOLIA ACKBAR, BATISTA THAIS, COULSON GEOFF ET GOMES ANTONIO TADEU A : Mapping ADL specifications to an efficient and reconfigurable runtime component platform. *In the 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005*, pages 131–140. IEEE, 2005. 104

[K07] Kelly K : A cloudbook for the cloud. *Luettu*, 24:2012, 2007. 30

[Kam13] KAMATERI ELENI, LOUTAS NIKOLAOS, ZEGINIS DIMITRIS, AHTES JAMES, D'ANDRIA FRANCESCO, BOCCONI STEFANO, GOUVAS PANAGIOTIS, LEDAKIS GIANNIS, RAVAGLI FRANCO ET LOBUNETS OLEKSANDR : Cloud4SOA : A Semantic-Interoperability PaaS Solution for Multi-cloud Platform Management and Portability. *In Service-Oriented and Cloud Computing*, pages 64–78. Springer, 2013. xiii, 37, 55

[Kar87] KARN PHIL ET PARTRIDGE CRAIG : Improving round-trip time estimates in reliable transport protocols. *ACM SIGCOMM Computer Communication Review*, 17(5):2–7, 1987. 124

[Kel85] KELLER JAMES M, GRAY MICHAEL R ET GIVENS JAMES A : A fuzzy k-nearest neighbor algorithm. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-15(4):580–585, 1985. 82, 147

[Kep03] KEPHART JEFFREY O ET CHESS DAVID M : The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. 17

[Kic01] KICZALES GREGOR, HILSDALE ERIK, HUGUNIN JIM, KERSTEN MIK, PALM JEFFREY, GRISWOLD WILLIAM G : An overview of AspectJ. *In ECOOP 2001—Object-Oriented Programming*, pages 327–354. Springer, 2001. 77

[Koe03] KOEHLER JANA, GIBLIN CHRIS, GANTENBEIN DIETER ET HAUSER RAINER : On autonomic computing architectures. *IBM Research, Zurich Research Laboratory, Tech. Rep*, 2003. 17

[Kra07] KRAKOWIAK SACHA : Middleware architecture with patterns and frameworks. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.140.3783, 2007. 101

[Kur11] KURZE TOBIAS, KLEMS MARKUS, BERMBACH DAVID, LENK ALEXANDER, TAI STEFAN ET KUNZE MARCEL : Cloud federation. *In CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs et Virtualization*, pages 32–38, 2011. 25

[La 09] LA HYUN JUNG ET KIM SOO DONG : A systematic process for developing high quality SaaS cloud services. *In Cloud Computing*, pages 278–289. Springer, 2009. 68

[Lan11] LANGER PHILIP, WIELAND KONRAD, WIMMER MANUEL ET CABOT JORDI : From UML profiles to EMF profiles and beyond. *In Objects, Models, Components, Patterns*, pages 52–67. Springer, 2011. 79

[Lea09] LEAVITT N. : Is cloud computing really ready for prime time. *IEEE Computer*, 42(1):15–20, Janvier 2009. 27

[Leb95] LEBLANG DAVID B. : Configuration management. *In* Walter F. TICHY, éditeur : *Configuration management*, chapitre The CM challenge : configuration management that works, pages 1–37. John Wiley & Sons, Inc., New York, NY, USA, 1995. 79

[Lee11] LEE BU SUNG, YAN SHIXING, MA DING ET ZHAO GUOPENG : Aggregating IaaS service. *In SRII Global Conference (SRII), 2011 Annual*, pages 335–338. IEEE, 2011. 10

[Lef11] LEFEVRE LAURENT, MORNARD OLIVIER, GELAS J-P ET MOREL MAXIME : Monitoring Energy Consumption in clouds : the CompatibleOne experience. *In Ninth International Conference on Dependable Autonomic and Secure Computing (DASC)*, pages 794–795. IEEE, 2011. 56

[Li 09] LI BO, LI JIANXIN, HUAI JINPENG, WO TIANYU, LI QIN ET ZHONG LIANG : Enacloud : An energy-saving application live placement approach for cloud computing environments. *In IEEE International Conference on Cloud Computing, CLOUD'09*, pages 17–24. IEEE, 2009. 55

[Lim09] LIM, H.C., BABU S., CHASE J.S. ET PAREKH S.S. : Automated control in cloud computing : challenges and opportunities. *In Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM New York, NY, USA, 2009. 57

[Lim10] LIM HAROLD C, BABU SHIVNATH ET CHASE JEFFREY S : Automated control for elastic storage. *In Proceedings of the 7th International Conference on Autonomic Computing*, pages 1–10. ACM, 2010. 57

[Lin92] LIN H.C. ET RAGHAVENDRA CS : A dynamic load-balancing policy with a central job dispatcher (LBC). *IEEE Transactions on Software Engineering*, 18(2):148–158, 1992. 58

[Lin05] LIN PAUL, MACARTHUR ALEXANDER ET LEANEY JOHN : Defining autonomic computing : a software engineering perspective. *In Proceedings. Software Engineering Conference, Australian*, pages 88–97. IEEE, 2005. 17

[Lit88] LITZKOW MICHAEL J, LIVNY MIRON ET MUTKA MATT W : Condor-a hunter of idle workstations. *In the 8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988. 129

[Liu11a] LIU TIANCHENG, KATSUNO YASUHARU, SUN KEWEI, LI YING, KUSHIDA TAKAYUKI, CHEN YING ET ITAKURA MAYUMI : Multi cloud management for unified cloud services across cloud sites. *In IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*, pages 164–169. IEEE, 2011. 61

[Liu11b] LIU WENJIE ET LI ZHANHUAI : Research and design of autonomic computing system model in cloud computing environment. *In International Conference on Multimedia Technology (ICMT)*, pages 5025–5028. IEEE, 2011. 17

[Luc08] LUCKHAM DAVID ET SCHULTE ROY : Event Processing Glossary - Version 1.1. *Processing*, 1.1(July):1–19, 2008. http://complexevents.com/wp-content/uploads/2008/08/epts-glossary-v11.pdf. 144

[Luc11] LUCAS SIMARRO JOSÉ LUIS, MORENO-VOZMEDIANO RAFAEL, MONTERO RUBEN S ET LLORENTE IGNACIO MARTÍN : Dynamic placement of virtual machines for cost optimization in multi-cloud environments. *In International Conference on High Performance Computing and Simulation (HPCS)*, pages 1–7. IEEE, 2011. 61

[Lyo13] LYONS BETH GOELZER ET PARKER TOM : Office 365 : tips to avoid turbulence while moving faculty and staff to the cloud. *In* Lisa BROWN, Laurie FOX et Jean TAGLIAMONTE, éditeurs : *SIGUCCS*, pages 131–136. ACM, 2013. 12

[Mac09] MACHADO GUILHERME SPERB, HAUSHEER DAVID ET STILLER BURKHARD : Considerations on the interoperability of and between cloud computing standards. *In 27th Open Grid Forum (OGF27), G2C-Net Workshop : From Grid to Cloud Networks, Banff, Canada*, 2009. 30

[Mal00] MALPANI NAVNEET, WELCH JENNIFER L ET VAIDYA NITIN : Leader election algorithms for mobile ad hoc networks. *In Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 96–103. ACM, 2000. 130

[Mal11] MALATHI M : Cloud computing concepts. *In the 3rd International Conference on Electronics Computer Technology (ICECT)*, volume 6, pages 236–239. IEEE, 2011. 10

[Man12] MANIAS ELTON ET BAUDE FRANÇOISE : A component-based middleware for hybrid grid/cloud computing platforms. *Concurrency and Computation : Practice and Experience*, 24(13):1461–1477, 2012. 54

[Mar03] MARCUS EVAN ET STERN HAL : *Blueprints for high availability*. Wiley, 2003. 154

[Mar10a] MARINO JIM ET ROWLEY MICHAEL : *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 2010. 25, 73

[Mar10b] Vukolić MARKO : The byzantine empire in the intercloud. *ACM SIGACT News*, 41(3):105–111, 2010. 61

[Mar10c] MARSHALL P., KEAHEY K. ET FREEMAN T. : Elastic site : Using clouds to elastically extend site resources. *In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52. IEEE Computer Society, 2010. 56

[Mau11] MAURER MICHAEL, BRESKOVIC IVAN, EMEAKAROHA VINCENT C ET BRANDIC IVONA : Revealing the MAPE loop for the autonomic management of Cloud infrastructures. *In IEEE Symposium on Computers and Communications (ISCC)*, pages 147–152. IEEE, 2011. 17

[Mau12] MAURICE GAGNAIRE, FELIPE DIAZ, CAMILLE COTI, CHRISTOPHE CERIN, KAZUHIKO SHIOZAKI, YINGJIE XU, PIERRE DELORT, JEAN-PAUL SMETS, JONATHAN LE LOUS, STEPHEN LUBIARZ ET PIERRICK LECLERC : Downtime statistics of current cloud solutions. *International Working Group on Cloud Computing Resiliency, Tech. Rep*, Juin 2012. 28, 152, 154

[Max09] MAXIMILIEN E MICHAEL, RANABAHU AJITH, ENGEHAUSEN ROY ET ANDERSON LAURA C : Toward cloud-agnostic middlewares. *In Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 619–626. ACM, 2009. 68

[MAX12] MAXMIND : Bases de données GeoIP et Services Web GeoIP. http://www.maxmind.com/fr/geolocation_landing, Novembre 2012. xv, 112

[Mel11] MELL PETER ET GRANCE TIMOTHY : The NIST definition of cloud computing (draft). *NIST special publication*, 800(145):7, 2011. 9, 10, 11, 12, 13, 14, 15

[MG 09] MG SIEGLER : What Went Down At Rackspace Yesterday ? A Power Outage And Some Backup Failures. http://tinyurl.com/34j9tyf, Juin 2009. 28

[Mic06a] MICROSOFT : ASP.NET. http://msdn.microsoft.com/en-us/centrum-asp-net.aspx, Mars 2006. 25

[Mic06b] MICROSOFT : .NET Compact Framework 2.0. http://microsoft.com/en-us/download/details.aspx?id=22808, Mars 2006. 25

[Mic07] MICROSOFT : Using Data Annotations. http://msdn.microsoft.com/fr-fr/library/dd901590%28v=vs.95%29.aspx, Décembre 2007. 79

[Mic09] MICROSOFT : Microsoft MSDN AZURE Outage. http://tinyurl.com/clp2x2v, Mars 2009. 28

[Mic12] MICROSOFT : Microsoft MSDN AZURE Outage. http://tinyurl.com/cdy4scn, Février' 2012. 28

[Mie08] MIETZNER RALPH ET LEYMANN FRANK : Towards provisioning the cloud : On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for saas applications. *In IEEE Congress on Services-Part I*, pages 3–10. IEEE, 2008. 55

[Mie10] MIETZNER RALPH : *A method and implementation to define and provision variable composite applications et its usage in cloud computing*. Thèse de doctorat, University of Stuttgart, 2010. http ://d-nb.info/1012539598. 68

[Mil11] MILOJIČIĆ DEJAN, LLORENTE IGNACIO M ET MONTERO RUBEN S : OpenNebula : A cloud management tool. *Internet Computing, IEEE*, 15(2):11–14, 2011. 12

[MOD13] MODACLOUDS : MODACLOUDS architecture. http://www.modaclouds.eu/wp-content/uploads/2012/09/MODAClouds_D3.2.1_MODACloudsArchitectureInitialVersion.pdf, Avril 2013. xiii, 40, 41

[Moh11] MOHAGHEGHI PARASTOO ET SÆTHER THOR : Software engineering challenges for migration to the service cloud paradigm : Ongoing work in the REMICS project. *In IEEE World Congress on Services (SERVICES)*, pages 507–514. IEEE, 2011. xiii, 38, 59

[Mor11a] MORÁN DANIEL, VAQUERO LUIS M ET GALÁN FERMIN : Elastically ruling the cloud : specifying application's behavior in federated clouds. *In IEEE International Conference on Cloud Computing (CLOUD)*, pages 89–96. IEEE, 2011. 63, 94

[Mor11b] MORENO-VOZMEDIANO RAFAEL, MONTERO RUBEN S ET LLORENTE IGNACIO M : Multicloud deployment of computing clusters for loosely coupled MTC applications. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):924–930, 2011. 61

[Mos96] MOSER LOUISE E., MELLIAR-SMITH P MICHAEL, AGARWAL DEBORAH A., BUDHIA RAVI K. ET LINGLEY-PAPADOPOULOS COLLEEN A. : Totem : A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996. 58

[Mos98] MOSBERGER DAVID ET JIN TAI : httperf a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, décembre 1998. http ://doi.acm.org/10.1145/306225.306235. 151

[Mos11] MOSCATO FRANCESCO, AVERSA ROCCO, DI MARTINO BENIAMINO, FORTIS T ET MUNTEANU VICTOR : An analysis of mOSAIC ontology for Cloud resources annotation. *In Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 973–980. IEEE, 2011. xiii, 33, 34, 59, 60, 102

[Mus11] MUSHI JOSEPH COSMAS, TAN GUAN-ZHENG, MUSAU FELIX ET WILSON CHERUIYOT : Modeling M-SaaS delivery model for threshold-based credit recharging using M-banking. *In the 3rd International Conference on Computer Research and Development (ICCRD)*, volume 2, pages 307–311. IEEE, 2011. 10

[Nur09] NURMI DANIEL, WOLSKI RICHARD, GRZEGORCZYK CHRIS, OBERTELLI GRAZIANO, SOMAN SUNIL, YOUSEFF LAMIA ET ZAGORODNOV DMITRII : The eucalyptus open-source cloud-computing system. *In 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. CCGRID'09*, pages 124–131. IEEE, 2009. 2, 12

[OAS07] OASIS : Service Component Architecture (SCA). http://www.oasis-opencsa.org/sca, Mars 2007. 25

[OAS08] OASIS : Solution deployment descriptor specification 1.0. http://docs.oasis-open.org/sdd/v1.0/os/sdd-spec-v1.0-os.pdf, Septembre 2008. :SSD

[OAS12a] OASIS : OASIS Cloud Application Management for Platforms (CAMP). http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html/, Août 2012. 31

[OAS12b] OASIS : Topology and Orchestration Specification for Cloud Applications. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html/, Janvier 2012. 32, 54

[Obe10] OBERLE KARSTEN ET FISHER MIKE : ETSI CLOUD–initial standardization requirements for cloud services. *In Economics of Grids, Clouds, Systems et Services*, pages 105–115. Springer, 2010. 54

[Paa13] PAASAGE : PaaSage Deliverable D1.6.1 . http://www.paasage.eu/images/documents/paasage_d161_final.pdf, Août 2013. xiii, 39, 40, 55, 59

[Pap07] PAPAZOGLOU MIKE P ET TRAVERSO PAOLO, DUSTDAR SCHAHRAM, AET LEYMANN FRANK : Service-oriented computing : State of the art and research challenges. *Computer*, 40(11):38–45, 2007. 74

[Pap08] PAPAZOGLOU MICHAEL P, TRAVERSO PAOLO, DUSTDAR SCHAHRAM ET LEYMANN FRANK : Service-oriented computing : a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008. 61

[Par] PARAISO FAWAZ, MERLE PHILIPPE ET SEINTURIER LIONEL : soCloud : A service-oriented component-based PaaS for managing portability, provisioning, elasticity et high availability across multiple clouds. *Springer Computing Journal*, To appear. 6

[Par12a] PARAISO FAWAZ, HADERER NICOLAS, MERLE PHILIPPE, ROUVOY ROMAIN ET SEINTURIER LIONEL : A federated multi-cloud paas infrastructure. *In IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 392–399. IEEE, 2012. 6, 121

[Par12b] PARAISO FAWAZ, HERMOSILLO GABRIEL, ROUVOY ROMAIN, MERLE PHILIPPE, SEINTURIER LIONEL : A Middleware Platform to Federate Complex Event Processing. *In Sixteenth IEEE International EDOC Conference*, pages 113–122, Beijing, China, septembre 2012. Springer. http ://hal.inria.fr/hal-00700883. 6, 116, 144

[Par13] PARAISO FAWAZ, MERLE PHILIPPE ET SEINTURIER LIONEL : Managing elasticity across multiple cloud providers. *In Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 53–60. ACM, 2013. 6, 15, 61

[Paw12] PAWLUK PRZEMYSLAW, SIMMONS BRADLEY, SMIT MICHAEL, LITOIU MARIN ET MANKOVSKI SERGE : Introducing stratos : A cloud broker service. *In IEEE CLOUD*, pages 891–898, 2012. xiv, 45, 47, 59, 61

[Pen12] PENG CHUNYI, KIM MINKYONG, ZHANG ZHE ET LEI HUI : VDN : Virtual machine image distribution network for cloud data centers. *In Proceedings IEEE. INFOCOM*, pages 181–189. IEEE, 2012. 123

[Pep11] PEPPLE KEN : *Deploying OpenStack*. O'Reilly, 2011. 12

[Pet11a] PETCU DANA : Portability and interoperability between clouds : challenges and case study. *In Towards a Service-Based Internet*, pages 62–74. Springer, 2011. 68

[Pet11b] PETCU DANA, CRĂCIUN CIPRIAN, NEAGUL MARIAN, PANICA SILVIU, DI MARTINO BENIAMINO, VENTICINQUE SALVATORE, RAK MASSIMILIANO ET AVERSA ROCCO : Architecturing a sky computing platform. *In ServiceWave 2010 Workshop Towards a Service-Based Internet*, pages 1–13. Springer, 2011. 22, 33

[Pet13a] PETCU DANA : Multi-cloud : expectations and current approaches. *In Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 1–6. ACM, 2013. viii, xiii, 9, 18, 19, 20, 21, 22, 64

[Pet13b] PETCU DANA, MACARIU GEORGIANA, PANICA SILVIU ET CRCIUN CIPRIAN : Portable cloud applications-from theory to practice. *Future Gener. Comput. Syst.*, 29(6):1417–1430, août 2013. 24, 54, 57

[Pie12] PIERRE GUILLAUME ET STRATAN CORINA : ConPaaS : a platform for hosting elastic cloud applications. *Internet Computing, IEEE*, 16(5):88–92, 2012. xiv, 43, 44, 57, 58, 59

[Pop10] POPOVIC K ET HOCENSKI Z. : Cloud computing security issues and challenges. *In MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 344–349, May 2010. 12

[Pro11] PRODAN RADU, WIECZOREK MAREK ET FARD HAMID MOHAMMADI : Double auction-based scheduling of scientific applications in distributed grid and cloud environments. *Journal of Grid Computing*, 9(4):531–548, 2011. 18

[Pyt06] PYTHON : Function annotations. http://www.python.org/dev/peps/pep-3107/, Décembre 2006. 79

[Qia07] QIAN HANGWEI, MILLER ELLIOT, ZHANG WEI, RABINOVICH MICHAEL ET WILLS CRAIG E : Agility in virtualized utility computing. *In Second International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, pages 1–8. IEEE, 2007. 58

[Ren04] RENNIE MICHAEL WILLIAM ET MISIC VB : Towards a service-based architecture description language, 2004. 62

[Ric09] RICH MILLER : Salesforce.com Hit by One Hour Outage. http://tinyurl.com/yfxfwm5, Juin 2009. 28

[Ric10] RICH MILLER : Amazon Addresses EC2 Power Outages. http://tinyurl.com/3akfv7q, Mai 2010. 28

[Ric11] RICH MILLER : Major Amazon Outage Ripples Across Web. http://tinyurl.com/3qpp2ts, Avril 2011. 28

[RJ11] Xu Cheng-Zhong et Wang Kun RAO JIA, Bu Xiangping : A distributed self-learning approach for elastic provisioning of virtualized cloud resources. *In IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 45–54. IEEE, 2011. 123

[Roc09] ROCHWERGER BENNY, BREITGAND DAVID, LEVY ELIEZER, GALIS ALEX, NAGIN KENNETH, LLORENTE IGNACIO MARTÍN, MONTERO RUBÉN, WOLFSTHAL YARON, ELMROTH ERIK ET CACERES JUAN : The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4–1, 2009. xiii, 2, 35, 36, 55, 57, 58, 59

[Rod10] RODERO-MERINO LUIS, VAQUERO LUIS M, GIL VICTOR, GALÁN FERMÍN, FONTÁN JAVIER, MONTERO RUBÉN S ET LLORENTE IGNACIO M : From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26(8):1226–1240, 2010. 57

[Roy11] ROY NILABJA, DUBEY ABHISHEK ET GOKHALE ANIRUDDHA : Efficient autoscaling in the cloud using predictive models for workload forecasting. *In IEEE International Conference on Cloud Computing (CLOUD)*, pages 500–507. IEEE, 2011. 17

[Sav11] SAVU LAURA : Cloud computing : Deployment models, delivery models, risks and research challenges. *In International Conference on Computer and Management (CAMAN)*, pages 1–4. IEEE, 2011. 12

[Sch12] SCHIPER ANDRÉ ET SANTOS NUNO FILIPE DE SOUSA : State machine replication, Août 2012. 165

[Sea] SEARCHCLOUDCOMPUTING : Cloud computing outages : What can we learn ? http://tinyurl.com/cnnlrg3. 27

[Sei09] SEINTURIER LIONEL, MERLE PHILIPPE, FOURNIER DAMIEN, DOLET NICOLAS, SCHIAVONI VALERIO ET STEFANI J-B : Reconfigurable SCA applications with the FraSCAti platform. *In International Conference on Services Computing, SCC'09.*, pages 268–275. IEEE, 2009. 73, 75

[Sei12] SEINTURIER LIONEL, MERLE PHILIPPE, ROUVOY ROMAIN, ROMERO DANIEL, SCHIAVONI VALERIO ET STEFANI JEAN-BERNARD : A component-based middleware platform for reconfigurable service-oriented architectures. *Software : Practice and Experience*, 42(5):559–583, 2012. 73, 75

[Sel13] SELLAMI MOHAMED, YANGUI SAMI, MOHAMED MOHAMED ET TATA SAMIR : PaaS-independent Provisioning and Management of Applications in the Cloud. *In IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 693–700. IEEE, 2013. 55, 60

[Set09] SETH ELIOT : A Summary of the Amazon Web Services June 29 Outage. http://tinyurl.com/cokv63t, Juin 2009. 28

[Sha11] SHARMA RITU ET SOOD MANU : A Model Driven Approach to Cloud SaaS Interoperability. *International Journal of Computer Applications*, 30(8):1–8, 2011. 61

[Shr95] SHREEDHAR M. ET VARGHESE GEORGE : Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev.*, 25(4):231–242, octobre 1995. http ://doi.acm.org/10.1145/217391.217453. 118

[Sil12] SILVA ELIAS, ADRIANO NOGUEIRA DA ET LUCRÉDIO, DANIEL : Software engineering for the cloud : A research roadmap. *In 26th Brazilian Symposium on Software Engineering (SBES)*, pages 71–80. IEEE, 2012. 61

[Sli11] SLIK D, SIEFER M, HIBBARD E, SCHWARZER C, YODER A, BAIRAVASUNDARAM LN, BAKER S, CARLSON M, NGUYEN H ET RAMOS R : Cloud data management interface (cdmi) v1. 0, 2011. 32

[Sot13] SOTIRIADIS STELIOS, BESSIS NIK, KUONEN PIERRE ET ANTONOPOULOS NICK : The inter-cloud meta-scheduling (ICMS) framework. *In IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 64–73. IEEE, 2013. 18

[Sou10] SOUNDARARAJAN VIJAYARAGHAVAN ET GOVIL KINSHUK : Challenges in building scalable virtualized datacenter management. *ACM SIGOPS Operating Systems Review*, 44(4):95–102, 2010. 1

[Sun89] SUN MICROSYSTEMS : JNDI Naming . http://www.antlr.org, Mai 1989. 120

[Sun00] SUN MICROSYSTEMS : Java Transaction API. http://www.jcp.org/en/jsr/detail?id=907, Juillet 2000. 117

[Sun01a] SUN MICROSYSTEMS : JMS JSR 914. http://jcp.org/en/jsr/detail?id=914, Juin 2001. 25

[Sun01b] SUN MICROSYSTEMS : JMX JSR 160. http://www.jcp.org/en/jsr/detail?id=160, Novembre 2001. 116

[Sun03a] SUN MICROSYSTEMS : EJB JSR 220. http://jcp.org/en/jsr/detail?id=220, Juin 2003. 25

[Sun03b] SUN MICROSYSTEMS : JAX-WS JSR 224. http://jcp.org/en/jsr/detail?id=224, Juin 2003. 25

[Sun04] SUN MICROSYSTEMS : Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html/, Août 2004. 79, 80

[Sun07] SUN MICROSYSTEMS : JAVA EE JSR 316. http://jcp.org/en/jsr/detail?id=316, Juillet 2007. 25

[Tar06] Willy TARREAU : Making applications scalable with load balancing. http://1wt.eu/articles/2006_lb/, Septembre 2006. 157

[Tch10] TCHANA ALAIN, TEMATE SUZY, BROTO LAURENT ET HAGIMONT DANIEL : Autonomic resource allocation in a J2EE cluster (regular paper). *In Utility and Cloud Computing, Chennai, India, 14/12/2010-16/12/2010*, page (electronic medium), http ://www.ieee.org/, dec 2010. IEEE. 55

[Tej08] TEJEDOR ENRIC ET BADIA ROSA M : COMP superscalar : bringing GRID superscalar and GCM together. *In the 8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID'08*, pages 185–193. IEEE, 2008. 49

[Teo01] TEO YONG MENG ET AYANI RASSUL : Comparison of load balancing strategies on cluster-based web servers. *Simulation*, 77(5-6):185–195, 2001. 118

[Ter00] TERENCE PARR : ANother Tool for Language Recognition (ANTLR). http://docs.oracle.com/javaee/1.4/tutorial/doc/Resources2.html, Mai 2000. 128

[Tor04] TORELL WENDY ET AVELAR VICTOR : Mean time between failure : Explanation and standards. *White Paper*, 78, 2004. 154

[Tsa10] TSAI WEI-TEK, SUN XIN ET BALASOORIYA JANAKA : Service-oriented cloud computing architecture. *In Seventh International Conference on Information Technology : New Generations (ITNG)*, pages 684–689. IEEE, 2010. 62, 68

[Uri13] URI BUDNIK : Lessons Learned from Recent Cloud Outages. http://tinyurl.com/qz5maey, Février 2013. 152

[Van02] VAN DEURSEN ARIE ET KLINT PAUL : Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, 10(1):1–17, 2002. 89

[Vaq08a] VAQUERO LUIS M, RODERO-MERINO LUIS, CACERES JUAN ET LINDNER MAIK : A break in the clouds : towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008. 9

[Vaq08b] VAQUERO LUIS M., RODERO-MERINO LUIS, CACERES JUAN ET LINDNER MAIK : A break in the clouds : Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, Décembre 2008. 9, 10, 11

[Vaq11] VAQUERO, L.M., RODERO-MERINO L. ET BUYYA R. : Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011. 56

[Vec09] VECCHIOLA CHRISTIAN, CHU XINGCHEN ET BUYYA RAJKUMAR : Aneka : a software platform for .net-based cloud computing. *High Speed and Large Scale Scientific Computing*, pages 267–295, 2009. xiv, 43, 45, 46, 55, 57, 59, 60

[Vil12a] VILLARI MASSIMO, BRANDIC IVONA, TUSA FRANCESCO, CELESTI ANTONIO, KECSKEMETI GABOR, CALCAVECCHIA NICOLÒ MARIA, DI NITTO ELISABETTA, MIKKILINENI RAO, MORANA GIOVANNI ET SEYLER IAN : *Achieving Federated and Self-Manageable Cloud Infrastructures : Theory and Practice*. Business Science Reference, 2012. 22

[Vil12b] VILLEGAS DAVID, BOBROFF NORMAN, RODERO IVAN, DELGADO JAVIER, LIU YANBIN, DEVARAKONDA ADITYA, FONG LIANA, MASOUD SADJADI S,ET PARASHAR MANISH : Cloud federation in a layered service model. *Journal of Computer and System Sciences*, 78(5):1330–1344, 2012. 61

[VMW12] VMWARE : Hyperic SIGAR API. http://hyperic.com/products/sigar, Décembre 2012. 116

[Wan97] WANG YI-MIN : Consistent global checkpoints that contain a given set of local checkpoints. *Computers, IEEE Transactions on*, 46(4):456–468, 1997. 131

[Wan10] WANG LIZHE, VON LASZEWSKI GREGOR, YOUNGE ANDREW, HE XI, KUNZE MARCEL, TAO JIE ET FU CHENG : Cloud computing : a perspective study. *New Generation Computing*, 28(2):137–146, 2010. 9, 10, 11

[Was90] WASSERMAN ANTHONY I : Tool integration in software engineering environments. *In Software Engineering Environments*, pages 137–149. Springer, 1990. 79

[Wat92] WATKINS CHRISTOPHER JCH ET DAYAN PETER : Q-learning. *Machine learning*, 8(3-4):279–292, 1992. 164

[Yan11] YAN SHIXING, LEE BU SUNG, ZHAO GUOPENG, MA DING ET MOHAMED PEER : Infrastructure management of hybrid cloud for enterprise users. *In 5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management (SVM)*, pages 1–6. IEEE, 2011. 12

[Yan13] YANGUI SAMI, MARSHALL IAIN-JAMES, LAISNE JEAN-PIERRE ET TATA SAMIR : CompatibleOne : The Open Source Cloud Broker. *Journal of Grid Computing*, pages 1–17, 2013. xiv, 46, 48, 49, 54, 55, 58, 59, 61

[Zdn12] ZDNET : Amazon cloud down ; Reddit, Github, other major sites affected. http://tinyurl.com/95kmk8y, Octobre 2012. 27

[Zha10a] ZHANG QI, CHENG LU ET BOUTABA RAOUF : Cloud computing : state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. 26

[Zha10b] ZHANG SHUAI, ZHANG SHUFEN, CHEN XUEBIN ET HUO XIUZHEN : Cloud computing research and development trend. *In Second International Conference on Future Networks, 2010. ICFN'10.*, pages 93–97. IEEE, 2010. 12

[Zha10c] ZHANG WEI, QIAN HANGWEI, WILLS CRAIG E ET RABINOVICH MICHAEL : Agile resource management in a virtualized data center. *In Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering*, pages 129–140. ACM, 2010. 58

[Zha10d] ZHAO GANSEN, RONG CHUNMING, JAATUN MARTIN GILJE ET SANDNES FRODE EIKA : Deployment models : Towards eliminating security concerns from cloud computing. *In International Conference on High Performance Computing and Simulation (HPCS)*, pages 189–195. IEEE, 2010. 2

[Zho11] ZHOU JIANTAO, ZHENG SHANG, JING DELIN ET YANG HONGJI : An approach of creative application evolution on cloud computing platform. *In Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 54–58. ACM, 2011. 9

[Zik11] ZIKOPOULOS PAUL ET EATON CHRIS : *Understanding big data : Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011. 144