# Création et autogestion de services pour l'Internet du Futur et le Cloud Computing

Tatiana Aubonnet

UPMC
SORBONNE UNIVERSITÉS

## Habilitation à diriger des recherches

Spécialité : Informatique

**Tatiana Aubonnet**

# Création et autogestion de services pour l'Internet du Futur et le Cloud Computing

**Soutenue le 1 juin 2015 devant le jury composé de**

| | |
|---|---|
| Guy Pujolle | Président |
| Jean Bézivin | Rapporteur |
| Omar Cherkaoui | Rapporteur |
| Michelle Sibilla | Rapporteur |
| Alfredo Goldman | Examinateur |
| Ahmed Serhrouchni | Examinateur |
| Noëmie Simoni | Examinateur |

Une *Fractale* désigne des objets dont la structure est invariante en changeant d'échelle.

L'ensemble de Mandelbrot peut s'écrire :
$(c, c^2 + c, (c^2+c)^2 + c, ((c^2+c)^2+c)^2 + c, (((c^2+c)^2+c)^2+c)^2 + c, ...)$

*à mes parents et Armand, à mon fils, à Joséphine . . .*

# Remerciements

Mes remerciements vont chaleureusement à Noëmie Simoni, qui m'a associé à ses thèmes de recherches, à une mouvance qu'elle à créée dans le domaine de la gestion de réseaux et de services, à la conférence GRES qu'elle a lancée et aux projets de recherche, notamment le dernier, OpenCloudware.

Je remercie sincèrement Ahmed Serhrouchni pour son aide, sa présence et son travail pour mettre en place la conférence GRES'2014 (*Gestion de Réseaux Et de Services*), ainsi que les échanges enrichissants dans le co-encadrement de thèse.

Je tiens à exprimer toute ma gratitude à Mr Guy Pujolle qui a accepté de présider ce jury.

Je remercie sincèrement les trois rapporteurs de ce manuscrit Mme Michelle Sibilla, Mr Jean Bézivin et Mr Omar Cherkaoui. Malgré les imprévus de la vie et leurs nombreuses occupations, ils ont accepté de rapporter mon travail.

Mes remerciements vont également à Mr Alfredo Goldman qui a accepté d'examiner mon travail.

Un grand merci aux partenaires du projet OpenCloudware Eric Madelaine et Ludovic Henrio (INRIA, équipe Oasis) et Fabienne Boyer (Université Joseph-Fourier-Grenoble) pour la collaboration et les discussions que nous avons eues, ainsi que pour les résultats obtenus lors de cette collaboration.

Je suis reconnaissante à Anne Wei et Eric Gressier de m'avoir accueilli dans l'équipe SEMpIA (*Systèmes Embarqués et Mobiles pour l'Intelligence Ambiante*), dont le thème est le plus proche de mes travaux de recherche.

Je remercie le groupe AIRS (*Administration et Ingénierie de Réseau et de Service*) de Télécom ParisTech, notamment Inès Ayadi et Soumia Kessal, ainsi que l'équipe de recherche SEMpIA du CNAM pour tous les échanges fructueux.

Je suis reconnaissante à Frédéric Lemoine pour son soutien précieux et sa participation dans l'article pour la revue JISA (*Springer Journal of Internet Services and Applications*).

Mes remerciements vont aussi aux doctorants et à tous les étudiants, notamment du Master ATOMS, qui ont apporté leur brique à ce manuscrit.

Enfin, mon grand merci à Michel Cotten pour son enthousiasme inextinguible pour ce travail.

# Résumé

Mes recherches couvrent l'ingénierie des services et sont orientées aujourd'hui vers la conception, le déploiement et la gestion des services pour l'Internet du futur et le Cloud Computing. Les services étant vus sous leurs aspects fonctionnels et non fonctionnels (qualité de services). Les résultats sont liés à mes travaux de post-doctorat à Alcatel (projet NGNSM), aux contributions du projet ANR SELKIS et surtout aux avancées obtenues dans le cadre du projet FSN OpenCloudware.

Ce manuscrit présente une partie de mes travaux de recherche sur la création et l'autogestion de services. Ces dernières années, les modèles à composants ont permis des avancées significatives dans la programmation. Cependant, aujourd'hui dans les architectures de cloud computing, nous sommes confrontés à un problème de gestion dynamique et proactive. Le caractère distribué et évolutif de l'environnement d'exécution du cloud et de l'Internet du futur, le nombre de composants logiciels et les propriétés de qualité de service attendues sont autant de critères de complexité qui impactent la phase de gestion de services. L'un des verrous majeurs pour l'ingénierie de service est la *gestion dynamique et autonome* des applications, nécessitant l'instrumentation des applications pour des actions appropriées lors de l'exploitation. Nous proposons des composants de service auto-contrôlables pour favoriser l'automatisation du déploiement, la reconfiguration dynamique du système et sa gestion lors de l'exploitation.

L'intégration de la gestion dès la création du service est le fil conducteur de ce manuscrit, dans lequel nous abordons dans un premier temps les capacités de gestion dynamique apportées par un *modèle à composants de service autocontrôlé ("self-controlled ")*, puis dans un deuxième temps les améliorations possibles à travers la *composition de services, l'ubiquité et une gestion autonome*. Un atelier de création rapide de services est envisagé pour permettre d'établir et de piloter une session de service personnalisée dans le cloud et l'Internet du futur.

Pour compléter notre contribution aux aspects non fonctionnels, un modèle générique de SLA (*Service Level Agreement*) introduit une cohérence entre les exigences de l'utilisateur SLO (*Service Level Objective*) et les offres de fournisseurs.

Mots-clés : *composant de service "self-controlled", composition de services, ubiquité, SLA.*

# Abstract

My research area covers engineering services and is presently oriented towards design, deployment and services management in the Future Internet and Cloud Computing. Services are considered here in their functional and non-functional aspects (Quality of Service). The present results are related to my post-doctorate work at Alcatel R&I (NGNSM project) and to my later work at the CNAM (Conservatoire National des Arts et Metiers) on the SELKIS ANR project, but mainly to contributions obtained in the FSN OpenCloudware project.

This manuscript presents the part of my research work dealing with services creation and self-management. In recent years, components models have enabled significant advances in programming. However, today in the Cloud Computing architectures, we are faced with a problem of dynamic and proactive management. The distributed and evolving nature of the execution environment in Cloud and Future Internet, the number of software components and the expected quality of service are the main criteria of complexity which impact the service management phase. One of the major technological challenges for service engineering is dynamic and autonomic management, requiring instrumentation of applications for appropriate actions during the operation phase. We propose self-controlled service components for deployment automation, dynamic reconfiguration of the system, and its management during the operation phase.

The integration of management starting with the service creation is the guiding principle of this document, in which we first, discuss the dynamic management capabilities provided by a self-controlled service component model and next, the possible improvements through the services composition, ubiquity and autonomic management. A workbench architecture for service creation is proposed to enable the establishing and piloting of a personalized service session in the Cloud and the Future Internet.

To complement our contribution to the non-functional aspects, a SLA (*Service Level Agreement*) generic model introduces consistency between the user requirements SLO (*Service Level Objective*) and the offers from providers.

# Table des matières

# Table des figures

# Liste des Abreviations

| | |
|---|---|
| ADL | Architecture Description Language |
| DTD | Data Type Definition |
| ETSI | European Telecommunication Standards Institute |
| GCM | Grid Component Model |
| IaaS | Infrastructure-as-a-Service |
| ITIL | Information Technology Infrastructure Library |
| MaaS | Monitor as a Service |
| MAPE | Monitoring, Analyse, Planning, Execution |
| NaaS | Network as a Service |
| NIST | National Institute of Standards and Technology |
| OMG | Object management Group |
| PaaS | Platform-as-a-Service |
| QoS | Quality of Service |
| SaaS | Software-as-a-Service |
| SCA | Service Component Architecture |
| SCC | Self-Controlled service Component |
| SLA | Service Level Agreement |
| SLO | Service Level Objective |
| SOA | Service Oriented Architecture |
| VCE | VerCors Component Editor |
| VPCN | Virtual Private Connectivity Network |
| VPEN | Virtual Private Equipment Network |
| VPSN | Virtual Private Service Network |
| VPUN | Virtual Private User Network |
| VSC | Virtual Service Community |

# Chapitre 1

# Introduction

La création et la gestion de services dans un environnement ouvert tel que l'Internet du Futur, c'est-à-dire Internet de services [1], [2] ou un environnement de Cloud Computing [3], [4] sont un enjeu économique majeur de nos jours. L'élaboration d'une *architecture* est donc nécessaire à la création rapide de services dans un environnement multi-acteurs, permettant l'évolution et la gestion de services, indépendamment des infrastructures. Ce défi technologique a été relevé par différentes instances de normalisation NIST (*National Institute of Standards and Technology*), ETSI (*l'European Telecommunication Standards Institute*), OMG (*Object management Group*), des programmes et projets européens [5].

Dans cet environnement complexe, les fournisseurs de service doivent normaliser et automatiser les processus de délivrance de ces services à travers leur cycle de vie. Les attentes en termes de qualité de service impactent toutes les phases du cycle de vie de services. Ce cycle est principalement formé des phases suivantes :

- La phase de conception, qui comprend la définition de l'architecture d'une application sous la forme d'un assemblage de composants logiciels associés à des propriétés de qualité de service différenciée ;
- La phase de développement, durant laquelle les composants logiciels, qui constituent les briques de base de l'application, sont mis en œuvre et assemblés selon l'architecture préalablement définie ;
- La phase de déploiement, généralement pilotée par des administrateurs humains, qui déploient et exécutent l'application dans un environnement d'exécution donné ;
- La *phase de délivrance (service delivery)*, correspondant à la délivrance du service personnalisé de l'utilisateur selon le SLA (*Service Level Agreement*) contracté. Elle doit assurer un ensemble de mécanismes pour contrôler et gérer la QoS (*Quality of Service*) au niveau service, offrant une automatisation de la gestion du niveau service. La délivrance des services tient compte du profil de l'utilisateur et du contexte ;
- La phase de gestion, durant laquelle l'application en cours d'exécution est supervisée pour satisfaire au mieux les besoins des utilisateurs.

Pour mieux maîtriser la complexité inhérente liée à chacune des phases de cycle de vie, nous nous intéressons à la modélisation des composants de service et à la gestion intégrée dès la phase de conception.

## 1.1 Contexte

Pour mieux comprendre les attentes en matière de création et de gestion de services, il convient de les situer dans l'architecture de l'Internet du Futur ou du Cloud Computing. Ce dernier introduit les notions de *"as a service"* et de virtualisation (aujourd'hui au niveau de l'infrastructure de réseau [6]).

Les modèles *as a service* se déclinent en services distincts. Les plus représentatifs dans le Cloud d'aujourd'hui sont : IaaS (*Infrastructure as a Service*), PaaS (*Platform as a Service*), SaaS (*Software as a Service*).

IaaS offre la possibilité à un opérateur de provisionner des ressources pour le traitement de l'information, le stockage et le réseau, ainsi que d'autres ressources fondamentales pour le traitement de l'information qui permettent à l'utilisateur de déployer et d'exécuter n'importe quel logiciel qui peut inclure un système d'exploitation et des applications.

PaaS permet à un programmeur de déployer, acquérir ou développer une application dans une infrastructure middleware virtualisée en utilisant un ou plusieurs langages de programmation et des outils fournis par le fournisseur de service.

SaaS offre la possibilité à un utilisateur final de pouvoir utiliser l'application d'un éditeur, typiquement accessible à partir d'un navigateur web, qui s'exécute dans un environnement Cloud.

La virtualisation, outre le fait qu'elle permet l'optimisation des ressources, permet surtout d'assurer le maintien de la cohérence du modèle pendent son exécution (runtime) par rapport à l'état dynamique d'une application [7].

L'Internet du Futur, celui des objets communicants, introduit la capacité de construire "un réseau de réseaux" assurant des sessions personnalisées et dynamiques des utilisateurs.

Pour répondre à ces nouveaux défis, nous avons besoin de repenser les services. Pour ce faire, nous avons travaillé sur plusieurs axes :

1. L'intégration des aspects fonctionnels et non-fonctionnels (Think) ;
2. L'intégration de l'instrumentation (monitoring) dès la phase de conception pour avoir des actions de gestion au plus près de chaque service lors de l'exploitation [8] ;
3. Une Architecture Orientée Services (SOA) [9] permettant la composition de service facilitant le couplage lâche des éléments (Build) ;
4. Une gestion [10], [11] à plusieurs niveaux qui se décline selon le temps de réaction (Run), soit
   - réaction dynamique : remplacement d'un composant ;
   - réaction tactique : boucle MAPE (Monitoring, Analyse, Planning, Execution) au niveau de la composition de service.

## 1.2 Évolution des approches

Le besoin de développement des services distribués de façon rapide, flexible et réutilisable a conduit à l'évolution des approches présentées dans la figure 1.1.

*L'approche orientée objet* (figure 1.1 [1]) a permis de concevoir des applications modulaires à travers des objets communicants et distribués [12]. Cependant, les notions d'héritage et de polymorphisme produisent un couplage fort entre les objets puisque la référence d'un objet fait

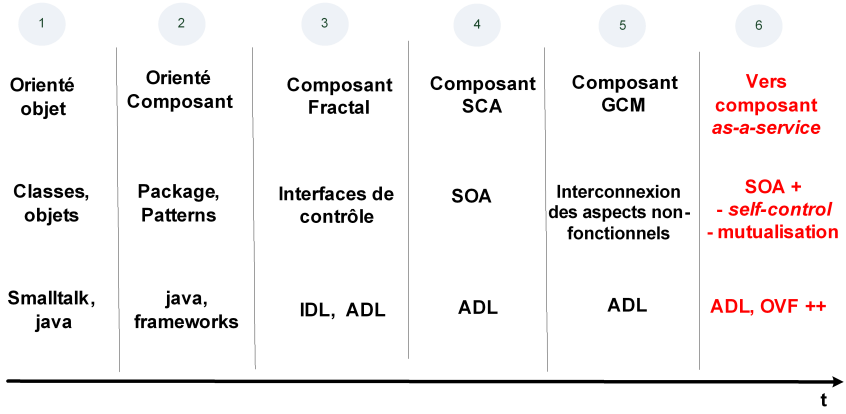



FIGURE 1.1 – Evolution des approches

appel à d'autres références d'autres objets. Les objets ont aussi plusieurs traitements (méthodes) qui s'exécutent sur les données (attributs). Ils sont donc difficilement partageables.

Pour dépasser ces limites, *l'approche orientée composant* (figure 1.1 [2]) est apparue avec l'objectif de pouvoir réutiliser des briques logicielles préexistantes lors du développement des applications réparties [13]. D'une part, le développement de briques logicielles réutilisables a été formalisé et simplifié. D'autre part, l'assemblage de ces briques en un système opérationnel ainsi que l'administration de ce dernier ont été organisés autour du concept d'architecture logicielle. Plusieurs définitions existent pour la notion de composant. Szyperski [14] définit le composant comme suit : "un composant logiciel est une unité de composition avec des interfaces spécifiées contractuellement et des dépendances de contexte explicites. Un composant peut être déployé indépendamment et être composé par des tiers pour former des applications ou des composants composites". Un composant peut être "instancié" et les composants peuvent être assemblés de manière hiérarchique. Par rapport aux objets, un composant définit explicitement ses dépendances, permettant de répondre aux besoins de configuration logicielle et de déploiement. Ainsi, un système à composants définit les entités architecturales (composants) et les relations entre ces entités (liaisons).

Il existe plusieurs modèles à composants. Notre étude porte sur les composants permettant un assemblage, reflétant au mieux une application distribuée. Ce sont les composants Fractal [15], SCA (*Service Component Architecture*) [16], GCM (*Grid Component Model*) [17].

*Le modèle à composant Fractal* (figure 1.1 [3]) a été développé par France Telecom R&D et l'INRIA en 2004. Le modèle Fractal propose une large gamme de contrôles applicables aux composants, allant d'aucun contrôle (boite noire d'objet classique) jusqu'à un contrôle incluant la manipulation du contenu du composant. Les aspects dynamiques dans Fractal sont assurés par la possibilité d'ajouter ou de retirer des composants à une architecture déjà déployée, grâce à la reconfiguration des liaisons et à l'introspection des composants composites.

Décrit en 2005, *le modèle à composant SCA* (figure 1.1 [4]) est un modèle permettant de construire des applications qui utilisent une architecture orientée services SOA (*Service Oriented*

*Architecture*). L'approche orientée service vise à apporter une grande flexibilité au développement des applications. L'idée générale est de créer des applications modulaires à liens lâches permettant d'adapter le développement des applications aux besoins [18]. S'appuyant sur une architecture SOA, le service représente l'unité d'abstraction pour le développement des applications. Trois principaux avantages sont à mettre à son actif :

- La description des services qui est réalisée de façon formelle dans les contrats de services. Cette description inclue l'interface de service, ses caractéristiques ainsi que ses opérations. En plus, elle peut spécifier des informations sémantiques pour expliciter le fonctionnement du service ;
- La composition qui est réalisée par la combinaison de services pouvant être soit basiques (élémentaires), soit eux mêmes composés. Cette composition permet de créer des nouveaux services répondant à des besoins plus complexes. La simplicité du processus de recomposition est due aux propriétés de couplage lâche et de réutilisation des services ;
- La découverte qui repose sur la publication des services. Des annuaires sont alors mis à la disposition des utilisateurs pour pouvoir choisir le service adéquat qui répond à leurs besoins.

*Le modèle à composant GCM* (figure 1.1 [5]) étend le modèle de composants Fractal. Dans le prolongement de Fractal, le GCM réutilise les fonctionnalités et les concepts de Fractal. Le composant GCM possède des capacités d'introspection intégrant les aspects non-fonctionnels (de gestion) et permettant de surveiller un système en cours d'exécution. Les objectifs principaux du modèle GCM sont "la séparation des fonctionnalités" et "l'autonomie", afin de concevoir, de déployer et de gérer le plus dynamiquement possible les systèmes logiciels complexes et distribués. La gestion de GCM se fait au travers de la boucle MAPE qui peut être intégrée dans chaque composant [19], [20], [21].

Bien que les capacités de reconfiguration dynamique et la gestion autonome fournies par les composants GCM soient significatives, la réponse apportée par ces solutions n'est pas suffisante et complète, à notre sens, pour l'Internet du Futur et le Cloud Computing. En effet, *les composants de service (as-a-service)* (figure 1.1 [6]) dans cet environnement ont besoin d'offrir à l'utilisateur qui va les sélectionner non seulement une fonctionnalité, mais un comportement (QoS) bien défini. Ce qui nous a conduit à proposer un auto-contrôle pour que le composant surveille sa conformité au contrat dans une architecture orientée services (SOA) permettant la composition de service à couplage lâche. Ce SCC (*Self Controlled service Component*) offrira une première réaction dynamique par substitution par un service "ubiquitaire" (équivalent en fonctionnalité et QoS), avant de déclencher la boucle MAPE qui agira au niveau de la composition de service.

C'est dans cette perspective que se situent mes travaux de recherche actuels.

## 1.3 Motivations

La préparation de l'HDR (*Habilitation à Diriger des Recherches*) marque une importante étape dans ma carrière d'enseignant-chercheur au CNAM longue de 10 ans, depuis l'obtention de mon doctorat de l'ENST en 2002 et mon post-doctorat à Alcatel R&D en 2003, qui sont à l'origine de mes motivations et de mes axes de recherche.

En effet, pendant ma thèse, je me suis intéressée à l'évolution des architectures pour le développement rapide de nouveaux services, avec l'intégration de la qualité de service dès la concep-

tion. Cet objectif n'a cessé de me guider dans les différentes mutations de notre environnement "réseaux et services".

Dans le Projet RNRT PILOTE (*Processus d'Ingénierie du LOgiciel de Télécommunications*), cadre de mes travaux de thèse ("Du réseau intelligent aux nouvelles générations de réseaux : création et qualité de service"), la problématique abordée relève des concepts de NGN (*Next Generation Networks*), avec un nouvel environnement de création de service où nous avions besoin, d'une qualité de service QoS (*Quality of Service*) différenciée [22], [23], [24].

Dans le projet Alcatel R&I NGNSM (*Next Generation Network and Service Management*), cadre de mon post-doctorat, j'ai poursuivi et fait évoluer mes *réflexions sur le plan de la gestion* [25], [26], [27], [28], [29]. Il m'a paru crucial de fournir une méthodologie pour concevoir et déployer les applications distribuées à base d'objets actifs et pro-actifs. Je me suis concentrée sur le modèle informationnel et architectural afin d'avoir une gestion dynamique de la qualité de service.

Mes avancées sur la gestion m'ont conduite à me focaliser sur la fonction de sécurité. C'est ainsi que, dans le projet SELKIS (*A development method of SEcure heaLth networKs Information Systems : from requirements engineering to implementation*), dont j'ai eu la responsabilité scientifique durant les années 2008-2012, j'ai introduit la modélisation de la sécurité dès la conception des applications [30], [31] en prenant en compte les propriétés de sécurité suivantes : disponibilité, intégrité, confidentialité et traçabilité de données.

Dans la thèse "Modeling security requirements in the early stages of the design of application" du projet SELKIS [32], [33], il a été possible d'intégrer les aspects fonctionnels et non-fonctionnels (sécuritaire) dès les premiers niveaux d'abstraction du cycle de vie de l'application : CIM (*Computational Independent Model*) et PIM (*Platform Independent Model*) de la démarche MDA (*Model Driven Architecture*). Ce qui m'a incité à exploiter l'intégration des deux aspects dans les modèles à composants.

Aujourd'hui mes recherches sont intégrées au laboratoire CEDRIC-CNAM. Mes travaux reposent sur une thèse[1] que je co-encadre, portant sur les aspects de modélisation de la sécurité en composants de services (identification, authentification, autorisation, firewall applicatif, chiffrement, vie privée *"privacy"*) [34], ainsi que sur le projet "OpenCloudware" [35], à travers une contribution d'un modèle avancé pour une plate-forme PaaS avec autocontrôle des services du Cloud Computing.

Par ailleurs, au niveau du rayonnement extérieur, j'ai participé à l'écriture d'un chapitre du livre "Des réseaux intelligents à la nouvelle génération de services", Hermès, 2007 [36] et j'ai eu à cœur de contribuer à porter les résultats obtenus sur la QoS sur le plan de la normalisation. C'est ainsi qu'en tant que membre de l'User Group de l'ETSI (*European Telecommunication Standards Institute*), les avancées de mes travaux sur la méthodologie pour l'identification des indicateurs de la QoS et sur la modélisation de SLA (*Service Level Agreement*) sont intégrées désormais dans trois normes européennes [37], [34], [38] une restant à valider. Ces travaux viennent compléter la vue globale de la QoS. En effet, le modèle SLA explicite et met en adéquation d'une part les exigences de l'utilisateur (Service Level Objective) et d'autre part les offres du fournisseur sous forme de services à travers le modèle et l'expression de la QoS.

1. Mayssa Jemel "Security and availability of client side storage for web application", thèse à TélécomParisTech

## 1.4 Objectifs et structure du document

Ce document présente les principes de création et de gestion de services à bases de composants, ainsi que les apports pour la mise en œuvre des services auto-gérables. Nous abordons les modèles à composants selon deux niveaux.

Tout d'abord, nous nous plaçons au niveau des utilisateurs des modèles à composants, à savoir les architectes d'une application, les programmeurs, et les administrateurs. Nous décrivons quels sont les paradigmes fournis par ces modèles, en accordant une attention particulière aux aspects suivants : les propriétés, la définition de composants non fonctionnels dans la membrane, la description des interfaces d'usage, de gestion et de contrôle. Ce que j'entends par la *membrane*, c'est une partie du composant (une enveloppe) qui nous permettra de réaliser sa gestion. Étant donné qu'elle est perméable, nous décrivons aussi les liens de communication entre la partie fonctionnelle du composant et sa membrane, partie non-fonctionnelle.

Le deuxième niveau que nous considérons est celui de la composition de services permettant de répondre aux sessions personnalisées. Pour assurer la prise en compte du SLA, nous considérons plusieurs niveaux de gestion, à savoir, celui au niveau des composants mutualisés selon une réaction dynamique, puis au niveau de la session selon une réaction tactique et finalement au niveau du système selon les processus FCAPS (*Fault, Configuration, Accounting, Performance, Security*) durant l'exploitation de services.

Ce document est organisé en sept chapitres. Nous commençons donc par la présentation des modèles à composants dans le chapitre 2, en exposant les principes des composants de service *"self-controlled"*. Notre contribution à cet axe de travail est présentée sous forme des avancées du projet OpenCloudware. Le chapitre 3 considère ensuite la composition de services et les différents modèles de réaction pour la gestion d'une session d'utilisateur. Le chapitre 4 présente une application Springoo fondée sur les composants SCC qui est spécifiée, vérifiée et validée sur la plate-forme VCE (*VerCors Component Editor*) et GCM/proactif. Dans le chapitre 5, c'est notre vision d'un atelier de création de services, pour le Cloud Computing et l'Internet du Futur, qui est présenté. Le chapitre 6 s'intéresse au SLA et montre comment la modélisation de composant de service autocontrôlé permet au fournisseur de proposer des contrats différenciés à travers une composition personnalisée. Ce document se termine par la présentation de nos conclusions et de nos perspectives de recherche dans le chapitre 7.

# Chapitre 2

# Vers un composant de service autocontrôlé (*"self-controlled"*)

L'évolution que nous venons de décrire dans le chapitre 1 (§ 1.2) montre les avancées apportées par chaque modèle. C'est ainsi que dans un premier temps, nous avons, dans le projet OpenCloudware travaillé sur le composant Fractal (§ 2.1). Les points forts, outre le composant fonctionnel et sa membrane, se situent aux niveaux des interfaces de gestion et de contrôle. Puis, pour enrichir la membrane du composant, nous avons investi le composant GCM, donnant la possibilité d'inclure des gestionnaires qui correspondent à la vision autonome (§ 2.2). Le *"as a service"* du Cloud nous a conduit à considérer le modèle SCA (§ 2.3) pour caractériser enfin le composant self-controlled proposé et décrire le comportement de nos composants avec le modèle générique de la QoS (§ 2.4). Puis, dans un deuxième temps, nous donnons une description des composants en langage ADL (§ 2.5). Cette description fournira une architecture pour intégrer les actions et les décisions relevant du contrôle et de la gestion.

## 2.1 Le modèle à composant Fractal

Fractal est un modèle à composant extensible et hiérarchique [39]. Chaque système Fractal peut retrouver à l'exécution son architecture (introspection), voire la modifier (intercession). Le modèle Fractal est également récursif (un composant peut être composé de sous-composants de façon hiérarchique à un niveau arbitraire) et réflexif (l'architecture est explicite et manipulable lors de l'exécution).

Fractal est fondé sur quatre notions : les composants, les interfaces, les liaisons et les contrôleurs [15]. Un composant est une entité exécutable. Fractal distingue deux types de composants : les composants primitifs et les composants composites. Les composants primitifs encapsulent une unité exécutable décrite dans un langage de programmation tandis que les composants composites sont des assemblages de composants. Une interface est un point d'accès ou de sortie sur un composant.

Le composant Fractal [40] est composé d'un contenu fonctionnel et d'une *membrane*, qui permet de réaliser l'introspection et la reconfiguration des fonctionnalités internes du composant.

Chaque composant Fractal implémente un certain nombre d'interfaces, dites serveurs, et peut être client d'autres composants via des interfaces clientes. Les interfaces sont décrites dans un langage de description d'interfaces (IDL) qui spécifie les méthodes d'une interface. Les interfaces

sont typées, par leur description IDL.

Un système Fractal est vu comme un assemblage de composants [41]. Étant donné que Fractal est un modèle hiérarchique, les composants sont à la fois en relation de parenté, mais également reliés via des liaisons de communication :

- La relation de parenté/sous-composants définit un graphe de composants ;
- Une liaison (*binding*) est un canal de communication entre une interface cliente d'un composant et une interface serveur d'un autre composant. Un appel (une communication) à une interface serveur est un appel d'une méthode de cette interface. Un composant Fractal est une entité à l'exécution, accessible par des interfaces bien définies.

Les interconnexions entre les instances sont représentées par des liaisons bidirectionnelles entre les composants Fractal correspondants. Cette bidirectionnalité facilite la navigation dans deux sens entre deux instances liées.

Notre première contribution dans le Fractal nous a permis d'explorer l'ensemble des propriétés de ce modèle. Il nous a paru important, dans un premier temps, de spécifier le composant QoS comme une extension de la DTD (*Data Type Definition*) Fractal, et dans un deuxième temps, d'illustrer l'utilisation des interfaces fournies par les composants et le déploiement des composants selon la QoS.

Afin de traiter les aspects comportementaux et dynamiques, nous avons introduit un composite QoS (QoSComposite Component) et son interface associée (QoSC), (figure 2.1).

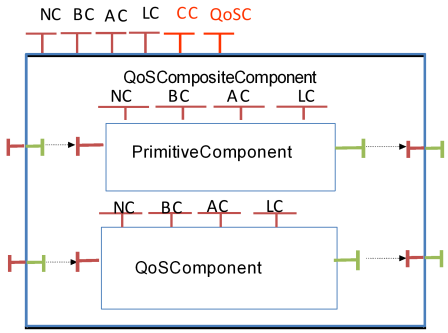



FIGURE 2.1 – Interfaces de composant "QoSCompositeComponent"

Le "QoS Component" peut avoir :

- *Le rôle actif* qui désigne l'état où le composant QoS joue le rôle de métrologue et de contrôleur de la QoS, et il notifie régulièrement, à qui de droit, le statut de QoS de son composant de service, c'est-à-dire s'il respecte toujours son contrat de service ou bien s'il est hors de contrat de service ("*In* contrat/*Out* contrat") ;
- *Le rôle passif* qui désigne l'état où le composant QoS assure uniquement le traitement interne. Il mesure la QoS et met à jour les valeurs courantes de QoS. Dans le cas de non-respect du contrat de QoS, c'est-à-dire le dépassement des valeurs seuils, il ne communique pas son état à son environnement sauf dans le cas où il est sollicité.

Les autres interfaces retenues sont :

- AC (*AttributeController*) : permet de modifier les attributs d'un composant ;
- BC (*BindingController*) : permet de créer/rompre une liaison primitive entre deux interfaces de composants ;

- LC (*LifeCycleController*) : permet de gérer le cycle de vie (minimaliste), représenté par deux états started/stopped, ajout/retrait de composants d'une architecture en cours d'exécution ;
- NC (*NamingController*) : permet de faire le *get* et le *set* du nom de l'élément ;
- CC (*ContentController*) : permet de lister, d'ajouter et de retirer des sous-composants qui le composent (introspection).

Cette extension au modèle Fractal nous a permis d'associer la qualité de service aux composants métiers (PrimitiveComponent) d'une application (figure 2.1). Cette extension a été conçue pour être indépendante et générique.

Dans notre approche, nous appliquons le méta-modèle de QoS (§ 2.5) au composant et son interface communiquera les notifications *In* ou *Out* contrat. C'est la présence de cette interface qui en fait un composant contrôlant la QoS. Le composant Fractal standard fournit des interfaces de contrôle pour permettre de le reconfigurer et donc de l'adapter. Ce composite QoS intègre, grâce à son interface les informations permettant de gérer le fonctionnement du système conformément à sa conception. Le composite QoS contient également les interfaces standards telles que AC, BC, LC, NC et CC.

Nous avons proposé une spécification de "QoSCompositeComponent" à l'aide de la grammaire DTD. Elle a été validée à l'aide de "Fractal ADL plug-in" dans l'IDE de l'Eclipse.

Selon notre modèle, le QoSCompositeComponent de la figure 2.1 est décrit par :

- Composant PrimitiveComponent ;
- Composant QoSComponent,
- Ses interfaces de contrôle standards qui sont NC, BC, AC, LC, CC et l'interface de contrôle spécifique à la QoS, qui est l'interface-QoSC.

  Cette partie de spécification est représentée dans la figure 2.2.

```
<!ELEMENT QoSCompositeComponent (PrimitiveComponent+, QoSComponent+,
NC+, BC+, AC+, LC+, CC+, interface-QoSC+)>

<!ELEMENT PrimitiveComponent (NC+, BC+, AC+, LC+)>

<!ATTLIST PrimitiveComponent

name CDATA #REQUIRED>
```

FIGURE 2.2 – QoSCompositeComponent

Comme mentionné précédemment, l'interface-QoSC est un nouveau contrôleur Fractal ajouté au QoSCompositeComponent (figure 2.3).

```
<!ELEMENT interface-QoSC (#PCDATA)>

<!ATTLIST interface-QoSC

name CDATA #REQUIRED

role (passive | active ) #REQUIRED

signature CDATA #REQUIRED >
```

FIGURE 2.3 – Interface-QoSC

Le composant QoS (QoSComponent) gère les aspects non-fonctionnels du composant de service. Cet élément est décrit par les deux notions telles que les quatre critères de QoS (QoSCriteria) et les paramètres de la QoS (QoSParameter), figure 2.4. Il possède les interfaces standards (NC, BC, AC, LC).

```
<!ELEMENT QoSComponent (QoSCriteria+, QoSParameter+, NC+, BC+, AC+, LC+)>
<!ATTLIST QoSComponent
name CDATA #REQUIRED
role (client | server )>
```

FIGURE 2.4 – Composant QoS

QoSCriteria se rapporte aux quatre critères de qualité de service (disponibilité, délai, capacité, fiabilité) du modèle générique de QoS que nous présenterons par la suite dans le paragraphe 2.4.3. Chaque critère s'évalue à partir de QoSParameter que sont les paramètres à mesurer. Notre modèle définit trois types de valeurs pour ces critères : conception, seuil et courante. La spécification de la DTD QoSCriteria est représentée dans la figure 2.5. Les critères sont évalués à travers les paramètres techniques (paramètres mesurables) de QoS.

```
<!ELEMENT QoSCriteria (#PCDATA)>
<!ATTLIST QoSCriteria
  Criteriatype (Availability | Delay | Capacity | Reliability)
  ValueType CDATA #REQUIRED
  roleValueType (Conception | Threshold | Current)>
<!ELEMENT QoSParameter (#PCDATA)>
<!ATTLIST QoSParameter
name CDATA #REQUIRED
value CDATA #REQUIRED>
```

FIGURE 2.5 – Critères de QoS et paramètres associés

L'étape suivante est l'intégration de cette spécification dans la section d'architecture de l'application (*AppArchitectureSection*) de l'OVF étendue (OVF++). Elle permet la description de QoS lié à chaque composant Fractal [42].

Pour illustrer l'utilisation des interfaces fournies par les composants et déploiement des composants selon la QoS, nous nous sommes focalisés sur un cas d'utilisation, une application Springoo, application web conforme à une architecture à trois niveaux de la plate-forme Java Enterprise Edition (JEE) : Apache, Jonas, MySQL.

Un composant Fractal implémente les fonctions de base de gestion de l'application à savoir : le déploiement, l'exécution à distance et le *wrapping* [43]. La fonction *wrapping* est associée à un patron d'utilisation mis en œuvre par un composant particulier appelé *wrapper*, qui définit comment utiliser le module encapsulé. Dans notre cas le *wrapper* est représenté par une classe

permettant le contrôle de composant. Chaque *wrapper* sera composé d'un objet Java qui définit les opérations de contrôle à effectuer sur le composant.

Dans notre exemple Springoo, nous avons un wrapper pour chaque composant, à savoir, *http-wrp, jee-wrp, ear-wrp et rar-wrp*. Puis, quatre *wrappers* ont été définis pour gérer le déploiement des éléments logiciels applicatifs selon la QoS. Le *http-wrp-QoS, le jee-wrp-QoS, ear-wrp-QoS et le rar-wrp-QoS* (figure 2.6) qui représentent respectivement les composants QoS gestionnaires du serveur frontal HTTP Apache, du serveur d'applications JONAS et des instances des applications EAR er RAR.

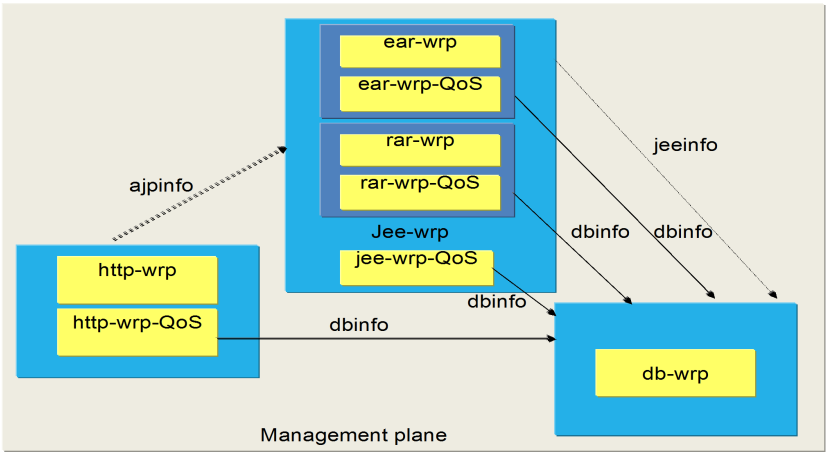



FIGURE 2.6 – QoS wrappers pour une application web

L'interface de contrôle "LifecycleController" sera utilisé pour démarrer ou arrêter le serveur Apache.

L'interface non-fonctionnelle *interface-QoSC* permettra d'envoyer les informations de la QoS comme les notifications ("*In/Out* contrat") ou les requêtes de monitoring vers la base de données (db-wrp).

Nous avons pu exploiter les aspects non-fonctionnels du modèle Fractal qui sont exposés comme les interfaces du composant. Néanmoins, il est impossible de se brancher sur ces interfaces, car elles doivent être invoquées directement, et il n'y a pas de support dans le modèle pour composer les aspects non-fonctionnels entre eux : tous sont implémentés directement au niveau du composant.

C'est pourquoi nous avons appréhendé le modèle à composant GCM pour avancer sur les deux aspects suivants :

- Définition d'un composant qui nous permettra d'avoir une structure pour les éléments non-fonctionnels de la membrane comme un assemblage de composants ;
- Construire un modèle permettant aussi l'interconnexion des aspects non-fonctionnels de façon similaire à ce qui est possible pour les aspects fonctionnels dans Fractal.

## 2.2 Le modèle à composant GCM (*Grid Component Model*)

Nous avons expérimenté le modèle à composant GCM [19], dans lequel nous souhaitons exploiter deux points importants : *"separation of concern"* [44] et une architecture favorisant "l'autonomie" [45].

Le premier point a été explicité pour la programmation dite par aspects, AOP (*Aspect Oriented Programming*) qui vise à permettre la séparation des différentes préoccupations *"separation of concern"*. Le but de l'AOP est de supprimer les dépendances entre code métier (fonctionnel) et code technique (non fonctionnel) pour isoler de façon encore plus poussée les modules. Ainsi l'AOP permet une nouvelle forme de réutilisation à travers une séparation des différentes préoccupations des programmeurs. Un jeu d'aspects peut se greffer sur le programme de base pour en modifier le comportement.

Étant donné que la QoS se focalise sur les aspects non-fonctionnels, nous présentons ces aspects dans la membrane du GCM. La structure de GCM nous permet de spécifier précisément les flux d'information non-fonctionnels. Nous pouvons d'ailleurs adapter cette structure plus finement pour le composant de QoS.

Le modèle à composants GCM est une extension de Fractal pour les systèmes distribués à large échelle [46], [47]. GCM a été proposé par le réseau d'excellence CoreGrid. Il hérite de Fractal toutes les capacités mentionnées dans le paragraphe 2.1. Les principaux ajouts sont :

- Des interfaces collectives, indispensables dans les grandes applications réparties, permettant des communications 1 vers N (multicast), N vers 1 (gathercast), ou N vers M entre composants ;
- Des mécanismes spécifiques pour la gestion des communications et des comportements répartis à base de requêtes ;
- Une meilleure séparation des aspects non-fonctionnels dans l'architecture des applications, avec l'introduction d'une membrane contenant des composants non-fonctionnels.

Nous sommes essentiellement intéressés ici par ce dernier point.

Dans le modèle à composant GCM [48], une structure est définie pour les éléments de la membrane : la partie non-fonctionnelle du composant peut ainsi être définie comme un assemblage de composants. Ces composants peuvent alors être connectés avec d'autres composants à l'intérieur de la même membrane ou avec des interfaces non-fonctionnelles d'autres composants.

La structure de GCM nous permet de spécifier précisément les flux d'information non-fonctionnels. Cette structuration des composants a été utilisée et spécialisée dans [49], [50] pour définir la gestion autonome des composants. Un composant peut avoir les connaissances et les règles qui lui permettent de prendre les décisions tout seul, pour remédier à un problème qui lui est propre ou qui relève de son domaine de responsabilité, et il doit envoyer des notifications à qui de droit [51].

Ceci correspond à un comportement "autonome", qui est représenté par un ou plusieurs composants supplémentaires dans la membrane, comme par exemple dans la figure 2.7. Dans cet exemple, nous avons la décomposition de la boucle autonome en 4 composants pour :

– Surveiller le comportement (Monitor) ;
– Analyser les données "monitorées" (Analyse) ;
– Identifier ou planifier les actions à mener (Planning) ;
– Exécuter les actions (Execute).

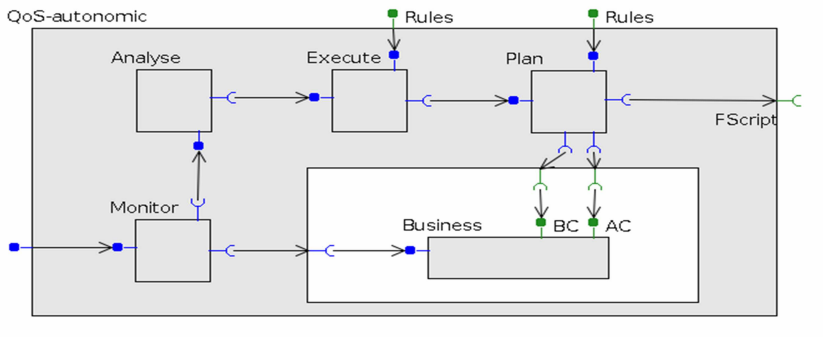



FIGURE 2.7 – Composant autonome

Le Planificateur et l'Exécuteur sont paramétrés par des règles qui peuvent être spécifiées (dynamiquement) par leurs interfaces non-fonctionnelles "Rules". L'exécuteur peut agir sur les contrôleurs standards (BC et AC) du composant métier, ou bien envoyer des commandes sur une interface de reconfiguration (ici "FScript").

Pour augmenter encore la décomposition structurelle de la membrane, et en conséquence augmenter la réutilisation des composants non-fonctionnels de QoS, il peut être intéressant de séparer ses fonctions internes, et de proposer une architecture qui sépare les fonctions de la QoS et de monitoring du reste des fonctions "dites de contrôle". Nous cherchons à spécifier ce modèle dans le projet OpenCloudware pour adresser les aspects comportementaux à travers la QoS. Avant de décrire ce modèle dans le § 2.4, nous avons aussi besoin de préciser les propriétés du composant fonctionnel. C'est pourquoi nous allons d'abord analyser les propriétés d'un composant SCA dans le § 2.3 suivant.

## 2.3 Le composant de service SCA (*Service Component Architecture)*

Une application SCA (*Service Component Architecture*) est composée d'un ou plusieurs composants. Quel que soit le langage d'écriture du composant ou le degré de distribution de l'application, SCA permet de définir les composants et de décrire comment ils interagissent [52].

SCA comme unité élémentaire de construction d'application s'appuie sur les propriétés suivantes :

– *Interconnexion* signifie que le service dispose des toutes les connectivités dont il a besoin ;
– *Autonomie* signifie qu'un service est capable de réaliser ses fonctionnalités sans avoir besoin d'une intervention humaine où d'un autre service. Nous proposons que le service soit une boite noire composée d'un ensemble d'opérations exécutées de la même façon et dans le même ordre pour toutes les demandes ;
– *Couplage faible* permet d'assurer la composition flexible des services, des liens lâches sont

maintenus entre les services composables pour éliminer tous types de dépendance fonctionnelle. La composition consiste à générer un service global composé d'un ensemble des services élémentaires. Cette composition est personnalisable et flexible en ajoutant, remplaçant, supprimant des éléments de services en fonction des besoins de l'utilisateur ;
- *Réutilisation* permet de simplifier le développement des services répondant aux nouveaux besoins : les services Cloud sont réutilisables. Grâce au caractère générique de ses interfaces (usage, contrôle et gestion), le service offre le même rendu indépendamment de l'environnement où il sera réutilisé.

Néanmoins, le modèle SCA n'est pas dynamique, car il ne permet pas, à l'exécution (*runtime*), l'ajout/retrait dynamique des composants et la modification des liaisons. Le fichier en langage SCDL (*Service Component Definition Language*) est conçu au moment de la conception et ne peut être modifié durant l'exécution de l'application. De ce fait, les modifications dans la composition de l'application à services nécessitent de stopper l'application et de la relancer pour utiliser une nouvelle composition. La reconfiguration dynamique et les autres aspects non-fonctionnels ne sont pas traités par les spécifications SCA. C'est à l'implémentation du support d'exécution de fournir de telles fonctionnalités. Par exemple la plate-forme FraSCAti [53] offre les contrôleurs Fractal apportant l'API d'accès à la boucle MAPE et permettant le déploiement de composants. La plate-forme ne permet cependant pas le chargement à chaud, ce qui interdit le design continu des composants.

Pour pallier ce manque de dynamique du composant SCA à l'exécution (*runtime*), nous proposons de compléter les propriétés préconisés par SOA [54], [55] que nous désignons par SOA+ (figure 1.1) pour aboutir au composant SCC (*Self-Controlled Component*).

En effet, Fractal/GCM/SCA nous apporte une forte structuration permettant l'expression de la composition, mais à ce stade, nous n'avons pas de *notion de contrat* qui pourrait aider à la gestion du SLA. Par ailleurs, les recommandations au niveau des SLA n'ont pas la *notion de composition de service.*

C'est pourquoi nous nous orientons vers un composant facilitant la conception et l'exécution de service autocontrôlé que nous décrivons dans le paragraphe ci-dessous et utilisable dans un SLA, que nous proposerons dans le chapitre 6.

## 2.4 Le composant de service *"self-controlled"*

Le Cloud Computing et l'Internet du Futur promettent un nouvel écosystème où tout serait sous forme de services selon une composition personnalisée et une gestion dynamique des ressources au moment de l'exécution. Les reconfigurations ne sont possibles que si le système est en mesure d'avoir et d'utiliser les informations pertinentes associées à une réaction dynamique au niveau de chaque ressource. Pour mettre en œuvre ces environnements dynamiques dans nos travaux du projet OpenCloudware [35], nous explorons les propriétés d'autocontrôle d'un composant de service (§ 2.4.1), un modèle dynamique et extensible. Pour assurer cette dynamique, nous allons définir un MaaS (*Monitoring as a Service*) (§ 2.4.2), un modèle à quatre critères génériques de QoS (§ 2.4.3) et les langages de description de contrat (§ 2.4.4).

### 2.4.1 Propriétés du SCC (*Self-Controlled service Component*)

Nous considérons dans ce paragraphe les propriétés supplémentaires qui nous paraissent indispensables à prendre dans le p*rocessus de modélisation de composant de service Cloud, à savoir le raffinement as-a-service* pour une clarification fonctionnelle afin de favoriser la composition. Ce raffinement permet de mieux appréhender le rendu du service et de pouvoir spécifier dés la conception son comportement (QoS).

Le composant fonctionnel du SCC (*Self-Controlled service Component*) [6] est représenté sur la figure 2.8.

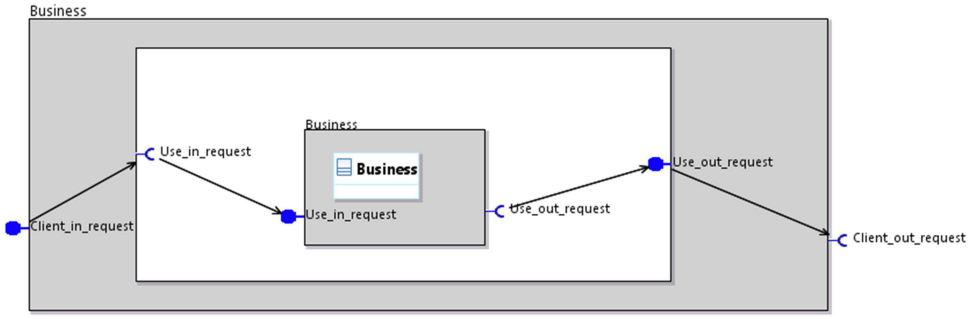



FIGURE 2.8 – Composant fonctionel

C'est un service qui, au delà des propriétés de SCA/GCM/SOA, doit avoir les capacités suivantes :

- *Sans état (stateless)* qui signifie que le service rend la même prestation à toutes les demandes sans aucune conservation de leurs données ou leurs contextes, permet de rendre le même service à tout le monde. Le composant de service aura une seule interface. Pour que le service soit sans état, ses opérations doivent être conçues pour effectuer les traitements sans dépendre des informations reçues au cours d'une invocation précédente ;
- *Mutualisation* qui signifie que le composant est un élément de service multi tenants, permet à plusieurs utilisateurs de l'invoquer. Cela renforce la propriété de "liens lâches" préconisée par SOA ;
- *L'ubiquité* qui définit des composants de service équivalents en fonctionnalité et en QoS, répartis sur une ou plusieurs plates-formes, favorise le choix des utilisateurs en fonction de leur localisation et de leurs préférences ;
- *Exposabilité* prévoit la description fonctionnelle et non fonctionnelle du composant de service afin que les utilisateurs puissent construire leur application ou leur plate-forme grâce à un catalogue ou un portail.

Ces propriétés permettront d'exposer les composants dans une bibliothèque (un catalogue), de mutualiser les composants pour les utiliser dans différentes applications, et de réaliser l'assemblage pour une session personnalisée.

Ce composant contient :

- Un contenu fonctionnel (business) ayant les propriétés que nous venons de définir ;
- Une interface d'usage correspondant aux interfaces externes (client et serveur) pour communiquer avec l'environnement ;
- Une membrane pour les aspects non-fonctionnels que nous allons décrire ci-dessous.

Comme nous l'avons constaté plus haut, l'autonomie du composant GCM repose sur la boucle MAPE [56], que nous pouvons mettre dans la membrane de chaque composant [57]. Mais nous venons de raffiner la fonctionnalité du composant pour le rendre *as a service* et du coup nous pouvons avoir une boucle MAPE simplifiée et générique. En effet, nous avons une interface d'usage unique et un rendu de service identique pour tous les utilisateurs, donc l'analyse se réduit à "être conforme ou non" par rapport au service que le composant est sensé rendre. C'est pourquoi nous proposons un composant de service SCC reposant sur le triptyque "*un moniteur en entrée*, *un moniteur en sortie* et *un composant de contrôle de conformité à un contrat QoS* qui traduit le comportement prévu à la conception et proposé dans l'offre"(figure 2.9). Ce qui nous conduit aussi à rajouter une propriété : QoS offerte qui permet de choisir un composant de service en fonction de sa fonctionnalité et de son comportement (QoS).

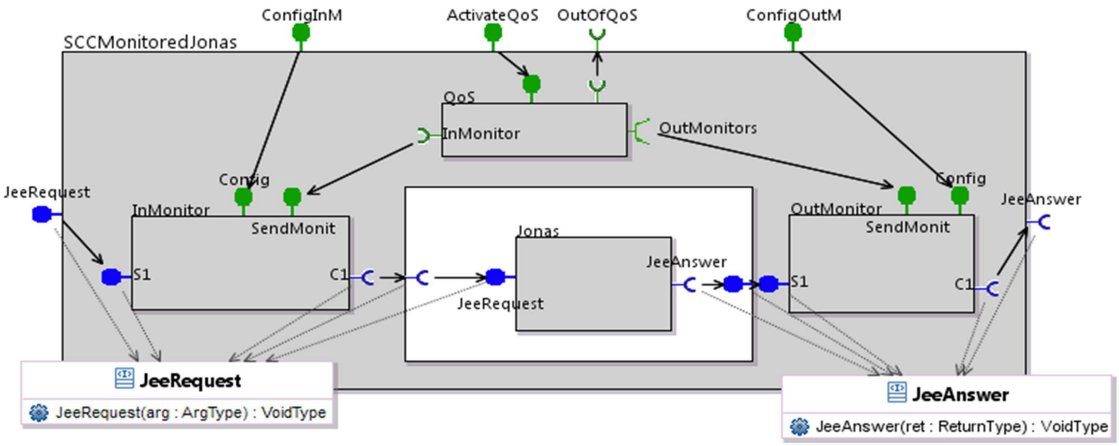



FIGURE 2.9 – Structure du SCC (*Self-Controlled service Component*)

La membrane du composant SCC contient (figure 2.9) :

- Un composant de monitoring en entrée (*InMonitor*) et un autre en sortie (*OutMonitor*). Il joue un rôle d'intercepteur. Les requêtes de service entrantes sont interceptées, puis transmises (inchangées) au contenu fonctionnel du composant, via les interfaces internes correspondantes. Celui de sortie intercepte les requêtes de service sortantes. Ils mettent à disposition les informations de mesure sur les interfaces où ils sont mis en coupure.
- Un composant QoS, associé au composant métier. Il est chargé de surveiller le respect du contrat de service ;
- Une interface non-fonctionnelle (cliente) de contrôle de QoS (*OutOfQoS*), par laquelle on enverra les indications de violation des contrats de QoS, c'est à dire les notifications de "*IN* contrat" tant que le comportement est conforme au contrat ou de "*Out* contrat" dans le cas contraire ;
- Une interface non-fonctionnelle (serveur) de configuration (Activate QoS, ConfigInM, ConfigOutM), dont le rôle est de recevoir des commandes de configuration du composant.

Nous obtenons ainsi un composant de service SCC (Self-Controlled service Component), composant s'auto surveillant et s'auto contrôlant [58]. Les sous-composants de la membrane (monitors et QoS) sont activés, afin d'effectuer le monitoring de la qualité de service et de notifier les cas de dégradation de qualité de service.

Ainsi le composant SCC possède 3 types d'interfaces :

- *L'interface d'usage* qui est une interface fonctionnelle (en bleu sur la figure 2.9). Elle comprend les fonctions de traitement métiers réalisées par le composant de service et offertes aux utilisateurs.
- *L'interface de gestion* qui est une interface non-fonctionnelle. Elle contient les mécanismes nécessaires pour gérer la configuration des composants non-fonctionnels dans la membrane.
- *L'interface de contrôle*, une interface non-fonctionnelle également, qui contient les mécanismes permettant l'autocontrôle du comportement du service. Elle vérifie si le composant de service remplit son contrat.

La logique de structuration de la membrane permet une grande réutilisation et le caractère générique de nos composants. Le triptyque (*IN Monitor*, *Out Monitor*, QoS) associé à chaque composant de service introduit une gestion homogène de composants de service, que nous exposons dans les paragraphes qui suivent.

### 2.4.2 Le composant monitoring

Dans le processus d'amélioration continue des services ITIL (*Information Technology Infrastructure Library*) rappelle qu'il faut remplir trois conditions de base pour pouvoir réaliser un management efficace [59] :

- "on ne peut gérer ce que l'on ne contrôle pas" ;
- "on ne peut contrôler ce que l'on ne mesure pas" ;
- "on ne peut mesurer ce que l'on n'a pas défini au départ".

Pour poursuivre notre approche vers l'autogestion, notre composant de service "self-controlled" a pour but de contrôler la conformité au contrat. Ce contrôle doit à son tour s'appuyer sur des mesures définies au départ, c'est à dire dès la phase de conception. D'où l'importance du composant de monitoring dans la membrane, que nous proposons de définir comme un composant *as a service*.

Les questions qu'on se pose alors pour le composant de "Monitoring as a Service" (MaaS) sont : où mesurer, quand mesurer, quoi mesurer et comment mesurer.

- Où mesurer ?
  Le MaaS se met en coupure en entrée et en sortie du composant fonctionnel. L'utilisation de deux composants de monitoring nous permettra d'avoir des valeurs numériques précises sur ce qui entre et sort du service. *InMonitor* et *OutMonitor* prendront les mesures mais ne feront ni l'analyse des métriques évaluées, ni la prise de décisions. C'est le composant de contrôle de QoS qui fera les calculs nécessaires pour évaluer le comportement du composant de service.
- Quand mesurer ?
  A chaque arrivée de requête et à chaque sortie de requête.
- Quoi mesurer ?
  Si nous nous penchons sur la seule finalité de la QoS, les mesures portent sur les paramètres qui permettront d'évaluer les critères de QoS (que nous définissons dans le paragraphe suivant. Mais en fait, si on veut un moniteur générique, on peut se poser la question de ce que nous pouvons mesurer aux emplacements désignés. Nous trouvons : le nombre de requêtes arrivant, le nombre de celles qui sont soit erronées, soit rejetées et le temps à l'arrivée de la requête.

- Comment mesurer ?
  Compte tenu des valeurs que nous venons de trouver, nous avons besoin de "Compteurs" et de pouvoir récupérer un temps système (timestamp). Dans le cas de surveillance du temps passé dans le service la sauvegarde de la date d'entrée et de sortie d'un paquet/requête sera nécessaire pour faire les calculs. Dans le cas de calcul de paquets/requêtes en rentrées ou en sorties (*IN/Out*) nous prévoyons :
- A chaque paquet rejeté augmenter la valeur Cpt.Rejetés := Cpt.Rejetés+1 ;
- A chaque paquet erroné augmenter la valeur Cpt.Eronnés := Cpt.Eronnées+1 ;
- A chaque paquet réussi augmenter la valeur Cpt.Réussi := Cpt.Réussi+1.

Le composant MaaS permettra de prendre les mesures nécessaires pour gérer les *In et Out* contrats par le composant QoS.

### 2.4.3 Le composant QoS

Pour décrire le comportement de nos composants et permettre une gestion homogène de la QoS, nous définissons un modèle de QoS générique [60], [61]. Ce modèle de QoS représente le comportement et les caractéristiques de chaque ressource et couvre l'aspect non-fonctionnel de cette dernière. Il s'appuie sur les propriétés de transparence de la fonction rendue par le service. En particulier, les transparences spatiale, temporelle et sémantique qui induisent quatre critères génériques définis ainsi :

- La disponibilité, qui définit le taux d'accessibilité d'un composant, c'est-à-dire le pourcentage de temps pendant lequel un composants peut accepter des demandes.
- La fiabilité, qui représente l'aptitude d'un composant de service à être exécuté sans altération des données.
- Le délai, qui représente le temps de traitement d'une requête pour un composant de service.
- La capacité, qui indique pour un composant de service la charge maximale.

Ces quatre critères de QoS sont génériques, nécessaires et jusqu'à présent sont suffisants. Ils s'évaluent à partir de différents paramètres mesurables qui varient selon le composant. Par exemple le critère de Capacité au niveau d'un équipement est représenté par sa mémoire et par la vitesse de traitement de son processeur, pour le cas d'une ressource réseau, elle correspond au débit et à la bande passante et pour un service au nombre de requête traiter.

Il faut noter que nous n'avons pas besoin de chercher une formulation qui combine ces quatre critères puisque toute variation d'un des critères de QoS d'une ressource traduit un autre type de comportement et donc définit un autre service. Par exemple pour HTTP les différents calculs que nous pouvons faire sont :

- HTTP Service Non-Accessibility,
- HTTP Setup Time [s],
- HTTP Data Transfer Cut-off Ratio[%],
- HTTP Data Transfer Cut-off Ratio [%]=(Incomplete Data transfers/successfully started data transfers).

Afin de permettre l'autogestion des ressources sélectionnées, nous définissons, pour chaque critère de QoS, trois types de valeurs :

- La QoS de conception composée des 4 valeurs correspondant au modèle générique et déterminées au moment de la conception du service. Elle définit les capacités maximales de traitement du composant de service ;

– La QoS courante ou la valeur de QoS instantanée surveillée en cours d'exploitation ;
– Les valeurs de QoS seuil qui sont les limites de QoS à ne pas dépasser pour que le composant assure son traitement normalement.

Le composant QoS vérifie si le comportement courant de la ressource est conforme ou non à celui négocié dans le SLA. Pour cela, il compare la valeur courante aux valeurs seuils à ne pas dépasser. C'est le composant QoS qui permettra d'envoyer les notifications *IN/Out* contrat dans l'idée de vérifier le comportement, par model-checking, un tel scénario correspond à une propriété de logique temporelle.

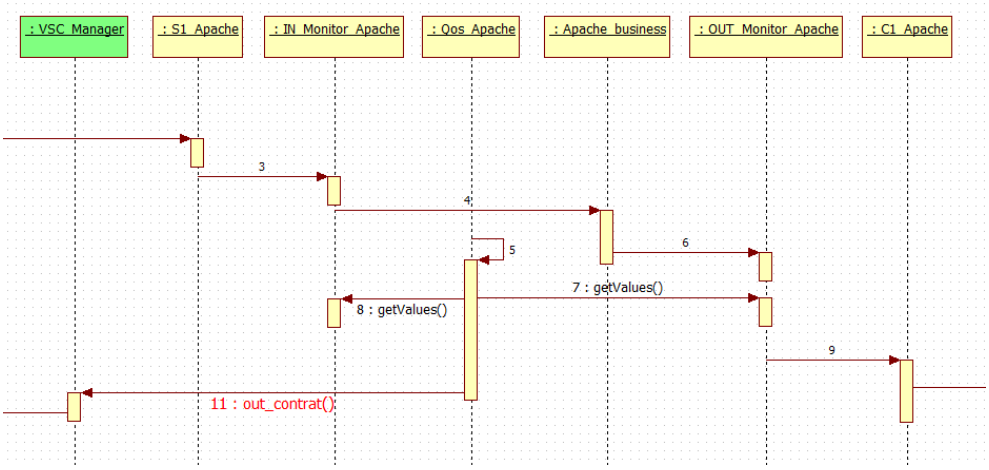



FIGURE 2.10 – QoS : Diagramme de séquences

Comme le montre le diagramme de séquence figure 2.10, le composant QoS déclenche un *timer* et demande aux moniteurs (*In Monitor* et *Out Monitor*) les valeurs de paramètres de quatre critères. Il envoie un "*out* contrat" si la valeur courante est inférieure ou supérieure à la valeur seuil (qui est en général une fourchette), afin de lancer par un manager de VSC extérieur (*VSC manager*), le processus de gestion dynamique de composant pour le remplacer par un service ubiquitaire et ainsi empêcher la coupure de la session. Sinon, il envoie un "*in*-contrat".

### 2.4.4 Langages de description de contrats

Le projet OpenCloudware nous a donné la possibilité d'utiliser plusieurs langages de description, à savoir :

– OCW-SLO, pour décrire le SLO (Service Level Objective) ;
– CSLA, pour la gestion de SLA dans le Cloud ;
– Fractal ADL, pour décrire le contrat QoS.

Dans un premier temps, dans le cadre du projet OpenCloudware, un langage nommé OCW-SLO a été défini et proposé [62] afin de décrire des objectifs SLO liés à l'élasticité d'une application déployée sur le Cloud. Le but était de définir de manière fine les objectifs que l'on souhaite contractualiser avec le fournisseur de service et de prendre en compte une certaine élasticité dans l'expression des objectifs.

Un méta-modèle a été défini pour décrire l'ensemble des concepts manipulés. Ce méta-modèle est décrit sous la forme de schémas XML (documents XSD-XML Schema Definition). Une première description définit les concepts de base d'un SLO, indépendamment d'un domaine particu-

lier. Il doit donc être étendu pour prendre en compte des objectifs spécifiques à un domaine, par exemple le Cloud, par le biais d'un second fichier XSD. Dans ce méta-modèle, chaque objectif (*SLO template*) à une expression temporelle. Cette dernière décrit la plage de temps pendant laquelle un objectif (un ensemble d'attributs qualité) est attendu. Cette expression temporelle est définie sous la forme d'une date (date) et d'une information horaire (*time*), toutes les deux optionnelles. La date spécifie la date de début (*from*) et la date de fin (*to*) de prise en compte de l'objectif. L'information horaire spécifie une heure de début (*from*), une heure de fin (*to*), mais également une fréquence (*frequency*) de répétition d'un objectif, par exemple tous les deux jours.

Dans un deuxième temps, dans le projet, nous avons analysé les points forts du langage Cloud Service Level Agreement (CSLA) [63], un langage pour améliorer la gestion de SLA dans le Cloud, en particulier la gestion des violations. Inspiré de SLA-SOI [64], WSLA [65]et WS-Agreement [66], CSLA propose de nouvelles propriétés directement au niveau langage ("Fuzziness", "Confidence" et "Penalty") pour éprouver l'élasticité du Cloud. "Fuzziness" définit une marge acceptable autour de la valeur de seuil d'un paramètre QoS, alors que la propriété "Confidence" définit un pourcentage de la conformité des clauses SLOs. En outre, CSLA comprend un modèle de pénalité qui permet à appliquer des sanctions en cas de violation en fonction de la durée. La conséquence directe d'un tel langage est qu'il permet au fournisseur de service de préciser ses stratégies de (re)configuration (par exemple ordonnancement, contrôle d'admission, allocation) pour arbitrer la gestion de ressources dans un contexte dynamique. La version actuelle du langage repose sur une syntaxe XML dont la grammaire est définie par des schémas XML. Cependant, CSLA peut être représenté par n'importe quel langage dédié pour n'importe quel service Cloud (SaaS, PaaS ou IaaS), éventuellement un langage graphique. Enfin, la spécification Fractal ADL de Composant QoS (QoSCompositeComponent) que nous avons présenté dans le paragraphe 2.1, nous a permis d'avoir une première contribution d'intégration du contrat QoS dans OVF (*Open Virtualization Format*) [67]. La Section Architecture d'Application de l'OVF étendue (OVF++) pourrait intégrer la description de QoS lié à chaque composant.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CompositeComponent SYSTEM "dtd_fractal.dtd" >
<CompositeComponent>
 <primitifComponent name="springoo-rar">
<QoSComponent name=" springoo-rar-QoS" role="server">
<QoSCriteria Criteriatype="Delay" ValueType="3" roleValueType="Current">QoSCriteria</QoSCriteria>
<QoSParameter QoSCriteriatype="Reliability" name="R1" pourcentageQoS="90" unit="%" value="89"> </QoSParameter>
<interface-QoSC name="QoS-rar" role="active" signature="void">interface QoSC</interface-QoSC>
</QoSComponent>
</CompositeComponent>
```

FIGURE 2.11 – OVF étendu intégrant le composant QoS

La figure 2.11 représente un exemple d'OVF++ du composant springoo-rar avec sa QoS intégrée [42].

## 2.5 ADL du composant SCC

Pour la spécification, la vérification et la validation des applications fondées sur SCC composants, dans le projet OpenCloudware, nous utilisons une plate-forme VCE de l'IRIA [68]. Une bibliothèque de composants intégrant les aspects non-fonctionnels (moniteurs et gestion de QoS) sera fournie, ces composants seront instanciés par l'utilisateur pour réaliser son architecture. Le VCE est alors en charge de vérifier la cohérence de cette architecture [69], et de produire un fichier ADL (*Architecture Description Language*) [70], qui sera inclus dans la spécification de l'application. Les figures 2.12, 2.12 représentent l'ADL du composant SCC de la figure 2.9.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath://org/objectweb/proactive/core/
component/adl/xml/proactive.dtd">
<!-- Automatically generated by Vercors, INRIA Sophia-Antipolis -->
<definition name="http-monitor-qos">
    <interface name="Use_in_business_interface" role="server" signature="interfaces.Use_business_interface"
interceptors="IN_Monitor.Interceptor"/>
    <interface name="Use_out_business_interface" role="client" signature="interfaces.Use_business_interface"
contingency="optional" interceptors="OUT_Monitor.Interceptor"/>
    <component name="http">
        <interface name="Use_in_business_interface" role="server" signature="interfaces.Use_business_interface"/>
        <interface name="Use_out_business_interface" role="client" signature="interfaces.Use_business_interface"
contingency="optional"/>
        <content class="classes.Http"/>
        <controller desc="primitive"/> </component>
    <binding client="this.Use_in_business_interface" server="http.Use_in_business_interface"/>
    <binding client="http.Use_out_business_interface" server="this.Use_out_business_interface"/>
    <controller desc="composite">
        <interface name="Managment_Configuration_QoS-controller" role="server"
signature="interfaces.Managment_Configuration_QoS"/>
        <interface name="Control_QoS-controller" role="client" signature="interfaces.Control_QoS"
contingency="optional"/>
        <interface name="Managment_Configuration_IN_Monitor-controller" role="server"
signature="interfaces.Managment_Configuration_Monitor"/>
        <interface name="Managment_Configuration_OUT_Monitor-controller" role="server"
signature="interfaces.Managment_Configuration_Monitor"/>
        <interface name="membrane-controller" role="server"
signature="org.objectweb.proactive.core.component.control.PAMembraneController"/>
        <interface name="interceptor-controller" role="server"
signature="org.objectweb.proactive.core.component.control.PAInterceptorController"/>
        <component name="IN_Monitor">
            <interface name="Use_monitor_interface" role="server" signature="interfaces.Use_monitor_interface"/>
            <interface name="Managment_Configuration_Monitor" role="server"
signature="interfaces.Managment_Configuration_Monitor"/>
            <interface name="Interceptor" role="server"
signature="org.objectweb.proactive.core.component.interception.Interceptor"/>
            <content class="classes.InMonitorHttp"/>
            <controller desc="primitive"/>
</component>
```

FIGURE 2.12 – ADL du composant SCC - partie 1

La description des applications réparties, telle qu'attendue dans la section (AppArchitectureSection) du format OVF étendu (OVF++), repose sur le langage de description d'architecture du modèle à composants Fractal et GCM, SCC. Le langage OVF++ permet de préciser la structure globale d'une application en termes de composants, de configuration et d'interconnexions. Chaque composant contient une information de localisation (virtual node), qui dans le cas présent, désigne la machine virtuelle sur laquelle le composant doit être embarqué. Ce langage fournit un niveau d'abstraction supérieur à des scripts de configuration spécifiques, ce qui permet d'automatiser les opérations de configuration de l'ensemble de la pile logicielle embarquée dans une machine virtuelle.

```
<component name="Qos">
            <interface name="Use_monitor_interface_IN" role="client" signature="interfaces.Use_monitor_interface"
contingency="optional"/>
            <interface name="Managment_Configuration_QoS" role="server"
signature="interfaces.Managment_Configuration_QoS"/>
            <interface name="Control_QoS" role="client" signature="interfaces.Control_QoS" contingency="optional"/>
            <interface name="Use_monitor_interface_OUT" role="client" signature="interfaces.Use_monitor_interface"
contingency="optional"/>
            <content class="classes.QosHttp"/>
            <controller desc="primitive"/> </component>
        <component name="OUT_Monitor
            <interface name="Use_monitor_interface" role="server" signature="interfaces.Use_monitor_interface"/>
            <interface name="Managment_Configuration_Monitor" role="server"
signature="interfaces.Managment_Configuration_Monitor"/>
            <interface name="Interceptor" role="server"
signature="org.objectweb.proactive.core.component.interception.Interceptor"/>
            <content class="classes.OutMonitorHttp"/>
            <controller desc="primitive"/> </component>
        <binding client="Qos.Control_QoS" server="this.Control_QoS-controller"/>
        <binding client="this.Managment_Configuration_IN_Monitor-controller"
server="IN_Monitor.Managment_Configuration_Monitor"/>
        <binding client="this.Managment_Configuration_QoS-controller" server="Qos.Managment_Configuration_QoS"/>
        <binding client="Qos.Use_monitor_interface_IN" server="IN_Monitor.Use_monitor_interface"/>
        <binding client="this.Managment_Configuration_OUT_Monitor-controller"
server="OUT_Monitor.Managment_Configuration_Monitor"/>
        <binding client="Qos.Use_monitor_interface_OUT" server="OUT_Monitor.Use_monitor_interface"/>
</controller>  </definition>
```

FIGURE 2.13 – ADL du composant SCC - partie 2

Le langage ADL présenté dans les figures 2.12, 2.13 et intégré dans la description OVF étendue, permettra de décrire l'application Springoo fondée sur les composants SCC. Une adaptation des *bindings* entre la partie ADL et les parties "virtual node", puis les ressources physiques, sera éventuellement nécessaire pour mener à bien cette intégration sur la plate-forme (PaaS) d'OpenCloudware.

# Chapitre 3

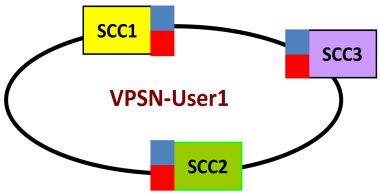
# Gestion de la composition de service

La composition de service constitue l'ensemble des composants de service invoqués dans une session d'utilisateur. Dans un contexte où ces composants sont des SCC, et afin d'utiliser les composants de service les mieux adaptés à la demande de l'utilisateur, en termes de fonctionnalité et de QoS, nous proposons de les présélectionner à l'établissement de sa session. La présélection (pré-provisioning) se fera selon la QoS demandée par l'utilisateur en adéquation avec la QoS offerte par les services. Cette composition de service constituera le VPSN (§ 3.1).

Une fois le VPSN constitué, en phase d'exploitation nous avons besoin de gérer le service rendu par la composition de service. Nous définissons trois types de réaction en fonction du niveau de décision : des décisions opérationnelles (§ 3.2), tactiques (§ 3.3) et stratégiques(§ 3.4).

## 3.1 La composition de services : le VPSN

Les composants de services étant de type SCC, à l'ouverture d'une session utilisateur, le VPSN constituera la composition personnalisée des SCCs pré-sélectionnés en fonction de la QoS demandée. Pour chaque VPSN créé, une table VPSN sera ajoutée à la base de connaissance [71] contenant le VPSN-ID, les SCCs-IDs et leur QoS offerte (tableau de la figure 3.1).




| VPSN-ID | | |
|---|---|---|
| Composition de services | QoS Offerte | @ |
| SCC1_ID | QoS_SCC1 | @ SCC1 |
| SCC2_ID | QoS_SCC2 | @ SCC2 |
| SCC3_ID | QoS_SCC3 | @ SCC3 |

FIGURE 3.1 – Creation de VPSN

Les SCC possèdent les propriétés préconisées, à savoir : sans état, ubiquité, QoS offerte, exposabilité et la possibilité de mutualisation. Donc, chaque composant SCC (partagé par différents utilisateurs) peut être attaché à différents VPSN. Par exemple dans la (figure 3.2), nous avons le composant SCC3 attaché au VPSN A, VPSN B et VPSN C [72]. La délivrance des services tient compte du profil de l'utilisateur et du contexte de l'utilisation de service.

Dans le paragraphe qui suit, nous allons nous intéresser à la gestion de composants en fonction des informations remontées par les différents moniteurs.

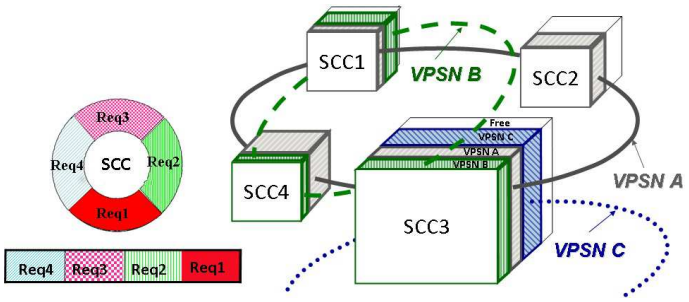



FIGURE 3.2 – Mutualisation de service dans VPSN.

## 3.2 Décisions opérationnelles : VSCs et SCCs sur le VPSN

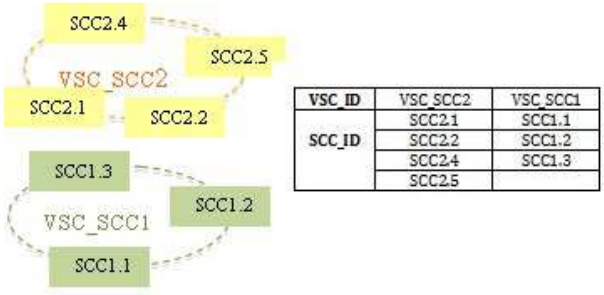
Le contrôle de QoS dans chaque composant de service SCC, nous permet de gérer les violations de contrat ("*Out* contrat") au moment où elles se produisent et ainsi de prendre les décisions opérationnelles adéquates. Nous préconisons un changement par un composant de service SCC ubiquitaire. Le remplacement des composants est géré par les VSCs [73]. Ces derniers regroupent les composants SCCs équivalents (ayant la même fonctionnalité et même QoS) par communautés.

Le concept de communautés d'intérêts (VSCs) nous permet de réagir dynamiquement à tout changement de contrat puis d'appliquer les changements dans le VPSN.

Nous décrivons d'abord la création de ces VSCs (§ 3.2.1), puis la réaction dynamique des VSCs et SCCs sur le VPSN (§ 3.2.2).

### 3.2.1 Création des VSCs (*Virtual Service Community)*

Les VSCs sont créés durant le déploiement des composants de service. Tout d'abord, chaque composant effectue une découverte de ses pairs ubiquitaires (même fonctionnalité et même QoS), puis souscrit à une communauté. S'il ne trouve pas de communauté qui réponde à sa fonctionnalité et QoS, il crée alors une nouvelle VSC, nouveau VSC-ID [74] et met à jour les tables VSCs (tableau 2 de la figure 3.3) dans la base de connaissance. Les VSCs peuvent être regroupés par localisations stratégiques, par opérateurs ou par plates-formes dans le but de répondre aux requêtes des utilisateurs.




| VSC_ID | VSC_SCC2 | VSC_SCC1 |
|---|---|---|
| SCC_ID | SCC2.1 | SCC1.1 |
| | SCC2.2 | SCC1.2 |
| | SCC2.4 | SCC1.3 |
| | SCC2.5 | |

FIGURE 3.3 – Processus d'ingénierie logiciel

Le projet UBIS (*User centric : uBiquité et Intégration des Services*) et les travaux de thèse de Houda Alaoui Soulimani [75] ont permis de développer un démonstrateur et montrer la faisabilité

d'intégration d'un agent de QoS et l'utilisation des communautés de service virtuelles (VSC). Le VSC a pour rôle de traiter la demande de remplacement d'un service par un autre ubiquitaire, pour maintenir le service avec le même temps de réponse suite à une dégradation de la QoS.

Après avoir présentés la phase de création des VSCs, nous détaillons la phase de gestion.

### 3.2.2 Réaction dynamique des VSCs et SCCs sur le VPSN

Durant l'exploitation, quand le SCC mutualisé envoie un "*Out* contrat", le VSC Management réagit à cette notification et remplace le composant SCC dans le VPSN concerné. Si le "*Out* contrat" fait suite à un taux d'erreur (fiabilité) dépassant le seuil alors le remplacement se fait dans tous les VPSNs auxquels il est attaché.

En effet le "*Out* contrat" résulte de différents évènements sur le service. Pour différencier les évènements, nous affectons au "*Out* contrat" un vecteur de quatre poids représentant chaque critère de QoS (disponibilité, fiabilité, délai et capacité). Il est nécessaire de préciser la cause d'un "*Out* contrat" et indiquer le critère (ou les critères) hors contrat. Selon les critères non respectés, le VSC manager sera en mesure de réagir en utilisant les règles spécifiques, et, par exemple, remplacer dans le VPSN le SCC concerné par un autre ubiquitaire (§ 3.2).

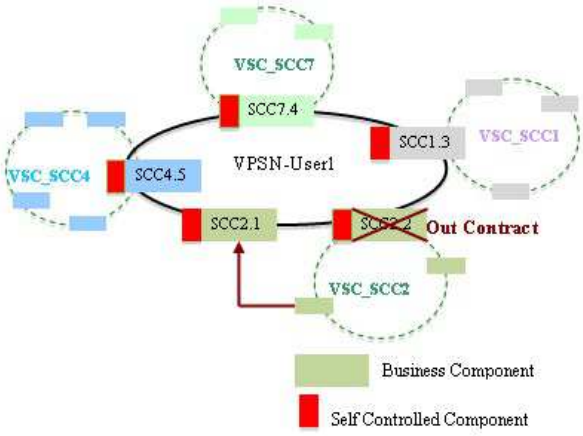



FIGURE 3.4 – Réaction dynamique de VSC et SCC sur le VPSN

C'est ce traitement qui se fait dans la phase de "délivrance" (*"service delivery"*) pour tenir compte du SLA de l'utilisateur, de sa mobilité, de ses préférences selon le type de service demandé. Nous avons besoin de cette dynamique afin de suivre l'usage de l'utilisateur à chaque instant.

Au dessus des décisions opérationnelles de gestion de la composition présentées, nous avons d'autres décisions de niveau tactique prises par la boucle MAPE que nous exposons dans la section suivante.

## 3.3 Décision Tactique : boucle MAPE

Dans le monde du Cloud et de la virtualisation, l'utilisateur ne paye que ce qu'il consomme. Ainsi, par rapport à la demande de l'utilisateur et sa QoS demandée (son SLO), nous avons des attributions de composants non partagés. Dans ce cas, la gestion de la composition consiste à

rajouter ou diminuer le nombre de composants dans la session selon la consommation. Ces décisions ne sont pas du niveau opérationnel. Elles dépendent des ressources disponibles et relèvent du niveau tactique. Elles sont définies en général selon les règles métiers contenues dans les bases de connaissances.

Dans cette architecture de gestion autonome à différents niveaux, la boucle MAPE s'occupera de gérer ces changements. Tout d'abord, l'Analyse va recevoir la notification du changement, puis le Planning va prendre la décision d'ajout ou de réduction de composants. L'Exécution se fera sur le VPSN. Par exemple, l'Analyse reçoit un "*Out* contrat" d'un composant réutilisable et non mutualisé avec variation de la "capacité" et/ou "du "délai", le Planning va consulter les règles métiers sur ce composant et ajouter un composant pour lequel il faudra aussi rajouter un *load balancer* pour partager la charge entre les deux composants. Dans le chapitre 4, avec le cas Springoo, nous allons montrer ce cas d'ajout de *load balancer* et, donc, des composants.

## 3.4 Décision stratégique

Au delà de ces actions, dans lesquelles nous trouverons toutes celles qui pourront être automatisées, nous avons les décisions de haut niveau, c'est à dire de niveau stratégique. C'est dans le modèle FCAPS (*Fault, Configuration, Accounting, Performance, Security*) que nous trouverons les règles de répartition entre ces trois types de décisions.

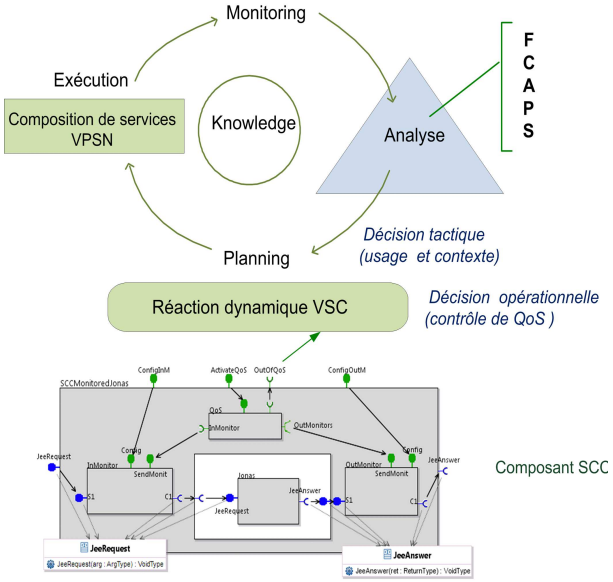



FIGURE 3.5 – Gestion de la composition

La figure 3.5 montre la complémentarité des actions de gestion : opérationnelle, tactique et stratégique. La décision opérationnelle correspond à réaction dynamique VSC. La décision tactique est réalisée par la boucle MAPE et basée sur l'utilisation et le contexte de service. La décision stratégique correspond à la gestion de FCAPS [76] permettant de gérer globalement les anomalies (fautes), les configurations, les performances et la sécurité.

# Chapitre 4

# Etude de cas Springoo

Springoo est une application web conforme à l'architecture trois-tiers de la plateforme JEE (*Java Enterprise Edition)* composée d'un serveur Apache, d'un serveur Jonas et d'une base de données MySQL.

Nous montrons dans ce chapitre les avantages de *self-controlled* et de la gestion de SLA de cette application. Pour implémenter Springoo, nos partenaires du projet OpenCloudware ont proposé un PaaS décrit comme une plate-forme ouverte et accessible aux architectes et développeurs du Cloud.

Nous allons appliquer le modèle défini précédemment dans le chapitre 2 au cas d'utilisation Springoo. Nos travaux nous ont permis de définir Springoo à partir de composants SCC *as a service* (triptyque Moniteur *In/Out* et QoS). Nous avons défini l'architecture décrite ci-après dans la figure 4.1.

L'ensemble des composants principaux de Springoo (Apache, Http, Https et Jonas) est défini à partir d'une rétro-modélisation. Ils sont presque tous de type *self-controlled*. Springoo est donc composé d'un ensemble de composants et de sous-composants de type *self-controlled*, comportant un monitor en entrée, un moniteur en sortie et un composant de contrôle de qualité de service (QoS). Chacun des composants s'auto surveillant et s'auto contrôlant.
Le composant ApacheHttp possède deux sous-composants : Http et Https également de type SCC. Ceux-ci permettent de traiter la requête en provenance du client en fonction du protocole utilisé (http et https).

La seule exception, concerne le composant Frontal qui est en attente du retour du traitement de Jonas. Cet élément est de type *statefull*. Le frontal permet de répondre au client sur une liaison bidirectionnelle. Les différents retours correspondent soit a des erreurs, par exemple lorsque la syntaxe de la requête est erronée (code 400), soit aux traitements en provenance de Jonas. La sortie de Jonas est en effet envoyée directement vers le frontal de Springoo.

La boucle MAPE a été intégrée dans la membrane du composant Springoo. Cinq composants internes définissent cette boucle :
- Un monitor en entrée/sortie analyse les requêtes du client et leurs réponses (*InOutMonitor*) ;

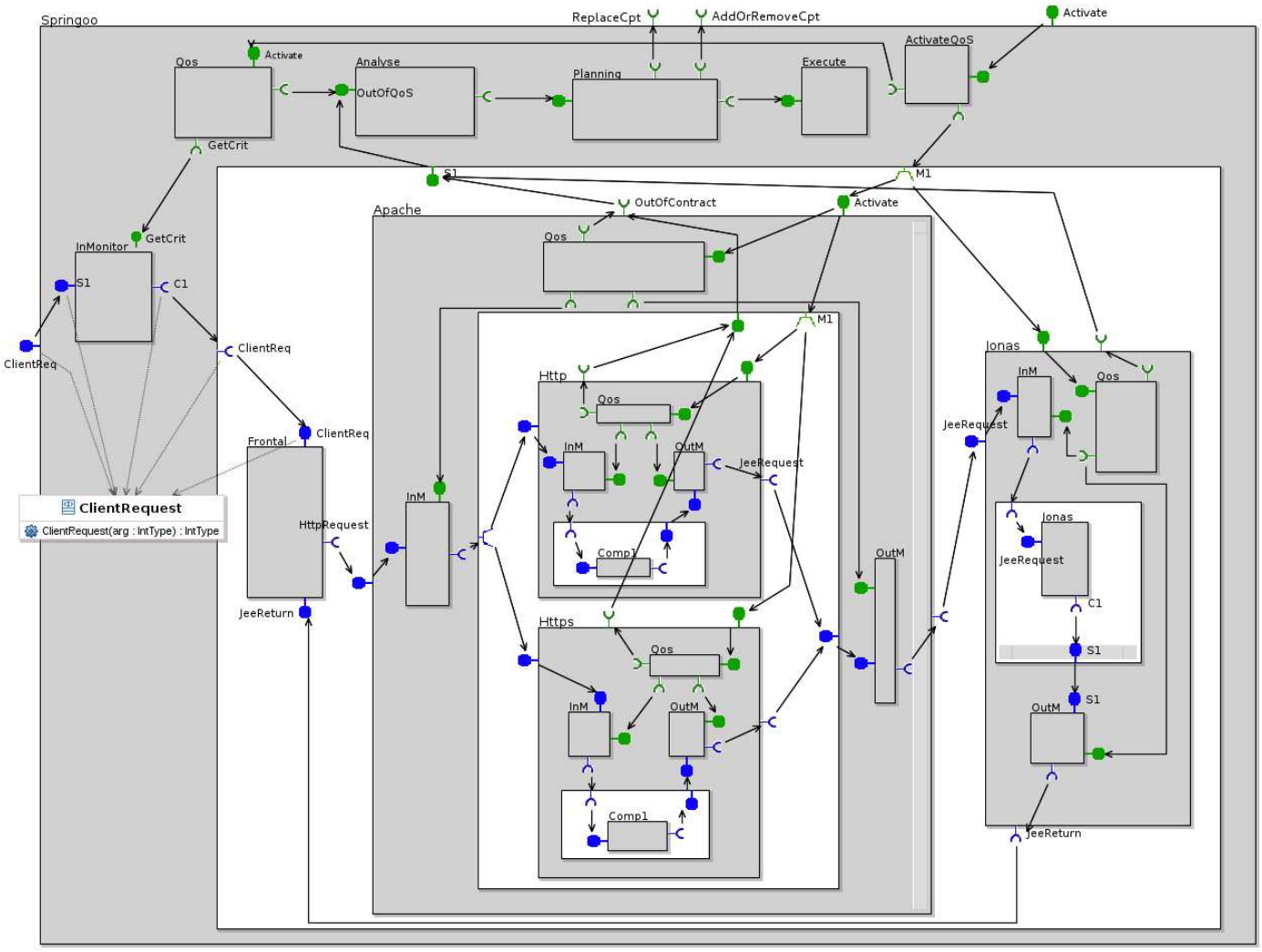



FIGURE 4.1 – Architecture *as a service* de Springoo

- Un composant d'analyse (Analyse) reçoit l'ensemble des indications de violation des contrats QoS des composants et sous-composants. Si c'est un SCC mutualisé, il transmet à l'extérieur (*VSC Manager)* pour être traité par la communauté VSC et que le composant soit remplacé. Si c'est un SCC non mutualisé c'est qu'il y a dépassement de capacité. Il surveille le respect du contrat de service à partir de l'ensemble des composants QoS de ses sous composants ;
- Un composant "Planning" est informé par le composant "Analyse" de toute violation de contrat et peut demander l'ajout d'un nouveau composant afin de répondre à une montée en charge si cela s'avère nécessaire (ressources insuffisantes par exemple) ;
- Un composant "Execute" chargé de réaliser l'ajout d'un nouveau composant (ex : Jonas) en cas de montée en charge. Pour cela, il active un *Load Balancer* (figure 4.2) de la nécessité de répartition de la charge entre les deux Jonas (dans cet exemple).

Dans cette boucle MAPE, nous avons ajouté deux composants liés à la QoS :

- Un composant "Activate" chargé d'activer ou de désactiver le contrôle de qualité de service de chaque composant interne (QoS) ;
- Un composant de QoS local chargé de surveiller le respect du contrat de service global à partir des requêtes clients. Il représente une partie de l'Analyse portant sur la conformité du contrat comme pour les autres composants, mais il est spécifique à une configuration donnée.

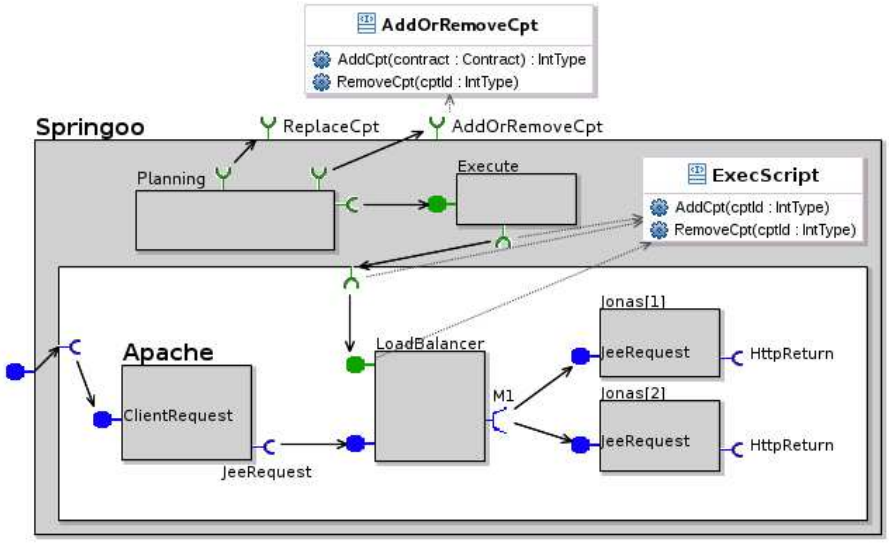



FIGURE 4.2 – Load Balancer

Les interfaces permettant de communiquer avec l'extérieur sont également définies, soit :

- Une interface d'usage d'entrée/sortie bi-directionnelle acceptant les requêtes clients et leurs réponses ;
- Une interface cliente informant un composant externe *VSC Manager* qu'un composant a besoin d'être remplacé ;
- Une interface serveur permettant d'activer ou de désactiver un composant nommé *Activate-QoS* (figure 4.2) permettant d'activer l'ensemble des composants de contrôle de conformité à un contrat QoS.

Le cas Springoo, intégrant les détails du *Load Balancer* est représente dans la figure 4.3.

Du point de vue hiérarchique, nous avons intégré un module de QoS pour la composition (http, load balancer, jonas[1], jonas[2]), permettant de l'avertir des "*Out* contrat" des composants internes.

Ce cas a été validé par la plate-forme de modélisation VCE. Le code ADL généré permet de décrire toute l'architecture de Springoo : les composants, les liens, les classes, et les interfaces.

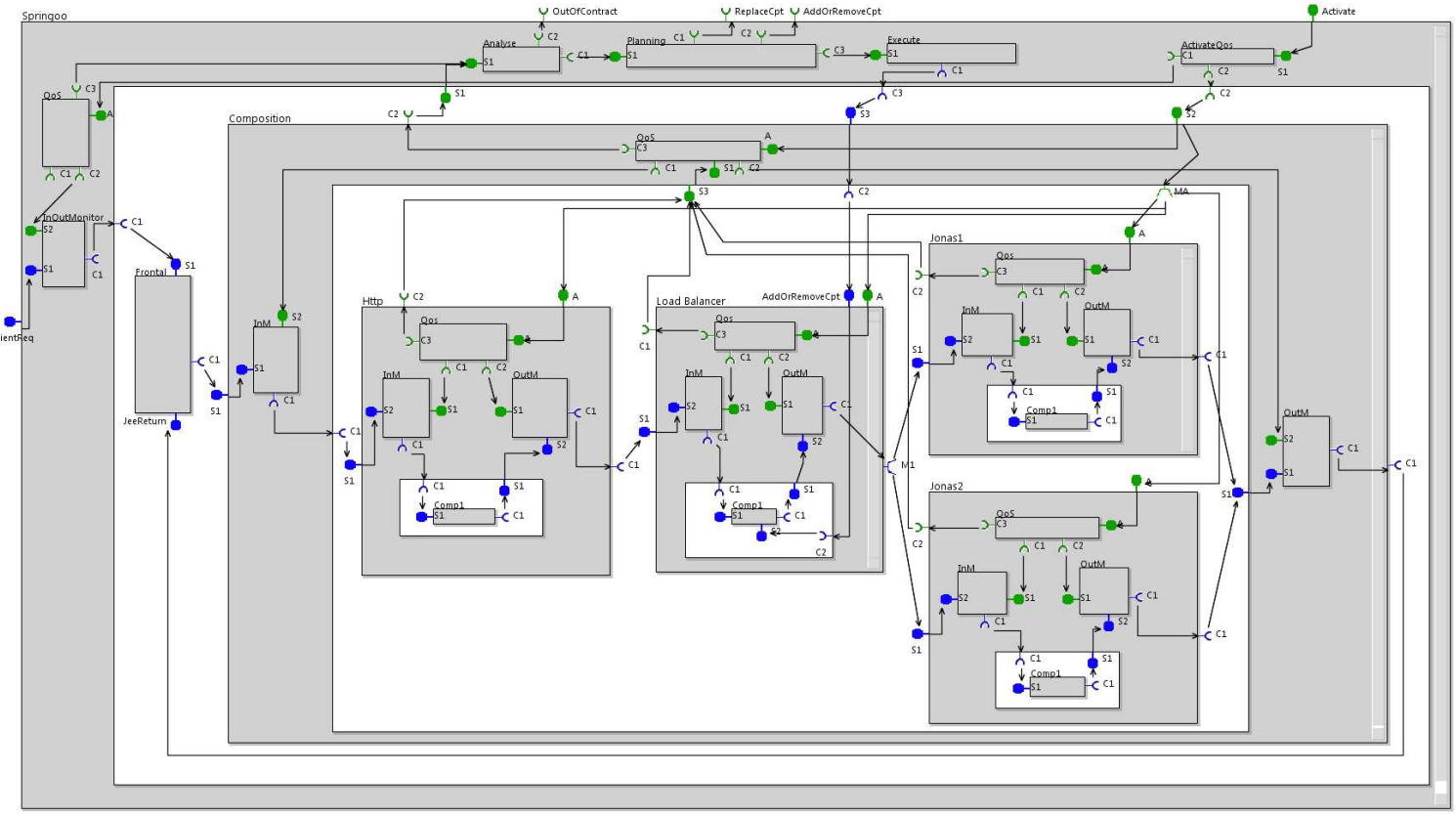



FIGURE 4.3 – Springoo : détails du *Load Balancer*

# Chapitre 5

# Atelier de création de services

Nous avons vu dans le chapitre 1 l'importance de l'évolution des composants pour spécifier les services intégrant la gestion dynamique, puis, dans le chapitre 2 le composant SCC permettant la modélisation de cette évolution. Ensuite, dans le chapitre 3, nous avons décrit nos propositions pour une gestion de composition de ces composants, c'est-à-dire indépendamment d'une part de la répartition des composants de service dans le réseau et d'autre part de la technologie sous-jacente.

Dans ce chapitre, nous allons proposer un atelier qui devrait faciliter la création de nouveaux services suite à la modélisation de l'ensemble de notre écosystème. En fait, au début de mes recherches dans le domaine de la conception et déploiement d'objets d'un système distribué, nous avions abouti à un atelier de création de service orienté objet [projet PILOTE (*Processus d'Ingénierie du LOgiciel de Telécommunications*)]. Forte des résultats obtenus [22], je pense que nous pouvons avoir la même synergie en suggérant un atelier fondé sur les composants de service SCC [77]. L'atelier de création de services s'appuie sur la modélisation permettant de proposer des référentiels de service virtualisés et déployables ainsi qu'un pilotage (processus d'exécution) personnalisé.

C'est ainsi que dans ce chapitre nous commençons par la description de l'architecture de l'atelier de création de service que nous proposons dans le paragraphe 5.1. Ensuite, dans le paragraphe 5.2 nous allons décrire la démarche de modélisation des composants et des liens. Un référentiel de composants et de liens sera décrit dans le paragraphe 5.3. Puis, dans le paragraphe 5.4, nous présentons la composition de services et dans le paragraphe 5.5 le pilotage. En conclusion, dans le paragraphe 5.6 nous proposons un scénario d'utilisation de l'atelier dans un contexte mobile.

## 5.1 Architecture de l'atelier de création de service

Nous décrivons ici l'architecture logique qui définit l'atelier de développement non limité à un secteur particulier. La proposition s'inscrit dans un cadre plus général de la démarche de Model Driven Engineering (MDE) [78], [79]. C'est la garantie de disposer d'outils, de langages et de méthodes plus éprouvées et d'assurer une certaine pérennité des choix d'environnement. L'architecture que nous proposons est destinée à accroître la maîtrise du processus d'ingénierie du logiciel dans un environnement du Cloud et de l'Internet du Futur.

L'atelier que nous proposons établit les constantes fortes du processus logiciel et propose d'en déduire une architecture générique d'un atelier de développement. La figure 5.1 schématise le cadre de l'atelier que nous proposons.

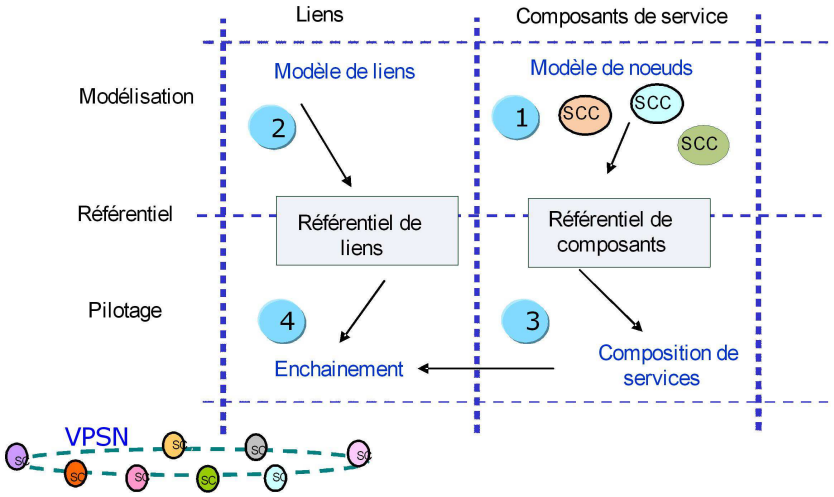



FIGURE 5.1 – Atelier de création de services

Nous distinguons une double séparation à des fins d'automatisation :
- d'un coté la séparation entre les *composants de service et les liens* ;
- et de l'autre côté de la séparation entre *la modélisation et le pilotage*.

Cette séparation permet une plus grande indépendance (statique et dynamique) et par conséquent de meilleures capacités de flexibilité ainsi qu'une mise en oeuvre plus efficace.

En effet, la partie statique (composant de service et les liens) permet de thésauriser et d'offrir des composants de service, sans pour autant imposer les choix en termes de pilotage : chorégraphie ou orchestration, reposant sur des connexions lâches, variant d'un processus à l'autre. Les quatre étapes à suivre basées sur l'architecture de l'atelier sont les suivantes(figure 5.1) :

1. Modélisation en composant de service SCC. Alimentation du référentiel ;
2. Modélisation des liens (composant de service "lien"). Alimentation du référentiel ;
3. Composition de services des différents niveaux de visibilité en puisant dans le référentiel.
4. Pilotage : une approche par les événements (*even driven approach*).

   Dans les paragraphes qui suivent, nous décrivons ces différentes étapes.

## 5.2 Modélisation des composants et des liens

L'architecture de l'atelier s'appuie sur nos propositions du modèle générique des composants *as-a-service* que nous avons décrit dans le chapitre 2, sachant que les liens se modélisent de la même façon *"liens as-a-service"*.

Au niveau de modélisation de composants, le composant GCM sur lequel nous nous appuyons, est standardisé par l'ETSI [46], [47], [70]. Il est aujourd'hui adopté par les plates-formes

de services. En effet, nous avons appliqué et vérifié l'adéquation de la notation GCM à la modélisation des concepts de développement d'applications réparties. Le raffinement de composant SCC permet en plus de s'adapter au contexte "as-a-service".

La modélisation de liens représente les différents liens qui peuvent être assignés aux différents protocoles réseaux, soit http, https, SIP, SOAP, etc. La qualité de service sera également intégrée à chaque composant de service "lien" selon le modèle générique de quatre critères. Une description en langage OVF est donnée dans la figure 5.2.

```
<!—Network as a Service (NaaS) constraint element definitions -->
<xs:element name="appE2EDelay" type="tns:linkSetVmSetConstraint"
                      QoS requested=" currentvalue " QoS offered=" thresholdvalue " />
<xs:element name="appE2EAvaibility" type="tns:linkSetVmSetConstraint"
                      QoS requested=" currentvalue " QoS offered=" thresholdvalue " />
<xs:element name="appE2EReliability" type="tns:linkSetVmSetConstraint"
                      QoS requested=" currentvalue " QoS offered=" thresholdvalue " />
<xs:element name="appE2ECapacity" type="tns:linkSetVmSetResourceConstraint"
           QoS requested=" currentvalue " QoS offered=" thresholdvalue " />
```

FIGURE 5.2 – Composants de service : lien E2E réseau

## 5.3 Référentiels de composants et de liens

Les composants de service et les liens modélisés exportés sont accessibles via les catalogues qui constituent un espace de partage sur la base d'une représentation commune. Ils alimentent les connaissances facilitant la réutilisation et la mutualisation. Ainsi le développeur s'appuyant sur le référentiel définira ses applications à travers la composition de service [80] et le contrôle souhaités.

Les interfaces entre la modélisation, le pilotage et le catalogue permettent l'échange des modèles en formats normalisés OVF, ADL.

## 5.4 Composition de services

Pour compléter la modélisation de notre écosystème nous faisons référence au modèle <Nœuds, Liens, Réseaux> et aux niveaux de visibilité permettant d'avoir une abstraction architecturale à des fins d'ingénierie dynamique et flexible d'une session utilisateur personnalisée. Selon le modèle abstrait <Nœuds, Liens, Réseaux>, que nous appelons "NLR" [61], et les niveaux de visibilité : service, transport network, équipement, correspondent SaaS/PaaS, NaaS (*Network as-a-Service*), IaaS dans l'architecture de Cloud Computing, les différentes sessions sont représentées dans la figure 5.3.

Les sessions horizontales par niveau sont le VPEN (*Virtual Private Equipment Network*), le VPCN (*Virtual Private Connectivity Network*), le VPSN (*Virtual Private Service Network*), soit :

- Composition de services VPSN : nœuds et liens de niveau service applicatif ;
- Composition de service VPCN : nœuds et liens de niveau connectivité réseau ;

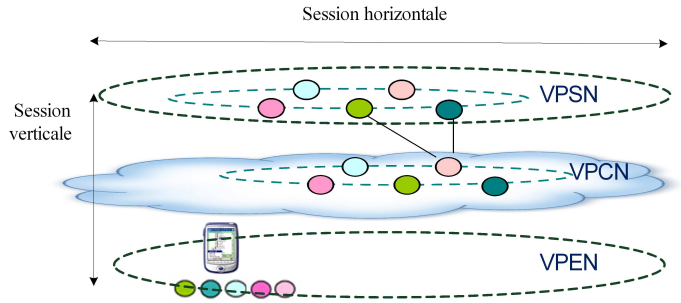



FIGURE 5.3 – Modélisation de l'écosystème

– Composition de service VPEN : nœuds et liens de niveau de l'équipement.

La session verticale représente une session d'utilisateur. Elle sera établie selon ses préférences et s'appuiera sur les sessions horizontales.

## 5.5 Pilotage

Le processus de pilotage devrait être suffisamment dynamique pour que, par simple choix d'enchainement des composants, on puisse obtenir autant de nouveaux processus que souhaité. Il y a en outre un besoin de réaliser automatiquement ces processus et pouvoir les recomposer.

Dans notre approche, le pilotage est axé sur une approche événementielle (*event driven approach*). Comme nous l'avons vu dans le chapitre 3, le Virtual Private Service Network représente la sélection des composants services et leur séquençage. Le lien représente les interactions entre les services au niveau logique. Le VPSN est créé au moment où l'utilisateur se connecte à la plateforme. Les offres commerciales sont disponibles dans les référentiels. Un composant de la plate-forme, le "traducteur", sait faire le *mapping* entre les offres du catalogue et les composants de services SCC qui fournissent cette offre. Le VPSN est créé lorsque tous les composants SCC correspondant à l'offre commerciale du développeur ont été attachés à une session.

Fondé sur l'autocontrôle (*self-controlled*), le pilotage permet de répondre en temps réel a une dégradation de service, en s'appuyant sur la boucle MAPE pour ajouter par exemple un composant (N+1) pour faire face à une augmentation de charge.

L'approche événementielle du pilotage permet différents processus d'organisation, de coordination et de gestion de services comme :

– l'orchestration de services ;
– la chorégraphie de services.

La démarche centralisée dans l'orchestration de service [81], est différente de la chorégraphie [82] agrégeant les services de façon décentralisée. Dans l'Internet du Futur, nous préconisons la création d'une fédération de service dans lequel les services seront composés suivant une approche combinée, où la chorégraphie et l'orchestration se complèteront. L'objectif est d'atteindre une grande flexibilité, tout en simplifiant l'intégration d'intra- ou d'inter-organisation selon les exigences d'un contexte spécifique (par exemple des garanties de sécurité élevée).

## 5.6 Conclusion

Un scénario d'utilisation de l'atelier dans un contexte de Cloud Computing mobile est représenté dans la figure 5.4. L'atelier de création de service permettra au développeur dans la phase de conception de :

- (1) Sélectionner les composants de service SCC dans le référentiel s'ils existent, sinon de les créer.
- (2) Définir la composition du service.
- (3) Sélectionner les liens.
- (4) Définir la logique de service à travers le VPSN.

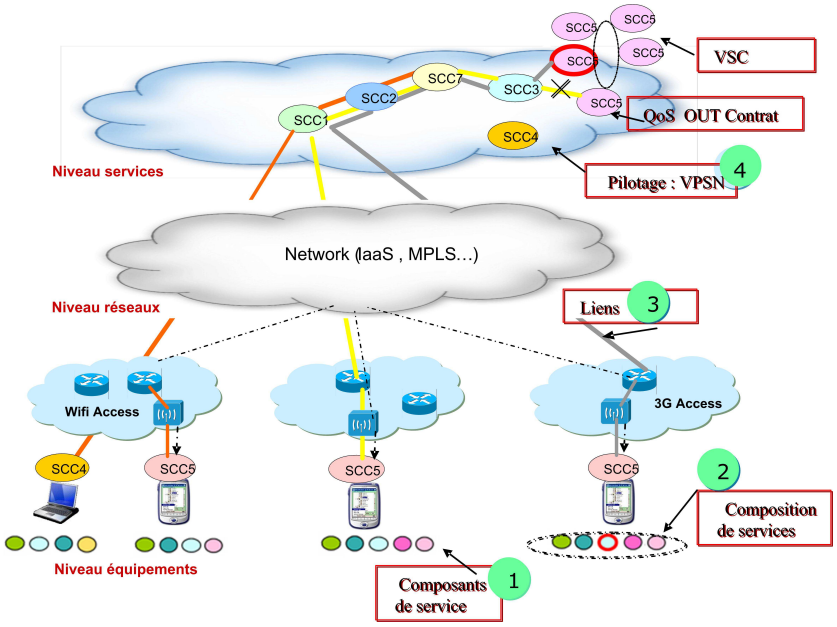



FIGURE 5.4 – Cas d'utilisation de l'atelier de création de services

Dans la phase de pilotage, dans le cas où le développeur souhaiterait implémenter le composant SCC, il pourra sélectionner les services ubiquitaires (exemple SCC5, figure 5.4) et créer le VSC, communautés virtuelles de composants de service ubiquitaires.

La modélisation de composant de services SCC apporte la modularité, la portabilité et la réutilisation. Les composants modélisés servent de base pour la définition d'une bibliothèque de composants de services. En outre, les techniques de modélisation utilisées offrent un niveau intéressant d'intégration des outils conformes aux normes open source : OVF, ADL.

L'atelier de création de service permettra de concevoir et développer des applications qui peuvent profiter pleinement de la composition centralisée ou décentralisée de services, être flexible et dynamique, ainsi que fiable. Plus précisément, l'atelier offrira des réponses pour aider les ingénieurs dans toutes les phases du cycle de vie du service : conception, développement, validation, ainsi que l'exploitation, axée sur l'orchestration et la chorégraphie pour l'Internet du Futur et du Cloud Computing.

# Chapitre 6

# Du composant SCC au SLA

Ce chapitre complète notre vue globale de l'apport des aspects non fonctionnels à travers un modèle de la QoS. Il présente le modèle générique de SLA (*Service Level Agreement*) que nous avons proposé dans le projet OpenCloudware [42] et comme document du projet (draft) à l'ETSI [38]. La spécificité de ce modèle est qu'il permet d'expliciter :

- D'une part les exigences de l'utilisateur (SLO) publiés dans les normes à l'ETSI [37], [34] ;
- D'autre part les offres du fournisseur de services (service et QoS associée) selon le même modèle de QoS (quatre critères génériques).

Pour répondre au SLO demandé par l'utilisateur, l'offre du fournisseur sera représentée par la composition de services fondée sur le composant SCC.

C'est ainsi que dans ce chapitre nous commençons par la description de notre approche de SLA (§ 6.1), puis celle du son modèle générique (§ 6.2). Enfin, dans le paragraphe 6.3 nous présenterons l'expression de SLO en adéquation avec les actions de gestion fondées sur le composant SCC.

## 6.1 Approche

Notre approche consiste à proposer un modèle SLA qui explicite et met en adéquation, d'une part, les exigences de l'utilisateur (SLO et Obligation) et d'autre part, les offres du fournisseur sous forme de services à travers le modèle et l'expression de la QoS [83].

Comme nous avons vu dans le paragraphe 3, un service (composant SCC) est défini par sa fonction et son comportement QoS, c'est-à-dire par le nom de critère de QoS, son type et sa valeur (*QoSCriteriaName, QoSCriteriaType et QoSCriteriaValue*). L'utilisateur exprimera son SLO à travers les quatre critères. Le fournisseur sélectionne dans son catalogue les SCC répondant au mieux à la demande (SLO).

La délivrance du service personnalisé de l'utilisateur est effectuée selon le SLA qui introduit un ensemble de services si nécessaire pour contrôler et gérer la QoS de bout en bout. Grâce au SCC et à la boucle MAPE de la composition nous avons une automatisation de la gestion du niveau service. La délivrance des services prend en compte le profil de l'utilisateur, le contexte et son SLO exprimé.

## 6.2 SLA (*Service Level Agreement*)

Le Service Level Agreement (SLA) est un contrat qui définit un accord spécifique et personnalisé entre un fournisseur de services et un client [84], [16], [85]. Notre modèle générique de SLA a été élaboré [42], [86] pour formaliser les éléments SLA qui constitueront le contrat entre un client et un fournisseur. Il est représenté dans la la figure 6.1 et composé par :

1. Les parties (parties signataires et tiers) ;
2. Les contraintes de haut niveau ;
3. L'utilisateur, correspondant à la demande : SLO, couverture et les conditions d'utilisation ;
4. Le fournisseur, correspondant à l'offre : services offerts et la qualité de service associée ;
5. Les conditions du contrat SLA.

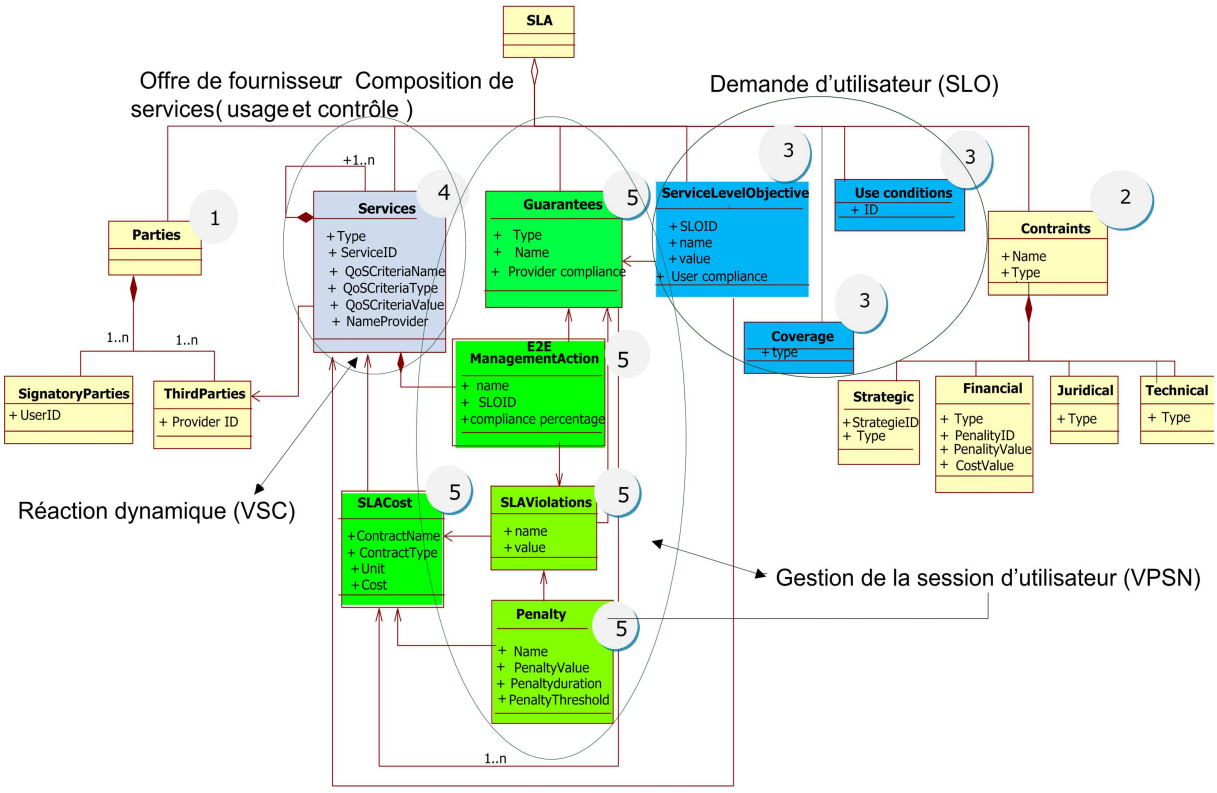



FIGURE 6.1 – Modèle générique de SLA

Les "Parties" (figure 6.1, boulette 1) indiquent les entités contractantes qui négocient le contrat SLA. Elles peuvent être soit les signataires, c'est à dire, les gestionnaires et les développeurs, soit des fournisseurs tiers tels que, par exemple, le fournisseur de IaaS dans le Cloud, le fournisseur de mesures.

Le modèle SLA proposé intègre les contraintes imposées par l'utilisateur et par le fournisseur de service. Ces contraintes sont décrites dans la classe "contraintes" qui spécifie leurs différents

types. Nous avons défini les contraintes stratégiques, financières, juridiques et techniques (figure 6.1, boulette 2).

L'utilisateur exprime sa demande et ses exigences en terme de SLO (*Service Level Objective*), de couverture géographique et aussi des conditions d'utilisation (figure 6.1, boulette 3).

Dans notre modèle, l'offre de fournisseur ((figure 6.1, boulette 4) représente une composition de composants de services (composant SCC) correspondant à la demande de l'utilisateur (SLO). Les fournisseurs peuvent proposer les mêmes services, mais différenciés par leur type de SLA, leur prix, ainsi que la façon avec laquelle ils sont construits, déployés et gérés.

Les conditions personnalisent le contrat SLA. Les autres données du contrat sont : durée du contrat, garanties, actions de gestion SLA propres au VPSN, SLA violation, pénalités et coûts liés à l'accord signé (figure 6.1, boulette 5). Les actions de gestion de SLA permettent de réaliser une session d'utilisateur personnalisée (VPSN).

## 6.3 SLO (*Service Level Objective)*

Service Level Objective (SLO) est un moyen pour l'utilisateur d'exprimer ses besoins. Les objectifs exprimés par l'utilisateur peuvent être liées à la qualité de service de bout en bout (E2E). Les objectifs de E2E est de définir le niveau de QoS de service final (service composé) fourni à l'utilisateur. En effet, si le client a besoin d'une composition de service, il peut alors préciser les conditions liées à leur fonctionnement, telles que les temps de réponse, de taux de disponibilité et sa mise à l'échelle (*scalability*).

Par exemple dans le cas Springoo, l'utilisateur exige que le temps de traitement de son service soit inférieur à 2 s dans 90 % des cas, si le nombre de requêtes traitées est inférieure à 1000 req/s. D'autre part, si le nombre de requêtes dépasse 1000 req/s, il peut toujours exiger un temps de traitement moins que 2s mais avec un taux de défaillance moins strict (par exemple 60 % au lieu de 90 % des cas). Les SLOs liés à cet utilisateur auront les valeurs suivantes :

- SLO1 : Temps de traitement E2E $<$ 2 s si le nombre de requêtes $<$ 1000 req/s dans 90% des cas.
- SLO2 : Temps de traitement E2E 2 s $<$ si le nombre de requêtes $>$1000 req/s dans 60 % des cas.

Les actions de gestion (figure 6.1, boulette 5) représentent les actions effectuées par le fournisseur afin d'obtenir le SLA requis, par exemple effectuer la reconfiguration dynamique. En effet, pour chaque SLO, les valeurs seuils sont définies pour indiquer que les violations de la SLA sont atteintes. Pour éviter les cas de violation, un ensemble d'actions est défini pour chaque condition de SLO.

Notre approche est cruciale, car le modèle de QoS devrait permettre une meilleure compréhension entre d'une part, les utilisateurs qui expriment leurs SLOs s'appuyant sur les normes (ETSI EG 202 009-1 [37], ETSI EG 202 009-2 [34]) et d'autre part, les fournisseurs qui ont une *template* SLA [38] pour répondre à cette demande personnalisée de SLOs. Une synthèse efficace des besoins des utilisateurs et des offres du fournisseur devrait en résulter.

# Chapitre 7

# Conclusions et perspectives de recherche

## 7.1 Conclusion

L'écosystème, du Cloud Computing et de l'Internet du Futur, est séduisant à plus d'un titre. Il permet de concevoir sa propre application à partir de composants proposés dans un catalogue. Il permet de se connecter à presque tout. Il permet des solutions "à la demande". L'étape de création est simplifiée.

En plus, l'un des principes de Cloud Computing est de ne payer que ce que l'on utilise. Au fournisseur de s'adapter. Sa capacité à gérer l'élasticité, la haute disponibilité et l'approvisionnement à la demande fait partie de son offre. L'intégration dés la conception de la gestion automatisée a toujours était souhaitable, mais elle est encore plus cruciale dans le contexte concurrentiel de notre nouvel écosystème. Or le fournisseur de Cloud satisfait ces propriétés en fonction des exigences des clients et des charges. Cette demande et cette offre doivent être explicitement consignées et garanties par le SLA. En effet, plusieurs fournisseurs de Cloud proposent les mêmes services qui se différencient par leurs niveaux de qualité de service, de sécurité, leur prix et la manière dont ils sont construits, déployés et gérés.

La question à laquelle nous avons essayé d'apporter des réponses est donc : comment faire évoluer les modèles à composant afin de faire converger la "conception partagée" et "la gestion automatisée" personnalisables et "négociables". Le résultat de la négociation étant consigné dans le contrat SLA entre le demandeur et l'offreur, une abstraction de haut niveau est nécessaire.

Ce manuscrit consigne nos réflexions et nos contributions pour la conception partagée, la gestion automatisée et pour la négociation de SLA.

La "conception partagée" repose sur la composition de service, les composants *as a service* et les liens lâches pour la composition personnalisée, les interconnexions entre composants variées à la demande.

La "gestion automatisée" s'appuie sur des informations au bon endroit. Les sondes peuvent capturer des informations au niveau du service permettant un approvisionnement à la demande et une réaction dynamique au plus prés du comportement de chaque composant. La négociation est représentée par le SLA à l'instar d'un contrat en bonne et due forme. Le fournisseur garantit un service en fonction de moyens de contrôle explicite. Nous avons abordé également plusieurs aspects cruciaux du SLA : des problèmes de modélisation, aux questions relatives à leur mise en

œuvre et à la gestion d'exécution.

L'avancée importante pour le Cloud Computing et l'Internet du Futur est la notion de service via les propriétés, de sorte que la composante *as a service* devienne une entité de composition de service. Ainsi, on peut choisir et assembler les services d'application dans un réseau de services qui peuvent être librement associés pour créer des processus flexibles et dynamiques (VPSN).

L'adoption du SCC aide à cette composition de services avec une garantie de qualité de service. L'approche que nous préconisons fournit une composition personnalisée et automatisée de services avec un niveau de service garanti. Les outils (VCE,GCM/proactif) et la mise en œuvre d'un exemple simplifié (Spingoo) montrent la faisabilité de nos propositions.

Notre approche permet de répondre aux intérêts des :
- Développeurs avec une architecture orientée service, la prise en compte des règles métiers et la surveillance des contrats de QoS ;
- Demandeurs de service avec une expression du SLO et une identification explicite des violations du SLA pour définir les pénalités associées ;
- Fournisseurs avec un modèle de SLA et des éléments pour un catalogue de service avec QoS permettant de faire une différentiation utile à la concurrence.

## 7.2 Perspectives de recherche

Le domaine de mes recherches est en pleine évolution. Les perspectives que j'envisage à moyen terme se déclinent en trois axes :

1. Sécurité dans le Cloud Computing Mobile ;
2. Service *"delivery"* dans le cycle de vie ;
3. Composition et urbanisation des services.

### 7.2.1 Sécurité dans le Cloud Computing Mobile (MCC)

Il s'agit de la suite de nos travaux sur les composants *as a service* dans la norme ETSI EG 202 009-2 [34]. L'utilisation du concept de *"security as a service"* vu comme un service de sécurité offert pour assurer la sécurité des applications est envisagé. Ainsi, la sécurité est fournie comme un service en se déclinant sous forme de composants de services qui doivent être composables selon les besoins (spatiaux et temporels), les préférences de l'utilisateur et les objectifs de sécurité à garantir.

En termes de sécurité dans le cloud mobile, l'enjeu est important. En effet, les frontières du système auparavant bien délimitées deviennent beaucoup plus ouvertes puisque le système de la plate-forme mobile se voit étendue par le système du cloud. Ce qui se traduit par des interactions entre les services eux-mêmes d'une part et entre les services et l'utilisateur d'autre part. L'hétérogénéité et la variété des ressources (terminaux, réseaux et services) impliquées dans la session de l'utilisateur, augmentent la complexité des tâches de sécurité. Cet environnement ouvert, hétérogène et mobile, qui peut présenter des risques significatifs en termes de sécurité, est devenu de plus en plus vulnérable. Pour cette raison, la sécurité ne peut plus être garantie de manière statique et centralisée. Le contrôle intégré sera alors nécessaires dans des situations complexes et évolutives [87], [88].

Parmi les avantages de notre proposition, on peut aussi noter le caractère open source d'OVF et de l'ADL que nous utilisons.

### 7.2.2 Service "delivery" dans le cycle de vie

J'ai montré dans le chapitre 3 les mécanismes permettant de gérer les dégradations de la qualité de service dans une session d'utilisateur. Les approches existantes, telles que *média delivery* correspond à la gestion du réseau de transport, afin de trouver la meilleure solution qui maintient le service conformément au SLA de l'utilisateur. La mobilité et l'hétérogénéité de l'environnement de l'Internet du futur et du cloud doit intégrer la gestion au niveau service. Cela suppose de prendre en compte les préférences de l'utilisateur, de fournir la personnalisation à travers une composition de services hétérogènes, de garantir la continuité de service de mobilités, d'assurer le re-provisioning des ressources durant la mobilité.

Dans ce contexte, je souhaite étudier la un service délivrance de service (*service delivery*) qui se place au dessus du *média delivery*, afin de gérer la mobilité et les changements dans l'environnement ambiant de l'utilisateur. Dirigé par le modèle générique décrit dans le paragraphe 2.4.3 et associé aux profils de cycle de vie (figure 7.1), un réseau de services pourra s'autogérer indépendamment des infrastructures de transport, afin d'assurer la continuité de service tout en maintenant la QoS demandée.

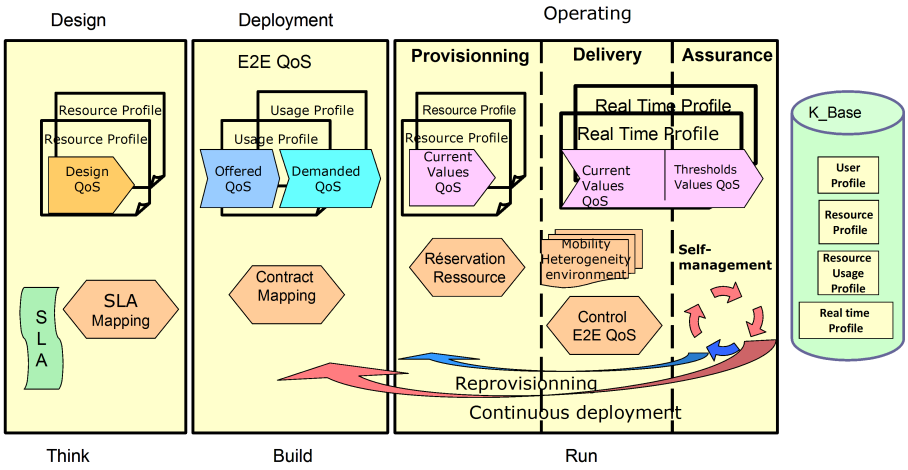



FIGURE 7.1 – Gestion autonome dans le cycle de vie

Le *service delivery* correspondra à la délivrance du service personnalisé selon le SLA contracté. Il fera la référence à un ensemble de composants SCC qui fournissent les services de création d'utilisateur, de la sécurité, de contrôle d'une session d'utilisation, de l'usage, de facturation, etc.

Cette délivrance des services tiendra compte du profil de l'utilisateur et du contexte. Les profils sont sollicités au cours des différentes phases du cycle de vie d'un composant SCC :

1. Profil de ressources dans la phase de conception (THINK) ;
2. Profil d'usage dans la phase de déploiement (BUILD) ;
3. Profil temps réel dans la phase d'exécution (RUN).

Le profil de ressources est associé à la phase de conception d'un composant de service. Il décrit les valeurs de conception des critères de qualité de service. Le profil d'usage est utilisé lors

de la phase de déploiement pour associer QoS offerte à QoS demandée et déterminer la valeur seuil des critères de qualité de service. Le profil temps réel (real-time) est utilisé pendant la phase d'exécution. Il décrit les capacités actuelles des ressources du service. Ces profils contiennent les valeurs courantes des critères de qualité de service. Les profils correspondants à chaque phase donneront une démarche de gestion autonome dans le cycle de vie de services.

Les liens que j'entretiens avec l'équipe AIRS (*Architecture et Ingénierie des Réseaux et Services*) de Télécom ParisTech pourraient trouver ici un cadre formel à ces travaux.

### 7.2.3 Composition et urbanisation des services

Il s'agit de développer notre collaboration avec l'équipe Oasis de l'INRIA et de poursuivre nos travaux sur le composant SCC. Notre proposition est validée par une plate-forme de conception et de vérification VCE. Dans ce contexte, je souhaite poursuivre l'étude pour construire les premiers modèles d'applications à base de composant SCC, vérifier leurs propriétés et générer le code pris en charge par un environnement d'exécution GCM/proactive [89], [69] .

Les différentes études de cas d'implémentation permettront de prendre en compte une urbanisation de services, c'est-à-dire une démarche qui prévoit des principes et des règles ainsi qu'un cadre cohérent, stable et modulaire, auquel les différentes instances décisionnaires peuvent se référer pour toute décision relative à la composition et la gestion de services. S'appuyant sur les composants SCC nous pouvons obtenir un contrôle de gestion souhaité.

# Bibliographie


[1] Mikoczy, E., Kotuliak, I., van Deventer, O. : Evolution of the converged ngn service platforms towards future networks. Future Internet **3**(1), 67–86 (2011)

[2] Lohier, S., Quidelleur, A. : Le Réseau Internet - Des Services aux Infrastructures, p. 416 (2010). https ://hal-upec-upem.archives-ouvertes.fr/hal-00687308

[3] Mell, P.M., Grance, T. : Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States (2011)

[4] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M. : A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010). doi :10.1145/1721654.1721672

[5] Future Internet Public Private Partnership (FI-PPP). `http://www.fi-ppp.eu/`

[6] Fajjari, I., Aitsaadi, N., Pióro, M., Pujolle, G. : A new virtual network static embedding strategy within the cloud's private backbone network. Computer Networks **62**, 69–88 (2014). doi :10.1016/j.comnet.2014.01.004

[7] Blaiech, K., Mounaouar, O., Cherkaoui, O., Beliveau, L. : Runtime resource allocation model over network processors. In : 2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014, pp. 556–561 (2014). doi :10.1109/IC2E.2014.33. http ://dx.doi.org/10.1109/IC2E.2014.33

[8] Van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D. : Continuous Monitoring of Software Services : Design and Application of the Kieker Framework (2009)

[9] Ruz, C., Baude, F., Sauvan, B. : Flexible Adaptation Loop for Component-based SOA applications. In : 7th International Conference on Autonomic and Autonomous Systems (ICAS 2011), pp. 29–36 (2011). Best paper awarded

[10] Akue, L., Lavinal, E., Desprats, T., Sibilla, M. : Runtime configuration validation for self-configurable systems. In : Turck, F.D., Diao, Y., Hong, C.S., Medhi, D., Sadre, R. (eds.) 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013, pp. 712–715 (2013)

[11] Marie, P., Desprats, T., Chabridon, S., Sibilla, M. : Extending ambient intelligence to the internet of things : New challenges for qoc management. In : Hervás, R., Lee, S., Nugent, C.D., Bravo, J. (eds.) Ubiquitous Computing and Ambient Intelligence. Personalisation and User Adapted Services - 8th International Conference, UCAmI 2014, Belfast, UK, December 2-5, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8867, pp. 224–231 (2014)

[12] Booch, G. : Object-oriented Analysis and Design with Applications (2Nd Ed.). Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1994)

[13] Gannon, D., Krishnan, S., Fang, L., Kandaswamy, G., Simmhan, Y., Slominski, A. : On building parallel & grid applications : Component technology and distributed services. Cluster Computing **8**(4), 271–277 (2005). doi :10.1007/s10586-005-4094-2

[14] Szyperski, C. : Component Software : Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

[15] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B. : The Fractal component model and its support in Java. Software : Practice and Experience **36**(11-12), 1257–1284 (2006). doi :10.1002/spe.767

[16] Ruz, C., Baude, F., Sauvan, B. : Enabling SLA Monitoring for Component-Based SOA Applications – A Component-Based Approach. In : 36th Euromicro Conf. on Software Engineering and Advanced Applications, SEAA, Work In Progress Session (2010)

[17] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., Pŕez, C. : GCM : a grid extension to Fractal for autonomous distributed components. Annals of Telecommunications **64**(1-2), 5–24 (2009). doi :10.1007/s12243-008-0068-8

[18] SCA Policy Framework Service Component Architecture Specifications (2011)

[19] Baude, F., Henrio, L., Ruz, C. : Programming distributed and adaptable autonomous components - the GCM/ProActive framework. Software, Practice and Experience (2015). doi :10.1002/spe2270

[20] Jamshidi, J., Ahmad, A., Pahl, C. : Autonomic resource provisioning for cloud-based software. In : Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS 2014, pp. 95–104. ACM, New York, NY, USA (2014). doi :10.1145/2593929.2593940. http ://doi.acm.org/10.1145/2593929.2593940

[21] Parashar, M., Liu, H., Li, Z., Matossian, V., Schmidt, C., Zhang, G., Hariri, S. : Automate : Enabling autonomic applications on the grid. Cluster Computing **9**, 161–174 (2006). doi :10.1007/s10586-006-7561-5

[22] Aubonnet, T., Simoni, N. : PILOTE : A service creation environment in Next Generation Network. In : Proceedings IEEE Intelligent Network Workshop (IN'2001), Boston, USA, pp. 36–40 (2001)

[23] Aubonnet, T., Simoni, N. : Création de service et qualité de service. In : Gestion de Réseau et de Service, GRES'2002, pp. 78–92 (2002)

[24] Aubonnet, T., Simoni, N. : Elaboration du module fonctionnel pilotage de processus : modele du processus réseau intelligent de bouygues télécom. Rapport technique, PILOTE, sous-projet 3.2 (2001)

[25] Aubonnet, T., Simoni, N. : End to end qos measurements. Rapport technique, ALCATEL, projet Next Generation Network and Service Management, (2002)

[26] Aubonnet, T., Simoni, N. : Information model for the qos dynamic management. Rapport technique, ALCATEL, projet Next Generation Network and Service Management, (2002)

[27] Aubonnet, T., Simoni, N. : Generic service model for end to end quality of service. Rapport technique, ALCATEL, projet Next Generation Network and Service Management, (2003)

[28] Aubonnet, T., Simoni, N. : Towards a pro-active sla (service level agreement. Rapport technique, ALCATEL, projet Next Generation Network and Service Management, (2003)

[29] Aubonnet, T., Simoni, N. : Service provisioning process-based component. Rapport technique, ALCATEL, projet Next Generation Network and Service Management, (2004)

[30] Mouza, C.d., Métais, E., Lammari, N., Akoka, J., Aubonnet, T., Comyn-Wattiau, I., Fadili, H., Cherfi, S. : Towards an automatic detection of sensitive information in a database. In : Proceedings of the 2010 Second International Conference on Advances in Databases, Knowledge, and Data Applications. DBKDA '10, pp. 247–252. IEEE Computer Society, Washington, DC, USA (2010). doi :10.1109/DBKDA.2010.17. http ://dx.doi.org/10.1109/DBKDA.2010.17

[31] Aubonnet, T. : Intégration de la sécurité dans la conception des systemes d'information. In : INFormatique des ORganisations et Systemes d'Information et de Décision, INFORSID'2005, pp. 12–24 (2005)

[32] Akoka, J., Aubonnet, T., Bucumi, J.-S., Comyn-Wattiau, I., Labiadh, M., Lammari, N., Ledru, Y., Richier, J. : Intégration des propriétés de sécurité dans uml. Rapport technique, CNAM, projet SELKIS (2010). pages 1-76

[33] Akoka, J., Aubonnet, T., Bucumi, J.-S., Comyn-Wattiau, I., Labiadh, M., Lammari, N., Ledru, Y., Richier, J. : A uml profile for security concepts. Tech. report, CNAM, projet SELKIS (2011). pages 1-18

[34] Aubonnet, T., Hebert, P.-Y., Simoni, N. : EG 202 009-2, V0.0.7 : User group ; quality of telecom services ; part 2 : User related parameters on a service specific basis. Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2014). standard. http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=41955

[35] The OpenCloudware project. http://www.opencloudware.org/

[36] Aubonnet, T., Simoni, N., Zakaia Benahmed, G.C. : La création de services. In : Hermes (ed.) Des Réseaux Intelligents Ãă la Nouvelle Génération de services. Série réseaux et télécoms, vol. Traité IC2, pp. 46–75 (2007)

[37] Aubonnet, T., Hebert, P.-Y., Simoni, N. : EG 202 009-1, V0.0.1 : User group ; quality of telecom services ; part 1 : Methodology for identification of indicators relevant to the users. Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2014). standard. http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=41182

[38] Aubonnet, T., Hebert, P.-Y., Simoni, N. : EG 202 009-3, V1.2.1 user group ; quality of telecom services ; part 3 : Template for service level agreements (sla). Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2014). standard. http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=19268

[39] Bruneton, E., Coupaye, T., Stefani, J.B. : The Fractal Component Model Specification. http ://fractal.objectweb.org/specification/index.html (2004)

[40] David, P.-C., Ledoux, T. : An aspect-oriented approach for developing self-adaptive Fractal components. In : Löwe, W., Südholt, M. (eds.) Software Composition. LNCS, vol. 4089, pp. 82–97. Springer, Berlin Heidelberg (2006). doi :10.1007/11821946_6

[41] David, P.-C., Ledoux, T., Léger, M., Coupaye, T. : Fpath and fscript : Language support for navigation and reliable reconfiguration of fractal architectures. Annals of Telecommunications **64**, 45–63 (2009). 10.1007/s12243-008-0073-y

[42] Aubonnet, T., Madelaine, E., Ledoux, T., Kouki, Y., Moreaux, P., Pourraz, F., Ayadi, I., Simoni, N. : Modele logique avancé. Rapport technique, projet OpenCloudware, (2013)

[43] Bouchenak, S., Boyer, F., Hagimont, D., Krakowiak, S., Palma, N.D., Quéma, V., Stefani, J. : Architecture-based autonomous repair management : Application to J2EE clusters. In : Second International Conference on Autonomic Computing (ICAC 2005), 13-16 June 2005, Seattle, WA, USA, pp. 369–370 (2005). doi :10.1109/ICAC.2005.9. http ://doi.ieeecomputersociety.org/10.1109/ICAC.2005.9

[44] Aldinucci, M., Danelutto, M., Kilpatrick, P. : Autonomic management of non-functional concerns in distributed and parallel application programming. In : Proc. of Intl. Parallel and Distributed Processing Symposium (IPDPS) (2009)

[45] Gueye, S., Palma, N., Rutten, E. : Component-based autonomic managers for coordination control. In : Nicola, R., Julien, C. (eds.) Coordination Models and Languages. Lecture Notes in Computer Science, vol. 7890, pp. 75–89 (2013)

[46] TC-GRID, E. : ETSI TS 102 827 : Grid ; grid component model ; part 1 : Gcm interoperability deployment. Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2008). standard. http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=26362

[47] TC-GRID, E. : ETSI TS 102 828 : Grid ; grid component model ; part 2 : Gcm application description. Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2008). standard. http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=30947

[48] Cansado, A., Madelaine, E. : Specification and verification for grid component-based applications : from models to tools. In : de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 180–203. Springer, Berlin Heidelberg (2009). doi :10.1007/978-3-642-04167-9_10

[49] Baude, F., Henrio, L., Naoumenko, P. : Structural reconfiguration : An autonomic strategy for gcm components. In : Proceedings of the 2009 Fifth International Conference on Autonomic and Autonomous Systems, pp. 123–128. IEEE Computer Society, Washington, DC, USA (2009). doi :10.1109/ICAS.2009.28. http ://portal.acm.org/citation.cfm ?id=1546680.1546888

[50] Nuno, G., Henrio, L., Madelaine, E. : Bringing coq into the world of gcm distributed applications. International Journal of Parallel Programming - Special issue HLPP'2013 (2013)

[51] Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S. : Comprehensive QoS monitoring of Web services and event-based SLA violation detection. In : Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing. MWSOC '09, pp. 1–6. ACM, New York, NY, USA (2009). doi :10.1145/1657755.1657756

[52] Ruz, C., Baude, F., Sauvan, B., Mos, A., Boulze, A. : Flexible soa lifecycle on the cloud using sca. In : Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International, pp. 275–282 (2011). doi :10.1109/EDOCW.2011.51

[53] Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.-B. : A component-based middleware platform for reconfigurable service-oriented architectures. Software : Practice and Experience **42**(5), 559–583 (2012). doi :10.1002/spe.1077

[54] Service Component Architecture Specifications (2011). http://www.oasis-opencsa.org/sca

[55] Mos, A., Boulze, A., Quaireau, S., Meynier, C. : Multi-layer perspectives and spaces in soa. In : Proceedings of the 2nd International Workshop on Systems Development in SOA Environments, pp. 69–74. ACM, New York, NY, USA (2008). doi :10.1145/1370916.1370934. http ://portal.acm.org/citation.cfm ?id=1370916.1370934

[56] IBM : An architectural blueprint for autonomic computing. white paper **Fourth Edition** (2006)

[57] Ruz, C., Baude, F., Sauvan, B. : Enabling SLA Monitoring for Component-Based SOA Applications – A Component-Based Approach. In : Proceedings of the Work in Progress Session SEAA 2010, pp. 41–42. Johannes Kepler University Linz, ? ? ? (2010)

[58] Aubonnet, T., Simoni, N. : Self-control cloud services. In : 2014 IEEE 13th International Symposium on Network Computing and Applications, NCA 2014, Cambridge, MA, USA, 21-23 August, 2014, pp. 282–286 (2014). doi :10.1109/NCA.2014.48

[59] Baud, J.L. : ITIL V3 : Préparation Á la Certification ITIL Foundation V3 : Plus de 30 Questions-réponses. Collection certifications. Ed. ENI, St Herblain (2011). http ://opac.inria.fr/record=b1132904

[60] Aubonnet, T., Simoni, N. : Service creation and self-management mechanisms for mobile cloud computing. In : Wired/Wireless Internet Communication - 11th International Conference, WWIC 2013, St. Petersburg, Russia. Proceedings, pp. 43–55 (2013). doi :10.1007/978-3-642-38401-1_4

[61] Simoni, N., Znaty, S. : Gestion de Réseau et de Service : Similitude des Concepts, Spécificité des Solutions, ISBN : 2225829802, Edition Masson, (1998)

[62] Fakhfakh, N., Verjus, H., Pourraz, F., Moreaux, P. : QoS aggregation for service orchestrations based on workflow pattern rules and MCDM method : evaluation at design time and runtime. Service Oriented Computing and Applications **7**(1), 15–31 (2013). doi :10.1007/s11761-012-0124-0

[63] Kouki, Y., de Oliveira Jr., F.A., Dupont, S., Ledoux, T. : A language support for cloud elasticity management. In : 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, IL, USA, pp. 206–215 (2014). doi :10.1109/CCGrid.2014.17

[64] SLA@SOI. `http://sla-at-soi.eu/`

[65] Nepal, S., Zic, J., Chen, S. : WSLA+ : web service level agreement language for collaborations. In : 2008 IEEE International Conference on Services Computing (SCC 2008), 8-11 July 2008, Honolulu, Hawaii, USA, pp. 485–488 (2008). doi :10.1109/SCC.2008.66. http ://doi.ieeecomputersociety.org/10.1109/SCC.2008.66

[66] Müller, C., Gutiérrez, A.M., Resinas, M., Fernandez, P., Cortés, A.R. : iagree studio : A platform to edit and validate ws-agreement documents. In : Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8274, pp. 696–699 (2013)

[67] DMTF : Open virtualization format specification. Technical report, Distributed Management Task Force (DMTF), Sophia-Antipolis, France (2009). standard

[68] Barros, T., Cansado, A., Madelaine, E., Rivera, M. : Model-checking distributed components : The vercors platform. Electronic Notes in Theoretical Computer Science **182**(0), 3–16 (2007). doi :10.1016/j.entcs.2006.09.028. Proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006)

[69] Ameur-Boulifa, R., Henrio, L., Madelaine, E., Savu, A. : Behavioural Semantics for Asynchronous Components. Rapport de recherche RR-8167, INRIA (2012). http://hal.inria.fr/hal-00761073

[70] TC-GRID, E. : ETSI TS 102 829 : Grid ; grid component model ; part 3 : Gcm fractal architecture description language (adl). Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2009). standard. http://webapp.etsi.org/workprogram/Report_WorkItem.asp?wki_id=28856

[71] Soulimani, H.A., Coude, P., Simoni, N. : User-centric and qos-based service session. In : 2011 IEEE Asia-Pacific Services Computing Conference, APSCC 2011, Jeju, Korea (South), December 12-15, 2011, pp. 267–274 (2011). doi :10.1109/APSCC.2011.64. http ://dx.doi.org/10.1109/APSCC.2011.64

[72] Kessal, S., Simoni, N. : Qos based service provisioning in NGN/NGS context. In : 7th International Conference on Network and Service Management, pp. 1–5 (2011)

[73] Simoni, N., Xiong, X., Yin, C. : Virtual community for the dynamic management of NGN mobility. In : Fifth International Conference on Autonomic and Autonomous Systems, ICAS 2009, Valencia, Spain, 20-25 April 2009, pp. 82–87 (2009). doi :10.1109/ICAS.2009.33. http ://doi.ieeecomputersociety.org/10.1109/ICAS.2009.33

[74] Nassar, R., Simoni, N. : Cloud Management Architecture in NGN/NGS Context - QoS-awareness, Location-awareness and Service Personalization. In : CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science, Noordwijkerhout, Netherlands, 7-9 May, 2011, pp. 629–635 (2011)

[75] Alaoui Soulimani, H. : Dynamic management of the end-to-end QoS of a "user-centric" session. Theses, Télécom ParisTech (June 2012). https://pastel.archives-ouvertes.fr/pastel-00834199

[76] TMN : M.3400 : TMN management functions. Technical report, ITU-T (1997). Standard

[77] Aubonnet, T., Simoni, N. : Service creation and self-management mechanisms for mobile cloud computing. In : Wired/Wireless Internet Communication - 11th International Conference, WWIC 2013, St. Petersburg, Russia, June 5-7, 2013. Proceedings, pp. 43–55 (2013)

[78] Bézivin, J., Paige, R.F., Aßmann, U., Rumpe, B., Schmidt, D. : Manifesto - model engineering for complex systems. CoRR **abs/1409.6591** (2014)

[79] Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bezivin, J. : Inter-dsl coordination support by combining megamodeling and model weaving. In : Proceedings of the 2010 ACM Symposium on Applied Computing. SAC '10, pp. 2011–2018. ACM, New York, NY, USA (2010). doi :10.1145/1774088.1774511. http ://doi.acm.org/10.1145/1774088.1774511

[80] Ngoko, Y., Goldman, A., Milojicic, D.S. : Service selection in web service compositions optimizing energy consumption and service response time. J. Internet Services and Applications **4**(1) (2013). doi :10.1186/1869-0238-4-19

[81] Fakhfakh, N., Verjus, H., Pourraz, F. : Qos aggregation in service orchestrations based on the choquet integral. In : Proceedings of the 2011 IEEE 8th International Conference on e-Business Engineering. ICEBE '11, pp. 77–84. IEEE Computer Society, Washington, DC, USA (2011). doi :10.1109/ICEBE.2011.58. http ://dx.doi.org/10.1109/ICEBE.2011.58

[82] Autili, M., Inverardi, P., Tivoli, M. : Automated synthesis of service choreographies. IEEE Software **32**(1), 50–57 (2015). doi :10.1109/MS.2014.131

[83] Ayadi, I., Simoni, N., Aubonnet, T. : Sla approach for "cloud as a service". In : Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, pp. 966–967. IEEE Computer Society, Washington, DC, USA (2013). doi :10.1109/CLOUD.2013.127. http ://dx.doi.org/10.1109/CLOUD.2013.127

[84] TMF : GB917 : SLA management handbook ; release 3.0. Technical report, TeleManagement Forum (TMF) (2011). Standard

[85] Lango, J. : Toward software-defined slas. Commun. ACM **57**(1), 54–60 (2014). doi :10.1145/2541883.2541894

[86] Ayadi, I. : La virtualisation de bout en bout pour la gestion des services Cloud sous contraintes de QoS. Theses, Télécom ParisTech (2014)

[87] Msahli, M., Serhrouchni, A. : PCM in cloud. In : 2014 IEEE International Conference on Granular Computing, GrC 2014, Noboribetsu, Japan, October 22-24, 2014, pp. 201–206 (2014). doi :10.1109/GRC.2014.6982835. http ://dx.doi.org/10.1109/GRC.2014.6982835

[88] Msahli, M., Chen, X., Serhrouchni, A. : Towards a fine-grained access control for cloud. In : Li, Y., Fei, X., Chao, K., Chung, J. (eds.) 11th IEEE International Conference on e-Business Engineering, ICEBE 2014, Guangzhou, China, November 5-7, 2014, pp. 286–291 (2014). doi :10.1109/ICEBE.2014.56. http ://dx.doi.org/10.1109/ICEBE.2014.56

[89] ProActive Parallel Suite. `http://proactive.inria.fr/`