Generic monitoring and reconfiguration for service-based applications in the cloud

Mohamed S. A. Mohamed

Laboratoire CNRS UMR 5157 SAMOVAR
Institut Mines-Télécom/Télécom SudParis

École doctorale S&I
Université d'Évry-Val d'Essonne

GENERIC MONITORING AND RECONFIGURATION FOR SERVICE-BASED APPLICATIONS IN THE CLOUD

PHD THESIS

7 Novembre 2014

Thèse de doctorat de Institut Mines-Télécom, Télécom SudParis
dans le cadre
de l'école doctorale S&I en co-accréditation avec
l'Université d'Évry-Val d'Essonne

(spécialité informatique)

par

Mohamed Mohamed

Président de jury:	Pr. Nazim AGOULMINE	Université d'Evry Val d'Essonnes
Rapporteurs :	Pr. Françoise BAUDE	Université de Nice Sophia-Antipolis
	Pr. Boualem BENATALLAH	University of South Wales, Australia
Examinateurs	Dr. Fabienne BOYER	Université de Grenoble
	Dr. Djamel BELAÏD (encadrant)	Télécom SudParis
	Pr. Samir TATA (Directeur de thése)	Télécom SudParis

Thèse numéro : 2014TELE0025

Acknowledgements

First, I would like to express my deepest sense of gratitude to my supervisor Djamel BELAÏD for his patient guidance, encouragement and excellent advices throughout this research. I would like also to express my sincere gratitude and appreciation to my thesis director Samir TATA who has continually and convincingly conveyed a spirit of adventure in regard to research.

I would like to thank all members of the jury especially Professor Fraçoise BAUDE and Professor Boualem BENATALLAH for their attention and thoughtful comments.

I owe my deepest gratitude and warmest affection to the members of the computer science department of Telecom Sud-Paris and to my fellows whose friendly company, during my thesis work, was a source of great pleasure.

Finally, I want to dedicate this thesis to my parents, Habib and Sakina, who have given me the chance of a good education, and so much love and support over the years. I probably owe them much more then they think. I am deeply and forever indebted to them for their love, support and encouragement.

For my parents,
in gratitude for their patience and love

Résumé

Le Cloud Computing est un paradigme émergent dans les technologies de l'information. L'un de ses atouts majeurs étant la mise à disposition des ressources fondée sur le modèle pay-as-you-go. Les ressources Cloud se situent dans un environnement très dynamique. Cependant, chaque ressource provisionnée offre des services fonctionnels et peut ne pas offrir des services non fonctionnels tels que la supervision, la reconfiguration, la sécurité, etc. Dans un tel environnement dynamique, les services non fonctionnels ont une importance critique pour le maintien du niveau de service des ressources ainsi que le respect des contrats entre les fournisseurs et les consommateurs. Dans notre travail, nous nous intéressons à la supervision, la reconfiguration et la gestion autonomique des ressources Cloud. En particulier, nous mettons l'accent sur les applications à base de services. Ensuite, nous poussons plus loin notre travail pour traiter les ressources Cloud d'une manière générale. Par conséquent, cette thèse contient deux contributions majeures. Dans la première contribution, nous étendons le standard SCA (Service Component Architecture) afin de permettre l'ajout de besoins en supervision et reconfiguration à la description des composants. Dans ce contexte, nous proposons une liste de transformations qui permet d'ajouter automatiquement aux composants des facilités de supervision et de reconfiguration, et ce, même si ces facilités n'ont pas été prévues dans la conception des composants. Ceci facilite la tâche au développeur en lui permettant de se concentrer sur les services fonctionnels de ses composants. Pour être en conformité avec la scalabilité des environnements Cloud, nous utilisons une approche basée sur des micro-conteneurs pour le déploiement de composants. Dans la deuxième contribution, nous étendons le standard OCCI (Open Cloud Computing Interface) pour ajouter dynamiquement des facilités de supervision et de reconfiguration aux ressources Cloud, indépendamment de leurs niveaux de service. Cette extension implique la définition de nouvelles Ressources, Links et Mixins OCCI pour permettre d'ajouter dynamiquement des facilités de supervision et de reconfiguration à n'importe quelle ressource Cloud. Nous étendons par la suite nos deux contributions de supervision et reconfiguration afin d'ajouter des capacités de gestion autonomique aux applications SCA et ressources Cloud. Les solutions que nous proposons sont génériques, granulaires et basées sur les standards *de facto* (i.e., SCA et OCCI). Dans ce manuscrit de thèse, nous décrivons les détails de nos implémentations ainsi que les expérimentations que nous avons menées pour l'évaluation de nos propositions.

Mots-clés: Cloud Computing, Supervision, Reconfiguration, Autonomic Computing, Composant, Service.

Abstract

Cloud Computing is an emerging paradigm in Information Technologies (IT). One of its major assets is the provisioning of resources based on pay-as-you-go model. Cloud resources are situated in a highly dynamic environment. However, each provisioned resource comes with functional properties and may not offer non functional properties like monitoring, reconfiguration, security, accountability, etc. In such dynamic environment, non functional properties have a critical importance to maintain the service level of resources and to make them respect the contracts between providers and consumers. In our work, we are interested in monitoring, reconfiguration and autonomic management of Cloud resources. Particularly, we put the focus on Service-based applications. Afterwards, we push further our work to treat Cloud resources. Consequently, this thesis contains two major contributions. On the first hand, we extend Service Component Architecture (SCA) in order to add monitoring and reconfiguration requirements description to components. In this context, we propose a list of transformations that dynamically adds monitoring and reconfiguration facilities to components even if they were designed without them. That alleviates the task of the developer and lets him focus just on the business of his components. To be in line with scalability of Cloud environments, we use a micro-container based approach for the deployment of components. On the second hand, we extend Open Cloud Computing Interface standards to dynamically add monitoring and reconfiguration facilities to Cloud resources while remaining agnostic to their level. This extension entails the definition of new Resources, Links and Mixins to dynamically add monitoring and reconfiguration facilities to resources. We extend the two contributions to couple monitoring and reconfiguration in order to add self management capabilities to SCA-based applications and Cloud resource. The solutions that we propose are generic, granular and are based on the *de facto* standards (i.e., SCA and OCCI). In this thesis manuscript, we give implementation details as well as experiments that we realized to evaluate our proposals.

Keywords: Cloud Computing, Monitoring, Reconfiguration, Autonomic Computing, Component, Service.

List of Publications

[1] "An Autonomic Approach to Manage Elasticity of Business Processes in the Cloud", Mohamed Mohamed, Mourad Amziani, Djamel Belaïd, Samir Tata and Tarek Melliti, Future Generation Computer Systems published by Elsevier, October, 2014

[2] "An approach for Monitoring Components Generation and Deployment for SCA Applications", Mohamed Mohamed, Djamel Belaïd and Samir Tata, in "Communications in Computer and Information Science" published by Springer-Verlag, September, 2014

[3] "Monitoring and Reconfiguration for OCCI Resources", Mohamed Mohamed, Djamel Belaïd and Samir Tata, CloudCom 2013, Bristol, UK,December 2-5, 2013

[4] "PaaS-independent Provisioning and Management of Applications in the Cloud", Mohamed Sellami, Sami Yangui, Mohamed Mohamed and Samir Tata, CLOUD 2013, Santa Clara Marriott, CA, USA, June 27-July 2, 2013

[5] "Self-Managed Micro-Containers for Service-Based Applications in the Cloud", Mohamed Mohamed, Djamel Belaïd and Samir Tata, WETICE 2013, Hammamet, Tunisia,June 17-20, 2013

[6] "Monitoring of SCA-based Applications in the Cloud", Mohamed Mohamed, Djamel Belaïd and Samir Tata, CLOSER 2013, Aachen, Germany, Mai 8-10, 2013

[7] "Adding Monitoring and Reconfiguration Facilities for Service-based Applications in the Cloud", Mohamed Mohamed, Djamel Belaïd and Samir Tata, AINA 2013, Barcelona, Spain, March 25-28, 2013

[8] "How to Provide Monitoring Facilities to Services when they are Deployed in the Cloud?", Mohamed Mohamed, Djamel Belaïd and Samir Tata, CLOSER 2012, Porto, Portugal, April 18-21, 2012

[9] "Scalable Service Containers", Sami Yangui, Mohamed Mohamed, Samir Tata and Samir Moalla, CloudCom 2011, Athenes, Greece, November 29 - December 1, 2011.

[10] "Service micro-container for service-based applications in Cloud environments", Mohamed Mohamed, Sami Yangui, Samir Moalla and Samir Tata, in WETICE 2011, Paris, France, June 27-29, 2011.

Contents

Chapter 1 Introduction **1**

1.1 General Context 1
1.2 Motivation And Problem Description 2
1.3 Research Goals 3
1.4 Approach 4
1.5 Thesis Structure 6

Chapter 2 Background

2.1 Introduction 7
2.2 Service Oriented Architecture and Service Component Architecture 8
2.3 Monitoring, Reconfiguration and Autonomic Computing 9
2.3.1 Monitoring 9
2.3.2 Reconfiguration 10
2.3.3 Autonomic Computing 11
2.4 Cloud Computing 12
2.5 Open Cloud Computing Interface (OCCI) 14
2.6 Conclusion 16

Chapter 3 State of the Art

3.1 Introduction 17
3.2 Monitoring Related Work 17
3.3 Reconfiguration Related Work 24
3.4 Autonomic Computing Related Work 28
3.5 Synthesis of Related Work 34
3.6 Conclusion 36

Chapter 4 Monitoring and Reconfiguration of SCA-based applications in the Cloud

4.1 Introduction . . . 37
4.2 Adding Monitoring and Reconfiguration facilities to component . . . 38
4.2.1 Extended SCA Model . . . 38
4.2.2 GenericProxy Service: . . . 42
4.2.3 Reconfiguration . . . 42
4.2.4 Monitoring by Polling . . . 42
4.2.5 Monitoring by Subscription . . . 43
4.2.6 Transformation Example: . . . 45
4.3 Monitoring and Reconfiguration within Scalable Micro-containers . . . 47
4.3.1 Deployment Framework and generated Micro-Container . . . 47
4.3.2 Adding Monitoring and Reconfiguration to Micro-Containers . . . 48
4.4 Adding FCAPS properties to components . . . 49
4.4.1 Extension of Deployment Framework for FCAPS management . . . 50
4.4.2 IMMC Anatomy . . . 51
4.4.3 FCAPS Manager and IMMC basic functionalities . . . 52
4.4.4 FCAPS Management . . . 53
4.5 Conclusion . . . 58

Chapter 5 Monitoring and Reconfiguration of Cloud Resources

5.1 Introduction . . . 59
5.2 Monitoring and Reconfiguration for Cloud Resources . . . 60
5.2.1 Adding Monitoring and Reconfiguration facilities to Cloud Resources . . . 60
5.2.2 OCCI description for Monitoring and Reconfiguration . . . 61
5.2.3 Example of using OCCI Monitoring and Reconfiguration extension . . . 67
5.3 Autonomic Management for Cloud Resources . . . 67
5.3.1 MAPE: Monitor, Analyze, Plan and Execute . . . 68
5.3.2 MAPE for Cloud Resources . . . 68
5.3.3 OCCI extension for Autonomic Management . . . 70
5.4 Autonomic Infrastructure for Cloud Resources . . . 77
5.5 Adding FCAPS Management to Cloud Resources . . . 78
5.6 Conclusion . . . 79

Chapter 6 Evaluation and Validation

6.1 Introduction . . . 81
6.2 Common Environment description . . . 82
6.3 Evaluation of Monitoring and Reconfiguration Approach of SCA-based applications in the Cloud . . . 82
6.3.1 Implementation . . . 82
6.3.2 Evaluation . . . 83
6.4 Evaluation of Autonomic Computing Approach for Cloud Resources . . . 85
6.4.1 Background . . . 85
6.4.2 Implementation . . . 94
6.4.3 Evaluations . . . 97
6.5 Conclusion . . . 105

Chapter 7 Conclusion and Perspectives

7.1 Contributions . . . 107
7.2 Perspectives . . . 108

Bibliography

List of Figures

2.1 SOA architecture . . . 8
2.2 Overview of the Standard Service Component Architecture Model. . . . 9
2.3 Autonomic control loop [1] . . . 11
2.4 Cloud Computing Conceptual Reference Model defined by NIST. . . . 13
2.5 OCCI's place in a provider's architecture [2]. . . . 14
2.6 UML Diagram of OCCI Core Model. . . . 15
2.7 Overview Diagram of OCCI Infrastructure Types. . . . 16

3.1 Nagios Monitoring plugin [3]. . . . 18
3.2 Proposed Monitoring information model [4]. . . . 19
3.3 Proposed Monitoring Architecture [4]. . . . 19
3.4 Architecture of Ganglia [5]. . . . 20
3.5 Structure of GRM [6]. . . . 20
3.6 Architecture of GridRM [7]. . . . 21
3.7 GMA and R-GMA components [8]. . . . 23
3.8 Support for dynamic reconfiguration [9]. . . . 25
3.9 Adding reconfiguration to a component in .Net platforms [10]. . . . 27
3.10 Autonomic Elements and their interactions [11]. . . . 29
3.11 Autonomic Control Loop [12]. . . . 30
3.12 System architecture for autonomic Cloud management [13]. . . . 30
3.13 Rainbow framework abstract model [14]. . . . 31
3.14 StarMX high level static architecture [15]. . . . 32
3.15 Overview of the CAPPUCINO distributed runtime [16]. . . . 32
3.16 Adding reconfiguration to an SCA component [17]. . . . 33

4.1 Extended SCA with monitoring and reconfiguration artifacts. . . . 39
4.2 Extended Component with Required Property. . . . 40
4.3 Component-based Application using required property. . . . 41
4.4 Definition of the GenericProxy interface. . . . 42
4.5 Reconfiguration and monitoring by polling. . . . 43
4.6 Monitoring by subscription multi Notification Service. . . . 44
4.7 Monitoring by subscription one Notification Service. . . . 44

4.8 Monitoring by subscription using hybrid schema. . . . 45
4.9 Extension of the Micro-container architecture with monitoring. . . . 48
4.10 Extended Deployment Framework and Micro-Container for FCAPS management. . . . 51
4.11 Intelligent Managed Micro Container Anatomy . . . 52
4.12 Replication and Load balancing scenario. . . . 54
4.13 Migration scenario. . . . 55

5.1 Reconfiguration extension for Cloud Resources. . . . 61
5.2 Monitoring by Polling extension for Cloud Resources. . . . 61
5.3 Monitoring by Subscription extension for Cloud Resources. . . . 62
5.4 OCCI Mixins for Monitoring and Reconfiguration. . . . 62
5.5 OCCI Links for Monitoring and Reconfiguration. . . . 64
5.6 Example of using our extension. . . . 67
5.7 Autonomic control loop for a Cloud resource . . . 68
5.8 Autonomic infrastructure for a Cloud resource . . . 69
5.9 OCCI Core Model and our extension (colored boxes) for Autonomic Computing. . . . 71
5.10 Autonomic Computing Infrastructure establishment for Cloud resources. . . . 77

6.1 Memory consumption of different types of micro-containers. . . . 84
6.2 Latency time using different scenarios of monitoring. . . . 84
6.3 Overview of the defined OCCI platform types. . . . 86
6.4 Overview of the defined OCCI application types. . . . 87
6.5 BPMN of a SBP example . . . 89
6.6 Petri net of a SBP example . . . 89
6.7 Example of the elasticity of a SBP . . . 92
6.8 HLPN model of the generic Controller . . . 93
6.9 Response time before and after adding our Autonomic Infrastructure. . . . 100
6.10 Memory consumption before and after adding our Autonomic Infrastructure. . . . 101
6.11 Autonomic Computing Infrastructure establishment for SBP elasticity. . . . 103
6.12 Response time of the SBP before and after adding our Autonomic Infrastructure. . . . 104
6.13 Memory consumption before and after adding our Autonomic Infrastructure. . . . 105

List of Tables

3.1 Synthesis of Related Work. . . . 35

5.1 Definition of the *Subscription Link* . . . 64
5.2 Actions defined for the Subscription Link . . . 64
5.3 Definition of the *Notification Link* . . . 65
5.4 Actions defined for the Notification Link . . . 65
5.5 Definition of the *Action Link* . . . 66
5.6 Actions defined for the Action Link . . . 66
5.7 Definition of the Autonomic Manager . . . 72
5.8 Actions defined for the Autonomic Manager resource . . . 72
5.9 Definition of the Analyzer . . . 73
5.10 Actions defined for the Analyzer resource . . . 73
5.11 Definition of the Planner . . . 74
5.12 Actions defined for the Planner resource . . . 74
5.13 Definition of the *Agreement Link* . . . 75
5.14 Actions defined for the Agreement Link . . . 75
5.15 Definition of the *Alert Link* . . . 76
5.16 Actions defined for the *Alert Link* . . . 76

6.1 Summary of the monitoring and reconfiguration Mixins. . . . 97
6.2 Description of monitoring metrics, thresholds, alerts and reconfigurations of our use case. 99
6.3 Average times needed to reconfiguration actions. . . . 100

Chapter 1

Introduction

Contents

1.1 General Context 1
1.2 Motivation And Problem Description 2
1.3 Research Goals 3
1.4 Approach 4
1.5 Thesis Structure 6

1.1 General Context

Over the last years, there has been an enormous shift in Information Technologies (IT) to Cloud Computing. Cloud Computing is a recent paradigm enabling an economic model for virtual resources provisioning. It refers to a model for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal effort or service provider interaction [18]. In this paradigm, there are basically three layers of services known as "IaaS" for Infrastructure as a Service, "PaaS" for Platform as a Service and "SaaS" for Software as a Service. Adoption of Cloud Computing is increasing due its economic model based on *pay-as-you-go* model. A Gartner survey [19] realized on august 2013 on the future of IT services shows that 38 percent of all organizations surveyed indicate cloud services use today. However, 80 percent of organizations said that they intend to use cloud services in some form within 12 months, including 55 percent of the organizations not doing so when the survey was performed.

As it is, the Cloud is well adapted to host Service-based applications that follow Service Oriented Architecture (SOA). SOA is a software architecture based on services which communicate with each other. Such type of applications can be described using Service Component Architecture (SCA) as a composite that contains a detailed description for different components of the application and links between them. All the elements in a composite must be described as one of the standard artifacts of the

SCA meta-model. A well described composite can be transmitted to a SCA runtime (e.g. FraSCAti [20], TUSCANY [21]) that instantiates the different components and links them as described in the composite.

However, SCA-based applications in the Cloud are exposed to dynamic evolution during their life-cycle. This evolution is due to the environment dynamic. Consequently, management of SCA-based applications in Cloud environments is becoming a challenging task. Therefore, monitoring and reconfiguration can play an important role to respect the agreements between the service's consumer and its provider. Monitoring process consists on informing the interested parts (users or application components) about the changes of the monitored system properties by notifying them on a regular way or whenever a change has occurred. While reconfiguration is a runtime modification of the structure or the implementation of an component [22]. In order to remain efficient and accurate against the environment dynamic, Cloud providers should enable dynamic monitoring and reconfiguration of the provisioned resources with a minimal cost and a minimal performance degradation. Over and above providing the tools of monitoring and reconfiguration, coupling these two functionalities has a big importance to cope with the dynamic evolution of Cloud environments [13]. This coupling is possible by means of autonomic loops [11]. An autonomic loop consists on harvesting monitoring data, analyzing them and generating reconfigurations to correct violations (self-healing and self-protecting) or to target a new state of the system (self-configuring and self-optimizing). Using these loops we can build Autonomic Computing systems. Autonomic Computing [11] is the ability to manage computing resources automatically and dynamically to respond to the requirements of the business based on SLA.

1.2 Motivation And Problem Description

Indeed, when applied to a SCA-based applications in a Cloud environment, monitoring becomes a complicated problem that has to face many challenges. First, the description of the need to consume monitoring information must be explicitly described in different granularities independently of the type of the components (i.e., Network, Compute, Storage, Software, etc.). Moreover, the nature of monitored components and the way their status should be retrieved depends on the component being monitored. This dependency along with the absence of standardization efforts render the monitoring more complicated. Finally, any monitoring solution must respect the scalability of the Cloud and should minimize the resources and energy consumption to be in-line with the economic model of this paradigm based on pay-as-you-go.

In the same context, reconfiguration is becoming more and more complicated since there is no standard description for reconfiguration requirements. Since reconfiguration is not the basic function of resources, not all the resources are designed with reconfiguration facilities. Consequently, we can face a problem if a component would like to reconfigure a given resource that do not offer any interface for that matter. Furthermore, components may require to reconfigure each other in different granularities. For example, a given component may require to reconfigure another component by changing one of its attributes, it can also require to stop the component or to delete many components. Then, the way of applying reconfiguration actions on component depends on the granularity of the action and the reconfigured component. This task is really sophisticated if we do not have enough information about the component.

Many attempts to provide monitoring and reconfiguration for applications in the Cloud exist in the state of the art detailed in Chapter 3, but as we will explain, almost all the proposed solutions give tooling solutions to monitor and reconfigure Cloud applications behavior. It is worth noting that the cited works do not target monitoring and reconfiguration at different granularities. Furthermore, there is no approach that expects to monitor or reconfigure components that were not designed with monitoring and reconfiguration facilities. In addition, almost all of the existing solutions either do not take care of scalability issue, or do not include an efficient solution to that problem.

As explained in Section 1.1, coupling monitoring and reconfiguration in a holistic loop is so important. We advocate that making monitoring and reconfiguration collaborate in an autonomic loop is really interesting since it allows to rapidly tackle violations and keep the cloud environment in a safe state. Moreover, coupling Autonomic Computing with Cloud computing increases the availability of Cloud resources and reduces their costs. However, this coupling remains critical and challenging according to Buyya et al. [13] since Cloud environments are composed of thousands of interconnected heterogeneous resources.

This coupling adds autonomic management behaviors to Cloud resources and particularly to applications. However, Cloud providers did not yet succeed to enable this kind of coupling. In contrast, they provide interfaces or APIs to manually reconfigure the behavior of the application or their containers from outside. That could tackle the resiliency of the application specially if the container is situated in an environment with a low bandwidth, in this case, some decisions are critical and must be as fast as possible. In our point of view, the fastest way is to take decisions from the closest location to the resource itself.

For autonomic management in a Cloud environment, we need to specify in a granular way what are the targets of monitoring and reconfiguration. Going from a simple attribute to a complex system. This issue brings another major challenge to autonomic computing which is the heterogeneity of the resources that need to be monitored and reconfigured in an autonomic loop. This problem is basically due to the fact that monitoring and reconfiguration are treated as separate aspects. Almost all the time, the resources used to these aims are not compatible with one another. The advantage brought by autonomic management is to use monitoring data to apply reconfiguration actions to ensure self configuration, self healing, self optimization, and self protecting.

Actually, almost all of the cloud computing projects suppose that monitoring is used just to gather Key Performance Indicator (KPI) for billing or to notify the interested client of eventual problems [23, 24, 25, 26]. They also consider that reconfiguration consists of a new deployment of resources [23, 24, 25, 26]. Therefore, reconfiguration is as expensive as a new deployment. The few other works concerning autonomic management cope just with specific use cases. Thus, they can not respond to the dynamic of the cloud in different situations.

1.3 Research Goals

The objectives of this thesis are to provide solutions for monitoring and reconfiguration of service-based applications in Cloud environments. However, the different solutions and mechanisms could be used separately or combined in a holistic loop providing autonomic management facilities. For example, using

our monitoring mechanisms, an interested part can retrieve monitoring data from a given component or application deployed in the Cloud. It can also apply reconfigurations to change the behavior of a given component or application. Moreover, it can retrieve monitoring data, analyze them and generates reconfigurations to be applied on one or more components.

The expected characteristics of the provided solutions are the following:

- Genericity: The description of monitoring and reconfiguration requirements must be generic and agnostic to the resource. Consequently, this description could be applied on heterogeneous resources without major modifications;
- Granularity: The proposed solution must enable monitoring and reconfiguring components at different granularities (i.e., attribute, component, application, etc.). Therefore, the proposed interfaces may allow an interested part to monitor or reconfigure different aspects of Cloud resources;
- Standardization: The proposed mechanisms should be conform to the *de facto* standards whenever it is possible (e.g., SCA, OCCI, REST, HTTP, etc.). Using standards will enforce the conformance of our solution to the requirements of monitoring and reconfiguration;
- Scalability: The solution must be in line with the scalability of Cloud environments. Using communication mechanisms known with their scalability can improve the overall scalability of the solution. We need also to use scalable mechanisms for applications deployment in the Cloud;
- Extensibility: The proposed solution should be extensible in order to enable adding new modules to respond to new requirements;
- Flexibility: In order to respond to different flavors, the user of our solution may be able to choose different ways to deploy and use the defined mechanisms and resources.

It is noteworthy that the proposed work in this thesis needs to be validated and evaluated. The implementation aspects should be detailed. Furthermore, evaluation scenarios should be described and experiments should be performed and analyzed.

1.4 Approach

In order to resolve the previously described problems, we propose different solutions to add monitoring and reconfiguration facilities to resources in Cloud environments.

First, we propose an extension for SCA meta-model to enable a generic description of a component's needs to monitor other components. This extension enables monitoring SCA-based applications at different granularities. And to take into account the case where components are designed without monitoring facilities, we propose different transformations to render a component monitorable. Then, another extension to SCA is proposed to enable generic description of component's needs to reconfigure other components. The description is granular so that it enables reconfiguring attributes, component or application composites. We also propose transformation mechanisms to render reconfigurable a given component.

Moreover, we propose a Deployment Framework able to add the described monitoring and reconfiguration requirements to components prior deploying them in the Cloud. This framework deploys components based on scalable micro-containers. These latter guaranty the scalability of the SCA-based application once deployed in the cloud.

Furthermore, in order to integrate the usage of monitoring and reconfiguration and to add self management aspects to SCA-based applications, we extend our work to describe FCAPS (i.e., Fault, Configuration, Accountability, Performance and Security) aspects for components. We extend the Deployment Framework to render it able to add monitoring and reconfiguration facilities to SCA components and encapsulates them in self-managed micro containers with a high scalability and mobility. These micro-containers have a high resiliency since they implement different transactions (i.e., replication, reconfiguration, recombination and repairing) and provide FCAPS management. Using their monitoring services, these Intelligent Managed Micro Containers (IMMCs) have the ability to dynamically and transparently face the changes of their environments. This is supported by triggering reconfiguration or migration mechanisms or by changing any of the strategies used for the FCAPS management.

Over and above SCA-based applications, we propose to extend our work in order to be independent from the service level (i.e., IaaS, PaaS or SaaS). To this aim, we propose a generic description of monitoring and reconfiguration of Cloud resources. To resolve heterogeneity problems, the most trivial way is to use standard interfaces. For Cloud Computing, the *de facto* standard is Open Cloud Computing Interface (OCCI). OCCI is defined by the Open Grid Forum as *"an abstraction of real world resources, including the means to identify, classify, associate and extend those resources"* [2]. Using OCCI standard allows us also to propose a granular description for monitoring and reconfiguration requirements. The extension mechanism of OCCI, based on the usage of Mixins, enables adding new functionalities easily. The rendering of the infrastructure, based on REST interfaces using HTTP [27], enforces the scalability of the proposed solution.

Afterwards, we push farther our work and we propose an *on demand* autonomic computing infrastructure based on OCCI standard. This infrastructure is based on new OCCI Entities (i.e., Resources and Links) and Mixins. In order to add autonomic computing facilities to resources, we propose to add Mixins to enable the following functionalities: (1) to inspect a specific SLA related to resources to be able to define the needed elements for the autonomic loop, (2) to use our monitoring mechanisms to extract data from resource, and (3) to enable reconfigurations on concerned resources based on our reconfiguration mechanisms. The rest of the infrastructure consumes monitoring data in order to analyze it and generate reconfiguration actions. The established infrastructure needs eventually some Mixins for specific processing (i.e., adding specific analysis rules and reconfiguration strategies, etc.). These Mixins could be adapted at runtime to change the behavior of the system. In this manuscript we do not cover all the self-CHOP (configuring, healing, optimizing and protecting) properties. Instead, we propose the needed mechanisms that could be used to cover these properties.

It is worthy to note, that for each of the proposed approaches, we will detail the different aspects of the realized implementations. We will also describe the realized scenarios and experiments performed to evaluate our proposals.

1.5 Thesis Structure

The remainder of this manuscript is organized as follows. Chapter 2 contains a description to the different concepts needed for the understanding of the rest of the work. In this chapter, we give the definitions of Service-based and SCA-based applications that we will target in our work. We also define the management aspects that we are interested in, followed by a definition of Cloud Computing and an overview of OCCI standard. Furthermore, we give an overview of the state of the art related to monitoring, reconfiguration as well as autonomic management in Chapter 3. Therein, we will cite the different works in each area with a reasoning to show the advantages that we would like to have in our solution and the drawbacks that we would like to escape. In Chapter 4, we present our proposals to add monitoring, reconfiguration as well as autonomic management to SCA-based applications in Cloud environments. The proposal entails the description model, the deployment mechanisms as well as different transformations proposed to render component monitorable and reconfigurable. Furthermore, we push farther our work in Chapter 5. In this chapter, we propose to dynamically add monitoring, reconfiguration and autonomic management mechanisms to Cloud resources while remaining level agnostic. In Chapter 6, we present the different aspects of evaluation of our proposals. It details the implementation aspects, the evaluation environment and scenarios as well as the results and findings. Finally, we sum up our contributions in Chapter 7. We conclude this last chapter with some perspectives that we aim to realize at short, medium and long term.

Chapter 2

Background

Contents

2.1 Introduction **7**

2.2 Service Oriented Architecture and Service Component Architecture **8**

2.3 Monitoring, Reconfiguration and Autonomic Computing **9**

2.3.1 Monitoring 9

2.3.2 Reconfiguration 10

2.3.3 Autonomic Computing 11

2.4 Cloud Computing **12**

2.5 Open Cloud Computing Interface (OCCI) **14**

2.6 Conclusion **16**

2.1 Introduction

In this work we aim to provide a generic solution for monitoring and reconfiguration of Service-based applications in cloud environments. In order to understand the remainder of this manuscript, one should has a basic knowledge on different paradigms and concepts. We dedicate this chapter to briefly introduce these basics to the reader. To do so, we will start by introducing the Service Oriented and Service Component architectures that represent application architectures that we basically target. Afterwards, we will define the different non functional aspects that we aim to deal with in this work resumed in monitoring, reconfiguration and autonomic computing. Finally, we will present the environment of our work which is Cloud Computing as well as the *de facto* standard in the Cloud known as Open Cloud Computing Interface (OCCI).

2.2 Service Oriented Architecture and Service Component Architecture

Service-Oriented Architecture (SOA) is the outcome of the Web services developments and standards in support of automated business integration [28] [29]. The purpose of this architecture style is to address the requirements of loosely coupled, standards-based, and protocol-independent distributed computing. As defined by Papazoglou [30], *SOA is a logical way of designing a software system to provide services to either end user applications or other services distributed in a network through published and discoverable interfaces*.

The main building blocks in SOA are services. Services are self-describing components that support rapid, low-cost development and deployment of distributed applications. Thus, using SOA, applications are defined as a composition of re-usable software services, which implement the business logic of the application domain.

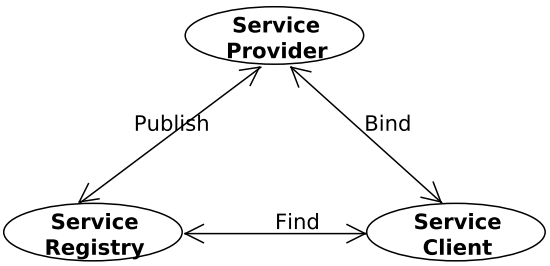


Figure 2.1: SOA architecture

The SOA architectural style is structured around the three basic actors depicted in Figure 2.1: Service Provider, Service Client and Service Registry while the interactions between them involve the publish, find and bind operations. Service Provider is the role assumed by a software entity offering a service. Service Client is the role of a requestor entity seeking to consume a specific service. However, Service Registry is the role of an entity maintaining information on available services and the way to access them.

The benefit of this approach lies in the loose coupling of the services making up an application. Services are provided by platform independent parts, implying that a client using any computational platform, operating system and any programming language can use the service. While different Service Providers and Service Clients may use different technologies for implementing and accessing the business functionality, the representation of the functionalities on a higher level (services) is the same. Therefore, it should be interesting to describe an application as a composition of services in order to be implementation-agnostic. This allows the separation of the business functionality from its implementation. Hence, an application can be executed by composing different services provided by heterogeneous parts with respect to their services descriptions.

OASIS describes Service Component Architecture (SCA) [31] as a programming model for building applications and solutions based on SOA. One basic artifact of SCA is the component, which is the unit of construction for SCA (see Figure 2.2). *"A component consists of a configured instance of a piece of code providing business functions. It offers its functions through service-oriented interfaces and*

may require functions offered by other components through service-oriented interfaces as well. SCA components can be implemented in Java, C++, and COBOL or as BPEL processes. Independent of whatever technology is used, every component relies on a common set of abstractions including services, references, properties and bindings" [31].

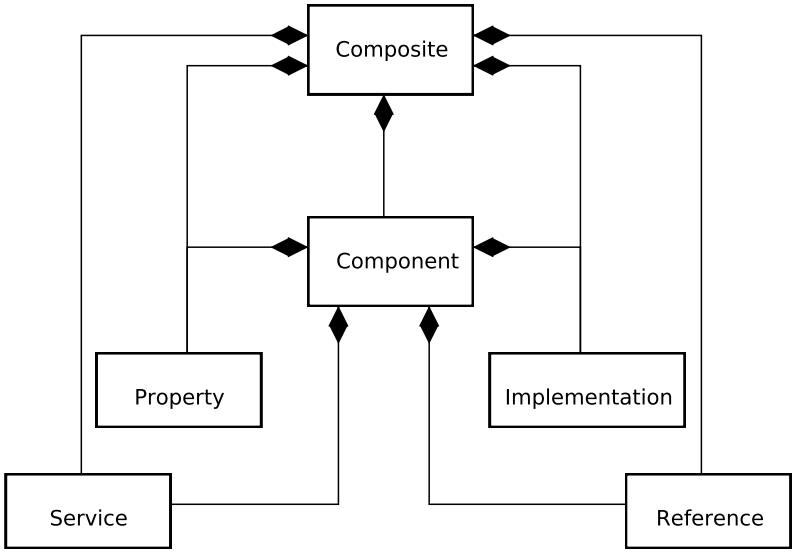


Figure 2.2: Overview of the Standard Service Component Architecture Model.

As shown in Figure 2.2, a service describes what a component provides, i.e., its external interface. A reference specifies what a component needs from the other components or applications of the outside world. Services and references are matched and connected using wires or bindings. A component also defines one or more properties [31] that allow their reconfiguration.

2.3 Monitoring, Reconfiguration and Autonomic Computing

2.3.1 Monitoring

Monitoring consists of informing interested parts of the status of a property or a service. Monitoring is important for both, service provider and service consumer. It is a key tool for controlling and managing hardware and software infrastructures. Indeed, monitoring provides key performance indicators for applications and platforms. It involves many activities such as resource planning and management, SLA management, billing, troubleshooting and security management.

In our work ([32], [33]), we consider two models of monitoring: monitoring by polling or by subscription. Polling is the simpler way of monitoring, as it allows the observer to request the current state of a property whenever there is a need. The interested part can generally interact with a specific interface

that provides a getter of the needed property. Monitoring by subscription model is based on a publish/-subscribe system which is defined as a set of nodes divided into publishers and subscribers. Subscribers express their interests by subscribing for specific notifications independently of the publishers. Publishers produce notifications that are asynchronously sent to subscribers whose subscriptions match these notifications [34]. Subscription allows an observing resource to be notified about changes of monitored properties using one of the following modes: (1) The subscription on interval implies that the publisher (producer) broadcasts the state of its properties periodically to the subscribers (consumers) and (2) The subscription on change implies that the publisher has to notify the subscribers whenever there are new monitoring information related to properties change, method calls or service time execution. The subscription on change concerns different aspects, in our work [33] we suppose various types of monitoring:

- **Property Changed Monitoring (PCM):** the monitored component has to send notifications to all subscribers whenever a monitored property is changed,
- **Method Call Monitoring (MCM):** the monitored component sends notifications whenever one of the service's methods is invoked,
- **Execution Time Monitoring (ETM):** the monitored component notifies the subscribers about the execution time whenever a service invocation occurred.

2.3.2 Reconfiguration

Reconfiguration is a runtime modification of an infrastructure or a system [22]. This modification can take the system from a current state (described by its properties, location, dependencies, implementations, etc.) to another objective state. It consists in readjusting the internal structure of the system without changing its main function, which is a recombination process of certain targets in a certain way. It can refer to making a recombination of the nodes topology structure in a distributed system, correcting service failures, updating existing services and system modules online, increasing and deploying new services dynamically and so on [35]. Reconfiguration is based on simple reconfiguration actions and complex ones. The usual reconfiguration simple actions are: (1) Adding/Removing a component; (2) Modification of an attribute; (3) Adding/Removing a link; and (4) Adding/Removing a resource interface. Complex actions could be represented as a combination of a set of simple actions (e.g., migration consists on adding a new component, adding a link, deleting a component and deleting a link).

In the literature, we can classify the reconfiguration in four classes:

- **Structural**: the reconfiguration leads to a change in the structure of the system (e.g., removing a component);
- **Geometric**: the reconfiguration leads to a new mapping of the existing components to new locations (e.g., migrating a component);
- **Implementation**: the reconfiguration leads to change the implementation of one or more components, but the structure of the system remains unchanged (e.g., adding an interface);
- **Behavioral**: the reconfiguration leads to a change of the behavior of one or more resources (e.g., modification of an attribute).

2.3.3 Autonomic Computing

Autonomic computing aims to manage the computing systems with decreasing human intervention. The term autonomic is inspired from the human body where the autonomic nervous system takes care of unconscious reflexes, the digestive functions of the stomach, the rate and depth of respiration and so forth [36]. Autonomic computing attempts to intervene in computing systems in a similar fashion as its biological counterpart. The term autonomic was introduced by IBM in 2001 [11]. IBM defines Autonomic Computing as the ability to manage computing resources that automatically and dynamically responds to the requirements of the business based on SLA.

Management tasks like monitoring, configuration, protection, optimization, are not the main functional objective of most applications, but if they are not properly addressed, the application cannot accomplish its task. The challenge, then, is to enable self-managing systems that take control of all these non functional tasks, letting the developers to focus on the main functional goals of applications. In order to really free developers from the burden of programming self-governing features on their applications, there must exist a way to develop these concerns independently and to integrate them with the application at some stage.

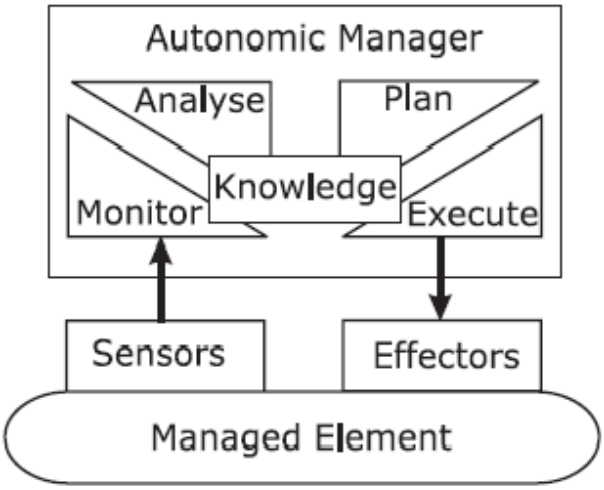


Figure 2.3: Autonomic control loop [1]

To achieve autonomic computing, IBM has suggested a reference model for autonomic control loops [1] as depicted in Figure 2.3. It is called the MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop. The central element of MAPE-K is the autonomic manager that implements the autonomic behavior. In the MAPE-K autonomic loop, the managed element represents any software or hardware resource that is coupled with an autonomic manager to exhibit an autonomic behavior. Sensors are used to collect information about the managed element, while effectors carry out changes to the managed element. Sometimes also a common element called Knowledge source is highlighted, which represents the management data that can be shared for all the phases of the control loop. Knowledge include for example, change plans, adaptation strategies, etc. Thus, based on the data collected by the sensors and the internal

knowledge, the autonomic manager will monitor the managed element and execute changes on it through effectors.

The phases of the autonomic control loop are defined as: Monitor, Analyze, Plan, and Execute.

- The Monitor function provides the mechanisms to collect, aggregate, filter and report monitoring data collected from a managed resource through sensors.
- The Analyze function provides the mechanisms that correlate and model complex situations and allow the autonomic manager to interpret the environment, predict future situations, and interpret the current state of the system.
- The Plan function provides the mechanisms that construct the actions needed to achieve a certain goal, usually according to some guiding policy or strategy.
- The Execute function provides the mechanisms to control the execution of the plan over the managed resources by means of effectors.

The use of autonomic capabilities in conjunction with SOA provides an evolutionary approach in which autonomic computing capabilities anticipate runtime application requirements and resolve problems with minimal human intervention. Regarding the requirements for monitoring and reconfiguration of SCA-based applications, the autonomic control loop cover the different aspects that we are interested in. The monitoring is essential to detect the changes of the execution environment. The reconfiguration is needed to adapt the application to its environment or to apply corrective actions on the environment itself to be well suited for the execution of the application. Consequently, in this thesis, we will give more focus on monitoring and reconfiguration as basic aspects in the autonomic loop. We will not cover the whole autonomic loop with its different aspects.

2.4 Cloud Computing

Cloud Computing is an emerging paradigm in information technology. It is defined by the National Institute of Standards and Technology [18] as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. Figure 2.4 shows an overview of Cloud Computing reference architecture viewed by NIST. It identifies the major actors, their activities and functions in the Cloud. This figure aims to facilitate the understanding of the requirements, uses, characteristics and standards of Cloud Computing.

Cloud Computing is characterized by its economic model based on "pay-as-you-go" model. This model allows a consumer to consume computing resources as needed. Moreover, resources in the Cloud are accessible over the network through standard mechanisms that promote their use by different platforms. The resources are offered to consumers using a multi-tenant model with heterogeneous resources assigned to consumer on demand. These resources are provisioned in an elastic manner that allows to

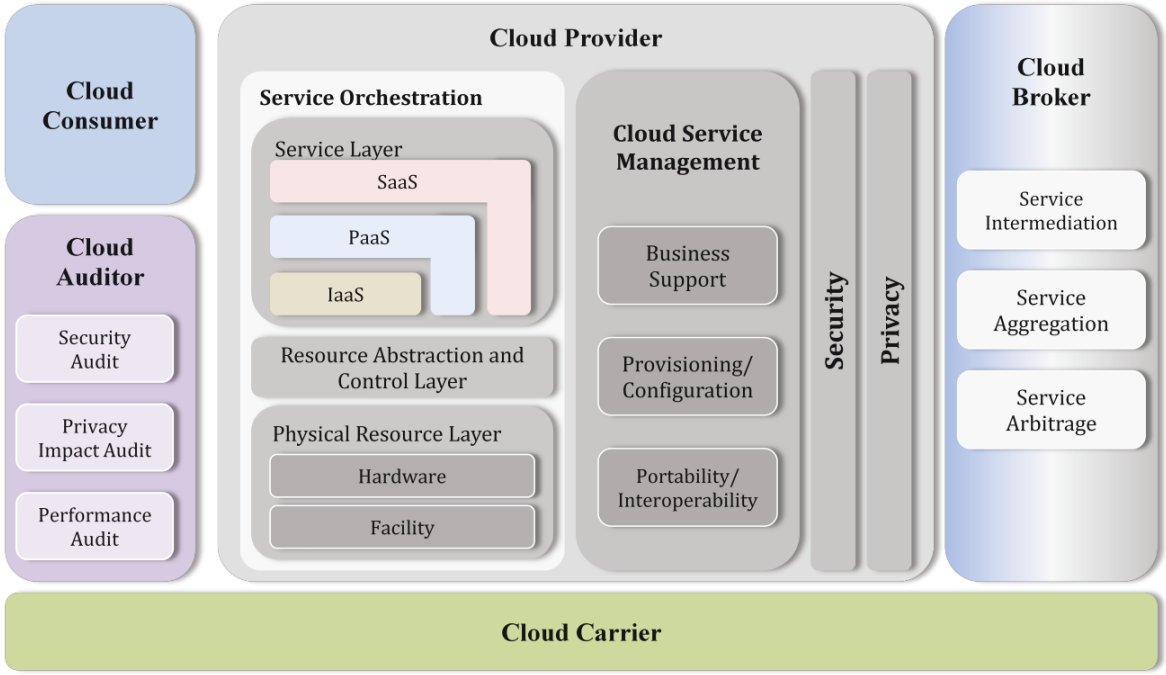


Figure 2.4: Cloud Computing Conceptual Reference Model defined by NIST.

scale up or down rapidly in line with demand. Furthermore, Cloud resources usage can be monitored and controlled in order to respect the "pay-as-you-go" model.

Services in the Cloud are basically delivered under three well discussed layers namely the Infrastructure as a Service (IaaS), the Platform as a Service (PaaS) and the Software as a Service (SaaS):

- **IaaS:** Consumers are able to access cloud resources on demand. These resources can be virtual machines, storage resources and networks. The provider takes the responsibility of installing, managing and maintaining these resources transparently.
- **PaaS:** Consumers are able to develop, deploy and manage their applications onto the cloud using the libraries, editors and services offered by the provider. The provider takes the responsibility to provision, manage and maintain the Infrastructure resources.
- **SaaS:** Consumers are able to use running applications on a IaaS or PaaS through an interface. They are not responsible of managing or maintaining the used Cloud resources.

Nowadays, more models appeared generally referred to as XaaS targeting a specific area. For example there is the DaaS for Desktop as a Service, NaaS for Network as a Service, etc.

At the same time, Clouds can be provisioned following different models according to the users needs. If the Cloud is used by a single organization, we talk about Private Cloud. In this case, this organization

owns the Cloud and is responsible of its management and maintenance. However, if the Cloud is owned by different organizations, we are talking about community or federation Cloud. Whenever the Cloud is exposed to public use, we are talking about Public Cloud. In this case, an organization owns the Cloud and manages it while it is used by other organizations. Finally, there is another model in which the cloud is composed of two or more clouds. In these Clouds, Public or Private clouds are glued together.

2.5 Open Cloud Computing Interface (OCCI)

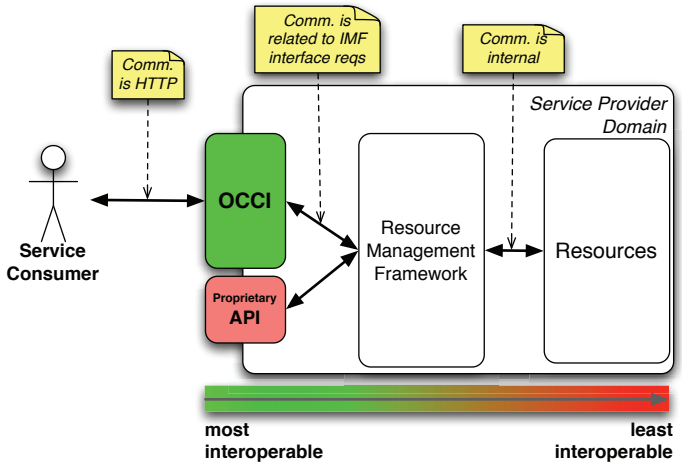


Figure 2.5: OCCI's place in a provider's architecture [2].

Open Grid Forum (OGF) defines OCCI [2] as "*an abstraction of real world resources, including the means to identify, classify, associate and extend those resources.*" OCCI provides a model to represent resources and an API to manage them. As shown in Figure 2.5, it allows providers to offer a standardized API that enables the consumption of their resources by users or other providers.

OCCI specifications consist essentially of three documents:

- **OCCI Core:** It describes the formal definition of the OCCI Core Model as seen by OGF [2].
- **OCCI HTTP Rendering:** It defines how to interact with the OCCI Core Model using the RESTful OCCI API [27].
- **OCCI Infrastructure:** It contains the definition of the OCCI Infrastructure extension for the IaaS domain [37].

OCCI Core formally describes the OCCI Core Model as shown in Figure 2.6. This Model defines a representation of instance types which can be manipulated through an OCCI rendering implementation. It is an abstraction of real-world resources, including the means to identify, classify, associate and extend those resources. This core model describes Cloud resources as instances of *Resource* or a sub-type thereof. Resources could be combined and linked to each other using instances of *Link* or a sub-type of

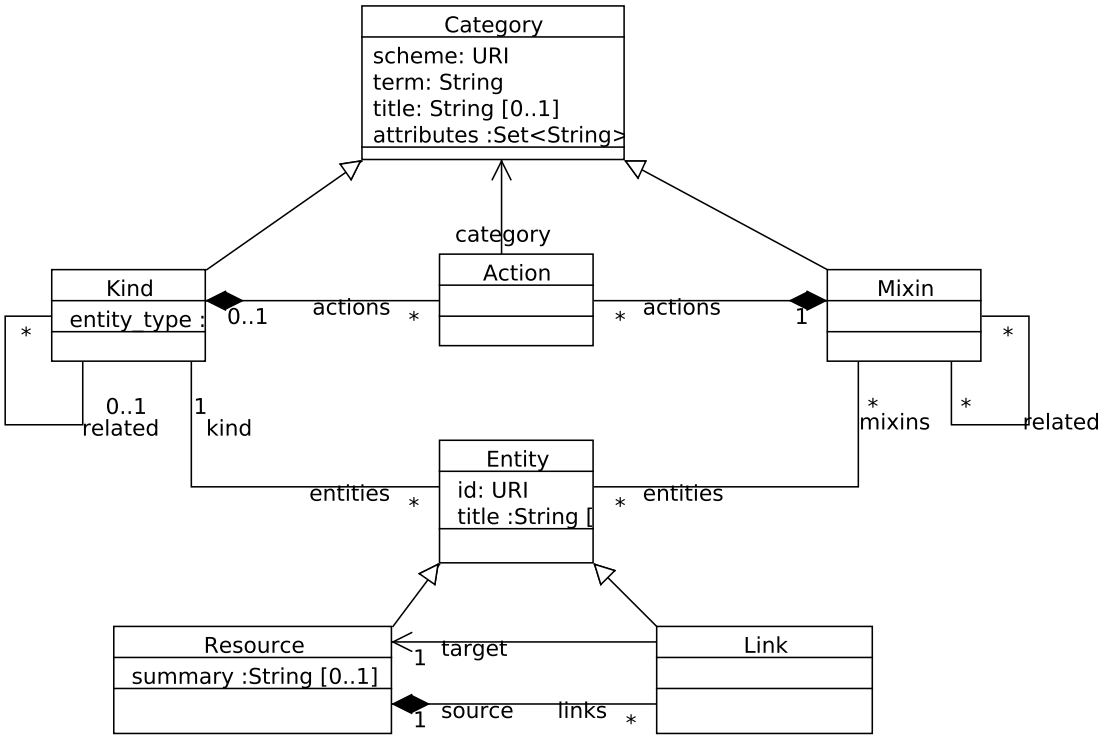


Figure 2.6: UML Diagram of OCCI Core Model.

this latter. *Resource* and *Link* are sub-types of *Entity*. As shown in Figure 2.6 [2], each *Entity* instance is typed using an instance of the class *Kind* and could be extended using one or more instances of the class *Mixin*. *Kind* and *Mixin* are subtypes of *Category* and each *Category* instance can have one or more instances of the class *Action*. A fundamental advantage of the OCCI Core Model is its extensibility. It is noteworthy that any extension will be discoverable and visible to an OCCI client at run-time. An OCCI client can connect to an OCCI implementation using an extended OCCI Core Model, without knowing anything in advance, and still be able to discover and understand, at run-time, all the instance types supported by that implementation. Extending OCCI core is possible using sub-types of the existing types or using Mixins.

The second document is OCCI HTTP Rendering [27]. It specifies how the OCCI Core Model [2], including extensions thereof, is rendered over the HTTP protocol. The document describes the general behavior for all interaction with an OCCI implementation over HTTP together with three content types to represent the data being transferred.

The OCCI model can be extended to cover different levels in the cloud (i.e., IaaS, PaaS, SaaS, etc.). And that is the purpose of the third document that is OCCI Infrastructure [37]. It is an extension of the Core Model to represent the cloud infrastructure layer. This extension brings basically the definition of new resources that inherit the core basic types *Resource* and *Link*. As shown in Figure 2.7 [37], new

Resource subclasses are *Network*, *Compute* and *Storage* while the new *Link* subclasses are *StorageLink* and *NetworkInterface*.

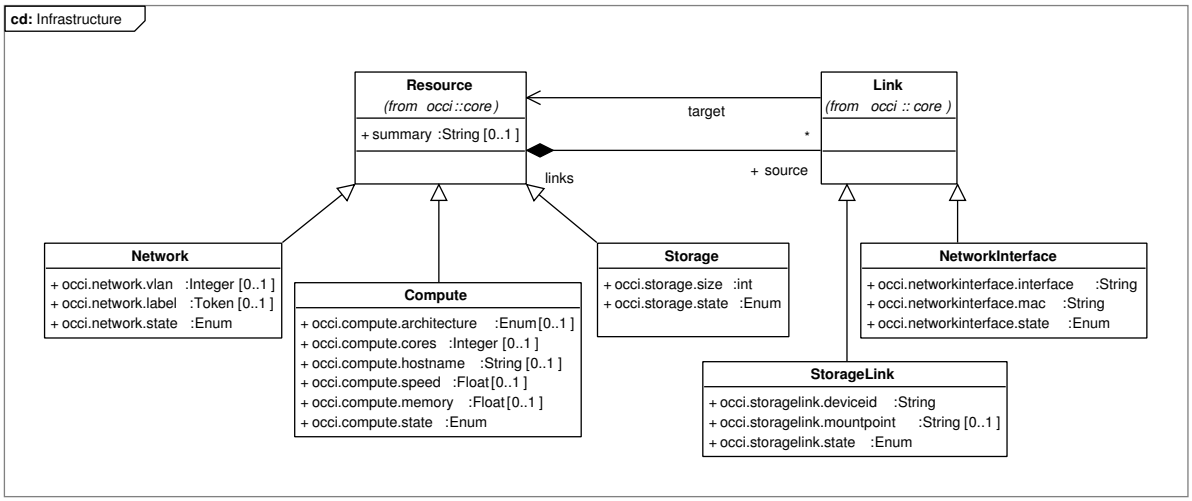


Figure 2.7: Overview Diagram of OCCI Infrastructure Types.

The *Compute* resource represents a generic information processing resource like a virtual machine. However, *Network* resource represents networking entity like a virtual switch. The last defined resource type for this extension is the *Storage* that represents resources that record information to a data storage device. The extension entails also the different interfaces that enable using the defined resources.

2.6 Conclusion

In this chapter, we introduced the basic concepts related to our work. We specified the architecture that we target namely Service Component Architecture (SCA). Afterwards, we defined the different management aspects that we aim to treat. Then, we introduced Cloud Computing as the target environment of our research. Finally, we gave an overview of OCCI that represents the *de facto* standard used in the Cloud. In the next chapter, we will go further in our investigations. We will give a detailed study of the state of the art related to the management functionalities that we are interested in.

Chapter 3

State of the Art

Contents

3.1 Introduction 17
3.2 Monitoring Related Work 17
3.3 Reconfiguration Related Work 24
3.4 Autonomic Computing Related Work 28
3.5 Synthesis of Related Work 34
3.6 Conclusion 36

3.1 Introduction

Monitoring, reconfiguration and autonomic management in the Cloud are not deeply explored. Meanwhile, there is a plethora of works dealing with these aspects long before the apparition of Cloud paradigm. In this chapter, we will present different works that treated monitoring, reconfiguration as well as autonomic management in Cloud environments or distributed systems. We will start by presenting the works treating monitoring in Section 3.2. Afterwards, we will present the reconfiguration state of the art in Section 3.3. In Section 3.4, we will present works on autonomic computing. We conclude this chapter by a synthesis of the presented works comparing them on the basis of the criteria that we dressed in our research objectives.

3.2 Monitoring Related Work

In this section, we will present a selection of works dealing with monitoring in cloud environments and in distributed systems. One should bear in mind that a plethora of works exists trying to provide monitoring solutions. However, in this manuscript we will refrain from citing all the works to highlight just a selection of them.

Nagios [3] is an open-source core system for network monitoring. It allows monitoring IT infrastructure to ensure that systems, applications and services are functioning properly. Monitoring can be applied on private or public services (private services are server properties and services while public services are services and properties available across network). To monitor any target, Nagios uses a list of plug-in that would be executed to poll the target status. Plug-ins act as an abstraction layer between the monitoring daemon and the monitored targets. As shown in Figure 3.1, a plugin enables remote command execution to know the status of the monitored target. There are several plug-ins for different protocols as SNMP (Simple Network Management Protocol), NRPE (Nagios Remote Plugin Executor) or SSH (Secure SHell) protocols. Monitoring using Nagios can result in high load on the monitoring server if applied on a large number of targets. There are different monitoring systems that are build on top of Nagios, like GridIce. This system has a main server based on Nagios that periodically queries a set of nodes to extract specific information. The collected information is stored in a DBMS and used to build aggregate statistics, trigger alerts and dynamically configure Nagios to monitor any newly discovered resource.

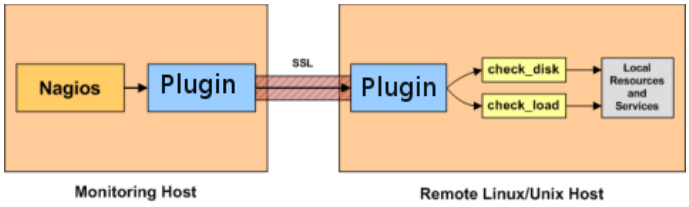


Figure 3.1: Nagios Monitoring plugin [3].

In [4], authors present the architectural design and implementation of a service framework that monitors the resources of physical as well as virtual infrastructures. The solution extends Nagios through the implementation of NEB2REST, a RESTful Event Brokering module. The implementation of this component is based on the Nagios Event Broker API (NEB) which offers the ability of handling events generated by Nagios. The proposed design realizes a twofold push and pull models: as shown in Figure 3.2 the information is pushed towards the Information Manager layer while each consumer can pull pieces of data from the Information Manager when needed. The proposed solution serves all three aforementioned layers. The Monitoring Service is a RESTful implementation. It is situated in the management layer along with the historical database. The role of the Information Provider is being fulfilled in this case by the physical or virtual infrastructure of a cloud provider. To this end, the proposed approach uses Nagios for collecting resource information from the nodes of the infrastructure, either virtual or physical as shown in Figure 3.3. Through Nagios, an administrator can define his own service-checks through which he can acquire the status of the nodes. NEB2REST component was developed to export monitoring information from the nodes to the Monitoring Service. It is a custom module that serves as a broker for the low-level monitoring infrastructure towards the Monitoring Service. For every service check realized by Nagios, the NEB2REST API exports all necessary information to the Monitoring Service through a POST HTTP method. The interaction of NEB2REST with the Monitoring Service is realized through the cURL and the libcurl APIs.

Ganglia [5] is a monitoring system for high performance computing systems. As shown in Figure 3.4

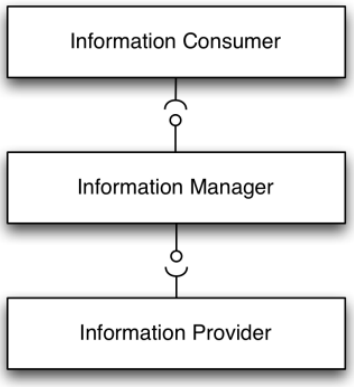


Figure 3.2: Proposed Monitoring information model [4].

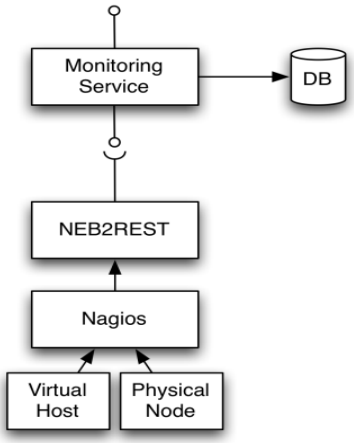


Figure 3.3: Proposed Monitoring Architecture [4].

Ganglia is based on a hierarchical design targeted at federations of clusters. It uses a multi-cast-based listen/publish protocol. Within each cluster, Ganglia uses heart beats messages on a well known multi-cast address as the basis of a membership protocol. Membership is maintained by using the reception of a heartbeat as a sign that a node is available. Each node monitors its local resources and sends multi-cast packets containing monitoring data on a well known multi-cast address. All nodes listen for monitoring packets on the agreed multi-cast address to collect and maintain monitoring data for all other nodes [5]. Aggregation of monitoring data is done by polling nodes at periodic intervals. Ganglia Monitoring is implemented as a collection of threads, each assigned a specific task: 1) collect and publish thread (gmond): collects local node information and publishes it on a well known multi-cast channel, 2) listening threads: listen on the multi-cast channel for monitoring data from other nodes and updates monitoring data storage, and 3) XML export threads: accept and process consumers requests for monitoring data. Ganglia also provides a meta daemon *gmetad* to merge data from separate *gmond* threads and exposes them to

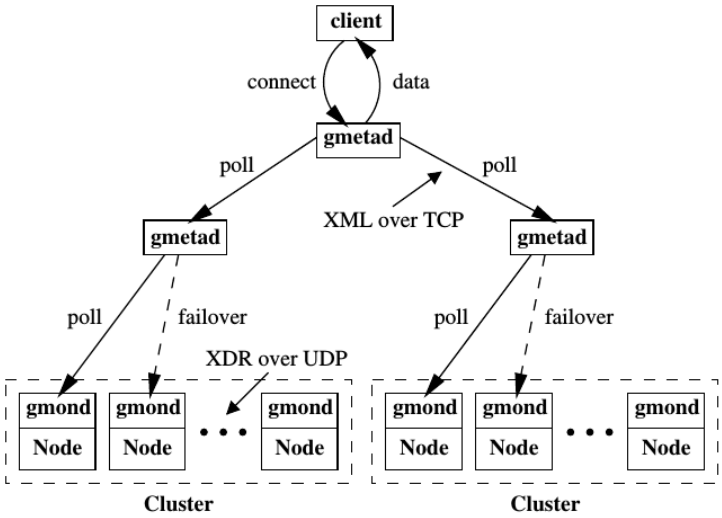


Figure 3.4: Architecture of Ganglia [5].

a web front-end accessible to consumers. Ganglia Monitoring system assumes the presence of a native multi-cast capability, an assumption that does not hold for the Internet.

In [38], authors proposed to use Mercury Grid Monitoring System along with GRM [6] monitoring system. GRM is used for monitoring parallel applications on a local resource. It as a monitoring system that consists of one Local Monitor per host (producer), a Main Monitor, and a Client Library. The structure of GRM is shown in Figure 3.5. Based on the Client Library, the user start by instrumenting

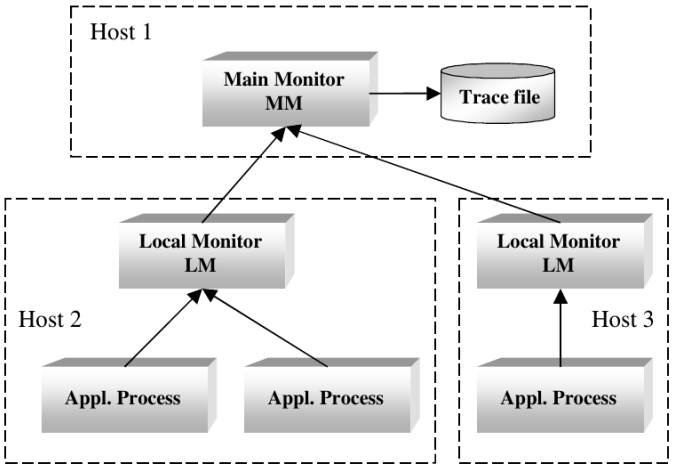


Figure 3.5: Structure of GRM [6].

his application with trace generation functions that provides user defined events. Consequently, Local monitors employ a set of sensors on each host to collect information about the local node, and send it to the main monitor. The latter aggregates Local Monitors data and performs synchronization among the different hosts. Mercury Grid Monitoring System offers a Monitoring Service which is also a client for the Main Monitor. The GRM Client Library is rewritten to publish monitoring data using an API proposed by Mercury.

H. Huang et al. [39] proposed an approach to monitor resources in the cloud using a hybrid model combining push and pull models. In these models, there are three basic components, the Producer, the Consumer and the Directory services. In the Push model, whenever the producer detects a change in a resource's status, it sends information to the consumer. Otherwise, in the Pull model, it's the consumer who asks the producer periodically about the resource's status. It is obvious that these two models have advantages and weakness. The authors propose a hybrid model that can switch to the best suited model according to the user requirements. The user can define his tolerance to the status inaccuracy between the producer and the consumer. Using this value, an algorithm can switch between pull and push models. In this approach the producer is in charge of subscriptions and sending notifications for all interested subscribers.

In GridRM [7], every organization has a Java-based gateway that collects and normalizes events from local monitoring systems. Accordingly, every gateway operates as a republisher of external producers. A global registry is used to support consumers in discovering gateways providing information of interest. Each gateway consists of a global and a local layer as shown in Figure 3.6. The former includes an abstract layer which interfaces with platform-specific consumer APIs (Java, Web/Grid Services, etc.) and a security layer that applies an organization's access control policy. The local layer, has several components including an abstract data layer and a request handler. The latter receives consumer queries from the global layer and collects real-time or archived data from appropriate sources depending on the query's type (last state or historical). The abstract data layer includes several JDBC-based drivers,

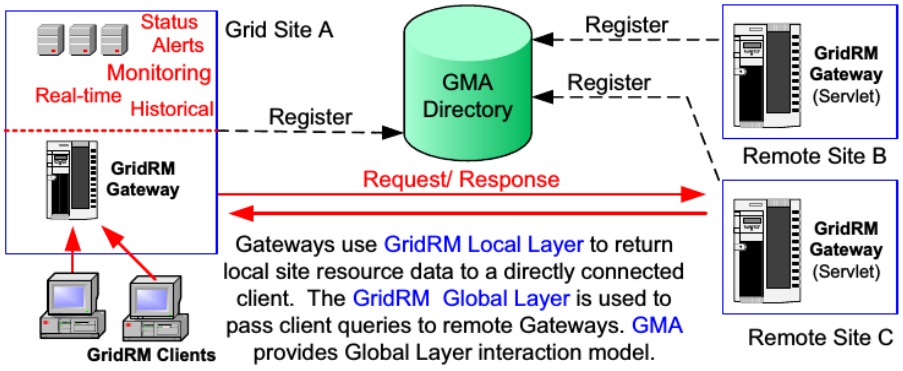


Figure 3.6: Architecture of GridRM [7].

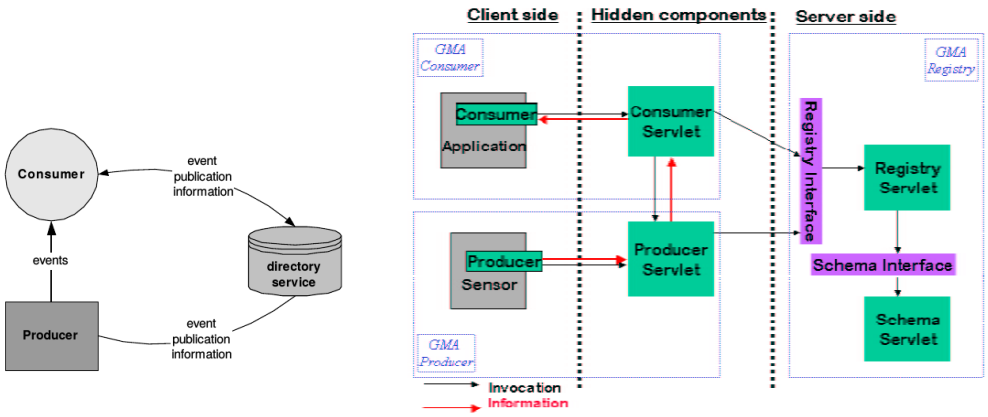
each one for retrieving data from a specific producer. GridRM hides the diversity of monitoring sources behind JDBC's widely used interface. Gateways represent events according to the GLUE schema [40]. Consumers form and submit SQL queries using GLUE as the vocabulary, and gateways forward the

queries to the appropriate drivers. Gateways are likely to become a bottleneck, whereas they also pose a single point of failure.

In [8], authors presented Globus Monitoring and Discovery Service (MDS). This service is based on two core protocols: the Grid Information Protocol (GRIP) and the Grid Registration Protocol (GRRP). The former allows query/response interactions and search operations. GRIP is complemented by GRRP, which is for maintaining soft-state registrations between MDS components. The LDAP is adopted as a data model and representation (i.e., hierarchical and LDIF respectively—LDAP Directory Interchange Format), a query language and a transport protocol for GRIP, and as a transport protocol for GRRP. Given the LDAP-based hierarchical data model, entities are represented as one or more LDAP objects defined as typed attribute-value pairs and organized in a hierarchical structure, called the Directory Information Tree (DIT). Globus was implemented as a web services-based framework aiming to enhance interoperability among heterogeneous systems through service orientation (i.e., hiding the underlying details by means of consistent interfaces). Afterwards, Globus was redesigned and implemented as part of the Open Grid services Architecture OGSA [41]. In OGSA, everything is represented as a grid service, that is, a web service that complies to some conventions, including the implementation of a set of grid service interfaces (portTypes in WSDL terminology). Every grid service exposes its state and attributes through the implementation of the GridService portType and, optionally, the Notification-Source portType, to support pull and push data delivery models, respectively. In this respect, the functionality of producers is encapsulated within grid services. In OGSA, the equivalent of the republishers is the Index Service which, among others, provides a framework for aggregation and indexing of subscribed grid services and lower level Index Services. Index Services are organized in a hierarchical fashion. Information is represented in XML according to the GLUE schema. Simple queries can be formed by specifying a grid service and one or more service data elements, whereas more complex expressions are supported using XPath. Consumers can submit queries to producers or republishers, or discover producers through republishers using GRIP.

The CODE (Control and Observation in Distributed Environments) framework [42] includes observers, actors, managers and a directory service. Each observer process manages a set of sensors and provides their events through an event producer interface, hence acting as a producer. Every actor process can be asked, through an actor interface, to perform specific actions, such as restarting a daemon or sending an email. A manager process consumes the events producers and, based on a management logic instructs actors to perform specific actions. Producers and actors register their locations, monitoring metrics in a LDAP-based registry where managers look up for the appropriate producers and actors. A management agent, consisting of an observer, an actor and a manager, is placed in each server of a Globus[8] installation. Events generated by management agents are forwarded to an event archive which is discovered through the registry. A GUI management front-end (i.e., a consumer) retrieves events from the archive (a republisher) to represent the current status of hosts and networks. The GGF's XML producer-consumer protocol is employed for exchanging events, while the event archive is an XML database queried using XPath.

In [43], authors presented R-GMA as a framework that combines grid monitoring and information services based on the relational model. That is, RGMA defines the GMA components, and hence their interfaces, in relational terms. R-GMA is based on Relational Database Management System and Java Servlets as shown in Figure 3.7. This system uses the notification of events where interested parts can



(a) GMA components (b) R-GMA components

Figure 3.7: GMA and R-GMA components [8].

subscribe with specific properties and filters. GMA is an architecture for monitoring components of grid platforms. As shown in Figure 3.7(a), it consists of Consumers, Producers and Registry. Producers register to the Registry while Consumers query the Registry to discover the existing Producers. As shown in Figure 3.7(b), R-GMA follows the same architecture as GMA but using SQL queries. It is noteworthy that R-GMA implementation is considered stable but suffers in terms of performance.

MonALISA [44] is a Jini-based extensible monitoring framework for hosts and networks in large-scale distributed systems. It can interface with locally available monitoring and batch queuing systems through the use of appropriate modules. The collected information is locally stored and made available to higher level services, including a GUI front-end for visualizing the collected monitoring events. MonALISA includes one main server per site or cluster within a grid, and a number of Jini look-up discovery services. The latter can join and leave dynamically, while information can be replicated among discovery services of common groups. A main server hosts, schedules, and restarts if necessary, a set of agent-based services. Each service registers to one or more discovery services to be found by other services. The registration in Jini is lease-based, meaning that it has to be periodically renewed, and includes contact information, event types of interest and the code required to interact with a given service. Each station server hosts a multi-threaded monitoring service, which collects data from locally available monitoring sources (e.g., SNMP, Ganglia, LSF, PBS, Hawkeye) using readily available modules. The collected data are locally stored and indexed in either an embedded or an external database, and provided on demand to clients (i.e., consumer services). A client, after discovering a service through the look-up service, downloading its code and instantiating a proxy, can submit real-time and historical queries or subscribe for events of a given type. Services and modules can be managed through an administration GUI, allowing an authenticated user to remotely configure what needs to be monitored. MonALISA uses Jini that is based on multicast that is not adequate to Cloud environments.

Java Management Extension (JMX) [45] is a standard Java extension that allows managing complex software systems. It defines three basic layers: instrumentation layer, agent layer and distributed services

layer. The instrumentation layer exposes the application as one or more managed beans (Mbeans). A Mbean is a Java class that implements one or more interfaces specified by JMX specifications. These interfaces are used later to get information about the Mbean. The agent layer provides monitoring and remote management access to all the registered Mbeans. Finally, the distributed services layer provides the interfaces and components used to interface with agent layer. For each class to be monitored, the developer needs to define the associate Mbean and to define the monitored properties. Afterwards, the agent layer can invoke the actions proposed by the Mbean to retrieve monitoring data. This data could be consumed by polling or by subscription through the tools provided by the distributed services layer.

Almost all of these monitoring approaches do not offer a granular description of monitoring requirements. In these solutions, the monitoring is added using a programmatic manner by adding annotations to the application or by using instrumentation mechanisms to detect monitoring information. That needs that the developers has the sufficient knowledge of the different properties or services to be monitored as well as the different functionalities provided by the monitoring solution. However, we did not find any work that adds automatically monitoring facilities to resources just from a given description. Moreover, in almost all of the stated works, the monitoring systems do not address scalability issues or do not offer an efficient solution in this direction. In contrast, in our approach, we will provide models to describe the monitoring requirements with a fine granularity. We will also provide needed mechanisms to render components and Cloud resources monitorable even if they were not designed with monitoring facilities. Finally, in our proposal, we will give solutions that are in line with the scalability of Cloud environments.

3.3 Reconfiguration Related Work

In the literature, there are different proposed approaches for dynamic reconfiguration. In this section we will present a list of these approaches. The list is far away of being exhaustive because of the large number of existing works addressing dynamic reconfiguration.

Authors of [46] presented an extension of CORBA programming model to enable component reconfiguration with minimal service degradation. The considered actions for reconfiguration are objects addition and deletion, object migration and binding modification. Authors defined two states of objects: execution state where the objects can accept requests and reconfiguration state where the state of the object is saved and the object can not accept any request. The state saving of an object is made by saving its internal state including its data structure and context. To facilitate this task, it is necessary to verify that there are no threads in execution. Since CORBA Identifier Object Reference (IOR) can change during the migration of objects, authors proposed an extension to the CORBA naming service to support the use of symbolic names for objects to identify them. Accordingly, an application is a collection of components collaborating to a global objective. Adding or deleting components is performed by adding or deleting existing binding between the components. Migration of a component could be performed by suspending its execution, saving its state, recreating it in the target location and restoring its execution. Finally, component replacement can be seen as an addition followed by a removal of component.

In [22], authors proposed a solution to address dynamic reconfiguration for component-based applications. A first type of reconfiguration is the replacement of a component by another one having compatible interfaces. The second type is the replacement of a subtree of the application components

hierarchy. Dynamic reconfiguration can be initiated by the application itself as well as from the outside of the application. Based on SOFA [47] (SOFtware Appliances), authors presented the Nested Factory Pattern as a solution to cover adding components and connections to the architecture. This pattern implies that the newly created component is a result of a method invocation on factory component. The newly created component is directly linked to the component that invoked the factory. To allow Adding/ Removing interfaces, authors introduced "utility interface" as an interface that could be used by all the components as a an external service.

A new approach for Platform-independent dynamic reconfiguration was proposed in [9]. In this work, reconfiguration aspects are described in an abstract manner. These aspects are satisfied by platform-specific realization of the application.

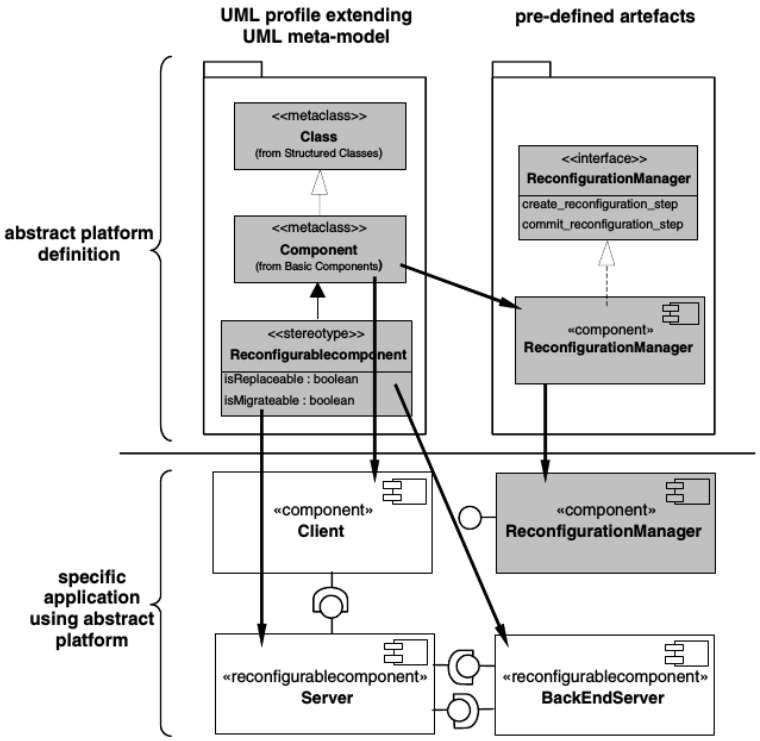


Figure 3.8: Support for dynamic reconfiguration [9].

To this end, they resume reconfiguration in a list of entities and operations that could be applied on these entities. Reconfiguration operations on entities are typically replacement, migration, creation and removal. In their approach, authors distinguish between reconfigurable and non reconfigurable components as shown in Figure 3.8. Reconfigurable components can be replaceable and/or migrateable. An application reconfiguration is an incremental evolution from an initial configuration (initial state) to a resulting configuration (objective state) through reconfiguration steps. A reconfiguration step can be composed of a set of reconfiguration steps. Reconfiguration steps are handled by a Reconfiguration

Manager.

Y. Li et al., [48] propose an adaptation approach to reconfigure SOA-based application to comply with a new quality of service (QoS) constraint by replacing its individual or multiple component services. They distinguish two major causes that can trigger the dynamic reconfiguration. Firstly, if one component service violation happens, thus the service needs to be replaced by an appropriate candidate in order to ensure an application working by complying with the initial QoS constraint. Secondly, if a user expects to improve the QoS of the application, there is a need to replace components in order to satisfy a given QoS constraint. In their work, authors focus on the second cause. For this, they propose to adapt the applications in two phases: 1) try to replace each individual component service or 2) replace multiple components services. Authors introduced a new factor named the significance factor, which reflects the influence of a component on the QoS of the application. For the both phases, the adaptation is achieved according to the global significance value. The global significance value is obtained by a weighted quantitative calculation of QoS attributes of each component. In fact, four QoS attributes of Web services are considered as follows: Response Time, Cost, Reliability, Availability. The importance of the global significance value of a component is proportional with its contribution on the improvement of the QoS of the application. Thus this component should be replaced firstly. Therefore, they go through the candidate services having similar functions to find a substituter. If all candidates cannot comply with the QoS constraint, they try to replace another service whose global significance value is the highest among the rest of component services and so forth. If all attempts failed in the first phase, the number of the replaced services will be increased gradually to replace multiple component services until the application meets the new QoS constraint required by users. Hence, using this approach allows to replace components gradually given their global significance values, whenever the user preferences change.

In their work [49], authors present an approach for repairing failed services by replacing them with new ones and ensuring that they meet the user specified end-to-end QoS constraints. The reconfiguration is triggered by services failures to deliver the QoS as requested by users because of, for example, network delay, host overload, unexpected inputs, etc. Their proposal consists of a reconfiguration algorithm designed to produce reconfiguration regions. A reconfiguration region is composed of one or more failed services. When one or more services in a service composition fail at runtime, they try to replace only those failed services. In that way, the resulting service process must still comply with the original end-to-end QoS constraints. The algorithm first identifies the regions that are affected by faulty services. The motivation from the reconfiguration region is to reduce the number of services that need to be replaced by the service recomposition algorithm. In this algorithm, the region identification procedure increases the size of affected region gradually. The range bound starts from zero, which means the algorithm will try to replace only the failed services first. For a failed service, an output reconfiguration region extends from it to all nodes that are connected to it. If it fails to replace the services successfully, the range bound will be increased, therefore, more services will be added to the region to be reconfigured. The algorithm will then try to compose the replaced components and regions. After the initial composition attempt, if there are still some regions that cannot be reconfigured, the algorithm will increase the range of the affected regions, which means more nearby components will be added into the region. The algorithm continues until all regions have a satisfactory composition.

MADAM (Mobility and ADaption enAbling Middleware) [50] proposes to reconfigure applications

by using plans. A plan describes an alternative application configuration. It mainly consists of a structure that reflects the type of the component implementation and the QoS properties associated to the services it provides. Plans are ranked by evaluating their utility with regards to the application objectives and the user preferences in order to select a plan with a highest utility. However, these configuration plans are predefined by a developer at design time, which limits the possible adaptations to these predefined plans. Moreover, the utility function is limited to the application and hence can not be used for other applications.

The PCOM system [51] also uses reconfiguration plan algorithms to support a PCOM component reselection whenever this later is no longer available. With respect to the application model defined by PCOM, which is presented as a tree, the reconfiguration consists of trying to replace the parent component. This replacement attempt continues until the problem is resolved by reselecting components. Thus, the replacement of a sub-tree starts from the predecessor of the component and may include its successors if it is necessary even if they are not concerned by the changes.

In [10], authors presented their design and implementation of a framework for dynamic component reconfiguration in a *.Net* environment. The used framework allows the description of application configurations and profiles. Based on measured environmental metrics, the framework loads the corresponding configuration. To handle reconfiguration aspects separately from the functional code, they use aspect oriented programming. To describe application configurations and profiles, they use an XML-based language that allows them to specify an observer element that provides monitoring data. Then, they describe the different profiles using Profile elements. Each profile refers to one configuration. Based on monitoring data, the framework loads the associated profile and apply the associated configuration. The framework has a central Configuration Manager, which is suspicious of forming a bottleneck in high scale environments such as Clouds.

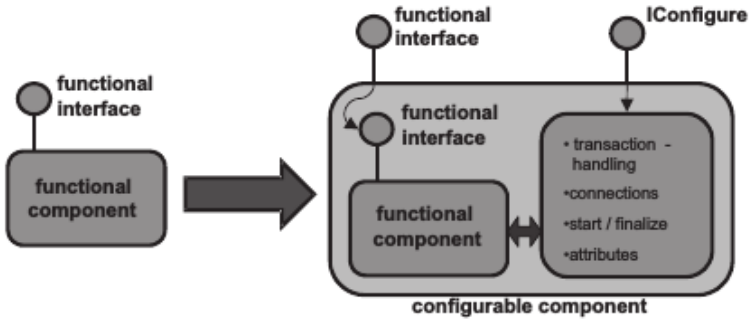


Figure 3.9: Adding reconfiguration to a component in .Net platforms [10].

For the configuration manager, to be able of reconfiguring components, each component must implement an interface that allows the management of the component as shown in Figure 3.9. The basic requirements for the reconfiguration algorithm used within the framework are minimizing the blackout period (the timespan an application does not respond to user requests) and the overall duration of the reconfiguration process. Dynamic reconfiguration in this approach deals with three basic operations: the addition of a component, the removal of a component, and the modification of a component's attribute

value. Complex reconfiguration operations can be broken up into series of those basic operations. To reconfigure an application, this latter must keep its consistency. This is possible if there are no pending requests between components. To get the application to a reconfigurable state, the framework blocks the connections between components. Using LOOM.NET [52], the framework enables adding configurability to a component with no interaction with the programmer. All what it needs, is to recognize the beginning and ending of a transaction.

Almost all of the proposed solutions, are specific to a given layer. Therefore, they can not be generalized to all layers in the Cloud. In our study of the state of the art, we noticed that the majority of the existing work proposed configurations based on adding/removing components or binding modification. The enabled reconfiguration are described at design time and could not be adapted later. Moreover, the existing works do not tackle neither scalability issues nor flexibility. In our case, since our proposals are based on SCA or OCCI, we can apply the first proposal to all components while we can apply the second proposal to all Cloud resources. The extensibility of our work is guaranteed using the mechanisms proposed by SCA and OCCI. Since our descriptions are generic, we can apply them on different systems without any modification. Furthermore, to enable different kinds of reconfigurations, all what we need is to provide new extensions to add the new specific behavior (i.e., SCA intent or OCCI Mixin to be detailed later). In our solutions, we will use mechanisms providing efficient solutions to scalability and flexibility issues.

3.4 Autonomic Computing Related Work

In Cloud and distributed environments, there are different research works around autonomic computing. In the following we will give an overview on these works.

One of the pioneer in the Autonomic Computing field is IBM who proposed an entire toolkit. It is called the IBM Autonomic Computing Toolkit [11]. Figure 3.10 shows IBM vision of autonomic elements and their interactions. Authors gave the IBM definition of Autonomic Computing as well as the needed steps to add autonomic capabilities to components. The proposed toolkit is a collection of technologies and tools that allows users to develop autonomic behavior for their systems. One of the basic tools is the Autonomic Management Engine that includes representations of the autonomic loop. Several tools are proposed to allow managed elements create log messages in a standard format comprehensive by the Autonomic Manager. This is realized by creating a touch-point containing a sensor and an effector. The sensor role is to sense the state of the managed elements and generate logs. While the effector is used to execute adaptations on the resource. Moreover, IBM proposed an Adapter Rule Builder that allows to create specific rules to generate adaptations. The proposed scenarios presented in [11] show the efficiency of this solution. In our point of view, the efficiency of the proposed tools could be adapted to Cloud environments by the usage of the *de facto* standards. Since we are not proposing this large number of tools in our proposal, we can adapt some of them to our needs in order to enhance our solution.

In [12], authors proposed a process to develop autonomic management systems for Cloud Computing. This process consists of a list of steps to respect in order to get an autonomic system. The first step is to identify the control parameters. These parameters are the different elements that need to be monitored

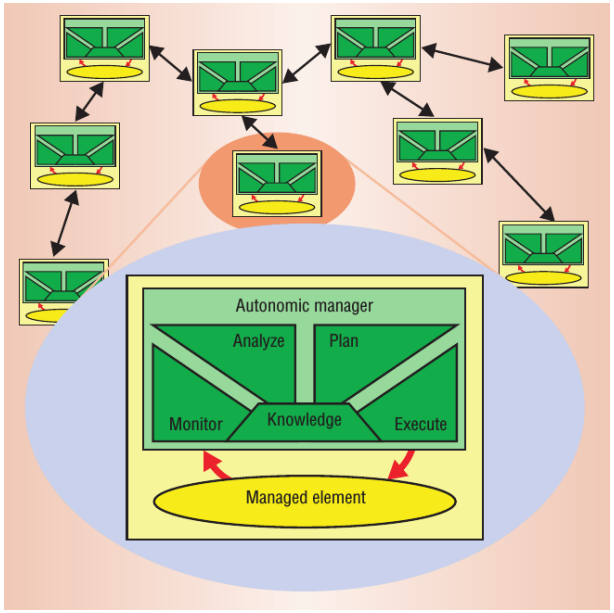


Figure 3.10: Autonomic Elements and their interactions [11].

or reconfigured in the system. The second step is to define a system model that provides a view of the managed system. The next step is to identify the inputs of the elements that need to be monitored. Afterward, there is the initialization of the parameters of the system model previously defined. Then, they need to decide the update rate of the model according to the measurements. A user needs also to specify the type of the decisional system. If the system is proactive, there is a need to create a prediction system. The user can also create a coordinator that represents a central link between the different components. Finally, sensors and filters could be deployed to take measurements. At this step the autonomic system is in place to control the managed system and apply reconfiguration actions on it. The autonomic system needs also to be capable of monitoring and reconfiguring itself when needed.

The realized autonomic system is resumed in Figure 3.11. In this loop, authors propose to gather monitoring data using sensors. The gathered data is aggregated and filtered and sent to a Coordinator. This latter models the incoming information and process it using an Estimator prior sending it to a Decision Maker. This Decision Maker decides the adaptations to be applied on the managed element or on the system itself.

In their work [13] R. Buyya et al. proposed a conceptual architecture to enhance autonomic computing for Cloud environments. The proposed architecture is basically composed of a SaaS web application used to negotiate the SLA between the provider and its customers, an Autonomic Management System (AMS) located in the PaaS layer and a IaaS layer that provides public or private Cloud resources. As shown in Figure 3.12, the AMS incorporates an Application scheduler responsible of assigning applications to Cloud resources. It incorporates also an Energy-efficient scheduler that aims to minimize the energy consumption of all the system. The AMS implements the logic for provisioning and managing virtual resources. The PaaS layer contains also a component to detect Security infraction and attack

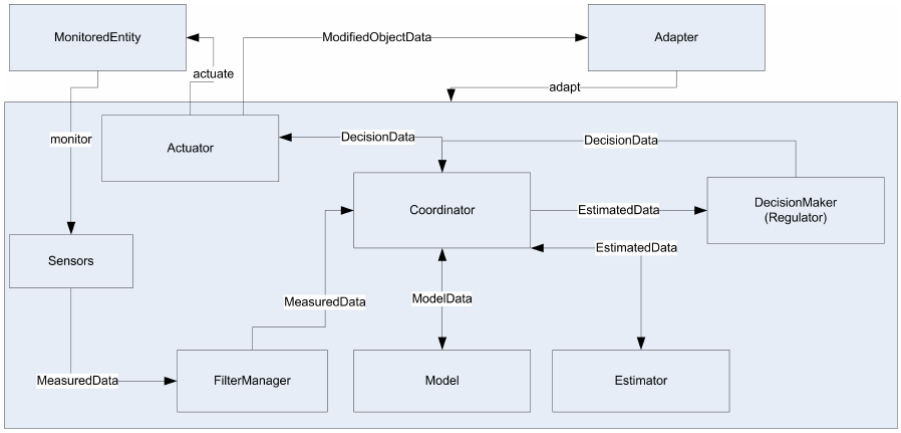


Figure 3.11: Autonomic Control Loop [12].

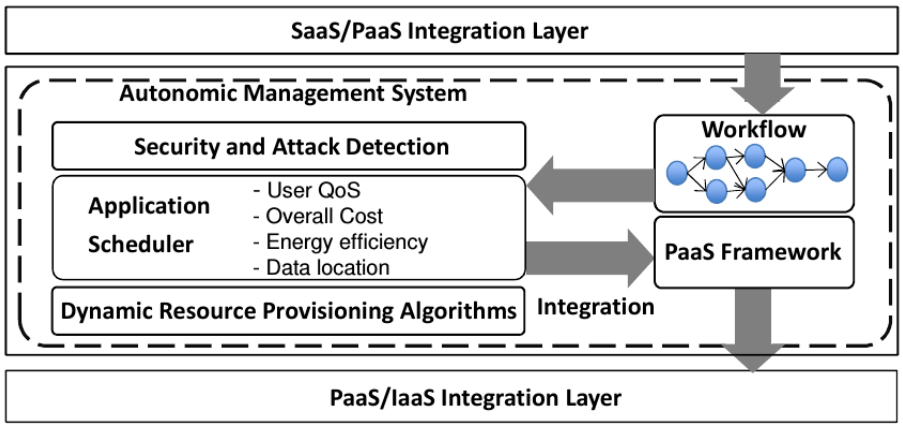


Figure 3.12: System architecture for autonomic Cloud management [13].

tentatives.

Autopilot [53] is a framework for enabling applications to dynamically adapt to changing environments. A real time monitor compares application's progress against the requirements of its contract and triggers corrective actions in case of violations. Application adaptability requires real time measurements, reasoning and instructing the application to perform corrective actions when needed. Autopilot's functionalities are implemented in separate components, namely sensors, actuators, clients and distributed name servers. Sensors and actuators are for remote reading and writing application variables. They have to register themselves to a name server (registry). Clients look up in the registry to find interesting sensors and subscribes to receive their events. The client uses an application specific logic to make decisions and instruct an actuator to perform adaptive instructions.

Rainbow [14] is a framework that uses an architecture-based approach to add self-adaptation mechanisms. It is based on an abstract model that represents the architecture of the application as a graph.

The graph is composed of nodes that represent components and arcs that represent connectors and interactions between components. The model is continuously adapted using a model manager. The model manager gathers monitoring data from monitoring probes. The collected data is analyzed by means of a constraint evaluator able of detecting violations and triggering adaptations. Rainbow uses an adaptation engine to choose the appropriate adaptation and apply it using executors. As shown in Figure 3.13, the

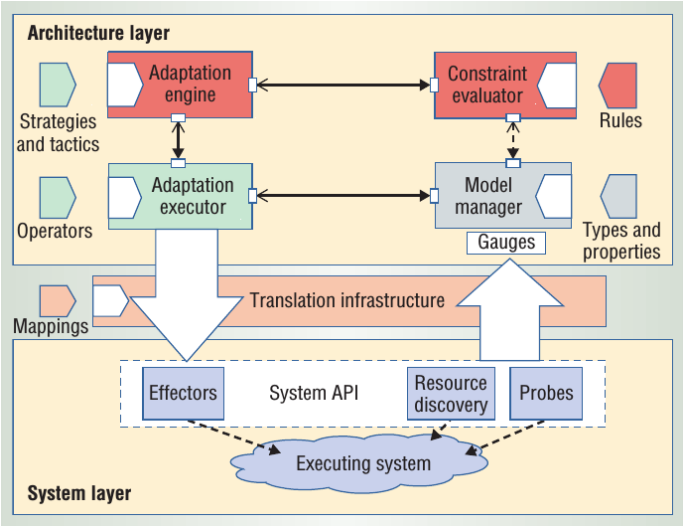


Figure 3.13: Rainbow framework abstract model [14].

framework is divided basically into an architecture layer and a system layer. On the one hand, the system layer describes the managed system and the ways to manage it by means of monitoring probes, effectors, management APIs, etc. On the other hand, the architecture layer contains the needed components to analyze monitoring data and to trigger adaptations when it is necessary. Between these two basic layers, there is a translation infrastructure used as a mediator for the exchanged information.

StarMX [15] is a generic framework that allows building autonomic Java-based applications. It is an open source project that uses JMX (Java Management eXtensions) to create autonomic loops. This framework implements adaptive behaviors using policy languages or Java code following condition/action model.

As shown in Figure 3.14, the framework implements management logic as a set of entities called processes. The composition of processes forms the autonomic manager and the autonomic loop. Each process implements one or more parts of the MAPE loop. At runtime, these processes are grouped in execution chains. These chains are invoked to ensure their specific objectives with given parameters and objects. The system uses anchors to get monitoring data and to execute actions. In their implementation, anchors are mostly Java Mbeans. The communication between execution chains and anchors is performed through services that allow initiating the activation mechanisms, querying anchors, data sharing, etc.

CAPPUCINO [16] is a runtime supporting SOA execution in ubiquitous environments. It is based on SCA as the basic model to build applications and their autonomic loops. Figure 3.15 describes CAP-

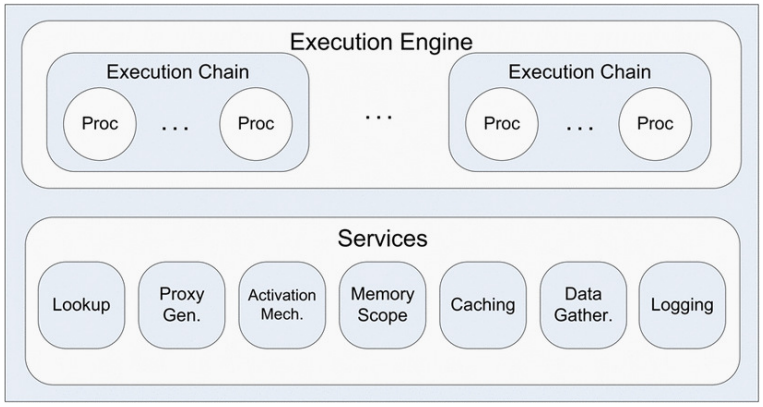


Figure 3.14: StarMX high level static architecture [15].

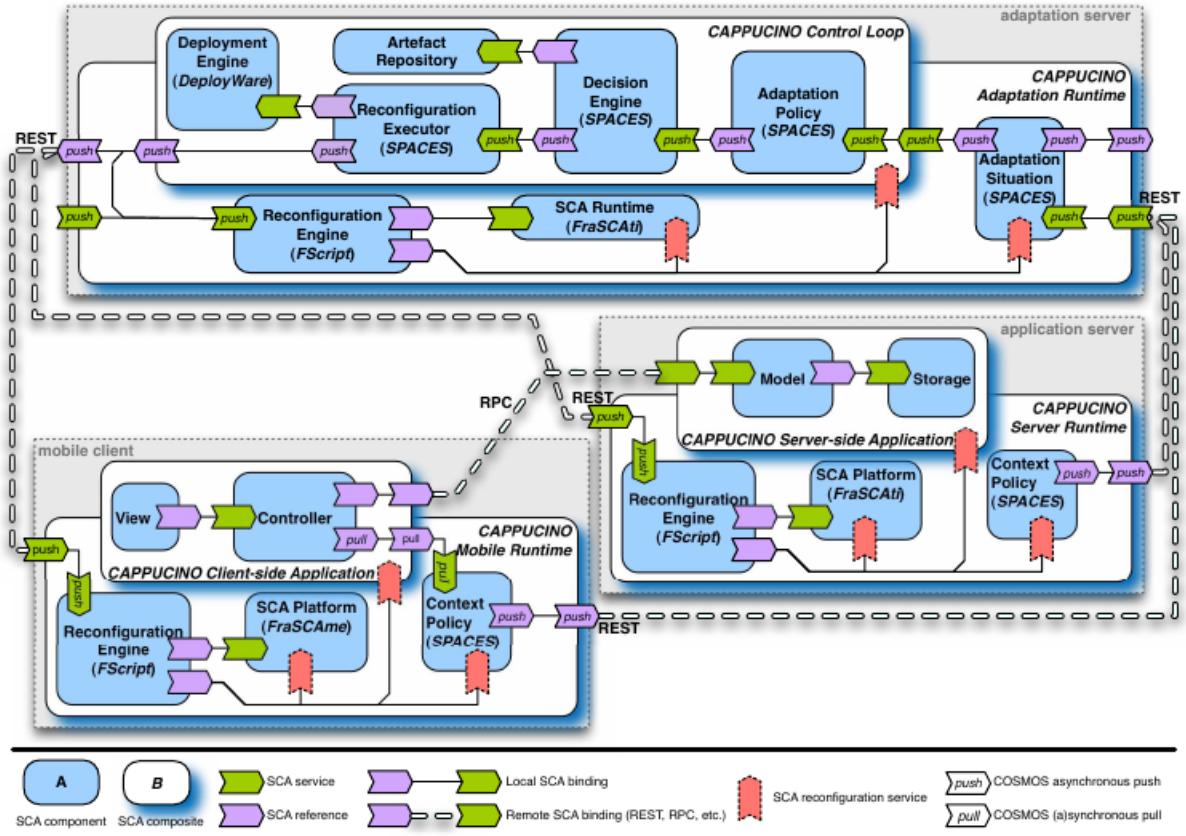


Figure 3.15: Overview of the CAPPUCINO distributed runtime [16].

PUCINO distributed runtime and its basic components. The Figure shows the usage of an adaptation server, a mobile client and an application server. The adaptation server contains the control loop related to an application. In order to deploy the proposed runtime, authors used FraSCAti [20]. They also used COSMOS [54] framework for context information management. The connection between CAPPUCINO and the managed applications is based on RESTful bindings.

C. Ruz et al. [17] introduced a framework that address monitoring, SLA management and adaptation strategies for component-based applications. In their approach, authors separate Monitoring, Analysis, Decision and Execution concerns by implementing each of them as separate components that could be attached to a component to manage it (see Figure 3.16).

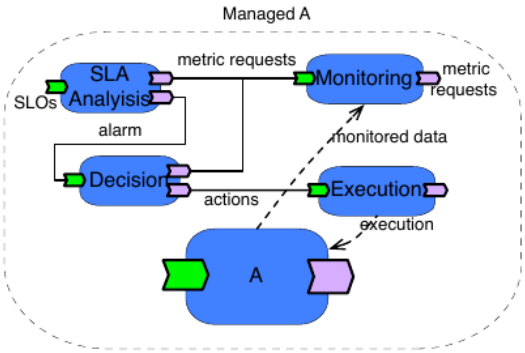


Figure 3.16: Adding reconfiguration to an SCA component [17].

These components can be added or removed when needed. Each component can communicate with other components (i.e., monitoring component can use monitoring services from other monitoring components). As shown in Figure 3.16, Monitoring component is responsible of collecting metrics from the managed component using JMX (Java Management Extension [45]). The SLA Analysis component is a consumer for monitoring data. It compares these data against previously defined SLA. The Decision component implements one or more strategies to react to SLA infraction. It executes the associated strategy to a notification and generates a list of actions to be applied on the managed component to meet the SLA. The Execution component applies the list of actions provided by the Decision component.

In the state of the art, there are other works related to autonomic computing applying this paradigm on different areas [55, 56, 57, 58, 59, 60, 61, 62]. Almost all of these works propose specific solutions. Specially when coupled with Cloud environments, the proposed works are really facing a specific area and could not be generalized. Moreover, in all the studied works, the autonomic management behaviors are added in a programmatic way by implementing specific interfaces or by placing instrumentation codes. This addition is made at design time and can not be modified without restarting the system. We did not find any work that adds autonomic management facilities to resources automatically based on a given description. However, in this manuscript we propose two generic solutions. The first one deals with components in the Cloud, while the second is more generic and could be applied to Cloud resource on all the levels. We propose to dynamically add autonomic management facilities to components and Cloud resources even if they were designed without them based on a given descriptor. The extensions

entails the description of needed autonomic elements as well as different mechanisms allowing to ensure our objectives specified in Chapter 1.

3.5 Synthesis of Related Work

In this chapter, we proposed an overview on existing works related to monitoring, reconfiguration and autonomic management in the Cloud and in distributed systems. In the following, we will show, how the presented works failed to cover all the objectives that we dressed in Chapter 1. In Table 3.1, we propose a synthesis of the studied works based on criteria derived directly from our requirements and objectives. We show for each work, whether it responds or not to our objectives. In this table, an empty cell means that the related work does not propose a solution for the given criteria. While, the ✓ is used to say that the proposed work treats the criteria.

Studied Solutions	Level	Genericity	Granularity	Standardization	Scalability	Extensibility	Flexibility
Monitoring							
Nagios [3]	IaaS					✓	
Katsaros et al. [4]	IaaS			REST		✓	
Ganglia [5]	IaaS				✓		✓
GRM [38]	IaaS						✓
CODE [42]	XaaS		✓		✓		
H. Huang et al. [39]	XaaS	✓	✓				
GridRM [7]	IaaS		✓				
Globus MDS [8]	IaaS		✓		✓	✓	✓
MonALISA [44]	XaaS	✓	✓			✓	✓
R-GMA [43]	XaaS	✓	✓				
Reconfiguration							
N.-C. Pellegrini [46]	SaaS	✓	✓	CORBA		✓	✓
P. Hnetynka et al. [22]	SaaS	✓	✓	SCA		✓	
J. Almeida et al. [9]	SaaS	✓	✓			✓	
Y. Li et al. [48]	SaaS	✓					
Y. Zhai et al. [49]	SaaS	✓	✓				
MADAM [50]	SaaS		✓				
PCOM [51]	SaaS	✓	✓		✓	✓	✓
A. Rasche et al. [10]	SaaS		✓				
Autonomic Management							
IBM Toolkit [11]	XaaS		✓		✓		✓
B. Solomon et al. [12]	XaaS		✓				✓
R. Buyya et al. [13]	XaaS		✓			✓	
Rainbow [14]	SaaS	✓	✓				✓
StarMX [15]	SaaS	✓	✓	JMX	✓		
Autopilot [53]	SaaS	✓	✓		✓		✓
CAPPUCINO [16]	SaaS	✓	✓	SCA		✓	
C. Ruz et al. [17]	SaaS	✓	✓	SCA, JMX	✓		

Table 3.1: Synthesis of Related Work.

In Table 3.1, we notice that there is no work that covers all our objectives. Meanwhile, there are some works close to offer a solution that responds to our requirements like Globus MDS [8], C. Ruz et al., [17] and CAPPUCINO [16]. Moreover, we did not find any work that adds monitoring, reconfiguration and autonomic management facilities to Cloud resources automatically based on a given description. The majority of these presented works propose to add management facilities in a programmatic manner (using annotations, adding instrumentation code to applications, etc.). Almost of them are limited to specific areas or specific layer in the Cloud. At the same time, we did not find any solution that gives an efficient solution to scalability issues. In our work, we tried to tackle the different drawbacks in the state of the art regarding our requirements. Our two proposals have to respond to all the dressed objectives. As we will detail later in this manuscript, our first proposal succeeds to realize almost all the objectives for components in the Cloud. However, our second proposal raises all the challenges to provide a generic solution that could be applied on Cloud resources independently of their levels.

3.6 Conclusion

Monitoring, reconfiguration and autonomic management exist way before Cloud Computing. So, there is a plethora of works dealing with them. In this chapter, we proposed an overview of the existing work in each area. It is worthy to say that there are some works that could be presented for one or more areas (e.g., C. Ruz et al work treats monitoring, reconfiguration and autonomic management). Table 3.1 presents an illustrative picture to show how each work responds to the objectives listed in Chapter 1. In our work, we will propose two solutions that respond to all the dressed objectives. Accordingly, in Chapter 4, we will propose a solution for monitoring, reconfiguration and autonomic management for SCA-based applications. This solution responds to all the objectives for components in the Cloud. In Chapter 5, we propose a more generic solution that responds to all our requirements and that spans over the different levels of the cloud.

Chapter 4

Monitoring and Reconfiguration of SCA-based applications in the Cloud

Contents

4.1 Introduction 37
4.2 Adding Monitoring and Reconfiguration facilities to component 38
4.2.1 Extended SCA Model 38
4.2.2 GenericProxy Service: 42
4.2.3 Reconfiguration 42
4.2.4 Monitoring by Polling 42
4.2.5 Monitoring by Subscription 43
4.2.6 Transformation Example: 45
4.3 Monitoring and Reconfiguration within Scalable Micro-containers 47
4.3.1 Deployment Framework and generated Micro-Container 47
4.3.2 Adding Monitoring and Reconfiguration to Micro-Containers 48
4.4 Adding FCAPS properties to components 49
4.4.1 Extension of Deployment Framework for FCAPS management 50
4.4.2 IMMC Anatomy 51
4.4.3 FCAPS Manager and IMMC basic functionalities 52
4.4.4 FCAPS Management 53
4.5 Conclusion 58

4.1 Introduction

Cloud environments are well suited to host service-based applications. As previously described (Chapter 2), a service-based application is a collection of services which communicate with each other. Such

type of applications can be described using Service Component Architecture (SCA) as a composite that contains a detailed description for different components of the application and links between them. All the elements in a composite must be described as one of the standard artifacts of the SCA model. In our work, we aim to dynamically add monitoring and reconfiguration non functional facilities to SCA-based applications. Doing so, we lighten the developers tasks from the burden of non functional facilities and let them focus on the business of their applications. We propose a list of transformations that dynamically adds management facilities to components even if they were designed without them. In this chapter, we will detail our proposal to add monitoring and reconfiguration to SCA-based applications. Then, in order to enforce our solution scalability, we will show how we use a light-weight container to deploy components in the Cloud. Further, we will present our extension to add Fault, Configuration, Accounting, Performance and Security (FCAPS) facilities to enhance the autonomic capacities of SCA-based applications. The proposal entails an extension of SCA model to add new elements supporting our solution.

4.2 Adding Monitoring and Reconfiguration facilities to component

In this Section, we will show how we add monitoring and reconfiguration facilities to components even if they where designed without them. Our idea consists in extending SCA model in order to allow a component expressing its requirements to monitor or reconfigure a property of another component. The extension entails all the needed elements to cover the different reconfiguration and monitoring types that interest us. In the following, we will describe this extension as well as its different usages.

4.2.1 Extended SCA Model

As defined by Szyperski [63] "A software component is a unit of decomposition with contractually specified interfaces and explicit context dependencies only". Thus, a component not only exposes its services but it also specifies its dependencies. Most of the existing component models [51] [64] [65] [31] allow specification of their dependencies for business services external to the component. However, they do not allow specification of their dependency for external properties. The ability to specify dependency for external properties that we refer to as Required Property has two important implications. First, it results in specification at relatively fine granularity thus helping the architects and designers in fine tuning the component's requirements. Second, this fine tuning helps in elaborating the contract between two components because the properties can be enriched with additional attributes that constrain the nature of the contract through appropriate policies.

Since the existing SCA model does not support the explicit description of Required Properties of a component, we decided to extend this model by adding some artifacts allowing the description of monitoring and reconfiguration facilities for component-based applications. These new artifacts allow a component to express its need to monitor or reconfigure properties of other components with a specific monitoring model (i.e. by polling or by subscription) and with needed characteristics related to monitoring and reconfiguration. We also added an artifact, to describe the need of a component to monitor services of another component. As shown in Figure 4.1, the basic added artifacts are *RequiredProperty* and *RequiredService*.

The *RequiredProperty* artifact is used to describe the need of a component to monitor or reconfigure one or more properties of another component. The *RequiredProperty* is further specified using the following added artifacts:

- Reconfiguration: used to say that the property is used for reconfiguration aims;
- Monitoring: used to say that the required property is to be monitored by polling or by subscription;

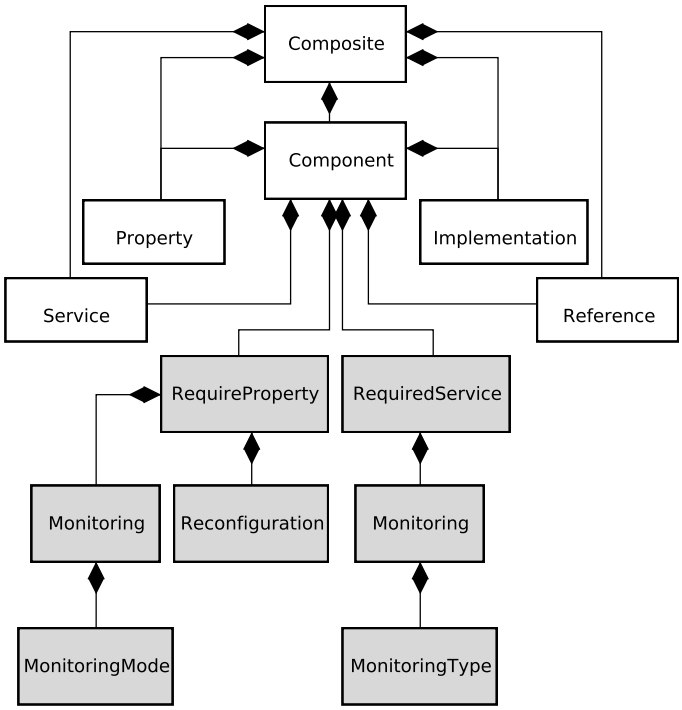


Figure 4.1: Extended SCA with monitoring and reconfiguration artifacts.

Monitoring by polling is used to say that the required property is to be monitored using polling model while monitoring by subscription is used to say that the required property is to be monitored using subscription model. In this case, we propose to use the *MonitoringMode* artifact to specify the mode of the subscription. Subscriptions could be on interval or on change. Monitoring by subscription on interval is used when a monitoring component needs to receive monitoring data periodically. In this case, the monitoring component may specify the start time and duration of its subscription, the interval of notifications as well as the callback interface to be used for the notifications. Monitoring by subscription on change is used when a component needs to be notified whenever there is a change on the value of the monitored property.

The *RequiredService* artifact is used to describe the need of a component to monitor one or more services of another component. For services, only monitoring by subscription on change is allowed.

However, the monitoring type could be further specified using the *MonitoringType* artifact. The monitoring types that we propose are Method Call Monitoring (MCM) and Time Execution Monitoring (TEM). MCM allows to receive notifications whenever the required service is invoked and TEM allows to receive monitoring information about the time needed for the required service to process the invocation.

The extended SCA model is shown in Figure 4.1. Some attributes related to monitoring may be declared for these artifacts like the start time of the subscription, duration of the subscription, and notification interval if the notification mode is on interval.

Figure 4.2 shows the main characteristics of a component. It provides a service through an interface and may require a service from other components through a reference. The component may expose properties through which it can be configured. Using our extension, the component can specify its dependency on certain property. This required property, which appears at the bottom of the component, will be satisfied if we can link this component with another component that offers the requested property, thus, solving the dependency.

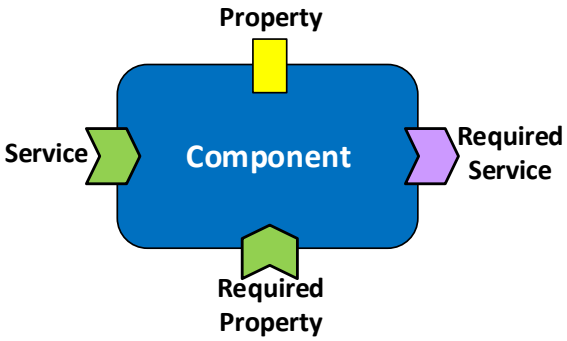


Figure 4.2: Extended Component with Required Property.

Components can be combined together in a composite as an assembly of components to build complex applications as shown in 4.3. A component (A) can specify its need to monitor properties of another component (B) and use the service offered by the component (C). The description of an application can be done using an Architecture Description Language (ADL). In this regard, SCA provides a rich ADL that details most of the aspects that we are looking for. Using the extended model of SCA, we can describe an assembly application using our extended SCA ADL as shown in Listing 4.1.

We take a generic example of a classic SCA-based application. The *ComponentA* is a component that requires to monitor the *propertyOfB* provided by *ComponentB*. This requirement is expressed using the *requiredProperty* element (Listing 4.1, Line 7). In this element, we can specify the monitoring type (i.e, by subscription or by polling), mode (i.e, on change or on interval) and schema (i.e, using one channel or multi-channel).

Listing 4.1: Description of a SCA-based Application using our extended SCA ADL

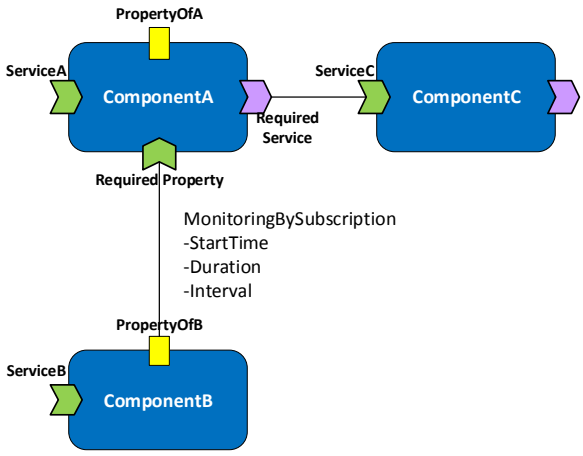


Figure 4.3: Component-based Application using required property.

```
1 <composite name="SampleApplicationomposite">
2  <service name="ServiceOfA" promote="ComponentA/ServiceOfA"/>
3  <component name="ComponentA">
4   <service name="ServiceOfA" >
5    <interface.java interface="example.ServiceOfAInterface"/>
6   </service>
7   <requiredProperty resource="ComponentB.propertyOfB" monitoring="BySubscription" monitoringMode="
        ON_CHANGE" multi_channel="true">
8    <property name="propertyOfB"/></requiredProperty>
9   ....
10 </component>
11 <component name="ComponentB">
12  <property name="propertyOfB">
13  <service name="serviceOfB" ><interface.javainterface="example.ServiceOfBInterface"/></service>
14  ....
15 </component>
16 </composite>
```

The extended SCA allows components to specify their needs to monitor other components' properties. However, these components can be designed without monitoring capabilities and cannot provide the status of their properties. To resolve this problem, we propose a list of transformations to apply to components to render them monitorable and reconfigurable. In the next subsections, we introduce the main features of monitoring and reconfiguration mechanisms and their transformation processes.

4.2.2 GenericProxy Service:

In order to add monitoring and reconfiguration facilities to components, we have defined a general purpose interface GenericProxy that provides four generic methods (see Figure 4.4). Each implementation of this interface is associated with a component for which the first method getProperties() returns the list of the properties of the component, the getPropertyValue() returns the value of a property, the setPropertyValue() changes the value of a property and the invoke() method invokes a given method on the associated component and returns the result.

```
public interface GenericProxy {
  Property[] getProperties();
  Object getPropertyValue(String propertyName);
  void setPropertyValue(String propertyName, Object propertyValue);
  Object invoke(String methodName, Object[] params);
 }
```

Figure 4.4: Definition of the GenericProxy interface.

The transformations that render a component monitorable and reconfigurable use a GenericProxy Component. The result of the transformation implements the GenericProxy Interface and the (proxy) services of that component.

4.2.3 Reconfiguration

A component may express its need to reconfigure a required property of another component. The reconfiguration of a property can be made by calling its setter method. However, the component that wishes to reconfigure a property of another component does not know a priori the type of this component. To complete the reconfiguration of any component from only the name and type of a property, the reconfigurator component uses an appropriate interface that provides the method setPropertyValue() to change the value of a property. However, the component to be reconfigured may not define its properties as reconfigurable resources despite the request. So we need to transform the component to make its properties reconfigurable by offering an appropriate reconfiguration interface. As shown in Figure 4.5, this can be done by encapsulating the component with a generated *GenericProxy* component as defined above. The two components are combined together in a single composite that offers the services of the original component as well as the *GenericProxy* interface. The component can be then reconfigured using the setPropertyValue() method provided by the *GenericProxy* component. Then, we replace the original component with the newly created composite in the application.

4.2.4 Monitoring by Polling

A component may express its need to monitor by polling a required property provided by another component. The monitoring by polling of a property can be made by calling its getter method. To complete the monitoring of any component from only the name and type of a property, the interested component

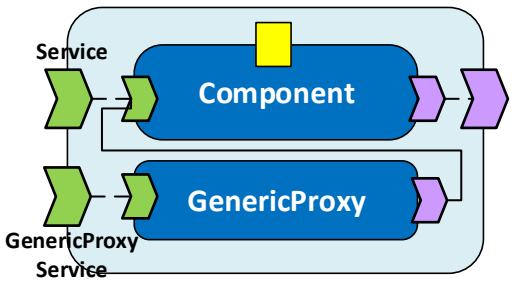


Figure 4.5: Reconfiguration and monitoring by polling.

often uses an appropriate interface that provides the method *getPropertyValue(propertyName)* to request the current state of a property. However, the component to monitor may not define its offered properties as monitorable by polling resources despite the request. Similarly to the reconfiguration, we need to transform the component to make its properties to be monitorable by offering an appropriate interface of monitoring. As shown in Figure 4.5, this can be done by encapsulating the component with a generated *GenericProxy* component as defined above. Consequently, the component can be monitored using the *getPropertyValue()* method provided by the composite. Then we use the newly created composite instead of the original component.

4.2.5 Monitoring by Subscription

A component may express its need to monitor by subscription a required property provided by another component. The monitoring by subscription of a property can be made by subscribing to a specific interface to receive monitoring notifications. However, the monitored component may not propose any interface to this aim. Consequently we propose to encapsulate the original component with a generated *NotificationByteCode* component and a predefined *Notification Service* component as shown in Figure 4.6. The *Notification Service* component is responsible of managing client subscriptions (Subscription service), receiving notifications (Notification service) from producer and sending them to interested subscribers.

If the monitoring mode is on interval, each time the *Notification Service* component have to notify the subscribers, it gets (or monitor by polling) the value of the required property of the monitored component via the *GenericProxyService* proposed by the *GenericProxy* component.

When the notification mode is on change for a required property of the monitored component, the *Notification Service* component offers a service of notification *Notification* to the *NotificationByteCode* component so that it can be notified of the changes of the required property and in turn inform all the subscribers of this change. The dynamically generated byte-code of the *NotificationByteCode* component allows to notify the *Notification Service* for the changes of the monitored component properties. It is worthy note that the same transformations are used for the Property Changed Monitoring, Method Call

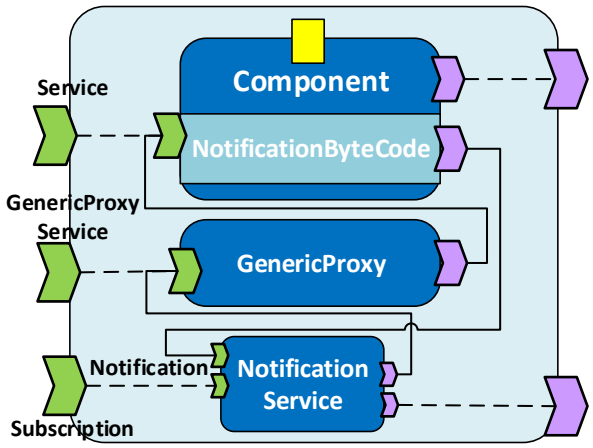


Figure 4.6: Monitoring by subscription multi Notification Service.

Monitoring and Time Execution Monitoring.

To be in line with Cloud environments scalability, it is really important to imagine the different schemas to enhance the scalability of the application. In order to add more flexibility to our approach, we propose to use one *Notification Service* monitoring (Figure 4.7) or multi *Notification Service* monitoring (Figure 4.6). One *Notification Service* monitoring implies that this *Notification Service* manages all producers notifications. In contrast, the multi *Notification Service* monitoring implies that every producer has its own *Notification Service* and will so manage just its monitored component.

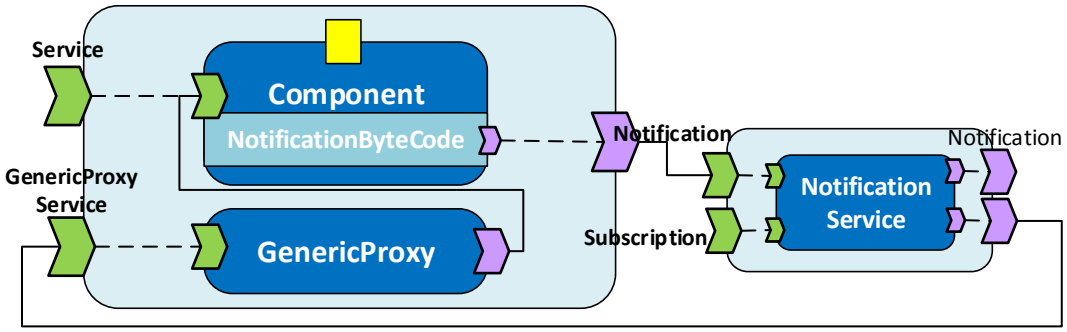


Figure 4.7: Monitoring by subscription one Notification Service.

We can also imagine a hybrid schema (as shown in Figure 4.8) in which we deploy a *Notification Service* component when needed. In each monitored component, we place a notification proxy component

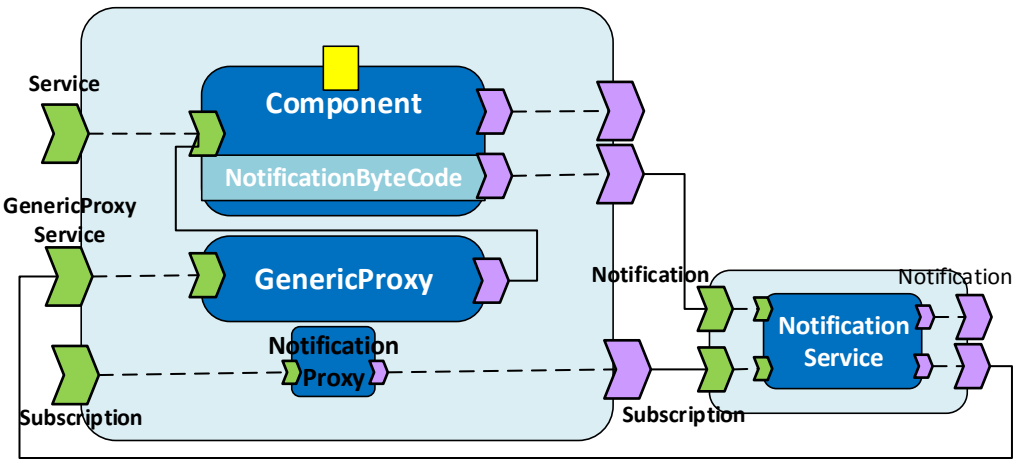


Figure 4.8: Monitoring by subscription using hybrid schema.

that forwards the incoming subscriptions to the related Notification Service component. This component is deployed once and instantiated as much as needed. This hybrid schema generalizes the two previously explained schemas (one and multi *Notification Service* monitoring). To emulate the multi *Notification Service* schema we can associate one instance of this component to each monitored component. For the one *Notification Service* schema, we can use just one instance for all the monitored components.

4.2.6 Transformation Example:

Going back to the example previously described in paragraph 4.2.1, we would like to transform the *ComponentB*, that was designed without monitoring facilities, to render it monitorable by subscription on change. After applying the needed transformations on this component, we get a new composite offering the *ServiceOfB* and new monitoring services. According to the used schema, the resulting composite could be represented as it is shown in Figure 4.6 and Figure 4.7.

The SCA ADL description of the newly created composite is described in Listing 4.2 following the multi-channel schema. This description is conform to the standard SCA description that could be instantiated using any SCA runtime.

Listing 4.2: Description of the Monitored Component after its transformation (multi-channel schema) using SCA ADL

```
1 <composite name="TransformedBComposite">
2  <service name="ServiceOfB" promote="ComponentB/ServiceOfB" />
3  <service name="GenericProxyService" promote="GenericProxy/GenericProxyService" />
4  <service name="SubscriptionService" promote="NotificationService/SubscriptionService" />
5  <reference name="NotificationService" propmote="NotificationService/Notification"/>
6
7  <component name="ComponentB">
```

```
8   <service name="ServiceOfB">
9    <interface.java interface="example.ServiceOfBInterface"/>
10  </service>
11  <implementation class="example.impl.ModifiedServiceOfBImpl"/>
12  <reference name="notification" target="NotificationService/NotificationService"/>
13  </component>
14
15  <component name="GenericProxy">
16  <service name="GenericProxyService">
17   <interface.java interface="GenericProxyInterface"/>
18  </service>
19  <implementation class="impl.GenericProxyImpl"/>
20 </component>
21
22 <component name="NotificationService">
23  <service name="SubscriptionService">
24   <interface.java interface="SubscriptionServiceInterface" />
25  </service>
26  <service name="NotificationService">
27   <interface.java interface="NotificationServiceInterface"/>
28  </service>
29  <reference name="GenericProxyService" target="GenericProxy/GenericProxyService"/>
30  <reference name="Notification"/>
31  <implementation class="impl.SubscriptionImpl"/>
32 </component>
33
34 </composite>
```

This same example could be presented following the one *Notification Service* monitoring schema. In this case the result of the transformation is two separate composites encapsulating the monitored component and the added components of monitoring.

Listing 4.3: Description of the Monitored Component after its transformation (one channel schema) using SCA ADL

```
1 <composite name="TransformedBComposite">
2  <service name="ServiceOfB" promote="ComponentB/ServiceOfB" />
3  <service name="GenericProxyService"promote="ComponentB/GenericProxyService" />
4  <reference name="Notification" promote="ComponentB/Notification"/>
5
6  <component name="ComponentB">
7   <service name="ServiceOfB">
8    <interface.java interface="example.ServiceOfBeInterface"/>
9   </service>
10  <implementation class="example.impl.ModifiedServiceOfBImpl"/>
11  <reference name="notification" target="NotificationServiceComposite/Notification"/>
12  </component>
13
14  <component name="GenericProxy">
15  <service name="GenericProxyService">
16   <interface.java interface="GenericProxyInterface"/>
17  </service>
```

```
18   <implementation class="impl.GenericProxyImpl"/>
19  </component>
20 </composite>
21
22 <composite name="NotificationServiceComposite">
23  <service name="Notification" promote="NotificationService/Notification"/>
24  <service name="Subscription" promote="NotificationService/Subscription"/>
25  <reference name="GenericProxyService" promote="NotificationService/GenericProxy" />
26  <reference name="Notification" promote="NotificationService/Notification"/>
27
28  <component name="NotificationService">
29   <service name="Subscription">
30    <interface.java interface="SubscriptionInterface"/>
31   </service>
32   <service name="Notification">
33    <interface.java interface="NotificationInterface"/>
34   </service>
35   <reference name="GenericProxy" target="TransformedBComposite/GenericProxyService"/>
36   <reference name="Notification"/>
37   <implementation class="impl.Subscription"/>
38  </component>
39 </composite>
```

At this stage, we did not treat yet the scalability issue related to Cloud environments. The next section show how we tackle this issue using a framework based on scalable micro-containers. We will describe this framework that we use to add monitoring to components and to deploy them in the Cloud.

4.3 Monitoring and Reconfiguration within Scalable Micro-containers

In [66], we introduced a new scalable and platform independent micro-container that enables components' deployment and execution in the Cloud. In this Section, we start by giving an overview of the deployment framework and the generated micro-container. In this work we want to add monitoring and reconfiguration capabilities to this micro-container.

4.3.1 Deployment Framework and generated Micro-Container

We designed a deployment framework able to generate light weight containers that we call micro-containers. As shown in Figure 4.9, this framework contains basically a processing module to ensure minimal micro-containers generation process. This module coordinates the generation of micro-containers based on an SCA descriptor. It adds the needed communication and processing modules (HTTP, RMI or other communication mechanisms) to be used by the component in order to communicate with clients or other components. Communication modules are chosen from existing generic modules available in the Communication Generic Package of the Deployment Framework. To assemble these elements, the Processing Module uses the Assembly Module to generate a micro-container. The deployment framework is modular. Consequently, we can plug new modules to respond to new requirements. The basic modules that make up the architecture of the generated micro-container are those needed to

ensure its minimal functionalities resumed at: 1) hosting a component and its context, 2) enabling the communication with clients and 3) query marshalling and demarshalling. The two latter functionalities are ensured by a Communication and Processing Component (CPC) dynamically generated based on the chosen communication module.

4.3.2 Adding Monitoring and Reconfiguration to Micro-Containers

To add monitoring and reconfiguration capabilities to the micro-container, we use the component model that we presented in paragraph 4.2.1 to represent components. Since some components can be designed without monitoring and reconfiguration facilities, we integrated the transformations presented in subsection 4.2 to render these components monitorable and reconfigurable. And in order to integrate these transformations in our framework, we added the monitoring module to the deployment framework. This latter contains several modules shown in Figure 4.9 and are as follows: 1) Processing Module: orchestrates all the steps to generate a micro-container, 2) Generic Communication Module: contains different modules implementing communication protocols, 3) Monitoring Module: supports different monitoring models, and 4) Assembly Module: generates a micro-container with monitoring capabilities.

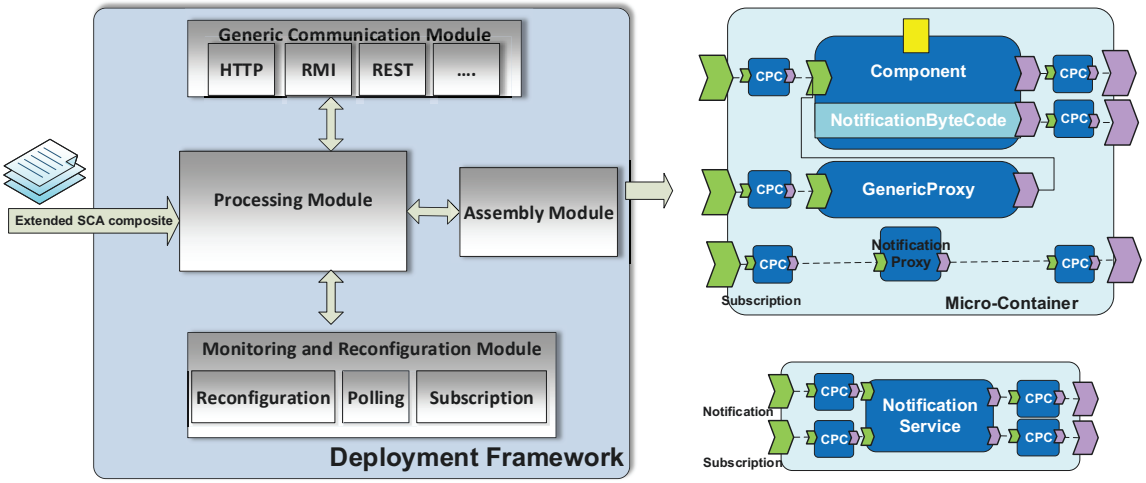


Figure 4.9: Extension of the Micro-container architecture with monitoring.

To generate a micro-container with a component hosted in, one must provide the needed implementation of the component and a descriptor describing how to assemble and deploy the micro-container into a Cloud environment. The processing module sends directly this descriptor to the assembly module before analyzing the implementation and generating the corresponding description. Then, the processing module detects the components' binding types and monitoring aspects. It selects the communication modules implementing the described bindings available at the Generic Communication Module and uses the chosen monitoring module to apply the needed transformations on the component. Finally, it sends the new resulting implementation to the assembly module whose responsibility is to generate the new

micro-container enhanced with monitoring capabilities. Prior deploying the generated micro-container, the *Deployment Framework* verifies if there is an already deployed micro-container containing the *Notification Service* composite. If it is not the case, it generates a new micro-container encapsulating the generic *Notification Service* composite. This is to assert that the generic *Notification Service* is deployed once and instantiated as much as needed.
As shown in Figure 4.9, the Monitoring and Reconfiguration Module supports reconfiguration, monitoring by polling and monitoring by subscription with its two modes: on interval and on change. Monitoring on change mode includes Property Changed Monitoring, Method Call Monitoring and Time Execution Monitoring.

The generated micro-container (Figure 4.9) is responsible of holding its service, managing its communication with the clients, and processing all incoming or outgoing messages. These communication and processing aspects are handled using the Communication and Processing Component (see CPC component in Figure 4.9). Moreover, it is important to notice that the client can interact with the micro-container either to invoke the contained service, to reconfigure the component properties, or to request monitoring information. It can also send subscription requests to receive notifications on change or on interval.

4.4 Adding FCAPS properties to components

Management of service-based applications in Cloud environments is becoming a challenging task. Particularly, fault, configuration, accounting, performance and security (FCAPS) management can play an important role to respect the agreement between the service's consumer and its provider. In fact, adding FCAPS management can improve the resiliency and autonomy of our container specially if we enforce it with a granular monitoring service. A mobility service can also enhance the management since it allows the migration of services between virtual machines or between different platforms.

We already extended SCA model to allow components to express their requirements for monitoring and reconfiguration of properties. We added a new *RequiredProperty* element to the SCA model. For the FCAPS requirements we use the *"requires"* attribute defined by SCA. This term is to say that the related component or service requires a non functional facility provided by an existing composite defined as an *"intent"*. In Listing 4.4, the *componentA* specifies that it requires (Line 3) the usage of *FaultIntent* and *AccountingIntent*. In the same Listing, the *componentB* requires the usage of *ConfigurationIntent* (Line 14).

Listing 4.4: Extended SCA example

```
1 <composite name="InputSampleComposite">
2  <service name="serviceComponentA" promote="ComponentA/serviceA"/>
3
4  <component name="ComponentA" requires="FaultIntent, AccountingIntent">
5   <service name="serviceA" >
6    <interface.java interface="example.serviceA"/>
7   </service>
8   ...
9   <requiredProperty resource="ComponentB.propertyOfB" monitoring="BySubscription" notificationMode="
        ON_CHANGE">
```

```
10     <property name="propertyOfB"/>
11    </requiredProperty>
12    ....
13   </component>
14
15   <component name="ComponentB" requires="ConfigurationIntent">
16    <property name="propertyOfB">
17    <service name="serviceB">
18     <interface.java interface="example.ServiceBInterface"/>
19    </service>
20     ........
21   </component>
22
23 </composite>
```

A SCA *"intent"* allows to add non functional facilities and links them with the appropriate composite or component responsible of their processing. Consequently, we have to prepare the description of the composite responsible of the execution of the non functional facility. In Listing 4.5 we show an example of a composite playing the role of an intent for Fault tolerance processing. Respectively, we defined the intents that we will use all over our work for the different FCAPS services. Each intent refers to a handler describing the processing mechanisms for a non functional facility. These intents could be described as Java implementation or Web Services, etc. For example, as shown in Listing 4.5 (Line 7), we specify the Java class implementing the intent responsible of Fault tolerance. Afterwards, any component can use this intent to use its provided non functional facility by referring to it via the *"requires"* attribute.

Listing 4.5: SCA intent example

```
1 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" name="FaultIntent">
2  <service name="faultIntentService" promote="faultIntentComp/faultIntentService"/>
3  <component name="faultIntentComp">
4   <service name="faultIntentService">
5    <interface.java interface="eu.telecom.intent.IntentHandler"/>
6   </service>
7   <implementation.java class="eu.telecom.intent.FaultIntentHandlerImpl"/>
8  </component>
9 </composite>
```

4.4.1 Extension of Deployment Framework for FCAPS management

In order to support the deployment of SCA components extended with monitoring, mobility and FCAPS management, we extended the Deployment Framework to add these facilities to our containers. We call these extended containers Intelligent Managed Micro-Containers (IMMCs). An IMMC is a light-weight container able to host an SCA component having self management capabilities. As shown in Figure 4.10, the extended framework is responsible of generating specific IMMCs for components. The Processing Module receives the extended SCA descriptor of the composite. It applies the needed monitoring transformations to add monitoring facilities to the service [32]. Then, it adds FCAPS management functionalities following the management descriptor. Besides, it adds the mobility service and the needed

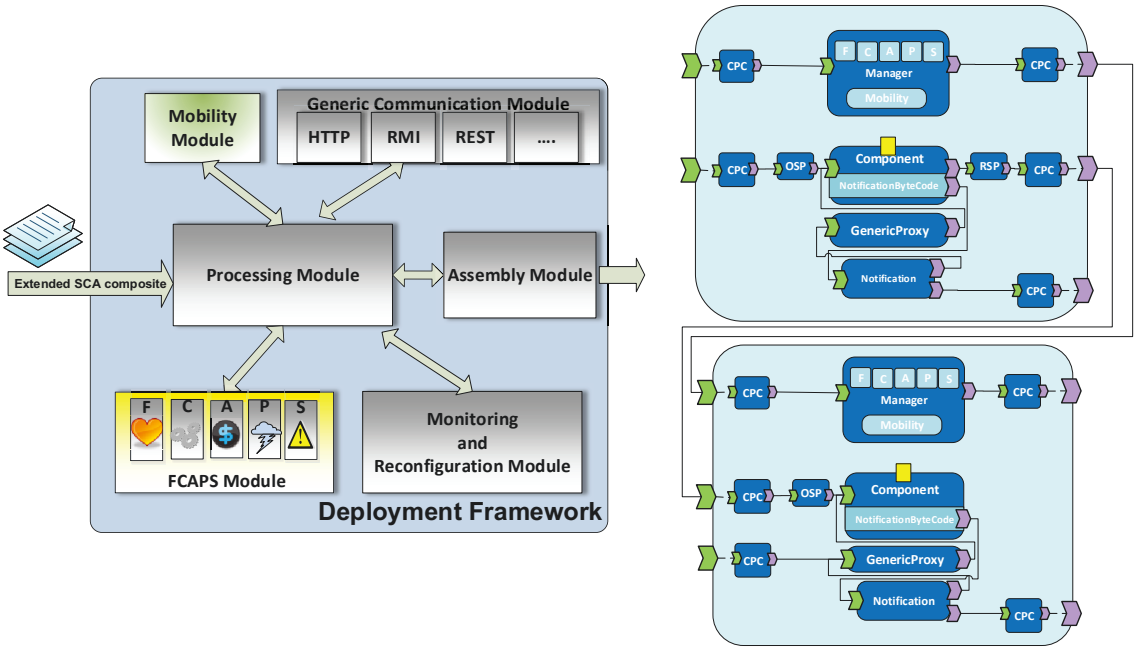


Figure 4.10: Extended Deployment Framework and Micro-Container for FCAPS management.

communication module. Finally, it generates an IMMC that could be deployed and instantiated in the Cloud. The generated IMMC has an FCAPS Manager able to consume monitoring data and to take decisions (see Section 4.4.4).

4.4.2 IMMC Anatomy

The IMMC is composed of different components as shown in Figure 4.11. The Communication and Processing Component (CPC) is responsible of the interactions with the outside (i.e., sending and receiving of messages). It performs also packing and unpacking of the incoming and outgoing messages to the specific message formats. Monitoring, Mobility and FCAPS management are performed using the appropriate component (respectively detailed in 4.2.5, 4.4.3.2 and 4.4.4).

Basically, the IMMC contains a composite offering the services of the original component with new services of monitoring offered by a dynamically generated component (described in Section 4.2).

Additionally, for each component to be deployed in an IMMC, we dynamically generate two proxies. The first one called *Offered Service Proxy* (OSP) and it offers the same service as the concerned component. It handles the incoming queries, adds specific functionalities (detailed in 4.4.4), and forwards the call to the component. The second proxy called *Required Service Proxy* (RSP) offers the same service as the one required by the component. It handles the outgoing calls, adds specific functionalities (detailed in 4.4.4), and forwards these calls to the called IMMC. These two proxies are important to ensure the

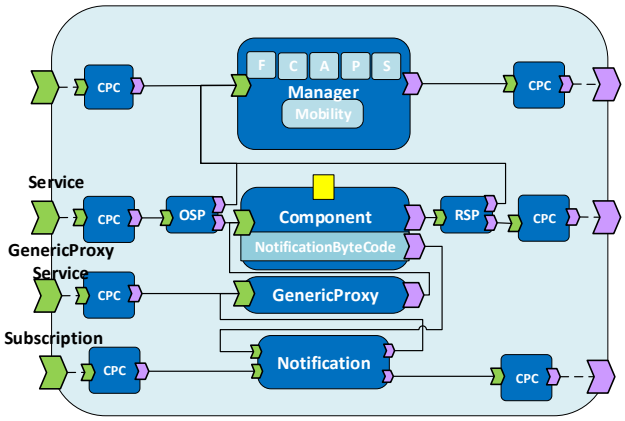


Figure 4.11: Intelligent Managed Micro Container Anatomy

rest of the management functionalities.

4.4.3 FCAPS Manager and IMMC basic functionalities

IMMC instances are able of deciding to apply create, replace, update and delete functionalities on themselves using their FCAPS Managers. The FCAPS manager (shown in Figure 4.11) is responsible of the different functionalities of FCAPS Management. The Fault, Configuration, Accounting, Performance and Security functionalities are encapsulated in the IMMC, following the description contained in the extended SCA composite description. When needed, the FCAPS Manager is able to request the Deployment Framework to instantiate a new IMMC in a specific virtual machine. It can also request replacing an existing IMMC by a new one. For example, if an IMMC is faulty, the FCAPS Manager sends a request to the Deployment Framework asking to replace the faulty IMMC with a healthy one. Moreover, this Manager has the ability to change the implementation of the encapsulated component. Finally, it can request the Deployment Framework to delete an existing IMMC. For all these operations, the Deployment Framework notifies the concerned IMMCs in order to update all the bindings to conserve the consistency of the application.

4.4.3.1 IMMC Replication and Load Balancing

The FCAPS Manager can decide to perform replication of IMMCs. Using the previously defined basic functionalities (described in Section 4.4.3), this manager asks the Deployment Framework to create a new instance of the concerned IMMC. Since this framework has all the needed information about the contained component, it instantiates a new IMMC with the same component and running services as the original one. In order to render useful this mechanism, we defined a Load Balancer component. This component could be deployed in a new IMMC and related to the different replicas of an IMMC. As shown in Figure 4.12, this Load Balancer is acting like a proxy providing the same services provided by the

replicated IMMC and requiring the services offered by the component. It has a specific table containing a list of the different IMMC services. It intercepts the different calls to the service, and forwards them to a specific IMMC instance following a specific strategy (e.g, Round Robin, Random). The Load Balancer is generated dynamically to fit the replicated IMMC business functionalities. Different strategies could be used by the FCAPS Manager in order to decide when to duplicate (add new instance) an overloaded IMMCs and when to consolidate (retrieve an instance) underloaded ones. In our work, the main strategies that we used are described in [67]. It was formally proved that these strategies maintain the semantics of the application. Figure 4.12 shows an example of replicating an IMMC and using a Load Balancer.

In Figure 4.12, we show the three sequences of the proposed example of replication and load balancing (noted (1), (2) and (3)). In (1), we show the initial step where we have two IMMCs hosting two components. ComponentA requires the use of the service provided by ComponentB. FCAPS Managers of the two IMMCs are also connected. In (2), the IMMC containing ComponentA notices that the second IMMC is overloaded. Consequently, it sends a *Replicate* request to the Deployment Framework. As shown in (3), the Deployment Framework replicates the overloaded IMMC and places a new IMMC containing a LoadBalancer to balance the queries between the two IMMCs.

4.4.3.2 Mobility

In [68], we proposed an extension for our micro-container that enhances it with mobility. To do that, we added a Mobility module responsible of adding a Mobility component to micro-containers. Here, we propose to delegate the mobility decisions to the FCAPS Manager of the IMMC. In fact, each FCAPS Manager is able to take decisions of migration for reasons of performance, scalability or energy efficiency.

As shown in Figure 4.13, whenever there is a need to migrate an IMMC, the FCAPS Manager of the concerned IMMC or any other IMMC connected to this latter can take the decision to send a migrate query to the Deployment Framework. This framework serializes the IMMC and instantiates it in the targeted virtual machine. Afterward, it updates all the bindings between the concerned IMMCs of the application. Finally, when it receives the acknowledgement of the binding updates, it deletes the old instance of the IMMC.

Figure 4.13 shows an example of migrating an IMMC and updating the binding accordingly (noted as (1), (2) and (3)). In (1), we show the initial sequence where two linked IMMCs are located in the same VM. In (2), The IMMC on the top notices that the other IMMC needs a migration to another VM. Consequently it sends a *Migrate* query to the Deployment Framework. This latter fetches the existing VMs and deploys the concerned IMMC in the available VM as shown in (3). Afterwards, the Deployment Framework updates the bindings of the IMMCs using the migrated one and deletes the old IMMC.

4.4.4 FCAPS Management

FCAPS management functionalities are added to IMMCs according to their SCA descriptors. In order to have an adaptable system, these functionalities could be adapted during the runtime. In the following, we will detail further the different FCAPS management functionalities.

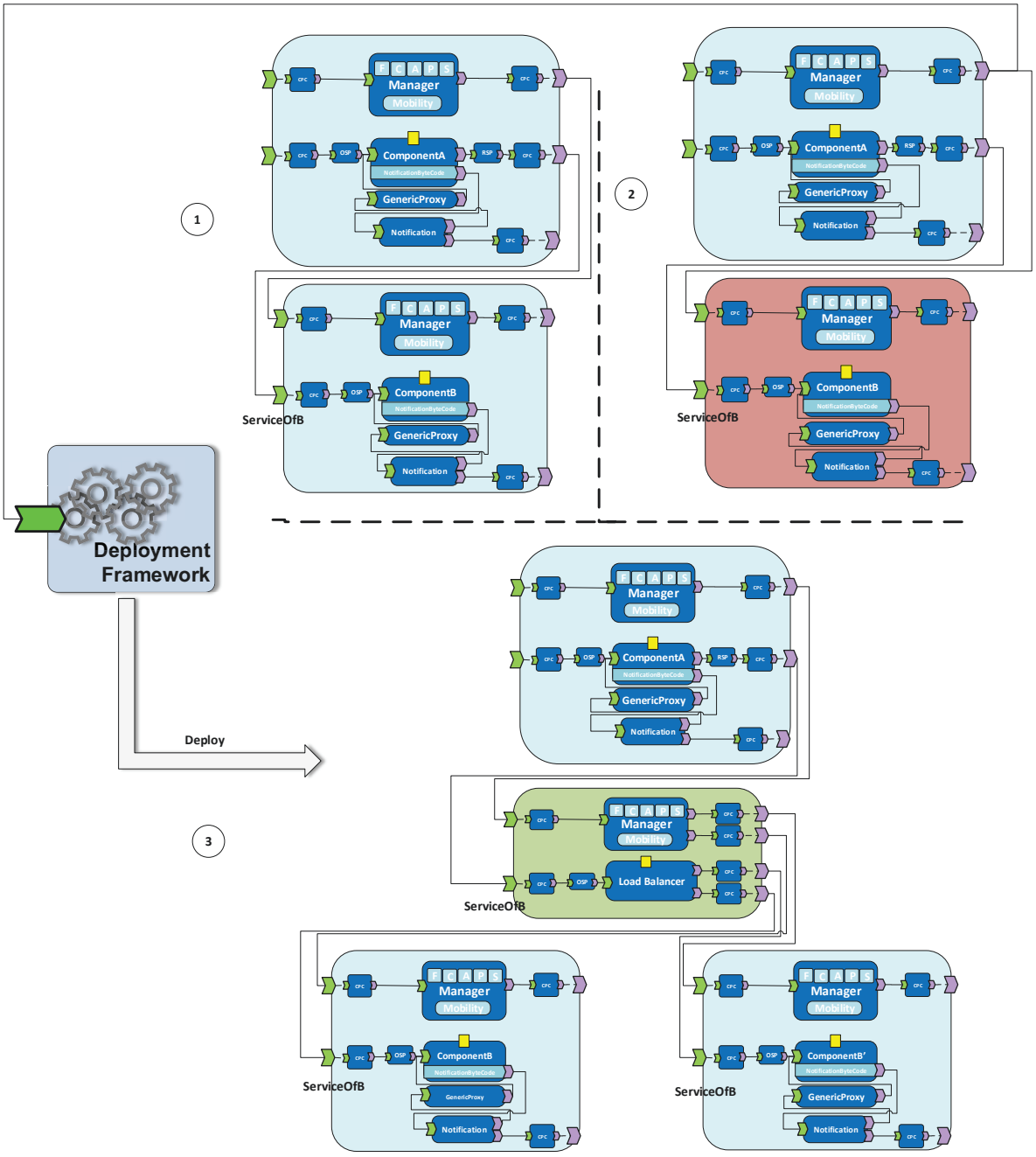


Figure 4.12: Replication and Load balancing scenario.

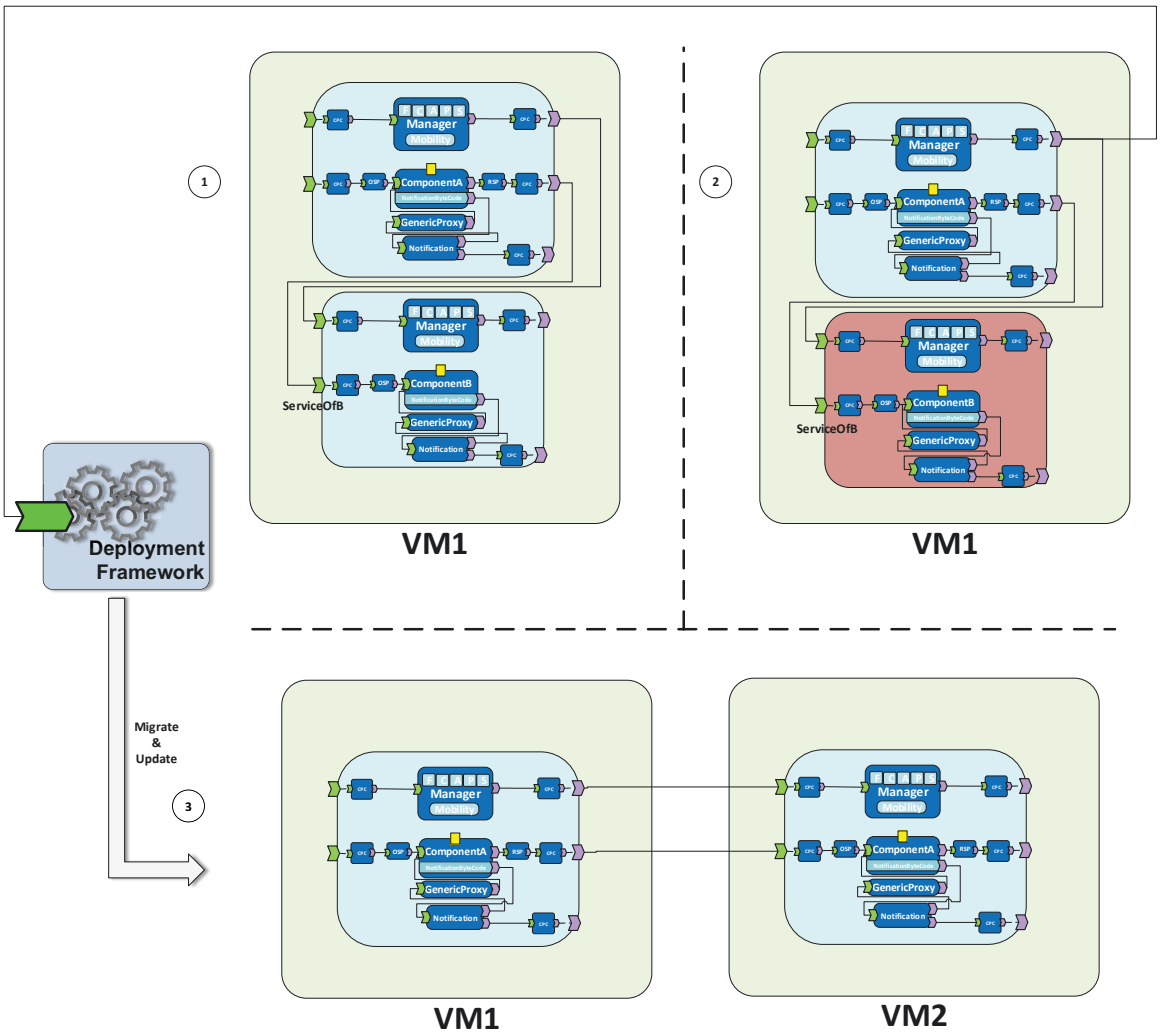


Figure 4.13: Migration scenario.

4.4.4.1 Fault

The Fault functionality of the FCAPS Manager is used to ensure the execution continuity of the service in an internal view (i.e., the FCAPS Manager is responsible of the management of its IMMC) or in an external view (i.e., when using a composition of IMMCs, an FCAPS Manager can monitor other IMMCs and take management decisions). This mechanism is based on the two added proxies OSP (Offered Service Proxy) and RSO (Required Service Proxy). In fact, the first proxy OSP intercepts the incoming

calls and forwards them to the called service and gets back the response. If it detects that the component is not responding it notifies the FCAPS Manager. This latter re-instantiates the faulty component and sends back its reference to the OSP to update its reference and to use the newly instantiated component. The OSP updates its references and continues its task to invoke the service and return the response to its caller. In a component based application, a component can detect that another component is not responding. In this case the RSO is responsible to detect that. In fact, the RSO intercepts all the outgoing calls of the managed component. It forwards them to the called component, gets back the response and forwards it to the encapsulated component. If it detects that the required IMMC is not responding, it notifies the FCAPS Manager. This latter may use its basic functionalities to request the instantiation of a new IMMC, to notify all the related IMMCs to update their references and finally after being notified from all these IMMCs that they updated their references, the faulty IMMC is deleted. After receiving the update notification, the FCAPS Manager notifies its RSP to update its references. This latter continues, then, its invocation to the newly instantiated IMMC and returns the response to the encapsulated component. In some cases, we can have a concurrence scenario of healing an IMMC. For example, if two IMMCs request to heal the same IMMC at the same time. To resolve this problem, we defined a Relaxation Time (RT) that allows the Deployment Framework to ignore any request to heal an IMMC that is being replaced. This RT is estimated based on the average time needed to replace an IMMC.

4.4.4.2 Configuration

The configuration can be done at the init time of the IMMC or at runtime. Runtime configuration (reconfiguration) is based on configuration strategies described for the SCA intent responsible of this functionality. Configuration strategies could be adapted at runtime. Our framework enables all the classes of reconfigurations that we defined in Section 2.3.2. To ensure the Structural reconfiguration, the basic functionalities of the FCAPS Manager allows it to decide when to create new IMMCs and to delete existing ones. Before deleting an IMMC, the Deployment Framework has to notify all the related IMMCs to update their references. These IMMCs update their references and update their RSPs. For the Geometric reconfiguration, the FCAPS Manager uses the mobility mechanism to migrate IMMCs from a location to another. Accordingly, it has to notify the related IMMCs so they can update their references to the migrated IMMC. Furthermore, the Implementation reconfiguration is enabled locally for each IMMC. The local FCAPS Manager of each IMMC is able to change the implementation of the encapsulated component. Accordingly, it has to update the OSP reference to the new instance. Finally, the Behavioral reconfiguration is enabled by the dynamically generated *GenericProxy* component (described in Section 4.2). This latter, allows to reconfigure an encapsulated component by modifying its attributes.

4.4.4.3 Accounting

This functionality consists on applying a specific strategy of accounting by the FCAPS Manager component. Accounting strategies could be based on different metrics as the resources consumption, the time of a transaction or even the number of services' invocations. These information could be retrieved from the notification component of each IMMC. To receive monitoring data concerning the IMMC usage, the FCAPS Manager component may subscribe to the Notification Service to receive notifications concerning the Execution Time or the Method Call number. Each IMMC uses its Accounting functionality to

collect monitoring data related to the encapsulated component. Accordingly, the FCAPS Manager component has all the needed information of the consumption of the IMMC. It has a local table containing all the needed data for accounting. Among others, this table contains data about the memory, CPU consumption of the IMMC. It saves also the incoming calls number and the total usage time of the IMMC. The FCAPS Manager can apply aggregation rules to generate a more consolidated accounting information. Accounting rules are defined for the intent responsible of this functionality in the SCA descriptor at deployment time and could be adapted at runtime. These rules may be specified as a formula that calculates the usage price or as a call to an external service that is in charge of this task.

4.4.4.4 Performance

The Performance functionality has a twofold role. It aims to enhance the performance of the IMMC and to optimize the consumed resources. It is a critical functionality of the FCAPS Manager in the elasticity of our solution. This functionality needs that the FCAPS Manager consumes monitoring data by subscribing to the Notification component. It receives information about the execution time of services, the resource consumption and the number of simultaneous service calls. Based on these information, it detects if there is a need to enhance the performances of the IMMC. If the response time of the component is greater than a given threshold, there are two possible causes for this degradation. It may be caused by the important number of simultaneous calls targeting the IMMC. In this case, the FCAPS Manager decides to replicate the overloaded IMMC. It instantiates a new IMMC with the same characteristics, places a Load Balancer IMMC, bind this latter to the two instances of the target IMMC and makes it intercept the simultaneous calls and balance them between the two instances of the IMMC. The second case, is when the response time is greater than the given threshold with a small number of simultaneous calls. In this case, the FCAPS Manager decides to migrate the IMMC to a more efficient virtual machine. Otherwise, the Load Balancer IMMC has a particular role that enhances the elasticity of our solution. It can detect that the number of used IMMCs is greater then what is required. In which case, its FCAPS Manager decides to reduce the number of used IMMCs. The FCAPS Manager updates the bindings of the related IMMCs and deletes the unused one. If the Load Balancer IMMC is not needed anymore, the FCAPS Manager requests to delete this IMMC. The Deployment Framework let the related IMMCs update their references to the targeted IMMC then delete the Load Balancer.

4.4.4.5 Security

The security functionality allows the FCAPS Manager to verify whether the incoming queries are authenticated and authorized to do what they aim to do. In fact, this functionality is based on the OSP that we introduced previously. In this case, this proxy intercepts the incoming queries and verifies their authentication and authorization. If the query is authenticated and authorized it forwards the query and sends back the response. Else it returns a response message to say that the query is not authorized. In our future work we can enhance this component by adding more security strategies (i.e., DDoS detection).

4.5 Conclusion

Adoption of Cloud platforms to host Service-based applications is increasing more and more. This type of applications could be described as SCA composites containing all the information of the application. The need of managing this kind of applications along with the dynamic nature of Cloud environments favored our research to describe suitable management mechanisms that could be used at different granularities and in a scalable manner. In this chapter, we proposed an extension of SCA Component model to allow the description of non functional facilities of monitoring and reconfiguration as well as Fault, Configuration, Accounting, Performance and Security management. However, to face the scalability issues of Cloud environments, we proposed a framework that adds management facilities to components automatically. This framework, packages the modified components in scalable Micro-Containers that could be deployed independently in the Cloud. We made our best effort to make this framework extensible so we can add new functionalities when needed.

After facing the challenges of adding non functional facilities to SCA-based applications, we were motivated to extend our work to cover different kinds of Cloud resources. We are interested in adding monitoring and reconfiguration facilities to any type of Cloud resources independently from its level (whether it is in IaaS, PaaS or SaaS). These facilities could form the pillar blocks in order to provision autonomic Cloud resources. In the following chapter, we will propose a level agnostic model to add monitoring and reconfiguration facilities to Cloud resources. These facilities are used to build an autonomic infrastructure on demand for any resource in the Cloud.

Chapter 5

Monitoring and Reconfiguration of Cloud Resources

Contents

5.1 Introduction 59
5.2 Monitoring and Reconfiguration for Cloud Resources 60
5.2.1 Adding Monitoring and Reconfiguration facilities to Cloud Resources 60
5.2.2 OCCI description for Monitoring and Reconfiguration 61
5.2.3 Example of using OCCI Monitoring and Reconfiguration extension 67
5.3 Autonomic Management for Cloud Resources 67
5.3.1 MAPE: Monitor, Analyze, Plan and Execute 68
5.3.2 MAPE for Cloud Resources 68
5.3.3 OCCI extension for Autonomic Management 70
5.4 Autonomic Infrastructure for Cloud Resources 77
5.5 Adding FCAPS Management to Cloud Resources 78
5.6 Conclusion 79

5.1 Introduction

In the previous chapter, we proposed a new approach to add monitoring and reconfiguration facilities to SCA-based applications prior deploying them in the Cloud. For deployment aspects, we used a Deployment Framework that grants a light weight container (micro-container) to each component. We also extended our approach to add more autonomy and self-management capabilities to the used micro-containers. To go beyond the boundaries of SCA-based applications, and to be agnostic from the Cloud resource level, we propose in this chapter an approach that adds monitoring and reconfiguration facilities

to Cloud resources independently of their level (i.e., IaaS, PaaS or SaaS). Later on, we extend this approach to use monitoring and reconfiguration facilities as pillar blocks to build an autonomic infrastructure attached to any Cloud resource. We describe our model as an extension of Open Cloud Computing Interface (OCCI). Hereafter, we will detail all the newly added Resources, Links and Mixins that enables the establishment of the aforementioned autonomic infrastructure.

5.2 Monitoring and Reconfiguration for Cloud Resources

In this section, we define our generic approach to add monitoring and reconfiguration facilities to Cloud resources. This approach is a generalization of our approach of monitoring and reconfiguration for SCA-based applications. We start by defining our model for monitoring and reconfiguration. Then, we describe this model based on Open Cloud Computing Interface specifications. Afterwards we propose an example to show the usage of our proposal.

5.2.1 Adding Monitoring and Reconfiguration facilities to Cloud Resources

In this section, we will explain the needed functionalities to be added to Cloud resources in order to enable their monitoring and reconfiguration. We will define these functionalities as extensions to Cloud resources in a generic manner. These extensions could be described later using any description model. In the rest of this work, the extended resources (with monitoring and reconfiguration) are referred to as managed resources.

5.2.1.1 Adding Reconfiguration for Cloud Resources

A Cloud resource may require to reconfigure another resource in order to ensure a global objective of the system. That implies that the reconfigured resource proposes an interface that allows to reconfigure it. However, not all Cloud resources are designed with reconfiguration facilities. And, some resources may not provide any interface to enable the reconfiguration of their properties. We propose to extend any Cloud resource to allow its configuration by other resources. The extension consists on adding the needed functionalities to change the properties of this resource to become reconfigurable. The class diagram in Figure 5.1 shows the defined extension for reconfiguration. The newly added functionalities allow a Cloud resource to reconfigure the managed resource (extended with Reconfiguration functionalities) by changing its attributes values or by applying more complicated reconfiguration functions.

5.2.1.2 Adding Monitoring by Polling for Cloud Resources

A Cloud resource may require to monitor by polling another resource to get its status and to take decisions accordingly. In this case, the monitored resource allows monitoring its status by mean of defined interfaces of monitoring by polling. However, not all Cloud resources are designed with monitoring capabilities. And some resources may not provide any interface to enable monitoring of their properties. We propose to extend any Cloud resource by adding monitoring by polling facilities. The extension consists on adding the needed functionalities to get the status of the managed resource (resource enhanced

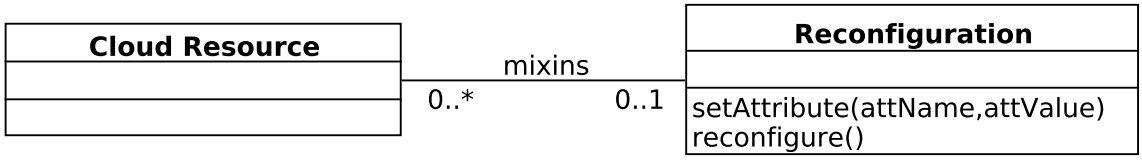


Figure 5.1: Reconfiguration extension for Cloud Resources.

with monitoring). Afterwards, any other resource can monitor the managed resource using its new extension. The class diagram in Figure 5.2 shows the defined extension for monitoring by polling. A resource can use this extension to get all the monitorable properties of the managed resource as well as the value of each property.

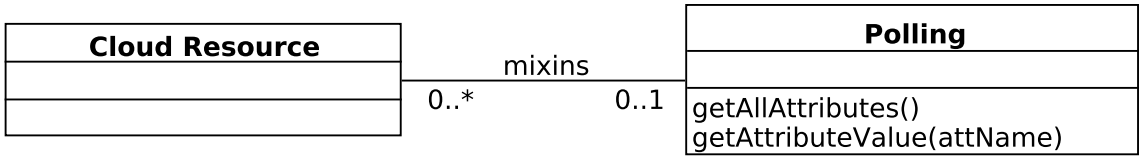


Figure 5.2: Monitoring by Polling extension for Cloud Resources.

5.2.1.3 Adding Monitoring by Subscription for Cloud Resources

A Cloud resource may require to monitor by subscription another resource. That means that the monitored resource provides the needed interfaces to receive and manage subscriptions for its properties. Since not all Cloud resources are designed with these facilities, we propose to extend them with monitoring by subscription. The extension allows any resource to subscribe on one or more properties. The subscription results on receiving eventual notifications about the status of the concerned property. Notifications could be periodic or on change of the property. The class diagram in Figure 5.3 shows the defined extension for monitoring by subscription. A resource can subscribe to a specific property, it can unsubscribe or update its subscription. The *Subscription* extension should save the list of subscribers that need to be notified whenever needed.

5.2.2 OCCI description for Monitoring and Reconfiguration

In Section 5.2.1 we defined the needed extension for Cloud resources to enable monitoring and reconfiguration facilities. Our definition was generic, that could be described using any description model. For our solution to be accepted in the Cloud community, we made the choice to use the standard OCCI

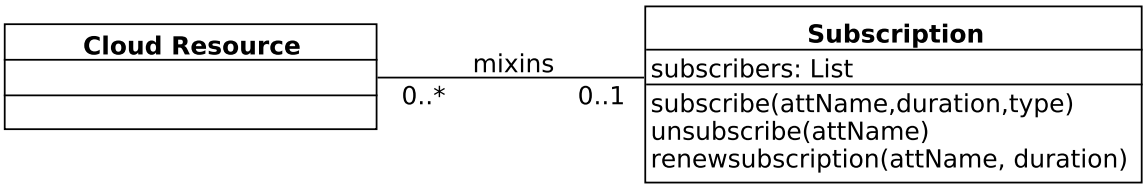


Figure 5.3: Monitoring by Subscription extension for Cloud Resources.

to describe our resources and extensions. Indeed, OCCI provides an extensible model based on Mixin mechanism. A Mixin is an extension that could be added to any Cloud resource in order to define new functionalities or attributes. In the following we describe the different elements that we need to add monitoring and reconfiguration facilities to Cloud resources based on OCCI. These elements are new Mixins to extend any Resource as it is defined in OCCI Core Model [2]. We also define new Links to enable monitoring and reconfiguration of remote resources based on different communication mechanisms. These Links extend the Link base type defined in OCCI Core Model [2]. They are described as abstract resources that could be specified using newly defined Mixins. OCCI fits well our needs to express the newly defined extensions for Cloud resources. In the following we describe the different Mixins and Links that we defined for monitoring and reconfiguration. The newly defined Mixins are shown in Figure 5.4. Moreover, the newly defined Links are shown in Figure 5.5.

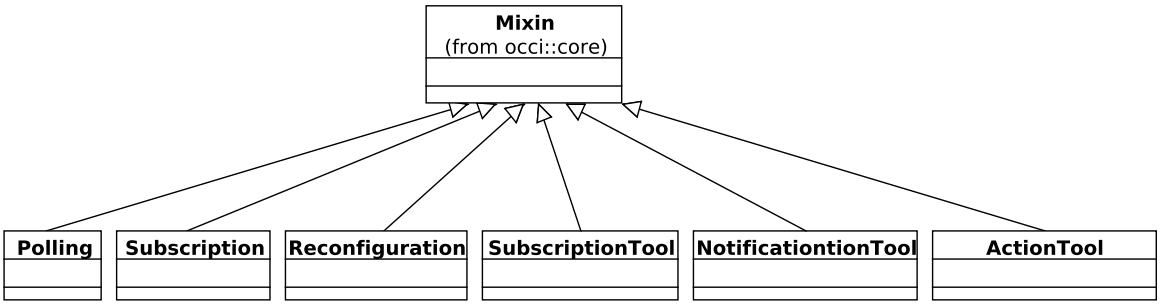


Figure 5.4: OCCI Mixins for Monitoring and Reconfiguration.

5.2.2.1 Reconfiguration Mixin

It provides the needed functionalities to ensure reconfiguration. By extending a Resource with this Mixin, we add new methods that render a resource configurable. Technically, this Mixin implements a *ReconfigurationInterface* that defines three abstract methods. The method *setAttributeValue(attributeName)* changes the value of a given attribute. The *getActions()* returns the list of actions that one could apply

on the managed Resource. The *invokeAction(action)* invokes a given action on the Resource. We also defined a *reconfigure()* method that allows to implement a specific way to reconfigure a Resource. For a specific use case, the implementation of this Mixin must describe exactly how to realize these actions. For example, we can implement the *reconfigure()* method to resize the disk of an OCCI *Compute* instance. In this case, the implementation can be a REST query to the infrastructure manager (e.g., OpenNebula or OpenStack) to resize the disk. It can also be a usage of libvirt API [1] through the *virt-resize* command that allows to resize a given disk.

5.2.2.2 Polling Mixin

It describes the needed operations to monitor a Resource by polling. By extending a Resource with this Mixin, we add a list of actions that render a Resource monitorable by polling. This Mixin is an implementation of the interface *PollingInterface* that describes two abstract methods *getAttributes()* and *getAttributeValue(attributeName)*. The first method *getAttributes()* returns the list of the attributes of the managed Resource. The second method *getAttributeValue(attributeName)* returns the value of a given attribute. These two methods are implemented by *Polling* Mixin according to the specific use case. For example, to be able to monitor by polling the memory usage of an OCCI *Compute* instance, the *Polling* can be implemented as a call to an external program able to retrieve this data or it can be a remote UNIX command "top | grep KiB" assuming that the Compute is a Linux instance.

5.2.2.3 Subscription Mixin

It allows an interested part to subscribe for monitoring data and to receive notifications. For each Resource that we want to manage, we add a *Subscription* Mixin to manage subscriptions on specific metrics in order to send notifications containing monitoring data. Notifications may be on interval so that the client receives a notification periodically containing the state of one or more metrics. It may also be on change, so the *Subscription* Mixin pushes the new state of the metric whenever a change occurred. This Mixin implements the interface *SubscriptionInterface* that defines a list of actions to manage clients subscriptions (i.e., subscribe(), unsubscribe(), etc.). The implementation must define how to realize these actions according to a specific scenario. For example, the *subscribe()* method could be implemented as an HTTP Restlet that can receive subscriptions via REST queries. It is worth noting that this Mixin plays the same role as the *Notification Service* proposed in Chapter 4.

5.2.2.4 Subscription Link

It is a Link between a subscriber (see *source* in Table 5.1)and the Resource to be monitored (see *target* in Table 5.1). It models the occurrence of subscriptions on specific metrics. A *Subscription Link* is related to one consumer that requires monitoring information and one managed Resource that have a *Subscription* Mixin. A Subscription is specified with a Mixin instance from the *SubscriptionTool* Mixin collection (described later).

[1] http://libvirt.org/

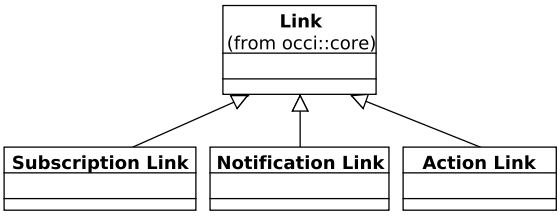


Figure 5.5: OCCI Links for Monitoring and Reconfiguration.

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Subscription
source	URI
target	URI
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#link

Attributes for the *Subscription Link*				
name	type	mut.	req.	Description
occi.subscriptionlink.name	string	yes	yes	Name of Subscription Link
occi.subscriptionlink.version	number	yes	no	Version of Subscription Link

Table 5.1: Definition of the *Subscription Link*

Action Term	Target State	Attributes
start	active	-
stop	inactive	-

Table 5.2: Actions defined for the Subscription Link

The actions applicable on instances of *Subscription Link* are depicted in the Table 5.2. The *Subscription Link* could be started by invoking the start action. It can be stopped by invoking the stop action.

5.2.2.5 Notification Link

It is a Link between a monitored Resource ((see *source* in Table 5.3)) and a subscriber (see *target* in Table 5.3). It models the activity of notifying subscribers about the state of a monitored metric. This activity depends on the type of the subscription. If the subscription is on change, a notification will occur whenever the monitored metric changes. If the subscription is on interval, a notification will occur periodically based on the period specified in the subscription. The notification aspects are described using an instance of the *NotificationTool* Mixin collection (described in Section 5.2.2.8). Eventually, a *Notification Link* is instantiated whenever a subscription occurred.

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Notification
source	URI
target	URI
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#link

Attributes for the *Subscription Link*				
name	type	mut.	req.	Description
occi.notificationlink.name	string	yes	yes	Name of Notification Link
occi.notificationlink.version	number	yes	no	Version of Notification Link

Table 5.3: Definition of the *Notification Link*

The different attributes of the Subscription Link are depicted in Table 5.3. The actions applicable on instances of *Notification Link* are depicted in the Table 5.4. The *Notification Link* could be started by invoking the start action. It could be stopped by invoking the stop action.

Action Term	Target State	Attributes
start	active	-
stop	inactive	-

Table 5.4: Actions defined for the Notification Link

5.2.2.6 Action Link

It is a Link that has a reconfiguring Resource (any OCCI Resource that needs to reconfigure the managed Resource) as source (see *target* in Table 5.5) and a managed Resource as target (see *target* in Table 5.5) on which the reconfiguration actions are applied. It models the transfer of actions from the source to be applied on the target. An *Action Link* is characterized by applying reconfiguration actions on other Resources. This is specified using a Mixin instance from the *ActionTool* Mixin collection (described later in Section 5.2.2.9).

The actions applicable on instances of *Action Link* are depicted in the Table 5.6. The *Action Link* could be started by invoking the start action. It can be stopped by invoking the stop action.

5.2.2.7 SubscriptionTool Mixin

It models a Subscription instance. It contains the different details of a subscription (i.e., subscription type, duration, and eventually filters). This Mixin may describe the mechanism of the subscription or may refer to an external program that handles this task. This Mixin implements a *SubscriptionToolInterface* that defines *subscribe()* method specifying how to pass a subscription from a subscriber to the

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Action
source	URI
target	URI
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#link

Attributes for the *Action Link*				
name	type	mut.	req.	Description
occi.actionlink.name	string	yes	yes	Name of Action Link
occi.actionlink.version	number	yes	no	Version of Action Link

Table 5.5: Definition of the *Action Link*

Action Term	Target State	Attributes
start	active	-
stop	inactive	-

Table 5.6: Actions defined for the Action Link

managed resource. For example, this method can prepare a REST query with the different details of the subscription and send it to the managed resource.

5.2.2.8 NotificationTool Mixin

It models the needed tools by which notifications are sent to subscribers. An instance of the *NotificationTool* may enable notification messages by different tools like emails, messages, or simple HTTP notifications. This Mixin may implement the needed mechanisms to send the notification or it may refer to an external program that handles this task. It implements a *NotificationToolInterface* that describes a *notify()* method specifying how to pass a notification from a managed resource to a subscriber. For example, an implementation of this method can create a REST query with the information of the notification and send it to the subscriber.

5.2.2.9 ActionTool Mixin

It models the needed mechanisms by which reconfiguration actions are applied on a specific Resource. An instance of the *ActionTool* may represent a simple *Action* as it is defined for the OCCI Core model [2]. It may also be a composed action that refers to a list of actions. This Mixin implements a *StrategySetInterface* that defines a method *applyAction(reconfigurationAction)*. An implementation of this Mixin must describe how actions are executed, they can be applied directly using the *Reconfiguration* Mixin of a given resource. They can be also a call to the IaaS or PaaS managers to add or retrieve new resources for example.

5.2.3 Example of using OCCI Monitoring and Reconfiguration extension

In real Cloud environments, there are different use cases of our resources. For example, as shown in Figure 5.6, ResourceB requires to monitor by subscription the ResourceA. To make that possible, we can extend the ResourceA to render it monitorable and reconfigurable. To this aim, we use Polling, Subscription and Reconfiguration Mixins. Then, we can subscribe the ResourceB to make it receive notifications of the status of the monitored resource. This subscription is based on the *Subscription Link* that we defined. The different aspects of the subscription could be specified using an instance of *SubscriptionTool* Mixin. At the occurrence of a subscription, a *Notification Link* instance is created to link the managed resource (i.e., ResourceA) to the monitoring resource (i.e., ResourceB). The different aspects of notification mechanism are specified using an instance of the *NotificationTool* Mixin. ResourceB processes the incoming monitoring information and can execute reconfiguration actions on the managed resource by mean of the *Action Link*. This latter is specialized using an instance of *ActionTool* Mixin.

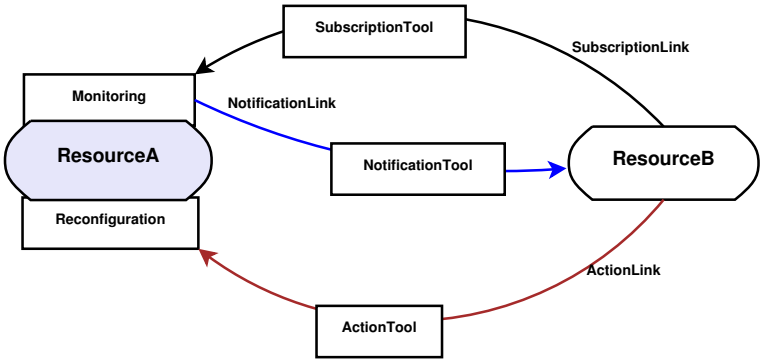


Figure 5.6: Example of using our extension.

5.3 Autonomic Management for Cloud Resources

IBM [11] defines Autonomic Computing as the ability to manage computing resources that automatically and dynamically respond to the requirements of the business based on SLA. In our work, we are interested in Cloud environments. Such environments are composed of an increasing number of heterogeneous resources. The management of these resources is becoming more and more complex. We advocate that management tasks like monitoring, configuration, protection, optimization, are not the main functional objective of most resources, but if they are not properly addressed, these resources cannot accomplish their tasks. The challenge, then, is to enable autonomic management to take control of all these non functional tasks, letting the developers focus on the main functional goals of the resource. In order to really free developers from the burden of autonomic management features on their applications, there must exists a way to dynamically add these features to Cloud resources. Autonomic management is usually presented as a Monitor, Analyze, Plane and Execute (MAPE) loop. In the following we will detail each function of this loop.

5.3.1 MAPE: Monitor, Analyze, Plan and Execute

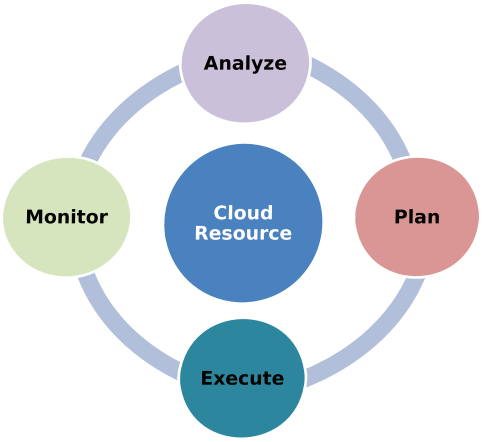


Figure 5.7: Autonomic control loop for a Cloud resource

Autonomic management features for Cloud resources are presented in Figure 5.7. In this autonomic loop, the central element represents any Cloud resource for which we want to exhibit an autonomic behavior. For this loop we need monitoring mechanisms to collect information about the managed resource. We need also configuration mechanisms to carry out changes of the managed resource. The different functions of the autonomic control loop are defined as:

1. **Monitor** function that provides the mechanisms to collect, aggregate, filter and report monitoring data collected from a managed resource;

2. **Analyze** function that provides the mechanisms that correlate and model complex situations and allow to interpret the environment, interpret the current state of the system and predict future situations;

3. **Plan** function that provides the mechanisms that construct a plan of actions needed to achieve a certain goal, usually according to some guiding strategies;

4. **Execute** function that provides the mechanisms to control the execution of the plan over the managed resources.

5.3.2 MAPE for Cloud Resources

The usage of autonomic capabilities in conjunction with Cloud computing provides an evolutionary approach in which autonomic computing capabilities anticipate runtime resource requirements and resolve problems with minimal human intervention. To add autonomic management features for Cloud resources, we propose to dynamically add the needed resources and extensions to ensure the MAPE

(Monitor, Analyze, Plan and Execute) functions. In Section 5.2, we presented how we add monitoring and reconfiguration facilities to Cloud resources. The mechanisms that we proposed are our pillar blocks to ensure *Monitor* and *Execute* functions of the MAPE loop. Consequently, we propose to dynamically transform the resource into a managed Resource (with monitoring and reconfiguration facilities).

For the *Analyze* and *Plan* functions, we propose to define two new abstract resources responsible of these functions. Later, we can customize these abstract resources according to the managed Resource. To support the *Analyze* function we define the *Analyzer* resource that allows analyzing monitoring data and eventually generating alerts whenever the SLA is not respected. Moreover, in order to support the *Plan* function, we define the *Planner* resource that receives alerts from the *Analyzer*. Based on these alerts, the *Planner* may generate reconfiguration actions to be applied on Resources. The customization of these abstract resources consists on specifying the strategies to be used by the *Analyzer* to process the incoming monitoring information. These strategies are extracted from the SLA of the managed Resource. A strategy can be an *event-condition-action* rule. For example, if the monitoring data is related to a service response time, we can specify a strategy that raises an alert if the value of the response time is over a given threshold. The customization of the *Planner* is based on specifying the actions to be used for the processing of the incoming alerts. Examples of these aspects will be detailed later in Chapter 6.

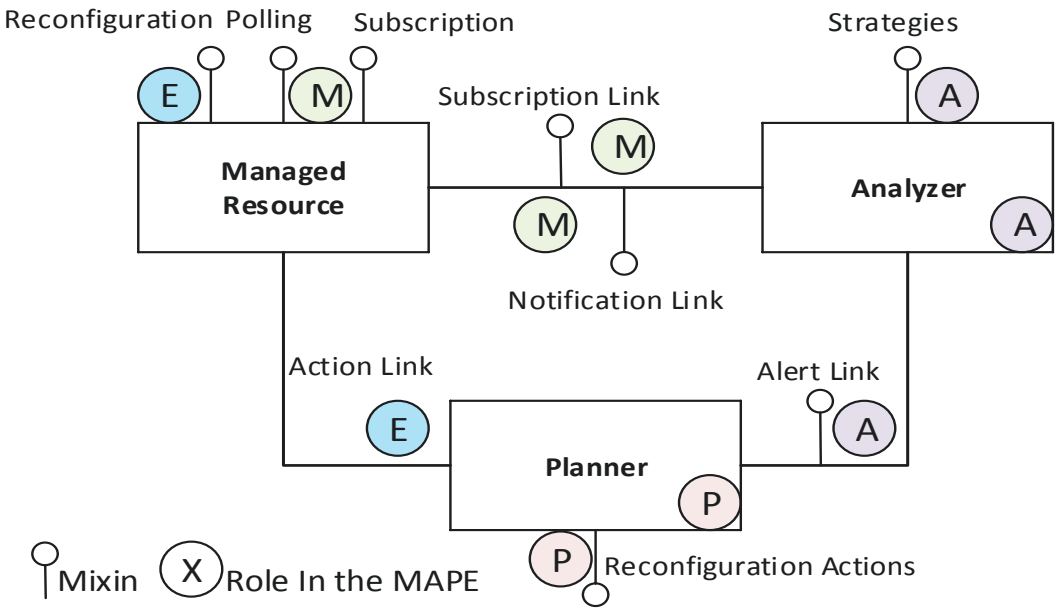


Figure 5.8: Autonomic infrastructure for a Cloud resource

The previously described extensions and resources are our basics in order to add autonomic management for Cloud resources. To enable the holistic autonomic loop, we need to link these elements. On

one hand, the *Analyzer* needs to be linked to the managed Resource in order to subscribe for monitoring data and to receive notifications. On the other hand, it needs to be linked to the *Planner* to send alerts. In Figure 5.8, we show how we extend a Cloud resource to render it manageable. We use also the following links that could be customized according to the use case (see Chapter 6 for an example of customization according to elasticity property of applications and Service-based Business Processes): (1) Subscription link that enables the subscription of an *Analyzer* to a managed Resource; (2) Notification link that delivers notifications from the managed Resource to the *Analyzer*; (3) Alert link that delivers alerts from the *Analyzer* to the Planner; and (4) Action link that executes reconfiguration actions on the managed Resource or on other resources related to this managed Resource. As shown in Figure 5.8, our extension covers the different functions of the MAPE loop to ensure autonomic management for Cloud resources.

For our solution to be widely adopted among Cloud providers, we need to build it upon a standard. The *de facto* standard for description of Cloud resources is the Open Cloud Computing Interface (OCCI). Consequently, we will use OCCI to describe the different elements for autonomic management infrastructure.

5.3.3 OCCI extension for Autonomic Management

To provide a generic description for autonomic computing using OCCI, we defined new Resources, Links and Mixins. As shown in Figure 5.9, the resources are the *Autonomic Manager*, the *Analyzer* and the *Planner*. These Resources inherits the Resource base type defined in OCCI core [2]. The Links that we added including those described in Section 5.2 are *Agreement Link*, *Subscription Link*, *Notification Link*, *Alert Link* and *Action Link*. These Links inherit the Link base type defined for OCCI core. Finally, we defined different Mixins inheriting the Mixin base type defined in OCCI core.

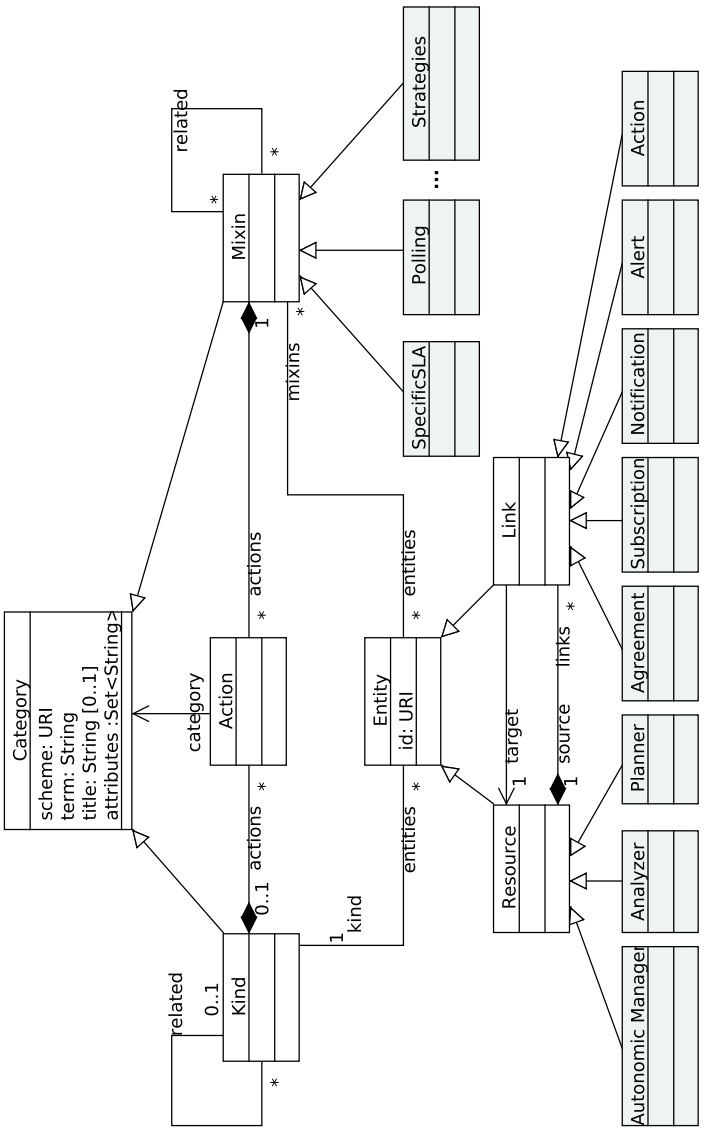


Figure 5.9: OCCI Core Model and our extension (colored boxes) for Autonomic Computing.

5.3.3.1 Autonomic Manager Resource

In order to automatize the establishment of the autonomic infrastructure, we defined an *Autonomic Manager* Resource as a generic OCCI Resource. It inspects a given SLA and carries out the list of actions to build the autonomic computing infrastructure. From a given SLA, this Resource determines monitoring targets (i.e., the needed metrics to be monitored). It is also responsible of extracting the strategies to be used by the *Analyzer* Resource (see Section 5.3.3.2) and the reconfiguration plans to be used by the *Planner* (see Section 5.3.3.3). After inspecting the contract (SLA), the Autonomic Manager instantiates the needed Entities (i.e., Resources and Links). Then, it customizes these Entities with the needed Mixins and eventually configure them with the needed parameters. Since the SLA can be described using different specifications (i.e., WS-Agreements, USDL, etc), the *Autonomic Manager* uses a *SpecificSLA* Mixin (detailed in Section 5.3.3.6) that describes how to deal with a specific SLA.

Table 5.7: Definition of the Autonomic Manager

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Autonomic Manager
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#resource

Attributes for the *Autonomic Manager*				
name	type	mut.	req.	Description
name	String	yes	yes	Autonomic Manager name
version	String	yes	no	Autonomic Manager Version
SLALocation	String	yes	no	SLA Location

As shown in Table 5.7, we specify the name of the Autonomic Manager to instantiate and its version. Moreover, we specify the location of the SLA. This SLA will be used to provision the Resource and to establish the rest of the autonomic infrastructure. The last attribute is not mandatory because we can get the SLA directly if we have an instance of the *Agreement Link* (see Section 5.3.3.4).

The actions applicable on instances of *Autonomic Manager* resource are depicted in Table 5.8. The *Autonomic Manager* resource could be started/stopped by invoking the start/stop action. And it can be reconfigured by invoking the reconfigure action passing as parameter a new SLA. It is important to note that reconfiguring the *Autonomic Manager* may result on changing the Mixins used to process SLA or to reconfigure other Mixins used in the autonomic infrastructure.

Action Term	Target State	Attributes
start	active	-
stop	inactive	-
reconfigure	reconfiguring	sla

Table 5.8: Actions defined for the Autonomic Manager resource

5.3.3.2 Analyzer

It is a generic OCCI Resource that allows to analyze monitoring data and eventually to generate alerts. Notifications are received through an instance of the *Notification Link* (described in Section 5.2). The *Analyzer* is an abstract resource that could be specified using *Strategies* Mixin collection (described later). It uses *Strategies* Mixin to specify rules to apply on incoming monitoring data. The definition of this Resource is depicted in Table 5.9.

Table 5.9: Definition of the Analyzer

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Analyzer
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#resource

Attributes for the *Analyzer*				
name	type	mut.	req.	Description
name	String	yes	yes	Analyzer name
version	String	yes	no	Analyzer version

As shown in Table 5.9, in order to provision an *Analyzer* Resource, we need to specify its name and version.The actions applicable on instances of *Analyzer* resource are depicted in Table 5.10.

Action Term	Target State	Attributes
start	active	-
stop	inactive	-
reconfigure	reconfiguring	Strategies

Table 5.10: Actions defined for the Analyzer resource

The *Analyzer* resource could be started/stopped by invoking the start/stop action. It can be reconfigured by invoking the reconfigure action passing as parameter the new reconfiguration (reference to new Strategies Mix-in).

5.3.3.3 Planner

It is a generic OCCI Resource that receives alerts from the *Analyzer* through an *Alert Link* (described later). To receive alerts, this Resource must be the target of one or more *Alert Link* instances. The *Planner* is an abstract resource that could be specialized using *Reconfiguration Actions* Mixin collection (described in Section 5.3.3.9). This latter is used to specify reconfiguration plans (list of actions) to apply for each received alert. Based on these alerts the *Planner* may generate reconfiguration actions to be applied on Resources. The definition of the *Planner* is depicted in Table 5.11. In order to instantiate a *Planner* Resource, we need to specify its name and version.

Table 5.11: Definition of the Planner

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Planner
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#resource

Attributes for the *Planner*				
name	type	mut.	req.	Description
name	String	yes	yes	Planner name
version	String	yes	no	Planner version

The actions applicable on instances of *Planner* resource are depicted in Table 5.12. The *Planner* resource could be started/stopped by invoking the start/stop action. It could be reconfigured by invoking the reconfigure action passing as parameter the new reconfiguration (i.e., new Plans).

Action Term	Target State	Attributes
start	active	-
stop	inactive	-
reconfigure	reconfiguring	Plans

Table 5.12: Actions defined for the Planner resource

5.3.3.4 Agreement Link

It is a Link between an *Autonomic Manager* (source in Table 5.13) and a managed Resource (target in Table 5.13). It allows an *Autonomic Manager* to inspect the SLA of the managed Resource. It is an abstract link that could be specified using an instance of *SpecificSLA* Mixin describing how to process a specific kind of SLAs (e.g., WS-Agreement, USDL).

The actions applicable on instances of *Agreement Link* are depicted in the Table 5.14. The *Agreement Link* could be started/stopped by invoking the start/stop action.

5.3.3.5 Alert Link

It is a Link between the *Analyzer* Resource (source in Table 5.15) and the *Planner* Resource (target in Table 5.15). Its role is to drive alerts from the *Analyzer* to the *Planner*. An alert is generated when the *Analyzer* detects that the SLA is not respected. An alert is specified by an instance of the *AlertTool* Mixin collection (described later).

As shown in Table 5.15, in order to instantiate and *Alert Link*, one must specify its name and version as well as the start time and duration of the establishment of this link. The actions applicable on instances of *Alert Link* are depicted in the Table 5.16. The *Alert Link* could be started by invoking the start action. It can be stopped by invoking the stop action.

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Agreement
source	URI
target	URI
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#link

Attributes for the *Agreement Link*				
name	type	mut.	req.	Description
occi.agreementlink.name	string	yes	yes	Name of Agreement Link
occi.agreementlink.version	number	yes	no	Version of Agreement Link

Table 5.13: Definition of the *Agreement Link*

Action Term	Target State	Attributes
start	active	-
stop	inactive	-

Table 5.14: Actions defined for the Agreement Link

To customize the different Resources and Links, we defined different Mixins. These latter inherits the Mixin base type defined in OCCI Core [2] (see Figure 5.9).

5.3.3.6 SpecificSLA Mixin

In Cloud Computing, there are different languages to describe SLA. To tackle this heterogeneity we defined this Mixin that describes the needed tools allowing an *Autonomic Manager* to extract the needed information from a specific SLA. This Mixin allows the *Autonomic Manager* to get the metrics that need to be monitored, the analysis rules and the reconfiguration strategies. Based on these metrics, the *Autonomic Manager* instantiates the needed Resources to establish the autonomic loop and configures them. Technically, a Mixin instance of SpecificSLA implements the interface *SpecificSLAInterface*. This interface defines an abstract method called *inspectSLA(File SLA)*. For each type of SLAs we can implement a Mixin that describes exactly how to process SLAs. An example is a WS-Agreement Mixin using a WSAgreementParser that we developed to inspect any SLA based on WS-Agreement. At this stage of the research, we don't tackle the semantics of the described SLA. The *SpecificSLA* Mixin, uses a simple XML file that syntactically matches the described information in the SLA with the existing Mixins. However, in our future work we need to extend this aspect to take into account semantic interpretation of SLA information.

5.3.3.7 Strategies Mixin

It represent a computation strategy that based on incoming monitoring information triggers specific alerts. It represents a function applied by the *Analyzer* to compare monitoring information values against

Model attribute	value
scheme	http://ogf.schemas.sla/occi/autonomic#
term	Alert
source	URI
target	URI
attributes	(see below)
related	http://ogf.schemas.sla/occi/core#link

Attributes for the *Alert Link*				
name	type	mut.	req.	Description
occi.alert.name	string	yes	yes	Name of Alert Link
occi.alert.version	number	yes	yes	Version of Alert Link
occi.alert.starttime	number	yes	yes	Start time of Alert Link
occi.alert.duration	string	yes	no	Duration of Alert Link

Table 5.15: Definition of the *Alert Link*

Action Term	Target State	Attributes
start	active	-
stop	inactive	-

Table 5.16: Actions defined for the *Alert Link*

previously defined thresholds. Whenever a condition is not respected, the Mixin instance makes the *Analyzer* trigger a specific alert. This Mixin implements the *StrategiesInterface* that defines a method *analyze(notification)*. An implementation of this method describes how the analyzer process the incoming notifications. An example of this Mixin can use the Event-Condition-Action model.

5.3.3.8 AlertTool Mixin

It models the needed tools by which alerts can reach a *Reconfiguration Manager*. This Mixin implements an *AlertToolInterface* that defines a method *alert()*. An implementation of this method can entail sending a simple message containing the description of the encountered violation or a creation of an alert and sending it to a given interface. It may also refer to an external program that handles the alerting.

5.3.3.9 Reconfiguration Actions Mixin

Each instance of this Mixin implements a computation plan, that based on incoming alerts triggers reconfiguration actions on specific Resources. It represents a function applied by the *Reconfiguration Manager* to process incoming alerts. This Mixin implements the *ReconfigurationActionsInterface* that defines a method *generateReconfigurationActions(alert)*. An implementation of this method describes how to process the incoming alert in order to generate reconfiguration actions. It can also refer to an external program that handles planning for reconfigurations. For example, this Mixin can make a direct matching between the alert and existing actions: if the incoming alert is a *scaleUpAlert*, this Mixin

generates a *scaleUp* action.

5.4 Autonomic Infrastructure for Cloud Resources

In this section, we detail the usage of the previously defined OCCI Entities (i.e., Resources and Links) and Mixins to establish an autonomic computing infrastructure (see Figure 5.10).

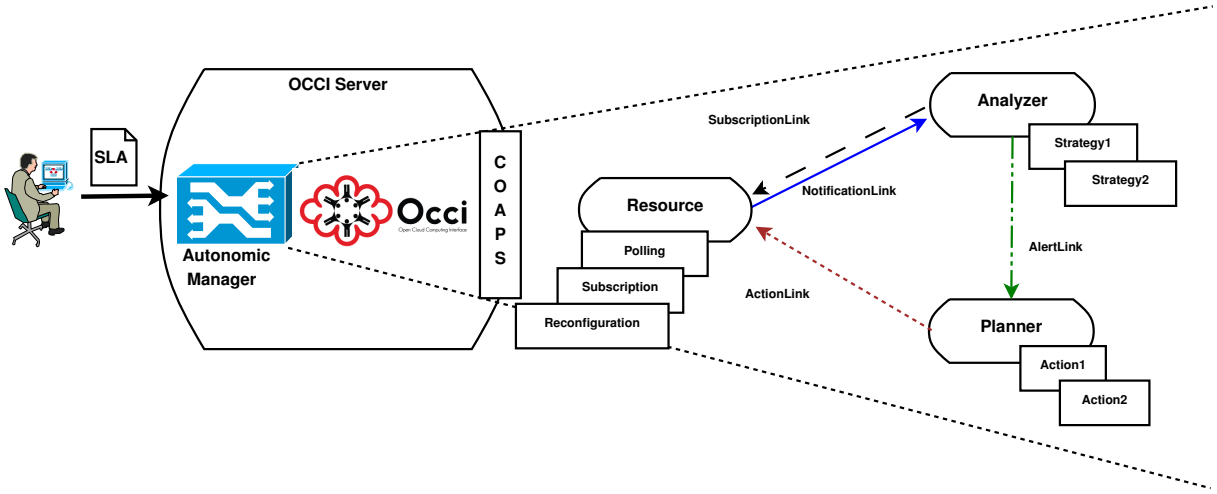


Figure 5.10: Autonomic Computing Infrastructure establishment for Cloud resources.

To establish our autonomic computing infrastructure, we start by setting up our OCCI Server. This server is responsible of instantiating any OCCI Entity. The first Resource instantiated in this server is the *Autonomic Manager* Resource. In our proposal, the *Autonomic Manager* plays an important role. It is responsible of processing the SLA and extracting all the needed information to establish and customize the needed Resources and Links. Using this information, it sends requests to the OCCI Server in order to build the autonomic infrastructure as shown in Figure 5.10. When the *Autonomic Manager* is activated, it is customized using an instance of the *SpecificSLA* Mixin to inspect each SLA using a different description language. Based on the SLA, the *Autonomic Manager* is able to detect the attributes or services that need to be monitored for the different resources. Consequently, it is able to extend the resources by a *Polling* Mixin to allow the retrieval of monitoring data or by a *Subscription* Mixin to allow subscriptions and thus notifications. It can also extend them using *Reconfiguration* Mixins to enable reconfigurations. The *Autonomic Manager* sends a request to the OCCI Server to instantiate the Mixed OCCI Resources (i.e., the Resources with newly added Mixins). Whenever this Resource is ready, the *Autonomic Manager* orders the OCCI Server to deploy and start them.

The next step realized by the *Autonomic Manager* is to carry out a series of queries addressed to the OCCI Server to instantiate and customize the needed Resources and Links. It starts by instantiating the *Analyzer* that consumes monitoring data and applies analysis strategies. To receive monitoring notifications, the *Analyzer* must be subscribed to the monitored Resource using a *Subscription Link* having as source the *Analyzer* itself and as target the Resource to be monitored. The *Subscription Link* being

abstract, needs to be specified using a *SubscriptionTool* Mixin. This latter contains the specific aspects of the subscription. An instance of this Mixin can specify the start time of the subscription, its duration and its type whether it is on interval or on change. These information are extracted from the SLA. Based on the type of the needed subscription, the *Analyzer* may receive periodic notifications if the subscription is on interval, and it may receive notifications whenever the monitored metric changes if the monitoring is on change. Notifications are received through an instance of the *Notification Link*. Technically, the aspects related to notifications are described using instances of the *NotificationTool* Mixin. Based on the content of the SLA, this Mixin may specify exactly how the notifications can reach the subscriber (i.e., the notification can be an email, a message or a HTTP callback, etc). At the reception of a notification, the *Analyzer* uses *Strategies* Mixin instances that specify analysis strategies to be applied on incoming monitoring data. These Mixin instances can be simple conditions comparing the data against previously defined values or they can be calls to an external program that applies some analysis. In the two cases, if the SLA is violated, the *Analyzer* may raise alerts to the *Planner*. Accordingly, the *Autonomic Manager* asks the OCCI Server to instantiate the *Planner* Resource and to link it to the *Analyzer* via an *Alert* Link. This latter specifies how the alerts are sent to the *Planner* using an instance of the *AlertTool* Mixin. The *Planner* uses specific reconfiguration plans to generate reconfiguration actions. These plans are instances of the *Reconfiguration Actions* Mixin. This latter Mixin can refer to a simple matching between incoming alerts and outgoing actions. It can be also a call to external program able to handle alerts and generate reconfiguration actions. The information about the used *Reconfiguration Actions* are extracted by the *Autonomic Manager* from the SLA.

The last step is to link the *Planner* to the managed Resource using an *Action Link* that uses the generated reconfiguration actions and applies them on the Resource. We specify how to apply these actions using an *ActionTool* Mixin. The generated actions could concern the managed resource as well as other resources.

We precise that an instance of the *Analyzer* Resource can subscribe to different Resources to receive monitoring data. Moreover, the *Planner* can be linked to one or more Resources to apply reconfiguration actions. Consequently, we can have one autonomic loop for multiple resources or one loop per resource. In this work, we don't treat the problem where different autonomic loops collaborate together to a global objective. We would rather include this problem in our future work.

5.5 Adding FCAPS Management to Cloud Resources

In order to show the usefulness of our autonomic mechanisms, we describe in the following how we can use our autonomic infrastructure to enable Fault, Configuration, Accounting, Performance and Security management services for Cloud resources. For each one of the FCAPS services, we can implement the associated Mixins in order to specify the autonomic infrastructure according to the required service.

Fault In order to ensure Fault management, we can implement monitoring Mixins responsible of gathering data related to the availability of the managed resource. For example, if the managed resource is a *Compute* instance provided by an OpenNebula IaaS manager, we can imagine that monitoring Mixin retrieves the state of resource from the IaaS manager through a REST query. It sends then, notifications to

the Analyzer containing a description of the state of the managed resource. The *Analyzer* uses its strategies that can be a simple comparison between the actual state value of the resource and the unavailability states values (e.g., *failed*, *pending* or *unknown*). If the actual state matches one of these unavailability states, the *Analyzer* raises an alert to the *Planner*. This latter uses its *Reconfiguration Actions* Mixin to generate reconfiguration actions. These actions could be a REST query sent to the IaaS manager to redeploy the *Compute* or to deploy a new instance having the same characteristics.

Configuration This non functional service is enabled using the *Reconfiguration* Mixin defined in Section 5.2.2.1. The implementation of this Mixin allows the different reconfiguration types (previously described in Chapter 2). To define new configuration behaviors, one should implement this Mixin respecting the implementation details described in Section 6.4.2.

Accounting Accounting remains an important service for both Cloud providers and consumers. Using our mechanisms can easily enable this service by implementing the desired manner to monitor resources usage. For example, we can implement a Mixin that monitors the memory consumption of a given resource. Using monitoring by subscription on interval, the *Analyzer* can receive periodic notifications of the memory consumption. Then, accounting strategies could be implemented and associated to the *Analyzer* to process monitoring notifications and to send alerts to the *Planner*. For this latter we can implement the desired billing models as *Reconfiguration Actions* Mixins allowing the calculation of the bills and describing how exactly to make them available to providers and consumers.

Performance Autonomic facilities are often used to control the performance of resources and whether they are performing their tasks. For example, the performance of a web-service could be calculated based on its response time. Accordingly, we can implement monitoring Mixins that allow to retrieve the response time of the service and send it to the *Analyzer*. This latter can use specific strategies to compare the response time received in the notifications against given thresholds. If a violation occurs, it can alert the *Planner* who is responsible of generating reconfiguration actions. We can implement reconfiguration actions for this kind of situation to enable the duplication or migration of the service in order to enhance its performances.

Security In order to add security services based on our mechanisms, we can implement monitoring Mixins to detect security breaches. These Mixins can notify the *Analyzer* on the occurrence of violations. This latter can use its strategies to verify if it needs to alert the *Planner*. In which case, this latter generates reconfiguration actions in order to face the violation. We can implement reconfiguration actions to rapidly block unauthorized access for example.

5.6 Conclusion

Monitoring and Reconfiguration of Cloud resources are critical to ensure a good QoS for Cloud resources. In this chapter, we proposed a level-agnostic approach to add monitoring and reconfiguration

facilities to Cloud resources. Our proposal is a generic model that allows to describe these non functional facilities. We used Open Cloud computing Interface (OCCI) which is the *de facto* standard for the Cloud to describe our model. We defined new OCCI Resources, Links and Mixins for monitoring and reconfiguration. Then we used this model as pillar blocks to build a holistic autonomic infrastructure for Cloud resources. We also described how one can use our model to establish an autonomic infrastructure for any Cloud resource.

In the next chapter, we will validate our contributions. The validation includes different use cases for monitoring, reconfiguration and autonomic management of SCA-based applications as well as Cloud resources. We performed different experiments to test our proposals in real Cloud environments.

Chapter 6

Evaluation and Validation

Contents

6.1 Introduction 81
6.2 Common Environment description 82
6.3 Evaluation of Monitoring and Reconfiguration Approach of SCA-based applications in the Cloud 82
6.3.1 Implementation 82
6.3.2 Evaluation 83
6.4 Evaluation of Autonomic Computing Approach for Cloud Resources 85
6.4.1 Background 85
6.4.2 Implementation 94
6.4.3 Evaluations 97
6.5 Conclusion 105

6.1 Introduction

Cloud Computing can significantly reduce investment and operating costs in IT field. However any proposed solution for this paradigm needs to be validated to ensure its contribution to enhance Cloud services behaviors. In our work, we proposed novel approaches to add monitoring, reconfiguration and autonomic management facilities to SCA-based applications as well as Cloud resources in general. In this chapter, we will evaluate our approach. The evaluation includes the implementation aspects as well as the experiments details and their results. For this validation, we used a real Cloud environment. This environment is common for all our evaluations. We will start with the description of the experiments environment. Then, we will detail the implementation and experiments of our approach of adding management facilities to SCA-based applications. Afterwards, we will detail the implementation and experiments of our approach of adding management facilities to Cloud resources. This last part contains two detailed use cases proving the efficiency of our approach to add autonomic management facilities to Cloud resources.

6.2 Common Environment description

To perform our evaluations we used the NCF (Network and Cloud Federation) experimental platform deployed at Telecom SudParis France. The NCF experimental platform aims at merging networks and Cloud concepts, technologies and architectures into one common system. NCF users can acquire virtual resources to deploy and validate their own solutions and architectures. The hardware component of the network is in constant evolution and has for information: 380 Cores Intel Xeon Nehalem, 1.17 TB RAM and 100 TB as shared storage. Two Cloud managers allow managing this infrastructure and virtual resources i.e., OpenNebula [69] and OpenStack [70]. In our work, we used OpenNebula which is a virtual infrastructure engine that provides the needed functionality to deploy and manage virtual machines (VMs) on a pool of distributed physical resources. To create a VM, we can use one of the three predefined templates offered by OpenNebula i.e. SMALL, MEDIUM and LARGE, or we can specify a new template. During our experiments, we used our specific template with the following characteristics: 4 cores (2.66 GHZ each core) and 4 Gigabytes of RAM.

6.3 Evaluation of Monitoring and Reconfiguration Approach of SCA-based applications in the Cloud

In our work, we proposed a novel approach to add monitoring, reconfiguration and FCAPS management facilities to SCA-based applications prior deploying them in the Cloud. In order to validate our approach and show its efficiency, we dedicate this section to the evaluation of this proposal. We start by detailing the implementation aspects of our work. Then we present the experiments scenarios and their results.

6.3.1 Implementation

The implementation process took place in different phases. We have first developed a minimal Java deployment framework, which allows developers to deploy a Java service on a hard-coded micro-container before deploying both of them in the cloud. To enhance the performance of the platform and facilitates updates and future changes, the deployment framework is made modular to allow us to plug or unplug specific modules. The generation process is based primarily on the parsing of the extended SCA composite.

The next phase was implementing a prototype of the monitoring framework as services that offer the transformation mechanisms to the components. The byte code of the managed component is transformed to add notification code. For this required byte-code level manipulation we used the Java reflection API and the open source software JAVA programming ASSISTant (Javassist) library [71]. The Java reflection API [72] provides classes and interfaces for obtaining reflective information about classes and objects. Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use of reflected fields, methods, and constructors to operate on their underlying counterparts objects. Javassist is a class library for editing Java byte-codes; it enables Java programs to define a new class and to modify a class file when the Java Virtual Machine (JVM) loads it.

For the Mobility service, we used the Java Serializable API for serialization and deserialization. We used SSH protocol to transfer the serialized code of the micro-containers. If the deployed service is a

stateful service, the state is saved in a specific file that will be serialized with the micro-container. After running this latter in the new VM, it reloads its state from the saved file.

Finally, we implemented Java classes for the FCAPS management. Each class is a thread that uses a list of rules extracted from the extended SCA composite to ensure a specific task. The Fault, Accounting and Performance tasks are clients for the monitoring service. They have the ability to consume notifications to take specific decisions.

6.3.2 Evaluation

In our work, we proposed a platform able to deploy components in the Cloud on top of light weight micro-containers, with the capability of transforming components to be self managed even if they were not designed with management facilities. The monitoring mechanism that we use is flexible in the way of choosing the best deployment scenario to meet the deployer requirements. As far as we know, almost all of the existing monitoring solutions in Cloud environments use one channel to deliver monitoring information, but in our approach we exhibit the possibility of using a channel (i.e., Notification Service) at the granularity of a component. In our experiments, we compare the results obtained using one *Notification Service* for all publishers and using one *Notification Service* per publisher. To this aim, we have considered two criteria: 1) Memory consumption: Memory size consumed by the micro-container with or without management facilities, and 2) Notification Latency Time: The elapsed time between the occurrence of the event and the notification reception from all subscribers.

Use Case description:

To perform our tests, we defined two scenarios that reflect the objectives we want to highlight in our experiments. In these experiments we want to: 1) Compare micro-container memory consumption before and after adding management facilities to estimate the overhead of the management modules on the micro-container consumption, and 2) Compare notification latency time in the micro-container using monitoring system with one *Notification Service* or with multi-*Notification Service* (i.e., one *Notification Service* per micro-container).
In the different series of tests, we deployed different numbers of services on top of micro-containers. The used service in these experiments is a service that has a changing property. Whenever this property changes the service sends a notification to its channel (Notification Service component). This latter pushes this notification to all the subscribers.

In the first series we deployed services on micro-containers without management capabilities and we took memory consumption measurements for each number. Then, we deployed the same number of services on top of the micro-container enhanced with management facilities. The purpose of these experiments was to estimate the overhead of the management modules on the memory consumption of the micro-container. Figure 6.1 shows the different values stored during these series including the JVM size.

These experiments show that the overhead of the management module on the memory consumption of the micro-container is fair. In fact, the memory consumption is linear, increasing with the number of deployed services. The results show that the overhead of the memory consumption using multi *Notification Service* is more important than the overhead using one *Notification Service* scenario, and this is due

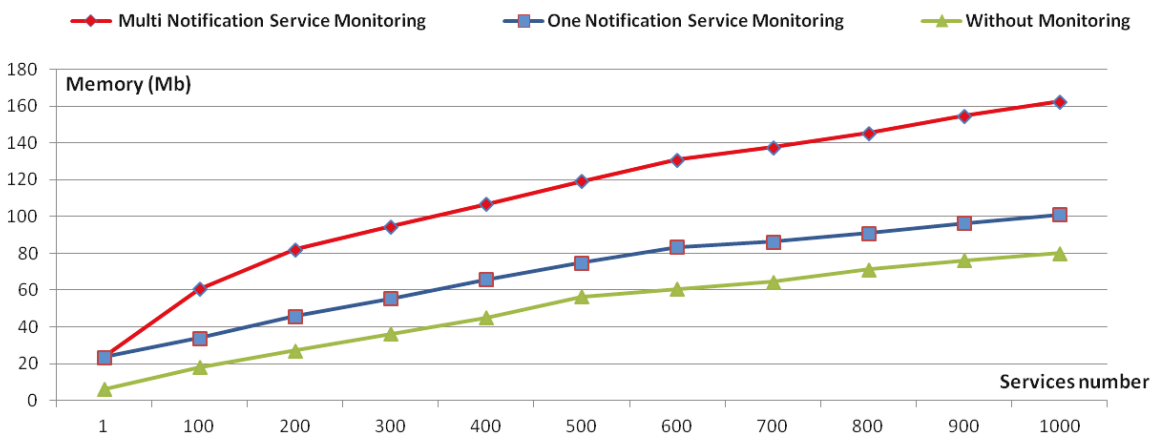


Figure 6.1: Memory consumption of different types of micro-containers.

to adding an instance of the *Notification Service* component to each micro-container. That adds the extra memory consumption noticed in Figure 6.1.

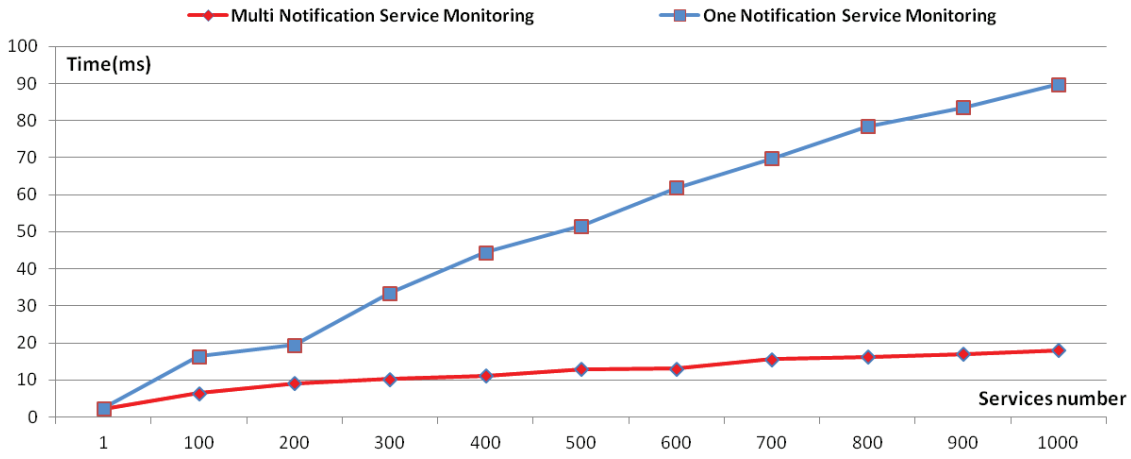


Figure 6.2: Latency time using different scenarios of monitoring.

In the second series of tests, we aimed to compare the notification latency time using the micro-container enhanced with monitoring mechanisms in the two cases: using one *Notification Service* and multi *Notification Service* monitoring. In each series, we fixed the number of deployed services and we changed the frequency of events occurrence. After storing these measurements, we calculate the average latency time for each series. The stored values are shown in Figure 6.2. Specifically, these values represent the needed time from the event's raise till the reception of the notification by all subscribers.

When the number of events becomes important, the *Notification Service* is exposed to a large number of notifications, since all notifications are targeting the same *Notification Service*. This latter should forward each notification to the list of the interested subscribers. When using multi *Notification Services*, every micro-container contains its own *Notification Service* component. Consequently, it is asked to manage just its own notifications. It will deal with a less number of notifications and subscribers. That explains the results shown in Figure 6.2 where the notifications' latency time is almost the same using the multi *Notification Service* monitoring system and it is increasing proportionally with the number of services when we use micro-containers with one *Notification Service* monitoring system.

6.4 Evaluation of Autonomic Computing Approach for Cloud Resources

In Chapter 5, we proposed a generic approach to add monitoring, reconfiguration and autonomic management facilities to Cloud resources independently of their service level. In this section, we advocate the genericity and efficiency of our proposal. To do so, we will apply our autonomic infrastructure on two use cases. The first allows to provision autonomic applications in the Cloud. The second enables provisioning elastic service-based business processes (SBPs). We will start by presenting a background of the used models to represent Platforms and Applications. Afterwards, we will chain up with the description of the SBP model that we used as well as the added elasticity mechanisms. Afterwards, we will detail the different aspects of our implementation. Then, we will describe the two use cases evaluating our work. We will also present the experiments that we performed for each use case. Finally, we will discuss the results of these experiments.

6.4.1 Background

In order to facilitate the understanding of our evaluation, we will start by introducing the different concepts used in our use cases. The first use case consists on provisioning autonomic applications in the PaaS based on OCCI extensions for Platforms and Applications. Accordingly, we will start by describing these models used to describe applications and their requirements of platform resources in Section 6.4.1.1. The second use case consists on provisioning elastic SBPs in the Cloud. To this aim, we used Petri nets to model SBPs and to add elasticity mechanisms. Accordingly, we will detail the SBP model and elasticity mechanisms in Section 6.4.1.2.

6.4.1.1 Application provisioning in the Cloud

In this section, we present and comment the PaaS and Application resources description models that we use to provision applications in the PaaS. This model extends the OCCI core model and it consists of types:

- Platform extension which describes the platform components (e.g., service containers, database instance, etc.) needed to made up the needed environment to host an application.
- Application extension which describes an application (i.e., any computer software or program) that can be hosted and executed in a PaaS and its related hosting environment,

Platform Extension:

Platform types model all resources that can be provisioned by a PaaS provider to make up an application hosting environment. Defined resources and the links between them are respectively derived from *Resource* and *Link* entities of the OCCI core model (see Figure 6.3). The main defined OCCI platform resources are *Database*, *Container* and *Router*. Over and above we defined a set of links to connect and interact with these resources. These Links are *ContainerLink*, *RouterLink* and *DatabaseLink*.

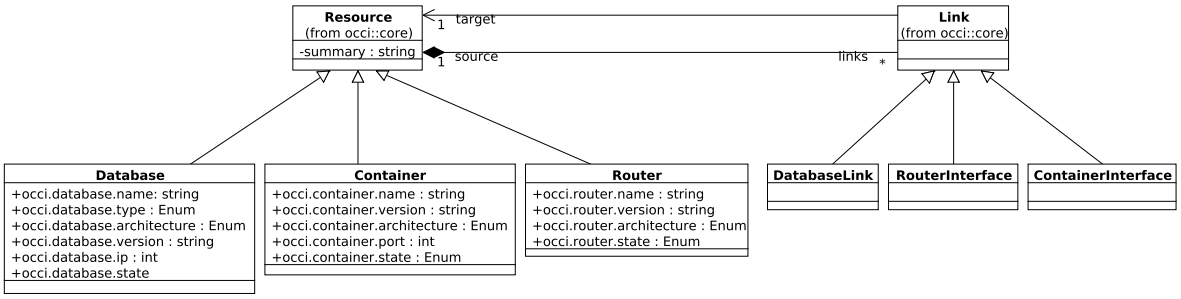


Figure 6.3: Overview of the defined OCCI platform types.

- *Database* resource type represents storage resources which can be provisioned by a PaaS provider for applications which process persistent Data (e.g., MySQL, Apache CouchDB, etc.).
- *Container* resource type represents service containers, application servers and engines provisioned by PaaS providers to host and run applications (e.g., Apache Axis, Oracle GlassFish Server, etc.).
- *Router* resource type models message format transformation and routing systems provided by PaaS providers to route and deliver messages between *Container* resources. *Router* entities are useful where deployed applications on Cloud platforms are multi-tenant and/or service-based and requires then several (may be heterogeneous) containers.
- *DatabaseLink* represents a binding between a *Container* and a *Database* resources. This enables a Database instance to be attached to a Container instance.
- *RouterLink* enables to connect one or several *Container* resources to a *Router* resource.
- *ContainerLink* allows to connect one or several *Container* resources between them.

More informations and details about defined attributes, actions and related state diagram of each resources of our PaaS resources description model can be found at [73] [74] [75].

Application Extension:

Application types extend OCCI core entities. The defined resources and the links between them are respectively derived from *Resource* and *Link* entities of the OCCI core model (see Figure 6.4). The defined OCCI application entities are *Application*, *Environment*, *Deployable* and *EnvironmentLink*.

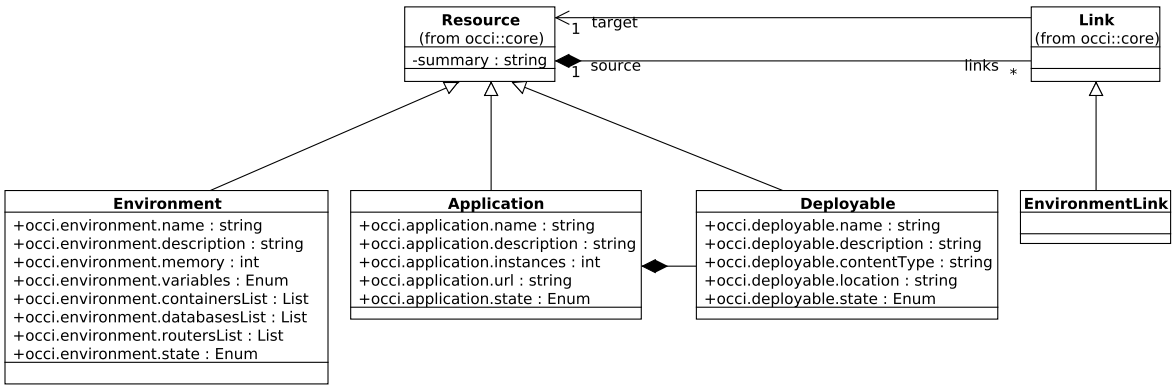


Figure 6.4: Overview of the defined OCCI application types.

- *Environment* resource models a set of components and services of the target Cloud platform. These resources are needed to host and run applications on PaaS. *Environment* resource includes, among others, the needed containers (e.g., Spring, Tomcat, Ruby, etc.), databases (MySQL, Redis, MangoDB) and optionally needed routers.
- *Application* resource models any computer software or program that can be deployed on top of a PaaS into a hosting *Environment*. This *Environment* is set up thanks to the instantiation and the configuration of necessary platform resources (see Section 6.4.1.1). To deploy an application, the end-user specifies a set of properties (e.g. application name, application description, etc.). Moreover, if the target PaaS provider supports the management of multiple instances (e.g., Cloud Foundry), the user can specify the desired number of active instances to ensure application scalability and availability. Once deployed, the *Application* can be invoked and executed through its public URL provided by the target PaaS.
- *Deployable* resource models the *Application* archives. By deployables, we mean all necessary artifacts (e.g., ZIP file, EAR file, etc.), configuration files and/or deployment descriptors (e.g., scripts, XML file, DAT file, etc.) needed to carry out the application deployment.
- *EnvironmentLink* is used to link an application to an environment. It models the deployment mechanism of an application on a given environment.

COAPS API:

We developed an OCCI-compliant API called COAPS to handle Application and Environment resources defined in our description model. Environment resources are in turn composed of platform resources (e.g. Container, Router, DatabaseLink, etc.). An early implementation of COAPS which is not OCCI-compliant was published in [76], [77].

COAPS API is based on both OCCI HTTP Rendering [27] and the Representational State Transfer (REST) architecture [78]. OCCI HTTP Rendering defines how to interact with an OCCI model using

a RESTful API. COAPS exposes a set of generic and RESTful HTTP operations (i.e., GET, POST, PUT and DELETE) for Cloud applications management and provisioning. Concretely, these operations implement actions that can be applied to platform and application resources.

To define a new COAPS implementation in order to provision resources from a given PaaS provider, one can simply implement COAPS generic interfaces according to the proprietary actions exposed by the target PaaS provider API. Several COAPS interfaces can be coupled to a single PaaS operation if needed (e.g., POST Deployables and POST Application operations for Cloud Foundry implementation) and/or some COAPS operations can be ignored (and then not implemented) when they are not supported by a specific PaaS provider (e.g., POST RouterLink operation for Cloud Foundry implementation insofar as Cloud Foundry manages itself the routing between its components).

Our provisioning mechanisms were adopted by the French FUI CompatibleOne [2] and the ITEA Easi-Clouds project [3]. It allows CompatibleOne Cloud broker to provision PaaS resources from existing PaaS providers. In Easi-Clouds project, partners have developed a collaborative IDE for the Cloud called CoRED [79]. It uses COAPS-Cloud Foundry implementation to deploy their developed applications on Cloud Foundry PaaS. In the near future, they plan to add a new implementation to deploy a part of their applications in AppScale PaaS. In addition, other partners added a GAE-PaaS implementation to deploy their services on Google App Engine PaaS [80].

6.4.1.2 Elasticity of Service-based business processes in the Cloud

Our focus in this section is on the Service-based Business Processes (SBPs) level and more particularly on SBPs elasticity. Cloud environments are being increasingly used for deploying and executing business processes and particularly SBPs. A SBP is a business process that consists in assembling a set of elementary IT-enabled services. These services realize the business activities of the considered SBP. Assembling services in a SBP can be ensured using any appropriate service composition specifications (e.g., BPEL). One of the expected facilities of Cloud environments is elasticity at different levels. The principle of elasticity is to ensure the provisioning of necessary and sufficient resources such that a Cloud service continues running smoothly even as the number or quantity of its use scales up or down, thereby avoiding under-utilization and over-utilization of resources [81]. Since the elasticity of platforms, process engines and service containers is not sufficient to ensure the elasticity of the deployed business processes[82]. We advocate that these latter should be provided with their own elasticity mechanisms to allow them to adapt to the workload changes. The main challenges in this direction are to define the needed mechanisms to perform elasticity of SBPs, and a way to couple these mechanisms with elasticity strategies in order to ensure SBPs elasticity.

Elasticity of a SBP is the ability to duplicate or consolidate as many instances of the process or some of its services as needed to handle the dynamics of the received requests. We believe that handling elasticity does not only operate at the process level but it should operate at the level of services too. In fact, it is not always necessary to duplicate or consolidate all the process (and so, all the services that compose the process) while the bottleneck comes from one or some services of the process. Consequently, our goal here is to provide elasticity mechanisms (duplication/consolidation) that allow deployed SBPs to

[2] http://compatibleone.org
[3] http://easi-clouds.eu/

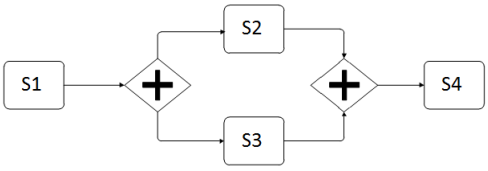


Figure 6.5: BPMN of a SBP example

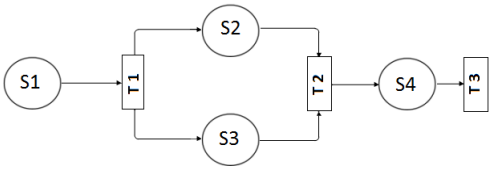


Figure 6.6: Petri net of a SBP example

scale up or down when needed. To scale up a SBP, these mechanisms have to create, as many instances as necessary, of some business services (part of the considered SBP). To scale down a SBP, they have to remove unnecessary instances of some services.

We propose in the following, a formal model to ensure SBPs elasticity by duplicating/consolidating services that compose the SBP, while preserving the semantics of the SBP.

SBP Modeling:

To model SBPs, several techniques can be used (BPEL [83], BPMN [84], Petri nets [85]). In our work, we are interested in the formal aspect of the model. So, we model the SBP using Petri nets. Many approaches model SBPs using Petri nets, but instead of focusing on the execution model of processes and their services, we focus on the dynamic (evolution) of loads on each basic service of the SBP's composition. In our model, each service is represented by at least one place (the set of places of each service are related with an equivalence relation). The transitions represent calls transfer between services according to the behavioral specification of the SBP. Our model can be considered as a deployment model in which we represent the way a process and its services are deployed and the way the load evolve between these services.

The Figure 6.5 shows a BPMN model of a SBP example composed by 4 services (S1,S2,S3 and S4). The modeling of this SBP example with our model gives the Petri net shown in Figure 6.6. It is noteworthy that we present here a simplified SBP model based on place/transition Petri nets that does not take into account state of services. A more rigorous model based on Colored Petri nets (CPN) for stateful SBPs can be found in [67].

Definition A SBP model is a Petri net $N =< P, T, Pre, Post >$:

- P: a set of places (represents the set of services/activities involving in a SBP).

- T: a set of transitions (represents the call transfers between services in a SBP).
- $Pre : P \times T \rightarrow \{0, 1\}$
- $Post : T \times P \rightarrow \{0, 1\}$

For a place p and a transition t we give the following notations:

- $p^\bullet = \{t \in T | Pre(p, t) = 1\}$
- $^\bullet p = \{t \in T | Post(t, p) = 1\}$
- $t^\bullet = \{p \in P | Post(t, p) = 1\}$
- $^\bullet t = \{p \in P | Pre(p, t) = 1\}$

Definition Let N be a Petri net, we define a net system $S = \langle N, M \rangle$ with $M : P \rightarrow \mathbb{N}$ a marking that associates to each place an amount of tokens.

The marking of a Petri net represents a distribution of calls over the set of services that compose the SBP. A Petri net system models a particular distribution of calls over the services of a deployed SBP. We assume in our model that all service calls are treated in the same manner.

Definition Given a net system $S = \langle N, M \rangle$ we say that a transition t is fireable in the marking M, noted by $M[t\rangle$ iff $\forall p \in^\bullet t : M(p) \geq 1$.

Definition The firing of a transition t in marking M changes the marking to M' s.t. $\forall p : M'(p) = M(p) + (Post(t, p) - Pre(p, t))$.

The transition firing represents the evolution of the load distribution after calls transfer. The way that calls are transferred between services depends on the behavior specification (workflow operators) of the SBP.

Elasticity Operations:

When a service has a lot of calls, it will be overloaded and this can lead to loss of QoS. A solution to this overflow problem is to duplicate the service (create a new instance of this service) in order to increase service capacity and ensure the desired QoS despite the increased load. When a service has few calls, the containers that host its instances use more resources than required. A solution is to consolidate the service (remove an unnecessary instance of this service) in order to avoid under utilization of resources.

Hereafter, we give the definition of two elasticity operators that duplicates/consolidates a service without changing underlying SBP.

Place Duplication:

Definition Let $S = \langle N, M\rangle$ be a net system and let $p \in P$, the duplication of p in S by a new place p^c ($\notin P$), noted as $D(S, p, p^c)$, is a new net system $S' = \langle N', M'\rangle$ s.t

- $P' = P \cup \{p^c\}$
- $T' = T \cup T''$ with $T'' = \{t^c | t \in ({}^\bullet p \cup p^\bullet)\}$
- $Pre' : P' \times T' \to \{0, 1\}$
- $Post' : T' \times P' \to \{0, 1\}$
- $M' : P' \to \mathbb{N}$ with $M'(p') = M(p')$ if $p' \neq p^c$ and 0 otherwise.

The Pre' (respectively $Post'$) functions are defined as follows:

$$Pre'(p', t') = \begin{cases} Pre(p', t') & p' \in P \wedge t' \in T \\ Pre(p', t) & t \in T \wedge t' \in (T' \setminus T) \\ & \wedge p' \in (P \setminus \{p\}) \\ Pre(p, t) & t \in T \wedge t' \in (T' \setminus T) \\ & \wedge p' = p^c \\ 0 & otherwise. \end{cases}$$

$$Post'(t', p') = \begin{cases} Post(t', p') & p' \in P \wedge t' \in T \\ Post(t, p') & t \in T \wedge t' \in (T' \setminus T) \\ & \wedge p' \in (P \setminus \{p\}) \\ Post(t, p) & t \in T \wedge t' \in (T' \setminus T) \\ & \wedge p' = p^c \\ 0 & otherwise. \end{cases}$$

Place Consolidation:

Definition Let $S = \langle N, M\rangle$ be a net system and let p, p^c be two places in N with $p \neq p^c$, the consolidation of p^c in p, noted as $C(S, p, p^c)$, is a new net system $S' = \langle N', M'\rangle$ s.t

- N': is the net N after removing the place p^c and the transitions $(p^c)^\bullet \cup^\bullet p^c$
- $M' : P' \to \mathbb{N}$ with $M'(p) = M(p) + M(p^c)$ and $M'(p') = M(p')$ if $p' \neq p$.

Example Figure 6.7-(a) shows an example of nets system (empty marking) that represents the SBP of Figure 6.6. Figure 6.7-(b) is the resulting system from the duplication of $s2\_1$. Figure 6.7(c) is the consolidation of the place $s2\_1$ in its copy $s2\_2$. The boxes represent the equivalence relations.

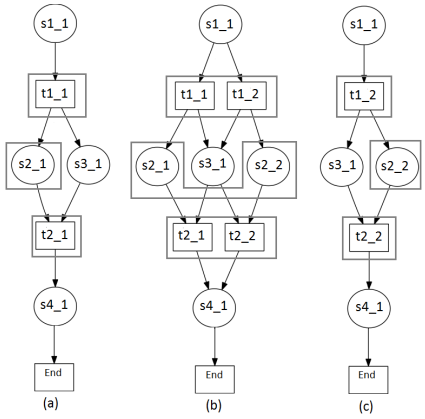


Figure 6.7: Example of the elasticity of a SBP

In a previous work [86], we proved that the semantics of the SBP is preserved by duplication and consolidation operators. To this aim, we proved that according to some equivalence relation, any sequence of transformation on a SBP is equivalent, according to some properties, to the original one. In this case here, as mentioned previously, the Petri net of a SBP does not denote an execution model but a dynamic view on the evolution of the SBP load (the marking). Therefore, we want to keep the same view of load evolution regardless of any transformation of a net. In order to do so, the following two properties have to be preserved:

- **Property 1:** By any transformation of the net using duplication/consolidation operators, we do not lose or create SBP invocations (i.e., the load in terms of the number of requests of all the instances of a given service must be the same as the load of the original one without duplications/consolidations).
- **Property 2:** The dynamics in terms of load evolution of the original process must be preserved in the transformed one (i.e., for any reachable load distribution in the original net there is an equivalent (according to property 1) reachable load distribution in the transformed net).

In this section, we introduced our formal model to describe SBPs and two elasticity operations (duplication/consolidation). In the next section, we present how we can couple these elasticity operations with a generic controller to ensure elasticity of deployed SBPs.

Generic Controller for SBPs elasticity:

Several strategies can be used [87, 88, 89] to manage SBPs elasticity. A strategy is responsible of making decisions on the execution of elasticity mechanisms (i.e., deciding when and how to use these mechanisms). The abundance of possible strategies requires a generic solution which allows the implementation and execution of different elasticity strategies. For this reason, we propose a framework, called generic Controller, for SBPs elasticity in which different elasticity strategies can be implemented. Our generic Controller has the capability to perform three actions:

- Routing: Is about the way a load of services is routed over the set of their copies. It determines under which condition we transfer a call. We can think of routing as a way to define a strategy to control the flow of the load (e.g., transfer a call if and only if the resulted marking does not violate the capacity of the services.)
- Duplication: Is about the creation of a new instance of an overloaded service in order to meet its workload.
- Consolidation: Is about the deletion of an unnecessary instance of a service in order to meet its workload decrease.

If we consider the three actions that can be performed by the elasticity Controller, any combination of conditions associated with a decision of routing, duplication and consolidation is an elasticity strategy.

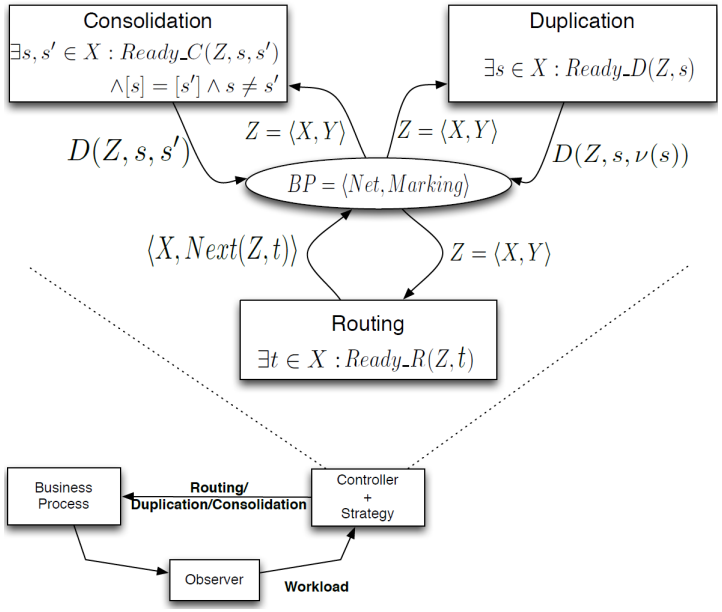


Figure 6.8: HLPN model of the generic Controller

It is worthy to note that our goal here is not to propose an additional elasticity strategy, but a framework, called generic Controller, to execute different strategies. In order to simulate the execution of an instantiated Controller, we propose to model the Controller as a high level Petri net (HLPN). Indeed, by instantiating our generic Controller one can observe and evaluate the execution of the implemented strategies using the HLPN analysis tools. The structure of the Controller is shown in Figure 6.8. The Controller contains one place (BP) of type net system. The marking of this place evolves either by calls transfer (Routing), by the duplication of an overloaded service (Duplication) or by the consolidation of an underused service (Consolidation). These three transitions (actions) are guarded by conditions that decide when and how to perform these actions:

- Routing: This transition is fireable if we can bind the variable Z to a net system $S = \langle N, M \rangle$ where exists a transition t fireable in S and the predicate $Ready\_R(S, t)$ is satisfied. The firing of the Routing transition adds the net system S after the firing of t ($Next(Z, t)$ returns the marking after the firing of t).
- Duplication: This transition is fireable if we can bind the variable Z to a net system $S = \langle N, M \rangle$ where exists a place s and the predicate $ready\_D(Z, s)$ is satisfied. The firing of the Duplication transition adds a new net system resulted from the duplication of s in S.
- Consolidation: This transition is fireable if we can bind the variable Z to a net system $S = \langle N, M \rangle$ where exist two instances of the same service, s and s', and the predicate $ready\_C(Z, s, s')$ is satisfied. The firing of the Consolidation transition adds a net system resulted from the consolidation of s' in S.

The execution of these actions is performed after checking the guards of the execution of these actions ($ready\_R$, $ready\_D$, $ready\_C$). The elasticity conditions that decide when duplicate/consolidate a service are implemented in predicates $ready\_D$ (for duplication) and $ready\_C$ (for consolidation) while the condition that decides on how the service calls are routed is implemented in the predicate $ready\_R$. In our Controller, the conditions ($Ready\_D$, $Ready\_C$, $Ready\_R$) are generic to allow the use of different elasticity strategies. By instantiating our generic Controller, one can analyze and evaluate behaviors and performances of the implemented strategies.

6.4.2 Implementation

To implement the different Resources, Links and Mixins that we previously defined we used a JAVA implementation provided by OCCI working group called occi4java [4]. This implementation is developed by the OCCI working group according to the specifications. The project is divided into three parts. Each of these parts corresponds to one of the three basic documents of OCCI specifications (i.e., OCCI Core, OCCI Infrastructure and OCCI HTTP Rendering). We already used this project to implement our extensions of OCCI Platform [74, 76] and Application [73]. Using occi4java, we extended the Resource class to create our own resources for autonomic computing. We also extended the Link class to define our links.

Since the Mixin mechanism is not a native functionality for JAVA, we used the mixin4j framework [5] to define our Mixins. This framework allows to create Java Mixins by the use of annotations. To this end, it is possible to define the classes to be extended as abstract classes annotated with *"@MixinBase"* annotation. It is to say that this abstract class will be extended using Mixins. To create the Mixin itself, it is necessary to create an interface for this Mixin. Then, the "*@MixinType*" annotation is used to say that this interface represents a Mixin. The implementation of the interface must be specified following the "*@MixinType*" annotation. Consequently, the framework allows to instantiate new classes from the abstract classes annotated with "*@MixinBase*" mixed with implementations of the interfaces annotated with "*@MixinType*". We annotated our OCCI Resources and Links with "*@MixinBase*" to be able to

[4] https://github.com/occi4java/occi4java
[5] http://hg.berniecode.com/java-mixins

extend them later. Then, we defined all the interfaces of our Mixins and annotated them with "*@MixinType*". Moreover, we implemented the different Mixins containing the needed functionalities for our autonomic infrastructure. In our implementation, the *Autonomic Manager* is implemented as an abstract Mixin Base. Using mixin4j, we mixed it with an implementation of the *SpecificSLAInterface* as a MixinType and a *WSAgMixinImpl* that represents the real implementation that uses *WSAgreementParser* to parse WS-Agreements SLAs. Similarly, the *Analyzer* and *Reconfiguration Manager* are implemented as MixinBase. The *Strategies* and the *ReconfigurationActions* Mixins implement respectively the MixinType interfaces *StrategiesInterface* and *ReconfigurationActionsInterface*.

For REST HTTP Rendering, the occi4java project uses Restlet framework [6]. Restlet is an easy framework that allows to add REST mechanisms. After adding the needed libraries, one needs just to add the Restlet annotations to implement the different REST actions (i.e., POST, GET, PUT and DELETE). Then to set up the server, one has to create one or more Restlet Components and attach the resources to them. We used the proposed annotations to define the needed actions for our OCCI Resources, Links and Mixins. That allowed us to use the HTTP REST Rendering to manage our autonomic infrastructure. And in order to enforce the scalability of our solution, we implemented different Restlet Components to allow the distribution of our infrastructure. We also used this framework to implement all the communication mechanisms used by Mixins to specify the Links communication (i.e., SubscriptionTool, NotificationTool, AlertTool and ActionTool Mixins use Restlet to send and receive data).

Furthermore, we used WS-Agreement to describe SLAs (see Listing 6.1). WS-Agreement is an extensible language to describe SLAs. The extendability of this language allows to define user specific elements and use them. In our example, we were able to specify the attributes to be monitored and their different thresholds. We started from the WS-Agreement XSD[7] and we added new elements to describe the analysis strategies (i.e., <*strategy:ScaleUpStrategy*>) and the reconfiguration actions (i.e, <*plan:ReconfigurationActions*>). The Mixin that we implemented to parse this kind of SLAs is used by the *Autonomic Manager* to detect the content of these elements and makes a syntactic matching between a list of existing Mixins for each type (*Strategies* or *ReconfigurationActions* Mixins). If it finds a Mixin that matches the description it uses it to extend the concerned resource. More work is needed in the specifications of strategies and reconfiguration actions in a more rigorous way.

Listing 6.1: WS-Agreement sample

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsag:AgreementOffer AgreementId="ab97c409">
3  <wsag:Name>xs:CFApplicationAg</wsag:Name>
4  <wsag:AgreementContext> ... </wsag:AgreementContext>
5  <wsag:Terms>
6   <wsag:All>
7    <wsag:GuaranteeTerm wsag:Name="g1" wsag:Obligated="ServiceProvider">
8     <wsag:ServiceLevelObjective>
9      <wsag:KPITarget>
10      <wsag:KPIName>averageResponseTime</wsag:KPIName>
11      <wsag:CustomServiceLevel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
12       xmlns:exp="http://www.telecom-sudparis.com/exp">
```

[6] http://restlet.org/
[7] http://schemas.ggf.org/graap/2007/03/ws-agreement

```
13          <exp:Less><exp:Variable>maxthreshold</exp:Variable> <exp:Value>1200</exp:Value></exp:Less>
14        </wsag:CustomServiceLevel>
15        <wsag:KPIName>minthreshold</wsag:KPIName>
16        <wsag:CustomServiceLevel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
17          xmlns:exp="http://www.telecom-sudparis.com/exp">
18          <exp:Greater><exp:Variable>minthreshold</exp:Variable><exp:Value>700</exp:Value></exp:Greater>
19        </wsag:CustomServiceLevel>
20      </wsag:KPITarget>
21    </wsag:ServiceLevelObjective>
22    <wsag:BusinessValueList>
23      <wsag:CustomBusinessValue xmlns:strategy="http://www.telecom-sudparis.com/strategy">
24        <strategy:ScaleUpStrategy><if> <ResponseTime from=maxthreshold/><then> scaleUpAlert</then></if>
25        </strategy:ScaleUpStrategy>
26        <strategy:ScaleDownStrategy><if> <ResponseTime to=minthreshold/><then> scaleDownAlert</then></if>
27        </strategy:ScaleUpStrategy>
28      </wsag:CustomBusinessValue>
29      <wsag:CustomBusinessValue xmlns:plan="http://www.telecom-sudparis.com/plan">
30        <plan:ReconfigurationActions>scaleUp</plan:ReconfigurationActions>
31        <plan:ReconfigurationActions>scaleDown</plan:ReconfigurationActions>
32        ...
33      </wsag:CustomBusinessValue>
34    </wsag:BusinessValueList>
35   </wsag:GuaranteeTerm>
36  </wsag:All>
37 </wsag:Terms>
38 </wsag:AgreementOffer>
```

In order to parse WS-Agreement SLAs, we used Eclipse Modeling Framework[8](EMF) to implement WSAgreementParser. EMF is a modeling framework that allows building tools and applications based on their structured data models (e.g., XSD). We used WS-Agreement XSD to generate the needed classes to process WS-Agreements. Based on these classes, we developed a parser that responds to our requirements. This parser is used by an instance of *SpecificSLA* Mixin.

To realize the evaluation use cases, we implemented different monitoring and reconfiguration Mixins based on OCCI REST queries. Table 6.1 resumes the different actions performed by each Mixin. Monitoring Mixin for Applications is based on the REST API proposed by NewRelic [90] service integrated in CloudFoundry PaaS. We also implemented three Mixins to monitor Compute resources. These three Mixins communicate with the OCCI server proposed by the IaaS manager (we used OpenNebula in our use case). These monitoring Mixins get the needed information about the resource and create notifications related to the monitored attributes and send them to the *Analyzer*. We implemented also different reconfiguration Mixins. Each one implements the *reconfigure()* method differently. The different Mixins implementations to reconfigure OCCI *Compute* resources, address REST queries to the OCCI server of the IaaS manager (i.e., OpenNebula). As shown in Table 6.1, we need to send an XML file describing the new targeted state of the *Compute* resource. Finally, we implemented a Mixin able to reconfigure applications deployed on CloudFoundry by adding or retrieving application instances. This latter uses a REST query to communicate with COAPS (detailed in Section 6.4.1.1). As shown in Table 6.1, the queries sent to COAPS must contain the identifier of the application as well as the instances number.

[8]http://www.eclipse.org/modeling/emf/

Mixins Role	REST Query
Monitoring *Compute* memory consumption	GET http://<OpenNebulaOCCIserver>/compute/<id>/
Monitoring *Compute* remain free disk	GET http://<OpenNebulaOCCIserver>/compute/<id>/
Monitoring *Compute* availability	GET http://<OpenNebulaOCCIserver>/compute/<id>/
Monitoring *Application* avg response time	GET http://<NewRelicInstance>/v2/applications.xml
Adding memory to *Compute*	PUT http://<OpenNebulaOCCIserver>/compute/<id>/vm.xml
Deducing memory from *Compute*	PUT http://<OpenNebulaOCCIserver>/compute/<id>/vm.xml
Hot-plugging a disk to *Compute*	PUT http://<OpenNebulaOCCIserver>/compute/<id>/vm.xml
Redeploying *Compute*	PUT http://<OpenNebulaOCCIserver>/compute/<id>/vm.xml
Hot-detaching a disk from *Compute*	PUT http://<OpenNebulaOCCIserver>/compute/<id>/vm.xml
Scaling up an *Application*	PUT http://<COAPSAddress>/coaps/<appid>/instances/<nbr>
Scaling down an *Application*	PUT http://<COAPSAddress>/coaps/<appid>/instances/<nbr>

Table 6.1: Summary of the monitoring and reconfiguration Mixins.

6.4.3 Evaluations

In this section, we will show how we can add autonomic computing management facilities to Cloud resources. To do so, we will describe two use cases. The first one deals with applications deployment in the Cloud. This use case uses the concepts described in Section 6.4.1.1 and it shows how our autonomic mechanisms can be applied on heterogeneous Cloud resources spanning over the different layers of the Cloud. The second one deals with provisioning elastic business processes in the Cloud and it is based on the concepts described in Section 6.4.1.2. It shows an example of adding a non functional service to Cloud resources based on our mechanisms.

6.4.3.1 Provisioning of Autonomic Applications on Cloud Foundry

In this section we show the usefulness of our approach. Our goal is to show how we can build an autonomic infrastructure spanning over the different layers of the Cloud. Our experiments may show that our mechanisms enhance the resource behavior. The established infrastructure have to harvest monitoring data from different target, to analyze it and to generate reconfiguration actions when needed.

Use Case description:

To evaluate our work we present a real use case that spans over the different levels of the Cloud (i.e., IaaS, PaaS and SaaS levels). It consists of the deployment of an application (i.e., the SaaS) on a private instance of CloudFoundry PaaS[9]. Our PaaS is deployed on three virtual machines managed by OpenNebula IaaS manager. OpenNebula provides an OCCI server that allows us to describe and manage the VMs as OCCI *Compute* resources.

It is noteworthy that we used OCCI extension for Infrastructure [37] to describe the IaaS level. Moreover, we used a generic model that we proposed in Section 6.4.1.1 to describe PaaS and SaaS resources as extensions of OCCI. We also used COAPS to interact with CloudFoundry PaaS.

[9] http://www.cloudfoundry.com/

The used application is a Servlet that performs numerical analysis on matrices. The Servlet receives client queries containing the size of the matrices. It generates the matrices and applies the desired computation and saves the result on the disk. In our use case, we will focus on matrices products calculation. We assume that these matrices could be with large sizes. Consequently, the used disk could be saturated. The client queries number can also increase or decrease. Consequently, the application instances could be more or less solicited. We also consider that the virtual machines instances could encounter failures or disconnections.

In order to ensure the continuity of the application execution with all its required resources, we applied our approach to add autonomic behaviors to Application and Compute resources. Our targeted metrics to monitor are (1) Compute disk usage, (2) Compute memory usage, (3) Application response time , and (4) Compute availability. The analysis of monitoring data of these metrics could result on the following reconfiguration actions (1) reconfiguring the used disk by the Compute instance by hot-plugging new disk or by resizing the original one, (2) resizing the Compute instance to add or to retrieve memory capacity, (3) auto-scaling of the application by adding or retrieving instances, and (4) redeploying a new instance of the Compute instead of the failing one.

To monitor the *Compute* instances (i.e., 3 used VMs), we implemented three monitoring Mixins. As described in Section 6.4.2, the first Mixin monitors the memory consumption of the VM, the second monitors its remaining free disk, while the third monitor its availability. All these Mixins are based on OCCI REST queries sent to OpenNebula OCCI server. The collected data is sent to the *Analyzer* as notifications. The *Analyzer* uses its Strategies Mixins to analyze these data and to generate alerts if the SLA is violated.

Accordingly, we need to monitor the Application resource. To do that, we used the associated Mixin that gathers response time related to a CloudFoundry Application. As well, this Mixin sends notifications to the *Analyzer* that applies the related strategies to verify whether the SLA is violated or not.

Table 6.2 resumes the monitoring metrics of our use case, their associated thresholds, alerts and reconfigurations. This information is directly used by our implemented Mixins. As shown in the table, we used two strategies for the *Application* response time. In each case, the *Analyzer* uses its strategies to verify if the monitoring data respects the defined thresholds or not. If the SLA is violated, the *Analyzer* sends an alert to the *Reconfiguration Manager* with the associated alert. The *Reconfiguration Manager* uses its Mixin to generate the associated reconfiguration actions. For example, if the *Compute* resource consumes more than 80% of its memory, the *Analyzer* would send an alert saying that the Compute memory is over-used. Consequently, the *Reconfiguration Manager* generates actions to reconfigure the *Compute* instance. This reconfiguration is based on the associated Mixin described in Section 6.4.2. It is worthy to note that the thresholds are deduced from the SLA, and that the analysis strategies are proactive. The strategies do not wait till the SLAs are violated, instead they anticipate by sending alerts before the occurrence of the violation. More work in this area will be conducted in our future work. But since it is not the basic objective of this work, we fixed the thresholds to $\pm$ 5 % of the hard thresholds specified in the SLA. This choice was made based on our experiments and it is specific to this use case.

The input of our OCCI Server is an SLA (similar to Listing 6.1) describing the needed *Compute* and *Application* instances and their requirements as well as the needed details for the autonomic aspects. It is passed to the *Autonomic Manager* in the following HTTP POST request:

```
> POST http://localhost:8182/amanager
```

Metrics	Thresholds		Alerts	Reconfigurations
Compute Memory	Max	80%	Memory overloaded	Add memory to Compute
	Min	20%	Memory underloaded	Retrieve memory from Compute
Compute Disk	Max	80%	Disk full	Hot-plug disk to Compute
Compute Availability	-		Compute unavailable	Redeploy Compute
Application Response Time	Max	S1: 1140	Application overloaded	Scale up Application
		S2: 760		
	Min	S1: 735	Application underloaded	Scale down Application
		S2: 525		

Table 6.2: Description of monitoring metrics, thresholds, alerts and reconfigurations of our use case.

```
> Category: amanager; class="kind";
X-OCCI-Attribute: name=AManager
X-OCCI-Attribute: version=1
X-OCCI-Attribute: slalocation=../wsAgSLA.xml
Content-Type: text/occi
```

The *Autonomic Manager* uses the SLA to build the environment where to deploy the application. It caries on the list of queries to the OCCI Server to instantiate the three Mixed *Compute* instances (with monitoring and reconfiguration Mixins) using their identifiers. These instances are configured to run a CloudFoundry instance. Once deployed, they offer an endpoint of this PaaS that we specified as *api.cloudfoundry.io*. Afterwards the *Autonomic Manager* sends the needed queries so that the OCCI server deploys and starts the Application extended with monitoring and reconfiguration Mixins. To this aim, the OCCI Server uses the actions proposed by COAPS.

After the provisioning of the application, the *Autonomic Manager* proceeds to the instantiation of all the needed Resources, Links and Mixins via the OCCI Server. From the SLA, the *Autonomic Manager* detects the metrics to be monitored. It extracts also the defined thresholds as well as the analysis strategies and the reconfiguration actions. These information are described using user specific elements that we added to WS-Agreement as shown in Listing 6.1 (lines 23-31).

Evaluation Results:

In these evaluations, we took different measurements of the needed time to deploy and start *Application* and *Compute* resources and the needed time to build the autonomic infrastructure. The measurements show that the deployment and starting time depends on the size of the Resource. However, the establishment of the rest of the autonomic infrastructure is independent of the Resource. The instantiation of a *Compute* instance takes 15.8 seconds. And since the queries are simultaneous, the instantiation of the three instances takes 38.7 seconds. To deploy our test application, having the size of 8.4 Kb, on CloudFoundry we measured 48.03 seconds. To start this application we measured 11.3 seconds. The needed time to establish the rest of the autonomic infrastructure is independent of the resources size, but it depends on the number of the managed resources. This time was around 5.3 seconds. This time is encouraging since it is negligible compared to the resources deployment and starting time. During

Reconfiguration Action	Time (s)
Adding Memory to Compute	23.5
Retrieving Memory from Compute	24.6
Hot-plugging disk to Compute	7.9
Hot-detaching disk from compute	8.2
Redeploying Compute instance	20.2
Scaling up an application	11.6
Scaling down an application	11.4

Table 6.3: Average times needed to reconfiguration actions.

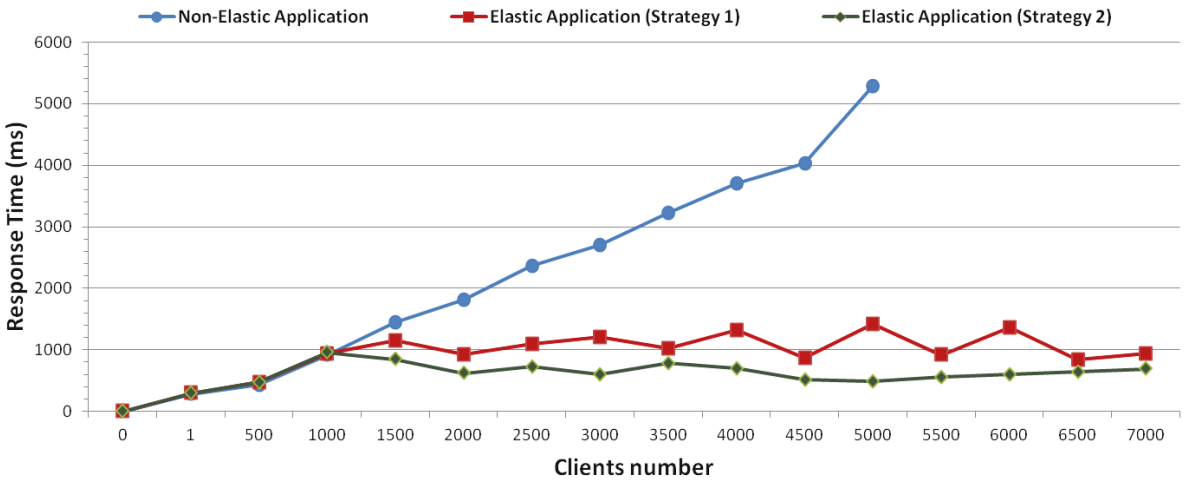


Figure 6.9: Response time before and after adding our Autonomic Infrastructure.

the experiments, we measured the needed time to apply the different reconfiguration actions. Table 6.3 resumes the average needed times to apply each reconfiguration action.

Furthermore, we compared the behavior of the application before and after adding our autonomic infrastructure. We used two strategies (see S1 and S2 in Table 6.2) to see the impact of the strategy choice on the autonomic behavior of the application. Moreover, we developed a REST client able to send queries to the deployed application. We deployed this client on different VMs in order to launch a large number of parallel calls. This is to create a client query burst targeting the application. In each series of this experiments we modified the number of the used clients targeting our application and we saved the measured response times. The results of this evaluation are shown in Figure 6.9.

The response time of the application without our mechanisms is increasing proportionally with the number of clients queries. Moreover, after 5000 parallel clients, the application went down and its execution continuity was broken. In contrast, using our mechanisms, whenever the response time approached the max threshold that we specified in the SLA, a reconfiguration action is triggered to scale up the application by adding a new instance. This reconfiguration decreases the response time of the application.

If the response time is approaching the min threshold, a reconfiguration action is triggered to scale down the application. The experiments show the efficiency of our approach to enhance the behavior of the application. In fact, over 1000 clients, the response time remains around the specified thresholds. The experiments show also that we reached 7000 clients without any downtime of the application. We noticed that using the two strategies the response time remains between the specified thresholds. However, Strategy 2 consumes more memory than Strategy 1 to conserve a lower response time. So whenever we want to decrease the thresholds we need to consume more memory since the infrastructure will add more instances of the application to guarantee the desired response time.

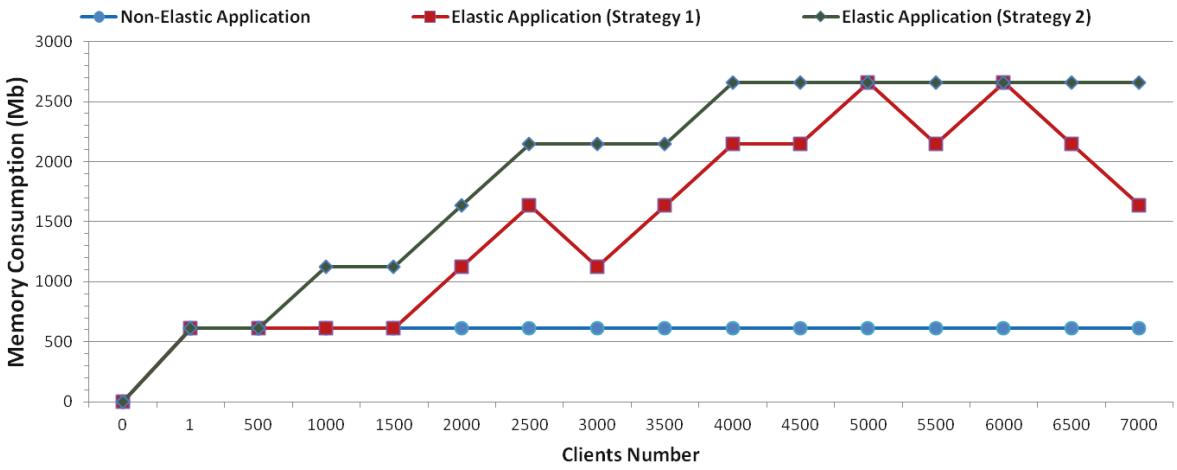


Figure 6.10: Memory consumption before and after adding our Autonomic Infrastructure.

Over and above the response time of the *Application*, we measured its memory consumption. The Figure 6.10 shows the difference between the memory consumption of an *Application* with and without our mechanisms (using two strategies). The memory consumption of a non autonomic one is constant since it has a constant allocated memory and no new instances are added. Meanwhile, for an autonomic *Application*, the curve shows that the memory consumption is changing whenever there is a duplication or a consolidation. The cost of a duplication correspond to 512 Mb that represents the size of an added application instance. These changes in the memory consumption could have an impact on the other metrics and reconfiguration actions (i.e., memory usage of the *Compute* instances and their configuration). Strategy 2 is consuming more memory in order to guarantee a lower response time. These results open new perspectives for further studies to fine tune the choice of the used thresholds.

As previously mentioned, the thresholds used for analysis strategies are deduced from the values specified in the SLA. We started our experiments by using the hard thresholds specified in the SLA and we noticed that there are some violations. In order to guaranty the accuracy of the system regarding the specified SLA, we start gradually to add $\pm$ percentage to each threshold value. After multiple experiments series, we reached 100% of accuracy by adding $\pm$ 5% to all the specified thresholds in the SLA. We note that this value is specific to this use case. This direction needs more attention in our future work.

In this section, we proved that our approach is efficient when applied to real resources spanning over

the different layers of the Cloud. The obtained results show that our mechanisms do not represent a large time or resource consumption overhead compared to traditional Cloud resources. In the following, we apply our approach on a Service-based Business Process to render it elastic by means of our mechanisms.

6.4.3.2 Provisioning of Elastic SBP on Cloud Foundry

Our focus in this section is on the Service-based Business Processes (SBPs) level and more particularly on SBPs elasticity. We will specialize our autonomic mechanisms to add elasticity to Service-based Business Processes (SBPs) in Cloud environment. We will use the different concepts described in 6.4.1.2. We will merely break the Controller between the *Analyzer* and the *Planner* Resources. Consequently, the *ready_D* and *ready_C* are implemented as instances of the *Strategies* Mixin while the Duplication and Consolidation are implemented as instances of the *Reconfiguration Actions* Mixin. Notwithstanding, spurred by the fact that almost all the existing Cloud providers offer the routing functionality, we delegate the routing mechanisms to the targeted environment in which we will deploy our SBP.

Use Case description:

To evaluate our work we present a real use case of Service-based Business Processes. It consists on the deployment of an SBP on the public PaaS CloudFoundry [10]. For the SaaS and PaaS descriptions and provisioning we are using COAPS 6.4.1.1. Our goal is to show how we add granular elasticity mechanisms to SBPs in a generic manner. For evaluation aims, we create an SBP composed of three services collaborating to generate random matrices and to apply numerical analysis on them. The first service receives the sizes of the matrices and the desired analysis operation from the client and it passes this parameter to the second service. This latter generates two matrices and saves them on the disk and passes their addresses to the third service along with the desired operation. The third service applies the operation on the matrices and sends back the result to the client. We included the needed information related to the deployment of each service of the SBP and the establishment of the infrastructure in an SLA expressed with the extended WS-Agreement that we presented in Section 6.4.2. Afterwards, we run our OCCI Server on VMs deployed by OpenNebula IaaS manager. The input of our OCCI Server is an SLA describing the SBP and its requirements as well as the needed details for the autonomic aspects. The *Autonomic Manager* uses the SLA to build the environment where to deploy the SBP. It caries on the list of queries to the OCCI Server to deploy and start services of the SBP after adding the needed Mixins (i.e., monitoring and reconfiguration Mixins) to each service. To deploy and start the services, the OCCI Server uses the actions proposed by COAPS to this aim. In the realized use case, the Subscription Mixin uses a *NewRelic* monitoring service [90] to get monitoring data about each service instance. The realized infrastructure is shown in Figure 6.11.

After the provisioning of the SBP, the *Autonomic Manager* proceeds to the instantiation of all the needed Resources and Links via the OCCI Server. From the SLA, the *Autonomic Manager* detects the need to monitor the response time of each service. It extracts also the defined thresholds as well as the analysis elasticity strategies and the reconfiguration actions. These information are described using user specific elements that we added to WS-Agreement. As shown in Listing 6.1 the max and min thresholds are fixed respectively to 1200 and 700 ms for all the services. The monitoring data gathered

[10] http://www.cloudfoundry.com/

by the Subscription Mixin are parsed by this latter. It extracts the response time of each service and sends eventual notifications to the *Analyzer* via the related *Notification Link*. For each notification, the *Analyzer* applies its elasticity strategies to verify that the response time respects the SLA. If there is any violation, the *Analyzer* resource triggers an alert targeting the *Planner* via the *Alert Link*. This *Planner* generates reconfiguration actions to duplicate or consolidate the concerned service.

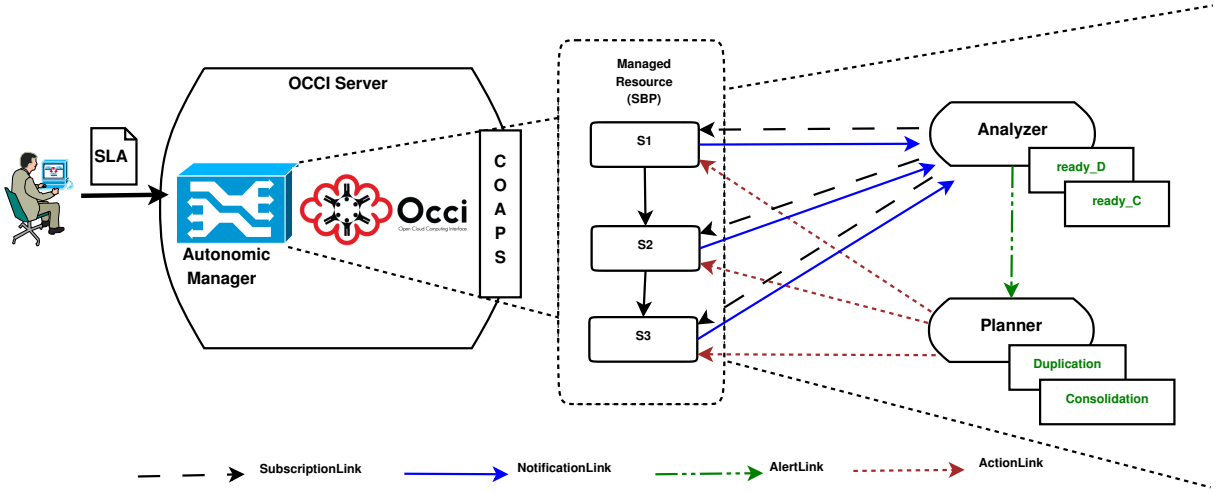


Figure 6.11: Autonomic Computing Infrastructure establishment for SBP elasticity.

At this step, we showed how the *Autonomic Manager* can instantiate the autonomic loop with all the needed Resources, Links and Mixins. When a service of the SBP is overloaded, the response time of this service will change and become greater than max threshold. Using its *Subscription* Mixin, the concerned service generates a notification containing the new value of the monitored attribute. The notification is sent via the *Notification Link* to notify the *Analyzer*. At the reception of the notification, this latter applies its strategies to verify if the SLA is violated or not. If the $ready_D$ is verified, the *Analyzer* raises a *duplicationAlert* and send it to the *Planner* via the *Alert Link*. Receiving this alert, the *Planner* verifies the list of applicable plans. Consequently, it generates the needed actions to ensure the *Duplication* via the *Action Link*. This latter asks COAPS to add more instances to the concerned service.

When a service is underloaded, the *Subscription* Mixin of the service will generate a notification and send it to the *Analyzer*. If the $ready_C$ strategy is verified, the *Analyzer* sends a *consolidationAlert* to the *Planner*. This latter, generates the needed actions to ensure the *Consolidation*.These actions are addressed to COAPS via the *Action Link*. In order to avoid the shutdown of the application, the *Action Link* verifies if the instances number is greater than 1 to forward the *Consolidation* action to COAPS.

Evaluation Results:

In our tests, we took different measurements of the needed time to deploy and start an SBP and to build the related autonomic infrastructure. To deploy our test SBP composed of three services to CloudFoundry we measured 155.12 seconds. To start this same SBP we measured 35.9 seconds. The time needed to establish the autonomic infrastructure is around 16.5 seconds.

Furthermore, to compare the behavior of the SBP in CloudFoundry before and after adding our autonomic infrastructure, we developed a REST Client able to call the deployed SBP. We deployed this client on different VMs on the NCF in order to launch a large number of parallel calls. In each series of this experiments we modified the number of the used clients targeting our SBP and we saved the measured response times. The results of this evaluation are shown in Figure 6.12.

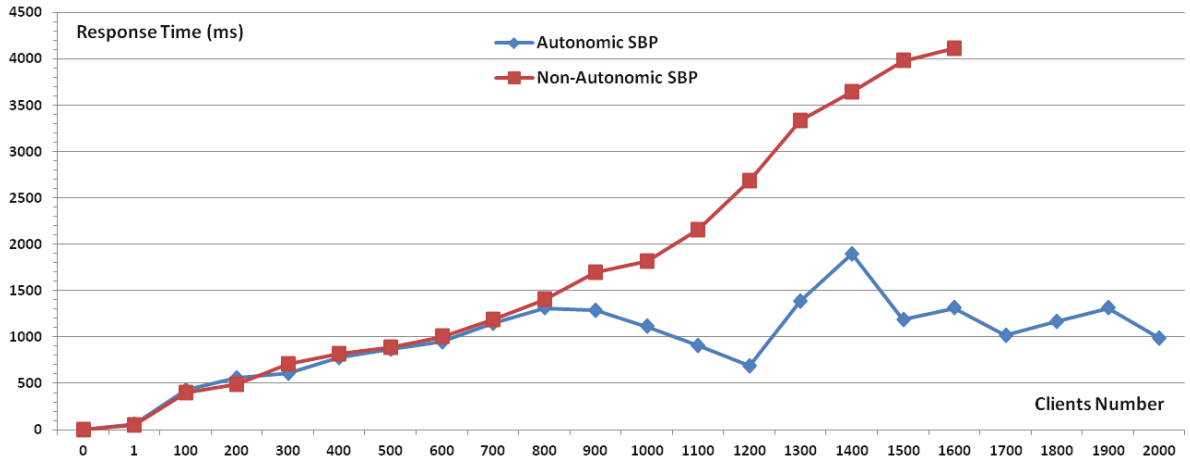


Figure 6.12: Response time of the SBP before and after adding our Autonomic Infrastructure.

The response time of the SBP without our mechanisms is increasing proportionally with the number of clients queries. Moreover, after 1600 clients, the SBP went down. In contrast, using our mechanisms, whenever the response time reached the max threshold that we specified in the SLA (i.e., 1200 ms in this case), a reconfiguration action is triggered to duplicate the overloaded service by adding a new instance to it. This reconfiguration decreases the response time of all the SBP. If the response time is under the min threshold that we specified in the SLA (i.e., 700 ms in this case), a reconfiguration action is triggered to consolidate the service instances. The experiments show the efficiency of our approach to enhance the behavior of the SBP. In fact, over 700 clients, the response time remains around the specified thresholds. The experiments show also that we reached 2000 clients without any downtime of the SBP. As shown in Figure 6.12, in some points, the response time is over the max threshold, this is due to the number of clients queries sent before the reconfiguration action takes effect.

During this series of experiments, we measured also the needed time to apply a reconfiguration action (i.e., duplication or consolidation). The needed time to reconfigure a service is about 11.8 seconds. This time is negligible compared to the needed time to redeploy and start all the SBP. Over and above the response time of the SBP, we measured its memory consumption. The results shown in Figure 6.13 show the difference between the memory consumption of a traditional SBP and an elastic one (enhanced with our mechanisms). The memory consumption of a traditional SBP is constant since each service has its allocated memory and no new instances are added. Meanwhile, for an elastic SBP, the curve shows that the memory consumption is changing whenever there is a duplication or a consolidation. The cost of a duplication correspond to 512 mb that represents the size of an added service instance. These results

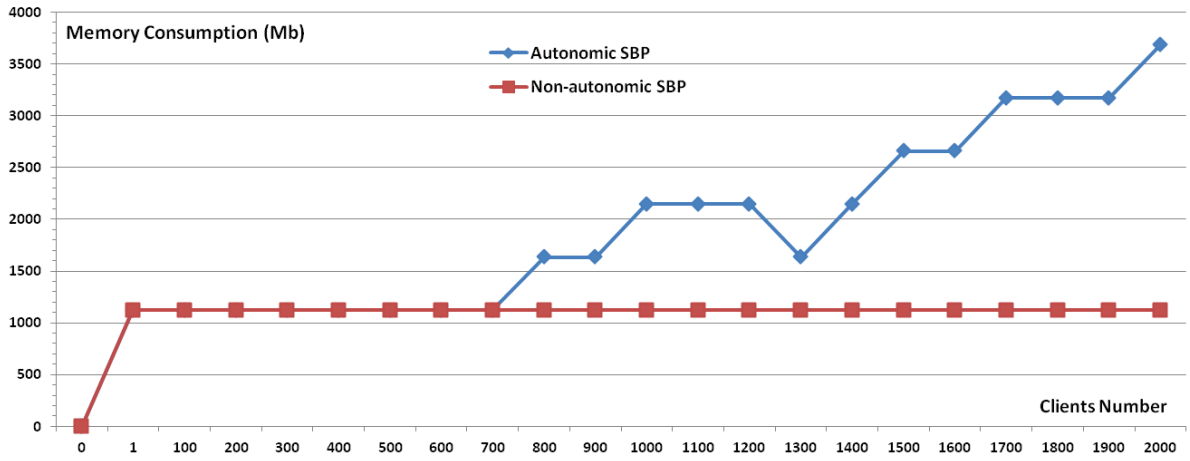


Figure 6.13: Memory consumption before and after adding our Autonomic Infrastructure.

open new perspectives for further studies to fine tune the choice of the used thresholds according to the user requirements.

In this section, we proved that our approach is efficient when applied to a real SBP deployed in a public PaaS. The obtained results open new perspectives and future work.

6.5 Conclusion

Cloud Computing is changing perceptions of IT technologies. Although many solutions are proposed to be in line with Cloud characteristics, consumers do not accept to adopt a solution that did not proved its efficiency in real environments. In this chapter, we proposed different scenarios of resources management in the Cloud. We showed the feasibility and efficiency of our approach of monitoring and reconfiguration of SCA-based applications in Cloud environments. Afterwards, we explained the usage of our autonomic infrastructure for Cloud resources by means of two use cases. The first use case shows how our autonomic mechanisms could be applied on heterogeneous resources spanned on different layers in the Cloud, while the second use case shows that using our mechanisms we can easily add non functional services to Cloud resources. We presented the different experiments that we realized to prove the efficiency of our work. Indeed, the results are encouraging. At the same time, they opened new perspectives to our work.

Chapter 7

Conclusion and Perspectives

Contents

7.1 Contributions 107

7.2 Perspectives 108

7.1 Contributions

Cloud Computing is a recent trend in Information Technologies (IT), it refers to a model for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal effort or service provider interaction [18]. In this paradigm, there are three well discussed layers of services known as "IaaS" for Infrastructure as a Service, "PaaS" for Platform as a Service and "SaaS" for Software as a Service. Other "XaaS" terms are used nowadays to name different resources provided as services in the cloud.

Cloud environments can be used to host applications that follow Service Oriented Architecture (SOA). SOA is a collection of services that communicate with each other [91]. Each service must be self-contained (i.e., it can always provide the same functionality, independently of other services). Such type of applications can be described using Service Component Architecture (SCA) as a composite that contains a detailed description for different components of the application and links between them. A well described composite can be passed to a SCA runtime (e.g, FraSCAti [20], TUSCANY [21]) that instantiates the different components and links them according to the composite description.

Management of SCA-based applications in Cloud environments still did not get the needed attention in research work. Particularly, monitoring and reconfiguration that face many challenges. The requirement for these non functional facilities needs a generic and granular description. Consequently, this description can be used to automatically add the needed mechanisms to gather monitoring data and to enable reconfigurations when needed. At the same time, the used mechanisms should be in line with the scalability of the environment.

In this PhD thesis, we dressed a list of objectives in order to provide a solution that satisfies our requirements of adding management facilities in the Cloud. We started by studying different existing works. Our study of the state of the art was divided to three parts treating respectively monitoring, reconfiguration and autonomic management. Afterwards, we made a synthesis of the presented work to recapitulate the advantages and drawbacks of each one. Consequently, we proposed two major contributions in order to cover the objectives that we dressed for our solution.

In the first contribution, we proposed an extension of Service Component Architecture (SCA) component model. At a first step, the extension allows to express monitoring and reconfiguration requirements related to components. Later steps allowed us to add self-management behaviors to components. Using SCA we were able to express FCAPS (Fault, Configuration, Accounting, Performance and Security) properties of components. In this work we proposed a list of transformations that allows to dynamically add the non functional facilities of monitoring, reconfiguration and FCAPS management to components even if they were designed without them. Furthermore, in order to be in line with the scalability of Cloud environment, we used a deployment mechanism based on a scalable micro-container able to host components. In this manuscript, we detailed all the aspects of our implementation as well as the evaluation. The different experiments that we performed in a real Cloud environment show the added value of our contribution in adding monitoring, reconfiguration and self-management facilities to service-based applications in the Cloud.

While generic and efficient, our first contribution remains limited to SaaS and PaaS levels. Consequently, we go further in our reflexion, and we proposed in a second contribution a generic solution agnostic to the Cloud service level. This contribution consists on a generic description of monitoring and reconfiguration for Cloud resources extending the *de facto* standard OCCI [2] (Open Cloud Computing Interface defined by the Open Grid Forum). The extension entails all the needed OCCI Resources, Links and Mixins that allow gathering monitoring data from resources and applying reconfiguration actions in order to ensure a specific business objective. The framework that we proposed adds monitoring and reconfiguration facilities to Cloud resources and deploys them in a Cloud environment (i.e., IaaS or PaaS). The framework offers the needed mechanisms to monitor and/or reconfigure any Cloud resource. Afterwards, we extended our work to couple monitoring and reconfiguration facilities in a holistic autonomic loop. This extension allows continuously monitoring the deployed resources and eventually adapting them according to a previously defined rules and strategies. To this end, we defined the needed Resources, Links and Mixins to create a holistic MAPE loop. The implementation of the different elements and mechanisms that we realized is detailed. We also evaluated our proposals in different scenarios to show how generic and efficient it is.

The contributions that we presented in this manuscript respect the research objectives that we dressed since the beginning of this work. Meanwhile, the first contribution is still limited to the SaaS and PaaS levels. However, the second contribution is generic and level-agnostic.

7.2 Perspectives

During our work, we faced different complex problems. We solved various of them and we included others in our future work. The evaluation also opened new research perspectives to follow in short,

medium and long term.

As short term work, we aim to go further in the dynamic processing of SLAs. In our evaluation, we started by proposing a Mixin able to process WS-Agreement SLAs. This Mixin is based on a parser that we implemented using Eclipse Modeling Framework. We are aware that this work is still in early stage and we would like to enhance it and evaluate its efficiency and accuracy. Since the syntactic matching between the requirements specified in the SLA and the proposed management mechanisms provided as Mixins is not always sufficient, we plan to add semantic processing to SLAs in order to allow the dynamic semantic matching between SLA contents and Mixins. We are aware that adding semantics to SLA will bring new challenges. But we are confident that the maturity of the research works in semantic processing could help us to provide a more efficient solution.

As medium term future work, we will focus on two major related challenges. First, we aim to enable the establishment of a standardized SLA negotiation mechanism. To do that, we can be inspired from existing works (e.g., CompatibleOne [23] using WS-Agreement and EASI-CLOUDs[24] using WS-Agreement and Linked-USDL) and the proposed mechanisms for negotiation. We are also aware of the effort made by OCCI working group on this direction. This latter work is still at first stage and it is in a slow evolution. Providing an OCCI-compliant model for SLA could be in line with our mechanisms for autonomic management. Second, we would like to enhance the analysis rules and strategies description and processing. As we noticed during our evaluation, there are possible violation of the SLA even when we use our mechanisms. These violation impacts the accuracy of the system regarding its SLA. In order to enforce the accuracy of our solution, we aim to add predictive analysis rules and strategies. These rules and strategies would be able to trigger adaptation before the occurrence of the violation. Based on the usage history, these rules and strategies are in a continuous adaptation. In this work, we have to propose algorithms that allows a consumer to choose the best strategies that suits its requirements.

Finally, as long term work, we will provide a generic model that describes the environment resources provided by complex systems. Consequently, we will be able to define the needed mechanisms that allow transforming any system to an autonomic one that has the ability to monitor and adapt itself following the changes of its environment and its behavior. To make that possible, we consider an autonomic component-based complex system as composition of autonomic components. Each autonomic component is responsible for managing its role to finalize the global business objective of the system. The autonomic management of each component may be based on dynamically generated strategies. These strategies can be extracted dynamically from the related Service Level Agreement. Based on this latter and on the environment description, the system deploys the components on the best matching resources. Therefore, the deployments of the system components are eventually distributed over different nodes or different environments. Thus, in order to ensure the global objective of the system, each component is governed using two types of strategies: local strategies that govern the local behavior of the component without having any impact on the rest of the system and global strategies that have an impact on one or more components. For strategies having an impact on one or more components, there is a need to specify how different autonomic components collaborate in order to realize the global autonomy of the system taking into account the dynamic nature of complex systems and their scalability.

Bibliography

[1] IBM Corp., "An architectural blueprint for autonomic computing," Oct. 2004. xiii, 11

[2] R. Nyren, A. Edmonds, A. Papaspyrou, and T. Metsch, "Open Cloud Computing Interface - Core," Tech. Rep., 2011. [Online]. Available: http://www.ogf.org/documents/GFD.183.pdf xiii, 5, 14, 15, 62, 66, 70, 75, 108

[3] Nagios Entreprises, "Nagios Documentation." 2010. [Online]. Available: http://www.nagios.org/documentation xiii, 18, 35

[4] G. Katsaros, G. Gallizo, R. Kübert, T. Wang, J. O. Fitó, and D. Henriksson, "A Multi-level Architecture for Collecting and Managing Monitoring Information in Cloud Environments." in *Proceedings of the third International Conference on Cloud Computing and Services Science*, 2011, pp. 232–239. xiii, 18, 19, 35

[5] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation And Experience," *Parallel Computing*, vol. 30, p. 2004, 2003. xiii, 18, 19, 20, 35

[6] Z. Balaton, P. Kacsuk, and N. Podhorszki, "Application Monitoring in the Grid with GRM and PROVE," in *Computational Science*, ser. Lecture Notes in Computer Science, V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. Tan, Eds. Springer Berlin Heidelberg, vol. 2073, pp. 253–262. [Online]. Available: http://dx.doi.org/10.1007/3-540-45545-0_34 xiii, 20

[7] M. Baker and G. Smith, "GridRM: an extensible resource monitoring system," in *Proceedings of IEEE International Conference on Cluster Computing*, Dec 2003, pp. 207–214. xiii, 21, 35

[8] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke, "A directory service for configuring high-performance distributed computations," in *Proceedings of The Sixth IEEE International Symposium on High Performance Distributed Computing.*, Aug 1997, pp. 365–375. xiii, 22, 23, 35, 36

[9] J. Almeida, M. van Sinderen, L. Pires, and M. Wegdam, "Platform-independent dynamic reconfiguration of distributed applications," in *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems* , May 2004, pp. 286–291. xiii, 25, 35

[10] A. Rasche and A. Polze, "Configuration and dynamic reconfiguration of component-based applications with Microsoft .NET," in *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May, pp. 164–171. xiii, 27, 35

[11] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin, *A Practical Guide to the IBM Autonomic Computing Toolkit*, 2004. [Online]. Available: http://books.google.fr/books?id=XHeoSgAACAAJ xiii, 2, 11, 28, 29, 35, 67

[12] B. Solomon, D. Ionescu, M. Litoiu, and G. Iszlai, "Designing autonomic management systems for cloud computing," in *International Joint Conference on Computational Cybernetics and Technical Informatics*, 2010. xiii, 28, 30, 35

[13] R. Buyya, R. Calheiros, and X. Li, "Autonomic Cloud computing: Open challenges and architectural elements," in *Third International Conference on Emerging Applications of Information Technology*, 2012. xiii, 2, 3, 29, 30, 35

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct 2004. xiii, 30, 31, 35

[15] R. Asadollahi, M. Salehie, and L. Tahvildari, "StarMX: A framework for developing self-managing Java-based systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, May 2009, pp. 58–67. xiii, 31, 32, 35

[16] D. Romero, R. Rouvoy, L. Seinturier, S. Chabridon, D. Conan, and N. Pessemier, "Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments," in *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*, M. Sheng, J. Yu, and S. Dustdar, Eds. Chapman and Hall/CRC, May 2010, pp. 113–135. [Online]. Available: http://hal.inria.fr/inria-00414070 xiii, 31, 32, 35, 36

[17] C. Ruz, F. Baude, and B. Sauvan, "Component-based generic approach for reconfigurable management of component-based SOA applications," in *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond*, 2010. xiii, 33, 35, 36

[18] NIST, "Final Version of NIST Cloud Computing Definition Published." http://www.nist.gov/itl/csd/cloud-102511.cfm, 2011. 1, 12, 107

[19] J. Rivera and R. van der Meulen, "Gartner Says the Road to Increased Enterprise Cloud Usage Will Largely Run Through Tactical Business Solutions Addressing Specific Issues," Tech. Rep., 2013. [Online]. Available: http://www.gartner.com/newsroom/id/2581315 1

[20] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable SCA Applications with the FraSCAti Platform," in *Proceedings of IEEE International Conference on Services Computing*, Bangalore, India, 2009, pp. 268–275. 2, 33, 107

[21] S. Laws, M. Combellack, H. Mahbod, and S. Nash, *Tuscany SCA in Action.*, 2011. [Online]. Available: http://dl.e-book-free.com/2013/07/tuscany_sca_in_action.pdf 2, 107

[22] P. Hnětynka and F. Plášil, "Dynamic Reconfiguration and Access to Services in Hierarchical Component Models," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, I. Gorton, G. Heineman, I. Crnković, H. Schmidt, J. Stafford, C. Szyperski, and K. Wallnau, Eds., vol. 4063. Springer Berlin Heidelberg, pp. 352–359. [Online]. Available: http://dx.doi.org/10.1007/11783565_27 2, 10, 24, 35

[23] "CompatibleOne: the first real Open Cloud Broker," 2013. [Online]. Available: http://www.compatibleone.org 3, 109

[24] " EASI-CLOUDS: Extensible Architecture and Service Infrastructure for Cloud-aware Software ," 2014. [Online]. Available: http://easi-clouds.eu 3, 109

[25] "Open Cloudware," 2013. [Online]. Available: http://www.opencloudware.org 3

[26] "Cloud4SOA," 2012. [Online]. Available: http://www.cloud4soa.eu 3

[27] T. Metsch and A. Edmonds, "Open Cloud Computing Interface - RESTful HTTP Rendering," Tech. Rep., 2011. [Online]. Available: http://www.ogf.org/documents/GFD.185.pdf 5, 14, 15, 87

[28] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web Services Architecture," *W3C Working Group Note*, vol. 11, pp. 2005–1, 2004. 8

[29] S. Burbeck, "The Tao of e-business services: The evolution of Web applications into service-oriented components with Web services," IBM Software Group, 2000. [Online]. Available: http://www-106 8

[30] M. P. Papazoglou, "Service-oriented computing: concepts, characteristics and directions," in *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, Dec. 2003, pp. 3–12. 8

[31] Open SOA Collaboration, "Service Component Architecture (SCA): SCA Assembly Model v1.00 specifications." 2008. [Online]. Available: http://www.osoa.org 8, 9, 38

[32] M. Mohamed, D. Belaïd, and S. Tata, "How to Provide Monitoring Facilities to Services When They Are Deployed in the Cloud?" in *Proceedings of the International Conference on Cloud Computing and Services Science*, 2012, pp. 258–263. 9, 50

[33] ——, "Adding Monitoring and Reconfiguration Facilities for Service-Based Applications in the Cloud," in *IEEE 27th International Conference on Advanced Information Networking and Applications*, 2013. 9, 10

[34] R. Baldoni, R. Beraldi, S. Piergiovanni, and A. Virgillito, "Measuring notification loss in publish/-subscribe communication systems." in *Proceedings of the 10th IEEE Pacific Rim International Symposium, Dependable Computing*, march 2004. 10

[35] S. Wang, F. Du, X. Li, Y. Li, and X. Han, "Research on dynamic reconfiguration technology of cloud computing virtual services," in *IEEE International Conference on Cloud Computing and Intelligence Systems*, Sept 2011, pp. 348–352. 10

[36] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing degrees, models, and applications," *ACM Computing Surveys*, vol. 40, no. 3, pp. 7:1–7:28, Aug. 2008. 11

[37] T. Metsch and A. Edmonds, "Open Cloud Computing Interface - Infrastructure," Tech. Rep., 2011. [Online]. Available: http://www.ogf.org/documents/GFD.184.pdf 14, 15, 97

[38] N. Podhorszki, Z. Balaton, and G. Gombás, "Monitoring Message-Passing Parallel Applications in the Grid with GRM and Mercury Monitor," in *Grid Computing*, ser. Lecture Notes in Computer Science, M. Dikaiakos, Ed. Springer Berlin Heidelberg, vol. 3165, pp. 179–181. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28642-4_21 20, 35

[39] H. Huang and L. Wang, "P&P: A Combined Push-Pull Model for Resource Monitoring in Cloud Computing Environment." in *IEEE 3rd International Conference on Cloud Computing* , july 2010, pp. 260–267. 21, 35

[40] S. Andreozzi *et al.*, "GLUE Specification," Tech. Rep., March 2009. [Online]. Available: http://www.ogf.org/documents/GFD.147.pdf 21

[41] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Tech. Rep., 2002. [Online]. Available: http://docencia.ac.upc.edu/dso/papers/ogsa.pdf 22

[42] W. Smith, "A system for monitoring and management of computational grids," in *Proceedings of International Conference on Parallel Processing*, 2002, pp. 55–62. 22, 35

[43] A. Cooke, A. J. G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, and A. Wilson, "R-GMA: An Information Integration System for Grid Monitoring," in *Proceedings of the 11th International Conference on Cooperative Information Systems*, 2003, pp. 462–481. 22, 35

[44] H. B. Newman, I. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, "MonALISA : A Distributed Monitoring Service Architecture," *CoRR*, vol. cs.DC/0306096, 2003. 23, 35

[45] T. Bowker, "Superior app management with JMX," june 2001. [Online]. Available: http://www.javaworld.com/article/2075350/java-web-development/superior-app-management-with-jmx.html 23, 33

[46] N.-C. Pellegrini, "Dynamic reconfiguration of Corba-based applications," in *Proceedings of Technology of Object-Oriented Languages and Systems*, 1999. 24, 35

[47] F. Plasil, D. Balek, and R. Janecek, "SOFA/DCUP: architecture for component trading and dynamic updating," in *Proceedings of Fourth International Conference on Configurable Distributed Systems*, May 1998, pp. 43–51. 25

[48] Y. Li, Y. Lu, Y. Y. Yin, S. Deng, and J. Yin, "Towards QoS-Based Dynamic Reconfiguration of SOA-Based Applications," in *IEEE Asia-Pacific Services Computing Conference*, Dec 2010, pp. 107–114. 26, 35

[49] Y. Zhai, J. Zhang, and K.-J. Lin, "SOA Middleware Support for Service Process Reconfiguration with End-to-End QoS Constraints," in *IEEE International Conference on Web Services*, 2009, pp. 815–822. [Online]. Available: http://dx.doi.org/10.1109/ICWS.2009.126 26, 35

[50] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using Architecture Models for Runtime Adaptability," *IEEE Software Journal*, vol. 23, pp. 62–70, 2006. 26, 35

[51] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - a component system for pervasive computing," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications*, March 2004, pp. 67–76. 27, 35, 38

[52] O. W. crew, "LOOM .NET Project," Tech. Rep. [Online]. Available: https://www.dcl.hpi.uni-potsdam.de/research/loom 28

[53] R. Ribler, J. Vetter, H. Simitci, and D. Reed, "Autopilot: adaptive control of distributed applications," in *Proceedings of The Seventh International Symposium on High Performance Distributed Computing*, Jul 1998, pp. 172–179. 30, 35

[54] R. Rouvoy, D. Conan, and L. Seinturier, "Software Architecture Patterns for a Context-Processing Middleware Framework," *IEEE Distributed Systems Online*, vol. 9, p. 1, june 2008. 33

[55] T. Baker, O. Rana, R. Calinescu, R. Tolosana-Calasanz, and J. Bañares, "Towards Autonomic Cloud Services Engineering via Intention Workflow Model," in *Economics of Grids, Clouds, Systems, and Services*, ser. Lecture Notes in Computer Science, J. Altmann, K. Vanmechelen, and O. Rana, Eds. Springer International Publishing, 2013. 33

[56] J. Diaz-Montes, M. Zou, I. Rodero, and M. Parashar, "Enabling autonomic computing on federated advanced cyberinfrastructures," in *Proceedings of the ACM Cloud and Autonomic Computing Conference*, 2013. [Online]. Available: http://doi.acm.org/10.1145/2494621.2494641 33

[57] M. Hasan, E. Magana, A. Clemm, L. Tucker, and S. Gudreddi, "Integrated and autonomic cloud resource scaling," in *IEEE Network Operations and Management Symposium*, 2012. 33

[58] D. Kurian and P. Chelliah, "An autonomic computing architecture for business applications," in *World Congress on Information and Communication Technologies*, 2012. 33

[59] D. Ionescu, B. Solomon, M. Litoiu, and M. Mihaescu, "A Robust Autonomic Computing Architecture for Server Virtualization," in *International Conference on Intelligent Engineering Systems*, 2008. 33

[60] Z. Zhao, C. Gao, and F. Duan, "A survey on autonomic computing research," in *Asia-Pacific Conference on Computational Intelligence and Industrial Applications*, vol. 2, 2009. 33

[61] G. Martinovic and B. Zoric, "E-health Framework Based on Autonomic Cloud Computing," in *Second International Conference on Cloud and Green Computing*, 2012. 33

[62] D. Bantz, C. Bisdikian, D. Challener, J. Karidis, S. Mastrianni, A. Mohindra, D. Shea, and M. Vanover, "Autonomic personal computing," *IBM Systems Journal*, vol. 42, no. 1, 2003. 33

[63] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley/ACM Press, 2002. 38

[64] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems." *Software Practice and Experience (SP&E)*, vol. 36, pp. 1257–1284, 2006. 38

[65] OSGI, "Open Services Gateway Initiative," 1999. [Online]. Available: http://www.osgi.org 38

[66] S. Yangui, M. Mohamed, S. Tata, and S. Moalla, "Scalable Service Containers." in *Proceedings of the third IEEE International Conference on Cloud Computing Technology and Science*, 2011, pp. 348–356. 47

[67] M. Amziani, T. Melliti, and S. Tata, "Formal Modeling and Evaluation of Stateful Service-Based Business Process Elasticity in the Cloud," in *OTM Conferences On the Move to Meaningful Internet Systems*, ser. Lecture Notes in Computer Science, R. Meersman, H. Panetto, T. Dillon, J. Eder, Z. Bellahsene, N. Ritter, P. De Leenheer, and D. Dou, Eds. Springer Berlin Heidelberg, 2013, vol. 8185, pp. 21–38. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41030-7_3 53, 89

[68] A. Omezzine, S. Yangui, N. Bellamine, and S. Tata, "Mobile Service Micro-containers for Cloud Environments," in *IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2012, pp. 154–160. 53

[69] "OpenNebula." 2011. [Online]. Available: http://opennebula.org 82

[70] "Openstack." 2011. [Online]. Available: http://www.openstack.org 82

[71] "JAVA programming Assistant." 2010. [Online]. Available: http://www.csg.is.titech.ac.jp/~chiba/javassist 82

[72] Oracle, "Java 2 Platform API Specification," 2010. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html 82

[73] M. Sellami, S. Yangui, M. Mohamed, and S. Tata, "Open Cloud Computing Interface - Application," Tech. Rep., 2013. [Online]. Available: http://www-inf.int-evry.fr/SIMBAD/tools/OCCI/occi-application.pdf 86, 94

[74] S. Yangui, M. Mohamed, M. Sellami, and S. Tata, "Open Cloud Computing Interface - Platform," Tech. Rep., 2013. [Online]. Available: http://www-inf.int-evry.fr/SIMBAD/tools/OCCI/occi-platform.pdf 86, 94

[75] "Open Cloud Computing Interface - Platform and Application," 2014. [Online]. Available: http://www-inf.it-sudparis.eu/SIMBAD/tools/OCCI 86

[76] M. Sellami, S. Yangui, M. Mohamed, and S. Tata, "PaaS-Independent Provisioning and Management of Applications in the Cloud," in *IEEE Sixth International Conference on Cloud Computing*, June 2013, pp. 693–700. 87, 94

[77] S. Yangui, I.-J. Marshall, J.-P. Laisne, and S. Tata, "CompatibleOne: The Open Source Cloud Broker," *Journal of Grid Computing*, vol. 12, no. 1, pp. 93–109. [Online]. Available: http://dx.doi.org/10.1007/s10723-013-9285-0 87

[78] L. Richardson and S. Ruby, *Restful Web Services*, 1st ed. O'Reilly, 2007. 87

[79] J. Lautamäki, A. Nieminen, J. Koskinen, T. Aho, T. Mikkonen, and M. Englund, "CoRED: Browser-based Collaborative Real-time Editor for Java Web Applications," in *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, 2012, pp. 1307–1316. [Online]. Available: http://doi.acm.org/10.1145/2145204.2145399 88

[80] E. Hossny, S. Khattab, F. Omara, and H. Hassan, "A Case Study for Deploying Applications on Heterogeneous PaaS Platforms," in *International Conference on Cloud Computing and Big Data*, Dec 2013, pp. 246–253. 88

[81] J. Geelan *et al.*, "Twenty-One Experts Define Cloud Computing," 2009. [Online]. Available: http://virtualization.sys-con.com/node/612375 88

[82] M. Mohamed, S. Yangui, S. Moalla, and S. Tata, "Web Service Micro-Container for Service-based Applications in Cloud Environments," in *IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2011, pp. 61–66. [Online]. Available: http://dx.doi.org/10.1109/WETICE.2011.51 88

[83] A. Alves *et al.*, "Web Services Business Process Execution Language 2.0 ." OASIS, Tech. Rep., 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html 89

[84] Open Management Group, "Business Process Model and Notation (BPMN)," Tech. Rep., 2011. [Online]. Available: http://www.bpmn.org 89

[85] C. A. Petri, "Communication with automata," Ph.D. dissertation, Universität Hamburg, 1966. 89

[86] M. Amziani, T. Melliti, and S. Tata, "Formal Modeling and Evaluation of Service-Based Business Process Elasticity in the Cloud," in *IEEE 22nd International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2013, pp. 284–291. 92

[87] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *IEEE International Conference on Cloud Computing*, July 2011, pp. 716–723. 92

[88] G. Galante and L. de Bona, "A Survey on Cloud Computing Elasticity," in *IEEE International Conference on Utility and Cloud Computing*, 2012, pp. 263–270. 92

[89] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically Scaling Applications in the Cloud," *SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925861.1925869 92

[90] Cloud Foundry Blog, "Monitoring Cloud Foundry Applications with New Relic," 2014. [Online]. Available: http://blog.cloudfoundry.com/2013/10/10/monitoring-cloud-foundry-applications-with-new-relic 96, 102

[91] D. van Thanh and I. Jrstad, "A service-oriented architecture framework for mobile services," in *Proceedings of the advanced industrial conference on telecommunications/service assurance with partial and intermittent resources conference/e-learning on telecommunications workshop*, July 2005, pp. 65–70. 107