Roboconf: an Autonomic Platform Supporting Multi-level Fine-grained Elasticity of Complex Applications on the Cloud

Manh Linh Pham

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministérial : 7 Août 2006

Présentée par

Pham Manh Linh

Thèse dirigée par **Prof. Noël de Palma**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Roboconf: une Plateforme Autonomique pour l'Élasticité Multi-niveau, Multi-granularité pour les Applications Complexes dans le Cloud

Roboconf: an Autonomic Platform Supporting Multi-level Fine-grained Elasticity of Complex Applications on the Cloud

Thèse soutenue publiquement le **04 february 2016**,
devant le jury composé de :

Prof. Noël de Palma
Université Grenoble Alpes, Grenoble, Directeur de thèse
Prof. Didier Donsez
Université Grenoble Alpes, Grenoble, Co-Directeur de thèse
Prof. Françoise Baude
Université de Nice - Sophia Antipolis, Nice, Rapporteur
Prof. Daniel Hagimont
INPT/ENSEEIHT, Toulouse, Rapporteur
Asc. Prof. Vania Marangozova-Martin
Université Grenoble Alpes, Grenoble, Présidente
Asc. Prof. Alain Tchana
INPT/ENSEEIHT, Toulouse, Examinateur

Abstract

Software applications are becoming more diverse and complex. With the stormy development of Cloud computing and its applications, software applications become even more complex than ever. The complex cloud applications may contain a lot of software components that require and consume a large amount of resources (hardware or other software components) distributed into multiple levels based on granularity of these resources. Moreover these software components might be located on different clouds. The software components and their required resources of a cloud application have complex relationships which some could be resolved at design time but some are required to tackle at runtime. Elasticity is one of benefits of Cloud computing, which is capability of a cloud system to adapt to workload changes by adjusting resource capacity in an autonomic manner. Hence, the available resources fit the current demand as closely as possible at each point in time. The complexity of software and heterogeneity of cloud environment become challenges that current elasticity solutions need to find appropriate answers to resolve. In this thesis, we propose a novel elasticity approach as an efficient solution which not only reflects the complexity of cloud applications but also deploy and manage them in an autonomic manner. It is called multi-level fine-grained elasticity which includes two aspects of application's complexity: the multiple software components and the granularity of resources. The multi-level fine-grained elasticity concerns objects impacted by elasticity actions and granularity of these actions. In this thesis, we also introduce Roboconf platform, an autonomic Cloud computing system (ACCS), to install and reconfigure the complex applications as well as support the multi-level fine-grained elasticity. To this end, Roboconf is also an autonomic elasticity manager. Thanks to this platform, we can abstract the complex cloud applications as well as automate their installation and reconfiguration that can consume up to several hundred hours of labour. We also use Roboconf to implement the algorithms of multi-level fine-grained elasticity on these applications. The conducted experiments not only indicate efficiency of the multi-level fine-grained elasticity but also validate features supporting this approach of Roboconf platform.

Keywords. Cloud computing; autonomic computing; Cloud deployment and reconfiguration; multi-level fine-grained elasticity.

Résumé

Les applications logicielles sont de plus en plus diversifiées et complexes. Avec le développement rapide du cloud computing et de ses applications, les logiciels deviennent plus complexes que jamais. Ces applications peuvent contenir un grand nombre de composants logiciels qui nécessitent et consomment une grande quantité de ressources (matérielles ou d'autres composants logiciels) réparties sur plusieurs niveaux en fonction de la granularité de ces ressources. En outre, ces composants logiciels peuvent être localiss sur différents clouds. Les composants logiciels et les ressources requises d'une application ont des dépendances complexes. Certaines pourraient être résolues lors de la conception, tandis que d'autres doivent être traitées à l'exécution. L'élasticité est l'un des avantages du cloud computing. C'est la capacité d'un système a s'adapter la charge de travail en ajustant ses ressources de manière autonome. Ainsi, les ressources correspondent au besoin en tout instant. La complexité des logiciels et l'hétérogénéité des environnements de type cloud sont des défis auxquels les solutions d'élasticité actuelles doivent faire face. Dans cette thèse, nous présentons Roboconf, un système autonome de cloud computing (ACCS) qui permet le déploiement, installation et gestion autonomes d'applications complexes dans le cloud. En utilisant Roboconf, nous avons implémentés plusieurs algorithmes d'élasticité à niveaux multiples et à grain fin qui prennent en compte les relations entre les composants logiciels et la granularité des ressources. L'évaluation menée permet non seulement de montrer l'efficacité de notre approche d'élasticité mais aussi de valider les fonctionnalités de Roboconf qui lui sont sous-jacentes.

Mots-clés. cloud computing, calcul autonome, déploiement cloud, reconfiguration cloud, élasticité à niveaux multiples et à grain fin.

Acknowledgments

"Some people grumble that roses have thorns; I am grateful that thorns have roses."

–Alphonse Karr, A Tour Round My Garden

This work would have been impossible if it was not dedicated to a number of people to whom I am greatly indebted.

From the deepest of my heart, I would like to give million thanks to my parents. I did not study the best school in the city. I did not attend the better university in the state. I did not get the best job in the country. But none of this matters because I have the best parents in the world and they made up for all these things. Without their love, inspiration, drive and support, I might not be the person I am today. Thank you mom and dad.

I wish to thank sincerely my supervisor, Noël de Palma, for his endurance, patience, inspiration and great discussions. He has created a very unique positive research environment that is rare to find anywhere else. I would also like to thank my co-supervisor Didier Donsez for the hard work, the great ideas, the long discussions, and the great feedback. Noël and Didier are not just supervisors, they are mentors, teachers, and above all friends. It has been a privilege working with both of you. Your positive impact will stay with me for the rest of my life.

I would like to thank Asc. Prof. Vania Marangozova-Martin for accepting to be the President of the jury committee of my defense. I also would like to thank Prof. Daniel Hagimont and Prof. Françoise Baude to become the reviewers of my thesis. Many thanks to Asc. Prof. Alain Tchana to become the member of the jury of my defense. I really appreciate your reports, advices as well as your precious feedback on my work.

I also would like to thank the Vietnamese Government and the Vietnam - France University (USTH) for giving me the grant to work in France.

A very big thank to awesome colleagues in the ERODS team and to friends in the LIG laboratory. Thank to Alain for helping me settle in the very first days and still coaching me then. Thank to Ahmed for answering every single of my naive questions. Thank to all the coffee sessions in the afternoon and to all Vietnamese

friends in Grenoble. You are the most beautiful memory I have during my stay in France.

On a more personal level, I fell in love with a girl long time before I started my PhD studies. She becomes my wife 2 years after I started my PhD work. Diem, thanks for being my leash when I was running too fast and my push when I was moving too slow, thanks for being my anchor when I was sinking too deep and my rope when I was flying too high. Last but not least, I would like to thank my future little prince. He is the most precious thing I am waiting for and the main source of joy in life!

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Contents

1.1 Context

"The cloud is for everyone. The cloud is a democracy."

–Marc Benioff, CEO - Salesforce.com

This thesis has been written in the era when Cloud computing has been recognized around the world. Cloud computing has a magical glamour and everybody is talking about it everywhere. One of Cloud computing promising characteristics is ability to provision virtually computational resources on demand. Cloud consumers utilize the resources provided by service providers in a pay-as-you-go style. It means the consumers pay only for the resources they actually used (e.g. CPU, memory, bandwidth, storage) to resolve their specific problems, not for entire IT system. This not only increases revenue for cloud providers but also decrease costs of cloud consumers. A recent survey of more than 930 IT professionals and decision-makers globally by Right Scale [1] revealed that cloud adoption rate in 2015 continues to rise, as 75% of respondents cited they are using at least one cloud platform and 15% are considering move to the Cloud. This growing number, up from 14% and 10% of the companies in 2010 respectively,

suggests that the Cloud is having a magnificent impression on the business world, encouraging CTOs to implement the technology in order to not only cut the IT costs but also help the companies focusing on their core businesses. Although there remains some doubts about insecurity, not interoperability, limited control or vendor lock-in issues of Cloud computing, this computing model is still full promising and plays an important role in reducing the initial investment for IT infrastructures.

The history of Cloud Computing back to the sixtieth when Licklider, who was responsible for development of ARPANET (Advanced Research Projects Agency Network, later became the Internet), introduced an "intergalactic computer network" in 1969 [2]. His vision was for everyone on the world to be connected and accessing programs and data from anywhere at any sites, that much like what we are calling "Cloud computing" nowadays. Since then, Cloud computing has evolved through a number of phases including grid and utility computing. In the ninetieth, explosion of the Internet and Web technologies are important premises for flying up of the Cloud in early 21st century. Other key factors which have contributed to evolution of the Cloud include the development of high-speed bandwidth, the standardization of software interoperability and especially the maturing of virtualization technology. First milestone was made by Salesforce [3] in 1999, which promoted the concept of SaaS (Software-as-a-Service) by delivering enterprise applications to end users with a simple website. The next poke was from Amazon Web Services [4] in 2002, which provided a suite of cloud-based services such as computation, storage and artificial intelligence. These services were commercialized in 2006 by launching of Elastic Compute cloud (EC2) [5] that allows individuals and enterprises to rent virtual resources on which to run their own applications. Amazon EC2/S3 was recognized as the first widely accessible infrastructure service on the Cloud. As Web 2.0 reached its maturity in 2009, Google offered browser-based enterprise applications through its web service which is Google App Engine [6]. Thenceforth, Cloud computing market is stirred continuously and incessantly by the technology giants such as Microsoft, IBM, HP, VMware, etc.

Software applications also evolve along with the growth of Cloud computing, from simple software programs running on a single machine in those days before the era of Internet to very complex distributed applications spanning multiple servers on different sites in recent days. Cloud computing has changed the way people develop and use software, the regular programs have transformed into services available on the Internet. A simple example is the evolution of Microsoft Office, a suite of office applications has leveraged premium features of Cloud computing to become Office 365, a service enables Microsoft Office on the Cloud. In this context, traditional applications (i.e. legacies) must be ready for modifications to be migrated to the Cloud. For many decades until now, we are seeing the

continuous growth of the complexity of applications, due to the development of new technologies on the one hand, and the emergence of new needs on the other hand. An application does not address a single problem but several. This growth of their complexity implies the same phenomenon regarding their execution environment (organization of physical machines or devices on which they run). For instance, this has brought forward a change from centralized to distributed and heterogeneous places of execution. All of this make human administration very difficult because they are errors prone, slow to respond (e.g. fault solving), and highly costly (e.g. wages).

The habits in the field of software development is also changing due to the increasing complexity of applications. Nowadays, the software developers cannot stand alone, they have to combine with testers and system administrators who must stick together in the same software life-cycle loop to ensure minimum software bugs and still have to satisfy the most demanding customers. This raises a new job called DevOps [149] where software developers are also the sysadmin who understand thoroughly the application from development, staging to production phases. This characteristic of agile software development also set the software product on an endlessly autonomous loop from development to customer delivery, named Continuous Delivery. This software engineering approach ensures producing of applications in short cycle and releasing of software product reliably at any time [27]. The appearance of the new software development trends requires software manufacturers have to modernize themselves by increasing levels of automation in the production process.

When deploying cloud services or applications, cloud providers and users both want to maximum benefits brought from Cloud computing. One of advantage is the efficient use of resources to support scalability. This consequently leads to saving in cost, energy and labour. The cloud infrastructure providers must try to optimize their infrastructures to save energy and provide on-demand resources. The cloud platform providers try to integrate advanced features into their platforms, which adjust capacity of resources so that it meets resource demand of workload fluctuation as closely as possible. The cloud application providers optimize management of applications to request for just-enough resources. To be able to do this, the process of application and resource management in the Cloud must be automated as much as possible to minimize repetition and risks from human errors. In this context, many cloud tools have been developed and introduced to realize this need, but it is still not enough.

In the early 2000s, IBM [7] proposed to automate the administration of complex applications throughout the use of what we called Autonomic Computing Systems (ACS for short). This practice consists in transferring human administration knowledge and behaviours to an ACS, which can be done in two ways: either by introducing autonomic behaviors into application components at its de-

velopment time [8] or by building a computing system (different to the application we want to administrate) which will make the application autonomous [9]. The autonomous applications own self-managing characteristics of distributed computing resources, which help them adapting to unpredictable changes and hiding complexity to sysadmins or users. However, implementation of cloud applications is a challenging domain for existing ACSs, as it introduces one or multiple intermediate levels of virtualization such as virtual machines (VM), lightweight containers and their various combinations. Moreover, it sometimes requires the use of several clouds at once (hybrid and multicloud). For example, running a financial/bank application within the cloud generally requires two clouds: a private cloud (located in the company to which the application belongs) to run business-critical part, and a public cloud (e.g. Amazon EC2) to run non-critical part of the application. Note that in some situations, the non-critical part can move from one cloud to another for price and competitiveness reasons. To make matter worse, cloud APIs are not standardized, which results in non interoperable clouds. In next section, existing problems of automating implementation of cloud applications using ACS are further discussed and pointed out.

1.2 Motivation

ACS solutions seem to be attractive for deployment and management of legacy applications. However, there are still various challenges when a company decides to move to the Cloud and apply ACS to automate the installation and management of their complex application. In this context, existing ACSs [10, 11, 12] are inappropriate for several reasons:

- Existing ACSs only consider one level of deployment/execution: an application runs within a physical machine (PM), whereas in the context of Cloud, the application often runs within a VM. The introduction of container-based virtualization, a technology has been mentioned since decades but has only been popular recently, not only brings forward the opportunities, but also put much complexity to the automation of deployment process. Nowadays, a container (called lightweight container from now on) hosting a service can run on bare metal, inside a VM or even reversely, a VM runs inside a container. The complexity increases strongly when these virtualized layers combine together (e.g. lightweight container on VM, software component on VM, software component on container, etc.).

- The target execution environment is not static in the context of Cloud, an application does not stay within the same clouds during its overall lifetime.

It can stay in a private cloud in the staging phase and then move to a public cloud in production phase. It can span multiple clouds and blur their limitations.

- Existing ACSs are built to administrate the whole environment (application and execution environment) while in the context of Cloud, administration is ensured by two actors (the cloud application provider administrates its application while the cloud provider is responsible for infrastructure).

Although generic ACSs [13] for grids and clusters of machines exist, their enhancement for clouds requires a high expertise for the deployer. Concerning cloud solutions, the Autonomic Cloud Computing Systems (ACCS for short) [6, 14, 15, 16, 17, 18] are either proprietary, devote to a specific application, or target a static cloud.

As summarized by [7], administration tasks can be divided into the following categories: installation or initial provisioning, reconfiguration and uninstallation. The former includes the description of application components and its runtime, the initialization of the execution environment, the installation of artifacts from repositories, the configuration of the application, and its start-up. About reconfiguration tasks, they are performed at runtime in order to reconfigure the application when a particular situation is detected (e.g. fault). It is response of the managed system to react properly to fluctuations of surrounding environment. It can be done manually by human conscious intervention or automatically by autonomous engine. Lastly, the uninstallation is to gracefully clean an application which is no longer necessary, outdated or damaged. The ACSs have proved their usefulness in all administrative tasks and now the major part of research in this topic focuses on the reconfiguration tasks [19].

The automation of reconfiguration is really challenging because it relates to a chain of concerns which include monitoring managed elements, gathering measured parameters, analyzing collected data, planning for reactions, executing plans and tracking feedbacks. In those challenges, planning for distributing and provisioning virtual resources to adapt the changes of environment, especially workloads, has extremely important implication. It promotes elasticity of the Cloud, which scales applications to ensure distributed resources to fit actual demands as closely as possible. In other words, applications on the Cloud should be elastic, i.e., they should be able to integrate or remove resources on-demand and automatically as well as use them efficiently to handle fluctuations of workloads.

Coming back to broadly recognized definition of Cloud computing of National Institute of Standards and Technology (NIST) in their publication "The NIST Definition of Cloud Computing" [20]: "Cloud computing is a model for enabling 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**."

With this definition, five essential characteristics have been defined that every true cloud computing implementation should have: on-demand self service, broad network access, resource pooling, rapid elasticity, and measured service. Among them, it is necessary to come back and resolve one of basic characteristics and requirements of Cloud computing: rapid elasticity which is challenging the existing ACCSs. Many elasticity solutions based on principles of Autonomic computing but only a couple of research paid proper attention to its "rapid" aspect. Why "rapid" is very important in terms of elasticity? For instance, in Event Stream Processing (ESP) applications, allocating and deallocating VMs can be expensive and thus further affects the system performance. In a research about elasticity on the Cloud [21], author proposed architectures using dedicated VM for each instance, which has a relative high (de)allocation cost compared to alternatives such as lightweight container and processes. VM provisioning on cloud platforms is a slow process on the order of minute. How can this be used for dynamic scaling in stream processing? In this situation, we need to consider complexity of applications as well as classify resources required by application into different levels based on resource granularity. An elasticity action which impacts a more fine-grained resource benefits elasticity with rapid provisioning in terms of reducing Service-level Agreement (SLA) violation rate and increasing resource exploitation ratio.

Classification of resources leads to provisioning more fine-grained resources in the elasticity actions. The ACSs do not have to deal with coarse-grained fatty VMs which needs a couple of minutes for its startup and may cause unnecessary over-provisioning or postpone scaling. With scaling in/out or horizontal scaling, they can instead use other fine-grained resources like lightweight container or software container. With scaling up/down or vertical scaling, adding and removing fined-grained resources like CPU, memory, storage device is a possible approach. We believe that this approach supported by an ACCS will help to obtain the truly rapid elasticity. This is also the motivation behind the work of this thesis.

1.3 Contribution

Provided the presented context, our contribution in this PhD work consists in the design, development and evaluation:

- (1) an ACS-based cloud platform, called Roboconf, for description, deployment and self-management of complex applications on multicloud. This

system is open source [1] and was introduced in our conference publications [22, 23].

- (2) a novel elasticity approach, namely multi-level fine-grained elasticity, and its algorithms to obtain rapid elasticity for cloud applications in an autonomic manner using Roboconf as the main elasticity manager.

With (1), we made following specific contributions with Roboconf:

- Introduce an application model of Roboconf and a hierarchical domain specific language (i.e. DSL) which allow description of cloud applications and their runtimes. This DSL includes a dedicated part for rules of elasticity.
- Detail architecture of Roboconf platform, deployment and reconfiguration process as well as its autonomic loop used for elasticity.
- Perform several experiments which validate all features of Roboconf and assess its overhead. We deploy a web application onto a hybrid cloud (a private VMware hosting center combined with Amazon EC2 and Microsoft Azure [24] clouds). We also deploy a smart home application (openHAB [25]) in embedded boards (BeagleBone Black) and VMs on EC2 and OpenStack [26] clouds. As a consequence, this work has led to a journal publication [130].

With (2), specific contributions are as follows:

- We define and propose algorithms for multi-level fine-grained elasticity approach which can be described under Roboconf DSL for elasticity.
- We use Roboconf platform to deploy RUBiS application [141] and apply the multi-level fine-grained elasticity to this application for evaluation.

1.4 Document Organization

The thesis consists of three main parts: Background and State-of-the-art (from Chapter 2 to 3) which describes and reviews all technologies related to our work which are Cloud computing, Autonomic computing and Elasticity; Contributions (from Chapter 4 to 6) which details our proposals; and Evaluation (from Chapter 7 to 8) which evaluates the proposals.

In Chapter 2, we review theory of Cloud computing and Autonomic computing. With Cloud computing, its definition, characteristics, models, stakeholders as

[1] http://roboconf.net/en/index.html

well as virtualization technologies are presented. With Autonomic computing, we presents its definition, the different phases of the MAPE-K autonomic loop, the properties an autonomic manager must have as well as the combination between Cloud computing and Autonomic computing.

Chapter 3 brings a complete picture about elasticity: its definition, classification accompanying a comprehensive summary of the state-of-the-arts and new issues that remains unsolved.

Our contributions begin from Chapter 4 with introduction of multi-level fine-grained elasticity approach. Chapter 5 dedicates to Roboconf hierarchical DSL and its application model. Architecture and principles supporting the multi-level fine-grained elasticity of Roboconf platform are detailed in Chapter 6.

Chapter 7 proposes algorithms for multi-level fine-grained approach which aim to obtain rapid elasticity. Two experiments to validate the algorithms are included.

Chapter 8 presents some experiments to validate all the features supporting the multi-level fine-grained elasticity of Roboconf. Several use cases of Roboconf used in practice are also mentioned.

Finally, at the end of the thesis in Chapter 9, we resume our approach, emphasize the positive achievements harvested and open up some potential research perspectives which can be built up from this work.

Chapter 2

CLOUD COMPUTING AND AUTONOMIC COMPUTING

Contents

2.1 Theory of Cloud Computing

During the last decade, Cloud computing emerges as a next evolution of Grid computing, which driven originally by economic needs. By providing on-demand computing resources, Cloud computing model allows more efficient utilization of resources in traditional datacenter as well as significant reducing the infrastructure investment cost. Like Grid computing, Cloud computing model is also very

attractive for scientific community with many promising research areas to be explored [39]. While Grid computing is mainly used on scientific research, Cloud computing has passed beyond this community and broadly used for commercial systems. Some of the characteristics that helps Cloud computing to admire the commercial users is its user-friendliness and on-demand scalability [40]. On the one hand, most grid systems are to handle applications with batch jobs. On the other hand, the Cloud supports more types of application including web services, data-processing applications and batch systems as well.

There are a lot of Cloud computing definitions such as [40, 41, 42], but they all do not reach an agreement. Authors in [40] defines Cloud computing as a realization of utility computing, while others [41, 42] consider the Cloud according to the online services it provides. A massive datacenter behind is the factor that some authors [43] argue it makes the main difference to make up the Cloud computing concept. Definition used in this thesis comes from the NIST definition [20]. In this document, they define Cloud computing as a resource sharing model that offers ubiquitous, convenient, on-demand network access to a pool of configurable computing resources. With minimum cloud provider intervention and management efforts, the computing resources are expected to be provisioned and released efficiently and rapidly. There are many advances in technology that contributes to building a cloud system. Advances in server power management [44], virtualization techniques [45], network bandwidth [46] are some key technologies among them. Commonly, cloud customer leases virtual resources to run their services, applications, computation jobs or to store their data. They are then glad to pay for amount of the leased resources or amount of time the resources had been really occupied. The most often, the resources are utilized to launch and maintain a web services which enable accessibility to multi-tenant. Airbnb, one of the new stars in homestay lodging and Netflix, the giant in streaming media, entrust their services on virtual resources of Amazon EC2. The reason is very clear and simple, their core businesses are not about IT infrastructure. By delegating the IT infrastructure for cloud providers, the former can focus on renting houses and the latter can sell streaming movies which are their favorite.

2.1.1 Cloud Computing Characteristics

There are five Cloud computing characteristics pointed out by NIST [20]. However this is not an exhaustive list. Discussed below is characteristics that we believe they are the more important ones when talking about the Cloud.

On-demand Provisioning

It is definitely the most important thing of Cloud computing. Without the human interaction with cloud provider, virtual resources are provisioned on-demand whenever Cloud users need and they only have to pay for the amount of resources

or slot of time they actually used. The promotion for pay-as-you-go style is indeed not new in public utility. Paying for amount of electrical kilowatt per hour or gallons of water has been carried out for a long time.

Broad Network Access

One of charming characteristics of Cloud computing is to allow access universally from various kinds of standard client devices ranging from smartphones, workstations to super-duper computing servers. Provided by a common communication protocol such as Internet or RESTful web services, cloud users can use more and more cloud-based popular services such as cloud storage, imaging sharing, social networking with any kinds of terminal devices.

Resource Pooling

A pooling of physical or virtual resources such as memory, storage, processing, network bandwidth can be assigned or reassigned dynamically using multi-tenant model according to fluctuation of demand. The resources are located at places beyond recognition and control of users. However, abstracted locations are usually provided for monitoring by cloud providers.

Rapid Elasticity

Capacity of resources can be provisioned or deprovisioned to cope with changes of workload. These resources are usually unlimited and purchasable with any quantity at any time. The scaling capacity should be automated and programmed carefully to provide just-enough elastic that avoids oscillation in resource allocation. Some authors define elasticity according to granularity of usage accounting [48]. When load declines, elasticity is not only a function of deprovisioning speed, but also depends on whether charging for the released resource stops immediately or hang on for a while.

Measured Service

Active measurement should be performed transparently and reported to both cloud users and providers. Measured services help users to manage efficiently their budget because the reported figures tell them know how much they have to pay for rented resources. At the provider site, a carefully monitored system brings forward valuable information about the waste of system resources helping them cut costs resulting in better marginal revenues.

Other Characteristics

Beside five characteristics mentioned officially in the NIST document. We believe that two following characteristics are essential for Cloud computing model.

- The Quality-of-Service (QoS) must be provided for the customers to meet the desired SLA. The service elements need to be quantified clearly using automatically metering tools.

- As the cloud systems are often built on inexpensive commodity hardware, the cloud infrastructure behavior suffers from sporadic faults and is often

non-uniform. Thus transparent fault tolerance mechanisms need to be developed to report failures caused from the customers.

2.1.2 Cloud Computing Models

The traditional classifications categorize cloud systems according to the service or deployment models. In the service models, type of services provided by cloud providers are described. Whereas, how a cloud service is implemented on the actual infrastructure is detailed on the deployment models.

Cloud computing service models Three main service models often named according to type of services they provide are Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

- Infrastructure-as-a-Service model, where cloud providers lease raw computing resources located at the lowest level of abstraction of the Cloud. Users specify requirements for their application in terms of the raw computing resources which can be either physical (i.e. Hardware-as-a-Service) or virtual thanks to virtualization technology. The permission for controlling and distribution of the cloud infrastructure belongs to the cloud providers. The users are responsible for implementing desired operating systems, programming frameworks, applications and other required software. Rackspace [47] and Amazon EC2 who own big data centers providing IaaS platforms are examples about IaaS providers.

- Platform-as-a-Service model, where cloud providers maintain and offer their platforms for targeted cloud users. The platform may provide from programming language, libraries, supported services, tools to execution environments. The PaaS provider, themselves, can build their infrastructure or rent one from an IaaS provider. The users use the platform to develop, release and/or deploy their own applications. Window Azure and Google App Engine are two of key players in this particular sector.

- Software-as-a-Service model, where cloud applications are offered as is and usually in multi-tenant mode. These applications are developed and provided by SaaS provider. The users do not have any controls to infrastructure or platform, exempt the service utilization. Salesforce and Dropbox [49] are typical examples for this kind of model.

Many other XaaS terms are used nowadays to name different provided services in the Cloud. Ben-Yehuda and her colleagues introduce the term Resource-as-a-Service in [52] which foresees that resources will be packed and sold in more fine-grained form such as CPU, memory, bandwidth network rather than fixing them in

a virtual machine. While Zhu et al. suggests adding three more XaaS for service models which are Data-as-a-Service (DaaS), Identity and Policy Management-as-a-Service (IPMaaS) and Network-as-a-Service [54]. A full taxonomy of XaaS can be found in papers [53, 124], such as Analytics as a Service (AaaS) or Framework as a Service (FaaS).

Cloud computing deployment models The deployment models are all about how and where the services are deployed on Cloud. They also tell us who can access resources from those models. According to NIST, the four main deployment models are as follows.

- Private cloud are used and managed by an organization who wish their resources to be accessed only for entities within their jurisdiction. All sensitive data are kept internal, thus offering a higher level of security. Private cloud is a good choice for companies who still concern about data safety on the Cloud. It is a fact that the NSA (USA National Security Agency) is using a private cloud [50].

- Community clouds are shared among organizations who have common concerns. Thus access to these clouds belongs to these organizations. The North Carolina Research and Education Network (NCREN) are an example about using community cloud in education [51].

- Public clouds lease computing resources and make them available to the general public or a large industry group and owned by an organization selling cloud services. Typically, the resources are shared among cloud customers. Salesforce, Amazon EC2, Google App Engine are all about public clouds.

- Hybrid clouds are combination of two or more clouds (private, community, or public) that enable data and application portability as well as remain unique entities bound by standardized or proprietary technologies. Sometime one cloud is not enough, a private cloud may not provide enough resources or a public cloud may be less secure. With hybrid cloud model, a cloud can expand its capacity by using resources from other clouds which may be in different kinds. In some cases, load balancing can be implemented on this cloud model. Plenty of hybrid cloud solutions are offered by popular names, some of them are Eucalyptus, HP hybrid Cloud Management and VMware vCloud Hybrid Service.

2.1.3 Actors in Cloud Ecosystem

There are many stakeholders participating in cloud ecosystem. In this section, we only define main actors playing important roles and these definitions are used

from now on throughout the thesis.

Cloud Provider (**CP**): is a company owning large data centers along with dedicated software systems that enable it to lease resources on-demand over the Internet and get back pay-per-use fees. A CP can be either an IaaS or PaaS provider. A PaaS provider can implement its own underlying infrastructure or rent infrastructure from one or several IaaS providers. In either ways they deliver hardware and software tools usually needed for application development.

Cloud Application Provider (**CAP**): is a stakeholder that rent the on-demand resources provided from the CP to develop and build its applications and services. The CAP then sell these services or applications to its customers to get pay-per-use fee. The customers could also pay a monthly/annual subscription fee to be used the applications. An CAP can be referred to as a SaaS provider or a CP User (**CP-U**). In this thesis, in some cases, we may use the term CP-U instead of CAP.

Cloud Application User (**CAU**) or common name "Cloud end user": It is the customer of the CAP who is willing to pay a pay-per-use fee for using the applications or services provided by the CAP. In this thesis, the terms CAU and Cloud end user may also be used interchangeably.

Figure 2.1 illustrates the three actors of cloud ecosystem, which are mainly used in the specific context of this thesis.

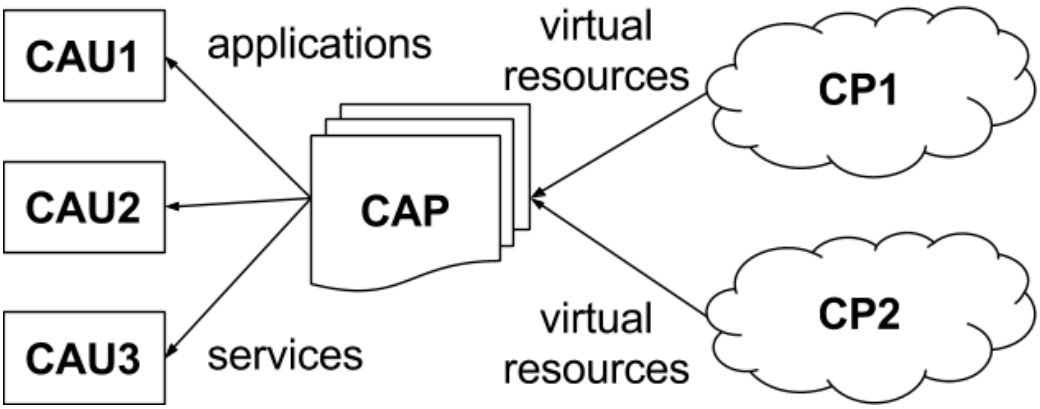

Figure 2.1: The relations among the actors in cloud ecosystem

2.1.4 Virtualization on the Cloud

Virtualization is a prerequisite which wings for development of the Cloud. By virtualization, hardware resources are abstracted and shared broadly on the Cloud. Many cloud users can use virtual resources which might be abstracted from the

same set of hardware resources. Core principle of virtualization is to create as many separating operating systems as possible on a given set of bare-metal hardware. To manage virtualization, it needs a management software in order to separate operating systems and virtualize hardware resources. The underlying hardware often provides architectural support for virtualization. There are many technologies coexist supporting two main types of virtualization: VM-based and Container-based virtualization.

VM-based virtualization uses Virtual Machine Monitor (VMM) or hypervisor to emulate hardware and make communication among host OS and its guest OSes. As the kernel of host OS is not shared, each VM must be installed an entire guest OS of itself, which quickly exhausts resources on the server such as RAM, CPU and bandwidth. This creates significant overhead because instructors must be translated by the VMM. The guest OSes also occupy resources of the host. However VM-based virtualization ensures a reliable isolation among VMs when wrapping the guest OSes in the separated VMs. This guarantees the isolation in both resource and performance aspects. Resource of different VMs cannot be utilized mutually and thus avoid the resource abuse which degrades performance of applications installed on them. There are two types of VM hypervisor demonstrated in Figure 2.2.

Type 1 or bare-metal hypervisor: runs directly on the bare metal hardware. The advantage of this type is the guest OS can communicate with hardware through only the hypervisor without existing of a host OS in between. This reduces the overhead caused by host OS level. However it is difficult to implement the Type 1 hypervisor on an existing and running system because it requires format and partitioning while installation. The Citrix XenServer and VMware ESX are examples of Type 1 hypervisors.

Type 2 or hosted hypervisor: The hypervisor runs on a host OS thus increase a level of virtualization overhead. However the Type 2 is easy to implement on a existing and running system as a tradeoff. Some examples of the Type 2 hypervisors include Oracle open-source VirtualBox and VMware Server.

Container-based virtualization: uses a container manager to effectively virtualized the host OS. Therefore it does not require a full-fledged guest operating system running on the VM. The containers running on the same operating system have their own abstracted networking layer, processes, devices and do not know they are sharing resources. In other words, containers are executed in different spaces that are isolated and from certain regions of the host OS. Containers exist on the host OS just lightweight as processes. Docker and LXC are two of technologies supporting container-based virtualization.

Utilization of such lightweight containers brings to some benefits. First, resource utilization is much more efficient. If a container does not execute anything, it also does not consume resources and these resources can be released for other

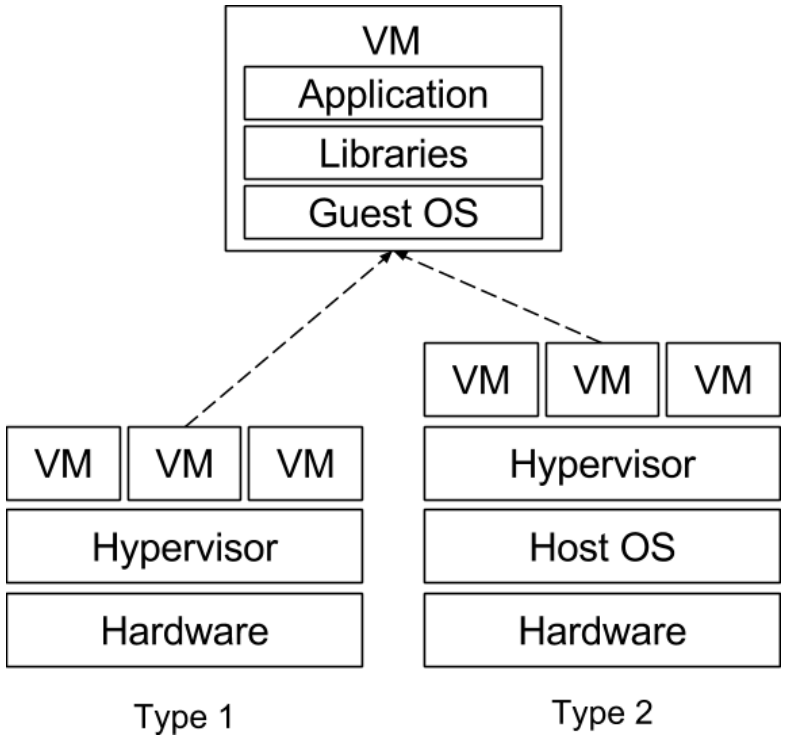

Figure 2.2: Type 1 and Type 2 VM-based hypervisor

containers. Secondly, it does not require a guest OS bootup or shutdown, thus creation and termination of containers is just as fast as creating or killing application processes. With the same reason that the containers do not have a guest OS and containers share the host OS kernel and many libraries and basic commands, size of images created from containers are smaller than VM images (VMIs). In summary, containers are cheap with near native creating speed. Figure 2.3 depicts the container-based virtualization.

Beside the advantages, container-based virtualization also has drawbacks. Sharing the host OS kernel implies that containers in the same physical node must have the same execution environment. It means we cannot have Window containers running on the Linux host OS and vice versa. Containers are not suitable for permanent data storage as they are easy to volatile, thus storage data used by containers need other advanced techniques.

Like Type 1 VM-based virtualization, container on bare metal is feasible but it is still in its infancy. Some technologies having announced to early support this type of virtualization are Triton Elastic Container of Joyent [1] and LXD of

[1] https://www.joyent.com/

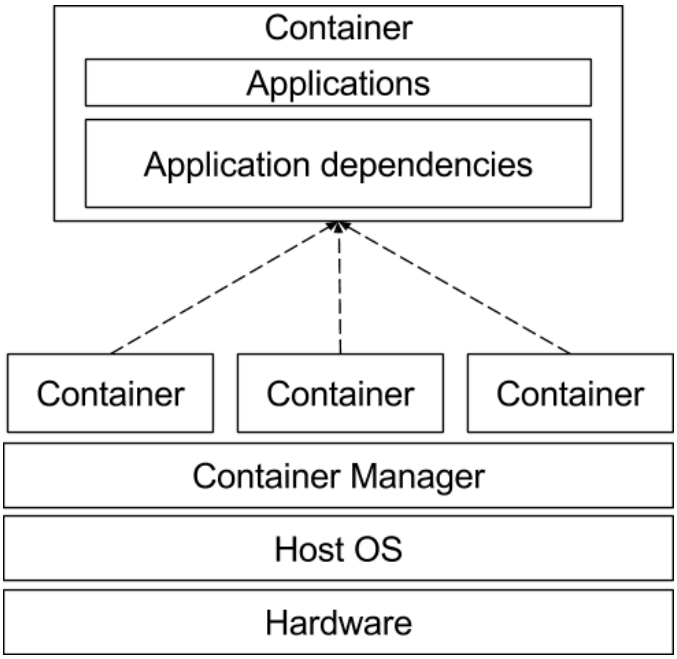

Figure 2.3: Container-based Virtualization

Ubuntu [2].

Paravirtualization [144]: is a form of virtualization which virtualizes a part of hardware and leaves remains for the host OS that provides some API to the images to be deployed there. This may significantly shrink the size of the OS kernel image in the VMIs but still larger than container-based approach. Those kernels should be slightly modified and adapted. Another good point is different VMs may use different kinds of OS in the same host.

2.1.5 Challenges

As a young and promising technology, Cloud computing have to face with many challenges to be mature. These challenges are discussed thoroughly by Armbrust et al. in [139]. One of very first challenge when a company decides to bring its application to the Cloud is how to make it be ready with the new environment, especially with the legacy applications. At that time, the company has to make an important decision to choose what cloud providers for managing its application. As the lack of common standards on the Cloud, the company possibly has to develop or modify its application following a proprietary archetype of a specific

[2]http://www.ubuntu.com/cloud/lxd

cloud provider. Thus the company will most likely run into the vendor lock-in problem and have to stick with the cloud provider, even when it is not satisfied with the service it received. As soon as the company chooses a cloud provider, it wants the application to be deployed and be monitored by automatic tools. These tools should ensure the application be available with the rate regulated in the SLA. The availability can be done by some methods such as using the load-balancing technique or hybrid cloud. However implementation on hybrid cloud will face obstacles coming from the heterogeneity of the Cloud. Security should be concerned from very beginning because it is the main worry of companies when they decide to move to the Cloud. An insecure cloud solution might lose the customers' confidential and sensitive information at the hand of exploited attackers and competitors. Thus data on the Cloud need to be encrypted although encryption might degrade the performance. Admission control and identity management should be other ways to fasten security on the Cloud. Scalability also needs to be taken into account to have the application still available even when demand of resource increases. A scalable cloud system will wing for elasticity which helps the system adapt and fit with the changes of workload as closely as possible. To be adaptive, rented resources should be used in optimal way because they are not really infinite, especially with cloud users who have to pay for them. Elasticity solutions should be rapid enough in terms of both speed and accuracy of elasticity actions. As mentioned, this is the motivation of this thesis and will be presented further in Chapter 4.

2.2 Theory of Autonomic Computing

2.2.1 Definition

It is an indisputable fact that computing systems are increasingly complex. Their complexity is beyond the handling of normal administration tasks. The management and maintenance of a large computing system soon become boring for even the most experienced professionals. It is worth noting that human often make mistakes when repeating tedious tasks. This is the most basic reason for IBM to propose a biologically-inspired computational model called Autonomic computing to automate the management of complex computing systems. IBM defines Autonomic computing as self-managing characteristics of distributed computing resources. The self-managing characteristic are to adapt to changes from the system and surrounding environment, which are usually elusive and unknown beforehand. An autonomic system often relates to an autonomic loop which collects information from one or multiple managed elements and then analyzes them to build appropriate plans for autonomic responses. These plans then will be imple-

mented on the managed elements to achieve goals of autonomic properties. The autonomic loop and properties will be discussed soon next.

2.2.2 Autonomic Loop

The MAPE-K is an autonomic loop reference defined by IBM itself. In this reference model, we see appearance of five core components, namely **M**onitor-**A**nalyze-**P**lan-**E**xecute-**K**nowledge, which operate seamlessly and continuously. Data are collected and control commands are triggered deliberately in this closed loop. A demonstration of the MAPE-K is found in Figure 2.4.

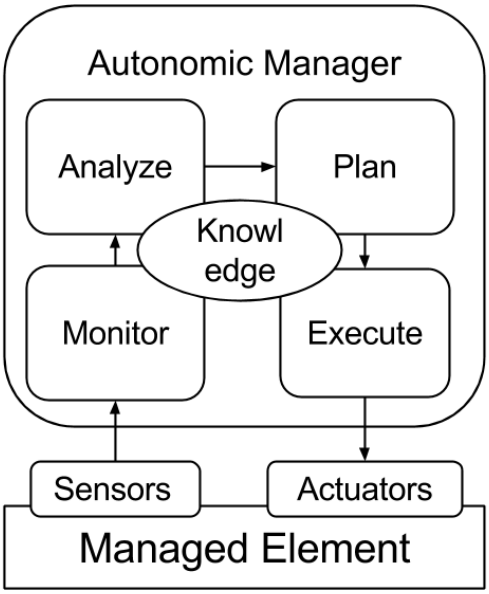

Figure 2.4: MAPE-K autonomic system loop

In this loop, information about the managed element will collected progressively and dynamically by sensors. These sensors can be independently small program or embedded software in a hardware device. Their mission is to invigorate, collect and transmit data from the managed element to the Monitor component using the given metrics. The monitor is responsible for monitoring the entire system, who will aggregate data received from the sensors, from the internal system and perhaps from other sources. This operation captures the current state of the system and then data will be delivered to the Analyze component for further analysis. Next, the analyzed information combined with some planning algorithms construct plan of changes to be applied on the managed element. Simply, these plans are described as a set of discrete instruction commands to modify

incrementally the current state of the managed element. In some cases, the autonomic manager needs to compare in advance the current state with the proposed plan to generate only differences which finally applied to the managed element instead of rebuilding the entire system. Planning is often based on knowledge which is either captured real-time from the system or retrieved from the historical statistics. Knowledge also could be put on the system from the very beginning. The instruction commands finally are executed on the managed element by the actuators. The mass of knowledge will accumulate and grow over time and is circulated continuously in the system. To circulate the knowledge as well as the control directives, it needs the help of a component responsible for communications such a messaging server. The instruction commands finally are executed on the managed element by the actuators, which put the managed element into the expected state described in the SLA. The system in the new state is also monitored regularly to detect changes and therefore constitute the autonomic loop.

2.2.3 Autonomic Properties

The fundamental properties of autonomic computing have been pointed out by Kephart et al. [7] are self-configuration, self-healing, self-optimization and self-protection. However this list is not exhaustive. Poslad [146] have added to this list many self-* properties such as self-regulation, self-learning, etc. They represent different purposes that an autonomic system desires to achieve.

Self-configuration Self-configuration enables the capability for a system to reconfigure itself to adapt to changes. The reconfiguration is to maintain system staying in a desired state which can be declared in the SLA or be setup in advance by system administrator. This might include adding/removing the components of an existing infrastructure, which affect to result of reconfiguration. For example, a VM can be added to a cluster in order to enhance the availability and reliability of a web service.

Self-healing Faults and errors sooner or later will appear in any computer systems, which range broadly from software bugs to hardware failure. The autonomic systems should be able to not only discover and correct existing faults, they should but also to predict and prevent potential errors. Therefore it needs to apply proactive solutions to make the system more robust.

Self-optimization An autonomic system needs to continuously improve and optimize itself in terms of cost, performance, etc. This ties closely to monitoring the resources provided to the system and the amount of resources actually used to give timely feedback to the system. The elasticity in the Cloud also expresses such the same behaviors, thus elasticity systems are often built on an autonomic loop. Operations of optimization can be either reactive or proactive depending on specific conditions of the system under consideration.

Self-protection It is a property which helps an autonomic system to detect and prevent intrusion from outside as well as anomalies from inside which cannot be handle by self-healing policies. The security threads and holes need to be identified and must be inhibited as early as possible.

2.2.4 Autonomic Cloud Computing

Since the cloud systems are more and more complex and heterogeneous, they need efficient mechanisms to fulfill the new requirements. Thereby the marriage between Cloud computing and Autonomic computing is essential and it is becoming an emerging research trend, namely Autonomic Cloud computing. It is a result of applying the aforementioned self-managing properties of Autonomic computing (self configuration, self-healing, self-optimization, self-protection) in the cloud environment. As pointed out by Kephart et al., the advantage brought from Autonomic computing is not only the autonomic elasticity management. This is a two-way relationship of mutual benefits. The properties of Autonomic computing can be used efficiently to resolve some challenges coming from Cloud computing such as resource allocation or infrastructure optimization. At the Cloud end, this is a fertile ground for the autonomic implementations because of its inherent complexity and dynamic nature.

2.3 Synthesis

In this chapter, we have pointed out the characteristics of Cloud computing, the benefits that it brings to and the challenges that it has to deal with. We revealed how Cloud computing can provide unlimited resources and leave IT complexities behind, which help its users to focus on their core businesses. To deploy applications and implement elasticity approaches on them, automation is indispensable. Autonomic Cloud computing with its ACCSs is the solution behind most Cloud management tasks. The ACCSs often implement a variant of the MAPE-K autonomic loop to obtain one or multiple properties of Autonomic computing: self-configuration, self-healing, self-optimization and self-protection. In the next chapters, we will see how Autonomic computing techniques are used to deploy cloud applications and implement elasticity solutions. This reflects the self-configuration and self-optimization properties of Autonomic computing.

Chapter 3

ELASTICITY

Contents

Elasticity term originally comes from a definition in physics as a properties of substance returning to its original state after a deformation. On the other hand, it is defined as sensitivity of a dependent variable to diversification in other variables in the theory of economics [56]. The concept of elasticity has been ingeniously transferred and popularly used in Cloud computing as a momentous characteristic. In comparison to other characteristics of Cloud computing model, elasticity

is gradually gaining more attractive from researchers both in academic and industrial sectors. Whereas much research reach to an agreement that elasticity is an unique characteristic of Cloud computing which distinguishes it from Grid computing, there is no consensus on its definition and classification. It is also missing an unification of requirements for an elasticity controller or a Cloud system supporting elasticity. This section discusses about elasticity based on an abundance of state-of-the-art solutions in the literature.

3.1 Definition

This section goes over definitions of elasticity, thereby equivalences and differences are analyzed thoroughly. According to the broadly recognized definition of NIST [20], elasticity is an essential characteristic of Clouds which enables elastic provisioning and releasing of capability and rapid scaling depending on demands of application. Although the automation for elasticity is not strictly required in the NIST definition, it is obligatory in some other works. For instance, elasticity definition of Schouten [58] includes removing "any manual labor needed to increase or reduce capacity", which highlights the role of automation in building an efficiently elastic solution. NIST delineates an ideal circumstance where the capabilities available for provisioning often appear to be unlimited and can be rented in any quantity at any time by customers. It is matter of fact that cloud providers are not able to provide a real "unlimited" sky of resources. In addition, demand of virtual resources for elastic applications is sometimes getting really high for group of special consumers, the cloud providers must tighten resource-provisioning upper bounds which weakens and blurs the no limitation of elasticity. Similar to the NIST definition, Leymann adds a statement saying that the elasticity also implies the changing over time of actual resource demands without any clues predicted beforehand [55]. Having consensus with [20] and [55], Galante and de Bona [60] propose a widely accepted definition which assigns elasticity the capability to "quickly request, receive and later release as many resources as needed. The elasticity implies that the actual amount of resources used by the Cloud user may be changed over time, without any long-term indication of the future resource demands".

The resource scaling to increase or decrease capacity is used as a critical element to define elasticity [20, 57, 58]. With ODCA [57], elasticity is ability to scale up and scale down capacity based on subscriber workload, whereas [58] even defines elasticity is just a "rename" of scalability. The confusion between elasticity, scalability and some other similar terms really exists and is going to be made clear in Section 4.3. Correlation of provisioned resources to fluctuation of workload demand is a part of elasticity definition in [59]. This statement more

or less gains some consent with [20] and [57]. Rich Wolski, CTO of Eucalyptus, promotes the resource-scheduling feature of elasticity when says that elasticity is a non-functional measure for ability "to map a single user request to different resources" [61] on Cloud. Although mentioning to quantifiability of elasticity, it is too succinct to cover all aspects of elasticity, especially in QoS.

It is worth mentioning a definition of Herbst which insists that elasticity is "the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible" [62]. This definition covers both scalability and adaptivity of elasticity. It forces to implement autonomic mechanisms to release elastic system from any human operators. Instead of drawing an idealistic picture of "perfect" elasticity, resource supply just needs to match workload demand as much as possible.

Although they are somewhat different, these definitions still have some unities in terms of scaling, adaptivity and autonomy. Other features have not really been mentioned as the requirements of an elastic manager will be pointed out in Section 4.3. Next section is specially dedicated to classification of elasticity solutions.

3.2 Classification

Industrial and academic solutions regarding elasticity can be classified based on many factors, mainly concerning requirements of an elastic system speedy and efficiently. In 2012, Galante and de Bona propose a classification of elasticity works in their especially successful survey [60], which based on main features found in the analyzed commercial and academic solutions. It contains four different axes: scope (infrastructure, platform, application), policy (manual, automatic reactive, automatic proactive), purpose (performance, infrastructure capacity, cost, power consumption) and method (replication, redimensioning, migration). This classification gives a fast and simple view of mechanisms using for elasticity. However, because of space limitation, this one does not detail in specific aspects that it describes.

Two year later, another approach to elasticity classification is suggested by Najjar et al. [63]. It introduces an analysis grid to classify state-of-the-art elasticity management solutions based on three main axes: strategy (quality goal, policy, mode), actions (type, granularity) and architecture (centralization, decentralization). Some axes in this grid interfere with some classes in the work of Galante. For instance, the mode in [63] completely matches the method in [60]. A valuable contribution of this paper is the considering SLA-awareness aspect of elasticity with various kinds of service level objectives (SLOs) such as QoS, quality

of business (QoBiz) and quality of experience (QoE).

The most recent survey that tries to resolve a taxonomy of elasticity is Coutinho et al.'s work based on a systematic review [64]. They focus their concern in analysis tools and evaluation metrics for elasticity. It also conducts a comprehensive work on its review procedure with some interesting statistical results. However their taxonomy for existing elastic solutions is quite simple with combination of two methods (horizontal scaling/replication, vertical scaling/resizing and replacement, migration) and models (reactive, proactive/predictive). It is unified with classifications of [60] and [63] except introduction of the replacement mechanism. This mechanism allows replacing less powerful servers by more powerful ones in clouds where on-the-fly resizing approach is not supported.

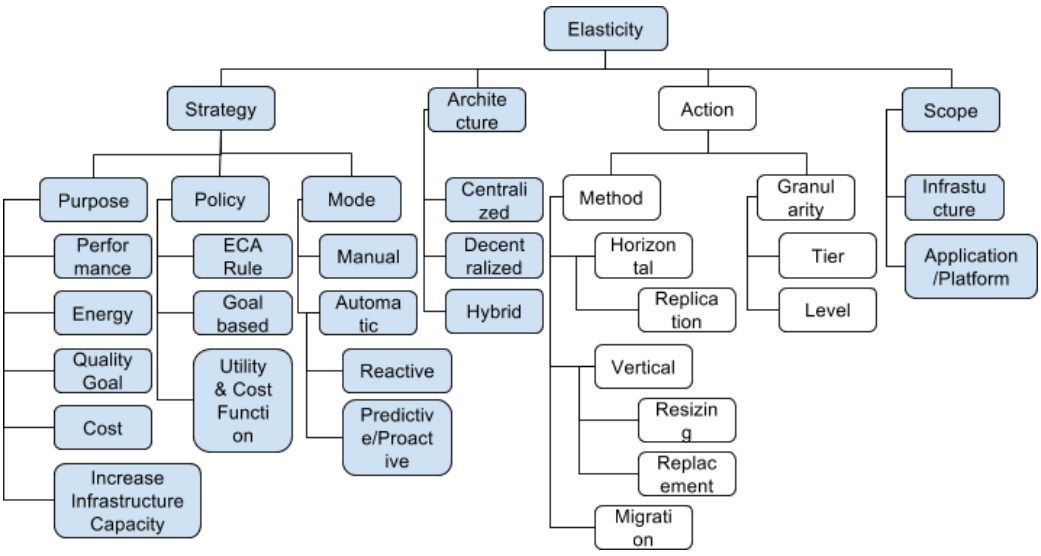

Figure 3.1: Analysis grid of elasticity solutions

Figure 3.1 demonstrates our taxonomy that combines holistically from the three mentioned approaches. The analysis grid of Najjar is expanded to include the classifications from [60, 64]. If traversing from top to bottom of any branches, an existing elasticity solution can be arranged into one of leaves of this branch. Thereby all elasticity state-of-the-art can be captured and distributed to the leaves. However a solution can implement many mechanisms, thus we can combine two or multiple leaves to have more fine-grained classification. For example, Amazon Auto Scaling [65] is a proprietary elasticity solution that implements the automatic reactive mode and based on replication method.

In comparison to the three taxonomies, we add one more class, the "Level", to the "Granularity" class of the "Action" axe, to cover multi-grained elasticity

solutions which is main focus of this thesis. As our research in elasticity and other related work are presented more easily if arranged into the "Action" axe, we leave discussing about this axe later in Section 3.3. The remaining of this section is dedicated to describing about the other axes of this comprehensive analysis grid as well as their state-of-the-art.

3.2.1 Strategy

It is necessary to identify strategies in elasticity decision-making process. These strategies can be drawn from purposes which the applied elasticity mechanisms need to be achieved such as performance, cost, power consumption, quality or infrastructure capacity expanding. It also can be derived from policies (ECA rules, goal-based or utility and cost functions) specifying how to obtain strategic purposes or from modes (manual, automatic reactive or predictive) stipulating manners the elastic system interacts with the Clouds.

Performance is the most concern among the mentioned purposes. Much research in elasticity relate to performance measurement [66, 67, 68, 69, 70, 71]. In these ones response time and CPU utilization are two of the most used performance metrics [63]. Using elasticity for optimizing cost of both cloud providers and customers also attracts the attention of some research groups [72, 73, 74, 75, 76], whereas reducing power consumption is a driving motivation of [21, 77]. In Grid computing, computing resources stay in its datacenter that limits resource provisioning capability. In context of cloud bursting, this limitation is pushed back because application staying in its private cloud can borrow additional resources from public cloud when the demand for computing capacity spikes. The capability to increase infrastructure capacity is one of purposes that able to be reached by elasticity [78, 79]. The SLA-awareness strategies also need to be concerned, especially in QoS, QoBiz and QoE. QoS is an objective quality measurement stated in a contract called Service Level Agreement (SLA). In context of Cloud computing, this contract can be signed between either a CP with its cloud provider user (CP-U or CAP, i.e. SaaS provider) or a CAP with its CAU. Using elasticity to ensure QoS is widely discussed in [67, 70, 76, 81]. In contrast of QoS, QoBiz is a subjective quality measurement including factors which are business considerations such as provider satisfaction or revenue. It concerns matters directly affecting the CP and CAP, which are pricing policy, billing time unit, penalties of SLA violation. We found [69, 71, 76, 80] which are solutions cope with QoBiz. Unlike QoS and QoBiz, QoE focuses on experience of customers about services that they used. However this quality goal are not being received adequate attentions. It is mentioned as a potential field of research in [81] and addressed on the network issue of Cloud gaming in [82].

Event-Condition-Action (ECA) is a simple way to obtain elasticity based on

rule-based control. Elasticity decisions are executed when specific events are captured and particular conditions are satisfied. These conditions often relate to the handling of metrics around their thresholds. Defining correct thresholds to avoid oscillations is the most challenging problem of this approach. Applying multiple [83] or dynamic [84] thresholds are some of solutions of this issue. Numerous elasticity solutions, both in industry and academy such as [16, 21, 65, 71], offer ECA because of its simplicity. Goal-based policy tells system a desired state to which system should reach after elasticity decisions. Using goal-based policy, firstly an overall model about system must be constructed and studied by the system itself. As benefits from elasticity is not enough to compensate the efforts spent to build the system, this research trend is still unresolved. Utility function uses preferences to weight attributes of a service or product which satisfies a customer in many levels. Cost function is simply an opposite meaning of utility function. System has to decide either maximize its utility or minimize its cost. Using these functions can capture preferences of three main players in cloud ecosystem, which are CP, CAP and CAU. [69, 72, 76, 80] are within research works based on this theory.

Manual mode means users have to intervene in monitor and decision-making process of elasticity operations. Cloud providers have to provide at least an API for interactions. Although automation is not strictly required, manual handling causes annoyance once complexity of system increases. Human is not able to carefully monitor or trace a huge and complex system without any mistakes or boredom. Elastic decisions, sometimes repeated, are tedious and error-prone if carried out by human. In academic research, Elastin [86] and Work Queue [87] offer resource manual management. While Rackspace and GoGrid [85] are cloud providers who implement manual policy for elasticity. On the other hand, automatic policy tries to reduce human intervention as much as possible. This way often implies the implementation of an autonomic elasticity manager which controls autonomic behaviors of the entire system. Otherwise, application itself must be recoded or upgraded to be elasticity ready. Reactive solutions are usually based on ECA-rule policy and commonly provided by cloud providers such as Amazon Auto Scaling, Rightscale [16], Scalr [17] as well as academic works [88, 89, 90, 91]. Proactive mode uses predictive techniques to forecast trends of upcoming workload and scale elastic system according to these trends. These techniques often apply the analytic/statistical maths and heuristics based on historical load data or load pattern inferred from a period of time. Proactive methods need to take into account overhead caused by predictive algorithms themselves. While it is easy to find plenty of studies in this specific aspect [17, 92, 93, 94, 95], the predictive solutions are received slowly in the industry since the missing of benchmark tools to evaluate which are better ones.

3.2.2 Architecture

With regard to architecture, general organization of elasticity system is taken into consideration, which falls into three categorizations: centralization, decentralization. Among the two, centralized architecture is used in almost elasticity solutions. We utilize the multi-component MAPE architecture usually used in autonomic elasticity solutions to demonstrate a possible example of the centralized architecture as in Figure 3.2. In this type of architecture, elasticity components stay together in an autonomic manager. The system monitoring is performed remotely by the elasticity managers itself or outsourced to third party tools, which in both cases locate outside the managed elements. Main inherent issue of this architecture is that the autonomic manager is a point of bottleneck in the system.

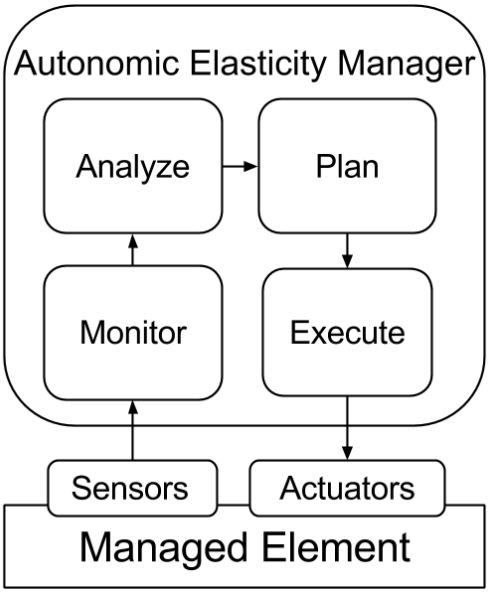

Figure 3.2: The centralized architecture as a variant of MAPE-K architecture

Multi-agent system [96] is usually found in the decentralized architecture. In this kind of architecture, agents are implemented on the managed elements. The agents are responsible for mitigating burden for the elasticity manager by carrying out most of autonomic and elastic works such as monitoring, analysis, planning and execution. Responsibility of the manager is to communicate with cloud providers for resource provisioning in response to elasticity requests from the agents [97]. Each managed element can contain one or multiple agents to manage some kind of resource units such as VM, CPU, memory, etc. These agents, called CUA (capacity and utility agent) in [98], probe and snoop the multi-grained resources to reflect a current picture of system based on resource parameters and

workload characteristics. In another research, Chen et al. develop an elasticity solution using combination of utility function and multi-agent based architecture to obtain the cost minimizing purpose [72]. A possible example about locations and missions of each component of the decentralized architecture is illustrated in Figure 3.3, also through using of the MAPE-K architecture.

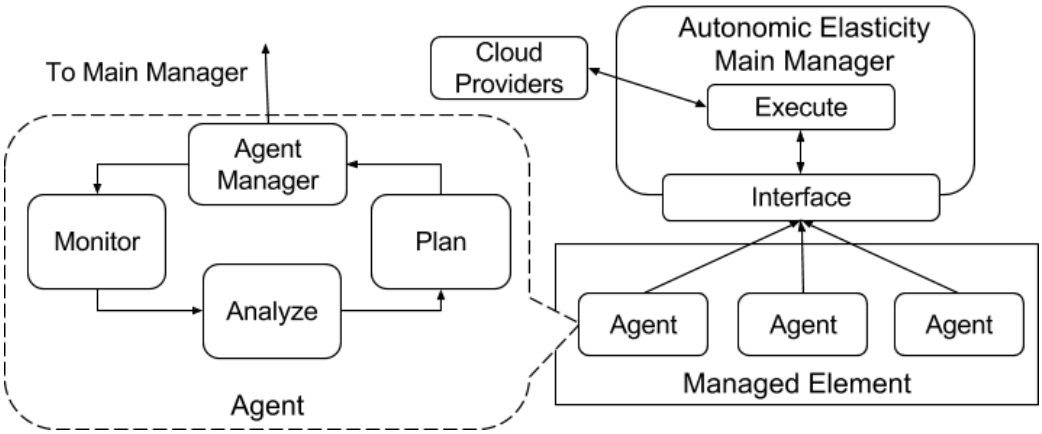

Figure 3.3: The decentralized architecture as a variant of MAPE-K architecture

3.2.3 Scope

Scope is an easy-to-understand classified way specifying where elasticity operations occur. All three layers of Cloud service model, i.e. IaaS, PaaS, SaaS, can host elasticity solutions. Elasticity is mostly implemented on the IaaS level with basic actions as adding new resources and releasing unused ones with regard to inconstancy of workload. Resources can be in coarse-grained form as VM or fine-grained unit as lightweight container, CPU or memory. Amazon, RackSpace, GoGrid and many other cloud providers offer elasticity IaaS. Much aforementioned elasticity research in this thesis can also be categorized into this kind of scope, some of them are [45, 77, 93, 94, 95].

Elasticity PaaS pays more attention to scaling software container, software component of platform itself or database. Any kinds of software containers can be applied elasticity from Java war, OSGi bundle to EJB container. Lightweight container is a kind of special resource. It can be either a scaling resource at IaaS level if it is deployed directly on PM and bare-metal or at PaaS level if it deployed on VMs on the Cloud. Google App Engine, Heroku Platform [29] and Microsoft Azure are cloud providers who more or less provide elasticity PaaS to

their customers. Elastin, Aneka [79] or Cloud Hosting Provider [78] are academic implementations of PaaS platforms supporting elasticity.

Some applications natively supports elasticity since there are some features integrated into these software from the very beginning. These features relate to parallel processing in terms of tasks, threads and data or load balancing-supported mechanisms. Unfortunately, not many applications own these enhancements, especially the legacy applications which are not Cloud-ready since inception. These applications need to be improved by doing some code changes that make them Cloud-ready and enable their elasticity capability as well. Using wrapper is another help for the legacies. Wrapper which "wraps" the legacy applications into elasticity-enabled cloud containers or components is a non code-change method used in [106, 107, 108]. Elasticity solutions for particular kinds of application are also available in the context of messaging [21, 100, 101], stream processing [102, 103, 104], or MPI [105].

3.3 Elasticity Actions

As mentioned in Section 3.2, this section devotes to the "Action" axe where we position our research about multi-level fine-grained elasticity there. This axe describes actions issued by elasticity decisions based on strategic algorithms to provision or deprovision cloud resources. There are two manifestations of an elasticity action:

- *Method*: type of action that an elasticity manager can perform or offer to obtain expected resources or forsook them. There are three main methods showing dimensions of expansion of resources, which are horizontal scale (replication), vertical scale (resizing, replacement) and migration.

- *Granularity*: affected scopes of elasticity actions. The scopes can be either tiers (one or multiple software tiers) or levels (one or multiple levels of resource granularity). An application may have one or multiple tiers based on complexity of its components. On the one hand, Apache itself is a standalone application with one tier, often called the web tier. On the other hand, LAMP is a software model for multi-tier applications usually relating to Linux, Apache, MySQL and PHP. This model implies implementation of three tiers which are web tier (Apache), application server tier (PHP application servers such as Zend [1]) and storage tier (MySQL). The level refers to variety in size of resource units. A resource may in coarse-grained level (VM, PM) or in fine-grained one (container, database, CPU).

[1] http://www.zend.com/en/products/zend_server

We apply the analysis grid on the combination of the "Method" and "Tier" classes of this axe to "trap" the current elasticity works more relevant to our research in this thesis. This consequently leads to a need to have a new class (the "Level") added to the analysis grid, which tries to resolve new issues of elasticity (see Chapter 4).

3.3.1 Horizontal Scaling on Various Tiers

The horizontal scale refers to using replication to multiple resource units. It is the most used method for elasticity in cloud environment. Improving scalability, availability and performance are within benefits of replication. To avoid SLA violation because of increasing workload, CPs or CAPs have to add instances or replicas of appropriate resource thanks to replication. Furthermore, they also ought to remove these replicas when the workload decreases. These instances usually are VMs, containers or applications. On the CP-Us or CAUs side, they need to vary the leased instances according to the resource demand of their applications for improving performance or saving cost. Evidently, changes at the CP-Us or CAUs side can only be performed if the CPs or CAPs offer them the corresponding scaling features. Elasticity actions can simultaneously affect many tiers of an application which implements multi-tier architecture. Following is solutions using replication as elasticity action which affects one or multiple tier of applications.

Fito et al. [78] present an SLA-aware resource management solution specially for web tier. This elasticity solution adds or removes web server replicas in accordance to dynamic load. Revenue of cloud providers is studied and maximized based on economic variables and functions. Zhao et al. [109] focus on database tier replication when trigger corrective actions based on SLA performance using his proposed framework. The corrective actions scale the database to reduce SLA violations. A solution for SaaS providers based on cost-aware is discussed in [74] by Liu et al. taking into account user requests and the current workload. The authors consider the solution as a problem of optimization with minimizing SLA violations to satisfy customers and maximizing profit for cloud provider. In order to achieve these objectives in reasonable time, a genetic algorithm called CSRSGA is developed and a validating simulation-based experiment is conducted using replication of EC2 instances. In contrast to the Liu work, the elasticity solution proposed by Hasan et al. [67] is completely agnostic to the cost-awareness of renting the VMs. In this work, the authors develop IACRS, a rule-based elasticity system that is more sophisticated than traditional ECA rule-based policies. In addition to its complex rules and metrics, the algorithm also employs an advanced mechanism of four thresholds. Based on these thresholds, the authors introduce a set of heuristic rules that infer when it is necessary to invoke or extinguish re-

sources from the Cloud using replication method. However this solution only consider per tier SaaS application.

Next, we discuss works presenting holistic horizontal scaling managers which pay attention to multi-tier applications. Han et al. [73] use cost-aware criteria to analyze and detect bottlenecks which could happen within a 5-tier cloud application. The authors propose CAS algorithm analyzing application behaviors to handle changing workloads to the five tiers. Thereby the CAS scales up and down the bottleneck tiers adaptively, mainly by adding/removing the amount of VMs needed to restore the response time to the desired threshold. In [111], a novel caching and replication infrastructure is presented by Perez-Sorrosal et al. in order to handle the workload variation. The infrastructure facilitates the elastic deployment and scalability of multi-tier architectures by adding and removing resources to change capacity at runtime without service interruption. Iqbal et al. [110] is one of elasticity research on two-tier applications composing of a web server tier and a database tier. This paper distinguish two kinds of workloads causing stress on different tiers, which are static requests targeting on web server and dynamic ones aiming at both web and databases. The tier causing bottleneck is detected by a heuristic algorithm and additional VMs are added to this tier. Authors use predictive mode for both scale-in and scale-out decisions.

All the studies in this section use replication action to trigger elasticity behavior in one or multiple tiers of cloud applications. The drawback of this approach is appearance of a load balancer which reroutes the requests made after the replication finishes. The load balancer distributing the requests to new replicas may cause overhead because of some internal processing operations. Almost load balancing services must be restarted to recognize the new replicas or to capture the changed model of system, thus puts even more overhead and slows elasticity. Furthermore, load balancer itself is a bottleneck, indisputably.

3.3.2 Vertical Scaling on Various Tiers

Vertical scale refers to resizing or reconfiguration of resources to adapt with fluctuations of workload. The resources are not replicated or cloned, they bulge out instead. Adding or removing memory or CPU cores to or from VMs is a simple example of vertical scale. Obviously, operations relating to resizing hardware resources must be authorized by cloud providers, especially the public ones. Unfortunately, not many providers allow such the operations [66]. Instead of that, they offer fixed-size instances, called flavors. For instance, Amazon has several kinds of VMs with different combination of hardware configurations such as micro, medium, large. Microsoft Azure launches a broader range of instance types including A and D series. When workload varies, the adaptive changing from a less resource VM (smaller flavor) to a more resource one (bigger flavor) is called

"replacement", a variant of resizing (or redimensioning). Replacement method is indeed an interim choice when cloud providers does not support adjusting resources of VMs or containers.

In some occasional cases, the cloud providers allows resizing the vCPUs, memory or network bandwidth of VMs. Even so, the hot resizing on runtime is still not accepted. In this situation, the hot resizing may be performed by some advanced techniques for VM such as ballooning in inflating/deflating memory or Xen credit scheduler [114] in adjusting number of vCPU and CPU time share [92]. With lightweight container technologies like Docker, OpenVZ or LXC, the similar things can be done by accessing the low-level cgroups settings. However, research on this aspect of container elasticity is really rare. With applications, resizing is a standard method in adding or removing data or control structures (i.e. structural dimensioning) in order to exploit resources available in VMs. This technique relates to tuning the parallelism of processes, threads [115] or MPI execution nodes of an application [105]. Elastin [86], a framework using a compiler to combine different program versions into a single application that we can switch between the versions at runtime without rebooting the application, can be considered as structural dimensioning.

Migration may be classified into the horizontal scale or in an independent class. When workload surges, some VMs, for instances, should be moved into bigger physical machines to have more resource satisfying the increasing demand. These old VMs and perhaps smaller PMs then are removed, thus not increase total number of VMs in the entire system. We consequently have the VMs located in stronger PMs where there are enough resources for the given demand. That is why we can put migration in horizontal scaling class. At the other end, when the workload declines, VMs may be collocated in order to reduce number of PMs, hereby saving energy and cost. The replacement method is a type of migration, but the VM does not move out of PM. In the point of VM view, the applications inside the old VM may be migrated to the new one (bigger or smaller depending on workload, for instance), thus replacement is a branch of migration in some cases. Although research on the horizontal scale are less than the vertical scaling, it is indeed not too hard to find in the literature. The remaining parts of this section identifies some of them.

Dawoud et al. in their research series about elastic VM [112, 113] for cloud resource provisioning introduces a vertical scaling architecture in comparison with Multi-instance architecture (horizontal scale). They compare two architectures based on both analytical queuing model and characteristic implementation. Their experiments on web and database tiers result in less response time and provisioning overhead for vertical scale, thus reduce SLA violations. Whereas Dawoud separately considers the elastic VM and Multi-instance architectures, an industrial solution, called OnApp Cloud [116], offer both replication and resizing that

bring more flexibly to elasticity decisions.

A predictive elasticity system, called PRESS [147], uses Fast Fourier Transform (FFT) to identify repeating patterns (signatures) used for its prediction. The CPU resource is re-configured using the Xen Credit CPU Scheduler. For reducing power consumption, CloudScale [77] takes advantage of modern processors which can switch and operate at different voltages dynamically. Without affecting the SLA, CloudScale can increase or decrease the CPU frequency or voltage, thus optimizes energy consumption. GoGrid develops a memory scaling tool which allows vertical scale in a fast and easy way, but a rebooting is required. Tran, in his thesis [114], describes a coordinate resource mechanism which provides both horizontal and vertical elasticity. With horizontal one, he adjust capacity of VM's vCPU and memory using Xen Credit CPU Scheduler and memory ballooning, respectively. The evaluation shows that using the coordinate mechanism can reduce performance overhead and increase virtualized and physical resource usage.

3.4 The Unresolved Issues

It must be recognized that the current elasticity solutions from the cloud providers and the academic research have partly resolved many challenges of elasticity. Nevertheless, as the cloud systems and its ecosystem are continuing on development, it poses new challenges and open issues which need to be addressed.

3.4.1 Resource Availability

It is a matter of fact that resource capacity of a cloud provider is limited by physical servers in its datacenter. It leads to a sooth that the cloud providers have to impose strict resource limitation on their customers, which neglects the infinitive resource premise [133]. For example, Amazon EC2 only allows 20 simultaneous requests for on-demand instances and 100 spot instances per region [60]. Microsoft Azure limits 50 VMs and 150 input endpoints per cloud service. Although using multicloud could be a solution for resource limitation, resource-intensive highly scalable applications will soon deplete the resources distributed dynamically for them.

A solution for cloud customers to overcome the number of VM limitation imposed by cloud providers could be utilization of the vertical scaling to resize the fixed VM. Likewise, using lightweight container such as Docker inside VMs to scale at component or application level is a suggestion worthy to be taken into consideration. In general, applying elasticity on fine-grained levels of resources would slow down the resource depletion.

3.4.2 Resource Granularity

Horizontal elasticity action broadly supported by current cloud providers. They usually allow fixed-size VMs to be scaled out depending on current workload demand. On the contrary, vertical elasticity is offered sparingly by the providers. Even if redimension is supported, resizing VM resources on the fly is prohibited. GoGrid allows its customers to increase RAM of VMs, but requires a VM reboot. Although EC2 introduces a wide range of VM instances with different sizes and configurations to simulate vertical scaling when needed (VM replacement or substitution), VM restart is still a must.

The coarse-grained scaling with fixed-size VMs often leads to resource provisioning overhead resulting in the over-provisioning. Research on elastic VM of Dawoud et al. [112, 113] about fine-grained scaling or of Rodero-Merino et al. [71] about VM substitution to simulate resizing have partly resolved this challenge. However, cloud providers is most likely prefer providing horizontal scaling with fatty VMs, research on combination of elasticity actions on multiple levels of resource granularity (at both coarse and fine-grained levels) is necessary. Furthermore, the lightweight container technologies introduce a thinner approach than VM, which could make elasticity actions more fine-grained.

3.4.3 Startup Time

Provisioning a VM in an elasticity scaling often takes sometimes to complete. This unavoidable time, namely startup or spin-up time, varies from one to dozen of minutes. In the ideal elasticity, resource should be provisioned instantaneously. Therefore start-up time may affect the speed and efficiency of elasticity mechanisms in handling highly dynamic workloads. Applying a predictive elasticity could be a good solution because VMs are pre-provisioned based on forecasting the workload. However, the predictive approach is often more complicated and historical behavior of workload must be well-studied in advance. Forecasting errors may still occur, resulting in unused resources which steal money from cloud customers.

Reactive scaling is still useful for unplanned events such as a flash crowd or slashdotting when a popular website links to a smaller site, causing an unpredictably massive increase in traffic. In this situation, a vertical scaling up or a container-level scaling out could be a solution (if possible) to gain rapid elasticity.

3.4.4 Rising of Container-based Virtualization

Given a fact that conventional virtualization comes with overhead and cloud providers are unlikely to offer adjusting their VMs at runtime, therefore, container-based

virtualization with technologies such as Docker, LXC could be an alternative. Although research on container-based as an object for elasticity is still rare, an easy-to-see benefit is that it reduces spin-up time of replicas because of its smaller image size in comparison to VM image. Moreover the container managers usually have well-managed mechanisms for resource allocation, the vertical scaling for lightweight containers is a realistic scenario. Wrapping software components in the containers is an efficient solution not only for workload isolation but also for component security. A research on container wrapping using Docker is about software consolidation [124].

3.4.5 Composability

Cloud application is often comprised of multiple software components or modules. A component provides some outputs for other components but it also needs some inputs from other sources. This creates a flow of processes and data. Majority of elasticity actions directly or indirectly affect these components. If one of the components becomes a performance bottleneck, we might only need to scale it firstly. Elasticity actions should focus on a particular component each time it is triggered. However dependencies among components also need to be taken into account.

3.5 Synthesis

In this chapter, we have presented a comprehensive research and works on elasticity classified by using the analysis grid. We have also discussed a representative range of approaches related to elasticity actions in two aspects: method and granularity. As we have mentioned, there are some new issues that remain unsolved by the current elasticity approaches. In the next chapter, we detail our contribution in definition of a novel elasticity approach that aims to resolve these issues.

Chapter 4

MULTI-LEVEL FINE-GRAINED ELASTICITY

Contents

This chapter devotes itself to our first contribution in this thesis: the multi-level fine-grained approach for elasticity.

4.1 Definition

In the last section, some new challenges and open issues of elasticity have been pointed out. An elasticity solution now should not only provide the efficient scaling but also consider using resources economically. We have also discussed a couple of potential solutions to partly resolve these issues. In general, our suggestions primarily focus on applying elasticity actions on multiple levels of resource granularity, which include all hardware, virtualization and software layers within boundaries of solution. In addition, the more fine-grained elasticity solutions should be used with priority as much as possible. We define here a new approach, namely multi-level fine-grained elasticity, which covers advantages of our suggestion.

Definition 1 (Multi-level fine-grained elasticity) *Multi-level fine-grained elasticity implies implementation of scaling actions on multiple types of resources.*

Each resource type is a dimension of adaptation process and possesses its own elasticity properties. If a resource type has containment relationships with other resource types, like in the case of a lightweight container having hosted by a VM, elasticity can be considered at multiple levels. At each level, more fine-grained elasticity methods should be considered with priority if possible.

Figure 4.1 is a map presenting potential resource types of a conventional cloud system which are distributed into multiple levels with respect to the containment relationship. The list of resource types, which are objects of elasticity actions, is not exhaustive.

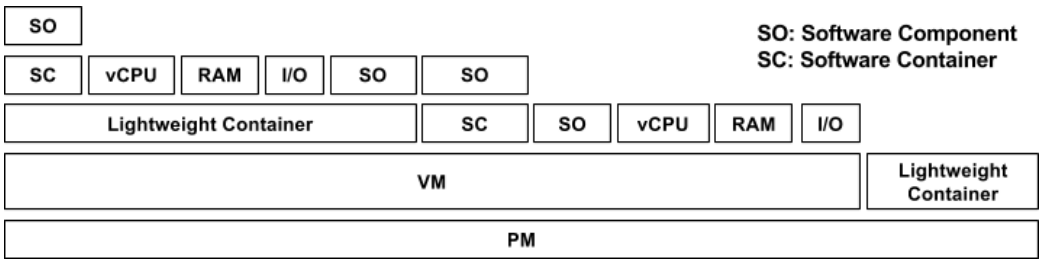

Figure 4.1: Multi-level resource types

On the cloud system, VMs are hosted on PMs at datacenters of IaaS providers. Each VM is allocated a set of resources including vCPU, RAM or I/O devices which may be objects of vertical scaling up/down. Especially, each VM in public clouds may contain lightweight containers (e.g. Docker, LXC, etc.) which can be scaled out with its entire set of resources (e.g. vCPU, RAM, etc). These resources of containers, in turn, may be adjusted to obtain the vertical scaling in/out. Software container such as Application servers (Tomcat, JOnAS) or OSGi platforms (Karaf, Equinox) is a kind of special resource. It can be deployed on both VMs or lightweight containers to provide execution environment and essential libraries for specific types of applications. A cloud application is often complex and consisted of multiple software components. These software components can be run directly on VMs (e.g. install an Apache server on a VM). In some cases, software containers may be required for installation of software components (e.g. a .war file with servlets needs a Tomcat server for running). With software consolidation, a software component or software container can be packed into a lightweight container to be more isolate and secure. Thereby, two software components or containers (may be at the same type) can be collocated inside the same VM. We see that the flexible mix of resource types opens an unlimited combining capability for elasticity actions.

When the elasticity actions target more fine-grained resources, we say that

these actions are using more fine-grained methods. Definition 1 suggests that more fine-grained elasticity methods should be considered with priority if possible. With regards to Figure 4.1, more fine-grained scaling resources are located at the upper positions of the map (from the top to bottom). Thence the software components or software containers should be the priorities and then considering technologies of virtualization such as lightweight container or VMs. The order of scaling resource types at the same level is resolved based on if we can apply vertical scaling (on RAM, memory, I/O devices) or structural scaling (on software component).

Although this new approach is very promising in resolving a part of the new elasticity issues in theory, it is necessary to know where is our elasticity approach in comparison to related work. According to [62], to evaluate a multi-level elasticity approach, we need to consider at least the following aspects:

- *Autonomic mechanism*: How to have autonomic scaling? What adaptation process have been used?
- *Resource type*: the adaptation process affects which set of resource types? (and at which levels?)
- *Resource granularity*: How is granularity of elasticity actions on the given set of resources? Which scaling method used in a specific elasticity action (horizontal, vertical or structure scaling)?

In next section, we use criteria in these questions to elaborate works similar to our approach. It means all state-of-the-arts of the "Level" class in the elasticity analysis grid (see Figure 3.1), which have the combining utilization of both horizontal and vertical scaling on various levels of resource granularity, are taken into account.

4.2 Related Work

As mentioned, elasticity actions may be classified into smaller classes according to granularity. Granularity is scope impacted by elasticity actions which could range from a tiny file to multiple coarse-grained machines [34]. If divided by logical separation of application or the way of organizing the code, we have scaling actions on various tiers as discussed in Sections 3.3.1 and 3.3.2. If taking into account the physical location of applications or where the code runs, we have scaling actions on different resources allocated on various levels as discussed in Section 4.1. These levels are various from the coarse-grained physical server, VM to the lightweight ones like container, application or even more fine-grained such

as amount of memory and number of vCPU cores. Naturally, there is not much research on vertical scale because of strict regulations from cloud providers. Research on vertical scale affecting multiple levels of resource granularity are even rarer. Thus we consider elasticity solutions applying both horizontal and vertical scaling methods in this section.

As migration is also a scaling action, elasticity can be obtained using migration scaling. Server consolidation/deconsolidation used in the cloud systems [103, 117, 134, 135] is a simple type of elasticity. It is used at cloud providers side to allocate and migrate VMs across physical servers to improve infrastructure utilization ratio. This technique guarantees the physical servers are well used, resulting in energy and cost saving for the providers, as long as the SLA is satisfied. A consolidation manager regularly monitors resource usage of PM/VM and their workload. The elasticity can be obtained by collocating VMs to the same PM which best fits to the workload or migrate VMs to bigger nodes which might serve better for load-balancing. The PM can be scaled by turning on, turning off or resuming which are forms of scaling out/in. Although elasticity actions for server (de)consolidation is often on two levels of resources (PM and VM), they consider only coarse-grained resource types. Thus minimizing the VM migration cost is also an issue of server consolidation. Using multi-level fine-grained elasticity approach, (de)consolidation can be performed on more fine-grained resource types such as lightweight container. The elastic VM also should be used to decrease number of migrations, thereby mitigate migration cost.

Autonomic mechanism: Consolidation manager
Resource Type: PM and VM
Resource granularity: PM scaling out/in, VM migration

Another work related to consolidation is an elasticity solution based on Entropy system [118]. Entropy [119] is a resource manager providing deployment and migration of VMs in clusters. It calculates migration plans based on many of inputs and constraints to optimize energy consumption. As Entropy depends on a constraint solver to satisfy the constraints and conditions, it takes time to pack and migrate a large number of VMs. To resolve this issue, the elastic Entropy system uses Entropy workers to manage random groups of Entropy instances (installed in each VM). An Entropy server provision these Entropy workers to ensure elasticity for Entropy system. This implementation brings to a significant improvement in performance as well as in reducing error rates in comparison to the traditional Entropy system. Like other consolidation solutions, the elasticity actions is mainly on PM and VM. Noting that the Entropy worker is introduced as an object for scaling, but in fact elasticity actions are put on VM (1 Entropy worker hosted in 1 VM). This costs a whole VM for Entropy worker which occupies space of regular

Entropy instances. Thus our approach suggests a more fine-grained approach by wrapping the Entropy workers into lightweight containers. This saves room for the Entropy instances.

Autonomic mechanism: Entropy server integrating the MAPE-K autonomic loop
Resource type: PM and VM
Resource granularity: PM scaling out/in, VM scaling out/in (only Entropy workers)

Nguyen et al. [120] consider three levels of resources representing by PM, VM and application environment (AE) in a research about resource management. The authors of this paper use a Local Decision Module (LDM) associated to an AE to monitor the AE itself. Based on information such as system load and available resource, the LDM adds or removes VMs to scale the AE accordingly. On the other hand, a Global Decision Module are responsible for installing the AE into VMs and VMs onto physical servers. Migration plan is calculated based on resolution of constraints using constraint solver, thus somewhat retards the elasticity actions. Like the elastic Entropy, the AE is a kind of resource (software container). However the LDM cannot scale the AE itself, the elasticity action is indeed put on VM. A lightweight container to wrap the AE could be a good solution in this situation.

Autonomic mechanism: Decision Modules
Resource type: PM and VM
Resource granularity: PM scaling out/in, VM scaling out/in

SmartScale [121], Cloudify [15] and soCloud [136] are Cloud platforms supporting elasticity. They perform both horizontal and vertical scales by changing number of VMs and adding/removing resources assigned to these VMs. The optimal VM size is calculated based on the corresponding throughput of application and then the platforms specify number of instances to optimize the throughput utilization. Unlike SmartScale, both Cloudify and soCloud can do scaling at software component level which leads to a shorter scaling time. Its components follow principle of the Service Component Architecture (SCA). However workload isolation among components is not mentioned in the soCloud paper.

Autonomic mechanism: Cloud platform integrating MAPE-K autonomic loop
Resource type: VM, VM resources and Software Component (only soCloud)
Resource granularity: VM scaling out/in, VM scaling up/down, Software Component out/in (only soCloud)

CRAMP [122] is a cost-efficient resource allocation solution for multiple web

applications with predictive scaling. Scaling algorithm of CRAMP predicts average load of application servers to add or remove VMs of the application server tier as well as deploy or undeploy web applications from the VMs. Isolation is performed by wrapping these applications into OSGi bundles [28] running on Apache Felix OSGi container [123]. This solution takes advantage of bundle portability which helps deploy and undeploy the bundles easily. Moreover, OSGi container can manage bundle versioning, thus switching between file configurations of different bundle versions can be done unopposed. However, an OSGi container like Felix only manages bundles installed inside it. It is blind with OSGi bundles located in other containers (same host) or in other hosts. Therefore, replication or migration of OSGi bundles across multiple hosts (VMs or PMs) is a challenging task. Because of this reason, CRAMP can only deploy and undeploy OSGi bundles locally for scaling. Furthermore, OSGi is a kind of container dedicated to Java applications, thereby it limits flexibility of the solution.

Autonomic mechanism: Global controller and Resource Allocator
Resource type: VM and Software Container (OSGi)
Resource granularity: VM scaling out/in, simple Software Container out/in

We know that software consolidation (collocated software) should be applied to SaaS/PaaS cloud model as an efficient energy and cost saving solution [124]. However if multiple applications run on the same host, it might cause the resource contention (CPU, memory or network bandwidth). Therefore the isolation and security of collocated applications are important things. If CRAMP uses OSGi container to isolate application, He et al. [125] introduces Elastic Application Container (EAC) as a lightweight solution to wrap applications into elastic containers which are managed by Elastic Application Server (EAS) in each host (PM or VM). An EAC do not require a guest OS to run applications like a VM does, thus it is a more lightweight approach than VM. The authors claim that using the EAC reduces overhead which is an inherent characteristic of VM-based approach as well as shortens migration time in comparison to both static and live VM migration. We see that the elasticity solution targets both VM and application levels. However scaling mechanism for VM is mentioned very briefly. At application level, it indirectly affects applications through a type of lightweight container (the EAC). Although EAC has its own set of resources but the vertical scaling of EAC is not mentioned in the paper.

Autonomic mechanism: EAC Cloud controller
Resource type: Lightweight Container (EAC)
Resource granularity: Lightweight Container out/in and migration

By dividing service into smaller pieces, Mohamed et al. [127] introduce Intelligent Managed Micro-Container (IMMC) that allows migrate or replicate small pieces of software among VMs in the Cloud. The pieces of software packed into the micro-containers are enhanced with the resiliency of cellular organisms assuring FCAPS (fault, configuration, accounting, performance and security) constraints described for each service.

Autonomic mechanism: IMMC deployment framework
Resource type: Software Container (Micro Container)
Resource granularity: Software Container scale out/in and migration

Imai et al. [126] also provide a solution for elasticity in the Cloud on two levels of resources (VM, software container) using both VM and application migrations. The authors focus mainly on migration and consolidation at application level using SALSA mobile actors. By using actor migration, applications composed of SALSA actors can be easily reconfigured at runtime regardless of the network type. Containerization of an application or a part of application (i.e. software components) is performed through these actors which have a similar role like OSGi container in the CRAMP solution. However this solution requires rewriting applications in SALSA actor language, which is always a time-consuming and error-prone task.

Autonomic mechanism: SALSA monitor and Node manager
Resource type: VM and Software Container (SALSA Actor)
Resource granularity: VM scaling out/in, Software Container migration

In summary, the mentioned related work all have drawbacks which could be improved by multi-level fine-grained elasticity. Thus our approach is a necessary complement to obtain the rapid and efficient elasticity.

4.3 Requirements of Autonomic Elasticity Managers

To support implementation of elasticity solutions on the Cloud, it needs to have research on autonomic elasticity managers which satisfy three most basic requirements as follows [145].

Autonomy: In the cloud environment, applications are often consisted of multiple software components. Moreover workloads to each of components are various and unpredictable. An effective elasticity solution must follow closely the fluctuation of the load to have timely responses. Such a mechanism cannot be operated manually. Therefore an autonomic manager should exist with sensors

and effectors on its managed elements. This autonomic manager should be able to cover all operations of the MAPE-K reference control loop (monitor, analyze, plan, execute and knowledge).

Scalability: is a prerequisite of elasticity. It sustains increasing of workloads using additional resources. However it does not consider temporal aspects of how fast, how often, and at what granularity scaling actions could be performed. It does not consider how the actual resource demands are fitted well to the provisioned resources at any point in time [62]. Anyway the cloud applications being provided and their resources should be able to scale. There are multiple scaling mechanisms for elasticity, which have been discussed in Chapter 3.

Adaptivity: Workloads of cloud applications are dynamic. An autonomic elasticity manager should be able to adapt dynamically to the changes of workload and system models. The adaptation reflects closely to the current demands, avoiding the over-provisioning unnecessary resources or under-provisioning required ones.

These three requirements are necessary and general for almost elasticity approaches. With a platform supporting multi-level fine-grained elasticity, extra requirements must be fulfilled.

Rapidness: An autonomic elasticity manager should compute and provision the required resource capacity fast enough to respect the QoS requirements. However these operations must be done accurately enough. An operation preserves the QoS requirements are always better than an optimal decision which takes longer to run. The rapidness of elasticity relates closely to terms of Speed and Accuracy which are defined as follows.

Speed: speed of a scaling up is the time needed for an elasticity system to escape from its under-provisioned state. Likewise speed of a scaling down is the time it takes to escape from the over-provisioned state. The speed relates closely to techniques used for scaling actions.

Accuracy: accuracy of scaling is the absolute difference between allocated resource and actual resource demand.

Component fine-grained hierarchical description and deployment: Composability is one of challenges of elasticity as pointed out in Section 3.4. An autonomic elasticity manager should be able to describe cloud applications as set of interconnected components which might have containment or runtime relationships. With its autonomy it should deploy applications automatically and resolve dependencies and constraints dynamically. Scaling actions may come simultaneously, thus software components could be deployed concurrently. Elasticity actions should prioritize more fine-grained components because deployment of smaller components saves cost and reduces overhead.

Multi-Cloud Deployment: Elasticity actions could be expanded to multiple software components and infrastructures which might be implemented on differ-

ent types of clouds. However it is a hard problem because the heterogeneous cloud implies no uniform and standard. Therefore an autonomic elasticity manager must own a language used to describe not only PaaS or SaaS services but also cloud infrastructures that PaaS and SaaS run on. By this way, the elasticity actions spanning multiple clouds (e.g. migration scaling) will be performed easier.

Genericity: Cloud applications are intrinsically complex and cloud ecosystem is heterogeneous. The descriptive language used in an elasticity solution has to be generic enough to abstract various resource types and their relationships. By this way, we can reuse definition of elasticity objects and need not to spend much effort for design the elasticity rules or policies.

Extensibility: A growing number of PaaS and SaaS services have been being integrated into cloud environments. Many new cloud providers introduce and launch their own IaaS services. Internet of Things (IoT) aims to bring myriad "things" to the Cloud context. Solutions to monitor autonomic system are constantly evolved as well. Elasticity solutions will also evolve continuously to adapt to those changes. Thus an autonomic elasticity manager should have an open mechanism to support extensibility when needed.

Roboconf platform such an autonomic elasticity manager supporting multi-level fine-grained elasticity approach will be detailed in next chapters as our contribution to this thesis.

4.4 Synthesis

In this chapter, we propose and detail our novel multi-level fine-grained approach for elasticity. This approach allows to resolve open issues of current elasticity solutions. A representative range of related work have also discussed. Requirements of an autonomic elasticity manager to efficiently support our approach have been pointed out. In the next chapters (Chapter 5 and 6), we will go to our second contribution in describing the Roboconf platform, an ACCS operating as an elasticity manager.

Chapter 5

MODEL OF ROBOCONF APPLICATIONS

Contents

As introduced, Roboconf is a platform supporting multi-level fine-grained elasticity. In this chapter, we detail Roboconf in providing a Domain Specific Language (DSL) to design complex distributed applications in multicloud as the first sub-contribution of our second contribution. Such applications are ready to be applied elasticity actions of the multi-level fine-grained approach. An application model and its concepts will also be presented.

5.1 Introduction

Today, the multi-cloud computing appears as a promising paradigm to support distributed applications in large scale. Multicloud computing is the use of multiple independent cloud environments that do not require a priori agreement between cloud providers or third parties. The design, portability and deployment of distributed applications in a multi-cloud environment is a very complex task and a real challenge, because there is not an uniform, simple and complete way to design applications that will run in heterogeneous cloud environments. Developers of cloud applications spend a lot of time to prepare, install, and configure their applications. In addition, after the development and deployment, applications can be moved from one cloud provider to another, which leads to problems of portability of applications. Currently, the state of the art has shown that there is no consensus on the style and architecture model supporting the development of distributed applications in the multi-cloud environment. These works have clearly shown considerable gaps to provide a simple and effective solution to address major challenges in designing distributed applications for a multi-cloud environment.

In this chapter we provide an overview of the design model of a Roboconf application. Then we present the problems and needs related to the design of it. This chapter presents the model used to support the design, specification, implementation and assembling Roboconf applications. A Roboconf application can be composed of one or multiple components. The model for Roboconf applications can describe each execution unit and the configuration properties associated with them. This model specifies the organization and the dependence between components within the architecture of a Roboconf application. Non-functional requirements (i.e. configuration constraints, placement, elasticity, hardware and software features) can be expressed on each execution unit via the model. We implement our approach by introduction of a motivating use case in the next section.

5.2 A Motivating Use Case

Let consider a company which wants to enjoy the benefits of Cloud computing. This company has an e-commerce application represented here by the RUBiS benchmark [141]. RUBiS is a JEE application based on servlets, which implements an auction web site modelled after eBay. RUBiS defines interactions such as registering new users, browsing, buying or selling items. To run this application, the administrator of company decides to use a web server provided by Apache HTTPD, an application server provided by Tomcat, and a set of database servers provided by MySQL. Apache relies on “mod_jk” connector to forward requests to Tomcat, via its AJP 13 connector. Let’s consider a scenario where the

company has the following requirements. Most of its clients (users who connect to its application) are located on the one hand in France (near Marseille) and on the other hand in Brazil (Sao Paulo). The company organizes data into two categories: business-critical (e.g. those which concern money) and non-critical (e.g. those which concern sold items). The former must be located on company premises, which is composed of a virtualized machine (provided by VMware vSphere) and a native machine. The virtualized machine runs the database which is in production while the native machine runs a backup. Regarding non-critical data, they are hosted (with the other application components) within any public clouds (the most cheapest one which is near the clients of company). The company capitalizes on competition among cloud providers and fully benefits from them. According to VM prices charged on cloud market, the company runs its application within two distinct clouds: Amazon EC2 and Microsoft Azure. Concerning the administration of the application, the administrator of company practices a fine-grained administration such as manipulating a .war package in a Tomcat server or a servlet in the .war package which is in the Tomcat container.

Furthermore, sometimes he needs to deploy an entire stack or just a part of the stack. For example, in the case of an intrusion on the VM hosting a Tomcat application server, does he need to redeploy the overall stack? If the intrusion is at the .war package level, only the deployment of the corresponding package and its servlets are needed. If the problem comes from a single servlet, only this servlet should be taken into account. Another need is the reconfiguration of the application, partly or entirely during its lifetime, especially in moving a portion from a cloud to another one. Figure 5.1 depicts this scenario. This example depicts a trend as to be shown in the 2014 State of the Cloud Survey, “the hybrid and multi-cloud implementations continue to be the end goal for the enterprise: 74% of enterprise respondents have a multi-cloud strategy, and 48% are planning for hybrid clouds.” [142].

In summary, this practical use case intuitively points out the following features from the ACCS which attempts to conveniently administer it. (1) The ACCS should be able to provide both hybrid and multi-cloud deployment features, with the target clouds unknown in advance. (2) It should provide a hierarchical language for expressing the use case in order to allow a fine-grained administration.

5.3 Model of Roboconf Applications

Roboconf is designed to see a distributed application as a set of “components”, and as a group of “instances” of these components. Let’s take as an example the three-tier distributed application “Apache-Tomcat-MySQL”. “Apache” is a component, while an installation of Apache on a particular machine is an instance.

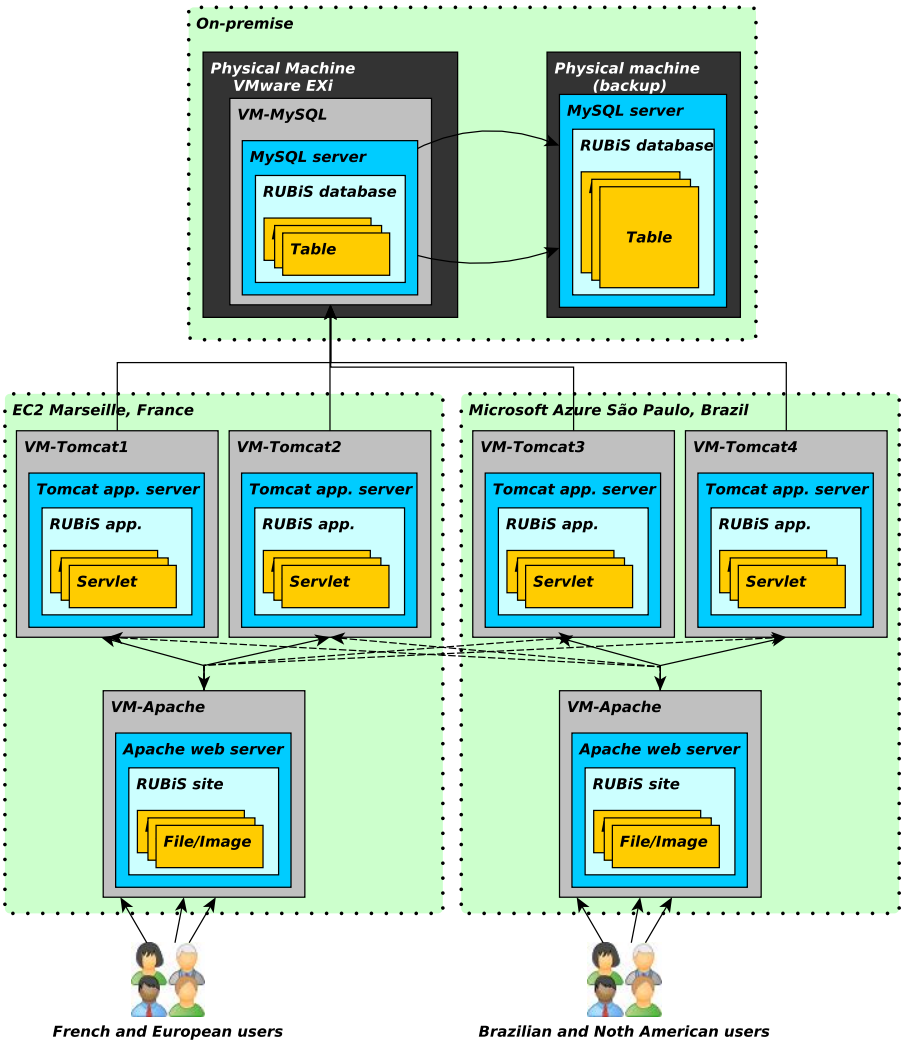

Figure 5.1: A multi-cloud deployment of the RUBiS benchmark

Another installation of Apache on another machine is another instance. Besides, Roboconf is built to see distributed application as a group of components that each one exchanges a group of simple data among each other. Data can be string or structured data. Components of a distributed application are composed of variables as for example the IP address or the port used. Parts of those variables may be needed by other components of the application, they are named “exported vars”, while variables coming from other components of the application are named “imported vars”. Moreover, definition of a component can be inherited by definition of another according to object-oriented design. It inherits all import/export

vars and default values. For instance, Tomcat component can inherit properties of a generic "Application Server" component. Now that we have a far view of an application, let's explain more precisely what are its components. In the above example, Apache in this case simply imports variables coming from Tomcat: the IP and port of application server. As we said earlier, we define elements (component and instance) as having a set of properties, and having exported and imported variables. In this case of Apache there are only imported variables. A sample of Apache component under Roboconf descriptive language could be as the following:

```
Apache { # Apache Load Balancer (a comment)
  installer: puppet;
  imports: Tomcat.portAJP, Tomcat.ip;
}
```

This small portion is made up of several regions. The *installer*: A component property mandatory and designates the Roboconf plug-in that will handle the life cycle of component instances. In this example, we are using "puppet" implementation. The *imports* lists the variables this component need to be resolved before starting. Variable names are separated by commas. They are also prefixed by the component that exports them. As an example, if Tomcat exports the ip variable, then a depending component will import Tomcat.ip. On the other hand, MySQL does not import data from other components, it is the one exporting data which are its IP and port. A definition of MySQL could be as following:

```
MySQL { # MySQL database
  installer: bash;
  exports: ip, port = 3306;
}
```

Here has a minor different from the *exports* which lists the variables this component makes visible to other components. The "ip" is a special variable name whose value will be set dynamically by Roboconf. All the other variables should specify a default value.

In terms of model and configuration files, Roboconf has the following concepts. The **application descriptor** contains meta-information of the application such as name, version qualifier and description. The **graph** is in fact a set of graphs. It defines software components which range broadly from the (virtual) machine, cloud platform to the application package. The graph defines containment relations and runtime relations. Two kinds of relations are defined as follows: (1) Containment means a component can be deployed over another one. As an example, a Tomcat server can be deployed over a VM. Or a web application (.war) can be deployed over a Tomcat server. (2) Runtime relations refer to

components that work together. For instance, a web application needs a database. More specifically, it needs the IP address and the port of the database. Generally, this information is hard-coded. Roboconf can instead resolve them at runtime and update components through the configuration or management APIs (e.g. JMX, REST). As an example, Apache, Tomcat and MySQL can be deployed in parallel. Tomcat will be deployed but will not be able to start until it knows where is the database. Once the database is deployed and started, Roboconf will update Tomcat configuration so that it knows where is MySQL. This is what runtime dependencies make possible. If the graph defines relations between components, **instances** represent concrete components. Like a Java class, a Roboconf component is only a definition. It needs to be instantiated to be used. Predefined instances aim at gaining some time when one wants to deploy application parts. As an example, the deployer could have defined a Tomcat component in the graph, and have four instances, one deployed on machine A, and another on machine B and other two on machine C. These would be four instances of the same component. The rules that apply to them are deduced from the graph, but they have their own configuration.

Roboconf is also designed to see an application as hierarchy of components. The main motivation of hierarchy is to allow Roboconf to exactly keep track of where instances are implemented in the system. It helps Roboconf to make right decisions in dynamic deployment as mentioned in the motivating example. A natural example of parent/children relationships among components of an OSGi application is depicted following:

```
# An Azure VM
VM_AZURE {
  installer: iaas;
  children: Karaf;
}
# Karaf: OSGi container
Karaf {
  installer: bash;
  exports: ip, agentID = 1;
  children: Joram, JNDI;
}
```

```
# Joram: OSGi JMS service
Joram {
  installer: osgi-bundle;
  exports: portJR = 16001;
  imports: Karaf.agentID,
           Karaf.ip;
}
# JNDI: naming service
JNDI {
  installer: osgi-bundle;
  exports: portJNDI = 16401;
  imports: Karaf.agentID,
           Karaf.ip;
}
```

There is a new important field: *children* which lists the components that can be instantiated and deployed over this component. In the example above, it means we can deploy Karaf over a VM instance. In turn, Joram and JNDI can be deployed

over instances of Karaf. While hierarchical model resolves the containment relations (i.e. vertical relationship) and the export/import variables model responsible for disentangling the runtime relations (i.e. horizontal relationship) among components, a bi-color Graph put everything together in a DSL introduced more details in next section. At runtime, the Graph is used to determine what can be instantiated, and how it can be deployed. Software components include the deployment roots (e.g. VMs, devices, remote hosts), databases, application servers and application modules (e.g. .war, .zip, etc). They list what the deployers want to deploy or possibly deploy. What is modelled in the graph is really a user choice. Various granularity can be described. It can goes very deeply in the description (Figure 5.2) or bundle things together such as associating a given .war with an application server. Multi-IaaS is supported by defining several root components. Each one will be associated with various properties (e.g. IaaS provider, VM type, etc.). It is worth noting that an instance in a hierarchy can be located using an absolute path in the Roboconf application model. For example, the "Joram1" application instance running on "Karaf1" OSGi container inside a "VMEC2" virtual machine can be referred to by the path "/vmec2/karaf1/joram1".

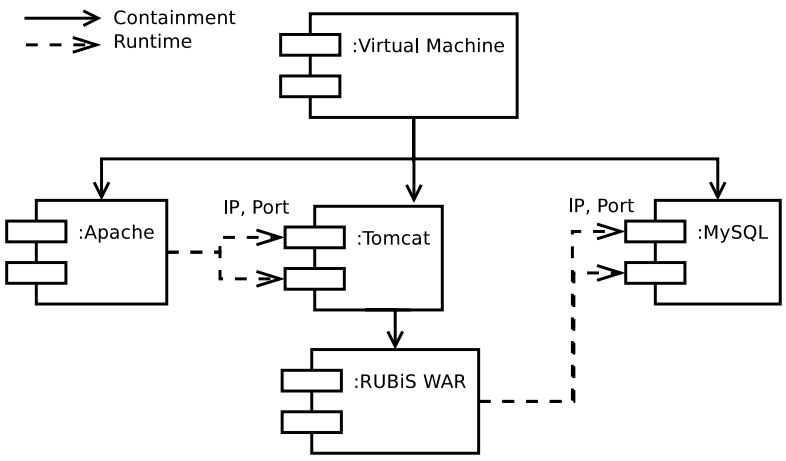

Figure 5.2: Illustration of a fine-grained description of components

5.4 Description of Roboconf Applications

5.4.1 The Application Descriptor

A Roboconf project must contain a descriptor directory with an application descriptor inside. This descriptor must be named "application.properties". It contains basic information related to the application itself. A example of application descriptor for the motivating use case as follows. Properties are explained in the comments.

```
# Application Descriptor for Roboconf

# The application name (required)
# It should be readable by a human.
application-name = RUBiS-J2EE

# The application qualifier (required)
# It can be a version number, a build ID, whatever.
# It should be readable by a human.
application-qualifier = snapshot

# The description of application (optional)
application-description = The description of \
                          your application

# The DSL ID (optional)
# For the moment, Roboconf only knows 1 DSL,
# but it may support other standards or definitions later.
application-dsl-id = roboconf-1.0

# The main graph file (required)
# A graph definition may contain several files.
# This property indicates the one to read first.
graph-entry-point = main.graph

# The main instances file (optional)
# An instance definition may be made up of several files.
# This property indicates the one to read first.
instance-entry-point = definition.instances
```

5.4.2 The Graph

A Roboconf project must contain an graph directory with the definition of a graph. As a reminder, like a Java class, a Roboconf component is only a definition. It needs to be instantiated to be used. By convention, graph definitions are expected in files with the "graph" extension. A component starts with the component name, followed by an opening curly bracket. Components can be defined in any order. Components of the RUBiS example may appear as follows.

```
# The VM
VM {
    installer: iaas;
    children: MySQL, Tomcat, Apache;
}

# MySQL database
MySQL {
    installer: puppet;
    exports: ip, port = 3306;
}

# Tomcat
Tomcat {
    installer: puppet;
    exports: ip, portAJP = 8009;
    imports: MySQL.ip, MySQL.port;
}

# Apache Load Balancer
Apache {
    installer: puppet;
    imports: Tomcat.portAJP, Tomcat.ip;
}
```

Components can be defined in separate files and in any order. This is achieved thanks to the "import" keyword. Graph definitions can mix imports and components declaration, or, it can only contain imports.

```
import graph-part-1.graph;
import graph-part-2.graph;

MyComponent{
    # whatever
}
```

Import can also be marked as optional. In this case, the instance will be able to start even if the imported variables are not resolved. As an example, in a cluster mode, a cluster member may need to know where are the other members.

```
ClusterMember {
    exports: varA, varB;
    imports: ClusterMember.varA (optional),
        ClusterMember.varB (optional);
}
```

It is also possible to group imports thanks to the wildcard symbol. As an example, the previous imports could simply be written as follows.

```
imports: ClusterMember.* (optional);
```

Besides, "extends" property indicates this component extends another component. A component that extends another one inherits its exports, its imports, its installer and its recipes. An extending component can override the value of an inherited element. It can also add new variables (for export or for import). It cannot remove inherited elements.

The graph model allows to define multi-container and distributed topologies. We can go from the machine (VM, device...) to an application component (e.g. a Web Application). However, and even if the model could support it, it is not considered as a good practice to define system requirements as graph nodes. If an application server needs a JVM or a library to run (such as Python), deployer should not rely on Roboconf to install it. It is not that you could not achieve it with a Bash script or something else. But it may be better to pre-install and configure such dependencies directly in the virtual images. As an example, there are mechanisms in Java Virtual Machines such as endorsed and policies that would be painful to configure with Roboconf. And, again, the problem here is not Roboconf, but the way the deployer would implement it with a Roboconf plug-in (Bash, Puppet, etc.). This has been experimented with NodeJS application. Write a Bash script that respect Roboconf requirements (be idempotent) that installs NodeJS and NPM was quite painful to do. Pre-installing them on virtual images was much more convenient. System requirements should not be deployed with Roboconf. They should be deployed and configured in the virtual images.

5.4.3 Instance Description

A Roboconf project may contain an instance directory with the definition of instances. By convention, instance definitions are expected in files with the "instances" extension. An instance starts with the 2-word keyword "instance of"

followed by the name of the component. As mentioned, instances must be defined hierarchically. A concrete example of an instance of VM component which hosts an instance of Tomcat component is depicted as follows. It notes that the Tomcat default port is overridden in this case (8081 instead of 8080).

```
instance of Vm_Tomcat {
    name: Tomcat VM1;

    instance of Tomcat {
        name: Tomcat;
        port: 8081;
    }
}
```

The Tomcat instance is defined inside a VM instance. Defining it else where would result in an error. Like components, instance definitions can be split into several files. It is indeed possible to import other instance definitions. Instances definitions may only contain imports or may mix imports and instances declaration as follows.

```
import database.instances;
import servers.instances;

instance of MyApp {
    # whatever
}
```

5.4.4 The Graph Resources

The graph definition has already been introduced. However, it does not answer the question: how does Roboconf deal with deployment configuration? Clearly, the graph model is artificial, thus one can put whatever he wants in it. The real logic processing will be handled by a Roboconf plug-in (e.g. Bash or Puppet). These plug-ins will use resources, associated with a component, to deploy real software. Therefor, for every component in the graph, a project must contain a sub-directory with the same name than the component. The file structure for the RUBiS application looks like follows.

```
MyApp
├── descriptor
│   └── application.properties
├── graph
│   ├── main.graph
│   ├── VM
│   ├── Tomcat
│   ├── MySQL
│   └── Apache
└── instances
    └── ...
```

These sub-directories will contain the component resources. These resources will be used by the Roboconf plug-in identified by the installer name. The kind of resources to put depends on the plug-in. As an example, if the Tomcat component uses the puppet installer, then its resource directory will contain a Puppet module. If it uses the bash installer, then it will contain bash scripts. Plug-ins are extensions for Roboconf, thus we can imagine various type of plug-ins. Roboconf plug-ins are discussed more detailed in Section 6.2.3. Writing the plug-in configurations is what takes most of the time. Roboconf includes a Maven plug-in so that people will be able to reuse graph model and resources among projects.

5.5 Roboconf Domain Specific Language

5.5.1 Configuration Files and Fine-grained Hierarchical DSL

As mentioned in Section 5.4.4, an application deployed by Roboconf should provide at least three files. The first is a descriptor application file containing the main Roboconf configuration. This file describes the application itself such as name, description, location of the main model files, etc. Second one is an acyclic graph describing both vertical and horizontal relationships among components of the application. The components can be software components to install, or VMs to deploy on, etc. Along to the graph file, users also need to provide all resources necessary to deploy the component (e.g. scripts, software packages, configuration files). The IaaSes on which the VM will be created are also defined in the resource directories of components in the corresponding "target.properties" files. We can choose VMs with pre-defined configuration such as "m1.large" of Amazon EC2 or "Standard A2" of Microsoft Azure. Or we can customize to create dedicated configurations in the case of private clouds or on-premise hosts. On runtime, Roboconf will provision the VMs based on these defined configurations.

An example of the target.properties files for EC2 IaaS can be as follows.

```
# These properties are specific to Amazon Web Services
target.id               = ec2
ec2.endpoint            = eu-west-1.ec2.amazonaws.com
ec2.access.key          = YOUR_EC2_ACCESS_KEY
ec2.secret.key          = YOUR_EC2_SECRET_KEY
ec2.ami                 = Your AMI identifier (ami-...)
ec2.instance.type       = t1.micro
ec2.ssh.key             = Your SSH Key
ec2.security.group      = Your Security Group
```

Figure 5.3a depicts components of the motivating use case described in hierarchical and fine-grained manner using Roboconf DSL. Final one is an instance file that lists all the initial instances. It means graph components will be pre-instantiated, ready to be deployed. The fact the instances are defined does not mean they will be deployed or started automatically but they will be already defined and configured. In this file, the instances must be defined hierarchically. If the graph defines a root component R with a child C, then an instance of C must be defined in an instance of R. The instance may also declare properties to override component properties. As an example, if a Tomcat component exports a port property with the default value 8080, the instance may override it (e.g. with 8081). An example of this file for the three-tier application is found in Figure 5.3b where an instance of Apache, one instance of MySQL, two instances of Tomcat and one instance of Rubis deployed on different clouds. Roboconf provides a DSL which is inspired from the CSS grammar. It was preferred over XML (easy but heavy), JSON (not user-friendly) and YAML (error prone when many levels of indentation). Its main force is to keep the thing simple, with the minimal set of characters to write. We developed an Eclipse plug-in operating as an editor providing semantic checking and syntax highlighting for the Roboconf DSL. So far, the system can understand the distributed application that users want it to deploy. The details about deployment process is discussed in next two subsections.

5.5.2 Reusability in the Roboconf Model

This section explains several ways of creating reusable graph portions. It is quite helpful to use and write reusable recipes. As a user, it is convenient to not have to write all the application description and all the recipes for its components. Roboconf has solutions to ease reusability. The first option is that a component can extend another one. It means the extending component will inherit all the properties and the recipe of the component it extends. We can extend the MySQL component to have a similar component in the RUBiS example as follows.

```
MySQL {
    exports: ip, port = 3306;
    installer: puppet;
    # There is a directory called
    # MySQL with a Puppet module.
}

My-Client-Database {
    extends: MySQL;
    exports: port = 3307;
    exports: username = something;
}
```

In this example, we have two components which are exactly the same, except they do not have the same name. This helps to distinguish them in terms of roles and behavior. We can associate different rules about monitoring and autonomy. An extending component can also override property values. In the My-Client-Database component, the default value of the port variable is 3307 instead of 3306. And as usual, instances can override the default values of their component. Extending components can also define new variables such as the "username" variable.

It is possible to split graph definitions into several files. For every application, there is one main file. By using the import keyword, this main file can import other definitions. These definitions can be in the same application or they can be in another Roboconf project. Other Roboconf projects can be included by using Roboconf Maven plug-in. Local imports are defined as follows.

```
import file-name;
import dir1/dir2/file-name;
```

Remote imports are only handled through the Roboconf Maven plug-in. Remote imports rely on Maven artifacts and repositories. This may seem constraining. However, Maven is a usual build tool, widely used and with lots of integration here and there. An important thing to highlight is that remote imports are resolved and copied in the application at build time. So, a packaged application contains all the definitions and the recipes it needs. This guarantees that if the Roboconf configuration is released, we will be able to take it in 10 years and it will still be working. No matter if the repositories of remote imports disappeared. Remote import can be defined in Roboconf DSL as follows.

```
import application-artifact-id/file-name;
import application-artifact-id/dir1/dir2/file-name;
```

(a)

```
# An Azure VM
VM_AZURE {
  installer: iaas;
  children: Tomcat,
         Apache, MySQL;
}
# An EC2 VM
VM_EC2 {
  installer: iaas;
  children: Tomcat,
         Apache, MySQL;
}
# A VMware VM
VM_VMWARE {
  installer: iaas;
  children: Tomcat,
         Apache, MySQL;
}
# MySQL
MySQL {
  installer: puppet;
  exports: ip, port = 3306;
}
# Tomcat with Rubis
Tomcat {
  installer: puppet;
  exports: ip,
        portAJP = 8009;
  children: Rubis
}
# Apache Load Balancer
Apache {
  installer: puppet;
  imports: Tomcat.portAJP,
         Tomcat.ip;
}
# RUBiS WAR Application
Rubis {
  installer: java-servlet;
  imports: MySQL.port,
        MySQL.ip;
}
```

(b)

```
# A VM Azure with Apache
instanceof VM_AZURE {
  name: vm-azure-apache;

  instanceof Apache {
    name: apache;
  }
}
# A VM EC2 with Tomcat
instanceof VM_EC2 {
  name: vm-ec2-tomcat-1;

  instanceof Tomcat {
    name: tomcat-1;
    instanceof Rubis {
      name: rubis-1;
    }
  }
}
```

```
# A VM VMware with Tomcat
instanceof VM_VMWARE {
  name: vm-vmware-tomcat-2;

  instanceof Tomcat {
    name: tomcat-2;
  }
}
# A VM VMware with MySQL
instanceof VM_VMWARE {
  name: vm-vmware-mysql;

  instanceof MySQL {
    name: mysql;
  }
}
```

Figure 5.3: Example of a Roboconf DSL: (a) graph and (b) instance files for 3-tier deployment

Reusable parts can be shared inside our community. But it is also possible to define its own reusable parts (e.g. in a company that would like to enforce some practices and configurations). There is quite a lot of possibilities offered by this system. Roboconf recipes can be defined and maintained anywhere. Official recipes are hosted on Github, every recipe having its own Git repository. It is up to users to determine whether they want to use them, create new ones, or even, create their own recipe repository. This is indeed an option some organizations may retain.

5.5.3 Roboconf DSL Dedicated to the Rules of Elasticity

The rules of elasticity for a cloud application also can be expressed using the fine-grained Roboconf DSL. The DSL are languages associated to a specific problem domain. The language provides dedicated high-level concepts for the elasticity of Roboconf application. This language allows to easily express rich conditions and correlation of events. We can express rules over a period of time. These mechanisms and expressiveness of the simplified language reduce efforts to develop a elastic system able to respond to complex situations.

Concepts

Our elasticity DSL is a language guided by events which means that the system triggers actions based on events specified in the rules. Each event is associated with a context. The context of the event is transmitted to the rules associated with it. The conditions and actions of the rules can access information in the context of the event. The elasticity language based on two basic concepts:

- *Creation of events*: this is the specification of an event in the language of elasticity, which causes the Roboconf system to trigger whenever this event occurs. Figure 5.4 shows syntax to create an event in Roboconf DSL. This offers capability for the developer to create its own events to monitor.

- *Reactions to the events*: provide guidance to describe how to achieve certain elasticity goals when specific conditions are met.

```
[EVENT measure-extension measure-name]
[ measures ]
```

Figure 5.4: Syntax to create an event in Roboconf DSL

This command indicates an event triggering a measure that an agent will have to perform regularly on its machine. An agent can use several options to measure something. The option or extension used to perform the measure is indicated on the same line with the measure name. Each measure is performed independently of the others. It means every result matching the rule results in a message sent to the DM. The agent measures and notifies when needed and it has not to interpret these measures. This is responsibility of the autonomic modules of the DM. The measure-extensions currently supported includes LiveStatus, REST and File. The LiveStatus [35], which is the protocol used by Nagios [36] and Shinken [37], allows to query a local Nagios or Shinken agent. An agent also can query a REST service. The result can be interpreted as an integer or as a string. The local file system is the third one can be checked by the agent. Depending on the existence of a file or a directory, or based on the absence of a given file, a notification will be sent to the DM. Developers can create a new measure-extension and integrate it into Roboconf source code.

Syntax

Figure 5.5 shows the general syntax of an event reaction in our dedicated language.

```
[REACTION measure-name reaction-handler]
Optional parameters for the handler
```

Figure 5.5: Syntax of an event reaction in Roboconf DSL

A reaction is triggered when one or more conditions are met. These conditions are defined in the corresponding reaction-handlers. Measures which are sent from the measure-name event are a source to calculate comparable metrics such as response time, cpu usage, mem usage, etc. Table 5.1 lists operators used in the elasticity conditions. Roboconf provides an interface class for developers to create their own reaction-handlers or overwrite built-in ones. There are 5 available built-in reaction-handlers described as follows.

- **Log** is to log an entry without any parameters. It is used mainly for debug.
- **Mail** is to send an email. It accepts only one parameter which is an email address.
- **Replicate-Service** is to replicate a component instance (including instances of VM, container and service component) from a source to a destination.

Table 5.1: List of Operators

Operator	Numeric Only
=	no
==	no
<	yes
>	yes
<=	yes
>=	yes

Source and destination can be in the same machine depending on the absolute pathname of the replicating instance. It takes a series of source-destination couples of instance names as parameters. These couples are separated by commas. This reaction is usually used in the horizontal scale.

- **Delete-Service** is to undeploy and remove a component instance that is not necessary anymore or just been replicated. It takes an instance absolute pathname as parameter.

- **Expand-Service** is to expand a instance vertically such as adding CPU, memory to appropriate components (usually instances of compartment components such as VM, lightweight containers). It takes an instance absolute pathname, a resource unit (currently supported vCPUCore, vCPUShare, memUsage, diskUsage, networkUsage) and quantity of this resource unit (positive means extending, negative means shrinking) as parameters. This reaction is usually used in the vertical scale.

As an example, to remove a specific instance WebappA hosted in a container Tomcat1 running on a virtual machine VM1, we need to provide the absolute pathname (*/VM1/Tomcat1/WebappA*) of this instance as follow.

```
[REACTION low-RT-1 Delete-Service]
/VM1/Tomcat1/WebappA
```

The supporting for multi-level fine-grained elasticity

Roboconf owns a multi-level fine-grained DSL, thus the elasticity DSL is also inherited this characteristic. Consider our three-tier motivating application described in Section 5.2. At the execution, each component has specific needs of scaling as required by the developer. Depending on the vertical or horizontal elasticity type defined by the developer, the components can be individually resized

or creating more component instances in different VMs or by allocating more or less computing resources. Thus, the elasticity may be expressed at different levels by different types of users (developers, administrators).

Second, the developer needs to express the elasticity requirements at different levels of resource granularity, whose specifications will be applied at different levels, as opposed to the usual approaches in the allocation and reallocation of resources. To achieve this, the elasticity controls must be supported at different levels following:

- *Application level*: the requirements of elasticity can be applied on the overall availability of the application.
- *Software Component level*: the developer can specify different requirements in function of the component type. For example, the nature of the requirements for a processing unit component may be different from that of a front-end component.
- *Infrastructure level*: the system admin may have different concerns in optimizing their infrastructure by applying elasticity policies on PMs or VMs.

To demonstrate for the multi-level fine-grained elasticity language of Roboconf, we take the motivating example to demonstrate implementations of the horizontal and vertical built-in reaction-handlers as follows. The illustration of the reaction chain is shown in Figure 5.6.

[I] To replicate the entire stack of */VM1/Tomcat1/WebappA* (all three instances):

```
[REACTION high-RT-1 Replicate-Service]
/VM1/Tomcat1/WebappA /VM2/Tomcat2/WebappB
```

It is worth noting that if an empty VM2 already exists, it will be reused and "filled" with a new Tomcat2 containing a new WebappB. Otherwise, a new entire stack will be created.

[II] To remove a specific instance WebappB of the stack */VM2/Tomcat2/WebappB*, we need to provide the absolute path of this instance:

```
[REACTION low-RT-1 Delete-Service]
/VM2/Tomcat2/WebappB
```

[III] To replicate a specific instance WebappA of the stack */VM1/Tomcat1/WebappA* to under the Tomcat2 (*/VM2/Tomcat2/*) and name it WebappC:

```
[REACTION high-RT-2 Replicate-Service]
/VM1/Tomcat1/WebappA /VM2/Tomcat2/WebappC
```

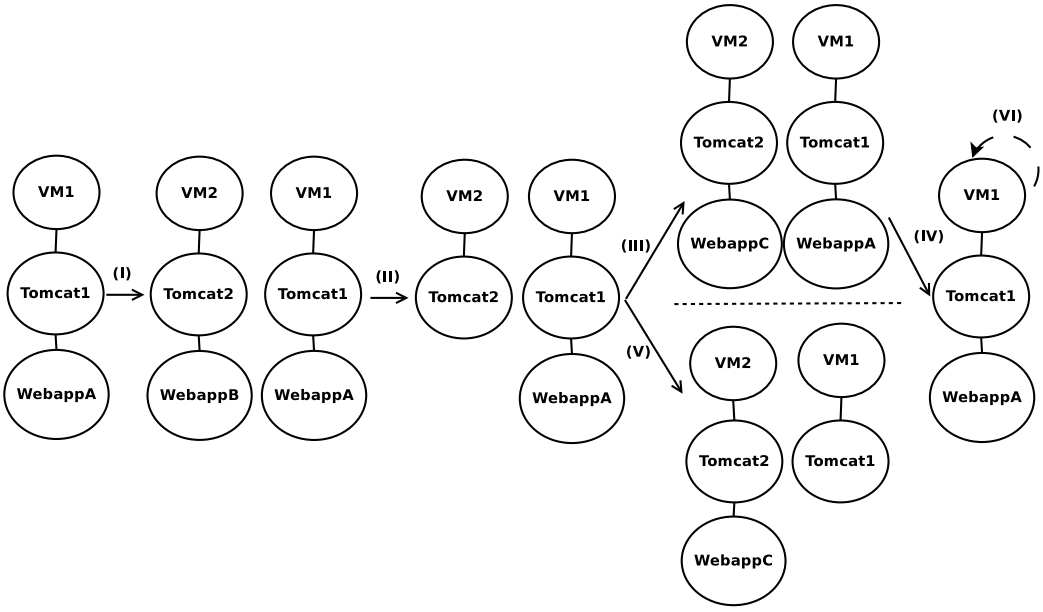

Figure 5.6: Illustration of reactions in a multi-level fine-grained manner

[IV] To remove the entire stack */VM2/Tomcat2/WebappC*, we only need to provide the absolute path of the root instance which is VM2 in this case:

```
[REACTION low-RT-2 Delete-Service]
/VM2
```

It is also worth noting that the VM2 and its children (Tomcat2, WebappC) will be gracefully stopped, undeployed and removed from the system orderly and automatically. In the case of migration, we combine both "Replicate-Service" and "Delete-Service" rules. For instance, after [III]:

[V] To migrate a specific instance WebappA of the stack */VM1/Tomcat1/WebappA* to under the Tomcat2 (*/VM2/Tomcat2/*) and name it WebappC:

```
# Replicate the WebappA first
[REACTION low-RT Replicate-Service]
/VM1/Tomcat1/WebappA /VM2/Tomcat2/WebappC
# Then remove the WebappA
[REACTION low-RT Delete-Service]
/VM1/Tomcat1/WebappA
```

[VI] To increase number of vCPU allocated for VM1 (adds 2 units):

```
[REACTION high-CPU-usage Expand-Service]
/VM1 vCPUCore 2
```

5.6 Synthesis

In this chapter, application model of Roboconf have been described. A Roboconf application is the composition of software components, infrastructure components (called target) and their corresponding instances and resources. We have also introduced the Roboconf DSL as a method to describe complexity and containment relationship of the applications. A part of Roboconf DSL dedicated to the rules of elasticity has been developed to fully support the multi-level fine-grained elasticity (and not only this approach). In the next chapter, detailed architecture and other elasticity-support features of Roboconf will be discussed.

Chapter 6

THE ROBOCONF PLATFORM

Contents

This chapter presents the second sub-contribution of our second contribution on architecture and implementing aspects of the Roboconf platform which not only allows to deploy, run and manage the Roboconf applications (presented in Chapter 5) but also supports multi-level fine-grained elasticity.

6.1 Introduction

In the cloud computing market, many operators provide different cloud services from infrastructure or IaaS services such as Amazon, Windows Azure, Rackspace to fully functional platform services or PaaS like Google App Engine, CloudBees, OpenShift. However, the heterogeneity of these platforms and infrastructure makes the deployment of service-oriented applications from one cloud provider to another difficult. It is the same for its management because these platforms require the use of proprietary APIs. Additionally, applications may have specific requirements of price, quality of service, availability, geolocation, databases, middleware. It is difficult for application developers to find an answer to all their requirements from a single cloud provider. Roboconf is a multi-cloud services platform that meets the requirements of autonomic elasticity manager identified in Section 4.3. Beside ability to describe any kinds of applications (genericity) and to concurrently deploy software components on multiple clouds, Roboconf implements the MAPE-K autonomic loop to support reconfiguration, scaling and adaptation. Owning mechanisms that supports for multi-level fine-grained elasticity, Roboconf brings rapidness to elasticity actions. Finally, Roboconf is also easy to extend with providing many interfaces for plug-in integration.

In next sections, we will describe Roboconf architecture and detail its modules. We present the deployment process and reconfigurability of the platform. Finally, we present Roboconf methods and mechanisms to manage elasticity as an autonomic system.

6.2 Architecture of the Roboconf platform

Roboconf is a distributed multi-cloud platform based on service-oriented architecture (i.e. SoA). The design of a distributed and robust system is a difficult issue. The architectural considerations therefore play an important role in the design of the Roboconf platform. The architecture covers the overall structure, organization, expression of nonfunctional requirements, management for both business applications and the platform itself.

In addition to the architectural problems, the main problems in the design of the platform come from the distribution. The interaction among software components is based on the communication layer. There exist different modes of interaction such as synchronous and asynchronous communications. Problems related to the composition and software components are central elements in the design of multi-cloud Roboconf platform, both for its structure and for business applications deployed on it. In a multi-cloud environment, the management in case of failure of Roboconf platform and its hosted applications raises the problem of

maintaining and persistence the coherence of the states. Life cycle management of applications includes functions such as configuration, deployment, monitoring, reaction to changes in the runtime environment, and reconfiguration. Spontaneous requirements and fast growing cloud environments lead to considering the automation (Autonomic computing) of management tasks. The architectural features of Roboconf platform go beyond what is currently offered. Thus the Roboconf platform provides an effective and simple way to deploy, run and manage multi-cloud applications. In addition, it supports mechanisms and approaches to satisfy the requirements posed for modern autonomic elasticity managers.

6.2.1 Design Details of the Roboconf Platform

Roboconf is a distributed solution to deploy distributed applications. It is an open-source software licensed under the terms of the Apache license version 2.0. It is a deployment framework for multi-cloud, but not only. It allows to describe distributed applications and handle deployment automatically of the entire application or a part of it. The objective of this framework is to be improvable with a micro kernel which is the core of Roboconf. This kernel implements all necessary mechanism to plug new behaviours for addressing new applications and new execution environments. Moreover, Roboconf supports scaling natively. Its main force is the support of dynamic (re)configuration. This provides a lot of flexibility and allows elasticity deployments. Roboconf is made up of several modules. A simplified drawing of Roboconf architecture is depicted in Figure 6.1 and explained more detailed as follows.

Deployment Manager: (or DM) is an application in charge of managing VMs and the agents (see below). It acts as an interface to the set of VMs or devices. New cloud environments can be integrated into Roboconf using **target handlers**. It is also in charge of instantiating VMs in the IaaS and physical devices such as embedded boards. It is responsible for implementing sub-modules of the Roboconf autonomic loop (see Section 6.4). The DM also offers an API using REST/JSON technology.

Agent: is a software component that must be deployed on every VM and device on which Roboconf wants to deploy or control something for bootstrapping. Agents use **plug-ins** to delegate the manipulation of software instances. The plug-ins can be life cycle management ones that support different implementation languages or frameworks such as Bash, Puppet, OSGi, Java, etc. It also can be a federated PaaS plug-ins such as a Heroku driver. Roboconf kernel is kept lightweight and the plug-ins can be flexibly plugged into the core. Roboconf agents communicate with each other through an asynchronous messaging server.

Messaging Server: is the key component acting as distributed registry of import/export variables that enable communications between the DM and the agents.

Roboconf includes the message definitions, the interface to interact with a given messaging server and their implementations. The DM and the agents always communicate asynchronously through this server. So far RabbitMQ [32] is the only messaging server used for Roboconf but any AMQP [33] messaging ones can be a candidate.

Instance Manager: is developed as a Roboconf module to delegate software life-cycle management on different software platforms and monitor software instances themselves.

Artifact and Image repositories: are responsible for distribution of software packages (i.e. artifact) and images (VM or container), respectively. Artifact repositories can be managed locally or retrieved from public repositories such as Maven center or NPM. Image repository is a database to map each required image of each target to corresponding infrastructure components. The required image can be an image available in the image marketplace provided by IaaS or a pre-built one created manually or automatically (e.g. using Dockerfile [30] or Vagrantfile [31]).

DSL: is a domain specific language developed to describe components of a Roboconf application and relationships among them (see Section 5.5). A part of Roboconf DSL dedicates to describing the rules of elasticity.

Admin console: is required to control the entire platform mainly through the DM. Roboconf comes with a shell-based console and an AngularJS web application providing various user interfaces to interact with the DM through REST. It contains utilities to transform Java beans into JSON.

6.2.2 Roboconf Targets

As mentioned, Roboconf is about applications. Applications are made up of instances, each instance having a component which defines its behavior with respect to other components. Some of these components designate machines, being virtual or physical. Lightweight containers such as Docker are included in this definition as well. These specific "scoped" instances recognized by the installers of their components are called **targets**. Roboconf can deploy an application on various targets. This includes cloud infrastructures (IaaS) or other kinds of targets. It can even deploy a part of an application on a given IaaS and some other parts on other IaaS (cloud bursting). This makes Roboconf suitable for hybrid deployments. Thus, an applications can be deployed in the cloud, in the self-hosted information system or even on the embedded systems (e.g. connected devices, Internet of Things). Target features may vary from one infrastructure to another. These features are implemented depending on requirements of users. Obviously, they can be extended if necessary.

To create, delete, and more generally, manage the life cycle of such a machine, Roboconf target installer needs additional information. As an example, creating a

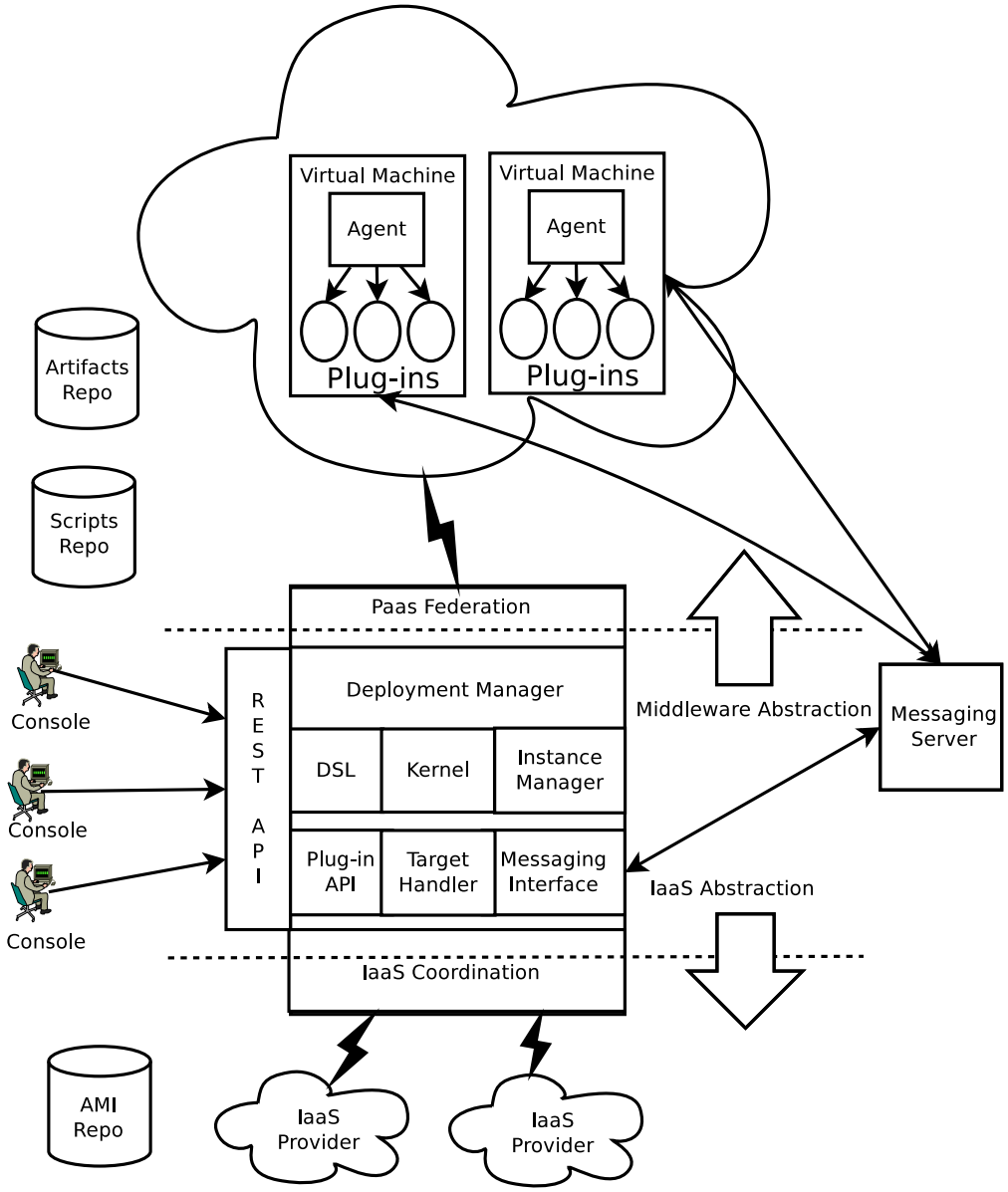

Figure 6.1: Simplified architecture of Roboconf

machine on Amazon Web Services or creating a Docker container is not exactly the same thing (not the same means, not the same API, not the same libraries). To specify the configuration and the library to perform the configuration, a scoped instance must be associated with a target. A target includes the identifier of a "target handler" library and some properties to indicate how to configure the machine. A

target is defined by:

- A **handler**, which points to a target handler (an OSGi bundle to deploy in Roboconf).
- A **name** (optional), that will be more explicit for users.
- A **description** (optional), that will help users to understand it.
- Various properties, that depend on the handler.

Beyond its properties, a target aims at being associated with scoped instances. This association is made per application. Every time designing and creating a new Roboconf application, the associations between its scoped instances and targets should be configured. It is also possible to set a given target as the default for a given application. When a scoped instance is not associated with any targets, and that we try to deploy something on it, then Roboconf will use this default target to deploy the instance. Multi-IaaS (for hybrid cloud) is handled by defining several scoped instances and by associating them with different targets. Thus, one can target an Openstack infrastructure, while another one can be deployed on Amazon Web Services.

Deployment targets can be defined and associated with applications by either at runtime using the web console or at design time using predefined targets. In both ways, it needs to make sure that corresponding target handlers of OSGI bundles had been already installed and started before. With the web console, this includes several steps:

- Creating and editing the properties of a target.
- Deleting a target, provided it is neither currently used nor referenced by an application.
- Associating a target with a specific instance of a given application.
- Defining a target as the default one for an application.

With the latter way, it is possible to embed predefined targets in deployment archives. Notice that these targets and their associations can be modifiable with the web console latter. For every predefined target, the definition follows the same schema. In the graph model, root components are associated with hardware elements. This can be virtual machines, existing machines or devices. These root components must be associated with the target installer. When Roboconf parses the model, it will recognize this installer and then search for the IaaS configuration. It means the resource directory associated with this root component may

contain a target.properties file. All the predefined instances of this same component will use this target. Subsequently created instances will use the default one associated with the application.

The DM will analyze the properties and deduce corresponding target handlers to pick up. It will then take the right client library to create the VM. Most of the target implementations of Roboconf rely on virtual images (or AMI or appliance). It means Roboconf creates a VM from a template and properties given in the IaaS properties. A definition (in target.properties file) of a target abstracting properties of Amazon EC2 VM is demonstrated in Figure 6.2a. A complete description of supported properties for Amazon EC2 VM and definitions of other targets officially advocated by Roboconf (OpenStack, EC2, MS Azure, VMWare, Docker, Apache JCloud, Embedded, In-memory) can be referenced in Appendix.

It is also possible to define an elastic IP address. It is not a good idea to set it in the target.properties, although it works. Indeed, all the VM instances created from this target configuration will try to use the same elastic IP. Since Amazon does not allow it, only the last created VM will be associated with this IP. In a general matter, VM instances can inject target parameters through their definitions. These properties must start with "data." followed by the target property. Therefore it is much better to define the elastic IP in the instance definition, as shown in Figure 6.2b.

6.2.3 Roboconf Plug-ins

Roboconf agents use plug-ins to manage the life cycle of instances. As a reminder, Roboconf agents are deployed either on VMs, or on devices. In any case, they are supposed to be deployed remotely with respect to the DM. An instance represents a concrete software component, that will interact with instances of other components deployed with Roboconf. The life cycle of an instance is composed of the following phases:

- **initialize**: check the prerequisite are available on the machines of agents.
- **deploy**: receive from the DM the files to deploy and deploy them effectively.
- **start**: start the deployed instance. At this moment, imports from other components are resolved. Other components are notified that a new instance is running and receive new exports.
- **update**: components this instance depends on have changed. Imports are updated and it may require an update of the configuration files.
- **stop**: stop the deployed instance. Other components are notified an instance is stopped and update their imports.

```
# Configuration file for EC2
handler = iaas-ec2
name =
description =

# EC2 URL
ec2.endpoint =

# Credentials to connect
ec2.access.key =
ec2.secret.key =

# VM configuration
ec2.ami =
ec2.instance.type = t1.micro
ec2.ssh.key =
ec2.security.group  =
```

(a)

```
instance of VM {
    name: VM1;
    data.ec2.elastic.ip:
    your-elastic-ip;

    # Put children
    # instances next...
}
```

(b)

Figure 6.2: a) Roboconf target definition of Amazon EC2 VM; b) example of configuring an elastic IP for a EC2 VM instance

- **undeploy**: delete the deployment files from the machine. Even be undeployed, the instance is still in the model. When an instance is registered in the model but that it is not running or even deployed, it is marked as not deployed. Therefore, it can have predefined instances in your application even if they are not running or deployed.

A Roboconf plug-in handles the complete life cycle of an instance. This means a plug-in implements all the life cycle steps. From a concrete point of view, configuring Roboconf plug-ins is what will take most of the time. Writing the Roboconf model (application description, graph definition and initial instances) is fairly easy and is achieved quickly. Every plug-in required resources located into the directory associated with a graph component. As an example, let's assume we may have a following component.

```
Apache_Load_Balancer {
    installer: puppet;
}
```

This implies there will be a directory called Apache_Load_Balancer which will contain at least one Puppet module. The fact it is a Puppet module is due to the

Puppet installer. If we had used the Bash plug-in, the directory would contain bash scripts. What files are expected depend on the plug-in itself. Basically, plug-ins are the way Roboconf can be extended. Thus, it is possible to write its own plug-ins to integrate and share a given behavior.

6.2.4 Extension of the Roboconf Platform

The cloud platform provides some extendable dimensions:

- **Roboconf plug-in**: A Roboconf plug-in is an extension for a Roboconf agent. There are some built-in plug-ins for Roboconf agents. The Script plug-in handles various scripts language including Bash, Shell, PERL, Python, etc. The Puppet plug-in interacts with Puppet agents. The File plug-in handles simple file manipulations. The Logger plug-in is a basic extension that only logs actions. The Roboconf plug-in is an OSGi bundle with specific meta-data. Roboconf uses iPojo to simplify OSGi development. A new plug-in needs to implement a PluginInterface interface (see Appendix).

- **Roboconf deployment target**: A deployment target designates a platform or a solution on which Roboconf can deploy software. Currently available targets include cloud infrastructures (AWS, Openstack...), Docker, etc. A deployment target is supported through an OSGi bundle with specific meta-data. Developers have to implement a TargetHandler interface for each new deployment target (see Appendix).

- **Roboconf monitoring handler**: A monitoring handler is an extension of the monitoring module at the side of Roboconf agent. These handlers are in charge of polling or verifying resources on the machines of agents. They are used for autonomic management. Currently available handlers include File, Nagios and Rest. A monitoring handler is an OSGi bundle with specific meta-data. New monitoring handlers can be added to Roboconf by implementing a MonitoringHandler interface class (see Appendix).

- **Roboconf elasticity rule**: To provide highly extensibility, Roboconf allow users to define elasticity rules specifying to each cloud application. Furthermore, Roboconf allows redundantly elasticity rules at multiple levels of resource granularity to ensure that failure of one of these mechanisms at a level has minimum impact to overall efficiency. A new elasticity rule can be added to Roboconf using the DSL dedicated to the rules of elasticity as described in Section 5.5.3.

6.3 Deployment Process

6.3.1 Instance Life Cycle

Roboconf applications do not have a life cycle. What does the life cycle of a distributed application mean? Assuming we start the application, it means all its parts are started. But some parts may be optional, the application may work without them. Besides, what is the state of the application when one of its vital part falls? For these reasons, it was decided to not associate a life cycle with a Roboconf application. Instead, such an application supports the following operations:

- *Addition*: load and add a new application to the list of managed applications.
- *Shutdown*: a commodity operation to undeploy all the parts of the application.
- *Removal*: remove the application from the managed applications. This is only possible when all the application parts have been undeployed.

Eventually, there are several operations to perform on the applications parts. In Roboconf, applications parts are called instances. An instance is associated with a component, itself defined in the graph model. An instance is a specific piece of software, running and working within the scope of a Roboconf application. It can be an application server, a database, an applicative module or even a VM. Instances do have a life cycle as shown in Figure 6.3.

Some of the steps are said unstable (or transitive): *deploying, starting, stopping* and *undeploying* will end up with a stable state (either *not deployed, deployed - started* and *deployed - stopped*). The transitive states are used to report information to the user. Indeed, *deploying* (or *starting*, or *stopping*, or *undeploying*) an application can be a long-running operation. The **unresolved** state is reserved to non-root instances. Before a *deployed* instance can be started, its dependencies are verified. If they are all satisfied (Roboconf knows where they are and that they were started), then the instance can be started. Otherwise, it will remain in the unresolved state until its dependencies are resolved. Once they are all satisfied, Roboconf will automatically start the instance. Thus when an instance is in this state, it will start as soon as all its dependencies are resolved.

The *problem* state is a little bit specific. It is reserved for root instances (often VM). If a started root instance has not sent a heartbeat for some time, the root instance will go into the problem state. If a heartbeat arrives, it will go back into the deployed - started state. If a root instance is in the problem state, it means either that the agent encountered a problem, that the VM has network issues, or that the messaging server had a problem. It does not mean application parts do not work. Another difference of root instances is that they do not reach the deployed -

stopped state. They are either deployed and started, or not deployed. They cannot go through the intermediate states.

Let's now illustrate the life cycle of an instance with an example. We will take the LAMP example (Apache load balancer, Tomcat and MySQL). We will focus on the Tomcat server.
1. We have created and started a VM for the Tomcat server.
2. We now create an instance in our model to declare a Tomcat server. State is **not deployed** which allows to declare instances without deploying them.
3. We deploy it. Its state first jumps to **deploying**. Once it is deployed, the state switches to **deployed - stopped**.
4. We start it. It goes to the **unresolved** state.
5. If all its imports are resolved (i.e., a MySQL database was deployed and started), then it can go to the **starting** state before ending (normally) in the **deployed - started** state. Otherwise, it will remain in the **unresolved** state until a MySQL database is started. Let's suppose a MySQL database was started. The state of the Tomcat instance is **deployed - started**.
6. Let's stop the MySQL database. Roboconf changes the Tomcat state to **unresolved** again. All the other instances that depend on this Tomcat instance will also update their life cycle if necessary (chain reaction). Restart the MySQL instance and the Tomcat will go back into **deployed - started** state.
7. We stop the Tomcat server. State goes through **stopping** before ending with **deployed - stopped**.
8. We undeploy the instance. State goes through **undeploying** before ending with **not deployed**.

The life cycle of application instances is managed through the DM. A user that wants to modify the state of an instance will have to use the REST API. This API can modify the state of an instance, or modify states in bulk mode.

6.3.2 Instance Synchronization

As discussed, before we can start an instance, its dependencies must be resolved. If all its imports are resolved, then it can go to the starting state before ending in the deployed - started state. Otherwise, it will remain in the unresolved state until all instances on which it depends started. Exchanging the export/import variables is done by the Agent, and only the message server is used in the process. The instances subscribe to and publish on topics in these message server. If henceforth the system runs into a stable state in a long term, the DM is indeed not necessary and can be suspended or shut down. The communication protocol for instance synchronization is demonstrated as follows.

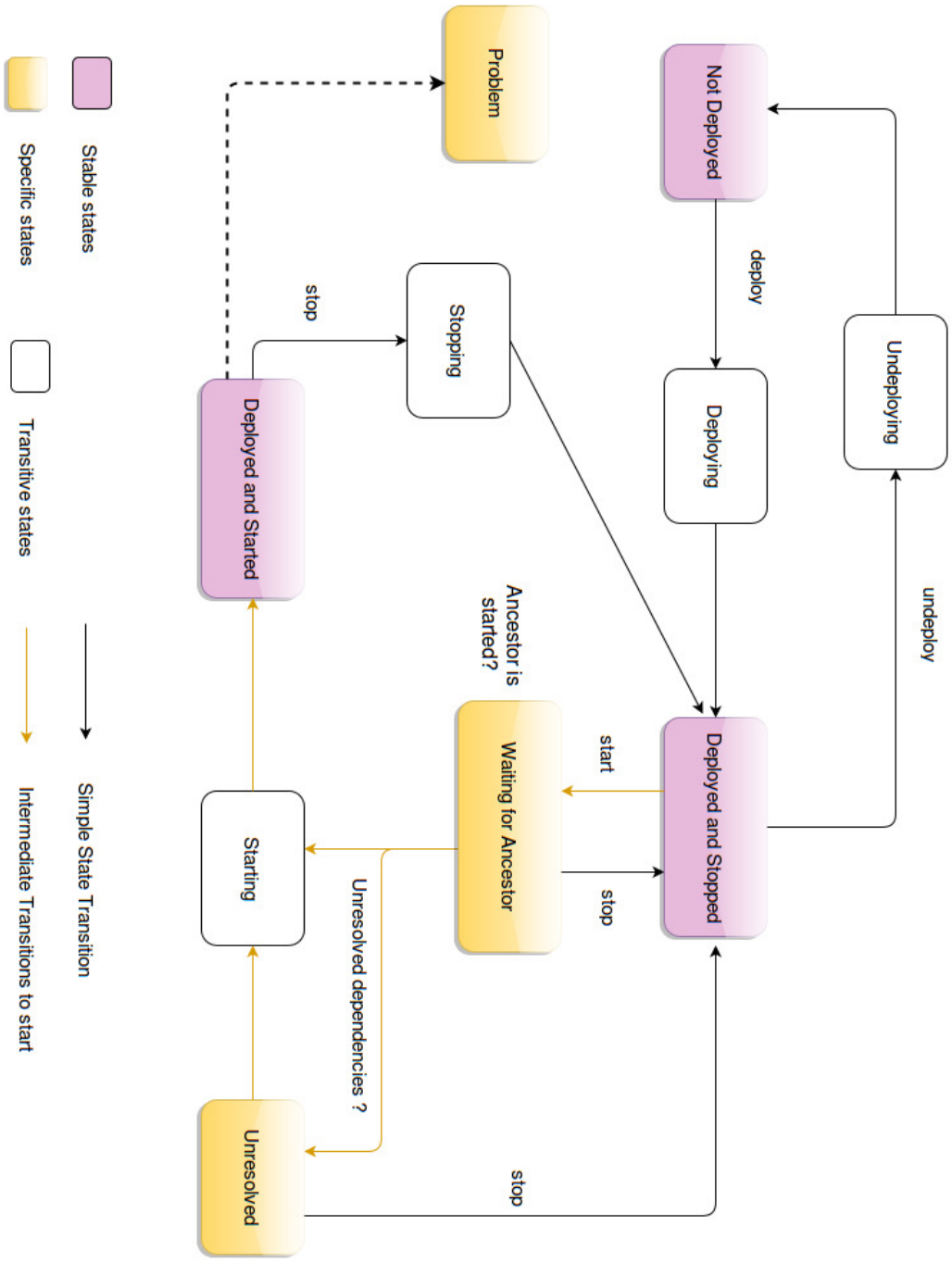

Figure 6.3: Life cycle of a Roboconf instance

```
foreach export:
  subscribe to "$applicationName.import.$exportName"
  publish exported vars on "$applicationName.export.
  $exportName"
foreach import:
  subscribe to "$applicationName.export.$importName"
  publish notification on "$applicationName.import.
  $importName"
(when listening to new messages,) if receive notification
-> publish exported vars on "$applicationName.export.
$exportName"
```

Using the asynchronous message server and thanks to the previous process, the system is able to configure in any order. Note that the "sub channel" field defined in the Instance and Type is used in the process below. For example instead of publishing on or subscribing to a channel named "$applicationName.export.$exportName", it publishes/subscribes to "$applicationName.export.$exportName.$subChannelName".

Below is an example demonstrating the synchronization protocol of a configuration exchange between an Apache and a Tomcat. In Figure 6.4a, the Apache begins the configuration process first, and Tomcat begins some time later. Instances are colored in orange and topics are in yellow.

In Figure 6.4b, the Tomcat starts the process first, and the Apache starts after. When an instance is removed or updated, a similar process as the one above is used to remove the instance and its configuration from the application or update other dependent instances, respectively.

6.3.3 Initial Deployment Process

We use the three-tier RUBiS example to understand the way Roboconf deployment works. As mentioned, dependencies between components is presented in Figure 5.2. In an IaaS elasticity scenario, multiple Tomcat nodes can be added/removed to adapt to traffic, but it requires a dynamic reconfiguration of the Apache node in order that "mod proxy" knows about all the available Tomcat nodes. Roboconf is told to deploy Apache, MySQL and Tomcat on three separate VMs that similar to Figure 5.3b. This includes updating the configuration files as soon as dependencies can be resolved (e.g. when it is aware of the MySQL IP/port, Roboconf will send them to the Tomcat node, so it can update its configuration and start). The application components (MySQL, Tomcat, Apache) are defined as Figure 5.3a. The VM is supposed to support the deployment of either Apache, Tomcat or MySQL components and each component is described in terms of imports/exports. With this description, Roboconf knows when a deployed compo-

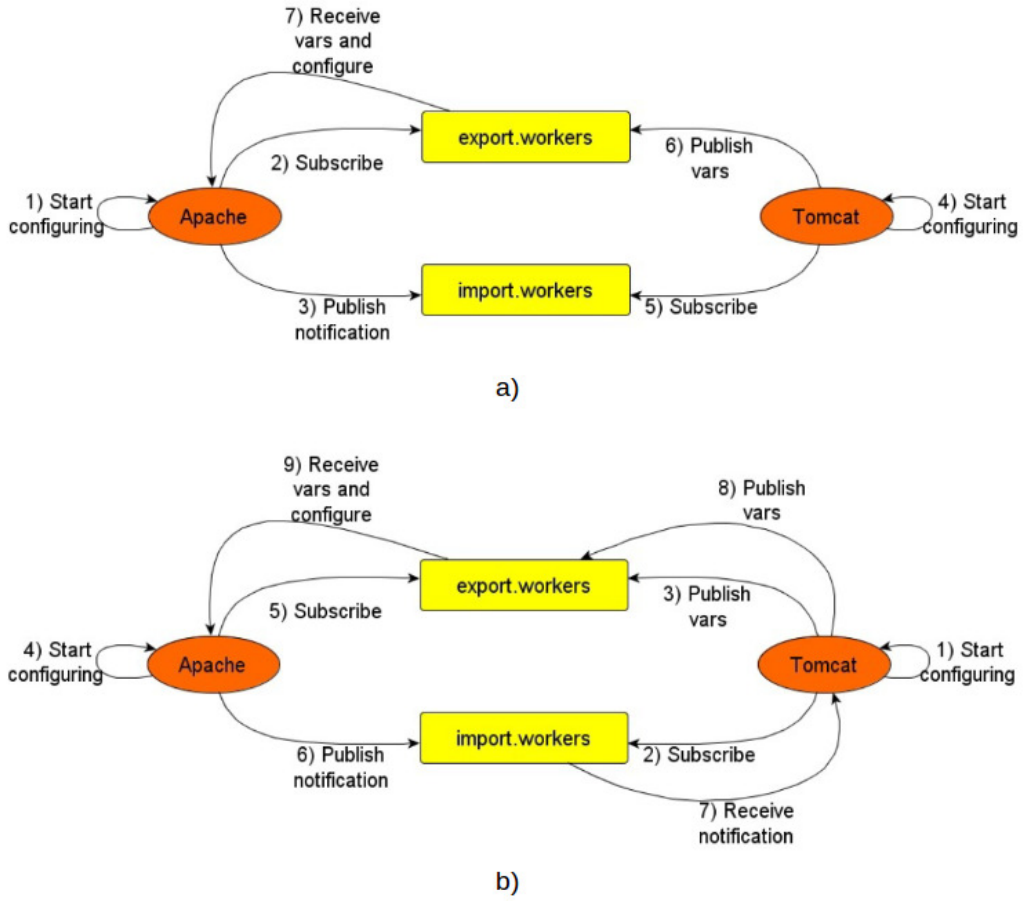

Figure 6.4: Communication protocol for instance synchronization

nent can be started. It is when all its imports are resolved! Roboconf is in fact responsible for import/export exchanges between components, and life cycle management (e.g. start the component when imports are resolved).

6.3.4 Reconfiguration Process

It often happens when everything is running, we need to create a new instance to adapt to changes from environment. It means the running system needs to indicate the component to instantiate, give it a name and define where it should go. In this particular example, due to an increasing workload a new Tomcat instance hosted by a VM instance has to be added automatically. We can either reuse an existing or create another VM instance. In this scenario, we take the latter. The Roboconf DSL provides set of autonomic rules to respond to the detected changes. The

```
[EVENT nagios peak-load]
GET services
WaitObject: $HOSTNAME CPU load
WaitCondition: CPU load > 80
WaitTrigger: check
```

```
[REACTION peak-load Replicate-Service]
/vmec2/tomcat1
```

Figure 6.5: Example of an autonomic rule of Roboconf DSL: (above) at the agent side, (bottom) at the DM side

agents measure anomalies frequently and send notifications to the DM. The DM responds to the notifications using corresponding imperative rules. Figure 6.5 depicts a rule that we apply for the example. At the agent side, we use LiveStatus which is the protocol used by Nagios and Shinken. The LiveStatus query retrieves measures of CPU load from a local Nagios or Shinken agent, if this parameter is over 80%, a notification will be sent to the DM. In turn, the DM applies the handler "Replicate-Service" to respond to the notification resulting in adding an entire new path "/vmec2/tomcat1". Both instances of this path, the "vmec2" and "tomcat1" will be added to the application model. It is one more example emphasizing hierarchy of the Roboconf DSL. At the very beginning of the adding process, both the two new are not started, and not even deployed. The DM is asked to deploy and start them. First, the DM provisions the VM. Once it is up, the DM sends the deployment command to the VM and a new Tomcat instance is deployed over it. The Roboconf agents then publishes the exports (i.e. a new Tomcat instance with a port and IP address). Since the Apache load balancer imports such components, it is notified a new Tomcat arrived. The agent associated with the Apache VM invokes a Roboconf plug-in to update the configuration files of the Apache server. Therefore, the load balancer is now aware of two Tomcat servers. If configured in round-robin, it will invoke alternatively every Tomcat server when it receives a request. It is worth noting that real magic with Roboconf is the asynchronous exchange of dependencies between software instances whereas the deployment and life cycle actions are delegated to plug-ins.

6.4 Elasticity Management as an Autonomic System

The context of the execution environment for applications deployed across multiple clouds can change quickly in minutes or even seconds. To manage a system in this changing so quickly, it is essential to react dynamically and automatically

to regulate events that occur or anticipate them. Autonomic systems and process control concepts can be used to implement a system that knows its own state and reacts to the change. The essential part of such a system is the controller that is external to the observed environment. The responsibilities of an elasticity controller are to monitor the system, analyze the metrics, plan actions and execute them. This is known as a control loop name MAPE-K (Monitoring, Analysis, Plan, Executing, Knowledge - see Section 2.2.2) introduced by IBM.

The Roboconf cloud platform manages elasticity to multiple levels in the same way. In fact, the management of elasticity in Roboconf does not focus on a specific layer of the cloud. Roboconf uses resources through the abstraction layers provided by the DM.Roboconf offers the capacity scaling for applications by allocating/deallocating resources as needed. For example, the Roboconf platform can add more resources if it detects a deterioration in the performance of the application. However, if resources are underutilized, shrinking might be necessary. This feature is managed as an autonomic control loop by Roboconf platform.

In practice, cloud resource supply is not instantaneous. Provisioning a new server may take a few minutes [140]. As described in Section 3.3, the vertical scaling often occurs more rapidly than the horizontal one. Moreover, using lightweight containers instead of coarse-grained VMs accelerates the provisioning time. The elasticity mechanism is based on the Roboconf DM that has an intelligence mechanism allowing it to allocate resources in a timely manner. As described above, the Roboconf elasticity mechanism follows the phases of the control loop.

We propose the autonomic management as an integrated feature of Roboconf, which consists of two parts. On one side, agents retrieve metrics on their local node. These metrics are compared against some values given in the configuration of agent. If they exceed, equal or are lower than given values, depending on the configuration rules, agents send a notification to the DM. On the other side, when the DM receives such a notification, it checks its configuration to determine which actions to undertake. These actions can range from a single log entry, to email notification or even replicating a service on another machine. Figure 6.6 sums up the way autonomic management works. While detection is delegated to the agents, reactions are managed by the DM.

The autonomic configuration is in fact defined in application projects. It means every project has its own rules and reactions. In this perspective, the project structure is enriched with a new directory, called autonomic. With the descriptor, graph and instances directories, it makes the fourth. The autonomic directory expects two kinds of files.

- **Measures** files include a set of measures to perform by the agent. Such a file is associated with a given component in the graph. Hence, we can

consider the autonomic rules as an annotation on a component in the graph. These files must be named *<component name>.measures*.

- **Rules** files define the actions to undertake by the DM when a measure has reached a given limit. Such a file is associated with the whole application. It must be named *rules.cfg*.

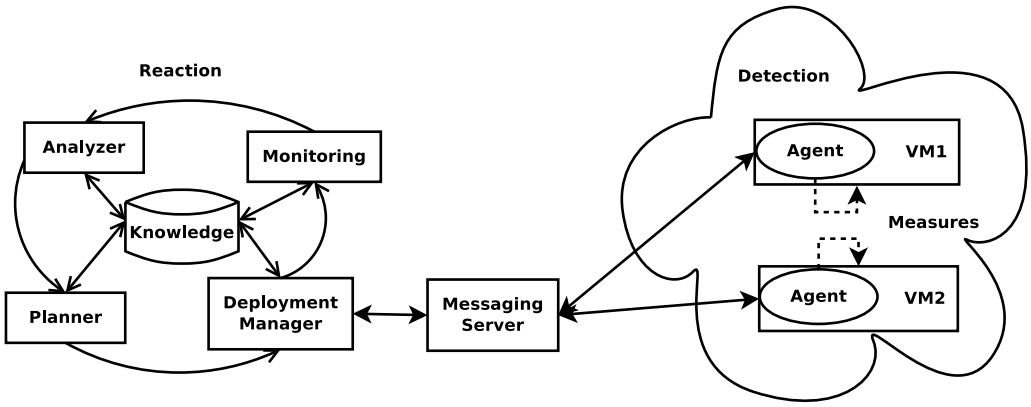

Figure 6.6: Roboconf autonomic loop for elasticity

6.4.1 Monitoring Phase

Deployed cloud applications are monitored and their configurations can be adjusted based on the metrics collected by the **Monitoring** sub-module that is integrated both on the agents (to monitor instances inside VMs) and the DM (to monitor the Roboconf platform itself). The monitoring component is both invasive and non-invasive applications. It is intrusive because it can instrument the application and execution environment at runtime. It is non-intrusive because it can monitor the phenomena outside the application such as network traffic, the percentage of CPU used, the amount of memory consumed.

As modules of Roboconf has been developed as OSGi bundles, the monitoring of the DM and the agent distributions can be relied on Apache Karaf. The OSGi bundles do not embed anything related to monitoring. But Karaf exposes a JMX interface which can be used to retrieve information and manage the OSGi servers. JMX access can be completed with the web consoles and shell access. Monitoring applications would be done by third-party distributed system monitors such as Ganglia or Nagios. These tools can be integrated into Roboconf by implementing the MonitoringHandler interface (see Section 6.2.4). Currently Roboconf supports the File, Nagios and Rest as monitoring mechanisms at the agent side.

6.4.2 Analyzing Phase

The monitored data is analyzed by the sub-module **Analyzer**. The users, depending on their intentions, can apply analytic algorithms or statistical models along to historical data retrieved from **Knowledge** databases (Cassandra in this work) to elaborate the data in this phase. The analyzed data can be used to update backward the Knowledge.

6.4.3 Planning Phase

Given the situation, the sub-module **Planner** will conduct the planning phase in order to create an action plan to bring the metric values to normal. This plan may be based on a set of rules governing the operations the DM (reactive) or a sophisticated model that works on the restoration of the system behavior (proactive).

6.4.4 Executing Phase

In the executing phase, the DM delegates to the agents with corresponding plugins to perform the actions decided in the planning phase. Once implemented, these actions will cause a change in the behavior of the system that will be notified to the DM in the control loop.

We will introduce an algorithm to orchestrate elasticity in planning phase, which takes advantage of the fine-grained multi-level provisioning and scaling supported by Roboconf in Chapter 7.

6.5 Synthesis

In this chapter, we described the architecture and the implementation of our Roboconf platform to take into account the architectural requirements of distributed applications in a multi-cloud environment. We offer the platform as a multi-cloud service to deploy, run and manage distributed applications. We not only described each component of the Roboconf architecture but also discussed the implementation choices and the technologies used to implement them. Roboconf takes as input the description of a whole application in terms of “components” and “instances”. Components can be seen as object definitions, while instances are obviously instances of these objects. From this model, it then takes the burden of launching VMs, deploying software on them, resolving dependencies dynamically among software components, updating their configuration and starting the whole stuff when ready.

Roboconf handles instance life cycle: hot reconfiguration (e.g. for elasticity

issues) and consistency (e.g. maintaining a consistent state when a component starts or stops, even accidentally). This relies on a messaging queue (currently RabbitMQ). Application parts know what they expose to and what they depend on from other parts. The global idea is to apply to applications the concepts used in component technologies like OSGi. Roboconf achieves this in a non-intrusive way, so that it can work with legacy software. Application parts use the message queue to communicate and take the appropriate actions depending on what is deployed or started. These appropriate actions are executed by common plug-ins such as bash, puppet or customized ones such as java-servlet, osgi-bundle. Roboconf is a distributed technology, based on AMQP and REST/JSON. It is both IaaS and PaaS-agnostic. Many well-known IaaS are supported including OpenStack, Amazon Web Services, Microsoft Windows Azure, VMware vSphere, a plug-in to deploy Docker container as well as a "local" deployment plug-in for on-premise hosts. In the PaaS aspect, not only potential type of applications are tensely brought up to the Cloud such as OSGi or IoT but also state-of-the-art PaaS are purposefully included such as Heroku Platform, Google App Engine and CloudBees. Roboconf satisfies most of state-of-the-art requirements of a modern multi-cloud platform such as component fine-grained hierarchical description, dynamic dependency resolution, concurrent component deployment, multi-cloud distributed deployment, genericity, extensibility, scalability and reusable/configurable deployment plans.

Chapter 7

EVALUATION OF THE MULTI-LEVEL FINE-GRAINED ELASTICITY WITH ROBOCONF

Contents

In this chapter, we assess the multi-level fine-grained elasticity approach with Roboconf platform. We use Roboconf hierarchical DSL to naturally describe structure of the motivating application as described in Section 5.2. We propose novel elasticity algorithms for two experiments showing efficiency when implementing multi-level fine-grained elasticity. These algorithms can be implemented by the autonomic modules of Roboconf and interpreted by Roboconf DSL dedicated to elasticity rules as mentioned in Section 5.5.3.

7.1 Multi-level Elasticity

Scaling mechanisms for the motivating application are partly mentioned in the Section 5.5.3 under Roboconf elasticity language. In this section, we conduct an experiment applying a multi-level scaling algorithm used for conducting elasticity on a variant of this application.

7.1.1 Experiment Setup

This experiment related to the elasticity context applying to the J2EE application. With application tier, we use in initials two Tomcat servers dedicating to serve two different webapps: **WebappA** and **WebappB**. The "mod_proxy" is used to build a cluster of Apache servers in order to avoid yet another bottleneck. Each of Apache server implements the "mod_jk" serving as a load balancer in front of the Tomcat servers. This experiment focuses on elasticity of application tier, thus without loss of generality, the database is shared among webapps and hosted on a single MySQL server. All the VMs used in this system have been implemented on our private OpenStack cloud, which are configured intentionally to have similar configuration to Microsoft Azure Standard A2 instances with 2 cores and 3.5 GB memory. Each Tomcat created in the elastic reactions is a Amazon EC2 m3.medium with 1 core and 3.75 GB memory. The managed system is called System Under Test (SUT) that we use CLIF server [38], a distributed load injector, to create load profile and generate workload for the SUT in order to observe how the system reacts to changes of average response time (ART). These reactions are empowered by autonomic rules aforementioned in Section 5.5.3. The topology of this scenario is depicted in Figure 7.1 with Roboconf as the autonomic system.

7.1.2 Test Scenario

The loads are injected into an entrance of the Apache cluster which is a virtual IP. Then this cluster distributes the loads to the corresponding webapps through the Tomcat servers. On the one hand, the WebappB often gets low load, thus has a load profile as in Figure 7.2 with 50 virtual users who try to send HTTP GET requests to the WebappB and then stop to "think" a couple of time randomly. Behaviors of the virtual users are captured from real-world operations using a capturing tool of the CLIF server. The owner of the WebappB need not any elastic mechanism provided by Roboconf. On the other hand, the WebappA usually receives high load and thus has a load profile as shown in Figure 7.2, which is also designed by the CLIF server. The WebappA usually takes the burden of about 450 virtual users who have similar behaviors as in the case of WebappB. The owner of the WebappA requires Roboconf to ensure an acceptable performance for his

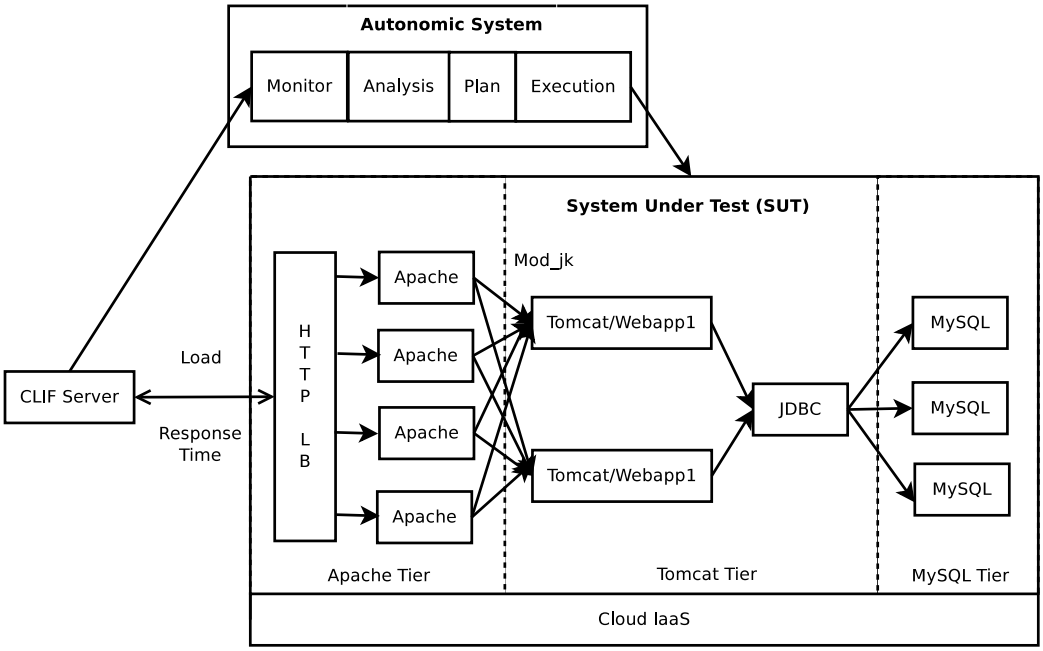

Figure 7.1: Topology of the J2EE test using CLIF load injector

webapp. Therefore, he demands an elastic load balancing solution to guarantee an ART as low as possible as stated in the SLA. To isolate workload among the webapps, we install Tomcat and the webapps into Docker containers. A container template hosting the *Tomcat-WebappA* is saved as Docker image for reusing in scaling decisions. When the ART varies, this solution includes provisioning a whole new */VM/Docker/Tomcat-WebappA* instance (See rule [I] in Section 5.5.3) or replicating only the *.../Docker/Tomcat/WebappA* container instance (rule [III]) while scaling out as well as removing the instances (rule [IV]) or migrating the webapps (rule [V]) while scaling in with minimum side effects to overall system.

The polling periods is set to 10 seconds that means the ART of all requests from all users are collected each 10 seconds. This gathered data are sent to the Roboconf monitoring and analyzer modules to be aggregated and further analyzed. The analyzed information then are delivered to the planner module to generate new configuration for the system based on ECA rules. The ECA rules decide whether the system should create an entire application server (a VM) or only replicate a container of webapp instance. To simplify the experiment, we only consider the horizontal scaling at VM and container levels.

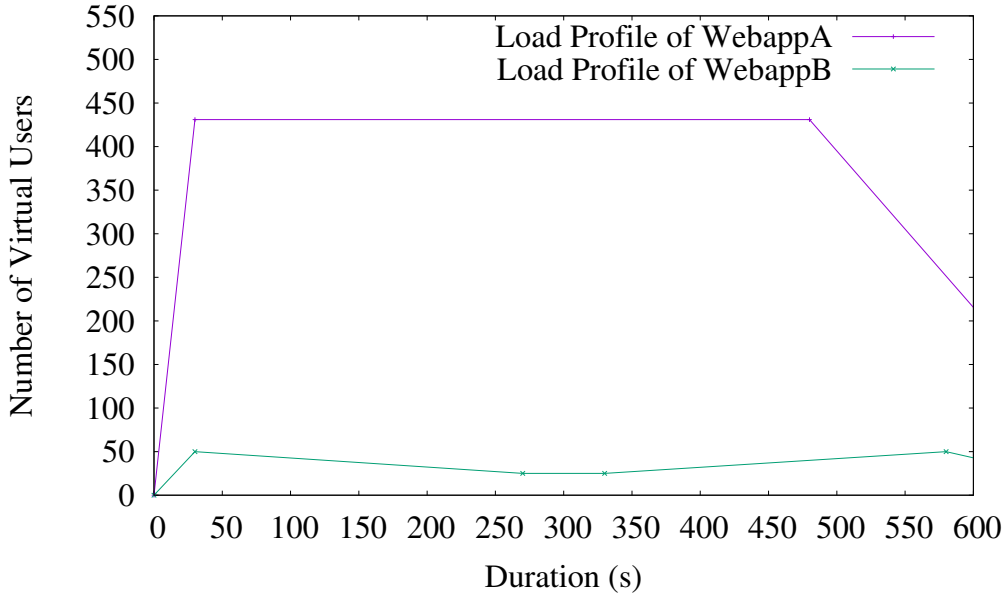

Figure 7.2: CLIF load profiles of the WebappA and WebappB

7.1.3 Scaling Algorithm

The multi-level scaling algorithms used for this experiment is shown from Algorithm 1 to Algorithm 5. To avoid oscillation, these algorithms apply restrictions to prevent multiple creating of new VMs or new instances in a short period of time (Restriction 1). At least the system needs to wait until it gets knowledge about the new one before another can be created automatically. It is called synchronization time which includes the VM provisioning (from a VM image including the required Docker, Tomcat, Webapp) and reconfiguration time for the existing Apache (to know the attendance of the new Tomcat) in the case of provisioning an entirely new VM. With replicating or migrating a Docker container hosting a Webapp instance, the synchronization time only contains the latter one. Another considered rule is to prohibit the migration/replication of an instance to spots where also are on-peak time (Restriction 2). Table 7.1 lists the main symbols used throughout the algorithms.

Table 7.1: Symbols Used in Scaling Algorithms

Parameter	Description
t^a	The observed response time of the application **a**
ζ^a_{VM}	Set of VMs which host the application **a**
ζ^a_C	Set of Containers which contain the application **a**
T^u_a, T^l_a	Upper bound T^u_a and lower bound T^l_a of the required response time of application **a**
$u(VM^a_i, r)$	Utilization of resource **r** in the VM **i** hosting the application **a**
$u(C^a_i, r)$	Utilization of resource **r** in the Container **i** containing the application **a**
$u^u(r)$	The threshold of utilisation for scaling up resource **r**.
$u^l(r)$	The threshold of utilisation for scaling down resource **r**.
$c(C^a_i), c(VM^a_i)$	Running cost of container C^a_i or VM^a_i
$c(C^a_i, r), c(VM^a_i, r)$	Running cost of one unit of resource **r** of container C^a_i or VM^a_i

Algorithm 1: MS (Multi-level Scaling)

```
1:  while (the application is still running)
2:      Monitor t^a & t^b in current time frame
3:      if (t^a > T^u_a) then
4:          if (t^b < T^u_b) then MSOC()
5:          else MSOVM()
6:      else if (t^a < T^l_a) then
7:          if (t^b > T^l_b) then MSIC()
8:          else MSIVM()
9:      end if
10: end while
```

In the Algorithm 1, Roboconf captures the observed response time t^a and t^b of the application **a** and **b** in each time frame (line 2). Then it based on the corresponding value of t^a and t^b to give the appropriate scaling decisions. The algorithm triggers a multi-level scaling out at container level (MSOC - see Algorithm 2) whenever the t^a is larger than the upper bound of its required response time T^u_a and, to satisfy the Restriction 2, the t^b is less than its upper bound of the required response time T^u_b (lines 3 and 4). In the case t^b larger than T^u_b, a multi-level scaling out at VM level (MSOVM - see Algorithm 3) is triggered (line 5). In the scaling in case, MSIC (Algorithm 4) and MSIVM (Algorithm 5) are used with the corresponding values of t^a and t^b (lines 6 to 8).

Algorithm 2: MSOC (Multi-level Scaling out at container level)

```
1:  Measure t^a, u(ζ^a_VM, r), u(ζ^b_VM, r)
2:  ζ_VMC = ζ^a_VM
3:  while (t^a > T^u_a && ζ_VMC ≠ φ )
4:      for (i=0 ; i < |ζ_VMC|; i++)
5:          calculate EUR(VM^a_i)
6:      end for
7:      Select VM^a_i with the smallest EUR(VM^a_i)
8:      if (VM^a_i.hasEnoughResources( )) then
9:          ζ^a_C.update{C^a_j}
10:         if (!ζ^a_VM.exist(VM^a_i)) then
11:             ζ^a_VM.update(VM^a_i)
12:         end if
13:         ζ_VMC = { }
14:     end if
15:     if (ζ_VMC ≠ φ) then ζ_VMC.remove(VM^a_i)
16:     end if
17:     Measure t^a, u(ζ^a_VM, r), u(ζ^b_VM, r)
18: end while
```

With MSOC, the criterion of efficiency of resource utilization (EUR) needs to be calculated to select the VM candidates able to host the new container of an application (lines 4 to 6). The EUR is the product of a weighted positive constant α, a product of resource utilization ratio of the VM $u(VM_i^a, r)$ of each resource r and a resource running cost of the VM c(VM_i^a) (Expression 1). To become the selected one, the VM should have the smallest EUR and has enough required resources for a container.

Expression 1:

$$\textbf{EUR}(VM_i^a) = \alpha . (\prod_{r=vCPU,memory,diskStorage} \mathrm{u}(VM_i^a, r)) . \mathrm{c}(VM_i^a)$$

Algorithm 3: MSOVM (Multi-level Scaling out at VM level)

```
1:  if (|ζ^a_VM ∩ ζ^b_VM| < VM_max) then
2:      ζ^a_VM.add(VM^a_i)
3:      ζ^a_C.update(C^a_j)
4:  end if
```

Algorithm 4: MSIC (Multi-level Scaling in at container level)

```
1:  if (ζ^a_C ≠ φ) then
2:      ζ^a_C.remove(C^a_j)
3:      for (i=0;i < |ζ^a_VM ∩ ζ^b_VM|; i++)
4:          check each VM in ζ^a_VM ∩ ζ^b_VM if it
            does not contain any containers
5:          if a VM is empty: ζ^a_VM.remove(VM^a_i)
6:      end for
7:  end if
```

Algorithm 5: MSIVM (Multi-level Scaling in at VM level)

```
1:  if (ζ^a_VM ≠ φ && ζ^b_VM ≠ φ) then
2:      ζ^a_VM.remove(VM^a_i)
3:  end if
```

There are some notices in the other algorithms. With MSOVM, the maximum number of VMs (VM_{max}) that allowed to create in the entire system needs to be taken into consideration (line 1). With MSIC, after removing a container, if it is the last container of a VM, the VM itself should be removed as well (lines 3 to 6). With MSIVM, a VM can only be removed when it is not the last one hosting either application **a** or **b** (line 1).

The very first 10-minute snapshot of the experiment with these algorithms is shown in Figure 7.3 and results are discussed deeply in the next section.

7.1.4 Result

Figure 7.3 shows the ART of both webapps and the corresponding reflections from the Roboconf to fluctuations of the response time. In addition, the figure also reports the changes in number of Tomcat servers while running the test case. The max response time of WebappA is set to 800ms, it means if the ART goes over this limitation, creating new Tomcat server or replicating the Webapp request should be made. In contrast, if the ART goes under min response time (200ms), a removing or migrating decision should be triggered.

We see that the ART of WebappA peaked at the 40^{th} second because of aggressive accesses of the 450 virtual users simultaneously. At the point "A1", a command to create a new Tomcat server was triggered instead of a replication due to a peak (400ms) happening in WebappB. The max and min response time of WebappB, which are not shown in Figure 7.3, were set to 400ms and 100ms, respectively. After this request, the framework silently observed the SUT without any further requests until it gets knowledge about the new server. This synchronization time finished at the 180^{th} second (the point "A2"), thus the WebappA

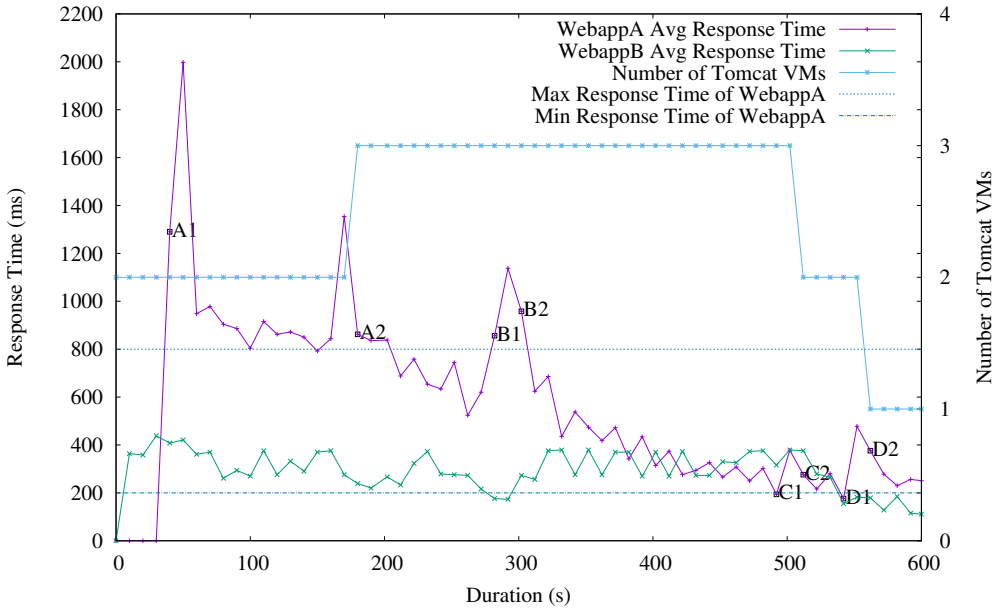

Figure 7.3: Autonomic responses with fluctuation of average response time of webapps

users continued experiencing slow accesses during 2 minutes 20 seconds more. At point "B1", once again the ART of WebappA was larger than the max limitation whereas the ART of WebappB was getting low. It is suitable to make a replication of WebappA (*/VM1/Docker1/Tomcat1/WebappA*) to under Tomcat2 and name it WebappA_2 (*/VM2/Docker2/Tomcat2/WebappA_2*). The synchronization time for creating the WebappA_2 was about 20 seconds which offered about 2 minutes better than the case of creating a new Tomcat server. Moreover, we avoided creating a totally new VM resulting in saving resources and money. In reverse, the system performed two times of the scaling in: a request to remove a Tomcat3 server at the point "C1" (which had been created at the point "A2") and a request to remove the Tomcat1 server at the point "D1" (which had been there from the beginning). The synchronization in both cases were almost the same (more or less than 10 seconds, finish at the points "C2" and "D2") because we do not care about the shutting down time of a VM. In spite of that, the result of this elasticity is the save of two VMs (from 3 VMs at the point "C1" to 1 VM at the point "D2") while the system were in low-load period.

The violation rate of the observed response time in the first 10 minutes using

the MS algorithm can be calculated as follows.

$$V_{MS} = \sum \frac{synchronization\,time}{600}.100\% = \frac{140 + 20 + 10 + 10}{600}.100\% = 30\%$$

We also conduct the same experiment on Roboconf with the algorithms implemented only at VM-level (without the Algorithms 2 and 4). The main difference is the synchronization time between the point "B1" and "B2" is longer (around 2 minutes 20 seconds). Likewise, the violation rate of the observed response time in the first 10 minutes without using the MS algorithm can be calculated as follows.

$$V_{woMS} = \sum \frac{synchronization\,time}{600}.100\% = \frac{140 + 140 + 10 + 10}{600}.100\% = 50\%$$

As we see when the workload changes extremely, the violation rate reduces significantly from 50% in the case of not using the MS algorithm to 30% in the case of using the MS one. Moreover, we can conclude that using the container-level scaling reduces the provisioning overhead in comparison to VM-level scaling (20 seconds in average in comparison to 140 seconds in average, respectively). Since the changes in load of a website usually happen, applying the multi-level scaling for elasticity brings forward cost saving and significant performance improvement as well.

7.2 Multi-level Fine-grained Elasticity

In this section, we conduct an experiment with another variant of the application in Section 7.1. We combine fine-grained vertical scaling algorithms with the ones introduced in Section 7.1.3.

7.2.1 Experiment Setup

This experiment is performed with the same J2EE application. The system setup for the experiment with the CLIF server is similar to the one depicted in the Figure 7.1, except the configuration of VMs and Docker containers. All the VMs used are configured on our OpenStack cloud with 2 cores and 9 GB memory. As we will only consider the utilization of memory in the Docker containers, each Docker container for WebappA and WebappB is allowed to use 4GB and 3GB of its hosting VM, respectively. While workload for WebappB is neglect, workload for WebappA is significant and shown in Figure 7.4.

7.2.2 Test Scenario

At the first minute, there are two VM1 and VM2 in the SUT. The VM1 hosts a container of WebappA and a container of WebappB. This VM allocated 7GB (4GB of WebappA and 3GB of WebappB) for both containers. The second VM hosts only a container of WebappB which consumed 3GB of memory. The initial state (I) of the experiment is demonstrated in Figure 7.5. The memory of Docker container can be adjusted on the fly by modifying cgroup files of each container [143]. The SUT keeps injecting workload from 150 virtual users until the 60^{th} second, then CLIF server doubles workload each 3 minutes. To simplify the experiment, we only consider the vertical scaling at container levels and horizontal scaling at both VM and container levels. The vertical scaling at PM and VM levels can be consulted in the work of Dawoud et al. [112, 113].

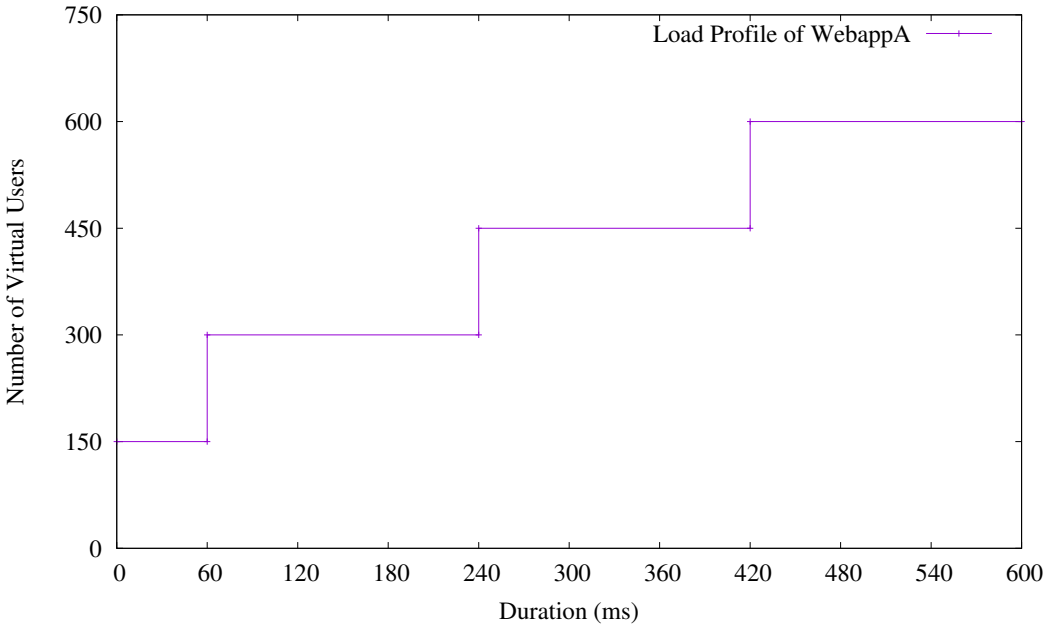

Figure 7.4: CLIF load profiles of the WebappA

7.2.3 Scaling Algorithm

In this section, we introduce multi-level fine-grained scaling algorithms which can be combined with the ones in Section 7.1.3. In this experiment, we replace the Algorithm 1 by Algorithm 6 which only considers the scaling of WebappA.

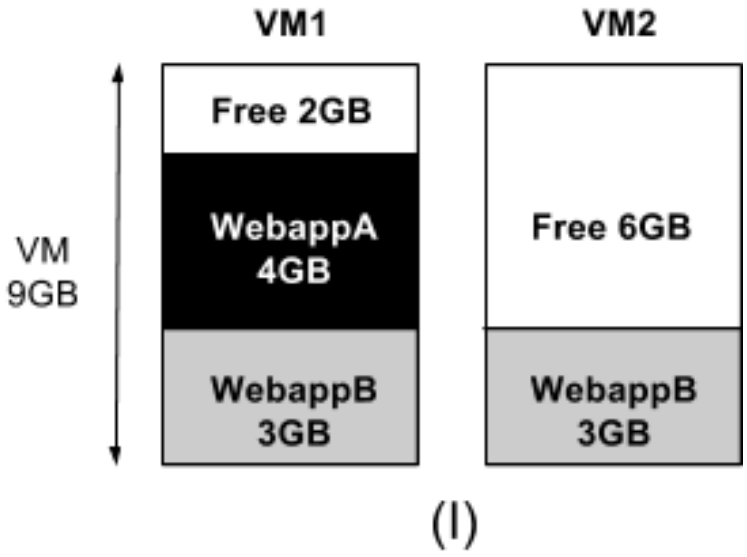

Figure 7.5: Initial state of the experiment with two VMs 9GB memory

Algorithm 6: MFS (Multi-level Fine-grained Scaling)

```
1:   while (the application is still running)
2:       Monitor t^a in current time frame
3:       if (t^a > T^u_a) then
4:           MFSUC()
5:       end if
6:       while (t^a > T^u_a)
7:           MSOC()
8:           MSOVM()
9:       end while
10:      if  (t^a < T^l_a) then
11:          MSIVM()
12:          MSIC()
13:          MFSDC()
14:      end if
15:  end while
```

In the Algorithm 6, Roboconf captures the observed response time t^a of the application **a** in each time frame (line 2). Then it based on the corresponding value of t^a to give the appropriate scaling decisions. The algorithm triggers a multi-level fine-grained scaling up at container level (MFSUC - see Algorithm 7) whenever the t^a is larger than the upper bound of its required response time T^u_a (lines 3 and

4). When the MFSUC fails its mission, then the scaling out at container and VM levels are performed consecutively (lines 6 to 9). With scaling down, the MFS algorithm aims to remove as many VMs, containers and container resources of the WebappA as possible, while still trying to held the response time between T_a^l and T_a^u and consider the present of WebappB. The algorithm first performs the VM-level scaling in to save the cost per unit of increased response time (line 11). The VM-level scaling in keeps running until the removing a VM would violate response time target. The MFS algorithm then conducts the container-level scaling in (line 12). The container-level scaling in also keeps running until the removing a container would again violate response time target. Finally, the resource-level scaling down of the WebappA containers is performed (line 13) and discussed later in this section.

Algorithm 7: MFSUC (MFSU at container level)

```
1:   Measure t^a, u(ζ^a_C, r)
2:   while (t^a > T^u_a && Lr ≠ φ)
3:       Lr = { }
4:       for (i=0;i<|ζ^a_C|;i++)
5:           if (u(C^a_i, r) > u^u(r) &&
                 VM(C^a_i).hasEnoughResources()) then
6:               Lr.add(C^a_i)
7:               calculate RRR(C^a_i, r)
8:           end if
9:       end for
10:      Select C^a_i with the smallest RRR(C^a_i, r)
11:      Add one unit of resource r to C^a_i
12:     Measure t^a, u(ζ^a_C, r)
13:  end while
```

With MFSUC, the scaling up is triggered when utilization of resource *r* in a container C_i^a is over the threshold of utilization for scaling up the resource *r*. Evidently, the VM containing the C_i^a must have enough the required resources (line 5). Moreover, the ratio of remaining resources *r* in the containers of WebappA (RRR) needs to be calculated to select the container candidates needed to be added an additional unit of the considering resource (2GB memory, in this experiment) (line 7). The RRR is the product of a weighted positive constant β, an exploitable rate of resource (1-u(C_i^a, r)) and a resource running cost of the container c(C_i^a) (Expression 2). To become the selected one, the container ought to have the smallest RRR and its hosting VM must have enough required resources for an additional unit.

Expression 2:

$$\mathbf{RRR}(C_i^a, r) = \beta.(1-u(C_i^a, r)).c(C_i^a)$$

With MFSDC, this algorithm selects a resource *r* with the largest RRR(C_i^a, r) (Expression 2) and removes one unit of *r* from the container C_i^a. This guarantees shrinking of the container with the largest free resources. The very first 10-minute snapshot of the experiment with these algorithms is shown in Figure 7.6 and results are discussed in the next section.

7.2.4 Result

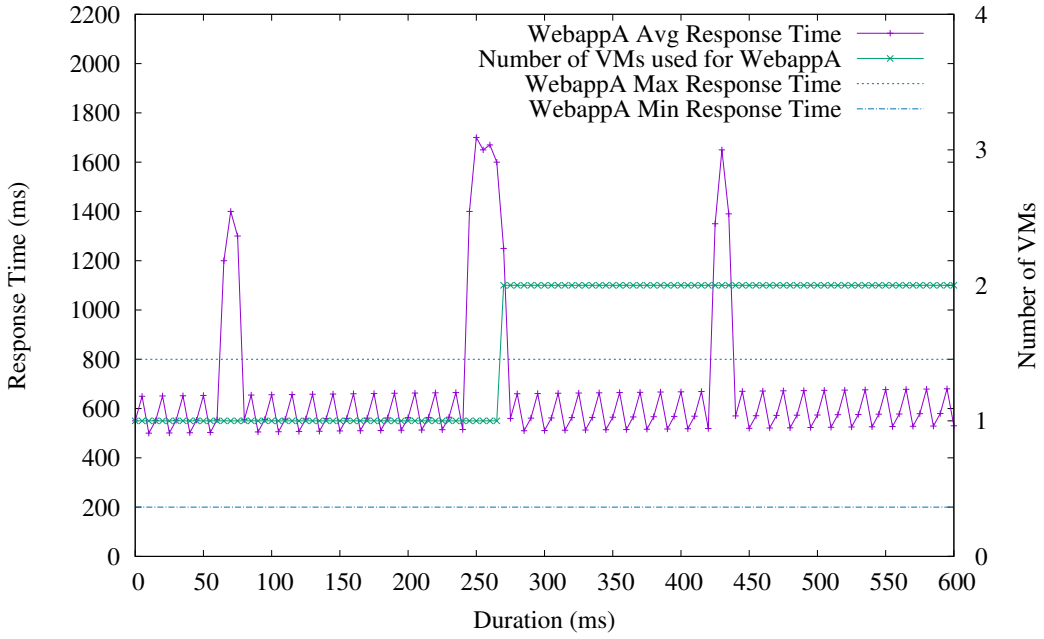

Figure 7.6: Autonomic responses with fluctuation of average response time of WebappA using MFS algorithm

Figure 7.6 shows the ART of the WebappA and the reflections from the Roboconf to fluctuations of the response time. We see clearly three peaks corresponding to three triggering of the MFS algorithm. Thereby, the SUT is undergone three state changes as shown in Figure 7.7.

The first peak at the end of the first minute triggers a resource scaling up at container level instead of a container scaling to bring the ART back to its desired range. The reason is the MFSUC algorithm knows that the VM1 still has enough memory resource (2GB) to allocate for the Docker container of the WebappA. Provided that a Docker container only can increase one unit of memory (2GB)

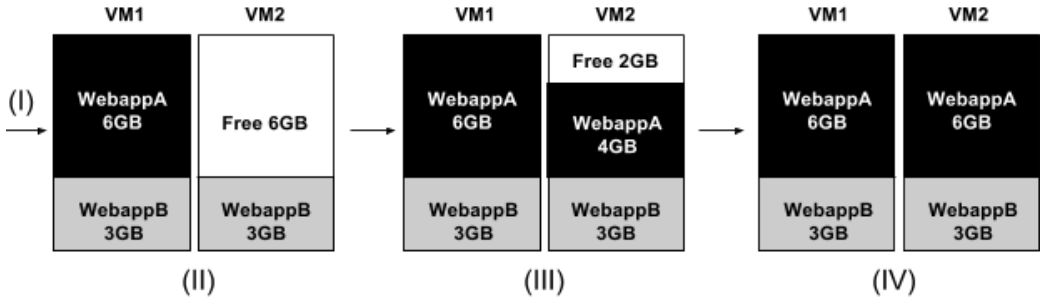

Figure 7.7: States of the experiment with MFS algorithm

each scaling-up time. The scaling synchronization time is about 5 seconds. This time spends mainly for the Docker daemon running on the VM1 recognizes the memory change. After the first scaling the SUT changes from the state (I) to the state (II) (see Figure 7.7).

Three minute later, the second peak is razed by a container-level scaling out. It is due to the fact that the VM1 does not have enough room to do a resource scaling up. This scaling out takes around 20 seconds for synchronization. It is mainly for Roboconf to know about new instances (including the second instance of the WebappA), for the Apache Load Balancer to recognize the new Tomcat and for Docker daemon in VM2 to understand its new container. The SUT at this moment changes from the state (II) to state (III). At last, the third peak is resolved by a fine-grained scaling up at container level as in the case of the first peak. The VM2 is filled up by a conventional unit of 2GB memory which is added to the container of the second instance of the WebappA. The state (III) becomes state (IV) until the experiment finishes.

The violation rate of the observed response time in the 10 minutes of the experiment using the MFS algorithm can be calculated as follows.

$$V_{MFS} = \frac{5 + 20 + 5}{600}.100\% = 5\%$$

We can measure Virtual Machine Occupation of the SUT to evaluate the effectiveness of the algorithm in terms of resource management toward minimizing the booked VM resource for the cloud customers. According to Tran [114], the Virtual Machine Occupation Ω_j of algorithm *j* is calculated as follows.

$$\Omega_j = \sum_{k=1}^{n} w_k \tag{7.1}$$

With w_k is the occupation of a resource type in a VM *k* (memory in our case) over the experiment time. It is calculated by Equation 7.2 with a given cap of a resource, we have $0 < c_{k,i} \leq 9$ (our VM has 9GB memory totally) in each duration $t_{k,i}$ (in seconds) allocated to a virtual machine VM_k.

$$w_k = \sum_{i=1}^{n} c_{k,i}.t_{k,i} \tag{7.2}$$

We see that the lower Ω_j, the less waste for booked resource of VM. Low Ω_j confirms the effectiveness of a provisioning policy. The customer saves cost if the scaling algorithm provides low Ω_j in the experiment.

Apply the Equation 7.1 to the MFS algorithm with n=2 (VMs):

$w_{VM1} = 7.60 + 9.180 + 9.180 + 9.180 = 5280$

$w_{VM2} = 3.60 + 3.180 + 7.180 + 9.180 = 3600$

$\Omega_{MFS} = 5280 + 3600 = 8880$

For comparison, we also conduct the same experiment on Roboconf with scaling-out algorithms implemented only at container level (without the Algorithm 7). The main differences are shown in Figure 7.8 and can be explained by state transitions depicted in Figure 7.9.

The first peak is resolved by a container-level scaling out because the amount of unused memory (2GB) in the VM1 is not enough for creating a new container (requires 4GB). Thus the MFS had to put the new container in the VM2 where can provide enough room for it (6GB available so far). In the state (II), each VM consumes 7GB memory and leave 2GB free. These two 2GB unused memory become resource holes that cannot be used for scaling out WebappA because of their small size. It leads to creation of an entire new 9GB VM3 which hosts a new 4GB container of WebappA in the state (III). It takes around 140 seconds for synchronization, mainly devotes for VM provisioning. This VM3 has enough room for creating a new container to overcome the third peak. The experiment ends in the state (IV) with 3 resource holes as shown in the Figure 7.9.

The violation rate of the observed response time in the 10 minutes of the experiment without using full MFS algorithm can be calculated as follows.

$$V_{wo-full-MFS} = \frac{20 + 140 + 20}{600}.100\% = 30\%$$

Apply the Equation 7.1 to the MFS algorithm with n=3:

$w_{VM1} = 7.60 + 7.180 + 7.180 + 7.180 = 4200$

$w_{VM2} = 3.60 + 7.180 + 7.180 + 7.180 = 3960$

$w_{VM3} = 4.180 + 8.180 = 2160$

$\Omega_{wo-full-MFS} = 5280 + 3600 + 2160 = 10320$

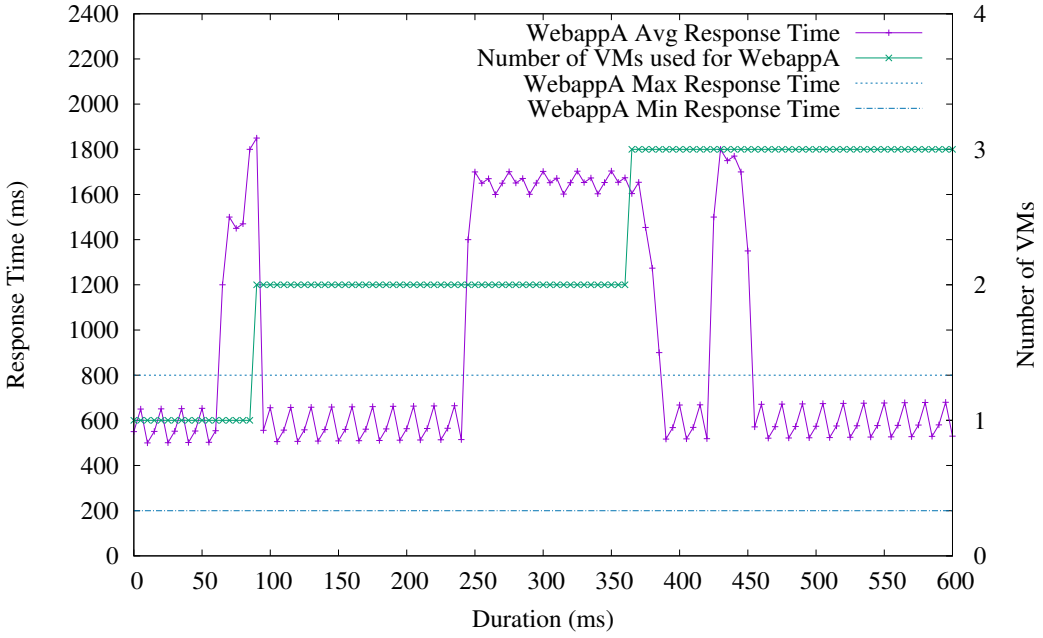

Figure 7.8: Autonomic responses with fluctuation of average response time of WebappA without full MFS algorithm

From calculated results, when applying full MFS, the ART violation rate reduces significantly from 30% to 5%. This helps SaaS providers to avoid penalties because of SLA violation. Moreover, full MFS makes PaaS providers use their allocated VMs more efficient and saving ($8880 < 10320$). This can easily be seen when the SUT had to use up three VMs in the case of not using full MFS, while it was two in the case of using the full one, be considered in the same period of time.

7.3 Synthesis

To assess efficiency of multi-level fine-grained algorithms for elasticity, we have implemented two experiments with the RUBiS distributed application on Roboconf platform. The first experiment is to evaluate the multi-level elasticity (MS algorithm) while the second one evaluate the combination between multi-level and fine-grained approach (MFS algorithm). Both experiments have been conducted on multi-cloud scheme. Results show that the two algorithms not only reduce the provisioning time in scaling actions but also alleviate the SLA violation rate

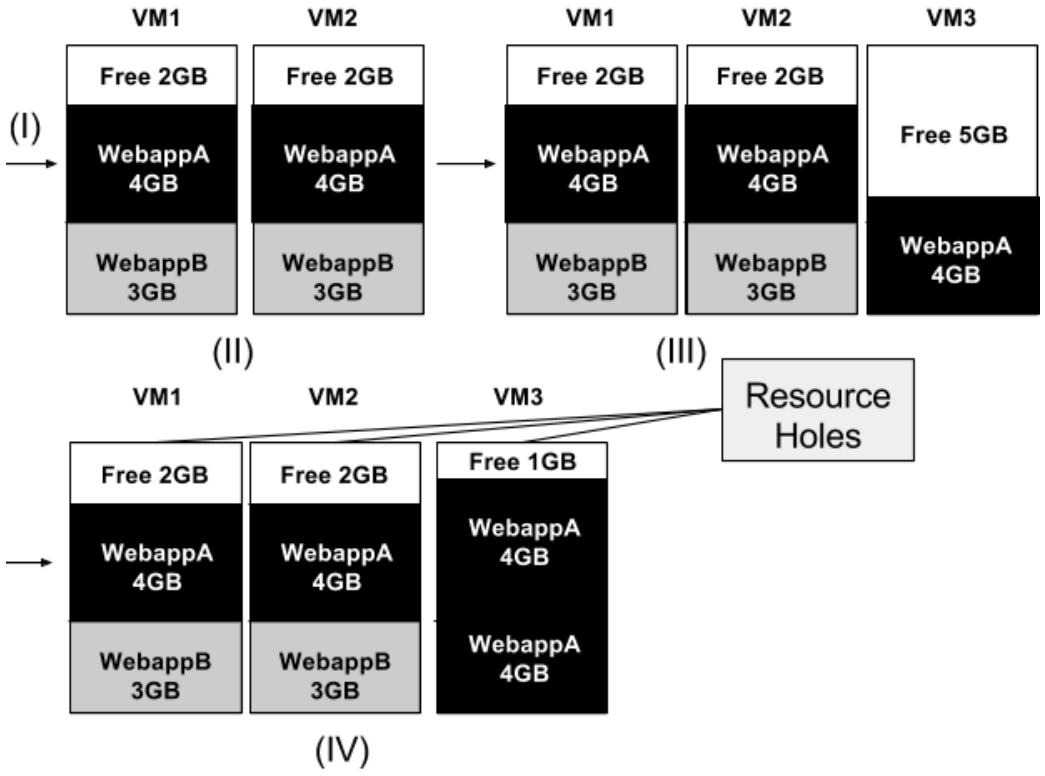

Figure 7.9: States of the experiment without full MFS algorithm

in terms of average response time of web application on the Cloud. To this end, the full MFS showed a better performance than the not full one. Furthermore, the MFS algorithm also increases the VM occupation, which helps IaaS providers and customers cut costs as well as exploit their resources more efficiently.

Chapter 8

EVALUATION OF THE ROBOCONF PLATFORM

Contents

To support effectively for the multi-level fine-grained elasticity, it is necessary to have a stalwart ACS platform. Thus in this chapter, we evaluate Roboconf itself as an autonomic platform which supports for multi-level fine-grained elasticity. Section 8.1 presents experiments to validate Roboconf features needed for our proposed elasticity solution. Section 8.2 details use cases as prototypes implemented from Roboconf application model and deployed in multi-cloud environments.

8.1 Experiments

As mentioned in Chapter 6, Roboconf platform provides the following features which are very positive supporters for elasticity: component fine-grained hierarchical description, dynamic dependency resolution, concurrent component deployment, multi-cloud distributed deployment, genericity, extensibility, scalability, and dynamic reconfiguration of the deployment plans. To validate those non-functional properties, we conducted a number of experiments with scenarios selected from practical use cases. The elasticity experiments in Chapter 7 validated the scalability and dynamic reconfiguration features. In this section, various other experiments are implemented on different types of application to prove the remaining features of Roboconf.

8.1.1 Experiment 1

The first type of experiments validates Roboconf in terms of dynamic dependency resolution and concurrent component deployment. To this end, we dissect Roboconf deployment process and compare it with state-of-the-art deployment platforms: Cloudify, RightScale, and Scalr (that all support concurrent deployment of VMs) in terms of deployment time. Deployment is repeated 8 times for each platform and the means are reported.

Scenario and Requirements

For this experiment, we chose EC2 as the target cloud and Puppet as the Roboconf installer plug-in. We started with a simple LAMP application which is implemented with all-in-one style on EC2 m3.medium VM instances (Ubuntu 12.04 with 1vCPU, 3.75GB RAM and 1x4GB SSD storage). The deployment is considered successful if user can connect and log into phpMyAdmin using any web browsers and start to create a database. The deployment order follows Table 8.1. As mentioned earlier, this experiment was performed on three deployment platforms. Since each of them has its own states of life cycle, without loss of generality, we distribute those states into two main phases based on classification of [12]:

- *Booting phase*: In this phase the deployment frameworks spend time to process following actions: send “Deploy” requests from client (client interface of deployment platform) to DM, DM processes the requests, transfer scripts and other necessary files to itself then sends hiring requests to IaaS, IaaS provisions and powers up needed VMs, run booting scripts (setup agents, send information of booting machine back to DM).

Table 8.1: Deployment Order of the LAMP Application

Order	Operation	Order	Operation
0	Provision + Boot VM	2	Start - Apache
1	Deploy - Apache	2	Start - MySQL
1	Deploy - MySQL	3	Start - phpMyAdmin
2	Deploy - phpMyAdmin		**Done!**

- *Operational phase*: In this phase the deployment platforms consume time to execute operational scripts (or recipes) run once a server is running, on services or components. It may include states: preInstall (download tarball or/and transfer scripts and necessary files to VM, prepare runtime environment, etc), install, postInstall (copy resources and configuration files to right place, set permissions, etc), preStart (resolves dependencies), start, postStart (update variables, configure monitoring), etc.

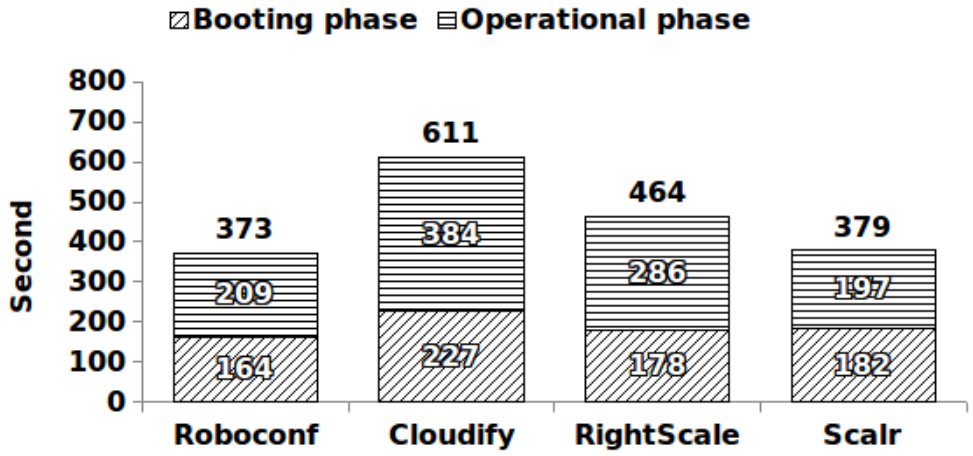

Figure 8.1: Deployment time with different deployment systems

Results

Figure 8.1 presents results of this experiment. In the operational phase, time is measured until last component is installed. We can see that Roboconf outperforms the others in terms of total deployment time, with Scalr being the nearest one (hence we only discuss about it). The runner-up, Scalr, took less time in the

operational phase than Roboconf because it was not consuming time for dynamic dependency resolution. In Roboconf, the dependency resolution is conducted dynamically at runtime as mentioned in Section 6.3.2. In Scalr, dependencies amongst components are resolved manually by configuring exchanged variables in its Web UI. In fact, it is a tedious, error-prone and time consuming job. We also found that in the booting phase, factors making difference are the processing requests and setting up agents, while dependency resolution is mainly diverse element in the operational phase.

8.1.2 Experiment 2

The second type of experiments demonstrates the advantage of the fine-grained hierarchical description provided by Roboconf DSL and its component reusability. For this experiment, EC2 was the target cloud.

Scenario and Requirements

We performed this experiment with an OSGi-based application (the JMS part of the SPECjms2007 benchmark). Regularly, an OSGi application is implemented on an OSGi container or platform (e.g. Karaf, Felix, Equinox) providing runtime environment and management framework for OSGi bundles such as Joram, JNDI, etc. We used two instances of EC2 m3.medium, each hosts two instances of the Karaf container. Each Karaf of a VM is customized to choose either Felix or Equinox as underlying OSGi platform and hosts an instance of Joram (an OSGi-based JMS-supported server), or an instance OSGi-based JNDI or a OSGi JMS client (publisher/subscriber). Deployment of Joram, JNDI and OSGi JMS clients is handled by the "osgi-bundle" installer, specific to this type of application. We chose Cloudify as comparative objective because it also offers scripting language that can be used to express the structure of a distributed application. However, Roboconf can describe the hierarchy of this application whereas Cloudify can only see the application as a flat structure. The comparison between these two views is shown in Figure 8.2.

Results

With its hierarchical DSL and extensibility, although both solutions need to write 6 sets of deployment and configuration (D&C) scripts for 6 components, Roboconf users only have to write one D&C script for EC2, one for Karaf and reuse one for multiple OSGI bundles (Joram, JNDI, subscriber, publisher). In the case of Cloudify, 6 D&C scripts are needed, each one for each component (EC2, Karaf, Joram, JNDI, Subscriber, Publisher). Table 8.2 shows statistics about number of

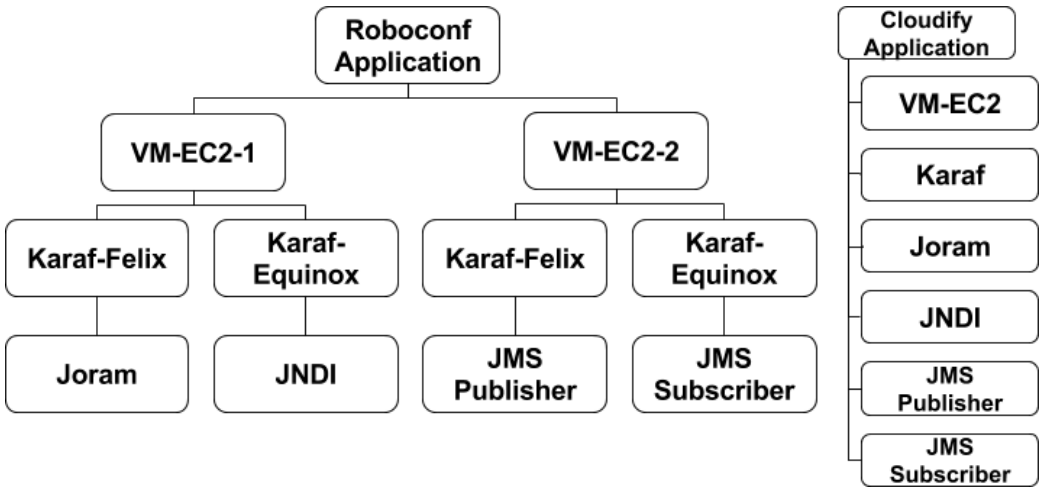

Figure 8.2: OSGi application: Roboconf hierarchical view vs. Cloudify flat view

Table 8.2: Number of D&C Scripts of the OSGi Application

Number of Plan	**Roboconf**	**Cloudify**
EC2	1	1
Karaf	1	1
Joram/JNDI/Pub/Sub	1	4
Total	**3**	**6**

the D&C scripts for Roboconf and Cloudify, respectively. In this case, Cloudify users have to write twice more D&C scripts than Roboconf ones. More details about reusability of component descriptions in the Roboconf DSL can be consulted back at Section 5.5.2.

8.1.3 Experiment 3

The third type of experiments gives some evidences for the correctness of Roboconf multi-cloud distributed deployment feature and its extensibility using target and agent plug-ins.

Scenario and Requirements

We compare deployment time of a Storm cluster [128] (an Event Stream Processing (ESP) application) on multi-cloud platforms using on the one hand the Roboconf platform and on the other hand a manual D&C process following installation guide from original owner. Storm is a part of a global solution for big

data analysis. Storm consists of Zookeeper cluster, Nimbus server, Storm supervisors and requires installation of JZMQ, ZeroMQ and Python. Figure 8.3 shows main components and inter-dependencies of the Storm cluster, which is equivalent to a graph definition in Figure 8.4.

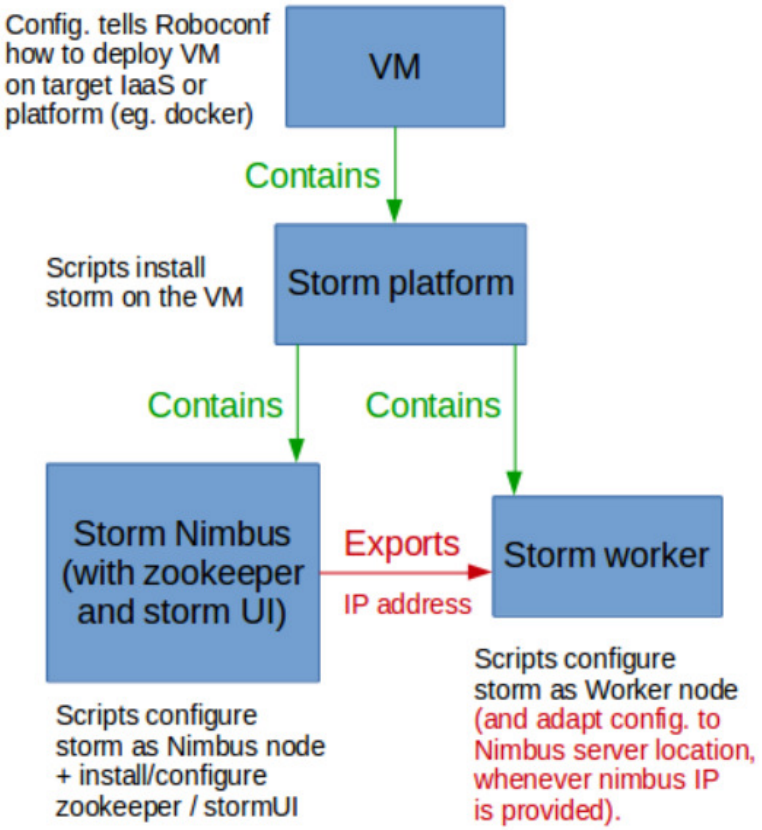

Figure 8.3: Components and inter-dependencies of the Storm cluster

The experiment was conducted in a multi-cloud environment combining two public clouds (EC2 and Azure) and a private cloud (VMware vSphere). The VM EC2 m3.medium instances are equipped with 1 core and 3.75 GB memory while the VM Azure Standard A2 instances are with 2 cores and 3.5 GB memory. Each VM in the VMware vSphere cloud is distributed 1 core and 2GB memory. The guest OS installed in all the VMs is Ubuntu 12.04. Three IaaS targets for these clouds have been developed to provide coordination among the three IaaS providers. Each target needs to implement one target handler interface of the Roboconf Target Handler API (see Section 6.2.4). The LOCs (lines-of-code) for the EC2 target is about 200, for the Azure is about 380, and for the VMware

```
# The VM
VM {
    installer: target;
    children: storm_platform;
}

# Storm base platform
storm_platform {
    installer: script;
    children: storm_nimbus, storm_worker;
}

# Storm nodes

# Storm master node (Nimbus, along with
# zookeeper + stormUI)
storm_nimbus {
    installer: script;
    exports: ip;
}

# Storm worker (slave) node
storm_worker {
    installer: script;
    imports: storm_nimbus.ip;
}
```

Figure 8.4: Component graph of the Storm cluster described under Roboconf DSL

vSphere is about 225 (see Appendix). Zookeeper cluster was installed on EC2 cloud, Nimbus server on Azure cloud and Storm supervisors on our VMware vSphere data-center to take advantage of our computing strength. In this experiment, the time for installing Storm manually is compared with the time to automate its installation using Roboconf.

Results

The online installation guide of Storm cluster is 8-page length specific to Storm itself and many external links to resource document of relevant dependent software. One of the author's colleague who had no knowledge about Storm and have never attempted to install this software previously tried to do manual installations.

Table 8.3: Execution Time and Additional Cost

Scenario	**Average execution time**	**Overhead introduced by Roboconf**
Application	8.95	-
Application + Roboconf	9.05	1.12%

It took him about 6 hours the first time, 3 hours and 30 minutes the second time, and up to 1 hour from the third one. Actions eating effort time were reading imprecise instructions, resolving environmental issues, seeking/downloading the required dependencies and debugging problems. On the Roboconf side, the same work has been carried out by another colleague who also has never known about Storm. With this approach, time devoted mainly for writing component descriptions and D&C scripts of Zookeeper, Nimbus, Supervisors, JZMQ, ZeroMQ and Python. About 120 LOC have been written for D&C scripts of all Storm components. These D&C scripts can be found in Appendix.

After installation, Storm cluster can be managed (deploy, start, stop, undeploy, update) via Roboconf and automatically connect to other applications. At the first time, total development time for Storm cluster in Roboconf was about 2 hours 15 minutes. This time was divided into 30 minutes for design of components, 70 minutes for writing the scripts and 35 minutes for debugging and testing. If the required packages are downloaded from the Internet, installation of Storm cluster needs 20 minutes and it takes around 7 minutes if the packages are retrieved from a local repository. The automation of the Storm cluster installation via Roboconf empowers Storm developers able to deploy their existing applications on multicloud with slight changes and no need to understand details of Roboconf. It warrants a repeatable procedure and can be used as a part of larger deployments (e.g. Ubiquitous analytics).

8.1.4 The Overhead of Roboconf

To evaluate the overhead introduced by the Roboconf platform, 1000 requests were generated and sent to the y-cruncher benchmark [129] to calculate the PI (π) to a specified number of digits (50 in this experiment) after the decimal point. We evaluated two cases: i) application deployed without Roboconf and ii) application deployed with Roboconf. The experiment is conducted on the VM Azure Standard A2 instances (2 cores and 3.5 GB memory). The requests sent were executed 20 times in both scenarios. Table 8.3 presents the results of the average execution time of each scenarios as well as the additional costs brought by the Roboconf platform.

From the results presented in Table 8.3, we can see that the overhead introduced by the Roboconf platform is only 1.12%. This additional cost is generated mainly by monitoring module which collects information for the elasticity autonomic mechanism. In summary, the overhead introduced by the Roboconf platform is negligible given the aforementioned advantages.

8.2 Use Cases

Furthermore, Roboconf has been used for the deployment of two use cases of practical projects: an Enterprise Social Network application (ESN) and a Cloud infrastructure for real-time ubiquitous big data analytics (CIRUS).

8.2.1 Enterprise Social Network (ESN)

Linagora is a French IT service company developing and hosting enterprise messaging solutions and identity management for their customers which are government agencies and SMEs. Linagora hosts its solution on its own private cloud managed with the OpenStack cloud management platform. The capacity of its hosting center can be busted during peak load, by requesting additional resources from a public cloud. As an open-source software editor, Linagora aims to provide a SaaS platform for its multi-tenant ESN. For the current release of the Linagora ESN, the main software components and runtimes are Node.js for running the webapp, MongoDB for data storage, Redis for publish-subscribe asynchronous messaging, ElasticSearch for text indexing as well as NPM and Git for module deployment and update of the application. The components are distributed on several Linux VMs for fault-tolerance and scalability on the Linagora private cloud and its external public cloud (AWS). The next release of the ESN will include more components such as LinShare for file transfer in organizations and Apache James for users mailing, which is backed up by a Cassandra database. The ESN components are globally managed with Roboconf for their quick provisioning and for the VM horizontal scaling on the IaaS of Linagora. Roboconf was chosen by Linagora instead of Cloudify and Scalr because firstly, it facilitates the continuous delivery of the ESN SaaS and secondly, it enables to burst its private cloud with the most affordable public IaaS on the market and without being vendor-locked. Figure 8.5 shows the ESN architecture.

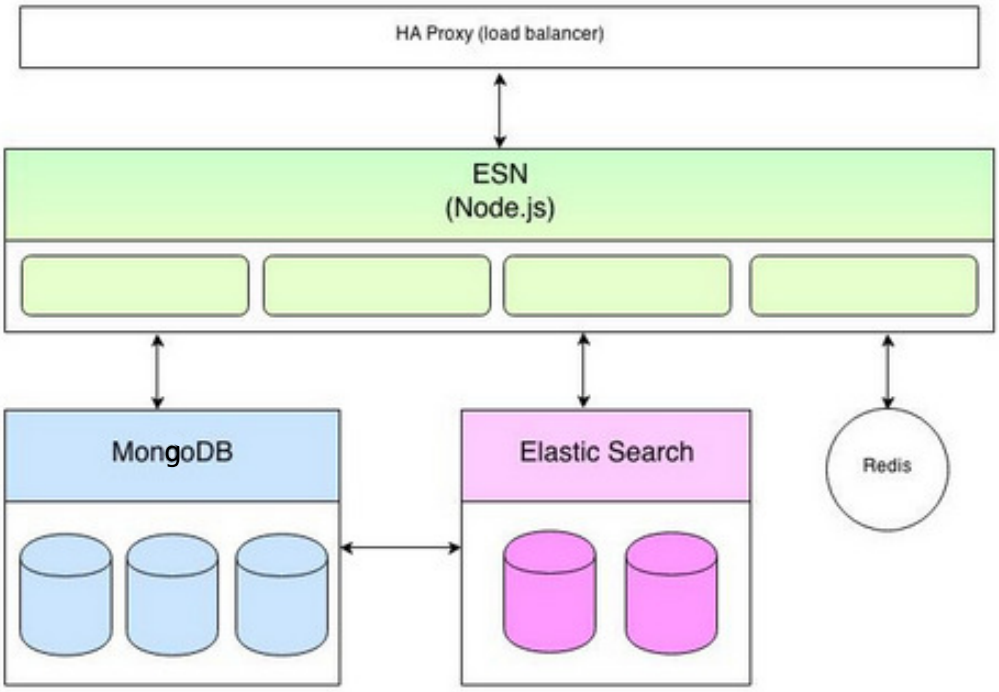

Figure 8.5: The ESN architecture

8.2.2 Cloud Infrastructure for Real-time Ubiquitous Big Data Analytics (CIRUS)

Ubilytics (ubiquitous big data analytics) [130] is one of the trendy topic in the Internet of Things. For instance, it enables a smart-grid provider to forecast region-wide load demand in the next minutes (1 min, 5 min, 15 min, 60 min) from the instantaneous data (load, indoor and outdoor temperatures, etc.) collected from individual smart plugs, thermometers and from history and habit of customers. The dataset was provided by SAP for the DEBS 2014 Grand Challenge. It contains measurements for 40 houses during 1 month. The size of the dataset is about 23 GB of raw data.

CIRUS is a self-adaptive PaaS infrastructure for real-time ubilytics, developed by the LIG lab. The infrastructure is composed of components deployed both on embedded boards and on an IaaS as depicted in Figure 8.6. On one hand, Roboconf manages the deployment of components such as OpenHAB/Eclipse Smart Home (a popular and open-source home automation platform developed by the Eclipse Foundation) on 20 embedded gateways (BeagleBone Black) and on 20 OpenHAB processes in EC2 VMs, which emulate the 40 houses as home automa-

tion boxes. On the other hand, Roboconf provisions Azure VMs and deploys on them a couple types of MQTT brokers (e.g. Mosquitto [132]) or RabbitMQ), a clustered Storm topology for real-time event stream processing, a clustered Cassandra DBMS [131]) for temporal series storage and various dataviz as well as dashboard web consoles for forecasters. As everyone knows, a demonstration, a system tuning, a continuous integration or a benchmark experiment of a distributed system are composed of a set of repetitive tasks, which are fastidious. Roboconf is very helpful in this context since it automates all the tasks and allows the users to safely manipulate a part of the application, without breaking down the rest. It also enables to stop running components and to release the VMs in order to save money on the IaaS account of experimenter. Figure 8.6 shows the CIRUS infrastructure implemented by Roboconf and its components are described as follows.

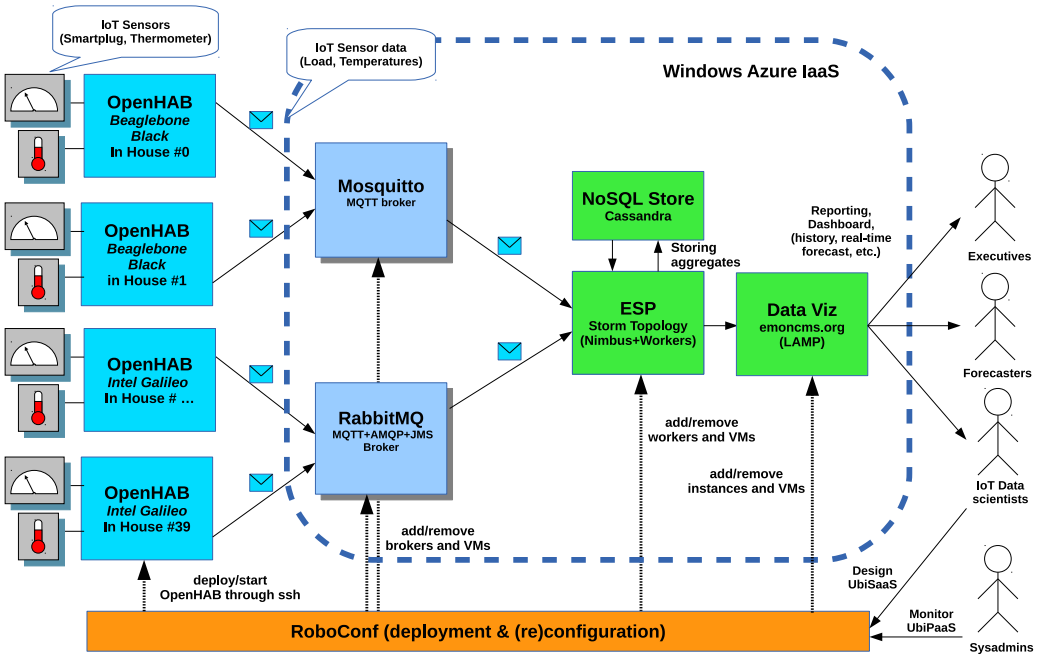

Figure 8.6: Real-time ubilytics scenario with Roboconf

IoT Gateways

For the sensor data collection, we have chosen the OpenHAB platform which provides an integration platform for sensors and actuators of the home automation. The OpenHAB platform is based on the Eclipse Equinox OSGi platform [138]. The communication paradigm amongst the inner components of OpenHAB is

Pub-Sub [137]. OpenHAB allows the users to specify DSL-based rules which will be parsed by its rule engine to update the commands of actuator upon the state changes of sensor using the OSGi Event Admin internal broker. The ECA paradigm is used by OpenHAB for executing the home automation actions. The OpenHAB rule engine evaluates and executes ECA rules which are written in a DSL based on Eclipse XText and XTend. ECA rules are triggered on sensor value changes, command emission and timer expiration. Events (e.g. state changes and commands) can be "imported" or "exported" using bindings for MQTT, XMPP, Twitter, etc. OpenHAB can be installed and run on embedded boards, some of which are Raspberry Pi, Beaglebone Black and Intel Galileo. For the smart-grid use case, we have developed a new OpenHAB plugin (called binding) in order to replay the sensors log files containing the smart-plug measurements (e.g. timestamped load and work) of each house. OpenHAB-CIRUS is the packaging of OpenHAB for the Ubilytics application including the plug-in and the data files. This package is deployed on both embedded boards and virtual machines of the Azure IaaS with one instance per house.

MQTT Brokers

MQ Telemetry Transport (MQTT) [148] is a transport data protocol for M2M networks. It is devised for supporting low-bandwidth and unreliable networks, as illustrated by satellite links or sensor networks. MQTT follows the pub-sub pattern between the sensors and one or more sinks like M2M gateways, etc. MQTT is now an OASIS standard. The main robust and open-source implementations of MQTT brokers are Mosquitto and RabbitMQ.

Speed Layer for Real-time Analytics

For the speed layer of the lambda architecture implemented in the smart-grid use case, we have chosen the Apache Storm cluster. Storm is a real-time event-stream processing system. It is designed to deploy a processing chain in a distributed infrastructure such as a Cloud platform (IaaS). Storm can be applied successfully to the analysis of real-time data and events for sensor networks (real-time resource forecasting, consumption prediction), log files system (monitoring and DDoS attack detection), finance (risk management), marketing and social networks (trend, advertising campaign). Initially developed by Twitter, its challengers are Apache S4 (Yahoo!), Spark Streaming, Millwheel (Google), and Apache Samza (LinkedIn). For the ubilytics platform, we have developed a new Storm input components (called spout) in order to generate sensor tuples from the MQTT brokers by subscribing on the MQTT topics with one spout per house.

Historical Data Storage

In the speed layer, the Storm topology needs to maintain some execution ongoing state. This is the case for the sliding window average of sensor values. To do this we use Storm with Cassandra for our real-time power consumption prediction. Cassandra is an open source distributed database management system (NoSQL solution). It is created to handle large amounts of data spread out across many nodes, while providing a highly available service with no single point of failure. Data model of Cassandra allows incremental modifications of rows.

Visualization Dashboard

For the forecast visualization, we have developed a simple dashboard displaying charts of current and forecast consumptions to supplier and consumers. The dashboard is a simple HTML5 webapp using the Grafana, Bootstrap and Angular Javascript libraries. The webapp gets the data from the historical storage and subscribes to real-time updates through a websocket.

Event Processing Topology for Consumption Forecast

A Storm topology uses Cassandra to store the aggregated consumption values per sensor and per house during a specified period (slices of $|s|$ seconds). This allows not to use the memory as storage. Indeed since the number of houses might be very huge, the volume of data to handle also becomes huge. It is much more efficient not to use the memory to avoid failures like "out of memory". Further predictions can be performed by reading from Cassandra the consumption values previously stored. These predictions are also stored in Cassandra allowing to evaluate the error rate.

The main advantages with this approach is that the consumption and prediction values are persistent allowing to implement more elaborated prediction methods by taking into account the previous records for the predictions and consumptions. Secondly, storing these values in Cassandra allows to implement stateless bolts. This facilitates to dynamically adjust the degree of parallelism of the bolts at runtime depending on the workload in order to meet the performance objectives.

Figure 8.7 is an excerpt demonstrating the CIRUS components under Roboconf DSL.

8.3 Synthesis

In this chapter, the experimental evaluation is organized to validate the features supporting for the multi-level fine-grained elasticity of Roboconf platform. The

```
# An Azure VM
VM_AZURE {
  alias: VM Azure;
  installer: iaas;
  children: Storm_Cluster,
            Cassandra;
}

# A BeagleBone Black
BOARD_BEAGLEBONE {
  alias: BeagleBone Black;
  installer: embedded;
  children: OpenHAB;
}
```

```
# Storm Cluster for ESP
Storm_Cluster {
  alias: Storm Cluster;
  installer: bash;
  imports: Nimbus.port,
           Nimbus.ip;
  children: Nimbus,
            Storm_Supervisor;
}

# OpenHAB: A Home
# Automation Bus
OpenHAB {
  alias: OpenHAB;
  installer: puppet;
  exports: ip,
    brokerChoice = Mosquitto;
  imports: Mosquitto.ip,
    Mosquitto.port;
}
...
```

Figure 8.7: Components of CIRUS under Roboconf DSL

first experiment justifies Roboconf in terms of dynamic dependency resolution and concurrent component deployment with a simple LAMP application. We measure Roboconf deployment time and compare it with state-of-the-art deployment platforms. At this end, Roboconf outperforms the others in terms of total deployment time. The second one uses an OSGi-based application to show benefit of using a hierarchical language like Roboconf DSL to describe multi-level applications. It saves time and labor effort while working on reusable and inherited components. The third experiment gives some evidence for the correctness of Roboconf multi-cloud distributed deployment feature and its extensibility using target interfaces and plug-ins. Like any other solutions, the Roboconf platform is likely to introduce an additional overhead. The results obtained allow us to highlight Roboconf as a low cost deployment platform. Compared to its benefits, this introduction of overhead is negligible.

To assess the model of Roboconf applications, we presented two practical use cases (ESN - an Enterprise Social Network application and CIRUS - a Cloud infrastructure for real-time ubiquitous big data analytics) deployed in a multi-cloud environment by Roboconf as proof-of-concept implementations.

Chapter 9

CONCLUSION AND PERSPECTIVES

Contents

9.1 Summary

Elasticity is one of the precious gifts coming from Cloud computing. However there are some causes which make current elasticity solutions not be able to thoughtfully resolve challenges of elasticity. First one comes from rapidly changing and heterogeneous environment of the Cloud itself. This implies introduction of new cloud technologies which might not be standardized. Elasticity solutions thus cannot foresee these new technologies being put into the Cloud day in and day out and often get stuck in proprietary technologies. Second one is that cloud applications are increasingly complex, which consist of multiple components developed, deployed and managed by different software platforms in various languages. This requires that elasticity solutions must take into consideration the structure of the applications that they impact. The third one is that elasticity solutions for huge and complex applications will quickly make rented resources be depleted blurring the sense of an unlimited Cloud. Therefore efficient resource

utilization for scaling using smart algorithms on lightweight fine-grained resource also plays a very important role.

After a comprehensive survey about elasticity research and solutions using analytic grid, the class "Level" is proposed to be added to the elasticity taxonomy in order to supplement a missing concern about granularity of elasticity actions. The thorough analysis about elasticity research in general and the scaling actions (horizontal, vertical and migration) used for elasticity in particular shows that there still have a lot of new issues of elasticity needed to be resolved. Our first contribution is the proposal of using the multi-level fine-grained elasticity as a new approach to partly solve these newly rising issues which are the resource availability, resource granularity, startup time, container-based virtualization and composability. In our approach, resource objects and their granularity of elasticity actions are especially concerned. We figured out that a resource type may have containment or/and runtime relationships with other resource types. Thus the resource and its dependencies could be described hierarchically (in multiple levels) at design time. It leads to a demand of considering the priority of resources on which the elasticity actions should be applied (which resource types at which levels). Our approach suggests that more fine-grained resource types should be taken into account firstly because it costs less when implementing the smaller types of resources. With our approach, resources are used more efficient and economical resulting in more resource available. Our approach also encourages applying elasticity actions on the more fine-grained resources. This might take advantage of the strength of newly virtualized resource types such as lightweight container or elastic VM. These resource types reduce significantly the spin-up time resulting in more speedy elasticity.

To support the multi-level fine-grained elasticity, we need to develop and use an elasticity manager satisfying the extra requirements: rapidness, component fine-grained hierarchical description and deployment, multi-cloud deployment, genericity and extensibility. To do that, the elasticity manager needs to implement an autonomic loop for dynamic scaling and adaptation. To this end, Roboconf expresses the noticeable features of an ACCS, thus it is also an ACCS. Roboconf platform, such an autonomic elasticity manager, is introduced as our second contribution. First, Roboconf introduces a hierarchical DSL which enables ability to describe complex cloud applications as set of components (software components or infrastructure components). These components have containment or runtime relationships declared in a Roboconf graph. The Roboconf DSL also advocates constructing the elasticity rules which might implement the multi-level fine-grained approach. These rules cause elasticity actions on different component instances of targeted application. Second, the design detail of Roboconf platform is described as set of modules. These modules offer the unique features which are the component fine-grained hierarchical description, dynamic dependency reso-

lution, concurrent component deployment, multi-cloud distributed deployment, genericity, extensibility, scalability, and dynamic reconfiguration of the deployment plans. All these features advocate well for the multi-level fine-grained elasticity. Roboconf Targets and Plug-ins are the mechanisms to extend Roboconf. It helps integrate more modules to the software repositories of Roboconf. Roboconf implement the MAPE-K autonomic loop to automate deployment and reconfiguration of Roboconf applications. Using pub-sub paradigm to exchange dependent variables among instances, Roboconf offers asynchronous deployment and dynamic reconfiguration. Elasticity management based on the autonomic modules of Roboconf is also discussed.

Finally two series of experiments have been conducted to evaluate the multi-level fine-grained elasticity approach and the Roboconf platform itself. Two experiments in the first series propose the novel multi-level fine-grained elasticity algorithms which can be described under Roboconf DSL for elasticity as well as deployed and monitored by Roboconf autonomic platform. These experiments have been performed with multiple levels of resource granularity such as VM, lightweight container, software container, software component. The result of these experiments show that the novel approach reduces resource startup time when using combination of both coarse-grained and fine-grained resources. This reduces total resource provisioning time taken for an elasticity solution. It consequently leads to the mitigation of SLA violation rate (in terms of average response time) as well as the more efficient and economical use of rented resources. The second serie includes three experiments conducted to evaluate the Roboconf platform itself in term of various features that it offers to support the multi-level fine-grained elasticity. The first experiment dissects Roboconf deployment process and compare it with the state-of-the-art deployment platforms: Cloudify, RightScale, and Scalr. The experiment gave the results showing that Roboconf outperforms the others in terms of total deployment time. The second in the serie is to demonstrate the advantage of the hierarchically fine-grained description provided by Roboconf DSL. Result of the experiment on design and deployment of an OSGi-based application shows that Roboconf deployers need less time and effort than its competitor, Cloudify platform, due to nature in the containment-relationship description and its component reusability. The last one devotes to give some evidences for the correctness of Roboconf multi-cloud distributed deployment feature and its extensibility using target and agent plug-ins. The advantage result is for Roboconf when we compared both design and deployment time of Storm platform using Roboconf and the original Storm manual installation guide. These works have been evaluated using private clouds running on VMware vSphere and OpenStack, as well as public ones including cloud providers such as Amazon EC2 and Microsoft Azure.

9.2 Perspectives

9.2.1 Enhancement of the Algorithms of Multi-level Fine-grained Elasticity

In this thesis, we limit ourself with elasticity strategies of reactive mode (see Figure 3.1). The reactive mode perfectly resolves phenomena which lead to uncontemplated changes of workload such as slashdotting. As reactive mode is suitable with every kind of workload, it is the choice for cases that workload cannot be predicted or formulated under a shape function. However the predictive mode still stands for cases that workload is stable over time or could be formulated. In these cases, workload prediction helps calculating amount of time to pre-provision VMs before actual scales occur. Thus elasticity actions are effective almost immediately. In general, the predictive mode is more complex than the reactive one and often incurs the risk of false prediction (both false positive and false negative) at a certain rate. In spite of that, we believe that a combined solution of both modes is necessary and would be a relevant future contribution.

9.2.2 Variety of Resource Dimensions

Our elasticity approach pays much attention to fine-grained resources, especially the lightweight container. Currently, the conducted experiments on lightweight containers consider to a resource dimension which is memory. To have better evaluation of our algorithms and the implementing platform, it is necessary to integrate more dimensions such as CPU share, number of vCPU, storage and network bandwidth. The combination of multiple dimensions will increase the complexity but it will bring our approach closer to reality. This provides more choices for users to manage their elasticity solution on Roboconf platform.

9.2.3 Mitigation of Lightweight Container Migration Time

In our experiments about the multi-level fine-grained algorithms, we assume that the Docker image of the application have been pre-integrated into the corresponding VMI. This method is feasible for scaling stateless applications that their container could be replicated anywhere without concerning about related ties such as user sessions or behind databases. On the other hand, scaling stateful applications is often related to capture the current state of an application instance into an image and transfer the image to a destination where it will be invoked by another instance. The time that migration process takes depends on the image size and the migration technology. Thus the Docker daemon must wait until this process finishes before the Docker container is actually created and started. Research on

mitigation the migration time is an open issue for applying the young lightweight container technologies into practice.

Bibliography

[1] Kim Weins. Cloud Computing Trends: 2015 State of the Cloud Survey. February 2015. http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2015-state-cloud-survey.

[2] Joseph C. R. Licklider. Topics for Discussion at the Forthcoming Meeting, Memorandum For: Members and Affiliates of the Intergalactic Computer Network. April 1963. http://www.kurzweilai.net/memorandum-for-members-and-affiliates-of-the-intergalactic-computer-network.

[3] Salesforce. 'http://www.salesforce.com," visited on November 2015.

[4] Amazon AWS. 'https://aws.amazon.com/," visited on November 2015.

[5] Amazon EC2. 'http://aws.amazon.com/ec2/," visited on November 2015.

[6] Google App Engine. 'https://appengine.google.com/," visited on November 2015.

[7] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[8] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. A federated multi-cloud paas infrastructure. In Rong Chang, editor, *IEEE CLOUD*, pages 392–399. IEEE, 2012.

[9] Suzy Temate, Laurent Broto, Alain Tchana, and Daniel Hagimont. A High Level Approach for Generating Models Graphical Editors. In *Information Technology: New Generations (ITNG), 2011 8th International Conference on*, pages 743–749. IEEE, 2011.

[10] Kyle Oppenheim and Patrick McCormick. Deployme: Tellmes Package Management and Deployment System. In *LISA '00 Proceedings of the 14th USENIX conference on System administration*, pages 187–196. ACM, 2000.

[11] Hyun J. La and Soo D. Kim. Dynamic Architecture for Autonomously Managing Service-Based Applications. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 515–522. IEEE, 2012.

[12] Kung-Kiu Lau, Ling Ling, and Perla V. Elizondo. Towards composing software components in both design and deployment phases. In *10th International Symposium, CBSE 2007, Medford, MA, USA, July 9-11, 2007. Proceedings*, pages 274–282. Springer, 2007.

[13] Alain Tchana, Suzy Temate, Laurent Broto, and Daniel Hagimont. TUNeEngine: An Adaptable Autonomic Administration System. *International Journal of Soft Computing and Software Engineering*, 3(3):524–535, 2013.

[14] AWS Elastic Beanstalk. 'https://https://aws.amazon.com/fr/elasticbeanstalk/," visited on November 2015.

[15] Cloudify. 'http://www.cloudifysource.org," visited on November 2015.

[16] RightScale. 'http://www.rightscale.com," visited on November 2015.

[17] Scalr. 'http://www.scalr.com," visited on November 2015.

[18] EnStratus. 'http://www.enstratius.com," visited on November 2015.

[19] Jeffrey O. Kephart. Autonomic Computing: The First Decade. In *ICAC '11 Proceedings of the 8th ACM international conference on Autonomic computing*, pages 1–2. ACM, 2011.

[20] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), September 2011.

[21] Ahmed El Rheddane. Elasticity in the Cloud. PhD Thesis, 2015.

[22] Linh Manh Pham, Alain Tchana, Didier Donsez, Noël de Palma, Vincent Zurczak, and Pierre-Yves Gibello. An adaptable framework to deploy complex applications onto multi-cloud platforms. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 365–372. IEEE, 2015.

[23] Linh Manh Pham, Alain Tchana, Didier Donsez, Vincent Zurczak, Pierre-Yves Gibello, and Noel de Palma. Roboconf: a Hybrid Cloud Orchestrator to Deploy Complex Applications. In *Computing & Communication Technologies - Research, Innovation, and Vision for the Future (RIVF), 2015 IEEE RIVF International Conference on*, pages 169–174. IEEE, 2015.

[24] Microsoft Azure. ‘http://windowsazure.com,” visited on November 2015.

[25] OpenHAB. ‘http://www.openhab.org,” visited on November 2015.

[26] OpenStack. ‘https://www.openstack.org,” visited on November 2015.

[27] Jez Humble and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. Book, Addison-Wesley Professional, July 2010.

[28] OSGi. ‘http://www.osgi.org,” visited on November 2015.

[29] Heroku Platform. ‘https://www.heroku.com/platform,” visited on November 2015.

[30] Docker. ‘https://www.docker.com,” visited on November 2015.

[31] Vagrant. ‘https://www.vagrantup.com,” visited on November 2015.

[32] RabbitMQ. ‘https://www.rabbitmq.com,” visited on November 2015.

[33] AMQP. ‘http://www.amqp.org,” visited on November 2015.

[34] M. Vardhan, D. K. Yadav, and D. S. Kushwaha. A service-oriented architecture framework for mobile services. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2012 Sixth International Conference on*, pages 389–394. IEEE, 2012.

[35] Livestatus. ‘https://mathias-kettner.de/checkmk_livestatus.html,” visited on November 2015.

[36] Nagios. ‘http://www.nagios.org,” visited on November 2015.

[37] Shinken. ‘http://shinken-monitoring.org,” visited on November 2015.

[38] CLIF Server. ‘http://clif.ow2.org,” visited on November 2015.

[39] Shuai Zhang, Shufen Zhang, Xuebin Chen, and Xiuzhen Huo. Cloud Computing Research and Development Trend. In *Future Networks, 2010. ICFN ’10. Second International Conference on*, pages 93–97. IEEE, 2010.

[40] Jeremy Geelan. Twenty-one experts define cloud computing. *Virtualization Journal*, January 2009. `http://virtualization.sys-con.com/node/612375`.

[41] Eric Knorr and Galen Gruman. What cloud computing really means. *InfoWorld*, April 2008. `http://www.infoworld.com/d/cloud-computing/what-cloud-computing-really-means-031`.

[42] T. Grandison, E. M. Maximilien, S. Thorpe, and A. Alba. Towards a Formal Definition of a Computing Cloud. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 191–192. IEEE, 2010.

[43] Paul McFedries. The cloud is the computer. *IEEE Spectrum Online*, August 2008. `http://spectrum.ieee.org/computing/hardware/the-cloud-is-the-computer`.

[44] M. Morari. Robust stability of systems with integral control. In *Decision and Control, 1983. The 22nd IEEE Conference on*, pages 865–869. IEEE, 1983.

[45] Shicong Meng, Ling Liu, and Vijayaraghavan Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In *Proceedings of the 11th International Middleware Conference Industrial track*, pages 17–22. ACM, 2010.

[46] Yanyan Zhuang, Justin Cappos, Theodore S. Rappaport, and Rick McGeer. Future Internet Bandwidth Trends: An Investigation on Current and Future Disruptive Technologies. Technical Report Online, NYU, January 2013.

[47] Rackspace. 'http://www.rackspace.com/cloud," visited on November 2015.

[48] Ricky Ho. Between Elasticity and Scalability. Online, July 2009. `http://horicky.blogspot.fr/2009/07/between-elasticity-and-scalability.html`.

[49] Dropbox. 'https://www.dropbox.com/," visited on November 2015.

[50] Nathanael Burton. OpenStack at the National Security Agency (NSA). Keynote, April 2013. `http://www.openstack.org/summit/portland-2013/session-videos/presentation/keynote-openstack-at-the-national-security-agency-nsa`.

[51] NCREN. 'https://k20.internet2.edu/organizations/research-education-network/north-carolina-research-and-education-network-ncren," visited on November 2015.

[52] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The Rise of RaaS: the Resource-as-a-Service cloud. *Communications of the ACM*, 57(7):76–84, 2014.

[53] Yucong Duan, Yuan Cao, and Xiaobing Sun. Various "aaS" of everything as a service. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pages 1–6. IEEE, 2015.

[54] Minqi Zhou; Rong Zhang; Dadan Zeng; Weining Qian. Services in the Cloud Computing era: A survey. In *Universal Communication Symposium (IUCS), 2010 4th International*, pages 40–46. IEEE, 2010.

[55] Frank Leymann. Cloud Computing: The Next Revolution in IT. In *Proc. 52th Photogrammetric Week. W. Verlag*, pages 3–12. Springer Verlag, 2009.

[56] Alpha C. Chiang and Kevin Wainwright. Fundamental Methods of Mathematical Economics. Book, McGraw-Hill Education, October 2004.

[57] OCDA. Master Usage Model: Compute Infrastructure as a Service. Technical report, Open Data Center Alliance, 2012. `http://www.opendatacenteralliance.org/docs/ODCA_Compute_IaaS_MasterUM_v1.0_Nov2012.pdf`.

[58] Edwin Schouten. Rapid elasticity and the cloud. Online, September 2012. `http://thoughtsoncloud.com/index.php/2012/09/rapid-elasticity-and-the-cloud/`.

[59] Reuven Cohen. Defining Elastic Computing. Online, September 2009. `http://www.elasticvapor.com/2009/09/defining-elastic-computing.html`.

[60] G. Galante and L.C.E. de Bona. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270, November 2012.

[61] Rich Wolski. Cloud Computing and Open Source: Watching Hype meet Reality. Online, May 2011. `http://www.ics.uci.edu/~ccgrid11/files/ccgrid-11_Rich_Wolsky.pdf`.

[62] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, pages 23–27, 2013.

[63] Amro Najjar, Xavier Serpaggi, Christophe Gravier, and Olivier Boissier. Survey of elasticity management solutions in cloud computing. In *Advances and Trends in Cloud Computing*, pages 235–263. Springer, 2014.

[64] Emanuel F. Coutinho, Flavio R. D. C. Sousa, Paulo A. L. Rego, Danielo G. Gomes, and Jose N. D. Souza. Elasticity in cloud computing: a survey. In *annals of telecommunications - annales des tlcommunications*, 70(7-8):289–309. Springer, 2015.

[65] AWS Auto Scaling. 'https://aws.amazon.com/autoscaling/," visited on November 2015.

[66] Emiliano Casalicchio and Luca Silvestri. Mechanisms for SLA provisioning in cloud-based service providers. In *Journal Computer Networks: The International Journal of Computer and Telecommunications Networking*, 57(3):795–810. ACM, 2013.

[67] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi. Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1327–1334. IEEE, 2012.

[68] Jonathan Kupferman, Jeff Silverman, Patricio Jara, and Jeff Browne. Scaling into the cloud. CS270-Advanced Operating Systems, 2009. `http://www.cs.ucsb.edu/~jbrowne/files/ScalingIntoTheClouds.pdf`.

[69] S. Genaud and J. Gossa. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 1–8. IEEE, 2011.

[70] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52. IEEE Computer Society, 2010.

[71] Luis Rodero-Merino, Luis M. Vaquero, Victor Gil, Fermin Galán, Javier Fontán, Rubén S. Montero, and Ignacio M. Llorente. From infrastructure delivery to service management in clouds. In *Journal Future Generation Computer Systems*, 26(8):1226–1240. Elsevier Science, 2010.

[72] Junliang Chen, Chen Wang, Bing B. Zhou, Lei Sun, Young C. Lee, and Zomaya AY. Tradeoffs Between Profit and Customer Satisfaction for Service Provisioning in the Cloud. In *HPDC '11 Proceedings of the 20th international symposium on High performance distributed computing*, pages 229–238. ACM, 2011.

[73] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. In *Journal Future Generation Computer Systems*, 32(-):82–98. Elsevier Science, 2014.

[74] Zhipiao Liu, Shangguang Wang, Qibo Sun, Hua Zou and Fangchun Yang. Cost-Aware Cloud Service Request Scheduling for SaaS Providers. *The Computer Journal*, Oxford University Press, 2013.

[75] Qian Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Services Computing, IEEE Transactions on*, 5(4):497–511. IEEE, 2012.

[76] V. Cardellini, E. Casalicchio, F. L. Presti, and L. Silvestri. SLA-aware Resource Management for Application Service Providers in the Cloud. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, pages 20–27. IEEE, 2011.

[77] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceeding SOCC '11 Proceedings of the 2nd ACM Symposium on Cloud Computing*, Article 5. ACM, 2011.

[78] Josep Oriol Fitó, Inigo Goiri, and Jordi Guitart. Sla-driven elastic cloud hosting provider. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 111–118. IEEE, 2010.

[79] Rodrigo. N. Calheiros, Christian Vecchiola, Dileban Karunamoorthy, and Rajkumar Buyya. The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid Clouds. In *Journal Future Generation Computer Systems*, 28(6):861–870. Elsevier Science, 2012.

[80] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 213–220. IEEE, 2012.

[81] T. Hobfeld, R. Schatz, M. Varela, and C. Timmerer. Challenges of QoE management for cloud applications. In *Communications Magazine, IEEE*, 50(4):28–36. IEEE, 2012.

[82] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hossfeld. An evaluation of QoE in cloud gaming based on subjective tests. In *Proceeding IMIS '11*

Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, pages 330–335. IEEE Computer Society, 2011.

[83] Tania Lorido-Botrán, José Miguel-Alonso, and Jose A. Lozano. Auto-scaling techniques for elastic applications in cloud environments. Technical Report, University of the Basque Country, September 2012.

[84] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceeding MGC '10 Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, Article 4. ACM, 2010.

[85] GoGrid. 'http://www.gogrid.com/," visited on November 2015.

[86] Iulian Neamtiu. Elastic executions from inelastic programs. In *Proceeding SEAMS '11 Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 178–183. ACM, 2011.

[87] D. Rajan, A. Canino, J. A. Izaguirre, and D. Thain. Converting a high performance application to an elastic cloud application. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 383–390. IEEE, 2011.

[88] Mohamed Mohamed, Mourad Amziani, Djamel Belaid, Samir Tata, and Tarek Melliti. An autonomic approach to manage elasticity of business processes in the Cloud. In *Journal Future Generation Computer Systems*, 50(C):49–61. Elsevier Science, 2015.

[89] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceeding EuroSys '07 Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 289–302. ACM, 2007.

[90] Rhodney Simoes and Carlos A. Kamienski. Elasticity management in private and hybrid clouds. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 793–800. IEEE, 2014.

[91] Lydia Yataghene, Mourad Amziani, Malika Ioualalen, and Samir Tata. A queuing model for business processes elasticity evaluation. In *Advanced Information Systems for Enterprises (IWAISE), 2014 International Workshop on*, pages 22–28. IEEE, 2014.

[92] Mohamed N. Bennani and Daniel A. Menascé. Resource allocation for autonomic data centers using analytic performance models. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 229–240. IEEE, 2005.

[93] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011.

[94] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559–570. IEEE, 2011.

[95] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. *ACM SIGARCH Computer Architecture News*, 40(1):423–436, 2012.

[96] L. Bellissard, Noël de Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An agent platform for reliable asynchronous distributed programming. In *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, pages 294–295. IEEE, 1999.

[97] H. N. Van, F. D. Tran, and J. M. Menaud. SLA-Aware Virtual Resource Management for Cloud Infrastructures. In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, pages 357–362. IEEE, 2009.

[98] T. C. Chieu TC and Hoi Chan. Dynamic resource allocation via distributed decisions in cloud environment. In *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on*, pages 125–130. IEEE, 2011.

[99] E. Kafetzakis, H. Koumaras, M. A. Kourtis, and V. Koumaras. QoE4CLOUD: A QoE-driven multidimensional framework for cloud environments. In *Telecommunications and Multimedia (TEMU), 2012 International Conference on*, pages 77–82. IEEE, 2012.

[100] Christophe Taton, Noël De Palma, Sara Bouchenak, and Daniel Hagimont. Improving the performances of jms-based applications. *International Journal of Autonomic Computing*, 1(1):81–102, 2009.

[101] Nam-Luc Tran, Sabri Skhiri, and Esteban Zimányi. Eqs: An elastic and scalable message queue for the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 391–398. IEEE, 2011.

[102] Smita Vijayakumar, Qian Zhu, and Gagan Agrawal. Dynamic resource provisioning for data streaming applications in a cloud environment. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 441–448. IEEE, 2010.

[103] Thomas Knauth and Christof Fetzer. Scaling non-elastic applications using virtual machines. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 468–475. IEEE, 2011.

[104] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[105] A. Raveendran, T. Bicer, and G. Agrawal. A framework for elastic execution of existing mpi programs. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 940–947. IEEE, 2011.

[106] Mohamed Mohamed. Generic Monitoring and Reconfiguration for Service-Based Applications in the Cloud. PhD Thesis, 2014.

[107] L. Broto, D. Hagimont, Patricia. Stolf, Noël de Palma, and Suzy Temate. Autonomic Management Policy Specification in Tune. In *Proceeding SAC '08 Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663. ACM, 2008.

[108] Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, Noël De Palma. Reliable Self-Deployment of Cloud Applications. In *Proceeding SAC '14 Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1331–1338. ACM, 2014.

[109] Liang Zhao, Sherif Sakr, and Anna Liu. A framework for consumer-centric sla management of cloud-hosted databases. *IEEE Transactions on Services Computing*, 8(4):1–1, 2013.

[110] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the

cloud. In *Journal Future Generation Computer Systems*, 27(6):871–879. Elsevier Science, 2011.

[111] Francisco Perez-Sorrosal, Marta Patiño Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. Elastic si-cache: consistent and scalable caching in multi-tier architectures. In *Journal The VLDB Journal The International Journal on Very Large Data Bases*, pages 20(6):841–865. Springer Verlag, 2011.

[112] Wesam Dawoud, Ibrahim Takouna, Christoph Meinel. Elastic VM for Cloud Resources Provisioning Optimization. In *First International Conference, ACC 2011, Kochi, India, July 22-24, 2011. Proceedings, Part I*, pages 431–445. Springer, 2011.

[113] Wesam Dawoud, Ibrahim Takouna, Christoph Meinel. Elastic virtual machine for fine-grained cloud resource provisioning. In *4th International Conference, ObCom 2011, Vellore, TN, India, December 9-11, 2011. Proceedings, Part I*, pages 11–25. Springer, 2012.

[114] Giang Son Tran. Cooperative Resource Management in the Cloud. PhD Thesis, 2014.

[115] D. Kumar, Z. Y. Shae, H. Jamjoom. Scheduling Batch and Heterogeneous Jobs with Runtime Elasticity in a Parallel Processing Environment. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 65–78. IEEE, 2012.

[116] OnApp. 'http://onapp.com/," visited on November 2015.

[117] Inkwon Hwang and Massoud Pedram. Hierarchical Virtual Machine Consolidation in a Cloud Computing System. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 196–203. IEEE, 2013.

[118] Ahmed El Rheddane, Noël De Palma, Fabienne Boyer, Frédéric Dumont, Jean-Marc Menaud, and Alain Tchana. Dynamic scalability of a consolidation service. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 748–754. IEEE, 2013.

[119] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.

[120] H. N. Van, F. D. Tran, and J. M. Menaud. Autonomic Virtual Resource Management for Service Hosting Platforms. In *Software Engineering Challenges of Cloud Computing, 2009. CLOUD '09. ICSE Workshop on*, pages 1–8. IEEE, 2009.

[121] S. Dutta, S. Gera, A. Verma, and B. Viswanathan. SmartScale: Automatic Application Scaling in Enterprise Clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 221–228. IEEE, 2012.

[122] A. Ashraf, B. Byholm, I. Porres. CRAMP: Cost-efficient Resource Allocation for Multiple web applications with Proactive scaling. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 581–586. IEEE, 2012.

[123] Felix. 'http://felix.apache.org," visited on November 2015.

[124] Alain Tchana, Noel Depalma, Ibrahim Safieddine, and Daniel Hagimont. Software Consolidation as an Efficient Energy and Cost Saving Solution for a SaaS/PaaS Cloud Model. In *21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 305–316. Springer, 2015.

[125] Sijin He, Li Guo, Yike Guo, Chao Wu, M. Ghanem, and Rui Han. Elastic Application Container: A Lightweight Approach for Cloud Resource Provisioning. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 15–22. IEEE, 2012.

[126] S. Imai, T. Chestna, and C. A. Varela. Elastic Scalable Cloud Computing Using Application-Level Migration. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 91–98. IEEE, 2012.

[127] Mohamed Mohamed, Djamel Belaid, and Samir Tata. Self-Managed Micro-containers for Service-Based Applications in the Cloud. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2013 IEEE 22nd International Workshop on*, pages 140–145. IEEE, 2013.

[128] Apache Storm Cluster. 'http://storm.apache.org/documentation/Setting-up-a-Storm-cluster.html," visited on November 2015.

[129] Y-cruncher Benchmark. 'http://www.numberworld.org/y-cruncher/," visited on November 2015.

[130] Linh Manh Pham, Ahmed El Rheddane, Didier Donsez, and Noël de Palma. CIRUS: an elastic cloud-based framework for Ubilytics. In *annals of*

telecommunications - annales des tlcommunications, 71(1-2):x–x. Springer, 2016.

[131] Apache Cassandra. 'http://cassandra.apache.org," visited on November 2015.

[132] Mosquitto. 'http://mosquitto.org/," visited on November 2015.

[133] Rostand Costaa and Francisco Brasileiro. On the amplitude of the elasticity offered by public cloud computing providers. Federal University of Campina Grande, Campina Grande, Tech. Rep., 2011. `http://www.lsd.ufcg.edu.br/relatoriostecnicos/TR-4.pdf`.

[134] Qingjia Huang, Sen Su, Siyuan Xu, Jian Li, Peng Xu, and Kai Shuang. Migration-Based Elastic Consolidation Scheduling in Cloud Data Center. In *Distributed Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference on*, pages 93–97. IEEE, 2013.

[135] D. Breitgand and A. Epstein. Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds. In *INFOCOM, 2012 Proceedings IEEE*, pages 2861–2865. IEEE, 2012.

[136] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. soCloud: A service-oriented component-based PaaS for managing portability, provisioning, elasticity and high availability across multiple clouds. In *Computing*, pages 1–27. Springer, 2014.

[137] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. In *Journal ACM Computing Surveys (CSUR)*, 35(2):114–131. ACM, 2003.

[138] Eclipse Equinox. 'http://www.eclipse.org/equinox/," visited on November 2015.

[139] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[140] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, and Kun Wang. A distributed self- learning approach for elastic provisioning of virtualized cloud resources. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 45–54. IEEE, 2011.

[141] RUBiS. 'http://rubis.ow2.org," visited on November 2015.

[142] Kim Weins. Cloud Computing Trends: 2014 State of the Cloud Survey. April 2014. http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2014-state-cloud-survey.

[143] Marek Goldmann. Resource management in Docker. September 2014. https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/.

[144] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Rethinking the design of virtual machine monitors. In *Computer*, 38(5):57–62. IEEE, 2005.

[145] Francesc D. Muñoz-Escoí, José M. Bernabeu-Aubán. A Survey on Elasticity Management in the PaaS Service Model. Technical Report, Universitat Politècnica de València, February 2015.

[146] Stefan Poslad. Autonomous systems and Artificial Life. In *Ubiquitous Computing Smart Devices, Smart Environments and Smart Interaction*, pages 317–341. Wiley, 2009.

[147] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[148] MQ Telemetry Transport. 'http://mqtt.org," visited on November 2015.

[149] Daniel Cukier. DevOps patterns to scale web applications using cloud services. In *SPLASH '13 Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 143–152. ACM, 2013.

Glossary

ACCS: Autonomic Cloud Computing System
ACS: Autonomic Computing System
CAP: Cloud Application Provider
CAU: Cloud Application User
CP: Cloud Provider
CP-U: Cloud Provider User
D&C: Deployment and Configuration
DevOps: Development and Operations
DSL: Domain Specific Language
ESN: Enterprise Social Network
ESP: Event Stream Processing
IoT: Internet of Things
PM: Physical Machine
QoBiz: Quality of Business
QoE: Quality of Experience
QoS: Quality of Service
SLA: Service-Level Agreement
SLO: Service-Level Objective
SoA: Service-oriented Architecture
SUT: System Under Test
VM: Virtual Machine
VMI: Virtual Machine Image
VMM: Virtual Machine Monitor

Appendix

- **Supported Targets**:

  OpenStack, EC2, MS Azure, VMWare, Docker, Apache JCloud, Embedded, In-memory.

  `http://roboconf.net/en/user-guide/list-of-deployment-targets.html`

- **Plugin Interface**:

  `https://github.com/roboconf/roboconf-platform/blob/master/core/roboconf-plugin-api/src/main/java/net/roboconf/plugin/api/PluginInterface.java`

- **Target Handler Interface**:

  `https://github.com/roboconf/roboconf-platform/blob/master/core/roboconf-target-api/src/main/java/net/roboconf/target/api/TargetHandler.java`

- **Monitoring Handler Interface**:

  `https://github.com/roboconf/roboconf-platform/blob/master/core/roboconf-agent-monitoring-api/src/main/java/net/roboconf/agent/monitoring/api/IMonitoringHandler.java`

- **Implementation of EC2 and Azure Targets**:

  `https://github.com/roboconf/roboconf-platform/blob/master/core/roboconf-target-iaas-ec2/src/main/java/net/roboconf/target/ec2/internal/Ec2IaasHandler.java`

`https://github.com/roboconf/roboconf-platform/blob/master/core/roboconf-target-iaas-azure/src/main/java/net/roboconf/target/azure/internal/AzureIaasHandler.java`

- **Storm D&C Scripts**:

  `https://github.com/roboconf/roboconf-examples/tree/master/storm-bash/src/main/model`