FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Software-defined Elastic Systems Management in Multi-cloud Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Juraj Čík BSc
Matrikelnummer 1228077

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Privatdoz. Dr.techn. Hong-Linh Truong

Wien, 22. April 2015

Juraj Čík   Hong-Linh Truong

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Software-defined Elastic Systems Management in Multi-cloud Environments

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Juraj Čík BSc
Registration Number 1228077

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dr.techn. Hong-Linh Truong

Vienna, 22nd April, 2015

	Juraj Čík	Hong-Linh Truong

Erklärung zur Verfassung der Arbeit

Juraj Čík BSc
Rolnicka 296, 831 07 Bratislava, Slovakia

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. April 2015

Juraj Čík

Acknowledgements

I would like to express my thanks to Hong-Linh Truong for his guidance and for the opportunity to conduct the thesis at the Distributed Systems Group. I would also like to thank my colleagues from the Distributed Systems Group working on the CELAR project for cooperation, especially Duc Hung Le.

Finally, I wish to thank my family and friends, particularly my parents, for support and encouragement.

This work is supported by the EU FP7 CELAR project under grant № 317790.

Abstract

Elastic systems are characterized by the ability to expand and contract in the cloud environment, through dynamic de/allocation of resources, based on their requirements. These requirements can be described, with respect to the internal structure of the elastic system, as a trade-off between resource, quality and cost. In order to make such systems feasible, a platform specialized on development and operation of elastic systems is needed. Within this platform, supporting services for deployment, monitoring, testing and elasticity control handle all the tasks for the additional aspect of elasticity, leaving the developer's hands free to deal with the system's domain-specific logic. Moreover, extending such a platform over multiple existing clouds results in an integrated multi-cloud environment.

The challenge is to manage elastic systems in this platform throughout their entire lifecycle. This includes coordination of elastic platform services, while providing traceability of actions executed by the services as well as by human users. Furthermore, it is necessary to support extensibility of the platform with new services and offer a uniform interface for interaction with the platform.

This thesis proposes a framework for management of elastic systems in the above described platform. The resulting management system is based on loose coupling of elastic platform services through an observable event-driven communication. The exchange of events is centered around a lifecycle of an elastic system, which serves for coordination of the involved components. The lifecycle reflects platform's use cases and characteristic relations between elastic platform services. The thesis further proposes components of the management framework, including management of dynamic elastic platform services, as well as, recording and analytics of events. Last but not least, a prototype is implemented for evaluation of the framework and of the introduced concepts.

Kurzfassung

Elastische Systeme sind durch die Erweiterbarkeit und die Kontraktilität in der Cloud -Umgebung charakterisiert. Dies geschieht mittels der dynamischen De/Allokation der Ressourcen, basierend auf deren Anforderungen. Diese Anforderungen können im Hinblick auf die interne Systemstruktur als Trade-off zwischen Ressource, Qualität und Kosten beschrieben werden. Um solche Systeme realisierbar zu machen, ist eine auf die Entwicklung und den Betrieb der elastischen Systeme spezialisierte Plattform erforderlich. Die Unterstützungsdienste für die Bereitstellung, Überwachung, Testing und elastische Kontrolle behandeln innerhalb dieser Plattform alle Aufgaben, die zusätzlich aus dem Elastizitätsaspekt hervorgehen. Dank dessen können sich die Entwickler uneingeschränkt mit der domänenspezifischen Logik des Systems befassen. Weiter hat eine Erweiterung einer solchen Plattform um mehrfache vorhandene Clouds eine integrierte Multi-Cloud-Umgebung zur Folge.

Die Herausforderung dabei liegt am Management des elastischen Systems in dieser Plattform während des gesamten Lebenszyklus. Dies beinhaltet die Koordination der Dienste der elastischen Plattform bei der Sicherstellung der Rückverfolgbarkeit der Tätigkeiten, die sowohl durch die Dienste als auch durch die Menschen ausgeführt werden. Darüber hinaus ist es notwendig, die Erweiterbarkeit mit neuen Diensten zu fördern und eine einheitliche Anwendungsschnittstelle für die Interaktion mit der Plattform anzubieten.

Diese Arbeit schlägt einen Rahmen für das Management des elastischen Systems in der oben beschriebenen Plattform vor. Das resultierende Management-System basiert auf einer lockeren Koppelung den Dienste der elastischen Plattform durch eine beobachtbare ereignisgesteuerte Kommunikation. Der Ereigniswechsel fokussiert sich auf den Lebenszyklus des elastischen Systems, das zur Koordination der einbezogenen Komponente dient. Der Lebenszyklus spiegelt die Anwendungsfälle der Plattform wider und charakterisiert die Beziehungen unter der Dienste der elastischen Plattform. Diese Diplomarbeit schlägt weitere zusätzliche Komponente des Management-Rahmens vor, einschließlich das Management der dynamischen elastischen Plattformdienste, als auch die weitere Protokollierung und Analytik der Ereignisse. Zu guter Letzt wird ein Prototyp für die Beurteilung des Rahmens und des vorgestellten Konzeptes implementiert.

Contents

CHAPTER 1

Introduction

One of the main advantages of a cloud environment is the possibility to develop and operate elastic systems, which utilize computing resources allocated dynamically at run-time according to their actual needs [1] [2]. Although this aspect of cloud computing has been acknowledged by many Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) providers, such as Amazon AWS [1], Google App Engine [2], Microsoft Azure [3] and many others, they usually allow elasticity to be expressed only through resources, e.g. CPU, memory usage or number of requests. In contrast to this, elasticity can be defined not only through infrastructure resources, but in a muli-dimensional space as a trade-off between resource, quality and cost [3]. An elastic system, defined in this way, can better adhere specific business requirements and more accurately adapt to real-life scenarios. However, building such elastic system inside an existing IaaS/PaaS would mean an additional complexity of the system, which might cause an unacceptable overhead for the system's developer. Furthermore, to allow elastic systems to expand behind individual clouds, exchange IaaS/PaaS providers and optimize system's cost, it is necessary to manage the elastic systems inside of this multi-cloud environment in a unified way.

These challenges are being addressed in [4], which introduces an architecture and basic principles of a platform capable supporting development and operation of elastic cloud services. The platform acts as an additional layer between a cloud service developer and the existing IaaS/PaaS solutions, presenting them as a unified multi-cloud environment. This platform is composed of services, which support the elastic behavior of the managed cloud services throughout their lifecycle. In this way, the additional complexity and challenges of elastic cloud services, are handled by the platform and not the developer her/himself. We will address these services as Elastic Platform Services (EPSs) from now on.

[1]http://aws.amazon.com
[2]http://cloud.google.com/appengine
[3]http://azure.microsoft.com

Each EPS concentrates on a specific task within elastic cloud service's lifecycle, such as deployment, monitoring, testing or acts as controller of the cloud service's elastic behavior. They can be supplied by different third-party developers, each service offering its functionality through an API independently from each other.

The elastic cloud services are internally composed of service topologies and service units[5]. Service units are atomic elements of the elastic cloud service's logic, e.g. virtual machine or application server. They are grouped in a service topology, which in turn can be part of another topology. We discuss these concepts in the Section 3.1, but for now it is important to know that the EPSs operate on this structure.

1.1 Problem Statement

All the EPSs can be used independently, acting upon a cloud service, without consideration of a broader context. However, this requires a substantial effort on the side of the cloud service's developer. The developer must have a good understanding of what operations does a EPS offer, how should these be executed, in what order and with what input data. Some output data from one EPS, may be needed as input data for another EPS, but restructured and enhanced with additional information. Although they are independent, an EPS may require the cloud service to fulfill certain preconditions, before being able to serve it. For example a cloud service must be deployed, before it can be monitored.

On the other side the provider of the platform is challenged by the problem of, how to create an easy-to-use and reliable environment, while being able to add, update and replace the EPSs or potentially expand the platform to a new IaaS/PaaS.

Moreover, relying on a set of independent subsystems (the EPSs) to support an elastic cloud service, creates a space for uncertainty and potential failure. If a cloud service developer has no means to verify how exactly each EPS modifies the state of the elastic cloud service, she/he has also no means to improve elasticity of the cloud service or detect a failure. This is especially important, when we consider that the constant change is an inherent feature of the elastic cloud services.

In order to satisfy all stakeholders involved in the development and operation of an elastic cloud service, we propose an integrated framework, which will coordinate the EPSs, but without creating direct dependencies between them and will offer their functionality in a unified, highly transparent way.

1.2 Motivating scenario

To illustrate the advantages of a unified management system of elastic cloud services, we will go through a lifecycle [6] [7] of a sample cloud service, with additional focus on the elasticity. However, we must stress that the goal of this scenario is not to exhaustively support the need for the elastic platform as such, but the need for a unified management system.

As the sample cloud service, let's assume a system that gathers data from many different sensors and stores them in a distributed database. This system is thus composed of servers for pre-reprocessing of the data from sensors and a cluster of nodes with the distributed database typical for BigData [8] problems.

The future multi-cloud platform for elastic cloud services stretches over several IaaS/PaaS cloud providers and besides the services provided by the clouds themselves, offers the EPSs. These are deployment, monitoring and elasticity controller, designed specifically to deal with the challenges that the elasticity of cloud services poses. For now, we do not need to go into specifics, these EPSs will be explained in more details in the Section 3.2.

An elastic cloud service can be constructed by selecting some of the many offered units of service (e.g. different flavors of VM, application servers, load balancers, databases)[5] and providing the custom code in form of a software artifact. Furthermore, a structure of the service and relationships can be defined, describing how should system be orchestrated.

The lifecycle of our elastic cloud service begins with design and development. The platform for elastic cloud services offers tools for integration testing, load generation and elasticity testing. Lets assume, development team decides to use this option, in order to speed up the development by moving the resource-heavy testing to the cloud and to ensure continuous integration during the development. After the distributed database is developed and successfully tested, work continues on the pre-processors. The tests need to be run frequently, but the management system allows selective redeployment of components, so the entire distributed database does not have to be redeployed for every test. When testing the elastic behavior, the developers use the management system's utility for recording and analyzing of elastic changes, which makes it possible to learn, how their elastic system behaves under different scenarios.

After the initial development is successfully finished, the elastic cloud service finally released for operation. Let us consider the case that, over time, first bugs appear on the distributed database and patches need to be applied. Management system enables sequential migration of the database's nodes, so the patch can be applied on the nodes one by one, not disturbing the operation of the cloud service. It might happen that, a few days later, a malfunction appears and the administrator investigates, what might be the cause of the failure. Analyzing the records of systems behavior, he realizes that the problem was, actually, not in the distributed database, but caused by an unrelated phenomenon. Scale out during the peak utilization of few pre-processors took too much time and important data were lost. Administrator might decide to transfer these components to a different cloud with faster allocation of virtual machines.

This scenario shows the various possibilities, how the unified management system simplifies the development and operation of elastic cloud services. Notice that the actors did not have to configure, start or stop each EPS individually or reconfigure the controller, when updating the components.

1.3 Aim of the thesis

The high level aim is to design an integrated management system for elastic cloud services. The management system should fulfill following properties:

- *Development and Operation* - Consideration of the entire lifecycle of an elastic cloud services. Management system should be designed to support and ease development, elasticity testing, operation and maintenance.
- *Coordination of EPSs* - The EPS should be coordinated in a way, that the specific requirements of an EPS, such as activation in a specific stage of lifecycle, certain order and input data, are transparent to the user. Users should only define the EPSs that are to be utilized or insert an optional configuration.
- *Information model* - The management system should be compatible with the current information model, which is being developed for an information service of the future platform for elastic cloud services.
- *Extensibility* - The management system should be extensible with new EPSs. This includes new versions of the current EPSs as well as new EPSs that are not fully defined yet, such as service for testing of elasticity.
- *Traceability* - Actions performed by users as well as their effect on the system and interactions of EPS should be traceable. This task is tightly coupled with provenance of data used and altered within the management system. [9]
- *Structure of service* - An elastic cloud service should be treated not as a solid block, but as composition of topologies and units. This means that the lifecycle as well as the interactions within the management system recognize this structure and allow finer-grained manipulation by elastic cloud services.
- *Offered as service* - The management system should be encapsulated as a service with REST interface.

1.4 Methodological approach

First of all, we review scientific literature that deals with the lifecycle of cloud applications. We discuss, how is the lifecycle of a elastic cloud service different and what additional aspects must be captured in it.

We continue by analyzing the current EPSs: deployment service SALSA [10], elasticity controller rSYBL [5] and monitoring service of elastic cloud services [11]. We investigate their interfaces, what input data they require, in which stage of cloud service's lifecycle they can be applied and how do they modify the state of the elastic cloud service.

Next, we identify the typical actors and the main use cases of the management system. The use cases are based on the research of scientific literature, analysis of existing PaaS solutions and practical experience.

Before developing the lifecycle of an elastic cloud service, we consider the roles that the EPSs play in it and how they influence it. Taking into consideration also the use cases and the roles that may be taken by EPSs. We formulate the lifecycle of an elastic cloud service composed of states and transitions between them.

Building on the the use cases and the lifecycle, we propose an architecture of the managements system. We take into account applicable patterns and styles used in the field of distributed systems.

We implement the integrated management system, using the current implementations of the above mentioned EPSs for demonstration.

Finally, we perform experiments and evaluation of the concept on a non-trivial example of an elastic cloud service.

1.5 Results

Part of the results is the lifecycle of an elastic cloud service and the use cases of the management system from development to operation. It also includes the approach to provide EPSs as supporting OSUs and management of dynamic EPSs as cloud services. Furthermore, the framework of the management system consisting of the the communication model and coordination through the Lifecycle Manager. Additionally, the recorder for capturing of the changing state and structure of elastic cloud services, together with events. Last but not least, the implemented prototype, which demonstrates the management framework integrating the EPSs SALSA, MELA ans rSYBL.

The results contribute to the CELAR project [4] as part of the CoMoT framework [4].

1.6 Structure of the Thesis

The rest of the thesis follows this structure:

- Chapter 2 discusses the current research on lifecycle of cloud applications, some of the current PaaS systems and briefly describes the effort for standardization of cloud application's topology and orchestration. Next, we provide a more in-depth insight of the current EPS and discuss some of the common communication patterns and architectures in distributed system.

- Chapter 3 analyses the information model that the managements system must be compatible with and requirements on the system laid by the EPS. Moreover, we introduce additional functional requirements in form of use cases.

[4]http://www.celarcloud.eu/

- Chapter 4 presents conceptual architecture of the management system and model of lifecycle of an elastic cloud service. It also clarifies the position of EPS in context of the information model.
- Chapter 5 documents the design of individual components of the management system and communication between them.
- Chapter 6 discusses implementation's specifics and details of how the current EPS are integrated with the management system. Furthermore, we evaluate the prototype and fulfillment of requirements in the design of the management system.
- Chapter 7 summarizes the thesis and proposes possible future improvements and research challenges.

CHAPTER 2

State of the Art

We start by looking into possible management tasks related to the elasticity of target systems that the management framework should support. As there are currently no other platforms that would acknowledge a multidimensional elasticity as a first class citizen, we investigate, in the Section 2.1, the management mechanisms related to elasticity in the current PaaS systems. Next, we discuss different approaches to define the lifecycle of a cloud applications in the Section 2.2, what will be the basis for the definition of a lifecycle of elastic cloud services. Because the management system should be designed as one layer of the platform for elastic cloud services, building on top of the EPSs, we describe closer the existing EPSs in the Section 2.3. Last but not least, a necessary requirement for the success of the management system is to integrate the EPSs, while ensuring exchangeability of the EPSs and observability of their interaction. For this reason, in the Section 2.4, we have a look at several architectures, which deal with communication of loosely coupled systems.

2.1 Application Management in Clouds

Amazon's cloud is accessible through a collection of web services called Amazon Web Services (AWS)[12][13]. We will look closer at services: AWS Elastic Beanstalk and AWS OpsWorks, which allow deployment and management of cloud application, utilizing other services of the AWS platform.

The focus of AWS Elastic Beanstalk[14] is on standard web applications implemented in Java, Node.js, PHP, Python, Ruby, .NET or a Docker-packaged application. Each application is managed in a separated environment characterized by, first of all, a type of the platform, second, whether it is a web server or back-end worker environment and finally, single-instance environment versus auto-scaling environment with load-balancing. An application may be configured to use any data-oriented service from AWS to store data. From the management perspective the AWS Elastic Beanstalk handles deployment either directly, via Git repository or an IDE. It manages versions of an application, which can be used for a rollback or in parallel in a separate environment. An application is automatically checked for health and monitored through AWS CloudWatch[1]. Users can define alarms for the monitored metrics, view event log of the environment or the logs of the application. Applications can use the Auto Scaling[2] service that allows manual, scheduled or dynamic scaling based on the metrics from the AWS CloudWatch service.

In contract to AWS Elastic Beanstalk, AWS OpsWorks [15] is not restricted to one architectural pattern of web application, but supports a wide variety of patterns. The top level entity managed in AWS OpsWorks is a stack composed of multiple predefined layers, such as application server, database server or load balancer, or additional custom layers. There can be multiple instances in one layer, configured as static or auto scaling instances through AWS CloudWatch service. A stack can be configured using Chef[3] cookbooks. AWS OpsWorks provides similar management capabilities as AWS Elastic Beanstalk, such as monitoring, versioning, logs.

Google App Engine [16] provides Java, Python, PHP, Go environments and managed VMs. Applications are made up of modules, which are versioned and composed of source code and configuration files. Every module may have multiple instances, that can be scaled manually, on request, or automatically based on resource metrics. The modules may communicate with each other through a queue service, data storage is handled by several types of data oriented services.

In contrast to PaaS providers with proprietary tools and interfaces for development, deployment and management of cloud applications, there is an effort to standardize these processes in order to enable exchangeability of cloud providers and interoperability of tools from different vendors. A milestone of this effort is the OASIS[4] specification TOSCA[17] - Topology and Orchestration Specification for Cloud Applications. It defines language for description of cloud application as topologies composed of nodes or other topologies, where nodes can provide capabilities and define requirements that should be fulfilled in order for nodes to realize a relationship. Components necessary for deployment of an application described by TOSCA, should be packaged in a Cloud Service Archive (CSAR).

OpenTosca [18] is an open source ecosystem for TOSCA-based applications, with an Eclipse plugin[5] for development, runtime environment for deployment and management and a portal for instantiating.

Gigaspaces Cloudify [19] is a cloud application orchestrator following TOSCA specification distributed as either open source or commercial version with additional features. It aims for deployment and management of cloud applications in multiple clouds, e.g. OpenStack, Apache CloudStack and others, through an open plug-in architecture. Furthermore, it is integrated with configuration and management tools, such as Chef,

[1]http://aws.amazon.com/cloudwatch
[2]http://aws.amazon.com/autoscaling
[3]http://www.chef.io
[4]http://www.oasis-open.org
[5]https://projects.eclipse.org/projects/technology.camf

Puppet or Docker, and multiple tools for logging, monitoring and real-time analytics.

The current PaaS platforms support the entire lifecycle and various aspects of cloud applications including monitoring, logging, redeployment with versioning or continuous integration through external tools. This is a challenge also for the future elastic platform, however we must take into account the far higher complexity of the cloud services that can be managed in the platform. The management framework should aspire to cover the above-mentioned aspects either directly or through the EPS.

2.2 Lifecycle of Cloud Applications

There are several publications, which attempt to define the lifecycle of cloud services from different points of view.

[7] discusses the lifecycle from application provider's perspective, while proposing an application-centric lifecycle management framework. This framework would offer combined functionality of IaaS and PaaS through a catalog of services. Furthermore, an application model is provided, in which an application consist of deployment model, components and maintenance model. In the framework, elasticity is defined in the maintenance model through a set of rules and performance metrics. However, the rules can only be defined for the entire application.

A different approach to application's lifecycle is presented in [6]. It compares cloud applications to SOA services and presents cloud governance to be a step forward for SOA governance. The new challenges of cloud service lifecycle, which were not present in SOA lifecycle, are distribution of services across different cloud providers and the elastic capabilities of cloud. This approach only utilizes elasticity of cloud services as a way to prevent SLA violation of monitored QoS constraints.

The principles of SOA lifecycle are also used in [20]. An integrated framework is proposed, which covers all levels of SOA lifecycle and places emphasis on an open source tooling support. This work is focused on monitoring of the system on all levels from infrastructure up to QoS, which provides monitoring data for management of the application. Elasticity is called "dynamic reconfiguration" and is supported by migration of a component to a more appropriate VM at runtime.

We use these works as a starting point, when formulating the lifecycle of an elastic cloud services. Additionally, we focus on the aspect of elasticity and how it effects the lifecycle.

2.3 Platform for Elastic Cloud Services

As already mentioned in the introduction, the proposed platform [4] is composed of services, the EPSs, which support the elastic cloud services in the multi-cloud environment. Currently there exist the following three EPSs:

Deployment Service

The deployment service SALSA implements the framework for multi-level configuration of cloud services in the multi-cloud environment [10]. It supports cloud services structured into service topologies, service units and unit instances, with possible configuration capabilities at each level. Furthermore, the service units may define relationships to each other, which express dependencies between them. The orchestration of software at runtime is managed as deployment stack composed of virtual machine, application container, web container and application. The framework's architecture consists of two main components: a central configuration service and local configurators that are placed on the managed virtual machines. The central configuration services, first of all, orchestrates the service units and topologies, second, configures the cloud infrastructure through an extensible set of cloud drivers. The local configurators are then issued to deploy the rest of the deployment stack, through different third-party configuration tools. SALSA utilizes TOSCA for description of cloud service's structure.

Monitoring Service

The monitoring service MELA is based on the framework for monitoring and analyzing of cross-layered, multi-level elasticity of cloud services[11]. The framework recognizes monitoring data, metrics, in the three dimensions of cost, quality and resource. The low-level metrics are presented in form of snapshot, while a user-defined elastic boundary describes the desired elastic behavior. Concepts of elasticity space and elasticity pathway capture all metrics defined by the user and the relationships between them over time. The motivation for such framework is that other stakeholders of the elastic cloud services, such as users or elastic controllers, need monitoring data about the cloud service on all levels including unit, topology and service, but the standard monitoring tools provide such data only for the virtual infrastructure level. Furthermore, the user-defined requirements on the elastic behavior may be expressed as complex metrics, e.g. serviceCost per client per hour, but the monitoring data capture only low-level metrics, such as cost per virtual machine or throughput. The framework overcomes this by aggregation of metrics through extensible metric composition rules.

Control Service

The control service rSYBL[5] is a service for an automatic fine-grained, multi-level control of cloud service's elasticity, which utilizes the SYBL engine [21]. SYBL is a language for controlling elasticity of cloud services. It recognizes these four types of directives: *monitoring* for specification of, what elasticity metrics should be observed, *constraints* describe the acceptable limits for metrics, *strategies* for defining of actions, which should be taken if certain constraints are violated of fulfilled. The last type of directive allows to determine *prioritiies* of other directives. The Listing 2.1 shows few examples of SYBL directives. For a comprehensive introduction follow the link.[6]

[6]http://www.infosys.tuwien.ac.at/research/viecom/SYBL

```
St1:STRATEGY CASE latency < 400 AND throughput < 900 : scalein
Co1:CONSTRAINT cpuUsage > 35 % AND cpuUsage < 75 %
Co2:CONSTRAINT cost < 100 $/h
Priority(Co2)>Priority(Co1)
```

Listing 2.1: Examples of SYBL directives

rSYBL is based on a composition model designed for definition and control of the elastic behavior in multiple levels, starting from programming level, through service units and topologies up to the whole cloud service. The elastic behavior is a combination of elasticity metrics, requirements, capabilities and relationships. Furthermore, rSYBL considers and aims for resolving of conflicts between requirements on different levels.

2.4 Communication in Distributed Systems

There is a multitude of design patterns and architectures in the field of distributed systems. We neither exhaustively compare, nor enumerate them here. Instead we discuss some of them, that may be relevant for the management system and coordination of EPS.

Service-Oriented Architecture

The reference model from OASIS defines SOA as: *"a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."*[22] The possibly heterogeneous distributed capabilities are encapsulated as services, which have discoverable, well defined, machine-readable interfaces and autonomous applications logic. The development of service-oriented systems should focus on complying with other important principles, such as loose coupling of services, reusability or abstraction of services from the underlying logic [23] [24]. Although in praxis SOA is often associated or even falsely mistaken with SOAP Web services, the architecture itself is independent from a specific language or environment.

Event-Driven Architecture

The central concept is the event, an occurrence of which announces some state, threshold or problem. Events are distributed to all interested parties, which may react on the event by performing some internal logic, or also trigger other events [25]. The involved systems are loosely coupled, the event source does not know about the event targets. Events can be processed in different styles called: simple event processing, stream event processing and complex event processing (the later two sometimes used interchangeably), but in principle, what is important to realize is, whether the events are processed as solitary events or as a series of events, possibly dispersed over longer time interval and from different sources.

A combination of service-oriented and event-driven approaches is called Even-driven SOA [26]. Both the original architectures are compatible, aiming for loose coupling of services. In contrast to standard SOA, where the services are bound through request-response pattern, in Even-driven SOA they communicate asynchronously through events [27].

Space-Based Architecture

Space-Based Architecture is centered around the concept of a space, shared by distributed agents (called processing units) that are coordinated by reading and writing information to this space [28]. An early platform for shared memory between processes [29], referred to as Tuple Space, consists of data organized as tuples, which can be written, read or taken from the space. Several other platforms, such as Java Spaces [30], XVSM [7] or Gigaspaces [8], build on this idea and extend it with notifications, transactions or additional concepts for management of space, such as containers. The focus of the Space-Based Architecture is on loose-coupling and coordination in scalable systems.

In the management framework we use the principles of the Event-driven Architecture for the interaction between the components and EPSs. Ideally, for a seamless integration into the management system, the EPSs should be designed according to recommendations of SOA.

[7]http://www.xvsm.org
[8]http://docs.gigaspaces.com/xap101.html

CHAPTER 3

Requirements and Use Cases

This chapter discusses the requirements on the management system that are placed by the components of the general architecture of elasticity platform, introduced in [4]. First, the management system should be compatible with the information model. The model is analyzed in the Section 3.1. Second, the management system should integrate the current EPSs, which were conceptually described already in the Section 2.3 and will be further analyzed with emphasis on their integration in the Section 3.2. Last but not least, the possible functional requirements expected by users of the management system are illustrated by use cases in the final Section 3.3 of this chapter.

3.1 Information model

The information model captures all the data about a cloud service that are necessary throughout its entire lifecycle. The model is a synthesis of the models introduced in [31] [5] [10]. We explain our syntactically minor, but powerful extension of this model for the needs of the management system in the Section 4.3. The model is presented as a UML Class Diagram, composed of classes, associations and generalizations between the classes, as shown in the Figure 4.8. Additionally, the classes in this diagram are divided into three different spaces, what corresponds to three logical spaces in the model.

- **Service provider** - Provider space contains building blocks, that can be used by a cloud service developer to build and support an elastic cloud service. The atomic building block is the *OfferedServiceUnit*, which offers several*Resources*, *Qualities* and *CostFunctions*.

- **Development** - A cloud service developer specifies the structure of an elastic cloud service by defining the *ServiceEntities* and *Relationships* between them.

- **Runtime** - When an elastic cloud service is deployed in the cloud, every instantiated service unit is represented by *UnitInstance* with realized *RelationshipInstances* between them.

Service Provider Space

An **OfferedServiceUnit** OSU is statically described by a *Provider*, set of *PrimitiveOperations* and a *Type*. Every OSU has also a set of *Resources*, *Qualities* and *CostFunctions*, from which the cloud developer can choose, when selecting an OSU. By configuring a concrete instance of a OSU a cloud service developer in fact create a new **OsuInstance**. The Service Provider Space is mostly domain of the Information Service. For the purpose of the management system is most important the concept of OSU as an atomic unit provided in the platform and its configurability and following instntiation as **OsuInstance**.

Developer Space

The **CloudService** is an umbrella entity for the entire information that is inserted into the management system by a cloud developer during the design of the elastic cloud service. It is composed of *ServiceTopologies*, that are composed of other *ServiceTopologies* and *ServiceUnits*. All these classes are specialized type of *ServiceEntity*. The *SyblDirective* contains single directive in the SYBL language, described by *directiveId* unique in the scope of a **CloudService**, *value* and *type* corresponding to either Strategy, Constraint, Monitoring or Priority.

The relationships are defined in a more flexible manner, but in fact the relationship **HostOn** makes sense, only when the target as well as the source are *ServiceUnits*. It expresses not only the dependency of one *ServiceUnit* from another, but more of a composition. For example a Web application Archive (WAR) file can be hosted on an application server that in turn, can be hosted on a virtual machine. The **ConnectTo** relationship denotes the direction of dependency between *ServiceEntites*. At the deployment time, this determines the order of deployment together with the *HostOn* relationship.

On purpose, we mention the class *Template* as the last, since it requires understanding of the rest of the Development Space. **CloudService** on one hand represent a service that can be instantiated, used and managed, but on the other hand, also represents a description that could be used as a template for creation for other instances or for composition of services into composed services. For this purpose the *Template* class. If a **CloudService** is referenced by a *Template* it means it is in fact a template, not an instanciable cloud service.

Runtime Space

When an elastic cloud service is successfully deployed in a cloud, its *ServiceUnits* correspond to one or multiple **UnitInstances**.

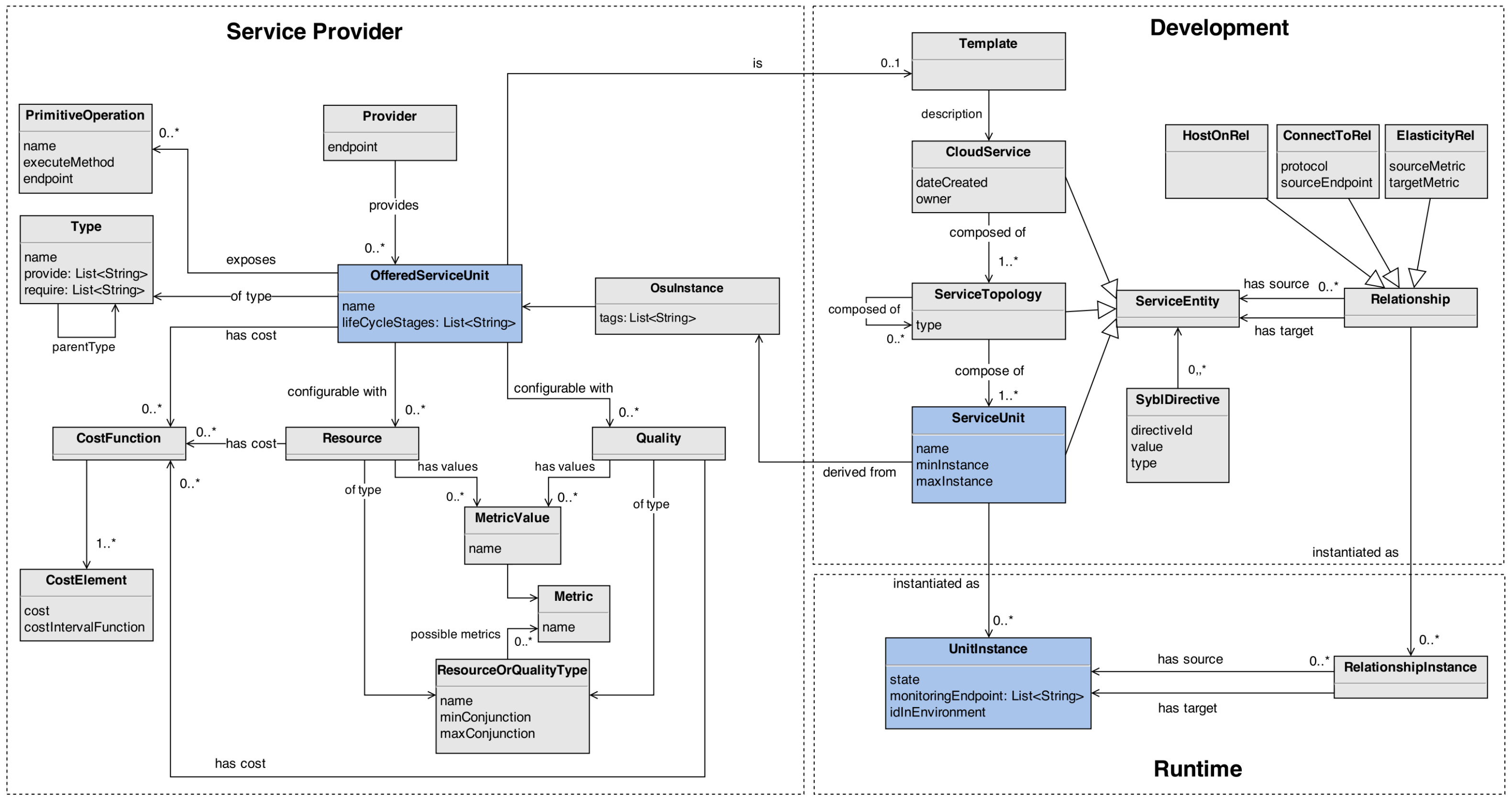


Figure 3.1: UML Class Diagram - Information model

As already mentioned, the OSU / *OsuInstance* can be seen as an atomic building block used in an elastic cloud services. But its role in the information model is more complex, being determined mostly by the following 2 important relations:

- **ServiceUnit** *derived from* **OsuInstance** - Each *ServiceUnit* can be associated with this relation only to one *OsuInstance*. It means that the information from *OsuInstance*, will be used at deployment time for creation of *UnitInstances*. Such *OsuInstance* can represent, e.g. a type of virtual machine or application server provided by an underling cloud. A custom artifact provided by a cloud service developer, such as WAR file, is also managed as an *OsuInstance*. However, an OSU used in this relation, which the *OsuInstance* originates from, can also be, in fact, a *Template* of a cloud service. This enables composition of cloud services in the Developer space.

- **OSU** *is* **Template** - This relation takes part in the composition of cloud services mentioned in the previous point. Let's imagine a following scenario: A cloud service developer designs a *CloudService* A. That is, he defines all the *ServiceEntities*, *Relationships* and OSUs to derive the *ServiceUnits* from at deployment time. She/he further defines that the *CloudService* A should be offered as an OSU in form of **Template**. After this, when creating a *CloudService* B in the future, she/he can use that OSU to to effectively set the *CloudService* A to be a *ServiceUnit* in the *CloudService* B.

3.2 Elastic Platform Services

In this section we concentrate on the EPSs and describe the main requirements for integration into the management framework. For more details, we analyze also the current prototypes, mainly, what are the possible operations to interact with these prototypes and the models of the input data. However, the goal is not list complete specification, but highlight the important aspects.

We point out following significant operations offered by **SALSA**[1]: Deployment operation accepts TOSCA as the input. SALSA utilizes the extensibility of TOSCA and defines properties for specification of virtual machines and custom actions. Deployed or undeployed can be also service topologies, units or instances selectively. It is possible to retrieve the current state of the deployment and the information about the cloud environment, such as an IP address or image identifier. This information is necessary for all other EPS, as they require it to be included in their input data.

MELA [2] is in fact composed of 4 separate services. The *Data Service* gathers the monitoring data, while the other three analytic services are extensions dependent on the Data Service.

- *Data Service* is activated for a certain cloud service by submitting the structure of the cloud service and including the deployment information. Furthermore a

[1]http://tuwiendsg.github.io/SALSA/
[2]http://tuwiendsg.github.io/MELA/

separate operations allow setting or change of metric composition rules, as well as requirements anytime during the execution of the monitoring. The monitoring data are accessible through a set of operations that mediate the access to the underlying database.

- *Elasticity Relationships Analysis* and *Space and Pathway Analysis* do not require a continuous subscription in order to provide data about a specific cloud service. But they are dependent on the data from the *Data Service*.

- *Complex Cost Evaluation Service* additionally requires the service structure enriched with used cloud offered services and allows a pricing scheme to be inserted.

rSYBL [3] requires for control of a cloud service the structure of the service enriched with the elasticity requirements in form of SYBL directives and elasticity relationships. Furthermore, is required the deployment information. Optionally metric composition rules and elasticity capabilities effects can be defined. The autonomous control itself can be started or stopped by separate calls. During the control, the elasticity requirements, metric composition rules and elasticity capabilities effects can be updated.

Each EPS offers a REST API. Each EPS also honors the concepts of cloud service, topology and service unit, but there is a difference in understanding of the service unit concept. On one hand, SALSA treats every element of the deployment stack as a service unit, on the other hand, rSYBL and MELA do not distinguish between types of service units, but handle them in a more abstract way. For example a virtual machine with an application server and a web application would be seen as three service units by SALSA, but as only single service unit by rSYBL or MELA.

To sum up, we stress the indirect influence between the EPS. First, the monitoring and control are strongly conditioned by the deployment service successfully deploying the cloud service. Second, if the control service triggers an elastic action, it may change the deployment structure of the cloud service, thus making the monitoring service outdated.

3.3 Use cases

We will start with complex use cases, which name the functional requirements on the management system and the actors that are associated with the use cases. In the last part of this section, we will break down the main use cases into simple ones and describe them in details.

Actors

In the use cases, we recognize following actors:

[3] https://github.com/tuwiendsg/rSYBL/wiki/User-Guide

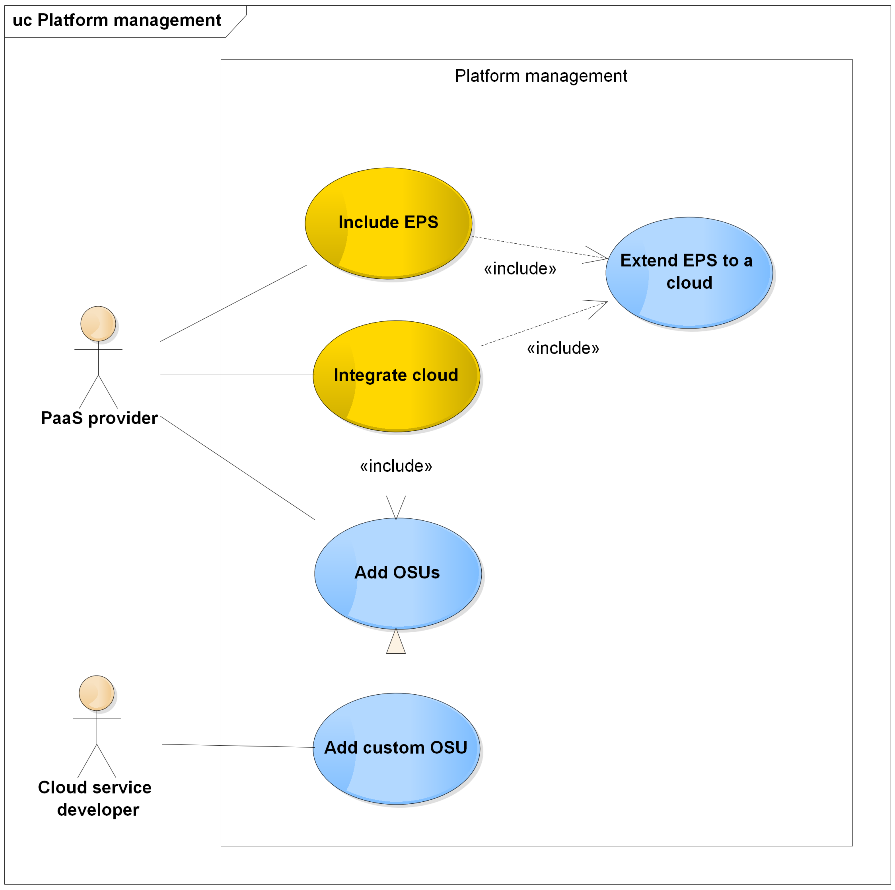


Figure 3.2: UML Use Case Diagram - Platform management

- *Cloud service developer* - responsible for design of cloud services starting from selection of offered service units and implementation of custom service units up to definition of structure and elasticity of the cloud service.

- *Cloud service administrator* - in contrast to the developer, administrator is not required to understand details about functionality of service units, but only their roles in the service as a whole. They are concerned about proper execution and elastic behavior of a cloud service over a long period of time. They administer updates of the cloud service or selected service units, as these might be provided by a third-party developers without the access to the elastic platform.

- *PaaS provider* - operates over multi-cloud environment and publishes infrastructure and services of individual clouds through offered service units in a repository. She/He provides additional tools for the developer and administrator in form of elastic platform services, such as deployment, monitoring and control.

First two use cases evolve around the PaaS provider and the need to set-up and maintain the elastic platform itself.

3.3.1 Integrate a cloud into the PaaS's multi-cloud environment

Different clouds need to be integrated as part of the set-up process of the elastic platform. Furthermore, to be able to keep up with the growing number of cloud providers, it is desired to add new clouds at any point during the lifetime of the platform. This means that a set of new offered service units, which will represent the cloud's infrastructure, should be added to the repository. Another sub-task is to deploy components of the EPS to the new cloud, so that they can perform their supporting activities also on service units deployed in this new cloud.

3.3.2 Include an elastic platform service into the PaaS

Another use case for the PaaS provider is to add an EPS. This can be due to, e.g. a new version of an elasticity controller or monitoring tool. Similarly, as in the case of a new cloud, EPSs need to be included at the set-up time of the platform, as well as during run-time. EPSs may be composed of a central component and local components, which must be deployed to all clouds of the multi-cloud environment.

Following use cases capture, how cloud service developer utilizes the provided tools and offered service units to build an elastic cloud service on top of them:

3.3.3 Develop an elastic cloud service

In a most simple scenario a cloud service can be developed by selecting OSUs, implementing custom service units and defining the relationship between them. However, developers need more advanced tools to support and speed up agile development of cloud services. When developing a complex system, which a cloud service certainly is, it is essential to be able to redeploy only certain components of the system, meaning service units or topologies. This enables incremental and iterative implementation of custom service units and convenient functional testing.

3.3.4 Fine-tune elasticity

After all the custom service units are developed and structure of the service defined, it is necessary to configure elasticity of the service. The developer may execute a performance test with different elastic configurations, to verify the elastic behavior of the service and to choose an optimal configuration. She/He may also test multiple OSUs, e.g. VMs from multiple clouds, to see which cloud is the most suited for which type of service unit. A recommendation mechanism to represent similarities between the OSUs, would be a helpful tool to ease the orientation in the set of the provided OSUs.

The longest lifecycle phase of a cloud service is the operation and maintenance, which is supervised by a cloud service administrator. There are several use cases, which should be highly automated, since these tasks are performed repeatedly over a long period of time and by a personal without an in-depth knowledge of the cloud service.

3.3.5 Update/migrate service units

Whether it is due to a release of a new feature, bug fix or transfer of a service unit to a different cloud, offered and custom service units will have to be, sooner or later, updated or rearranged during the operation. A cloud service may be composed of tens or hundreds of service units and to redeploy the entire service only to update one of them would waste resources or cause interruption of the service. That is why service units should be redeployed selectively. There may be several strategies, how to update service units, e.g. all instances at once or one by one, so that the performance of the cloud service would not be hindered. Either way, it is important that the data contained in the service units is not lost during the update and stays consistent. For this reason, an update of a service unit can also be considered as a migration from one instance of the service unit to another instance.

3.3.6 Elasticity analysis and optimization

The lifetime of a cloud service may stretch over several years. In this period, service units are updated to a new version several times, moved from one cloud to another, hosted on different OSUs, supported by different EPSs and adhere different elastic behaviors. Without any tool support, it would be extremely difficult for a service administrator to quickly and correctly analyze, whether there are any radical anomalies in the elastic behavior of the cloud service and what have caused them. The administrator should be able to select a metric, e.g. response time and receive all events that correlate witch the deviations of this metric. Subsequently the optimization is performed either through an update/migration of service units or change in the elastic configuration.

3.3.7 Failure diagnostic and recovery

If a service unit or the entire cloud service crashes during operation, it is necessary to diagnose the cause of the failure, which service units are effected and resume the operation. The failure may occur during or after an update, in which case it might be desired to return to the previous version of the cloud service. It might also be caused by a temporary malfunction of a single cloud provider, malfunction of an elastic platform service or simply by a bug in a third's party component. An administrator could choose automatic recovery policies that are triggered, when the management system recognizes a certain type of failure. A semi-automatic solution could notify an administrator and inform him of a probable cause and suggest a set of actions.

3.3.8 Detailed description

In Figure 3.2 are depicted use cases 1 and 2, which relate to the management of the elastic platform, decomposed into simpler use cases. We will explain each elementary use case one by one:

- *Include EPS* - A new EPS is added to the platform during set-up of the platform or at run-time. This is composed of deployment of the EPS's main component, update of the managers REST API and GUI followed by extension of EPS to all clouds of the multi-cloud environment.
- *Extend EPS to a cloud* - If there is a local part of an EPS, it is deployed to a cloud.
- *Integrate cloud* - Integration of a new cloud to the multi-cloud environment includes, e.g. development of connector for a deployment EPS and publishing of the cloud's infrastructure and services as uniform OSUs.
- *Add OSUs* - Publish a VM or software such as application server, load balancer as an offered service unit. This may be done while integrating a new cloud, but also at any other time, when there is a new software to add.
- *Add custom OSU* - When developer wants to deploy a custom software, e.g. packaged as a war file, she/he must define it as a type of OSU first.

Figure 3.3 depicts the rest of the use cases, 3 to 7, which deal with the management of the cloud services.

- *Develop elastic CS*, *Fine-tune elasticity*, *Elasticity analysis*, *Update/migrate SU or topology* and *Failure diagnostic and recovery* are the complex use cases as described in the Section 3.3
- *Select OSUs* - Choose from the provided OSUs and optionally define a custom OSU.
- *Define topologies and relationships* - Relationships between service units, such as host-on or connect-to.
- *Deploy CS* - A cloud service can be deployed only after the OSUs were selected and topologies and relationships defined. For each deployment of a cloud service a set of supporting EPS can be defined.
- *Select EPSs* - Such as monitoring or control.
- *Create private EPS instance* - Instead of a shared EPS instance, a new instance EPS instance is created, which the developer may use for support of one or more of her/his cloud services.
- *Redeploy SU or topology* - Parts of a cloud service are redeployed without considering the state of the old instances. However, the redeployment respects relationships.

- *Define elasticity* - Set SYBL directives for service, topologies and units. Either as part of service development, fine-tuning or after an analysis of the elasticity.
- *Test elastic configurations* - Run multiple test with different SYBL directives.
- *Compare and test OSUs in multi-cloud* - Consider recommended alternatives to the selected OSUs. E.g. VM types in different clouds and test them.
- *Exchange OSU* - E.g. if the creation of a new VM after a scale-out takes in last months too long, it may be better to change the cloud and thus the OSU that represent the VMs.
- *Update structure or relationship* - May be part of the use case *Update/migrate SU or topology*.
- *Move a SU to different cloud* - This can be requested explicitly, e.g. change the host of a service unit or implicitly, when a VM is exchanged by a VM in a different cloud, all the service unit hosted on it must be migrated to the new cloud as well.
- *Automatic recovery* and *Semi-automatic failure analysis*- Possible two options, how the *Failure diagnostic and recovery* may evolve. In contrast to the other use cases it is not triggered by the actor, but by the system itself, when a failure occurs.

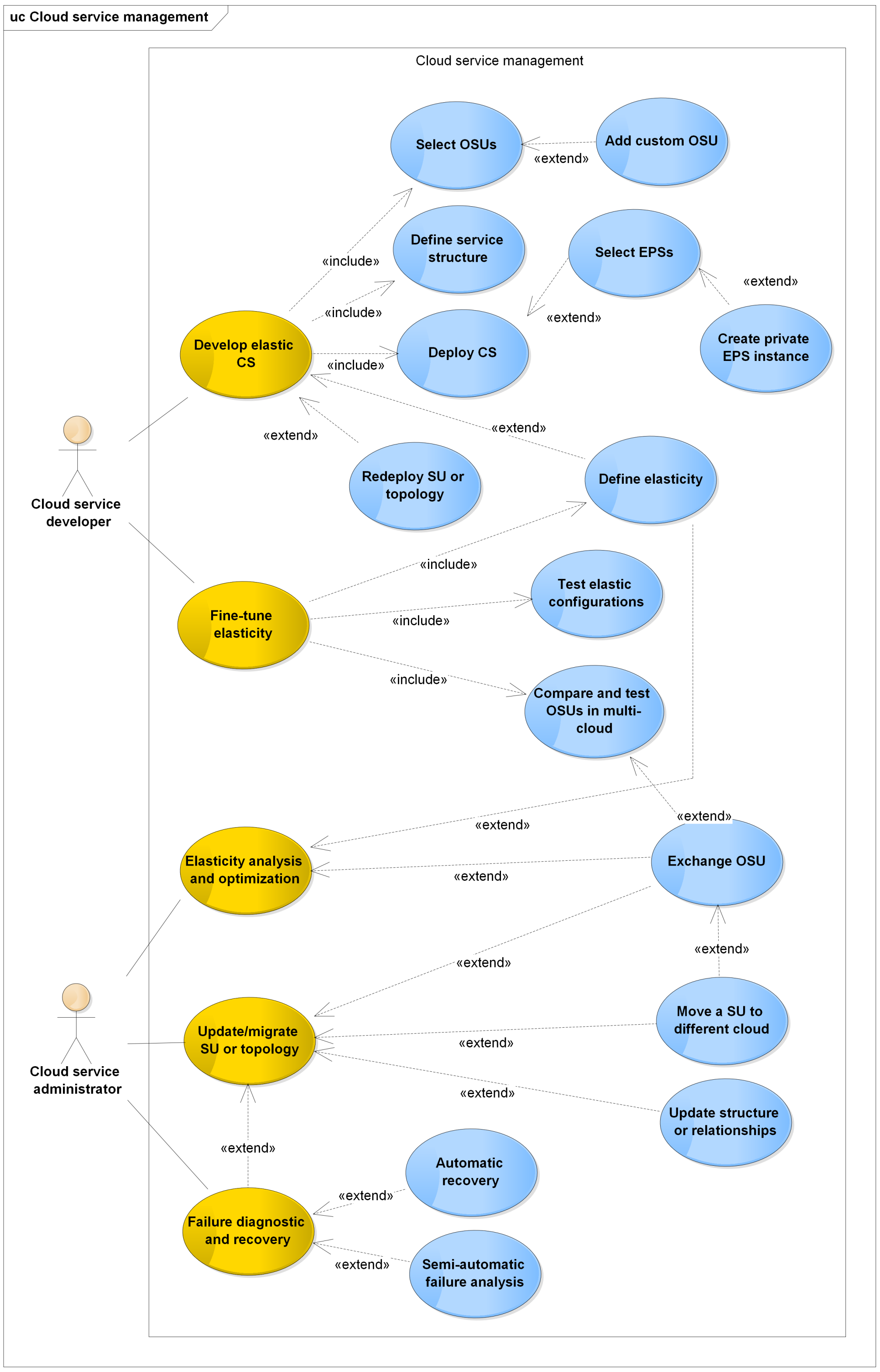


Figure 3.3: UML Use Case Diagram - Cloud service management

CHAPTER 4

Framework and Concepts

4.1 Architecture

Based on the properties stated in the Section 1.3, use cases identified in the previous section and requirements laid by the EPSs, we propose the following architecture of the management system and a communication model for coordination and information sharing among all parties. In this chapter we describe the high-level view of the architecture and in the Chapter 5 we will design the management system in more detail, while complying to the principles of this architecture.

The core concept of the management system is the lifecycle of elastic cloud service (see the Section 4.2), which is observable by all participants of the cloud service's life. A participant can be any software component that has some interest in the lifecycle of a cloud service. The lifecycle can be thought of as a finite-state machine [1] composed of states and transitions between these states. A transition from a current state to a next state occurs, when an appropriate event is triggered by a participant. As depicted in the Figure 4.1, every participant can trigger an event and every participant, if subscribed, is notified about occurred events and the resulting state of the elastic cloud service. Every participant accesses the information model to gather an additional information that he/she needs about a cloud service. We build on the principles of the Event-driven architecture as described in Section 2.4, that is why the participants are not passive or react only when directly called, but are aware of the lifecycle through the events and know, in which state they should execute what functionality. Furthermore, participant can trigger also custom events, which are not defined in the lifecycle. These are also delivered to other participants, but have no direct effect on the lifecycle itself.

We will continue by briefly introducing all the components of this architecture and describe the communication between them.

[1]http://en.wikipedia.org/wiki/Finite-state_machine

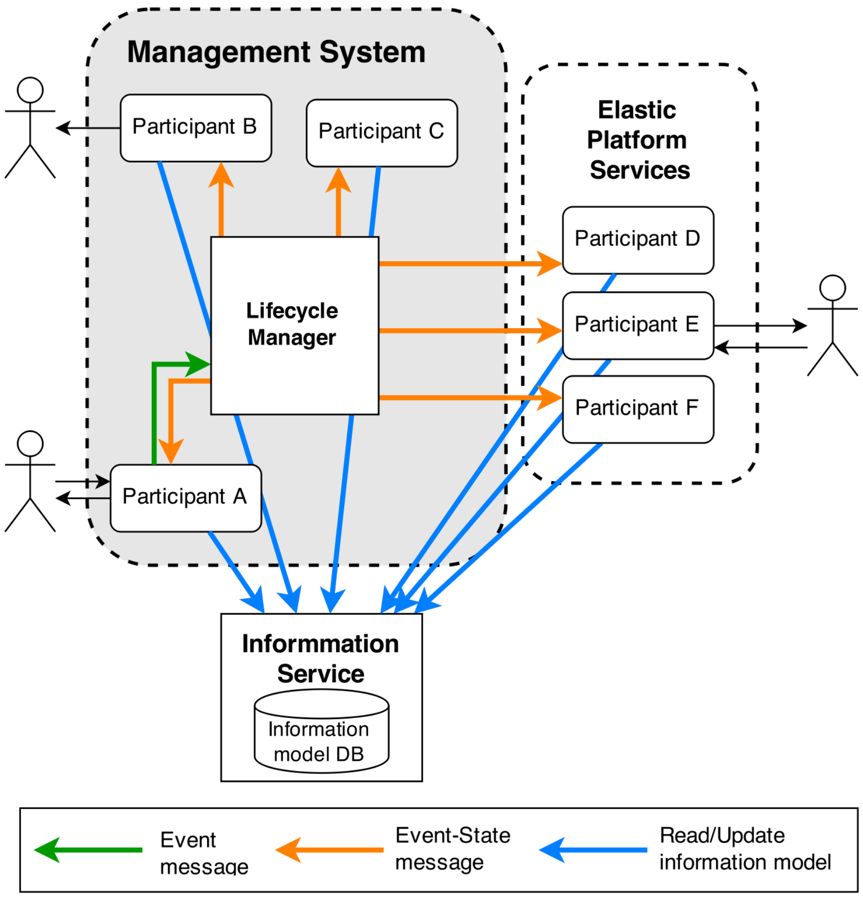


Figure 4.1: Architecture of the management system. When a participant sends an Event message, all interested participants are notified in form of an Event-State message. Every participant can access the Information service.

4.1.1 Lifecycle Manager

The Lifecycle Manager is the central component that maintains the lifecycle of every cloud service, service topology, service unit and instance of a service unit.

Lifecycle Manager distinguishes two types of events: **Lifecycle events** that may (or may not) trigger a transition from one state to another. They are evaluated on the lifecycle of the target and accepts them only in states, where these events are permitted. The second type are so called **Custom events**. These can be defined by any participant and are distributed by the Lifecycle manager without restrictions.

Events can be targeted to the whole cloud service, a topology, a service unit or a service unit instance. However, a Lifecycle event is always evaluated on a service unit instance, or multiple service unit instances included in the target service unit, topology or service. The resulting states of all the influenced entities is determined by aggregation of states of all included service unit instances.

For example when an event, e.g. undeploy, is triggered on a service unit, the lifecycle manager evaluates this event on each included service unit instance separately. Using an aggregation strategy it decides the state of the targeted service unit and propagates the change also to the upper levels, meaning toplogies and the whole cloud service, where the state is also determined by the aggregation strategy.

The Lifecycle Manager will be described in more details, including the subscriptions, event structure and event targets in the Section 5.3;

4.1.2 Information Service

Although, it is not an internal part of the management system, it is a key service that the management system relies on. The information service stores the information model and offers an interface to search and read the stored data. It is used as a central repository for all information related to the cloud service throughout the lifecycle. The concept of the information model is described in the Section 3.1, but we do not deal with specifics of the information service itself. We simply assume a set of operation for storing and querying of data.

4.1.3 Participants and Roles

Any software component interested in the lifecycle of a cloud service. These can internal components of the management system or EPSs. Participants send events only through the Lifecycle manager not directly to each other. If a participant is interested in events about particular cloud service, state or state of certain part of a cloud service, it can subscribe to them and will be notified in future, when the event occures.

Each of the participants is assigned one or multiple roles. These roles can be divided into three categories, which are supersets and subsets of each other, as shown in the Figure 4.2. The most exclusive roles, Decision Makers, are also Influencers, while all Influencers are also Observers. These three categories express the measure of impact, that a participant has on the lifecycle of an elastic cloud service and describe the control flow in the management system.

Decision Makers The only roles that are able to produce a decision, meaning they determine the development of the lifecycle.

- The Manager represents any decision that originates outside of the elastic cloud service. It determines if and how long will a *CloudService* exist, whether only passively or also actively in the multi-cloud environment and delegates control to other participants. The Manager may also influence the elastic cloud service by issuing an update of the elastic behavior.
- The Elastic Controller, on the other hand, acts on behalf of the elastic behavior defined within the elastic cloud service. It may influence the elastic cloud service by creating, removing or re-configuring the *UnitInstances* in cloud.

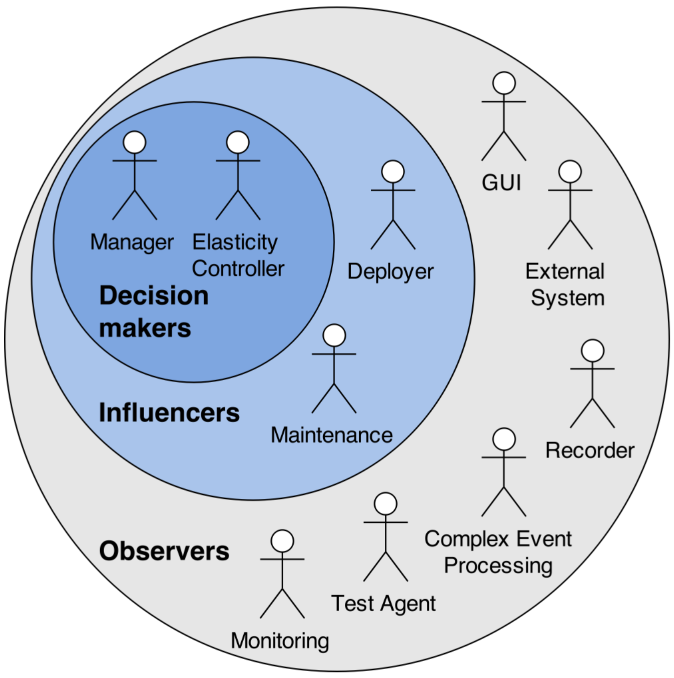


Figure 4.2: Roles taken by the participants of a Cloud Service's lifecycle

Influencers As the name suggests they, act upon a *CloudService* by changing its state, structure or behavior. That is why, their access to the *CloudService* must be coordinated and ensured that their actions do not overlap.

- Deployer creates *CloudService* and its parts in the multi-cloud environment and tears it down. But the control to perform this is, given to the Deployer by a Decision Maker.
- Maintenance may influence the structure and application logic of an active *CloudService*. This can include, e.g. update to a new version of software, without loosing data or re-configuring a custom software. Similarly to the Deployer, it executes the maintenance, but must be instructed to do so by a Decision Maker.

Observers In contrast to Influencers, Observers are only interested in reading of the state and changes, which happen in a *CloudService*. Their behavior may be determined by the observations, but they never directly influence the cloud service. In the Figure 4.2 we show a few examples, but there can be many additional roles that do not need to be strictly defined, as they do not influence the lifecycle itself. A Test Agent can observe the lifecycle to verify if certain preconditions or assertions of a test case are fulfilled. There could be a separate component for Complex Event Processing analyzing events of multiple services searching for predefined patterns or any External System that wishes to integrate with the elastic platform.

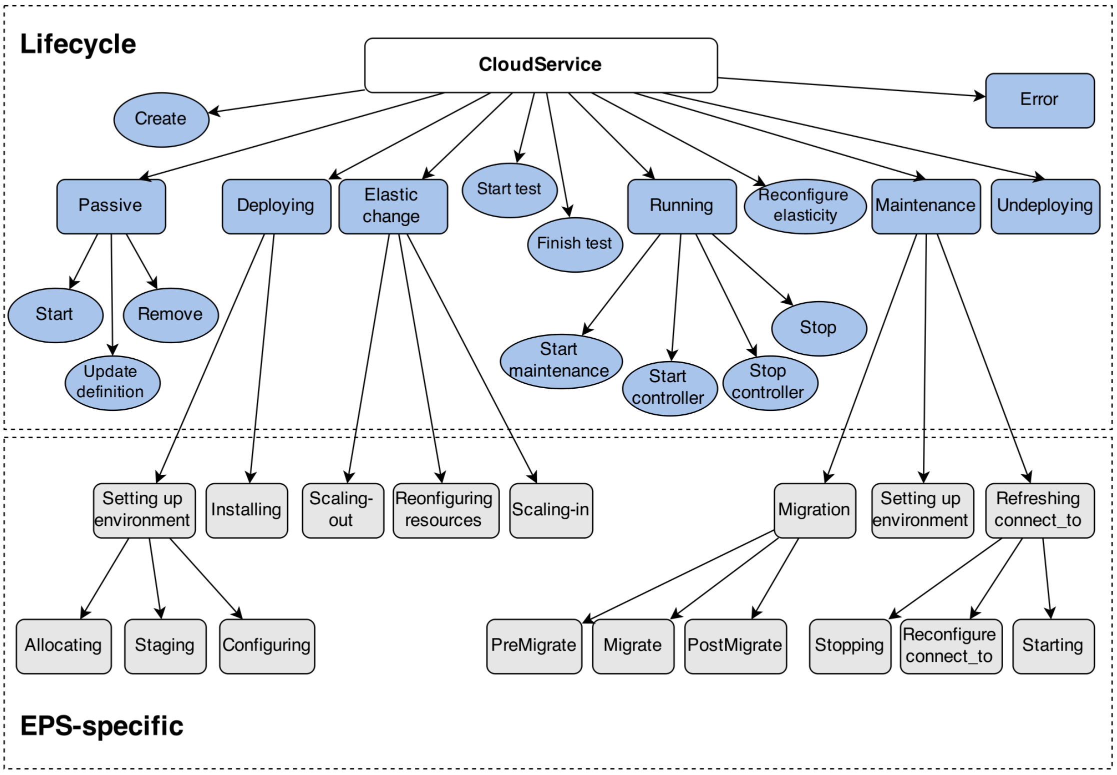


Figure 4.3: Lifecycle of an Elastic Cloud Service.

To sum up, it is important not to forget that all participants are also Observers and Decision Makers are also Influencers. These three categories can be seen as three aspects or abilities of a participant. Any participant may read the lifecycle, participants may change the state of the elastic cloud service only from the role of an Influencer and finally, the flow of control between Influencers can be redirected only by a participant in a role of Decision Maker.

4.2 Lifecycle

We have studied the different approaches to define a lifecycle of a cloud service in the Section 2.2. Our approach is shaped by the concept of different roles of participants introduced in the previous section. We will explain the resulting lifecycle depicted in the Figure 4.3. The boxes stand for possible states of the entity, while the ovals represent solitary events. The start and the end of a state is always determined by corresponding border events (those are not shown on this diagram). The diagram is divided into 2 parts:

Lifecycle The states are determined by the concept of Roles, with each Influencer operating on the cloud service in its separate state and are a backbone of the

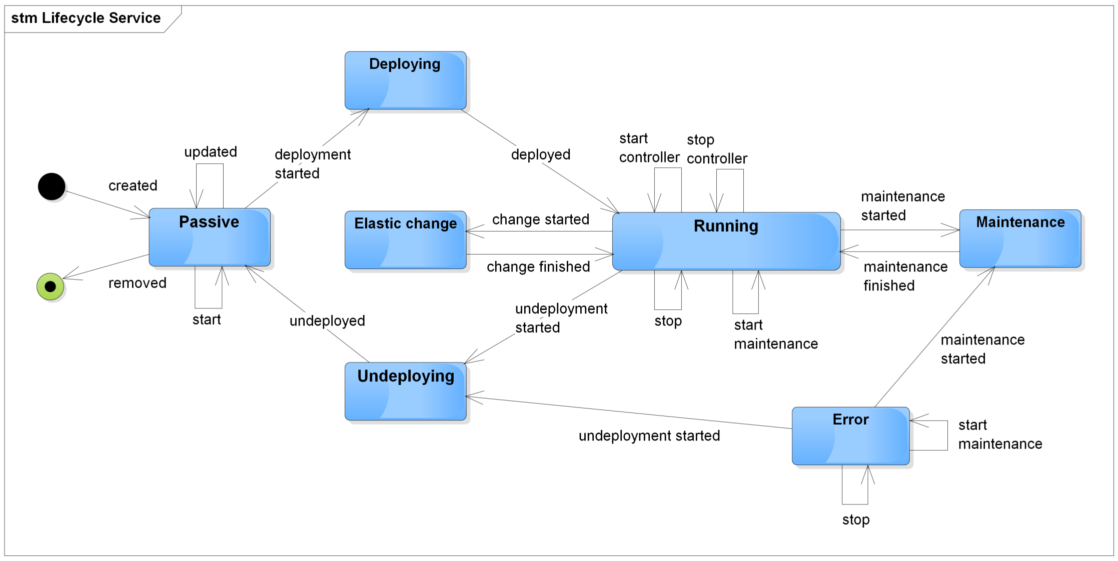


Figure 4.4: UML State Machine Diagram - Lifecycle of a Cloud Service

management system. In the states Deploying, Undeploying, Updating and Elastic Change, the control is in hands of corresponding Influencers. The states Passive, Running and Error give control to the Manager role.

Although a cloud service is only in one state at a time, it is composed of topologies, units and instances of these units, all of which have a state of their own. This structure can be imagined as a tree, where the root is the *CloudService* and the leafs are the *UnitInstances*. State of every node in this tree is determined by the states of its children nodes. So, even though, we speak about a lifecycle of a cloud service, it is more of an aggregate of all the lifecycles of included *UnitInstances*. We will discuss this tree structure in more details in the Section 5.3. Notice that the states marked by blue color are identical to the states of the state machine diagram in the Figure 4.4.

EPS-specific It is an extension for the lifecycle of a cloud service. The states under the control of Influencers (other than Manager) can be freely extended with sub-states and events by each participant in the role of the respective Influencer. The management system supports this part in form of custom events. States in this section are a refinement of the more generic states above them. Deployment is divided by SALSA into setup of environment and installation of custom artifacts. rSYBL distinguishes between scale-out, scale-in and reconfiguration of resources and the refined state Maintenance illustrates states used by the proposed Cloud Service Updater component (see the Section 5.8).

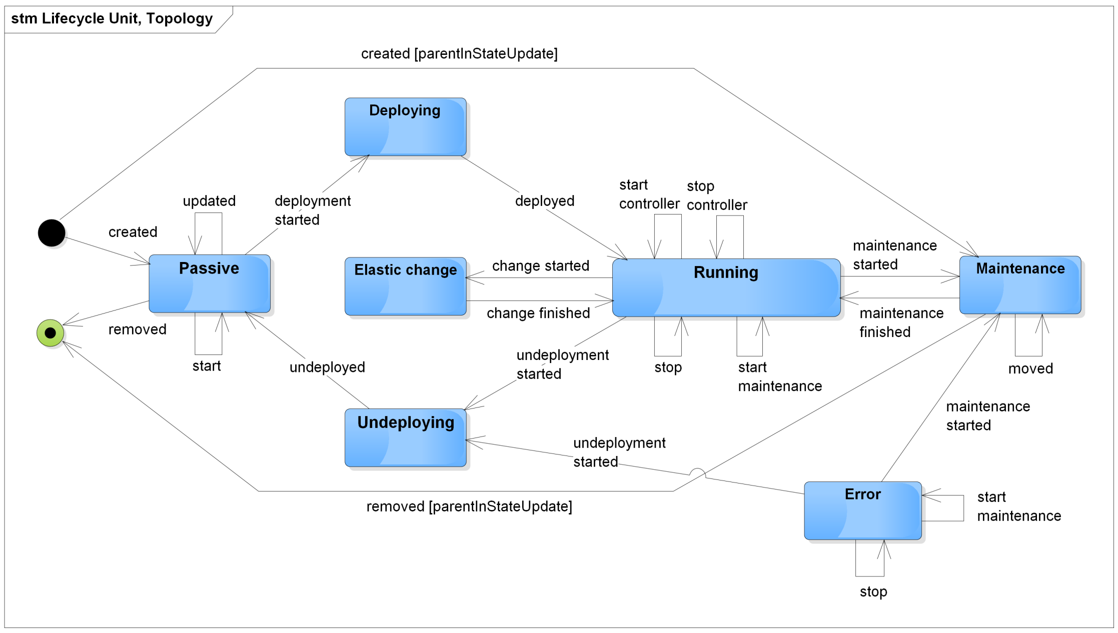


Figure 4.5: UML State Machine Diagram - Lifecycle of a Service Unit and Topology

4.2.1 States and Transitions

Now, we will look closer at the blue-colored states of a cloud service from the Figure 4.3 and all the possible Lifecycle events that can occur in these states. The states also with the transitions are depicted separately in the Figure 4.4 the entire *CloudService*, the Figure 4.5 for service units and topologies, finally the Figure 4.6 for instances of service units. All of these finite-state machine contain similar state and transitions. We will start by describing the lifecycle of the *CloudService* and explain the differences to the other two lifecycle at the end.

- **Passive** - When a *CloudService* is created in the information model and the management system, but not started yet. It can also be removed from the system or updated in this state, because it is not deployed in the cloud yet. The role that has control over the state Passive is the Manager. By triggering the event *started* it, offers the control to a Deployer. Deployer sends event *deployment started*, thus accepting the control and moving the lifecycle to the state Deploying.

- **Deploying** - Deployer deploys the cloud service and during this process emits events about the progress. When finished, by sending event *deployed*, it releases control back to the Manager and state changes to Running.

- **Running** - This is a default state during operation of the *CloudService*. Other events such as *stop* and *update* uses the Manager to offer control to Updater or Deployer for undeployment.

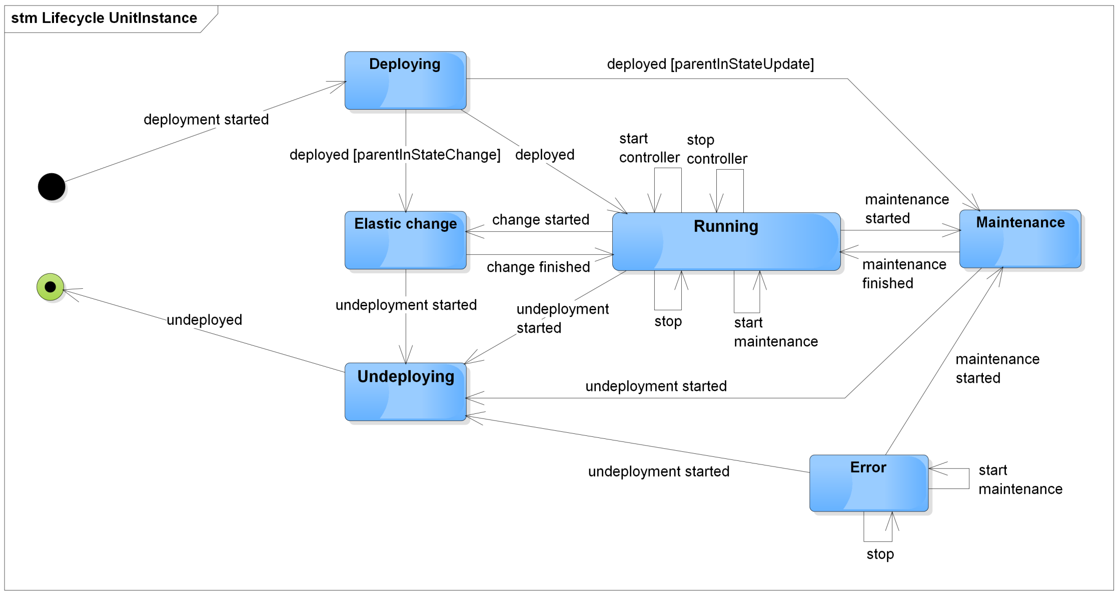


Figure 4.6: UML State Machine Diagram - Lifecycle of a Unit Instance

- **Elastic change** - Elasticity Controller uses this state to practice the elasticity of the service. Because it is a Decision Maker, same as the Manager, the Controller can take the control any time, thus changing the current state to Elastic change. It returns control to Manager and returns the state to Running by the event *change finished*.
- **Maintenance** - Similarly to the Elastic change, this state discourages from other actions then the execution of the maintenance procedure. However, if the Maintenance and Elasticity Controller are able to target their events only on a particular unit or topology, the structure of the cloud service and mechanism for evaluation of lifecycle state, would allow simultaneous maintenance and elastic change of, e.g. two different service units.
- **Undeploying** - Similarly to Deploying, Manager offers control to Deployer, who undeploys the service and returns control back in the state Passive.
- **Error** - There exists a transition to the Error state from all states including Error state, but for the sake of readability of the diagram, we have not explicitly captured these transitions. To resolve an Error, service parts can be either deployed or fixed in Maintenance.

The lifecycle of a service unit or topology in the Figure 4.5 differs from the lifecycle of a service only in the additional transitions to/from the state Maintenance. This is caused by the fact that a unit/topology can be created/moved/removed from the service as a result of the maintenance.

The lifecycle of a UnitInstance in the Figure 4.5 lacks the state Passive, because it is the inherent property of the state Passive that there are no UnitInstance deployed in the multi-cloud. They are created directly to the state Deploying and depending on the state of their parent service unit, continue to the state Elastic Change, Running or Maintenance. Similarly with undeployment.

4.2.2 Lifecycle Events

Previous sections already mentioned several Lifecycle Events that cause transitions in the lifecycle. We continue by systematically listing all the Lifecycle Events. They can be divided into three categories: First, Events that cause transition from one state to another, thus transferring the control from one Influencer role to another. Additionally this category includes also event that notify about change in cloud service's structure. See the Table 4.1 for details.

Name	**Description**
`CREATED`	Service created in the management system and information model.
`DEPLOYMENT_STARTED`	A participant took a role of the Deployer and assumed control or created a new unit instance.
`DEPLOYED`	Deployer released control, possibly changed Runtime information of a unit instance.
`ELASTIC_CHANGE_STARTED`	A participant took a role of the Elasticity controller and assumed control.
`ELASTIC_CHANGE_FINISHED`	Elasticity controller released control, possibly changed information in the Runtime space.
`MAINTENANCE_STARTED`	A participant took a role of the Maintenance and assumed control.
`MAINTENANCE_FINISHED`	Maintenance released control, possibly changed information in the Developer or Runtime space.
`UNDEPLOYMENT_STARTED`	A participant took a role of the Deployer and assumed control.
`UNDEPLOYED`	Deployer released control or a unit instance was undeployed, thus removed.
`REMOVED`	Terminated management of the service, topology or unit, removed from the information model.
`ERROR`	An error occurred.
`KILL`	Results in an ungraceful shutdown.

MOVED	It is important for the Lifecycle manager itself, as it marks change of services structure.
UPDATED	Notifies about changed description of a service in the information model. Similarly to the event MOVED, this is crucial for the Lifecycle manager.

Table 4.1: Lifecycle Events that trigger transitions between states or announce change in the service's structure.

Second category, Events, which all belong to the role of Manager, trigger self-transition within one state and serve for announcement of a decision. These are summarized in the Table 4.2.

Name	**Description**
START	Start (deployment) of the service requested.
STOP	Stopping (undeployment) of the service requested.
START_CONTROLLER	Start Elasticity controller.
STOP_CONTROLLER	Stop Elasticity controller.
START_MAINTENANCE	Give control to Maintenance.

Table 4.2: Lifecycle Events that initiate change of control.

The last, third category shown in the Table 4.1, notifies participants about a significant event. These events are not restricted to a specific state. Although they could theoretically be defined only as Custom events, it is important to define them in the core of the framework, so that the future participants can rely on a basic skeleton and appropriately configure their adapters.

Name	**Description**
RECONFIGURE_ELASTICITY	Notifies about a change in the elastic behavior of the cloud service.
START_TEST	Elasticity test start. May initiate, e.g. a testing sequence of a Test Agent or higher accuracy of a Monitoring.
FINISH_TEST	Elasticity test end.

Table 4.3: Lifecycle Events that mark other notable events.

4.3 Elastic Platform Service as an Offered Service Unit

Based on the use cases we distinguish between static and dynamic EPSs. A static EPS exist only in one instance that is not manageable by the management system. The system only connects to it as to a service and offers its functionality further to cloud service developers. Dynamic EPSs are fully managed by the management system from their creation to their removal. They can be requested by the cloud service developer and used to support one or multiple **CloudService**.

The EPSs, obviously, share many similarities with OSUs. They are both offered to cloud service developers by the platform, EPSs could also be characterized by *Resources*, *Qualities* and *CostFunctions* and the dynamic EPSs can even be described as *CloudServices*. That is why we propose to handle EPSs as a special type of OSUs. The information model is powerful enough to accommodate EPSs as OSUs, but it needs to be extended with one additional relationships:

- **CloudService** *supported by* **OsuInstance** - The analogy of an OSU / **OsuInstance** as a building block, which we used before, is not best suited for this case. This analogy emphasizes the concept of OSU as a structural or functional part of a cloud service. That is, however, not any more the only exclusive role of OSU. To keep the analogy consistent, we extend it, by saying that the OSU can be not only a building block of a cloud service, but it can also represent a non-functional aspect of a cloud service. This way, the *support by* relation enables the EPS defined as OSU, to support the non-functional aspect of an elastic cloud service, the elasticity.
- **OsuInstance** *is* **CloudService** - Once the user have selected an OSU (a dynamic EPSs) and configured it by selecting the *Resources*, *Qualities* or *CostFunctions*, a new *OsuInstance* is created in the model. Because this *OsuInstance* corresponds to an instance of a dynamic EPS, there is also a new created *CloudService* in the model. That means that the *OsuInstance* is in fact **CloudService**

For completeness we point out that the relation **OSU** *is* **Template**, mentioned in the Section 3.1, captures the fact that a dynamic EPS is described as a standard *Template* for cloud service.

The introduced changes result in, at the first glance, complicated relations as a *Template* can be an OSU and an OSU can be an EPS. To clarify this phenomenon, we enumerate the 5 possible combinations, denoted by the same numbers in the Figure 4.7 :

1. Only **Cloud Service Template** - This is a standard template for elastic cloud service defined by a cloud service developer, without any additional roles, intended for simple reuse for multiple instances of a cloud service.
2. Only **OSU** - Defined based on the services offered by the the underlying cloud. It can be a type of virtual machine or a software component. Such services are typically included in the current PaaS/IaaS. An OSU unifies the way how these services are manipulated with, by the actors of the management system.

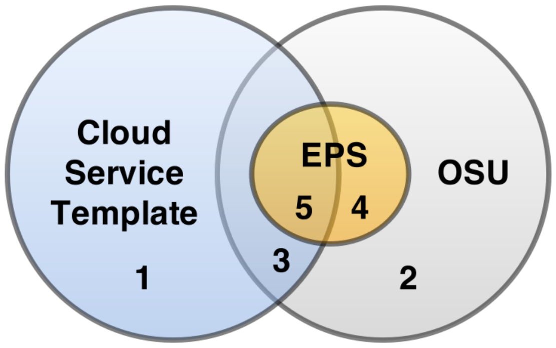


Figure 4.7: Clarification of the relations between terms Cloud Service Template, EPS and OSU.

3. **Cloud Service Template** and **OSU** - A cloud service template offered to be used as a part of other cloud service.

4. **EPS** and **OSU** - Static EPS, such as the Central Deployment Service. The lifecycle of static EPSs is not managed by management system, it only connects to these services through an adapter. The OSU contains only information, how to connect to and call the static service. These OSUs can be used only in the *supported by* relation.

5. **EPS** and **OSU** and **Cloud Service Template** - Dynamic EPSs which are fully under the control of the management system. They have all the properties of a standard cloud services. If they are to be instantiated, they need to be *supported by* a Deployment Service. Similarly to the static EPSs, they can be used only in the *supported by* relation.

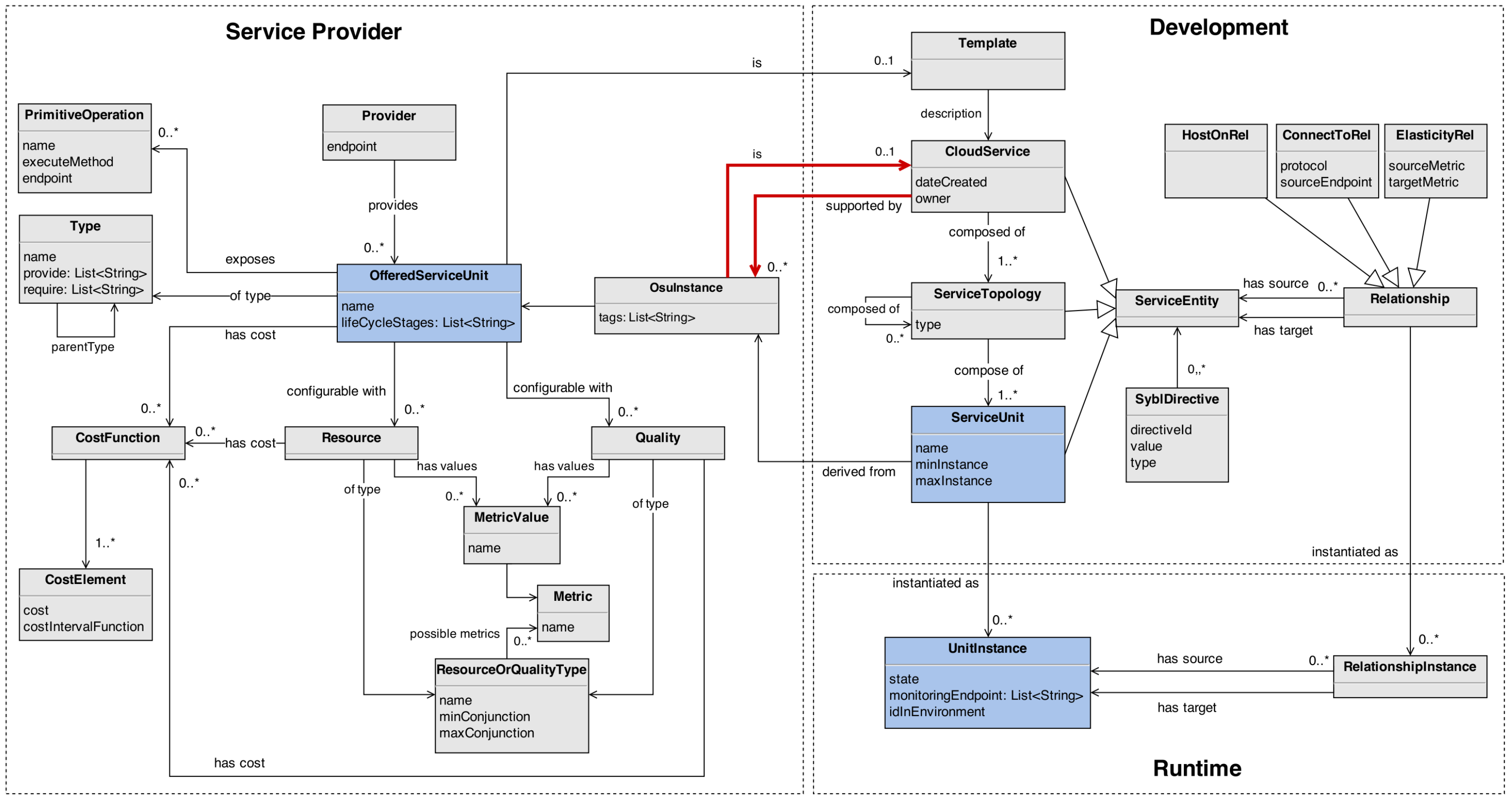


Figure 4.8: UML Class Diagram - Enhanced Information model

CHAPTER 5

Management System Design

In the previous chapter, we proposed a high-level architecture of the management system composed of a Lifecycle manager, the Information service and participants of the of cloud service's lifecycle. We will continue with the participants in the Section 5.1, which are designed to cover the functionality expected in the use cases, as well as general requirements for the management system. The Section 5.2 further refines the communication model and the Section 5.3 the Lifecycle manager. Rest of the chapter discusses in each section one of the participants mentioned in the overview.

5.1 Participants Overview

Adapter

An adapter observes the Lifecycle and Custom events, acting as a delegate of an EPS. It knows, which state or which event should activate certain functionality of the associated EPS. If such condition is fulfilled, the adapter reads the information necessary as an input data for the EPS from the information model and directly calls the EPS. Adapters also send events to the Lifecycle manager on behalf of the EPS, e.g. when a deployment is finished.

User-interaction Adapter

The same way that the regular Adapter represents the EPS in the communication withing the management system, the User-interaction adapter represents human or software actor that is outside of the management system. It translates actor's actions into events or reads the data in the information model. E.g. when user starts a new cloud service, the coordinator triggers an event that marks starting of a cloud service.

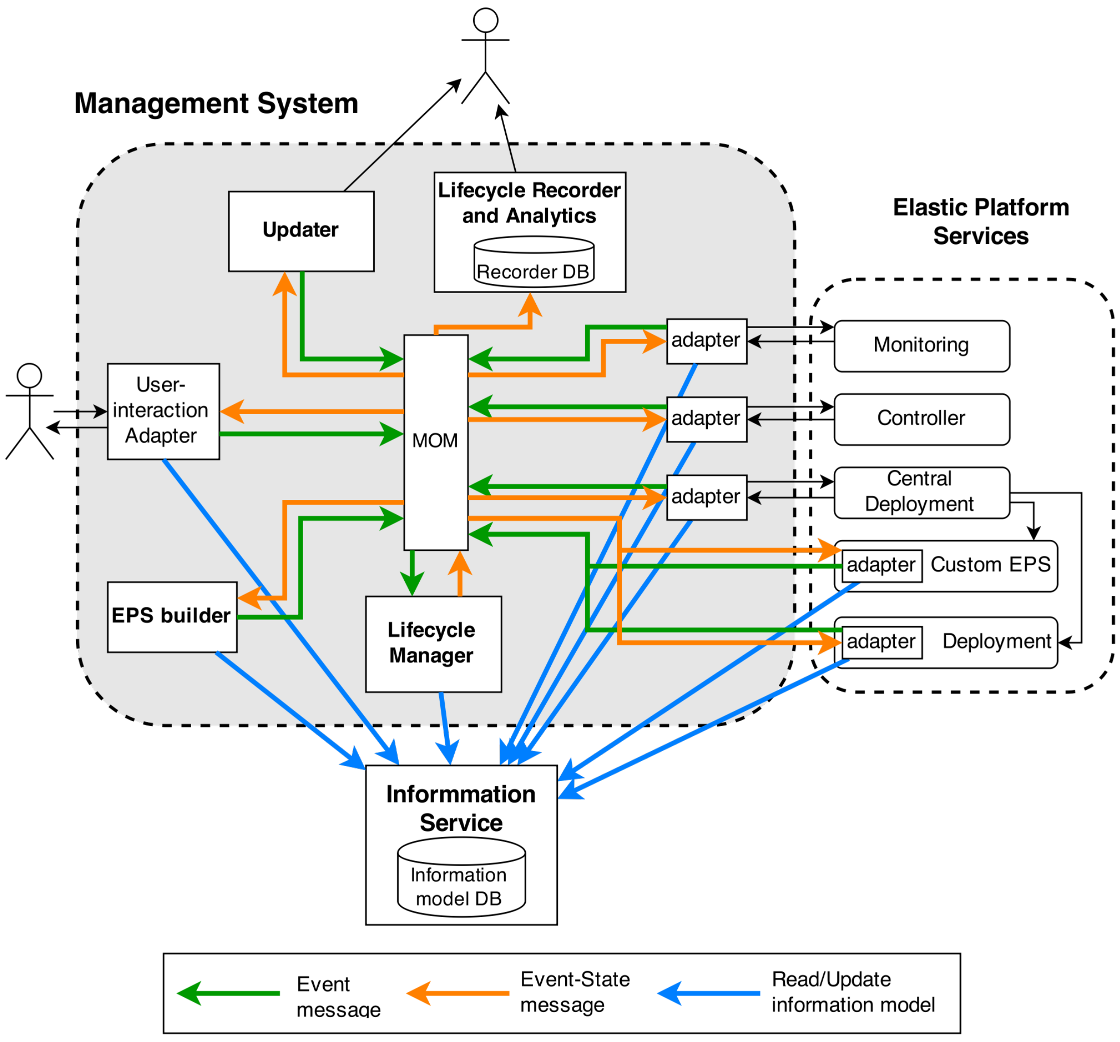


Figure 5.1: Architecture of the management system

EPS Builder

The EPS Builder manages dynamic EPSs and creates adapters for static EPSs at the set-up time of the management system. The dynamic EPSs are described in the information model as standard elastic cloud service. On the other hand, the static EPSs are not managed by the Builder as whole, but only their adapters. The builder responds to a set of custom events that mark assignment of EPSs and creation/removal request of dynamic EPSs.

The Builder can not properly function without a static deployment service, as it does not have capability to deploy a cloud service on its own. When the Builder is issued to instantiate an EPS, it does it through a sequence of Lifecycle events and writes to the information model, the same way as a user would create and start any other cloud service.

EPS Builder supports use cases, where we need to either include a completely new EPS to the elastic platform (see 3.3.2) or instantiate a private instance of an EPS (3.3.3).

For more details see the Section 5.4.

Lifecycle Recorder and Analytics

In order to perform the analytic use cases (3.3.4, 3.3.6, 3.3.7), we record all events and the resulting states during the lifecycle of a cloud service. This participant does not produce events, only listens to all the communication that occurs in the environment of the management system. The recorded data are further analyzed, but can also be offered to the cloud service developer or PaaS provider as a datasource for building of a custom EPSs.

The analytic part works with the data gathered by the Recorder and historical data from some EPSs such as monitoring service. The analytic tasks could include comparisons after test of multiple cloud service's elastic configurations (3.3.4) or one service can be observed over longer period of time to search for undesired behavior and further optimization of the elasticity (3.3.6). Periodic analysis can be performed to monitor for failure of service units and to trigger an alert and a recovery procedure (3.3.7).

More information about the Lifecycle Recorder and Analytics can be found in the Section 5.7.

Cloud Service Updater

Cloud Service Updater is responsible for selective redeployment of service units or topologies during development of cloud services (3.3.3). Moreover, it executes complex update and migration of service units (3.3.5), based on the comparison of the new cloud service description, against the description of the currently deployed service, which is available in the information service.

When extending an instantiated dynamic EPS to a new cloud (3.3.1), the updater is able to deploy a local component to the new cloud by updating the EPS defined as a cloud service.

The Updater has its own set of custom events that it responds to and utilizes Lifecycle as well as Custom events to redeploy parts of elastic cloud services in a fine-grained way.

Although we elaborate on the Updater in the Section 5.8, we will not implement this component in the prototype. It serves only for illustration of the possibilities of participants within the Maintenance state. The functionality of the proposed Updater could also be contained within a specialized EPS or further extended for other Maintenance tasks.

5.2 Communication Model

The events are propagated from a participant to the Lifecycle Manager and then distributed to all interested participants using the Message passing pattern. This ensures referential decoupling of the Lifecycle manager and the participants, which is essential for the ability of the management framework to accept also future EPSs.

5.2.1 Message-Oriented Middleware (MOM)

For the exchange of messages between the Lifecycle Manager and the participants, we assume an existence of a MOM with following capabilities:

- There are message queues and mechanism for notification of a consumer, who is subscribed to the queue.
- There are endpoints, where messages can be send to.
- Consumers can pre-define, what types of messages, from which endpoints should be routed to their queue.

Messages are exchanged indirectly, through the MOM, from the Lifecycle manager to participants and from a participant to the Lifecycle manager. In detail, a messages is sent to a logical endpoint in the MOM, then routed to the respective queues and from there accepted by the consumer of the queue. We utilize the MOM in the following way:

- Every participant is subscribed as a consumer to one or multiple queues. Every such queue belongs exclusively to that single participant. A participant pre-defines, which messages will be sent to its queue, based on the type of the message, cloud service and its state or development of the Lifecycle.
- Lifecycle Manager accepts messages through queues, where it is also an exclusive consumer. Messages send by Lifecycle Manager are cloned and distributed to all the queues that have the filter set to accept them.
- We define three logical endpoints, which correspond also to three basic types of messages: 1) Event-endpoint and Event Message, 2) Event-State-endpoint and Event-State Message 3) Exception-endpoint and Exception Message.

We will describe the three types of the messages and fields that they contain. Furthermore, we also define the properties, according to which the routing of messages to the message queues can be specified.

5.2.2 Messages

Event Message

Event messages are sent from participants through the endpoint to the Lifecycle manager. An event is either of the type Lifecycle event or of the type Custom event. Both are represented by a message that contains all the fields described in the Table 5.1. There are also several additional field characteristic for each type of the Event messages that are explained in the rest of this section.

Field	**Description**
Service-Id	Identifier of the cloud service, that the event is targeted to.
Group-Id	Identifier of the target Group (see the Section 5.3)
Event-Name	The name of the event must be uniquely identified in the scope of either Lifecycle events or scope of Custom events.
Source-Participant-Id	Identifier of the participant from which the event originates.
Timestamp	Time when the event was created by the Source-Participant.

Table 5.1: Fields common for the Custom event as well as the Lifecycle event.

In the **Lifecycle event message** in contrast to the Custom events, there is a set of predefined values (see the Section 4.2.2), which can be used as Event-Name. Furthermore, a specialized type of a Lifecycle event is used for events that modify the structure of the service itself and thus require additional information. These are events `CREATED`, `DEPLOYMENT_STARTED`, `DEPLOYED` and `MOVED`, `UPDATED` `RECONFIGURE_ELASTICITY`. This specialized Lifecycle events may contain following additional fields described in the Table 5.2.

Field	**Description**
Parent-Group-Id	In events `CREATED` and `DEPLOYMENT_STARTED` this field identifies the group to which a new group (topology, unit or unit instance) will be appended. In event MOVE it identifies the new parent, where the targeted group will be appended.
Service-Part	Representation of either service, topology or unit that is `CREATED` by this event. Or representation of unit instance that was created by deployment, in event `DEPLOYMENT_STARTED` and `DEPLOYED`.

Table 5.2: Additional fields of a Lifecycle events of the type `CREATED`, `DEPLOYMENT_STARTED`, `DEPLOYED` and `MOVED`.

A **Custom event message** contains all the fields from the Table 5.1 and additionally two optional fields as described in the Table 5.3.

Field	Description
Target-Participant-Id	A custom event can be intended as a command for one specific participant, which is marked by this field.
Custom-Message	Any text that is intended as an input parameter for a participant. This could be, e.g. JSON or XML.

Table 5.3: Optional fields of a Custom Event.

Event-State Message

Event-State messages are sent from Lifecycle manager to participants. They are composed of the corresponding events that were processed by the Lifecycle manager, the current structure of the targeted service and information about the state transitions of all groups. For every group the last Transition is sent, no matter whether that transition happened during evaluation of this event or already before. Each transition is composed of properties as described in the Table 5.4. Although, the amount of information about a cloud service sent in every Event-State message might be considered unnecessarily extensive, we should not forget that the purpose of these messages is to notify every participant that is interested in it, about the complete runtime state of the cloud service.

Property	Description
Group-Id	Identifier of the target group
Group-Level	The level of the group. Possible values are SERVICE, TOPOLOGY, UNIT, INSTANCE
Current-State	Current state of the group.
Previous-State	Last state before the current state.
Fresh	Whether this transition happened due to this event or already before, due to some other event.

Table 5.4: Properties of of Transition contained in the Event-State Message.

Exception Message

Exception messages can be be sent by participants as well as Lifecycle manager, but are received only by participants. We distinguish between two main classes of exceptions: First, a Lifecycle Exception occurs, if a Lifecycle action is validated as not allowed in the current state in the Lifecycle manager. This means that the Event will not be sent to participants as an Event-State message, but will be discarded and sent as Exception Message. The second class, Standard Exception indicates any unexpected situation either in the Lifecycle Manager or a Participant. The fields of Exception Messages are described in the Table 5.5.

Field	Description
Service-Id	Identifier of the cloud service.
Source-Participant-Id	Identifier of the participant from which the event originates.
Timestamp	Time when the event was created by the Source-Participant.
Type	(Optional) Identifier of the type of the exception.
Message	(Optional) The main description of the exception, preferably description in a natural language.
Detail	(Optional) Detailed description, e.g. stack trace.
Event	(Optional) Specific for Lifecycle Exceptions, contains the Event that caused the Exception.

Table 5.5: Fields of Exception Message

5.2.3 Routing Filters

For each endpoint the properties for routing of a message to message queues are little different. Some properties might be considered redundant, but they are useful to enable the participants easy definition of filtering of the messages that they are interested in. The Table 5.6 describes these properties and shows which property is considered for which type of endpoint. Additionally, the Event-State endpoint distinguishes between Event-State messages that contain Lifecycle event and those that contain a Custom event.

Properties	Message Type				Description
Service-Id	ES(lc)	ES(c)	E	EX	Identifier of the cloud service instance.
Event-Name	ES(lc)	ES(c)	E		The type of the event must be uniquely identified in the scope of either Lifecycle events or scope of Custom events.
Target-Level	ES(lc)	ES(c)	E		The level of the target group. Possible values are SERVICE, TOPOLOGY, UNIT, INSTANCE
Change	ES(lc)				Whether the event caused a change of state at the SERVICE level.

State-Before	ES(lc)				State at the level SERVICE that was before this event was evaluated.
State-After	ES(lc)				State at the level SERVICE that is after evaluation of this event.
Source-Participant-Id	ES(lc)			EX	Identifier of the participant from which the event originates.
Target-Participant-Id		ES(c)			Identifier of the participant that the event is intended for.

Table 5.6: Properties of a message used for filtering and routing of messages. ES(lc) - Event-State messages that contain Lifecycle event, ES(c) - Event-State messages that contain Custom event, E - Any Event message, EX - Exception message

5.3 Lifecycle Manager

The internal structure of the Lifecycle manager is shown in the Figure 5.2. It is composed of *Creator* and a set of *Instance Lifecycle Managers*. The *Creator* receives all Event Messages of the type `CREATED` targeted at level SERVICE from the *Manager Queue*, which is created at start-up time of the management system. For every such message it creates a new instance of *Service Lifecycle Manager*. That is responsible for the entire Lifecycle of single cloud service and will be removed, when the service is removed from the management system. The *Service Lifecycle Manager* processes the `CREATED` event and creates its own message queue, where it will receive all the Event messages that concern his cloud service. It dispatches the `CREATED` Event-State message to the Event-State endpoint to notify participants, that a cloud services was created, as well as notifying the source participant of the `CREATED` event that it has been processed and the Lifecycle manager is ready for other events that concern the cloud service.

When an Event message is received and it is a Custom event, the snapshot of the state of the cloud service instance is created and inserted together with the Custom event to the outgoing Event-State message. However, if the Event message is a Lifecycle event, the event must first be verified and executed on the Lifecycle. Both these cases can be observed in the Figure 5.3.

Groups

The evaluation of a Lifecycle event on a cloud service is executed with help of the structure depicted in the Figure 5.4 with the central role played by the concept of Groups. A Group unifies the concepts of cloud service, service topology, service unit and unit instance, from the perspective of state within the lifecycle. It is based on the fact that a cloud service, service topology and service unit are abstract entities and that only a

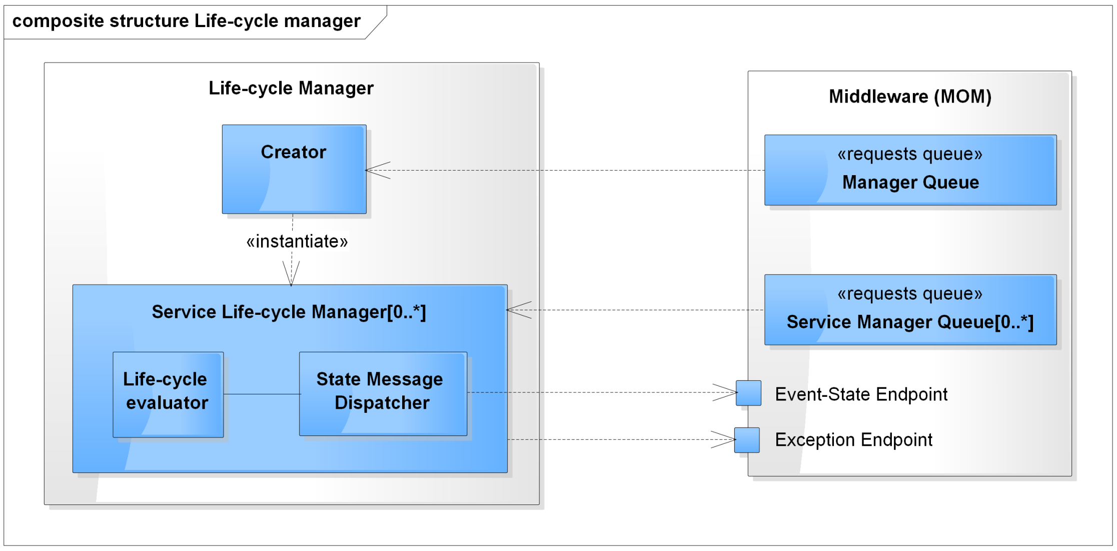

composite structure Life-cycle manager
Life-cycle Manager
Creator
«instantiate»
Service Life-cycle Manager[0..*]
Life-cycle evaluator
State Message Dispatcher
Middleware (MOM)
«requests queue» Manager Queue
«requests queue» Service Manager Queue[0..*]
Event-State Endpoint
Exception Endpoint

Figure 5.2: Design of Lifecycle manager

unit instance, in fact, has a state of its own. A state of the other entities is therefore determined as an aggregate of the states of the unit instances included within that entity.

Groups are organized in a tree structure, with cloud service as the root Group. That Group contains service topologies as its member Groups, which in turn contain other topologies and service units as Groups. Service units have also member Groups, unit instances. Unit instances are characteristic by never containing any member Groups, but have no special properties of their own. Each group is uniquely identified by an ID, that is identical to the ID used for underlying entity inside the information model. Each groups also keeps the current state, previous state, the type of entity it belongs to and reference to its immediate parent in the tree structure.

Groups perform evaluation of a Lifecycle event with help of a *LifeCycleCollection* and *AggregationStrategy*. There are currently three slightly different *LifeCycles* as identified in the Section 4.2.1: one for Groups of type SERVICE, one for TOPOLOGY and UNIT and the last one for the type INSTANCE. A *LifeCycle* is composed of sets of *States*, *Actions* and allowed *Transitions* between the states. *Transitions* that can not be captured in the *LifeCycle* itself are defined in the *LifeCycleCollection*. An example can be the situation, when a unit instance is in state UNDEPLOYING, the last instance is being undeployed and thus moving to state FINAL (a unit instance know no state PASSIVE). But a service instance without unit instances should move to the state PASSIVE. Standard aggregation of group's state is handled by the *AggregationStrategy*, which can decide the resulting state with use of current and previous state of the group itself, its type, its members or its parent. The default AggregationStrategy of Groups is to accept the state of its members, once all the members are of the have the same current state.

The Algorithm 1 shows in pseudo code, how are the states of the groups evaluated at each Lifecycle event. The input Group for the function **executeAction** is the Group

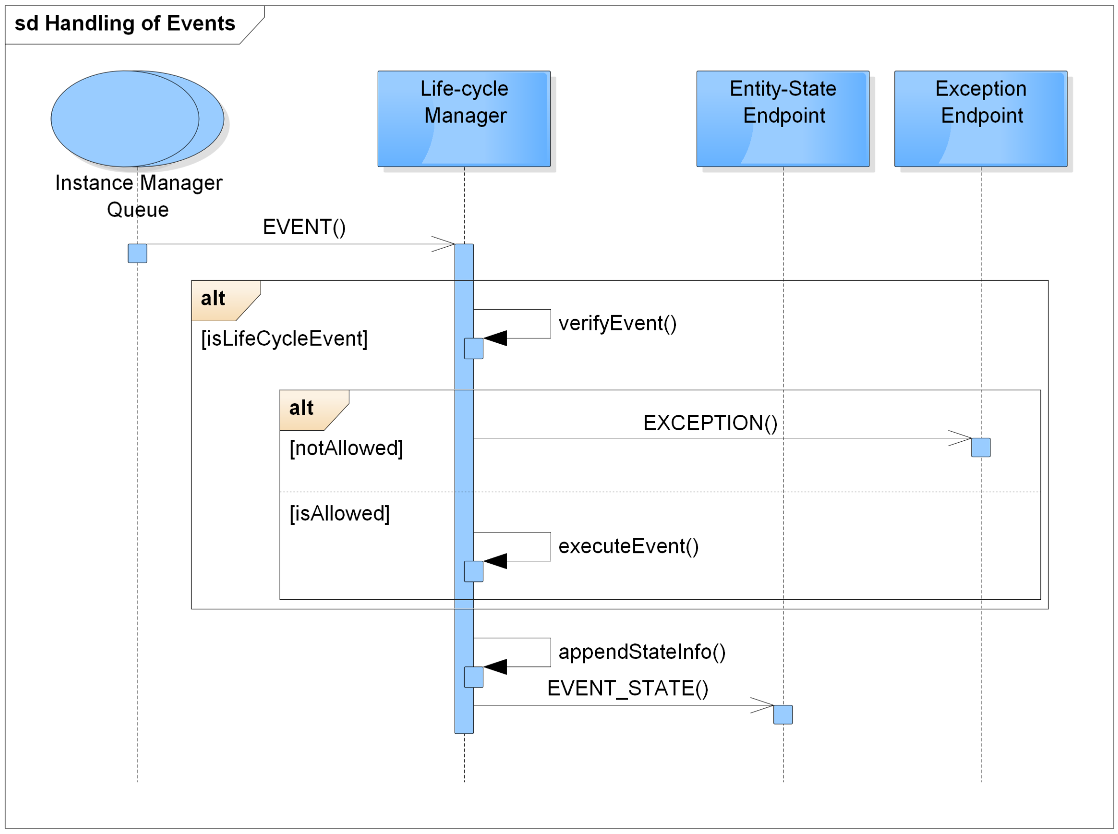


Figure 5.3: Generic handling of events by the Lifecycle manager

targeted by the Event. This Group determines the sub-tree of Groups that will be evaluated. We traverse the sub-tree depth-first and at every leaf Group (a Group without member, a unit instance or service unit), call the appropriate *LifeCycle* to determine the next state. If the current Group has an immediate parent, this parent is issued to refresh his state. The **refreshState** function is composed of two steps. First, the *AggregationStrategy* returns the new state, second if the state is different from the old state, the the signal for refreshment of state is send recursively up until the root.

Besides evaluating the Events, the Lifecycle manager uses Events that modify the structure of the service itself to adapt the tree of Groups accordingly. This happens, however, only if the Lifecycle event is evaluated as acceptable, given the current state of Groups. Moreover, it updates the Service Developer Space and the Runtime Space in the information model. In this way the information stored in the model and the information sent in the Event-State messages is allays consistent.

- `CREATED` - If targeted on SERVICE level, results in initiation of a new *Service Lifecycle Manager* and creation of the initial tree structure of Groups. When targeted on levels TOPOLOGY of UNIT, creates new topology/unit, but such event requires

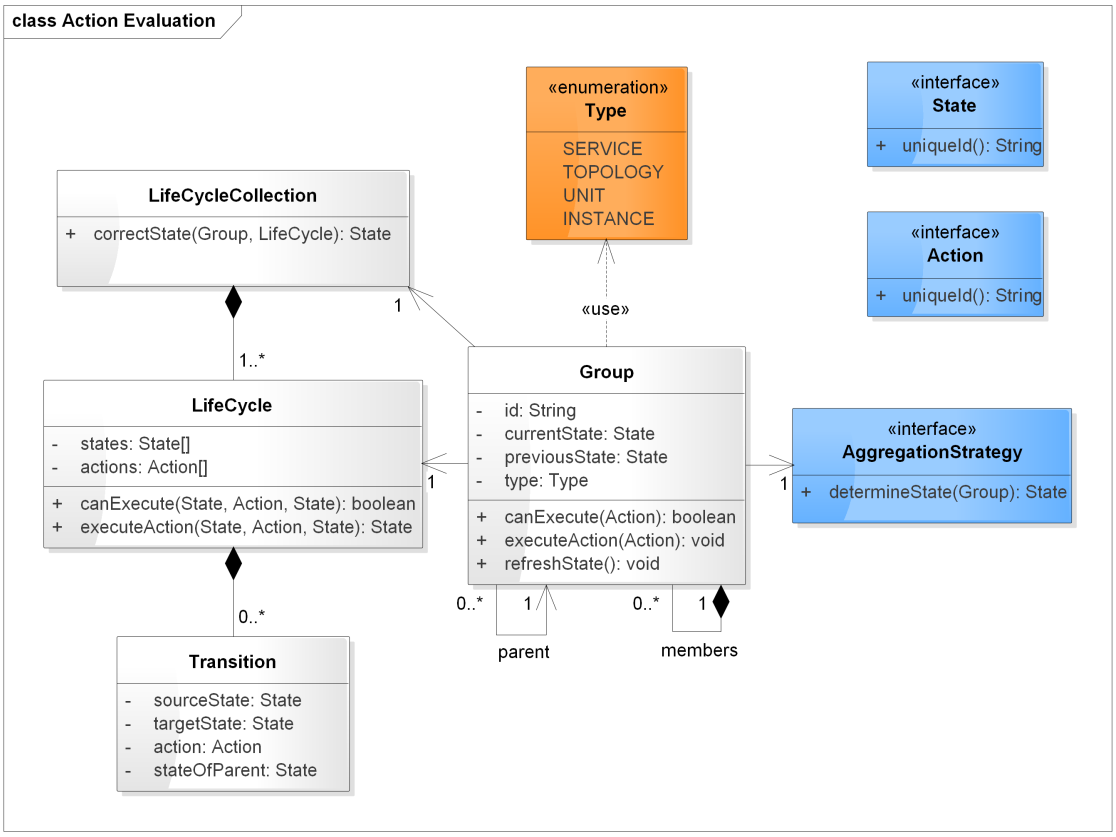


Figure 5.4: UML Class Diagram - Domain of Lifecycle event evaluation

the optional field *Parent-Group-Id* and *Service-Part*. The groups in all cases are created recursively, for any Service Entities included in the root *Service-Part*.

- DEPLOYMENT_STARTED - If targeted on the level INSTANCE, it means that a new Unit Instance was created, thus also new Group will be created respectively. If targeted on any other level, no change of structure is implied.
- DEPLOYED - If targeted on level INSTANCE, although there is no update of Groups needed, the information model is updated with the new state. Similarly to DEPLOYMENT_ STARTED, if targeted on any other level, no change of structure is implied.
- UNDEPLOYED - That means that Groups at level INSTANCE, move to the state FINAL and will be removed from the tree. Groups at other levels only adapt their state according to the Lifecycle and AggregationStrategy.
- REMOVED - Only SERVICE, TOPOLOGY or UNIT can be the target of this event. In all cases, if the event is valid, the target group and its nested members are removed. If targeted on level SERVICE, additionally also the entire *Service Lifecycle Manager* and its queue are removed (after sending the Event-State message).

Algorithm 1 Recursive evaluation of Lifecycle event on the cloud service's structure and states of its parts.

```
function EXECUTEACTION($group$, $event$, $lifeCycleCollection$, $strategy$)
    if $group$.hasNoMember() then
        $lifeCycle$ = $lifeCycleCollection$.lifecycleForGroup($group$)
        $group.previousState$ = $group.currentState$
        $group.currentState$ = $lifeCycle$.nextState($group.currentState$, $event$)
        if $group$.hasParent() then
            refreshState($group.parent$, $strategy$)
    else
        for each $member$ of $group$ do
            executeAction($member$, $event$, $lifeCycleCollection$, $strategy$)
function REFRESHSTATE($group$, $strategy$)
    $nextState$ = $strategy$.determineNextState($group$)
    if $nextState$ not equal $group.currentState$ then
        $group.previousState$ = $group.currentState$
        $group.currentState$ = $nextState$
        if $group$.hasParent() then
            refreshState($group.parent$, $strategy$)
```

- `MOVED` - Acceptable only on levels TOPOLOGY or UNIT. It enables change of structure, without requiring undeployment of Unit Instances. It can be used only in state UPDATING and results in rearrangement of Groups.
- `UPDATED` - Recreates the targeted Group and its sub-tree. Updates the information model.
- `STOP` or `START_MAINTENANCE` - If there is an elasticity controller assigned to the cloud service, it first sends the `STOP_CONTROLLER` event and cashes the original `STOP` or `START_MAINTENANCE` event.
- `STOP_CONTROLLER` - The elasticity controller returns this event after it stopped. The Lifecycle manager sends the original event `STOP` or `START_MAINTENANCE` to acknowledge that the cloud service may really move to the requested stage.
- `RECONFIGURE_ELASTICITY` - Updates elasticity configuration in the information model.

5.4 EPS Builder

The design of the EPS Builder (or just Builder), from the perspective of communication, is in fact an example of a simple generic participant. As seen in the Figure 5.5, it is composed of a *Listener* that reads messages from the queue and calls the *Core logic*

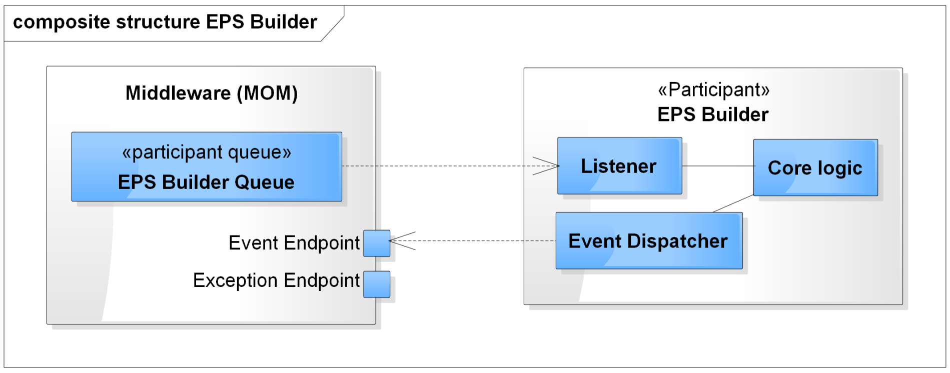


Figure 5.5: Design of the EPS Builder.

component to process an Event-State message. The *Event Dispatecher* sends produced Event messages to the *Event Endpoint*.

The Builder reads information about EPSs from the information model. An OSU is considered represent an EPS if it is of type EPS. Static EPSs are distinguished from dynamic EPSs by the fact that OSUs representing a static EPS are not associated to a CloudService by the *is* relation.

At this place we have to mention, that besides the Lifecycle events there are also other core events of the management systems. They represent an EPS in the Lifecycle of a cloud service and are modeled as Custom events. They are listed in the Table 5.7. The support expresses whether an EPS provides its service to a cloud service. If an EPS supports an instance it listens to the Events about that instance. This relation is stored also in the information model as the *suppoted by* relation between a *CloudService* and an *OsuInstance*.

Name	**Description**
EPS_SUPPORT_REQUESTED	A participant have requested support the and EPS should react to this request.
EPS_SUPPORT_ASSIGNED	An EPS marks the moment, from which it will serve Events from certain cloud service.
EPS_SUPPORT_REMOVED	Immediately terminates the support.
EPS_DYNAMIC_REQUESTED	Request for a new instance of a dynamic EPS.
EPS_DYNAMIC_CREATED	A dynamic EPS is created and ready for EPS_ SUPPORT_REQUESTED events.
EPS_DYNAMIC_REMOVED	Undeployment of a dynamic EPS requested.

Table 5.7: Custom events that define interaction with all EPSs.

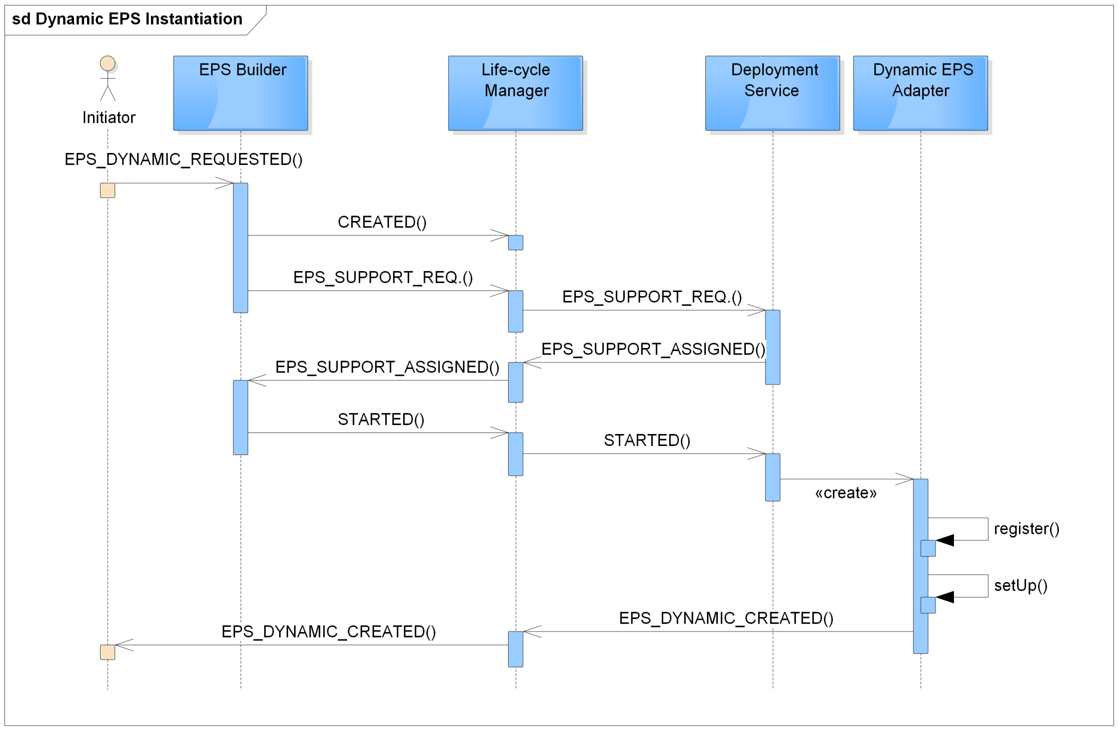


Figure 5.6: Process of creating a dynamic EPS.

The Adapters of static EPSs are created at start-up time, based on the data from information model, while dynamic EPSs are instantiated with their adapters. The figure Figure 5.6 describe this process. There are 4 life-lines representing the Builder, Lifecycle Manager, Deployemnt Service and the dynamic EPS represented by the dynamic Adapter. The Lifecycle manager in this diagram, is an abstraction of the communication model composed of queues and endpoints. For the sake of readability of the diagram, we show the messages, as if they were sent directly to the respective participants, although they are, in fact, sent allays to endpoints and from there routed to the interested queues.

The sequence is initiated by a EPS_DYNAMIC_REQUESTED event for a dynamic EPS. The Builder starts a new process, which in its generic form, may be considered a standard process for creation and deployment of cloud service. The Builder creates the cloud service and requests for support of a deployment service. After the support is granted, it starts the cloud service. When a dynamic Adapter is successfully deployed, it is started with the information about host and port of the middleware as well as of the information service. Moreover, it receives the unique identifier of the cloud service that it belongs to. It uses this information to create its *OsuInstance* in the database and obtaining in this way its unique participant identifier. That is essential for joining of the management system. Finally, the new Adapter of the dynamic EPS notifies other participants of its existence by the EPS_DYNAMIC_CREATED Event.

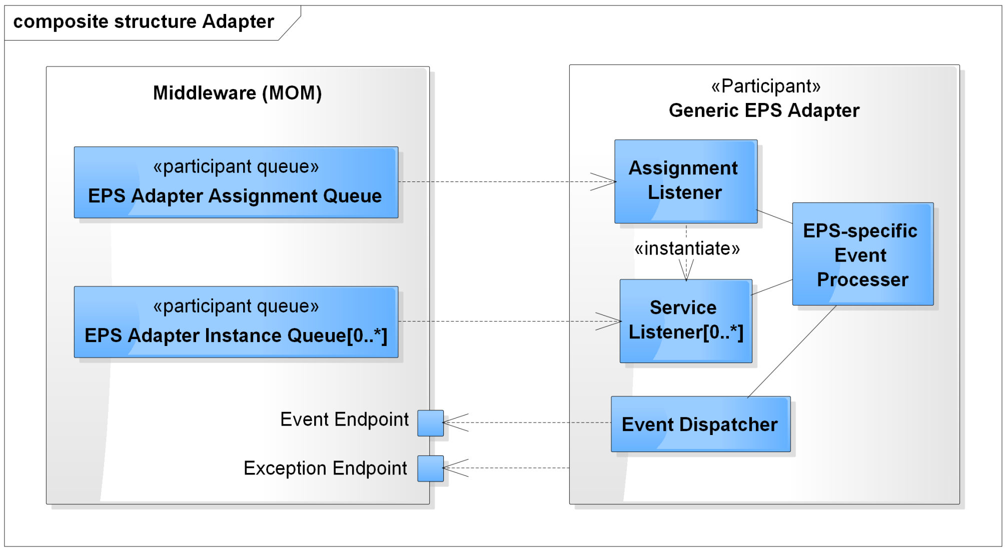


Figure 5.7: Design of a generic EPS Adapter and its relation to the middleware.

5.5 Generic EPS Adapter

We propose a following generic Adapter to intermediate the communication between the management system and an EPS. Despite this, any software component capable of the communication as described in the Section 5.2, could be used instead, as long as it suits the needs of the related EPS. As depicted in the Figure 5.7, the Adapter is composed of 4 main logical components.

The **Assignment Listener** is created at the start-up time of the Adapter and is responsible for creation of the *EPS Adapter Assignment Queue* and its routing filter. The *Assignment Listener* processes only Custom events with name EPS_ASSIGNMENT_REQUESTED targeted to the SERVICE level with specified Target-Participant-Id matching the ID of the Adapter (every Adapter is a participant, that is why it must have an ID unique in the scope of all participants). Every such event means that the adapter was assigned to a new cloud service, thus the *Assignment Listener* creates a new instance of *Service Listener* and matching *EPS Adapter Service Queue*, which will attend only to the events from this specific cloud service. This ensures that the events concerning one service will not delay Event-State messages for another cloud service, but still will be served in the same order as produced by Lifecycle manager. After the *EPS-Adapter Service Queue* is created, the assignment is confirmed by sending a EPS_ASSIGNED Event. Thanks to this, the other participants are notified, when it is guaranteed that the EPS will be able to receive an Event. The **Service Listener** receives Events only from one specific cloud service and calls the *EPS-specific Event Processor* to process them. The **EPS-specific Event Processor** contains the mapping of certain Event-State messages to the operations of the EPS. t also evaluates the returned values as determines, when to send new Events. The *EPS-specific Event Processor* also defines the routing filter for *Service Listeners*. The Events produced by the adapter are created and dispatched by the **Event Dispatcher**. Last but not least, as any other participant, the EPS Adapter can also send Exception events to the Exception endpoint.

5.6 User-interaction Adapter

In order to allow basic interaction with the management system also to systems that are not compatible with the communication model, we design a reduction that partially exposes the management system through a REST API. A User is considered any such system, which influences and observes the management system only through the intermediary User-interaction Adapter.

Multiple resources are identified by an URI containing path parameters *serviceId*, *epsId* and *epsInstanceId*. We will not repeat the explanation at each resource, but rather state it now: A *serviceId* is a unique identifier of a cloud service, *epsId* is a unique identifier of an EPS, e.g. RSYBL and *epsInstanceId* identifies instance of an EPS, e.g. RSYBL_DYNAMIC_1 or SALSA_STATIC.

HTTP Method	POST
URI	/services
Description	Create a new cloud service in the information model as well as in Lifecycle manager. This triggers the CREATED event.
Parameters	**Body:** Description of the new cloud service to create.
Response	A unique identifier of the newly created cloud service instance (instanceId).

HTTP Method	DELETE
URI	/services/{serviceId}
Description	Remove a cloud service from the management system, including information model and the Lifecycle manager. This triggers the `REMOVED` event.
Parameters	**Path:** serviceId

HTTP Method	PUT
URI	/services/{serviceId}/active
Description	Start the cloud service by triggering the `START` event. If there is a deployment service assigned, the service will be deployed.

Parameters	**Path:** serviceId

HTTP Method	DELETE
URI	/services/{serviceId}/active
Description	Stop the cloud service by triggering the `STOP` event and consequently undeploying the service.
Parameters	**Path:** serviceId

HTTP Method	PUT
URI	/services/{serviceId}/kill
Description	Immediately undeploys cloud service.
Parameters	**Path:** serviceId

HTTP Method	PUT
URI	/services/{serviceId}/elasticity
Description	Reconfigure elasticity with the new directives.
Parameters	**Path:** serviceId **Body:** New elasticity configuration.

HTTP Method	GET
URI	/services/{serviceId}/events
Description	Register for all events from the Lifecycle of the given cloud service. This is realized through Server Sent Events (SSE) [1], what opens a stream (text/event-stream) for events to be pushed from the HTTP server to the client.
Parameters	**Path:** serviceId

HTTP Method	PUT
URI	/services/{serviceId}/supporting_eps/{epsInstanceId}

[1] http://www.w3schools.com/html/html5_serversentevents.asp

Description	Request for support of the specified EPS instance to the cloud service. Results in the EPS_SUPPORT_REQUESTED event being triggered.
Parameters	**Path:** serviceId, epsInstanceId

HTTP Method	DELETE
URI	/services/{serviceId}/supporting_eps/{epsInstanceId}
Description	Terminate the support of the specified EPS instance to the cloud service. Triggers EPS_SUPPORT_REMOVED event.
Parameters	**Path:** serviceId, epsInstanceId

HTTP Method	POST
URI	/services/{serviceId}/supporting_eps/{epsInstanceId}/ events/{eventName}
Description	Trigger a Custom event identified by *eventName* parameter. It is directed to a specific EPS instance specified by the *epsId* and the cloud service.
Parameters	**Path:** serviceId, epsInstanceId, eventName - the name of the custom event **Body:** May hold the optional message in the ”text/plain” format.

HTTP Method	PUT
URI	/eps/{epsId}/instances
Description	Create new instance of a dynamic EPS.
Parameters	**Path:** epsId

HTTP Method	DELETE
URI	/eps/{epsId}/instances/{epsInstanceId}
Description	Remove an instance of a dynamic EPS
Parameters	**Path:** epsId, epsInstanceId

HTTP Method	GET
URI	/lifecycle/level
Description	Read a Lifecycle corresponding to a certain level.
Parameters	**Path:** level - either SERVICE, TOPOLOGY, UNIT, INSTANCE
Response	Lifecycle valid for the specified level.

Table 5.8: REST API of the User-interaction adapter

5.7 Lifecycle Recorder and Analytics

The recorder should be capable for storing all the Event-State and Exception messages that are exchanged in the system. Furthermore, any change in the data describing a cloud service in the information model, whether runtime data, elasticity configuration or structure of the service, should be recorded as well. In order to be able to analyze the gathered data, it should be stored in a queryable way.

Because the information service uses a graph database for persistent storage, we will use graph database to store the changes of data in the information model and all the messages as well. Moreover, a graph database is an appropriate choice, since it allows high flexibility without usage of a schema and is performant, when executing complex queries over many entities.

As storing of changes to the information model, in a fashion that the data is not redundant and easily queryable is the most challenging task of the Recorder, we can thin of this problem, as of a problem of versioning of data in a graph database. That is why we investigate, what solutions are already proposed to this problem is the scientific literature and industry.

Versioning of Graph

One approach described in [32] is to manage revisions completely independently from the domain model. This solution is composed of a *DataGraph* and a *VersionGraph*. The *DataGraph* is the domain model graph not influenced by the versioning. In the *VersionGraph*, every version of every node and relationship from the *DataGraph*, is represented as a node. Relationships between these nodes are transitions from one version to another. This results in a framework independent from the domain and pluggable to any existing system.

Another existing approach [33] stores versions in the original domain model graph. The key principle here, is to separate structure from state and use time-based versions (as opposed to revision-id-based). There are two types of nodes and relationships. So-called

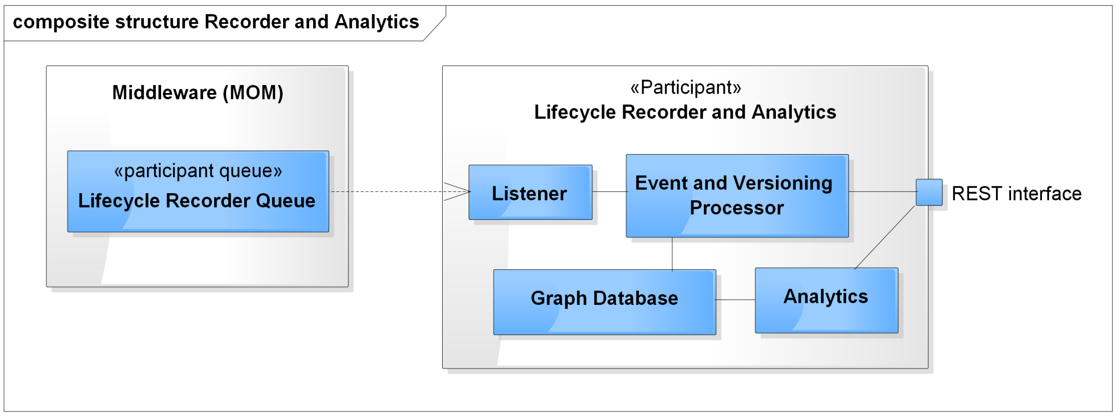


Figure 5.8: Design of the component Lifecycle Recorder and Analytics

Identity nodes represent entities from domain model and *Structural Relationships* define relationships between these entities. *State nodes* describe a state of an entity and are connected to an *Identity node* through a *State Relationship*. All relationships have from-to time stamps, thus determining, which states and relationships between domain entities were valid at a certain point in history.

We base our solution on the second approach, keeping the domain model and versions in one graph. This way, the queries can be relatively simple and utilize the entities and relationships from the domain model.

Design

The Lifecycle Recorder and Analytics receives events through a single queue, since in this case distinguishing between several queues each for one service has no added value. *The Event and Versioning Processor* stores Event-State messages in the *Graph Database* and transforms the graph to accommodate the newly changed cloud service description or its part if there is any. The *Analytics* analyzes the data in the database on-demand.

We divide the *Graph Database* into regions, each region being exclusive for single cloud service. The Figure 5.9 shows a simple example of one such region. The region node (red color) is connected to every *Identity node* (yellow nodes) with a _MANAGE relationship, which identifies the given node as part of the cloud service. Every versionable part of a cloud service that means every class of the information model from the Section 3.1, is represented by one such *Identity node*. Relationships between classes are the *Structural Relationships* with matching domain names. Properties of an *Identity node* are extracted into a *State node* (blue nodes) and connected by a _HAS_STATE relationship. The events are stored as the CHANGE relationship between the purple revision nodes. Every *Structural Relationships* or *State Relationships* created or terminated as an effect of an event, is be marked by the same timestamp value as the change, either as *from* or *to* property.

The example in the Figure 5.9 shows a cloud service with 2 topologies and one unit,

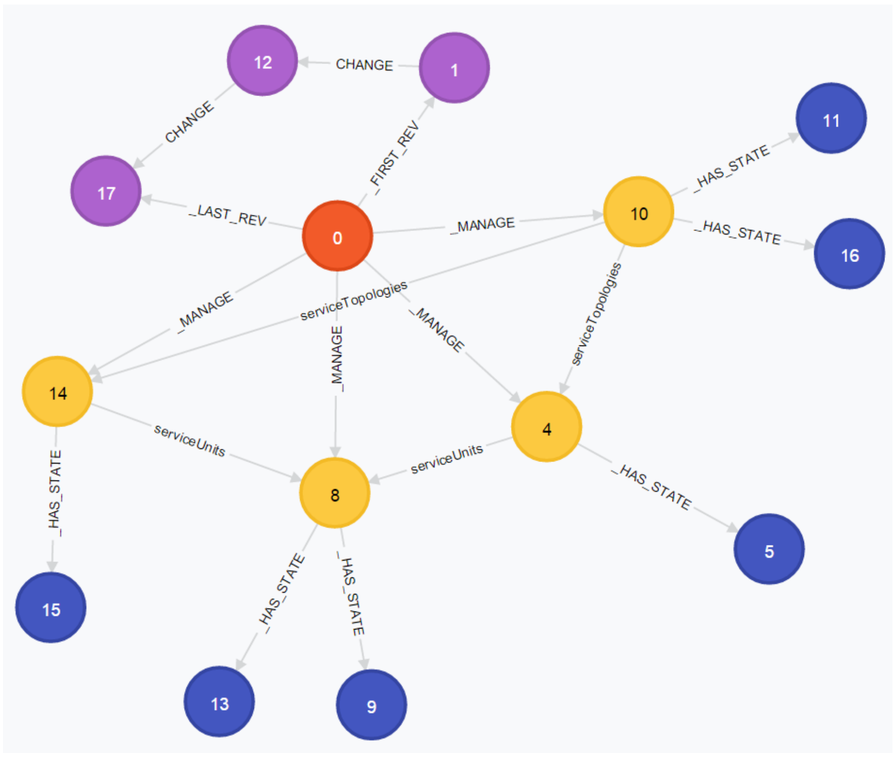


Figure 5.9: Example of a recorded cloud services as a region with following color coding of nodes: *Region* - red, *Revisions* - purple, *Identity nodes* - yellow, *State nodes* - blue

after 2 events. The properties of the cloud service and the unit changed once, that is why they each have 2 *State nodes*. Furthermore, the unit was moved from one topology to another thus having a current and historical *Structural Relationship* with both of them.

The recorder publishes an API that serves for easy recording of a new version of a cloud service or its part, retrieval of the cloud service or its part valid at a specific time in history and reading of the revision meta-data. More complex analytic tasks can utilize the querying API of the graph database itself.

5.7.1 Lifecycle Analytics

For the analysis we combine data recorded by the Lifecycle recorder as well as the data provided from the monitoring service. We can divide this primary data to following groups:

- *cloud service configuration:* service's structure, selected service units, elasticity's configuration
- *run-time data:* state of instance, length of transition between two states, number of instances of a service unit

- *monitoring data:* value of a metric, e.g. CPU usage or response time
- *events:* triggering of an elastic action, update of the structure, update of the elasticity's configuration

We propose several scenarios, which cloud be supported by the analytic engine:

Deployment and migration analysis

Based on changes of an instance's state over time, we are able to determine if a deployment or a migration of a cloud service is developing as expected, or if an error have occurred. For example if instantiating of a service unit takes several times longer than expected, we can assume that an error occurred. By looking into the type of the associated offered service unit, we find out that the unit is a VM. This could lead to realization, that the underlying cloud is actually lacking necessary resources.

Offered service unit and service unit statistics

From the perspective of a PaaS provider as well as cloud developer, it might be helpful to know the real data, about how long a service instance stays in a certain stage of its life-cycle. One use-case, could be a PaaS provider monitoring the average deployment time of all instances of particular offered service unit. Or a cloud service developer, monitoring only instances that are part of one cloud service. Such information could directly influence cloud developer's future choice of offered service units with shorter start-up time, over similar, but slower offered service units from a different cloud. Thus crucially influencing the elasticity of the resulting cloud service.

Correlation analysis

Over the life-time of a cloud service, there are multiple event that influence the elastic behavior of a cloud service. There can be, as already mentioned, update of the cloud service, update of elasticity's configuration, elastic action or updates of offered service units provided by third parties. By enabling cloud service administrator to see all these past events and compare them with the changes of certain chosen metric such as response time, we make it possible for her/him to quickly and effortlessly identify effects of events, or the other way around, which event have caused an unexpected fluctuation. Moreover, this process could be assisted by utilization of correlation algorithms.

Elasticity controller impact analysis

For verifying of controller's decisions, we could set up a watchdog to monitor, if a scale-out and scale-in actions do not occur in an inefficient way. A dangerous pattern could be an oscillation that happens too quickly in comparison to the time, which it takes to instantiate the service unit.

5.8 Cloud Service Updater

A simple deployment and undeployment of an entire cloud service can be performed through a deployment service. The objective of this component, is to execute a fine-grained update of an already deployed/instantiated cloud service, in a way, that only a minimal necessary amount of service units will be modified.

So if we, e.g. want to replace a war-packaged application in all instances of a service unit with a new version of that application, only the war file should be redeployed and not an entire VMs or the whole cloud service.

This component serves as theoretical illustration, of possible participant that operates in the Maintenance state.

5.8.1 Two-step update

As already mentioned, a cloud service can be decomposed into service topologies and service units, with an OSU assigned to every service unit. We will further distinguish service units with an OSU of type virtual machine (VM).

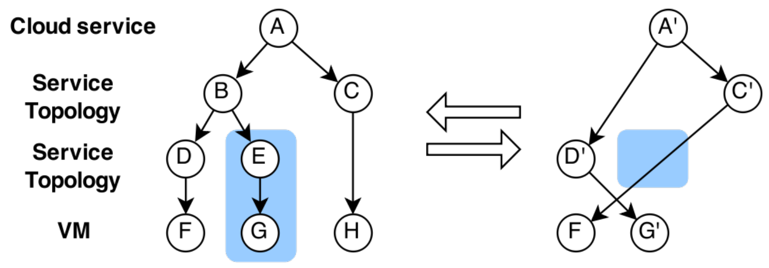
When resolving an update, it might be necessary to execute changes on the arrangement of service topologies and VMs as well as on the service units. But these two concepts are inherently different since a topology is a logical grouping of VMs, whereby a service unit is a pieces of an instantiated software. If a VM is to be moved to a different service topology, it does not always have to be re-instantiated. Only in cases, when the target service topology is in a different cloud as the source topology. In contrast to that, a service unit must always be re-instantiated, if it should be updated and, moreover, triggers an update of all the other service units deployed on it.

That is why an update of a cloud service should be handled in two subsequent steps. In the first step, we update structure, which means reorganizing the VMs and topologies according to the new configuration. In the second step, we will re-instantiate service units that need to be updated.

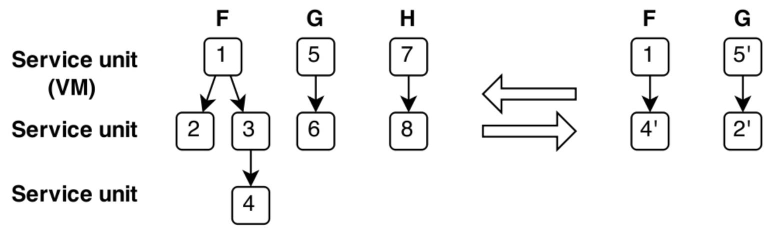
An example of a two-step update can be seen in Figure 5.10. The first part 5.10a shows a cloud service *A*, on the left side, composed of nested topologies *B* - *E*. Each topology contains one VM *F* - *H*. The topology *E* exists in a different cloud than the rest of the service. Notice the difference between the VMs *F* and *G*: *G* is moved to a different cloud, that is why it must be re-instantiated, on the other hand *F* is moved to a different topology but in the same cloud, so in this step, no re-instantiation is needed. The part 5.10b shows the service units of the entire cloud service and how they change after an update. We will discuss the second step of updates in greater detail in the next part.

5.8.2 Hooks

In the second step, we consider service units to be organized into separate trees according to the host-on relationships as shown in the example in Figure 5.10b. All nodes of these trees are on one hand service units, but on the other hand, have also corresponding unit instances in a cloud. When this cloud service is deployed in a multi-cloud environment,



(a) Step 1 - Structure tree



(b) Step 2 - Service unit trees

Figure 5.10: Two-step update of a cloud service

the instances of the service units are created. If a service unit is to be updated all the unit instances need to be in fact re-instantiated.

Thus the problem of updating of a service unit can be reformulated as a new problem of replacing one instance of this node by another instance. In order to be able to do this also with the service units that capture state, such as in DB or file system, we need to provide a generic mechanism to extract the state from the old instance and inject it into the newly created instance. That is why we define 8 hooks as depicted in the Figure 5.11.

A unit instance can be created in two ways: as a first instance of a service unit or through an update of a service unit that already has been instantiated. If it is the first instance, the *Deploy* defined by the cloud service's developer is called and executed. This hook is already used in the current deployment service. In contrast to that, the creation of a new instance through an update is more complex and might require a cooperation of the old instance. When an update is started, first of all, the hook *PreMigrate* is called on the old instance. In this hook, the developer should execute all necessary actions that are needed before the migration from one instance to another. Steps may be taken to preserve the data, that are to be injected into the new instance or the old instance can be removed. After that, the hook *Migrate* of the new version of the service unit is called and thus the new instance is created. *Migrate* might contain the exact actions as the *Deploy* or also additional actions to inject the data from the old instance. At the end, the hook *PostMigrate* is called on the old instance, which must remove all traces of the old instance, if it have not already been done in the *PreMigrate*.

We stress 2 different example scenarios of using *PreMigrate* and *PostMigrate*. In one case when, e.g. moving a DB from one VM to another the old instance of the DB would

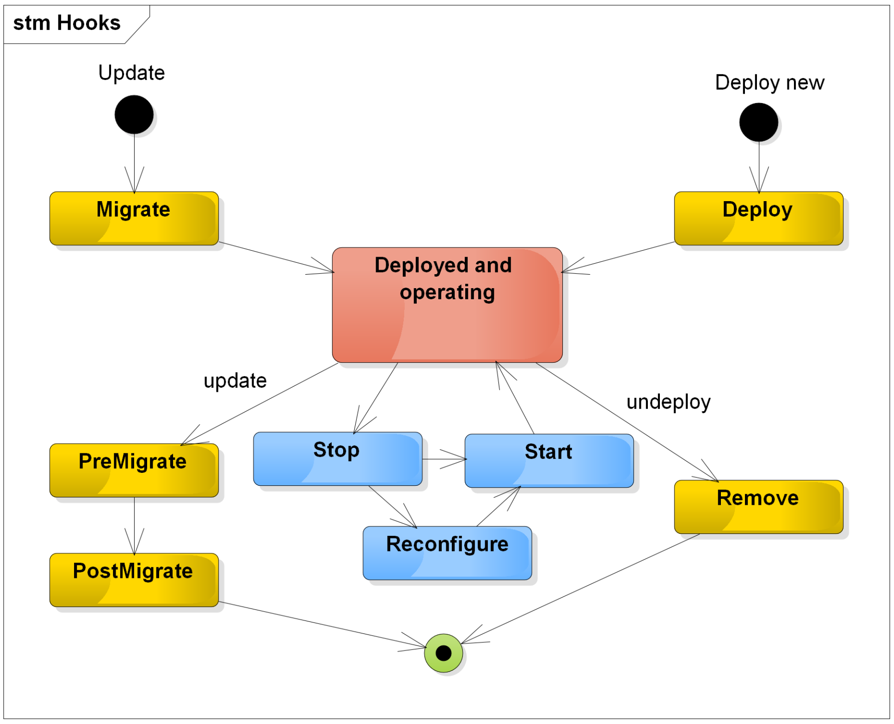


Figure 5.11: Lifecycle of a service unit instance and the 8 hooks

Table 5.9: Overview of hook and their execution during an update of a service unit.

Order	Callback	Instance
1.	Stop	all connected
2.	PreMigrate	old
3.	Migrate	new
4.	PostMigrate	old
5.	Reconfigure	all connected
6.	Start	all connected

be deleted in *PostMigrate* after the migration. In the second scenario, when we want to, e.g. replace an instance of a web server on the same VM, the old instance of the web server would be deleted in *PreMigrate*, before the creation of the new instance, so that we can utilize the same network port.

We have not mentioned the hooks *Stop*, *Reconfigure* and *Start* yet. If the service unit under update, e.g. a load balancer, has another units connected to it through connect-to relationship, the *Stop* is called on the connected units before the *PreMigrate*. After the

PostMigrate, *Reconfigure* and *Start* are called. This way the connected units are aware of the change and can reconnect to the new instance. The execution order of all hooks can be seen in Table 5.9

If an instance is only to be deleted, *Remove* hook is responsible for removal of the instance.

5.8.3 Resolving service unit trees

When an update of a cloud service is triggered, we need to traverse the service unit trees of the new configuration to determine, which sub-trees should be updated and in what order.

At the beginning all service units that need to be re-instantiated are marked. A service unit is marked if:

- is a VM and was moved to a different cloud
- was changed (e.g. a flavor of a VM or the entire OSU was exchanged)
- any unit on the path from this unit to the root was marked or path have changed (e.g. a unit was removed or added)

As a result, if there is a service unit that should be re-instantiated, the entire sub-tree determined by this unit as root is re-instantiated as well.

All VMs that need to be re-instantiated or new VMs are created automatically. The hooks of other nodes are triggered sequentially as follows:

- First, starting from leaf nodes of the sub-trees to the root *PreMigrate* is called.
- Secondly, *Migrate* is called on nodes starting by the root and ending with leaf nodes.
- Finally, *PostMigrate* , the same as *PreMigrate*, is called from leafs to the root.

To make it possible to transfer state from the old instance of a node to the new instance, *Migrate* is always aware of the IP where the old instance is hosted and the *PreMigrate*/*PostMigrate* is always aware of the IP where the new instance will be hosted. This might be in fact the same IP if both instances are/will be hosted on the same VM.

5.8.4 Limitations

In this concept, the atomic unit of structure is the service unit. Therefore, it is not possible to split one service unit and its state to two new service units during an update. Similarly, it is not possible to merge states of two units into one new unit.

Last but not least, it is cloud service developer's responsibility to design the architecture of his/her cloud service in a way that it is well maintainable and upgradeable in the future.

CHAPTER 6

Implementation and Evaluation

The implemented prototype includes the Lifecycle Manager, the EPS Adapters for SALSA, MELA and rSYBL, User-interaction adapter that publishes a REST API and a web application GUI, the Recorder with two sample analytic operations, first, for retrieval of statistics about deployment times of instances, second, synthesis of rSYBL's Action Plans and real effects. The implementation does not include the proposed Cloud Service Updater component. The Section 6.1 discusses the main technologies used for implementation and the specifics of the implemented EPS Adapters. The Section 6.2 summarizes, how the proposed framework reflects the high-level aims mentioned in the Section 1.3. Finally, we perform experiments and evaluate the features of the management system in the Section 6.3.

6.1 Implementation

The prototype is implemented in Java 1.7, accessible in GitHub [1], packaged as a Maven [2] project.

As the implementation of the MOM, we use the messaging broker **RabbitMQ** [3] compatible with AMQP 0.9.1[4] specification. In comparison to the standard Java messaging API, JMS [5], AMQP 0.9.1, provides more powerful routing mechanism [34]. In addition to Topics and Queues in JMS, AMQP 0.9.1 defines concepts of exchanges and bindings. Bindings between an exchange and a queue, or between two exchanges determine, where is a message routed. We use this to distribute and filter messages from Life-cycle Manager to participants. Moreover, AMQP 0.9.1 is language-agnostic, so it allows implementation of EPS adapters or additional components of the management system in any

[1]http://github.com/tuwiendsg/COMOT/tree/master/comot-manager
[2]http://maven.apache.org
[3]http://www.rabbitmq.com
[4]http://www.amqp.org/specification/0-9-1/amqp-org-download
[5]http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

programming language. The endpoints introduced in the Section 5.2 are represented in the RabbitMQ by the exchanges and routing filters by bindings with binding keys.

For the Lifecycle recorder we use a **Neo4j** [6] graph database. We further utilize the Spring Data Neo4j library [7], partially to manipulate the events, but most importantly to define requirements on the objects that can be versioned in the recorder. The recorder is implemented in a very generic way, so that it can version object of any Java Class that fulfills the requirements given by the Spring Data Neo4j and additionally, having one property annotated with our custom annotation for region-scope specific ID. The underlying mechanism is implemented with help of the Java Reflection API [8].

6.1.1 Adapters

We have implemented a prototype Adapter for each of the current EPSs: SALSA, MELA, rSYBL. They are based on the Generic Adapter described in the Section 5.5, distinguishing only by a different implementation of the *EPS-specific Event Processor*. For each EPS, we describe, what exact conditions triggers, which functionality of the EPS and what Events they produce. The triggering may be influenced by a combination of several factors: the defined routing filter, content of the Event-State message, information from the information model and the state of the Adapter or EPS itself. When an Adapter calls the REST interface, it translates the information required by the EPS from the model of the information service to the EPS-specific input model. For completeness, we repeat that an EPS Adapter receives messages only concerning the cloud service instance that it is assigned to.

Deployment service - SALSA

SALSA adapter produces following Custom events: STAGING, CONFIGURING, INSTALLING and STAGING_ACTION. The first three correspond to Salsa-specific states during deployment, STAGING_ACTION represents a Salsa-specific action that may occur independently. The Custom events, by the definition, are not mapped to any particular State in the life-cycle.

Assuming control for deployment

Trigger	**Event-State:** `START` targeted at level SERVICE **EPS State:** The service instance not managed yet.
Reaction	Attempt to take control by sending `DEPLOYMENT_STARTED`

[6]http://neo4j.com
[7]http://projects.spring.io/spring-data-neo4j
[8]http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html

Assuming control for undeployment

Trigger	**Event-State:** `STOP` targeted at level SERVICE **EPS State:** The service instance is managed.
Trigger	**Event-State:** Lifecycle event `KILL` targeted at level SERVICE **EPS State:** The service instance is managed.
Reaction	Attempt to take control by sending `UNDEPLOYMENT_STARTED`

Deploying

Trigger	**Event-State:** `DEPLOYMENT_STARTED` targeted at level SERVICE. Source-Participant-Id equal to this participant.
Reaction	Deploy service instance. Start new thread to periodically check the state of the service instance. If a new UnitInstance is created send `DEPLOYMENT_STARTED` targeted at level INSTANCE, if a Salsa-state of a UnitInstance changes, send a Custom event. If the Salsa-state is error send `ERROR`, if deployment finished, send `DEPLOYED`

Undeploying

Trigger	**Event-State:** `UNDEPLOYMENT_STARTED` targeted at level SERVICE.
Reaction	Undeploy and send `UNDEPLOYED` targeted at level SERVICE.

Undeploying due termination of support

Trigger	**Event-State:** `EPS_SUPPORT_REMOVED` targeted at level SERVICE. Source-Participant-Id equal to this participant. **EPS State:** The service instance is managed.
Reaction	Send `UNDEPLOYMENT_STARTED` targeted at level SERVICE, undeploy and send `UNDEPLOYED` targeted at level SERVICE.

Start checking state

Trigger	**Event-State:** `ELASTIC_CHANGE_STARTED` **Adapter-State:** Not checking the state for this service instance yet.
Reaction	Start checking the state of the service instance periodically, if a change of Salsa-state of a UnitInstance is detected, send `DEPLOYMENT_STARTED`, `UNDEPLOYMENT_STARTED` or a Custom event respectively, targeted at level INSTANCE.

Stop checking state

Trigger	**Event-State:** `ELASTIC_CHANGE_FINISHED` and no Group is in the state ELASTIC_CHANGE.
Reaction	Stop checking the state of the service instance.

Table 6.1: Mapping of conditions in the life-cycle to operations of SALSA.

Control service - rSYBL

The services handled by rSYBL can be in 2 states: managed and controlled. Managed service is registered by submitting a service description and a deployment description, but is not yet actively controlled. Controlled means that the elasticity controller monitors the service and may modify its runtime. Because rSYBL does not allow to distinguish between these two states through the REST API, we implement the adapter as state-full. The adapter recognizes 3 internal states possible for each service: controlled, managed, none. It always holds that if a service is controlled, then it is also managed.

Beside the REST API, the adapter communicates with rSYBL through a JMS message queue. The messages are produced by rSYBL, published in the queue and consumed by the adapter. There are several types of messages describing stages of elastic actions and other notable events. These are all propagated to the management system as Custom events with the content marshaled as JSON in the field *Custom-Message*. The adapter distinguishes ActionPlanEvents, which are produced before the start of an elastic change and after the change is finished. These are additionally mapped also to the Life-cycle events `ELASTIC_CHANGE_STARTED` and `ELASTIC_CHANGE_FINISHED` targeted at SERVICE level.

Start - automatically

Trigger	**Event-State:** State at level SERVICE changed from DEPLOYING to RUNNING.

Start - by assignment

Trigger	**Event-State:** Custom event EPS_SUPPORT_ASSIGNED and state at SERVICE level RUNNING.

Start - manual

Trigger	**Event-State:** Custom event RSYBL_START and state at SERVICE level RUNNING.
Reaction	Start management and control.

Stop - after kill

Trigger	**Event-State:** Lifecycle event KILL and state at SERVICE level **Adapter-State:** Controlled or managed.

Stop - by removal of assignment

Trigger	**Event-State:** Custom event EPS_SUPPORT_REMOVED **Adapter-State:** Controlled or managed.
Reaction	Stop management and control.

Stop - preemptive

Trigger	Lifecycle event `STOP_CONTROLLER`
Reaction	Stop control if controlled. Send `STOP_CONTROLLER` to acknowledge that rSYBL will not attempt to execute an Action Plan.

Stop - manual

Trigger	Custom event RSYBL_STOP **Adapter-State:** Controlled.
Reaction	Stop control.

Set Metric Composition Rules

Trigger	**Event-State:** Custom event SET_MCR **EPS State:** Is monitored.
Reaction	Set Metric Composition Rules.

Table 6.2: Mapping of conditions in the life-cycle to operations of rSYBL.

Monitoring service

The adapter for MELA recognizes additional Custom event MELA_START, MELA_STOP and SET_MCR. It refreshes the deployment information after each change in the cloud service.

Start - automatically

Trigger	**Event-State:** Transition at SERVICE level from the state DEPLOYING to RUNNING.

Start - by assignment

Trigger	**Event-State:** Custom event EPS_SUPPORT_ASSIGNED and state at SERVICE level RUNNING. **EPS State:** Not monitored yet.

Start - manual

Trigger	**Event-State:** Custom event MELA_START and state at SERVICE level RUNNING. **EPS State:** Not monitored yet.
Reaction	Start monitoring.

Stop - automatically

Trigger	**Event-State:** Transition at SERVICE level from the state RUNNING to UNDEPLOYING. **EPS State:** Is monitored.

Stop - by removal of assignment

Trigger	**Event-State:** Custom event EPS_SUPPORT_REMOVED **EPS State:** Is monitored.

Stop - after kill

Trigger	**Event-State:** Lifecycle event `KILL` **EPS State:** Is monitored.

Stop - manual

Trigger	**Event-State:** Custom event MELA_STOP **EPS State:** Is monitored.
Reaction	Stop monitoring.

Refresh monitored elements

Trigger	**Event-State:** Event `ELASTIC_CHANGE_FINISHED` or `MAINTENANCE_FINISHED` at any level. **EPS State:** Is monitored.
Reaction	Update monitored elements.

Set Metric Composition Rules

Trigger	**Event-State:** Custom event SET_MCR **EPS State:** Is monitored.

Reaction	Set Metric Composition Rules.

Table 6.3: Mapping of conditions in the life-cycle to operations of MELA.

Custom Elastic Platform Services

Although there are implementation of the AMQP 0.9.1 client in several programming languages, so the adapters for EPSs can be created in any of them, we provide an interface for easy creation of Adpters in Java. The `IProcessor` corresponds to the interface of a *EPS-specific Event Processor* from the Generic Adapter.

```
public interface IProcessor {

  public void setDispatcher(IDispatcher dispatcher);

  public List<Binding> getBindings(String queueName, String serviceId);

  public void start(String participantId) throws Exception;

  public void onLifecycleEvent(StateMessage msg) throws Exception;

  public void onCustomEvent(StateMessage msg) throws Exception;

  public void onExceptionEvent(ExceptionMessage msg) throws Exception;
}

public interface IDispatcher {

  public void sendLifeCycle(Type targetLevel, LifeCycleEvent event)
      throws AmqpException, JAXBException;

  public void sendCustom(Type targetLevel, CustomEvent event)
      throws AmqpException, JAXBException;

  public void sendException(ExceptionMessage message)
      throws AmqpException, JAXBException;
}
```

Listing 6.1: Interface for a custom *EPS-specific Event Processor*

6.2 Requirements and Design Evaluation

In this section we discuss, how the proposed management system framework fulfills the requirements introduced in the Section 1.3.

- *Development and Operation* - The roles of participants described in the Section 4.1.3 and the lifecycle resulting from this roles, contains states and events needed for development, testing as well as operation and maintenance.

- *Coordination of EPSs* - The coordination is not based on direct invocations, but on an active involvement of all participants. The states and Lifecycle events create a predefined frame for coordination of EPSs. The consistency is ensured by the Lifecycle manager that processes the events sequentially in the scope of one cloud service. This guarantees that EPSs, which modify cloud services can not interfere with each others actions. All EPSs are able to observe the changing cloud service and decide on their actions.

- *Information model* - The management system is compatible with the information model and designed to utilize the future information service. Moreover, we utilized the concept of Offered Service Unit for representation of EPSs in the information model.

- *Extensibility* - The frame determined by the states and Lifecycle events allows EPSs that need to change the cloud service, to define their own Custom events within this states. All EPSs have access to the information service and events in the management system, which means any EPSs can be added or remover from the environment at any time.

- *Traceability* - Every event contains Timestamp, Source-Participant-Id and data to identify the purpose and target of the event. By storing these events and the changes in the description and state of the cloud service, it is possible to trace the activities within the management system.

- *Structure of service* - Events can be targeted to the entire service or to individual topologies, units or unit instances. The events are evaluated on the tree structure of these entities in the Lifecycle Manager.

- *Offered as service* - The User-interaction adapter provides a REST interface for interaction with external systems.

6.3 Functional Evaluation

The set-up for testing and evaluation is depicted in the Figure 6.1. The application logic of the management system is running on a computer in one process. The message

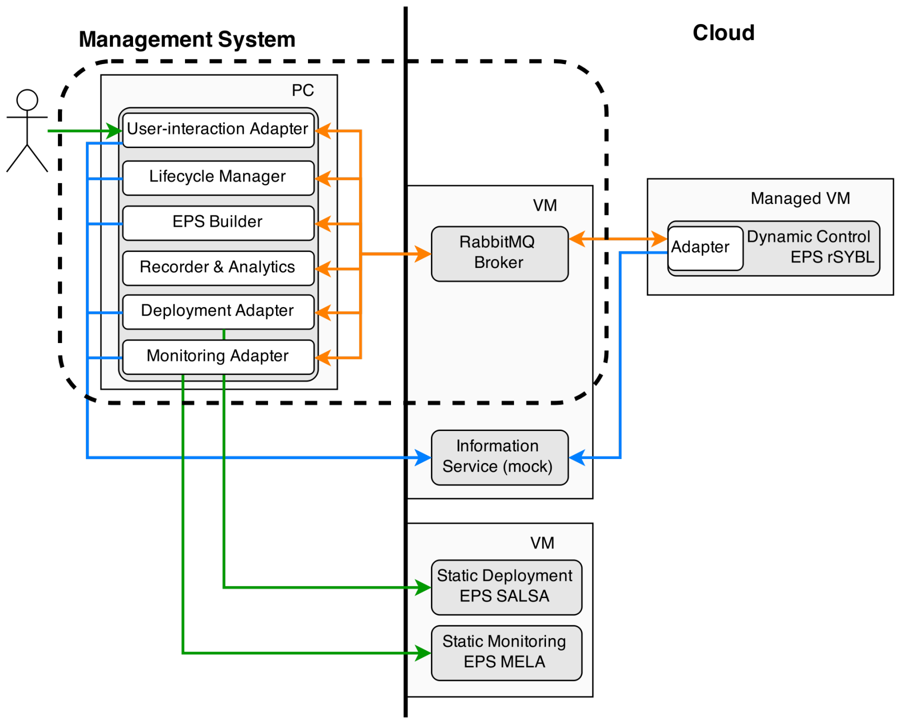


Figure 6.1: Environment for the evaluation of the Management system

broker RabbitMQ is located in the cloud (OpenStack[9]) on a virtual machine (VM) with a public IP, so that is is accessible from the cloud as well as from the Internet. Also the Information iervice must be accessible by each participant from the Internet and from the cloud. For illustration, we show the deployment service SALSA and monitoring service SALSA as static EPSs and the controller service rSYBL as a dynamic EPS.

However, this configuration does not represent neither the most compact, nor most distributed of the possible configurations. Any configuration is acceptable, as long as every component of the Management system and every Adapter can access the Broker and the Information service. The most compact configuration would be running all components, except the dynamic EPSs on one machine (either locally or in cloud). In case of the most distributed configuration, every service could be hosted on a different machine.

For evaluation we use an elastic cloud service, which processes data from sensor. It is composed of a Load Balancer unit and an Event Processing unit, each of them hosted on a matching operating system (OS) unit. The Load Balancer unit consists of only one instance and distributes the HTTP requests from the sensors among the instances of the Event Processing unit. The Figure 6.2 shows a model of such service. The arrows represent *connect to* relationships, the diamonds a composition through *host on* relationship. The relationships are between units as well as between instances of the

[9]http://openstack.infosys.tuwien.ac.at/horizon/nova/

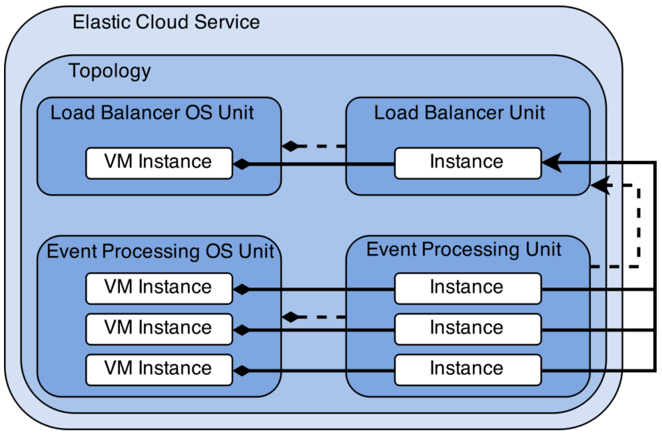


Figure 6.2: Structure of the elastic cloud service used for evaluation.

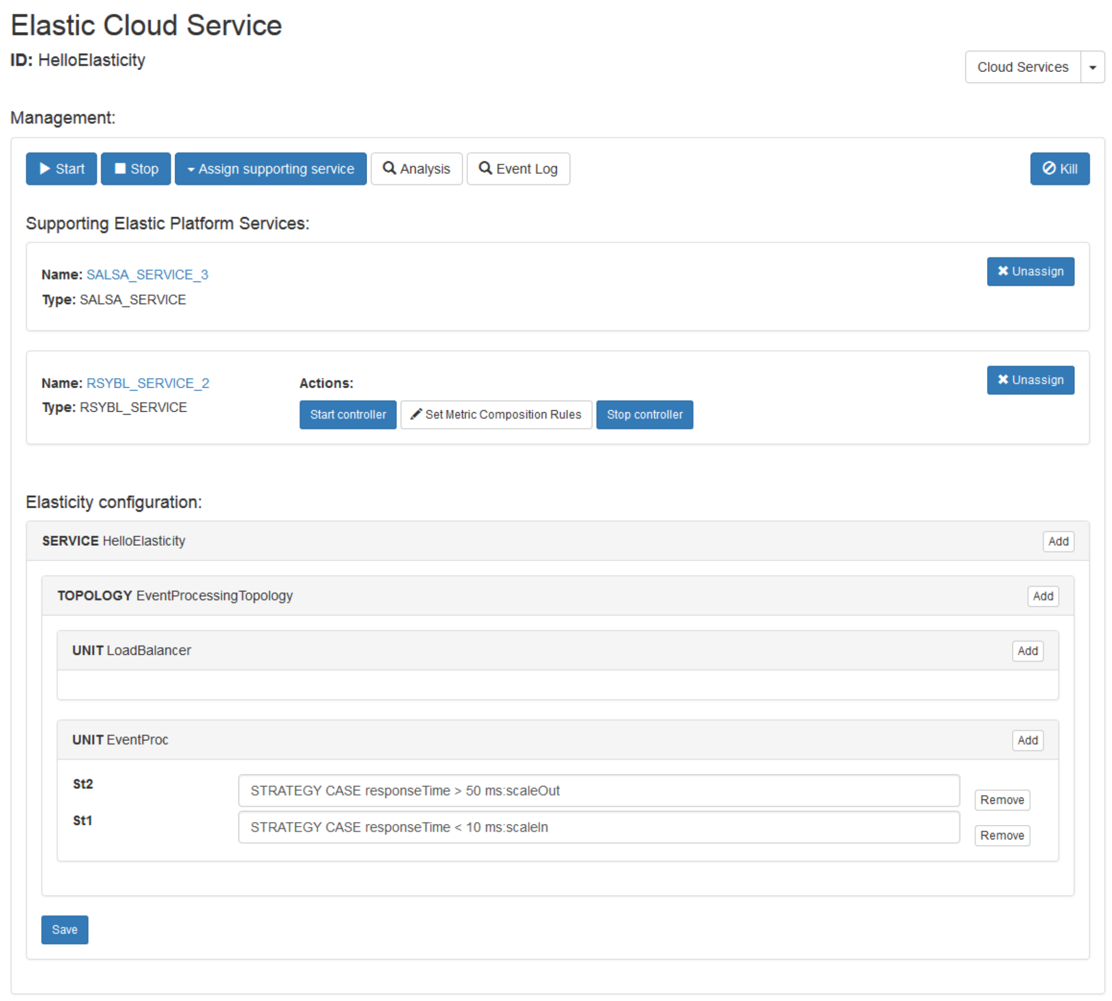
units. Overall, it is a representative elastic cloud service of a sufficient complexity as it contains a scaleable service unit, *host on* and *connect to* relationships between service units.

Cloud Service Management

This scenario is fully supported by the provided GUI, which utilizes the REST interface of the User-interaction adapter. Part of the management GUI can be seen in the Figures 6.3 and 6.4. We begin with creation of a new cloud service in the Management system, through submission of cloud service's description as a TOSCA XML. The TOSCA contains not only information needed for deployment, but also SYBL directives, which define the elastic behavior. The User-interaction adapter translates the TOSCA model into the information model and triggers the `CREATED` event.

We request support of the static EPSs: deployment service SALSA and elasticity controller service rSYBL. That means, that for each EPS a `EPS_SUPPORT_REQUESTED` is triggered. Each of the EPS Adapters initiates support by creating a queue dedicated to the events about our test elastic cloud service and confirm the support by an `EPS_SUPPORT_ASSIGNED` event. Furthermore, we provide Metric Composition Rules for the controller service, by sending a custom event `SET_MCR`.

The service is started by a `STARTED` event. This causes the deployment adapter to take the control by sending `DEPLOYMENT_STARTED` and when it receives the event, it knows that it can start with the deployement procedure. For every new unit instance it sends a `DEPLOYMENT_STARTED` targeted at INSTANCE level also with its description. For OS unit instances it emits custom events `CONFIGURING` for the Load Balancer unit and Event Processing unit events `STAGING`, `CONFIGURING` and `INSTALLING`. The final event for every instance is `DEPLOYED`. After the last instance is deployed, the state of entire cloud service changes to RUNNING. At this point the controller EPS is configured automatically through its adapters.


Elastic Cloud Service

ID: HelloElasticity

Management:

Supporting Elastic Platform Services:

Name: SALSA_SERVICE_3
Type: SALSA_SERVICE

Name: RSYBL_SERVICE_2
Type: RSYBL_SERVICE
Actions:

Elasticity configuration:

SERVICE HelloElasticity

TOPOLOGY EventProcessingTopology

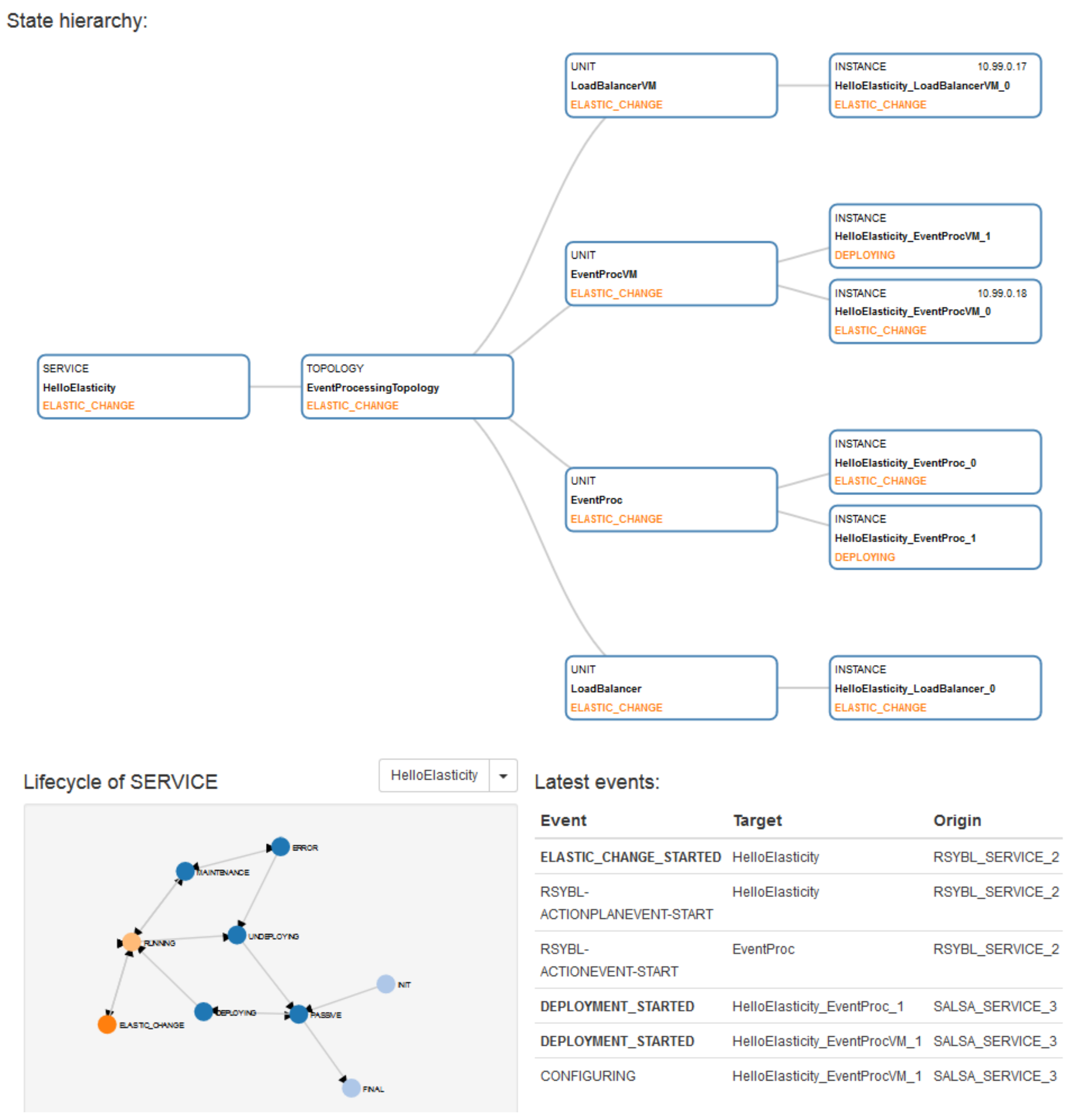
UNIT LoadBalancer

UNIT EventProc

St2	STRATEGY CASE responseTime > 50 ms:scaleOut
St1	STRATEGY CASE responseTime < 10 ms:scaleIn

Figure 6.3: Management view for the example elastic cloud service with assigned rSYBL and SALSA EPSs.

We simulate sensor's requests by generating a load. The load is increased and decreased in periodic intervals, so that we ensure scale in/out actions of the controller. After an hour, we reconfigure elasticity with new SYBL directives, sending a `RECONFIGURE_ELASTICITY` event. This updates the elasticity controller as well as the information service. The services continues operating for another hour. We stop the service by the `STOP` event, which causes the Lifecycle manager to request rSYBL to terminate the active elasticity control by sending the `STOP_CONTROLLER` event. Once rSYBL terminates the controlling, it responds with the `STOP_CONTROLLER` even, what Lifecycle manager translates into `STOP` and propagates. The deployment adapter assumes control by sending the `UNDEPLOYMENT_STARTED` event, followed by `UNDEPLOYED` event, when undeployemnt finishes. Finally we, remove the cloud service


State hierarchy:
SERVICE HelloElasticity ELASTIC_CHANGE
TOPOLOGY EventProcessingTopology ELASTIC_CHANGE
UNIT LoadBalancerVM ELASTIC_CHANGE
INSTANCE 10.99.0.17 HelloElasticity_LoadBalancerVM_0 ELASTIC_CHANGE
UNIT EventProcVM ELASTIC_CHANGE
INSTANCE HelloElasticity_EventProcVM_1 DEPLOYING
INSTANCE 10.99.0.18 HelloElasticity_EventProcVM_0 ELASTIC_CHANGE
UNIT EventProc ELASTIC_CHANGE
INSTANCE HelloElasticity_EventProc_0 ELASTIC_CHANGE
INSTANCE HelloElasticity_EventProc_1 DEPLOYING
UNIT LoadBalancer ELASTIC_CHANGE
INSTANCE HelloElasticity_LoadBalancer_0 ELASTIC_CHANGE
Lifecycle of SERVICE
HelloElasticity
ERROR
MAINTENANCE
RUNNING
UNDEPLOYING
INIT
DEPLOYING
PASSIVE
ELASTIC_CHANGE
FINAL
Latest events:

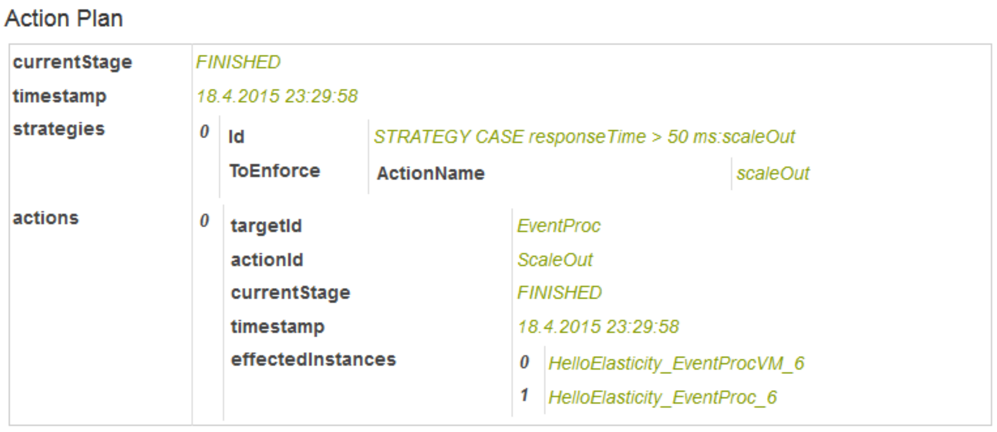
Event	Target	Origin
ELASTIC_CHANGE_STARTED	HelloElasticity	RSYBL_SERVICE_2
RSYBL-ACTIONPLANEVENT-START	HelloElasticity	RSYBL_SERVICE_2
RSYBL-ACTIONEVENT-START	EventProc	RSYBL_SERVICE_2
DEPLOYMENT_STARTED	HelloElasticity_EventProc_1	SALSA_SERVICE_3
DEPLOYMENT_STARTED	HelloElasticity_EventProcVM_1	SALSA_SERVICE_3
CONFIGURING	HelloElasticity_EventProcVM_1	SALSA_SERVICE_3

Figure 6.4: Visualization of the current states of individual groups, model of lifecycle for each Group at different level and overview of the latest events.

by the event `REMOVED`, this results in removal of the service from the Lifecycle manager, rSYBL adapter and information service.

Service History and Auditing

The previous scenario generated 223 events stored in the Recorder, which we analyze to learn about details of the cloud services execution. First, we compute duration of


Action Plan

currentStage	FINISHED			
timestamp	18.4.2015 23:29:58			
strategies	0	Id	STRATEGY CASE responseTime > 50 ms:scaleOut	
		ToEnforce	ActionName	scaleOut
actions	0	targetId	EventProc	
		actionId	ScaleOut	
		currentStage	FINISHED	
		timestamp	18.4.2015 23:29:58	
		effectedInstances	0	HelloElasticity_EventProcVM_6
			1	HelloElasticity_EventProc_6

Figure 6.5: Report of an Action Plan executed by rSYBL and its effects.

each stage during deployment of the 26 UnitInstances, using the *Timestamp* property. The results can be seen in the Figure 6.6. The stages and duration are computed by filtering all events between events UNDEPLOYMENT_STARTED and DEPLOYED that have the *Source-Participant-Id* of an EPS type SALSA. The average overall deployment time of the Event Processing Unit is 366 seconds, but 266 seconds is waiting for the virtual machine unit to allocate and configure.

The response time of the operation, which computes this data is 1 minute 3 seconds, what is very high considering relatively small amount of instances. This is caused by an algorithm not optimized for performance, which includes traversal of the entire sequence of events for each individual instance. However, a more sophisticated mechanism for storing of event could be also considered in a future version of the prototype.

Next, we analyze the events to reconstruct rSYBL's elastic actions and their effects. Each Action Plan is enclosed by starting and final event. We extract the ID of effective directive from the Action Plan event and search the database for the state of the directive valid at the time of the event. From events UNDEPLOYED and DEPLOYED we discover, which instances were effected by the action. An example of one such Action Plan from the experiment can be seen in the Figure 6.5.

Error and Exception Handling

We simulate an error by faulty definition of a cloud service, concretely specifying an unexisting URL on an artifact. During deployment of the service, SALSA fails to retrieve the artifact and indicates through the adapter, that an ERROR occurred. Since there is currently no component capable of the selective update and redeployment at runtime, we can only use the KILL event to command an undeployment of the entire cloud service.

To demonstrate the handling of exceptions, we start the cloud service with assigned SALSA. Once in state RUNNING we start to generate the load. rSYBL triggers ELASTIC_CHANGE_STARTED and continues with execution of an elastic action. At this

point the standard undeployment is not possible, thus the Lifecycle manager does not propagate the event, instead publishing an ExceptionMessage. We retrieve the event from the Recorder and learn the reason, why the event was not allowed:
Action 'STOP' is not allowed in state ELASTIC_CHANGE. Group HelloElasticity
ROOT: Action STOP is not allowed in state ELASTIC_CHANGE patentState ELASTIC_CHANGE. GroupId=HelloElasticity_LoadBalancerVM_0 , type=INSTANCE

This feature may be especially important for the provider of the platform, as he may track also, e.g. Java exceptions thrown by the EPSs.

Multiple Cloud Services and Multiple Dynamic EPSs

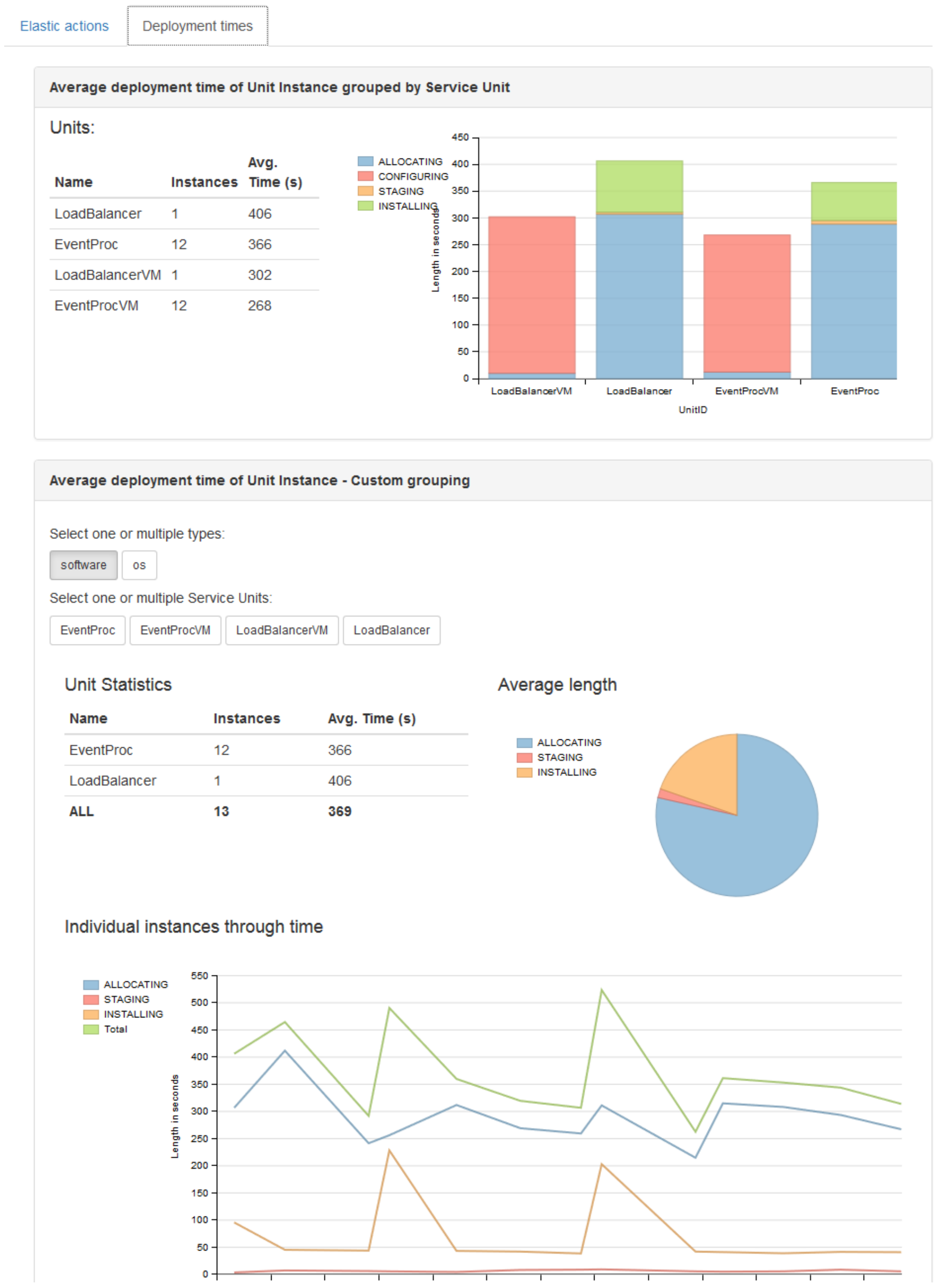
In the last scenario, we perform parallel management of multiple cloud services. We compare a case, when all cloud services are supported by one EPS SALSA, versus a case, when each cloud service has its private dynamic EPS SALSA at disposal.

We start the cloud services sequentially with 30 seconds delay, because OpenStack fails to properly allocate multiple VMs, if they are requested simultaneously. In case of the tests with dynamic EPS, we first set up all the dynamic EPSs and start test afterwards. We run the test only with 5 cloud services, because OpenStack fails to allocate VMs for more services even with the 30 seconds delay. Any longer delay would basically mean that the EPS does not actively deal with more cloud services at the same time.

We measured the duration of STAGING phase of deployment and time from event's creation, till its storage in the Recorder. The average time of the STAGING phase is 6,5 seconds, the average travel-time of an event is in both cases around 0,8 seconds. The performance differences between the two setups are negligible.

We have shown that it is possible to manage multiple cloud services and utilize the dynamic EPS, but the performance limits are set by the underlying cloud, not the EPS or the management system itself. Because allocation of VMs is extremely time consuming, measured in minutes, not seconds or milliseconds, it is almost irrelevant, how performant are the supporting systems.

Furthermore, it is currently not possible to combine dynamic rSYBL with dynamic SALSA, because rSYBL communicates in the background with SALSA, thus it may happen that one rSYBL would be required to discover a dynamic SALSA at runtime, which is currently not possible. Generally, the framework does not handle dependencies between EPS. The dynamic rSYBL can be currently used only if the cloud service is supported by a static SALSA.



Average deployment time of Unit Instance grouped by Service Unit

Units:

Name	Instances	Avg. Time (s)
LoadBalancer	1	406
EventProc	12	366
LoadBalancerVM	1	302
EventProcVM	12	268

Average deployment time of Unit Instance - Custom grouping

Select one or multiple types:

Select one or multiple Service Units:

Unit Statistics

Name	Instances	Avg. Time (s)
EventProc	12	366
LoadBalancer	1	406
ALL	**13**	**369**

Average length

Individual instances through time

Figure 6.6: Statistics about times of deployment stages of Unit Instances.

CHAPTER 7

Conclusions and Future Work

7.1 Conclusion

This thesis proposed a framework for a management system of elastic cloud services in the specialized platform composed of EPSs. This management system utilizes the model of the introduced elastic cloud service's lifecycle to coordinate the EPSs, thus significantly automating the management of elastic cloud services. The formulated lifecycle covers all stages from development, deployment, operation, maintenance and the stage of elasticity's change. The management system further adopts the principles of event-driven architecture to decouple the introduced internal components of the system as well as the EPSs. This ensures extensibility and creates environment that is highly observable and traceable.

Furthermore, the thesis presents the idea of managing the dynamic EPSs as elastic cloud services. This has the advantage of utilizing the management system's tools also for the management of some of its own parts. Such possibility is especially interesting for the provider of the platform, who can easily manage and maintain the offered EPSs.

7.2 Future Work

Because currently there is no full implementation of the information service available, only the mock service developed for the purpose of this thesis, it is inevitable to integrate the management system with the information service, once a prototype exists.

The thesis includes design and prototype for recording and analysis of events exchanged in the management system and changes of the cloud service's description. It is necessary to further research the behavior of elastic cloud services as well as the key challenges that the service developers face, when manipulating the elastic cloud service throughout their lifecycle. Future analytic tools, building on the recorder's capabilities, should support richer developer-oriented scenarios.

The architecture of the management system creates optimal conditions for testing of elasticity and comparison of different configurations. The future testing EPS could utilize the events for observation and control of testing environment as well as the recorded data for the evaluation of tests.

Although the management system acknowledges the maintenance stage of the lifecycle, it remains for future research, to develop a component that would operate in this stage. We proposed a concept of an Updater, that could be implemented as either stand-alone component or an extension of the deployemnt EPS.

List of Acronyms

EPS Elastic Platform Service. 1–7, 9, 11–13, 16, 17, 19, 20, 25, 27, 35, 36, 39–42, 51–54, 56, 65, 66, 71–76, 78, 81, 82

IaaS Infrastructure as a Service. 1–3

MOM Message-Oriented Middleware. 42, 65

OSU Offered Service Unit. 5, 14, 16, 19, 20, 35, 36, 51

PaaS Platform as a Service. 1–3, 5, 9

SOA Service Oriented Architecture. 11, 12

Bibliography

[1] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[2] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. Ieee, 2010.

[3] S. Dustdar, Yike Guo, B. Satzger, and Hong-Linh Truong. Principles of elastic processes. *Internet Computing, IEEE*, 15(5):66–71, Sept 2011.

[4] Hong-Linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc-Hung Le, and Daniel Moldovan. CoMoT - A Platform-as-a-Service for Elasticity in the Cloud. In *IEEE International Workshop on the Future of PaaS*, 2014.

[5] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Multi-level elasticity control of cloud services. In *Service-Oriented Computing*, pages 429–436. Springer, 2013.

[6] Victor Ion Munteanu, T Fortis, and Viorel Negru. Service lifecycle in the cloud environment. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 457–464. IEEE, 2012.

[7] Terence Harmer, Peter Wright, Christina Cunningham, John Hawkins, and Ron Perrott. An application-centric model for cloud management. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 439–446. IEEE, 2010.

[8] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533. ACM, 2011.

[9] S. Rajbhandari and D.W. Walker. Incorporating provenance in service oriented architecture. In *Next Generation Web Services Practices, 2006. NWeSP 2006. International Conference on*, pages 33–40, Sept 2006.

[10] Duc-Hung Le, Hong-Linh Truong, Georgiana Copil, Stefan Nastic, and Schahram Dustdar. Salsa: A framework for dynamic configuration of cloud services. In *6th International Conference on Cloud Computing Technology and Science*, 2014.

[11] Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Mela: Monitoring and analyzing elasticity of cloud services. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 80–87. IEEE, 2013.

[12] Sajee Mathew. Overview of amazon web services. *Amazon Whitepapers*, 2014.

[13] Peter Dalbhanjan. Overview of deployment options on aws. *Amazon Whitepapers*, 2015.

[14] Amazon Web Services Inc. AWS Elastic Beanstalk. `http://aws.amazon.com/elasticbeanstalk`. Accessed: Februar 2014.

[15] Amazon Web Services Inc. AWS OpsWorks. `http://aws.amazon.com/opsworks`. Accessed: Februar 2014.

[16] Google. Google App Engine: Platform as a Service. `https://cloud.google.com/appengine/docs`. Accessed: Februar 2014.

[17] OASIS. Topology and orchestration specification for cloud applications version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html`, 2013.

[18] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. Opentosca–a runtime for tosca-based cloud applications. In *Service-Oriented Computing*, pages 692–695. Springer, 2013.

[19] GigaSpaces Technologies. Cloud Orchestration and Automation Made Easy. `http://getcloudify.org`. Accessed: Februar 2014.

[20] Cristian Ruz, Françoise Baude, Bastien Sauvan, Adrian Mos, and Alain Boulze. Flexible soa lifecycle on the cloud using sca. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International*, pages 275–282. IEEE, 2011.

[21] Georgiana Copil, Daniel Moldovan, Hong Linh Truong, and Schahram Dustdar. Sybl: An extensible language for controlling elasticity in cloud applications. In *CCGRID*, pages 112–119. IEEE Computer Society, 2013.

[22] C Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, Rebekah Metz, and Booz Allen Hamilton. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.

[23] Patrick Finger and Klaus Zeppenfeld. *SOA und WebServices*. Springer-Verlag, 2009.

[24] Liam O'Brien, Paulo Merson, and Len Bass. Quality attributes for service-oriented architectures. In *Proceedings of the international Workshop on Systems Development in SOA Environments*, page 3. IEEE Computer Society, 2007.

[25] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.

[26] Zakir Laliwala and Sanjay Chaudhary. Event-driven service-oriented architecture. In *Service Systems and Service Management, 2008 International Conference on*, pages 1–6. IEEE, 2008.

[27] O. Levina and V. Stantchev. Realizing event-driven soa. In *Internet and Web Applications and Services, 2009. ICIW '09. Fourth International Conference on*, pages 37–42, May 2009.

[28] Eva Kühn, Richard Mordinyi, László Keszthelyi, and Christian Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '09, pages 625–632, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[29] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[30] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional, 1999.

[31] Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Quelle–a framework for accelerating the development of elastic systems. In *Service-Oriented and Cloud Computing*, pages 93–107. Springer, 2014.

[32] Arnaud Castelltort and Anne Laurent. Representing history in graph-oriented nosql databases: A versioning system. In *Digital Information Management (ICDIM), 2013 Eighth International Conference on*, pages 228–234. IEEE, 2013.

[33] Ian Robinson. Time-based versioned graphs. `http://iansrobinson.com/2014/05/13/time-based-versioned-graphs`. Accessed: December 2014.

[34] Mark Richards. Understanding the differences between amqp & jms. *NFJS Magazine*, 2011.