D3.4.1	
Version	1.0
Author	TUB
Dissemination	PU
Date	27/01/2015
Status	Final

D3.4.1 Elastic Media Manager v1

Project acronym:	NUBOMEDIA
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2016-09-30
Project type:	STREP
Project reference:	610576
Project web page:	http://www.nubomedia.eu
Work package	WP3 Cloud Platform
WP leader	Giuseppe Carella
Deliverable nature:	Prototype
Lead editor:	Giuseppe Carella
Planned delivery date	01/2015
Actual delivery date	27/01/2015
Keywords	Elasticity, Cloud Computing, Management and Orchestration

The research leading to these results has been funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement nº 610576

FP7 ICT-2013.1.6. Connected and Social Media

Contributors:

Giuseppe Carella (TUB)
Lorenzo Tomasini (TUB)

Internal Reviewer(s):

Luis Lopez (URJC)

Version History

Version	Date	Authors	Comments
0.1	06.2014	Giuseppe Carella, Lorenzo Tomasini	Initial version
0.2	08.2014	Giuseppe Carella, Lorenzo Tomasini	Added State of the Art in cloud orchestration
0.3	09.2014	Giuseppe Carella	Initial version of the Elastic Media Manager architecture
0.4	10.2014	Lorenzo Tomasini	Class diagrams of R3 implementation
0.5	12.2014	Giuseppe Carella	Final version of the document ready for the internal review
0.6	12.2014	Lorenzo Tomasini	Final version of the document ready for the internal review
1.0	01.2015	Giuseppe Carella, Lorenzo Tomasini	Fixed some sections based on feedbacks given by the internal reviewer

Table of contents

List of Figures:

Acronyms and abbreviations:

API	Application Programming Interface
CM	Connectivity Manager
CPI	Cloud Provider Interface
EMM	Elastic Media Manager
HOT	Heat Orchestartion Template
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
QoS	Quality of Service
REST	Representational State Transfer
SDN	Software Defined Network
SLA	Service Level Agreement
VM	Virtual Machine

1 Executive summary

This document provides an overview of the Elastic Media Manager (EMM) system. In particular this deliverable focuses primarily on its software architecture, providing a detailed description of which functionalities have been implemented in those initial two releases.

2 Introduction

The Elastic Media Manager (EMM) is the component part of the cloud infrastructure that manages the lifecycle of the NUBOMEDIA Applications, called also the NUBOMEDIA platform. The EMM offers a northbound API to the administrator of the NUBOMEDIA platform and interfaces with different cloud elements for requesting virtual resources.
In particular the EMM executes different phases of the service lifecycle. Those phases are:

1. Deployment
2. Provisioning
3. Runtime management

During the deployment phase, the Elastic Media Manager (EMM) has to create the Virtual Resources on the NUBOMEDIA Virtualized Infrastructure, as specified in deliverable D3.1.1 on top of which the NUBOMEDIA platform will be instantiated. In order to reduce the time needed to start a new set of components, the images used are already containing installed software which only needs to be configured before being executed.

Due to the dynamicity of the cloud environment, each time can be assigned a different network configuration to a new virtual resource. Therefore in order to properly execute the NUBOMEDIA platform components, it is required to dynamically configure them. The provisioning phase is the one in which the EMM configures the software components running on the Virtual Resources.

When all the Virtual Resources are deployed and all the software is up and running, the EMM has to manage the runtime phase. Typically runtime several situations can occur: users sending update requests, crashing of hardware components, overload of system components. Therefore there are many aspects which need to be taken into account while executing this phase. In the scope of this project we are mainly addressing the autoscaling problem. Autoscaling means that the system actively reacts based on rules assigned by the Administrator. A typical example is the instantiation or removal of new instances of a specific component when the CPU load is over/under a specified threshold. This scenario is useful for saving Virtual Resources, which are not all the time required, and to cope with peak loads moments that may occur in specific day hours.

Summarizing, the EMM provides the users a management tool through which it is possible to deploy a NUBOMEDIA platform inside an already existing NUBOMEDIA Virtualized Infrastructure. Next sections describe how this has been realized, and especially which technology has been selected for satisfying those required functionalities.

2.1 State of the Art

First of all it is given a brief introduction of existing technologies for the management and orchestration of cloud applications.

2.1.1 Cloudify

Cloudify [3] is an enterprise-class open source PaaS stack that lays between your application and your chosen cloud. Thanks to it your application is able to focus on doing what it does best, it will be the task of Cloudify to manage the resources it needs and to make sure that they are available independently of which cloud and stack will be employed.

Cloudify supports public clouds (Amazon EC2, Windows Azure, Rackspace, etc.) and private clouds (OpenStack, CloudStack, VMWare vCloud, Citrix XenServer, etc.).

The Cloudify framework is designed to be flexible enough to handle any application stack. On the other hand, the application is completely isolated from the underlying CPI to support enterprises that want to deploy the same application in multiple environments (for cloud bursting).

If the bottom layers are hidden to you it can happen to loose some degrees of control and in that way you are not able to monitor and ne-tune it as you are used to do with traditional data centers and applications. Cloudify supplies a much greater level of control because it does have access to the infrastructure, but only if you actually want it. Differently, by using a collection of predefined application blueprints, it hides their complexities.

2.1.1.1 How Cloudify works

Cloudify makes use of recipes in order to define an application, its services and their interdependencies, how to monitor, self-heal, and scale its services and their resources. So the process to deploy and manage an application results from:

1. Preparing your deployment
    a. Set up your cloud and describe your machines in the cloud driver
    b. Prepare the binaries required for your services
    c. Describe the application lifecycle and its services in recipes
2. Deploying the services and application
    a. Provisions machines in your cloud using cloud drivers
    b. Downloads, installs, and configures your services
    c. Installs your application
    d. Configures your monitoring and scaling features
3. Monitoring and managing the deployment using the Cloudify web management console or the Cloudify Shell

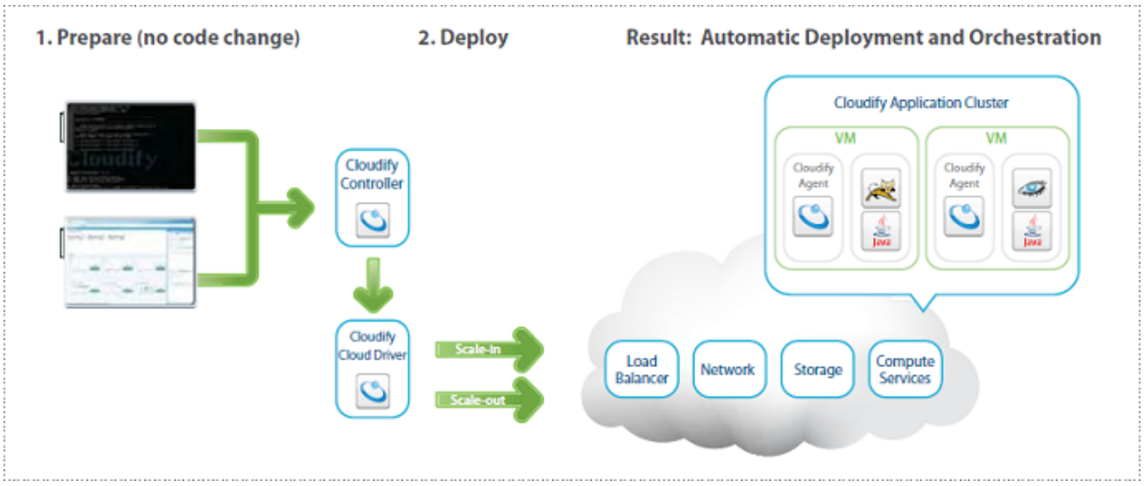


Figure 1. Cloudify

Cloudify recipes are simple Groovy files that describe the execution plans for the lifecycle of the application, practically the steps of installing, starting, orchestrating, and monitoring your application stack.
Moreover the recipes make use of cloud drivers configuration les. These les outline machines and their images specifically for the chosen cloud, for provisioning machines and for your services.

A recipe can look like the following example:

```
service {
      name "mysql"
      icon "mysql.jpg"
      type "DATABASE"
      lifecycle{
            install "mysql_install.groovy"
            start "mysql_start.groovy"
      }
}
application {
      name="petclinic"
      service {
             name = "mysql"
      }
      service {
            name = "tomcat"
            dependsOn = ["mysql"]
      }
}
scalingRule {
      serviceStatistics {
            metric "Requests per second"
            movingTimeRangeInSeconds 20
      }
      highThreshold {
            value 100
            instancesIncrease 1
      }
}
```

2.1.1.2 Cloudify architecture

The architecture of Cloudify is a layered architecture thus to hide as much as possible the low implementation details and to help a simple deployment procedure; it is shown in the following figure:

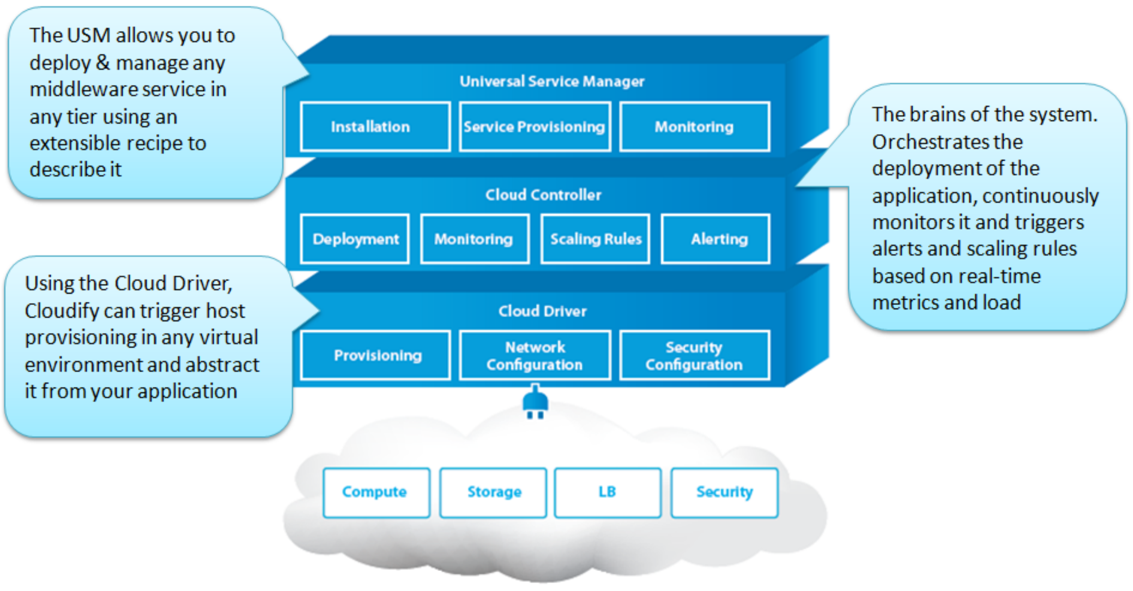

The USM allows you to deploy & manage any middleware service in any tier using an extensible recipe to describe it
Using the Cloud Driver, Cloudify can trigger host provisioning in any virtual environment and abstract it from your application
Universal Service Manager
Installation
Service Provisioning
Monitoring
Cloud Controller
Deployment
Monitoring
Scaling Rules
Alerting
Cloud Driver
Provisioning
Network Configuration
Security Configuration
Compute
Storage
LB
Security
The brains of the system. Orchestrates the deployment of the application, continuously monitors it and triggers alerts and scaling rules based on real-time metrics and load

Figure 2. Cloudfiy architecture

Cloudify's Universal Service Adapters allow the system to obtain of all the achievements that cloudify offers: no code change, no lock-in, and full control. They have the task of translating the raw recipe le into actions such as install, initialize, monitor and so on. They are, in fact, deployed to every Cloudify machine.
The special quality of these adapters is that they are service aware thus they are able to run any different type of service as long as its recipe is properly configured.

It is possible to write recipes that describe all the needed components to run your application and it is Cloudify that will handle all the details of implementing the environment.

2.1.1.3 Cloudify features

Summarizing, the features proposed by Cloudify (in the documentation) are:

- **Any App, Any Stack:** Deploy any middleware stack using a recipe based deployment mechanism
- **Automation of the Entire Lifecycle:** Deploy, manage, and update your application using a single platform
- **Automatic Self-Healing:** Crashed nodes and machines are automatically replaced by new ones, following recipe instructions
- **Any Cloud:** Support all major cloud and virtualization platforms. Your application is completely decoupled from the cloud API
- **Auto-Scale, Your Way:** Automatic scaling of your application services based on out-of-the-box or custom metrics. Scale-in or scale-out
- **Cluster-Aware Monitoring & Performance Management:** Pluggable monitoring, collects tier and application Key Performance Indicators

- **Fully Testable on Your Laptop:** Easily start, debug & test on your laptop with a fully functional cloud emulator. No VMs, no hassles

2.1.2 Heat

The Openstack Heat Orchestrator [2] is a utility able to manage multiple composite cloud applications using templates, over both an OpenStack-native ReST API and a CloudFormation-compatible Query API.

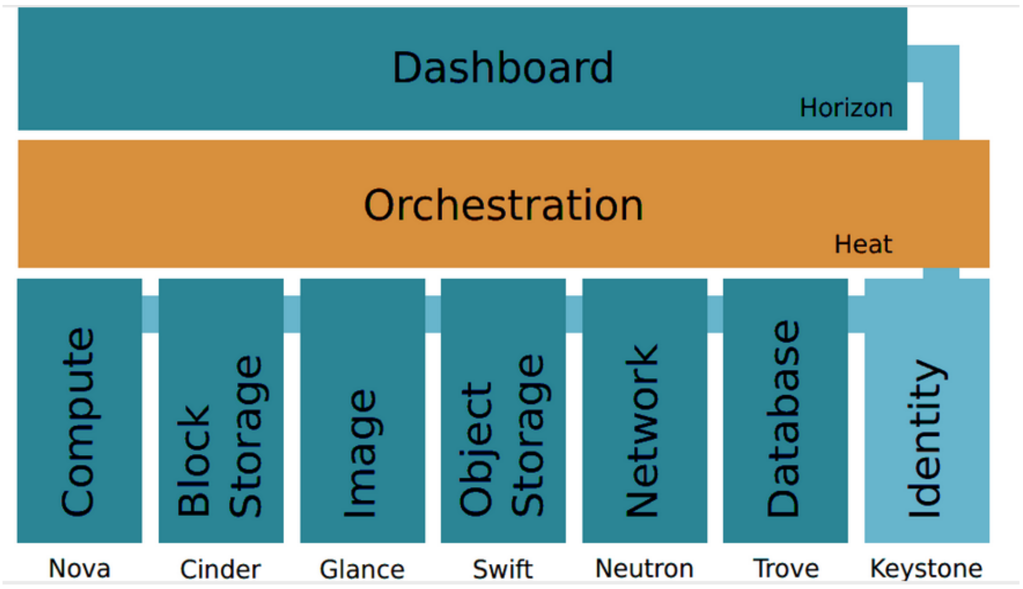


Figure 3. Heat and OpenStack

The assignment of the Heat template is to define a topology infrastructure for a Cloud application, using a readable and writable way of representation, for instance a text file that can be analyzed and corrected by humans.

An original Heat template protocol is still in progress, but Heat can be also used with the Amazon Web Service (AWS) CloudFormation template format, in a compatibility way, so many existing CloudFormation templates can run on OpenStack. Heat provides both an OpenStack-native ReST API and a CloudFormation-compatible Query API. The template is able to represent different kind of infrastructure like servers, volumes, security groups, users and so on.

The auto-scaling engine is also included, integrated with Ceilometer. In that way, in the template, you can represent a scaling-group. There are also relationships between resources, that imply a particular launch order on Openstack. Those ones are likewise represented in the templates and Heat will follow the correct launching order.

Heat templates typically take the shape of plain Yaml documents. Yaml is a strict superset of Json, so Json is still fully supported and templates can be converted between the two formats without information loss. Likewise the serialization format, Heat (for now) is really close to the CloudFormation template model. It is also possible to combine different templates, in order to create a bigger hierarchical infrastructure. The Provider Resource is the method for defining nested stack resources. It provides a very flexible way for both users and those deploying Heat, to define custom resources based on Heat templates if linked with the environments capability to connect template resource names to non-default implementations.

It is possible to automatically scale in a group of virtual servers if only you define it inside the template that contains the resources involved. That is possible thanks to the OpenStack Metering module (Ceilometer) that provides “alarms". Scaling policies and a load balancer are also present. The Ceilometer alarms call a webhook, usually provided by the scaling policy resource in Heat, under particular conditions, and a scaling policy will adapt the scaling group following these input rules and its own configuration.

2.1.2.1 Heat architecture

Heat comprises a number of Python modules shown in the following figure:

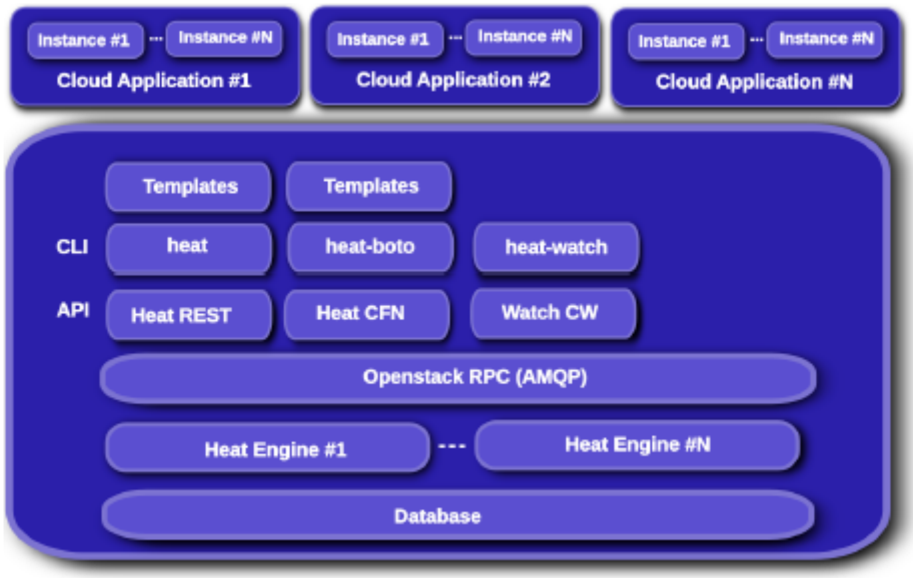


Figure 4. Heat architecture

Those are the different modules:

- **Heat**: the heat tool is a Command Line Interface which communicates with the heat-api to execute AWS CloudFormation API. Of course this is not required, developers could also use the Heat APIs directly.
- **Heat-api**: the heat-api component provides an OpenStack-native ReST API that processes API requests by sending them to the heat-engine over RPC.
- **Heat-api-cfn**: The heat-api-cfn component provides an AWS-style Query API that is compatible with AWS CloudFormation and processes API requests by sending them to the heat-engine over Remote Procedure Call (RPC).
- **Heat-engine**: The heat engine does the main work of orchestrating the launch of templates and providing events back to the API consumer.

2.1.3 Bosh Cloud Foundry

Bosh [5] is like a server that orchestrates the deployment process of a distributed system. There are three prerequisites that are necessary to Bosh in order to deploy the whole system: a Stemcell, a Release and a deployment manifest (they will be described later on).

The Bosh Command Line Interface tool upload them to Bosh. After that, a Bosh installation of a distributed system typically has to follow these major steps:

1. If some packages has to be compiled in the release, Bosh rst creates a few temporal VMs (worker VMs) to compile them. After compiling the packages, Bosh destroys the worker VMs and stores the binaries to its internal Blobstore.
2. Bosh creates a pool of the VMs which will be the nodes where the release will be deployed on. These VMs are reproduced from the Stemcell with a Bosh agent installed.
3. For each job of the release, Bosh picks a VM from the pool and updates its configuration according to the Deployment Manifest. The configuration may include IP address, persistent disk size etc.
4. When the reconfiguration of the VM is completed, Bosh sends commands to the agent inside each VM. The commands tell the agent to install software packages. During the installation, the agent may download packages from Bosh and installs them. When the installation finishes, the agent runs the starting script to launch the job of the VM.
5. Bosh repeats step 3,4 until all jobs are deployed and launched. The jobs can be deployed simultaneously or sequentially. The value “max_in_flight" in the manifest le controls this behaviour. When it is 1, it means the jobs are deployed one by one, otherwise it means that the job are deployed in parallel. This value can be advantageous for a slow system to avoid time-out caused by resource congestion.

2.1.3.1 Internal components

In the following figure the components of Bosh and their relationships are shown.

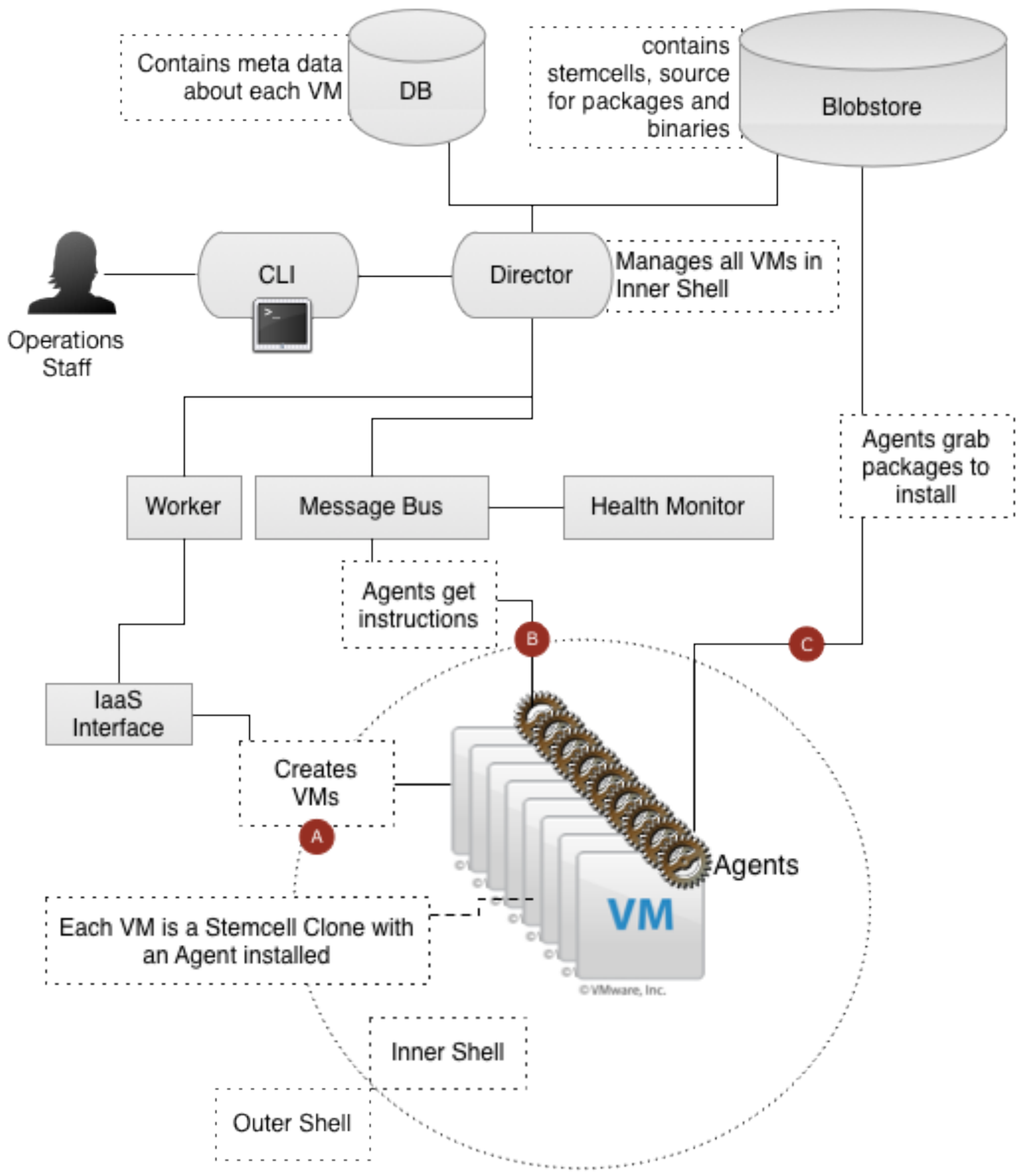

Contains meta data about each VM
DB
contains stemcells, source for packages and binaries
Blobstore
Operations Staff
CLI
Director
Manages all VMs in Inner Shell
Agents grab packages to install
Worker
Message Bus
Health Monitor
Agents get instructions
B
C
IaaS Interface
Creates VMs
A
Agents
VM
Each VM is a Stemcell Clone with an Agent installed
Inner Shell
Outer Shell

Figure 5. Internal components of Cloud Foundry

2.1.3.2 Director

The main component of the whole orchestrating system is the Director. It has the task to manage the creation of VMs, the deployment and generally, it manages the lifecycle events of the services. The Director-Agent receives the commands and the controls only after that the CPI has finished creating resources.

2.1.3.3 Messaging

As communication system Bosh uses NATS message bus. NATS is designed and implemented to be like a reliable publish-subscribe service, it has to be always on and available. By the way, NATS does not support durability or transactions, and it has only an interest-based queuing model. It is able to protect itself to meet the requirements of reliability and availability. This means a distributed messaging system scalable and reliable, but it misses some important features needed by enterprise applications.
NATS has built in primitives to prune the answer receiving interest graph, which reduces the loads of the clients from CPU boost during the throwing of the messages.

2.1.3.4 Health Monitor

The Bosh Health Monitor has the task of observing the health status and lifecycle events. It can get that information from the Bosh Agent and it can send alerts using some kind of notification (i.e. email). The Health Monitor has not a fully complex overview of the events in the system, thus not to alert when an updating of a component is finished.

2.1.3.5 Stemcell

A Stemcell is a VM template with an embedded Bosh Agent so that Bosh can take control of VMs cloned from the stemcell. For Cloud Foundry only a standard Ubuntu 10.04 LTS distribution is used as Stemcell, but stemcells for CentOS 6.4 on some IaaS providers are produced by Bosh. It is possible to upload the Stemcells using the Bosh Command Line Interface and they will be used by the Bosh Director for creating VMs through the CPI, passing along configurations for networking and storage, as well as the location and credentials for the Message Bus and the Blobstore. VMs that come from the same Bosh stemcell are identical at the beginning. But after that the VMs are setted up with different values of CPU, memory, storage or network, and installed with different software packages, they have different behaviour.

2.1.3.6 Releases

A Release is a container of collections of software bits and configurations which have to be installed onto the target system. When you deploy a VM, a collection of software, called job, is linked to it. Congurations can be imagined as templates where are written parameters such as IP address, port number, user name, password or domain name. All these information can be substituted at deploy time by different parameters

2.1.3.7 Deployment Manifest

In the Deployment Manifest is contained the actual values of parameters needed by a deployment. So deployment is something that turns a static release into runnable software on VMs. Bosh changes the parameters present in the release during a deployment process and and makes the software run on the configuration as planned.

2.1.3.8 Deployment Manifest

The content of the Releases has to be stored by the Bosh Blobstore. It is the Bosh Director that has the task of putting into the Blobstore the Releases after that they were uploaded by the Bosh Command Line Interface. At the moment of deploying a Release, the compilation of packages is managed by Bosh and it stores the result in the Blobstore. The Bosh Agent pull a Job and related Bosh Packages, that are located in the Blobstore, as soon as the job is deployed into a VM. If there are some messages that are longer than the maximum size for messages in the message bus, the Blobstore can be used as a temporary store location.
Three Blobstores are supported in Bosh at the moment:

1. Atmos
2. Amazon Simple Storage Service (Amazon S3)
3. Simple blobstore server5.

2.1.3.9 Agent

Bosh Agents receive instructions from the Bosh Director and there is one Agent in every VM. The Director-Agent gives to VMs Jobs or role, within Cloud Foundry. For instance, in case that the a VM has the job to run MySQL, the Agent will receive instructions from the Director regarding all the packages that have to be installed and all their settings.

2.1.4 Openshift

OpenShift [6] is a cloud-computing Platform as a Service product from Red Hat that helps developers to build and deploy web applications. There are three versions of Openshift. The version for private cloud is called OpenShift Enterprise. OpenShift Origin is the software where the service runs and it is an open-source software. Here we will focus on OpenShift Origins.

It is also possible to run binary programs as web applications on OpenShift, whilst they are able to run on Red Hat Enterprise Linux which provides integrated application runtimes and libraries. For that reason, the use of different languages and frameworks is possible. So one task of OpenShift is to watch over the services that lay under the application and scaling the application by need. OpenShift supplies integrated tools that help the developer in supporting the application life cycle, including Eclipse integration, JBoss Developer Studio, and Jenkins.

2.1.4.1 Overview

A lot of languages and tools are supported so the user can develop his application with the ones he wants. And moreover, he can push his application changes to a built-in Git source code repository. All the steps needed to turn code into a running application, including the building step, can be run in OpenShift. It does not matter if it is a simple script or even an external built system, it is always possible to prepare the code for execution in the cloud.

Regarding the deploy it is important to know that every application is made of cartridges: self-contained bits of the application stack, like web servers and databases, that makes simpler the maintenance and configuration. A user only needs to choose the technologies by selecting cartridges. It will be OpenShift that will spontaneously arrange those services.

Furthermore it is possible to manage your running application in the cloud, because it provides tools for monitoring, debugging and tuning on the fly. Nevertheless, OpenShift can scale your application automatically or allocate capacity before the need. An OpenShift application is composed by code, an environment and shared configurations, with one or more cartridges that have the tasks of supplying the languages and services your application needs.

Cartridges have to be deployed to one or more secure containers for the code, these containers are called gears. Different cartridges could get only one gear or have access to all the gears. It is OpenShift that decides where to deploy a cartridge at the moment of adding it to the app based on the type and needs of the cartridge. Your applications are connected together across the gears, so the user references the services those cartridges expose via environment variables.

OpenShift Online currently offers three sizes of gear:

- Small gears provide 512MB of RAM, 100MB of swap space, and 1GB of disk space

- Medium gears provide 1GB of RAM, 100MB of swap space, and 1GB of disk space
- Large gears provide 2GB of RAM, 100MB of swap space, and 1GB of disk space

2.1.4.2 Components

The relations between the main components are shown in the following figure:

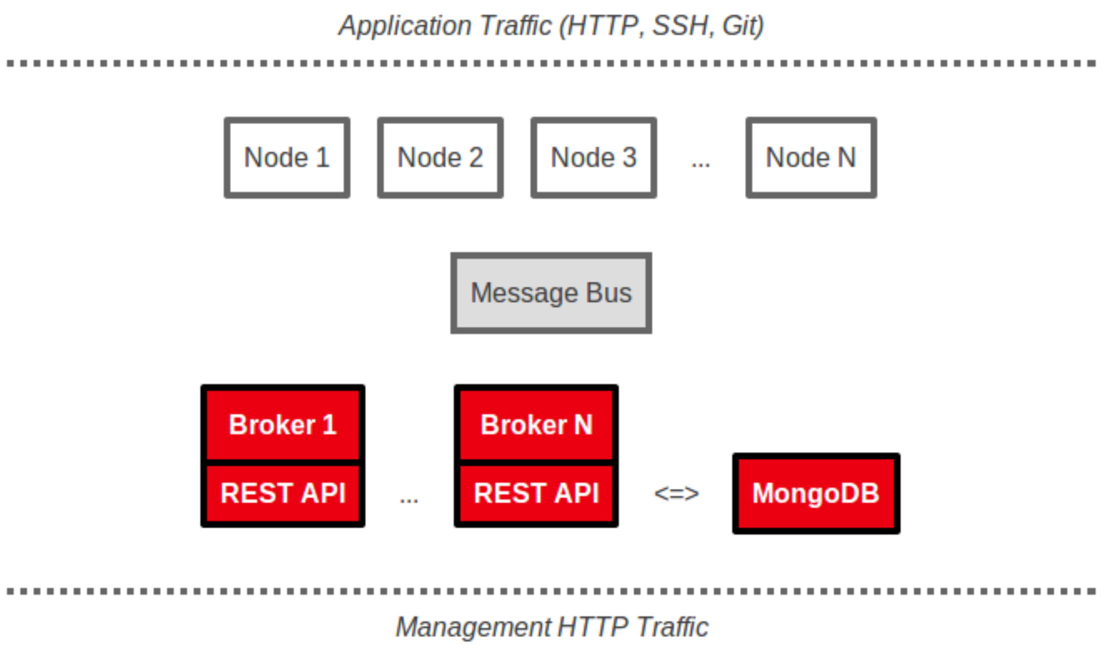


Figure 6. OpenShift Architecture

2.1.4.2.1 Broker

The broker is the single point of contact for all application management activities. It is responsible for the following tasks:

- Managing user logins
- DNS
- Application state
- Application orchestration
- Services and operations

In addition the brokers expose a variety of ReSTful API, that can be reached by:

- Web console
- Command Line Interface tools
- Eclipse JBoss Developer Studio

All these tasks are provided thanks to several components that comprise OpenShift Enterprise. Client applications and the web console interact with the broker via ReST API to manage applications, while developers directly push code to their gears in the nodes.

2.1.4.2.2 Node

An OpenShift node is a host that runs the applications. A Broker can manage many deployed nodes. Gears that contain applications are supported by a node. One of the features of OpenShift is that it is a true multi-tenancy environment. This is accomplished by using SELinux and Cgroup restrictions, used also to separate all applications' resources and data. There is the possibility to deploy a node and a broker on the same host but that is not recommended. Thanks to resource sharing allowed by Nodes, multiple gears can run on a single physical or virtual machine.

2.1.4.2.3 Cartridges

Cartridges have the task of granting the functionality in order to run applications. There are numerous cartridges available with the aim of supporting languages such as Perl, PHP, and Ruby, as well as many database cartridges such as PostgreSQL and MySQL.

2.2 Solution analysis

This section describes the requirements that the NUBOMEDIA EMM needs to meet and a comparison with the existing orchestrators described above, with an accent on the EMM requirements.

2.2.1 Requirement analysis

The comparison of these orchestrators reveals that orchestrators might differ completely from the application-level point of view. For instance, Heat is one of the projects within the Openstack environment building a complete deployment solution. It provides basic functionalities regarding the deployment and management of a virtual infrastructure. Furthermore, Heat is capable to settle auto scaling mechanisms in combination with Openstack's Ceilometer. Thanks to this you can define scaling policies that are responsible to observe the addressed groups of resources adding or removing new and existing instances.

In contrast to that, Cloudify is more like a service orchestrator which means that it provides an advanced solution for automating and managing the application's deployment and post deployment processes. Using Cloudify in combination with Openstack enables the capability to use Cloudify as a service on-top of the heat environment where Heat sets up the Openstack infrastructure (machines, storage, networking) and Cloudify provides a rich set of services, starting with application topology and modeling to complete monitoring analytics and deployment. In addition to that, Cloudify provides also interoperability across other platforms, such as AWS, CloudStack, Microsoft Azure and VMWare just to name a few of them. Nevertheless, next versions of Cloudify are going more and more in the direction of OpenStack but not becoming a piece of Openstack-only software. Besides that, the integration into OpenStack will not duplicate functions existing already in Heat. Cloudify will simply forwarded required steps to Heat gaining a seamless integration of the two tools. In conclusion, it can be claimed that Cloudify is a very powerful tool for productive orchestration across the borders of different clouds allowing to manage virtual machines of individual cloud providers.

Bosh's CloudFoundry is also more than a basic deployer for virtual infrastructures like Heat. It is an enhanced environment tool for managing and automating service orchestration providing also the ability to integrate it into the Openstack environment. In contrast to Cloudify it doesn't use heat but nova and neutron to provide its services on top of a predefined infrastructure.

The Elastic Media Manager has to comply with multiple features, including:

- Deployment of the NUBOMEDIA infrastructure: Capability of deploying and managing virtual machines with specific images and configurations.
- Creation, Recovery and Management of NUBOMEDIA components: they are responsible for providing different types of application services exposed by virtual machines. In practical terms this means that: while creating new media elements of a given type that will be connected with other media elements, it is important to specify parameters as a hint for the placement algorithms. Furthermore, for recovering a specific media element through its unique id, the media element type shall be associated to a specific proxy that will enable to manage it remotely.
- Creation, Recovery and Management of distributed pipelines: NUBOMEDIA applications are built by chaining different media elements into a media pipeline applying a specific workflow over the media stream.
- Integration with Connectivity Manager: The Connectivity Manager contains the logic for monitoring and managing the computing infrastructure. It should expose an OCCI interface extended for offering more functions to the elastic media manager layer, including placement.
- Integration with Openstack: The Openstack environment is selected as the virtual infrastructure and therefore it is needed to integrate it with Openstack.
- Auto scaling mechanisms: When specific instances are running out of resources, it is needed to provide further resources to fulfill the current amount of requests properly, and vice versa in case of unused resources. By providing auto scaling mechanisms we need also a monitoring system and furthermore the possibility to create application-related meters. Another important feature is the termination of specific instances during the scaling-in process. That is why we need explicit termination rules described below more in detail.
- Monitoring/custom meters: Auto scaling mechanisms are based on monitored resources. So on one hand it is required to provide a basic monitoring system and on the other hand it is needed a mechanism to take into account application-related meters. Application-related meters are mandatory because basic monitoring systems do not cover all the meters that may needed for a high-level auto scaling system.
- Termination rules: It is required to define termination rules applied to the scaling-in process of media elements to take care about active sessions and essential instances. Otherwise, it may happen that the orchestrator terminates an instance that is active (e.g. processing requests). In this case the user requests will be canceled directly and leads to an undesired behavior of requested services. So the orchestrator have to provide a mechanism where we can define a specific termination rule. These termination rules depends on the provided services and therefore it is mandatory to provide the opportunity to define these rules as needed. In most cases we want to terminate completely idle instances only but it might be possible to need different termination rules.
- Bootstrapping: Starting and configuring services while adding new instances is done after the instance's boot-up. This is why the orchestrator has to provide a bootstrapping system that executes commands directly after the launching new instances.

- RESTful API/Command line interface: The RESTful programming application interface and command line interface is needed for managing the NUBOMEDIA infrastructure, for instance, deploying and disposing the NUBOMEDIA infrastructure or creating and managing services.

The following taxonomy depicts which of these features are provided, provided partially and not provided by existing orchestrators and what the NUBOMEDIA orchestrator, called Elastic Media Manager (EMM), needs to support. Depending on the selected orchestrator it is needed to extend this one to fulfill the requirements completely.

	Cloudify	**Bosh CloudFoundry**	**HEAT**	**EMM**
Deployment of the basic infrastructure	YES	YES	YES	YES
Creation, Recovery and Management of Media Elements	PART[1]	PART[1]	PART[1]	YES
Creation, Recovery and Management of distributed Pipelines	PART[2]	PART[2]	PART[2]	YES
Integration with the Connectivity Manager	NO	NO	NO	YES
Integration with Openstack	YES	YES	YES	YES
Auto scaling mechanisms	PART[3]	PART[3]	PART[3]	YES
Monitoring / custom meters	YES / YES	YES / NO	YES / YES	YES / YES
Termination rules	NO[4]	NO[4]	NO[4]	YES
Bootstrapping	YES	YES	YES	YES
RESTful API / Command line interface	YES / YES	YES / YES	YES / YES	YES / YES

[1] In general, it is the main task of an orchestrator to deploy a basic cloud environment with certain components. But apart from that the physical placement of these components bases on the integration with the Connectivity Manager and this is not provided by any existing orchestrator because the Connectivity Manager is part of the specific architecture of NUBOMEDIA and therefore must be integrated directly.
[2] As mentioned above the distributed pipelines are responsible for applying a specific workflow over the media stream. This is also a very specific functionality of NUBOMEDIA and hence it is not provided by any existing orchestrator.
[3] Basic auto scaling mechanism are provided by all of these orchestrators but what we need is an advanced auto scaling system that provides monitoring, enables the creation of custom meters and allows also the definition of certain termination rules.
[4] As described above, termination rules are quite important for the lifecycle of virtual machines and its services. Obviously, each of these Orchestrators uses termination rules for the scaling-in process but these rules are very static and not editable. So the EMM have to provide a mechanism to define these termination rules depending on the provided services.

Table 1. Features comparison

2.2.2 Our solution

Considering the evolution of the NUBOMEDIA platform, and also its requirements, we started within Release 2 using Heat as orchestrator, while moved to our own implementation from Release 3. Indeed two of the most important requirements from the NUBOMEDIA infrastructure are related with:

- in-scaling procedures: it shall be possible to define a two steps procedures before removing completely a NUBOMEDIA component while performing in-scaling operations. Once a NUBOMEDIA component is selected for removal this procedure should allow first to signal it to stop accepting new incoming requests, and send a signal back once all the sessions it was serving are terminated
- Media session placement: the EMM shall expose an interface for runtime session placement on the media servers available inside the NUBOMEDIA infrastructure

While utilizing Heat during Release 2, it was analyzed that the above-mentioned requirements were not satisfied.
In particular Heat does not provide any placement algorithms: the selection of the Hardware on which the VMs are instantiated is done by the OpenStack scheduler and its algorithm cannot be modified while using Heat.
Regarding the in-scaling problem, Heat does not support any mechanisms for communicating runtime with the instantiated resources.

3 Functional Architecture

As already described in the introduction, the EMM should support the lifecycle management of the NUBOMEDIA platform providing the following functionalities:

- On-demand instantiation: given an existing cloud platform, an administrator may request the instantiation of a NUBOMEDIA platform via an API exposed by the EMM
- Cloud-manager independent: the technology used as a cloud manager shall not by tied in the EMM.
- Elasticity: it shall provide mechanisms for scaling in or out each specific NUBOMEDIA component, based on metrics coming from the application or the infrastructure level
- Multi tenancy: it shall be possible to instantiate multiple instances of the NUBOMEDIA platform in parallel.

In order to realize such functionalities, a functional architecture of the EMM is proposed in Figure 7.

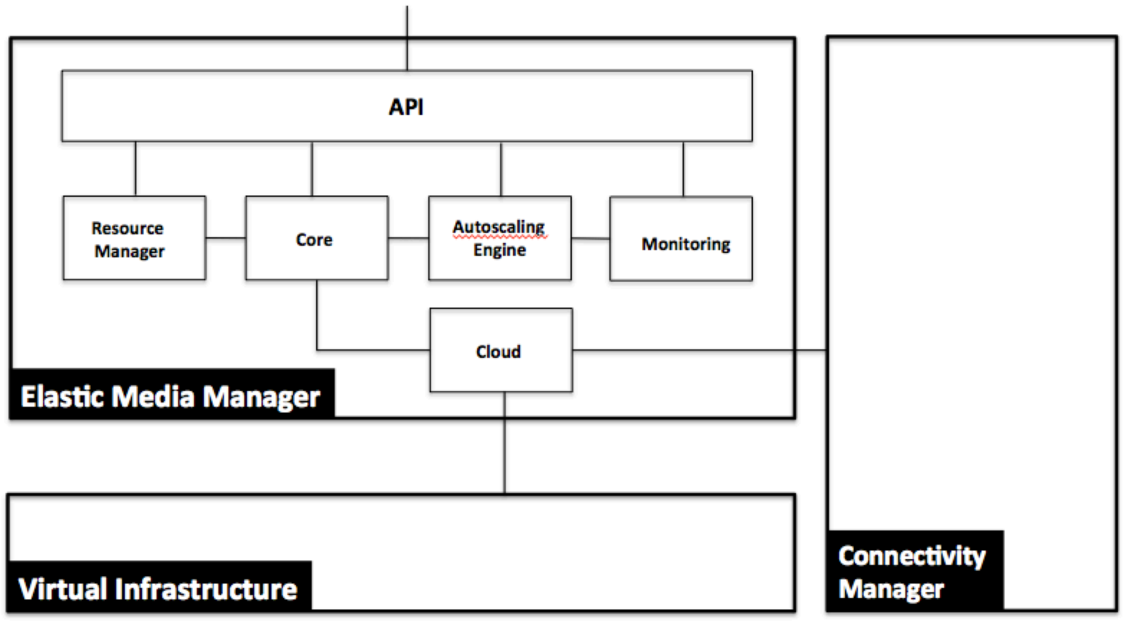


Figure 7. Functional Architecture

The EMM exposes a northbound API to the Command Line Interface (CLI) or the Graphical User Interface (GUI) through which the administrator of NUBOMEDIA can interact with it. In particular this entity is in charge of exposing all the functionalities carried by the EMM. Those main functionalities are logically decoupled in four different entities, which are explained into the next subsections.

3.1 Resource Manager

The Resource Manager is mainly a catalogue of available NUBOMEDIA components. Basically this entity is a catalogue of the available resources. Whenever a new component is added to the NUBOMEDIA platform, it has to be registered by the Resource Manager, so that it will be available to be deployed whenever an admin will require it.

3.2 Autoscaling Engine

It provides mechanism for managing the runtime phases of the NUBOMEDIA platform lifecycle. In particular this component provides an interface (through the API) to the NUBOMEDIA administrators for creating policies on specific NUBOMEDIA components. Those policies provide information on how to handle load situations.

3.3 Monitoring

It abstracts the different monitoring systems which might be available, providing a homogenous interface towards the Auto-scaling Engine and the API components for retrieving monitoring information at all levels: from the Infrastructure to the Application.

3.4 Cloud

This entity represents the southbound interface to the Cloud. In particular it provides an abstraction of the different technologies which might be used for managing the Virtual Infrastructure. Decoupling the Core component from the specific cloud technology will allow easily to migrate from OpenStack to a different cloud.

3.5 Core

The Core is the main entity in the EMM Architecture. The most important function provided by the Core is the lifecycle management of the NUBOMEDIA platform. It exposes an interface to the APIs through which the administrator can request the instantiation/removal of a NUBOMEDIA instance. It interoperates with the Cloud entity for requesting virtual resources.

4 Software Architecture

The EMM has been implemented with modular software architecture. It has been implemented in Python, and for the current NUBOMEDIA release (Rel. 3) provides the following packages and functionalities:

- Interfaces: contains the interfaces of the EMM core services
- Util: contains some utility methods and classes
- Clients: contains all the classes that interface the OpenStack [1] clients
- Emm_exceptions: contains all the custom exception used in the EMM.
- Model: contains all the classes composing the EMM Model
- Wsgi: contains the REST APIs that are exposed
- Core: contains the core classes that handle all possible operations
- Services: contains the implementations of the interfaces
- Test: contains a set of tests

In Figure 8 the high level software architecture is shown. In the following lines all the component implementing the functionalities are described but for a more detailed view on the components please see section 4.6

2)

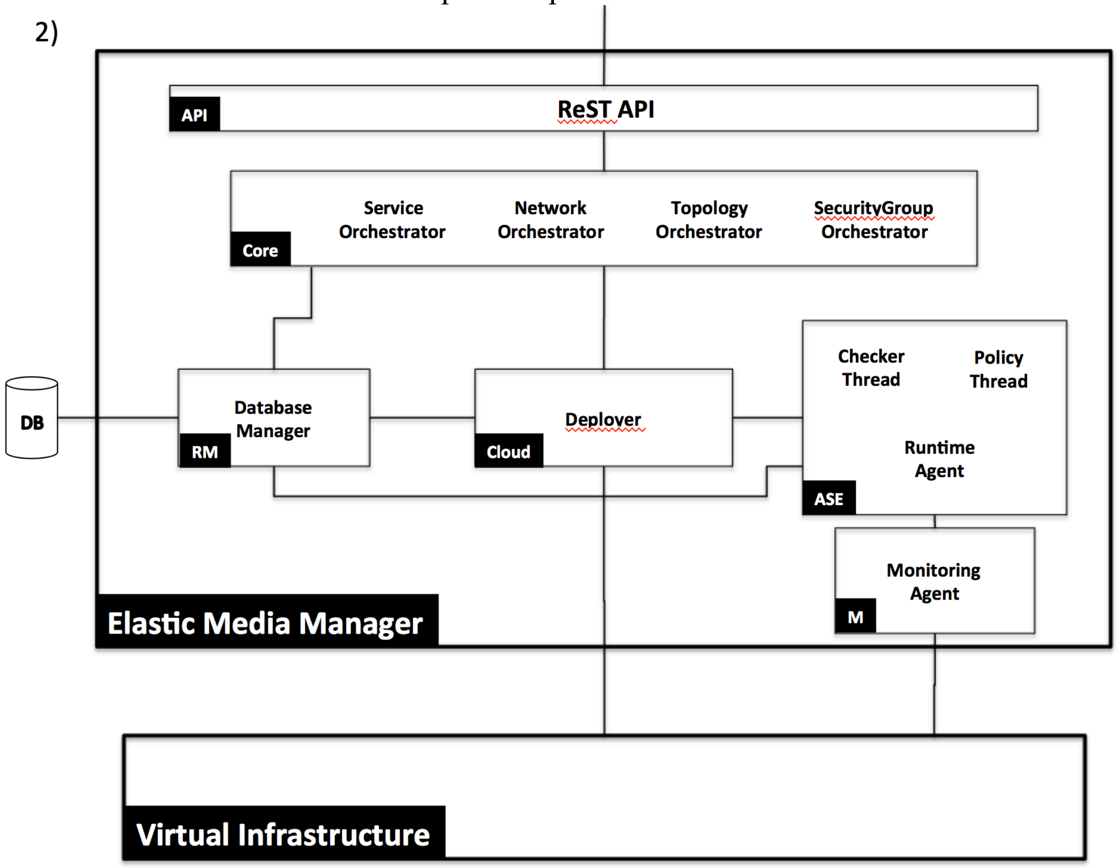


Figure 8. NUBOMEDIA EMM software architecture

First of all the EMM should expose an API for allowing remote control of the NUBOMEDIA platform components. This API should expose basic CRUD operations on the elements available in the cloud environment. Considering the nature of the objects managed by the EMM, a REST API seems to be the most suitable candidate for supporting those operations.
Under the ReST API, the core functionalities are implemented by a set of orchestrators that have the task of executing the logic needed depending on the API called. The ServiceOrchestrator is in charge of executing the API regarding the Services, the TopologyOrchestrator is in charge of executing the API regarding the Topologies, The SecurityGroupOrchestrator is in charge of executing the API regarding the SecurityGroups and the NetworkOrchestrator is in charge of the operation regarding the Networks.
Those components have a relation with the Resource Manager and the DatabaseManager in order to be able to store the state of the object passed to and by the API module.
In particular one of the main functions of the TopologyOrchestrator is to deploy a Topology, for this reason it has a relation with the Deployer implementing the Cloud functionalities. The Deployer is also related with auto-scaling engine related components such as the Runtime Engine and the Checker Thread. In order to apply the auto-scaling functionalities the Runtime Agent must have access to the Monitoring functionalities in particular to the Monitoring Agent. The Deployer and the Monitoring Agent have an interface to the Virtual Infrastructure in order to be able to achieve their tasks.
As previously said a more detailed description of the NUBOMEDIA EMM components interfaces and functionalities can be found at Section 4.6.

This architecture enables to create a NUBOMEDIA infrastructure starting from a simple request. This process from a high point of view is described in its phases in the following subsections.

4.1 Deployment

The deployment phase starts when an administrator requests the instantiation of NUBOMEDIA. Inside the appropriate request, the administrator specifies what are the capabilities required for this instance of the platform. Based on the information received, the EMM needs to instantiate virtual resources on the Virtualized Infrastructure. For this step there are several ways of doing it:

- Request directly OpenStack virtual networks and compute resources
- Request to Heat the required resources creating a template

For simplifying and making homogenous the whole process, we selected the second option. The EMM has to create a Heat template based on the required resources which are requested by the admin. Once the template is created, it is sent to Heat using its API. At the end of the deployment process, Heat sends back all the information related with the instantiated resources which the EMM can use for moving to the next phases.

4.2 Provisioning

During the provisioning phase it is required to configure the NUBOMEDIA software components. The installation of a NUBOMEDIA element has been decoupled in two different phases:

- Installation of the Software, which may include download of source code, installation of external libraries, compilation of the code
- Configuration of the Software, which involves configuration of parameters which typically varies after each new deployment, for instance the IP of a virtual machine where a Media Server is running, or the port number where the server should bind.

The initial phase has been included in a separate process, which is not part of the provisioning. Basically whenever a new version of the Software is available, a new Software image is created and published on the image repository of the Virtualized Infrastructure. The reason of applying this process is mainly to reduce the time needed to boot a new NUBOMEDIA component. Indeed, if the Software is already installed in the VM, after booting it should be only configured.

Hence, the role of the EMM in the second phase is to configure parameters that are always dynamically updated. As previously said, Heat [2] provides the possibility to run a script after the deployment of a new virtual machine. This script is defined inside the HOT (Heat Orchestration Template) under the field user_data. Inside this script, it is possible to refer to parameters defined in the output section of the HOT, but not concrete at the moment of the script definition. These parameters can include IPs or Ports for instance. These are values that change dynamically on each deployment, but in this way we are able to refer them as variables.
For instance we can run:

```
sudo  sed  -i  \"s/^\\([0-9]*\\.[0-9]*\\.[0-9]*\\.[0-9]*[
\\t]*localhost\\)/\\1 $HOST_NAME/\" /etc/hosts
```

In order to modify the /etc/hosts file.

Or even:

```
sudo sed -i \"s/^\\([ \\t]*\\\"address\\\"[ \\t]*\\):[
\\t]*\\\"Broker private IP Address\\\"*.*$/\\1:
\\\"$BROKER_IP\\\",/\" /etc/kurento/control-
server.conf.json
```

In order to set the IP of another VM (not yet known) in our configuration file.

4.3 Runtime

Once all the NUBOMEDIA components are deployed and started, it is needed to actively check which specific levels of SLAs are met. The EMM provides a runtime system which actively controls the situation of a group of NUBOMEDIA components, and based on some policies, which the administrator inserted, decides whether to scale in or out them.

When the administrator requests the instantiation of a NUBOMEDIA platform, it puts also specific policies to specific elements. A policy is just a set of alarms and actions. There are two different ways of realizing such system via triggering mechanisms:

- Actively: the EMM actively checks whether the conditions of the alarm are met. For doing it, it typically requests to the monitoring system last values of the metrics involved, and performs some basic operations for evaluating the status.
- Passively: the EMM pushes the alarm into the monitoring system. When the conditions are met, the monitoring system triggers the alarm to the EMM.

For NUBOMEDIA Release 3, it was decided to use the first approach. Once the EMM realizes that an alarm is in ACTIVE state, it has to trigger the execution of the action contained in the Policy. Typically this policy can involve the instantiation/removal of the NUBOMEDIA elements.

4.4 Internal class diagrams

In Figure 9 it is represented a diagram of the most relevant classes of the EMM.

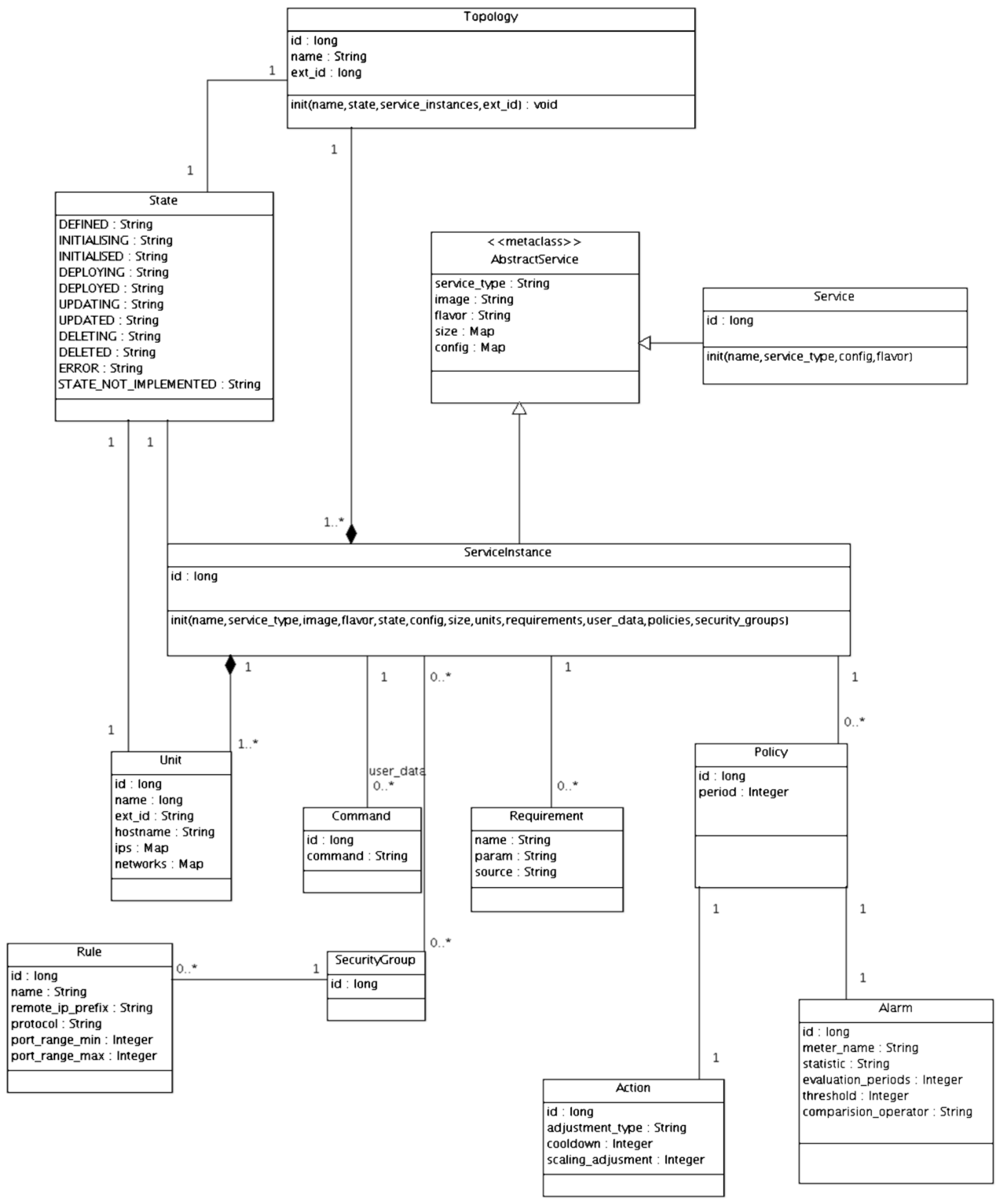


Figure 9. Class diagram of the EMM

4.4.1 Topology

The Topology object represents the complete NUBOMEDIA platform, with all the components and virtual resources.

4.4.2 Service Instances

A Topology contains a list of Service Instances. Each Service Instance is an instance of a service deployed in one or more VMs. It takes its default field values from the Service and update them with the ones contained in the API request. The Service Instance can be installed on one or more VMs, this is why it contains a list of Units (see 4.4.3). It contains also the auto-scaling policies, in this way the EMM can apply different policies to different Service Instances. The auto-scaling actions (scale in or out) will affect the units field, adding or removing a Unit. For instance, at run-time, a possible deployed topology could be as shown in Figure 10.

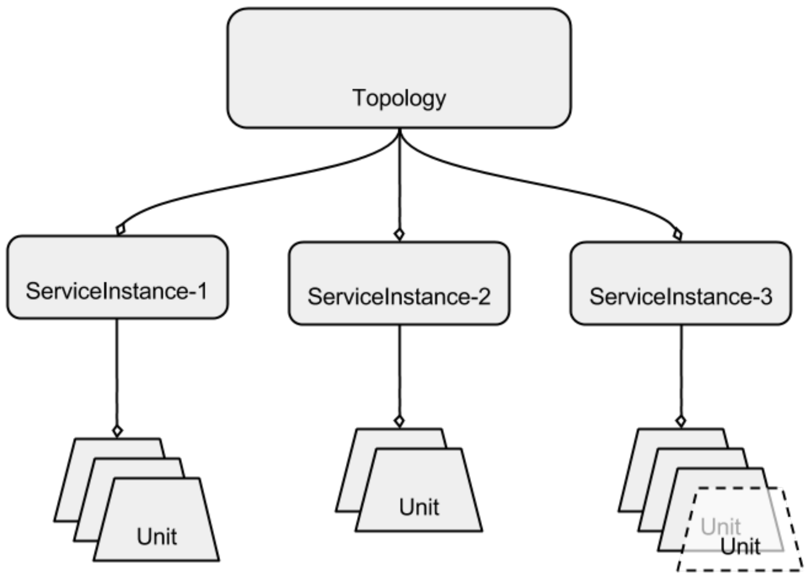


Figure 10 Run time structure of a topology

In this figure only instances are represented. As previously said, a Topology contains one or more Service Instances, which in turn contain one or more Units.

A Service Instance has different fields:

- Image: the image to use in Openstack
- Flavor: the flavor to use in Openstack
- Size: the size of the VM
- Service_type: the type of the Service to be deployed (see section 4.4.4)
- Config: a general set of configuration parameters

4.4.2.1 Command

The Service Instance has a list of commands. These commands will be executed in the VM during the Cloud Init (A set of python scripts and utilities that handles initialization and configuration of cloud instances. It is installed in the standard Ubuntu Cloud Images and also available in the official Ubuntu images on EC2)

4.4.2.2 Requirements

The Requirement is an object describing the relation between a Service Instance and another one.

- Name: of the required parameter
- Param: the required parameter
- Source: the Service Instance that provides the parameter

4.4.2.3 Security Group

Each Service Instance can have a pointer to a different Security Group. Each Security Group is an abstraction of the Heat Security Group [7]. It has, in fact, a list of Rules.

4.4.2.3.1 Rule

A Rule contains:

- Name
- Remote_ip_prefix
- Protocol
- Port_range_min
- Port_range_max

4.4.2.4 Policy
The Policy represents an auto scaling policy. It has a period that is the frequency of checking the Alarm and an Action to be executed when the Alarm results active

4.4.2.4.1 Alarm
The Alarm represent an equation where:
- Meter_name: is the name of the meter item
- Evaluation_period: is the period on which the meter has to be applied
- Threshold: is the threshold
- Comparison_operator: relates the meter value to the Threshold
- Statistic: can be Average

4.4.2.4.2 Action
The Action is the action to be executed when the Alarm is active:
- Adjustment_type: add or remove Unit
- Scaling_adjustment: how many Units to remove or add
- Cooldown: a period where all the alarms are silent, no action can be executed during this period in orders to avoid wrong scaling adjustment caused by fake values.

4.4.3 Unit
The Unit is a representation of a Virtual Machine. It has all the fields that can be retrieved from Openstack regarding a VM such as:
- IPs: the list of ips it has
- Networks: the network it is attached to
- Hostname: the hostname
- Ext_id: the id in Openstack

4.4.4 Service
A Service represents an entry in the catalog. The catalog consists in all the available Services stored in the database. When creating a Topology, the Service Instances in it must be a copy of a Service already existing in the catalog. This is done because a Service already contains default values that will be possibly overwritten during the deployment.

4.5 EMM Core
The modules **core**, **interfaces** and **services** represent the heart of the EMM.

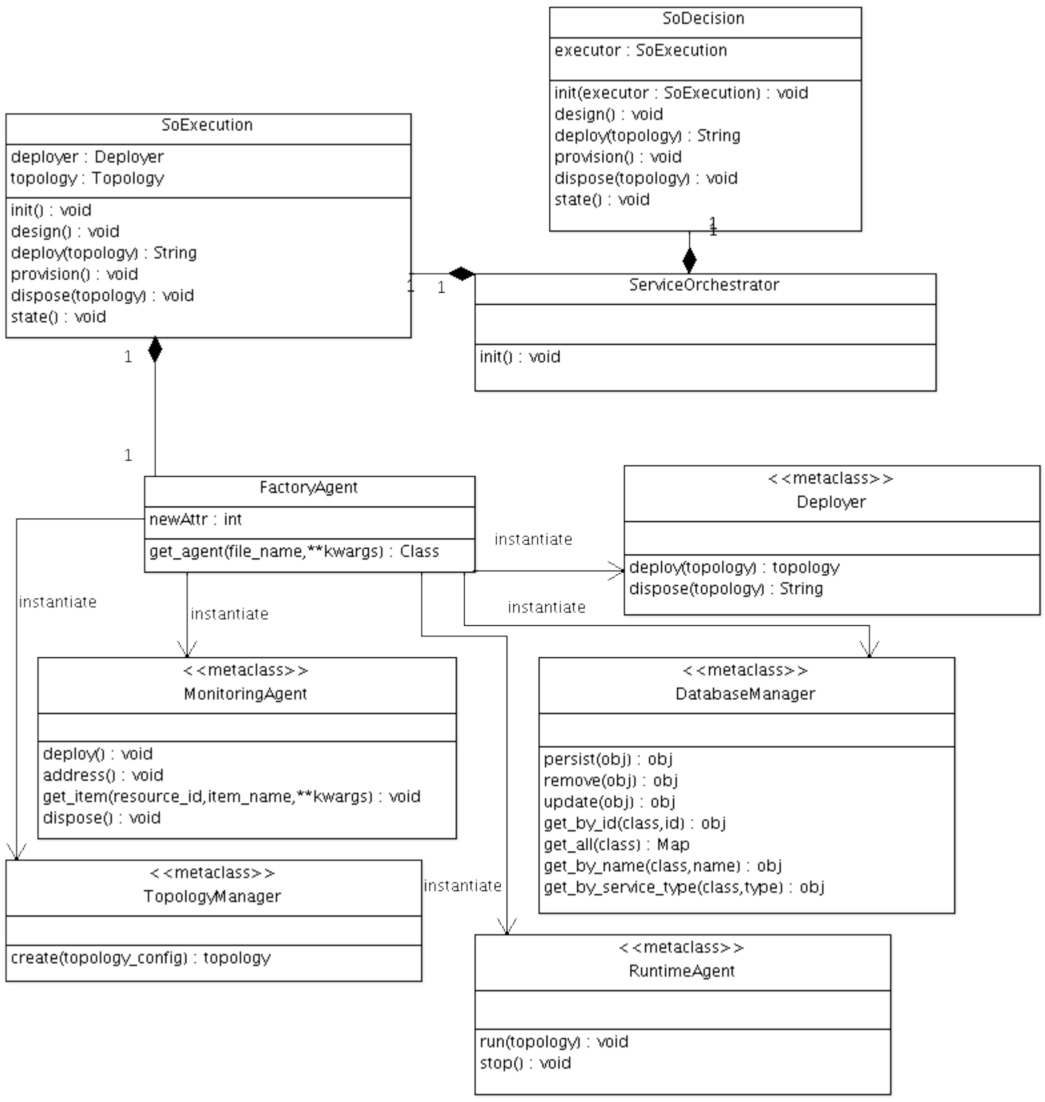


Figure 11. Core, Interface and Factory Agent modules

When the wsgi module receives a request, it uses an implementation of one interface, depending on the request. Basically it uses an instance of the ServiceOrchestartor class. This instance asks to the FactoryAgent an implementation of the required interface. For example, it will request an implementation of the Deployer in case a create topology is triggered.

The FactoryAgent reads the configuration file (./etc/emm.properties) where the implementations to be used are defined and will dynamically instantiate the requested implementation. The FactoryAgent is a highly dynamic Factory. Inside the configuration file there are present the name of the implementation classes that the Factory needs to instantiate. So thanks to the reflection the FactoryAgent instantiates an instance of the correct class. From the point of view of a developer, in case a new implementation of an interface is needed, he has just to put the new file in the services module and change the name in the configuration file.

4.6 NUBOMEDIA interfaces' implementations

Figure 12 shows that the FactoryAgent instantiates all the implementations of the interfaces. The most relevant are the NuboDeployer and the NuboRuntimeAgent. In the

Figure 13 and Figure 14 there are more detailed information regarding the functional relations between these interfaces.

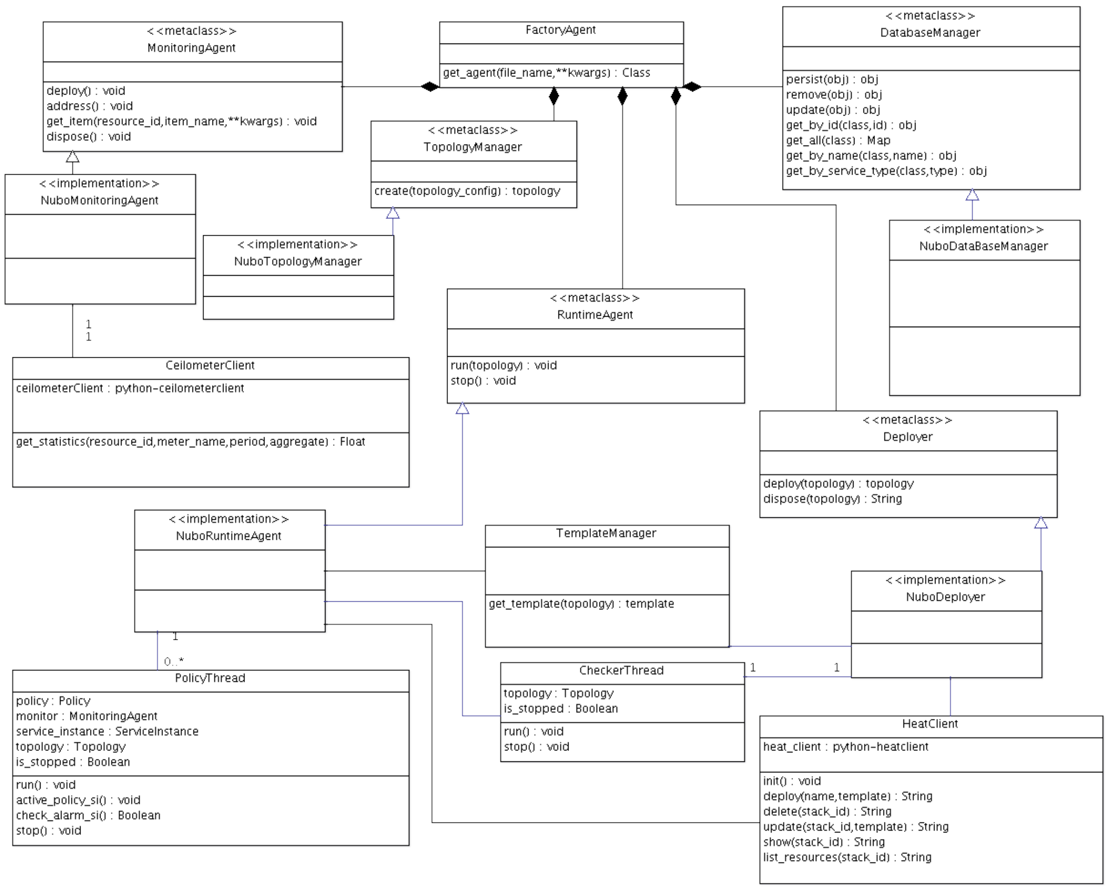


Figure 12 NUBOMEDIA interfaces' implementation

4.6.1 Deployer

When a create topology is triggered, the Deployer has the task of creating the topology on the Cloud, in our case for the deployment we use the Heat [2] client. We called our implementation of the Deployer NuboDeployer. After the topology, translated into HOT, is sent to Heat. In case there are not errors, the RuntimeAgent is invoked and the Deployer, before terminating, returns the created topology to the User.

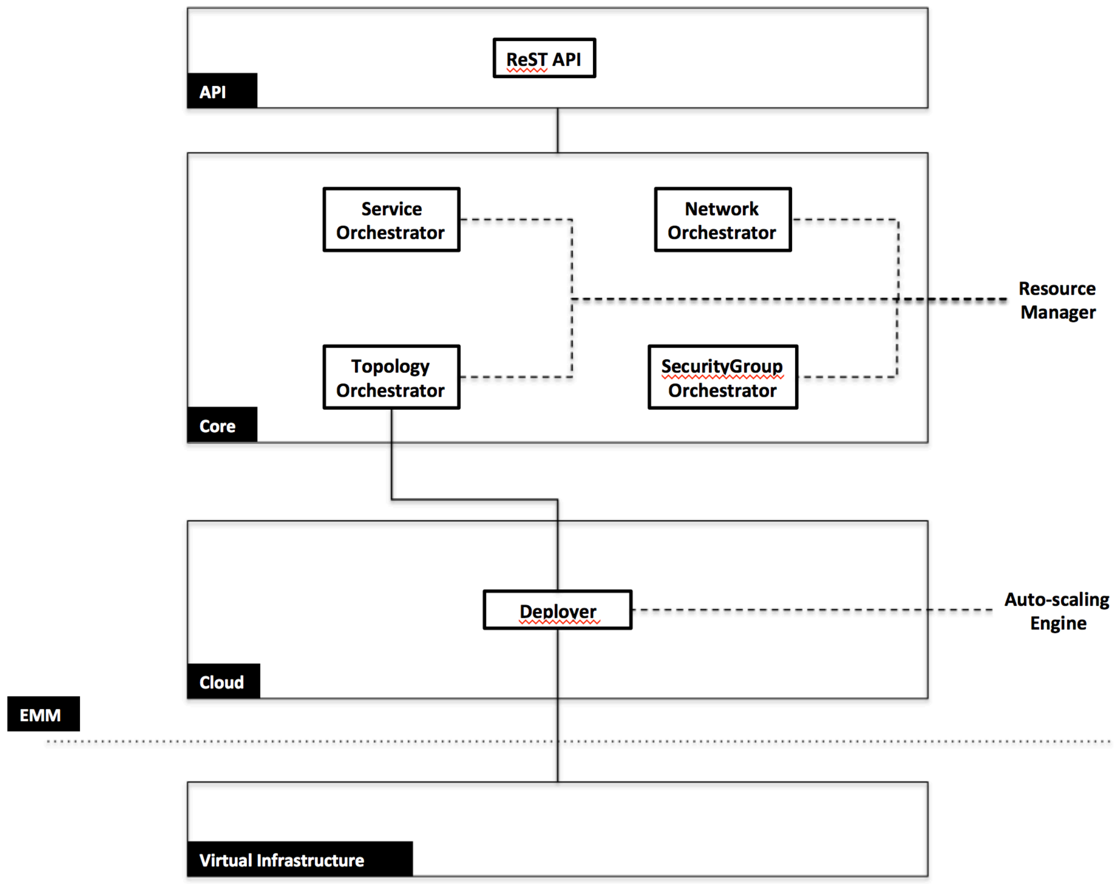


Figure 13 Deployer relations

4.6.2 TopologyManager

The TopologyManager has the task of translating the Users' requests into objects belonging to our model. Our particular implementation is called NuboTopologyManager. The Users' requests come from the Rest API (See Section 6 and **¡Error! No se encuentra el origen de la referencia.**) and contain json body. This body is firstly translated into dictionary and after the NuboTopologyManager translates the dictionary into a Topology object (See 4.4.1)

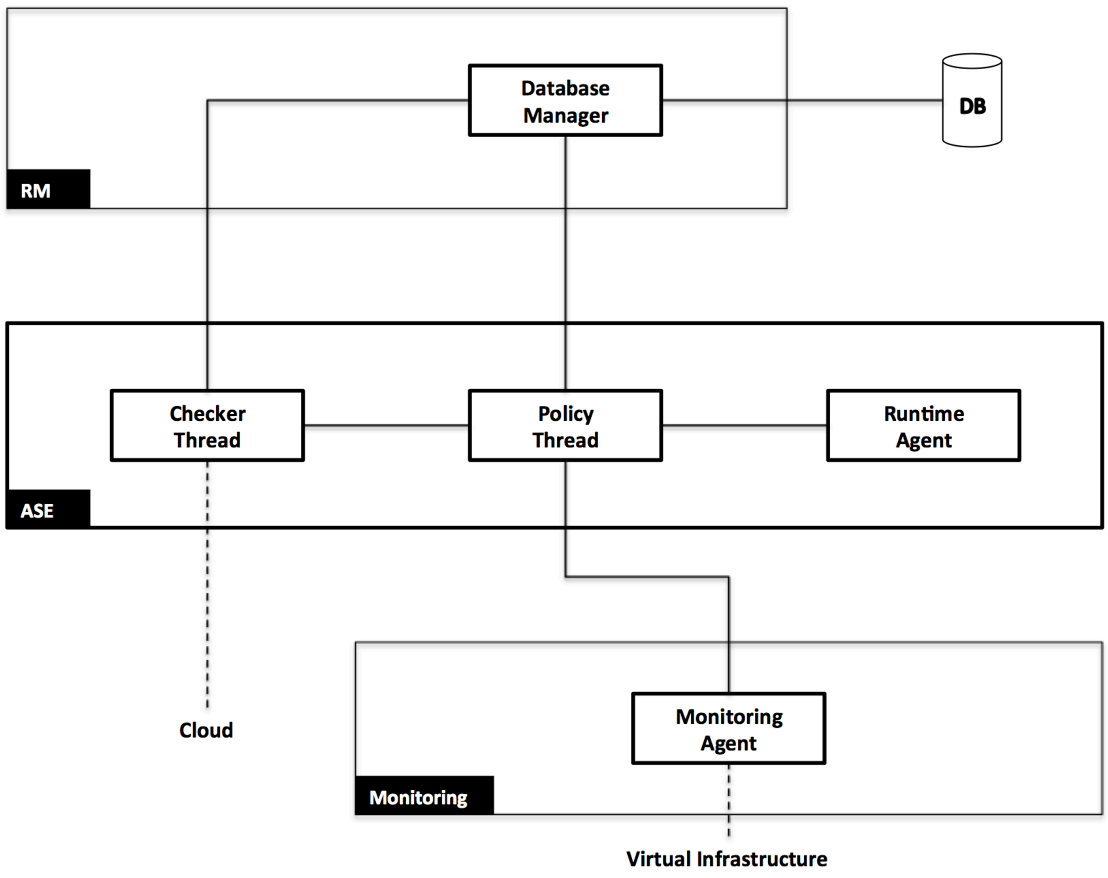


Figure 14 Runtime Agent, Monitoring Agent and Database Manager relations

4.6.3 Runtime Agent

Our implementation of the RuntimeAgent metaclass is called NuboRuntimeAgent. After the NuboDeployer has sent the template corresponding the topology to Heat, the NuboRuntimeAgent spawns a thread called CheckerThread. This thread has the task of checking the state of the whole topology and also of updating the new states into the database. Each time there is a Policy in the Service Instances of the Topology, the NuboRuntimeAgent creates a new PolicyThread that is in charge of the auto scaling. The PolicyThread checks whenever one Alarm is active, in that case, it scales the service in or out, depending on the Action. When a Topology is deleted also all the PolicyThreads and CheckerThreads related to that topology are terminated. Examples of possible operations are described in the Sequence Diagram section (See 5)

4.6.4 MonitoringAgent

The MonitoringAgent has the task of interfacing the monitoring system. In our case we chose Celiometer [8] and we called the real implementation NuboMonitoringAgent. So the NuboMonitoringAgent offer an API that allows the EMM to obtain Celiometers' values of a specific VM. The main method of the MonitoringAgent is the get_item(resource_id, item_name, **kwargs). This method provides a easy way to retrieve the value of an item (for example the CPU usage percentage). The resource_id refers to the VM external id and the item_name refers to the name of the requested item. Inside the dictionary kwargs may be present two value of time indicating a period. In case these values are in the dictionary, the result will be an average of the requested item over the indicated period.

4.6.5 DatabaseManager

The DatabaseManager manages the connection with the selected Database. In implementation is called NuboDatabaseManager and uses MySql. This entity is a singleton and manages the high level concurrency. In particular it exposes the classic CRUD (Create, read, update, delete) operations:

- Persist
- Update
- Remove
- Get_by_id
- Get_all

Plus other specific operations:

- Get_by_name
- Get_by_service_type

This last two methods help to find an object using its name or to find a Service using its field service_type.
Before running each method, the connection is created to the database. Then, when the operation is done, the NuboDatabaseManager tries to commit the changes and if there are not errors the connection is closed. In case of errors, a rollback is applied and the state of the database returns to the moment before the operation is invoked.
All these operations are “synchronized” to all the system. This means that each method can be invoked by only one thread at the same time. Moreover the NuboDatabaseManager is a singleton, this guarantees that there is only one instance of it in all the system. These two features provide a non-concurrent access to the database.

5 Sequence Diagrams

This section shows some of the sequence diagrams for the main operations executed by the EMM.

5.1 Topology deployment

The sequence diagram shown in Figure 15 illustrates what happens during the deployment of a new topology. It starts with the user request sending a configuration file to the EMM and ends with the response including all the details of the deployed topology. It is assumed that addressed services and security groups are already stored in the database. Otherwise, the deploy function will return an error message that the requested service or security group was not found. In the following, each step is described more in detail:

1. The user sends the configuration file of the topology in json format to the Rest application programming interface (HTTP POST to /topologies) to request the creation and deployment of a new topology. This configuration file contains all the needed parameters for the description of the virtual infrastructure of the defined topology.
2. In the next step the EMM starts the modeling of the topology described in the configuration file. Therefore, the TopologyManager requests predefined services and security groups from the DatabaseManager. If the services and security groups do not exist, the user have to create them before and have to start from

step 1 again. If they are already there, the TopologyManager creates the topology, persists it to the database and returns the topology object.

3. Afterwards, the deploy function of the Orchestrator is called and gets the topology object as input. The Orchestrator in turn calls the Deployer deploy function.
4. The Deployer uses the TemplateManager to create the HOT template file. In the following, this template file is sent to the HeatClient. The HeatClient deploys and creates the corresponding stack on the Openstack environment and returns the stack details coming directly from the Openstack heat client. Afterwards, the RuntimeAgent triggers the execution of the CheckerThread and all the PolicyThreads if needed (depends on auto scaling bases on policies described in the topology). The CheckerThread checks the states frequently and updates the topology and its components. As mentioned before, the PolicyThread is responsible for the auto scaling mechanism. Therefore, each policy runs in its own PolicyThread and evaluates gathered values frequently. If the groups of resources reach a certain threshold (coming from the specific policy), the auto scaling is triggered and updates the topology (add or remove units).
5. Finally, the topology goes back and is returned to the user in json format.

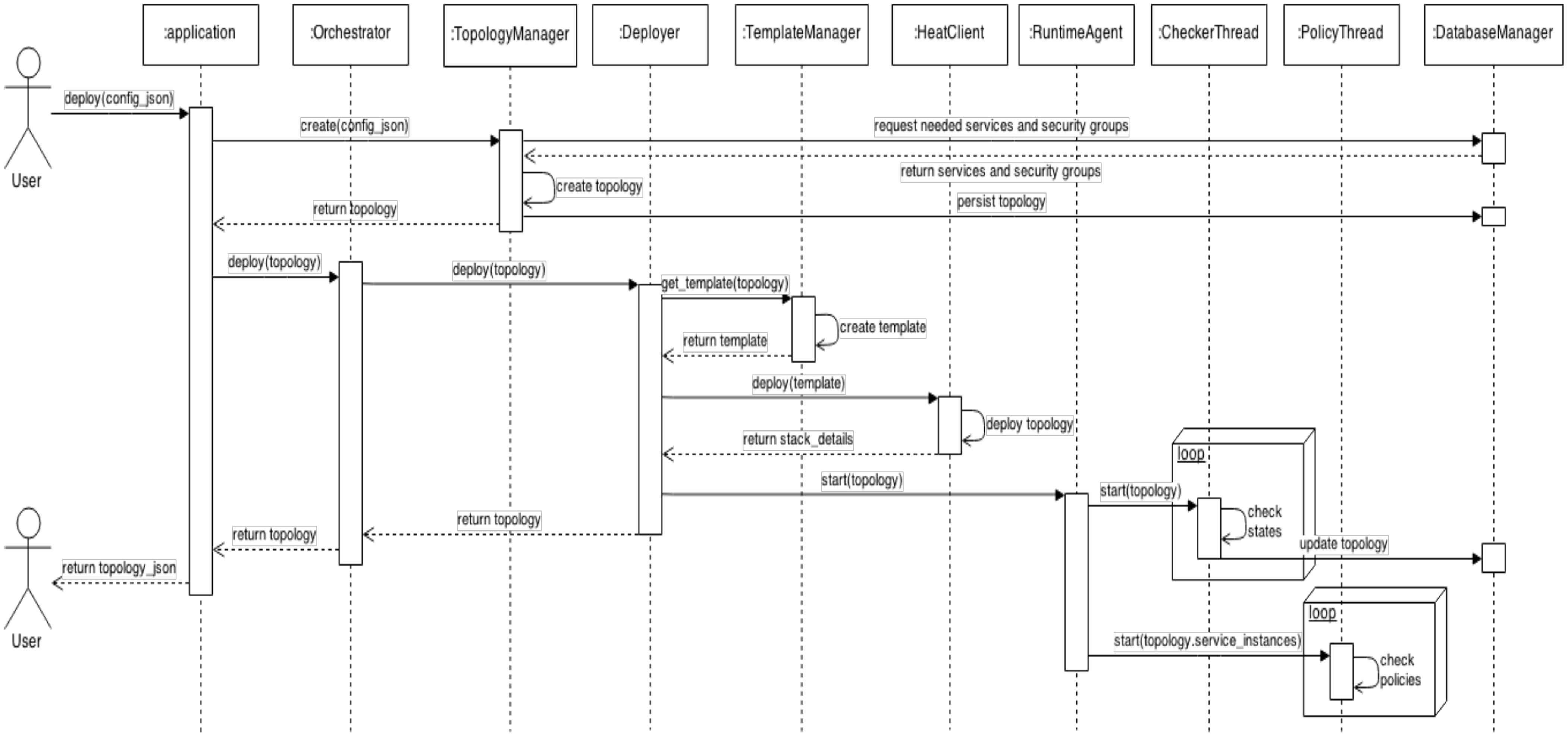


Figure 15. Sequence diagram showing the deployment of a topology

5.3 AutoScaling (Upscaling)

The AutoScaling mechanism is responsible for the creation and deletion of units to overcome highly- or less-used infrastructures for avoiding performance issues or unused resources. The sequence diagram below shown in Figure 17 illustrates how the AutoScaling mechanism works during the runtime in case of an upscaled group of resources. This process starts directly after the deployment of a new topology is finished and ends when the topology is deleted. So basically this process runs in an infinite loop and checks specific meters of addressed resources all the time. Therefore, it is quite important to involve a monitoring system that observes the resources on the base of predefined meters. In the following, every step is described more in detail:

1. As mentioned before, this method runs in an infinite loop to observe the resources at any time.
2. The first step is the acquirement of an lock object. If the lock object is already locked, the process waits until it is released again to continue after the lock object was locked by this process. This lock object is used by every PolicyThread that is associated to the same service instance. Otherwise, it could happen that two PolicyThreads try to update the same service instance at the same time and this will result in an undesired behavior of the scaling mechanism.
3. The next step will check if it is possible to add a new unit to the service instance considering the current size of the resource group, maximal size of this resource group and the scaling adjustment. If the maximal size would be exceeded, the PolicyThread releases the lock and triggers the sleeping period of this policy before it starts from step 1 again. If the maximal size is not reached, the PolicyThread begins the gathering of the meter values from all the units for the addressed service instance. Therefore, each meter value is requested from the MontioringAgent and is used afterwards to compute a final value for the evaluation process. In this process the computed value is compared with the threshold. If the threshold is not reached, the method jumps directly to the lock release, waits for the policy period and starts from step 1 again. If the threshold is reached, the actual scaling procedure is triggered by the PolicyThread executing the following steps:
    a) Add the specified amount of new units (scaling adjustment) to the topology.
    b) Update the topology in the database done by the DatabaseManger.
    c) Call the TemplateManager to create a new template with the updated topology.
    d) Request the HeatClient to update the associated stack on OpenStack with the new template. The response of that function is returned to the PolicyThread.
    e) Wait until the update process of the stack is finished.
    f) Wait for the cooldown period defined in the policy.
    g) Release the lock object.
    h) Wait for the policy period defined in the policy and restart the procedure from step 1 again.

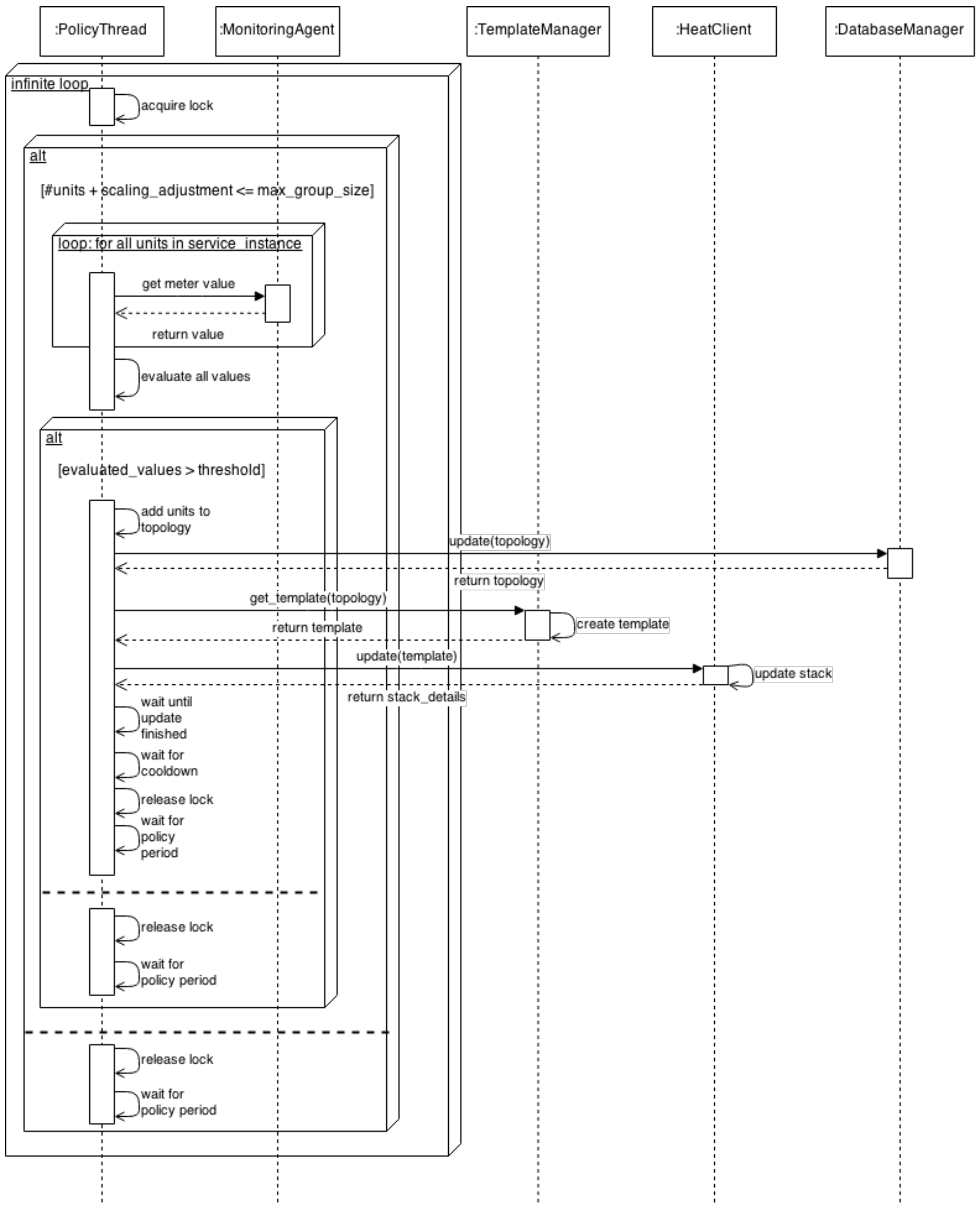


Figure 17. Sequence diagram showing autoscaling mechanisms

6 API

In the following paragraph are described the REST APIs exposed by the EMM.

Method	Path	Body	Description
GET	/topologies	Empty	List all the deployed topologies
POST	/ topologies	Description of the Topology in a JSON format	Create a new topology
GET	/ topologies /<id>	Empty	Display one topology with id <id>
DELETE	/ topologies /<id>	Empty	Delete one topology with id <id>
GET	/services	Empty	List all the available services
POST	/services	Description of the Service in a JSON format	Create a new service
PUT	/services	Description of the updated Service information in a JSON format	Update an existing service
GET	/services/<id>	Empty	Display one service with id <id>
DELETE	/services/<id>	Empty	Delete one service with id <id>
GET	/secgroups	Empty	List all the available security groups
POST	/secgroups	Description of the Security Group in a JSON format	Create a new security group
PUT	/secgroups	Description of the updated Security Group information in a JSON format	Update an existing security group
GET	/secgroups/<id>	Empty	Display one security group with id <id>
DELETE	/secgroups/<id>	Empty	Delete one security group with id <id>

Table 2. REST APIs exposed by EMM

6.1.1 Create topologies

Create a new topology with the provided description in a JSON format. The config file of the user consists of user specific demands for the topology such as name, flavors, key names, service instances, and their setting for instance min size, max size and the policies, etc (See annex 0 for a complete example).
It is possible to deploy several Topologies at the same time.

6.1.2 Get Topologies

Get an overview of all the deployed stacks. It is returned a list of deployed stacks. What information are provided for each stack is described in the next section.

6.1.3 Get Topology by Id

Get an overview of the topology with the given stack id. Here we get information about the deployed Topology. It consists of an general overview of the Topology such as stack id, status, creation time, parameters, outputs (IPs for example).

6.1.4 Delete Topologies

Delete the Topology with the given Topology id from the Virtual Infrastructure and from the database.

6.1.5 Create Service

Create a new Service (see 4.4.4). An example of the JSON file to be sent is shown in the Annex.

6.1.6 Get Services

This API returns the list of all the available Services in the system. The output has the json format.

6.1.7 Delete Service

Calling this API will delete a Service with the specified Id from the list of the available Services.

6.1.8 Update Service

Through this method is possible to update an existing Service with newer values.

6.1.9 Get Service by Id

This API returns the specified Service available in the system or a 404 not found in case no Service was found with the specified Id. The output has the json format.

6.1.10 Create Security Group

This API allows creating a new Security Group that will be stored in the Database and will be available in the following operations. An example of the json file to be sent can be found in annex 0.

6.1.11 Get Security Groups

This API returns a list of Security Groups available in the system. The output has the json format.

6.1.12 Get Security Group by Id

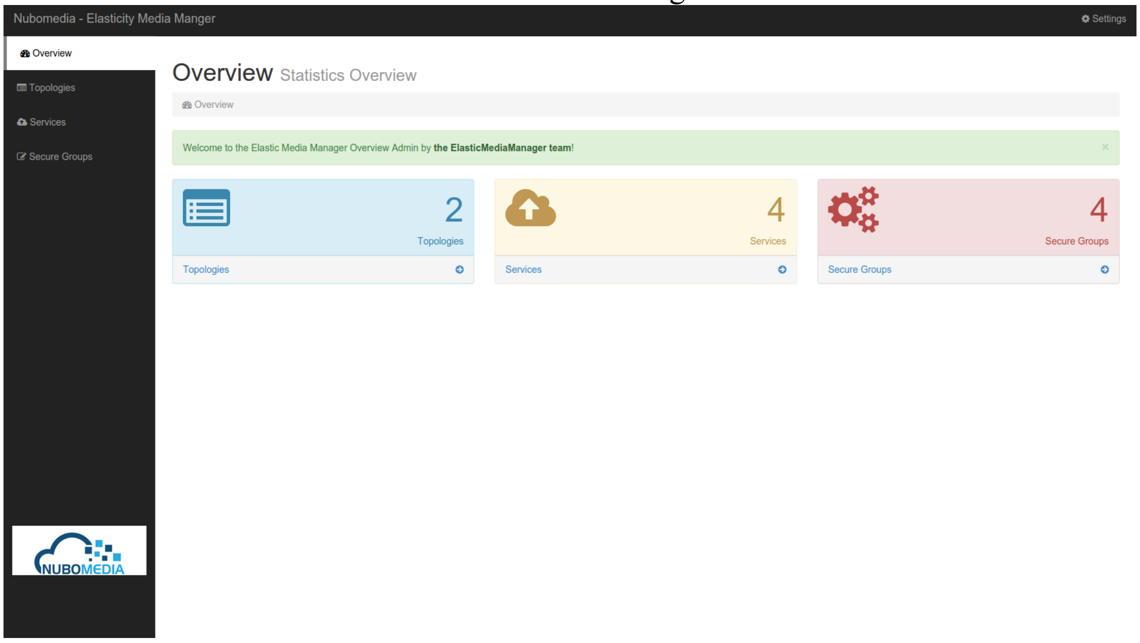
This API return the Security Group with the specified Id or a 404 not found in case there is not a Security Group with the specified Id.

6.1.13 Delete Security Group

Calling this method, the Security Group with the specified Id will be deleted from the list of Security Groups.

6.2 Graphical User Interface (GUI)

The GUI of the NUBOMEDIA Elastic Media Manager is a web base GUI as shown in



Welcome to the Elastic Media Manager Overview Admin by **the ElasticMediaManager team!**

Figure 18 GUI home page

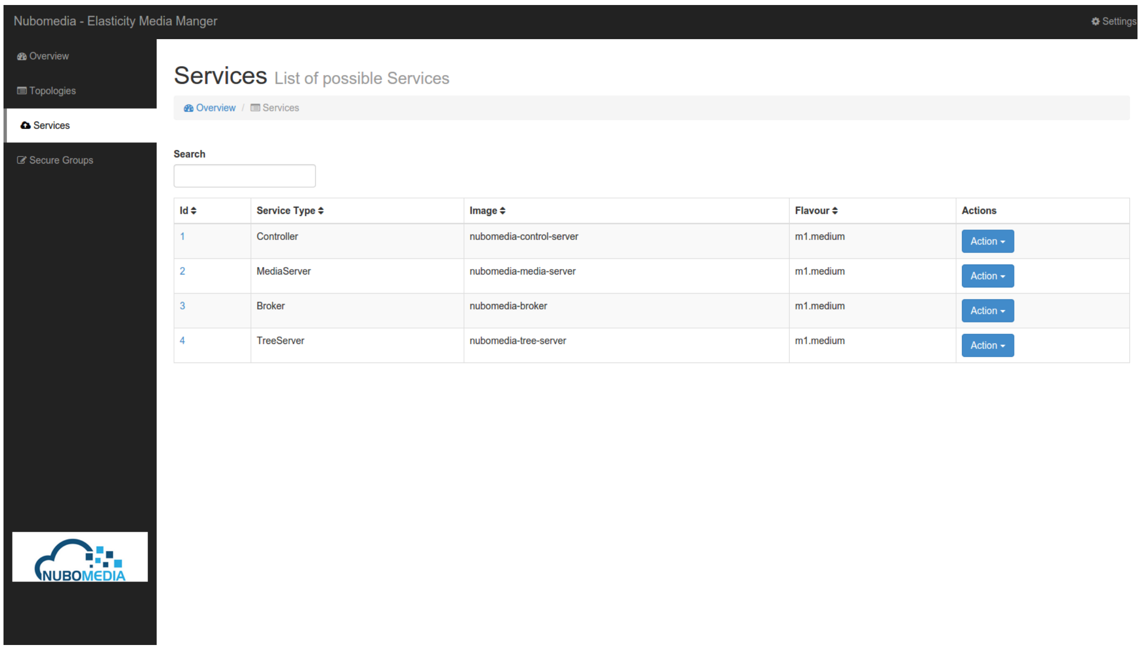
In the home page is given an overview of the system:

- The number of deployed Topologies
- The number of available Services
- The number of deployed Unit

On the menu panel on the left, it is possible to choose the page to be viewed. There is a page for each particular component.

6.3 Services

In the Services page, it is possible to see all the available Services as shown in Figure 19.


Nubomedia - Elasticity Media Manger

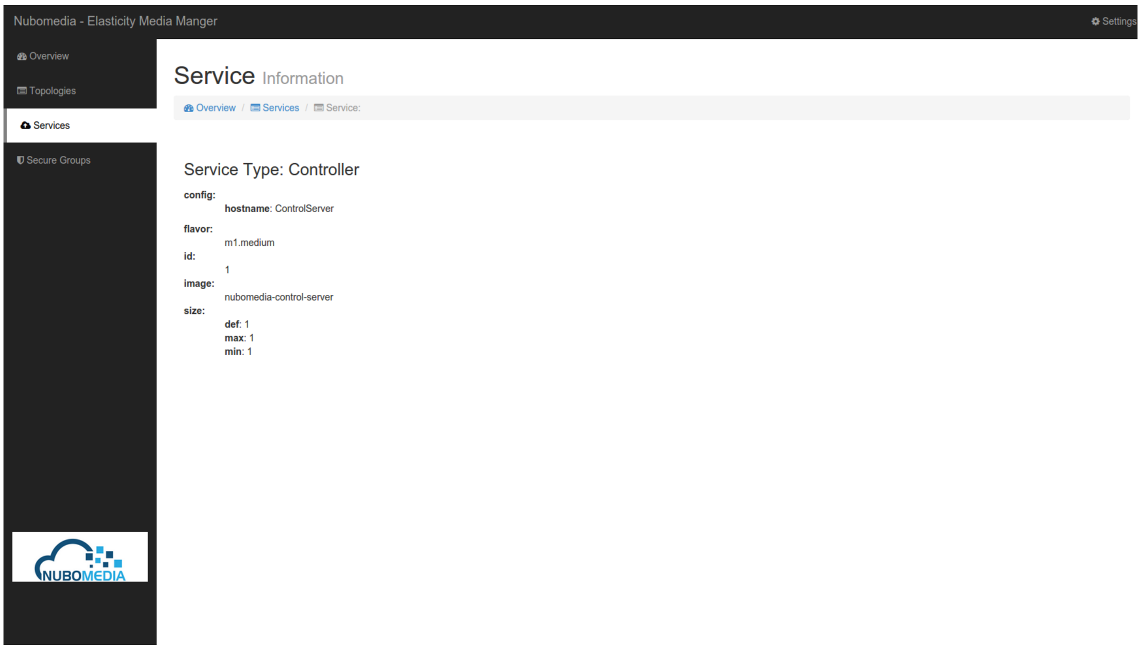
Services List of possible Services

Search

Id	Service Type	Image	Flavour	Actions
1	Controller	nubomedia-control-server	m1.medium	Action
2	MediaServer	nubomedia-media-server	m1.medium	Action
3	Broker	nubomedia-broker	m1.medium	Action
4	TreeServer	nubomedia-tree-server	m1.medium	Action

Figure 19 Services page

Here is also possible to update an already existing Service into the system. Moreover is possible to show all the fields and values of a Service when clicking on the hyperlink on the id, as in Figure 20.


Nubomedia - Elasticity Media Manger

Service Information

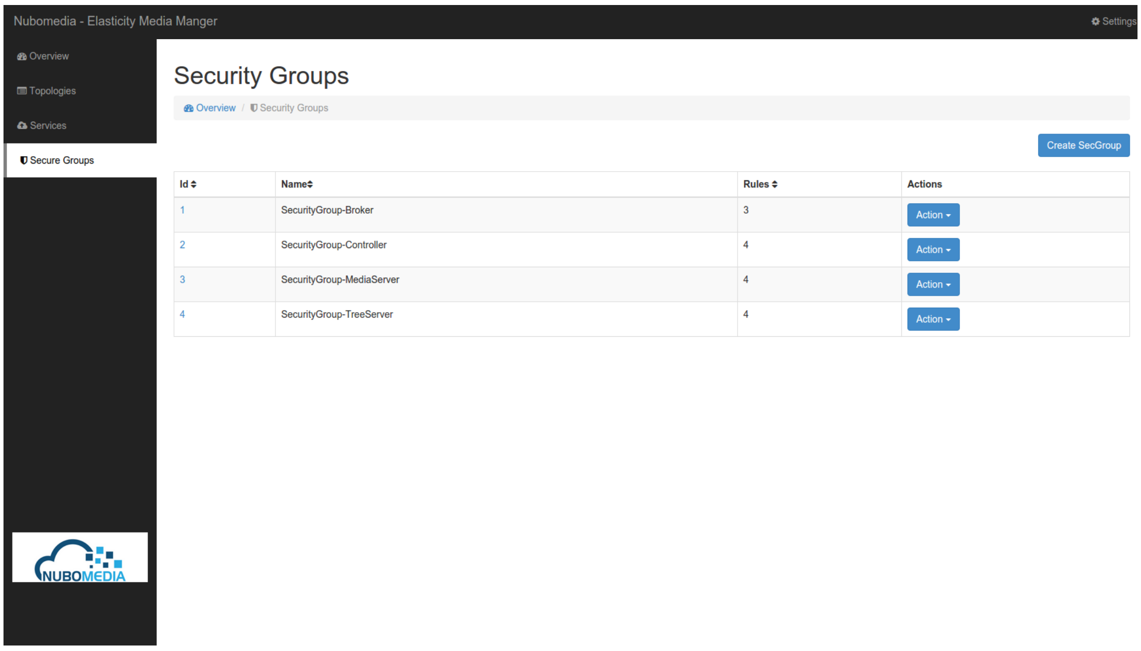
Service Type: Controller

config:
hostname: ControlServer
flavor:
m1.medium
id:
1
image:
nubomedia-control-server
size:
def: 1
max: 1
min: 1

Figure 20 Service details

6.4 Security Groups

In the Security Group page is possible to see all the available Security Group in the system as shown in Figure 21.



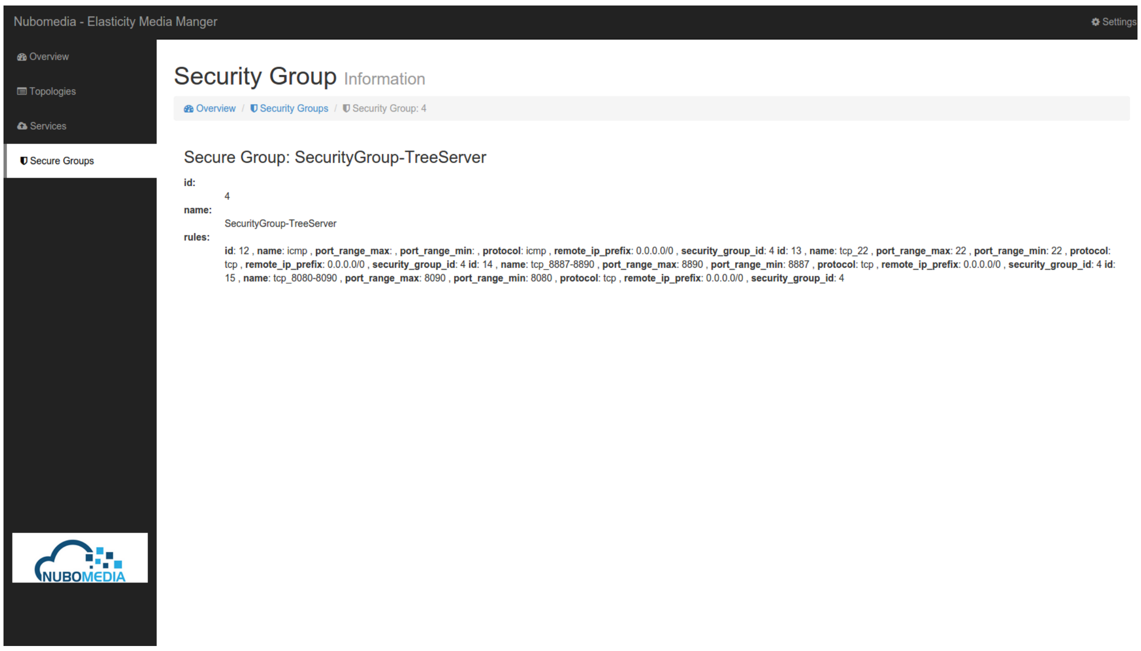
Security Groups

Overview / Security Groups

Id	Name	Rules	Actions
1	SecurityGroup-Broker	3	Action
2	SecurityGroup-Controller	4	Action
3	SecurityGroup-MediaServer	4	Action
4	SecurityGroup-TreeServer	4	Action

Figure 21 Security Groups page

In this page is also possible to update or create a Service Group. The details contained in the Security Group are shown when after have clicked on the hyperlink on the id of a Security Group, as shown in Figure 22.



Security Group Information

Overview / Security Groups / Security Group: 4

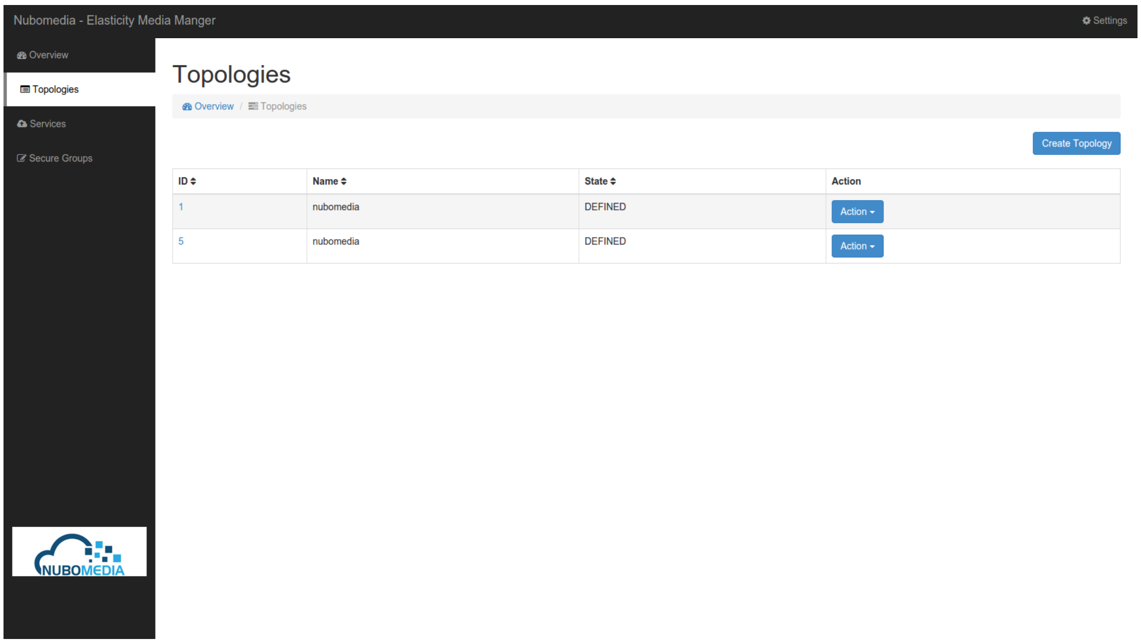
Secure Group: SecurityGroup-TreeServer

id:
4
name:
SecurityGroup-TreeServer
rules:
id: 12 , **name**: icmp , **port_range_max**: , **port_range_min**: , **protocol**: icmp , **remote_ip_prefix**: 0.0.0.0/0 , **security_group_id**: 4 **id**: 13 , **name**: tcp_22 , **port_range_max**: 22 , **port_range_min**: 22 , **protocol**: tcp , **remote_ip_prefix**: 0.0.0.0/0 , **security_group_id**: 4 **id**: 14 , **name**: tcp_8887-8890 , **port_range_max**: 8890 , **port_range_min**: 8887 , **protocol**: tcp , **remote_ip_prefix**: 0.0.0.0/0 , **security_group_id**: 4 **id**: 15 , **name**: tcp_8080-8090 , **port_range_max**: 8090 , **port_range_min**: 8080 , **protocol**: tcp , **remote_ip_prefix**: 0.0.0.0/0 , **security_group_id**: 4

Figure 22 Security Group details

6.5 Topologies

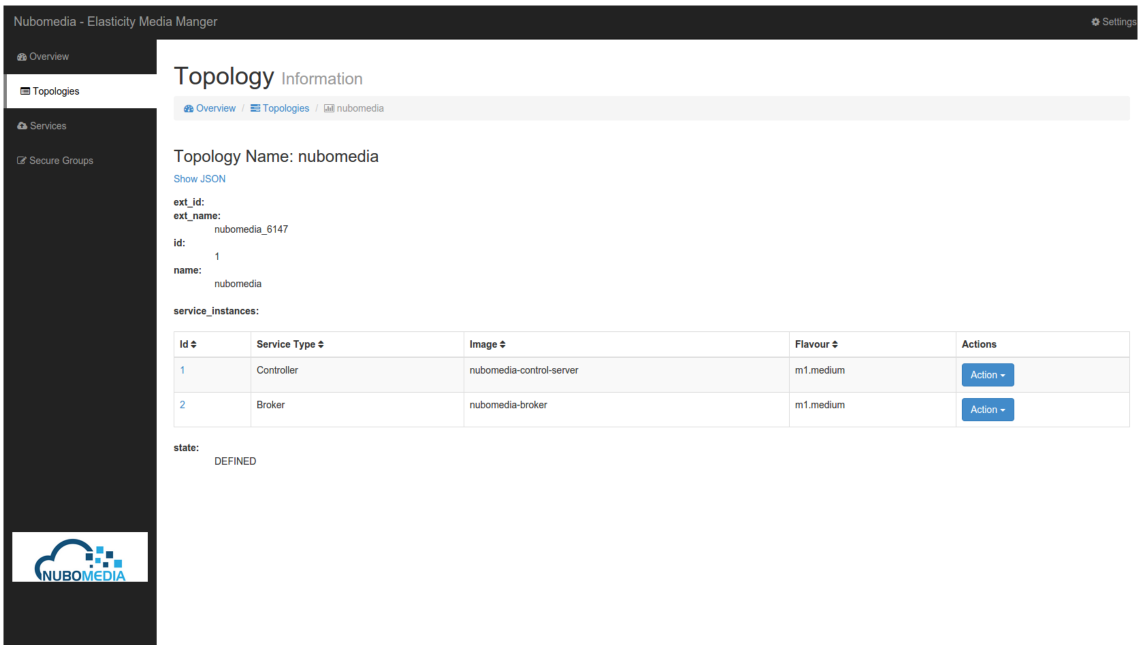
The Topologies page has the task of accessing the already deployed topologies and gives the opportunity to create a new Topology (Figure 23).



ID	Name	State	Action
1	nubomedia	DEFINED	Action
5	nubomedia	DEFINED	Action

Figure 23 Topologies page

Here is possible to delete a Topology, clicking the action button, or see the Topology details, clicking the hyperlink in the Topology id, as shown in Figure 24.



Topology Name: nubomedia

ext_id:
ext_name:
nubomedia_6147
id:
1
name:
nubomedia
service_instances:

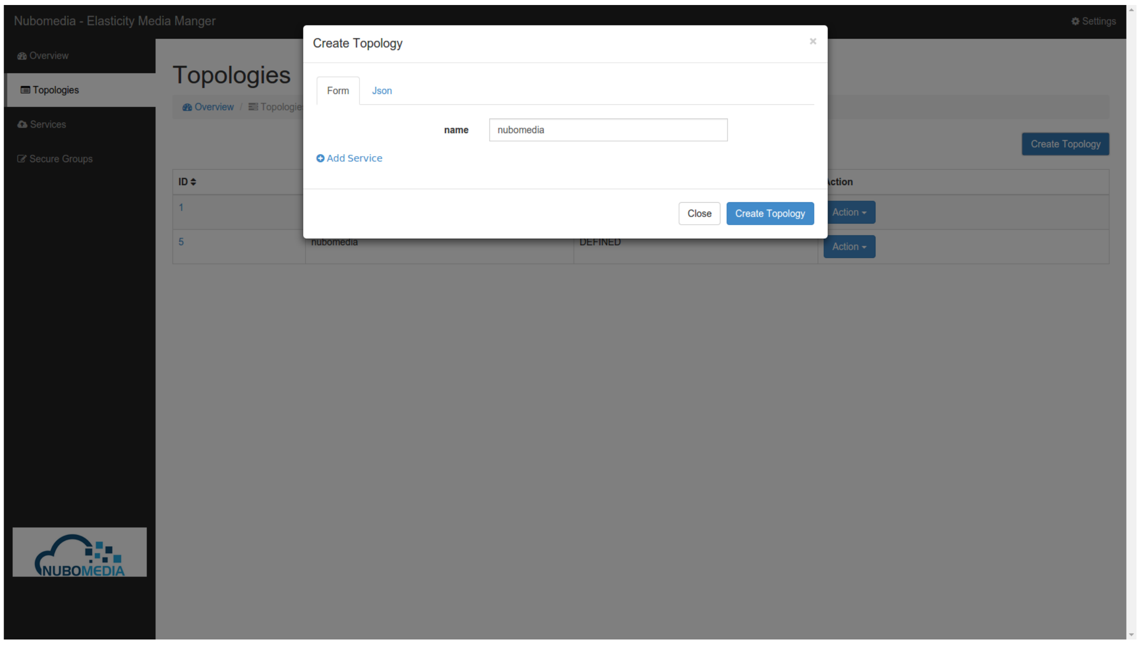
Id	Service Type	Image	Flavour	Actions
1	Controller	nubomedia-control-server	m1.medium	Action
2	Broker	nubomedia-broker	m1.medium	Action

state:
DEFINED

Figure 24 Topology details

6.5.1 Create Topology

Hitting the create Topology button, a new modal will be prompted as the one in Figure 25



Create Topology

Form | Json

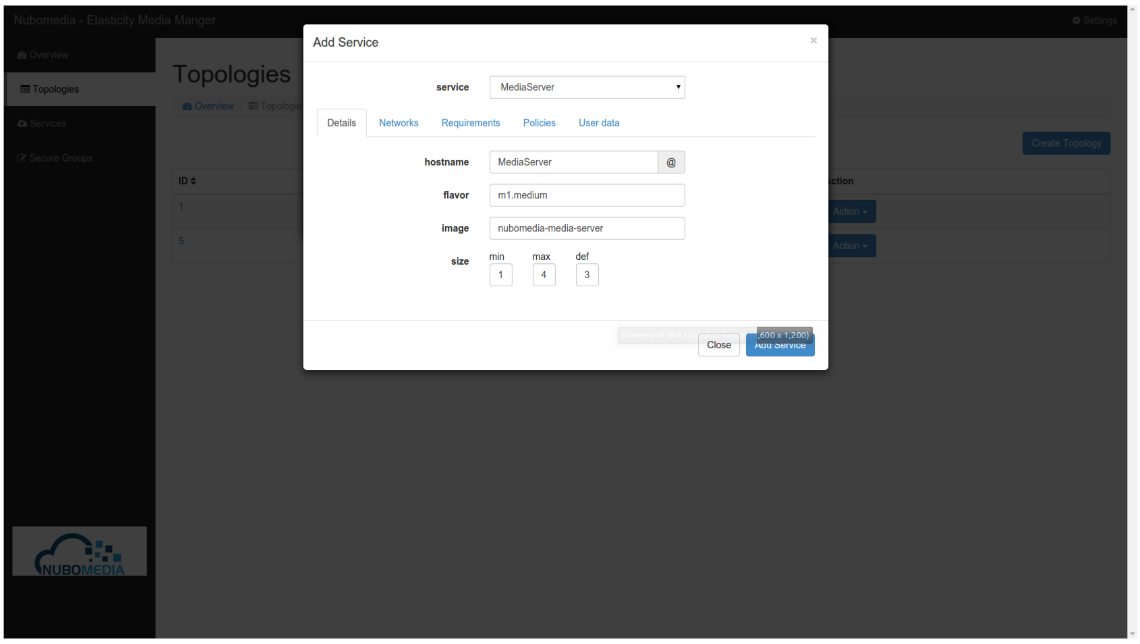
name: nubomedia

Add Service

Close | Create Topology

Figure 25 Create Topology form

In this modal there are two tabs depending the chosen way to create a Topology. It is possible to achieve this filling a form or sending a json file directly. The form fields to be filled will be explained in the following lines.
The name field corresponds to the Topology name. Then it is possible to add Service Instances to the desired Topology by hitting the add service instance hyperlink. In that case another modal will be prompted like the one in Figure 26.



Add Service

service: MediaServer

Details | Networks | Requirements | Policies | User data

hostname: MediaServer

flavor: m1.medium

image: nubomedia-media-server

size: min 1, max 4, def 3

Close | Add Service

Figure 26 Add Service Instance form

Based on the service type chosen in the combo box the fields in the details tab can change and they will be always filled with default values that can be changed in the Services page (Section 6.3). The chosen service type will define the service type of the new Service Instance.
In the Network tab is possible to add network to the desired Service Instance. It is possible to add multiple Networks with or without public access.

In the tab Requirements is possible to create a logic link to another Service Instance already defined in the new Topology.
Once chosen the other Service Instance, it is necessary to fill the name of the parameter (at a later stage usable in the user_data tab) and the parameter to be required.
The Policies tab allows the user to set policies, setting the name the action and the action to be executed with all the subfields necessary.
In the last tab, the user data can be set simply adding commands to be executed during the Cloud Init.
When finished the Service Instance will be added to the new Topology and the "create Topology" button can be clicked (Figure 25). This will call the create topology API and after a while the requested Topology will be shown in the list of the page in Figure 23.

6.6 Accessing and executing the source code

6.6.1 Prerequisites

All the required third party libraries are installed during the installation process, except for the database:

- A MySql server has to be installed and running. It can be installed on the same host were the EMM runs, or remotely.

6.6.2 Installation

The Elastic Media Manager can be either installed on a physical host or a virtual host. The only requirement is that the EMM gets direct connection to the OpenStack Controller node where all the services are running, and which exposes the APIs for being managed.

As first step, the code has to be downloaded in a folder, for instance /opt/nubomedia:

```
git clone
http://git.nubomedia.eu/giuseppe.a.carella/elastic-
media-manager.git /opt/nubomedia
```

Once the code is checked out it needs to be installed. For doing it, there is a script inside the /opt/nubomedia folder that guides through the process of the installation.

```
cd /opt/nubomedia
sudo ./emm.sh install
```

This command will compile and install the server and create the default configurations that have to be changed running the "init" command

```
sudo ./emm.sh init
```

While this command is running you will be prompt to set the OpenStack and database credentials, and after check if the database exists. In case it does not exist, the script will create a new database.
In particular, the requested credentials are:

- Openstack username
- Openstack password
- Openstack auth_url
- Openstack tenant
- Database username
- Database password
- Database name
- Database url

After these configuration steps, it is possible to start the Elastic Media Server with:

```
emm.sh start
```

6.7 Initialization

The “init” command creates some default entries in the database. In particular regarding some default services and some default security groups. It is always possible to add new services and security groups at runtime but in order speed up the initialization process it is possible to change the default values inside the files in the folders:

```
data/json_file/services
```

or

```
data/json_file/security_groups
```

here there are several json files that are sent to the Elastic Media Manager during the start and that contain some default values.

Annex

A JSON file examples

This is an example of a json file that has to be sent to the EMM API module in order to create a topology.

```
{
  "name":"NUBOMEDIA_template_1",
  "service_instances": [
     {
       "name":"Controller",
       "service_type":"Controller",
       "config": {
          "hostname":"ControllerServer",
          "key_name":"NUBOMEDIA"
       },
       "networks": [
          {
             "name":"Network-1",
             "private_net":"fd704f1b-9238-4c2c-a0f5-4ffb4543e33a",
             "private_subnet": "ab4595bf-12d5-4e92-baa8-b5dfb3c1a31d",
             "public_net": "2e2bc7f9-c29c-467c-94b6-5ef3724d79ac",
             "security_groups": [  "SecurityGroup-Controller" ] } ],
       "requirements": [
          {  "name":"$BROKER_IP",
             "parameter":"private_ip",
             "source":"Broker",
             "obj_name": "Network-1" },
          {  "name":"$MediaServer_IPS",
             "parameter":"private_ip",
             "source":"MediaServer",
             "obj_name": "Network-1" }],
       "user_data": [
          "#!/bin/bash -v",
          "HOST_NAME=$(cat /etc/hostname)",
          "sudo sed -i \"s/^\\([0-9]*\\.[0-9]*\\.[0-9]*\\.[0-9]*[ \\t]*localhost\\)/\\1
$HOST_NAME/\" /etc/hosts",
          "sudo sed -i \"s/^\\([ \\t]*\\\"address\\\"[ \\t]*\\):[ \\t]*\\\"Broker private IP
Address\\\"*.*$/\\1: \\\"$BROKER_IP\\\",/\" /etc/kurento/control-server.conf.json",
          "#sudo sed -i \"s/^\\([ \\t]*\\\"address\\\"[ \\t]*\\):[ \\t]\\\"*[0-9]*\\.[0-9]*\\.[0-
9]*\\.[0-9]\\\"*.*$/\\1: \\\"$BROKER_IP\\\",/\" /etc/kurento/kurento.conf.json",
          "sudo service kurento-control-server restart",
          "screen -m -d -S 8081 mvn -X -f /opt/kurento-tutorial-java/kurento-hello-
world/pom.xml compile exec:java -Dserver.port=8081 -
Djava.security.egd=file:/dev/./urandom",
          "screen -m -d -S 8082 mvn -X -f /opt/kurento-tutorial-java/kurento-magic-
mirror/pom.xml compile exec:java -Dserver.port=8082 -
Djava.security.egd=file:/dev/./urandom",
          "screen -m -d -S 8083 mvn -X -f /opt/kurento-tutorial-java/kurento-one2many-call/pom.xml compile exec:java -Dserver.port=8083 -Djava.security.egd=file:/dev/./urandom",
          "screen -m -d -S 8084 mvn -X -f /opt/kurento-tutorial-java/kurento-one2one-call/pom.xml compile exec:java -Dserver.port=8084 -Djava.security.egd=file:/dev/./urandom"
       ]
     },  {
       "name":"Broker",
       "service_type":"Broker",
       "config": {
          "hostname":"BrokerServer",
          "key_name":"NUBOMEDIA"
       },
       "networks": [{
             "name":"Network-1",
             "private_net":"fd704f1b-9238-4c2c-a0f5-4ffb4543e33a",
             "private_subnet": "ab4595bf-12d5-4e92-baa8-b5dfb3c1a31d",
             "public_net": "2e2bc7f9-c29c-467c-94b6-5ef3724d79ac",
             "security_groups": [
                "SecurityGroup-Broker"
             ]
          }
       ],
       "requirements": [
       ],
       "user_data": [
          "#!/bin/bash -v",
          "HOST_NAME=$(cat /etc/hostname)",
          "sudo sed -i \"s/^\\([0-9]*\\.[0-9]*\\.[0-9]*\\.[0-9]*[ \\t]*localhost\\)/\\1 $HOST_NAME/\" /etc/hosts"
       ]
     },
     {
       "name":"MediaServer",
       "service_type":"MediaServer",
       "size": {
          "min":1,
          "max":5,
          "def":2
       },
       "config": {
          "hostname":"MediaServer",
          "key_name":"NUBOMEDIA"
       },
       "networks": [
          {
             "name":"Network-1",
             "private_net":"fd704f1b-9238-4c2c-a0f5-4ffb4543e33a",
             "private_subnet": "ab4595bf-12d5-4e92-baa8-b5dfb3c1a31d",
             "public_net": "2e2bc7f9-c29c-467c-94b6-5ef3724d79ac",
             "security_groups": [
            "SecurityGroup-MediaServer"
          ]
        }
      ],
      "policies": [
        {
          "name": "msg_scaleup",
          "period": 60,
          "alarm": {
            "meter_name": "cpu_util",
            "comparison_operator": "gt",
            "threshold": 50,
            "statistic": "avg",
            "evaluation_periods": 1
          },
          "action": {
            "adjustment_type": "ChangeInCapacity",
            "scaling_adjustment": 1,
            "cooldown": 60
          }
        },
        {
          "name": "msg_scaledown",
          "period": 60,
          "alarm": {
            "meter_name": "cpu_util",
            "comparison_operator": "lt",
            "threshold": 10,
            "statistic": "avg",
            "evaluation_periods": 1
          },
          "action": {
              "adjustment_type": "ChangeInCapacity",
              "scaling_adjustment": -1,
              "cooldown": 60
          }
        }
      ],
      "requirements": [
        {
          "name":"$BROKER_IP",
          "parameter":"private_ip",
          "source":"Broker",
          "obj_name": "Network-1"
        }
      ],
      "user_data": [
        "#!/bin/bash -v",
        "HOST_NAME=$(cat /etc/hostname)",
        "sudo sed -i \"s/^\\([0-9]*\\.[0-9]*\\.[0-9]*\\.[0-9]*[ \\t]*localhost\\)/\\1
$HOST_NAME/\" /etc/hosts",
        "cp /etc/kurento/kurento-broker.conf.json /etc/kurento/kurento.conf.json",
            "sudo  sed  -i  \"s/^\\([  \\t]*\\\"address\\\"[  \\t]*\\):[  \\t]*.*[0-9]*\\.[0-9]*\\.[0-
9]*\\.[0-9]\\\"*.*$/\\1: \\\"$BROKER_IP\\\",/\" /etc/kurento/kurento.conf.json",
            "sudo service kurento-media-server restart"
         ]
      }
   ]
}
```

The following example represents the json file to be sent to the EMM in order to create a new Service.

```
{
  "service_type": "Broker",
  "image": "NUBOMEDIA-broker",
  "flavor": "m1.medium",
  "config": {
     "hostname": "BrokerServer"
  },
  "size": {
     "max": 1,
     "def": 1,
     "min": 1
  }
}
```

The following example represents the json file to be sent to the EMM in order to create a new Security Group.

```
{
   "name": "SecurityGroup-MediaServer",
   "rules": [
      {
         "name": "icmp",
         "protocol": "icmp",
         "remote_ip_prefix": "0.0.0.0/0"
      },
      {
         "name": "tcp_22",
         "protocol": "tcp",
         "remote_ip_prefix": "0.0.0.0/0",
         "port_range_min": "22",
         "port_range_max": "22"
      },
      {
         "name": "tcp_8888",
         "protocol": "tcp",
         "remote_ip_prefix": "0.0.0.0/0",
         "port_range_min": "8888",
         "port_range_max": "8888"
      },
      {
         "name": "udp",
         "protocol": "udp",
         "remote_ip_prefix": "0.0.0.0/0",
         "port_range_min": "1024",
         "port_range_max": "65535"
      }
   ]
}
```

References

[1] Openstack: Open source software for creating public and private clouds. See http://www.openstack.org/.
[2] Heat: Openstack Orchestrator, See https://wiki.openstack.org/wiki/Heat.
[3] Cloudify and Openstack Heat. See http://www.gigaspaces.com/system/files/private/free/resource/Cloudify%20and%20OpenStack%20Heat.pdf.
[4] GigaSpaces Technologies. Cloudify. See http://www.cloudifysource.org/.
[5] Cloudfoundry. Bosh. See http://docs.cloudfoundry.com/docs/running/bosh/.
[6] Openshift Origin, The open source upstream of OpenShift. See https://www.openshift.com/products/origin.
[7] Heat Orchestration Template Guide, Resource Types. http://docs.openstack.org/developer/heat/template_guide/openstack.html#OS::Neutron::SecurityGroup.
[8] Celiometer OpenStack Telemetry . See https://wiki.openstack.org/wiki/Ceilometer.