D3.3

Version	1.1
Author	TUB
Dissemination	PU
Date	04/05/2017
Status	Final

D3.3 Cloud Platform v3

Project acronym:	**NUBOMEDIA**
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2017-01-30
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	11/2016
Actual delivery date	04/05/2017
Keywords	Elasticity, Cloud Computing, NFV, SDN, Management and Orchestration, Monitoring

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:

Alin Calinciuc (USV)
Giuseppe Carella (TUB)
Alice Chaembe (FOKUS)
Flavio Murgia (FOKUS)
Luis Lopez (URJC)
Michael Pauls (TUB)
Cristian Spoiala (USV)
Lorenzo Tomasini (TUB)

Internal Reviewer(s):

Michael Pauls (TUB)

Version History

Version	Date	Authors	Comments
0.1	09.2016	Giuseppe Carella	Initial version
0.2	09.2016	Giuseppe Carella	Added content in some section
0.3	10.2016	Giuseppe Carella, Michael Pauls, Lorenzo Tomasini, Flavio Murgia, Alice Cheambe	Added content in most of the sections
0.4	11.2016	Giuseppe Carella, Michael Pauls	Added evaluation section
1.0	11.2016	Giuseppe Carella	Final version
1.1	04.2017	Giuseppe Carella, Michael Pauls, Alin Calinciuc	Improved quality of the document based on the feedbacks received by reviewers

Table of contents

List of Figures:

Acronyms and abbreviations:

API	Application Programming Interface
AS	Application Server (refers to function instance)
ASS	Application Server Services
CFM	Cloud Functions Manager
CMA	Connectivity Manager Agent
CN	Compute Node
CPU	Central Processing Unit
EMS	Element Management System
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
ME	Media Element
ML2	Multi Layer 2
MP	Media Pipeline
MS	Media Server (refers to function instance)
NAI	NUBOMEDIA Autonomous Installer
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NFVO	NFV Orchestrator
NS	Network Service
NSD	Network Service Descriptor
NSR	Network Service Record
PA	Platform Application
PaaS	Platform as a Service
PAL	Platform Application Logic
PNF	Physical Network Function
QoS	Quality of Services
RFC	Request For Comments
RTC	Real-Time Communications
SaaS	Software as a Service
SLA	Service Level Agreement
UE	User Equipment
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFM	VNF Manager
VNFR	Virtual Network Function Record
VXLAN	Virtual Extensible LAN
WS	WebSocket
WWW	World Wide Web

1 Executive summary

This document stands as a public report including all contributions until the latest available Release of NUBOMEDIA (Release 9) and constitutes Deliverable 3.3, providing the final details about the Cloud Platform components and their integration from WP3 perspective into the NUBOMEDIA Architecture defined in D2.4.3. It reports a detailed description of the Cloud Platform Functional Architecture and its Software implementation. Furthermore, it provides some details about changes realized to the first version of its Architecture (already published with Deliverable D3.2) for supporting new requirements coming from the NUBOMEDIA developer community collected in D2.3. Those requirements have been analyzed and a solution/feature has been provided for most of them.

To summarize, the final version of the NUBOMEDIA Cloud Platform has been composed by the following components:

- A Platform as a Service (PaaS) allowing the deployment and provisioning of several type of applications and supporting services. It allows developers to benefit from scalability and elasticity mechanisms provided at each level (data and signaling plane).
- An ETSI NFV [1] compliant framework for the Management and Orchestration of Multimedia Network Functions have been designed and implemented. This framework allows the instantiation of multiple multimedia Slices on the same Physical Infrastructure.
- A Connectivity Manager engine able to provide different level of QoS to deployed applications.
- A powerful monitoring system capable of providing metrics from each level, Application, Media Elements, and Infrastructure have been designed. Several existing monitoring solutions have been evaluated, and one of them has been selected and integrated in the Cloud Platform.

Overall, this work-package has implemented and integrated together numerous components, offering a rich set of features to the application developers providing the foundation for a PaaS of Multimedia applications. Each component has been implemented using an Agile development methodology. Testing driven development allowed a smooth development process. Using standardized interfaces reduced the complexity of integration among all components.

2 Introduction to the NUBOMEDIA Functional architecture

The main goal of WP3 is to design and develop a Cloud Platform supporting the on-demand deployment of Multimedia Applications. In this section will be briefly give a recap of the NUBOMEDIA Architecture as already presented in D2.4.2 [2] and D2.4.3[2], and will be highlighted some of the changes made during the last releases of the platform for supporting the requirements coming from the NUBOMEDIA developer community collected in D2.3.3.

2.1 Requirement analysis

In this section, it is given a summary of the requirements which have been identified in WP2, particularly D2.3.3. This list has been considered in the last year of the project driving the development of new features. Some of them have been implemented, while some others have been rejected either because of the large amount of work needed for supporting them or because an alternative solution was already available.

In particular, starting from the list of installation requirements, we had:

- IST_R1: Enable full installation of NUBOMEDIA on top of virtual computing resources. This requirement has been considered in the context of the Autonomous Installer and further described in Section 4.1.

From the list of debugging requirements:

- DBG_R1: Access to the environment where the application/media server is executing. This requirement has been rejected as similar to requirement DRT_R3.
- DBG_R2: Simplify media server logs has been considered and implemented, so now all developers have all the media-server logs available on the Debug section of their application on the PaaS GUI.
- DBG_R3: Graphical tools for filtering the logs. This requirement has been considered in the context of the NUBOMEDIA Monitoring tools.

From the list of deployment & runtime settings requirements:

- DRT_R1: Control Specific KMS Instance. This requirement has been considered in the context of the NUBOMEDIA Media Plane and further described in Section 3.3.
- DRT_R2: Dynamically scale out limit settings. This requirement has been considered in the context of the NUBOMEDIA Media Plane, however it has not been considered for implementation as required several changes on many components. Further information is given in Section 3.3.
- DRT_R3: SSH into media server instances. This requirement has been considered in the context of the NUBOMEDIA PaaS and NUBOMEDIA and further described in Section 3.4.
- DRT_R4: Deploy additional services together with an application. This requirement has been considered in the context of the NUBOMEDIA PaaS and further described in Section 3.4.
- DRT_R5: Deploy Apps of any language. This requirement has been considered in the context of the NUBOMEDIA PaaS, however no further actions has been taken as the NUBOMEDIA PaaS support by default several programming languages being based on container technologies.
- DRT_R6: Pool size runtime settings. This requirement has been considered in the context of the NUBOMEDIA Media Plane and further described in Section 3.3.

- DRT_R7: Display App deployment time. This requirement has been considered in the context of the NUBOMEDIA PaaS and further described in Section 3.4.
- DRT_R8: Restart application. This requirement has been considered in the context of the NUBOMEDIA PaaS and further described in Section 3.4.
- DRT_R9: Redeploy Apps. This requirement has been considered in the context of the NUBOMEDIA PaaS and further described in Section 3.4.
- DRT_R10: KMS Buildpacks for custom module installation. This requirement has been on our backlog but hasn't been moved into selected for development because we didn't have any application needing this kind of feature in the context of NUBOMEDIA.
- DRT_R11: Provide billing capabilities. This requirement has been considered in the context of the NUBOMEDIA PaaS, however no further actions have been taken as it would require the implementation of several additional technologies. It may be considered for future releases of the platform.
- DRT_R12: Provide fault tolerance capabilities. This requirement has been considered in the context of the NUBOMEDIA Media Plane, however no further actions have been taken as it was not required by the NUBOMEDIA Demonstrators. Anyway, an analysis of a possible implementation solution has been given in Section 3.3.
- DRT_R13: Geo-aware multi-site. This requirement has been considered in the context of the NUBOMEDIA Media Plane. Even though the implementation may have been straight forward as Open Baton provides multi-site deployment capabilities, it has not been satisfied as at the moment NUBOMEDIA is implemented in a single testbed location.

From the list of PaaS GUI requirements:

- PGUI_R1: default autoscaling behavior and units are obscure and not specified.
- PGUI_R2: scaleInOut parameter name in PaaS GUI is obscure.
- PGUI_R3: coolDownPeriod should also be configurable parameter in the PaaS GUI in relation to autoscaling behavior.
- PGUI_R4: scaleInThreshold should also be configurable parameter in the PaaS GUI in relation to autoscaling behavior.
- PGUI_R5: PaaS GUI does not specify scaling parameters of a deployed application.
- PGUI_R6: Monitor machines memory, media pipelines and media elements of a media-server.
- PGUI_R7: Waiting notification in PaaS GUI.
- PGUI_R8: Deployment status should be shown in application tab.

All those requirements have been considered in the context of the NUBOMEDIA PaaS component. In particular, the PaaS GUI has been redesigned in order to simplify the interaction between the Application developers and the NUBOMEDIA platform. Furthermore, the guide on how the autoscaling mechanism works has been improved and it's not part of the NUBOMEDIA developer guide[1]. More details are provided in the NUBOMEDIA PaaS Section 3.4.

2.2 Functional Architecture Overview

The NUBOMEDIA high-level architecture is depicted in Figure 1. This architecture complies with the ETSI NFV MANO specification [MANO]. Details about each of the

[1] http://nubomedia.readthedocs.io/en/latest/paas/autoscaling/

NUBOMEDIA components are provided in sections below. At a very high-level perspective, and following top down approach, these are the core functions:

- NUBOMEDIA PaaS being the intermediate level between the infrastructural resources and the users of the platform. Particularly it includes the NUBOMEDIA PaaS Manager exposing the iD interface to the developers for deploying their applications. Due to this, this interface is also called the PaaS Manager API along this document. This level also holds the NUBOMEDIA PaaS, a System hosting the applications and exposing their capabilities to the End-Users through the iS interface, which is also called the NUBOMEDIA Signaling interface or just Signaling interface, along this document.
- NUBOMEDIA Media Plane composed by the media plane capabilities (Media Servers and Cloud Repository) whose lifecycle is managed by the Network Function Virtualization Orchestrator (NFVO) and Virtual Network Function Manager (VNFM) following the guidelines of the ETSI NFV specification for the virtualization of Network Functions. Among the media server capabilities that are exposed to the external world we can find media transports, which are represented through the iM interface.
- NUBOMEDIA IaaS composed by the infrastructure resources in terms of Compute Nodes (CN) providing virtual compute, storage and networking resources to the upper layers, and the Virtual Infrastructure Manager (VIM) providing the abstracted APIs for controlling their lifecycle.

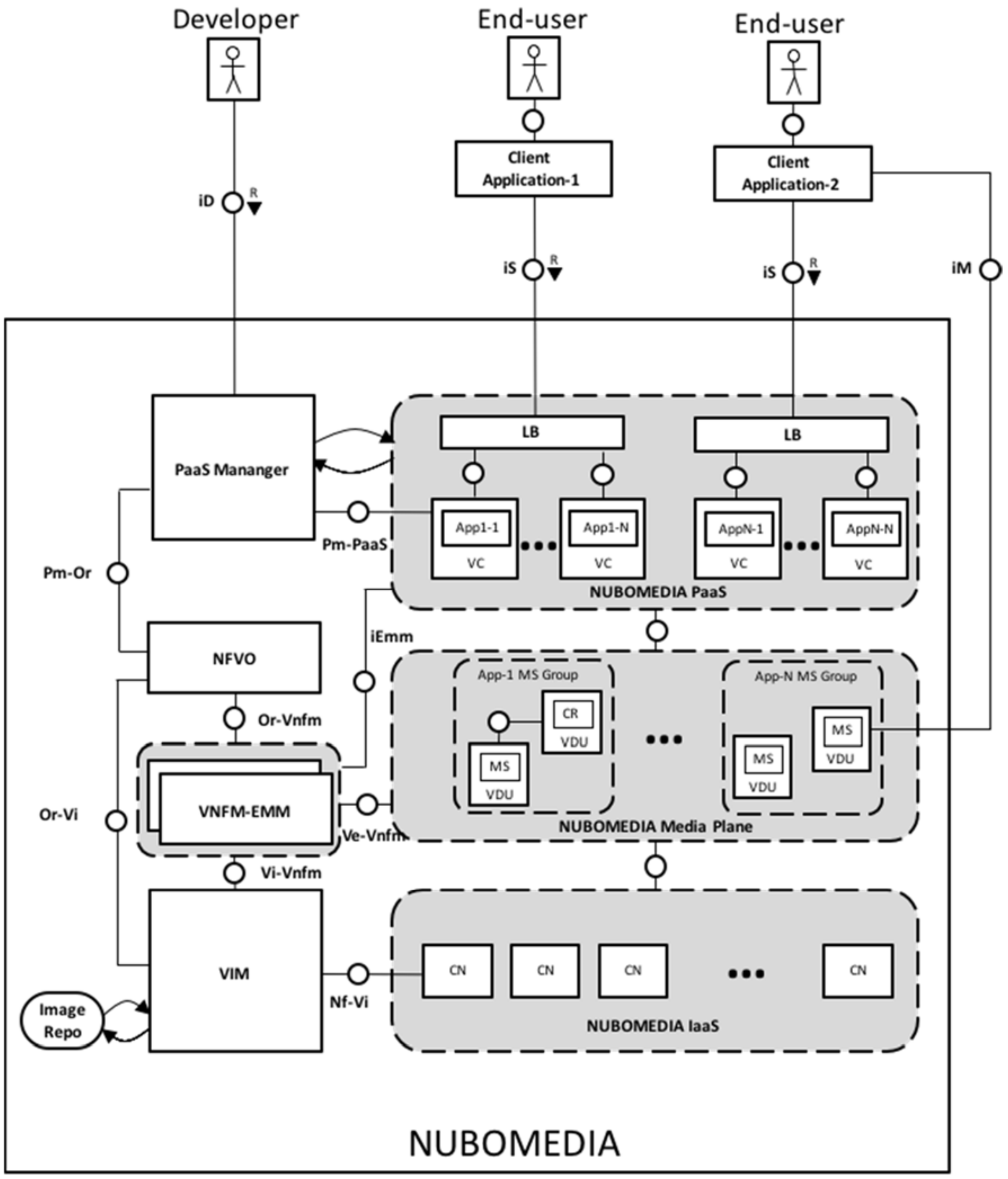


Figure 1. This figure shows the NUBOMEDIA system architecture following the Fundamental Modeling Concepts (FMC) block diagram visual notation. The NUBOMEDIA architecture is based on the ETSI NFV Management and Orchestration (MANO) recommendation but extends it with a PaaS layer. Thanks to it, when developers deploy their applications through a PaaS API, the PaaS Manager orchestrates all the required actions for the provisioning of the appropriate resources and services required for applications to run in a reliable and scalable way.

Application deployment overview

First of all, the main objective of NUBOMEDIA is to build a Cloud Platform exposing capabilities for dynamically manage and orchestrate the different tiers of a multimedia application. The NUBOMEDIA three tier model architecture introduced in Figure 1 of D2.4.3 depicts the different layers composing an application. The NUBOMEDIA PaaS system designed and developed in the context of the NUBOMEDIA project, provides support for managing and orchestrating dynamically the lower two layers of that architecture, meaning the Media Services and Application Services. Therefore, when referring to an “application”, we assume always the combination between those two layers.

As already mentioned, the application logic implemented by NUBOMEDIA application developers needs to be executed on top of an application servers, previously defined Application Services layer, in order to be fully functional. Media elements, executing in the Media Services layer, shall be available and reachable from the Application Service layer. The Application Services are managed by the PaaS system, while the Media Services are managed by the NFV framework

The iD interface (depicted in Figure 1) can be consumed by NUBOMEDIA application developers to deploy applications on top of the NUBOMEDIA Cloud Platform. Those interface primitives are designed for capturing all the information that are necessary for the deployment of the Media and Application services. First of all, developers can specify information related with the Media Service layer (i.e. how many instances of media elements are required runtime, the autoscaling policies, STUN and TURN server addresses). Secondly developers could provide information about the application itself (i.e. the Git repository URL where to find the software artifacts of the application, required support services like DBMS, application name and desired number of replicas). Both the Media and Application Services layer components are executed on top of containers. After building the information model of its application, a developer can request to the PaaS Manager to deploy its application via the iD interface. Figure 2 provides a high-level overview of a typical application managed by the NUBOMEDIA Cloud Platform.

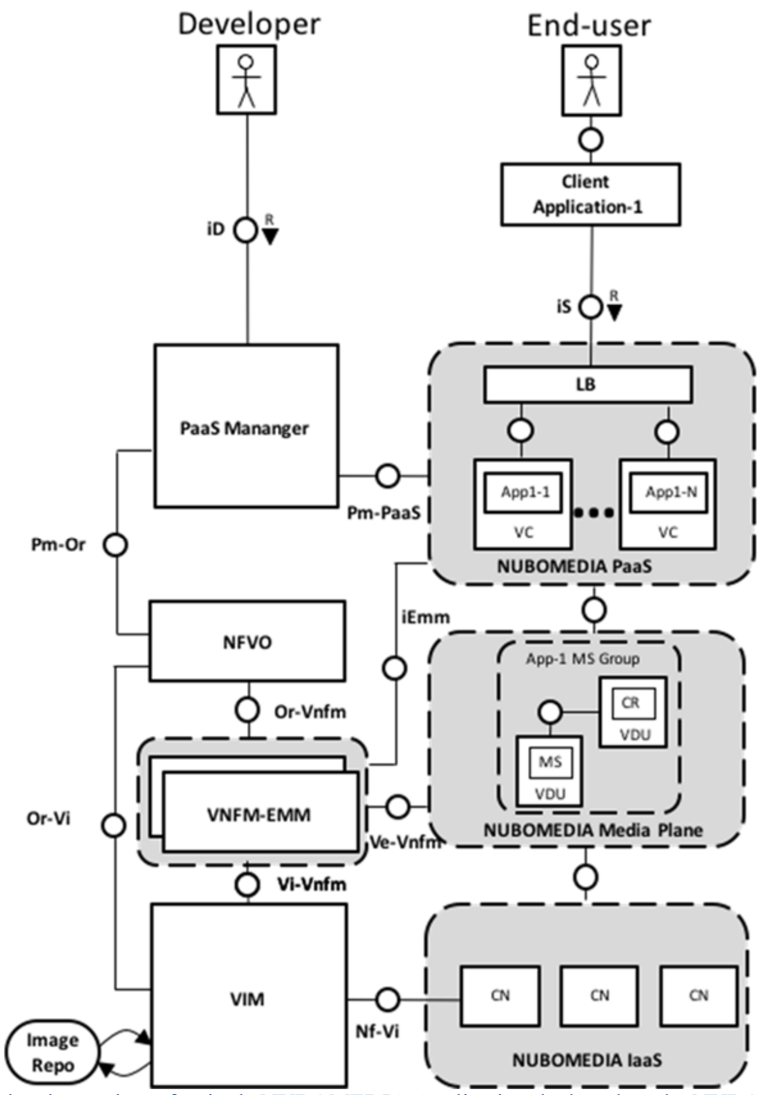


Figure 2. High-level overview of a single NUBOMEDIA Application deployed on the NUBOMEDIA Cloud Platform. The Application logic and Supporting Services (DBMS, etc) are deployed on the PaaS system, while the Media Service components are deployed on the IaaS using the NFV layers.

Following a bottom-up approach, the NUBOMEDIA IaaS provides via the Virtual Infrastructure Manager (VIM) virtual compute, network, and storage resources on demand hosted on top of Compute Nodes (CN). Those resources are consumed by the NFV layer, taking care of building up the NUBOMEDIA Media Plane, hosting the Media Service layer. The NFV components deploys the required virtual resources and configure the Media Service entities required by the application's developer. Those Media Service entities, typically media server instances, are executed on top of virtual containers, and the on-demand discovery is achieved via the iEmm interface.

The Application Service layer is composed by one or more containers providing the Application logic and (optionally) one or more Supporting Services (like DBMS, etc.) hosted by the PaaS System. The Software artifacts comprising the application logic are packaged by developers as container images (typically Dockerfile) and their execution is managed by the PaaS system on top of Virtual Containers (VC). In order to retrieve dynamically information about the Media Service layer, the PaaS Manager injects runtime information (like the endpoint of the VNFM-EMM and additional unique identifiers) inside the containers executing the application logic as environment variables.

For scaling the Application Service layer, a HTTP Load Balancer is deployed seamlessly by the NUBOMEDIA PaaS each time an application is deployed. More details on those procedures are going to be provided in the following sections.

Application consumption overview
Once an application has been deployed on top of the NUBOMEDIA Cloud Platform, developers receive an endpoint which could be provided to end-users for allowing them consuming the services via the iS interface. Through this interface, signaling messages arrive to the NUBOMEDIA PaaS hosting all the deployed applications. As already mentioned, a Load Balancer module performs message routing to the appropriate application instance, where the specific message semantics is executed. For this to happen, the PaaS Manager component controls, in runtime, the lifecycle of application containers so that they are instantiated and removed accordingly to end-user scaling needs.

Application runtime autoscaling overview
As already mentioned in the previous paragraph, the runtime information related to the available media server instances are provided to the upper Application Service layer dynamically. An overview of the mechanism designed for autoscaling in and out the Media Service layer is shown in Figure 3.

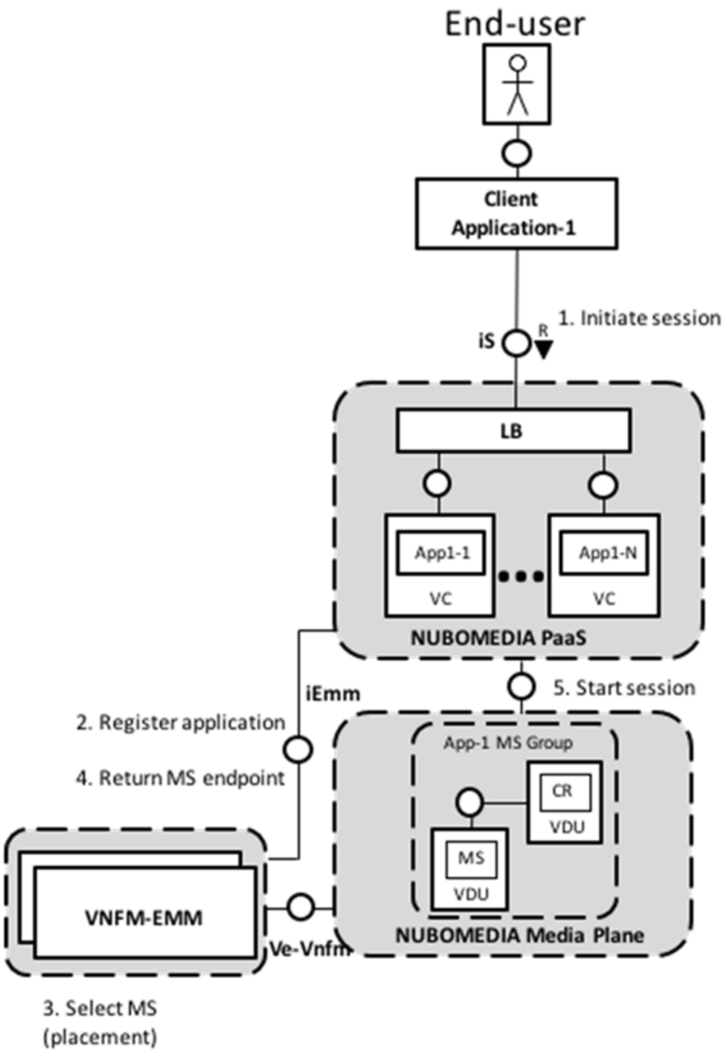


Figure 3. High level overview of the mechanism designed for auto-scaling the Media Service Layer

The Client initiates a session (step 1) via the iS interface based on the end-user request (via web browser or mobile application). In order to instantiate the required media capabilities, the application logic requires the endpoint of one media server instance. This operation is achieved consuming the iEmm interface provided by the VNFM-EMM component. In particular, the application sends a registration request (step 2) to the VNFM-EMM, which internally selects (step 3) which MS instance to use. This operation, is called session placement, because the VNFM-EMM collects information about the resource usage of the Media Plane components and returns the best suitable MS instance for the required session. Further details about this operation are provided in following sections. Once the MS instance has been selected, the VNFM-EMM returns the MS endpoint (step 4) to the application. At that point, based on the application logic, the sessions can be started (step 5) using the media capabilities provided by the MS instance.

This mechanism allows the dynamic discovery of MS instances, therefore whenever the VNFM-EMM realizes that there are no more resources available for instantiating new sessions, it starts automatically scaling out new MS instances. This operation is completely seamless to the application developers, since the iEmm interface is embedded in Software Development Kits (SDKs) and APIs provided to developers by the NUBOMEDIA framework.

2.3 NUBOMEDIA modules

2.3.1 The PaaS Manager

The PaaS Manager is the central part of the system enabling developers to deploy and manage the lifecycle of their applications. Developers could deploy and scale on demand their applications using the PaaS Manager without having to manually manage and configure each layer of their multimedia applications. The PaaS Manager is the component in charge of the full lifecycle of an application hosted on the PaaS, and its required media functions, hosted on the NUBOMEDIA Media Plane, and NUBOMEDIA IaaS.

The main set of functionalities provided by this component are:

- Deployment of Application Service and Supporting Services
- Runtime management of Applications
- Inventory of existing Applications
- Deployment of required Media Service capabilities (via the NFV/IaaS layers)
- Control of the lifecycle of Applications
- User authentication and authorization

Basically, the PaaS Manager is an intermediate component combining the functionalities provided by the NUBOMEDIA PaaS System and the NFV/IaaS layers. Each time a developer requests the instantiation of an application, the PaaS Manager firstly requests the deployment of a set of Virtualized Network Functions (VNFs), defined as Network Service (refer to section 2.3.3 for more details), then instantiates the application on the PaaS System providing all the details of the media functions that are dedicated to it. As shown in Figure 4, it comprises six main modules which are going to be further detailed in upcoming sections.

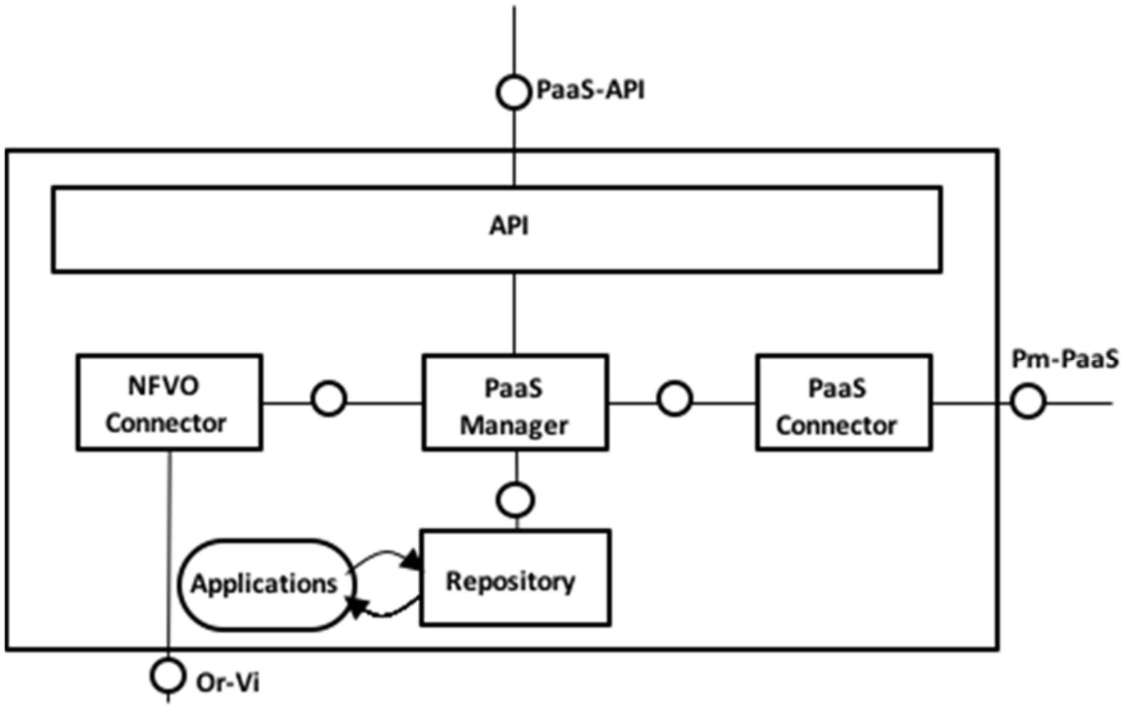


Figure 4. PaaS Manager internal architecture.

2.3.1.1 Application deployment scenario

As depicted in Figure 5, the PaaS Manager provides four main functionalities to the user interacting with NUBOMEDIA.

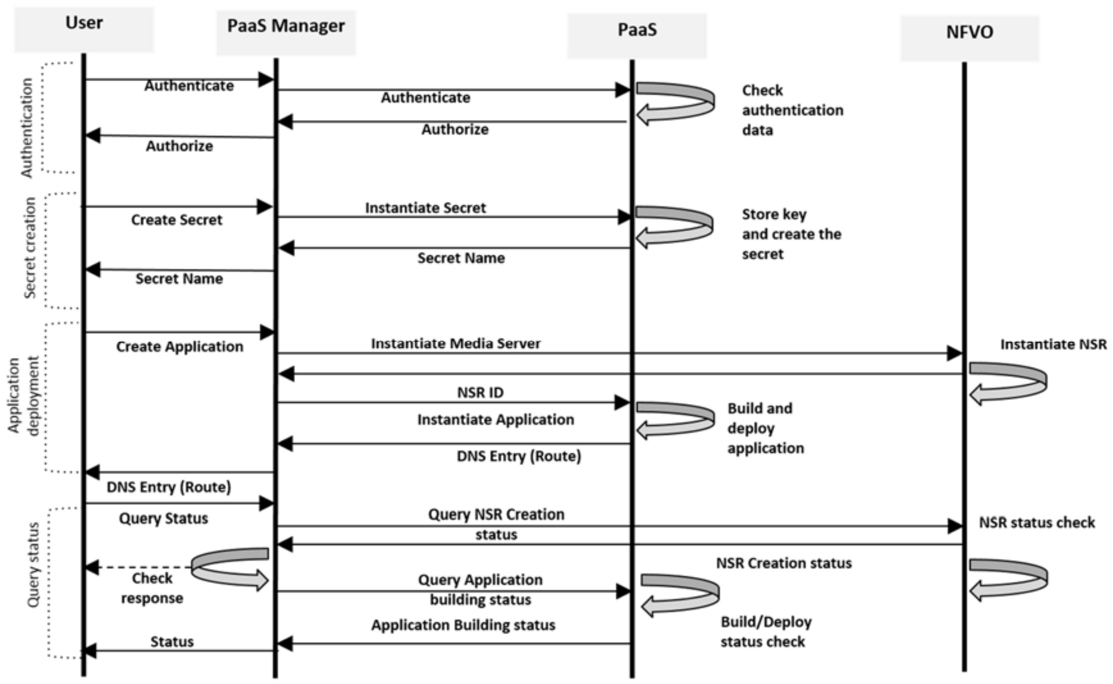


Figure 5. PaaS Manager Interactions

Authentication: Before the user can interact with the PaaS, it needs an authorized token for using the PaaS API. For this purpose, users can use the PaaS Manager API to authenticate themselves on the PaaS. With the response, a token is sent to the user which will be used in subsequent request in the duration of the session.

Secret creation (optional): The Secret object provides a mechanism to hold sensitive information such as passwords, PaaS configuration files, configuration files and private source repository credentials. The User can create a secret data object and send a request via the PaaS Manager to the PaaS requesting storage of the secret data. The PaaS stores the secret data and generates a secret name to identify this data, which is sent back to the PaaS Manager, which forwards this to the user. An example where this is applicable is when developers use private repositories for their applications. They can then create a secret object with the credentials to access this private repository. This information is used in the application deployment phase.

Application deployment: To deploy an application, the user creates a JSON object containing the meta-data of the application as described in section 4.3.1.3. Thereafter the PaaS manager orchestrates a series of requests to the NFVO component for deploying the Network Service containing the media service capabilities required, and a set of requests to the PaaS for deploying (using application meta-data and secret data object if required) the application logic on a specific Application Server. With successful deployments, the user is provided a route pointing to a DNS entry from which the application can be reached, otherwise an error message.

Query deployment status: Depending on the application specific deployment configuration requirements, the deployment procedure on the PaaS and on the media plane could take a few seconds or longer. For long procedures, the user always has the option to query the deployment status via the PaaS Manager REST interface.

Looking deeply into the Application deployment, Figure 6, developers have to make a HTTP Post Request (via curl or the NUBOMEDAI PaaS GUI) providing a JSON object as body (details about the APIs are provided in Section 2.3.1.3). The PaaS manager will get that parameters from API and dispatch them to NFVO and PaaS through connectors.

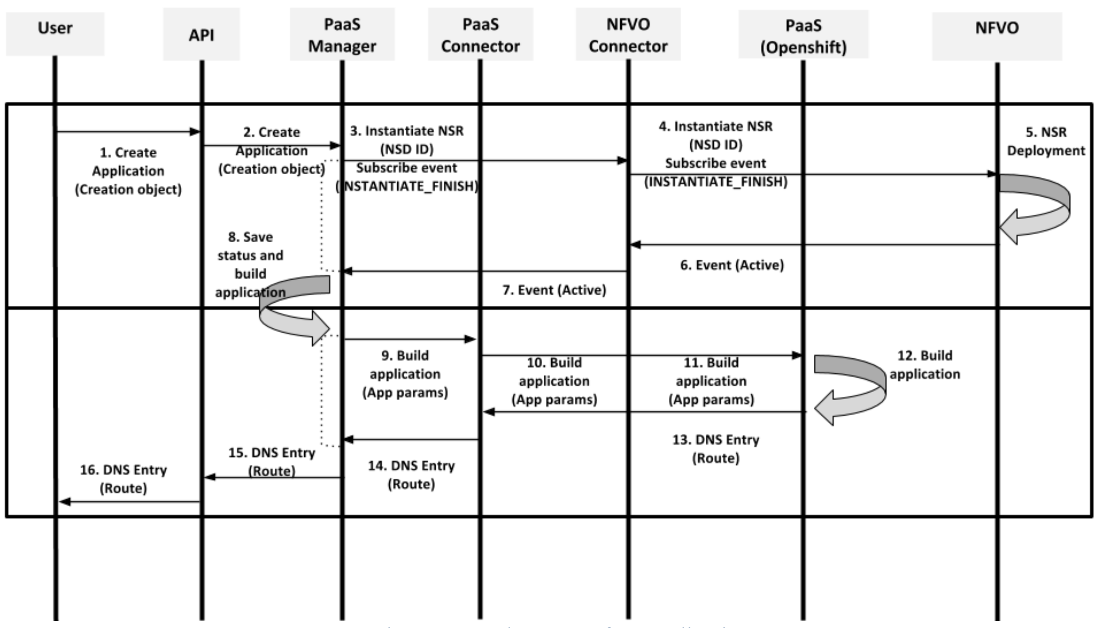

User
API
PaaS Manager
PaaS Connector
NFVO Connector
PaaS (Openshift)
NFVO
1. Create Application (Creation object)
2. Create Application (Creation object)
3. Instantiate NSR (NSD ID) Subscribe event (INSTANTIATE_FINISH)
4. Instantiate NSR (NSD ID) Subscribe event (INSTANTIATE_FINISH)
5. NSR Deployment
6. Event (Active)
7. Event (Active)
8. Save status and build application
9. Build application (App params)
10. Build application (App params)
11. Build application (App params)
12. Build application
13. DNS Entry (Route)
14. DNS Entry (Route)
15. DNS Entry (Route)
16. DNS Entry (Route)

Figure 6. Deployment of an application

2.3.1.2 The PaaS Manager internal modules

The PaaS Manager internal module (let us qualify it as "internal", for distinguishing from the container high-level module) provides the orchestration logic for application lifecycle management. It gives semantics to requests received by the PaaS API translating them into sequences of operations.

Repository
The NUBOMEDIA repository allows to store and manage media and associated metadata in a persistent manner (e.g. using the file system or a DBMS). The repository interoperates with the persistent storage that store application meta-data and provides CRUD operations. The metadata of the deployed applications include:

- Application identifier
- Application name
- Network service record identifier
- URL to the Git repository
- Target ports exposed for the application
- Number of replicas to be instantiated
- The secret key for accessing private repositories
- The deployment flavour for the media component
- Additional configuration parameters which may be required by the media service components

NFVO Connector
This component consumes the services of the NFVO, via the Pm-Or interface, for requesting the instantiation of the network services required by the applications for providing media transport, processing and archiving.

PaaS Connector
The PaaS Connector connects the PaaS Manager to the PaaS System, via the Pm-PaaS interface, by abstracting the PaaS specific technology adopted, therefore providing an abstracted interface which can facilitate the integration of different PaaS systems.

2.3.1.3 PaaS Manager API

This API abstracts the underlying complexity involved in deploying applications on the NUBOMEDIA Cloud Platform. As show on Table 1, it is based on a simple REST interface offering CRUD primitives enabling creating, updating and removing applications. Further information about the implementation of this API is given in upcoming sections.

Method	URL	Description
POST	/api/v1/nubomedia/paas/oauth/authorize	Authenticate a PaaS User
POST	api/v1/nubomedia/paas/users	Create a new PaaS User
DELETE	/api/v1/nubomedia/paas/users/{name}	Deletes a PaaS User
POST	/api/v1/nubomedia/paas/app	Create (build and deploy) a new application
POST	/api/v1/nubomedia/paas/app/{id}	Retrieve status of a application with the given id.
GET	/api/v1/nubomedia/paas/app	Return the list of all deployed applications
DELETE	/api/v1/nubomedia/paas/app/{id}	Delete application with the given id
POST	/api/v1/nubomedia/paas/secret	Create a secret
DELETE	/api/v1/nubomedia/paas/{name}	Deletes a secret

Table 1. Summary of the REST API exposed by NUBOMEDIA at the iD interface. This interface is also called the PaaS Manager API along this document and makes possible for application developers to deploy their NUBOMEDIA-enabled applications into the NUBOMEDIA PaaS.

More details about the APIs and its definition are given in Section 3.

2.3.2 NUBOMEDIA PaaS System

The NUBOMEDIA PaaS System hosts the server-side application logic on top of on demand deployed Application Servers. For this, it needs to provide a set of services to the PaaS Manager, via the Pm-PaaS interface, which mediates between developers and the PaaS. These services must make possible to deploy, scale and manage the application lifecycle. As shown in Figure 7, the NUBOMEDIA PaaS architecture comprises a number of building blocks, which are the following:

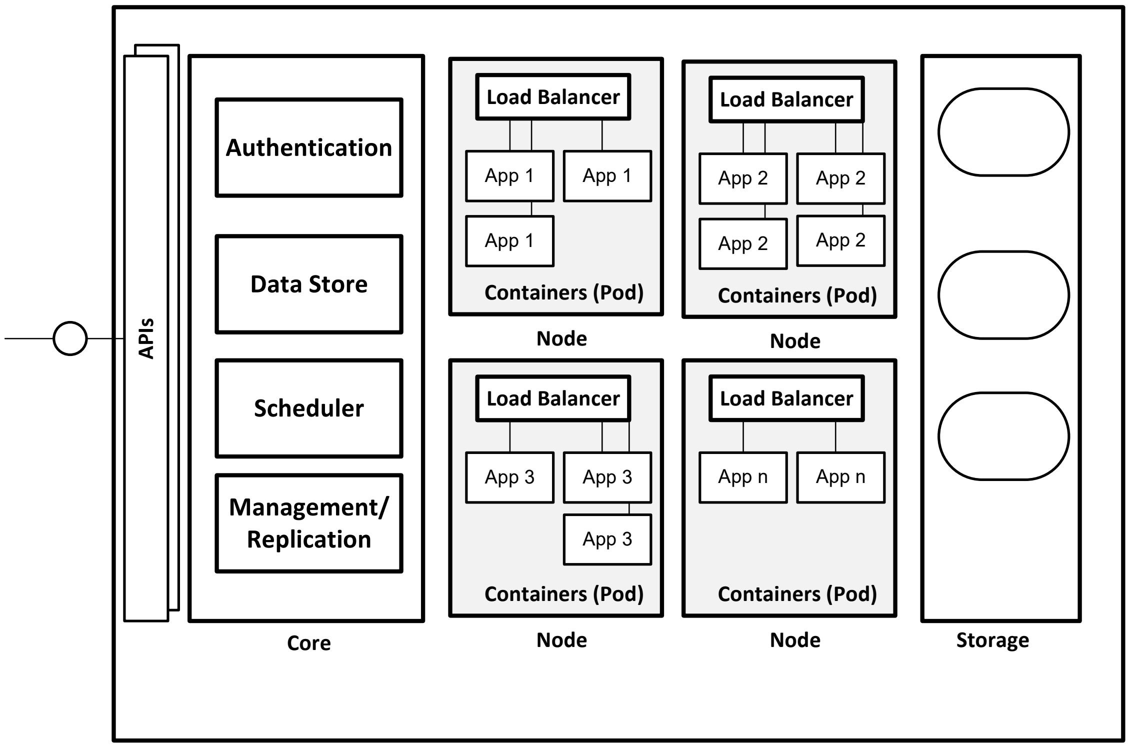


Figure 7. The NUBOMEDIA PaaS architecture

Core services. These include the following:

- Authentication: providing the authentication mechanisms enabling to determine the applicable security policies and permissions for a user.
- Data Store: this service provides data persistence and recovery capabilities for the different types of information required to deploy and manage an application.
- Scheduler: this service is responsible of the placement of application instances onto nodes within the cluster (see discussion below for understanding what's a pod and a node).
- Management/Replication: this service ensures that a specific number of application instance replicas are running at all times. This service manages application scalability so that, whenever there are too many pods running some are killed, and whenever too few pods are in execution some are launched. This service also replaces pods terminated pods (e.g. failure, disruptive node maintenance, etc.)

APIs. The core services of the NUBOMEDIA PaaS are provided as micro-services. Micro-services are a software architecture approach in which complex applications are split into small independent processes which communicate among each other using language agnostic APIs. As shown above, our core components are micro-services, which interact among each other and with the external world through common REST APIs.

Method	**URL**	**Description**
POST	/oapi/v1/imagestreams	Create an ImageStream
DELETE	/oapi/v1/namespaces/{namespace}/imagestreams/{name}	Delete an ImageStream
POST	/oapi/v1/deploymentconfigs	Create a DeploymentConfig
DELETE	/oapi/v1/namespaces/{namespace}/deploymentconfigs/{name}	Delete a DeploymentConfig
POST	/oapi/v1/namespaces/{namespace}/routes	Create a Route
DELETE	/oapi/v1/namespaces/{namespace}/routes/{name}	Delete a Route
POST	/oapi/v1/buildconfigs	Create a BuildConfig
DELETE	/oapi/v1/namespaces/{namespace}/buildconfigs/{name}	Delete a BuildConfig
POST	/oapi/v1/builds	Create a Build
DELETE	/oapi/v1/namespaces/{namespace}/builds/{name}	Delete a Build
GET	/oapi/v1/builds	List or watch objects of kind Build
GET	/oapi/v1/namespaces/{namespace}/builds/{name}/log	Read log of the specified BuildLog
POST	/api/v1/namespaces/{namespace}/services	Create a Service
DELETE	/api/v1/namespaces/{namespace}/services/{name}	Delete a Service
DELETE	/api/v1/namespaces/{namespace}/replicationcontroller/{name}	Delete a ReplicationController
DELETE	/api/v1/namespaces/{namespace}/pods/{name}	Delete a Pod
GET	/api/v1/namespaces/{namespace}/pods	List or watch objects of kind Pod
GET	/api/v1/namespaces/{namespace}/pods/{name}/log	Read log of the specified Pod
POST	/api/v1/namespaces/{namespace}/secrets	Create a Secret
DELETE	/api/v1/namespaces/{namespace}/secrets/{name}	Delete a Secret
GET	/api/v1/namespaces/{namespace}/serviceaccounts	List or watch objects of kind ServiceAccount
PUT	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	Replace the specified ServiceAccount

Table 2. Pm-PaaS interface definition

Containers, pods and nodes. Containers, pods and nodes are concepts imported from Kubernetes (http://www.kubernetes.io). Hence, the interested reader should refer to Kubernetes documentation for further information. For the objectives of this document, it is enough to say that containers are the basic unit of NUBOMEDIA PaaS applications. A container corresponds with the Linux notion of containers as a lightweight mechanism for isolating running processes. The NUBOMEDIA PaaS uses Docker containers (http://www.docker.com). Nodes, in turn, are worker machines, physical or virtual, where pods are run. Following Kubernetes terminology, a *pod* is an application logical host in a containerized environment. In other words, pods are collocated groups of containers that make up the application. Hence, nodes are capable of running containers and of managing groups of containers. The interesting feature about nodes is that they expose a simple network proxy and a load balancer suitable for forwarding traffic basing in round-robin mechanisms across sets of containers (i.e. pods). Hence, given that pods can be replicated at any time, and given that the same pods can be deployed across multiple nodes, the PaaS provides the appropriate horizontal scaling and redundancy for any service packaged into a container image.

Registry. The registry is a service for storing and retrieving Docker images. It contains a collection of Docker image repositories. In the case of NUBOMEDIA, all applications are deployed using Docker containers and the PaaS provides an internal registry for managing custom Docker images.

The relationship between Docker containers, images, and registries is depicted in Figure 8. All applications are deployed using Docker containers. A Docker container uses Docker image which is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing the application's needs (e.g. require Java Runtime Environment, etc.) and capabilities of the application. Different Images for different versions of the application can be defined and stored in the Registry and used later during deployment on different containers.

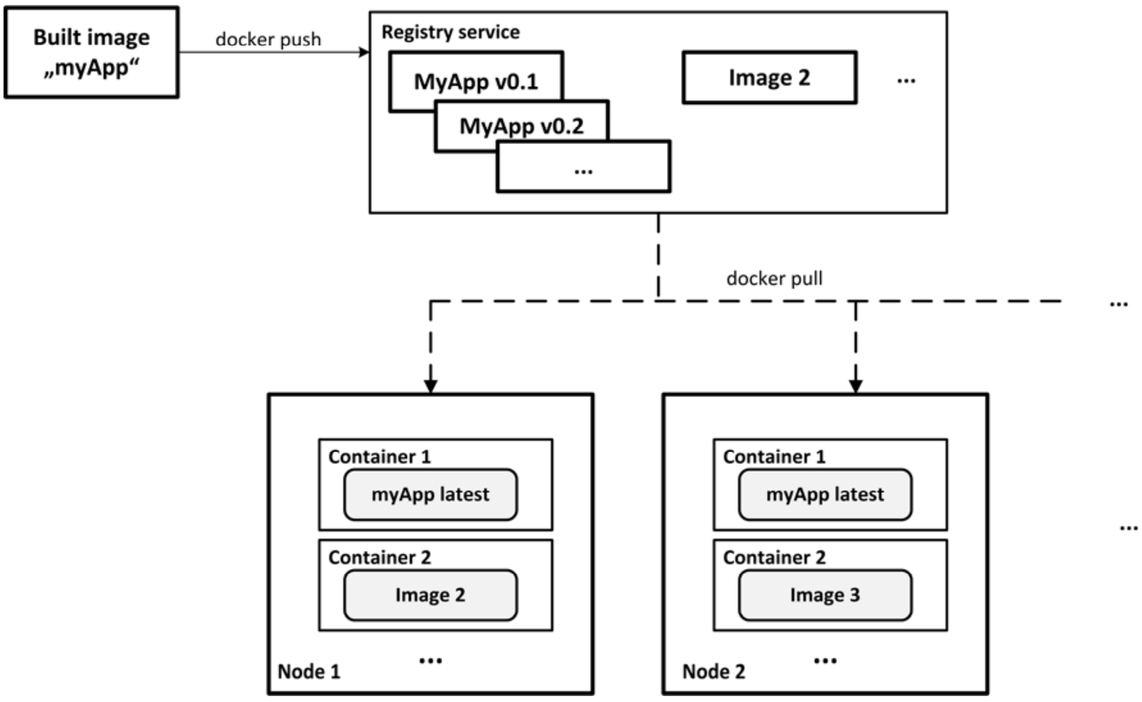


Figure 8. Relationship between Docker containers, images, and registries

2.3.3 Network Function Virtualization Orchestrator

In the ETSI white paper [ETSI_WP] the definition of NFV is that it "*aims to transform the way that network operators architect networks by evolving standard IT virtualization technology to consolidate many network equipment types onto industry standard high volume servers, switches and storage, which could be located in Datacenters, Network Nodes and in the end user premises. It involves the implementation of network functions in software that can run on a range of industry standard server hardware, and that can be moved to, or instantiated in, various locations in the network as required, without the need for installation of new equipment.*"

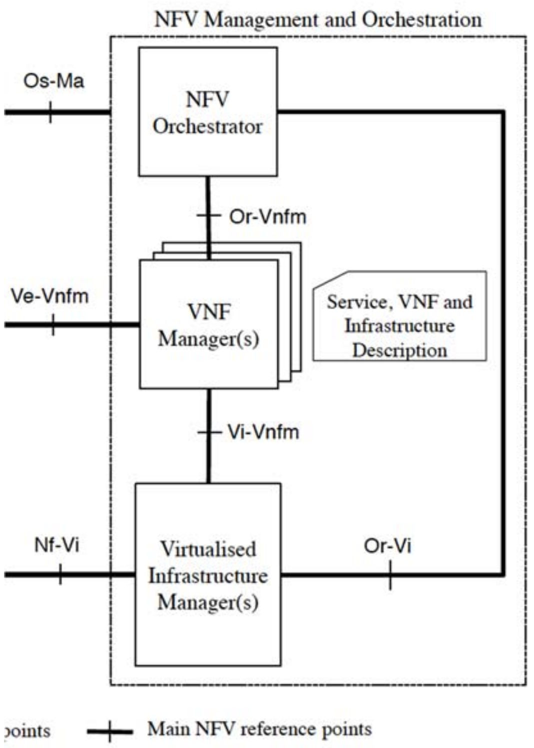
NFV is designed to consolidate and deliver the networking components needed to support a fully virtualized infrastructure, including virtual servers, storage and even other networks. It utilizes standard IT virtualization technologies that run on high-volume service, switch and storage hardware to virtualize network functions. It is applicable to any data plane processing or control plane function in both wired and wireless network infrastructures.

Network Function Virtualization considers the implementation of NFs as entities only in software that run over the Network Function Virtualization Infrastructure. Considering that Media Servers are intrinsically Network Functions, NUBOMEDIA followed the ETSI NFV specification for the management and orchestration of the media service components. The Network Function Virtualization Orchestrator (NFVO) is the component managing the lifecycle of a Network Service (NS) composed by multiple MS instances and (optionally) a Cloud Repository.

The NFV information model is generally divided in two sets of entities: Descriptors and Records. Information inside Descriptor elements are rather static information mostly used for initiating the on-boarding process of the Virtualized Network Functions (VNFs). Information inside Record elements are instead, dynamic information that are the result of the on-board process and will vary during the time. The cause of this variation can be a lifecycle operation completion or even a result of an unexpected event (i.e. a fault).

The main entity is the NS that describes the relationship between VNFs and possibly Physical Network Functions (PNFs) that it contains and the links needed to connect VNFs that are instantiated on the Network Function Virtualization Infrastructure (NFVI), which is basically what is defined with NUBOMEDIA IaaS in this document. A VNF is a Network Function implemented as a software component, a functional block with a clear interface and behavior. Hence, it is possible to deploy it in a virtualized infrastructure. The VNFD (Virtual Network Function Descriptor) describes a VNF in terms of its deployment and operational behavior requirements. A VNF can be instantiated either on a single or multiple VNF Components (VNFC). The VNF Manager (VNFM) makes use of the VNFD while instantiating and managing the lifecycle of the VNF. The information provided in the VNFD are also used by the NFVO to manage and orchestrate Network Services and virtualized resources on the NFVI. The VNFD contains connectivity, interface and KPIs requirements that can be used by NFV-MANO functional blocks to establish appropriate Virtual Links within the NFVI between its VNFC instances, or between a VNF instance and the endpoint interface to the other VNF.

This information model is internally used by the NFVO, by the VNFM and by the Virtualized Infrastructure Manager (VIM). The Figure 9, taken from the ETSI NFV MANO [MANO] specification, shows the ETSI NFV architecture for the MANO domain and specifies how the main three components are connected. As it can be noticed, NUBOMEDIA followed the ETSI NFV MANO architecture and extended it for supporting additional capabilities.


NFV Management and Orchestration
Os-Ma
NFV Orchestrator
Or-Vnfm
Ve-Vnfm
VNF Manager(s)
Service, VNF and Infrastructure Description
Vi-Vnfm
Nf-Vi
Virtualised Infrastructure Manager(s)
Or-Vi
points
Main NFV reference points
Figure 9. ETSI NFV MANO Architecture

While NFVO is mainly responsible of the management of the lifecycle of the Network Services, the VNFM is responsible for the lifecycle management of a single VNF (e.g. instantiation, update, query, scaling, termination, etc.). From the NFV specification either NFVO or VNFM shall instantiate virtual resource through the VIM. As it can be observed in Figure 9, both components have an interface to the VIM. The selection of either the first or the second approach is left completely to the VNF developer, however the allocation of resources is done by the NFVO which is the only component having a complete view of available resources on the NFVI.

The VNFM provides an API to the NFVO for managing the lifecycle of the VNF. It can be specific, in case the VNF it manages requires specific mechanisms for being managed, or generic, in case the VNF all the lifecycle events can be described in the descriptor and do not require additional capabilities. In NUBOMEDIA both approaches were used. In particular, the media service components are managed by the Elastic Media Manager (EMM) Virtual Network Function Manager (VNFM), as it extends the standard functionalities of the VNFM providing also specific autoscaling mechanisms and media session placement.

As already mentioned, the NFVO manages the lifecycle of a Network Service. It exposes an interface to the PaaS Manager providing functionalities for managing the Network Service. This interface enables the following operations: 1) instantiation of the virtual computing resources (via the VIM interface) required by the Network Service, 2) provisioning of guaranteed networking resources and 3) management of the lifecycle of the different VNFs via the VNFMs. As shown in Figure 10, the NFVO comprises the following modules:

- API, which exposes the NFVO-API consumed by the PaaS Manager. Following ETSI terminology, this API makes possible to instantiate Network Service Records (NSRs) from Network Service Descriptors (NSDs). An NSD is a deployment template that describes a Network Service in terms of its deployment and operational behavior requirements. NSD are represented as JSON files and are stored into a catalog. Inside the NSD we can find the VNF Descriptors (VNFD) containing the Virtual Deployment Unit (VDU) Descriptors describing aspects such as instance flavor, deployable components, scaling limits, etc. An instantiation of an NSD is known as an NSRs, while an instantiation of an VNFD is an VNF Record (VNFR). Table 3 provides an overview of the Pm-Or interface employed in NUBOMEDIA. Further details are given into the implementation section.

Method	**URL**	**Description**
POST	/api/v1/ns-descriptors	Adding a Network Service Descriptor
POST	/api/v1/ns-records/{nsdId}	Deploying a Network Service Record from an existing NSD
GET	/api/v1/ns-records/{nsrId}	Retrieving a Network Service Record
DELETE	/api/v1/ns-records/{nsrId}	Removing a Network Service Record
POST	/api/v1/ns-records/{id}/vnfrecords/{idVnf}/vdunits/{idVdu}/vnfcinstances/{idVNFCI}/start	Starting a VNFC Instance
POST	/api/v1/ns-records/{id}/vnfrecords/{idVnf}/vdunits/{idVdu}/vnfcinstances/{idVNFCI}/stop	Stopping a VNFC Instance
POST	/api/v1/ns-records/{nsrId}/vnfrecords/{vnfrId}/vdunits/{vduId}/vnfcinstances	Add (scale-out) VNFC Instance to NSR:VNFR:VDU
DELETE	/api/v1/ns-records/{nsrId}/vnfrecords/{vnfrId}/vdunits/{vduId}/vnfcinstances/vnfciId	Remove (scale-in) VNFC Instance from NSR:VNFR:VDU

Table 3. Pm-Or interface definition

- Repository, which exposes to the NFVO the Catalogue: a repository where the different resources available to the NFVO (such as the NSDs) are stored.
- Virtual Infrastructure Manager Connector (VIMC): it is a module used by the Core for interoperating with the VIM.
- Core, represents the central module of the NFVO. It coordinates all the lifecycle events of the Network Service Records instantiation. It has several fundamental functionalities: from the management of the catalogue to the actual instantiation of a Network Service Record (NSR) starting from a Network Service Descriptor (NSD).

- VNFM Connector (VNFMC), which provides an internal interface for dispatching messages to the different VNFMs which are registered.

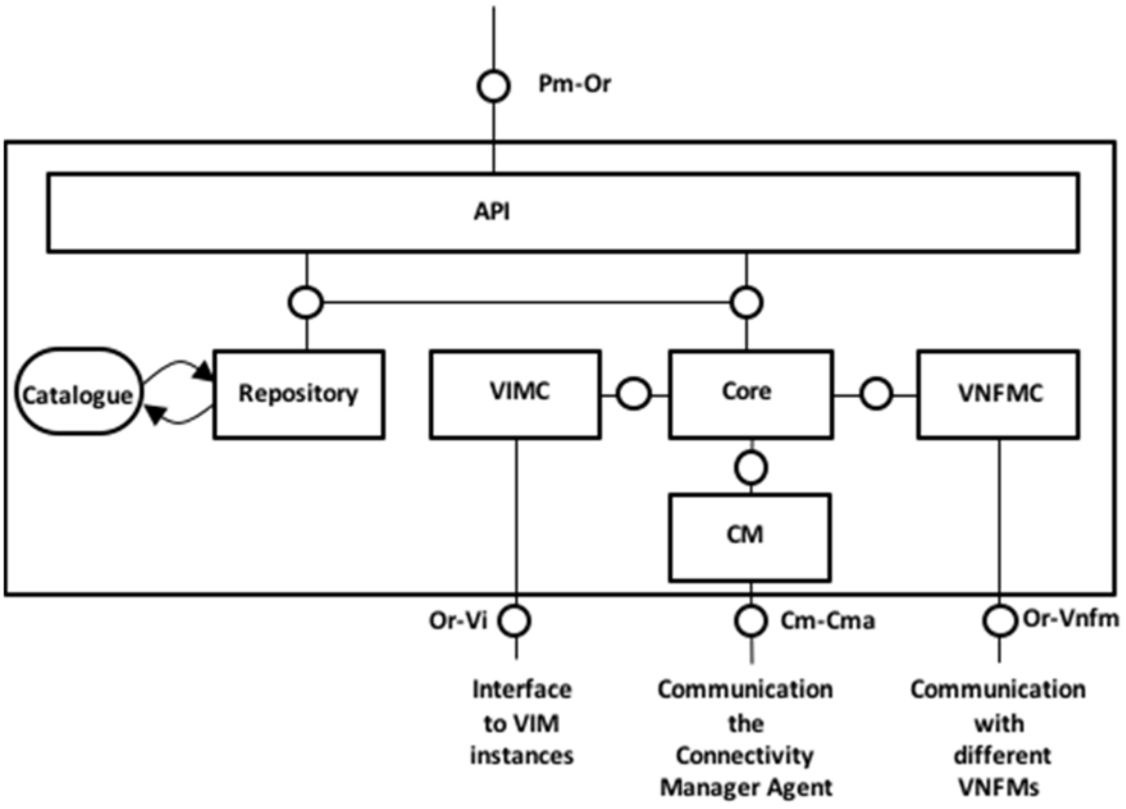

Pm-Or
API
Catalogue
Repository
VIMC
Core
VNFMC
CM
Or-Vi
Cm-Cma
Or-Vnfm
Interface to VIM instances
Communication the Connectivity Manager Agent
Communication with different VNFMs

Figure 10. NFVO internal architecture.

2.3.4 Connectivity Manager

While deploying Media Functions on top of a shared Virtualized infrastructures it is important to consider the networking requirements which each individual application may have. In particular, while having different kind of users deploying applications on top of the same physical infrastructure it is required to introduce some kind of mechanism for guaranteeing that a particular level of QoS is maintained, without any interferences from other users of the infrastructure. The Connectivity Manager (CM) provides mechanisms for controlling QoS parameters, like bandwidth, between media service components running on the IaaS. Basically, the CM provides capabilities for enforcing specific level of QoS between VNF Components. Although most of the time in large datacenters the bottleneck is on the physical link towards the public internet, it is also important to clarify that in case of congestions (due to high bandwidth-consuming kind of applications) it is possible to have situations in which the quality of the media is deteriorating.

For this reason, the CM makes use of Software Defined Networking (SDN) technologies, in particular for limiting the network bandwidth rates of the virtual links across several media server instances, using traffic shaping.

For achieving this, it has been necessary to extend the NFV architecture with two additional functional components, the CM integrating at the MANO level, and the Connectivity Manager Agent (CMA) which have to be deployed together with the VIM control entities managing the NFV Infrastructure (NFVI).

The CM, placed in the middle between the NFV Orchestrator (NFVO) and the CMA, has a higher level of abstraction (intended as more oriented towards the ETSI NFV data representation) with respect to the CMA which has a pure platform data model.

The main concept is to let the CM extract the requirements in terms of required network capabilities of each different slice and enforce them on the NFVI making use of SDN mechanisms. Thereafter, the CM interacts with the NFVO for retrieving the requirements contained in the Network Service Record (NSR), and the list of VNFC instances used for implementing the NS.

The CMA provides a simplified API for associating a specific level of QoS required between the VNFs and corresponding VNFC instances composing the NS.

The mechanism used for interacting with the NFVO is based on events. The CM subscribes to the NFVO for receiving the NSR when a new network service is instantiated. The event payload containing the NSR is then parsed to retrieve all information of the physical allocations of each VNFC instance and the required QoS of each virtual link.

Then, the CMA retrieves the VNFC instances locations to get an overall vision of the network service topology and allocate the desired slice characteristics for each VNFC at its point of presence. The CMA has to be aware of the entire network topology of each data center under the control of NFVO and also of the topology that is connecting these data centers, in order to define a path involving each VNFC instance that is involved in the network service. The CMA is also in charge of providing a northbound API of the SDN driver and acts as a common contract at the SDN controller side which simplifies the allocation and deletion of the network slice for the data center.

The SDN Driver performs the requests to data center's SDN controller, which could be different for each data center. This driver interacts with the SDN controller, maintaining the necessary data and translates the slice requirements from the data defined in the QoS Interface to data type required by the SDN platform.

The CM includes a Policy model which is inspired by the DiffServ traffic class. The policies are divided into classes; each class has its specific application parameters. These supported parameters are the maximum and minimum bandwidth rates. The model includes three classes of Slices, but it could be extended to the bare number as parameters: GOLD, SILVER and BRONZE.

2.3.5 Virtual Network Function Manager

The VNFM, as already mentioned before, provides lifecycle management for a VNF. The VNFM can be specific (i.e. adapted to a specific type of VNF) or generic. In NUBOMEDIA we use the two. The VNFM-EMM is specific and its functionalities are adapted to managing Media Server instances. On the other hand, the Cloud Repository one (CR-VNFM) is based on a generic architecture. Both expose the same type of interface to the NFVO, as specified by ESTI MANO. In both cases also, the main function of the VNFM is to deploy and manage a specific VNFR (Virtual Network Function Record), as specified in the corresponding VNFD.

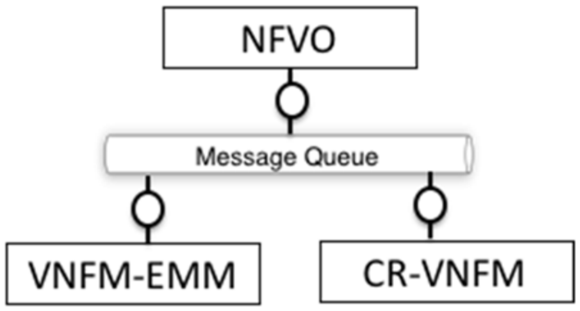


Figure 11. Following ETSI MANO architecture, the VNFM-EMM and the CR-VNFM are managed by the NFVO through a control bus

The core of the VNFM-EMM, shown in Figure 12, contains the following components:

- Media Server Manager: The Media Server Manager is in charge of processing all the operations of the Lifecycle Management (i.e. VNFR instantiation, VNFR termination, start, modify, configure, etc.) that are triggered by the NFVO at the predicted point in time. So, first it starts with the VNFR instantiation requested by the NFVO. Once all the components of the specific VNFR are instantiated, the NFVO will call the start procedure of the VNFM to prepare available Media Servers, start the elasticity, and finally let the VNFR go into the ACTIVE state reflecting that all Components are up and running ready for the foreseen activities. Once the termination of a specific VNFR is requested by the NFVO to the VNFM, the Media Server Manager releases all resources and terminates the management of the corresponding VNFR by notifying the NFVO.
- Media Server Management: The Media Server Management is responsible for managing the Media Servers that offers resources for Applications and Media Pipelines. Media Servers are extensions of *VNFCInstances* (Virtual Network Function Component Instances - or simply Virtual Machines) where new Applications will be assigned to. Once new Applications are registered, the capacity of corresponding Media Servers will be reduced by the amount of capacity the Applications have requested. After unregistering Applications, the capacity of Media Servers will be released respectively.
- Application Management: The Application Management is in charge of managing Applications by providing the capabilities for registering and unregistering Applications. For this purpose, it requests the Media Server Management for allocating and releasing corresponding resources for newly created or removed Applications and Media Pipelines.

Furthermore, the VNFM-EMM provides an API to applications deployed in the NUBOMEDIA PaaS for retrieving dynamically the available Media Servers. This API is consumed by the NUBOMEDIA Media API implementation to determine in which specific Media Server instance a newly created Media Pipeline is placed. Allocating new Media Pipelines is transparent to the developer, who just needs to create the Media Pipelines without worrying about where they are located. For this, the VNFM has been extended with the autoscaling component, providing semantics to that placement interface and guaranteeing the availability of Media Servers through a horizontal autoscaling mechanism based on two operations: scaling-out (i.e. to add resources when necessary) and scaling-in (i.e. to remove resources when they are no longer required). Both, the scaling-in and -out, are fired by simple autoscaling policies based on QoS metric thresholds, so that, for example, further resources are added when the average CPU load of Media Server instances is over an upper limit and resources are collected when it’s under a lower bound.

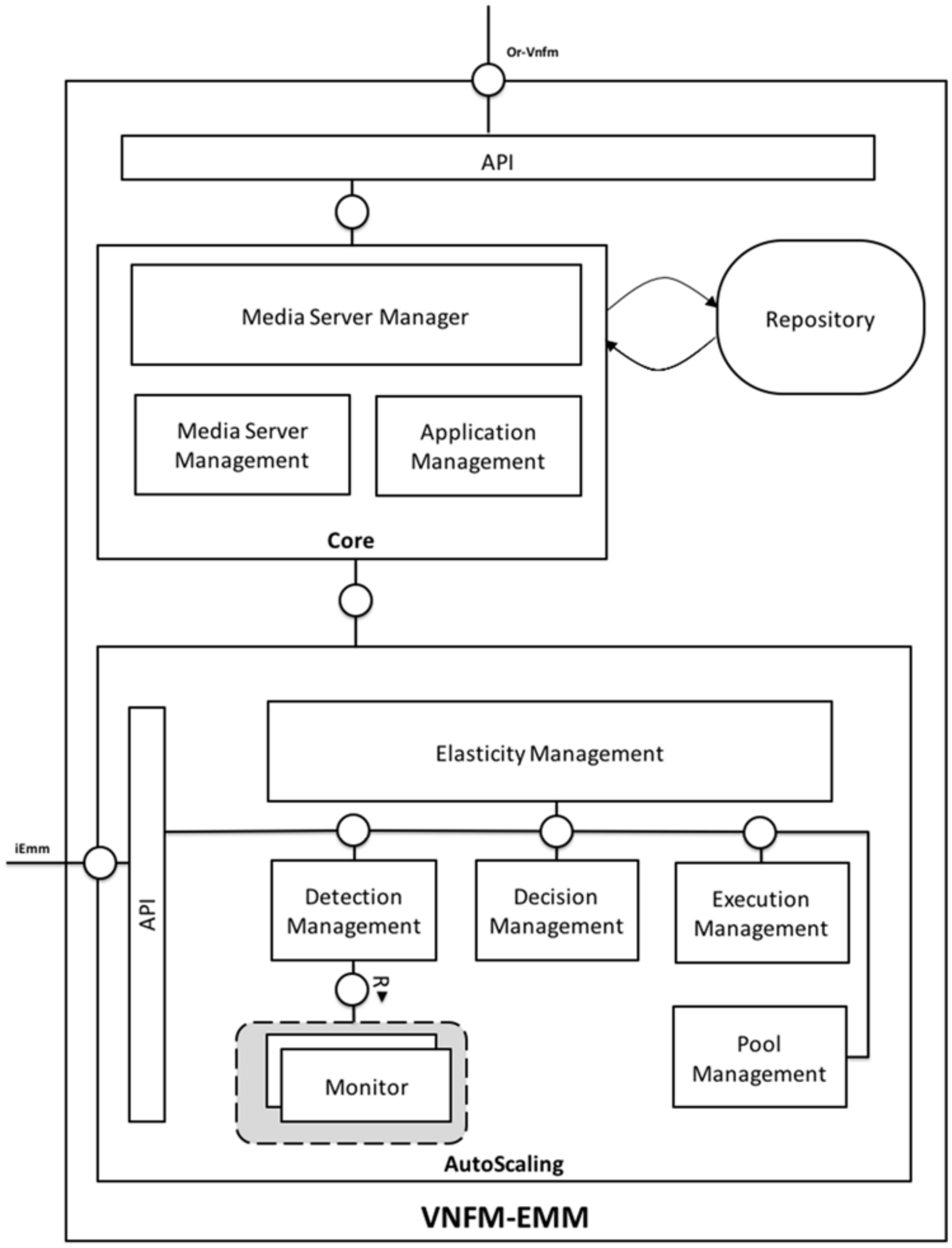


Figure 12. VNFM-EMM Architecture extended the basic VNFM with the EMM

As specified in the ETSI NFVO MANO, autoscaling is spread over several components in order to detect the need of scaling (Detection Management), to make decisions (Decision Management) and finally to execute the actions (Execution Management) that were decided before. Additionally, the autoscaling system is extended with a Pool mechanism (controlled by the Pool Management) to decrease significantly the time of allocating new Media Servers. This pool contains Media Servers that allows to provide already prepared and launched Media Servers on demand without waiting the time needed for deploying a completely new Media Server. At runtime, the Alarm Detection is triggered by Elasticity Management whereas the Elasticity Management is either notified by the NFVO once the initial deployment is finished or directly through the VNFM when the VNFR goes into ACTIVE. The CR-VNFM, in turn, is simpler and manages the lifecycle of Cloud Repository elements. For this, it just receives information about the virtual resources instantiated by the NFVO and installs the repository components.

2.3.6 Virtual Infrastructure Manager

The VIM provides an interface for controlling the NUBOMEDIA IaaS. As in the ETSI NFV specification, the VIM follows the OpenStack approach and, through its APIs, offers the ability to start new computing resources by using already pre-configured images containing the appropriate artifacts for every of the required functions. The computing nodes are instantiated on the NUBOMEDIA IaaS on top of one or more Compute Nodes. Computing nodes are distributed across physical machines depending on the required flavors and resources.

2.3.7 Network Service Record (NSR) deployment scenario

In order to deploy a NSR, some steps need to be done first. A descriptor of the VIM in use needs to be uploaded to the NFVO. In particular, it is important to provide the authUrl of the OpenStack installation, username, password, keypair to use, a list of security groups and a name. The name needs then to be used into a NSD, in particular into the Virtual Deployment Unit, defining where all the components belonging to that VDU will be deployed. Once these two steps are done, the NFVO is ready to deploy a NSR starting from the information contained into the NSD uploaded. That process is described into the following sequence diagram:

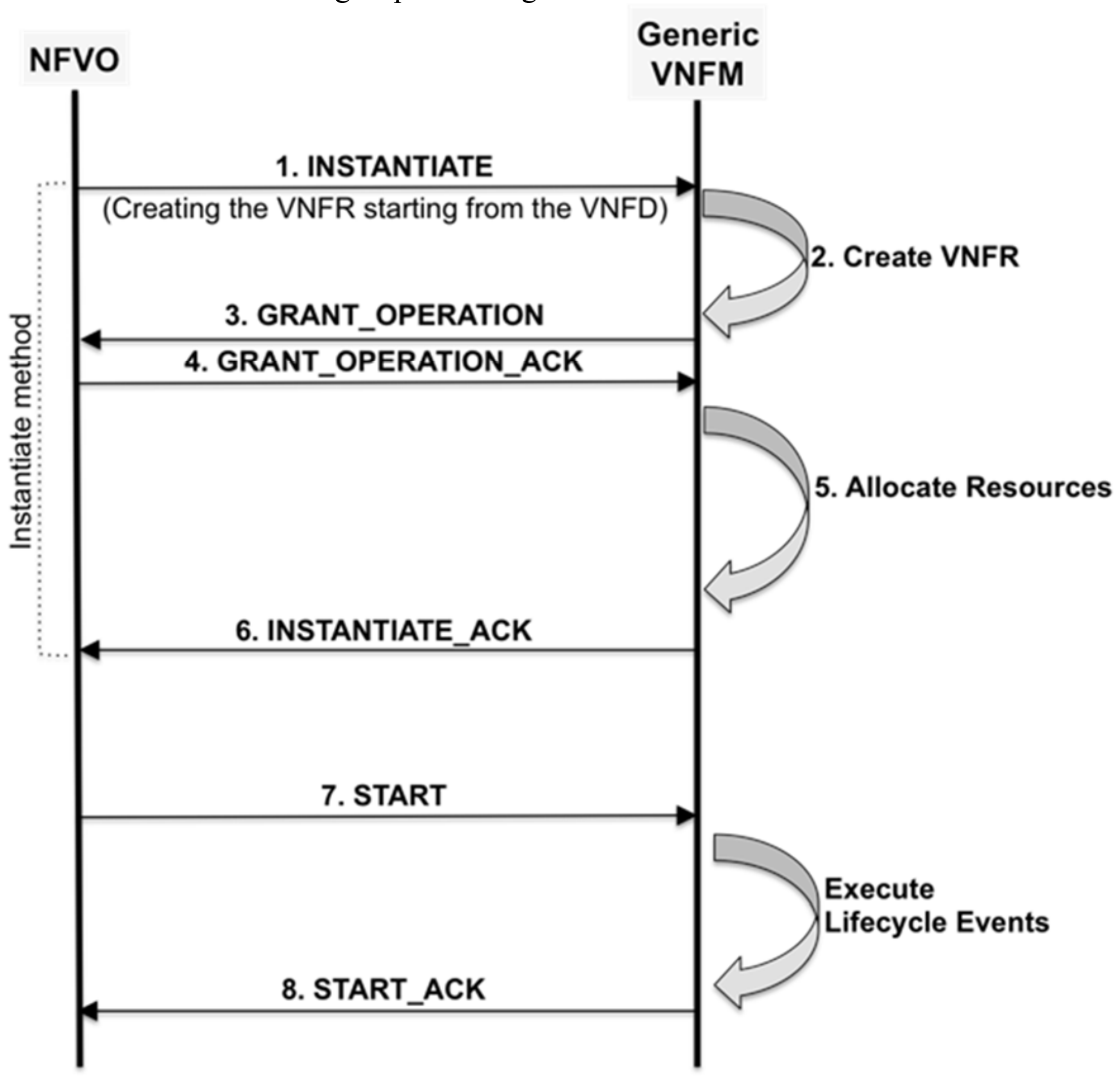


Figure 13. NFVO, VNFM and EMS sequence diagram for NSR deployment

2.3.7.1 Instantiate

The first message sent to the Generic VNFM is the INSTANTIATE message (1). This message contains the VNF Descriptor and some other parameters needed to create the

VNF Record, for instance the list of Virtual Link Records. The VNFM calls then the createVirtualNetworkFunctionRecord method (2) and the Virtual Network Function Record is created and sent back to the NFVO into a GRANT_OPERATION message (3). This message will trigger the NFVO to check if there are enough resources to create that VNF Record. If so, then a GRANT_OPERATION message with the updated VNF Record is sent back to the VNFManager. Then there are two options: either the VNFManager creates an ALLOCATE_RESOURCES message with the received VNF Record and sends it to the NFVO or the VNFM creates the Virtual Resources on its own. In the purpose of this project the VNFM creates the resources (5) without asking to the NFVO to do so. After that step, the instantiate method is finally called. Inside this method, the scripts (or the link to the git repository containing the scripts) contained in the VNF Package is sent to the EMS, the scripts are saved locally to the VM and then the VNFManager is in charge of calling the execution of each script defined in the VNF Descriptor, if any. Once all of the scripts are executed and there wasn't any error, the VNFManager sends the Instantiate message back to the NFVO (6).

2.3.7.2 Modify

If the VNF is target for some dependencies, like the iperf client, the MODIFY message is sent to the VNFManager by the NFVO. Then the VNFManager executes the scripts contained in the CONFIGURE lifecycle event defined in the VNF Descriptor, and sends back the modify message to the NFVO, if no errors occurred. In this case, the scripts environment will contain the variables defined in the related VNF dependency. If no dependencies are defined into the NSD, this method is ignored.

2.3.7.3 Start

Here exactly as before, the NFVO sends the START message to the VNFManager (7) and the VNFManager calls the EMS for execution of the scripts defined in the START lifecycle. And the start message is then sent back to the NFVO meaning that no errors occurred (8).

2.3.7.4 Application management (Placement)

In Section 2.2 it has been introduced an overview about the autoscaling mechanism designed in NUBOMEDIA. Basically, the application discovers dynamically the media server instance which has to be used for starting a new session each time a new client connects. This mechanism is provided via the iEmm interface, registering and de-registering applications onto a particular media server instance. The idea is to “place” sessions based on their requirements, defined in terms of capacity (points) of media capabilities required.

The following algorithms are applied when registering or unregistering Applications at the EMM level. In general, when registering a new Application, it will be occupied a specific amount of resources, whereas unregistering an Application means, that the resource occupied before by the Application will be released again. The key approach of these algorithms is the points mechanism where points reflect the capacity of the Media Components on the one hand and the amount capacity required by the Applications on the other hand.

Another important feature is the Heartbeat mechanism. This mechanism is used for keeping the Application alive actively. Missing Heartbeats will lead to release resources of specific Applications automatically.

2.3.7.4.1 Register new applications

The main purpose of registering a new Application is to provide a Media Component where Applications can be established on. Hence, the goal of this algorithm is to find a Media Component that satisfies the requirements of the Application in the meaning of Capacity. Figure 14 depicts the whole workflow of registering a new Application. This is done as follows:

1. The EMM receives the request to register a new Application with a specific amount of points. Optionally, the external Application ID could be passed as well to ensure that the same Application will not be registered multiple times what would mean to occupy resources of Media Components multiple times by the same Application.
2. (Optional) If the external Application ID is passed in the request, check first if there is already an Application registered with that external Application ID. If there is already an Application registered with that external Application ID, use the Media Component where the Application was assigned to in the previous request. If the Application was not registered before, continue with the next step.
3. First, check if there is any Media Component in the ACTIVE pool that can be assigned to establish another Application. If not, it is taken one from the IDLE pool.
4. The selected Media Component (either from the ACTIVE or IDLE pool) that satisfies the requirements of the Application, in the meaning of enough points are left, is assigned to establish the Application. This means, the left capacity (points) of the Media Component will be updated by reducing the capacity of the Media Component by the amount of capacity that is requested by the Application. If the Media Component comes originally from the IDLE pool, it will be moved to the ACTIVE pool.
5. (Optional) Scaling can be triggered in multiple ways by checking pool sizes and adding or removing Media Servers respectively. The following three options are available:
    a. Alarms and Actions for scaling are defined in AutoScalePolicies. Alarms are frequently checked and Actions are executed when a threshold crossing is detected. An Alarm contains the pool size as monitoring parameter and the threshold to check at runtime. The Action contains the number of Media Server that reflects the size of the corresponding pool.
    b. Events or Actions are sent to the VNF Manager at the end of the registration process. The VNF Manager is responsible for checking pool sizes and triggers scaling if needed.
    c. At the end of the registration process: The EMM itself checks pool sizes and takes decisions about scaling. If scaling is needed, the EMM send a scaling request to the VNF Manager explicitly.
6. Finally, the response returned contains the Media Component with the corresponding IP where the requested Application can be established.

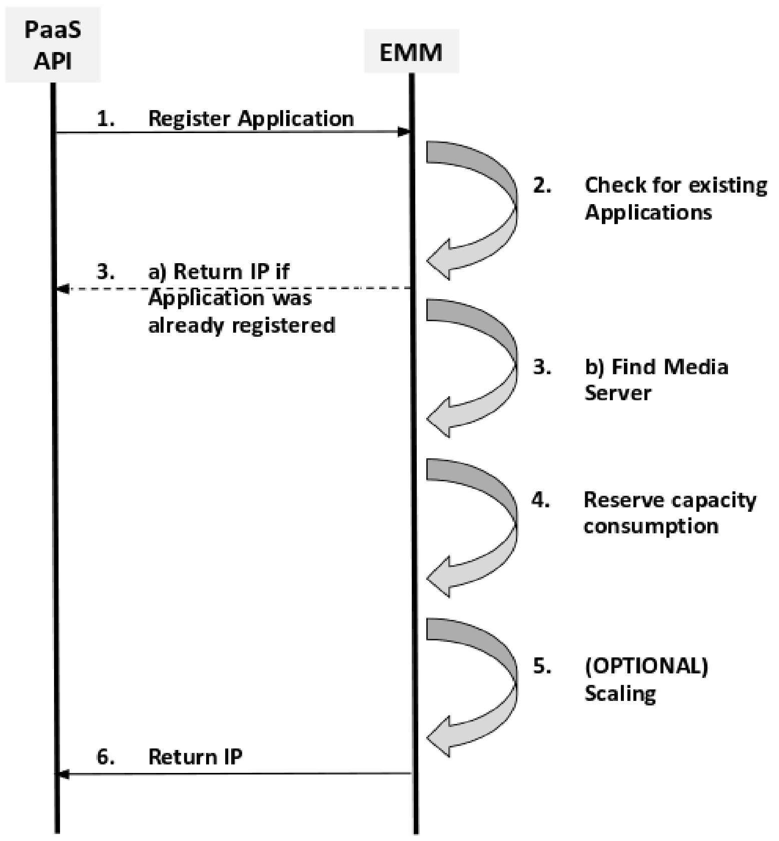

PaaS API
EMM
1. Register Application
2. Check for existing Applications
3. a) Return IP if Application was already registered
3. b) Find Media Server
4. Reserve capacity consumption
5. (OPTIONAL) Scaling
6. Return IP

Figure 14. Registration of an Application

2.3.7.4.2 Unregister an application

Unregistering Applications means to remove an Application and release the corresponding resources of the Media Component that were occupied. This works as follows:

1. The EMM receives the request to unregister a specific Application.
2. First, it is checked which Media Component is currently occupied by this Application.
3. Then the Media Component; responsible for this Application, releases the resources by releasing the assigned capacity (points). Afterwards it is verified if the Media Component is still occupied by another Application. If yes, the Media Component stays in the ACTIVE pool. If not, the Media Component will be moved to the RELEASE pool.
4. (Optional) Scaling can be triggered in multiple ways by checking pool sizes and adding or removing Media Servers respectively. The following three options are available:
    a. Alarms and Actions for scaling are defined in AutoScalePolicies. Alarms are frequently checked and Actions are executed when a threshold crossing is detected. An Alarm contains the pool size as monitoring parameter and the threshold to check at runtime. The Action contains the number of Media Server that reflects the size of the corresponding pool.
    b. Events or Actions are sent to the VNF Manager at the end of the registration process. The VNF Manager is responsible for checking pool sizes and triggers scaling if needed.
    c. At the end of the registration process: The EMM itself checks pool sizes and takes decisions about scaling. If scaling is needed, the EMM send a scaling request to the VNF Manager explicitly.

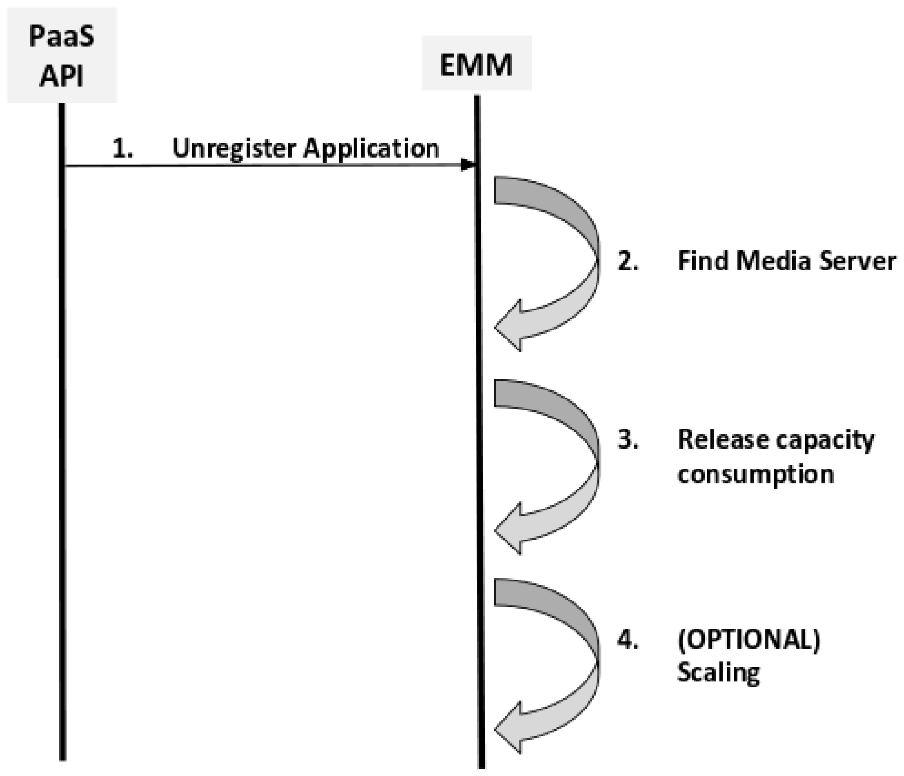


Figure 15. Unregistering an Application

2.3.7.5 Heartbeat mechanism

The Heartbeat mechanism is responsible for keeping the Applications alive actively. Actively means, that the EMM needs to receive Heartbeats in specific time intervals (heartbeat period). If Heartbeats for specific Applications stay off for defined time intervals or conditions (e.g. number of session is 0), the Application will be removed automatically by releasing consumed capacity. All the cases are shown in the following sequence diagrams. Steps shown in the sequence diagrams are explained below each figure.

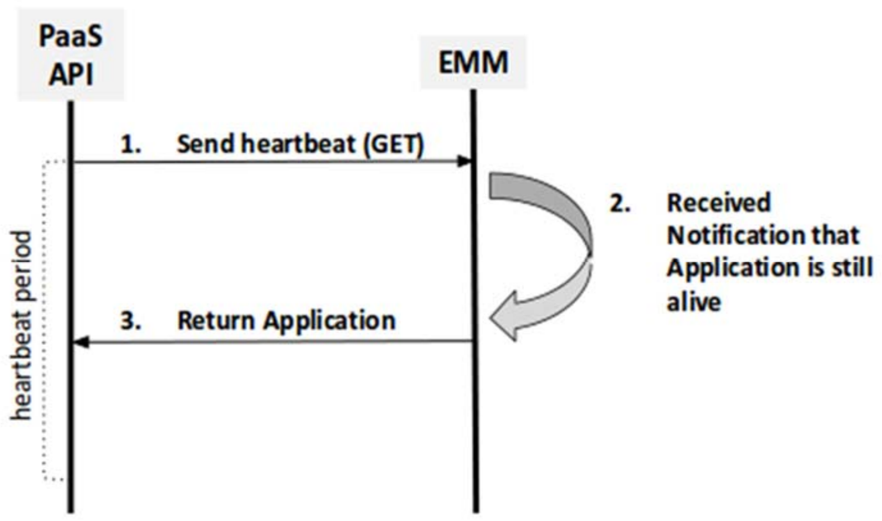


Figure 16. Heartbeat mechanism

In Figure 16 it is shown how a normal Heartbeat operation works. In this case, the PaaS API sends the Heartbeat directly to the EMM to signalize that the specific Application is still controlled by an external component. In the following each step is explained in more detail:

1. The PaaS API sends the Heartbeat to the EMM by invoking a GET request by passing the Application ID and the VNFR ID where the Application is deployed on.
2. The EMM receives the Heartbeat and updates the time of the last Heartbeat received for this Application.
3. All information about the deployed Application are sent back to the PaaS API.

The next sequence diagram (see Figure 17) covers the case when one Heartbeat was missed already by the EMM. When the EMM receives a delayed Heartbeat, it might have started already with checking the number of sessions for removing the application automatically. Each step is more described below:

1. Expected Heartbeat by the EMM stays off. So, the EMM does not receive the Heartbeat in the expected time period and it starts with checking conditions for releasing reserved capacity in case the Application is not alive anymore.
2. EMM requests the Monitor for the number of sessions of the considered Application.
3. Monitor sends the number of sessions back to the EMM.
4. EMM recognizes that there are still sessions for this Application.
5. Once the release check period is over, the EMM starts again with checking the number of sessions and requests, therefore, the Monitor again.
6. The Monitor send the monitoring result back to the EMM.
7. While the EMM is in the process of checking left sessions, the EMM receives a heartbeat and stops immediately with trying to remove the Application caused by the previously missed Heartbeat.
8. The PaaS API sends another Heartbeat to the EMM to signalize activity.
9. The EMM processes the heartbeat.
10. Application is returned to the PaaS API answering to the request received right before.

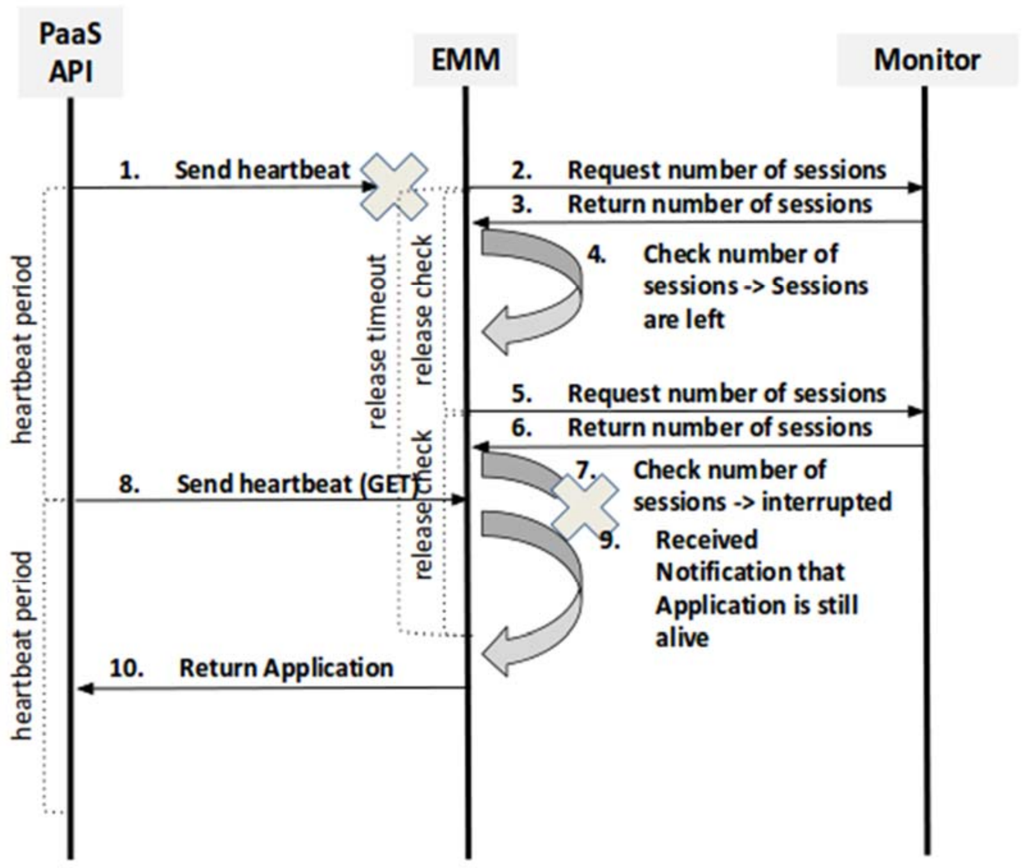

PaaS API
EMM
Monitor
1. Send heartbeat
2. Request number of sessions
3. Return number of sessions
4. Check number of sessions -> Sessions are left
5. Request number of sessions
6. Return number of sessions
7. Check number of sessions -> interrupted
8. Send heartbeat (GET)
9. Received Notification that Application is still alive
10. Return Application
heartbeat period
heartbeat period
release timeout
release check
release check

Figure 17. Heartbeat interrupts release check

The next two sequence diagrams cover the scenarios where Applications are removed automatically. Either by recognizing that no more sessions are active or by exceeding a timeout that indicates a misbehavior of the Application itself. In Figure 18 it is shown the first case where the Heartbeat is missed and the number of session went to 0 that is an indicator for inactivity of the Application. The steps are explained in the following:

1. Expected Heartbeat by the EMM stays off. So, the EMM does not receive the Heartbeat in the expected time period and it starts with checking conditions for releasing reserved capacity in case the Application is not alive anymore.
2. Emm requests the Monitor for the number of sessions of the considered Application.
3. Monitor sends the number of sessions back to the EMM.
4. EMM recognizes that there are still sessions for this Application.
5. Once the release check period is over, the EMM starts again with checking the number of sessions and requests, therefore, the Monitor again.
6. The Monitor send the monitoring result back to the EMM.
7. EMM recognizes that the number of sessions went to 0. This and the missing Heartbeat is the indicator for unregistering the Application.
8. EMM unregisters the Application and releases consumed capacity.

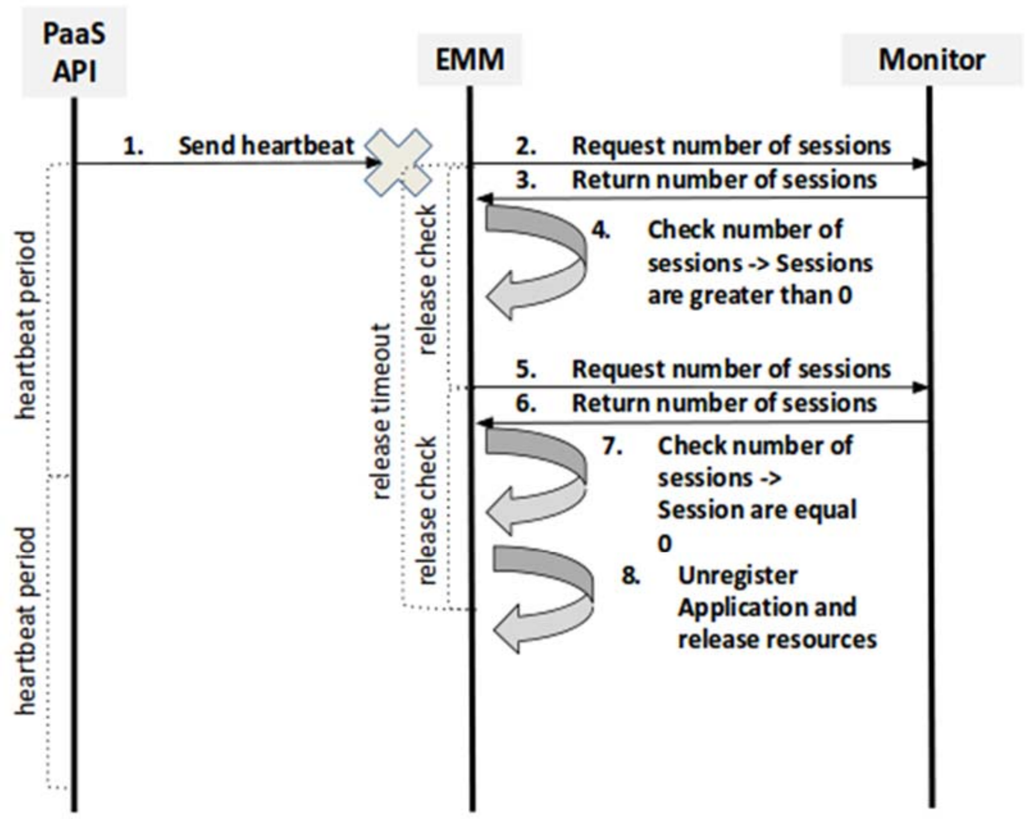

PaaS API
EMM
Monitor
1. Send heartbeat
2. Request number of sessions
3. Return number of sessions
4. Check number of sessions -> Sessions are greater than 0
5. Request number of sessions
6. Return number of sessions
7. Check number of sessions -> Session are equal 0
8. Unregister Application and release resources
heartbeat period
heartbeat period
release timeout
release check
release check

Figure 18. Missing Heartbeats and release check

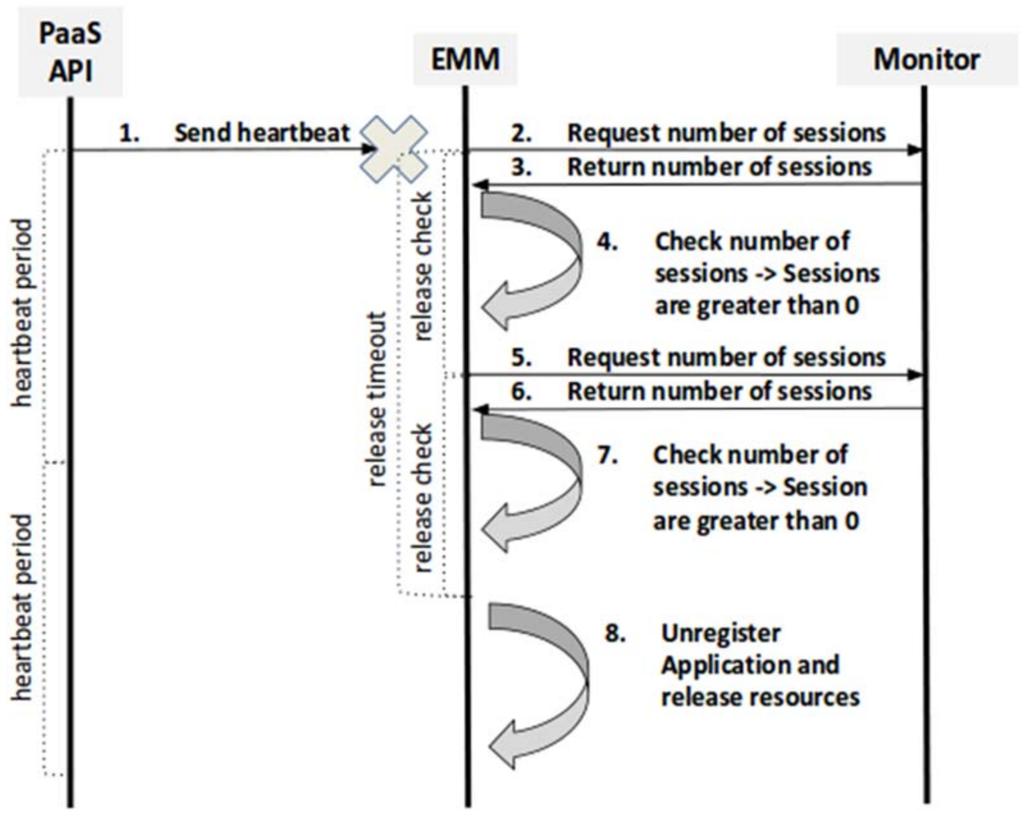

PaaS API
EMM
Monitor
1. Send heartbeat
2. Request number of sessions
3. Return number of sessions
4. Check number of sessions -> Sessions are greater than 0
5. Request number of sessions
6. Return number of sessions
7. Check number of sessions -> Session are greater than 0
8. Unregister Application and release resources
heartbeat period
heartbeat period
release timeout
release check
release check

Figure 19. Missing Heartbeats and release timeout

In Figure 19 it is depicted the automatic removal of an Application when the Heartbeat is missed for a longer time and the release timeout is exceeded. This happens when the number of session is greater than 0 all the time. Steps are explained below:

1. Expected Heartbeat by the EMM stays off. So, the EMM does not receive the Heartbeat in the expected time period and it starts with checking conditions for releasing reserved capacity in case the Application is not alive anymore.
2. EMM requests the Monitor for the number of sessions of the considered Application.
3. Monitor sends the number of sessions back to the EMM.
4. EMM recognizes that there are still sessions for this Application.
5. Once the release check period is over, the EMM starts again with checking the number of sessions and requests, therefore, the Monitor again.
6. The Monitor send the monitoring result back to the EMM.
7. EMM recognizes that there are still sessions for this Application.
8. Once the release timeout is exceeded, the Application will be unregistered automatically. Reserved resources will be released automatically as well.

3 Software Architecture

In this section, it is provided a description about the Software Architecture of the NUBOMEDIA Cloud Platform, based on the functional architecture presented in Section 2. In particular, the main focus will be on how the functional elements presented above have been implemented and how they interact with each other.

3.1 Software Architecture Overview

The NUBOMEDIA Software Architecture is based on the distributed NUBOMEDIA Functional Architecture. Each Software Component has been mapped to a different functional element designed by the Functional Architecture. In particular, each of the layer of the Functional Architecture consists of a mix between existing open source activities, extensions to those existing tools, and completely newly implemented ones. Figure 20 shows the Software Architecture diagram.

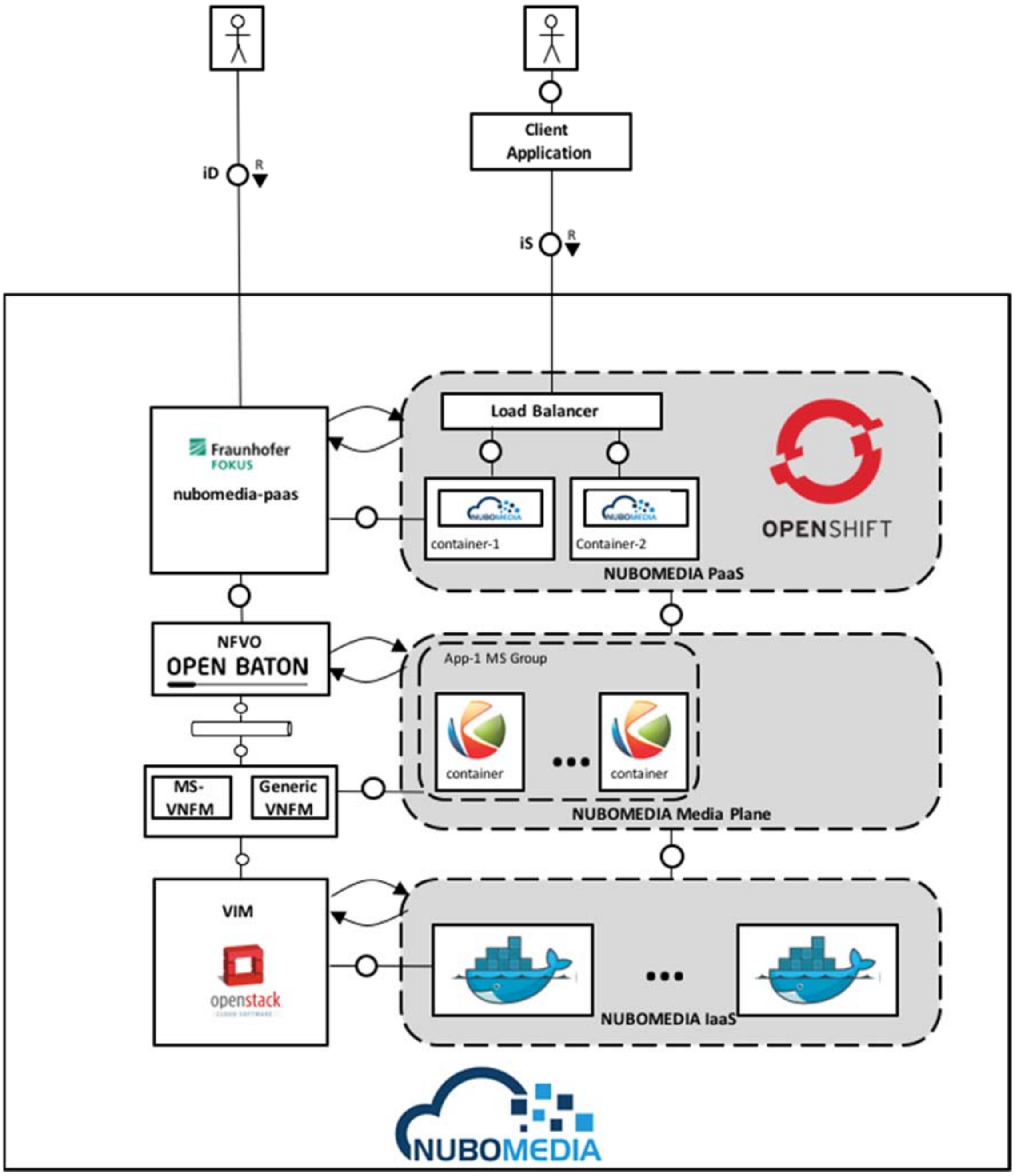


Figure 20. NUBOMEDIA Software Architecture

In particular:

- The NUBOMEDIA IaaS has been composed mainly by OpenStack Services extended for fully supporting instantiation of Docker containers on distributed Compute Nodes. Additionally, several monitoring and logging systems have been integrated into the OpenStack platform.
- For the Media Plane control elements, Open Baton[3] has been selected and further extended as fully compliant NFV Orchestrator (NFVO) [5].In order to manage the Media Server a new VNFM (VNFM-EMM) have been implemented, which extends the functionalities provided in previous releases by the EMM. While for the Cloud Repositories has been employed the Generic-VNFM, and a set of scripts have been implemented in order to control its lifecycle.
- The PaaS layer has been implemented with an existing open source component, OpenShift (add link), and a newly implemented software artifacts named nubomedia-paas mapped on top of the PaaS-API and PaaS-Manager functional element.

As a summary, the final Release of the NUBOMEDIA Cloud Platform has been composed by the following components and respective versions as listed in the table below.

Software Component	**Functional Element**	**Version**
OpenStack + nova-docker driver	Virtual Infrastructure Manager and NFV Infrastructure	Kilo
Open Baton	NFV Orchestrator	3.0.0
VNFM-EMM	Elastic Media Manager VNF Manager	3.0.0
Connectivity-Manager	Connectivity Manager	1.0.0
Connectivity-Manager-Agent	Connectivity Manager Agent	1.0.0
OpenShift	PaaS	Origin v3
NUBOMEDIA PaaS	NUBOMEDIA PaaS	1.3.4
NUBOMEDIA Marketplace	Marketplace	1.1.0

In order to simplify the development of the Platform, it has been provided a single GitHub repository where (almost) all the components developed or used in NUBOMEDIA have been added as submodules. The repository is available at: https://github.com/nubomedia/nubomedia-controller.

The Autonomous Installer has been further extended in order to simplify the deployment of the newest version of the Platform as per IST_R1. The description about the extended functionality is reported in the next sections. The documentation on how to use the Autonomous Installer is available at: http://nubomedia-developer-guidelines.readthedocs.io/en/latest/tools/autonomous-installer/

3.2 NUBOMEDIA IaaS

The IaaS is composed by hardware, software and networking resources providing an environment on top of which cloud services can be executed. NUBOMEDIA IaaS hides the underlying complexity of the IaaS physical resources and enables users to build their applications without the need of custom infrastructure.

3.2.1 Physical infrastructure

In order to determine the needed hardware equipment for creating an IaaS for NUBOMEDIA, and in special for WebRTC, we analyzed the hardware and software requirements of all platform components. Analyzing the system requirements of the Kurento Media Server we found that it requires Ubuntu 14.04 LTS (64bits) as Operating System to run[2], 1 x86_64 CPU and at least 1GB of RAM, the recommended value being 2GB of RAM.

The hardware requirement of a Compute Node, the component on top of which the Virtual Compute resources are executed, should support a 64bits architecture with at least 4GB of Memory, in order to provide 2GB to the compute node OS. Considering that a QEMU image for a compute node requires 1.3GB and the operating system requires ~10GB we should consider to have compute nodes with at least 20GB of disk space.

The networking should be based on SDN technologies in order to allow provisioning of dynamic traffic management mechanisms and to meet the SLAs of a real-time platform. All applications should have public IP addresses in order to be accessible from outside of the IaaS platform, meaning that we'll need to have a high number of public IPs available for being associated to different NUBOMEDIA components.

We carried out research to define strategies for the use of hardware accelerated facilities for NUBOMEDIA platform but we haven't found the PowerXCell8i a good candidate because of the following reasons:

- SPE Coprocessors from PowerXCell8i have only 256k memory and to utilize them it requires to do a memory update
- OpenStack Ironic is still not supporting PowerXCell8i virtualization
- They are huge power consumers, and because of that IBM removed them from production since few years ago.

3.2.2 Virtual Computing Infrastructure

3.2.2.1 Overview

One of the features that make the recently developed Cloud computing concept so appealing is the ability to provide, virtualize, and dynamically extend its resources as a service. So many IT technologies such as virtualization, storage, web services, service oriented architecture, clusters, etc., together with business models that deliver these IT capabilities as a service, on demand, scalable and elastic are all brought together by cloud computing [18].

The most recent computing trends point to IT resources as dynamically scalable, virtualized and exposed as a service on a local network or over the internet. Cloud computing is supposed to facilitate new mechanisms allowing users access to a virtually infinite number of resources on demand, using a pay-on-use system.

[2] http://www.kurento.org/docs/current/installation_guide.html

Some of the most well-known cloud computing providers worldwide are: Amazon Web Services (AWS), Microsoft Azure and Microsoft Private Cloud from Microsoft, Google Cloud, Rackspace and IBM bluemix. Regarding the private cloud computing providers, we should mention here the names of the most relevant, OpenStack and Apache CloudStack [18].

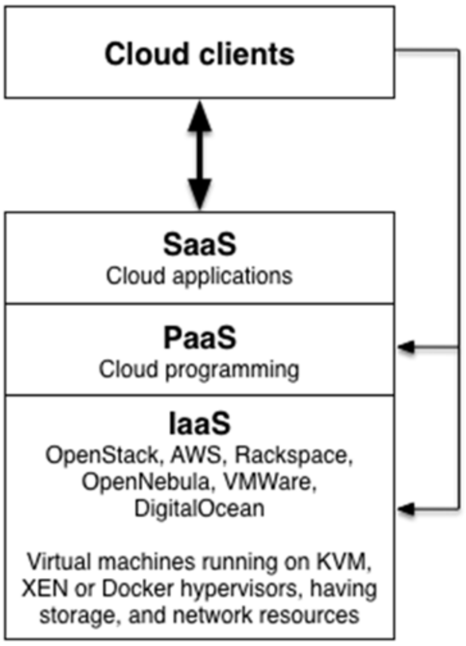
According to the definitions provided by NIST (National Institute of Standards and Technology of USA), Definition of Cloud Computing [19], cloud computing relies on three service models:

- Infrastructure as a Service (IaaS) which allows the consumer to provision processing, storage, networks, and related essential computing resources thus ensuring the deployment and running of arbitrary software, including operating systems and applications. The underlying cloud infrastructure is not managed nor controlled by the consumer. Instead, they may control the operating systems, storage, and deployed applications, and only limited control over the selection of networking components (e.g., host firewalls). OpenStack and CloudStack are considered leading names for IaaS private and public clouds, and Amazon EC2, Microsoft Azure, Google Cloud and Rackspace for IaaS public clouds.
- Platform as a Service (PaaS) allows the consumer the deployment of consumer-created or acquired applications created using provider supported programming languages, libraries, services, and tools onto IaaS. In this case, neither the management nor the control of IaaS the including network, servers, operating systems, or storage is a viable option for the consumer who may only control the deployed applications and configuration settings for the application-hosting environment. Key PaaS public providers are Heroku, Google Apps Engine, Windows Azure, Amazon AWS, and OpenShift and Cloud foundry are PaaS private solutions providers.
- Software as a Service (SaaS) offers the possibility to use the provider's applications that run on a cloud infrastructure. The client can access applications from their devices using a thin client interface (e.g., web-based email) or a program interface. The consumer is not allowed to manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or individual application capabilities. The only exceptions are the limited user-specific application configuration settings. Google Apps, salesforce, Twitter, Facebook, Flickr, FIWARE are well-known SaaS providers.

The lower level consists of physical hardware resources (computing, storage, network), and hypervisors that are able to virtualize access to the physical machine resources (CPU, memory and peripheral hardware). The instances of virtual servers are dynamically added or removed by the hypervisor on one physical server [1].

Well-known commercial cloud platforms such as Amazon Web Services (AWS) and Rackspace use Xen hypervisor whereas, AT&T, HP, Comcast, and Orange prefer KVM hypervisor. The early open source inclusion with Linux of Xen and KVM hypervisors brought them popularity. OpenStack project also favors KVM as hypervisor. Microsoft Azure and Microsoft Private Cloud from Microsoft use its Hyper-V hypervisor. Oracle Corporation provides Oracle VM VirtualBox hypervisor for x86 processors.

The second level requires no knowledge of the hardware or operating system management to deploy and manage applications.


Cloud clients
SaaS
Cloud applications
PaaS
Cloud programming
IaaS
OpenStack, AWS, Rackspace, OpenNebula, VMWare, DigitalOcean
Virtual machines running on KVM, XEN or Docker hypervisors, having storage, and network resources
Figure 21. Relationship between clients and SaaS/PaaS/IaaS systems

When we searched for IaaS solutions for implementing NUBOMEDIA we found few options from which the most popular is OpenStack, which is a private cloud solution based on free open source components which meets the requirements for the physical infrastructure of NUBOMEDIA. OpenStack has a very big community with more than 4,788 developers (http://activity.openstack.org/dash/browser/) across the whole globe, being supported by all major players like CISCO, IBM, Intel, RedHat, Juniper (https://www.openstack.org/foundation/companies/). With this kind of momentum, OpenStack is not only being labeled as “the next Linux” but it is gaining adoption at a rate that will hit the same critical mass in five years that took Linux 15 years. This is also be confirmed by Google trends. (Figure 22).

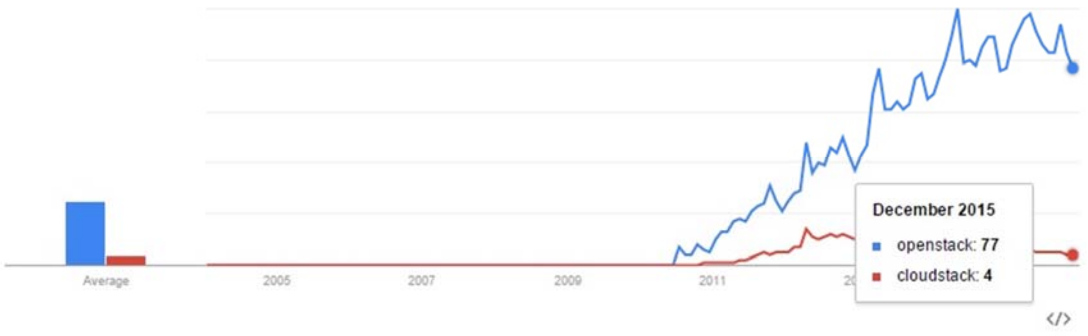


Figure 22. Google trends for OpenStack vs CloudStack

3.2.2.2 NUBOMEDIA IaaS deployed architecture

The IaaS is composed of a master node and several compute nodes:

- 1 Master node running the following OpenStack services:
  - nova-scheduler service, for allocating VMs to the compute nodes
  - cinder-scheduler service, for allocating block storage on the compute nodes
  - glance-registry service, for managing the KVM and Docker images
  - neutron-server service, for managing the network inside the instances
  - heat-api service, for creating and orchestrating services on OpenStack
  - keystone service, for managing the identity inside OpenStack
- N x Compute nodes running the following services:

- nova-compute service, running KVM or Docker hypervisor
- neutron-agent service, for managing VM inside the compute node

3.2.2.3 Docker hypervisor

OpenStack supports a large variety of hypervisors. A list of all hypervisors available on OpenStack can be found here[3]. From all hypervisor types, containers are for sure a hot topic today and The OpenStack User Survey indicates that more than half of the participants are interested in working with containers on top of their OpenStack public or private cloud. Thanks to open source initiatives, Docker containers have gained significant popularity lately among developers (Figure 21).

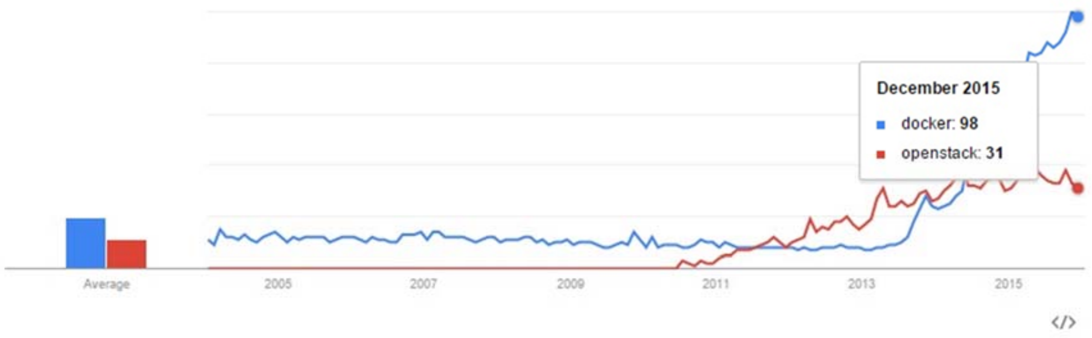


Figure 23. Docker vs OpenStack

Rather than running a full OS on virtual hardware like KVM does, container based virtualization like Docker uses an existing OS and modifies it in order to provide extra isolation. Generally, this involves adding a container ID to each process and adding new access control checks for every system call. Thus, containers can be viewed as an additional level of access control in addition to the user and group permission system. In practice, Linux uses a more complex implementation described below.

Linux containers are a concept built on the kernel namespaces feature, originally motivated by difficulties in dealing with high performance computing clusters. This feature, accessed by the clone() system call, allows creating separate instances of previously-global namespaces. Linux implements filesystem, PID, network, user, IPC, and hostname namespaces. For example, each filesystem namespace has its own root directory and mount table, similar to chroot() but more powerful.

Namespaces can be used in many different ways, but the most common approach is to create an isolated container that has no visibility or access to objects outside the container. Processes running inside the container appear to be running on a normal Linux system although they are sharing the underlying kernel with processes located in other namespaces. Containers can nest hierarchically, although this capability has not been much explored so far.

Unlike a VM, which runs a full operating system, a container can contain as little as a single process. A container that behaves like a full OS and runs init, inetd, sshd, syslogd, cron, etc. is called a system container while one that only runs an application is called an application container. Both types are useful in different circumstances. Since an application container does not waste RAM on redundant management processes it generally consumes less RAM than an equivalent system container or VM. Application

[3] http://docs.openstack.org/developer/nova/support-matrix.html

containers generally do not have separate IP addresses, which can be an advantage in environments that lack IP address spaces.

If total isolation is not desired, it is easy to share some resources among containers. For example, bind mounts allow a directory to appear in multiple containers, possibly in different locations. This is implemented efficiently in the Linux VFS layer. Communication between containers or between a container and the host (which is really just a parent namespace) is as efficient as normal Linux IPC.

The Linux control groups (cgroups) subsystem is used to group processes and manage their aggregate resource consumption. It is commonly used to limit the memory and CPU consumption of containers. A container can be resized by simply changing the limits of its corresponding cgroup. Cgroups also provide a reliable way of terminating all processes inside a container. Because a containerized Linux system only has one kernel and the kernel has full visibility into the containers there is only one level of resource allocation and scheduling.

An unsolved aspect of container resource management is the fact that processes running inside a container are not aware of their resource limits. For example, a process can see all the CPUs in the system even if it is only allowed to run on a subset of them; the same applies to memory. If an application attempts to automatically tune itself by allocating resources based on the total system resources available, it may over-allocate when running in a resource-constrained container.

Securing containers tends to be simpler than managing Unix permissions because the container cannot access what it cannot see and thus the potential for accidentally over-broad permissions is greatly reduced. When using user namespaces, the root user inside the container is not treated as root outside the container, adding additional security. The primary type of security vulnerability in containers is system calls that are not namespace-aware and thus can introduce accidental leakage between containers. Because the Linux system call API set is huge, the process of auditing every system call for namespace-related bugs is still ongoing. Such bugs can be mitigated (at the cost of potential application incompatibility) by whitelisting system calls using seccomp.

Due to its feature set and ease of use, Docker has rapidly become the standard management tool and image format for containers. A key feature of Docker not present in most other container tools is layered filesystem images, usually powered by AUFS (Another UnionFS). AUFS provides a layered stack of filesystems and allows reuse of these layers between containers reducing space usage and simplifying filesystem management. A single OS image can be used as a basis for many containers while allowing each container to have its own overlay of modified files (e.g., app binaries and configuration files). In many cases, Docker container images require less disk space and I/O than equivalent VM disk images. This leads to faster deployment in the cloud since images usually have to be copied over the network to local disk before the VM or container can start. Our measurements have shown than containers can start much faster than VMs (less than 1 second compared to 20 seconds on our hardware) because unlike VMs, containers do not need to boot another copy of the operating system. In theory we can even perform live migration of containers, but it may be faster to kill a container and start a new one if we can mount the same data on the other host.
To conclude, administrators and developers are interested in containers for 4 major reasons:

1. Fast boot time. Our measurements have shown than containers can start much faster than VMs (less than 1 second compared to 2 minutes on our hardware) because unlike VMs, containers do not need to boot another copy of the operating system.
2. No need to migrate container, you can start a new one and just balance the traffic to the newly created container and kill the old one, all this in just few seconds.
3. Application containers, compared with virtual machines, are very lightweight – minimizing compute, storage, and bandwidth requirements for deployment
4. Have direct access to hardware functions
5. The other advantage is that containers are portable, effectively running on any hardware that runs Linux based operating system. That means developers can run a container on a workstation, create an app in that container, save it in a container image, and then deploy the app on any virtual or physical server running the same operating system - and expect the application to work[4]

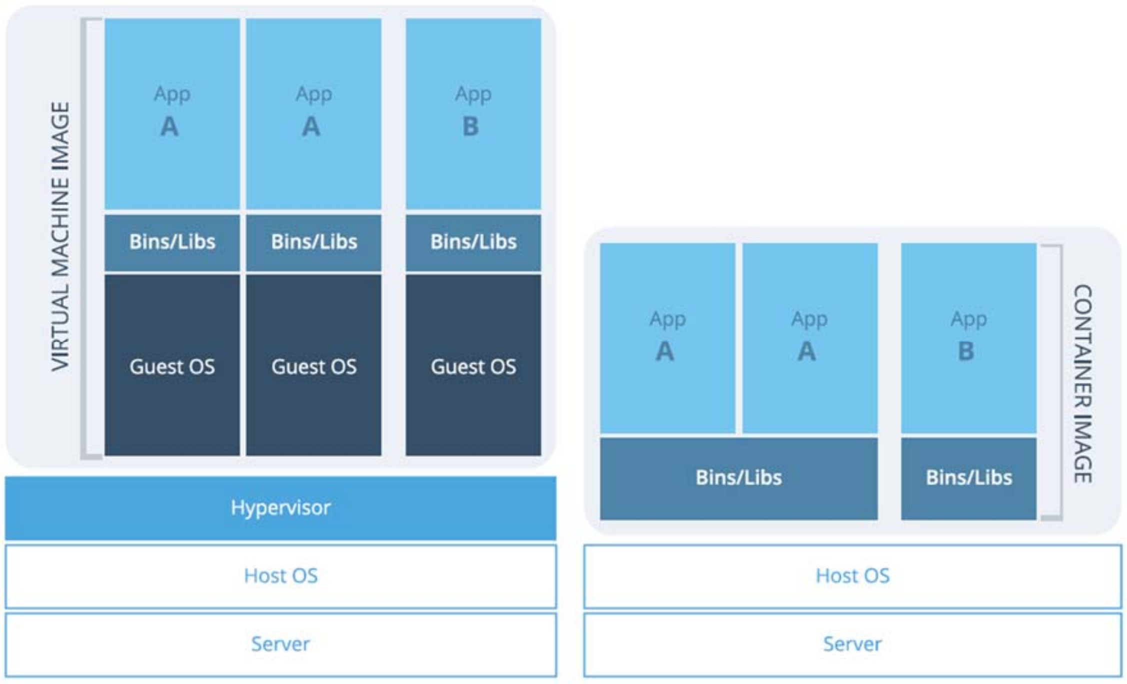


Figure 24. Docker vs KVM

Our objective is to allow users to create and manage containers with an experience consistent with what they are used to have using the Nova service to get virtual machines. The aim is to offer developers a single set of compatible APIs to manage their workloads, whether those run-on containers or virtual machines.

There are multiple OpenStack projects leveraging container technology to make OpenStack better: Magnum, Kolla and Murano. Basically:

- Magnum is designed to offer container specific APIs for multi-tenant containers-as-a-service with OpenStack.
- Kolla is designed to offer a dynamic OpenStack control plane where each OpenStack service runs in a Docker container.
- Murano is an application catalog solution that allows for packaged applications to be deployed on OpenStack, including single-tenant installations of Kubernetes.

[4] https://www.openstack.org/assets/pdf-downloads/Containers-and-OpenStack.pdf

- Nova-docker is a hypervisor driver for Openstack Nova Compute. It was introduced with the Havana release, but lives out-of-tree for Icehouse, Juno and Kilo. Being out-of-tree has allowed the driver to reach maturity and feature-parity faster than would be possible should it have remained in-tree. Now the driver will return to upstream Nova in the Liberty release.

On NUBOMEDIA we decided to use Docker as a hypervisor with nova-docker on top of OpenStack because of the image size, direct access to hardware resources and boot time needed in real-time micro-service based applications like NUBOMEDIA.

3.2.3 Docker as hypervisor

The Docker driver is a hypervisor driver for Openstack Nova Compute. It was introduced with the Havana release, but lives out-of-tree for Icehouse and Juno. Being out-of-tree has allowed the driver to reach maturity and feature-parity faster than would be possible should it have remained in-tree.[5]

Docker is an open-source engine which automates the deployment of applications as highly portable, self-sufficient containers which are independent of hardware, language, framework, packaging system and hosting provider.

Docker provides management of Linux containers with a high-level API providing a lightweight solution that runs processes in isolation. It provides a way to automate software deployment in a secure and repeatable environment. A Docker container includes a software component along with all of its dependencies - binaries, libraries, configuration files, scripts, virtualenvs, jars, gems, tarballs, etc. Docker can be run on any x64 Linux kernel supporting cgroups and aufs.

Docker is a way of managing multiple containers on a single machine. However, used behind Nova makes it much more powerful since it's then possible to manage several hosts, which in turn manage hundreds of containers. The current Docker project aims for full OpenStack compatibility.

Containers don't aim to be a replacement for VMs, they are complementary in the sense that they are better for specific use cases. In example VMs are better for cases when you save data on the actual instance because with docker you lose all the data when you restart a container.

Considering the advantages and disadvantages of containers, we configured several compute nodes with Docker as a hypervisor and performed several tests to prove that now Docker is a production ready technology on OpenStack.

How does the Nova hypervisor works?
The Nova driver embeds a tiny HTTP client which talks with the Docker internal Rest API through a Unix socket. It uses the HTTP API to control containers and fetch information about them.

The driver will fetch images from the OpenStack Image Service (Glance) and load them into the Docker file-system. Images must be placed in Glance by exporting them from Docker using the 'docker save' command.

[5] https://wiki.openstack.org/wiki/Docker

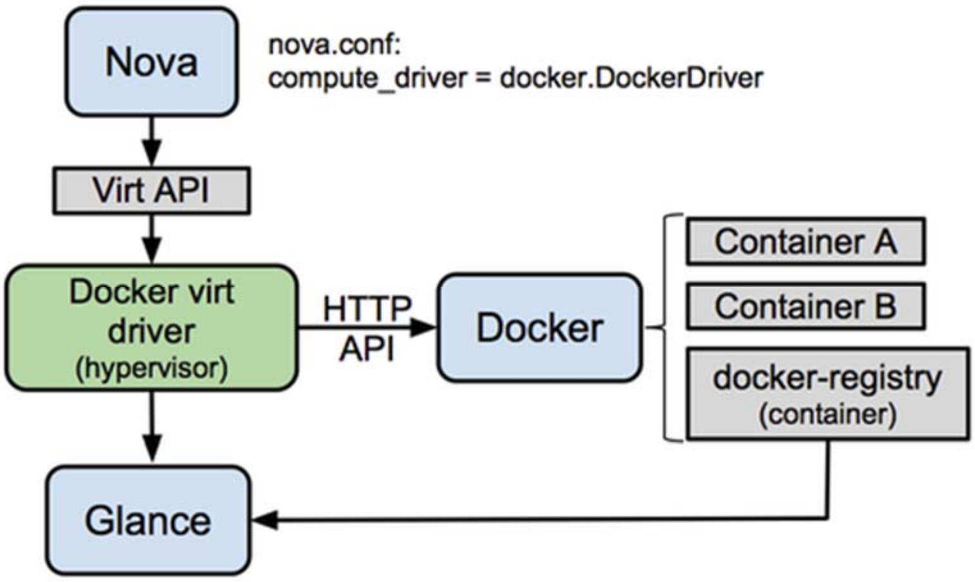
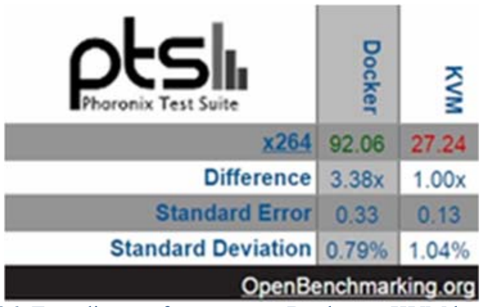

Nova
nova.conf:
compute_driver = docker.DockerDriver
Virt API
Docker virt driver (hypervisor)
HTTP API
Docker
Container A
Container B
docker-registry (container)
Glance

Figure 25. Nova-docker architecture

We have done performance tests comparing Docker and KVM instances running the same flavors and found that almost for each test Docker won the battle. For almost all the tests we used OpenBenchmarking.org for generating the graphics:

- Boot time between a Docker instance and a KVM instance. For this we created a python script, which can be found here[6] . It measures the time needed for a fresh Ubuntu 14.04 x86_64 KVM image to boot and the time needed for a Docker container with ubuntu:14.04, but it does not take into consideration the time needed for the Ubuntu to actually run the initrd and user space.
- Encoding with a video using x264 H.264/AVC, using just the CPU (with OpenCL support deactivated)


pts Phoronix Test Suite
OpenBenchmarking.org

	Docker	KVM
x264	92.06	27.24
Difference	3.38x	1.00x
Standard Error	0.33	0.13
Standard Deviation	0.79%	1.04%

Figure 26. Encoding performance on Docker vs KVM hypervisor

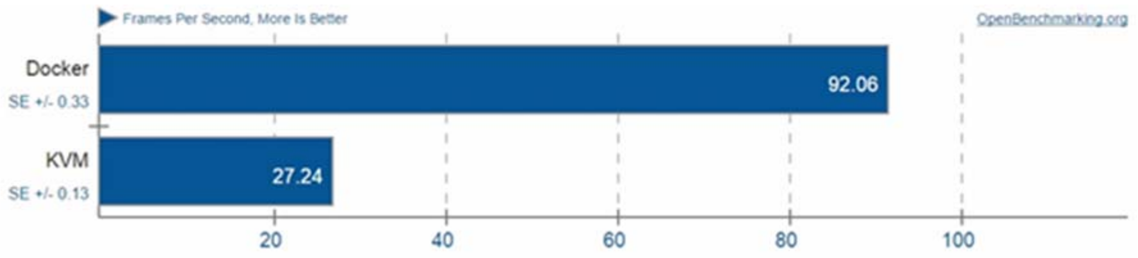


Figure 27 Encoding performance on Docker vs KVM hypervisor 2

[6] https://github.com/alincalinciuc/instance-boot-time-openstack

SciMark2 runs the ANSI C version of SciMark 2.0, which is a benchmark for scientific and numerical computing developed by programmers at the National Institute of Standards and Technology. This test is made up of Fast Fourier Transform, Jacobi Successive Over-relaxation, Monte Carlo, Sparse Matrix Multiply, and dense LU matrix factorization benchmarks.

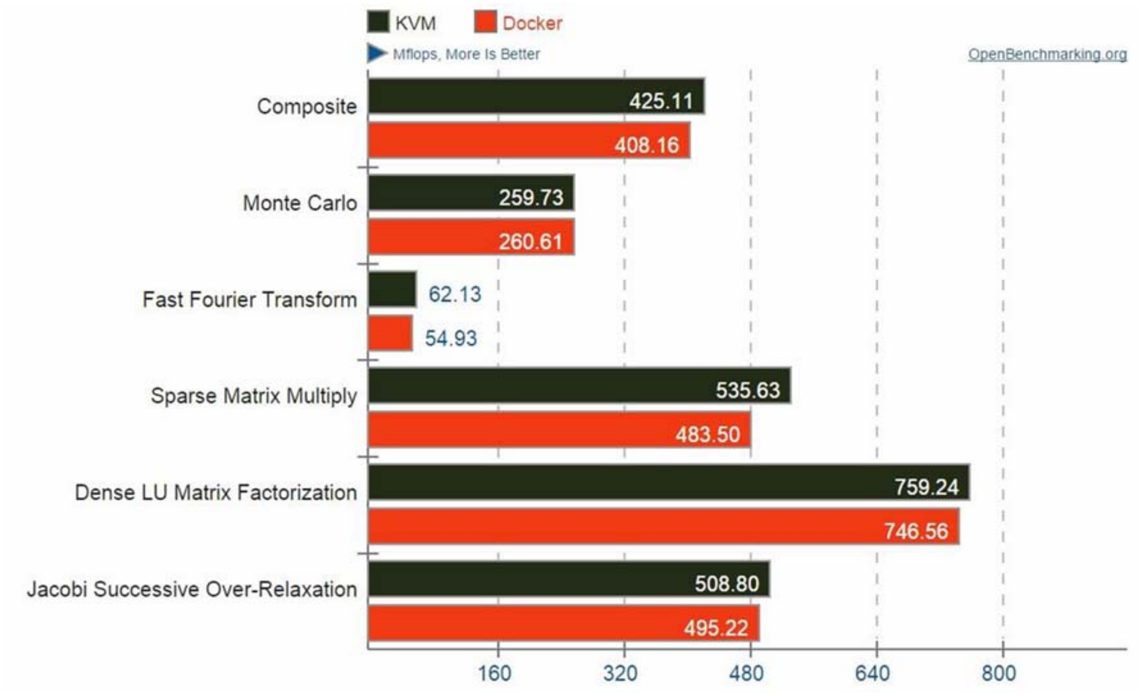


Figure 28 Docker vs KVM running scientific and numerical computation

For memory testing we used ramspeed and stream.

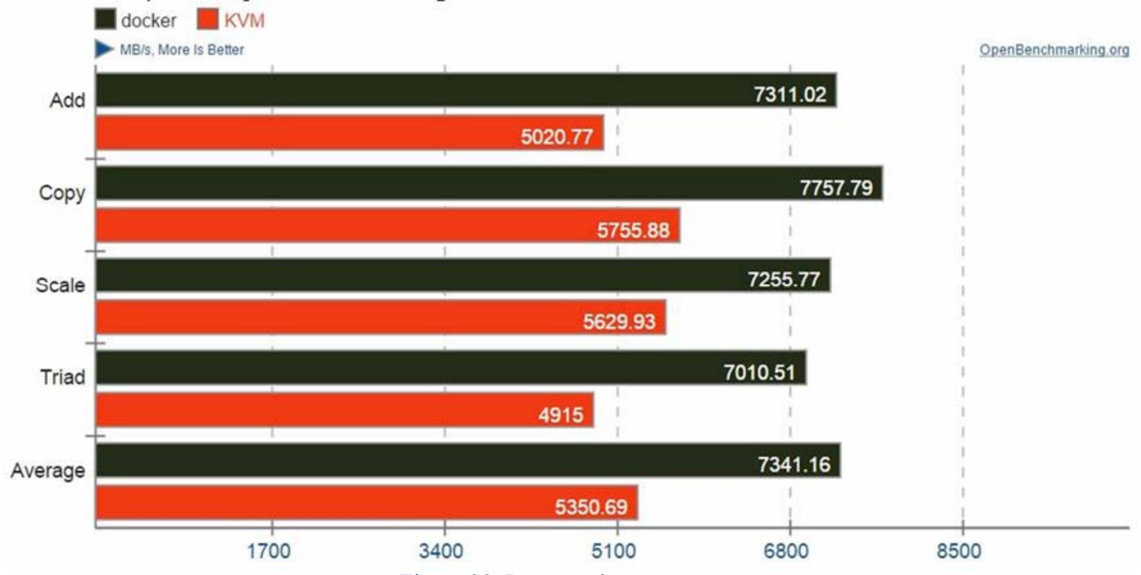


Figure 29. Ramspeed memory tests

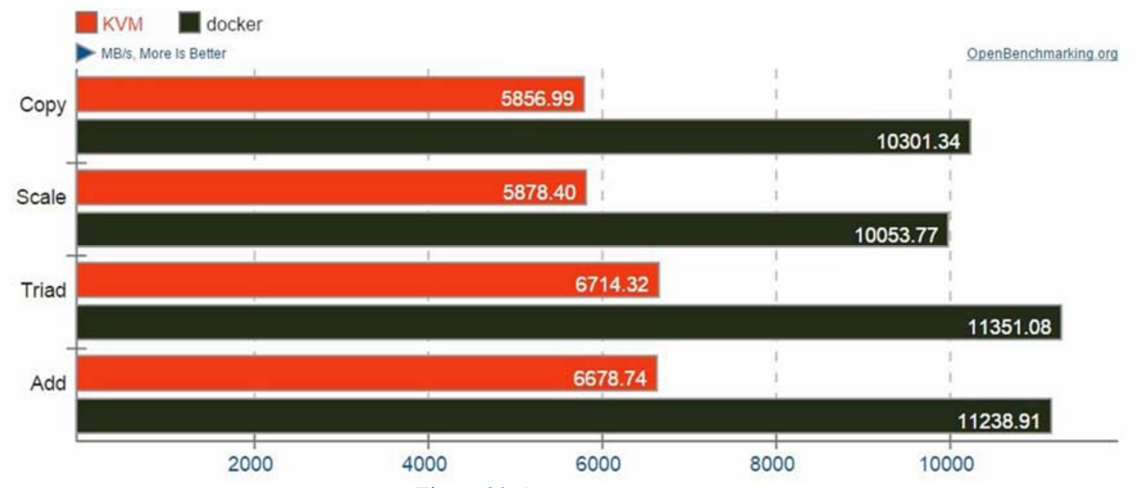


Figure 30. Stream memory tests

3.2.3.1 Nova-docker customization

The missing thing of nova-docker is that when you add a new docker image that image is not available on all compute nodes running the docker as hypervisor, and you have to login on all of them and run *docker pull image-name* in order to have it available on each compute node. If you are not doing this thing you will not be able to instantiate new instances using the newly added image. Another problem was that docker images that are started on OpenStack don't have associated any public key on them and SSHD is not running on boot, so you can't actually login on them in case you need to do any debugging or anything like that on them.

The Docker instances have now all ports automatically exposed, the configured public key is added to the machines and on start the init script is ran. All these make from Docker an ideal environment for NUBOMEDIA components.

We developed a python application that pulls Docker images from the public repository on each compute node that has Docker as hypervisor. We also do cleanups for unused docker images in order to keep the physical machines clean. For R6 we plan to integrate our code with the nova-docker[7] main repository in order to allow other IaaS operators to easily operate docker as hypervisor in their environments.

The source-code for the nova-docker patch can be found on the following GitHub repository: https://github.com/usv-public/nubomedia-nova-docker

3.2.3.2 The selected OpenStack distribution

For NUBOMEDIA release 6 we have installed, configured an IaaS based on OpenStack Kilo which was released in May 2015. The master nodes are all running on top of CentOS 7.1 which is a Community Enterprise Operating System, being a free rebuild of source packages from the Red Hat Enterprise Linux.

For the OpenStack deployment, we evaluated two possible choices: RedHat RDO and Mirantis Fuel.

- Fuel offers a very nice and intuitive interface for deploying and managing the whole IaaS but it lacks the support for new and emerging modules like nova-docker for supporting Docker as a hypervisor inside OpenStack and neutron plugin ML2 that allow the usage of VXLANs, VLANs, GRE tunnels and flat

[7] https://github.com/openstack/nova-docker

networking at the same time. For this reason, we decided to use RDO (Red Hat OpenStack).
- RDO is a community software used for deploying OpenStack on CentOS, Fedora, and Red Hat Enterprise Linux. On the RDO FAQ page we find that RDO stands for RPM Distribution of OpenStack or Rediculously Dedicated OpenStackers who are here to help you Rapidly Deploy OpenStack, in a way that is Really Darned Obvious. RDO is Rebuilt Daily, Regularly Delivered, OpenStack.

To install and configure OpenStack we used the tool named Packstack from RDO. Packstack is an installation utility which uses Python and Puppet modules to deploy OpenStack.

3.2.3.3 Neutron ML2

On the previous release, we used OpenStack Havana with openvswitch plugin that allowed us to have the internal networking based on tagged VLANs. This required us to configure the hardware switches in the appropriate way to support the traffic for any new tenant. At the moment with the new Neutron plugin we can use any type of networking without doing any configurations at the hardware level.

Since Juno release a new Neutron plugin was lunched, ML2 (Modular Layer 2), which is a production ready neutron plugin that allows OpenStack networking to simultaneously utilize a variety of layer 2 network technologies like VLAN, VXLAN, Flat and GRE tunnels in complex real-world data centers. It currently works with the existing openvswitch, linuxbridge, and Microsoft Hyper-V L2 agents, and is intended to replace and deprecate the monolithic plugins associated with those L2 agents. The ML2 framework is also intended to greatly simplify adding support for new L2 networking technologies, requiring much less initial and ongoing effort than would be required to add a new monolithic core plugin.

3.2.3.4 ML2 drivers

Drivers within ML2 implement separately extensible sets of network types and of mechanisms for accessing networks of those types. Unlike with the metaplugin, multiple mechanisms can be used simultaneously to access different ports of the same virtual network. Mechanisms can utilize L2 agents via RPC and/or use mechanism drivers to interact with external devices or controllers. Type and mechanism drivers are loaded as python entry points using the stevedore library.

3.2.3.4.1 Type Drivers

Each available network type is managed by an ml2 TypeDriver. TypeDrivers maintain any needed type-specific network state, and perform provider network validation and tenant network allocation. The ml2 plugin currently includes drivers for the local, flat, vlan, gre and vxlan network types.

3.2.3.4.2 Mechanism Drivers

Each networking mechanism is managed by an ml2 MechanismDriver. The MechanismDriver is responsible for taking the information established by the TypeDriver and ensuring that it is properly applied given the specific networking mechanisms that have been enabled.

The Mechanism Driver interface currently supports the creation, update, and deletion of network and port resources. For every action that can be taken on a resource, the mechanism driver exposes two methods - ACTION_RESOURCE_precommit, which is called within the database transaction context, and ACTION_RESOURCE_postcommit, called after the database transaction is complete. The pre-commit method is used by mechanism drivers to validate the action being taken and make any required changes to the mechanism driver's private database. The precommit method should not block, and therefore cannot communicate with anything outside of Neutron. The post-commit method is responsible for appropriately pushing the change to the resource to the entity responsible for applying that change. For example, the post-commit method would push the change to an external network controller, that would then be responsible for appropriately updating the network resources based on the change.

3.2.3.5 ML2 in NUBOMEDIA

We configured Neutron in order to use ML2 with VXLAN (Virtual Extensible LAN) network type drivers. VXLAN is a network virtualization technology that attempts to ameliorate the scalability problems associated with large cloud computing deployments. It uses a VLAN-like encapsulation technique to encapsulate MAC-based OSI layer 2 Ethernet frames within layer 4 UDP packets. We used VXLANs as network type because it has more flexibility when it comes to new network creation on Neutron. With regular VLANs you should first create a tagged VLAN on the switch that connects all compute nodes, then you have to enable that VLAN for all compute node ports that you have connected on OpenStack, and if you, let’s say had compute nodes in another building or at another floor level, at 3 switch hops, you then should have created a VLAN trunk to that location, having to configure 4 or more switches, for just one new tenant that needs a new LAN. With VXLAN this becomes even simple, you just have to define the VXLAN VNI IDs that are available for tenant network allocation with the minimum value is 0 and maximum value is 16777215. This means you can have up to 16777215 networks inside a single OpenStack deployment, while with VLANs the maximum number would have been 4096. Using VXLAN you also don’t have to configure anything on the switches because it encapsulates MAC-based OSI Layer 4 frames within layer 4 UDP packages that are then send between compute nodes using IP protocol.

3.2.4 OpenStack Security

On a normal OpenStack deployment, the identity part is configured to allow access to its entry-points only from the local network. For this we had to configure the identity service to allow requests from the outside network for almost all services and also to open the firewall for the ports used on those entry-points.

By default, the IaaS dashboard is running on HTTP 80 port. HTTP, or hypertext transfer protocol, is the way a Web server communicates with browsers. HTTP lets visitors view a site and send unencrypted information back to the Web server. HTTPS on the other hand is HTTP through a secured connection. Communications with an HTTPS server are encrypted by a secure certificate known as an SSL. The encryption prevents third-parties from eavesdropping on communications to and from the server. Considering this we generated a free SSL certificate using the open source certificate provider from letsencrypt.org. Let’s Encrypt is a free, automated, and open certificate authority created by Internet Security Research Group (ISRG).

3.2.5 NUBOMEDIA Infrastructure Level Monitoring and Debugging Tools

In order to be able to monitor also the performances of the Multimedia Applications running on the PaaS, it was necessary to introduce a Real-Time Monitoring System. Furthermore, it was necessary to expose those monitored metrics through APIs in order to be consumed by external components like the NFVO and VNFM. This system was based on Graphite and is explained in more details on 3.2.5.1

For monitoring of hardware infrastructure which was critical for enabling NUBOMEDIA operator to optimize the power usage of the hardware we introduced Icinga.

3.2.5.1 Graphite

The OpenStack solution, Ceilometer, is based on MongoDB and is developed to have entry points of minutes (1 minute or 5 minutes) which is not satisfying the requirements of real time monitoring information. Additionally, to scale Ceilometer we had to optimize and scale MongoDB database. Considering these factors, we decided to use an opensource tool called Graphite [10]. The tool provides a time-series optimized database on which the performance depends only on raw performance of the storage. Also, Graphite provided visualization tools and a basic server to collect the data.

Current API for collecting data from Graphite is a basic TCP socket and pushing metrics was not a familiar method for developers so we expanded Graphite by integrating an additional REST API based on Backstop [11]. We created a guide for NUBOMEDIA developers to use the both APIs for submitting metrics and also for reading them.

For optimizing the realtime usage of Graphite we tested multiple storage options to find the parameters to be able to collect and display the data in realtime (1 to 3 seconds' intervals).

Metrics from containers and virtual machines are pushed with collectd tool which was installed and configured on all the images.

3.2.5.2 Logstash

For log collection, we deployed and customized a tool based on an opensource tool Logstash [12] that is collecting centrally all the logs for NUBOMEDIA operator. We configured Logstash for all the logs including containers (which normally are difficult to be accessed) are pushed and available in the dashboard. For pushing the logs to the tool, we used an open source tool logstash-forwarder which was installed and configured on all instances.

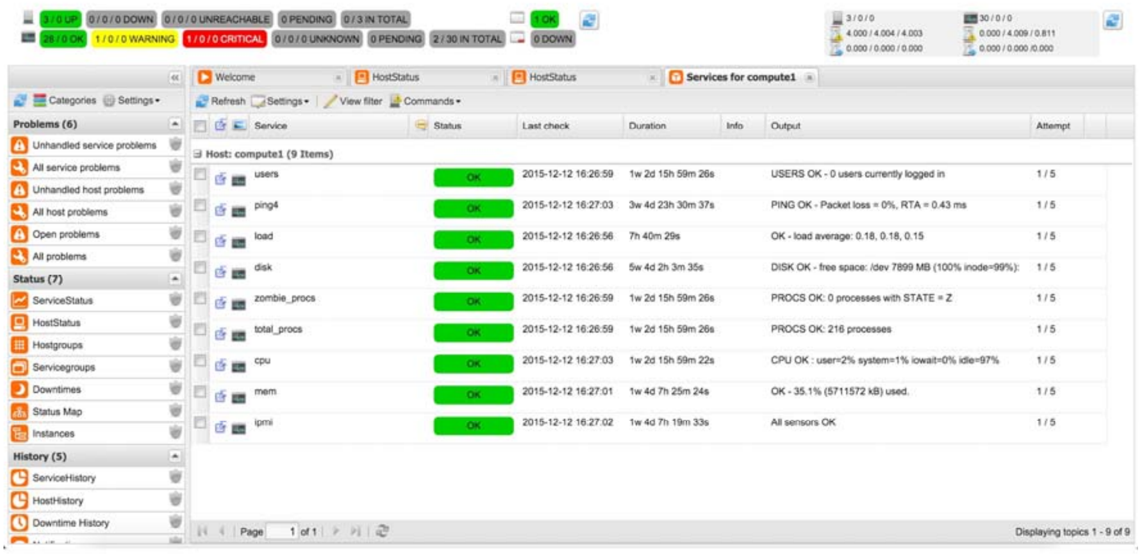
For visualizations of logs by NUBOMEDIA operator we integrated in the Management Console an open source tool Kibana which is part of the Logstash. The tool was integrated into the NUBOMEDIA console. In the Appendix, it is possible to find a guide on how to interact with the monitoring system.

On the last year of the project we have updated the monitoring tool that collects data from the Kurento Media Server by using its internal APIs and publishes those metrics to Logstash in order to make them available on the NUBOMEDIA PaaS monitoring tab on each application.

3.2.5.3 Icinga

Physical machines (hosts) which are of interest for NUBOMEDIA operators, are monitored (performance usage and hardware stats) with an open source tool Icinga [13]. We customized Icinga so all the metrics recorded will be stored on Graphite, so they can be accessed through Graphite APIs.

Icinga system monitors hardware failures, disk failures, network parameters, and sends email alerts to system administrators that can take appropriate actions to minimize the effects on the NUBOMEDIA physical infrastructure.



Host: compute1 (9 Items)

Service	Status	Last check	Duration	Info	Output	Attempt
users	OK	2015-12-12 16:26:59	1w 2d 15h 59m 26s		USERS OK - 0 users currently logged in	1 / 5
ping4	OK	2015-12-12 16:27:03	3w 4d 23h 30m 37s		PING OK - Packet loss = 0%, RTA = 0.43 ms	1 / 5
load	OK	2015-12-12 16:26:56	7h 40m 29s		OK - load average: 0.18, 0.18, 0.15	1 / 5
disk	OK	2015-12-12 16:26:56	5w 4d 2h 3m 35s		DISK OK - free space: /dev 7899 MB (100% inode=99%):	1 / 5
zombie_procs	OK	2015-12-12 16:26:59	1w 2d 15h 59m 26s		PROCS OK: 0 processes with STATE = Z	1 / 5
total_procs	OK	2015-12-12 16:26:59	1w 2d 15h 59m 26s		PROCS OK: 216 processes	1 / 5
cpu	OK	2015-12-12 16:27:03	1w 2d 15h 59m 22s		CPU OK : user=2% system=1% iowait=0% idle=97%	1 / 5
mem	OK	2015-12-12 16:27:01	1w 4d 7h 25m 24s		OK - 35.1% (5711572 kB) used.	1 / 5
ipmi	OK	2015-12-12 16:27:02	1w 4d 7h 19m 33s		All sensors OK	1 / 5

Figure 31. Icinga dashboard displaying status of a host OpenStack compute

Using Icinga we are monitoring the entire physical infrastructure that hosts the IaaS. We gather metrics from the two Bladecenter H chases and also from all the 25 compute nodes that are hosting the IaaS on top of which the NUBOMEDIA platform is deployed.

Metrics that are gathered are the following:

- ambient temperature of the two Bladecenters
- blower speed of the fans that are cooling the Bladecenters
- power consumption of each compute node
- total power consumption of each bladecenter
- CPU load for each compute node
- available/used disk space for each compute node
- used/free memory
- number of running or stopped virtual machines on each compute node (https://github.com/alincalinciuc/nagiosplugins)

We collected basic resource usage as CPU, memory, networking and disk and also advanced metrics like energy consumption of each physical machine. Using an IPMI interface and SNMP we developed Icinga scripts to monitor power consumption, voltages and temperatures of the hardware infrastructure. Fault detection was configured to be sent to NUBOMEDIA operators in cases of failures or thresholds were passed.

Besides monitoring the services and providing alerts, Icinga also generates reports of availability of the hosts machines. Stats from agents and Icinga are pushed to a collecting software component based on Graphite that was fine-tuned to meet the real-time requirements of the NUBOMEDIA project.

3.2.5.4 Energy policies

There are multiple ways of monitoring the physical infrastructure behind an OpenStack IaaS. Some good examples of tools that perform energy monitoring are: Kwapi, Nagios and Telemetry with IPMI agents.

With the help of the monitoring tools we are collecting basic resource usage as CPU, memory, networking and disk and also advanced metrics like energy consumption for each physical machine. Using an IPMI interface and SNMP we developed Icinga scripts to monitor power consumption, voltages and temperatures of the hardware infrastructure. Fault detection was configured to be sent to NUBOMEDIA operators in cases of failures or thresholds were passed.

Using the information provided by the monitoring system, the NUBOMEDIA expert system administrator can decide to stop physical compute nodes in order to optimize energy consumption and to start powered off compute nodes when the load on the platform is increasing.

For this we analyzed the possibility to create a Python application to manage the lifecycle of physical compute nodes automatically, but after checking the time needed for a physical compute node to start, we concluded that by doing this, the IaaS will not be able to meet the needs of real time applications similar to the ones that are deployed on NUBOMEDIA platform. We concluded that the load of a production environment of NUBOMEDIA will require to start many KMS instances in a very short period and we can't afford to wait the time needed for a physical machine to boot.
In the following chart, we can see that for booting a physical machine it requires more than 3 minutes before the system passes the bios, which also requires ~1 min., so in total 4-5 minutes to have a compute node powered on.

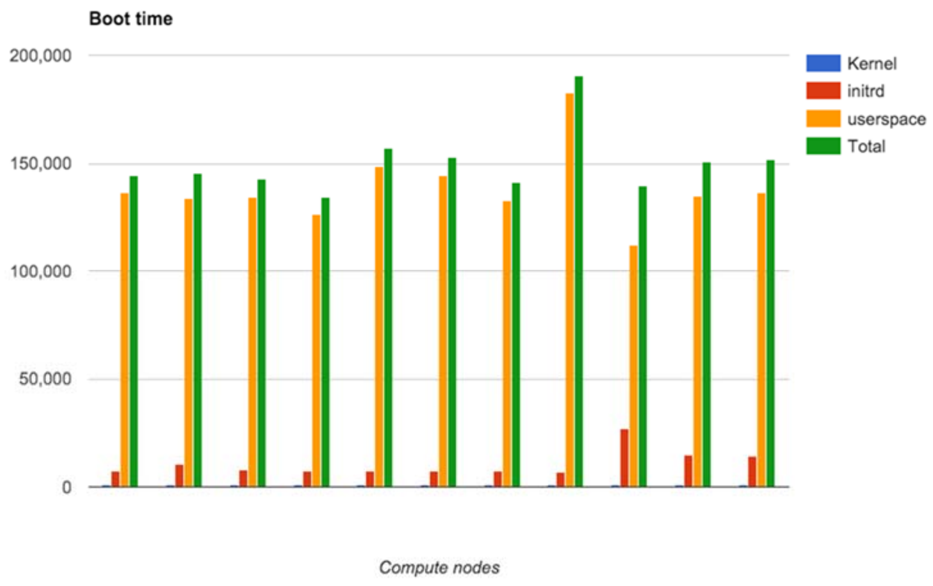


Figure 32 Compute nodes boot time

Another tool that was analyzed is Kwapi which supports different kinds of energy consumption metrics via IP or serial links. In order to setup Kwapi you have to install drivers that are connected to one or more sensors and send their values each second to

Ceilometer. All the metrics from Kwapi are then published for visualization on a web interface based on RRDtool and Flask like the following one.

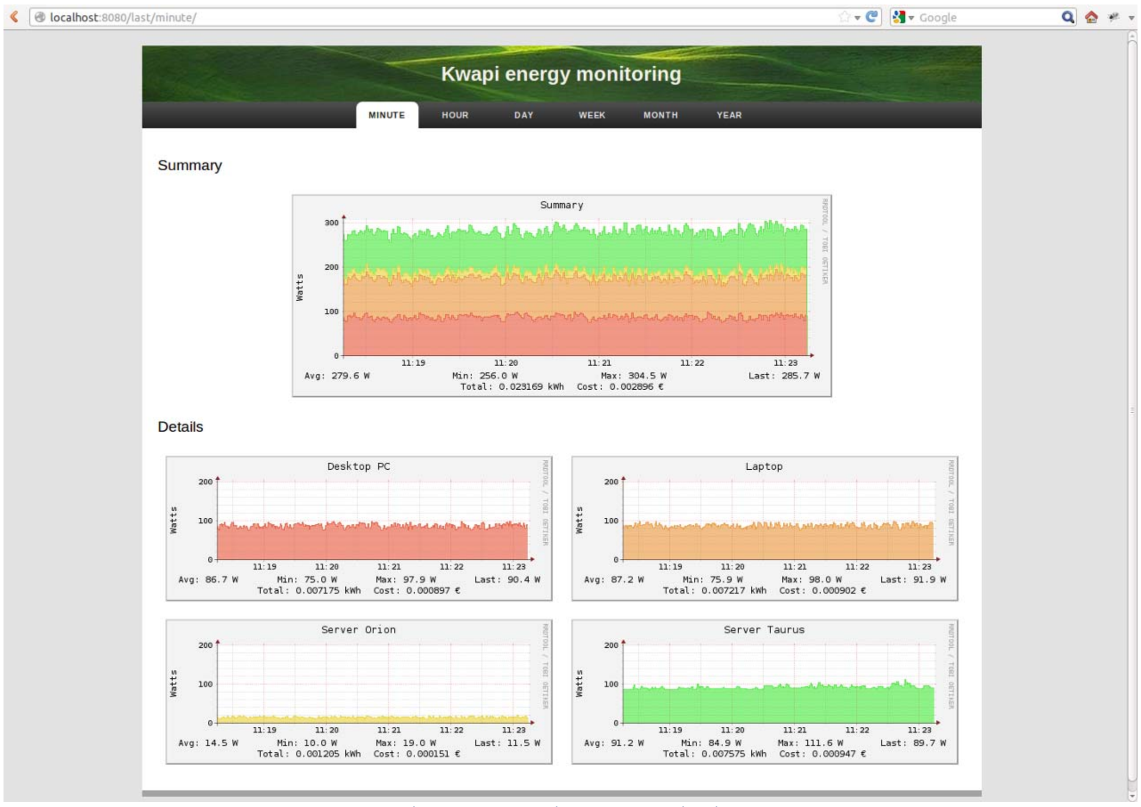


Figure 33 Kwapi energy monitoring

By using the energy consumption metrics provided by Kwapi in conjunction with the instance information provided by Nova we can decide what and if one node can be shut-down in order to minimize the energy consumption. In order to automate this process, we should install and configure a tool that offers us a better understanding of the usage of resources. A good example of such tool is the Arcus OpenStack Energy Monitoring Tool[8]. The tool will enable you to understand basic parameters relating to both the energy consumption and usage of servers inside the OpenStack IaaS. As described in the paper *Adding Energy Efficiency To Openstack*[9] the operating range of the compute nodes can vary from 150W to 300W depending on the processing jobs that are running on each node, so we save 150W for the time each compute node is powered off.

There are also some downsides for powering off machines that are unused for small period of times. The hardware is degraded much more if it is put to multiple start-up cycles. The energy consumption when the hardware powers on is also high. Another downside for the NUBOMEDIA platform is that Docker containers cannot be used for live-migration and that the sessions are immutable, so we cannot move users from one server to another in order to power it down to save energy.

[8] https://github.com/icclab/arcus-energy-monitoring-tool
[9] http://dl.ifip.org/db/conf/ifip6-3/sustainit2015/CimaGMB15.pdf

3.2.5.5 Overall architecture

All these monitoring tools are converted into a single configuration and the following figure highlights how they are integrated in overall architecture.

The communication between components is achieved with specific protocols of each tool. Icinga agents from physical hosts are communicating with Icinga server with a secure NRPE protocol. Logs from containers and virtual machines are securely pushed with Lumberjack Protocol. Monitoring metrics are pushed by collectd with a TCP custom based protocol to Graphite.

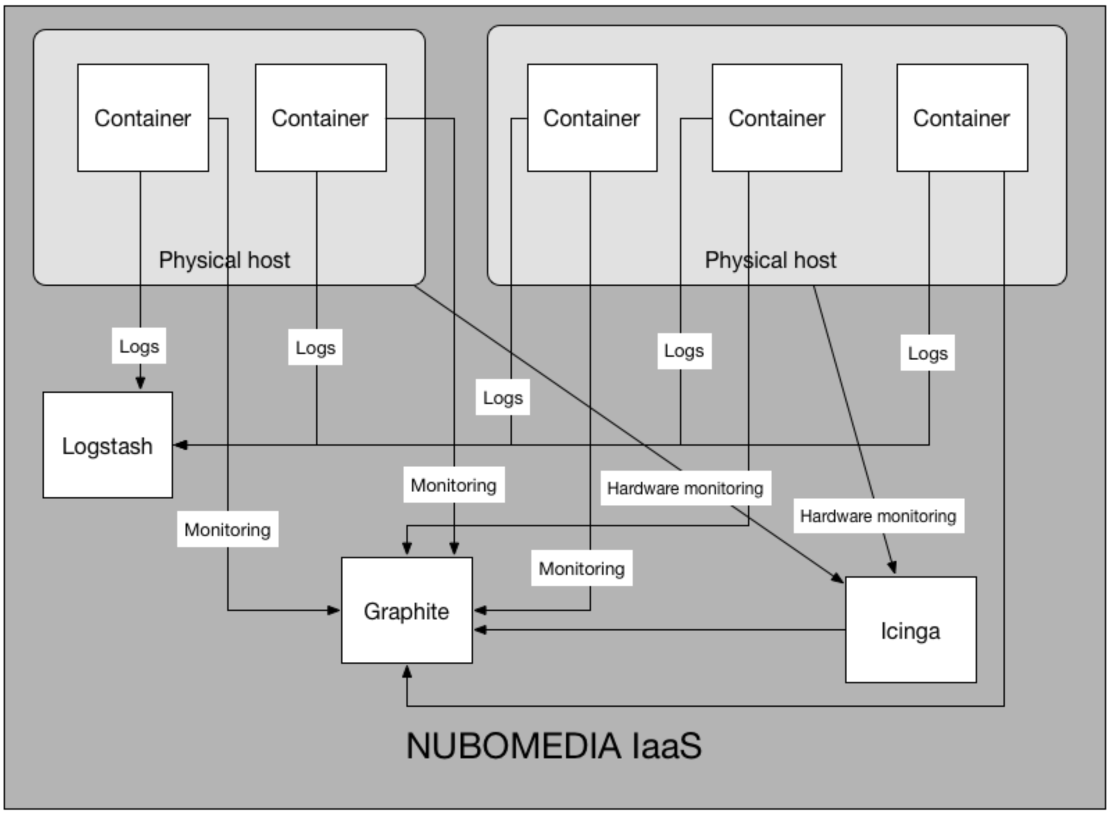


Figure 34. Monitoring integration on the NUBOMEDIA IaaS

3.2.6 Connectivity Manager Agent (CMA)

The Connectivity Manager Agent component was not modified after year 1. Even though new releases of OpenStack slightly modified the way networking is working, the API used by the CMA for interacting with OpenVSwitch were not changed.

3.2.7 Conclusions

Considering our research, we believe that utilizing OpenStack with Docker integration as NUBOMEDIA IaaS is the best approach. In addition to computing, storage and networking capacity, the NUBOMEDIA IaaS also provide centralized metrics, logs collecting and monitoring system for all the architectural components in order to provide the upper layers' information about the real-time situation of the IaaS.

OpenStack controller components are mapped on the Virtualized Infrastructure Manager functional element. The interfaces exposed by the VIM are practically implemented by the ones exposed by the OpenStack services as REST API.

3.3 NUBOMEDIA Media Plane

The NUBOMEDIA Media Plane components developed in the context of the NUBOMEDIA Cloud Platform, are mainly management and orchestration components providing functionalities for controlling the lifecycle of the Media Plane entities on top of the IaaS.

Open Baton, shown in Figure 35, was chosen as a reference implementation of a NFV MANO environment [3]. It was used and extended in order to provide:

- NFV Orchestration of Network Services
- A generic and a specific VNF Management solution
- An Element Management System (EMS)
- Java SDKs for VNF Managers and plugins
- Dashboard

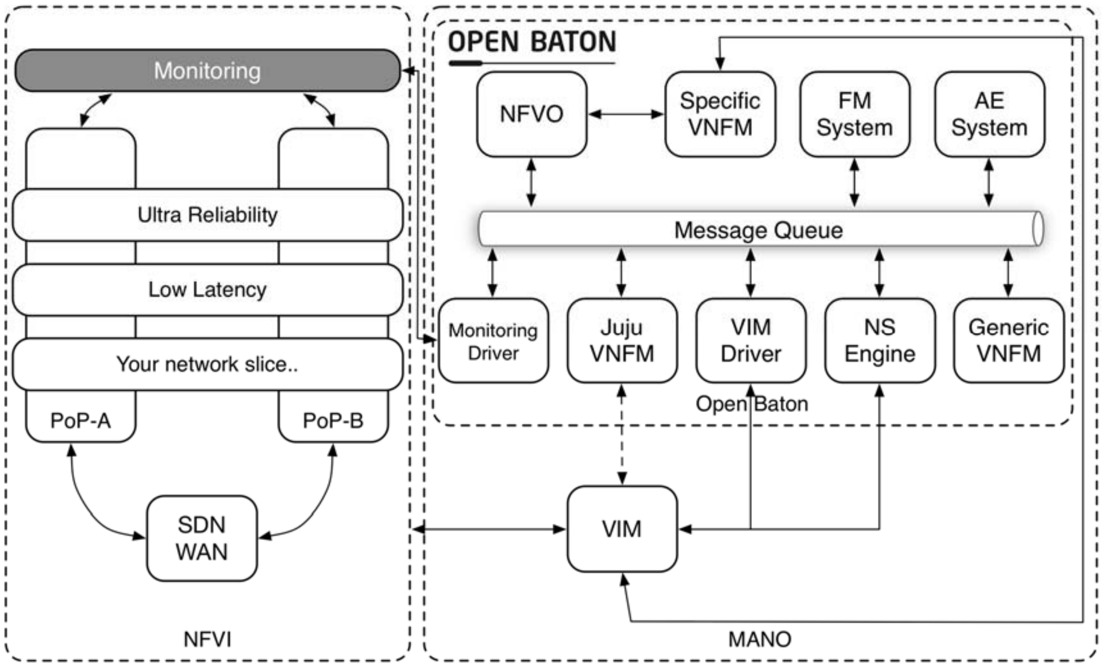


Figure 35. Open Baton release 3 architecture

Internal communication between the different components is realized via Pub-Sub mechanism using RabbitMQ as message bus. The Advanced Message Queuing Protocol (AMQP) JSON-RPC is the Remote Procedure Call mechanism used for calling methods between components.

Additionally, it was necessary to implement a new VNFM specific (the VNFM-EMM) for the management of the Media Server functions. The changes which have been developed in the context of NUBOMEDIA on the NFVO, have been released and made available also to the Open Baton open source community.

3.3.1 Network Function Virtualization Orchestrator (NFVO)

The NFVO manages the lifecycle of a Network Service. In the ETSI terminology a Network Service is a set of multiple Virtual Network Functions. In NUBOMEDIA a Network Service may be composed by one or more Media Server instances and a Cloud repository. It exposes an interface (Pm-Or) to the PaaS Manager providing the functionalities for instantiating and disposing a Network Service. A Network Service is associated with a single application, and multiple Network Services can be executed in parallel on the same IaaS. The main functionalities provided by the NFVO are:

- Allocation of virtual resources (via the Or-Vi-VIM interface) required by the Network Service as specified by the PaaS API using the descriptor (in the Annex it is possible to find an example of the Network Service Descriptor used in NUBOMEDIA).
- Generation of events (finished instantiation of a Network Service Record, execution of scaling procedure for a VNF, etc.) based on a PUB-SUB mechanism via the NFVO-events interface
- Request the execution of the VNFs via the NFVO-VNFM interface.

The NFVO part of the Open Baton is implemented in Java on top of the Spring Framework. It is composed by different modules:

- API: is the component exposing the Pm-Or API interface consumed by the PaaS layer for on boarding Network Service Descriptors (NSD) and requesting the instantiation of Records. Refer to D2.4.2 [2] for more details about the definition of Descriptors and Records. In the Annex, it is possible to find also an example of a Descriptor used by the PaaS Manager to deploy Media Plane entities. Furthermore, at http://get.openbaton.org/api/ApiDoc.pdf it is possible to find a description about the Pm-Or interface definition presented previously.
- Repository: interoperate with the catalogue where the different resources are stored. It exposes an internal interface to other two modules, the API and the CORE. The API are also exposing the catalogue content via the NFVO-API
- VIM: it is a module used by the Core for interoperating with the Point of Presences (PoPs) available. This module provides the capability of interoperating with multiple type of VIMs using a driver mechanism.
- Core: represent the central module of the NFVO. It coordinates all the lifecycle events of the Network Service Records instantiation. It has several fundamental functionalities: from the management of the catalogue to the actual instantiation of a Network Service Record (NSR) starting from a Network Service Descriptor (NSD). It also contains the logic of generating events to external modules via the NFVO-events interface.
- Virtual Network Function Management: provides an internal interface for dispatching messages to the different VNFMs which are registered. It is in charge of dealing with the communication with all the VNF Managers. It provides the REST and AMQP APIs for interacting with the VNFMs. This module contains also the state machine able to manage the Virtual Network Function Records (VNFRs) and decide the state changes.

3.3.1.1 NFVO to VNFM communication

The NFVO communicates with the VNF Managers through REST APIs or using a messaging system. In the context of NUBOMEDIA it was employed the second approach as it allows ease integration of multiple technologies which may be used for implementing the VNFM.

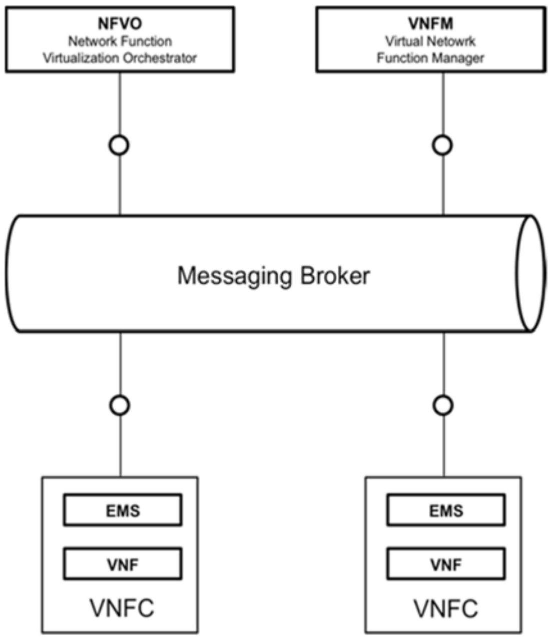


Figure 36. NFVO, VNFM and EMS interactions

The NFVO creates different queues:

- Two queues for event registration and deregistration. This is basically the implementation of the NFVO-events interface. The NFVO provides two queues on top of which external entities (for instance the Connectivity Manager) registers themselves for receiving particular events.
- Two queues for VNFM registration and deregistration. This is the implementation of the NFVO-VNFM interface. Each time a new VNFM is deployed (for a newly available VNFM) it sends a registration message to the NFVO using the message queue. With the registration, the NFVO is aware of the VNFM endpoint (which is also the name of the queue instantiated by the VNFM, see below), and therefore is able to communicate lifecycle events for the particular VNF managed by that VNFM.
- One queue to which all the VNFMs send back the answers of each lifecycle event execution requested by the NFVO.

When the VNFM is deployed, it creates one queue. The name of that queue depends on the type of the VNF that this VNFM deployed is in charge of managing but the purpose of that queue is to receive the action that the NFVO sends to that particular VNFM. The name of the queue contains the type of the VNF in order to allow the NFVO to choose the right VNFM for the right VNF.

The Generic VNFM, used in NUBOMEDIA for managing the lifecycle of the Cloud Repository components, creates on demand multiple queues:

- One queue devoted to the registration of the EMS instances. Each time a virtual compute resources is instantiated, it executes the EMS process (as agent running in the compute resource) which registers to the VNFM as responsible for the specific compute resource.
- Two queues per VNFC Instance. These two queues contain the hostname of the compute resource. In this way, the Generic VNFM is able to send the commands to be executed to the correct compute resource and knows which one has concluded the execution of a specific command.

3.3.1.2 The plugin mechanism

The NFVO provides a plug and play mechanism used for installing or removing runtime plugins. The plugin mechanism is realized using JSON-RPC. Basically, the plugin starts as an independent process and implements some of the methods which could be called by the NFVO to interact with different types of VIM.

This mechanism is used for instance for supporting multiple implementations of the NFVO-VIM interface, called internally VIM Driver. The VIM Driver is the interface allowing the NFVO or a VNFM to communicate with the VIM. Considering that in NUBOMEDIA the VIM is implemented by OpenStack, it was used a plugin that implements the main functionalities for creating networks, VMs or containers, collecting quotas and so on. This plugin uses the apache library JClouds.

3.3.2 Virtual Network Function Managers (VNFMs)

The VNFM, as specified by ETSI NFV, is a component providing lifecycle management of a specific VNF. In NUBOMEDIA there are two types of VNFMs: one for managing the Media Servers, and one, as already mentioned before, for the Cloud Repository. Both are using the PUB/SUB mechanism presented before for communicating with the NFVO.

The main function of the VNFM is to deploy a specific VNF on top of virtual resources allocated on the IaaS. This is achieved triggering the execution of certain scripts (as defined in the lifecycle events of the Network Service) on the virtual resources where those elements have to be installed.

In NUBOMEDIA, a completely new VNFM was implemented, adding to its basic ETSI NFV functionalities also the logic of managing Media sessions and components, basically the functionalities of the Elastic Media Manager (EMM).

3.3.2.1 VNFM-EMM

The EMM manages the acquisition of Media Components of particular applications and therefore, it exposes an interface to the Application layer providing the capabilities of reserving the required resources on a Media Component. In particular, it makes sure that there are always the required resources for the Applications, adapting the utilized virtual resources to the workload changes.

This, so called autoscaling mechanism, provides two main operations, scaling in and scaling out, to horizontally scale a set of Media Components. Scaling out means adding additional resources when they are required.

Basically, all the scaling decisions and Media Components selections is done by the EMM. This bases on a specific algorithm including certain parameters like the operation that is called by the Application layer (basically, register new applications and delete existing applications), the utilization of existing Media components (based on a point mechanism) and the current set of Pools where each Media Component can be assigned to respectively.

3.3.2.1.1 Media Content

A Media Component is the representation of a running Instance where Applications can be established. In detail, each Media Component is represented by a Virtual Network Function Component Instance (VNFCInstance) and provides identical performance in the meaning of computation power, disk space, network capabilities, etc. These Media

Components can handle a specific amount of Applications based on the required points. From the Media Component point of view, these points describe the entire capacity of the Media Component that can be occupied by several applications in parallel without losing any performance assumptions. This means that each Media Component can handle a fixed amount of points that is based on the predefined performance capacity provided by the corresponding VNFCInstance (depends on the chosen DeploymentFlavour defined in the Network Service Descriptor).

From the Application point of view, these points reflect the capacity that is requested by the Application layer for ensuring a proper execution of the Application itself. Depending on the specific Media Component and registered Applications, the Media Component is represented by different states at specific points in time: IDLE, ACTIVE, RELEASE. Status IDLE means, that no Applications are registered at all at that moment. Once a new Application is registered the status goes to ACTIVE and remains until all Applications are unregistered again. If there are no more Applications running on the Media Component goes to status RELEASE.

Furthermore, all Media Components are available via internal and optionally external IPs exposed by the assigned VNFCInstances. So, once a new Application is registered to the EMM, the IP will be used by the Application layer to establish a new Application on the Media Component belonging to this specific IP and the corresponding VNFCInstance.

3.3.2.1.2 Application

An Application provides a specific functionality to the customer. Moreover, an Application consumes a predefined amount of capacity of the Media Plane. An Application inside the EMM is represented by specific parameters listed in the following:

- id: The ID is the Application ID assigned by the EMM for internal identification of Applications.
- vnfr_id: The vnfr_id is the ID of the VNFR where the Application is registered to (chosen by the Application layer)
- points: The points represent the consumed points by the Application. The amount of consumed points comes from the Application layer and influences the decision making of selecting the Media Component where the new Application can be established.
- mediaServerId: The mediaServerId is the ID of the Media Component (internally defined by the VNFCInstance) that is chosen for the establishment of the Application.
- ip: The ip is the IP that belongs to the Media Component that is selected for the establishment of the Application. The IP can either be private or public, depending on the configuration of the Network Service.
- extAppId: (optional) The external Application ID extAppId represents the external ID of the Application and can be defined when registering a new Application. This is used for identifying Applications with an ID that comes from the outside of the EMM.

3.3.2.1.3 EMM interface

The Interface exposed by the EMM to the Application layer, follows the RESTful approach, in the meaning of providing a well-defined interface for acquiring Media components on demand. Basically, it provides methods for registering and unregistering Applications, list details about specific Applications and also about all registered

Applications to a specific Virtual Network Function Record (VNFR). Furthermore, it can be requested a list of all VNFRs managed by the EMM. The following table gives an overview of all operations provided by the Interface. Each operation is described more in detail afterwards.

Application parameters:

Name	**Description**	**Required**	**Schema**
`vnfr_id`	Contains the ID of the VNFR which is managing this application	true	String
`mediaServerId`	Contains the ID of the kurento media server which is used by the application	true	String
`ip`	Contains the IP of the kurento media server which is used by the application	true	String
`points`	Defines the number of points which is required/requested by the application	true	Integer
`extAppId`	An optional field which can contain the external application ID as it is given by the PaaS Manager. Basically, needed for improving application management	false	String
`created`	Contains the date when this application was created at the MS-VNFM side	true	Date
`heartbeat`	Contains the date when the last heartbeat for this application was received	true	Date
`missedHeartbeats`	Counts the missed heartbeats. This depends on the interval defined in the configuration file. If predefined number of heartbeats were not received, the application is removed automatically. Heartbeat mechanism can be deactivated via the configuration file	true	Integer

Media Server parameters:

Name	**Description**	**Required**	**Schema**
`vnfrId`	Contains the ID of the VNFR which contains this kurento media server	true	String

`hostName`	Contains the human readable identifier of this kurento media server	true	String
`ip`	Contains the IP of the kurento media server	true	String
`usedPoints`	Contains the already used capacity (point-based) which is occupied by registered applications	true	Integer
`maxCapacity`	Contains the maximum of capacity that is offered by this media server which can be consumed by applications	true	Integer
`status`	Contains the status of this media server in term of registered applications: ACTIVE, INACTIVE, IDLE, RELEASE	true	Enum

Create a new application
Requests the creation of a new application EMM-side which is managed by a selected Kurento media server. This operation will occupy capacity of the chosen kurento media server.

```
POST /vnfr/{vnfrId}/app
```

Parameters

Type	Name	Description	Required	Schema
Body Parameter	application	Contains the application which is requesting a kurento media server	true	Application

Responses

HTTP Code	Description	Schema
200	success	Application

Consumes
- application/json

Produces
- application/json

Delete an application
Deletes a given application EMM-side. Consumed capacity of a certain kurento media server will be released afterwards.

```
DELETE /vnfr/{vnfrId}/app/{appId}
```

Responses

HTTP Code	Description	Schema
200	success	Application

Consumes

- /

Produces

- /

Get information of all applications which are managed by a given VNFR
Returns information hold by the EMM with regard to requested application registered for a given VNFR.

```
GET /vnfr/{vnfrId}/app/{appId}
```

Responses

HTTP Code	Description	Schema
200	success	Application

Consumes

- /

Produces

- application/json

Get information of all applications which are managed by a given VNFR
Returns all information hold by the EMM with regard to the requested application registered for a given VNFR.

```
GET /vnfr/{vnfrId}/app
```

Responses

HTTP Code	Description	Schema
200	success	Application list

Consumes

- /

Produces

- application/json

Send a heartbeat for a specific application
This method manages the heartbeat for keeping certain applications alive.

```
PUT /vnfr/{vnfrId}/app/{appId}/heartbeat
```

Responses

HTTP Code	Description	Schema
200	success	Application

Consumes

- /

Produces

- /

Get all media servers of a VNFR
This method returns all kurento media servers which a managed by are given VNFR.

```
GET /vnfr/{vnfrId}/media-server
```

Responses

HTTP Code	Description	Schema
200	success	MediaServer list

Consumes

- /

Produces

- application/json

Get the number of media servers of a VNFR
This method returns the number of all kurento media servers which are managed by a given VNFR.

```
GET /vnfr/{vnfrId}/media-server/number
```

Responses

HTTP Code	Description	Schema
200	success	Integer

Consumes

- /

Produces

- Integer

Get the history of the number of media servers of a VNFR
This method returns the history of the number of all kurento media servers which are managed by a given VNFR.

```
GET /vnfr/{vnfrId}/media-server/number/history
```

Parameters

Responses

HTTP Code	Description	Schema
200	success	String

Consumes

- /

Produces

- String

Get the consumed capacity over all media servers of a VNFR
This method returns the average capacity occupied over all kurento media servers which are managed by a given VNFR.

```
GET /vnfr/{vnfrId}/media-server/load
```

Responses

HTTP Code	Description	Schema
200	success	Float

Consumes

- /

Produces

- Float

Get the history of consumed capacity over all media servers of a VNFR
This method returns the history of the average capacity occupied over all kurento media servers which are managed by a given VNFR.

```
GET /vnfr/{vnfrId}/media-server/load/history
```

Responses

HTTP Code	Description	Schema
200	success	String

Consumes

- /

Produces

- String

A swagger definition of this interface is also available at: https://raw.githubusercontent.com/nubomedia/nubomedia-msvnfm/master/swagger.json

3.3.2.1.4 Register a new application

This method registers a new Application to the VNFR with the ID <vnfr_id> and returns the Media Component information (including the IP) where the new media session can be established. The registration of new Applications will occupy a specific amount of points of the chosen Media Component.

The request contains the <vnfr_id> in the URL and the consumed points in the body. Optionally, the external Application ID can be defined in the body to give the EMM more information about the Application to make the registration of new Applications idempotent. This means, if there is already a registered Application (registered to the requested VNFR) with that external Application ID, it will be returned the same response got at the first registration containing the same Media Component without occupying Media Components again. Without defining the external Application ID, it will be occupied Media Components every time this registration request is received.

3.3.2.1.5 Unregister applications

This method unregisters previously registered Application and release the points from the Media Components occupied previously by this Application.

The request contains the ID <vnfr_id> of the VNFR in the URL where the Application with that ID <app_id> is registered to. This request does not send back any response.

3.3.2.1.6 Get details about a specific application

This method returns a detailed description of an already registered Application.
The request contains the ID <vnfr_id> of the VNFR where the Application is registered to. Furthermore, it contains also the ID <app_id> of the Application assigned by the EMM.

The response contains all parameters specific for that Application like the ID, ID of the VNFR, consumed Points, the ID of the Media Component and its IP. Moreover, if defined, the external Application ID as well.

3.3.2.1.7 Get details about all applications registered to a specific VNFR

This method returns a detailed list of all Applications registered to a specific VNFR. The request contains the ID <vnfr_id> of the specific VNFR in the URL. The response contains a detailed list of all Applications registered to the VNFR with the ID <vnfr_id>.

3.3.2.1.8 Get details about all applications managed by the EMM

This method returns a list of all VNFRs and corresponding NSRs managed by the EMM.

3.3.2.1.9 Pools of Media Components for application placement

In general, we can distinguish between pools of the following types: ACTIVE, IDLE, RELEASE. This means, depending on the status of the Media Component, each Media Component will be assigned to one of these pools respectively.

Basically, each pool has its own characteristics and is used in order to provide groups of resources containing Media Components considered when registering or unregistering Applications. Characteristics and constraints of these pools are explained in the following.

3.3.2.1.9.1 IDLE pool of media components

The pool of idle components contains Media Components that are completely idle at this point in time. Completely idle means: there are no Applications established on any Media Component in this pool at all. This pool is used to fetch new Media Components immediately once the demand appears, for instance, when no more Media Component in the ACTIVE pool can satisfy the registration of a new Application. In general, the IDLE pool of Media Components provides already deployed resources to reduce the time significantly of launching new ones. The amount of Media Components of this pool is fixed. So, if any Media Component of this pool is taken for establishing a new Application, it will be removed from this pool but another one will be added at the same time. This is done either by launching a new Media Component or getting a Media Component that might go potentially back to the IDLE state.

3.3.2.1.9.2 ACTIVE pool of media components

The pool of idle components contains Media Components that are completely idle at this point in time. Completely idle means: there are no Applications established on any Media Component in this pool at all. This pool is used to fetch new Media Components immediately once the demand appears, for instance, when no more Media Component in the ACTIVE pool can satisfy the registration of a new Application. In general, the IDLE pool of Media Components provides already deployed resources to reduce the time significantly of launching new ones. The amount of Media Components of this pool is fixed. So, if any Media Component of this pool is taken for establishing a new Application, it will be removed from this pool but another one will be added at the same time. This is done either by launching a new Media Component or getting a Media Component that might go potentially back to the IDLE state.

3.3.2.1.9.3 RELEASE pool of media components

The RELEASE pool of Media Components contains all the resources that are not occupied anymore by any Applications. For this reason, it is frequently checked for either terminating these Media Components or putting them to the IDLE pool of Media Components if the current size is less than the predefined size of the IDLE pool.

3.3.2.1.10 Autoscaling capabilities

The autoscaling capabilities of the Media Server VNFM were further extended in order to support an advanced scaling in mechanism. Usual scale in mechanisms are not considering sessions running on an existing instance, which may cause in some situations some loss of connectivity from the end user perspective. In the context of NUBOMEDIA, scaling in a media server having some running sessions would cause a termination of a multimedia flow with the end users. Therefore, the proposed mechanism considers the number of running sessions in a media server as a metric for the scale in procedure.

In particular, once the VNFM recognizes the need of scaling in any media server (based on a defined threshold), it checks first if scaling in can be performed. Scaling in a media server is only permitted if no sessions are running on the potential media server, which means that, the media server is idle and no capacity of the media server is consumed by any application. If the VNFM finds any media server which is not occupied by any sessions, scaling in is permitted and executed.

3.3.2.1.11 Start and Stop specific Media Server instances

The NFVO was further extended to support two main functionalities as requested by DRT_R1: Start/Stop a VNF Components, meaning Kurento Media Server instances.

As initial step the NFVO APIs were extended to expose those functionalities to the PaaS Manager. In particular, the PaaS Manager requests the NFVO where the NFVO forwards these requests to the Media Server VNFM which is in charge of controlling the VNF Components and its life cycle. After the initial deployment, all the KMS instances are started.

As shown in the Figure 37, in the first step the PaaS Manager requests the NFVO in order to stop a given Media Server. In the second step the NFVO forwards the requests to the Media Server VNFM. When the Media Server VNFM receives the request for stopping a specific instance, it removes it from the list of available media servers (step 3). Once the Media Server VNFM finished this action (step 4), it sends an acknowledgement back to the NFVO. Afterwards, this stopped instance is still running and serving requests from already registered session but they are not available anymore for new sessions. Nevertheless, already registered sessions, can be removed again by executing the common action for unregistering session. Since the request for stopping requests is asynchrony, the NFVO does not send any acknowledgements back to the PaaS Manager.

If an instance has been stopped, it can be started again by calling the start method on a specific instance (step 5). The NFVO requests again the Media Server VNFM in order to start the specific VNF Component (step 6). Once the Media Server VNFM received the request, it starts the instance and puts the given media server again into the list of available media servers (step 7). Finally, the manager sends an acknowledgement back to the NFVO. After this action, the media server is available again and can be used for assigning new sessions in order to consume capacity. Similarly, to the process of stopping a media server, this method is asynchrony, so the NFVO does not send back any message to the PaaS Manager.

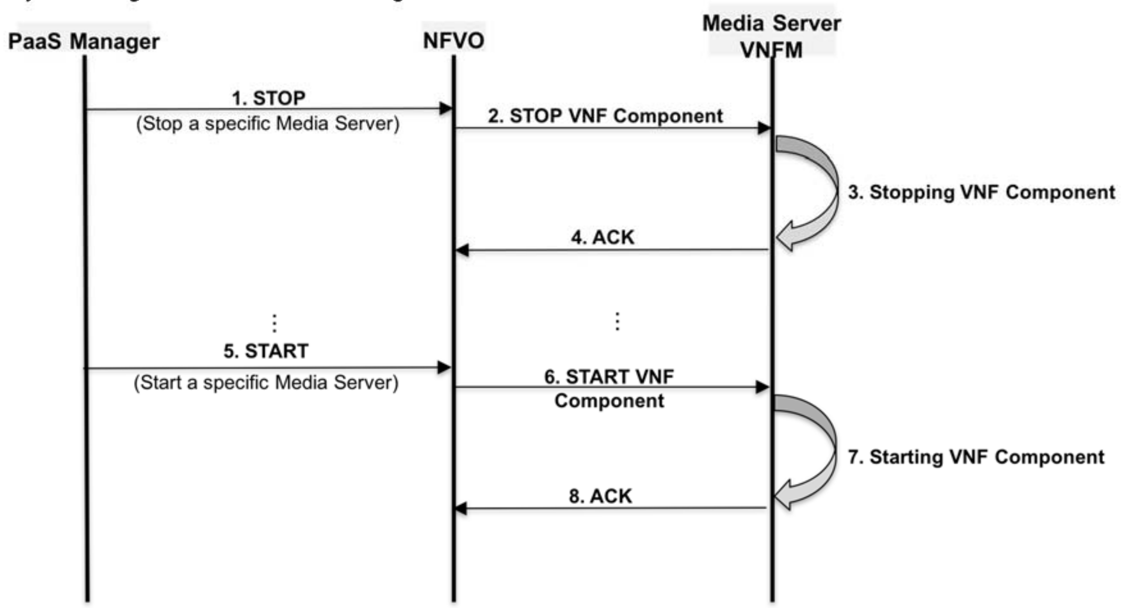

PaaS Manager
NFVO
Media Server VNFM
1. STOP
(Stop a specific Media Server)
2. STOP VNF Component
3. Stopping VNF Component
4. ACK
5. START
(Start a specific Media Server)
6. START VNF Component
7. Starting VNF Component
8. ACK

Figure 37. Sequence diagram showing the start and stop functionality

3.3.2.2 Cloud Repository role in the NUBOMEDIA infrastructure.

The Cloud Repository has not been extended in the last releases as its main functionalities were enough to support requirements from application developers. The Cloud Repository scripts and descriptors used for being deployed via Open Baton have been updated in order to work with release 3.0.0.

3.3.2.2.1 Cloud Repository

Based on the decisions previously taken in the deliverable D3.2.1 [13], MongoDB was chosen as Cloud Repository implementation. As written in the deliverable document, MongoDB fits perfectly our requirements, also because of the GridFS technology for storing files bigger than 16 MB.

MongoDB is also very suitable in a cloud environment because of its shared architecture (Figure 38).

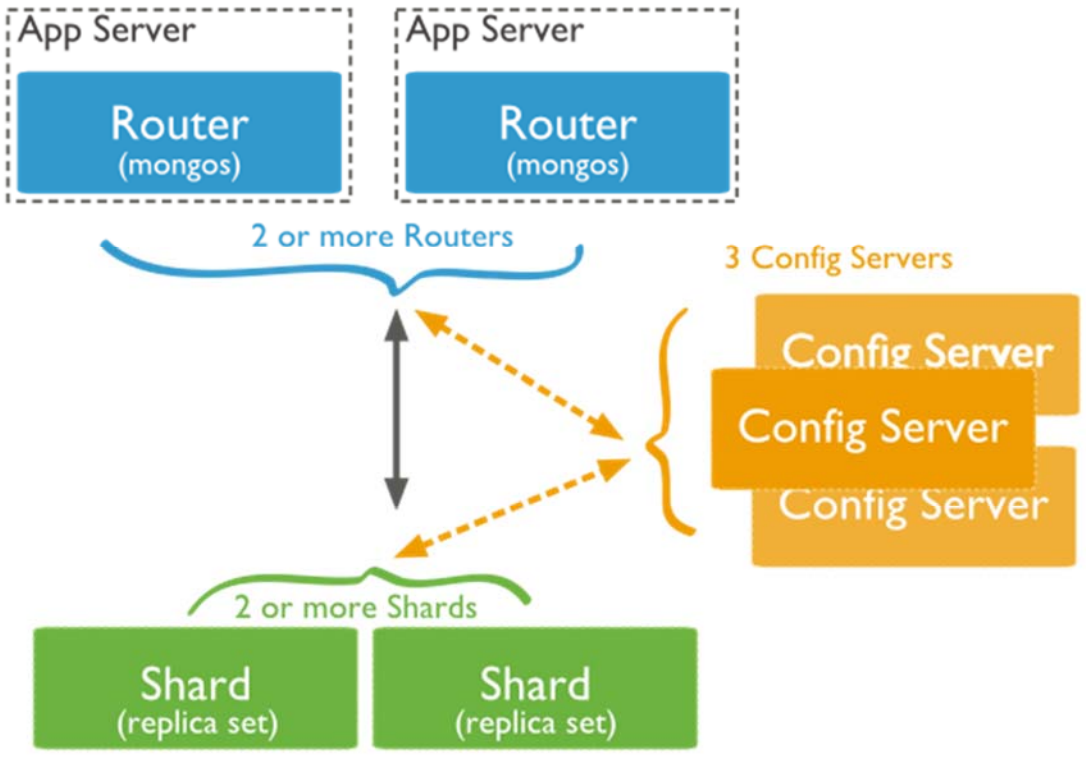


Figure 38. MongoDB sharded architecture

In a cloud production environment with this MongoDB configuration, it is possible to handle a considerable number of requests in parallel. With the features provided by the flexibility of a NFV environment, this architecture could easily adapt itself in order to satisfy a possible peak of requests. It is anyway possible to have a single machine architecture where all these components are collapsed in only one.

3.3.2.2.1.1 Integration into the NFVO

In order to integrate the Cloud Repository into the NUBOMEDIA infrastructure, we decided to create two types of NSs, and therefore different VNFDs, for supporting the version with or without clustering of MongoDB instances.

The first one represents the full architecture with three VNFs: a config server, a router and a shard. Both config and shard VNFs need a dependency to the router VNF in order to be able to provide their IPs to it and to be properly configured. Both config and shard has also a configuration parameter called "smallfiles" that allows MongoDB process to be executed even if there is less the 5 GB of root disk free. This option was really useful during the development process.

The other NSD describes the MongoDB single instance architecture. As before, there is the opportunity to execute the MongoDB instance in "smallfiles" configuration but in this case we have also included the USERNAME and PASSWORD configuration parameters because in that deployment descriptor the authentication is enabled so username and password are needed to access the Mongo instance.

In order to be able to deploy these NSDs using OpenBaton, another VNFM is needed as we can see from the endpoint field: The Generic VNFM.

3.3.2.2.1.2 Generic VNFM

The Open Baton software suite comes with a standalone VNFM called Generic that is implemented in such a way in order to be able to handle every network function able to follow some conventions.

The Generic VNFM is a simple component that executes some scripts into the VM defined into the lifecycle_event field. For doing that, the Element Management System (EMS) is required to be running in the VM and the NFVO is in charge to define the userdata script that will download and install the EMS during the cloud-init [14]. The EMS connects to the RabbitMQ broker where the NFVO and VNFMs are connected to, in this way it is able to register and communicate with the Generic VNFM.

The scripts are uploaded into the VM at the beginning of the INSTATIATE action by the Generic VNFM and usually they require some parameters. Inside the script, it is possible to use the configuration parameters defined in the VNFD just as environment variables and the name is the name defined in the JSON file. If some dependencies are defined in the NSD, then a particular action (the MODIFY) is triggered on the Generic VNFM. The dependency is defined as source and target where the source provides some parameters, either runtime or more static, to the target VNFR. So, during the MODIFY action, the source parameters are passed to the target VNFR and injected into the scripts as environment variables. For use these parameters the VNF script developer needs to be aware of the fact that the foreign parameters are defined as <type of the foreign VNFR>_<name of the parameter>.

In the Cloud Repository case, we uploaded the scripts into a Github repository, the Generic VNFM sends the link to the EMS and they will be directly downloaded from the EMS in the VM. The following example shows the router script called shard_configure.sh.

```
#!/bin/bash

export LC_ALL=C

mongo --port 27017 --eval
        "sh.addShard( '`echo $shard_internal_nubomedia`:27017' )"

mongo --port 27017 --eval  'sh.enableSharding("nubomedia")'
```

As written above, there is a variable called $shard_internal_nubomedia. That variable points to the shard IP on the internal_nubomedia Virtual Link (VL).

3.3.2.2.1.3 Cloud Repository role in the NUBOMEDIA infrastructure.

In case an application needs a media repository system, it is possible to include the MongoDB VNFRs (single or shard) including them while deploying an application. The PaaS-Manager will automatically inject the cloud repository IPs to the Application after its instantiation. In this way, it is possible to connect to MongoDB instances through the Kurento Client APIs.

3.3.3 Connectivity Manager (CM)

The Connectivity Manager (CM), already introduced in previous sections and in D3.3.1 [10], provides mechanisms for controlling QoS parameters, like bandwidth, between Media Components running on the IaaS. Basically, the CM provides capabilities for enforcing specific level of QoS between VNF Components interoperating with the CMA.

The implementation of the NSE component is inspired by the implementation of the Open Baton Framework. The Connectivity Manager is implemented in java using the Spring Framework, and it exposes two interfaces:

- the northbound interface to the NFVO for receiving events about instantiation and removal of NSRs. This interface is implemented by a set of message bus producer and consumer. Those libraries are available on the Open Baton project for being used also by other third party components.
- the southbound interface with Connectivity Manager Agent for requesting particular bandwidth requirements between virtual compute resources on the same virtual link

The first thing it does is to subscribe to the NFVO on the event NSR instantiation finished. Every time a NSR is instantiated, the CM receives it and parses it for analyzing the QoS requirements as described in the record. The event subscription towards Open Baton was made through the Open Baton Client SDK, which uses REST to manage event subscriptions. This has to be done in order to be aware of newly instantiated and terminated network services to manage the QoS on the virtual infrastructure. The usage of the provided SDK allows the Network Slicer Engine to seamlessly communicate with the Open Baton NFVO using a messaging broker system. This messaging broker system is based on Advanced Message Queueing Protocol (AMQP), which uses RabbitMQ to implement this protocol. The interaction with RabbitMQ is implemented using the Spring native library for RabbitMQ which exposes all objects to declare all requested objects by the protocol.

The NSR is sent by the NFVO to the CM as part of the payload of the INSTANTIATE_FINISH or RELEASE_RESOURCE_FINISH events. The NSR management was delegated to a Spring bean, which instantiates a different thread for each record received in order to parse it and collect all information for QoS enforcing onto the SDN Controller - Connectivity Manager Agent in this implementation.

The payload, represented in JSON, is de-serialized using the Google GSON library (which is configured as the default JSON serializer/de-serializer) and “parsed” for extracting information about the Virtual Link Record (containing QoS parameters). If there are QoS requirements the CM reads the data of VNFC Instance which requires QoS requirements and aggregates them for starting the allocation of queues and flows.

Internally, there are two beans implemented to dispatch the QoS requirements and Flow definitions. Once the technical requirements are extracted, it uses the provided Spring RestTemplate to perform REST requests towards the CMA. All the REST requests that are exposed by the CMA are mapped to this bean.

[10] https://github.com/nubomedia/additional-documentation/raw/master/WP3/D3.3.1_CrossLayerConnectivityManager_V1.0_27-01-2015_FINAL-PC.pdf

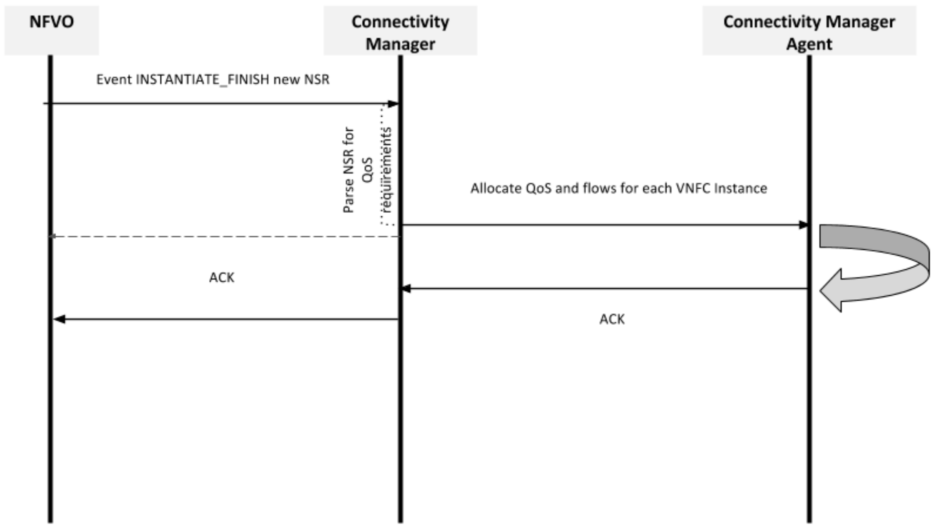

NFVO
Connectivity Manager
Connectivity Manager Agent
Event INSTANTIATE_FINISH new NSR
Parse NSR for QoS requirements
Allocate QoS and flows for each VNFC Instance
ACK
ACK

Figure 39. Interactions between the NFVO, CM and CMA.

The “aggregated” data is sent to the CMA via the southbound interface. This interface, implemented as REST API, hasn’t been changed from the one proposed during year 1 of the project [7].

The Connectivity Manager Agent is implemented in Python and utilizes the Bottle Python web framework to expose northbound REST APIs in order to enable an independent programming language interaction with any tool that enables enforcement of QoS related configuration capabilities. Those APIs are documented below:

Server parameters:

Name	**Description**	**Required**	**Schema**
`id`	ID of the Server	true	String
`name`	Name of the Server	true	String
`interfaces`	List of Interfaces	true	InterfaceQoS list

InterfaceQoS parameters:

Name	**Description**	**Required**	**Schema**
`ip`	IP connected to the interface	true	String
`ovs_port_number`	Port number of the open vSwitch	true	String
`qos`	QoS configurations	true	QoS

QoS parameters:

Name	Description	Required	Schema
qos_uuid	ID of QoS configuration	true	String
queues	Queues for QoS	true	QosQueue

QosQueue parameters:

Name	Description	Required	Schema
queue_uuid	ID of queue	true	String
rates	Min and max rates for the queue	true	QosQueueValues

Flow parameters:

Name	Description	Required	Schema
ovs_port_number	Port number of the open vSwitch	true	String
src_ipv4	IPv4 source address	true	String
dest_ipv4	IPv4 destination address	true	String
dest_hyp	Hypervisor in charge for setting up the flow	true	String
protocol	Defines the protocol for used for the flow: UDP, TCP	true	String
priority	Contains the priority of the flow	true	String
queue_number	Defines the queue number	true	String

FlowServer parameters:

Name	Description	Required	Schema
hypervisor_id	ID of the Hypervisor	true	String
server_id	ID of the Server	true	String
qos_flows	List of Flows	true	Flow list

AddQueue parameters:

Name	Description	Required	Schema
`hypervisor_id`	Id of the Hypervisor	true	String
`values`	Contains QoS configuration related to the hypervisor	true	QoS list

QosQueueValues parameters:

Name	Description	Required	Schema
`min_bitrate`	Defines the min bitrate	true	String
`max_bitrate`	Defines the max bitrate	true	String

Host parameters:

Name	Description	Required	Schema
`hypervisors`	Contains the list of hypervisors	true	Datacenter list

Datacenter parameters:

Name	Description	Required	Schema
`name`	Defines the min bitrate	true	String
`max_bitrate`	Defines the max bitrate	true	String

Get all hosts
Returns the list of hosts under control.

```
GET /hosts
```

Responses

HTTP Code	Description	Schema
200	success	Host list

Consumes

- /

Produces

- application/json

Get information of server of hypervisor
Returns information of a given server on a given hypervisor.

```
GET /server/{hypervisor_name}/{server_name}
```

Responses

HTTP Code	Description	Schema
200	success	Server

Consumes

- /

Produces

- application/json

Configure QoS
Configures the given QoS to the given interfaces.

```
POST /qoses
```

Parameters

Type	Name	Description	Required	Schema
Body Parameter	body	Contains the Server configuration (Interfaces, QoS) to be applied	true	Server

Responses

HTTP Code	Description	Schema
200	success	Server

Consumes

· application/json

Produces

· application/json

Delete QoS configuration
This method destroys all QoS configurations on the given hypervisor.

```
DELETE /qoses/{hypervisor_hostname}/{qos_id}
```

Responses

HTTP Code	Description	Schema
200	success	Server

Consumes

- /

Produces

- application/json

Create Queue
Create a new queue

```
POST /queue
```

Parameters

Type	Name	Description	Required	Schema
Body Parameter	body	Contains the QoS configuration for a specific hypervisor	true	AddQueue

Responses

HTTP Code	Description	Schema
200	success	QoS

Consumes

- application/json

Produces

- application/json

Delete Queue
Deletes previously applied QoS configuration for a specific hypervisor and queue

```
DELETE /queue/{hypervisor_hostname}/{queue_id}/{queue_number}/{qos_id}
```

Responses

HTTP Code	Description	Schema
200	success	Server

Consumes

- application/json

Produces

- application/json

Create Flow
This method assigns a flow to a given queue.

```
POST /flow
```

Parameters

Type	Name	Description	Required	Schema
Body Parameter	body	Contains the information about Hypervisor, Server, QoS and Flows to be applied	true	FlowServer

Responses

HTTP Code	Description	Schema
200	success	FlowServer

Consumes
- application/json

Produces
- application/json

Delete Flow
Deletes all flows associated with a given IP and protocol on the given hypervisor.

```
DELETE /flow/{hypervisor_hostname}/{flow_protocol}/{flow_ip}
```

Responses

HTTP Code	Description	Schema
200	success	String

Consumes
- /

Produces
- application/json

The Connectivity Manager Agent's Core module receives and parses the requests for applying QoS enforcement. In addition, it checks if all the requested resources are available by using the southbound layer to retrieve all the information. In case of feasibility, the same interface is used to configure the QoS parameters on the correct instance(s) among the controllable virtual switch instances.

The southbound layer uses the OpenStack python clients to retrieve the data of all VMs that compose the network service and their network topology (such as mapped port to the virtual switch, and port number in the virtual switch).

3.4 NUBOMEDIA PaaS

In this section will be explained how the PaaS layer software components have been implemented. In particular, the PaaS layer is composed by two main components:

- The PaaS Manager: implemented from scratch in Java using the Spring framework
- The PaaS System: implemented by the open source OpenShift framework

3.4.1 NUBOMEDIA PaaS Manager

In this section is explained how the PaaS layer software components have been implemented using the Spring framework. As already mentioned in Section 2.3.1, the PaaS Manager simplify the way developers are instantiating their applications providing a high-level abstraction of the information required by the lower levels. The main functionalities provided by the PaaS Manager are:

1. Building and deployment of Applications on NUBOMEDIA PaaS
2. Requesting the instantiation of Media Components interacting with the NUBOMEDIA Media Plane management components

The PaaS Manager is a Spring Boot application that uses Spring framework functionalities to expose REST APIs, perform REST requests to the NUBOMEDIA PaaS and store developers' application useful data. It also uses NFVO SDK to interact with Open Baton for requesting the instantiation of Media Components. Therefore, it makes use of the RabbitMQ messaging system for the JSON-RPC calls towards the NFVO. The PaaS Manager is composed by five main components:

- API: the external REST API where developers could send requests to perform authentication, application creation/deletion, etc.
- PaaS Manager: a component that interact with connectors to request media server allocation and application building and deployment
- PaaS Connector: is used to perform request to the PaaS through its REST API
- NFVO Connector: request to the NFVO for allocating Media Components on the NUBOMEDIA IaaS/Media Plane
- Repository: this component stores the Application data that will be used for deletion, query status and UI

3.4.1.1 Application lifecycle

Figure 40 shows the entire life cycle of an application from the initial request up to the moment where the application is running. In step 1 the application developer triggers the deployment of an application by providing the application definition based on a JSON configuration file. What such an application request contains, is already described above. Once the application is persisted in the database, the application status goes to CREATED. Once the application is persisted in the database, required resources will be requested (step 2a) by requesting the NFVO. If the request was properly sent, the application status goes into INITIALISING. If not, the status will end up in FAILED (step 2b). In step 3a required resources will be allocated before the status goes to INITIALISED. If something went wrong during the allocation (e.g. no resources left), the status goes to FAILED. The INITIALISED status indicates that the virtual instances containing the kurento mediaserver are present and configured. Once the PaaS Manager received the notification from the NFVO, it starts to build the application by requesting OpenShift (step 4). If the building fails, it goes directly to FAILED (step 5b). If the building of the application finishes without any problems (step 5a), the deployment is triggered in step 6a. If problems during the deployment will occur, the application itself goes to FAILED. Finally, if the deployment finished without any exceptions, the application goes to the status RUNNING. Once it goes to RUNNING, the application itself will start. From this status, it might happen that exceptions may occur at runtime. In this case, it goes to FAILED (step 7) but here it is possible to recover the application and bring it back to status RUNNING (step 8).

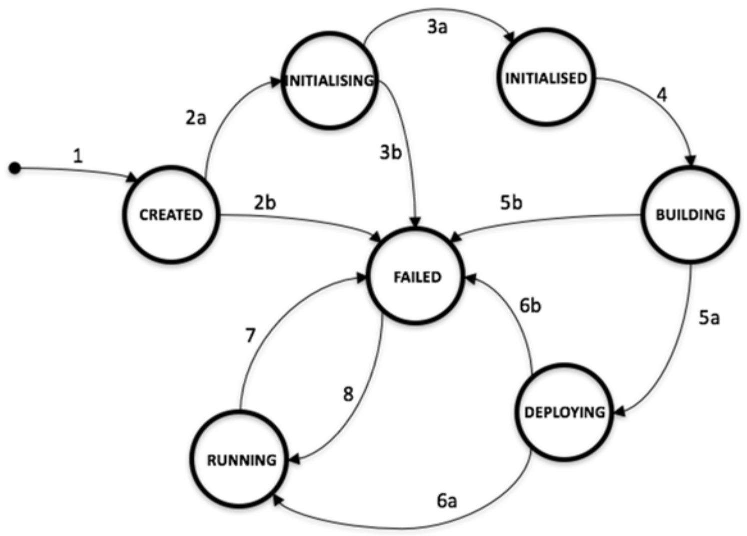


Figure 40. State diagram of an application

3.4.1.2 PaaS-API

The API component uses the RestController to map request paths, request's body and/or path parameters. The root path for all the requests is "/api/v2/nubomedia/paas". It uses GSON as default de-serializer of the requests body.

Every request has to be authenticated with a "Auth-Token" in the request headers. The PaaS API are RESTful APIs accessible via HTTP(s) on the PaaS Manager server. These REST APIs expose abstractions over specific operations to developers allowing them to deploy and manage their end-user applications on the NUBOMEDIA PaaS. gives an overview of the exposed REST APIs.

Before going into the details of all methods provided by the PaaS-APIs it is given a brief description about the Application object, which is the information model describing an Application, which is used by the PaaS Manager to deploy it. The JSON object like the one shown below:

```
{
  "gitURL":"https://github.com/example/example-app.git",
  "name":"my-app-name",
  "cloudRepository":boolean,
  "cloudRepoPort":"cloud-repository-desired-port",
  "cdnConnector":boolean,
  "flavor":"SMALL/MEDIUM/LARGE",
  "ports": [
    {
      "port": 8443 (for example),
      "targetPort": 8443 (for example),
      "protocol": "TCP" (for example)
    }
  ],
  "replicasNumber":2,
  "secretName(optional)":"defined-secret-name",
  "numberOfInstances":3,
  "stunServerActivate":boolean,
  "stunServerIp(mandatory if enbaled)":"stun-server-ip",
  "stunServerPort(mandatory if enbaled)":"stun-server-port",
  "turnServerActivate":boolean,
  "turnServerUrl(mandatory if enbaled)":"turn-server-url",
  "turnServerUsername(mandatory if enbaled)":"username",
  "turnServerPassword(mandatory if enbaled)":"password",
  "scaleOutLimit":maximum-number-of-instances,
  "scale_out_threshold":average-capacity-consumed-scaling-out,
  "scale_in_threshold":average-capacity-consumed-scaling-in
  "qualityOfService(optional)":"BRONZE/SILVER/GOLD",
  "services": [
    {
      "name":"name-of-supporting-service",
      "dockerURL":"url-of-docker-image",
      "replicasNumber":1,
      "ports": [
        {
          "port":3306,
          "targetPort":3306,
          "protocol":"TCP"
        }
      ],
      "envVars": [
        {
          "name": "MYSQL_DATABASE",
          "value": "nubo"
        }
      ]
    }
  ]
}
```

Where the parameters have the following description:

- gitURL: git repository where the jar and Dockerfile (and even other files that are necessary for the application) are commited (if the repository is public the link has to be the https version, if is private has to be the ssh version)
- appName: the application name that will be used also to create the DNS entry to use your application
- cloudRepository: boolean value to require the Cloud Repository;
- qualityOfService: optional value to require the QoS for intra-mediaserver communication
- flavor: enumerative to set the flavor of the mediaserver
- ports: an object that maps the ports that are used from container to the ports that has to be exposed to outside, with relative protocol
- replicasNumber: the number of containers that has to be created by the PaaS after the building phase
- secretName (optional): the name of the secret that has to be used only if your application is on a private git repository
- turnServerActivate: boolean value to enable the turn server on the mediaserver, if turnServerIp, turnServerUsername and turnServerPassword are not specified the mediaserver will use the default one;
- turnServerUrl: the url of the turn server if different from the default one (example: turn:192.168.43.12:8080)
- turnServerUsername: the username of the turn server (mandatory in case is specified the turnServerIp)

- turnServerPassword: the password of the turn server (mandatory in case is specified the turnServerIp)
- stunServerActivate: boolean value to enable the stun server on the mediaserver, if stunServerAddress and stunServerPort are not specified the mediaserver will use the default one;
- stunServerIp: the stun server ip, if is settled also the stunServerPort has to be settled;
- stunServerPort: the stun server port (as string), is mandatory if the stunServerAddress is settled
- scaleOutLimit: the maximum number of mediaserver instances for scaling
- scale_in_threshold: the maximum capacity of the media server to scale in
- scale_out_threshold: the minimum capacity of the media server to scale out
- services: supporting services for providing additional services used by the main application, e.g. databases and others services.
    - name: human readable name of the supporting service
    - dockerURL: URL where the docker images is available, e.g. docker hub
    - replicasNumber: is a numeric value describing the number of containers
    - ports: indicates the transport protocol and ports exposed for the supporting service
    - envVars: list which contains key/value pairs that defines the environment variables used by the supporting service for configuration.

Table 4 provides a summary of the APIs exposed by the PaaS Manager.

Method	**URL**	**Description**
POST	/api/v1/nubomedia/paas//oauth/authorize	Authenticate a PaaS User
POST	api/v1/nubomedia/paas/users	Create a new PaaS User
DELETE	/api/v1/nubomedia/paas/users/{name}	Deletes a PaaS User
POST	/api/v1/nubomedia/paas/app	Create (build and deploy) a new application
POST	/api/v1/nubomedia/paas/app/{id}	Retrieve status of a application with the given id.
GET	/api/v1/nubomedia/paas/app	Return the list of all deployed applications
DELETE	/api/v1/nubomedia/paas/app/{id}	Delete application with the given id
POST	/api/v1/nubomedia/paas/secret	Create a secret
DELETE	/api/v1/nubomedia/paas/{name}	Deletes a secret

Table 4. Summary of the REST API exposed by NUBOMEDIA at the iD interface. This interface is also called the PaaS Manager API along this document and makes possible for application developers to deploy their NUBOMEDIA-enabled applications into the NUBOMEDIA PaaS.

Each individual API is further described here, while the swagger[11] definition of this interface is available at the following URL: https://raw.githubusercontent.com/nubomedia/nubomedia-paas/master/swagger.json

[11] http://swagger.io/

Authenticate a PaaS User
In order to interact with the PaaS, the user has to authenticate towards the PaaS with username and password.

```
POST /oauth/token
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	username	Username of the user that wants to authenticate	true	String
Header Parameter	password	Password of the user that wants to authenticate	true	String
Header Parameter	grant_type	Type of granting. In this case password	true	String

Responses

HTTP Code	Description	Schema
200	success	OAuth2AccessToken

Consumes
- application/x-www-form-urlencoded

Produces
- application/json

Create a new PaaS user
This method creates a new PaaS user with the given roles for defined projects. This method can be issued by a user with administrator rights only.

```
POST /api/v1/users
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String
Body Parameter	body	The body contains the information about the user that is going to be created	true	User

Responses

HTTP Code	Description	Schema
200	success	User

Consumes

- application/json

Produces

- application/json

Get information of a user

This method return information of a given user.

```
GET /api/v1/users/{id}
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String
Body Parameter	body	The body contains the information about the user that is going to be created	true	User

User parameters:

Name	Description	Required	Schema
`username`	Name of the user	true	String
`password`	Password of the user	true	String
`enabled`	Indicates if the user is enabled or not	true	Boolean
`email`	Email address of the user	false	String
`roles`	Contains the roles of the user. It is a map between the project and the role.	false	Role list

Role parameters:

Name	Description	Required	Schema
`project`	Name of the project	true	String
`role`	Role according to the project that is given to a user (GUEST, USER, ADMIN)	true	Enum

Responses

HTTP Code	Description	Schema
200	success	User

Consumes

- /

Produces

- application/json

Get information of all users
This method returns information of all existing users.

GET /api/v1/users

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String
Body Parameter	body	The body contains the information about the user that is going to be created	true	User

Responses

HTTP Code	Description	Schema
200	success	User list

Consumes
- /

Produces
- application/json

Delete a PaaS user
This method deletes the given PaaS user based on his name. This method can be issued by a user with administrator rights only. The default admin user cannot be deleted.

DELETE /api/v1/users/{name}

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
204	success	

Consumes
- /

Produces
- /

Create a new PaaS project
This method creates a new project inside the PaaS Manager. New projects can be created by administrators only.

```
POST /api/v1/projects
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String
Body Parameter	body	The body contains the information about the project that is going to be created	true	Project

Responses

HTTP Code	Description	Schema
200	success	Project

Consumes
- application/json

Produces
- application/json

Get information of a PaaS project
This method return information of a given project inside the PaaS Manager.

POST /api/v1/projects/{id}

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
200	success	Project

Consumes
- /

Produces
- application/json

Get information of all PaaS projects
This method returns information of all projects inside the PaaS Manager. Can be requested only by administrators.

POST /api/v1/projects

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Project parameters:

Name	Description	Required	Schema
name	Name of the project	true	String
description	Human readable description of the project	true	String

Responses

HTTP Code	Description	Schema
200	success	Project list

Consumes

- /

Produces

- application/json

Delete a PaaS project
This method deletes the given PaaS user based on his name. This method can be issued by a user with administrator rights only. The default admin user cannot be deleted.

`DELETE /api/v1/projects/{name}`

Parameters

Type	Name	Description	Required	Schema
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
204	success	

Consumes

- /

Produces

- /

Create a new application

This method creates and deploys a new application. This method can be issued by any user within the selected project. The user must have the priviliges to deploy applications in the selected project. The request body contains the definition of the application to be deployed.

```
POST /api/v2/nubomedia/paas/app
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	true	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String
Body Parameter	body	The body contains the definition of the Application itself.	true	Application

Application parameters:

Name	Description	Required	Schema
`gitURL`	the Git repository URL that contains your project (source or jar), Dockerfile and other files that are necessary to run your application. If the repository is public the link has to be the HTTPS version, if is private has to be the SSH version	true	String
`name`	It is the name of your application as you would want it to appear on the PaaS. This name is used for creating the DNS entry for your application	true	String
`cloudRepository`	Boolean value of `true` or `false` indicating if your application will be needing the cloud repository (the cloud repository is a running instance of the Kurento repository server application)	false	Boolean
`cdnConnector`	Boolean value of `true` or `false` indicating if your application will be needing the CDN Connector (the CDN Connector is a running instance of	false	Boolean

	the NUBOMEDIA CDN Connector)		
`flavor`	This defines the size of the KMS instances. With MEDIUM flavor, you get 2 VCPU and with LARGE flavor you have 4VCPU. The capacity is defined as 100 points for VCPU.	true	String
`ports`	Indicate the transport protocol and ports exposed for the application. The port is the port exposed by the container on which the application will be running, and the target port is the external port on which the application is reachable for the outside. So, there is a mapping coming on within the PaaS for the port and target port. In principle, it can be left the port and target port the same, unless your application has special requirements.	false	Port list
`replicasNumber`	It is a numeric value indicating the number of containers to be created for your application by the PaaS. This value is used for load balancing.	true	Integer
`numberOfInstances`	It is a numeric value that defines the number of KMS instances launched at the very beginning	true	Integer
`turnServerActivate`	Boolean value of `true` or `false` indicating if you want a TURN server to be set on the path of your application	false	Boolean
`stunServerActivate`	Boolean value of `true` or `false` indicating if you want a STUN server to be set on the path of your application	false	Boolean
`stunServerIp`	The IP address of the STUN server you wish to use	false	String
`stunServerPort`	The port of the SUN server you wish to use	false	String
`turnServerUrl`	The IP address and port of the TURN server you wish to use	false	String
`turnServerUsername`	The username to be used as credentials to access the TURN server	false	String
`turnServerPassword`	The password to be used as credential to access the TURN server	false	String
`scaleOutImit`	Defines the MAX number of media servers (KMS) instances that will be instantiates at runtime, e.g. by the auto scaling system	false	Integer
`scaleOutThreshold`	This is the threshold (in terms of	false	Integer

	average number of points) which will be used for the policy of the autoscaling system. Check which flavor you are going to use before defining this threshold		
`qualityOfService`	If enabled it provides dedicated bandwidth levels between media server instances (optional). Possible values are: GOLD, SILVER, BRONZE	false	String
`services`	Supporting services for providing additional services used by the main application, e.g. databases and other services provided by docker hub. Information are provided back to the main application based on the name of the application and parameters of this supporting service. The URL to the service (together with other parameters) are available in the main applications composed by the name of the service and the keys of the environment variables. In case of the URL, for instance, `<NAME>_HOST`	false	Service list

Port parameters:

Name	Description	Required	Schema
`port`	The port is the port exposed by the container on which the application will be running	true	Integer
`targetPort`	the target port is the external port on which the application is reachable for the outside	true	Integer
`protocol`	The protocol definport is the port exposed by the container on which the application will be running	true	String

Supporting Service parameters:

Name	Description	Required	Schema
`name`	Human readable name of the service. Used for composing the name of the environment variables provided back to the main application. See also envVars.	true	String
`dockerURL`	URL where the docker image is available, e.g. docker hub	true	String
`replicasNumber`	Is a numeric value describing the number of containers deployed by Openshift	true	Integer
`ports`	Indicate the transport protocol and ports exposed for the supporting service	false	Port list
`envVars`	List of key and value pairs that defines the environment variables passed to the supporting service. This environment variables are provided back to the main	false	EnvVar

	application combined with the name so that the main application can make use of it, for instance, to make use of the serivce		

Environment Variable (EnvVar) parameters:

Name	Description	Required	Schema
`name`	Name of the environment variable injected while service creation	true	String
`value`	Value of the environment variable injected while service creation	true	String

Responses

HTTP Code	Description	Schema
`200`	success	Application

Consumes

- application/json

Produces

- application/json

Request information of an existing application
This method returns details of a given application ID.

```
GET /api/v2/nubomedia/paas/app/{id}
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	true	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
200	success	Application

Consumes

- /

Produces

- application/json

Request information of all applications deployed
This method returns details of all applications which are currently deployed in the given project.

```
GET /api/v2/nubomedia/paas/app
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	true	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
200	success	Application list

Consumes

- /

Produces

- application/json

Delete an application
This method deletes a given application.

```
DELETE /api/v2/nubomedia/paas/app/{id}
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	true	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
202	success	Application

Consumes

- /

Produces

- application/json

Get build logs of an application
This method returns the build logs of a given application.

```
GET /api/v2/nubomedia/paas/app/{id}/buildlogs
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	true	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
200	success	String

Consumes
- /

Produces
- application/json

Get application logs of an application
This method returns the application logs of a given application for a given pod.

```
GET /api/v2/nubomedia/paas/app/{id}/logs/{podName}
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	true	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
200	success	String

Consumes
- /

Produces
- application/json

Create a new secret
This method creates a new secret inside a given OpenShift project used for accessing private Git repositories.

```
POST /api/v2/nubomedia/paas/secret
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in	True	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	True	String
Body Parameter	secret	Contains the private key	True	String
Body Parameter	projectName	Defines the project name in OpenShift	True	String

Responses

HTTP Code	Description	Schema
200	success	String

Consumes

- application/json

Produces

- application/json

Delete a secret
This method deletes a given secret.

```
POST /api/v2/nubomedia/paas/secret/{projectName}/{secretName}
```

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	True	String
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	True	String

Responses

HTTP Code	Description	Schema
200	success	String

Consumes

- /

Produces

- application/json

3.4.1.3 Identity Management

The identity management manages access rights of the PaaS Manager itself. The concept introduced is user-driven which means that identities, namely users, are assigned to one or more projects with specific roles. Hence, every user can have a specific role in a certain project. In turn, a certain project may contain several users which have access to all applications deployed under the project. User-role relations define the rights a user have. In particular, a user with the role “USER” has full access to the assigned project. Full access mean more in detail, the user can deploy and remove applications whereas a user with the role “GUEST” can access the PaaS Manager and the assigned project but can only gather information, for instance, retrieve application details and information for using the application itself. However, a “GUEST”-user cannot deploy or remove any applications. Overall permissions are given to “ADMIN” users which have access to the entire PaaS Manager. Administrators can select all existing projects and moreover, admins have also access to the identity management in order to create, modify and remove projects, users and roles. A particular role is given to the “admin” project and user. Both the “admin” project and “admin” user is existing by default and cannot be deleted by any administrator. Additionally, the “admin” project shows all existing applications including also all applications from all the other projects. In addition, the “admin” user is the initial and only user which is present from the beginning in order to access the PaaS Manager.

3.4.1.4 Application logs

Application logs can be retrieved at both instantiation time and run time. Hence, the PaaS Manager communicates directly with OpenShift in order to request the logs while building, deploying and executing the application. These logs are provided back to the GUI of the PaaS Manager which gives the application developer/maintainer the ability to debug and observe the status of the application at any time.

3.4.1.5 Marketplace

The newly introduced marketplace serves as a platform in order to offer and distribute applications to the community. Hence, it contains the definitions of applications which can be downloaded and on-boarded to the PaaS Manager in a single step. The Marketplace is a Java Application making using an external database for storing application definitions. It exposes a REST API for CRUD operations upon application descriptors. Therefore, the PaaS Manager GUI directly interact with the Marketplace in order to expose a dynamic catalogue to application developers who could store or retrieve application descriptors anytime. Moreover, the PaaS Manager is configurable in order to change the marketplace URL by defining its IP. This enables also the usage of different marketplaces, either for using a public marketplace populated by the community or a private marketplace containing the definition of applications which should not be publicly available for the community.

3.4.1.6 Manual scaling of media servers

Manual scaling of media servers: The PaaS Manager allows manual scaling actions, in particular, scaling in and scaling out. Therefore, once the PaaS Manager received the request for scaling, it requests the NFVO in order to execute the given scaling action. The NFVO forwards the request to the Media Server Manager which executes finally the scaling action. If scaling out is requested, the Manager allocates a new mediaserver and puts it to the list of available instances which might be used directly for registering new sessions and occupying capacity. If a scaling in action is requested, the Media Server Manager terminates the given instance immediately without taking care about any applications or any sessions using this specific instance. Applications or sessions which used this instance will be removed from point of view of the Media Server Manager.

3.4.1.7 Deploy supporting services with an application

As specified by requirement DRT_R4, supporting services are introduced in order to give the application developer an easy mechanism to make use of additional and existing services. These services provide additional functionalities which are consumed by the main application, e.g. database and other services. Information of these additional services are provided back to main application based on the name of the supporting service itself and the parameters provided. The URL for accessing the service are available in the main application composed by the name of the service and the keys of the environment variables. In case of the URL, for instance, <NAME>_HOST.

3.4.2 OpenShift as the NUBOMEDIA PaaS System

OpenShift is one of the most used platforms in PaaS segment, its infrastructure is based on three main actors:

- Docker as technology for running containers, it uses the linux kernel namespace feature to provide operative systems abstraction.
- Kubernetes, a middleware provided by Google, it handles clusters of containers (as the ones created by docker)
- Project Atomic: as project to run upgrades, scaling and rollback of containers

These three technologies are container-centric and OpenShift act as a glue between them to make a high-performance PaaS.⊠OpenShift is available in three different editions:

- Online: OpenShift online is the public version of the PaaS, is available for all developers using all official docker images from platform developers and OS developers (such as JBoss, Java, Wordpress)
- Community: is opensource and available for installation on all distribution from RedHat, is a fully functional version
- Enterprise: has the same functionalities of the community with additional support for installation, security configuration (for LDAP Identity Provider and more features that can use external services). OpenShift has at least 5 configuration file, those files have to be sent (or some of them will be generated by platform itself using default values with cli commands) using REST APIs to specific paths. The referencing between these configuration files are made using names, so for the developer is useful to define a name convention and use it when it sends all the files (as JSON objects) through the REST APIs.

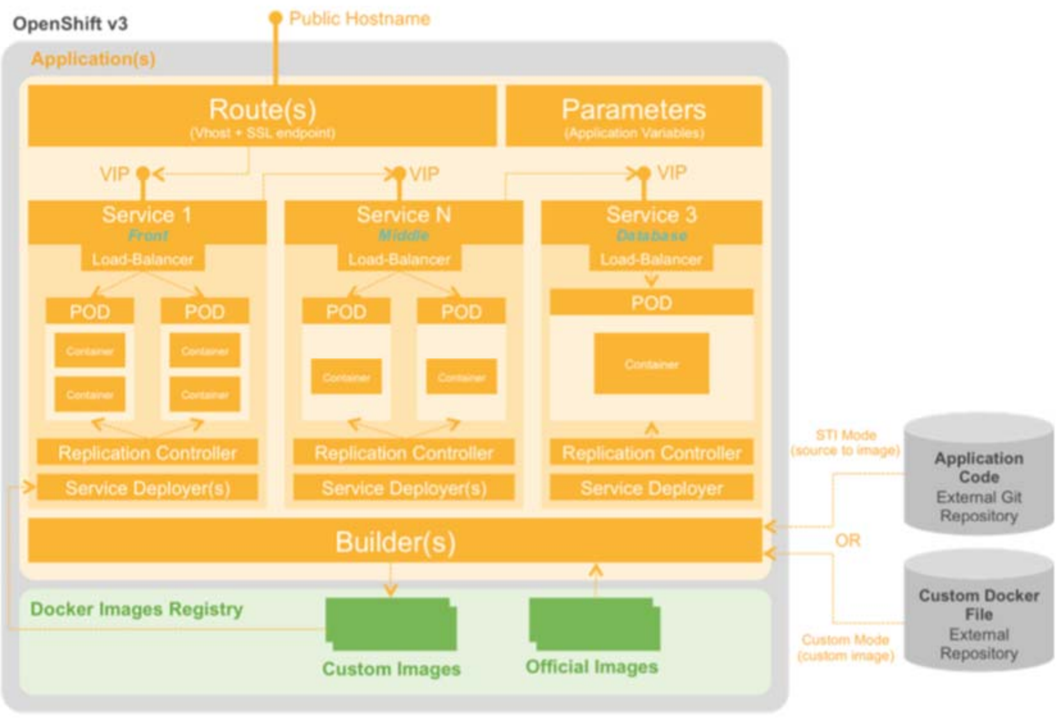


Figure 41. OpenShift general architecture

The PaaS requires an ImageStream configuration to define the output Docker image (or other internal resources, that can be "translated" in one or more Docker images); it has to define a name that will be used by OpenShift to define the Docker output image (compiling the field "dockerImageRepository"). The image name as to be provided to the BuildConfig configuration file.

3.4.2.1 Build

OpenShift uses a BuildConfig configuration file where the developer (through the REST API or web interfaces, for the latest releases) has to provide essentials data for creating, building and running the application; this data consists at least of:

- Name: a unique name for the BuildConfig, that will be used for referencing builds and can cause the Duplication Error in case of same name definition (HTTP Error 409)
- Git Repository: a public (or private, but it requires a secret) git repository where is present the application code/binaries and eventually a Dockerfile (depends on which kind of build procedure is available for that application)
- Secret: is a parameter that has to be used only in case the Git repository is private; it consists in a data structure that encapsulate the private ssh key (better if is "deployment-only") to perform the git clone operation. This key will be secured using base64 transformation algorithm without any other secret or passphrase (for the moment)
- SourceImage/BuilderImage name: depending on which building approach the developer is using, it is the name of the source image or the builder image that OpenShift has to use to start creating the final image
- Triggers: the build process could be started in three ways:
  - Manually: the build process starts clicking a button on web interface or sending a command using the CLI
  - ConfigChange: the build process starts when something changes in the configuration file (even when the configuration file itself is created)

  - ImageChange: the process will start when something in the source image is changed (is not used very often, more in deployment process)
- OutputImage: is the name that will be retrieved creating the ImageStream

The build process consists in a Pod that runs an OpenShift docker container for the chosen build process. This pod will exit (with return value equal 0), if the build process ended without any problems and OpenShift will take the result from the status of the Pod. The status could be:

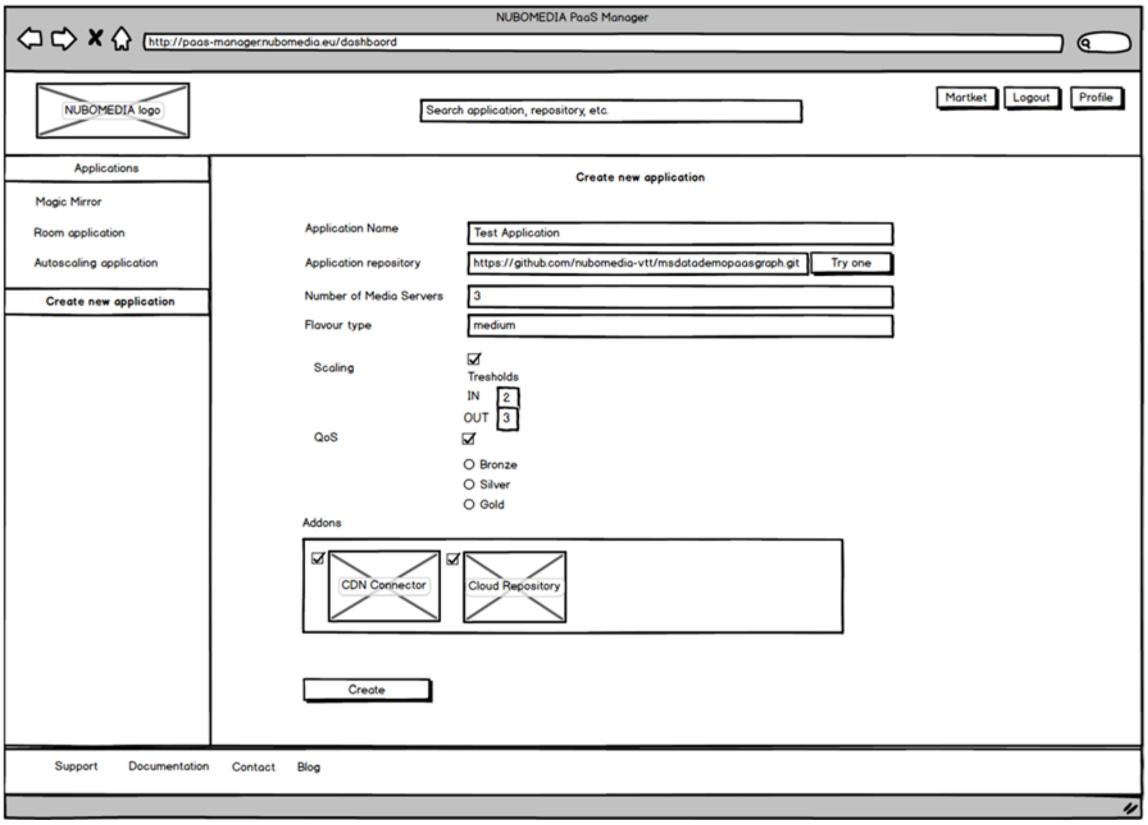
- Running: the build process is still on going, the build logs could be retrieved from REST APIs, CLI or Web Interface (latest versions)
- Failed: something goes wrong during the build, OpenShift will update the build status with failure and the deployment stage won't start.
- Complete: the build process ended with a success (this does not mean that the application was built correctly, it depends on the instructions provided to builder Pod) and the Deployment stage will be executed. OpenShift defines three different kind of build process: Docker build process, Source to Image (S2I) build process and Custom build process.

The first could use a different source image for each application, the second one define a common image with building scripts and take in consideration only the source code and the third could be defined entirely by the developer itself. The chosen build process has to be specified in the BuilConfig file with appropriate JSON keys.

3.4.3 NUBOMEDIA PaaS GUI

The NUBOMEDIA PaaS GUI provides an easy-to-use web application which could be used by developers for managing and troubleshooting the lifecycle of their applications in addition to the REST APIs provided by the PaaS Manager. Nonetheless, the PaaS GUI uses the PaaS APIs for interacting with the PaaS Manager, retrieving details about running applications.

In order to have the PaaS GUI implemented we needed to first gather the user requirements on D2.3.3. and then to identify what capabilities are needed to be implemented on the PaaS. We then needed to find what is the best place inside the interface. For this we first designed a mockup with just basic functionalities that was then subject for discussion and changes during two weeks of iterations. Some of the first mockup pages can be seen in the next two figures:

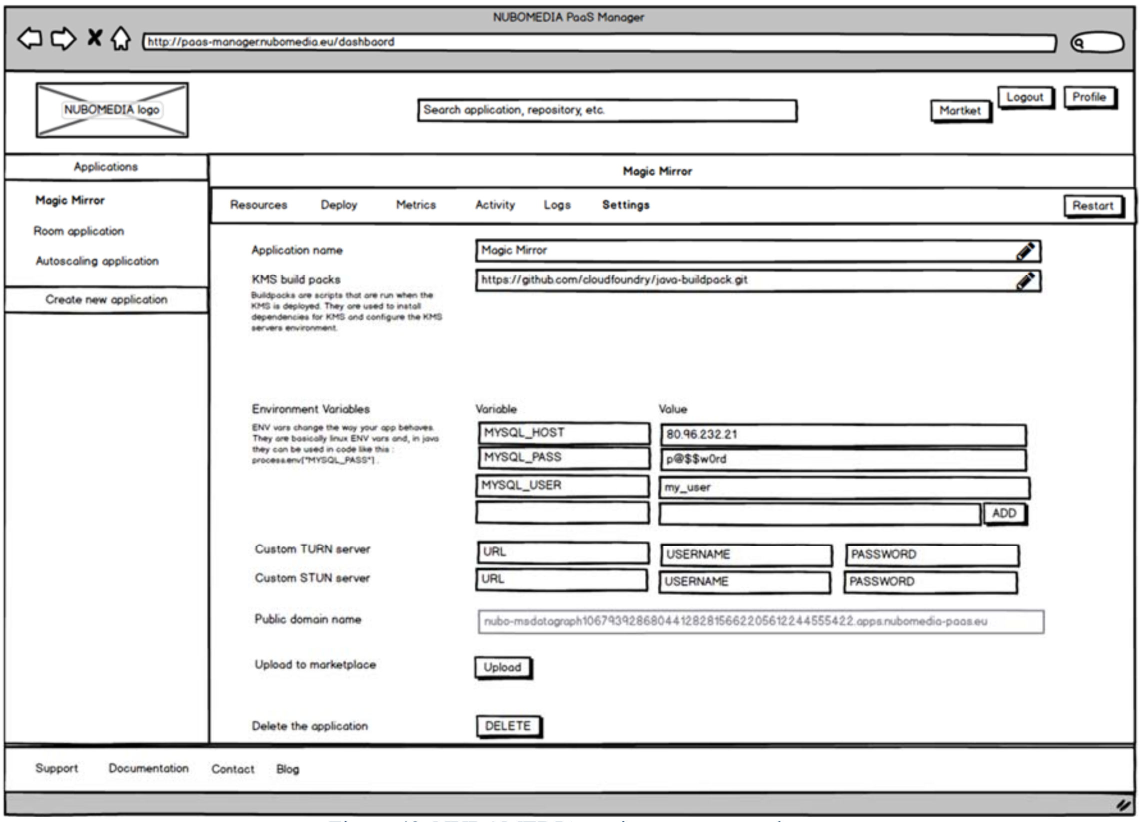


Create new application

Application Name	Test Application
Application repository	https://github.com/nubomedia-vtt/msdatademopaasgraph.git [Try one]
Number of Media Servers	3
Flavour type	medium
Scaling	☑ Tresholds IN 2 OUT 3
QoS	☑ ○ Bronze ○ Silver ○ Gold
Addons	☑ CDN Connector ☑ Cloud Repository

Create

Figure 42. NUBOMEDIA application deployment - mockup



Application name: Magic Mirror

KMS build packs: https://github.com/cloudfoundry/java-buildpack.git

Buildpacks are scripts that are run when the KMS is deployed. They are used to install dependencies for KMS and configure the KMS servers environment.

Environment Variables

ENV vars change the way your app behaves. They are basically linux ENV vars and, in java they can be used in code like this : process.env["MYSQL_PASS"] .

Variable	Value
MYSQL_HOST	80.96.232.21
MYSQL_PASS	p@$$w0rd
MYSQL_USER	my_user
	[ADD]

Custom TURN server	URL	USERNAME	PASSWORD
Custom STUN server	URL	USERNAME	PASSWORD

Public domain name: nubo-msdatagraph106793928680441282815662205612244555422.apps.nubomedia-paas.eu

Upload to marketplace: Upload

Delete the application: DELETE

Figure 43. NUBOMEDIA settings page – mockup

After concluding on the functionalities part, we then started the design phase, which had as an output an Invision[12] project. Invision allows you to transform your designs into clickable, interactive prototypes that ca then be shared so you can then receive feedback

[12] https://projects.invisionapp.com/share/W87RRH9CV#/screens/169667441_1

directly on the design and you can collaborate with others. We've added figure 42 in order to present the proposed design in case Invision link is no longer working.

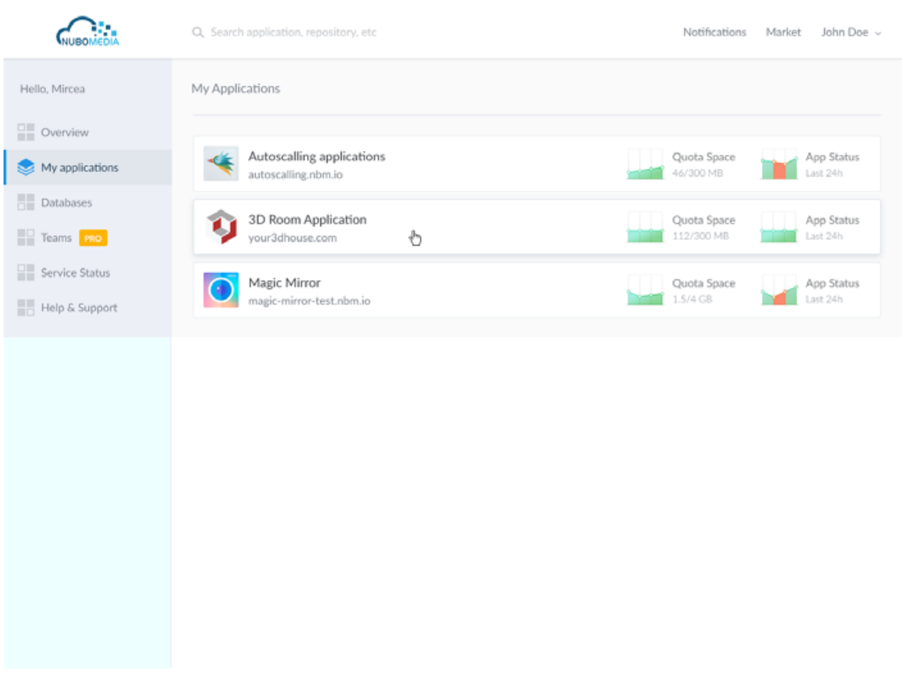


Figure 44. NUBOMEDIA PaaS GUI – My applications page design

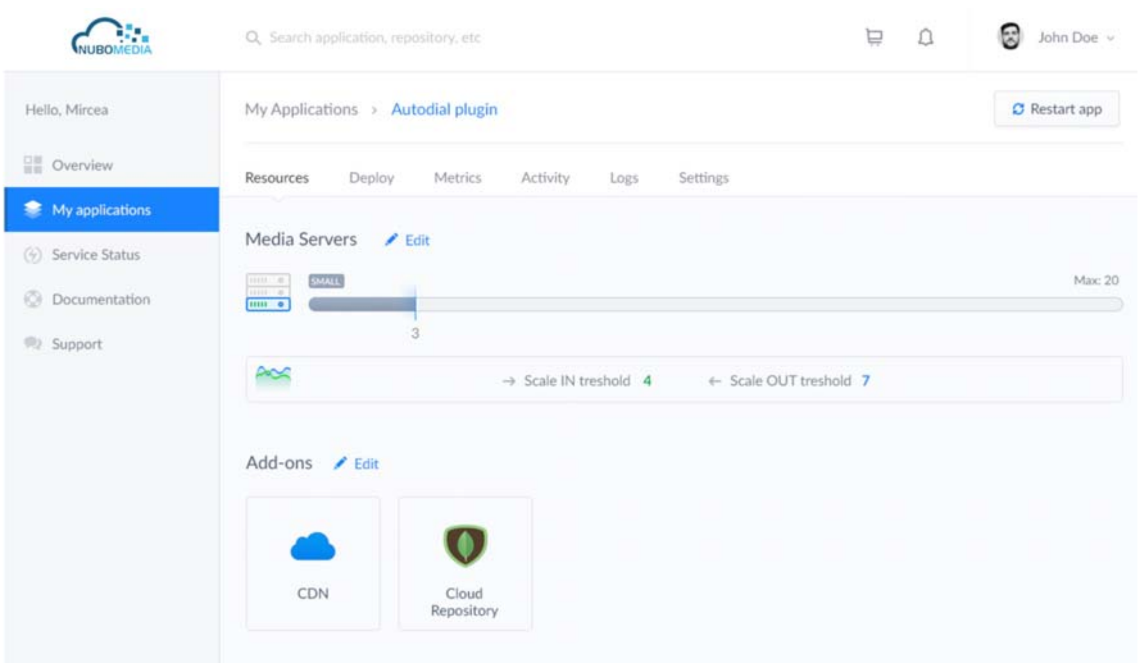
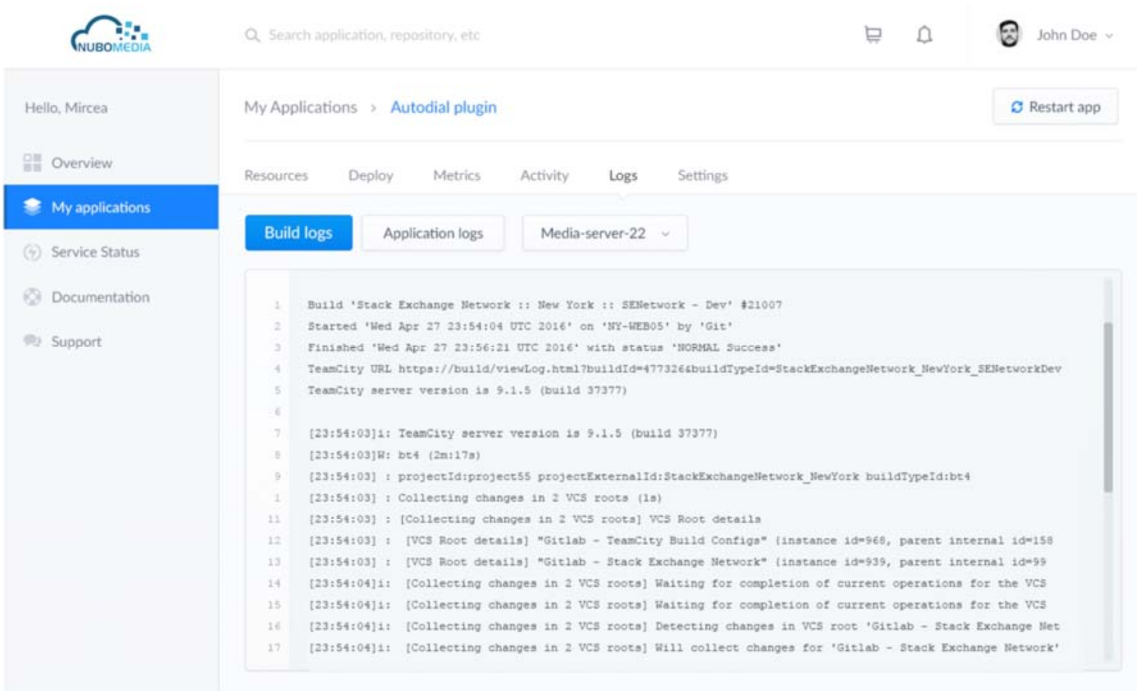


Figure 45 NUBOMEDIA PaaS GUI - Application overview



```
Build 'Stack Exchange Network :: New York :: SENetwork - Dev' #21007
Started 'Wed Apr 27 23:54:04 UTC 2016' on 'NY-WEB05' by 'Git'
Finished 'Wed Apr 27 23:56:21 UTC 2016' with status 'NORMAL Success'
TeamCity URL https://build/viewLog.html?buildId=477326&buildTypeId=StackExchangeNetwork_NewYork_SENetworkDev
TeamCity server version is 9.1.5 (build 37377)

[23:54:03]i: TeamCity server version is 9.1.5 (build 37377)
[23:54:03]W: bt4 (2m:17s)
[23:54:03] : projectId:project55 projectExternalId:StackExchangeNetwork_NewYork buildTypeId:bt4
[23:54:03] : Collecting changes in 2 VCS roots (1s)
[23:54:03] : [Collecting changes in 2 VCS roots] VCS Root details
[23:54:03] :  [VCS Root details] "Gitlab - TeamCity Build Configs" {instance id=968, parent internal id=158
[23:54:03] :  [VCS Root details] "Gitlab - Stack Exchange Network" {instance id=939, parent internal id=99
[23:54:04]i:  [Collecting changes in 2 VCS roots] Waiting for completion of current operations for the VCS
[23:54:04]i:  [Collecting changes in 2 VCS roots] Waiting for completion of current operations for the VCS
[23:54:04]i:  [Collecting changes in 2 VCS roots] Detecting changes in VCS root 'Gitlab - Stack Exchange Net
[23:54:04]i:  [Collecting changes in 2 VCS roots] Will collect changes for 'Gitlab - Stack Exchange Network'
```

Figure 46 NUBOMEDIA PaaS GUI - Logs section

3.4.3.1 Technologies behind the NUBOMEDIA PaaS GUI

NUBOMEDIA PaaS GUI is a single page web application that was created using mainly AngularJS framework for JavaScript functionality and Bootstrap framework for designing components with HTML and CSS.

A single-page application (SPA) is a web application that fits on a single web page with the goal of providing a user experience similar to that of a desktop application. In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although the location hash can be used to provide the perception and navigability of separate logical pages in the application, as can the HTML5 pushState() API. Interaction with the single page application often involves dynamic communication with the web server behind the scenes.

Although Bootstrap was used, lots of components from the application were implemented using custom CSS and HTML. In order to have a clean and a modular approach the CSS was written using Sass preprocessor and in this way all the CSS components were stored in separated files giving the project a more robust and stable file structure. Some of the ways that frameworks can help a project:

- Prevent repetition between projects
- Utilize responsive design to allow your website to adapt to various screen sizes – mobile, desktop, and everything in between
- Add consistency to design and code between projects and between developers
- Quickly and easily prototype new designs
- Ensure cross-browser compatibility

Others plugins were used in order to make the development much easier. For example, GULP was used to compile the Sass in CSS and to concatenate source files. Font awesome was used in order to have a set of icons that are represented as fonts and as a result can be scaled much easier. We used Google charts in order to draw the applications main charts on the debug section.

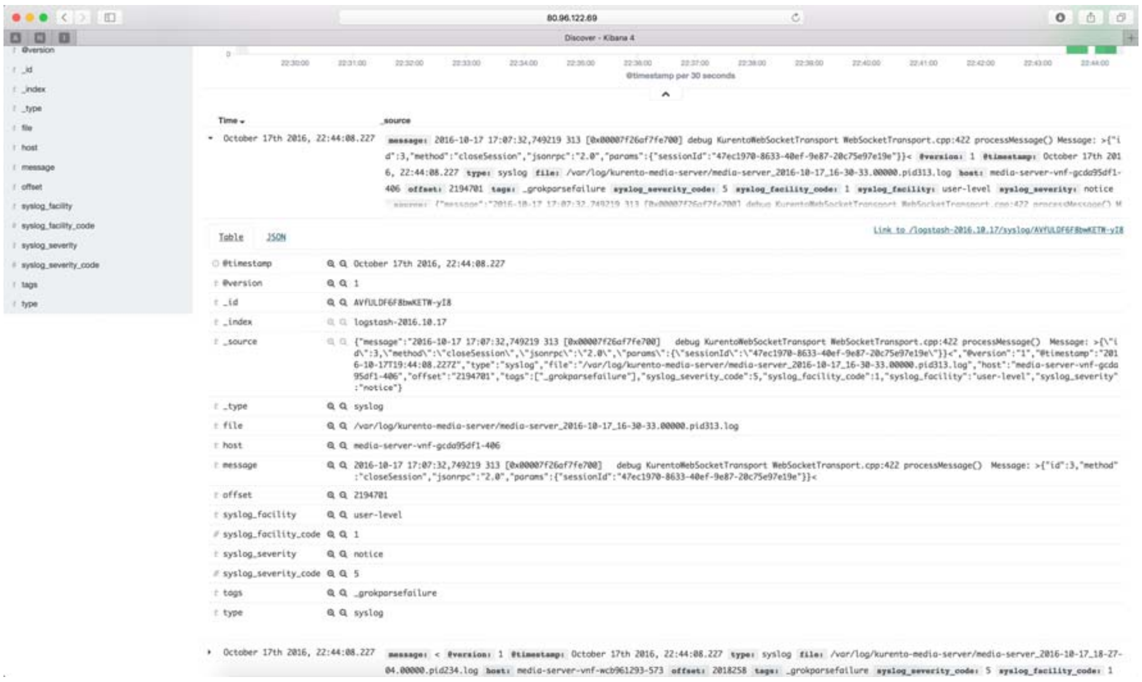
3.4.4 NUBOMEDIA Application Level Monitoring and Debugging Tools

Its main aim is to monitor the distributed execution of the NUBOMEDIA computing units: Media Server instances. We developed a Java based monitoring system that was then evolved on this review period by adding new capabilities to it for collecting WebRTC metrics from the media-servers and pushing them to Graphite on the monitoring infrastructure.

The monitoring capabilities are offered to the orchestration layer of the platform using a RESTful API. Using these metrics, the VNFM can take actions like scaling out or in. Considering the technical requirements KMS_R1 coming from the internal developers we updated the monitoring tools by adding WebRTC metrics like:

- media elements represent the number of unit performing a specific action on a media stream;
- media pipelines represent the number of active pipelines inside the media server;
- inbound jitter represents the time difference between package arrival measured in seconds;
- inbound byte count represents the number of bytes received, not including headers or padding;
- inbound packet lost represents the total number of RTP packet lost;
- inbound delta nacks represents the number of not acknowledged packets;
- inbound delta plis represents the Picture Loss Indication at inbound;
- inbound fraction lost represents the average packet Fraction Lost (packet lost / total packets ratio);
- outbound byte count represents the total number of byte count for the media-server;
- outbound Rtt represents the Round Trip Time and it’s measured in seconds;
- outbound delta plis represents the Picture Loss Indication at outbound;
- outbound delta nacks represents the number of not acknowledged packets at outbound;
- outbound target bitrate is the current target bitrate configured calculated in bits / second;

On the monitoring system, we considered the developer requirement DGB_R3 and implemented deployed Kibana 4 to help us gather the logs from all the media-server and then provide a RESTful API to the NUBOMEDIA PaaS GUI in order to offer developers a better way to diagnose any problem that their application may have. We also wanted to provide the possibility to filter the logs by different log types like debug, error or warning thing that can be helpful when debugging media-server issues. Kibana also offers the possibility filter logs using regular expressions which can be helpful for more in depth analysis.



```
Time ▾                          _source
▾ October 17th 2016, 22:44:08.227   message: 2016-10-17 17:07:32,749219 313 [0x00007f26af7fe700] debug KurentoWebSocketTransport WebSocketTransport.cpp:422 processMessage() Message: >{"i
                                d":3,"method":"closeSession","jsonrpc":"2.0","params":{"sessionId":"47ec1970-8633-40ef-9e87-20c75e97e19e"}}< @version: 1 @timestamp: October 17th 201
                                6, 22:44:08.227 type: syslog file: /var/log/kurento-media-server/media-server_2016-10-17_16-30-33.00000.pid313.log host: media-server-vnf-gcda95df1-
                                406 offset: 2194701 tags: _grokparsefailure syslog_severity_code: 5 syslog_facility_code: 1 syslog_facility: user-level syslog_severity: notice

Link to /logstash-2016.10.17/syslog/AVfULDF6F8bwKETW-yI8

@timestamp            October 17th 2016, 22:44:08.227
@version              1
_id                   AVfULDF6F8bwKETW-yI8
_index                logstash-2016.10.17
_source               {"message":"2016-10-17 17:07:32,749219 313 [0x00007f26af7fe700]   debug KurentoWebSocketTransport WebSocketTransport.cpp:422 processMessage()  Message: >{\"i
                      d\":3,\"method\":\"closeSession\",\"jsonrpc\":\"2.0\",\"params\":{\"sessionId\":\"47ec1970-8633-40ef-9e87-20c75e97e19e\"}}<","@version":"1","@timestamp":"201
                      6-10-17T19:44:08.227Z","type":"syslog","file":"/var/log/kurento-media-server/media-server_2016-10-17_16-30-33.00000.pid313.log","host":"media-server-vnf-gcda
                      95df1-406","offset":"2194701","tags":["_grokparsefailure"],"syslog_severity_code":5,"syslog_facility_code":1,"syslog_facility":"user-level","syslog_severity"
                      :"notice"}
_type                 syslog
file                  /var/log/kurento-media-server/media-server_2016-10-17_16-30-33.00000.pid313.log
host                  media-server-vnf-gcda95df1-406
message               2016-10-17 17:07:32,749219 313 [0x00007f26af7fe700]   debug KurentoWebSocketTransport WebSocketTransport.cpp:422 processMessage()  Message: >{"id":3,"method"
                      :"closeSession","jsonrpc":"2.0","params":{"sessionId":"47ec1970-8633-40ef-9e87-20c75e97e19e"}}<
offset                2194701
syslog_facility       user-level
syslog_facility_code  1
syslog_severity       notice
syslog_severity_code  5
tags                  _grokparsefailure
type                  syslog

▸ October 17th 2016, 22:44:08.227   message: < @version: 1 @timestamp: October 17th 2016, 22:44:08.227 type: syslog file: /var/log/kurento-media-server/media-server_2016-10-17_18-27-
                                04.00000.pid234.log host: media-server-vnf-wcb961293-573 offset: 2018258 tags: _grokparsefailure syslog_severity_code: 5 syslog_facility_code: 1
```

Figure 47. Kurento Media Server Logs on Kibana

4 How to install NUBOMEDIA Media Plane PaaS

The NUBOMEDIA management tools are meant to help monitoring and deploying the NUBOMEDIA platform in an easy and efficient way. For this we researched, designed and prototyped a monitoring system based on Free Open Source tools like LogStash, Kibana, Graphite and Icinga. We then created an autonomous installer using the native APIs of OpenStack that enables the deployment of the entire NUBOMEDIA platform in rapid time without having to know the internals of it.

4.1 NUBOMEDIA Autonomous Installer

The Autonomous Installer is meant to deploy a new NUBOMEDIA instance into an IaaS environment based on OpenStack and a PaaS environment based on OpenShift Origin v3.

The internal developers have requested the support for also take into consideration the installation of OpenShift as part of the autonomous installer on installation requirement IST_R1. We started to find a stable way to deploy OpenShift Origin v3. We researched the way to deploy it by using Ansimble scripts available on the official repository but we had no success. We have worked on this path for almost 1 month but we got into problems one after another and we tried to get help from the OpenShift community on their Github issue[13] track repository and Freenode channel #OpenShift. In the end, we concluded that the installation of OpenShift can be a complex work and should be done by experts, and we, in context of the NUBOMEDIA, should have OpenShift credentials as part of initial requirements. The good part of this is that we were contacted by one O'Relly Media producer in order to provide feedback for a book that was planned to be released, the book is now released with the following name: "OpenShift Enterprise for Developers" by Steven Pousty and Grant Shipley.

The OpenStack must be running the Kilo release alongside with the nova-docker plugin for the compute nodes. For this you will first have to uninstall the python-pbr package installed for the nova-compute, then install the version 0.10.1 needed for nova-docker installation. After this you will have to download the latest version of the nova-docker Kilo release that is available here: https://github.com/openstack/nova-docker/releases/tag/kilo-eol . Next you will have to follow the steps for installing and configuring it and then you should update back the python-pbr to version 1.8.1-2.

The PaaS should be running OpenShift Origin v3 version 3.1 with the following adjustment: To relax the security in your cluster so that images are not forced to run as a pre-allocated UID, without granting everyone access to the privileged SCC. In order to make these changes you have to run the following command:

```
oc edit scc restricted
```

This will open you the configuration in edit mode. Here you will have to update the runAsUser parameter in the following way:

```
runAsUser:
  type: MustRunAsRange
```

[13] https://github.com/OpenShift/OpenShift-ansible/issues/1338

Having a constantly evolving architecture and components, creating a NUBOMEDIA installer that could install every piece of software from binaries was not feasible. Considering this, during this review period we kept maintaining the NAI functionality and extended it in order to support deployment of new capabilities that were developed during the review period. Some of the newly capabilities added are the NUBOMEDIA Cloud Repository, the NUBOMEDIA Development GUI and the NUBOMEDIA Market Place.

During this review period, we added support for installation of the entire platform in interactive mode. This functionality would help an expert system administrator installing NUBOMEDIA in an interactive mode, allowing him to input every installation details like MySQL username and password that would be used when creating the database for the NUBOMEDIA PaaS Manager or even configure the public IP addresses that would be assigned to each of the services hosts.

4.1.1 NAI Prerequisites

In order to be able to run the NUBOMEDIA Autonomous Installer you should start by checking that your current python version is 2.7 using the following command:

```
python -version
```

Then you should install pip and after that you should install the dependencies using the following commands. When asked for any kind read it first and then confirm:

```
easy_install pip
pip install -r requirements.txt --upgrade
```

If you want to use the Docker images for the Kurento Media Server you should have root access to the OpenStack environment in order to install and configure the nova-Docker hypervisor and add the patch for it from (https://github.com/usvpublic/nubomedia-nova-docker) on the compute nodes that run docker as a hypervisor. The production ready Docker images are available here : https://hub.docker.com/r/nubomedia/kurento-media-server/ and the development version is available on https://hub.docker.com/r/nubomedia/kurento-media-server-dev/ .

4.1.2 Update the configuration file

Before starting the installation, you must first rename the *variables-example.py* to *variables.py* and then gather the required variables:

- **enabled_logging** can take either true or false. If you set this to true it will create a file name installer.log that will give you verbose logging of the installation;
- **iaas_ip** is the OpenStack public IP address, where the endpoint for nova, glance, cinder and neutron are available;
- **auth_url** is the authentication endpoint of Keystone;
- **username** is the username of the OpenStack platform;
- **password** is the password for the previous declared user;
- **tenant_name** is the project name inside OpenStack;
- **glance_endpoint** is the API endpoint of the Glance image service. The list of all endpoints can be found on the "Access and Security > API Access" page of your OpenStack deployment;
- **master_ip, master_user, master_pass, master_key** are credentials that are needed in case you will use the docker image for Kurento Media Server. With the following credentials the installer will connect to all compute nodes running

docker as a hypervisor and will pull the latest development and production docker images of KMS;

- **floating_ip_pool** is the name of the external network of OpenStack;
- **private_key** is the name of the Public Key file that will be attached to all the instances started by the NUBOMEDIA Autonomous Installer and then, the NUBOMEDIA Platform;
- **OpenShift_ip** is the IP address of the OpenShift environment on top of which all the applications will be run;
- **OpenShift_keystore** represents the Keystore holding the OpenShift SSL certificate. You can create the Keystore by using Portacle.
- **OpenShift_domain** represents the wildcard domain name that will be used by as URLs for the applications that will be deployed inside OpenShift; For this you will have to define a wildcard DNS record on the IP address of OpenShift;
- **OpenShift_token** will be used to authenticate the API calls from the PaaS manager to OpenShift API endpoint. In order to optain the token you will have to ssh to the OpenShift instance and then get the list of all secrets available by using the following command: "*oc get secrets*". Then you will have to type "*oc describe secrets PROJECT-NAME-TOKEN-XXX*" in order to get the token;
- **nubomedia_admin_paas** represents the password for user admin of the NUBOMEDIA PaaS that will be deployed by the Autonomous Installer;
- **use_kurento_on_docker** can take either true or false, representing the choice between KMS on Docker and KMS on KVM;

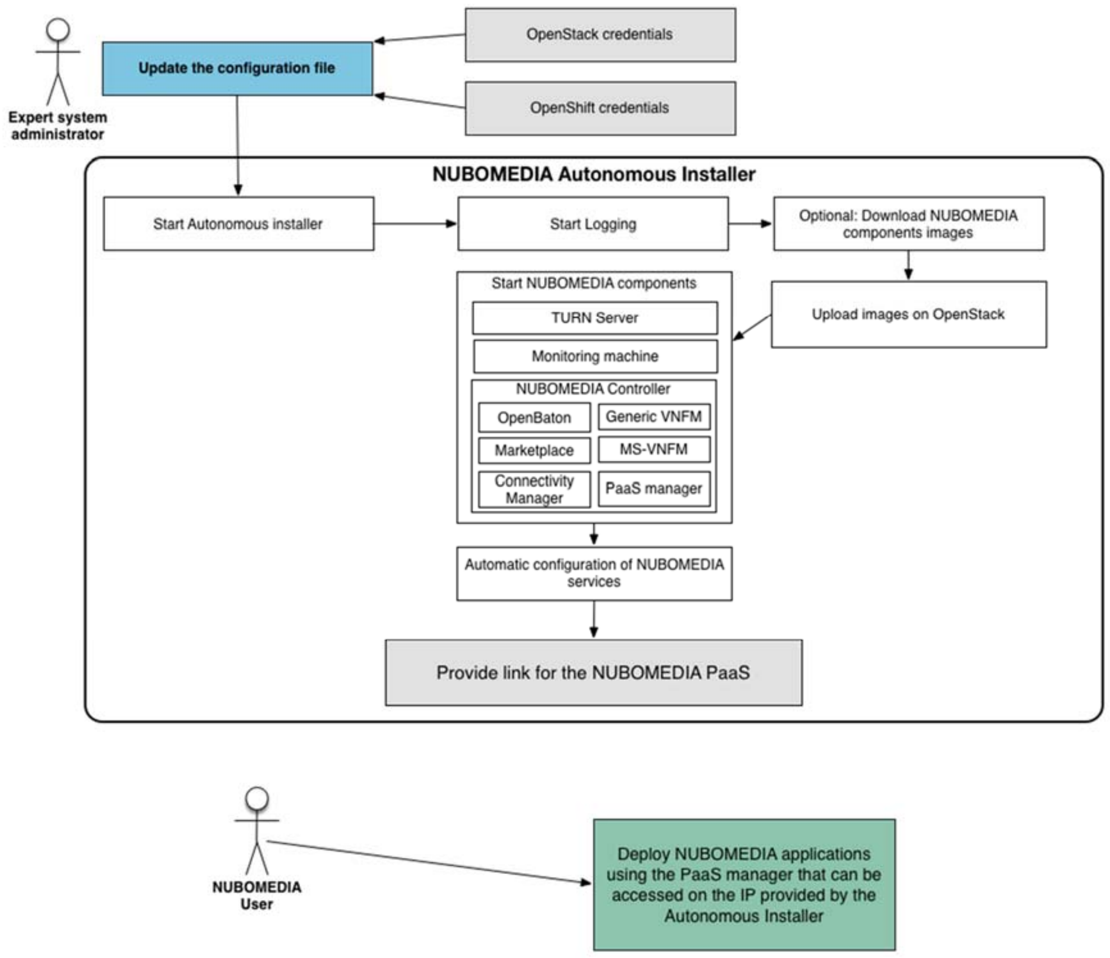

Expert system administrator
Update the configuration file
OpenStack credentials
OpenShift credentials
NUBOMEDIA Autonomous Installer
Start Autonomous installer
Start Logging
Optional: Download NUBOMEDIA components images
Upload images on OpenStack
Start NUBOMEDIA components
TURN Server
Monitoring machine
NUBOMEDIA Controller
OpenBaton
Generic VNFM
Marketplace
MS-VNFM
Connectivity Manager
PaaS manager
Automatic configuration of NUBOMEDIA services
Provide link for the NUBOMEDIA PaaS
NUBOMEDIA User
Deploy NUBOMEDIA applications using the PaaS manager that can be accessed on the IP provided by the Autonomous Installer

Figure 48 Autonomos Installer - Process flow

4.1.3 Start the installer

At this point you should have all the necessary configurations steps ready you can start the installer the actual installation of the NUBOMEDIA platform by running the following command:

```
python main.py
```

There are two ways of using the Autonomous Installer, one full automatic mode that uses the configuration file previously defined and another mode that is an interactive mode requiring the expert system administrator to input the needed details at each of the steps. After all the required information is available the NUBOMEDIA Autonomous installer will start the process of installing the platform on the IaaS you specified. In order to check the logs, you should use the tail command to monitor the installer.log file.

When the installer has completed you should receive a success message indicating the URL for the PaaS GUI where you can login with user admin and the password configured on the variables.py file, nubomedia_admin_paas.

5 How to use the NUBOMEDIA Media Plane PaaS

In order to use the NUBOMEDIA Media Plane PaaS using the PaaS GUI, a user have to login first. The user journey starts with the login form, Figure 49. There are two types of users, admin and normal user.

Admin user has access to all the views of the application while a normal user has access only to the information that is related to him. His applications and the project that he belongs to. From now on the application will be presented from the admin point of view because the admin user as was said has access to all views. After the credentials are entered the user will see the first page of the application which is a dashboard with general information about users, projects and active applications.

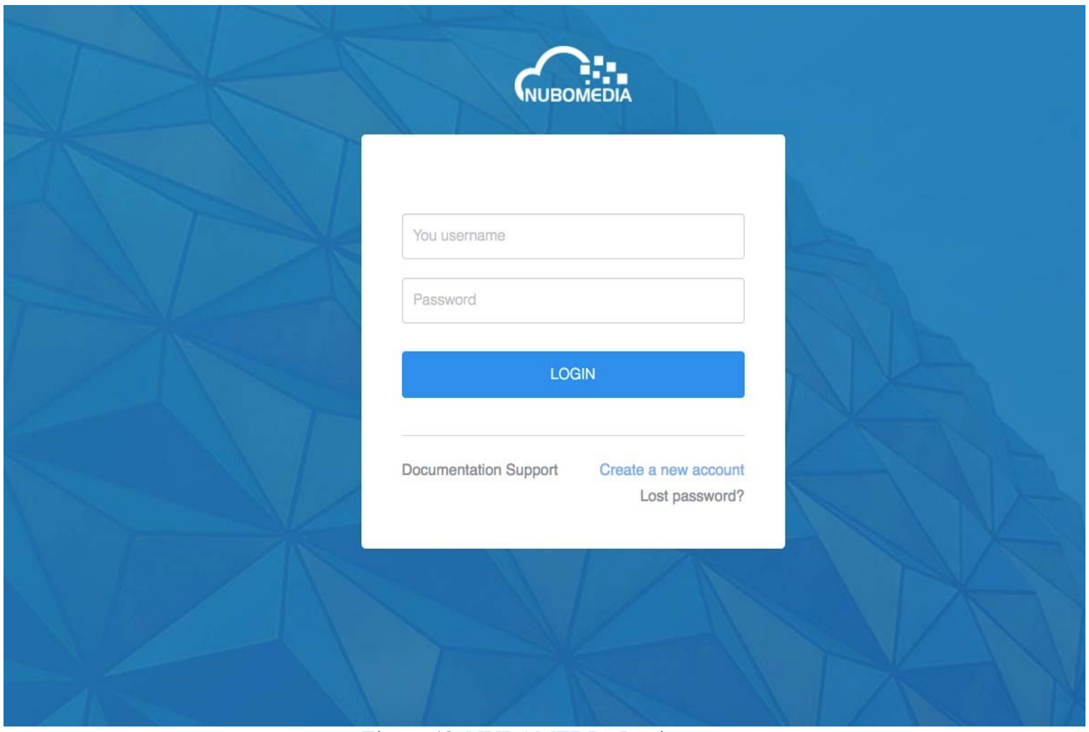


Figure 49. NUBOMEDIA Login page

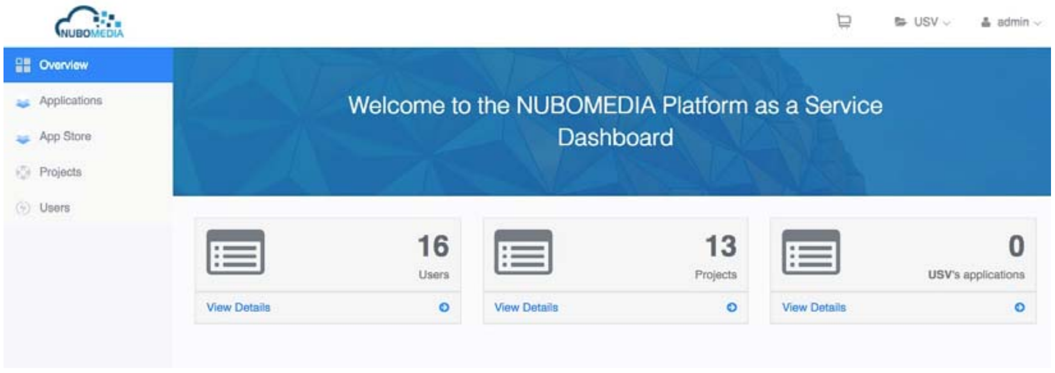


Figure 50. Dashboard view

As it can be seen in Figure 50, on the left we have the application's main navigation. In top right, we have the user name, the current active project and a link to the market applications. In the middle, we have some cards with general information about: users number, projects numbers and applications numbers. We are able to navigate to other pages using the main navigation from left or using the links from dashboard cards. Next going to Applications page (Figure 51) we have the following view:

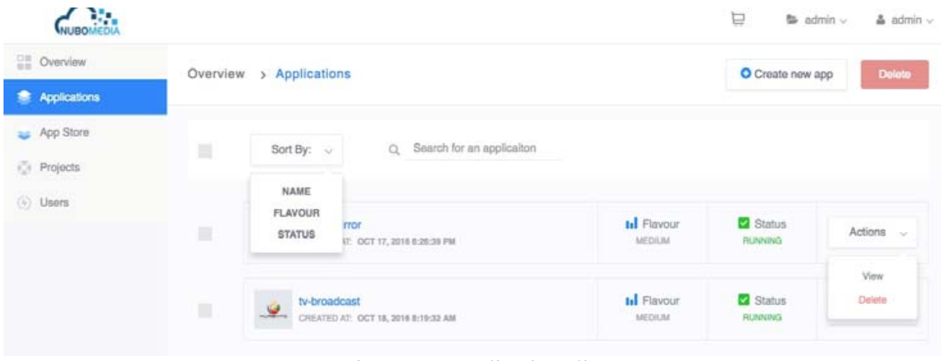


Figure 51. Applications list

In this view, we can see the list of applications related to a specific project. We also can see here the application name, the date when the application was created, the application status and its flavor.

In this view, we have the possibility to search for a specific application using the Search for an application form. The search form allows us to search not only by name but also by created date, flavor or application status. It does a smart search and the important aspect of this is that the search is not made using some predefined terms. What this means? It means that if we decide to display some extra information about applications like for example the number of servers for every application. When we would use the search, we could also search for an application that has a specific number of servers. All this without changing the search functionality.

Then we have the Sort by functionality that sorts the applications depending on what we are choosing from select box: name, flavor or status. We also have some secondary actions buttons for every application. We can delete an application or we can view the full application description and all the fields that are related to that application, we will see an example of that.

Figure 52. Breadcrumbs and main actions buttons

On the left side of the picture before we have the breadcrumb that tells the user where he is and on the right, we have the main actions buttons. Delete action that is enabled when one or more applications are checked permits the user to delete multiple applications. Create new app is redirecting the user to another view where he can create a new application that on completion will show up next to other applications. When clicking on create new app button the user is redirected to a new view where he can create a new custom application.

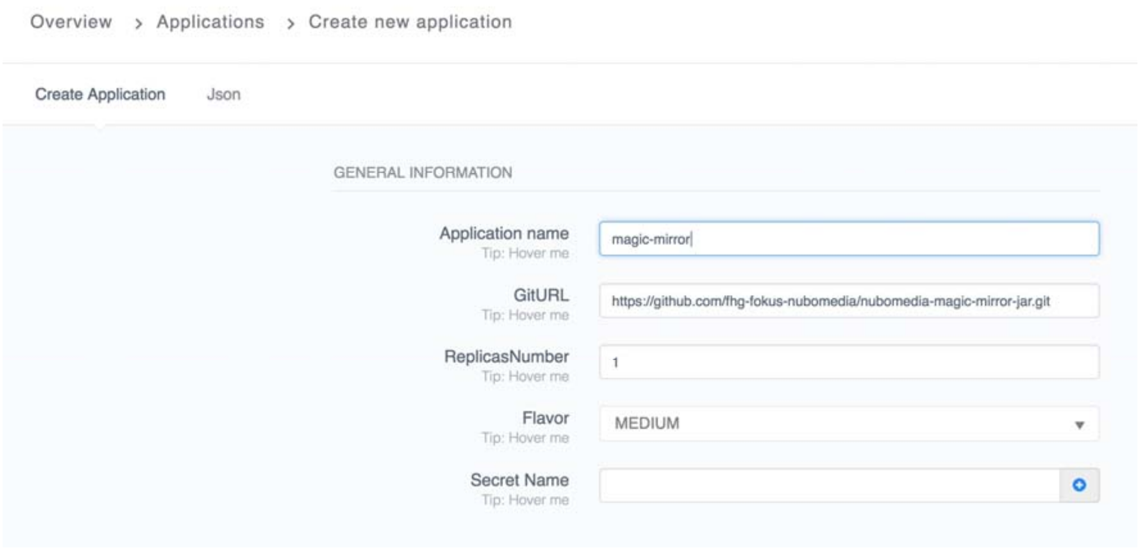
Create new application view is practically a form where the user will add specific information about the new application that he wants to create. This form is split in 3 main sections: General information, Services and Additional information.

5.1 Applications

In order to create a new application a user has to input the following fields in the general information section:

- Application name
- Git URL
- Replicas number

- Flavour
- Secret name


Overview > Applications > Create new application

Create Application | Json

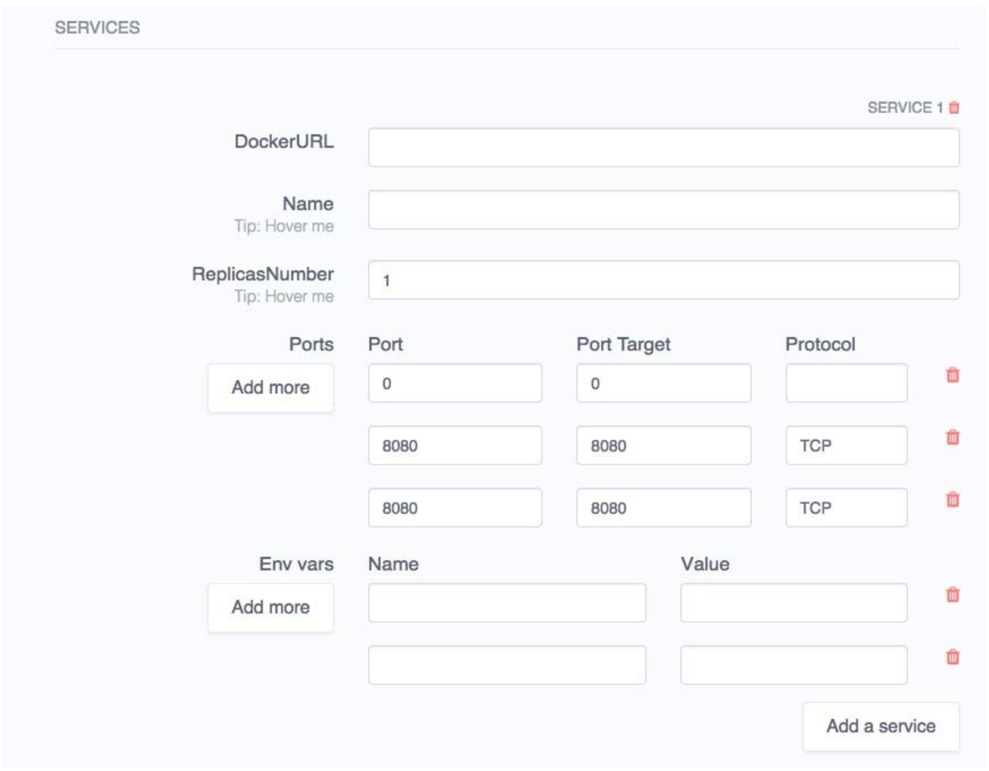
GENERAL INFORMATION

Application name (Tip: Hover me)	magic-mirror
GitURL (Tip: Hover me)	https://github.com/fhg-fokus-nubomedia/nubomedia-magic-mirror-jar.git
ReplicasNumber (Tip: Hover me)	1
Flavor (Tip: Hover me)	MEDIUM
Secret Name (Tip: Hover me)	

Figure 53 NUBOMEDIA PaaS GUI - Create new app view: general information section

The services section is optional and the user has the possibility to add multiple services using the Add a service button. In this section, you should add the following details about the add-on:

- Docker URL
- Name
- Replicas number
- Multiple ports
- Multiple environment variables


SERVICES

SERVICE 1

DockerURL	
Name (Tip: Hover me)	
ReplicasNumber (Tip: Hover me)	1

Ports

Port	Port Target	Protocol
0	0	
8080	8080	TCP
8080	8080	TCP

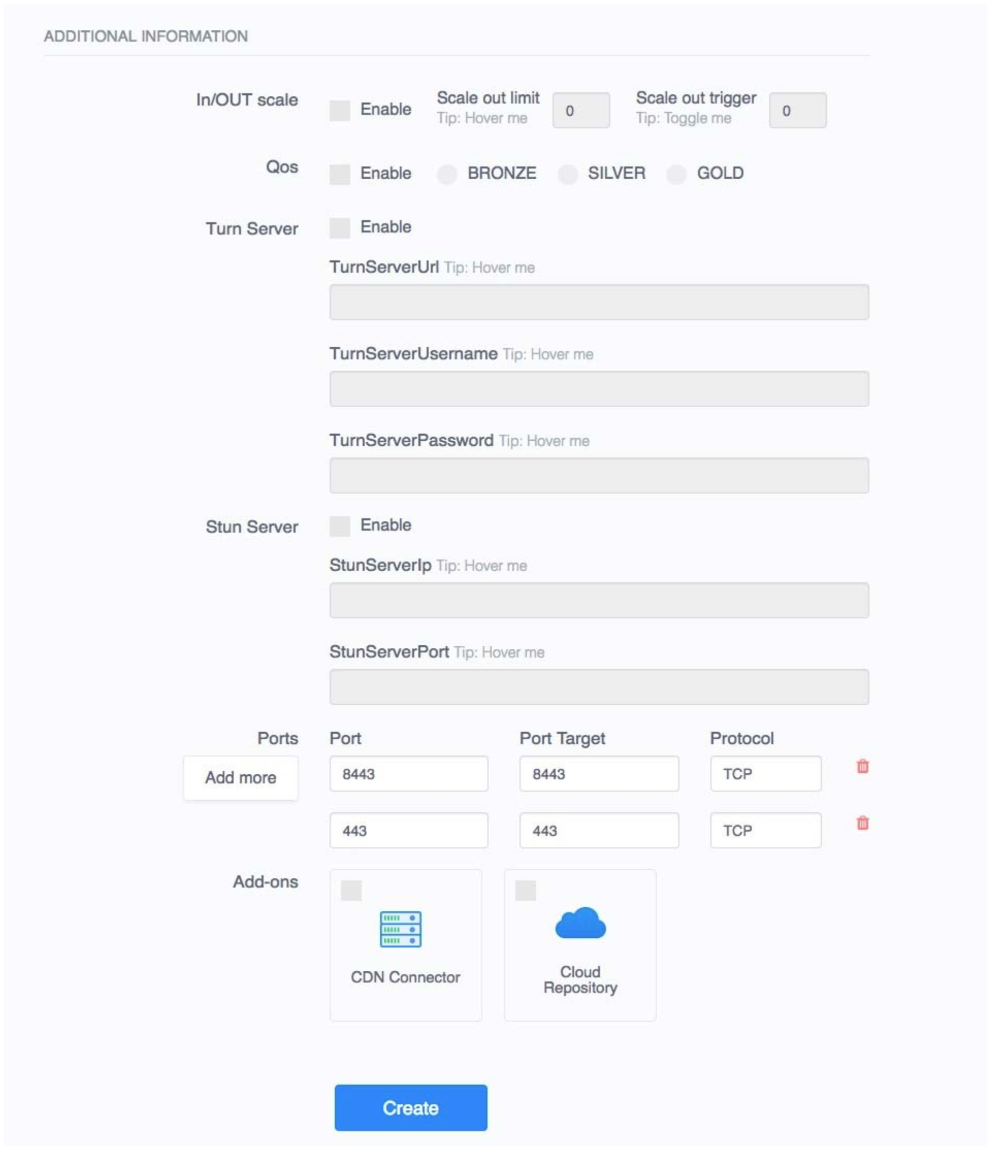
Env vars

Name	Value

Figure 54 NUBOMEDIA PaaS GUI - Additional services

In Additional information section, the user can opt to enable auto scaling of the media servers by configuring the Scale out limit and Scale out trigger. We enabled the auto-scaling of the media servers considering the requirement PGUI_R1 gathered from internal and external users.

A user is also able to override the configurations for having a custom Turn or Stun server or define some custom ports that his application need to expose. Another option is to have deployed along with the application also a CDN Connector or a Cloud Repository with APIs that can be consumed by the deployed application. All these custom options are illustrated in the following screenshot.


ADDITIONAL INFORMATION

In/OUT scale — ☐ Enable — Scale out limit (Tip: Hover me) 0 — Scale out trigger (Tip: Toggle me) 0

Qos — ☐ Enable — ○ BRONZE ○ SILVER ○ GOLD

Turn Server — ☐ Enable

TurnServerUrl Tip: Hover me

TurnServerUsername Tip: Hover me

TurnServerPassword Tip: Hover me

Stun Server — ☐ Enable

StunServerIp Tip: Hover me

StunServerPort Tip: Hover me

Ports

Port	Port Target	Protocol
8443	8443	TCP
443	443	TCP

Figure 55 NUBOMEDIA PaaS GUI - Create application with additional information

After all the necessary fields are completed the user can click the create button and a new application will be created with the specified parameters.

An important aspect of this step is that the user has the possibility to create the same application but using the JSON tab of the form. This tab of the form permits the user to create a new application by uploading a configuration JSON file from his computer.

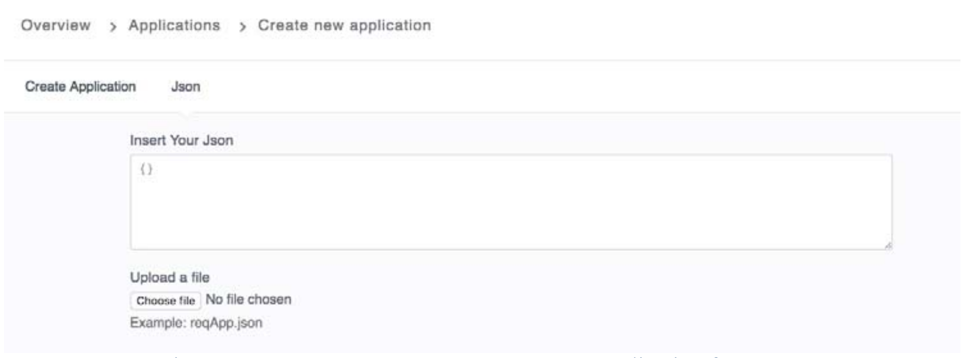


Figure 56 NUBOMEDIA PaaS GUI - Create application from Json

In application details view the user has all the information about an application separated in 4 tabs:

- Overview
- Metrics
- Debug
- Settings

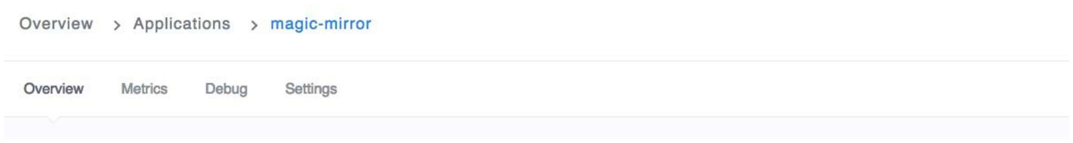


Figure 57 NUBOMEDIA PaaS GUI - Application Details

Application overview tab has the application URL, media servers' information and the application scalability features like powering on or stopping one KMS instance or even scale OUT/IN the number of media-servers.

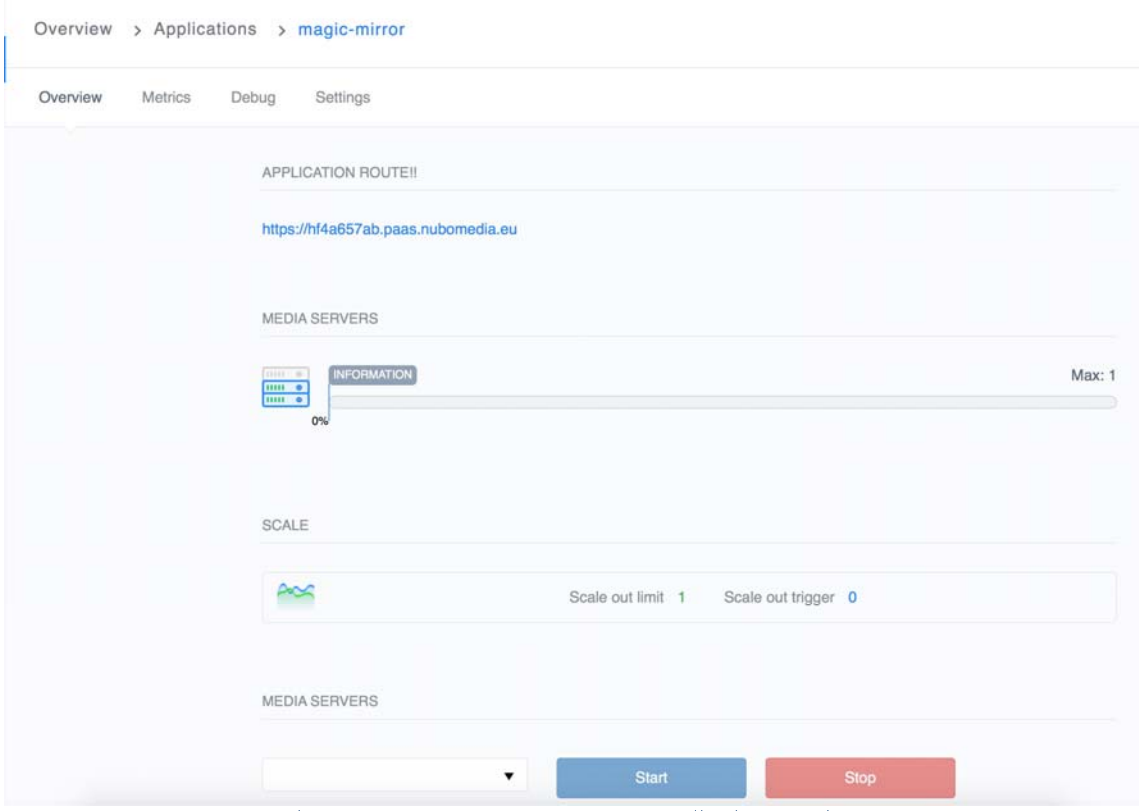


Figure 58 NUBOMEDIA PaaS GUI - Application overview

Metrics tab contains various charts with information requested in technical requirement gathered in D2.3.3, DRT_R11 having the following description: "it shall be possible to monitor the CPU, Memory, and other metrics available".Some of the collected metrics are the following:

- Numbers of KMS instances
- Total capacity used
- Media servers' metrics: Memory, Elements, Pipelines

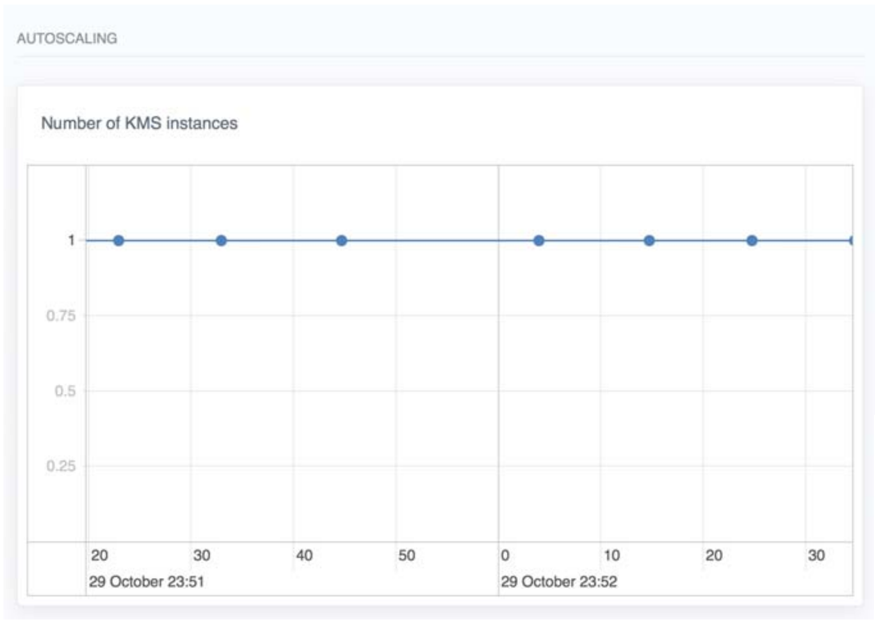


Figure 59 NUBOMEDIA PaaS GUI - Scaling information

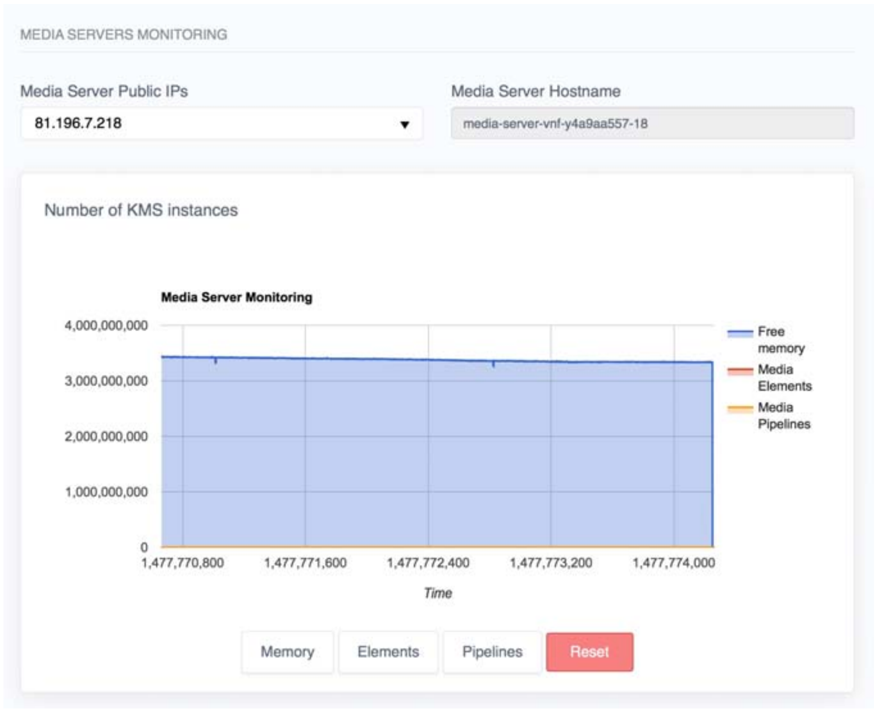
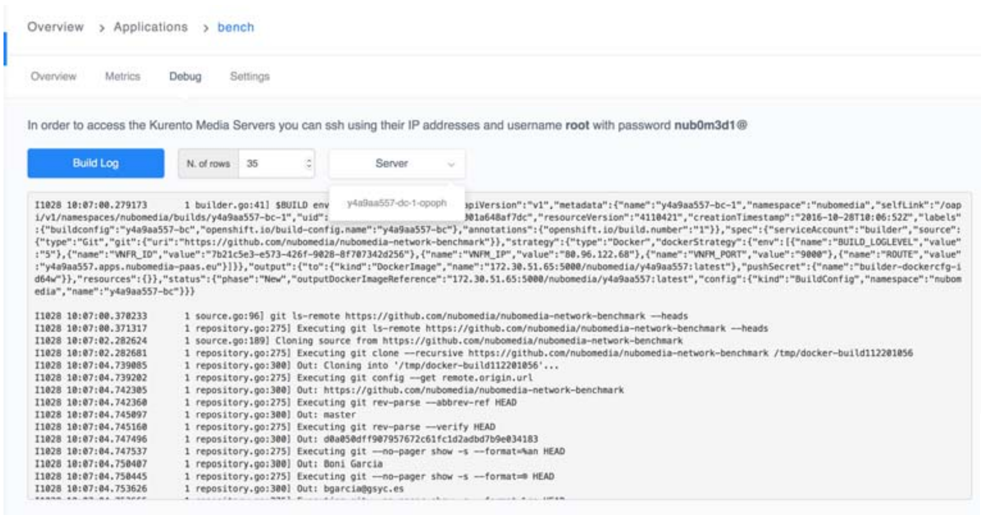


Figure 60 NUBOMEDIA PaaS GUI - Media Server monitoring information

Debug tab contains more information about media servers. Here the user has the possibility to build application logs. Also, a user can start or stop a specific server.

In order to fulfill the developer requirement DRT_R3, to allow the developer to SSH into the KMS instances, we configured all media-server Docker containers and instances with a username and password so anyone can login on any of the IP addresses of the media-servers found on the media-server monitoring tab. This feature is now not so important because we also have now the possibility to check the media-server logs directly on the PaaS GUI interface.



In order to access the Kurento Media Servers you can ssh using their IP addresses and username **root** with password **nub0m3d1@**

```
I1028 10:07:00.279173       1 builder.go:41] $BUILD env                  apiVersion":"v1","metadata":{"name":"y4a9aa557-bc-1","namespace":"nubomedia","selfLink":"/oap
i/v1/namespaces/nubomedia/builds/y4a9aa557-bc-1","uid":                  01a648af7dc","resourceVersion":"4110421","creationTimestamp":"2016-10-28T10:06:52Z","labels"
:{"buildconfig":"y4a9aa557-bc","openshift.io/build-config.name":"y4a9aa557-bc"},"annotations":{"openshift.io/build.number":"1"}},"spec":{"serviceAccount":"builder","source":
{"type":"Git","git":{"uri":"https://github.com/nubomedia/nubomedia-network-benchmark"}},"strategy":{"type":"Docker","dockerStrategy":{"env":[{"name":"BUILD_LOGLEVEL","value"
:"5"},{"name":"VNFR_ID","value":"7b21c5e3-e573-426f-9028-8f707342d256"},{"name":"VNFM_IP","value":"80.96.122.68"},{"name":"VNFM_PORT","value":"9000"},{"name":"ROUTE","value"
:"y4a9aa557.apps.nubomedia-paas.eu"}]}},"output":{"to":{"kind":"DockerImage","name":"172.30.51.65:5000/nubomedia/y4a9aa557:latest"},"pushSecret":{"name":"builder-dockercfg-i
d64w"}},"resources":{}},"status":{"phase":"New","outputDockerImageReference":"172.30.51.65:5000/nubomedia/y4a9aa557:latest","config":{"kind":"BuildConfig","namespace":"nubom
edia","name":"y4a9aa557-bc"}}}

I1028 10:07:00.370233       1 source.go:96] git ls-remote https://github.com/nubomedia/nubomedia-network-benchmark --heads
I1028 10:07:00.371317       1 repository.go:275] Executing git ls-remote https://github.com/nubomedia/nubomedia-network-benchmark --heads
I1028 10:07:02.282624       1 source.go:189] Cloning source from https://github.com/nubomedia/nubomedia-network-benchmark
I1028 10:07:02.282681       1 repository.go:275] Executing git clone --recursive https://github.com/nubomedia/nubomedia-network-benchmark /tmp/docker-build112201056
I1028 10:07:04.739085       1 repository.go:300] Out: Cloning into '/tmp/docker-build112201056'...
I1028 10:07:04.739202       1 repository.go:275] Executing git config --get remote.origin.url
I1028 10:07:04.742305       1 repository.go:300] Out: https://github.com/nubomedia/nubomedia-network-benchmark
I1028 10:07:04.742360       1 repository.go:275] Executing git rev-parse --abbrev-ref HEAD
I1028 10:07:04.745097       1 repository.go:300] Out: master
I1028 10:07:04.745160       1 repository.go:275] Executing git rev-parse --verify HEAD
I1028 10:07:04.747496       1 repository.go:300] Out: d0a050dff907957672c61fc1d2adbd7b9e034183
I1028 10:07:04.747537       1 repository.go:275] Executing git --no-pager show -s --format=%an HEAD
I1028 10:07:04.750407       1 repository.go:300] Out: Boni Garcia
I1028 10:07:04.750445       1 repository.go:275] Executing git --no-pager show -s --format=%ae HEAD
I1028 10:07:04.753626       1 repository.go:300] Out: bgarcia@gsyc.es
```

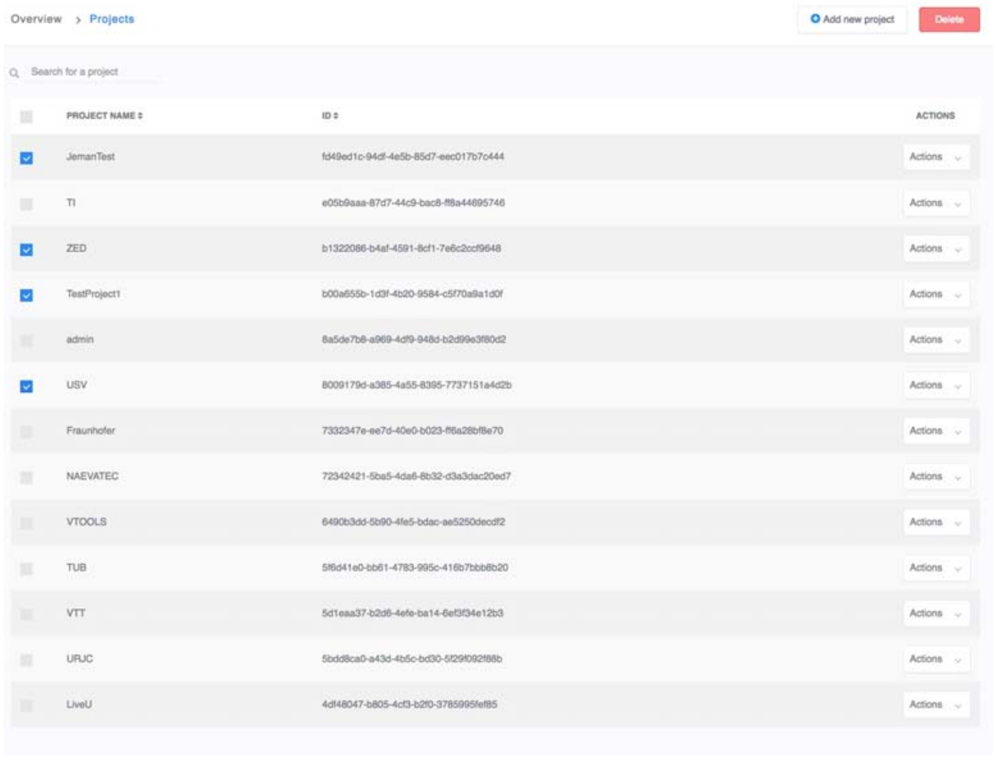
Figure 61 NUBOMEDIA PaaS GUI - Debug section

In settings tab user have access to other application information. All the information about the application is also available in a JSON format. All the fields in this tab are split in 4 sections:

5.2 Projects

Projects are basically separated workspace environments. You can combine users with projects in order to have the desired matrix of permissions.

As an admin, the projects page gives the user the possibility to view all the projects. The user can filter the projects by project name or id. Also, here we can search for a specific project. We can create or delete a project.



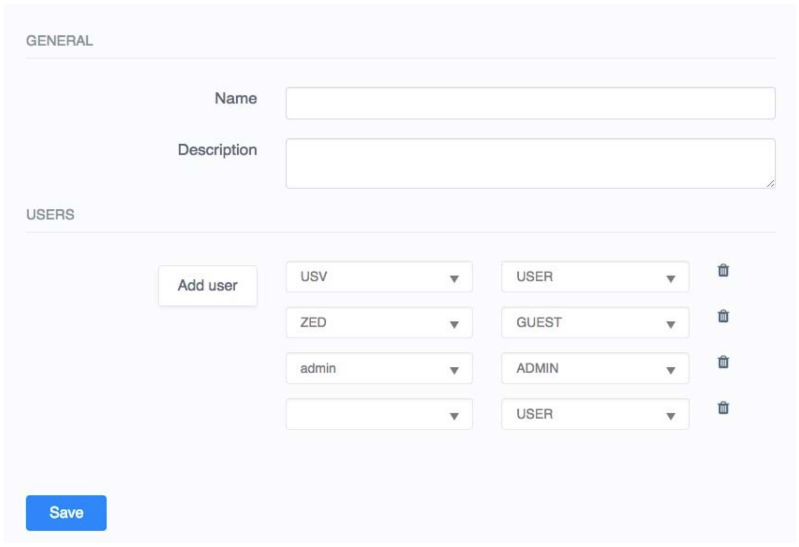
	PROJECT NAME ⇕	ID ⇕	ACTIONS
✓	JemanTest	fd49ed1c-94df-4e5b-85d7-eec017b7c444	Actions
	TI	e05b9aaa-87d7-44c9-bac8-ff8a44695746	Actions
✓	ZED	b1322086-b4af-4591-8cf1-7e6c2ccf9648	Actions
✓	TestProject1	b00a655b-1d3f-4b20-9584-c5f70a9a1d0f	Actions
	admin	8a5de7b8-a969-4df9-948d-b2d99e3f80d2	Actions
✓	USV	8009179d-a385-4a55-8395-7737151a4d2b	Actions
	Fraunhofer	7332347e-ee7d-40e0-b023-ff6a28bf8e70	Actions
	NAEVATEC	72342421-5ba5-4da6-8b32-d3a3dac20ed7	Actions
	VTOOLS	6490b3dd-5b90-4fe5-bdac-ae5250decdf2	Actions
	TUB	5f6d41e0-bb61-4783-995c-416b7bbb8b20	Actions
	VTT	5d1eaa37-b2d6-4efe-ba14-6ef3f34e12b3	Actions
	URJC	5bdd8ca0-a43d-4b5c-bd30-5f29f092f88b	Actions
	LiveU	4df48047-b805-4cf3-b2f0-3785995fef85	Actions

Figure 62 NUBOMEDIA PaaS GUI - Projects

The form for creating a new project is simple. There are only two small sections:

- General
  - Name
  - Description
- Users
  - User
  - User type (guest, user, admin)

We can add multiple users to a project and set their type from a drop-down list.



GENERAL

Name:

Description:

USERS

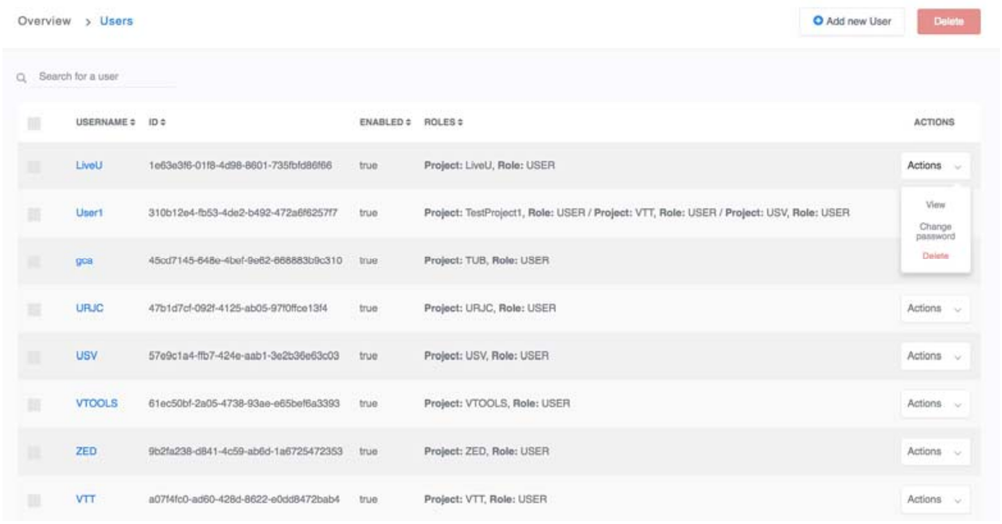
Add user	USV	USER
	ZED	GUEST
	admin	ADMIN
		USER

Figure 63 NUBOMEDIA PaaS GUI - Create new project

5.3 Users

There are three types of users on the NUBOMEDIA PaaS:

- **Guest** are users that can only see deployed applications.
- **Normal** users can login into their account and can access information related to their applications and projects. A normal user can't change his password, in order to achieve that he must send an email to NUBOMEDIA admin that will perform that action for him.
- **Admin** users have access to all views of the applications, can perform various application actions on different projects.



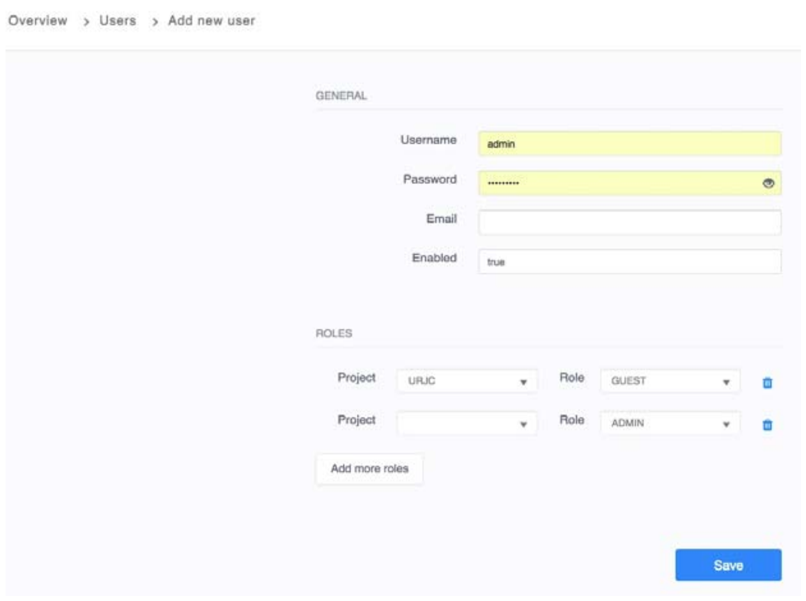
USERNAME	ID	ENABLED	ROLES	ACTIONS
LiveU	1e63e3f6-01f8-4d98-8601-735fbfd86f66	true	**Project:** LiveU, **Role:** USER	Actions
User1	310b12e4-fb53-4de2-b492-472a6f6257f7	true	**Project:** TestProject1, **Role:** USER / **Project:** VTT, **Role:** USER / **Project:** USV, **Role:** USER	
gca	45cd7145-648e-4bef-9e62-668883b9c310	true	**Project:** TUB, **Role:** USER	
URJC	47b1d7cf-092f-4125-ab05-97f0ffce13f4	true	**Project:** URJC, **Role:** USER	Actions
USV	57e9c1a4-ffb7-424e-aab1-3e2b36e63c03	true	**Project:** USV, **Role:** USER	Actions
VTOOLS	61ec50bf-2a05-4738-93ae-e65bef6a3393	true	**Project:** VTOOLS, **Role:** USER	Actions
ZED	9b2fa238-d841-4c59-ab6d-1a6725472353	true	**Project:** ZED, **Role:** USER	Actions
VTT	a07f4fc0-ad60-428d-8622-e0dd8472bab4	true	**Project:** VTT, **Role:** USER	Actions

Figure 64 NUBOMEDIA PaaS GUI - Users list

This view is can be accessed only by an admin user. Here, an admin is able to:

- Create a new user
- Delete an existing user
- Delete multiple users
- View user's details
- Change user's password

A specific user can be find using the search form. Also, the users can be filtered by username, id or roles.


Overview > Users > Add new user

GENERAL

Username	admin
Password	....
Email	
Enabled	true

ROLES

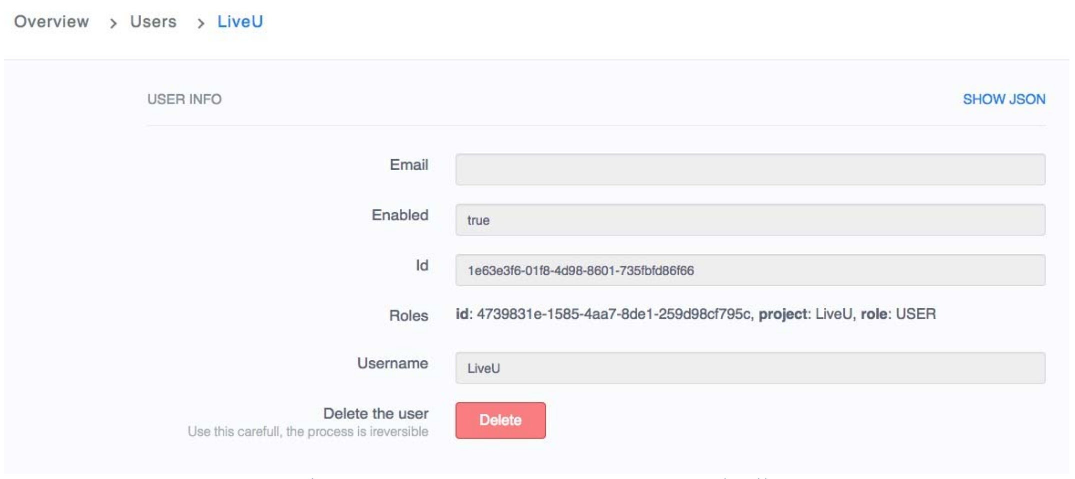
Project	URJC	Role	GUEST
Project		Role	ADMIN

Figure 65 NUBOMEDIA PaaS GUI - Create user form

When creating a new user, the admin has to complete the followings fields:

- Username
- Password
- Email
- Enable
- Roles

In user details view, the admin user can view specific user information, he can also delete the account or view it in JSON format.


Overview > Users > LiveU

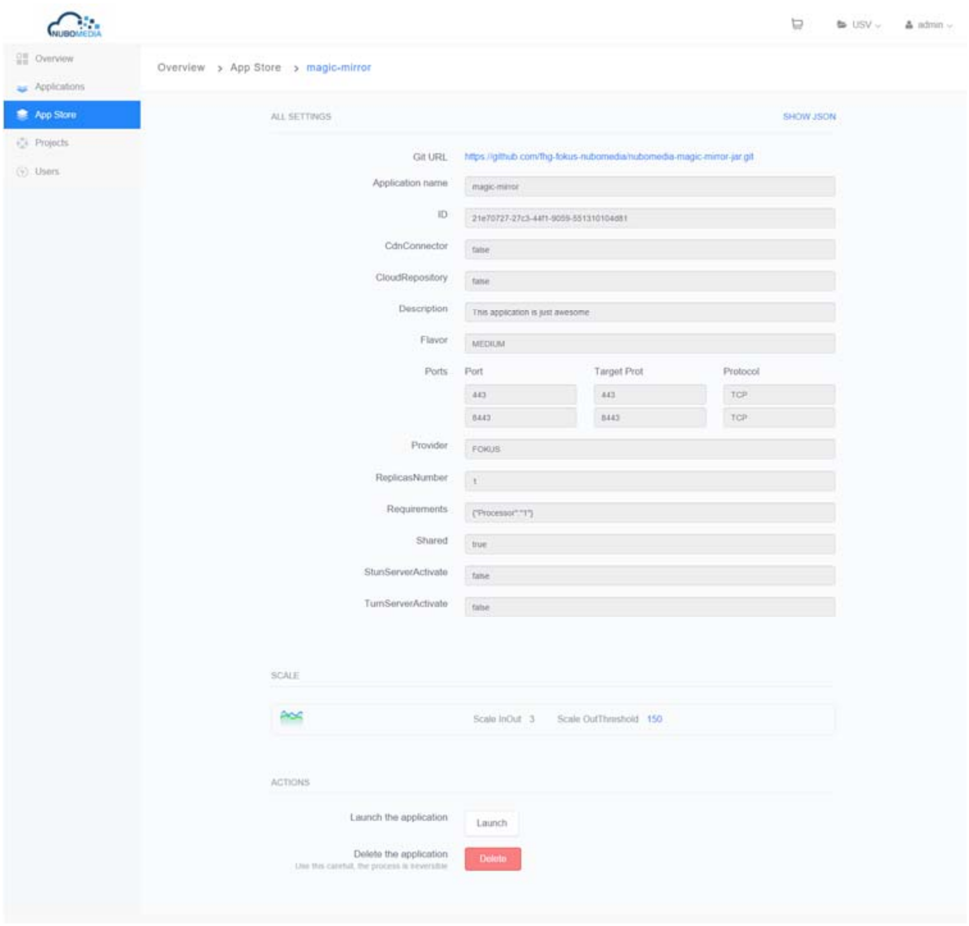
USER INFO

Email	
Enabled	true
Id	1e63e3f6-01f8-4d98-8601-735fbfd86f66
Roles	**id**: 4739831e-1585-4aa7-8de1-259d98cf795c, **project**: LiveU, **role**: USER
Username	LiveU
Delete the user Use this carefull, the process is ireversible	Delete

Figure 66 NUBOMEDIA PaaS GUI - User details

5.4 Marketplace

The NUBOMEDIA Market place is meant to gather a list of basic applications that any user on the platform is able to launch and test. Any logged in user is able to add a new application to the market by clicking on the Store app button top right side of the Store view.



Git URL	https://github.com/fhg-fokus-nubomedia/nubomedia-magic-mirror-jar.git
Application name	magic-mirror
ID	21e70727-27c3-44f1-9059-551310104d81
CdnConnector	false
CloudRepository	false
Description	This application is just awesome
Flavor	MEDIUM

Ports	Port	Target Prot	Protocol
	443	443	TCP
	8443	8443	TCP

Provider	FOKUS
ReplicasNumber	1
Requirements	{"Processor":"1"}
Shared	true
StunServerActivate	false
TurnServerActivate	false

SCALE

Scale InOut 3 Scale OutThreshold 150

ACTIONS

Figure 67 NUBOMEDIA PaaS GUI - Application Store

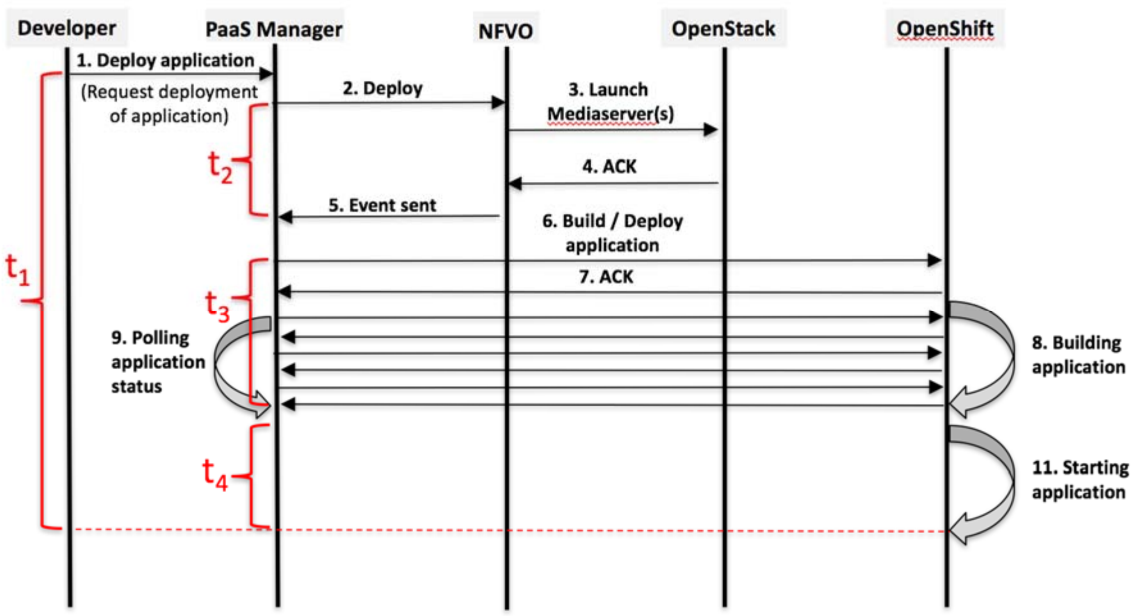
6 Evaluation results on the NUBOMEDIA testbed

In this section are presented the results of the evaluation of some operations executed by the NUBOMEDIA PaaS. Some sequence diagrams for the most important procedures of the NUBOMEDIA components are provided for simplifying the understanding the different steps executed. In particular, the deployment of applications and the scaling out procedure of media servers.

6.1 Deployment of an application

In this evaluation scenario, it is measured how much time it takes to deploy an application in different ways. As depicted by Sequence diagram 1 the deployment of an application involves several components and includes basically three main steps. Hence, three time intervals are considered in order to measure the latency of the management operations. In the end five measurements per test case were executed to build the average values of certain steps. In both cases, it is used the same application. Differences are only in the Dockerfile which defines the way of preparing and configuring the container for the application execution and the type of application where we distinguish between a source code version and a binary version.

Time interval t_2 depicts the time which is needed for allocating and configuring the virtual containers. Time interval t_3 covers the step where the PaaS Manager requests OpenShift for building and deploying the application. After the initial request, a polling mechanism starts in order to check the current application status. Interval t_3 ends once the application is in status RUNNING. The final execution of running the application, so that the application is also available for the consumer, depends on the type of application. Here we differentiate between applications where the source code must be compiled or an application which provides already a pre-compiled binary version. Using option 2 avoids the time consumption where the application must be compiled. Therefore, in the following the two different types of application are used in this evaluation part. This time interval is depicted by t_4. The entire time covering all three steps is t_2



Sequence diagram 1. Sequence diagram depicting time intervals measured

As depicted by Table 5 the overall time for deploying an application is around 5 minutes. Most of the time is consumed by building and deploying the application itself which requires the major time with 4:25 minutes. Minor time is needed for the deployment of the virtual resources on OpenStack. In fact, this takes around 29 seconds only whereas the start of the application takes even less with 9 seconds.

	Deploy virtual resources t_2	Building/deploying application t_3	Starting application t_4	**Total** **t_1**
1	00:25	04:25	00:07	**04:57**
2	00:30	04:25	00:14	**04:57**
3	00:30	04:25	00:10	**05:05**
4	00:30	04:20	00:05	**04:55**
5	00:30	04:30	00:10	**05:10**
avg	**00:29**	**04:25**	**00:09**	**05:03**

Table 5. Measurements of the deployment of an application where the code is present as binary

Table 6 depicts the measurements which were taken by deploying an application including also the compilation of its source code. Hence, the application must be compiled and this takes obviously more time than the case shown where the precompiled jar archive is already provided. In fact, this step lasts now 2:37 minutes in average. The build and deployment of the application takes 3:48 minutes. The time is reduced here due to the fact that less steps are required for preparing the image stream and containers. Finally, the source code version of the application requires around 7 minutes which is two minutes more than the binary version.

	Deploy virtual resources t_2	Building/deploying application t_3	Starting application t_4	**Total** **t_1**
1	00:30	03:35	02:42	**06:47**
2	00:35	03:55	02:53	**07:23**
3	00:30	03:55	02:42	**07:07**
4	00:40	03:40	02:07	**06:27**
5	00:35	03:55	02:44	**07:14**
avg	**00:34**	**03:48**	**02:37**	**06:59**

Table 6. Measurements of the deployment of an application where the code is present as source code

As shown, the deployment of an application depends strongly on the application provided in the meaning of how it is provided. The two main options provided are:

- The application source code is precompiled
- The application has to be compiled on demand when starting it

To summarize, deployment times may also vary and be improved based on different aspects. This may affect all the time intervals used before. The deployment of virtual resources depends on the cloud infrastructure. In our test cases, we used container based deployment of media server instances. For instance, using a hypervisor based deployment will increase the time needed for launching the virtual machines.

The results of t_3 might be affected by changing the base image used inside the containers running in OpenShift. In the given scenarios, prepared images were used which were already present in the local Docker registry of OpenShift. Other cases might be that the base image must be down/up-loaded and registered or even the creation of an image from the scratch.

As already seen, time interval t_4 depends on the way the application is started. It was shown that the precompiled version of the application required much less time for starting the application whereas the source code version spend a lot of time for compiling the application before executing it. But this is only one scenario where we used the same application. Other applications may take less or more time for compiling the source code or even for executing the application until it is available for the consumer.

6.2 Scaling out evaluation

In the following it is evaluated the needed time for scaling out new components. In the first scenario, it is shown how long it takes to scale out components without using the proposed pool mechanism whereas the section after evaluates the times which are needed by using the pool mechanism.

As depicted by both figures (see Figure 69 and Figure 69) the detection of the need to scale varies from 56ms to 78ms, where the gap is basically caused by the response time of the Monitoring System. The next step of requesting the Decision-maker is around 27ms and almost the same for both. The main difference comes when executing the scaling actions. Obviously, the reason is the number of components to be scaled-out. Scaling-out a single component takes 16570ms whereas scaling-out five components take 83372ms. In fact, when scaling-out 5 components in a row, each scaling-out operation takes around 16674ms in average.

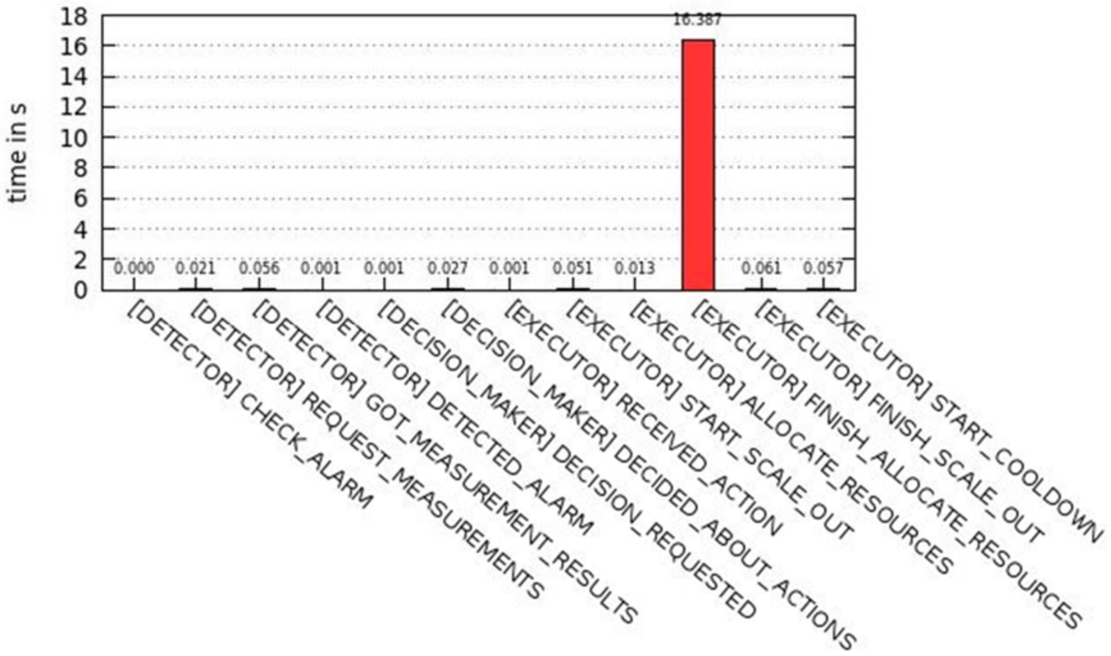


Figure 68. Timings scaling-out 1 instance without pool mechanism

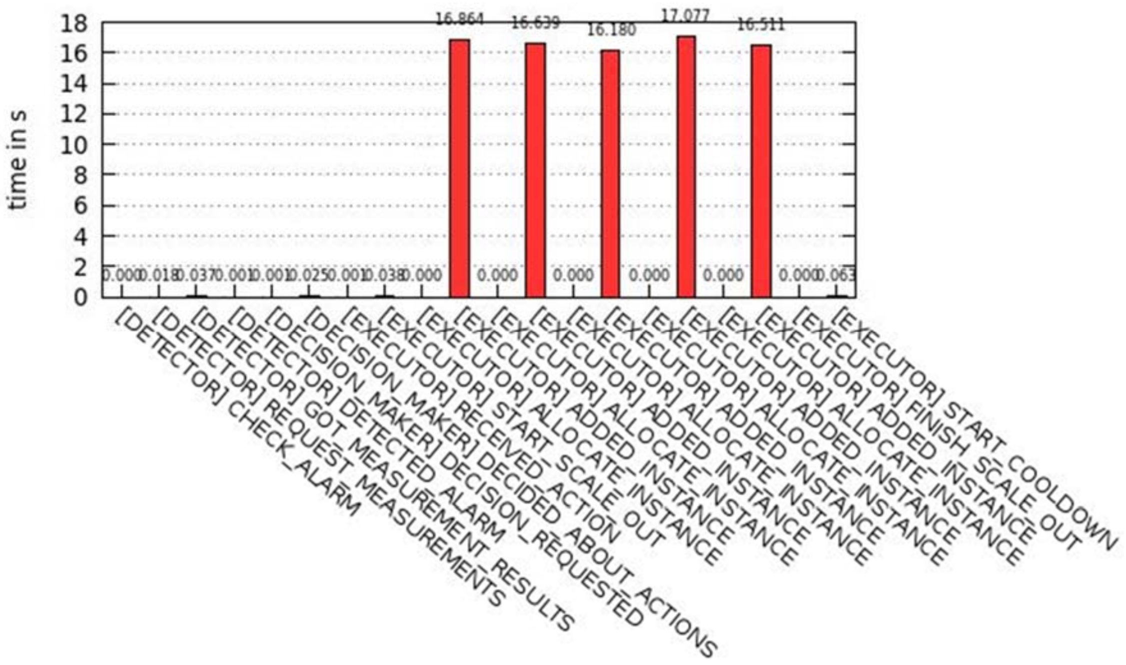


Figure 69. Timings scaling-out 5 instances without pool mechanism

Concluding, in this approach without the pool mechanism the time of detecting the need to scale and deciding about actions is very short. Both operations together need around 100ms whereas the scaling operation itself needs roughly 16000ms to 17000ms for scaling-out a single component. In case of scaling-out a single component it takes 99,36% of the overall time from detecting to finish scaling. In the scenario of scaling-out five components in a row the scaling takes even 99,9% of the entire time. As seen, scaling-out five components in a row needs five times more and is caused by scaling-out step-by-step. Obviously, the time needed for scaling-out 5 components can be improved by parallelizing the scaling operations. Another promising approach is the usage of the proposed pool mechanism that is evaluated in the next section.

The following section covers the evaluation of the proposed pool mechanism. Two scenarios were taken into account equally to the two scenarios described before. One test covers the scaling-out of a single component and the other scenario scales out five components step-by-step. Mainly, the pool mechanism was introduced to decrease the required time needed for launching new components. As seen in the previous section the time for launching new components takes most of the time of the entire scaling process from detecting need to scale, over making scaling decision and finally the scaling operation itself, and therefore, it decreases the reaction time of the AutoScaling System dramatically in order to provide new instances on demand.

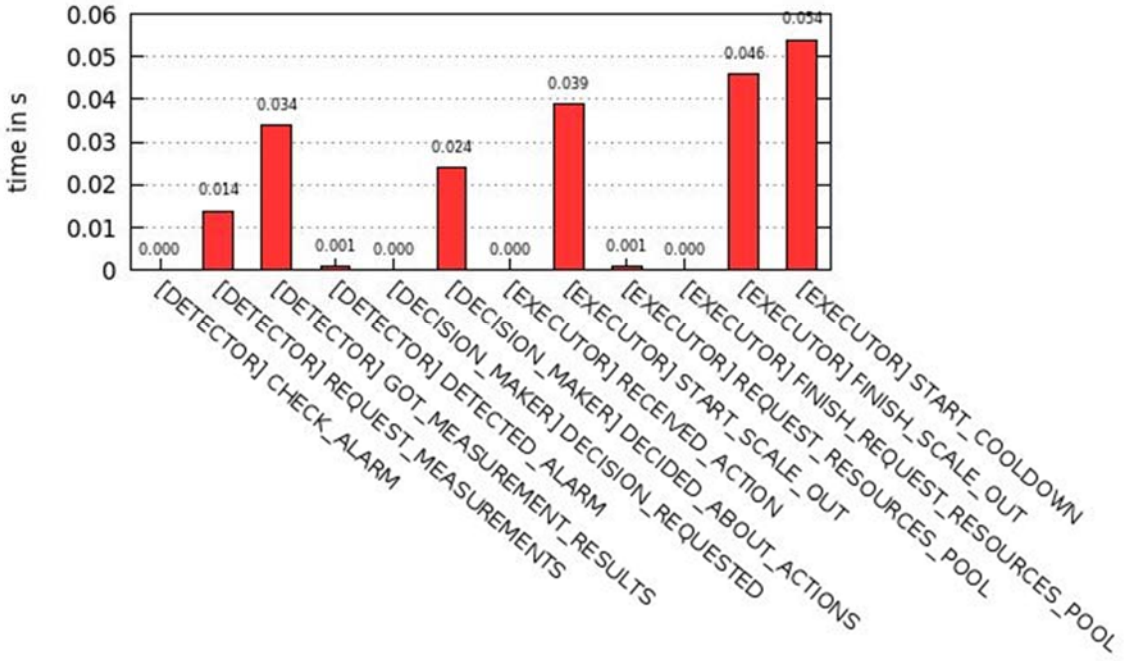


Figure 70. Timings scaling-out 1 instance using pool mechanism

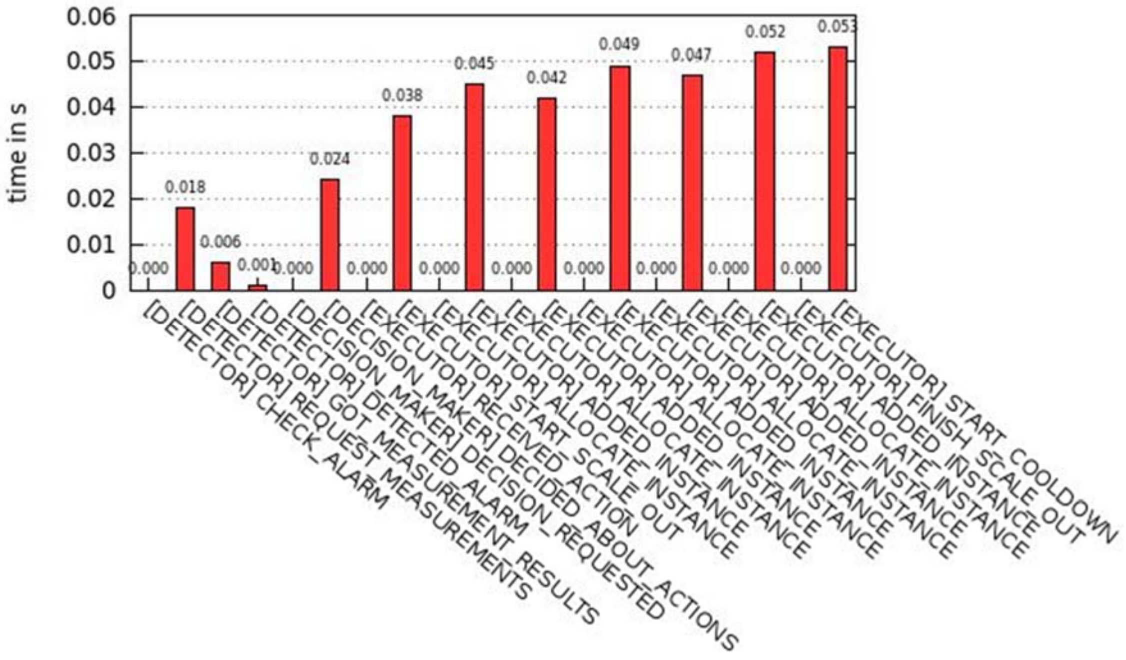


Figure 71. Timings scaling-out 5 instances using pool mechanism

Figure 70 shows the measurements when scaling-out a single component and Figure 71 shows the case of scaling-out five components. Similar to the scenario before without the pool mechanism, in both scenarios the time of detecting and decision-making takes around 24ms. But in contrast to the previous test the Executor does not request the VIM for allocating new resources on demand but it requests the Pool Manager in order to get a prepared VNF component. As depicted in Figure 70 in case of adding a single component, the whole scaling operation needs 140ms. Figure 71 reveals that scaling-out five components in a row takes 350ms only.

Compared with the time needed from detecting the need to scale to finish the scaling, the scaling procedure itself in case of adding a single component takes 140ms and 65,7% of the time whereas scaling-out five components take 350ms and 87,93%.

Considering the time needed in the scenarios without the pool mechanism, scaling-out a single component and five components in a row is improved dramatically. In fact, the scaling-out action to add a new component takes this time 0,85% of the time that was needed without the pool mechanism. Scaling-out five components take this time 0,42% only compared with the time in the previous scenario. Additionally, after each scaling operation the VNFR is updated by requesting the NFVO and this time is also included in each scaling action that takes most of the time.

7 Conclusion

This deliverable presented the overall design and implementation of the NUBOMEDIA Cloud Platform, based on the requirements gathered from the developers' community. An innovative architecture satisfying most of the requirements have been proposed. Its implementation based on state-of-the-art open source technologies, provides one of the unique Platform as a Service for multimedia applications.

Major changes have been applied mainly in the context of the NUBOMEDIA PaaS Manager, being the entry point for application developers. A new version of its API has been developed and implemented, with also a new refactoring of the PaaS GUI exposing more functionalities for controlling the lifecycle of the application.

The good evaluation results obtained also demonstrates that the design decisions were correctly taken, and application developers can benefit from the NUBOMEDIA PaaS for the management of their application lifecycle. They can deploy and expose to end users a complete elastic application without requiring any knowledge on the infrastructure itself.

Annex 1. Open Baton - Network Service Descriptor JSON file example

The JSON file below represents the NSD created for instantiating the Media Plane components via the NFVO. As already mentioned in the previous sections, the NSD is generated dynamically by the PaaS-Manager and on boarded on the NFVO whenever a new Application instantiation request is received.

```
{
  "vendor":"NUBOMEDIA",
  "name":"MS-NSD",
  "version":"0.1",
  "vnfd":[
    {
      "vendor":"TUB",
      "version":"0.1",
      "name":"media-server-vnf-1",
      "type":"media-server",
      "endpoint":"media-server",
      "vdu":[
        {
          "vm_image":[
            "nubomedia/kurento-media-server"
          ],
          "vimInstanceName":"nubomedia-vim",
          "scale_in_out":3,
          "vnfc":[
            {
              "connection_point":[
                {
                  "floatingIp":"random",
                "virtual_link_reference":"internal_nubomedia"
                }
              ]
            }
          ]
        }
      ],
      "auto_scale_policy": [
        {
          "name":"scale-out",
          "threshold":100,
          "comparisonOperator":">=",
          "period":30,
          "cooldown":60,
          "mode":"REACTIVE",
          "type":"VOTED",
          "alarms": [
            {
              "metric":"CONSUMED_CAPACITY",
              "statistic":"avg",
              "comparisonOperator":">=",
              "threshold":70,
              "weight":1
            }
          ],
          "actions": [
            {
              "type":"SCALE_OUT",
              "value":"2"
            }
          ]
        },
        {
          "name":"scale-in",
          "threshold":100,
          "comparisonOperator":">=",
          "period":30,
          "cooldown":60,
          "mode":"REACTIVE",
          "type":"VOTED",
          "alarms": [
            {
              "metric":"CONSUMED_CAPACITY",
              "statistic":"avg",
              "comparisonOperator":"<=",
              "threshold":30,
              "weight":1
            }
          ],
          "actions": [
            {
              "type":"SCALE_IN",
              "value":"2"
            }
          ]
        }
      ],
      "virtual_link":[
        {
          "name":"internal_nubomedia"
        }
      ],
      "lifecycle_event":[
        {
          "event":"ALLOCATE",
          "lifecycle_events":[
            "allocation of vdu"
          ]
        },
        {
          "event":"RELEASE",
          "lifecycle_events":[
            "release of vdu"
          ]
        }
      ],
      "deployment_flavour":[
        {
          "flavour_key":"d1.medium"
        }
      ],
      "vnfPackageLocation":""
    },
    {
      "vendor":"TUB",
      "version":"0.1",
      "name":"mongodb",
      "type":"mongodb",
      "endpoint":"generic",
      "configurations":{
        "name":"config_config",
        "configurationParameters":[
          {
            "confKey":"PORT",
            "value":"27018"
          },
          {
            "confKey":"SMALLFILES",
            "value":"true"
          }
        ]
      },
      "vdu":[
        {
          "vm_image":[
            "ubuntu-cloud-64bit"
          ],
          "vimInstanceName":"nubomedia-vim",
          "scale_in_out":4,
          "vnfc":[
            {
              "connection_point":[
                {
                  "floatingIp":"random",
                  "virtual_link_reference":"internal_nubomedia"
                }
              ]
            }
          ]
        }
      ],
      "virtual_link":[
        {
          "name":"internal_nubomedia"
        }
      ],
      "lifecycle_event":[
        {
          "event":"INSTANTIATE",
          "lifecycle_events":[
            "install.sh"
          ]
        },
        {
          "event":"START",
          "lifecycle_events":[
            "start-single-mongo.sh"
          ]
        }
      ],
      "deployment_flavour":[
        {
          "flavour_key":"m1.small"
        }
      ],
      "vnfPackageLocation":"https://github.com/tub-nubomedia/cloud-
repository-scripts.git"
    }
  ],
  "vld":[
    {
      "name":"internal_nubomedia"
    }
  ],
  "vnf_dependency":[
  ]
}
```

References

[1] Network Function Virtualization Management and Orchestration. (2014). Retrieved December 14, 2015 from http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf

[2] D2.4.2 NUBOMEDIA Architecture v2

[3] D2.4.3 NUBOMEDIA Architecture v3

[4] Open Baton: An open source Network Function Virtualisation Orchestrator (NFVO) fully compliant with the ETSI NFV MANO specification http://openbaton.github.io/

[5] D2.2.2: State-of-the-art revision document v2

[6] Openstack: Open source software for creating public and private clouds. See http://www.openstack.org/

[7] D3.3.1: NUBOMEDIA Cross-Layer Connectivity Manager v1, https://www.nubomedia.eu/sites/default/deliverables/WP3/D3.3.1_CrossLayerConnectivityManager_V1.0_27-01-2015_FINAL-PC.pdf

[8] https://www.openstack.org/assets/pdf-downloads/Containers-and-OpenStack.pdf

[9] Graphite: An open source graphing and time-series database for storing realtime metrics https://graphite.readthedocs.org/en/latest/

[10] Backstop is an open source REST API to submit data to Graphite https://github.com/obfuscurity/backstop

[11] Logstash is an opensource tool to centralize and process logs https://www.elastic.co/products/logstash

[12] Icinga is an opensource tool based on Nagios and improved by open source community for monitoring systems https://www.icinga.org

[13] D3.2.1: Cloud Repository

[14] http://cloudinit.readthedocs.org/en/latest/

[15] https://github.com/tub-nubomedia/cloud-repository-scripts

[16] https://docs.mongodb.org/ecosystem/drivers/

[17] Sefraoui, O.; Aissaoui, M.; Eleuldj, M., Management platform for Cloud Computing, International Conference on Technologies and Applications (CloudTech), (2015):1-5.

[18] ETSI TR 103 126 V1.1.1 (2012-11), CLOUD; Cloud private-sector user recommendations, Technical Report (TR), ETSI Technical Committee CLOUD (CLOUD), (2012).