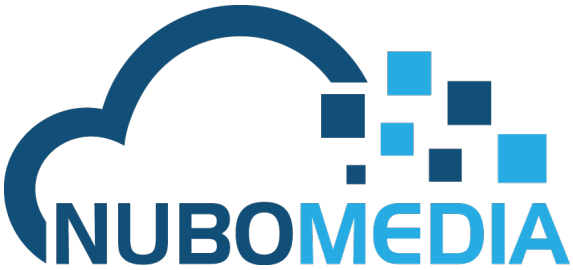
D3.2	
Version	1.0
Author	TUB
Dissemination	PU
Date	27/01/2015
Status	Final



D3.2 Cloud Platform v2

Project acronym:	NUBOMEDIA
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2016-09-30
Project type:	STREP
Project reference:	610576
Project web page:	http://www.nubomedia.eu
Work package	WP3 Cloud Platform
WP leader	Giuseppe Carella
Deliverable nature:	Prototype
Lead editor:	Giuseppe Carella
Planned delivery date	01/2016
Actual delivery date	30/01/2016
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)
Luis Lopez (URJC)
Carlo Maiorano (FOKUS)
Michael Pauls (TUB)
Cristian Spoiala (USV)
Lorenzo Tomasini (TUB)

Internal Reviewer(s):

Alin Calinciuc (USV)

Version History

Version	Date	Authors	Comments
0.1	10.2015	Giuseppe Carella	Initial version
0.2	11.2015	All	Added initial description and structure of the different sections
0.3	12.2015	All	Added final contributions in almost all sections
0.4	01.2016	Alin Calinciuc	Final contributions added and review
1.0	01.2016	Giuseppe Carella	Final version of the document

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 contributions from Release 4,5, and 6 and constitutes Deliverable 3.2, providing the details about the Cloud Platform components and their integration from WP3 perspective. It reports on the final architectural changes and key findings when implementing the Cloud Platform foundations. Furthermore, it presents some initial results from the virtualisation of Media Elements.

It includes all the details about the Cloud Platform components and how they can be combined together for building a PaaS for Multimedia Applications. To summarize, this deliverable will show the final findings on the major components making up the infrastructure foundation for a Cloud Platform for Multimedia Applications:

- While dealing with Real-Time Multimedia Applications, it is important that the Infrastructure supports the required capabilities, and provides low delay and high computational resources. Cloudification, should have a minimal impact on the QoE perceived by the final users.
- A Platform as a Service (PaaS) solution have been designed and integrated. It allows the deployment and provisioning of several type of Applications. It allows developers to benefit from scalability and elasticity mechanisms provided at each level.
- 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.
- Networking aspects on typical cloud datacenters have been analyzed and considered for ensuring the required level of SLAs expected by Application developers. In particular, a new set of components have been introduced on the Cloud infrastructure leveraging state-of-the-art Software Defined Networking (SDN) capabilities.
- 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, which together offer 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 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 given a recap of the NUBOMEDIA Architecture as already presented in D2.4.2 [2], and some more details about the functionalities provided by the NUBOMEDIA Cloud Platform.

2.1 Functional Architecture Overview

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 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 Infrastructure as a Services (IaaS) composed by the infrastructure resources in terms of Compute Nodes (CN) and the Virtual Infrastructure Manager (VIM) providing the compute, storage and networking resources to the upper layers

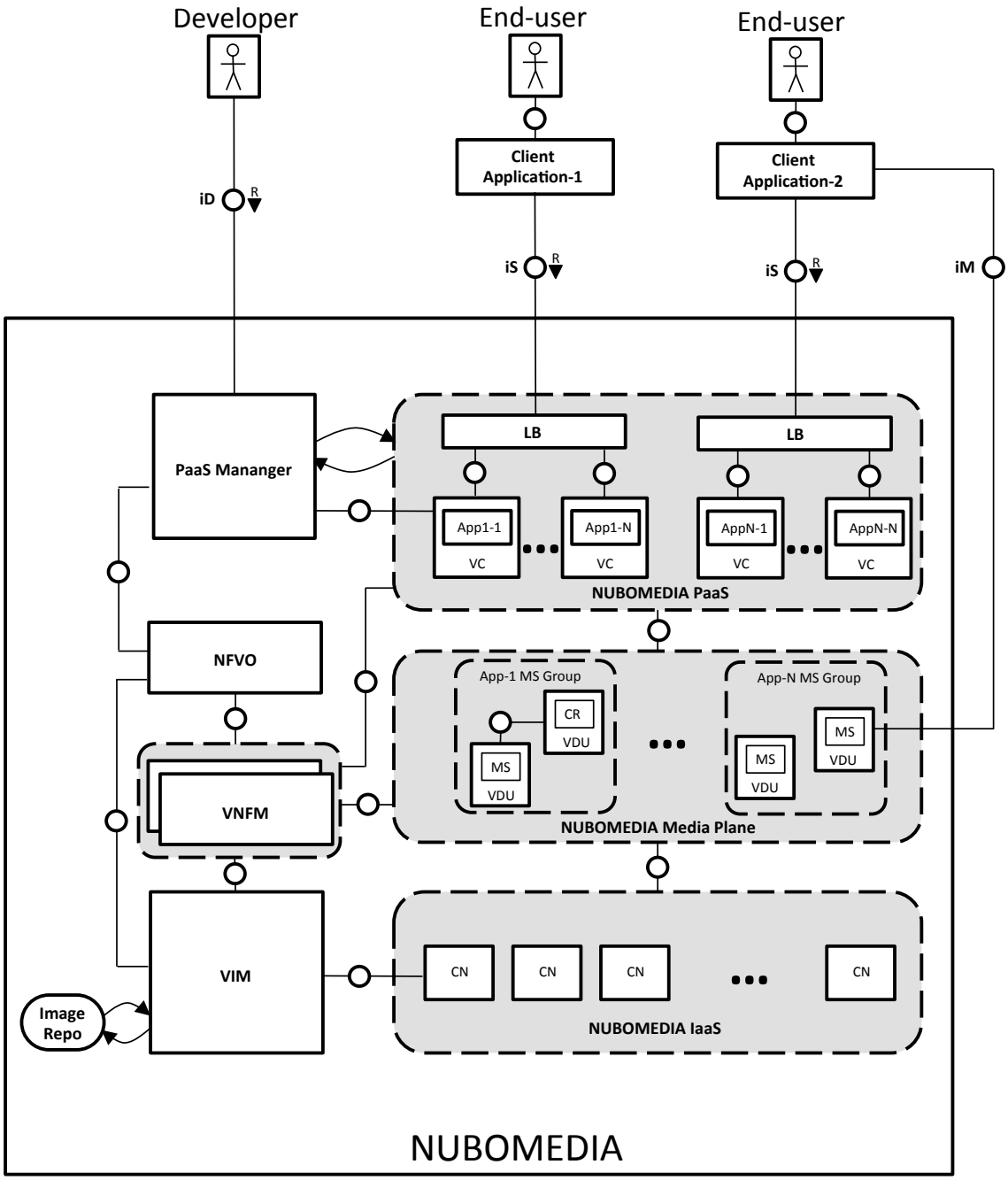


Figure 1. NUBOMEDIA Functional Architecture

In particular, WP3 focuses on the Management and Orchestration functionalities at the three different layers, while Media Functions and Applications are out of scope of WP3 and will be described in other WPs.

3 Software Architecture

In this section it will be shown the Software Architecture of the NUBOMEDIA project. In particular, the main focus will be on how the functional elements have been implemented and how they interact with each other.

3.1 Software Architecture Overview

In order to maintain the same multi layered architectural approach, it has been necessary to identify different Software Components which could be mapped on the particular functional elements. In particular, each of the layer consists of a mix between existing open source activities, extensions to those existing tools, and completely newly implemented ones.

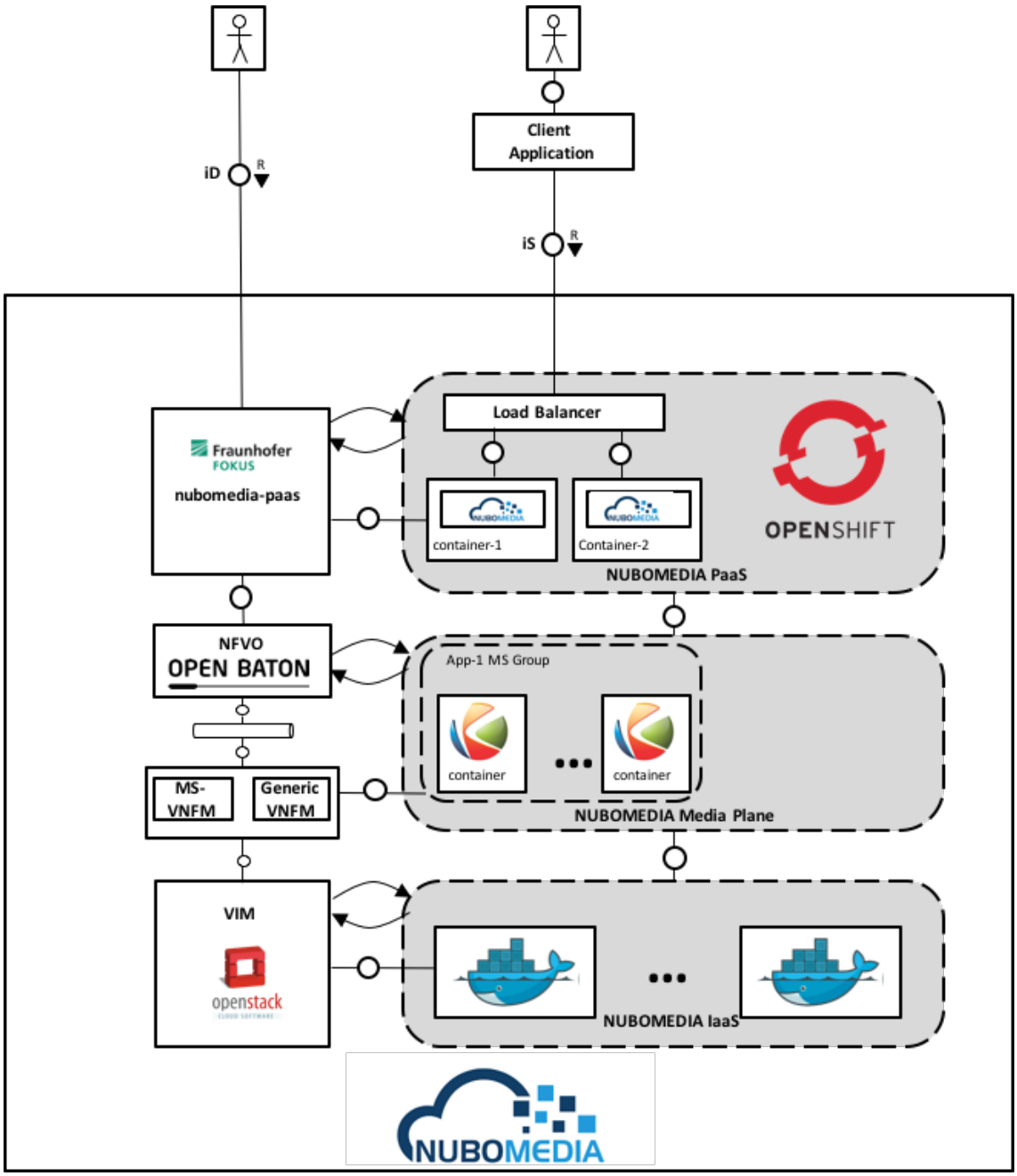


Figure 2. NUBOMEDIA Software Architecture

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 the only fully compliant NFV Orchestrator (NFVO) [4].In order to manage the Media Server a new VNFM (MS-VNFM) 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

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 (64 bits) as Operating System to run[1], 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 64 bits 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

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

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.

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.

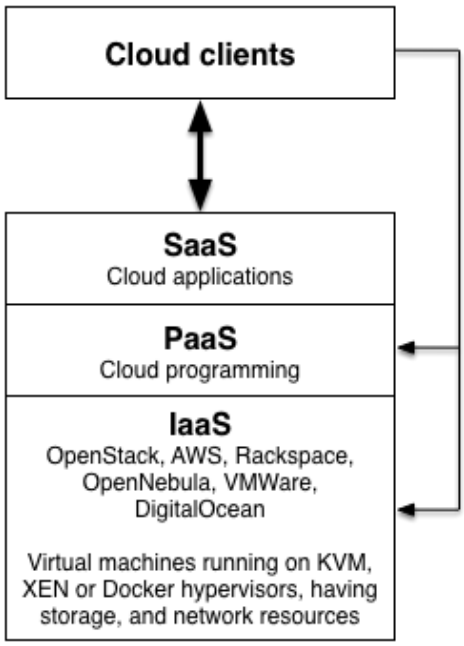


Figure 3. Relationship between clients and SaaS/PaaS/IaaS systems

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 4)

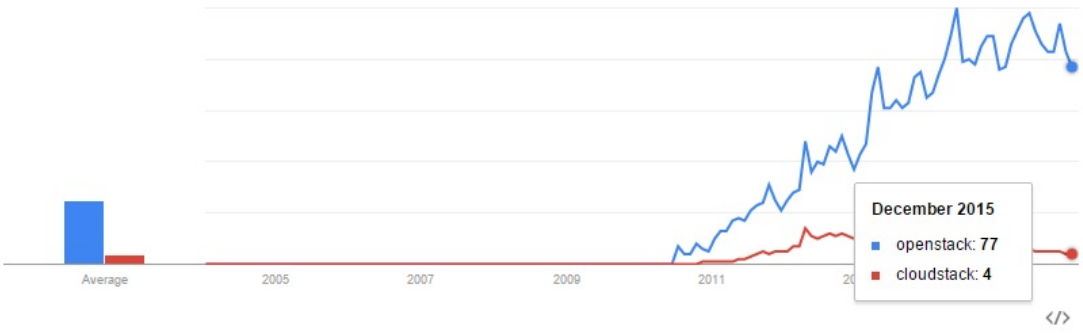


Figure 4. Google trends for OpenStack vs CloudStack

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 a 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[2]. 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 5).

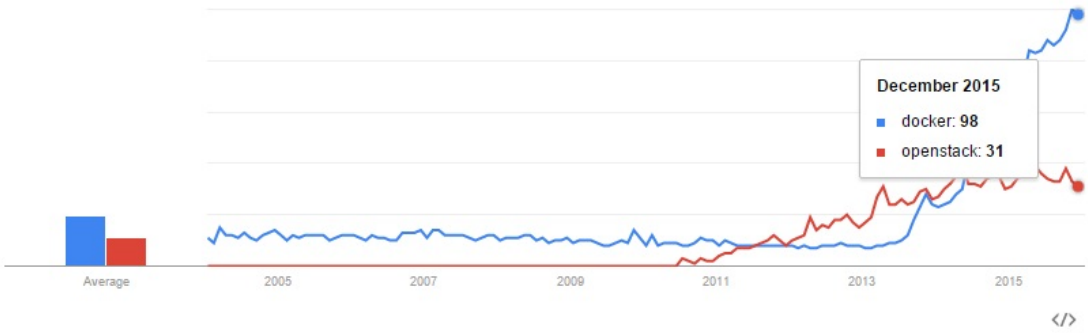


Figure 5. Docker vs OpenStack

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

[2] 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[3]

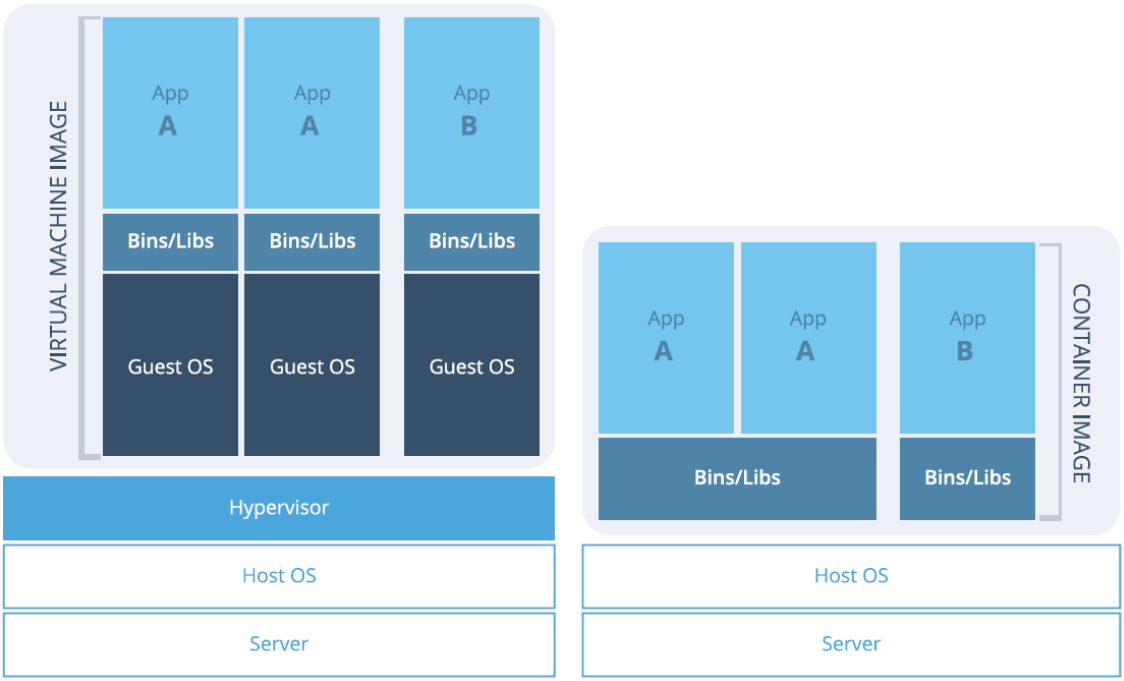


Figure 6. Docker vs KVM

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. Figure 2 shows how Magnum integrates with other OpenStack components.
- 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.
- 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-

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

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.[4]

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 loose all the data when you restart a container.
Considering the advantages and disadvantages of containers for this release 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 work
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.

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

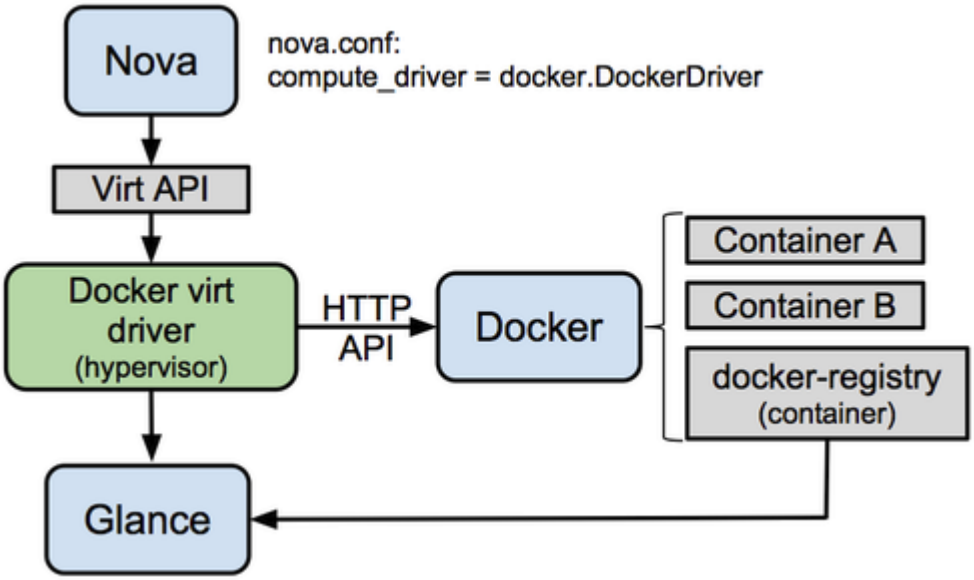


Figure 7. Nova-docker architecture

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[5] . 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 takes into consideration the time needed for the Ubuntu to actually run the initrd and userspace.
- Encoding with a video using x264 H.264/AVC, using just the CPU (with OpenCL support deactivated)

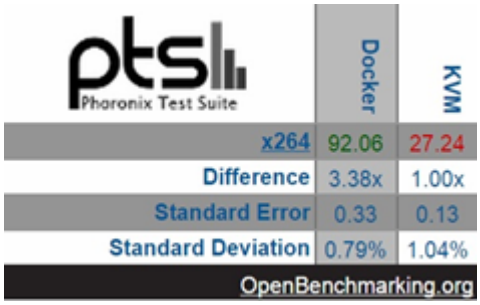


Figure 8. Encoding performance on Docker vs KVM hypervisor

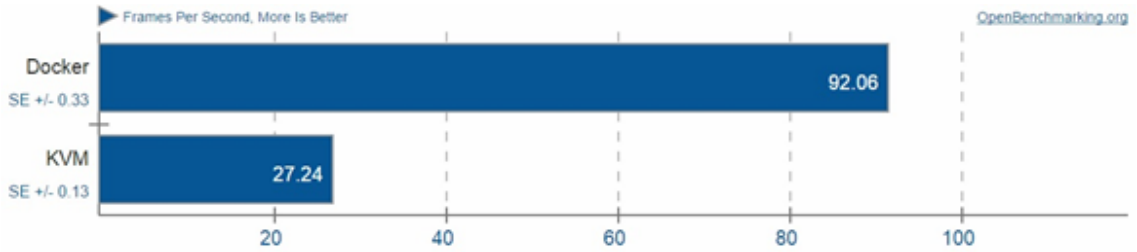


Figure 9 Encoding performance on Docker vs KVM hypervisor 2

[5] 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.

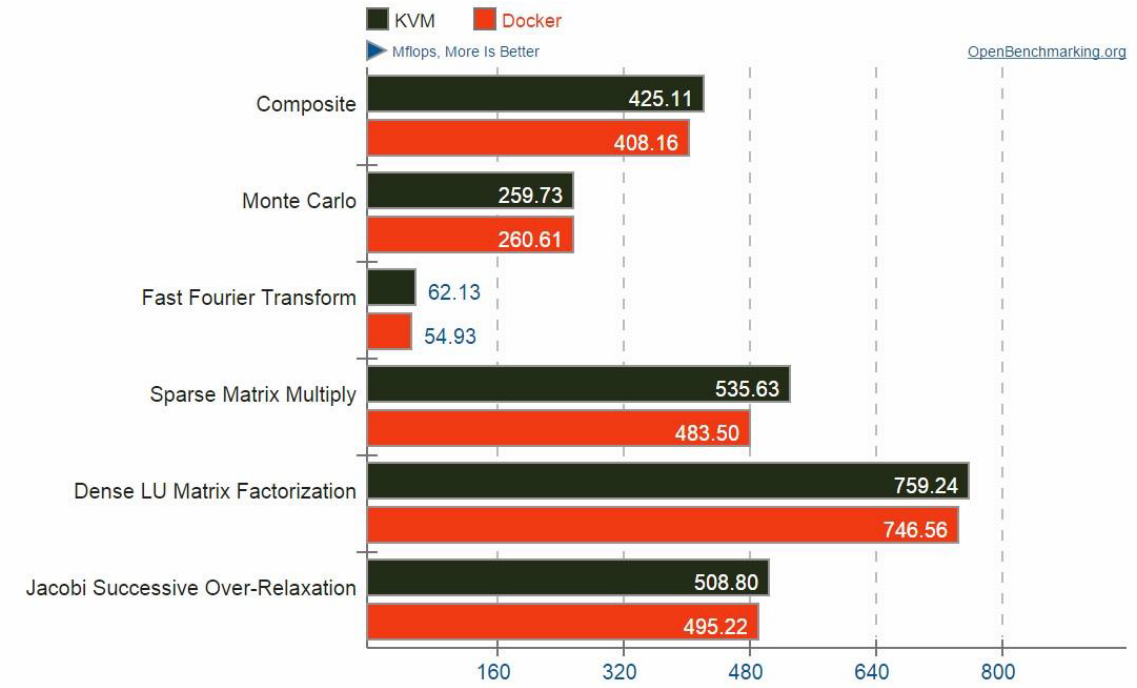


Figure 10 Docker vs KVM running scientific and numerical computation

For memory testing we used ramspeed and stream.

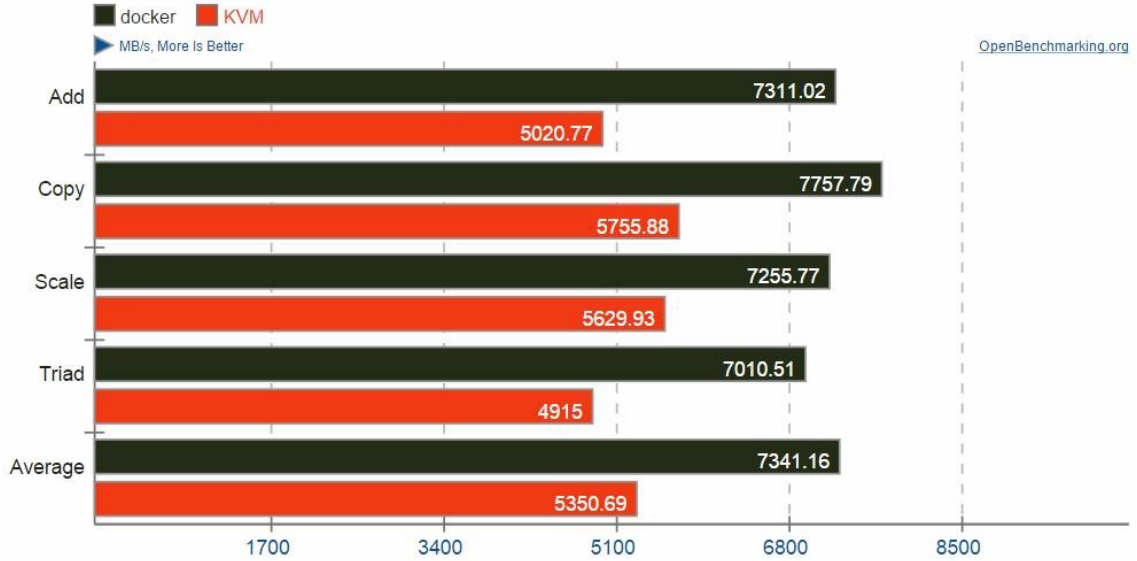


Figure 11. Ramspeed memory tests

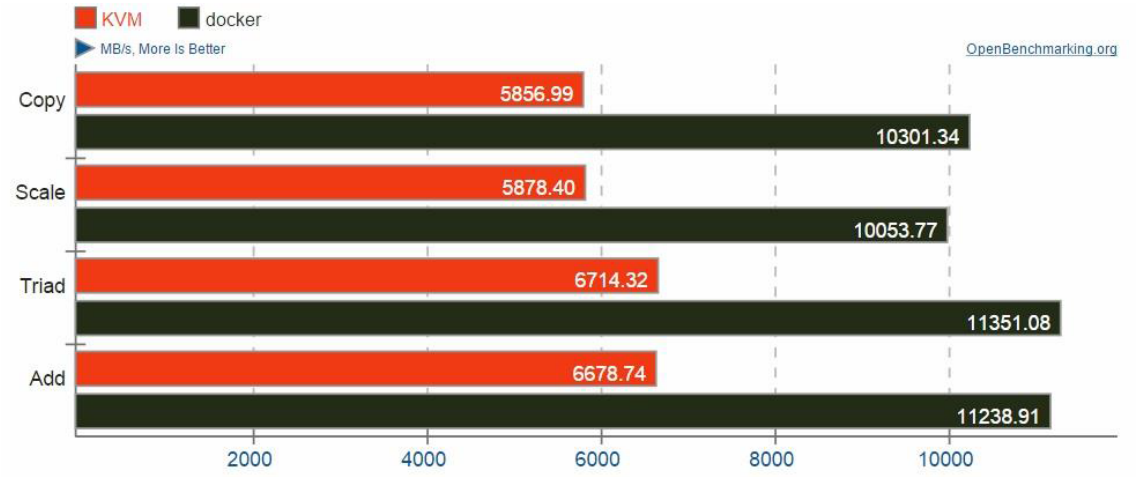


Figure 12. Stream memory tests

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.

For this release 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[6] 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 networking at the same time. For this reason we decided to use RDO (Red Hat OpenStack).

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

- 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

In this release 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 it's 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 Monitoring 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, explained in more details on 3.2.5.3

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[7] 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.

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.

[7] https://docs.google.com/document/d/1cMdPQ0sb6ENLNu1X4x98SSXAOUfZLsVcHB-5iGKYO7Q/edit

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.

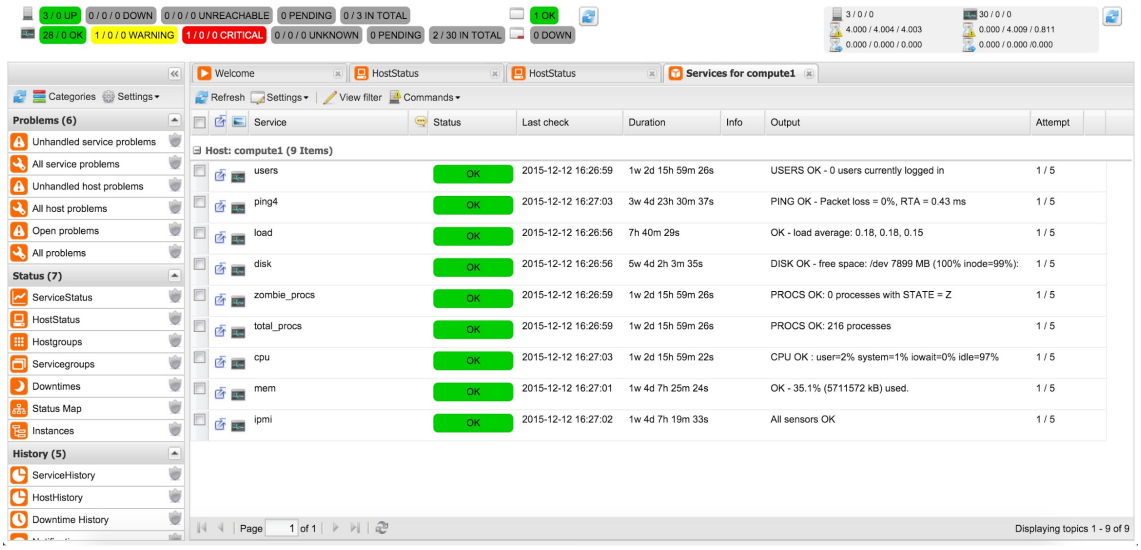


Figure 13. Icinga dashboard displaying status of a host OpenStack compute

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

Using the information provided by the monitoring tools like Icinga, NUBOMEDIA operator 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 the 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 of time 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.

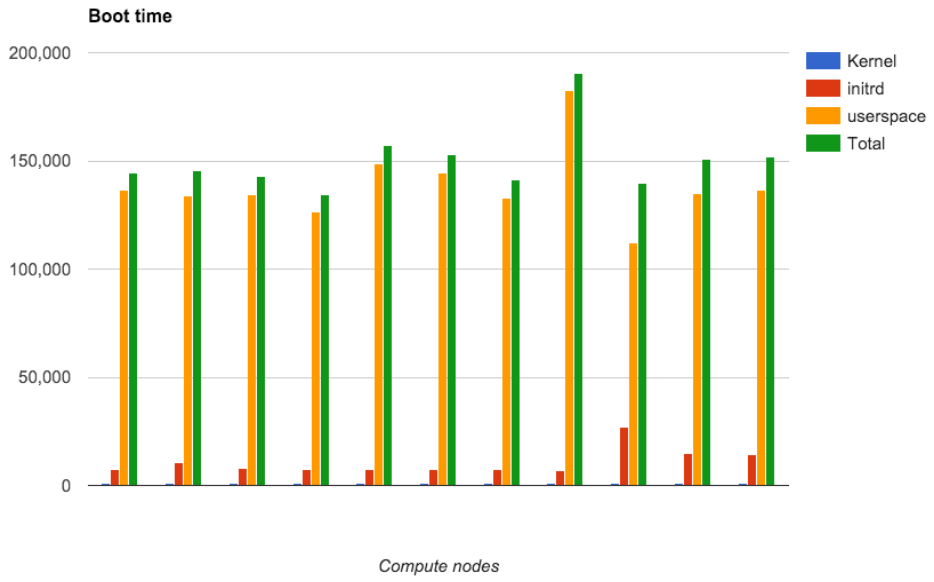


Figure 14. Time to boot a physical machine

3.2.5.5 Overall architecture

All these monitoring tools are converged into a single configuration and on Figure 16 is highlighted 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.

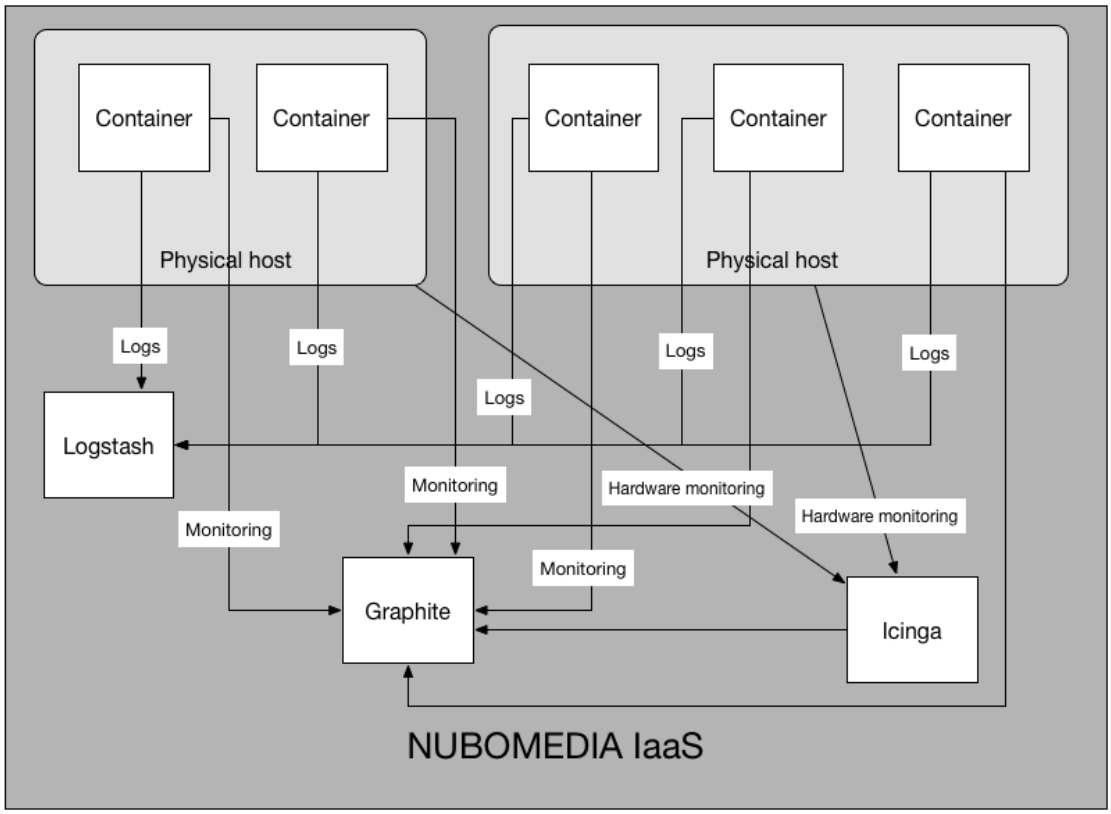


Figure 15. Monitoring integration on the NUBOMEDIA IaaS

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

Open Baton 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 VNF Manager
- Element Management System (EMS)
- Java SDKs for VNF Managers and plugins
- Dashboard

Additionally, it was necessary to implement a new VNFM specific for the management of the Media Server functions. This VNFM contains also all the logic described in the previous deliverables as part of the Elastic Media Manager (EMM).

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 (NFVO-API) 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 NFVO-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.

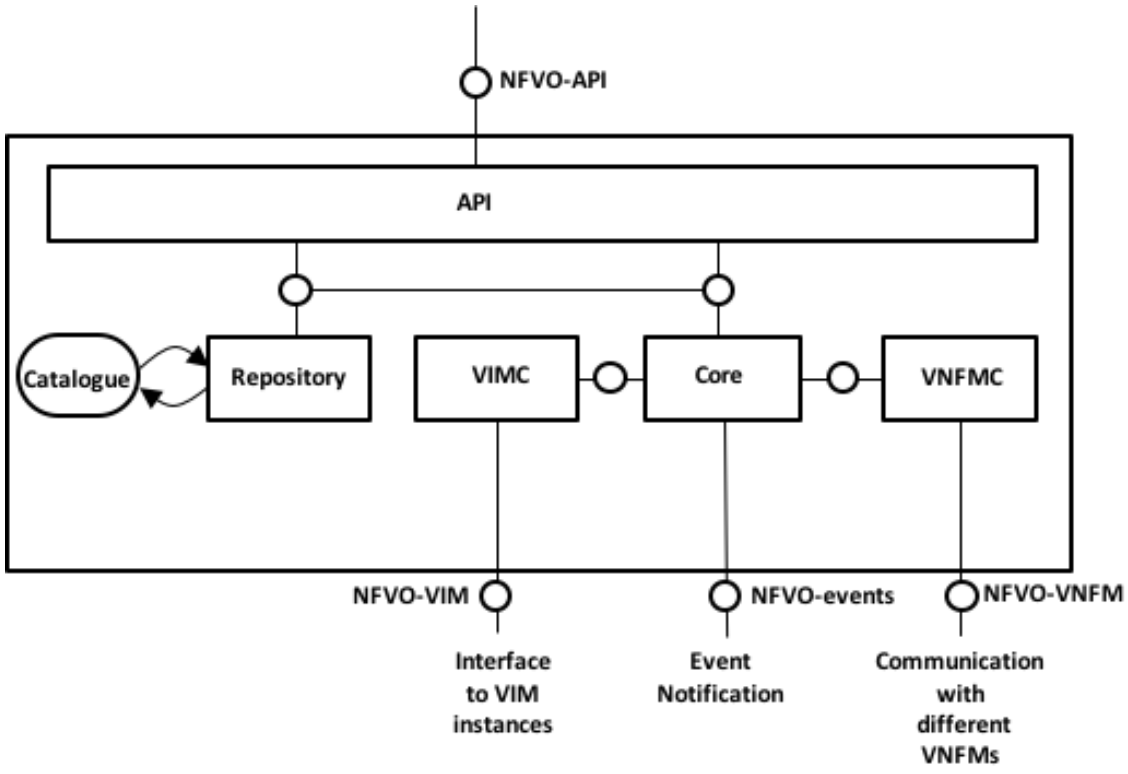


Figure 16. NFVO Functional Architecture

The NFVO part of the Open Baton is implemented in Java on top of the Spring Framework. Figure 16 shows the internal architecture of the NFVO. It is composed by different modules:

- API: is the component exposing the NFVO-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.
- 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. The messaging system protocol supported is the Advanced Message Queuing Protocol (AMQP), in particular its implementation chosen is RabbitMQ.

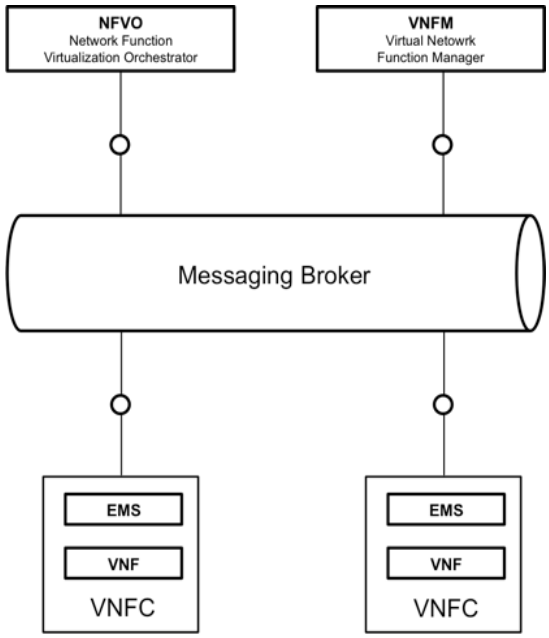


Figure 17. NFVO, VNFM and EMS interactions

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. A plugin in Open Baton is a RMI server implementation that connects

to the RMI registry spawned by the NFVO providing a specific implementation for a generic interface.
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 Cloud Repository

Based on the decisions previously taken in the deliverable D3.2.1 [14], 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 sharded architecture (Figure 18).

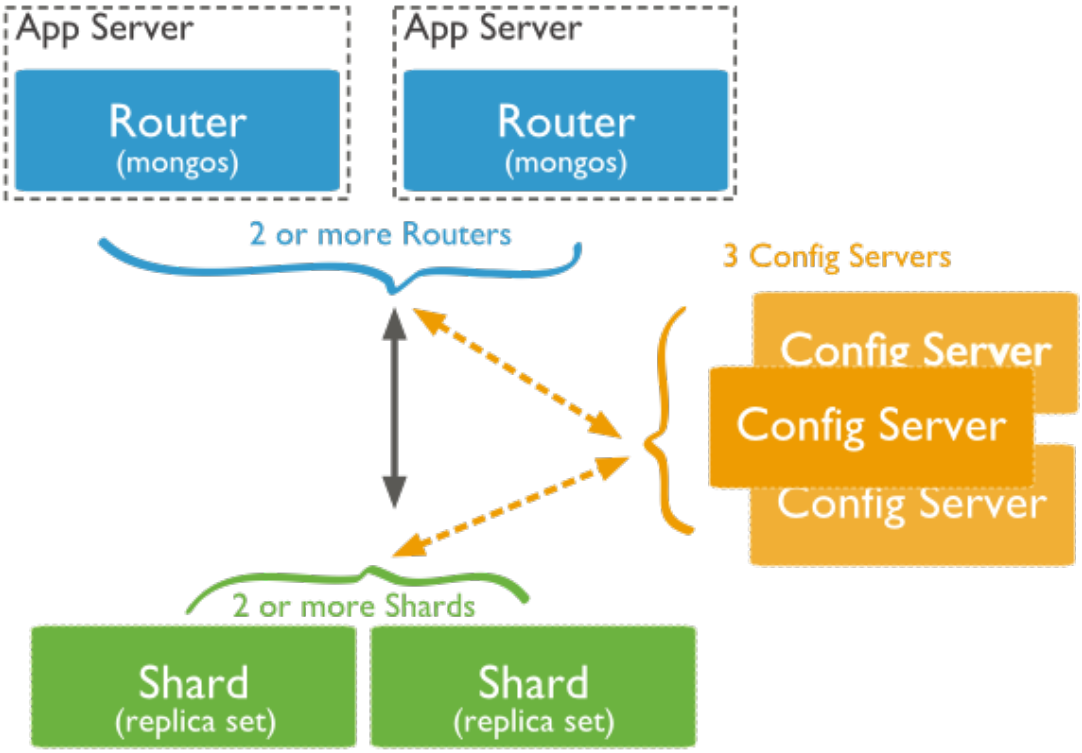


Figure 18. MongoDB sharded architecture

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.

3.3.3 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.

3.3.4 Generic VNFM

OpenBaton 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 [15]. 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).

3.3.5 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.

3.3.6 Media Server VNFM

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).

3.3.6.1 Elastic Media Manager

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.

3.3.6.2 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 loosing 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.

3.3.6.3 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.

3.3.6.4 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.

Table 1. EMM REST API

Method	URL	Description
POST	/vnfr/<vnfr_id>/app	Registers a new App to the VNFR with this <vnfr_id>
GET	/vnfr/<vnfr_id>/app	Returns a list with detailed information about all apps registered to the VNFR with this <vnfr_id>
GET	/vnfr/<vnfr_id>/app/<app_id>	Returns detailed information about the App with this <app_id> registered to the VNFR with this

		<vnfr_id>
DELETE	/vnfr/<vnfr_id>/app/<app_id>	Unregisters an App with this <app_id> registered to the VNFR with this <vnfr_id>.
GET	/vnfr	Returns a list of VNFRs managed by the EMM

3.3.6.4.1 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 an 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.

3.3.6.4.2 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.

3.3.6.4.3 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.

3.3.6.4.4 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>.

3.3.6.4.5 Get details about all applications managed by the EMM

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

3.3.6.5 Pools of Media Components

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.

3.3.6.5.1 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.

3.3.6.5.2 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.

3.3.6.5.3 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.

3.3.7 Connectivity Manager (CM)

The Connectivity Manager (CM), as described already in D3.3.1, 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 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 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 NSR is sent by the NFVO to the CM as part of the payload of the INSTANTIATE_FINISH or RELEASE_RESOURCE_FINISH events. 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.

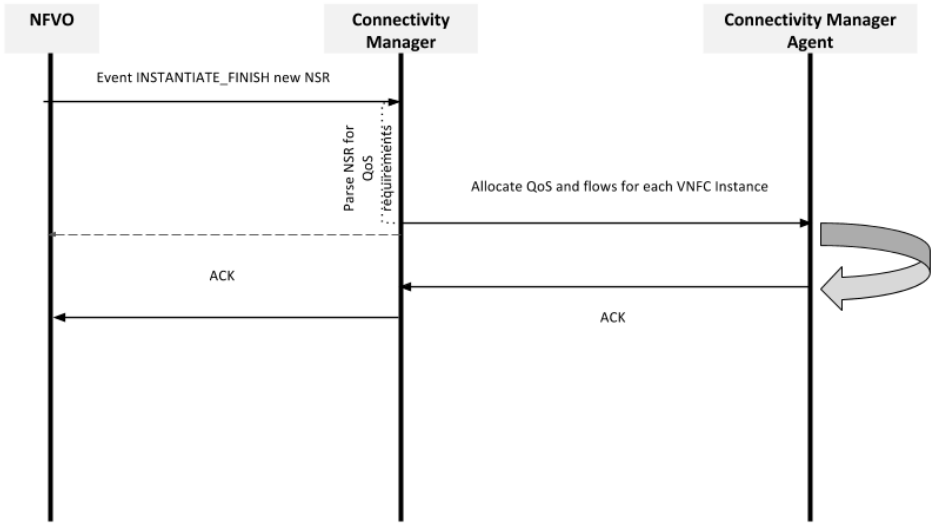


Figure 19. Interactions between the NFVO, CM and CMA.

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 [8].

3.4 NUBOMEDIA PaaS software components

In this section will be explained how the PaaS layer software components were designed and implemented. Figure 20 shows the implementation of the PaaS Manager functional elements and its APIs.

As already mentioned in D2.4.2 [2], 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.

NUBOMEDIA PaaS component is Openshift Origin v3, a PaaS platform developed by Redhat that offer containerization using Docker, Kubernetes and Project Atomic.

As shown in Figure 20 ,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 store the Application data that will be used for deletion, query status and UI

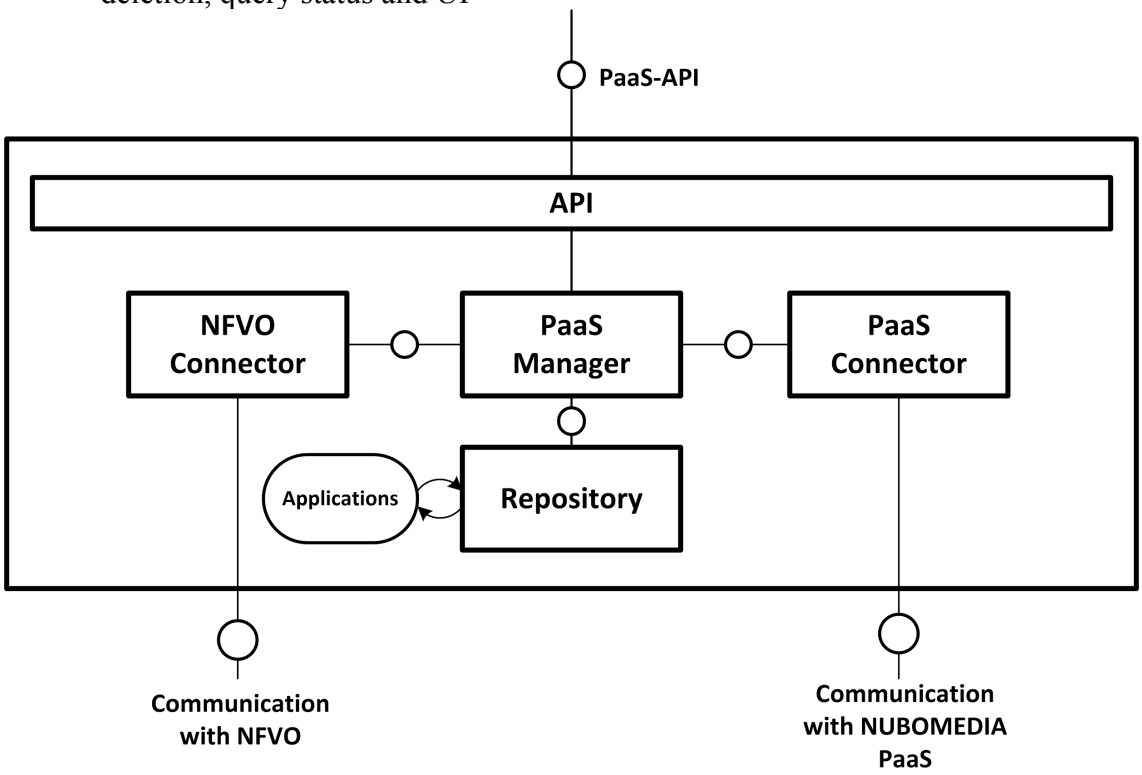


Figure 20. PaaS Manager archtiectural view

More details about the internal modules are given in the following subsections.

3.4.1 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/v1/nubomedia/paas". It uses GSON as default de-serializer of the requests body. The component is used to receive external request from developers for authentication, secret creation/deletion, application creation/deletion, retrieval of the building status.
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. Table 2 gives an overview of the exposed REST APIs.

Table 2. NUBOMEDIA PaaS REST API

Method	URL	Description
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
GET	/api/v1/nubomedia/paas/app/{id}	Retrieve the status of an application with the given id.
GET	/api/v1/nubomedia/paas/app	Return the list of all deployed projects
DELETE	/api/v1/nubomedia/paas/app/{id}	Delete project with the given id
POST	/api/v1/nubomedia/paas/secret	Create a secret[8]
DELETE	/api/v1/nubomedia/paas/secret/{name}	Deletes a secret
POST	/api/v1/nubomedia/paas/auth	Authenticate a PaaS User

For creating an Application it is necessary to send a JSON object like the one shown below:

```
{
  "gitURL":"https://github.com/example/example-app.git",
  "appName":"my-app-name",
  "cloudRepository":boolean,
  "cloudRepoPort":"cloud-repository-desidered-port";
  "ports":[
       {
          "targetPort":          8080(for          example),
          "port":8088             (for             example),
          "protocol":       "TCP"       (for       example)
       }
  ],
  "qualityOfService(optional)":"BRONZE/SILVER/GOLD",
  "flavor":"SMALL/MEDIUM/LARGE",
  "replicasNumber":2,
  "secretName(optional)":"previously-defined-secret-name",
  "turnServerActivate":"boolean",
  "turnServerIp(optional)":"stun-server-ip",
  "turnServerUsername(mandatory          with          the
turnServerIp)":"username",
  "turnServerPassword":"password"
  "stunServerActivate":"boolean",
```

[8] The Secret object type provides a mechanism to hold sensitive information such as passwords, client config files, dockercfg files, private source repository

```
  "stunServerIp":"stun-server-ip",
  "stunServerPort":"stun-server-port",
  "scaleInOut":integer-maximum-number-of-scale
  "scale_in_threshold":"number-of-sessions-to-scale-in",
  "scale_out_threshold":"number-of-sessions-to-scale-out"
}
```

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;
- stunServerAddress: 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
- scaleInOut: 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

3.4.2 PaaS Manager

The PaaS manager receives the external request from the API, saves the application data in the repository and dispatch the requests to the NFVO Connector and PaaS Connector. The PaaS Manager is the central application logic module for holding everything together. It receives requests via the PaaS API and forwards requests to the appropriate internal service connectors subsequently aggregating together the received information.

It requests NSR creation and subscribe to the INSTATIATE_FINISH event for that NSR. When it receives the event check if the event action is the right one (could be ERROR) and make the request to the PaaS through PaaS Connector. It uses CRUD operations to save, delete and retrieve application data.

3.4.3 NFVO Connector

The NFVO Connector uses the NFVO SDK to perform requests to the NFVO. It receives a deployment flavour (with a generated Application ID) and sends a request to the NFVO for creating a NSR (specifying the NSD Id) with that specific deployment flavour. It also subscribes to the event of INSTATIATE_FINISH for that NSR (with that ID).
The connector is also used to retrieve the status of the NSR instantiation and delete it when the delete request is sent to the API.

3.4.4 PaaS Connector

This connector defines all the POJOs, message configurations and beans to interact with the NUBOMEDIA PaaS (Openshift) through its REST API. It is responsible for the Application instantiation, query status, deletion, secret creation, secret deletion, build log, status query and authentication (through PaaS Identity Provider).

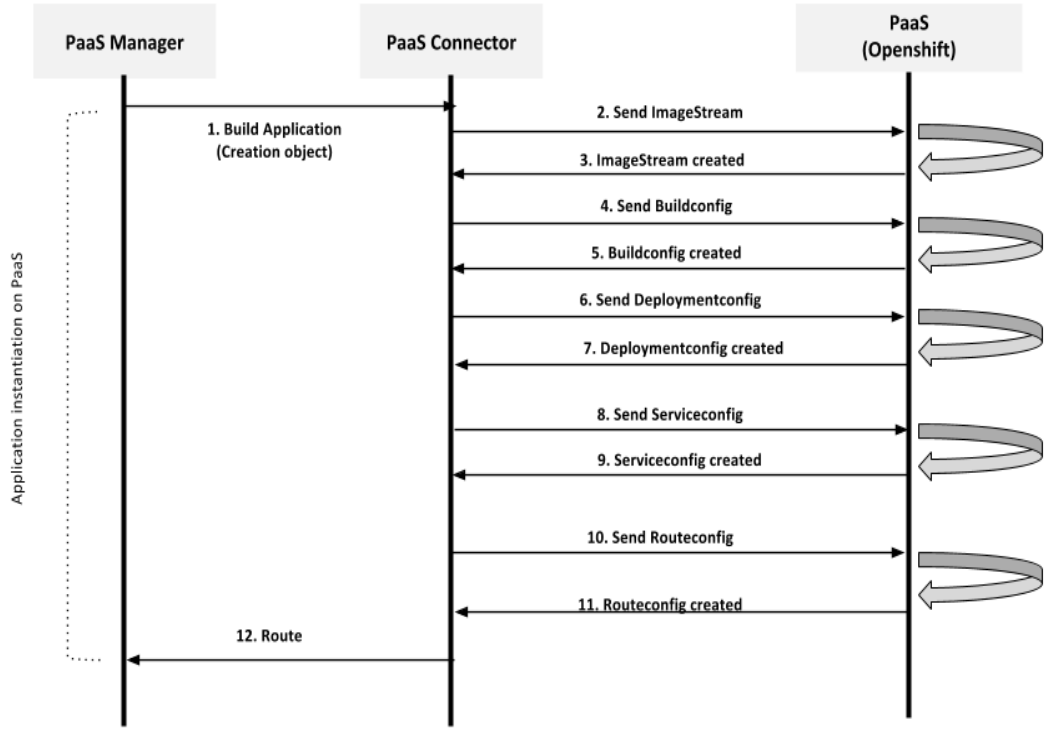


Figure 21. PaaS Connector sequence diagram

The application instantiation is triggered by the PaaS Manager using data received by the user through the API. The connector creates a service configuration for each part of the Application and upload them to the PaaS one after the other. Internally the PaaS uses a name convention to define services, the PaaS connector respects this convention using application name for each service configuration. The PaaS Connector defines a bean for each module of the PaaS, and map all the REST requests for that specific module.

Requests are made via the RestTemplate library, available on the Spring framework, using the authentication header for all the operations involving applications and uses a custom request factory to avoid the redirect for authentication API provided by PaaS.

3.4.5 Repository

The application repository uses CrudRepository<Application, String> for managing POJO and their persistency on the database. The CrudRepository interface provides all methods to interact with a Database, it offers the mapping of CRUD operation for a specific Java Object on a Database.
The metadata of the deployed applications includes

- 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

3.4.6 OpenShift as the NUBOMEDIA PaaS

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.

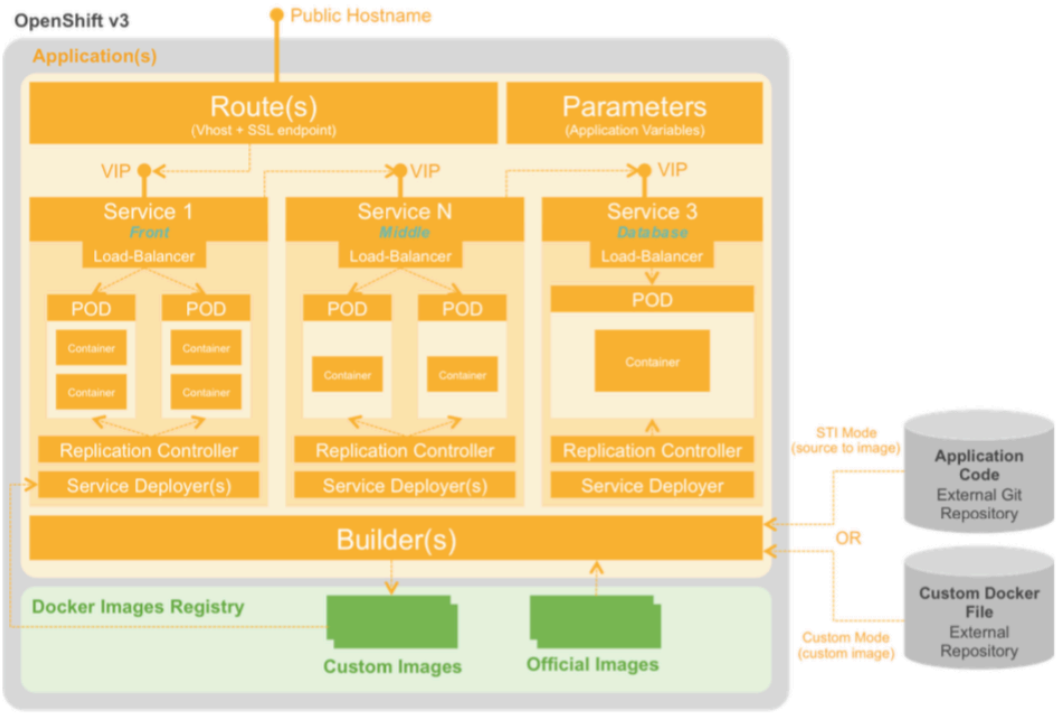


Figure 22. OpenShift general architecture

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.

3.4.6.1 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.

3.5 NUBOMEDIA autonomous installer

The NUBOMEDIA Autonomous Installer[9] is able to install new NUBOMEDIA instances into IaaS environment and having the capability to add new hardware in a "plug and play" environment. Most IaaS open-source Cloud computing solutions (e.g. OpenNebula, OpenStack, CloudStack) have been designed to be easily adapted to any infrastructure and easily extended with new components (hardware, instances). The virtualisation infrastructure can interface with multiple datacenter services by two types of interfaces: end-user cloud and system interfaces.

Cloud interfaces are specially used to develop new API tools targeted to the end-user. On the other hand, the system interfaces adapt and tune the behavior of IaaS to the target infrastructure. The Autonomous installer manages virtual machines, networks and images through a simple and easy-to-use of OpenStack REST API.

For developing the NUBOMEDIA Autonomous installer we worked with the identity service, image service, network service and of course compute service.
NUBOMEDIA, exposes several API that are consumed by the following SDKs:

- Identity: Keystone API - https://github.com/openstack/python-keystoneclient
- Image service: Glance API - https://github.com/openstack/python-glanceclient
- Network: Neutron API - https://github.com/openstack/python-neutronclient
- Compute: Nova API - https://github.com/openstack/python-novaclient

When creating the NUBOMEDIA autonomous installer we created a document[10] evaluating several possibilities like: Chef, Puppet, SaltStack, Ruby and Python. In the end we decided to create a python application that uses the SKDs presented in previous sections in order to enable the deployment of NUBOMEDIA platform on top of OpenStack.

In the following sections we tried to describe the main steps needed to deploy all the NUBOMEDIA components using the autonomous installer.

[9] https://github.com/usv-public/nubomedia-autonomous-installer
[10] https://docs.google.com/document/d/16MT-3lYtNOXtmgmSNBIbrkSza2OblPcwBdQ_mrfzHug/edit

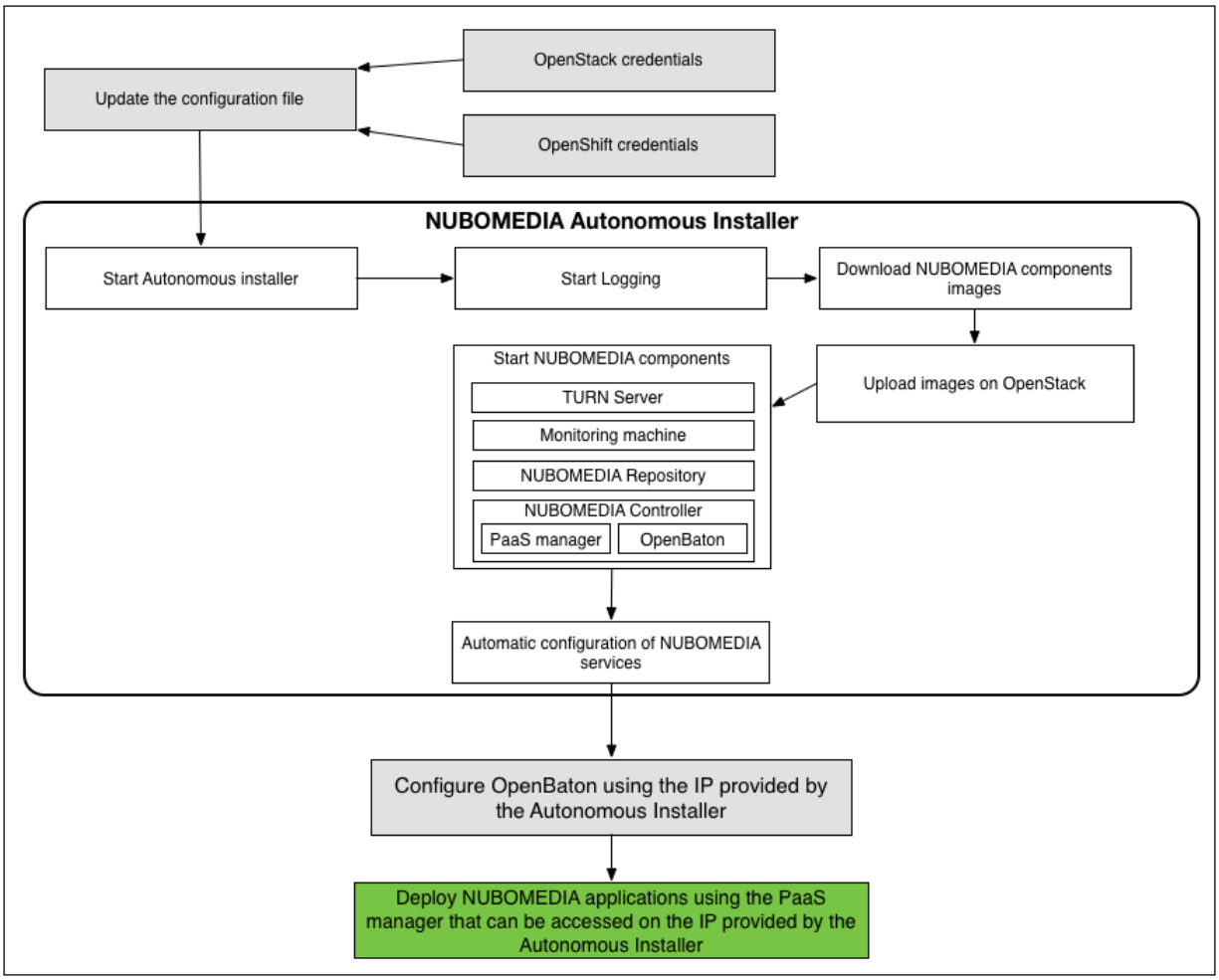


Figure 23 NUBOMEDIA Autonomous Installer process scheme

On the NUBOMEDIA Autonomous Installer process scheme we presented the steps needed to deploy NUBOMEDIA platform and then a application on top of it. Parts of the scheme with gray background represent actions that the System Administrator doing the deployment of the platform should configure before and after the Autonomous Installer job. The part in green represents the final point where NUBOMEDIA users deploy their application on top of the architecture.

3.5.1 NAI prerequisites

In order to be able to run the nubomedia-autonomous-installer you have to check that your current python version is 2.7.XX with the following command:
python --version

Then you should install pip and after that you should install the dependecies using the following commands. When asked for any kind read it first and then confirm:
easy_install pip
pip install -r requirements.txt –upgrade

You should have root access to the OpenStack environment in order to install and configure the nova-docker hypervisor and the patch for it from (https://github.com/usv-public/nubomedia-nova-docker) on the compute nodes that run docker as a hypervisor.

3.5.2 Update the configuration file

Before starting the installation, you must rename the variables-examples.py to variables.py. And then replace each variable with the desired value.

mv variables-example.py variables.py

In the Keystone part of the configuration file you should replace x.x.x.x with the public IP address of the OpenStack environment and the username, password and tenant name on which you want to deploy the NUBOMEDIA platform. Glance endpoint IP address should also be the IaaS public IP.
If the IaaS environment on which you are deploying the NUBOMEDIA platform does not have applied the patch for nova-docker hypervisor then you should define the master public IP address with username and password or private key file in order to allow the autonomous installer to provision the Docker image of Kurento Media Server on the platform.
You should define the floating (public) IP pool name for the OpenStack, in order to allow provisioning of public IP addresses for the deployed components.
You should first add a public key on the OpenStack tenant you want to deploy NUBOMEDIA and then add the private key file to the autonomous-installer directory in order to allow it to customize the instances after deployment.
For configuring the PaaS Manager we need to have the public IP address of the OpenShift.

3.5.3 NUBOMEDIA images configuration

You should download all the NUBOMEDIA images and store them on the *repository/images/* directory inside the installer.

3.5.3.1 NUBOMEDIA Kurento Media Server - qemu image for KVM

Kurento Media Server requires a flavor with at least 2GB of ram, 1 x86_64 CPU and 5GB of disk space.
kms_qemu_img = 'resources/images/kurento-media-server.qcow2'
kms_image_name = 'kurento-media-server'
kms_image_description = 'Kurento Media Server image for KVM hypervisor'
kms_qemu_flavor = 'm1.medium'

3.5.3.2 NUBOMEDIA Kurento Media Server Docker image for – Docker

The Docker image for kurento-media-server is stored on the NUBOMEDIA dockerhub repository (https://hub.docker.com/r/nubomedia/kurento-media-server/). The minimum flavor type would be at least 2GB of ram, 1 x86_64 CPU and 5GB of disk space.
kms_docker_img = 'nubomedia/kurento-media-server'
kms_docker_image_description = 'Please login with root user and your docker image root password'
kms_docker_flavor = 'd1.medium'

3.5.3.3 NUBOMEDIA Monitoring machine - qemu image for KVM

The NUBOMEDIA Monitoring machine runs logstash, Kibana, and Graphite. All the logs of the media-server instances are stored on this machine. The required flavor for running it should have at least 2GB of ram, 1 x86_64 CPU and 15GB disk space.
monitoring_qemu_img = 'resources/images/nubomedia-monitoring.qcow2'
monitoring_image_name = 'nubomedia-monitoring'
monitoring_image_description = 'Please login with ubuntu user and your private_key'
monitoring_flavor = 'm1.medium'

3.5.3.4 NUBOMEDIA TURN Server machine - qemu image for KVM

The TURN Server should have at least 1G of RAM, 1 x86_64 CPU and 5GB disk space. All the NUBOMEDIA Kurento Media Server instances should be configured to use this server IP for TURN and STUN.
turn_qemu_img = 'resources/images/nubomedia-turn.qcow2'
turn_image_name = 'nubomedia-turn'
turn_image_description = 'Please login with ubuntu user and your private_key'
turn_flavor = 'm1.small'

3.5.3.5 NUBOMEDIA Repository Server machine - qemu image for KVM

The NUBOMEDIA repository machine hosts the debian repository and on the http://repository.nubomedia.eu/apps/files/ you have the kurento-tutorial-java repository so you can use it as the APP server for some of the Kurento tutorials. The recommanded configuration is 1G of RAM, 1 x86_64 CPU and 5GB of disk space.
repository_qemu_img = 'resources/images/nubomedia-repository.qcow2'
repository_image_name = 'nubomedia-repository'
repository_image_description = 'Please login with ubuntu user and your private_key'
repository_flavor = 'm1.medium'

3.5.3.6 NUBOMEDIA Conroller machine - qemu image for KVM

The NUBOMEDIA Controller instance hosts the VNFM and the NUBOMEDIA PaaS manager. The flavor type of for the nubomedia-controller should have at least 8GB of ram, 2 x86_64 CPU and 10GB disk space.
controller_qemu_img = 'resources/images/nubomedia-controller.qcow2'
controller_image_name = 'nubomedia-controller'
controller_image_description = 'Please login with ubuntu user and your private_key'
controller_flavor = 'm1.xlarge'

3.5.4 Run the installer

After all the necessary variables are defined you can start the installer with the following command:
python main.py

3.5.5 Future plans

For the next release we plan to support multiple types of IaaS platforms like Rackspace, AWS and maybe Microsoft Azure.

4 Integrated scenario

In this section are presented some sequence diagrams for the most important procedures of the NUBOMEDIA components.

4.1 Deployment of an application

To build and deploy an application on the NUBOMEDIA infrastructure the developers have to make a HTTP Post Request (or use the GUI) using a JSON object as body. The PaaS manager will get that parameters from API and dispatch them to NFVO and PaaS through connectors. These two steps are done in sequence otherwise the application won't find an NSR.

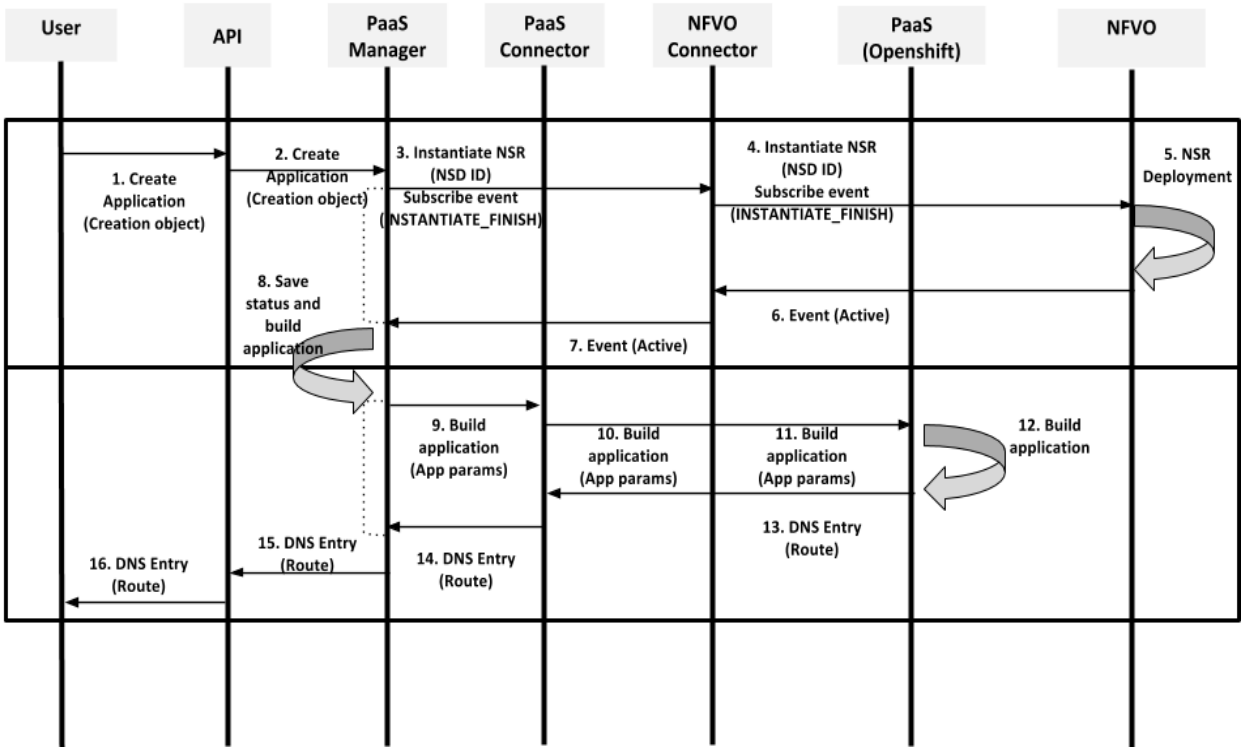


Figure 24. Deployment of an application

4.1.1 Network Service Record (NSR) deployment sequence diagram

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. A vim instance JSON file is shown in the annex A. 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. In the same annex there is a small example of the NSD used in this project. 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:

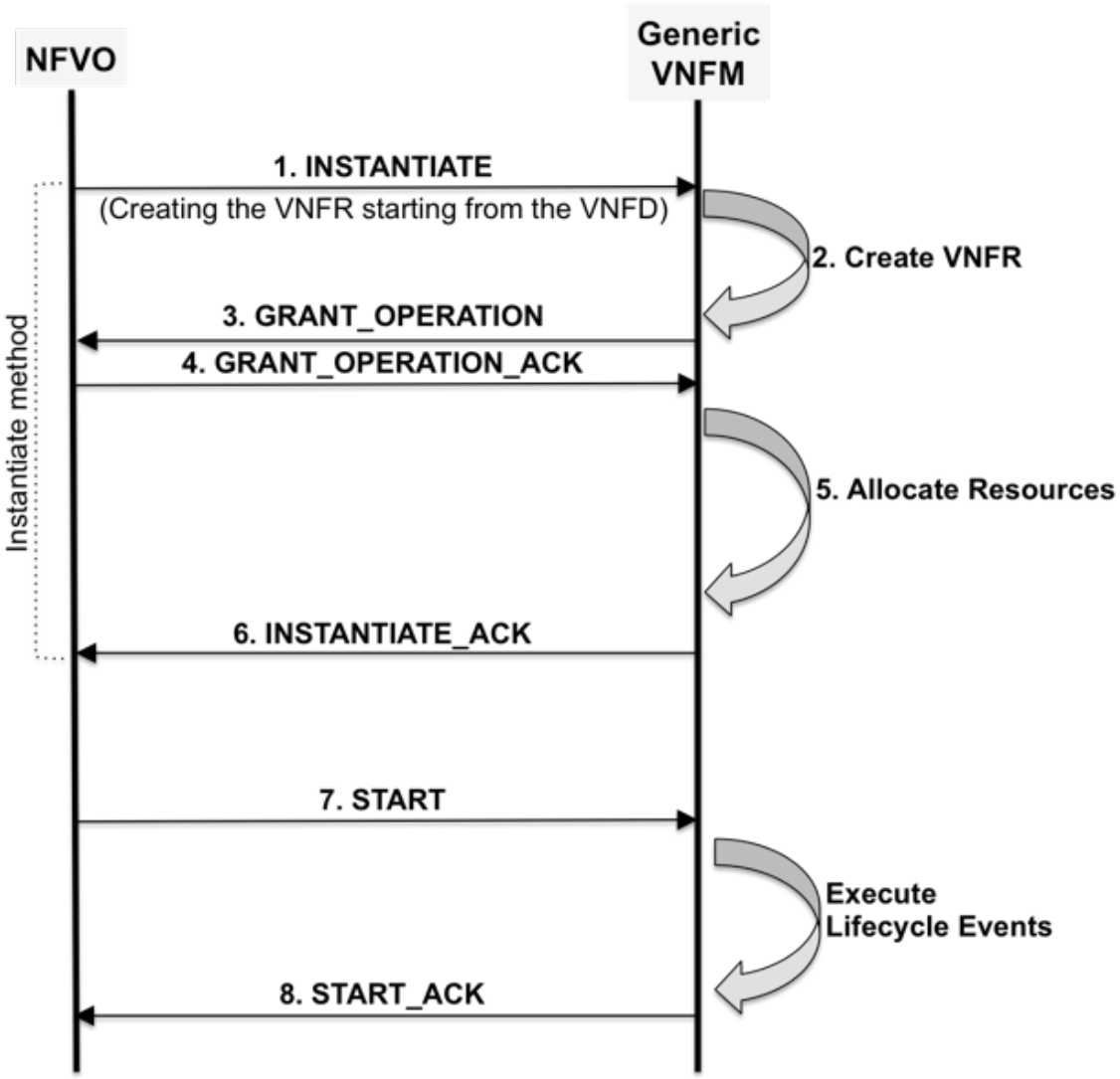


Figure 25. NFVO, VNFM and EMS sequence diagram for NSR deployment

4.1.1.1 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).

4.1.1.2 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.

4.1.1.3 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).

4.1.2 Application management

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.

4.1.2.1 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 26 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.

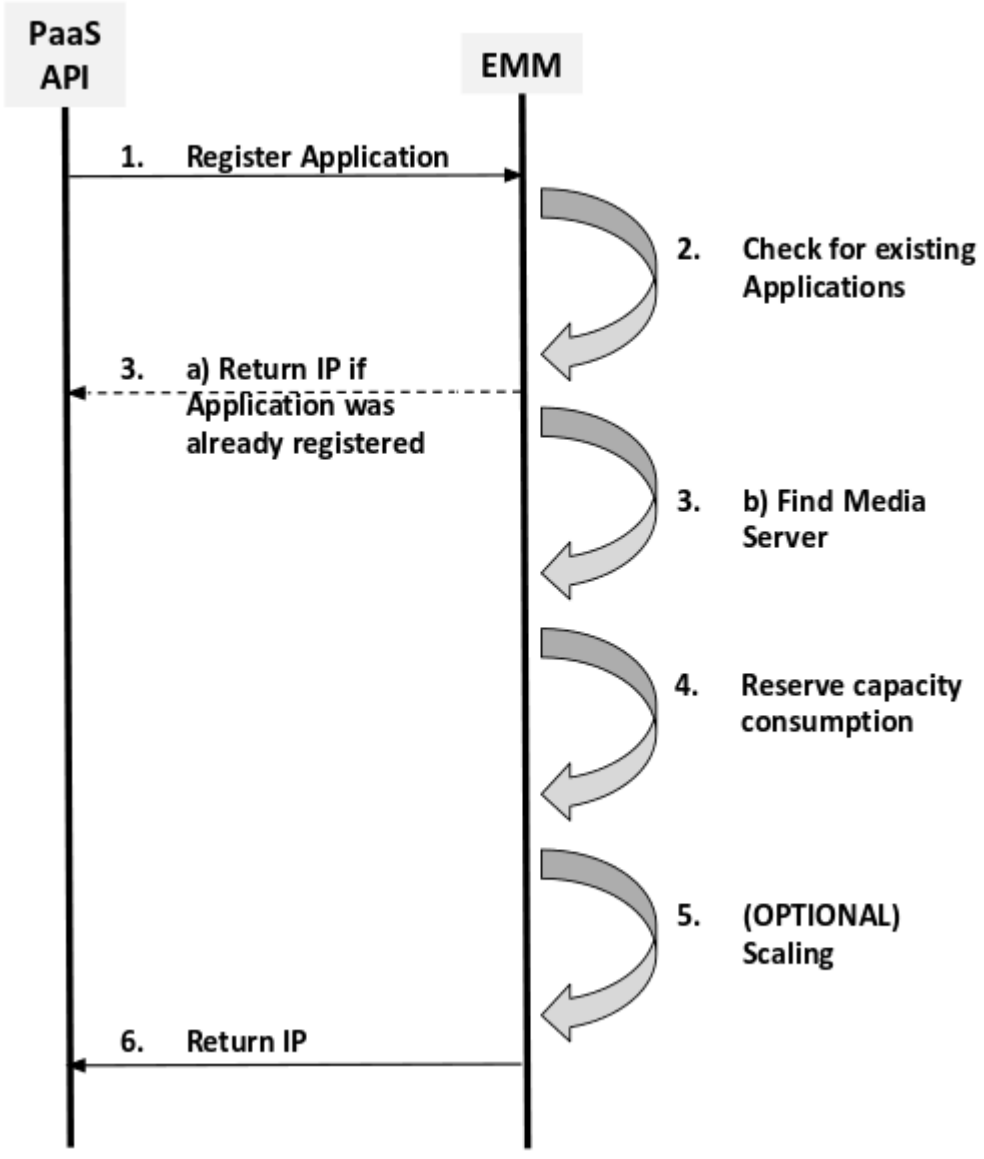


Figure 26. Registration of an Application

4.1.2.1 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.

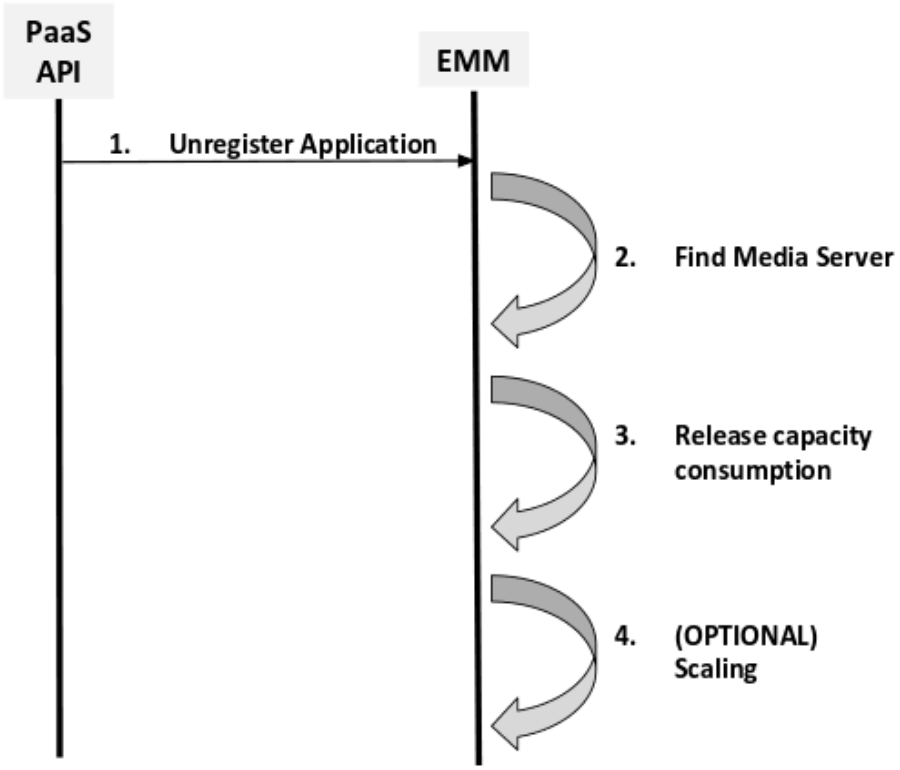


Figure 27. Unregistering an Application

4.1.2.2 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.

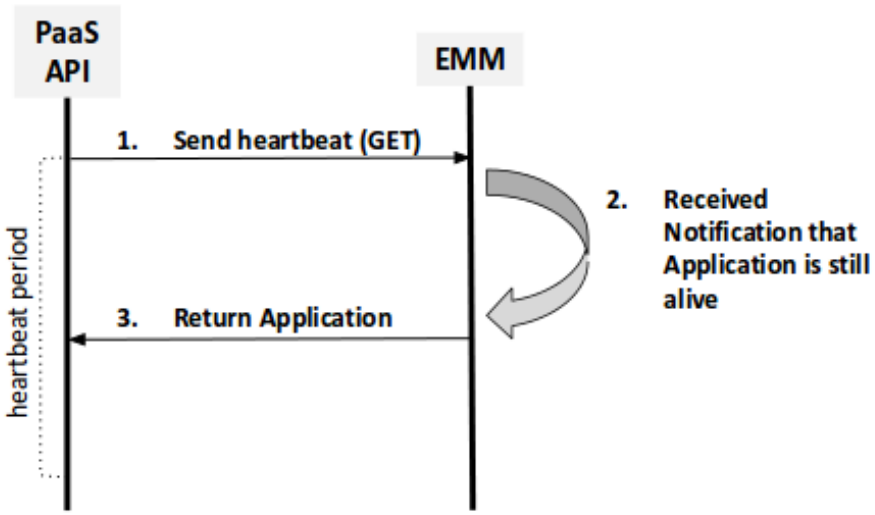


Figure 28. Heartbeat mechanism

In Figure 28 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 29) 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.

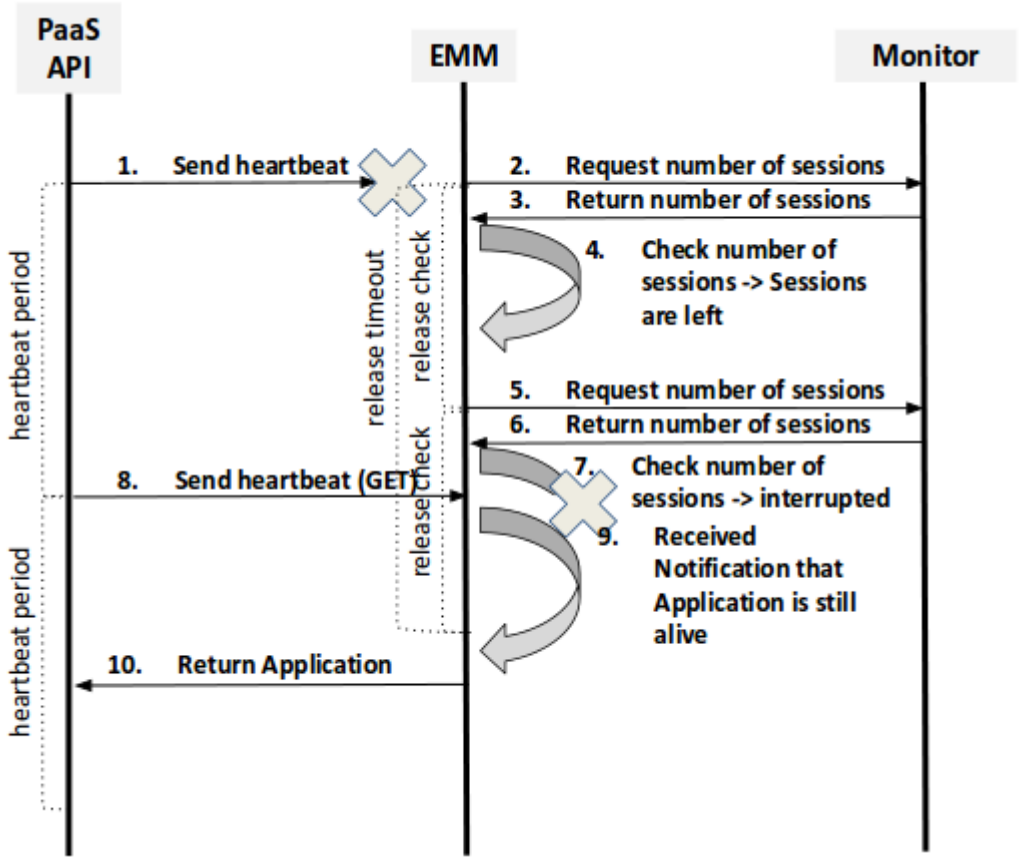


Figure 29. Heartbeat interrupts release check

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 30 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.

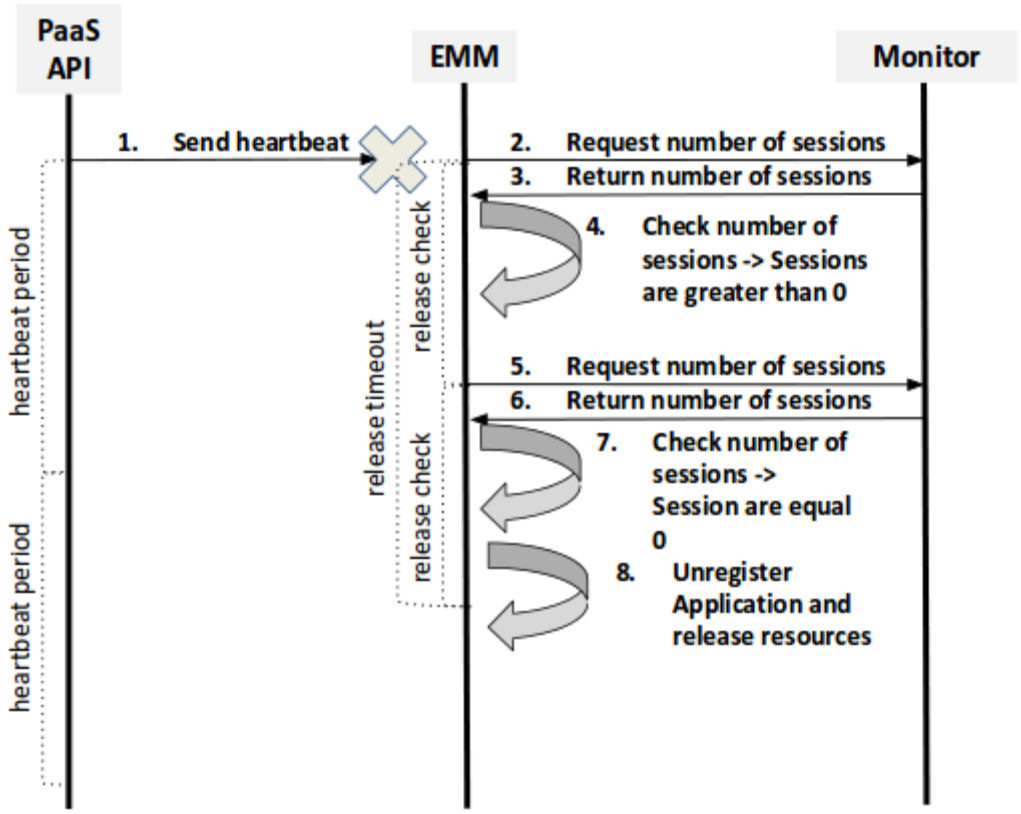


Figure 30. Missing Heartbeats and release check

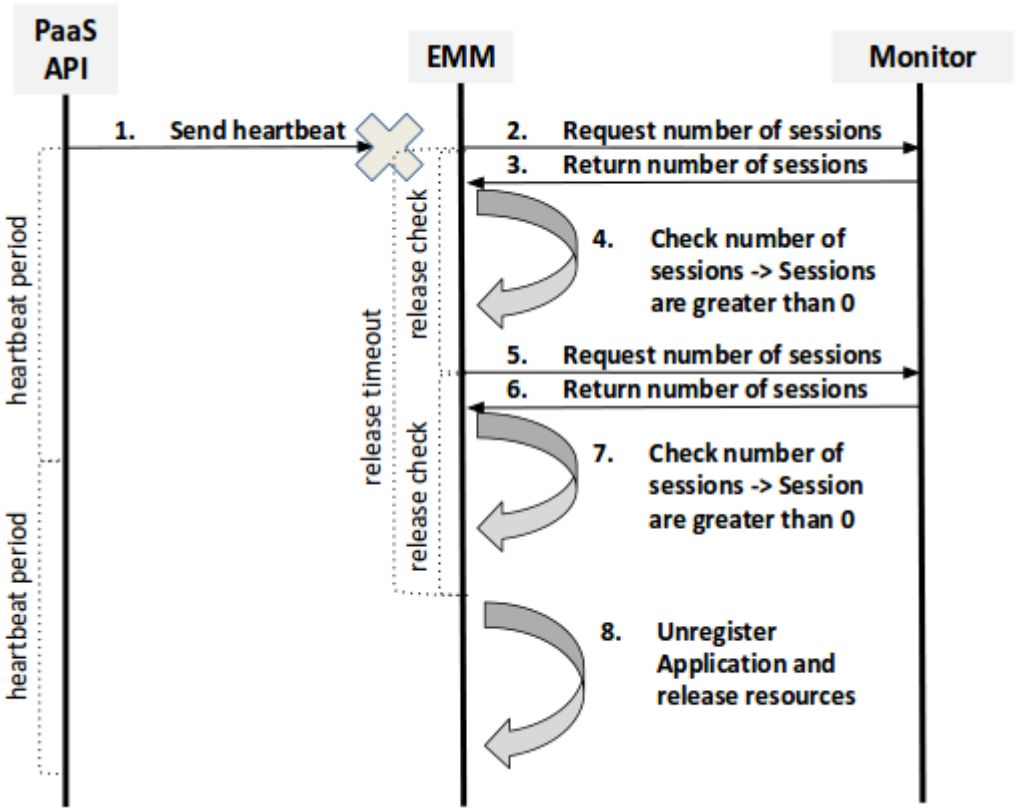


Figure 31. Missing Heartbeats and release timeout

In Figure 31 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.

Annex

A JSON file examples

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":[
    ]
}
```

B Gathering monitoring information

Monitoring information is gathered with a collectd agent that is installed on all images. Collectd is pushing metrics to the Monitoring platform which is based on an open source tool Graphite.

We gather the following metrics from each instance:

- CPU usage (idle, nice, system, user, IO wait)
- CPU load (short, mid and long term)
- Network usage (packets and octets)
- Memory usage (buffered, cached, free, used)

On Graphite platform can be analyzed the metrics through the dedicated dashboard. They are stored at 30s interval for 3 days, 60s interval for 1 year and 5 min interval for 10 years.

For more information about how metrics can be sent/retrieved to and from the monitoring system we added an API documentation which can be found on the Annex, Realtime stats monitoring system API part.

Collectd and logstash-forwarder are installed on the nubomedia/base_image docker image which is available on docker hub. All NUBOMEDIA applications developed on top of docker that will be deployed on the PaaS must use the base_image.
For the actual NUBOMEDIA instance we've configured the monitoring server to have 80.96.122.90 IP address, but for new NUBOMEDIA deployments you must modify the configuration file from */etc/collectd/collectd.conf* replacing the MONITORING_DEV variable to the coresponding IP address of the monitoring machine on your environment.
Logstash-forwarder configuration file located at /etc/logstash-forwarder.conf should also be updated with the corresponding IP address needed of the new monitoring instance.
This can be done running the following bash script:

```
#!/bin/bash
#replace with current hostname
sed -i -e "s/HOSTNAMEMONITORING/$HOSTNAME/g" /etc/collectd/collectd.conf
#replace with the current graphite server
MONITORING_DEV='80.96.122.69'
sed -i -e "s/80.96.122.69/$MONITORING_DEV/g" /etc/collectd/collectd.conf
sed -i -e "s/80.96.122.69/$MONITORING_DEV/g" /etc/logstash-forwarder.conf
/etc/init.d/collectd restart
service logstash-forwarder restart
```

For *nubomedia/kurento-media-server* docker image and *nubomedia-media-server-6.2* KVM image you must run the script at instance provisioning using the User-data functionality available on the IaaS.

In order to access the monitoring tools you should first login on the https://devconsole.nubomedia.eu and on the middle top, near the tenant name you the monitoring link, like presented in the following screenshot:

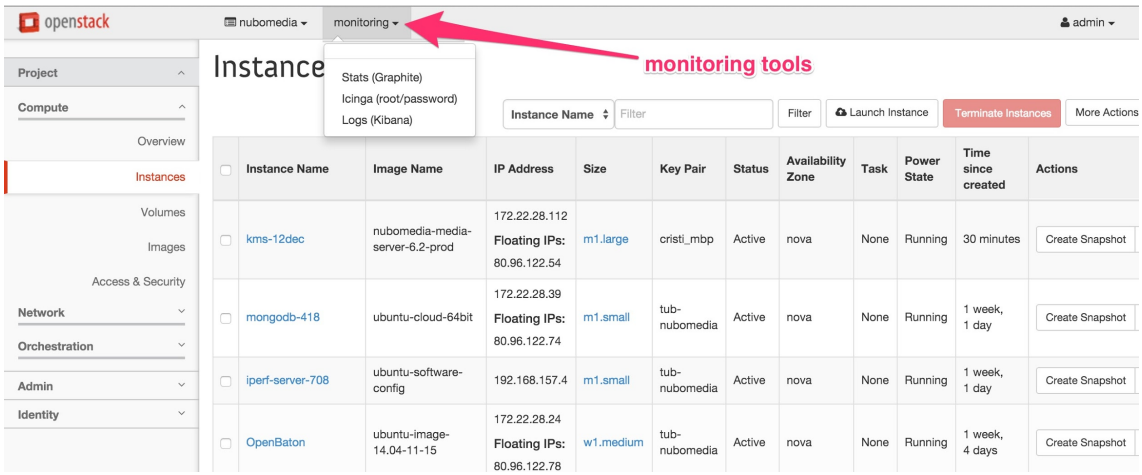


Figure 32. Monitoring tools link on the dashboard

On the Graphite monitoring system we can see the metrics in almost realtime.

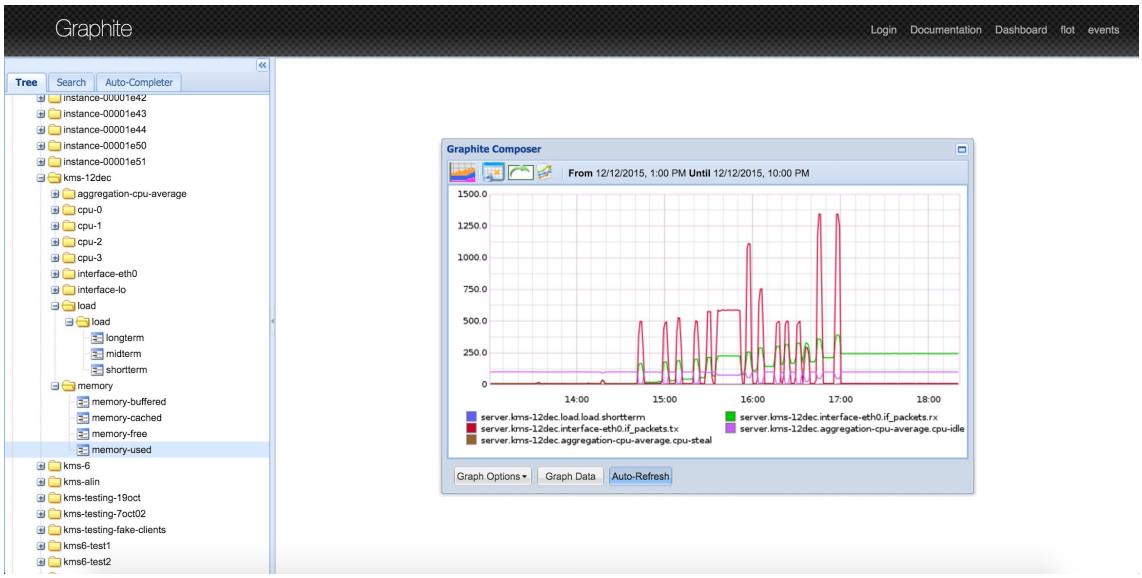


Figure 33. Graphite metrics in real-time

Kibana is used to store all the logs from all docker KMSs and applications deployed on top of the PaaS.

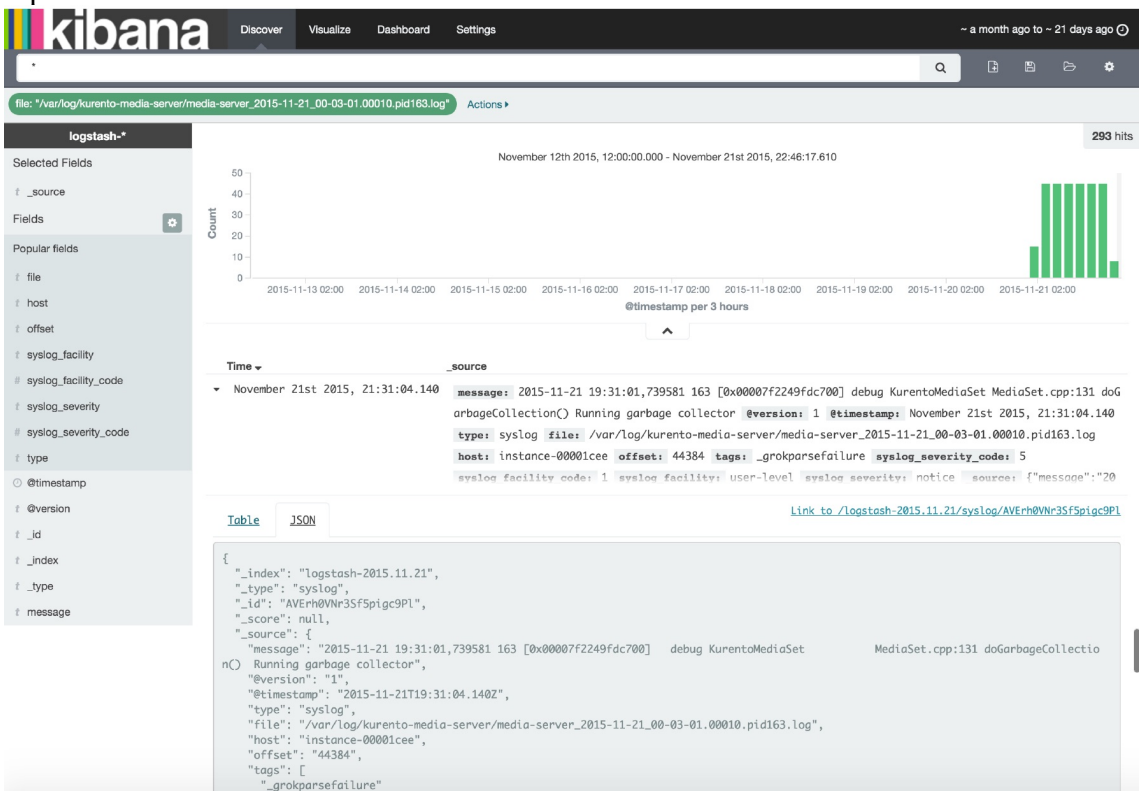


Figure 34. Kibana log monitoring for the KMS docker instances

Real time stats monitoring system API are described in the following document[11].

11 https://docs.google.com/document/d/1cMdPQ0sb6ENLNu1X4x98SSXAOUfZLsVcHB-5iGKYO7Q/edit

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

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

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

[6] http://www.top500.org/lists/2015/11/

[7] Joe Pizzini, "GPU Rendering vs. CPU Rendering – A method to compare render times with empirical benchmarks, http://blog.boxxtech.com/2014/10/02/gpu-rendering-vs-cpu-rendering-a-method-to-compare-render-times-with-empirical-benchmarks/ (2014).

[8] 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

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

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

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

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

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

[14] D3.2.1: Cloud Repository

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

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

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

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

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