D2.4.3

Version	3.1
Author	URJC
Dissemination	PU
Date	04/05/2017
Status	Final

D2.4.3: NUBOMEDIA Architecture v3

Project acronym:	NUBOMEDIA
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2017-02-01
Project type:	STREP
Project reference:	610576
Project web page:	http://www.nubomedia.eu
Work package	WP2: User centric analysis of user scenarios and requirements
WP leader	VTOOLS
Deliverable nature:	Report
Lead editor:	Luis López Fernández
Planned delivery date	30/11/2016
Actual delivery date	04/05/2017
Keywords	Architecture

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:

Luis López Fernández (URJC)
Giuseppe Carella (TUB)
Lorenzo Tomasini (TUB)
Javier López Fernández (NAEVATEC)
Alin Calinciuc (USV)
Alice Cheambe (Fhg FOKUS)
Boni García (URJC)

Internal Reviewer(s):

Giuseppe Carella (TUB)

Version History

Version	Date	Authors	Comments
0.1	28/07/14	Luis López Fernández	First initial version
0.2	8/10/14	Luis López	Cloud architecture as functional architecture.
0.3	3/01/15	Luis Lopez	Mayor modifications to architecture in order to simplify it and to adapt to a flatter model where all the logic of distributed media pipeline management is left to applications.
1.0	19/01/15	Luis Lopez	Added CFD descriptions for TUB. Added new reference points descriptions.
1.1	30/06/2015	Alice Cheambe	Updated DSFE
1.2	13/07/2015	Giuseppe Carella	Restructuring the document considering the comments received by the reviewers after year 1 review
2.0	20/09/2015	Luis Lopez	Complete factorization of document for adapting to reviewer requirements.
2.1	2/01/2016	Giuseppe Carella	Contributions for NFVO and VNFM
2.2	2/01/2016	Alice Cheambe	Contributions for PaaS Manager and PaaS
2.3	14/01/2016	Giuseppe Carella	Added NVF basic introduction. Added EMM description.
2.4	14/01/2016	Alin Calinciuc	New version of the NUBOMEDIA IaaS.
2.5	18/01/2016	Alice Cheambe & Giuseppe Carella	PaaS manager main interactions. Reviewed EMM FMC architecture.
3.0	03/10/2016	Luis Lopez	Update metadata for D2.4.3
3.1	06/04/2017	Boni García, Giuseppe Carella	Update according to reviewers' comments

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
CPU	Central Processing Unit
DAS	Distributed Application Server (refers to function class)
DMP	Distributed Media Pipeline
DMS	Distributed Media Server (refers to function class)
DoD	Denial of Service
DSFE	Distributed Signaling Front End (refers to function class)
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
ME	Media Element
MP	Media Pipeline
MS	Media Server (refers to function instance)
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
PAMC	Platform Application Media Control
PNF	Physical Network Function
RFC	Request For Comments
RTC	Real-Time Communications
SaaS	Software as a Service
SFE	Signaling Front End (refers to function instance)
UE	User Equipment
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFM	VNF Manager
WS	WebSocket
WWW	World Wide Web
XMPP	eXtensive Message and Presence Protocol

1 Executive summary

This document contains a specification of the NUBOMEDIA architecture. This specification provides the basis on for driving the technological developments in WP3, WP4 and WP5.

The NUBOMEDIA Platform is an elastic cloud PaaS that provides elastic scalability for media delivery and autoscaling mechanisms able to orchestrate media components. These components are provided by a modular media plane, suitable for hosting media capabilities including media transport, media (de)coding, media filtering, media interoperability and media processing.

This deliverable contains a detailed description of the NUBOMEDIA architecture. This architecture has been specified using the Fundamental Modeling Concepts (FMC) framework. The main components of the NUBOMEDIA Platform are the following:

- NUBOMEDIA PaaS. Level between the infrastructural resources and the users of the platform. Particularly it includes the NUBOMEDIA PaaS Manager exposing an interface to developers for deploying their applications.
- 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.
- NUBOMEDIA IaaS. Composed by the infrastructure resources in terms of Compute Nodes (CN) providing virtual compute, storage and networking resources to the upper layers, and the Virtual Infrastructure Manager (VIM) providing the abstracted APIs for controlling their lifecycle.

Last but not least, the NUBOMEDIA Platform provides three groups of APIs suitable for complying with the requirements of a wide spectrum of developers:

- Media Capability APIs: NUBOMEDIA Media and Repository APIs.
- Signaling APIs: JSON-RPC over WebSocket client and server APIs.
- Abstract Communication APIs: Room and Tree client and server APIs.

2 Terminology and format conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

All architectural diagrams in this document have been created following the conventions of the Fundamental Modeling Concepts [FMC] framework.

Following FMC conventions, active components are depicted as named squared-corner boxes and passive components as named rounded-corner boxes. Components names comprise one or several upper-case terms. Interfaces are represented through named or un-named circles (see Table 1). For named interfaces, we accept the convention of them starting with a lower-case “i” character.

FMC notation is convenient for representing the compositional structure of software systems through three types of graphical elements: agents, storages and channels. Agents are active systems provided of some kind of logic, which have access to adjacent passive systems. Example of agents are active software components or human users. Agents are presented through squared boxes. Storages are used by agents to store and recover data. Agents are represented through circled shapes. Channels, in turn, represent communications among agents. Channels are represented through small circles qualifying an access type, represented through directed or undirected lines. For further information about FMC visit the [FMC reference].

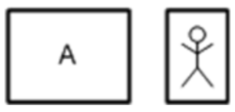
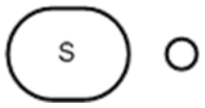
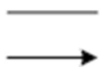
FMC Element	Name	Description
	Active system component: agent, human agent	Serves a well-defined purpose and therefore has access to adjacent passive system components and only those may be connected to it. A human agent is an active system component exactly like an agent but the only difference that it depicts a human.
	Passive system component (location): storage, channel	A Storage is used by agents to store data. A channel is used for communication purposes between at least two active system components.
	Access type	Directed and undirected edges represent the kind of access an active system component has to a passive system component.

Table 1. Fundamental Modeling Concepts (FMC) notation basic elements.

3 Introduction

NUBOMEDIA is the first open source elastic cloud PaaS (Platform as a Service) specifically designed for real-time interactive multimedia services, which exposes its capabilities through simple APIs. For understanding how a real PaaS works we just need to think on services such as Heroku, the Google App Engine or Pivotal PCF. All of them expose to developers the ability of uploading, deploying and executing applications that leverage the PaaS capabilities through custom SDKs. By using them, developers do not need to worry about aspects such as the provisioning, the scalability, the resilience or the security of the services they consume as they are provided off-the-shelf by the PaaS. This model is quite convenient as it lets developers to concentrate on creating their application logic while all the complex aspects of deploying, scaling and securing them are assumed by the PaaS.

The NUBOMEDIA Platform has been created in the context of the NUBOMEDIA Project to comply with the following objectives:

- To provide elastic scalability for media delivery, so that physical and virtual computing and networking resources are allocated on demand to them accordingly to their QoS requirements.
- To provide a modular media plane suitable for hosting interacting media capabilities including media transport, media (de)coding, media filtering, media interoperability and media processing.
- To provide advanced autoscaling mechanisms able to orchestrate complex stateful media components.
- To provide a set of APIs suitable for complying with the requirements of a wide spectrum of developers by enabling, at the same time, abstraction and flexibility.

Therefore, NUBOMEDIA makes possible to upload, deploy and execute developers' applications written in the Java programming language. At the same time, NUBOMEDIA is a WebRTC platform as it provides the ability of accessing scalable, secure and reliable WebRTC capabilities. Thanks to this, NUBOMEDIA combines the simplicity and ease of development of WebRTC API PaaS services with the flexibility of real PaaS infrastructures. Hence, as NUBOMEDIA is a Java PaaS, developers can leverage all the capabilities of the Java platform for creating their applications. The only difference with other PaaS services it that NUBOMEDIA makes available WebRTC capabilities through a specific API. Hence, WebRTC just becomes another of the SDKs that can be used while programming. Once an application is completed, developers just need to deploy it into NUBOMEDIA and it will scale in a secure and reliable way with full transparency.

The objective of this document is to present the architecture for the NUBOMEDIA platform concentrating on understandability more than on exhaustivity or completeness. To this aim, the architecture represents NUBOMEDIA as a system (i.e. the dynamical system that emerges from the execution of the NUBOMEDIA software artifacts) and not NUBOMEDIA as a software (i.e. a number of source code artifacts). As a result, this system architecture is understood as a high-level abstraction showing the different components comprising the platform and their interactions.

4 NUBOMEDIA Architecture

4.1 Introduction

The NUBOMEDIA Platform has been conceived for simplifying the way developers deal with RTC interactive media applications. Its aim is to provide simple mechanisms and APIs for creating such applications. For this, NUBOMEDIA tries to abstract all the low-level details of service deployment, management, and exploitation allowing applications to transparently scale and adapt to the required load while preserving QoS guarantees.

For enhancing understandability and simplicity, the NUBOMEDIA Platform has been designed in such a way that, from the developer's perspective, its capabilities are accessed through APIs inspired by the popular WWW three tier model, as shown in Figure 1.

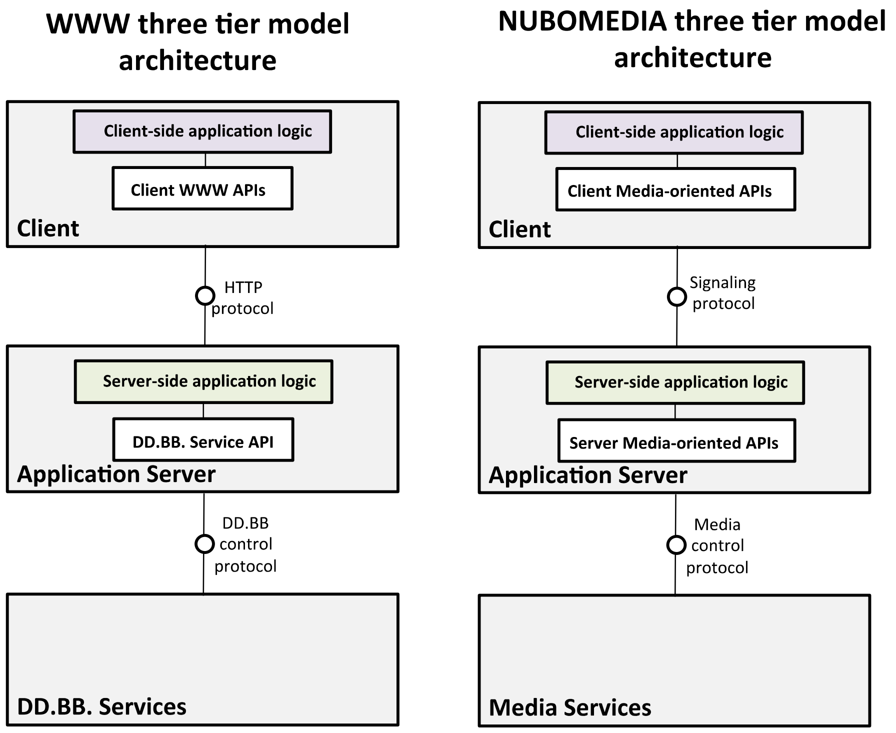

WWW three tier model architecture
Client-side application logic
Client WWW APIs
Client
HTTP protocol
Server-side application logic
DD.BB. Service API
Application Server
DD.BB control protocol
DD.BB. Services
NUBOMEDIA three tier model architecture
Client-side application logic
Client Media-oriented APIs
Client
Signaling protocol
Server-side application logic
Server Media-oriented APIs
Application Server
Media control protocol
Media Services

Figure 1. The NUBOMEDIA development model (right) is based on the popular tree tier development model of WWW applications (left), so that application developers create their programming logic consuming different service APIs: DD.BB. APIs in the case of WWW and media-oriented APIs in the case of NUBOMEDIA.

In the same way that WWW developers program their application logic consuming Data Base (DD.BB.) APIs, NUBOMEDIA makes it possible to create rich media services through media-oriented APIs. These APIs are based on the concept of Media Pipeline: a chain of Media Elements, where a Media Element is an abstraction holding a specific media capability and hiding its complexities behind a comprehensive interface. The parallelism is straightforward and understandable for developers:

- In the WWW development model, at the very top we have the client: a web browser. As browsers are thin client, they expose development APIs suitable for controlling the presentation and user interaction capabilities of the application. In NUBOMEDIA, the client is again understood as thin. It exposes media-oriented APIs enabling media capture, media rendering and media transport toward the infrastructure.

- Below the client, we have the Application Server layer, which interacts at its northbound interface with the client through some kind of control protocol that, in the case of WWW applications is typically the HTTP protocol, and in the case of NUBOMEDIA is a RTC signaling protocol. This layer holds the business logic of the application, which translates the client requests intro the appropriate orchestrated commands toward the service layer.
- At the very bottom of the architecture, the service layer holds data services, in the case of WWW applications, and RTC media services, in the case of NUBOMEDIA.

Furthermore, NUBOMEDIA provides a Cloud Platform for hosting all those different layers, using a Platform as a Service concept. Basically, a developer focuses on the implementation of the application logic, making use of the APIs provided by the NUBOMEDIA project to conceive innovative multimedia services. Once done, developers make use of the NUBOMEDIA Cloud Platform for deploying and runtime managing their applications. This Cloud Platform, provides enhanced capabilities for automatically scale any layers of the NUBOMEDIA application layers, using standard de-facto open source tools.

4.2 Architecture overview

The NUBOMEDIA Platform is a complex piece of software that requires a non-trivial architecture providing many different functions, capabilities and interfaces. For the sake of clarity and brevity, in this document we only identify its main parts and provide a definition of the main interfaces exposed by the NUBOMEDIA components, without digging deeply into its complex details. Additional details of the individual components, Cloud Platform, Media Elements, and APIs, are provided respectively in Deliverable D3.3[1], D4.3[2], and D5.3[3].

As mentioned previously, a NUBOMEDIA application is composed by three parts: the client-side application logic (executed at the client browser), the server-side application logic (executed inside an Application Server), and the RTC media services (usually Network Functions hosted in service provider' datacenters). In order to orchestrate and expose such services via API, it was employed the same multi-layered architecture.

Starting from the bottom, the first important innovation is represented by the RTC media services cloudification. This is achieved introducing and extending NFV concepts in the NUBOMEDIA Architecture. The RTC media services layer scales dynamically accordingly to the application needs.

Continuing, the server-side application logic exposes advanced APIs for building a modular media plane consuming the media capabilities provided by the lower layers. This application can be deployed on the NUBOMEDIA PaaS using the provided NUBOMEDIA PaaS API. The innovation introduced by this approach is that the NUBOMEDIA Platform will instantiate a set of RTC media services which are dedicated to the application, and which the application can dynamically scale via the advanced APIs exposed by the media service layer.

[1] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP3/D3.3_Cloud_Platform_R9_V1.1-04-05-2017_FINAL-PC_rev1.pdf
[2] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP4/D4.3_NUBOMEDIA_Media_Server_V3_R9_11-11-2016-FINAL-PC.pdf
[3] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP5/D5.3_NUBOMEDIA_framework_APIs_R9_30-11-2016_FINAL-PC.pdf

NUBOMEDIA architecture overview

The NUBOMEDIA high-level architecture is depicted in Figure 2. This architecture complies with the ETSI NFV MANO specification [MANO]. Details about each of the NUBOMEDIA components are provided in sections below. At a very high-level perspective, and following top down approach, these are the core functions:

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

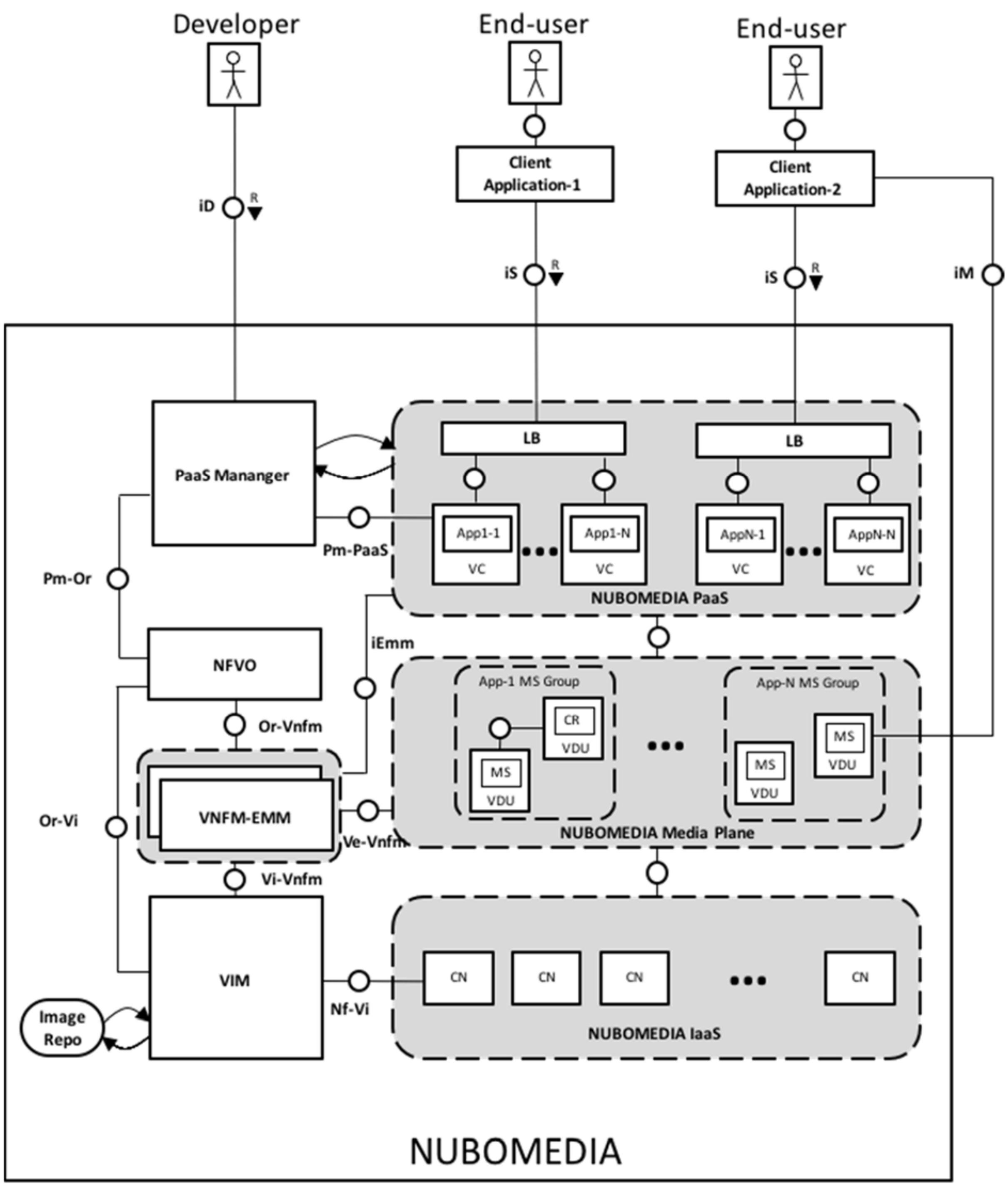


Figure 2. This figure shows the NUBOMEDIA system architecture following the Fundamental Modeling Concepts (FMC) block diagram visual notation. The NUBOMEDIA architecture is based on the ETSI NFV Management and Orchestration (MANO) recommendation but extends it with a PaaS layer. Thanks to it, when developers deploy their applications through a PaaS API, the PaaS Manager orchestrates all the required actions for the provisioning of the appropriate resources and services required for applications to run in a reliable and scalable way.

Application deployment overview

First of all, as already mentioned in the previous section 4.1, the main objective of NUBOMEDIA is to build a Cloud Platform exposing capabilities for dynamically manage and orchestrate the different tiers of a multimedia application. The NUBOMEDIA three tier model architecture introduced in Figure 1 depicts the different layers composing an application. The NUBOMEDIA PaaS system designed and developed in the context of the NUBOMEDIA project, provides support for managing and orchestrating dynamically the lower two layers of that architecture, meaning the Media Services and Application Services. Therefore, when referring to an "application", we assume always the combination between those two layers.

As already mentioned, the application logic implemented by NUBOMEDIA application developers needs to be executed on top of an application servers, previously defined in Figure 1 as Application Services layer, in order to be fully functional. Media elements, executing in the Media Services layer, shall be available and reachable from the Application Service layer. The Application Services are managed by the PaaS system, while the Media Services are managed by the NFV framework

The iD interface (depicted in Figure 2) can be consumed by NUBOMEDIA application developers to deploy applications on top of the NUBOMEDIA Cloud Platform. Those interface primitives are designed for capturing all the information that are necessary for the deployment of the Media and Application services. First of all, developers can specify information related with the Media Service layer (i.e. how many instances of media elements are required runtime, the autoscaling policies, STUN and TURN server addresses). Secondly developers could provide information about the application itself (i.e. the Git repository URL where to find the software artifacts of the application, required support services like DBMS, application name and desired number of replicas). Both the Media and Application Services layer components are executed on top of containers. After building the information model of its application, a developer can request to the PaaS Manager to deploy its application via the iD interface. Figure 3 provides a high-level overview of a typical application managed by the NUBOMEDIA Cloud Platform.

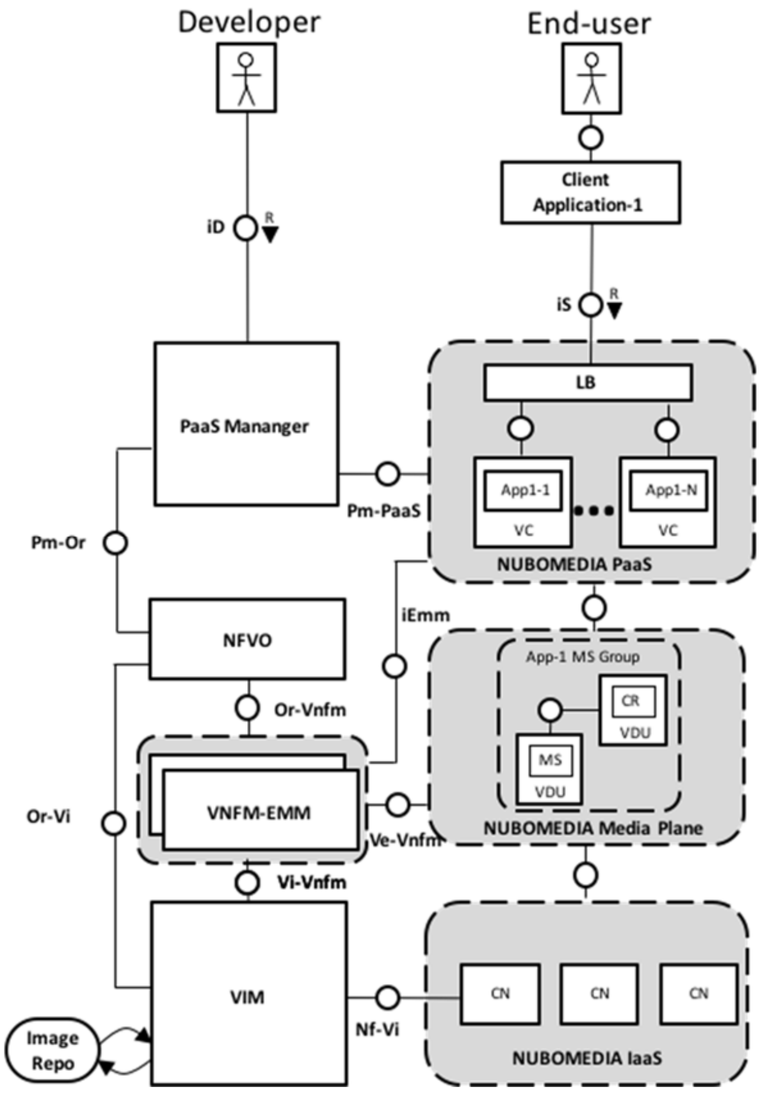


Figure 3. High-level overview of a single NUBOMEDIA Application deployed on the NUBOMEDIA Cloud Platform. The Application logic and Supporting Services (DBMS, etc) are deployed on the PaaS system, while the Media Service components are deployed on the IaaS using the NFV layers.

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

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

For scaling the Application Service layer, a HTTP Load Balancer is deployed seamlessly by the NUBOMEDIA PaaS each time an application is deployed.

More details on those procedures are going to be provided in the following sections and are also available in Deliverable D3.3[4].

Application consumption overview

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

Application runtime autoscaling overview

As already mentioned in the previous paragraph, the runtime information related to the available media server instances are provided to the upper Application Service layer dynamically. An overview of the mechanism designed for autoscaling in and out the Media Service layer is shown in Figure 4.

[4] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP3/D3.3_Cloud_Platform_R9_V1.1-04-05-2017_FINAL-PC_rev1.pdf

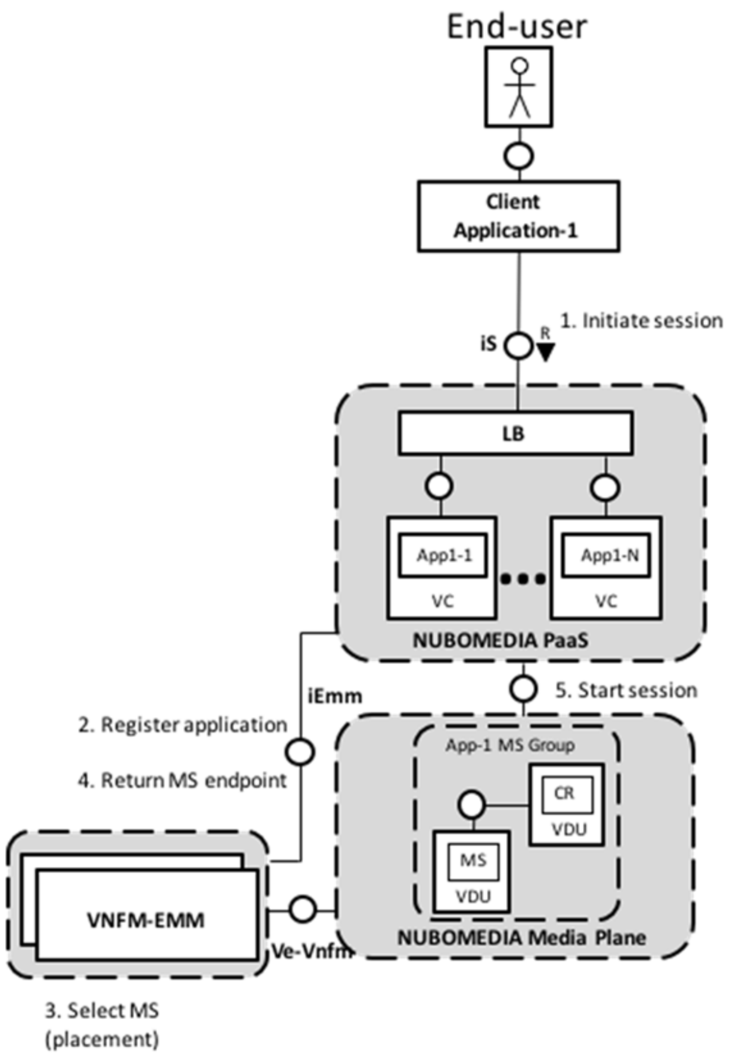


Figure 4. High level overview of the mechanism designed for auto-scaling the Media Service Layer

The Client initiates a session (step 1) via the iS interface based on the end-user request (via web browser or mobile application). In order to instantiate the required media capabilities, the application logic requires the endpoint of one media server instance. This operation is achieved consuming the iEmm interface provided by the VNFM-EMM component. In particular, the application sends a registration request (step 2) to the VNFM-EMM, which internally selects (step 3) which MS instance to use. This operation, is called session placement, because the VNFM-EMM collects information about the resource usage of the Media Plane components and returns the best suitable MS instance for the required session. Further details about this operation are provided in following sections and Deliverable D3.3[5]. Once the MS instance has been selected, the VNFM-EMM returns the MS endpoint (step 4) to the application. At that point, based on the application logic, the sessions can be started (step 5) using the media capabilities provided by the MS instance.

This mechanism allows the dynamic discovery of MS instances, therefore whenever the VNFM-EMM realizes that there are no more resources available for instantiating new sessions, it starts automatically scaling out new MS instances. This operation is

[5] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP3/D3.3_Cloud_Platform_R9_V1.1-04-05-2017_FINAL-PC_rev1.pdf

completely seamless to the application developers, since the iEmm interface is embedded in Software Development Kits (SDKs) and APIs provided to developers by the NUBOMEDIA framework.

4.3 NUBOMEDIA modules

4.3.1 The PaaS Manager

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

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

Basically, the PaaS Manager is an intermediate component combining the functionalities provided by the NUBOMEDIA PaaS System and the NFV/IaaS layers. Each time a developer requests the instantiation of an application, the PaaS Manager firstly requests the deployment of a set of Virtualized Network Functions (VNFs), defined as Network Service (refer to section 4.3.3 for more details), then instantiates the application on the PaaS System providing all the details of the media functions that are dedicated to it. As shown in Figure 5, it comprises six main modules which are going to be further detailed in upcoming sections.

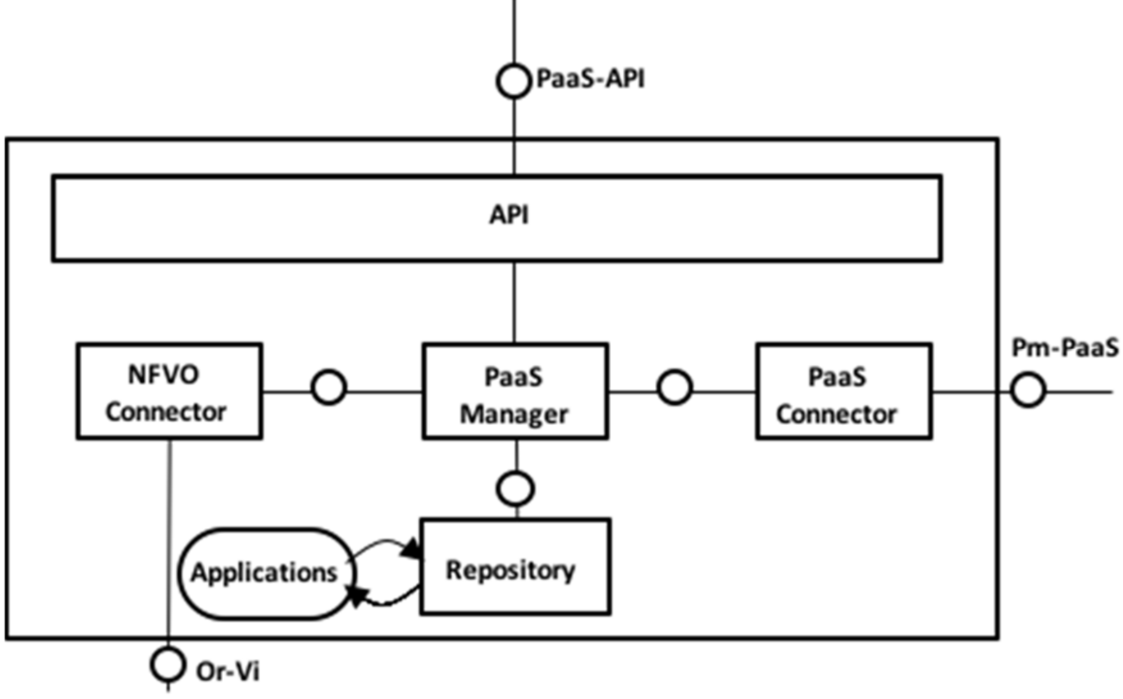


Figure 5. PaaS Manager internal architecture.

4.3.1.1 Application deployment scenario

As depicted in Figure 6, the PaaS Manager provides four main functionalities to the user interacting with NUBOMEDIA.

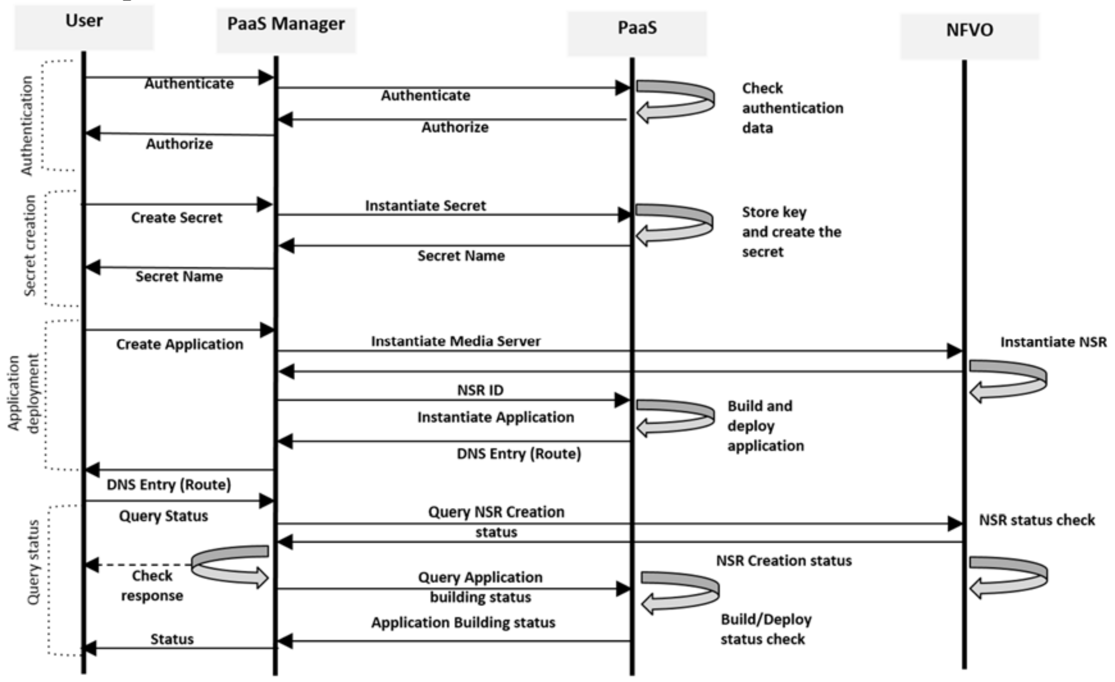


Figure 6. PaaS Manager Interactions

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

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

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

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

Looking deeply into the Application deployment, Figure 7, developers have to make a HTTP Post Request (via curl or the NUBOMEDAI PaaS GUI) providing a JSON object as body (details about the APIs are provided in Section 4.3.1.3). The PaaS manager will get that parameters from API and dispatch them to NFVO and PaaS through connectors.

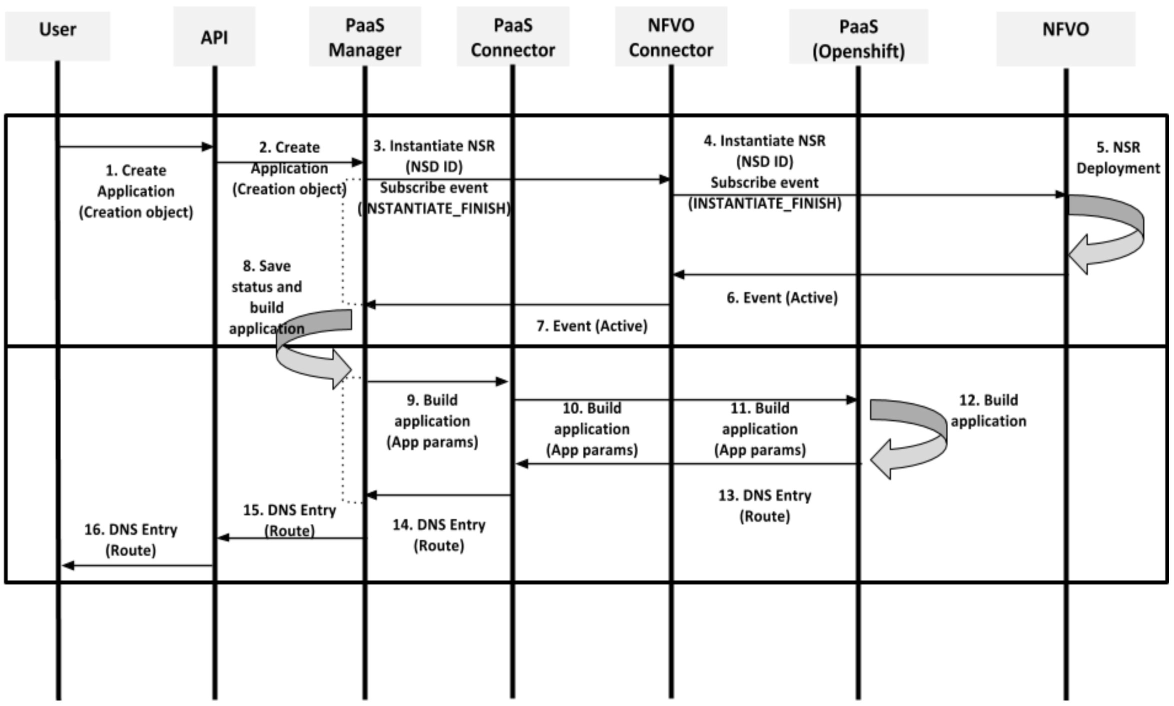

User
API
PaaS Manager
PaaS Connector
NFVO Connector
PaaS (Openshift)
NFVO
1. Create Application (Creation object)
2. Create Application (Creation object)
3. Instantiate NSR (NSD ID) Subscribe event (INSTANTIATE_FINISH)
4. Instantiate NSR (NSD ID) Subscribe event (INSTANTIATE_FINISH)
5. NSR Deployment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

Figure 7. Deployment of an application

4.3.1.2 The PaaS Manager internal modules

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

Repository

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

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

NFVO Connector

This component consumes the services of the NFVO, via the Pm-Or interface, for requesting the instantiation of the network services required by the applications for providing media transport, processing and archiving. Further details about this interface and its implementation are provided in D3.3[6].

PaaS Connector
The PaaS Connector connects the PaaS Manager to the PaaS System, via the Pm-PaaS interface, by abstracting the PaaS specific technology adopted, therefore providing an abstracted interface which can facilitate the integration of different PaaS systems. Further details about this interface and its implementation are provided in D3.3[6].

4.3.1.3 PaaS Manager API

This API abstracts the underlying complexity involved in deploying applications on the NUBOMEDIA Cloud Platform. As show on Table 2, it is based on a simple REST interface offering CRUD primitives enabling creating, updating and removing applications. For further information about this API, check NUBOMEDIA Project Deliverable D3.3 NUBOMEDIA cloud platform[6].

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

Table 2. Summary of the REST API exposed by NUBOMEDIA at the iD interface. This interface is also called the PaaS Manager API along this document and makes possible for application developers to deploy their NUBOMEDIA-enabled applications into the NUBOMEDIA PaaS.

Each individual API is further described here, while the swagger[7] definition of this interface is available at the following URL: https://raw.githubusercontent.com/nubomedia/nubomedia-paas/master/swagger.json

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

```
POST /oauth/token
```

Parameters

Type	Name	Description	Required	Schema

[6] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP3/D3.3_Cloud_Platform_R9_V1.1-04-05-2017_FINAL-PC_rev1.pdf
[7] http://swagger.io/

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

Responses

HTTP Code	Description	Schema
200	success	OAuth2AccessToken

Consumes

- application/x-www-form-urlencoded

Produces

- application/json

Create a new PaaS user

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

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

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	User

Consumes

- application/json

Produces

- application/json

Get information of a user

This method return information of a given user.

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

Parameters

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

User parameters:

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

Role parameters:

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

Responses

HTTP Code	Description	Schema
200	success	User

Consumes

- /

Produces

- application/json

Get information of all users

This method returns information of all existing users.

```
GET /api/v1/users
```

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	User list

Consumes

- /

Produces

- application/json

Delete a PaaS user

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

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

Parameters

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

Responses

HTTP Code	Description	Schema
204	success	

Consumes

- /

Produces

- /

Create a new PaaS project

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

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

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	Project

Consumes

- application/json

Produces

- application/json

Get information of a PaaS project

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

```
POST /api/v1/projects/{id}
```

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	Project

Consumes

- /

Produces

- application/json

Get information of all PaaS projects

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

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

Parameters

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

Project parameters:

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

Responses

HTTP Code	Description	Schema
200	success	Project list

Consumes

- /

Produces

- application/json

Delete a PaaS project

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

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

Parameters

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

Responses

HTTP Code	Description	Schema
204	success	

Consumes

- /

Produces

- /

Create a new application

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

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

Parameters

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

Application parameters:

Name	Description	Required	Schema
`gitURL`	the Git repository URL that contains your project (source or jar), Dockerfile and other files that are necessary to run your application. If the repository is public the link has to be the HTTPS version, if is private has to be the SSH version	true	String
`name`	It is the name of your application as you would want it to appear on the PaaS. This name is used for creating the DNS entry for your application	true	String
`cloudRepository`	Boolean value of true or false indicating if your application will be needing the cloud repository (the cloud repository is a running instance of the Kurento repository server application)	false	Boolean
`cdnConnector`	Boolean value of true or false indicating if your application will be needing the CDN Connector (the CDN Connector is a running instance of the NUBOMEDIA CDN Connector)	false	Boolean
`flavor`	This defines the size of the KMS instances. With MEDIUM flavor, you get 2 VCPU and with LARGE flavor you have 4VCPU. The capacity is defined as 100 points for VCPU.	true	String
`ports`	Indicate the transport protocol and ports exposed for the application.	false	Port list

	The port is the port exposed by the container on which the application will be running, and the target port is the external port on which the application is reachable for the outside. So, there is a mapping coming on within the PaaS for the port and target port. In principle, it can be left the port and target port the same, unless your application has special requirements.		
`replicasNumber`	It is a numeric value indicating the number of containers to be created for your application by the PaaS. This value is used for load balancing.	true	Integer
`numberOfInstances`	It is a numeric value that defines the number of KMS instances launched at the very beginning	true	Integer
`turnServerActivate`	Boolean value of true or false indicating if you want a TURN server to be set on the path of your application	false	Boolean
`stunServerActivate`	Boolean value of true or false indicating if you want a STUN server to be set on the path of your application	false	Boolean
`stunServerIp`	The IP address of the STUN server you wish to use	false	String
`stunServerPort`	The port of the SUN server you wish to use	false	String
`turnServerUrl`	The IP address and port of the TURN server you wish to use	false	String
`turnServerUsername`	The username to be used as credentials to access the TURN server	false	String
`turnServerPassword`	The password to be used as credential to access the TURN server	false	String
`scaleOutImit`	Defines the MAX number of media servers (KMS) instances that will be instantiates at runtime, e.g. by the auto scaling system	false	Integer
`scaleOutThreshold`	This is the threshold (in terms of average number of points) which will be used for the policy of the autoscaling system. Check which flavor you are going to use before defining this threshold	false	Integer
`qualityOfService`	If enabled it provides dedicated bandwidth levels between media server instances (optional). Possible values are: GOLD, SILVER,	false	String

	BRONZE		
`services`	Supporting services for providing additional services used by the main application, e.g. databases and other services provided by docker hub. Information are provided back to the main application based on the name of the application and parameters of this supporting service. The URL to the service (together with other parameters) are available in the main applications composed by the name of the service and the keys of the environment variables. In case of the URL, for instance, <NAME>_HOST	false	Service list

Port parameters:

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

Supporting Service parameters:

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

Environment Variable (EnvVar) parameters:

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

`value`	Value of the environment variable injected while service creation	true	String

Responses

HTTP Code	Description	Schema
200	success	Application

Consumes

- application/json

Produces

- application/json

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

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

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	Application

Consumes

- /

Produces

- application/json

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

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

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application will be deployed in.	true	String

Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
200	success	Application list

Consumes

- /

Produces

- application/json

Delete an application
This method deletes a given application.

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

Parameters

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

Responses

HTTP Code	Description	Schema
202	success	Application

Consumes

- /

Produces

- application/json

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

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

Parameters

Type	Name	Description	Required	Schema
Header Parameter	project-id	Defines the Project ID where the application	true	String

		will be deployed in.		
Header Parameter	Authorization	For authorization, this request must contain the token retrieved from the authorization request	true	String

Responses

HTTP Code	Description	Schema
200	success	String

Consumes

- /

Produces

- application/json

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

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

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	String

Consumes

- /

Produces

- application/json

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

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

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	String

Consumes
- application/json

Produces
- application/json

Delete a secret
This method deletes a given secret.

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

Parameters

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

Responses

HTTP Code	Description	Schema
200	success	String

Consumes
- /

Produces
- application/json

4.3.2 NUBOMEDIA PaaS System

The NUBOMEDIA PaaS System hosts the server-side application logic on top of on demand deployed Application Servers. For this, it needs to provide a set of services to the PaaS Manager, which mediates between developers and the PaaS. These services must make possible to deploy, scale and manage the application lifecycle. As shown in Figure 8, the NUBOMEDIA PaaS architecture comprises a number of building blocks, which are the following:

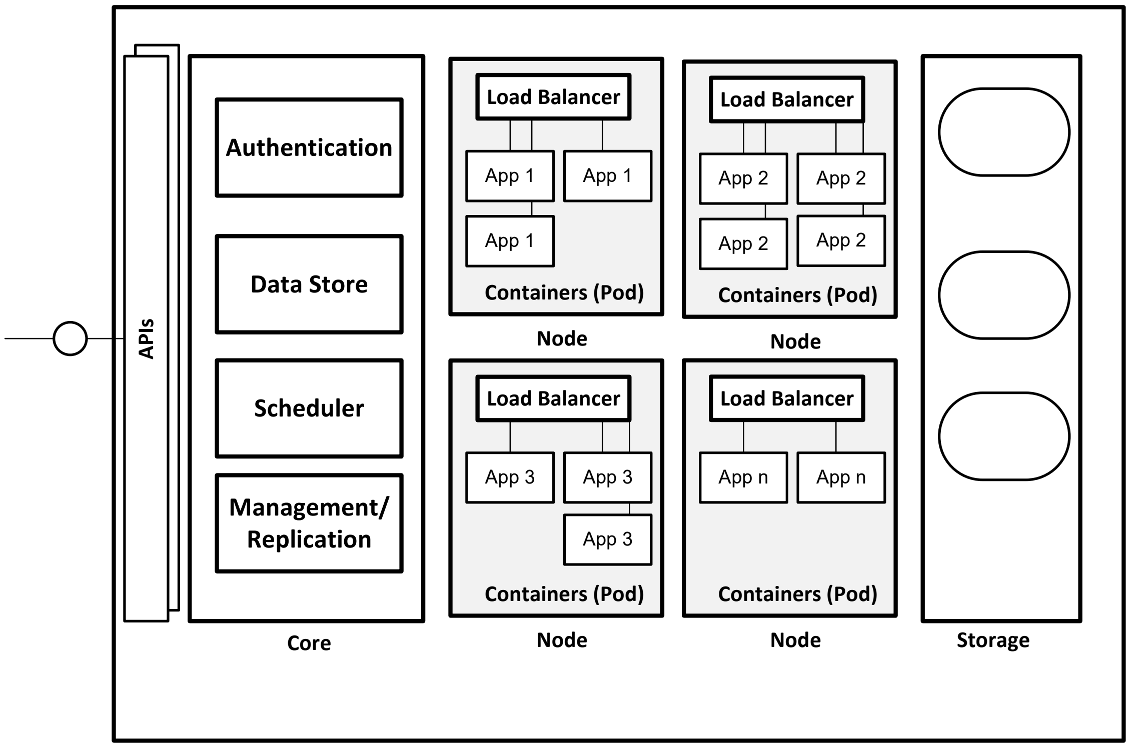


Figure 8. The NUBOMEDIA PaaS architecture

Core services. These include the following:

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

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

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

PUT	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	Replace the specified ServiceAccount

Table 3. Pm-PaaS interface definition

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

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

The relationship between Docker containers, images, and registries is depicted in Figure 9. All applications are deployed using Docker containers. A Docker container uses Docker image which is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing the application's needs (e.g. require Java Runtime Environment, etc.) and capabilities of the application. Different Images for different versions of the application can be defined and stored in the Registry and used later during deployment on different containers.

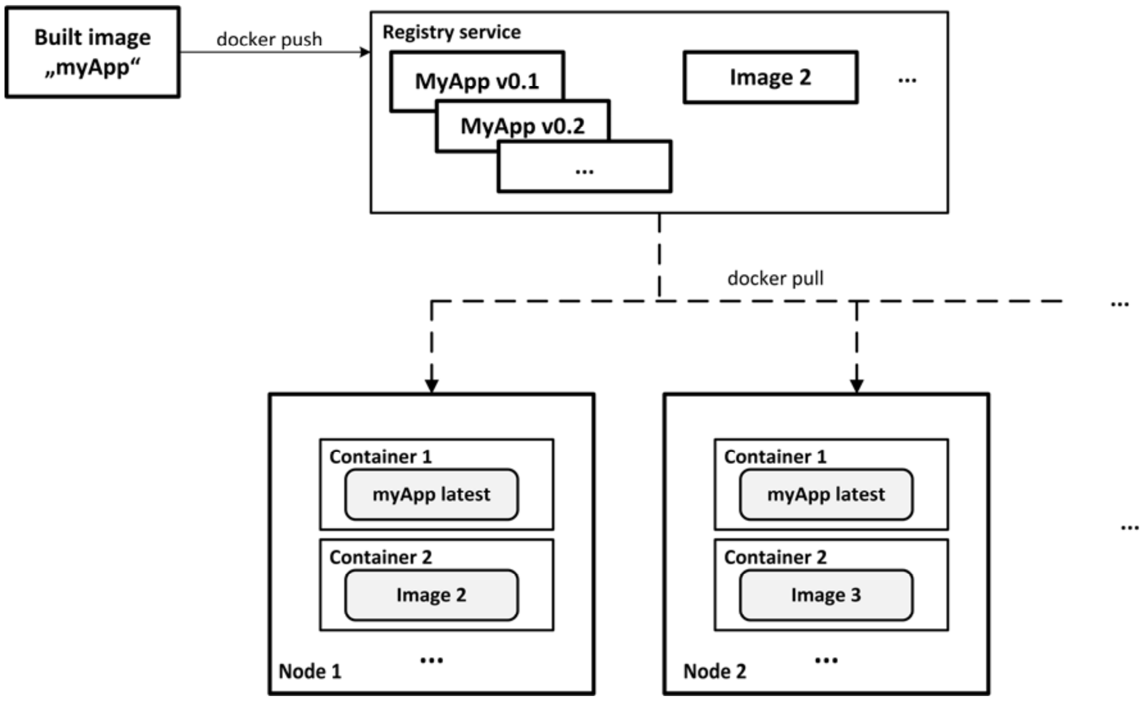


Figure 9. Relationship between Docker containers, images, and registries

4.3.3 Network Function Virtualization Orchestrator

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

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

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

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

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

This information model is internally used by the NFVO, by the VNFM and by the Virtualized Infrastructure Manager (VIM). The Figure 10, taken from the ETSI NFV MANO [MANO] specification, shows the ETSI NFV architecture for the MANO domain and specifies how the main three components are connected. As it can be noticed, NUBOMEDIA followed the ETSI NFV MANO architecture and extended it for supporting additional capabilities.

While NFVO is mainly responsible of the management of the lifecycle of the Network Services, the VNFM is responsible for the lifecycle management of a single VNF (e.g. instantiation, update, query, scaling, termination, etc.). From the NFV specification either NFVO or VNFM shall instantiate virtual resource through the VIM. As it can be observed in Figure 10, both components have an interface to the VIM. The selection of either the first or the second approach is left completely to the VNF developer, however the allocation of resources is done by the NFVO which is the only component having a complete view of available resources on the NFVI.

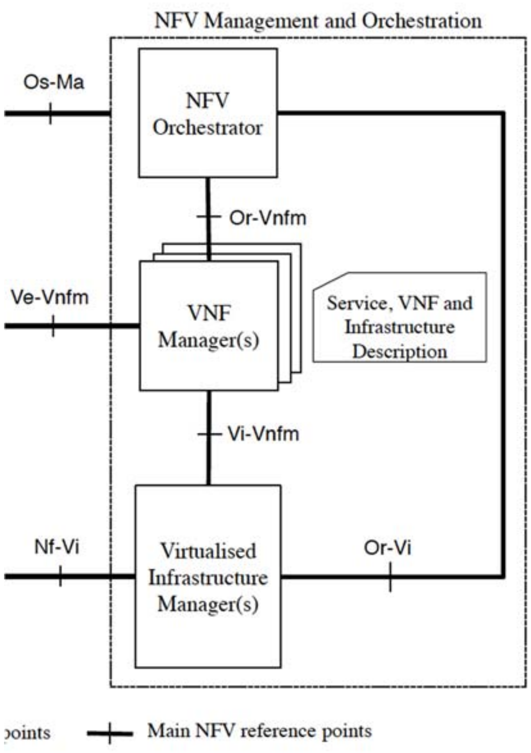

NFV Management and Orchestration
Os-Ma
NFV Orchestrator
Or-Vnfm
Ve-Vnfm
VNF Manager(s)
Service, VNF and Infrastructure Description
Vi-Vnfm
Nf-Vi
Virtualised Infrastructure Manager(s)
Or-Vi
ɔoints
Main NFV reference points

Figure 10. ETSI NFV MANO Architecture

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

As already mentioned, the NFVO manages the lifecycle of a Network Service. It exposes an interface to the PaaS Manager providing functionalities for managing the Network Service. This interface enables the following operations: 1) instantiation of the virtual computing resources (via the VIM interface) required by the Network Service, 2) provisioning of guaranteed networking resources and 3) management of the lifecycle of the different VNFs via the VNFMs. As shown in Figure 11, the NFVO comprises the following modules:

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

Method	**URL**	**Description**

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

Table 4. Pm-Or interface definition

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

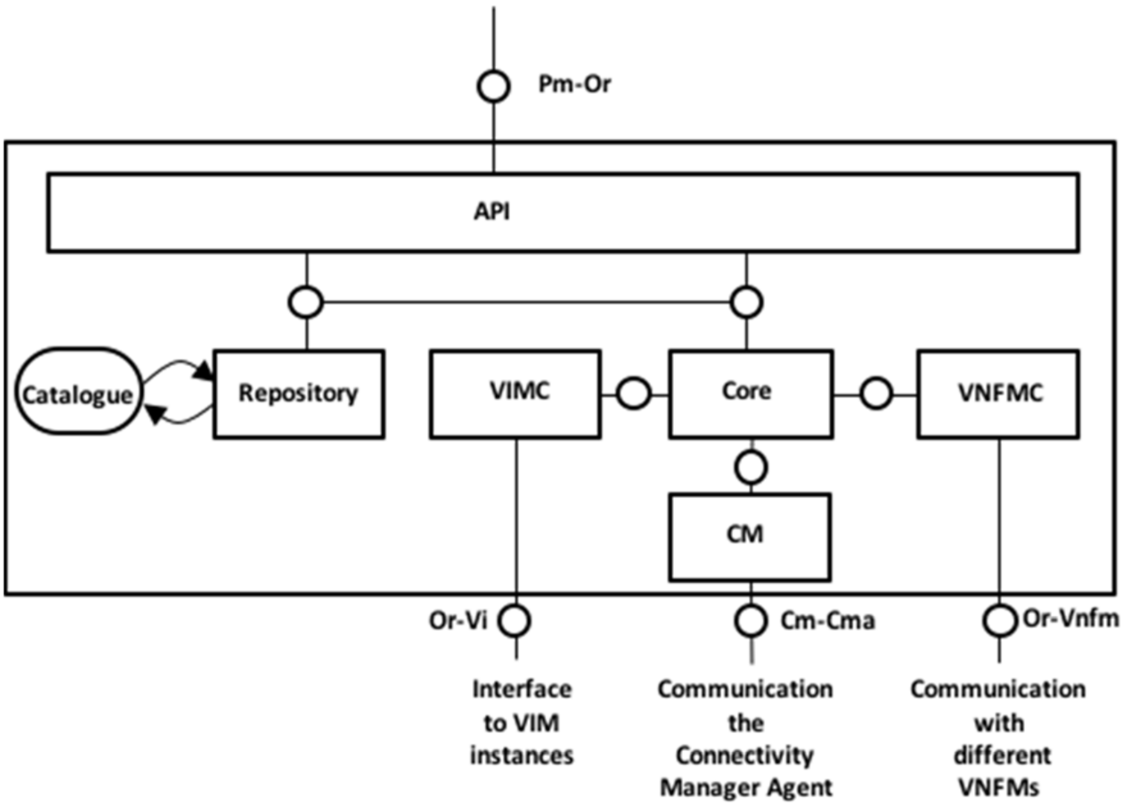

Pm-Or
API
Catalogue
Repository
VIMC
Core
VNFMC
CM
Or-Vi
Cm-Cma
Or-Vnfm
Interface to VIM instances
Communication the Connectivity Manager Agent
Communication with different VNFMs

Figure 11. NFVO internal architecture.

4.3.4 Connectivity Manager

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

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

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

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

The main concept is to let the CM extract the requirements in terms of required network capabilities of each different slice and enforce them on the NFVI making use of SDN mechanisms. Thereafter, the CM interacts with the NFVO for retrieving the requirements contained in the Network Service Record (NSR), and the list of VNFC instances used for implementing the NS.
The CMA provides a simplified API for associating a specific level of QoS required between the VNFs and corresponding VNFC instances composing the NS.

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

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

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

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

4.3.5 Virtual Network Function Manager

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

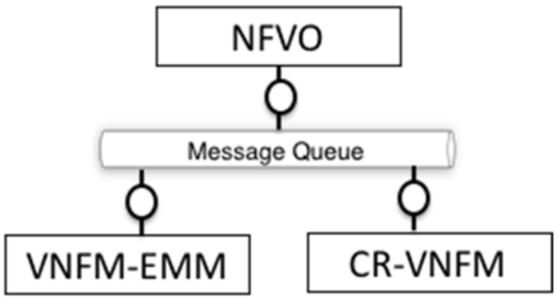


Figure 12. Following ETSI MANO architecture, the VNFM-EMM and the CR-VNFM are managed by the NFVO through a control bus

The core of the VNFM-EMM, contains the following components:

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

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

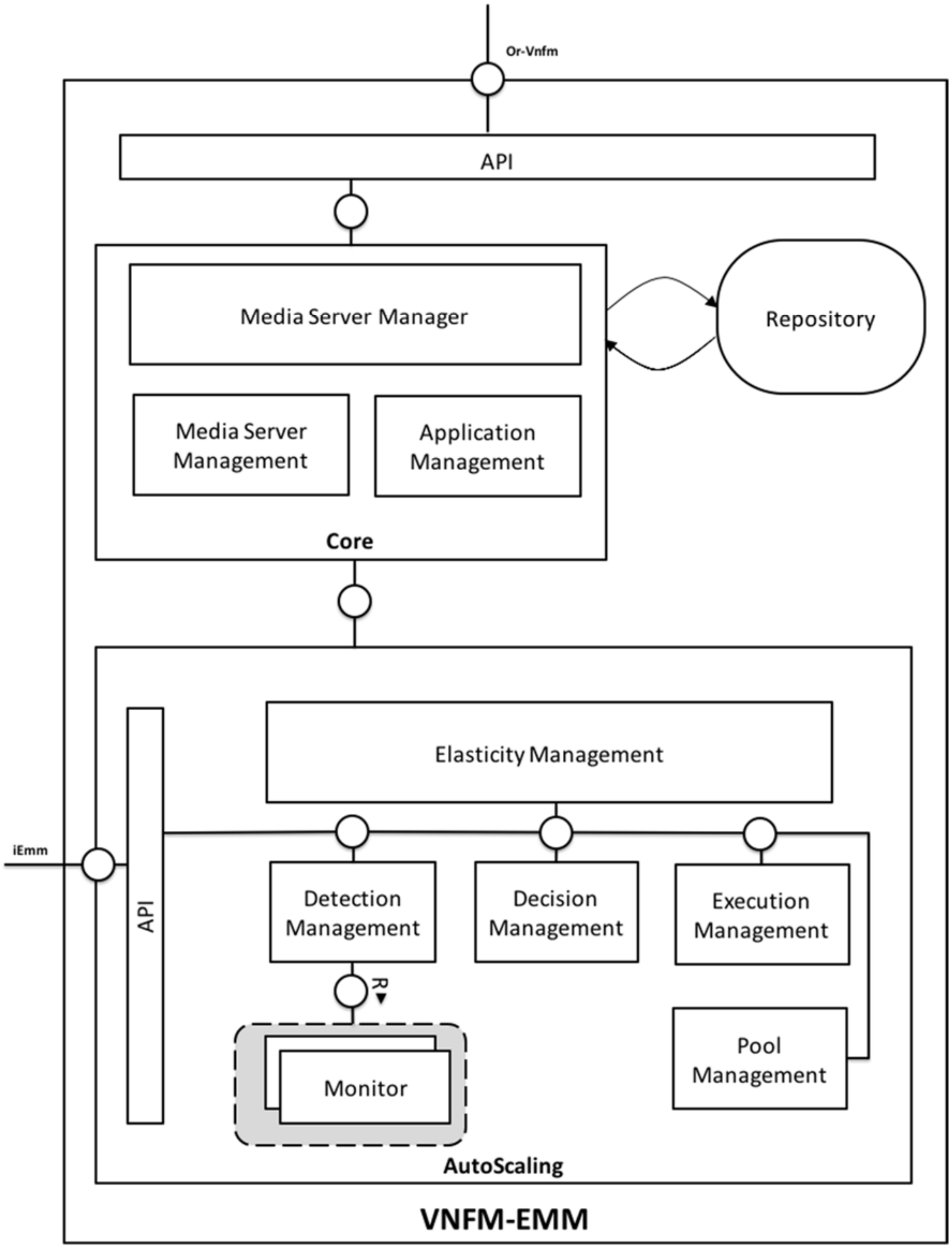


Figure 13. VNFM-EMM Architecture extended the basic VNFM with the EMM

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

4.3.6 Virtual Infrastructure Manager

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

4.3.7 Network Service Record (NSR) deployment scenario

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

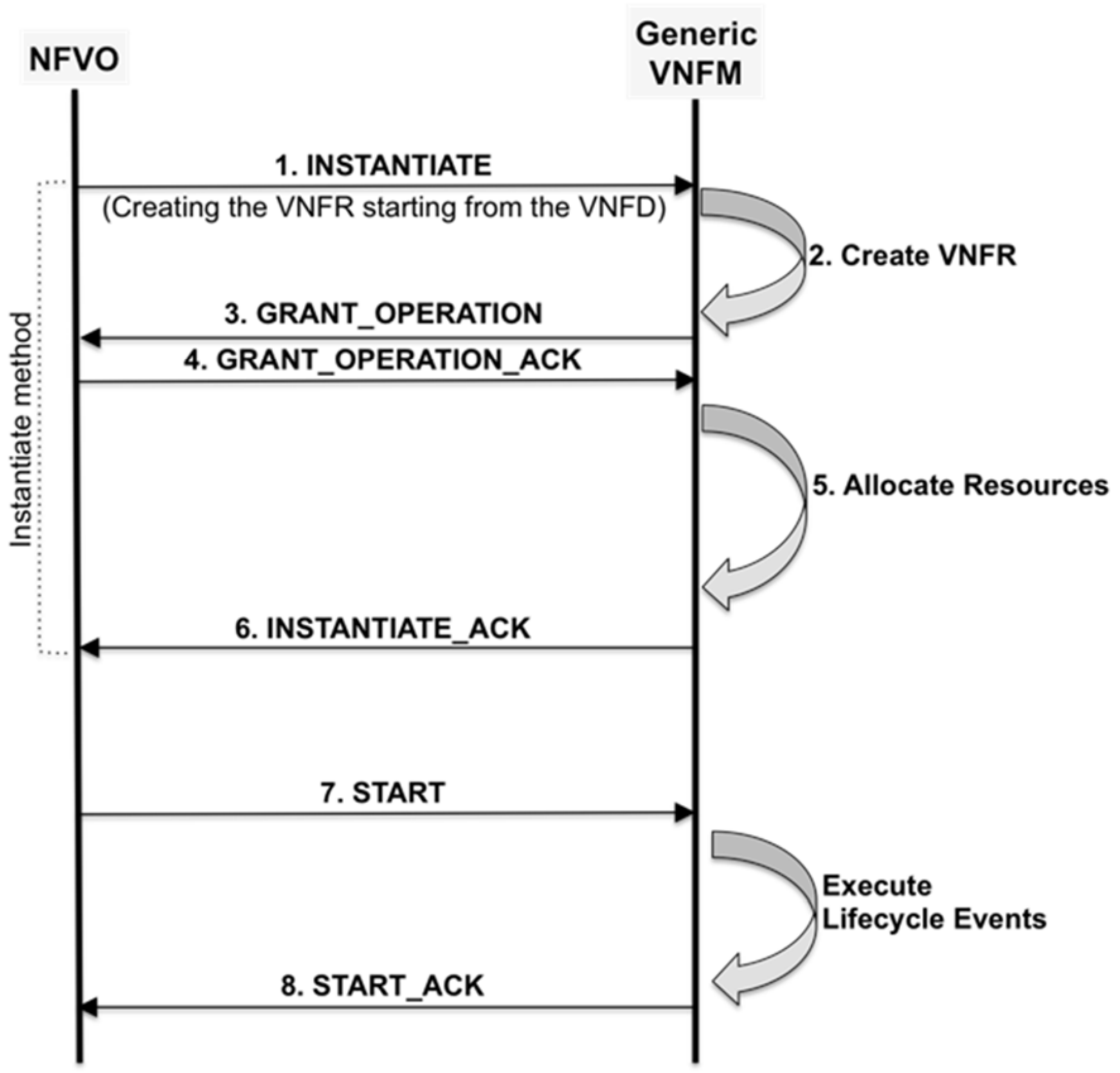


Figure 14. NFVO, VNFM and EMS sequence diagram for NSR deployment

4.3.7.1.1 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 GitHun 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).

4.3.7.1.2 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.

4.3.7.1.3 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).

4.3.7.2 Application management (Placement)

In Section 4.2 it has been introduced an overview about the autoscaling mechanism designed in NUBOMEDIA. Basically, the application discovers dynamically the media server instance which has to be used for starting a new session each time a new client connects. This mechanism is provided via the iEmm interface, registering and de-registering applications onto a particular media server instance. The idea is to "place" sessions based on their requirements, defined in terms of capacity (points) of media capabilities required.

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

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

4.3.7.2.1 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 15 depicts the whole workflow of registering a new Application. This is done as follows:

1. The EMM receives the request to register a new Application with a specific amount of points. Optionally, the external Application ID could be passed as well to ensure that the same Application will not be registered multiple times what would mean to occupy resources of Media Components multiple times by the same Application.
2. (Optional) If the external Application ID is passed in the request, check first if there is already an Application registered with that external Application ID. If there is already an Application registered with that external Application ID, use the Media Component where the Application was assigned to in the previous request. If the Application was not registered before, continue with the next step.

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

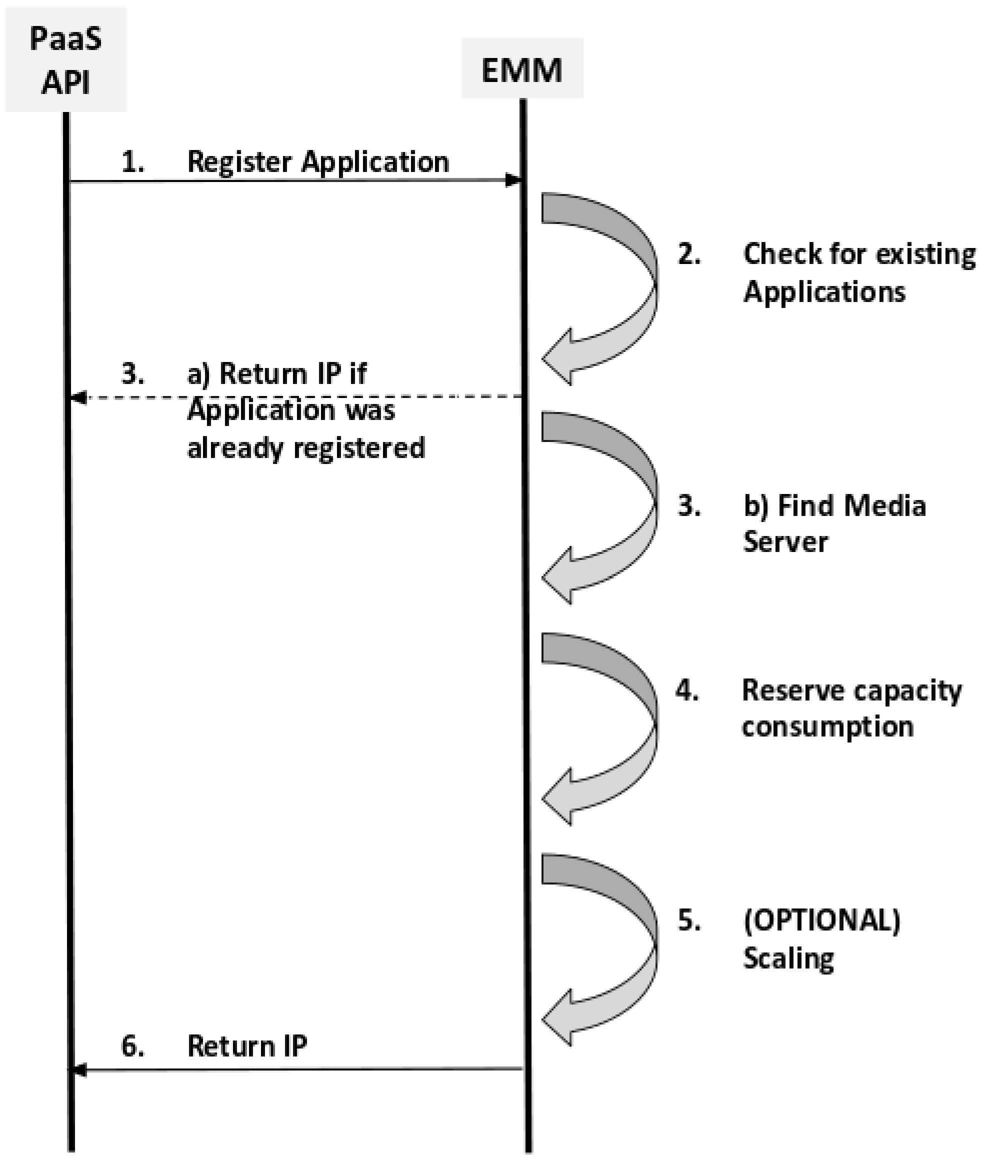

PaaS API
EMM
1. Register Application
2. Check for existing Applications
3. a) Return IP if Application was already registered
3. b) Find Media Server
4. Reserve capacity consumption
5. (OPTIONAL) Scaling
6. Return IP

Figure 15. Registration of an Application

4.3.7.2.2 Unregister an application

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

1. The EMM receives the request to unregister a specific Application.
2. First, it is checked which Media Component is currently occupied by this Application.
3. Then the Media Component; responsible for this Application, releases the resources by releasing the assigned capacity (points). Afterwards it is verified if the Media Component is still occupied by another Application. If yes, the Media Component stays in the ACTIVE pool. If not, the Media Component will be moved to the RELEASE pool.
4. (Optional) Scaling can be triggered in multiple ways by checking pool sizes and adding or removing Media Servers respectively. The following three options are available:
    a. Alarms and Actions for scaling are defined in AutoScalePolicies. Alarms are frequently checked and Actions are executed when a threshold crossing is detected. An Alarm contains the pool size as monitoring

parameter and the threshold to check at runtime. The Action contains the number of Media Server that reflects the size of the corresponding pool.

b. Events or Actions are sent to the VNF Manager at the end of the registration process. The VNF Manager is responsible for checking pool sizes and triggers scaling if needed.

c. At the end of the registration process: The EMM itself checks pool sizes and takes decisions about scaling. If scaling is needed, the EMM send a scaling request to the VNF Manager explicitly.

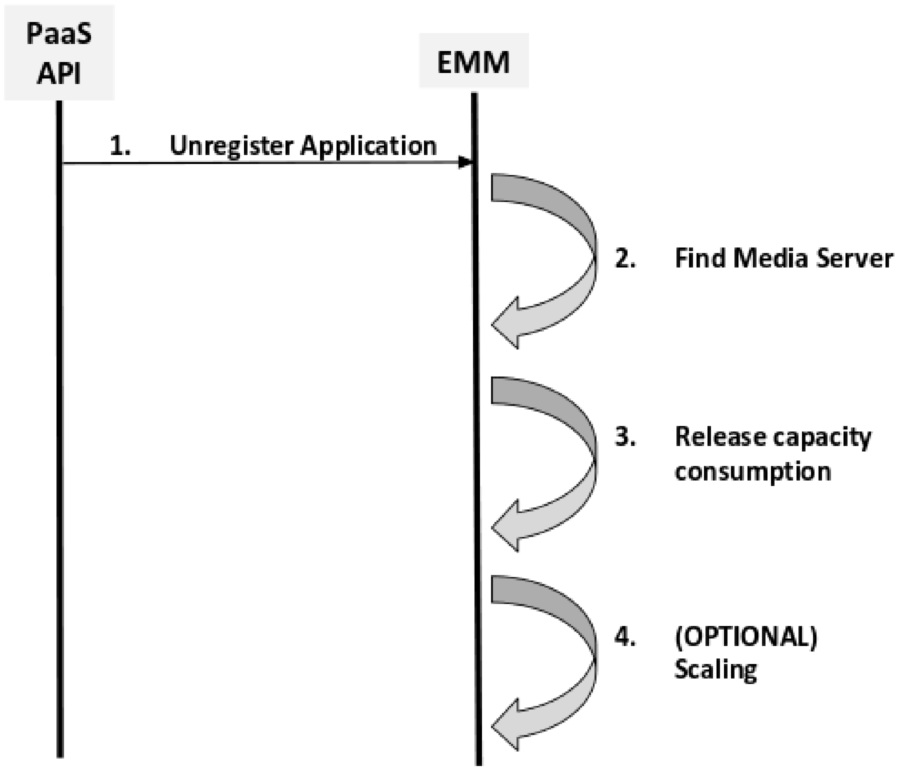


Figure 16. Unregistering an Application

4.3.7.3 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.

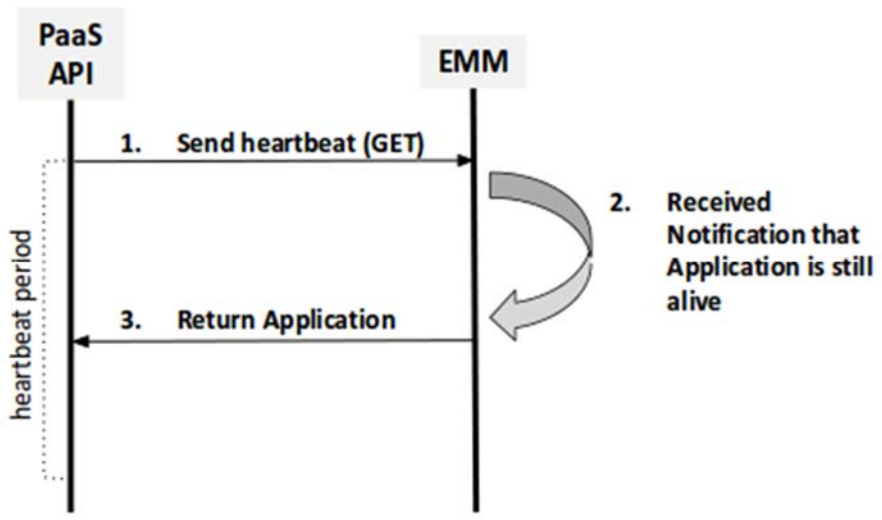


Figure 17. Heartbeat mechanism

In Figure 17 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 18) covers the case when one Heartbeat was missed already by the EMM. When the EMM receives a delayed Heartbeat, it might have started already with checking the number of sessions for removing the application automatically. Each step is more described below:

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

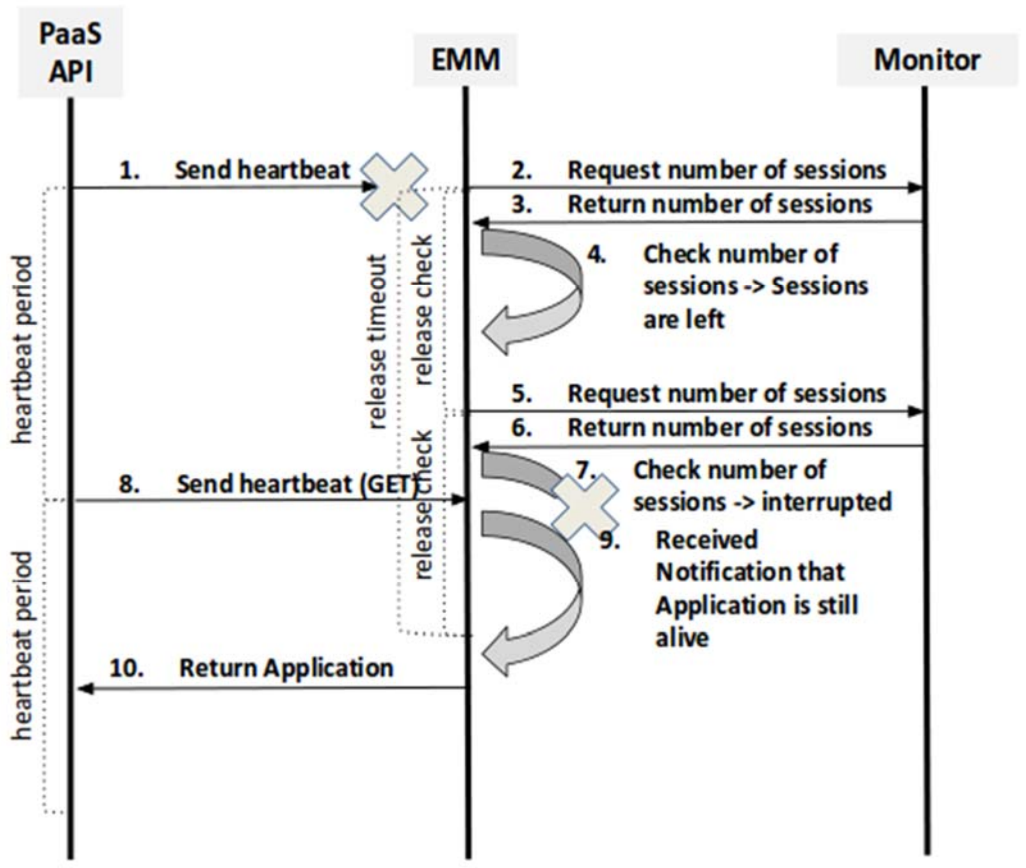

PaaS API
EMM
Monitor
1. Send heartbeat
2. Request number of sessions
3. Return number of sessions
4. Check number of sessions -> Sessions are left
5. Request number of sessions
6. Return number of sessions
7. Check number of sessions -> interrupted
8. Send heartbeat (GET)
9. Received Notification that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

Figure 18. 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 19 it is shown the first case where the Heartbeat is missed and the number of session went to 0 that is an indicator for inactivity of the Application. The steps are explained in the following:

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

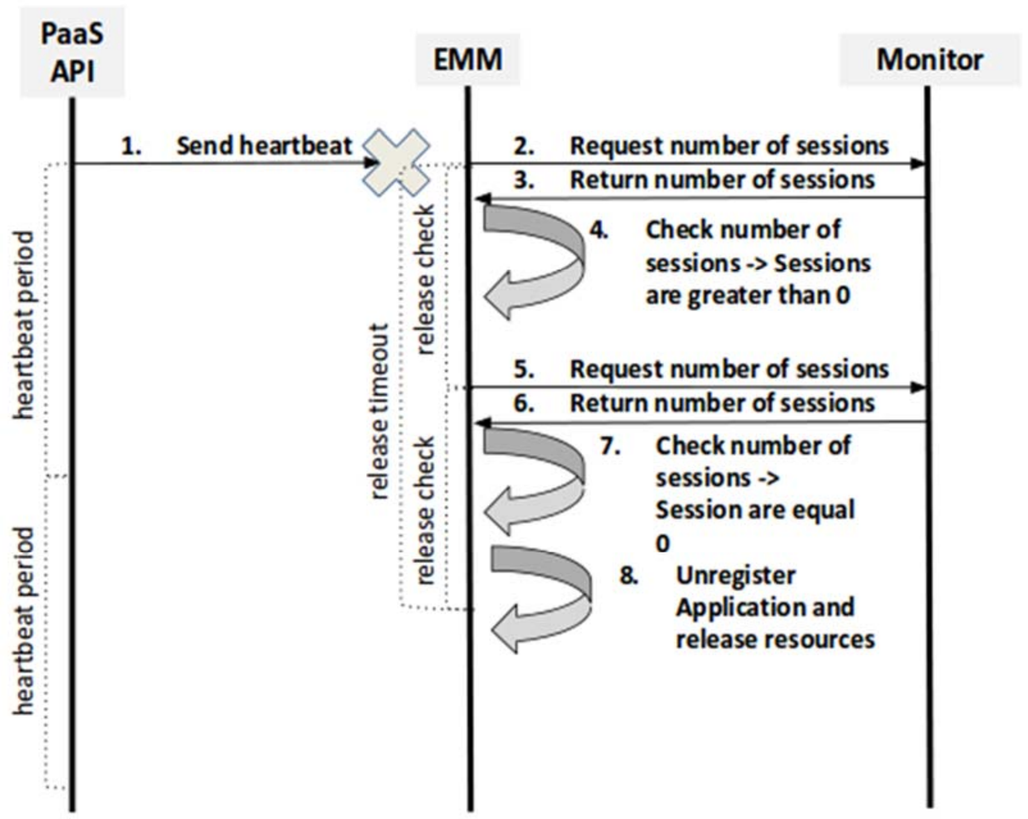

PaaS API
EMM
Monitor
1. Send heartbeat
2. Request number of sessions
3. Return number of sessions
4. Check number of sessions -> Sessions are greater than 0
5. Request number of sessions
6. Return number of sessions
7. Check number of sessions -> Session are equal 0
8. Unregister Application and release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

Figure 19. Missing Heartbeats and release check

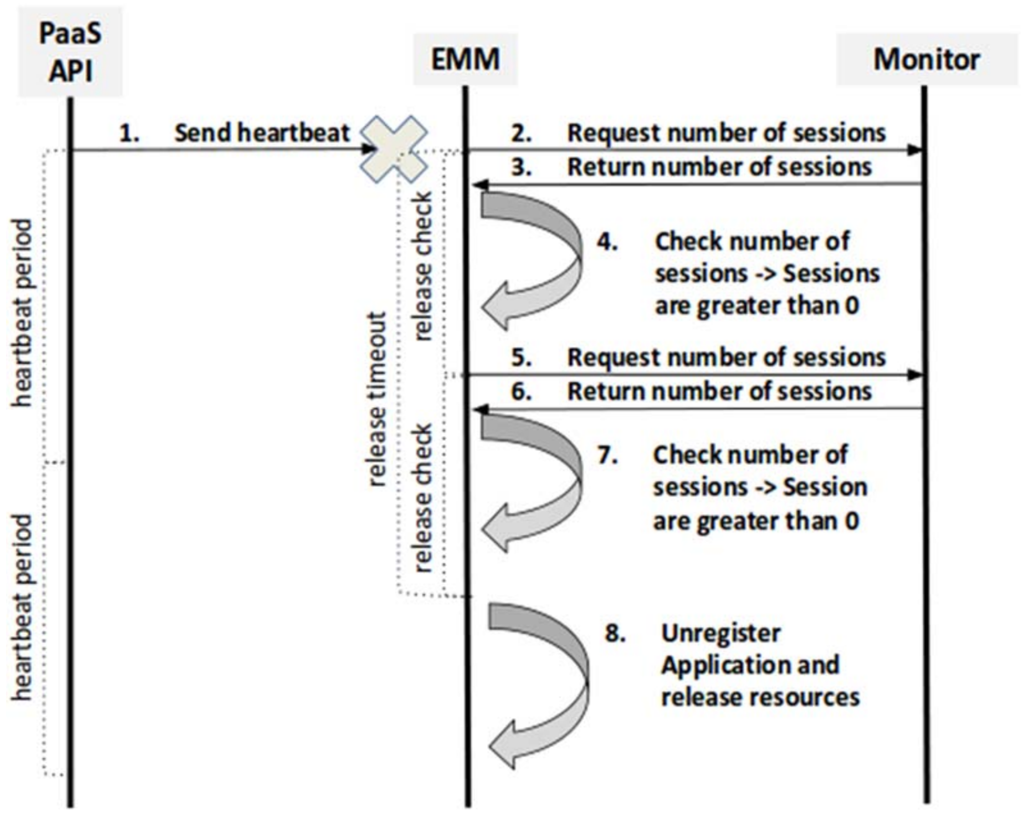

PaaS API
EMM
Monitor
1. Send heartbeat
2. Request number of sessions
3. Return number of sessions
4. Check number of sessions -> Sessions are greater than 0
5. Request number of sessions
6. Return number of sessions
7. Check number of sessions -> Session are greater than 0
8. Unregister Application and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

Figure 20. Missing Heartbeats and release timeout

In Figure 20 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.

4.3.8 NUBOMEDIA Media Plane

The NUBOMEDIA Media Plane provides the media capabilities to NUBOMEDIA applications. These capabilities include media transport, media archiving and media processing. The scalability of the NUBOMEDIA Media Plane is controlled by the NFVO component, which adapts it to the load offered by applications.

As shown in Figure 2, the NUBOMEDIA Media Plane basic component is the Media Server (MS), so that it can be seen as a collection of MS instances whose lifecycle is controlled by the NFVO and VNFM-EMM and whose relationships and interdependencies depend on the application logic. MS instances are grouped in a per-application basis so that each application is assigned, in exclusivity, a MS group. This means that each application has an independent and isolated media plane. The low-level details of the media plane are defined in deliverable D4.3, NUBOMEDIA Media Server and modules[8].

4.3.8.1 Media Server architecture

As shown in Figure 21, Media Servers hold a number of media capabilities, which may include media transport, media transcoding, media processing, media mixing and media archiving. These media capabilities are encapsulated into abstractions called Media Elements (ME), so that Media Elements hide the complexities of a specific media capability behind a comprehensive interface. MEs have different flavors (i.e. types) in correspondence with their different specialties (e.g. *RtpEndpoint*, *RecorderEndpoint*, *FaceDetectorFilter*, etc.).

[8] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP4/D4.3_NUBOMEDIA_Media_Server_V3_R9_11-11-2016-FINAL-PC.pdf

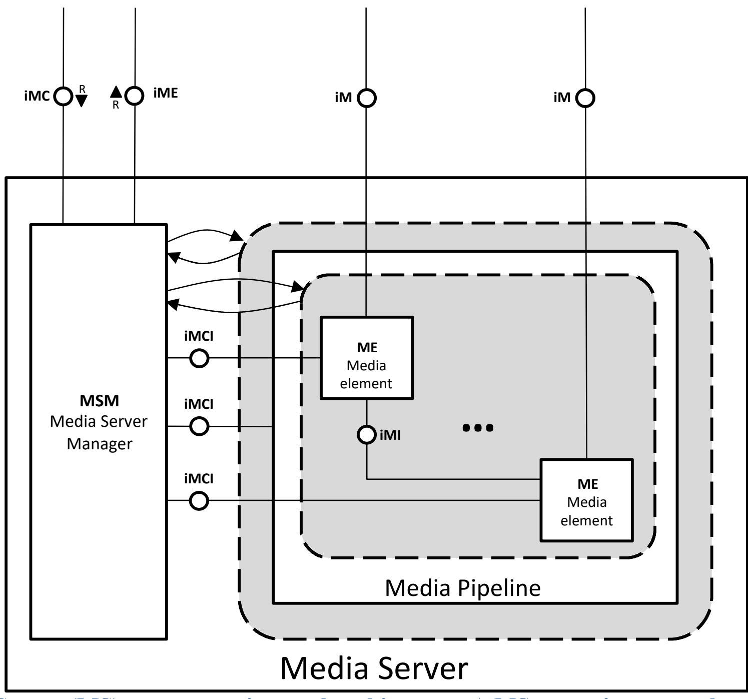


Figure 21. Media Server (MS) component internal architecture. A MS comprises a number of Media Pipelines (MPs). MPs are graphs of interconnected Media Elements (MEs). Each ME implements a specific media capability: media transport, media archiving or media processing. MS pipelines and elements are controlled by the Media Server Manager, which communicated with the external world through the Media Control interface (iMC) and the Media Events interface (iME).

The Media Element component

As shown on Figure 22, a Media Element can be seen as a black box taking media from a sink, applying a specific media capability (e.g. processing, transporting, etc.), and issuing media through a source. As a result, Media Element instances can exchange media among each other through the Media Internal interface (iMI) following this scheme:

- A Media Element can provide, through its source, media to many sink Media Elements.
- A Media Element can receive, through its sink, media from only one source Media Element.

Media Element instances can also exchange media with the external world through the Media interface (iM). Hence, the iM interface represents the capability of the Media Server to send and receive media. Depending on the Media Element type, the iM interface may be based on different transport protocols or mechanisms. For example, in a Media Element specialized in RTP transport, the iM interface shall be based on RTP. In a Media Element specialized in recording, the iM interface may consist on a file system access.

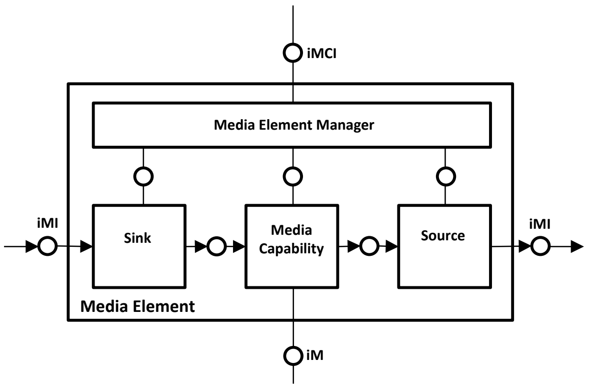


Figure 22. Media Element (ME) component internal architecture. A ME has a sink, where media may be received and a source, where media can be send. Sinks and Sources exchange media with other MEs through the Media Internal interface (iMI). ME specific capabilities are encapsulated behind an interface that it made available to the rest of MS components through the Media Control Internal interface (iMCI)

Remark, however, that not all Media Elements need to provide a iM interface implementation. Media Elements having the ability of exchanging media with the external world (and hence, providing an iM implementation) are called **Endpoints**. Media Elements which do not provide an iM implementation are called **Filters**, because their only possible capabilities are to process and transform the media received on its sink. Their architectural differences are shown on Figure 23.

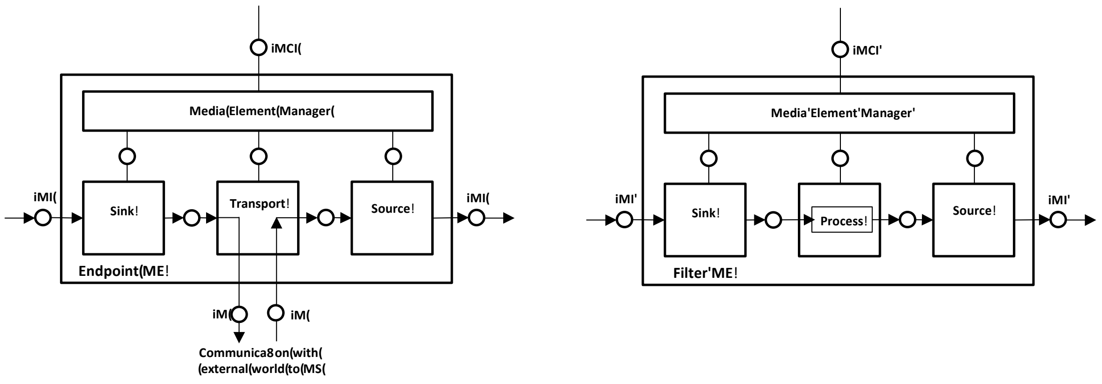


Figure 23. Endpoint MEs (left) are MEs with the capability of exchanging media with he external world through the iM interface. In an Endpoint, media received through its Sink is forwarded to an external entity (e.g. the network). Media received from the external entity is published into the Endpoint Source. Typically, Endpoint MEs publish events associated with the transport status (e.g. media-connection-established, etc.) Filter MEs (right) do not implement the iM interface and cannot exchange media with the external world. Hence, Filter MEs can only be used for processing the media received through their Sinks, which is then published to their Sources. Filter MEs typically publish events associated to the specific media processing being performed (e.g. face-detected in a *FaceDetector* filter)

The Media Element behavior is controlled by the Media Element Manager (MEM), which communicates with other MS components through the Media Control Internal interface (iMCI). The iMCI provides:

- A mechanism for sending commands to the Media Element.
- A mechanism for publishing media events and state information from the Media Element.

There is special type of Media Element called a Media Hubs. A Media Hub extends Media Elements capabilities with the ability of receiving a variable number of media streams at its input. In other words, Media Hubs are Media Elements with a variable number of sinks. Media Hubs a typically used for managing groups of streams in a consistent and coherent way.

The Media Pipeline component

Media Element capabilities can be complemented through a pipelining mechanism meaning that media issued by a specific Media Element can be fed into a subsequent Media Element whose output, in turn, could be fed with the next Media Element and so on and so forth. A graph of interconnected Media Elements is called a Media Pipeline (MP). Hence, a Media Pipeline is a set of Media Elements plus a specification of their interconnecting topology. Remark that Media Pipeline topologies do not need to be "chains" (sequences of interconnected Media Elements) but can take arbitrary graph topologies as long as the Media Element interconnectivity rules specified above are satisfied.

Hence, in a Media Pipeline one can combine in a very flexible way Media Element capabilities enabling applications to leverage different types of communication topologies among end-users, which may include any combination of simplex, full-duplex, one-to-one, one-to-many and many-to-many.

Media Pipeline capabilities are managed by the Media Pipeline Manager module, which controls the lifecycle of its composing Media Elements, forwards them commands from the external world and publishes Media Element events to subscribers. The Media Pipeline Manager uses the iMCI interface for performing these actions. A Media Server instance can hold zero or more Media Pipelines.

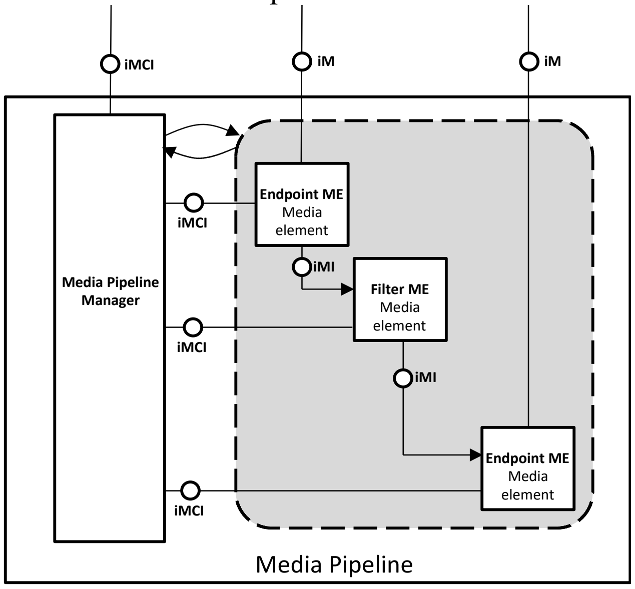


Figure 24. Media Pipelines (PMs) are graphs or interconnected Media Elements (MEs) that implement a specific media processing for an application. The Media Pipeline Manager controls the lifecycle and behavior of its MEs by brokering control messages and events among the MEs and the Media Server components our of the MP.

The Media Server Manager component

As shown in Figure 21, the Media Server Manager (MSM) component manages the lifecycle and behavior of MPs and MEs on behalf of applications. For this, The Media Server Manager exposes two interfaces to the external world:

- The Media Control Interface (iMC): This interface is based on a request/response mechanism and enables applications to send commands to both Media Pipelines and Media Element instances. Through these commands, applications can access features such as:
    - Create and delete Media Pipelines and Media Elements
    - Execute specific built-in procedures on Media Pipelines or Media Elements

    - Configure the behavior of Media Pipelines and Media Elements
    - Subscribe to specific events on Media Pipelines and Media Elements

  The Media Server Manager translates commands received on the iMC interface into invocations to the corresponding Media Elements and Media Pipelines through the iMCI interface.
- The Media Events interface (iME): This interface is based on a request/response mechanism and enables Media Element instances to send events to applications, that are then acknowledged. These events may include semantic media information (e.g. detected a face in a *FaceDetector* filter), media transport information (e.g. media negotiation completed), error information, or any other kind of information considered relevant by the Media Element developer.

The protocol for the messages exchanged between MSM and the external world in these two interfaces is based on the standard JSON-RPC v2. The JSON-RPC mechanism is a stateless, light-weight remote procedure (RPC) call protocol using JSON [RFC 4627] as data format [JSON-RPC v2].

Each request message in JSON-RPC v2 has the following fields:

- *jsonrpc*: a string specifying the version of the JSON-RPC protocol. It must be exactly “2.0”.
- *id*: unique identifier established by the client that contains a string or number. The server must reply with the same value in the response message. This member is used to correlate the context between both messages.
- *method*: a string containing the name of the method to be invoked.
- *params*: a structured value that holds the parameter values to be used during the invocation of the method.

When an RPC call is made, the server replies with a response message. In the case of a successful response, the response message will contain the following members:

- *jsonrpc*: a string specifying the version of the JSON-RPC protocol. It must be exactly “2.0”.
- *id*: this member is mandatory and it must match the value of the id member in the request message.
- *result*: its value is determined by the method invoked on the server.

In the case of an error response, the response message will contain the following members:

- *jsonrpc*: a string specifying the version of the JSON-RPC protocol. It must be exactly “2.0”.
- *id*: this member is mandatory and it must match the value of the *id* member in the request message. If there was an error in detecting the *id* in the request message (e.g. parse error, invalid request, etc.), it is equals to *null*.
- *error*: a message describing the error through the following members:
    - *code*: an integer number that indicates the error type that occurred.
    - *message*: a string providing a short description of the error.
    - *data*: a primitive or structured value that contains additional information about the error. It may be omitted. The value of this member is defined by the server.

Through the iMC, the following types of messages can be exchanged (see Table 5 for an example of request and response for each message type):

- *ping*: In order to warranty the connectivity between the client and the Media Server, a keep-alive method is implemented. This method is based on a *ping* method sent by the client, which must be replied with a *pong* message from the server. If no response is obtained in a time interval, the client is aware that the connectivity with the media server has been lost. The parameter *interval* is the time out to receive the Pong message from the server, in milliseconds. By default, this value is 240000 (i.e. 40 seconds).
- *create*: Instantiates a new media object, i.e. a Media Pipeline or a Media Element. The parameter *type* specifies the type of the object to be created. The parameter *constructorParams* contains all the information needed to create the object. Each message needs different *constructorParams* to create the object. Media Elements have to be contained in a previously created Media Pipeline. Therefore, before creating Media Elements, a Media Pipeline must exist. The response of the creation of a Media Pipeline contains a parameter called *sessionId*, which must be included in the next create requests for Media Elements.
- *invoke*: Calls a method of an existing media object. The parameter *object* indicates the *id* of the object in which the operation will be invoked. The parameter *operation* carries the name of the operation to be executed. Finally, the parameter *operationParams* has the parameters needed to execute the operation. The response message contains the value returned while executing the operation invoked in the object or nothing if the operation doesn't return any value.
- *release*: Deletes the object and release resources used by it. Release message requests the release of the specified object. The parameter *object* indicates the *id* of the object to be released. The response message only contains the *sessionId*.

	Request	Response

Ping	{
	"id": 1,
	"method": "ping",
	"params": {
	"interval": 240000
	},
	"jsonrpc": "2.0"
	}
		{
		"id": 1,
		"result": {
		"value": "pong"
		},
		"jsonrpc": "2.0"
		}

Create	{
	"id": 2,
	"method": "create",
	"params": {
	"type": "MediaPipeline",
	"constructorParams": {},
	"properties": {}
	},
	"jsonrpc": "2.0"
	}
		{
		"id": 2,
		"result": {
		"value": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline",
		"sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
		},
		"jsonrpc": "2.0"
		}

Invoke	{
	"id": 5,
	"method": "invoke",
	"params": {
	"object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/76dcb8d7-5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
	"operation": "connect",
	"operationParams": {
	"sink": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/087b7777-aab5-4787-816f-f0de19e5b1d9_kurento.WebRtcEndpoint"
	},
	"sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
	},
	"jsonrpc": "2.0"
	}
		{
		"id": 5,
		"result": {
		"sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
		},
		"jsonrpc": "2.0"
		}

Release	`{`
	`    "id": 36,`
	`    "method": "release",`
	`    "params": {`
	`        "object": "6ba9067f-cdcf-4ea6-a6ee-`
	`d74519585acd_kurento.MediaPipeline",`
	`        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"`
	`    },`
	`    "jsonrpc": "2.0"`
	`}`
		`{`
		`    "id": 36,`
		`    "result": {`
		`        "sessionId": "bd4d6227-0463-`
		`4d52-b1c3-c71f0be68466"`
		`    },`
		`    "jsonrpc": "2.0"`
		`}`

Table 5. Examples of the JSON-RPC request/response messages that can be exchanged in the iMC interface.

Through the iME, the following types of messages can be exchanged (see examples in Table 6):

- *subscribe*: Creates a subscription to an event in an object. The parameter *object* indicates the *id* of the object to subscribe for events. The parameter *type* specifies the type of the events. If a client is subscribed for a certain type of events in an object, each time an event is fired in this object, a request with method *onEvent* is sent from Media Server to the client. This kind of request is described few sections later. The response message contains the subscription identifier. This value can be used later to remove this subscription.
- *unsubscribe*: Removes an existing subscription to an event. The parameter subscription contains the subscription *id* received from the server when the subscription was created. The response message only contains the *sessionId*.
- *onEvent*: When a client is subscribed to a type of events in an object, the server sends an *onEvent* request each time an event of that type is fired in the object. This message has no *id* field due to the fact that no response is required. The request that server send to client has all the information about the event:
    - *source*: the object source of the event.
    - *type*: The type of the event.
    - *timestamp*: Date and time of the media server.
    - *tags*: Media elements can be labeled using custom information. These tags are key-value metadata that can be used by developers for custom purposes.

	Request	Response

Subscribe	`{`
	`    "id": 11,`
	`    "method": "subscribe",`
	`    "params": {`
	`        "type": "EndOfStream",`
	`        "object": "6ba9067f-cdcf-4ea6-a6ee-`
	`d74519585acd_kurento.MediaPipeline/76dcb8d7-5655-445b-8cb7-`
	`cf5dc91643bc_kurento.PlayerEndpoint",`
	`        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"`
	`    },`
	`    "jsonrpc": "2.0"`
	`}`
		`{`
		`    "id": 11,`
		`    "result": {`
		`        "value": "052061c1-0d87-4fbd-`
		`9cc9-66b57c3e1280",`
		`        "sessionId": "bd4d6227-0463-`
		`4d52-b1c3-c71f0be68466"`
		`    },`
		`    "jsonrpc": "2.0"`
		`}`

Unsubscribe	`{`
	`    "id": 38,`
	`    "method": "unsubscribe",`
	`    "params": {`
	`        "subscription": "052061c1-0d87-4fbd-9cc9-`
	`66b57c3e1280",`
	`        "object": "6ba9067f-cdcf-4ea6-a6ee-`
	`d74519585acd_kurento.MediaPipeline/76dcb8d7-5655-445b-8cb7-`
	`cf5dc91643bc_kurento.PlayerEndpoint",`
	`        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"`
	`    },`
	`    "jsonrpc": "2.0"`
	`}`
		`{`
		`    "id": 38,`
		`    "result": {`
		`        "sessionId": "bd4d6227-0463-`
		`4d52-b1c3-c71f0be68466"`
		`    },`
		`    "jsonrpc": "2.0"`
		`}`

OnEvent	
	```
	{
	  "jsonrpc":"2.0",
	  "method":"onEvent",
	  "params":{
	     "value":{
	        "data":{
	           "source":"681f1bc8-2d13-4189-a82a-
	2e2b92248a21_kurento.MediaPipeline/e983997e-ac19-4f4b-9575-
	3709af8c01be_kurento.PlayerEndpoint",
	           "tags":[],
	           "timestamp":"1441277150",
	           "type":"EndOfStream"
	        },
	        "object":"681f1bc8-2d13-4189-a82a-
	2e2b92248a21_kurento.MediaPipeline/e983997e-ac19-4f4b-9575-
	3709af8c01be_kurento.PlayerEndpoint",
	        "type":"EndOfStream"
	     }
	   }
	}
	```
		No response is needed for this message

Table 6. Examples of the JSON-RPC request/response messages that can be exchanged in the iME interface.

A detailed description of the Media Server Manager internals is shown on Figure 25.

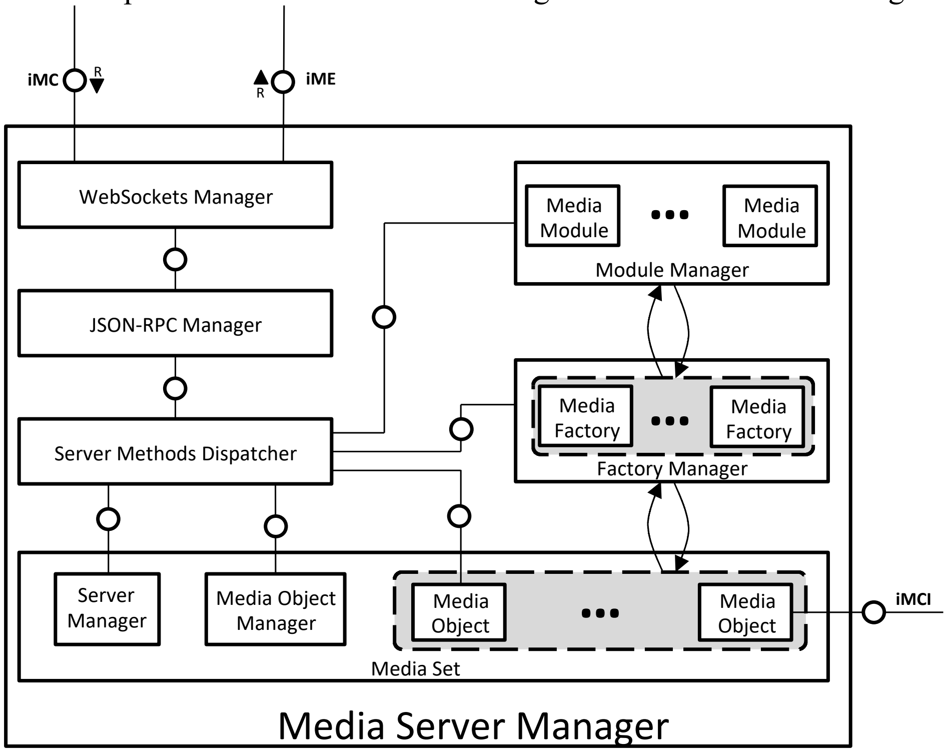


Figure 25: The Media Server Manager (MSM) holds the appropriate logic for performing two functions. The first is to enable communications with applications through the iMC control interface and the iME event publishing interface. The second is to provide the appropriate logic for dispatching, managing and providing semantics to control messages and events, including the ones in charge of media capabilities lifecycle management (e.g. create, release, etc.)

As it can be observed, MSM comprises several modules playing specific functions each:

- The WebSockets manager provides connection-oriented full-duplex communications with applications basing on the WebSocket protocol. This module is in charge of the WebSocket line protocol and of transporting data on top of it.
- The JSON-RPC Manager provides JSON-RPC marshaling and unmarshalling of messages, accordingly to JSON-RPC v2 standard.
- The Server Methods Dispatcher (aka Dispatcher) module contains the logic for routing messages to their appropriate destinations, so that the message semantics can be provided. The Dispatcher can orchestrate actions on several other modules for obtaining that semantics.

- The Module Manager is where module information is held. A module is a collection of one or several media capabilities (i.e. Media Element). All the required information for using a module in the Media Server is called a Media Module in our architecture. Media modules are loaded upon Media Server startup. The Media Module contains the module meta information (e.g. module name, module locations, etc.) The Module Manager exposes a service for querying Media Module information to the rest of modules.
- The Factory Manager controls the lifecycle of Media Factories. Upon Media Module loading, the Module Manager registers one or several Media Factories into the Factory Manager. Each Media Factory has the ability of creating a specific media capability among the ones provided by the module. This creation process is mediated by Media Stubs, so that Media Factories only create Media Stubs and are the Media Stubs themselves which create the corresponding Media Element instances.
- The Media Set holds Media Objects, which are proxies to the media capabilities themselves (i.e. the real Media Elements and Media Pipelines). It also holds a number of facilities for managing them. In particular, the Media Object Manager translates media object references (as managed in the iMC interface) into object references pointing to the appropriate Media Objects. The Media Objects provide a mechanism for mediating between the Media Server Manager and the media capabilities themselves. All requests coming from the iMC interface targeting a specific Media Element or Pipeline are routed by the Dispatcher to the corresponding Media Object, which provides it semantics consuming the Media Element or Pipeline primitives through the iMCI interface.
- The Media Object Manager also takes the responsibility of media session management. Media sessions are an abstraction provided by the iMC interface through a media session id. All JSON-RPC messages exchanged in the context of a session carry out the same id, which is unique for the session. The Media Object Manager leverages media sessions for implementing a Distributed Garbage Collection mechanism. For this, the Media Object Manager maintains the list of Media Objects associated to a specific media session. Whenever a media session is not having any active WebSocket connection during a given time period it is considered as a death session. In this case, all its Media Objects are released together with their corresponding media capabilities.
- The Server Manager is an object enabling Media Server introspection. For this, it queries the Media Object Manager to recover the list of Media Objects, which represent the Media Pipelines and Media Elements, as well as their interconnecting topologies. This makes possible for applications to monitor, instrument and debug what is happening inside the Media Server.

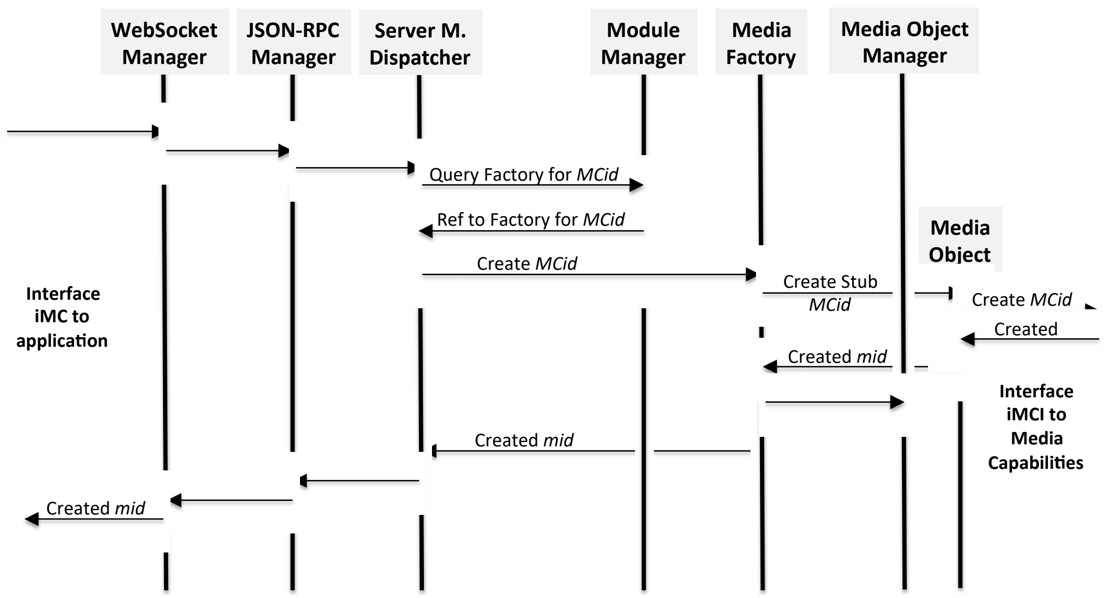


Figure 26. Flow diagram showing the interactions among the Media Server Manager modules for creating a new media capability (i.e. Media Element or Media Pipeline). The specific media capability type is identified by the string MCid (*Media_Capability_id*), that needs to be registered among modules capabilities into the Module Manager. Once the capability has been instantiated, it receives a unique id (mid or *Media_id*), which identifies it during its whole lifecycle

For a better understanding of the interactions among MSM modules lets observe Figure 26, where the sequence diagram for the instantiation of a specific media capability is shown. From top to bottom, the steps are the following:

- A message asking for media capability creating is received at the MSM WebSocket Manager. This message needs to identify the specific media capability type that is to be created through an ID, which we call MCid (*Media_Capability_id*) in the diagram.
- This message is given to the JSON-RPC Manager which unmarshalls it as a data structure.
- The unmarshalled message is given to the Dispatcher, which recognizes it as a capability creation message, and orchestrates its semantics.
- The Dispatcher queries the Module Manager for recovering the specific Media Factory knowing how to create MCid capabilities. We assume one Media Module has registered factories for MCid upon Media Server startup.
- The Module Manager provides the Dispatcher with a reference to the specific Media Factory creating MCid capabilities.
- The Dispatcher asks the Factory Manager to recover the appropriate Media Factory and commands it to create a MCid capability.
- The Media Factory Creates the appropriate Media Object associated to the MCid capability.
- The Media Object executes whatever actions are necessary for instantiating the MCid capability and for initializing it.
- The Media Factory receives the unique id of the newly created Media Object (mid).
- The Media Factory registers the new Media Object into the Media Object Manager, where it shall be recovered later when commands arrive to that specific stub.
- The mid is propagated to the application following the appropriate path (i.e. marshaling and WebSocket transport).

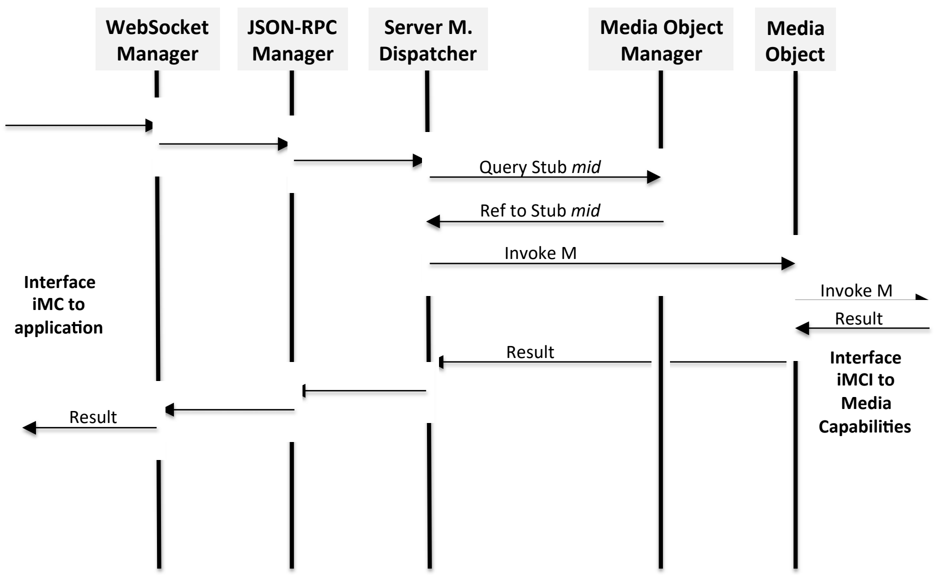


Figure 27. Flow diagram showing the interactions among the Media Server Manager modules for invoking a method (identified as M) into a specific Media Element (identified as mid)

Once a media capability (e.g. Media Element) has been created, it can be accessed and controlled following the scheme depicted in Figure 27. From top to bottom, the steps are the following:

- Requests arrive from the application through interface iMC asking for invoking a specific primitive or method (identified as M in the diagram) onto the Media Element whose id is mid.
- The requests are unmarshalled at the JSON-RPC module.
- The Dispatcher detects the request has invocation semantics and orchestrates the appropriate sequence for it.
- First, the Media Object Manager is queried for recovering the Media Object whose id is mid.
- Second, the invocation is launched to the specific Media Object holding that mid. That object knows how to reach the real Media Element and executes the appropriate actions through the iMCI interface for having the primitive executed.
- As a result, a return value may be generated. This return value is dispatched back to the application following the reverse path (i.e. marshaling and WebSocket transport).

4.3.8.2 Media Server main interactions

As introduced above, the main interactions between the Media Plane and the external world take place through three MS interfaces: iMC and iME for control and events and iM for media exchange.

Interactions through the iMC and iME interfaces take place through a request/response protocol based on JSON-RPC over WebSocket, as can be inferred by observing Figure 25. At the iMC interface, this protocol exposes the capability of negotiating media communications. The main entities and interactions associated to the iMC interface are the following:

Interacting with a Media Server: media negotiation and media exchange

In a media application involving a Media Server we identify three types of interacting entities:

- The Client Application: which involve the native media capabilities of the client platform plus the specific client-side application logic. The End-user in in contact with the Client Application.
- The Application Server: which is a container of procedures implementing the server-side application logic. The Application Server mediates between the Client Application and the Media Server in order to guarantee that the application requirements (e.g. security, auditability, dependability, etc.) are satisfied.
- The Media Server: which provides the media capabilities accessed by the Client Application.

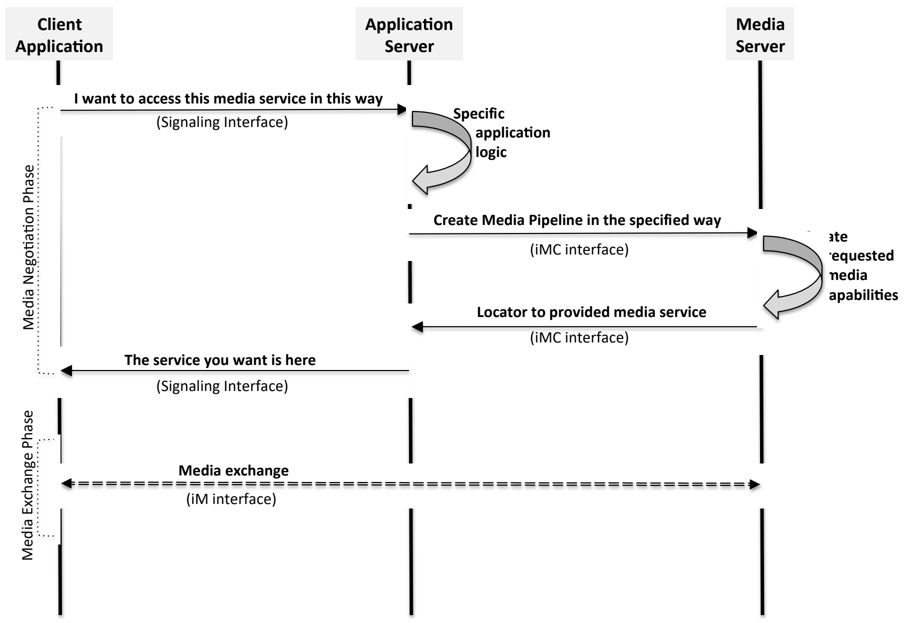

Client Application
Application Server
Media Server
I want to access this media service in this way
(Signaling Interface)
Specific application logic
Create Media Pipeline in the specified way
(iMC interface)
ate requested media apabilities
Locator to provided media service
(iMC interface)
The service you want is here
(Signaling Interface)
Media exchange
(iM interface)
Media Negotiation Phase
Media Exchange Phase

Figure 28. This diagram shows the typical entities and interactions involved in a media application based on a Media Server. In the case of NUBOMEDIA, the Application Server corresponds with the NUBOMEDIA PaaS component and, more specifically, with the container where the specific application logic is executing.

A shown in Figure 28, the interactions among these three entities typically occur in two phases: the media negotiation and the media exchange phase. The media negotiation phase requires a signaling protocol that is used by the Client Application to issue messages to the Application Server requesting to access a specific media service. Typically, those requests specify the specific media formats and transport mechanisms that the Client Application supports.

The Application Server provides semantics to these signaling messages by executing a server-side application logic. This logic can include produces for Authentication, Authorization and Accounting (AAA), Call Detail Record (CDR) generation, resource provisioning and reservation, etc.

When the Application Server considers that the media service can be provided, and following the instructions provided by developers when creating the application logic, the Application Server commands the Media Server to instantiate and configure the appropriate media capabilities. Following the discussions in sections above, in the case of NUBOMEDIA, this means that the Application Server creates a Media Pipeline holding and connecting the appropriate Media Element instances suitable for providing the required service.

To conclude, the Application Server receives from the Media Server the appropriate information suitable for identifying how and where the media service can be accessed. This information is forwarded to the Client Application as part of the signaling message answer.

Remark that during the above-mentioned steps no media data is really exchanges. All interactions have the objective of negotiating the what's, how's, where's and when's of the media exchange.

The media exchange phase starts when both the Client Application and the Media Server are aware on the specific conditions in which the media needs to be encoded and transported. At this point, and depending on the negotiated scheme (i.e. simplex, full-duplex, etc.) the Client Application and the Media Server start sending media data to the other end, where it is in turn received. The Media Server processes the received media data following the specific Media Pipeline topology created during the negotiation phase and may generate, as a result, a processed media stream to be send back to Client Applications.

Interactions with WebRTC clients
A specific relevant case for NUBOMEDIA happens when dealing with WebRTC [WEBRTC] clients. In this case, the negotiation phase takes place through the exchange of Session Description Protocol [RFC4566] messages that are send from the Client Application to the Application Server through the signaling protocol and which are forwarded then to the Media Server instance through the iMC interface.

For WebRTC to be supported, a specific Media Element needs to be created supporting the WebRTC protocol suite. Following the accepted WebRTC naming conventions [ALVESTRAND], lets imagine that we call *WebRtcEndpoint* to such Media Element. In that case, the *WebRtcEndpoint* needs to provide SDP negotiation capabilities through the iMC interface as well as any other of the WebRTC features required for the negotiation (e.g. [TRICKLE ICE], etc.)

As a result of this negotiation, an SDP answer shall be generated from the MS *WebRtcEndpoint* to the Application Server. This will be forwarded them back to the Client Application that, in turn, can use it for starting the media exchange phase. This procedure is summarized in Figure 29.

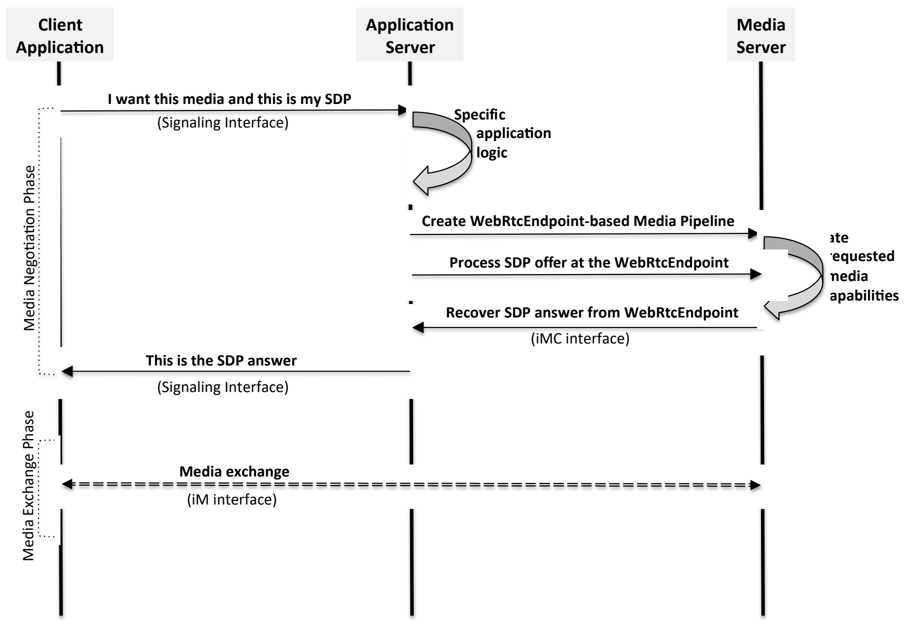

Client Application
Application Server
Media Server
I want this media and this is my SDP
(Signaling Interface)
Specific application logic
Create WebRtcEndpoint-based Media Pipeline
Process SDP offer at the WebRtcEndpoint
ate equested nedia apabilities
Recover SDP answer from WebRtcEndpoint
(iMC interface)
This is the SDP answer
(Signaling Interface)
Media Negotiation Phase
Media exchange
(iM interface)
Media Exchange Phase

Figure 29. Flow diagram schematizing the different interactions taking place for establishing a WebRTC session between the Client Application and the Media Server.

4.3.8.3 The NUBOMEDIA Media Repository

The Cloud Repository provides scalable archiving capabilities to NUBOMEDIA. The Cloud Repository is able to store Repository Items, which comprise multimedia data and metadata. Repository Items can be recovered later querying them through ID or through any kind of information present into its metadata. Repository Item metadata consists on arbitrary JSON, which is quite convenient for developers. The components of the Cloud Repository can be seen in Figure 30 and are the following:

- The Cloud Repository Manager, which is responsible of orchestrating the repository behavior and of providing semantics to the Cloud Repository API that it exposes. This API is based on REST and exposes simple CRUD operations for creating, recovering, updating or deleting Repository Items. When a new Repository Item is created, the Repository Manager enables a unique URL at the HTTP Repository Connector where the multimedia data of the item can be send through HTTP POST. When a Repository Item is queried for reading, the Repository Manager offers a unique URL at the HTTP Repository Connector for receiving it through an HTTP GET.
- The HTTP Repository Connector provides the ability of communicating multimedia information with the external world through the HTTP protocol. Media ingesting takes place through HTTP POST requests, which support Chunked and multi-part encoding. Media is send out of the repository through HTTP GET requests. Every time a media exchange is to take place, the client needs to command the repository through the Cloud Repository API and the Cloud Repository Manager commands this connector for perform the operation in a unique and on-the-fight created URL.
- The Metadata Manager is module providing metadata query management capabilities, transforming the query requests received through the repository API into native queries of the Blob Database.

The Blob Database is a persistent elastic storage where Repository Items are stored. For this, it needs to provide the ability of recording binary blobs of arbitrary size and structured JSON metadata.

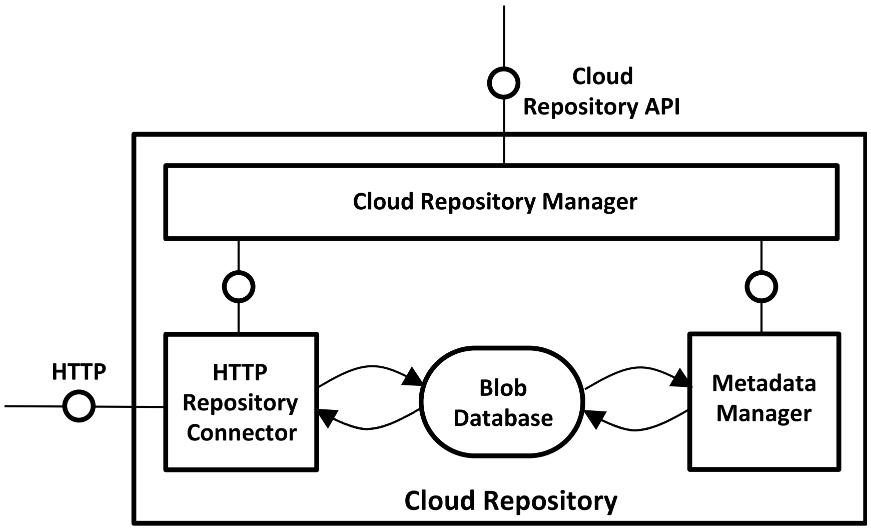


Figure 30. Cloud Repository architecture

4.3.8.4 Cloud Repository main interactions

The Cloud Repository main interactions take place through the Cloud Repository API. There are two types of main interactions: recording items into the repository and playing items from the repository. We analyze them separately.

The flow diagram for recording a new item into the repository is depicted in Figure 31. At it can be observed, it gets through three steps:

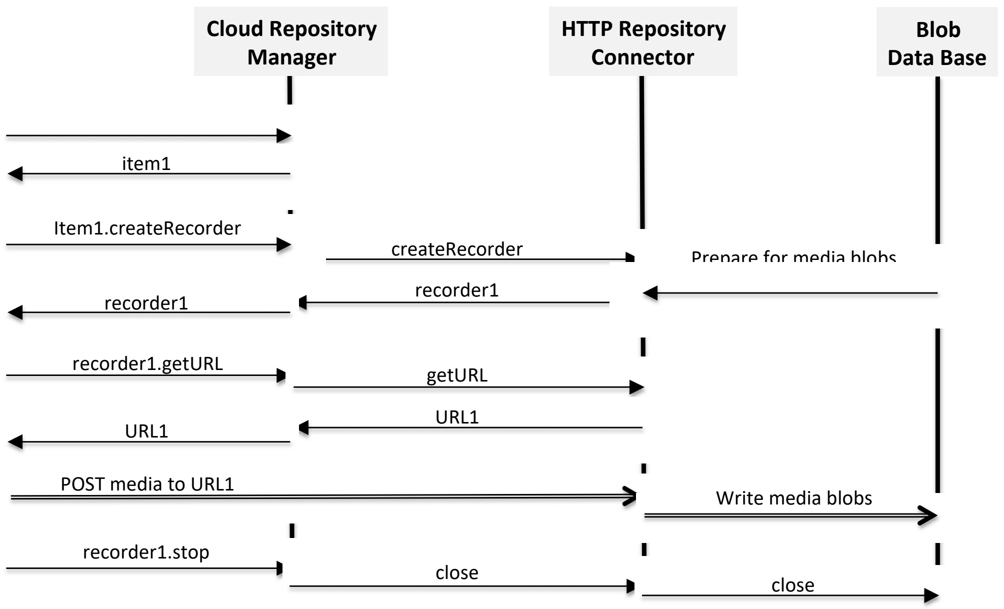


Figure 31. Main interactions for recording an item into the Cloud Media Repository.

- First, the client application uses the Cloud Repository API for creating a new Repository Item. Upon conclusion, the operation returns the unique id of the corresponding item.
- Second, the application developer can create a new recorder associated to this item. This recorder requires a number of resources in the HTTP Repository Connector, which needs to prepare for receiving the media stream and writing it to the Blob Database.
- Third, the media transfer takes place from the application to the HTTP Repository Connector using HTTP POST messages. The connector them slices the media into the appropriate blobs, which are written into the Blob Database.

The flow diagram for playing pre-recorded repository item streams is shown in Figure 32. At is can be observed, it also goes through 3 steps.

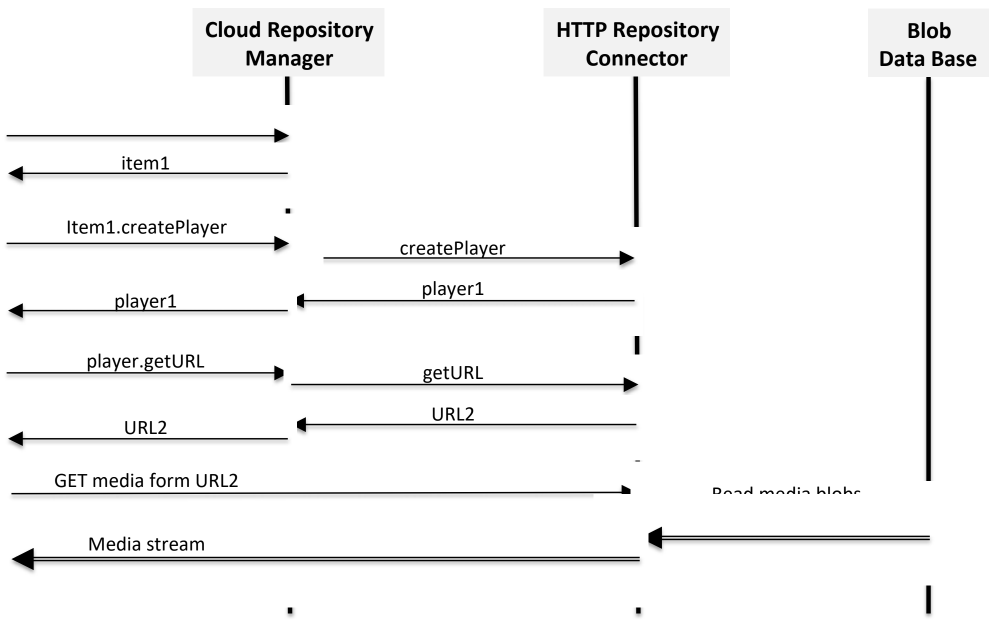


Figure 32. Main interactions for playing an item from the Cloud Media Repository.

- First, the application developer needs to obtain the id of the repository item. This can be done in different ways including querying for items whose metadata complies with a given specific pattern. Once the id has been obtained, we can recover a reference to it.
- With such reference, it is possible to create a player. The player has associated a unique URL where the media stream can be read from the HTTP Repository Connector.
- To conclude, the application can recover the media stream using an HTTP GET transaction. Upon reception of the GET request, the HTTP Repository Connector performs the appropriate operations for reading the blobs associated to the repository database in the appropriate order.

4.3.9 NUBOMEDIA IaaS

NUBOMEDIA PaaS hides the underlying complexity of an IaaS platform that enables developers to build multimedia applications without infrastructure knowledge. The elastic nature of the IaaS enables developers to build multimedia apps that scale easily with an increase of usage. IaaS capabilities are exposed through APIs that are used by the Virtual Infrastructure Manager to provide hardware resources to the upper levels of the NUBOMEDIA IaaS. Resources provided by the NUBOMEDIA IaaS are the following:

- APIs
- Core services
    - Management console
    - Compute capabilities
    - Network capabilities
    - Storage capabilities
    - Image service
- Additional capabilities
    - Metric system

- Alerting system
- Centralized logging system
- Object and Block Storage

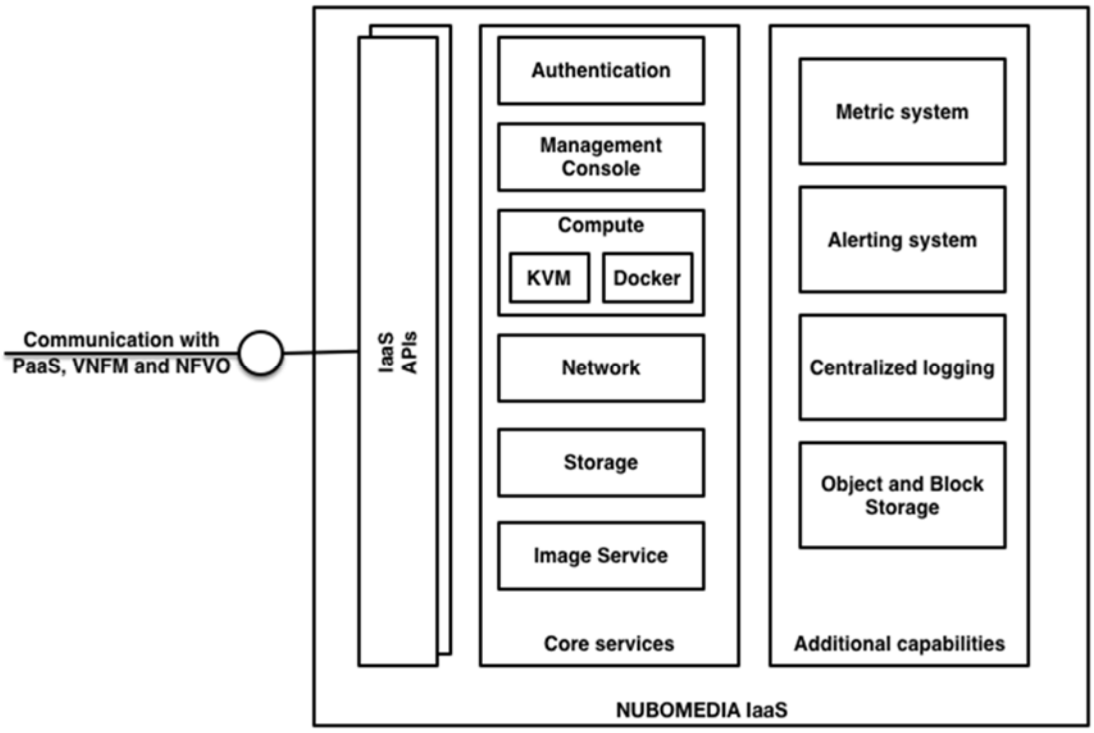


Figure 33 Functional architecture of NUBOMEDIA IaaS

4.3.9.1 IaaS APIs

Using the NUBOMEDIA IaaS APIs you can launch server instances, create images, manage networking resources, assign metadata to instances and images, create containers, push metrics to the monitoring system, forward all logs to a centralized logging system where it can be further analyzed, see the health status of the software and hardware infrastructure, and complete other actions in NUBOMEDIA cloud. Using REST APIs, the Virtual Infrastructure Manager can request hardware resources from IaaS that are consumed by the upper levels of the NUBOMEDIA PaaS.

4.3.9.2 Core services

Authentication: Interaction of the NUBOMEDIA components with the IaaS are executed through users. The IaaS user is assigned to a tenant which is the same to a project and represents a set of resource allocation in the IaaS. Every user must authenticate with either user and password or with token in order to access the system. API requests are usually made through tokens that have expire date and can be reissued on demand. Once authenticated, the role assigned to the user determines what he is authorized to do.

Management Console: The NUBOMEDIA's IaaS management console provides a web based user interface to all the IaaS components allowing the NUBOMEDIA platform administrator to manage its components and check the status of the platform.

Compute: This is the main part of an IaaS system. It is designed to manage and automate pools of computer resources and works with both KVM and Docker as hypervisors. It is written in Python and uses many external libraries such as Eventlet (for concurrent programming), Kombu (for AMQP communication), and SQLAlchemy (for database access). Compute's architecture is designed to scale horizontally on standard hardware with no proprietary hardware or software requirements and provide the ability to integrate with legacy systems and third-party technologies.

Network: The network service is made to manage networks and IP addresses inside the IaaS. It ensures the network is not a bottleneck or limiting factor in the NUBOMEDIA cloud, and gives users self-service ability, even over network configurations, allowing the platform administrator to configure the firewall for NUBOMEDIA instance in order to allow traffic only to relevant ports.

Storage: The block storage provides persistent block-level storage devices to be used on compute instances as additional disk space. The block storage system manages the creation, attaching and detaching of the block devices for both KVM and Docker instances. Block storage volumes are fully integrated into the compute service and the management console allowing the platform administrator to easy manage the storage needs. Snapshot management provides powerful functionality for backing up instances on block storage volumes. Snapshots can be restored or used to create a new instance or block storage volume.

Image Service: The image service provides discovery, registration, and delivery services for disk server images and container images. Stored images can be used as a template.

4.3.9.3 Additional capabilities

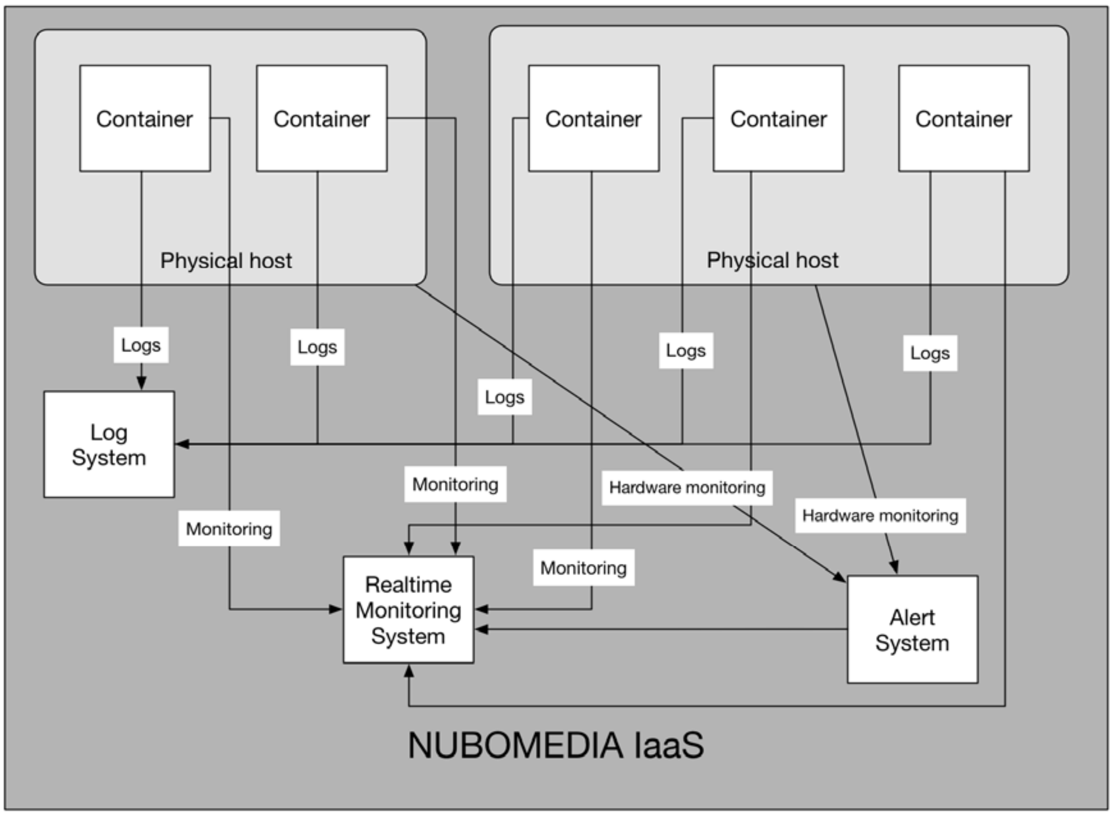


Figure 34 Architecture of additional capabilities

Metric System: The metric system gathers statistics about the system as it is running and stores this information. Those statistics can then be used to find current performance bottlenecks (i.e. performance analysis) and provide this information to the

VNFM though it's API so it can take decisions whether to scale up or down resources inside the NUBOMEDIA platform.

Alerting system: It monitors the entire NUBOMEDIA infrastructure from physical components to service health to ensure everything is functioning properly. In the event of a failure, the alarm system can alert technical staff of the problem, allowing them to take actions fast, before it affects end-users, or customers.

Centralized logging system: The logging system is a tool for keeping all the Docker instances events and logs centralized, so they can be further accessed by the platform manager of even by the NUBOMEDIA user. We use it to collect logs, parse them, and store them for later use. All the logs are stored in a time series database in order to allow fast searching on them. The system also provides an interface where you can view and analyze them in a flexible analytics and real-time way.

Object and Block Storage: The object and block storage system consists on a NoSQL database which is configured to take advantage of load balancing and data replication features over multiple machines for storing files.

5 NUBOMEDIA Development APIs

As specified in sections above, from the developer's perspective, NUBOMEDIA capabilities are accessed through a set of APIs inspired in the three-tier development model, as depicted in Figure 1. Hence, for creating NUBOMEDIA enabled applications developers just need to understand these NUBOMEDIA Development APIs so that they may use them for creating their rich RTC media applications. The NUBOMEDIA Development API stack is architected following the scheme depicted in Figure 35. As it can be observed, this stack offers a complete set of APIs that can be classified in three groups that we call: Media Capabilities APIs, Signaling APIs and Abstract Communication APIs.

The following sections in this deliverable presents a high-level description of each of the NUBOMEDIA APIs. For further details please read deliverable D5.3, NUBOMEDIA framework APIs and tools[9].

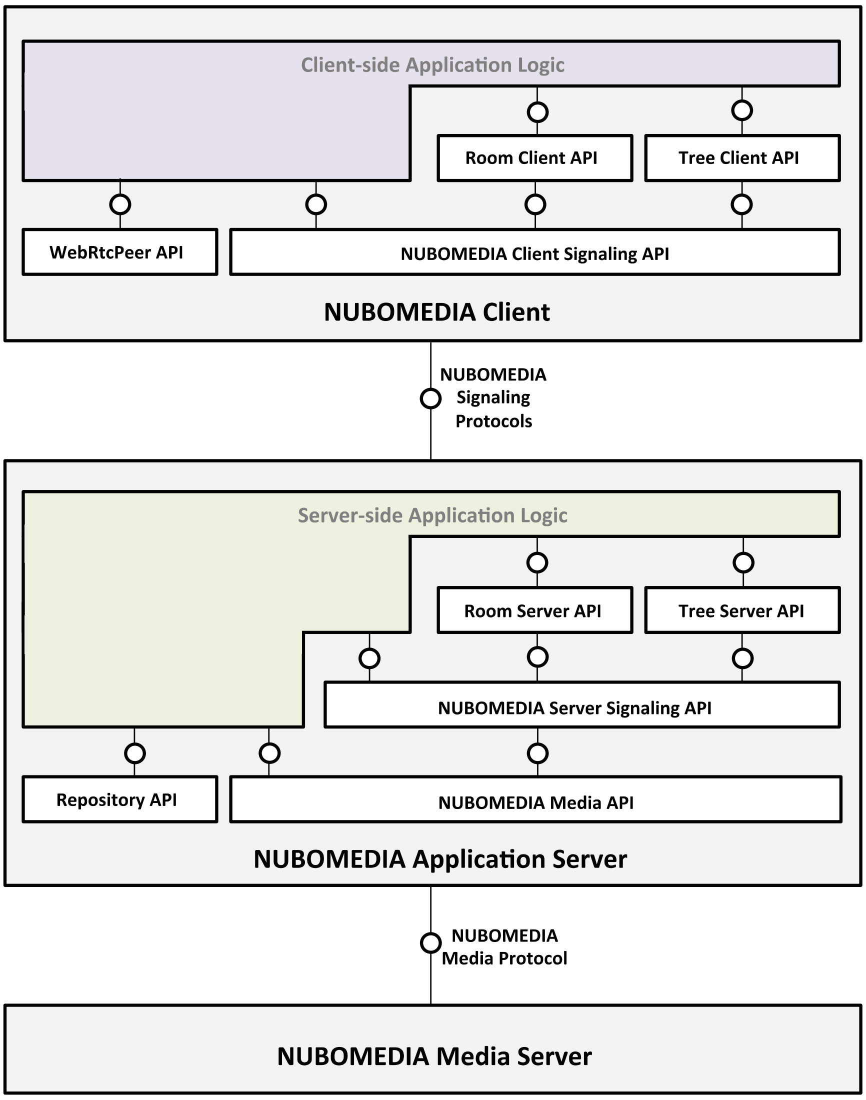

Client-side Application Logic
Room Client API
Tree Client API
WebRtcPeer API
NUBOMEDIA Client Signaling API
NUBOMEDIA Client
NUBOMEDIA Signaling Protocols
Server-side Application Logic
Room Server API
Tree Server API
NUBOMEDIA Server Signaling API
Repository API
NUBOMEDIA Media API
NUBOMEDIA Application Server
NUBOMEDIA Media Protocol
NUBOMEDIA Media Server

Figure 35. Architectural diagram showing the NUBOMEDIA API stack which comprises three types of APIs: Media Capability APIs (NUBOMEDIA Media and Repository APIs), Signaling APIs (JSON-RPC over WebSocket client and server APIs), and Abstract Communication APIs (Room and Tree client and server APIs). In this picture, the NUBOMEDIA Media Protocol corresponds to the iMC and iME interfaces described in Figure 21, while the NUBOMEDIA Signaling Protocols correspond to the iS interface described in Figure 2.

[9] https://raw.githubusercontent.com/nubomedia/additional-documentation/master/WP5/D5.3_NUBOMEDIA_framework_APIs_R9_30-11-2016_FINAL-PC.pdf

5.1 Media Capabilities APIs

These APIs expose to the application logic the low-level media capabilities of NUBOMEDIA. These capabilities are basically the ones offered by the NUBOMEDIA Media Server and the NUBOMEDIA Media Repository, as described in sections above. As a result, this group includes two APIs:

- The NUBOMEDIA Media API, which enables developers consuming the NUBOMEDIA Media Server capabilities among which we can find media transport, media archiving, media processing, media transcoding, etc. This API is based on two main concepts: Media Elements and Media Pipelines. This API is shipped for developers as an SDK abstracting the iMC and iME interfaces specified in Figure 21.
- The NUBOMEDIA Repository API, which makes possible to access an elastic scalable media repository for archiving media information and meta-information. It is based on the notion of Repository Item: an object with a unique identity which may contain media data and meta-data. This API is shipped for developers as an SDK abstracting the REST Cloud Repository API specified in Figure 30.

In addition, in this group we also include a NUBOMEDIA API abstracting the client side media capabilities for developers. In current state-of-the-art, these capabilities basically correspond to WebRTC and comprise both the ability of capturing and rendering media plus the ability of communicating RTC media. In NUBOMEDIA, we have abstracted all these capabilities behind the WebRtcPeer API.

5.2 Signaling APIs

The Media Capabilities APIs introduced above are signaling agnostic, meaning that they neither require nor assume any kind of specific characteristic for signaling. Hence, NUBOMEDIA capabilities can be accessed through any kind of signaling protocol including SIP, XMPP or REST. However, for simplifying common development tasks, in our architecture we propose a simple signaling protocol based on JSON-RPCs that is suitable for most applications not requiring specific interoperability features. This protocol has been wrapped through an abstract API which enables full-duplex signaling exchanges between Application Server and Clients. To this aim, the API is split into two complementary parts:

- The NUBOMEDIA Client Signaling API. This is a client-side API enabling the creation of JSON-RPC sessions and the sending and receiving of messages through them.
- The NUBOMEDIA Server Signaling API. This is a server-side API enabling the creation of JSON-RPC sessions and the implementation of server-side business logic for messages following a request/response communication scheme.

5.3 Abstract Communication APIs

Developers typically use RTC media capabilities for creating applications devoted to person-to-person communications. In this case, the application business logic needs to manage the communication topologies among participants in an RTC multimedia session. In the general case, this logic needs to be programmed by the application developers. However, there are a number of communication topologies that are quite popular and appear systematically in applications. Due to this, in our architecture, we propose specific communication API abstracting the low-level details of the media logic on these topologies so that developers may use them in an agile and efficient manner. In particular, we have identified two of these common topologies: Room Topology and Tree Topology. For them two specific APIs have been implemented:

- The Room API. This API has two complementary sides, the Room Server and the Room Client APIs. Through them, this API makes possible for developers to create applications basing on Room Topologies. Room topologies are based on the intuitive idea of a physical room where people may communicate. If you enter into the room you may talk to others and others may talk to you. Hence, the main characteristic of a Room Topology is that the multimedia session enables a full-duplex communication exchange of media among all participants. Hence, the session is called informally room and all room members may publish their media streams to the room or subscribe to the streams of the rest of room members.
- The Tree API. Like the Room API, it also has Server and Client sides. However, this API is based on a Tree Topology. In our context, a tree can be sees as a one-way (i.e. simplex) mechanism providing one-to-many RTC media. The interesting aspect of this topology is that the number of leaves of the tree may be large, which makes convenient for providing real-time media broadcasting capabilities to large audiences.

The NUBOMEDIA API stack architecture above mentioned is the result of a complex design process where we have prosecuted to enable simplicity without scarifying flexibility and expressiveness. This stack has been created for abstraction, understood as the ability of making simple things simple and complex things possible for developers. For this, the rationale behind its design is based on a number of principles application developers need to understand in order to choose the appropriate APIs for each implementation objective.

In NUBOMEDIA, the most abstract APIs are the Room and Tree APIs. These are abstract because they hide most of the low-level details of the media complexities and expose to developers' high-level notions such as “Rooms” and “Trees”, that are logical objects suitable for managing media transport through specific communication topologies. The NUBOMEDIA internal use-cases analysis (see NUBOMEDIA Project Deliverable D2.1.2), as well as the accumulated experience of developers worldwide out of NUBOMEDIA evidence that Room and Tree topologies are at the base of a significant fraction of RTC media services. Hence, developers wishing to create applications just providing room group communications or tree one-to-many media broadcasting services can use these APIs directly without requiring any further understanding on the rest of the API stack.

However, there is still a fraction of developers for which the Room and Tree APIs do not fit. These may include the ones with specific interoperability requirements (e.g. SIP or legacy RTP) or needing special features (e.g. custom media processing, non-common communication topologies, non-linear media logic, etc.) In that case, lower-level NUBOMEDIA APIs might be needed.

The NUBOMEDIA Signaling APIs makes possible for developers to create custom signaling protocols in a seamless and efficient way. Hence, this API might be useful whenever specific signaling mechanisms beyond rooms and trees are required. Just for illustration, lets imagine a video surveillance application with the ability of detecting intruders in a specific area of the camera viewport. Whenever an intruder is detected an alarm needs to be fired to all the connected clients, so that they may rewind the streams and visualize it again for assessing the severity of the incident.

Clearly, this type of logic cannot be provided through the Room or Tree APIs, which do not have alarm-sending capabilities nor the ability of seeking the media for a playback. Hence, developers creating this application needs a custom signaling protocols. This developer may use the signaling protocol she wishes given the PaaS nature of NUBOMEDIA. However, among the available options, a natural choice is the NUBOMEDIA Signaling API. This API makes possible to create a custom signaling protocol just be defining some simple JSON-RPC messages, which is quite convenient given the familiarity of developers with this format. Once this is done, the API makes straightforward to send such messages and to implement the appropriate business logic upon their reception. The only limitation of this scheme is that the API mandates the protocol to be based upon JSON-RPC over WebSocket transport. Hence, whenever this restriction is not an impediment, the NUBOMEDIA Signaling API shall be useful for crating specific and customized signaling mechanisms.

The NUBOMEDIA Media API, in turn, exposes the low-level media capabilities. Through this API developers can create arbitrary and dynamic topologies combining them with media processing, transcoding or archiving. The Room and Tree topologies are just particular cases of what this API can provide. Hence, mastering this API is a must for all developers wishing to take advantage of all NUBOMEDIA features. This API is complemented with the Repository API which enables media data and metadata persistence onto an elastic scalable repository. Hence, the Repository API may be of help whenever large amounts of media information need to be recorded and recovered.

All in all, and as Figure 35 shows, developers are free to combine the NUBOMEDIA APIs without restrictions so that for example, an application may consume at the same time the Room API, the Signaling API and the Media API if it is needed.

References

[ALVESTRAND] https://tools.ietf.org/html/draft-ietf-rtcweb-overview-14

[ETSI_WP] ETSI. Network functions virtualisation - introductory white paper, October 2012. At the SDN and OpenFlow World Congress, Darmstadt Germany.

[FMC] http://www.f-m-c.org

[FMC REFRENCE] FMC notation reference. http://www.fmc-modeling.org/notation_reference

[JSON-RPC v2] http://www.jsonrpc.org/specification

[MANO] 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

[RFC4566] https://tools.ietf.org/html/rfc4566

[RFC2119] Key words for use in RFCs to Indicate Requirement Levels. https://www.ietf.org/rfc/rfc2119.txt

[RFC2616] Hypertext Transfer Protocol -- HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html

[RFC 4627] The application/json Media Type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627.txt

[TRICKLE ICE] https://tools.ietf.org/html/draft-ietf-mmusic-trickle-ice-02

[WEBRTC] http://www.webrtc.org