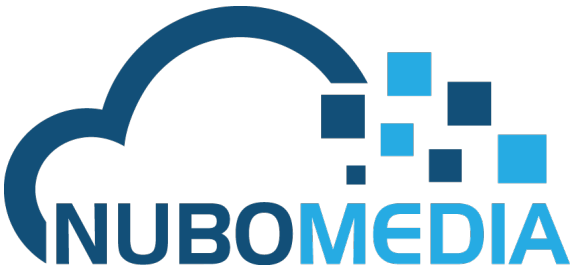
D3.3.1	
Version	1.0
Author	TUB
Dissemination	PU
Date	27/01/2015
Status	Final



D3.3.1: NUBOMEDIA Cross-Layer Connectivity Manager v1

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

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

FP7 ICT-2013.1.6. Connected and Social Media

Contributors:

Giuseppe Antonio Carella (TUB)
Julius Müller (TUB)
Jakub Kocur (TUB)
Benjamin Reichel (TUB)

Internal Reviewer(s):

Luis López (URJC)

Version History

Version	Date	Authors	Comments
0.1	04/2014	Julius Müller, Giuseppe Carella	Created initial version of the document
0.2	07/2014	Julius Müller	Added content into the State of the Art section
0.3	09/2014	Benjamin Reichel, Jakub Kocur	Added Functional Architecture
0.4	01/2015	Benjamin Reichel	Added Software Architecture and installation guide
1.0	01/2015	Giuseppe Carella, Benjamin Reichel	Finalized and reviewed all the document

Table of contents

List of Figures:

Acronyms and abbreviations:

CDPI	Control to Data-Plane Interface
CM	Connectivity Manager
CPI	Cloud Provider Interface
EMM	Elastic Media Manager
NBI	Northbound Interface
OFP	OpenFlow Protocol
ONF	Open Networking Foundation
QoS	Quality of Service
SDN	Software Defined Network
SLA	Service Level Agreement
VM	Virtual Machine

1 Executive summary

This document provides an overview of the Connectivity Manager (CM). In particular this deliverable focuses primarily on its software architecture, providing a detailed description of which functionalities have been implemented.

2 Introduction

The Connectivity Manager (CM) is part of the NUBOMEDIA platform and is placed between the virtual network resource management of the cloud infrastructure and the multimedia application. The main focus of the CM is related to management and control of network functions of the virtual network infrastructure provided by OpenStack. NUBOMEDIA applications require different types of network capabilities. The CM ensures the implementation of the requirements on the cloud infrastructure and decides whether and where to place an application inside the cloud.

In this document, the state of the art of virtual network infrastructure control and management is discussed as well as the specific cloud infrastructure OpenStack with its network components used in this project. After that, in Section 4, we investigate different ways of integrating the functional requirements for this project and we analyze the available options looking for a final decision. Based on this decision the design and implementation is created. In Section 6 the functional architecture is discussed. In 7 the software architecture is described and in 8 the API is presented. In the annex the installation guide and the throughput evaluation of the current OpenStack testbed can be found.

3 State of the Art

First of all, we give a brief introduction of existing technologies for the management and orchestration of cloud applications.

3.1 Software Defined Network Concepts

Traditional network architectures are ill suited to fulfill today's requirements for new applications and network services. Changing traffic patterns, the "consumerization of IT", the rise of cloud services and processing "Big data" are only a few examples of new network criteria that arise in the last years and need to be solved urgently since the hierarchical networks, built with tiers of Ethernet switches organized in a tree structure are not capable to handle dynamic computing and storage needs of today’s network services. The complexity to solve these problems leads us to stasis regarding the huge amount of protocols, inconsistent policies, inability to scale and the vendor dependencies.

The Open Networking Foundation (ONF) is a user-driven organization, which develops a new network architecture to overcome these challenges. This new architecture is called Software-defined Networking and is one of the most promising approaches.
Software-defined Networking (SDN) is an emerging architecture where the network control is decoupled from forwarding functions. The network intelligence is logically centralized and this gives us the opportunity to program the network control directly without taking care about the underlying infrastructure that is completely abstract for applications and network services.
Today's static networks will be transformed into flexible, programmable platforms with much more intelligence to allocate resources dynamically for instance.
Thanks to this approach, it makes the new network architecture dynamic, highly-automated, secure, cost-effective, adaptable and ideal for high-bandwidths, and that is

why it fits optimal for today's and future requirements. SDN developers say that SDN is on the way to become the new norm for networks.

3.2 SDN Architecture and Protocols

The following section is going to describe the key architectural principles of SDN. This **architecture** (shown in figure Figure 1) decouples the network control and forwarding functions. Furthermore, it provides open interfaces to the application layer to enable an easy development of applications that are independent from the underlying infrastructure. This gives us the possibility to control the connectivity provided by a set of network resources and the flow of network traffic through them.

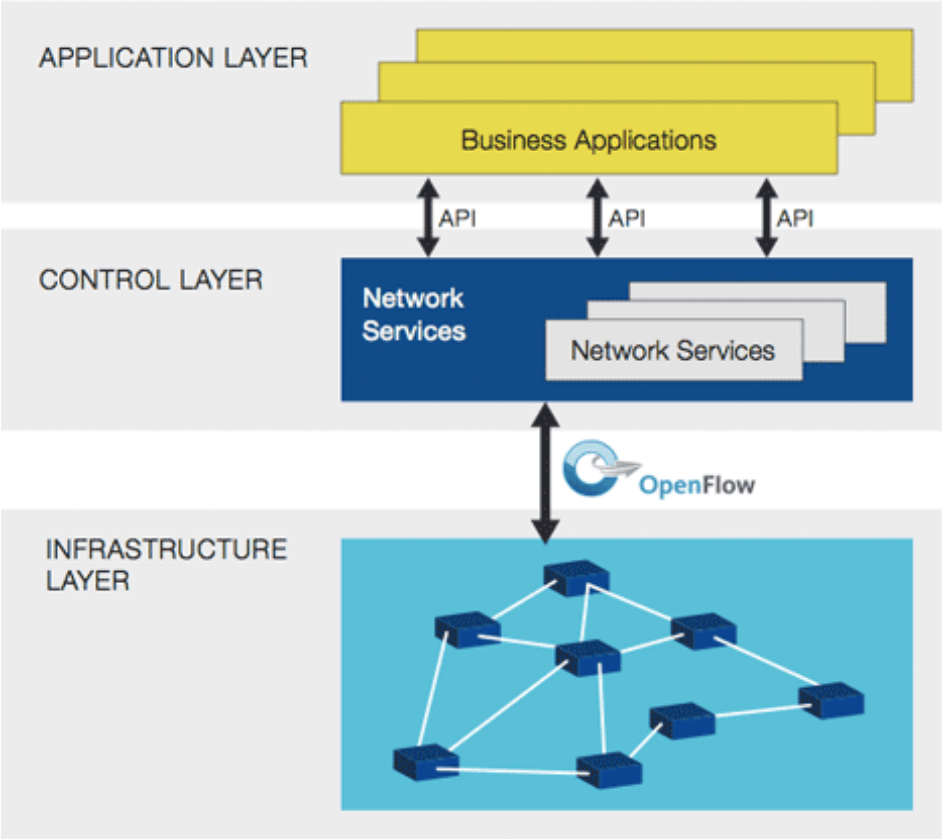


Figure 1: Software Defined Network Architecture [1]

Back to the architecture, at bottom we have the **Infrastructure Layer** (data plane) consisting of network elements (e.g. router, switches) where the SDN Datapaths are determined through the SDN Controller. The SDN Datapaths are logical network devices advertising forwarding and data processing capabilities. Each of these devices comprises a SDN Control to Data-Plane Interface (CDPI) agent and a set of one or more traffic forwarding engines and zero or more traffic processing functions including simple forwarding between the datapath's external interfaces, internal traffic processing or termination functions.

The **SDN CDPI** is defined between an SDN Controller and an SDN Datapath. It provides at least programmic control of all forwarding functions, capabilities advertisement, statistic reporting and event notification. The CDPI is usually implemented in an open, vendor-neutral and interoperable way. The first standard

communications interface defined between the control and forwarding layers of an SDN architecture is OpenFlow. OpenFlow will be described in the next section.

As mentioned before, the data forwarding is controlled by the **Control Layer** (SDN Controller) located in the middle of that architecture. It is a logically centralized entity. Basically, the controller is responsible for the low-level control translating the requirements from the SDN Application layer down to the SDN Datapaths. It also provides an interface (Northbound API) for the application layer with an abstract view of the network and important functions.

On top, applications in the **Application Layer** make use of that API to ease the usage and development of new applications by communicating their requirements to the controller via Northbound Interface (NBI) Drivers.

SDN NBIs are interfaces between SDN Applications and SDN Controller providing an abstract view on the network and enabling direct expressions of network behavior and requirements. It is also implemented in an open, vendor-neutral and interoperable way.

Furthermore, we have a **Management and Admin plane** that covers static tasks including physical equipment setup, coordinating reachability, credentials among logical and physical entities, configuring bootstrapping and assigning resources to clients. Usually it is better to handle those tasks outside the application, control and data planes.

All interfaces mentioned before (NBI, CDPI) are implemented by a driver-agent pair. The agent represents the bottom or infrastructure facing side and the driver represents the top or application facing side.

3.2.1 OpenFlow Overview

OpenFlow is an open standard to deploy innovative protocols in production networks. Furthermore, it is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. The key function is to enable direct access to and manipulation of the forwarding plane of network devices (e.g. router, switch) by moving the network control out of the networking switches to logically centralized control software.

The main concept is the use of flows. Flows identifies the network traffic based on pre-defined match rules that can be statically or dynamically programmed through the SDN control software. It gives the possibility to define how traffic should flow through network devices by using parameters such as usage patterns, applications and cloud resources.

In combination with SDN - OpenFlow-based SDN architecture - it provides an extremely granular control, for instance, responding to real-time change at the application, user and session levels.

OpenFlow consists of different components (shown in figure Figure 2). The important ones are the OpenFlow Switch, the OpenFlow Controller and the OpenFlow Protocol. The **OpenFlow Switch** consists of one or more flow tables and a group table. The

switch performs packet lookups and the forwarding. The **OpenFlow Controller** communicates with the OpenFlow Switch and is responsible to manage their tables by adding, updating or deleting flow entries in the tables. The communication between the OpenFlow Switch and OpenFlow Controller is done by the **OpenFlow Protocol (OFP)**.

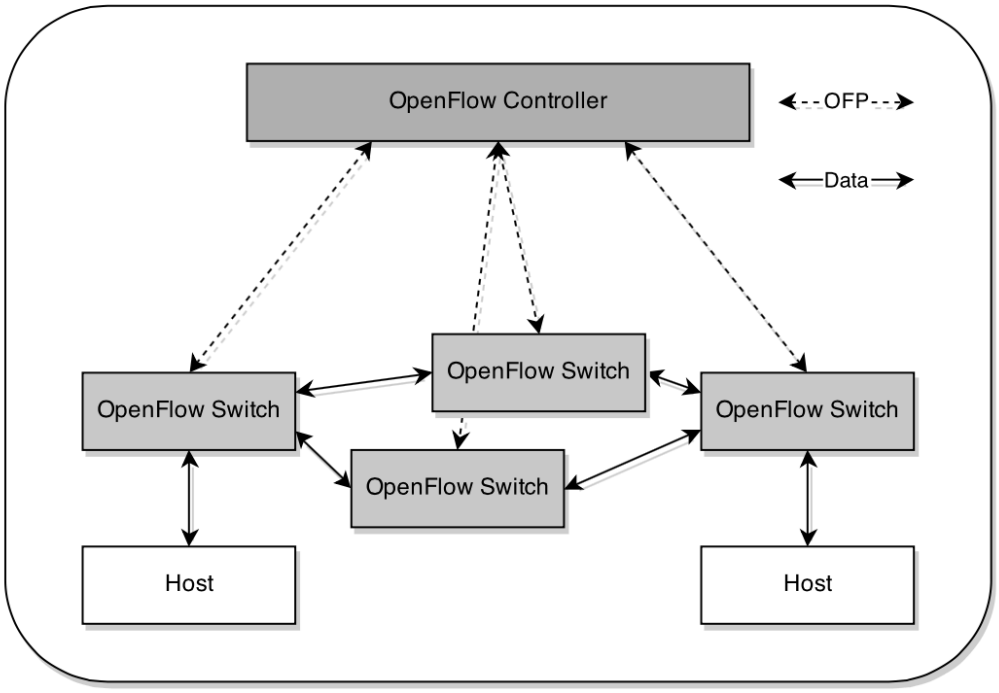


Figure 2: OpenFlow Network Architecture

3.2.2 OpenFlow Switch

OpenFlow separates control and switching functionality by extracting the routing decision point from the switch into the controller. An OpenFlow Switch needs to support a minimal subset of OpenFlow functionalities:

- Flow tables with flow entries.
- Control interface towards an OpenFlow controller allowing commands and packets to be sent between controller and switch.
- OpenFlow protocol support for remote flow table manipulation through the control interface.

For the communication with the OpenFlow Switch, the OpenFlow Controller provides an interface that offers a secure SSL/TCP connection between these two components. Depending on the flow description, it is possible to aggregate flows with wildcard flow entries for a large number of flows or to match every flow individually to achieve a fine grain control. A wildcarded flow entry is shown in Figure 3, a fine grained flow entry in Figure 4.

In Port	Ethernet			IP			TCP	
	SA	DA	Type	SA	DA	Prot	Src	Dst
1	*	*	*	*	*	*	*	*

Figure 3: Wildcard Flow Entry

In Port	Ethernet			IP			TCP	
	SA	DA	Type	SA	DA	Prot.	Src	Dst
1	34:4f:2f:d2:7c:8e	34:4f:2f:d2:7c:44	*	192.168.2.1	192.168.2.2	TCP	445	80

Figure 4: OpenFlow Matching Example

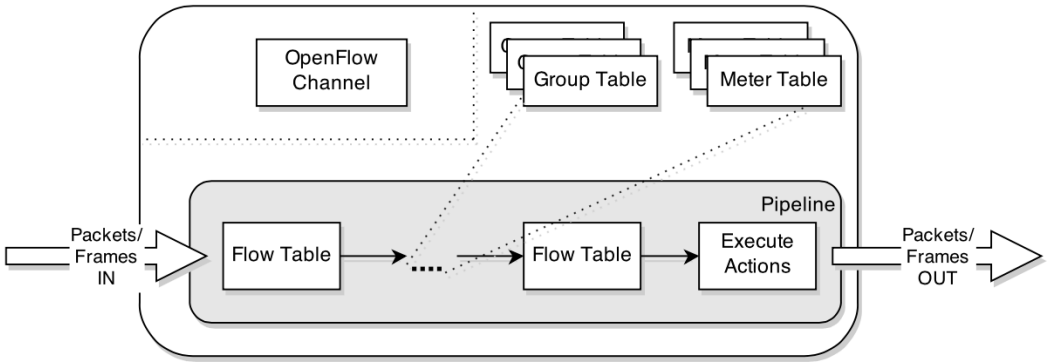


Figure 5: OpenFlow Switch Components

A specific flow can match a flow table on different layers. The matching properties can be Switch Port, MAC, IP, TCP Port or Protocol type, which are representative for different Layers (1-4).
Mainly there are two types of OpenFlow Switches:

- OF-only: Only OpenFlow pipeline forwards packets
- OF-hybrid: either OpenFlow or normal routing resumes packet forwarding

Shown in Figure 5, an OpenFlow Switch consists of a pipeline, that itself contains flow tables, group tables or meter tables. Inside the pipeline, incoming packets are looked up in the flow tables or group tables in order to decide, whether to forward, drop or manipulate the packet. The OpenFlow Channel represents the connection to the external controller. Packets that are not captured by the forwarding rules inside the pipeline are forwarded to the controller to allow a high- level decision. Each flow table in the switch contains a set of flow entries. Each flow entry consists of matching fields, counters, and a set of instructions for applying to matching packets.

3.2.3 OpenFlow Controller

This section is going to describe the fundamentals of the OpenFlow Controller.
In comparison to the former switch architecture, the controller is separated from the switch to become an independent function block, that can be deployed on a discrete machine. The controller itself contains two interfaces. The northbound interface offers a formatted API to the application layer, that can be used to implement any kind of application to manage or control the underlying network. Within the southbound interface the controller connects to an OpenFlow enabled set of switches. The interaction between the switch and the controller is bidirectional. Many different commands are available to control the switch e.g. adding, modifying or deleting of several switch components.

3.2.3.1 Centralized vs. Distributed controller.

There are several configurations possible in the architecture for OpenFlow. The first one is a centralized controller, that keeps up the communication to every of its switching elements. This turns the controller to a single point of failure. To obtain this situation

OpenFlow allows the switch to connect to multiple controllers. This scenario can be used to build a backup controller that takes over, if the default one has crashed. This approach is presented as logically centralized to the application but internally distributed. Due to [3], it has several drawbacks in look-up speed and potential trade-offs in consistency and staleness while distributing the state in the set of controllers.

To use controllers in a distributed way, a controller-to-controller protocol has to be designed. This task is not covered yet through the OpenFlow initiative. An interesting approach of deploying multiple controllers, that are using the same set of switches in an isolated manner, is Flowvisor [2]. With this approach it is possible to slice the OpenFlow network and let multiple controllers use overlapping parts of it. It is a way of virtualizing the network.

3.2.3.2 Reactive vs. Proactive Policies

There are two different strategies of pushing rules. Proactively pushing rules to the switch means that the controller has to know about the network configuration. This also means, that configuration is rarely static but no delay occurs like in reactive strategy. Pushing reactively means, a flow is established after consulting the controller. Based on a certain rule, the flow is pushed to the switch, the flow can continue in line rate. The request mechanism creates a delay based on the remote nature of the controller. It is always good to know what kind of flows are present to optimize the behavior of the network.

Table 6 shows a list of available controller for comparison in OpenFlow Protocol Version, Programming Language and activity

	OpenFlow Version	Programming Language	Active Branch
NOX	1.0/1.2/1.3	C++/Python	
POX	1.0	Python	Features for later OF versions
Trema	1.3.1	Ruby/C	
Beacon	1.0	Java	
Floodlight	1.0	Java	1.2/1.3 in progress
Ryu	1.0/1.2/1.3/1.4	Python	

Table 6: Available OpenFlow Controller

3.3 QoS

The term Quality of Service is widely used but different models of understanding are common. One model is defined as the “intrinsic” and its quality is determined by the transport network design and provisioning of network access, terminations and connections. The quality is achieved by selecting the specific protocol, the QoS assurance mechanism and other configurable parameters. The evaluation of the QoS values is measured and compared with the expected values. [14]

The most interesting parameters that are used in the context of the intrinsic qos are:

- Bitrate: throughput of packets of the service that can be achieved
- Jitter: is the variation in the time packets arrive due to network congestion and route changes
- Packet loss rate: is the proportion between packets that are lost and packets that are sent
- Delay: is the time between the action and the following up action. Either in relation to end-to-end or in regards to a particular network element

3.4 OpenStack

Openstack is a bunch of open source software tools to build and manage public and private cloud infrastructure. Rackspace Cloud and NASA founded the project and currently more than 200 companies are part of the open-source community. The OpenStack Foundation manages the development and the community.

Openstack consists of many different services that are taking over functions such as controlling of controlling resources, storage, networking, processes, etc. OpenStack defined nine core parts where each of them offering an API to the end user.

Core Modules:

- OpenStack Compute (Nova)
- OpenStack Object Storage (Swift)
- OpenStack Image Service (Glance)
- OpenStack Identity (Keystone)
- OpenStack Dashboard (Horizon)
- OpenStack Networking (Neutron)
- OpenStack Block Storage (Cinder)

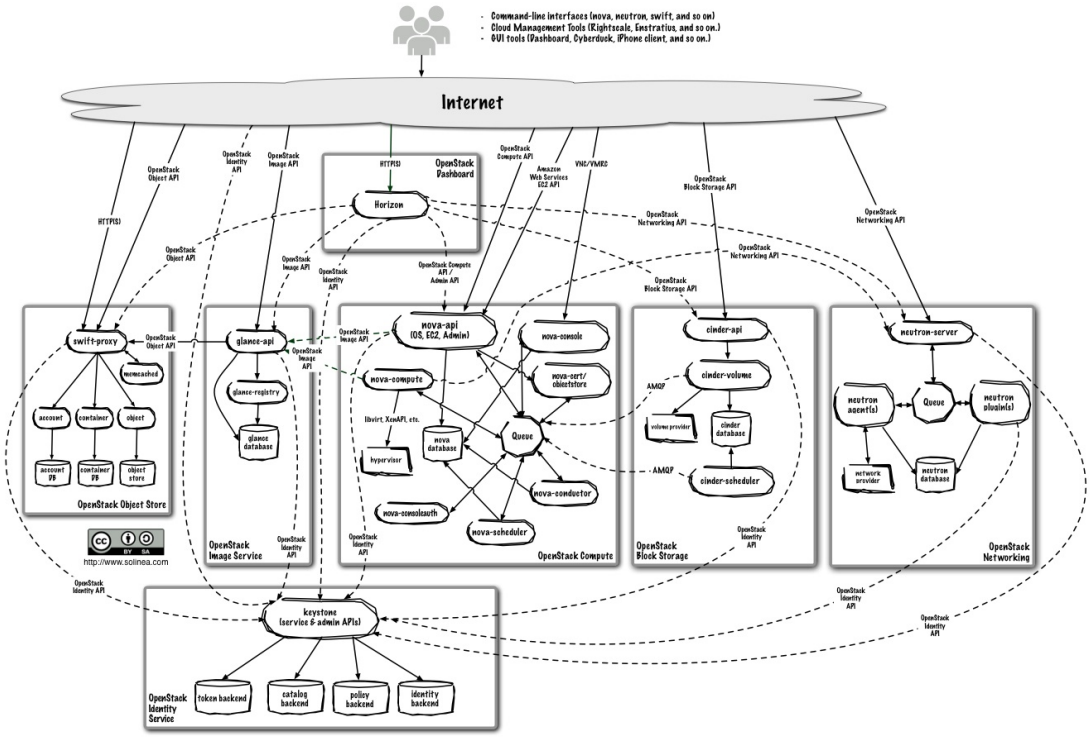


Figure 1. OpenStack Logical Architecture [9]

For this project the most interesting part of OpenStack that needs to be observed are the services Nova and Neutron. Nova manages the lifecycle of virtual machines interoperating via a driver mechanism with an extensible number of hypervisors. All components are implemented in Python, like the rest of OpenStack services and it scales horizontally without requiring particular hardware resources.

3.4.1 Neutron

Neutron provides an API for managing network connectivity and addressing in the cloud. The Neutron API allows users to define [11]:

- Network: An isolated L2 segment, analogous to VLAN in the physical networking world.
- Subnet: A block of v4 or v6 IP addresses and associated configuration state.
- Port: A connection point for attaching a single device, such as the NIC of a virtual server, to a virtual network. Also describes the associated network configuration, such as the MAC and IP addresses to be used on that port.

Additional network services are ranging from L3 forwarding and NAT to load balancing, edge firewalls, and IPSEC VPN. Figure 2 shows the Neutron architecture and its main functional elements.

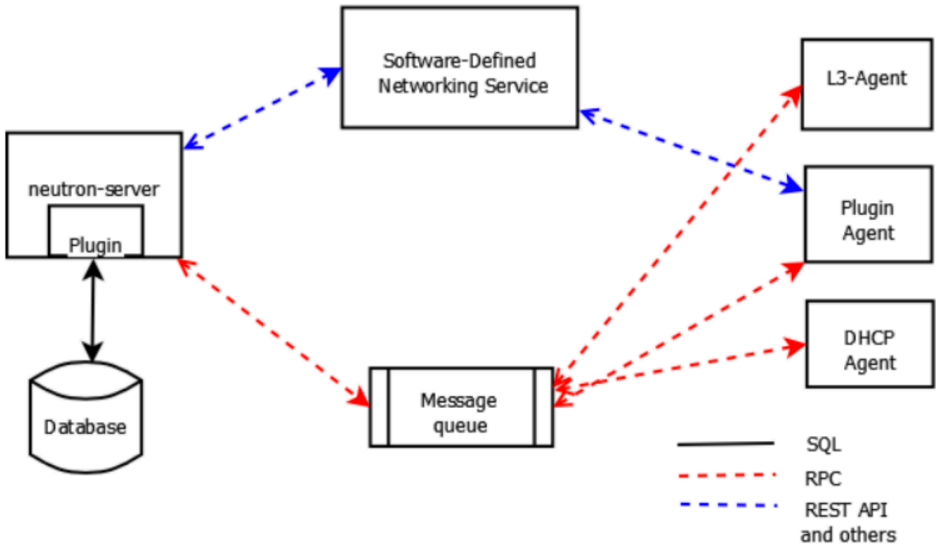


Figure 2. Neutron Architecture [10]

Those components are [10]:

- neutron server (neutron-server and neutron-*-plugin): this service runs on the network node to service the Networking API and its extensions. It also enforces the network model and IP addressing of each port. The neutron-server and plugin agents require access to a database for persistent storage and access to a message queue for inter-communication.
- plugin agent (neutron-*-agent) Runs on each compute node to manage local virtual switch (vswitch) configuration. The plug-in that you use determines which agents runs. This service requires message queue access. Optional depending on plugin.
- DHCP agent (neutron-dhcp-agent). Provides DHCP services to tenant networks. This agent is the same across all plug-ins and is responsible for maintaining DHCP configuration. The neutron-dhcp-agent requires message queue access.
- L3 agent (neutron-l3-agent). Provides L3/NAT forwarding for external network access of VMs on tenant networks. Requires message queue access. Optional depending on plug-in.
- network provider services (SDN server/services). Provide additional networking services to tenant networks. These SDN services might interact with the neutronserver, neutron-plugin, and/or plugin-agents through REST APIs or other communication channels

3.4.2 ML2 Plugin

Neutron plugins represent the interface between neutron and backend technologies such as SDN, Cisco, VMware NSX and so on. The currently most interesting plugin that is used within the neutron-server is the ML2 Plugin [12]. It provides a framework to utilize a variety of L2 networking technologies simultaneously. The ML2 plugin is organized in two major parts: types and mechanism shown in Figure 7.

The type defines the type of network that is being implemented and mechanism is the technology used to implement that type. The implemented mechanisms are modular drivers such as OpenvSwitch, linuxbridge or vendor specific implementations.

The ML2 type driver maintains the type-specific state, provides tenant network allocation, validates provider network attributes and models network segments by using provider attributes. Some ML2-supported network types include GRE, VLAN, VxLAN, Local and Flat.[13] The ML2 mechanism driver calls resources on the vendor product such as Cisco Nexus, Arista, LinuxBridge or Open vSwitch. Multiple mechanism drivers can access the same network simultaneously with three types of models : agent-based, controller-based and ToR switch-based. Examples of agent-based models include Linuxbridge and OVS. Controller-based models might include OpenDaylight and ToR (top-of-rack) switch-based models include (but not limited to) Cisco Nexus, Arista and Mellanox. [13]

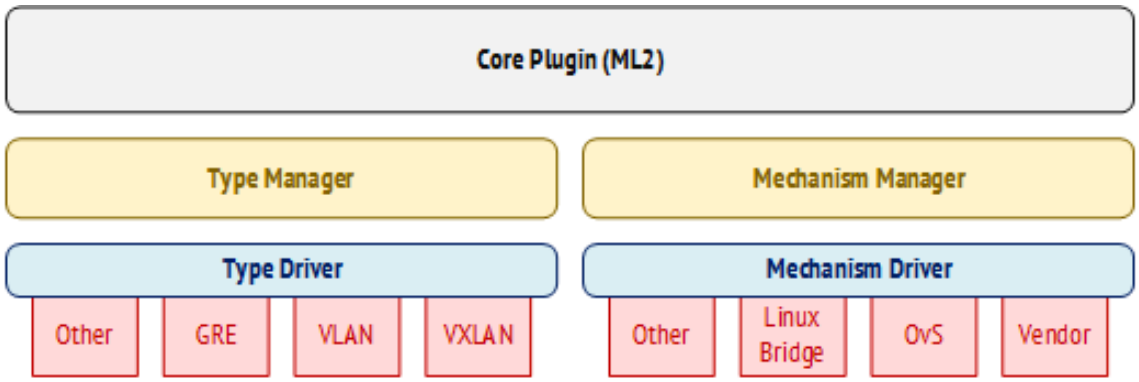


Figure 7: ML2 Plugin structure [13]

Neutron provides a driver mechanisms through which different type of data plane entities are supported. By default those entities are running instances of OpenVSwitch which are executed in each of the compute node.

4 Investigations on QoS using OpenStack

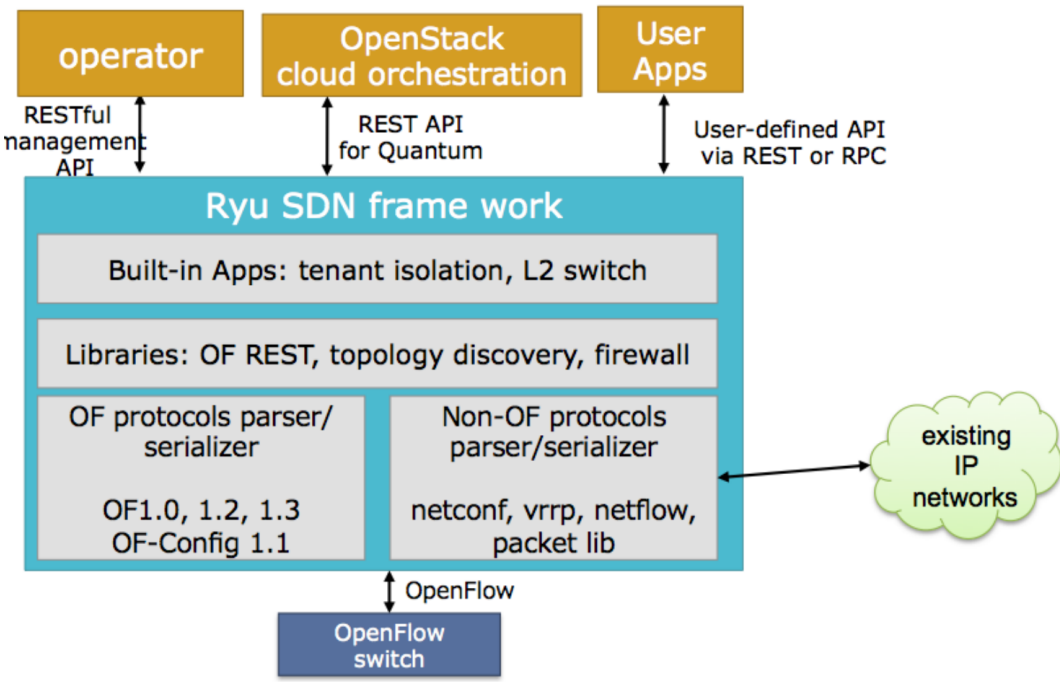
As mentioned in the previous section the interface of OpenStack to configure the network is provided by Neutron. Apart from that, there are different ways to provide an SDN infrastructure inside the cloud. To determine an appropriate solution that supports our requirements or to reach our objectives, three solutions were tested and investigated.

All of the following solutions are implemented and make use of OpenStack Neutron. For testing the components an all-in-one node of OpenStack was deployed, which was then configured to use the corresponding ML2 mechanism drivers that build the counterpart within Neutron to the SDN controllers.

4.1 Ryu

Ryu is a component-based software defined networking framework which fully supports OpenFlow 1.0, 1.2, 1.3 and 1.4 switches and is fully written in Python. Ryu is a full featured OpenFlow controller that supports GRE and VLAN tunneling. The OpenFlow controller that is embedded in the agent sets Flows on the switch by sending OpenFlow messages to the switch. It includes a set of apps which build the base of the SDN controller like L2 switch, ReST interface, topology viewer and tunnel modules . Ryu also includes an app that allows to set QoS rules through a ReST interface which

uses a OVSDB interaction library to apply those. The QoS rules can be either applied to a specific Queue within a VLAN or a Switch port. It supports DSCP tagging and setting the min-rate and max-rate of an interface.



4.1.1 Architecture of Ryu

As of OpenStack IceHouse Ryu has been renamed to OFagent and is included in the Neutron repository. In order to use it as the SDN framework for OpenStack Neutron, OFagent has to be set as both the ML2 mechanism driver (running on the Control / Network node) and the Neutron agent (running on the Compute node).

Figure 8:Architecture of RYU [4]

The test of Ryu was unsuccessful due to a number of errors while stacking the test environment using Devstack. It was not possible to launch instances and test the QoS features. The lack of proper documentation for the interaction with OpenStack Neutron led us to look more into other SDN controllers for our particular use case. Currently Ryu doesn't support the Distributed Virtual Routing feature that has been introduced with OpenStack Juno.

4.2 OpenDaylight

The Virtualization edition of OpenDaylight which offers integration with Neutron consists of components presented in Figure 8.

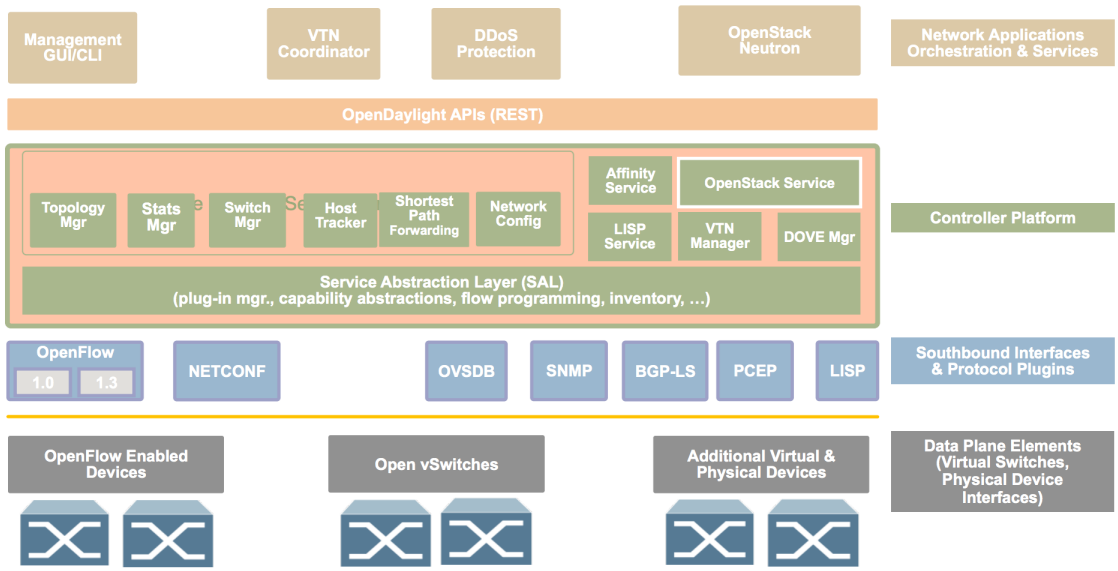


Figure 9: OpenDayLight Architecture

OpenDaylight is fully implemented in Java. The Controller platform has multiple Northbound & Southbound interfaces. OpenDaylight exposes a single common OpenStack Service Northbound API which exactly matches the Neutron API. The OpenDaylight OpenStack Neutron Plugin simply passes through and therefore pushes complexity to OpenDaylight and simplifies the OpenStack plugin. The ML2 mechanism driver in Neutron has to be set to the OpenDaylight ML2 plugin with the ODL agent running on the Compute Nodes. The OpenDaylight controller can be run on the Control Node or on a separate VM. The Open vSwitch database (OVSDB) Plugin component for OpenDaylight implements the OVSDB management protocol that allows the southbound configuration of vSwitches. The OpenDaylight controller uses the native OVSDB implementation to manipulate the Open vSwitch database. The component comprises a library and various plugin usages. The OVSDB protocol uses JSON/RPC calls to manipulate a physical or virtual switch that has OVSDB attached to it.

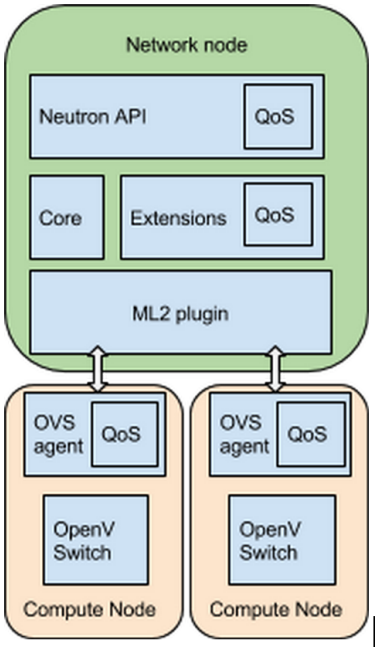
The OVSDB component is accessible through a Northbound ReST API, which enables the operator to connect to the OpenFlow controller and modify various OVSDB tables. Through this API QoS rules can be deployed. Because it connects directly to the OpenVSwitch tables, all the QoS types that come with OpenVSwitch can be deployed (DSCP marking, setting priority, min-/max-rate for switch ports & OpenFlow Queues). In the local testbed we were able to successfully deploy QoS rules on the ports of Virtual Machines.

The local testbed used for the integration of OpenStack Juno and OpenDaylight Helium consisted of 2 hosts, one running the OpenStack control node and OpenDaylight Controller and a OpenStack Compute Node on the second host. During the tests it was not possible to get the public network access for the Virtual Machines working. This and the fact that it ODL is very complex to debug and understand all underlying processes led us to the decision not to use OpenDaylight.

4.3 Neutron QoS implementation

A Neutron extension has been implemented for OpenStack IceHouse based on the following API blueprint [8]

The proposal includes applying QoS rules to Neutron networks and specific ports. The implementation has not been finished and merged into Neutron, however the basic deployment of QoS seem to have been tested successfully.



At the moment it is not clear if the OpenStack team will keep working on this, according to the whiteboard it was deferred to Juno, but it's not included in the current release and no active development is stated in the code / documentation platforms of the OpenStack community.

The patch consists of an extension to the Neutron API which allows setting QoS rules through the Neutron Python client, the actual Neutron extension with the QoS, QoS Driver in the OpenVSwitch agent and an addition to the Neutron Database that includes QoS.

4.4 Conclusion

The two outlined SDN controllers and the Neutron QoS extension in their present form don't offer the features and implementations that are needed for fulfill the requirements of the CM in the NUBOMEDIA project. The Table 1 shows an overview of all solutions tested.

Requirements/Tools	Provides QoS	Provides OpenFlow Controller	Works appropriate with OpenStack
Ryu	yes	yes	no (no stable version)
OpenDaylight	yes	yes	no (no stable version)
Neutron	not fully implemented	no	not tested

Table 1: Comparison between OpenStack related QoS programs

The result of this comparison is that it is much more valuable to implement the CM without any support of this tools.

5 Objectives

The Connectivity Manager is the part in the NUBOMEDIA Architecture that realizes the control and management of the SDN topology provided by the OpenStack virtual network infrastructure. This requires an integration of the implementation into the virtual technology chosen in Deliverable D3.1.1 (OpenStack). One of the key objectives of the CM is to grant different SLAs on the links between compute nodes. Where in our solution the SLA is specified as a set of Quality of Service parameters such as minimal rate and maximum rate. This objective is also related to the placement problem. The placement problem arises from compute and network conditions that are present in a running cloud infrastructure. To ensure the best connectivity and best resource utilization it is necessary to determine the optimal positioning of a virtual instance.

In addition to that the CM should also handle appropriate routing priorities for the cloud SDN controller. This requirement as well as additional monitoring capabilities are postponed to the next release in the NUBOMEDIA project. The placement algorithm has not yet proper logic to provide a significant value. That's why it also needs to be advanced in the upcoming release.

6 Functional Architecture

The CM of the NUBOMEDIA Infrastructure provides the following functionalities:

- Optimal Instance Placement: during the deployment of an NUBOMEDIA Infrastructure an algorithm decides where to place each NUBOMEDIA instance inside the cloud infrastructure
- SLA enforcement: due to quality requirements arising by the media type or an explicit configuration the cloud infrastructure will be modified to fulfill this requirements

In order to realize such functionalities, a functional architecture of the CM is proposed in Figure 10.

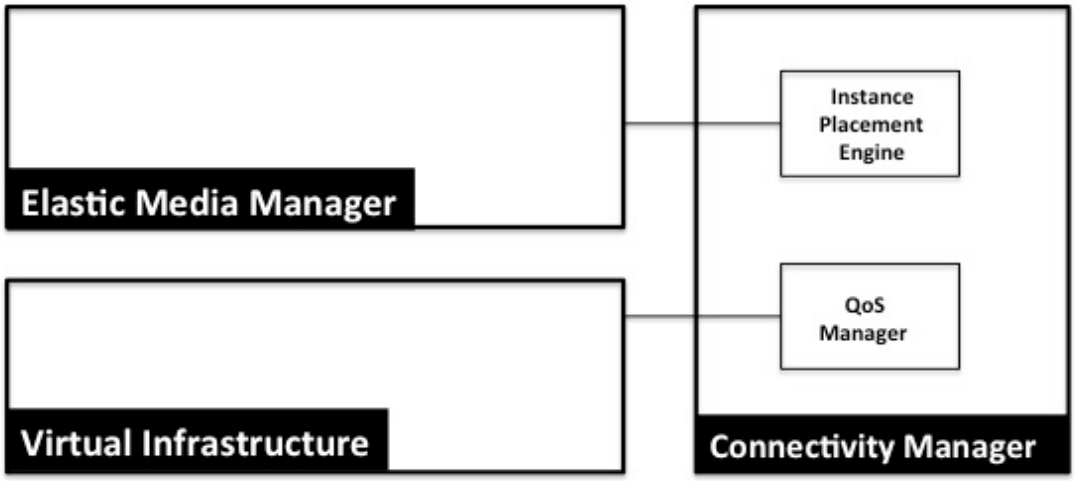


Figure 10: Functional Architecture

6.1 Instance Placement Engine

The Instance Placement Engine calculates the best location for a virtual instance inside the Virtual Infrastructure. This function decides where to place the specific element based on different indicators such as resource utilization of the compute nodes and available network capacity.

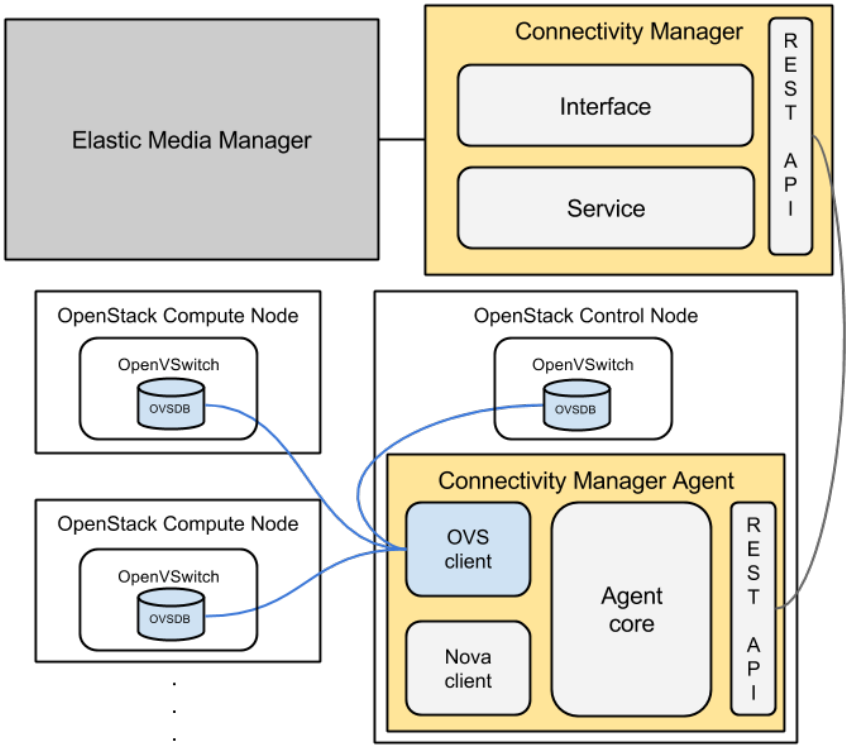
6.2 QoS Manager

The QoS Manager provides functions to enforce e specific QoS class for a specific host. According to previous calculations, hosts can be equipped with guarantied and maximum bitrate.

7 Software Architecture

The Connectivity Manager implementation is based on the framework of the Elastic Media Manager (EMM) D3.3.1. Therefore it is also implemented in Python and consists of two components:

- Connectivity Manager, which is incorporated into the EMM framework as set of modules extending the basic functionalities of the EMM
- Connectivity Manager Agent, which is running on the OpenStack Control Node in the datacenter



7.1 Modular Architecture

In the following section the modular architecture of the two components will be described. The modules are split into multiple Python packages.

7.1.1 Connectivity Manager

For the implementation of the Connectivity Manager it uses parts of the EMM, as described in Deliverable D3.4.1.

- Interfaces / Connectivity Manager: This package contains a generic interface to the EMM core services.
- Services / Connectivity Manager: This package contains the implementation of the CM interface and builds the core of the Connectivity Manager.
- Test: This package contains a set of module tests.
- WSGI: This package contains the REST API that is exposed to the Connectivity Manager Agent. The API documentation is documented in a separate section in this document.

7.1.2 Connectivity Manager Agent

The Agent consists of the following packages:

- Clients: contains the modules that instantiate the OpenStack Clients and the implementation of the OpenVSwitch Client
- Core: contains the core implementation of the Connectivity Manager Agent
- Test: contains a set of module tests
- WSGI: contains the REST API that is exposed to the Connectivity Manager

7.2 Connectivity Manager - Services implementation

The implementation of the Connectivity Manager Service is shown in Figure 11. The Connectivity Manager uses the implementation framework

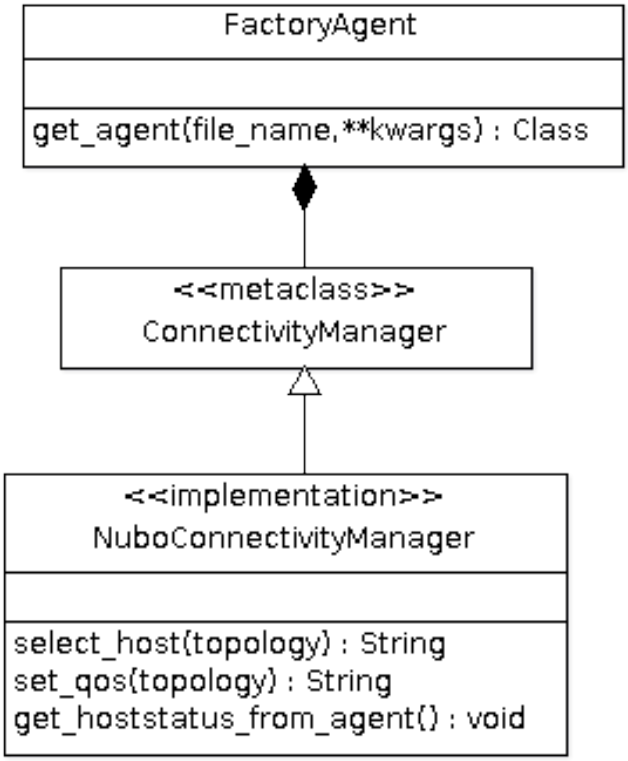


Figure 11: Class diagram of Connectivity Manager Service

7.2.1 NuboConnectivityManager

The NuboConnectivityManager implements the function of the optimal application placement. This means that information such as compute capacity and deployed QoS levels received from the Connectivity Manager Agent are evaluated. Based on this information the decision where to place the Virtual Machine is taken and additional QoS rules are set according to predefined QoS classes.

The first version of the algorithm for the placement of the compute instance is very simple. One random compute node will be chosen and its CPU values will be compared.

This means if the limit of CPUs is reached or will be exceeded by the placement of the compute instance another random compute node will be chosen.

7.3 Connectivity Manager Agent - Module implementation

Figure 12 shows the class diagram of the Connectivity Manager Agent.

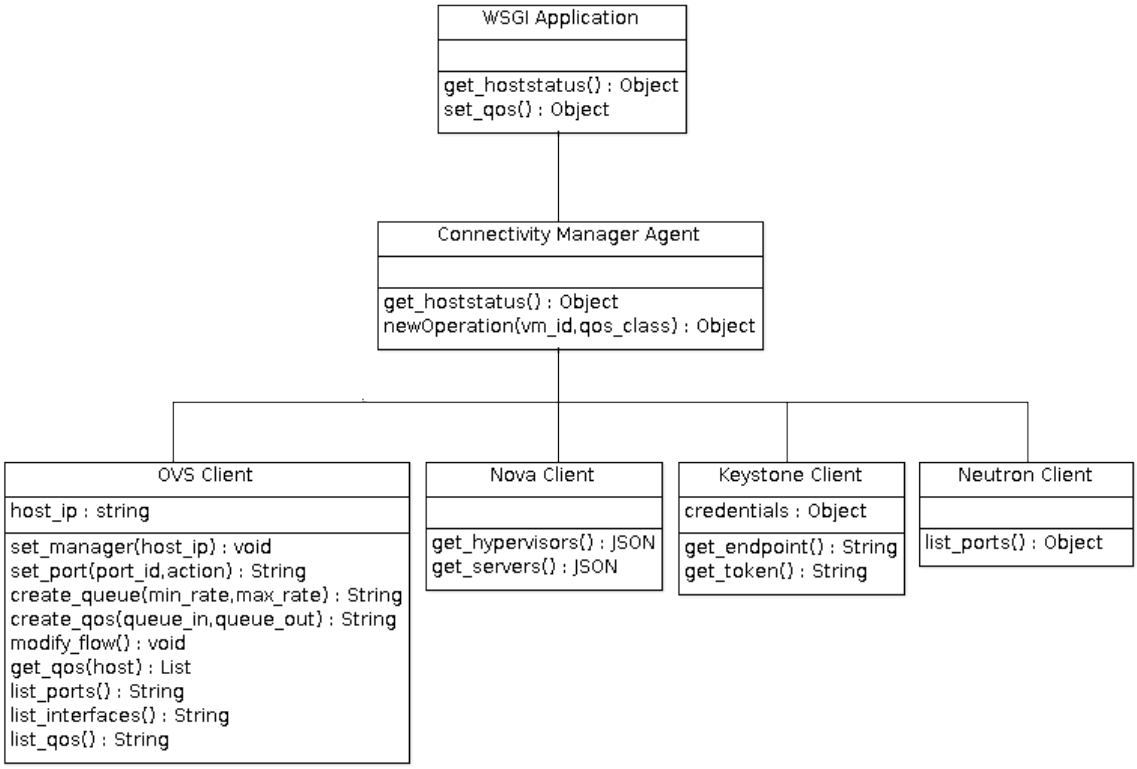


Figure 12: Class diagram of Connectivity Manager Agent

7.3.1 Core / Agent

The main component of the Connectivity Manager Agent is the agent module within the core package. This class handles all the methods for retrieving and parsing the information about all running hosts and setting QoS rules on them.

7.3.2 WSGI / Application

The application module within the WSGI package provides the ReST interface for the RPC to the Connectivity Manager and forwards the requests to the core.

7.3.3 Clients

The Connectivity Manager Agent requires two Clients in order to perform its operations.

The NovaClient uses the python-novaclient package to receive information about the Hypervisors that run Nova (so called Compute Nodes) and their current usage within the OpenStack tenant. The OVSClient is an implementation of calls using the ovs-vsctl command, which is a high-level interface to the configuration database of OpenVSwitch. The KeystoneClient is used to authenticate the CM against OpenStack. This is necessary access the neutron API. The NeutronClient provides information about provisioned ports available in the virtual network infrastructure of OpenStack. This is necessary to identify the ports related to the OpenVSwitch.

7.4 Sequence diagram

Figure 13 shows the sequence diagram of the interaction between the Elastic Media Manager, Connectivity Manager and Connectivity Manager Agent. This Sequence diagram captures all current implemented actions between the components listed in the figure.

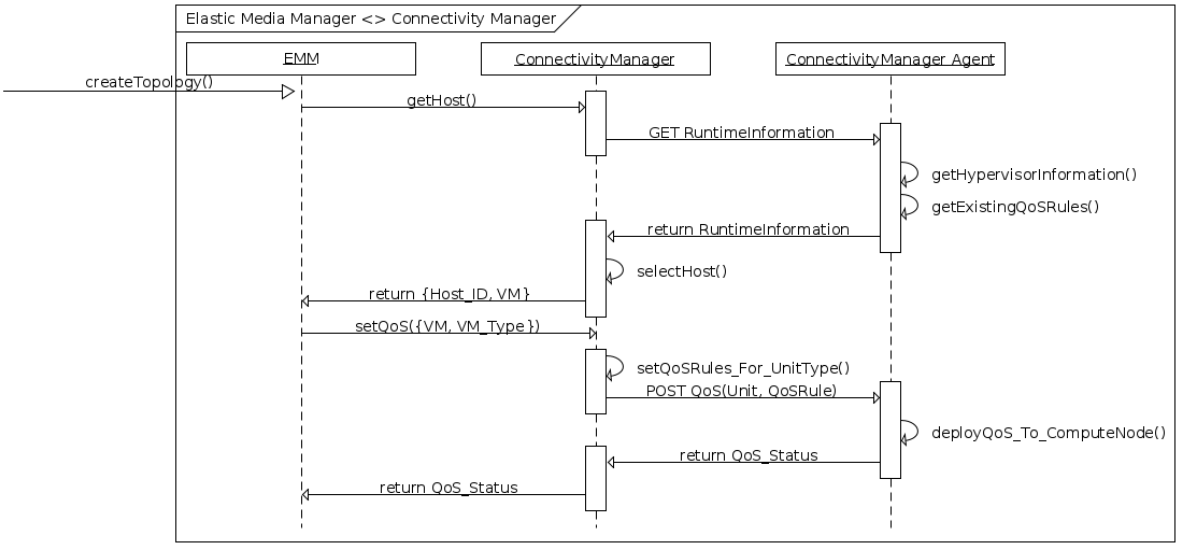


Figure 13: Sequence Diagram of EMM – CM interconnection

8 API

In the following table the REST API that the Connectivity Manager exposes to the Connectivity Manager Agent is described.

Method	Path	Method Body	Description
GET	/hosts	Empty	Lists all available hosts (OpenStack Compute Nodes)
POST	/qoses	{Topology:{VM:{QoSclass}}}	Sets specific QoS Rule for VMs according to predefined QoS class

8.1 List all available hosts

This calls requests information about the current state of the available hosts within the OpenStack tenant. It includes details about the RAM usage, used CPUs, amount of deployed Virtual Machines, deployed QoS rules and information about the OpenVSwitch status of each host with Compute Node capabilities.

8.2 Set QoS rule

This calls to the Connectivity Manager Agent triggers the creation of new QoS rules in the OpenVSwitch configuration of the host that the Virtual Machines are running on. The body of this request contains information that identify the virtual network port of

the VM and the QoS class that is required for the service that this machine is offering to the overall NUBOMEDIA stack.

Annex

Installation Guide

The installation of the Connectivity Manager is separated in two parts. The first part is the Connectivity Manager. Its source code is a part of the Elastic Media Manager and therefor will be installed during the installation of the Elastic Media Manager. The second part is the Connectivity Manager Agent, which is located on the Cloud Infrastructure. In the current case implementation we stick tightly to OpenStack.

Installation of the Connectivity Manager:
The installation guide is part of the Elastic Media Manager installation guide located in D3.4.1 Section "Installation"

Installation of the Connectivity Manager Agent:
Requirements:

- sudo rights inside the one of the OpenStack physical instance for the ovs-vsctl tool
- Credentials for authenticate to the OpenStack API
- Connection to all compute nodes that are part of OpenStack

Installation:
1. Check out the git repository

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

2. move to the cm-agent folder

```
cd /opt/nubomedia/cm-agent
```

3. build and install the cm-agent

```
sudo python setup.py build && sudo python setup.py install
```

4. adjust the credentials in the keystone file

```
cm-agent/clients/keystone.py
```

5. start the agent and listen on port 8091

```
sudo python wsig/application.py
```

Throughput Evaluation of the OpenStack testbed

This evaluation is created to determine the necessity of a placement algorithm for the compute instances. The measurements reflect the difference between two compute instances on the same compute node of OpenStack and two compute instances on different compute nodes. The result of this evaluation is based on the former release of OpenStack named "IceHouse". It demonstrates a bandwith limitation of around 8 Gbit/s

for the inter compute connection and 1 Gbit/s for intra compute connections. The latest release called "Juno" should provide a technology called Distributed Virtual Routing which will lead in better performance due to a redundant deployment of network functions. Due to stability leaks this version will not be used in the current revision but is foreseen for the next release of the CM.

OpenStack version	IceHouse	
Measurement	2 VMs on 1 host	2 VMs on 2 hosts
Average flow setup latency (measured in millisecs)	0.4066	0.7832
Average steady-state latencies (millisecs)	0.093	0.119
Max TCP unicast throughput under varying MSS (viz., 90 bytes, 1490 bytes, 9000 bytes) --> Bandwidth	7.67 Gbits/sec 7.91 Gbits/sec 7.85 Gbits/sec	75.6 Mbits/sec 936 Mbits/sec 939 Mbits/sec
UDP under varying number of parallel sessions (viz., 1 and 5)	1 parallel session: received / sent bandwidth ratio: 97.28% 5 parallel sessions: received / sent bandwidth ratio: 49% 1 Parallel session: Sent bandwidth: 810 Mbits/sec Received bandwidth: 788 Mbits/sec 2 datagrams received out-of-order 5 Parallel sessions: Sent bandwidth: 189 Mbits/sec 188 Mbits/sec 189 Mbits/sec 190 Mbits/sec 189 Mbits/sec [SUM] 946 Mbits/sec Received bandwidth: 94.2 Mbits/sec 2 datagrams received out-of-order	1 parallel session: received / sent bandwidth ratio: 99.88% 5 parallel sessions: received / sent bandwidth ratio: 25.58% 1 Parallel session: Sent bandwidth: 810 Mbits/sec Received bandwidth: 809 Mbits/sec 109 datagrams received out-of-order 5 Parallel sessions: Sent bandwidth: 375 Mbits/sec 372 Mbits/sec 375 Mbits/sec 368 Mbits/sec 373 Mbits/sec [SUM] 1.86 Gbits/sec Received bandwidth: 95.8 Mbits/sec 38 datagrams received out-of-order 97.0 Mbits/sec 42 datagrams

	93.9 Mbits/sec 1 datagrams received out-of-order 94.5 Mbits/sec 6 datagrams received out-of-order 94.3 Mbits/sec 14 datagrams received out-of-order 94.3 Mbits/sec 10 datagrams received out-of-order	received out-of-order 95.9 Mbits/sec 21 datagrams received out-of-order 93.4 Mbits/sec 64 datagrams received out-of-order 93.7 Mbits/sec 52 datagrams received out-of-order
Max Multicast/broadcast throughput (in packets/sec)	0:24.62elapsed 28%CPU 95% unanswered	0:17.51elapsed 39%CPU 93% unanswered
Max allowed TCP flows between known hosts, without zero drops (measured in flows/sec)	300 packets: 6.0 3000 packets: 5.0 30000 packets: 4.9 300000 packets: 1.5	300 packets: 6.1 3000 packets: 6.7 30000 packets: 7.6 300000 packets: 2.9

References

[1] Software-Defined Networking: The New Norm for Networks, https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] FlowVisor: https://github.com/OPENNETWORKINGLAB/flowvisor/wiki

[3] Logically Centralized? State Distribution Trade-offs in Software Defined Networks, http://www.net.t-labs.tu-berlin.de/papers/LWHHF-LCSDTSDN-12.pdf

[4] RYU OpenFlow Controller, https://nsrc.org/workshops/2014/nznog-sdn/raw-attachment/wiki/WikiStart/Ryu.pdf

[5] Neutron/OFAgent/ComparisonWithOVS – OpenStack https://wiki.openstack.org/wiki/Neutron/OFAgent/ComparisonWithOVS

[6] ODL OpenStack integration, https://github.com/opendaylight/docs/blob/master/manuals/developers-guide/src/main/asciidoc/ovsdb.adoc

[7] Download OpenStack, https://www.openstack.org/assets/presentation-media/osodlatl.pdf

[8] Neutron QoS Blueprint, https://docs.google.com/document/d/1nGUqEb4CEdabbTDyeL2ECVicnBRNrK3amJcNi-D4Ffo/edit

[9] OpenStack logical Architecture, http://docs.openstack.org/admin-guide-cloud/content/logical-architecture.html

[10] Neutron Architecture, http://openstack.booktype.pro/security-guide/neutron-architecture/

[11] Neutron OpenStack Network Service, http://cloudn1n3.blogspot.de/2014/11/openstack-series-part-8-neutron.html

[12] ML2 Plugin, https://wiki.openstack.org/wiki/Neutron/ML2

[13] ML2 Plugin Structure, http://www.aqorn.com/understanding-openstack-neutron-ml2-plugin/

[14] Gozdecki, J. ; Jajszczyk, A. ; Stankiewicz, R., “Quality of service terminology in IP networks“