Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Studiengang Computer Engineering
Fachbereich 1

Design and implementation of a connectivity manager for virtual scalable network environments

Bachelorarbeit
von

Manuel Bergler

01. Dezember 2014 – 08. Februar 2015

Referent:	Herr Prof. Dr. Thomas Baar
Betreuer:	Herr Benjamin Reichel M. Sc.

Manuel Bergler
Urbanstr. 26
10967 Berlin

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Internet-Quellen vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Berlin, den 06. Februar 2015

Manuel Bergler

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The demands on networks have changed dramatically in the past two decades, with an ever-growing number of people and devices relying on interconnected applications and services. The underlying infrastructure has been left mostly unchanged and is reaching its limits. In order to resolve this, Software Defined Networking (SDN) will extend and replace parts of traditional networking infrastructures. SDN separates the network into control and forwarding planes and therefore allows a more efficient orchestration and automation of network services.

The use of virtualized cloud-based services, with not only competitive pricing but also high-availability and fast network access, is taking over traditional purely hardware-based data centers. The ease of administration and deployment of new servers as Virtual Machines (VMs) on the fly make it possible to effortlessly create a topology of multiple servers.

Network services have different requirements, depending on the type of data packets and their characteristics. A classification and prioritization of network traffic can be achieved through Quality of Service (QoS). A new approach has to be made to enable and configure the use of QoS in virtualized cloud infrastructures like OpenStack through an API. The traffic flow should be controlled right from the deployment of Virtual Machines onwards.

1.2 Network Architecture

Today's traffic patterns, the rise of cloud computing and "big data" to only name a few examples, are exceeding the capacity of classical network architectures. With scalable computing and storage, the common-place tree-structured networking infrastructure are not efficient and manageable enough.

The increasing complexity of problems that have to be faced in networks and the need to control network traffic through software, are only a selection of the reasons why the Open Networking Foundation (ONF) developed an approach called Software-Defined Networking (SDN).

SDN is a leading-edge approach where the network control is separated from the forwarding functions. The centralized network intelligence allows programming the network, without a need to access its underlying infrastructure. With the help of this architecture a shift of today's networks leading to more flexibility, programmability and scalability is taking place.

1.3 Objective

The primary objective of this work is the development of a Connectivity Manager which acts as a network orchestrator and is able to apply Quality of Service to the network interfaces of Virtual Machines. These VMs are deployed on compute nodes that run the OpenStack Nova-service and are interconnected with Open vSwitches. Another task of the CM is to select which OpenStack compute node new servers should run on. All functions of this software should be applicable in environments at scale.

1.4 Scope

The scope of this work includes a Connectivity Manager which is interfaced with the already existing implementation of the Elastic Media Manager. The second application is a Connectivity Manager Agent that runs on the cloud controller within the OpenStack infrastructure in order to provide access to the hosts of OpenStack Nova. These two components have to be implemented and integrated with the existing SDN software that OpenStack makes use of. The entire virtual network switching will utilize Open vSwitch, which is a fully-compliant OpenFlow switch. The deployment of a cloud topology is tested on different performance characteristics such as the maximal network bandwidth, jitter, CPU utilization and memory usage.

In virtualized cloud infrastructure like OpenStack, the placement of Virtual Machines (VMs) on a particular compute node can be decided on by comparing different run-time parameters. The network connectivity between those VMs has to be prioritized and classified depending on the type of service that the server provides.

Currently there are a number of solutions for managing network connectivity between virtual servers that partially fit the requirements of this work. A comparison and their current limitations follows in the next section. The selected approach is the extension of existing network control and management services with Quality of Service (QoS) capabilities and the ability to choose a host for the deployment of the topology. In support of the thesis the Connectivity Manager will be implemented and the evaluation of the network performance will be performed on basis of the services of the NUBOMEDIA project.

1.5 Overview

Chapter 1 begins with the motivation for this thesis and gives a brief introduction into the objectives and the scope.

Chapter 2 gives an overview of traditional network concepts and a introduction to SDN and its components. Furthermore the different services that OpenStack consists of will be described.

Chapter 3 conceptualizes the state-of-the-art solutions that are currently available and evaluates their implementation and limitations.

Chapter 4 contains an analysis of requirements.

Chapter 5 gives an architectural overview of the Connectivity Manager. Moreover design aspects are introduced and illustrated in relation to their requirements.

Chapter 5 examines the implementation of the Connectivity Manager and Agent.

Chapter 6 evaluates the network performance tests on the basis of one use-case.

Chapter 7 summarizes the results of this work and gives an outlines possible future work.

Chapter 2

Fundamentals and Related Work

2.1 Software-Defined Networking

The origins of Software-Defined Networking (SDN) go back to as early as 1995, however the first implementations were only developed in 2001 and the promotion of SDN began with the foundation of the non-profit industry consortium Open Networking Foundation (ONF) in 2011. [1] The ONF is dedicated to push open standards like OpenFlow into the industry and adapt them. In this following section a brief overview of the SDN architecture and concepts, including the OpenFlow standard and Open vSwitch, is given.

2.1.1 Motivation

Today's internet is part of the modern society for private users, enterprises and vital infrastructure services. Networks are required to evolve in order to address the challenges that are entailed with new applications, services and a growing number of end-users.

With a more detailed view on the challenges of current network infrastructures one comes to see the following limitations: [2]

- **Inability to scale**: With the expansion of data centers, networks grow too. Configuring and managing these additional network devices comes at a high administrative effort. With the virtualization of data-centers network traffic patterns become more and more dynamic and unpredictable. With multi-tenancy an additional complication is introduced, because different end-users and services need require distinct network characteristics. Network management and scaling of the architecture cannot be achieved with a manual configuration of the underlying infrastructure.
- **Complexity**: In the past decades various new networking protocols have been adapted by the industry. To add or move a device, multiple of the already existing switches, routes and firewalls must be touched in order to manage and introduce new protocol-based mechanisms. With the virtualization of servers the amount of network interfaces that need connectivity is on the rise. The distribution of applications across a number of virtual machines are another demand that the current fairly static connections cannot adapt to.

- **Inconsistent policies**: For IT departments to apply a network- or data center-wide policy, a lot of devices and mechanisms may need to be reconfigured. Virtual Machines can be created and rebuilt within no time, but if access or security rules need to be updated then the benefits of this dynamic can't be utilized to its full extent.
- **Vendor dependence**: Standards are needed to match the requirements of the markets with the capabilities of networks and enable data center operators to customize the network to specific environments.

Traditionally, decisions about traffic-flows through the network are made directly by each network device, because the control logic and forwarding hardware are tightly coupled.

2.1.1.1 Classical Switches & Routers

Packet forwarding (data plane) and routing decisions (control plane) in classical switching and routing are both within one device. The main components depicted in Figure 2.1 have the following functions:

1. The **forwarding path** typically handles data path operations for each packet. It generally consists of Application-Specific Integrated Circuits (ASIC), network-processors or general-purpose processors that forwards frames and packets at wire speed (line-rate). Their lookup functions can be further increased with memory resources like Content Addressable Memory (CAM) or Ternary Content Addressable Memory (TCAM) to contain the forwarding information.
2. The elements in the **control plane** are based on general-purpose processors that provide services like routing and signaling protocols, including ARP, MAC Learning and forwarding tables.

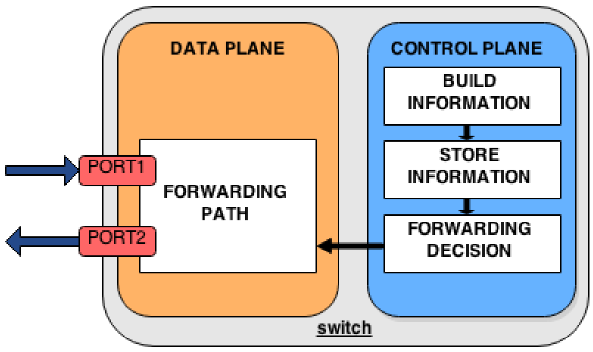


Figure 2.1: "Classical" switch components

A switch consists of multiple ports for incoming and outgoing data. Internal forwarding tables classify the packets and forward them to one specific or multiple ports. It does so by collecting MAC addresses and storing their corresponding port in tables. Layer 2 switches also support the segregation into virtual LANs (VLAN), which enables the network operator to logically isolate networks that share a single switch.

Routers forward packets on the Network layer (Layer 3) and routing-decisions are made based on IP addresses. They contain a routing table wherein paths to neighbouring networks are stored, so that packets can be forwarded to their destination IP address. Other features that can be configured with routers are Quality of Service (QoS), Network Address Translation (NAT) and packet filtering.

The main differences between the classical architecture and SDN will be further described in the next section.

2.1.2 Software-Defined Networking Concept

SDN represents a novel dynamic, manageable, cost-effective and adaptable architecture [3] that is built to serve the dynamic infrastructures that are needed as a backbone for modern data centers. Opposed to the traditional approach, network control and forwarding functions are decoupled and thus can be programmed and divided into different applications and network services. The work of the Open Networking Foundation laid out the OpenFlow standard as the base for modern SDN solutions.

2.1.3 SDN Architecture

SDN separates the architecture into three distinct layers that communicate with each other through different APIs. This separation is shown in Figure 2.2.

- **Infrastructure Layer:** here all physical and virtual devices (e.g. switches and routers) that are capable of the OpenFlow Protocol provide forwarding mechanisms on different Network Layers.
- **Control Layer:** represents the 'network intelligence' and collects a global view of the network, by communicating with the switching elements through the so-called Southbound API.
- **Application Layer:** consists of business applications that allow the network operator to extend the SDN controller on an abstracted level, without being tied to the actual details of the implementation of the infrastructure. This communication with the Control Layer is also called Northbound API.

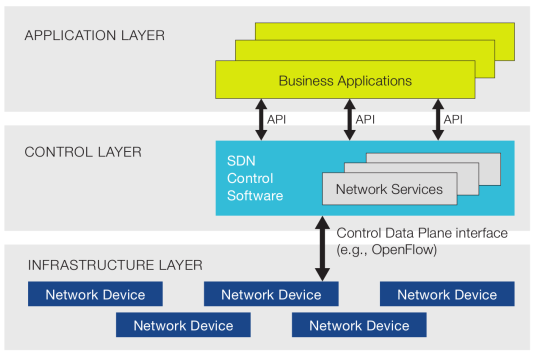


Figure 2.2: SDN architecture
[3]

2.1.4 OpenFlow

With OpenFlow the Open Networking Foundation defined the first standard communications interface between the SDN architecture's control and forwarding layers. It enables manipulation and direct access to the forwarding plane of physical as well as virtual (hypervisor-based) network devices such as switches and routers. [2]

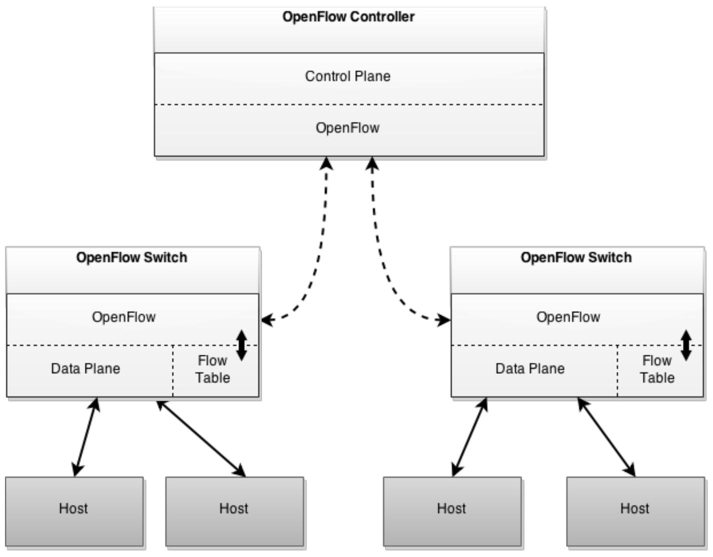


Figure 2.3: OpenFlow network architecture

OpenFlow first of all stands for the communications protocol that is used by SDN controllers to fetch information and configure switches. Additionally it is a switch specification with full support of OpenFlow.

Many of the OpenFlow-enabled switches and controllers up to today still solely support the OpenFlow version 1.0 (released in December 2009). The newest version at this date is 1.4, however the characteristics of OpenFlow in this work will focus on version 1.3, since it is the most recent specification which is supported by Open vSwitch.

The main features added since version 1.0 are, among others, support for VLANs, IPv6, tunnelling and per-flow traffic meters. [4]

Generally the switches are backwards-compatible to version 1.0. In the following description the focus lies on the required features of all OpenFlow capable devices, however it has to be mentioned that some of them are optional features.

2.1.4.1 OpenFlow Controller

The OpenFlow controller is segregated from the switch and has two interfaces. The northbound interface is an API to the application layer for developing applications that control the network. The southbound interface connects with the underlying switches using the OpenFlow protocol.

2.1.4.2 OpenFlow Switch

There are two varieties [5] of OpenFlow-compliant switches:

- **OpenFlow-only:** in these switches all packets are processed by the OpenFlow pipeline and have no legacy features.
- **OpenFloy-hybrid:** support OpenFlow and normal Ethernet switching (including traditional L2 Ethernet switching, VLAN isolation, L3 routing, ACL and QoS). Most of the commercial switches that are available on the market today are of this type.

An OpenFlow switch includes one ore more flow tables and a group table, which have the function of carrying out packet lookups and forwarding. Another component is the OpenFlow channel that is connected to the external controller.

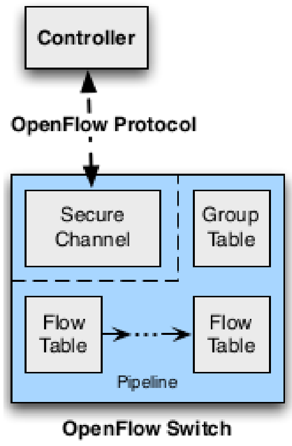


Figure 2.4: OpenFlow switch components
[5]

Through the connection using the OpenFlow protocol, the controller is able to add, update and delete flow entries in Flow tables. This action can be performed either reactively or proactively. Sets of flow entries are stored in each flow table and each flow entry consists of *match fields*, *counters*, and a set of *instructions* used for matching packets.

Matching flow entries starts in the first flow table, however it may continue to additional flow tables. The first matching entry from each table is used and the instruction that is linked with that specific entry is performed. For packets without any matches a table-miss flow entry can be configured. Flow entries are usually forwarded to a physical port.

The instructions can either include actions or else modify the pipeline processing. Packet forwarding, packet modification and group table processing are available actions. Packets can be permitted to be sent to other tables with pipeline processing and additionally metadata can be exchanged between tables.

Packets can also be directed to a group, which contains a set of actions for flooding and more complex forwarding semantics (e.g. multipath, fast reroute and link aggregation).

2.1.4.3 OpenFlow Ports

OpenFlow ports are the network interfaces used for passing packets between OpenFlow processing and the rest of the network [5]. There are various types of ports that are supported by OpenFlow. This section will give a short overview about this port abstraction. Incoming OpenFlow packets enter the switch on an ingress port, are then processed by the OpenFlow pipeline and forwarded to an output port. (See OF Tables figure for processing).

There are three types of OpenFlow ports that must be supported by an OpenFlow switch:

- **Physical ports:** are hardware interfaces on a switch.
- **Logical ports:** don't directly interact with a hardware interface.
- **Reserved ports:** contain generic forwarding actions (e.g. sending to the controller, flooding or forwarding using traditional switch processing)

2.1.4.4 OpenFlow Tables

Pipeline Processing

The OpenFlow pipeline specifies how packets correspond with each of the flow tables [5].

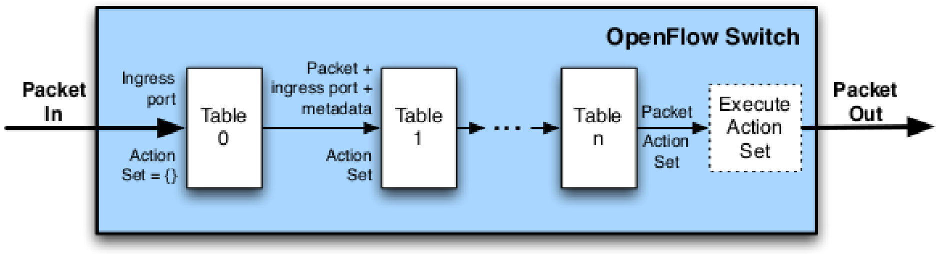


Figure 2.5: OpenFlow pipeline processing
[5]

As illustrated in the figure, each packet is matched against the flow entries starting at the first flow table, called flow table 0. The outcome of the match then decides if other of the sequentially numbered tables may be used. In the following sections the components of the Flow table, the matching procedures and different instructions will be described.

Flow Table

A flow table contains flow entries, each of which consist of the following fields [5]:

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Table 2.1: Fields within a Flow table

- **match fields:** ingress port, packet headers and optionally metadata
- **priority:** sets the priority of the flow entry
- **counters:** is updated for matching packets

- **instructions:** to alter the action set or pipeline processing
- **timeouts:** set maximum amount of time or idle time before expiration of the flow
- **cookie:** is an opaque data value chosen by the controller

Each flow table entry is uniquely identifiable by its match fields and priority.

Packet Matching

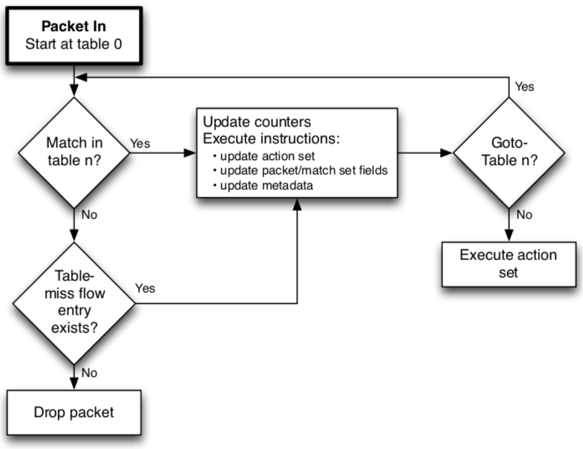


Figure 2.6: Packet flow through an OpenFlow switch
[5]

Upon a packet's arrival at the Flow Table, the packet match fields are extracted and used for the table lookup [5]. They include different packet header fields. Additionally the ingress port can be matched with metadata fields. If the values in the packet match fields are equal, then only the flow entry with the highest priority is selected. Furthermore the associated counters are updated and the instruction set is applied.

When the instruction set associated with a matching flow entry does not specify a next table, the pipeline processing stops. Only then the packet is processed with it's action set and in most cases forwarded as shown on the right-hand side of Figure 2.6. However, if the lookup phase does not match any of the entries, a table-miss event occurs.

Table-miss

Each flow table must support a table-miss flow entry which specifies how to process packets that are unmatched by other flow entries. The instructions associated with this entry are very alike to any other flow entries. Packets are either forwarded to other controllers, dropped, or it is continued with the next flow table. In case the table-miss flow entry is non-existent, unmatched packets are dropped by default.

Group Tables

A group table consists of group entries and it provides a way to direct the same set of actions as part of action buckets to multiple flows. A flow entry is pointed to a group and enables additional methods of forwarding (e.g. broadcast or multicast).

Meter Tables

Meters are on a per-flow level and allow OpenFlow to implement various QoS operations, such as rate-limiting, but it can also be combined with per-port queues to implement more complex QoS like DiffServ.

The main components of a meter entry in the meter table are:

Meter Identifier	Meter Bands	Counters

Table 2.2: Fields within a meter entry in the meter table

- **meter identifier:** a 32 bit unsigned integer uniquely identifying the meter
- **meter bands:** each meter band specifies the rate of the band and the action that is triggered by exceeding the limit
- **counters:** is updated when packets are processed by the meter

The rate of packets assigned to a meter are measured and controlled. Meters are directly attached to flow entries, as opposed to queues that are attached to ports. A meter is able to have one or more meter bands, each of which specifies the rate and the way packets should be handled. If the current measured meter rate reached the rate-limit, the band applies an action.

A meter band is identified by its rate and consists of the following fields:

Band Type	Rate	Counters	Type specific arguments

Table 2.3: Fields within a meter band

- **band type:** defines how the packets are processed
- **rate:** selects the meter band for the meter and defines the lowest rate at which the band can apply
- **counters:** is updated when packets are processed by the meter
- **type specific arguments:** some bands have optional arguments

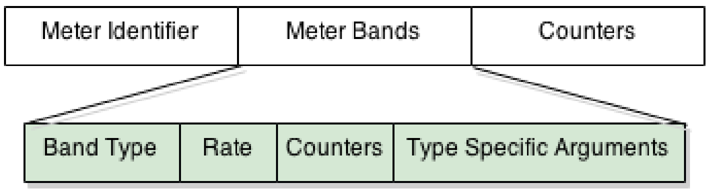


Figure 2.7: OpenFlow QoS as a meter

Instructions

Instructions are executed when a packet matches the flow entry [5]. Their result is either a change to the packet, action set and/or pipeline processing. There are different instruction types and some of them are required for an OpenFlow-enabled switch whereas others are optional:

- **Meter *meter_id*:** Directs packet to the specified meter. The packet may be discarded as the result of the metering.
- **Apply-Actions *action(s)*:** Applies the specific action(s) instantly, without any change to the Action Set.
- **Clear-Actions:** Immediately clears all the actions in the action set.
- **Write-Actions *action(s)*:** Merges the specified action(s) into the current action set.
- **Write-Metadata *metadata / mask*:** Writes the masked metadata value into the metadata field.
- **Goto-Table *next-table-id*:** Indicates the next table in the processing pipeline.

A maximum of one instruction of each type is associated with a flow entry and they are executed in the order as specified by the given list. Flow entries can also be rejected if the switch is not able to execute its instruction.

Action Set

An action set is associated with each packet, which is empty by default [5]. The action set can be modified using a *Write-Action* or a *Clear-Action* instruction. If there is no *Goto-Table* instruction within the instruction set of a flow entry the pipeline processing is halted and the actions in the action set of the packet are executed.

Actions

The following action types are available on OpenFlow-enabled switches [5]:

- **Output:** A packet is forwarded to a specified OpenFlow port.
- **Set-Queue:** Sets the Queue ID for a packet. This ID helps determining which Queue is used for scheduling and forwarding the packet. This forwarding behavior allows to enable basic QoS support on a port.
- **Group:** Process the packet through the specified group.
- **Push-Tag/Pop-Tag:** The ability to push/pop tags such as VLAN.
- **Set-Field:** Modifies the values of header fields in a packet.
- **Change-TTL:** Set the values of IPv4 TTL, IPv6 Hop Limit or MPLS TTL in a packet.

2.1.5 Open vSwitch

2.1.5.1 Concept & Functionality

Open vSwitch (OVS) is an open source software switch that is used in virtualized server environments. It is able to forward traffic between virtual network interfaces (e.g. connected to Virtual Machines) and the physical network, as well as between different VMs on the same physical host. It can be controlled using OpenFlow and the OVSDB management protocol. It can run on any Linux-based virtualization platform i.e. KVM, VirtualBox, XEN, ESXi and is part of the mainline kernel as of Linux 3.3 but can be run on kernel 2.6.32 and newer [6].

OVS supports the following features [7]:

- 802.1Q VLAN model
- Link Aggregation Control Protocol (LACP)
- GRE and VXLAN tunneling
- fine-grained QoS control
- OpenFlow
- per VM interface traffic policing
- High-performance forwarding using a Linux kernel module

The internals of OVS are as follows:

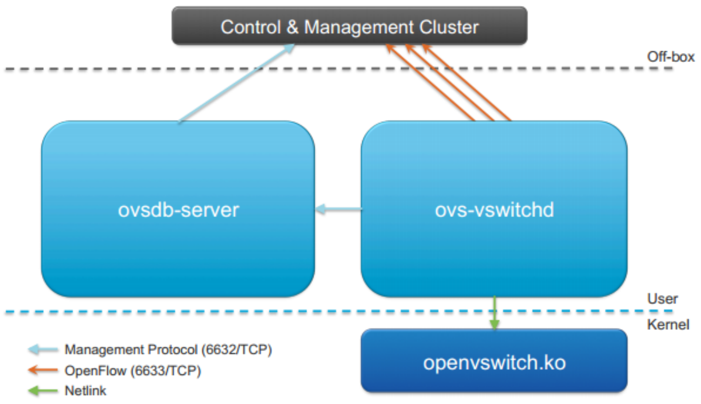


Figure 2.8: Architecture of Open vSwitch: kernelspace / userspace separation [8]

The daemon which implements the switch is *ovs-vswitchd* and is shipped with an additional Linux kernel module for flow-based switching that it communicates with using the netlink protocol [9]. The configuration for the switch is queried from a lightweight database server named *ovsdb-server*. Generally the decision about how a packet is processed is made in userspace, yet all following packets hit the cached entry in the kernel.

It is also possible to run the switch entirely in userspace, but it decreases the performance drastically.

2.1.5.2 OVSDB

Each Open vSwitch daemon has a database (OVSDB) that holds it's configuration [10]. The database is divided into multiple different tables with different purposes, with the ones related to this work outlined below:

- **Open_vSwitch**: Open vSwitch configuration
- **Port**: Port configuration
- **Interface**: A physical network device within a Port

- **QoS**: Quality of Service configuration
- **Queue**: Output queue for QoS
- **Controller**: OpenFlow controller configuration
- **Manager**: OVSDB management connection

2.1.5.3 Open vSwitch Management

Multiple different configuration utilities exist for OVS, however only ovs-vsctl is explained in this section. It is used for querying and updating the configuration of the switch through interaction with the ovsdb-server.

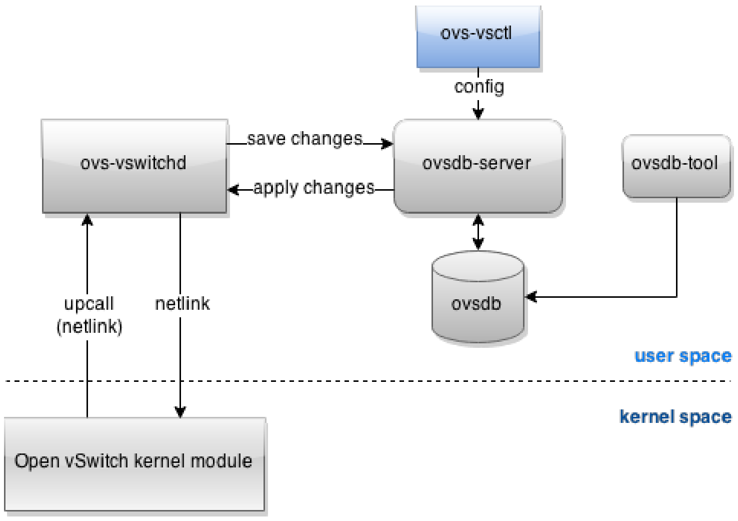


Figure 2.9: Visualization of ovs-vsctl within a OVS

The tool also configures ovs-vswitchd and can be seen as a high-level interface for the database. The commands below are available for the OVS configuration that is needed to get a basic network setup running:

```
$ ovs-vsctl add-br %bridge%
$ ovs-vsctl add-port %bridge% %port%
$ ovs-vsctl list-ports %bridge%
$ ovs-vsctl set Interface %port%
```

In order to add Flow entries to a bridge the *ovs-ofctl* is needed. The default Flow entry contains the following parameters:

```
$ ovs-ofctl add-flow br-int hard_timeout=0,idle_timeout=0,
   priority=1,actions=normal
```

2.1.5.4 QoS

With Open vSwitch Quality of Service can be configured for ports that virtual machines are connected to. The minimal and maximal rate-limits can be defined in bit / s and applied to a queue. With the use of this feature the network traffic from the switches internal bridge to a virtual network interface can be shaped (egress).

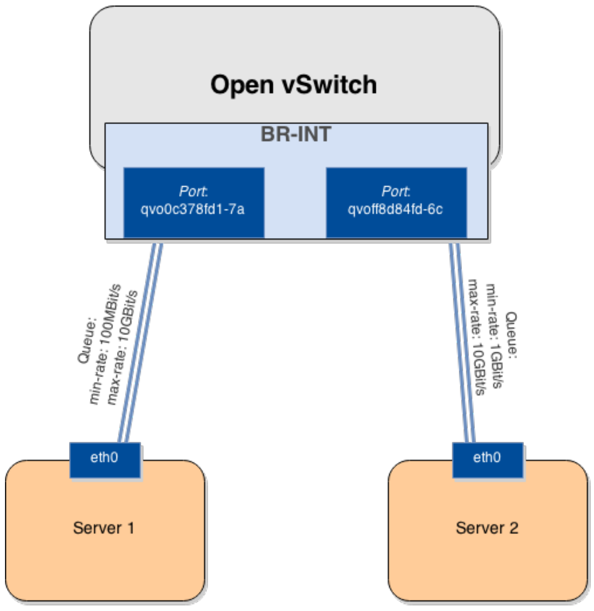


Figure 2.10: QoS Queues attached to a Port in OVS

The approach used in this work does not modify the OF Flow and their actions, which means that all Queues on a OVS are within the same Flow entry. The QoS port assurances in OVS make use of the 'tc' implementation which is included in the Linux kernel.

Traffic control (tc) uses 'queuing discipline' (qdisc) for configuring the network interface [11]. They are the fundamental schedulers used under Linux. When a packet is sent, it is enqueued to the qdisc for the interface and shortly after the kernel is trying to get as many packets as it can from the qdisc, so they can be forwarded to the network adaptor driver. By default the 'pfifo_fast' qdisc is set in the kernel, which is a pure 'First In, First out' queue.

When setting QoS in OVS a classful qdisc named Hierarchy Token Bucket (HTB) is used. HTB is meant as a more understandable, intuitive and faster replacement for the Class Based Queuing (CBQ) qdisc in Linux [12]. It helps to control the use of the outbound bandwidth on a given link.

For further understanding the following graph shows a HTB tree that contains on qdisc root and two classes (leaves) with different configurations:

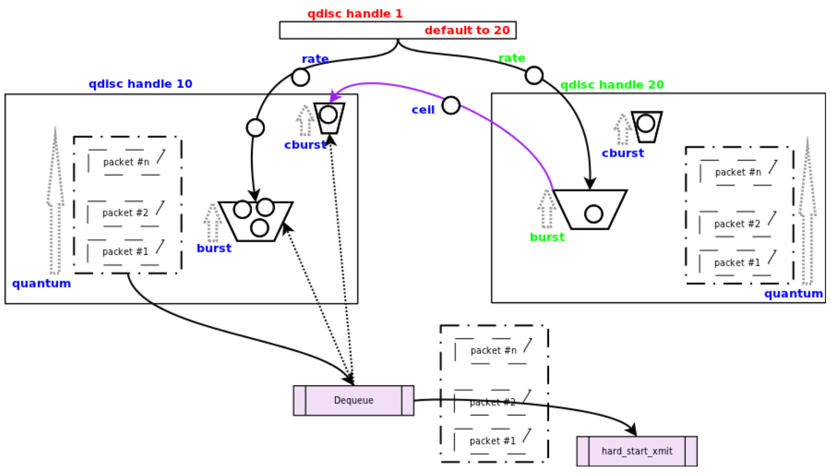


Figure 2.11: HTB: qdisc classes linked together as a tree [13]

The terminology that is used for defining HTB classes is as follows:

- **burst:** the size of the token bucket
- **rate:** the speed at which tokens are generated and put in the bucket (in the leaf)
- **quantum:** the amount of bytes to serve from a single leaf at once
- **ceil:** the definite upper class rate (this is the value that is set by the maximal rate defined in OVS QoS)
- **cburst:** the burst for ceil (gets computed)

This tree was created using the following commands:

```
$ tc qdisc add dev eth0 root handle 1: htb default 20
$ tc class add dev eth0 parent 1:0 classid 1:10 htb rate 200
   kbit ceil 400kbit prio 1 mtu 1500
$ tc class add dev eth0 parent 1:0 classid 1:20 htb rate 200
   kbit ceil 200kbit prio 2 mtu 1500
```

In figure 2.11 a root named 'qdisc handle 1' which applies filters to decide the direction of packets is depicted. It includes two leaves: 'qdisc handle 10' and set as default another leaf named 'qdisc handle 20'.

With the above listed commands the leaves were configured with the following parameters:

Leaf parameter	#10	#20
Rate value	200 KBit/s	200 KBit/s
Ceil value	400 KBit/s	200 KBit/s
Priority	1	2

Table 2.4: Qdisc leaf configuration

The difference between the rate and ceil value in leaf #10 means it can borrow 200 KBit/s more than it's actual rate, while leaf #20 cannot because its rate value equals the ceil value.

The HTB leaves always match one of the following three statuses [13]:

- **HTB_CAN_SEND:** the class can send using its own tokens
- **HTB_CANT_SEND:** the class can't send nor borrow and no packets are permitted to leave the class
- **HTB_MAY_BORROW:** the class can't send using its own tokens, but can try to borrow from another class

When a group of packets enters *tc* that are marked with the flag #10 and are directed to it, however the bucket does not contain enough tokens to let the first packets through then it will try to borrow tokens from leaf #20. The quantum value defines the maximal amount that leaf #10 will try to send at once. It is 1500 bytes, as that is the default MTU. If the first packet has a size of 1400 bytes and the bucket in leaf #10 has enough tokens for 1000 bytes it will try to borrow the remaining 400 bytes from the neighbor leaf. The quantum value can be used to privilege a leaf compared to others, because it specifies the tokens that it can borrow from other leaves in the tree.

By configuring *tc* the quantum value is not directly managed. It is automatically calculated based on an intermediary parameter named *r2q* with the following formula: $quantum = \frac{rate}{r2q}$ This value is set on the root qdisc and is used across all of its leaves. Its default value is set to 10.

2.1.5.5 VXLAN

Virtual Extensible LAN tunnels are used with Open vSwitch in this work to encapsulate isolated layer 2 network traffic between compute nodes. The benefits of VXLAN are that it can adress up to 16M tenant networks as opposed to the 4K in VLAN, it allows for better scaling and because the VXLAN support is only required at the endpoints of the tunnels it permits a vendor flexibility [14]. The tunnels can be used to perform a large scale isolation within a network.

2.2 Cloud Computing Infrastructures

2.2.1 OpenStack

The OpenStack project was founded by Rackpace Cloud and NASA, however up to this day more than 200 companies are contributing to it. With OpenStack it is possible to design, deploy and maintain a private or public cloud installation. It is a flexible, scalable and open-source approach that combines multiple technologies into one Infrastructure-as-a-Service (IaaS) [16]. All of the interrelated services include an API that offers administrators different ways of controlling the cloud, be it through a web-interface, a command-line client or a Software Development Kit. All of the core components that are part of OpenStack are implemented in Python.

Conceptual architecture

The following graph shows the interaction between different OpenStack services that are involved in launching a virtual machine.

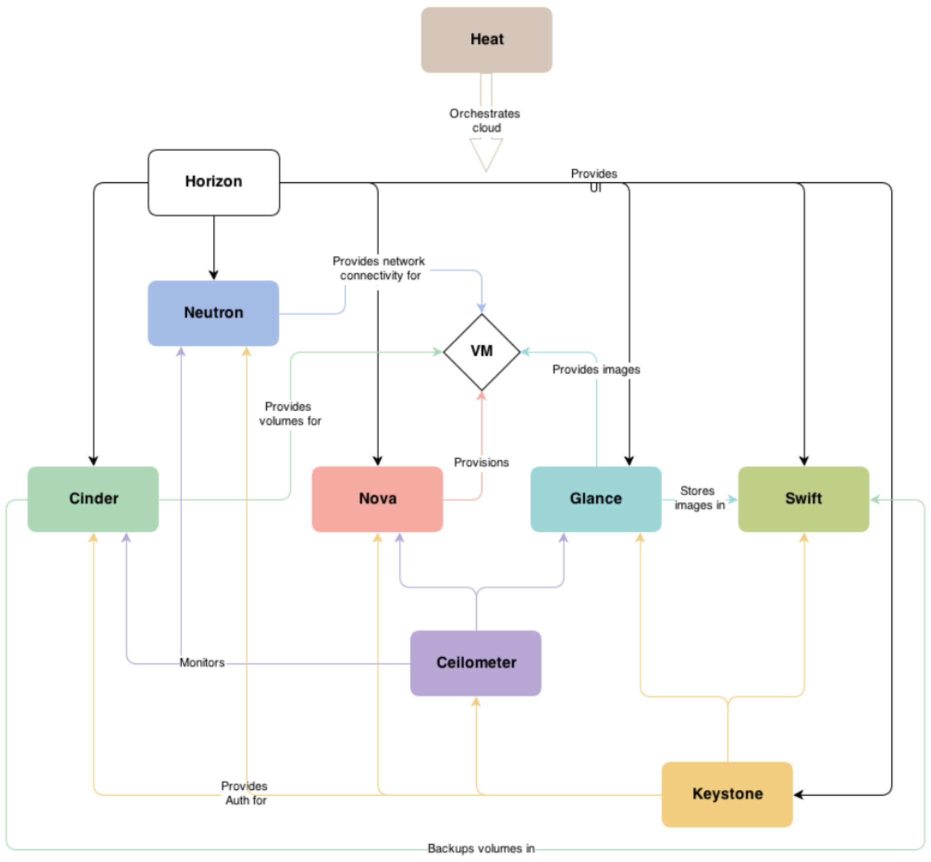


Figure 2.12: Interaction among OpenStack services
[18]

2.2.2 OpenStack Compute (Nova)

Nova is used to provision and manage large amounts of virtual machines . It supports different hypervisors and the number of physical hosts running the compute services can be scaled horizontally with no requirement of hardware resources from specific vendors. Hosts that provide Nova services are also called 'Compute Nodes'. Data center can be divided into so called tenants, which are isolated users with their own servers, security groups and externally reachable IP addresses (Floating IP addresses).

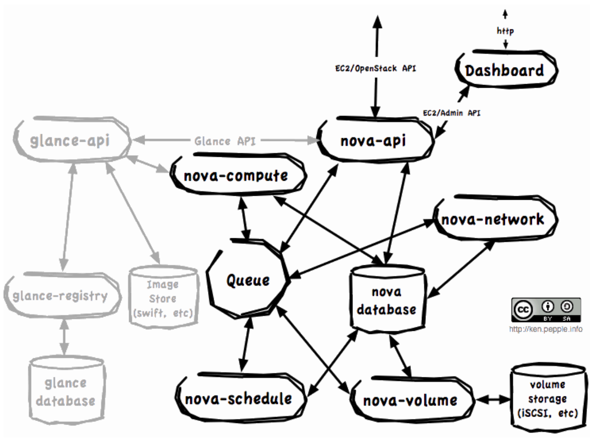


Figure 2.13: OpenStack Compute service
[19]

Compute Node segregation

An OpenStack cloud can be logically and physically grouped on different levels [16]:

- **Region:** A Region has its own full OpenStack deployment and can be physically at a different location. Regions share a set of Keystone and Horizon services to provide access control and the graphical management interface.
- **Availability Zone:** Inside of a Region, it is possible to logically group multiple compute nodes into Availability Zones. This zone can be specified when new servers or stacks (via Heat) are instantiated.
- **Host Aggregates:** Compute nodes can also be logically grouped into Host Aggregates by using meta-data to tag them. This feature can be used to separate nodes with certain hardware characteristics (e.g. with SSD drives) from others.

For zoning compute nodes availability zones will be used in the Connectivity Manager in order to achieve the best networking performance between individual servers.

Default VM Placement

Nova makes use of the nova-scheduler service in order to regulate on which host a new VM is launched. The nova-scheduler interacts with other components through the queue and central database repo. All of the compute nodes periodically send an update about their status, available resources and hardware capabilities to the nova-scheduler. It collects this data to make a decision about the placement based on a filter scheduler which takes the following criteria into account:

- are within the requested availability zone
- have sufficient RAM available
- are capable of servicing the request

By default Nova allows its RAM and vCPU resources to be overcommited, which means that more than the physically available resources can be deployed. However this is to the detriment of their performance.

2.2.3 OpenStack Orchestration (Heat)

Heat provides a template-based orchestration service for creating and managing cloud resources. This means multiple OpenStack resource types (such as virtual machines, floating IP addresses, volumes, security groups and users) can be generated and also maintained with additional functionality like auto-scaling.

2.2.4 OpenStack Neutron

In the early versions of OpenStack, virtual networking was exclusively a sub-component of Nova called Nova-network. This service has its limitations, because it is closely coupled with networking abstractions and no API available. With Neutron the implementation is decoupled from the network abstraction and it provides a flexible management interface to administrators and users.

2.2.4.1 Networking Concepts

Neutron is responsible for defining network connectivity and addressing within OpenStack. In the main network abstraction the following components are defined: A network as a virtual layer 2 segment, a subnet as a layer 3 IP address space used within a network, a port as an interface to a network or subnet, a router that performs address translation and routing between subnets, a DHCP server responsible for IP adress distribution, a security group for filtering rules acting as a cloud firewall and Floating IP addresses to give VMs external network access.

Neutron exposes an extensible set of APIs for creating and administering its components. Neutron consists of the following elements [20]:

- **neutron-server:** Provides the logic for SDN, however it does not contain any of its functionality in itself. It provides a generic API for the network operations, is modular and extended with the following agents.
- **L2 agent:** Plugin-specific agent that manages networking on a compute node. For more details, see Modular Layer 2 section.
- **DHCP agent:** Provides DHCP services to tenant networks through dnsmasq instances.
- **L3 agent:** Provides L3/NAT forwarding to allow external network access for VMs (virtual routers).
- **Metadata agent:** Acts as a proxy to the metadata service of Nova.

The different agents can interact with the neutron-server process through a message queue and the OpenStack Networking API. In most use-cases the neutron-server and the different agents can run on the controller node or on a separate network controller node, with the plugin agent running on each compute node.

2.2.4.2 Modular Layer 2

The ML2 plugin is a framework that allows the simultaneous usage of multiple layer 2 networking technologies. It also helps vendors develop new agents, because an abstract method for their services is already specified.

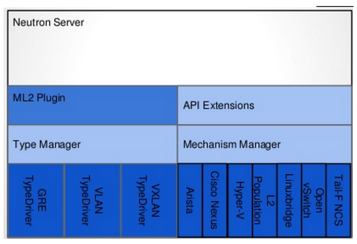


Figure 2.14: Neutron modular framework, including ML2 drivers

The plugin integrates the type driver and the mechanism driver. The type driver defines the network types that can be declared when a new network is created and currently includes: local, flat, vlan, gre and vxlan. The mechanism driver enables the mechanism for accessing these network types, i.e. Open vSwitch, Linux Bridge or other vendor-specific solutions.

ML2: Open vSwitch

Open vSwitch is the ML2 mechanism driver that is set as default when installing OpenStack with Neutron enabled using the devstack script. It is also the most commonly deployed agent.

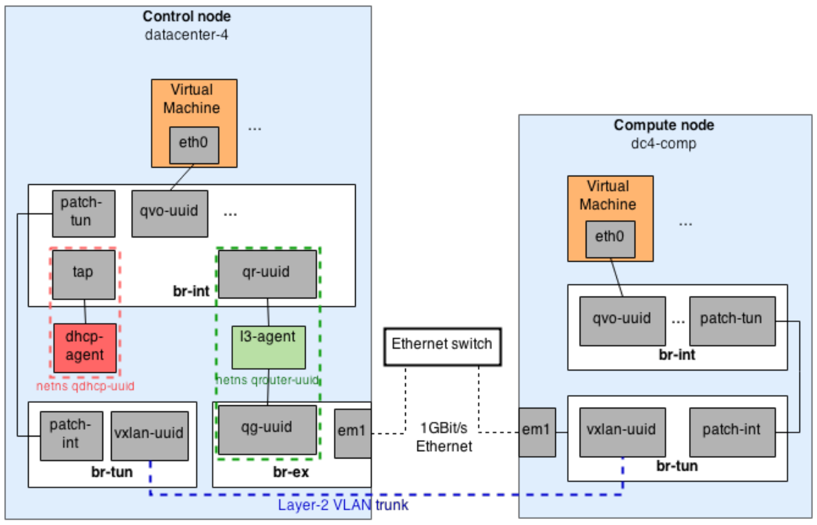


Figure 2.15: Neutron with OVS and VXLAN enabled

The above figure shows the connectivity of a VM, which has a 'qvo'-Port located in the internal bridge of the OVS. Each network within Neutron has its own network namespace and the external gateway uses the 'qrouter' namespace for routing. The connection between the compute nodes in this example uses VXLAN as the network type driver.

2.2.4.3 Network Distinction

Neutron is connected to different networks. For internet routable connections the external network is used. The management network is created by the network operator and is mapped to pre-existing networks within the datacenter, which is used to connect the different hosts. The tenants within OpenStack have their own isolated self provisioned private networks. Those can optionally be connected to other tenant or external networks. The abstraction of those tenant networks is possible through network namespaces. This allows overlapping IP addresses within the datacenter.

2.2.4.4 Neutron Workflow

From the time when Neutron is started as a process within the OpenStack controller node to booting a virtual machine, it passes through the following steps:

1. Start Neutron-Server
2. Start Open vSwitch Agent
3. Start L3-Agent
4. Start DHCP-Agent
5. Start Metadata-Agent
6. Create Networks
7. Create Routers
8. Boot VMs

Once the VM is started it has full network connectivity within the tenants network and if a floating IP address was attached to it, then it can also be reached from the internet.

2.3 Conclusion

This chapter laid out the technological fundamentals for this thesis. Starting with the motivation for the use of SDN and the difference to the traditional networking approach and going on to describe the OpenFlow standard which forms the basis for the work of the Open Networking Foundation. The Open vSwitch makes full use of the OpenFlow standard and protocol and forms the basis for the network switching infrastructure used in this thesis. Furthermore it provides the service for enabling network traffic shaping which is described on a deep technical level. All necessary components that are used in connection to this work are introduced.

Chapter 3

Requirements

3.1 Functional Requirements

This section identifies the functional requirements of the Connectivity Manager, specifically as needed for the NUBOMEDIA use-case.

3.1.1 Service-Level Agreement Enforcement

One of the key objectives of the Connectivity Manager is to grant different Service-Level Agreements (SLA) to the links between Virtual Machines. The agreement is set as Quality of Service assurances with the minimal and maximum bandwidth rate set. Network performance problems can provide a negative experience for the end-user, as well as productivity and economic loss. This is why some services require an assured bandwidth rate.

3.1.2 Optimal Virtual Machine Placement

The placement of Virtual Machines makes a tremendous difference in terms of their network and overall resource performance. It is important to evaluate which placement makes for the best-available network bandwidth between VMs within the internal network. The current utilization of hosts need to be taken into account as well. A stack should only be deployed if the resources are available at the time of deployment without overcommitting any hardware resources.

3.1.3 Integration with Elastic Media Manager

The Connectivity Manager needs to integrate with the Elastic Media Manager (EMM) which is used for deploying a topology of resources within a cloud infrastructure. Furthermore it provisions the instances and manages them during their runtime through services such as upscaling the amount of instances after certain utilization alarms are triggered. The CM communicates with the EMM in order to enable the two previously-mentioned requirements for the overall platform.

3.2 Non-functional Requirements

Non-functional requirements generally specify criteria in relation to the operation of a system and not in relation to its behavior. Thus the Connectivity Manager should also fit the following characteristics.

3.2.1 Scalability

Today's data-centers can grow at a fast-pace, especially in connection with automated up-scaling of compute resources at a certain level of utilization. This is why the underlying virtualized network software needs to be scalable too.

3.2.2 Modularity

Building modular software not only simplifies further development for a third-party, but also makes it easier to exchange certain parts of the software for improvements or maintenance at a later date. The separation into two different components with a defined API makes it more flexible.

3.2.3 Interoperability

In the case of the use of Open vSwitch, interoperability is given because of the availability for various architectures. The integration into the Linux Kernel and the use of standardized protocols such as OpenFlow are a significant factor.

Chapter 4

State of the Art

4.1 Overview

Three existing solutions for extending the services that Neutron provides with additional SDN features have been tested. They were selected based on the requirements given in the previous section.

4.2 OpenDaylight SDN Controller

OpenDaylight is fully implemented in Java. 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 [21]. The Open vSwitch database (OVSDB) Plugin component for OpenDaylight implements the OVSDB management protocol that allows the configuration of Open vSwitches. The OpenDaylight controller uses the native OVSDB implementation to manipulate the OVSDB. The component comprises a library and various plugins. The OVSDB protocol uses JSON/RPC calls to manipulate a physical or virtual switch that has OVSDB attached to it [22].

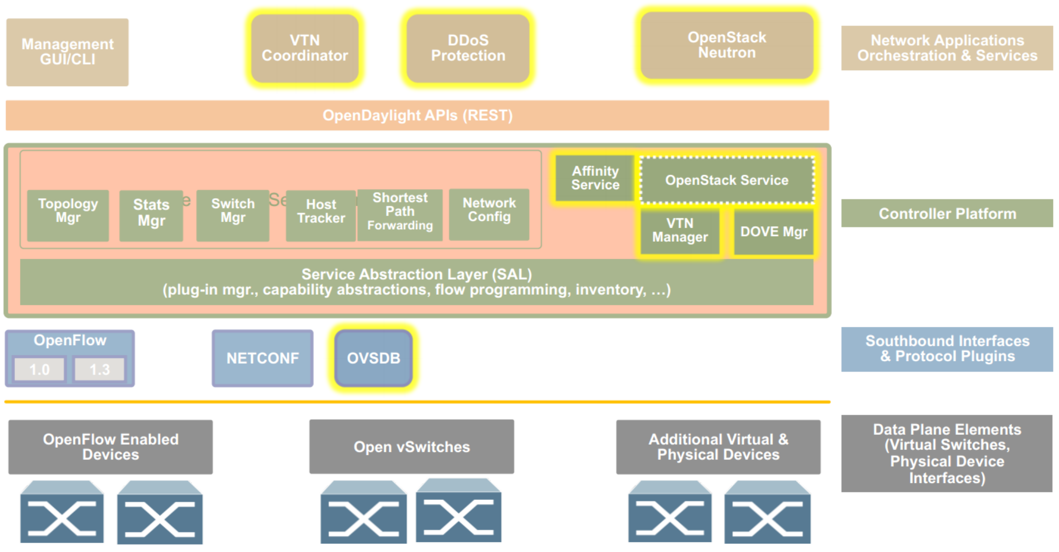


Figure 4.1: Architecture of OpenDaylight Virtualization edition

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 virtual network ports within OpenFlow Queues). In the local testbed it was possible to successfully deploy QoS rules on the ports of Virtual Machines.

4.3 Ryu SDN Controller

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 [23]. Ryu is a full featured OpenFlow controller that supports GRE and VLAN tunnelling. The OpenFlow controller that is embedded in the agent sets Flows on the switch by sending OpenFlow messages to the switch [24]. It includes a set of apps which build the base of the SDN controller: 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.

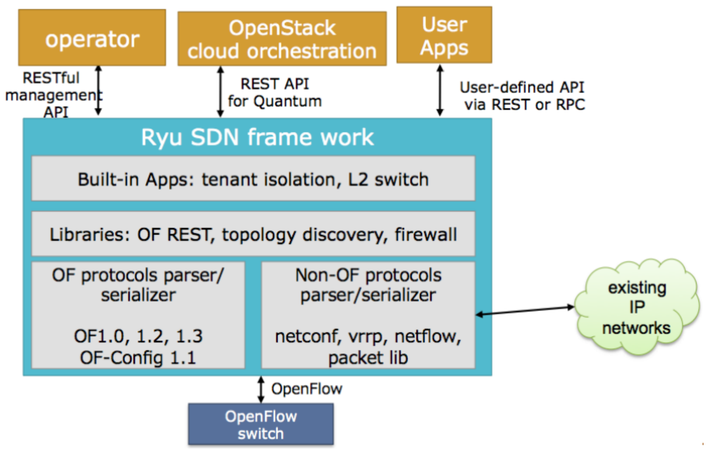

operator
OpenStack cloud orchestration
User Apps
RESTful management API
REST API for Quantum
User-defined API via REST or RPC
Ryu SDN frame work
Built-in Apps: tenant isolation, L2 switch
Libraries: OF REST, topology discovery, firewall
OF protocols parser/ serializer
OF1.0, 1.2, 1.3
OF-Config 1.1
Non-OF protocols parser/serializer
netconf, vrrp, netflow, packet lib
existing IP networks
OpenFlow
OpenFlow switch

Figure 4.2: Ryu architecture

As of OpenStack IceHouse Ryu makes use of the OFagent and is included in the Neutron core git repository. In order to use it as the SDN framework within Neutron, the 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).

4.4 OpenStack Neutron - QoS Extension

A Neutron extension has been partially implemented for OpenStack IceHouse which includes an API for setting QoS on a per-tenant and per-port basis in combination with the Open vSwitch agent [25]. This approach makes a lot of sense as it extends the already existing Neutron API and the framework that is given for custom Neutron Extensions.

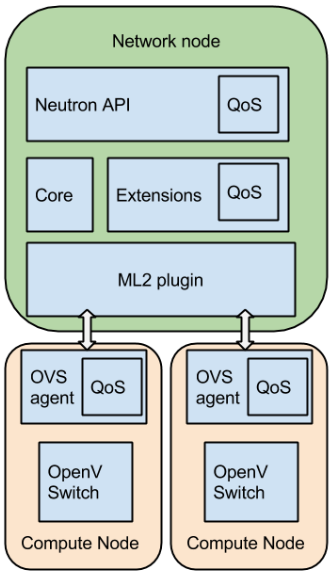


Figure 4.3: Neutron QoS Extension architecture

4.5 Problem Statement

This section lists the restrictions that have been discovered with the previously mentioned solutions.

Problems encountered with OpenDaylight:

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, thus the L3 routing did not work. 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.

Problems encountered with Ryu:

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.

Problems encountered with Neutron QoS Extension:

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 community will keep working on this. According to the code review comments, it was deferred to Juno, but it is 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 Open vSwitch agent and an addition to the Neutron Database that includes QoS.

For the scope of this work, extending Neutron with QoS is not within the time frame of the project.

4.6 Conclusion

The listed problems further strengthen the motivation for the implementation of the Connectivity Manager. Additionally, the second major requirement for the Optimal Virtual Machine Placement are not given in any of the outlined solutions. The following table lists a comparison and analysis of their features:

Tool/Solution	QoS support	OF Controller	OpenStack (Juno) integration
Ryu	X	X	No (tests in Juno failed)
OpenDaylight	X	X	No (tests in Juno failed)
Neutron QoS extension	not fully implemented	No	No (IceHouse patch not ported to Juno)

Table 4.1: State-of-the-art: Feature comparison

Chapter 5

Design

5.1 Architecture Overview

The Connectivity Manager is logically located between the EMM and the cloud infrastructure and provides the following two functionalities:

- **Optimal Instance Placement:** During the deployment of a stack an algorithm chooses where individual instances are placed within the cloud infrastructure.
- **Service-Level Agreement Enforcement:** Depending on the services that an instance provides to the rest of the stack, certain requirements for its network performance need to be fulfilled.

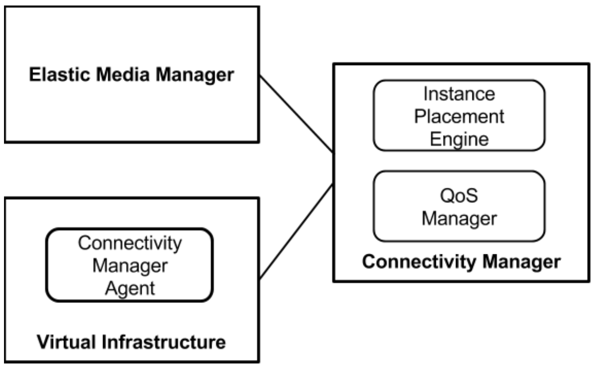


Figure 5.1: High-level architecture of the Connectivity Manager

The *Instance Placement Engine* determines if and where the instances should be deployed. It does so by comparing the current utilization and capacity of the available compute nodes within the availability zone.

The *QoS Manager* enforces different QoS policies based on the type of service that the instance is grouped in. A guaranteed and maximum bit-rate for the network port of an instance can be set. This way a certain network performance can be insured opposed to the commonplace best-effort packet delivery.

5.2 Interface between CM Manager & Agent

The Connectivity Manager and Agent are two separate applications that communicate using a ReST API.

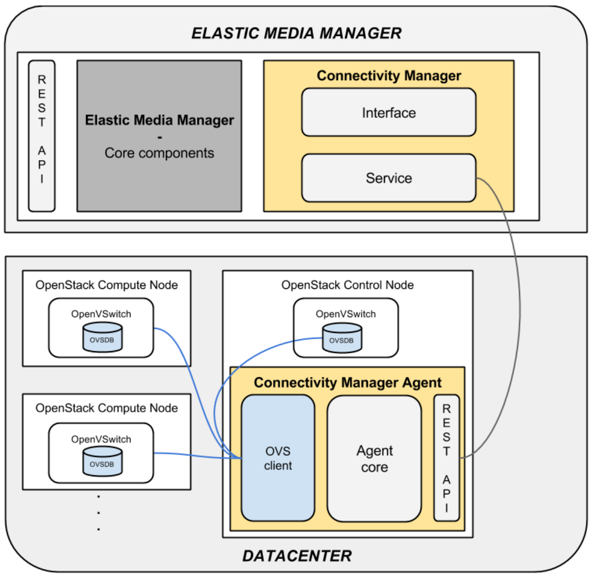
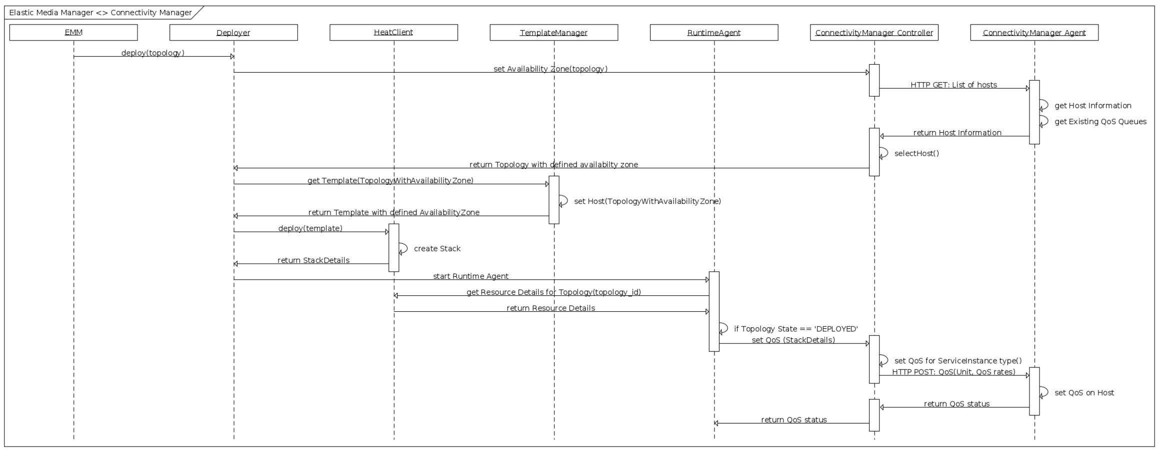


Figure 5.2: Minimized architecture of the Connectivity Manager and its integrations

This design was chosen firstly because the Connectivity Manager is integrated within the EMM, which is required to be placed outside of the data center. Secondly, the Connectivity Manager Agent needs access to the OVSDB on the compute nodes and consequently needs to be within the internal management network of the OpenStack infrastructure.

The sequence diagram below displays the work-flow that the CM passes during the run-time.



Elastic Media Manager <> Connectivity Manager
EMM
Deployer
HeatClient
TemplateManager
RuntimeAgent
ConnectivityManager Controller
ConnectivityManager Agent
deploy(topology)
set Availability Zone(topology)
HTTP GET: List of hosts
get Host Information
get Existing QoS Queues
return Host Information
selectHost()
return Topology with defined availabilty zone
get Template(TopologyWithAvailabilityZone)
set Host(TopologyWithAvailabilityZone)
return Template with defined AvailabilityZone
deploy(template)
create Stack
return StackDetails
start Runtime Agent
get Resource Details for Topology(topology_id)
return Resource Details
if Topology State == 'DEPLOYED'
set QoS (StackDetails)
set QoS for ServiceInstance type()
HTTP POST: QoS(Unit, QoS rates)
set QoS on Host
return QoS status
return QoS status

Figure 5.3: Deployment of a topology & assignment of QoS policies

As visible in the above figure, the Connectivity Manager receives the Topology object that contains a description of the configuration and specifications of the whole stack. For the placement decision the CM to needs to get the information about the current state of the infrastructure. This exchange with the CM Agent occurs through the HTTP API. Upon reception of that data, the placement algorithm sets the availability zone for each instance within the topology. The topology is then converted into a Heat template by the Template Manager. Once the template was deployed by the Heat Client, the runtime agent starts. The purpose of the runtime agent is to continuously check the state of the stack. Once the stack has reached the 'DEPLOYED' state, the runtime agent requests the CM to set the QoS policies according to previously configured values. This configuration is subsequently transmitted to the CM Agent whose task is to then enable the policies on the associated ports of the instances within the Open vSwitches.

5.3 Design of Connectivity Manager

The design of the Connectivity Manager is based on the framework that already exists in the Elastic Media Manager.

It consists of a highly dynamic factory, abstract interfaces and the actual implementation as a service. The Factory Agent reads a configuration file in which the class name of the service for the associated interface is defined. It then instantiates a instance from that given class. This means it is easy to replace the actual implementation while the interface to the other components can remain unchanged.

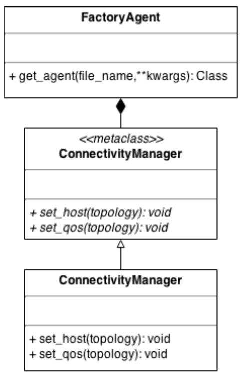

FactoryAgent
+ get_agent(file_name,**kwargs): Class
<<metaclass>>
ConnectivityManager
+ set_host(topology): void
+ set_qos(topology): void
ConnectivityManager
+ set_host(topology): void
+ set_qos(topology): void

Figure 5.4: Metaclass (Interface) and implementation (Service) instantiated by Factory Agent

The Connectivity Manager interface and service only contain the methods that receive calls from the Elastic Media Manager. However, additional helper classes are needed in order to provide the appropriate configuration and the communication with the Connectivity Manager Agent, as shown in the next figure.

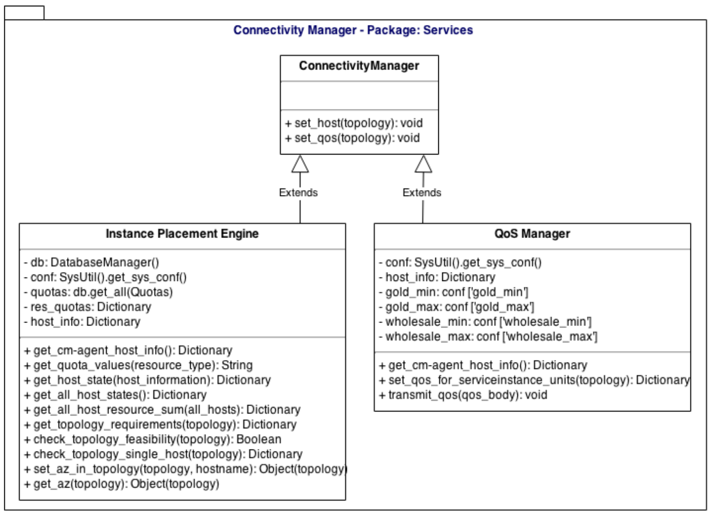

Connectivity Manager - Package: Services
ConnectivityManager
+ set_host(topology): void
+ set_qos(topology): void
Extends
Extends
Instance Placement Engine
- db: DatabaseManager()
- conf: SysUtil().get_sys_conf()
- quotas: db.get_all(Quotas)
- res_quotas: Dictionary
- host_info: Dictionary
+ get_cm-agent_host_info(): Dictionary
+ get_quota_values(resource_type): String
+ get_host_state(host_information): Dictionary
+ get_all_host_states(): Dictionary
+ get_all_host_resource_sum(all_hosts): Dictionary
+ get_topology_requirements(topology): Dictionary
+ check_topology_feasibility(topology): Boolean
+ check_topology_single_host(topology): Dictionary
+ set_az_in_topology(topology, hostname): Object(topology)
+ get_az(topology): Object(topology)
QoS Manager
- conf: SysUtil().get_sys_conf()
- host_info: Dictionary
- gold_min: conf ['gold_min']
- gold_max: conf ['gold_max']
- wholesale_min: conf ['wholesale_min']
- wholesale_max: conf ['wholesale_max']
+ get_cm-agent_host_info(): Dictionary
+ set_qos_for_serviceinstance_units(topology): Dictionary
+ transmit_qos(qos_body): void

Figure 5.5: Connectivity Manager service and its helper classes

The topology that is created by the EMM contains the following required resources for the stack:

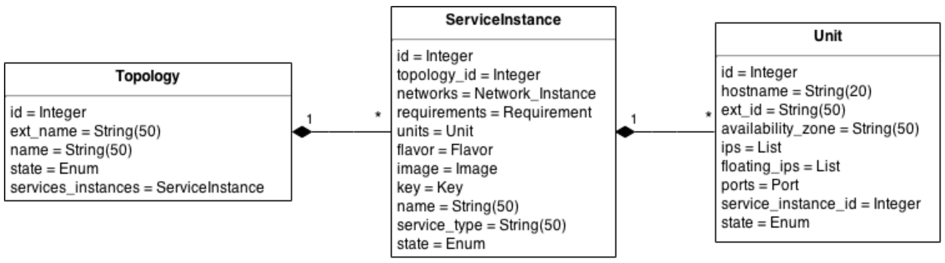


Figure 5.6: Deployment: Topology object from EMM

This object is handed over to the CM during the deployment phase. The topology can contain multiple Service Instances, which defines general parameters such as what networks the underlying instances need to be connected to, the flavor wherein the resources (e.g. amount of RAM & CPU) are specified [26], the image that contains the operating system and additional software packages plus the key that is needed to establish a connection to the server via SSH.

Each Service Instance can contain multiple Units (servers). Among others it contains parameters like IP addresses, the availability zone that it gets deployed on and the external ID which is unique across the whole infrastructure.

5.3.1 Algorithm for Instance Placement

The following section describes the algorithm that is used for setting the availability zone for the instances, and thus placing them in the most advantageous way based on the project requirements.

Each tenant has a set of quotas which limit the resource usage within the OpenStack infrastructure. Limitations can be made based on the following resource types (excerpt): Amount of instances, vCPU's, RAM, Floating IP addresses and fixed IP addresses [26].

Considering the requirement to have optimal connectivity between individual instances, it is preferable to position them on the same compute node, given that its resources allow this.

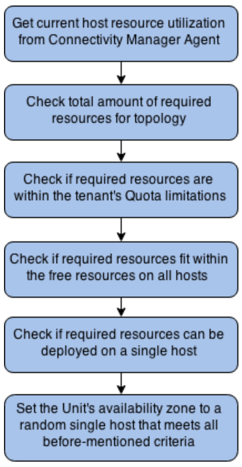

Get current host resource utilization from Connectivity Manager Agent
Check total amount of required resources for topology
Check if required resources are within the tenant's Quota limitations
Check if required resources fit within the free resources on all hosts
Check if required resources can be deployed on a single host
Set the Unit's availability zone to a random single host that meets all before-mentioned criteria

Figure 5.7: Instance Placement work-flow

5.3.2 QoS Manager

The QoS is deployed on the port of the internal bridge of the Open vSwitch that is connected to the virtual ethernet interface of a Unit. This design compared to deploying the same quality-of-service assurances across a whole tenant was chosen, because the traffic needs to be differentiated between the various service instances. The bandwidth rates are required to be configurable and therefore can't be hard-coded. By default the following rates and classes are set:

Name	Minimum rate	Maximum rate
Wholesale	100 MBit/s	1 GBit/s
Gold	100 MBit/s	10 GBit/s

Table 5.1: QoS classes and their default rates

All Units that are part of the 'Media Server' service instance are associated with the Gold class and all units of other types in the topology are part of the Wholesale class.

5.4 Design of Connectivity Manager Agent

The CM Agent is separated into the WSGI Application wherein the routes for the ReST API and the corresponding methods that will be called are defined. The Agent class calls the different Clients to get the status of the resources and make changes to their configuration.

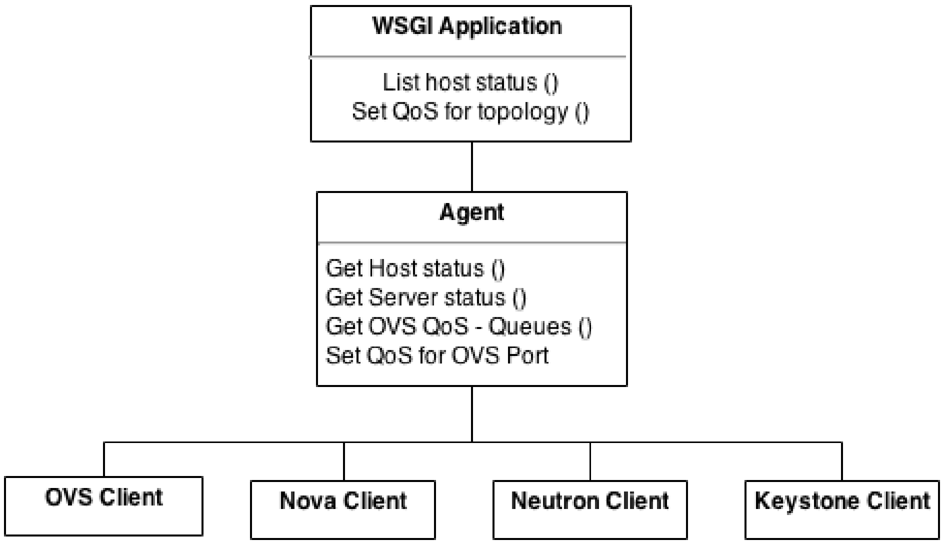


Figure 5.8: Abstract design of the Connectivity Manager Agent

For connecting to the OVSDB the OVS Client is needed on each host. It contains methods for listing all of its ports, interfaces, qos's and queues as well as for applying actions to a port, creating queues and creating new qos's.

The OpenStack Identity API is what the Keystone Client is connecting to in order to get the authentication endpoint and token for Neutron.

The Nova Client binds to the OpenStack Nova API and is required for getting the status of all Compute Nodes and the servers running on it.

Each server can have multiple network ports, which are managed by Neutron. The Neutron Client can retrieve a list of all ports for the servers that exist for a tenant.

Finally the OVS Client makes use of various calls to the OVS control tool (ovs-vsctl) for reading various tables of the OVSDB and making changes for enabling QoS on a per-port basis.

During the design process the following package structure was decided on:

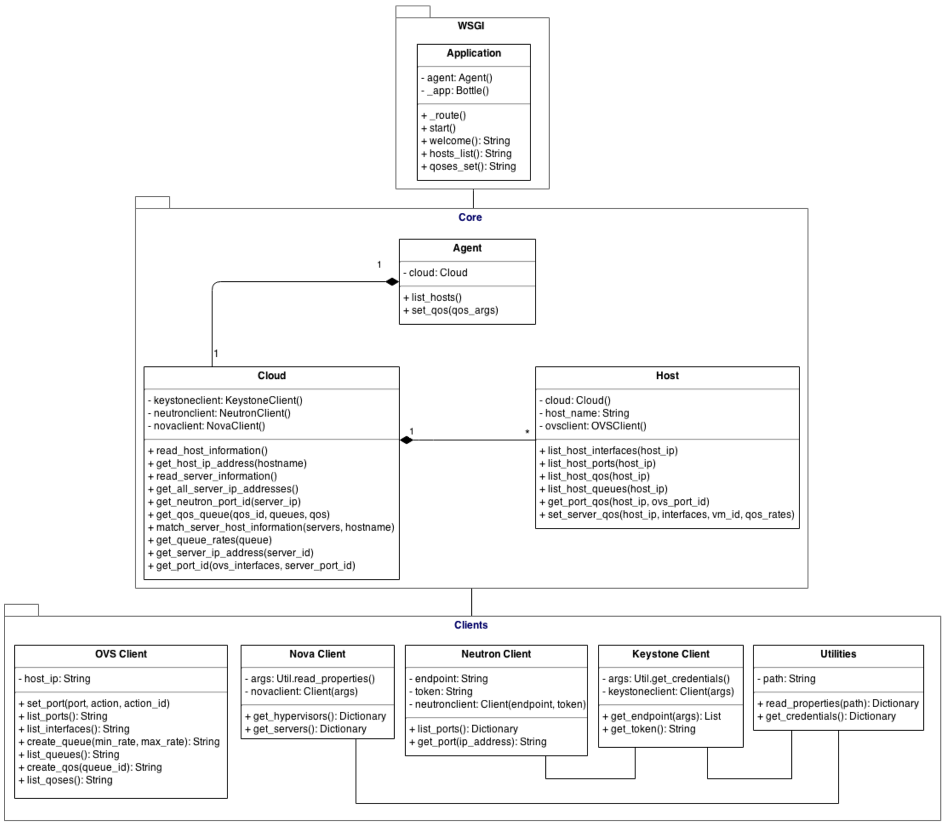

WSGI
Application
- agent: Agent()
- _app: Bottle()
+ _route()
+ start()
+ welcome(): String
+ hosts_list(): String
+ qoses_set(): String
Core
Agent
- cloud: Cloud
+ list_hosts()
+ set_qos(qos_args)
1
1
Cloud
- keystoneclient: KeystoneClient()
- neutronclient: NeutronClient()
- novaclient: NovaClient()
+ read_host_information()
+ get_host_ip_address(hostname)
+ read_server_information()
+ get_all_server_ip_addresses()
+ get_neutron_port_id(server_ip)
+ get_qos_queue(qos_id, queues, qos)
+ match_server_host_information(servers, hostname)
+ get_queue_rates(queue)
+ get_server_ip_address(server_id)
+ get_port_id(ovs_interfaces, server_port_id)
1
*
Host
- cloud: Cloud()
- host_name: String
- ovsclient: OVSClient()
+ list_host_interfaces(host_ip)
+ list_host_ports(host_ip)
+ list_host_qos(host_ip)
+ list_host_queues(host_ip)
+ get_port_qos(host_ip, ovs_port_id)
+ set_server_qos(host_ip, interfaces, vm_id, qos_rates)
Clients
OVS Client
- host_ip: String
+ set_port(port, action, action_id)
+ list_ports(): String
+ list_interfaces(): String
+ create_queue(min_rate, max_rate): String
+ list_queues(): String
+ create_qos(queue_id): String
+ list_qoses(): String
Nova Client
- args: Util.read_properties()
- novaclient: Client(args)
+ get_hypervisors(): Dictionary
+ get_servers(): Dictionary
Neutron Client
- endpoint: String
- token: String
- neutronclient: Client(endpoint, token)
+ list_ports(): Dictionary
+ get_port(ip_address): String
Keystone Client
- args: Util.get_credentials()
- keystoneclient: Client(args)
+ get_endpoint(args): List
+ get_token(): String
Utilities
- path: String
+ read_properties(path): Dictionary
+ get_credentials(): Dictionary

Figure 5.9: Class diagram: Connectivity Manager Agent - All packages

API

In the following table the ReST API that the Connectivity Manager Agent exposes to the Connectivity Manager is described:

HTTP method	Path	Method body	Description
GET	/hosts	-	List all available hosts and their resources
POST	/qoses	JSON: {'Hostname': {'Server ID': {'QoS rates'}}}	Set QoS rates for servers.

Table 5.2: Connectivity Manager Agent - API description

Chapter 6

Implementation

6.1 Environment

The software developed in this thesis is completely realized using the Python programming language. This choice was made because OpenStack offers Python clients that connect to their API and the Elastic Media Manager is also programmed in Python.

PyCharm was selected as the Integrated Development Environment (IDE) to simplify the programming and testing lifecycles. The code is under revision control using Git and the repository that contains both the code for the CM and CM Agent consists of two main branches: master and develop. The develop branch holds the latest changes and upon successful testing those were merged back into master, which is always in a production-ready state.

6.1.1 Project Structure

The code is separated into two different projects in order to allow testing of the integration concurrently. The following graphic outlines the focus within the Elastic Media Manager (EMM), whilst the CM Agent is separated and has its own structure.

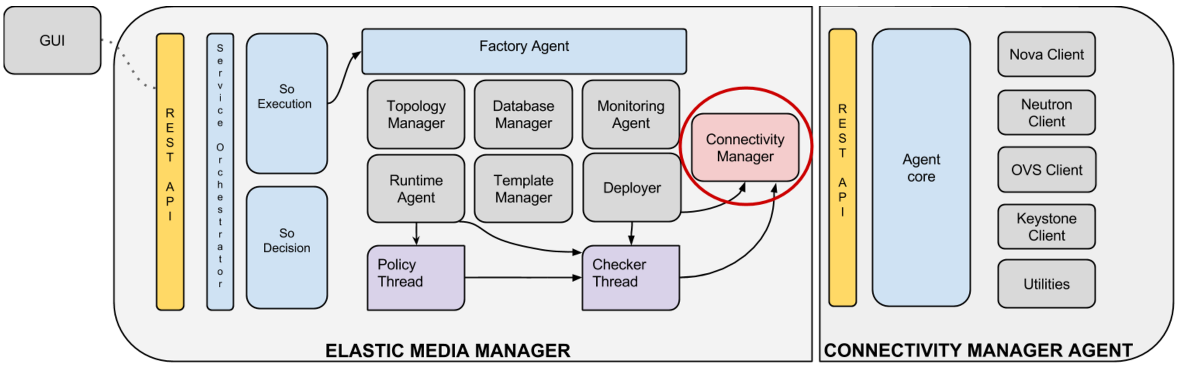


Figure 6.1: Implementation focus

The structure for the CM is dictated by the already existing implementation of the EMM and the scope of this thesis includes solely its extension with a Connectivity Manager interface and service plus the needed changes in the other interfaces to make use of the methods within the CM.

The Connectivity Manager contains the ReST API, the Agent core, clients and other helper classes.

6.1.2 Local OpenStack Test Environment

In order to test the Connectivity Manager Agent and the use of its OpenStack API clients a test-bed was installed. This test-bed was set up using Vagrant, as it allows starting of the virtual machines from the command-line and can easily be provisioned and managed. In order to test the software across multiple compute nodes a setup with 2 VM's was installed.

For the installation of OpenStack, the devstack script was used, which takes care of not only the deployment of the different components but also their configuration. The configuration parameters are set in the 'local.conf' file. For the OpenStack cluster controller the following configuration was used:

```
[[local|localrc]]
ADMIN_PASSWORD=pass
DATABASE_PASSWORD=pass
RABBIT_PASSWORD=pass
SERVICE_PASSWORD=pass
SERVICE_TOKEN=a682f596-76f3-11e3-b3b2-e716f9080d50

HOST_IP=192.168.120.15
OVS_PHYSICAL_BRIDGE=br-ex
MULTI_HOST=1

# Enable Logging
LOGFILE=/opt/stack/logs/stack.sh.log
VERBOSE=True
OFFLINE=True
RECLONE=no
LOG_COLOR=True
SCREEN_LOGDIR=/opt/stack/logs

# Neutron
disable_service n-net
enable_service q-svc
enable_service q-agt
enable_service q-dhcp
enable_service q-l3
enable_service q-meta

# OpenStack API paths
MYSQL_HOST=192.168.120.15
RABBIT_HOST=192.168.120.15
```

```
GLANCE_HOSTPORT=192.168.120.15:9292
KEYSTONE_AUTH_HOST=192.168.120.15
KEYSTONE_SERVICE_HOST=192.168.120.15

IMAGE_URLS="$IMAGE_URLS,http://cloud-images.ubuntu.com/releases
   /trusty/
release/ubuntu-14.04-server-cloudimg-amd64-disk1.img"
```

The configuration file for the second node, which solely runs Nova, the Open vSwitch agent and the Rabbit MQ is identical except for its enabled services and host IP address:

```
HOST_IP=192.168.120.16
ENABLED_SERVICES=n-cpu,rabbit,neutron,q-agt,q-l3
```

During the implementation and testing phase the Connectivity Manager Agent was installed on the controller node while the EMM was executed from the local machine.

6.2 Connectivity Manager - Components and Operations

6.2.1 Selection of Best-Performing Hypervisor

Step 1: Retrieve current host utilization from CM Agent

In order to decide on the placement, the Connectivity Manager firstly needs to retrieve the host information from the Agent. It then sums up the different resource utilizations: amount of servers currently running on it, the total amount of used RAM & vCPUs.

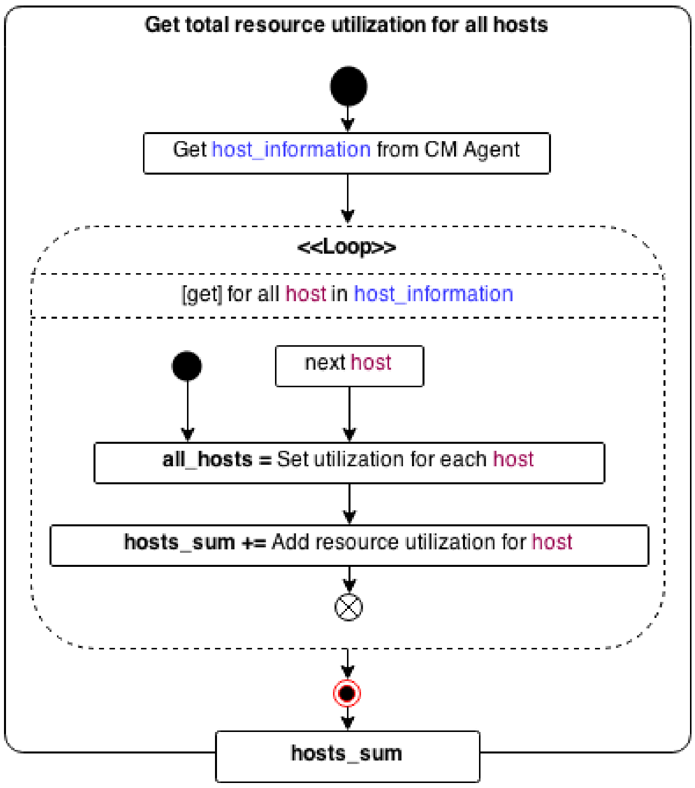

Get total resource utilization for all hosts
Get host_information from CM Agent
<<Loop>>
[get] for all host in host_information
next host
all_hosts = Set utilization for each host
hosts_sum += Add resource utilization for host
hosts_sum

Figure 6.2: Check resource utilization of hosts

The amount of resources that are required in total to deploy the topology on the tenant needs to be calculated by adding up the amount of resources that are needed for each Unit. This can easily be done by checking its flavor.

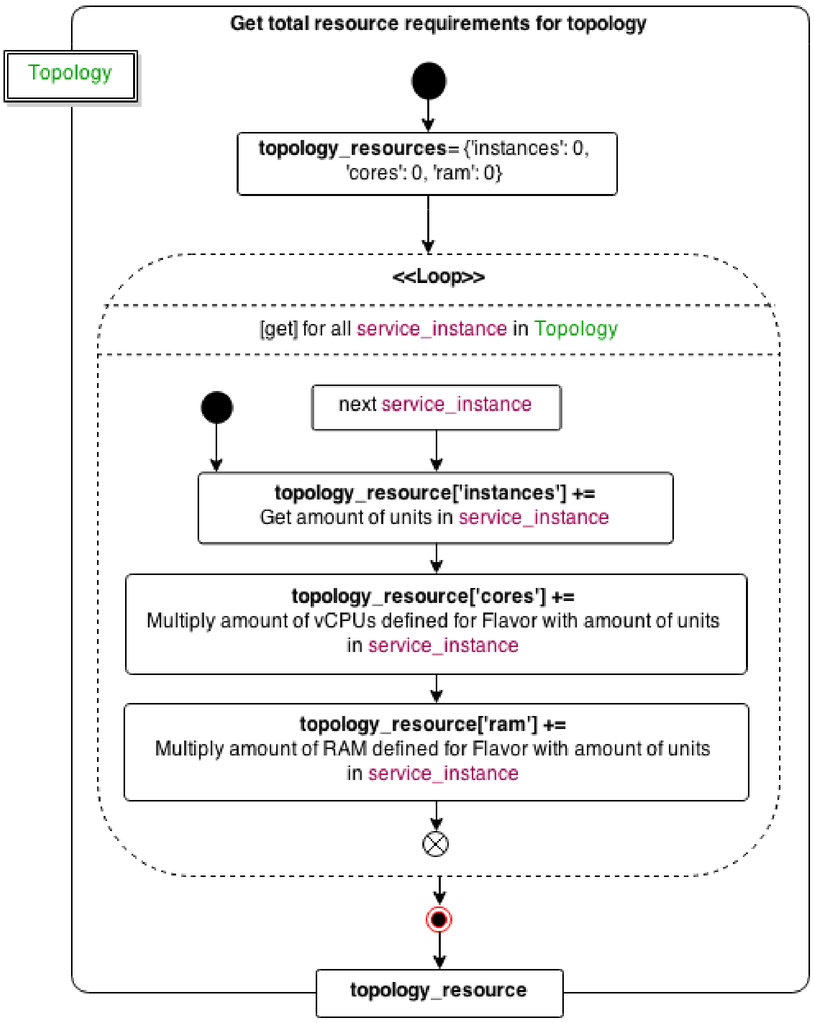
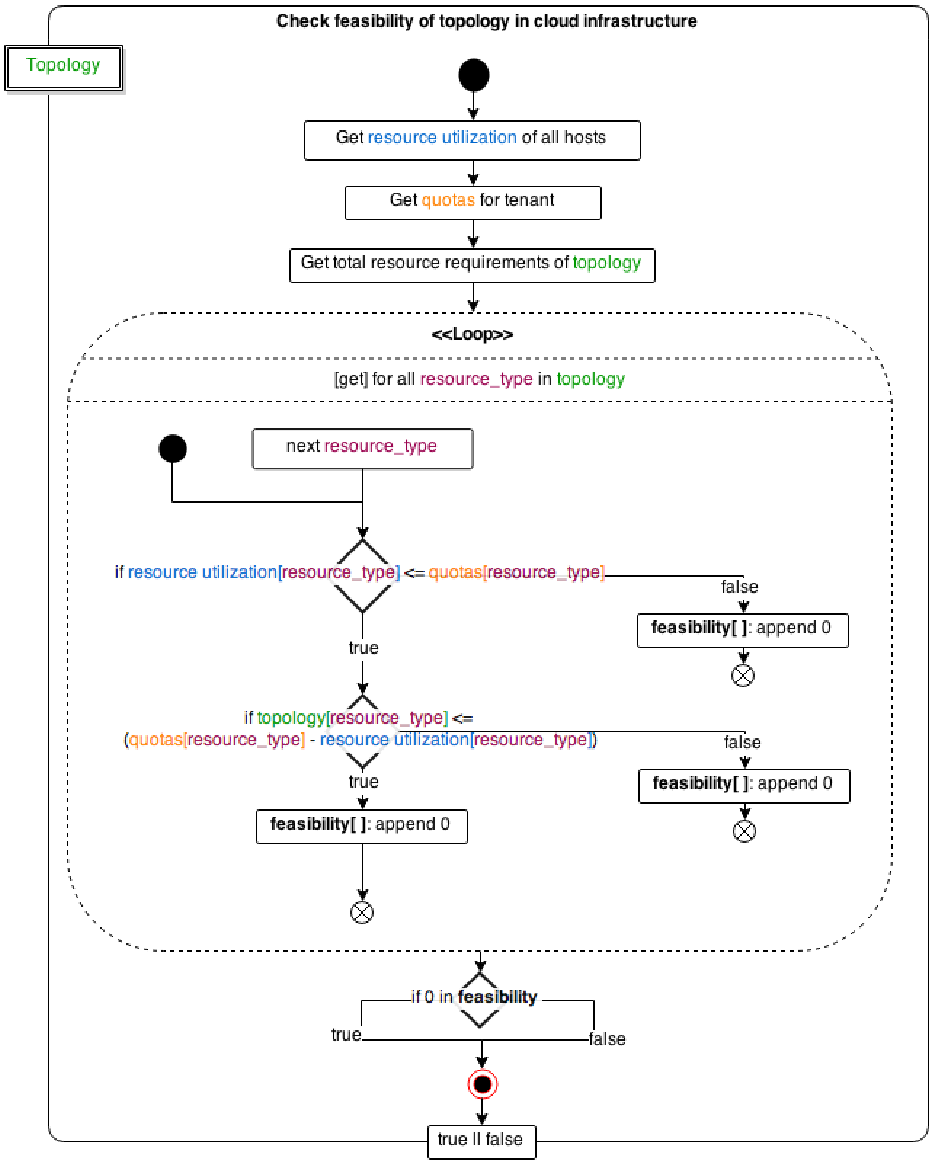

Get total resource requirements for topology
Topology
topology_resources= {'instances': 0, 'cores': 0, 'ram': 0}
<<Loop>>
[get] for all service_instance in Topology
next service_instance
topology_resource['instances'] += Get amount of units in service_instance
topology_resource['cores'] += Multiply amount of vCPUs defined for Flavor with amount of units in service_instance
topology_resource['ram'] += Multiply amount of RAM defined for Flavor with amount of units in service_instance
topology_resource

Figure 6.3: Get total amount of required resources for topology

Step 2: Check if Topology is within the limitations of the Quota and currently available resources on the tenants hosts.

In the second step it is checked if the topology is feasible for deployment. This decision is made by first comparing the values from Step 1 to the available Quota values (amount of VMs, RAM, and vCPUs required for the sum of all servers) and secondly finding out if the required resources are less or equal to the currently not utilized resources in the tenants infrastructure.



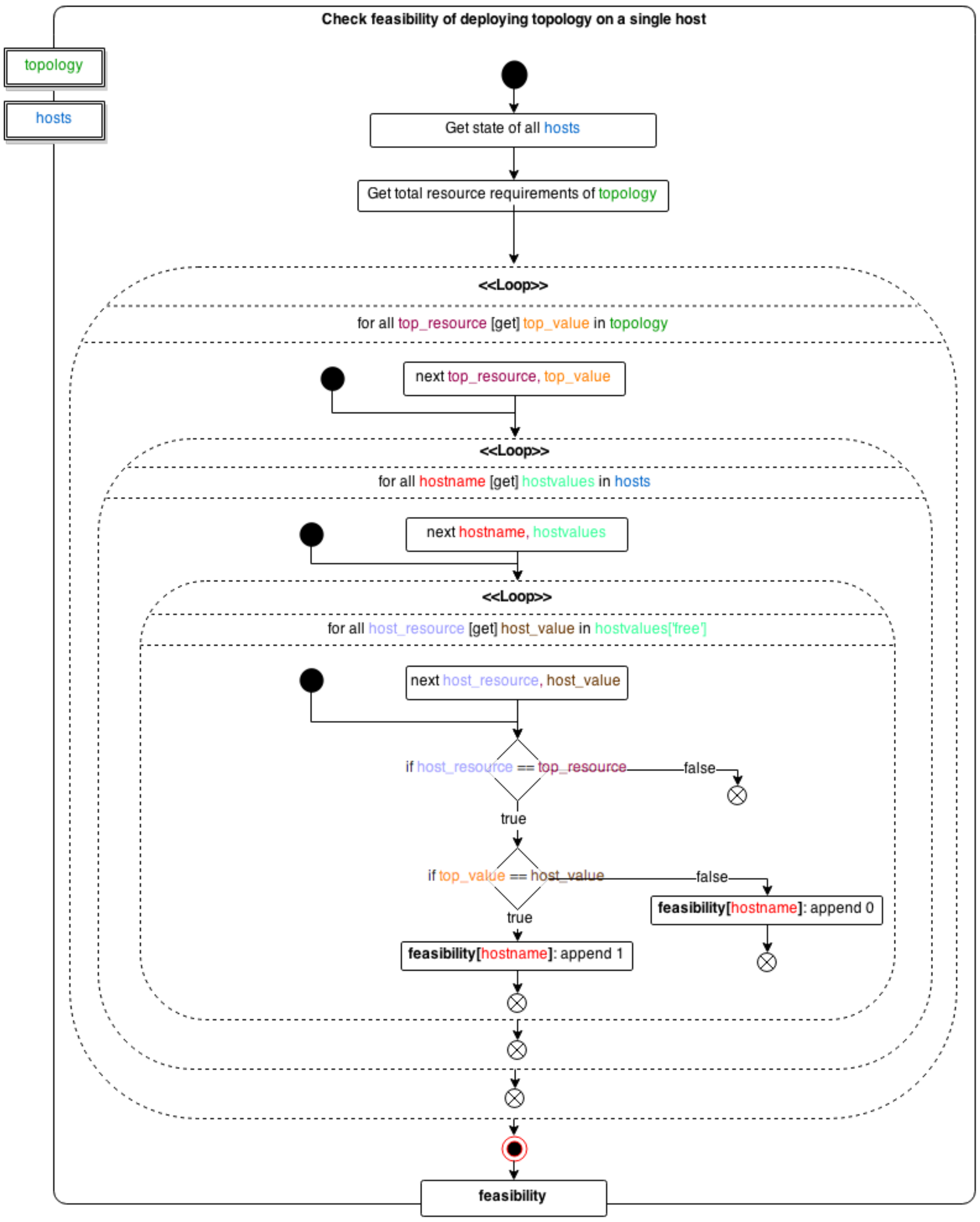
Check feasibility of topology in cloud infrastructure
Topology
Get resource utilization of all hosts
Get quotas for tenant
Get total resource requirements of topology
<<Loop>>
[get] for all resource_type in topology
next resource_type
if resource utilization[resource_type] <= quotas[resource_type]
false
feasibility[ ]: append 0
true
if topology[resource_type] <= (quotas[resource_type] - resource utilization[resource_type])
false
feasibility[ ]: append 0
true
feasibility[ ]: append 0
if 0 in feasibility
true
false
true ll false

Figure 6.4: Deployment feasibility check

These comparisons need to be made, in order to conform to the requirement of not over-committing any resources.

Step 3: Check whether the Topology can be deployed on a single host.

The next step checks if the total amount of topology resources from Step 1 can be deployed on a single host. If there are multiple hosts that have enough capacity available, the first one within the list is returned by this method.



Check feasibility of deploying topology on a single host
topology
hosts
Get state of all hosts
Get total resource requirements of topology
<<Loop>>
for all top_resource [get] top_value in topology
next top_resource, top_value
<<Loop>>
for all hostname [get] hostvalues in hosts
next hostname, hostvalues
<<Loop>>
for all host_resource [get] host_value in hostvalues['free']
next host_resource, host_value
if host_resource == top_resource
false
true
if top_value == host_value
false
true
feasibility[hostname]: append 0
feasibility[hostname]: append 1
feasibility

Figure 6.5: Single host deployment check

Step 4: Set selected host as availability zone for each Unit.

Lastly the chosen host needs to be set in the topology, so it can be returned to Heat for continuing with the deployment process. The syntax for a Unit's availability zone needs the 'nova' suffix, so this string needs to be added in front of the host name.

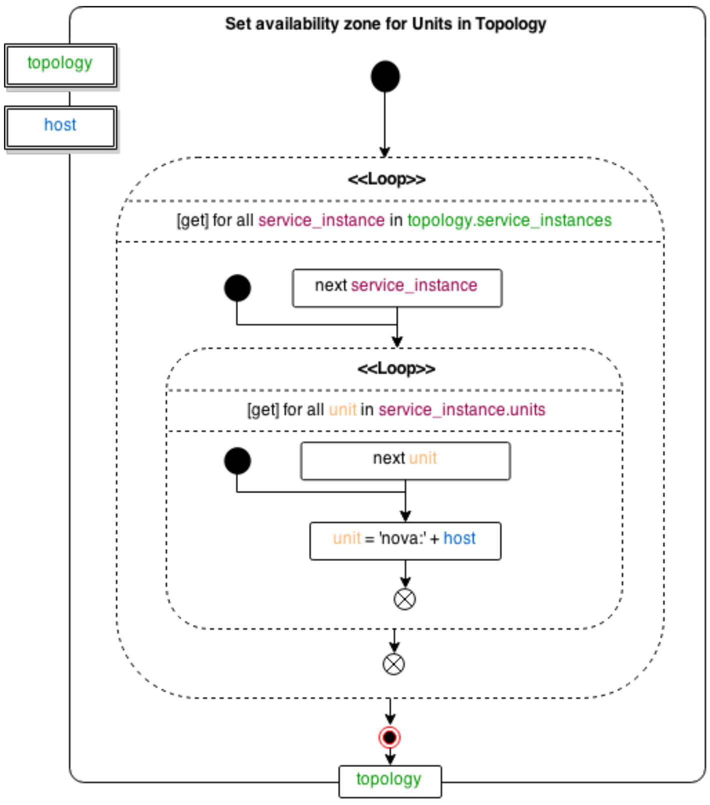

Set availability zone for Units in Topology
topology
host
<<Loop>>
[get] for all service_instance in topology.service_instances
next service_instance
<<Loop>>
[get] for all unit in service_instance.units
next unit
unit = 'nova:' + host
topology

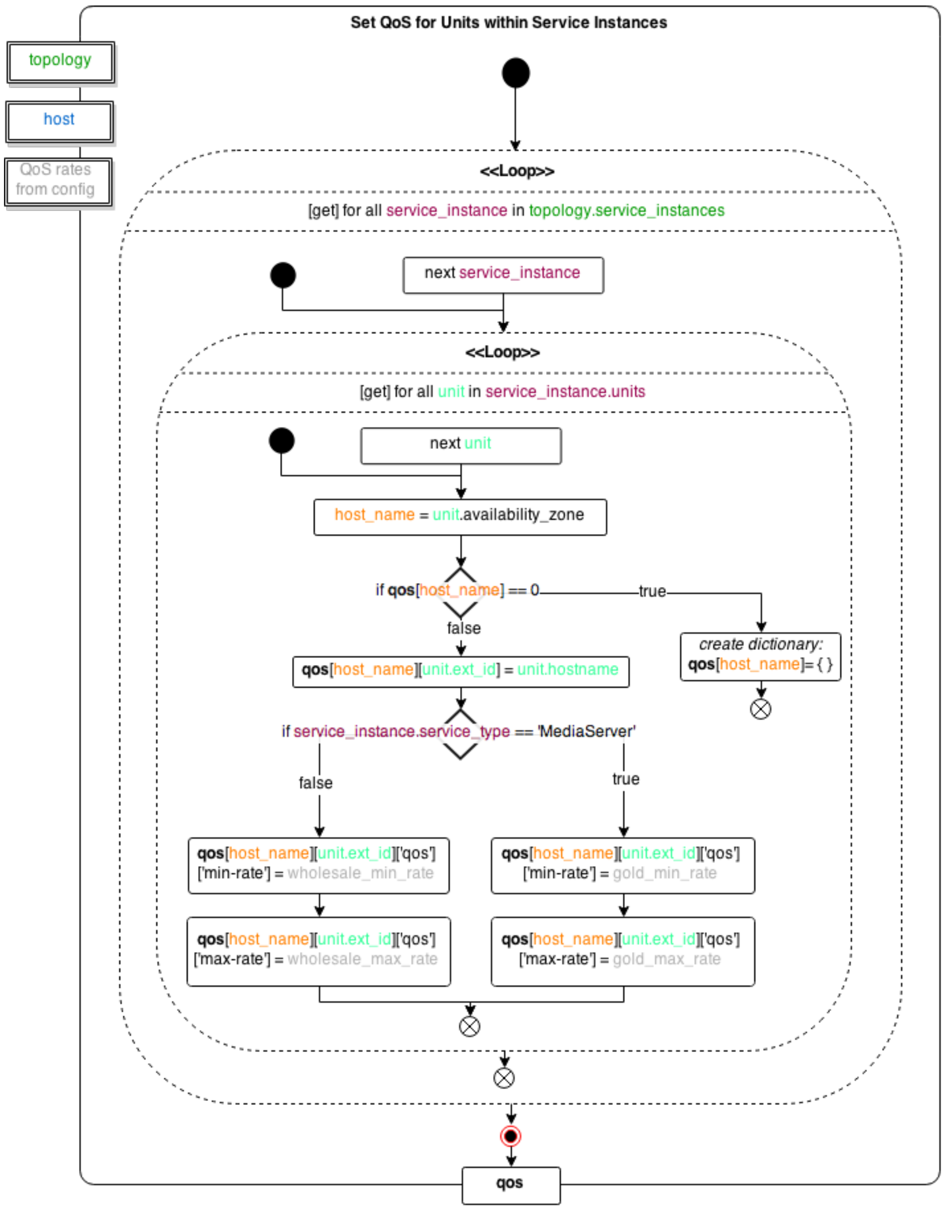
Figure 6.6: Set AZ per Unit

6.2.2 Enabling QoS for servers

The rates for the QoS classes can be set in the configuration of the EMM. The default location is */etc/nubomedia/emm.properties* and it contains the following default rates:

```
# CONNECTIVITY MANAGER PROPS & QOS RATES (IN BIT/S)
cm_agent_ip=192.168.41.45
# GOLD = 100MBIT/S - 10GBIT/S
gold_min=100000000
gold_max=10000000000
# WHOLESALE = 100MBIT/S - 1GBIT/S
wholesale_min=100000000
wholesale_max=1000000000
```

For setting QoS for all Units within a Service Instance, the following workflow is passed through:



Set QoS for Units within Service Instances
topology
host
QoS rates from config
<<Loop>>
[get] for all service_instance in topology.service_instances
next service_instance
<<Loop>>
[get] for all unit in service_instance.units
next unit
host_name = unit.availability_zone
if qos[host_name] == 0
true
false
create dictionary: qos[host_name]={ }
qos[host_name][unit.ext_id] = unit.hostname
if service_instance.service_type == 'MediaServer'
false
true
qos[host_name][unit.ext_id]['qos']['min-rate'] = wholesale_min_rate
qos[host_name][unit.ext_id]['qos']['max-rate'] = wholesale_max_rate
qos[host_name][unit.ext_id]['qos']['min-rate'] = gold_min_rate
qos[host_name][unit.ext_id]['qos']['max-rate'] = gold_max_rate
qos

Figure 6.7: Method for setting QoS for all Units

The dictionary that this method returns is then converted to the JSON format and sent to the Connectivity Manager Agent by performing a HTTP POST to the */qoses* path of the IP address that is set for the *cm_agent_ip* property in the configuration file.

6.3 Connectivity Manager Agent - Components and Operations

The two major operations that the Connectivity Manager needs to perform in order to provide the required services to this project is retrieving the resource status of the data center and setting QoS for the servers ports.

For using the OVS Client it needs to access the OVSDB that is located on each host. In order to allow that, the following command needs to be executed on them once:

```
$ sudo ovs-vsctl set-manager ptcp:6640
```

This enables remote administration through a passive TCP connection using the local IP address and port 6640.

6.3.1 Get list of all hosts and their utilization state

The following figure displays the way in which the CM Agent creates a dictionary containing the list of all hosts within the selected tenant. It contains information such as the amount of already running virtual machines, the amount of allocated RAM in MB and the number of vCPUs in use. For each of the virtual machines it shows their ID, name, and further resource information that is later needed for setting QoS. In case the VM already has a QoS assurance attached to its port, the corresponding minimal and maximal rates are shown.

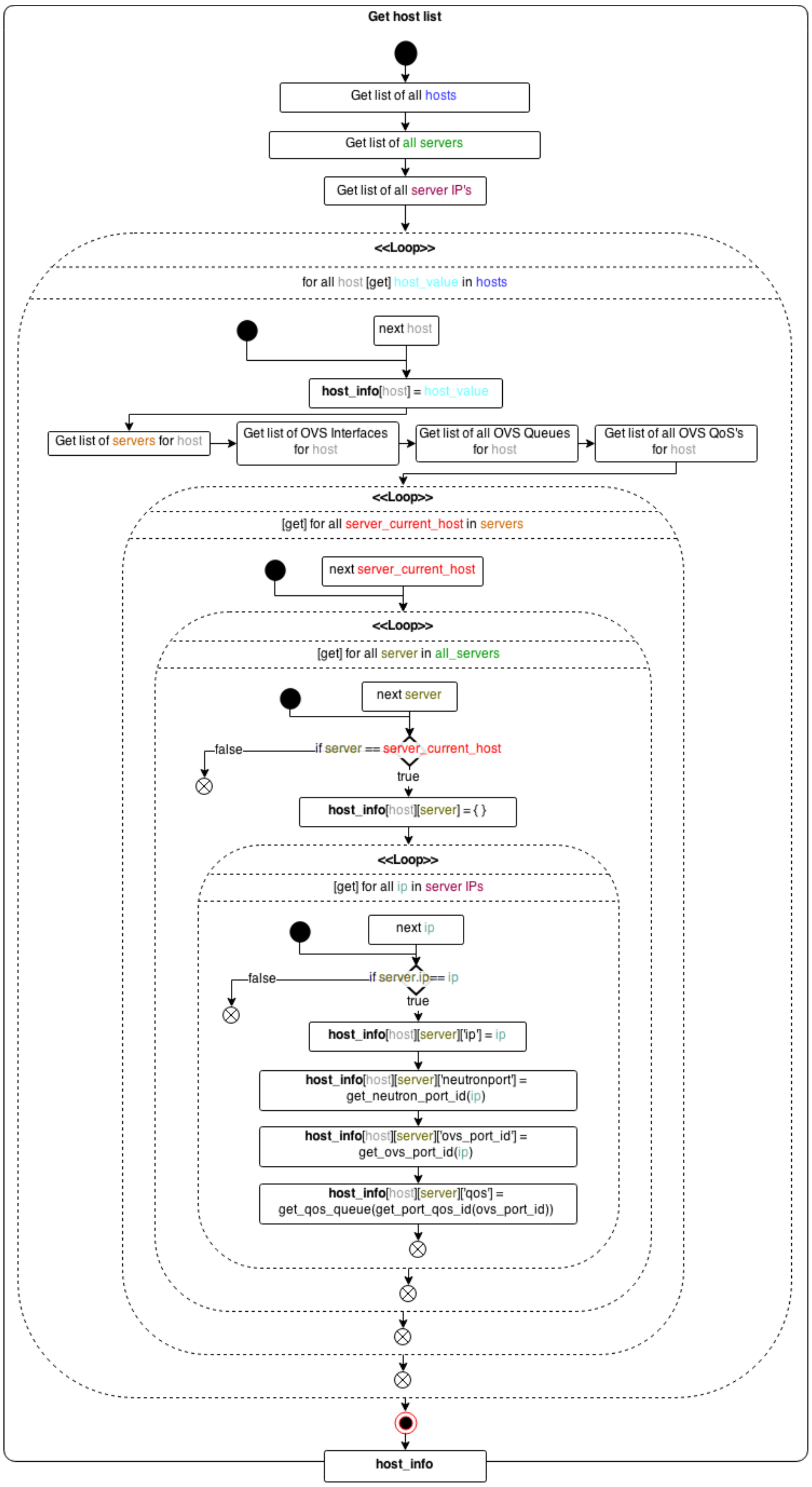

Get host list
Get list of all hosts
Get list of all servers
Get list of all server IP's
<<Loop>>
for all host [get] host_value in hosts
next host
host_info[host] = host_value
Get list of servers for host
Get list of OVS Interfaces for host
Get list of all OVS Queues for host
Get list of all OVS QoS's for host
<<Loop>>
[get] for all server_current_host in servers
next server_current_host
<<Loop>>
[get] for all server in all_servers
next server
if server == server_current_host
false
true
host_info[host][server] = { }
<<Loop>>
[get] for all ip in server IPs
next ip
if server.ip== ip
false
true
host_info[host][server]['ip'] = ip
host_info[host][server]['neutronport'] = get_neutron_port_id(ip)
host_info[host][server]['ovs_port_id'] = get_ovs_port_id(ip)
host_info[host][server]['qos'] = get_qos_queue(get_port_qos_id(ovs_port_id))
host_info

Figure 6.8: Get list of hosts

In order to identify the rates that might have already been attached to a VM's port, a few steps need to be taken. With the server's IP address the Neutron Port ID can be retrieved. This Neutron Port ID has a Port ID within Open vSwitch that can be retrieved from the OVSDB.

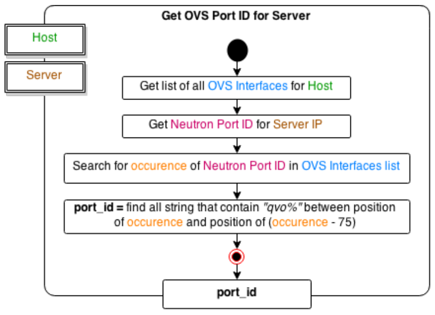

Get OVS Port ID for Server
Host
Server
Get list of all OVS Interfaces for Host
Get Neutron Port ID for Server IP
Search for occurence of Neutron Port ID in OVS Interfaces list
port_id = find all string that contain "qvo%" between position of occurence and position of (occurence - 75)
port_id

Figure 6.9: Get OVS Port ID for server

Each Port contains a field for QoS, which holds its QoS ID. This ID can then be filtered from the Queue table.

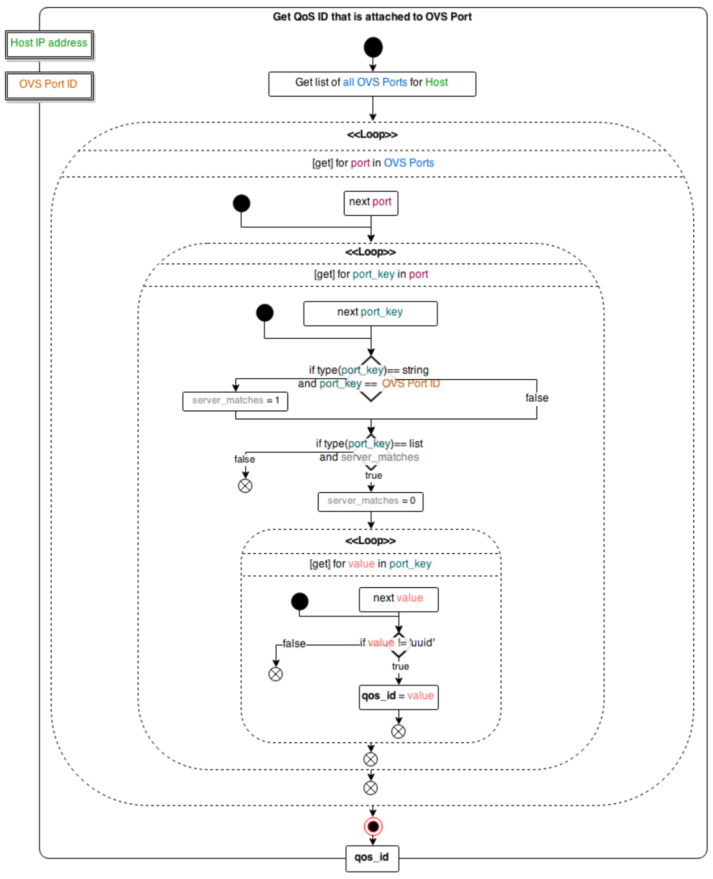

Get QoS ID that is attached to OVS Port
Host IP address
OVS Port ID
Get list of all OVS Ports for Host
<<Loop>>
[get] for port in OVS Ports
next port
<<Loop>>
[get] for port_key in port
next port_key
if type(port_key)== string and port_key == OVS Port ID
server_matches = 1
false
if type(port_key)== list and server_matches
false
true
server_matches = 0
<<Loop>>
[get] for value in port_key
next value
false
if value != 'uuid'
true
qos_id = value
qos_id

Figure 6.10: Get QoS ID for OVS Port

The QoS ID has a Queue attached to it, wherein the set minimum and maximum bandwidth rates are specified.

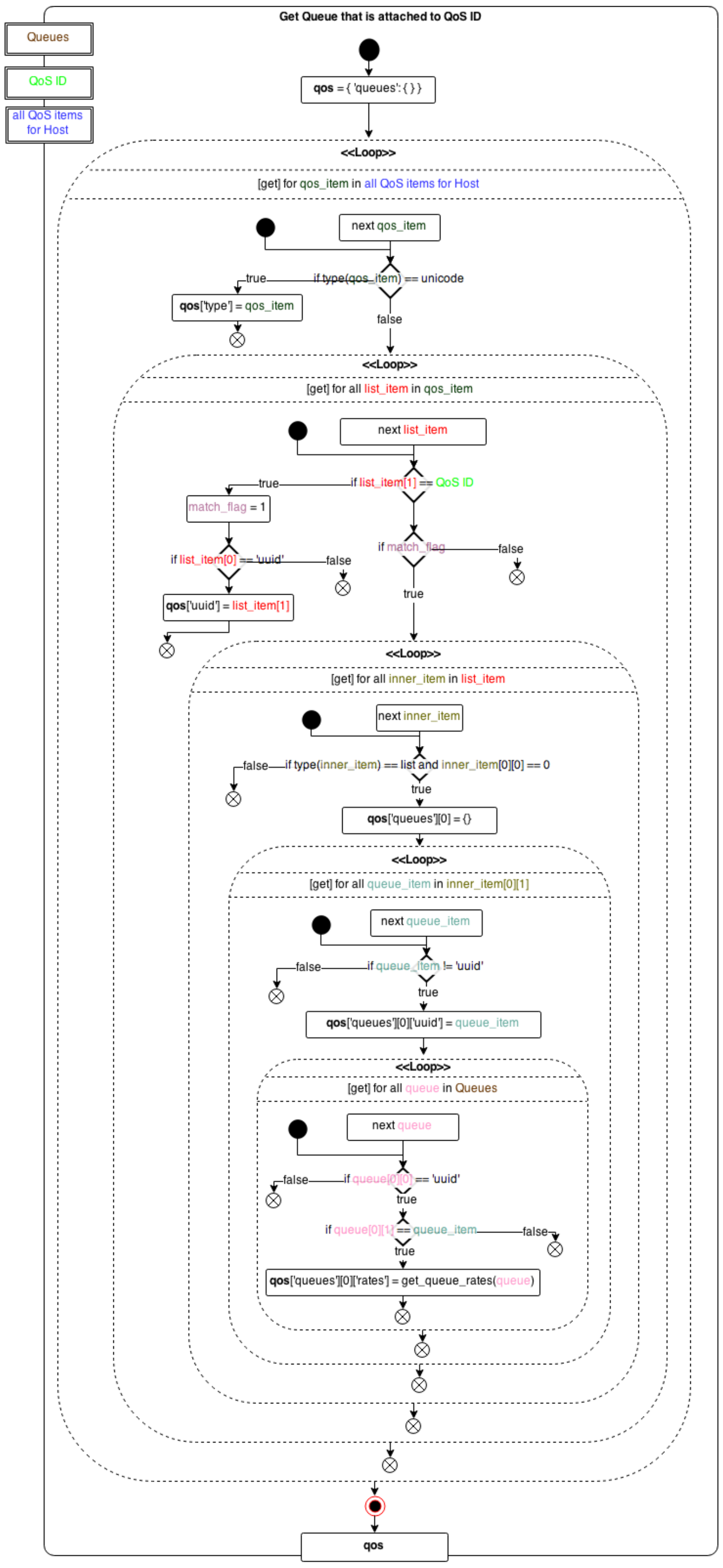

Get Queue that is attached to QoS ID
Queues
QoS ID
all QoS items for Host
qos = { 'queues': { } }
<<Loop>>
[get] for qos_item in all QoS items for Host
next qos_item
true
if type(qos_item) == unicode
qos['type'] = qos_item
false
<<Loop>>
[get] for all list_item in qos_item
next list_item
true
if list_item[1] == QoS ID
match_flag = 1
if match_flag
false
if list_item[0] == 'uuid'
false
true
qos['uuid'] = list_item[1]
<<Loop>>
[get] for all inner_item in list_item
next inner_item
false
if type(inner_item) == list and inner_item[0][0] == 0
true
qos['queues'][0] = {}
<<Loop>>
[get] for all queue_item in inner_item[0][1]
next queue_item
false
if queue_item != 'uuid'
true
qos['queues'][0]['uuid'] = queue_item
<<Loop>>
[get] for all queue in Queues
next queue
false
if queue[0][0] == 'uuid'
true
if queue[0][1] == queue_item
false
true
qos['queues'][0]['rates'] = get_queue_rates(queue)
qos

Figure 6.11: Get Queue rates for a QoS ID

6.3.2 Set QoS rates for all servers

In order to set the bandwidth rates for the servers this method first retrieves the list of all hosts that was mentioned in the previous figure and description. When the *Set QoS* method is called through a request to the API the method body from the HTTP POST is passed onto it. This dictionary contains all QoS rates that need to be set for each server.

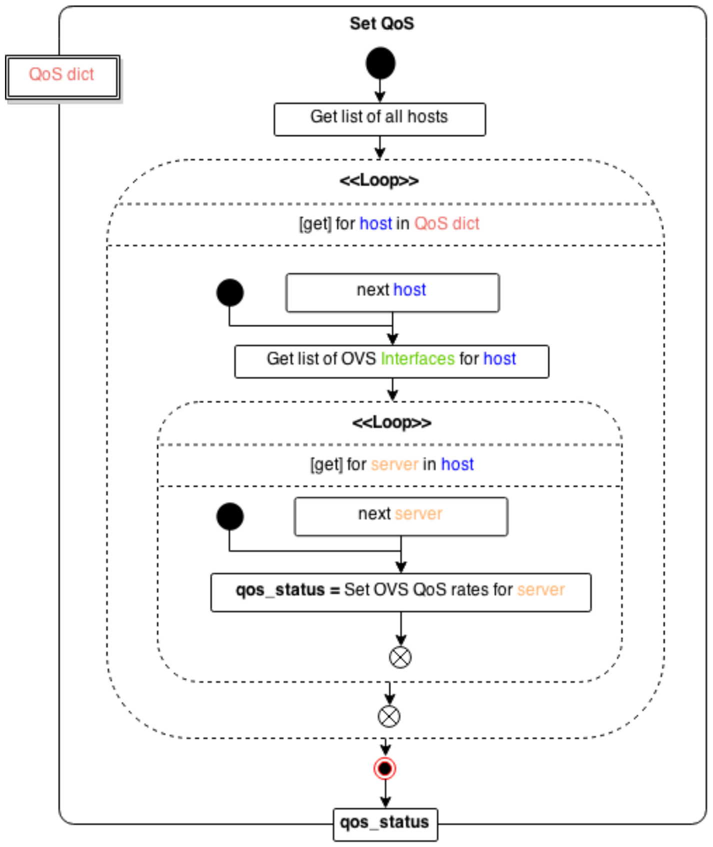


Figure 6.12: Agent: Set QoS rates for all servers

The method that is called from within the above figure for setting QoS for a single server performs the following activities:

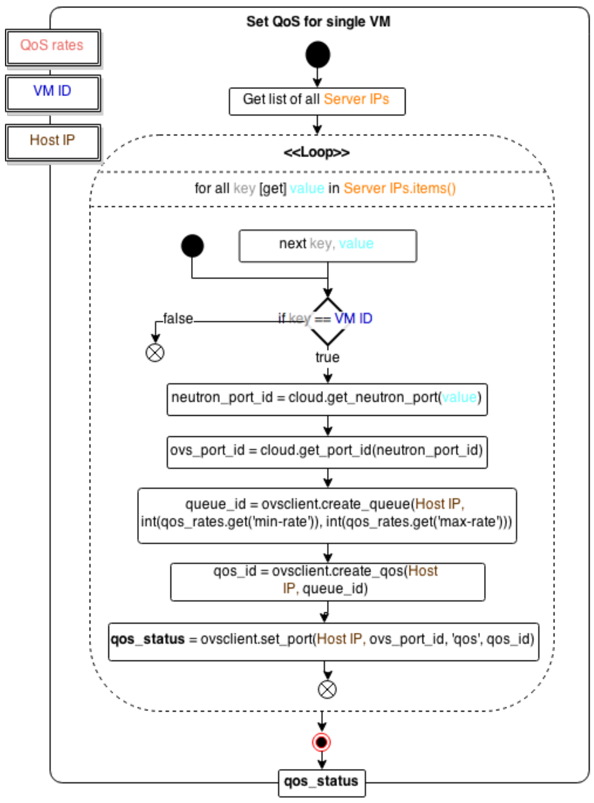

Set QoS for single VM
QoS rates
VM ID
Host IP
Get list of all Server IPs
<<Loop>>
for all key [get] value in Server IPs.items()
next key, value
false
if key == VM ID
true
neutron_port_id = cloud.get_neutron_port(value)
ovs_port_id = cloud.get_port_id(neutron_port_id)
queue_id = ovsclient.create_queue(Host IP, int(qos_rates.get('min-rate')), int(qos_rates.get('max-rate')))
qos_id = ovsclient.create_qos(Host IP, queue_id)
qos_status = ovsclient.set_port(Host IP, ovs_port_id, 'qos', qos_id)
qos_status

Figure 6.13: Server: Set QoS rates for a single server

6.3.3 API

The API was implemented using Bottle. It is a fast, simple and lightweight WSGI micro web-framework [27]. Therein three routes were defined:

```
# Welcome Screen
self._app.route('/', method="GET", callback=self._welcome)
# Host method
self._app.route('/hosts', method="GET", callback=self.
    _hosts_list)
# QoS method
self._app.route('/qoses', method=["POST", "OPTIONS"], callback=
    self._qoses_set)
```

The first route contains a welcome message and can be used to check if the WSGI app is currently running. The /hosts route calls the list_hosts() method in the Agent when a HTTP GET is received. When the /qoses route is called with the QoS parameters in its HTTP body it calls the set_qos() method in the Agent class.

The server uses the localhost IP address and is served and listens on port 8091. It is important that this IP address is whitelisted in case a firewall exists. In the case of using a Vagrant box the port needed to be forwarded to the local machine.

6.4 Tests

For testing the components of the Connectivity Manager Agent the code was synchronized to a local testbed with two virtual machines, one of which acted as the control node and the other just as a compute node.

These functional tests contain the following methods of the main components:

- Agent: instantiates the Agent class and tests listing the information about all hosts
- Keystone: tests retrieving the token and endpoint
- Neutron: lists all ports and shows the Neutron Port ID for a selected IP address
- Nova: retrieves all hosts and their servers
- OVS: tests the main functionalities that are used within this work: listing all interfaces, ports, queues, qos's and creating a new queue and qos that are then attached to a OVS port
- QoS: takes the parameters set in the *test_qos_config.json* file as an input and posts it to the API in order to try and set QoS on the defined server ports
- ReST: tests retrieving the list of all hosts through the API

The tests for the Connectivity Manager are part of the testing packages of the Elastic Media Manager, because that is the only way they are called. The *test_deploy.py* was used for checking if the integration works successfully. It uses a predefined topology from a local file. The following minimized configuration was used during the tests:

```
{
    "name":"local_nm_template_minified",
    "service_instances": [
        {
            "name":"Controller",
            "service_type":"Controller"
        },
        {
            "name":"Broker",
            "service_type":"Broker"
        }
    ]
}
```

To find out more about the parameters that each of the service_types corresponds to, please check the *Topology Definition* in the Evaluation section.

Chapter 7

Evaluation

7.1 Connectivity Manager Integration with NUBOMEDIA project

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 is an elastic Platform-as-a-Service (PaaS) cloud for interactive multimedia services [28]. Its architecture is based on media pipelines: chains of elements providing media capabilities such as encryption, transcoding, augmented reality or video content analysis. These chains allow building arbitrarily complex media processing for applications. As a unique feature, from the point of view of the pipelines, the NUBOMEDIA cloud infrastructure behaves as a single virtual super-computer encompassing all the available resources of the underlying physical network. A big part of currently developed applications within the platform make use of WebRTC, which can use TCP or UDP traffic. This is why both protocols need to be evaluated.

7.2 Network Performance Analysis

The following section contains information about the configuration used, followed by the different scenarios that were used to evaluate the effectiveness of the Connectivity Manager. All scenarios used the same topology and were deployed on a tenant without any other running servers or Heat stacks. The bandwidth tests were performed using the iperf tool.

7.2.1 Test-bed Configuration

The test-bed consists of two nodes with the following hardware characteristics:

Name	**Control node**	**Compute node**
Hostname	datacenter-4	dc4-comp
RAM	12 GB	8 GB
CPU	8-core Intel Core i7-4765T CPU @ 2.00GHz	4-core Intel Core i3-2120T CPU @ 2.60GHz
OS	Ubuntu 14.04.1 LTS	Ubuntu 14.04.1 LTS
Kernel version	GNU/Linux 3.13.0-32-generic	GNU/Linux 3.13.0-32-generic
Ethernet card	Intel Corporation Gigabit Ethernet Connection I217-LM	Intel Corporation 82579LM Gigabit Network Connection

Table 7.1: Hardware resources of testbed used for evaluation

The two nodes are connected to a Gigabit-Ethernet switch. The installation of OpenStack was performed using the devstack script as outlined in the Devstack section in the Implementation chapter.

7.2.2 Installation of Connectivity Manager Agent

A setup script exists in order to make it easier to get the CM Agent running. It builds and installs all the necessary Python packages in a virtual environment.This has the benefit that all packages are isolated from already existing Python installations and ensures that all packages are in the required version and don't interfere with the ones that are needed by OpenStack or other applications.

The first step is to clone the Git repository from the remote Git server. For the installation the cm-agent.sh script needs to be executed with the 'install' option.

```
$ ./cm-agent.sh
Usage: cm-agent.sh option
options:
  install    - install the server
  update     - updates the server
  start      - start the server
  uninstall - uninstall the server
  clean      - remove build files
```

The installation process includes setting up the virtual environment, installing all required Python packages and copying the configuration file to the /etc/nubomedia folder.

The configuration file needs to be customized, so it contains the correct IP address of the control node and OpenStack tenant credentials:

```
$ cat /etc/nubomedia/cm-agent.properties
os_username=admin
os_password=pass
os_auth_url=http://192.168.41.45:5000/v2.0
os_tenant=demo
```

Lastly the application can be started in a screen session using the following command:

```
$ venv/bin/python cm-agent/wsgi/application.py
```

The *venv/bin/python* command means that the program uses the Python interpreter that is located in the virtual environment.

7.2.3 Topology Definition

The topology that is used for the evaluation contains the following services instances:

data/json_file/topologies/topology_local.json:

```
{
    "name":"local_nm_template_minified",
    "service_instances": [
        {
            "name":"Controller",
            "service_type":"Controller"
        },
        {
            "name":"Broker",
            "service_type":"Broker"
        },
        {
            "name":"MediaServer",
            "service_type":"MediaServer"
        }
    ]
}
```

It can be deployed using a test script which performs a HTTP POST to the EMM API at the */topologies* path.

The service types are further defined in another JSON file, which includes their configuration, networks and other parameters that are needed for provisioning. As one example the Media Server service is given below:

data/json_file/services/MediaService.json:

```
{
    "service_type": "MediaServer",
    "version":"1",
    "image": "trusty-iperf",
    "flavor": "m1.mini",
    "key":"nubomedia",
    "configuration": {
    },
    "size": {
        "min": 1,
        "def": 3,
        "max": 5
    },
    "networks": [
        {
            "name":"Network-1",
            "private_net":"8048fd67-70a6-447d-a779-8a86f9eeb35d",
            "private_subnet": "0df3f54c-d1af-4b82-8376-18baa11d0e98",
            "public_net": "62024eab-23c2-4a81-a996-87af4d252282",
            "security_groups": [
                "SecurityGroup-MediaServer"
            ]
        }
    ],
    "requirements": [
        {
            "name":"$BROKER_IP",
            "parameter":"private_ip",
            "source":"Broker",
            "obj_name": "Network-1"
        }
    ]
}
```

7.2.4 Scenario 1: Without Instance Placement Engine & QoS enabled

In this first scenario the deployment without any influence of the Connectivity Manager is shown. Here Nova randomly decides about the placement of the servers. In this case 4 servers were placed on the control node and one server on the compute node.

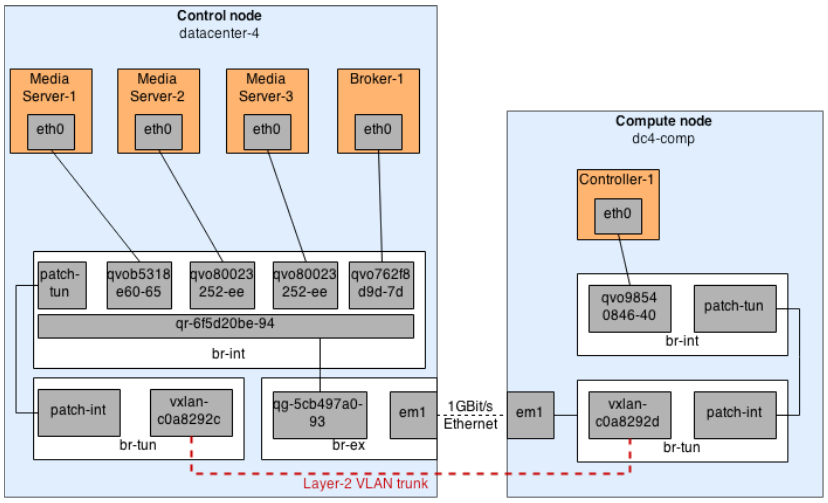


Figure 7.1: Scenario 1: Placement of servers without Connectivity Manager

TCP traffic

For testing the bandwidth the MediaServer-2 was used as a TCP server using iperf. All other servers connected to it in client mode sending and retrieving TCP packets in a timeframe of 10 seconds. The following graph shows the bandwidth usage of the servers.

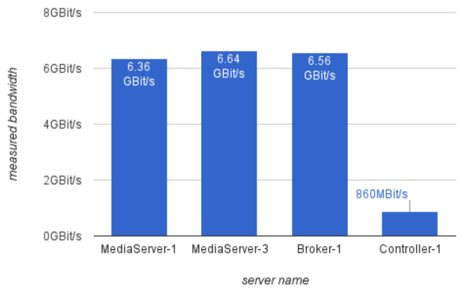


Figure 7.2: Scenario 1: TCP bandwidth comparison

As is clearly visible, the network performance of the server on the separate node performs much worse, which is also due to the fact that there is only a Gigabit-Ethernet connection between the nodes.

UDP traffic

In order to test the bandwidth with UDP traffic, two servers were started as a UDP server and each of them received one client connection. The selected bandwidth limit that the packets were streamed at is 800 MBit/s.

Server name	Measured bandwidth	Ratio lost/total datagrams	Datagrams out of order	Jitter
MediaServer-1	780 MBit/s	3.4%	18	0.011 ms
Controller-1	779 MBit/s	2%	49689	0.078 ms

Table 7.2: Scenario 1: Network characteristics with generated UDP traffic

The comparison of the network statistics shows that while the measured bandwidth is the same across all connections (even for the server that is located on the separate compute node), the datagrams that are received by the server out of order is tremendously larger. Multiple tests showed also that the jitter is always of a greater value, despite only having a minimal difference.

7.2.5 Scenario 2: With Instance Placement Engine enabled, but without QoS

In this next scenario, the Instance Placement Engine was enabled and therefore the availability zone in the topology was successfully set to a single host.

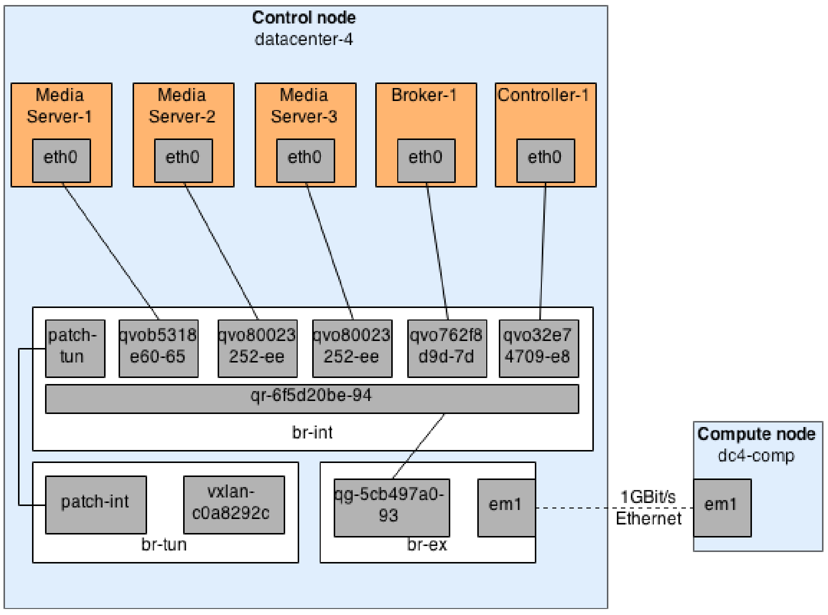


Figure 7.3: Scenario 2: Placement of servers with Connectivity Manager

TCP traffic

The following graph shows that the available bandwidth using TCP packets is now evenly distributed for all servers. However the traffic of the MediaServers should be prioritized, which is why QoS is needed to further improve the connectivity according to the requirements.

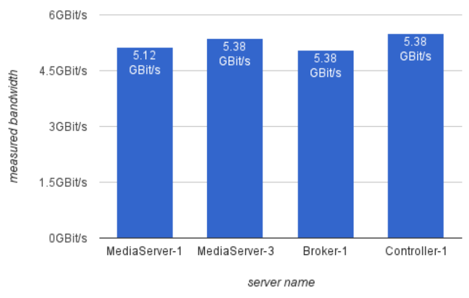


Figure 7.4: Scenario 2: TCP bandwidth comparison

UDP traffic

The traffic generation was configured in the same way as in Scenario 1, except that now all servers are deployed on the same compute node.

Server name	Measured bandwidth	Ratio lost/total datagrams	Datagrams out of order	Jitter
MediaServer-1	791 MBit/s	2.3%	9	0.018 ms
Controller-1	789 MBit/s	2.7%	13	0.013 ms

Table 7.3: Scenario 2: Network characteristics with generated UDP traffic

With the servers all running on the same compute node, the datagrams that were received out of order could be minimized.

7.2.6 Scenario 3: Instance Placement Engine and QoS Manager enabled

The servers are again placed on a single host and the Quality of Service configuration that was previously set in the configuration is applied. All media servers have a guaranteed bandwidth rate of 100 MBit/s and a maximum rate of 10 GBit/s, while all servers of other instance types have a rate between 100 MBit/s and 1 GBit/s.

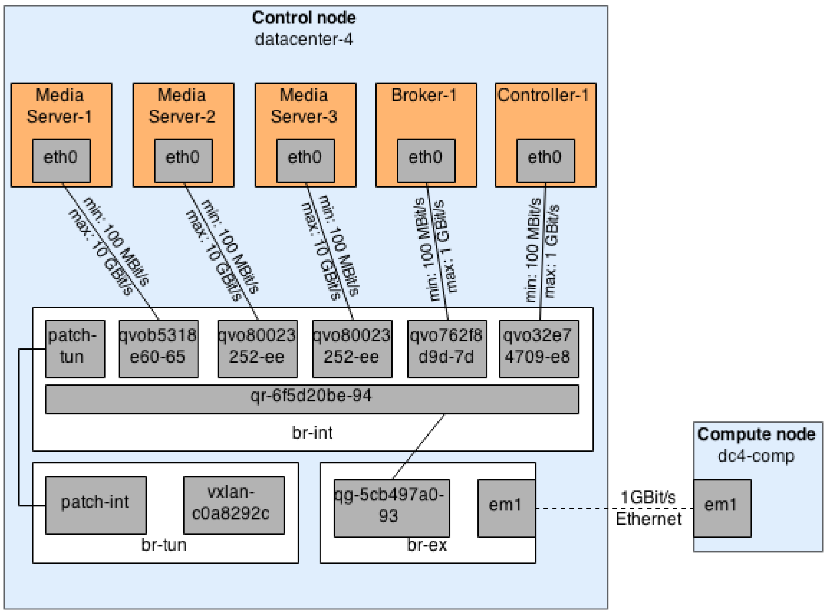


Figure 7.5: Scenario 3: Placement of servers using the Connectivity Manager with QoS enabled

The topology with it's configured QoS Queues is also visible by performing a HTTP GET to the Connectivity Manager Agent at the /hosts path (excerpt):

```
{
   "dc4-comp":{
      "cpu_total":4,
      "ip":"192.168.41.44",
      "servers":{ },
      "instances":0,
      "ram_total":7788,
      "ram_used":0,
      "cpu_used":0,
      "id":2
   },
   "datacenter-4":{
      "cpu_total":8,
      "ip":"192.168.41.45",
      "servers":{
         "9a2b6c59-72fd-4576-8a51-27c506b77980":{
            "ip":"10.0.0.15",
            "ovs_port_id":"qvob5318e60-65",
```

```json
            "qos":{
               "queues":{
                  "0":{
                     "rates":{
                        "max-rate":"10000000000",
                        "min-rate":"100000000"
                     },
                     "uuid":"e9183562-fe04-4341-99e0-8a3b73659101"
                  }
               },
               "uuid":"b3a0edc0-60bc-4c06-ac5a-6219a2add194"
            },
            "name":"MediaServer-1",
            "neutron_port":"b5318e60-6534-4f80-ab05-8d4747f1022a"
         },
         { "..."
         },
         "dedd78db-2be4-4655-b08a-8cac40087823":{
            "ip":"10.0.0.16",
            "ovs_port_id":"qvo762f8d9d-7d",
            "qos":{
               "queues":{
                  "0":{
                     "rates":{
                        "max-rate":"1000000000",
                        "min-rate":"100000000"
                     },
                     "uuid":"8b0f7fbb-e511-46ee-b20d-eb7b0f0a5c18"
                  }
               },
               "uuid":"edcfcbb5-1bfc-4797-a101-845ceb55bc1c"
            },
            "name":"Broker-1",
            "neutron_port":"762f8d9d-7d7f-45fb-836b-eaa3ebaad71b"
         }
      },
      "instances":5,
      "ram_total":11813,
      "ram_used":3072,
      "cpu_used":5,
      "id":1
   }
}
```

TCP traffic

For this last scenario two iperf-servers and three iperf-clients were configured. The *'Controller-1'* server received packets from the *'MediaServer-2'* and the *'MediaServer-3'* from *'MediaServer-1'* and *'Broker-1'*.

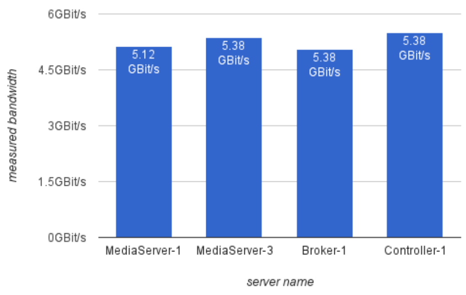


Figure 7.6: Scenario 3: TCP bandwidth comparison

The first two bars show that the egress port on the Open vSwitch that the *'MediaServer-3'* is connected to is limited to a combined bandwidth of 10 GBit/s. This is why the two connections share the available bandwidth and are nearly equal. It has to be remarked that the network performance is the average of a 10-second bandwidth test.

In the other QoS class the *'MediaServer-2'* which is connecting to *'Controller-1'* uses almost the full bandwidth of the available 1 GBit/s.

UDP traffic

The setup of iperf for generating the traffic is equal to Scenario 2.

Server name	Measured bandwidth	Ratio lost/total datagrams	Datagrams out of order	Jitter
MediaServer-1	761 MBit/s	5.9%	7	0.015 ms
Controller-1	782 MBit/s	3.5%	41	0.102 ms

Table 7.4: Scenario 3: Network characteristics with generated UDP traffic

The MediaServer-1 was connecting to the MediaServer-2 and the Controller-1 opened up a connection to Broker-1 in this case.

With Quality of Service enabled at different rates, the difference in network statistics for UDP traffic is not that great. The only difference that is visible is that the Gold traffic of the MediaServer-1 is prioritized to the one of the Controller-1 which is part of the Wholesale QoS class.

7.3 Conclusion

The three test scenarios show an improvement of the network performance, which was achieved by placing the servers on a single host. Additionally, with Quality of Service enabled, the network bandwidth for TCP traffic is shaped according to the requirements. It has been shown that the connection between Media Servers is prioritized to other service instance types. The Service-Level-Agreement of the Gold and Wholesale classes are fulfilled.

Chapter 8

Conclusion

8.1 Summary

With the use of SDN and OpenFlow, a framework for creating programmable networks already exists. However the integration of an SDN Controller can further extend the capabilities of SDN in OpenStack. By default, the network bandwidth supplied to different servers is on a best-effort basis and topologies that are deployed using the orchestration service Heat cannot be placed on specific hosts. The work described in this thesis explains how these limitations can be overcome. With the help of the Open vSwitch configuration tool it is possible to get information about the current state of the switched network and enable QoS using Queues. The required configuration and information can now be accessed through an API. The Elastic Media Manager is able to access this information and make decisions about where the topology has the best-performing network connectivity. The bandwidth rates according to their Service-Level Agreements can be easily updated and extended. The evaluation gives a step-by-step explanation of the improvements in bandwidth rates with the use of the Connectivity Manager in relation to the requirements.

8.2 Problems Encountered

One significant problem during the development process was getting a stable test-bed running with the correct configuration. Devstack helped significantly in this regard, because aside from adjusting the configuration file, not many manual corrections needed to be made. However some of the features that are in the 'stable' version of OpenStack Juno still need further bug-fixing and cannot currently be used in a production environment. During the testing of various state-of-the-art solutions, a significant amount of time was spent on testing different versions, which was unsuccessful. Due to the significant time taken, comparisons to other solutions for enabling Quality of Service could not be evaluated.

8.3 Future Work

For future development, the algorithm for selecting the best-performing host could take more dynamic factors into account. One possibility is to check the network speed of all the host's network interfaces. Furthermore, it would be a good approach to implement the features of the Connectivity Manager Agent as a Neutron extension and make use of the existing Neutron API. The implementation of QoS could be based on the existing blueprint and be shared with the OpenStack community to achieve an integration into the Neutron repository.

Additional, one further step that could be evaluated for Quality of Service is to perform Flow modifications, as opposed to the currently used procedure of attaching the Queue to the OVS Port and having a rate-limit on the egress traffic. With this approach, the ingress traffic for a specific Port could be filtered with enqueuing and then set as the Flow action.

Bibliography

[1] N. Feamster, J. Rexford, E. Zegura: „The Road to SDN“, ACM Queue, Volume 11, Issue 12, December 30, 2013.

[2] Open Networking Foundation: „Software-Defined Networking: The New Norm For Networks “, ONF White Paper, April 13, 2012.

[3] Open Networking Foundation: „Software-Defined Networking (SDN) Definition “, `https://www.opennetworking.org/sdn-resources/sdn-definition`

[4] S. Natarajan, SDN Hub: „OpenFlow version 1.3 tutorial “, `http://sdnhub.org/tutorials/openflow-1-3/`

[5] Open Networking Foundation: „OpenFlow Switch Specification “, Version 1.3.0, June 25, 2012.

[6] Open vSwitch: „Open vSwitch - Frequently Asked Questions “, `http://git.openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob_plain;f=FAQ;hb=HEAD`

[7] Open vSwitch: „Open vSwitch - Readme “, `https://github.com/openvswitch/ovs/blob/master/README.md`

[8] J. Gross, VMware: „Programmable Networking with Open vSwitch “, LinuxCon, September, 2013. `https://events.linuxfoundation.org/sites/events/files/slides/OVS-LinuxCon%202013.pdf`

[9] J. Pettit, E. Lopez, VMware: „OpenStack: OVS Deep Dive “, 07 November, 2013.

[10] Open vSwitch: „Open vSwitch Manual“, `http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf`

[11] B. Hubert: „tc - Linux man page“, `http://linux.die.net/man/8/tc`

[12] M. Devera, D. Cohen: „HTB Linux queuing discipline manual“, `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`

[13] J. Vehent: „Journey to the Center of the Linux Kernel: Traffic Control, Shaping and QoS“, `http://wiki.linuxwall.info/doku.php/en:ressources:dossiers:networking:traffic_control`

[14] A. Pech, Arista Networks: „Running OpenStack over a VXLAN Fabric“, `https://www.openstack.org/assets/presentation-media/OpenStackOverVxlan.pdf`

[16] OpenStack Foundation: „OpenStack Operations Guide“, February 01, 2015, http://docs.openstack.org/openstack-ops/openstack-ops-manual.pdf

[16] OpenStack Foundation: „OpenStack Configuration Reference“, Compute-Scheduler section, February 01, 2015, http://docs.openstack.org/juno/config-reference/content/section_compute-scheduler.html

[17] OpenStack Foundation: „OpenStack Compute“, 2015, https://www.openstack.org/software/openstack-compute/

[18] OpenStack Foundation: „OpenStack Installation Guide“, January 29, 2015, http://docs.openstack.org/juno/install-guide/install/apt/content/

[19] K. Pepple: „OpenStack Nova Architecture“, April 22, 2011, http://ken.pepple.info/openstack/2011/04/22/openstack-nova-architecture/

[20] OpenStack Foundation: „OpenStack Training Guide“, Chapter 7, January 31, 2015, http://docs.openstack.org/training-guides/content/operator-network-node.html

[21] K. Mestery, D. Meyer, Linux Foundation: „Introduction to OpenDaylight and Hydrogen“, 2014, https://www.openstack.org/assets/presentation-media/osodlatl.pdf

[22] C. Dixon, OpenDaylight: „OVSDB Integration“, October 1, 2014, https://github.com/opendaylight/docs/blob/master/manuals/developers-guide/src/main/asciidoc/ovsdb.adoc

[23] D. Pemberton, A. Linton, S. Russell, University of Oregon: „RYU OpenFlow Controller“, 2014, https://nsrc.org/workshops/2014/nznog-sdn/raw-attachment/wiki/WikiStart/Ryu.pdf

[24] Yamamoto, OpenStack: „Neutron OFAgent - Comparison with OVS“, December 5, 2014, https://wiki.openstack.org/w/index.php?title=Neutron/OFAgent/ComparisonWithOVS&oldid=69704

[25] S. Collins, OpenStack: „Neutron QoS“, October 25, 2013, https://wiki.openstack.org/w/index.php?title=Neutron/QoS&oldid=34054

[26] OpenStack Foundation: „OpenStack Admin User Guide“, February 01, 2015, http://docs.openstack.org/user-guide-admin/user-guide-admin.pdf

[27] M. Hellkamp: „Bottle documentation“, January 18, 2015, http://bottlepy.org/docs/dev/bottle-docs.pdf

[28] L. Lopéz, NUBOMEDIA: „What's NUBOMEDIA?“, http://nubomedia.eu/page/whats-nubomedia

Appendix A

List of acronyms

ACL → Access Control List

API → Application Programming Interface

AZ → Availability Zone

CBQ → Class Based Queueing

CM → Connectivity Manager

CPU → Central Processing Unit

DHCP → Dynamic Host Configuration Protocol

DiffServ → Differentiated Services

DSCP → Differentiated Services Code Point

EMM → Elastic Media Manager

GBit/s → Gigabit per second

GRE → Generic Routing Encapsulation

HTB → Hierarchy Token Bucket

HTTP → Hypertext Transfer Protocol

IaaS → Infrastructure-as-a-Service

IP → Internet Protocol

JSON → JavaScript Object Notation

KVM → Kernel-based Virtual Machine

MBit/s → Megabit per second

ML2 → Modular Layer 2

MPLS → Multiprotocol Label Switching

MQ → Message Queue

MTU → Maximum Transmission Unit

NAT → Network Address Translation

OF → Open Flow

ONF → Open Networking Foundation

OVS → Open vSwitch

OVSDB → Open vSwitch Database

PaaS → Platform-as-a-Service

qdisc → Queueing Discipline

QoS → Quality of Service

RAM → Random-Access Memory

ReST → Representational State Transfer

RPC → Remote Procedure Call

SDN → Software-Defined Networking

SLA → Service-Level Agreement

SSH → Secure Shell

TC → Traffic Control

TCP → Transmission Control Protocol

TTL → Time to live

vCPU → Virtual Central Processing Unit

VLAN → Virtual Local Area Network

VXLAN → Virtual Extensible Local Area Network

VM → Virtual Machine

WSGI → Web Server Gateway Interface