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 08. Februar 2015

(Unterschrift)

Manuel Bergler

Contents

List of Figures

List of Tables

List of Algorithms

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 approaching its limits. In order to resolve this, Software Defined Networking (SDN) is going to be extending and replacing 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 cloud-based services, with not only competitive pricing but also high-availability and fast network access, is taking over the traditional self-hosted data centers. The ease of administration and the deployment of new Virtual Machines (VMs) on the fly make it possible to create a Topology of Computers with no effort.

Network services have different requirements, depending on the type of data and their importance. The classification of network traffic can be done through Quality of Service (QoS). A new approach has to be made to enable the use of QoS in virtualized cloud infrastructures like OpenStack, to achieve controlled traffic from the deployment of Virtual Machines on.

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 classic network architectures. With scalable computing and storage the common-place tree-structured network infrastructure with Ethernet switches 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 the underlying infrastructure. Therefore a shift of today's networks to more flexibility, programmability and scalability is going to take place.

1.3 Objective

The primary objective of this work is the development of a network orchestrator which is able to apply Quality of Service to the network interfaces of Virtual Machines. These Virtual Machines are deployed with OpenStack Nova and connected to an OpenVSwitch, which uses OpenFlow. Another task of the Connectivity Manager is to select which OpenStack hypervisor new VMs should be running on, which takes different runtime parameters into account. The CM should be able to be applied in environments with scalable hypervisors and VMs.

1.4 Scope

The scope of this work includes a Connectivity Manager which will have a Connectivity Manager Agent running on the cloud controller within the OpenStack infrastructure, to provide access to the hypervisors of OpenStack Nova. These two components have to be implemented and integrated with the existing OpenVSwitches. As a reference for a cloud infrastructure, multimedia communications like the Nubomedia project will be used. The deployment of this cloud is then tested on different performance characteristics like network bandwidth, latency, 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 into different classes, depending on the service that are running on it.

Currently there are a number of solutions for managing network connectivity between VMs. A comparison and their current limitations follows in the next section. The chosen approach is to extend the existing network control and management services with Quality of Service (QoS) capabilities. In support of the thesis the Connectivity Manager will be implemented and the differences in bandwidth usage will be shown in one use-case.

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 make up OpenStack 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 and an architectural overview of the Connectivity Manager. Moreover design aspects are introduced and illustrated according 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 a particular use-case.

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

Chapter 2

Fundamentals and related work

2.1 Software-Defined Networking

The origin of Software-Defined Networking (SDN) began already in 1995, however the first use cases were only developed in 2001 and the promotion of SDN only began with the foundation of the non-profit industry consortium Open Networking Foundation (ONF) in 2011. The ONF is dedicated to push and adapt open standards like the OpenFlow into the industry. In this following section a brief overview of the SDN architecture and concepts, including the OpenFlow protocol is given.

2.1.1 Motivation

Today's internet is part of the modern society, be it for private users, enterprises or 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 networks one comes to see the following limitations:

- **Inability to scale**: With the expansion of data centers, networks must grow too. Configuring and managing these additional network devices comes at a high administrative effort. With the virtualization of data centers network traffic patterns becomes more and more dynamic and unpredictable. With multi-tenancy a further complication is introduced, because different end-users and services need different network performance and might require traffic steering. Such scaling and network management cannot be done with a manual configuration of the underlying infrastructure.
- **Complexity**: In the past decades new networking protocols have been adapted by the industry. To add or move any device, multiple existing switches, routes, firewalls must be touched in order to manage protocol-based mechanisms on a device-level. With the virtualization of servers the amount of interfaces that need network connectivity and the distribution of applications over a number of virtual machines (VMs) are another demand that the current fairly static networks cannot dynamically adapt to.

- **Inconsistent policies**: For IT to apply a network- or data center-wide policy a lot of devices and mechanisms may need to be reconfigured. Virtual Machines are created and rebuilt within no time, but if for example access or security needs to be updated, the benefits of this dynamic are subverted(?).
- **Vendor dependence**: Standards are needed to match the requirements of the markets with the capabilities of networks and enable network operators to customize the network to specific environments.

Traditionally decisions about traffic flowing 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. In figure .. the main components that are depicted 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.

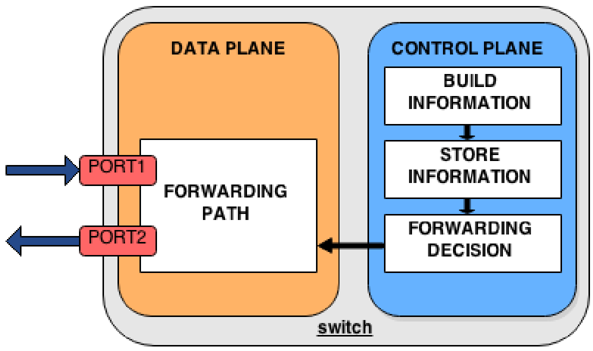

Figure 2.1: "Classical" switch components

A switch consists of multiple ports for incoming and outgoing data. Internal forwarding tables classify the packets and forward them to one or many specific ports. It does so by collecting MAC addresses and storing their corresponding port in specific 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 where paths to neighbour 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 coming sections.

2.1.2 Software-Defined Networking concept

SDN represents a new dynamic, manageable, cost-effective and adaptable architecture that is built to serve the dynamic infrastructures that are needed as a backbone for today's 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 protocol 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. In figure .. this separation is shown.

- **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 global view of the network, by communicating with the switching elements through the so called South-bound 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

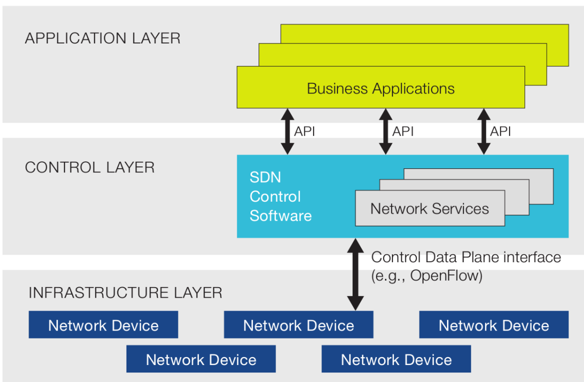

Figure 2.2: Software-Defined Network architecture

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.

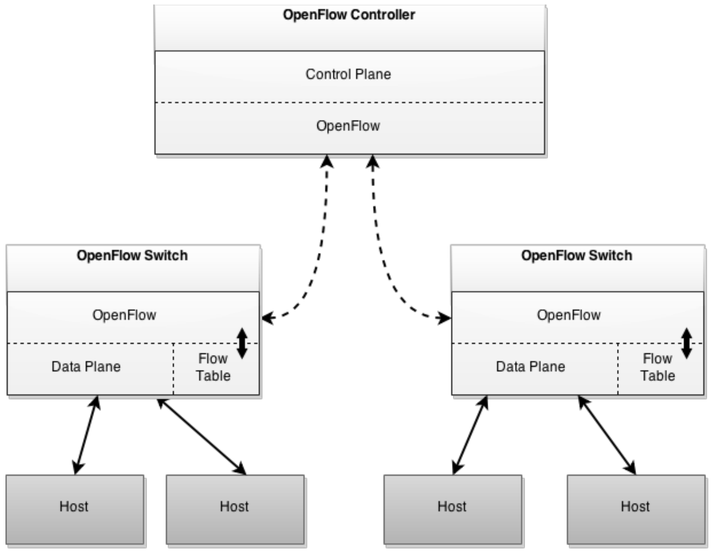

Figure 2.3: OpenFlow Network Architecture

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 that defines its minimum capabilities in order to support OpenFlow.

Most of the OpenFlow-enabled switches and controllers currently still only support the OpenFlow version 1.0 (released in December 2009). The newest version at this date is 1.4, however this explanation of OpenFlow will be focussed on version 1.3 since that is the most recent specification which is supported by OpenVSwitch. The main features added since version 1.0 are among others support for VLANs, IPv6, tunnelling and per-flow traffic meters.

Generally the switches are backwards-compatible down 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 there is also a set of optional features.

2.1.4.1 OpenFlow Controller

The OpenFlow controller is separated from the switch and has two interfaces. The northbound interface is an API to the application layer for implementing 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 of OpenFlow-compliant switches:

- **OpenFlow-only:** in these switches all packets are processed by the OpenFlow pipeline and they 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 this type.

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

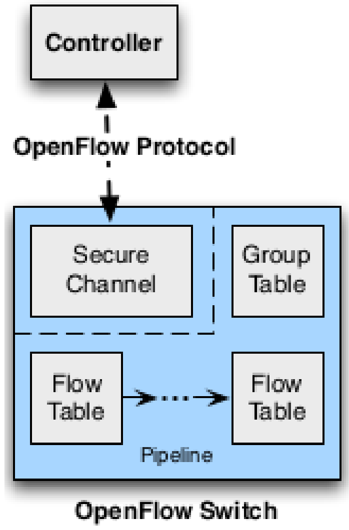

Figure 2.4: OpenFlow Switch components

Through the connection using the OpenFlow protocol, it is possible for the controller 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. (see OF Tables section)

The matching of flow entries begins at the first flow table, however it may continue to additional flow tables, and it uses the first matching entry from each table and performs the instruction that is linked with that specific entry. 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 modify pipeline processing. Packet forwarding, packet modification and group table processing are the possible actions. With pipeline processing packets can be permitted to be sent to other tables for further processing and 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. 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 defines specifies how packets correspond with each of the flow tables.

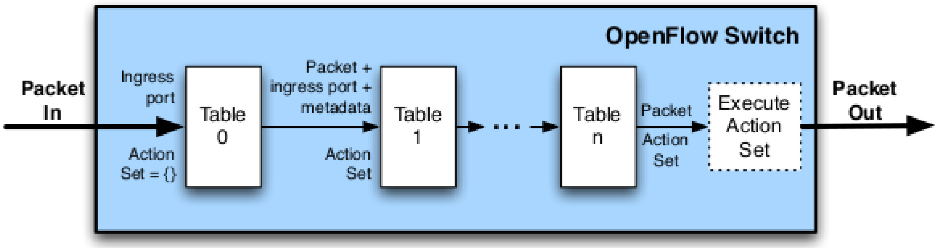

Figure 2.5: OpenFlow pipeline processing

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 which consist of the following fields:

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

- **match fields:** ingress port, packet headers and optionally metadata
- **priority:** set 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 a opaque data value chosen by the controller

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

Packet Matching

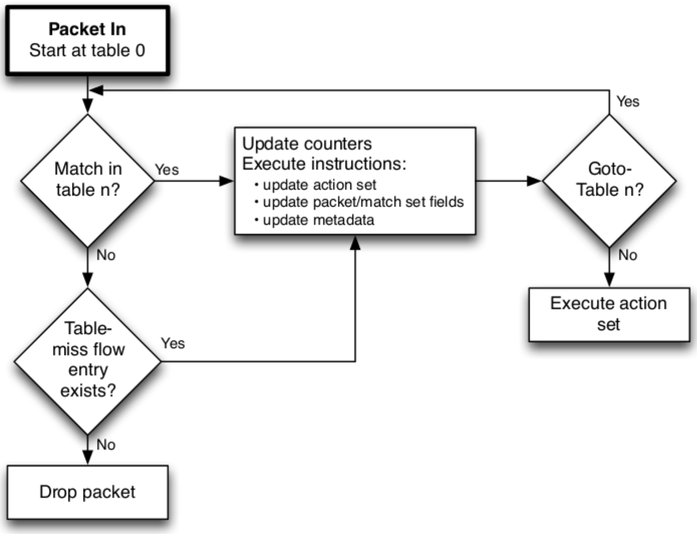

Figure 2.6: Packet flow through an OpenFlow switch

On a packet's arrival at the Flow Table, the packet match fields are extracted and used for the table lookup. They include different packet header fields. Additionally matches can be made against the ingress port and metadata fields. If the values in the packet match fields equate only the flow entry with the highest priority is selected. The associated counters are updated and the instruction set 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 in 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

- **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

- **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 band have optional arguments

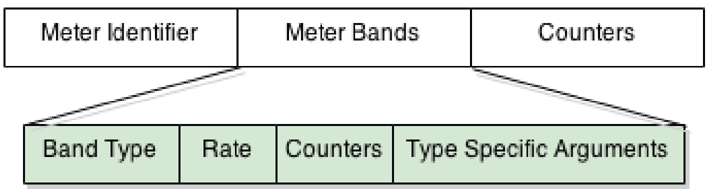

Figure 2.7: OpenFlow QoS as a meter

<u>Instructions</u>

Instructions are executed when a packet matches the flow entry. 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*:** direct 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. 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:

- **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 attached to this port is used for scheduling and forwarding the packet when the packet is forwarded to a port using the output action. This forwarding behaviour allows to enable basic QoS support.
- **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 open source software switch that is used in virtualized server environments. It is able to forward traffic traffic between Virtual Machines (VMs) 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 run on kernel 2.6.32 and newer.

OVS supports the following features:

- 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:

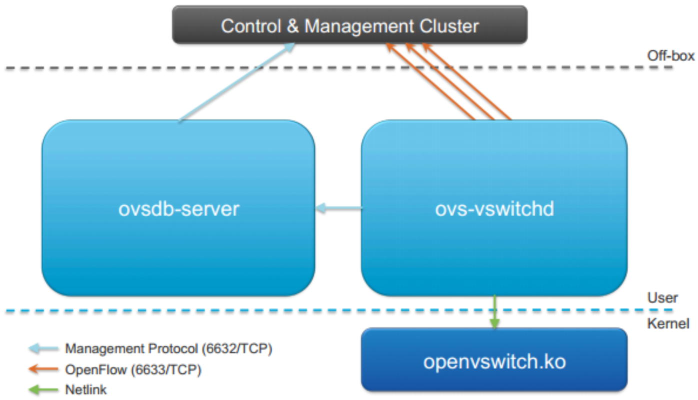

Figure 2.8: Architecture of Open vSwitch: divided into kernelspace and userspace

ovs-vswitchd, a daemon that implements the switch, along with a companion Linux kernel module for flow-based switching. ovsdb-server, a lightweight database server that ovs-vswitchd queries to obtain its configuration.

The daemon which implements the switch is *ovs-vswitchd* and it is shipped with an additional Linux kernel module for flow-based switching that it communicates with using the netlink protocol. 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 it completely 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. The database is divided into multiple different tables with different purposes, with the ones related to this project outlined below:

- **Open_vSwitch**: Open vSwitch configuration
- **Port**: Port configuration
- **Interface**: A physical network device within a Port
- **QoS**: Quality of Service configuration
- **Queue**: QoS output queue

- **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.

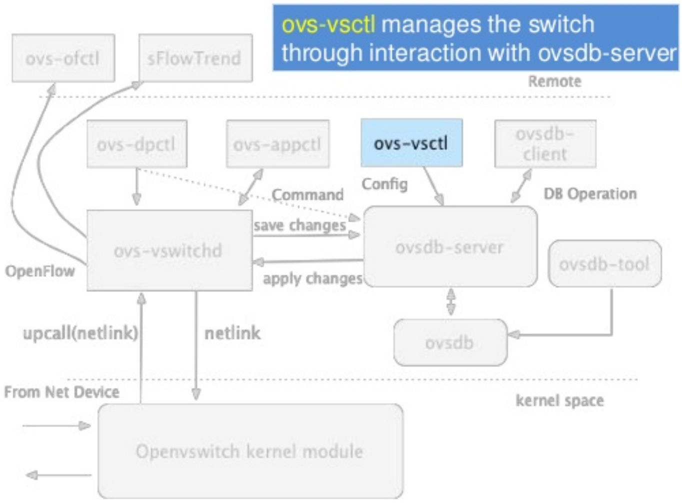

Figure 2.9: Visualization of the interaction of the ovs-vsctl tool

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

- ovs-vsctl add-br %bridge%
- ovs-vsctl list-br
- ovs-vsctl add-port %bridge% %port%
- ovs-vsctl list-ports %bridge%
- ovs-vsctl get-manager %bridge%
- ovs-vsctl get-controller %bridge%
- ovs-vsctl list %table%

2.1.5.4 QoS

With Open vSwitch QoS can be configured for ports or the so-called virtual network interfaces that virtual machines get when they are connected to the internal bridge of the switch. The minimal and maximal rate-limits are defined in bytes and applied to a queue, which operates as egress shaping. The QoS port policies make use of the 'tc' implementation that is included in the Linux kernel.

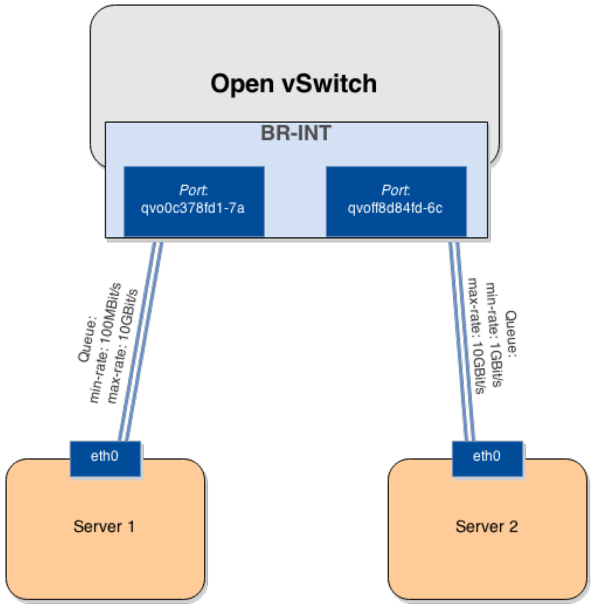

Figure 2.10: QoS Queues attached to a Port in OVS

Traffic control (tc) uses 'queueing discipline' (qdisc) for configuring the network interface. 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 Queueing (CBQ) qdisc in Linux. It helps to control the use of the outbound bandwidth on a given link.

2.1.5.5 GRE

Generic Routing Encapsulation (GRE) is used in OpenStack to tunnel the traffic between multiple nodes. It provides a private and secure path by encapsulating data packets.

2.2 Cloud computing infrastructures

2.2.1 OpenStack

The OpenStack project was founded by Rackpace Cloud and NASA, however currently more than 200 companies are contributing. With OpenStack one is able to design, deploy and maintain a private or public cloud. It is a flexible, scalable and open-source approach that combines multiple technologies into one Infrastructure-as-a-Service (IaaS). 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 come with 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.

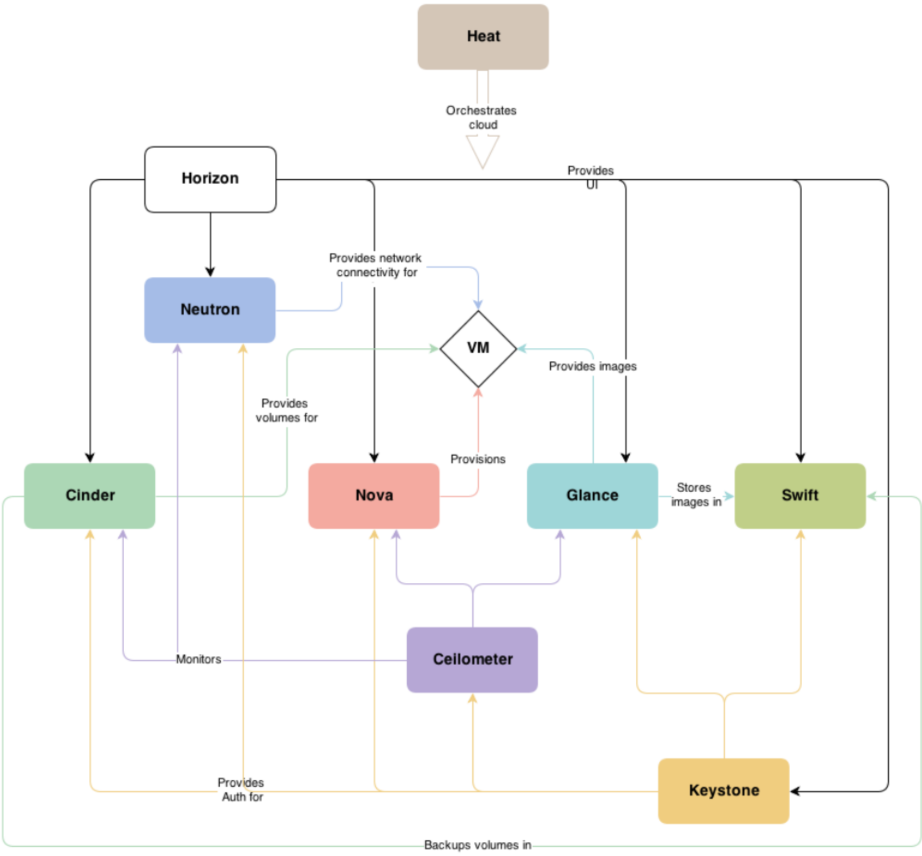

Figure 2.11: Interaction among OpenStack services

2.2.2 OpenStack Compute (Nova)

Nova is used to host and manage cloud computing systems. 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).

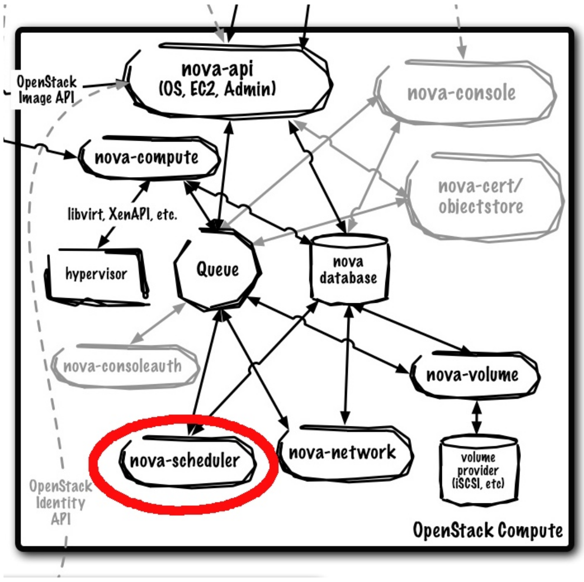

Figure 2.12: OpenStack Compute service

Compute Node segregation

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

- **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.

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 a sub-component of Nova called Nova-network. This service had it's limitations, because it was closely coupled with networking abstractions and there were no APIs 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 IPs to give VMs external network access.

Neutron exposes an extensible set of APIs for creating and managing those. Neutron consists of the following elements:

- **neutron-server:** Provides the logic for SDN and does not contain any SDN 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 ML2 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 RPC 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, however the plugin agent is running on each hypervisor.

2.2.4.2 Modular Layer 2

The ML2 plugin is a framework that allows the simultaneous usage of multiple layer 2 networking technologies.

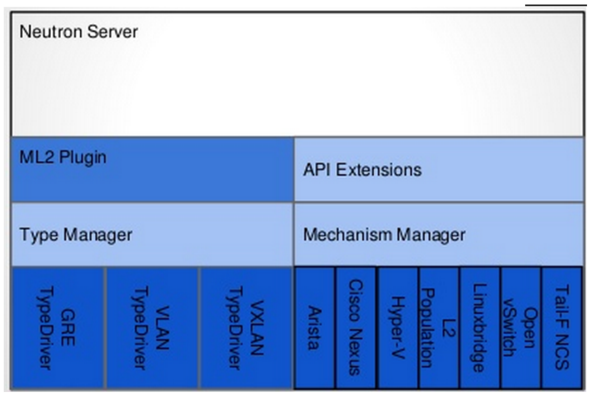

Figure 2.13: Neutron modular framework, including ML2 drivers

The plugin interfaces with 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 specifies the mechanism for accessing these networks, 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 using Devstack and is also the most commonly deployed agent.

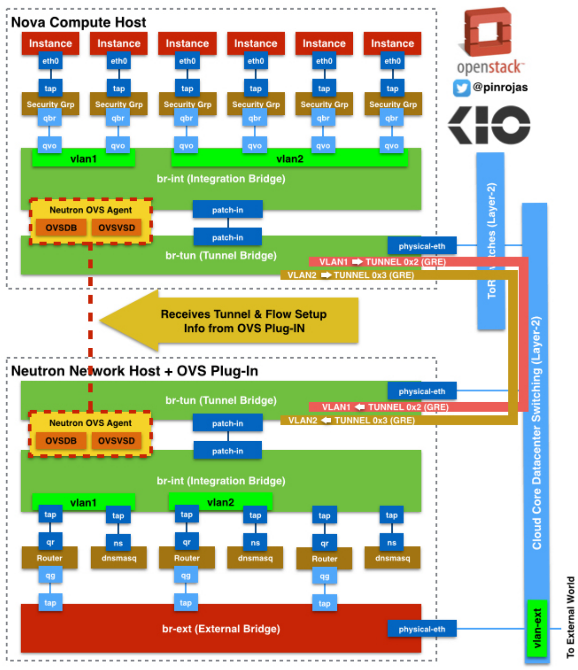

Figure 2.14: GRE tunneling between Controller Node and Compute Node

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

The workflow for Neutron, from starting to booting VMs is as follows:

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

Neutron - Nova interaction

1. Request: Create VM connected to network X (API)
2. Create VM (RPC: Nova API to Nova conductor)
3. Nova schedules VM
4. Create VM (RPC: Nova conductor to Nova compute)
5. Create Port (API: Nova compute to Neutron service)
6. Create tap device
7. Notify L2 agent (RPC)
8. get_device_details (RPC: L2 agent to Neutron service)
9. Configure local VLAN, OVS flows
10. Send port_up notification (RPC: L2 agent to Neutron service)
11. Send port_up notification (API: Neutron service to Nova)
12. port_up (RPC: Nova service to Nova compute)
13. Nova compute boots VM

2.3 Conclusion

Chapter 3

Requirements

3.1 Functional requirements

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

3.1.1 SLA Enforcement

One of the key objectives of the Connectivity Manager is to grant different Service Level Agreements to the links between Virtual Machines. The agreement is set as Quality of Service 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.

3.1.2 Optimal Virtual Machine Placement

The placement of Virtual Machines makes a difference in terms of connectivity and resource utilization. VMs that run on the same Compute Node have a better connectivity then ones that need to communicate over wire. The fact that the networking is virtualized besides the virtualized computing environment means that a more utilized Compute Node will also have less resources available for switching and routing. A part of the motivation for this requirement can be found in the Evaluation section.

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 for services like upscaling the amount of instances after 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 to do with the operation of a system and not with it's behavior. Thus the Connectivity Manager should also fit the following characteristics.

3.2.1 Scalability

Today's data-centers can grow in 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. 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 it is made available 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 Neutron with additional SDN features have been tested. They were selected based on the requirements given in section 3.

4.2 OpenDaylight SDN controller

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

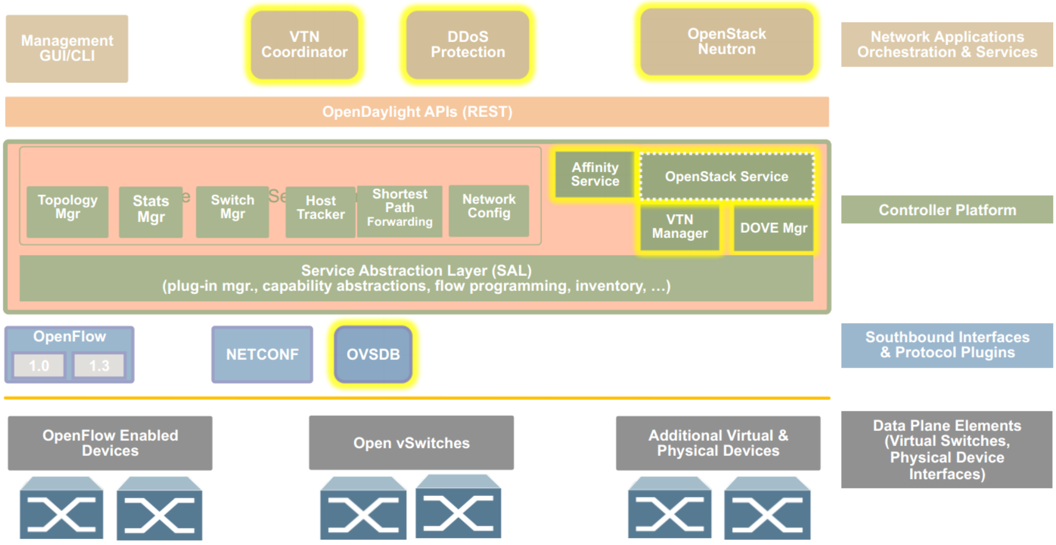

Figure 4.1: Architecture of OpenDaylight Virtualization edition

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 switch ports & OpenFlow Queues). In the local testbed we were able 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. 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. It includes a set of apps which build the base of the SDN controller like L2 switch, ReST interface, topology viewer and tunnel modules. Ryu also includes an app that allows to set QoS rules through a ReST interface which uses a OVSDB interaction library to apply those. The QoS rules can be either applied to a specific Queue within a VLAN or a Switch port. It supports DSCP tagging and setting the min-rate and max-rate of an interface.

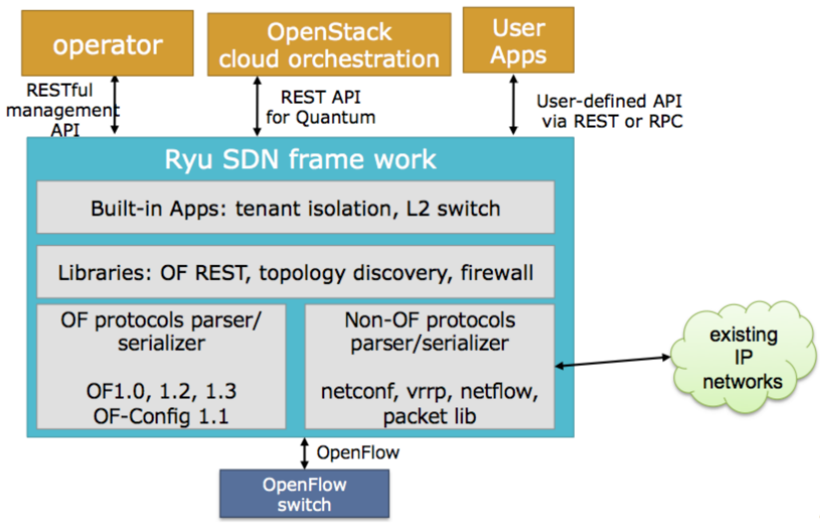

Figure 4.2: Ryu architecture

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

4.4 OpenStack Neutron - QoS Extension

A Neutron extension has been partially implemented for OpenStack IceHouse which includes an API for setting and retrieving QoS on a per-tenant and per-port basis.

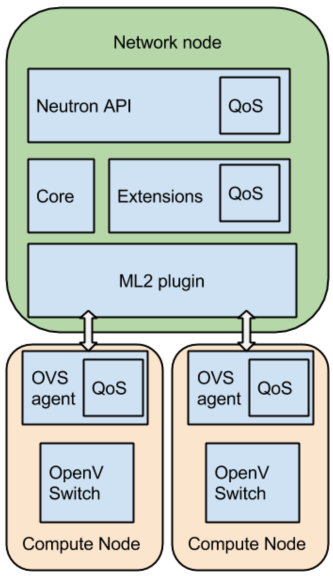

Figure 4.3: Neutron QoS Extension architecture

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 whiteboard it was deferred to Juno, but it's not included in the current release and no active development is stated in the code / documentation platforms of the OpenStack community.

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

4.6 Conclusion

The listed problems further strengthen the motivation for the implementation of the Connectivity Manager. A feature comparison and analysis is given.

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)

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.

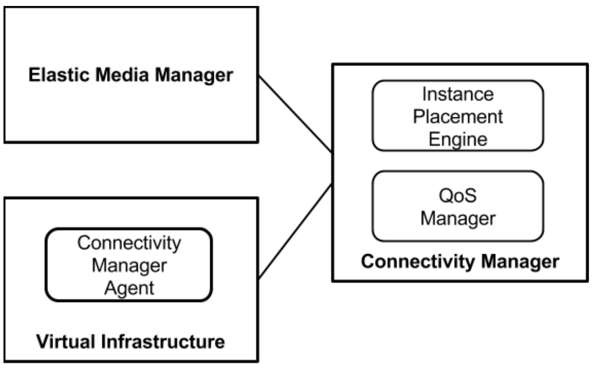

Figure 5.1: High-level architecture of the Connectivity Manager

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.

5.2 Connection between Manager & Agent

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

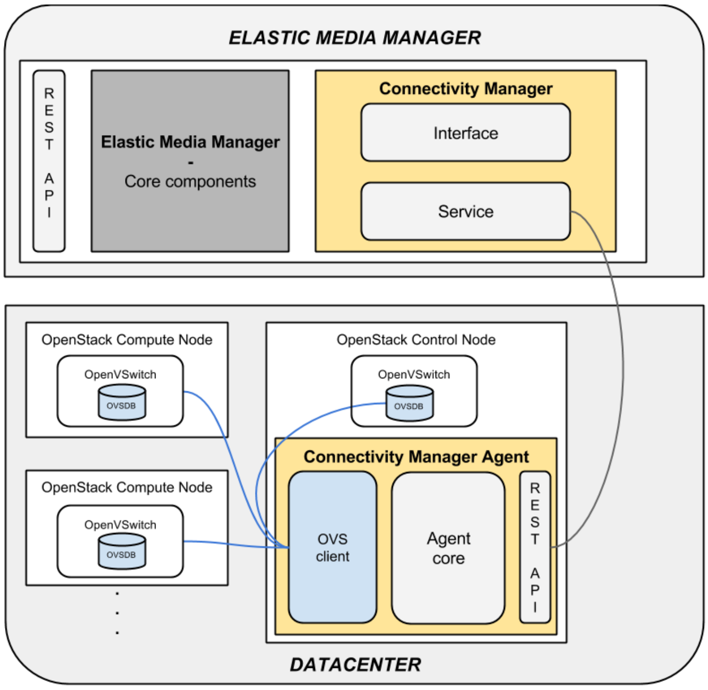

Figure 5.2: Minimized architecture of the Connectivity Manager and its integrations

This design was chosen first of all because the Connectivity Manager is integrated in the EMM, which is required to be placed anywhere outside of the data center. Second of all the Connectivity Manager Agent needs 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.

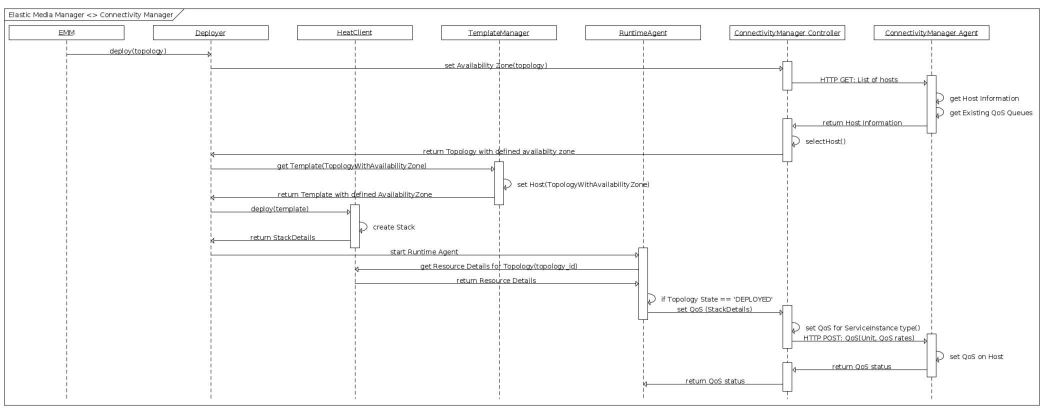

Figure 5.3: Workflow: Deployment of stack & Assignment of QoS policies

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 cloud. 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 given 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 got deployed by the Heat Client a 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 then to enable it on the according ports of the instances within the Open vSwitch.

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 according 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 remains identical.

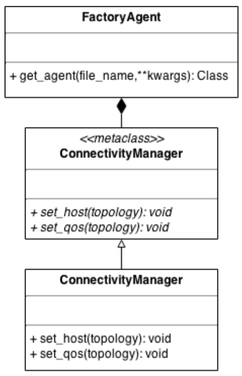

Figure 5.4: Class diagram: Metaclass (Interface) and implementation (Service) instantiated by Factory Agent

The Connectivity Manager interface and service only contain the methods that are later called by 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.

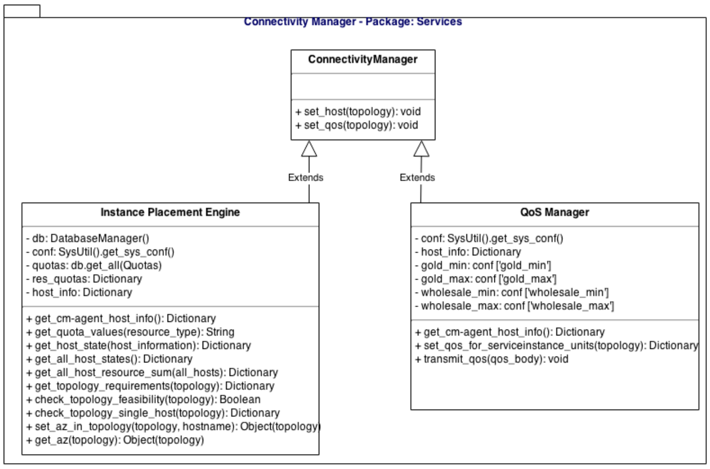

Figure 5.5: Class diagram: Connectivity Manager service and its helper classes

The topology that is created by the EMM and contains the required resources for the stack are visible in the next diagram.

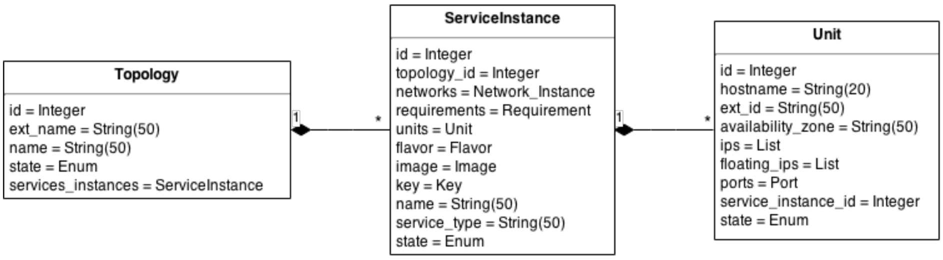

Figure 5.6: Deployment: Topology object from EMM

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 which networks the underlying instances need to be connected to, the flavor wherein the resources (e.g. amount of RAM & CPU) are specified, the image that contains the operating system and additional software packages and the key that is needed to establish a console connection via SSH.

Each Service Instance can contain multiple Units (instances). 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.

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.

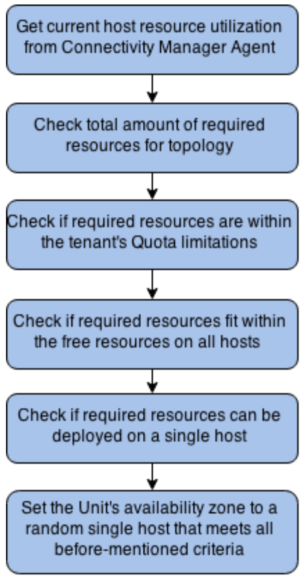

Figure 5.7: Instance Placement activities

5.3.2 QoS Manager

The QoS is deployed on the Units port. This design was chosen, because it needs to be differentiated between different services. The bandwidth rates are required to be changeable and therefore can't be hard-coded. By default the following rates and classes will be set:

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

All Units that are part of the 'Media Server' service instance will be associated with the Gold class and all units of other types in the topology will be 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 resources status and make changes to their configuration.

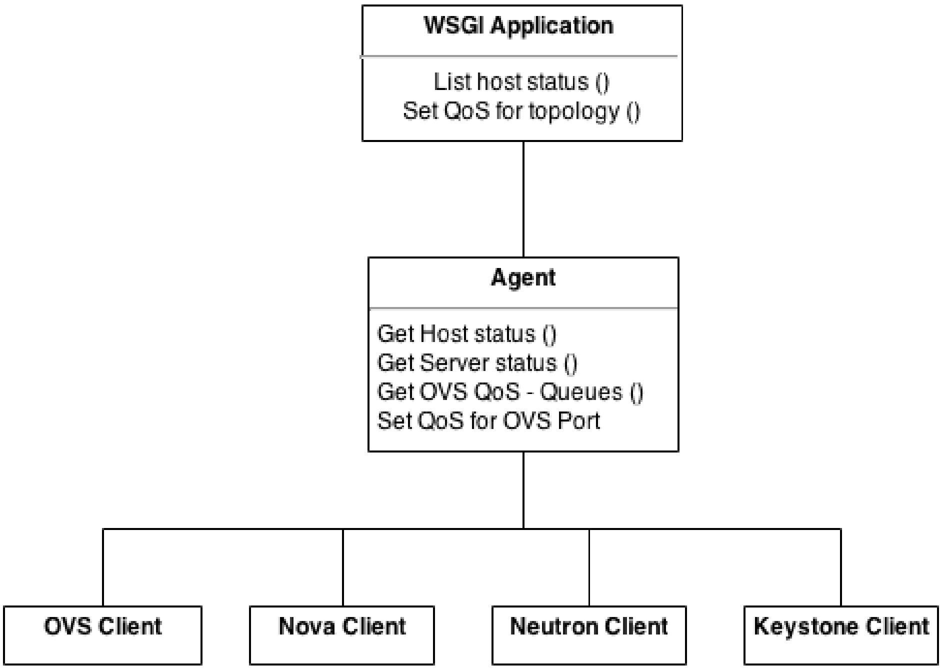

Figure 5.8: Design of Connectivity Manager Agent

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

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. Its Client can retrieve a list of all ports for the servers within one 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.

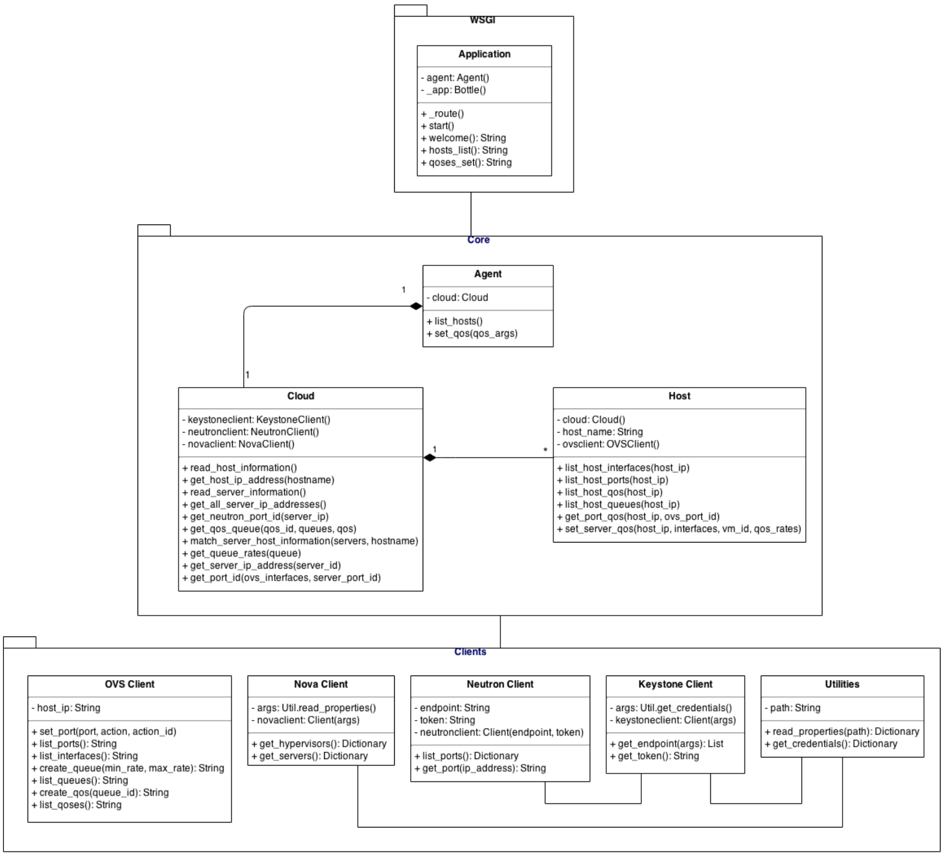

Figure 5.9: Class diagram: Connectivity Manager Agent - All packages

5.4.1 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.

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 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.

6.1.0.1 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.

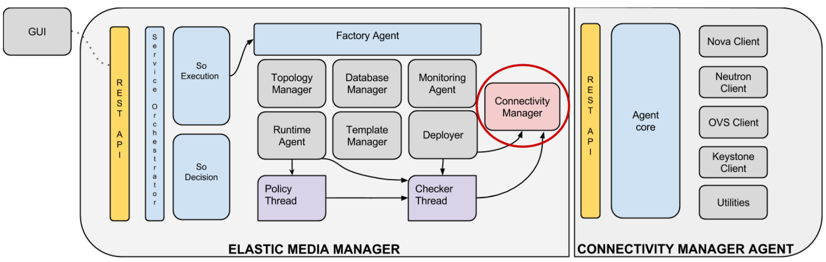

Figure 6.1: Implementation focus

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.

6.1.0.2 Local OpenStack test environment

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

6.2.0.3 Selection of best-matching Hypervisor

Step 1: Retrieve current host utilization from CM Agent

In order to decide on the placement the Connectivity Manager first of all 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.

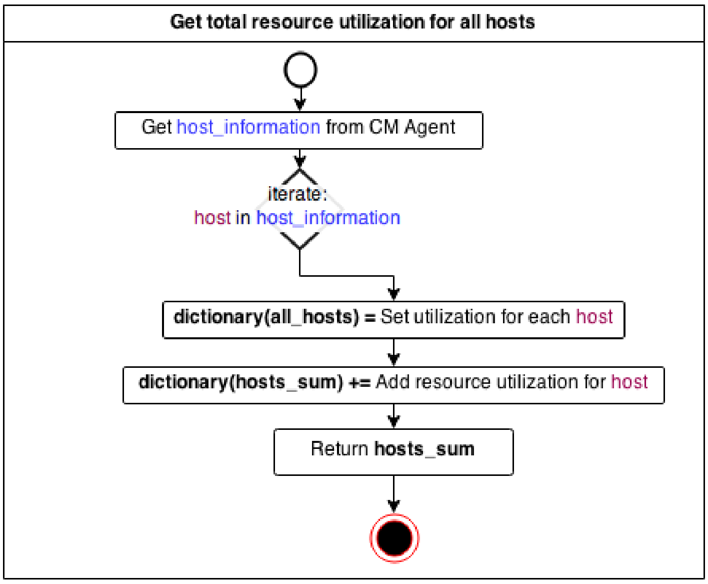

Figure 6.2: Check resource utilization of hosts

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.

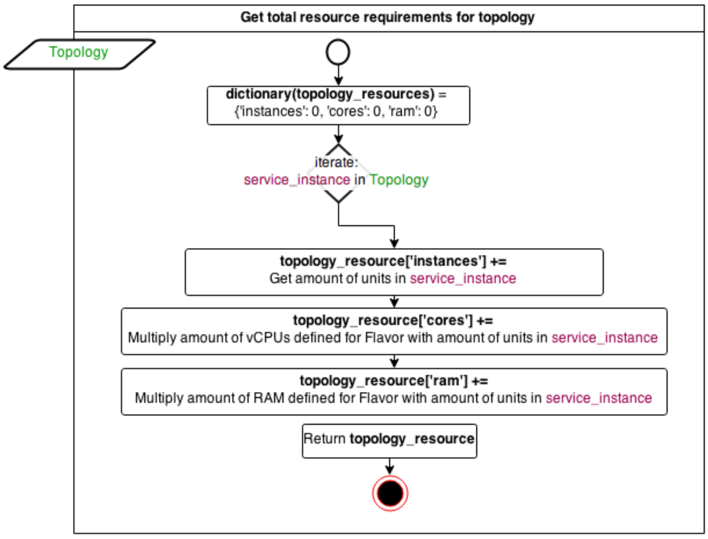

Figure 6.3: Get total amount of required resources for topology

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

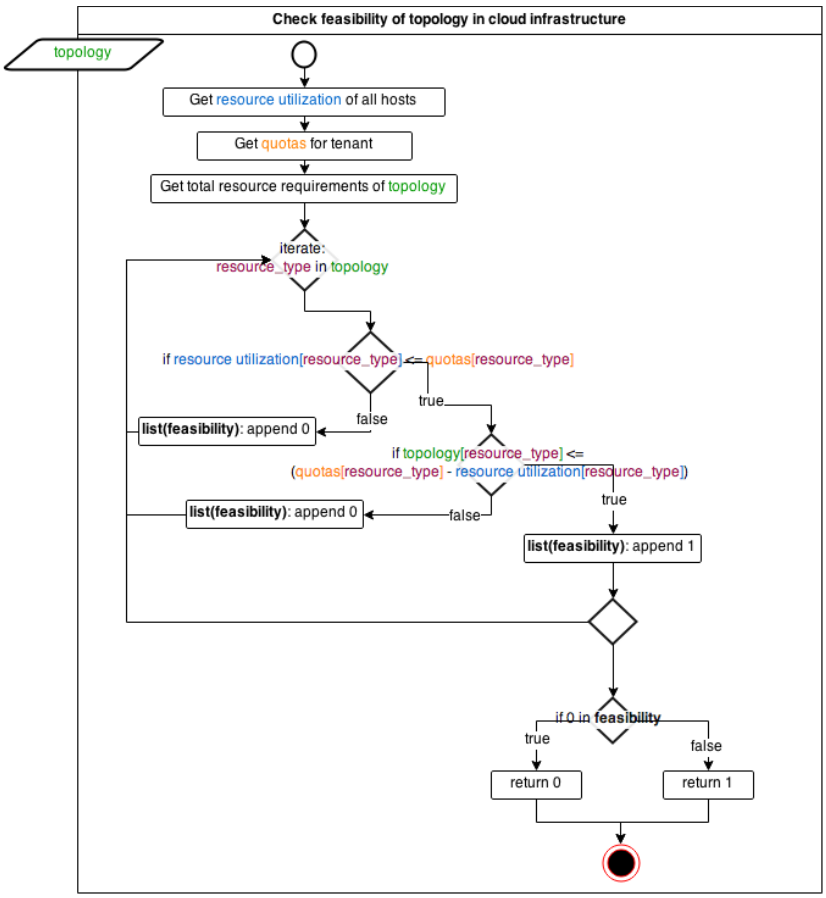

Figure 6.4: Deployment feasibility check

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

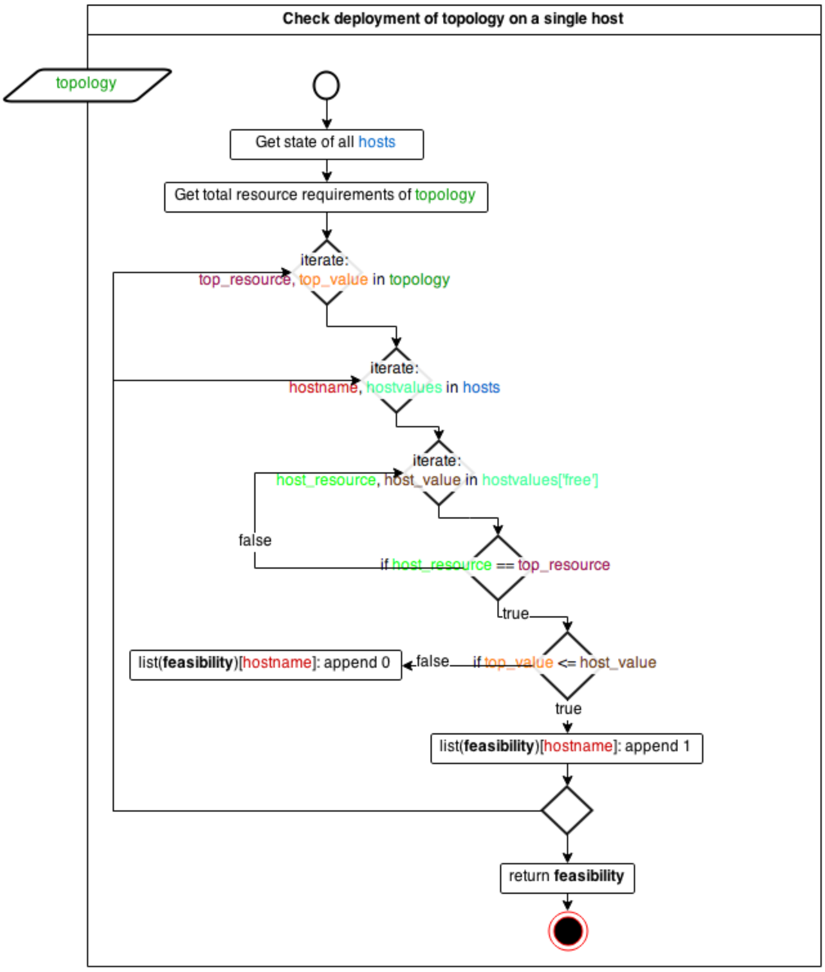

Figure 6.5: Single host deployment check

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

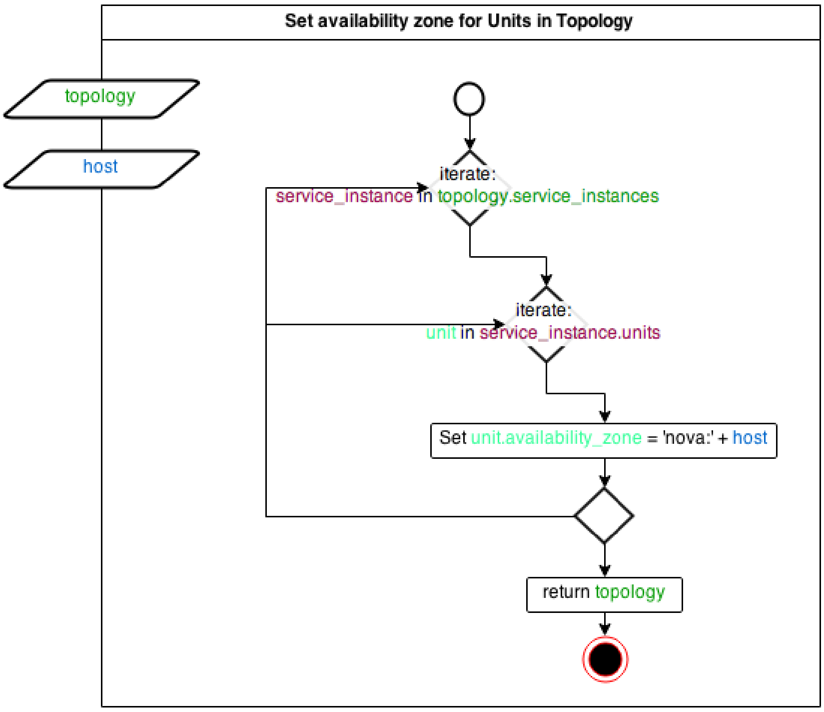

Figure 6.6: Set AZ per Unit

6.2.0.4 Enabling QoS for VM

.. ToDO: Mention configuration file!!

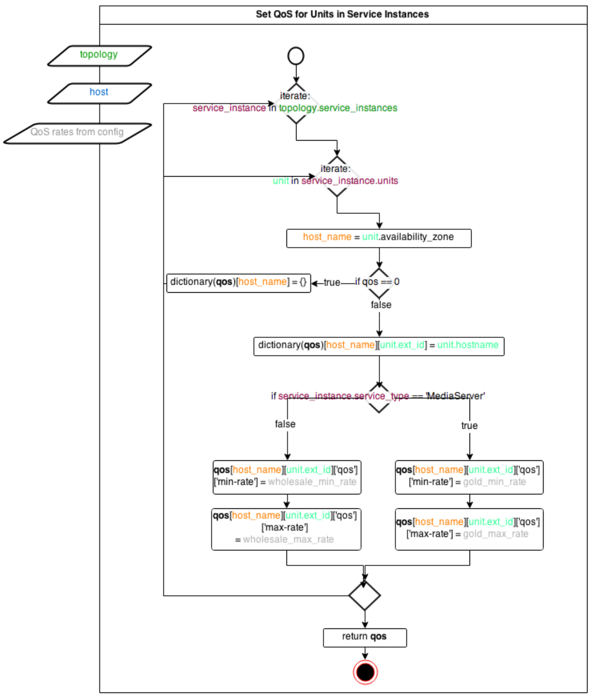

Figure 6.7: Method for setting QoS for all Units

6.3 Connectivity Manager Agent components and operations

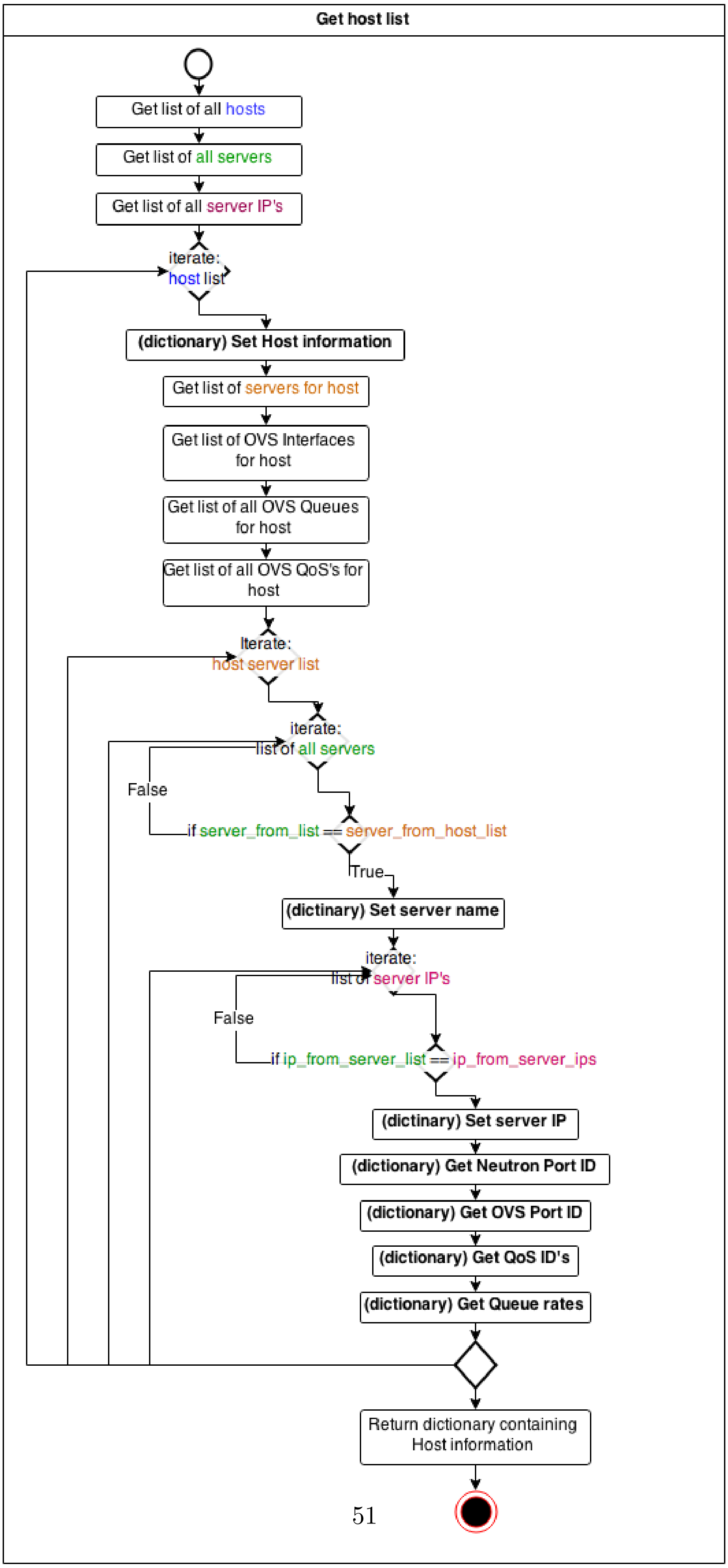

Figure 6.8: Activity diagram: Get list of hosts

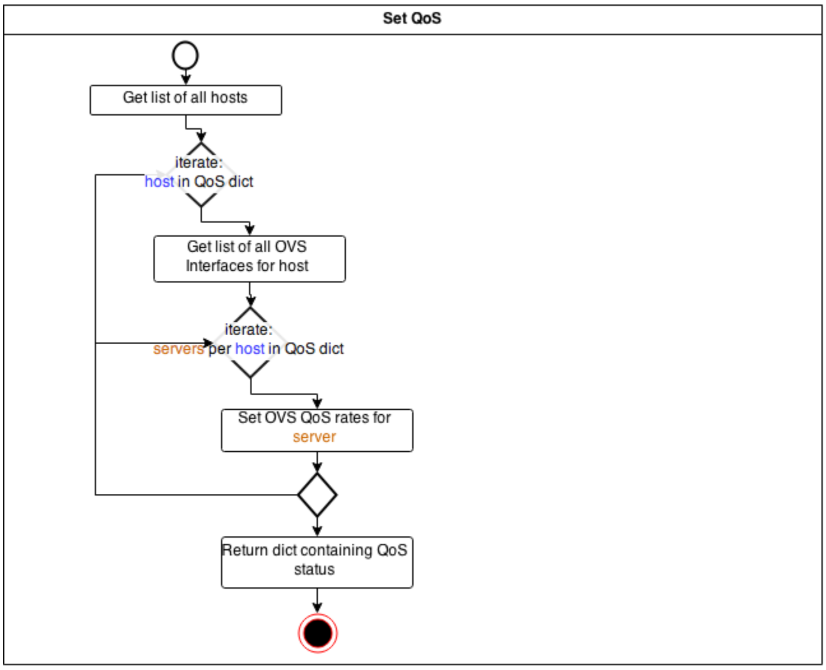

Figure 6.9: Activity diagram: Set QoS rates for all servers

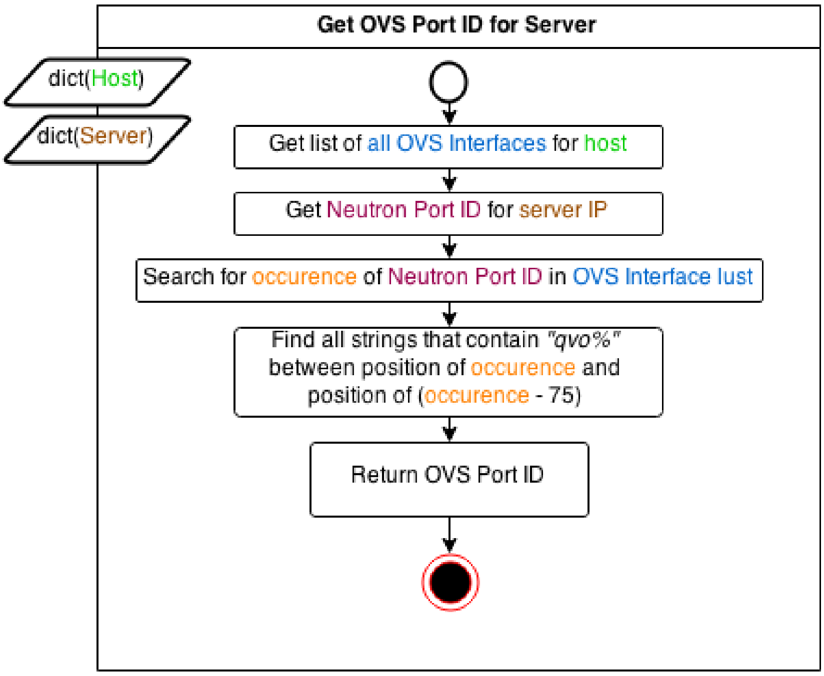

Figure 6.10: Activity diagram: Get OVS Port ID for server

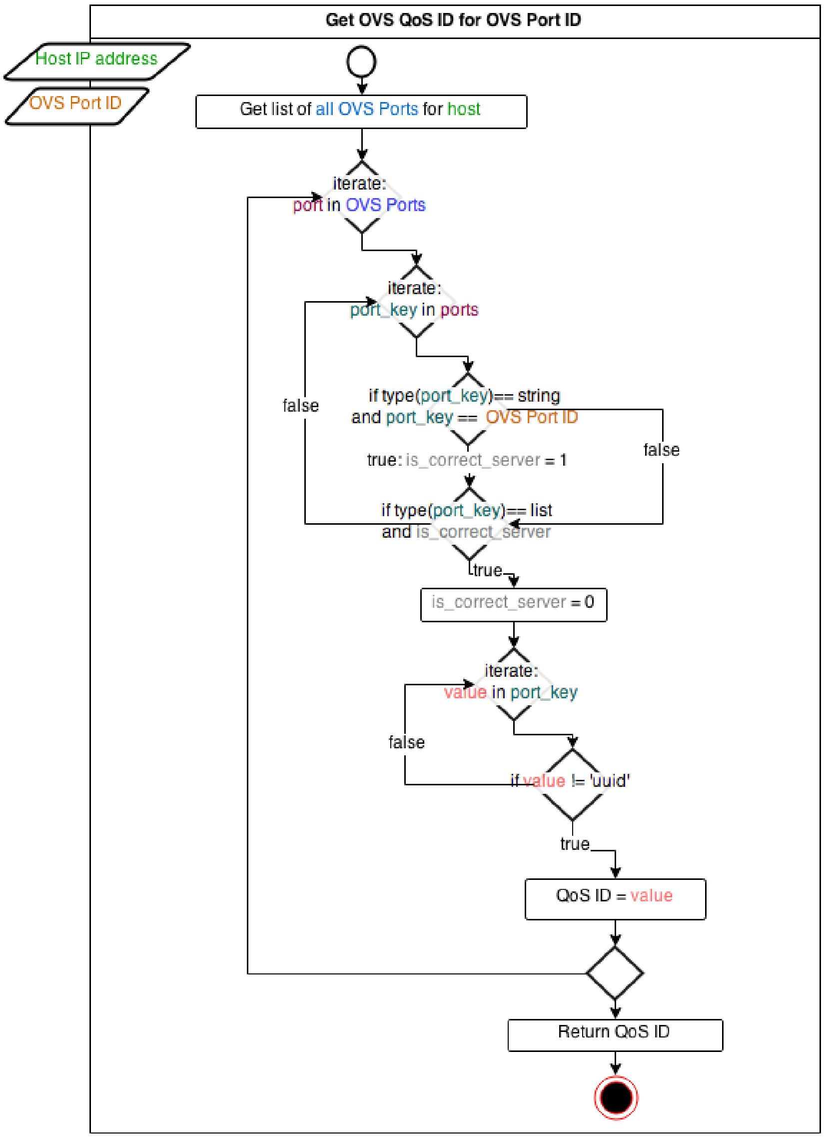

Figure 6.11: Activity diagram: Get QoS ID for OVS Port

Tests

6.4 Conclusion

Chapter 7

Evaluation

7.1 Feature analysis

7.2 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 social multimedia. 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.

7.3 Network performance analysis

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

7.3.1 Test-bed configuration

The testbed consists of two nodes with the following hardware characteristics:

Name	**Controller node**	**Compute node**
Hostname	datacenter-4	dc4-comp
OS	Ubuntu 14.04.1 LTS	Ubuntu 14.04.1 LTS
RAM	12 GB	8 GB
CPU	8-core Intel Core i7-4765T CPU @ 2.00GHz	4-core Intel(R) Core(TM) i3-2120T CPU @ 2.60GHz
Ethernet card	Intel Corporation Gigabit Ethernet Connection I217-LM	Intel Corporation 82579LM Gigabit Network Connection

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

7.3.2 Installation of Connectivity Manager Agent

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

First of all the git repository needs to be cloned from the remote git server. For the installation the cm-agent.sh script needs to be executed with the 'install' option.

```
stack@datacenter-4:~/nubomedia$ ./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 IP address of the controller node and the correct OpenStack credentials:

```
stack@datacenter-4:~$ 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 it can be run in a screen session using the following command:
$ venv/bin/python cm-agent/wsgi/application.py

7.3.3 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 application 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"
            }
        ]
}
```

7.3.4 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.

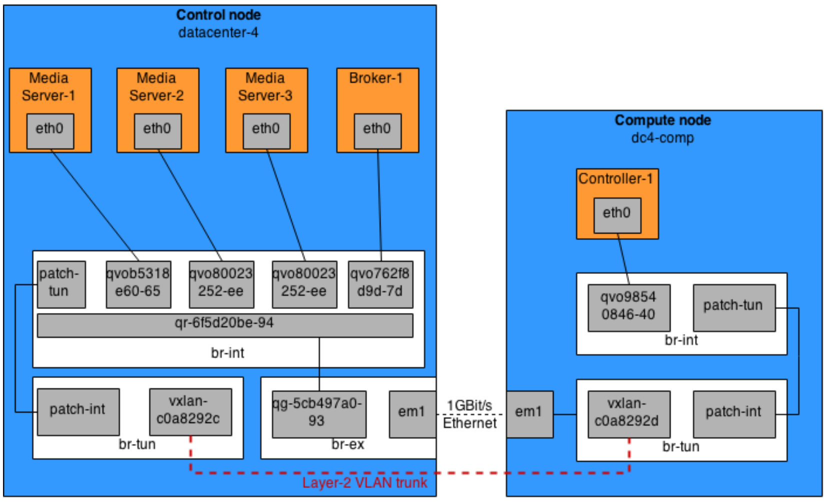

Figure 7.1: Scenario 1: Placement of servers without Connectivity Manager

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.

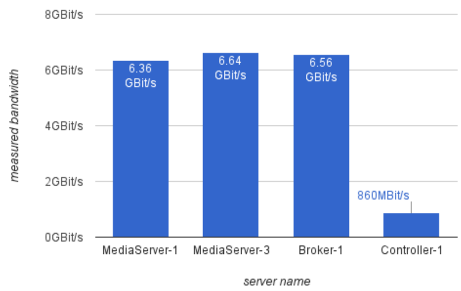

Figure 7.2: Scenario 1: Bandwidth comparison

As 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.

7.3.5 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.

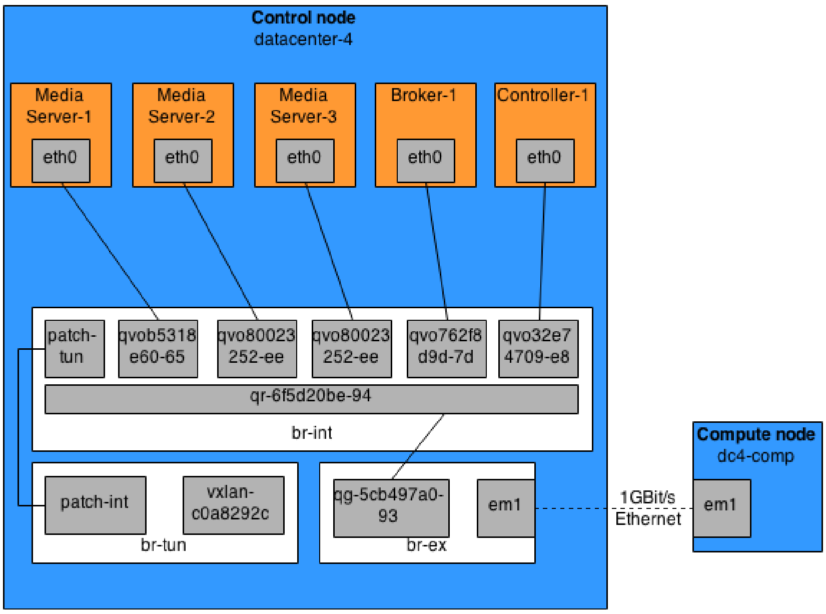

Figure 7.3: Scenario 2: Placement of servers with Connectivity Manager

The following graph shows that the available bandwidth 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.

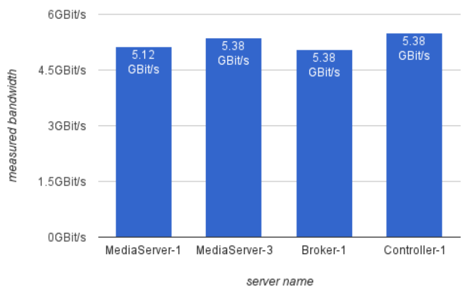

Figure 7.4: Scenario 2: Bandwidth comparison

7.3.6 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.

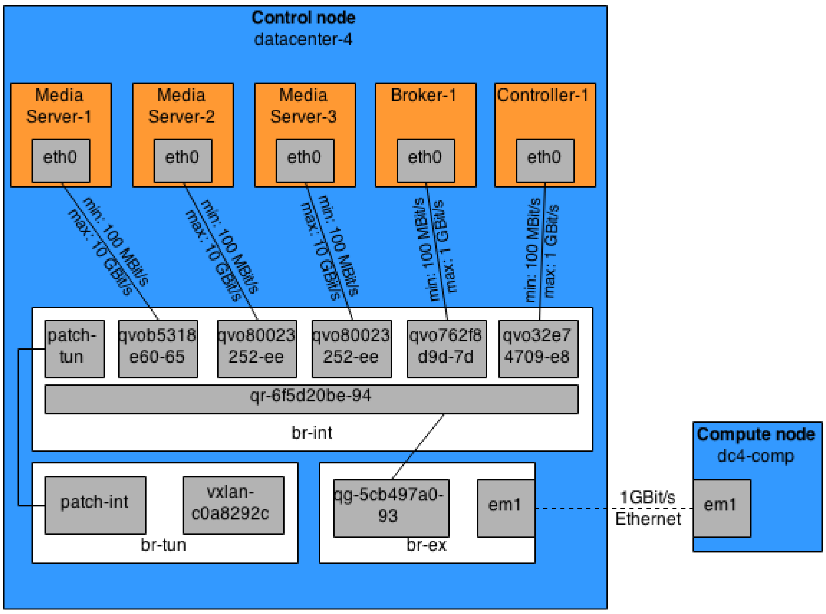

Figure 7.5: Scenario 3: Placement of servers using the Connectivity Manager with QoS enabled

For this last scenario two servers were set as iperf-servers and three servers as iperf-clients. The *'Controller-1'* server acted as the server for a connection with the *'MediaServer-2'* while the *'MediaServer-3'* received packets from *'MediaServer-1'* and *'Broker-1'*.

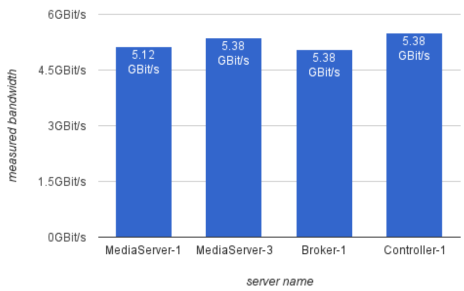

Figure 7.6: Scenario 3: Bandwidth comparison

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.

7.4 Conclusion

The three test scenarios show an improvement of the network performance, which is given through placing the servers on a single host. Additionally with enabling Quality of Service, the network bandwidth is shaped according to the requirements. It has been shown that the connection of MediaServers is prioritized to the connection of servers of other service instance types. The Service-Level-Agreement of the Gold and Wholesale classes are fulfilled.

Chapter 8

Conclusion

8.1 Summary

8.2 Problems encountered

Devstack problems OVS JSON invalid?

8.3 Future work

Integrate with Neutron API as Extension / Plugin QoS Flow matching

Appendix A

List of source codes

Appendix B

Glossar

SDN $\rightarrow$ Software-Defined Networking.

Bibliography

[Abdel-Aziz 71] Y. I. Abdel-Aziz and H. M. Karara, „Direct linear transformation from comparator coordinates into object space coordinates in close-range photogrammetry“, in: Symposium on Close-Range Photogrammetry, issue 11, pp. 1–18, University of Illinois at Urbana-Champaign, 1971.

[AutTech 07] Firma Automation Technology GmbH in 22946 Trittau, Produktübersicht, Downloads und Datenblätter. URL: `http://www.automationtechnology.de`