Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Informatica M

A QoS Controller Framework Compliant with the ETSI Network Function Virtualization Specification

Relatore:
Chiar.mo Prof. Ing. Paolo Bellavista

Presentata da:
Pasquale Carlo Maiorano Picone

Correlatori:
Ing. Giuseppe Carella
Prof. Dr. Thomas Magedanz

Sessione III
Anno Accademico 2014/2015

CONTENTS

INTRODUZIONE

Il Network Slicing è stato introdotto a livello concettuale con la definizione dei requisiti della rete mobile di quinta generazione, in quanto con questa generazione vengono definiti una moltitudine di casi d'uso con requisiti di rete molto diversi tra loro; per soddisfare questi requisiti di rete prima si definivano reti diverse impiegando hardware dedicato, ora visto l'alto numero di casi d'uso, la varietà di risorse impiegate e la varietà di locazioni dei singoli servizi (alcuni devono risidiere per forza ai margini della rete) risulterebbe molto costoso e difficile da mantenere una infrastruttura diversa per servizio quindi è stato introdotto il Network Slicing.
Il Network Slicing consiste in una divisione in "fette" della rete fisica, ogni fetta viene definita con tutte le caratteristiche richieste dal servizio che andrà a utilizzarla. L'attuazione di questo concetto è possibile grazie all'utilizzo di tecnologie quali Software Defined Networking, Network Function Virtualisation e l'utilizzo di piattaforme cloud. L'allocazione della rete utilizzata dal servizio viene fatta tramite SDN, ma la definizione delle caratteristiche richieste viene fatta tramite meccanismi di allocazione di qualità del servizio (sempre tramite SDN).

La qualità del servizio è definita come un set di parametri quali banda garantita, jitter, latenza a livello di rete stabiliti al fine di avere un servizio più o meno affidabile; l'esigenza di questa è nata con la definizione dei primi servizi di streaming.
Le richieste di una "fetta" sono completamente assimilabili ai parametri della qualità del servizio, pertanto in questa tesi abbiamo deciso di definire una *slice* andando ad allocare caratteristiche di qualità del servizio sull'infrastruttura.
La definizione dei requisiti di una *slice* viene descritta nella definizione del Network Service, questa definizione risponde alla specifica ETSI NFV; è stato scelto la compatibilità con questa specifica in quanto è largamente supportata dagli operatori di rete mobile.
Questo lavoro di tesi si occuperà di allocare i requisiti di una slice definiti all'interno di un Network Service Record sull'infrastruttura tramite la definizione di apposite risorse, i requisiti verranno allocati come parametri definiti dalla qualità del servizio.
L'elaborato andrà ad utilizzare tecnologie ben affermate sulle attuali piattaforme cloud quali virtual switch ed Open Flow controller, si andrà inoltre ad integrare ed utilizzerà il framework Open Baton come implementazione di riferimento della specifica ETSI NFV.
I capitoli a seguire andranno ad approfondire il concetto di Network Slicing e presenteranno anche soluzioni già esistenti e piattaforme utilizzate per l'enforcing di parametri riguardanti la qualità del servizio, a seguire verrà presentata l'architettura proposta della mia soluzione; il capitolo successivo tratterà invece l'implementazione della soluzione ed in seguito verranno presentati i risultati ottenuti.

CHAPTER 1

BACKGROUND IN NETWORK SLICING

The 5th generation of mobile networking introduces a mixed scenario, made using all the technologies available. 5G services introduce the concept of fully mobile and connected society, requiring a multi-layer densification of networks to enable differentiated services based on the use case for the business model expected.

The multilayer network also requires an "envelope" of performances to provide them where needed. This envelope requires also a suitable network flexibility, there is also the requirements in terms of speed, scalability, security, reliability and latency plus an improved power management for the new radio interface to get a longer battery life.

All of this needs require an infrastructural update for the RAN to match the thousands of new use cases, many different subscribers types and varying application uses that as to be supported. Evolving the radio access technologies (like LTE and the future 5G RAT) will enable cross-domain integration in environments that could allow multiple radio access technologies; this upgrade has to enable concepts like very low latency and higher bandwidth.

The 5G scenario will increase the traffic by a thousand times, requires an increase of the average speed connection and the applications made for this "environment" could have stringent requirements in terms of bandwidth and latency. Not all the use cases has stringent requirements for bandwidth and/or zero (or closer to zero) latency and/or super smart network, instead they could have normal requirements with "relaxed" constraints; this could bring to a flexible network, flexibility intended as an "adaptable" network.
This flexibility is useful to introduce the concept of network slicing. Network slicing is a logical division of the network into "slices", where each slice has different characteristics in terms of bandwidth, latency and all required parameters for different services; the slices are made to meet the demands for each use case.
For example we could have a service with high requirements for a limited geographical area and another service with "standard" requirements in the same area, nowadays this two will be implemented as two separated networks with two infrastructures; this different implementation has near-double costs for the operator and after the "disposal" of first (the high performance one) service the specific purpose network will be teared down with relative costs and the infrastructure remain unused. In 5G network will be an unique base with very high performance and the capacity of divide the network into slices (as depicted in figure 1.1), that will be allocated using parameters compliant with the services request.

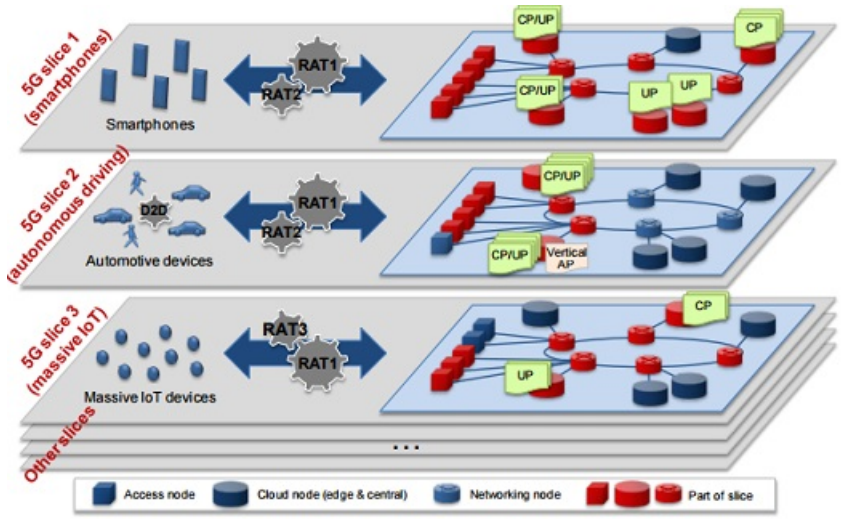

Figure 1.1: The network slicing in 5G architecture

Among the services that are heavily used in the LTE (4G) networks there is also the video (and media content in general) streaming, which use a significant portion of the overall network traffic; this service has different requirements based on the accounting of provider, which could have different requirements based on the "class" of service that it want provide. This differentiation in LTE was not made because the architecture does not support multi RAT technology, and many services available under LTE coverage are not available under the older RAT generations. This different classification of traffic under RAT (LTE or 5G) will be possible, will be also possible a quality scale depending on which radio technology is available for that determined geographic area.

To satisfy every use case, all the technologies that are used to compose the 5G architecture relies on logical instead of physical resources; this solution enable the possibility to deliver network on as-a-service base. This flexibility allows operators to create networks on demand and tailor the network slices as requested.

However the resulting architecture could not scale in a proper way on telecommunication operators networks, in this contest high adaptability and reliability in extreme use cases (for example emergency situations, natural disaster); A big solution is represented by the Cloud Computing technology and the Software Defined Networking, combined with NFV there are solutions for handling all aspects of the vertical architecture and manage all the proposed use cases. Using this services divide in building blocks all the elements of vertical architecture, using this solution in chain having as result an horizontal network.

In 5G architecture, using NFV on SDN and Cloud Computing, a service could be abstracted in a network slice, using software defined functions that has control in a specific geographical area with defined network requirements such as latency, minimum bandwidth, security, robustness.

Slicing networks provides a huge optimization in network resource usage, allowing the maximum customization for each slice to match the level of delivery desired by the services that will use the slice.

Main purpose of this thesis is to enable the Network Slicing on infrastructural level, implementing features and using platforms which allocate quality of service in order to guarantee the slice requirements. Enabling this feature inside operators' network will increase the efficiency of infrastructure, reduces the cost for maintenance of the entire system decreasing the hardware diversification for each service; moreover the environment become more flexible to hardware faults enabling the path redefinition and offer a more reliable service level agreement.

1.1 Use cases

There are many use cases for the 5G architecture, most of them are focused on multimedia traffic delivery and M2M communications; as wide area network infrastructure has to cover also the mission critical services and be resilient to natural disaster.

1.1.1 M2M Communications

The sensor networks, smart meters services has very high rate of data transfer with small payloads, they requires a guaranteed bandwidth and near-zero latency to provide a efficient service and significant informations; the requirements in terms of security are medium/low.
To satisfy these requirements could be allocated a network function, which will provide security and replication for data "gathering"; The NFV Orchestrator could use the interface with SDN Controller to enable the requirements of network slice on the physical and virtual resources.

1.1.2 Multimedia Service Delivery

The multimedia service delivery requires a high network throughput to support a high video quality, requires also a very low latency and near-zero skew.
To meet these requirements could be defined a network function with a robust and maybe replicated service providers and high performance enabled slice.
In this case the requirements of high performance network is the key to achieve a very high quality, the security requirements are low.

1.1.3 Mission Critical Services

The mission critical services are defined to guarantee a reliable first aid immediately after a disaster, this services requires a considerable amount of bandwidth, near-zero latency and a very reliable infrastructure; this kind of services are requested from governments, first aid organizations, the critical aspect of this use case is the business agreement which is very stringent with a probably damaged infrastructure.

1.2 Network Slicing in NFV Environment

Network slice requires SDN and NFV to be in place with Cloud computing platforms to hide the underlining physical infrastructure and provides logical resources for CaaS (Connectivity as a Service).
The devices will use the connectivity in a smart way, using a minimal signaling part and lean data transmission. The infrastructure provides E2E (End to End) Security which enables the mashing of multiple services ensuring the trust from each source. The E2E security has to be achieved from services but the infrastructure could support this feature allowing multi path for key negotiation.
The Network Slicing concept relies strongly on NFV, because it enable the implementation programmatically instead of design and create new hardware only for that specific function (as in the first generation networks). NFV brought also the benefit of executing virtual network function in different locations, with different slice requirements; this feature also enables the management of data centers at the “edge” points of the network, that could be addressed and where could be deployed specific network functions that requires to be placed at the edges.

Another functionality on which Network Slicing relies is Software Defined Networking (SDN). SDN provides a logical vision of the network hiding the real network infrastructure, enables also the programmability of networks allowing different logical behaviors on the same physical network. It allows also the differentiation between services, in terms of network requirements; enabling de facto the capability of create network slices for a single location (as depicted in figure 1.2).

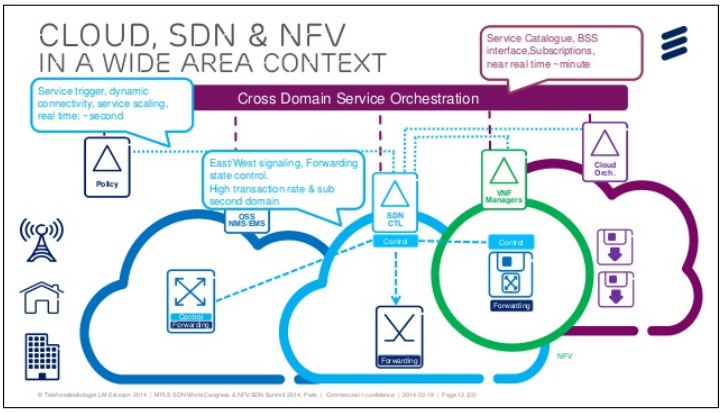

Figure 1.2: The NFV plus SDN scenario[1]

This scenario requires a high capacity of provisioning, automation using less resources possible; could became necessary a centralized "command point" like an Orchestrator and a SDN Controller in every location (or a federated structure). Another aspect that become relevant is the increased requirement of SLA because the network, as we said for the Network Slicing, became to have an active role in this scenario.
The automation is provided by orchestration, that is one of the main components in NFV.

1.2.1 The ETSI NFV Architecture

ETSI NFV MANO[17] is a specification which describes the management and orchestration framework for Network Function Virtualisation (NFV). It defines the provisioning, related operations (for example the configuration of virtual network functions and relative infrastructure on which they relies). The document addresses problems of management and orchestration, starting from the architecture framework define the guidelines for information elements, interfaces, provisioning, configuration and operational management including integration with present systems.

The architecture relies on one fundamental element that is the NFV Orchestrator, which has an overall vision of every resource allocated to each data center under its "control". The document defines three main functional blocks:

- **Virtualised Infrastructure Manager:** is the functional block that communicates with the NFVI, which represents the physical infrastructure of the cloud environment
- **NFV Orchestrator (NFVO):** its objective is to provide the orchestration level, location independent; it uses VIM and VNF manager to instantiate Network Services, the decision on who perform the operations is on behalf of the VNF manager.
- **VNF Manager (VNFM):** this component handle a specific Virtual Network Function, the specification consider also a Generic VNFM which delegates all instantiation operation to the NFVO

The architectural framework defines also the data repositories where store descriptors and define configurations, this elements are called Catalogue.

The document specifies also a other functional block (in figure 1.3) in order to share reference points with NFV-MANO:

- **Element Management System:** is the component in-place on the VNFC instances which composes the virtual network function
- **Virtualised Network Function (VNF):** is the network function that is developed by external vendor and uses the NFV Infrastructure to be deployed
- **NFV Infrastructure:** is the representation of the cloud platform, it abstract the physical infrastructure in one element which uses the entry points that the infrastructure expose to instantiate a VNF

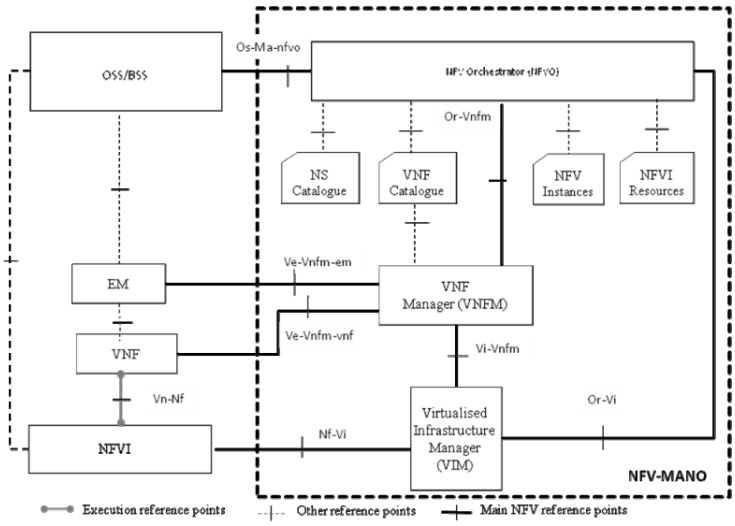

Figure 1.3: The architecture of ETSI main functional blocks

QoS in ETSI NFV Architecture

The documents expose also how has to be defined (in a structural way, not implementation) the virtual network function. It defines the Network Service

as a composition of multiple Virtual Network Function (VNF), each VNF is composed by multiple VDUs that is the definition of "physical components" of the VNF and virtual link.

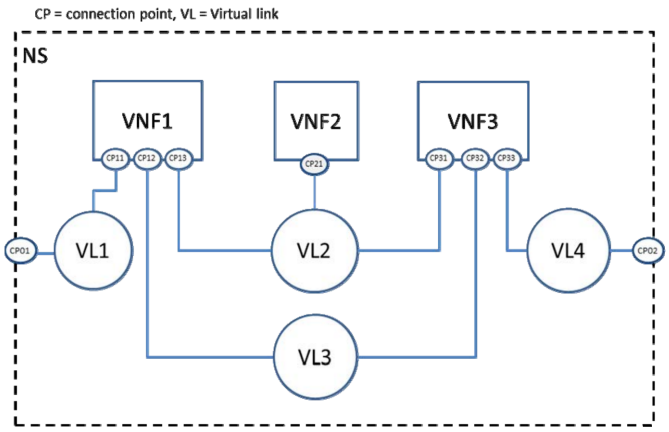

Figure 1.4: Virtual Link in ETSI Architecture

The Virtual Link is defined with multiple properties (in figure 1.5), one of them is QoS; the QoS properties of the virtual link is applied at every VDU that referees that virtual link in the connection point.

Identifier	Type	Cardinality	Description
id	Leaf	1	Unique identifier of this internal Virtual Link.
connectivity_type	Leaf	1	Connectivity type (e.g. E-Line, E-LAN or E-Tree).
connection_points_references	Reference	2...N	References to Connection Points (vnfd:vdu:vnfc:connection_point:id, vnfd:connection_point:id), e.g. of type E-Line, E-Tree, or E-LAN.
root_requirement	Leaf	1	Describes required throughput of the link (e.g. bandwidth of E-Line, root bandwidth of E-Tree, and aggregate capacity of E_LAN).
leaf_requirement	Leaf	0...1	Describes required throughput of leaf connections to the link (for E-Tree and E-LAN branches).
qos	Leaf	0...N	Describes the QoS options to be supported on the VL e.g. latency, jitter, etc.
test_access	Leaf	0...1	Describes the test access facilities to be supported on the VL (e.g. none, passive monitoring, or active (intrusive) loopbacks at endpoints.

Figure 1.5: The Virtual Link Definition

The QoS parameters are a superset of the properties required by the Network Slice definitions, specifying these a developer of a specific VNF could easily define a network slice for its specific network function; as consequence in one Network Service could be defined (and coexist) more than one single network slice.

1.2.2 NFV and SDN

NFV, as we already said, uses SDN to provide a scalable, customized and reliable service in each location, uses also it to define multi path provisioning to reach the edge network and allows the communication between multi point of presence Network Services.

NFV relies on SDN for the entire part of network definition. SDN hide all the infrastructure and the physical network to the VNFs defining overlays network; the VNF has the vision of this defined network with the capacity in terms of bandwidth, latency and other parameters that are requested from

the NFV "plan".

SDN

Software Defined Networking decouples the control plane from the data plane (as depicted in figure 1.6), enabling a centralized entity for network control and path definition. Typically the SDN controllers define a new layer on top of the infrastructure layer (the physical network), this layer enables the programmability of it.

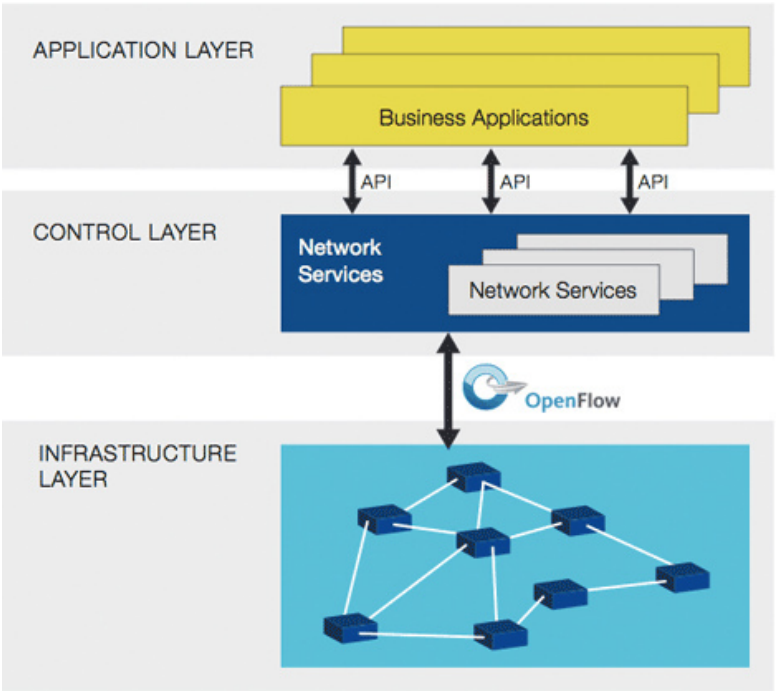

Figure 1.6: The SDN stack

SDN presents an architecture that is:

- **Directly Programmable:** as we already said, the network is directly programmable by developers using the decoupling of control plane from the data plane

- **Agile:** the programmability enables also the dynamic, on demand, configuration of the network allowing the modification of the traffic flow network-wide
- **Programmatically configured:** the SDN architecture could be orchestrated using standard application that uses the controllers APIs and is not dependent on proprietary software and interfaces.

Software Defined Networking was defined to address common problems, such as:

- *Changing Traffic Patterns:* for application that uses different database in different locations (in different moments) is required an extremely flexible traffic management and access to the bandwidth on demand
- *The rise of cloud services:* the definition of new services in cloud environment requires an high reconfigurability of the network, because the user could require different networks with different characteristics
- *Ability to scale:* the software defined networks could add easily more users, allowing the automatic provisioning of new resources.
- *Any-to-any connection:* the agility of SDN enables the reconfiguration of paths on demand, this favorites applications for Big Data processing

The main protocol used in SDN is OpenFlow, which defines all the control plane and management protocol to enable custom network definition in cloud and wide-area environments.

NFV

Network Function Virtualisation is the new approach for instantiation, management, provisioning and orchestration of Network Service used by telecoms

operator; this paradigm relies on Cloud platforms and Software Defined Networking. Figure 1.7 depict the ETSI NFV Architecture.

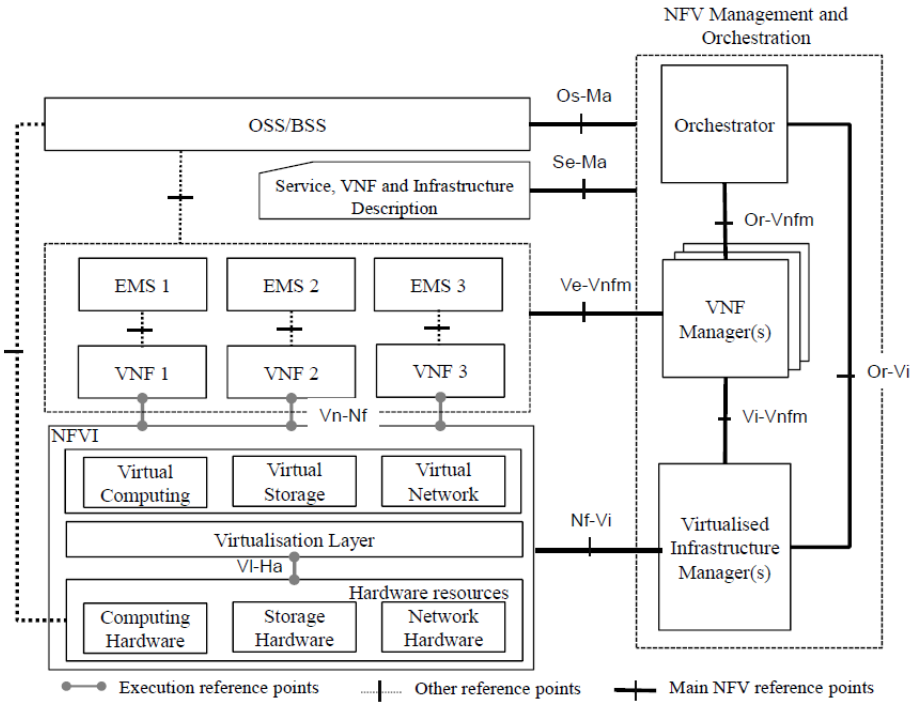

Figure 1.7: The ETSI NFV Architecture

NFV exposes different functional blocks, the main interest is in NFV Orchestrator which represents a centralized point of management, provisioning and handling of Network Services.

The Network Services are a set of virtual network function which are combined to define a service. This services could be localized on a single datacenter or they can be designed to have a core inside the main datacenter and some other functions on the edge of the network. The Orchestrator deploy this functions on the location specified on the descriptor (or deploy template) and define the intra-functions network delegating to the SDN Controller the definition of paths (with requirements or less).

1.3 Introducing mechanisms for enforcing QoS requirements

Resource virtualization (including the network resources, such as switch, router and entire network itself) define a new paradigm called SDN (Software Defined Networking). SDN is about defining abstractions that expose an appropriate level of detail for complex network functions, for implement this level of abstraction n APIs for programmability, automatic provisioning, configuration and management has to be defined.
The programmability (with all the features that derives from it) is one of the keys of SDN, another big feature is the re-usability of the configurations and the flexibility for modification and adaptation to a new scenario, decoupling the logic control from the data plane.
The services providers make a large use of cloud computing platforms for service hosting, exploiting the flexibility for configuration of that environment; this feature has as cons the increased difficulty for traffic management from and to the cloud platforms. The heterogeneity of services hosted on cloud platforms brings different requirements depending on the type of the service itself.
The Cloud hosted applications maintains the same QoS negotiation mechanism. As a result automated negotiation is needed to accommodate different consumer's QoS requirements. The result of such a negotiation is a Service Level Agreement(SLA), an electronic contract that establishes all relevant aspects of the service. The SLA contains the QoS requirements of Applications hosted by a cloud based computing platform, such as timeliness, scalability, response-time, throughput, failure probability, and dependability (availability, security, safety, reliability, etc.); which are normal QoS requirements for

a cloud host, when the hosted application has also network QoS requirements the cloud platforms has some limitations.

In SDN the network are virtualized as well as the other resource, but virtual networks has some limitations. SDN networks are based on VLAN, VXLAN and GRE overlay networks that encapsulate the normal IP packets and allows sharing of the same physical link for more than one virtual network. VLAN is a broadcast domain partitioned and isolated in many computer network. Each computer network is attached to a port of network switch. Simpler network devices can only partition per physical port (if at all), in which case each VLAN is connected with a dedicated network cable (and VLAN connectivity is limited by the number of hardware ports available). GRE segmenation (and VXLAN) also provides network isolation, and also allows overlapping subnets and IP ranges. It does this by encapsulating tenant traffic in tunnels.

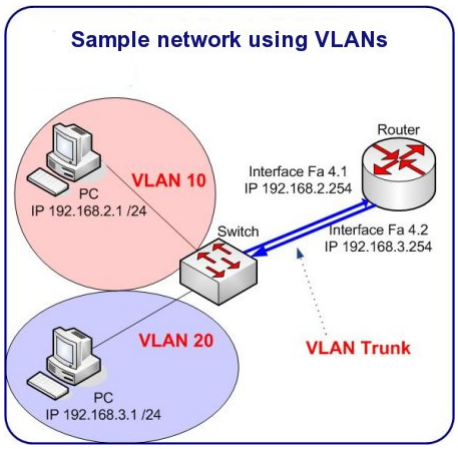

Figure 1.8: A simple VLAN schema

Cloud computing exploit the VLAN (and VXLAN and GRE) network isolation, enabling the overlapping of network address. This is possible using a virtualized network switch (usually Open vSwitch) which create separate

VLAN for network on each tenant.

No. of bits	40	4	4	48	16	24	24
Frame field	DA	TYPE	USER	SA	LEN	AAAA03(SNAP)	HSA

No. of bits	15	1	16	16	8 to 196,600 bits (1 to 24,575 bytes)	32
Frame field	VLAN	BPDU	INDEX	RES	ENCAP FRAME	FCS

Figure 1.9: The VLAN packet

The network isolation and packet encapsulation are made "encapsulating" the entire packet in a VLAN (or GRE or VXLAN, depending on which virtual network the cloud platform are using) packet which transform completely the pakcets (as shown in figure).
The network QoS support is possible only at switch level, which is the only component that has a "complete" vision of VLANs and network infrastructure of the cloud platform.
The most used and common implementation of virtual network switch is Open vSwitch, is a production quality, multilayer virtual switch. An instance of this virtual switch is present in every node (compute and controller node) of cloud infrastructure.

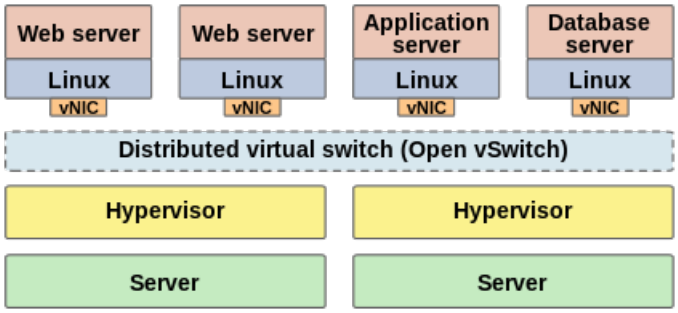

Figure 1.10: Open vSwitch distributed configuration

Open vSwitch uses a database to store all platform (or user) defined components. The components are, in hierarchy:

- **Bridge:** is the representation of a virtualized bridge where can be allocated one or more ports, for example Openstack define two bridges (br-int, br-ex) to define internal ports and floating ports
- **Port:** represent the physical port of the bridge, it can be connected one interface at time; if the port fail the relative interface must be migrated
- **Interface:** is the physical interface that belong to the virtual machine, this interface is connected to the bridge port
- **Qos:** is the identifier of allocated queues it can be linked to a port, a port could not have more than one Qos identifier
- **Queue:** is a representation of internal queue with specific QoS parameters, is similar a class of traffic in DiffServ; the supported configuration are:
  - *other_config : min-rate* represents the minimum bandwidth (guaranteed) for that queue

– *other_config : max-rate* represent the bandwidth upper bound, the maximum throughput of the connection will be limited to this value

– *other_config : priority* is the priority between the other queues that belongs to the same qos

The OVSDB has more entries but is not used to define QoS, instead are used to store internal data; in the database there is also the flow table, but it does not belong to any hierarchy; the delete operations on this table are made by pattern matching, so is possible to delete more than one row with one command.

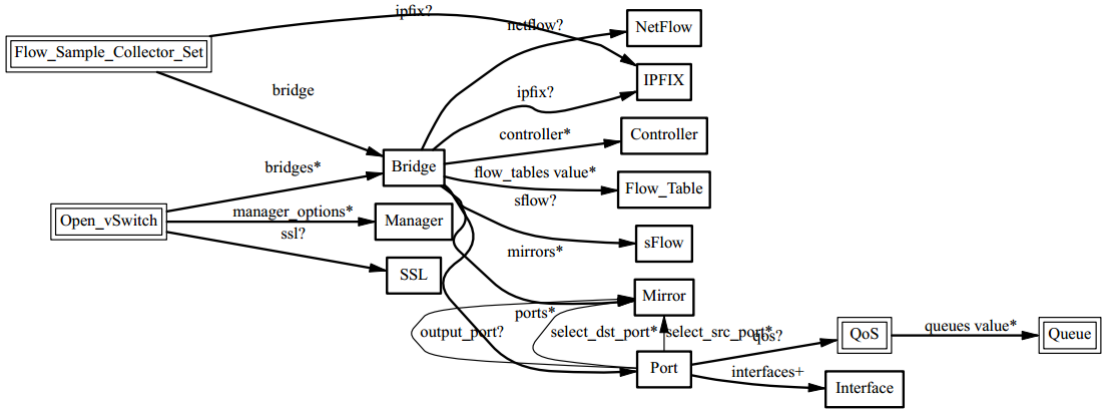

Figure 1.11: OVSDB schema

Open vSwitch expose only command line interfaces to developers, it has a kernel space module which uses the Linux Traffic Control implementation for the Linux kernel; tc uses queuing discipline (qdisc) for network interfaces' configuration. qdisc is considered as the essential scheduler of Linux. Pure First In First Out (PFIFO) qdisc is the default queue type used in Linux kernel. QoS in OVS uses Hierarchy Token Bucket (HTB) class-based qdisc to

schedule packets of queues. It exposes also binaries to interact with OVSDB from the QoS side (*ovs-vsctl*) and OpenFlow side (*ovs-ofctl*). The distributed deployment of Open vSwitch could be administrated using a controller (which could be settled using cli) that allows creation QoS and flows to the node which is settled as controller.

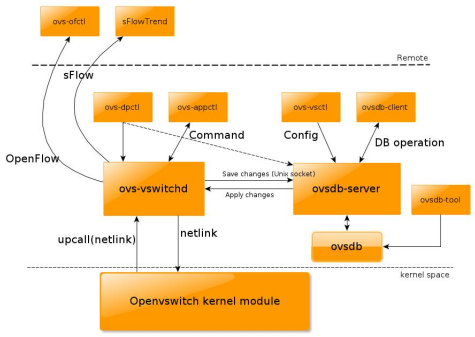

Figure 1.12: Open vSwitch internal architecture

1.4 Existing solutions for enforcing QoS requirements

The NFV architecture started to facing QoS problems since when the first streaming platform began to use the mobile telco infrastructure. This kind of platforms has a big requirements of bandwidth and very reduced latency. In a scenario which there are multiple data centers involved and each data center has its own network architecture, with different hardware and capability. This differentiation plus the wide area network bring to define an architecture with a **QoS Controller** plus **SDN Controller**.

The QoS Controller is the component that handle the paths (or the connectivity) between the machine/client, it can use different technologies or different algorithms (for path definition) or both. There are also solution

that uses a "all-knowing" SDN controller to allocate flows on the physical switches with a path-decision algorithm that runs inside the QoS Controller, which has a overall view of the topology.
The SDN Controller is, instead, a commercial product such as Open Day-Light, Floodlight, Onos or other custom application developed for allocate and define QoS requirements and flows in the datacenter. Normally this products exposes REST APIs (or RESTConf as Open Daylight does) to enable a path definition in dynamic mode, but also define a GUI (typically Web GUI) to instantiate flows between physical and/or virtual machine in a static way.
On top of the architecture (depicted in figure 1.13) is mandatory to have the Network Function Virtualisation Orchestrator which define, instantiate and manage new Network Service Records in order to define services with associated network slices. The slices are defined in the Network Service Descriptor, but in order to avoid a complex scenario all the related works with QoS controller are made using a static use case with predefined services and using categorized slices; as is made in DiffServ use case.

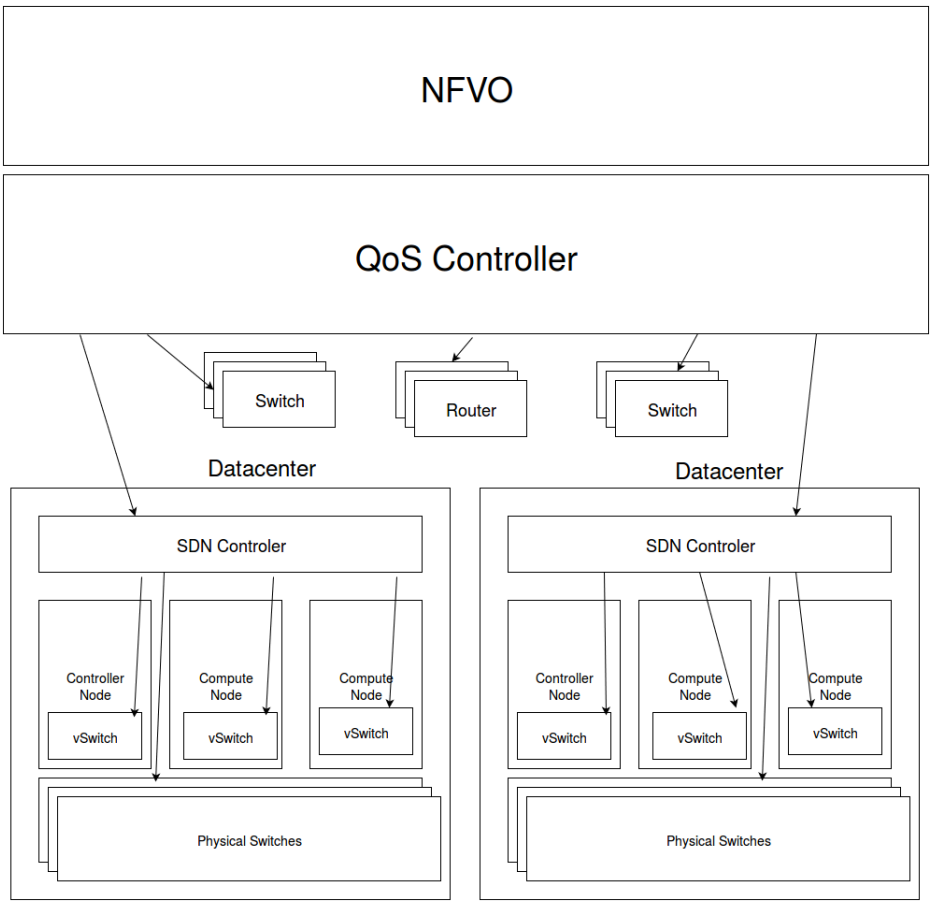

Figure 1.13: The proposed architecture in literature

To define a QoS controller there is always the assumption that the public network is known as the topology of network. This assumption was made to present a use case that could be feasible for presenting significant results.

1.4.1 SDN Controllers

The SDN Controllers are the main actors to handle a single data center (and sometimes not only a single datacenter), this platforms are designed to control all the hardware and software infrastructure for networking.

The SDN Controllers has specific requirements for hardware designed to han-

dle the networks:

- **OpenFlow Compliant:** OpenFlow requires a control "port" that enables external control (made by the Openflow Controller)
- **Persistency of Flow Definition:** requires a persistency for the flow definitions, in case of reboot or failure is better to have the information stored to have a restart time very reduced (otherwise we could violate the SLA) definitions)

The SDN controller act as strategic control point in the SDN network, manage flow control to the switches using the business logic defined in the applications; the flow and switch management are made through the southbound APIs, whereas the logic is "injected" through the northbound APIs. The virtualization technology define a lot of SDN networks, which are handled through common interfaces such as OpenFlow or Open vSwitch database (OVSDB) and custom interface for example the one that are vendor specific for the switches.
A SDN controller typically contains a collection of "on demand pluggable" modules that can handle different devices (physical or virtual) and/or which can perform different network tasks. They has basic module for handling common devices (OpenFlow virtual controller, OVSDB), gathering devices informations, network capacity; a developer or a vendor could create a specific extension to support new features or add new devices.
The two main protocol used are:

- *OpenFlow:* is a network protocol that gives to developer access to the forwarding plane of (physical or virtual) switches through the control plane, is used to determine the path of network packages.

- *OVSDB:* is a protocol that define access to Open vSwitch database, where are stored all the configurations for a switching daemon, the information stored represents the behavior of a single virtual switch and does not describe routing for the entire system.

OpenFlow attempts to centralize all the forwarding decisions, so a packet follows the flow rules and the source and destination switches applies the configured behavior for that packet (or that kind of packet).
Currently all the Open vSwitch implementations has an OpenFlow Controller inside them, exposing also a command line interface (or JSON RPC APIs) to define flows inside virtual switches.

SDN Controllers main implementation

There are many implementations of SDN Controllers, everyone of them wants to orchestrate the vendor equipment and possibly extendend to use the other; there are also Open Source controllers that are vendor neutral.
The most used open source implementations are:

- *Nox:* is the first OpenFlow Controller, developed by Nicira Networks, is written completely in C++ (new Nox) and supports less application than the others; it supports only normal switches and OpenFlow switches. Nox has also event support to detect the join or leave of new switches in the network (could be virtual or physical) or other "upper level" event for example flow events or ports events. The figure 1.14 depicts the Nox architecture

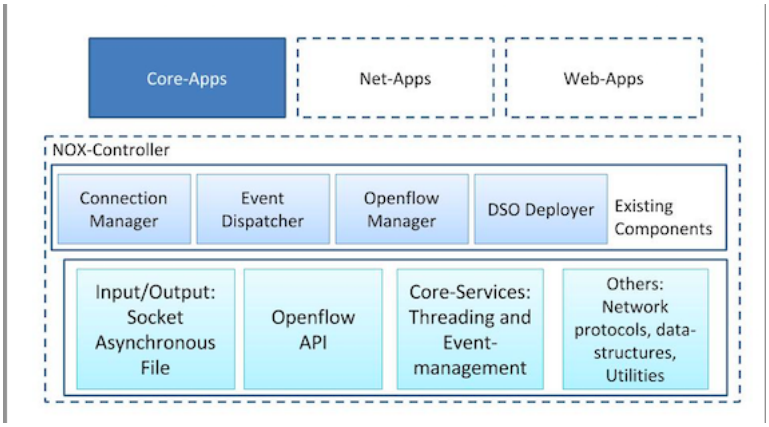

Figure 1.14: The NOX overview

- *Beacon[14]:* the most popular implementation of OpenFlow Controller, is completely written in java and is used as base for the most moderns implementations such as Floodlight and (partially) OpenDaylight. Beacon uses the Spring framework[6] to expose northbound REST APIs to the application, uses also the OpenflowJ library[12] to interact and deserialize messages from OpenFlow. The figure 1.15 depicts the Beacon architecture

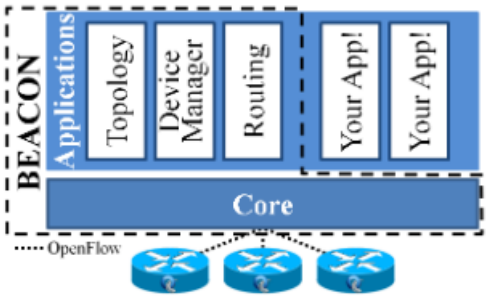

Figure 1.15: Beacon Overview

- *Ryu:* is an SDN Controller which exposes well-defined APIs to developers (as depicted in figure 1.16), written in python is maintained by

the community with contributions by Openstack until it switched to OVSDB with direct access from Neutron.

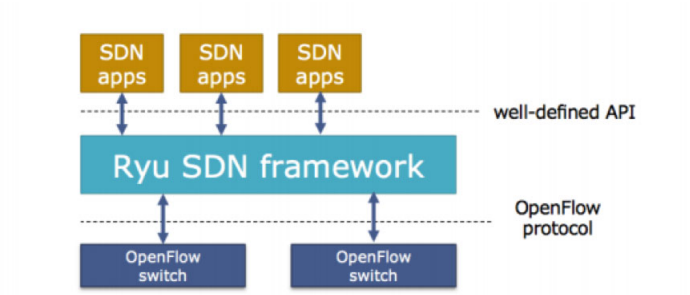

Figure 1.16: Ryu Overview

The presented SDN Controller are used (especially Beacon) as base for more complex and rich platforms that are currently used in production environments. Most of this complex platforms could be integrated in a cloud environment.
The most popular cloud platforms are: OpenDaylight, Floodlight and Onos.

Floodlight

Floodlight[5] is an Open SDN controller, works with physical and virtual switch. It is based on Beacon (originally was a fork project) to interact with OpenFlow.
The Floodlight controller (depicted in figure 1.17) exposes northbound API to the Application Tier to enable the applications to define the expected network behavior. The southbound APIs uses Indigo[9] to realize the communication and the flow definition to interact with the physical or virtual OpenFlow compliant switches.

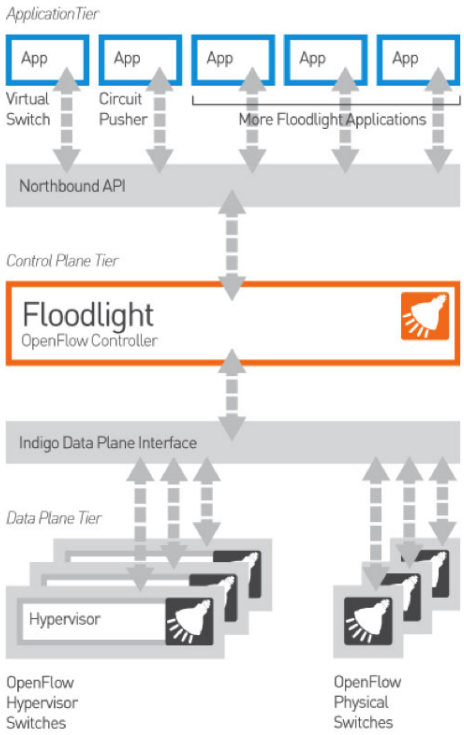

Figure 1.17: The FloodLight architecture

The controller realizes a set of common functionalities to control and inquire an OpenFlow network, depending on which module are enabled the functionalities available on Application Tier are differentiated. Each module is independent from the others as depicted in figure 1.18. Floodlight controller is language independent on the Application Tier, also the modules that it expose are "callable" through the APIs.

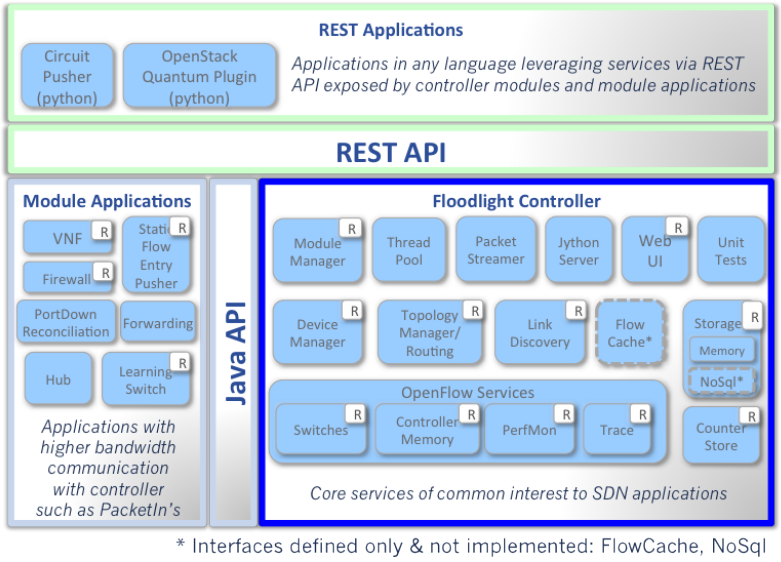

Figure 1.18: Floodlight Diagram

This SDN Controller could be configured as Neutron backend for Openstack enabling Floodlight as default network manager for the datacenter; Floodlight realizes this feature using two modules: Virtual Network Filter and RestProxy.

OpenDaylight

OpenDaylight[15] (ODL) is one of the most famous SDN controller. The figure 1.19 shows that it exposes northbound REST APIs to developers, has an intermediate level where are located the modules that could be activated to run different types (also called OpenDaylight internal projects) feature that OpenDaylight exposes; under the module layer is located the Service Abstraction Layer in which lie all datastores, messaging systems and allows to support multiple protocols providing consistent services for the upper layers. The southbound is the protocol layer, all the supported protocols are linked to this layer using a plugin mechanism; as default OpenDaylight starts only

the plugin OpenFlow 1.0.

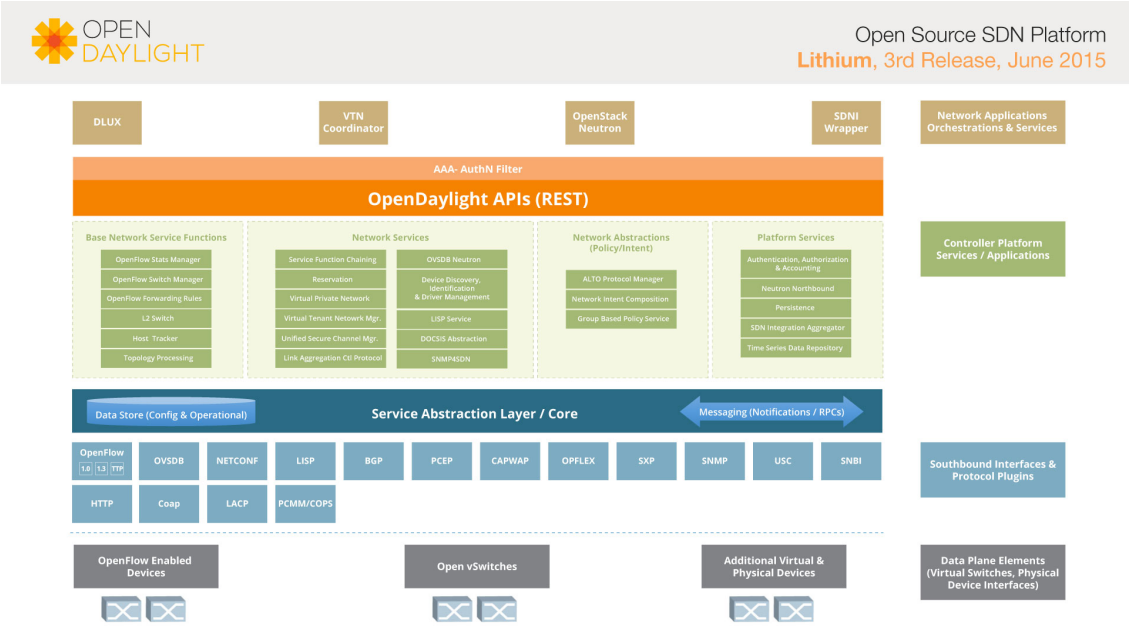

Figure 1.19: The OpenDaylight Layered architecture

The OpenDaylight controller is written in Java and could run on all machine that supports a standard JVM.
The modules that are available on OpenDaylight are pluggable on demand, most of them was introduced with the recent releases and most of them was on it since the first release.

- *AAA:* is the module that provides Authentication and Authorization for all operations that the developer want to execute on the Controller.

- *OpenFlow Protocol Library:* is the component that mediates the communication between the controller and OpenFlow enabled switches (physical or virtual)

- *OVSDB Integration:* The OVSDB NetVirt project is a project for OpenDaylight that will implement a network virtualization solution. The northbound module will handle the all the Open vSwitch based virtualization platform and the southbound protocol plug in will interact with all OVSB based switches.

- *NeutronNorthbound:* The neutron northbound project is the API definition for Neutron Server to register OpenDaylight as Neutron plugin. It aims to only to handle the neutron data and store in the ODL data store to enable other providers registered in the controller to use it (based on the Neutron requests).

OpenDaylight support also the High availability in a cluster using controller replication.
The integration of OpenDaylight in a cloud environment is possible just using the base exposed services plus the Neutron Northbound project to declare the platform as Neutron main controller.

Onos

Onos[13] (Open Network Operating System) is a distributed SDN controller with instance coordination and replication (as depicted in figure 1.21). This platform has 4 main features:

- *Distributed Core:* this component provides scalability, high availability and fault tolerance; the core instances collaborate also to define a global overview of the network infrastructure. All the request performed through the Northbound APIs are dispatched transparently to the instances of the network core.

- *Northbound APIs:* the northbound APIs are divided in two big frameworks:
  - *The Intent Framework:* this framework allows the developer of an application to request a network service just providing the requirements, ignoring how are performed the actions or how is structured the network; its objective is to provide high level programmability specifying only a policy statement or connectivity requirement. Figure 1.20 depicts the architecture of the Intent Framework

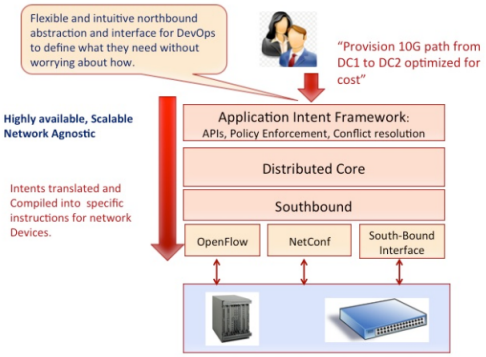

Figure 1.20: The intent framework

  - *The Global Network View:* provides to applications, network administrators and developers the global view of the network, including hosts, switches, links; this framework maintains a snapshot of the network traffic and network flows. This framework could provide different level of informations based on the developers requirements insulating the underlying levels.

- *The Southbound Abstraction:* this component abstract the network elements in object in generic form; the network elements are switches, hosts or links. The element abstraction simplifies the visualization to

Distributed Controller, who sees every element like a generic object with common properties as state; this make the controller southbound and driver agnostic. Architecturally the southbound is composed (from the upper level):

  - *Southbound API:* the API exposed to the controller, to perform operation on the generic network objects
  - *Adapters:* the definitions of the adapters, like Devices, Hosts and Link
  - *Protocols:* the supported protocols like OpenFlow, NetConf etc
  - *Network Elements:* the physical or virtual network elements that composes the architecture

- *Modularity:* like ODL also Onos is modular, so is easy to plug a new protocol, application or "behavior"

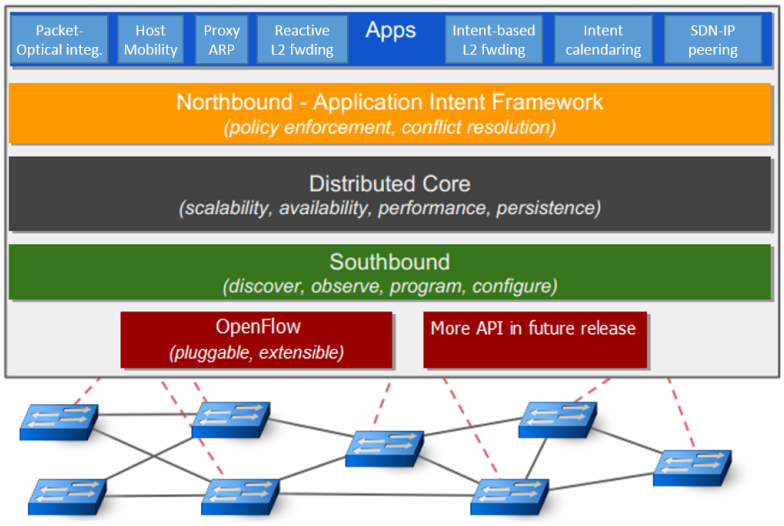

Figure 1.21: The Onos Architecture

The complete architecture is a multi module application which can enforce the desired behavior (if the application intent framework has the appropriate module) and allow an high level programmability with a sufficient amount of information.

1.4.2 QoS Controllers

The SDN Controllers has a lot of implementations and also commercial products (see the previous section), instead the QoS controllers doesn't have commercial product but there are some research work.
There are also related works that are introducing the concept of SD WAN (Software Defined Wide Area Network) but are still experimental.
This kind of controllers assumes that hardware in the path between the datacenters or between the service and the clients are monitorable or under direct control of developers (or directly under control of a SDN Controller).
The objective of QoS controller are: check if is possible to create a path between the source and destination (or the two datacenters), evaluate the possibilities in terms of costs and SLA requirements (if presents), choose the path that fits perfectly the requests and (if possible) is more reliable and enforce this parameters on this path through a SDN Controller.

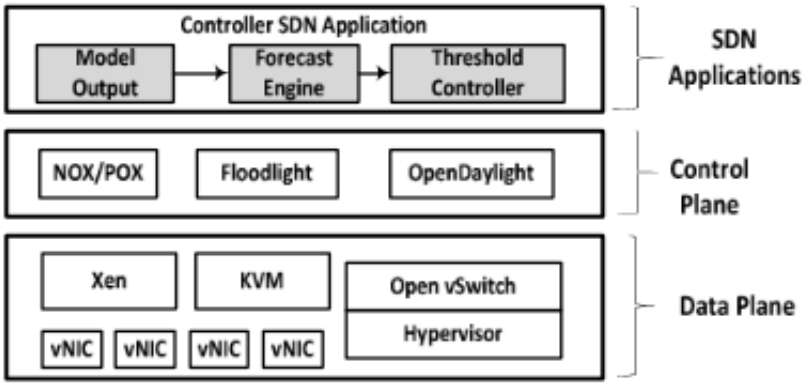

Figure 1.22: An example of system architecture

Most of the works present in literature are considering the use case of enforcing QoS in client/server interaction, but the client could be replaced by another data center.
There are a lot of works that uses the cloud platform virtual switch to enforce QoS parameters and evaluate it enforcing different quality of services.

HiQoS

HiQoS[20] is a multipath SDN and QoS manager, it tries to enforce QoS using the Differentiated Service model and also want to define the routing path between the source and destination.

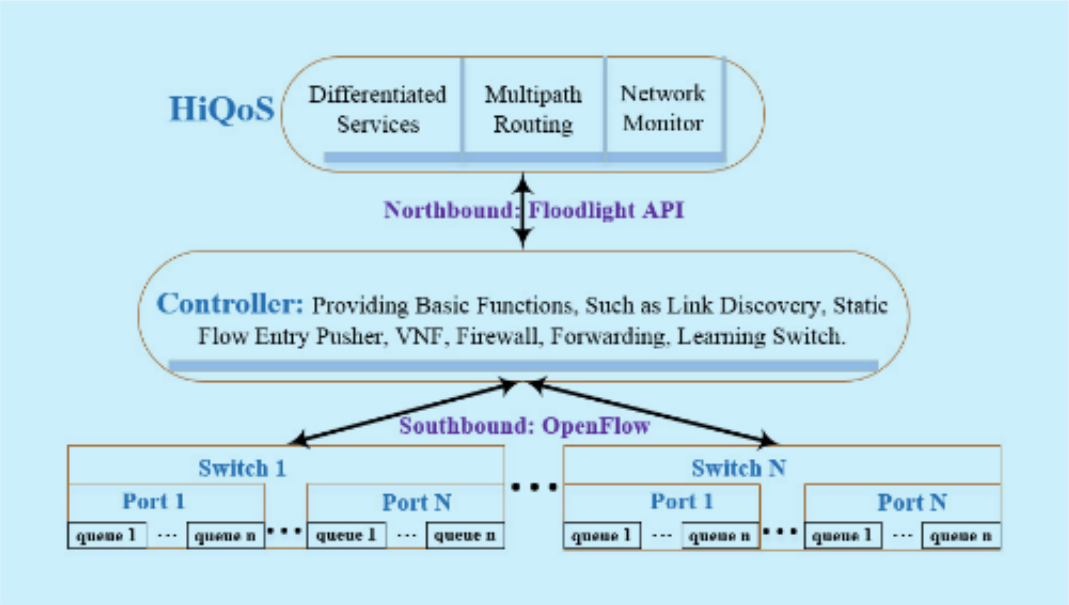

Figure 1.23: HiQos architecture

The architecture show the presence of an SDN controller (Floodlight) which is used to create the path between source and destination and discover the network links in order to provide an updated network topology to HiQos. This application can be divided in two different parts, the Differentiated Service traffic shaper and the Multipath Routing component.

The Differentiate Service traffic shaper uses Q-Ctrl to enforce the QoS parameters on the virtual switch present on the server machine, it defines three different queues on the virtual switch (which is an instance of OVSDB compliant switch, which means that is an Open vSwitch instance). The application define three different static queues, one for each level of QoS, on the virtual switch; when a request for a specific traffic level arrives it will instruct the switch to redirect the flow on a specific queue for that class of traffic.

The traffic shaping is executed by Floodlight and orchestrated by MultiPath

Routing component. The association of a client to a specific type of traffic in this case is based on the source IP address, but the available field in OpenFlow are ToS, MAC address, traffic class header, source port and many more; HiQos will check the source ip address of the request and make the following steps:

1. Calculates the path through the Multipath Routing Component

2. Instructs the SDN Controller to address all the packets from that source trough the calculated path

3. Instrcuts the Controller also to enqueue the traffic on the defined queue on the virtual switch, based on the traffic class (decided by the application)

The path is calculated using a modified Dijkstra Algorithm[2], to calculate multiple paths that could satisfy the QoS requirements; the paths are stored and checked in poll if there are still valid using the SDN Controller. The Controller also enable the full knowledge of the topology to the application, who associates a weight to each node and runs the algorithm to create the path.

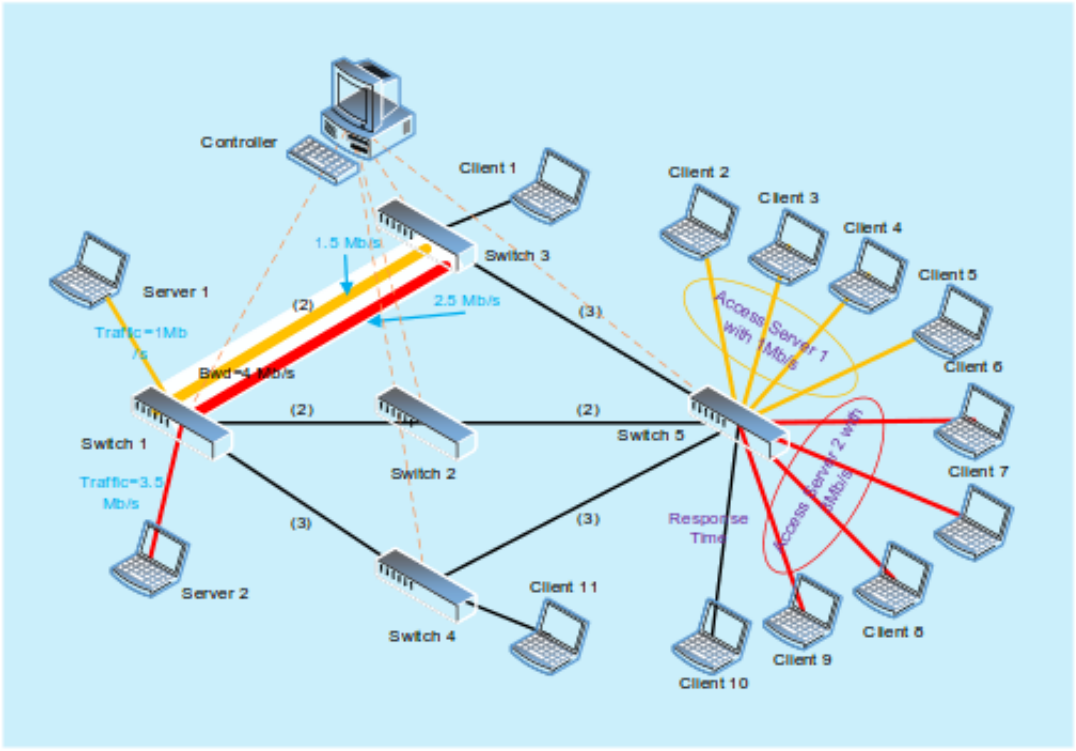

Figure 1.24: The HiQos experimental topology

The figure shows the topology of the experiment, where and which the queues are defined and allocated. Shows also that they allocated a different queue for each port of a switch, but there is also the possibility to filter the traffic (in each different queue for the same port) for a single specific server.

Q-Ctrl

Q-Ctrl[11] is a QoS controller in SDN based Cloud environments, this application receives QoS requests from the clients, schedules the QoS enforcement based on the network topology.

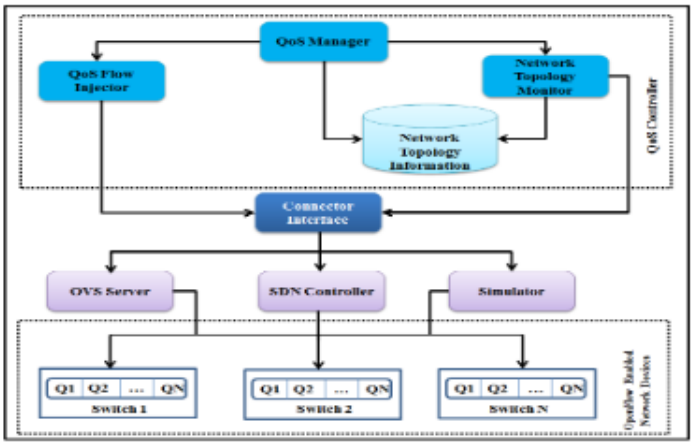

Figure 1.25: Q-Ctrl architecture

The reference architecture (in figure 1.25) shows a Network Topology Monitoring and also a QoS Manager that use both the Connector Interface. The Network Topology Monitoring communicates with the Connector Interface to maintain a reliable and updated network Topology, this component queries in polling the SDN Controller to get the latest changes in the network topology.

The QoS Manager could work in two modes: Direct or Controller. In Direct Mode the manager enforces directly the QoS and Flows on the switches using the OVS Connector which uses the Command Line Interface APIs exposed by Open vSwitch. In the Controller mode instead uses the intermediation of a SDN Controller (as we said Floodlight) and its REST APIs.

Q-Ctrl perform operation to enforce the QoS following its life cycle:

1. *Queue Creation:* the queues are allocated on the virtual switches (also on the physical switches)

2. *QoS Flows Addition:* trough the SDN Controller or directly trough the OVS Connector the flows are allocated on the switches

3. *QoS Flow Modification:* to modify the bandwidth allocation and the links between the vms

4. *QoS Flow Deletion:* when the client requires this operation means that the application that are using a lot of bandwidth is terminated

5. Queue Deletion: after the flow deletion also the queues are deleted from the switch

The SDN Controller is used only to instruct the switches in path definition and also to retrieve a fully updated topology of the network.

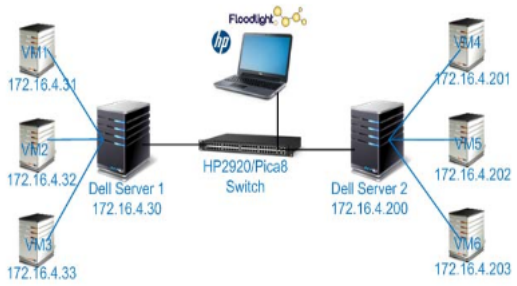

Figure 1.26: The Q-Ctrl experimental scenario

The scenario for the evaluation expose the SDN Controller with an active role in flow definition, also the OVS connector was used to create the queues on the instances of the virtual switch. The network infrastructure is quite simple, so they do not have to define an algorithm to find the path with minimum cost, they just need to control different kind of switches.

OpenQoS

OpenQos is an implementation of a SDN controller which assumes to define only the path with fixed queues for QoS definition. They try to optimize the path between server and client, using Constrained Shortest Path (CSP)

algorithm using as constraints the QoS requirements in terms of jitter and latency applied to a video streaming.
They use as architecture (depicted in figure 1.27) three physical nodes and three switches to calculate the feasible path using the QoS constraints.

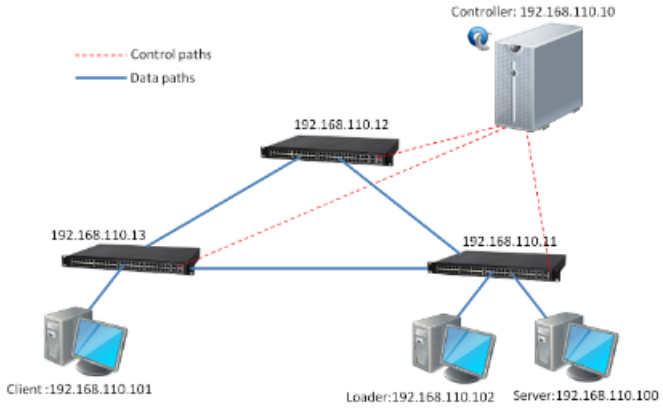

Figure 1.27: The OpenFlow Test network

For the bandwidth constraints they have allocated manually on the physical switches the lower and higher values. To inject the path inside the switches uses Floodlight as SDN Controller, the controller is also used to retrieve informations about the topology, link and bandwidth state.
OpenQoS introduces two components:

- *Route Management:* this module check in poll mode if the topology, link state, available bandwidth are changed and collect it, after the data retrieval it tries to check if there are congestions and if there are find a new path. The link state for the application could be Congested or Non-Congested, the congestion index is calculated using the formula in figure 1.28

$$g_{ij} = \begin{cases} \frac{T_{ij} - 0.7 \times B_{ij}}{T_{ij}}, & 0.7 \times B_{ij} < T_{ij} \\ 0, & 0.7 \times B_{ij} \geq T_{ij} \end{cases}$$

Figure 1.28: The Congestion Formula

If the link results congested they try to reroute the traffic deleting flows and calculating a new path

- *Route Calculation:* is the module which calculates actively the path, it detect if the first packet is a multimedia one and calculate the path based on the type (using the ToS classifier); it calculates two kind of path: the shortest path for non-multimedia packets and the QoS-Optimized path for multimedia packets.

This implementation assumes that the QoS paths are preallocated and dynamically (based on the type of packet) could defines a route; they also could notice if there is a congestion situation and redefine routes also in QoS paths.

CHAPTER 2

DESIGN AND SPECIFICATION

Open Baton is the first real implementation of the ETSI NFV MANO information model, it provides the implementation of almost all functional blocks that are defined in the specification. Open Baton implements each functional block as different module, defines a plug in mechanism and an event dispatching to enable the integration of external systems and is also used for internal module communication.

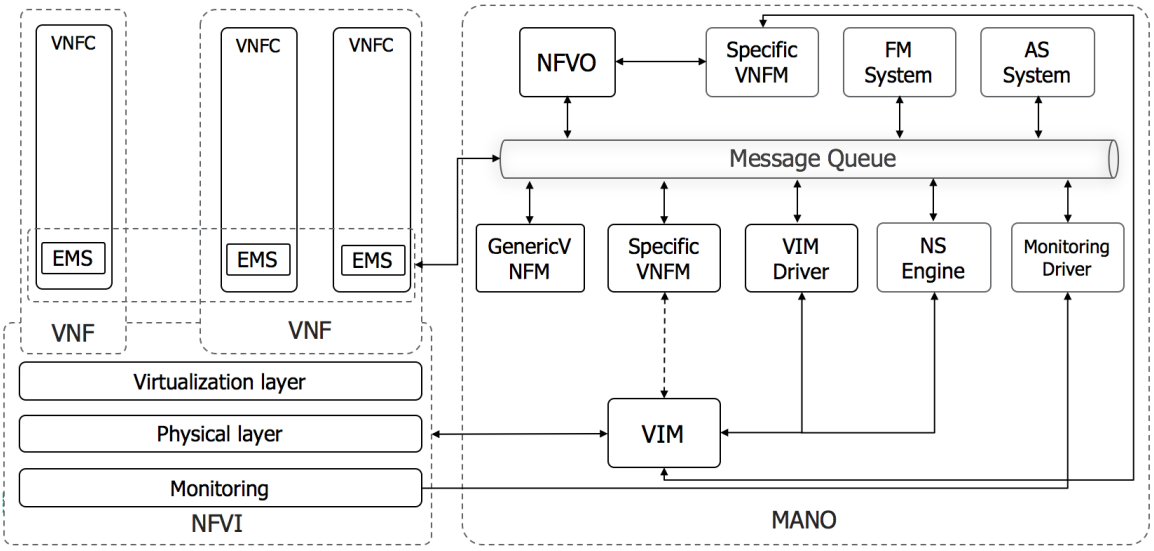

Figure 2.1: The Open Baton internal architecture

The main architecture includes the basics modules which are:

- **NFVO:** is the implementation of the NFV Orchestrator, it defines, manages and starts the provisioning of the resources
- **GenericVNFM:** it represents the implementation of the "default" VNFM, which has the default behavior for the VNF management and can handle different VNFs of the same or different type.
- **Fault Management System:** this module receives and manages with the fault at each level of the NFV architecture, it implements default policies to manage faults which could be extended or redefined
- **Auto Scaling System:** this module enables the VNF autoscaling based on developer defined (or also default) VNF thresholds

- **Network Slicer Engine:** is the module (developed in this thesis) that defines network slices (if required) with requirements specified in the Network Service Descriptor

- **Monitoring Driver:** this functional block provides a communication system with the target monitoring system used on the operator system.

- **EMS:** this block is the in-place component that enables communication with the VNFM to provide allocation on the VNFC instances of the specific functions, provisioning for external resources.

- **VIM Driver:** is the component which interfaces the specific VIM with Open Baton. The VIM Driver expose standard ETSI interfaces to the overlying functional blocks.

The figure shows also other functional blocks, one of them is the Specific VNFM; this manager could be implemented by the vendor of the VNF, it could be implemented using the VNFM SDK provided by Open Baton. The custom VNFM could decide to allocate the resources for target VNF.
The Generic VNFM instead of the custom is the implementation of the ETSI specification, it handles the communication with the NFVO and EMS; to the NFVO requires the resource allocation using the messages GRANT_OPERATION (to check if there are enough resources to allocate the VNF) and ALLOCATE_RESOURCE (to delegate the allocation of the VNF to NFVO), to the EMS send the scripts that it has to run on the VNFC instance to install, configure and start the services on the instance. The communication sequence is depicted in figure 2.2

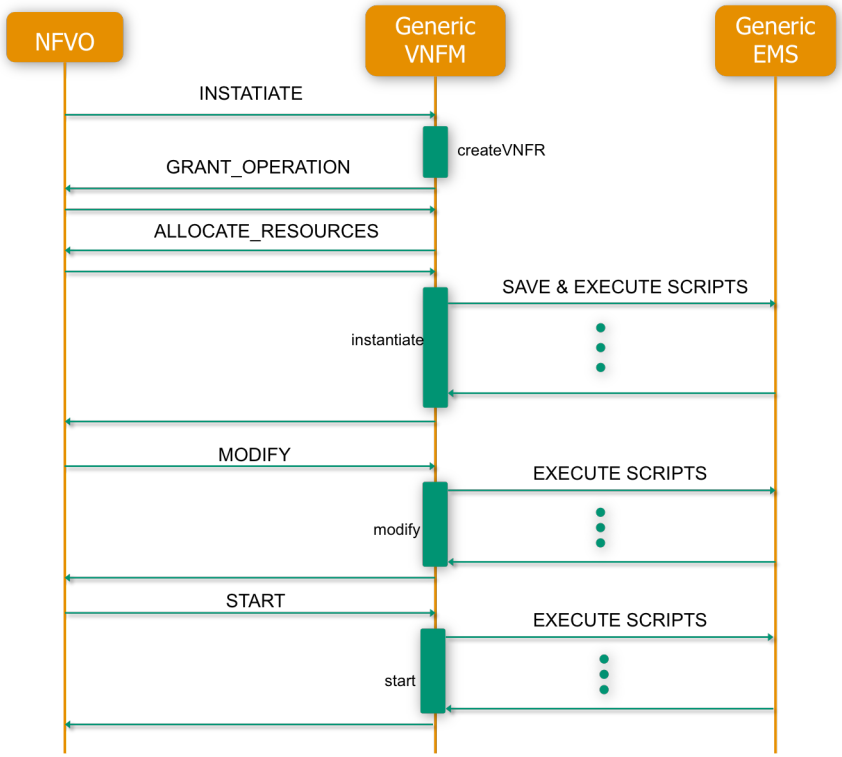

Figure 2.2: The Generic VNFM[18] communication diagram

The Orchestrator is the main component of the Open Baton architecture, it is implemented following the ETSI NFV MANO specification. Beyond the ETSI specification it also provides a GUI where are available all the features to register a PoP, onboard a NSD and launch it, upload VNF packages.
It also exposes an event mechanism to enable the integration of external systems, facilitates the module implementation for events related to the network services allocation done by the NFVO (or an eventually custom VNFM).
The Network Slicer Engine uses this event framework to retrieve informations regard the new network service allocation.

2.0.1 NFVO

The NFVO is the core of the Open Baton architecture, it executes the quota control, resources reservation and instantiation (unless there is another manager which declare itself as resource allocator for VNFs which declares it as manager). It also resolves all the internal dependency (for instantiation), has quota control, define an event mechanism.

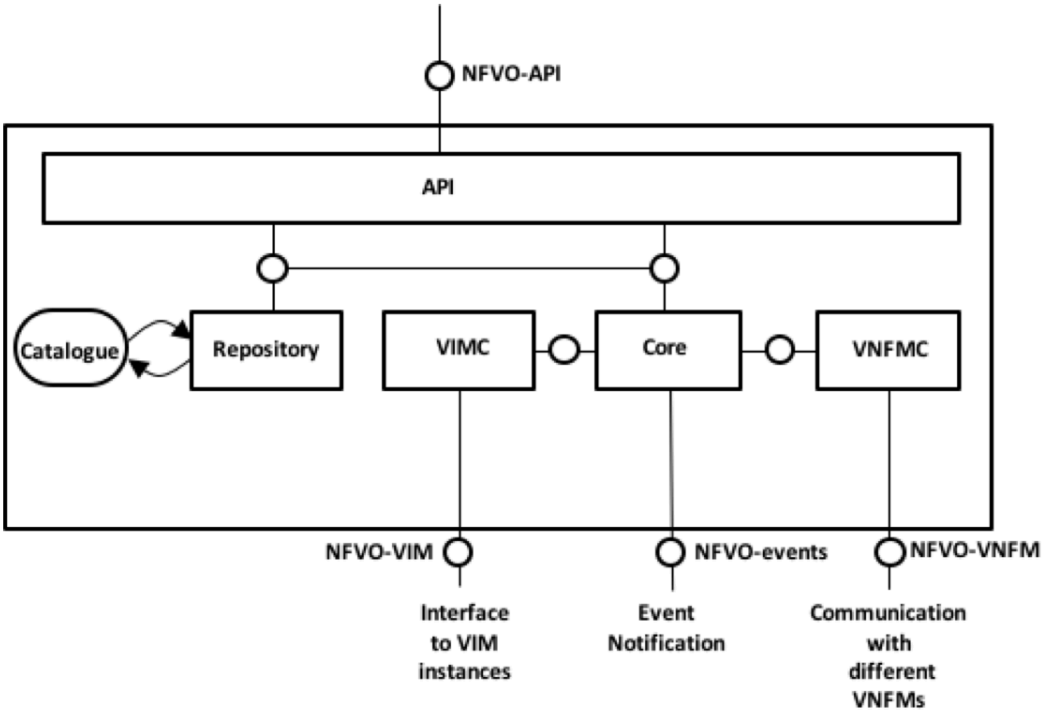

Figure 2.3: The NFVO internal architecture

It is composed by different modules (as depicted in figure 2.3):

- API: represents the northbound interface to the "outside word", enables the onboard of descriptors, instantiation of records (related to

the descriptors already boarded), definition of different PoP and VNF packages following the ETSI[17] definition

- VIMC: this module defines the interaction between the NFVO and the NFVI Point of Presence, for each registered PoP it defines an interaction with it using a driver mechanism based on the type of PoP (for example Openstack, AWS, Google Compute Engine) currently the only PoP supported is Openstack but can be extended defining a new plug in which extends the `VimDriver` class.

- Core: is the central component of the NFVO, coordinates the life cycle events for the Network Service instantiation, manages the catalogue when is requested the instantiation of a descriptor; also the event triggering and dispatching is on behalf of the core module

- VNFMC: provides the interaction with the registered VNFMs using internal interfaces, can use the messaging system or REST for the interaction. It also accomplishes the management of internal state machine for each Network Service Record, based on the communication with VNFM that is instantiating the target record (or a "part" of it).

- Repository: this module defines the persistence, based on the catalogue definition of each elements defined in the ETSI[17] specification

The NFVO also defines four different endpoints (one for northbound and three for southbound respectively) which are used by the different modules for the communication.

- NFVO-API: is the unique northbound endpoint, is responsible for all the communication from outside applications, GUI etc with the API module; the protocol that it uses is only REST over HTTP

- NFVO-VIM: this southbound endpoint is used by the module which defines the interaction with each point of presence, this is achieved using the plug in mechanism which use the messaging system

- NFVO-Events: provides an endpoint for event dispatching, could use the messaging system or REST over HTTP to dispatch the events; the subscription must to be made through the northbound endpoint (NFVO-API)

- NFVO-VNFM: this endpoint communicates with the VNFMs, is used by the VNFMC module, it enables the communication through the messaging system or the REST protocol.

The network service is created as the final result of single VNF instantiation procedures performed by the relative VNFMs. During the instantiation of multiple VNFs, the NFVO resolves the dependencies between VNFs. When a VNFa depends on VNFb, the VNFa is the target and the VNFb is the source.
The messaging system is used to handle the instantiation process (with the correct lifecycle) with the (or multiple) VNFM(s). The VNFM exposes a set of methods that will be executed on the VNF in order to instantiate, update, modify, query the VNF; this methods will change the internal state of the VNF and according to the VNF life cycle possible status (in figure 2.4).

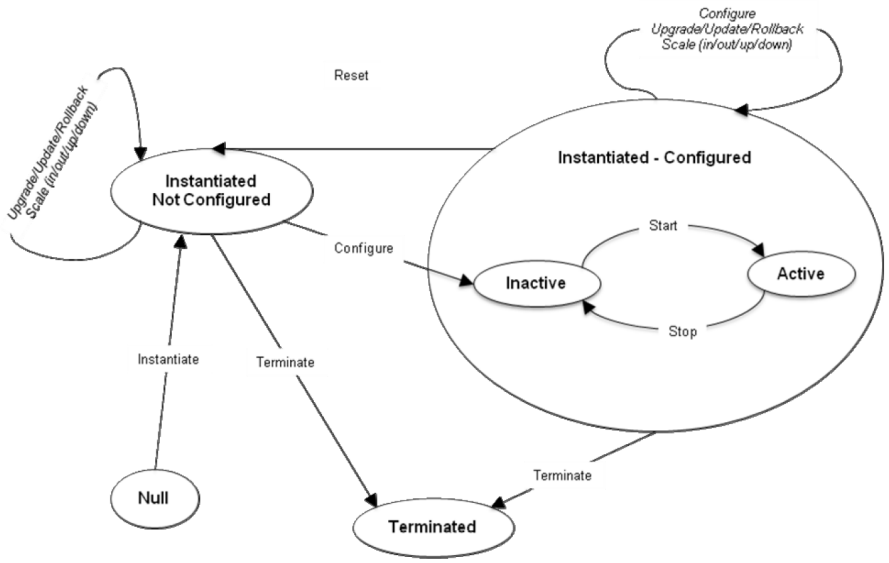

Figure 2.4: VNF Life cycle diagram

Each method of the VNF life cycle management interface change the VNF status. For example, the instantiate method changes the status of the VNF from Null to "Instantiated Not Configured". The method Modify change the status of the VNF from "Instantiated Not Configured" in Inactive.

2.0.2 Generic VNFM

This VNFM is the implementation of the ETSI[17] specification for a generic virtual network function manager, with default behavior regarding the instantiation process and the EMS communication.
Internal architecture is composed by a single module, which uses all the features implemented by the framework. This module communicates with the EMS using the messaging system and with the NFVO also using the already cited system.
It communicates with the NFVO through the NFVO-VNFM endpoint.

2.1 Network Slicer Functional Architecture

The Network Slicer architecture is composed by two different functional blocks, the Network Slicer Engine is the block whic has not requirements in terms deployment location, instead the Connectivity Manager Agent has to be deployed on the controller node of the NFV infrastructure (assuming that lies on Openstack[8] as VIM).

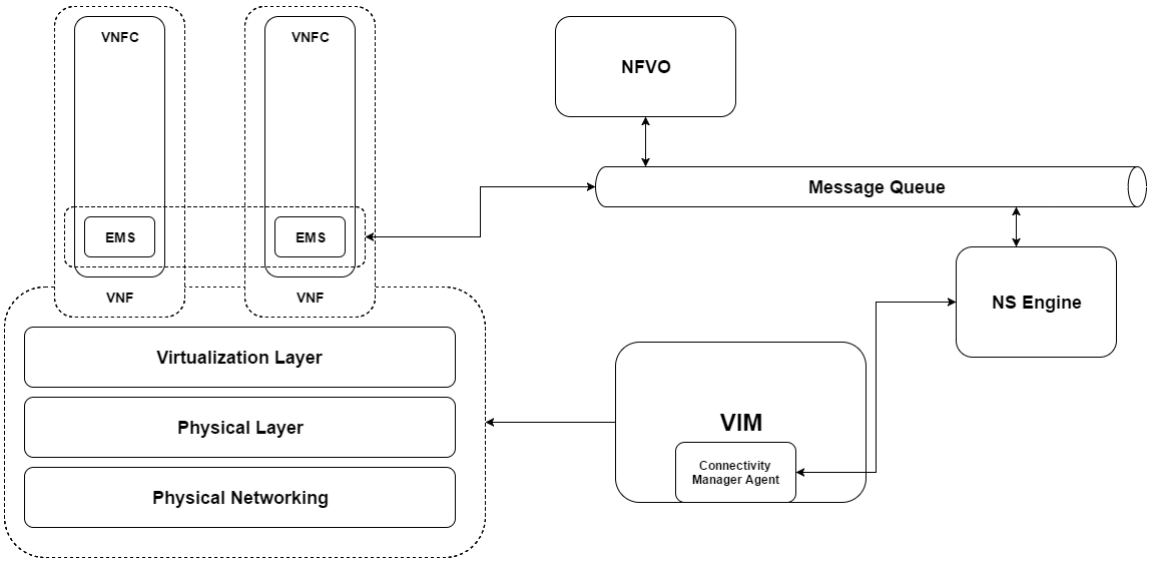

Figure 2.5: The complete architecture

The overall architecture2.5 shows three main components, one of that is the NFVO as event dispatcher, network service manager and all the feature that has already listed; the other two are the Network Slicer Engine and the Connectivity Manager Agent, the first one is in between the orchestrator and the Connectivity manager and has a level of abstraction (intended as more oriented towards the ETSI data representation) higher respect the Connectivity Manager Agent that has purely a platform data model.

Network Slicer Engine

The Network Slicer Engine is composed by two modules, the QoS controller and the SDN driver; this two modules uses a common interface to have a uniform representation of the slice data (as depicted in figure 2.6).

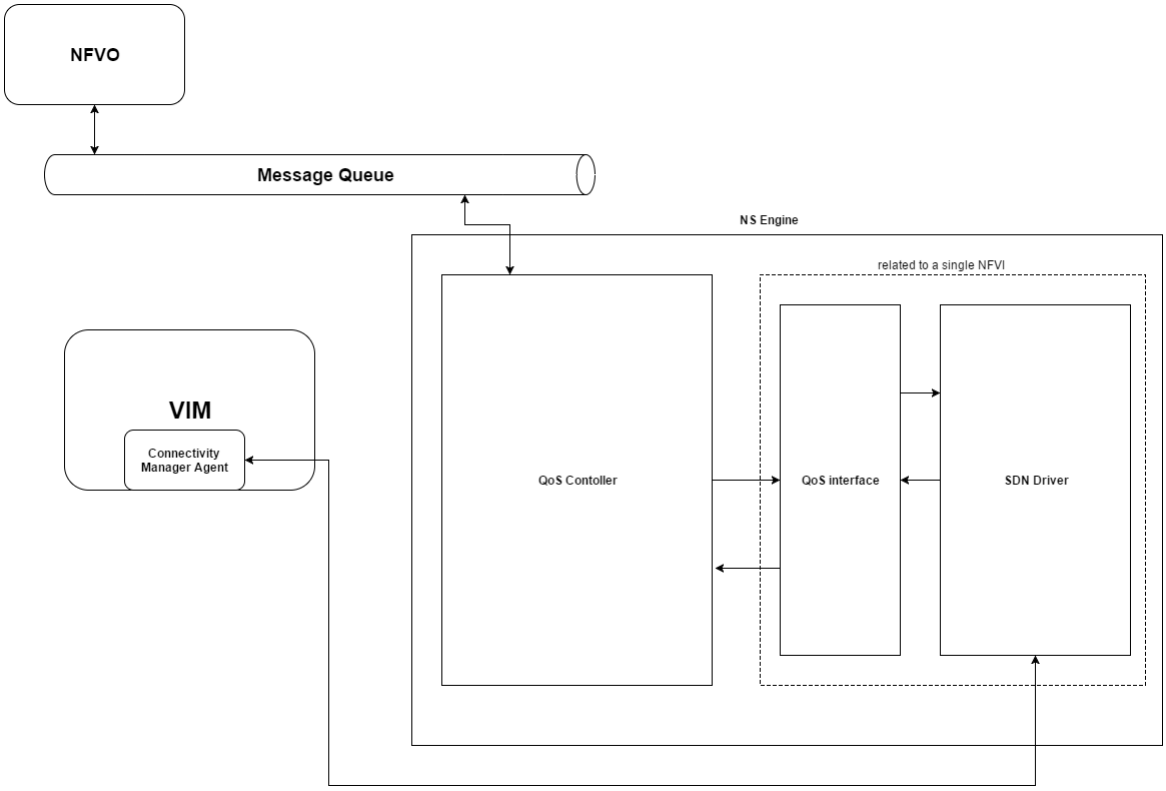

Figure 2.6: The Network Slicer Engine architecture

The figure shows two components (plus an interface) that compose the Network Slicer Engine. The QoS controller is the elements that "talks" with Open Baton on the northbound and with the Connectivity Manager Agent on the southbound.

The QoS controller receives new event from Open Baton, the event is com-

posed:

- *Action:* is the type of received event (could be a distinctive factor if the endpoint of the event is the same for multiple kind of subscription)
- *Payload:* is the subject of the action, for example an INSTANTIATE_FINISH could have as subject a Network Service Record.

The event payload will be parsed to retrieve all information about the physical allocations of each VNFC Instance and the information about the Quality of Service of every virtual link in the Record.
The QoS controller retrieves the VNFC Instance locations to get an overall vision of the network service topology and allocate on for each VNFC at VNFC's point of presence the desired slice characteristics.
The controller has to be aware of the entire network topology of each data center under the control of NFVO and also of the topology that are in between these data centers, in order to define a path involving each VNFC Instance that are involved in the service.
The QoS Interface will be exposed and discussed in the Interfaces Section(2.1.1).
The SDN Driver performs the requests to data center's SDN controller, which could be different for each data center. This driver implements all the logic to interact with the SDN controller, maintain the necessary data (using a database if required) and translate the slice requirements from the data defined in the interface to data type required by the SDN platform.

Connectivity Manager Agent

The Connectivity Manager Agent is our SDN Controller, has to be installed on the controller node of the cloud infrastructure in order to be able to

control the virtual switch and retrieve from the cloud platform the correct informations.

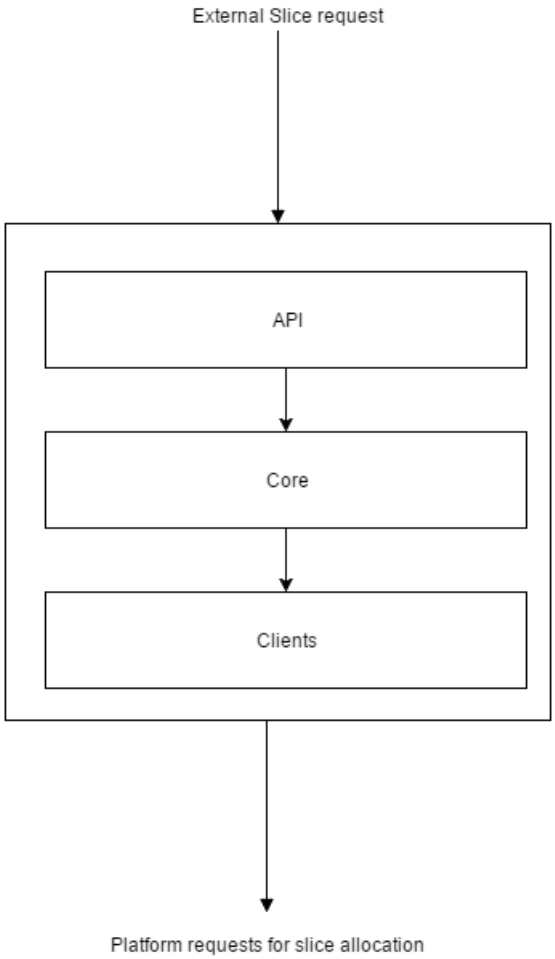

Figure 2.7: The Connectivity Manager Agent functional architecture

This controller is a unique macro-block but internally is organized in three submodules (as depicted in figure 2.7):

- API: this module exposes REST endpoints to the Network Slicer Engine (or whatever application that could be built on top of this component) to instantiate queues, define paths (flows) and retrieve the updated topology of the distribution of virtual resources inside the data center.

- Core: is the main module, it performs the requests to different entities (trough the clients), with the information retrieved sends back the response through API module and/or parse it and find data for other requests.

- Clients: in this module are located the implementation of clients used by the core to interact with the client-specific endpoint

This controller works directly on top of the virtual switches, declaring itself as controller for the flow controller and manager for the queue mechanism; it also interact with the referenced cloud platform using the platform's northbound APIs.

2.1.1 Interfaces

The ETSI specification does not provide any interface regarding the QoS allocation, because it is infrastructure agnostic and also controller agnostic.
To enable a uniform interaction between the QoS controller and multiple instance of SDN Drivers which have different behaviors and data representation, we define a simple interface which expose the two methods to allocate and delete the slice for that datacenter.

```
public interface QoSInterface {

    public boolean addQoS(List<QoSAllocation>
       vnfc_instances,
      FlowAllocation vnfc_flows, String nsrId);
    public boolean removeQoS(List<String>
       vnfc_instances,
```

```
        String nsrID);
}
```

This is interface, as already said, is determined in order to define a northbound for the SDN driver and a common contract on QoS Controller side.

2.2 Integration with Open Baton framework

The Network Slicer Engine is integrated with the Open Baton framework and refers to Open Baton as principal implementation for ETSI NFV specification. It relies on the NFVO-Events southbound endpoint to get events regarding the instantiation of new Network Service Records.

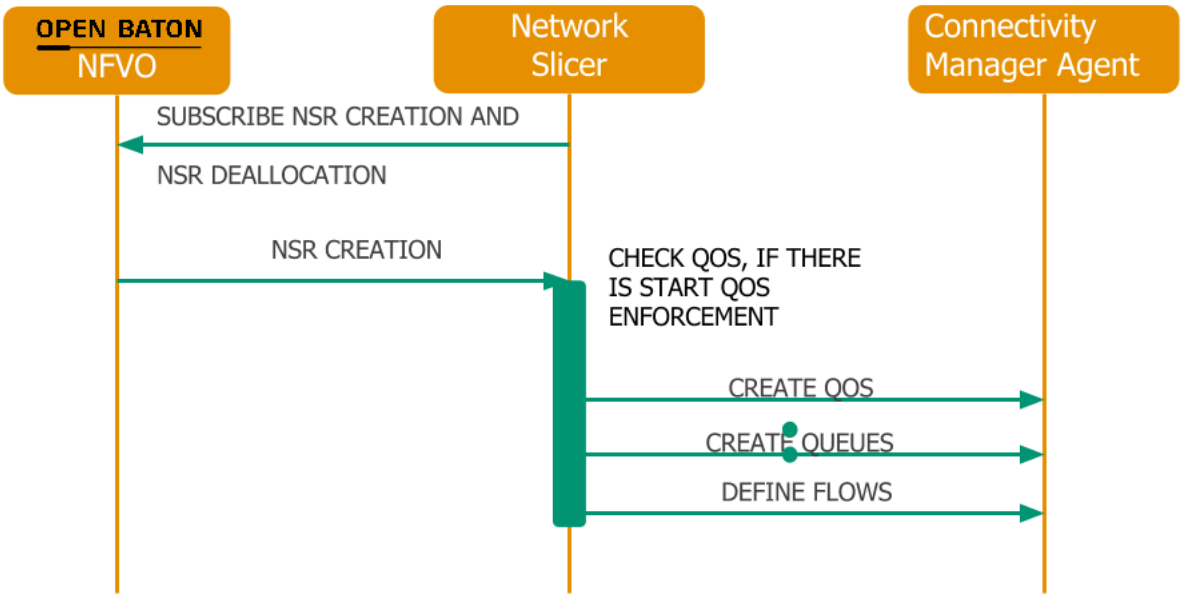

Figure 2.8: New NSR creation sequence diagram

The diagram 2.8 depicts the interaction flow between the Network Slice Engine, the NFVO and Connectivity Manager Agent. The interaction, except for the first one, are always repeated for every Network Service creation or deletion.

The first interaction, made through the Open Baton client sdk provided by the framework, registers the Network Slicer Engine to events of new Network Service instantiation finish and events for the deallocation of Network Services already deployed. The event of completed instantiation of the Network Service is triggered after that all resources are allocated and configured, instead the event deletion is triggered when the requests for the resource deallocation is scheduled.

When the event for new NSR creation is triggered the Network Slicer Engine receives the data through the Open Baton messaging system, de-serializes it and parses the record to check and retrieve (if there are) slice requirements; checks the VNFC instance location, aggregates the data in order to send correct informations to the Connectivity Manager Agent.

After these checks the Network Slicer will retrieve the topology from the Connectivity Manager Agent, "translates" the data from the NFVO data model to the CMA data model and performs request to the northbound endpoint of the Connectivity Manager Agent.

The ETSI specification does not consider any data model for the Quality of Service requirements (formally our slice requirements), in order to have references the Network Slicer maintains a mixed data model in order to have the platform data in relation to the NFVO data model.

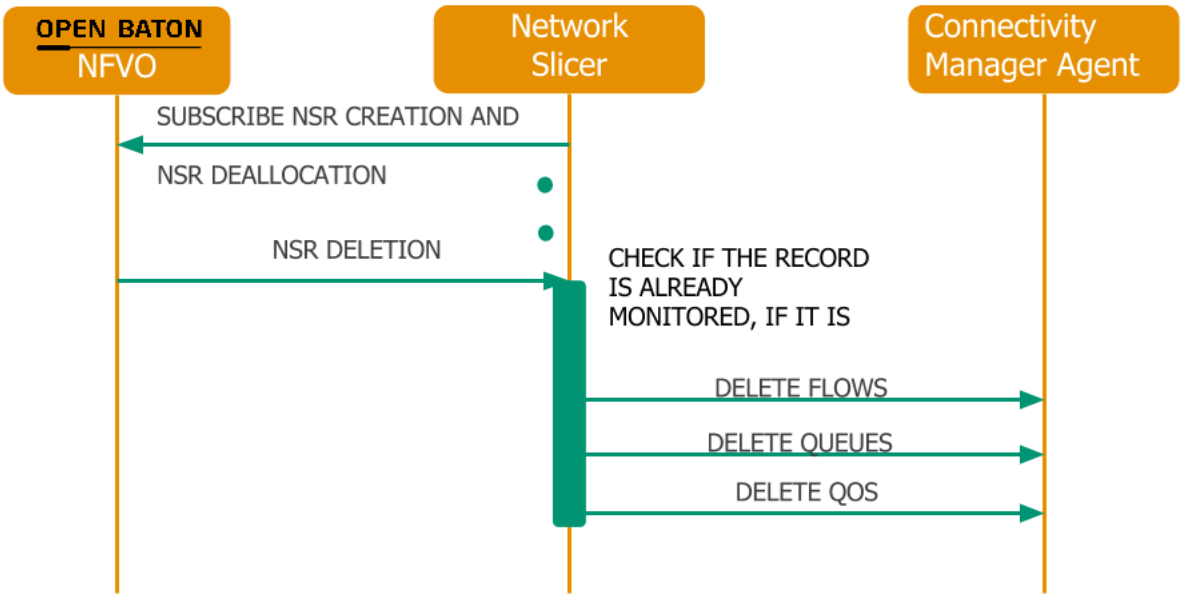

Figure 2.9: NSR deletion sequence diagram

The data model maintained allows a faster check to delete eventually slice requirements already allocated for the Network Service Record.
The Figure2.9 depicts the NSR deletion scenario, the Network Slicer Engine retrieves all the data previously allocated and sends it to the Connectivity Manager Agent to remove the allocated queues and to delete the defined flows for that network service record.

2.3 Network Slicing Policies

The Network Slicing Policies model is inspired to the DiffServ traffic class, the policies are divided in class which has specific slice parameters. The

parameters supported are the maximum and minimum bandwidth for now. We defined three class of Slices, but could extended to the “bare” number as parameters:

- GOLD
- SILVER
- BRONZE

Every element of this class provides a guaranteed bandwidth which will be preserved also in case of network congestion limiting the packet loss.
The values are:

Class	Minimum Bandwidth	Maximum Bandwidth
GOLD	200 Mbit/s	150 Mbit/s
SILVER	100 Mbit/s	50 Mbit/s
BRONZE	50 Mbit/s	25 Mbit/s

CHAPTER 3

IMPLEMENTATION

The technology chosen for the implementation of the Network Slicer is strictly related to the one on which Open Baton is implemented. The Network Slicer Engine is implemented in Java, instead the Connectivity Manager Agent is implemented in Python.

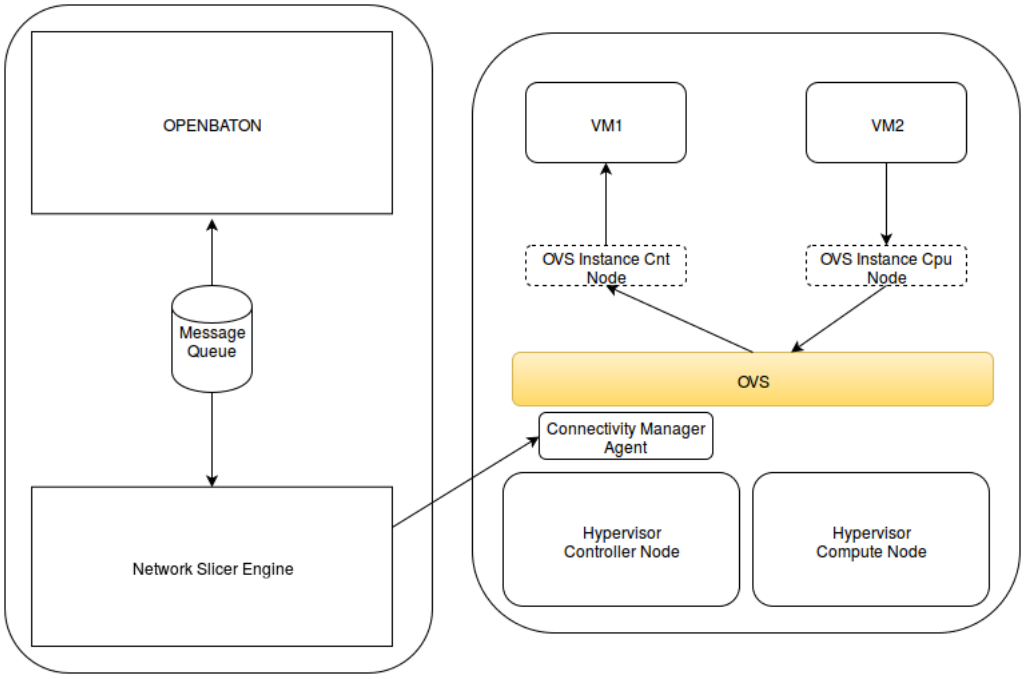

Figure 3.1: The general architecture

The figure 3.1 depicts the overall architecture, which allow us to understand the implementation choices. In this chapter we will explain the implementation of each component.

3.1 Open Baton Implementation

Open Baton is implemented following the ETSI NFV MANO specification [17]. Beyond the ETSI specification it also provides a GUI where are available all the features to register a PoP, onboard a NSD and launch it, upload VNF packages. It defines an event mechanism to enable the integration of external systems, facilitates the module implementation for events related to the network services allocation done by the NFVO (or an eventually custom VNFM). The NS uses this event mechanism to retrieve the instantiation of

new network service record. The implementation architecture of the NFVO is depicted in the Figure 3.2.

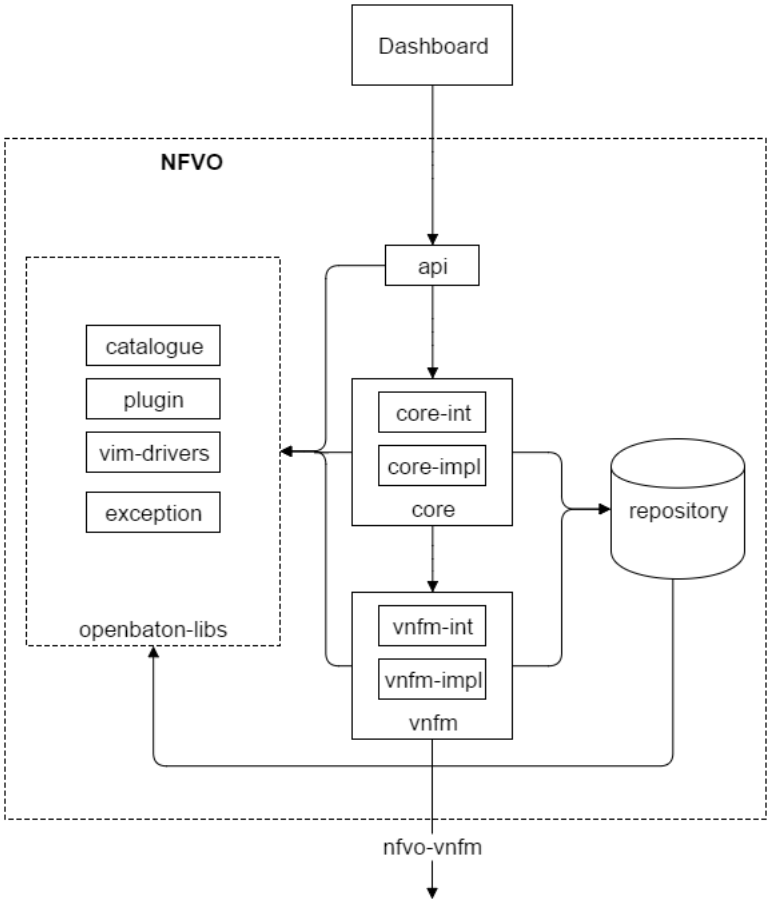

Figure 3.2: The NFVO internal architecture

3.1.1 NFVO Implementation

The NFVO is implemented in Java on top of the Spring framework[6], the implementation relies on the IoC technology available from this framework; the different modules uses also other features:

- The API module uses the Spring RestController in order to expose

REST (relative) paths to developers, the Open Baton client software development kit and the Dashboard.

- Repository module uses the Spring CrudRepository technology to enable the persistence for each entity of the data model, this feature could work with every database that expose the CRUD operations.

- Plug-in system on the NFVO side uses the Rabbitmq Spring library for RPC communications, also the vim-drivers uses this library. It bases the interaction using interface as "contract" to the RPC and uses the message broker as transport to deliver the request.

- NFVO-Events endpoint uses the Rabbitmq Spring library for "normal" Rabbitmq communication and also (as we said in the architecture) RestTemplate which is the REST client library in spring. This mechanism will be explained in section 3.1.2

- The NFVO-VNFM uses aso the AMQP library and/or the RestTemplate REST client for the communication with VNFM(s); this interaction is regulated and defined in the ETSI specification[17]

- The Command Line Interfaces uses a the Spring-Shell project and also some functionalities exposed by the plug in mechanism to define commands like `InstallPlugin` (and its dual `UninstallPlugin`) and `listPlugins`.

The plugins are normal java application that uses the Java AMQP protocol library in order to become the endpoint for the RPC mechanism of the plugin system.

3.1.2 NFVO-Event

The NFVO-Event endpoint enables the dispatching of events related to network service record life cycle, when the network service record pass trough one state an event was triggered and dispatched to all the subscripted applications. The available action for the subscription are:

- GRANT_OPERATION
- ALLOCATE_RESOURCES
- SCALE_IN
- SCALE_OUT
- SCALING
- ERROR
- RELEASE_RESOURCES
- INSTANTIATE
- MODIFY
- HEAL
- UPDATEVNFR
- SCALED
- RELEASE_RESOURCES_FINISH
- INSTANTIATE_FINISH
- CONFIGURE

- STAR

The event subscription has also other parameters more than the simple action, in order to increase (or decrease if not settled) the coarseness of subscription; could be enabled a specific "interest" for a network service record setting the `networkServiceId` parameter, or more specific for a VNFR setting the `virtualNetworkFunctionId` parameter.
In order to dispatch the event an endpoint has to be settled in the subscription declaration with the relative endpoint type, the endpoint types available are:

- RABBIT
- REST

The endpoint itself is a URL or queue name, depending on the endpoint type in the subscription, NFVO will send the event to that specific endpoint using the specified endpoint type.
The event subscription are exposed through the NFVO-API endpoint, an external application could subscribe itself to different actions for each network service record (higher granularity) or multiple actions without any network service record specified (lower granularity) which means receive events with that specified action(s) for each network service record.

3.1.3 SDKs

The Open Baton framework defines three different software development kit in order to achieve openness and extendability, they encapsulate the communication (and relative logic) with one of the NFVO endpoints:

- **vnfm-sdk(-amqp):** this sdk defines the interaction with NFVO-vnfm endpoint, it encapsulates all the logic to communicate with the NFVO for network service record definition, instantiation and life cycle functions (for example GRANT_OPERATION and ALLOCATE_RESOURCE from the VNFM to NFVO, or QueryVNF from the NFVO to VNFM); it uses the Spring library for AMQP to provide this functionalities

- **plugin-sdk:** defines the contract-based interaction with the defined plugin, it encapsulates the Rabbitmq RPC mechanism enabling the NFVO (or other components that uses this system) to call the methods defined on the interface and transparently invokes them on the remote plugin

- **nfvo-sdk (known also as OpenBaton client):** is the sdk which encapsulates the nfvo-api endpoint, it uses the UniRest library to define the interaction with NFVO REST APIs, uses also the Gson library to define a "pojo vision" of json messages used to send requests as body and responses. The figure 3.3 depicts the UML diagram of this sdk.

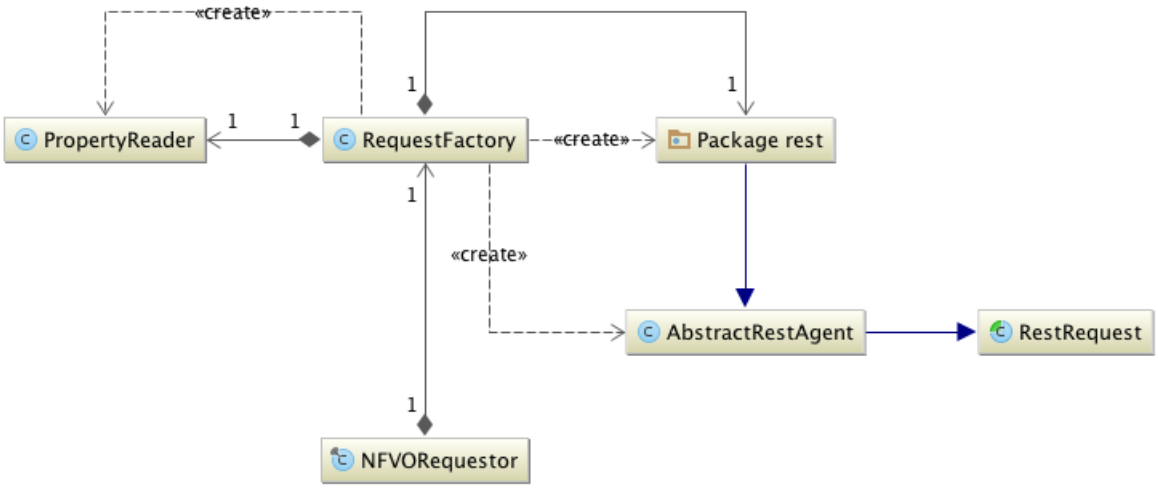

Figure 3.3: The NFVO sdk documentation[4]

It could be also used as "entry point" for the event mechanism, defining subscription to events specifying endpoints and other parameters

Each sdk could be imported into the application logic as dependency through the Open Baton maven repository, in order to enable the user-defined plugin, vnfm or other application on top of the NFVO to interact with it in agile way.

3.2 Network Slicer Engine Implementation

The Network Slicer Engine is also implemented in Java using the already cited Spring framework[**spring**]. It uses the NFVO-Event endpoint to retrieve new instantiation or deallocation of network service record, defines an internal threading model in order to have an efficient request dispatching. The Figure 3.4 depicts the internal component architecture.

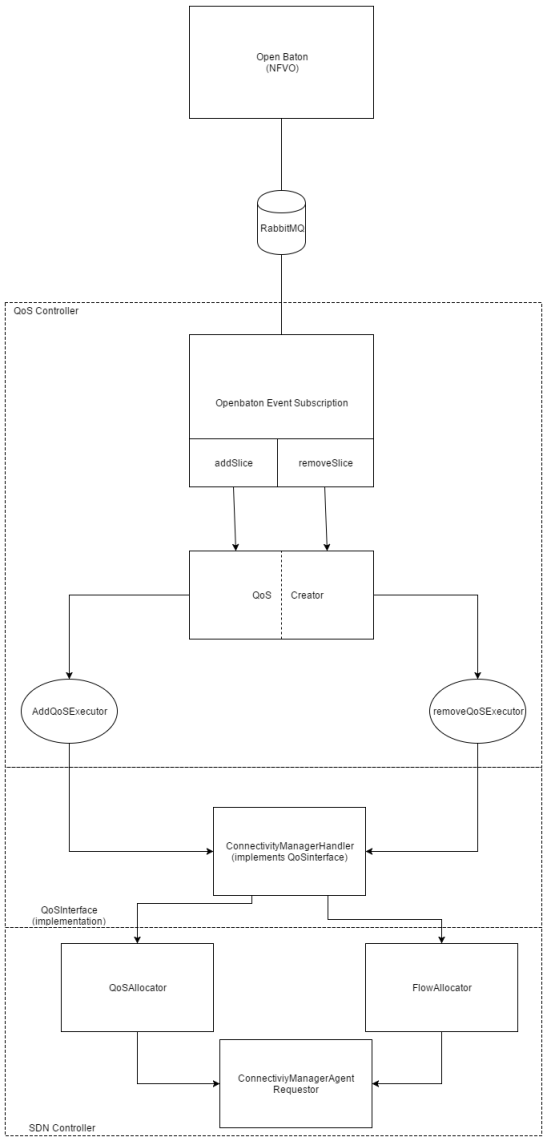

Figure 3.4: The Network Slicer Engine internal architecture

The Network slicer engine defines a northbound endpoint where retrieves the events from the NFVO and a southbound endpoint which communicates with the Connectivity Manager Agent.

3.2.1 Northbound endpoint

The northbound is defined using a single Spring bean, this endpoint receives event from the NFVO, the event endpoint chosen is the message system; this is implemented using the Spring AMQP library to define the interaction with the message broker.

The entities necessary to queue instantiation on the message broker are defined in a Configuration[3] bean, which defines two queues for the two different event subscriptions; it also defines all the "protocol objects" in order to have an interaction ready infrastructure at bootstrap, including the callback methods when a new message is received.

The endpoint is represented by a Spring bean which implements the previously defined callback methods.

```
@Service
public class OpenbatonEventSubscription {
...
  public void receiveNewNsr(String message) {
    OpenbatonEvent evt;

    try {
        logger.debug("Trying to deserialize it");
        evt = mapper.fromJson(message,
           OpenbatonEvent.class);
    } catch (JsonParseException e) {
      ...
    }

    NetworkServiceRecord nsr = evt.getPayload();
```

```
    logger.debug("NSR is " + nsr.toString());

    vnfrloop:
    for (VirtualNetworkFunctionRecord vnfr :
       nsr.getVnfr()) {
        logger.debug("VNFR: " + vnfr.toString());
        for (InternalVirtualLink vlr :
           vnfr.getVirtual_link()) {
            logger.debug("VLR: " + vlr.toString());
            if (!vlr.getQos().isEmpty()) {
                for (String qosAttr : vlr.getQos()) {
                    logger.debug("QoS Attribute: " +
                       qosAttr);
                    if
                       (qosAttr.contains("minimum_bandwith"))
                       {
                       ...
                        creator.addQos(nsr.getVnfr(),
                           nsr.getId());
                        break vnfrloop;
                    }
                }
            }
        }
    }
    logger.info("[OPENBATON-EVENT-SUBSCRIPTION] Ended
       message callback function at " + new
       Date().getTime());
}
```

```
...
}
```

Listing 3.1: The nsr creation callback methods implementation

The listing 3.1 shows the source code of the method which parses the received Network Service Record (after the appropriate control on the received message) and checks in every virtual network function record if there are Quality of Services requirements. If there are, the bean will demand to the QoSAllocator the allocation of the required resources to define a network slice with selected requirements.

```
@Service
public class OpenbatonEventSubscription {
  ...
  public void deleteNsr(String message){

      OpenbatonEvent evt;

      try {
        logger.debug("Trying to deserialize it");
        evt = mapper.fromJson(message,
           OpenbatonEvent.class);
      } catch (JsonParseException e) {
        //exception catch
      }



      NetworkServiceRecord nsr = evt.getPayload();
```

```
        vnfrloop:
        for (VirtualNetworkFunctionRecord vnfr :
           nsr.getVnfr()) {
          logger.debug("VNFR: " + vnfr.toString());
            for (InternalVirtualLink vlr :
               vnfr.getVirtual_link()) {
              logger.debug("VLR: " + vlr.toString());
                if (!vlr.getQos().isEmpty()) {
                  for (String qosAttr : vlr.getQos()) {
                      if
                         (qosAttr.contains("minimum_bandwith"))
                         {
                          creator.removeQos(nsr.getVnfr(),
                             nsr.getId());
                            break vnfrloop;
                        }
                    }
                }
            }
        }
    }
    ...
}
```

Listing 3.2: The callback function to delete a network service record

The above listing 3.2 depicts the callback method defined to handle a network service record deallocation, it always search if there are network slicing requirements; confirmed the presence of slice features it delegates the QoSAllocator to start the deallocation process.

The QoSAllocator is a Spring bean which schedules thread in order to support multiple allocation (or deallocation or both) requests at the same time. The Java task scheduler mechanism is used in order to define thread scheduling, the scheduling policies are defined in different ways for the allocation and deallocation operation. The policy for allocation process is defined using a little delay, it becomes necessary because the messaging system (used for event dispatching) has a time out for the callback method; after exceeding the time out the message will be redelivered and could flood the component, so I decided to schedule a thread using a "fire-and-forget" model in order to give back the control to the caller bean and finish the method.
The policy for deallocation is completely different (uses always the fire-and-forget model), uses a higher delay modeled on the event characteristics. The RELEASE_RESOURCE_FINISH (is the Action for event deallocation of a network service record) event is triggered when the resource termination is scheduled on the VIM side and the QoS mechanism could not be removed until the virtual resources are not terminated (see the Connectivity Manager Agent section 3.3.1); after some tests we found that the correct delay is approximately ten seconds.

3.2.2 Network Slicer Engine threads

Internally, as already said (see section 3.2.1), the Network Slicer Engine uses threads, in order to avoid a possible flooding from the messaging system and enable the multiple request at the same time; two types of thread have been defined, one for allocation and one for deallocation. Both of them use the defined QoSInterface (see section 2.1.1), in order to have a contract-based interaction with plugin (that for our case is the Connectivity Manager Agent plugin).

The thread defined for allocation of QoS resources (`AddQoSExecutor`) retrieves and "groups" data from the network service record.

```
public class AddQoSExecutor implements Runnable{

  //constructor and internal data

  @Override
  public void run() {
    ...
    FlowAllocation flows = this.getSFlows(vnfrs);
    List<QoSAllocation> qoses = this.getQoses(vnfrs);
    boolean response =
       connectivityManagerHandler.addQoS(qoses,flows,nsrID);
    ...
  }

  private List<QoSAllocation>
     getQoses(Set<VirtualNetworkFunctionRecord> vnfrs) {
    //internal logic to group data for QoS parameter
       injection
  }

  private boolean hasQoS(List<VldQuality> qualities,
     Set<VNFDConnectionPoint> ifaces, String vnfrId){
    //check if the connection point is related to a
       virtual link with QoS requirements
  }
```

```
  private FlowAllocation
     getSFlows(Set<VirtualNetworkFunctionRecord> vnfrs){
    //this method group data for flow allocation
  }

  private List<VldQuality>
     getVlrs(Set<VirtualNetworkFunctionRecord> vnfrs) {
    //method scope: find the
  }

  private List<FlowReference>
     findCprFromVirtualLink(Set<VirtualNetworkFunctionRecord>
     vnfrs, String vlr){
    //method logic to find the connection points from
       virtual link names
  }
}
```

Listing 3.3: The methods of the AddQoSExecutor thread

The listing 3.3 depicts the thread code, who exposes methods for parsing the network service descriptor and defines the appropriate data structures, based on the Open Baton data model which are described in the QoSInterface definition. After the appropriate "grouping", it uses the already mentioned interface to delegate the SDN Controller plugin of the Network Slice allocation through the SDN Controller.

```
public class RemoveQoSExecutor implements Runnable{

  //Constructor and internal data
```

```
  @Override
  public void run() {
    List<String> servers =
       this.getServersWithQoS(vnfrs);
    boolean response =
       connectivityManagerHandler.removeQoS(servers,nsrID);
  }

  private  List<String>
     getServersWithQoS(Set<VirtualNetworkFunctionRecord>
     vnfrs){
    //this method retrieves all the name of the
       instances with QoS
  }

  private Map<String,Quality>
     getVlrs(Set<VirtualNetworkFunctionRecord> vnfrs) {
    //this method retrieves the quality from each
       virtual link (in each vnfr)
  }

  private Quality mapValueQuality(String value){
    //method which get the enumerative (with relative
       values) from a string
  }
}
```

Listing 3.4: The methods of RemoveQoSExecutor

The code of RemoveQoSExecutor (listing 3.4) shows an easier data aggregation, instead of the one performed by the AddQoSExecutor, and delegates directly to the SDN Controller plugin passing only the server list (which is the host names of each single VNFC Instance with QoS defined)

3.2.3 The QoSInterface implementation

The ETSI NFV MANO specification did not defined a common interface, which could enable a uniform communication between the NFVO (or an integrated module) and the SDN Controller platforms; indeed there are many SDN Controllers with different APIs and data representation so define an interface could be a good compromise to enable a level of abstraction closer to the specification and also maintain the extendability.
The implementation of this interface (defined in the architecture section 2.1.1) is the `ConnectivityManagerHandler`, it was implemented using the Spring bean technology.
Whenever a new network record is instantiated, the handler requests an updated topology in order to get an overall vision of the deployment of each VNFC Instance inside the data center in terms of node location for each deployed virtual machine; after it gets an updated topology it calls two different beans in order to define an appropriate data structure for queues allocation and flow definition on the controller.

```
@Service
@Scope ("prototype")
public class ConnectivityManagerHandler implements
   QoSInterface{
```

```
    //PostConstruct initializator and internal data


    public boolean addQoS(List<QoSAllocation> queues,
        FlowAllocation flows, String nsrId){
      this.updateHost(); //retrieves the updated
         topology...
      List<Server> servers =
         queueHandler.createQueues(hostMap, queues);
      internalData.put(nsrId,servers);
      flowsHandler.createFlows(hostMap,servers,flows);

      return true;
    }

    private void updateHost() {
      this.hostMap = requestor.getHost();
    }

    public boolean removeQoS(List<String> servers,String
        nsrID){

      List<Server> serversList;
      queueHandler.removeQos(hostMap,serversList,servers);
      flowsHandler.removeFlows(hostMap,servers,internalData.get(nsrID));
      internalData.remove(nsrID);
      return true;
    }
}
```

Listing 3.5: The ConnectivityManagerHandler implementation

The above listing 3.5 shows the implementation of methods defined in the QoSInterface and also the method which retrieves the topology from Connectivity Manager Agent (using defined Requestor, see the southbound section 3.2.4).
The `internalData` is a data structure which has the network service record id as "key" and all the platform-related information retrieved from the SDN Controller (Connectivity Manager Agent).

3.2.4 Network Slicer Engine southbound

The engine define also a southbound in order to communicate with SDN Controller, this endpoint is composed by the `QoSHandler` and `FlowHandler` which use another bean `ConnectivityManagerRequestor` to perform requests to the controller.
The `QoSHandler` uses the topology to create the appropriates data structures in order to define the queues on virtual switch instances, resolves the slicing policies defined in the network service record and extracted from the northbound of the application.

```
@Service
@Scope ("prototype")
public class QoSHandler {

  //internal data and initialization method
  public List<Server> createQueues(Host hostMap,
     List<QoSAllocation> queues, String nsrId){
```

```
    List<ServerQoS> queuesReq = new ArrayList<>();
    List<Server> servers = new ArrayList<>();

    for(QoSAllocation allocation : queues){

      String serverName = allocation.getServerName();
      String hypervisor = hostMap.belongsTo(serverName);
      Server serverData =
         requestor.getServerData(hypervisor,serverName);
      servers.add(serverData);
      ServerQoS serverQoS =
         this.compileServerRequest(serverData,allocation.getIfaces(),hyperv
      queuesReq.add(serverQoS);
    }

    QosAdd add = new QosAdd(queuesReq);
    add = requestor.setQoS(add);

    servers = this.updateServers(servers, add);
    return servers;
  }

  private List<Server> updateServers(List<Server>
     servers, QosAdd add) {
    //update the data structure defined for the request
  }

  private ServerQoS compileServerRequest(Server
     serverData, List<QoSReference> ifaces, String
```

```
      hypervisor) {
    //prepare the request body using the defined pojo
  }

  private InterfaceQoS addQuality(InterfaceQoS
     serverIface, Quality quality) {
    //add the slice requirements in terms of bandwith
  }

  public void removeQos(Host hostMap, List<Server>
     servers, List<String> serverIds, String nsrId){
    for (Server server :servers){
      if (serverIds.contains(server.getName())){
      String hypervisor =
         hostMap.belongsTo(server.getName());
        for (InterfaceQoS iface :
           server.getInterfaces()){
          Qos ifaceQoS = iface.getQos();
          requestor.delQos(hypervisor,
            ifaceQoS.getQos_uuid());
        }
      }
    }
  }
}
```

Listing 3.6: QoSHandler source code

Listing 3.6 shows the internal definition of methods to add and remove queues, based on the slicing policy (Quality enumerative); the add has more

complex data aggregation instead the delete which relies on the internal data. The `createQueues` method transform the data from ETSI data model to Connectivity Manager Agent data model.
The `FlowHandler` instead performs a data aggregation based on relations between the VNFC Instances, a relation is intended as two virtual network function component instance which have a connection point which referencing the same virtual link.

```
@Service
@Scope ("prototype")
public class FlowHandler {

  //Internal data and initialization method

  public void createFlows(Host host, List<Server>
     servers, FlowAllocation allocations, String nsrId){
    //business logic to define relations
  }

  public void removeFlows(Host hostmap, List<String>
     serversIds, List<Server> servers, String nsrId){
    //flow deletion based on source
    //address and destination address
    //retrieved from internal data

  }

  private Server getServerRefFromIp (List<Server>
```

```
      servers, String ip){
    for (Server server : servers){
      if(server.getFromIp(ip) != null){
        return server;
      }
    }

  return null;
  }
}
```

Listing 3.7: ìFlowHandler source code

The `createFlow` method defines the relation definition based on virtual link identification reference, the ETSI specification does not specify relation between VNF in terms of QoS; the possibility of defining relations is exposed in the Virtual Network Function Forwarding Graph but is more oriented on Service Function Chaining.
The `ConnectivityManagerRequestor` maps all the available requests to the SDN Controller (Connectivity Manager Agent), uses the Spring RestTemplate library in order to have a more integrated client with all the features exposed by the Spring framework REST client.

3.3 Connectivity Manager Agent

The Connectivity Manager Agent (friendly CMA), is our SDN Controller which runs on the controller node of the cloud platform infrastructure. This component is written in Python on top of the Bottle.py[7] framework, exposes REST APIs in order to enable decoupling for interaction without using a

specific programming language.
The controller uses in-place virtual switches to enable the QoS allocation and, using also the switches, defines a path between the host and destination enabling the enqueue mechanism; this strategy enable the injection of quality of service inside the data center.

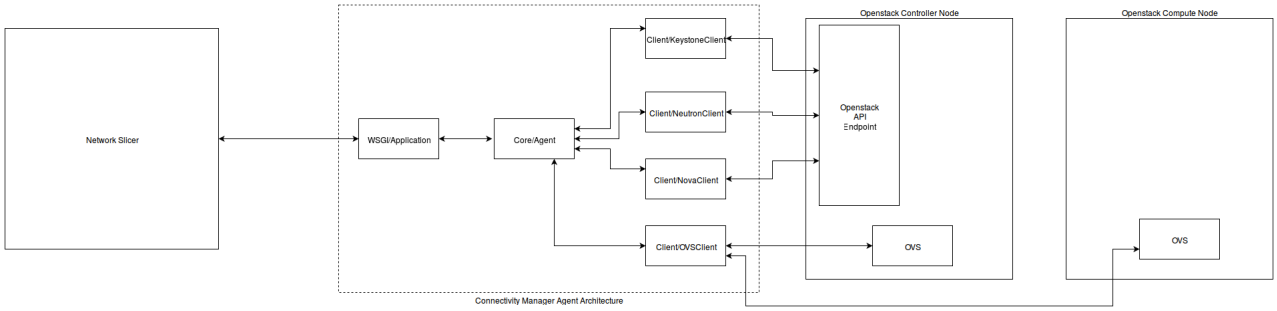

Figure 3.5: Connectivity Manager Agent software architecture

The software architecture (depicted in figure 3.5) could be divided in three different submodules, which is a common practice in SDN controller development; the modules are:

- Northbound (API): this module enables the interaction from other applications, in particular the Network Slicing Engine

- Core: receives the request from northbound and uses the clients (southbound) to retrieve all the required data and allocates the slice requirements

- Clients (Southbound): this module is composed by the clients for interaction with Openstack components and Open vSwtich

3.3.1 Connectivity Manager Agent Northbound

The northbound is composed by a single class which define all the REST path and map each request to a method exposed by core module.

```
class Application:
  def __init__(self, host, port):
  self._host = host
  self._port = port
  self._app = Bottle()
  self._route()
  self._debug = True
  self.agent = CMAgent()

  def _route(self):
  # Welcome Screen
  self._app.route('/', method="GET",
     callback=self._welcome)

  # Hypervisor methods
  self._app.route('/hosts', method="GET",
     callback=self._hosts_list)

  self._app.route('/server/<hypervisor_name>/<server_name>',
     method="GET", callback=self.get_server_info)

  # QoS methods
  self._app.route('/qoses', method=["POST", "OPTIONS"],
     callback=self._qoses_set)
```

```
# QoS methods
self._app.route('/qoses/<hypervisor_hostname>/<qos_id>',
    method=["DELETE", "OPTIONS"],
    callback=self._delete_qos)

# QoS methods
self._app.route('/queue/<hypervisor_name>/<queue_id>/<queue_number>/<qos_
    method=["DELETE", "OPTIONS"],
    callback=self._delete_queue)

self._app.route('/queue', method=["POST","OPTIONS"],
    callback=self.add_queue_to_qos)

# Flow methods
self._app.route('/flow', method=["POST", "OPTIONS"],
    callback=self._assign_flow_to_queue)

self._app.route('/flow/<hypervisor_name>/<flow_protocol>/<flow_ip>',
    method=["DELETE","OPTIONS"],
    callback=self._delete_flow)

...
```

Listing 3.8: Defined REST paths

The listing 3.8 shows the different path exposed by the application northbound, from the url to retrieve the updated topology (`/hosts`) passing through the path to retrieve single data for a VNFC Instance (which is a virtual machine on Openstack) to the path for QoS, single queue and flow allocation.

3.3.2 CMA Core

The Core of Connectivity Manager Agent receives requests for QoS allocation and Flow definition from the northbound, it uses the Client module in order to retrieve data for each virtual machine (involved in the network service record) and define QoS, queues and flows.
The module use Openstack clients to retrieve each relevant data in order to interact with the virtual switch instance correctly, this clients are used only to retrieve data (and perform authentication).
This layer is composed by the Agent only which define three different classes:

- Host: represent the single node of the cloud platform, defines all the methods which involves a single entity of the infrastructure
- Cloud: is the class that interacts with the Openstack REST APIs in order to get the overall topology, has a wide vision of the infrastructure
- Agent: this class defines all the methods callable from the northbound, it uses the Cloud and the Host class to achieve the slice allocation

The listing 3.9 exposes the methods defined in the Agent class, they uses the already cited class to achieve functionalities.

```
class Agent(object):
  def __init__(self):
    self.cloud = Cloud()

  def get_hypervisor_map(self):
    //retrieve the overall topology, with the
       allocation of all the vms running in that
       datacenter
```

```
def set_qos(self, qos_args):
  //method to add qos to an nsr (called from the
     northbound)

def add_new_queue(self,qos_json):
  //method to add a queue on an existing QoS


def destroy_qos(self,hypervisor_hostname,qos_id):
  //removes the qos allocated for all the elements
     for a specific hypervisor

def
   destroy_queue(self,hypervisor_name,queue_id,queue_number,
   qos_id):
  //remove a single queue allocated to a particular

def set_flow(self, flow_args):
  //define the flows for the network service record

def
   _remove_flow(self,hypervisor_name,flow_protocol,flow_ip):
  //remove the defined flows for a single virtual
     machine

def
   get_new_server_info(self,hypervisor_name,server_name):
  //retrieve the informations for a single virtual
```

```
machine
```

Listing 3.9: Agent source code

The Cloud class, as already said, represents the overall infrastructure, it uses the Openstack clients in order to:

- Retrieve the topology: location of vm in terms of hypervisor (node) on which is instantiated.
- Retrieve hypervisors info: in terms of computational resources available.
- Get all the informations about network: interaction with Neutron to get the necessary information which are used by the Host class

In particular the Cloud class interacts with Neutron and Nova (using the respective clients) to retrieve also server info on "Openstack side" and also gets the necessary data from the networking agent to enable data retrieval for the virtual switch.
The Host class uses all the data passed from the "upper" class and interact with the OVS client to define the flow and QoS (and queues). This class has also a reference to Cloud because the "data extraction" has to involve the Neutron client, managed by Cloud class, to retrieve the correct port which corresponds to the ip assigned to the virtual machine. This class implements also the algorithm to properly delete QoS and flows from the virtual switch, because OVS has a hierarchy as depicted in figure 3.6

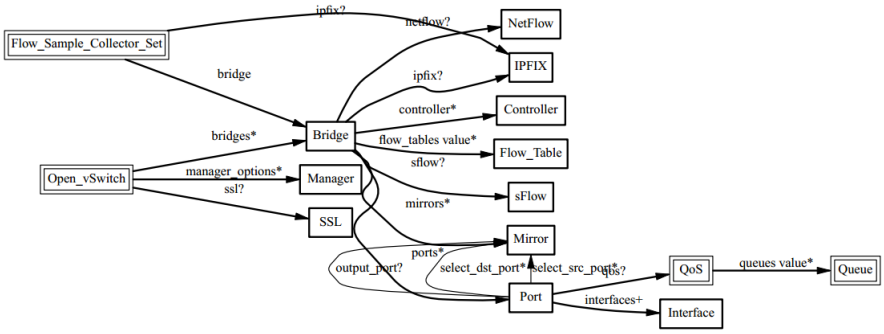

Figure 3.6: The OVSDB schema

To respect this hierarchy, assuming that the port was already deleted by Openstack because is the normal behavior, it deletes the QoS and after the queues; after proceeds to delete flows. This mechanism was implemented per virtual machine in order to support an eventually scaling of the network service record.

3.3.3 Clients

The clients could be divided in two different types:

- Openstack clients: are the simple python libraries exposed by the Openstack framework in order to enable a REST "wrapped" interaction with the platform
- OVS client: this is written from scratch, it enables the interaction with Open vSwitch regarding the queue creation and flow definition

The Openstack clients used, as already said, are three: the Nova client to retrieve all the information regarding virtual machines, Neutron client to retrieve data about the port name, ip and hypervisor datas, the Keystone client to get tokens for authorization and authentication.

The OVS client defines and interaction with Open vSwitch, using the Python `subprocess` library in order to use the CLI entry point offered by the switch platform. The listing 3.10 depicts some methods exposed to define all the structures necessary.

```
class Client(object):
  ...
  def create_queue(self, hypervisor_ip, min_rate,
     max_rate):
    //create a queue on Open vSwitch

  def
     set_qos_ovs(self,hypervisor_ip,qos_id,queue_id,queue_number):
    //link the queue to already defined QoS

  def list_queues(self, hypervisor_ip):
    //list all the queues in a determined node
       (hypervisor)

  def create_qos(self, hypervisor_ip, queue_string,
     queues):
    //create a QoS id in the Open vSwithc database and
       link directly to it one or more queues

  def del_queue(self, hypervisor_ip,
     qos_id,queue_id,queue_number):
    //delete a queue

  def del_qos(self, hypervisor_ip, qos_id):
    //delete a qos entry
```

```
...
def add_flow_to_queue(self, hypervisor_ip, src_ip,
   destination_ip, protocol, priority,
   ovs_port_number, queue_port):
  //flow definition with enqueue action

def
   remove_flow_dest(self,hypervisor_ip,destination_ip,
   protocol):
  //flow deallocation with a specific destination ip

def remove_flow_src(self,hypervisor_ip,src_ip,
   protocol):
  //flow deallocation with a specific source ip
```

Listing 3.10: Methods for create a queue, QoS and flows

This client exposes also all the methods mandatory to properly delete QoS and Flow respecting the OVSDB integrity, otherwise an exception will be raised.

CHAPTER 4

VALIDATION AND EVALUATION

Our component was validated using the Iperf[16] scenario to check if the bandwidth limitations (upper and lower bound) are respected.
The measurements were achieved using Zabbix[19] to get the real time monitoring of bandwidth throughput from the client to server. Other measurements were made to check the system performances in order to get an average response time when a new INSTANTIATE_FINISH event is triggered.

4.1 Tools

The tools used to evaluate the Network Slicer (plus the Connectivity Manager Agent) are:

- Zabbix: in order to monitor the throughput in terms of bandwidth
- Iperf: in order to use all the bandwidth available on the testbed
- Generic VNFM: in order to require the instantiation of

- at: in order to schedule the start of iperf scenario to allocate the QoS infrastructure before

Now we will analyze the tools used in order to "locate" them in the scenario and define each considered use case in order to get the overall vision of what each component do.

4.1.1 Zabbix

Zabbix is one of the most used platform for real time monitoring of virtual and physical machines, this system could monitor a huge number of hosts with different metrics for each host.
Zabbix uses a manager-agent architecture which could be "expanded" to a manager-proxy-agent infrastructure, in order to monitor hosts in separate networks (for example in different tenants or locations).

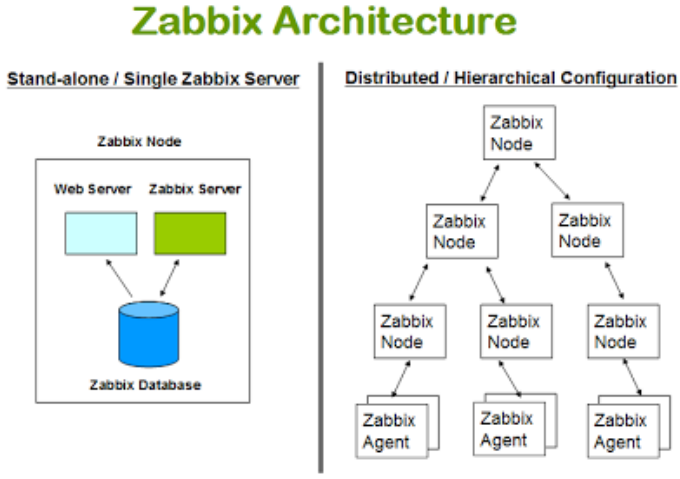

Figure 4.1: Zabbix Architecture

The figure 4.1 depicts the two different architectures Zabbix, for the evaluation I chose the simple one because our different services has to lies on the

same network in order to evidence the behavior of Network Slicer in presence of different services.

Zabbix enables the simultaneous monitoring of multiple metrics for various components, such as cpu, memory, network interfaces, hard drives; we were interested in all metrics regarding network interfaces such as throughput, packet loss and real time values.

4.1.2 Iperf

Iperf is tool for active measurements of bandwidth in IP networks, it could use different type of L4[10] protocols (such as TCP, UDP) on both versions of L3[10] protocol (IP v4 and v6).

This software uses all the bandwidth available in order to retrieve measurements of average bandwidth used, packet loss, latency and many more (as depicted in figure 4.2).

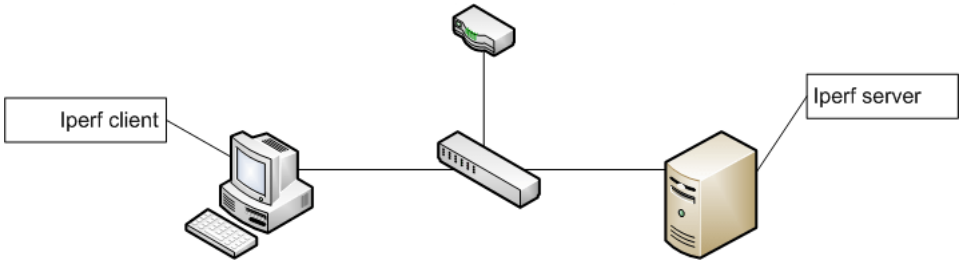

Figure 4.2: Iperf scenario

This software is used in traditional networks to measure the network performance in order to verify that all the links are perfectly functional and active, is used also to verify quality of services queues and flow definition in combination with network performances.

Iperf has a lot of options, to retrieve metrics and performances also increase the measurement time; sometimes it is used to test the overall throughput of

a system placing the Iperf client as “input” for the platform and Iperf server to the other “end”.

4.1.3 Generic VNFM

The Generic VNFM is also implemented in Java on top of the Spring framework[**spring**], as already explained (see section 2.0.2) its architecture is composed by only one module.
The VNFM used the vnfm-sdk to define the interaction with the NFVO, uses the Rabbitmq library for Spring to send the script to EMS (in order to execute all the steps for each life cycle event) and also to retrieve the results of execution (including errors).
It implementation architecture is defined in only one Spring bean that implements all the logic, because it has a “passive” behavior (as specified in the specification[17]) regarding the instantiation of the resources and the quota managing (granting operations).
I used the Generic VNFM as manager to instantiate the target Iperf scenario and handle the life cycle of each Network Service Record instantiated for our test.

4.2 Scenarios

We have used two scenarios to make the system measurements. First scenario was defined to retrieve result about the bandwidth limitation (upper bound for the queue) and system response time, the second test tries to achieve the validation of guaranteed bandwidth and also evaluate the system response time.

4.2.1 One Network Service Record

The first scenario considers the allocation and deallocation of one network service descriptor with one iperf client and one iperf server with `GOLD` quality of service in order to verify the slice "ceiling". The Network Service Descriptor (in the listing 4.1) underline the slice requirements at virtual link level, we have also defined a dependency between the two vnf in order to obtain the instantiation of the iperf server before the iperf client.

```
{
  "name":"iperf-NS",
  ...
  "vnfd":[
    {
      "name":"iperf-server-gold",
      ...
            "connection_point":[
              {
                "virtual_link_reference":"private"
              }
        ...
    "virtual_link":[
      {
        "name":"private",
        "qos":[
          "minimum_bandwith:GOLD"
        ]
      }
    ...
```

```
    {
      "name":"iperf-client-gold",
      ...
            "connection_point":[
              {
                "virtual_link_reference":"private"
              }
      ...
      "virtual_link":[
        {
          "name":"private",
          "qos":[
            "minimum_bandwith:GOLD"
          ]
        }
  ...
}
```

Listing 4.1: The Network Service Descriptor used for this test

In order to have an accurate evaluation of the response time for allocation and deallocation we ran this test two times. The figure 4.3 depicts the response time after the reception of INSTANTIATE_FINISH event till the allocation of network slice.

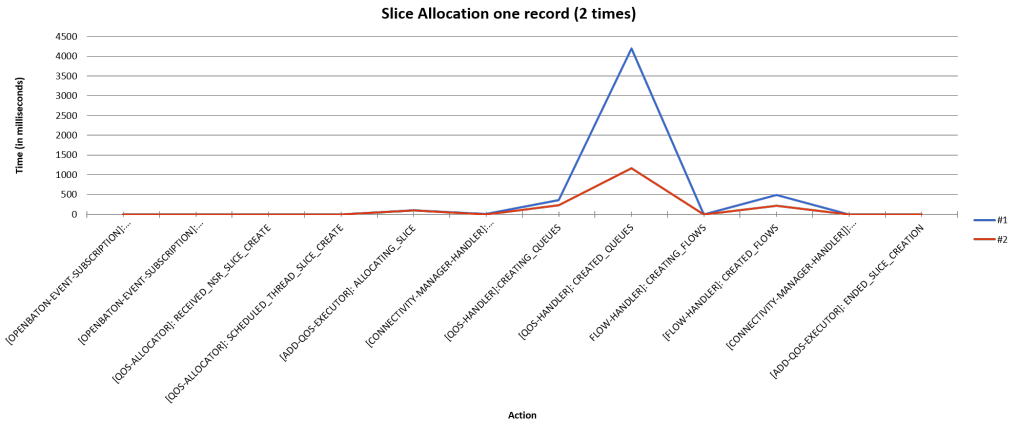

Figure 4.3: Results of allocation 1 network service record (for two times)

The figure 4.4 depicts the deallocation of already allocated network service record, the "peak" in terms of time response is the already discussed (see section 3.2.2) delay to remove the queues and flow respecting the integrity of Open vSwitch database.

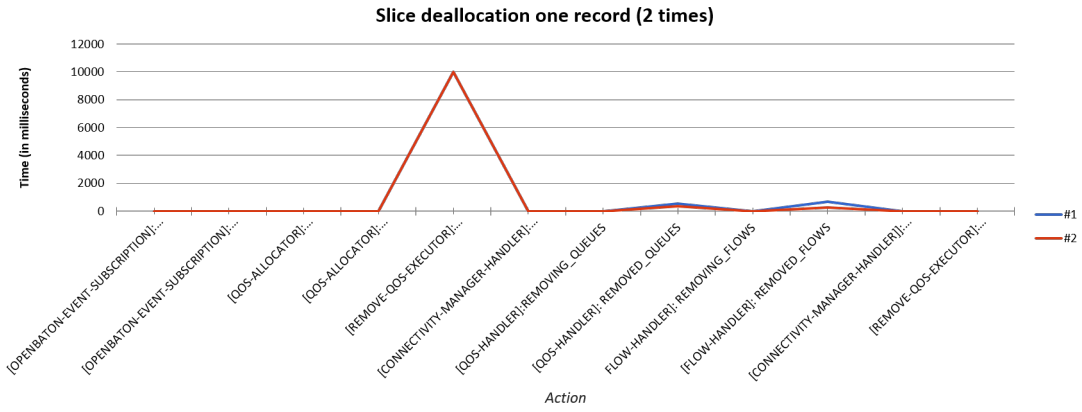

Figure 4.4: Result of deallocation 1 network service record (for two times)

We also followed (as shown in figure 4.5) the traffic flow through Zabbix[19] in order to get the real time throughput of the Iperf client.

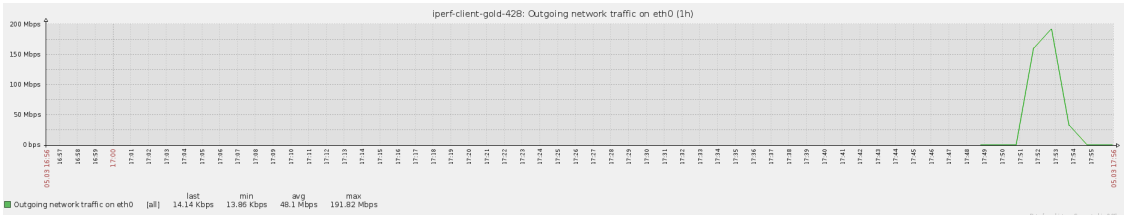

Figure 4.5: The real time throughput of the Iperf Client

4.2.2 Two Network Service Records

This test was repeated two times, one time with two network service descriptor with different QoS requirements (GOLD the first one and SILVER the second one) and one time with a descriptor with QoS requirements (GOLD) and one with best effort quality.

Two Network Service Records with Different QoS

The first test was ran in order to check if different slice requirements are applied on the same switch (always checking the response time) with the specified values (for this test the SILVER requirements have been defined to 5 Mbit/s of guaranteed bandwidth and 10 Mbit/s of maximum in order to emphasize the bandwidth difference).
The figure 4.6 shows the real time throughput of the iperf-client-gold which highlights that the minimum bandwidth is always guaranteed from the Open vSwitch instance.

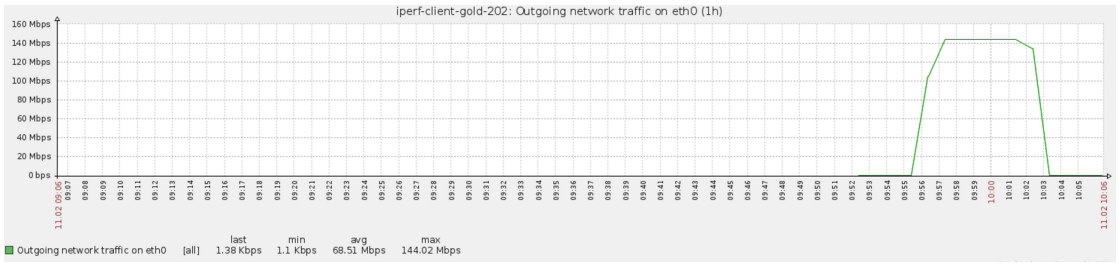

Figure 4.6: The Iperf Client Gold output

Also the iperf-client-silver zabbix graph (in figure 4.7) shows that the minimum bandwidth is always guaranteed also for different slice requirements.

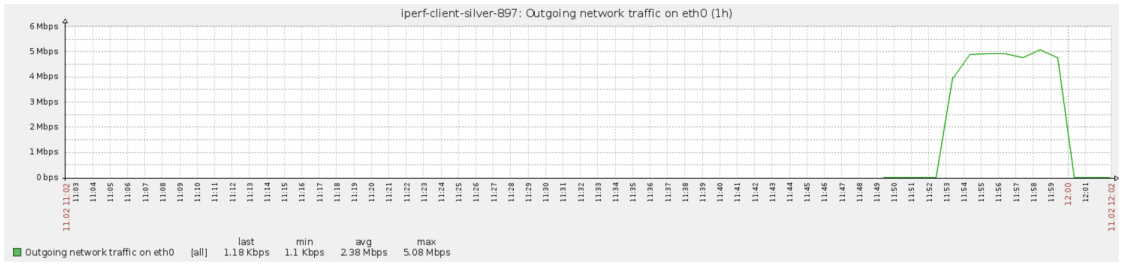

Figure 4.7: The Iperf Client Silver output

The records allocation is initiated with a difference of less than a second (from the NFVO Dashboard) but the event dispatching depends also on the instantiation time (from the Generic VNFM, which depends also from network latency in order to download all the packets specified in the script installation) so the event start highlighted in figure 4.8 are different.

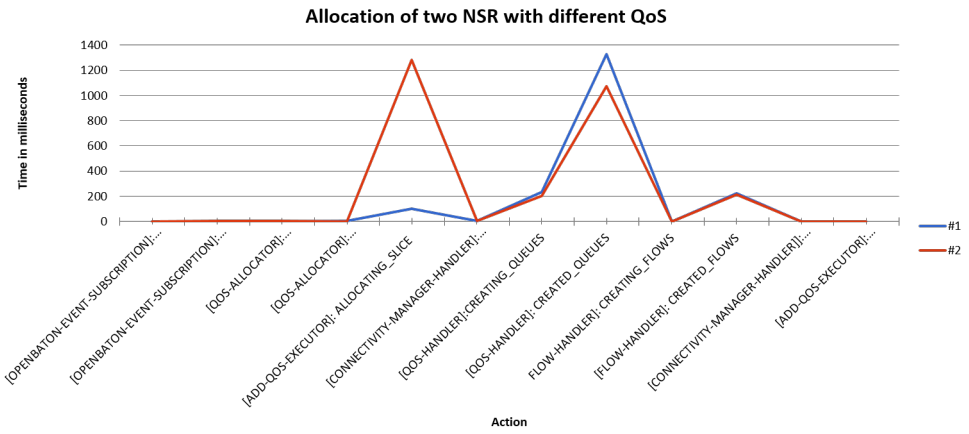

Figure 4.8: Allocation of two network service records with different QoS

The deallocation event instead is dispatched as soon as the VIM driver "confirm" the scheduled deletion of the virtual resources, as expected behavior of the Generic VNFM, meaning that the two events are dispatched with some milliseconds of delay; the figure 4.9 depicts the event deallocation of the two network service record from the point of view of the Network Slicer Engine.

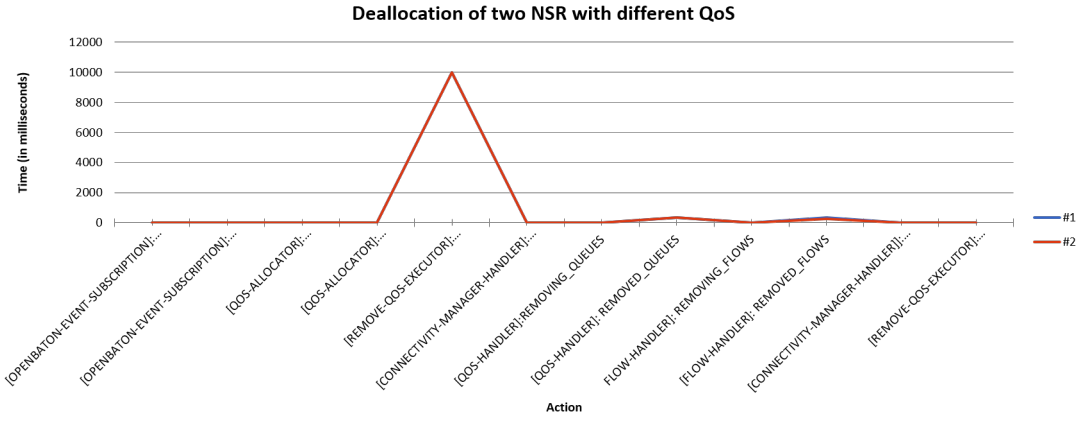

Figure 4.9: Deallocation of two network service records with different QoS

This test highlights that queue definition on OVS are respected and the guaranteed bandwidth is allocated to the VNFC Instances (the Iperf Server and Iperf Client).

Two Network Service Records one with QoS

The objective of this test is to demonstrate that our infrastructure respects the guaranteed bandwidth, also in presence of another service (which insists on the same network). We defined two different network service descriptor, one with QoS requirements (policy `GOLD`) and one without any QoS policy; this test leverages on the characteristics of Iperf suite which uses all available bandwidth in order to retrieve network performance. After instantiation the expected result was a measured bandwidth of at least 150 Mbit/s for the network service record with GOLD policy.
The figure 4.10 depicts the real time monitoring of the best effort client used to flood the network in order to check the "performance" of infrastructure.

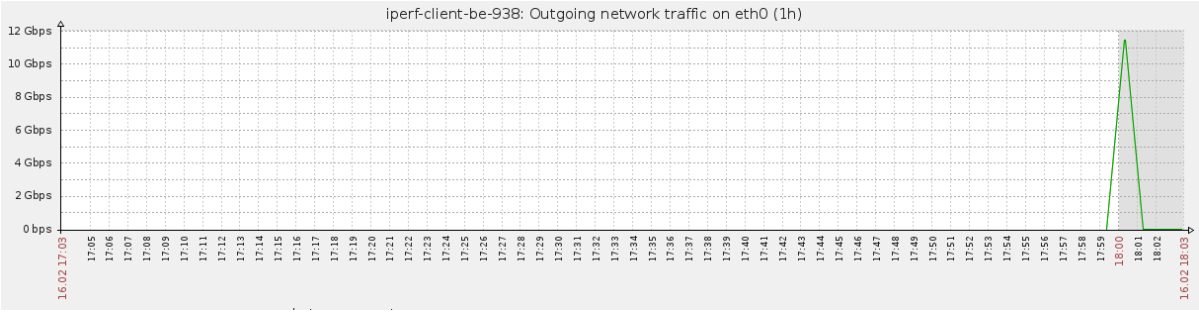

Figure 4.10: The best effort client bandwidth measurements

The figure 4.11 instead shows the real time evolution of bandwidth used by the client with `GOLD` policy allocated, which respects the defined "rules" and does not convey the flooding.

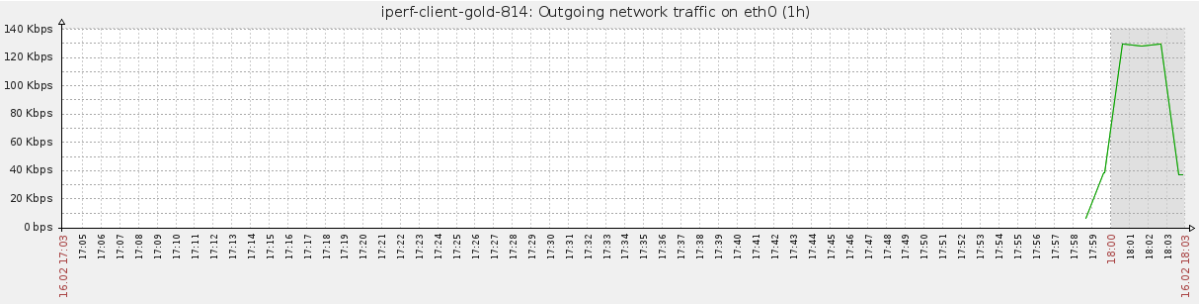

Figure 4.11: The gold client bandwidth measurements

The response time of the entire system corresponds to the response time of the allocation of one nsr, because when the best-effort network service record INSTANTIATE_FINISH event was dispatched to the Network Slicer Engine it was discarded; the rejection of the record is the normal behavior for Network Slicer Engine because there are not defined any QoS policy inside of it.

4.3 Considerations

Measurements shows the expected behavior of the infrastructure with high responsiveness from the entire system when a new event is triggered.
The infrastructure responsiveness indicates also a good feasibility with more complex scenarios with different slice requirements also on different links.

CONCLUSIONS

The Network Slicer Engine is defined as independent module for QoS parameter definition, used also as main mechanism to enable the Network Slice feature. This feature is one of the main principle of the 5th generation network infrastructure which will become the main connectivity system after the year 2020.

The examination of related work show us that the QoS parameters are defined at the edge of network and network traffic is addressed defining a path through the internal network. We had also to be compliant with the ETSI NFV specification in order to define a component which is usable by every platform ETSI compliant.

After the analysis of Open Flow and Open vSwitch as reference technologies to enable the QoS parameters in cloud environment we have developed the Connectivity Manager Agent (CMA) which has become our SDN Controller. The CMA uses Openstack as reference cloud platform and interact with Open vSwitch using the virtual switch controller `ovs-vsctl` and Open Flow controller `ovs-ofctl`; it allows the interaction from application layer through REST APIs.

In order to be compliant with the ETSI NFV specification and also to enable the integration with Open Baton framework we defined the Network Slicer Engine to receive events for instantiation of a new network service record, retrieves the QoS requirements from the ETSI data model, "translates" this requirements in the SDN Controller data model and demands the allocation to the latter. Internally the Network Slicer Engine is divided in two macro blocks plus one interface, this division has become necessary because we want to achieve the extendability of the platform also with other SDN platforms. The first "block" is the QoS Controller which receives events and parse the record (payload of the event) and checks for QoS requirements, if there are invokes a method of the interface in order to communicate with the SDN driver. The second "block" is the SDN Driver which will be directly invoked (through the interface) from the QoS controller and communicates with the SDN Controller in order to define the network slice (defined in turn by the QoS requirements in the network service record). The interface was defined to achieve the extendability and compatibility with other SDN platform but is not ETSI compliant, simply because the specification does not define any interface for QoS allocation. Further development could be done on the Network Slicer Engine, it could be expanded in order to support the multi data center environment to support a Network Service Record distributed in a multi VIM environment.

Another improvement always on Network Slicer Engine side, could be the integration with plug in system of Open Baton framework in order to handle multiple SDN Controllers in multiple locations.

The Connectivity Manager Agent could be replaced by another SDN controller such Onos or could be expanded in order to handle the physical hardware in place into the data center.

LIST OF FIGURES

BIBLIOGRAPHY

[1] Ericsson Network Service Business. http://www.slideshare.net/Ericsson/network-service-business.

[2] Dijkstra. *A note on two problems in connexion with graphs*. 1959.

[3] Spring Configuration Bean documentation. http://docs.spring.io/spring-javaconfig/docs/1.0.0.M4/reference/html/ch02s02.html.

[4] The NFVO sdk documentation. http://openbaton.github.io/documentation/nfvo-sdk/.

[5] Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[6] Spring Framework. http://spring.io.

[7] The Bottle.py framework. http://bottlepy.org/docs/dev/index.html.

[8] Openstack Reference Guide. http://www.openstack.org/.

[9] Indigo OpenFlow implementation. https://github.com/floodlight/indigo.

[10] Open System Interconnection. https://en.wikipedia.org/wiki/OSI_model.

[11] Kong Chee Meng Kannan Govindarajan. *Realizing the Quality of Service (QoS) in Software-Defined Networking (SDN) Based Cloud Infrastructure.*

[12] Openflow Java Library. https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi.

[13] Onos White Paper. http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf.

[14] The Beacon Paper. http://yuba.stanford.edu/~derickso/docs/hotsdn15-erickson.pdf.

[15] OpenDaylight reference platform. https://www.opendaylight.org/.

[16] The Iperf Scenario. https://iperf.fr/.

[17] ETSI NFV MANO Specification. https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf.

[18] The Generic VNFM. http://openbaton.github.io/documentation/vnfm-generic/.

[19] Zabbix website. http://www.zabbix.com/.

[20] ZHAng Hailong YAn Jinyao1. *HiQoS: An SDN-Based Multipath QoS Solution.*