TNOVA

NETWORK FUNCTIONS AS-A-SERVICE
OVER VIRTUALISED INFRASTRUCTURES

GRANT AGREEMENT NO. 619520

Deliverable D4.22

SDN Control Plane – Final

Editor Letterio Zuccaro (CRAT)

Contributors F. Cimorelli, F. Delli Priscoli, F. Liberati, R. Baldoni, A. Pietrabissa, V. Suraci, S. Mascolo, D. Gheorghita (CRAT), G. Xilouris, C. Sakkas (NCSRD), B. Parreira, J. Silva, J. Carapinha (PTIn), I. Trajkovska, D. Baudinot (ZHAW), S. De Domenico, B. Coffano, M. Di Girolamo (HPE), A. Abujoda (LUH)

Version 1.0

Date 30th April, 2016

PUBLIC (PU)

Executive Summary

This deliverable presents the current activities and interim results of the Task 4.2 of the T-NOVA project, focused on the implementation and characterisation of the T-NOVA's SDN Control plane.

Task 4.2 is specifically conceived to develop an SDN control framework allowing the management of virtual networks over datacentre physical infrastructures, so as to support the deployment of VNF services. To this end, relevant issues related to network isolation and traffic steering have been addressed when providing connectivity services, handling either WAN or intra-datacentre connections. In this context, leveraging on state-of-the-art network virtualisation frameworks, several approaches are proposed and detailed.

In addition, a major activity within this Task concerned the virtualisation of the control plane to overcome SDN centralisation issues. Research has focused on load-balancing mechanisms in distributed-virtualised controllers, with the purpose of providing an elastic control plane able to scale with the workload. Moreover, experimental analysis on the capability of the SDN controller to ensure persistence of the network configuration in case of failures is also treated.

Specifically, starting from the analysis of the key requirements affecting the SDN control plane procedures, a survey of the current state-of-the-art SDN frameworks has been conducted, aiming at selecting the reference baseline for the SDN Control Plane implementation. Then, the architecture of the Control Plane has been designed, identifying the functional components and additional features required to meet the T-NOVA needs. In this regard, several activities focused on the research, design and implementation aspects of the Control Plane core functionalities have been carried out. They include *(i)* approaches for steering traffic in SDN networks, *(ii)* algorithms for balancing the workload among multiple controllers, *(iii)* techniques for providing network slicing and isolation with support to QoS, *(iv)* analysis of the persistency of network configuration and, finally, *(v)* solutions to provide integration of WAN connectivity in T-NOVA.

The content presented in this deliverable reports the progress on the above-mentioned activities.

Table of Contents

Index of Figures

Index of Tables

1. INTRODUCTION

In the T-NOVA system the SDN Control Plane plays a key role, being responsible at southbound for the configuration, management and monitoring of the SDN-compatible network entities, while supplying northbound the orchestrator and management systems with enhanced network connectivity services.

It is a fact that SDN may benefit NFV applications with a scalable, elastic and on-demand network infrastructure, leveraging the programmability of the southbound network elements. However such elements, both physical and virtualised, need to be properly configured to fit the applications' requirements. This tricky task represents the main goal of the SDN Control Plane.

In this regard, leveraging existing SDN management frameworks, Task 4.2 proposes to design and develop an enhanced SDN controller for network services provisioning to support NFV applications. The activities within the task have been split in three main working areas:

- Programmatic network control: Dynamic and intelligent control of network resources, as well as flexible establishment of service function chaining through configuration of policies for steering network traffic.
- Network virtualisation: It concerns the deployment of virtual networks supporting QoS and overlay encapsulation, through a deep analysis of frameworks (i.e. Open vSwitch), protocols (i.e. Openflow [MAB+08]) and tunnelling solutions (i.e. NVGRE, VxLAN). The final aim is to provide an open, flexible and extensible interface for the instantiation, configuration and monitoring of isolated virtual networks.
- Control Plane virtualisation: It refers to the virtualisation of the network controller to ensure reliability and high availability in large-scale scenarios, as well as persistency of the network configuration. For these purposes, cloud computing capabilities combined with clustered approaches have been investigated in order to ensure elasticity, auto-scaling and load balancing of the SDN control plane.

Within T4.2, the work initially focused on determining the SDN platform best fitting the T-NOVA requirements. The selection was carried out after a thorough analysis of the current SDN controller implementations, the features they offer, the mechanisms they support for enhanced network services (i.e. slicing, traffic steering, QoS), the way they approach the distribution of the control workload.

Starting from the requirements laid out in D2.32 [D2.32], a high-level overall architecture is proposed and functional components are detailed. Then, considering the state-of-the-art solutions for the SDN control plane, the identification of missing functionalities and the investigation of potential extensions to be implemented in T-NOVA resulted in different research and development activities.

2. T-NOVA REQUIREMENTS

The activity started with the identification of key requirements affecting the network controller procedures and mechanisms. The table below provides a summary of the high-level requirements identified in T-NOVA concerning the SDN Control Plane. The full list of requirements has been collected and documented in detail in Deliverable 2.32 [D2.32].

Requirement	Description
Network connectivity and isolation	Applications and services should be connected to isolated networks, making sure that processing of packets on each network is independent of all the others.
Resource Monitoring	The provision of monitoring information should make management and orchestration entities aware of status and performance of the network infrastructure
QoS support	Applications and services should have specific performance needs, requiring mechanisms for QoS provisioning over the network infrastructure.
Performance	In large-scale scenarios with many nodes to be controlled, the control plane may suffer slower performance in terms of processed requests per second/average response time. So, mechanisms to limit this issue should be provided.
Scalability	The control plane should adapt to a variety of applications and scale according to their network load. This means that in some cases a distributed control plane may be required; therefore the T-NOVA control plane must be able to accommodate this requirement.
Robustness/Fault tolerance	Through redundancy mechanisms, it must be guaranteed that the controller does not represent a single point of failure.
Service chaining support	The network controller must be able to dynamically enforce and modify the chaining of network service functions, by properly steering the data traffic.
Inter-datacentre connectivity	The solution adopted for the control plane should be able to support inter-datacenter (inter-DC) connectivity by enforcing tunnelling rules and establishing trunks to allow network continuity, as in many practical cases this will be required due to the physical dispersion of resources.

Table 2-1 Key requirements for the SDN Control Plane

3. REFERENCE SDN CONTROLLERS

As briefly introduced, a deep investigation and analysis of a set of SDN controllers available in the state of the art has been carried out, in order to select a solid

reference baseline for the T-NOVA SDN control plane development. The aim was to identify a starting point of work to be properly extended in support of the specific T-NOVA requirements.

The analysis was performed in two stages. First, a wide set of controllers has been selected, including most of the currently available commercial and open-source SDN frameworks, and qualitatively evaluated at a very high level in order to extract a subset of candidate controllers. Then, a subsequent phase focused on a detailed feature-based comparison of the selected controllers.

In the following sections, an overview of the available SDN control plane is presented, including both commercial and open-source solutions, as input of the preliminary analysis that was carried out.

3.1. Commercial solutions

Alcatel Lucent Nuage VSP

Nuage Networks VSP (Virtualised Services Platform) [NUAGE] has an architecture based on three components:

- VSD (policy and analytics engine)
- VSC (control plane)
- VRS (forwarding plane)

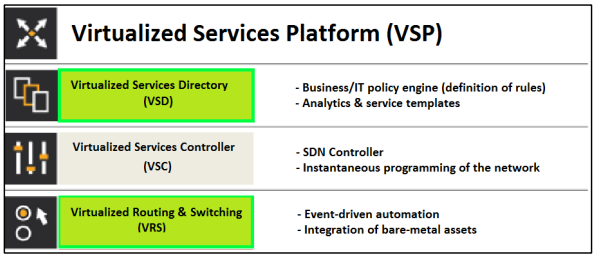


Figure 3-1 Nuage VSP architecture

VSD is the upper layer, providing service definition, policy establishment, service templating and all the analytics functions, including reporting. It is alike the SDN application layer in OpenDaylight. VSC is the actual SDN controller, providing features like auto-discovery and tenant slicing. VRS is the L2-L4 virtual switch, based on Open vSwitch, providing tunnel encapsulation and programmed by VSC via Openflow.

Nuage VSP has control of the whole underlying network, allowing to manage traffic flows even over the transport layer, interconnecting virtual machines residing in different nodes or sites.

Nuage VSP exploits a Neutron plugin, connecting to VSD via a REST API. Plus, there is a Nuage VRS agent on each Nova node, which monitor VM lifetime events and

configure the network topology accordingly. Once an event is captured (e.g., a new VM instantiation), the VRS agent requests configuration information from VSC. If VSC is already aware of the VM, it will supply all the necessary configuration information via Openflow. If this is a new VM, VSC will request policy information from VSD. VM configuration includes tunnel configuration for traffic between hypervisors and other components, carried via VXLAN between VSP components or MPLS over GRE for compatibility with Provider Edge (PE) routers.

Nuage VSP is also part of the HPE Distributed Cloud Networking (DCN) network virtualisation solution for service providers. HPE DCN creates a secure virtual network overlay across distributed datacentre sites. Nuage VSP is the DCN component federating SDN implementations in individual data centers, to create an uber-SDN overlay across the wide area.

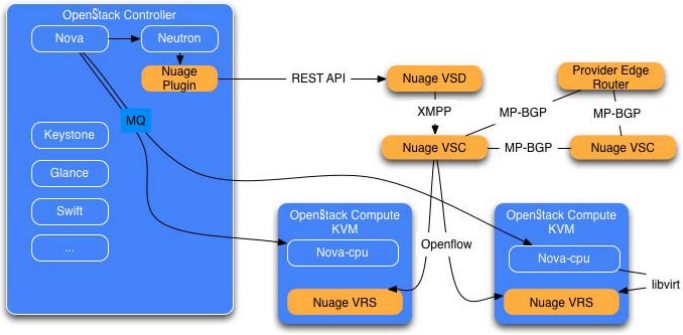
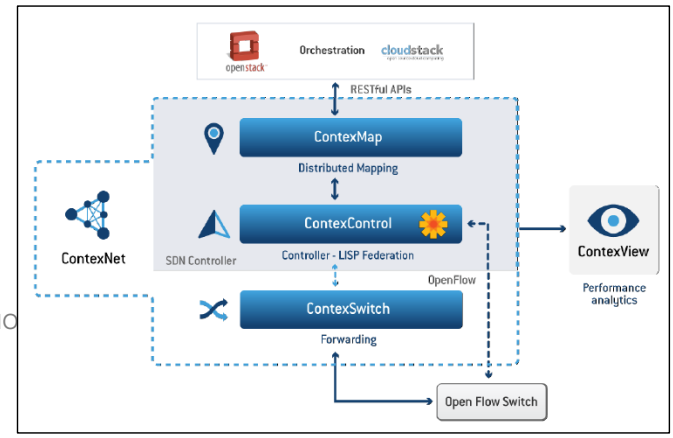


Figure 3-2 Nuage Neutron plugin

HPE ContexNet

ContexNet [HPECTX] is an OpenDaylight-based carrier grade distributed SDN fabric running on off-the-shelf computing platforms and fully hypervisor-agnostic. It allows to create a full service abstraction layer on top of legacy networks, allowing for instance a customised SFC by classifying and steering the traffic according to the specific user flow. It exploits a distributed controller and a set of virtual switches to implement global load balancing (covering physical and virtual resources).

The overlay abstraction layer enables a centralised policy-based control of the



network, regardless the actual physical placement of virtual machines and endpoints. It is in-network, with Openflow switches in charge of traffic encapsulation-decapsulation, and a set of APIs enabling fine-grained traffic control.

Figure 3-3 HPE ContextNet

Juniper Contrail

Contrail [JUNICON] consists of two main components: a Controller and a vRouter. The Controller is logically centralised but physically distributed, and it also embeds the management and analytics functionalities. vRouter runs in the hypervisor, and is similar to Open vSwitch, but it also provides routing services.

The Contrail model is similar to MPLS L3VPN and EVPN (for layer 3 and 2 overlays). In the forwarding plane, it supports MPLS over GRE/UDP and VxLAN. As control plane protocol, it uses BGP + Netconf. Finally, the protocol between Controller and vRouter(s) is XMPP.

The physical nodes hosting a distributed Contrail controller can be of three types: configuration nodes (dealing with the management layer), control nodes (implementing the logically centralised function of the Control Plane), and analytics nodes (monitoring data collection, processing and presentation).

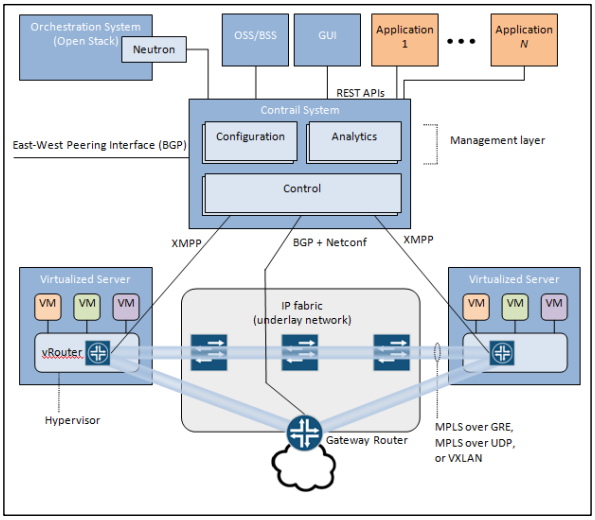


Figure 3-4 - Contrail architecture

VMware NSX

VMware NSX is a network virtualisation platform using flow-based forwarding via Openflow to instantiate the network flows. Flow forwarding exposes various L2-L4 header fields, along with Layer 1 logical and physical interfaces. It is based on a vSwitch embedded in the ESX hypervisor, attached to the actual NSX controller. NSX builds tunnels between different vSwitches using VxLAN originating and terminating

in VxLAN Tunnel Endpoints (VTEPs). The VTEPs connect each vSwitch to the IP network.

In the NSX architecture, when a VM boots its host registers with the NSX controller. The controller consults a table that identifies the tenant, and returns the topology the host should participate in to the vSwitch. The key identifier for virtual isolation is the VNI, which maps to a tenant's VxLAN-segmented topology. Layer 3 forwarding between broadcast domains is supported at the edge of the NSX network in the vSwitch. This is performed by ARPs being punted to the controller and looking up the location of the destination MAC and destination VTEP in a host table in the controller.

If the host is not found, the traffic can be dropped or forwarded to a BUM traffic service node. Host discovery is forwarded to the VTEPs in the tenant tunnel overlay with multicast.

3.2. Open source solutions

Besides the commercial solutions, a set of available open-source SDN controllers was identified during the evaluation phase.

Beacon

Open-source controller developed by Stanford University [BEACON], implemented in Java. It offers support for the Openflow v1.0 protocol. Beacon is not actively developed at this time.

Floodlight

Open-source controller developed by Big Switch Networks [FLOODLIGHT], implemented in Java. It offers support for the Openflow v1.0 protocol and a Neutron plug-in for OpenStack support. Floodlight is not actively developed at this time.

Maestro

Open-source controller developed by Rice University [ZC11], implemented in Java. It offers support for the Openflow v1.0 protocol. Maestro is not actively developed at this time.

MUL

Open-source controller developed by Kulcloud [MUL], implemented in C. It offers support for the Openflow v1.0, Openflow v1.3 and Openflow v1.4. MUL is actively developed at this time.

Nodeflow

Open-source controller developed by CISCO [NODEFLOW], implemented in Javascript. Nodeflow is not actively developed at this time.

NOX

Open-source controller developed by Nicira Networks [NOX], implemented in C++ and Python. It offers support for the Openflow v1.0 protocol. NOX is not actively developed at this time.

ONOS

Open-source SDN controller platform developed by ON.LAB [ONOS][BGH+14]. ONOS is designed for high availability, performance and scalability within Service Provider network. ONOS is actively developed at this time..

OpenContrail

SDN platform released by Juniper Networks [JUNIOPENC], as open source counterpart of the commercial Contrail solution. The OpenContrail Controller, which is part of the platform, is implemented in Python, while the projects comprising OpenContrail are implemented in various programming languages (Python, C++ and JavaScript). It offers OpenStack support but the current version lacks of Openflow support. OpenContrail is actively developed at this time.

OpenIRIS

Open-source controller developed by ETRI [OPENIRIS], implemented in Java. It offers support for the Openflow v1.0 to v1.3. OpenIRIS is actively developed at this time.

OpenDaylight

Open-source platform for network programmability developed by Linux Foundation [ODL], implemented in Java. It offers support for OF v1.0 to 1.4, as well as OpenStack support via Neutron plugin. OpenDaylight is actively developed at this time.

POX

Open-source controller developed by Nicira Networks [POX], implemented in Python. It offers support for the Openflow v1.0 protocol. POX is not actively developed at this time.

Ryu

Open-source controller developed by NTT, implemented in Python. It offers support for OF vv1.0, 1.2, 1.3 and 1.4, as well as OpenStack support. Ryu is actively developed at this time.

Trema

Open-source controller developed by NEC [TREMA], implemented in C and Ruby. It offers support for the Openflow v1.0, Openflow v1.2 and Openflow v1.3.X protocol and a Neutron plug-in for OpenStack support. Trema is actively developed.

3.3. Comparison and selection of the reference baseline

The controller selection phase was driven by a feature-based comparison of the available solutions [KZM+14], whereby the first discriminating criterion was to consider only those under active development and released as open-source software. Then, key aspects in the choice were the modularity and the extensibility of their

architecture, to be easily enhanced in support of novel functions, as well as the openness of northbound interfaces, for new APIs to be exposed without the need of tweaking internal controller services.

Therefore, leveraging also on the consortium's hands-on experience with various controllers, **OpenDaylight (Lithium release)** and **ONOS (Cardinal 1.2 release)** were selected as candidate controllers for the final evaluation phase, considering the latest stable versions available at the time. For a detailed description of both ONOS and OpenDaylight, please refer to Section 4.6 of D2.32.

The final comparison involving the two selected SDN controllers resulted in a qualitative analysis of their main features, mainly focusing on the capabilities required by the T-NOVA system. A summary of such a qualitative comparison is reported in Table 3-1.

Feature	OpenDaylight Lithium	ONOS Cardinal 1.2
Modular and extensible architecture	Yes. Built on top of Karaf, an OSGi framework, ODL provides dynamic module loading	Yes. Built on top of Karaf, an OSGi framework, ONOS provides dynamic module loading
Network virtualisation support	Yes. Different built-in solutions (VTN, DOVE, OVSDB)	Not built-in. External frameworks (OpenVirtex) address network virtualisation
Service Insertion and Chaining	Preliminary implementation available (ODL SFC project)	Not yet available. Implementation on-going (see ONOSFW project)
VIM integration	Fully integrated in Openstack through the Neutron (ML2 Plugin)	Not yet available. Implementation on-going (see ONOSFW project)
Openflow support	1.0-1.4	1.0-1.3
Clustering support	Clustering support for multi-controller deployments	Clustering support for multi-controller deployments
Documentation	Very extensive and detailed	Enough information available

Table 3-1 SDN Controllers Features Comparison

In the light of above, the final choice between OpenDaylight and ONOS fell on the **OpenDaylight Lithium** platform for the great interest and development it is undergoing and its numerous features matching the T-NOVA requirements.

For the sake of completeness, it should be noted that, even if Lithium has been selected as favourite OpenDaylight release, the implementation and testing activities carried out by Task 4.2 and described in the rest of the document, affected also previous releases of ODL (i.e. Hydrogen, Helium), as they were preferred for the

maturity and reliability of their features at the time they were explored. Similarly, it is not excluded the possibility of adopting even more recent releases of ODL for the final T-NOVA demonstrator, if they should lead to tangible benefits for the SDN Control Plane deployment.

4. T-NOVA SDN Control Plane

4.1. Functional architecture

Within Task 4.2, a key activity was focused on the design of a preliminary architecture for the network control plane. Such activity concerned the identification of components and modules contributing to the accomplishment of the purposes of the SDN Control Plane in T-NOVA.

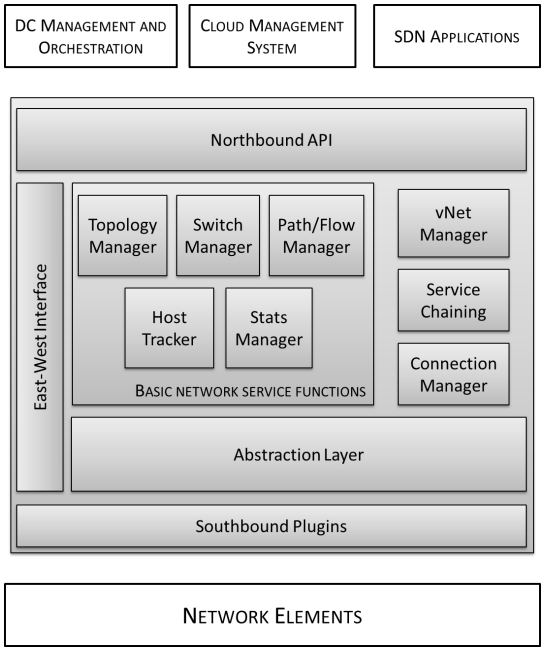


Figure 4-1 T-NOVA SDN Control Plane Architecture

Figure 4-1 depicts the SDN control plane functional architecture. To this end, a model of idealised SDN framework was selected as reference point [NG13] to be further extended and properly adapted to fulfill the requirements previously described.

As already described in Deliverable 4.01, the following table outlines the main functional components that have been identified with a brief description of their role within the SDN control plane.

Component	Functionalities
Topology Manager	The Topology Manager has to learn and manage topology information about devices and their reachability. Information gathering about the networks' elements is essential to discover the topology.
Switch Manager	The Switch Manager is in charge of storing, managing and providing information (e.g. switch id, software version, capabilities, etc.) about the network nodes as they are discovered.
Path/Flow Manager	This module provides the flow programming services including forwarding rule installation and removal for the configuration of data paths. Used typically when high-level policies specified by the northbound are translated into flows by a service module (Service Chaining, Slice Manager) that in turn would talk to this module to proactively push the flows down to the network elements. Path reconfiguration (after network failures or VM migration) and QoS support are in charge of this module.
Host Tracker	The host tracker module learns, statically or dynamically about IP hosts in the network. It stores and provides host information, such Host's IP address, MAC address, switch ID, port, and VLAN. Moreover it periodically refreshes the hosts' data to track the element location (switch, port, MAC, or VLAN), and notifies the listening applications to the hosts related event.
Stats Manager	This module stores and provides network statistics data with different data granularity (flow, port and table statistics).
vNet Manager	This functional module allows the creation of multiple, isolated, virtual tenant networks on top of a single physical network, in order to enable the complete separation between the logical and physical plane, hide the complexity of the underlying network and also optimizes the network resources usage.
Service Chaining	This functional module has to manage the deployment of services chains as ordered list of a network services (e.g. firewalls, load balancers) by configuring accordingly traffic steering.
Connection Manager	This module manages information regarding the connection between the control plane and network entities. It plays a key role in multi-controller scenarios.

Table 4-1 SDN Control Plane Functional Components

4.1.1. Virtualised SDN Control Plane

The SDN paradigm looks highly complementary to NFV having the potential to provide a scalable, elastic and on-demand network infrastructure. Additionally, the centralised view of the network allows the SDN network controller to make optimal forwarding decisions. However, the controller may be subject to overload or failure issues and increasing the computational and memory capacity may not be enough. These issues have an impact on the control plane reactiveness, and consequently degrade the overall network latency. It becomes more evident when the network size grows, thus a way to overcome these limitations is needed to make SDN/NFV a pillar technology of DC networks.

In this regard, the concept of a "physically distributed, but logically centralised" controller has been investigated so as to develop an instance of the virtualisation layer applying for a distributed control of the network elements [ZCD+15]. The proposed SDN/NFV Control Plane is based on the virtualisation of the network controller through multiple controller instances organised in *cluster*, while keeping the benefits of having a global view of the network by means of a distributed data store. The key concept is to deploy each instance of SDN controller on dedicated virtual machines, favouring the distribution of the network control workload across the cluster. In this way, the controller virtualisation may help in overcoming scalability and centralisation issues, which affect the SDN controller performances in large data center hosting NFV applications.

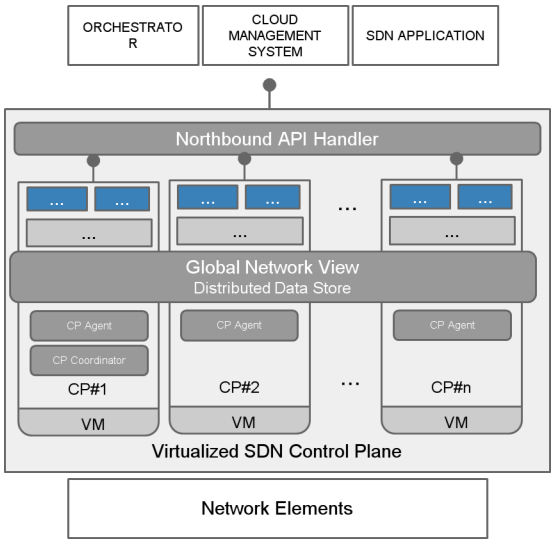


Figure 4-2 Virtualised SDN Control Plane Architecture

Therefore, the high-level architecture of the SDN Control Plane has been extended to support deployments in a large-scale scenario, by introducing the following functional components:

- **Distributed Data Store:** it is responsible for consistently maintaining a global view (topology and the state of the network) across the control plane instances belonging to the cluster. Northbound applications/internal Control Plane components can take advantage of the global network view in making forwarding and policy decisions
- **Northbound Request Handler**: it is mainly in charge of spreading the northbound requests among the available controller instances, it is essential to make the network control plane accessible through the northbound API as a unique single instance.
- **CP Coordinator**: it supervises the operation in the cluster. Specifically it has to dynamically configure the controller-to-switch connections; decide whether to add or remove a controller instance to the cluster depending on the network needs. This role is played by one of the instances available in the cluster, by means of a procedure of leader election.
- **CP Agent**: it collects information about the resource utilisation (CPU load, memory usage, control messages arrival rate, etc.) at each Control Plane instance and enforces the switch-to-controller instance connection rules used by each switch to identify the controller instance/s to which the southbound requests must be forwarded.

4.2. Relevant features in T-NOVA context

As previously mentioned, OpenDaylight has been selected as the reference framework for the SDN control plane software implementation in T-NOVA, due to its extensible modular architecture, and the wide set of services, appliances and northbound primitives available for data centre deployments. Subsequently, the selection phase was followed by the identification of missing functionalities and the investigation of potential extensions to be implemented in T-NOVA, as reported in Table 4-2.

Functionalities in T-NOVA	Existing modules	Missing features / Potential extensions
Path/Flow Manager	Forwarding Rule Manager / Flow Programmer (ODL)	QoS support Path reconfiguration
Virtual Network Manager	VTN Manager (ODL)	None
Service Chaining	SFC (ODL)	Traffic steering mechanisms for the provisioning of service chains.
High-Availability	Clustering Service (ODL)	Load balancing across clustered controllers by properly managing connection with the switches.
Network Configuration Persistency	Clustering Service (ODL)	None

Inter-DC/WAN integration	None	Integration of WAN connectivity services and connectivity between data centers

Table 4-2 Mapping between T-NOVA SDN Control Plane and OpenDaylight

The following sections detail the activities carried out for each identified topic, in order to fill the gap with the SDN control plane functionalities requested by the T-NOVA system.

4.3. Traffic steering mechanisms in virtualised network

A key aspect in T-NOVA and, more generally, in the NFV context, is the support to the service chaining. It translates into the ability of the SDN Control plane to configure network flows in order to steer packets through a sequence of network nodes. This section highlights the activities undertaken in this field, starting from analysis of the current available solutions, focused on OpenDaylight as selected controller. Then, this section presents the approach that has been adopted as the most suitable to be implemented in T-NOVA, next to other alternative approaches that have been nevertheless studied and examined.

4.3.1. Service Function Chaining

Service Function Chaining enables the creation of ordered list of network services, called Service Functions (SFs) aimed to be applied over specific set of packets that will traverse the path of those functions. The service functions are stitched together in overlay on the top of the network forming the so-called a Service Chain. SFC and its use cases have been introduced in several IETF RFCs ([SFC00], [SFC03], [SFC04], [SFC11], [NSH]).

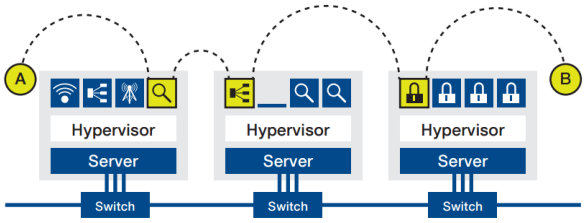


Figure 4-3 Example of SFC

An example of Service Function Chaining is shown in Figure 4-3. A flow originating from endpoint A passes through a network monitoring VNF, a load balancing VNF and finally a firewall VNF before arriving at destination point B.
Today some of the problems the current service deployments encounter on a network level include: topological dependencies, complexity in configuration, packet classification, agile/elastic service delivery, enforcement of consistent ordering of service functions [ODL-SFC1]. Therefore a novel approach is required in order to address these challenges from a network point of view.

SDN simplifies service chain provisioning and management because the SDN controller has a centralised view of the network and thus it facilitates end-to-end chains across different network nodes.
There are several possible approaches to achieve SFC on a network level that address the above listed challenges. One is to use encapsulation where the end to end traffic is treated as an overlay connection either (1) between the service nodes or (2) independent of the network topology. Today a dedicated protocol header (Network Service Header) [ODL-NSH] is currently under development as example of an encapsulation SFC approach. Network Service Header is an IETF data-plane protocol that represents a service path in the network. NSH is expandable header that is inserted in the packet via a classifier at the service plane entry and carried along the chain. It has a limited lifetime only within the SFC domain. NSH contains two major components: *Path Information* and *Metadata*. *Path Information* is akin to a subway map: it tells the packets where to go without requiring per flow configuration. *Metadata* is information about the packets, and can be used to define policy for the chain.
The NSH SFC implementation is built on the top of the NSH solution. Some of the terminologies (components) introduced by this protocol specification include:

- Service Function Forwarder (SFF): Switch/Data Plane Node
- Service Function (SF): any application such as DPI/FW/LB/TIC
- Service Function Chain (SFC): the intended list of SFs that the packets have to traverse in a definite order
- Service Function Path (SFP): actual instance of the services that are traversed, or a specific instance of the SFC
- Service Classifier: Function that helps in packet classification
- Metadata: Information that is carried across nodes
- Network Service Header: SFC encapsulation used by SFC-aware nodes, in case of SFC-unaware nodes, SFC-proxy has to be used
- Nodes could be either SFs or SFFs

4.3.1.1. SFC support in OpenDaylight

The OpenDaylight framework (Figure 4-4) includes an SFC project implementation that leverages the NSH protocol. It requires augmented version of OVS that tells ODL to use the *sfcovs* southbound protocol to communicate with the actual device.

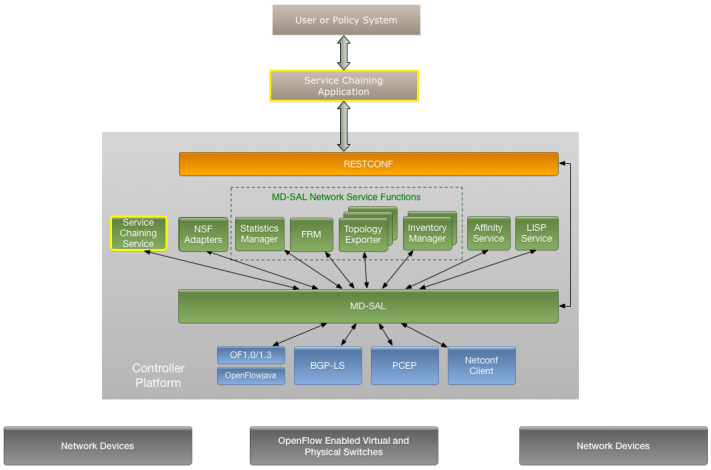


Figure 4-4 SFC project inside the ODL - High Level Architecture

In ODL the SFC data model is defined in Yang files that put the information in the MD-SAL at a compile time. RESTCONF APIs and southbound hooks are created from Yang. In a typical scenario the client sends packet to SFF which processes and sends it to the SF; the SF decrements the index and sends back to SFF; SFF receives the packet back from SF and sends it back to client [ODL-SFC2] [ODL-SFC3].

Figure 4-5 depicts an example scenario of SFC using the SFC agent from the ODL SFC project. It depicts in a graphic way the Service Chain that a client is using. Each Service Function adds a different HTTP header and an Apache Web Server detects the HTTP headers and returns different web pages. Clients assigned to different Service Chains use a web browser to navigate to the same Web Server, but get different results depending on the Service Chain used.

Overall there are several issues of why employing this solution in T-NOVA (that is the current NSH OVS implementation as SFF) is not stable at the moment. One includes the dynamic process of development and constant improvement and testing of this functionality that makes it prone to errors and not fully functional at current stage (it was introduces in ODL Lithium recently). Moreover a different, augmented version of OVS is required (NSH-aware). To interoperate with the current implementation of NSH in OVS the encapsulation used (encapsulate VxLAN-NSH-Ethernet-Legacy) is very peculiar - it is VxLAN (not GPE) + NSH over port 6633. This introduces additional overheads because the packet the SFF sends back to client is a VxLAN + NSH packet and not a plain IP packet.

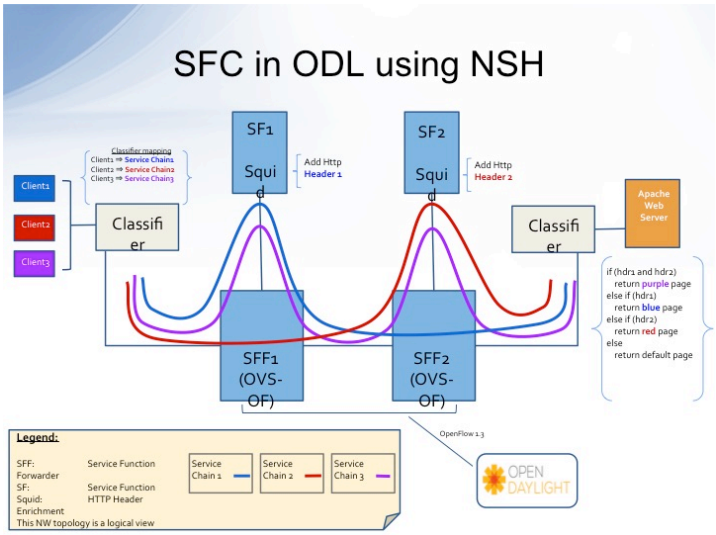


Figure 4-5 Example of SFC approach based on NSH protocol

With SDN it is also possible to do address rewrites along the chain or to simply force the routing over a specific path. Ericsson has implementation based on the former approach contributed to ODL (Figure 4-6).

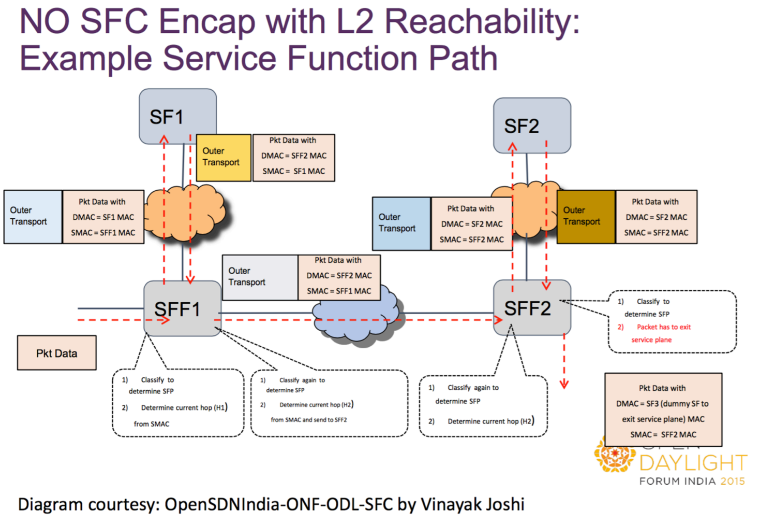


Diagram courtesy: OpenSDNIndia-ONF-ODL-SFC by Vinayak Joshi

Figure 4-6 Non-NSH approach for SFC based on MAC/VLAN matching

Their solution does not require NSH encapsulation or other transport tunnelling encapsulation such as VxLAN or GRE. Instead, it is based on Openflow 1.3. where the L2 reachability of SFs and SFFs is provided in the Yang files of ODL. Here, the packets are reclassified at every SFF and the service hop is based on the MAC address of the previous hop using the VLAN ID in the packet header.

4.3.1.2. Flow programming approach based on Openflow

The latter option (flow based approach) can be implemented on Open Flow enabled switches on the southbound in order to program the SFFs. Such approach has the benefit of essentially leaving the datagrams untouched along the chain, while providing a routing logic, which does not require the overhead of tunnelling or encapsulation. Having the original datagrams along the chain has an additional benefit with network functions in particular, because they can rely on seeing datagrams as if they would be routed through a chain-less connection. This solutions avoids headers, proxies, or additional third party components.

A flow programming approach (as can be seen through an example on Figure 4-7), has been introduced in ODL from the NEC team as alternative solution to the NSH-based one [OOSC]. It is based on the Virtual Tenant Network (VTN) project that provides network virtualisation in OpenStack multi-tenant environment, traffic isolation and abstraction of physical network. Some of the key features form this project include: ability to insert service functions dynamically; OpenStack integration; does not require NSH capability; works with Openflow switches; ability to visualize end-to-end flows. It involves two main components: *VTN coordinator* and *VTN Manager*. The *VTN coordinator* provides VTN API (Northbound), builds VTN models using OpenDaylight API and controls multiple SDN controllers. The *VTN Manager* enables multi-tenancy and end-to-end dynamic path control. The example shown on the figure depicts a SFC case based on L3 IP address matching. As alternative to the source and destination IP matching and according to the specification, the actions can be enforced to other matching criteria enclosed in the Openflow protocol types.

The strong point of this approach is that it has been designed to coexist with OpenStack Neutron ML2 Plugin. The disadvantage however is that the agent that embraces the virtual graph mapping as well as the chaining logic is proprietary and heavily based on the VTN abstraction model. It is yet under development process and tightly coupled to the NEC dedicated facilities adopted and designed to fully sport this use case within their isolated experimental environment. From here we are not able to adapt and leverage this solution for the T-NOVA networking model and therefore this approach would not be considered any further as T-NOVA SFC baseline model.

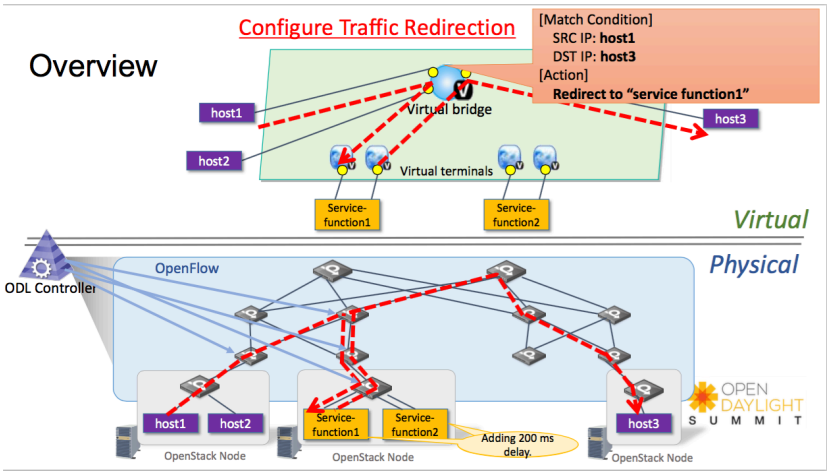


Figure 4-7 Flow programming based SFC approach based on VTN in ODL

Finally the SFC approach from the OPNFV community is addressed in the projects OpenStack Based VNF Forwarding Graph [OPNFV-FG] and Service Function Chaining [OPNFV-SFC].

Leveraging the OpenStack work on VNFFG (Virtual Network Function Forwarding Graph) and ONF Openflow work on service chaining, this project tends to show automatic set up of end-to-end VNF services through VNFFG so that different tenants' flows can be steered through different sequence of VNFs (Service Function). The second collaborative development project will base on the first one to create a link between two Linux Foundation projects, OpenDaylight and OPNFV. It will provide service chaining capabilities in the OPNFV platform, i.e. provide ordered set of abstract service functions (e.g. NAT, load balancing, QoS, firewall) and ordering constraints that must be applied to packets and/or frames and/or flows selected as a result of classification [OPNFV-SFC].

These projects provide Openflow programmed chains for L2 VLAN and MPLS encapsulation. They also follow the VxLAN overlay based service chains for VxLAN-GPE encapsulation with NSH headers. Some of the additional features that are supported are: basic load balancing at SFC in the ODL Lithium release, and programmatic service function selection algorithms like round robin, load balanced (choose the least loaded service function) or random allocation.

Finally all the previously presented approaches based on the ODL implementation lack the support and integration with Open Stack, which is one of the key technologies to be used in T-NOVA. The T-NOVA specific Virtual Network Functions (VNFs) would be instantiated and deployed on Open Stack VMs and therefore the chaining mechanism has to be supported and fully functional with the Open Stack networking standards. To apply the NSH solution in Open Stack environment, the SFC would need to know the following data form each of the service functions VMs: IP

Address, encapsulation details (VxLAN, NSH enabled), OVS switch and port the SF is connected to.

4.3.2. T-NOVA approach for SFC based on OpenFlow

Having analysed the previous solutions on SFC we came to conclusion that at a current state, an alternative approach, native to T-NOVA will be the most feasible solution. Hereby we summarize the reasons:

- OpenStack support: current solutions does not allow integration with OpenStack and that might require adaptations in order to make it work in short term scale. T-Nova uses OpenStack as deployment infrastructure for the VNFs.
- ODL support: this is the de facto controller in T-NOVA and therefore one of the requirements is to keep the solution compliant with ODL.
- T-NOVA requires stable solution that will not depend on proprietary implementations of the dependent features. For example the VTN solution (at current state) was tested on specifically triggered topology and bases on integrating VTN plugin within the ODL controller.
- T-NOVA requires a simple isolated solution that is robust and independent of the work in progress prototypes that require the installation and support of specific software libraries, especially based on certain type of hardware implementations (like NSH-enhanced OVS), etc.
- The SFC approach in T-NOVA has to offer integration with the Infrastructure Virtualisation Management (IVM) and the Orchestrator (TeNOR) in order to expose APIs for VNF placement and virtual network graph definition. As the SDK for SDN includes libraries and APIs for establishing specific network services, this service can be integrated in seamless fashion with other T-NOVA components.

Currently two SDN-based approaches has been tested on the ZHAW testbed: (1) based on SFC specific flows installation along the OVSs (including the physical switch) of the nodes (hops) in the service path, (2) based on MAC rewriting.

To enable these routing rules in an OpenStack environment it is necessary to disable the iptables rules which are automatically applied to OpenStack instance ports. These rules prevent traffic to pass to and from instances that does not match the instances MAC and IP addresses. Service instances have to have two separate interfaces, which are dedicated to the service chain mechanism. These interfaces then can only be attached to Neutron ports with disabled iptables firewall rules and should only be used for the chaining mechanism. Remaining security concerns then have to be delegated to flows instantiated for the respective OVS ports.

For both approaches to work, a requested chain can only be realised if the network is fully SDN enabled. The chain routing can then be applied across the whole chain path onto the network. The resulting routing flows need to be maintained to reflect alterations in the function chain. For the chain routing to be deterministic, a challenge to be addressed is the service path identification (service path classification or entity

that determines what traffic needs to be chained based on policy) and the service hop identification (or a field which keeps track of the chain hops).

One possible approach to access this in T-NOVA is to use TTL matching and modification, since OVS flows can match on the TTL field. It is however currently not possible to match the TTL field with standard Open Flow, so we are elaborating about using the VLAN ID as a workaround for this purpose, which would be compliant with our networking model, since the chain routing does not follow the standard Ethernet routing.

As alternative to this, the second approach was tested in Mininet and proved feasible for the established basic scenario of SFC. The idea for the OpenFlow based MAC rewriting approach was conducted based on the requirements imposed by the testing scenarios using vTC as T-Nova specific VNF. Our original goal was to maintain the packet structure along the service chain as if it would be normally forwarded from enpoint to endpoint, which we couldn't do for two reasons:

- The VNF deployment (physical location of each VNF) may be in a way so we would get non-deterministic paths.

- The introduction of the Traffic Classifier VNF, which has to encode the chosen service chain into the datagrams.

This solution is our way to deal with those problems, but it is not in any way set in stone for the SDK but rather shows one possible way we can implement using the SDK while having minimal overhead. Other possible solutions could be based on tunneling (MPLS/GRE) or other rewrite patterns, which could be implemented using the Flow Pattern abstraction of the SDK.

This problem of SFC have been further addressed in the SDK for SDN part of the WP4 and detailed in the Deliverables D4.31-Interim [D4.31] and D4.32-Final [D4.32].

4.3.3. Alternative approaches to traffic steering

4.3.3.1. OpenStack extension using OpenDaylight

Besides the above-mentioned approaches, a Traffic Steering extension to OpenStack has been implemented and evaluated, allowing the forced redirection of traffic across VMs, which indirectly enables the implementation of service function chaining.

This extension was implemented by extending the OpenStack Neutron API to support the definition of classification resources, L2-L4 traffic filters, and a sequence of redirections, i.e. a list of Neutron ports in which the packets are forced to pass through. Moreover, the OpenStack API itself only provides a means to declare the previously mentioned resources; therefore, in order to enforce the necessary configurations in OVS (i.e. the default virtual switch implementation used in OpenStack), OpenDaylight was chosen. OpenDaylight is capable of programming OVS, and features an extension mechanism which was used to support the Traffic Steering extension.

In the following sections the extension mechanism and the Traffic Steering extension in OpenDaylight are presented. Additionally, the work realised to extend the

Openstack API to support Traffic Steering is described. Finally, an example is provided focusing on how the API is used to realize a forwarding graph.

OSGi and OpenDaylight

In this section the mechanisms to extend OpenDaylight to support traffic steering are presented. OpenDaylight includes a modular, pluggable, and flexible controller platform at its core. This controller is implemented strictly in software and is contained within its own Java Virtual Machine (JVM). Therefore, it can be deployed on any hardware and operating system platform that supports Java.

The controller exposes open northbound APIs which are used by applications. OpenDaylight supports the OSGi framework and bidirectional REST for the northbound API.

OSGi is a modular framework that allows for dynamic loading of Java modules. This means that applications can be loaded and unloaded, started and stopped without interruption of the running JVM platform. This allows applications and protocols to plug into the framework to fit different use cases and vendor strategies. These modules are called bundles, more specifically they consist of jars with manifest files that define what is exported/imported to/from other bundles in addition to other details such as bundle name, activator, version etc, see Figure 4-8.

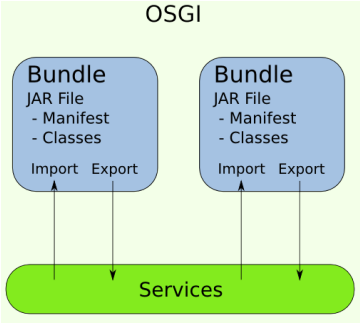


Figure 4-8 OSGi overview

A bundle alone can act as a service provider or a service consumer. Services are specified by Java interfaces. Bundles can implement this interface and register the service with the Service Registry.

Implementation of Traffic Steering bundle

To add the traffic steering functionality to OpenDaylight, a Maven OSGi project was created that, when compiled, creates a bundle dynamically deployable into the OpenDaylight controller. Basically, this bundle extends the OpenDaylight Northbound REST API to create two new base endpoints:

- */ts/steering_classifiers/*

- */ts/port_chains/*

These two endpoints provide the Create, Read, Update, Delete (CRUD) functionalities to the main objects, Classifiers and PortChains, respectively.

In order to implement the bundle, an OSGi activator class is needed. The latter enables registering a component with the OSGi framework and defines the exposed services that will be used in the Northbound API. The rest of the java classes will implement the logic of the services themselves and implement the new Northbound API resources, see Table 4-3.

Description	Steering_Classifiers	Port_chains
Object Data Model	*Classifier.java*	*PortChain.java*
Interface that defines the CRUD methods	*IClassifierCRUD.java*	*IPortChainCRUD.java*
Implementation of the interface	*ClassifierInterface.java*	*PortChainInterface.java*
Definition and implementation of the API	*ClassifiersNorthbound.java*	*PortChainNorthbound.java*

Table 4-3 Java classes implementing traffic steering

The OSGi framework allows to share the different services exposed by each module. In the module which implements the port chain the following external services are needed:

- *OVSDBConfigService*: to get OVSDB tables information needed to learn the OVS port where the VMs are connected
- *INeutronPortCRUD*: to get Neutron ports information and details
- *IForwardingRulesManager*: Manager of all the Forwarding Rules, this component takes care of forwarding rules and is the one managing conflicts between them
- *ISwitchManager*: Component holding the inventory information for all the known nodes in the controller. All the components that want to have access to a port name, node name or any inventory information, will find them by querying the SwitchManager.

The packets can only be manipulated in the OVS, where the flows are installed. The OVSDB tables information is used to find and create a map with the connections links between the neutron ports and the OVS ports. The flows are constructed with this information in addition to which is defined by the classifiers in the PortChain. One important note, since the packets are redirected to a different hop, it is also needed to change the packet mac address destination, so it can be accepted by the host/function. The *IForwardingRulesManager* service is used to insert (also to delete

and update) a static flow in the OVS for each hop in the chain based on the flow configuration built by the previous steps.

OpenStack extension for traffic steering

This section refers to the traffic steering extension implementation in OpenStack. The plugin structure is somewhat alike to ML2 and Group Policy plugins, including a steering manager, context objects, driver API, a dummy driver and an OpenDaylight driver. Like other Neutron plugins, administrator users will have to enable this plugin and the respective driver in Neutron configuration file. This extension has been confirmed to work in OpenStack Icehouse version.

The traffic steering extension in OpenStack consists of two concepts:

- Traffic Classification - a policy for matching packets, e.g. HTTP traffic, that is used for the identification of the appropriate actions to apply to the packets. It can be for example an explicit forwarding entry in a network device that forwards packets from one address, identified for example by an IP or MAC, into the Service Function Chain;
- Traffic Steering - ability to manipulate the route of traffic, i.e. delivering packets from one point to another, at the granularity of subscriber and traffic types. Neither the actual network topology nor the overlay transports are modified to accomplish this.

The manipulation of traffic redirection occurs at the port level. For example, all HTTP traffic coming from a VM interface is directed to another VM interface instead of going to the network gateway. With this in mind two new resources were added to Neutron:

- Steering Classifier - traffic classification which supports the following filters: protocol, source/destination MAC address, source/destination IP address and source/destination port range;
- Port-Chain - sequence of traffic redirections. This is done with a dictionary of lists of Neutron ports where the dictionary keys are the Neutron port UUIDs. Ingress traffic from these ports is steered to all ports in the dictionary value (a list of Neutron ports) according to a classification criterion.

The Traffic Steering API provides a flag that alerts in situations where there is more than one path from a source to the same destination, or there is a path that can form a loop. Note that when redirecting traffic intended for the network gateway to another VM in OpenStack, it is necessary to manipulate the packets so that the VM can process the packet. More specifically, the destination MAC address has to be replaced with the MAC address of the VM interface. The rule to perform this action is automatically inserted in Open vSwitch by the OpenDaylight module.

The Neutron traffic steering plugin extends the Neutron database and adds two tables to store the steering classifiers and the port chains. Similar to other Neutron plugins, the traffic steering functionality is exposed in Neutron python client, Neutron command-line client and Neutron REST API. See Table 4-4 for a list of available operations.

Command line	URI	HTTP Verb	Description
steering-classifier-create	*/classifiers*	POST	Create a traffic steering classifier
steering-classifier-delete	*/classifier/{id}*	DELETE	Delete a given classifier
steering-classifier-list	*/classifiers*	GET	List traffic steering classifiers that belong to a given tenant
steering-classifier-show	*/classifier/{id}*	GET	Show information of a given classifier
steering-classifier-update	*/classifier/{id}*	PUT	Update a given classifier
port-chain-create	*/port_chains*	POST	Create a port chain
port-chain-delete	*/port_chain/{id}*	DELETE	Delete a port chain
port-chain-list	*/port_chains*	GET	List port chains that belong to a given tenant
port-chain-show	*/port_chain/{id}*	GET	Show information of a given port chain
port-chain-update	*/port_chain/{id}*	PUT	Update a port chain

Table 4-4 Traffic Steering available operations

Defining a forwarding graph using the Neutron Traffic Steering extension

It is up to the user to instantiate the network service by provisioning a machine and configuring the network service software in it, making sure it is attached to a Neutron network.

In the following example, a network configuration is implemented in OpenStack, as represented in Figure 4-9.

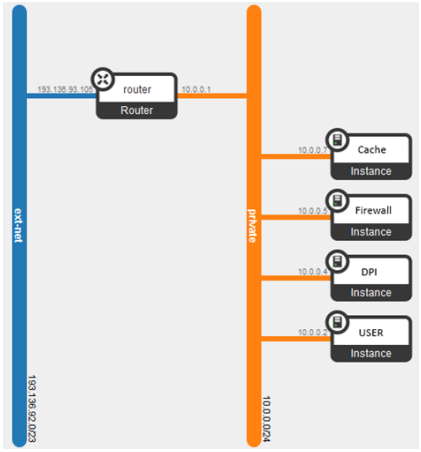


Figure 4-9 Example of OpenStack configuration

Using this setup in OpenStack a forwarding chain will be created (see Figure 4-10), which only applies to traffic matching the following classifier:

- HTTP or FTP traffic type
- Sent by the host USER (10.0.0.2)

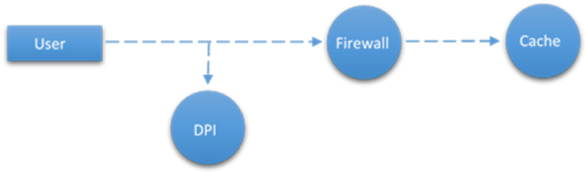


Figure 4-10 Forwarding graph

The classifier needs to be created first to be referenced during the port chain creation:

```
neutron steering-classifier-create --name FTP --protocol tcp --dst-ports 20:21 --src-ip 10.0.0.2

neutron steering-classifier-create --name HTTP --protocol tcp --dst-ports 80 --src-ip 10.0.0.2
```

Finally the port chain is created:

```
neutron port-chain --name USER-DPI_AND_FW-CACHE --classifiers FTP,HTTP --ports "user_port:dpi_port,fw_port" --ports "fw_port:cache_port"
```

The traffic originated from the host USER with the matching classifier FTP or classifier HTTP will be steered to port fw_port and also to port dpi_port (the packet is replicated to the two ports). When packets reach the Firewall, they are forwarded, because the Firewall is not the final destination. After leaving the Firewall they can be captured in OVS to be steered to the cache_port. If the Cache host is not the intended destination, the packets are once again forwarded (repeating the behavior previously described for the Firewall host). When the chain is complete the packet follows its normal path.

The following constraints were identified during the development of this module:

- The bundle was only tested with the first version of OpenDaylight software (Hydrogen) because it was the only available at the time. Before this work can be used in the new ODL releases, some updates of the source code are needed. Also, due to the same reasons it only works with Openflow version 1.0.
- OpenDaylight Hydrogen version has some limitations when creating provider networks in OpenStack.
- Because OpenDaylight does not have its own database, once the application is terminated the existing runtime data is lost.

Steering classifiers can only be created using NeutronPorts UUIDs. Because of this, incorporating a host located outside of OpenStack in the chain may lead to

unpredictable behavior, due to the fact that a virtual node is necessary to receive all traffic sent by the real host.

4.3.3.2. Source routing for service chaining on datacenter network

It is a fact that service chaining requires the installation of flow entries in switches located in datacenters, such that traffic will traverse the NFs in the exact order specified in the service chain. This requirement generates the need to install a large number of flow entries in switches, especially with an increasing number of customers. This, in turn, can raise a data scalability issue for the T-NOVA system, as datacenter switches typically have relatively small flow table size (i.e., several thousand entries).

To mitigate this problem, an alternative solution employs source routing to steer traffic through the NFs of a service chain. Source routing embed the path of each service chain on the traffic packets headers obviating the need to install flow entries in DC switches, i.e., switches forward packets based on the path information carried by the packet header. Source routing is an attractive solution for datacenters where the number of switches per path is relatively small (typically, there are three switches between access gateways and any server in a DC) in comparison to ISP and enterprise networks. Furthermore, given that it is performed within the DC and under the control of the DC operator, source routing raises less security concerns. For instance, only switches and hypervisors (on virtualised servers) managed by the DC operator can insert or remove paths from packet headers. However, source routing raises a set of challenges in terms of scalability and performance. In particular, the embedded path might increase the packet size beyond maximum allowed length (e.g. 1500 bytes for Ethernet packets). Therefore, we need to minimize the number of bytes consumed to perform source routing per packet. This might entail a trade-off between minimizing the state per switch and the source routing header size. Furthermore, source routing should provide forwarding rates that are as high as rule-based forwarding.

Architecture overview

Consider the example in Figure 4-11 where traffic needs to be steered through three NFs deployed on servers within a DC to form a service chain (i.e. traffic should traverse NFs in particular order). A straightforward approach is for each service chain to install a forwarding entry in each switch on the path connecting the NFs. Despite the simplicity of this approach, it requires maintaining the state of each service chain on multiple switches (in this example we need 7 entries for one chain) which limits the number of service chains that can be deployed on a DC (since the flow table size of each switch is relatively small). On other hand, by using source routing, the path between the NFs can be embedded on the packets header as they arrive to the DC (at root switches) requiring no state maintenance on each switch on the path.

We propose to embed a sequence of switches output port numbers on each packet such that each port in this sequence corresponds to a switch on the path. For instance, to steer traffic through NF1 and NF2, each packet should carry the sequence

of port numbers 1, 3, 5. By reading the port number, the switch learns to which output port, it should forward the packet (e.g., switch A will extract and forward through port number 1). To allow the switches to identify their corresponding port numbers, we add a counter field to the routing header. This counter identifies the next port number to be read by the next switch on the path, i.e., the counter field specifies the location of the switch corresponding port number from the beginning of the routing header. For example, when reaching switch B, the counter will carry the value of 6 indicating that switch B should read the six field on the port number starting from the beginning of the routing header (Figure 4-11). The value of the counter is decremented at each switch to indicate the next port number to be read.

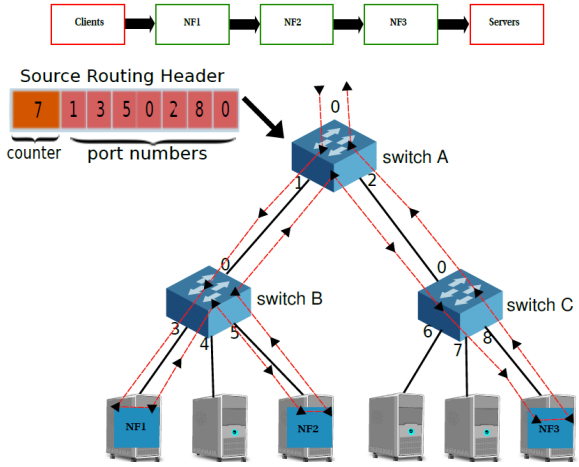


Figure 4-11 Example of service chaining through source routing

To this end, a SDN architecture has been developed for service chaining through source routing. The architecture consists of four main components (Figure 4-12):

1. **Access switch:** is an Openflow switch which inserts the source routing header on each arriving packet based on the configuration provided by the source routing controller. In DC, root switches can play the role of access switches. Typically, a DC has multiple root switches which enables balancing routing header insertion load across multiple access switches.
2. **Source routing controller:** provides: (i) *topology discovery* to keep track of the different links and switches on DC network, (ii) *path-to-ports translation* to identify the corresponding switch output ports which form the path between NFs, and (iii) *flow table configuration* to install flows on the access switch to embed source routing headers on arriving packets.

3. **Source routing switch:** extracts the output port from the source routing header and accordingly forwards the arriving packet. Aggregation and access switches can play the role of source routing switches.
4. **Southbound interface:** provides an interface for the controller to install forwarding rules on access switches and perform topology discovery (i.e., collect information about the switches port numbers and active links).
5. **Northbound interface:** enables the NFV orchestrator to submit the service chain assignment (the assignment of NFs to servers and DC paths) to the source routing controller.

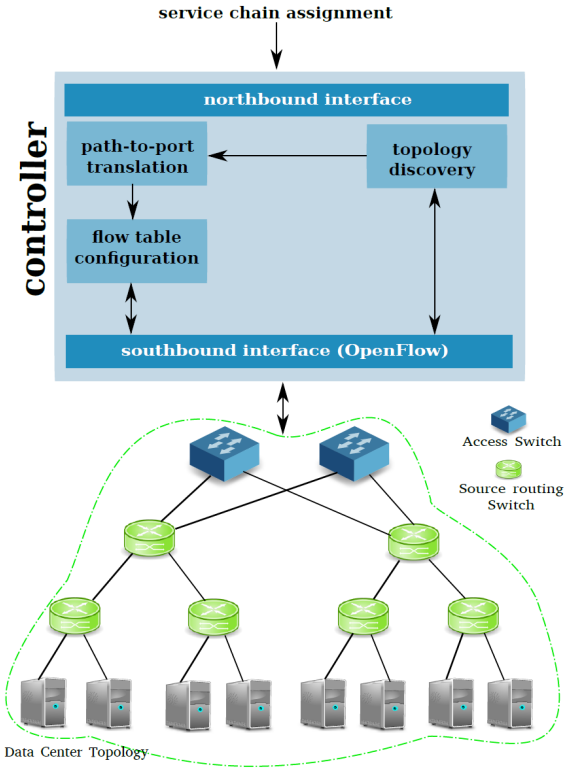


Figure 4-12 Source routing architecture

Embedding the port numbers of the whole path on the arriving packets might significantly increase the packet size leading to high bandwidth consumption and/or larger packet sizes beyond the maximum length. For instance, a service chain consisting of 10 NFs which are assigned to different racks requires a routing header with 30 port numbers. Assuming each port number is carried on 8-bits field, a routing header adds 30 extra bytes to each arriving packets. This leads to the consumption of 46% more bandwidth for packets with the size of 64 bytes and to the need for fragmentation for packets with a size bigger than 1470 bytes (with Ethernet link). To

overcome this problem, we propose using more than one access switch along the path. In this respect, the service chain is divided into pathlets where each pathlet starts and ends with an access switch. The location and number of access switches is identified by the DC operator based on real world service chains deployment and the employed embedding algorithm. By embedding NFs in as small as possible number of servers and racks, less number of ports is required for routing traffic and subsequently, less number of access switches. An alternative approach is to deploy both access switch and source routing functionality within all DC switches, allowing the DC operator to dynamically install routing headers on packets in different locations of the network. To identify packets reaching the end of their pathlet, we use the zero value of the counter field (indicating no further ports to read). While this approach provides more flexibility, it increases the complexity of the DC switch design.

Implementation

In this section, we present the implementation of our architecture components:

Source routing header: To embed the routing header on the service chain packets, we use the destination MAC address and further add a VLAN tag and an MPLS label stack to each packet. In particular, we encode the port numbers in the destination MAC address, the VLAN ID and the MPLS label (Openflow 1.0 which we use for our implementation does not allow modifying other VLAN and MPLS fields). By combining these fields, we can store 10 port numbers per packet, where each field is 8 bit long and supports switches with up to 256 ports. To store the counter value, we use the TTL field of the MPLS header. We can further increase the number of embedded ports by inserting more MPLS stack labels in each packet.

Source routing controller: We use POX [POX] to implement the different components of our controller. Using Openflow, the controller collects and stores the port numbers of each switch. Based on this information, the controller translates the service chain path to switch ports. After the translation, the controller ensemble the port numbers into a bit vector. This bit vector is further broken down into the MAC destination address, the VLAN ID and the MPLS label. This step is followed by installing a flow entry in the access switch using OFPT_FLOW_MOD message. This message carries the flow matching fields (e.g., source/destination IP, source/destination port numbers and protocol) as well as the physical output port number.

Access switch: We rely on OpenvSwitch [PPK+09] to embed routing headers on the arriving packets. OpenvSwitch exposes an Openflow interface to the controller to install forwarding rules which insert the routing headers on the service chain packets by adding a VLAN tag and an MPLS label stack to each packet and updating its destination MAC address.

Source routing switch: We extend Click Modular Router [KMC+00] with a new element, *SourceRouter*, which extracts the values of the VLAN ID, MPLS label, MPLS TTL and the destination MAC address and subsequently combines them into the routing header (see Figure 4-11). Based on the counter value, the element reads the corresponding port number through which the packet is forwarded. By combining

our element with Click elements for packet I/O, we implement a source routing switch.

Evaluation

We evaluate our source routing switch on a emulab-based testbed using 3 servers, each one equipped with an Intel Xeon E5520 quad-core CPU at 2.26 GHz, 6 GB DDR3 RAM and a quad 1G port network interface cards (NICs) based on Intel 82571EB. All servers run Click Modular router 2.0 on kernel space with Linux kernel 2.6.32. We deploy our source routing switch on one server and use the other two as source and destination to generate and receive traffic, respectively. Since the switch on our testbed filters packets with VLAN and MPLS header, we encapsulate our packets in IP-in-IP header. We measure the packet forwarding rate of our switch with various packet input rates and packet size of 85 bytes including the routing header and the IP encapsulation header. We further compare our switch performance with rule-based forwarding where we forward packets based on packets' destination IP address using a routing table with 380K entries. Figure 4-13 shows that our source routing switch achieves more than 30% higher forwarding rate than rule-based routing. This performance is achieved using a single CPU core.

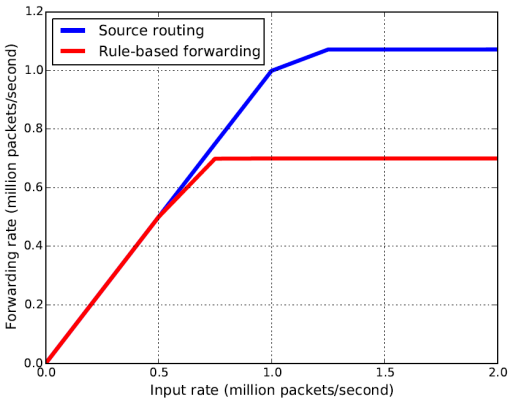


Figure 4-13 Source routing switch forwarding rate

To evaluate the performance of our controller, we add another server with the same specifications to our setup (4 servers in total). We use two servers as traffic source and sink and the other two to host the controller and the access switch. Initially, we measure the flow setup time (i.e., the time required to insert the source routing header at the access switch) which we define as the time elapsed from the flow's first packet arrival at the access switch ingress port till its departure from the egress port.

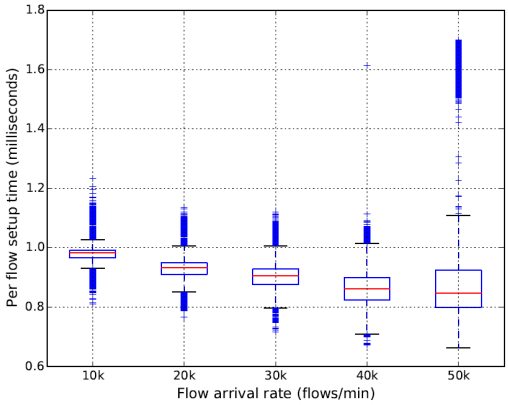


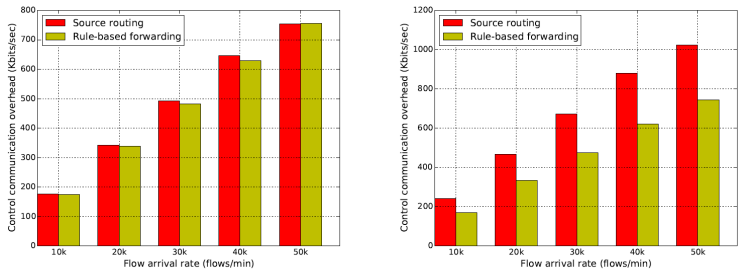
Figure 4-14 Setup time per flow.

As depicted in Figure 4-14, the flow setup time does not change significantly across the various flow arrival rates. We further break down the setup time into multiple components. In particular, we measure the time required for the source routing header computation, the time the control packet takes to traverse the POX and kernel stack on the controller server in both directions (between the controller and the switch), the RTT (between the controller and the access switch) and the access switch processing time. As shown in Table 4-5, source routing header computation consumes less than 29% of the total setup time. Instead, most of the setup time is spent in POX, kernel stack, and the access switch processing.

Table 4-5 Flow setup time

Element	Time (milliseconds)
Source routing header computation	0.25
POX processing + kernel stack	0.24
RTT	0.1
access switch processing	0.3
Total	0.89

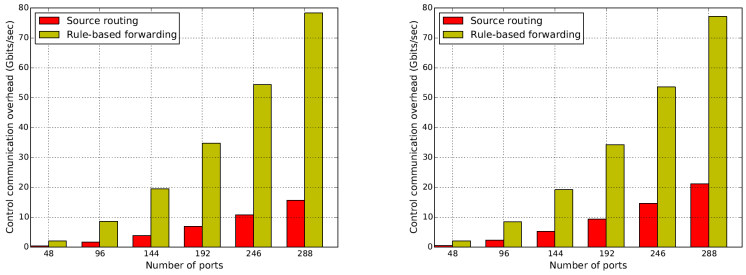
We also compare source routing with rule-based forwarding in terms of control communication overhead. We first measure the communication overhead on a single switch for different flow arrival rates. In this respect, we measure the control traffic between the switch and controller per direction (noted as uplink and downlink overhead). Our measurements show that both source routing and rule-based forwarding consume the same amount of bandwidth at the uplink (Figure 4-15 (a)), since both approaches use the same packet size and format (i.e., *OFPT_PACKET_IN*) to transfer the packet fields to the controller. On the other hand, for downlink, source routing consumes more bandwidth than rule-based forwarding (Figure 4-15 (b)), due to the extra packet fields (i.e., VLAN and MPLS) required to install the source routing header.



(a) Uplink overhead (b) Downlink overhead

Figure 4-15 Control overhead for a single switch.

Using the results we obtained for a single switch, we further calculate the communication overhead with a diverse number of switches. For our calculation, we consider DCs with a fat-tree topology [FAT TREE]. The DC components in a fat-tree topology can be modeled based on the number of ports per switch. More precisely, a fat-tree topology with k-port switches has $(5(k^2))/4)$ switches where $k^2/4$ of these switches are core switches and k^2 are aggregation and edge switches.



(a) Uplink overhead (b) Downlink overhead

Figure 4-16 Control overhead for multiple switches.

We calculate the control overhead of source routing and rule-based forwarding for fat-tree topologies with a diverse number of ports per switch. As shown in Figure 4-16, source-routing introduces significantly lower communication overhead in comparison to rule-based forwarding. We observe that the savings in communication overhead increase with the size of the DC network, since the source routing controller needs to communicate only with the core switches of the DC.

4.4. Load balancing in multicontroller scenarios

The SDN architectural model introduces a single logical node in charge of controlling all the data-path layer behaviour. The following section deals with scenarios where multiple controllers cooperate in a cluster, in order to provide scaling capabilities and avoid failure issues.

In this context, a load balancing algorithm has been proposed, as well as different implementation approaches, with the aim of managing the control traffic between switches and multiple controllers. Such algorithm aims at balancing the switch-to-controller connections according to the current controllers' load estimation. Specifically it has to assign each switch to a subset of controllers, of which one will be the master, in order to reduce the control traffic load while maintaining resiliency. By balancing the load of the controllers, fair exploitation of the control resources is achieved which, in turn, increases the overall throughput and minimizes the control connection latency.

4.4.1. Clustering Service in OpenDaylight

The OpenDaylight controller supports a cluster-based HA model where several instances of controllers act as a single logical controller, while the global state of the network is maintained through a distributed datastore.

In the Hydrogen release, OpenDaylight included the Clustering Service Provider module for this purpose. It provides clustering services to all the functional components of the controller as well as to applications running on top of the controller. From the northbound side the cluster is accessible via RESTful API and each request can land in any controller in the cluster. From southbound, Openflow switches would need to explicitly connect to the controllers in the cluster via their IP address. In this regard, the Connection Manager is in charge of managing connections between the ODL instances and the OF switches. For the time being, the connection schemes supported are: *SINGLE_CONTROLLER* (all the switches connected to only one controller) and *ANY_CONTROLLER_ONE_MASTER* (any switch connected to any controller, with only one master). Other connection schemes (i.e. *ROUND_ROBIN* and *LOAD_BALANCED*) were defined but not yet implemented. In this regard, one implementation approach proposed in T-NOVA aims at extending the Clustering Service offered by ODL Hydrogen with a load balancing algorithm implementing the *LOAD_BALANCED* connection scheme.

Starting from Helium, OpenDaylight moved to the popular Akka [AKKA] technology to operate in a server cluster configuration, thus dismissing the previous clustering implementation. This feature, installed through Karaf, replaces the non-clustered datastore access methods with methods that replicate the datastore transaction within the configured cluster. The cluster, consisting of at least three physical server nodes, is configured to enable coordination between the member nodes. Once the configuration process is completed, a cluster leader is elected according to the RAFT [RAFT] convergence process, using the Gossip protocol. Therefore, in order to integrate load balancing into ODL Helium (and beyond), an alternative implementation has been carried out in T-NOVA. This new approach leverages on the OpenFlow protocol, which, in version 1.3 [OF1.3], introduced the concept of Role (*EQUAL*/*MASTER*/*SLAVE*) of controllers. The main goal is to balance control traffic by dynamically assigning the *MASTER* controller of each switch, as described in Section 4.4.6.

4.4.2. Experimental plan

In order to evaluate the benefits of having multiple controller instances, a preliminary test environment has been setup, using Mininet and OpenDayLight (Hydrogen release).

The testing environment in Figure 4-15 is composed by a SDN controller cluster of 2 instances and 26 virtual switches created using OpenvSwitch [OVS]. Every switch Si is connected to the Si+1 switch and to the host Hi that simulates the data traffic generator. The control traffic is generated by the switch when there is no flow entry for an incoming data packet that the host wants to send. Hence, the switch encapsulates the data packet in a OF control packet (packet-in) and sends it to its controllers. Then, exactly one controller should send a reply message containing the flow entry that the switch will install in its table. Let flow-mod be the name of these messages.

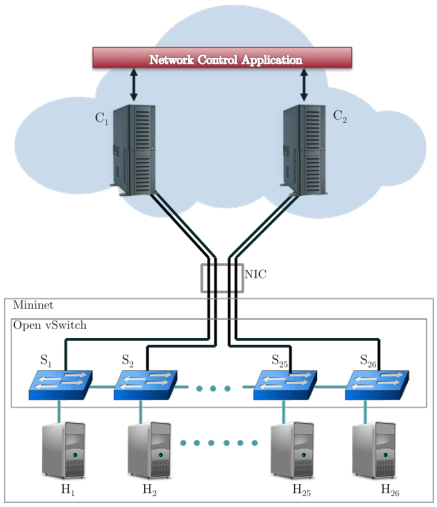
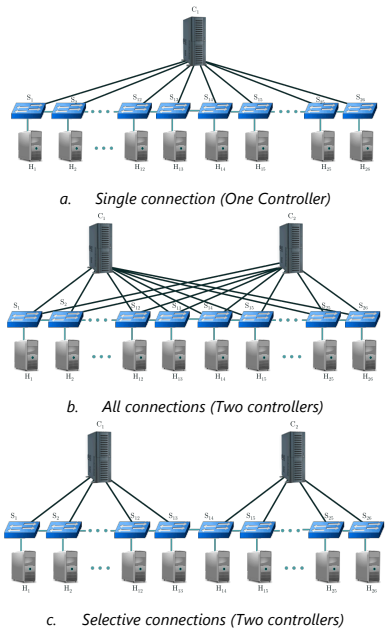


Figure 4-15 Testing environment

We performed the test using the following physical devices: n.1 Quad core Intel Q8300 machine with 4 GB RAM and a gigabit Ethernet card hosting the Mininet network emulator and the *tcpdump* packet sniffer; n. 2 Quad core AMD Athlon X4 750K with 8 GB RAM and a gigabit Ethernet card, both running the OpenDaylight Hydrogen controller; n. 1 gigabit switch. The performance is measured in the machine with Mininet as all packet-in and flow-mod messages pass through its network interface (see Figure 4-15).

In Figure 4-16 the testing scenarios are presented. In the configuration (a) all the switches are connected to one instance of controller.

The scenario *(b)* is interesting to see what happens when every switch sends the *packet-in* control message to more than one controllers. One controller should answer to the request while the other one should ignore it. However, the ignoring decision takes some computational resources and we will see the its impact. In *(c)* there are two controllers and the first 13 switches are connected to the first one while the remaining ones are connected to the second controller.



a. Single connection (One Controller)

b. All connections (Two controllers)

c. Selective connections (Two controllers)

Figure 4-16 Testing scenarios

Figure 4-17 depicts the 95-percentile response time of the 3 cases described above.

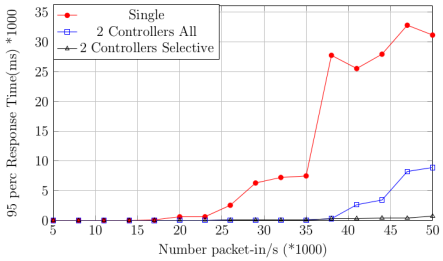


Figure 4-17 95-percentile response time comparison between the different configurations

We chose to measure the 95 percentile response time instead of the average because the latter does not take into account spikes if they are less than 5%. Another reason is that the average does not always report the real behavior since it may hide a significant higher response time for an important part of the requests. For example, a 15 milliseconds 95 percentile response time means that 95% of all responses were processed in 15 milliseconds or less.

We noticed that the difference between the All and the Selective connection case becomes relevant when the number of packet-in/s is greater than 40,000. At 50,000 packet-in/s this difference becomes more significant (about 8 seconds).

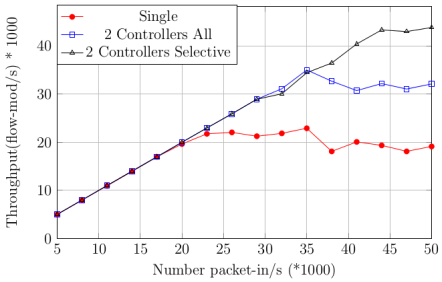


Figure 4-18 Throughput comparison between different configurations

Similar results are reported in Figure 4-18. When the network load is high (above 45000 packets/s) the throughput of the Selective test is the highest one, being 66% greater than the one in the All test scenario and 100% greater than the one in the Single test scenario. These plots clearly highlight the benefits of a control plane with multiple controllers. It is easy to observe that relevant performance boost is obtained only under certain level of load. Hence it is important to find the load levels that should trigger an increment or a decrease of the number of controllers. Moreover, shrinking the cluster when the load drops under some decided value may reduce the operational cost.

The performance gain highlighted by the tests represents a good motivation for using multiple controllers. They indicate that increasing the cluster size may improve the performance but this could have relevant results only under certain level of load.

4.4.3. Load balancing algorithm

As mentioned before, OpenDaylight controller supports several connection schemes between control and data planes, i.e. each switch could be statically connected to one, more than one or all of the controllers. In this section, we propose a load balanced approach providing a dynamic selective switch-controller connection scheme.

The balancing algorithm dynamically decides the switch-controller mapping aiming at optimizing the resource utilisation. In the clustered SDN control plane, at any moment there is an elected leader (i.e. an instance of Controller), that performs both monitoring and migration operations. It collects load information about every Controller in the cluster and then uses it to decide which switches should be moved to different controllers. In this implemented version, the monitored data consist of the average CPU load and the number of Openflow messages received from every switch that the specific Controller is connected to, in a closed time interval.

It should be taken into account that the decision of finding the best mapping could lead to a migration of a very large number of switches, moreover, it would further slowdown the system due to the computations needed to solve it, so is preferable to migrate a small number of switches. The benefits obtained from the optimal solution are not always significant, a 10% imbalance will be hardly noticed in the system response time. It should also be considered that the traffic could be highly dynamic and the perfect solution could be useless because in a very short period of time the load will change and that will trigger a new load balancing operation. The Load Balancing algorithm tackles the problem of a potential large number of switch migrations and avoids the migration of switches that have a load impact smaller than a defined threshold.

The algorithm receives as input the average load and statistics about every controller. By using these monitoring data and a predefined activation threshold (THRESHOLD BALANCE), the algorithm compute two set of controller, namely the overloaded and the non-overloaded Controllers.

From the data plane point of view, the switches are assigned to classes of load. The load generated by one switch is given by the fraction of number of messages that it has sent and the total number of messages that the controller has received from all its switches.

At the core of the algorithm, given a non-overloaded controller C_{dst}, the algorithm tries to act a best-fit technique by transferring switches from an overloaded controller C_{src} to C_{dst}, until its load becomes at most the average plus half of the THRESHOLD BALANCE. This additional factor aims to decrease the problems caused by fragmentation.

```
Algorithm    Load Balancing Algorithm
overloaded ← ∅
nonOverloaded ← ∅
foreach  contr in loadEveryContr do
    if contr.load > avgLoad + THRESHOLD_BALANCE then
        overloaded ← overloaded ∪ {contr}
    else
        nonOverloaded ← nonOverloaded ∪ {contr}
    end
end
balanceGetMigrations (avgLoad,loadEveryContr ) begin
    migrations ← ∅ overloaded
    /* dividing the switches in load classes                    */
    mapLoadSwitches ← [ ]
    foreach  contr in overloaded do
        foreach  switch in contr.connectedSwitches do
            listSwContr= get value associated with key=switch.loadGenrated_contr
            add (switch,contr) to listSwContr
            put (switch.loadGenrated_contr,listSwContr)
        end
    end
    foreach  contr in nonOverloaded do
        availabDst ← avgLoad − contr.load + THRESHOLD_BALANCE / 2;
        if availableDst < THRESHOLD_MIGR_SINGLE_SW then
            continue
        end
        while it is possible to transfer more load to contr do
            (bestSw,src) ← getMostLoadingSwitchUnder(availableDst,
            mapLoadSwitches)
            migrations ← migrations ∪ (src,dst,bestSw)
            availabDst ← availabDst − bestSw.loadGenerated_src
            remove entry {bestSw} from the set associated to key=load_bestSw
            from mapLoadSwitches
        end
    end
```

Figure 4-19 Load Balancing Algorithm

4.4.3.1. Switch migration

The number of exchanged control messages between a switch and a controller is variable and often unpredictable, it depends on the hosts that connect to this switch. Hence, a switch can migrate from one controller to another in such a way that the system load remains balanced. Such switch migration feature is fundamental for the load balancing functionality.

The load balancing algorithm generates a set of switch migration operations. Figure 4-20 represents a migration of a switch (X) between two controllers (A and B).

The implemented migration algorithm uses the Openflow 1.0 feature. Such protocol version implementation does not support roles, however, in the following sections we refer to "master" as the controller that should handle all asynchronous Openflow messages coming from a certain switch. We assume there is a distributed cache

replicated in all controllers that is mainly used to establish which controller must answer to asynchronous messages.

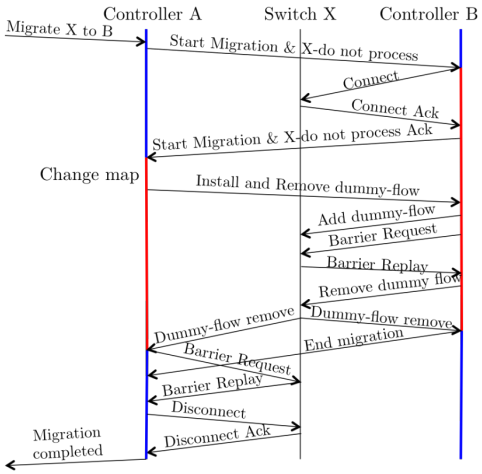


Figure 4-20 Migration protocol

The dummy flow is a flow that can be easily distinguished from the normal flows which are installed in the switches. When it is received by a switch it changes its master. In particular, the match that identifies the dummy-flow have the initial master and final master as source and destination addresses. The concept of dummy-flow has also been proposed in [DHM+13].

As shown in Figure 4-20, the algorithm starts with the "Start Migration & X-Do not process" message. B should remember that during this migration should not process the asynchronous messages received from X until it receives the dummy-flow removal message. Controller B should also connect to the switch X. When A receives the acknowledgment for the previous operations, it should instead remember to process the asynchronous messages and right after that A changes the distributed map. Then it requests to B to go ahead and install and remove the dummy-flow. Controller B does it and when X removes and notifies the removal, both controller will receive and process this event and each of them will continue to use the changed map where B is the master of X.

Such migration protocol guaranties that no duplicated flows are installed and that there is always one controller that answers to packet-in messages.

4.4.4. Implementation

The first implementation of the Load Balancer makes use of features built in the OpenDaylight Hydrogen release. Specifically, the following bundles/API are used and extended:

- Connection Manager - It manages the 'role' of each controller instance in the cluster. Only two connection schemes were available, namely *Single Controller* and *Any Controller*. Here the *Selective Controller* scheme, implemented by the switch migration algorithm was added.
- Openflow Plugin - It is the module that programs the Openflow switches throught the Openflow protocol 1.0.
- OVSDB Plugin - The module responsible for connecting the switches to the controllers, for their configuration and for disconnecting them. Such plugin was enriched with a function that is able to disconnect a switch from the controller. The OVSDB API set were updated with this API and another able to allow to connect a switch to another controller.

In addition to the above bundles, a new one, carrying on the Load Balancer logic was developed, responsible for taking all the balancing-related decisions.

Figure 4-21 highlights the interactions among all of the involved bundles.

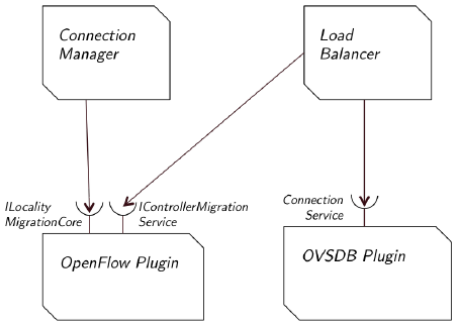


Figure 4-21 Main interactions among the bundles of interest in OpenDaylight

A migration is requested by the cluster coordinator (or cluster leader), that is also the one that collects all the load statistics and decides which switch should change its master.

Figure 4-22 illustrates how the controllers A (initial Master), B (final Master), and C (coordinator) communicate between them. Controller A or B could also be the cluster coordinator. In the figure there is the interaction among system components during a switch migration. From the Load Balancer bundle, the cluster coordinator contacts the TCP migration servers in the controller A that is overloaded and in controller B that is not overloaded. The OVSDB Plugin connects and disconnects (if needed) the switch that is being migrated.

Every controller has a migration server that receives migration requests from the coordinator. It is implemented and runs in the Openflow Plugin bundle. The TCP migration server can receive "Start migration" and "Install and remove dummy-flow".

The TCP migration client is the one that makes the migrations requests. It is implemented and run in the Load Balancer bundle. For every migration there are 2 clients: one that communicates with the current master and one with the target master.

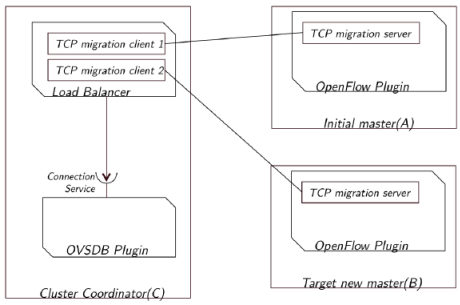


Figure 4-22 Bundles interaction during a switch migration

4.4.5. Evaluation results

This section outlines the test performed by the Load Balancer to measure the overall performance in specific conditions. In the following, the response time before and after the activation of the Load Balancer algorithm is plotted.

In every test, the system is imbalanced in the first half of the total test time, then the load balancing is activated, this in order to measure the improvements brought by the auto-balancing. We measure the 95 percentile response time before and after the activation. For the sake of simplicity, every switch sends an equal amount of packet-in (reported on the X axis).

The test results are illustrated in Figure 4-23.

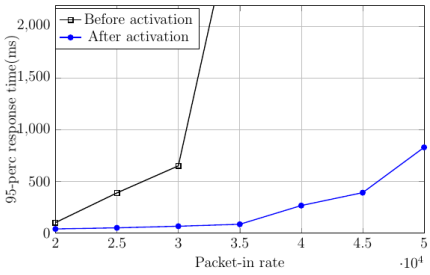


Figure 4-23 Mean response time comparison before and after load balancing event

The differences between the two scenarios are noticeable, such differences become huge when the controllers - without Load Balancer - reach their point of saturation (ie they process the max number of Openflow messages per second). In the latter

scenario, the Load Balancer algorithm aims to increase the overall controller cluster throughput by distributing the load generated by a single switch across the instances.

4.4.6. Role-based load balancing

4.4.6.1. Overview

In order to endow more recent OpenDaylight releases (Helium and beyond) with load balancing capabilities, an alternative solution has been implemented. It leverages the Openflow protocol which, starting from version 1.3, regulated a new architectural model by introducing the concept of role (MASTER, SLAVE and EQUAL) of a controller.

This new model enables two modes of operations when multiple controllers connect to the switches: *equal* and *master/slave*. In *equal* interaction, there can be more than one controller with EQUAL role for a switch. These controllers may receive and respond to any of the events belonging to that switch. On the contrary, in *master/slave* interaction, for each switch, only one MASTER, responsible for all the events corresponding to that switch, must exist, whereas the SLAVE controllers can be more than one but they do not receive any events.

The OpenDaylight controller supports both *equal* and *master/slave* operations. Specifically, the Openflow Plugin is the module in charge of statically assigning the controllers' roles and providing mechanisms to manage this information within the cluster (Figure 4-24)

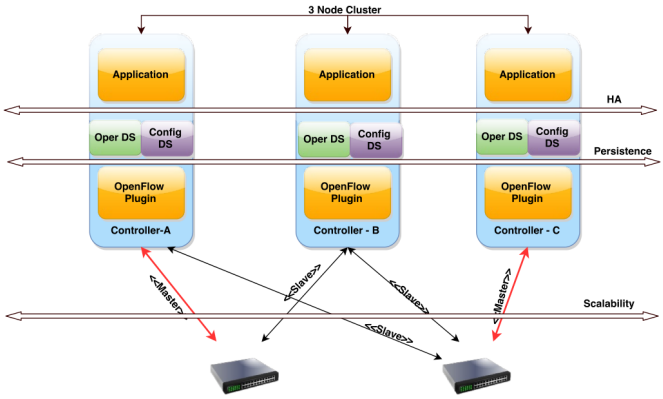


Figure 4-24 Openflow Roles in OpenDaylight

Following the role-based approach, the proposed implementation aims at dynamically changing the controllers' roles (MASTER/SLAVE) of each switch so as to balance the cluster workload, in an easier way than connecting/disconnecting switches using OVSDB (as done in the previous implementation). This makes the migration mechanism more efficient and, at the same time, reduces significantly the network control traffic.

4.4.6.2. Implementation

The implementation of the role-based load balancing relies on existing features built in the OpenDaylight Lithium, together with components developed from scratch.

The reference scenario is depicted in Figure 4-25.

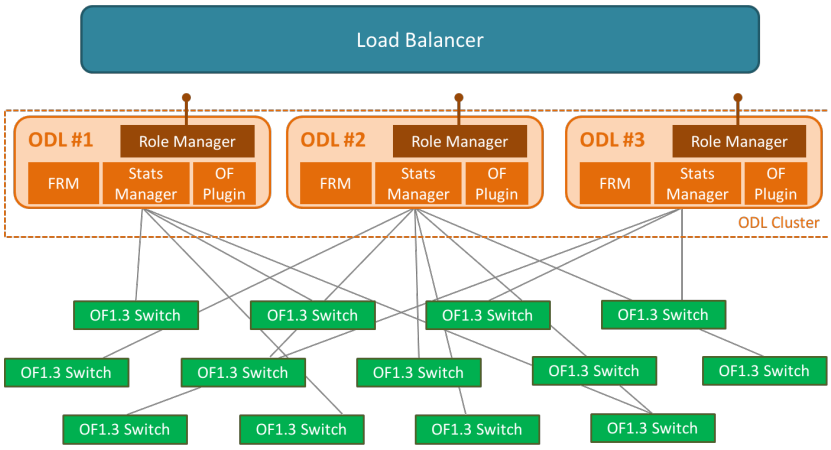


Figure 4-25 Load balancing reference scenario

Specifically, the following existing ODL features have been used and extended:

- *OpenFlow Plugin (OF Plugin)*
  This module has been extended to allow the configuration of the switch ownership on-demand and to manage external Role Messages through Java interfaces
- *Forwarding Rule Manager (FRM)*
  This module has been extended to support the on-demand switch ownership change in order to re-route internally a request
- *Statistics Manager (Stats Manager)*
  The Statistics has been enhanced to collect and expose per-switch OpenFlow statistics.

In addition, the following components have been implemented:

- *Role Manager*
  The Role Manager is an ODL bundle which exposes RESTful API to northbound applications and, in particular, to the Load Balancer for configuring the controller role and providing statistics data of each switch. The description of the REST interface is available in the table below:

URI	Method	Parameter	Description
/set-switch-role	*POST*	*Switch IDs, OpenFlow-Role*	*Enforces the given OpenFlow-Role for the target switches on the invoked controller instance.*
/get-switch-role	*POST*	*Switch IDs*	*Returns the OpenFlow-Role of the target switches for the invoked controller instance.*
/get-switch-stats	*POST*	*Switch IDs*	*Returns aggregated per-switch OpenFlow statistics (messages sent/received) of the target switches for the invoked controller instance.*

Table 4-6 Role Manager APIs

- *Load Balancer*
  The Load Balancer is a stand-alone Java application in charge of monitoring each controller of the cluster and determining the best controller-to-switch mapping according to the workload. It also provides a web GUI to display the roles mapping assignments within the cluster.

4.4.6.3. Evaluation

The role-based implementation of the load balancer has been evaluated on a test-bed composed by a cluster of 3 OpenDaylight controllers connected to a Mininet network of 16 Openflow (v1.3) switches. The aim of the test was to demonstrate the load balancer capability of providing the switch-to-controller mapping that equally distributes the workload of the cluster.

Once started, the Load Balancer triggers the Stats Manager of each controller to get updated information about the machine resources usage (CPU/RAM) and the exchanged Openflow traffic. Then it computes the best switch-to-controller mapping on the basis of the load information, as detailed in Section 4.4.3. Finally, it applies the mapping by dynamically changing the controller roles for each switch, through the Role Manager interfaces.

Figure 4-26 shows the assignments statically done by the ODL cluster when it starts and connects to the Mininet network. Each blue block represents a running controller instance, including the switches to which the instance is connected. The light yellow items represent Openflow switches, whose controller has a SLAVE role, while the green items are switches whose controller is MASTER. In the case represented in Figure 4-26, the network switches are connected to all the controllers belonging to the cluster but the first controller acts passively since it has no switches under its (master) control.

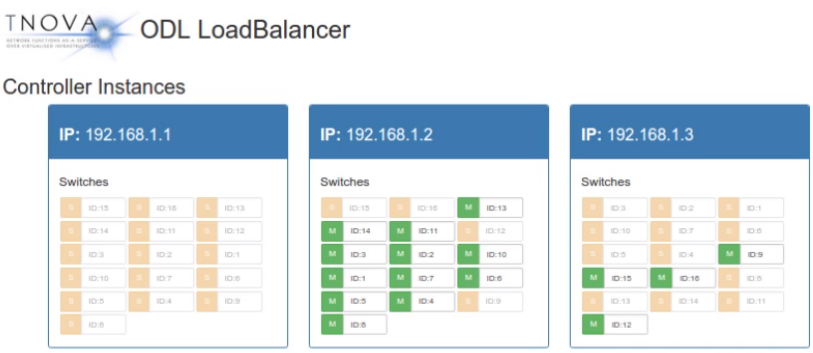


Figure 4-26 Load Balancer Web GUI (before balancing)

Figure 4-27 outlines the role assignment after the activation of the load balancer. Specifically, a subset of switches are moved to the first instance of the controller by changing its role, so as to balance the load of the switches on the cluster.

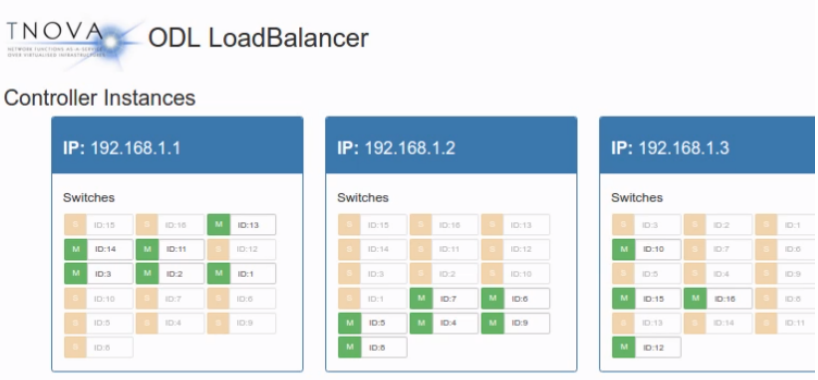


Figure 4-27 Load Balancer Web GUI (after balancing)

Further details on the tests performed, as well as the source code of the developed components, are available at the public repository [LBGIT].

4.5. Network isolation and QoS support

The NFV Infrastructure as envisaged by T-NOVA is a product of integrating Openstack and OpenDaylight. As the current support for network isolation and QoS support are not well integrated, two are the current options that are under test in the frame of Task 4.2. The first option is to use the current integration provided by the neutron ML2 plugin for interfacing of Neutron with OpenDaylight controller or re-use the above solution but also interface directly with the SDN Controller and the OVS instances for reasons that will be explained in this section. The trade-off between the aforementioned solutions is that the first is simpler to implement, as the APIs and interfaces used are between the Orchestrator and those provided by the Openstack,

however the functionalities and QoS capabilities currently supported are somewhat limited. On top of that the network resource management currently supported by the first option is limited to policing ingress traffic at the Cloud Neutron Node external interface. This sections discusses early effort in assessing the situation w.r.t network isolation and QoS support.

4.5.1. Network Isolation

Network isolation for a multitenant environment is the efficient isolation in the network namespace in order different tenants to be able to create isolated networks segments reusing addressing segments operating in a completely isolated manner. Note should be added in the fact that this isolation does not mean strict isolation also in the network resources. This will be attempted through resource management schemes, integrated to future Openstack versions, via integration of QoS mechanisms already available or planned at OpenDaylight (i.e ODL Lithium Reservation Project [ODL-RESERV]) and OVS components.

4.5.1.1. Openstack supported Network Isolation

OpenStack has been designed to be a multi-tenant environment. Users can co-exist within the same OpenStack environment and share compute, storage, and network resources or they can have dedicated compute, storage, and network resources within the same OpenStack environment. A user can create Neutron tenant networks that are completely isolated from any Neutron tenant network created by any other user, even if the users are sharing resources. The network isolation is a feature that does not require intervention from a Systems Administrator. This functionality is possible through the use of Network Namespaces, a feature implemented in the Linux kernel. When two users create two different Neutron tenant networks, a Network Namespace is created for each one. When the users create OpenStack instances and attach those instances to their respective Neutron tenant network, only those instances within the same Network Namespace can communicate with each other, even if the instances are spread across OpenStack compute nodes. This is very similar to having two physical Layer 2 networks that have no way of communicating with each other until a router is put between them.

OpenStack supports network isolation through the use of several mechanisms that ensure the isolation between different networks. Isolation mechanisms that are supported in OpenStack are VLANs (IEEE 802.1Q tagging), VxLANs or L2 tunnels using GRE encapsulation. The isolation technique to be used is configured in the initial setup. When VLAN tagging is used as an isolation mechanism, a VLAN tag is allocated by Neutron from a pre-defined VLAN tags pool and assigned to the newly created network. Packets on the specific network contain IEEE 802.1Q headers with a specific VLAN tag. By provisioning VLAN tags to the networks, Neutron allows the creation of multiple isolated networks on the same physical link. The main difference between OpenStack and other platforms is that the user does not have to deal with allocating and managing VLANs to networks. The VLAN allocation and provisioning is handled by Neutron, which keeps track of the VLAN tags and is responsible for

allocating and reclaiming VLAN tags. Similarly, VxLANs and GRE tunnels are allocated and managed automatically by Neutron.

When OpenStack is integrated with OpenDaylight SDN Controller, the VTN (Virtual Tenant Network) framework can be used in order to control and manage the networking. VTN creates a virtual networking environment, in which each network inside is a different VTN and is managed as an independent network. Features of VTN are:

- Virtual network provisioning
  - Add, remove, modify VTN
  - Add, remove, modify VTN model
- Flow control on virtual network
  - flow filter(pass, abandon, redirect, remarking)
- QoS control on virtual network
  - policing (pass, abandon, penalty)
- Virtual network monitoring
  - Stats info of traffic
  - Failure event

The components used to create such networks are presented and described in the following Table.

Policy Target			Description
VTN			logical representation of tenant network
Virtual node (vNode)	vBridge		logical representation of L2 switch function
	vRouter		logical representation of L3 router function
	vTerminal		Logical representation of virtual node that is connected to an interface mapped to a physical port
	vTunnel		logical representation of Tunnel (consists of vTEPs and vBypass(es))
	vTEP		logical representation of Tunnel End Point (TEP)
	vBypass		logical representation of connectivity between controlled networks
Virtual Interface	Interface		representation of end point on the virtual node (VM, servers, appliance, vBridge, vRouter, etc)

Table 4-7 VTN abstraction components

To create an instance of tenant network, as in the following illustrated in Figure 4-28, the following actions must be performed in the correct order:

- Creation of VTN
- Creation of vBridge
- Creation of interface
- port-mapping

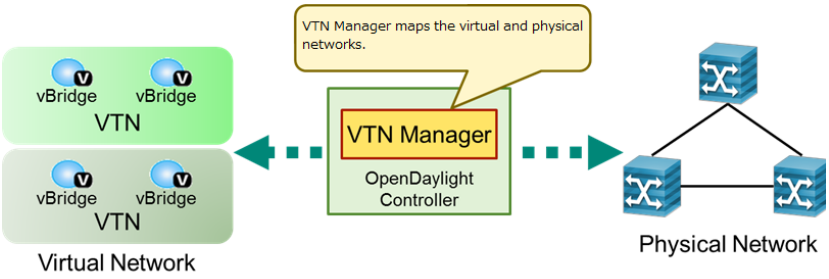


Figure 4-28 VTN Mapping of physical network to virtual networks

There are two types of port mapping available, simple port mapping and VLAN mapping.

Simple port mapping maps the VLAN on physical port of specific switch to vBridge according to the following Figure (Figure 4-29). Physical ports cannot be mapped to physical ports where other OpenFlow switches are connected.

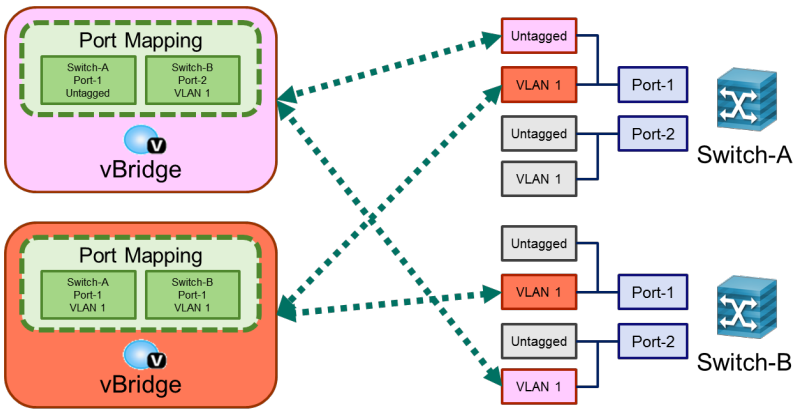


Figure 4-29 Simple VTN port mapping

The VLAN mapping works by mapping any VLAN to the vBridge, as shown in the Figure 4-30. When a physical switch is specified, only the VLAN on specified physical switch is mapped. When a physical switch is not specified, the VLAN on all managed switches are mapped.

A physical port connected to OpenFlow switch is not in scope for VLAN mapping. Also port mapping settings are given priority. VLAN on port mapped physical port is not in scope for VLAN mapping.

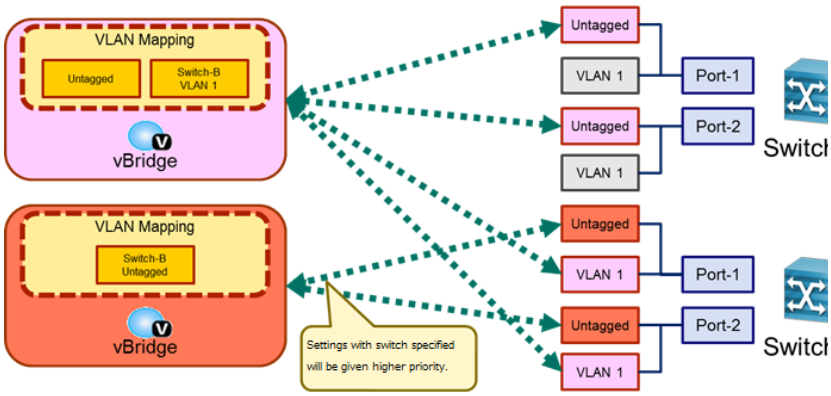


Figure 4-30 VLAN VTN port mapping

After the virtual networks are created and assigned to the physical network, in each virtual interface a flow filter can be applied. Flow filters are used to match specific conditions and apply actions and policies. The figure below depicts two cases of filter usage.

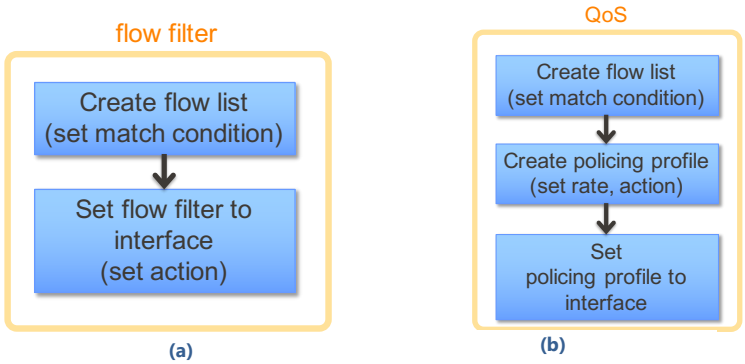


Figure 4-31 Flow filters: (a) Simple filter, (b) QoS filter

4.5.2. QoS Support

4.5.2.1. QoS Support at the OVS level

Since OpenFlow 1.0, queues are supported for rate-limiting egress packets in a switch port for QoS implementation. Queues are designed to provide a guarantee on the rate of flow of packets placed in the queue. As such, different queues at different rates can be used to prioritize "special" traffic over "ordinary" traffic.

Queues, although very useful, are defined outside the OpenFlow protocol. OpenFlow merely provides a wrapper around existing switch queuing mechanisms in order to inform the controller of the available queues. Queues must be defined/instantiated at

the switch using out of band signalling. This means that queues must set up with native OVS commands prior to to using them with OpenFlow. This is analogous to adding ports to an OpenFlow instance or virtual switch.

Moreover, OVS has various ways of providing rate limiting mostly based to the available Linux queue disciplines (qdisc) supported by the Linux kernel. In order to support statistical QoS schemes or support DifServ based schemes the Hierarchical Token Bucket (HTB) scheduler can be used via tc-htb element. Alternatively, the available policers or token bucket (TB) schedulers might be used to provide network resource guarantees for particular flows on designated OVS ports.

4.5.2.2. QoS Support at OpenFlow level

OpenFlow version 1.3 introduced meters support at the OpenFlow protocol. Meters complement the queue framework already in place in OpenFlow, by allowing for the rate-monitoring of traffic prior to output. More specifically, with meters, we can monitor the ingress rate of traffic as defined by a flow. Flows can direct packets to a meter using the goto-meter OpenFlow instruction, where the meter can then perform some operation based on the rate it receives packets.

In turn a configured queue, accepts packets for output and processes them at a min/max specified rate. As such, note that meters and queues are complementary and are not different implementations of the same thing. A common misconception is that meters are a replacement for queues.

Unlike queues though, which are rather rigid and must be defined by the switch out of band (e.g. if using Open vSwitch (OVS) with OVS commands), meters can be installed, modified, and removed at runtime using OpenFlow. In fact, we can link meters to flows themselves. OpenFlow defines an abstraction called a meter table, which simply contains rows of meters. These meters can be manipulated in a similar manner to flows. Also like flows, meters receive packets as input and (optionally) send packets as output.

A meter table consists of meter entries, defining per-flow meters. Per-flow meters enable OpenFlow to implement various simple QoS operations, such as rate-limiting or policing.

A meter measures the rate of packets assigned to it and enables controlling the rate of those packets. Meters are attached directly to flow entries (as opposed to queues which are attached to ports). Any flow entry can specify a meter in its instruction set, the meter measures and controls the rate of the aggregate of all flow entries to which it is attached. Multiple meters can be used in the same table, but in an exclusive way (disjoint set of flow entries). Multiple meters can be used on the same set of packets by using them in successive flow tables.

Each meter entry is identified by its meter identifier and contains:

- meter identifier: a 32-bit unsigned integer uniquely identifying the meter
- meter bands: an unordered list of meter bands, where each meter band specifies the rate of the band and the way to process the packet
- counters: updated when packets are processed by a meter

Each meter may have one or more meter bands. Each band specifies the rate at which the band applies and the way packets should be processed. Packet are processed by a single meter band based on the current measured meter rate, the meter applies the meter band with the highest configured rate that is lower than the current measured rate. If the current rate is lower than any specified meter band rate, no meter band is applied.

Each meter band is identified by its rate and contains:

- band type: defines how packet are processed
- rate: used by the meter to select the meter band, defines the lowest rate at which the band can apply
- counters: updated when packets are processed by a meter band
- type specific arguments: some band types have optional arguments

There is no band type "Required" by this specification. The controller can query the switch about which of the "Optional" meter band types it supports.

- Optional: drop: Drop (discard) the packet. Can be used to define a rate limiter band.
- Optional: dscp remark: decrease the drop precedence of the DSCP field in the IP header of the packet. Can be used to define a simple DiffServ policer.

An example REST API request used at ODL is illustrated below:

```
Using PostMan: Set Request Headers
Content-Type: application/xml
Accept: application/xml
Use URL: http://<ip-address>:8080/restconf/config/opendaylight-
inventory:nodes/node/openflow:1/meter/1
Method:PUT
Request Body:
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<meter xmlns="urn:opendaylight:flow:inventory">
   <container-name>abcd</container-name>
   <flags>meter-burst</flags>
   <meter-band-headers>
         <meter-band-header>
            <band-burst-size>400</band-burst-size>
           <band-id>0</band-id>
           <band-rate>2048</band-rate>
             <dscp-remark-burst-size>5</dscp-remark-burst-size>
           <dscp-remark-rate>12</dscp-remark-rate>
          <prec_level>1</prec_level>
          <meter-band-types>
                <flags>ofpmbt-dscp-remark</flags>
           </meter-band-types>
         </meter-band-header>
    </meter-band-headers>
    <meter-id>1</meter-id>
<meter-name>Foo</meter-name>
</meter>
```

4.5.2.3. QoS Support in Virtual Tenant Network (VTN)

To perform QoS in VTN (Virtual Tenant Network) which works on top of OpenDaylight, flow filters must be used. The flow-filter function discards, permits, or redirects packets of the traffic within a VTN, according to specified flow conditions. Priority and dscp remarking can be performed in VTN.

The following two figures are examples of different QoS policies used in a VTN. sets the dscp to 55 in order to differentiate the selected packets.

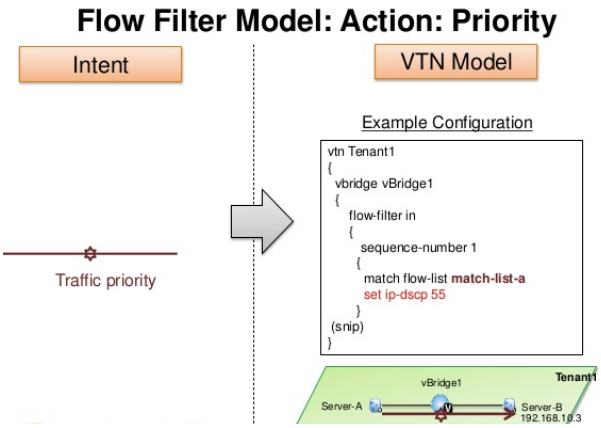


Figure 4-32 Prioritising Traffic in VTN

Figure 4-33 provides a more complicated example which uses three color marking in order to apply specific thresholds and limits to the appropriates packets.

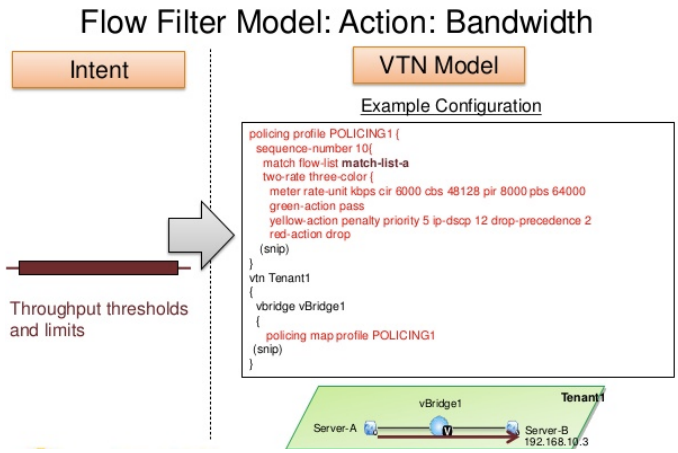


Figure 4-33 Three color marker in VTN

VTN Supported filter actions for the provision of QoS are illustrated in the following Table (Table 4-8)

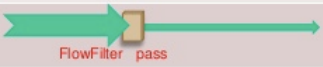
Intent	Description	Behavior
Pass	Pass packets	
Drop	Drop packets	
Redirect	Redirect packets to a specified point	
Priority	Set a priority of packets	
Bandwidth	Set policing	
Statistics	Collect statistics information	

Table 4-8 VTN Filter Actions

4.5.2.4. LibVirt QoS Support

QoS related information is signalled to libvirt configuration via the <*bandwidth*> element [LIBVIRT]. The element allows setting quality of service for a particular network under the constrain that the network type is <*forward*> i.e route, nat or no mode at all. This element is defined as a subelement of a domain's <*interface*>, a subelement of a <*network*>, or a subelement of a <*portgroup*> in a <*network*>. Reciting from the libvirt relevant info page [LIBVIRT]:"As a subelement of a domain's <*interface*>, the bandwidth only applies to that one interface of the domain. As a subelement of a <*network*>, the bandwidth is a total aggregate bandwidth to/from all guest interfaces attached to that network, not to each guest interface individually. If a domain's <*interface*> has <*bandwidth*> element values higher than the aggregate for the entire network, then the aggregate bandwidth for the <*network*> takes precedence. This is because the two choke points are independent of each other where the domain's <*interface*> bandwidth control is applied on the interface's tap device, while the <*network*> bandwidth control is applied on the interface part of the bridge device created for that network."
Inbound and Outbound traffic shaping is supported independently. The supported configuration options allowed in the <*bandwidth*> element are:

- *average* i.e. average bitrate for the shaped interface;
- *peak* i.e. the maximum rate at which bridge can send data;
- *burst* i.e amount of kbytes for a single burst;
- *floor* i.e. applicable only for inbound traffic, guarantees minimal throughput for shaped interfaces.

4.5.2.5. Openstack QoS Support

The QoS support in Openstack is a product of the current implemented interfaces with the available mechanisms of the components and drivers being utilised, namely libvirt, neutron and supported ml2 plugins, OpenDaylight and VTN, and finally OVS.

OpenStack Liberty release introduces built-in Quality of Service support, a feature that allows OpenStack tenant administrators to offer different service levels based on application needs and available bandwidth. Within the current Openstack roadmap, the focues is to provide an extensible API and reference implementation that enables bandwidth limiting through egress bandwidth limit rules (i.e. rules that apply on the traffic exiting the VMs). QoS support in Liberty is perceived as an advanced service plug-in that is decoupled from the rest of the Neutron code on multiple levels and it is available through the ml2 extension driver[OSTACK-QOS][NEUT-QOS].

There are two different ways to apply QoS policies:

- Per network: All the ports plugged on the network where the QoS policy is applied get the policy applied to them.
- Per port: The specific port gets the policy applied, when the port had any network policy that one is overridden.

To create a QoS policy, a policy name must be chosen and then used in the following command:

```
root@liberty-aion:~# neutron qos-policy-create bw-500
Created a new policy:
+-------------+--------------------------------------+
| Field       | Value                                |
+-------------+--------------------------------------+
| description |                                      |
| id          | 0c2723b1-65da-4ce3-a81a-62e1379872f9 |
| name        | bw-500                               |
| rules       |                                      |
| shared      | False                                |
| tenant_id   | 563e4e68140d45b6945f39fec0140afb     |
+-------------+--------------------------------------+
```

After issuing the previous command, we need to configure the bandwidth limit rules in the policy, by specifying the maximum kbps and maximum burst (expressed in kbytes)[1] that are allowed:

[1] NOTE: The burst is actually calculated over excess bytes allowed, the printed command has a typo.

```
root@liberty-aion:~# neutron qos-bandwidth-limit-rule-create \
> 0c2723b1-65da-4ce3-a81a-62e1379872f9 \
> --max-kbps 500 --max-burst-kbps 50
Created a new bandwidth_limit_rule:
+---------------+--------------------------------------+
| Field         | Value                                |
+---------------+--------------------------------------+
| id            | a806ba29-521c-46cd-869c-fd274f9d7db4 |
| max_burst_kbps | 50                                   |
| max_kbps      | 500                                  |
+---------------+--------------------------------------+
```

Finally, to associate the created policy with an existing neutron port, use the following command with the relevant IDs:

```
root@liberty-aion:~# neutron port-update a3fde36b-b7d9-4b6a-8cf2-9ad34f4ffda7 \
> --qos-policy 0c2723b1-65da-4ce3-a81a-62e1379872f9
Updated port: a3fde36b-b7d9-4b6a-8cf2-9ad34f4ffda7
```

Also rules can be modified in runtime. Rule modifications will be propagated to any attached port.

```
root@liberty-aion:~# neutron qos-bandwidth-limit-rule-update \
> a806ba29-521c-46cd-869c-fd274f9d7db4 0c2723b1-65da-4ce3-a81a-62e1379872f9 \
> --max-kbps 2000 --max-burst-kbps 200
Updated bandwidth_limit_rule: a806ba29-521c-46cd-869c-fd274f9d7db4
```

To review the QoS policies, use the following command:

```
root@liberty-aion:~# neutron qos-policy-list
+--------------------------------------+------------+
| id                                   | name       |
+--------------------------------------+------------+
| 0c2723b1-65da-4ce3-a81a-62e1379872f9 | bw-500     |
| 2c64bc0a-fe02-4f6c-ae5d-8ffdcce9922f | bw-500     |
| 913356db-6f40-418a-8264-5c98b9f04728 | bw-limiter |
| db403273-d086-43da-bf43-bca10eb746eb | bw-limiter |
+--------------------------------------+------------+
```

To view the details of a specific QoS policy, use the following command:

```
root@liberty-aion:~# neutron qos-policy-show 0c2723b1-65da-4ce3-a81a-62e1379872f9
+-------------+-------------------------------------------------------------+
| Field       | Value                                                       |
+-------------+-------------------------------------------------------------+
| description |                                                             |
| id          | 0c2723b1-65da-4ce3-a81a-62e1379872f9                        |
| name        | bw-500                                                      |
| rules       | a806ba29-521c-46cd-869c-fd274f9d7db4 (type: bandwidth_limit) |
| shared      | False                                                       |
| tenant_id   | 563e4e68140d45b6945f39fec0140afb                            |
+-------------+-------------------------------------------------------------+
```

To view a rule of a specific QoS policy, use the following command:

```
root@liberty-aion:~# neutron qos-bandwidth-limit-rule-show \
> a806ba29-521c-46cd-869c-fd274f9d7db4 0c2723b1-65da-4ce3-a81a-62e1379872f9
+----------------+--------------------------------------+
| Field          | Value                                |
+----------------+--------------------------------------+
| id             | a806ba29-521c-46cd-869c-fd274f9d7db4 |
| max_burst_kbps | 50                                   |
| max_kbps       | 500                                  |
+----------------+--------------------------------------+
```

Some validation tests were conducted with iperf to verify the application of QoS policies. The VM in the following example had an ingress limit of 2 Mbps and the traffic generated was in the area of 1 to 5 with 0.5 mbps increment. The following figures show that the traffic reaching the server is less or equal than 2 Mbit, thus complying with the applied QoS policy. The duration of each iperf measurement was 10 sec.

```
local 192.168.66.3 port 37326 connected with 10.143.0.213 port 50
Interval       Transfer     Bandwidth
 0.0- 5.0 sec  2.98 MBytes  5.00 Mbits/sec
Sent 2127 datagrams
Server Report:
 0.0- 5.3 sec  1.25 MBytes  1.99 Mbits/sec  15.675 ms 1237/ 2126
```

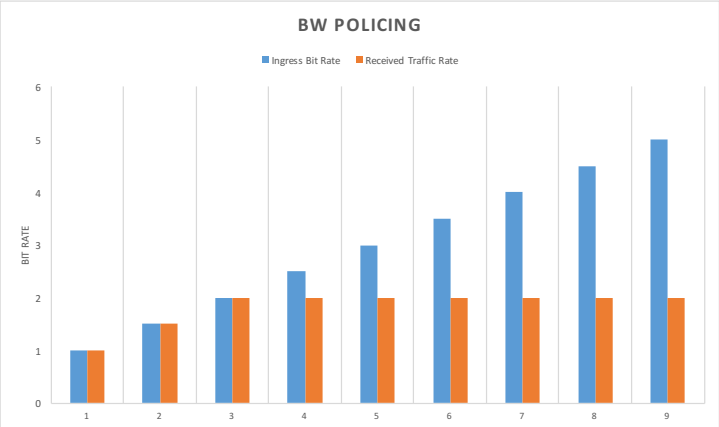


Figure 4-34 BW Policing example

The QoS study in the frame of the project is considered as research topic. Extensions of the above measurements are considered to be delivered within WP7 frame and contributed to Deliverable D7.2.

4.5.3. QoS Considerations

From the analysis above, it appears that full QoS support in the Openstack environment is only achievable via additional interfacing with the actual components that participate in the Openstack environment. The use of SDN in conjunction with Openstack allows bigger freedom in the QoS models that maybe supported and dynamic and flexible control on the granularity of the solutions. However, Openstack environment withholds no knowledge of configurations happening without the use of its own APIs. Therefore any QoS scheme to be attempted should be also retained and managed by a resource management component operating at the Orchestrator level.

In T-NOVA the QoS specific requirement are declared via the descriptors as adapted by ETSI i.e. Network Service Descriptor (NSD), Virtual Network Function Descriptor (VNFD) and Virtual Link Descriptor (VLD). The aforementioned descriptor will be decomposed by the Orchestrator and HEAT files will be generated and signalled to the Virtual Infrastructure Manager (VIM) that are responsible for the resource allocation with the designated PoPs (it might be the case where more than one VIM will be used considering a multi-PoP environment). The file snippet below reveals the VNFD, and VLD parts where the QoS attributes are declared.

VLD	VNFD

…	…
vitual_links:	vl_id: "mngt"
- vld_id: "vld0"	connectivity_type: "E-Line"
root_requirements: "10Mbps"	connection_points_reference: ["vdu0:mngt0", "vnf0:mngt0"]
leaf_requirements: "10Mbps"	root_requirement: "10Mbps"
- qos:	leaf_requirement: "10Mbps"
average: "5Mbps"	- qos:
peak: "6Mbps"	average: "5Mbps"
burst: "50"	peak: "6Mbps"
…	burst: "50"
	net_segment: "192.168.1.0/24"
	dhcp: "TRUE"

Table 4-9 VNFD/VLD examples for QoS configuration.

Depending on the scenario to be implemented the qos field in the above files needs to be expanded in order to support e.g. DiffServ or statistical QoS within the NFVI-PoP or across the PoP over the WAN. The above scenarios is anticipated to be tackled at the final version this deliverable (D4.22).

Comment [LZ1]: To be provided by NCSRD

4.6. Persistency of Network Configuration

As stated in previous sections, a distributed control plane is a must in T-NOVA framework. Since the T-NOVA control plane is based on OpenDaylight, the several SDN controllers are in Clustering Mode.

Among the advantages of OpenDaylight clustering mode (e.g. High Availability of control plane), persistency of network configuration is provided. Data persistence means whenever a controller instance is manually restarted or even crashes, any data gathered by the controller is not lost, but network configuration is persisted to a database. This database can be in the local drive of the server on which the controller instance is deployed on, or even in a remote database, in order to also introduce data redundancy.

Thanks to this mechanism, when the functionality of a controller instance is correctly restored after a crash, all the data stored on the disk will allow the controller to reconstitute the previous network configuration. With network configuration is intended the inventory of network equipment, the interconnections, and for each switch the list of installed flows, meters and statistics.

After an initial synchronisation phase with the other cluster members, the restored controller instance is fully operational and ready to take in charge the management of the switches.

In order to validate this stated behaviour, a batch of tests was run using three instances of OpenDaylight Lithium (each one running in a separate VM) configured as a cluster.

The orchestration of the aforementioned VMs was performed by OpenStack Liberty (community standard flavour). OpenStack Liberty was run in a 4-node configuration (Figure 4-35):

- Controller Node and Network Node were virtual machines on an external vSphere farm;
- Compute Nodes used were HPE Proliant BL465 Gen8.

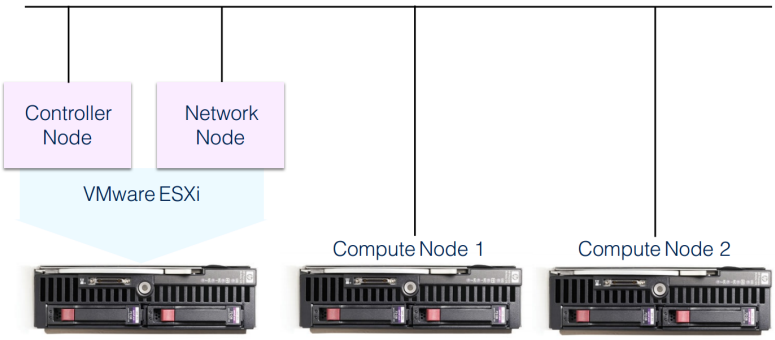


Figure 4-35 Testbed configuration

To run our tests, an SDN application built over OpenDaylight was run, called Persistence Test Application. Thanks to this tool, it is pretty straightforward to test the controller behaviour in case of failure verifying the network configuration persistence. In particular, it allows users to generate data to be stored, persisted and eventually retrieved.

Our tests were performed with the following configurations:

- A single instance of OpenDaylight Lithium;
- A cluster of three instances of OpenDaylight Lithium.

Clustering mode in OpenDaylight can be also run with a single instance of the controller. With this setup, network configuration is still persisted, but now the control plane is centralised, with the consequence of becoming a single point of failure. Obviously, with a single controller, High Availability is not provided anymore. This case is unfeasible with T-NOVA architecture, therefore it was not considered as a test scenario.

The workflow for each test can be divided in the following steps:

- With the controller correctly working, Persistence Test Application was started;
- Through this application were created:
  - an additional user, with a name and a set of permissions;
  - a network device, setting for it a name and an IP address.
- The controller was stopped rebooting the VM hosting it, in order to simulate a controller failure;
- After the controller is restarted, the Performance Test Application was run again, checking if the data previously inserted were correctly persisted by the controller.

As we expected, when OpenDaylight was run in single instance mode, the controller reboot caused the loss of the manually inserted network configuration.
On the contrary, when OpenDaylight was configured to run in clustered mode, data manually inserted were correctly persisted and available to be retrieved, even after the controller simulated failure.

These tests highlighted the importance of running the T-NOVA control plane in a distributed environment, in order to have (in addition to High Availability) the persistence of network configuration, available to be recovered upon a failure.

4.7. Inter-DC/WAN integration - WAN Infrastructure Connection Manager (WICM)

4.7.1. Overview

In T-NOVA, a service is composed of two basic components:

- VNF as-a-Service (VNFaaS), a set of associated VNFs hosted in one or multiple NFVI-PoP.
- A connectivity service, in most cases including one or multiple WAN domains.

The end-to-end provisioning of the T-NOVA service to a customer in a real scenario relies on the proper integration of these two components, as illustrated in Figure 4-36 for a typical enterprise scenario:

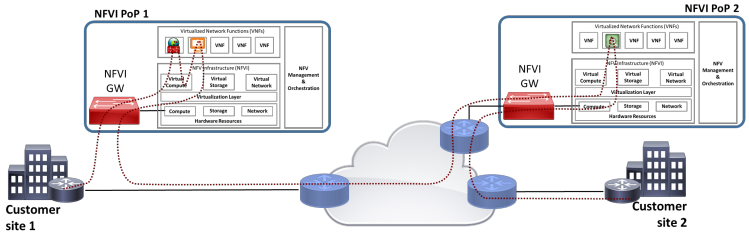


Figure 4-36 T-NOVA service: end-to-end view

The integration of these two components in T-NOVA is carried out by the SDN-based WAN Infrastructure Connection Manager (WICM). The functionality provided by the WICM has been initially described in section 3.8 of T-NOVA Deliverable D2.32. In this section, WICM design and implementation is reported.

The WICM is the architectural component that handles the integration of WAN connectivity services with the NFVI-PoPs that host the VNFs. To understand the role of the WICM, an important concept to be taken into account is the VNF location. As described in D2.32, two types of VNF location should be distinguished:

- Logical VNF location, which corresponds to the point in the customer network where the service is installed (typically, a customer attachment point to the service provider network);

- Physical VNF location, which identifies the actual placement of the VMs that support the VNF (NFVI-PoP ID and hosts).

The logical location is supposed to be decided by the customer and specified at the time of service subscription, whereas the physical location is decided by the service provider following the service mapping algorithm.

As noted in D2.32, the role played by the WICM greatly depends on the proximity between logical and physical locations, more specifically whether or not they are separated by a WAN segment (or, perhaps more precisely, whether or not an existing WAN connectivity service is affected by the insertion of the VNF in the data path).

For the sake of simplicity, the following sections (until 4.7.5) are based on two assumptions:

- VNF logical and physical locations are close to each other, i.e., there is no WAN segment in between (case A, following D2.32 section 3.8);
- The VNFs that compose a network service are located in a single NFVI-PoP.

Section 4.7.6 will discuss the extensions to the basic model to cover the general case where these restrictions do not apply.

The basic scenario considered in the next few sections is illustrated in Figure 4-37.

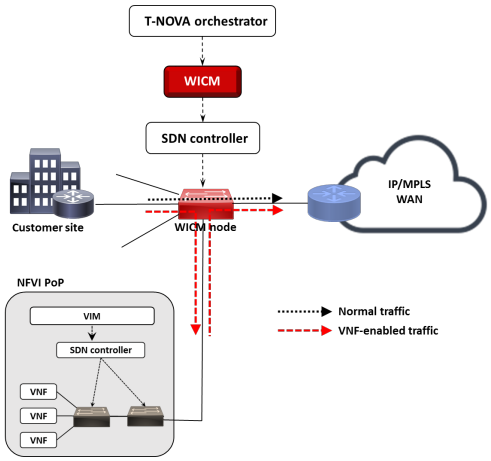


Figure 4-37 NFVI-PoP and WAN integration with WICM

A pivotal role in this scenario is played by a WAN switching node, controlled by the WICM (WICM node in the figure above), which is in charge of forwarding the traffic received from the customer directly to the service provider edge node (if no VNF is included in the data path), or to divert it to a nearby NFVI-PoP (if one or more VNFs are included in the data path). The same applies to the traffic coming from the WAN towards the customer.

It is up to the WICM to decide how and when to enforce the rerouting of the customer traffic. Typically, the actions performed by the WICM are triggered by the subscription of a new VNF service by the customer.

The sequence of events is as follows:

1. Through the marketplace dashboard, the customer selects the VNF service and the respective location (i.e. the logical location, following the definition above).
2. The marketplace sends the following info to the NFVO: [Customer ID, NAP ID, Service ID]
3. The NFVO decides about service mapping and provides the following information to the WICM:
    - Customer ID (same as received from Marketplace)
    - NAP ID (same as received from Marketplace)
    - Service Descriptor
    - NFVI-PoP (if service mapping is executed)
4. Through consultation of customers' database, the WICM maps [Customer ID, NAP ID] into the concrete network location. This follows the same convention used for PoPs, followed by [switch ID, physical port ID, VLAN tag], e.g. GR-ATH-0012-345-678.
5. Based on the network location obtained in the previous step, the WICM identifies the switch and customer VLAN[2] and allocates a VLAN to the network segment between the NFVI-PoP GW and the WAN switching node. If the customer VLAN is already being used in that segment, the WICM needs to allocate a new VLAN and inform the NFVO.
6. The WICM acknowledges the allocation process success or failure, if successful returns the external customer VLAN ID to the NFVO.
7. The NFVO interacts with VIM to create the NS instance using the VLAN ID provided earlier by the WICM.
8. The VIM acknowledges the NS deployment success or failure to the NFVO. Moreover, this step successful conclusion implies that the VNFs are in place and ready to start processing the customer's traffic.
9. Finally, the NFVO is now able to notify the WICM to start enforcing the customer's traffic redirection and avoid any service loss.

Figure 4-38 illustrates steps 3 to 9 from the previous sequence of events.

[2] This is the VLAN that carries the customer traffic between the CE and the PE network elements.

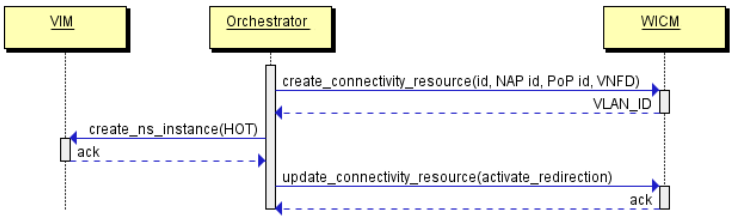


Figure 4-38 WICM procedure

4.7.2. WICM Architecture

The WICM architecture, shown in Figure 4-39, is composed of four functional components:

- WICM API - this component provides an interface through which the NFVO can make traffic redirection requests
- WICM DB – the database enables the persistent storage of network services, customers logical location, NFVI-PoPs location and other necessary data elements.
- OpenDaylight Controller – this component enables the centralised management and control of the network infrastructure.
- WICM Traffic Redirection Services – this component is responsible for the realisation of traffic redirection services received by the API. To enforce the services it uses the DB and the OpenDaylight controller.
- WAN-SW – these represent the network elements that are capable of redirecting the customer's network traffic to a nearby NFVI-PoP.

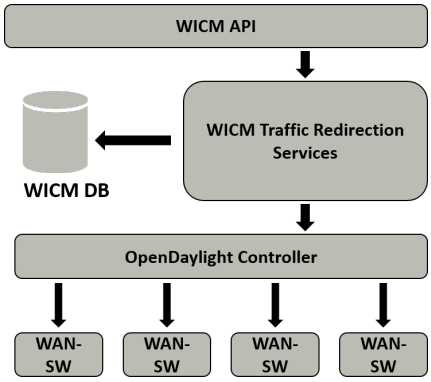


Figure 4-39 WICM Architecture

Each WICM connectivity resource has one the following states:

State	Description
ALLOCATED	The switch logic port is allocated to the resource (VLAN ID), but traffic is not yet being redirected.
ACTIVE	Traffic redirection is active.
ERROR	There was a problem in the redirection configuration.
NOT AVAILABLE	It is not possible to allocate the connectivity resource.
TERMINATING	The connectivity resource is being destroyed.
TERMINATED	The connectivity resource is destroyed, traffic redirection is disable.

Table 4-10 WICM Resource states

When a connectivity creation request is received, the WICM queries the database retrieving all used VLANs (either ALLOCATED or ACTIVE states) for the same NFVI-PoP. If VLANs are available, the resource is registered as ALLOCATED and the VLANs returned. Otherwise the resource is registered as NOT AVAILABLE.

After the successful resource creation, the next step is to activate the traffic redirection. When such request is received, the WICM queries the database to retrieve the resource information (NAP, NFVI-PoP, VLANs). With this information, the WICM requests the SDN controller to put the Openflow rules in place for traffic redirection. Once the redirection is in place, the WICM removes the pre-VNFaaS Openflow rules, using the SDN controller, to save space in the switch. Finally, the resource is set as active. If anything fails, the resource state is set to ERROR and Openflow rules for pre-VNFaaS mode are put in place.

A customer may also wish to unsubscribe VNF services hosted on the NFVI-PoP, which implies returning to pre-VNFaaS mode, thus disabling traffic redirection. The WICM handles this request by querying the database to retrieve the resource information and sets its state as TERMINATING. Then the WICM requests the SDN controller to instantiate the pre-VNFaaS forwarding rules and once this rules are in place the old rules are deleted. To finish this operation the state of the resource is set to TERMINATED.

4.7.3. WICM API

The description of the REST interface used between the NFVO and the WICM is available in the table below:

URI	Method	Parameter	Description
/vnf-connectivity	POST	ns_instance_id; NAP ID; Service Descriptor; NFVI-PoP ID	Create a connectivity resource reserving two VLAN IDs in WICM. Said IDs are returned to the NFVO in the response body. (Steps 3-4)
/vnf-connectivity/:ns_instance_id	GET	N.A.	Query the status of **ns_instance_id** resource.
/vnf-connectivity/:ns_instance_id	PUT	N.A	Update **ns_instance_id** resource, enabling traffic redirection. (Steps 5-6)
/vnf-connectivity/:ns_instance_id	DELETE	N.A.	Delete **ns_instance_id** resource, disabling traffic redirection.

Table 4-11 WICM APIs – Single PoP scenario

4.7.4. WICM Dependencies

WICM is fully implemented using python 2.7, the database storing all connectivity resources information is implemented using MySQL version 5.5 and uses the Lithium (3.0.3) version of OpenDaylight requiring the following features: *odl-openflowplugin-all* and *odl-restconf-all*.

The following python libraries are needed to run the WICM:

Software library	Description
Flask – 0.10.1	Used to provide the rest API for the orchestrator.
MySQL-python – 1.2.5	MySQL plugin required for database access.
SQLAlchemy – 2.0	Object relational mapper.
requests – 2.7.0	Used to control the SDN.

Table 4-12 WICM Software dependencies

4.7.5. WICM Demonstration

In a pre-VNFaaS mode of operation, traffic flows directly from the customer edge to the respective provider edge and vice-versa. The introduction of the VNFs raises the need to redirect traffic to network functions located in datacenters while still maintaining transparency from the customer point of view.

Figure 4-40 stands as a demonstration scenario for the WICM.

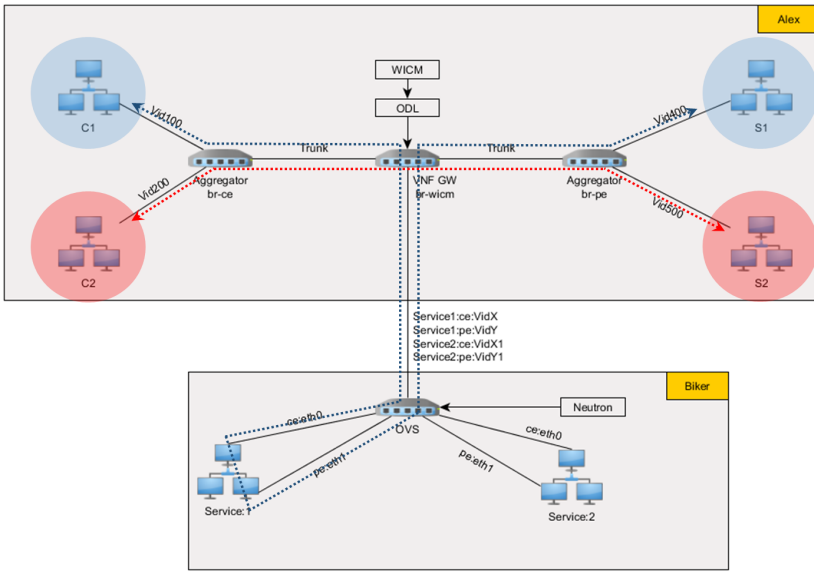


Figure 4-40 WICM demonstration scenario

The upper block Alex (named after the machine where it is instantiated) contains all that is needed for the pre-VNFaaS mode of operation, in this case provider and customer edges are in different VLANs as shown in the image. C1 and C2 are supposed to be located in the customer domain (br-ce being the equivalent of the customer edge node), while S1 and S2 are servers located beyond the provider edge, represented by br-pe. S1 communicates with C1, while S2 communicates with C2.

Alex also contains three interconnected switches: two aggregator switches placed one at each edge and one in the middle introduced to act as the VNF gateway. The edge switches are responsible for handling VLAN tagging, providing intra-edge connectivity and forwarding to the respective complementary edge, through the VNF gateway. The responsibilities of VNF GW switch, named br-wicm, encompass redirecting traffic to the correct datacenter and then forward the traffic back from the datacenter to the correct edge (further explanation will follow on how this is realised). When a given edge pair (client/provider) has no active VNFs, the VNF GW defaults to VLAN ID switching and forwarding, ensuring that there is no loss of connectivity. This last switch is operated by OpenDaylight (ODL), which is controlled by WICM. WICM accepts requests to either enable or disable redirection to VNFs from the orchestrator.

The other presented block, named Biker, represents the datacenter where VNFs are placed using OpenStack. In this example, for the sake of brevity, only one VNF is used instead of a full service chain as it is all that is necessary to verify the correctness of the operation performed by the WICM. VNFs in Biker are connected to an OVS switch managed by OpenStack's Neutron.

Alex and Biker are connected through a physical wire, where the redirected traffic flows. VLANs are used to distinguish which VNF is to be used and also to distinguish the direction of the flows. These VLANs are assigned by WICM following a /vnf-connectivity POST, which returns two IDs for a given redirection request – one for client-provider and another for provider-client directions. Considering these IDs, the orchestrator instantiates the VNF with two "Provider Network" connections, one for each VLAN assigned by WICM. In this example, consider that redirection is enabled only the pair C1-S1, with assigned VLAN IDs 1 and 2, as illustrated in Figure 4-41.

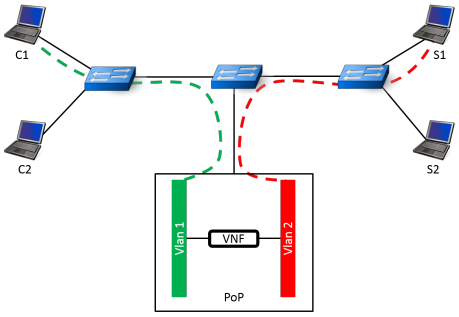


Figure 4-41 NFVI-PoP VLANs and connectivity with external network

Figure 4-42 shows the Openflow flows implemented in the br-wicm, a stylied output of the command "*ovs-ofctl –O openflow13 dump-flows br-wicm*". Cookie is used to represent the type of flow:

- NFVI (Network Virtual Function Instance): redirection flow
- NAP (Network Access Point): normal operation
- DEFAULT: indicating what to if all matches fail.

Priority is the Openflow priority and match/actions are the implemented Openflow rules.

```
The br-wicm openflow state:
Bridge: br-wicm
+---------+----------+--------------------------+-------------------------------------+
|  Cookie | Priority | Match                    | Actions                             |
+---------+----------+--------------------------+-------------------------------------+
|   NFVI  |     1000 | INPUT=openstack,VLAN_ID=2 | SET_FIELD:400->VLAN_VID,OUTPUT:br-pe   |
|   NFVI  |     1000 | INPUT=openstack,VLAN_ID=1 | SET_FIELD:100->VLAN_VID,OUTPUT:br-ce   |
|   NFVI  |     1000 | INPUT=br-pe,VLAN_ID=400   | SET_FIELD:2->VLAN_VID,OUTPUT:openstack |
|   NFVI  |     1000 | INPUT=br-ce,VLAN_ID=100   | SET_FIELD:1->VLAN_VID,OUTPUT:openstack |
|   NAP   |      100 | INPUT=br-pe,VLAN_ID=500   | SET_FIELD:200->VLAN_VID,OUTPUT:br-ce   |
|   NAP   |      100 | INPUT=br-ce,VLAN_ID=200   | SET_FIELD:500->VLAN_VID,OUTPUT:br-pe   |
| DEFAULT |        1 | *                         | DROP                                |
+---------+----------+--------------------------+-------------------------------------+
```

Figure 4-42 Traffic flows in br-wicm

Analysing each line of the table the packets travel in the following manner:

1. All incoming traffic from OpenStack (Biker) tagged with VLAN ID 2 is changed to VLAN ID 400 and forwarded to the provider aggregator switch.
2. All incoming traffic from OpenStack (Biker) tagged with VLAN ID 1 is changed to VLAN ID 400 and forwarded to the client aggregator switch.

3. All incoming traffic from the provider aggregator switch tagged with VLAN ID 400 is changed to VLAN ID 2 and forwarded to the OpenStack (Biker) for VNF processing.
4. All incoming traffic from the client aggregator switch tagged with VLAN ID 100 is changed to VLAN ID 1 and forwarded to the OpenStack (Biker) for VNF processing.
5. All incoming traffic from the provider aggregator switch tagged with VLAN ID 500 is changed to VLAN ID 200 and forwarded to the client aggregator switch.
6. All incoming traffic from the client aggregator switch tagged with VLAN ID 200 is changed to VLAN ID 500 and forwarded to the provider aggregator switch.
7. If no rules match, drop the packet.

The first four lines implement the redirection, cookie NFVI, and the next two lines provide simple connectivity for the pair C2-S2.

4.7.6. Extensions to the basic WICM scenario

In the previous sections a description of the basic functionality of the WICM has been provided following the implementation of the initial prototype. As noted in section 4.7.1, two assumptions were made to facilitate an early implementation of the WICM module:

- VNF logical and physical locations are close to each other, i.e., there is no WAN segment in between;
- All VNFs that compose a network service are located in a single NFVI-PoP.

The following sub-sections analyse the requirements to generalize the above scenario, where the above restrictions do not apply. For the sake of simplicity, the examples analyse the unidirectional flow of traffic, extending to the bidirectional case should as simple as of applying the same approach in both directions.

4.7.6.1. Remote logical VNF location

In the case where physical and logical locations of a VNF are different, the connection between the WICM switching node and the NFVI-PoP has to be extended across one or more WAN domains. Two possible approaches are illustrated in Figure 4-43 and Figure 4-44 below, where the traffic flowing from left to right is represented by a red dotted line.

The first case corresponds to the direct replacement of the connection between the WICM node and the local NFVI-PoP (represented before in Figure Figure 4-37) by a tunnel that may cross one or several WAN domains and is terminated at another WICM node located at, or next to, the remote NFVI-PoP. Traffic leaving the initial WICM node is sent to the NFVI-PoP through a tunnel and then sent back to the same WICM node. From that point, the normal path to the destination is resumed. This is a simple approach, fully transparent from the point of view of the existing WAN service, as only the tunnel endpoints are aware of modifications in the path between source and destination. The price to pay is routing inefficiency and increased latency

motivated by the so-called "tromboning" effect , which is minimized by the alternative approach described below.

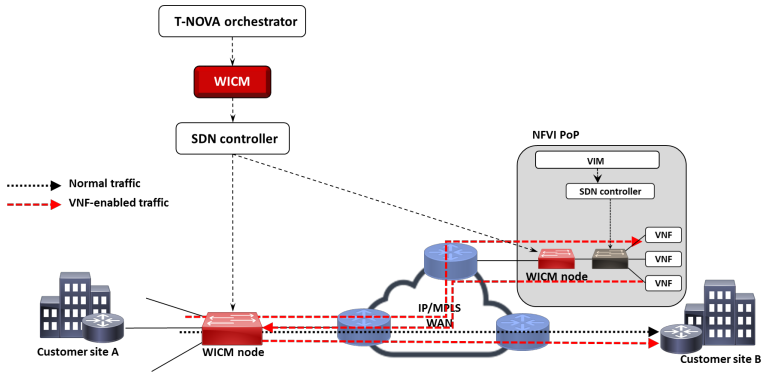


Figure 4-43 NFVI PoP in remote location – first approach

In the second approach, represented in Figure 4-44, the packets leaving the NFVI-PoP are not tunnelled back to the initial WICM node, but rather forwarded directly to the final destination. This is a more efficient approach compared to the one described above but introduces added complexity, because it is not transparent from the point of view of the existing WAN connectivity service

and would require an overarching orchestration entity to perform integrated control of VNF lifecycle and the WAN connectivity service. A detailed discussion of such scenario is beyond the scope of this document.

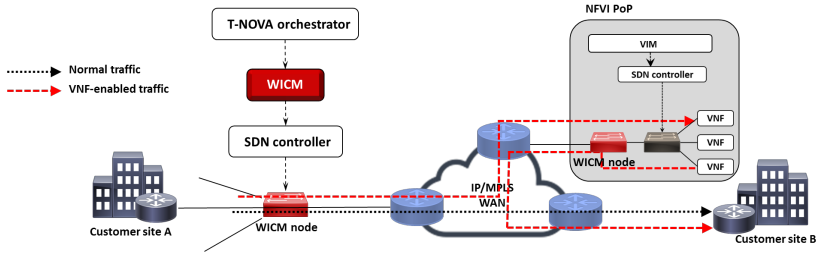


Figure 4-44 NFVI PoP in remote location – second approach

4.7.6.2. Multiple NFVI-PoPs

Up to now, it has been assumed that all VNFs that compose a network service are instantiated in a single NFVI-PoP. However, for several reasons, this may not always be the case – it makes sense that different VNFs may be located at different NFVI-

PoPs, running different VIM instances (This corresponds to the scenario depicted in Figure 4-36, in section 4.7.1).

This scenario can be seen as a special case of service function chaining in which the several VNF components are spread across multiple NFVI-PoPs.

To support this scenario, the interface between the orchestrator and the WICM must be extended in such a way that an ordered list of NFVI-PoP IDs, not a single NFVI-PoP ID, is received by the WICM.

In a multi-NFVI-PoP environment, it makes sense that the functionality of WICM node, performing the steering of the traffic across the multiple NFVI-PoP, is instantiated at the service provider network edge, as well as at every NFVI-PoP as shown in Figure 4-45. It should be noted that the figure only represents the case of unidirectional flow from customer site A to B – for the reverse direction, a WICM node would also be necessary at the network edge close to customer site B, if traffic redirection was required at that point.

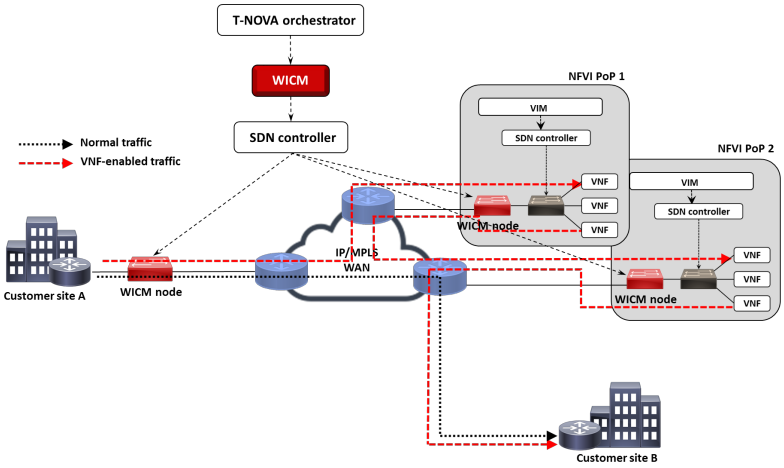


Figure 4-45 Multiple NFVI-PoP scenario

In terms of WICM API, in order to accommodate the extensions described above, the only change required to the basic version described in Table 4-11, is that instead of a single NFVI-PoP, an ordered list of NFVI-PoPs may be needed, as shown in Table 4-13 .

URI	Method	Parameter	Description
/vnf-connectivity	POST	ns_instance_id; NAP ID; Service Descriptor; [NFVI-PoP ID1, …, NFVI-PoP IDn]	Create a connectivity resource reserving two VLAN IDs in WICM. Said IDs are returned to the NFVO in the response body. (Steps 3-4)
/vnf-connectivity/:ns_instance_id	GET	N.A.	Query the status of **ns_instance_id** resource.
/vnf-connectivity/:ns_instance_id	PUT	N.A	Update **ns_instance_id** resource, enabling traffic redirection. (Steps 5-6)
/vnf-connectivity/:ns_instance_id	DELETE	N.A.	Delete **ns_instance_id** resource, disabling traffic redirection.

Table 4-13 WICM APIs – MultiPoPs scenario

With regard to the WICM process described before in Figure 4-38, the only relevant difference is that now the orchestrator has to deal with multiple VIM instances (on the left hand site of the Figure 4-38). In relation to the interface between the orchestrator and the WICM, nothing changes apart from the extension of the API, as shown in Table 4-13.

Going one step further in terms of generality, one may assume that NFVI-PoPs hosting VNFs may be located at different administrative domains. Figure 4-46 provides a general overarching scenario, including multiple NFVI-PoPs located at multiple provider domains. Detailed elaboration of this scenario is out of the scope of this document.

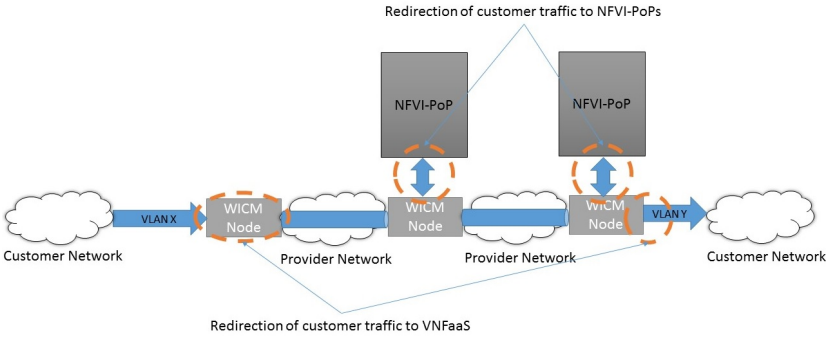


Figure 4-46 Overall multi-domain multi-PoP scenario

5. VALIDATION TESTS

Besides the testbeds described above, focused on the individual validation of each feature developed within Task 4.2, dedicated test environments have been designed for enabling the integration of the Control Plane in the T-NOVA IVM layer.

This section presents the validation tests that have been carried out focusing on the SDN Control Plane and the SFC with WICM components.

5.1. SDN Control Plane

The implemented testbed is depicted in Figure 5-1.

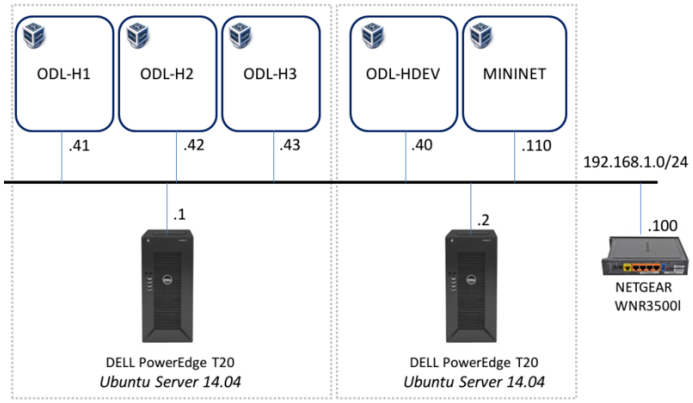


Figure 5-1 SDN Control Plane testbed

As also reported in D4.52, the testbed comprises two physical nodes, specifically Dell PowerEdge T20 equipped with CPU Intel Xeon E3-1225 (Quad core) CPU, 8 GB RAM, 1TB Hard Disk, 1GB Ethernet port. They both are used to host five virtual machines, organized as follows:

- *ODL-H1*, *ODL-H2*, *ODL-H3* host three instances of OpenDaylight controller (Lithium release) forming a cluster of controllers.
- *ODL-HDEV* is used for development and building purposes. It holds the OpenDaylight source code which can be built and deployed on the *ODL-H{DEV,1-3}* machines.
- *MININET* is used to provide a SDN network topology in order to evaluate the control plane operations.

The functional tests were intended to validate the proper functioning of the SDN Control Plane, with particular focus on the clustering service, considering two different scenarios: one working with a single instance of controller and another one working with a cluster of multiple controllers.

For this purpose, the following test cases have been identified and then implemented:

Single Instance of Controller

Aim of the test: This test is responsible for validating the functionality of recovery from persistent data when using the clustering service. The test has been carried out using a simple Openflow network with three nodes. The OpenDaylight controller is connected to the network nodes in order to control them.

Precondition : A single instance of controller (with clustering service activated) is deployed and running (*ODL-H1*)

Description of the test:

1. Start *MININET* and connect the network to the controller
2. Check if all nodes and the tables for those nodes are present in the operational data using the RESTful interface (RESTCONF)
3. Stop the controller
4. Start the controller again
5. Repeat step 2.
6. Exit *MININET*

Result:

When the clustering service is activated on the controller instance, the step 2 and the step 5 output the same data. So, as explained in section 4.6, the clustering service is essential to ensure that the network configurations stored into the persistent local data are recovered and applied again into the network nodes.

Multiple Instances of Controllers

Aim of the test: This test is responsible for validating the functionality of high availability of the control plane after that the one instance (specifically the leader) of the cluster fails.

Precondition: Multiple instances of controllers (with clustering service activated) are deployed and running (*ODL-H1, ODL-H2, ODL-H3)*

Description of the test:

1. Start *MININET* and connect the network to the cluster
2. Stop the controller acting as current leader of the cluster (i.e. O*DL-H1*)
3. Leader moves to one of the remaining controllers (*ODL-H2* becomes the new leader)
4. Use the new leader and continue performing requests (through the RESTful interface) for operations on the network.
5. Exit *MININET*

Result:

The new leader is able to process incoming RESTful requests, making the cluster of controllers failure-proof, since it detects the fault and elects a new leader. Therefore, this test reveals pivotal role of the clustering service in ensuring high availability of the control plane.

5.2. Service Function Chaining

In section 4.3, Service Function Chaining was described as one of the target use cases inside the T-NOVA domain. This capability was a reference application in the development of the SDK4SDN, called Netfloc (elaborated in details in Deliverable [D4.31]). The aim was to demonstrate a first end-to-end scenario that was enabled by directly employing its dedicated traffic steering library. Moreover the product of the SFC implementation has delivered a successful integration of several T- NOVA components into a holistic chaining solution, demonstrated at the Year2 T- NOVA review in Brussels.

5.2.1. SFC in the scope of T-NOVA and demonstration details

Figure 5-2 depicts the overall scenario. On the left side is the NFVI-PoP, i.e. the cloud infrastructure consisting of three OpenStack Nodes, an SDK4SDN (Netfloc) node and a physical switch, whereas on the right, the WICM resides with its own SDN controller instance. The T-Nova Orchestrator manages the WAN and the NFVI-POP and it is in charge of instantiating the VNFs and starting the network services onto the underlying infrastructure. On a higher level, once a new client registers in the NFVI Marketplace to request for a specific network service, the orchestrator triggers the WICM for a new isolated Client ID space. The WICM then creates a client's instance ID and reserves VLAN IDs for the clients' (User1 and User2 in the scenario) network service management.

As detailed in Section 4.7, once the orchestrator receives an approval from the WICM, it requests for traffic redirection coming from User1 to User2 to pass via the NFVI PoP. The orchestrator is in charge of managing the network services' lifecycle by instantiating the required VNFs in the OpenStack Cloud infrastructure and mapping the Neutron port IDs of the VNF VMs to the respective service descriptors. The VNFs such as the vTC and vTU, are in charge of functions like packet classification, enhanced deep packet inspection, video analysis etc., depending on the particular service specification.

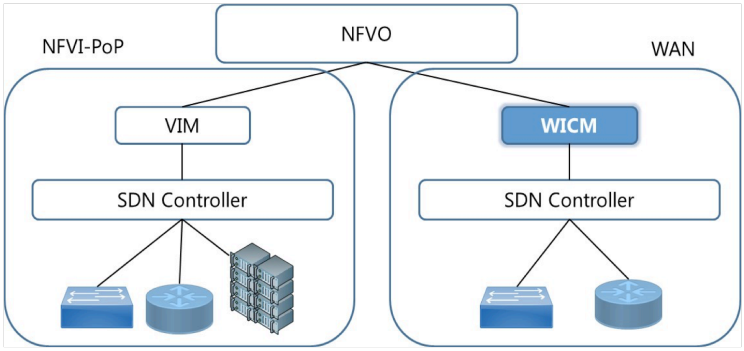


Figure 5-2 SFC flows installed on OpenStack Control Node

Once the VNFs have been instantiated and their services successfully started, the orchestrator calls the Netfloc Restful API for chain creation in order to request for traffic steering along a designated service path. Following are the CREATE and DELETE Chain API call examples, according to the SDK4SDN specification:

CREATE CHAIN API:

```
curl -H 'Content-Type: application/json' -X POST -d '{"input": {neutron-ports :  "06ba1679-25df-448e-8755-07bc4a356c85,d481a967-44b6-4e64-9be3-cf49b80702d3,bac1fa34-b515-47e0-9698-9e70d549330b,d09befd1-385c-4fb5-86bc-debfe6fa31dc"}}' --verbose -u admin:admin http://127.0.0.1:8181/restconf/operations/netfloc:create-service-chain
```

The Create Chain JSON data consists of a comma separated array of the Neutron port IDs that are associated to the ingress/egress pair interfaces residing along the specific network path.

DELETE CHAIN API:

```
curl -H 'Content-Type: application/json' -X POST -d '{"input": {service-chain-id : "1"}}' --verbose -u admin:admin http://127.0.0.1:8181/restconf/operations/netfloc:delete-service-chain
```

The Delete Chain JSON data contains the Chain ID, returned by Netfloc on service chain creation.

5.2.2. SDK for SDN (Netfloc) directed chain graph

In depth representation of the prototype scenario is shown in Figure 5-3. User1 and User2 are the endpoints in the chain connected to the Pica8 switch that, upon being instructed by the WICM, sends to the NVFI PoP bidirectional VLAN tagged traffic (ex. ID 400 and 401 for sending and receiving). The OpenStack VMs are hosting the VNFs involved in the scenario. The Service Forwarder VM (SF) is the main entry and exit VM for the outside traffic into the cloud.

Figure 5-3 points out a simplified bridge and interface setup in comparison to the baseline OpenStack model [OSN].

This effect is a direct product of using Netfloc as a fully SDN based counterpart of the OpenStack ML2 networking service, in order to achieve simplification of the tunnelling protocols to grant a tenant isolation. Instead, using fully SDN-enabled L2 approach, Netfloc ensures tenant isolation implemented in one of its reference SDK libraries.

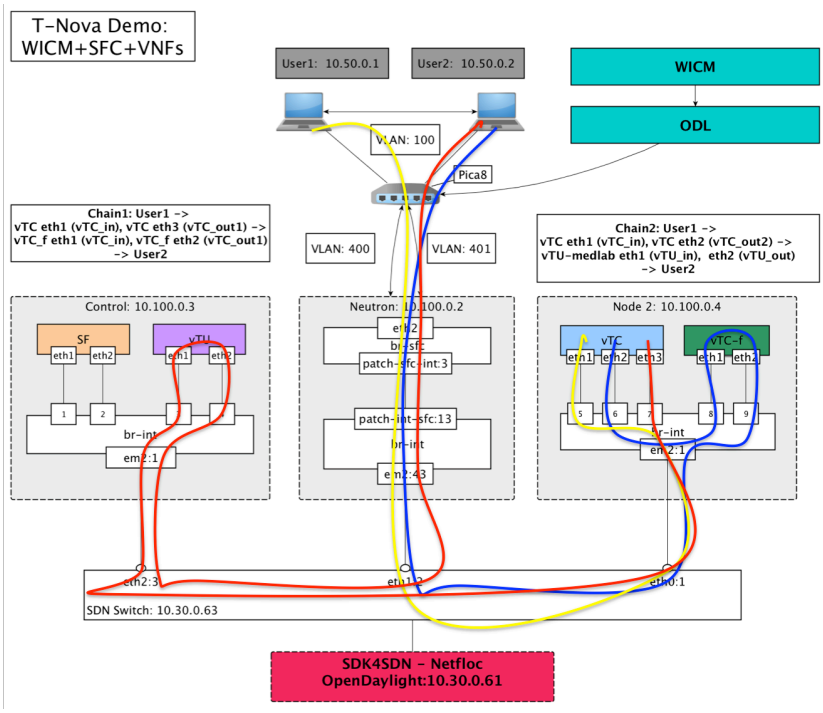


Figure 5-3 T-NOVA SFC pilot setup

The lines on Figure 5-3 depict the chain paths (two in the case of the demonstration). The yellow line is common for both chains and shows the traffic coming onto the vTC ingress port, as a first interface in the chain. From there the vTC classifies the traffic and sends it out on two different egress interfaces, *eth2* – the blue line and *eth3* – the red line.

The blue line follows the ICMP data traffic that is further steered into the vTC-f (forwarding VNF) and out of the PoP, whereas the red line shows the UDP video traffic that is further steered into the vTU VNF in order to get transcoded. The vTU then puts a watermark, before sending the video traffic out. The modified video is shown in User2, while the data traffic is captured by using *tcpdump* command in order to verify the distinction between the two network chains, Figure 5-4.

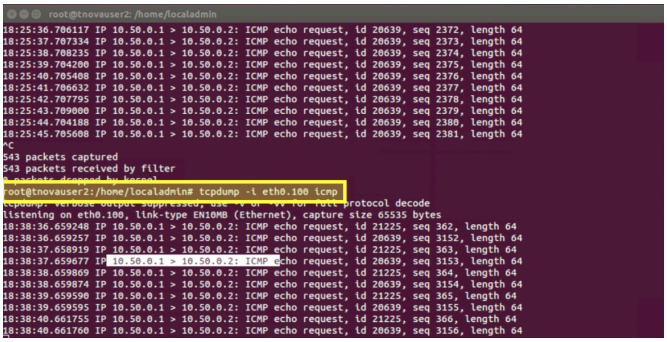


Figure 5-4 Chain1: ICMP traffic output from User1 captured in User2

Figure 5-5 shows a video capture as perceived by User2 in two cases: disabled traffic redirection by the WICM (left), and enabled traffic redirection with SFC enforced by the SDK4SDN-Netfloc (right). The video used for the test is an excerpt from the animated movie "Big Buck Bunny" [BBB].

Figure 5-5 Chain2: Video output in User2 without redirection (left) and with redirection, i.e. SFC steered traffic (right)

Furthermore we point out some of the flow actions installed on the SDN switch's *br-int* interface and the Neutron Node's *br-sfc* interface in order to follow up the traffic steering inside the physical hosts of the SFC-PoP.

Figure 5-6 shows the flows installed on the Neutron network Node. Since the Neutron nodes is the first entry point in the SFC-PoP, the packets coming from outside (from User1) are tagged with VLAN numbers assigned by the WICM in the Pica8 SDN controller. The packets tagged with VLAN number 400 in this case are the packets from User1, whereas the VLAN number 401 packets are coming from User2 towards User1 for bidirectional communication.

Figure 5-6 SFC flows installed on OpenStack Network Node

Finally, Figure 5-7 outlines the SDN switch flows installed automatically by Netfloc on service chain creation. They have the highest priority (20) in the instructions set and provide MAC rewriting based traffic steering as elaborated in details in Deliverable D4.31 [D4.31].

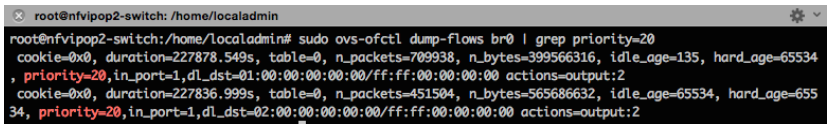


Figure 5-7 SFC flows installed on Switch

6. CONCLUSIONS

In this deliverable we reported the results of the activities of research, design and implementation done in Task 4.2 "SDN Control Plane" of T-NOVA project.

The functional architecture of the SDN control plane based on the requirements outlined in previous T-NOVA deliverables is described. Analysis of a variety of candidate technologies has been carried out in order to identify a suitable solution for the SDN controller implementation based on a balanced view of the available and missing features. To this end, OpenDaylight has been selected as the Network Controller with Virtual Tenant Network (VTN) as the multi-tenant network virtualisation framework and Clustering Service for the controller deployment in large-scale environments as well as the persistency of the network configuration. Several approaches have been investigated to provide traffic steering functionalities supporting service function chaining for NFV deployments. In addition, different solutions for load balancing in multi-controllers scenarios have been developed.

Moreover, Task 4.2 have developed experimental plans to evaluate the performance of the selected technologies under a number of different scenarios. Test plan have been implemented to collect quantitative data in order to evaluate the Controller architecture options i.e. single instance vs. clustered, in terms of high availability and resiliency of the network controller. Last but not least, WAN Infrastructure Connection Manager have been developed in order to provide integration of WAN connectivity services.

7. List of Acronyms

Acronym	Description
API	Application Programming Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
BUM	Broadcast, Unknown unicast and Multicast
CP	Control Plane
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
DC	Data Centre
DCN	Distributed Cloud Networking
DOVE	Distributed Overlay Virtual Ethernet
DPI	Deep Packet Inspection
FTP	File Transfer Protocol
FW	Firewall
GPE	Generic Protocol Extension
GRE	Generic Routing Encapsulation
GW	Gateway
HA	High Availability
HPE	Hewlett Packard Enterprise
HTTP	HyperText Transport Protocol
IP	Internet Protocol
ISP	Internet Service Provider
IVM	Infrastructure Virtualisation Management
JVM	Java Virtual Machine
L2	Layer 2
L3	Layer 3
L4	Layer 4
LB	Load Balancer
MAC	Medium Access Control

MD-SAL	Model-Driven Service Abstraction Layer
ML2	Modular Layer 2
MPLS	Multi-Protocol Label Switching
NAP	Network Access Point
NFV	Network Function Virtualisation
NFVO	NFV Orchestrator
NIC	Network Interface Card
NSH	Network Service Header
NVGRE	Network Virtualisation using Generic Routing Encapsulation
ODL	OpenDaylight
OF	Openflow
OSGi	Open Services Gateway initiative
OVS	Open vSwitch
OVSDB	Open vSwitch Database
POP	Point Of Presence
QOS	Quality of Service
REST	Representational State Transfer
SDK	Software Development Kit
SDN	Software Defined Networking
SF	Service Function
SFC	Service Function Chaining
SFF	Service Function Forwarder
SFP	Service Function Path
TTL	Time-To-Live
UDP	User Datagram Protocol
UUID	Universal Unique Identifier
VIM	Virtual Infrastructure Manager
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNFAAS	VNF As A Service
VNFFG	Virtual Network Function Forwarding Graph
VPN	Virtual Private Network

VRS	Virtualised Routing & Switching
VSC	Virtualised Services Controller
VSD	Virtualised Services Directory
VSP	Virtualised Services Platform
VTEP	VxLAN Tunnel Endpoint
VTN	Virtual Tenant Network
vTC	Virtual Traffic Classifier
VxLAN	Virtual Extensible Local Area Network
WAN	Wide Area Network
WICM	WAN Infrastructure Connection Manager

8. REFERENCES

[AKKA] "Akka". [Online]. Available: http://akka.io/ , accessed 20-Feb-2016.

[BBB] "Big Buck Bunny Reference Video". Available: https://peach.blender.org/ , accessed: 19-Apr-2016

[BEACON] "Beacon". [Online]. Available: https://openflow.stanford.edu/display/Beacon, accessed 21-Nov-2015.

[BGH+14] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, and W. Snow, "ONOS: towards an open, distributed SDN OS" in Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014, pp. 1–6.

[D2.32] T-NOVA Consortium, "Deliverable D2.32 Specification of the Infrastructure Virtualisation, Management and Orchestration - Final" . 2015.

[D4.31] T-NOVA Consortium, "Deliverable D4.31 SDK for SDN - Interim" . 2015.

[D4.32] T-NOVA Consortium, "Deliverable D4.32 SDK for SDN - Final" . 2016.

[DHM+13] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," presented at the Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, Hong Kong, China, 2013.

[FAT TREE] Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. ACM SIGCOMM Computer Communication Review, pp. 63-74, 38(4),(2008)

[FLOODLIGHT] "FloodLight". [Online]. Available: http://Floodlight.openflowhub.org/, accessed 21-Nov-2015.

[HPECTX] "HPE ContextNet White Paper". [Online]. Available: http://www8.hp.com/h20195/v2/GetPDF.aspx/c04725726.pdf, accessed 21-Nov-15

[JUNICON] "Contrail Architecture". [Online]. Available: http://www.juniper.net/us/en/local/pdf/whitepapers/2000535-en.pdf, accessed 21-Nov-2015

[JUNIOPENC] "OpenContrail". [Online]. Available: http://www.opencontrail.org , accessed 21-Nov-2015

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," ACM Trans. Comput. Syst., vol. 18, pp. 263-297, 2000.

[KZM+14] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based comparison and selection of Software Defined Networking (SDN) controllers," in Computer Applications and Information Systems (WCCAIS), 2014 World Congress on IEEE, 2014.

[LBGIT] Github repository of the SDN Control Plane Load Balancer [Online]. Available https://github.com/CRAT-EU/T-NOVA, accessed 19-Feb-2016

[LIBVIRT] Libvirt Documentation. [Online]. Available https://libvirt.org/formatnetwork.html, accessed 21-Nov-2015

[LIBVIRT] LibVirt QoS Element, online: http://libvirt.org/formatnetwork.html#elementQoS

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, 2008.

[MUL] "MUL SDN Controller". [Online]. Available: http://sourceforge.net/p/mul/wiki/Home/, accessed 21-Nov-2015.

[NEUT-QOS] Neutron QoS API extension, on-line: http://specs.openstack.org/openstack/neutron-specs/specs/liberty/qos-api-extension.html

[NG13] T. Nadeau and K. Gray, "SDN: Software Defined Networks". Sebastopol, CA, USA: O'Reilly Media, 2013.

[NODEFLOW] "NodeFlow". [Online]. Available: https://github.com/dreamerslab/node.flow, accessed 21-Nov-2015

[NOX] "About NOX". [Online]. Available: http://www.noxrepo.org/nox/about-nox/, accessed 21-Nov-2015.

[NSH] "Network Service Header (NSH)". [Online]. Available: https://tools.ietf.org/html/draft-quinn-sfc-nsh-03 , accessed 21-Nov-2015

[NUAGE] "Nuage: Virtualized Services Platform". [Online]. Available: http://www.nuagenetworks.net/products/virtualized-services-platform/, accessed 21-Nov-15

[ODL] OpenDaylight: A Linux Foundation Collaborative Project. [Online]. Available: http://www.opendaylight.org, accessed 21-Nov-2015.

[ODL-NSH] "Service Function Chaining in OpenDaylight using NSH protocol". [Online]. Available: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main , accessed 21-Nov-2015

[ODL-RESERV] ODL Lithium: Reservation Project, on-line: https://wiki.opendaylight.org/view/Reservation:Main

[ODL-SFC1] "Service Function Chaining: Helium, Lithium an beyond". Available: http://events.linuxfoundation.org/sites/events/files/slides/SFC-Helium-Lithium-and-beyond.pdf , accessed 21-Nov-2015

[ODL-SFC2] "OpenDaylight service function chaining use-cases". [Online]. Available: https://wiki.opendaylight.org/images/8/89/Ericsson-

Kumbhare_Joshi-OpenDaylight_Service_Function_Chaining.pdf , accessed 21-Nov-2015

[ODL-SFC3] "OpenDaylight Service Function Chaining". [Online]. Available: http://opentechindia.org/wp-content/uploads/2012/07/OpenSDNIndia2015-Vinayak.pdf, accessed 21-Nov-2015

[OF13] "Openflow Switch Specification v1.3". [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, accessed 21-Nov-2015

[ONOS] ON.LAB, "ONOS: Open Network Operating System." [Online]. Available: http://onosproject.org/, accessed 21-Nov-2015.

[OOSC] "OpenStack Orchestrated Service Chaining". [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/ODL%20Summit%202015%20-%20OpenStack%20Orchestrated%20Network%20Service%20Chaining_0.pdf, accessed 21-Nov-2015.

[OPENIRIS] "OpenIRIS: The Recursive SDN Openflow Controller by ETRI". [Online]. Available: http://openiris.etri.re.kr , accessed 21-Nov-2015

[OPNFV-FG] "OpenStack Based VNF Forwarding Graph". [Online]. Available: https://wiki.opnfv.org/requirements_projects/openstack_based_vnf_forwarding_graph , accessed 21-Nov-2015.

[OPNFV-SFC] "OPNFV Project: Service Function Chaining". [Online]. Available: https://wiki.opnfv.org/service_function_chaining , accessed 21-Nov-2015.

[OSTACK-QOS] http://docs.openstack.org/networking-guide/adv_config_qos.html

[OSN] OpenStack networking details. Available: https://www.rdoproject.org/networking/networking-in-too-much-detail, Accessed: 19.4.2016.

[OVS] "Open vSwitch". [Online]. Available: http://openvswitch.org, accessed 21-Nov-2015.

[POX] "A Python-based Openflow Controller". [Online]. Available: http://www.noxrepo.org/pox/about-pox/, accessed 21-Nov-2015.

[PPK+09] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," in Eighth ACM Workshop on Hot Topics in Networks (HOTNETS VIII), New York, USA, 2009.

[RAFT] "RAFT consensus". [Online]. Available: https://raftconsensus.github.io, accessed 20-Feb-2016.

[RYU] "Ryu". [Online]. Available: http://osrg.github.com/ryu/, accessed 21-Nov-2015.

[SFC00] "Service Function Chaining (SFC) Control Plane Components & Requirements". [Online]. Available: https://tools.ietf.org/html/draft-ietf-sfc-control-plane-00 , accessed 21-Nov-2015

[SFC03] "Service Function Chaining (SFC) Use Cases In Data Centers". [Online]. Available: https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-03 , accessed 21-Nov-2015

[SFC04] "Service Function Chaining (SFC) Use Cases in Mobile Networks". [Online]. Available: https://tools.ietf.org/html/draft-ietf-sfc-use-case-mobility-04 , accessed 21-Nov-2015

[SFC11] "Service Function Chaining (SFC) Architecture". [Online]. Available: https://tools.ietf.org/html/draft-ietf-sfc-architecture-11 , accessed 21-Nov-2015

[TREMA] "Trema". [Online]. Available: http://trema.github.com/trema/, accessed 21-Nov-2015.

[ZC11] Zheng Cai, "Maestro: Achieving Scalability and Coordination in Centralized Network Control Plane", Ph.D. Thesis, Rice University, 2011

[ZCD+15] L. Zuccaro, F. Cimorelli, F. Delli Priscoli, C. Gori Giorgi, S. Monaco, V. Suraci, "Distributed Control in Virtualized Networks". 10th International Conference on Future Networks and Communications (FNC 2015)