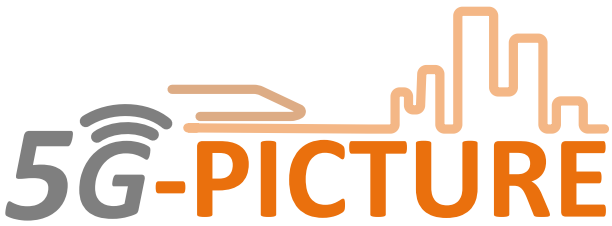


5G Programmable Infrastructure Converging disaggregated network and compUte REsources

D5.2 Auto-adaptive hierarchies

This project has received funding from the European Union's Framework Programme Horizon 2020 for research, technological development and demonstration

5G PPP Research and Validation of critical technologies and systems

Project Start Date: June 1st, 2017 | **Duration**: 33 months

Call: H2020-ICT-2016-2 | **Date of delivery**: 31st May 2019

Topic: ICT-07-2017 | **Version** 1.0

Project co-funded by the European Commission

Under the H2020 programme

Dissemination Level: Public

Grant Agreement Number:	762057
Project Name:	5G Programmable Infrastructure Converging dis-aggregated network and compUte REsources
Project Acronym:	5G-PICTURE
Document Number:	**D5.2**
Document Title:	Auto-adaptive hierarchies
Version:	1.0
Delivery Date:	31st May 2019
Responsible:	University of Thessaly (**UTH**)
Editor(s):	Kostas Choumas (**UTH**), Daniel Camps-Mur (**I2CAT**/**UPC**)
Authors:	Kostas Choumas (**UTH**), Sevil Dräxler (**UPB**), Hadi Razzaghi (**UPB**), Azahar Machwe (**ZEETTA**), Osama Arouk (**EUR**).
Keywords:	Network Function Virtualization, Software-Defined Networking, Heterogeneous Infrastructure.
Status:	Final
Dissemination Level	Public
Project URL:	http://www.5g-picture-project.eu/

Revision History

Rev. N	Description	Author	Date
0.1	Table of contents and document structure	UTH	02.05.2019
0.2	First contributions from UTH, EUR and ZEETTA	EUR, ZEETTA	06.05.2019
0.3	First contribution from UPB to Section 2 and related work. Second contribution from UTH to Section 3.	UPB, UTH	09.05.2019
0.4	Section 3.2 is added by UPB. Corrections by EUR.	UPB, EUR	16.05.2019
0.5	Executive summary, introduction and conclusion	UTH	17.05.2019
0.6	Addressing 1st internal review comments from I2CAT	UTH, ZEETTA	22.05.2019
0.7	Addressing 1st internal review comments from I2CAT	UTH	23.05.2019
0.8	Addressing 1st internal review comments from I2CAT	ZEETTA, UPB, EUR	28.05.2019
1.0	Final review and submission to the EC	IHP	31.05.2019

Table of Contents

List of Figures

List of Tables

Executive Summary

This document is deliverable, D5.2: *"Auto-adaptive hierarchies"*, which belongs to Task 5.2: *"An auto-adaptive, cross-concept orchestrator and controller hierarchy"* of the 5G-PICTURE project. This deliverable provides the 5G-PICTURE solutions on the challenges for scalable management, orchestration and control. More specifically, it describes in detail the special features of the 5G Operating System (5G OS), as it is defined by 5G-PICTURE in D5.1: "*Relationships between Orchestrators, Controllers, slicing systems*", that enable the support for auto-adaptive, cross-concept and hierarchical management, orchestration and Software Defined Networking (SDN) control. A 5G-PICTURE network is too large for a single (even proactive) controller/orchestrator. This suggests subdividing the domain of responsibility of a controller/orchestrator, creating a hierarchy. Our approach is implemented using state-of-the-art software tools and evaluated through testbed experimentation and simulations, showing the scalability of the adaptable, hierarchical design of 5G OS. More details on our implemented prototypes will be provided in the next deliverable, D5.4: *"Integrated prototype (across tasks and work packages)"* (due Nov. 2019). Furthermore, mathematical model is also used for extracting heuristic algorithms for quick response on real-time problems.

1 Introduction

The 5G Operating System (5G OS) is designed in 5G-PICTURE to face a set of major challenges related to the management, orchestration and control of resources, such as the virtualisation support and the creation of a layer of abstraction.

As every operating system, 5G OS is also responsible for:

1. providing control over heterogeneous resources,
2. seamlessly recovering from resource failures, and
3. being efficient even if the pool of resources is extended.

The existence of multiple instances for the Orchestrator, Controller and Network Function Virtualization (NFV) Management and Orchestration (MANO) components in Figure 1, depicts the necessity for addressing efficiently the three aforementioned points. At first, the support of multiple domains as group of resources, which may be providing the same functionality using different technologies, enables the cross-concept handling of heterogeneous resources. The abstract layer built on top of these domains facilitates the unified management, orchestration and control of the underlying resources. The cooperation of multiple domains is supported by the coexistence of multiple interconnected components (Controller, NFV MANO), one for each domain. Their strong interoperability and efficient utilisation are challenges that we address in 5G-PICTURE, aiming at a failure resilient and scalable 5G OS.

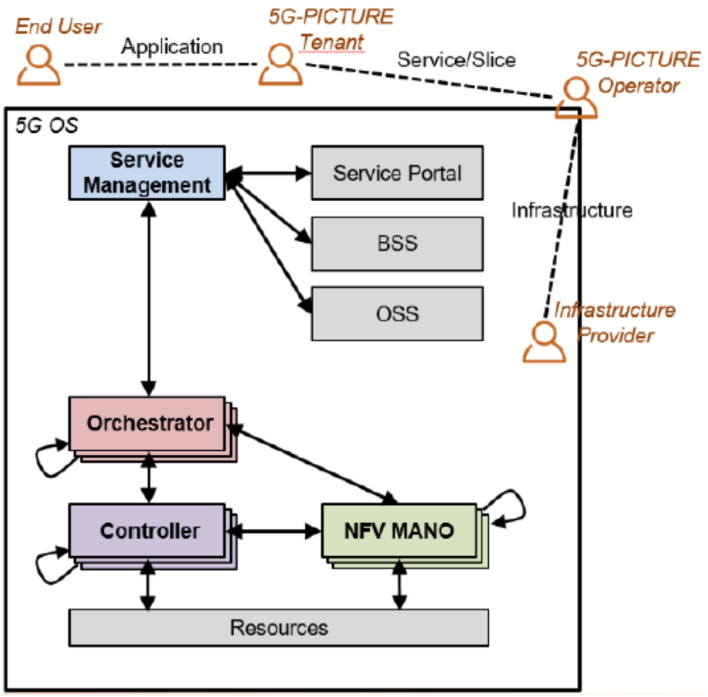


Figure 1: 5G OS high-level architecture.

In Section 2, we present the approach followed in 5G OS for building the links between the Controller and NFV MANO components, extending the functionality already supported by the state-of-the-art tools for implementing these components, such as OpenDayLight [20], OpenStack [18], Kubernetes [12], JOX [8] and Open Source MANO (OSM) [16]. In this way, we fulfill the 5G OS requirement for cross-concept (multi-domain) orchestration. For example, NFV MANO may be responsible for the management and orchestration of compute resources relying on multiple datacenters, which are interconnected through network resources controlled by a Software Defined Networking (SDN) controller. In this case, the synergy between these components is required for the efficient end-to-end deployment of a Network Service (NS) over all these resources. We also present how the Virtual Network Functions (VNFs) of this NS can be placed or even moved between multiple NFV MANO instances of different domains, changing their version according to the utilized Virtual Infrastructure Managers (VIM) of the new NFV MANO. More details on the support for multi-version services will be given in Deliverable D5.3 [2]. Finally, the end-to-end VNF placement of a NS over multiple domains requires

the appropriate configuration of all network parts, including the transport network and the Radio Access Network (RAN), which is described how it is done in the last two subsections of this Section.

The scalable support of multiple domains, as well as the ability of the infrastructure to recover seamlessly from resource failures, are keystone features of 5G OS. These features require 5G OS to be organised in a hierarchical fashion, using a structure that is extensible on demand, in cases that there are new resources or some of them are removed. 5G OS organises the NFV MANO and Controller components in a hierarchical architecture, being auto-adaptive to either scheduled or unexpected changes on the availability of the resources. For example, if new network resources are added, then new controllers should be raised on demand and placed appropriately, such as the resources of their responsibility to be close to them. The same holds for the compute resources and the NFV MANO components respectively.

In Section 3, we present the auto-adaptive and hierarchical management, orchestration and control of 5G-OS. At first, we present the hierarchical SDN control of 5G-PICTURE, which is inherited from 5G-XHaul [5] and extended. The focus of 5G-XHaul was only on networking resources, thus we extend the SDN control produced by 5G-XHaul to be complemented with an orchestrator responsible for the computing resources. Then, we show the hierarchy on the NFV MANO components, which is necessary for scalable management and orchestration. At the end, we conclude with the support of the dynamic placement of eNodeBs or SDN controllers, as well as the algorithms for deciding how many new instances are needed, depending on the current scale of the infrastructure.

Finally, we present in Section 4 how multi-tenancy is supported, which is one of the most important aspects of 5G OS. Multi-tenancy can be defined as hosting multiple service providers (or tenants) on fully or partially shared resources where each tenant is provided a sub-set of the available resources (referred to as a network slice) depending on their requirements.

Section 5 presents the related work and Section 6 concludes the document with a summary of the discussions.

2 Cross-concept Management and Orchestration

In this section, we describe how 5G OS addresses the requirement for cross-concept management and orchestration, handling multiple domains and Points of Presence (PoPs) of various technologies and resources. More specifically, Section 2.1 presents the capability of 5G OS to orchestrate one domain consisting of multiple PoPs and the interconnecting networks, using OSM (together with OpenStack) as NFV MANO and OpenDayLight as Controller. In Section 2.2, the multi-domain capability of 5G OS is presented, which is supported by Pishahang. Pishahang is a multi-domain Orchestrator and partially a NFV MANO component, complemented by OpenStack and Kubernetes for the NFV MANO operation. Section 2.3 provides information for the slicing and VNF placement in RAN, supported by JoX, which is the RAN Controller. Finally, Section 2.4 presents the Zeetta Slicing Engine, which is a Domain Orchestrator enabling slicing and VNF placement in the transport network.

2.1 Multi-PoP orchestrator using Open Source MANO (OSM), OpenStack and OpenDayLight

According to the ETSI Standardization group, the NFV Management and Orchestration (NFV-MANO) [14] is a challenging task that requires the synergy of several functional blocks, organized in an architectural framework and collaborating through specified reference points, as it is depicted in Figure 2. Except for the VNF Manager (VNFM), which is in charge of the VNF lifecycle, the other two functional blocks of our interest are the NFV Orchestrator (NFVO) and the Virtualized Infrastructure Manager (VIM). The NFVO has two main functions, the NS Orchestration (NSO) and the Resource Orchestration (RO), which implement the lifecycle management of the NSs and the orchestration of the NFVI resources across multiple VIMs respectively. The VIM is responsible for controlling and managing the NFVI resources within a single PoP, leveraging on hypervisors and SDN controllers for the control of the computation/storage and network resources respectively. There are also specialized VIMs, such as the WAN Infrastructure Manager (WIM) that controls only network resources. We focus on networks and compute PoPs (e.g. data-centers), controlled by WIMs and VIMs respectively, and we investigate the role of SDN in their operation. The interaction and functionality of the SDN controller within the NFV-MANO architecture is not clearly defined, even in the report on the SDN usage in the NFV-MANO [15], since the SDN controller could either be part of the VIM or the NFVI, among other options. We shed some light on the functionalities that could be offered by the SDN controller, either as a part of the VIM or constituting the WIM, especially for enabling the multi-PoP orchestration.

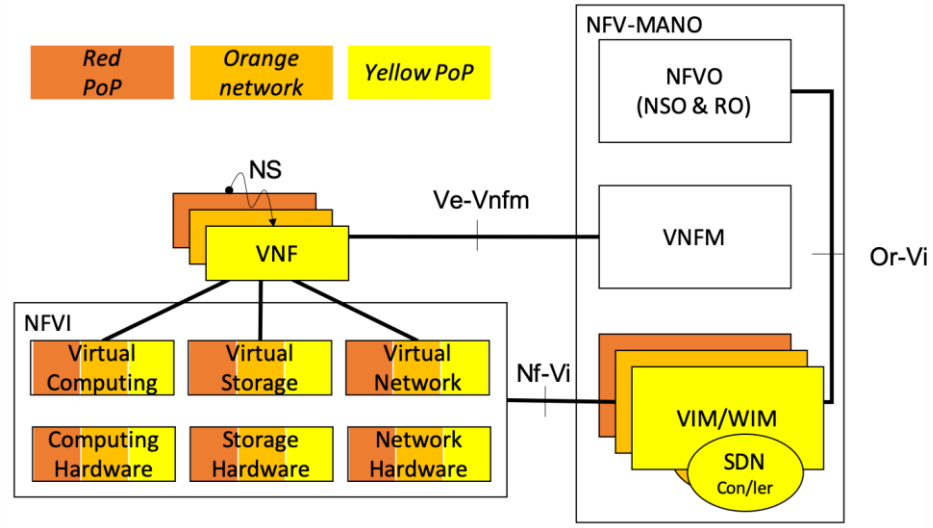


Figure 2: NFV-MANO and SDN controller.

For our deployment we exploit and extend the state-of-the-art open-source software tools, named OSM [16], OpenStack [17], [18] and OpenDayLight [19], [20], which are used for the NFVO/VNFM, the VIM and the SDN control respectively. This triangle (OSM, OpenStack and OpenDayLight) is one of the most widely-used options for implementing the NFV-MANO, and from now on is denoted as Open3. Although Open3 enables a remarkable set of operations, it still does not support multi-PoP NS deployment. We present a solution that enables the VNF deployment of a NS over multiple compute PoPs, as well as their interconnection through a network,

relying on the Open3 tools and our custom-made and open-source software[1]. Our software leverages on the functionalities offered by the OpenDayLight SDN controller. Last but not least, we evaluate our implementation and we showcase the results of our experimentation in the NITOS testbed [36].

We present Open3++ [13], which extends the networking functionality of Open3 by enabling in more scenarios the interconnection of VNFs belonging to the same NS, either if they are collocated inside a single PoP or spread to multiple PoPs. Before proceeding, it will be helpful to clarify the terminology and the classification of the networks followed by OpenStack, since our software extending Open3 to Open3++ is mainly handling these networks with use of OpenDayLight. In a single compute PoP managed by OpenStack, the VNF interconnection is built either through tunnels over the OpenStack management network, which are named tenant or project networks, or using overlay networks on top of the OpenStack guest network, which are called provider networks. Figure 3 depicts these networks. Our focus is on the latter case, since the guest network can be physically connected to other networks and the provider networks may be used for interconnecting VNFs of different compute PoPs. In Open3, the provider networks exist "out there" and OpenStack simply interfaces them. But in Open3++, they are deployed on demand by exploiting the SDN-LAN and SDN-WAN controllers with use of our extensions. The following sections give more details on the SDN-LAN and SDN-WAN controllers, as well as their usage for the creation of flat or vlan provider networks (flat provider networks forward untagged traffic, while vlan provider networks expect for VLAN tagged traffic to forward.) connecting the compute nodes of single or multiple PoPs.

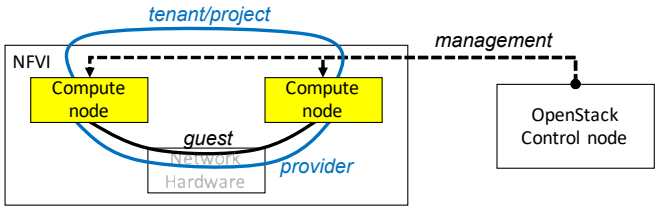


Figure 3: OpenStack networks.

2.1.1 SDN-LAN & SDN-proxy

Let's consider the single-PoP scenario illustrated in Figure 4. In this scenario, all compute nodes belong to the "yellow" PoP, which is managed by a single VIM. In Open3, OpenStack implements the VIM and optionally leverages on the OpenDayLight instance, named SDN-OvS in our examples, for configuring the OvS bridges running in the compute nodes (usually named as br-int). When OpenStack receives from the OSM's RO a request for three VNFs of a new NS, connected through a provider network, it chooses the compute nodes that these VNFs will be deployed according to its scheduling policy. Then, OpenStack deploys the VNFs and informs SDN-OvS to configure the bridges of the chosen compute nodes. The bridge configuration is sufficient for connecting the VNFs located in the same compute node (e.g. VNF 1 and VNF 2), or to export the traffic to the physical NIC, when it is directed from a VNF to another non-collocated one (e.g. from VNF 2 to VNF3). However, the traffic forwarding between the NICs of the compute nodes requires the appropriate configuration of the OpenStack guest network connecting them. This is the task of the extra OpenDayLight instance, which we call SDN-LAN.

[1] http://repo.nitlab.inf.uth.gr/karamiha/odl-project/tree/test
http://repo.nitlab.inf.uth.gr/karamiha/orchestrator

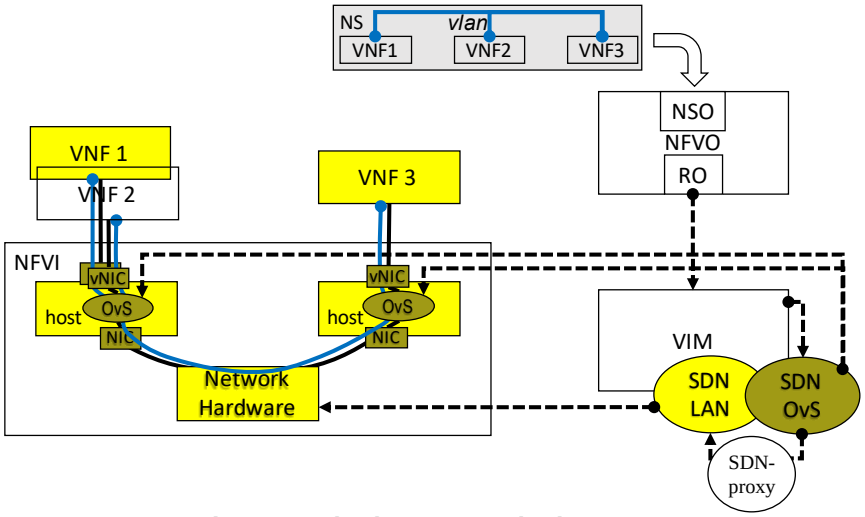


Figure 4: Single-PoP NS deployment.

Open3 does not include any interface to SDN-LAN, assuming that it is standalone and proactively builds and keeps active the provider networks, even in time periods that they are not used. On the other hand, Open3++relies on a software daemon, named SDN-proxy, which repetitively checks SDN-OvS and prompts dynamically SDN-LAN to deploy and keep only the needed provider networks on top of the PoP's underlying network. More specifically, SDN-proxy is a python daemon that uses the REST API of SDN-OvS for retrieving the required provider networks and then talks to the REST API of SDN-LAN for enforcing it to form an isolated overlay network foreach provider network, which functions as an abstract layer-2 switch. This function is completed with the assistance of the Virtual Tenant Network (VTN) Manager plugin, which is used by both OpenDayLight instances implementing SDN-OvS and SDN-LAN. VTN-Manager enables the creation of a virtual bridge (vBridge) for each provider network and VLAN mapping is used for assigning the VLAN traffic of the provider network to the respective vBridge (VLAN 0 corresponds to the untagged flat network).

Figure 5 shows the sequence of REST/HTTP messages exchanged between our daemon and the OpenDaylight instances, SDN-OvS and SDN-LAN. SDN-proxy periodically:

i) gets from SDN-OvS the provider networks (first GET message),
ii) gets from SDN-LAN the existing vBridges (second GET message),
iii) forces SDN-LAN to create a new vBridge for each missing provider network (first POST message),
iv) maps each vBridge to the VLAN of the related provider network (second POST message), and
v) removes the vBridges corresponding to non-existing provider networks (last POST message).

The latter three messages are in loops, since they are repeated for every vBridge that has to be added or removed.

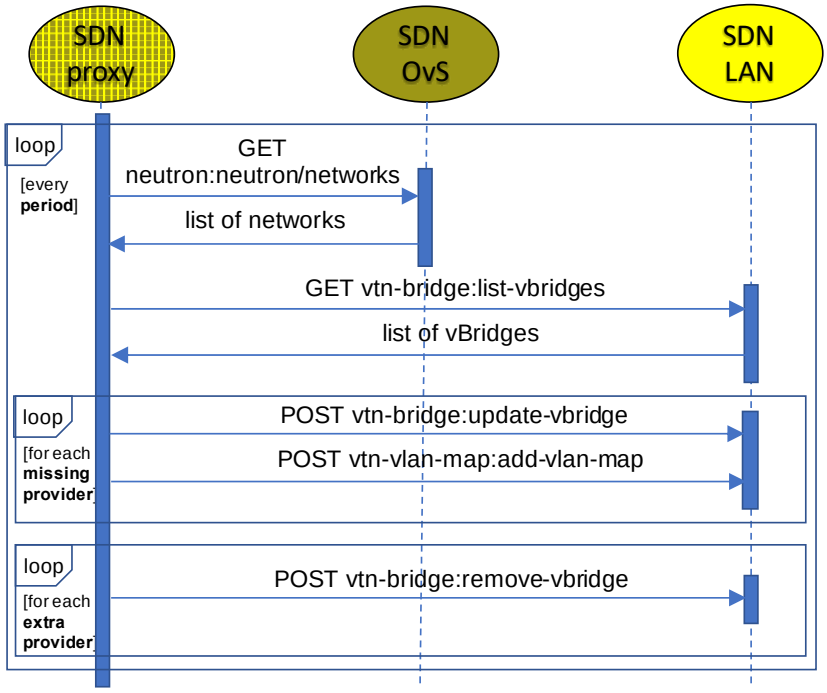


Figure 5: Messages from/to the SDN-proxy.

The disadvantage of this SDN-proxy implementation is that SDN-OvS is periodically checked, thus SDN-LAN gets updated even after a period since SDN-OvS has changed. Another solution is to make OpenStack log these changes (added or removed provider networks) in a file, using the logging mechanism of the OpenStack Neutron component, while SDN-proxy periodically reads this file and tracks the changes. The time period of this process is significantly lower. The two first GET messages of Figure 5 are replaced with the tracking of the log file, while the two inner loops are repeated again, for every added or removed network respectively. On the other hand, the disadvantage of this approach is that SDN-proxy has to be collocated with OpenStack, and the logging mechanism of its Neutron component must be activated.

2.1.2 SDN-WAN & Orchestration-proxy

Let's now focus on the multi-PoP NS deployment, illustrated in Figure 6, where the three VNFs belong again to the same NS, but they are deployed to different PoPs, the "yellow" and the "red" one, which are connected through the "orange" network. In Open3++, we developed a daemon called Orchestration-proxy, which is a proxy between the RO and the VIMs/WIMs. It pretends to be a VIM instance for the RO and a RO for the underlying VIMs/WIMs. In particular, Orchestration-proxy presents to the RO as a single VIM responsible for a single PoP and receives the requests for the NS resources. Then, it "breaks" the set of resources and assigns the subsets to various PoPs, interacting with their VIMs. For these VIMs, it is behaving as the RO, requesting the VNF deployment to their PoPs. Except for this task, the Orchestration-proxy is responsible to interact with the WIM(s) managing the network(s) interconnecting these PoPs, in order to build the VNF connections. Each WIM is actually an SDN controller, called SDN-WAN, responsible for a network used for the interconnection of other PoPs. Open3++ uses OpenDayLight and the VTN-Manager plugin to implement SDN-WAN, which again creates one vBridge for each provider network connecting VNFs, and maps this vBridge to the VLAN traffic of the provider network. The goal of both SDN-LAN and SDN-WAN is to create overlay layer-2 networks, using VLAN for their isolation. The overlays of the same VLAN but of different underlay networks are stitched together. This is a simplified scenario for illustrating the 5G OS capabilities for multi-PoP orchestration and management, however, as we will show in Section 3, the variety of the supported use cases are not limited only to the ones relying on the usage of the same VLAN end-to-end in the whole domain, or the exploitation of only OpenFlow enabled devices.

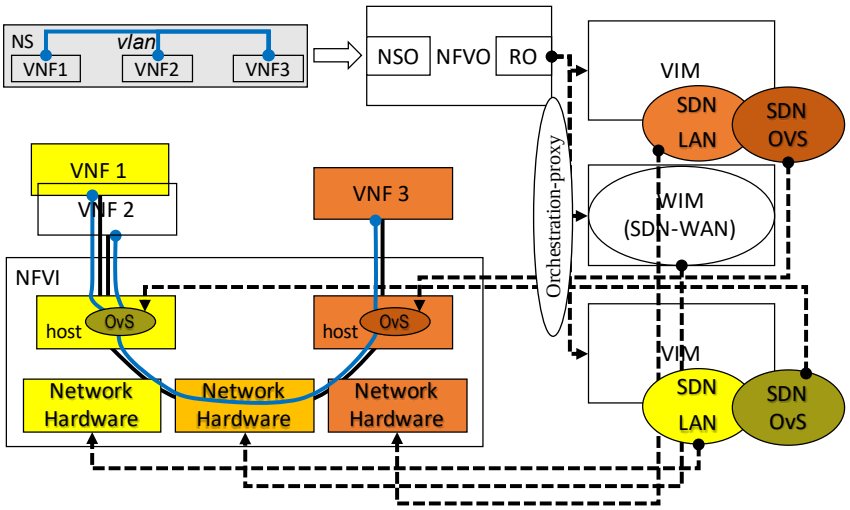


Figure 6: Multi-PoP NS deployment.

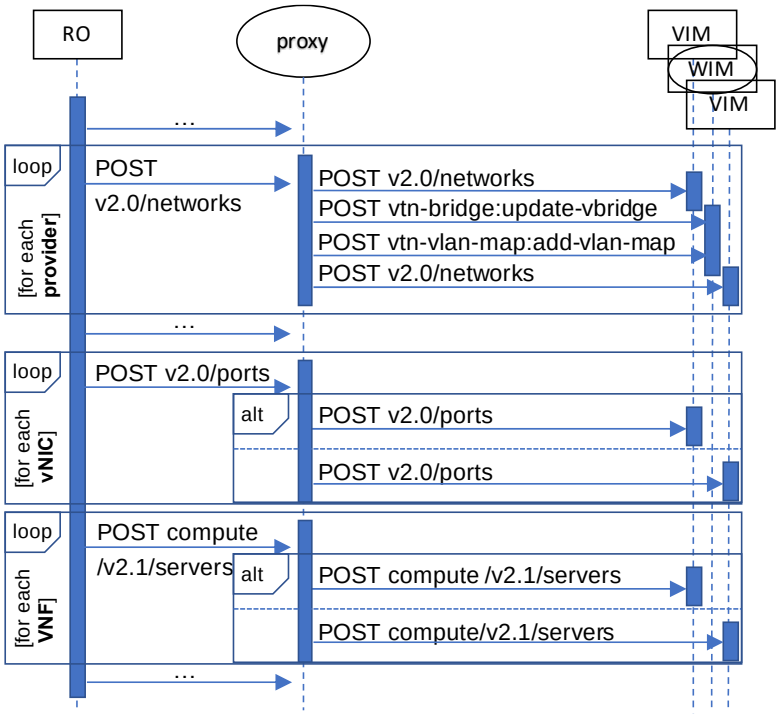


Figure 7: Messages from/to Orchestration-proxy.

Figure 7 depicts the REST/HTTP messages exchanged between Orchestration-proxy and the other functional blocks, when a new NS is deployed. More specifically, Orchestration-proxy is a python daemon that receives messages through its REST API from the RO and sends new messages to the VIMs/WIMs through their REST APIs. Among other messages, we distinguish the following:

i) The RO requests from Orchestration-proxy the utilisation of a provider network (first POST message from the RO), and Orchestration-proxy, in turn, requests from each VIM/WIM the same network (four first POST messages from the Orchestration-proxy). These requests are either OpenStack POST messages to the VIMs of the PoPs (similar to the request from the RO) or OpenDayLight POST messages to the WIM of the network (similar to the two first POST messages in Figure 4(a)).

ii) Then, the RO requests for a new vNIC (or port) for each VNF included in the NS (second POST message from the RO). This request is copied to one of the VIMs (first alt).

iii) Finally, the RO requests the creation of a new VM (or server) for each VNF (third POST message from the RO), which again is copied to a single VIM (second alt).

2.1.3 Evaluation

Both SDN-LAN and SDN-WAN are OpenDayLight instances that utilize the VTN-Manager plugin, in order to build the provider networks. Once again, we mention that this is an example illustrating the multi-PoP support of 5G OS. In this example, the network slicing could be operated by other mechanisms, such as the Zeetta's Slicing Engine, which is presented in Section 2.4 and can also handle non OpenFlow enabled devices. VTN-Manager exploits the Link Layer Discovery Protocol (LLDP) to discover the underlying network and retrieve the shortest path between each pair of switches. Each link is weighted with a cost, which can be modified through the VTN-Manager northbound interface, enabling the flexible redefinition of the shortest paths. VTN-Manager reactively configures flows, which means that the first packet sent from a VNF to another need some extra time to be forwarded, until the controller configures the flows for this. Each flow is specific enough to connect a particular VNF couple, matching their MAC addresses, the incoming port and the VLAN tag of their injected traffic, enabling in this way the same MAC addresses to be used by different NSs.

The shortest path connecting a VNF couple is chosen by VTN-Manager during the forwarding process of the very first ARP request. If the first packet is forwarded from VNF1 to VNF 2, then the corresponding ARP request has the MAC of VNF 1 as source address and the broadcast MAC as destination address, thus, the SDN controller learns the ingress switch and the port that VNF 1 is attached to and pushes the packet to all other switches. The other switches, in turn, forward this packet to all ports except for the ones discovered by LLDP, delivering the packet only to VNFs and not to switches. Once VNF 2 responds with an ARP reply, the SDN controller learns the ingress switch and the port of VNF 2 and calculates the shortest path between the ingress switches of the two VNFs. Then, it deploys the flows to the switches participating in this path, matching only the packets sent from VNF 1 to VNF 2.

Although Open3++ benefits from the dynamic deployment or removal of provider networks comparing to Open3, its delay performance may be worse at the first packets ex-changed between the VNF couples, due to its reactive flow configuration. This evaluation is to estimate the average time needed for the flow configuration connecting a new VNF couple, when these VNFs are i) in the same PoP or ii) in different PoPs.

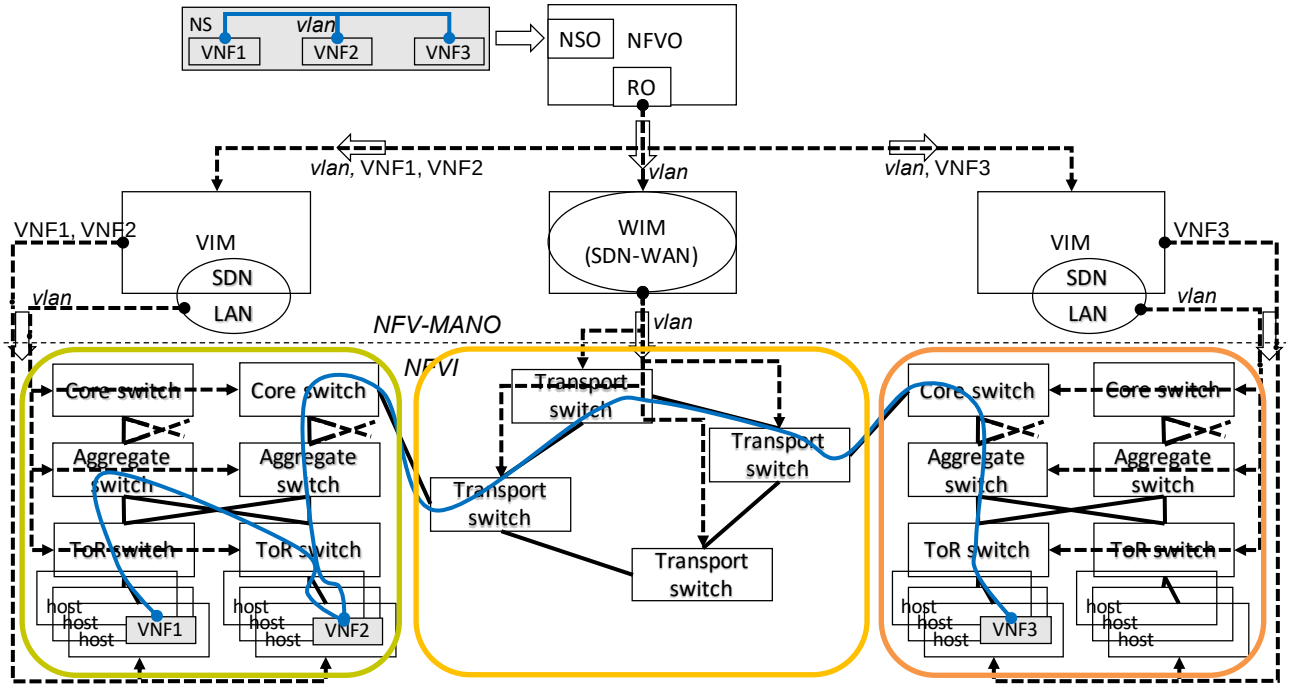


Figure 8: Simple scenario illustrating the deployment of a NS with 3 VNFs over two PoPs (or data-centers), the yellow and the red one, interconnected through a network, the orange one.

2.1.4 Intra-PoP delay

Let's assume an SDN network with multiple switches connected in a three-tier fat-tree network topology. This is a typical scenario in a PoP (or data-centre) with multiple interconnected compute nodes, using access or Top of Rack (ToR), aggregate and core layer switches [26], as it is depicted in Figure 8. In general terms, the compute nodes of a data-center are grouped to racks, each one equipped with a ToR switch connecting its

compute nodes. Each ToR switch is connected to one or more aggregate switches, which are connected to one or more core switches.

In the example of Figure 8, there is a NS with three VNFs, named VNF 1, 2 and 3. VNFs 1 and 2 are deployed in the yellow PoP, in two different compute nodes belonging to different racks. Although the ToR switches connecting these two VNFs are unique, the aggregate switch connecting the two ToR ones is not exclusive. In this example, the left yellow aggregate switch is chosen by the VTN-Manager of SDN-LAN, instead of the right yellow one. If the ToR switches cannot be physically connected through an aggregate switch, then a longer path is used, including two aggregate switches and a core one. As follows, two VNFs of the same PoP are either i) directly connected, if they are hosted on the same compute node, or connected through ii) a single ToR switch, or iii) two ToR and an aggregate switch or iv) two ToR, two aggregate and a core switch. From our experimentation in NITOS, using an out-bound Ethernet network for the control plane, we checked that the ping delay for the first packet sent from VNF 1 to VNF 2 is between 8−12 ms, including the delay of the ARP and ICMP forwarding, as well as the delay of the reactive flow configuration. The upper and lower delay limits are not significantly affected by the path length, except for the case that the VNFs are directly connected. The compute nodes are NITOS nodes, featuring Intel Core i7-2600 CPU at 3.40 GHz and 8M Cache, while the switches are emulated by executing Mininet in a separate NITOS node. All NITOS nodes are interconnected through an OpenFlow HP 3800 switch.

2.1.5 Inter-PoP Delay

In the same example of Figure 5, VNFs 2 and 3 are deployed in different PoPs, connected through a network. When VNF 2 pings VNF 3, the yellow SDN-LAN pushes the right yellow core switch to forward the ARP request of VNF 2 to the left orange switch. For the orange SDN-WAN, the left orange switch is the ingress switch of this ARP request, thus repeating the same process with before, the packet is forwarded from the right orange switch to the left red core switch. Finally, the red SDN-LAN receives this ARP request and similarly pushes it to the left red ToR switch to be forwarded to VNF 3. The duration of the ARP request forwarding from VNF2 to VNF 3 is expected to be almost three times higher comparing to the previous case, between VNF 1 and VNF2. If the involved controllers were more, having more networks between the two PoPs, the duration would be respectively higher. This is also verified by our experimentation in NITOS, since the duration of the ping between VNF 2 and VNF 3 is between 20−33 ms. The switches of each PoP are emulated executing Mininet in separate NITOS node, one for each PoP.

2.2 Multi-domain orchestration using OpenStack and Kubernetes

Pishahang[2], the multi-domain orchestrator implemented by UPB, supports orchestration of services that are distributed across infrastructures managed by OpenStack and Kubernetes [12]. Pishahang has been implemented upon state-of-the-art NFV, SDN, and Cloud computing tools and technology and provides lifecycle management and service function chaining for network services composed of Virtual Machine (VM)- and Container (CN)-based VNFs. In the Pishahang framework, OpenStack works as a service orchestrator for VM-based workloads and Kubernetes orchestrates CN-based workloads. On a higher level, Pishahang orchestrator carries out the intra-domain management tasks and orchestrate the service as a whole. Pishahang also includes a set of components (NSD splitter, NSD translator, MANO wrapper) to support hierarchical management and orchestration of services across multiple domains. We describe these components in Section 3.2. Since Pishahang has been implemented to support orchestration of multi-version services, it is comprehensively described in Deliverable 5.3 [2] that is related to multi-version services.

2.3 Slicing and VNF placement, from the perspective of RAN, using JoX

Radio access network (RAN) slicing is one of the key enablers for realizing the service-oriented 5G vision. To dynamically manage and orchestrate many slices [47], the RAN's virtualisation on the top of the same physical RAN infrastructure is needed, in order to provide customized and programmed RAN as per slice requirements. The underlying RAN infrastructure can be either monolithic or disaggregated. In the latter case, RAN is decomposed into radio unit (RU), distributed unit (DU) and centralized unit (CU) with flexible functional split among them [48]. Such disaggregation provides the capability to abstract the resources and the function modules, so that it can be reusable across slices. Additionally, different levels of isolation and sharing among resource can be provided. RAN slicing remains challenging in providing different levels of resource isolation

[2] https://github.com/CN-UPB/Pishahang

and sharing while enabling slice orchestration for multi-service RAN infrastructures. To this end, we use JoX for the E2E slicing, including RAN and radio parts.

In the following, we firstly introduce the network slicing concept, and then followed by VNF placement. After that, we presents JoX, and then an example on slices/subslices deployed is detailed.

Network Slicing: A network slice can be viewed as an independent logical network tailored to a certain service [6], while its behavior is implemented by a Network slice Instance (NSI) [7]. Accordingly, a network slice contains all the necessary elements to create an independent logical network, like a subset of physical resources and/or virtual network functions. A network slice may span all the network parts, i.e. both Radio Access Network (RAN) and Core Network (CN), forming the so-called end-to-end (E2E) network slice. The NSI can be composed of one or more network sub-slice instance (NSI) [7]. According to 3GPP, the lifecycle of an NSI consists of three phases:

1. *Preparation:* It includes the creation and verification of network slice template, in addition to the on boarding and preparing the necessary network environment that is used to support the lifecycle of the NSI. In this phase, the NSI does not exist yet.
2. *Instantiation:* Instantiation, Configuration, and Activation phase: This phase can be divided into two main steps:
    a. Instantiation and Configuration: In this step, all the resources that are shared/dedicated to the NSI are created and configured. At the end of this step, the NSI is ready for the operation.
    b. Activation: Any actions that make the NSI active (e.g., diverting the traffic to it and provisioning dedicated databases).
3. *Run-time:* In this phase, the NSI is capable of handling the traffic for supporting communication services. This phase includes supervision/reporting (e.g., Key Performance Indicators-KPI) and modification (e.g., update, reconfiguration, NSI scaling, NSI capacity/topology change, etc.).
4. *Decommissioning:* This phase includes deactivation and reclamation of dedicated resources, e.g., termination or re-use of network function(s), in addition to the configuration of shared/dependent resources. At the end of this phase, the NSI does not exist anymore.

For managing and orchestrating the lifecycle of NSI and NSSI, we use JoX [8], a tool developed by Eurecom. JoX supports the different phases of the lifecycle of the network service, as described later. Compared to 5G-PICTURE OS, JoX currently includes NFV orchestrator, slicing manager and monitoring.

VNF Placement: It is one of the more challenging issues facing the deploying of VNFs. Function placement should consider the following: i) functions' latency (since some of the functions have very strict latency like the physical layer functions), ii) functions' bandwidth for fronthaul/midhaul/backhaul that depends on the user traffic per slice and/or functional split, and iii) resource and power consumption (which is related to the first two points). To sum up, the objective of function placement is to optimisation of the resource utilisation and power consumption while respecting the requirements of every function composing the network slice, as detailed in D4.2 [3].

In JoX, we currently support two ways of (simple) placement (while placement with many objectives like resource and power optimisation is ongoing):

1. *Static placement:* In this case, the user can specify on which machine the function should be placed.
2. *Auto-placement:* In this type of placement, JoX places (deploys) the function in a dynamic way either by choosing an already exist and available resources (e.g., already exist lxd/kvm/physical machine) or by creating new lxd/kvm machine.

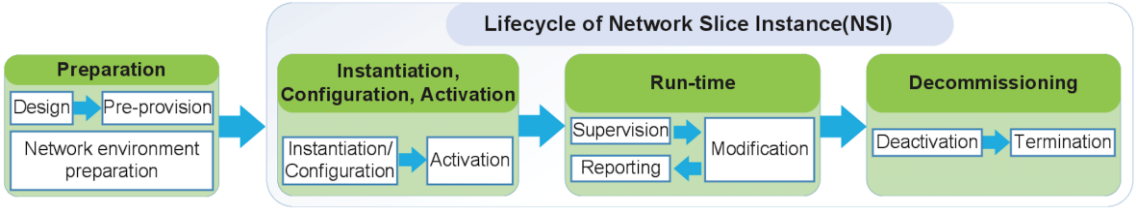


Figure 9: Lifecycle of network slice instance according to 3GPP (NSI) [7].

JoX's overview: It is domain orchestrator, where its architecture, as illustrated in Figure 10, is ETSI-compliant. JoX mainly consists of the following components:

- NorthBound Interface (NBI) [9]: JoX provides a set of endpoints allowing the user to manage and orchestrate (e.g., onboard slice template, deploy and monitor slice, delete slice, etc.) their slices and sub-slices remotely.
- Package manager: It is responsible of creating and packaging the template of slice and sub-slice, which can be onboarded later via the JoX NBI. The templates of slice and sub-slice are described according to OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) [10]. TOSCA is open standard and adopted by various lead cloud players like IBM, Rackspace, Huawei, RedHat and others.
- Network Function Virtualization Orchestrator (NFVO): The core component of JoX. According to ETSI-MANO, it is composed of two main sub-components; i) network service orchestrator (NSO) and network resource orchestrator (RO). NSO is mainly responsible for:
    - Managing the network service deployment templates and VNF Packages (e.g. VNF on-boarding.
    - Network service instantiation and lifecycle management, e.g. updating, collecting performance measurement results, and deleting the network instance.
    - Management of the instantiation of VNF Managers where applicable.

  Through the interaction with the VIM, RO is responsible for:
    - NFVI resource management (e.g., resource reservation and allocation to the network instances).
    - Locating and/or accessing one or more VIMs as needed and providing the location of the appropriate VIM to the VNFM, when required.
    - Collect usage information of NFVI resources by VNF instances, which is achieved via monitoring manager.
- Virtualized Network Function Manager (VNFM): It is responsible for managing the lifecycle of the VNFs. Juju is used as VNFM, while supporting Kubernetes as VNFM is ongoing.
- Virtualized Infrastructure Manager (VIM) for managing the infrastructure, where lxd, kvm, and physical machines are already supported.
- Network Function Virtualization Infrastructure (NFVI): It represents the available infrastructure for deploying the VNFs.
- Common services: It represents the set of common services for all or some of the components of JoX, such as database data repositories, JoX store (where the onboarded packages are stored there), etc.
- Set of plugins to interact with other components: JoX supports many plugins like elastic search network monitoring, and FlexRAN as Controller. More details on FlexRAN are provided in deliverable D4.2 [3].

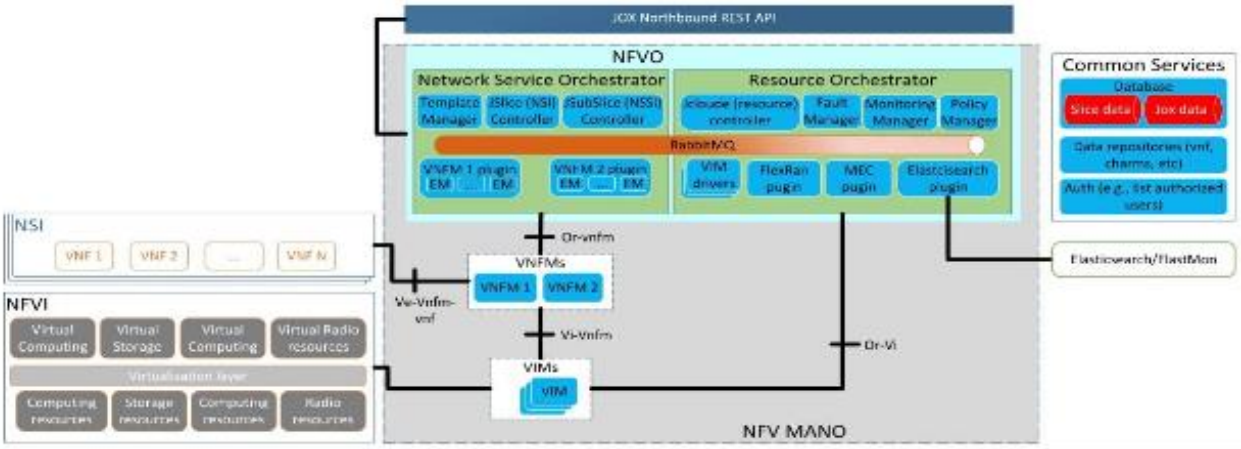


Figure 10: JoX architecture.

Note that JoX is already reported in deliverable D5.1 [1], where we, here, highlight the main functionalities and novelties brought to JoX:

1. New design with ETSI-compliant architecture.

2. Support slicing and network slicing.
3. Template compliant with TOSCA-Simple-Profile-YAML-v1.2.
4. Package manager.
5. Rich NBI for managing and monitoring slices and subslices.
6. Support lxd/kvm/physical machines as VIM.
7. Set of plugins to interact with other components (e.g., FlexRAN).

Example on network slicing: Here, we show a simple example on how the network slicing is realized via JoX (please refer the following sections for more details on JoX architecture). Figure 11 illustrates a slice (dubbed as NS in the figure) consisting of three sub-slices (dubbed as NSSI_1, NSSI_2, and NSSI_3). The first two subslices represent the core network part, while the third subslice represents the RAN part. Note that the green line between the applications represents the relation.

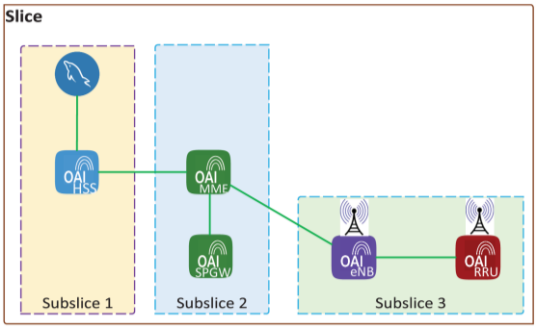


Figure 11: Deploying oai-5g-cran slice suing JoX.

The template of the slice is illustrated in Figure 12, which consists of the three main parts:

1. *Header*: It may contain different information that is general (e.g., *description* and *metadata*) and additional important inputs like the *inputs*. In our example, the *inputs* tell us that there are two subslices in the current slice.
2. *Body*: The body consists of many parts, such as the relations (*relationships_template*) and topology templates (*topology_template*). In the *topology_template*, which represents subslice) in the slice, where a node (subslice) can have one or more egress/ingress function, i.e. the functions that have relations with other functions hosted on different subslice(s). The *relationships_template* defines this relations between the functions hosted in different subslices.

```
description: template for deploying package <nsi-oai-5g-cran> with 3 subslices
tosca_definitions_version: tosca_simple_yaml_1_0

imports: [nssi_1, nssi_2, nssi_3]
metadata:
  ID: nsi-oai-5g-cran
  author: Eurecom
  vendor: Eurecom
  version: 1.0
  date: 2019-03-10}

relationships_template:
  connection_epc:
    type: tosca.relationships.ConnectsTo
    source:
      inputs:
        type: tosca.relationships.AttachesTo
        name: eg1
        node: oai-hss
      node: nss_first
      parameters: nssi_1
    target:
      inputs:
        name: ing1
        node: oai-mme
        type: tosca.relationships.DependsOn
      node: nss_second
      parameters: nssi_2
  connection_ran:
    type: tosca.relationships.ConnectsTo
    source:
      inputs:
        name: eg1
        node: oai-mme
        type: tosca.relationships.AttachesTo
      node: nss_first
      parameters: nssi_2
    target:
      inputs:
        name: ing1
        node: oai-ran
        type: tosca.relationships.DependsOn
      node: nss_second
      parameters: nssi_3

topology_template:
  node_templates:
    nss1:
      requirements:
        egress:
          eg1:
            node: oai-hss
            relationship:
              type: tosca.relationships.AttachesTo
        nssi: nssi_1
      type: tosca.nodes.JOX.NSSI
    nss2:
      requirements:
        ingress:
          ing1:
            node: oai-mme
            relationship:
              type: tosca.relationships.DependsOn
        nssi: nssi_2
      type: tosca.nodes.JOX.NSSI
    nss3:
      requirements:
        egress:
          eg1:
            node: oai-mme
            relationship:
              type: tosca.relationships.AttachesTo
        nssi: nssi_2
      type: tosca.nodes.JOX.NSSI
    nss4:
      requirements:
        ingress:
          ing1:
            node: oai-ran
            relationship:
              type: tosca.relationships.DependsOn
        nssi: nssi_3
      type: tosca.nodes.JOX.NSSI
```

Figure 12: Slice template for deploying EPC via JoX.

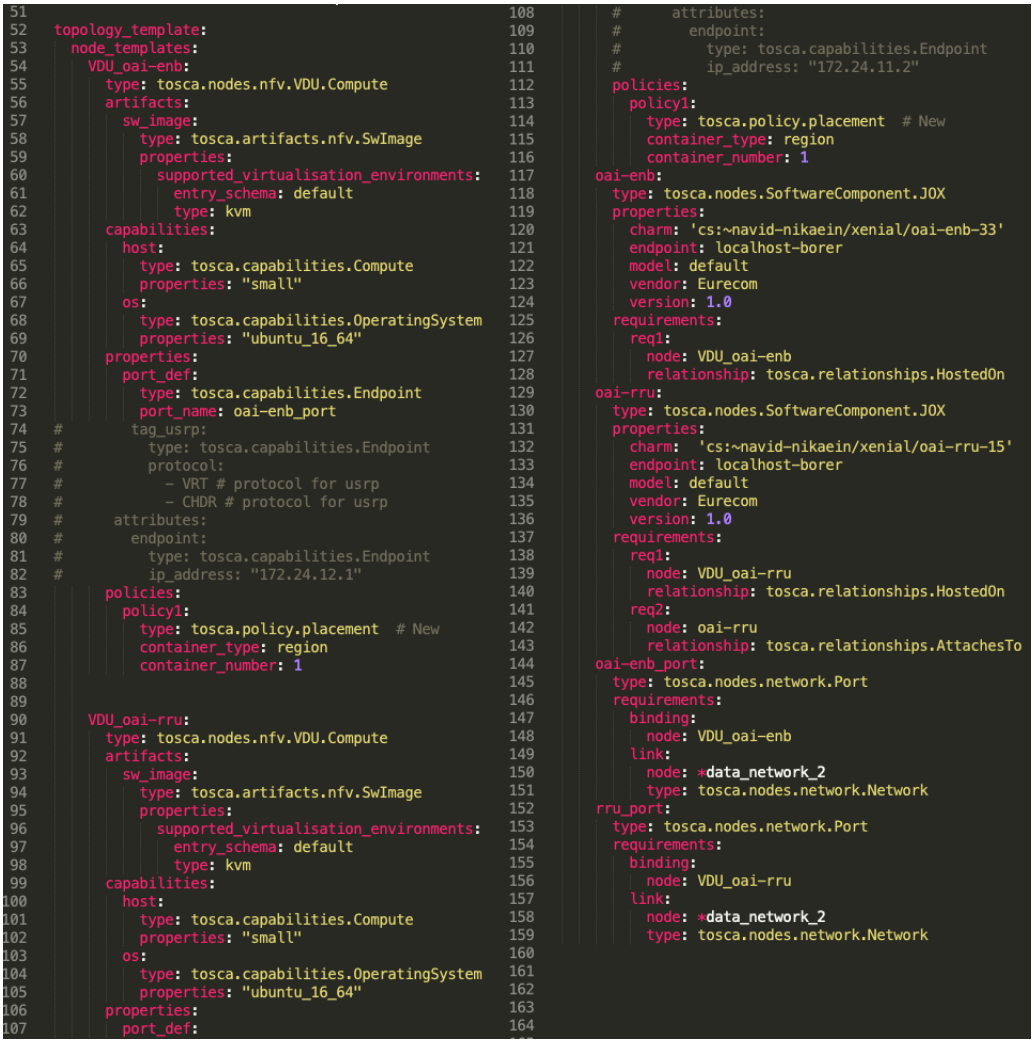
```yaml
description: template for deploying package <oai-epc> with 2 subslices (nssi_1)
tosca_definitions_version: tosca_simple_yaml_1_0

imports: []
metadata:
  ID: nssi_1
  author: Eurecom
  vendor: Eurecom
  version: 1.0
  date: 2019-03-08

dsl_definitions:
  host_tiny: &host_tiny
    disk_size: 5
    mem_size: 512
    num_cpus: 1
  host_small: &host_small
    disk_size: 5
    mem_size: 1024
    num_cpus: 1
  os_linux_u_16_x64: &os_u16
    architecture: x86_64
    distribution: Ubuntu
    type: Linux
    version: 16.04
  os_linux_u_18_x64: &os_u18
    architecture: x86_64
    distribution: Ubuntu
    type: Linux
    version: 18.04
  data_network: &data_network_1
    properties:
      network_name: "net1"
      ip_version: 4
      cidr: "10.70.21.0/24"
      start_ip: ""
      end_ip: ""
      gateway_ip: "0.0.0.0"
  data_network_2: &data_network_2
    properties:
      network_name: "net2"
      ip_version: 4
      cidr: "192.168.122.0/24"
      start_ip: ""
      end_ip: ""
      gateway_ip: "0.0.0.0"
  containers:
    region1:
      - network: *data_network_1
    region2:
      - network3: *data_network_2
```

Figure 13: Header of the template of the first sub-slice.

Similarly, the template of a subslice consists of header and body, as illustrated in Figure 13 and Figure 14 respectively, that represents the subslice 3 (NSSI_3). Note that since the template of the three subslices are similar, we introduce only the RAN-related subslice. In addition to what is defined for the slice, we can see from Figure 13 that other common information can be defined, such as the description of the host and operating system (OS) that is needed to deploy a target application. Moreover, for the placement, it is also possible with JoX to define the domain from which the resources to deploy an application are taken. In the body, we can define the nodes, that can be either of type *VDU* (virtualized data unit), which describes the resources required to deploy a function, or of type *softwareComponent* that describes that function. We can also see from Figure 14 that a node of network type is defined, where it is destined to define the network to which the VDU will be attached.



```
topology_template:
  node_templates:
    VDU_oai-enb:
      type: tosca.nodes.nfv.VDU.Compute
      artifacts:
        sw_image:
          type: tosca.artifacts.nfv.SwImage
          properties:
            supported_virtualisation_environments:
              entry_schema: default
              type: kvm
      capabilities:
        host:
          type: tosca.capabilities.Compute
          properties: "small"
        os:
          type: tosca.capabilities.OperatingSystem
          properties: "ubuntu_16_64"
      properties:
        port_def:
          type: tosca.capabilities.Endpoint
          port_name: oai-enb_port
#         tag_usrp:
#          type: tosca.capabilities.Endpoint
#          protocol:
#            - VRT # protocol for usrp
#            - CHDR # protocol for usrp
#       attributes:
#         endpoint:
#           type: tosca.capabilities.Endpoint
#           ip_address: "172.24.12.1"
      policies:
        policy1:
          type: tosca.policy.placement  # New
          container_type: region
          container_number: 1


    VDU_oai-rru:
      type: tosca.nodes.nfv.VDU.Compute
      artifacts:
        sw_image:
          type: tosca.artifacts.nfv.SwImage
          properties:
            supported_virtualisation_environments:
              entry_schema: default
              type: kvm
      capabilities:
        host:
          type: tosca.capabilities.Compute
          properties: "small"
        os:
          type: tosca.capabilities.OperatingSystem
          properties: "ubuntu_16_64"
      properties:
        port_def:
#       attributes:
#         endpoint:
#           type: tosca.capabilities.Endpoint
#           ip_address: "172.24.11.2"
      policies:
        policy1:
          type: tosca.policy.placement  # New
          container_type: region
          container_number: 1
    oai-enb:
      type: tosca.nodes.SoftwareComponent.JOX
      properties:
        charm: 'cs:~navid-nikaein/xenial/oai-enb-33'
        endpoint: localhost-borer
        model: default
        vendor: Eurecom
        version: 1.0
      requirements:
        req1:
          node: VDU_oai-enb
          relationship: tosca.relationships.HostedOn
    oai-rru:
      type: tosca.nodes.SoftwareComponent.JOX
      properties:
        charm:  'cs:~navid-nikaein/xenial/oai-rru-15'
        endpoint: localhost-borer
        model: default
        vendor: Eurecom
        version: 1.0
      requirements:
        req1:
          node: VDU_oai-rru
          relationship: tosca.relationships.HostedOn
        req2:
          node: oai-rru
          relationship: tosca.relationships.AttachesTo
    oai-enb_port:
      type: tosca.nodes.network.Port
      requirements:
        binding:
          node: VDU_oai-enb
        link:
          node: *data_network_2
          type: tosca.nodes.network.Network
    rru_port:
      type: tosca.nodes.network.Port
      requirements:
        binding:
          node: VDU_oai-rru
        link:
          node: *data_network_2
          type: tosca.nodes.network.Network
```

Figure 14: Body template of the third sub-slice (RAN sub-slice).

2.4 Slicing and VNF placement, from the perspective of transport network

The Zeetta Slicing Engine (referred to as the Engine) is an implementation of the Domain Orchestrator component. It sits under a Multi-Domain Orchestrator (e.g. Pishahang) and above Domain Controller (e.g. NetOS Controller/Virtualisation Engine, OpenDayLight, etc.) and NFV MANO components (e.g. OSM). The northbound interfaces allow provisioning of network connectivity with different levels of service, where each level maps to a particular QoS policy. In the project demonstrations, the Engine is used to provide a transit network to the 5G OS Operator (see Section 4.1) for connectivity with three levels of service: Gold, Silver and Bronze. Beyond basic connectivity, it targets three main VNF use-cases:

1. VNF requested by the user as part of the Slice definition, here VNF deployment facility has been exposed to the slice requester.
2. Slice definition is implemented, either fully or partially, using VNFs, here the slice definer may or may not be aware that VNFs are being used to implement the slice.
3. Slice definition contains resources that can be used to deploy VNFs, here the slice definer has asked for a slice that can support VNFs.

Slicing Engine Outline

The Zeetta Slicing Engine has two major components: Orchestrator and Virtualization Engine. This can be seen in Figure 15. They communicate with each other via the Mapping interface. The Orchestrator calculates the mapping of the requested slice over the underlay network and provides it to the Virtualization Engine which is responsible for implementing the slice in the underlay and providing northbound Use APIs for the next layer of slices. The layer of slices can be built up with each slice using the Use APIs exposed by the slice below. At

the lowest layer the northbound APIs provided the network devices/controllers/orchestrators. The Orchestrator has its own Slice Definition APIs.

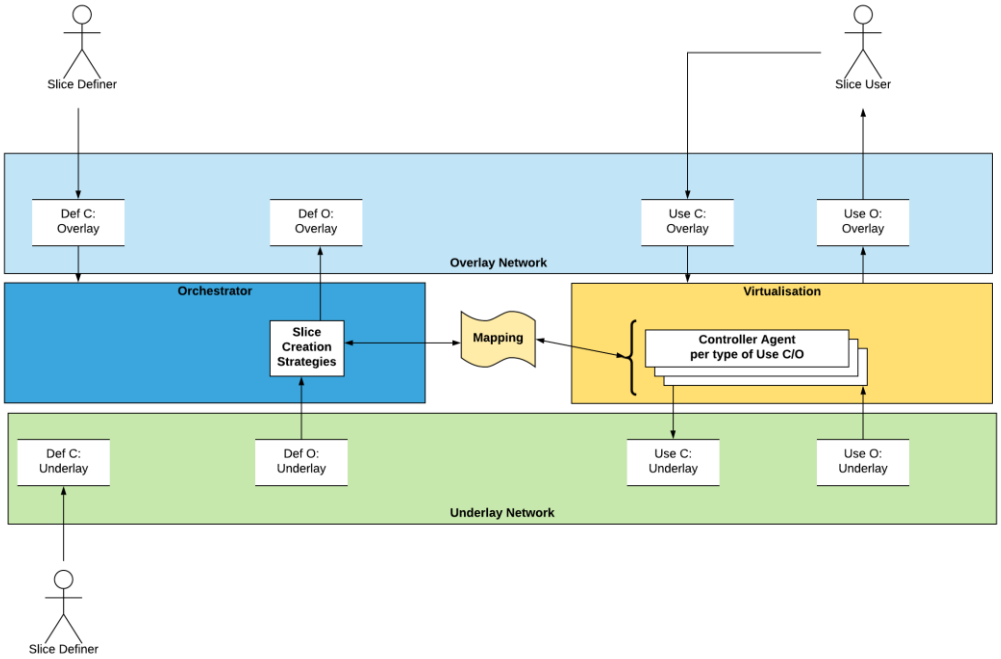


Figure 15: Zeetta Slicing Engine with Definition, Use and Mapping APIs.

Details of the Zeetta Slicing Engine can be found in deliverable D5.1 [1].

VNF Use in Slice Implementation

In this scenario, the Slice Definer requests a network slice with a set of services. The Orchestrator in this case has a choice: to implement the slice using physical devices to provide the services requested or to instantiate VNFs to provide them. The VNFs provide additional flexibility but also add overhead.

Using the concept of function layering, the functions in the top layers, so called user exposed functions (see Overlay Use-C and O in Figure 15), can be virtualized independent of the underlay function type (whether PNF or VNF). The actual mapping to VNFs/PNFs can happen in the lower layers.

The Slice Creation Strategy ensures optimal use of the two options while working within operational and commercial constraints. The Strategy needs to delegate VNF management to the Virtual Resource Manager (VRM). This setup is shown in Figure 16. VNFs can be either VMs (OSM) or Containers (OSM over Kubernetes) or whatever abstraction the underlay supports. As long as we have the appropriate plugin within the VRM to drive the provider in the underlay it should be possible to provision the required VNF and layer virtualized components over it. This is a DO-MANO interface implementation.

The VRM has three important functions:

1. Decide placement and version of the VNF to be used.
2. Drive MANO to create/configure/modify/decommission/monitor/connect VNFs.
3. Ensure Use APIs are available to the Virtualization component.

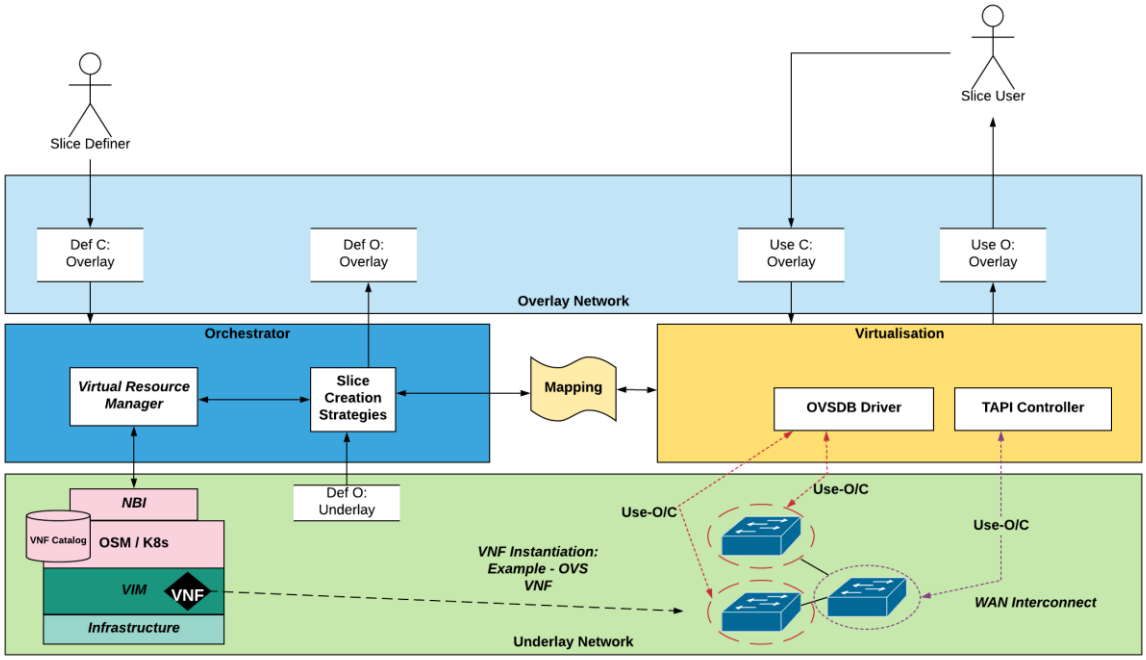


Figure 16: Zeetta Slicing Engine using VNFs to implement a slice without Slice Definer asking for a VNF.

The VNFs will provide their own northbound Use APIs that the Virtualization Tactics can use to configure and provide Use APIs of their own for the next layer up. For example: if the VNF is an Open vSwitch implementation then the VNF should appear as an OpenFlow switch with OpenFlow and Open vSwitch DataBase (OVSDB) interfaces. The OVS interfaces can be used by the Virtualization Tactics to provide different Use APIs northwards such as L2 Connectivity (underlay: OVSDB, overlay: L2-Connectivity with VLAN Configuration) and Priority (underlay: OpenFlow, overlay: Priority Definition API).

VNFs Requested in Slice Definition

Slice definitions can explicitly request for VNFs that provide various services. These VNFs may be definer supplied or provided by the Service Provider. In case the MANO is being provided internally (by the Service Provider) then the request can be processed as shown in Figure 16. This is also the case where MANO is provided externally but as a dedicated instance to the Service Provider.

In case the MANO is provided externally then the Virtual Resource Manager has lot less control and needs to delegate responsibility to the external MANO provider. It is assumed here that the Slice Definer provides VNF artefacts (containers/VM images). The Service Provider still needs to plan the VNF placement and use the appropriate interface to the external provider to manage the deployment. In this case the Service Provider is dependent on the external provider for the VNF management interface (Item 2 in previous section) as well as the Use APIs (Item 3 in previous section). Therefore, the Slice Creation Strategy delegates the VNF placement to a VNF Placement Planner component. This is shown in Figure 17.

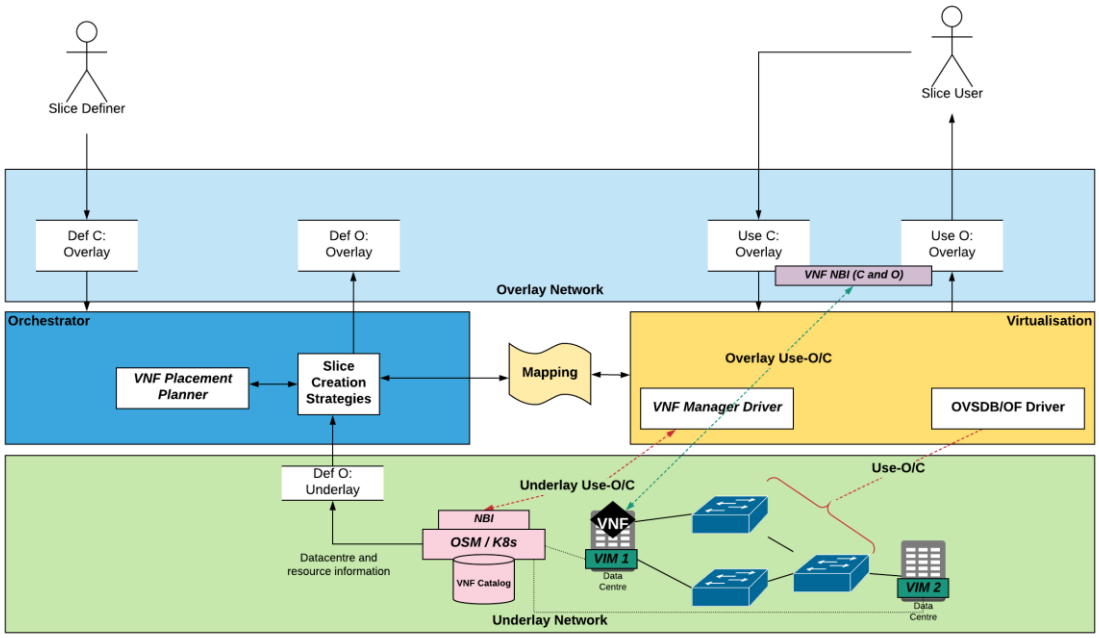


Figure 17: Zeetta Slicing Engine implementing a slice definition that includes VNFs.

Slice to Provision VNFs

This is the scenario where the Slice Definition request a slice capable of hosting VNFs. This is a Mobile Virtual Network Operator use-case. In this case the underlay must provide hosting facility for a virtual MANO instance (e.g. OSM). As in the previous cases here the virtual MANO instance must be placed on the underlay. The management of this virtual MANO and exposing its APIs northwards so that the Slice Definer can provide slices that make use of VNFs. This is shown in Figure 18.

A Slice that allows VNFs deployment can be used as a base for the previous VNF use-cases.

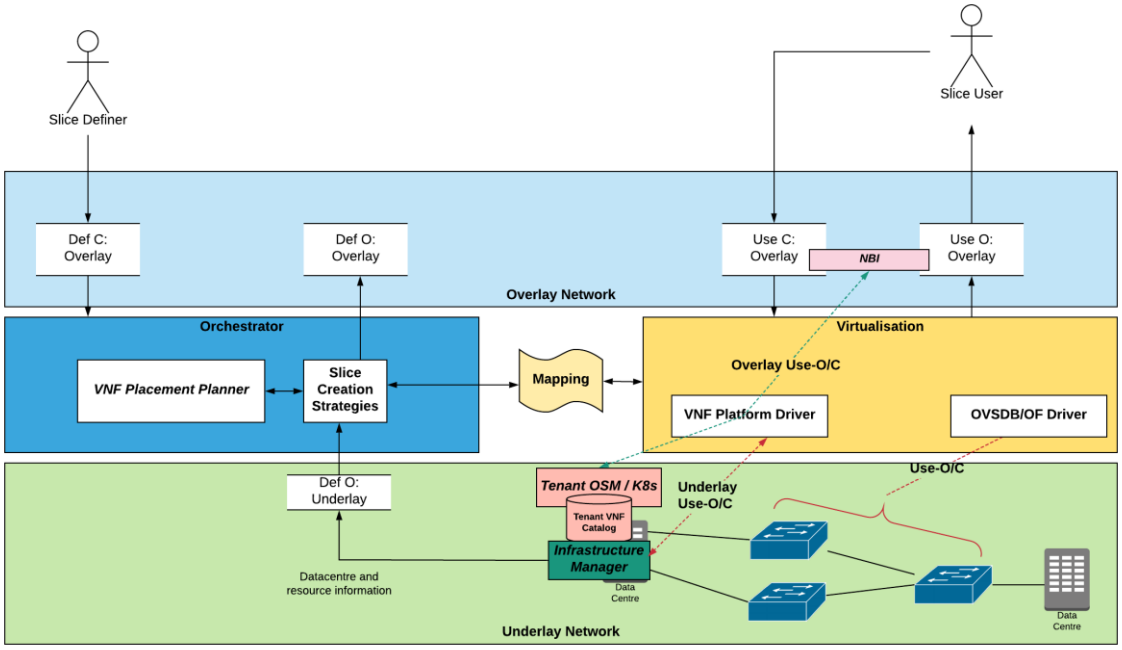


Figure 18: Zeetta Slicing Engine providing a Slice that can be used to provision VNFs.

The main challenge here is to ensure all components required for provisioning VNFs are present. This includes underlay network access for connecting the VNFs to each other as well as external connectivity.

VNF Placement

A significant Orchestrator problem in the above scenarios is that of VNF placement. This requires the Orchestrator to solve a multi-objective mapping problem with a diverse set of constraints. The objective may be cost minimization, network latency minimization, resource minimization or a combination of these and other objectives.

VNFs require compute resources and network connectivity. There are certain constraints on the compute side depending on the VNF requirements (for example request for specific compute resources like GPUs). The network connectivity also has constraints related to the overall performance of the VNFs that the service is composed of. The network is a shared resource which may or may not be under the direct control of the Service Provider.

Therefore, the VNF Placement must consider the network connectivity when deciding where to place different VNFs. Ideally all the VNFs that are part of a given slice should be mapped to a single data-centre. But this is not always possible (e.g. due to unavailability of special resources like GPUs) or desirable (e.g. where the slice consists of edge and cloud compute requirements).

In such cases the network connectivity between the data-centres could be evaluated first and only those data-centres (whether edge or cloud) considered that can be interconnected without violating any constraints. This sub-set of data-centres can then be evaluated for VNF placement.

In case of network failures, network connectivity should be re-evaluated between available data-centres before planning VNF migrations.

3 Auto-adaptive and Hierarchical Management, Orchestration and Control

In this section, we present the auto-adaptive and hierarchical management, orchestration and control of 5G OS. In Section 3.1, we present the hierarchical Controller, which enables 5G OS to handle a variant number of domains, since domains are added or removed when the network scales up or down. In Section 3.2, the corresponding hierarchical NFV MANO is described, which complements the hierarchical Controller. The auto-adaptive nature of 5G OS is illustrated in Section 3.3, which shows how 5G OS is able to dynamically insert new eNodeBs to the RAN, as well as in Section 3.4, which addresses how 5G OS deals with the controller placement problem (how many controllers and where they should be placed).

3.1 Controller hierarchy

We present a control plane architecture for multi-tenant multi-domain 5G transport networks, as envisioned by this project. Figure 19 gives an overview of this architecture, which is also presented in [27]. To the best of our knowledge, the proposed solution is the first hierarchical SDN control plane supporting virtualization in a multi-domain (and multi-PoP) environment. We extend our work in Section 2, which presents SDN-WAN and SDN-LAN in a single domain with multiple PoPs, in a way that a scalable hierarchy of multiple controllers at different levels is created. As we will describe later in detail, we formulate a controller hierarchy that uses SDN-LANs as local agents and SDN-WANs as Level-0 controllers at the basis of the hierarchy, which is completed with the addition of two extra levels of controllers, Level-1 and Top level. We evaluate scalability of our proposed architecture experimentally, measuring path provisioning latency for different number of domains, and find it to be well below 1 second for most scenarios of practical interest for 5G transport networks.

3.1.1 Layer 2 Overlay to support virtualization

In terms of the dataplane abstraction, the 5G-PICTURE transport network can be seen as a virtual network, where each tenant brings its virtual entities, namely Virtual Network Functions (VNFs) and virtual DataPaths (vDPs). VNFs are the end points and the vDPs are the tenant controlled datapaths. Each virtual network is a slice of the transport network infrastructure. A transport slice is composed of virtual Layer 2 segments where virtual entities (VNFs/vDPs) are attached. A virtual Layer 2 segment emulates a broadcast domain and is identified by a Layer 2 Segment Identifier (L2SID). L2SIDs are unique system wide, meaning that L2SIDs cannot be reused within or across slices. VNFs are identified within a slice by a Media Access Control (MAC) address scoped to a single Layer 2 segment. vDPs contain custom network control logic defined by the tenant, for instance they may correspond to a virtual switch. Unlike a VNF, a vDP may have several interfaces, each one connected to a different virtual Layer 2 segment. Each interface of a vDP is identified again by a MAC address scoped to the L2SID where it is attached. Note that, in our architecture, Layer 3 forwarding between different virtual Layer 2 segments is the responsibility of the tenant and can take place at its vDPs, that is, only a Layer 2 overlay is provided to the tenants as a service.

3.1.2 SDN based multi-domain transport underlay

The 5G-PICTURE architecture is deployed over a transport network infrastructure consisting of domains, often technology-specific, to which we also refer as areas. It is composed of three main functions in the dataplane, namely the Transport Nodes (TNs), depicted as circles in Figure 19; the Edge Transport Nodes (ETNs), depicted as squares in Figure 19; and the Inter-Area Transport Nodes (IATNs), depicted as triangles in Figure 19.

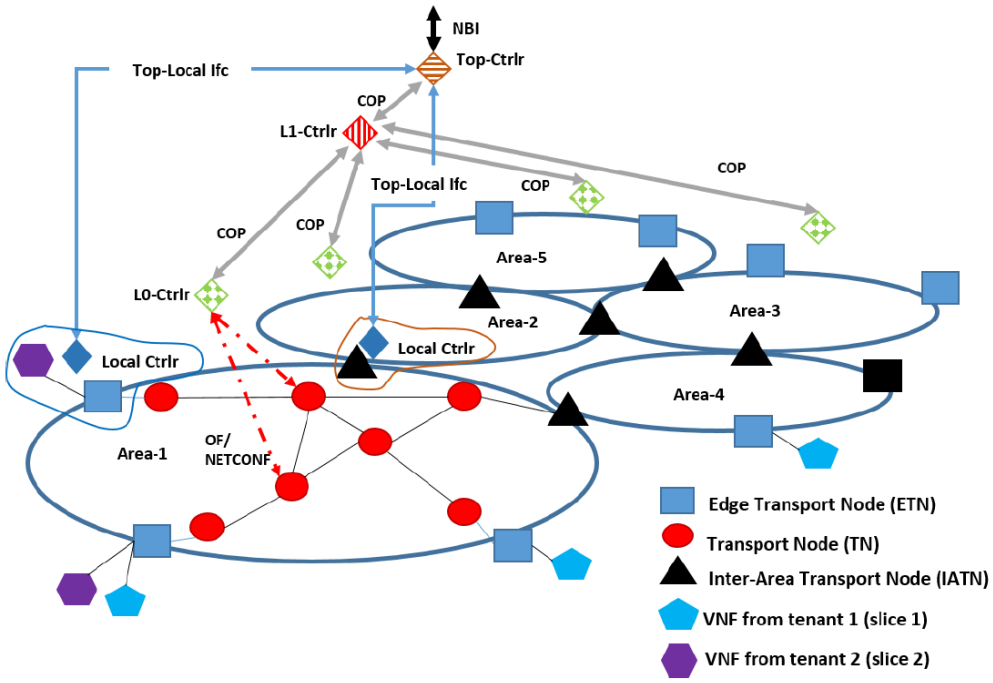


Figure 19: Overview of the proposed transport network architecture and associated control plane hierarchy.

All transport nodes embed one major function, the Forwarding Information Base (FIB). FIB is in charge of forwarding packets between VNFs/vDPs, which are either collocated in a single ETN or bounded to different ETNs. In the second case, packets are inserted into pre-instantiated transport tunnels, implemented with use of encapsulation. Traffic from multiple slices can be combined into a single tunnel. The Transport network Adaptation Function (TAF) is responsible for pushing (or popping) the corresponding transport header before (or after) injecting the packets into the transport network, whereby the transport header signals at least three major pieces of information: i) the address of the destination ETN (which might be located in the same or in another area), ii) the transport slice ID and iii) the tunnel ID. Tunnel IDs determine how encapsulated packets are routed throughout the underlay. They are area specific, meaning they generally change from area to area, assuming that the tunnel spans multiple areas. Transport slice IDs can be used to apply differentiated policies depending on the slice, such as binding slice traffic to a particular Quality of Service (QoS) class. Only ETNs/IATNs feature a TAF, corresponding to the transport technology used in the area where the tunnel is located.

The TNs act as regular tenant-agnostic transport nodes, and could be instantiated by different technologies, such as wireless or Time Shared Optical Networks (TSON). The TN datapath forwards incoming packets according to their tunnel ID, and optionally applies differentiated policies according to the transport slice ID. ETNs and IATNs are interconnected through transport tunnels, which are based on the forwarding services of the TNs.

An ETN would typically be implemented as a software datapath in a network hypervisor, which is equivalent to the external bridge component in Neurton (br-ext), and holds all tenant related state thus enabling virtualization/multi-tenancy over the 5G-PICTURE transport network. The main function of each ETN is to host virtual entities from several tenants, as well as to offer a datapath abstraction that connects the hosted virtual entities to the transport network. ETN implements FIB and TAF. In Figure 20 we see the flow table structure of an ETN, following the OpenFlow v1.3 specifications and using Provider Backbone Bridging (PBB) as the transport encapsulation technology.

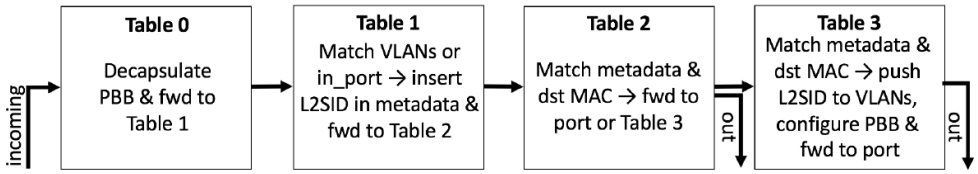


Figure 20: ETN as a datapath.

Finally, an IATN (black triangle in Figure 19) stitches pre-defined connections between two or more areas. Each IATN is in charge of collecting all incoming packets from one area and forwarding them to the appropriate tunnels of another area. For each incoming packet, IATN first executes TAF to decide which tunnel will be used for the forwarding of this packet (tunnel ID and possibly even the transport technology could be different between adjacent areas), and then changes its tunnel ID and forwards it through the appropriate interface completing the FIB action.

3.1.3 Hierarchical Control Plane Architecture

The control plane of 5G-PICTURE transport network is designed based on the principles of full address space virtualization and scalability. It is composed of a hierarchy of logical controllers, as illustrated in Figure 19. The top-level controller, referred to as the Top controller, is responsible for provisioning per tenant slices and orchestrating the required connectivity across different areas (e.g. optical transport domain, wireless transport domain). The Level-0 controller (that corresponds to SDN-WAN of Section 2.1), also referred to as Area controller, is responsible for the provisioning and maintenance of transport tunnels between ETNs and IATNs of a given area; a Level-0 controller operates at the level of individual network elements. A set of Level-0 controllers, which are technology-specific, are logically organized under a Level-1 controller. The latter is technology-agnostic, is in charge of maintaining connectivity between the corresponding areas, and operates with a higher level of abstraction, namely maintains state at the area level. Finally, ETNs and IATNs, which lie at the edges of transport areas, are directly controlled by Local Agents (that correspond to SDN-LAN of Section 2.1), which are the glue between their datapaths and the Top controller with which they interact. TNs are directly controlled by a Level-0 controller, and do not feature such Local Agents.

The Level-0 controllers have been implemented as control plane functions in WP4 and reported in D4.2 [3], while the Level-1 and Top controllers are being implemented and validated as part of D4.3 [4]. The supported transport technologies are the ones defined in WP4, namely TSON in the optical domain, TSN in the Ethernet domain, mmWave/IEEE 802.11ac in the wireless domain and the Zeetta slicing engine for the slicing support.

The main principle adopted in the design of the higher layers of the 5G-PICTURE control plane is the separation of responsibilities between the L1 controller and the Top controller, whereby the Top controller interfaces with the ETNs, e.g. in order to provision a new VNF, and with the IATNs in order to stitch domains. On the other hand, the L1 controller's job is to act as an aggregator of L0 controllers, thus interacting only with these controllers, which end up programming the TNs of each domain.

A local ETN agent's main responsibility is to maintain mappings from virtual entity addresses to the remote ETNs hosting them, whenever these entities are attached to the same Layer 2 segment as at least one virtual entity hosted locally at the agent's ETN. For these remote ETNs of its interest, it also maintains mappings to the respective tunnel IDs that must be used for forwarding encapsulated traffic towards them. Here we note that in cases where a tunnel traverses multiple areas, only the first tunnel ID is stored, the one leading to the first IATN along the route. This is in accordance with abstracting out unnecessary information; the ETN does not have to care about whether the destination ETN lies in the same or another area, in fact it does not know anything about areas. The ETN Local Agent also maintains mappings from local ports to the respective L2SIDs, and from local virtual entity addresses to local ports. These mappings are summarized in Table 1.

Table 1: ETN Local Agent mappings.

Key	Notes	Value
port	All local ports	L2SID
{L2SID, MAC}	All local virtual interfaces	port
{L2SID, MAC}	Remote virtual interfaces with an L2SID locally present	hosting ETN
destination ETN	ETNs hosting virtual entities attached to at least one L2SID locally present	Tunnel ID

A local IATN agent manages a table of mappings, which allows the underlying IATN to forward packets across areas using the appropriate interface, as well as to modify the tunnel ID information, when required. This mapping can be viewed in Table 2.

Table 2: IATN Local Agent mapping

Key	Value
{in port, in tunnel ID}	{out port, out tunnel ID}

3.1.4 Inter-Controller Communications

There are three main interactions taking place among the controllers in the hierarchy. The Top controller communicates with the ETN/IATN Local Agents, the Top controller communicates with the L1 controller, and the L1 controller communicates with L0 controllers.

The Top-Local interfaces are different for the Local Agents hosted at ETNs and those hosted at IATNs, as these types of nodes are responsible for different functionalities. Both of them, however, are Representational State Transfer (REST) based interfaces.

In the case of ETNs, the basic resources made available to the Top controller are called virtual interfaces, as an abstraction that covers both interfaces directly connected to a VNF, and interfaces of a virtual datapath (vDP) in a tenant's slice topology. These are uniquely identified by the L2SID where they are attached and the corresponding MAC address, scoped at the specific L2SID. Each virtual interface is hosted at one and only one ETN at a given time (migrations are of course allowed), and this information is a property of the virtual interface resource. The other type of managed resource are tunnels, also called paths, whose source node is the particular ETN, and destination is some remote ETN. These are identified by the (area-specific) tunnel ID attached to packets of that path leaving the ETN, which in general is a VLAN ID field. They feature at least one associated value, indicating the destination ETN, but they can also optionally feature information about the QoS guarantees provided by that path, if any, thus allowing differentiated tunnel selection decisions for flows of special requirements.

In the case of IATNs, the resources exposed and managed by the Top controller are tunnel mappings between areas attached to the IATN. The tunnels are organized by incoming IATN port, since in our design each IATN port corresponds to an area. They are uniquely identified by this port and the incoming tunnel ID, while the associated properties are the outgoing tunnel ID and the outgoing IATN port. Remember that these translations are required to keep full independence of different areas (domain stitching functionality). So, in fact the IATN Local Agent is a simple thin layer acting as a proxy between the Top controller and the IATN datapath.

Based on the interaction with the local agents of ETNs and IATNs just described, the Top controller is responsible for performing a number of operations. The first one is deployment of VNFs and vDP interfaces at specific ETNs, thereby instantiating a given tenant slice virtual topology step by step. Another operation is to provide the functionality of simple migration of virtual interfaces from one ETN to another. The important thing to realize is that the Top controller makes sure that each ETN is kept up to date about the location of all virtual interface addresses it might need to send packets to. This requires a global view of the location of all virtual interfaces, and could not be addressed locally. Yet another operation is informing the local agents of all tunnel IDs they should be aware about. This comes as the subsequent step after an initial step of actual path establishment, which involves the interaction between the Top controller and L1 controller. This interaction is analyzed in the following subsection.

The interactions between Top controller and L1 controller, as well as between L1 controller and L0 controllers, make use of the Control Orchestration Protocol (COP), originally defined in the Strauss project [28] and extended in 5G-Xhaul [5]. COP is a REST based protocol, which defines a set of data models to allow REST endpoints to offer network related services. COP has been proposed as a research-oriented transport API, technology and vendor independent, that permits to abstract technology specifics of a given transport domain. It provides a multi-layer hierarchical control plane approach using YANG and RESTconf.

The following two main services are offered by the L1 controller towards the Top controller:

- A topology dissemination service, able to retrieve topologies from individual L0 controllers and then aggregate them into an end to end topology.
- A path provisioning service, able to receive a path provisioning request involving nodes in different areas and resolve it into separate path requests for each of the involved areas. If there are multiple paths between two areas, the L1 controller performs routing at the area level.

In addition, the L0 controller offers the same two services towards the L1 controller:

- A topology dissemination service, in which the L0 controller exports the topology specific to its domain in COP format. We note that the L0 controller might choose to report only a summarized version of its topology, where the only critical information to be exposed to the L1 controller are the nodes connecting to an ETN or an IATN.
- A path provisioning service, whereby the L0 controller receives a request to connect to TNs under its control, and the L0 controller responds with the corresponding tunnel identifiers.

The COP service-topology data model consists of a list of nodes, representing network devices, and edges, representing network links. In COP, nodes embed multiple edge-ends, representing a port or an interface, and edges refer to the nodes at each side of the link.

Since the L1 controller only interacts with the L0 controllers, not ETNs or IATNs, the COP topology will only report TNs in its list of nodes. However, the Top controller needs to be able to resolve an ETN or IATN into a TN in order to issue a path request to the L1 controller. Since the mapping between ETN/IATNs and TNs is expected to be something fairly static, we opt to manually provision the L1 controller with this information. In particular, we enable an additional REST endpoint in the L1 controller that allows to specify information about IATNs and ETNs connecting to one of the L0 domains under the control of this L1 controller. Then, the L1 controller exposes the IATN/ETN information as a COP edge. This information is sufficient for the Top controller to match ETN/IATNs with TNs and issue a path request.

In particular, the L1 controller maps 5G-PICTURE functions into COP topology objects in the following manner:

Node	TN_A
Regular Edge	TN_A : PortA → TN_B : PortB
IATN Edge	$IATN_A$: PortA → $IATN_B$: PortB
ETN Edge	ETN_A : PortA

The basic operation that requires interactions among all controllers in the hierarchy is that of an end-to-end path establishment between two ETNs that are located in different areas. The time sequence of interactions required for this procedure is summarized in Figure 21 for an example with two adjacent areas.

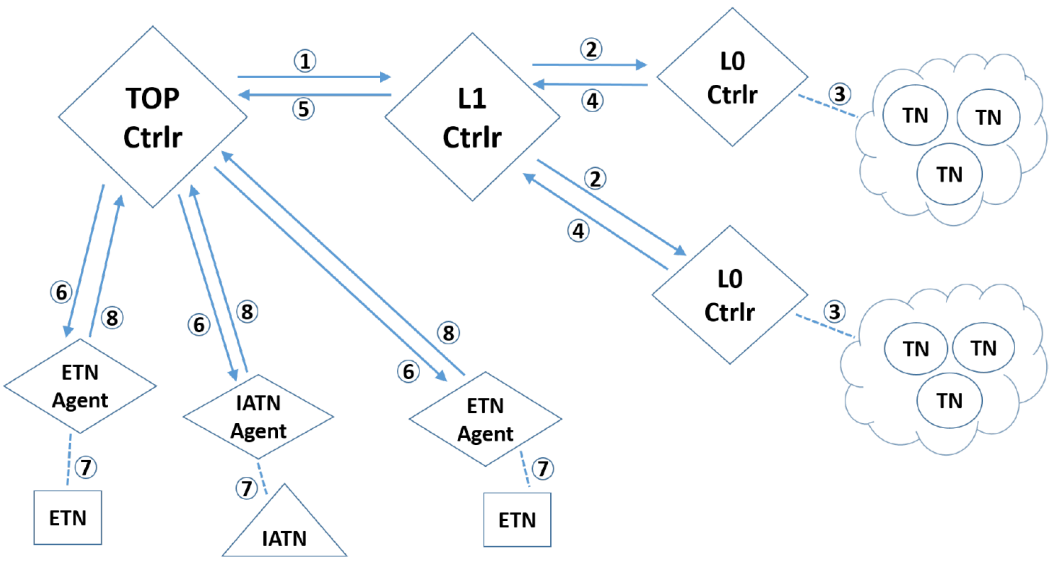


Figure 21: Interactions among controllers for end to end path establishment.

At step 1, the Top controller sends a request to the L1 controller, using COP/REST for establishing a path between two ETNs. The L1 controller examines the request and finds out that the path must traverse two adjacent areas. It then breaks the original request into two sub-requests, intended for the L0 controllers of these areas. Essentially, these subrequests, sent in parallel via COP/REST at step 2, are for provisioning of connections between each ETN and the IATN stitching these areas. The L0 controllers build the connections by programming the required flows at their subordinate TNs during step 3, using OpenFlow, NETconf or another standard protocol. At steps 4 and 5, confirmation of path establishment travels from the L0 controllers to the L1 controller, and from there to the Top controller. The procedure is not yet finished, because the ETNs

and the IATN need to be instructed as well. Therefore, at step 6 and using the respective REST interfaces, the Top controller informs the ETN Local Agents of the tunnel IDs they must associate to the remote ETN, and the IATN Local Agent of the tunnel translation it must perform. These messages are sent at step 6. At step 7, the Local Agents install the required flows into the underlying datapaths, typically using OpenFlow. Finally, at step 8, the Local Agents send confirmations to the Top controller. After this point, the tunnel is ready for use.

3.1.5 Experimental Evaluation of Control Plane Latency

In this section we evaluate the performance of the 5G-PICTURE hierarchical control plane looking at the time required to provision an end-to-end tunnel between two separate ETNs, involving multiple control plane areas. To carry out this evaluation we have conducted repeated experiments at the NITOS testbed [36], where we implemented the topology depicted in Figure 22, with Open vSwitch being used for the data plane.

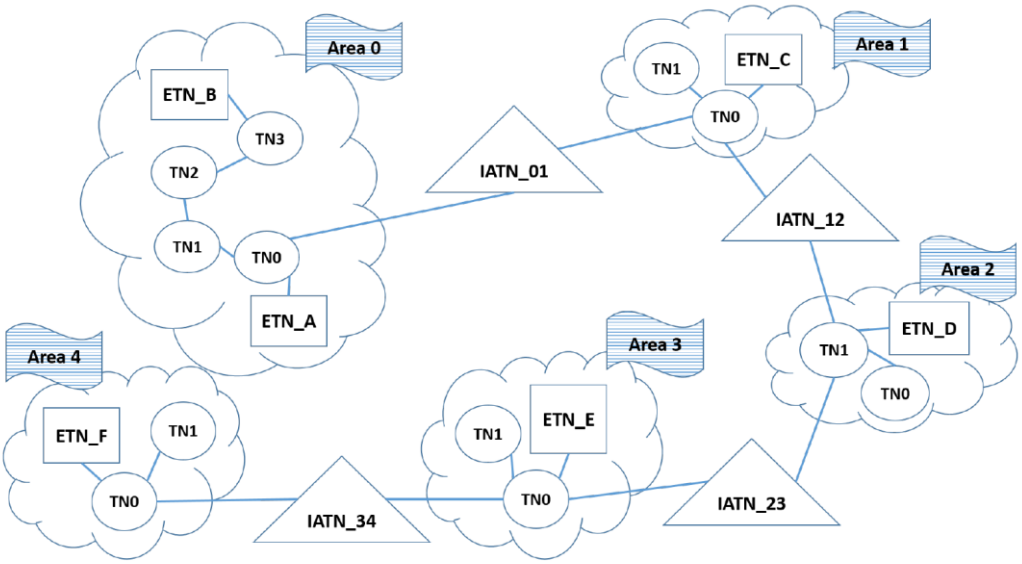


Figure 22: Experiment setup for latency evaluation.

The topology consists of five Ethernet based areas connected in a chain by IATNs. The 5G-PICTURE control plane is implemented in a set of VMs, including the five L0 controllers, the L1 controller, the Top controller, and Local Agents for all ETNs and IATNs in the topology. All VMs are hosted in NITOS physical nodes, and are connected to the same local testbed network. Each area was emulated by a Mininet instance running at the same VM as the respective L0 controller.

Using this setup we carry out a set of measurements to estimate the overall time required to establish an end-to-end tunnel. In this experiment, we examine the establishment of a bidirectional connection, which is very usual in practice, but our solution also supports unidirectional connections.

The overall time can be broken into two sequential components. The first component involves the Top controller's request to the L1 controller for the latter to provision an end-to-end bidirectional path between two given ETNs. The response of the L1 controller contains, among other information, the tunnel IDs to be used in each direction in every area used by the path. The second component involves the Top controller's instructions to the Local Agents of the two ETNs and of the IATNs traversed by the returned path. This component must strictly follow the first one, as the Top controller must instruct the Local Agents of the specific tunnel IDs to be used, and also it must be aware of which IATNs need to be updated. Therefore, we take separate measurements for the two components.

We are particularly interested in examining how our solution behaves for an increasing number of areas between the two ETNs, as we want to study the scalability of our solution. Our topology was designed to allow this. By keeping one of the two ETNs fixed, specifically ETN_A in the first area, and changing the other ETN to be ETN_B, ETN_C, ETN_D, ETN_E and ETN_F, we essentially examine the scenarios where the two ETNs are in the same area, or lie at the edges of an increasing number of areas, from 2 to 5, connected in chain.

For each of the five scenarios, we execute 25 different tests. The resulting cumulative distribution functions (CDFs) of the latencies for the two components of path establishment are reported respectively in Figure 23 and Figure 24. For both components, we observe a linear increase of the latency with the number of areas

between the ETNs. This makes sense. The L1 controller must partition the request and distribute it to a respective number of L0 controllers, which in turn need some time to program the TNs under their control. Increasing the number of areas also means the Top controller must instruct an increasing number of IATNs to install rules for tunnel ID translations. For each added area, the increase is approximately 50 ms for the first component and 70 ms for the second, i.e. a total of approximately 120 ms on the average.

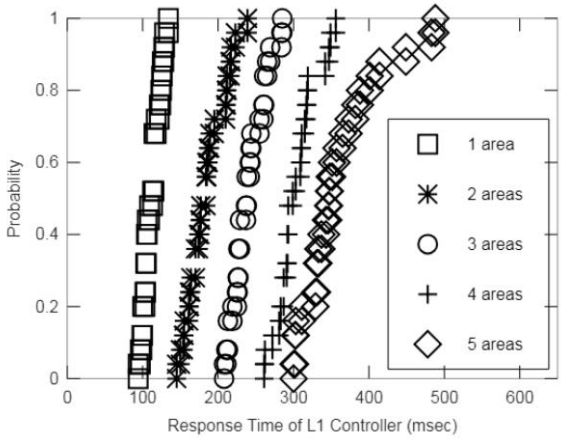


Figure 23: Scaling of L1 controller response latency CDFs with increasing number of areas in provisioned path.

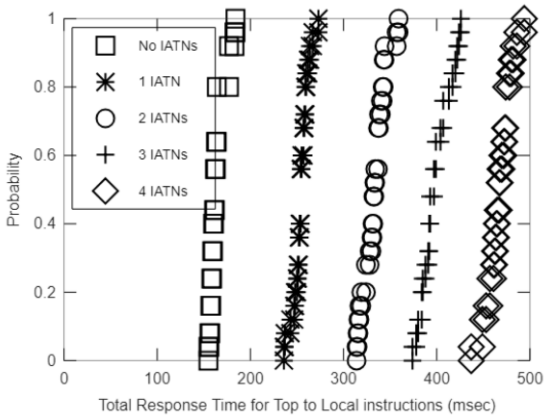


Figure 24: Scaling of total response time of Local Agents of ETNs and IATNs in provisioned path.

We have seen how the path establishment latency scales with the increasing number of areas. Yet another latency scaling result of interest is how the size of each area affects the latency. Specifically, how it is affected by the number of TNs within each area along the end-to-end path. We have examined this scenario using a fixed number of areas. Our results indicated that there was no significant fluctuation in the latency of a L0 controller for provisioning the path in its area with respect to the number of TNs involved. This is because the L0 controller has all the topology information allocated in its memory and can instantiate the paths using multiple threads.

Note that in the experimental evaluation above, all controllers were hosted at VMs connected to the same local network, therefore propagation delays were quite low. The TNs were also in the same local network as the respective L0 controllers. Therefore, for implementations where the architectural elements are distributed over a wide area, we should add the respective propagation delays to the above results, wherever they apply. These can be easily measured with the ping utility. As an example, we mention preliminary experimental results (without scaling) we obtained in [5], where areas and controllers were split between two sites, located at University of Thessaly (UTH) and i2CAT. Specifically, one area per site was deployed, the Top controller was running at UTH, the L1 and L0 controllers at i2CAT, the IATN Local Agent at UTH, and the ETN Agents at the site of the area where they were attached to. Through ping, an additional average round trip latency of 50 ms was measured between the two sites, which aggravated the measured control plane latency. The CDF of the overall path provisioning time measured in this experiment can be seen in Figure 25.

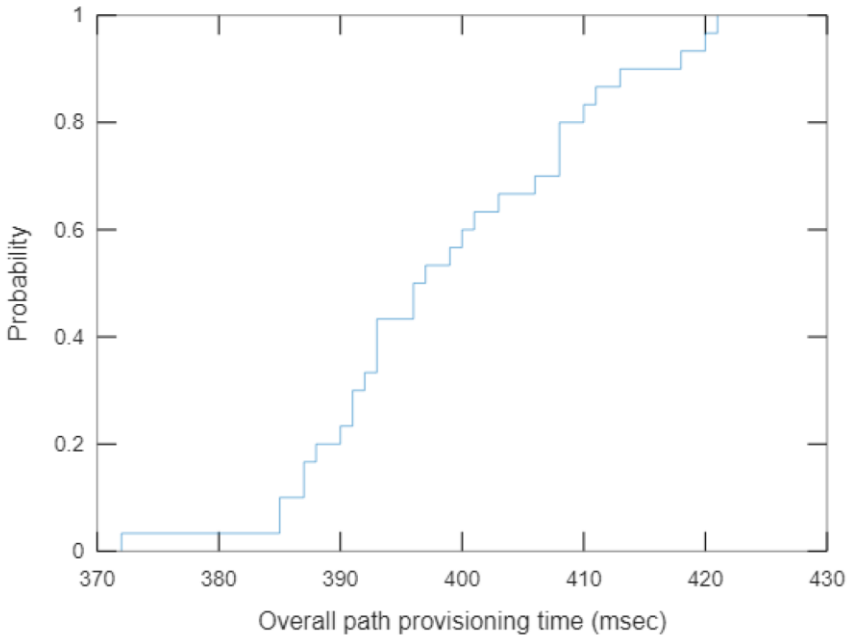


Figure 25: CDF of overall delay for path provisioning in two-area experiment with areas and controllers distributed at two remote sites.

Summarizing, the key observation to make is that the overall connection establishment time is below 1 second, which is negligible compared to service provisioning time in current IP networks, which often require manual intervention. We also point out that, for 5G transport networks, ETNs are expected to be separated by no more than a few areas, for instance due to data plane latency concerns. For two areas, expected latency is less than 500ms. Finally, we believe we can significantly reduce the path establishment latency through optimization of the software implementations of our controllers, by further employing multi-threading wherever possible, which is part of our ongoing work.

3.2 NFV MANO hierarchy

In this section, we describe how Pishahang realizes scalability with hierarchical management and orchestration (MANO). This is provided using a set of plugins namely service descriptor translator, descriptor splitter and MANO wrapper. These components have been implemented and integrated into Pishahang MANO framework and support orchestration of services across OSM and Pishahang MANO frameworks. Service requests received by the higher-level MANOs in the hierarchy can be split, translated, and sent to lower-level MANOs in the hierarchy using these plugins. In case of using Pishahang as the child MANO, each child MANO allows multiple other MANOs to be added as a child to it, hence achieving hierarchical orchestration (See Figure 26).

These plugins provide three main services that we describe in detail in the rest of this section:

- A wrapper (adaptor) that abstracts the APIs of different MANO frameworks.
- A splitter that breaks a network service descriptor (NSD) into create sub-NSDs, which different MANO frameworks in the hierarchy can use together to deploy the network service.
- A translator for VNFDs and NSDs between MANO frameworks.

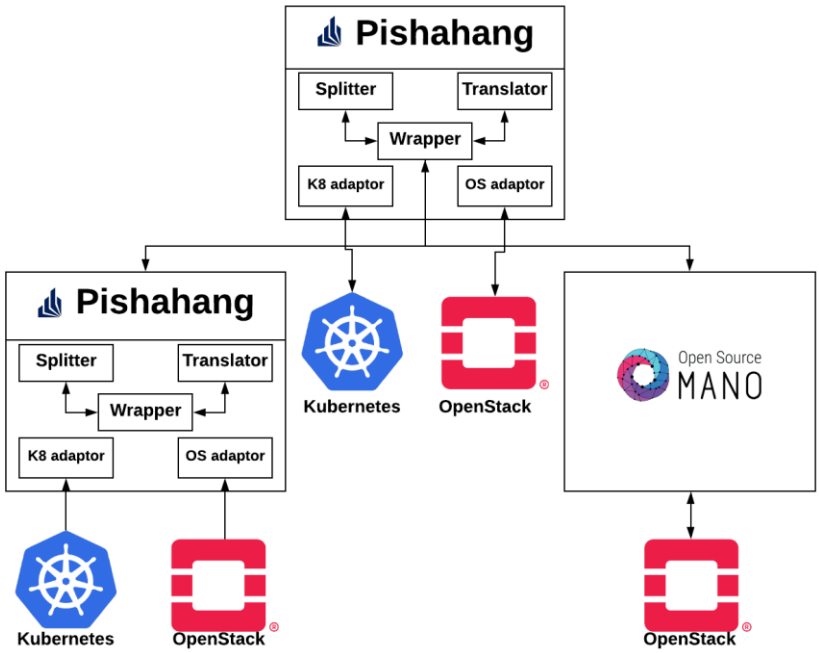


Figure 26: Architecture of Pishahang plugins.

3.2.1 MANO Wrappers

Python MANO Wrappers (PMW) is a uniform python wrapper library for various implementations of NFV MANO REST APIs. PMW is intended to ease the communication among different MANO frameworks by providing a unified, convenient and standards-oriented access to MANO API. To achieve this, PMW follows the conventions from the ETSI GS NFV-SOL 005 (SOL005) [46] RESTful protocols specification. This makes it easy for developers to use similar processes when communicating with a variety of MANO implementations. PMW is easy to use and well-documented. Code usage examples are available along with the detailed documentation[3].

PMW in Pishahang helps in intercommunication of different instances of MANO, creating opportunities for hierarchical scaling. Operations such as on-boarding of NSD and VNFD, instantiation and termination of NS can be performed with ease.

A standards-based approach is the fundamental design principle behind PMW's design. We have designed a common interface template in compliance with SOL005 [46] which contains the blueprint for all the methods mentioned in the standards. These methods are divided into the following sections as per SOL005:

- **auth:** Authorization API.
- **nsd:** NSD Management API.
- **nsfm:** NS Fault Management API.
- **nslcm:** Lifecycle Management API.
- **nspm:** NS Performance Management API.
- **vnfpkgm:** VNF Package Management API.

In Figure 27, different sections of PMW are visualized. Till now, the support for OSM and Pishahang frameworks have been developed within Pishahang. This is represented by the dotted lines to OSM and Pishahang modules. These modules are based on the common interface and implement the methods it has defined.

PMW can be installed using pip:

```
pip install python-mano-wrappers
```

Some examples of using these wrappers are shown in the rest of this section.

[3] https://python-mano-wrappers.readthedocs.io/en/adaptor/

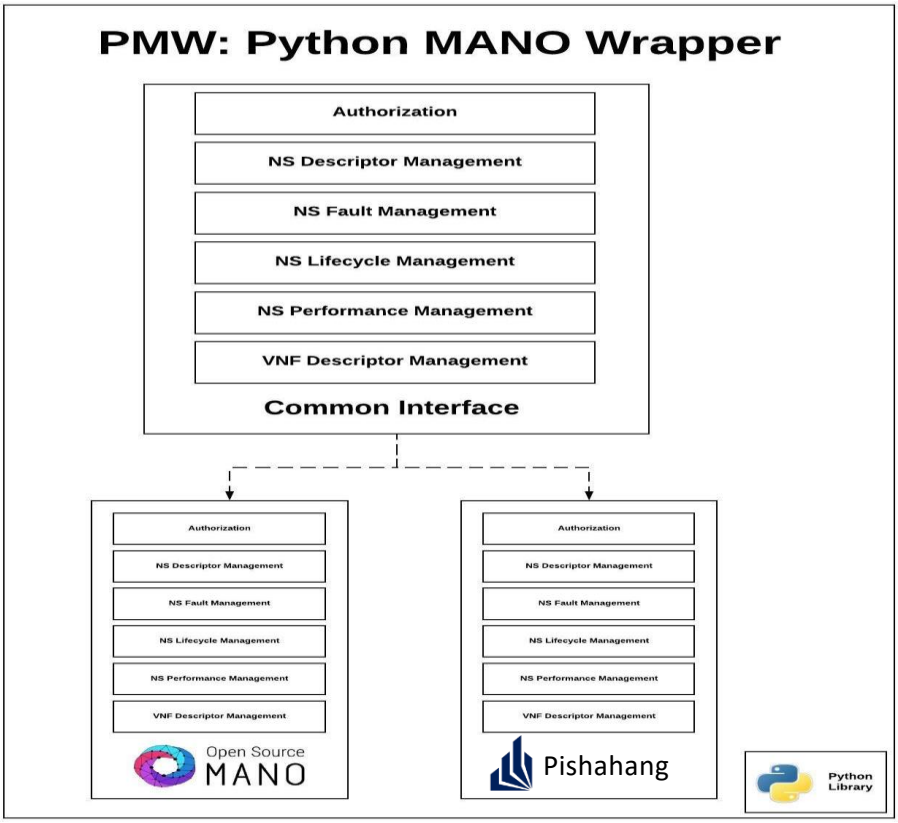


Figure 27: Architecture of Python MANO Wrappers (PMW).

Example: Fetching Auth token:

```
import wrappers

username = "user" password = "pass" mano = "osm"
# mano = "pishahang"
host = "localhost"
if mano == "osm":
_client = wrappers.OSMClient.Auth(host)
elif mano == "pishahang":
_client = wrappers.PISHAHANGClient.Auth(host)

response = _client.auth(username=username, password=pass-
word)
print(response)
```

Example: Instantiating a service in OSM:

```
from wrappers import OSMClient

USERNAME = "user" PASSWORD = "pass"
HOST_URL = "localhost"

osm_nsd = OSMClient.Nsd(HOST_URL)
osm_nslcm = OSMClient.Nslcm(HOST_URL)
osm_auth = OSMClient.Auth(HOST_URL)

_token = json.loads(osm_auth.auth(username=USERNAME, pass-
word=PASSWORD))
_token = json.loads(_token["data"])

_nsd_list = json.loads(osm_nsd.get_ns_descriptors(token=_to-
ken["id"]))
_nsd_list = json.loads(_nsd_list["data"])
_nsd = None
for _n in _nsd_list:
if "test_osm_cirros_2vnf_nsd" == _n['id']:
_nsd = _n['_id']

response = json.loads(osm_nslcm.post_ns_instances_nsin-
stanceid_instantiate(
                        token=_token["id"],
                        nsDescription=NSDESCRIPTION,
                nsName=NSNAME,
                        nsdId=_nsd,     vimAccountId=VIMAC-
                COUNTID))
response = json.loads(response["data"])
print(response)
```

3.2.2 NSD Splitter

Depending on the decision in Pishahang to delegate the deployment of a set of VNFs in a service using different MANOs, the splitter plugin will be triggered to generate different sub-NSDs. Before actual splitting of an NSD, the splitter first validates if the request it received is valid or not. Following things are validated by the splitter.

- The total number of NFs mentioned in all the set matches the number of NFs defined in the original NSD.
- There are no invalid VNFDs in the sets received.

After validation, the actual splitting starts. We have created classes for different sections of an NSD which encapsulate all the attributes and its values into a single unit. Once the objects are set, they are passed to different splitting functions based on their type. We have two different processing units for OSM and Pishahang. Following are some functions responsible for splitting the NSD.

- **Set General information:** This function copies all the general information from the main NSD to the sub-NSDs. Information includes vendor, author, version, description, etc.
- **Split Network Functions:** This function splits the Network functions from NSD to subsets according to the request parameter received.
- **Split Virtual Links:** When an NSD is split into different parts, its topology changes. Changes in topology result in changing of Virtual Links. For example, if A, B and C are three NFs and we are splitting them in such a way so that A and B remain in one NSD and C in separate NSD, the link between B and C should be broken down and B's output should be connected to the external end point which was connected to C's input earlier. This function splits these kinds of Virtual Links.
- **Split Forwarding Graph: O**nce the topology changes, the respective forwarding graph also changes. Split forwarding graph pulls out the set of connection points and newly created virtual links and sets them in the sub-NSDs.

Once the Splitting is done, the "create file" function is responsible for creating YAML files depending on the number of sub-NSDs created. These files are saved in the file system which can be downloaded or moved forward to the adopter for deployment purpose. Figure 28 represents the splitting architecture.

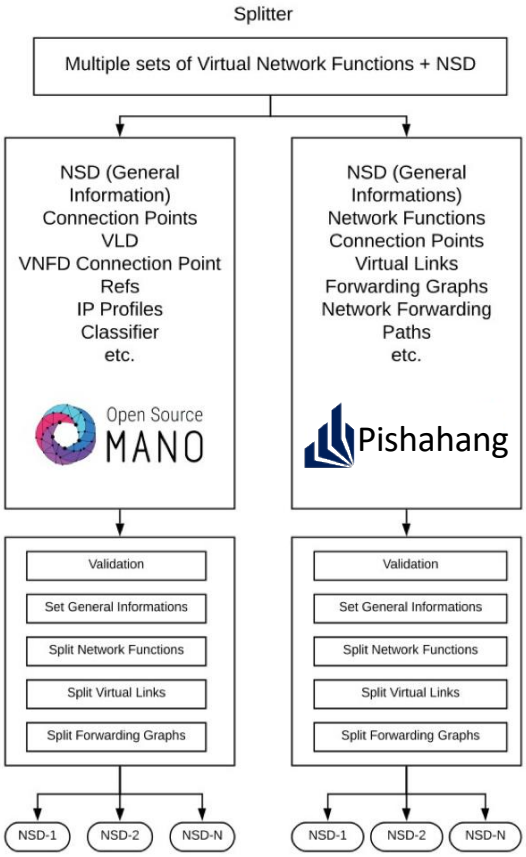


Figure 28: Splitter Architecture.

3.2.3 VNFD and NSD Translator

In a hierarchical architecture involving different MANOs, there is a need to convert of descriptors to the schemas of respective MANO. Service Descriptor Translator (SDT) serves the purpose of translating network descriptors, namely NSDs and VNFDs from the schema of Pishahang to that of OSM and vice versa.

In a scenario, where a MANO, e.g., Pishahang decides to deploy one of the network services in a MANO lower than itself in the hierarchy, e.g., OSM, the NSD and VNFD(s) need to be converted to the descriptor schema of OSM. In such an event, Pishahang calls the translator service and sends the descriptors to the SDT, where the translation of the descriptors takes place and the translated descriptors are sent to the wrapper utility for deployment in appropriate MANO. Figure 29 gives a high-level view of the translator.

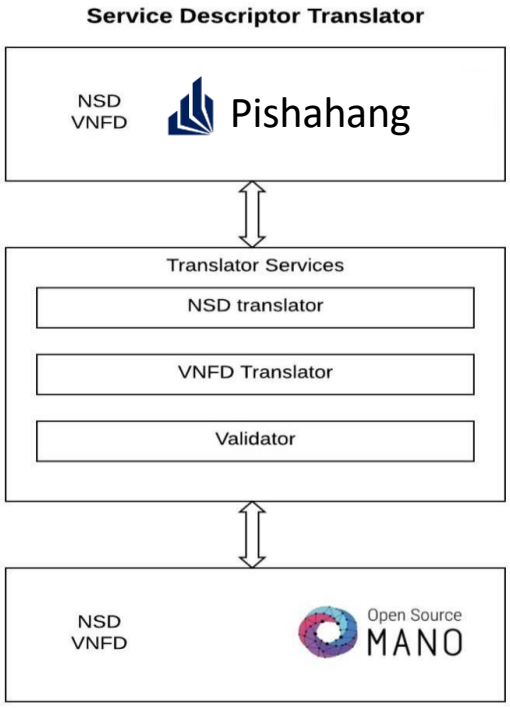


Figure 29: Architecture of the VNFD and NSD Translator

3.3 Dynamic eNodeB placement using FLEXRAN & JOX

In order to control the physical and/or virtualized base eNodeB, JoX supports a plugin to interact with FlexRAN (RAN Domain Controller). The FlexRAN agent that resides within the eNodeB communicates and interacts with a FlexRAN real-time controller entity. This agent exposes a set of RAN APIs, which can be used to monitor and control one or many local network functions in the eNodeB. The FlexVRAN [11] (an improved version of FLEXRAN to support virtualized RAN over heterogeneous deployments) provides an abstraction between the underlying physical infrastructures, logical base stations (BSs), and slice-specific virtual BSs, and thus easing the placement of slice-related base stations.

3.4 Dynamic controller placement using OpenDayLight

The SDN controller is one of the most crucial components of 5G OS, since it is responsible for the control plane of the 5G network. A network with a single controller comes with several weaknesses, such as reliability and scalability, since the control plane is centralized. To address this issue, we exploit the ODL clustering mechanism, in which multiple controllers operate, achieving high availability and fault tolerance. In short, a robust software-defined network requires a clustering mechanism, in order to maintain its functionality. Thus, we extend the hierarchical architecture presented in Section 3.1, by creating clusters of controllers that replace the single controller instances and increase in this way the failure resilience of the network. For example, instead of using a single L0 controller for an area, a cluster of L0 controllers becomes responsible for this area.

Running multiple controllers introduces new problems, due to the aim of operating as a single controller with a centralized logic. When different controllers are operating together, the overall state of the cluster is distributed among them. For example, one L0 controller belonging to a cluster, which is equipped with the COP adapter, communicates with the L1 controller for collecting the information for the aggregate topology and synchronizes its MD-SAL with the other controllers of the cluster, in order to distribute this information to them. The state of each individual controller should not differ from the rest of the cluster's controllers. Thus, a mechanism to synchronize the functionality of the cluster's controllers is essential. ODL implements various techniques and algorithms to address these problems, which are presented in the following sections of this research. However, the ODL clustering is bandwidth consuming, since it initiates extra control traffic.

The minimization of the bandwidth required for the control traffic is the highest priority objective in networks with low capacity control plane, such as low speed wireless networks with in-band control [31]. Moreover, the less the control traffic, the less the extra energy consumed for non-data transmissions, which is very important for limited-energy sensor networks. Finally, the minimization of the time delays between the controllers and

the switches is not essential, in case that the SDN control is proactive. For all these reasons, the minimum control traffic is very important e.g. in IoT networks exploited by massive machine-type communications (mMTC) in 5G. The application of SDN in these networks, especially regarding the controller placement models [39], is an open issue with a lot of ongoing research.

There are various studies that have been conducted to investigate the performance and scalability of SDN (OpenFlow) controllers. The research in [29] focuses on the comparison between three well-known controllers, the Open Network Operating System (ONOS), Floodlight and POX, based on the way they communicate with switches connected with them. However, this research does not cover the performance of those controllers as a part of a cluster. In [30], it addresses failover time, failback time and QoS issues in ODL clustering. Still, there is lack of research on how the ODL cluster scales while switches are being connected to the network and flow rules are being installed. In [31], a theoretical and experimental study have been conducted, regarding the bandwidth usage of the control traffic in a cluster of Kandoo controllers [32], which is now extended to the mostly used ODL controller.

3.4.1 OpenDayLight Clustering Overview

ODL's datastore is segmented in three shards: inventory, topology and toaster. Inventory shard contains the flow rules that are installed to the cluster, while topology shard holds data about the network topology. Topology shard's size is increasing when extra components are added in the network, e.g. switches. Toaster shard defines the Remote Procedure Calls (RPC) that can be executed in the corresponding shard, along with the actions that should be taken upon each call. Each shard is distributed across the cluster and is assigned to a node of the cluster. After the datastore distribution, in order to avoid data loss while a node failure is encountered, each shard from every node is replicated to all the nodes in the cluster. The shard and replica distribution is shown in Figure 30. Therefore, all shards and replicas in the cluster should always be synchronized, in order to achieve consistency. For this reason, ODL implements the Raft consensus algorithm [33], which achieves consensus in a distributed system.

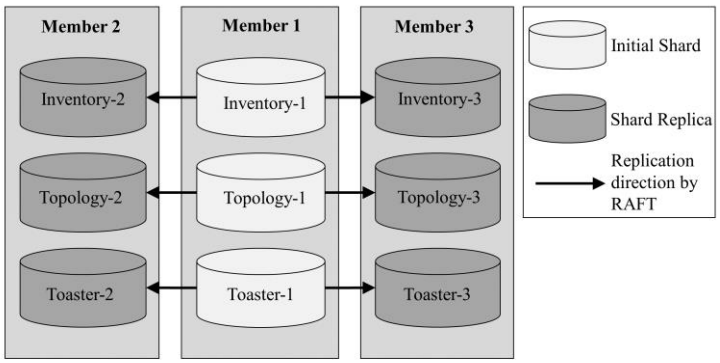


Figure 30: Distribution of shards from the leader shards to the other nodes. In this scenario, Member 1 controller hosts all the leader shards.

The Akka Cluster [34] provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck. This is achieved by using the Gossip protocol and an automatic failure detector. Every time a node starts with the intention to join the Akka cluster, it initiates a 4-way-handshake defined by the Gossip protocol. It communicates with the seeder nodes defined in the Akka configuration file by sending InitJoin messages. The seeder confirms the InitJoin request by replying with an InitJoinAck message. The node now sends a Join message and it has successfully joined the cluster when it receives a Welcome message back from the seeder. The Gossip protocol is not only responsible for replicating the current state of the cluster to all its nodes, but also it is crucial for the cluster's persistency. Heartbeat messages are sent among the cluster nodes periodically and expect a HeartbeatRsp back within a timeout to consider a node valid.

In each shard, the Raft algorithm [33] provides one of the three states, namely leader, candidate or follower, to each cluster node. Raft starts by initializing an election among the cluster nodes. The node which gets the most RequestVote messages becomes the shard leader. The leader is responsible for discovering inconsistencies among the cluster nodes related to the shard and replicating the proper data by sending AppendEntries messages. Moreover, empty AppendEntries messages are periodically sent by Raft, having the role of heartbeat messages. The leader must respond with a HeartbeatRsp message within a timeout, otherwise followers

should change their state to candidate and a new election round begins. In the case where a follower has new information in it, e.g. a switch connected to it, it sends a *CreateTransaction* message to the leader to notify him about the new data entering the cluster. After the transaction is done, the shard leader forwards the new data to the other node replicas of this shard to keep them updated. By default, ODL uses one node as the leader of all shards. The role of this leader is very crucial, since all shards forward changes in their data to it, and it is the leader's responsibility to replicate it to the rest of the followers. Visualization of a transaction from a follower to the leader and then the replication from the leader to all followers is shown in Figure 31. As it is shown in the experimentation results, the overall communication of the cluster seems to be centralized towards the leader node. This occurs because the shard leaders are all concentrated in a specific node, usually the one that starts first, as the research in [35] has demonstrated. This results in a node that contains all the leader shards, while the rest only have the follower shards.

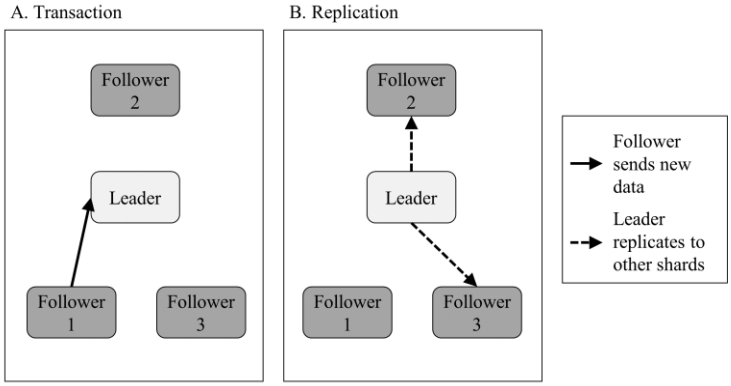


Figure 31: In A, the Follower 1 sends to the Leader its new data. Then in B, Leader replicates the new data to the rest of the followers. Notice there is not direct replication between the followers.

3.4.2 Network Model

In what follows there will be a description of the process of building a network model representing the total control traffic produced in an ODL cluster. A similar notation and approach is followed as in our previous work [31], however, the new model is adjusted to the traffic patterns of ODL clustering. An undirected connected graph G = (S, L) is considered, representing the control plane of an SDN network, where S represents the set of SDN switches and L represents the set of network links. Let $S = |S|$ and $L = |L|$ be the number of the switches and the number of the links respectively. Without loss of generality, it is assumed that the routing algorithm used for the control traffic is the shortest path routing, the path connecting the couple of switches $s_1, s_2 \in S$ is $p(s_1; s_2)$ and the number of links included in this path is $|p(s_1; s_2)|$. Finally, $C \subseteq S$ is the subset of switches where $C = |C|$ controllers are placed and grouped in a cluster, while $c_l \in C$ is the leader of this cluster. From now on, we may refer to $c \in C$ as a controller or the switch hosting it, interchangeably. Let $c_s \in C$ denote the controller that switch $s \in S$ is assigned to. Vector $c = (c_l \in C; (c_s \in C: s \in S))$ describes a controller placement and switch assignment. The first vector's value indicates the leader controller. Each other vector's coordinate maps to a switch $s \in S$ and the vector's value indicates the corresponding controller $c_s \in C$ of this switch.

The bandwidth usage for the Controller-to-Switch (Ctr-Sw) traffic from all network links is noted as

$$B^S \triangleq \sum_{s \in \mathcal{S}} \sum_{h \in p(s, c_s)} b^s = \sum_{s \in \mathcal{S}} w^s b^s \quad (1)$$

where b^s is the bandwidth required for the Ctr-Sw traffic exchanged between switch s and controller c_s, while $w^s = |p(s; c_s)|$ is the length of the path connecting s to c_s. Similarly, the bandwidth usage for the Control-to-Control (Ctr-Ctr) traffic from all network links is noted as

$$B^C \triangleq \sum_{c \in \mathcal{C} - \{c_l\}} \sum_{h \in p(c, c_l)} b^c = \sum_{c \in \mathcal{C} - \{c_l\}} w^c \left(b^c_{\text{inv}} + b^c_{\text{topo}} \right) \quad (2)$$

where b^c is the bandwidth required for the Ctr-Ctr traffic exchanged between controller c and leader c_l, that is the sum of the b^c_{inv} traffic produced by the inventory shard and the b^c topo traffic produced by the topology shard, while $w^c = |p(c; c_l)|$ is the length of the path connecting the two controllers c and c_l.

This work aims to formulate a mathematical problem that gives as solution the optimal controller placement C*, the optimal controller leader c_l* and the optimal switch assignment, given in the vector c* = (c_l*, (c_s* ∈ C*: s ∈ S)), which minimizes the total bandwidth required for the control traffic. This problem is expressed as:

$$\mathbf{c}^* \triangleq \arg\min_{\mathbf{c}} \Big(\sum_{s \in \mathcal{S}} w^s b^s + \sum_{c \in \mathcal{C} - \{c_l\}} w^c \big(b^c_{\text{inv}} + b^c_{\text{topo}} \big) \Big) \tag{3}$$

At this point, we make the following remarks that are validated by our experimentation with ODL controllers, which are organized in a cluster and exploit the OpenFlow protocol for their southbound interface and the features of the Layer 2 Switch project for the flow configuration. More details about our experimentation results will be provided later.

***REMARK 1**: The required bandwidth for the Ctr-Sw traffic exchanged between a switch and its controller is proportional to the number of flows existing in this switch.*

Let β^s denote the required bandwidth for each flow. If f^s is the number of flows existing in s ∈ S, then $b^s = f^s \beta^s$, ∀s ∈ S.

***REMARK 2**: The required bandwidth for the Ctr-Ctr traffic exchanged between a follower and the leader, due to the inventory shard, is proportional to the number of flows existing to the switches assigned to the follower. It is also proportional to the number of flows existing to the switches assigned to all other controllers (including also the leader).*

Let β^{int}_{inv} denote the required bandwidth for the connection between a controller and the leader, for each flow configured by this controller, and β^{ext}_{inv} denote the corresponding bandwidth, for each flow configured by all other controllers (including the leader). If $f^c = \Sigma_{s \in S\,:\,cs = c} f^s$ is the number of flows existing at the switches assigned to controller c∈C and $F = \Sigma_{s \in S} f^s$ is the number of all existing flows, then $b^c_{inv} = f^c \beta^{int}_{inv} + (F - f^c) \beta^{ext}_{inv}$, ∀c∈C.

***REMARK 3**: The required bandwidth for the Ctr-Ctr traffic exchanged between a follower and the leader, due to the topology shard, is proportional to the number of switches assigned to the follower. It is also proportional to the number of switches assigned to all other controllers (including the leader).*

Let β^{int}_{topo} denote the required bandwidth for the connection between a controller and the leader, for each switch assigned to this controller, and β^{ext}_{topo} denote the corresponding bandwidth, for each switch assigned to all other controllers (including the leader). If $y^c = \Sigma_{s \in S:cs=c} 1$ is the number of switches assigned to controller c∈C, then $b^c_{topo} = y^c \beta^{int}_{topo} + (S - y^c) \beta^{ext}_{topo}$, ∀c∈C.

Based on these Remarks, the problem presented in Equation (3) changes to:

$$\begin{aligned} \mathbf{c}^* = \arg\min_{\mathbf{c}} \Big(& \sum_{s \in \mathcal{S}} w^s f^s \beta^s + \\ & \sum_{c \in \mathcal{C}} w^c \big(f^c \beta^{\text{int}}_{\text{inv}} + (F - f^c) \beta^{\text{ext}}_{\text{inv}} \big) + \\ & \sum_{c \in \mathcal{C}} w^c \big(y^c \beta^{\text{int}}_{\text{topo}} + (S - y^c) \beta^{\text{ext}}_{\text{topo}} \big) \Big) \Rightarrow \end{aligned} \tag{4}$$

$$\begin{aligned} \mathbf{c}^* = \arg\min_{\mathbf{c}} \Big(& \sum_{s \in \mathcal{S}} w^s f^s \beta^s + \\ & \sum_{c \in \mathcal{C}} w^c \big(f^c (\beta^{\text{int}}_{\text{inv}} - \beta^{\text{ext}}_{\text{inv}}) + y^c (\beta^{\text{int}}_{\text{topo}} - \beta^{\text{ext}}_{\text{topo}}) \big) \Big) \end{aligned} \tag{5}$$

Problem 5 is equivalent to Problem 4, since the quantities $F \beta^{ext}_{inv}$ and $S \beta^{ext}_{topo}$ are independent constants for a given network. This problem can be solved using Integer Quadratic Programming (IQP), as we did in our previous work in [31].

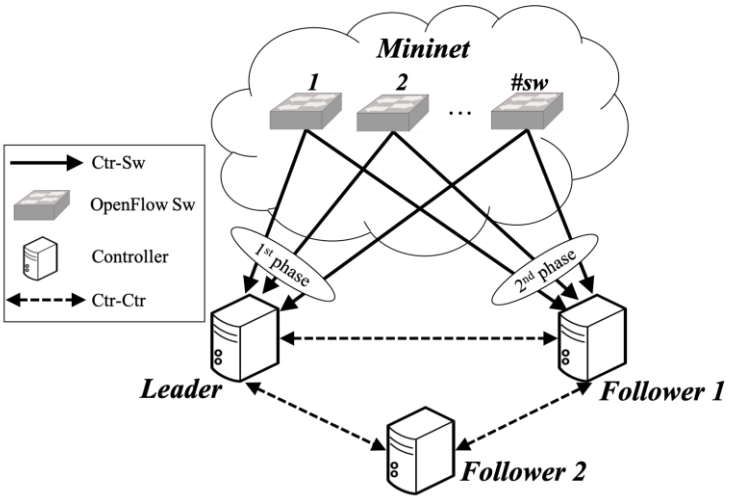


Figure 32: #sw switches are connected to either Leader (1st phase) or Follower 1 (2st phase), for measuring the Ctr-Ctr traffic exchanged for the topology shard.

3.4.3 Experimentation and Results

In order to test the performance of the communication among the cluster's nodes, we used the NITOS experimentation testbed [36]. A cluster of ODL controllers was used for the control plane, as well as the Mininet emulator [37] for the data plane. Each controller of the cluster was running on a separate physical node in the testbed. A total of three controllers was used to set up the ODL cluster, each of them using the OpenFlow protocol and the features of L2 Switch projects for the flow configuration. Finally, we used iftop [38], a free software command-line system monitor tool that produces a frequently updated list of network connections, to monitor the communication among the cluster's nodes.

As described in Section III, topology shard holds data regarding the network topology. When a topology shard in a cluster's member updates its data, the new data is replicated through the rest of the members. For this purpose, a Mininet topology was configured with various numbers of switches connected to each controller, and we measured the changes in the used bandwidth for the synchronization of all topology shard replicas. The topologies that were set up during these experiments are depicted in Figure 32.

Table 3 summarizes the results of these measurements. The leftmost column of the table represents the number of switches (#sw) connected to each controller, where *l* stands for Leader node and f_1 for Follower 1. Follower 2 (f_2) never has any switches. The columns labeled as x→y give the bandwidth usage in Mbps of the traffic sent from x to y, where x and y are either *l*, f_1 or f_2, while columns *l*→y give the total bandwidth usage for both *l*→y and y→*l*. Finally, columns avg_1 and avg_2 give the average increase per added switch of the bandwidth usage, for both directions, for the communications between *l*→f_1 and *l*→f_2 respectively. For example, when 2 and 3 switches are controlled by *l*, the values in column avg_1 are 0.23 = (0.64 – 0.18)/2 and 0.22 = (0.84 – 0.18)/3 respectively.

Table 3: Topology shard test results: Bandwidth usage for various number of switches at each controller.

#sw		bandwidth usage (Mbps)							
l	f_1	$l \rightarrow f_1$	$f_1 \rightarrow l$	$l \leftrightarrow f_1$	avg_1	$l \rightarrow f_2$	$f_2 \rightarrow l$	$l \leftrightarrow f_2$	avg_2
0	0	0.09	0.09	0.18	–	0.09	0.09	0.19	–
1	0	0.34	0.11	0.44	0.26	0.34	0.10	0.44	0.25
2	0	0.53	0.11	0.64	0.23	0.53	0.11	0.64	0.23
3	0	0.73	0.11	0.84	0.22	0.72	0.11	0.83	0.21
5	0	1.04	0.12	1.16	0.20	1.05	0.11	1.16	0.20
10	0	1.65	0.12	1.77	0.16	1.65	0.12	1.77	0.16
20	0	3.21	0.14	3.35	0.16	3.21	0.14	3.35	0.16
50	0	7.89	0.18	8.07	0.16	7.91	0.19	8.10	0.16
100	0	15.70	0.32	16.02	0.16	15.70	0.34	16.04	0.16
0	1	0.51	1.09	1.60	1.42	0.33	0.10	0.43	0.25
0	2	0.84	2.09	2.93	1.37	0.45	0.11	0.56	0.19
0	3	1.15	3.08	4.23	1.35	0.66	0.12	0.78	0.20
0	5	1.83	5.08	6.91	1.35	0.97	0.14	1.10	0.18
0	10	3.45	10.05	13.50	1.33	1.81	0.17	1.98	0.17
0	20	6.24	19.60	25.84	1.28	3.46	0.22	3.68	0.16
0	50	14.40	48.50	62.90	1.25	7.78	0.41	8.19	0.16
0	100	28.50	97.12	125.62	1.25	15.70	0.73	16.43	0.16

For the first phase of our experimentation, in order to explore how the data is replicated by the leader to the following shards, we increase the number of switches connected with the leader. The results of this experimentation phase are depicted in the upper half of Table 3, including all rows having non-zero values below *l*. It was found that bandwidth usage has been increasing in communication from the side of the leader to the follower nodes. No bandwidth increase was observed in the communication from the follower nodes to Leader. This occurs because the follower nodes do not hold any new data in their topology shard that needs to be replicated. The average bandwidth increase per added switch converges to 0.16 Mbps, either for the $l \rightarrow f_1$ or $l \rightarrow f_2$ communication, which is the marked value in the middle of both avg_1 and avg_2 columns.

For the second phase, the number of switches connected to Follower 1 was increased. The relative results are grouped in the bottom half of Table 3, including all rows having non-zero values below f_1. In this case, Follower 1 communicates with the Leader to inform it about the new topology data that have appeared in the network, and afterwards the Leader proceeds with the replication of the new data to the outdated topology shards. Indeed, as we can see in Table 3, there is a bandwidth increase in the communication channel from Follower 1 to Leader. Moreover, increase in bandwidth usage is present in the communication from Leader to Follower 1 and Follower 2 (due to data being replicated). In this case, the average bandwidth increase per added switch, as it is illustrated in the avg_1 and avg_2 columns, is 1.25 Mbps and 0.16 Mbps for the $l \rightarrow f_1$ and $l \rightarrow f_2$ communication channels respectively.

Based on these results, it is safe to assume that Remark 3 is confirmed with β^{int}_{topo} = 1.25 Mbps and β^{ext}_{topo} = 0.16 Mbps. Finally, there are no columns depicting the communication between the two followers, since it has been constantly stable and negligible in terms of bandwidth usage (almost 0.055 Mbps), compared to the other communications. This is due to the shard replication and how it is performed. When a follower shard has new data, it communicates with the leader shard and the second one is now responsible for the replication to the other followers. This means that there is no direct communication concerning the replication among the followers. The only interaction among them mainly regards periodic messages sent by the Akka, Gossip and Raft protocol, such as heartbeat messages.

In these series of experiments, the cluster behavior was tested when new information appears in the inventory shard. Inventory shard contains the flow rules that are installed to the cluster. Three switches are kept, each one connected to a single controller. Figure 33 depicts the topologies that were set up during these experiments. The flows are installed by using the ovs-ofctl command line tool, which is generally used for monitoring and administering OpenFlow switches.

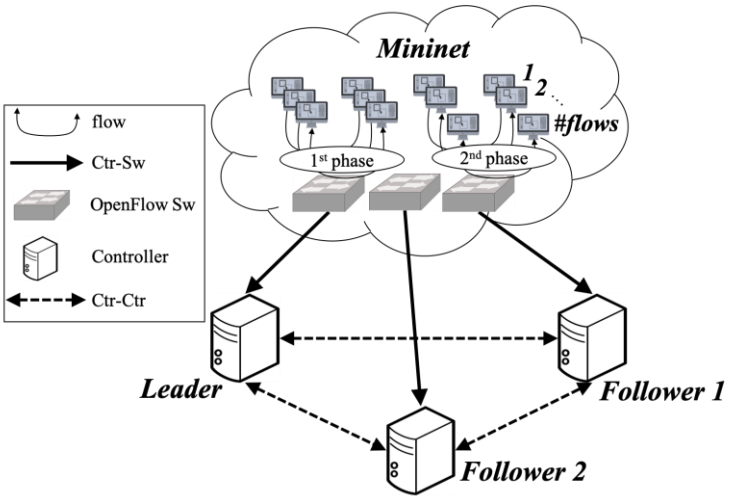


Figure 33: #flows flows are configured to the single switch connected to either Leader (phase 1) or Follower 1 (phase 2), for measuring the Ctr-Sw and Ctr-Ctr traffic exchanged for the inventory shard.

Table 4: Inventory shard test results: Bandwidth usage for various number of flows at each switch.

#flows		*bandwidth usage (Mbps)*							
l	f_1	$l \to f_1$	$f_1 \to l$	$l \leftrightarrow f_1$	avg_1	$l \to f_2$	$f_2 \to l$	$l \leftrightarrow f_2$	avg_2
0	0	1.00	1.10	2.10	–	1.00	1.10	2.10	–
30	0	1.04	1.10	2.14	0.001	1.02	1.10	2.12	0.001
60	0	1.08	1.10	2.18	0.001	1.08	1.10	2.18	0.001
120	0	1.19	1.10	2.29	0.002	1.19	1.10	2.29	0.002
240	0	1.45	1.10	2.55	0.002	1.45	1.10	2.55	0.002
0	30	1.09	1.25	2.34	0.003	1.09	1.10	2.19	0.003
0	60	1.21	1.35	2.56	0.004	1.09	1.10	2.19	0.002
0	120	1.46	1.60	3.06	0.004	1.20	1.10	2.30	0.002
0	240	1.85	2.03	3.88	0.004	1.39	1.10	2.49	0.002

At first, the number of flows installed in the switch connected to the leader node were increased. Similarly, to the topology shard case, when inserting new data in the leader inventory shard, an increase in the bandwidth usage is noticed from the leader's side to the follower nodes, while the communication from the followers' side remains stable. The bandwidth monitored during this experiment is demonstrated in Table 4, where the first column now is labelled as #flows and shows the number of flows at the switch of each controller. The avg_1 and avg_2 columns show the average bandwidth increase per added flow at the switch controlled by f_1 and f_2 respectively. Both avg_1 and avg_2 converge to 0.002 Mbps, as we can see at the bottom marked values of the upper half of this table.

For the second phase, flow rules were installed in the switch connected to Follower 1. Once again, the major role of the leader is obvious in the replication of the new data to the rest of the followers. Table 4 makes it evident that there is an increase in the bandwidth from Follower 1 to Leader, while Follower 1 is forwarding the new data to Leader. The leader node immediately replicates the new information to the rest of the followers, which results in the increase of the bandwidth usage noted in columns $l \to f_1$ and $l \to f_2$. The bottom marked values of the avg_1 and avg_2 columns converge to 0.004 Mbps and 0.002 Mbps respectively. As follows, Remark 2 is confirmed, resulting in β^{int}_{inv} = 0.004 Mbps and β^{ext}_{inv} = 0.002 Mbps.

In this series of experiments, we examined the Ctr-Sw traffic. The results of these experiments are given in Table 5, where columns sw→ctr and ctr→sw refer to the traffic sent from the switch to the controller and the opposite. In this set of experiments, the first phase of the previous experiments was repeated, as it is depicted in Figure 33, increasing the number of flows installed in the switch connected to Leader and monitoring the sw→ctr and ctr→sw traffic. Obviously, the average increase per added flow, as it is presented in the avg column, is 0.18 Kbps. Thus, Remark 1 is confirmed and β^s = 0.18 Kbps.

Table 5: Ctr-Sw test results: Bandwidth usage for various number of flows at the switch.

#flows	bandwidth usage (Kbps)			
	sw → ctr	ctr → sw	sw ↔ ctr	avg
0	18.10	1.88	19.98	–
10	19.80	1.92	21.72	0.17
20	21.53	1.95	23.48	0.18
50	26.73	2.04	28.77	0.18
100	36.00	2.19	38.19	0.18
200	53.90	2.50	56.40	0.18

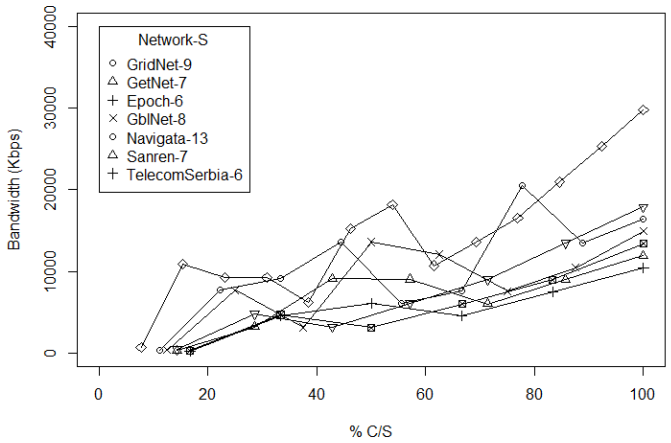


Figure 34: Relationship between the control traffic and C/S for various network topologies. The legends indicate the name and the size S of the corresponding network topology, as it is given by the Internet Topology Zoo collection.

At this point, it is worth mentioning that β^s is the traffic produced for each flow entry because of the statistics kept by the controller for this entry. This is the long-term average control traffic produced for each flow entry, which may be much lower than the traffic pick produced when the flow entry is configured to the switch. Moreover, the switches can be connected to multiple controllers, belonging to the cluster, increasing in this way their resilience to controller failures. The one controller suggested by the solution of our problem is the master, while the rest ones are called slaves. Our experimentation results show that the control traffic exchanged between the switch and the slave controllers is much lower and negligible, comparing to the other control traffic. The network model described in the previous subsection is used, in an attempt to simulate the Ctr-Ctr and Ctr-Sw traffic in various network topologies. The model is built based on the β^s, β^{int}_{inv}, β^{ext}_{inv}, β^{int}_{topo} and β^{ext}_{topo} measurements that were taken in the previous experiments. The IQP problem of Equation 5 is solved using R [40] and CPLEX [41] in multiple network topologies, provided by the Internet Topology Zoo Collection [42].

Reviewing the experimentation results, the minimum control traffic occurs when there is a single controller in the network. However, operating a network with one controller implies zero resiliency in controller failures. A minimum of three ODL controllers is required in order to create a fault tolerant controller cluster. Figure 34 shows the total control traffic for various *C / S*. As we increase the network's resiliency, the control traffic is not following a linear increase. Thus, a new minimum can be found for the control traffic in each network, while the number of controllers is bigger than three, and this is an interesting conclusion.

4 Support of Multi-tenancy

Multi-tenancy can be defined as hosting multiple service providers (or tenants) on fully or partially shared resources where each tenant is provided a sub-set of the available resources (referred to as a network slice) depending on their requirements. Each tenant is unaware of other tenants' slices and can operate independently of them. Only the provider of the network slice is aware of the resource allocation/division between tenants. The infrastructure provider may or may not be aware of end-users of the tenants.

4.1 Requirements for Multi-tenancy

Multi-tenancy support requires the following:

1. Service Catalogue – to allow tenants to choose the services they want which may be implemented as one or more network slices.
2. Service Management – to allow tenants to manage and configure the services (at the very least stop/start/suspend) as well as for billing and accounting information.
3. Service Monitoring – to allow tenants to monitor their services for failure**.**

Service Catalogue
Service Catalogue provides a listing of available services (service definitions). The Catalogue may also provide the ability to on-board custom services. The Catalogue should also provide additional information about the listed services, such as: service configuration, service SLA, cost, preconditions for use and dependencies.

Service Management
Tenants need to manage and configure any services they request. But precisely what aspects of the service can be configured (both before and after instantiation) are dependent on the service definition and the tenants' requirements. Assuming basic management of the service state is always required (start, stop and suspend the service), we can have three types of services based on configurability after instantiation:

- Services offering no control – these services do not provide any configuration (control) capabilities once instantiated. These may be implemented as one or more static slices. An example is a 'connectivity' service with a fix class-of-service.
- Services offering full control – these services provide one of more configuration interfaces once instantiated. An example is a network slice made up of Open vSwitch VNFs where each Open vSwitch instance provides a full OpenFlow and OVSDB interface for configuration. In this case the tenant could deploy their own SDN Controller that can interact using the available interfaces and control their network slice independently of other tenants and the provider.
- Services offering partial control – these services provide limited configuration capabilities. For example, a slice containing VNFs for Firewall and Access Control could provide interfaces to configure firewall rules as well as add devices/users to the ACL but no control whatsoever of the connectivity between VNFs. Another example is a network slice that is made up of shared and private resources. Here the tenant may not be able to control the shared parts so as not to break network slices owned by other tenants.

Service Monitoring
Service monitoring involves providing information about the service as well as the network slice used to implement it. Assuming basic service state monitoring as a minimum requirement, the monitoring information falls in one of the given categories:

- Fault monitoring – services can provide fault information at various levels of detail. This can allow use-cases related to SLA management and slice preservation.
- Performance monitoring – services can provide performance monitoring including compute/storage use, link utilisation and latency. This can enable use-cases related to SLA management, slice preservation and resource re-selling where real-time availability of the resource in a given network slice can be used to provision/optimise slices provisioned on top. This is the case where a tenant is providing multi-tenancy to its customers.
- Underlay information – in certain cases information about the network slice providers infrastructure may be visible. This information could contain geographical locations, raw device status and device performance data. An example of such a use-case is where a tenant has requested bulk-resources such as dedicated fibre links and optical nodes because it is providing multi-tenancy.

4.2 Domains and Multi-tenancy in 5G OS

Typical networks are made up of multiple technical, administrative and geographic domains. 5G OS Architecture Deliverable 5.1 [1] supports multi-tenancy as well as using different domains to create such services using the concept of Domain specific Orchestrators, MANO and Controllers and one or more Multi-domain Orchestrators to map services over the different domains. Furthermore, a provider of network slices may use other infrastructure providers for parts of their network. This results in multiple different domains that offer different class of north-bound APIs with different capabilities (as described in the previous section).

Figure 35 shows an example of three different types of services requested by three different tenants, each with different class of north-bound API.

Tenant 1 requests a slice that has a full control north bound API. The tenant also wants to run their own Orchestrator and Controller over this slice. This requires any external provider that the slice is mapped on, must provide similar service for that segment. For example, if the external provider is providing transit connectivity across their network then APIs for controlling the connectivity (e.g. bandwidth, admin state) must be provided.

Tenant 2 requests a slice that has no control interfaces but may involve one or more components (e.g. VNFs) that allow specific configuration interfaces.

Tenant 3 requests a slice that represents transit connectivity with QoS.

A thing to note here is that the Tenants could be using the provider in Figure 35 as a domain within their own slicing infrastructure.

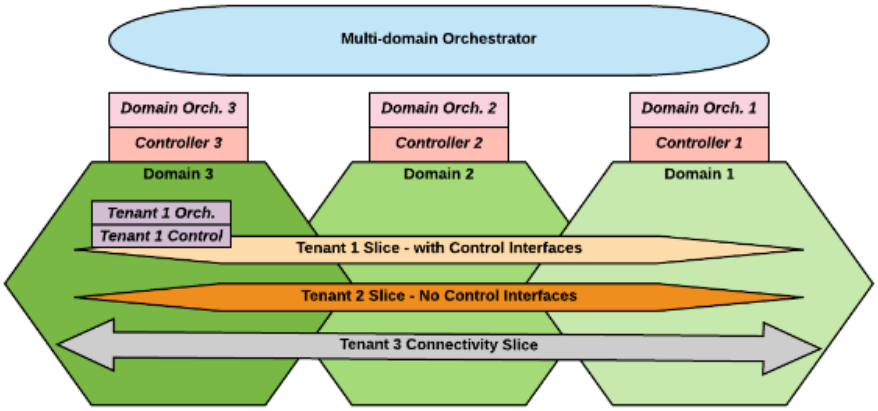


Figure 35: Domains with Multi-tenancy**.**

4.3 Controller-Orchestrator Hierarchies in Multi-Tenancy

A multi-tenancy environment is hierarchical in its nature. The infrastructure provider tenants have tenants of their own (even if they are end users). For example, an infrastructure provider giving connectivity using network slices through a metro network to various mobile operators. Here the mobile operators are tenants of the infrastructure provider. Mobile network users (so called 'end-users') are in turn tenants of the mobile operators. The tenants of the mobile operators will usually be invisible to the infrastructure provider yet they directly impact the provider.

While an infrastructure provider can directly react to its tenants changing behaviour, it is difficult to react to end-users changing their behaviour. There are also regional aspects to this for example consider the same scenario in a city scenario where the infrastructure provider has both fixed line broadband and mobile operators as its tenants. During the day mobile traffic may be significantly higher than fixed line broadband but the pattern might change during the evening or weekends. Both the services could hit peak demand in certain conditions or regions (e.g. mega-events, in and around stadiums).

To handle time and location variable loads, especially when dealing with multiple tenants, orchestration and control components must be easily deployed/decommissioned as well as be able to scale horizontally. This involves these components working within a hierarchical framework where scaling happens at each level.

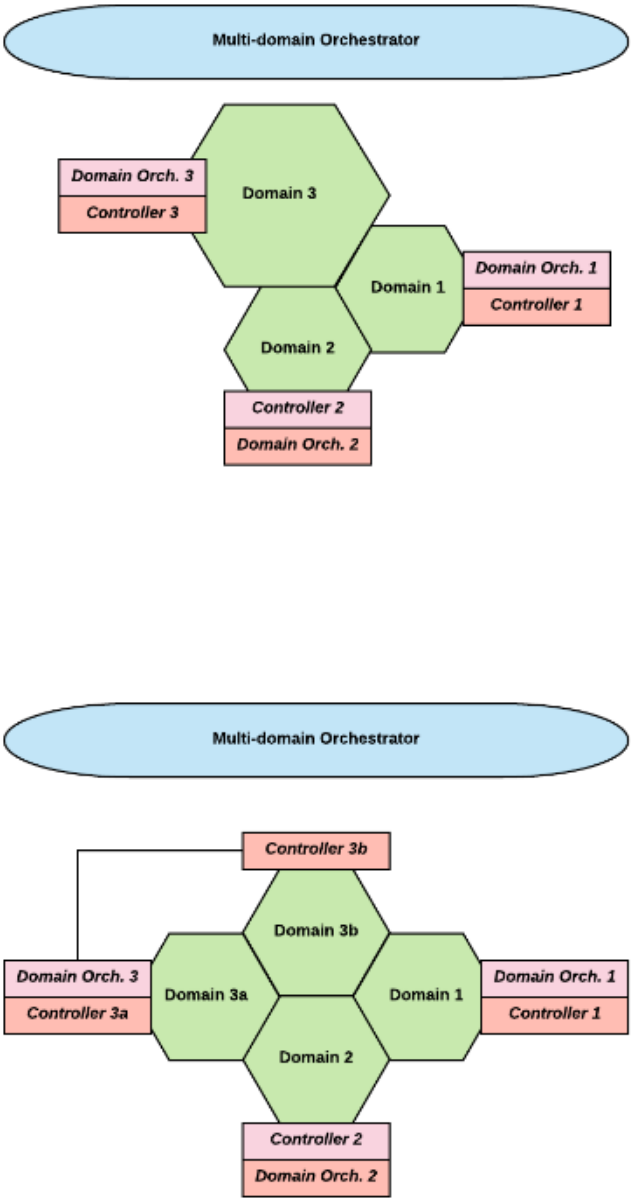


Figure 36: Dynamic splitting of Domain 3 due to changing demand.

The 5G OS Architecture allows for hierarchies of orchestrators and controllers. The natural division is in terms of domains which may be technological (e.g. optical, wireless, Layer 2), administrative or geographical. The Multi-domain Orchestrator (MDO) is responsible for provisioning services across different domains. Each domain has its own Domain Orchestrator that orchestrates over one or more NFV MANO and Domain Controllers.

Figure 36 (top) shows such a setting where a 5G OS Operators' network is divided into three domains (based on geographical regions). Each domain has its own Domain Orchestrator and Domain Controller (only one Controller shown for clarity – but there can be multiple Domain Controllers and NFV MANO instances).

In response to additional requests for resources from existing tenants and/or new service requests the provisioned orchestrator and controller for domain 3 cannot guarantee required performance. This may be measured in terms of number of controller performance (e.g. flows calculated and added per second, controller CPU/memory utilization, management traffic latency etc.) or end-user experience (e.g. new connections taking longer, loss of frame synchronization in video streams).

To improve controller performance the controller domain may be divided into two domains (3a and 3b), Figure 36 (bottom), where each domain contains lesser number of devices to be controlled. An additional controller is provisioned to handle devices in domain 3b. This would also help with CPU/memory utilization and management traffic latency (due to the smaller number of nodes being managed).

The trade-off here is that the Domain Orchestrator now has to orchestrate over two Controller Domains. Therefore, the network at the Domain Orchestrator level must be modified to reflect the new domains of responsibility. This requires standard interaction mechanisms between Orchestrators and Controllers that use the standard abstractions to communicate topology and other relevant network information. This would allow aggregation mechanisms to be built easily.

Another solution to variable demand is to have Orchestrators and Controllers that can operate in cluster mode where additional cluster members are deployed as the load increases. In this case the Control domain splits internally but to any external components (e.g. Domain Orchestrator) the network representation has not changed from what is show in Figure 36 (top). Therefore, no aggregations are required. While this may help with some problems like CPU/memory utilization, it is unlikely to help with issues like management network latency. In fact, it is likely to add to the congestion especially if peer-to-peer clustering is used [31].

4.4 Zeetta Slicing Engine Implementation in Multiple Domains

As the Zeetta Slicing Engine has been described in deliverable 5.1 [1] we provide a short introduction here to provide the context for this section. The Zeetta Slicing Engine consists of two main components: Orchestrator and Virtualisation Engine. The Orchestrator is responsible for mapping requested slices to underlay resources. The Virtualisation Engine consists of plugins (called Tactics) that provide control agents to implement the slice abstractions over northbound API provided by other controllers (real or virtualised) such as NetOS®. Figure 37 shows these components interacting for a single layer of slicing.

One example of a Tactic is control agent that provides virtualisation that represents an OpenFlow Table over a northbound OpenFlow API provided by a controller such as NetOS®. The Tactic provides its own Standardised API for Flow management (referred to as Use API for that slice). This Use API can be 'used' as a southbound by another Tactic that is able to write Flows for specific use-cases such as Access Control and Prioritisation. This layering of northbound Use APIs-Virtualisation Tactics-southbound Use APIs is shown in Figure 38, for a single domain. The Standardized API (in Figure 38) represents the Use API for the slice. The layering can be repeated with the right set of Tactics to provide multiple layers of slicing.

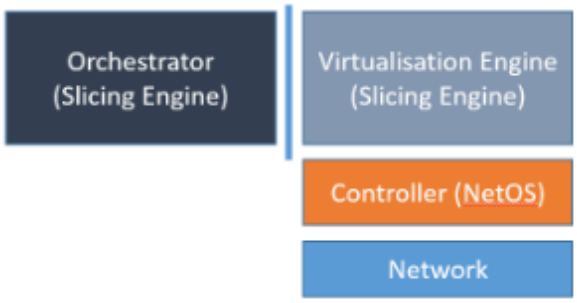


Figure 37: Overview of Zeetta Slicing Engine – Orchestration and Virtualisation Engine.

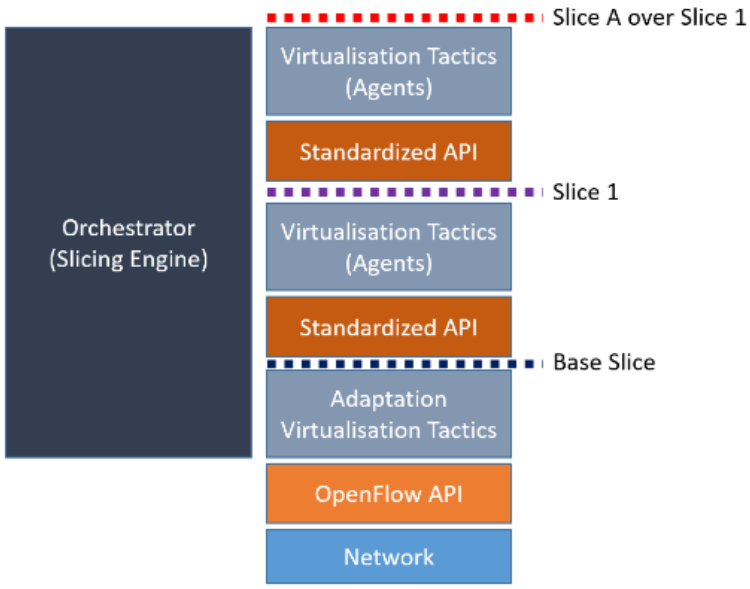


Figure 38: Single Domain Layering using Zeetta Slicing Engine.

Extending this concept to multiple domains requires an orchestrator component to map a slice across different domains. This can be seen in Figure 39 where two examples of Zeetta Slicing Engine integration are provided using a Transit Domain example. The overall network consists of three Domains owned and administered by 5G OS Operator. Domain 2 and 3 are connected via a Transit provider running Zeetta Slicing Engine.

The tenant 'Tenant 1' of the Service Provider has requested for an end-to-end slice with an associated Use API. Part of this slice is mapped to another infrastructure provider which provides transit connectivity. There is a Use API associated with this transit link as well. The Use APIs from the Service Provider (5G OS Operator) and Infrastructure Provider form the full Use API of the Tenant 1's slice.

In the top example, the MDO run by the 5G OS Operator integrates directly with the Transit domain running Zeetta Slicing Engine as a Domain Orchestrator. This has the advantage of allowing different services to be exposed by the transit domain (e.g. those including VNFs for access control). The drawback is that an industry wide common API would be required for such an integration otherwise it will be difficult to change Transit providers. Another disadvantage of using standardized APIs is that introduction of novel services via such APIs can be a slow process requiring formal changes in the specification.

In the bottom example, the MDO run by the 5G OS Operator only integrates directly with their own Domain Orchestrators. The Transit domain exposes a Controller interface that appears as another Controller within the Domain 3. This can be a direct Domain Controller integration with the Domain Orchestrator (for Domain 3) or it can be integrated within a Controller hierarchy using a common interface. The COP interface is one such option, but it only provides connectivity with class of service. While this is relevant for a transit provide it may not work for a provider with a wider catalogue of services. This is the concept of giving the requester the kind of network and control they request.

There are pros and cons to both the approaches:

1) Integration via Domain Orchestrator – Domain Controller interface: same as the top case – difficult to change providers but each Controller can provide novel services.
2) Integration via Common API: if they implement the COP interface then the providers can be switched easily (even dynamically), but it makes it difficult to provide novel services.

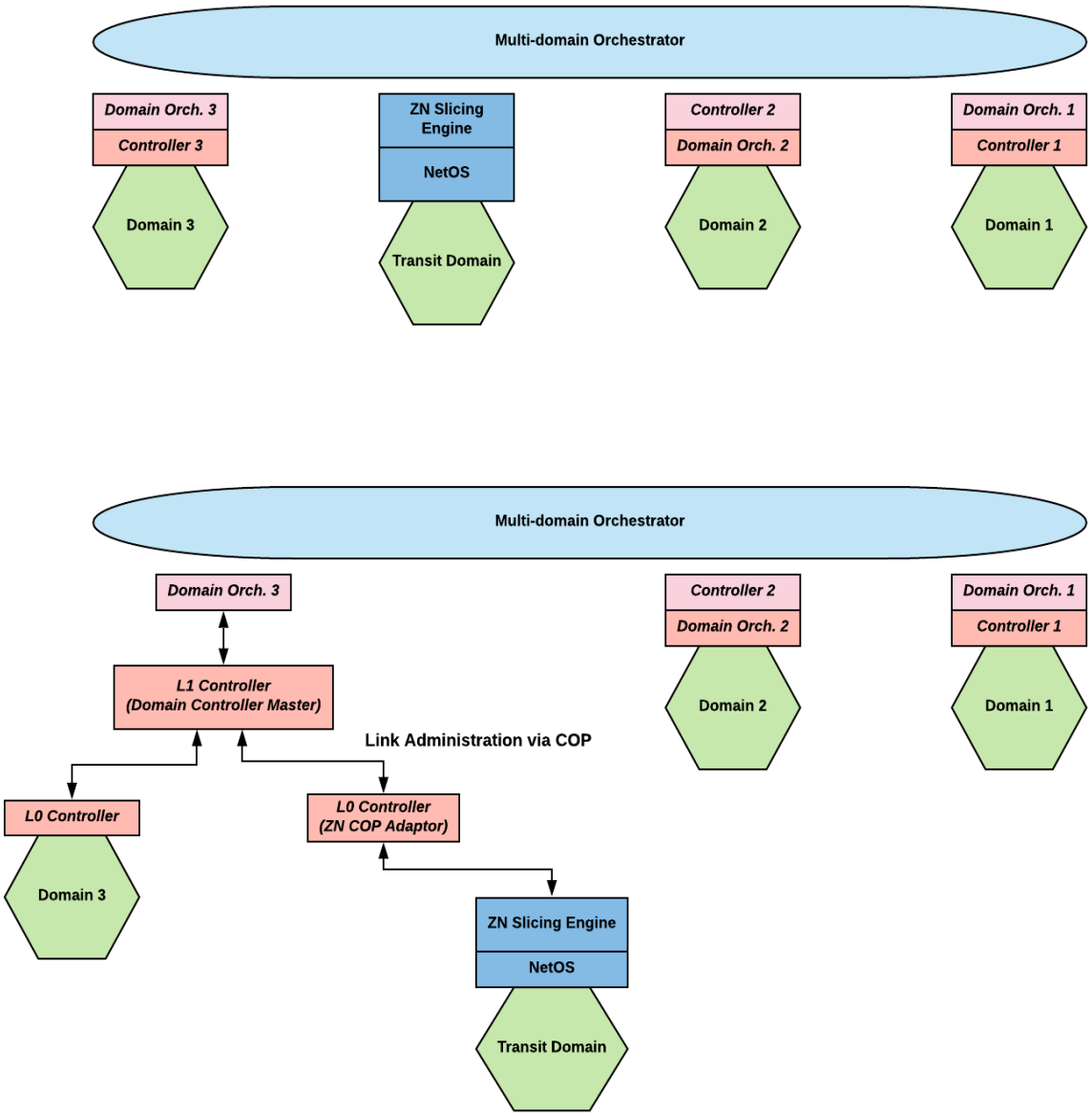


Figure 39: Multi-domain layering using Zeetta Slicing Engine – directly as Domain Orchestrator and via COP.

4.5 Using COP as Integration Interface

The Engine will be integrated within the 5G-PICTURE project as a transit connectivity provider. While it occupies the Domain Orchestrator component, it is desirable for transit connectivity to appear local to the 5G OS Operator. Within the 5G OS Architecture, it is possible if 'transit' is seen as another Domain to be controlled. This requires the Transit Provider to provide an interface that mimics a Domain Controller. The layers of virtual components used to implement connectivity is shown in Figure 40. The top-most layer provides the simple control NorthBound Interface (NBI) to disable/enable ports in the path used to provide transit connectivity. After that, the Priority Layer builds on the OpenFlow Layer below it to provide different class of service. The VLAN Layer provides Layer 2 isolation to the transit connection.

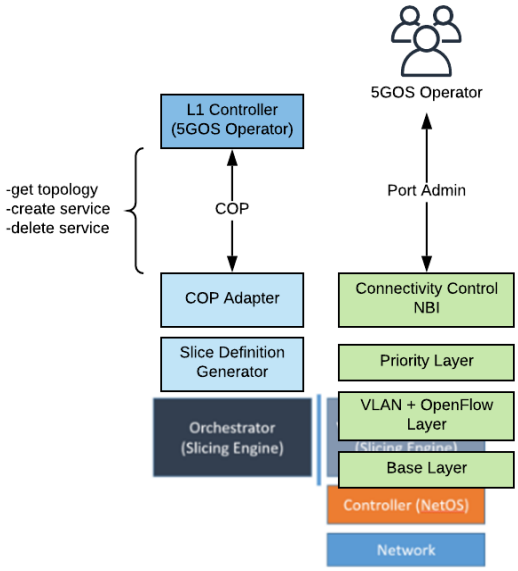


Figure 40: COP Adapter with Zeetta Slicing Engine.

The integration on the orchestration side can also been seen in the same Figure 40. The COP Adapter implemented by Zeetta allows integration with the 5G-XHaul controller hierarchy using the COP protocol. Here the COP Adapter provides the L0 Controller interface from the 5G-XHaul controller hierarchy. The COP Adapter in-turn drives the Orchestrator via the Slice Definition Generator. Based on the connectivity requests, slice definitions are generated on the fly, based on A and Z points in the connection request as well as the required Class of Service.

The interaction between the L1 Controller, the COP Adapter and the Orchestration Engine is shown in Figure 41.

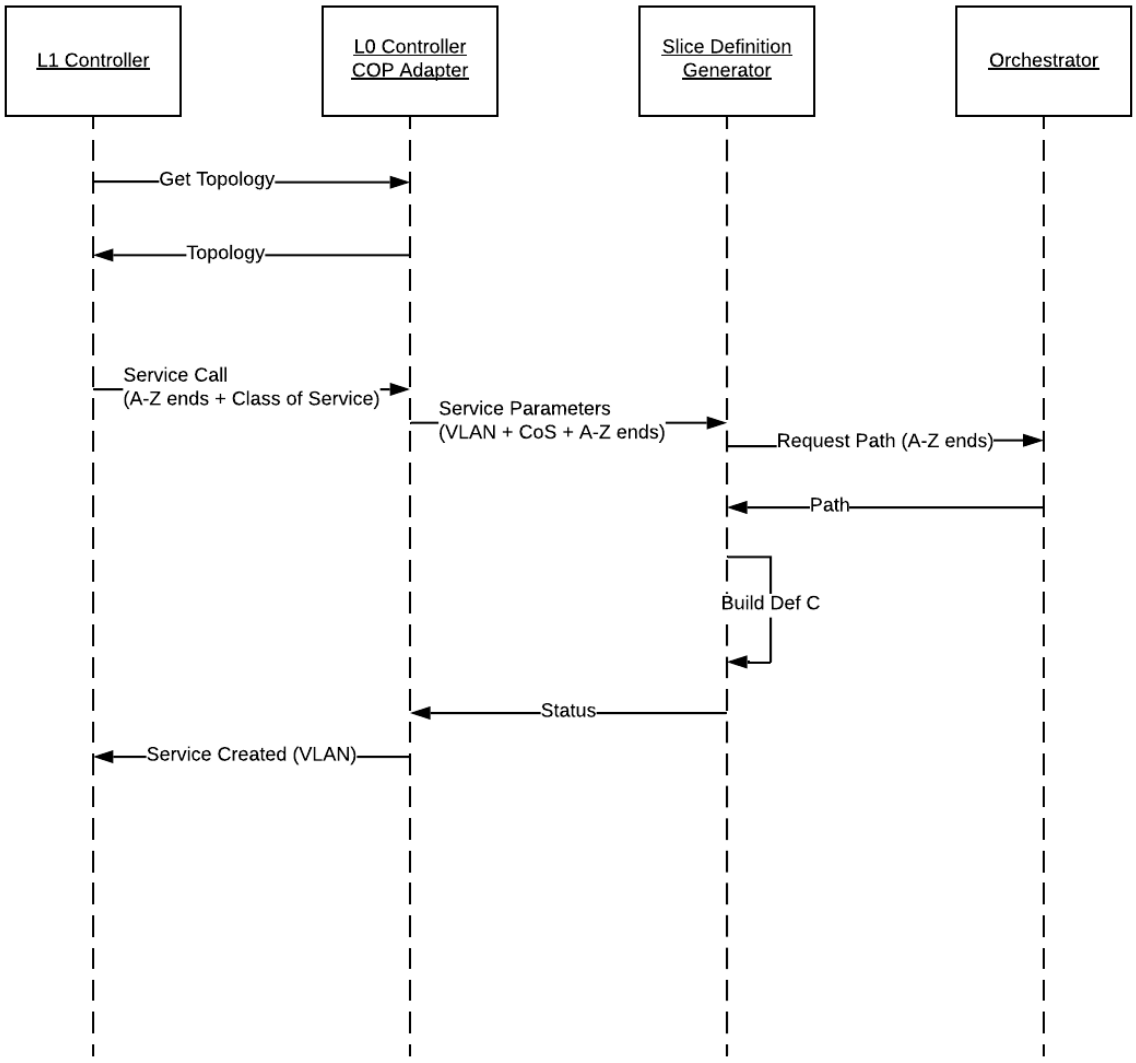


Figure 41: Interaction between L1 Controller and Zeetta Slicing Engine.

5 Related Work

5.1 Multi-domain Orchestration

The most representative implementation of the NFV MANO is the OSM, supported by ETSI. The OSM community works on being aligned with the evolution of the ETSI NFV specifications, while these specifications are updated according to the feedback coming from the use of OSM. Although OSM has its own implementation of the VIM, named as OpenVIM, OpenStack is more stable, well-known and used by many projects. OpenDayLight is similarly one of the state-of-the-art SDN controllers, which has already developed the required interfaces and plugins for synergy with OSM and OpenStack.

At the moment of writing this deliverable, the most updated and stable versions of OSM (Release FOUR), OpenStack (Queens) and OpenDayLight (Nitrogen-0.7.2) enable only the single-domain NS deployment. Although OSM can interact with multiple VIMs in parallel, it uses only one VIM each time a new NS is requested, which has to be declared on the NS request. Thus, OSM cannot use VNFs offered by different VIMs for multi-domain NS deployment, and of course WIMs do not exist in its ecosystem. Moreover, even in the single-domain NS deployment, OpenStack is able to control and manage only the physical machines (computing/storage hardware) hosting the VNFs of the deployed NS, and not the network hardware connecting these machines. OpenDayLight is only and optionally used for the configuration of the Open virtual Switches (OvS) existing in the host machines (or compute nodes).

SDN-Assist [21] is an extra plugin offered by OSM, which can be used for the single-domain NS deployment. It is in charge of connecting the VNFs located in different compute nodes, if and only if the compute nodes use physical Network Interface Cards (NICs) with Peripheral Component Interconnect express (PCIe) pass-through capabilities. These NICs exploit the Single-Root Input/Output Virtualization (SR-IOV) functionality to appear to be multiple separated physical devices, each of them mapped to and used by a single VNF. Although SR-IOV enables higher performance, it is not yet compatible with the majority of the utilized software (hypervisors) or hardware (NICs) components. The majority of the existing deployments depend on compute nodes with non-supporting SR-IOV NICs, while the VNFs use virtual NICs (vNICs) connected to virtual bridges (operated by OvS) residing on top of the physical NICs, as it is depicted in Figure 4. Moreover, SDN-Assist cannot not be used for multi-domain NS deployment, as the one depicted in Figure 6.

There are also some works proposing solutions for multi-domain NS deployment. The main disadvantage of these approaches is their lack of support for VNF interconnection inside the domains, since they focus only on the WIM implementation and not the SDN controller needed as part of each VIM [22], [23], [24]. Moreover, they use their own proprietary software for the orchestration (FROGv4, TNOVA and TeNOR respectively), instead of the widely-used and well-tested OSM. 5GEx [25] is one of the most well-known and successful projects for enabling cross-domain orchestration of services over multiple administrations or over multi-domain single administrations, however, it relies on a proprietary multi-domain orchestrator. To the best of our knowledge, we are the first extending OSM for multi-PoP orchestration, by designing and implementing the SDN-LAN and SDN-WAN components, which fit well in the whole controller hierarchy created by 5G-PICTURE.

5.2 Multi-version Orchestration across VM and Container domains

In both Cloud and NFV environments, there are tools and frameworks that carry out management and orchestration of VM- and Container-based services. The focus of NFV-based solutions such as OSM and SONATA have so far been on the orchestration of VM-based VNFs and they do not support CN-based VNFs in their frameworks. On the other hand, in the cloud computing environment, although both CN- and VM-based workloads are supported, they cannot meet NFV requirements. For example, OpenStack and Kubernetes do not meet multi-domain orchestration as required for NFV services, and they are usually used as Virtual Infrastructure Manager (VIM) in NFV. Terraform is a tool that sits on top of K8 and OpenStack and provides multi-cloud [43] functionalities for cloud service providers and allows them to deploy cloud services on multiple cloud infrastructures such as AWS and Google Cloud. But, like other cloud orchestrators, Terraform[4] also has been implemented for the cloud environment and does not fit into NFV environments as it cannot provide the requirements of NFV services such as service chaining. Cloudify is a cloud management system that has been extended to support NFV requirements. It utilizes ARIA [44] to orchestrate VNFs and provides a plugin that allows operators to deploy CNs on a K8 cluster. Cloudify supports hybrid VM/CN VNFs where CN-based VNFs

[4] https://www.terraform.io/

are nested on VMs. However, it does not support chaining of network services where CN and VM-based VNFs are deployed across CN and VM domains.

To support service function chaining across VM and CN domains, networking-related issues of CN should be tackled. In this regard, Intel proposed and implemented new services [45] for K8 that can solve a subset of the problem. Examples are the Multus Container Network Interface (CNI) plugin that allows CNs to be connected to more than one network interface (i.e., it is needed for VNFs to provide redundancy of the network and separate data plane from control plane). Google, also, funds a project called MetalLB[5] that provides a network load balancer implementation for bare-metal clusters. Its services (like address allocation) allow providing fixed IP addresses for CNs that can be used for chaining services.

Despite having all these tools and technologies originated from different technological environments (NFV, Cloud), there has not been any orchestration tool that can chain CN- and VM-based VNFs across different technological domains.

5.3 Controller Placement for Auto-adaptive Control Hierarchy

In the field of SDN, the most dominant protocol is OpenFlow. There are various studies that have been conducted to investigate the performance and scalability of SDN (or OpenFlow) controllers. The research in [29] focuses on the comparison between three well-known controllers, the Open Network Operating System (ONOS), Floodlight and POX, based on the way they communicate with switches connected with them. However, this research does not cover the performance of those controllers as a part of a cluster. In [30], it addresses failover time, failback time and QoS issues in ODL clustering. Still, there is lack of research on how the ODL cluster scales while switches are being connected to the network and flow rules are being installed. In [31], a theoretical and experimental study have been conducted, regarding the bandwidth usage of the control traffic in a cluster of Kandoo controllers [32], which is now extended to the mostly used ODL controller.

In this deliverable, we investigate how the communication between the controllers of the cluster is affected by connecting more switches to the network in terms of bandwidth usage. Next, a similar experiment is conducted by installing flows in the switches and observing the bandwidth usage inside the cluster. Then, we monitor the communication between the controllers and each switch connected to them. Last, we conclude with a heuristic real-time method for answering the controller placement problem, which is the most time-demanding problem when the network adapts to a failure and recomputes the required controllers for a cluster.

[5] https://metallb.universe.tf/

6 Conclusions

In this document, we have provided a detailed description of the design and realization of the cross-concept, auto-adaptive and hierarchical management, orchestration and control, in 5G OS. We have presented how 5G OS is able to handle multiple domains of different technologies, relying on a hierarchical architecture both for the NFV MANO components, as well as for the SDN controllers, which is auto-adaptive on dynamic changes of the 5G network.

More specifically, we have presented how state-of-the-art software tools are used, such as OpenDayLight, OSM and Pishahang (produced by SONATA and now extended by 5G-PICTURE) for enabling multi-domain management and orchestration, and how this functionality is supported by the SDN control. We showcased with our experimentation that our approach succeeds a satisfactory delay performance that is almost 20-30 ms in multi-domain scenarios. We also proved the ability of 5G OS to support multi-version deployment of NSs, as well as to slice the transport network and RAN for the support of multi-tenancy.

Moreover, we have presented the hierarchical model that we adopted in 5G-PICTURE for the organization of the NFV-MANO components and the SDN controllers. The hierarchical structure enables 5G network to be scalable and auto-adaptive, adding or removing components on demand. One of the results of our research is a heuristic algorithm for real-time resolving of the controller placement problem, which is to answer how many controllers are needed and where they should be placed, which depends on the diversity of the underlying resources.

Last but not least, we have showed how 5G OS support multi-tenancy. We have implemented our approach, showing the scalability of the adaptable, hierarchical design of 5G OS. Finally, we have provided a comprehensive state-of-the-art and related work, including standardization activities and academic studies in the fields of slicing, network control, and NFV MANO.

7 References

[1]. 5G-PICTURE Deliverable 5.1 “Relationships between Orchestrators, Controllers, slicing systems”.

[2]. 5G-PICTURE Deliverable 5.3 “Support for multi-version services”.

[3]. 5G-PICTURE Deliverable 4.2 “Complete design and initial evaluation of developed functions”

[4]. 5G-PICTURE Deliverable 4.3 “Integration of developed functions with 5G-PICTURE orchestrator”

[5]. 5G-XHaul Deliverable 3.3, [online available]: https://www.5g-xhaul-project.eu/download/5G-XHaul_D3_3.pdf

[6]. IETF, “Network Slicing Architecture”, [online available]: https://tools.ietf.org/id/draft-geng-netslices-architecture-01.html, 2017.

[7]. 3GPP TR 38.801 V2.0.0, “Study on New Radio Access Technology; Radio Access Architecture and Interfaces”, March 2017.

[8]. K. Katsalis, N. Nikaein and A. Huang, "JOX: An event-driven orchestrator for 5G network slicing", NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, Taipei, 2018, pp. 1-9.

[9]. “JOX northbound API documentation”, [online available]: http://mosaic-5g.io/apidocs/jox/, 2017.

[10]. OASIS, “TOSCA Simple Profile in YAML Version 1.2”, [online available]: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html, 2017.

[11]. Robert Schmidt, Chia-Yu Chang and Navid Nikaein, “FlexVRAN: A Flexible Controller for Virtualized RAN over Heterogeneous Deployments”

[12]. “Kubernetes” Production-Grade Container Orchestration, [online available]: https://kubernetes.io/.

[13]. P. Karamichailidis, K. Choumas and T. Korakis, “Enabling Multi-Domain Orchestration using Open Source MANO, OpenStack and OpenDaylight”, IEEE LANMAN 2019.

[14]. ETSI GS NFV-MAN 001 v1.1.1 (2014-12), “Network Functions Virtualisation (NFV); Management and Orchestration”.

[15]. ETSI GS NFV-EVE 005 v1.1.1 (2015-12), “Network Functions Virtualisation (NFV); Ecosystem; Report on SDN Usage in NFV Architectural Framework”.

[16]. ETSI OSM, “Open Source MANO”, https://osm.etsi.org.

[17]. O. Sefraoui, M. Aissaoui and M. Eleuldj, “OpenStack: toward an open-source solution for cloud computing”, International Journal of Computer Applications, vol. 55, no. 3, pp. 38-42, 2012.

[18]. “OpenStack”, [online available]: https://www.openstack.org.

[19]. J. Medved, R. Varga, A. Tkacik and K. Gray, “OpenDaylight: Towards a Model-Driven SDN Controller architecture”, Proc. of IEEE WoWMoM 2014, Sydney, NSW, Australia, June 2014.

[20]. “OpenDaylight”, [online available]: https://www.opendaylight.org.

[21]. “EPA and SDN assist”, [online available]: https://osm.etsi.org/wikipub/ index.php/EPA_and_SDN_assist.

[22]. R. Bonafiglia, G. Castellano, I. Cerrato and F. Risso, “End-to-end service orchestration across SDN and cloud computing domains”, IEEE NetSoft, 2017.

[23]. J. Caparinha, et al, “Deployment of Virtual Networks Functions over multiple WAN interconnected PoPs”, IEEE NFV-SDN, 2017.

[24]. J.F. Riera, et al, “TeNOR: Steps Towards an Orchestration Platform for Multi-PoP NFV Deployment”, IEEE NetSoft, 2016.

[25]. 5G Exchange (5GEx), [online available]: http://www.5gex.eu/.

[26]. M. Al-Fares, A. Loukissas and A. Vahdat, “A Scalable, Commodity Data Center Network Architecture”, ACM SIGCOMM, 2008

[27]. D. Giatsios, K. Choumas, P. Flegkas, T. Korakis, J.J.A. Cruelles and D.C. Mur, “Design and evaluation of a hierarchical SDN control plane for 5G transport networks”, IEEE ICC, 2019.

[28]. Straus project Deliverable 3.2 “Preliminary report on the building blocks for the network virtualization, openflow control and sdn orchestrator”, June 2015.

[29]. B.-Y. Yu, G. Yang and C. Yoo, “Comprehensive Prediction Models of Control Traffic for SDN Controllers”, Proc. of IEEE NetSoft 2018, 2018.

[30]. A. R. D. Nugraha, R. M. Negara and D. D. Sanjoyo,"High Availability Performance on OpenDayLight SDN Controller Platform (OSCP) Clustering and OpenDayLight with Heartbeat-Distributed Replicated Block Device (DRBD)", Journal Infotel, vol. 10, no. 3, p. 149, 2018.
[31]. K. Choumas, D. Giatsios, P. Flegkas and T. Korakis, "The SDN Control Plane Challenge for Minimum Control traffic: Distributed or Centralized?", Proc. of IEEE CCNC 2019, Las Vegas, USA, January 2019.
[32]. S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications", Proc. of HotSDN 2012, Helsinki, Finland, 2012.
[33]. D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm", ATC 14, vol. 22, no. 2, pp. 305–320, 2014.
[34]. "Akka Cluster", [online available]: https://doc.akka.io/docs/akka/2.5/ common/cluster.html
[35]. T. Kim, S.-G. Choi, J. Myung and C.-G. Lim, "Load balancing on distributed datastore in opendaylight SDN controller cluster", Proc. of IEEE NetSoft 2017, Bologna, Italy, July 2017.
[36]. "Network Implementation Testbed using Open-Source platforms (NITOS)", [online available]: https://nitlab.inf.uth.gr/NITlab/nitos
[37]. Mininet: http://mininet.org/
[38]. Iftop: https://en.wikipedia.org/wiki/Iftop
[39]. T. M. C. Nguyen, D. B. Hoang and Z. Chaczko, "Can SDN Technology Be Transported to Software-Defined WSN/IoT?", Proc. of iThings- GreenCom-CPSCom-SmartData, 2016.
[40]. "The R project for Statistical Computing", [online available]: http://www.r-project.org.
[41]. "IBM ILOG CPLEX for Faculty", [online available]: http://www- 01.ibm.com/software/commerce/optimization/cplex-optimizer.
[42]. "The Internet Topology Zoo", [online available]: http://www.topology-zoo.org/dataset.html.
[43]. D. Petcu, "Multi-Cloud: Expectations and Current Approaches", Proceedings of the international workshop on Multi-cloud applications and federated clouds, pp. 1-6, 2013.
[44]. "Apache ARIA TOSCA Orchestration Engine", [online available]: http://dpdk.org/.
[45]. "Multus [n. d.]. Multus Container Network Interface (CNI)", [online available]: https://github.com/intel/multus-cni.
[46]. "ETSI GS NFV-SOL 005". Network Functions Virtualisation (NFV) Release 2, https://www.etsi.org/deliver/etsi_gs/NFV-SOL/001_099/005/02.04.01_60/gs_NFV-SOL005v020401p.pdf
[47]. IMT-2020 Deliverables, ITU-T Focus Group, 2017.
[48]. C.-Y. Chang, N. Nikaein, R. Knopp, T. Spyropoulos, and S. Kumar, "FlexCRAN: A Flexible Functional Split Framework over Ethernet Fronthaul in Cloud-RAN," in Proceedings of IEEE International Conference on Communications (ICC), 2017, pp. 1–7

8 Acronyms

Acronym	Description
5G OS	5G Operating System
DC	Domain Controller
DO	Domain Orchestrator
IaaS	Infrastructure as a Service
IQP	Integer Quadratic Programming
LLDP	Link Layer Discovery Protocol
MANO	Management and Orchestration
MDO	Multi-Domain Orchestrator
NF	Network Function
NFaaS	NF as a Service
NFFG	Network Function Forwarding Graph
NFV	Network Function Virtualization
NFVO	Network Function Virtualization Orchestration
NS	Network Service
NSaaS	NS as a Service
NSD	NS Descriptor
OVSDB	Open vSwitch DataBase
PaaS	Platform as a Service
PNF	Physical Network Function
PNFD	PNF Descriptor
pPNF	Programmable PNF
PoP	Point of Presence
QoS	Quality of Service
RAN	Radio Access Network
SaaS	Slice as a Service
SBP	Slice Blueprint
SDN	Software-Defined Networking
SFC	Service Function Chaining
SLA	Service-Level Agreement
SLO	Service-Level Objective
SM	Service Management
TSON	Time-Shared Optical Network
VIM	Virtualized Infrastructure Manager
VNF	Virtual Network Function
VNFD	VNF Descriptor
VNFM	VNF Manager
WIM	Wide-Area Network Infrastructure Manager