POLITECNICO DI TORINO

Master of Science in Computer and Communication Networks Engineering

Master Thesis

# Introducing Flexible SDN/NFV Services on an Real Campus Network


**Supervisor**
prof. Fulvio Risso

**Candidate**
Farman Ullah

October 2017

To my family.

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

The cloud computing model is based on the idea of resource virtualization, many kinds of physical resources can be gathered together and the service provider will advertise an abstracted view of them to the external users. This paradigm indicates remarkable technical advantages. One of the most important breakthroughs consists in the opportunity of giving on-demand benefits; this advancement has been accomplished by a more accurate resources management which makes the resources more adaptable, maximizes their usage and reducing human technical interventions. From the specialized side, the main advantage is the overall cost reduction of some operations. At the same time, a user can take advantage of the possibility of requesting and obtaining service rapidly. Initially, the main virtualization techniques were mainly focused on computing and storage resources. Over the most recent couple of years, the accentuation has shifted to the virtualization of the physical network since new paradigms, for example, Software Defined Networking (SDN) and Network Function Virtualization (NFV) have emerged. The combination of these two paradigms permits the virtualization of an entire physical network.

SDN provides a centralized view of the distributed network for more efficient control, orchestration and automation of the network and their services. SDN approach is based on the partition of the control plane from the data plane. The control plane is managed by the SDN controller, a key part of the SDN which is able to act as the "brain" of the network, the data plane acts as the "muscles" of the network and keeps its working standards substantially unaltered. The SDN controller orchestrates traffic on the network and relays information to switches and routers through a southbound interface, while it communicates with applications through a northbound interface. One of the well-known protocols used by SDN Controllers is OpenFlow, it was defined to implement the communication between

the controller and the switches, which became internally simpler devices since they don't self-sufficiently take any decision about the traffic forwarding yet they work in accordance with the flow-rules received from the controller.

NFV is related to the deployment and management of network services, which are composed of network functions entirely implemented in software (they are then called Virtual Network Functions, or VNFs). Examples of network functions are network address translation, firewalling, intrusion detection and domain name service; being all software, these components can run using standard virtualization technologies. NFV brings flexibility for deployment, accelerating service provisioning and innovation, particularly within the service provider environments. NFV is a complementary approach to SDN. The end user has the possibility to install their preferred services on the operator's network. It is important to recall that network functions and services need an underlying network technology for talking to each other.

## 1.1 Goal of the thesis

This thesis aims to extend and validate the functionality of the FROG, which is SDN/NFV/Cloud orchestration architecture capable of deploying VNFs in multiple heterogeneous technological domains. FROG is an open source architecture developed by the NetGroup [1] research group at Polito, made of several components such as Open Stack cloud domains, SDN/Open Flow network domains, home gateways etc. All of these components are using different restful APIs to communicate within the FROG, which makes the whole system complex and. Then, it was decided that all the APIs of the FROG components must be clearly defined by following a coherent and consistent design, in order to provide a more agile and usable experience to the end users which is the purpose of this thesis as well.

In addition, there was a need to define a new communication standard which could be usable through a proper user interface so that it can easily be used by the end user. Our solution provides a new efficient and usable way to communicate to the various services of FROG from the easily used web-based GUI. Our solution empowers several performing actors (e.g., real end users, network providers and so on.). The graphical user interface of this solution permits to design complex virtualized service graphs. It can connect and communicate both with the Global orchestrator and with the particular domain orchestrator. Another important functionality

of this solution is that it also communicates with the Datastore via the GUI. The datastore is a helper module that contains NFFGs, VNF images and templates. Using this user can save the graphs, retrieve the graphs for future use and also deploy these graphs on the Global orchestrator using Restful standard APIs.

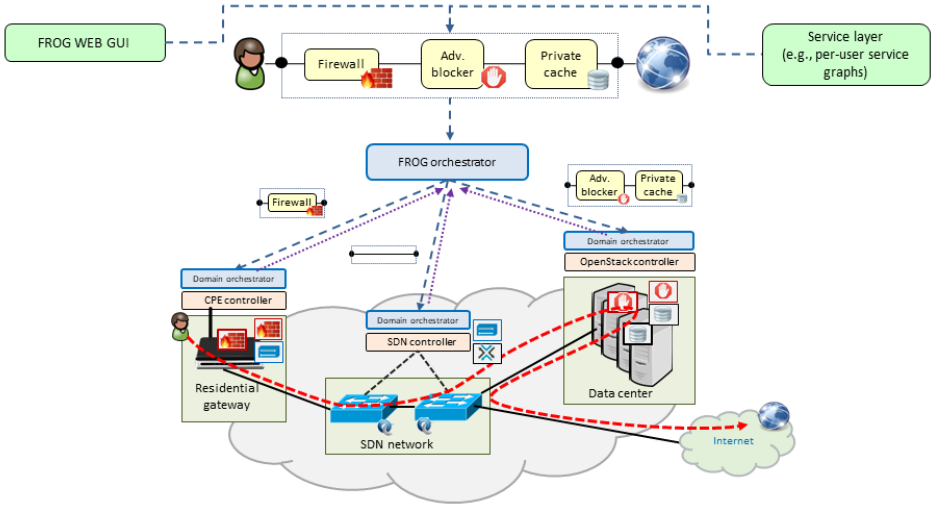


Figure 1.1. Deploying NFV services on the FROG.

The most important aspect of this thesis was to investigate the SDN and NFV technologies when end users use it in a single domain. In our solution, we introduce a flexible service such as exploiting these innovations on a multi-domain infrastructure. By accepting it, under the control of a single regulatory element, the so-called FROG orchestrator, it permits to create, modify and view complex network function forwarding graph by using standard APIs. Moreover, we permit the deployment of NFV services over various domains, even if different in terms of infrastructures. These domains can be specifically associated, connected to another domain or likewise connected to the Internet. This case is especially valuable to distribute services over domains portrayed by various capabilities, for instance, we can assume that we need to instantiate two NFV services (a DHCP and a Firewall) on two distinct domains, associated with a domain that does not support the deployment of NFVs, but rather it offers just network capabilities. For instance, it can be reasonable to deploy the Firewall on the user's domain; while the DHCP can be sent on an alternate domain where can serve numerous users.

Our work enables and helps that use cases searching for the ideal approach to deploy and associate virtual services, when conceivable. This can be exceptionally productive for network operators who need to distribute their network functions and

administrations over the network, achieving a high level of flexibility and scalability not given by the present work.

The whole network infrastructure is controlled by a service logic that performs the identification of the user that is connecting to the network itself. consequent to successful identification proof, the best possible set of network functions picked by the user is instantiated in one of the nodes (perhaps, even the home gateway) accessible on the provider network, and the physical infrastructure is configured to convey the user traffic to the above set of functions. This prompts an infrastructure which can deal with all the important and wanted services, conceivably sample for a user. This turns out to be more essential the fast Internet of things approach is developing, as a matter of fact, a lot of new connected device can't stand to have a same level of security as a traditional computer or cell phones, because of their restricted resources and this solution can give similar functionalities to all devices.

Another big challenge was an integration of changes. As all the components of FROG are interrelated, they are somehow dependent on each other, so making changes to one component used to disturb many other components and hence we needed to track all the necessary changes in the other components and implement them as well. We also, analyze the advantages and disadvantages of the orchestration of SDN/NFV services over various domains of heterogeneous technologies. Obviously, the GUI and the orchestrators need to know some details about the underlying infrastructures to make more profitable decisions. These specific domains can be linked directly or can be reachable each other through another domain or even through the Internet. The connection will be conceivable by setting up a direct tunnel over the Internet using GRE or VLAN.

This thesis is structured as follows:–

- **Chapter 2:** Explains in an exhaustive manner the problems that this thesis is proposed to solve, describing the state of the art of solutions and architectures used in the context of network service orchestration.
- **Chapter 3:** Proposes an overview of the architectures on which this thesis is based and of the technologies used.
- **Chapter 4:** Describes the architecture of the FROG, the reference software framework for this project and in which the thesis work lies.
- **Chapter 5:** Describes the architecture and we will expose the details of our solution, used in this thesis.

- **Chapter 6:** Describes the Implementation of unaligned APIs of the FROG components and extended support for multi-domain
- **Chapter 7:** Explains how to deploying services on a real campus network, using Cisco ISR router.
- **Chapter 8:** provides an overview of performance evaluations of some use cases detailed in previous chapters.
- **Chapter 9:** Exposes the conclusions and provides some projects that will follow this thesis work.

# Chapter 2

# Introducing Flexible SDN/NFV Services on Multi-Domain Networks using Real User

This chapter describes the orchestration scheme prevailing in the SDN/NFV and the problems it faces in cases where it is intended to support general defects. We introduce the issue that this thesis is proposed to solve, depicting the state of the art architectures and solutions used as a part of network service orchestration. New services, for example, cloud computing and the expanding host's mobility, alongside the growing demand for content delivery, security and quality of service convey too much more dynamic demands on the communication infrastructure. Customer networks need, within seconds, to be equipped with the associated network functions such as switching, routing, firewalling and load balancing. This network functions should be able to be dynamically moved from one computing device to another one when needed and the necessary configuration changes must be made automatically. Consequently, the large cloud providers like Amazon and Google decided to push towards a Software Defined Networking approach. This choice makes network definitely agiler than before.

## 2.1 Current networks conception problems

The previous 30 years have been set apart by the networking principle of distributed intelligence. Switches and routers basically used to choose independently where they would have forwarded packets and what information they would have exchanged with neighboring devices. None of these devices used to know the whole network

topology however just a little part of the end-to-end path. This approach ended up being extremely steady and scalable, yet in increasingly sluggish and not really flexible. For example, the need of even little changes in network configuration requires the reconfiguration of all devices, which is very time-consuming. Current networks are implemented with hard, inflexible and super-fast hardware, and then it is very difficult for a service provider to offer to final users and companies a flexible and innovative service with extreme effortlessness. Therefore, a hardware-centric networking approach prompts slower innovation. Truth be told, we are now still using network technologies which were introduced many decades ago. Baking the software into silicon circuits dramatically lengthens production cycles and reduces the number of features which can be incorporated into one system; in the long run, once baked in, the hardware can't be effectively modified. Firmware only softens this compromise, yet it doesn't really change the underlying choice. Besides, all the devices needed on nowadays networks come from different vendors, with very different proprietary solutions and firmware; this gets the re-configuration issue even worse. The need of even little changes in network configuration requires the reconfiguration of all devices, which is very time-consuming.

## 2.2 The ongoing evolution

The software is infinitely flexible; however, it is still much slower than hardware. However, expanding hardware performances and multi-core processing is slowly narrowing the gap in performance, while new software development practices, virtualization and open standards have made software much more modular, flexible and simple to develop than ever. These principles allowed the recent fast progress in Information Technology. Software and computing already experienced deep changes because of these innovations, in any case, nowadays, it is computer networking which is experiencing the greatest change since 1970, when they were born. From now on, networks should not be configured at the device level anymore, but as a whole system. This makes possible to build networks which are more agile, flexible, robust and secure than yesterday networks.

The so-called **Software Defined Networking** approach intends to grant this extraordinary flexibility changing the basics of networking as we know it, by introducing a centralized controller which manages lots of simple general-purpose devices that can be re-configured in a glance just changing a bunch of networking rules via software. This is because of the abstraction level reached by this kind of solution, where networks become more and more independent from underlying

hardware devices and easily programmable. That also introduces a degree of operational flexibility never seen before in this environment; now it becomes possible to do an extremely precise traffic management, making rules based on transport or even application layer details and, also, load balancing and asymmetrical paths are now truly simple to realize, contrasted with the past

On the other hand, **Network Functions Virtualization** [2] approach wants to harness innovations brought from software-defined networks and merge them with the benefits of virtualization and cloud computing. This consists in converting all typical functions of computer networks (switching, routing, so forth...) and collateral ones (security, contents delivery, etc...) from dedicated hardware platform to software virtual functions which can be executed on almost any computing device. The advantages introduced from this approach are countless, starting from flexibility and cost reduction arriving at better integration between IT and TLC worlds. It seems clear that we are in front of a paradigm change which will probably bring to a robust evolution of telecommunication networks and services provided by network operators, yet the continuous transformation is not only a technical innovation yet includes also a cultural revolution.

## 2.3 Software Defined Networking

SDN is rapidly becoming a fundamental part in telecommunication networks evolution, starting a deep modification in how these networks are organized and managed. It allows overcoming all the problems of traditional networks thanks to the centralization of the controller logic and an open and standardized interface for network nodes configuration. The first characteristic allows abandoning the old distributed network state concept transferring the control to an external software entity, which bases its decisions on an abstract, the global and consistent vision of the whole network. The second one allows collecting information regarding the state of the network and configuring all the devices in a unique standard way, without dealing with different hardware, firmware or vendor solutions.

Summarizing, the main aspect of SDN approach is to separate the control of the network behaviour from the physical infrastructure itself. This allows generalizing decisional processes about packet traffic and their actuation. Moreover, it simplifies the creation of virtualized partitions of the physical network (named slices) which will then be assigned to a different centralized controller; thus dividing the unique physical network into many separated logical networks, each one with its network

protocols, forwarding rules and addressing plans. From the functional point of view, the SDN approach can be divided into three sub-layers. Applications determine high-level politics about how packets should flow into the abstract model of the network (or a sub-partition) exported from the control plane. In this way, applications do not have to worry about all the complexity of the physical network and the underlying system. Applications interact with the network controller through an interface, called northbound API.

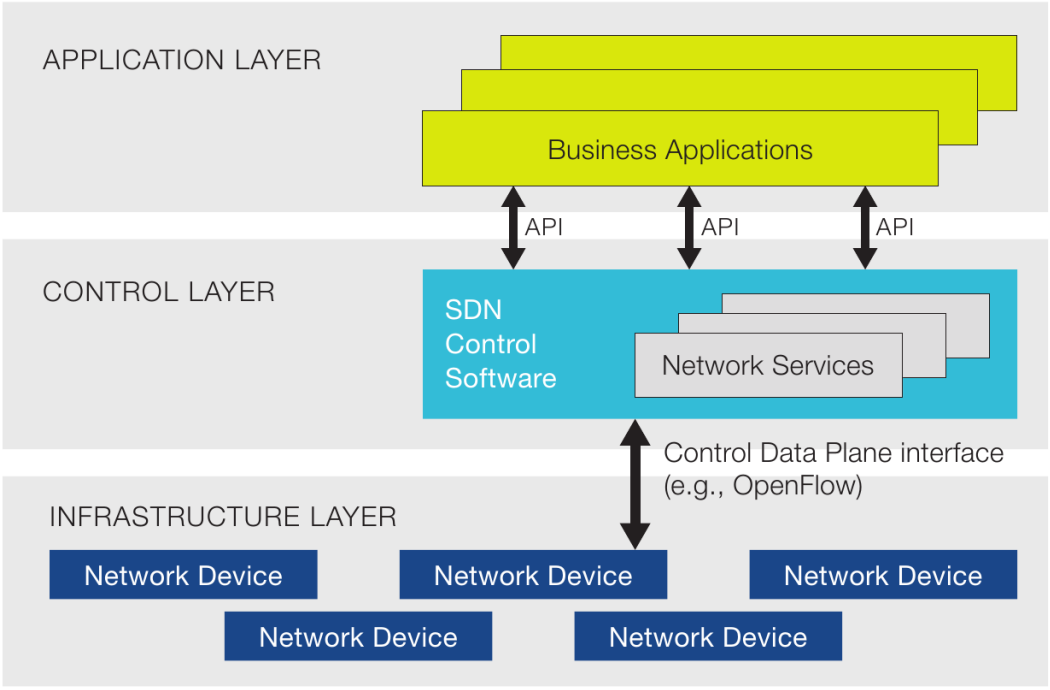


Figure 2.1. General Architecture SDN

The control plane is performed by a network operating system, which builds and presents to applications the abstract model of the whole network or one single slice, maintaining the correspondence with the effective physical topology. Moreover, the controller translates applications high-level policies in pre-defined instructions for the physical devices. Interactions with network nodes happen through a standardized interface called southbound API; actually, **OpenFlow** [3] is the standard de-facto in this role. The control plane can be a unique controller or a group of software components, where everyone manages a precise functionality. These components can both be allocated to a single node or can be distributed on many nodes, thus realizing a distributed controller

## 2.4 Network Functions Virtualization

SDN introduces virtualization into the networking world and, combined with other technologies which are becoming mainstream like cloud computing and data centers, prepares the road for the introduction of virtualization into applications, policies and services. More generally, we talk about Network Functions Virtualization (NFV), a term invented by ETSI Industry Specification Group to refer to abstract software entities which implement network functions (NAT, firewall, etc..) running on commodity servers (often virtual machines on hypervisors). The main goal of these virtual functions is to eliminate the need of dedicated network appliance, like routers, switches and firewalls.

While SDN has been created by researchers and data-center architects, NFV has been ideated by a group of Internet service providers (ISPs). SDN moves network control from hardware network devices to software processes and NFV moves all functions (control and services) from dedicated devices to general-purpose computing machines. While SDN aims to provide a centralized vision of the network for orchestration and a more efficient automation of network services, NFV focuses on optimizing these services themselves. Even if they are two different topics, they are not completely unrelated but rather they complete each other. NFV goals could be reached also without SDN techniques; however, using an SDN approach based on control and data planes separation can simplify operative procedures, improve performances and compatibility between different physical implementations. On the other hand, NFV can be used to host SDN software modules into its infrastructure.

## 2.5 Technological transition

SDN and NFV have all the credentials to become mainstream solutions for networks of the future yet, right now it is too early for their massive deployment on production infrastructures. With respect to the development process, NFV is still some step behind SDN. The second one isn't yet an entire system; the standardization process isn't finished yet (there is not a unique northbound API and every vendor has it's one) and furthermore the existing components are still work in progress (from the most famous controllers to the OpenFlow protocol itself), however it is starting to be used by big companies networks like Amazon and Google.

Rather, NFV is as yet a pure research topic and it requires a great deal of research in order to discover all its advantages and downsides. A lot of difficulties still have to be faced before reaching a satisfying implementation of NFV, for instance where to

place the computing resources into a SDN and NFV geographical network scenario (if into big datacenters far from the final users or distributed at the edge of the network) or how to reduce the expenses of a massive transition from traditional networks to this new infrastructure. Nobody knows whether this will truly transform into the revolution it is expected to be yet the quality of the research work that universities and service providers will do in this field can play a fundamental role to determine whether it will be a win or a disappointment.

## 2.6 Beyond network functions

As we have seen, the revolution of computer networking began with the progress of hardware platform and software virtualization, which prompted the birth of Cloud Computing. This approach permitted optimizing and improving typical computing services like web hosting and data storage then, the SDN approach started to redesign networks, trying to bring the benefits typically generated from software flexibility into this hardware-centric world. That opened the road to a lot of new approaches and research scenarios, for example, NFV which aims to collapse all the best features of cloud computing and SDN to provide flexible network services to end users.

The NFV approach is rather restricted since its definition; actually, it focuses particularly only network functions, which are the classic functions we use in everyday networking (NAT, DHCP, Firewall, etc...) however it doesn't consider applicative functions at all. Surely, it appears reasonable that, if we can realize in software and virtualized functions which are so bound to hardware, such as switching and routing, we can also introduce in the similar architecture functions which are much simpler to realize in software or which are already implemented in this way. These considerations explain why we decided to expand the concept of network functions virtualization and to include also this kind of features which are not proper network functions but are anyway extensively used by users over the Internet, like web servers, email servers and all that wide range of applicative servers (and customers, obviously!) effectively existing.

## 2.7 Deploying the FROGv4, Orchestrators unaligned APIs and Extended support for multi-domain

Since we are introducing flexible SDN/NFV services on a real campus network, using FROG architecture, as we have been facing many problems, the list of these problems is following.

- The problem was how to define a standard APIs that allows the FROG components to accept stander pattern.
- Each component of the FROG was problematic, having issues because of the usage of different APIs and standards. One of the main problems was the lake of coherency and consistency among the different components. This problem was preventing FROG to perform as a one unit having all its component aware of each other's and communicating each other for the end results.
- After defining the standard pattern of the APIs, we need to implement this pattern in each component of the FROG e.g. the Universal Node, the OpenStack domain, the OpenFlow domain, the Global orchestrator, the Web GUI, and the DataStore. This was a very challenging task because these components have been developed using different technologies. We needed to understand all of these components first and then update the code for our changes. Another big challenge was an integration of changes. As all the components of FROG are interrelated, they are somehow dependent on each other, so making changes to one component used to disturb many other components and hence we needed to track all the necessary changes in the other components and implement them as well.
- The FROG4-orchestrator control multi-domain environment, once the graph deployed on FROG4 orchestrator and then it wants to split the graph and sends to different domains, but currently it's not working properly and the more critical problem was when we want to update the already deployed graph on different domains. So under the update request, we needed DELETE, PUT, POST methods. DELETE was required when in the updated graph want to delete the old domain, PUST was wanted when the updated graphs use the same domain and post was demanded when the updated request added the new domain. As we had required this logic for the multi-domain environment.
- It was also causing duplication of NFFGs. Another problem with the existing architecture was that of the deadlock. Different APIs standards and because

of that, the poor communication among the components lead to inefficiency and deadlocks, For example, if two different users want to deployed different graphs using the same component of the FROG and both users using the same NFFG-ID, even the graphs was totally different but as result, we had fined one NFFG. There was also the security problem for the FROG.

- The FROG4-orchestrator operations (create, read, update, delete) was not working properly.
- We do not have the communication channel between FROG-orchestrator and FROG web GUI.
- We don't have security manager for user authentication to keep track of the users that have been authenticated in the system, without user authentication, it is not possible to draw a graph on GUI and then deploy graph on FROG-orchestrator.
- We don't have the connection between GUI and Datastore.
- We don't have CRUD (create, read, update, delete) APIs for NFFGs in FROG4 datastore.
- We needed to test real user on Cisco router using SDN/NFV services, which was the black box for me.

## 2.8 Current domain categorization

In the context of the management of network services through new technologies such as SDN and NFV, technological domain means a set of physical resources under the control of a single centralized entity that is part of the infrastructure on which the services are instantiated. The services provided by each domain can be very different from the node in the node, depending on the nature of each of them and the resources it is able to offer at a particular time. The orchestrator, the central component that coordinates the provisioning of services across domains, must take into account when scheduling the capabilities of each node so that each request can be distributed on an infrastructure that can satisfy it. In order to meet this need, the approach currently adopted is to categorize domains based on the presence or absence of generic computing capability.

# Chapter 3

# Background

The international research scenario about SDN and NFV is wide and extremely variegated, due to the wide range of connected topics.

## 3.1 The international context

Worldwide studies about SDN could be classified depending on the levels involved: physical infrastructure, control plane and applications.

SDN contributed to renew the interest in software switches realized on generic hardware platform, usually cheap and easily accessible. SDN software switches also allow realizing a simple and fast traffic commutation between virtual machines, thus introducing the possibility to integrate hypervisor capabilities into commutation devices. From this point of view, the most interesting software switches projects are the ones based on **OpenWRT** [4], a Linux distribution for embedded systems and, especially, the **OpenvSwitch** project [5], a production quality and multilayer virtual switch licensed under the open source Apache 2.0 license, which will be extensively used in this thesis work.

Even if SDN is focused on software, its growth is pushing also a certain innovation in the hardware devices world; network appliance vendors are changing their product lines to include SDN-capable devices and to find efficient ways to implement in hardware some of the SDN basics, like packet classification functions with the support of Ternary Content Addressable Memories (TCAMs) and flow-based packet forwarding with specialized Network Interface Controllers (NICs).

**Control plane** is the principal aspect of the SDN paradigm, since SDN itself is based on the existence of a centralized controller which can convert applications

requests coming from the northbound interface into packet forwarding instructions for devices linked through the southbound interface and to collect network status information from nodes. Seen its fundamental role, many research activities focused on the control plane and the results are a series of different controller implementations, diverging for the programming language used (C++, Python, Java, Ruby, etc..), for performances obtained and features offered and also for the southbound protocol used, even though OpenFlow is rapidly becoming the only one. The most famous open source controllers, developed thanks to the collaboration of many different actors in international projects, are **OpenDaylight** and FloodLight (both based on OpenFlow).

**Application layer** is actually the least explored part of SDN world and, then, the most open to innovation. For SDN applications, it is intended the software which implements high-level control and management functions of the network, interacting with the controller through the northbound interface. In other words, if the controller defines how an SDN network works, applications define what that network should do. The most studied topics at the moment are the problems regarding packet forwarding (load balancing, cross-layer design, etc..), network management (failure resilience, diagnostics), mobility support, security and energetic efficiency. Also the solutions considered in this thesis work are part of this wide group of SDN applications for network management and services virtualization.

Another aspect of interest in SDN networks is the possibility to compose flexible chains of virtualized network functions, thanks to the NFV approach defined by ETSI. Also this aspect is still almost unexplored and only recently some international research projects, made up of universities and private IT and TLC companies, started to analyse the possibilities offered from this innovative paradigm.

## 3.2 The ETSI proposal

The European Telecommunications Standard Institute (ETSI) is an institution that produces globally-applicable standards for Information and Communications Technologies (ICTs). It ranges from fixed to mobile, radio, aeronautical, broadcast and Internet technologies and is officially recognized by the European Union as a European Standards Organization. In November 2012 seven of the world leading network operators selected the ETSI to be the home of the Industry Specification Group (ISG) for Network Function Virtualization (NFV). Now, three years later, a large community of experts are working intensely to develop the required standards for

Network Functions Virtualization as well as sharing their experiences of NFV development and earlier implementations. In order to better understand what exactly NFV approach is, we are now presenting the principal guidelines of the relative ETSI proposal [6] [7].

### 3.2.1 ETSI goals

From a high level view, the objectives of the ETSI NFV group are:

- **Improve capital efficiencies**, if comparing NFV to the one obtained through dedicated hardware implementations. This is achieved by using "commercial off-the-shelf" (COTS) hardware - general purpose servers and storage devices to provide Network Functions (NFs) through software virtualization techniques. Because of their nature, these functions are commonly referred as Virtualized Network Functions (VNFs). Also the sharing of hardware and reducing the number of different physical server architectures in a network will contribute to this objective in the sense of allowing larger stock orders and hardware re-usage.

- **Improve flexibility in assigning VNFs to hardware**. This aids both scalability and largely separates functionality from location, which allows software to be located in the most appropriate places - referred to from now on as NFV Infrastructure Points of Presence (NFVI-PoPs). In the following example VNFs may be deployed at customers' premises, at network exchange points, in central offices, datacenters and so on. These features enable time of day re-usage, support for test of alpha/beta and production versions, enhance resilience through virtualization and facilitate resource sharing.

- **Provide and support a rapid service innovation** throughout automated software-based deployment.

- **Improve operational efficiency** resulting from common automation and operating procedures.

- **Reduce power usage**; this will be achieved by migrating workloads and powering down unused hardware (i.e., an idling server can be shut down).

- **Provide standardized and open interfaces** between virtualized network functions, the infrastructure and associated management entities so that such decoupled elements can be provided by different vendors.

### 3.2.2 High level framework

Network Functions Virtualization envisages the implementation of NFs as pure software entities that run over the NFV Infrastructure (NFVI). As evident from figure 3.1, three main working domains are identified in network function virtualization framework.

- **NFV Infrastructure (NFVI)**, including the diversity of physical resources and the way in which they can be virtualized. NFVI supports the execution of the VNFs.
- **Virtualized Network Function**, in the sense of the software implementation of a NF, which is capable of running over the NFVI.
- **NFV Management and Orchestration**, which covers the arrangement and life-cycle governance of physical and/or software resources that support the infrastructure virtualization other than the life-cycle management of VNFs. This point focuses on all virtualization-specific management tasks necessary in the NFV framework.

The NFV framework enables dynamic construction and management of VNF instances and the relationships between them in terms of data, control, management, dependencies and other attributes. To this end there are at least three architectural views of VNFs that are focused on different points of view and contexts of a VNF. These perspectives include:

- A virtualization deployment/on-boarding angle where the context can be a VM.
- A vendor-developed software package perspective where the context can be several inter-connected VMs and a deployment template that describes their attributes.
- An operator point of view where the context can be the operation and management of a VNF received in the form of a vendor software package.

Within each of the just mentioned contexts, at least the following relations exist between VNFs:

- A VNF Set covers the case where the connectivity between VNFs is not specified.

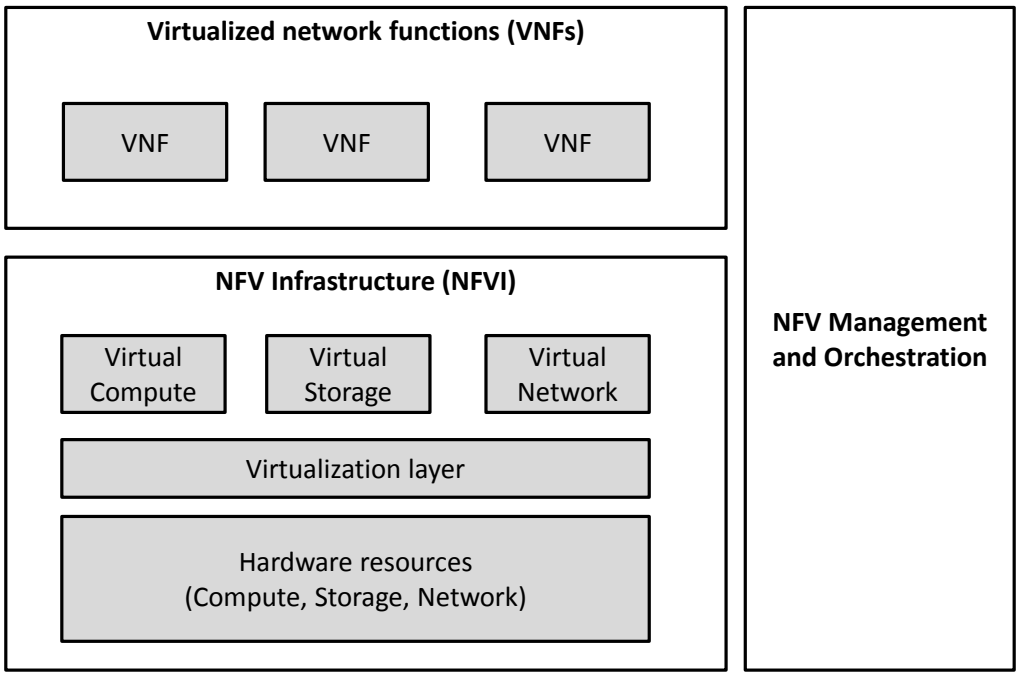


Figure 3.1. High-level view of the ETSI framework

- A VNF Forwarding Graph (VNF-FG) covers the case where network connectivity does matter, for instance a chain of VNFs in a web server tier (e.g., firewall, NAT, load balancer).

### 3.2.3 Network services

An end-to-end network service (e.g., mobile voice/data, Internet access, a virtual private network, etc..) can be described by a Network Function Forwarding Graph (NF-FG) of interconnected Network Functions (NFs) and end-points. The termination points and the NFs of the network service are represented as nodes and correspond to devices, applications, and/or physical server applications. A NF-FG can have network function nodes connected by logical links that can be unidirectional, bidirectional, multicast and/or broadcast.

In figure 3.2 is shown an example of an end-to-end network service and the different layers that are involved in its virtualization process. The depicted example offers a clear view of the abstraction (upper part) and how it is remapped on the underlying physical infrastructure (NFVI). It consists in an end-to-end network

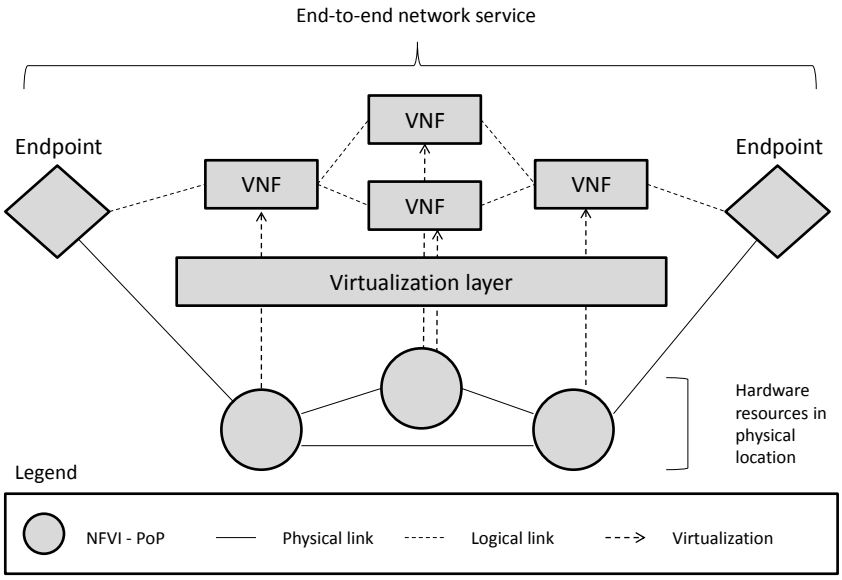


Figure 3.2. End-to-end network service

service composed of five general purpose VNFs and two termination (end) points. The decoupling of hardware and software in NFV is realized by a virtualization layer. This layer abstracts hardware resources of the NFV Infrastructure.

### 3.2.4 NFV architecture

The NFV architectural framework identifies functional blocks and the main reference points between such blocks. The functional blocks are:

- **Virtualized network function (VNF)**
- **Element management system (EMS)**
- **NFV infrastructure**, including:
  - Hardware and virtualized resources
  - Virtualization layer
- **Virtualized infrastructure manager(s)**
- **Orchestrator**
- **VNF manager(s)**

- **Service, VNF and infrastructure description**
- **Operations and Business support systems (OSS/BSS)**

The illustrated architectural framework focuses on the functionalities necessary for the virtualization and the consequent operation of operators' networks. It does not specify which network functions should be virtualized, as that is solely a decision of the network owner.

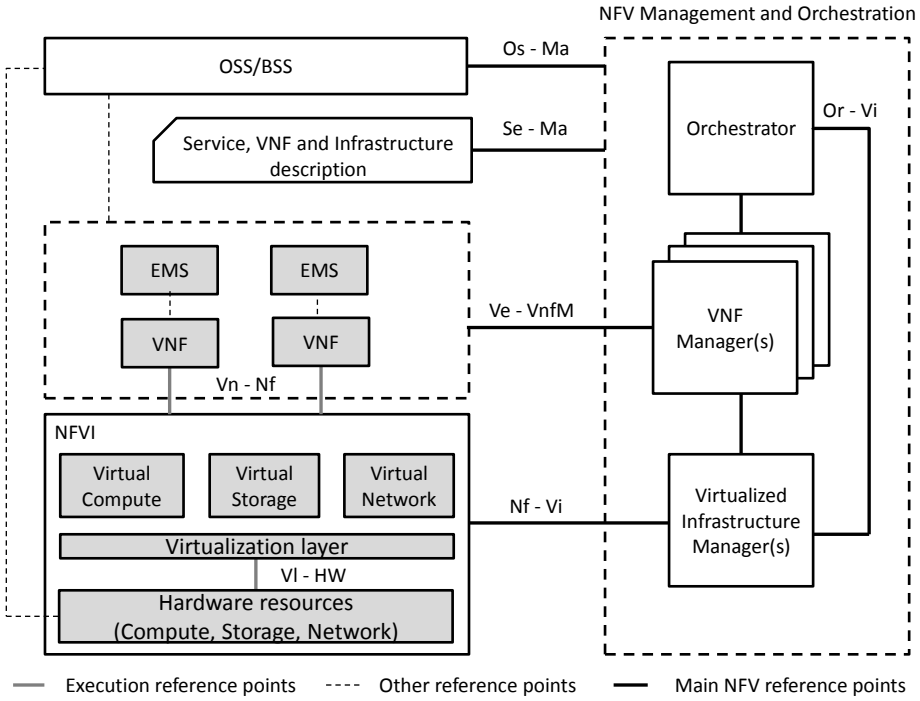


Figure 3.3. Detailed NFV framework architecture

A functional block defined by the ETSI is the basic unit and consists of:

- A set of input interfaces
- A state
- A transfer function
- A set of output interfaces

A fundamental property of functional blocks is the complete and formal separation of the static from the dynamic. Using a more IT oriented terminology, the input, output, and internal (i.e., state) data structures and all the methods (i.e., the transfer function) are static. They shall not change. Only the values given as input parameters, and therefore the outputs, can change; these values are the only things labelled as dynamic. Functional blocks are linked together following two fundamental rules:

- They can be interconnected, by connecting an output interface of one functional block with the input interface of another functional block.
- When a number of functional blocks are interconnected together forming a topology, some input and some output interfaces may remain disconnected. In this case the resulting topology is, in turn, considered as a functional block itself in which the inputs and outputs are the endpoints that remained unlinked in the previous passage. The new obtained functional block follows the very same rules as a standard one.

### 3.2.5 Templates

ETSI introduces five descriptor for deployment and life-cycle management of virtual network functions (VNF) and network services (NS):

- **Network Service Descriptor (NSD)**
- **VNF Descriptor (VNFD)**
- **VNF Forwarding Graph Descriptor (VNFFGD)**
- **Virtual Link Descriptor (VLD)**
- **Physical Network Function Descriptor (PNFD)**

A **Network Service Descriptor** is a deployment template for a Network Service referencing all other descriptors which, in turn, describe components that are part of that Network Service. In addition of containing descriptors, NSD also contains connection points and, optionally, dependencies between VNFs. The connection point is an information element representing the virtual and/or physical interface that offers connectivity between instances of NS, VNF, VNF Component (VNFC), Physical NF Descriptor (PNF) and a Virtual Link (VL).

Examples of virtual and physical interfaces are virtual ports, virtual NIC addresses, physical ports, physical NIC addresses or endpoints of an IP VPN. The

meaning of dependencies between VNFs is quickly explained throughout an example; a function must exist and be connected to the service before another can be deployed and connected.

A **VNF Descriptor (VNFD)** is a deployment template which describes the way a VNF has to be deployed and its operational behaviour requirements. It is primarily used by the VNF Manager during the process of instantiation and lifecycle management of a VNF instance. The information provided in the VNFD is also used by the NFV Orchestrator to manage and orchestrate Network Services and virtualized resources all over the NFV Infrastructure. The VNFD also contains information for management and orchestration layer (MANO) functional blocks that allow establishing appropriate virtual links with NFVI between its VNF Component (VNFC) instances or between a VNF instance and the endpoint interface that has to be linked to the other network functions.

A **VNF Forwarding Graph Descriptor (VNFFGD)** is a deployment template that differs from the others because it takes care of describing the topology of a Network Service (or a portion of it) by referencing VNFs, Physical NFs (PNF) and Virtual Links that interconnect them. Essentially, it defines the paths that different kinds of traffic have to follow and the ordered list of VNFs that they must go through.

A **Virtual Link Descriptor (VLD)** is a deployment template which describes the resource requirements that are needed for a link that will be used to connect VNFs, PNFs and endpoints of the network service; requirements could be expressed by various link options that are available in the NFVI. The NFV Orchestrator can select an option after consulting the VNFFG to determine the appropriate NFVI to be used. The choice can be based on functionality (e.g., two distinct paths to provide resiliency) and/or other needs (e.g., network physical topology, regulatory requirements, etc..).

Finally, the **Physical Network Function Descriptor** delineates the connectivity, the interface and key performance indicator requirements of virtual links that are terminated on one side by a Physical Network Function (PNF); this flexibility is needed if hardware devices are incorporated in a Network Service, for example to facilitate the transition toward a fully virtualized environment.

## 3.3 OpenFlow

The Open Networking Foundation (ONF) [3], a user-led organization dedicated to promotion and adoption of software-defined networking, manages the OpenFlow standard. ONF defines OpenFlow as the first standard communications interface defined between control and forwarding layers of an SDN architecture.

OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based). It is actually the most diffused protocol implementing the SDN southbound interface and enables controllers to determine the path of network packets through the network of switches. This separation of the control from the forwarding, together with the flexibility of software, allows a more sophisticated traffic management than what is feasible using access control lists (ACLs) and classic routing protocols. In fact, OpenFlow provides packet filtering from network to transport layer and supports a lot of protocols. Also, OpenFlow allows switches from different suppliers, often each with their own proprietary interfaces and scripting languages, to be managed remotely using a single, open protocol.

OpenFlow allows remote administration of packet forwarding tables of a layer 3 switch, by adding, modifying and removing packet matching rules and actions. This way, routing decisions can be made periodically or ad hoc by the controller and translated into rules and actions with a configurable lifespan, which are then deployed to the *Flow Table* of a switch, leaving the actual forwarding of matched packets to the switch at wire speed for the duration of those rules. Packets which are unmatched by the switch can be forwarded to the controller. The controller can then decide to modify existing flow table rules on one or more switches or to deploy new rules, to prevent a structural flow of traffic between switch and controller. It could even decide to forward the traffic itself, provided that it has told the switch to forward entire packets instead of just their header.

Actually, this protocol represents one of the most concrete successes of the software-defined networking evolution process. A number of network switch and router vendors have announced intent to support or are shipping supported switches for OpenFlow, including Alcatel-Lucent, Brocade, Huawei, Cisco, Dell, IBM, Juniper, Hewlett-Packard, NEC, and others.

### 3.3.1 Benefits of OpenFlow-Based SDN

The benefits that enterprises and carriers can achieve through an OpenFlow-based SDN architecture include:

- **Centralized control of multi-vendor environments**: SDN control software can control any OpenFlow-enabled network device from any vendor, including switches, routers, and virtual switches. Rather than having to manage groups of devices from individual vendors, IT can use SDN-based orchestration and management tools to quickly deploy, configure, and update devices across the entire network.

- **Reduced complexity through automation**: OpenFlow-based SDN offers a flexible network automation and management framework, which makes it possible to develop tools that automate many management tasks that are done manually today. These automation tools will reduce operational overhead, decrease network instability introduced by operator error, and support emerging IT-as-a-Service and self-service provisioning models. In addition, with SDN, cloud-based applications can be managed through intelligent orchestration and provisioning systems, further reducing operational overhead while increasing business agility.

- **Higher rate of innovation**: SDN adoption accelerates business innovation by allowing IT network operators to literally program-and reprogram—the network in real time to meet specific business needs and user requirements as they arise. By virtualizing the network infrastructure and abstracting it from individual network services, for example, SDN and OpenFlow give IT and, potentially even users, the ability to tailor the behavior of the network and introduce new services and network capabilities in a matter of hours.

- **Increased network reliability and security**: SDN makes it possible for IT to define high-level configuration and policy statements, which are then translated down to the infrastructure via OpenFlow. An OpenFlow-based SDN architecture eliminates the need to individually configure network devices each time an end point, service, or application is added or moved, or a policy changes, which reduces the likelihood of network failures due to configuration or policy inconsistencies. Because SDN controllers provide complete visibility and control over the network, they can ensure that access control, traffic engineering, quality of service, security, and other policies are enforced consistently across the wired and wireless network infrastructures, including

branch offices, campuses, and data centers. Enterprises and carriers benefit from reduced operational expenses, more dynamic configuration capabilities, fewer errors, and consistent configuration and policy enforcement.

- **More granular network control**: OpenFlow's flow-based control model allows IT to apply policies at a very granular level, including the session, user, device, and application levels, in a highly abstracted, automated fashion. This control enables cloud operators to support multitenancy while maintaining traffic isolation, security, and elastic resource management when customers share the same infrastructure.

- **Better user experience**: By centralizing network control and making state information available to higher-level applications, an SDN infrastructure can better adapt to dynamic user needs. For instance, a carrier could introduce a video service that offers premium subscribers the highest possible resolution in an automated and transparent manner. Today, users must explicitly select a resolution setting, which the network may or may not be able to support, resulting in delays and interruptions that degrade the user experience. With OpenFlow-based SDN, the video application would be able to detect the bandwidth available in the network in real time and automatically adjust the video resolution accordingly.

## 3.4 DoubleDecker

DoubleDecker [8] is a hierarchical distributed message system based on ZeroMQ which can be used to provide messaging between processes running on a single machine and between processes running on multiple machines. It is hierarchical in the sense that message brokers are connected to each-other in a tree topology and route messages upwards in case they don't have the destination client beneath themselves. DoubleDecker currently supports two types of messaging, Notifications, i.e. point-to-point messages from one client to another, and Pub/Sub on a topic. The Pub/Sub mechanism furthermore allows scoping when subscribing to a topic. This means that a client can restrict the subscription to messages published only within a certain scope, such as to clients connected to the same broker, a specific broker, or different groups of brokers. DoubleDecker supports multiple tenants by authenticating clients using public/private keys, encrypting messages, and enforcing that messages cannot cross from one tenant to another. Additionally there is a special tenant called 'public' that can cross tenant boundaries. This can be used in order to

connect clients that are intended to provide a public service, such as a registration service for all tenants, a name-lookup service, or similar.

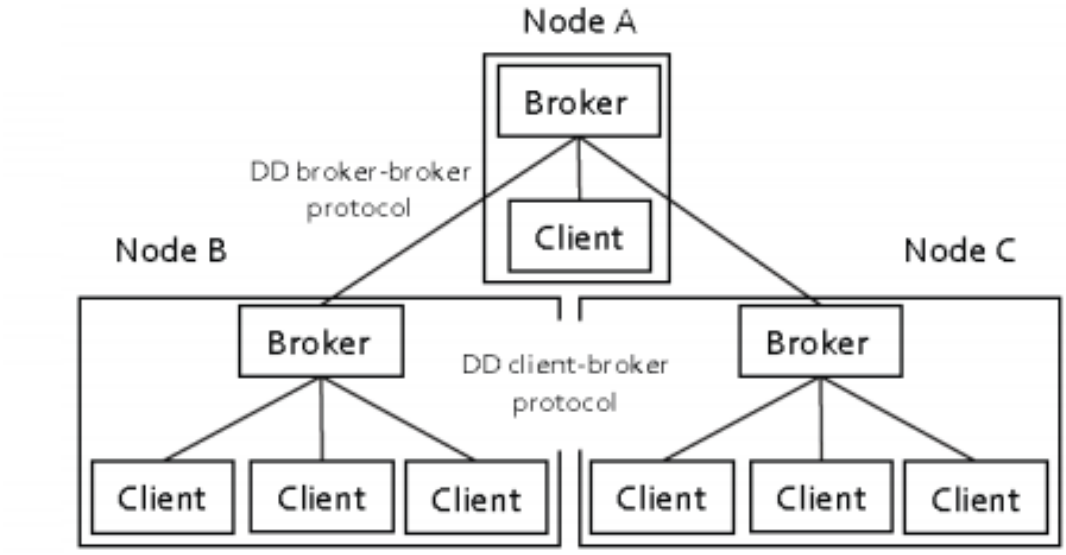


Figure 3.4. DoubleDecker's hierarchical architecture (Source: Unify [9] )

In this thesis DoubleDecker has been used as a message broker, in order to manage the connectivity between various software modules, that not necessarily are in the same machine, used in the architecture that will be presented later. We used mainly the Pub/Sub type of messaging in a single broker architecture with the idea that a software module should subscribe only to topics that are relevant for its tasks. On the other hand, information published are always related to a topic, so only who previously subscribed such topic will receive the message. Furthermore, we use the secure version of the message broker where all messages are encrypted and every software that uses DoubleDecker has its keys.

# Chapter 4

# FROG General Architecture

A recent work done at Polytechnic University of Turin proposes a model in which network service functions can be deployed per-user, by means of lightweight virtual machines [10]. In this scenario, each user is potentially allowed to customize his service chain through the insertion of functions that, operating in the network, are able to process the traffic independently from the physical terminal in use (smartphone, laptop). Elaborating more on that proposal, we can imagine that multiple actors (end users, corporate ICT managers, service providers, network providers) may be allowed to instantiate different functions, operating on the traffic of a selected group of users. In this case, the global service experimented by each user will be the composition of his own functions with the ones under the control of other actors; for example a corporate ICT manager can activate a function that prevents corporate employees from sending confidential documents to external recipients.

The system developed follows ETSI guidelines illustrated in the previous paragraph and harnesses various open source "off-the-shelf" products. it adds an additional orchestration layer which is capable of managing and organizing the work done by all these different pieces of software to achieve a global common result.

This chapter gives more details about how this orchestration layer is designed.

## 4.1 Software architecture

The FROG4 is software that is able to orchestrate NFV/cloud services across multiple heterogeneous domains. The FROG is based on multiple domain orchestrators, each one responsible for a single infrastructure domain, that cooperate by timely exporting the capabilities and the available resources in each domain to an overarching orchestrator, which has to coordinate the deployment of the service across the entire

infrastructure. Supported domains include not only traditional infrastructures with a network (e.g., Open Flow only) or compute (e.g., OpenStack) capabilities, but also resource-limited SOHO home gateways, properly extended to make it compatible with the FROG orchestrator. The set of capabilities and resources exported by each single domain, coupled with the constraints specified by the service itself (e.g., the IPSec client endpoint must stay on the tenant home gateway and not on the data centre) determines how the orchestrator splits the service graph, originating the proper set of sub-graphs that are deployed on the selected infrastructure domains. The FROG overarching orchestrator that will receive a service request will query the different infrastructure domains for their capabilities/resources and it will dynamically partition the requested service graph across the selected available infrastructure domains, determining also the network parameters that have to be used in the interconnections between domains. For instance, the orchestrator will be able to set up either the proper GRE tunnels, or VLAN-based connections, or OpenFlow paths, based on the resource exported by the involved domains, the cost of the solutions, and the constraints given by the service. This demo shows also the possibility to integrate resource-constrained devices, such as existing home gateways, in the controlled infrastructure. For instance, we will show how an home gateway, extended with NFV support, can dynamically recognize a new user connecting to it and consequently create a GRE tunnel to deliver that traffic to the proper set of VNFs that are instantiated in the operator data centre. Furthermore, some more powerful home gateways are shown as well that can execute a limited number of VNFs that is implemented as a "native software", i.e., applications running on the bare hardware, in addition to the traditional VM-based or Ducker based VNF support. Finally, the FROG architecture relies on an intermediate message bus instead of using the traditional REST API to interconnect the different components. This solution provides a clear advantage when the recipient of the information published is not known such as in the bootstrapping process, or when different components (e.g., service layer and orchestrator) need to know the same information coming from the infrastructure domains to perform their job.

The FROG orchestration architecture heavily relies on an intermediate message bus, which complements the traditional REST API to interconnect the different components. This solution provides a clear advantage when the recipient of the information published is not known such as in the bootstrapping process, or when different components (e.g., service layer and orchestrator) need to know the same information coming from the infrastructure domains to perform their job.

This system is logically composed of three main sub-modules; the service layer, the orchestration layer and the infrastructure layer.

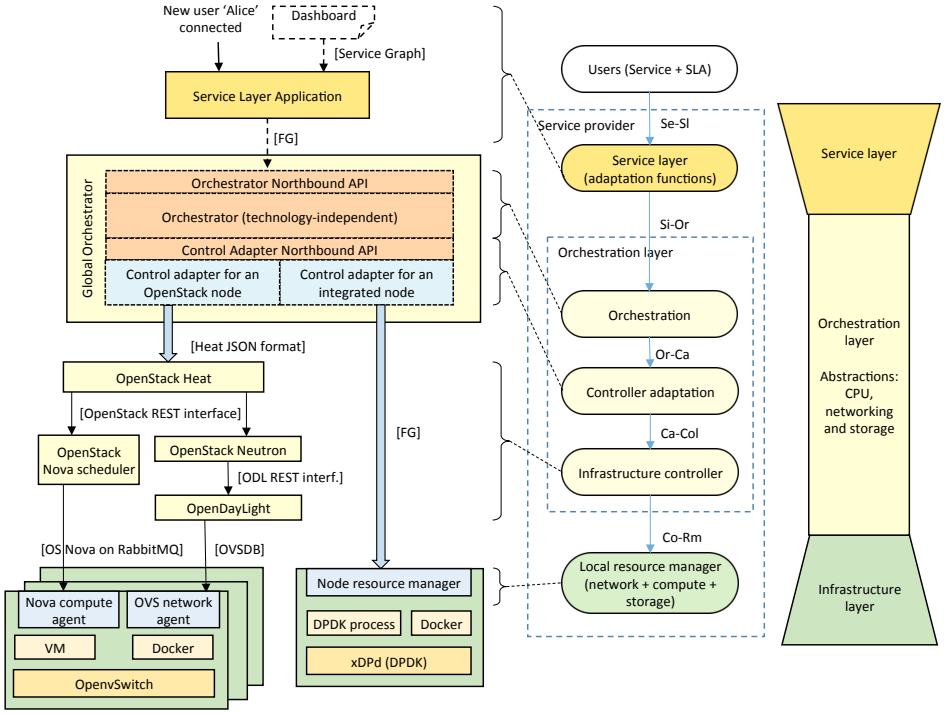


Figure 4.1. Overall view of the system architecture

The **service layer** represents the external interface of our system and allows the different actors that can potentially use our solution (e.g., end users, the network provider, third-party organizations) to define their own network services. The input of this architectural part is hence a per-actor service description, expressed in a high level formalism (called service graph) capable of describing every type of service and also the potential interactions among different services. In order to facilitate the service creation, the actors should be provided with a graphical interface that makes available the components of the service (e.g., the VNFs), and which is integrated with a marketplace that enables the selection of a precise VNF among the many available. Given the above inputs, the service layer should be able to translate the service graph specification into an orchestration-oriented formalism, namely the forwarding graph. This new representation provides a more precise view of the service to be deployed, both in terms of computing and network resources and interconnections among them. As depicted in figure 4.1, the service layer includes a component implementing the service logic (identified with the service layer application block),

in addition to an interface that can be called when specific events occur (e.g., a new end user attaches to the network) and that triggers the deployment/update of a service graph.

The **orchestration layer** sits below the service layer, and it is responsible of two important phases in the deployment of a service. First, it manipulates the forwarding graph in order to allow its deployment on the infrastructure; these transformations include the enriching of the initial definition with extra-details required from below layers. Second, it implements the scheduler that is in charge of deciding where to instantiate the requested service functions. It is composed of three different logical sub-layers. First, the orchestration sub-layer implements the orchestration logic (forwarding graph transformation and scheduling) in a technology-independent approach, without dealing with details related to the infrastructure implementation. The next component, called controller adaptation sub-layer, implements instead the technology-dependent logic that is in charge of translating the (standard) forwarding graph into the proper set of calls for the northbound API of the different **infrastructure controllers**. These controllers correspond to the bottom part of the orchestration layer, and are in charge of applying the above commands to the resources available on the physical network; the set of commands needed to actually deploy a service is called infrastructure graph. In practice, the infrastructure controllers transform the forwarding graph into the proper set of calls for the northbound API of the different infrastructure controllers, to be executed on the physical infrastructure in order to deploy the service. The infrastructure controllers should also be able to identify the occurrence of some events in the infrastructure layer (e.g., a new packet from an unknown device arrives to one node), and to notify it to the upper layers of the architecture. As shown in figure 4.1, different kind of nodes require different implementations for the infrastructure controllers (in fact, each type of nodes has its own controller), which in turn require many control adapters in the controller adaptation sub-layer. Moreover, the orchestration sub-layer and the controller adaptation sub-layer are merged together into the global orchestrator module.

The **infrastructure layer** sits below the orchestration layer and includes the physical resources where the required service is actually deployed. From the point of view of the orchestration layer, it is organized in nodes, each one having its own infrastructure controller; the global orchestrator can potentially schedule the forwarding graph on each one of these nodes. Given the heterogeneity of modern networks, we envision the possibility of having multiple nodes implemented with different technologies. Each node of this kind actually consists of a cluster of physical machines managed by the same infrastructure controller.

## 4.2 Data models

Data models used are inspired by ETSI NFV standards which propose a service model composed of functional blocks connected together to flexibly realize a desired service. All these data models are stored and manipulated as JSON files which contain all the required information.

### 4.2.1 Service graph

A service graph is a high level representation of the service to be implemented on the network, and it includes both aspects related to the infrastructure (e.g., which network functions implement the service, how they are interconnected among each other) and to the configuration of these network functions (e.g., network layer information, etc..). From the point of view of the infrastructure, the SG consists of the set of basic elements which describe the service.

The basic elements of a service graph are:

- **Network function**: it represents a functional block that may be lately translated into one (or more) VNF images. Each network function is associated with a template describing the function itself in terms of RAM and CPU required, number and types of ports, and other information.
- **Active port**: it defines the attaching point of a network function that needs to be configured with an IP address, either dynamic or static.
- **Transparent port**: it defines the attaching point of a network function whose associated virtual network interface card (vNIC) does not require any network level address.
- **LAN**: The availability of this primitive facilitates the creation of complex services that include not only transparent VNFs, but also traditional host-based services that are usually designed in terms of LANs and hosts.
- **Link**: it defines the logical wiring among the different components, and can be used to connect two VNFs together, to connect a port to a LAN, and more.
- **Traffic splitter and merger**: functional block that allows to split the traffic based on a given set of rules, or to merge the traffic coming from different links.

- **Endpoint**: it represents the external attaching point of the SG. It can be used to attach the SG to the Internet, to an end user device, but also to the endpoint of another service graph, if several of them have to be cascaded in order to create a more complex service.

As cited above, the SG also includes aspects related to the configuration of the network functions required by the service; particularly, this information includes network aspects such as the IP addresses assigned to the active ports of the VNFs, as well as VNF-specific configurations, such as the filtering rules for a firewall. In fact, they represent important service-layer parameters to be defined together with the service topology, and that can be used by the control/management plane of the network infrastructure to properly configure the service.

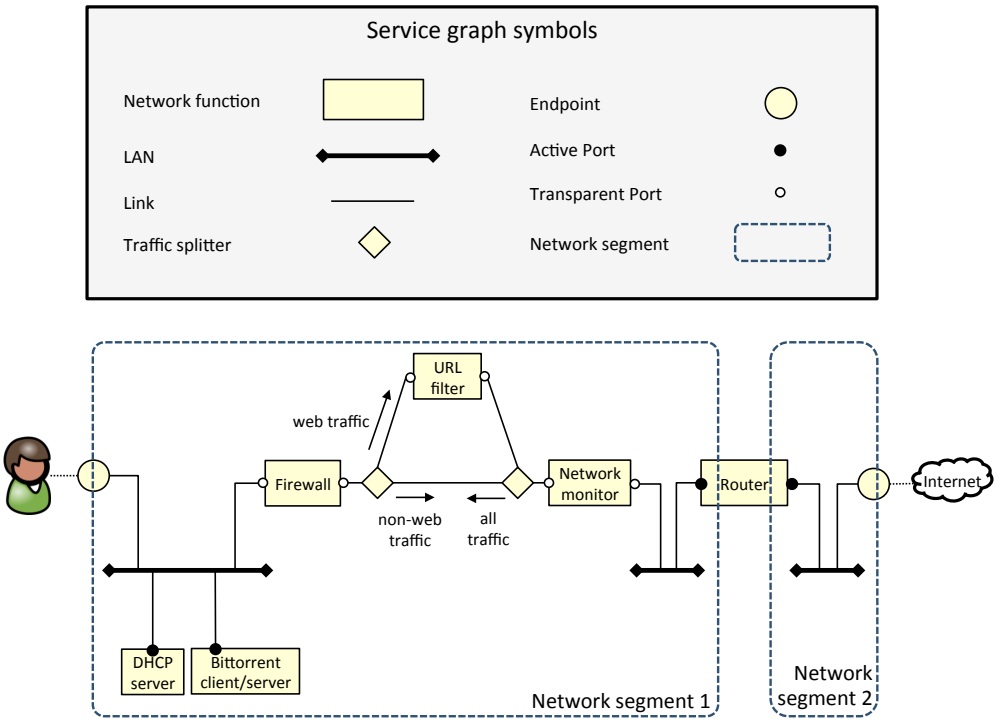


Figure 4.2. Service graph example

### 4.2.2 Forwarding graph

The service graph provides a high level formalism to define network services, but it is not adequate to be deployed on the physical infrastructure of the network, since

it does not include all the details needed by the service to operate. Hence, it must be translated into a more resource-oriented representation, namely the forwarding graph (FG), through the so called lowering process.

The sequence of operations which composes the lowering process is:

- The service is enriched with the **control and management network**, which may be used to properly configure the VNFs of the graph. In fact, most network functions require a specific vNIC dedicated to the control/management operations; although this may be an unnecessary detail for the user requiring the service, those network connections have to be present in order to allow the service to operate properly.

- All the LANs expressed in the SG are replaced with VNFs implementing the MAC learning switch. This step is needed in order to translate the abstract LAN element available in the SG into a VNF actually realizing the broadcast communication medium.

- The graph is analysed and enriched with those functions that have not been inserted in the SG, but that are required for the correct implementation and delivery of the service, for example DHCP and NAT services.

- A VNF may be decomposed in a number of VNFs, properly connected in a way to implement the required service. Moreover, these new VNFs are in turn associated with a **template**, and can be recursively expanded in further subgraphs; this is an implementation of the *recursive functional blocks* concept provided by NFV definition in ETSI standard.

- Consolidation, through the replacement with a single VNF, of those VNFs implementing the L2 forwarding that are connected together, in order to limit the resources required to implement the LANs.

- The graph **endpoints** can be converted in physical ports of the node on which the graph will be deployed; tunnel endpoints (e.g., GRE) used to connect two pieces of the same service but on different physical servers or endpoints of another FG, if many graphs must be connected together in order to create a more complex service.

- Finally, the **flowrules** definition concludes the lowering process. In particular the connections among the VNFs, as well as the traffic steering rules (expressed through the traffic splitter/merger components in the SG) are represented with

a sequence of rules, each one indicating which traffic has to be delivered to a specific VNF (on a given port of that VNF), or the physical port/endpoint through which the traffic has to leave the graph.

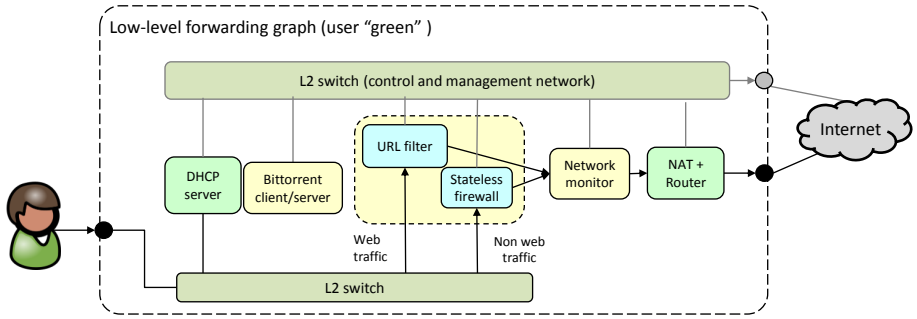


Figure 4.3. Forwarding graph example

As already introduced, the formalism used for graphs is JSON. At the first level, the structure of the FG is the following:

```
{
  "Forwarding-graph": {
    "VNFs": [],
    "endpoints": [],
    "name": "Forwarding_graph_name"
  }
}
```

Listing 4.1. High-level view of a NFFG

Essentially, it presents a list of virtual network functions and a list of endpoints, plus a unique identifier and a name for the user's profile. Virtual network functions are characterized by a descriptor, a list of ports, a name and an identifier.

```
"VNFs":[
        {
            "vnf_descriptor":"manifest",
            "ports":[],
            "name":"VNF_name",
```

```
            "id":"VNF_id"
        },
        ...
    ],
```

Listing 4.2. High-level view of a VNF

The ports described in VNFs JSON object are the ports actually used by a VNF, while a complete list of ports available for a VNF is contained in the VNF template. The descriptor is an URL containing the manifest of the virtual network function.

The port list contains an id, an ingoing label, and an outgoing label. The port id is composed of two different parts, the part before the column identifies the label of the port; the second part is an id for all ports with same label. The outgoing label contains flow rules that only identify outgoing traffic from the port, while ingoing label contains flowrules only for ingoing traffic to that port. While the outgoing labels are mandatory, the ingoing labels are needed only when a port is connected to an endpoint, since the endpoint does not have any flowrule associated. In addition, flowrules contained in an ingoing label have an additional field to identify the endpoint from which the traffic comes. This field is called ingress endpoint and it is a leaf of flowspec object. Finally, the flowrule object, as discussed before, contains a list of matches on packets and the relative action.

```
"ports":[
            {
                "id":"port_id",
                "ingoing_label":{
                    "flowrules":[]
                },
                "outgoing_label":{
                    "flowrules":[]
                }
            }
    ],
```

Listing 4.3. High-level view of the ports list

```
"flowrules":[
            {
                "id": "flowrule_id",
                "action":{
```

```
                    "VNF":{
                        "id":"VNF_id",
                        "port":"VNF_port"
                    },
                    "type":"output"
                },
                "flowspec":{
                    "matches":[
                        {
                            "priority":"priority",
                            "id":"match_id"
                        }
                    ]
                }
            }
```

Listing 4.4. High-level view of a flowrule

The flowspec supports all the fields defined by Openflow 1.0 (although new fields can be defined), while the action can refer to a forwarding of packets either through a physical port, through a logical endpoint, or through a port of a VNF. Hence, the FG is actually a generalization of the OpenFlow data model that specifies also the functions that have to process the traffic into the node, in addition to define the (virtual) ports the traffic has to be sent to. It is worth noting that all the flowrules of a single port must forward the totality of traffic, hence, a rule of a specific port cannot purge the traffic, and if we want to drop some kind of traffic we must do that in a VNF (for example in a firewall function).

```
"endpoints":[
        {
            "name":"endpoint_name",
            "type":"endpoint_type",
            "id":"endpoint_id",
            "interface":"physical_interface"
        },
    ],
```

Listing 4.5. High-level view of an endpoint

The endpoints are the termination of graph. In the FG, instead of the SG, the

endpoints can assume various characterizations like tunnel terminations, physical ports or virtual ports. This characterization is needed to effectively connect graphs among each other and map the endpoint concept on physical resources. With regard to name, it provides a tool to implement the logic of connection between graphs.

### 4.2.3 Infrastructure graph

An infrastructure graph consists instead of the sequence of commands to be executed on the physical infrastructure in order to properly deploy the required VNFs and to create the paths among them. The IG is obtained through the so called reconciliation process, which consists in the mapping of the FG description on the resources available on the infrastructure, thanks to the infrastructure managers APIs. For example, it convert the endpoints of the graph into physical ports of the node on which it is going to be deployed and, if required, instruct the infrastructure controller to create GRE tunnels to connect graphs.

### 4.2.4 Functions Template

Each network function is associated with a template, which describes the VNF itself both in terms of infrastructure and in terms of configuration.

```
{
  "CPUrequirements": {
    "platformType": "x86",
    "socket": [
      {
        "coreNumbers": 1
      }
    ]
  },
  "memory-size": 2048,
  "name": "firewall",
  "functional-capability": "firewall",
  "ephemeral-file-system-size": 0,
  "vnf-type": "docker",
  "uri-image-type": "docker-registry",
  "swap-disk-size": 0,
  "expandable": false,
```

```
18    "ports": [
19      {
20        "name": "eth",
21        "min": "1",
22        "label": "inout",
23        "ipv4-config": "none",
24        "position": "0-10",
25        "ipv6-config": "none"
26      }
27    ],
28    "root-file-system-size": 40,
29    "uri-image": "firewall"
30  }
```

Listing 4.6. Example of a function template

As evident from listing example, the template contains information related to the hardware required by the VNF, namely the amount of memory and CPU, as well as the architecture of the physical machine that can execute it and the requirements of disk in terms of swap, root file system and ephemeral file system size. Moreover, the boolean element *expandable* indicates if the VNF consists of a single image, or if it is actually a sub-graph composed of several VNFs connected together. In the former case, the *uri* element refers to the image of the VNF, while in the latter it refers to a graph description, which must replace the original VNF in the forwarding graph. In case of non-expandable VNF, the template also specifies the type of the image; for instance, the firewall described is implemented as a single virtual machine.

Moreover, the template provides a description of the ports of the VNF, each one associated with several parameters. In particular, the *label* specifies the purpose of that port, and it is useful in the definition of the SG, since it helps to properly connect the VNF with the other components of the service. The label could assume any value, and it is meaningful only in the context of the VNF. The parameter *ipv4-config*, instead, indicates if the port cannot be associated with an IPv4 address (none), or if it can be statically (static) or dynamically (DHCP) configured, the same applies to *ipv6-config*. The field position specifies both the number of the ports of a certain type and the internal index of the interfaces. The number of ports is given by the difference between the second and the first number of the range more one (e.g. "position": "1-2" means there are 2 ports of that label). Position specifies also, along to the field name, the effectively name of the internal interface of VNF. In particular, the first number in the range of field position acts as an offset for the

id used to reference a port in FG. For example, if in the FG there is a port with id equals to "internal:2" (hence we have at least three ports labelled as internal), the name of the internal interface of the VNF is "eth4", because the value position for ports labelled as internal is "2-N" and the value of field name is "eth".

### 4.2.5 Domain abstraction

A single domain orchestrator manages computing and networking infrastructures whose technical details have to be kept from the outside world whereas its resources and capabilities should be provided "as a Service". Therefore, the DO exports an abstract idea of itself according to the *Big-Switch Approach*: the DO is characterized as a switch with several endpoints, resources and capabilities. Every endpoint has in turn some characteristics (e.g., subinterfaces, neighbors, VLAN tags, GRE tunnels, etc.).

The domain abstraction is stated in a manually written file by the domain administrators; moreover, since the forwarding graphs cause internal changes (e.g. resource consumption), this file is dynamically updated by the DO in order to always export the actual status of its resources and capabilities.

This file is written in JSON format and complies with the OpenConfig [11] data model; some addictions and customizations have been made by the NetGroup. Listing 4.7 gives an example about the JSON structure.

```
{
  "netgroup-domain:informations": {
    "name": "domain_1",
    "type": "UN",
    "netgroup-network-manager:informations": {
      "openconfig-interfaces:interfaces": {
        "openconfig-interfaces:interface": [
          {
            "name": "eth1",
            "config": {
              "type": "ethernetCsmacd",
              "enabled": true
            },
            "openconfig-interfaces:subinterfaces": {
              "openconfig-interfaces:subinterface": [
                {
                  "config": { ... },
```

```
18                    "capabilities": {
19                      "gre": "true"
20                    },
21                    "netgroup-if-gre:gre": [
22                      {
23                        "config": {
24                          "name": "gre_500",
25                          "enabled": true
26                        },
27                        "state": { ... },
28                        "options": {
29                          "local_ip": "3.3.3.3",
30                          "key": "156"
31                        }
32                      }
33                    ]
34                  }
35                ]
36              },
37              "openconfig-if-ethernet:ethernet": {
38                "openconfig-if-ethernet:config": {
39                  "mac-address": "aa:bb:cc:dd:ee:ff"
40                },
41                "openconfig-vlan:vlan": {
42                  "openconfig-vlan:config": {
43                    "interface-mode": "TRUNK",
44                    "trunk-vlans": ["1..20",25,39,65]
45                  }
46                },
47                "netgroup-neighbor:neighbor": [
48                  {
49                    "domain": "domain_2",
50                    "interface": "eth3"
51                  }
52                ]
53              }
54            }
55          ]
```

```
56          }
57        }
58     }
59 }
```

Listing 4.7. OpenConfig data model, with NetGroup customizations (JSON)

Digging into details this is the resource abstraction provided by a domain orchestrator that regards the domain "domain_1" as evident looking at the *name* field. It contains information related to the interface "eth1". Line 19 tells that this interface is able to create GRE tunnels. Starting from line 21, there is description of GRE tunnels that insist on that interface. From line 37 we can see the description of the interface at the ethernet level and VLAN configuration of this interface, if any. In this case this interface is in trunk mode and a list of free VLAN tags is provided. Finally, the neighbor field claims that this interface is directly connected to the interface "eth3" of "domain_2".

All these information are used by the global orchestrator to take decisions when a graph is intended to be split. For this purpose, the most valuable information are those regarding GRE and VLAN capabilities and also the neighbor element. The latter can be set to "internet" if such interface has connectivity to the Internet; in that case a GRE tunnel is suitable to connect such interface to another domain . Furthermore, free VLAN tags are very important when connecting to another domain because the orchestrator has to perform a sort of *negotiation* phase, looking for, eventually, VLAN tags free on both parts; otherwise, if there are no available VLAN tags, VLANs cannot be used to interconnect those interfaces. It is worh pointing out that this formalism supports single and ranges of VLANs; referring to this example free VLAN tags are from 1 to 20 (included) and 25, 39 and 65.

## 4.3 Dynamic functions instantiation

In order to dynamically instantiate users' network functions graphs, the service layer must be able to recognize when a new end user attaches to the network and authenticate him. Since the infrastructure layer does not implement any (processing and forwarding) logic by itself, this operation requires the deployment of a specific graph which only receives traffic belonging to unauthenticated users, and which includes some VNFs implementing the user authentication. Moreover, in order to enable the resource consolidation, this authentication graph is shared among several end users.

In addition, the user SG must be completed with a way to inject, in the graph itself, all the traffic coming from/going to the end user terminal, so that the service defined by an end user operates only on the packet belonging to that user. This characteristic allows attaching and clearly detaching the user device dynamically from a certain SG. This may allow the user device to communicate on a certain SG and be suddenly detached and reattached on another graph when a certain event happens (e.g.: user's device has been authenticated). This abstraction is massively leveraged by the service layer, which is in charge of managing the application logic.

The service layers application designed for dynamic instantiation supposes that all the users are always connected to the authentication graph through a rule that sends all the traffic to that graph. In this way, the user can immediately reach the authentication web portal and authenticate himself. The application is then able to intercept the event of a new flow available in a node at the edge of the network, and receives, through a proper API, from the lower layers of the architecture the update of the authentication graph, so that it can properly handle the traffic of the new user device. Through this API, the application also knows the source MAC address of the new packets, which can be used to uniquely identify all the traffic belonging to the new end user. Hence, the authentication graph is enriched with a rule matching the specific MAC address; this way, the new packets enter into the graph for the user authentication, wherever the graph itself have been deployed. Finally, the updated graph is provided to the orchestration layer, which takes care of applying all the operations on the physical infrastructure, as explained before. This modification permits a dynamic connection of users to our network service, and makes it much more flexible.

As expected, the user authentication triggers the instantiation of his own SG and, at this point, the application retrieves the proper JSON graph description from database and starts the lowering process aimed at translating this high level view of the service into a FG. Before being finally provided to the orchestration layer, the FG is completed with a rule matching the MAC address of the user device, so that only the packets belonging to the user himself are processed into his own graph.

Moreover, the orchestrator keeps track of user sessions, in order to allow a user to connect to his own SG through multiple devices at the same time without any duplication of the graph itself. In particular, when an already logged-in user attaches a new device to the network, the service application retrieves the FG already created from the orchestration layer; extends it by adding rules so that also the traffic coming from/going to the new device is injected into the graph and, eventually, sends the new FG to the orchestration layer.

# Chapter 5

# Extension and validation of the FROG

This chapter deals with the Integration and validation of a FROG4-Orchestrator. Beyond the standard features described in **Chapter 4**, this orchestrator has some more specific characteristics that we will illustrate in details. As a consequence of the idea to create an extremely flexible working environment to support the users and offer them a very simple mechanism to implement and then manage their independent virtual spaces using multi-domain environment.

## 5.1 FROG4 Orchestrator

The Global orchestrator [12] corresponds to the first two levels of the orchestration layer, and consists of a technology dependent part and a technology independent part; we replace a technology dependent part with southbound API. We have no more technology dependent part. The technology independent part receives the user login requests and NFFGs created by the web GUI or service layer, through the northbound API. It manipulates the requests and gives them to the southbound API; which sends the resulting NFFGs to the proper infrastructure domains. It is worth noting that our architecture consists of a single global orchestrator that sits on top of multiple infrastructure domains, even implemented with different technologies e.g. the Universal Node, the OpenStack domain, and the OpenFlow domain. When the technology independent part of the global orchestrator receives the NFFG, it executes the following operations.

The global orchestrator creates virtual topology basing on current domain information (which outlines capabilities and neighbor Information). The virtual topology

is created using the following information:

- The "neighbor" parameter indicates whether a connection between two interfaces (of different domains) may exist or not.
- A virtual channel is established between two interfaces per each pair they have in common, Virtual channels can be established also through domains attached to a legacy network.

At this point, the global orchestrator schedules the NFFGs on the proper infrastructure domains. Although the general model presented in Section 4.2.5 supports a scheduling based on the set of capabilities and resources exported by each single domain, coupled with the constraints specified by the service itself. The FROG orchestrator sees the network infrastructure as a set of domains

- Associated with a set of functional capabilities
- Interconnected through "virtual channels"
- The FROG orchestrator is not aware of the nature of the domain (e.g., it does not know whether the selected domain is an SDN network or a data center)

The resulting NFFG is then provided to the southbound API, it takes care of translating the NFFG provided by the technology independent part into a formalism accepted by the all the infrastructure domains, which send the commands to the infrastructure layer. Moreover, they convert the endpoints of the graph into physical ports of the domain on which it is going to be deployed and, if required, instruct the infrastructure domain, to create a GRE tunnel or VLAN on the infrastructure layer. GRE tunnel could be used to connect together two pieces of the same service but deployed on different domain of the infrastructure layer; a GRE tunnel or VLAN is required when the NFFG associated with an end user is deployed on a different domain than the one used by his traffic to enter in the provider network, but also to bring the traffic generated by new end users to the authentication graph.

However, since the current implementation of the scheduler splits a graph into multiple parts but it does not work properly, As a final remark, the global orchestrator now support the updating of existing graphs which was a more difficult task. In fact, when it receives an NFFG from the web GUI, it checks if this graph (i.e., an NFFG with the same identifier) has already been deployed; in this case, the Global orchestrator computes an operation to discover the differences between the

two graphs and according to the outcome of this operation the graph is updated, preserving unchanged parts, deleting removed parts and adding new parts. For example, if the updated graph contains the information for the same domain and also addressed to a different domain compared to the graph already instantiated. The graph will be split, the updated subgraph is sent to the same domain orchestrator of the existing graph using PUT method. The new graph is instantiated on the new domain by means of the appropriate domain orchestrator, but using POST request and finally, the old graph is deleted from the old domain through the concerned domain orchestrator. In the conclusion, under the update splitting graph request there are PUT, POST, and DELETE operations take place.

At the current status of development, the abstraction level described above has not yet been reached. The northbound and southbound interfaces are not pure magnetic interfaces. Currently, the "magnetic paradigm" is only used for the resource description; therefore, a "magnetic interface" only reads and exports services, resources and capabilities whereas data are exchanged via the REST interface of each component.

As the Global Orchestrator is a special component; it doesn't control any infrastructure domains directly however it coordinates several domains to deploy the requested NFV and SDN services. Basically, this orchestrator performs the following operations:

- User Authentication and Token system
- Receives the NF-FG from upper layer (Web-GUI or service layer), which outlines NFV and SDN services Using standard APIs
- Receives the Domain Information from Infrastructure domains, which outlines Capabilities and Hardware Information
- Figures out which domains must be included with the deployment
- Split the NF-FG into many sub-graphs and sends each sub-graph to some particular domains Using standard APIs

## 5.2 Security Manger

Global orchestrator and domain orchestrator both are requiring the authentication; indeed, every REST request must include valid authentication data, otherwise, the

orchestrator returns the HTTP response “401 Unauthorized”. To improve security, a token-based authentication is implemented to avoid a continuous exchange of username and password; these credentials are only used when a new token is needed. To get a new token, orchestrator provides a specific REST URL where an HTTP POST request has to be sent.

```
Request URL: "/login"
Request Method: POST
Header:
        Content-Type: application/json
Payload (example):
        {"username":"demo", "password":"demo"}
```

Listing 5.1. Login POST request example

The HTTP POST response only returns the token, it will compliant if user not used a proper JSON schema. All the subsequent REST requests must include this token value inside the header field “X-Auth-Token”. This is an example response, when the credentials are valid:

```
Status Code: 200 OK
Payload (example):
   6a188b06-3786-4c38-9a2c-4ac1910975
```

Listing 5.2. Successful user Login response example

```
Request URL: "/login"
Request Method: POST
Header:
        Content-Type: application/json
Payload (example):
        {"username":"demo", "password":"demo",
           "tenant":"demo_tenant"}
```

Listing 5.3. Old Login request example

As old method had three types of parameters (username, password, and tenant) and there was no concept of a token-based verification and every time before any operation we continuous exchange of username, password and tenant. For instance, we want to draw a graph on GUI and then want to deploy the same graph on FROG afterwards we should use Username, Password, and Tenant, but we don’t need the tenant field from GUI, So we removed the tenant parameter from Global orchestrator.

### 5.2.1 Users authentication API and token system Implementations

In FROG the users must authenticate themselves before performing any operation. After the user authentication, the FROG contacts the Domain orchestrators through the southbound API and triggers the deployment of the proper NFFGs on the infrastructure domain. This service is intended to provide authentication.

Global orchestrator stores password and username for each approved user, the approval procedure includes getting a token to be used later to give a proof of personality. Once the token has been acquired, the user can perform many operations, such as deploying, modifying and deletion of the NFFGs on the FROG. A token is made once user authentication has been checked, we can also set the time limit from the configuration file of the FROG-orchestrator, after the time limit the token will expire and user needs to get a new token. Credentials are just used when another token is required. To get another token, FROG-orchestrator gives a particular REST URL where a POST request must be sent.

- POST /login: login, so as to identify the users who make the requests for further operations and in the response of user identification, user will get a token;

## 5.3 FROG4 Orchestrator Northbound API

The REST interface that is available thanks to a HTTP server, it is submodule of the FROG-orchestrator that interacts with the web GUI and service layer in order to perform some operations needed to deploy a new graph, update an existing graph, delete an existing graph, get the list of already deployed graphs, get the list of domain information and authenticate to GUI.

The PUT API of Global-Orchestrator at the URL is; {Global_Orchestrator _Address/NF-FG} while the PUT API of un-orchestrator at the URL is; {Domain_Orchestrator _Address/NF-FG/Graph-id}. When GUI deploys a graph on Global-Orchestrator using PUT API {Global _Orchestrator_Address /NF-FG/Graph-id}, this method was not allowed in the PUT API of Global-Orchestrator. We made some changes in the APIs; we decided to use the PUT API for an update of the graphs and created new POST API for new deploying graphs.

The CRUD (create/POST, read/GET, update/PUT, delete) operations are now supported for NFFGs. The main new REST commands available are detailed in the

remainder of this section.

- POST /NF-FG/: New instantiation of a graph on the FORG-Orchestrator;
- PUT /NF-FG/nffg-id: Update an already deployed graph on the FORG-Orchestrator;
- GET /NF-FG/: Returns already deployed graphs on the FORG-Orchestrator;
- GET /NF-FG/status/nffg-id: Returns the status of an instantiated graph on the FORG-Orchestrator;
- DELETE /NF-FG/nffg-id: Deletes an instantiated graph on the FORG-Orchestrator;
- GET /NF-FG/nffg-id: Get a JSON representation of a graph instantiated on the FORG-Orchestrator;
- POST /login: login, so as to identify the users who make the requests for further operations and in the response of user identification, user will get a token;
- GET /Domain-information/: Returns an active domain in the FORG-Orchestrator;

### 5.3.1 NFFGs Deployments/Create/POST Request

This is the main operation supported by our architecture and the most used. It corresponds to an HTTP POST operation performed on the global orchestrator through northbound API, a new instantiation of NFFGs using HTTP POST method.

http://Global_Orchestrator_Address:port/NF-FG/

Where we set the Global-orchestrator address and port, the global orchestrator is listing to mention port. We don't declare the graph id, because it will decide by the FROG, and sent back to the response of this POST operation.

The northbound API receives the POST request, created by the web GUI or REST client. At first, the users must give a proof of personality as a token, once the token verifies the farther operations take place. After the user authentication, orchestrator validates the NFFG JSON schema. If the request does not follow the standard JSON schema, it returns the validation error message. The graph ID is a unique identifier for each graph; we had mentioned the graph-id in two places one in the request path and other one keeps inside the JSON of the graph. We removed the graph-id from the graph JSON. The new POST request must be sent without using the graph id.

After the validation of the graph, the orchestrator starts exploring the graph json sent from the upper layers, we suppose that we already received domains information. It creates virtual topology base on current domain information. The virtual topology has created a base on the "neighbour" parameter indicates whether a connection between two interfaces (of different domains) may exist or not and A virtual channel is established between two interfaces per each pair they have in common, Virtual channels can be established also through domains attached to a legacy network. If the virtual topology finds feasible domains then it performs the scheduling algorithm, tag NFFG untagged elements with the best domain. For more details, you can check the Southbound API section 6.1

At this stage, we stored the NFFG details in the database, as we needed this information for an update, delete and get operations. NFFG-Id is assigned by the FROG4 Orchestrator. We set the UUID as a string, for example

```
1 {
2     "nffg-uuid":"99a18b46-3786-4c38-5t2c-4a75c1910975"
3 }
```

Listing 5.4. NFFG UUID Example

After the successful deployment of the graph, return the graph-id which is used for updating, deletion and getting the graphs.

### 5.3.2 NFFGs List/Read/GET Request

All the active graphs of the Global orchestrator can be retrieved by the GUI, contacting the northbound API interface through the GET method. In particular the GUI can ask for the specific graph, or all the active graphs of the FROG orchestrator. Two different types of GET APIs implemented which were not present.

- only one specified graph retrieved and URL is

  http://Global_Orchestrator_Address:port/NF-FG/graph-id:

- All active graphs retrieved and URL is

  http://Global_Orchestrator_Address:port/NF-FG/:

The FROG-orchestrator receives this request, first, it checks the user's token, once the token verifies, it checks the existence of the graph indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. Otherwise, it starts building of the correspondent NFFG JSON model and

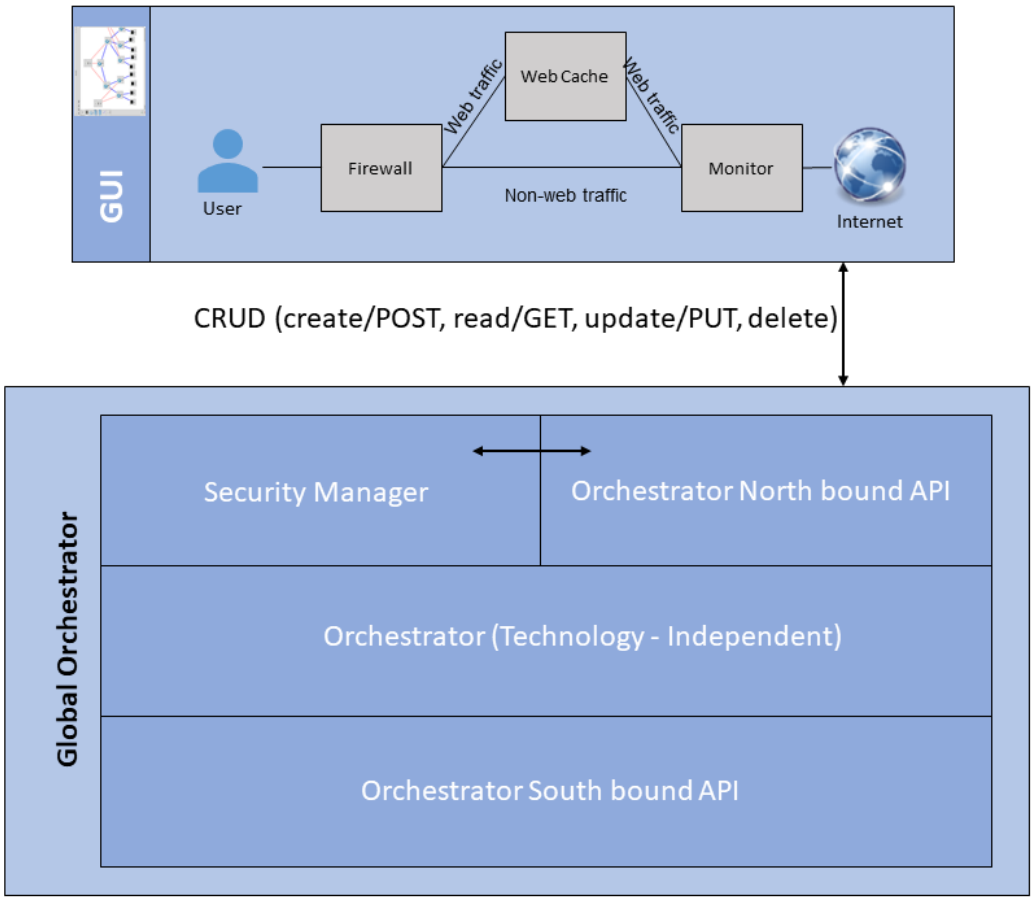


Figure 5.1. Global orchestrator supported the CRUD (create/POST, read/GET, update/PUT, delete) operations for NFFGs

exports the graphs. Since Graph ID is deleted from the json of NFFG, so when we retrieved all the deployed graphs, they are formatted as follows:

```
{
  "NF-FG": [
    {
      "Forwarding-graph": {
        "name": "Test NFFG 1",
        "VNFs": [...],
        "end-points": [...],
        "big-switch": {
          "flow-rules": [...]
```

```
10            }
11          },
12          "nffg-uuid": "1ace6489-2e27-4990-b7f0-a74ac4594bb8"
13        },
14        {
15          "Forwarding-graph": {
16            "name": "Test NFFG 2",
17            "VNFs": [...],
18            "end-points": [...],
19            "big-switch": {
20              "flow-rules": [...]
21            }
22          },
23          "nffg-uuid": "f703c8b8-f19d-4cad-b258-46f36c4572eb"
24        }
25    ]
26  }
```

Listing 5.5. List of the all the deployed graphs

The retrieved graph contains all the information of NFFG including the "domain" tag when the FROG sends NFFG to underline domains then this "domain" parameter is omitted by the FROG.

### 5.3.3 NFFG Update/PUT Request

All the deployed graphs of the Global Orchestrator can be updated by the GUI, contacting the northbound API interface of the FROG orchestrator through the PUT operation. The GUI can ask for the specific graph which is going to update. This is the main operation, now supported by our architecture.It corresponds to an HTTP PUT operation performed on the Global orchestrator through northbound API. The PUT method URL is:

http://Global_Orchestrator_Address:port/NF-FG/graph-id

The Global orchestrator receives PUT request, first, it checks the user's token and checks its validity, once the token verifies, it checks the existence of the graph with the same graph-id, indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. After the checking of the existence graph, orchestrator validates the NFFG JSON schema; if the request does

not follow the standards JSON schema, it returns the validation error message. The orchestrator starts exploring the graph JSON schema sent from the upper layers. It calculates the difference between the old deployed graph and new update graph. The remaining procedure is quite the same of the one for the POST method.

At this point, the updated NFFG are now saving in the database. After the successful updating of the graph, it doesn't return the Graph ID, because we already have it and it only returns status "202".

### 5.3.4 NFFGs Deletion Request

The global orchestrator obviously supports for the deletion of instantiated graphs. The GUI contacting the northbound API interface of the Global orchestrator through the DELETE operation. The GUI asks for the specific graph which is going to be deleted. In this case, the operation performed on the REST APIs of the orchestrator is an HTTP DELETE containing the ID of the Forwarding Graph that should be deleted. The DELETE request at URL is:

http://Global_Orchestrator_Address:port/NF-FG/graph-id

The FROG orchestrator receives DELETE request, first, it checks the user's token and checks its validity, once the token verifies, it checks the existence of the graph indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. After the checking of the existence graph, it deletes the graph from the orchestrator. In the other cases, it returns an error message to specify if the graph wasn't finding. After the successful deletion of the graph, it just returns status "204".

## 5.4 FROG4 Web GUI

Now it is possible to add a GUI on top of the FROG4-Orchestrator and of each domain orchestrator The GUI can be used to:

- Draw a service graph (on load it from an external file) and deploy it through the interaction with the underlying orchestrator
- View and modify service graphs already deployed, through the interaction with the underlying orchestrator
- Upload NF images, templates and Network Function Forwarding Graph in the Datastore.

Graphical User Interface [13] that permits to create, modify and view complex virtualized service graphs. It can connect both with the Global orchestrator and with the particular domain orchestrator. Nonetheless, it is just a proof of concept and can be replaced by other service layers. The GUI facilitates the end users interested in using the underlying orchestrator without knowing its implementation aspects by providing a user-friendly graphical user interface that avoids the direct use of the REST APIs to communicate with the underlying orchestrator.

You can do the following through the GUI:

- **Create a new document**: Starting from an empty Graph, appropriate buttons allow the creation and characterization of service access points, the addition of network functions, and the creation of flow rules that link endpoints and VNFs through specified matches and actions user;
- **View instantiated graphs**: the GUI is able to query the underlying orchestrator and return the list of graphs deployed on the domains. through a select box is you can decide which graph is to display;
- **Save/load a graph**: Once you've built a graph, you can choose to save it on a json file (with the formal NF-FG) for future use or a possible sharing. Similarly, you can load one graph from the outside, in order to use or make changes; moreover you can also save this graph in datastore for future use.
- **Instantiate a graph**: through a simple click, the user may choose to instantiate the service graph on the underlying orchestrator or modify a pre-existing one, so that it can take advantage of the new service.

### 5.4.1 GUI Southbound API for NFFGs and Connection to word FROG4 Orchestrator

The GUI should also define some APIs to be exported to the FROG4 orchestrator the code could need to be slightly changed to adapt to the new compunctions channel. The REST interface that is available thanks to Southbound API, it interacts with the Global orchestrator in order to perform some operations needed to deploy a new service graph, update an existing service graph, delete an existing service graph, get the list of already deployed service graphs. The PUT API of Global Orchestrator at the URL is; {Global_Orchestrator_Address/NF-FG}.When GUI deploys a graph on Global Orchestrator using PUT API {Global_Orchestrator_Address/NF-FG/Graph-id}, this method is not allowed in the PUT API of FROG. We made

some change in the APIs; we choose to use PUT API for updating of the service graphs and create new POST API for deploying new service graphs on the GUI. The CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for service graphs.

The main new REST commands available are.

- POST /NF-FG/: New instantiation of a service graph on the underlying orchestrator;
- PUT /NF-FG/: nffg-id: update the already deployed service graph on the underlying orchestrator;
- GET /NF-FG/: Returns an already deployed service graphs on the underlying orchestrator;
- GET /NF-FG/status/nffg-id: Returns the status of an instantiated service graph on the underlying orchestrator;
- DELETE /NF-FG/nffg-id: Deletes an instantiated service graph on the underlying orchestrator;
- GET /NF-FG/nffg-id: Retrieved a JSON representation of a service graph instantiated on the underlying orchestrator;
- POST /login: login, so as to identify the users who make the requests for further operations and in the response of user identification, user will get a token;

Create/POST is the main operation supported by the GUI. It corresponds to an HTTP POST operation performed on the underlying orchestrator through northbound API, new deploying of service graph using the HTTP POST request and the URL is.

http://Underlying_orchestrator_Address: port/NF-FG/

At the beginning of the deploying service graph, when we click on the new document button the service graph id was generated automatically by the GUI, which was incorrect, the graph-id will returns after the deploying service graph on the Underlying orchestrator, then Update the service graph id at the top corner of GUI, and from now GUI will not decide the Service Graph ID, it is also deleted from the JSON of the graph. To send this POST request to Global orchestrator, it add token into the header of the request, if the token is not valid it receive error message for the token. Example of use of the token:

```
Request URL: "/NF-FG/"
Request Method: POST
Header:
   Content-Type: application/json
   X-Auth-Token: 6a188b06-3786-4c38-9a2c-4ac1910975
Payload:
    ...
```

Listing 5.6. Example of use of the token

All the active Graphs of the FROG orchestrator can be updated by the GUI contacting the northbound API interface of the FROG-orchestrator through the PUT request. It corresponds to an HTTP PUT operation performed on the global orchestrator through northbound API. The PUT method URL is:

http://Underlying_orchestrator_Address: port/NF-FG/graph-id

To send this PUT request to Global orchestrator, it add token into the header of the request, if the token is not valid it receive error message for the token. If the updated graph not found in the FROG-orchestrator, it receives a 404 NOT FOUND message.The GUI also supports for the deletion of instantiated graphs. The GUI contacting the northbound API interface of the FROG-orchestrator through the DELETE request. The operation performed on the REST APIs of the GUI is an HTTP DELETE containing the ID of the service graph that should be deleted. The DELETE request at URL is:

http://Underlying_orchestrator_Address: port/NF-FG/graph-id

The token and the Graph existence will be checked by FORG orchestrator, if it doesn't find the correspondent graph, GUI will receive 404 NOT FOUND message. All the active graphs of the frog-orchestrator can be retrieved by the GUI, contacting the northbound API interface through the GET method. When we use GUI to deploy graph on the frog, we have element name Gui-position in graph Json but this element was missing in nffg-library in JSON Schema. We added this parameter in many places in Graph JSON schema.

From the northbound API of GUI, we created two interfaces one is used by the Global orchestrator and domains orchestrators, which supports all the CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for service graphs. Another interface is used for the Data Store which also managed the CRUD (create/POST, read/GET, update/PUT, delete) operations are supported

for NFFGs.

## 5.5 FROG4 Datastore

Now it is possible to add a Datastore [14] for the GUI to stores NF-FGs, the datastore is a helper module that contains Network Function Forwarding Graphs, NF images and NF templates. The Datastore can be used to:

- **NF-FG**: file that describes a network functions forwarding graph, written according to a proper schema.
- **NF template**: file that describes the characteristics of the network function, such as its capability (e.g., firewall, NAT), required resources (e.g., amount of CPU, amount of memory), required execution environment (e.g., KVM hypervisor, Dockers, etc), number and type of virtual interfaces, and more. Examples of templates, which have to follow a proper schema, are available in sample-templates;
- **NF Capability**: list of all the functional capabilities supported by the FROG v.4 architecture. Such a list is updated each time a template with a new capability is uploaded into the datastore.
- **NF image**: a raw image of the NF (e.g., VM disk, archive file). The NF image can be installed either directly in the Datastore, or in a different backend (e.g., Open Stack Glance);
- **YANG model**: schema used to validate the configuration forwarded to the VNF by means of the configuration service
- **YIN model**: JSON representation of a YANG model, it is used by the GUI in order to provide the users with a simple with interface for the management of their services
- **User**: stores all the user information (username, password and broker keys). Moreover, the data store provides a rudimental authentication service that can be exploited by all the FROG v.4 architecture components.
- **VNF Configuration**: configuration that will be loaded into a VNF at booting time

- **Active VNF**: stores information of the VNF that are currently active (instance ID, bootstrap configuration and REST endpoint that can be used in order to configure the VNF)

### 5.5.1 Rest API Implementation and communication channel between Datastore and Web GUI

The Datastore should also define some APIs to be exported to the GUI, The REST interface that is available thanks to northbound API, it interacts with the GUI in order to perform some operations needed to stored a new NF-FGs, update an existing NF-FGs, delete an existing NF-FGs, get the list of already stored NF-FGs. The CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for Network Function Forwarding Graphs. The new REST APIs for Network Function Forwarding Graphs available are.

- POST /v2/NF-FG/: New insertion of a Graph in the datastore;
- PUT /v2/NF-FG/nffg-id: Update an already stored Graphs in the datastore;
- GET /v2/NF-FG/: Returns JSON demonstration of an already stored Graphs in the datastore;
- GET /v2/NF-FG/nffg-id: Retrieved a JSON representation of a graph saved on the in the datastore;
- DELETE /v2/NF-FG/nffg-id: Delete the stored graph in the datastore;
- GET /v2/NF-FG/digest:Returns the names and graph-ids of the all saved graphs in the datastore;
- GET /v2/NF-FG/capability: Returns the list of all the functional capabilities supported by the FROG architecture.

Since we don't have Rest APIs for Network Function Forwarding Graphs in the Datastore. The first API that we created was the POST API, it is the main operation supported by Datastore and it corresponds to an HTTP POST operation performed in the datastore through northbound API, new insertion of NF-FGs using HTTP POST method.

http://Data_store_Address:port/v2/NF-FG/

We don't have mention graph-id because it is chosen by the datastore, it will send back to the response of this POST request.

The northbound API receives the POST request, created by the web GUI. The data store checks the validation of the Network Function Forwarding Graphs JSON schema; if the request does not follow the standard JSON schema, it returns the validation error message. After the validation we stored the NFFG in a database, Graph ID is assigned by the datastore. After the successful saving of the graph, return the graph ID, which is using for updating, deletion and listing of the graphs. All the saved graphs of the datastore can be updated by the GUI contacting the northbound API interface of the Datastore through the PUT request. It corresponds to an HTTP PUT operation performed on the data-store through northbound API, The PUT method URL is:

http://Data_store_Address:port/v2/NF-FG/nffg-id

The data store checks the validation the NFFG JSON schema; if the method does not follow the standard JSON, it returns the validation error note. At this point, the updated NFFG are now saving in the datastore. After the successful updating of the graph, it doesn't return the graph id because we already have it and it returns the status "202". The datastore also supports for the deletion of saved graphs. The GUI contacting the northbound API interface of the datastore through the DELETE request. The operation performed on the REST APIs of the Datastore is an HTTP DELETE containing the ID of the forwarding graph that should be deleted. The DELETE request at URL is:

http://Data_store_Address:port/v2/NF-FG/nffg-id

The graph existence will be checked by the data store, if it doesn't find the correspondent graph, GUI will receive 404 NOT FOUND message. All the saved graphs of the datastore can be retrieved by the GUI, contacting the northbound API interface of the datastore through the GET request.

From the northbound API of GUI, we created two interfaces one is used by the Global orchestrator and domains orchestrators, which supports all the CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for service graphs. Another interface is used for the Data Store which also managed the CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for NF-FGs. Now when a user using GUI, wants to save their graphs in the datastore and then later wants to retrieve the graph at the same time, a user can deploy it on FROG4 orchestrator.

# Chapter 6

# Implementation of unaligned APIs and extended support for multi-domain

In this chapter, we discuss how graphs can be deployed on the infrastructure domain using standards REST APIs. It is a strong base for the multi-domain support that is discussed in this chapter.

The operations supported by the FROG orchestrator from North to south:

- From FROG Orchestrator to domain orchestrators
- Carries service graphs described according to the NF-FG formalism
- Based on a REST interface
- CRUD (create, read, update, delete) operations supported on service graphs

The operations supported by the FROG orchestrator from South to north:

- Carries the domains description from domain orchestrators to the FROG orchestrator
- Based on a message bus that allows communication through the publisher/-subscriber paradigm

  – The interested modules (e.g., the FROG orchestrator) subscribe to a specific topic

  – The domain orchestrators publish the domains description on that specific topic

- A broker module is actually in between all the modules that are part of FROG
- The message bus allows the FROG architecture to be easily extended with new modules that are interested in information about domains E.g., new service layer applications

We have defined three domain orchestrators, capable to deploy service graphs in different domains

- OpenStack-based data center:
  - OpenStack to manage virtual machines and intra-domain traffic steering
  - (optional) ONOS/OpenDaylight to manage the inter-domain traffic steering
- SDN network under the control of ONOS or OpenDaylight
- Universal node orchestrator.
  - Lightweight orchestrator for resource-constrained devices (e.g., CPE)
  - Can start VNFs in virtual machines, Docker containers and on the bare metal

## 6.1 FROG4 Orchestrator Southbound API

The interaction between the infrastructure Orchestrator and the FROG-orchestrator are possible thanks two different technologies. The first one is the Double Decker bus, that was explained before, and that is used by the infrastructure Orchestrator, to export the domain Information, which outlines capabilities and hardware information. The second one is the REST technology, which allows the FROG-orchestrator to contact the infrastructure Orchestrator and send commands to it. The problem was how to define a standard APIs that allows the infrastructure Orchestrator to accept the FROG-orchestrator pattern. The REST interface that is available thanks to a HTTP server, it is sub module of the FROG-orchestrator that interacts with the infrastructure Orchestrator in order to perform some operations needed to deploy a new graph, update an existing graph, delete an existing graph, get the list of already deployed graphs, and authentication of the user. The CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for NFFGs.

The main new REST APIs available are detailed in the reminder of this section.

- POST /NF-FG/: New instantiation of a Network Function Forwarding Graph on the infrastructure domains;

- PUT /NF-FG/nffg-id: Update an already deployed Network Function Forwarding Graph on infrastructure domains;

- GET /NF-FG/: Returns an already instantiated Network Function Forwarding Graph on infrastructure domains;

- GET /NF-FG/status/nffg-id: Returns the status of a deployed Network Function Forwarding Graph on the infrastructure domains;

- DELETE /NF-FG/nffg-id: Deletes an instantiated NFFG on the infrastructure domains;

- GET /NF-FG/nffg-id: Get a JSON representation of a Graph deployed on the infrastructure domains;

- POST /login: login, so as to identify the users who make the requests for further operations and in the response of user identification, user will get a token;

In addition to FROG components, there is a module that implements the REST descript interfaces, and another that has the Double-decker client function, which as described in the previous chapter, deals with exporting an abstract template describing the domain information, which outlines capabilities and hardware information. Each domain orchestrator maintains and publishes a description of its resources. The description includes the nodes/interfaces of each domain that may be used to reach other domains, including the supported technologies (e.g., GRE tunnels, VLAN). When a domain orchestrator sends the description on the message bus for the first time, the FROG orchestrator becomes aware of such domain and learns how to contact it. Resources descriptions examples can be found in the configuration directory of each domain orchestrator repository. Particularly, it is important to set the domain orchestrator IP and port in the management-address field, to choose a domain name in the name field and to describe each interface. This information will be used by the FROG orchestrator to eventually split and deploy over multiple domains an incoming service graph.

Anyway, we are not diving into the source code but we are just illustrating it from its behaviour. The source code is open and available in our public repository [15], for a person who wants to read it. The main operations are presented below.

### 6.1.1 Authentication and Token system for infrastructure Domains

Every operation requested to the global orchestrator requires a preliminary authentication phase and the same applies between the global orchestrator and the domain orchestrators. To achieve this, every orchestrator has its copy of the user's database and at every interaction the requester identity is verified. This can be seen as a limitation but we assume that the global orchestrator and the various domain orchestrators can be in very different environments. From our point of view domain orchestrators can be even inside the Customer Premise Equipment (CPE) and it should have a local database, provided that the latter should be light and simple because we have to support also resource-constrained devices.

The global orchestrator requires that every request received by its northbound API contains the user's token. In case the token is not valid no action is performed and an error message is returned. Otherwise, if a token is correct, the operation requested could involve also one or more domain orchestrators. If this is the case, another authentication phase is needed, between the global and the domain orchestrators:

1. Username and password are sent to the domain orchestrator's authentication URL through an HTTP POST operation

   – If credentials are correct the domain orchestrator replies with a token that is used in all subsequent operations.

   – If credentials are not valid the process is stopped and an error is returned.

2. The global orchestrator stores the token related to that specific domain orchestrator and includes it in every request till the token is valid.

3. If the token stored by the global orchestrator is expired, the first operation that includes it will fail and the login described in point 1 is repeated.

The login is performed only at the first interaction between the global orchestrator and that specific domain orchestrator. All subsequent interactions, until the token expires, include the token in the HTTP header. In addition, some domain orchestrators' northbound API supports HTTPS; therefore all data including sensitive information are secure. It Sends authentication request to the domain orchestrator on basis of current operation and just only to an involved domain.

### 6.1.2 NFFG deployments/Create/POST operation on the infrastructure Domains

This is the main operation supported by **Southbound API** of the Global orchestrator. It corresponds to an HTTP POST operation performed on the infrastructure Domains northbound API. We already have the domain information, thanks to the domain descriptions exported by domain orchestrators; the FROG orchestrator sees the network infrastructure as a set of domains

- Associated with a set of functional capabilities
- Interconnected through "virtual channels"
- The FROG orchestrator is not aware of the nature of the domain (e.g., it does not know whether the selected domain is an SDN network or a data center)

The first phase is to create virtual topology basing on current domain information. The virtual topology is created using the following information

- The "neighbor" parameter indicates whether a connection between two interfaces (of different domains) may exist or not.
- A virtual channel is established between two interfaces per each pair they have in common, Virtual channels can be established also through domains attached to a legacy network.

The second step is fetching a list of feasible domains for each Network Functions and endpoints of the NFFGs. Then perform the scheduling algorithm, selects the most suitable domain(s) involved in the graph deployment. Based on the description of each domain provided by the domain orchestrator. Identify best domain(s) that will actually implement the required Network Functions, links and endpoints. The FROG orchestrator uses a greedy approach that minimizes the distance between two VNFs/endpoints directly connected in the service graph. Some endpoints are forced to be mapped to specific domain interfaces because they represent the entry point of the user traffic into the network. A VNF must be executed in a domain that advertises the corresponding functional capability and Links between VNFs/endpoints deployed in different domains require the exploitation of virtual channels for inter-domain traffic.

The third step is to generate a sub graph for each involved domain. This subgraph includes

- VNFs assigned to that domain
- Possibly, new endpoints generated during the placement process
- Originated by links that connects VNFs/endpoints mapped to different domains
- Two endpoints originated by the same link are connected through a virtual channel
- If VNFs are assigned to two domains connected by means of a third domain
- An additional sub-graph is generated for the intermediate domain as well, This sub-graph just includes network connections and endpoints
- Obviously, if all elements are tagged with the same domain the splitting is not necessary and in this case, the entire graph will be instantiated on that domain.

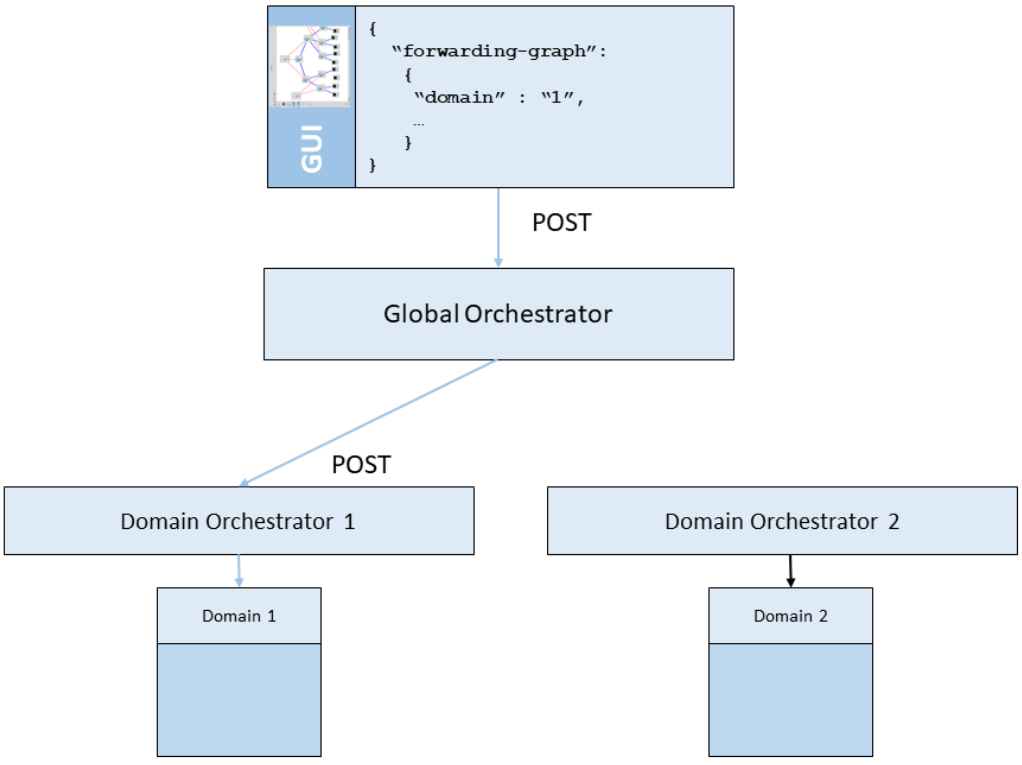


Figure 6.1. NFFG deployments on the FROG

At this stage, the subgraph is ready to deploy for each involved domain. Save the sub-graph information in the database which is different than the main graph received by Northbound API, some information like the "domain" field is omitted, if graph also uses VLAN or GRE information, this information is also saved in the database of the FROG4- orchestration, this information is useful for updating case. Then deploy the subgraph on each involved domain using POST API.

Once the subgraph deploys on each involved domain, FORG orchestrator waits for an ACK from domain orchestrators, it sent back the subgraph ID, which is used for updating, deletion of the sub-graph.

Looking at figure 6.1, we omitted Double Decker because we suppose that the global orchestrator has already received the domains information from the depicted domains. In this example NFFG, whose snippet is shown in the upper part of the figure, arrives to the global orchestrator through an HTTP POST and the latter forwards the POST operation to the appropriate domain orchestrator (domain orchestrator 1 in this case), that will actually deploy the NFFG on the underlying domain

### 6.1.3 NFFG Update/PUT operation on the infrastructure Domains

When the Network Function Forwarding Graph arrives through an HTTP PUT operation to the orchestrator, it checks if a graph with the same ID is already instantiated, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. After the checking of the existence graph, if the graph already exists. then it follows the same step as we already described in the previous section, we can check this details in section 6.1.2 . And requires more steps to complete the update operation on the infrastructure Domains

Depending on which domain the updated graph is intended to be deployed on each involved domain, the global orchestrator will organize the appropriate operations. The FROG-orchestrator computes an operation to discover the differences between the two graphs and according to the outcome of this operation the graph is updated, preserving unchanged parts, deleting removed parts and adding new parts. If the updated graph is addressed to a different domain compared to the graph already instantiated, the old graph is deleted from the old domain through the concerned domain orchestrator. The new graph is instantiated on the new domain by means of the appropriate domain orchestrator, but using POST request not PUT request on each involved domain.

If the updated graph is targeted to the same domain of the graph already instantiated, the updated graph is sent to the same domain orchestrator of the existing graph using PUT method.

If the updated graph contains the information for the same domain of the graph and also addressed to a different domain compared to the graph already instantiated. The graph will be split, the updated sub graph is sent to the same domain orchestrator of the existing graph using PUT method. The new graph is instantiated on the new domain by means of the appropriate domain orchestrator, but using POST request each involved domain and finally, the old graph is deleted from the old domain through the concerned domain orchestrator. In the conclusion, under the update graph request there are PUT, POST, and DELETE operations take place.

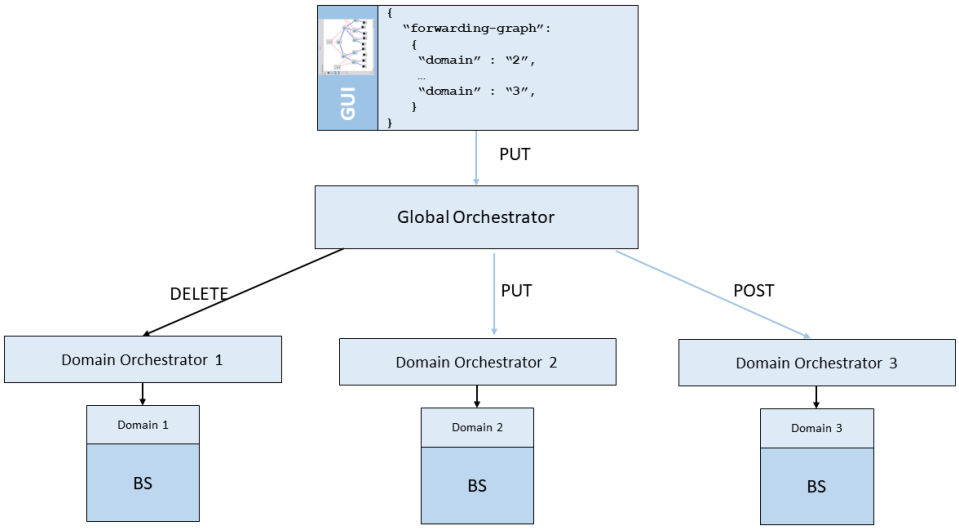


Figure 6.2. NFFG Update on the FROG

The figure 6.2 follows the previous example and describes the last case explained in this section: this is an update, because a NFFG with the same graph ID is already instantiated, but the new NFFG is addressed to the domain 3, while the currently NFFG is on domain 1 and domain 2. This situation triggers the deletion of the NFFG instantiated on domain 1 through the domain orchestrator 1, the update of the old NFFG on domain 2 through the domain orchestrator 2 and the new instantiation of the new NFFG on domain 3 through the domain orchestrator 3.

### 6.1.4 NFFG deletion

The global orchestrator obviously supports also the deletion of instantiated graphs. In this case, the operation performed on the REST APIs of the orchestrator is an HTTP DELETE containing the ID of the graph that has to be deleted. the orchestrator forwards that delete operation to the correct domain orchestrator, who oversees the actual deletion of the graph from its domain and then notifies the global orchestrator about the outcome of the operation.

Following the running example of this chapter, figure 6.3 deletes the NFFG that is instantiated on domain 1, by means of the domain orchestrator 1.

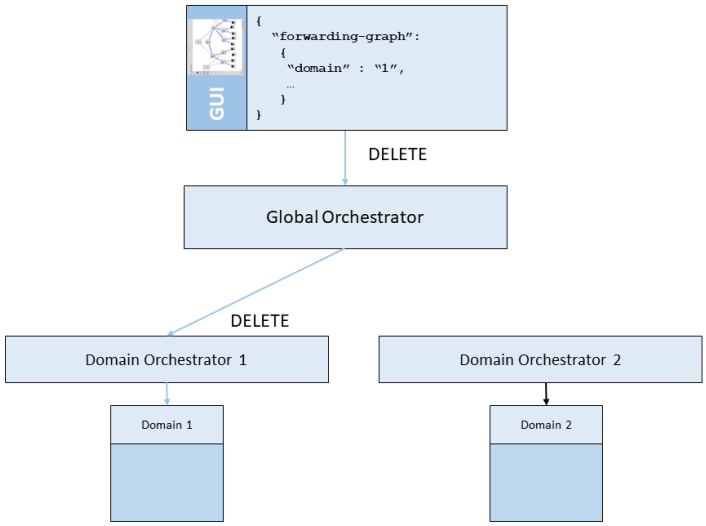


Figure 6.3. NFFG Deletion

## 6.2 SDN Domain Orchestrator REST API for NF-FGs

The OpenFlow domain [16] is a group of OpenFlow switches managed by an SDN controller (we support ONOS and OpenDaylight). The SDN domain orchestrator interacts with the controller in order to retrieve domain information to be exposed on the message bus E.g., boundary interfaces and available VNFs. VNFs are implemented as software bundles executed in the SDN controller to create network paths and start the software bundles implementing the required VNFs Use VLAN

tags to set up intra-domain paths. SDN domain orchestrator knows all the internal details of the underlying domain through its northbound API, interacts with the infrastructure-specific domain controller to fulfil requests coming from the FROG orchestrator.

The Open Flow Domain Interface provides a set of REST APIs which the external users can use to interact with the orchestration actions. List of all these REST APIs explains how to compose the HTTP requests and illustrates the respective HTTP responses. The REST interface that is available thanks to a HTTP server, is the submodule of the OpenFlow Domain Orchestrator that interacts with the web GUI and FROG-orchestrator in order to perform some operations needed to deploy a new graph, update an existing graph, delete an existing graph, get the list of already deployed graphs, and user authentication. The CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for NFFGs.

The new main REST APIs available for NFFGs.

- POST /NF-FG/: New instantiation of a graph on the OpenFlow Domain Orchestrator;
- PUT /NF-FG/nffg-id: Update an already deployed graph on the OpenFlow Domain Orchestrator;
- GET /NF-FG/: Returns an already instantiation graphs on the OpenFlow Domain Orchestrator;
- GET /NF-FG/status/nffg-id: Returns the status of an instantiated graph on the OpenFlow Domain Orchestrator;
- DELETE /NF-FG/nffg-id: Delete a deployed graph on the OpenFlow Domain Orchestrator;
- GET /NF-FG/nffg-id: Get a JSON representation of a graph instantiated on OpenFlow Domain Orchestrator;
- POST /login: login, so as to identify the users who make the requests for further operations and in the response of user identification, user will get a token;

The POST/create is the main operation supported by OpenFlow Domain Orchestrator. It corresponds to an HTTP POST operation performed on the OpenFlow Domain Orchestrator through northbound API. We made some changes in the APIs,

we decided to use PUT API for an update of the NFFGs and created new POST API for new instantiation graph. The POST method URL is:
http://OpenFlow_Domain_Orchestrator_Address:port/NF-FG/

The northbound API receives the POST request, created by the web GUI or FROG-orchestrator. At first, the users must give a proof of personality as token, once the token verifies the farther operations take place. After the user authentication, orchestrator validates the NFFG JSON schema; if the request does not follow the standard JSON format, it returns the validation error status. The orchestrator starts exploring the graph Json sent from the upper layers

- ✓ Set up GRE tunnels if any
- ✓ Send flow rules to Network Controller
- ✓ Activate needed applications
- ✓ Update the resource description

At this stage, we stored the NFFG in the database, as we need this information for an update, delete and get operations. NFFG ID is assigned by the OpenFlow Domain Orchestrator. We set the UUID [17] as a string, for example

```
1 {
2     "nffg-uuid":"0b2451e1-13a3-4747-ba0e-d66281c92702"
3 }
```

Listing 6.1. OpenFlow Domain Orchestrator NFFG UUID Example

After the successful deployment of the graph, return the graph-id which is used for updating, deletion and getting the graphs. All the active graphs of the OpenFlow Domain Orchestrator can be retrieved by the GUI or FROG-orchestrator, contacting the northbound API interface through the GET method. In particular, the GUI can ask for the specific graph or all the active graphs of the OpenFlow Domain Orchestrator. Two types of Getting APIs implemented which were not present before.

- Only one specified active graph retrieved and the URL is :
  - http://OpenFlow_Domain_Orchestrator_Address:port/NF-FG/graph-id
- All active graphs retrieved and the URL is
  - http://OpenFlow_Domain_Orchestrator_Address:port/NF-FG/:

The OpenFlow Domain Orchestrator receives GET request, first, it checks the user's token, once the token verifies, it checks the existence of the graph indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. Otherwise, it starts building of the correspondent graph JSON schema and exports the graphs, when we retrieved all the graphs, they are formatted as follows:

```
{
  "NF-FG": [
    {
      "forwarding-graph": {
        "name": "Test NFFG 1",
        "VNFs": [
          {
            "id": "00000001",
            "name": "nat",
            "functional-capability": "nat",
            "vnf_template": "nat",
            "ports": [
              {
                "id": "inout:0",
                "name": "data-port"
              }
            ]
          }
        ],
        "end-points": [
          {
            "id": "00000001",
            "name": "ingress",
            "type": "interface",
            "interface": {
              "if-name": "s2-eth1"
            }
          }
        ],
        "big-switch": {
          "flow-rules": [
```

```
32                {
33                  "id": "000000001",
34                  "priority": 1,
35                  "match": {
36                    "port_in": "endpoint:00000001"
37                  },
38                  "actions": [
39                    {
40                      "output_to_port":
                          "vnf:00000001:inout:0"
41                    }
42                  ]
43                }
44              ]
45            }
46          },
47          "nffg-uuid": "d8765271-d27b-41d2-8b00-7cd538ec6903"
48        },
49        {
50          "forwarding-graph": {
51            "name": "Test NFFG 2",
52            "VNFs": [..],
53            "end-points": [..],
54            "big-switch": {
55              "flow-rules": [...]
56            }
57          },
58          "nffg-uuid": "7989f810-1860-47ce-9cec-24979c99418a"
59        }
60    ]
61  }
```

Listing 6.2. List of the all the OpenFlow Domain Orchestrator deployed graphs

All the active graphs of the OpenFlow Domain Orchestrator can be updated by the Global orchestrator or GUI contacting the northbound API interface of the OpenFlow Domain Orchestrator through the PUT request. In particular, the FROG orchestrator or GUI can ask for the specific graph which is going to be updated, it corresponds to an HTTP PUT operation performed on the OpenFlow Domain

Orchestrator through northbound API. The PUT method URL is:

http://OpenFlow_Domain_Orchestrator_Address:port/NF-FG/graph-id

The OpenFlow Domain Orchestrator receives PUT request, it checks the existence of the graph with the same graph-id, indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. After the checking of the existence graph, the orchestrator starts exploring the graph JSON sent from the upper layers

- ✓ The OpenFlow Domain Orchestrator computes an operation to discover the differences between the two graphs and according to the outcome of this operation the graph is updated, preserving unchanged parts, deleting removed parts and adding new parts.

- ✓ Delete useless endpoints and flow rules, from database and Network Controller.

- ✓ Update the database.

The remaining procedure is quite the same of the one for the POST method. At this point, the updated NFFG are now saving in the database. After the successful updating of the graph, it doesn't return the graph-id because we already have it and it just returns status "202". The OpenFlow Domain Orchestrator supports for the deletion of instantiated graphs. The FROG-orchestrator or GUI contacting the northbound API interface of the OpenFlow Domain Orchestrator through the DELETE request. The GUI asks for the specific graph which is going to delete. In this case, the operation performed on the REST APIs of the orchestrator is an HTTP DELETE containing the ID of the graph that should be deleted. The DELETE request at URL is:

http://OpenFlow_Domain_Orchestrator_Address:port/NF-FG/NF-FG/graph-id

The OpenFlow Domain Orchestrator receives DELETE request, first, it checks the user's token and checks its validity, once the token verifies, it checks the existence of the graph indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. After the checking of the existence graph, it deletes the graph from the orchestrator, in the other cases, it returns an error message to specify if the graph wasn't found. After the successful deletion of the graph, it just returns status "204".

## 6.3 Open Stack Domain Orchestrator REST API

This orchestrator controls an OpenStack domain [18] . It is able to deploy service graphs according to the NF-FG used throughout the FROG architecture. In addition to the creation of NFV service chains, it allows steering traffic from an OpenStack port to another OpenStack port or to an external port and vice versa (e.g., a port that connects to the user located outside the OpenStack domain). This result is achieved by interacting with the SDN controller which in turn has to be configured as the mechanism driver of the OpenStack's Neutron module.Currently, this domain orchestrator works with OpenStack Mitaka and ONOS 1.9 as (optional) SDN controller. Support of OpenDaylight is instead deprecated.

To deploy a service graph, the domain orchestrator executes the following steps:

- Interacts with Neutron to create
  – One internal network for each link of the service graph
- Interacts with Nova in order to start the VNFs (as virtual machines)
  – Virtual machines images are store in the OpenStack image repository (i.e., Glance)
  – Nova, in turn, interacts with Neutron to create VNF ports and connect them to the proper internal network
- Interacts with the SDN controller to create links towards the external world and set up the inter-domain traffic steering
  – OpenStack is in fact not able to manage such connections, but only connections among virtual machines (i.e., intra-domain)
  - Through the SDN controller, the domain orchestrator
  • Creates the proper links towards the external world
  • Configures the network so that traffic that exits/enters the datacenter is properly encapsulated/decapsulated, according to the information associated with the graph endpoints

The REST interface that is available in the OpenStack Domain Orchestrator that interacts with the FROG-orchestrator and web GUI. In order to perform some operations needed to deploy a new graph, update an existing graph, delete an existing graph, get the list of already deployed graphs, and authenticate User. The CRUD (create/POST, read/GET, update/PUT, delete) operations are supported for NFFGs.

The main new REST APIs available are:

- POST /NF-FG/: New instantiation of a graph on the Open Stack Domain Orchestrator;
- PUT /NF-FG/nffg-id: Update an already deployed graph on the Open Stack Domain Orchestrator;
- GET /NF-FG/: Returns already instantiated graphs on the Open Stack Domain Orchestrator;
- GET /NF-FG/status/nffg-id: Returns the status of an already deployed graph on the Open Stack Domain Orchestrator;
- DELETE /NF-FG/nffg-id: Delete a deployed graph on the Open Stack Domain Orchestrator;
- GET /NF-FG/nffg-id: Get a JSON representation of a graph instantiated on the Open Stack Domain Orchestrator;
- POST /login: login, so as to identify the users who make the requests for further operations and in the response of user identification, user will get a token;

  The main operation supported by this Orchestrator is POST operation. It corresponds to an HTTP POST operation performed on the OpenStack Domain Orchestrator through northbound API, a new instantiation of NFFGs using HTTP POST method. We made some changes in the APIs; we choose to use PUT API for updating of the graphs and created new POST API for a new deploying graph. The POST method URL is:

  http://Open_Stack_Domain_Orchestrator_Address:port/NF-FG/

  The northbound API receives the POST request, created by the FROG-orchestrator or web GUI. The first operation performs by OpenStack Domain Orchestrator is, to check the user's proof of personality as a token. Once the token verifies the farther operations take place. After this operation, orchestrator validates the Network Function Forwarding Graph JSON schema; if the request does not follow the standard JSON schema, it returns the validation error message. The orchestrator starts farther exploring of the graph JSON, sent from the upper layers

- ✓ Instantiate Endpoints
- ✓ Openstack Resources Instantiation
- ✓ Instantiate Flowrules
- ✓ Update the resource description

At this stage we stored the NFFG in database, as we need this information for update, delete and get operations. NFFG ID is assigned by the OpenStack Domain Orchestrator We set the UUID as a string, for example

```
1 {
2     "nffg-uuid":"907d0e0a-5340-403e-aa85-1d2a5b70f7b4"
3 }
```

Listing 6.3. Open Stack Domain Orchestrator NFFG UUID Example

After the successful deployment of the graph, return the graph-id which is used for updating, deletion and getting the graphs. All the active graphs of the OpenStack Domain Orchestrator can be retrieved by the FROG-Orchestrator or GUI, contacting the northbound API interface through the GET method. The GUI can ask for the specific graph or all the active graphs of the OpenStack Domain Orchestrator. Two types of Getting APIs implemented which were not present in the current architecture.

- Just one active graph retrieved

  http://Open_Stack_Domain_Orchestrator_Address:port/NF-FG/graph-id:
- All active deployed graphs retrieved

  http://Open_Stack_Domain_Orchestrator_Address:port/NF-FG/ :

The OpenStack Domain Orchestrator receives GET request, first, it checks the user's token, once the token verifies, it checks the existence of the graph indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. Otherwise, it starts building of the correspondent graph JSON schema and exports the graphs, since Graph ID is deleted from the JSON of NFFG, so when we retrieved all the graphs, they are formatted as follows:

```
{
  "NF-FG": [
    {
      "nffg-uuid":
         "d1f90d99-b537-4333-84e4-eb899755cf6e",
      "forwarding-graph": {
        "name": "Forwarding graph",
        "VNFs": [
          {
            "id": "00000001",
            "name": "vnf1",
            "vnf_template": "D8YW44",
            "ports": [
              {
                "id": "inout:0"
              }
            ]
          }
        ],
        "end-points": [
          {
            "id": "00000001",
            "name": "ingress",
            "type": "interface",
            "interface": {
              "node-id": "130.192.225.182",
              "if-name": "eth1"
            }
          }
        ],
        "big-switch": {
          "flow-rules": [
            {
              "id": "1",
              "priority": 40001,
              "match": {
                "port_in": "endpoint:00000001"
```

```
37                  },
38                  "actions": [
39                    {
40                      "output_to_port":
                          "vnf:00000001:inout:0"
41                    }
42                  ]
43                },
44                {
45                  "id": "2",
46                  "priority": 40001,
47                  "match": {
48                    "port_in": "vnf:00000001:inout:0"
49                  },
50                  "actions": [
51                    {
52                      "output_to_port":
                          "endpoint:00000001"
53                    }
54                  ]
55                }
56              ]
57            }
58          }
59        },
60        {
61          "Forwarding-graph": {
62            "name": "Test NFFG 2",
63            "VNFs": [...],
64            "end-points": [...],
65            "big-switch": {
66              "flow-rules": [...]
67            }
68          },
69          "nffg-uuid":
              "ebb57b3f-2b19-450f-9d17-ccddf0a16871\n"
70        }
71    ]
```

```
72 }
```

Listing 6.4. List of the all the OpenStack Domain Orchestrator deployed graphs

The retrieved graph contains all the information of NFFG. All the active graphs of the OpenStack Domain Orchestrator can be updated by the FROG-Orchestrator or web GUI communicating the northbound API interface of the OpenStack Domain Orchestrator through the PUT request. The web GUI asks for the specific graph which is going to be updated. It corresponds to an HTTP PUT operation performed on the OpenStack Domain Orchestrator through northbound API. The PUT method URL is:

http://Open_Stack_Domain_Orchestrator_Address:port/NF-FG/graph-id

The OpenStack Domain Orchestrator receives PUT request, the operation performs by OpenStack Domain Orchestrator is, to check the user's proof of personality as a token and check its validity. Once the token verifies, it checks the existence of the graph with the same graph-id, indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. After the checking of the existence graph, orchestrator validates the NFFG JSON schema; if the request does not follow the standards JSON schema, it returns the validation error message. The orchestrator starts exploring the graph JSON sent from the upper layers. It calculates the difference between the old deployed graph and newly updated graph. The remaining procedure is quite the same of the one for the POST method. At this point, the updated NFFG are now saving in the database. After the successful updating of the graph, it doesn't return the graph-id because we already have it and it just returns status "202".

The global orchestrator supports for the deletion of instantiated graphs. The GUI contacting the northbound API interface of the OpenStack Domain Orchestrator through the DELETE request. The GUI asks for the specific graph which is going to delete. In this case, the operation performed on the REST APIs of the orchestrator is an HTTP DELETE containing the ID of the forwarding graph that should be deleted. The DELETE request at URL is:

http://Open_Stack_Domain_Orchestrator_Address:port/NF-FG/graph-id

The OpenStack Domain Orchestrator receives DELETE request, first, it checks the user's token and checks its validity. Once the token verifies, it checks the existence of the graph indicated by the GUI, and if it doesn't find the correspondent graph, responds with a 404 NOT FOUND message. After the

checking of the existence graph, it deletes the graph from the orchestrator, in the other cases, it returns an error message to specify if the graph wasn't found. After the successful deletion of the graph, it just returns status "204".

## 6.4 Deploying graph on a single domain orchestrator using new APIs

The figure 6.4 shows the typical architecture of a single domain controlled by its domain orchestrator that, in turn, is controlled by the global orchestrator. Between the global orchestrator and the domain orchestrator, there is the Double-decker bus which is responsible for the communication from the bottom to the upper part of the architecture. Communications that follow the inverse direction are done through REST APIs bypassing the Double-decker component, as shown in the figure. It is clear that the three software components are not constrained to be on the same machine; in fact, the distribution of the software modules across different machines gives scalability to the whole system.

Each domain orchestrator exports the domain information (which outlines capabilities and hardware information), to the global orchestrator. The content of that information is not needed in this use case because the graph will be instantiated on a single domain and we are not interested in neighbour domains, but the communication between the two parties itself is necessary to the global orchestrator in order to:

- Be informed about the existence of a domain orchestrator along with the domain it controls.
- Know how that domain orchestrator can be reached.

Lacking this sort of handshake or communication between the global and the domain orchestrators, the former has no information on that particular domain and it will be unable to deploy graphs on it. Network Function Forwarding Graphs need to be explaining with the "domain" tag, either in the root of the NFFG or in each element of the NFFG. In this section we are presenting the first case, where the entire graph has to be instantiated on a single domain.

Assuming that the global orchestrator has knowledge of the domain orchestrator, we can see a high-level view of the operations made by the global

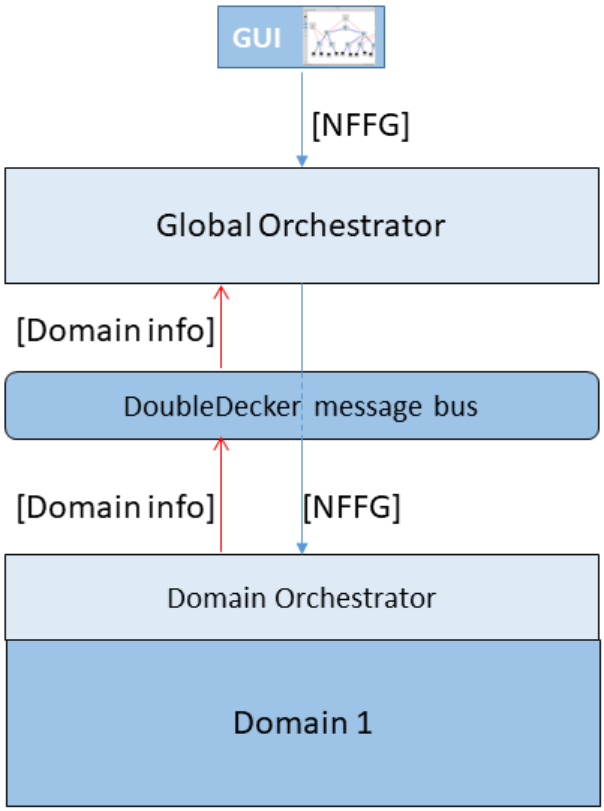


Figure 6.4. NFFG instantiation on Single domain

orchestrator when a Network Function Forwarding Graph, with the "domain" fieldset, arrives at its northbound API:

- It determines if that graph is already instantiated in the specified domain. In this case, an update operation of the previous graph is performed.
- the scheduler module checks the validity of both the domain field set in the NFFG and the related information previously stored (e.g., IP address and port of the domain orchestrator that is actually in charge of deploying the graph).
- It performs the authentication on the domain orchestrator.
- It sends the NFFG to the domain orchestrator and waits for the result.

The NF-FG sent in the final step is slightly modified compared to the original

graph in order to be fully comprehensible by every domain orchestrator; some information like the "domain" field is omitted because unnecessary beneath the global orchestrator's level. The abstraction the global orchestrator can support domains very different and heterogeneous. Once the global orchestrator has performed the above-mentioned operations, the deployment of the Network Function Forwarding Graph is a matter of the involved domain orchestrator. We can think of domain orchestrators as software modules similar to the Global orchestrator but closer to the practical aspects of the NFFG deployment. They share also the same internal architecture: in fact, Now they have a REST northbound interface, same like the global orchestrator, that support the same NF-FG and many operations are mapped one-to-one between the Global and the domain orchestrators. It is through the domain orchestrator's REST APIs that the global orchestrator performs the authentication, sends the graph and waits for a response. Then the global orchestrator notifies the result to whom has originally triggered the operation.

## 6.5 Deploying graphs on multiple infrastructure domains

Deploying a graph on multiple infrastructure domains is not an easy task. It requires a series of operations in order to establish whether this is possible. This section is intended to present steps needed to perform this operation, starting with the need to know information regarding the underlying domains, the actual graph splitting phase followed by the capabilities match between domains and, finally, the characterization of split endpoints that makes the sub graphs ready to be deployed on the chosen domains through the appropriate domain orchestrators. In the last paragraph, it is worth pointing out that those concepts are valid also in this scenario but they are not repeated here. For example, the graph deployment, deletion, update and the authentication are the same of the FROG4 Orchestrator Southbound API section 6.1, but applied multiple times if more than one domain orchestrator is concerned.

Each domain orchestrator exports information, which as described in the previous chapter, deals with exporting an abstract template describing the domain information, which outlines capabilities and hardware information. Each domain orchestrator maintains and publishes a description of its resources. The description includes the nodes/interfaces of each domain that may be used to reach other domains, including the supported technologies (e.g., GRE tunnels, VLAN). When a domain orchestrator sends the description on the message bus for the first time, the FROG orchestrator becomes aware of such domain and learns how to contact

it. Resources descriptions examples can be found in the configuration directory of each domain orchestrator repository. Particularly, it is important to set the domain orchestrator IP and port in the management-address field, to choose a domain name in the name field and to describe each interface. This information will be used by the FROG orchestrator to eventually split and deploy over multiple domains an incoming service graph.

The prerequisite of deploying a graph on multiple infrastructure domains is that such domains have to export to the orchestrator some information. This information concerns the topology and the interconnections between domains, describing capabilities of external interfaces of each domain. Capabilities include, for example, support for GRE tunnels or to VLAN tags. It is clear that only if the orchestrator has this kind of information can try to split a graph. In the absence of domains information, the orchestrator is obliged to deploy the incoming graph on a single domain and this is the scenario detailed in the previous section. For more details about the formalism of domains information and how they are exported see section *Domain abstraction* 4.2.5

### 6.5.1 NFFG Splitting

Keeping in mind the end goal to deploy a graph on multiple infrastructure domains it is important to split it into no less than two sub-graphs. This is the base operation for supporting different domains. The operation comprises of :

- ✓ VNFs assigned to that domain
- ✓ Possibly, new endpoints generated during the placement process
- ✓ Originated by links that connects VNFs/endpoints mapped to different domains
- ✓ Two endpoints originated by the same link are connected through a virtual channel
- ✓ If VNFs are assigned to two domains connected by means of a third domain
- ✓ An additional sub-graph is generated for the intermediate domain as well, This sub-graph just includes network connections and endpoints
- ✓ Deleting elements, on each sub-graph that have a place with the opposite side

✓ Obviously, if all elements are tagged with the same domain the splitting is not necessary and in this case, the entire graph will be instantiated on that domain.

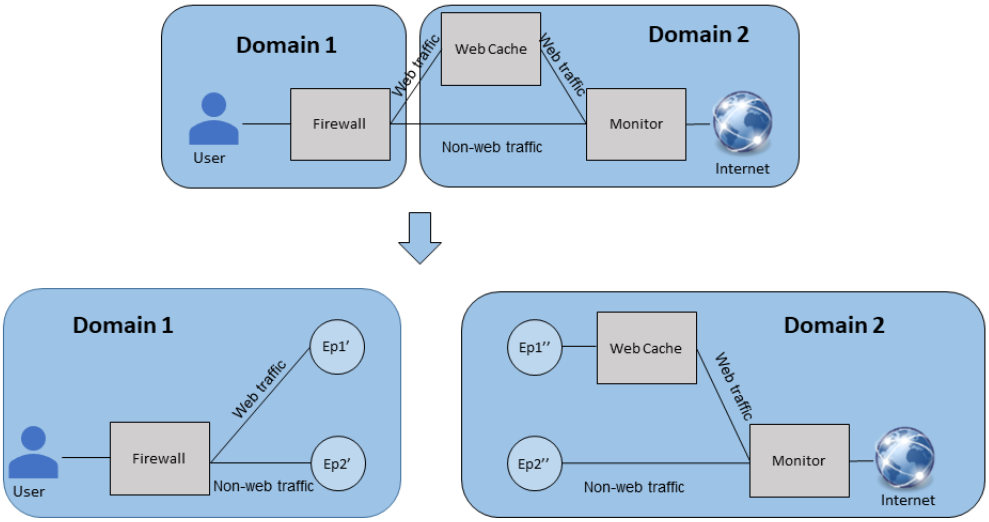


Figure 6.5. NFFG splitting on muti-domain

The split function needs to know which endpoints and VNFs have to be on one side and which ones on the other side. In this prototype we let the user annotate every VNF and endpoint with the "domain" tag that indicates where that element will be instantiated. In this way, directly from the NF-FG we have exactly what the split function needs to work: the two sets of elements that have to be on the left and on the right sub graphs after the splitting. The scheduler is the software module whose tasks are to call this function and to process the resulting graphs. A graph obtained by a split operation can be further split. Looking at the figure 6.5. Example of NF-FG splitting resulting in two couples of generated endpoints we can see, as an example, that the user's endpoint is marked with domain 1 while the other elements are marked with domain 2. Executing the split function with these inputs, the resulting graphs, with two couples of generated endpoints, are shown in the same figure. The left sub graph will be then instantiated on domain 1 while the right on domain 2, if the following phases will be successful.

# Chapter 7

# Deploying services on a real campus network

This chapter considers the design and implementation of a functions virtualization system in a Real Campus Network. It will operate over a local area network, made of commercial network equipment, and will enrich it with the possibility of running virtual machines on its edge nodes. It will turn the corporate network in a sort of "datacenter" with functions running really close to final users, thus optimizing the outgoing and incoming traffic whether compared to a solution where functions run in a centralized data-center placed somewhere in the enterprise campus.

## 7.1 Scenario

The setup we created is a simplified version of the complete scenario just described and simulates a branch office network connected to the main enterprise network, and then to the Internet, by an edge router enriched with the capability to execute the above mentioned per-user virtualized network functions. The result will be the prototype of a distributed and customizable user-oriented NFV platform for local and wide networks. The idea is to create a network which is able to recognize the user who is currently connecting to the edge router and, based on his profile, configure the network to force his traffic to traverse a given set of functions, which are dynamically instantiated on the router itself. The user will be able to configure his own functions through a management system, for example a web GUI which allows to select functions from a marketplace-like datastore and to design functions graphs, and could have multiple profiles, with different functions. In a further evolution of the system, the corporate ICT manager would be able to add also some additional

function to the ones added by the users. The actual service provided by the whole enterprise network will be the composition of a set of functions selected by different corporate entities.

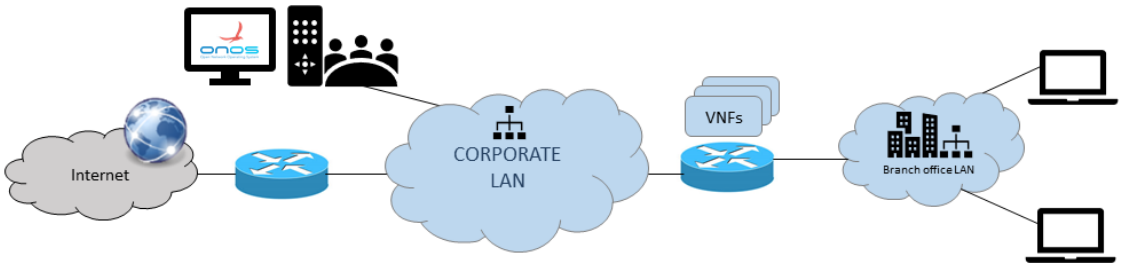


Figure 7.1. Cisco ISR router network topology setup

## 7.2 Challenges

While there are already formal definitions and some research works which try to solve similar issues regarding NFV; realizing the use case illustrated presents some new challenges and problems. First of all, it is necessary to choose carefully which kind of networking devices to use in the developing and testing phases. In fact, NFV requires hardware platforms with some precise features and technical details:

- **High-level performances**, again, virtualization is not a problem in general purpose hardware but networking devices usually have limited resources and common routers probably could not support the CPU and memory load resulting from tens of virtual machines running on them.
- **Hardware virtualization support**: it is quite obvious why this feature is needed but, while it is already a mainstream characteristic in computation hardware, it is not so common in networking devices.
- **Software virtualization support:** networking devices usually run dedicated operating systems (e.g.: Cisco IOS, etc. . .) which are optimized to do their job perfectly but they cannot be extended to do things which are commonly associated to hypervisors

Then, know that both for performances and security reasons, the system controller logic should not be placed in a device at the edge of the corporate network

but we should also find a way to get the most optimal traffic path as possible, without unnecessary delays in the delivery of packets. This involves designing proper network functions services but also avoiding packets to often transit in the controller or in another point not included in the natural path. For what concerns the software necessary to run VMs, manage Network Functions Forwarding Graphs, authenticate users and all the other things, the best way seems to be re-using existing software, open-source official products (e.g., KVM, etc. . . ) and research prototypes. This approach, thus allowing not wasting time-solving problems someone else already solved in the past, involves some efforts to integrate all these heterogeneous products in a unique coordinate system.

## 7.3 Equipment involved

All the devices involved in this test-bed are commercial systems available on the global professional network appliance market. This is very important because one of the basic goals of this thesis work is to prove that the network functions virtualization approach, still, a research topic for the moment is effectively realizable on common commercial devices and in a realistic production environment. The edge router we decided to use, after considering all bounds explained in the previous paragraph, is not exactly a common low-level router but rather a quite complex system, composed of three main sub-components:

- ✓ ***Cisco 2921 Integrated Service Router (ISR)*** with a slot for an integrated service module and many slots for additional interfaces cards [19].
- ✓ ***Cisco UCS E150S M4 integrated server*** for general purpose computing [20].
- ✓ ***Cisco enhanced high-speed WAN interface card (EHWIC)*** which provides four additional Gigabit Ethernet switched interfaces [21].

The **Cisco 2921 ISR router** is a new generation router designed to act as an access router in branch offices or small businesses. It has all the typical features of this kind of devices but, for our purposes, the most important characteristic is its expandability. In fact, we need some general purpose computing device with hardware virtualization technology where to run our virtual Functions. This has been accomplished installing the UCS integrated server into the dedicated slot. Our 2921 router has installed Cisco IOS 15.4; this is important because UCS blades are fully supported only on IOS versions equal or greater than 15.2M.

This high-density, a single-socket blade server is designed to introduce virtualization and hypervisors directly into branch office routers, so it fits perfectly our scenario requirement. Its hardware features provide good performances, even though the number of virtual functions which can be run at the same time is rather limited for a scenario where every connected user has a set of virtual machines draining resources. However, this problem can be mitigated either using more performing integrated servers or lightening virtual services, or even substituting them with more lightweight containers, as explained in the last paragraph of this chapter. The hardware specifications of the UCS server used in our testbed are the following:

- **CPU :** one Intel Xeon processor E3-1105C v2 with 4 cores, each one with a 1.00 GHz frequency rate and a 6MB cache.
- **Memory :** 32GB (four 8GB DIMM modules)
- **Interfaces :** 2 internal and 2 external Gigabit Ethernet NICs
- Hardware virtualization support

It is worth to underline the particular connection system that internally links the UCS blade with its host router because it has been exploited deeply during the design phase of the system, as explained in the next section.

At first sight it results a bit difficult to understand but, from the logical point of view, the blade and the router share two different internal links:

- A PCIexpress interface which provides an internal layer 3 Gigabit Ethernet link between the router and the E-Series Server.
- An MGF VLAN interface which provides an internal layer 2 Gigabit Ethernet link between the router and the E-Series Server

While the first one is quite intuitive to understand and use, the second one is more complicated but, at the same time, offers better configuration alternatives and flexibility. It is realized through a high-speed backplane switch module called Multi-Gigabit Fabric (MGF) [22] which provides layer 2 connections between the router CPU and all the additional modules included the UCS server and also the EHWIC interfaces expansion. In fact, this additional four ports module has been added exactly for the purpose of getting a layer 2 trunk link where connecting the branch office network with the UCS server. This is needed cause of the logic of the software part, which exploit some layer 2 functions, like DHCP and MAC learning. In the end, it is needed also a canonical server where the controller logic will be

installed. It could be any kind of commercial server with hardware virtualization. In alternative, the machines of the controller could also be separated on different servers, but it seems to be more logical to have them in the same place.

## 7.4 Implementation

Instead of creating another ad-hoc system for our use case, we decided to work with the already existing solution for NFV detailed in 4 and try to extend and configure it to cover this use case too. The software which performs almost all the logic of this scenario is based on that solution. However, it required to be customized in order to fit this scenario requirement; in fact, it had been thought to realize a single integrated node where every software module runs, while we are trying to apply this approach in a distributed local area network with real networking devices. We decided to follow this way because that system already provides some of the characteristics we needed. The most important are:

- User's authentication and dynamic functions deployment.
- Support for a large number of network function and general services.
- Possibility to concatenate different graphs.
- Compatibility with the ETSI NFV group guidelines.

## 7.5 Universal node

**Universal node** [23] usage in this architecture represents an interesting choice for legacy compatibility, as well as it allows us to reuse all the features it already implements. The Universal Node can be considered a sort of "datacenter in a box", hence providing functions similar to an OpenStack cluster, but limited to a single server. In a nutshell, it handles the orchestration of compute and network resources, hence managing the complete lifecycle of computing containers (e.g., VMs, Docker, DPDK processes) and networking primitives (e.g., OpenFlow rules, logical switching instances, etc). It receives commands through a REST API according to the Network Functions Forwarding Graph (NF-FG) formalism and takes care of implementing them on the physical node. Due to its peculiar characteristics, it can be executed either on a traditional server (e.g., workstation with Intel-based CPU) or on a resource-constrained device, such as a residential gateway.

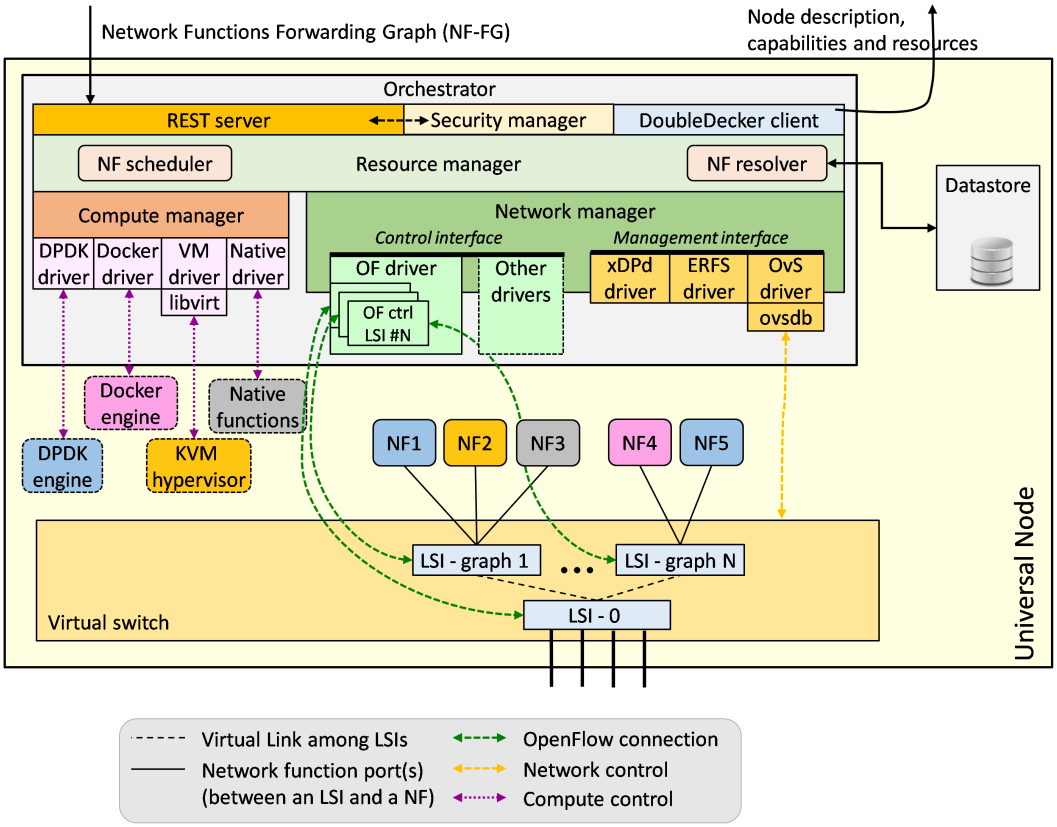


Figure 7.2. Universal Node Orchestrator architecture

More in detail, when it receives a command to deploy a new NF-FG, it does all the operations required to actually implement the requested graph:

- retrieve the most appropriate images for the selected virtual network functions (VNFs) through the datastore;
- configure the virtual switch (vSwitch) to create a new logical switching instance (LSI) and the ports required to connect it to the VNFs to be deployed;
- deploy and start the VNFs;
- Translate the rules to steer the traffic into OpenFlow flow mod messages to be sent to the vSwitch (some flow mod are sent to the new LSI, others to the LSI-0, i.e. an LSI that steers the traffic towards the proper graph.)

Similarly, the un-orchestrator takes care of updating or destroying a graph, when the proper messages are received.

A high-level overview of this software is given by the figure 7.2.

As evident in the figure 7.2 the un-orchestrator includes several modules; the most important ones are the network controller and the compute controller, which are exploited to interact respectively with the vSwitch and the hypervisor(s).

The VNF-selector selects instead the best implementation for the required VNFs, according to some parameters such as the amount of CPU and RAM available on the Universal Node, or the efficiency of the network ports supported by the VNF itself (e.g., standard virtio vs. optimized ports). Moreover, the VNF scheduler optimizes the binding VNF/CPU core(s) by taking into account information such as how a VNF interacts with the rest of the NF-FG. While the network and compute controllers are detailed in the following sections,

### 7.5.1 The network controller

The network controller is the sub-module that interacts with the vSwitch. It consists of two parts:

- **the OpenFlow controller(s)**: a new OpenFlow controller is created for each new LSI, which steers the traffic among the ports of the LSI itself;

- **The switch manager**: it creates/destroys LSIs, virtual ports, and more. In practice, it allows the un-orchestrator to interact with the vSwitch in order to perform management operations. Each virtual switch implementation (e.g., xDPd, OvS) may require a different implementation for the switch manager, according to the API exported by the vSwitch itself.

Currently, the un-orchestrator supports OpenvSwitch (OvS), the extensible DataPath daemon (xDPd) and the Ericsson Research Flow Switch (ERFS) as vSwitches. Note that, according to the figure 7.2, several LSIs may be deployed on the UN. In particular, in the boot phase, the network controller creates a first LSI (called LSI-0) that is connected to the physical interfaces and that will be connected to several other LSIs. Each one of these additional LSIs corresponds to a different NF-FG; hence, it is connected to the VNFs of such an NF-FG, and takes care of steering the traffic among them as required by the graph description. Instead, the LSI-0, being the only one connected to the physical interfaces of the UN and to all the other graphs, dispatches the traffic entering into the node to the proper graph, and properly handles the packets already processed in a graph.

### 7.5.2 The compute controller

The compute controller is the sub-module that interacts with the virtual execution environment(s) (i.e., the hypervisor) and handles the lifecycle of a Virtual Network Function (i.e., creating, updating, destroying a VNF), including the operations needed to attach VNF ports already created on the vSwitch to the VNF itself. Each execution environment may require a different implementation for the compute controller, according to the commands supported by the hypervisor itself.

Currently, the prototype supports virtual network functions as (KVM) VMs, Docker, DPDK processes and native functions, although only a subset of them can be available depending on the chosen vSwitch. Also, in this case, further execution environments can be supported through the implementation of a proper API. The un-orchestrator natively supports the deployment of NF-FGs described with initial JSON-based format defined in WP5 and used in the initial part of the project. For more detailed, the Netgroup public repository available on Github [23]

## 7.6 Integrating various components

The whole system is made of extremely different components, from commercial hardware network platform to complex open source software. Putting all together in a correctly working system has not been an easy task. The software components used have been designed for use-cases that are quite different from our scenario, so it required some ad-hoc configurations and modifications. The following paragraphs illustrate the choice made from theoretical point of view; the basic steps for setting the router and the UCS-E blade up can be found in related links in Bibliography, in particular [19] and [20]. Particularly, the interesting part of the second manual is from the beginning till the "Accessing CIMC" section, where there is explained how to access it for the first time. This could not be so easy if you have no experience with these kinds of stuffs. At this point, we are assuming that the basic configuration of the devices is already completed.

First of all the router and the UCS server had to be configured in order to force all the incoming and outgoing traffic passing into the virtual functions running on the server.

Figure 7.3 shows how we configured the router to meet our requirements. The UCSE interfaces represent the internal high-speed connection between router and server, ucse1/0 is the layer 3 interface and ucse1/1 is the layer 2 one. These interfaces can be seen from server operating system as normal Gigabit Ethernet interfaces (in our case Ubuntu names them enp3s0f0 and enp3s0f1). Since the link between branch

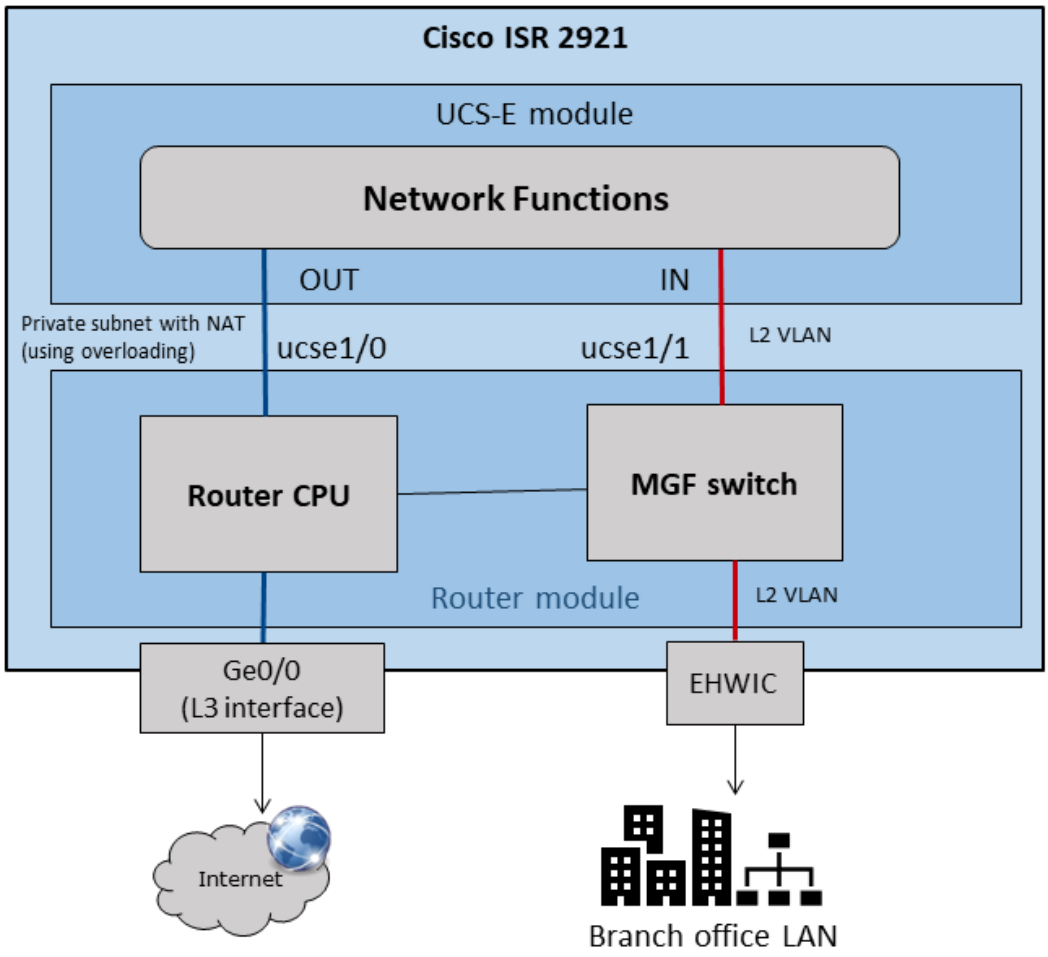


Figure 7.3. Cisco 2921 ISR Router configuration

office LAN and the server had to be a layer 2 connection, ucse1/1 has obviously been chosen to play this role. On the other side, the LAN had to be plugged to one of the EHWIC module interfaces, because these are the only ones with direct access to the MGF internal switch, letting us create a direct layer 2 "connection" between the EHWIC interfaces and the ucse1/1 port, where to force all the traffic coming from the branch office to reach the internal enp3s0f1 server interface. To achieve this behaviour, a "layer 2 VLAN" between these interfaces has been created. This VLAN spans only from EHWIC to ucse1/1 interface, which is both access ports, so 802.1Q tags are not propagated outside the router. Then, this hack is not visible to users and nor to virtual functions. With this configuration, all the traffic that reach

EHWIC interfaces is brought to the server internal enp3s0f1, and the vice versa, thus simulating a direct cable connection to the server. The link between the server and the router CPU, which is used to communicate with the corporate network and the Internet, is then the layer 3 link through ucse1/0 interface. Basically, it could be configured in two different ways:

- ✓ Public subnet even if this is the best choice for performances, it brings to a huge waste of addresses because every per-user network functions chain requires one address from this subnet.
- ✓ private subnet with NAT this solution saves a lot of addresses because NAT can be configured with overloading, thus turning it into a PAT which aggregates all the NFs chains on a single address using instead different layer 4 ports. In addition, this configuration adds a certain separation between the physical network and virtual functions, which is a great thing for security.

We added a DHCP daemon that dynamically assigns addresses from that subnet to network functions chains when they are launched, in order to increase flexibility and avoid static addresses as much as possible.

```
1  ip dhcp excluded-address 10.10.0.1
2  ip dhcp excluded-address 10.10.0.2
3  ip dhcp pool dhcp-pool
4      import all
5      network 10.10.0.0 255.255.255.0
6      default-router 10.10.0.1
7      dns-server 8.8.8.8
8
9  ip nat pool natpool 130.192.225.238 130.192.225.238
       prefix-length 25
10 ip nat inside source list 7 pool natpool overload
11 ip nat inside source static 10.10.0.2 130.192.225.242
12 ip route 0.0.0.0 0.0.0.0 GigabitEthernet0/1
       130.192.225.254
13
14 access-list 7 deny 10.10.0.2
15 access-list 7 permit 10.10.0.0 0.0.0.255
16
```

```
17 interface GigabitEthernet0/1
18     ip address 130.192.225.244 255.255.255.128
19     ip nat outside
20
21 interface ucse1/0
22     ip address 10.10.0.1 255.255.255.0
23     ip nat inside
```

Listing 7.1. Cisco 2921 ISR configuration for UCSE-Router link

Since the first case is easier to configure, we presented an example which illustrates the configuration with NAT we had in our test bed for ucse1/0. Please note that ACL denies the static management address to avoid possible conflicts between static and dynamic NAT translations. The link that connects users through the EHWIC module is needed is an address-less VLAN for letting layer 2 packets pass and each of the EHWIC Gigabit Ethernet interface we had assigned different VLAN, for different users. From the router interface ucse1/1 to MGF internal switch we had used VALN trunk.

```
1  interface ucse1/1
2      switchport mode trunk
3      no ip address
4
5  interface GigabitEthernet0/0/0
6      switchport access vlan 2
7
8  interface GigabitEthernet0/0/1
9      switchport access vlan 3
10     no ip address
11
12 interface GigabitEthernet0/0/2
13     switchport access vlan 4
14     no ip address
15
16 interface GigabitEthernet0/0/3
17     switchport access vlan 5
18     no ip address
19
20
```

```
21 interface Vlan2
22     no ip address
23     no spanning-tree vlan 2
24
25 interface Vlan3
26     no ip address
27     no spanning-tree vlan 3
28
29 interface Vlan4
30     no ip address
31     no spanning-tree vlan 4
32
33
34 interface Vlan5
35     no ip address
36     no spanning-tree vlan 5
```

Listing 7.2. Cisco 2921 ISR configuration for UCSE-EHWIC link

After configurations, modifications and installation of the Universal node orchestrator, the only thing left to configure is the Ubuntu networking configuration. Follows an example of that configuration, where the addresses used are the same of the example configuration explained in the previous page:

```
1 auto enp3s0f0
2 iface enp3s0f0 inet static
3 address 10.10.0.2
4 netmask 255.255.255.0
5 gateway 10.10.0.1
6 dns-nameservers 8.8.8.8 4.4.4.4
7
8 auto enp3s0f1
9 iface enp3s0f1 inet manual
```

Listing 7.3. Ubuntu interfaces configuration (/etc/network/interfaces)

Once we configured the router we had to deploy the NFV services in order to test and validate our work. After deployment, the user can use the NFV services. The User traffic goes through the NFV service and they can reach to the internet. Hence, we successfully configured a router which can be used as a default gateway

for a small office or a lab which could manage customized services for real users.

## 7.7 Deploying graph for LAN connection

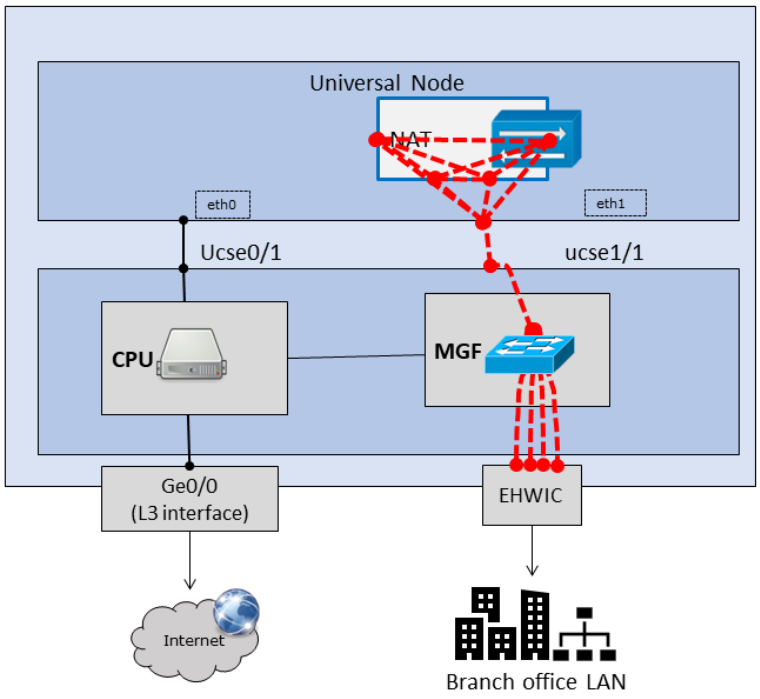


Figure 7.4. Cisco 2921 ISR, Deploying graph for LAN connection

we deployed a graph 7.4 on the one universal node using Cisco router for LAN, 8 flow rules, 4 endpoints and 1 Network Function used. When the user graph has been instantiated and all virtual Functions are running, server interface sends the traffic towards to MGF internal switch then to Brach office user interface. Of course, the response packet will follow the same path but in the opposite direction. Brach office user can now ping to each other. For example, if we connect the Lab9 network cable to one of the EHIWC interfere the others branch office User traffic goes through the NFV service and they can reach to the internet.

```
1 {
2   "forwarding-graph": {
3     "name": "Cisco test 1",
4     "VNFs": [
```

```
5         {
6           "id": "00000001",
7           "name": "nat",
8           "functional-capability": "nat",
9           "vnf_template": "3SZCDP",
10          "ports": [
11            {
12              "id": "inout:0",
13              "name": "data-port"
14            },
15            {
16              "id": "inout:1",
17              "name": "data-port"
18            },
19            {
20              "id": "inout:2",
21              "name": "data-port"
22            }
23          ]
24        }
25      ],
26      "end-points": [
27        {
28          "id": "00000001",
29          "name": "ingress",
30          "type": "vlan",
31          "vlan": {
32            "vlan-id": "2",
33            "if-name": "enp3s0f1"
34          }
35        },
36        {
37          "id": "00000002",
38          "name": "egress",
39          "type": "vlan",
40          "vlan": {
41            "vlan-id": "3",
42            "if-name": "enp3s0f1"
```

```
43            }
44          },
45          {
46            "id": "00000003",
47            "name": "egress",
48            "type": "vlan",
49            "vlan": {
50              "vlan-id": "4",
51              "if-name": "enp3s0f1"
52            }
53          }
54        ],
55        "big-switch": {
56          "flow-rules": [
57            {
58              "id": "00000001",
59              "priority": 1,
60              "match": {
61                "port_in": "endpoint:00000001"
62              },
63              "actions": [
64                {
65                  "output_to_port": "vnf:00000001:inout:0"
66                }
67              ]
68            },
69            {
70              "id": "00000002",
71              "priority": 2,
72              "match": {
73                "port_in": "vnf:00000001:inout:0"
74              },
75              "actions": [
76                {
77                  "output_to_port": "endpoint:00000001"
78                }
79              ]
80            },
```

```
81          {
82            "id": "00000003",
83            "priority": 3,
84            "match": {
85              "port_in": "endpoint:00000002"
86            },
87            "actions": [
88              {
89                "output_to_port": "vnf:00000001:inout:1"
90              }
91            ]
92          },
93          {
94            "id": "00000004",
95            "priority": 4,
96            "match": {
97              "port_in": "vnf:00000001:inout:1"
98            },
99            "actions": [
100             {
101               "output_to_port": "endpoint:00000002"
102             }
103           ]
104         },
105         {
106           "id": "00000005",
107           "priority": 5,
108           "match": {
109             "port_in": "endpoint:00000003"
110           },
111           "actions": [
112             {
113               "output_to_port": "vnf:00000001:inout:2"
114             }
115           ]
116         },
117         {
118           "id": "00000006",
```

```
119             "priority": 6,
120             "match": {
121               "port_in": "vnf:00000001:inout:2"
122             },
123             "actions": [
124               {
125                 "output_to_port": "endpoint:00000003"
126               }
127             ]
128           }
129         ]
130       }
131     }
132 }
```

Listing 7.4. Deploying NFFG for LAN connection

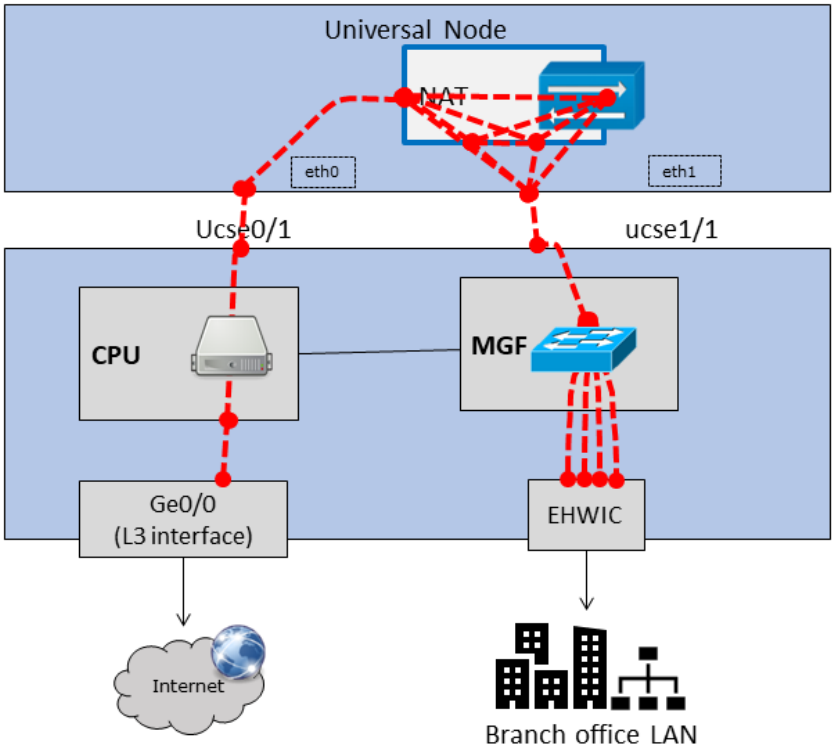


Figure 7.5. Cisco 2921 ISR, Deploying graph for WAN connection

## 7.8 Deploying graph for WAN connection

We deployed another graph on the one universal node using Cisco router for WAN. 10 flow rules, 5 endpoints and 1 Network Function used, when the user or network operator graph has been instantiated and all virtual Functions are running, server interface sends the traffic towards the CPU and then to internet port. Another interface of the server sends the traffic towards to MGF internal switch which forwards it to the EHWIC. Of course, the response packet will follow the same path but in the opposite direction. The User traffic goes through the NFV service, router CPU, and they can reach to the internet as shown in figure 7.5.

```
{
  "forwarding-graph": {
    "name": "Cisco tesing graph 2",
    "VNFs": [
      {
        "id": "00000001",
        "name": "switch",
        "functional-capability": "switch",
        "vnf_template": "3SZCDP",
        "ports": [
          {
            "id": "inout:0",
            "name": "data-port"
          },
          {
            "id": "inout:1",
            "name": "data-port"
          },
          {
            "id": "inout:2",
            "name": "data-port"
          },
          {
            "id": "inout:3",
            "name": "data-port"
          },
          {
```

```
28              "id": "inout:4",
29              "name": "data-port"
30            }
31          ]
32        }
33      ],
34      "end-points": [
35        {
36          "id": "00000001",
37          "name": "egress",
38          "type": "interface",
39          "interface": {
40            "if-name": "enp3s0f0"
41          }
42        },
43        {
44          "id": "00000002",
45          "name": "ingress",
46          "type": "vlan",
47          "vlan": {
48            "vlan-id": "2",
49            "if-name": "enp3s0f1"
50          }
51        },
52        {
53          "id": "00000003",
54          "name": "ingress",
55          "type": "vlan",
56          "vlan": {
57            "vlan-id": "3",
58            "if-name": "enp3s0f1"
59          }
60        },
61        {
62          "id": "00000004",
63          "name": "ingress",
64          "type": "vlan",
65          "vlan": {
```

```
66            "vlan-id": "4",
67            "if-name": "enp3s0f1"
68          }
69        },
70        {
71          "id": "00000005",
72          "name": "ingress",
73          "type": "vlan",
74          "vlan": {
75            "vlan-id": "5",
76            "if-name": "enp3s0f1"
77          }
78        }
79      ],
80      "big-switch": {
81        "flow-rules": [
82          {
83            "id": "00000001",
84            "priority": 1,
85            "match": {
86              "port_in": "vnf:00000001:inout:0"
87            },
88            "actions": [
89              {
90                "output_to_port": "endpoint:00000001"
91              }
92            ]
93          },
94          {
95            "id": "00000002",
96            "priority": 1,
97            "match": {
98              "port_in": "endpoint:00000001"
99            },
100           "actions": [
101             {
102               "output_to_port": "vnf:00000001:inout:0"
103             }
```

```
104             ]
105           },
106           {
107             "id": "00000003",
108             "priority": 1,
109             "match": {
110               "port_in": "endpoint:00000002"
111             },
112             "actions": [
113               {
114                 "output_to_port": "vnf:00000001:inout:1"
115               }
116             ]
117           },
118           {
119             "id": "00000004",
120             "priority": 1,
121             "match": {
122               "port_in": "vnf:00000001:inout:1"
123             },
124             "actions": [
125               {
126                 "output_to_port": "endpoint:00000002"
127               }
128             ]
129           },
130           {
131             "id": "00000005",
132             "priority": 1,
133             "match": {
134               "port_in": "vnf:00000001:inout:2"
135             },
136             "actions": [
137               {
138                 "output_to_port": "endpoint:00000003"
139               }
140             ]
141           },
```

```
142         {
143           "id": "00000006",
144           "priority": 1,
145           "match": {
146             "port_in": "endpoint:00000003"
147           },
148           "actions": [
149             {
150               "output_to_port": "vnf:00000001:inout:2"
151             }
152           ]
153         },
154         {
155           "id": "00000007",
156           "priority": 1,
157           "match": {
158             "port_in": "vnf:00000001:inout:3"
159           },
160           "actions": [
161             {
162               "output_to_port": "endpoint:00000004"
163             }
164           ]
165         },
166         {
167           "id": "00000008",
168           "priority": 1,
169           "match": {
170             "port_in": "endpoint:00000004"
171           },
172           "actions": [
173             {
174               "output_to_port": "vnf:00000001:inout:3"
175             }
176           ]
177         },
178         {
179           "id": "00000009",
```

```
180             "priority": 1,
181             "match": {
182               "port_in": "vnf:00000001:inout:4"
183             },
184             "actions": [
185               {
186                 "output_to_port": "endpoint:00000005"
187               }
188             ]
189           },
190           {
191             "id": "000000010",
192             "priority": 1,
193             "match": {
194               "port_in": "endpoint:00000005"
195             },
196             "actions": [
197               {
198                 "output_to_port": "vnf:00000001:inout:4"
199               }
200             ]
```

Listing 7.5. Deploying NFFG for WAN connection

# Chapter 8

# Results Validation

In previous chapters, we proposed a way to deploy generic virtual services on different networking scenarios. Solutions provided seem to accomplish well to their duty but, in order to verify if they can really fit real-world production use-cases, a performance evaluation phase is necessary. We decided to focus on the aspects which are more interesting in a user-oriented service: deployment time, updating time, latency and throughput.

## 8.1 Hardware platform

The testing phase has been done on the FROG, which have powerful Cisco UCS E150S M4 integrated server. Since in the LAN scenario a similar server has been used, it is evident that conclusions obtained after these performance tests can be easily extended to that scenario as well. The server has the following hardware configuration:

- Two Intel Xeon E5-2430L CPU with 6 cores and 12 threads each
- 15MB L3, 250KB L2 and 32KB L1 caches
- Intel Hyper-Threading and Virtualization technologies
- 32GB RAM DDR3 memory

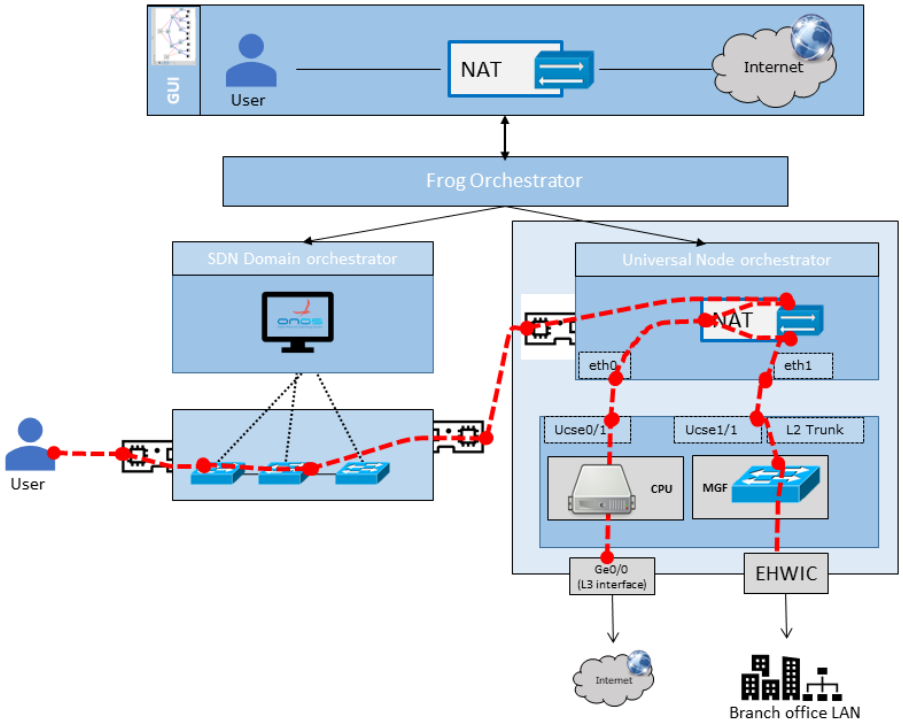


Figure 8.1. Scenario used for architecture validation.

## 8.2 Graph instantiation time

To test the deployment time of a user's graph and to analyze how much time is taken from every software component of the solution, many graphs have been deployed, measure the time taken for the deployment in different situations. It is very important to take these measures because we have built a system which dynamically deploys users' graphs when needed, so a long deployment time would mean a long wait for the user before being able to use required services.

We start to measure the time taken from the four test graphs, in order to discover the influence on performances of the number of virtual functions and also the time consumed from the different sub-modules composing our solution. This analysis could be useful in the future, in an eventual performances optimization phase. The **FORG** was evaluated by measuring the amount of time spent to deploy several kinds of NFFGs on a *Cisco router*. Being a system, which dynamically instantiates services, the most important evaluation parameter was the reactivity of this operation. The time taken to deploy a services chain is the time the user has to wait before being able to use it. For this purpose we performed various tests as discussed

below:

| Test Type | Test#1 | Test#2 | Test # 3 | | | | Test # 4 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | UN_LA | UN_WA | Global_Orc | UN_1_C | UN_2 | Total | Global_Orc | Universal_C | SDN | Total |
| Deployment[sec] | 3.21 | 3.47 | 5.63 | 2.37 | 2.46 | 5.63 | 7.23 | 2.43 | 3.90 | 7.23 |

Figure 8.2. Deployment time for different graphs

- Test# 1: In this test, we deployed a graph on the one universal node (the one on UN_LA) using Cisco router for LAN, 8 flow rules, 4 endpoints and 1 Network Function are used.
- Test# 2: In the second test we deployed a graph on the one universal node using Cisco router for WAN. 10 flow rules, 5 endpoints and 1 Network Function are used. The deploy time is more than test because it has more flows.
- Test# 3: We deployed a graph on Global orchestrator, it split the graph and sent it to the two universal nodes (the left sub-graph on UN_1_C, this node inside the Cisco router and the right one on UN_2, is in the corporate LAN).we used 8 flow rules 3 endpoints and 2 Network Functions.
- Test# 4: Is the same of the test #3 except we replaced the second node on the SDN domain (which in the corporate LAN). We used 6 flow rules, 3 endpoints and 1 Network Function on the main NFFG for the Global orchestrator. Total time is the same for the Global orchestrator because once the graph is spilt it sends that to the one domain and after receiving the ACK then it sends the other part of the graph to another domain and waits for the ACK. So, global orchestrator shows the total time of the graph deployment.

All these tests measure the time between when the command is received from the global orchestrator on its northbound API and when all resources are actually deployed. Results obtained are coherent with what we expected, at first Global orchestrator have to prepare data, then infrastructure controllers REST APIs are called to deploy resources. As usually, REST APIs returns immediately but effective deployment times are very different; in fact, if Universal Node flows installation is almost immediate, the OpenFlow domain orchestrator is the slowest because it has to deal with the actual network controller. The orchestrator returns only when all NFV operations are finished.

## 8.3 Graph update time

After measuring the time needed to instantiate an entire graph from scratch, it is interesting to analyze the time needed to update a graph. Of course, it changes depending on how much and what type of resources have to be updated, so we established many different tests to verify the most common case

- All tests, a virtual function is substituted by another one.

As usual, start time is taken at the moment the instantiation request is received and end time is taken when the last resource is correctly deployed. All tests are done with the deferent graphs of figure 8.3, as usual.

| Test Type | Test#1 | Test#2 | Test # 3 | | | | Test # 4 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | UN_LA | UN_WA | Global_Orc | UN_1_C | UN_2 | Total | Global_Orc | Universal_C | SDN | Total |
| Updating[sec] | 1.52 | 1.64 | 4.78 | 1.58 | 2.4 | 4.78 | 6.57 | 1.56 | 4.21 | 6.57 |

Figure 8.3. Graph Update time for different graphs

## 8.4 Latency and throughput

The third phase of performance evaluation has been dedicated to analyzing network performances of our graphs. The test considers completely different situations. The first test considers from one branch office user to same branch office user or to different Brach office user. As for the second test, it examines form the branch office of all users to the internet or to same branch office users. For example, now the branch office user can use the NFV services can easily reach to the internet. The third one regards form the user of the UN_2 (which is in the corporate LAN) to the UN_1_C (that is inside the router) and then goes to the internet. Of course, the response packet will follow the same path but in the opposite direction. The last test is identical to the third one, but we only replace UN_2 with the SDN domain.

In the last phase, the throughput is measured with the tool iperf. The user acts as an iperf client and sends traffic to an iperf server on the internet side, the traffic is using TCP. We measure the bit rate of transmitting on the client, and the bit rate of receiving on the server.

The goal of the validation phase was to prove that our solution can be really used in the real world and to understand if further work on this prototype could be

| Test Type | Test#1 | Test#2 | | Test # 3 | | | | Test # 4 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | UN_LA | UN_WA | | Global_Orc | UN_1_C | UN_2 | Total | Global_Orc | Universal_C | SDN | |
| Latency [ms] | 0.714 | 0.714 | 3.48 | 1.443 | | | | 0.944 | | | |
| Throughput[Mbit/s] | 94.5 | 94.5 | | 94.0 | | | | 92.2 | | | |

Figure 8.4. Latency and throughput of different test

useful. From this point of view, tests gave encouraging results and underlined which aspects are critical in order to improve performances. Despite that, the deployment time for different graphs is intolerable for the user. If we consider that our VMs are not optimized for performances at all (they can be lightened or even substituted with other lighter containers, like Dockers), then, from this test, we can immediately learn a lesson, in a future hypothetical production environment. Results obtained from the second test are pretty coherent with what we were expecting. The time consumed by the Global orchestrator is linked to code execution time, database operations and waiting for ACK from the domains orchestrator and it's because to know that our NFV service is correctly deployed. Tests on the update functions gave a good result for user's access point change and that alone confirmed its utility. The benefit of updating could have been greater with more complex graphs.

# Chapter 9

# Conclusions and future works

The thesis work turned out to be more interesting and challenging of what had been envisioned at the beginning, because of the power of the technologies we used and the many problems and choices that came out during this way. The limitations and impossibility to solve easily some uses cases, forced us to think to more various and creative solutions to keep the project as coherent as possible to the initial idea.

The Global orchestrator corresponds to the first two levels of the orchestration layer, and consists of a technology dependent part and a technology independent part; we replace a technology dependent part with southbound API. We have no more technology dependent part. The technology independent part receives the user login requests and NFFGs created by the web GUI or service layer, through the northbound API. It manipulates the requests and gives them to the southbound API; which sends the resulting NFFGs to the proper infrastructure domains. It is worth noting that our architecture consists of a single Global orchestrator that sits on top of multiple infrastructure domains, even implemented with different technologies e.g. the Universal Node, the OpenStack domain, and the OpenFlow domain. The REST interface that is available to each component of FROG through an HTTP server interacts with the web GUI in order to perform all the operations needed to deploy a new graph, update an existing graph, delete an existing graph, get the list of already deployed graphs and provide a security layer to authenticate the user. Previously, each component of the FROG was problematic, having issues because of the usage of different APIs. One of the main problems was the lake of coherency and consistency among the different components. This problem was preventing FROG to perform as a one unit having all its component aware of each other's and communicating each other for the end results. It was also causing duplication of Network Function Forwarding Graphs (FFGs). Another problem with the existing architecture was that of the deadlock. Different APIs standards and because of that,

the poor communication among the components lead to inefficiency and deadlocks. There was also the security problem for the FROG. By implementing new standard well defined REST APIs for each of components of the FROG, most of the problems with the existing architecture have been solved. In the new design, all the APIs for different components now follow the same usage and design pattern hence the complexities have been removed by adopting a common pattern. A new security layer was added to the Global orchestrator which works as security manager by taking care of the user authentication by using a token-based authentication. This kind of security solution avoids a continuous exchange of username and password and hence improves security. The new design is much more efficient and usable as compared to the older one. The CRUD (create/POST, read/GET, update, delete) operations can be performed very easily and efficiently on the NFFGs. The different patterns are listed below.

- POST /NF-FG/: New instantiation of an NFFG on each component of the FROG;
- PUT /NF-FG/nffg-id: Update an already deployed NFFG on each component of the FROG;
- GET /NF-FG/: Returns an already deployed NFFGs on each component of the FROG;
- GET /NF-FG/status/nffg-id: Returns the status of an instantiated NFFG on each component of the FROG;
- DELETE /NF-FG/nffg-id: Delete an already deployed NFFG on each component of the FROG;
- GET /NF-FG/nffg-id: Get a JSON representation of a graph on each component of the FROG;
- POST /login/: login; to identify the users who make the requests for further operations and in the response of user authentication, user will get a token.

After defining the standard pattern of the APIs, we implemented this pattern in each component of the FROG e.g. the CPE, the OpenStack domain, the OpenFlow domain, the Global orchestrator, Web GUI, and DataStore. This was a very challenging task because these components have been developed using different technologies. We needed to understand all of these components first and then update the code for our changes. Another big challenge was an integration of changes. As

all the components of FROG are interrelated, they are somehow dependent on each other, so making changes to one component used to disturb many other components and hence we needed to track all the necessary changes in the other components and implement them as well.

The newly implemented global orchestrator functionalities are create/POST, read/GET, update, delete. The FG formalism previously used has been modified in order to support not covered use cases and a multi-domain instantiation of services. The main contribution of this work lies in the ability of splitting graphs and the subsequent capabilities match phase that supports a variety of possible interconnections between the involved domains, starting from the case of domains directly connected and supporting also domains not directly in contact. This has been possible thanks to the Big-Switch approach that provides to the global orchestrator data useful to take adequate decisions in every situation. Also implemented new functionality when the splitting NFFG is going to update, the Global orchestrator computes an operation to discover the differences between the two graphs and according to the outcome of this operation the graph is updated, preserving unchanged parts, deleting removed parts and adding new parts. For example, if the updated graph contains the information for the same domain and also addressed to a different domain compared to the graph already instantiated. The graph will be split, the updated sub graph is sent to the same domain orchestrator of the existing graph using PUT method. The new graph is instantiated on the new domain by means of the appropriate domain orchestrator, but using POST request and finally, the old graph is deleted from the old domain through the concerned domain orchestrator. In the conclusion, under the update splitting graph request there are PUT, POST, and DELETE operations take place. From the northbound API of GUI, we created two interfaces one is used by the Global orchestrator and domains orchestrators, which supports all the CRUD operations for NFFGs. Another interface is used for the data store which managed the backend of the NFFGs. In order to validate our prototype, tests have been carried out also on a campus network and this has confirmed the potentialities of this solution. In particular, the test concerning three different domains has been encouraging because it has put successfully together this work and other works that play different roles in the whole architecture but, at the same time, need to be in close contact to reach an efficient outcome. We tested the multi-domain orchestration capability involved end users equipments. It was a great result implemented on the Universal Node domain orchestrator inside a user's home gateway and then orchestrating services based on the user's preferences.

Finally, I worked on the Cisco 2921 Integrated Service Router (which features computing blades and switched network ports). I started working on this which was

a Black Box for me, and then successfully configured the UCS blade and Router. After configuration of the router, I worked on the installation and moved Universal node orchestrator into the router. I performed all of these configurations and operations through the very complex console port of the UCS blade and router. Once we configured the router we had to deploy the NFV services in order to test and validate our work. After deployment, the user can use the NFV services, The User traffic goes through the NFV service and they can reach to the Internet. Hence, we successfully configured a router which can be used as a default gateway for a small office or a lab which could manage customized services for real users.

As a plan for the future, we foresee that some component of this system can be improved in order to fit a production scenario, for example introducing virtual machines live migration during the graph update. Moreover, we are planning to extend the compatibility of the domain orchestrator's, in order to support dynamically domain orchestrator's resources exportations and validations and Network Function to Network Function links in SDN Domain Orchestrator.

# Bibliography


[1] *Computer Networks Group (NetGroup) - Polytechnic University of Turin*. URL: http://netgroup.polito.it/.

[2] SDN and OpenFlow World Congress. «Network Functions Virtualization white paper». In: Oct. 2012. URL: http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[3] *Open Networking Foundation website*. URL: https://www.opennetworking.org.

[4] *OpenWRT project website*. 2015. URL: https://openwrt.org/.

[5] *OpenvSwitch project website*. 2015. URL: http://openvswitch.org/.

[6] NFV ETSI Industry Specification. «Network Functions Virtualization; Architectural framework». In: Oct. 2013. URL: http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf.

[7] Ersue Mehmet. *ETSI NFV Management and Orchestration - An Overview*. URL: http://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf.

[8] *DoubleDecker public git repository*. URL: https://github.com/Acreo/DoubleDecker.

[9] *UNIFY website*. URL: https://www.fp7-unify.eu.

[10] Mignini Fabio. «User-oriented Network Service on a Multi-domain Infrastructure». Politecnico di Torino, Dec. 2014.

[11] *OpenConfig working group website*. URL: http://www.openconfig.net.

[12] *FROG4 - Overarching orchestrator submodule - Public git repository*. URL: https://github.com/netgroup-polito/frog4-orchestrator/.

[13] *Graphical User Interface for the FROG4 orchestration - Public git repository*. URL: https://github.com/netgroup-polito/frog4-gui.

[14] *Datastore - Public git repository.* URL: https://github.com/netgroup-polito/frog4-datastore.

[15] *Main Public git repository of FROG v.4, a cloud/NFV orchestrator supporting heterogeneous infrastructure.* URL: https://github.com/netgroup-polito/frog4.

[16] *SDN Domain Orchestrator Public git repository.* URL: https://github.com/netgroup-polito/frog4-sdn-do.

[17] *Universally unique identifier wikipedia website.* URL: https://en.wikipedia.org/wiki/Universally_unique_identifier.

[18] *OpenStack Domain Orchestrator Public git repository.* URL: https://github.com/netgroup-polito/frog4-openstack-do/.

[19] *Cisco ISR 2921 details and documentation.* URL: https://www.cisco.com/c/en/us/products/routers/2921-integrated-services-router-isr/index.html.

[20] *Cisco UCS E-series Getting Started Guide.* URL: http://www.cisco.com/c/en/us/td/docs/unified_computing/ucs/e/1-0/gs/guide/b_Getting_Started_Guide/b_Getting_Started_Guide_chapter_010.html.

[21] *Cisco Gigabit Ethernet EHWIC description.* URL: http://www.cisco.com/c/en/us/products/collateral/routers/3900-series-integrated-services-routers-isr/data_sheet_c78-612808.html.

[22] *Cisco ISR G2 Multi Gigabit Fabric documentation.* URL: http://www.cisco.com/c/en/us/td/docs/routers/access/interfaces/software/feature/guide/mgfcfg.html.

[23] *Universal Node Orchestrator public repository.* URL: https://github.com/netgroup-polito/un-orchestrator.