POLITECNICO DI TORINO

III Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

User-oriented Network Service on a Multi-domain Infrastructure

Relatore
prof. Fulvio Risso

Candidato:
Fabio Mignini

Dicembre 2014

In this laboratory, strange things happen.
All the guys of LAB 9

Acknowledgements

There are a lot of people who deserve to be named. First of all, all the guys in "Lab 9", especially Ivano Cerrato and Alex Palesandro, each of them has helped me in these eight months of thesis, and from whom I learned a lot. Thank you to Matteo Tiengo, for staying up late at night with me in the laboratory, and thank you to the annoying alarm of "Politecnico di Torino", for reminding us the ending of the canonical hours. Thank you to my university supervisor Fulvio Risso for all the precious advice and continuous support.

Thank you to Andrea and Corrado, without whom I would never have come to Torino. Thank you to my cousin Catia, that supported me in difficult moments in a city far from home. Thank you to my parents, who gave me this fantastic opportunity.

I would thank all my friends in my hometown Cupra Marittima, that every time I come home they make me feel as if I never left. I would also thank all people I met during the years in Torino.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The way network services are delivered has dramatically changed in the last few years thanks to the arise of the Network Functions Virtualization (NFV) paradigm, which allows network services to experiment the same degree of flexibility and agility already available in the cloud computing world. In fact, NFV proposes to transform the network functions that today run on dedicated appliances (e.g., firewall, WAN accelerator) into a set of software images that can be consolidated into high-volume standard servers, hence replacing those middleboxes with the so called Virtual Network Functions (VNFs).

NFV is currently being seen as a technology targeting network operators, which can exploit the power of the IT virtualization (e.g., cloud and datacenters) to deliver network services with unprecedented agility and efficiency. However, end users have never been taken into consideration as active players in the NFV domain; furthermore, NFV is limited to the datacenter, leaving most of the telecom operator network out of the picture.

This thesis tries to invert this trend by presenting a solution that is oriented to deliver *generic* network services, selected by *multiple actors*, which allows the *dynamic* instantiation of *per-user* network services. In future, this can be used on the large infrastructure of the telecom operators, possibly starting from the home gateway installed in the customer premises till the data center, as depicted in Figure 1.1.

Our solution enables several actors (e.g., network providers, end users such as the xDSL customers, etc.) to define their preferred network services; moreover, is it general enough that services include both the traditional middlebox functions considered in NFV as well as traditional host-based network services. For example, the network service selected by a customer can include a deep packet inspection (DPI) VNF that handles HTTP traffic, an email scanner to analyze and clean his electronic correspondence, and a Bittorrent client that downloads the objects specified by the user without interfering with the rest of the traffic. In our solution, the entire network infrastructure is controlled by a service logic that performs the identification of the user that is connecting to the network itself. Upon a successful identification, the proper set of network functions chosen by the user is instantiated in one of the nodes (possibly, even the home gateway) available on the provider network, and the physical infrastructure is configured to deliver the user traffic to the above set of VNFs.

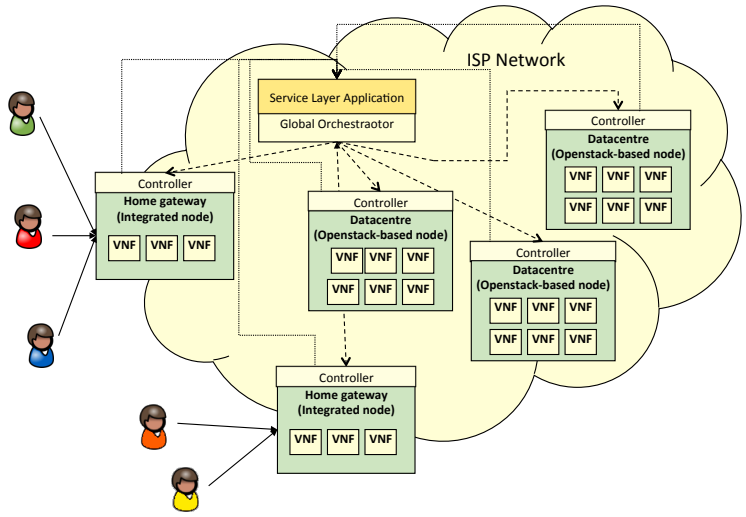


Figure 1.1: Deployment of virtual network functions on the wide telecom provider network.

More in detail, the thesis describes the service-oriented layered architecture to achieve those objectives, a possible set of data models that are used to describe and implement the requested network services (starting from an high-level and user-friendly view of the service, which is then converted into a set of primitives (e.g.,

virtual machines, virtual links) that are actually used to instantiate the service on the physical infrastructure), and two possible implementations of the nodes of the infrastructure layer on which the service is actually deployed.

Particularly, we explored two solutions for the nodes hosting the VNFs, which are based on different technologies and have different requirements in terms of hardware resources. The first proposal is based on the OpenStack open-source framework, and it is more appropriate to be integrated in (existing) cloud environments; the second one, instead, exploits mostly dedicated software, and it is oriented either to demanding environments (e.g., high performance network services) or to the embedded segment (resource-constrained CPE). This second infrastructure layer have been implemented by other netgroup members and is presented in this thesis only for validate the higher layers of the architecture.

The reminder of this thesis is structured as follows. Chapter 2 details the reasons that led us to adopt an NFV and SDN approach in our architecture. Chapter 2 provides an overview of the related works, in particular the ETSI NFV, while Chapter 4 gives a detailed overview of OpenStack and OpenDaylight (an open-source SDN controller), which are used in our architecture. Chapter 5 introduces an architecture to deploy general network services on the wide provider network, while Chapter 6 details some formalisms expressing the service to be deployed, which are then exploited in the use case discussed in the first section of Chapter 8. Chapter 8 details the preliminary implementation of the architecture, which is then validated in Chapter 9, both in terms of functionalities and performance. Finally, Chapter 10 concludes the thesis and provides some plans for the future.

Chapter 2

Problem

The past 30 years have been marked by the networking principle of "distributed intelligence". Switches and routers basically decide independently where they forward packets and what information they exchange with neighboring devices. This approach has proven to be very stable, but also increasingly sluggish. For instance, the need of change in network configuration requires the reconfiguration of all devices, which is very time consuming. New services such as cloud computing and the increasing mobility provide 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 or load balancing. This network functions should be able to be automatically moved from one data center to another. The necessary configuration changes must be made automatically. Therefore the large cloud providers like Google and Amazon decided to push towards a Software-Defined Networking approach. This choice makes network definitely more agile than in the past.

2.1 Current networks conception problems

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 simplicity. Therefore, an hardware-centric networking approach leads to slower innovation. Baking the software into silicon lengthens production cycles and reduces the number of features you can incorporate into the system. Worse, once baked in, the hardware cannot be easily modified. Firmware only softens this compromise, not really changing the underlying choice.

2.2 SDN and NFV approach

Although software is infinitely flexible, but slower than hardware. Multicore processing is gradually narrowing the gap in performance. Moreover, new software development practices, virtualization and open standards have made software much more modular, flexible and easy to develop.

Today, computer networks are experiencing the biggest upheaval since the 70' when they was born. Networks should not be configured at the device level, but as a whole. This makes it possible to build networks that are more agile, flexible, robust and secure than today's networks. To achieve this, a nontrivial architecture that exploit the SDN and NFV paradigm come into play, in this thesis we are going to analyze both ETSI and our architecture proposal. Either of them introduce fundamental components in network architectures that go beyond the traditional conception of network world. These fundamental components are cloud-computing and orchestrator. The first one is needed because virtual network functions (VNFs) are taking place of network functions coded in hardware, so the compute is in charge to manage them. VNF means a software version of a network function (e.g. dhcp, firewall, nat) so we can find VNF in the form processes running in a virtual machine, a linux container or in the form of simple process. The orchestrator component instead allows a network administrator to provide a service on the network, without having to deal with each individual architectural device, taking care about steer the right input the right module of the architecture.

Potentially this new approach opens up new opportunities and above all, as mentioned before, gives an agility to the network unthinkable heretofore. For example,

the network can dynamically react to an attack or can dynamically respond to a congested link, duplicating network function and inserting a load balancer.

2.3 Technological transition

On the path leading effective implementation of these services there are a lot of challenges to be faced; for instance, where to place the computing resources and on the basis of which criterion decide where to instantiate the virtual network functions. There is not a simple and unique solution to these problems, for the first problem for example, google are bringing on users' home a gigabit fiber-based connection, thus reducing the latency and bandwidth problems, adopting as a solution to keep the computing resources in the network core, taking so geographically distant users and services. This solution is not thinkable for a country like Italy with a population highly distributed in a highly irregular area. Hence, another type of approach is to bring some computing resources till user's home and trying to bring the network functions required by users as close as possible to them.

Another significant problem is to be able to make as easy and inexpensive as possible for service providers the transition between the old and the new concept of networks. This can be done only trying to take advantage of the general purpuse software tools, which companies already use on general purpose hadware, to provide this new level of service. It is in this context that OpenStack comes.

Chapter 3

State of the art

In the following of this chapter a brief description of the ETSI NFV main concepts and architecture will be provided in order to better understand the problem and the implemented solutions proposed and discussed in Chapter 5 and Chapter 6 of this thesis.

3.1 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 an European Standards Organization. In November 2012 seven of the world's leading telecoms network operators selected the ETSI to be the home of the Industry Specification Group (ISG) for Network Function Virtualization (NFV). Now, two 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.

3.1.1 Objectives

From a high level view, the objectives that the ETSI NFV group [5] are:

- Improve capital efficiencies, if comparing NFV's 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, etc. 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.
- 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 shout 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.

3.1.2 High-level NFV framework

Network Functions Virtualization envisages the implementation of NFs as software-only entities that run over the NFV Infrastructure (NFVI). Figure 3.1 provides an

high-level view of the NFV framework. As evident, three main working domains are identified in network function virtualization:

- **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 centered around 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.

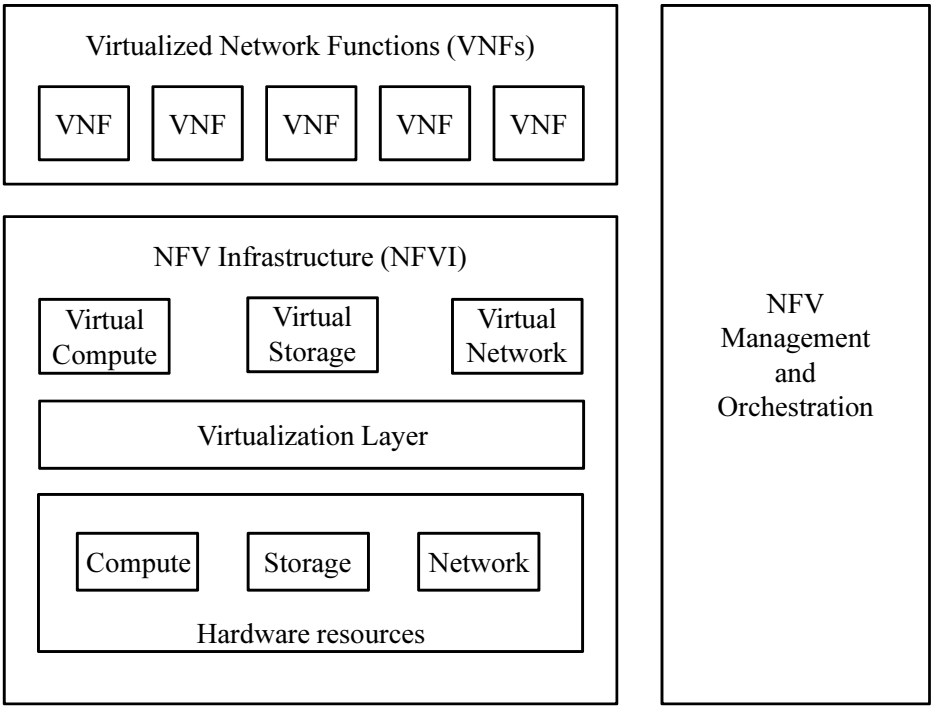


Figure 3.1: High-level NFV framework architecture.

- 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)

3.1.3 Network services in NFV

An end-to-end network service (e.g., mobile voice/data, Internet access, a virtual private network) 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 underlaying physical infrastructure (NFVI). It consists in an end-to-end network 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.

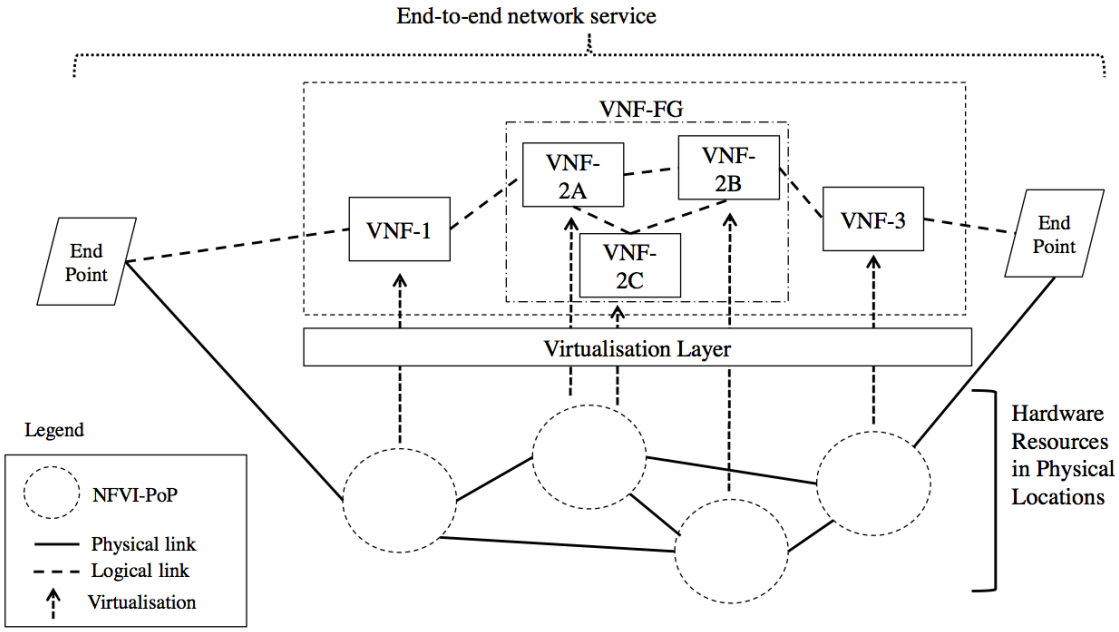


Figure 3.2: End-to-end network service with VNFs and nested forwarding graphs example.

3.1.4 Architecture of NFV

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).

Figure 3.3 shows the NFV architectural framework depicting the functional blocks and reference points in the NFV framework. 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 owner of the network.

3.1.4.1 Functional blocks

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

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

For the sake of clarity, a view of a functional block is given in Figure 3.4. 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,

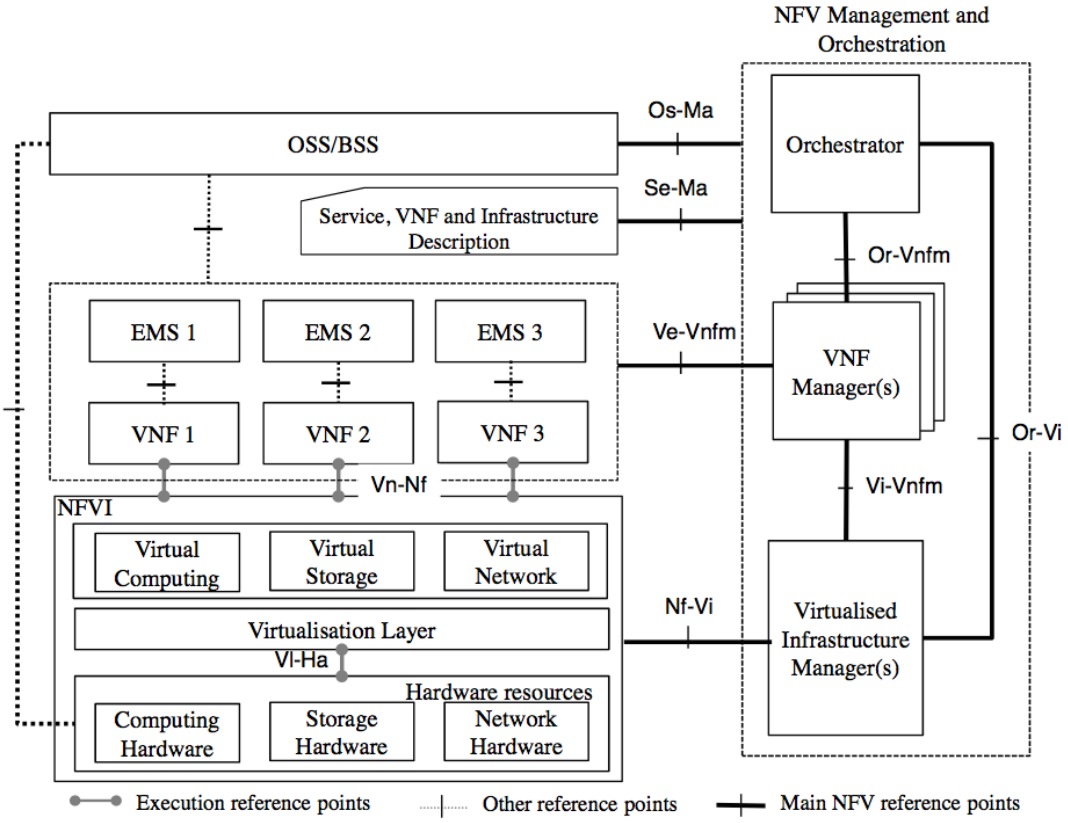


Figure 3.3: Detailed NFV framework architecture.

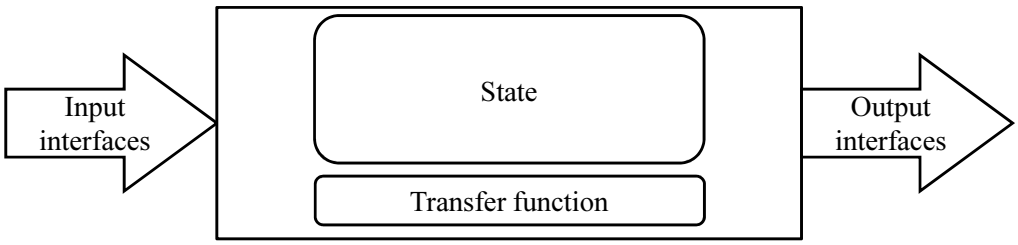


Figure 3.4: The fundamentals of a functional block.

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 labeled 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.

3.1.5 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 Componet (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 initiated/deployed and connected.

A **VNF Descriptor (VNFD)** is a deployment template which describes the way a VNF has to be deployed and its operational behavior requirements. It is primarily used by the VNF Manager during the process of instantiation and life-cycle 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 to establish 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 kind 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 separated and 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[8] 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 torward a fully virtualized environment.

Chapter 4

Backgroud

4.1 Openstack

4.1.1 Overview

OpenStack[13] is an open-source cloud toolkit consisting of a series of interrelated projects that control pools of processing, storage and networking resources in a datacenter. This project, started in 2010, aims to behave as a datacenter operating system or cloud OS. As a traditional operating system, a cloud operating system has the main purpose to export an abstraction of the physical hardware but, instead of piloting directly the integrated circuits via drivers as in the single server case, it has the ability to interface itself with an arbitrary number of agents resident over the hypervisor - or virtual machine monitor (VMM) - of each physical server's operating system, in other words it behaves as a manager of hypervisors. Cloud operating systems simplify the management of a datacenter in which virtual machines are heavily utilized. In fact, they provide a level of abstraction in which all the physical servers are seen and can be managed from a single point.

The OpenStack project aims to provide a centralized interface to be used by datacenter administrators, giving them an overview about resources availability, usage and status and therefore putting them in the position to plan maintenance or provision hardware resizing in order to keep up with the demanded computational

power. Being a commercial product and not only a proof of concept, OpenStack is also engineered to allow the coexistence of administrators and normal users, the firsts are in charge of monitoring the resources availability and manage the infrastructure of the datacenter, providing to the seconds the possibility to instantiate virtual machines and interconnecting them without worrying about configuring the physical infrastructure and the actually available resources. Normal user is a general term that will refer to an actor that does not have administrative privileges (i.e., he does not have full control over the system); in the case of OpenStack, a user can correspond to either a single person, a department in a company or even to an entire firm that decides to rely on an external infrastructure in where to deploy the virtual servers that provide the services needed for business. Of course this automation and level of abstraction come at a price which is affordable without any problem in a datacenter environment; on the other hand, in the case of very small amount of physical machines, the overhead is not negligible even if not exaggerate, as can be seen in table 4.1 (note that the data are taken when the system is at operating speed and many of the OpenStack components works with a caching system). Let's say that in general, when the number of equipments exceeds the few units and the virtual machines creation and deletion volume is high enough to overload the amount of requests that administrators can keep up with, the effort to install the OpenStack system will be well rewarded.

4.1.2 General design architecture

OpenStack has been developed having in mind five guidelines, which are:

- Component based architecture, to quickly add new behaviors.
- Highly capable of easily scale up/down.
- Fault tolerant, isolated processes avoid cascading failures.
- Recoverable, failures should be easy to diagnose, debug, and rectify.
- Open standards, be a reference implementation for a community-driven API.

As mentioned above, OpenStack is not a single project but a pool of independent modules that communicate together, each one in charge of providing one or more functionalities (e.g., computing and network management, authentication, etc). Modularity has two main benefits: the first one is to permit datacenter administrators to choose what features they want and what they do not, enabling the possibility to offer users different typologies of service; the second advantage is the interchangeability of modules that are designed for the same purpose (e.g., network management) so that an administrator can pick the module that suits the best or even write one by himself from scratch. The majority of modules are, in turn, split into submodules and designed to allow plugging in special purpose add-ons written to accomplish specific tasks or made the module behave in a certain desired way.

4.1.3 Physical architecture

The deployment of OpenStack requires a total amount of three logically separated entities that can be remapped into one or more physical machines. These three entities are generally referred as the following:

- **Controller node**.
- **Compute node**.
- **Network node**.

In the **Controller node** resides a set of supporting services, basic (or core) components and a series of optional features. Supporting services are not properly OpenStack components but are necessary for its operations; two of these components are the database management system (e.g., MySQL[9]) for persistent storage and the message broker (e.g., RabbitMQ[15]) for modules intercommunication. Having them in the controller node is not mandatory but since the major share of requests to this components comes from services also resident in the control node, delays due to the network are eliminated. Basic components are all those OpenStack modules that provide the very basic functionalities and that will become the "brain" in charge of managing the whole infrastructure. In addition, in the controller node can

be installed optional components that are not crucial but offer useful functionalities such as Ceilometer. Ceilometer is the OpenStack telemetry service that tracks resources usage aggregated by user, giving the possibility to manage billing in case of a datacenter offered as a service. Another optional component is the orchestrator known as Heat that is in charge of communicating with both the network and the compute manager. The controller node needs to communicate directly with all the other nodes (via a management interface) to dispatch commands and receive updates from them as depicted in Figure 4.2 and further detailed in Figure 4.1.

The **Compute node** is in charge of actually hosting the virtual machines. This node is a single physical server and, if we consider the controller node as the brain of OpenStack, this compute node can be seen as the muscles. Since all the control services are hosted in the controller node, the amount of software required to be installed in the compute node is minimal, saving computing resources for the virtual instances that will be deployed on it. The only pieces of code required are an agent that communicates with the local hypervisor and - optionally - another agent that takes care of accepting particular commands to provide traffic isolation between users. In addition, in case of usage of optional services that require a distributed agent to retrieve statistics, also these agents run in the compute node. Compute nodes are connected to the controller node in order to receive instructions, and have a separated network interface, referenced as `"instance tunnel"` in Figure 4.1, to communicate with each other, permitting to users' VMs to exchange traffic. An edge of this network of Compute nodes there is the Network node.

The **Network node** is the border between the OpenStack domain and the outside world; in particular, is in charge of being the only node that has the capability to connect to the external network and let VMs traffic in and out to the Internet. It is also connected to the Controller node to receive commands. VMs are not instantiated in this frontier component, but it runs agents that will be in charge of providing basilar network services such as network address translation (NAT), ip assignation (DHCP) and privacy policies via a firewall.

In the classic architecture of an OpenStack domain, each compute node have two virtual switches that can either be linux bridges, openvswitches[14] or xDPd[17]: one

	RAM [MB]
Controller node - full installation	2100
Controller node - minimal installation	1500
Compute node	590
Network node	250

Table 4.1: OpenStack nodes - resource analysis.

for attaching the VMs (generally called br-int) and the other (referred as br-net or br-tun) to create virtual tunnels with the other compute nodes, establishing a full mesh network that will carry packets from one server to another, thus making the intermediate network infrastructure completely transparent. The Network node, in addition to these two bridges, has another one (usually referred as br-ex) to which is bridged the physical network interface card (NIC) that has access to the Internet. As said before, the network node does not host virtual instances but instead it hosts network services; these services are attached to the br-int. The br-int and the br-ex in the Network node are not linked in any way so that neither traffic can leave the inside of the datacenter, nor outside packets can enter the private network, unless a specific service (a router) is deployed and linked with two virtual interfaces, one connected to each of the two software switches.

Logical schemes for better understanding the complex architecture is provided in Figure 4.2, 4.1 and 4.3, more details of the depicted services are available in section 4.1.4.

The logical architecture just described, depending on the size and computing power of the datacenter, can be remapped in various ways; in particular, the three entities can be aggregated as the administrator likes at installation time: a common deployment is to devolve one server serving as controller node and one as a network node, plus a number of compute nodes. However, in case of a very small datacenter in which there are just some physical machines nodes can be collapsed, even, OpenStack can be used to manage just one server; in this particular case one single machine will have all the three entities at the same time. Evidently the last example is a critical situation in which the introduced overhead is really influencing the performance, therefore this kind of configuration is useful only for developing and testing purposes.

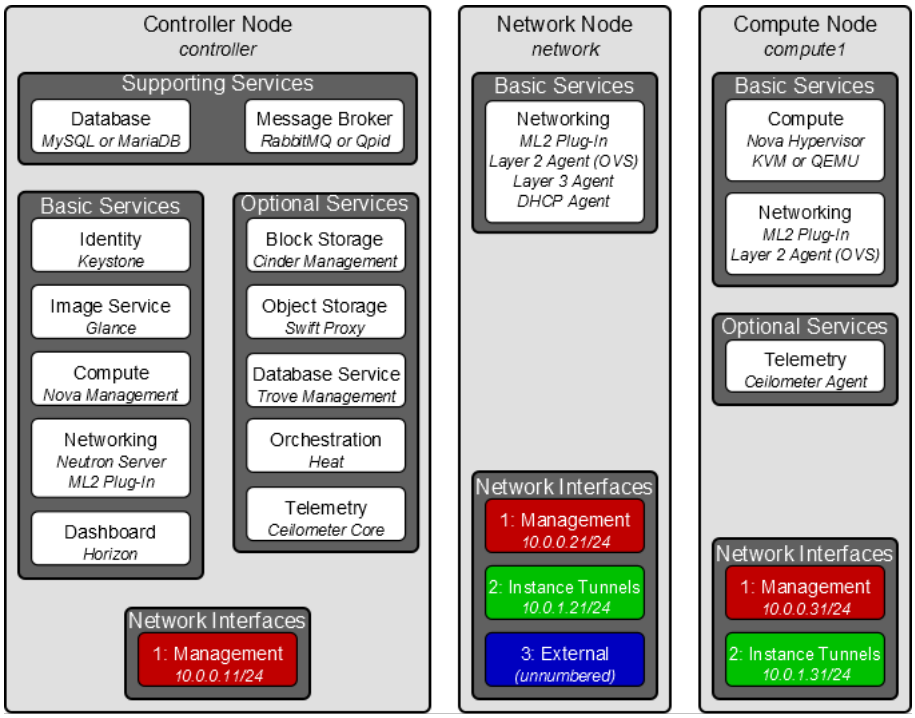


Figure 4.1: OpenStack logical nodes - services and interfaces.

4.1.4 Main components

OpenStack modules (called services in the follows) communicate together either via RESTful API or via a message broker server and may have the need to query a database. Additionally, in case of a datacenter as a service provided to external users, OpenStack offers a web interface - the Horizon dashboard - that requires a web server in order to serve users' requests. These three additional services are not part of the core components of OpenStack but, based on the standard configuration, they are necessary and essential for the system to work properly. As visible in Figure 4.4, which represents the main OpenStack components and their interactions, the services are clearly separated by functionality. From the image, it is also evident that two components are more involved in communications compared to the others. These two components are the previously cited (optional) dashboard and the identity service, which provides authentication to both services and users.

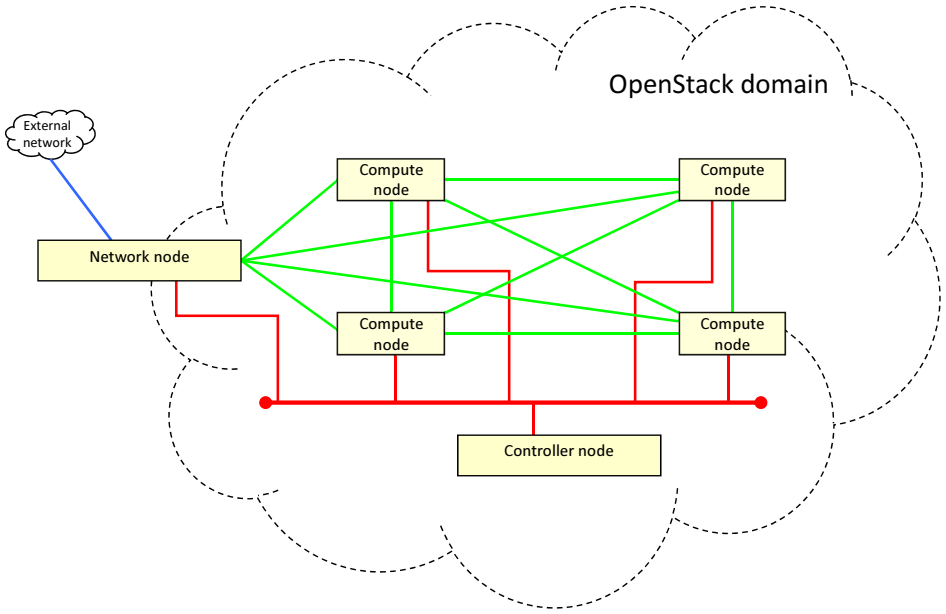


Figure 4.2: OpenStack domain - Nodes interconnections.

4.1.4.1 Identity - Keystone

As a distributed environment, the system has to provide secure access and hence both users and services must authenticate themselves before performing any operation. OpenStack developers wrote a specific service called **Keystone** for this purpose; it stores username and password for each authorized user/service, the authorization process involves obtaining a token to be used later to perform requests to other services. The general concept behind a token-based authentication system is simple: it allows users to enter their username and password in order to obtain a token, which enables them to fetch a specific resource - without using their username and password again. Once the token has been obtained, the user can use the token, which offers access to a specific resource for a time period. Identity also correlate a user to its role in the system, and the visibility of resources. Administrators are able to see everything that is going on in the system, while a common user can only see the virtual resources he created. This separation of group belonging in keystone

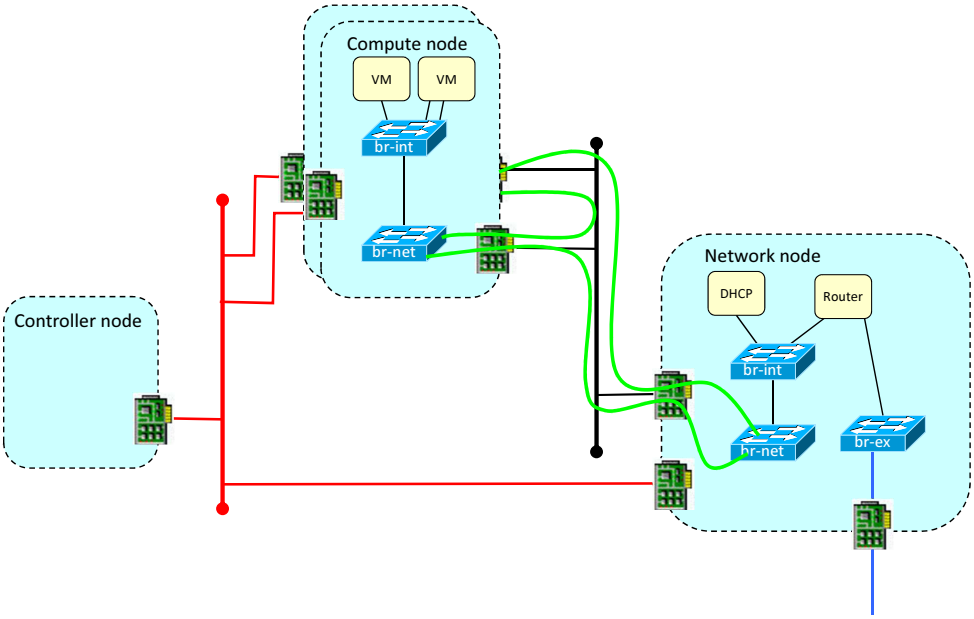


Figure 4.3: OpenStack domain - Nodes interconnections in detail.

is called role and is used by other services to check whether the user is authorized or not to perform such operation, basically defining a sandbox for each user so that anyone can only see what he is authorized to. Alongside the user concept, keystone defines three fundamental terms which are group, tenant (or project) and domain; in this way rules can be associated to them instead of directly to a user. These terms are remapped to internal data-types; here below a list of Keystone resources is given alongside with a brief description and usage:

- User: contains account credentials and is associated with one or more projects or domains.
- Group: a collection of users; it is associated with one or more projects or domains.
- Project (Tenant): unit of ownership in OpenStack; it contains one or more users part of a Group.

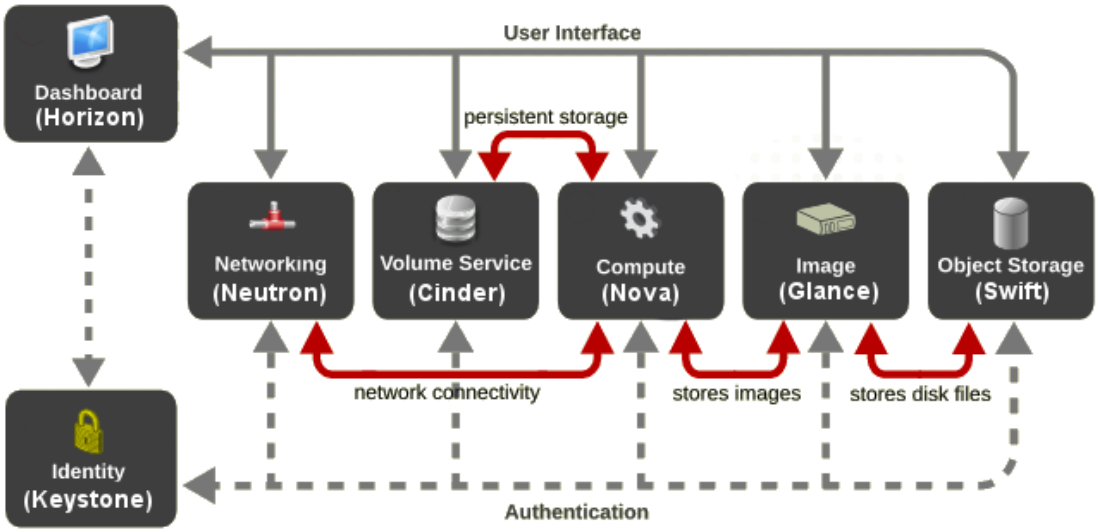


Figure 4.4: Internal components interaction.

- Domain: contains users, groups and projects.
- Role: a piece of metadata associated with many user-project pairs.
- Token: identify credential related to a user or to a user-project pair.
- Extras: bucket of key-value metadata associated with a user-project pair; it is intended to provide a method to send extra fields in a REST call without changing its interface.
- Rule: describes a set of requirements for performing an action.

Since credentials are generally built from the user metadata in the `"extras"` field of the Identity API, adding a `"role"` to a user just means to add the role to the metadata. Data-types are used inside keystone to provide a set of services that are exposed to users via one or more HTTP endpoints:

- Identity: this service is intended to provide authentication, credential validation and data about Users, Groups, Projects, Domains and Roles, as well as any associated metadata. In the basic case all these data is managed directly

by the service itself that exposes all the CRUD methods associated with the data.

- Token: service that validates and manages tokens that are used when performing requests, typically to other services; a token is created once a user credentials have been verified.
- Catalog: which provides a registry used for endpoint discovery, this permits to know how to contact all the other services.
- Policy: this service provides a rule-based authorization engine and the associated rule management interface.

Policies based on a set of rules can be specified for each OpenStack component; all components have their own *"policy.json"* file that permits to associate rules to all Openstack service actions. In listing 4.1 is an example of policy.json file of Compute service (Nova) in which is defined - lines 14 and 15 - a policy that only permits to the user identified as the owner or an administrator to start or stop a VM, meaning that someone that shares a project with the previous user is not allowed to manage a VM not belonging to him.

```
1  {
2          "context_is_admin":  "role:admin",
3          "admin_or_owner":  "is_admin:True␣or␣
               user_id:%(user_id)s",
4          "default": "rule:admin_or_owner",
5
6          "cells_scheduler_filter:TargetCellFilter":
               "is_admin:True",
7
8          "compute:create": "",
9          "compute:create:attach_network": "",
10         "compute:create:attach_volume": "",
11         "compute:create:forced_host": "is_admin:True",
12         "compute:get_all": "",
```

```
13          "compute:get_all_tenants": "",
14          "compute:start": "rule:admin_or_owner",
15          "compute:stop": "rule:admin_or_owner",
16          "compute:unlock_override": "rule:admin_api"
17  }
```

Listing 4.1: policy.json file relative to the Nova service.

4.1.4.2 Storage - Cinder, Swift and Glance

Numerous projects have been started focusing on different specific purposes related to disk management.

Block storage - known as Cinder - allows to dynamically attach additional disks to virtual machine instances. This ability allows to scale up the available disk size of a specific virtual machine avoiding the need to destroy and recreate the same instance only with more storage space.

Object storage - codename Swift - is meant to manage a distributed, API-accessible storage platform that can be integrated directly into applications or used for backup, archiving and data retention. This comes particularly handy in the typical scenario of a datacenter where the computing power is kept separated from the large hard drives where data reside. Swift provides similar functionalities that a network-attached storage (NAS) does.

Unlike the two above cited services that are not identified as OpenStack core components, the **image service** (Glance) is very important and provides the ability to retrieve, copy and snapshot a server image and store it leveraging the others storage services or interacting directly with the local file system. Images can be uploaded in various formats and pinned as public or stay private, meaning that they can be used only by the user that created them. A service that takes care to allow customers to manage their own base images and create snapshots is crucial to provide a flexible customizable experience to end users. Glance has a RESTful API that allows querying for snapshot's metadata as well as retrieval of the actual image. This function is crucial in the moment of instantiation of a virtual machine since at startup time a bootable disk is essential. Associating information to disk image

is also trivial at instantiation time when resources are reserved and assigned to a virtual machine. In this moment is essential to check if the given resource quotas are enough to support the chosen image and therefore make the VM work correctly.

4.1.4.3 Computing - Nova

The compute part is one of the critical services of OpenStack, being the one that actually makes virtual machine instances run. It is designed to manage and automate pools of compute resources and can work with widely available virtualization technologies, as well as baremetal and high-performance computing (HPC) configurations. Given its complexity, this component, in turn, has been divided into submodules, each one in charge of one particular task. The two components of particu-

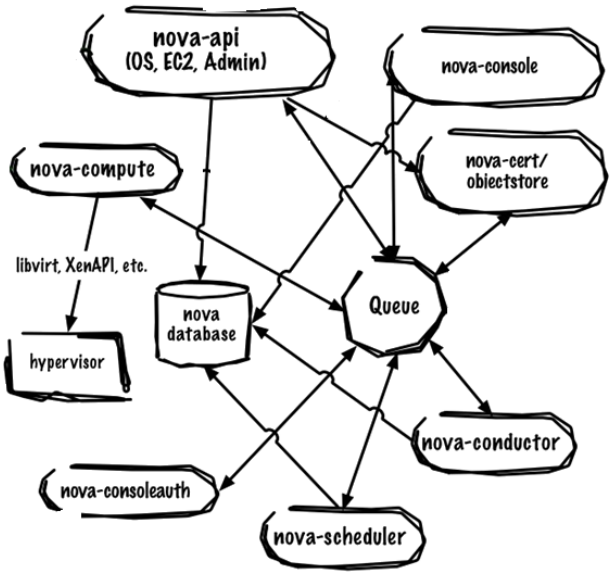


Figure 4.5: Nova components and interactions.

lar interest are the compute and the scheduler submodules. The former is a worker daemon called **nova-compute** and resides in each compute node. It is in charge of creating and terminating virtual instances through hypervisor APIs, for example XenAPI for XenServer/XCP, VMwareAPI for VMware, libvirt for KVM and others as depicted in Figure 4.6. Given that KVM and XenServer are popular choices for

hypervisor technology and recommended for most use cases, OpenStack also wants to meet the needs of those administrators that do not have at their disposal a large computational power and might want to use a linux container technology such as LXC, and wish to minimize virtualization overhead and achieve greater efficiency and performance. In addition to different hypervisors, OpenStack supports Intel, ARM and other hardware architectures. The nova-compute agent is also in charge of letting the other nova submodules located in the controller node aware of its existence and status, this job is done by sending periodical messages to the message broker service. In the controller node are located both the module that manages

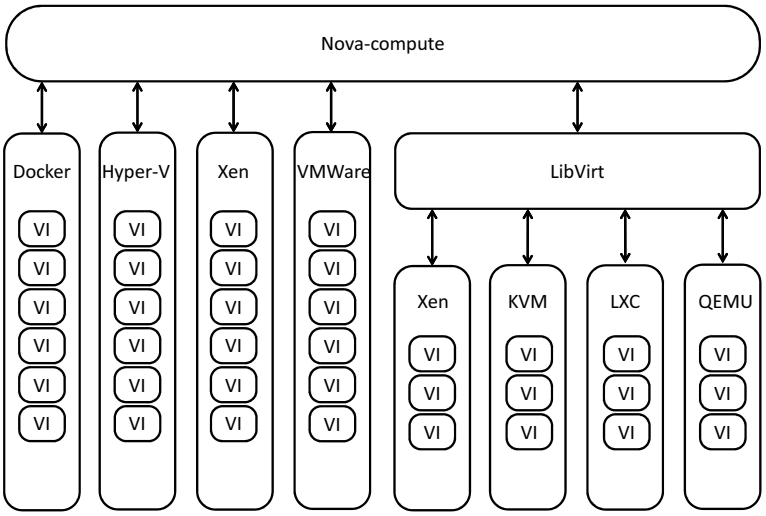


Figure 4.6: Nova compute hypervisors support.

requests (**nova-api**) and the **nova-scheduler**, which is in charge of choosing where the virtual machines are going to be instantiated. The scheduler submodule is very important and valuable for an administrator that wants to maximize the efficiency of the entire system. As default, OpenStack comes with an algorithm that takes decisions in a two-steps process called filter and weight, a representation of this sequence of events is given in Figure 4.7. In the first step, it excludes from all the

available hosts, the ones that do not meet specific requirements (e.g., free resources, architecture, hypervisor, etc), while in the second step all feasible hosts remained after the filtering are weighed according to a predefined cost function. The filter manager module is explicitly thought in a way that new constraints can be defined to further skim the pickable nova-compute module. On the contrary of the filter module, the weighing function is customizable but unique so it has to be tuned with consciousness; its only purpose is to model each remaining host with a number, then the host with the highest value is chosen and the command to instantiate the virtual machine is dispatched to the corresponding nova-compute agent. Both filter and weight steps take into consideration the information - generally referred as host status - exported by each nova-compute that typically consists in free resource availability.

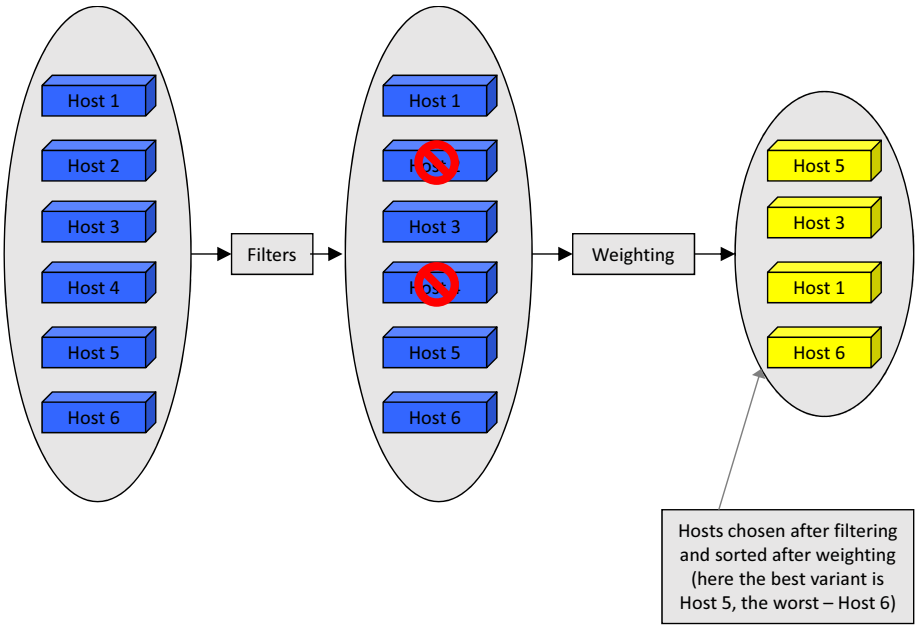


Figure 4.7: Filter and weight scheduling process.

4.1.4.4 Network - Neutron

As well as Nova, **Neutron** is also a core component of OpenStack and it is in charge of managing the network that interconnects the virtual machines. Actually, it would

be more correct to say that it is in charge of managing the virtual overlay network that interconnects the virtual machines as it does not have any knowledge about the physical interconnections between compute nodes. Neutron, formerly known as Quantum, provides users an abstraction that allow them to interconnect their own virtual machines and define some basic network services such as router, dynamic host configuration protocol (DHCP), firewall and virtual private network (VPN) terminator, decide which IP address assign to a VM and choose whether a VM is reachable from the Internet or not. When a user decides to modify his own network topology or performs operations such as launching or stopping a virtual machine, Neutron quickly reconfigures the virtual switches - br-int, br-tun/br-net - to provide connectivity and isolation. In order to understand the level of abstraction provided by this component, it is necessary to look at the network at three different levels. First comes the lower layer, composed of the physical equipments such as Ethernet cables, optical fibers and hardware switches; on top of this, a full mesh of generic routing encapsulation (GRE) tunnels are established between all the br-tun/br-net virtual switches infrastructure explained in section 4.1.3. Over this layer lays the actual user defined networks, modeled with virtual local area network (VLAN) and carrying the traffic through the GRE tunnels. This behavior is modifiable by intervening on the configuration files of Neutron and therefore adapting this service to better suits the datacenter communication infrastructure. Being this flexible requires Neutron to have a structure as the one in Figure 4.8, where five levels are indicated. The first two from the top are in charge of receiving requests, serialize and transform them into objects that models Neutron resources. The user is provided with a set of primitive to create his own network topology; these primitives are nothing less than Neutron resources that are:

- Network - defines a L2 broadcast domain.
- Subnet - in a network, it defines an IP address range
- Port - in a subnet, it defines an attachment point for either a VM, a router or a firewall.

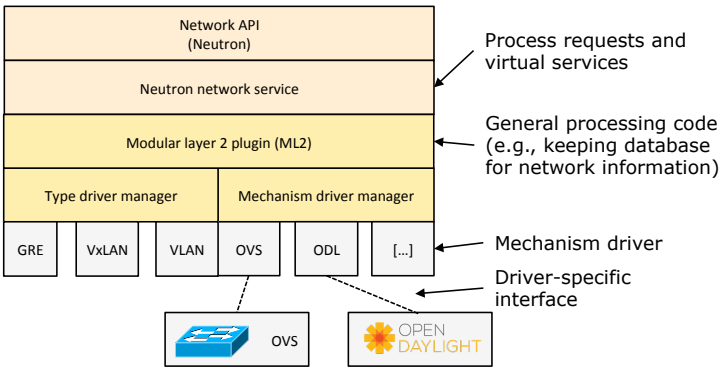


Figure 4.8: Neutron architecture.

Scrolling down the architectural stack (shown in figure 4.8), there is the Modular Layer 2 plug-in (ML2), which is in charge of keeping the internal status of the network service to provide robustness and recoverability after a major failure of the neutron component. It also fulfill the `"GET"` requests that are useful for example to the dashboard to represent in a graphical manner all the information relative to the user's network topology. Below the ML2 there are two managers layer that are in charge of dispatching requests to type and mechanism driver modules. Both the drivers kind are intended to provide and implement per user and - if required - per user's network isolation. Let's say that the type driver decides the standard that is going to be used and the mechanism is in charge of implementing it. Type drivers are pluggable modules in which information about how to guarantee isolation are added to the Neutron objects. The modified object is then returned to the third level of the stack that finally passes it to the other manager which is in charge of dispatching the received object to one or more drivers. A mechanism driver is specifically designed to communicate with a particular virtual switch that resides in the compute nodes. This component is crucial and each vendor can easily develop one that remaps Neutron objects (listed in 4.1.4.4) to commands specific to its device; OpenStack comes with already a lot of drivers from which to choose; for example the one for openvswitch, linuxbridge and the one developed expressly for the integration with

OpenDaylight which will be described in section 4.2. Mechanism drivers are actually called twice each time a resource is created, updated or deleted: before the ML2 database operation and then again after that the data-base transaction is concluded successfully. This double call allows to the mechanism driver to prepare and check the feasibility of the operation and only after it the grant from the upper layer, the operation is actually performed. To summarize, Neutron other than providing connectivity between VMs, also establishes user isolation in a way that inter-user communication can happen only via a public network defined by the administrator for this specific purpose.

4.1.4.5 Orchestration - Heat

Heat is the main project in the OpenStack Orchestration program. It implements an engine to launch multiple composite cloud applications based on templates in the form of text files that can be treated like code. It aims to export a unique-common API usable by the user to instantiate a complete network topology comprehensive of virtual machines in the form of a template making just one call; the Heat engine then takes care of translating such template into calls for both Nova and Neutron. In order to improve performance, all these calls will be made in parallel unless dependencies between resources are implicitly or explicitly specified. As an example of implicit dependency let's consider neutron resources listed in 4.1.4.4: a Neutron port attached to a network can exist only if that specific network has already been created. On the other hand, as explicit dependence the user can point out that a virtual machine must be created before another one - a valid motif can be that the first VM provides services without which the second one cannot boot correctly. Another useful feature is the management of updates. When a request is received by Heat and it contains a reference to an already instantiated template, the engine is able to understand what is changed between the allocated resources and the ones present in the request - this is done similarly to the `"diff"` unix/linux command. Once this task has come to an end, the creation, modification and/or deletion of resources will be handled; the new ones will be created and the no more needed can be deleted. For what concerns the update of a resource, Heat allows two ways of

behavior. The first one is to rawly destroy the elder and then create a new one with the up-to-date characteristics. A more sophisticated approach is to define in each service the way to handle the update locally (e.g., change the static ip address of a port), however this is not always possible - an example is the case of changing the disk image of a VM.

4.1.4.6 Dashboard - Horizon

Usability is essential in complex systems, hence programmers typically offer a user friendly interface, avoiding the unexperienced users the pain of going through a complex command line tool. Under this aspect, OpenStack is not an exception and provides a well structured web dashboard called Horizon, from which administrators can monitor the system and users can easily perform operations that otherwise require a long list of command-line inputs. The dashboard follows the OpenStack's main concept of abstraction and models VMs, networks and network services in a way that even an unexperienced user can easily understand. In Figure 4.9 is given a screen-shot of what a user sees when a simple topology is deployed. As is evident from the image, there is no indication about the physical location of the VMs (VM1, VM2, VM3), the service (Router) and other user instances.

4.1.4.7 Command line clients

Since all services are meant to receive commands via REST API, OpenStack developers have produced a command-line based client for each project that automates otherwise pedantic operation and complex typing of unicode strings. These clients perform a series of cURL requests to obtain the required result. As an example, consider that a user that wants a list of all his virtual machines with some details, using the client, it is easy: just type `"nova list"`. To achieve the same result, it is mandatory to acquire an authentication token, then use it to query the nova-api and obtain the list of instances. If more details are required, a request for each machine has to be dispatched. Other than automation, clients provide also data presentation so that instead of receiving responses in form of raw JSON or XML, better-looking and compact tables are displayed.

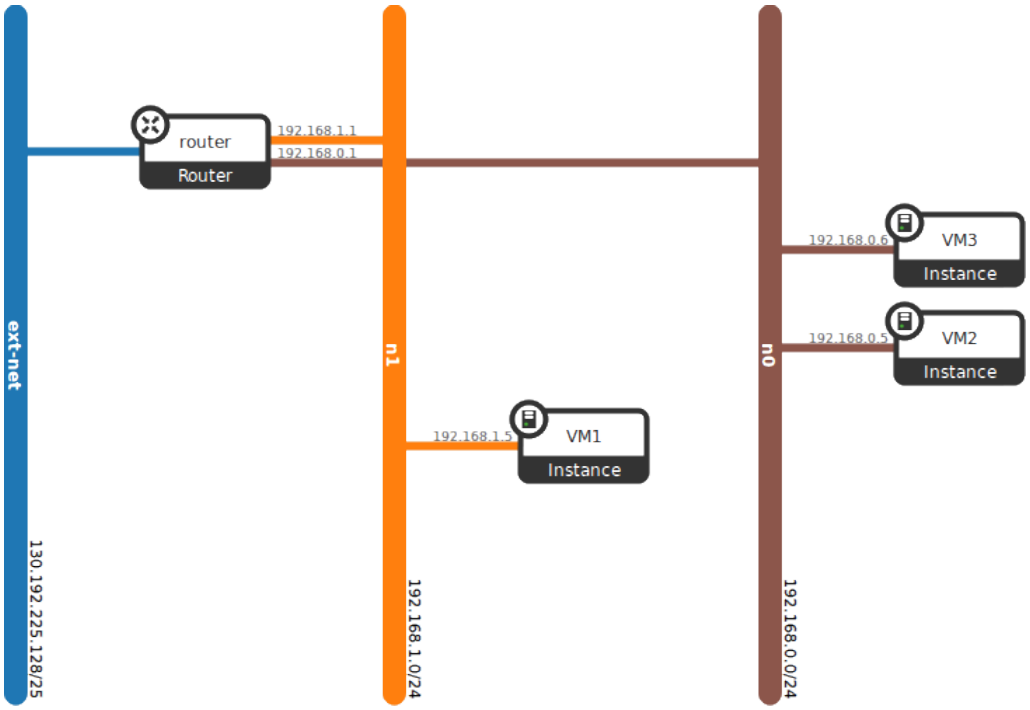


Figure 4.9: Dashboard view of network topology design with VMs.

4.1.5 OpenStack towards NFV

The OpenStack community continues to grow and the number of companies that decide to adopt the open-source solution is constantly augmenting. As it is easy to imagine, these companies work in different fields and have various needs. Keep up with all the requests of new features implementations is almost impossible but the community does its best to at least try. In this optic a new squad, called NFV team, is trying to propose new solutions that should be integrated into OpenStack to better support network function virtualization. As stated in their official page[16], their mission aims to define the use cases, identify and prioritize the requirements that are needed to run Network Function Virtualization workloads on top of OpenStack. This work includes identifying functional gaps, creating blueprints, submitting and reviewing patches to the relevant OpenStack projects and tracking their completion in support of NFV. The requirements expressed by this group should be made so that each of them have a test case which can be verified using an open-source implementation. This ensures that tests can be done without any special hardware or

proprietary software, which is key for continuous integration tests in the OpenStack gate. If special setups are required which cannot be reproduced on the standard OpenStack gate, the use cases proponent will have to provide a 3rd party CI setup, accessible by OpenStack infra, which will be used to validate developments against. All the proposals they have ever made can be found on their website[16] along with their statuses. Looking at the number of active blueprints it can be seen that as a group they are very active so there is quite the possibility their work towards a NFV fully-capable OpenStack will lead to nice results. As a down side, each one of their proposition has to be discussed, reviewed,validated, implemented and finally tested. It is evident that such procedure requires time, also it has to be compatible with all the already existent features. As an example, let's take the *Services Insertion, Chaining and Steering*[10] extension they proposed for the neutron module. Taking a look at the white-board it can be easily seen the time it is required to finally come to a working implementation; in fact it the proposal came up on late 2013 and as far as October, the 13th 2014 this project still has not been marked as approved. Even if its purpose is quite similar to the FlowRule abstraction, this proposed extension has evolved during the year coming to look more like the ETSI service chain instead of appearing like an OpenFlow rule as the FlowRule does. Instead of defining a port-to-port connection with matching rules, they decided to introduce two kinds of resources: *ports chain* and *label*. A label is no more than a series of OpenFlow-like fields that will identify a traffic flow. Consequently a set of neutron ports are ordinately grouped to create a ports chain. Finally a ports chain can be associated to a label; in this way all the information to perform traffic steering are provided to neutron. With the wisdom of hindsight this approach looks more elegant than the one introduced by us; it has to be said though that our extension, (with all its limits) have been implemented in just few weeks. Furthermore, being an ad-hoc design, is able to support traffic steering between ports that are not been instantiated by OpenStack, permitting to bypass the limitations imposed by the network node (e.g., obligation for the traffic incoming and outgoing to pass through the virtual router). Taking another look at the NFV team's blueprint list, we can find a set of interesting proposals from which our architecture can benefit. As an example there

are the *unaddressed interfaces for NFV use cases* that brings up the idea of defining transparent ports and this is precisely our use case. Furthermore they also have numerous blueprints that aim to improve the nova scheduler, unfortunately no one focused toward a network aware scheduling.

4.2 OpenDaylight

OpenDaylight (ODL), as presented in the official project website[11], is an open platform for network programmability to enable SDN and NFV for networks. The software is a combination of components including a fully pluggable controller, interfaces, protocol plug-ins and applications. With this common platform both customers and vendors can innovate and collaborate in order to commercialize SDN and NFV based solutions. If stripped to its very minimal core, it results to be just an openflow controller, which is able to instruct the controlled switches to behave in a certain way. There are numerous openflow controller projects since the standard came out back in December 2009 but there are factors that really differs OpenDaylight from the others; first there is the community that keeps developing it, some major vendors such as Cisco, IBM, HP, Brocade Juniper, Microsoft and others also support the project both economically and with workforce. Secondly it has been engineered to make easy for developers to add their additions in form of a bundle that will easily integrate and interact with the core components and other plug-ins either available in the public repository, proprietary or created by other developers.

4.3 Integration between the two projects

Already in the first official stable release of OpenDaylight - codename Hydrogen - published in February 2014 and in particular in the `virtualization` version, a plug-in for the integration with OpenStack was made available. This add-on represent the first step towards what will likely become the de facto standard for datacenters management. Being in its early times, the OpenDaylight framework and therefore the functionalities exported to OpenStack Neutron, the behavior is

sometimes different from the expected one; either way making this two software products work together in a cooperative way is an aspect of particular interest. In fact as mentioned before OpenStack sees the entire underlaying physical network as a commodity and has no control over it, OpenDaylight on the other hand has been projected to do exactly so. The integration between the two projects will result with the possibility to aggregate the full control of the datacenter in the aspects of both managing the compute and tuning the network accordingly in an automated and robust way with lot of space for improvement and optimization of performance and capabilities resulting in a better quality of service (QoS), lower management cost and even in the chance of offering users new custom functionalities. However leveraging these functionalities requires an OpenDaylight fully controlled network and in order to allow that the whole datacenter communication system needs to be openflow-capable. Given that openflow physical switches are still not so common on the market, having the possibility to pilot them with the very same controller that manages the virtual switches in the compute nodes will be an enormous plus for the datacenter network administrators whom will be allowed to control, configure and monitor all the devices from a unique and centralized point.

Chapter 5

General Architecture

Our reference architecture to deliver network services on the wide provider network is shown in Figure 5.1. As evident from the picture, it allows the deployment of network services through three main portions, namely the *service layer*, the *orchestration layer* and the *infrastructure layer*.

5.1 Service layer

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 an high level formalism (called *service graph* and detailed in Section 6.1) 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 an AppStore-like marketplace that enables the selection of a precise VNF among the many available. The service graph should be provided together with several non-functional parameters. Particularly, we envision a set of *Key Quality Indicators (KQIs)* specifying, for instance, the maximum latency allowed between two VNFs, or which is the maximum latency

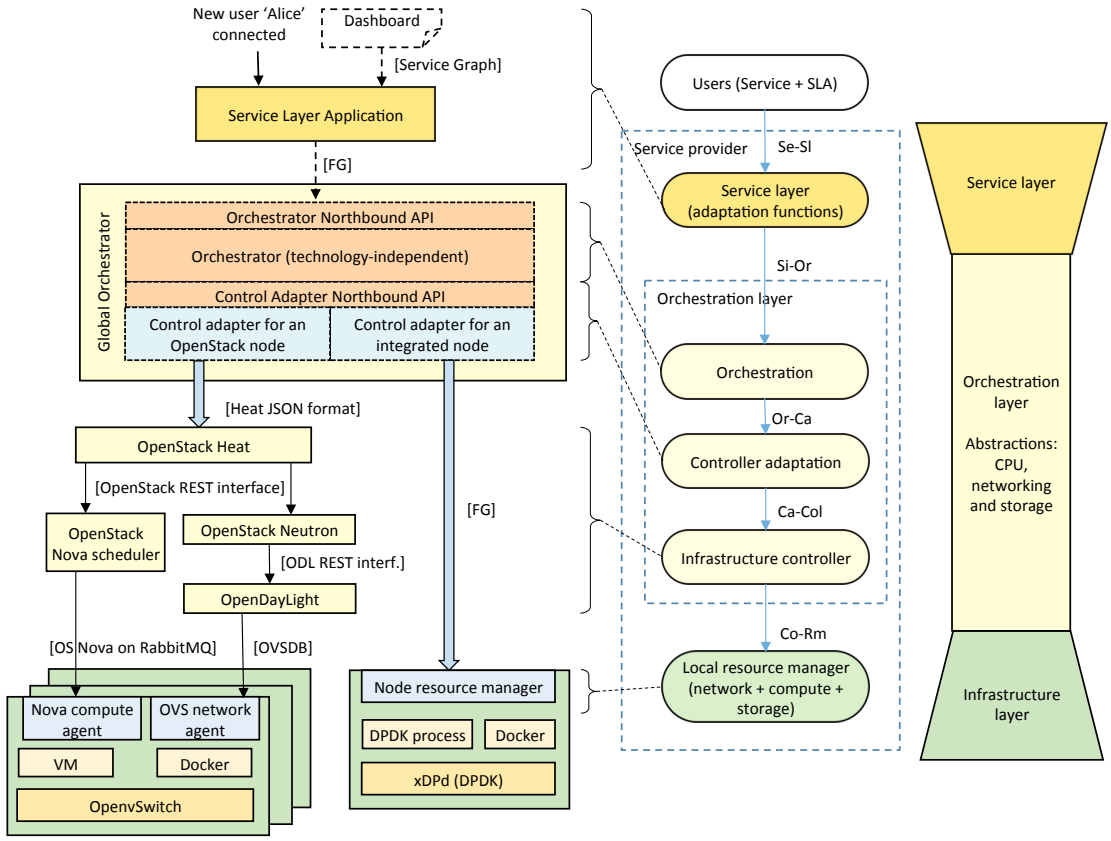


Figure 5.1: Overall view of the system.

that can be introduced by the entire service. We also foresee the definition of a list of high-level policies to be taken into account during the deployment of the service. An example of such policies could be the requirement of deploying the service in a specific country because of legal reasons.

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* detailed in Section 6.2. This new representation provides a more precise view of the service to be deployed, both in terms of computing and network resources, namely Virtual Network Functions (VNFs) and interconnections among them, always conserving KQIs and policies imposed by the actor who defined the service.

The service layer should also define some APIs to be exported to the lower layers

of the architecture, so that it can be notified of some low level events that could be exploited to implement the service logic. An example of such events may be the connection of a terminal device (e.g., a laptop) to the network, which could trigger the deployment of a new service graph in the service layer.

As depicted in Figure 5.1, the service layer includes a component implementing the service logic (identified with the service layer application (SLApp) block in the picture), 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. This logic (and the corresponding implementation) is strictly related to the use case under consideration. Particularly, in this thesis we consider the case in which the *end users* connected to the network, as well as the *Internet Service Provider (ISP)*, can provide a description of services to be implemented in network of the ISP itself.

However, it is worth pointing out that the service northbound interface design principles are very similar to a generic public-cloud API, such as Amazon Web Services (AWS) [1]. In fact, the service layer could not only be used to conceive services in a typical ISP provider-customer scenario, but it opens the door to third-party providers to be involved into custom-service definition. In effect, this interface enables a cloud-like service delivering in which 3rd-party providers (e.g., content-providers) could offer services on an ISP infrastructure, orchestrating resources on demand and being billed for their utilization in a pay-per-use fashion.

5.2 Orchestration layer

The orchestration layer sits below the service layer, and it is responsible of two important phases in the deployment of a service on the physical infrastructure. 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 such as new VNFs, as well as the consolidation of several VNFs into a single one. Second, the orchestration layer implements the scheduler that is in charge of deciding where to instantiate the requested service. The scheduling could

be based on different classes of parameters: (i) information describing the VNF, such as the CPU and the memrory required; (ii) high-level policies and KQIs provided with the forwarding graph; (iii) resources available on the physical infrastructure, such as the presence of a specific hardware accelerator on a certain node, as well as the current load of the nodes themselves.

As depicted in Figure 5.1, the orchestration layer 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* (Section 6.3), and changes according to the resource on which the service is going to be instantiated. 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 flow from an unknown device arrives to one node), and to notify it to the upper layers of the architecture.

As shown in Figure 5.1, 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.

Having in mind the heterogeneity (e.g., core and edge technologies) and size of the provider network, it is quite evident how the orchestration layer (and in

particular the global orchestrator, which sits on top of many resources) is critical in terms of performance and scalability of the entire system. For this reason, according to the picture, the global orchestrator has syntactically identical northbound and southbound interfaces (in fact, it receives a forwarding graph from the service layer, and it is able to provide a forwarding graph to the next component), which opens the door to a hierarchy of orchestrators in our architecture. This would enable the deployment of a forwarding graph across multiple administrative domains in which the lower level orchestrators expose only some information to the upper level counterparts, so that the architecture can scale up with a huge number of physical resources in the infrastructure layer. Although such a hierarchical orchestration layer is an important aspect of our architecture, it is out of the scope of this thesis and it is not considered in the implementation detailed in Section 8.2.

5.3 Infrastructure layer

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; in particular, we consider two classes of infrastructure resources.

The first class consists in cloud-computing domains, and it is called *OpenStack-based node* in Figure 5.1, referencing one of most popular cloud management toolkit (Chapter 4). Each OpenStack-based node actually consists of a cluster of physical machines managed by the same infrastructure controller.

The second class of resources is instead completely detached by traditional cloud-computing environments. In fact, we envision that some useful resources to execute VNFs could be outside of the datacenter, such as the traditional (or improved) home-gateways hosted in the end users' homes. An element of this class, shown in the bottom-right part of Figure 5.1 and called *integrated node*, consists of a single

physical machine and it is mostly based on dedicated software. Note that, in this case, the infrastructure controller is integrated in the same machine hosting the required service.

As a final remark, the infrastructure layer does not implement any logic (e.g., packet forwarding, packet processing) by itself; in fact, it is completely configurable, and each operation must be defined with the deployment of the proper forwarding graph. This makes our architecture extremely flexible, since it is able to implement each type of service and use case defined in the service layer.

Chapter 6

Data models

This section details the three data abstractions introduced in Section 5, which are used by the architecture shown in Figure 5.1 to deploy the network services on the physical infrastructure. Our data models are inspired by the NFV ETSI standard 3, which proposes a service model composed of "*functional blocks*" connected together to flexibly realize a desired service. In order to meet the objectives described in the introduction, we declined this abstract model in multiple flavors according to the variegated granularity and details needed in the different layers. All those models are inspired by the objective of integrating this functional component description of network services and topology and the possibility to model also existing network topology and services.

6.1 Service graph

The **service graph (SG)** is an 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, policies, etc.).

From the point of view of the infrastructure, the SG consists of the set of basic elements shown in Figure 6.1, which have been selected among the most common

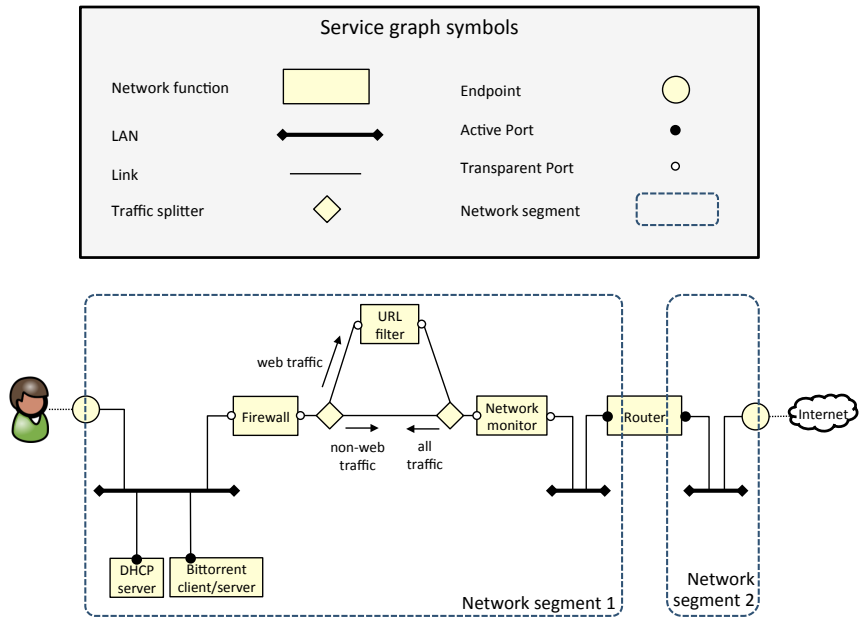


Figure 6.1: Service graph: basic elements and example.

elements that we expect are needed to define network services. In particular, a SG may include the following seven basic primitives:

- **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 (detailed in Section 6.4) describing the function itself in terms of RAM and CPU required, type of processor on which it can run (e.g., x86-64), number and types of ports, etc.

- **Active port**: it defines the attaching point of a network function that needs to be configured with a network-level address (e.g., IP), either dynamic or static. Packets directed to that port are forwarded by the infrastructure based on the link-layer address of the port itself (e.g., MAC address).

- **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. If traffic has to be delivered to that port, the network infrastructure has to "guide" packets to it, e.g., through traffic steering elements,

since the natural forwarding of the data based on link-layer addresses does not cross those ports.

- **Local area network (LAN)**: it represents the (logical) broadcast communication medium, i.e., the well-known primitive that allows data-link frames to be delivered to the correct recipient. 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. Furthermore this provides an abstraction similar to the one available in cloud management systems, such as OpenStack, that usually offer the *network* as one of the fundamental building blocks.

- **Point-to-point 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/merger**: it represents a functional block that allows to split the traffic based on a given set of rules, or to merge the traffic coming from different links. For instance, it can be used to redirect only the outgoing web traffic toward an URL filter (Figure 6.1), while the rest does not crosses that network function.

- **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. To this purpose, each endpoint is associated *(i)* with an identifier, which can be used as a reference to connect two SGs together, and *(ii)* with a cardinality, which indicates if the endpoint can be connected to one or many other endpoints.

In the example of SG provided in Figure 6.1, three network functions are connected to a LAN, featuring both active (e.g., the DHCP server and the bittorrent machine, which need to be configured with IP addresses) and transparent ports

(the firewall). The outgoing traffic exiting from the firewall is received by a splitter/merge block, which redirects the web traffic to an URL filter and from here to a network monitor, while the non-web traffic travels directly from the firewall to the network monitor. Finally, the entire traffic is sent to a router before exiting from the service graph. It is worth noting that traffic splitter/merger modules are bidirectional and may have different behaviors in the opposite directions. For instance, the traffic splitter/merger on the right will send *all* the traffic coming from Internet to the firewall, without sending anything to the URL filter as this block needs to operate only on the outbound traffic.

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.

A SG engine may also assess formal properties on the above configuration parameters; for example, the service may be analyzed to check if the IP address assigned to the VNFs active ports are coherent among each others. To facilitate this work, the SG includes the concept of **network segment**. According to Figure 6.1, each network segment is the set of LANs, links and ports that are either directly connected or that can be reached through a network function by traversing only its transparent ports. Hence, it corresponds to an extension of the broadcast domain, as in our case data-link frames can traverse also network functions (through their transparent ports), and it can be used to check that all the addresses (assigned to the active ports) of the same network segment belong to the same IP subnetwork. As shown in the picture, a network segment can be extended outside of the SG; for instance, if no L3 device there exists between an end user terminal and the graph endpoint, the network segment also includes the user device.

As a final remark, the configuration parameters for the network functions, as well as the possibility of assessing formal properties on them, are out of the scope

of this paper and will be investigated in our future work.

6.1.1 Cascading service graphs

As introduced above, the SG endpoints are associated with multiple parameters that are used to connect SGs together (cascading graphs). Particularly, the name is the foundation of the SG attaching rules, indeed the logic that connects the endpoints is demanded to service layer but is done on the basis of their names.

The cardinality specifies if that endpoint can be used to connect the graph with one or many other graphs, i.e., to implement a one-to-one or a one-to-many connection. Finally, the optional ingress matching rule indicates which traffic is allowed to enter into the graph through that particular endpoint, e.g., only the packets with a specific source MAC address.

The cardinality specifies if that endpoint is used to connect the graph with one or many other graphs, i.e., to implement a one-to-one or a one-to-many connection. Finally, the optional ingress matching rule indicates which traffic is allowed to enter into the graph through that particular endpoint, e.g., only the packets with a specific source MAC address.

The rules that define how to connect several graphs together change according to both the cardinality of the endpoints involved and to the presence of an ingress matching rule on such endpoints. While the case in which only two endpoints have to be connected (i.e., direct connection) does not present significant issues, Figure 6.2 presents two examples in which an one-to-one endpoint (in the SG on the left) must be connected to a common graph through its one-to-many endpoint. In this example we consider a first packet flowing from the left to the right, while the return packet follows the opposite path.

Example 1 in Figure 6.2 presents two egress endpoints that are associated with a ingress matching rule specifying which traffic must enter into the graph through that endpoint. This ingress matching rule must be used, in case of return traffic, to deliver the desired packets to the correct graph, notably HTTP traffic to the "HTTP-SG" and FTP traffic to the "FTP-SG". This is achieved by transforming the ingress endpoint of the "TCP-SG" into the set of components enclosed in the

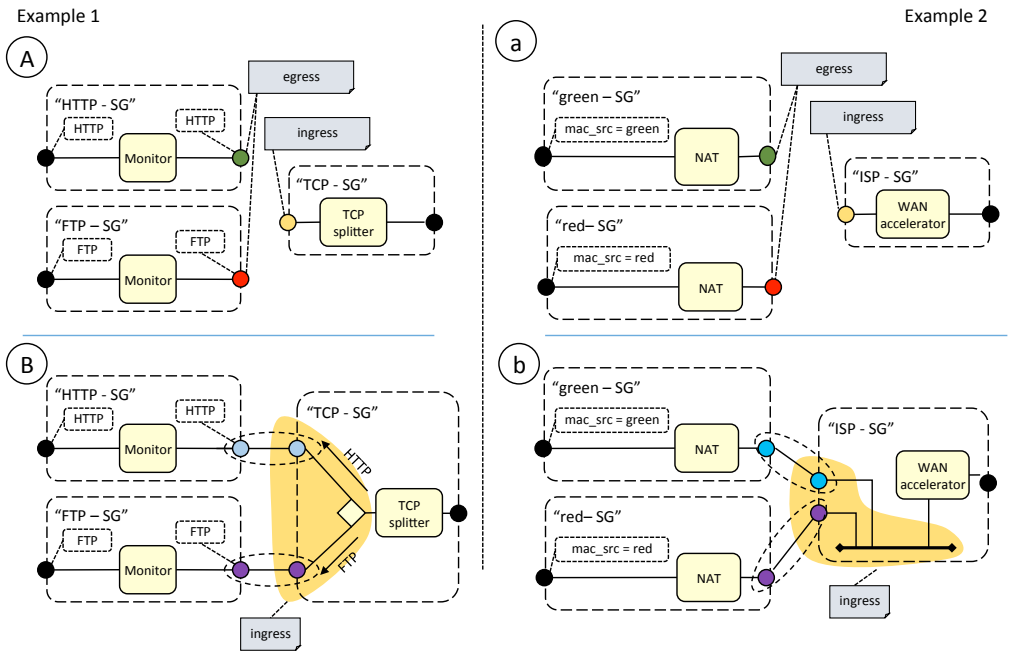


Figure 6.2: Connection of many user SGs to a single ISP SG.

green shape of Example 1(b), notably a traffic splitter/merger module attached with many new endpoints, each one connected to a different graph. This way, the common "TCP-SG" will be able to dispatch the packet answers to the proper graph. Example 2 in Figure 4 shows instead the case in which the egress endpoints are not associated with any ingress matching rule, which makes impossible to determine the right destination for the packets on the return path as a traffic splitter/merger module cannot be used in the "ISP-SG" to properly dispatch the traffic among them. In this case, the ingress endpoint of the common "ISP- SG" is transformed into a LAN connected to several new endpoints, each one with one-to-one cardinality and dedicated to the connection with a single other graph. This way, thanks to the MAC-based forwarding guaranteed by the LAN, the "ISP-SG" can dispatch the return packets to the proper graph, based on the MAC destination address of the packet itself.

6.2 Forwarding graph

The SG provides an 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 that are needed by the service to operate. Hence, it must be translated into a more resource oriented representation, namely the **forwarding graph (FG)** (Chapter 3), through the so called **lowering process**.

The differences between the SG and the FG, together with the steps needed to transform a the first representation into the second one (i.e., the lowering process), are shown in Figure 6.3 and discussed in the following:

- **Control and management network expansion**: 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. An example of this step is evident by a comparison between Figure 6.3(a) and Figure 6.3(b), in which a control/management network consisting of a L2 switch VNF has been added to the graph[1].

- **LAN expansion**: all the LANs expressed in the SG are replaced with VNFs implementing the MAC learning switch. This step is again shown in Figure 6.3(b), and it is needed in order to translate the abstract LAN element available in the SG into a VNF actually realizing the broadcast communication medium.

- **Service enrichment**: the graph is analyzed 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. As shown in the example provided in Figure 6.3(c), if the graph analysis determines, for instance, that the graph

[1]It is worth pointing out that, although the control and management network in Figure 6.3(b) only includes a L2 switch, this network could also include other VNFs, such as a firewall.

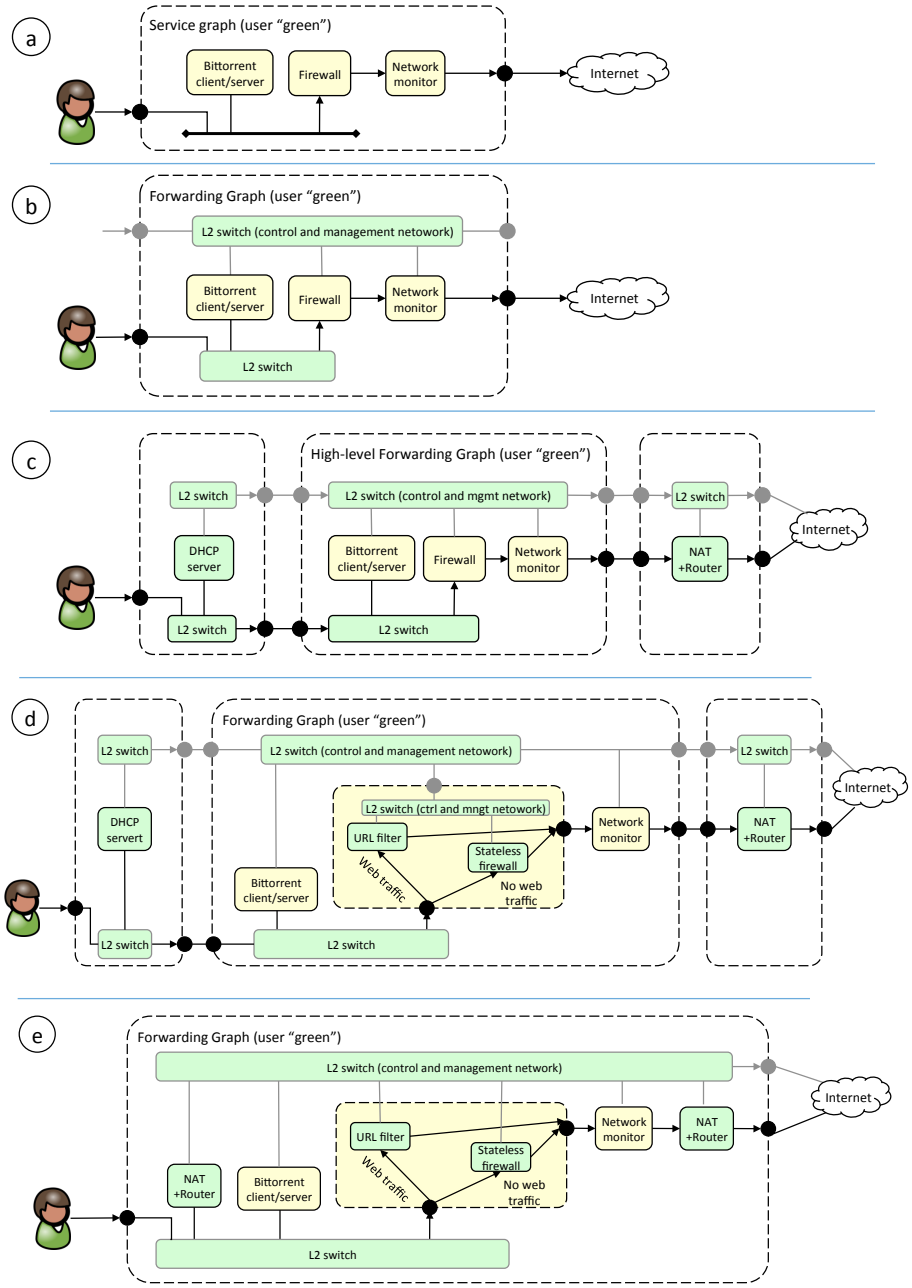


Figure 6.3: From the service graph to the forwarding graph: the lowering process.

does not include a DHCP server and does not terminate with a VNF acting as a router and as a NAT, these functions are automatically added at this step of the lowering process.

- **VNFs expansion**: according to its template described in Section 6.4, a VNF may be expanded in a number of VNFs, properly connected in a way to implement the required service. As an example, the firewall in Figure 6.3(c) is replaced, in Figure 6.3(d), with a subgraph composed of an URL filter only operating on the web traffic, while the non-web traffic is delivered to a stateless firewall. As evident, the ports of the "original" VNF are now the endpoints of the new subgraph, which also have a control network dedicated to the new VNFs. 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 (Chapter 3).

- **Consolidation**: it consists in 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 on the physical infrastructure. An example of this step is provided in Figure 6.3(e).

- **Endpoint translation**: it is the last step of the lowering process, in which 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; 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) can be represented with a sequence of "flowrules" (Listing 6.1), 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.

```
1 "flowrules": [
2   {
```

```
3       "action": {
4         "VNF": {
5           "id": "Traffic monitor XY",
6           "port": "in/out:0"
7         },
8         "type": "output"
9       },
10      "flowspecs": [
11        {
12          "match": {
13            "etherType": "0x800",
14            "protocol": "tcp",
15            "destPort": "80"
16          },
17          "priority": "1000",
18          "id": "fa126"
19        }
20      ]
21    }
22  ]
```

Listing 6.1: Example of flourule associated to specific VNF port.

The flowspec supports all the fields defined by Openflow 1.0 [12] (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.

6.2.1 Structure of the FG

This section contains a detailed analysis of the forwarding graph description.

The notation used to describe the forwarding graph is JSON (JavaScript Object

Notation). A first level description of FG is given by Listing 6.2.

```
{
  "VNFs": [],
  "endpoints": [],
  "id": ""
}
```

Listing 6.2: High-level view of FG.

The information contained in FG is: *(i)* a list of virtual network functions, *(ii)* a list of endpoints, *(iii)* a unique identifier of the FG. With regards to VNFs as shown in Listing 6.3, they are characterized by a `vnf_descriptor`, a list of `ports`, a `name` and an `id`.

```
"VNFs": [
  {
    "vnf_descriptor": VNF_URL,
    "ports": [],
    "id": "Client_Switch",
    "name": "Switch"
  },
  ...
]
```

Listing 6.3: High-level view of VNFs.

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 (Section 6.4). Now we are going to analyze each field of the `VNFs` element in detail.

The `id` is an unique identifier of the VNF in the FG, while the `name` identify the type of VNF. The `vnf_descriptor` is an URL containings a manifest of the virtual network function that are (described in section 6.4). The `port` list that are shown in the Listing 6.4 contains an `id`, an `ingoing_label`, and an `outgoing_label`. The port `id` is composed of two different parts, the part before the column identify the label (this one will be explained in section 6.4) 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 flow rules only for ingoing traffic to that port. While the outgoing labels are mandatory, the ingoing labels is needed only when a port is connected to an endpoint, since the endpoint does not have any flowrule associated. In addiction, flow rules contained in an ingoing label have an addictional field to identify the endpoint from which the traffic comes. This field is called `ingress_endpoint` and it is a leaf of flowspec object (line 10 of Listing 6.1). Finally the flowrule object, as discussed before (Listing 6.1) contains a list of matches on packets and the relative action.

```
1  "ports": [
2    {
3      "id": "L2Port:0",
4      "ingoing_label": {
5        "flowrules": []
6      },
7      "outgoing_label": {
8        "flowrules": []
9      }
10   },
11   {
12     "id": "L2Port:1",
13     "outgoing_label": {
14       "flowrules": []
15     }
16   },
17   {
18     "id": "L2Port:2",
19     "outgoing_label": {
20       "flowrules": []
21     }
22   }
23 ]
```

Listing 6.4: High-level view of ports

It is worth noting that all the flow rules of a single port must forward the totality of traffic, hence, a rules of specific port cannot purge the traffic, and if we want to drop some kind of traffic we must do that in a VNF. Therefore it is clear that, the only type of actions can be "output".

The `endpoints` (Listing 6.2) are the termination of graph. In the FG instead of SG, the endpoints can assume various characterizations like for example: *(i)* tunnel termination, *(ii)* physical port or *(iii)* virtual port. For instance, the example in Listing 6.5 shows an endpoint that is actually a physical port called "ge0". This characterization is needed to effectively connect graphs among each other and map the endpoint concept on physical resources.

```
1 "endpoints": [
2   {
3     "id": "Egress_endpoint",
4     "name": "ISP_INGRESS",
5     "connection_cardinality": "1_to_n",
6     "type": "physical",
7     "port": "ge0"
8   }
9 ]
```

Listing 6.5: NF-FG - Example of endpoint definition.

With regard to name, it provides a tool to implement the logic of connection between graphs, together with a further field called `connection_cardinality`, which specify the type of connection. It can assume two different values: `(i) one-to-many` or `(ii) many-to-many` as specified in Section 6.1.1.

6.2.2 Operations

As stated above, several actions can be executed on a FG. Particularly, these actions are: *(i)* endpoint connection like in the passage between Figure 6.3(d) and Figure

6.3(e) and *(ii)* the node expansion like in the passage between Figure 6.3(c) and Figure 6.3(d).

6.2.2.1 Connection

The connection between different graphs is made through the endpoints, which represent the ingress/exit point from the graph. The connection depends on the service layer application implementation, indeed there are two possible types of connections: *(i)* the graphs are logically connected, so the output of this operation is a new graph that doesn't have anymore the endpoints used for the connection, or *(ii)* the connection operation preserve the endpoints used for the connection, so they can be used again for an other connection. The former case is evident by comparing Figure 6.3(d) and Figure 6.3(e), while the latter case represents the situation in which the results of the operation are in turn two different graphs that communicate with each other through the endpoint (e.g. user graph connected to an ISP graph). The connection process requires Cartesian product of the ingoing labels of ports directly connected to the endpoints involved in the operation of one graph and the outgoing label of specular ports on the other graph. In this way problem can arise in the resulting flows (especially with regard to the priorities, seen that we use Openflow in our implementation).

6.2.2.1.1 The flow rule merging problem Endpoints connection has a problem merging the flowrules. Analyzing the problem, only the intersection between flow rules should remain after the connection of two endpoints, so when we try to connect two ports that were connected to two endpoints, if the intersection of the outgoing rules towards the endpoint of one and the ingoing rules from the endpoint of the other is null, then this two ports shouldn't be connected. To better explain this problem, consider the example shown in Figure 6.4. As evident the connection between VNF3 and VNF1 should disappear since there are not traffic matches in common. Furthermore, the traffic from VNF4 to VNF2 becomes the intersection between the totality of traffic (*) (prensent on the rule that bring the traffic from VNF4 to endpoint) and the totality of traffic less tcp traffic on port 80 (*-TCP80)

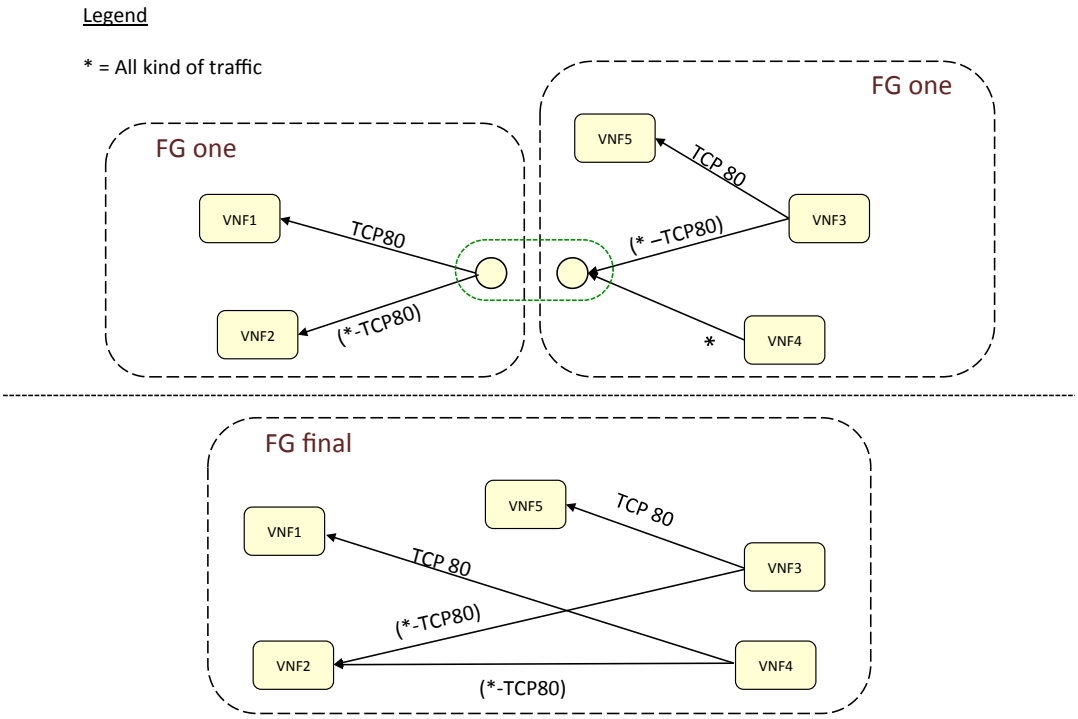


Figure 6.4: The merging problem.

(prensent on the rule that bring the traffic from endpoint to VNF2), that is *-TCP80.

The biggest problem related to the fusion of rules is to assign the right priority to the traffic. In our formalism is not possible to express the rule *all traffic* ***except*** *tcp traffic on port 80*. To better understand consider the high part of Figure 6.5 where the following rules are expressed on a port of VNF3: one that brings traffic to VNF5 that has matches all TCP traffic on port 80, and having a certain priority, and the other rule for traffic to the endpoint which matches all the traffic having a lower priority than the other. Hence, the priorities of the rules play a fundamental role in the actual construction of the network topology, and allow to realize the rule "all traffic less". In the example shown in Figure 6.5, are included the priority. As we can see from the "FG final 1" of Figure 6.5, assigning to flow rules on the port of VNF1 the priority of the flow rules of "FG one" the flow towards VNF5 is never used because it more specific than the other and have the lowest priority. Furthermore using, for all pair of flows generated from the connection, the priority of starting FG

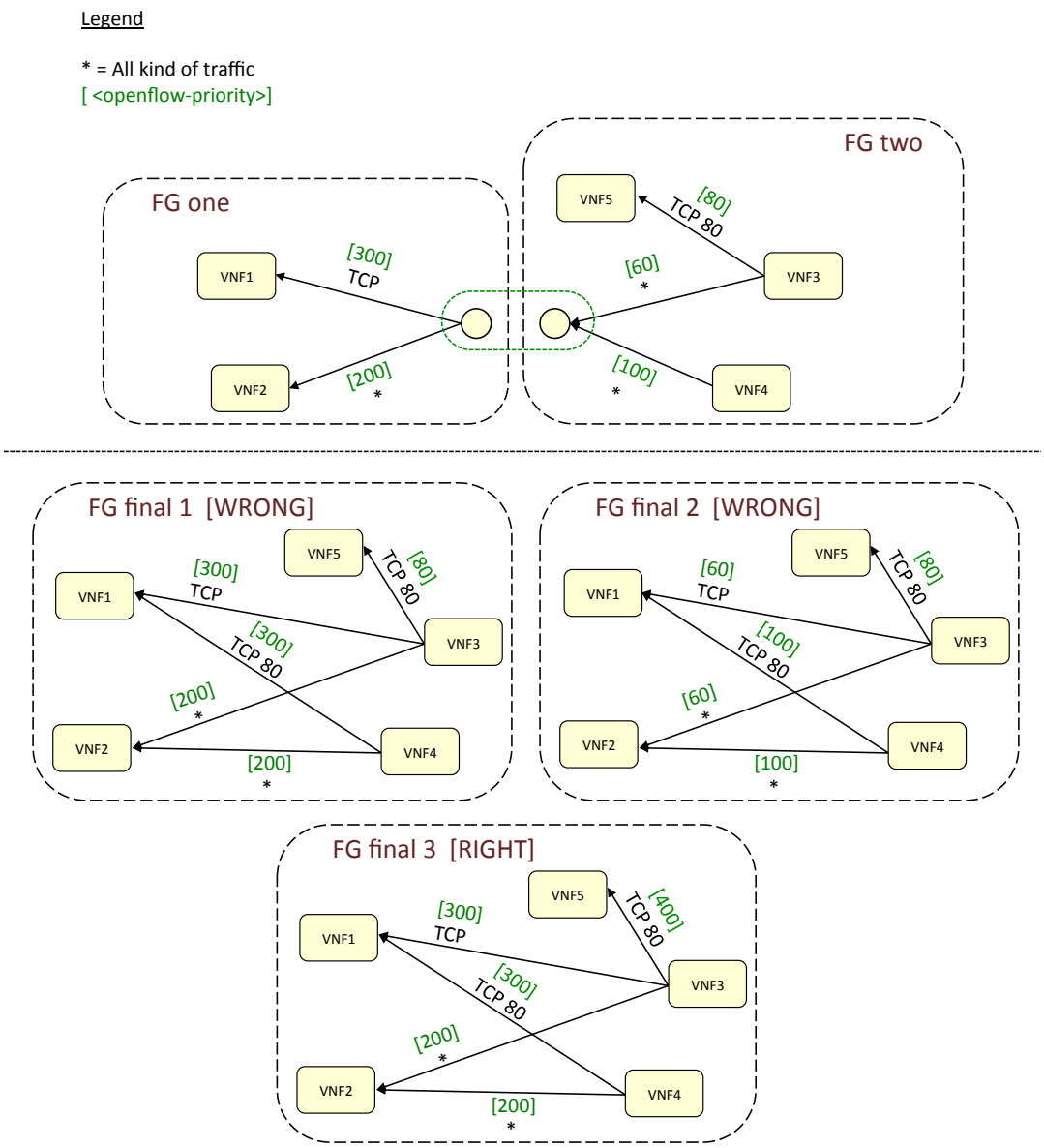


Figure 6.5: NF-FG - The merging problem.

we will have the same priority to multiple rules. In this case, is not guaranteed the direction of flow (“FG final 2” in the bottom right of Figure 6.5). Hence the right choice is more complicated of the simple copy of priority and in certain cases we should change accordingly a lot of rules (“FG final 3” in the bottom of Figure 6.5).

For these reasons and because this is not the core of the thesis the code implemented has the limitation of handle only *1 to 1* or *1 to n* rules fusion, so the case *n to n* like for VNF3 in Figure 6.5 is not taken into account.

6.2.2.2 Expansion of a VNF

The terms VNF expansion means to replace a VNF with a new graph, in which the endpoints of the graph correspond to the ports of the original VNF. For instance, looking at Figure 6.3(c), we can see that the firewall with three ports is replaced with a graph having three endpoints in figure 6.3(d). Furthermore is, possible to have a recursive expansion, for example, the firewall VNF of Figure 6.6(a) can be expansed like in Figure 6.6(b) where there is a VNF named load balancer. This VNF, in turn, is expanded in two functions, an openflow switch and an openflow controller that permits, with a proper logic in the openflow controller, to operate the switch as a load balancer.

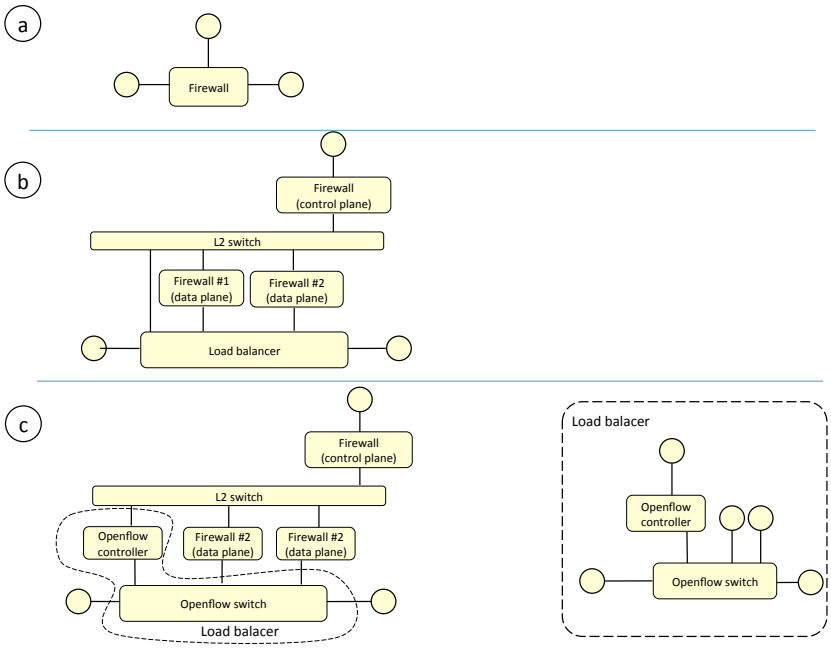


Figure 6.6: Recursive VNF explosion.

Substantially, the expansion operation of a VNF may be reduced to a sequence

of connection operations between logical graphs. In fact, once the VNF is replaced with the new graph, it is performed a connection of the endpoints of the new graph with endpoints born by cutting the links of the VNF expanded. Obviously, the rules associated with the ports of the old VNF are now associated at the ports of the boundary VNFs of new graph.

6.3 Infrastructure graph

The **infrastructure graph (IG)** is a further representation of the service to be deployed, which is semantical, but not syntactical, equivalent to the FG. In fact, it consists of the sequence of commands to be executed on the physical infrastructure in oder to properly deploy the required VNFs and to create the paths among them.

The IG is obtained through the so called **reconciliation process**, which is described in the following of this section and consists in the mapping of the FG description (Section 6.2) on the resources available on the infrastructure.

First, some of the VNFs in the FG could be mapped on some modules (both software and hardware) available on the node on which the graph is going to be deployed, instead of being implemented with the specific image indicated in the template. For example, if the node is equipped with a virtual switch (vSwitch) implementing the MAC learning algorithm (right part of Figure 6.7), the L2 switch VNFs in the FG are removed, and their functionalities are mapped on the vSwitch itself. Instead, as depicted in the left of Figure 6.7, if the node is equipped with a pure Openflow vSwitch, all the VNFs specified in the FG will be implemented through the proper images. Obviously, other mappings between the VNFs and the resources available on the node are possible, according to the specific implementation of the infrastructure layer. This translation is another implementation of the "*recursive functional blocks*" concept [5]; in fact, starting from the same FG, it will produce a different number of deployed VNFs according to the capabilities (e.g. L2 switching native support) of the node actually hosting the service.

Second, the flow rules defining the links of the FG, i.e., the connections among the VNFs, are properly translated according to the technology used by the physical

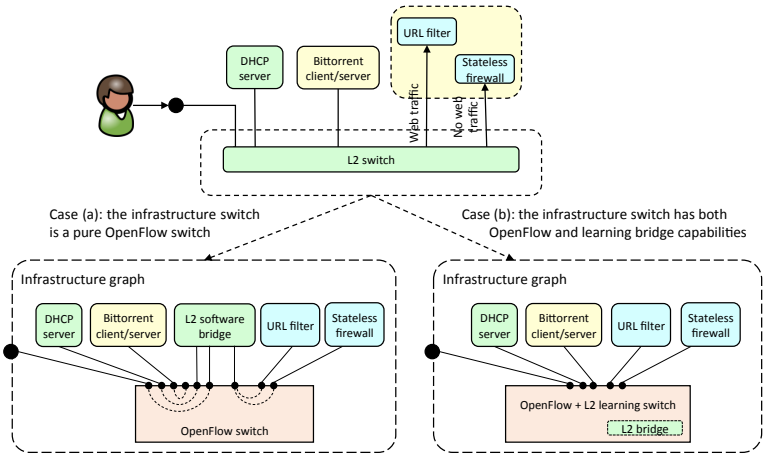


Figure 6.7: Part of the reconciliation process.

node to implement the graph. For example, if the physical node interconnects the VNFs through an Openflow vSwitch, each flow rule is converted in a number of Openflow `flowmod` messages. However, other implementations of the infrastructure layer could implement these connections through other technologies, such as GRE tunnels or VLAN tags.

Similarly, each VNF is converted in the commands required to retrieve the image and start it up; also in this case, these commands depend on the technology implementing the VNFs (e.g., virtual machine, Docker containers, etc.).

6.4 VNF template

As introduced above, each network function is associated with a template, which describes the VNF itself both in terms of infrastructure and in terms of configuration; an example of such a template is provided in Listing 6.6.

```
1 {
2   "name": "firewall",
3   "expandable": false,
4   "uri": "http://repository_of_vnf_descriptor/example",
```

```
5   "vnf-type": "virtual-machine",
6   "memory-size": 4096,
7   "root-file-system-size": 40,
8   "ephemeral-file-system-size": 0,
9   "swap-disk-size": 0,
10  "cpu-requirements": {
11    "platform-type": "x86",
12    "socket": [
13      {
14        "coreNumbers": 1
15      }
16    ]
17  },
18  "ports": [
19    {
20      "position": "0-0",
21      "label": "control",
22      "min": "0",
23      "ipv4-config": "dhcp",
24      "ipv6-config": "none",
25      "name": "eth"
26    },
27    {
28      "position": "1-1",
29      "label": "external",
30      "min": "1",
31      "ipv4-config": "none",
32      "ipv6-config": "none",
33      "name": "eth"
34    },
35    {
36      "position": "2-N",
37      "label": "internal",
```

```
38          "min": "1",
39          "ipv4-config": "none",
40          "ipv6-config": "none",
41          "name": "eth"
42       }
43    ]
44 }
```

Listing 6.6: NF-FG - VNF template.

As evident from listing example, the template contains some 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 subgraph 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 in Listing 6.6 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 (e.g., the external port of the firewall should be connected towards the Internet, while the internal ones should be connected towards the users). 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), it is

also possible to insert `N` as value of last number of the range to indicate a variable number of interfaces available on VNF. For instance, the VNF of the example has one control port (or no one), one external port, and at least one internal port (in fact, it has a variable number of internal ports, which can be selected during the definition of the SG). 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 the FG there is a port with id equals to "internal:2" (hence we have at least three ports labeled as internal), the name of the internal interface of the VNF is "eth4", because the value position for ports labeled as internal is "2-N" and the value of field name is "eth". Obviously, as explained above, a VNF must have not more of one port type with a name with variable number interface.

Chapter 7

Use case: user defined network services

Chapter 5 and Chapter 6 respectively provide a general overview of the architecture and a description of the associated data-models; those concepts could be used in a plenty of different use cases involving multiple actors in defining completely virtualized services. In order to provide a concrete use case for these abstract models, in the following of the thesis we consider a particular scenario in which *end users*, such as xDSL customers, can define their own service graphs to be deployed on the wide Internet Service Provider (ISP) infrastructure.

The ISP controls the entire infrastructure; particularly, it provides a service layer through which the end users can define the SG, the orchestration layer and infrastructure resources. In addition, the end users devices connect to the network through ISP-controlled equipments (i.e., integrated nodes or OpenStack-based nodes), which represent the entry point of user traffic within the operator network.

Finally, an end user's SG can only operate on the traffic of that particular end user, i.e., on the packets he sends/receives through his terminal device. On the contrary, the ISP is enabled to define a SG that includes some VNFs that should operate on all the packets flowing through the network; hence, this SG must be shared among all the end users connected to the provider infrastructure.

7.1 Challenges

The use case just introduced presents some interesting challenges to be solved.

First, the service layer must be able to recognize when a new end user attaches to the network in order to authenticate the user himself. Note that, since the infrastructure layer does not implement any (processing and forwarding) logic by itself, this operation requires the deployment of a specific graph that 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 should be shared among several end users.

Second, after that the user is authenticated, the service layer must retrieve his SG and then connect it to the ISP graph; particularly, the two graphs must be connected in a way so that the user traffic, in addition of being processed by the service defined by the user himself, is also processed by the VNFs selected by the Internet provider. This interconnection of several graphs in cascade can be realized by exploiting the graph endpoints elements, which are provided by the SG formalism.

Third, the user SG must be completed with some rules 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 (only) operates on the packet belonging to the user himself.

The data-model proposed in Chapter 6 could flexibly model such work-flow. All SGs described above (e.g. end-user SG, authentication SG, ISP SG) could separately be defined in an independent way by the different actors involved. The architecture stack assures the deployment of the resulting SG, firstly obtaining the FG by the lowering process and then instruct the infrastructure component through the IG, leveraging the reconciliation process to optimize the whole process. In particular, a general problematic that looks particularly important in this use-case is represented by SG interconnection. In fact, all cited steps are concerned by a dynamical and sometimes timing-related SG interconnection. Moreover, such interconnection model has to be detached, as is done for VNF definition (Section 5), from infrastructure technology details. As mentioned before, user devices are connected thanks to rule to his graph, such method allows to attach and cleanly detach the user device dynamically from a certain SG. This may allow the user device to communicate on a

certain SG and be unaware detached when a certain event happen (e.g. User device has been authenticated). This abstraction is massively leveraged by the service layer while composing services.

Finally, the service layer must require (to the lower layers) to deploy the user graph, and to create the proper tunnels on the network infrastructure so that the user traffic is bring, from the network entry point, to the graph entry point, which could have been deployed everywhere on the physical infrastructure.

Chapter 8

Prototype implementation

This chapter presents the preliminary implementation of the architecture introduced in Section 5, detailing the components used and the engineering choices that have been made in order to create our prototype. Both global orchestrator and service layer application have been written from scratch in python. Being their webapp are used Gunicorn [7] ("Green Unicorn") and Falcon [6]. Gunicorn is a Python Web Server Gateway Interface HTTP Server for Unix, whereas Falcon is a Python minimalist WSGI framework which allow developers to write Web applications or services without having to handle such low-level details as protocols. First of all a description of our use case is given.

8.1 The service layer

In our implementation of the service layer, the **service layer application (SLApp)** is mostly based on custom code, and it is provided with some APIs to interact with the orchestration layer and with other components required to implement the use case described in Section 7.

Particularly, it exploits two OpenStack modules: an extended version of **Keystone** (detailed in Section 4.1.4.1), and **Glance** (detailed in Section 4.1.4.2), used to store the VNF templates.

The choice of Glance to store the templates is not a very good choice, a better

solution could be using **Swift** (Section 4.1.4.2), because it is an object storage instead of **Glance** that is an image repository. We have made this choice only to reduce the number of OpenStack components installed for performance reasons.

According to our use case, at the startup of the infrastructure, the SLApp provides both the ISP graph and the authentication graph to the orchestration layer, which in turn deploys them on one (or more) node(s) available in the infrastructure layer.

It is worth noting that, in addition to the VNFs implementing the authentication graph, the end users authentication process exploits the OpenStack module Keystone (available in our service layer), which has been extended to store the SG, in addition to all the other information needed to authenticate a user.

In the current implementation of SLApp all the user are always connected to the authentication graph through a rule that sends all the traffic to that graph. So the user can immediately reach the authentication web portal and authenticate himself. Whatever the future implementation of SLApp should be able to intercept the event of a new flow available at one node at the edge of the network, and should receive 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 SLApp 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 the modifications on the physical infrastructure. This modification permits a dynamic connection of users to our network service, and makes it much more flexible.

The user authentication triggers the instantiation of his own SG; at this point, the SLApp retrieves the proper SG description from Keystone, connects it to the ISP-defined SG, and starts the lowering process aimed at translating this high level view of the service into a FG. In particular, the SLApp executes the "*control and management network expansion*" and the "*LAN expansion*", as shown in Figure 6.3(b).

Moreover, it performs the "*service enrichment*" step, aimed at connecting the graph with network functions that are needed for the correct delivering of the service defined by the user. For instance, as already discussed above (Figure 6.3(c)), these VNFs may include a DHCP server and a VNF implementing the NAT and router functionalities; although they are not of interest for the end user, these VNFs are automatically added by the SLApp because required for the correct behavior of the network. At this moment the addition of this VNFs is forced by the logic of SLApp, even if this function is already present in user service graph. This would call for a more fine control that is left to future developments.

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, as required by the implemented use case. Moreover, the graph endpoint through which the traffic sent by the end user enters into the graph is associated with the entry point of such traffic in the provider network. This way, the orchestration layer will be able to configure the network in order to bring the user traffic from its entry point into the network to the node on which the graph is deployed.

As mentioned before, the logic of endpoints connection is managed by the SLApp. Indeed it is in charge of connecting graphs according to the logic of our use case (described in Section 7). Hence, when an user graph is instantiated it checks the name of the endpoints, `"USER_INGRESS"` for the endpoint that manage the traffic through the user, `"USER_EGRESS"` for the endpoint connected to ISP and `"USER_CONTROL"` for the endpoint of the control network (inserted by the SLApp). The endpoint named `"USER_EGRESS"` will be connected with the endpoint of ISP graph named `"ISP_INGRESS"`. This connection is made changing the `id` of the endpoint with the `id` of ISP endpoint. Being the endpoints of ISP connected to user a `one-to-many` endpoint, this requires an update of the ISP graph, which create a new `one-to-one` endpoint, this update is needed every time a graph is attached to it. In the real implementation, the ISP graph is instantiate with a fixed number of endpoints and no one updates is done, because the current implementation of infrastructure layer does not support updates of endpoints.

As a final remark, Global orchestrator has to keep 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 to the network with a new device, the SLApp: *(i)* retrieves the FG already created from the orchestration layer; *(ii)* extends it by adding rules so that also the traffic coming from/going to the new device is injected into the graph; *(iii)* sends the new FG to the orchestration layer.

8.1.1 Authentication graph

The *authentication graph* is required in our use case in order to authenticate an end user when he connects to the provider network with a new device. In particular, as shown in the left of Figure 8.1, the SG defined in the service layer consists of a LAN connected to: a VNF that takes care of authenticating the users, a DNS server and a DHCP server. However, according to its template, the VNF implementing the users authentication actually consists of three VNF images: an Openflow switch, an Openflow controller and a captive portal. The resulting FG, completed with a control network, is shown in the right of the picture; as evident from the picture, this graph is connected to the Internet only through the control network, so that end user's traffic cannot go outside of the graph itself.

As detailed above, when an unauthenticated user connects to the network, his traffic is bring to the authentication graph, so that the user can authenticate himself and trigger the instantiation of his own service. In particular, thanks to the DHCP server and the DNS server, the user device retrieves the network parameters and it is able to resolve the names; this way, the user can send HTTP packets and finally start the actual authentication process. In fact, the HTTP traffic entering into the Openflow switch is sent to the Openflow controller through an Openflow `packet in` message; this VNF modifies the original MAC and IP destination addresses with those of the web captive portal, and then sends back the packet to the Openflow switch, which will provide it to the proper VNF. Before responding with a login page, the web captive portal provides a HTTP `302 temporary redirect` message to the user, in order to notify the client of the redirection and avoiding wrong caching.

After the user authentication, the web captive portal contacts the SLApp through the control network and triggers the deployment of the proper SG on the infrastructure. In the deployment process the flow that bring the user traffic to its graph is instantiated last, this for maintain the connection with the captive portal until all resources of user graph are deployed. In this way when the user is no longer able to reach the captive portal, he is already connected to a new graph and thanks to a very short lease provided from DHCP of authentication graph, he immediately gets the new address without even noticing the change.

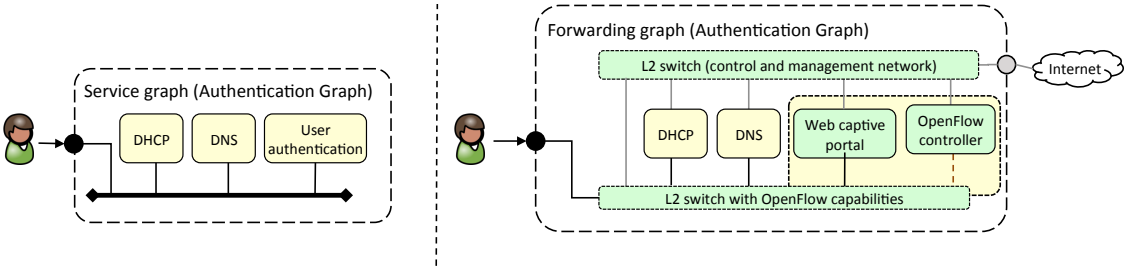


Figure 8.1: Authentication SG and FG.

8.2 Global orchestrator

As depicted in Figure 5.1, 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. This module receives the FG created by the service layer through the northbound API, manipulates it and sends the resulting FG(s) to the proper infrastructure controller(s). It is worth noting that our architecture consists of a single global orchestrator that sits on top of many physical nodes, even implemented with different technologies (e.g., the integrated node and the OpenStack-based node).

When the technology independent part of the global orchestrator receives the FG, it executes the following operations.

First, for each VNF specified in the graph, it retrieves a description of the function itself (i.e., its template) from Glance as detailed in Section 6.4, the template

contains information such as the amount of CPU, RAM and disk required by the VNF but it could also specify that the VNF is actually a sub-graph composed by other VNFs. In this case, the technology independent part of the global orchestrator executes the "*VNFs expansion*" step depicted in Figure 6.3(d), and retrieves the description of the new VNFs (which, in turn, could be recursively expanded in further VNFs). Then, the "*consolidation*" step of the lowering process is performed, so that the FG is simplified as shown in Figure 6.3(e).

At this point, the global orchestrator schedules the FG on the proper node(s) of the physical infrastructure. Although the general model presented in Section 5.2 supports a scheduling based on parameters such as CPU and memory requirements of the VNFs, KQIs (e.g., maximum latency, expected throughput) and high level policies, such a scheduler is let as a future work. In fact, in our first implementation of the global orchestrator, it simply instantiates the entire FG on the same node used as a network entry point for the traffic to be injected into the graph itself.

The resulting FG is then provided to the proper control adapter, according to the technology implementing the node of the infrastructure layer on which the graph is going to be scheduled. So far, two control adapters have been defined, in order to support the deployment of FGs into OpenStack domains and into integrated nodes; these adapters take care of translating the FG provided by the technology independent part into a formalism accepted by the proper *infrastructure controller*, which is in charge of sending the commands to the infrastructure layer. Moreover, they 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 a GRE tunnel on the infrastructure layer. This tunnel could be used to connect together two pieces of the same service but deployed on different nodes of the infrastructure layer; according to our use case, a GRE tunnel is required when the FG associated with an end user is deployed on a different node 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 never splits a graph in multiple parts, and deploys the graph on the node through which the traffic enters into the network, this feature

is never exploited by the global orchestrator.

As a final remark, the global orchestrator also supports the updating of existing graphs. In fact, when it receives a FG from the service layer, it checks if this graph (i.e., a FG with the same identifier) has already been deployed; in this case, both the FGs (the one deployed and the new one) are provided to the proper controller adapter, which sends to the infrastructure layer: *(i)* the difference between the two graphs in case of integrated node, or *(ii)* both the FGs in case of OpenStack-based node (As described in Section 8.4, in this case the difference is already calculated by the Heat module of OpenStack). Furthermore a graph update is done when it receive a graph with an endpoint ID equal to that of an other already instantiate. In particular this endpoint is instantiated in a different node. So, when the service layer attaches two graphs and this two graphs are instantiated in different nodes. In this case the orchestrator should update the old endpoint with the information needed to connect it to the other endpoint (i.e. information needed to create a tunnel GRE), and enrich the new endpoint with the same information.

8.3 The integrated node

This section describes the **integrated node** [2], i.e., a node of the infrastructure layer consisting of a single physical machine mainly running dedicated software; the overall architecture of this implementation is provided in Figure 8.2. Each integrated node is independent from the other nodes of the infrastructure layer and hence, as shown in the figure, its infrastructure controller can be *integrated* on the same server running the VNFs of the graphs.

Going into details, the FG is received through a REST API by the **node resource manager**, which is the component that takes care of instantiating the graph on the node itself; this requires the execution of the reconciliation process, to start the proper VNF images (downloaded from a **VNFs repository**) and to configure the paths among them. Particularly, the last two operations are executed through two specific modules of the node resource manager, namely the **compute controller** and the **network controller**.

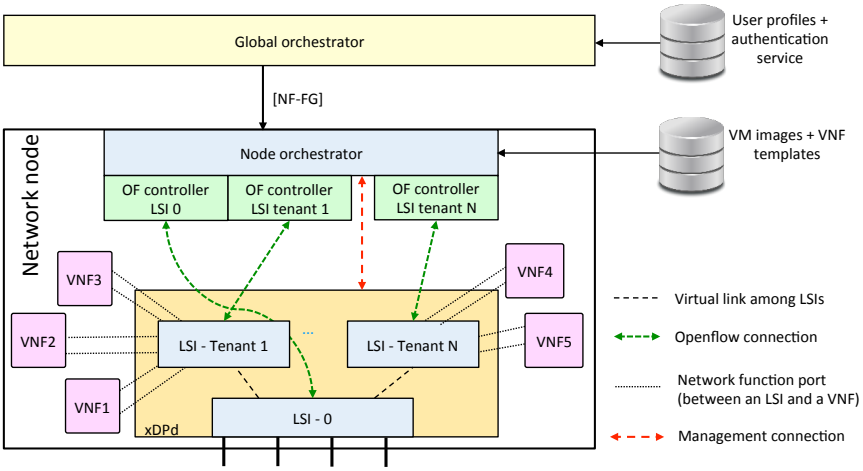


Figure 8.2: Logical architecture of the integrated node.

According to Figure 8.2, the traffic steering among the VNFs is based on **xDPd** [17], a framework that supports the dynamic creation of several (pure) Openflow switches called **Logical Switch Instances (LSIs)**; each LSI can be connected to physical interfaces of the node, to VNFs, and to other LSIs. In the prototype, a different LSI (called `tenant-LSI`) is dedicated to steer the traffic among the VNFs of a specific graph, while the `LSI-0` is in charge of classifying the traffic coming from the network (or from other graphs) and of delivering it to the proper `tenant-LSI`. The `LSI-0` is the only one allowed to access the physical interfaces, and the traffic flowing from one `tenant-LSI` to another has to transit through the `LSI-0` as well.

Being the LSIs Openflow switches, the rules describing how to steer the traffic in the graph are translated into Openflow `flowmod` messages. Moreover, since the LSIs are *pure* Openflow switches, the reconciliation process described in Section 6.3 does not remove the VNFs implementing the L2 switch, which are then implemented using the proper software images and are not mapped on the vSwitch used to interconnect the components of the graph.

When a FG description (either a new one or an update of an existing FG) is received by the node resource manager, this module: *(i)* retrieves a software image

for each VNF required and installs it; *(ii)* instantiates a `tenant-LSI` on xDPd and connects it to the `LSI-0` and to the proper VNFs; *(iii)* crates an OpenFlow controller that is exploited to insert forwarding rules into the flow table(s) of the new LSI, in order to steer the traffic among the VNFs as required by the graph. In particular, the rules defining the paths among VNFs (and physical ports) originates two sequences of Openflow `flowmod` messages: one to be sent to the `LSI-0`, so that it knows how to steer traffic among the graphs deployed on the node and the physical ports; the other used to drive the `tenant-LSI`, so that it can properly steer the packets among the VNFs of a specific graph.

It is worth noting that, when a new flow enters into the `LSI-0`, it provides the new packets to its own Openflow controller through an Openflow `packet in` message; at this point the Openflow controller, through the network controller, notifies the upper layers of the architecture of this event, which will react properly according to the use case implemented in the service layer.

The integrated node supports three flavors of VNFs: DPDK processes [4], VNFs deployed in Docker containers [3], and VNFs running in VMs. While the former type provides better performance (in fact, an LSI exchanges packets with DPDK VNFs with a zero-copy mechanism), Docker containers and VMs guarantee properties such as isolation among VNFs, as well as they allow to limit the CPU and memory usage of VNFs themselves.

As a final remark, the integrated node has been designed having in mind a network aware scheduling of the VNFs on the cores available on the physical machine, although this feature has not been implemented yet in our current prototype. In fact the node resource manager, which takes care of both deploying the VNFs and configuring the vSwitch to properly steer the traffic among them, receives the entire FG at the same time from the upper layer, which describes both the VNFs to be executed and the paths among them. As a consequence, this module could schedule the VNFs in a way to optimize the packets flow among the functions themselves and between the VNFs and the NICs, by taking into account the way in which they are interconnected within the FG.

8.4 The OpenStack-based node

This section presents the **OpenStack-based node**, a node of the infrastructure layer consisting of a cluster of servers within the same OpenStack domain. As shown in Figure 8.3, all the physical machines of the cluster are managed by a single infrastructure controller, which is composed of a number of OpenStack modules and a SDN controller.

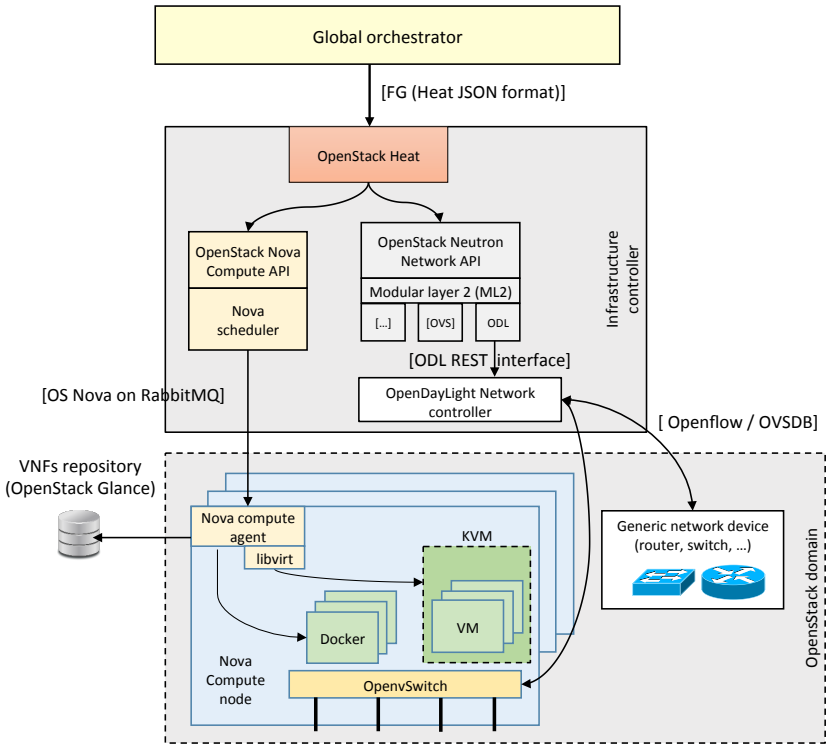


Figure 8.3: OpenStack-based node.

Openstack usage in our architecture represents an interesting choice for legacy compatibility, as well as it allows us to reuse all the features it already implements. However, since OpenStack was designed to support the deployment of cloud services, several modifications have been done in order to enable the deployment of FGs as well, in order to use it in our architecture. Any changes made to OpenStack are designed to keep the legacy behavior. These modifications, as well as the operations executed to instantiate the FG are described in the following of this section.

As depicted in Figure 8.3, to implement the OpenStack-based node and its infrastructure controller we use the following components: *(i)* **Nova**, which is the compute service; *(ii)* **Neutron**, namely the network service; *(iii)* the orchestration layer, called **Heat**; *(iv)* the repository for virtual machine (VM) images, called **Glance**. Openstack is able to start **VMs** by interacting with a wide range of different hypervisors (e.g. KVM, Xen, VMware); moreover, in order to properly steer the traffic between the several servers under its control, it can be integrated with a SDN controller: in our prototype we use **OpenDaylight (ODL)** (Section 4.2). As evident from the picture, Heat, Nova scheduler, Nova API, Neutron and ODL compose the infrastructure controller, while each physical machine executing the VNFs is a Nova compute node, which runs a Nova compute agent, **Open vSwitch (OVS)** [14] and the **KVM hypervisor**.

When the global orchestrator decides to deploy a FG in an OpenStack-based node, the proper control adapter translates the FG description into a format supported by Heat, performing a reconciliation step that removes, from the graph itself, all the VNFs implementing the L2 switch, since this functionality will be mapped on the OVS instances running on the physical servers. In fact, OVS is not pure Openflow switch: it is both able to forward traffic based on traffic steering rules as well as to implement the MAC learning algorithm. The current control adapter despite OpenStack uses OVS, has not yet been implement the reconciliation process because OpenStack manages isolation between networks with VLANs. This means that if the user traffic enters through the switch (to reconcile) the port where the user traffic comes from must be an access port that tags traffic with the right tag belong to the `neutron` network. To be used in our prototype, Heat has been extended in order to support the `flow-rule` primitive, which describes how to steer the traffic between the ports of the VNFs composing a graph. This primitive provides an interface similar to the OpenFlow 1.0 `flowmod`; however, it allows the traffic steering between virtual ports without knowing in advance the physical server on which the respective VNFs will be scheduled.

As soon as Heat receives the FG, it sends a sequence of commands to Nova for each VNF of the graph in order to deploy and start the VNF itself; at this point, the

Nova scheduler *(i)* selects the physical server on which the VNF must be deployed using the standard OpenStack "filter & weight" algorithm, *(ii)* sends the proper command to the Nova compute instance on the selected node, which in turn *(iii)* retrieves the VNF image from Glance and finally *(iv)* starts the VM.Note that no modification is done in Nova compute, in order to support the deployment of the FGs. The scheduling algorithm over cited is very simple and consists of two steps: in the first one, all the Nova compute nodes are considered, and those that are not able to run a VM are filtered (e.g., because the VM requires an hypervisor that is not available on the node). In the second phase, a weight is associated with each one of the remaining servers, and the one with higher weight is selected to run the VM. The weights are calculated by considering the resources available on the machine. It is worth noting that Nova scheduler has two limitations: *(i)* it schedules a VNF as soon as it receives the command from Heat; *(ii)* it does not have any information on the paths among the VNFs in the graph. As a consequence, the FG could be split on the available compute nodes without taking into account the paths among the VNFs, resulting in suboptimal performance.

After that the VMs implementing the required VNFs have been started, Heat sends a `flow-rule` at a time to Neutron, which takes care of creating the proper connections among these VNFs. Hence, also Neutron has been extended to support the `flow-rule` primitive, in addition to the standard abstractions used to implement traditional L2 networks (e.g., broadcast domains, IP subnets, ports). It is worth noting that our `flow-rule` is functional equivalent to the Neutron official traffic steering extension [10]. However, it has not been used in our prototype because: *(i)* its implementation is not available (at the beginning of November 2014) while our demo of traffic steering in OpenStack was released in July; *(ii)* it does not support ports that are outside the OpenStack domain. When Neutron receives a `flow-rule`, starting from the network topology (retrieved through ODL) it creates a set of openflow-like rules and sends those to ODL. At this point, the `flowmods` are created to OpenDayLight, which sends them to the proper switches; note that these switches could be either inside a Nova compute node, or physical switches used to interconnect several servers, in case the VNFs have been instantiated by the Nova

Table 8.1: Integrated vs OpenStack-based node.

	Integrated node	OpenStack-based node
Compatible with existing cloud environments	no	yes
Complete control of the FG	yes	no (due to the network node)
Support to smart scheduling of the FG	possible	requires many changes to the internals of OpenStack
Type of VNFs	Docker containers, DPDK processes, Virtual machines	Virtual machines, Docker containers (not completely supported)

scheduler on many compute nodes.

To conclude, the OpenStack-based node notifies the upper layers of the architecture, when a new flow (belonging to a new device) enters into the node itself. As already stated above, this information may be exploited by the service layer logic in order to implement a specific use case.

8.5 Discussion: Openstack-based node vs. integrated node

Table 8.1 summarizes the main features of the Integrated node and of the OpenStack-based node, as well as it shows the main differences among them. As evident from the table, unlike the integrated node, the OpenStack-based node supports the deployment of a FG in a cloud environment, in which a single OpenStack instance manages a cluster of physical servers (like in boxes on the right of Figure 8.4) called Nova compute nodes. Further OpenStack manages also the case of an edge box, like boxes on the left of Figure 8.4. This because it permits the instantiation of the only `compute node` on this boxes and instead the `controller node` and a `network node` in the remote server.

However, it has several limitations compared to the integrated node, as described in the following.

First, the upper layers do not have the complete control on the service actually deployed in an OpenStack-based node since, as described in Section 8.4, each OpenStack domain is connected to the Internet through the so called *network node*. As a consequence, packets towards/from the Internet are handled by the functions

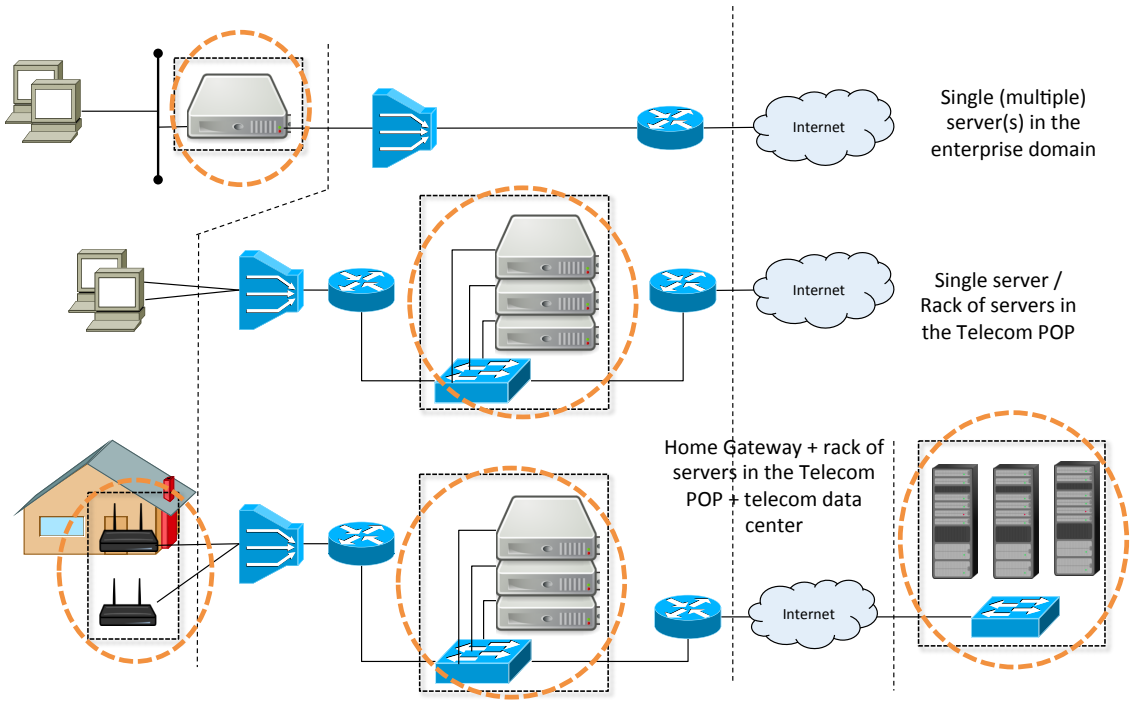


Figure 8.4: Target scenario.

running in this component (e.g., the NAT, the router), before/after being actually processed by the VNFs implementing the required service.

Moreover, according to Section 8.3, the integrated node could potentially schedule the VNFs on the available cores in a way that considers the connections between these VNFs in the graph, in order to improve the performance of the system. This optimization is possible because the node resource manager receives the entire FG description at the same time, and hence it has all the information related to the VNFs required and the paths among them, before actually deploying the VNFs themselves. Instead, in an OpenStack-based node, the entire graph description is only received by Heat, and not by the components actually starting the VNFs and creating the paths on the vSwitch(es). In particular, the Nova compute agent receives the commands to start the VNFs one at a time, and it does not receive

informations among the paths among them at all; in fact, this information is received by the OVS running on the physical servers through an Openflow connection established with OpenDaylLight.

Similarly, the Nova scheduler (which sits on top of all the physical machines belonging to an OpenStack domain) receives the information related to the VNFs to be started one at a time, while it does not receives the connection among them at all; as a consequence, it cannot select the compute node on which a VNF must be instantiated based on the connection among the VNFs in the FG.

It is worth pointing out that, currently, this network aware scheduling is neither implemented in the integrated node, nor in the OpenStack-based node; however, while it can be easily introduced in the former prototype, it would require many modifications to the internal operations of OpenStack, in order to be implemented in the Nova scheduler and in the Nova compute node.

Another difference between the integrated node and the OpenStack-based node is in the type of VNFs supported. Particularly, while former node supports VNFs implemented as Docker containers, VMs and DPDK processes, the latter only runs VNFs within virtual machines. In fact, although Docker is officially supported by OpenStack, we encountered some limitations in using it in our architecture

Chapter 9

Prototype validation

To validate the architecture described in the thesis, we carried out several tests aimed at both testing the functionalities implemented, and to measure the performance of the infrastructure layer in terms of throughput, latency introduced and resource required.

9.1 Service overview

The FGs deployed in the tests are shown in Figure 9.1; according to our use case implementation, these graphs include the authentication graph used to authenticate new end users connected to the network, and the ISP graph, which provides connectivity to the Internet and is crossed by the traffic generated from/going towards all the end users. The control network of this ISP graph also includes a firewall, so that only the authorized entities (e.g., the ISP itself) can control and configure the deployed VNFs.

As evident from the picture, each end user graph provides an example of traffic steering, since it requires that the packets towards the Internet is split so that the web traffic is provided to a traffic monitor and then to a firewall that blocks the HTTP GET towards specific URLs, while the other packets simply traverse a second traffic monitor VNF. It is worth noting that, thanks to the control interface of the traffic monitors we are able to observe the packets flowing through the specific VNF,

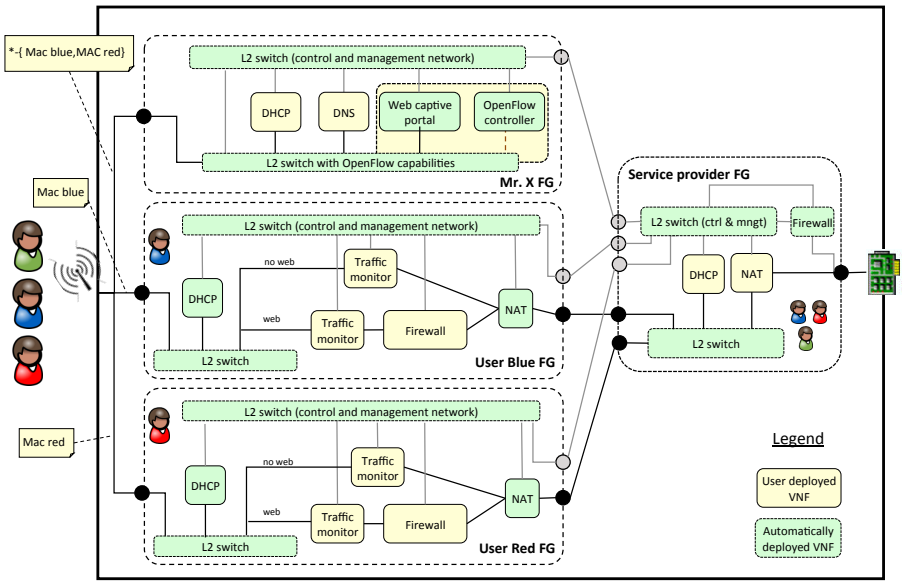


Figure 9.1: Use case scenario.

and hence to validate the correct behavior of the traffic steering mechanism.

During the tests carried out on the OpenStack-based node, the VNFs are implemented as VMs running on the KVM hypervisor and in a second OpenStack-based node VNFs are implemented in Docker containers. Instead, in the tests with the integrated node, the firewall is implemented as a DPDK process, while all the others VNFs are implemented in Docker containers. In particular, both the VMs and the Docker containers run an Ubuntu operating system, and the VNFs are implemented through *standard* Linux tools (e.g., iptables).

As a final remark, according to our use case and the current implementation of the architecture, the end users are directly connected to the node on which their graphs are deployed.

9.2 Performance evaluation

This section shows the tests executed in order to measure the performance obtained with the preliminary implementation of our architecture.

During the tests, a machine is dedicated to the execution service layer (i.e.,

service layer application, Keystone) and the global orchestrator; it is equipped with 16 GB RAM, 500GB HD, Intel i7-2620M @ 2.7 GHz (one core plus hyperthreading) and OSX 10.9.5, Darwin Kernel Version 13.4.0, 64 bit, which is the same for both the infrastructure nodes.

The infrastructure layer is implemented on a set of servers with 32 GB RAM, 500GB HD, Intel i7-3770 @ 3.40 GHz CPU (four cores plus hyperthreading) and Ubuntu 12.04 server OS, kernel 3.11.0-26-generic, 64 bits. In case of OpenStack-based node, a first machine hosts the infrastructure controller (Heat, Nova scheduler, Nova API, Neutron and ODL), the network node, while another one is dedicated to the implementation of the Nova compute node, connected through a Gigabit Ethernet link. In addition to the nova compute that runs VMs another one node, that runs Docker container, is connected in the same OpenStack domain. In case of the integrated node, one of those machines executes all the software.

The memory required by the different components of the system is reported in Table 9.1, in which the consumption related to the Nova compute node and to the integrated node has been measured without any VNF deployed.

Table 9.1: Memory consumption.

Component	**Memory [MB]**
Service layer + global orchestrator (SLApp + Keystone + Horizon + additional (minor) libraries/components)	558.2
Infrastructure controller (Heat + Nova scheduler + Nova API + Neutron + ODL + additional (minor) libraries/components)	3396.7
Nova compute node (Nova compute + OVS + additional (minor) libraries/components)	294.8
Integrated node (node resource manager + xDPd + additional (minor) libraries)	120.5

As evident, the infrastructure controller for the OpenStack-based node is the heaviest component, while the requirements of the integrated node, which is almost based on ad hoc modules, is quite reduced.

9.2.1 Deployment time

The first kind of tests performed is concerned about the measuring of service deployment time. The table 9.2 shows the times of instantiation of the user service graph detailed before, and the transitory time between the instantiation of the graph and the time in which the user is able to ping the server. It also shows the total time of the SG deployment, as we can see, those times are anything but low. What appears to be obvious looking at the Table 9.2 is that, the imputable for those results is the infrastructure layer, which is in charge of both VMs/dockers instantiation and the creation of paths between them. It is worth pointing out that this time does not include the download of the VNF images from the repository, which are already cached on the physical nodes.

Table 9.2: Deployment time.

Component	Service layer	Global orchestrator	Infrastructure layer	Ping	**Tot**
Openstack single node (virtual machine)	0.83	0.61	127	49	177.44
Openstack sigle node (docker)	0.83	0.61	82	13	96.44
Integrated node	0.83	0.49	72.67	11	84.99

This test has been repeated using different graphs with a variable number of VNFs and the results are shown in Figure 9.2. All the VNFs use the same image, and all nodes used for the test have it stored locally. One OpenStack node uses VMs that run an Ubuntu Server 12.04 and the size of the image is 1.6 GB. The other one OpenStack node and the integrated node use the Docker containers that run an Ubuntu Server 12.04 and the size of the image is 0.324 GB. Furthermore, in both VMs and Docker containers are used linux bridges to forward of traffic between the vNICs. In these tests the ping time is not considered. Hence, this means that after the time shown in Figure 9.2 the user is not immediately able to access to his service. Indeed, as we can see looking the ping value in Table 9.2, especially in OpenStack with VMs, the time for starting the services is very high.

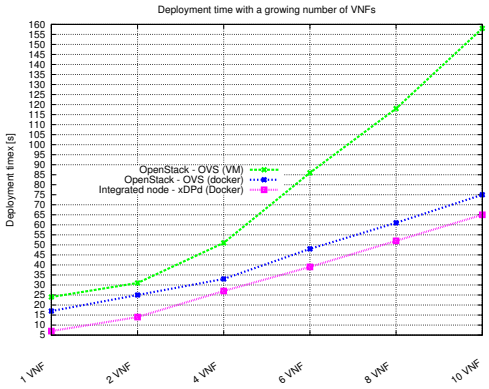


Figure 9.2: Startup times.

Anyway these performance show a behavior almost linear according to a growing number of VNFs.

9.2.2 Throughput and latency

The second phase of performance evaluation has been dedicated to analyze the throughput that we can achieve on the different infrastructure layers. As shown on the left of the Table 9.3, the different infrastructure layers used are *(i)* OpenStack-based node that runs virtual machines, *(ii)* OpenStack-based node that runs Docker container, and *(iii)* the integrated node. Also the tests of Figure 9.3 are performed connecting directly the user and the server machine. The first test carried out aims at measuring the latency introduced by the deployed services (Figure 9.1); in particular, the user device(s) sends 100 `ping` towards the server, and the results were averaged and reported in the left part of the table. The second and third tests are made on TCP traffic so they are most interested about perception of quality of service by end user. In particular the second test aimed at measuring throughput obtained during the download of a file of 4 GB from the server, and it is performed using `wget`, a Linux tool that uses the HTTP protocol. Whilst the third test is done using `iperf`, a network testing tool that can create TCP and UDP data streams and measure the throughput of a network that is carrying them. Finally, the last test is performed for evaluate the performance on UDP traffic, even in this case has

been used `iperf`. As a consequence, according to the user graph, while the ping are not handled by the firewall, this VNF is instead involved during the file transfer.

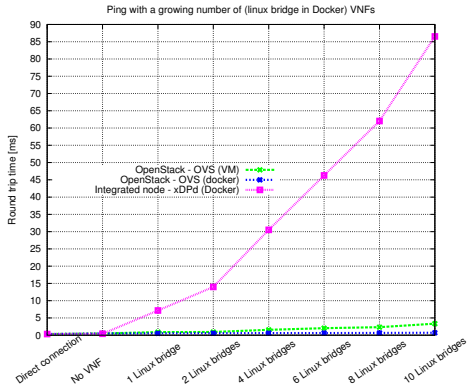
Table 9.3: Performance of the infrastructure layer.

	Ping (avg) [ms]	File transfer (wget) [Mbps]	TCP test (iperf) [Mbps]	UDP test (iperf) [Mbps]
#1 Direct connection	0.41	864	934	810
#2 OS compute node - VMs (i7)	3.33	770.4	841	779
#3 OS one compute node - docker (i7)	0.71	795.2	932	761
#4 Integrated node (i7)	62.04	210.4	247	410

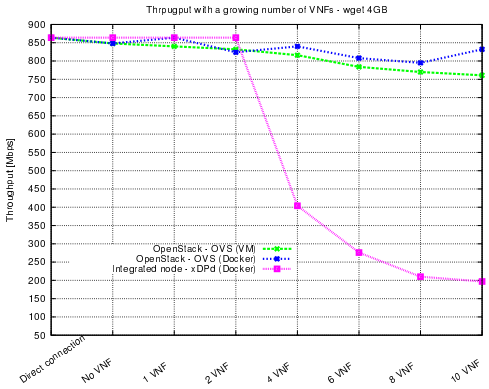
Furthermore all performance tests have been repeated using graphs with a variable number of VNFs, as done for evaluating the deployment time. The results are shown in Figure 9.3.

As expected, the deployment of a SG on the network does not come for free, since the results obtained are lower with respect to the case in which no service is instantiated between the user device and the server. However, as evident by a comparison between line `#1`, `#2` and `#3` of the table, this penalty is limited when the graph is deployed on an OpenStack-based node that use VMs, and almost zero when we use an OpenStack-based node using Docker containers.

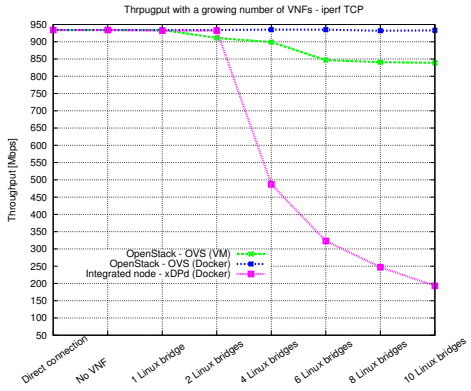
Instead, when the user graph is scheduled in the integrated node (line `#4`), performance are worse in all tests evaluated.



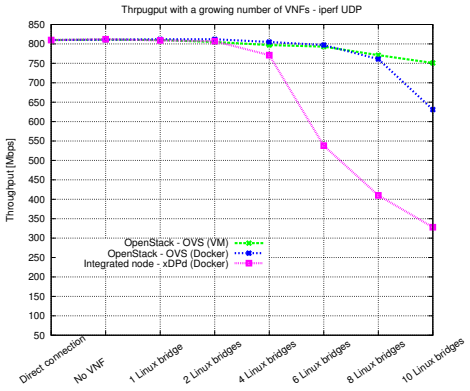
(a) Round trip time.



(b) Througput TCP - *wget*.



(c) Througput TCP - *iperf*.



(d) Througput UDP - *iperf*.

Figure 9.3: Performance tests.

Chapter 10

Conclusion and future works

This thesis presents a network orchestration architecture that, starting from the service required by external actors (e.g., end users, Internet providers), takes care of instantiating it on the physical infrastructure of the network, by exploiting the opportunities offered by the Network Functions Virtualization (NFV) and Software Defined Networking (SDN) paradigm.

The contribution of this thesis is twofold. First, we proposed a new formalism, called *service graph (SG)*, to flexibly model end-to-end network services. The SG data-model describes how to deliver flexible network services, leveraging existing elements and the traffic steering primitives introduced by NFV/SFC. It is worth noting that this SG definition is completely compliant with NFV principles of abstract description of a Service, but enriches its traditional expressiveness to model legacy networks and services.

The second contribution is made by the introduction of the *forwarding graph (FG)* and all the "lowering process" that leads to the deployment of an optimized service. This process of translation is capable to adapt the service delivering to available resources of the underlying infrastructure; moreover, it is also able to detect specific capabilities of selected nodes adapting the infrastructure graph obtained as output.

In order to validate our model, we implement the *OpenStack-based node* prototype and we also tested the *integrated node* for physical infrastructure.

While the latter consists of a single server mainly based on ad hoc components, the former is implemented as a cluster of server orchestrated by the an extended version of the OpenStack framework. Experimental results showed that, while the integrated node has low requirements in terms of memory, its performance are overcome by the OpenStack-based node in almost all the tests carried out except for tests about the deployment time, despite they are not so good are even better that the OpenStack-based results. The time of startup of the service is definitely the largest showed by the experimental results, because an user cannot wait that times for access their services.

As a plan for the future, we foresee two different challenges to be pursued in order to let this architecture to properly scale to the ISP network size. First, the proposal of an algorithm to implement a network-aware and resources-aware scheduling, capable of deploying VNFs on the nodes of the physical infrastructure by considering the paths expressed into the graph and also the features and the current usage of resources in single node (or domain). This require in the orchestrator the ability of splitting a single graph in multiple subgraphs and the instantiation of these in different nodes. Second, the definition of a hierarchical orchestration layer would be expedient in order to be able to scale out to potentially the whole ISP network. This would allow the deployment of a FG across multiple administrative domains, in which the lower level orchestrators expose only some information to the upper level counterparts. This scenario is perfectly compatible with our architecture and will be object of further analysis; in fact, the global orchestrator presented in the thesis has syntactically identical northbound and southbound interfaces (in fact, it receives a FG from the service layer, and it is able to provide a FG to the next component), and hence a hierarchy of orchestrators is possible.

To conclude, in the *OpenStack-based node* we would have to get rid of *network node* that force all outbound traffic from OpenStack to go from a router/nat positioned on node where *network node* is installed, which potentially affect the performance of our prototype.

Bibliography

[1] *Amazon web service*. URL: http://aws.amazon.com.

[2] Ivano Cerrato et al. "User-specific Network Service Functions in an SDN-enabled Network Node". In: (2014).

[3] *Docker*. URL: http://www.docker.com.

[4] *Dpdk*. URL: http://dpdk.org.

[5] *European Telecommunication Standards Institute (ETSI), Network Functions Virtualization Industry Specification Group*. URL: http://portal.etsi.org/NFV.

[6] *Falcon*. URL: http://falconframework.org/.

[7] *Gunicorn*. URL: http://gunicorn.org/.

[8] *key performace indicator*. Nov. 2014. URL: http://en.wikipedia.org/wiki/Performance_indicator.

[9] *MySQL official website*. Nov. 2014. URL: http://www.mysql.com/.

[10] *Neutron Services Insertion, Chaining and Steering blueprint*. Sept. 2014. URL: https://blueprints.launchpad.net/neutron/+spec/neutron-services-insertion-chaining-steering.

[11] *OpenDaylight project official website*. Nov. 2014. URL: http://www.opendaylight.org/software.

[12] *Openflow 1.0*. URL: https://www.opennetworking.org/.

[13] *OpenStack project official website*. Nov. 2014. URL: http://www.openstack.org/.

[14] *openvswitch project official website*. Nov. 2014. URL: http://openvswitch.github.io/.

[15] *RabbitMQ official website*. Nov. 2014. URL: http://www.rabbitmq.com/.

[16] *The OpenStack NFV Team web page*. Sept. 2014. URL: https://wiki.openstack.org/wiki/Teams/NFV.

[17] *xDPd project official website*. Nov. 2014. URL: http://www.xdpd.org/.