POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Integration of Smart Orchestration in an Open Source NFV Framework

Supervisors
prof. Guido Marchetto
prof. Riccardo Sisto

Candidate
Daniele Decaro

March 2019

To Dalia,

You will always be my guiding light.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Telecommunication industry service provisioning has been bound for years to a traditional model involving manual deployment of proprietary hardware and devices, often as part of architectures requiring strict components chaining and interdependencies. This led to long products and services lifecycle and longer time to value, dependency from specialized equipment and technicians skills, also possibly resulting in vendor lock-in, low service agility and an overall increase of capital and operational expenditure for the telecommunications service providers (TSPs).

The Network Functions Virtualization (NFV) paradigm has been proposed by a task force of leading TSPs to address the challenge of achieving a flexible way to meet always more demanding customer needs, and at the same time reducing costs and product time-to-market. By leveraging the potential of virtualization technology, it is possible to decouple the network function from the hardware it runs on, implementing network components in software-defined Virtual Network Functions (VNFs) that can be subsequently consolidated on commodity hardware such as high volume servers, storage, and switches. In this way, the TSP can chain together a set of VNFs to provision a network service, and then be able to instantiate it and relocate it according to the necessities without the need to purchase additional hardware capacity.

Although NFV paradigm is drawing considerable interest from both the academic and enterprise worlds, the ecosystem of tools and solutions supporting it is still in its infancy: this thesis work the main goal is to enrich one or more NFV orchestration platforms with support for smart service composition, VNF forwarding graph formal verification and deployment optimization by integrating the capabilities of Verifoo[1],

a project whose name is a contraction for Verification and Optimization Orchestrator (VerifOO). This solution would improve service composition and prevent human error in descriptor design, identify unused VNFs in service chains and optimize deployment.

Chapter 2

Network Functions Virtualization

Network Functions Virtualization (NFV) is an emerging network paradigm proposed to address issues in service development, delivery and derived costs for service providers. This chapter is intended to present the main concepts behind NFV, its architecture and characteristics, and related technologies.

2.1 Virtualization

During last two decades, virtualization has continued gaining traction in enterprise and private contexts enabling infrastructure scalability, agility, and efficiency in managing physical assets, and continues growing thanks to innovations as hardware-assisted virtualization technology (Intel VT-x and AMD-V in particular)[2], support of major operating systems, and a growing landscape of software. The term refers to the concept of creating software based representations of devices or resources in a layer that is abstracted from the physical hardware underlying. Typical components of a computer system that can be virtualized are CPU, memory, storage, network devices, and I/O peripherals. Most commonly, these resources are combined into a virtual machine, a full-fledged computer environment on which an operating system can be executed: in this case, the machine used for virtualization is called host whereas the guest is the virtual machine. Software executed on the guest machine behaves as it was running directly on hardware, but its access to physical resources such as network, storage and peripherals can be, and often is, managed more strictly than the host.

This approach yields many advantages: for desktop users, virtualization can be

used amongst all for executing applications meant for a different operating system without the need to reboot into another system or switch computer. On the enterprise level, this technology allows cost mitigation by consolidating multiple servers in a larger physical one and therefore reducing effort spent in hardware acquisition and maintenance, and increasing resource utilization efficiency: the server compaction optimizes inconsistent workloads, resulting in considerable energy costs saving, and a much lighter footprint from an environmental point of view. Manageability is another advantage that a virtualized environment can offer: machines can be easily provisioned, inspected from a remote site and moved from a physical host to another. Moreover, their execution status can be saved and backed up persistently, allowing their use in disaster recovery scenarios deriving from faulty hardware or human error. Lastly, virtual machines create a sandboxed execution environment for the guest operating system and applications running on top of it, so that faults occurring cannot harm the host operating system, and applications do not have access to physical resources, enhancing the security of the infrastructure. Likewise, if a guest operating system becomes corrupted, for example from malware infection or installation of misbehaving software, the virtual machine can be effortlessly discarded or replaced with a previously backed up version.

Drawbacks of this technology include magnification of physical failures, because malfunctioning hardware can impact many different systems meant to be kept separate, and degraded performance[3]: virtualization overhead is not always negligible, and estimating how many additional resources are necessary is not trivial, the higher the number of servers pooled into a single physical host, the more the virtualization overhead increases. In this context, consolidation approach is valid because in enterprise data centers servers are for most of the time utilized for less than 30% of their available capacity, yet under peak demand, some of the hosts might be overloaded, with the consequence of increased latency and lower overall quality of service.

The firmware or software enabling creation and execution of virtual machines is called hypervisor, or virtual machine monitor, allowing a host to share its resources to support multiple guest operating systems. Historically it has been possible to make a distinction between two types of hypervisors, described visually in figure 2.1:

- **Type I**, or *bare metal* hypervisors: hypervisors running directly on hardware, consisting of a micro-kernel based operating system having direct access to physical resources by means of their own device drivers. Between commercial and open-source products belonging to this category, we can name *VMWare*

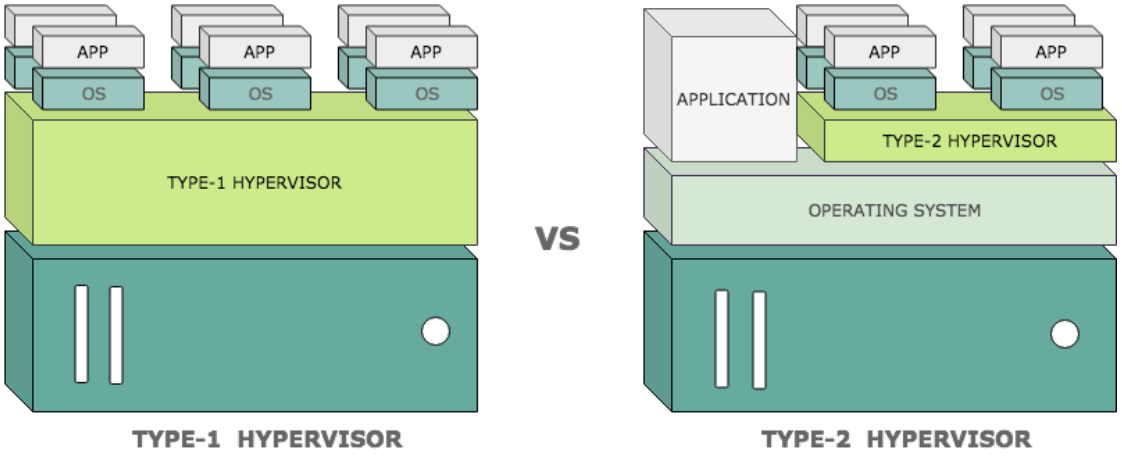

Figure 2.1: Type I and type II hypervisors.

ESXi, *Xen*, *Microsoft Hyper-V*.

- **Type II**, or *hosted* hypervisors: hypervisors running as a process on a general purpose operating system and requiring hardware access through its features. *VMware Workstation*, *VirtualBox*, *Parallels Desktop for Mac* and *QEMU* are type-2 hypervisors.

However, this categorization is not suitable to embrace all different types of software developed through the years: *Kernel-based Virtual Machine (KVM)*, for instance, is a Linux kernel module allowing the system to operate as a hypervisor; in this case, being part of the kernel, KVM has direct interaction with hardware as in bare metal virtual machine monitors, yet guests do share memory blocks, CPU instructions and other components of the underlying Linux operating system as in hosted virtualization platforms, hence it cannot be considered either a type-1 nor a type-2 hypervisor.

2.1.1 Lightweight Virtualization

Debated solutions involve running a separate kernel for each guest virtual machine, enforcing strong separation between environments and enabling the possibility of executing different operating systems on the same hardware. However, different technologies have been developed to achieve user-space process isolation without the need for a full system emulation. These solutions are commonly referred to with the

term of operating-system-level virtualization or containerization: they provide the operating system's kernel the capability of instantiating different isolated execution contexts, called containers, virtual environments, or jails, and can be considered as a lightweight alternative to traditional hypervisor-based solutions. Examples of container-based virtualization products are *Docker*, *OpenVZ*, *Solaris Containers*, *Linux LXC*, and *FreeBSD jail*.

Applications running in containers see only resources and devices associated with the container and have their own execution context, but share the kernel with the host operating system, as well as the operating system libraries. Using a container-based solution has different advantages:

- **Reduced CPU and memory overhead**[4]: operations execute at near native speed, resulting in applications performance advantage compared to hardware virtualization
- **Lower footprint**[5]: containers show lower memory and static image storage footprint by orders of magnitude, and severe decrease of guest OS initialization time, allowing easier back-up, snapshot, and live migration of virtual environments for resource optimization purposes.
- **Higher density**:[3] a much higher number of virtual instances can be executed concurrently on the same host without incurring in significant performance degradation.
- **Application packaging**: developers can use container technology to package in a single image all dependencies of an application and run it on a different environment with consistent results, allowing faster development lifecycle and improved productivity.

Container-based solutions present also disadvantages in comparison to hardware virtualization, and the most relevant are:

- **Security**: containers isolation is weaker than virtual machines; the kernel of the host system is shared between all containers, this means that a kernel vulnerability it can jeopardize the security of the other containers as well, while in a virtual machine security threats such as malware and intrusions are not able to spread over.

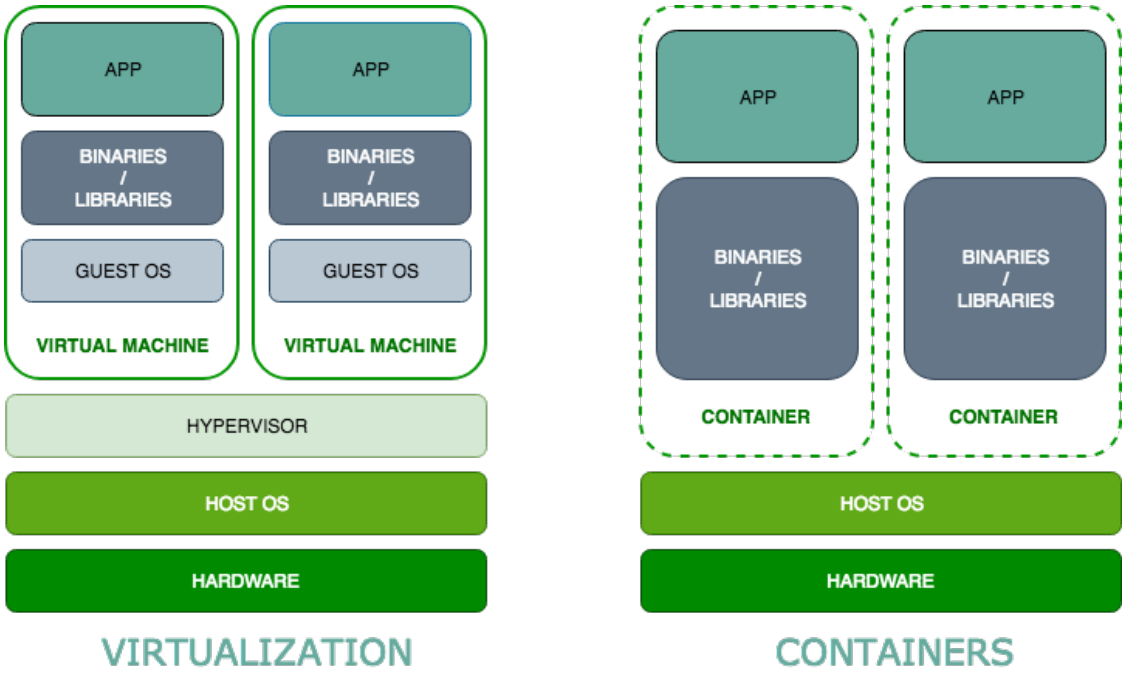

Figure 2.2: A comparison between virtualization and containers technology.

- **Operating systems**: since the hardware access of container operating system happens through the host kernel, it is not possible to run different operating systems on the same host, which could be an issue for complex enterprise applications.
- **Networking**: achieving adequate networking capabilities with sufficient isolation and security is a complex issue.

With the advent of Docker in 2014[6] and its exponential growth in popularity, container-based solutions and their ecosystems are experiencing a tremendous momentum and a revolution in how applications are packaged and released, challenging traditional hypervisor-based virtual machines infrastructures, also encouraging application design paradigms such as service-oriented architectures and microservices.

2.2 Cloud Computing, SDN and NFV

Virtualization has been the main enabler technology for a new paradigm which aims to transform computing into a commodity-like service, delivered the same way traditional utilities, electricity, gas, water, and telephony are. The name *Cloud*

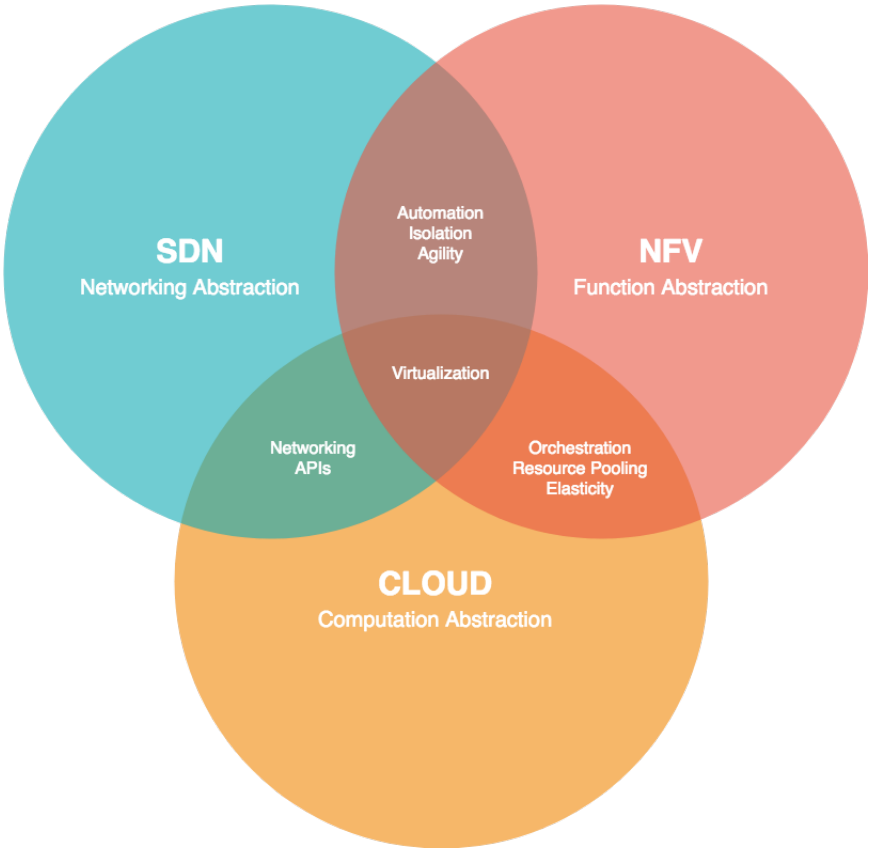

Figure 2.3: Relationship between NFV, SDN and Cloud Computing concepts.

Computing for this model was made popular in 2006 when *Amazon.com* created its subsidiary company *Amazon Web Services* and advertised its product Elastic Compute Cloud (EC2)[7], even though the cloud has been a symbol of distributed network computing since ARPANET[8].

2.2.1 Cloud Computing

According to National Institute of Standards and Technology (NIST)[9] cloud computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction". In this context, the "cloud" represents the infrastructure to which users, both private and enterprise, are able to access on demand from anywhere in the world. The cloud computing paradigm, in

the NIST perspective, defines five essential characteristics:

- **On-demand self-service**. Computing resources and services are provided as autonomously requested by the user, automatically without the need for human intervention from the service provider.
- **Broad network access**. The network is the mean enabling access to offered computing capabilities, which are available for the consumer through standardized protocols and mechanisms allowing usability from multiple platforms and clients.
- **Resource pooling**. Provider's resources are collected in pools and allocated and released dynamically to multiple consumers in a multi-tenant model. The consumer has no perception of where the resources are located, but might be able to choose a geographical macro area, such as country, region or datacenter.
- **Rapid elasticity**. The consumer can provision and release resources to meet application demands, growing or shrinking allocated computing elastically.
- **Measured service**. Consumed resources are monitored and measured, and can be controlled and reported transparently for both the user and the service provider.

Moreover, the service can be offered in different models to meet the needs of the customer:

- **Software as a Service (SaaS)**. The consumer can use the provider's application running on a cloud infrastructure, accessing it from anywhere with a thin client, such in the case of a web-based application, or a software interface, without the possibility of monitoring and managing the underlying infrastructure.
- **Platform as a Service (PaaS)**. The provider allows access to a platform on which the consumer can deploy applications built using tools supported by the provider, having access to application-specific settings and environment configuration, but not the underlying infrastructure.
- **Infrastructure as a Service (IaaS)**. The consumer can provision computing resources as processing, storage, and networks from the cloud infrastructure to deploy and run software appliances. The consumer has control over operating systems, storage, and most aspects of networking.

2.2.2 Software-Defined Networking

With the advent of cloud computing, has become even clearer how traditional IP networks are hard to manage and to maintain. In a context of multitenancy, in which every consumer's resources must be completely isolated although existing in the same infrastructure, and configure quickly in response to customer needs, the providers' need is to express network policies based on different users and services. Being the IP network distributed by design, each individual device should be configured in order to set up a network policy, often by hand using vendor specific commands, or by executing scripts; in either case, human intervention is essential, making this task error-prone and not scalable. Moreover, in a cloud environment, connections should be able to be resilient to faults and adapt to changes in traffic load: traditional networks have no means of feedback reconfiguration. In addition, networking devices integrate together both control plane, which decides how to route the traffic, and data plane, that forwards packets according to decisions taken at the control plane level, further reducing flexibility, and increasing costs and effort necessary to operate an IP network.

To address these issues, a new network paradigm has been proposed: Software-Defined Networking (SDN) is a network architecture that decouples control logic from underlying physical devices forwarding the packets, introducing a centralized logic controller able to communicate and dynamically configure and manage routers and switches, which expose a common and well-defined management interface (API) and become plain forwarding devices. The core of this architecture is the SDN Controller, which centralizes the logic of the traffic routing and offers an abstract view of the network topology and devices, including events and metrics; exposes a Northbound Interface to enable application and orchestration systems to interact with the network and its data path policies. The controller, through the SDN Control to Data-Plane Interface (CDPI), is able to translate application requirements to programmatically control forwarding rules, send and receive events notifications and collect statistics through a Southbound Interface. One of the values of the SDN architecture is that expects this interface to be standardized, hence every vendor would be able to introduce compliant forwarding devices.

By leveraging the capabilities of SDN architecture, cloud providers are able to solve the issue of multitenancy in network infrastructures, as IP and Ethernet technology have virtual network capabilities but limited in terms of number of supported tenants and isolation between them[10], to ensure per-tenant service level as minimum bandwidth and latency guarantee, and to offer Networking-as-a-Service

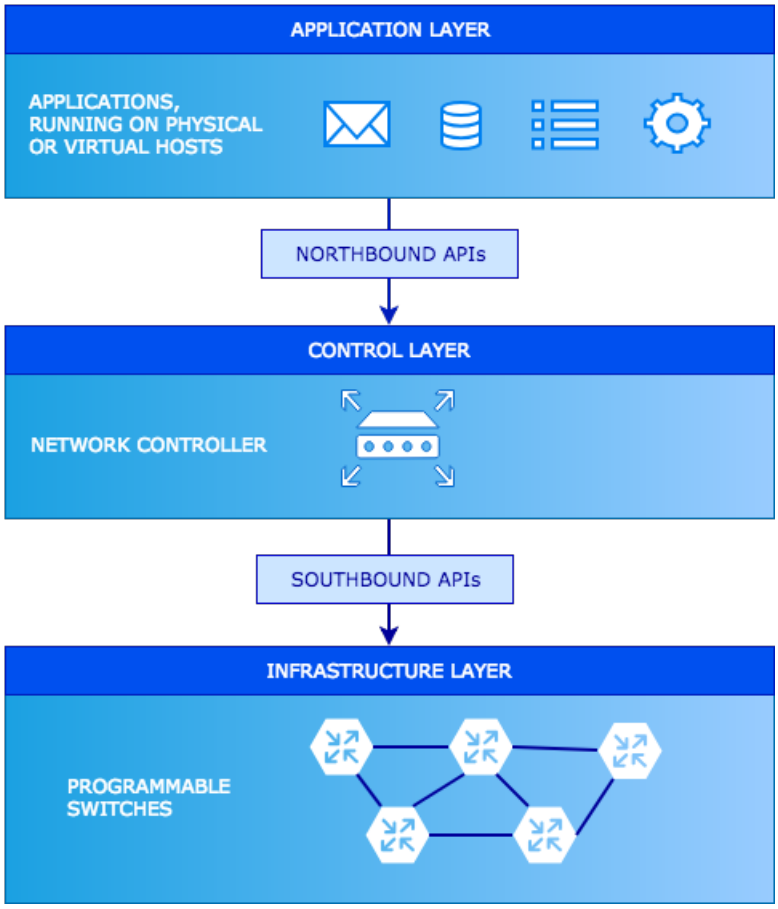

Figure 2.4: Type I and type II hypervisors.

capabilities.

2.2.3 NFV and relationship to Cloud Computing and SDN

The same need for flexibility, agility, and reduction of capital and operating expenses that inspired the birth and growth of the SDN architecture, pushed telecommunications service providers (TSPs) into building a consortium to develop a new concept of networking, founding the European Telecommunications Standards Institute (ETSI). Taking advantage of virtualization technology, and using concepts deriving from cloud computing, Network Function Virtualization (NFV) architecture abstracts networking capabilities from the hardware effectively performing them, increasing resource utilization efficiency and reducing the need for special-purpose

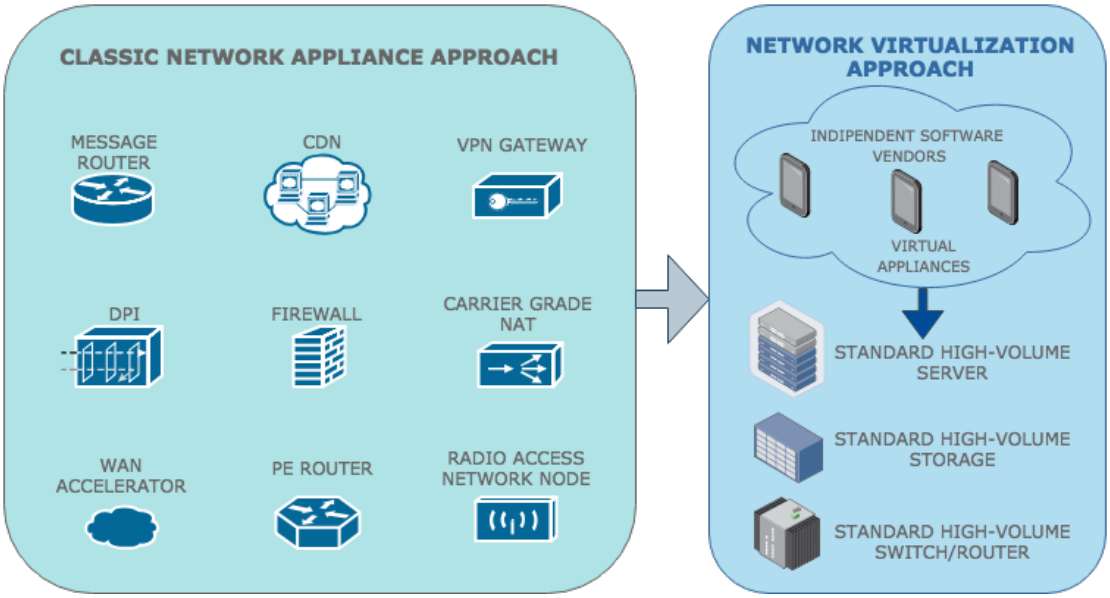

Figure 2.5: Transition from dedicated hardware devices to virtualized network functions.

hardware. In the original white paper[11], ETSI declares the problems that NFV aims to solve, proposing the NFV solution:

> Network Operators' networks are populated with a large and increasing variety of proprietary hardware appliances. To launch a new network service often requires yet another variety and finding the space and power to accommodate these boxes is becoming increasingly difficult; compounded by the increasing costs of energy, capital investment challenges and the rarity of skills necessary to design, integrate and operate increasingly complex hardware-based appliances. Moreover, hardware-based appliances rapidly reach the end of life, requiring much of the procure-design-integrate-deploy cycle to be repeated with little or no revenue benefit. [...]
>
> Network Functions Virtualisation aims to address these problems by leveraging standard IT virtualization technology to consolidate many network equipment types onto industry standard high volume servers, switches and storage, which could be located in Datacentres, Network Nodes and at the end user premises. We believe Network Functions Virtualisation is applicable to any data plane packet processing and control plane function in fixed and mobile network infrastructures.

In this way, a network service can be decomposed in a series of elementary Virtual Network Functions (VNFs), each running as a virtual instance on Commercial-Off-The-Shelf (COTS) hardware, becoming building blocks that may connect or chain together to create required services. The same network functions can be effortlessly scaled independently with fine granularity to meet load requirements as they evolve, as well as relocated in a different geographic area to provide services targeting customers in a specific location. Examples of use cases for NFV technology include virtualized load balancers, CDNs, tunneling devices as VPN gateways, security functions as firewalls or intrusion detection systems, and traffic analysis elements as DPIs (figure 2.5).

Network Functions Virtualization takes advantages from cloud computing technologies: virtualization through hypervisors and containers as enabling mechanics for functions decoupling, as well as utilization of virtual ethernet switches; cloud infrastructures can be leveraged to orchestrate and manage virtual appliances, in order to ease instantiation, back-up and restore, snapshot, migration, and termination of virtual machines executing network functions, management of image catalog and storage attached to instances, and creation of virtual networks and network interfaces.

At the same time, Network Functions Virtualisation has a strong relationship with SDN: both architectures can be implemented without being dependent one from the other, but the two concepts can be combined and potentially achieve greater synergy. NFV can benefit from the concepts of separation of data and control plane defined from SDN, simplifying operations, increasing manageability, and enhancing performance. On the opposite side, SDN can take advantage of Network Function Virtualization running virtualized SDN-compatible devices, further strengthening the concepts of SDN of interoperability, agility, and vendor neutrality.

2.3 ETSI NFV Architectural Framework

European Telecommunications Standards Institute (ETSI) established in late 2012 the Industry Specification Group for NFV (ETSI ISG NFV), in charge of defining the requirements and guidelines for the NFV paradigm. One of the most important deliverables produced by this group can be identified in the Architectural Framework specification document[12]. In this paper, ETSI described the actors involved in the NFV context, and a reference architectural framework, outlining the different

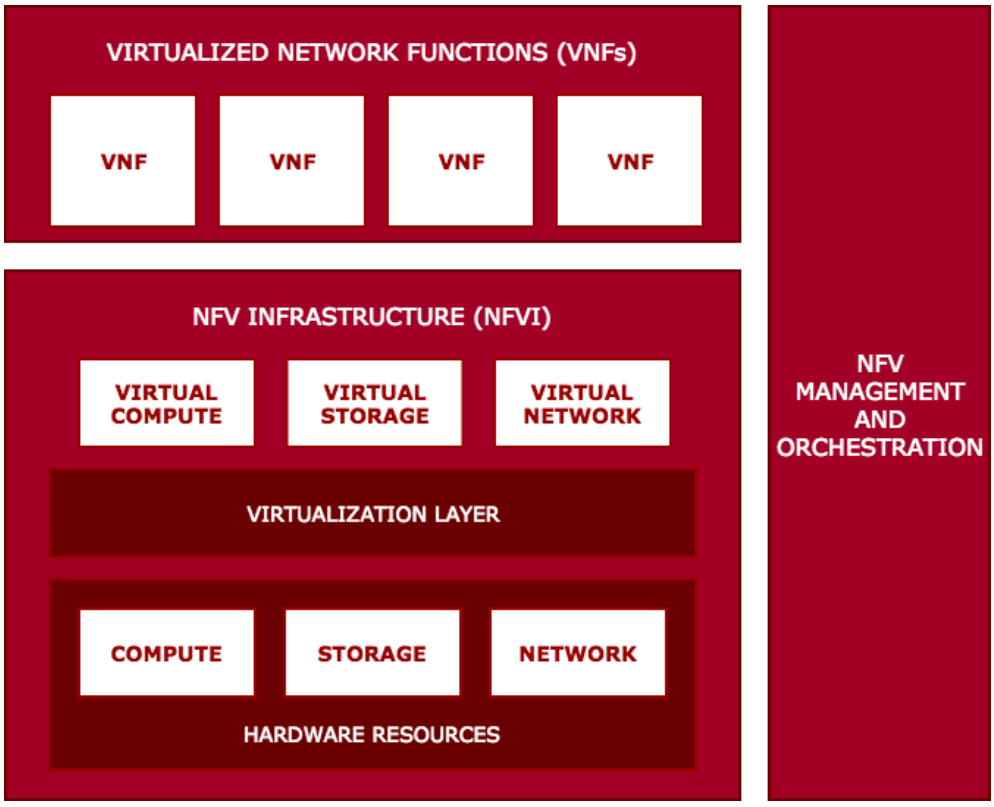

Figure 2.6: An overview of ETSI NFV architecture.

components of the framework and the connections and interfaces between them, with the aim of creating a modular ecosystem driven by open and standardized interfaces; in this way, every building block can be supplied by a different producer, enabling a multi-vendor interoperable NFV solution.

As visually described in figure 2.6 the Architectural Framework document delineates three core domains:

- **Network Function Virtualization Infrastructure (NFVI)**. The environment providing computing, storage and network capabilities on which Virtual Network Functions are instantiated; includes both physical resources such as servers, network-attached storage and switches, and software, hypervisors, operating systems, and virtual infrastructure managers. Characteristics of NFVI will be detailed in section 2.3.1.

- **Virtualized Network Functions (VNFs)**. A Virtualized Network Function is the basic component of the NFV architecture and represents the abstraction

of a network function running in software as a virtual appliance in the NFVI. VNF structure, features, and relationships between VNFs will be further described in section 2.3.2.

- **NFV Management and Orchestration (MANO)**. The MANO framework takes care of provisioning, operation, and lifecycle management for VNFs interacts with NFV Infrastructure and with the virtualization layer in order to instantiate necessary virtual resources, interfaces with VNFs for configuration and reporting. Development of NFV MANO framework is supported by a working group of the ETSI organization, and for the purposes of this thesis work requires an in-depth analysis, which will be addressed in section 2.3.3.

2.3.1 Network Function Virtualization Infrastructure

With Network Function Virtualization Infrastructure, ETSI defines the context in which VNF lifecycle takes place, from instantiation to operation and dismission. It builds up of different hardware and software components that allow VNF execution:

- **Hardware resources**. The physical resources including processing, storage, and connectivity that allow execution and communication between VNFs. The computing resources are assumed to be COTS hardware, instead of special-purpose dedicated hardware; the same way, storage could be supplied by a storage area network (SAN) or by disks residing on the servers, commonly pooled to offer an abstraction of storage capacity. Networking resources include high-capacity L2/L3 switches, routers, and links, and can be distinguished in two main types[12]: the NFI-PoP network, interconnecting resources internal to a PoP, and the transport network, connecting PoPs between them, to other networks owned by third parties, or other network appliances outside NFVI network.

- **Virtualization Layer**. The virtualization layer provides an abstraction between VNFs and hardware, allows partitioning and access to physical resources, allowing an hardware-independent lifecycle for VNFs. This virtualization services can be provided by essentially two means: an hypervisor-based solution, in which each VNF has its own operating system inside a virtual machine (VM), or a lightweight virtualization solution (see 2.1.1), allowing the execution of each VNF in a separated and sandboxed context, but with a lower

degree of isolation. NFV framework does not constraint the virtualization solutions which can be used for the network functions, is also possible to leverage different technologies according to the specific function to be virtualized. Benefits of using containerization for VNFs are several: a much faster startup time, which could be critical when instantiating services which need to be created and deleted in a short amount of time, or autoscaling policies with dynamic load; I/O performance is close to bare metal, critical for carrier-grade applications, even if new virtualization technologies allow virtual machines to access hardware directly in passthrough mode to avoid these issues; and an overall higher flexibility for the network service, at the cost of less isolation and a higher attack surface from the security point of view.

- **Virtualized Infrastructure**. On top of the latter layer, the virtualized infrastructure encloses all resources provisioned to support network functions including virtual computing, storage, and networking, which is abstracted from the hardware level to provide connectivity between VNFs based on virtual network paths. This is allowed by means of multiple techniques, like Virtual Local Area Network (VLAN), an overlay network, and encapsulation protocols, often implemented using SDN techniques.

NFV Architecture is designed to be distributed, therefore the NFVI can be present in a single physical location or can stretch across multiple sites, in order to satisfy requirements of latency and geographical locations that might exist when designing a network service; in this case, the network connecting different sites is considered part of the infrastructure. From the VNF point of view, the different components of the NFVI appear as a single entity providing required resources, offering contact points through which it is possible to request resource allocation in the form of Points of Presence (PoPs).

2.3.2 Virtualized Network Functions

A Virtualized Network Function is, according to ETSI, "virtualization of a network function in a legacy non-virtualised network"[12]. Examples of network functions that could be virtualized are firewalls, NAT devices, VPN gateways, Dynamic Host Configuration Protocol (DHCP) servers, or WAN routing devices. A chain of Network Functions, whether virtualized or not, represents an end-to-end Network Service (NS), which solves a functionality for the end user and will be addressed more specifically in subsection 2.3.2.1. From the user's perspective, and from the point

of view of the NS, there is no difference between Virtualized Network Functions and Physical Network Functions, the functional aspects and external interfaces are assumed to be identical, so these components could be interchangeable; the same applies to VNF capabilities, degradation of performance introduced by the virtualization layer must be largely negligible, so that delivered behavior can be equivalent in case of a virtualized or non-virtualized environment.

A VNF is not always a monolithic block of software, yet could be composed internally by different internal components communicating together. When this is the case, these components may be running each in a different virtual machine, or in a single one, depending on the VNF implementation.

VNFs are executed on top of the NFV Infrastructure virtualized infrastructure; necessary resources and connectivity are provisioned through NFI-PoPs. NFV architecture highlights the need for the end-to-end service to be completely unaware of the physical placement of VNF instances, given that requirements for redundancy, high availability and failover, and geolocation awareness as in a distributed cache or CDN, are satisfied. In this way, VNF instances could be geographically stretched across multiple datacenters, as well as implemented using physical devices, different hypervisors, and virtualization techniques as long as service requirements and constraints are met.

2.3.2.1 Network Services and VNF Forwarding Graph

The description of an E2E service goes by the name of Network Service (NS) and can be achieved by means of a list of interconnected Network Functions, and one or more VNF Forwarding Graphs (VNF-FG). A VNF-FG represents a path which packets follow in a network service; it is composed by a set of nodes corresponding to network appliances, devices or server applications, and logical links, which could be unidirectional or bidirectional, unicast, multicast or broadcast. Each Network Service can have multiple VNF-FGs spanning its set of nodes, each of them implementing a different subset of capabilities for the service: as an example, there could be different kind of traffic, such as user traffic and control traffic, which need to be routed in different paths across VNFs, yet both necessary to implement the same service for the end users; this scenario would require two different forwarding graphs for a single NS. Nodes of a VNF-FG are Network Functions, while logical links are implemented on top of physical links provided by the NFV Infrastructure often by means of Service Function Chaining, the capability of routing packets through

logical links instead of using traditional L3 routing techniques.

2.3.3 NFV MANO Architecture

The introduction of a conceptually innovating network paradigm such as NFV presents a new set of challenges, concerning service to NFV network mapping, VNF instantiation and orchestration, resource allocation and management, metrics collection and reporting, which need to be addressed in order to have a successful deployment and operation of the networking model. For this purpose, ETSI created since the early stages a working group to build a comprehensive reference framework to ensure compatibility between different vendor solutions and to create a standardized approach to deal with NFV management.

NFV MANO architecture describes different building blocks, which communicate with each other through well-defined interfaces; the figure 2.7 provides a bird's eye view over MANO components and interfaces.

In this picture, it is clear the extent of MANO framework, its reference points between internal components, and the interactions with external elements: there are three main constituents of the architecture:

- *NFV Orchestrator (NFVO)* (2.3.3.1).
- *VNF Manager (VNFM)* (2.3.3.2).
- *Virtualized Infrastructure Manager (VIM)* (2.3.3.3).

Each component solves a specific function and will be detailed in its own subsection. The interfaces listed are standardized and can be used to decouple the different components in a microservice-based architecture. Moreover, NFV Management and Orchestration can be used in conjunction with existing management systems, such an Operation Support System/Business Support System (OSS/BSS), used to support various end-to-end telecommunication services, inventory, and service lifecycle, an NFV Infrastructure Manager, often identifiable with a Cloud Management System, and other tools able to support telecommunication service providers.

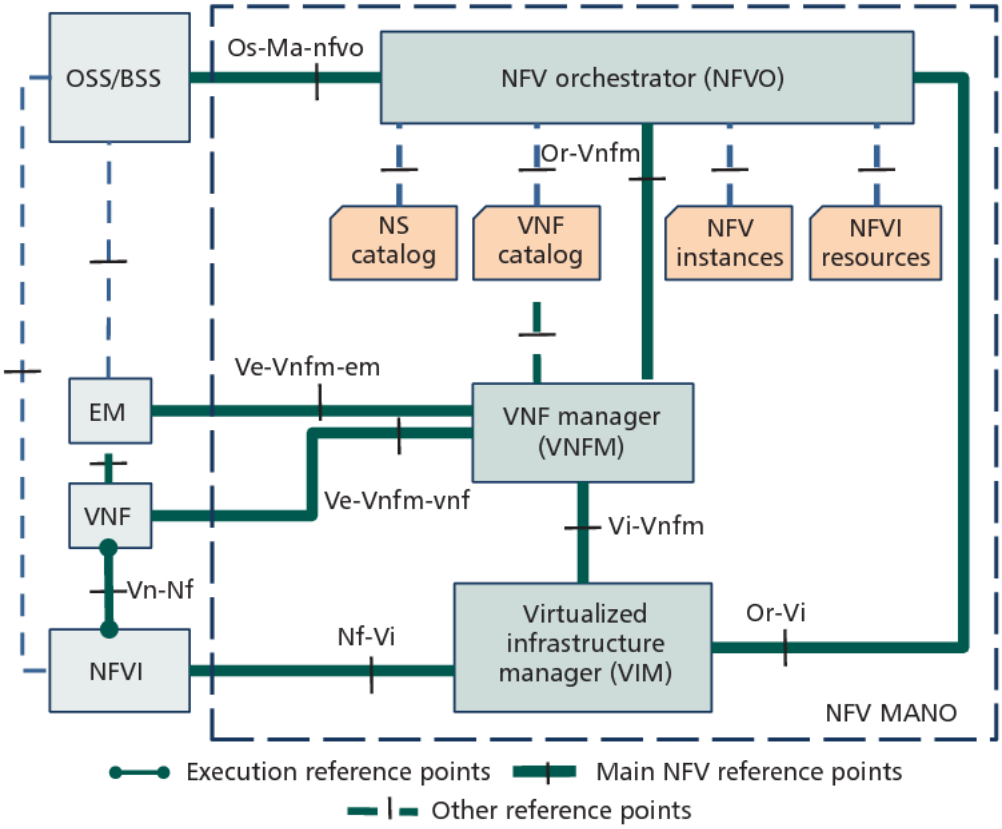

Figure 2.7: NFV MANO architecture overview [13]

2.3.3.1 Network Function Virtualization Orchestrator (NFVO)

The NFV Orchestrator is the front-end for the user to interact to: provides user interfaces and APIs for integration with existing software components. It is in charge of managing different critical aspects of NFV architecture:

- *Catalog.* The NFVO is responsible for onboarding and management of descriptors catalogs. In detail, manages Network Services definitions and linked resources, VNF descriptors and service topologies in the form of VNF-FGs.
- *Network Service Lifecycle.* Manages Network Service instantiation, termination, upgrade and scaling in and out. In order to offer an end-to-end network service, the NFVO interacts with and coordinates VNF instantiation and lifecycle, which might be managed by different VNF Managers; it is up to the NFVO to monitor each VNF to establish network service status and eventually

proceed to VNF auto-healing.

- *Resource Orchestration.* It is in charge of engaging the NFV Infrastructure through the infrastructure manager to allocate and release resources and connections necessary to the VNF instantiation.

2.3.3.2 Virtual Network Function Manager (VNFM)

The VNF Manager is used to control, manage and monitor the VNFs lifecycle under the direction of the NFVO, in particular, instantiation, starting and stopping, upgrade, scaling in or out and termination of VNF instances. Controls also the Element Management System (EMS) and the Network Management System (NMS). If required, there could be more than one VNF Manager for each datacenter, each managing its domain of VNFs, or a single VNFM could be used for multiple VNFs.

2.3.3.3 Virtualized Infrastructure Manager (VIM)

The Virtualized Infrastructure Manager is responsible for virtual resource management, coordinated by the NFVO. In particular, the VIM deals with:

- *Resource management.* The VIM is in charge of allocating, and releasing resources from the NFVI when requested by the NFVO; keeps track of used resources and their physical allocation, optimizing hardware usage. Manages the pooling of hardware resources, and the list of available virtual and physical resources available.
- *Networking.* Provides the networking infrastructure supporting the VNF Forwarding Graphs in the form of virtual links, networks, and subnets.
- *Image management.* Manages the catalog of software images available to the NFVO, and allows creation, update, and deletion of images.
- *Reporting.* Collects metrics about performance and faults, making information available when requested from the NFVO.

2.3.4 ETSI Information Model

The loosely coupled architecture proposed from ETSI aims to promote the development of a multi-vendor ecosystem in which every component evolves independently

from the whole system. In this context, standardized interfaces and a proper interaction model between different entities are critical since the beginning of the paradigm introduction, and so is a coordination framework such as MANO, which grants interoperability and avoids vendor lock-in. For this reason, ETSI developed a reference information model[14]. This document defines the data structures necessary for service definition and instantiation dividing them into three different categories of information:

- *Descriptors.* Descriptors contain static deployment information provided as input to the orchestration system, used for resource template onboarding and instantiation.
- *Records.* Records represent resource instances; they include static data from descriptors, integrating it with run-time information from the environment and updates about resource status and lifecycle.
- *Exchange information.* Information flowing between components interfaces.

Descriptor templates created by the user are added to a catalog, specific for each different descriptor category, through the onboarding operation. Upon instantiation, descriptors are enriched with user-provided data which could be used for instance customization and configuration, and with information exchanged through orchestrator's functional blocks. As a result of the instantiation process, the representation of created resources is obtained through the creation of a record, which models the current state of newly created instances, and is updated upon property changes. In the following sections, the different possible descriptors and their corresponding records are outlined in detail.

2.3.4.1 Network Service

The root of the information model is represented from the Network Service Descriptor (NSD), which is the deployment template for the Network Service. It contains information concerning service properties, such as external endpoints definition, auto scale and security policies, and references to the different elements descriptors composing the service, which can be of four types:

- *Virtualised Network Function Descriptors (VNFDs)* (2.3.4.2).
- *Virtual Link Descriptors (VLDs)* (2.3.4.2).

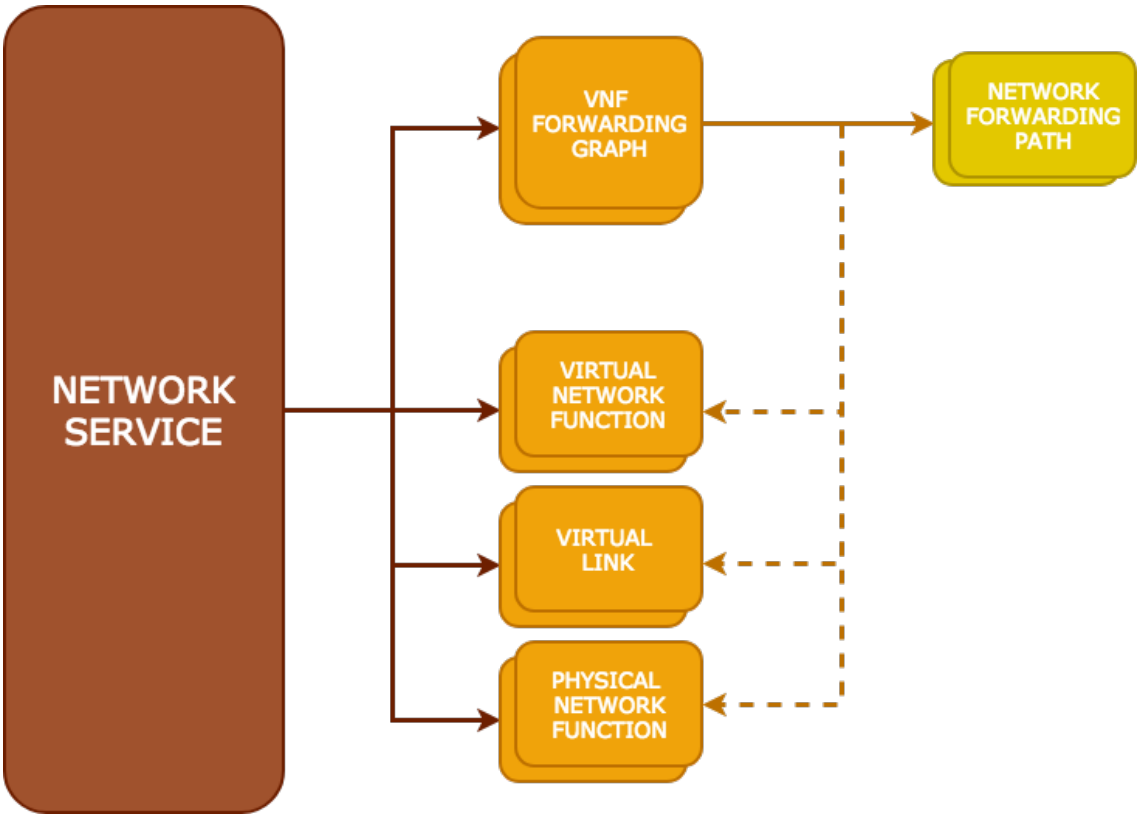

Figure 2.8: ETSI MANO information model representation.

- *Physical Network Function Descriptors (PNFDs)* (2.3.4.3).
- *VNF Forwarding Graph Descriptors (VNFFGDs)* (2.3.4.5).

Upon instantiation, a Network Service Record (NSR) is created, the most relevant information the NSR contains is:

- *Network Service Descriptor (NSD).* The NSR holds a reference to the NSD used to create the instance.
- *Status and history.* The NSR keeps track of the status of the instance, as well as the lifecycle events history and logs.
- *Records references.* Subcomponents records are stored in the NSR to keep a reference to service instances.

2.3.4.2 Virtual Network Function

A VNF Descriptor (VNFD) represents a VNF template and its deployment requirements, dependencies, and resources. It is used by the VNFM during instantiation of the VNFs and lifecycle management, and by the NFVO to allocate and instantiate necessary resources from the NFVI for each VNF instance. Among the most important information contained are:

- *Virtual Deployment Unit (VDU).* A Virtual Deployment Unit describes the hardware characteristics of the virtualized instance the VNF will be deployed on. It contains information about the CPU and memory requirements and the virtual machine image to be deployed; in addition, other constraints can be specified, about the hypervisor type, storage, network interfaces, PCIe connections, processor operating frequency, memory speed, and others. These parameters, although not required for instantiation, could be useful for deployment optimization.

- *Connection Points (CP) and Internal Virtual Links (IVL).* Connection Points represent the accessibility of the VNF through a Virtual Link. If the VNF is composed of different subcomponents, Internal Virtual Links can be defined and specify the connections between them. A Connection Point defines an interface of the VNF over a subnet and can be attached to a Virtual Link defined in the VLD section at the network service level or even to an IVL.

- *Lifecycle Hooks.* Defines scripts or actions to execute upon lifecycle events, such as initialization, termination, scaling in or out, upgrade. Is used from the VNF Manager to instantiate lifecycle management for the VNF.

- *Dependencies.* Describe dependencies between VDUs, in order to define an instantiation order between VNFs.

Upon instantiation, a VNF Record is created to track VNF instances and relative allocated resources, allowing cross-reference between all resources involved. This record includes all static data coming from the VNFD, in addition to runtime information about the VNF, such as status and network address, and VNFC instances.

2.3.4.3 Physical Network Function

If a physical device needs to be included in the Network Service, this can be described using a Physical Network Function Descriptor (PNFD). This can be used by the NFVO to create necessary links between VNFs and the existing physical interfaces, together with the required virtual link KPIs such as latency and throughput. Information included within the PNFD describes the network requirements and connectivity needed to integrate the PNF in the service and contains the Connection Points that the PNF exposes, and the definition of the Virtual Links these CPs connect to.

After instantiation, a PNF Record is created to index the created networking resources and linked PNFs.

2.3.4.4 Virtual Links

A Virtual Link Descriptor (VLD) is a template descriptor used by the NFVO to instantiate network infrastructure based on the connectivity requirements between VNFs, PNFs, and Network Service external endpoints. The information contained is used to determine the type of connection to instantiate, and the VIM responsible for service provisioning based on requirements. Three different connection types have been identified in the NFV context: E-Line, E-LAN, and E-Tree. An E-Line connection represents a simple point-to-point link between the existing network and the VNF; an E-LAN acts a local network on which VNFs can exchange information between each other; finally, an E-Tree is a rooted multipoint virtual connection in which leaf nodes cannot exchange data between each other, useful in case of load balancing services, internet access or video over IP. The VLD provides information about the connectivity topology (E-Line, E-LAN or E-Tree), bandwidth parameters, Quality of Service requirements (latency, jitter, etc...) and references to Connection Points attached to the VL.

Virtual Link Records are stored after deployment to keep track of instanced links and their status, allocated capacity, and the references of Network Services and VNFFGRs they participate to.

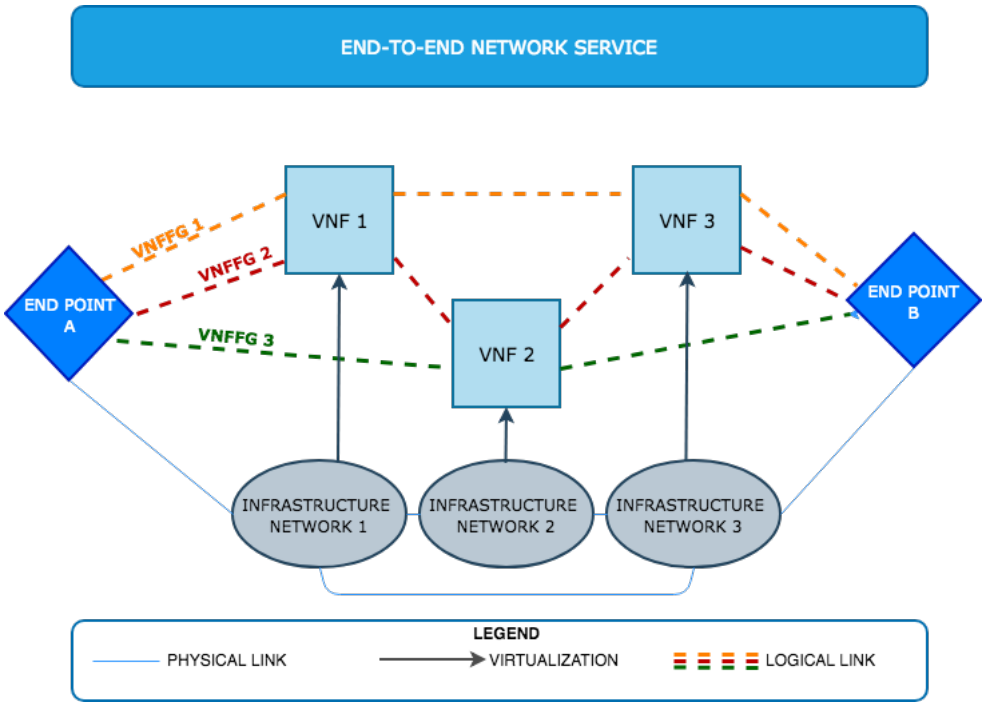

Figure 2.9: Multiple VNFFGs in end-to-end network service.

2.3.4.5 VNF Forwarding Graphs

VNF Forwarding Graphs define traffic flows through VNFs in terms of a set of Network Forwarding Paths (NFP), which represent a chain of VNFs, together with policies for which it can be traversed by traffic. Each VNF-FG can include one or more NFP, which control the actual packet flow. A VNF Forwarding Graph Descriptor (VNFFGD) defines a template used from the NFVO for VNF-FG instantiation. It references dependent NSD elements such as VNFDs, PNFDs and VLDs, and relative Connection Points to describe the constituent components of the forwarding graph, and at least two Network Forwarding Path Descriptors (NFPD) determining the traffic routing and policies. Each NFPD defines a policy to apply and a set of references to Connection Points to be joined.

Upon setup of a VNF-FG, a VNFFG Record (VNFFGR) is created to store graph instance information.

Chapter 3

Requirements Definition and Analysis

As described in previous chapters, the NFV paradigm represents a promising solution for software and hardware decoupling in complex networks, yet it cannot be implemented disregarding a management framework such as NFV MANO. The reference MANO implementation makes use of user-defined templates called descriptors which have to be composed and validated by hand by an administration user before onboarding on the orchestrators' catalogs, and lack the capability of analysis of these descriptor templates, as well as the possibility of smart deployment of the solutions taking into consideration the NFV Infrastructure and service topology. These thesis work aims to develop a solution able to enhance NFV MANO capabilities to fulfill resource allocation optimization considering underlying infrastructure, and service graph formal verification, to ensure that once deployed, the network service will behave as expected, and produce a deliverable which will be used both a proof-of-concept and as a starting point for future development scenarios.

In the following sections, an outline of the functional requirements will be presented, together with an overview of the tools to be integrated into the development, Verifoo (3.2), and Open Baton (3.3). Finally, an analysis of possible approaches will follow, unitedly to the evaluation of their strengths and weaknesses, and the solution implementation choices.

3.1 Objectives definition and functional requirements

This thesis work objective is to produce a deliverable able to extend a MANO orchestrator capabilities in order to produce an optimized deployment. In order to pursue this goal, the solution to be implemented should comply with the following requirements:

- **Verifoo integration**. The solution should interact with Verifoo, a service graph optimization tool described in section 3.2, transmitting data related to service graph and NFV infrastructure.
- **Graph validation**. Graph verification performed by Verifoo should be used for service graph validation and formal reachability assurance. Deployment of an invalid graph instance will be denied, or a notification will be produced, informing the user.
- **Deployment optimization**. The deployment should be optimized according to Verifoo VNF-FG analysis and available NFVI resources, both by using configuration information provided by Verifoo and considering NFVI resources allocation.
- **MANO interaction**. The solution should be able to interact with an open source MANO NFV orchestration software, in order to enrich its features with service analysis and optimization. In the context of this thesis work, the product chosen to implement the solution is Open Baton, which will be detailed in section 3.3.

3.2 Verifoo

Verifoo is an open source software developed at Politecnico di Torino, a "Verification and Optimization Orchestrator component for joint Service Graph mapping and verification", and has been designed to be the validation and verification engine behind an NFV orchestrator. It makes use of the z3 library, a Satisfiability Modulo Theories (SMT) solver developed at Microsoft, in order to solve the Virtual Network Embedding (VNE) problem. It offers a RESTful interface through which is possible

to input the instance topology in order to compute policy validation and deployment satisfiability and scheduling based on the infrastructure specifications.

3.2.1 Information Model

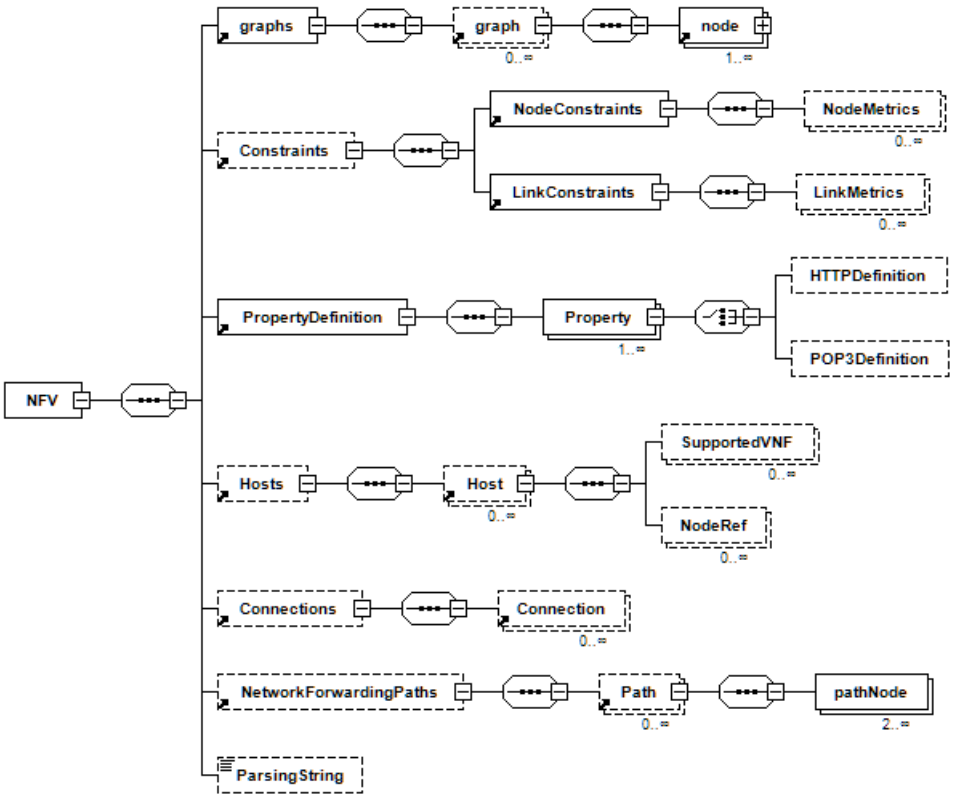

Figure 3.1: Verifoo NFV diagram

Verifoo makes use of XML Schema Definition language to describe its data model. Figure 3.1 shows a view of the main component of the information model, the *NFV* object, and its subcomponents. Elements drawn in solid lines are required, while dashed-line is optional, and under boxes identifying multiple cardinality items, minimum and maximum occurrences are shown. For space constraints, properties of the *node* object have not been shown and are compressed into their parent node, specifying it as a plus sign on the side: due to its considerable importance in the information model, this object will be represented in figure 3.2. Follows a detailed description of each NFV child element.

3.2.1.1 Graphs

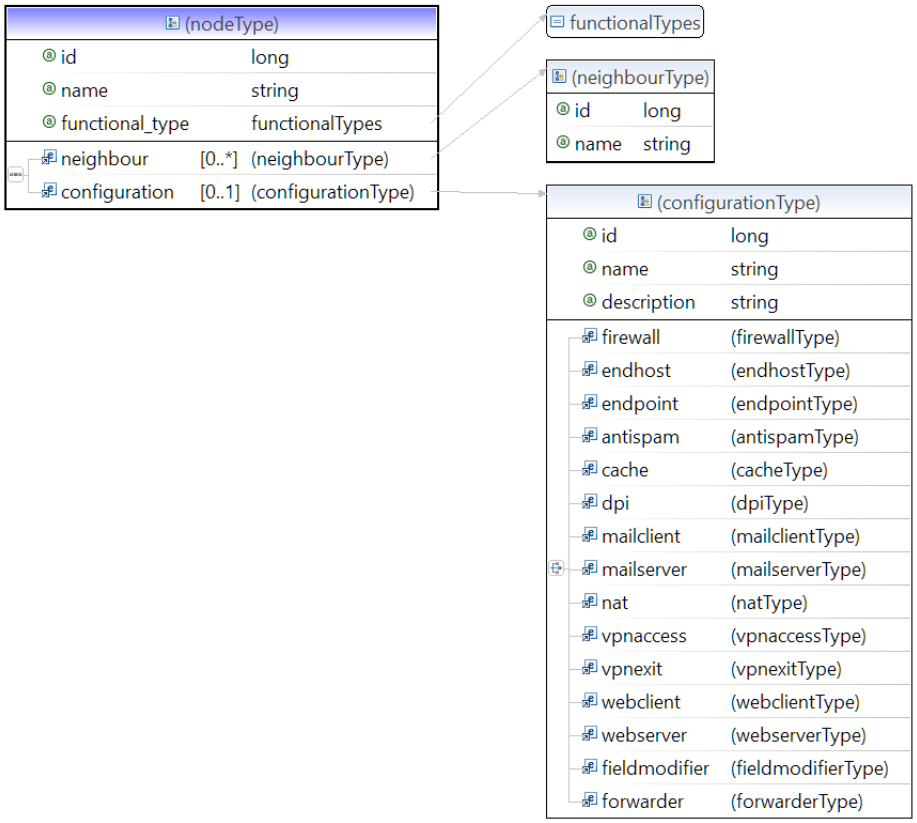

Figure 3.2: Node element diagram

The *graphs* element models a multiplicity of VNF-FGs in an NFV environment. It contains a set of *graph* elements, each of them contains a list of nodes composing the graph. The *node* object represents a VNF: as can be seen from figure 3.2, it contains a configuration object, and a list of neighbor nodes, in order to form a chain. Node attributes *id* and *name* identify the node in the graph, while *functional type* represents an enumeration of the VNF types supported by Verifoo. The *configuration* object child element assumes a different value depending on the functional type of the VNF the node is modeling. The node element functional type can be one of the following, each corresponding to its own configuration type:

- *FIREWALL*: models a simple ACL-based firewall VNF; its configuration is composed by a list of elements representing an ACL (Access Control List). Each ACL defines a couple of source and destination nodes, and the action that will be taken upon connections from source to destination, which could be

blocking or allowing packets. Optionally, additional properties can be defined to increase traffic control granularity, such as considered protocol, source, and destination ports, and if the ACL is effective in one direction or in both.

- *ENDHOST*: An end-host represents a network node which exchanges traffic with a destination node in the network. It serves as a generic model of a node which is the beginning or the end of a VNF-FG chain.

- *ENDPOINT*: A generic node which can represent an edge entity in the forwarding chain, for example, a user client.

- *ANTISPAM*: models an anti-spam VNF which can be placed in the chain connecting an email client with the corresponding server. It is configured in order to reject POP3 packets in which the sender address corresponds to one of the *source* configuration elements.

- *CACHE*: simple web cache modeled using the concepts of external and internal networks. Cache configuration includes the addresses of all nodes belonging to the internal network that should be cached.

- *DPI*: represents a Deep Packet Inspection (DPI) network function which can be configured to drop traffic including in the body a set of blacklisted strings.

- *MAILCLIENT*: an email client derived from the end-host model, which can only send *POP3_REQUEST* or *SMTP_REQUEST* messages, and receive *POP3_RESPONSE* or *SMTP_RESPONSE*. The destination mail server must be specified in its configuration.

- *MAILSERVER*: a particular form of end-host which represents an email server, receives *POP3_REQUEST* or *SMTP_REQUEST* messages from an email client, and can generate only *POP3_RESPONSE* or *SMTP_RESPONSE* messages, depending on the request type, addressed to the origin of the requests.

- *NAT*: models a Network Address Translation function. As the web cache, needs the notion of internal and external networks: in its configuration is listed a set of private addresses that represent the hosts in the internal network.

- *VPNACCESS*: a Virtual Private Network (VPN) access gateway, requires reference to a *VPNEXIT* element.

- *VPNEXIT*: a VPN termination; must be configured with a reference to the access gateway.
- *WEBCLIENT*: VNF based on the end-host model which can generate only *HTTP_REQUEST* packets; needs a *WEBSERVER* as a destination node.
- *WEBSERVER*: models an HTTP web server which sends *HTTP_RESPONSE* to the origin of web client requests.
- *FIELDMODIFIER*: represents a content-aware proxy able to modify the message payloads passing through it. Cannot be an edge node of the graph.
- *FORWARDER*: a VNF which forwards traffic, can be seen as a firewall permitting all packets.

3.2.1.2 Property Definitions

Property definitions allow specification of graph characteristics to be verified in order for the graph to be valid. Definitions are instances of the *Property* object and the property name is an enumeration value defining which attributes of the graph will be ensured to be valid. Except for the name, the object carries information about the source and destination nodes, layer-4 and layer-7 protocol considered, and source and destination ports. Currently, two different property types have been defined:

- *ReachabilityProperty*: can be imposed to verify that packets generated by the source are able to reach destination address.
- *IsolationProperty*: checks that traffic going out from the source node cannot reach the destination address.

Each property instance holds an attribute stating its satisfiability in the current NFV context.

3.2.1.3 Hosts

NFV *Hosts* object contains a set of *Host* items, each of them representing a physical host in the NFV Infrastructure. In Verifoo data model, a host holds information about its hardware specification, such as CPU an CPU cores, memory and disk storage, the maximum number of supported VNFs, and can have a type which can

be *CLIENT*, *SERVER* or *MIDDLEBOX*. Moreover, every host can have a list of supported VNF types to impose scheduling constraints during deployment. In order to represent nodes deployed on hosts, the *NodeRef* element is used, an array of references to nodes in a graph.

3.2.1.4 Connections

Physical links between hosts are represented using the *Connections* object, an array of items of type *Connection*. Each connection has a couple of values referencing source and destination host, plus information about average latency between them. In Verifoo, the concept of a network does not exist and must be represented using a mesh of connections spanning hosts in the same address domain.

3.2.1.5 Constraints

The *Constraints* object defines constraints affecting graphs and NFV infrastructure, imposing restrictions in deployment and property satisfiability. Two types of constraints can be enforced, *node constraints* and *link constraints*. The former specifies information about the VNF nodes in a graph, as the required CPU cores, memory and storage, and node optionality, while the latter describes requirements for connections between hosts, such as required latency.

3.2.1.6 Network Forwarding Paths

The *NetworkForwardingPaths* object is composed of a set of *Path* element, each one representing a logical connection between two or more different nodes in the NFV architecture. It is used to implement the NFV concept of more VFG-FGs spanning the same network service.

3.2.1.7 Parsing String

A string element accepting as a value the raw output of Verifoo execution used to convert a raw model to deployment information.

Resources	Method	Query Params	Req. body	Status	Resp.body	Meaning
ROOT	GET			200	Hyperlinks	OK
deployment	POST	complete:boolean	NFV	200	NFV	OK
				400	ApplicationError	Bad Request
				500	ApplicationError	Server Error
converter	POST	complete:boolean	NFV	200	NFV	OK
				400	ApplicationError	Bad Request
				500	ApplicationError	Server Error
log	GET			200	HTML	OK

Figure 3.3: Verifoo REST operations

3.2.2 REST API

In figure 3.3 are described endpoints available through Verifoo REST API. The interface is structured according to the principles of HATEOAS (Hypermedia as the Engine of Application State), a constraint of the REST architecture which allows the construction of a hypermedia-driven system in which client applications can access REST interfaces dynamically by following hyperlinks included in server responses.

```
<Hyperlinks>
    <Link rel="self" href="http://localhost:8080/verifoo/rest/"
        type="application/xml" method="GET" />
    <Link rel="deployment" href="http://localhost:8080/verifoo/rest/deployment"
        type="application/xml" method="POST" />
    <Link rel="converter" href="http://localhost:8080/verifoo/rest/converter"
        type="application/xml" method="POST" />
    <Link rel="log" href="http://localhost:8080/verifoo/rest/log"
        type="text/html" method="GET"/>
</Hyperlinks>
```

Listing 3.1: HATEOAS-style Hyperlinks XML object

Operations exposed from the API are:

- **Root**. A *GET* method on the root resource returns a collection of URLs describing available operations and their reference endpoints in a *Hyperlink* object (3.1).
- **Deployment**. The main API for interaction with Verifoo, allows the *POST* HTTP verb receiving as an input in the request body an NFV object (figure 3.1). As a response, the client receives the same NFV object with integrated

information about properties verification, elements configuration and deployment nodes layout across hosts. The *complete* query parameter is a boolean value indicating whether the response body should contain hosts and connections that have not been involved in graph deployment.

- **Converter**. The *converter* API allows conversion between the raw output of Verifoo execution, to be included in the *ParsingString* NFV element (3.2.1.7), and deployment XML output. Permits a *POST* HTTP request with an NFV object in the request body, and returns the same object including deployment information.

- **Log**. A convenience operation allowing reading the log output of the service from the web interface. Returns an HTML object that can be rendered through a web browser.

3.3 Open Baton

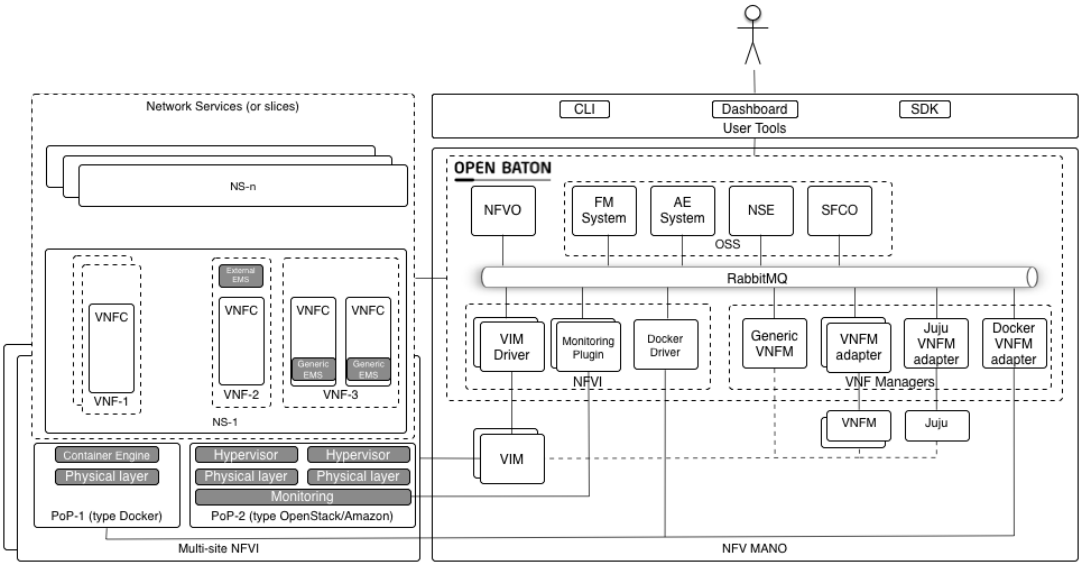

Figure 3.4: Open Baton architecture. Source: [15]

Open Baton[16] is an open source platform compliant to the standards of ETSI NFV Management and Orchestration (MANO) specification, developed by Fraunhofer FOKUS (Fraunhofer Institute for Open Communication Systems) and Technical University of Berlin. It is designed with the main goals of extensibility and

flexibility, leveraging a message broker to allow decoupling between the different component of the orchestrator architecture, which can be plugged in according to the user needs.

As can be seen from figure 3.4 the orchestrator revolves around an instance of RabbitMQ Message Broker[17], an open-source software implementing the Advanced Message Queuing Protocol (AMQP) to support asynchronous communication between modules. The orchestrator supports a wide range of VNF solutions, and through its generic VNFM and EMS allows the definition of VNFs at runtime through integration with the VNF lifecycle event engine. It is natively a multi-PoP orchestrator, allowing the definition of different types of VIMs though a pluggable VIM driver implementation which permits deployment over different NFV infrastructure without modifications in the orchestration logic.

3.3.1 Network Function Virtualisation Orchestrator

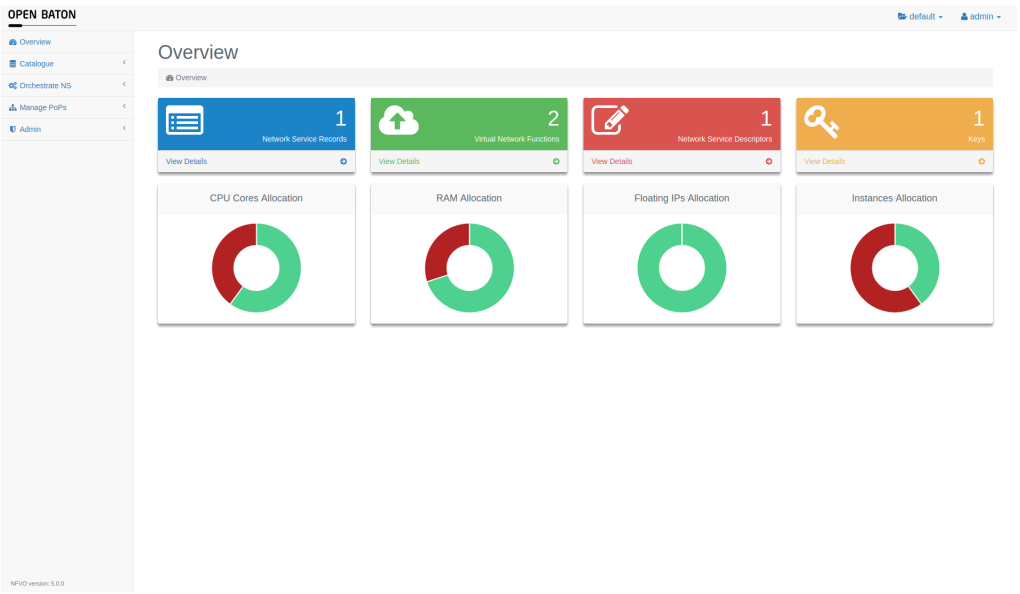

Figure 3.5: Open Baton management dashboard.

The core component of Open Baton allowing user interactions and service delivery is its MANO-compliant NFV Orchestrator, designed and implemented following ETSI specifications. Its main features include:

- **Descriptors repository**: the NFVO allows creation, retrieval, modification and deletion of NS and VNF descriptors, and SSH key catalog management;

VNFs onboarding can be achieved also through the VNF Package format, which consists of a tar archive containing descriptors, VDU image files, and metadata necessary for VNF instantiation.

- **Service lifecycle management**: allows instantiation, upgrade, scale-in and scale-out, and deletion of Network Services; takes care of orchestrating deployment of VNF instances over existing NFVI PoPs, and NFVI compute and network resource allocation. Deployed services are stored as NSRs, and the NFVO acts as a registry of allocated instances.

- **NFVI PoP management**: through the NFVO is possible to register NFVI PoPs in order to make them available for deployment of VNF instances. PoP onboarding capabilities are exposed from VIM driver components, which will be detailed in section 3.3.3. Aggregating information from the different PoPs, the NFVO is able to offer an overview of the NFVI available and used resources, such as total CPU cores or memory allocation.

- **Access control**: Open Baton implements security through authentication and identity management ([18]), which is achieved by means of projects and users with assigned roles so that the same environment could be safely shared between multiple users. Open Baton uses the concept of the project to represent the top-level isolation level between resources: users can be assigned to different projects, each adopting a different role, so that project-specific resources such as PoP instances, NS and VNF descriptors could be accessed only by authorized users. Possible roles a user can have in a project are *GUEST*, allowing read-only access to resources, and *USER*, which gives permissions to create, update and delete PoPs and descriptors, while a user with the *ADMIN* role is able to manage the whole system independently from the project.

- **Marketplace**: the orchestrator is able to import NSDs, VNF descriptors and packages, VDU images and other plug-in components such as VIM drivers from a public marketplace portal. This can be used to download example descriptors to get started with the tools or to ease VIM images onboarding and installation of plug-ins. At the time of writing, only a limited set of items uploaded from the Open Baton team is featured, but in following releases it is planned to add support for user uploads.

- **Dashboard**: Open Baton features a web interface (figure 3.5) which can be used to manage all aspects of the NFV environment; it allows control over all components, offering an overview over NFV infrastructure in terms of free

and used resources, PoPs accessible for deployment, information about their virtual networks and available images, and tracks NS deployments and their status, VNF instances and information about resourced assigned to them. The dashboard is a necessary feature for scenarios in which one or more administration users are in charge of service management, offering a user-friendly interface for operating the NFVO and controlling all its features, overview, and management of the deployed network services, and their creation.

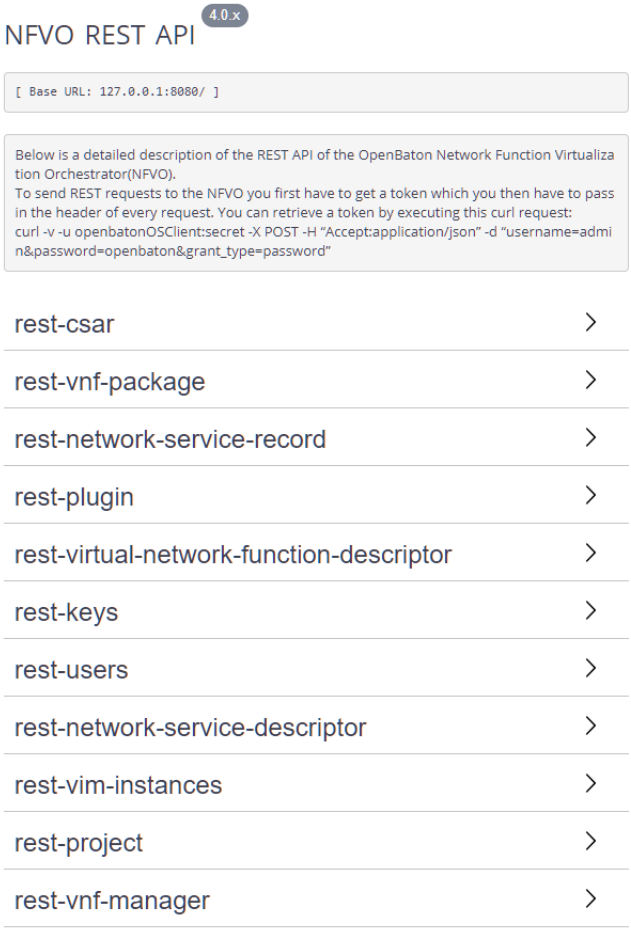

Figure 3.6: NFVO REST operations overview. Source: [19]

- **REST API**: to enable use cases in which the NFVO operation is not handled by a user, but instead it is integrated through external custom tools or included in a service-oriented architecture, all features which can be controlled through the dashboard are also exposed through a RESTful interface for programmatic consumers. An outline of operations which can be performed can be seen in

figure 3.6, grouped according to the category they belong to.

Open Baton RESTful operations make use of an information model compliant to ETSI specifications and described in section 2.3.4, serialized through the JSON data interchange format.

- **CLI**: another tool which can be used to operate the orchestrator is a command line interface (CLI), used to send a command through the terminal as an alternative to the dashboard.

3.3.2 Virtual Network Function Managers

In order to control VNF instantiation and lifecycle management, multiple VNF managers can be added to the orchestrator leveraging the plug-in system, each addressing a particular domain of VNFs. VNF managers available to be installed at the moment of writing are detailed in the following sections.

3.3.2.1 Generic VNFM

The Generic VNF manager is a general-purpose VNFM implemented according to ETSI specifications, managing VNF instantiation and configuration through interaction with the virtual machine instances on which the VNF software is installed. Generic VNF lifecycle hooks are defined specifying a set of lifecycle events for the VNF, and associating to each event one or more script files contained in a VNF package or referenced as a link by the VNFD: upon the occurrence of these events, the VNFM executes the corresponding scripts by way of its tight interdependence with the Open Baton Element Management System (EMS), which is installed as an agent on the VM instance hosting the VNF (figure 3.7). The VNFM is capable of handling errors occurred during the execution of these scripts, and resuming the VNF lifecycle event execution from the failed script.

Although current trends in day-1 operations lean towards the use of a configuration management tool such as Puppet, Chef, Salt or Ansible, this approach even if simplistic brings great flexibility and can be applied to any network function software. Listing 3.2 describes an example of lifecycle scripts inside a VNF descriptor; non relevant sections of the VNFD have been omitted for brevity.

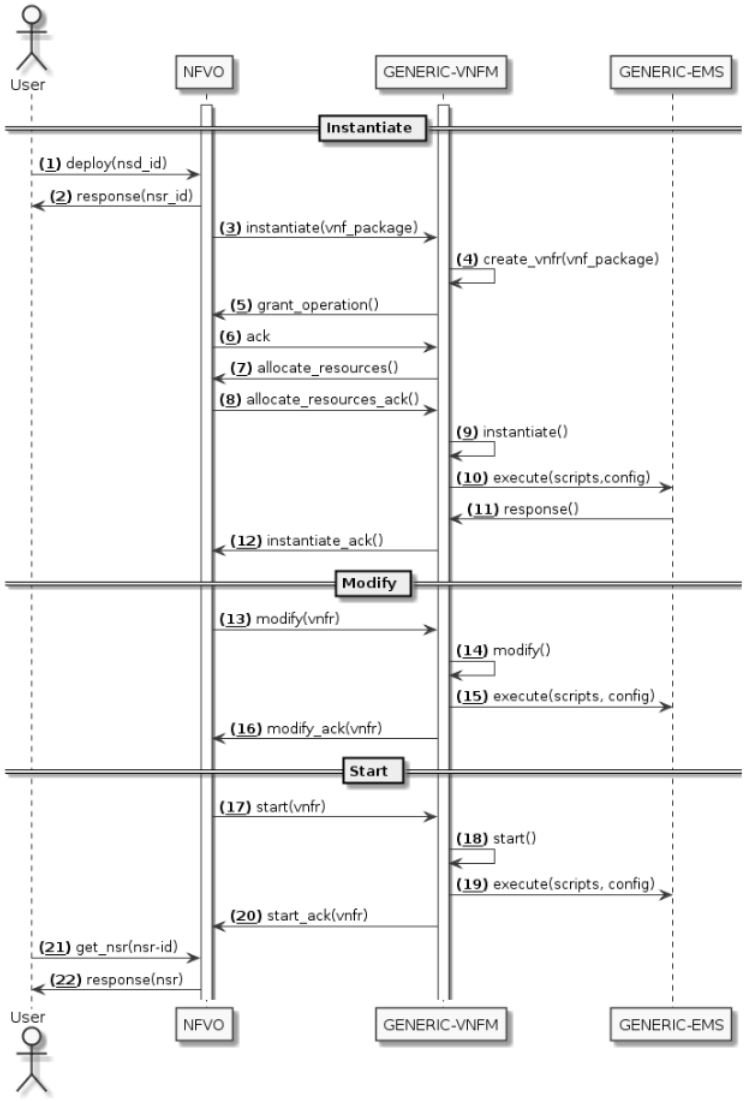

Figure 3.7: Open Baton Generic VNFM sequence diagram. Source: [20]

3.3.2.2 Juju VNFM adapter

Juju ([21]) is an open-source application modeling tool, backed by Canonical, the company behind the Ubuntu distribution of the Linux operating system. It allows deployment, configuration, and management of software across a virtualized environment such as private or public cloud platform. In contexts where applications are not shipped in a standalone fashion, but they have dependencies, relationships and connections with each other Juju allows service definition through an abstraction level offering an outline view of the complexity of the applications involved and their

```
{// VNFD
...
"lifecycle_event":[
    {
        "event":"INSTANTIATE",
        "lifecycle_events":[
            "pre-install.sh",
            "install.sh"
        ]
    },
    {
        "event":"CONFIGURE",
        "lifecycle_events":[
            "server_configure.sh"
        ]
    },
    {
        "event":"START",
        "lifecycle_events":[
            "start.sh"
        ]
    },
    {
        "event":"STOP",
        "lifecycle_events":[
            "stop.sh"
        ]
    },
    {
        "event":"TERMINATE",
        "lifecycle_events":[
            "terminate.sh"
        ]
    }
],
...
}
```

Listing 3.2: VNFD lifecycle event block example

configurations, and ease management operations such as deployment, scale-in and scale-out, and monitoring.

From the point of view of the NFVO, it offers a flexible solution to handle VNF lifecycle and day-1 operations, fulfilling the role of both VNFM and EMS. Through the Juju VNFM it is possible to deploy VNF packages uploaded to the catalog, VNFDs described through lifecycle hooks and *Charms* from the Juju Charm Store

([22]), a marketplace gathering an ecosystem of application packages ready to be deployed. At the moment of writing, complete interoperability between the lifecycle engine of the generic VNFM and Juju is not granted but is part of the roadmap for future development.

In order to successfully employ Juju as a VNFM, it is necessary to have it installed and configured to interact with a supported VIM, e.g. an on-premises virtualization platform as an OpenStack or VMWare vSphere instance, or a public cloud provider such as Amazon AWS, Google Cloud Platform or Microsoft Azure. Alongside the tool, on the same host, the Open Baton plug-in adapter must be installed and connected to the orchestrator through the AMQP protocol. The ETSI specifications state that resource allocation in MANO environment could be performed from the NFVO interaction with the VIM or directly from the VNFM: while the general approach of Open Baton is to comply to the first option, this scenario belongs to the latter case, therefore it is necessary to specify a dummy VIM driver in the VNFD when using Juju as VNFM.

Although being a very powerful tool for service definition and lifecycle management, Juju has not been designed to act as a VNFM in an ETSI-compliant environment: there are different disadvantages that could dissuade the MANO administrator from exploiting this VNFM at the current state, even if solutions for these issues are planned for future releases of Open Baton. The most restricting issues include low reliability for NSR/VNFR status updates, lack of scaling support, and the impossibility of passing configuration parameters to Charms deployed from the Charm Store.

3.3.2.3 Docker VNFM

Docker is a containerization platform (see 2.1.1) which enables packaging of applications and required libraries into portable and self-sufficient distribution units and ship them as containers. The Docker VNF manager, if used in conjunction with the Docker VIM driver (3.3.3.2) enables instantiation of VNFs running in containers on top of a Docker engine. The VNFM works with the upstream NFVO logic and is perfectly integrated, yet due to the conceptual differences between virtual machines and Docker containers, some fields in the VNFD might assume different meanings; the most relevant difference in management relies in the fact that while VM instances are a complex entity and can be configured, Docker images are thought to be used without being modified, therefore the lifecycle engine is disabled for container VNFs.

3.3.2.4 Custom VNFM adapter

This adapter plugin allows Open Baton interaction with users' VNF managers; the NFVO exposes a REST interface for communication with the VNFM, which can be consumed to integrate with the NS lifecycle. Figure 3.8 shows the sequence diagram of the instantiation of an NS, describing the different types of exchanges between the two components: upon user instantiation request, the NFVO invokes REST operations on the VNFM, subsequently waiting for the corresponding callback function.

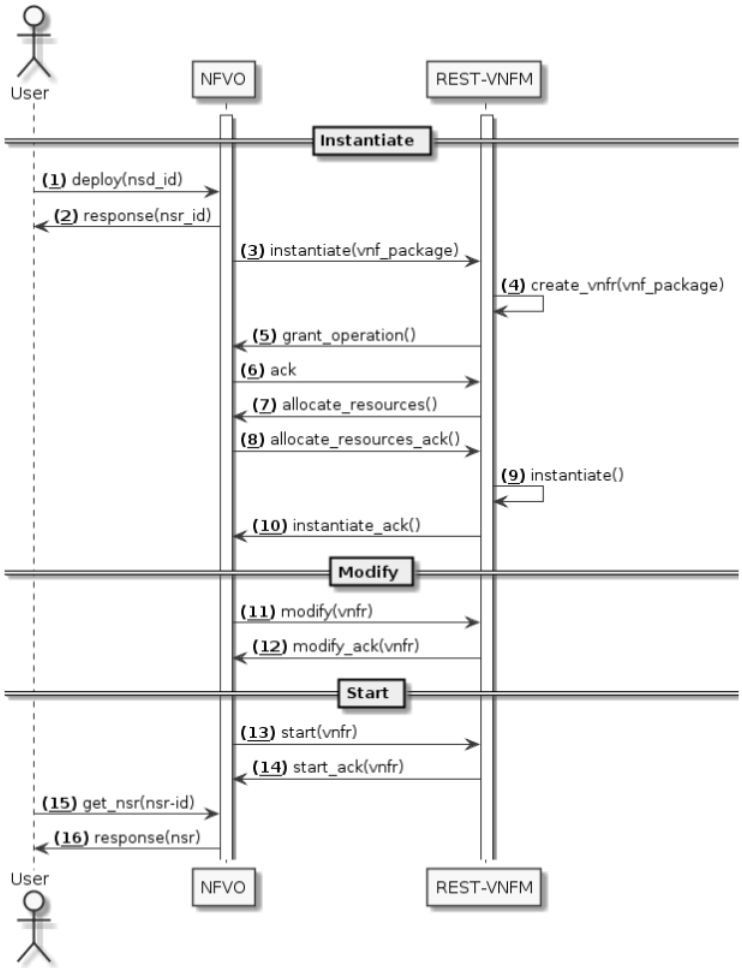

Figure 3.8: Open Baton VNFM interaction sequence diagram. Source: [23]

In order to be used for VNF instantiation, custom VNFMs must be registered to the NFVO though a REST operation including the request body reported in listing 3.3.

```
{
    "type":"the VNFM type handled, referenced in the VNFD using the field
        'endpoint'",
    "endpointType":"REST or AMQP",
    "endpoint":"the URL the NFVO will use to connect with the VNFM, in the form
        'http://<IP>:<PORT>'",
    "description":"Custom description",
    "enabled":"true/false"
}
```

Listing 3.3: VNFM registration request payload

3.3.2.5 Custom-built VNFM

It is also possible to create a custom VNFM which could be plugged-in to the orchestrator via a REST-based interface or connecting to the message broker using AMQP. To this purpose, Open Baton released several Software Development Kits (SDK), each with different languages support (Java, Python, GO), which can be used to build a personalized VNF manager without the need of implementing the low-level communication details and focusing on the VNF lifecycle logic.

3.3.3 Virtualized Infrastructure Manager Drivers

Open Baton interacts with the NFV Infrastructure Manager through VIM drivers, plugins implementing interfaces used by the NFVO which enable resource management. Differently, from the other plug-in components, VIM drivers are built based on a Remote Procedure Call (RPC) architectural style and are registered at the NFVO at startup time as JAR libraries in a specific installation folder. Follows a list of currently implemented and generally available drivers.

3.3.3.1 OpenStack VIM driver

Open Baton OpenStack VIM driver leverages the OpenStack4J library to allow interaction with PoPs managed by the de-facto standard for NFVI managers consuming its REST APIs. It is able to allocate all necessary resources for VNF execution, including networks, routers, floating IP addresses and compute instances. Documentation ([24]) suggests the use of an admin user to allow these resources to be created in case of necessity, while without administration rights it could not be possible to create new networks and set up routing policies. When making use of an

OpenStack instance, security groups have to be created beforehand and affect all resources deployed on the same PoP.

3.3.3.2 Docker VIM driver

Together with the Docker VNFM (3.3.2.3) allows the deployment of network services over a Docker engine. It offers the possibility to create both container and networking resources interacting with PoPs managed by a local or remote Docker engine, both through Unix sockets or over the network.

3.3.3.3 Amazon VIM driver

The Amazon VIM driver allows interaction with Amazon EC2 (see 2.2) using the Amazon Java SDK to invoke AWS REST APIs. In order to use this driver is suggested to create supporting resources such as VPC (Virtual Private Cloud) and security groups before deploying compute instances. Networking resources could be created from the driver or, if existing, can be referenced from the NS descriptor and VNF instances can be attached.

3.3.3.4 Test VIM driver

This VIM driver simulates the behavior of an actual VIM driver without instantiation of any real resource; allows service testing and dry-run, and can be used in a situation in which the VNFM handles resource allocation as for example in case of Juju VNFM (3.3.2.2).

3.3.4 Operation Support System (OSS)

Given the nature of Open Baton architecture, it is relatively easy to integrate external components to support service provisioning and management. OSS plugins provided by the project are detailed below.

3.3.4.1 Monitoring Plugin

The Open Baton orchestrator is able to interact with multiple monitoring systems through a standard communication interface using the plugin mechanism; plugins

implementing this interface act as an intermediate component between the NFVO and the monitoring solution allowing consumers (NFVO, VNF managers, other OSS) to interact with a well-defined and consistent API.

An example is the Zabbix plugin, integrating Open Baton with Zabbix Server. Zabbix ([25]) is an open-source monitoring solution able to collect metrics about, among others network performance, hosts resource consumption, and cloud services. This plugin extends the VIM driver interface implementing two ETSI-defined interfaces ([26]), *Virtualised Resources Performance Management* and *Virtualised Resources Fault Management*, enabling external components to create monitoring policies and query performance metrics.

3.3.4.2 AutoScaling Engine (ASE)

The AutoScaling Engine can be installed as a service plugin to the orchestrator and allows the automatic management of scale-in and scale-out operations based on policies defined by the user. It makes use of performance metrics collected by the monitoring interface (3.3.4.1) to define monitoring event subscriptions and the corresponding actions to be taken. The default plugin used is Zabbix plugin, but the integration with the standard interface makes possible to use any preferred monitoring system.

3.3.4.3 Fault Management System (FMS)

The Open Baton FMS plugin can be used to provide high availability and automatic fault remediation features to the orchestrator. It exploits metrics collected by monitoring plugins described in section 3.3.4.1 to enable the back-up to a standby instance in an active-passive failover scenario in case of instances malfunction, and supports actions executions upon VNF failure to restore normal execution status.

In the VNFD, a section called *fault_management_policy* can be provided at the VDU level, containing information about the monitoring events for which to create alarms and how to react: the FM engine takes advantage of the Drools rule management system[27] to create a policy and define which alarms to create and how to react.

3.3.4.4 Network Slicing Engine (NSE)

This OSS component enables quality of service (QoS) policy definition and enforcement in virtual networks according to information provided by the NFVO in NS descriptors. Currently, the only supported driver is OpenStack Neutron starting from Mitaka version but can be extended to introduce more networking services providers.

3.3.5 Event Engine and Plugins

Open Baton features a powerful event system based on a publish/subscribe paradigm which allows to easily extend the orchestrators features through external plug-in modules. Using the provided SDK an application can register for notifications upon the occurrence of events in the NS lifecycle, and perform actions in response. The orchestrator will submit requests to endpoints specified during subscription, filtering based on the events the plugin is interested to receive; the connection can be established using AMQP or REST protocols. Events that could be subscribed are relative to the NSR or the VNFR, or both of them: notifications are generated and submitted asynchronously including the JSON representation of the records in the request payload.

3.4 Solution high-level analysis

In order to meet functional requirements, different design approaches have been considered, taking into account all possible use case scenarios and used software capabilities and characteristics. Verifoo functionalities that have been considered for implementation, and considerations regarding their implementation possibilities in the context of extending Open Baton, are identified as follows:

- **Graph validation**: the input graph can be validated according to policies defined by the user, such as reachability or isolation between nodes. This feature allows the user designing the service to check its formal verification. Can extend Open Baton rejecting catalog upload of invalid service graphs, or preventing their deployment; in any case, a meaningful message should be provided to the service administrator in order to allow the design to be reviewed and policies adjusted.

- **Graph optimization**: Verifoo modifies the input graph pruning unnecessary nodes, and automatically configuring components according to specified policies; this should be reflected in target NSD deployments. Open Baton can benefit from this feature in two scenarios: when the service graph is modeled redundantly can identify necessary nodes for policy fulfillment and remove the others, and in place of day-1 configuration, to avoid manually configuring VNFs after deployment. In the first case, the pruning needs to be done at a catalog upload level, while in the latter the VNF configuration can happen at service definition or at instantiation independently.

- **Smart deployment**: providing a description of the NFV Infrastructure, Verifoo is able to optimize VNF deployment. The infrastructure must be defined in terms of hosts and interconnections between them. This feature can bring an advantage in resource optimization but is very situational considering that VIMs as OpenStack, VMWare vCloud and others often do not offer access to single hosts for machine deployments, instead they have internal scheduling mechanisms for resource management. In the eventuality VIMs are chosen between public cloud infrastructure providers, hosts hypervisor interface is not even accessible, nullifying benefits deriving from scheduling optimization.

Follows an analysis of the main aspects of the software solution necessary to fulfill functional requirements, together with high-level design principles and architectural analysis.

3.4.1 Single-PoP versus multiple-PoP deployment optimization

In order to implement a virtual machine placement optimization two use case scenarios have been considered, using a single PoP or multiple PoPs. Follows a detailed analysis of these possible options.

3.4.1.1 Single PoP

In case of deployment optimization of resources managed by the same PoP in a VIM, several considerations are worth to be mentioned. Allocation optimization in a single PoP scenario is the main Verifoo use case, and it means to create a smart placement policy able to select appropriate hosts where to deploy VNF instances based on

currently available resources, placement of VNFs deployed in the infrastructure, and constraints restricting placement for nodes such as dependency between functions or minimum compute or memory requirements.

After computing a distribution plan, the infrastructure management tool should support custom scheduling of instances over accessible hosts: this represents a limitation in terms of the possible VIM that could be employed, narrowing down the possible choices to OpenStack and VMWare vSphere, which are the products currently allowing hosts selection during instantiation. Infrastructure providers such as public cloud platforms in which the customer is agnostic to the physical infrastructure are therefore excluded from the use case; the same applies are other hypervisors solutions which do not offer a management framework, in which every host should be dealt with singularly, and for light virtualization solutions such as LXD or Docker.

In order to select hosts on which to deploy VNFs a custom VIM driver should be implemented, specific for the NFVI management platform. Moreover, more than one component should have been affected: the NFVO in order to be made aware of Verifoo integration, in charge of interaction control and results interpretation, the VIM driver to correctly control resource requisition, and *Or-Vi* interface, to allow host preference parameters to be correctly parsed and passed to VIM driver.

3.4.1.2 Multiple PoP

The scenario in which multiple PoPs, possibly managed by different types of VIM, is another significant use case which has been considered in the analysis phase of this thesis work to implement deployment management and optimization at the NFVO level. In this context, analysis of the optimum deployment schedule could be happening to take into account different PoPs as "hosts" in the multi-PoP infrastructure, each providing information about available and used resources from their respective pools. Each NS instantiation request could be this way evaluated in order to allocate each VNF on the best-suited PoP considering requirements in terms of computing, memory but also network latency and throughput.

Comparing this approach with Verifoo capabilities and data model, with particular interest to hosts (3.2.1.3) and connections (3.2.1.4) can be seen how it could be possible to treat the PoPs as the *Host* elements, and introduce *Connection* items representing the LAN or WAN links between them, including information about required latency and throughput. In this way, Verifoo could be able to provide a smart placement strategy for VNFs to be instantiated.

3.4.2 Integration design

Different approaches could be undertaken to achieve integration of service graph validation and deployment management in the tool: the following sections present the considered ones, together with their strengths and weaknesses.

3.4.2.1 Open Baton plugin

As a low-impact and integrated solution for extending Open Baton functionalities[28], the plugin has been a first-class option. As described in section 3.3.5, a plugin is a software component able to receive notifications about NSR lifecycle through registration to an event system. In response to events generated from the NFVO, the plug-in can take actions receiving as input data the generated VNF Record or the NS Record depending on the type of event the plug-in subscribed to.

This solution, even being the most integrated and better-documented way of extending the NFVO, does not satisfy the requirements, as it cannot directly interact with NS Record lifecycle once started deployment: it could be possible for the plugin to take actions consuming Open Baton REST interface, yet the service descriptor cannot be object of modifications until deployment, renouncing to service graph validation. Virtual instances placement is in any case not being considered as plugin capabilities do not allow to interact with the resource allocation from the NFVO.

3.4.2.2 Open Baton Contribution

A contribution to Open Baton project has been the most flexible solution considered, as deep integration with the MANO architecture allows fulfillment of requirements. Graph validation can be achieved integrating Verifoo with the NFVO component, and can be performed at onboard time, rejecting invalid service graphs uploads, or could interact with the lifecycle validating services upon deployment requests. Graph optimization can be performed together with validation or could happen at a separate time. As for service deployment management, the possibilities depend on the use cases defined in section 3.4.1: in case the infrastructure architecture is based on a single-PoP, as previously stated is necessary to modify ETSI-defined interfaces to allow interactions including deployment schedules information; differently, a multiple-PoP use case can be handled complying to MANO standards.

Although the most flexible solution, a contribution to the Open Baton project

leveraging an external tool for service validation introduces a dependency that could be seen from the community as an excessive burden to the project, especially considering the highly decoupled architecture the tool is built on. Moreover, it could be overcomplicated to maintain the software to make it compliant to the newer releases of the orchestrator.

3.4.2.3 External Software

The last option taken into consideration has been an external software integrating with both Open Baton and Verifoo through exposed REST interfaces. This solution allows to interact with the orchestrator introducing graph validation and optimization in the phase of Network Service catalog upload, thus allowing onboarding of a service graph only after a validation check, rejecting formally invalid descriptors, and possibly updating the graph to optimize configurations and service design. As tools involved in the solution have to be operated as an end user, consuming available user-facing services, platform-specific characteristics cannot be overridden, as could be happening in a contribution scenario; in particular it is necessary to renounce to NFV placement optimization feature, available in Verifoo, and instance deployment orchestration must be delegated to the MANO platform. Advantages in undertaking this approach are the interoperability of the solution, which could be able to support the lifecycle of both Verifoo and Open Baton in a decoupled fashion.

3.4.3 Analysis conclusions

Albeit the interesting possibilities offered by the multi-PoP architectural approach, also considering current trends in multi-domain and geographically-distributed NFV orchestration ([29]), it has emerged as a non-functional requirement that the use cases for this thesis work focus mainly on single-PoP scenarios. This considered, and taking into account the strengths and weaknesses of possible solution designs, the development of an external software interacting with the two existing tools has been evaluated as the best choice. In the following chapter will be detailed the design choices and implementation methods undertaken to achieve the desired result.

Chapter 4

Veribaton

With the label of "Veribaton" has been identified for the sake of brevity the solution developed to extend the capabilities of the Open Baton orchestrator in terms of service graph verification and validation, by integrating the tool Verifoo. This chapter intends to present an outline of the development process of Veribaton, and is structured as follows:

- Section 4.1 presents the design principles of Veribaton.
- Section 4.2 describes the implementation process, difficulties encountered and how they have been overtaken.
- Section 4.3 illustrates testing procedures and a demonstration scenario used to validate the solution.

4.1 Design

The aim of this development process is to obtain a product able to interact with Open Baton during the operation of NSD catalog upload in order to perform verification and validation over the graph instance before onboarding. Figure 4.1 shows the sequence diagram representing the graph uploading activity: as it can be seen, Veribaton receives as input the NSD, converts this information into a representation which can be digested by Verifoo, subsequently consumes the verification service to obtain an optimized version of the original graph. After obtaining the result, converts the result in ETSI NSD representation to feed it to Open Baton for catalog

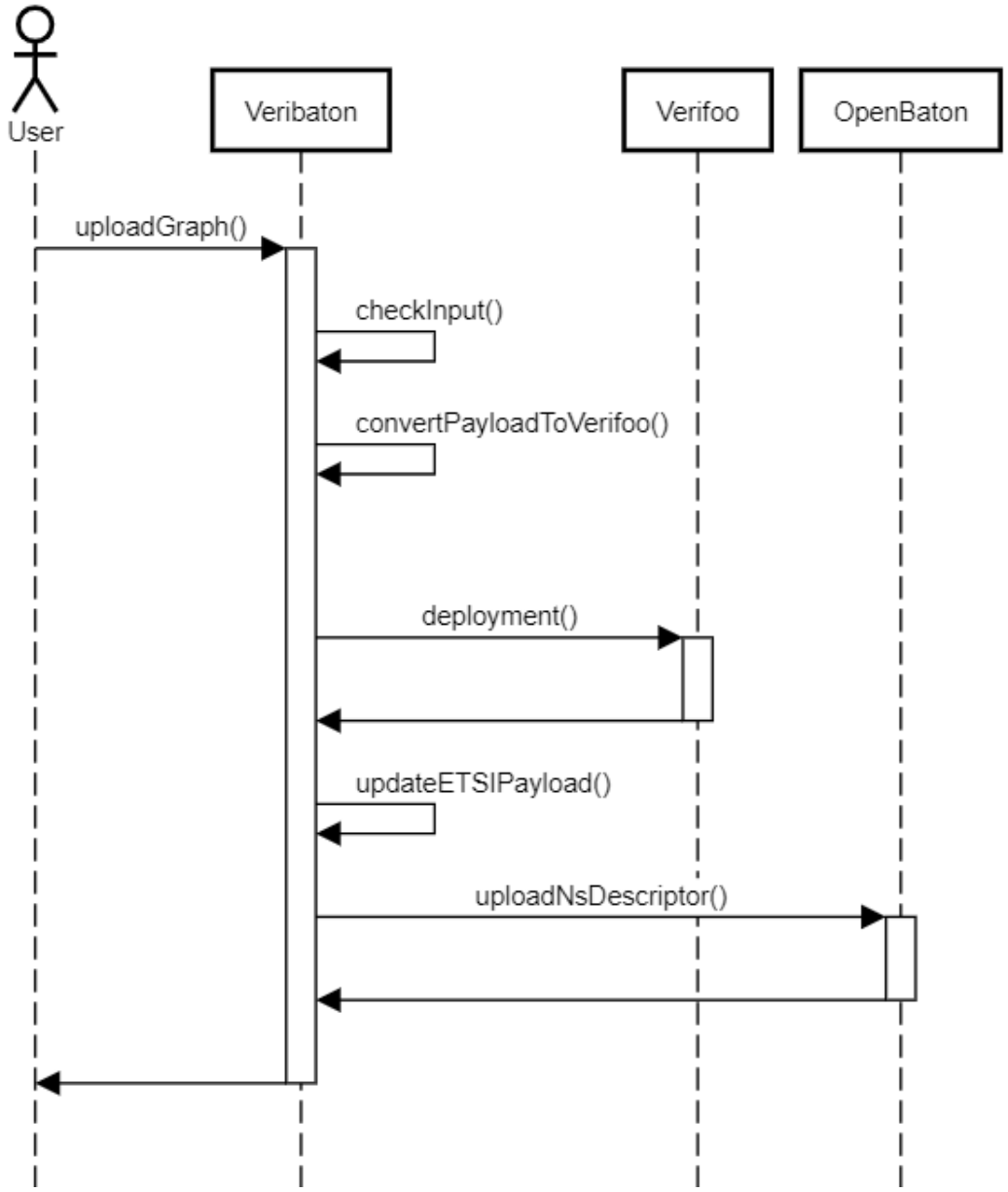

Figure 4.1: Veribaton sequence diagram.

onboarding. In the end, the user receives a feedback message stating the success or failure of its operation.

This architecture has been designed to comply with the following principles:

- **Interface compatibility**: the interface used to interact with Veribaton service should match completely the interface exposed by Open Baton for NSD

onboarding. In this way, the end user can be unaware of Veribaton presence if not interested, and behave as it was interacting with Open Baton; the main advantage in this context is that network services can be developed following ETSI data model, making Veribaton validation capabilities pluggable at the user discretion, based on the endpoint where the request is placed. Veribaton becomes this way a sort of "proxy" which could be used depending on the needs, with NSD instances built directly for Open Baton. As communication with Open Baton happens through a REST interface over HTTP, Veribaton will be itself a RESTful API server.

- **Input validation**: Verifoo acts as a validator for the input provided to Veribaton, implying that an invalid service graph will be blocked before reaching Open Baton with suitable feedback for the user. In particular, Verifoo is in charge of verifying that nodes are correctly organized in a chain, and policies specified as input such as reachability and isolation between VNFs are satisfiable. In this way, it is prevented the scenario in which a service present in the catalog once deployed does not behave as expected.

- **Graph optimization**: once received the optimal service configuration from Verifoo, the original input should be modified according to it. Possible scenarios include removal of nodes from the graph and automatic configuration of elements such as firewalls, which should be reflected on the NSD to be uploaded to Open Baton.

4.1.1 Development framework

The design phase of the tool included the choice of the programming environment to be used to develop Veribaton. The following sections define an outline of the selected tools and the reasons for which they have emerged as the preferred solution.

4.1.1.1 Java programming language

The programming language chosen to implement Veribaton is Java[30], an object-oriented general-purpose language born in the 1990s which has constantly grown in popularity over the years and appreciated for its versatility and compatibility. It has been selected for different reasons:

- **Ease of use**: being statically-typed, object-oriented with a syntax deriving from C and C++, Java language is fairly well understandable and easy to use. It relies on a garbage collector for automatic memory management, relieving the programmer of memory allocation issues. It's type system, although complicated, is well designed and allows great flexibility in type definition.

- **Platform independence**: Java is an interpreted language designed to execute on top of any operating system and hardware architecture providing run-time support for it. This is achieved by compiling the source Java code into an intermediate representation called *bytecode*, a set of low-level instructions which can be executed from an interpreter, the Java Virtual Machine (JVM), compatible with the host execution environment. In this way, Java bytecode can be exported everywhere a JVM is supported and be executed without modification.

- **Popularity and support**: during the last decades, Java has grown exponentially as a programming language, becoming a sort of *lingua franca* of enterprise software development. This growth has generated a complex and rich ecosystem of applications, libraries, and tools supporting the language, each followed by its specific documentation, which constitutes real value for the community and for the programmer. A prolific production of open-source libraries provides implementations for many of the most common development issues.

- **Object-oriented paradigm**: Java is built as an all-around object-oriented language, its consistent and efficient type system allows to organize and extend the code, and is better suited for management of a large-scale code base than most of the general-purpose languages.

- **Performance**: modern JVMs leverage Just-In-Time (JIT) compilation to compile Java bytecode into native binaries targeting the machine executing the code at run-time. This provides excellent performance, almost comparable to native binaries built compiling low-level languages such as C or C++ into machine code, surely outrunning interpreted programming languages like Python or JavaScript[31].

- **Maintainability**: due to the popularity of Java language and its easy understandability, united to the tooling which allows documenting the deliverable in an appropriate way, it will be possible for the open-source community to maintain and extend the software.

- **Open Baton SDK availability**: although a simple REST client could be sufficient for interacting with Open Baton, an SDK is offered as a Java library to ease operations.

Nevertheless, different drawbacks might emerge from the usage of this programming language:

- **Memory management**: the usage of a virtual machine executing the bytecode, and garbage collection without instant reclaim, cause Java programs to consume memory in orders of magnitude higher than systems with manual memory management such as C or C++. Memory deallocation is performed by the garbage collector in some implementations of the JVM in a stop-the-world fashion, interrupting program execution until unreachable objects are not freed, in others is handled incrementally, yet consuming more resources in the process. However, in modern systems with fewer constraints in terms of memory, this approach does not constitute a significant problem for general-purpose and non-real-time software.
- **Verbosity**: Java as a language has been designed with simplicity and understandability in mind, and the features that make it so are also causing programs written in Java to be fairly longer than other programming languages; some frameworks implementing the paradigm of dependency injection and inversion of control (Spring Framework among them, detailed in section 4.1.1.2), reduce the amount of code necessary to the program execution at the cost of clarity of the interactions between classes.

Other languages considered for Veribaton implementation include Python, C, JavaScript, and Golang among languages known by the author, yet for the reasons described Java emerged as the preferred tool.

4.1.1.2 Spring Boot

There exist several ways of implementing a RESTful service in Java, including using instances of the *Servlet* interface to accept HTTP requests and provide responses, using a JAX-RS API[32] implementation like Jersey or Apache CXF, or using a more complex framework such as Spring or Apache Struts. Using low-level HTTP handlers reduces applications dependencies, yet makes hard implementing REST paradigm from the ground up. Leveraging JAX-RS, an interface for RESTful service defined

by the Java EE (Enterprise Edition) platform; implementations of this standards come in different forms and depend on the application server the web service will be deployed on. A framework as Spring or Struts abstracts low-level details to allow focusing on business logic.

Spring Boot[33] is a project aiming to simplify configuration and deployment in building a Java Spring application by packaging in one executable all necessary components to execute it, providing a completely stand-alone application working out-of-the-box. It extends the Spring framework by embedding the required servlet container (Tomcat, Jetty or Undertow), eliminating the need of packaging the application into a WAR archive and deploy it to the desired application server; moreover, takes care of the aspects that received criticism of Spring, such as the need of verbose XML configuration, providing standards for configuration externalization without the need of writing XML, and adding nonfunctional features like security and support for authorization and authentication, metrics and health-checks. Being well-documented and allowing full personalization of the features and behaviors of the framework, in addition to enabling a loosely coupled architecture, Spring Boot has been considered appropriate for selection in Veribaton development.

4.1.1.3 Gradle

Spring Boot application projects are typically accompanied by a build automation utility tool for libraries dependency management, source code compilation, binary packaging, and unit tests execution. Any build system could be used, yet fully supported build systems are *Maven* and *Gradle*. Maven is an Apache project defining a POM (Project Object Model) as an XML file defining the project structure and its configuration details used to build the project. Gradle, instead, achieves dependency management and build automation leveraging a DSL (Domain-Specific Language) based on Groovy, a JVM-based scripting language) to define and run tasks. Both of them allow project compilation and dependency resolution, downloading necessary libraries from external repositories, defining the tasks to execute in order to build the binary and in which order, and packaging the application in different formats.

In the context of this thesis work, the reasons behind the choice of Gradle as a build management tool can be described as follows:

- *Dependency graph*: while conventional systems use a linear representation in the form of a series of steps to describe tasks and their dependencies, Gradle

makes use of a directed acyclic graph (DAG) to model these relationships, thus allowing more precise and granular task definition.

- *Incremental builds*: due to the DAG structure, Gradle is capable of determining whether a task needs to be executed or not based on the update of its dependencies; in this way unnecessary tasks execution is avoided, dramatically reducing build time.

- *Performance*: in addition to incremental builds, Gradle is able to reuse the output of previous runs of tasks when detects they are being run with the same inputs, implementing this way a caching system to further reduce build times. Moreover, Gradle makes use of a compilation daemon running in the background, to minimize build initialization time deriving from the instantiation of the JVM, as well as to leverage in-memory caching for achieving the best performance.

- *Flexibility*: Gradle is extensively customizable, it allows the definition of custom tasks and scripting using Groovy.

4.1.2 Open Baton API interface

Open Baton offers a broad catalog of RESTful operations allowing integration with the orchestrator system as a whole, exposing all possible interactions with the system. Among documented interfaces, it is necessary to select a subset for implementation in order to expose them as Veribaton APIs.

Tables 4.1, 4.2 and 4.3 show API operations considered, divided by scope; each of them receives as an input parameter as a request HTTP header the identificator of the project it is referring to, therefore the parameter has been omitted for clarity. Open Baton API catalog is fairly larger, yet it has been chosen to not include APIs scopes not relevant to this analysis, such as users management or VIM onboarding, and to filter out preemptively operations not meaningful for the purposes of this thesis work.

Selected contexts of interest, as can be seen, include:

- **NSD**: intercepting CRUD (Create, Retrieve, Update and Delete) operations on NSD resources is the most straightforward mean of interacting with service

Table 4.1: Open Baton NSD API operations.

Scope	Method	Endpoint	Description	Codes
NSD	GET	/api/v1/ns-descriptors	Returns all NSDs onboarded.	200 404
NSD	POST	/api/v1/ns-descriptors	Onboards the NSD received as payload.	201 404
NSD	GET	/api/v1/ns-descriptors/{id}	Returns the NSD with the specified ID.	200 404
NSD	PUT	/api/v1/ns-descriptors/{id}	Creates or updates the NSD with the specified ID.	201 202 404
NSD	DELETE	/api/v1/ns-descriptors/{id}	Deletes the NSD with the specified ID.	204
NSD	POST	/api/v1/ns-descriptors/multipledelete	Deletes NSDs which IDs are specified in payload.	204 404

graph; in particular NSD onboarding can be leveraged to verify graph validity; in case of invalid service definition, the operation can be stopped, differently, the request can be forwarded to Open Baton.

- **VNFD**: APIs related to the VNFD catalog have been considered for implementation, as in order to be used in a network service, VNFs need to comply Verifoo specifications in terms of functional type and configuration (details in section 4.1.3).
- **NSR**: implementing NSR-related operations would mean the possibility of interacting with NSD deployment, therefore could be possible to implement service verification at deployment time. These APIs allow modification of an already deployed service, which could indicate the update of an existing service in order to comply with formal verification of the service graph.

Looking at functional requirements, the workflow step considered more appropriate for enforcement of correctness and reachability policies has been the NSD

Table 4.2: Open Baton VNFD API operations.

Scope	Method	Endpoint	Description	Codes
VNFD	GET	/api/v1/vnf-descriptors	Returns all VNFDs onboarded.	200 404
VNFD	POST	/api/v1/vnf-descriptors	Onboards the VNFD received as payload.	201 404
VNFD	GET	/api/v1/vnf-descriptors/{id}	Returns the VNFD with the specified ID.	200 404
VNFD	PUT	/api/v1/vnf-descriptors/{id}	Creates or updates the VNFD with the specified ID.	201 202 404
VNFD	DELETE	/api/v1/vnf-descriptors/{id}	Deletes the VNFD with the specified ID.	204
VNFD	POST	/api/v1/vnf-descriptors/multipledelete	Deletes VNFDs which IDs are specified in payload.	204 404

onboarding, therefore the implementation of operations related to NSD catalog is required.

Analyzed use cases for Veribaton involve uploading of a service graph as a whole, given Verifoo data model, therefore exposing VNFD-related operations has been considered not essential; if necessary, could be implemented in future releases of the tool.

In this context, the instantiation of the verified NSD is imagined to be executed in a following moment at the hands of a system administrator able to manually inspect the graph; moreover, after onboarding through Veribaton, it would be redundant to perform validation for the service. This set of APIs could be implemented in the scenario of update of a deployed service, which for the moment has not been considered.

Table 4.3: Open Baton NSR API operations.

Scope	Method	Endpoint	Description	Codes
NSR	GET	/api/v1/ns-records	Returns all NSRs deployed.	200 404
NSR	POST	/api/v1/ns-records	Deploys the NSR receiving an NSD as payload.	201 404
NSR	GET	/api/v1/ns-records/{id}	Returns the NSR with the specified ID.	200 404
NSR	POST	/api/v1/ns-records/{id}	Deploys an NSR with the NSD having the ID specified in path.	201 404
NSR	PUT	/api/v1/ns-records/{id}	Updates the NSR with the specified ID.	201 202 404
NSR	DELETE	/api/v1/ns-records/{id}	Deletes the NSR with the specified ID.	204
NSR	POST	/api/v1/ns-records/multipledelete	Deletes NSRs which IDs are specified in payload.	204 404

4.1.3 Data model conversion

As summarized in table 4.4 the most critical aspect which should be taken into consideration during the design of the solution derives from the difference in the data model between Verifoo and Open Baton representation of the service graph:

- **Serialization format**: Open Baton is using JSON as its data model serialization format, while Verifoo makes use of an XML document adhering to an XML Schema Definition: it will be necessary to convert between the two formats during graph validation step. In order to comply with Verifoo schema, it

Table 4.4: Open Baton and Verifoo information model comparison

	Open Baton	Verifoo
Serialization	ETSI-defined JSON	XSD-based XML
VNFFG	Nodes connected to virtual links	Adjacency between nodes
Node descriptors	VM image	Functional type
Configuration	Key-value pairs	Node-specific configuration
Infrastructure	N/A	Physical hosts and links

is necessary to import the XSD file and validate the generated XML payloads against the schema. The most viable solution to integrate XML validation seamlessly is automatic code generation from the schema, using one of the supporting tools for the technology.

- **Graph model**: the representation of the VNF forwarding graph between the two tools significantly differs conceptually; in Open Baton and in ETSI guidelines the VNF-FG is represented as a set of paths interconnecting different VNF descriptors to form a graph, while in Verifoo the separation between nodes descriptors and connections enabling traffic flow between them is not highlighted, and graphs are created using the idea of adjacency between neighboring nodes. In this case, it will be critical to understand how to convert the VNF paths model to an adjacency-based model without losing information.

  At the moment of writing, Open Baton does not completely support Service Function Chaining (SFC), preventing routing configuration through SDN-based techniques, therefore a heavy usage of the VNFFGD section (2.3.4.5) seems inappropriate for the first assumption of the design principles, according to which compatibility with Open Baton represents a crucial aspect. Moreover, Verifoo data model includes the concept of forwarding graph in the definition of nodes, and for this reason, it is impossible to represent different graphs spanning over the same VNF nodes. Given these assumptions, has been decided to drop multiple graph support in favor of simplicity in the definition of the network service: in service design, adjacency between nodes will be represented by VNFs connected to the same virtual link, and a simple algorithm

will extract the neighbor of each node based on its connections.

- **VNF descriptors**: in Verifoo, VNFs are represented as nodes, each having as an attribute a functional type describing the role the VNF embodies in the service graph in a broad outline, such as *FIREWALL*, *NAT* or *WEBSERVER*. This conflicts with ETSI specifications, in which a VNF is represented using its Virtual Deployment Unit (VDU); VDU information includes, other than CPU cores, memory, storage capacity of the VNF, information about the virtual machine image used to clone the VNF, without any specific information about its functional behavior. For this reason, it is compulsory to define in VNF metadata information about the role it is assuming in order to propagate this information to Verifoo during graph validation. Open Baton information model offers the *type* attribute at the VNF level which can be fit for the purpose. Is to be considered that Verifoo functional type can assume a limited number of different values, this will cause a restriction in the ETSI data model which provides a free format string as an input value.

- **Configuration**: another noticeable difference which should be taken into account in solution design is how VNFs configuration parameters are specified. In Verifoo, each VNF functional type includes its own configuration item, each defining one or more parameters the node can receive. In Open Baton and ETSI in general, parameters are passed to VNFs as key-value pairs without any particular restriction; this approach leaves the user a higher grade of flexibility and customization for the functions, yet on the other side lacks input control, definition of required and optional parameters, format of configuration values, and complex types such as arrays and dictionaries, not negligible in case of heterogeneous scenarios in which VNFs might require articulated settings.

  In order to achieve the desired result in terms of configuration, it is necessary to define for each functional type which parameters to support for Verifoo verification; selected ones will be evaluated by Verifoo and can be used for service graph properties such as reachability or isolation between nodes as in case of firewall rules or NAT settings, other parameters will pass through validation and will be meaningful to Open Baton only. The disadvantage of this approach is that this might likely result in hard-coded handling of possible enumeration values, hence implying a lower overall code quality in terms of flexibility and modularity.

- **Infrastructure description**: Open Baton data model is infrastructure agnostic, deployment is entirely handled by the VIM driver, while infrastructure

knowledge is delegated to the VIM internals. Verifoo, instead, has knowledge of physical resources and uses them to compute an optimal deployment plan. At the beginning of the design process, Verifoo was not able to verify the graph and fulfill deployment information without specification of the infrastructure the VNFs will be deployed on, hence it has been necessary to implement a Veribaton version creating a dummy infrastructure on which Verifoo could operate. Verifoo second version has been enhanced to overcome these limitations, therefore it has been possible to neglect the notion of underlying infrastructure.

4.2 Implementation

This section addresses the development phase following the design stage of Veribaton. It focuses on interaction with the framework and guidelines used in order to improve Veribaton user and developer experience in terms of configuration management, maintenance, and code readability; outlines the core business logic illustrating the implementation of the algorithms for data conversion between the subsystems; moreover, it describes the documentation process of the code and the exposed interface, through the tools of Javadoc and Swagger respectively.

4.2.1 Project configuration

Spring framework does not set any constraint related to the project directory layout, although encourages a conventional code structure as a best practice to ease development and maintainability. In the case of Veribaton, the directory layout chosen for the project complies with the standard defined by Maven[34], and more specifically represented in figure 4.2.

The structure of the directories reflects the anatomy of a typical Gradle project. As can be seen, the source code of the application resides completely in the `src` folder, under the project root. As for Maven default, java classes are placed at the path `src/main/java` inside a sub-tree of directories representing the class package they belong to. At the first level, in a root package above other classes, is placed the application main class `Application.java`, in charge of bootstrapping the service: this class defines a minimal main method in charge of simply running the Spring application, and is annotated with `@SpringBootApplication`, an annotation setting the main entry point for the framework, and enabling three features:

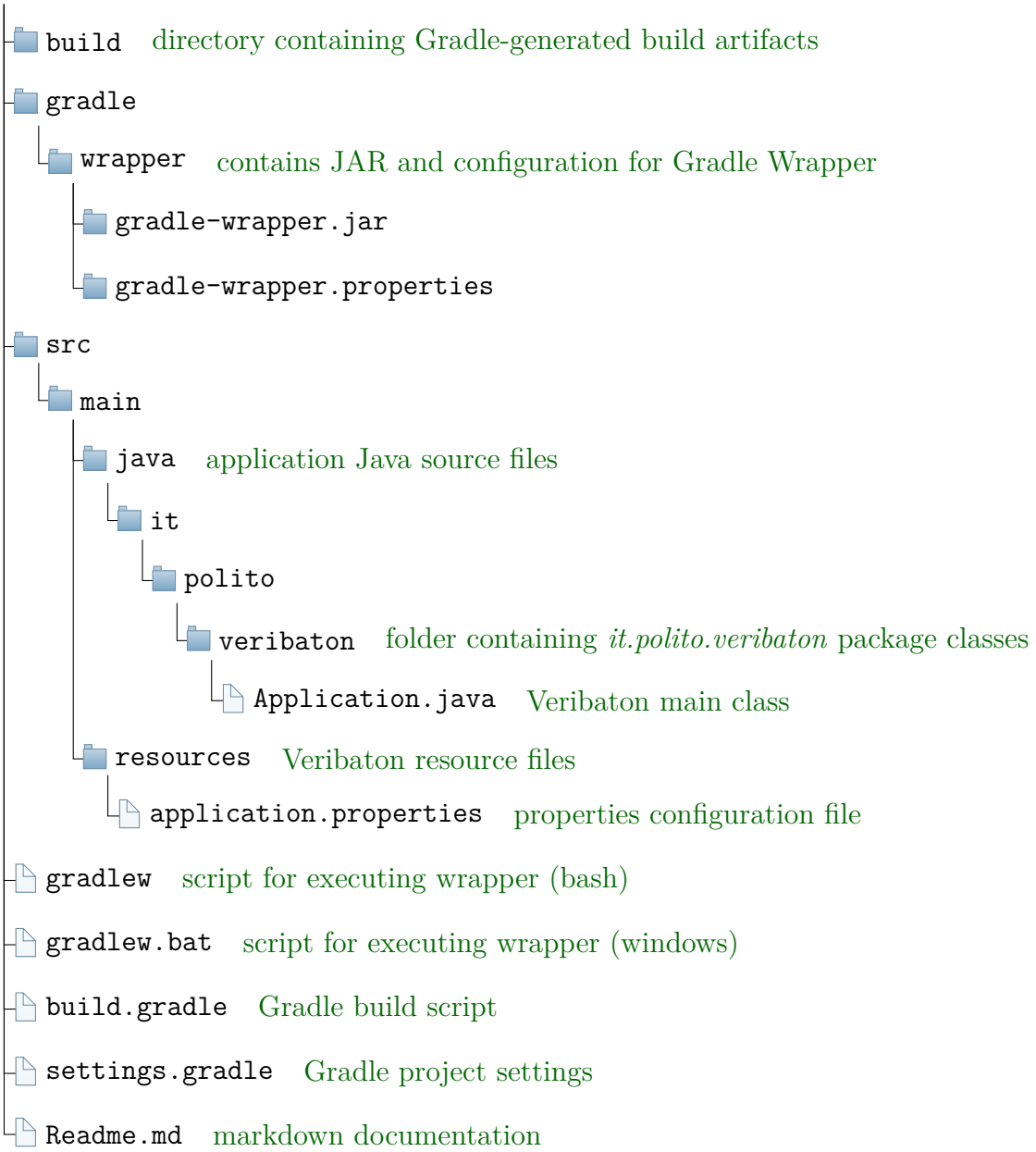

Figure 4.2: Project directory tree

- Component scan, detecting and registering `@Component` annotated classes starting from the main class package.

- Auto-configuration, for which the framework attempts to provide configuration based on dependencies included in the classpath.

- Configuration, allowing importing configuration classes.

Application resource files reside under the path `src/main/resources`: this includes static files and assets used by the application; furthermore, the framework by default reads the `application.properties` file at this location in order to load configuration options. This file contains key-value pairs separated by an equal sign as described by Java properties format that will be added to Spring Environment for later use.

At the project root, the `gradle` folder is present, containing `gradle-wrapper.jar`, a JAR file including Gradle Wrapper classes, and `gradle-wrapper.properties`, describing its respective configuration. Gradle Wrapper is the recommended system for execution of Gradle builds, which consists of a script able to invoke a specific version of Gradle, downloading it beforehand if not installed. This allows tying the project build configuration script to a specific Gradle version, assuring it is always possible to build the project even on systems in which Gradle is not installed, and always using the same version of the build tool, leading to more reliable builds. In case the Gradle version needs to be updated, it is sufficient to update the wrapper configuration file. As a result, the project has a dramatically low setup time, as it is not required to follow manual installation procedures. For this purpose, two scripts are present in the root directory, `gradlew`, a shell script for unix-based systems, and `gradlew.bat`, a batch file for Microsoft Windows operating systems, which behave as the entry point of the tasks to be executed for performing the build.

File `settings.gradle` defines the configuration required to instantiate the project hierarchy in case of multi-project builds. This is not the case of Veribaton, which consist of a single module, therefore this file is optional; however, this script allows defining the root project name, as by default it would have been defined after the name of the root folder. The `build.gradle` script is the core of the build configuration for the project. In order to enable Spring Boot specific tasks, it is necessary to add the Spring Boot Gradle plugin to the configuration through the lines described in listing 4.1.

```
1 buildscript {
2     repositories {
3         mavenCentral()
4     }
5     dependencies {
6         classpath("org.springframework.boot:spring-boot-gradle-plugin:2.0.5.RELEASE")
7     }
8 }
9
```

```
apply plugin: 'org.springframework.boot'
apply plugin: 'java'
```

Listing 4.1: Gradle Spring Boot plugin configuration.

This script adds the plugin to the classpath referencing it through the Maven central repository, an artifact storage system provided by the community, and then includes it through the `apply plugin` directive. This plugin, when used together with the `java` plugin, enables a task able to collect and package all libraries on the classpath into a single executable JAR file, which can be easily moved for deployment and used to launch the service. Moreover, provides a dependency resolver that configures all dependencies version numbers according to the selected Spring Boot version, in any case leaving the possibility to override default settings.

In order to instantiate a RESTful interface replying to incoming HTTP requests, Spring Boot is able to embed a web server component into the application, to avoid packaging the classes into a WAR archive for deployment on a servlet container platform; for this purpose, there are available a set of libraries to add as dependencies which include the basic components to build a web-enabled application, together with the HTTP server. The default bootstrap library is `spring-boot-starter-web`, including Tomcat, but it is possible to use Jetty or Undertow instead by changing it with a more specific one. To include required JARs, it is possible to create a `dependencies` block inside the `build.gradle` file as in listing 4.2.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
}
```

Listing 4.2: Gradle starter dependencies block.

To build and run the solution, Gradle offers two possible tasks: `bootJar` packages a JAR file including the service and its relative dependencies, which is possible to execute using the command `java -jar build/libs/<JAR file name>`, and the `bootRun` task, which uses the Gradle plugin to build and directly run the application. All build artifacts generated from Gradle tasks are stored in the `build` directory under the root of the project.

4.2.2 Resource representation

To comply with the principles of REST[35] architecture the HTTP service interface should be organized in resources, each of them having its own URL and representation. Therefore, it is necessary to build classes that model the web service resources, in the form of POJOs (Plain Old Java Object), a class which implements `java.io.Serializable` and is not bound to other classes, with a constructor that receives no argument, and having standard getters and setters for its properties. When a similar class is defined, Spring is able of using the Jackson JSON library to automatically serialize the class in its JSON representation, and to unmarshal received input into instances of the class.

Since Veribaton implements the resource representation of Open Baton, an open-source project which is implemented using Java language, it has been possible to import the Open Baton SDK, a library including the required data model for the NSD API and the helper methods for interaction to the NFVO, and defining the resources using Open Baton classes to ensure full compatibility. In case of update of the NFVO, it could be sufficient to update the dependency to the required version.

4.2.3 Controller methods

In Spring approach to RESTful services, a `@Controller` or `@RestController` annotation defines a class handling HTTP requests. Each controller class is typically associated with a single URL defining a resource or a collection. In Veribaton case, only one controller has been implemented, as operations defined in table 4.1 all share a common path segment, `/api/v1/ns-descriptors`.

The class `NetworkServiceDescriptorController` has been defined, and the path mapping implementation for the NSD collection controller has been achieved using the annotation `@RequestMapping("/ns-descriptors")`, which associates all methods belonging to the class to a common root URL.

Subsequently, for each operation which should be implemented in the service a method of the class is provided; two annotations allow the correct handling of the HTTP request: `@RequestMapping` is used to define the allowed methods, the media type of the operation input and output, and bindings for parameters passed in the path; `@ResponseStatus`, instead, sets the response HTTP status code upon successful completion of the request.

The arguments of handler methods can be defined as Java objects and primitives, and wired to the HTTP request body through `@RequestBody`, parameters present in the resource path with `@PathVariable`, or HTTP headers using `@RequestHeader`.

4.2.4 Externalized configuration

As Veribaton use case involves multiple moving pieces, namely the two different back-end services Verifoo and Open Baton, it has been deemed useful to adopt an approach based on configuration files to define the services URLs and settings which may vary often, especially taking into consideration different lifecycle environments such as local development, staging, and production. Spring offers a highly dynamic system for configuration externalization[36], in which configuration properties are loaded hierarchically in a particular order sensible to user overrides, to allow the same application code to work independently from the environment.

In particular, a `@SpringApplication` annotated class will check and load properties from different sources: the `application.properties` configuration file present under the folder `src/main/resources` serves as a definition of default values for variables, while property files defined outside the packaged JAR gain precedence, especially if contained in a directory `/config`. These configuration values get overridden by, in order, operating system environment variables, JVM system properties, and command line arguments. There exist other possible options, as only the most common have been described, and the load order is customizable at preference.

Table 4.5 defines Veribaton properties and their default value, configured in the `application.properties` file.

These property values can be injected into class attributes using the `@Value` annotation, and allow agile service configuration. An exception is represented by the `server.port` property, which is defined by the Spring Boot framework and determines the HTTP listener port: normally the default for the framework is 8080, it has been overridden to avoid conflicts specifically during local development, in which all services should be running on the same host.

4.2.5 Verifoo annotated classes generation

Verifoo REST service communicates through an XML-based interface in which representation of resources relies on XML Schema Definition for validation. In order to

Table 4.5: Veribaton configurable properties.

Property	Description	Default value
server.port	The TCP port the REST API server will listen on	9090
verifoo.scheme	Scheme for verifoo URL, can be http or https	http
verifoo.host	Verifoo base URL, can be a hostname or IP address	localhost
verifoo.port	Port on which Verifoo is listening	8090
verifoo.baseUri	Base URI for Verifoo REST service	/verifoo/rest
verifoo.deploymentUri	URI for deployment service on Verifoo REST API	/deployment
openbaton.host	Open Baton NFVO address, can be a hostname or IP	localhost
openbaton.port	Port on which Open Baton is listening on	8080
openbaton.username	Username provided when requesting services from Open Baton	admin
openbaton.password	Password for Open Baton user	openbaton
openbaton.ssl	Whether Openbaton REST uses https or not, can be true or false	false

interact with the service, it is necessary to obtain Java classes models of the XML document; this can be achieved using the `xjc` shell script. This tool name stands for XML to Java Compiler and it is able, based on an existing XSD schema, to generate classes annotated with JAXB (Java Architecture for XML Binding) annotation, the standard Java framework for XML processing, which can be later used for marshaling and unmarshaling of XML objects.

After importing Verifoo XSD files, `nfvSchema.xsd` together with the other schema files it references, under the resources path of the project, it is possible to execute `xjc` from the terminal shell to obtain generated source classes. However, to fully integrate the generated classes into the project lifecycle, it is necessary to add the

code generation step to the build automation, in order to make sure Verifoo classes are kept up-to-date upon Verifoo versions and schema change. Adding the `xjc` tool to Gradle script is not officially supported through a specific plugin as it does for Maven, yet there exists an Ant plugin which could be invoked by Gradle. To do so, the `jaxb` dependency should be added, and a custom task needs to be defined in `build.gradle`. In this way is possible to add in the dependency block of the Gradle script the classes built by the compiler, which will be executed each time a clean action is performed.

4.2.6 ETSI JSON to Verifoo XML conversion

Interacting with Verifoo for service validation implies converting the NSD received as an input of operations performing the catalog onboarding. This operation presents different complexities, as described in 4.1.3: this section purpose is to describe the implementation methods and choices taken to overcome such limitations.

4.2.6.1 Serialization

Spring framework controllers handle input unmarshaling from JSON format as described in 4.2.2, by automatically instantiating objects defined by resource classes, using the Jackson library. As a result, it is possible to wire the request body to a parameter in the method signature. For interaction with the NFVO instead, low-level serialization details are handled by the Open Baton SDK library: the methods signatures make use of the same classes defined as input for the controller methods.

Once instantiated the objects representing Verifoo resources, XML marshaling is handled transparently from JAXB; HTTP client functionalities are handled by using Spring `RestTemplate`, a class exposing a simple interface for most common REST scenarios, which can be used to execute requests to Verifoo.

4.2.6.2 Graph model

ETSI NSD represents the service in the form of VNFs connected to Virtual Links, while Verifoo data model is based upon the concept of adjacency between the nodes, where each element has a list of neighbors. In order to convert the graph between the two information models it has been necessary to design and implement an algorithm performing the following steps:

- **Extract networks**: all Virtual Links are extracted from the service definition, and for each link, an empty set of VNF objects is created.
- **Populate networks with nodes**: iterating over available VNFs, for each node a row in an adjacency matrix is created; then, the node is added to the set of nodes of each network it is attached to. In this way, at the end of the iteration, for each Virtual Link, there is a list of nodes connected to it, and there exists an empty list of neighbor nodes for each VNF.
- **Create adjacency matrix**: iterating over the lists of nodes connected to each network, the row of the adjacency matrix of each VNF is updated adding to it all the nodes belonging to the same network, except the node itself; duplicate nodes will not be present more than once. In this way, at the end of the step for each VNF there will be a list of neighbor nodes.
- **Define neighbors**: for each row in the adjacency matrix, create a node in Verifoo representation and add to it the list of its neighbors.

After interaction with Verifoo service the created graph might experiences modifications: the tool applies the validation logic, verifying that reachability and isolation between nodes defined as input are satisfied, and it is possible that some nodes, which have been marked as optional, are in fact unnecessary and are removed from the network service, as they do not concur to the policies fulfillment. In this case, it is necessary to apply an algorithm able to apply the same modifications to the original NS descriptor. In detail, for each optional node in Verifoo graph which has not been deployed, the corresponding VNF in the NSD is removed from the service; moreover, since Service Function Chaining support is not complete for the current release of Veribaton, Virtual Links in the original NSD are collapsed into a single link, with the purpose of guaranteeing traffic flow between VNFs.

4.2.6.3 VNF Descriptors and configuration

The difference in data structure between ETSI VNFDs and Verifoo nodes has a number of implications. In the first place, it is necessary to impose a constraint on the field `NSD -> VNFD -> type`: this is a free format string in ETSI data model, yet it should be used from the service designer to define the functional type of the corresponding Verifoo node, implemented as an enumerated value as described in section 3.2.1.1; this restricts possible values of the VNF definition to the ones supported by Verifoo. Moreover, Open Baton configuration format consists in a set

of key-value pairs, allowing uniquely scalar values as parameters; Verifoo, on the other hand, allows a specific configuration element to be defined for each VNF type, often making use of composite types for values. This results in a lower expressiveness when converting from ETSI format, and an unavoidable loss of information when reversing the process, forcing the definition of a custom serialization format for complex parameters.

In order to define Verifoo nodes on the basis of VNFDs, it has been necessary to rely on the flexibility of ETSI configuration, defining special purpose keys which would be ignored by the instantiated VNFs. Depending on the value of the `type` field, the related specific configuration is created in Verifoo graph, then it is populated by extracting the values from configuration objects in which the key string matches the property name. In case of list values, it is necessary to duplicate the keys in the entries for the NSD; this does not reflect when converting a configuration array back to the ETSI format, as values having the same key will be overridden by the NFVO.

To complete information related to the node it is also necessary to add node constraints (3.2.1): the `optional` configuration key holds a boolean value which is used to determine whether the node can be removed if considered unnecessary from the verification algorithm or not. As concerns other node constraints, e.g. CPU cores and memory, they are not yet supported in Open Baton, therefore have been ignored.

Leveraging configuration was also the strategy used for the implementation of graph properties, specifically reachability and isolation. In this case, the keys used to represent this concept are `canReach` and `cannotReach`: this allows to obtain properties definitions in which the source node is the VNF including such parameter keys, while the parameter value represents the target node.

4.2.6.4 Infrastructure

At the beginning of this thesis work the first Verifoo version has been used: this implementation required an infrastructure definition to be present in the service graph, in order to deploy nodes on a set of physical hosts and connections. Not having information about this domain in the NSD, it has been necessary to programmatically generate a dummy one for the sake of correct interaction with the service. This infrastructure definition has been generated in the following way:

- Each edge node in the service graph, specifically a *WEBCLIENT*, *WEBSERVER*, *MAILCLIENT*, *MAILSERVER*, *ENDHOST*, and *ENDPOINT*, generates a dedicated host of type *CLIENT* or *SERVER* to which it is tied to.
- A single cauldron host is created with the type *MIDDLEBOX* to accommodate all other functional types; it is defined with a high value of resources in order to avoid being a bottleneck during the execution of the scheduling algorithm.
- A mesh of connections between hosts is created.

In the following versions of Verifoo the algorithm completes successfully even in absence of hosts and interconnections, hence is not necessary to generate them.

4.2.7 Exception handling

Being Veribaton dependant on a multiplicity of external components, different exceptions can arise during operation of the tool related to errors of the downstream services, which should be interpreted for their functional value, whether they indicate an invalid service graph, or depend on different causes. Intercepting these errors correctly allows notifying the client in an explanatory way while failing to do so could result in an unusable service. Follows the detail of different scenarios which have been considered, together with their HTTP error codes[37].

- Invalid input format: in case the NSD input is not compliant to the format, the client receives an HTTP *Bad Request* error; this scenario includes the cases in which the VNF type cannot be correctly parsed as a Verifoo functional type.
- Invalid service graph: Verifoo response contains, for each property that has been verified, the *isSat* attribute, indicating whether the property was satisfiable or not; at the first unsatisfiable property, the service responds with a *Bad Request* error, containing the details of which property was not satisfied.
- Invalid Verifoo payload: upon receiving a 400 error code from Verifoo, this is forwarded to the client; the most common occurrence of this scenario is when VNFs are not correctly organized in a chain.
- Verifoo service error: if Verifoo experiences an unexpected server error, a 500 code is generated, which is dispatched to Veribaton consumer.

- Verifoo communication error: in case of communication errors, this is reported using an *Internal Server Error* response.

- NFVO operation error: due to the usage of the Open Baton SDK, interaction happens through a library method invocation, meaning that no fine-grained support for errors in provided; in any case, errors in communicating with the NFVO result in a 500 HTTP response.

For the sake of clarity, two custom exceptions have been defined in Veribaton, `InvalidGraphException` and `UnsatisfiedPropertyException`; these allow precisely defining the cause of the error.

Spring allows the customization of the error handler in order to specify the properties of responses in case of controller methods exceptions. In Veribaton case, a the `VeribatonErrorController` class has been implemented: it takes care of outputting JSON objects in the occurrence of an error. The error properties are wired automatically by the framework using the annotation `@Autowired` for the object of class `ErrorAttributes`, and include a textual representation of the timestamp the exception was raised, the HTTP code and meaning, the message available and the path of the request.

4.2.8 OpenAPI documentation

OpenAPI[38], formerly known as Swagger, is an open-source collaborative project driven by the Linux Foundation and has the purpose of creating a specification defining a standardized file format for describing a REST API. Specifications can be written in JSON or YAML[39], and the structure is intended to be both human-readable and machine-readable. On top of this format, a growing set of tools has been developed, starting from *Swagger UI*, a web interface able to create dynamic API documentation, the *Swagger Codegen*, able to automatically generate client and server code in multiple languages compliant to the RESTful interface described in the OpenAPI file, and many others.

In the context of Veribaton, this specification allows creating a live documentation of the API, tightly bound to the server implementation and hence always up-to-date with the code, useful to both programmatic and human API consumers. Through the use of *SpringFox* library[40] it is possible to automate Swagger file generation based on implemented controller methods.

In order to serve the Swagger file by means of an endpoint of the service, it is necessary to create a controller class, the `SwaggerController`, and annotate it using `@EnableSwagger2`, which will cause the endpoint to respond to HTTP *GET* requests with a Swagger JSON document, compliant to version 2 of the OpenAPI spec, built by scanning controller classes in the project and their respective methods. A configuration bean allows selecting how the documentation file is generated: gives the possibility to filter operations to include based on the package their controller is, the type of documentation, the description of the elements and more. Although OpenAPI version is currently 3.0, this library allows up to version 2 of the specification. For each controller method, it is possible to customize the name of the operation and relative description in order to have them included in the documentation in the preferred form, differently a default automatically generated will be used.

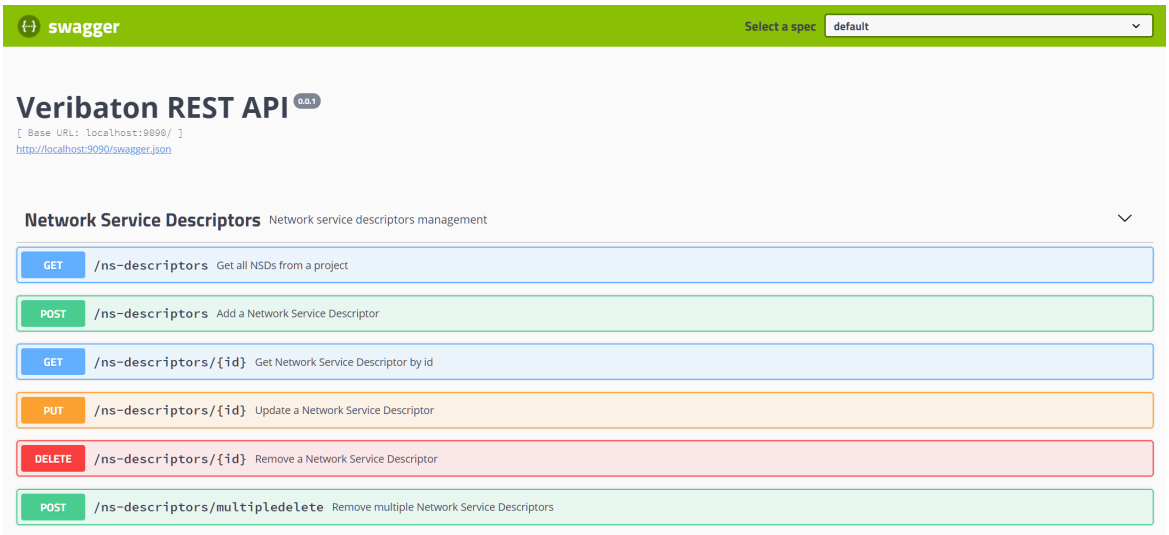

Figure 4.3: Veribaton Swagger UI page.

Adding as a dependency the library `springfox-swagger-ui` adds to the service the static files necessary to serve the Swagger UI page and adds several endpoints:

- A `/swagger.json` resource serving the Swagger JSON file.
- The `/swagger-resources` URI, which is possible to invoke to retrieve the list of all swagger resources configured for the application.
- A resource serving the Swagger UI page, at `/swagger-ui.html`.

As a result, as the Swagger UI interface is accessed by the client, it will take

care of configuring itself using information gathered using the `/swagger-resources` endpoint in order to correctly render the service API descriptor file.

For the purpose of simplifying the administrator in retrieving the Swagger UI, a redirect entry has been added, listening at the path `/swagger`, pointing to the HTML file location.

4.3 Testing and validation

With the purpose of verifying the applicability of the implemented solution, it has been necessary to create a demonstration environment in which to test the interaction between the different components; moreover, one or more Network Service Descriptor instances had to be fed as an input to the software to confirm the expectations about the service validity. As a final step, onboarded services had to be deployed to verify communication between the VNFs, and the correct behavior of the orchestrator in terms of VNF instantiation and configuration.

4.3.1 Environment setup

To set up a demonstration environment for Veribaton, two different physical hosts have been used: a laptop, on which the solution has been developed and tested, and a host on which Open Baton and Verifoo have been deployed. Follows the hardware specifications of the hosts.

Laptop:

- **CPU**: Intel i7 5500U @ 2.40 GHz
- **RAM**: 8 GB DDR3
- **OS**: Ubuntu Linux 18.04.2 LTS (Bionic Beaver)

NFVO host:

- **CPU**: Intel Core Duo Q8300 @ 2.50 GHz
- **RAM**: 8 GB DDR2
- **OS**: Ubuntu Linux 18.04.2 LTS (Bionic Beaver)

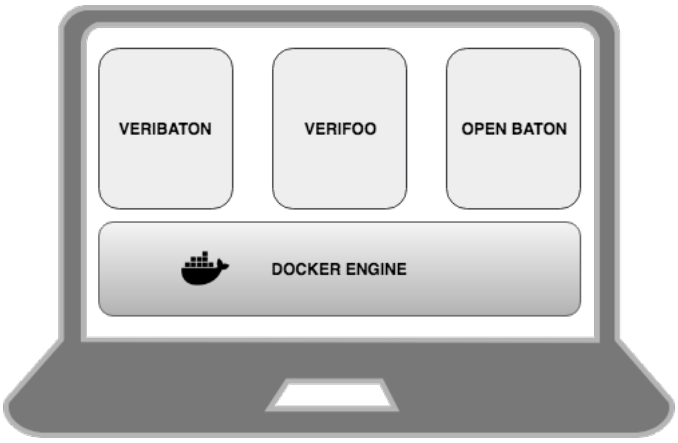

Figure 4.4: Local testing environment.

Two different testing configurations have been realized to verify different use case scenarios:

- **Local**: all services are located on the same host. Figure 4.4 visually describes this scenario, in which all components have been installed in the laptop machine, and their connectivity is achieved through the use of the loopback interface of the network card.
- **Remote**: Veribaton acts as a client application consuming REST APIs of the external services located on a different host. In this scenario, Veribaton is installed on the laptop while other services are configured on the NFVO host; the two hosts are connected through their network interface to the same LAN, as shown in figure 4.5

The different use cases derive from the low resources available on the development environment, therefore it has been necessary to separate the components when testing instances of service graphs with several nodes. Moreover, the remote configuration allows verifying that connectivity does not represent an issue for the operation of the solution.

4.3.1.1 Docker VIM

Docker[6] is an operating-system-level virtualization platform, able to create and manage container instances running in isolation. The Docker Engine, the core tool of the platform, is a client-server application based on multiple components: a daemon

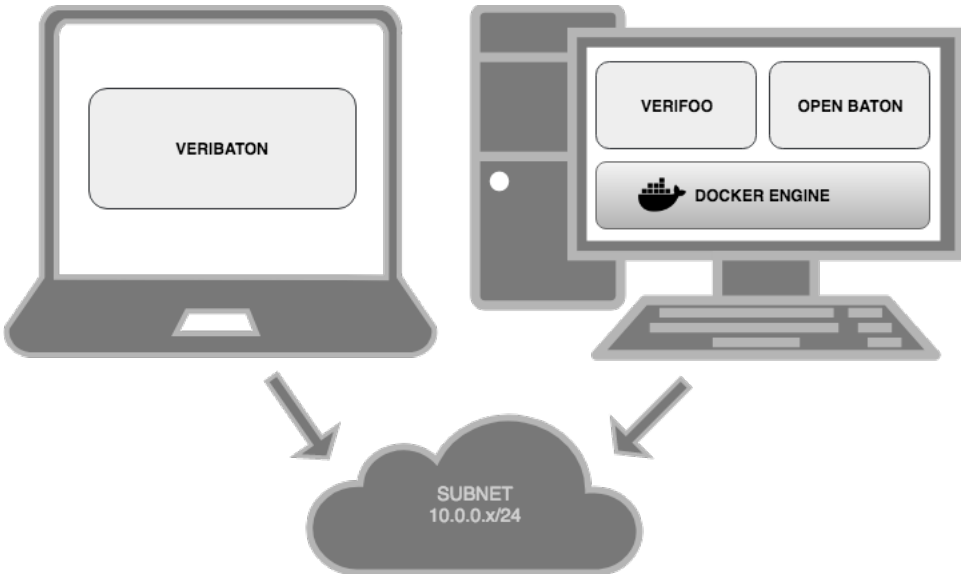

Figure 4.5: Remote testing environment.

server process executing persistently in the background, called `dockerd`, which is in charge of containers lifecycle and all resources related such as storage volumes and networking; a REST API exposing a uniform interface to interact with the daemon and send commands to it; and a Command Line Interface (CLI) as a user tool for interacting with the API to perform administration tasks.

Docker creates containers based on *images*, which are templates for container instances: they contain a snapshot of the filesystem of an application, including all required libraries and executables, and a set of instructions to start the packaged software, relying on the host kernel to be executed. Docker images are commonly stored in *registries*, repositories from where the Docker daemon is able to import them and execute the related container instances; the Docker Hub is a publicly available registry where both official images and ones created by the users are uploaded and available to the community, and Docker is configured by default to use it to search for images, even if it is possible to modify the setting to address a private repository.

As described in section 3.3.3.2, it is possible to use Docker as Virtualized Infrastructure Manager, as it allows the creation of VNFs based con container images, and is able to configure networking in order to allow connectivity between containers. In order to do this in Veribaton environment, it has been sufficient to install the software on the physical hosts.

4.3.1.2 Open Baton

As a consequence of its architecture, Open Baton is consisting of multiple components in communication through an instance of the RabbitMQ message broker, which should be installed separately and configured to interact with each other. To solve the complexity of the setup, the Open Baton team chose as the preferred installation solution *Docker Compose*, a tool which allows defining and running applications composed by multiple Docker containers. It requires the Docker Engine to be installed on the host

Leveraging this tool, it is relatively easy to install the different components using a definition file, written in YAML format, containing the description of all necessary containers together with their configuration options. The minimum configuration of installed components to instantiate network services on the Docker virtual environment is composed of:

- **NFVO**: the main component, allows service orchestration and funnels user interactions.
- **RabbitMQ Broker**: the backbone of Open Baton, allowing communication between the components.
- **NFVO Database**: persists data for the NFVO, such as descriptor catalog and state of deployed services.
- **Docker VNFM**: allows configuration of container instances on top of an installed Docker Engine; depends on the Docker VIM driver.
- **Docker VIM Driver**: together with the Docker VNFM allows interaction with the Docker Engine for containers and networking instantiation.

To use the Docker Engine instance as VIM is necessary to onboard it as a PoP: communication happens locally and is achieved through UNIX sockets. A VIM instance can be uploaded using the payload specified in listing 4.3.

It is then possible to reference the PoP during deployment through the value used for the *name* field. In case the Docker daemon is configured to accept unauthenticated connections, the *username* and *password* fields are not required and can be substituted by a blank string.

```
{
  "name": "vim-instance",
  "authUrl": "unix:///var/run/docker.sock",
  "username": "admin",
  "password": "openbaton",
  "type": "docker"
}
```

Listing 4.3: Docker VIM instance ondoarding payload.

4.3.1.3 Verifoo

Verifoo is implemented as a Java Enterprise Edition application, exposing a RESTful interface through the JAX-RS (Java API for RESTful Web Services) interface, in particular using the Jersey library. For this reason, in order to deploy the service it is necessary to package compiled class files into a WAR archive and deploy it to a servlet container; according to Verifoo documentation the supported tool is Apache Tomcat[41], an implementation of several Java EE specifications, among which the Java Servlet technology, and provides an HTTP web server environment listening for TCP connections and on which Java code can be executed.

Verifoo can be installed on a Tomcat instance either statically, by moving the WAR file in a specific folder of the host, or dynamically, by uploading the archive containing compiled classes through the Tomcat manager application, offering an interface for deployment management both programmatically and interactively. In particular, Verifoo makes use of the Ant[42] build tool that includes a set of tasks for automatic distribution of the application on a Tomcat environment.

Installation of Verifoo involves the following steps:

- Installation of the Java Runtime Environment version 7 or later.
- Installation of Apache Tomcat version 8.5.
- For the correct functioning of the application, it is necessary to install and include in the Java Library Path the Microsoft Z3 native library.
- Deployment of the application using an already packaged version or by executing the Ant script tasks `ant deploy` or `ant deployWS`.

4.3.1.4 Veribaton

The only prerequisite for Veribaton installation is an installed Java Runtime Environment. Execution of the application is achievable in two different ways:

- **JAR deployment**: Veribaton classes can be packaged as an executable JAR file by the means of the Gradle task `bootJar`. This archive can be used to bootstrap the service using the command `java -jar veribaton.jar`.
- **Gradle plugin**: the Gradle task `bootRun` compiles and executes the service starting from the sources.

The service can be configured, as described in section 4.1.1.2, by providing a properties file overriding default parameter values, which are provided to support the local development environment. The remote use case scenario needs the properties related to the network addresses of external services to be overridden accordingly to the environment configuration.

4.3.2 NSD test instances

To verify the capabilities of the software has been necessary to design several NSD instances representing common use cases of the tool and feed them as input to Veribaton. During the testing of the solution, Verifoo upgraded from version 1.0 to version 2.0: in the following sections are highlighted the differences in service graph design for the two renditions of the tool.

4.3.2.1 Verifoo 1.0

NSDs implemented while integrating Verifoo 1.0 consider the limitations imposed by the version of the tool:

- **Link redundancy**: in order to exploit Verifoo capability to detect and remove unused nodes for the satisfiability of the properties given as input, it is possible to flag nodes as optional. It is necessary, however, to provide an additional connection between the node preceding the optional node and the one following it, because it does not have the capability of recreating the adjacency between VNFs in case of removal of a node. While this issue can affect smaller instances

in a lower measure, for graphs with a higher number of nodes this results in a proliferation of connections not useful to the purposes of the network services, and seriously affect service design. For this reason, only NSDs with a relatively low number of nodes have been considered, and each optional node presents the additional neighbors necessary to complete the node removal functionality.

- **Autoconfiguration issues**: Verifoo has the capability of computing and applying the correct configuration for nodes in accordance with the properties specified in the NFV object, with particular attention to the VNFs having the *FIREWALL* functional type. More specifically, it can detect the default routing policy for packets in the form of denying or allowing traffic forwarding, and the rules necessary to satisfy the required conditions. In this version, some of these features are not always behaving as expected, therefore it has been necessary to adapt the network service design according to the presented issues.

- **Performance**: in some specific cases it has been detected that graph verification demonstrated performance issues affecting the user experience in a substantial way. A memory leak has been identified in the construction of the NAT VNF model preventing validation of instances including a NAT to complete successfully; in other cases, graphs with less than ten nodes, therefore instances which are considered minimal, experienced a considerably high solution time, so that Veribaton would encounter a time-out without providing responses to the user.

Having reported instances affected by these issues as test instances, it has been possible to verify that in the subsequent release of the software they have been addressed correctly. Test cases performed are detailed in the following sections.

4.3.2.1.1 Scenario 1A Figure 4.6 describes visually the configuration of this test instance including two web client nodes, two firewalls and one NAT VNFs connecting to a web server.

- Reachability is required between node A and node B.
- Reachability is required between node C and node B.
- Node 1 and node 3 are optional and are not configured.
- Node 2 is required.

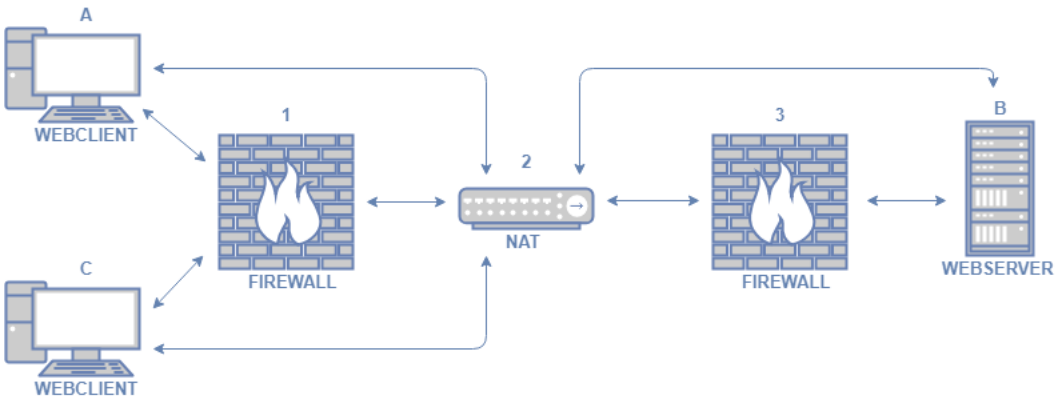

Figure 4.6: NSD with two web clients, two firewalls, one NAT and one web server VNFs, with node connections redundancy.

The test instance completes successfully behaving as expected: both node 1 and node 3 are removed as they are not necessary for guaranteeing reachability.

4.3.2.1.2 Scenario 1B This scenario node configuration is kept equivalent to the one described in figure 4.6, with the following properties:

- Reachability is required between node A and node B.
- Isolation is required between node C and node B.
- Node 1 and node 3 are optional and are not configured.
- Node 2 is required.

In this case, the test is not successful: again both node 1 and node 3 have been removed, yet to ensure isolation between node C and node B it is necessary to introduce at least a firewall instance in the service graph.

4.3.2.1.3 Scenario 1C The NSD for this scenario is described in figure 4.7; is composed of three we client nodes, three firewalls connected to two web servers. The properties of the graph are as follows:

- Reachability is required between node A and node B.
- Isolation is required between node C and node B.

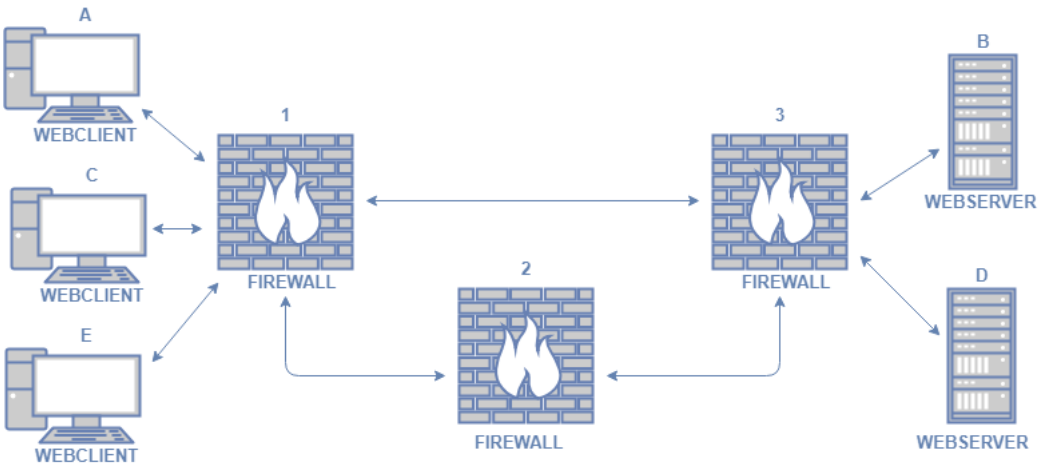

Figure 4.7: NSD with three web clients, three firewalls, and two web server VNFs.

- All firewall nodes are required.

This service instance does not complete as receives an *Internal Server Error* from Verifoo. According to the service logs, it depends on an error invoking the Z3 library through the JNI interface.

4.3.2.2 Verifoo 2.0

In the following version of Verifoo, restrictions enforced by the tool have been overcome, leading to the test instances described in the following paragraphs.

4.3.2.2.1 Scenario 2A As shown in figure 4.8, test case involving the same service graph as scenario 2B, yet removing redundant links. The settings of the service are:

- Reachability is required between node A and node B.
- Isolation is required between node C and node B.
- Node 1, node 2 and node 3 are optional and are not configured.

As a result, service validation is successful; as expected, node 3 has been removed, while node 1 has been configured to allow traffic directed from node A to node B, denying everything else.

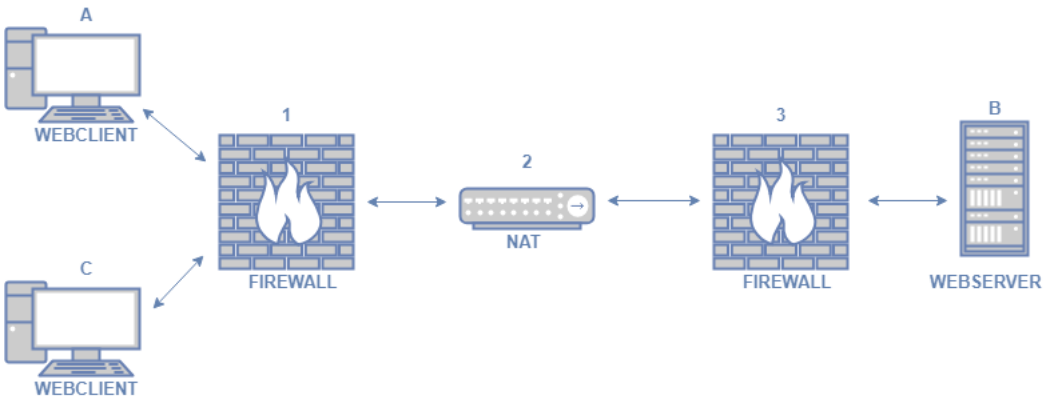

Figure 4.8: NSD with two web client nodes, two firewalls, one NAT, and one web server.

4.3.2.2.2 Scenario 2B This test case makes use of the service definition shown in figure 4.7; configuration follows:

- Reachability is required between node A and node B.
- Reachability is required between node E and node B.
- Isolation is required between node C and node B.
- Firewall nodes are required and not configured.

As expected, none of the nodes has been removed; firewall nodes 1 and 2 are both configured to allow connections between node A and node B, and from node E to node B, while blocking the rest of the traffic. Firewall 3 has been left unconfigured.

4.3.2.2.3 Scenario 2C This scenario modifies the properties of the service defined in figure 4.7 by flagging all middlebox VNFs as optional:

- Reachability is required between node A and node B.
- Reachability is required between node E and node B.
- Isolation is required between node C and node B.
- Node 1, node 2 and node 3 are optional and are provided an empty configuration.

The service graph properties result satisfiable; node 1 and node 3 are removed as not necessary, while firewall VNF 2 is configured to forward packets from A to B and from E to B, dropping all the others.

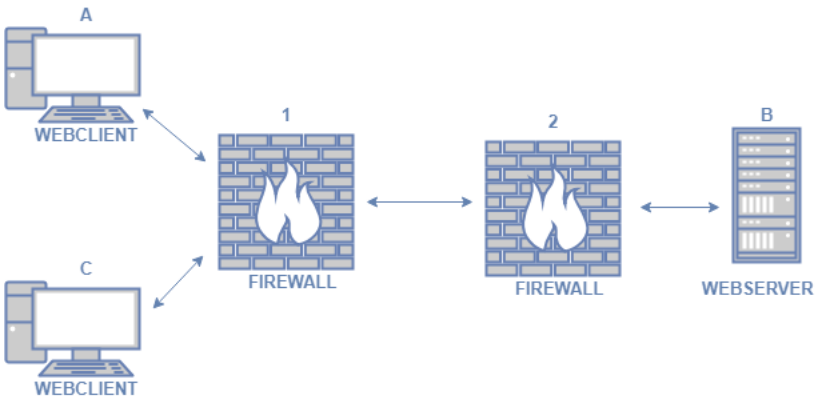

Figure 4.9: NSD with two web client nodes, two firewalls, and one web server.

4.3.2.2.4 Scenario 2D This use case is represented by figure 4.9, including a simpler instance composed of two web clients, two firewalls and a web server instance. The service is configured as follows:

- Reachability is required between node A and node B.
- Isolation is required between node C and node B.
- Node 2 is optional.
- Node 1 is required but provided without configuration.

The service instance completes successfully, node 2 is removed and the firewall instance is configured in order to allow traffic from A to B and deny traffic from C to B.

4.3.2.2.5 Scenario 2E For this test case the same node chaining defined in figure 4.9 is used, yet is intended to provide an incorrect configuration by using a conflicting set of properties to be verified. The properties follow:

- Reachability is required between node A and node B.

- Isolation is required between node A and node B.
- Isolation is required between node C and node B.
- Node 2 is optional.
- Node 1 is required but provided without configuration.

As can be noticed the first two properties conflict; it is expected to receive as output the notification of the faulty graph instance, either by flagging one of the two as not satisfied, or returning a service error. The actual output shows the instance as successfully validated: this highlights a fault in Verifoo conflict solving algorithm.

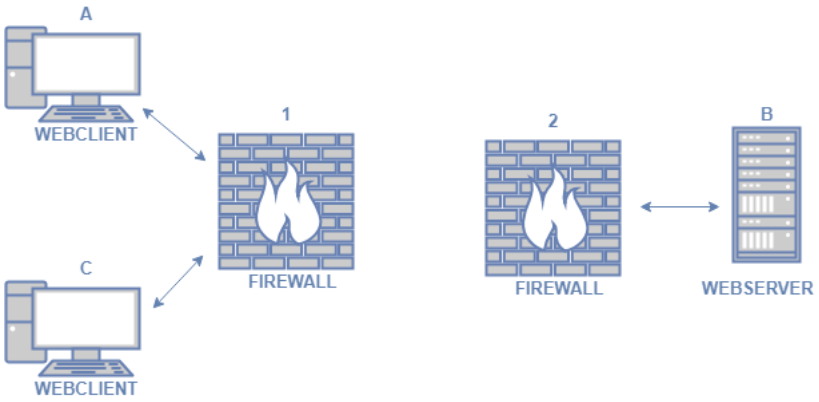

Figure 4.10: Faulty NSD with two web client nodes, two firewalls not connected to each other and one web server.

4.3.2.2.6 Scenario 2F This test scenario used a faulty NSD in order to test the detection of incorrect design of graph instances. In figure 4.10, representing the considered graph, it is possible to notice the absence of a link connecting node1 to node 2. This is expected to be identified as an issue causing the NSD to be rejected from onboarding. The configuration used is:

- Reachability is required between node A and node B.
- Isolation is required between node C and node B.
- Node 2 is optional.
- Node 1 is required but provided without configuration.

As intended, the service responds with an error stating the incompleteness of the graph as nodes should be connected with each other.

4.3.3 VNF implementation

Deployment of the network services test cases required implementation of a minimal working example of VNF software to verify the platform capabilities and the interaction between the orchestrator and Docker containers, as well as connectivity and networking. For this reason, it has been necessary to implement Docker images supporting the service scenarios described in section 4.3.2, with particular interest to the firewall and web client VNF.

The configuration of VNFs, from the point of view of the orchestrator, can be performed in two possible cases, not mutually exclusive:

- **Startup**: the VNF instances are created having a set of environment variables corresponding to the configuration keys defined in the VNFD; these variables are set to the value of the parameter and can be read by the software running in the virtualized instance to define its configuration.
- **Runtime**: after VM creation, lifecycle scripts can be executed on the instance to obtain the desired state.

Since Docker VNFM does not allow the execution of lifecycle scripts because of the characteristics of the container platform, which encourages packaging the application in images that should be run off the shelf, it has been necessary to define VNF software able to leverage environment variables passed to the container instance for configuration.

4.3.3.1 Web Client

In order to deploy a container instance acting as a web client which could be accessed interactively by the administrator to test connectivity, a Docker image had to be developed so as the deriving containers could be able to be launched in a detached state. Moreover, to compensate for the lack of Service Function Chaining capabilities of the orchestrator, the VNF routing had to be configured in order to forward the traffic through the correct node in the chain, as all the containers in the graph are required to be attached to the same virtual link for reachability purposes.

```
FROM node:latest

RUN apt-get update && apt-get install -y \
  bridge-utils \
  net-tools \
    inetutils-traceroute

ADD run.sh /tmp/run.sh
RUN chmod +x /tmp/run.sh
ENTRYPOINT ["/tmp/run.sh"]
```

Listing 4.4: Web Client VNF Dockerfile.

The Docker file used to build the image is shown in listing 4.4.

As it is possible to note, the official Node.js image has been used as a base, assuming a simple JavaScript client; as this image is not configured to be launched in a detached state, yet it runs with the specified parameters before exiting, it has been necessary to add a shell script executing in the background an infinite loop, preventing the VNF to exit.

As shown in listing 4.5, in addition the script configures the routing table so as packets directed to the address stored in variable `nameWebServer` are sent through the host which name is contained in the environment variable `netxHop`. This has the purpose of providing a way of configuring the VNF based on the information gathered using the Verifoo service.

```
#!/bin/bash
ip route add "$(getent hosts "$nameWebServer" | awk '{ print $1 }')" via
    "$(getent hosts "$nextHop" | awk '{ print $1 }')"
while true; do sleep 15 ; done
```

Listing 4.5: Web Client *run.sh* script.

4.3.3.2 Firewall

For the firewall VNF implementation, a simple iptables-based firewall has been used, which could be configured to read the environment to apply iptables policies for packet filtering.

As Verifoo configuration for firewall nodes is composed by a list of complex entries, each containing information concerning the action to perform based on source

and destination address of the packets, it is necessary to define a simple serialization format to include all available information as a string value to be used as a configuration parameter. In addition, the VNF software needs to be able to correctly parse the information contained in the environment and apply corresponding iptables filters.

```
FROM glanf/base:latest

ADD run.sh /tmp/run.sh
RUN chmod +x /tmp/run.sh
ENTRYPOINT ["/tmp/run.sh"]
```

Listing 4.6: Firewall VNF Dockerfile.

Listing 4.6 reports the Dockerfile used to build the firewall image; as a base has been used an open-source image provided by the Network Laboratory of the Glasgow University, in the context of the Glasgow Network Functions (GNF) project[43]. On top of this, a custom script has been added to allow parsing of the configuration variables and applying the related firewall rules.

```
#!/bin/bash
# $deny and $allow are in form src1,dst1;src2,dst2
IFS=';' read -ra ADDR <<< "$deny"
for i in "${ADDR[@]}"; do
    IFS=',' read -ra SRCDST <<< "$i"
    iptables -A FORWARD -s "${SRCDST[0]}" -d "${SRCDST[1]}" -j REJECT
done
# IFS=';' read -ra ADDR <<< "$deny" ...
if [[ $defaultAction = [Aa][Ll][Ll][Oo][Ww] ]]; then
    iptables -t filter -A FORWARD -j ACCEPT
elif [[ $defaultAction = [Dd][Ee][Nn][Yy] ]]; then
    iptables -t filter -A FORWARD -j REJECT
fi

while true; do sleep 15 ; done
```

Listing 4.7: Firewall *run.sh* script.

As it can be read in listing 4.7, the variable `deny` is expected to contain a set of key-value pairs separated by the semicolon character; each pair, in turn, represents a couple of source and destination addresses that the firewall should avoid forwarding, therefore the appropriate rule is applied to iptables. Although omitted for brevity, the same procedure is performed for the `allow` variable, this time permitting the

traffic through the iptables rule. At the end of the startup script, the variable `defaultAction` is checked to verify whether in case packets reach the end of the processing chain they should be dropped or forwarded.

4.3.4 Service deployment

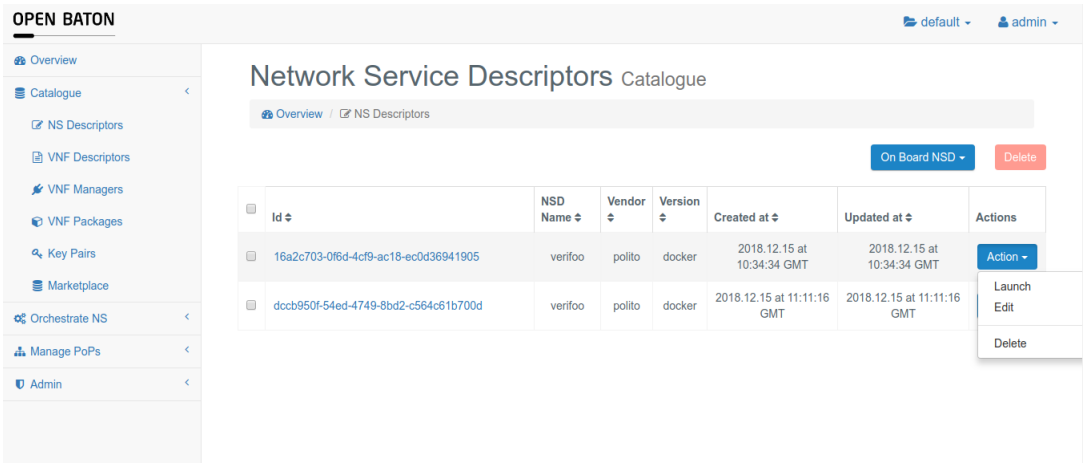

Figure 4.11: Open Baton deployment console.

Test NSD instances that completed validation successfully are then modified according to Verifoo service response and uploaded on Open Baton catalog for deployment. An administration user which is able to access the management console can subsequently verify the presence of the instance in the catalog and launch the network service as shown in figure 4.11.

During instantiation, it is possible to choose for each VNF the PoP on which it will be located; given the nature of this thesis work, the only available possibility will be the local Docker Engine onboarded during the environment configuration. For the correct completion of deployment, it is necessary for the Docker PoP to import the images used in the VDU section of the VNF, by pulling them from a registry or building the Dockerfiles.

Upon deployment, each VNF lifecycle can be followed through the Network Service Records section of the administration dashboard: assuming instantiation completed, each of the nodes should be reported in the *ACTIVE* state. It is possible to verify directly on the VIM that containers have been created and are running; by issuing the command `docker ps` on the host, a container process will be shown for each VNF. Moreover, the creation of the virtual link between containers can be

verified using `docker network ls`; this command will list all available virtual networks for the Docker Engine, among which a link of type *bridge* can be noticed.

A connectivity test can be performed from the point of view of the VNF by interacting directly with the container. It is possible to log on a container instance using the command `docker exec -it <node_name> bash`. This allows the execution of the *bash* program on a specified container; the flags used, *i* and *t* indicate respectively that the program should be run interactively, meaning that it will keep expecting input from the STDIN until explicitly detached, and that a pseudoterminal will be allocated for the command. Inside the container, it is then possible to perform actions aimed to verify the network service has been configured correctly. As an example, with reference to the service instances described in section 4.3.2, the following activities can be carried out:

- Verifying reachability and isolation by logging on the nodes defined as source in the property; executing inside the bash session a *curl* or *ping* command specifying the target node should lead to a failure in case isolation is requires, and to success in case of reachability. This should be satisfied as it derives from formal verification.
- Checking routing policies; executing the *traceroute* program from a web client container targeting the corresponding web server should demonstrate that traffic is conveyed through the host configured as a next hop. In case of incorrect VNF configuration, as containers are attached to the same virtual link, packets flow directly to the target node.
- Examine iptables routing policies logging on the firewall VNF container; this information should correspond to what is defined in the *allow*, *deny* and *defaultAction* environment variables.

Chapter 5

Conclusions

The goal of this thesis work was to integrate external graph verification capabilities provided by Verifoo to a tool deriving from the open-source community and able to be used in a telecommunication service scenario. For this purpose, Veribaton has been developed, successfully presenting a solution capable of translating the information model based on the ETSI specifications to the application-specific format defined by the verification engine albeit the substantial differences in data representation and structure.

However, it has not been possible to integrate all Verifoo features; specifically, given the nature of orchestrators analyzed in the open-source panorama and their interaction with the NFVI management platforms, it is not possible to perform the resource allocation schedule in place of the VIM. Even considering infrastructure managers offering the possibility of deployment host selection, ETSI-defined interfaces offer no support and are not suited for this interaction format with the NFVI.

As concerns Verifoo autoconfiguration capabilities, it has been possible to enrich the VNFD based on the computed information; although, it must be noted that the configuration model of the orchestration tools available as open-source software an based on ETSI principles is designed to be agnostic to the chosen VNF implementation, leaving to the service designer the responsibility of applying specific parameters according to the interfaces exposed by the virtual appliances performing the network function. Configuration options offered by Verifoo are defined on the basis of the functional type of the VNF, therefore is necessary to adapt them to fit the chosen implementation of each VNF type, actually departing from the concept of vendor neutrality professed by ETSI. This limitation has been overcome for the

purposes of this thesis work by developing VNF software exposing an interface based on autoconfiguration settings obtained by Verifoo, yet in a different context, it could be necessary to design a solution suitable to be applied to a cross-vendor scenario. Furthermore, at the moment this feature can be exploited only for firewall nodes; to obtain a complete framework for graph optimization, support of additional types of VNF will represent a significant enhancement in future applications.

It should be taken into account, nevertheless, that Verifoo is to be considered still in development, and therefore imposes some limitations; during the progress of this work, issues arisen in its usage have contributed to identify aspects presenting the possibility for improvement, leading to a major release. Despite that, as demonstrated in the previous chapters, the tool has not yet reached the state of the art, allowing a considerable margin of refinement.

The proposed solution leaves open a wide array of development prospects. The lack of Service Function Chaining and SDN support in Open Baton imposed several constraints during service design, which could be overcome upon implementation of this feature in the platform, and might be refined if considering different NFV orchestrators. An interesting opportunity is represented by the possibility of coordinating a multi-PoP infrastructure: leveraging resource allocation capabilities of Verifoo, an optimal deployment plan can be computed in a geographically-distributed network service scenario, taking into account resource availability over a distributed NFV Infrastructure, and connectivity properties such as latency and bandwidth between multiple sites, opening the way for WAN-based use cases such as VNF placement over public cloud services providers infrastructure.

In conclusion, Veribaton might be considered a starting point for enrichment of MANO orchestration software; its design principles, unitedly to the modular structure and ease of extensibility, allow it to be used as a basis to a comprehensive framework for formally-verified network services, which can be integrated in a lean and unobtrusive way with orchestration platforms available on the market.

Bibliography

[1] *Verifoo github repository*. URL: https://github.com/netgroup-polito/verifoo.

[2] Rajeshwari Ganesan et al. "Empirical study of performance benefits of hardware assisted virtualization". In: Aug. 2013. DOI: 10.1145/2522548.2522598.

[3] Pradeep Padala et al. "Performance Evaluation of Virtualization Technologies for Server Consolidation". In: *Technical Report HPL-2007-59R1, HP Laboratories* (Jan. 2007).

[4] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering* (2015), pp. 386–393.

[5] George Collins and Yahav Biran. "Multi-tenant utility computing with compute containers". In: Sept. 2015, pp. 213–217. DOI: 10.1109/ICCE-Berlin.2015.7391238.

[6] Dirk Merkel. "Docker: lightweight Linux containers for consistent development and deployment". In: *Linux Journal* 2014 (Mar. 2014).

[7] *Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta*. URL: https://aws.amazon.com/it/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/.

[8] Christine Miyachi. "What is" Cloud"? It is time to update the NIST definition?" In: *IEEE Cloud Computing* 5.3 (2018), pp. 6–11.

[9] Peter Mell and Timothy Grance. "National Institute of Standards and Technology, "The NIST Definition of Cloud Computing"". In: *Nist Special Publication* 145 (Jan. 2011).

[10] S. Azodolmolky, P. Wieder, and R. Yahyapour. "SDN-based cloud computing networking". In: *2013 15th International Conference on Transparent Optical Networks (ICTON)*. June 2013. DOI: 10.1109/ICTON.2013.6602678.

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

[12] ETSI. *Network Functions Virtualisation (NFV); Architectural Framework*. Oct. 2013. URL: https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf.

[13] Faqir Zarrar Yousaf et al. "Cost analysis of initial deployment strategies for virtualized mobile core network functions". In: *IEEE Communications Magazine* 53 (2015), pp. 60–66.

[14] ETSI. *Network Functions Virtualisation (NFV); Management and Orchestration*. Dec. 2014. URL: https://www.etsi.org/deliver/etsi_gs/nfv-man/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf.

[15] *Open Baton architecture diagram*. URL: https://openbaton.github.io/documentation/images/openbaton-release-5.png.

[16] *Open Baton website*. URL: http://openbaton.github.io/.

[17] *RabbitMQ Website*. URL: https://www.rabbitmq.com/.

[18] *Open Baton identity management documentation*. URL: https://openbaton.github.io/documentation/security/.

[19] *Open Baton REST API documentation*. URL: https://openbaton.github.io/documentation/api/.

[20] *Open Baton Generic VNFM sequence diagram*. URL: http://openbaton.github.io/documentation/images/generic-vnfm-or-vnfm-seq-dg-v2.png.

[21] *Juju website*. URL: https://jujucharms.com/.

[22] *Juju charm store*. URL: https://jujucharms.com/store.

[23] *Open Baton VNFM interaction sequence diagram*. URL: http://openbaton.github.io/documentation/images/nfvo-rest-vnfm-seq-dg-v2.png.

[24] *Open Baton OpenStack VIM driver documentation*. URL: http://openbaton.github.io/documentation/openstack-driver/.

[25] *Zabbix website*. URL: https://www.zabbix.com/.

[26] ETSI. *Network Functions Virtualisation (NFV); Management and Orchestration; Or-Vi reference point - Interface and Information Model Specification*. Apr. 2016. URL: https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/005/02.01.01_60/gs_nfv-ifa005v020101p.pdf.

[27] *Drools website*. URL: https://www.drools.org/.

[28] *Openbaton plugin development documentation*. URL: https://openbaton.github.io/documentation/how-to-register-event/.

[29] Mohammad Abu-Lebdeh et al. "NFV orchestrator placement for geo-distributed systems". In: *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)* (2017), pp. 1–5.

[30] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[31] Mathieu Fourment and Michael R Gillings. "A comparison of common programming languages used in bioinformatics". In: *BMC bioinformatics* 9.1 (2008), p. 82.

[32] *JAX-RS API JSR*. URL: https://jcp.org/en/jsr/detail?id=339.

[33] *Spring Boot overview*. URL: https://spring.io/projects/spring-boot.

[34] *Introduction to Maven Directory Structure*. URL: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html.

[35] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture". In: *ACM Trans. Internet Technol.* 2.2 (May 2002). DOI: 10.1145/514183.514185. URL: http://doi.acm.org/10.1145/514183.514185.

[36] *24. Externalized Configuration*. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html.

[37] Roy Fielding and Julian Reschke. "RFC 7231-Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content". In: *Internet Engineering Task Force (IETF)* (2014).

[38] *Open API website*. URL: https://www.openapis.org/.

[39] *YAML website*. URL: https://yaml.org/.

[40] *SpringFox Documentation*. URL: https://springfox.github.io/springfox/docs/current/.

[41] *Apache Tomcat home page*. URL: https://tomcat.apache.org/.

[42] *Apache Ant home page*. URL: https://ant.apache.org/.

[43] *Glasgow Network Functions (GNF)*. URL: https://netlab.dcs.gla.ac.uk/projects/glasgow-network-functions.

[44] Rashid Mijumbi et al. "Network Function Virtualization: State-of-the-Art and Research Challenges". In: *IEEE Communications Surveys & Tutorials* 18 (Sept. 2015). DOI: 10.1109/COMST.2015.2477041.

[45] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." In: vol. 17. Jan. 1973, p. 121. DOI: 10.1145/361011.361073.

[46] Sapan Gupta and Deepanshu Gera. "A comparison of LXD, Docker and Virtual Machine". In: *International Journal of Scientific & Engineering Research* 7.9 (2016).

[47] *Lightweight Virtualized Containers for Network Function Virtualization (NFV)*. Mar. 2018. URL: https://software.intel.com/en-us/articles/lightweight-virtualized-containers-for-nfv.

[48] Moritz Raho et al. "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing". In: *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in*. IEEE. 2015, pp. 1–8.

[49] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[50] *Software-Defined Networking (SDN) Definition*. URL: https://www.opennetworking.org/sdn-definition/.

[51] Q. Duan, N. Ansari, and M. Toy. "Software-defined network virtualization: an architectural framework for integrating SDN and NFV for service provisioning in future networks". In: *IEEE Network* 30.5 (Sept. 2016). ISSN: 0890-8044. DOI: 10.1109/MNET.2016.7579021.

[52] Diego Kreutz et al. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76.

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

[54] *Open Baton VNF Package documentation*. URL: https://openbaton.github.io/documentation/vnf-package/.

[55] Michael Till Beck and Juan Botero. "Scalable and Coordinated Allocation of Service Function Chains". In: *Computer Communications* 102 (Oct. 2016). DOI: 10.1016/j.comcom.2016.09.010.

Acknowledgements

The path towards this work has been winding and tangled; its completion is in the large part due to the amazing people who have supported and motivated me to pursue this goal.

My first thanks goes to my parents, whose incredible strength inspired me to never surrender in the face of difficulties. Without my mom and dad, and their infinite patience and love, this work would never have seen the light.

Thanks to my friends, who shared with me moments of joy and sorrow. As an old Irish proverb states, a good friend is like a four-leaf clover: hard to find and lucky to have, and I am incredibly lucky to have all of you.

I would like to express the deepest appreciation and the most sincere thanks to Irene, the person who most of all has been able to understand, support and encourage me during the hardest times. Thanks for believing in me, without you, none of this would have been possible.

Finally, my biggest thought is for Dalia, whose smile will always lead my way. This work is for you.