ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Design and implementation of an ETSI Network Function Virtualization compliant container orchestrator

---

Thesis Supervisor:
Prof. BELLAVISTA Paolo

Candidate:
CILLONI Marco

Thesis Advisors:
Dr. CARELLA Giuseppe
Prof. FOSCHINI Luca

III Session
2016/2017

Contents

List of Figures

List of Tables

Acronyms

API	**A**pplication **P**rogramming **I**nterface
E2E	**E**nd To *(*"**2**"*)* **E**nd
EM	**E**lement **M**anagement
EMS	**E**lement **M**anagement **S**ystem
IP	**I**nternet **P**rotocol
NF	**N**etwork **F**unction
NFV	**N**etwork **F**unction **V**irtualisation
NFV-MANO	**NFV M**anagement **AN**d **O**rchestration
NFVI	**N**etwork **F**unction **V**irtualisation **I**nfrastructure
NFVI-PoP	**NFVI P**oint **o**f **P**resence
NFVO	**N**etwork **F**unctions **V**irtualisation **O**rchestrator
NGN	**N**ext **G**eneration **N**etwork
NGNI	**N**ext **G**eneration **N**etwork **I**nfrastructures
NS	**N**etwork **S**ervice
NSD	**N**etwork **S**ervice **D**escriptor
NSR	**N**etwork **S**ervice **R**ecord
OSS	**O**perations **S**upport **S**ystem
PNF	**P**hysical **N**etwork **F**unction
PNFD	**PNF D**escriptor
PNFR	**PNF R**ecord
PoP	**P**oint **o**f **P**resence
QoE	**Q**uality **o**f **E**xperience
QoS	**Q**uality **o**f **S**ervice
SA	**S**ervice **A**vailability
SDN	**S**oftware-**D**efined **N**etworking
SLA	**S**ervice **L**evel **A**greement
TTM	**T**ime **T**o **M**arket
VIM	**V**irtualised **I**nfrastructure **M**anager
VLD	**V**irtual **L**ink **D**escriptor
VLR	**V**irtual **L**ink **R**ecord
VM	**V**irtual **M**achine
VNF	**V**irtualised **N**etwork **F**unction
VNFC	**V**irtual **N**etwork **F**unction **C**omponent
VNFD	**VNF D**escriptor
VNFFG	**VNF F**orwarding **G**raph
VNFFGD	**VNFFG D**escriptor

VNFFGR	**VNFFG R**ecord
VNFM	**VNF M**anager
VNFR	**VNF R**ecord

Abstract

Network Function Virtualisation (NFV) is an innovative approach to network architecture design that is quickly transforming the landscape of network provider infrastructures. Aimed at simplifying how network resources are deployed and controlled, NFV allows network services to be decoupled from the physical devices and appliances they run on, through their complete virtualisation.
The traditional concrete building blocks of telecommunication networks (called Physical Network Functions, or PNFs) are transformed into logical Virtual Network Functions (VNFs), capable to represent the functionalities and services provided by the infrastructure as easily deployable virtual elements. Combined with Software Defined Networks (SDN), NFV is the driving force behind the migration of network operator infrastructures towards standard de-facto cloud-based distributed systems; VNFs allow Network Functions to be easily relocated to data centres closer to the current users of the services they host, without the encumbering of the staff and appliances costs involved with physical devices.

The European Telecommunications Standards Institute (ETSI) has in recent years supported the efforts of the industry in migrating towards Network Function Virtualisation based infrastructures through the development of a comprehensive set of standards. The ETSI NFV specification

provides guidelines and architectures for supporting vendor independent Management and Orchestration (MANO) of virtualized appliances, on top of distributed infrastructures provided by Virtualized Infrastructure Managers (VIM) solutions.
OpenStack, an open source project providing a cloud management tool suitable for on-demand deployment of compute, storage and networking resources, has been used as the de-facto VIM by almost every current MANO implementation, leveraging hypervisor technologies such as Xen or KVM to host VNF components on top of virtual machine instances. In recent years, however, the rise of lightweight containers technologies provided by solutions such as Docker have started to shake the datacentre landscape, with their ease of deployment, inferior costs and shorter development times.

This thesis has considered how an existing NFV framework like Open Baton can be extended to fully leverage the capabilities offered by these new containerisation systems. The task has involved the design of the components and logical concepts necessary to correctly integrate these two worlds together, as a way to offer a cloud ready, highly scalable NFV Infrastructure (NFVI).
The realised components have been designed to provide a high degree of independence from the underlying support systems, and provide a generic solution potentially capable to satisfy the demands of the largest possible range of cases. The Docker VIM prototype and its related MANO components developed during this thesis have been designed to be as independent from each other as possible, opening the system to further extensions and high levels of reusability.

The analysis carried out on the Docker-based NFV container orchestration solution created during the implementation step of the thesis has yielded

very positive results regarding the overhead imposed on memory and storage resources by the deployed container-based VNF instances.

Chapter 1 will briefly introduce the reader to the goals of the thesis, explaining the context involved with them. A more thorough explanation of it will given by Chapter 2, which will illustrate the state of the art regarding NFV and software containers.
Chapters 3 and 4 will fully delineate the solution identified by this document, respectively describing its design and architectural structure and providing a deep insight in how the prototype has been implemented.
Finally, Chapter 5 will show how the correctness and performances of the solution have been validated, to ensure that the goals predetermined have been reached.

Introduzione (ITA)

La Network Function Virtualisation (NFV) è un innovativo approccio al design di architetture di reti che sta rapidamente trasformando il mondo delle infrastrutture degli operatori di reti di telecomunicazioni. Progettato per semplificare le modalità in cui le risorse sono fornite e controllate, NFV permette un completo disaccoppiamento dei servizi offerti dalla rete dai dispositivi fisici e gli apparati su cui essi risiedono attraverso la loro completa virtualizzazione.
I tradizionali blocchi realizzativi delle reti di telecomunicazione (chiamati Physical Network Function, o PNF) sono trasformati in Virtual Network Functions, blocchi logici in grado di rappresentare le funzionalità e i servizi forniti dall'infrastruttura come elementi virtuali, semplici da configurare e da lanciare in caso di necessità.
Combinata con le Software Defined Networks (SDN), NFV è la principale forza dietro la migrazione delle infrastrutture dei provider di reti verso sistemi distribuiti basati, nella quasi totalità dei casi, su metodologie cloud; l'uso di VNF permette alle Network Functions di essere agevolmente rilocate in data centers prossimi agli utenti finali dei servizi che offrono, evitando i pesanti costi in personale ed apparecchiature coinvolti nel caso dei dispositivi fisici.

L'Istituto Europeo per gli Standard di Telecomunicazione (ETSI) ha negli

ultimi anni supportato l'impegno dell'industria nel migrare verso soluzioni basate su NFV attraverso lo sviluppo di un comprensivo insieme di standard. La specifica ETSI NFV fornisce linee guida ed architetture volte al supportare l'amministrazione ed orchestrazione (MANO) di apparati virtualizzati, sfruttando le infrastrutture fornite da Virtual Infrastructure Managers (VIM).

OpenStack, un progetto open source per il cloud management volto al deployment di risorse di calcolo, storage e di rete, è stato utilizzato come l'implementazione de-facto di un VIM da pressocché ogni soluzione MANO corrente, sfruttando hypervisors come Xen o KVM come piattaforme di hosting per componenti di VNF tramite macchine virtuali. Negli ultimi anni, la crescita delle tecnologie legate ad applicazioni contenute in containers fornite da soluzioni come Docker ha iniziato a cambiare pesantemente l'ambiente datacenter, grazie alla loro facilità di deployment, costi sensibilmente inferiori e minori tempi di sviluppo.

Questa tesi ha affrontato le modalità con cui un framework NFV esistente, come Open Baton, possa essere esteso per sfruttare appieno le capacità fornite da questi nuovi sistemi di containerizzazione. Il lavoro ha coinvolto il design dei componenti e concetti necessari per correttamente integrare i due mondi assieme, con il fine di offrire una infrastruttura NFV (NFVI) altamente scalabile e cloud-ready.
I componenti realizzati sono stati progettati per fornire un alto livello di indipendenza dai sistemi sottostanti, dando luogo ad una soluzione potenzialmente in grado di soddisfare il maggior numero possibile di casi. Il prototipo di VIM basato su Docker e i relativi componenti MANO sviluppati durante questa tesi sono stati pensati per essere il più possibile indipendenti fra loro, per mantenere il sistema riusabile ed aperto ad estensioni future.

L'analisi compiuta sulla soluzione per l'orchestrazione di container NFV basata su Docker creata durante lo step implementativo della tesi ha mostrato risultati molto positivi riguardo l'overhead sull'utilizzo di risorse di memoria e di storage da parte delle istanze di VNF basate su container.

Il Capitolo 1 introdurrà brevemente il lettore agli obiettivi della tesi, spiegandone il contesto. Una più esauriente spiegazione di questo sarà data nel Capitolo 2, che verterà sullo stato dell'arte relativo all'NFV e i container software. I Capitoli 3 e 4 delineeranno la soluzione identificata da questo documento, descrivendone rispettivamente il design e struttura architetturale e fornendo una esaustiva visione di come il prototipo sia stato implementato. Infine, il Capitolo 5 mostrerà come la correttezza e le performances della soluzione siano state validate per assicurare il raggiungimento degli obiettivi prefissati.

Chapter 1

Problem Statement

1.1 About Network Function Virtualization

Network Function Virtualization **(NFV)** is a modern, innovative approach to the design of network architectures that aims to completely decouple network resources and services from the physical devices and appliances they run on, through their virtualization into **Virtual Network Functions (VNF)**, logical blocks that abstractly represent the several services and components provided by the infrastructure (**Network Functions** or **NF**).

NFV poses itself as a solution to the shortcomings of the current network solutions when they are asked to face the ever increasing complexities and demands of modern telecommunications, replacing the physical, dedicated devices upon which telecom networks have for many years been run. These proprietary solutions usually implement Network Functions on top of physical appliances, which leads to high maintenance, upgrade and personnel training costs plus a general lack of scalability caused by the inadequacy of this model to dynamically match the current demands of the network.

The inability of physical devices to scale properly is one of the most important motivations behind the rise of NFV; virtualizing the network functions allows them to be allocated on the fly on general purpose hardware , where, when and in the right amount that they are needed.

1.1.1 Next Generation Networks

A **Next Generation Network (NGN)** is a networking concept defined by the Internetional Telecommunication Union as

> a packet-based network able to provide services including Telecommunication Services and able to make use of multiple broadband, QoS-enabled transport technologies and in which service-related functions are independent from underlying transport-related technologies. It offers unrestricted access by users to different service providers. It supports generalized mobility which will allow consistent and ubiquitous provision of services to users. [1]

NGN represent a major shift in the way core and access networks work and are designed. The transition to the exclusive usage of **IP packets** on the previously telephone-centric telecom infrastructures allows them to fully decouple themselves from applications and services; the transport layer becomes a service agnostic communication channel, upon which network functions can easily be enabled by defining them at its endpoints.

Network Function Virtualization and NGN are therefore closely intertwined in defining modern networks, moving their core infrastructures to cloud-based solutions and standard servers communicating through Internet technologies. The figure below shows the ETSI [2] approach to network virtualization.

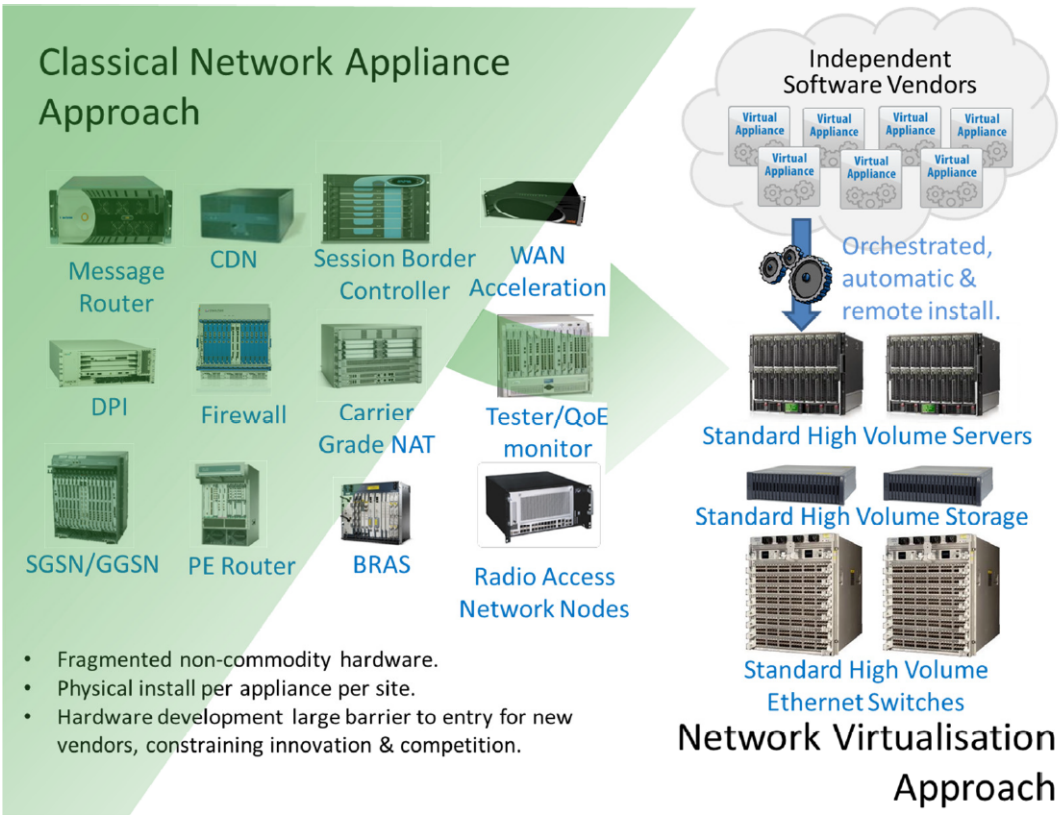

FIGURE 1.1: A comparison of NFV and classical NF

1.1.2 Overview of the advantages of NFV

As said before, **Virtual Network Functions (VNF)** have many relevant advantages over **Physical Network Functions (PNF)**:

- VNFs have a much shorter **Time to Market (TTM)** than PNFs, thanks to their abstract nature that reflects into them not needing any physical modification of the current set of network resources. A VNF can be created on demand on the existing infrastructure, without the necessity of knowing and implementing any physical implementation; this leads to a much shorter design process.

- VNFs have an inherently more scalable nature: a service can be scaled in on the fly to match lower demands (thus cutting power costs), or it

can be scaled out when more clients are available and the number of requests becomes higher.

- NFV is better suited to face geographically different demands. VNFs can be moved into datacenters closer to the current users of the service, and afterwards moved again in a completely different location. This was not really feasible before because of the high costs of relocating staff and appliances.

- The network topology between the services is an abstraction, and can therefore be easily changed on demand.

- The same physical infrastucture can offer different connectivity options to multiple concurrent tenants; the hardware is independent from the services hosted on it and it can be shared or leased.

- VNFs' virtualized nature makes them more resistant to faults. Extra VNFs can be allocated and put into standby, ready to be enabled in case a fault happens in an existing resource; this makes service outage avoidance much more simpler to achieve.

- The optimization of available resources achieved through the reactive scaling approaches enabled by NFV allows datacenters to be used more efficiently, cutting costs further down.

- **Network Orchestration** functionalities allow the network to handle all of the aforementioned scaling operations automatically, without the direct intervention of the multiple administrators of the various physical network segments; the instances of an application can be scaled in and out following the precise demand for resources in a centralized fashion.[2] These functionalities are provided by **NFV Management and Orchestration (NFV-MANO)** frameworks, one of the key components of the NFV infrastructure.

1.2 Architecture of NFV

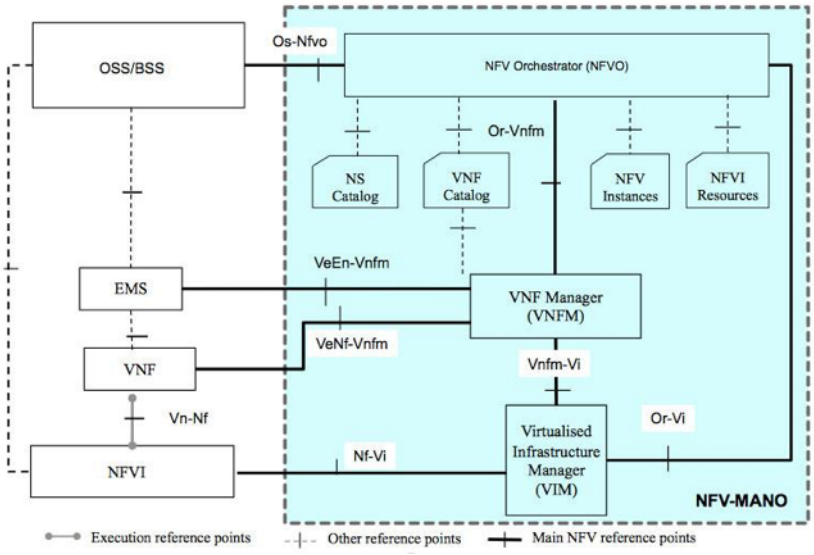

FIGURE 1.2: Architecture of an NFV [3]

The NFV architecture comprises several interconnected components:

- **Virtual Network Functions (VNF)** are entities responsible of the task of handling specific Network Functions. It generally runs on **virtual machines** on top of the physical network infrastructure; multiple VNFs can be connected together to build full scale **Network Services (NSs)**.

- The **NFV Infrastructure (NFVI)** consists in the various hardware and software components upon which the virtual network runs; this also includes the **NFVI Points of Presence (NFVI-PoP)**, i.e. the physical nodes upon which the VNFs are deployed. The NFVI works together with the VNFs and the **Virtual Infrastructure Managers (VIMs)** to interconnect and support the resources contained in the NFVI-PoPs, and provides the abstraction of the underlining network infrastructure that enables the VNFs to perform their functions without the geographic limitations of traditional network architectures.

- **NFV Management and Orchestration (NFV MANO)**, a term that encompasses the various components that handle and manage the VNFs and the NFVI, providing the orchestration and scaling capabilities of NFV. The next subsection will further expand on MANO, with particular emphasis on the **ETSI MANO** [4] specification.

1.2.1 NFV Orchestration

NFV Management and Orchestration (NFV-MANO) is a standard [4], open framework defined by the **ETSI** [5] **Network Functions Virtualisation (NFV) Industry-Specification Group (ISG)** to define open tecnologies and paradigms to orchestrate **Network Function Virtualization Infrastructures (NFVIs)** and **Virtual Network Functions (VNFs)**.
This includes a set of standard components to manage the whole stack of resources in the datacenter and in the cloud, and to allow quick setupping of network services while avoiding the high costs in terms of money and time that traditional methods had. [6]

The MANO architecture is composed of several different interconnected components, handling the various tasks involved with the management and orchestration of the VNFs and the resources provided by the NFVI. These generally reside or are associated with an **NFVI Point-of-Presence (NFV-PoP)**, a local infrastructure element that hosts network functions as VNFs and represents a concrete element of the network infrastructure, like a datacenter or a physical server, and can grouped into three main categories:

- **Compute resources**, like physical servers, virtual servers or in general devices capable of doing computations and execute tasks;

- **Network resources**, like network bandwidth, networks, subnetworks, links and IP addresses, that generally require to be organized, configured and (for limited resources like IP) distributed to the various VNFs grant them connectivity and thus the capability to provide their service;
- **Storage resources**, like databases and volumes, either virtual or physical, to provide persistence and accessibility of data to the network components.

One of the most pressing tasks of the orchestration framework is thus to dynamically match the demands of running and newly instantiated VNFs with the resources offered by the various PoPs present in the current infrastructure. This involves a constant monitoring of the VNF through the various states of its lifecycle, to ensure it has its requirements satisfied an at the same time enforcing any eventual desired policies.
This also means that there should always be the possibility for the orchestration process to stop, resume and spawn new instances of the VNF to face changing requirements, new constraints and requests, using VNF templates that can be deployed to any PoP that may be available at a given time.

1.2.2 Network Service Orchestration

Network Services are usually based on a combination of multiple interdependent VNFs, and thus have more complex lifecycles, and more requirements.
An NS:

- needs to be describable by a descriptor **(NSD)** that defines the NS, its VNFs and their runtime requirements and dependencies;
- should have and NSD that can be stored in a catalogue of services, ready to be instantiated when requested;

- should scale and be scaled to reflect its necessities or to reduce its capacity;
- should be updatable and modifiable if any change in the network configuration or in its requirements makes necessary to do so;

Also, the termination of a Network Service should free all of its dependencies and VNFs, releasing any resources it may have held back to the infrastructure.

The management of the functions required to handle those tasks is called **Network Service Orchestration**, and the architectural component tasked with it is the **NFV Orchestrator (NFVO)**. Chapter 2 will expand more on how ETSI MANO works, analyzing the standard it defines with more detail.

1.3 Software Containers

In recent times, the advent of software containerization has dramatically changed the software industry and developers' approach to how software is deployed and managed. The isolation and versatility provided by containerization solutions like Docker [7] make them obvious choices when choosing which platforms are more suited to host and be deployment targets for Network Services and Virtual Network Functions.

The rest of the chapter will focus on how containers work and how they can fit into the NFV model.

1.3.1 Introduction to containers

A **Software Container** (also known as a *jail*) is an isolated *user space instance* of an operating system. Using **Operative System level virtualization**, a single machine can spawn and run a large number of containers, each one having his own isolated file system and set of installed applications; every single instance behaves as a separate virtual machine, trading the isolation and security of the latter with immediate startup times and almost zero performance overhead.

Some of the most relevant container implementations are *FreeBSD Jails*, *Linux Containers (LXC)*, *OpenVZ*, *rkt* and **Docker**.

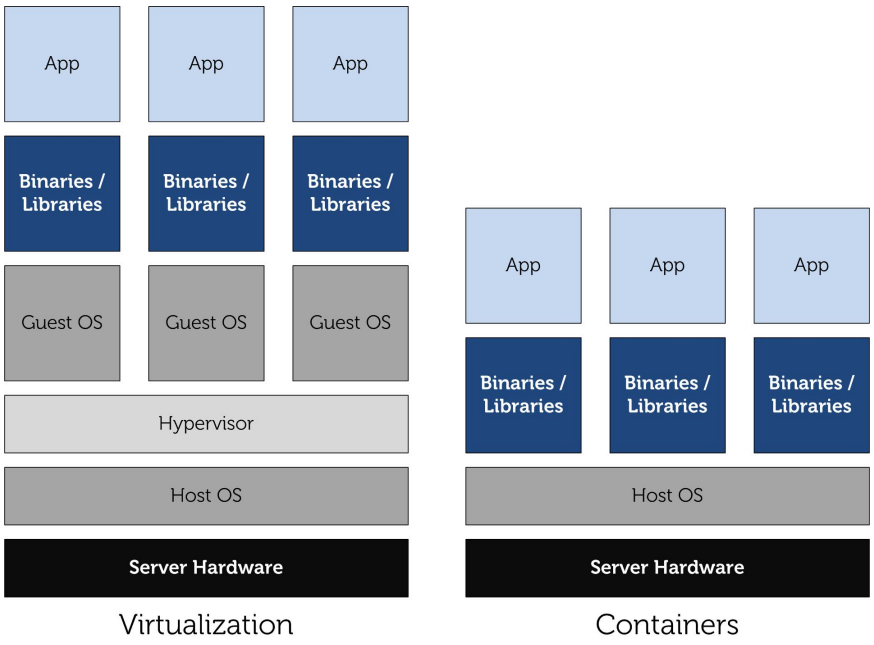

FIGURE 1.3: Architectural differences between containerization and virtualization. Notice the lower amount of layers of the latter.

Thanks to their near bare metal performances combined with application sandboxing capabilities, containers are quickly becoming the de facto standard for running and deploying complex applications.

1.4 Goals

The aforementioned characteristics of containers make them a very appropriate choice for NFV: a VNF can be represented by a container image, by nature easily deployable and scalable through its ability to be spawned several times in a really short time, with lower hardware requirements and complexity than those of a Virtual Machine.

This thesis will address the challenges faced while designing and implementing an NFV solution on top the **OpenBaton** [8] MANO infrastructure, capable of using containers as an a physical backend for the deployment of Network Services. This includes the creation and development of the mappings needed to contextualize the lifecycle and paradigms of a container with regards to Virtual Network Function.
A secondary (but not less important) goal of this thesis is to also focus on future reusability and extensibility of the created solution, enabling the further development of solutions based on heterogeneous and differentiated container technologies. This will involve the definition of several communication protocols, to completely decouple the infrastructure from any constraint caused by its implementation details.

Several components will be described and implemented to realise a complete MANO compliant system, capable to provide the aforementioned characteristics following simple, well documented behaviours.

Chapter 2

State of the Art

The previous chapter briefly introduced Network Function Virtualization, with its goal to further advance the reliability and scalability of current telecommunication systems, and containerisation technologies. The first sections of this chapter will focus on how the ETSI MANO standard [4] defines Management and Orchestration facilities for Network Function Virtualisation, and how it achieves its goal to define a widely shared, free and vendor-independent standard upon which build NFV based networks.
The final sections of this chapter will instead be a showcase of how Operating-system-level virtualisation works, and how using containers based on it differs from the usage of Virtual Machines under hypervisors.

2.1 Introduction to the NFV MANO Standard

The ETSI NFV MANO standard [4] describes a framework to provision, manage and orchestrate Virtual Network Functions, including definitions of the necessary operations to manage their functioning, their lifecycle, their configuration and the network infrastructure they run on. The main objective of the standard is to address the need of a shared framework to define an NFV

architecture, complete with well defined interfaces and concepts, capable of interworking with existing management systems and infrastructures.

2.1.1 Basic principles

Although the NFV-MANO standard does not mandate any specific realization of its framework, it still recommends the adherence to a set of core principles to ensure that the architecture supports correctly the amalgamation of heterogeneous concepts and domains that NFV is.

- The functional blocks that provide orchestration and management are architecturally equal. There should be no primacy of one over the other while distributing orchestration functionalities, and a general architectural principle of horizontal equality between the NFV-MANO functional blocks should subsist;
- NFV-MANO should abstract and provide the services offered by the infrastructure to the VNF and Network Services it manages. These abstract services should also provide fully embedded, selectable resource policies;
- The NFV-MANO functionality should allow different realisations, such as monolithic instances, distributed systems, extensions of pre-existing cloud infrastructures or separate systems that interfaces with them. The leverage of cloud management techniques should be possible thanks to the knowledge of the availability of the aforementioned abstract services;
- The framework should be fully implementable in software, and should not require any special-purpose hardware solution to run. VNF software should feasibly be decoupled from the hardware that hosts it;

- The framework should allow for complete automation, reacting to events in real time with no human intervention;
- The framework should scale across the NFV Infrastructure, in order to support multiple locations and improve service availability;
- The framework should provide standard interfaces to lend itself to open implementations;
- The abstract modelling of the NFVI resource requirements of a VNF should be fully supported;
- Orchestration and management of VNFs and Network Services should be able to access and use resources from single or multiple NFVI-PoPs.

2.1.2 NFV-MANO Architectural Framework Functional Blocks

NFV-MANO defines several functional blocks, each one with a well-defined set of responsibilities. Each one of those applies management and orchestration operations on well-defined entities, leveraging the services offered by the other functional blocks.

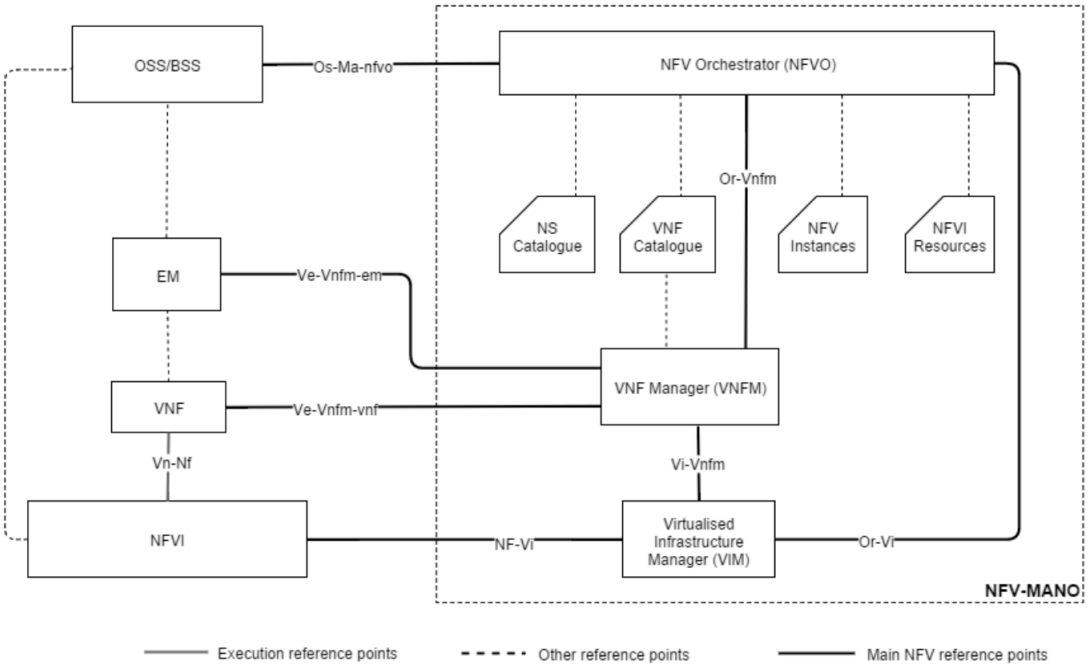

FIGURE 2.1: The NFV-MANO architectural framework with reference points

2.2 NFV-MANO Fundamental Functional Blocks

The NFV-MANO architectural framework consists in three core functional blocks:

- **Virtualised Infrastructure Manager (VIM)**
- **NFV Orchestrator (NFVO)**
- **VNF Manager (VNFM)**

The standard also defines several repositories to store descriptors and informations about current VNF and NS instances; see subsection 2.2.4 for further details.

2.2.1 NFV Orchestrator (NFVO)

The **NFV Orchestrator (NFVO)** is the component of the architecture responsible of the orchestration of NFVI resources across multiple VIMs, and of the lifecycle management of Network Services.

These two responsibilities of the NFVO fulfil two main Management and Orchestration aspects:

- The **Resource Orchestration** aspect is satisfied through functions that handle the release and allocation of the resources of an NFVI, like computation, storage and network resources.
- The **Network Service Orchestration** aspect is satisfied through the provision of functions to handle the on-boarding, instantiation, scaling, updating and termination of Network Services and any operation on their associated VNF Forwarding Graphs.
  The NFVO uses the Network Service Orchestration functions to coordinate groups of VNF instances together to provide Network Services that realise more complex functions; it manages their joint instantiation and configuration, the required connections between different VNFs, and dynamically changes their configurations as required during their operation (e.g. for scaling the capacity of the Network Service in case of high demand).

Resource Orchestration

The NFVO uses its Resource Orchestration functionality to abstract access to the resources provided by an NFVI to services, avoiding them from depending to any VIM.
Some of the features provided by this aspect are the following:

- Validation and authorization of NFVI resource requests from the VNF Managers, to control how the allocation of the requested resources interacts within one NFVI-PoP or across multiple NFVI-PoPs;
- NFVI resource management, including the distribution, reservation and allocation of NFVI resources to NS and VNF instances; these are either retrieved from a repository of already known NFVI resources, or queried from a VIMs as needed. The NFVO also resolves the location of VIMs, providing it to the VNFMs if required;
- Management of the relationship between a VNF instance and the NFVI resources allocated to it, using NFVI Resources repositories and information received from the VIMs;
- Policy management and enforcement, implementing policies on NFVI resources. This may involve access control, reservation and/or allocation of resources, optimization of their placement based on affinity, geographical or regulatory rules, limits on resource usage, etc.;
- Collection of informations regarding the usage by single or multiple VNF instances of NFVI resources.

Network Service Orchestration

The Network Service Orchestration function of the NFVO uses the services exposed by the VNF Manager function and by the Resource Orchestration function to provide several capabilities, often exposed by means of interfaces consumed by other NFV-MANO functional blocks or other external entities:

- Management of Network Services deployment templates and VNF Packages, including the on-boarding of new Network Services and VNF Packages; the NFVO verifies the integrity, authenticity and consistency of deployment templates, and stores the software images provided in

VNF Packages in one or more of the available NFVI-PoPs, using the support of a VIM;

- Network Service instantiation and Network Service instance lifecycle management, through operations like updating, querying, scaling and terminating a Network Service. This also includes collecting performance measurement results and recording events;
- Management of the instantiation of VNF Managers;
- Management of the instantiation of VNFs, in coordination with VNF Managers;
- Validation and authorization of any NFVI resource request that may come from a VNF Manager, to control its impact on the current Network Services;
- Management of the VNF Forwarding Graphs that define the topology of a Network Service instance;
- Automated management of Network Service instances, using triggers to automatically execute operational management actions for NS and VNF instances, following the instructions captured in the on-boarded NS and VNF deployment templates;
- Policy management and evaluation for the Network Service and VNF instances, implementing policies related with affinity/anti-affinity, scaling, fault and performance, geography, regulatory rules, NS topology, etc;
- Management of the integrity and visibility of the Network Service instances through their lifecycle; the NFVO also manages the relationship between the Network Service instances and the VNF instances.

2.2.2 Virtualised Infrastructure Manager (VIM)

The **Virtualised Infrastructure Manager (VIM)** is the NFV-MANO function responsible for controlling and managing the resources contained in a NFV Infrastructure, including the compute, storage and network capabilites provided throughout its NFVI-Points-of-Presence.
A VIM may specialize in handling a certain type of NFVI resource (e.g. it may be compute-only, storage-only or networking-only), or it may be capable of managing multiple types of NFVI resources at once, exposing a northbound interface to other functions to manage them.

VIMs interface with a variety of hypervisors and Network Controllers, in order to perform the functionalities exposed through their northbound interfaces; these usually consist in:

- Orchestrating the allocation, upgrade, optimisation, release and reclamation of NFVI resources, managing their association with corresponding compute, storage and network physical resources. To achieve this, the VIM keeps an inventory of how virtual resources have been associated to physical resources, like a server pool or a table of IP addresses;

- Supporting the creation, querying, updating deletion of VNF Forwarding Graphs, by creating and maintaining Virtual Links, Networks, subnets and ports and managing the security group policies that provide network and traffic access control [9];

- Keeping and managing a repository containing the informations related to the NFVI hardware and software resources, and the methods for discovering their capabilities and features;

- Management of the capacity of virtualised resources, including the forwarding of information related to NFVI resources capacity and their usage;
- Management of software images, allowing other NFV-MANO functional blocks like the NFVO to add, delete, update, query or copy an image from repositories of software images maintained by the VIM itself. Each image must be validated before being stored;
- Collection of performance and fault information from concrete (either hardware or software) or virtualized resources, like hypervisors and virtual machines and their forwarding to other functional blocks;
- Management of catalogues of virtualised resources, to be consumed from the NFVI, potentially in the form of virtualised resource configurations and/or templates.

OpenStack

Openbaton, and NFV-MANO solutions in general, usually realise their default VIM implementations on **OpenStack**, a fully-open, vendor independent platform for management and deployment of processing, storage and network resources.
OpenStack provides all of the functionalities required to control and manage an NFVI by handling the (in the most common scenario) Virtual Machines that host and run the VNFs, including the creation and management of IP addresses and the networks between them.
This dependence on Virtual Machines is undesirable, and it is one of the main driving points behind the work this thesis is currently describing; although OpenStack provides facilities to use and handle containers itself, it suffers deeply from being a very large system that requires skilled professionals to be set up and maintained. A lighter and more streamlined solution is highly

desirable for the purpose of deployment of VNF on top of container technologies.

2.2.3 VNF Manager (VNFM)

The VNF Manager is a NFV-MANO function that satisfies the Management and Orchestration aspects of VNFs through the lifecycle management of their instances.
A single VNF instance is uniquely associated to a given VNF Manager; this manager may handle several other instances, of the same or different types. While a VNFM must support the requirement of the VNFs associated to it, most of the VNF Manager functions are generic, and do not depend from any particular type of VNF.

Like the other functions, the VNFM exposes functionalities to other elements of the NFV-MANO architecture, often as interfaces.
This functionalities include:

- VNF instantiation and (if needed) VNF configuration, using a VNF deployment template;
- Checking if the instantiation of VNF instantiation is feasible;
- Update the software contained in a VNF instance;
- Modify a running VNF instance;
- Handle the scaling out/in and up/down of instances;
- Collect NFVI performance measurement results, faults and events correlated with its VNF instances;
- Provide assisted or automated healing of VNF instances;

- Handful the termination of VNF instances;
- Handle notifications caused by changes in the VNF lifecycle;
- Management and verification of the integrity of a VNF instance through its lifecycle;
- Coordinate and handle configuration and event reporting between the VIM and the EM.

Each VNF is defined in a template called **Virtualised Network Function Descriptor (VNFD)**, stored in a **VNF catalogue**, that corresponds to a **VNF Package**. A VNFD defines the operational behaviour of a VNF, and specifies how it should be deployed providing a full description of its attributes and requirements. NFV-MANO uses VNFDs to create instances of VNFs, to manage their lifecycle, and to associate to a VNF instance the NFVI resources it requires; to ensure full portability of VNF instances from different vendor and different NFVI environments, the requirements must be expressed in terms of abstracted hardware resources.

The VNFM has access to a repository of available VNF Packages; each package may be present in several versions, all represented using a VNFDs, to allow for different implementations of the same function on different execution environments (like different hypervisor technologies and implementations).

2.2.4 Data Repositories

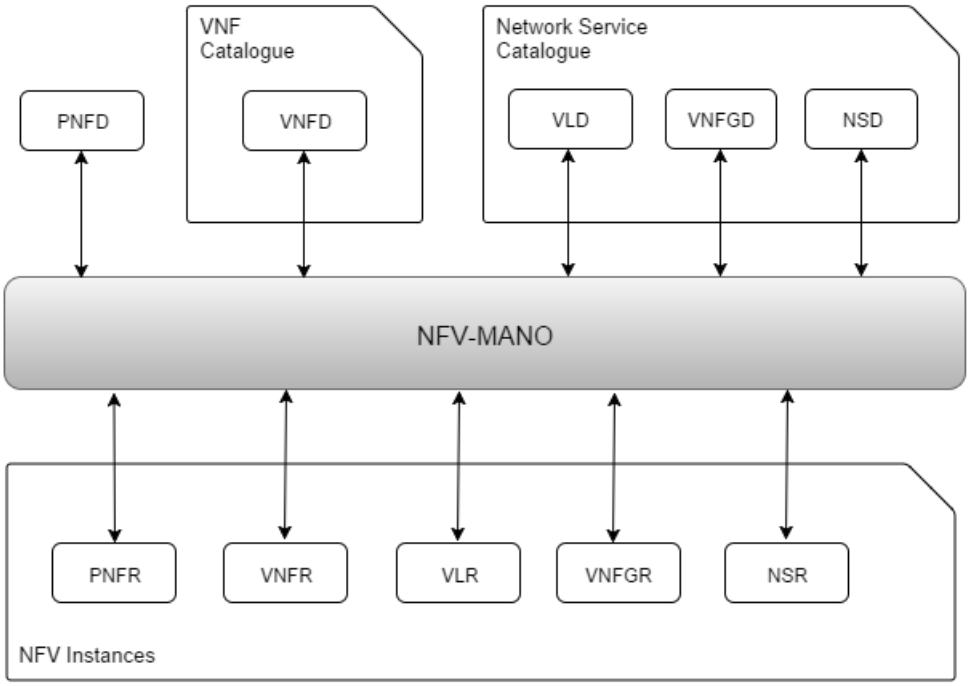

FIGURE 2.2: Relationship between NFV-MANO, NFV instances and their descriptors

The data associated with the orchestration and management operations is kept in several repositories:

- The **NS Catalogue**, that contains all of the on-boarded Network Services. The NS Catalogue is designed to support the creation and management of NS deployment templates, like **Network Service Descriptors (NSDs)**, **Virtual Link Descriptors (VLDs)** and **VNF Forwarding Graph Descriptor (VNFFGD)**.
  The NS Catalogue is kept by the NFVO, and it is made available through interface operations;

- The **VNF Catalogue** is a repository of all of the on-boarded **VNF Packages**, and supports their creation and management (including VNF Descriptors (VNFD)) via interface operations exposed by the NFVO. Both the

NFVO and VNFMs can query the VNF Catalogue to search, retrieve and validate VNFDs;

- The **NFV Instances repository** holds information regarding all the VNF and NS instances, represented by their respective **VNF Records (VNFs)** and **NS Records (NSRs)**. Those records are updated during the lifecycle of the respective instances, reflecting changes resulting from execution of lifecycle management operations.

  The NFV Instance repository supports the NFVO's and VNFM's responsibilities in maintaining the integrity and visibility of the instances, and the relationship between them Network Services and VNFs;

- The **NFVI Resources repository** holds information about available, reserved and allocated NFVI resources, as abstracted by the VIMs. This data is useful for resources reservation, allocation and monitoring purposes; as such, the NFVI Resources repository plays an important role in supporting the NFVO's Resource Orchestration and governance roles, by allowing the tracking of NFVI reserved and allocated resources against their associated NS and VNF instances.

2.2.5 Typical VNF instantiation flow

Figure 2.3 describes in detail how the various components of the NFV-MANO architecture typically interact with each after a VNF instantiation request:

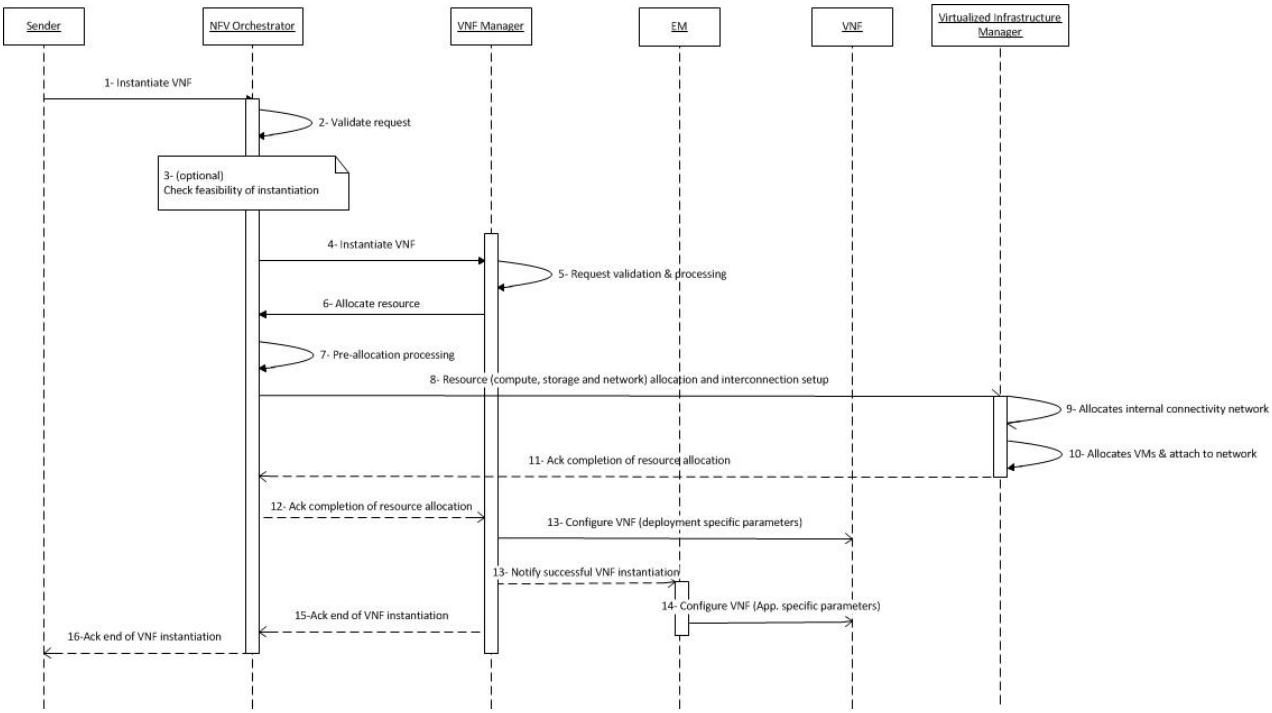

FIGURE 2.3: VNF instantiation message flow, as described by the ETSI NFV-MANO standard [4]

The main steps for the instantiation of a VNF can be summarized as follows:

1. The NFVO receives an **Instantiate VNF**, either from the OSS or a VNFM, along with instantiation data required to deploy the VNF;

2. The NFVO validates the VNF instantiation request, optionally running a feasibility check to reserve the necessary resources before executing it;

3. The NFVO sends to the VNF Manager a **Instantiate VNF** request, to instantiate the VNF with its instantiation data and (if it has been done

before) the reservation informations of the resources previously allocated;

4. The VNFM validates the request and processes it, complementing the instantiation data with data contained in the VNFD data or other sources;

5. The Manager then calls the NFVO, to ask for resource allocation through an **Allocate Resource** request;

6. The NFVO prepares for allocation of the requested resources, doing any necessary pre-allocation processing work;

7. The NFVO sends an **Allocate Resource** request to the VIM , to allocate the needed compute, storage and network resources for the various VDUs that compose the VNF instance;

8. The VIM allocates the internal connectivity network;

9. The VIM interfaces with the NFVI and allocates the requested compute (i.e., VMs) and storage resources, and attaches the instantiated VMs to the previously allocated internal connectivity network;

10. The VIM sends a completion acknowledgement response back to the NFVO;

11. The NFVO acknowledges back the VNFM about the successfully completion of the allocation process, returning appropriate configuration information;

12. The VNFM configures the VNF, using appropriate configuration operations. If present, the EM is notified of the newly instantiated VNF;

13. The EM (if present) also applies its configuration parameters on the VNF;

14. The VNFM returns a response to the NFVM acknowledging the completion of any configuration operation;

15. The NFVO can finally declare the VNF instantiation operation complete, by acknowledging it to the sender.

After these steps, the VNF is up and running, and it is ready to serve.

2.3 Other Functional Blocks

The following functional blocks are not part of NFV-MANO themselves, but they are tightly involved with the Management and Orchestration process and exchange informations with NFV-MANO functional blocks (NFVO, VNFMs, VIMs and repositories):

- The **Virtualised Network Functions (VNFs)**;

- The **Element Management (EM)**, which responsible of the FCAPS (Fault, Configuration, Accounting, Performance and Security) Management functionality for a VNF.

  The EM is a component that handles the task of configuring the network functions provided by a VNF, providing fault management, usage accounting, performance measurement collection and security management to them. The EM may be aware of virtualisation, and collaborates with the VNFM to perform those functions that require exchanges of information regarding the NFVI Resources associated with the VNF;

- The **Operations Support System/Business Support System (OSS/BSS)** are the combination of the operator's other operations and business support functions. They usually exchange data with the functional blocks in the NFV-MANO architectural framework, and may provide management and orchestration to legacy systems not covered by NFV-MANO;

- The **Network Functions Virtualisation Infrastructure (NFVI)**, which as mentioned before is a term that generalises all the hardware and software components that together provide the infrastructure resources where VNFs are deployed, including any partially virtualised Network Function still present on the network (like hardware switches and load balancers) under the control of OSS/BSS.

2.4 NFV-MANO reference points

Several reference points are defined between NFV-MANO and external functions:

- **Os-Ma-nfvo**, a reference point between OSS/BSS and NFVO that involves NSDs management (and VNF packages), management of Network Services and forwarding of requests between OSS/BSS and VNFMs, plus policy enforcement and event forwarding;
- **Ve-Vnfm-em** and **Ve-Vnfm-vnf**, two reference points between a VNFM and EM or a VNF respectively, used by a VNFM for management and control of VNFs;
- **Nf-Vi**, a reference point used by VIM to control a NFVI, including the management of Virtual Machines and forwarding of events, configurations and usage records;
- **Or-Vnfm**, a reference point between NFVO and VNFM used to authorize the allocation and release of resources to VNFs, to instantiate VNFs and to retrieve and update informations regarding VNF instances;
- **Or-Vi**, a reference point between NFVO and VIM, used by the NFVO to handle NFVI resources, to receive events and reports, and for management of VNF software images;

- **Vi-Vnfm**, a reference point used by VNFMs for NFVI resources information retrieval and allocation from VIMs.

These links allow the NFV-MANO components to receive informations about the systems under their management, using standard defined interfaces.

2.5 Operating-system-level virtualisation

As previously mentioned, **Operating-system-level virtualisation** is a virtualisation technology that allows a single OS kernel to run multiple isolated user space instances at the same time, greatly reducing the overhead associated with traditional virtualisation techniques while preserving a great level of sandboxing and isolation for the deployed system images.
This chapter will introduce in more detail how this technology works, highlighting its similarities and differences with Hardware-level virtualisation techniques.

2.5.1 Introduction to Virtual Machines

The term **Virtual Machine (VM)** generically describes a software implementation of a machine architecture, computing platform or execution environment. Each VM provides the functionality of a full machine on top of existing hardware and platforms, and they are often indistinguishable, as far as the user is concerned, from a real, concrete implementation of it.

Virtual Machines have been used and developed for decades, and they are well suited for many different tasks and applications [10]:

- They allow to reduce costs by allowing to shrink the number of servers needed by an infrastructure, through consolidation of several different machines into virtual ones, running on a single host;

- By their nature, a virtual machine is isolated (at different levels of effectiveness, depending on the underlying technology) from its host and other instances running on it; this helps increasing fault and intrusion tolerance of the system, isolating untrusted code and services from the physical machine;

- A virtual machine exists as a software, data resource, and therefore is independent from the hardware it runs on. This makes migrating it migrated to a different system feasible in case the necessity of doing it arises, improving its resistance to hardware faults;

- Virtual machine make for perfect test beds, allowing to test a software in an environment isolated from the rest of the machine;

- A virtual machine may emulate a wholly different platform than the one it runs on, allowing its user to run software normally not executable on the system.

Virtual Machines can simulate a machine at different levels of abstraction, and several types of virtualisation can be defined depending on how much of the hardware and software stack of the VM is shared with the host. A solution can therefore range from full, complete machine emulation, to rudimentary file system isolation in solutions like UNIX chroot.

2.5.2 Hypervisors

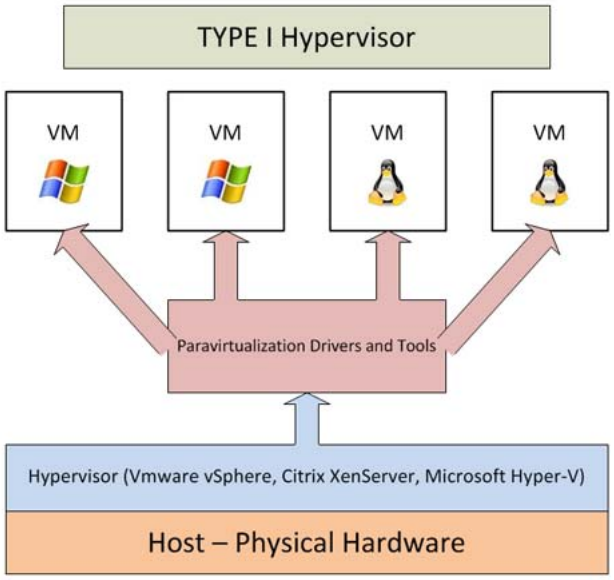

FIGURE 2.4: Architecture of type 1 hypervisors

Type-1 Hypervisors have long been the most popular and widely deployed form of virtualisation in the datacentre space, helping the industry in its shift towards more efficient and less expensive infrastructures capable of better optimisation of the available resources.

These hypervisors use generally **Hardware virtualisation** technologies to virtualise a full machine, including its firmware, networking and storage resources. This type of virtualisation has become feasible in the last decade with the advent of solutions like **Paravirtualisation** and **Hardware-backed Full Virtualisation** [11], that have made its performance acceptable for pretty much any kind of workload, avoiding the high costs involved with CPU emulation technologies like binary translation.

Type 1 Hypervisors run directly on top of the hardware, providing to their guest VMs direct or paravirtualised access to the hardware resources they manage, like virtual networks, cpu allocation and storage devices.

Comparison with Operating-System-level virtualisation

Both Hypervisors and containerisation technologies can be used to provide to end users virtual environment that, for all tasks and purposes, are indistinguishable from real ones. However, they are still based on fundamentally different concepts:

- While containerisation solutions provide isolation of a container from the both host machine and other guests, a VM running under an hypervisor has its own kernel instance, private and not shared. This is especially important if running an operating system different than the host is a requirement, given the fact that operating-system-level virtualisation can only run user space instances of the same type of the host;
- A container has almost zero runtime overhead compared to bare metal. Sharing the same kernel with the host means that there's no need for hardware to be emulated or virtualised in any way; the kernel itself is responsible of the isolation and management of the processes running in the container (using namespacing techniques), and sharing data between the environments is computationally inexpensive;
- A container can be set up and torn down in a very short amount of time, a particularly useful characteristic for highly demanding or performance constrained tasks that also need isolation and fault tolerance.

2.5.3 Containerisation platforms

Several implementations of software containers exist, for pretty much every modern operating system available today [12]:

- The UNIX **chroot** command, which can be considered an unsafe, rudimentary early example of container;

- **FreeBSD Jails**;
- **Solaris Zones**;
- **OpenVZ**, for the Linux kernel;
- **Linux Containers (LXC)**, plus the **LXD** daemon;
- The **libcontainerd** and its OCI fork **runc**, used by the **Docker** project;
- **systemd-nspawn**, a container solution directly integrated into the systemd init daemon;
- **rkt**, a Docker-compatible solution from the CoreOS GNU/Linux distribution.

Application Containers

Operating-system-level technologies have been available and widely used for decades as low cost substitutes for traditional virtual machines. In the last five years, the industry interest around them increased considerably thanks to their usage in **Application Containerisation** solutions, like **Docker** [7] (by far the most popular container solution at the moment) and **rkt**.
The application container concept revolves around the (often automated) creation of pre-packaged software images, stored in a local or remote repository. Each one of these images represents a complete filesystem tree that contains everything is needed to run one or more software applications, such as their files and dependencies. An user is then able to deploy these on demand into multiple containers, greatly reducing the times and costs involved with installing and maintaining the software; each container image defines a fully reproducible environment, guaranteed to be the same each time it is instantiated, and therefore capable to be moved to other systems, to be scaled into multiple instances of itself, or to be reset if the necessity of doing so arises.

All of this can be done with performances akin to bare metal deployments, thanks to the usage of containerisation technologies that allow these environments to be run in process namespaces and virtual filesystems (some mention worthy mechanisms on Linux are **cgroups**, kernel namespaces, and **OverlayFS**, an union mount filesystem).

The images are generally kept in a common format (initially developed by Docker and then standardised under the **Open Container Initiative** [13]) that allows them to be transferred to other systems or incrementally updated using layering solutions.
Docker and rkt also provide registry infrastructures to store, publish and easily retrieve ready made images from official or third party sources; this allows everybody to run, modify and suit them to their needs, greatly simplifying how software is deployed and developed.

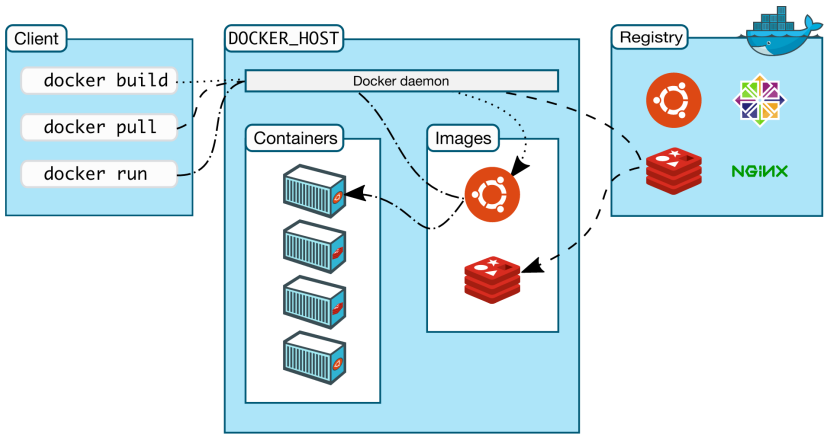

FIGURE 2.5: Scheme of how pulling an application image from the Docker Registry works

2.6 Requirements for NFV Platforms

NFV has very peculiar requirements, and often represents an edge case for many technologies involved with it because of its very singular demands. This section will outline which main challenges may arise while building an NFV platform, focusing on those related with choosing a virtualisation technology.

2.6.1 Performance-related constraints

The shift from Physical Network Functions to Virtual Network Functions in recent years has brought to the middlebox-oriented world of telecommunication networks the many advantages of virtualisation, like server consolidation, higher availability, scalability and increased cost effectiveness thanks to the usage of less expensive off-the-shelf hardware.

Although NFV is build on top of a consolidated stack of cloud technologies, its performance and networking requirements do not match the same demands involved with a standard cloud context [14]. Some of the potential challenges related to deployment performance are [15]:

- The **provisioning time** of a VNF, namely the time taken by the infrastrucure to spin-up a new VNF together with its application-specific and additional dependencies, it is an important performance factor that is more influenced by implementation choices such as the hypervisor, the guest and host OS used and their needs than the VNF startup process itself[15]. This is not helped by the potential large size of VM images, that may sum up to several GiB and thus require longer deployment and migration times [16];

- The **runtime performance** of a VNF directly depends on how the amount of resources (e.g., virtual CPUs, RAM) allocated to the VM influences factors like the achievable throughput, the line rate speed, the maximum concurrent sessions that can be maintained and the number of new sessions that can be added per second.

  Picking the right values is a non trivial task; under-provisioning of the available resources is undesirable because it may lead to an increased number of scaling operations (and thus additional latency caused by VM allocation), while over-provisioning them often leads to an under-utilization the physical resources [15].

  The VM technology itself should also be considered while evaluating runtime performance of a VNF; a particular virtualisation platform may indeed introduce higher efficency and computational costs when compared to another similar solution [16];

- The performances of the **virtualised network** infrastructure between the virtual machines is critical to the overall networking performance of the system. Because in the NFV model a VNF has been confined inside of a virtual machine, how well it performs with regards to serving latencies and inter-communication mainly depends from how good the link provided by the virtualisation platform is.

2.6.2 Continuity, Elasticity and Portability challenges

Several factors may temporarily or critically interrupt a VNF during its service; some of these include events like software updates, software crashes, lack of necessary resources and hardware faults. It is therefore necessary to define further requirements, to make a VNF more resilient to interruptions and less dependent from its current location [15].

- Porting a VNF from its current system to another requires a two-step process; the required resources of the current platform are first matched with equivalent ones of the target, and then the VNF is brought back to a working state by repeating its provisioning steps. This is necessary because most VNFs are coupled with their underlying infrastructure, and therefore have a dependency from their guest OS, their hypervisor or some other component;

- The used virtualisation technology needs to provide an efficient high availability solution (or a quick restoration mechanism) that can bring back the VNF to an operational state after a fault, to respect ts service continuity requirements. To restore the VNF to a working state after an anomaly (that may have been caused by a hardware failure, for instance), it is necessary to first provision the VM (or container), spin-up and configure the VNF process inside the VM, setup the interconnections to forward network traffic, manage the VNF-related state and finally update any dependent runtime agents;

- A complete view of the underlying resources is necessary to when addressing the service elasticity challenges.
  To get such an holistic view the system has also to consider a few additional challenges:

  - The system should provide continuous scalable monitoring of the individual resource's current state, to allow it to spin-up additional resources (i.e. to auto-scale or auto-heal) if the system encounters performance degradation or to optimize resource usage by spinning down idle resources;
  - The system should handle CPU-intensive VNFs differently than I/O-intensive VNFs: degration in the former primarily depends on the VNF's processing functionality, while, on the other hand, an I/O intense workloads can have a significant overhead depending of the hypervisor, its host's features, its type, the number of VMs (or containers) it manages, etc.

2.6.3 Security considerations

Speaking of the security of a VNF actually consists of speaking of either the security features provided by the VNF itself to manage its state, or the security of the VNFs and its resources. To satisfy both cases, it is necessary for the solution to provide secure slicing of the infrastructure resources, ensuring the following requirements [15]:

- Provisioning of the network functions, guaranteeing complete isolation across resource entities like hardware units, hypervisors and virtual networks; this also includes providing secure communication and access between the VMs and between host and guest. Sharing of resources across network functions must be possible, to maximise resource utilization and service elasticity;
- Ensuring service continuity for the unaffected resources, when a resource component is put in quarantine after being compromised;

- Provide mechanisms to securely recover and restore the network functions to an operational state after runtime vulnerabilities or attacks. It is important for this requirement to be achievable with minimal or no downtime;

- Avoid resource starvation, to preserve availability: applications hosted in VMs or containers can starve the underlying physical resources, making their co-hosted entities unavailable. The ideal response to this is to provide countermeasures to monitor the usage patterns of individual guests, to enforce fair limits on their usage of individual resources.

Realizing the above requirements is a complex task in any type of virtualisation option, including both virtual machines and containers.

2.6.4 Management issues

The challenges involved with management and operational aspects of NFV are primarily focused on managing the VNF lifecycle management and its related functionalities. The solution must handle the management of variables and events such as failures, resource usage, state processing, smooth rollouts, and security, including those that have been discussed in the previous sections. Some of the features management solutions provides include [15]:

- Centralized control and visibility, supporting web clients, multi-hypervisor management, single sign-on, inventory searches, alerts and notifications;

- Proactive Management, allowing features like the creation of host profiles, resource management of VMs or containers, dynamic resource allocation, auto-restart in an High-Availability model, audit trails, patch management, etc;

- An extensible platform, with the possibility to define roles, permissions and licenses across resources. APIs can also be used to integrate with other solutions;

The above features lead to the definition of the following requirements for a NFV management solution:

- It should be simple to operate and deploy VNFs with it;
- It should use well-defined standard interfaces, to allow seamless integration with different vendor implementations;
- It should provide the possibility to automate the handling of VNF lifecycle requirements;
- It should provide well defined APIs that abstract the complex low-level information of the architecture from external components;
- It must provide security.

Satisfying the aforementioned requirements for a management solution is not simple; hypervisors, guest OS, VNFs functionalities, and the network state add a whole layer of multi-dimensional complexity to the problem, further complicating the challenges faced while designing a feasible and robust system capable to address them. Chapter 3 will address further in detail this challenges, and how these have been tackled while designing the NFV containerisation solution at the core of this thesis.

2.7 Container technologies and NFV

The last two sections of this chapter have briefly introduced operating-system-level virtualisation to the reader and the challenges faced while building a platform for VNF deployment, illustrating how network infrastructures and datacentres can leverage containerisation technologies to design and deploy more scalable and robust systemss.
In light of the advantages offered, it is very compelling to inspect and study how lightweight virtualisation solutions can be adapted while constructing an NFV infrastructure as a concrete alternative to traditional techniques, e.g. hypervisors and virtual machines; this section will discuss in depth this issue, bringing some experimental data to justify the effort.

2.7.1 Potential benefits of containers and NFV

Container ecosystems (like Docker and such) have the potential to address some of the challenges that virtualisation poses when used together with NFV [16], such as the aforementioned performance and efficiency costs [17], the potential deployment slowdown issues caused by very large software images, and networking I/O overhead:

- As previously explained, applications in containers run on the host OS without any hardware indirection; thus, they can run more efficiently than their VM-based counterparts in many cases [18];
- A container does not ship a whole OS image, because it shares the same underlying kernel and system with its host. This leads images taking much less disk space than comparable VMs, allowing an higher application density on a host [19] and significantly decreasing time to deploy and migrate. Research is also being made towards live migration of stateful containers [20];

- The novel packaging system provided by Docker and similar technologies can remove some of the variability in hosting requirements that a VNF may express, by shipping pre-packaged, full configured environments. Projects like the Open Container Initiative (OCI) [13] aim to standardize container formats, and make them even more platform agnostic.

Technology	**Time (ms)**	**Mem (MiB)**	**Size (MiB)**
Xen VM	6500	112	913
KVM VM	2988	32	913
Docker Container	1711	3.8	61

TABLE 2.1: Performance comparisons [15]

As seen in the example presented by table 2.1, containers have much lower startup times and an overall inferior footprint on the system, thanks to their significantly lighter usage of system resources.

Network I/O advantages are less clear: containers are typically used to provide isolation for services that communicate using one or more network sockets bound to a port on the host; thus, it is critical to assess how much network virtualisation impacts them.
The inbound and outbount traffic towards and from the container is handled by the host's network stack, using a software switch (such as the Linux bridge), that may add additional performance costs compared with bare metal. While many services typically deployed in containers are not bounded by network performance, most use cases for NFV have strict requirements for network throughput and delay; also, common techniques to improve network I/O performance such as Generic Receive Offload (GRO) and TCP Segmentation Offload (TSO) may not be appropriate for NFV, since some VNFs

(like L2 bridges, NAT, ...) need to work directly at data link level, inspecting Ethernet frames instead of TCP/UDP segments.

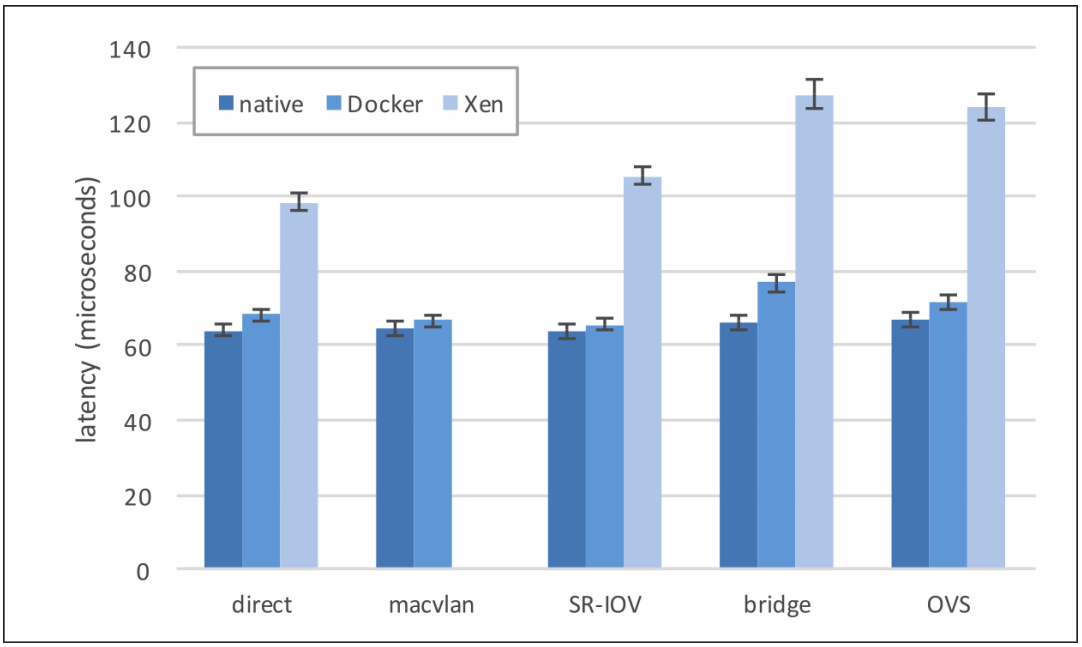

FIGURE 2.6: Latency of different virtualisation technologies [16]

Several studies show that containers have the potential to reduce network latency when compared with a virtual machine running an off-the-shelf server operating system [16] [15] [14], but this is not without a few drawbacks.

2.7.2 Conclusions

In light of the benchmarks above and of further research data, it is possible to summarise how containers and VM satisfy the requirement delineated before [15]:

- A container can spin up much faster than a VM, thus providing more service agility and elasticity;

- A container shares resources with their host, and thus has inherently less memory requirements than an equivalent VM: this is cause by the need of having a separate OS instance for each one of the guest;
- Hypervisors generally offer better security and isolation for their guest, compared with operating-system-level virtualisation; sharing resources with the host also means exposing them to the guest, increasing potential number of issues in case vulnerabilities are discovered. These technologies are also much older, and so they have been tested more thoughtfully for bugs and vulnerabilities; containers are relatively new, and as such still have a number of open issues. Use of kernel security modules like SELinux or AppArmor may help mitigate some of these concerns, offering the required features for a secure VNF deployment. This can also be integrated with resource quota techniques, to provide at least some of the resource guarantees necessary to a VNF deployment;
- Both virtual machines and containers have largely supported, open-source and fully-featured management frameworks, so this is not an issue;
- Both containers and standard VMs can run any application a general-purpose OS can run. Containers are inherently restricted to be of the same type than their host OS, but this is generally non an issue.

These technologies have therefore both strengths and different usage scenarios, depending on which security, isolation, performance and compatibility criteria are required by the NFV operator. For instance, an operator may choose an hypervisor-based solution if isolation and multi-tenancy are mandatory requirements; if instead ease of application deployment is preferred, it may lean towards the benefits offered by containers.

Hybrid solutions, where containers are run within VMs, are also possible; a single VM may host several containers, offering at least full isolation from the host. For instance, Microsoft proposes an alternative solution to normal Windows containers, that allows a single instance to run with its own stripped-down kernel, using the Hyper-V hypervisor [21]. This provides full isolation to the container, while offering better performances than a full fledged virtual machine.

2.8 Open Baton

Open Baton is an open source project, launched by TU Berlin and Fraunhofer FOKUS, which aims to provide a complete NFV orchestration framework based on the ETSI NFV-MANO and TOSCA standards. [22]
Open Baton allows the creation of complete NFV environments, providing the main functional blocks required by the specification:

- A service that implements the **NFV Orchestrator (NFVO)**, capable of carrier-grade orchestration of network functions and infrastructure resources;
- A **Generic VNFM** to dynamically manage the VNFs;
- An **Auto Scaling Engine**, for automatic runtime management of the scaling operations of the Virtual Network Functions;
- A **Network Slicer Engine**, a module that can optinally define slices of network conforming with the requirements expressed by the Network Service Descriptor;
- The **EMS (Element Management System)**, a component loaded inside of a VNFC to provide a communication link with a VNFM. This is used

to allocate specific functions to the VNFCs, and to provide provisioning for external resources;

- A **plugin mechanism** that allows the NFVO to support different types of Virtualized Infrastructure Manager (VIMs) without having to re-write anything in the orchestration logic;
- An **OpenStack VIM Driver plugin**, a plugin that allows the NFVO to interface with OpenStack. The OpenStack plugin exposes the OpenStack resources (like virtual machines, networks and such) to its overlying functional blocks, using standard ETSI interfaces;
- A Juju VNFM, to interoperate with Juju;
- A Zabbix-based **Fault Management System**, which can be used for automatic runtime management of faults in the various levels of the architecture;
- An **event engine**, based on a pub/sub mechanism, used for the dispatching and execution of lifecycle events;
- **User interfaces**, to easily access orchestration features through the CLI and a graphical dashboard.

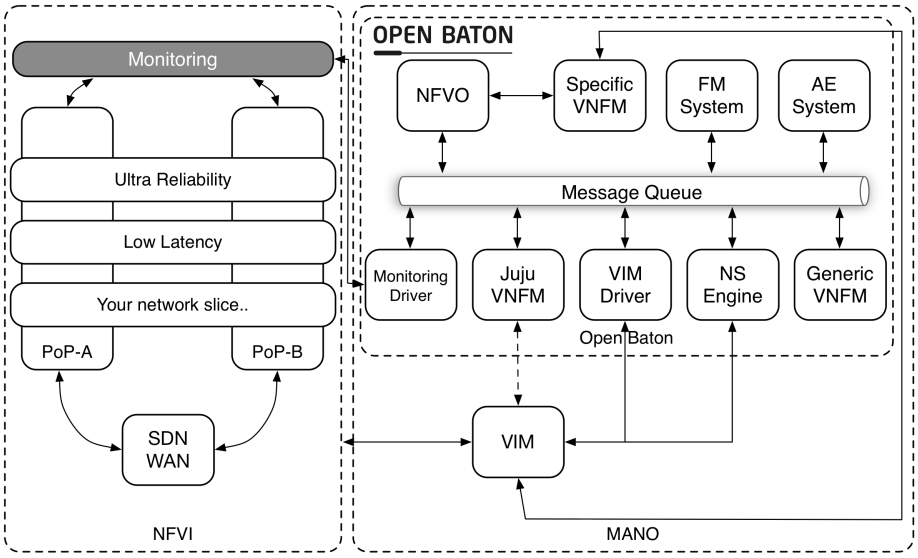

FIGURE 2.7: Architecture of Open Baton

Open Baton strives to enable the deployment of Virtual Network Services on top of multiple cloud-infrastructures, helping its users to achieve higher scalability and flexibility on their network infrastructures; it makes easier to move Network Functions to the cloud, with all of the advantages of NFV and SDN technologies.

One of the most important features that differentiate Open Baton from other orchestration solutions is its extensibility-oriented approach. Instead of being all contained in a single, monolithic block, many additional features are offered as separated, external components: **AMQP** (Advanced Message Queuing Protocol), as implemented by RabbitMQ, is used to provide a standard interface, allowing plugins and services (like the auto-scaling system and the fault management system) to communicate with the NFVO and to be interoperable with external components and Virtual Network Function Managers (VNFMs).

This extensible architecture, combined with SDKs and libraries to develop new components in several languages, allows for fast prototyping of new

advanced features, without requiring any modification to the orchestrator or the system.

This thesis will use and extend Open Baton to show how a container-aware solution can be designed and used to fulfil the main targets and goals of scalability and efficiency required by the NFV world.

2.8.1 Brief overview of other NFV MANO solutions

As mentioned above, Open Baton is not the only solution capable of providing NFV Management and Orchestration following the ETSI NFV-MANO specification.
Among the most relevant ones, it is worthy to mention [23]:

- **OpenSource Mano (OSM)**, a project directly hosted by ETSI to develop an Open Source NFV MANO software stack aligned with ETSI NFV;
- **OpenMano**, developed by **Telefónica**, which also provides an orchestration and NFVI management solution striving to be conformant to ETSI NFV;
- **OPEN-O**, a project realised under the auspices of the Linux Foundation and sponsored by several Chinese telecommunication companies, to develop an open source framework and orchestrator for agile software-defined networking (SDN) and NFV operations;
- **Cloudify**, a solution offering an open source orchestration framework using **TOSCA** based definitions.

All of these projects are generally written in the Python language, and claim adherence to open standards like TOSCA while supporting **OpenStack**, to

provide a model which largely follows (or at least resembles) the ETSI NFV MANO standard described in this chapter.

Open Baton has been chosen by this thesis thanks to its high level of adherence with the ETSI NFV-MANO standard (version 1.1), and its unique plugin based VIM driver system, which has greatly simplified the design and integration of a new Virtual Infrastructure Management solution in its already existing MANO framework.

Chapter 3

Specification and Design

The previous chapter has briefly introduced the reader to the state of the art in Network Function Virtualisation and OS container technologies, focusing on their respective characteristics and the benefits they may offer to the industry during this phase of transition towards the cloud and software defined networks. Chapter 3 will focus on the challenges encountered while designing an NFV solution based on application containers and the Open Baton NFV MANO framework, illustrating how tackling these issues has led to certain design choices in the final architecture of the orchestration solution at the core of this thesis.

Sections 3.1 and 3.2 will again focus on the ETSI NFV-MANO standard to understand how Virtual Network Functions are deployed on virtualised resources and which requirements and features the virtualisation platform hosting them should provide.

Sections 3.3 and 3.4 will analyse how containerisation solutions work in more detail, illustrating what they offer and the considerations needed to achieve the final goal of deploying VNFs on top of them.

Finally, sections 3.5 and 3.6 will study how the acquired knowledge can be applied to the Open Baton framework, understanding which components need to be realised and how they should work together with each other and the other components of the MANO infrastructure.

3.1 Structure of a VNF

As previously said, a Virtual Network Functions represents the virtualisation of a network function, whose functional behaviour and state are largely independent from whether the NF is virtualised or not; the external behaviour of the component is expected to be identical in both cases. [24]

Each VNF is composed of one or multiple components, whose mapping over virtual machines is completely implementation defined: a VNF can be deployed on top of a single VM, or multiple instances can be used instead, to host each single component of the VNF in complete isolation.

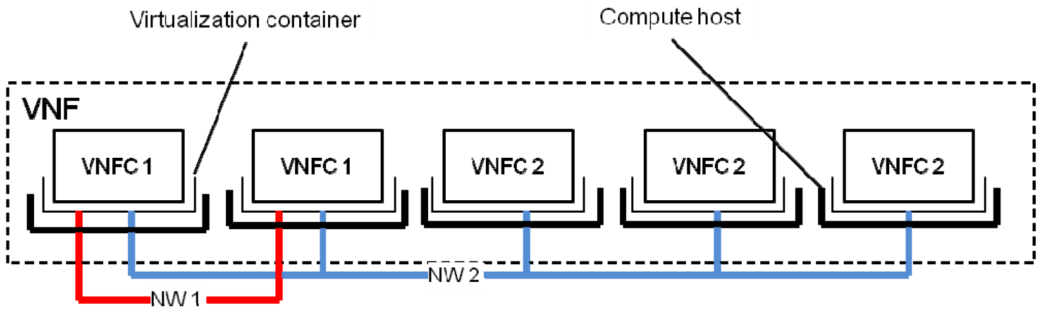

FIGURE 3.1: A VNF can be distributed into several VNFCs, interconnected by networks.

3.1.1 VDU and VNFCs

The **VNF Descriptor** describes one or more **Virtual Deployment Units (VDUs)**, entities representing a single unit of deployable **VNF Components (VNFCs)**.

A VNFC represents a component comprising the VNF; its descriptor, contained in the VDU, describes how the VNFC instances are linked together to create a complete Network Function connected through the **Connection Points** they specify. A single component generally matches with a single Virtual Machine under control of a VIM instance.

A VDU contains informations like the base image to be used when instantiating a VNFC instance, the parameters and requirements necessary to create the components, and how much the unit can scale.

One of the most important tasks to be accomplished while designing and implementing the container-aware NFV solution at the core of this thesis is to determine how a software container can be used to host VNFC instances. This will be explained in further detail in section 3.6 and in Chapter 4, where full design and implementation details will be given to the reader.

3.1.2 Virtual Links

Another important requirement to be considered is how the deployed VNFCs will connect with each other (and the world outside if necessary) after being deployed into an OS container.

The VNFD specifies the **Virtual Links (VLs)** and **connection points** of the deployed instances. These entities represent the interconnections of the VNFs with each other and with the outside network, as shown in Figure 3.2.

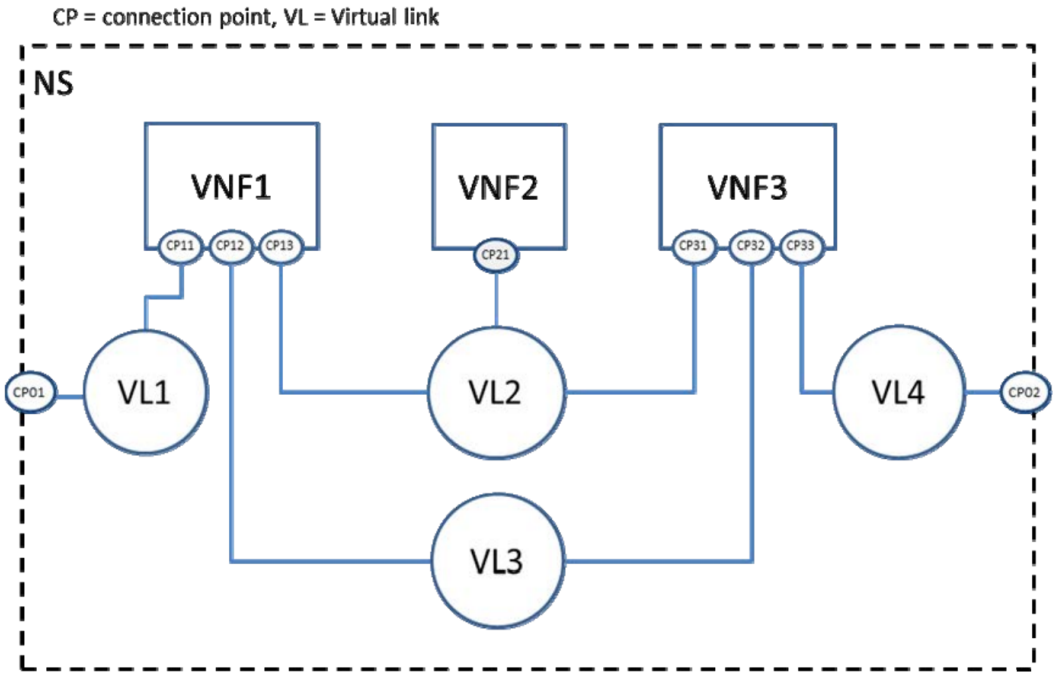

FIGURE 3.2: How Virtual Links and Connection points interconnect VNFs in a typical deployment

The connectivity options exposed to the NFVO are described in specified **Virtual Link Descriptors (VLDs)**; the orchestrator also obtains information from a **VNF Forward Graph**, a graph of all of the VNFs interconnections which is needed to determine how the functions are interconnected together. The data coming from the instances will be passed to a lower level system, to enable logical configuration of pre-existing hardware and software networking components.

The implementation of these services within the infrastructure network is often dependent on the physical locations of the end points, and the nature of the technology itself (i.e., the link between two VNFs sharing a hypervisor on one server could be connected using a Virtual Switch under a given virtual technology, or can otherwise be based on an external Ethernet switch). The VLD contains a description of each Virtual Link, useful to determine

where the VNF should be placed with respect to the current infrastructure; this in particular is useful to determine which available VIM between those indicated by the descriptor will be responsible for the management of the virtualised network resources.

The VIM (or another Network Controller) can use this information to establish the appropriate paths and VLANs, using the basic topology described in the VLD; it also needs to ensure that the other parameters required by the VNFs and the system are also respected.

For instance, the link can define constraints and requirements on:

- The bandwidth of the link (i.e., the maximum capacity the link can offer);
- The QoS expected from the link, like how much jitter and latency is tolerated from the channel;
- Which test access facilities are offered (like passive monitoring, or active loopbacks at the endpoints).

A link can also be defined as being internal; in this case, it will be reserved to internal communications between the VNFCs.

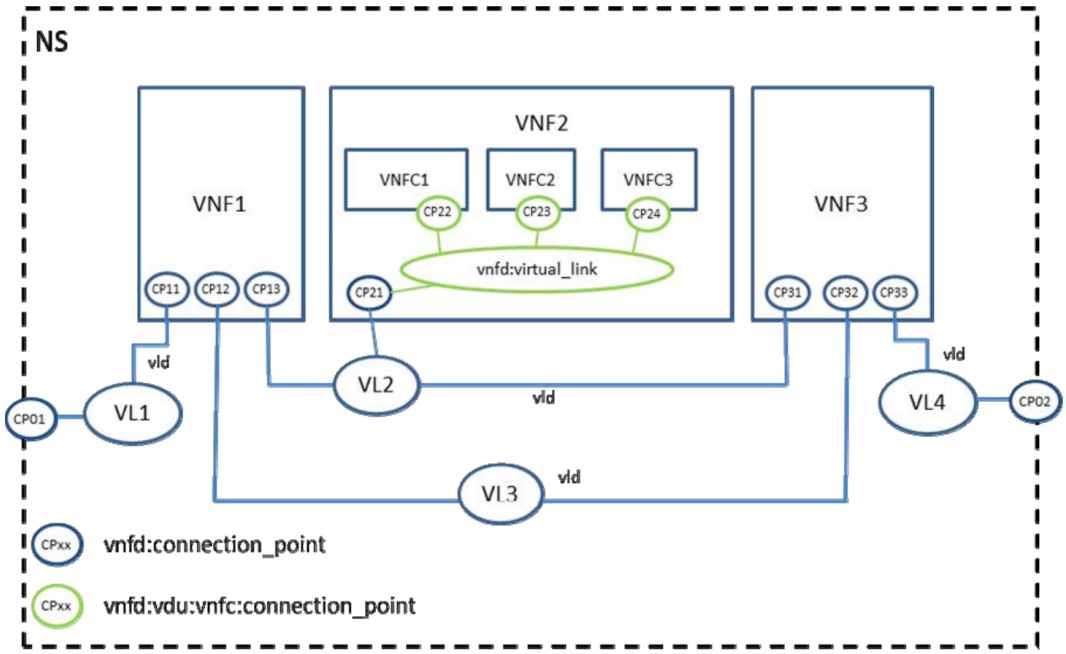

FIGURE 3.3: Internal and External links. An internal link is used to connect the components of VNF2 together

3.2 NFVI Requirements

While not strictly an hypervisor, the containerisation solution chosen and designed in this thesis must still provide all of the features required by a VIM (as specified in 2.2.2); it is therefore useful to analyse what the ETSI NFV-MANO standard requires from the Hypervisor Domain of the NFV Infrastructure, which supports the deployment and execution of virtualised network appliances. [25]

Figure 3.4 illustrates the reference framework of NFV as defined by the standard, including its **Hypervisor, Compute and Infrastructure domains**:

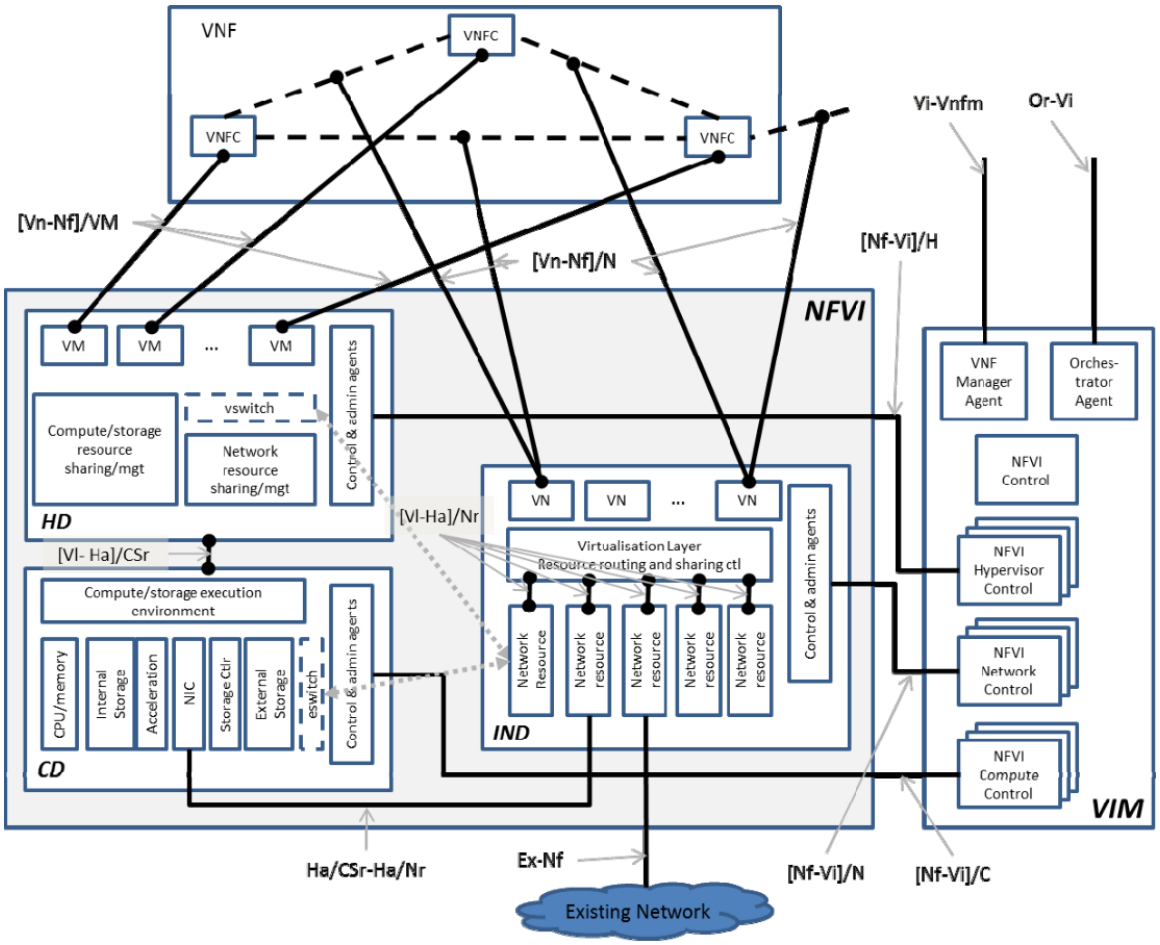

FIGURE 3.4: NFV Reference Architectural Framework as defined by [25]

While the standard itself primarily focuses on the use of full type-1 hypervisors for virtualisation, it also interestingly mentions that:

> *requirements are similar if not the same for implementing Linux containers or other methods for virtualisation.*
> *[...] There needs to be further research w.r.t to Linux Containers, including developing the ecosystem.* [25]

3.2.1 Definition of the Hypervisor Domain

The solution identified in the following pages will at least strive to comply with the main properties defined for an hypervisor:

- **Equivalence** between the environment provided to the programs by the hypervisor and the original machine. Providing this property to the system involves allowing the execution of the same operating systems, tools and application software that can be used in a bare metal-environment, through paravirtualisation and other optimization techniques;

- **Resource control**, mediating the resources of the computer domain to the virtual machines hosting the software appliances. As previously introduced to the reader in Chapter 2, hypervisors provide a greater abstraction from the actual hardware than OS containers, allowing very high levels of portability of virtual machines through the emulation of every piece of the hardware platform (including in some cases even a CPU instruction set).

  Such emulation, however, has a significant performance cost, because of the increased number of CPU cycles needed to emulate a virtual CPU cycle. One of the main targets of Operating System Virtualisation is to largely remove those expenses, at cost of cross-OS portability and reduced isolation;

- **Efficient execution of programs** under the virtualised environment; the difference in speed shown should at worst be only a minor decrease. Even when not emulating a completely the hardware underneath the VM, there can still be significant performance hits caused by certain aspects of virtualisation. While VT extensions offered by computer architectures (like Intel VT and AMD-V) may provide means to offload many CPU bound tasks directly to the host hardware, several other components may yet have to be emulated completely, with a significant performance hit. Software containers are again very strong contenders with regards to efficiency claims, offering a valid virtualisation

platform suitable even for very inexpensive hardware [26]

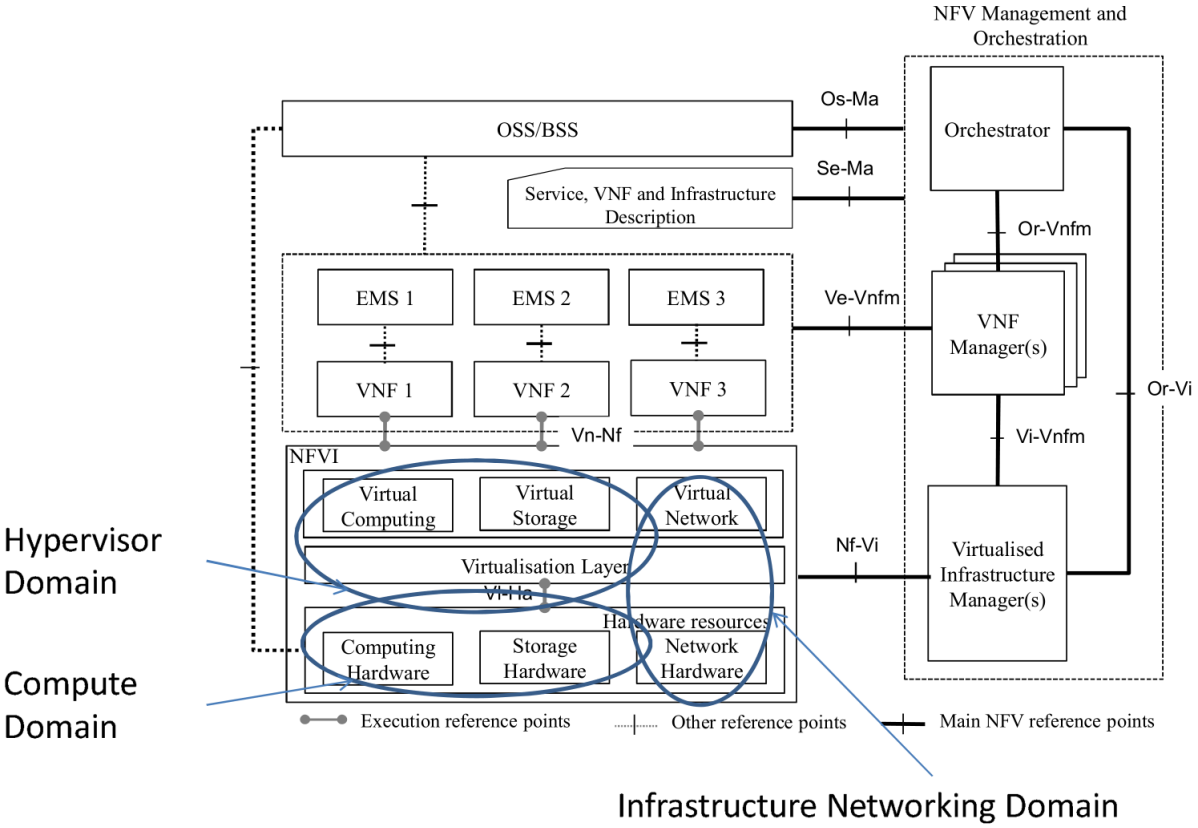

FIGURE 3.5: General Domain Architecture and Associated Interfaces. The Hypervisor domain and the Compute domain are highlighted with blue circles.

3.2.2 VIM-Hypervisor interface

Before integrating a new virtualisation solution into the NFV architecture, it is necessary to either identify or design a suitable Virtualisation Infrastructure Manager (**VIM**) to handle and monitor the operation of the containerisation infrastructure hosting the VNFs.

The standard defines the **Nf-Vi-H** interface as the gateway between the hypervisor and the VIM, and mainly serves two fundamental purposes:

1. To allow the VIM to monitor the hypervisor and the underlying infrastructure, using vendor specific packages, through an event system or

polling. This is caused by the lack of a common standard hypervisor monitoring API, which forces the VIM into using specific hypervisor monitor packages to achieve this funtionality;

2. To allow the VIM to send the necessary commands, configurations, alerts, policies, responses and updates to the hypervisor, of which it is the sole controller.

3.2.3 Requirements for the Hypervisor Domain

It is also important to focus on which requirements the standard directly specifies regarding the hypervisor domain, in order to get a better global view of what the containerisation technology used should provide to the NFV platform, after taking the obvious differences between different virtualisation techniques into consideration.

General requirements

- The platform must offer to the service providers the capability to partially or fully virtualise the network functions they need, in order to create, deploy and operate the services they provide;
- In case parts of the system are kept as legacy, non-virtualised elements, there shall only be a manageable, predictable and acceptable impact on their performance and operations. This also applies to their management systems, that should not be excessively disrupted by the migration;
- The framework must be compatible with heterogeneous services composed of PNFs together with VNFs, implemented across data centre composed of multiple environments.

A containerisation solution should not be less capable of satisfying such requirements than an hypervisor based one. The same guarantees and considerations done while designing the existing VM systems should therefore also apply in this case.

Portability requirements

An important factor to consider is also how well VMs or containers can be moved from an host to another, to respond to outages in their host systems. An abridged version of the requirements needed from an hypervisor to satisfy the challenges portability creates follows below:

- It should be possible to load, execute and move virtual machines across different, standardised data centre multivendor environments. To make this possible, these factors should be satisfied:
  - The VNF shall be able to run on any standard hypervisor;
  - There should be a consistent, multivendor API common with standard hypervisor;
  - A system to allow communications between a VM and its hypervisor needs to be present;
  - The hypervisor needs to have the capability to decline requests;
  - There should be a system to notify a VM when it is going to be moved;
  - Standard, open interfaces should be available for VM intercommunication;
  - The network connectivity services (internal or with the host) must expose their configurations through standard or open interfaces;
- Interfaces to decouple software instances from the underlying infrastructure (e.g. virtual machines and hypervisors) must be provided:

  - An hypervisor provide mechanisms to unbind the VM from the hardware it is bound to. This is necessary before moving the VM instance;
  - The hypervisor shall provide a mechanism for VM migration between hosts, i.e. a system to move the actual VM image from an host to another;
  - Migrating a VM across hosts must be done while preserving the atomicity of the instance, i.e. the image must be copied as a whole, and not in chunks;
  - The hypervisor should provide and export metrics to be used by MANO entities to make predictions regarding how much the migration will impact the performances of the system;

- The migration should also try to optimise the location of the destination target for the VM.

- The hypervisor must be capable of moving the machines while they are running, mapping the resources they required on their original host into resources of their new target destination.

Live VM portability between heterogeneous systems (when hypervisors, host OS and CPU architectures do not coincide) is generally not possible because of software or hardware constraints. Both hypervisors and containerisation offer the ability to move, pause and restore a virtual machine, providing efficient tools to allow portability between compatible hosts and to create checkpoints.

Elasticity and scaling requirements

The solution under analysis ought to provide to the MANO layer adequate tools to allow VNFs to scale accordingly with the demand and the requirements of the network; this generally involves making available to the NFVO and the management components informations regarding the resources available for scaling, and to correctly satisfy the VM allocation demands coming from the VIM.
This requirements are generally very easy to satisfy, and are intrinsically intertwined with the VM or container management itself; a well designed VIM coupled with a containerisation solution should be capable to provide the required functionalities without additional work.

Resiliency requirements

Mechanisms should be in place to allow the MANO and the NFV framework to recreate a VNF in case of failure; while ensuring uninterrupted availability of resources and services is core responsibility of MANO components, the hypervisor still needs to provide at least a well defined way for checking the liveliness of a VM hosting a VNF. The defined solution will therefore need to expose in some way the state of the containers running under it.

Security requirements

Providing security and is a fundamental requirement for a virtualisation solution: the transition from single, standalone physical devices to virtualised instances running on the same machine arises several issues regarding the need to contain a single function into its own virtual environment. A single virtual machine failing, both in terms of security breaches or wrong behaviour, should not put into jeopardy the rest of the infrastructure.

- Management agents and tools should not be accessible by normal users, to disallow attackers from compromising them before executing malicious actions;
- A single machine should not be able to compromise the hypervisor in any way, including the other machines that may be hosted on it at the same time;
- A single machine must not be able to access resources not allocated to it from the hypervisor;

Containerisation solutions can provide some form of isolation, through process namespacing, storage isolation and the usage of a virtualised network stack, but the same security warranties offered by of type-1 hypervisors cannot be ensured. The sandboxing and resource limitations are directly related to how strong and how vulnerability free are the security features provided by the shared OS kernel, making it a much bigger target for exploitation under the OS Virtualisation model. While technologies to put containers into nano VM, to increase their isolation from each other and from the host, are currently being investigated by major players in the field [21], containers have been shown to provide sufficient security requirements, even when considering the shortcomings described above [27].

Another important security consideration involves the attack surface exposed by the Nf-Vi-H interface. The VIM's necessity to fully control the underlying virtualisation technology means that the interface must offer a great degree of access to the management APIs of the containerisation technology. This can lead to critical vulnerabilities if not managed correctly, giving a potential attacker full capability to create, destroy or modify running containers, compromising the integrity and confidentiality of the VNF data.

Taking into account the technical necessity of many containerisation technologies to be executed by root or Administrator accounts of their host systems further worsens the bill; securing the control channel from unauthorised access is therefore an important challenge to be faced when developing the system.

Service Continuity requirements

It would be ideal for the hypervisor to provide status management functionalities, like container status updates, as it has been previously mentioned above.

Operational and Management requirements

The correct functionality of MANO blocks involved with management largely depends on the availability of resource informations from the virtualisation technology used; MANO uses the available VIMs to handle the lifecycle of the VNFs, and to get informations related to the network and the status of the whole architecture. These operations are accomplished through the VIM, and the hypervisor shall attempt to fulfil the incoming requests of allocation and monitoring.

The new solution will therefore have to expose a complete interface towards the MANO blocks, to allow management operations to be carried on a running container.

Energy Efficiency requirements

Network infrastructures consume significant amounts of energy, and this consumption can benefit greatly from the usage of NFV, thanks to the minor amount of hardware required to run the services, and the possibility to

optimise and scale down unused nodes on the fly thanks to the improved scaling capabilites of VNFs.

The usage of containers can furthermore reduce the energy consumption requirements, thanks to higher efficiency levels enabled by decreased necessity to virtualise the underlying hardware. Reusing the same kernel among multiple containers instead of single instances per machine can lead to lower server loads, and thus to reduced power requirements [28].

Guest Runtime Environment requirements

Software containers are bound to a single OS - CPU architecture pair, and therefore a containerisation solution cannot run the same large selection of images that an hypervisor can run without resorting to hardware virtualisation.
This limitation is largely mitigated by the industry trend to largely converge towards portable runtimes on top of Linux-based OSs on x86-64 machines.

Runtime environments like the JVM, the BEAM Erlang VM and scripting languages like Python are widely supported by Docker and similar software, offering pre-packaged container images that reduce greatly the complexity and times involved with software deployment and setup.

Coexistence and migration requirements

The coexistence and migration requirements of an NFV containerisation solution, with regards to PNFs and legacy systems, are largely similar to those of standard Virtual Machines.
Coexistence with VM-based NFV solutions should be trivial, thanks to the abstractions provided by the architecture itself; a container should behave and be like a VM as long as the MANO components are involved.

3.3 Evaluation of container solutions

The creation of the **Docker** project has increased exponentially the interest in containers as solutions to enable faster, more scalable cloud deployments. A number of solutions have been increasingly been built around the model of containers popularised by Docker, like **Docker Swarm**, **Nomad**, **Kubernetes** and the container oriented GNU/Linux distribution **CoreOS**.

The main innovation introduced to mainstream fame by Docker has been to provide an easy way to wrap a complete setup of a service or application (like a web server, a development environment, ...) inside a filesystem image, containing all of the necessary software already pre-installed and configured together with a stripped down GNU/Linux distribution, ready to be moved, shipped and redeployed several times using the native containerisation features offered by Linux, like namespaces (to isolate processes and resources), OverlayFS (to create multiple virtual file systems on top of the same image) and cgroups (to limit the resource usage of a set of processes).

3.3.1 OCI containers

The success and interest around the technology has led some of the major players in the field of cloud computing to standardise these efforts under the **Open Container Initiative (OCI)**, a lightweight, open governance project under the the Linux Foundation which strives to create open industry standards around a shared container format and a runtime. [13]

FIGURE 3.6: Current members of the Open Container Initiative

The OCI provides two specification regarding a standard runtime and an image format, called the **Runtime Specification (runtime-spec)** and the **Image Specification (image-spec)**, specifying respectively how a filesystem image can be executed and a filesystem image format shared between the various implementations. OCI images are by their own nature extremely easy to move and scale, and are designed to be easy to download and execute. Each image specifies an entry point (i.e. a starting executable) that can be run without no parameters, enforcing the 1:1 identity between a container and the service that runs within it.

A focal point of this thesis will be to design a solution to enable the usage of OCI-like images containing pre-made VNFCs as building blocks of services based on VNFs.

3.4 Docker

Docker has been chosen in this thesis as the target platform on top of which the images will be deployed. The decision to directly use Docker instead of focusing on more complex solutions like Kubernetes has been taken in light of a few important considerations:

- **Popularity**: Docker is by far the most widely used solution based on lightweight OCI container model. Packaged for almost every GNU/Linux distribution and with native support for Windows containers, it is at the core of Kubernetes and most of the containerisation solutions used in the datacentres today;
- **Simplicity**: Docker is lightweight and very simple to setup;
- **Features**: Docker offers many of the required features described above, like networking support, lightweight containers through the **runC** runtime, and a registry of pre-packaged images ready to be used as building block for NFV images. In case such necessity arises, Docker offers a growingly popular Kubernetes competitor called **Docker Swarm** as a feature directly built in into the Docker daemon itself, offering container orchestration, clusterisation and high availability features;
- **Focus**: creating a solution based on Docker instead of a platform using it (as Kubernetes) allows to better understand the principles at the core of the container model, helping to focus more towards a deeper understanding of the task itself.

Using Docker as the reference implementation for this solution doesn't mean that an hard logical dependency on Docker is acceptable: the final design must be abstract enough to allow extensibility and easy adaptability with several container runtimes, offering a generalisable solution for future works involving similar goals.

3.4.1 Overview

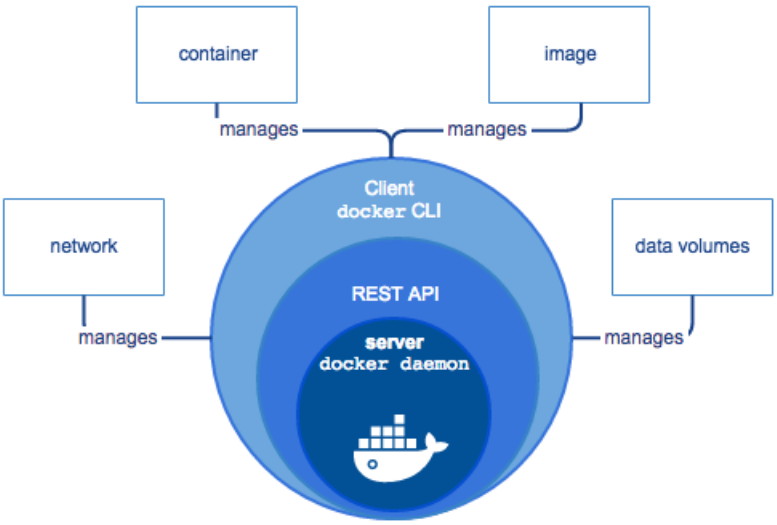

FIGURE 3.7: High level overview of the Docker architecture

Docker provides its containerisation services through the **Docker Engine**, an open source, Apache 2.0 licensed software solution based on a client-server infrastructure that uses the runC OCI runtime as its default backend. The main server daemon of the engine (**dockerd**) offers a REST API to allow clients (like the CLI `docker` tool) to send commands and query the state of the system and its entities [29]. This server process creates and manages Docker objects, such as images, containers, networks, and data volumes, exposing them to the API as well-defined interfaces and objects accessible from

local or remote clients through UNIX sockets, Windows Named Pipes or TCP connections (secured through TLS).

Images

Docker images contain a read-only operating system filesystem, upon which one or more applications have been installed and configured, together with the instructions to create instances of containers based on it.
Docker images are not a single, monolithic entity, but a series of incrementally built layers. Docker uses union file systems technologies (like UnionFS or OverlayFS) to combine these layers together into a single image at the time of container instantiation, allowing files and directories belonging to separate images to be transparently overlaid to create a single, coherent file system.

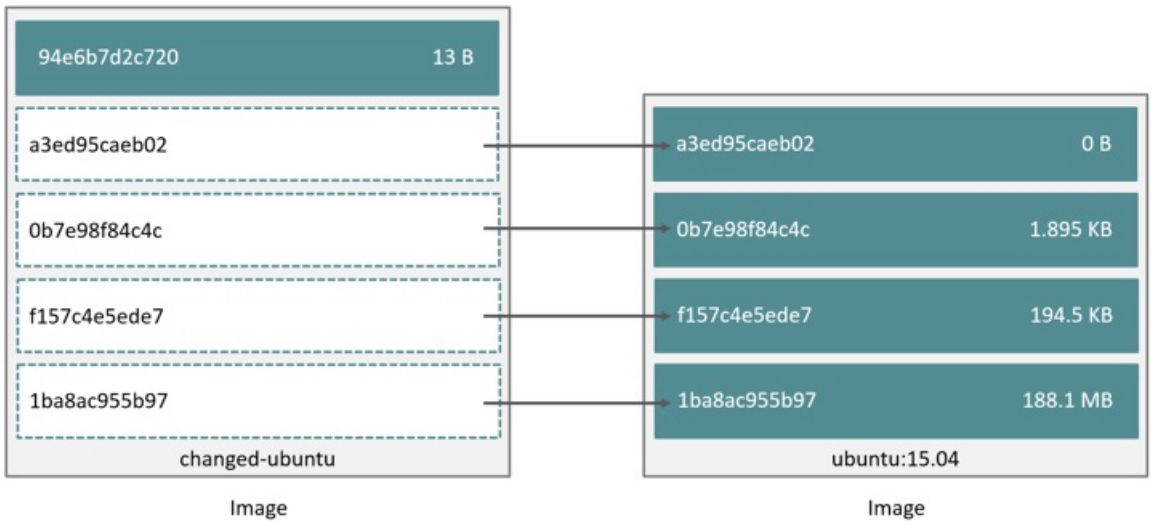

FIGURE 3.8: Incremental derivation of a Docker Image

These layered system enables an high degree of reuse for Docker images, allowing an image to be used as it is as base to another derived one; every change (such as in case of updates to an application) is reflected by a new layer, that either replaces or simply stacks on top of the existing ones, which

are left unaltered. Distributing an update or a new derived image therefore only requires the transfer of those changed layers, speeding up download and setup times and reducing storage usage [30]. The layer dependency resolution process is hidden to the user; the daemon takes care to resolve the correct dependencies at instantiation time.

Docker images can either be referenced through unique hashes or through tags defined at build time, simplifying and abstracting their usage to the user.

Dockerfiles

Docker images are generally built through either commits of the current state of a container's filesystem (`docker commit`), import of an existing OS filesystem in `TAR` format (`docker import`, to create a *base image*) or through a textual description of the instructions necessary to achieve the desired image (**Dockerfile**); because of its inherently not automated nature, the process of manually creating images from ad-hoc containers is error prone and generally discouraged.
A Dockerfile defines a custom language, focused on a set of useful commands to create images. A small sample of the possible statements follows below[1].

- `FROM`, which specifies the image that will be used as a parent of the image under construction;
- `MAINTAINER`, to indicate the person that maintains the image;
- `RUN`, to specify a command to be executed inside the image environment during the build process;

[1]The complete list of the possible Dockerfile commands is much longer, and it is specified in the online Docker documentation, at which every interested reader should refer to.

- `COPY` and `ADD`, to insert files, directories or remote resources into the image;
- `ENV`, to set environment variables in the image environment, which will be present at the time of container instantiation;
- `CMD` and `ENTRYPOINT`, to set which executable will be executed by default after instantiating the container.

The statements contained in the Dockerfile are read and evaluated sequentially during the `docker build` process, committing the results into a new image layer. The final of these will represent a finite image, ready to be deployed as one or more containers.

Dockerfiles will be used by the solution under construction as a way to define easily deployable VNF components, as shown in the next chapter.

Registries and Docker Hub

The images can be pushed and stored into a **Docker Registry**, an either public or private service that can be accessed from a Docker daemon to fetch and retrieve prebuilt Docker images.

The **Docker Hub** is a public Docker registry provided by Docker Inc., containing a huge and open collection of existing images from first and third party sources; everybody is free to push and pull images from and to this registry, enabling a container model in which the access to ready made, high quality images is extremely easy and accessible and in which updates to container images can be easily obtained from a central, remote repository.

The Hub will be used as the source of image templates used as base to the custom VNFC images for this NFV container solution.

3.4.2 Containers

A Docker container is an instance of a Docker image, managed by the Docker Engine. It consists of the image filesystem itself, plus a writeable layer on top to store file system modifications and the status of a container. Each container built from an image shares the same (immutable) filesystem with the others; the data state is kept inside a thin writeable layer uniquely related with the container, that is permanently deleted after its parent is removed, except if it was committed into a new image. Docker uses multiple storage drivers to manage and provide both the image layers and the writable container layer, like AuxFS, Btrfs, ZFS, OverlayFS ans such. Each one of these provides the stackable image layers and the copy-on-write (CoW) capabilities that Docker images requires.

Each container has its own namespaced network stack, configured either dynamically or through configuration metadata at creation time. This data also keeps parameters like the environment variables set for a single container instance (helpful to pass parameters to a container before starting it), the ports that need to be forwarded towards the host, and several other options.

The containers can be seen as extremely small virtual machines, running on top of the same Linux kernel. Each time the user orders the daemon to run a container, this sequence of steps is carried out by the Docker Engine to accomplish the request:

1. The availability of the requested image is evaluated: if the image already exists locally, the Docker Engine uses it for the new container straight away. Otherwise, the Engine pulls it from the Docker Hub;

2. A new container is created: Docker uses the image to create a container, as specified;
3. A new filesystem is created, and a read-write layer is mounted on top of it;
4. A new network interface is created for the container (plus a bridge if outside connectivity is desired) under a kernel namespace
5. The network interface is attached to a Docker network, and an IP address is assigned to it, either manually (through an API parameter) or automatically from a pool;
6. The process specified from the Start request (or the `CMD/ENTRYPOINT` if not present) is executed inside the container;
7. If requested, the standard streams are attached to the application output, to allow interactive interaction with it.

The application is now running under its own separate container, which can be managed and interacted with in a fashion similar to virtual machines.

This thesis will focus on how the NFV layer can interact with Docker, to create and orchestrate VNFs contained inside containers.

3.4.3 Container Lifecycle

The states a container can be are those exposed by the Docker API; Docker Remote API 1.26 defines the following statuses for containers:

- **created**, when it has been created but not yet started;
- **restarting**, during the process of being restarted;
- **running**, when running normally

- **paused**, when its processes have been paused using the capabilities offered by the Linux kernel;
- **exited**, when the container has run completely and has exited;
- **dead**, when the container failed to stop for some reason.

An important point to be taken into consideration during the implementation step is how these states can be matched into the states of a VNFC Instance. Correct abstractions should be put into place to abstract these issues behind an appropriate layer of isolation, to avoid the solution to be too attached to the Docker API and its assumptions.

3.4.4 Access security

The Docker daemon generally runs as the Administrator of the system, or under an user with comparable permissions. It is therefore critical for the VIM manager under design to provide options that do not involve having `dockerd` listening on an external network, to avoid to expose a large attack surface to potential intruders.

3.5 Open Baton

Section 2.8 has introduced the reader to **Open Baton**, an ETSI and TOSCA compliant NFV MANO solution, explaining the role of its NFVO and its principal components and features. This section will now briefly analyse the design of the Open Baton NFV framework, to better explain the decisions that have been taken to reach the final goal of this thesis to deploy Virtual Network Functions on top of software containers.

3.5.1 VIM support

The Open Baton NFVO needs to be able to obtain virtualised resources from a virtualisation host, like an hypervisor or, in this case, a software containers solution like Docker. This is generally accomplished through a Virtualised Infrastructure Manager, a component whose main task is to both provide an access point to an NFVI and to manage the resources hosted in it; VIMs have been described in detail at 2.2.2.

The Open Baton NFVO provides specific support both generic support for arbitrary VIMs, through an RPC based plugin interface; each plugin shall implement support the functions listed in Table 3.1 [31]

A typical Open Baton setup generally uses OpenStack as its VIM, to allocate Servers to be used as deployment targets for the VNFCs that need to be allocated, to manage their connections, the images uses to instantiate them and in general the resources available to the NFVI. This is implemented through the usage of the **openstack-plugin**, a standalone component that uses the plugin RPC interface of the NFVO to provide access to OpenStack instances as NFV entities.

It is important to also mention that NFVO plugins are generally designed to be stateless with regards to the invocation of their functions; each call to a plugin contains all of the parameters necessary to accomplish it, without any state kept inside the plugin between two consecutive calls. This allows a single plugin to be used to manage several VIM instances, enabling parallel requests without interferences between them.

Function	Description
`listImages`	Queries the VIM for a list of the available NFV Images
`listServer`	Returns a list of the server under control of the VIM
`listNetworks`	Returns all of the known networks
`listFlavors`	Returns a list of available flavours
`launchInstanceAndWait`	Creates an instance of a Server, waiting for it to start if necessary
`deleteServerByIdAndWait`	Deletes a Server with a given ID, waiting for it to shut down if necessary
`createNetwork`	Creates a new network
`getNetworkById`	Returns the network having the given ID
`updateNetwork`	Updates a network
`deleteNetwork`	Deletes a network
`createSubnet`	Creates a new subnet
`updateSubnet`	Updates a subnet
`deleteSubnet`	Deletes a subnet
`getSubnetsExtIds`	Returns the list of the ExtIDs of subnets
`addFlavor`	Adds a new flavour
`updateFlavor`	Updates a flavour
`deleteFlavor`	Deletes a given flavour
`addImage`	Adds a new NFV Image
`updateImage`	Updates an NFV Image
`copyImage`	Copies a NFV Image to a new one
`deleteImage`	Deletes an NFV Image
`getQuota`	Returns the resource Quota of the VIM
`getType`	Returns the type of the current VIM

TABLE 3.1: Functions defined by the Open Baton VIM Driver plugin interface

3.5.2 VNFM considerations

The VNF Manager is a fully independent entity in the Open Baton architecture that communicates with the NFVO and the other components through a shared message bus or other RPC protocols. The manager exposes an interface that is invoked by the NFVO when the necessity to manage a VNF arises, operating through its **VNF Record (VNFR)** to modify and manage the state of the function throughout its lifecycle. The available operations exposed by the VNFM interface are those described in Table 3.2.

Open Baton provides a Generic VNFM as a mean to manage VNFCs providing a management service named **Element Management System**, that carries out remote configuration scripts as instructed by the VNFM; such solution is largely used to configure the otherwise bare VM images created by OpenStack from a base OS image.

A VNFM can, according to the MANO standard, directly allocate the resources needed by the VNFs it manages using a direct access to a VIM through its `Vi-Vnfm` reference point, instead of relying on the NFVO to carry the allocation task [4]; in an Open Baton context, this can avoid in certain specific cases the necessity to design an additional VIM plugin.

Function	Description
instantiate	Creates a new VNFR from a descriptor
query	Retrieves the state of a VNF instance
scale	Scales a VNF (in/out, up/down)
checkInstantiationFeasibility	Checks if a VNF can be instantiated
heal	Handles a failed VNF instance, to support healing capabilities
updateSoftware	Applies a very limited software update
modify	Instructs the VNFM to make structural changes to a VNF instance
upgradeSoftware	Applies a new software release to a VNF instance
terminate	Manages the termination of a VNF instance
handleError	Handles an NFVO error, in response to a previous action
checkEMS	Checks if the EMS is available on a VNF instance
checkEmsStarted	Checks if the EMS has started on a VNF instance
start	Starts a previously created VNF instance
stop	Stops a previously started VNF instance
startVNFCInstance	Starts a VNFC Instance
stopVNFCInstance	Stops a VNFC instance
configure	Configures a VNF instance
resume	Resumes a VNF instance
notifyChange	Provides notifications about the state changes of a VNF instance

TABLE 3.2: Functions defined by the Open Baton VNFM interface

In case the VNFM chooses to opt-out from carrying out the allocation step itself, it will need to request from the NFVO the allocation of a VNF before returning a VNFR in `INSTANTIATE`; the VNFM runtime should call the `GRANT_OPERATION` (to ask the NFVO to query the required VIM for resources) NFVO method followed by an `ALLOCATE_RESOURCE` request, to request the allocation of a VNF by the server and serve the `INSTANTIATE` request correctly.

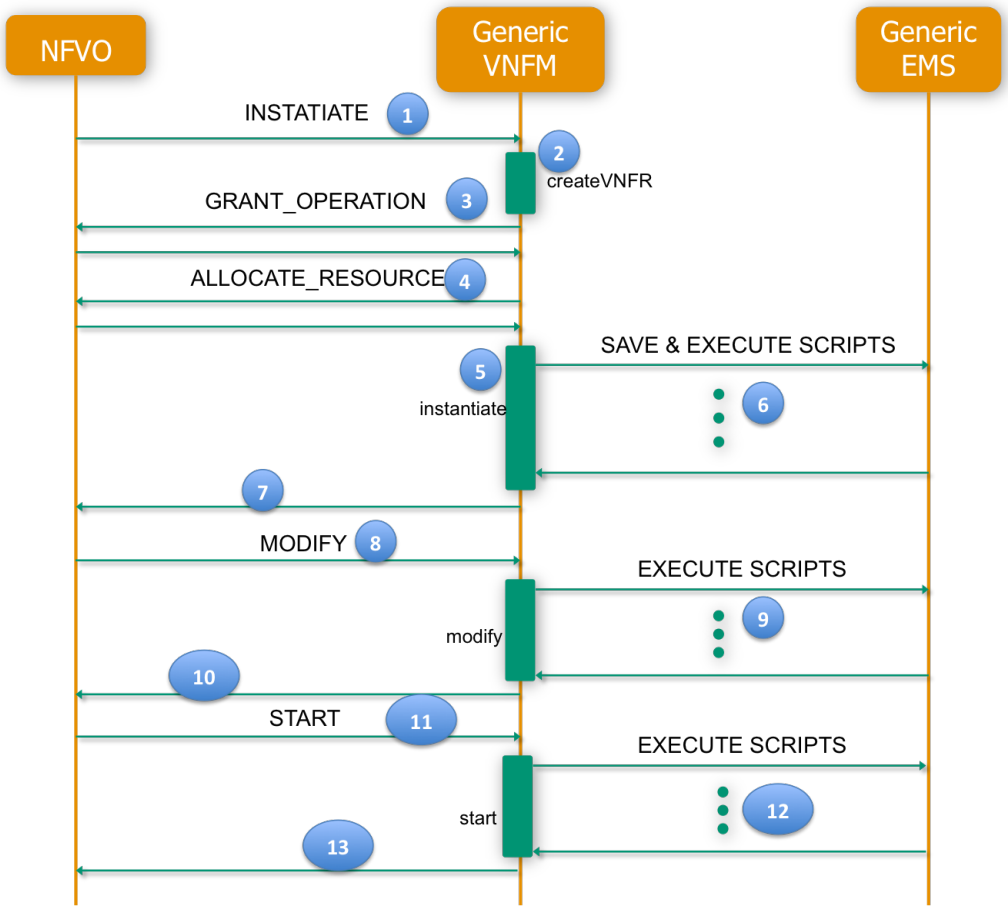

FIGURE 3.9: Sequence diagram representing the sequence of operations carried out by the infrastructure to allocate a VNF, using the Generic VNFM.

A last consideration should also be done regarding the different nature of

the software shipped through application containers and the role (and usefulness) that an EMS can potentially have in an application container based context. The fact that a container contains by default the complete set of scripts needed to instantiate the VNF strips away from the EMS the important duty of installing and configuring the service software, and therefore it is necessary to rethink the (possile) task this component may have inside the system designed by this thesis. Omitting the it from the final architecture will require a new VNFM, unencumbered from the hard dependency on the EMS that Generic has.

3.5.3 VNF lifecycle considerations

The Open Baton framework defines VNF entities with a precise life cycle, as shown in Figure 3.10.

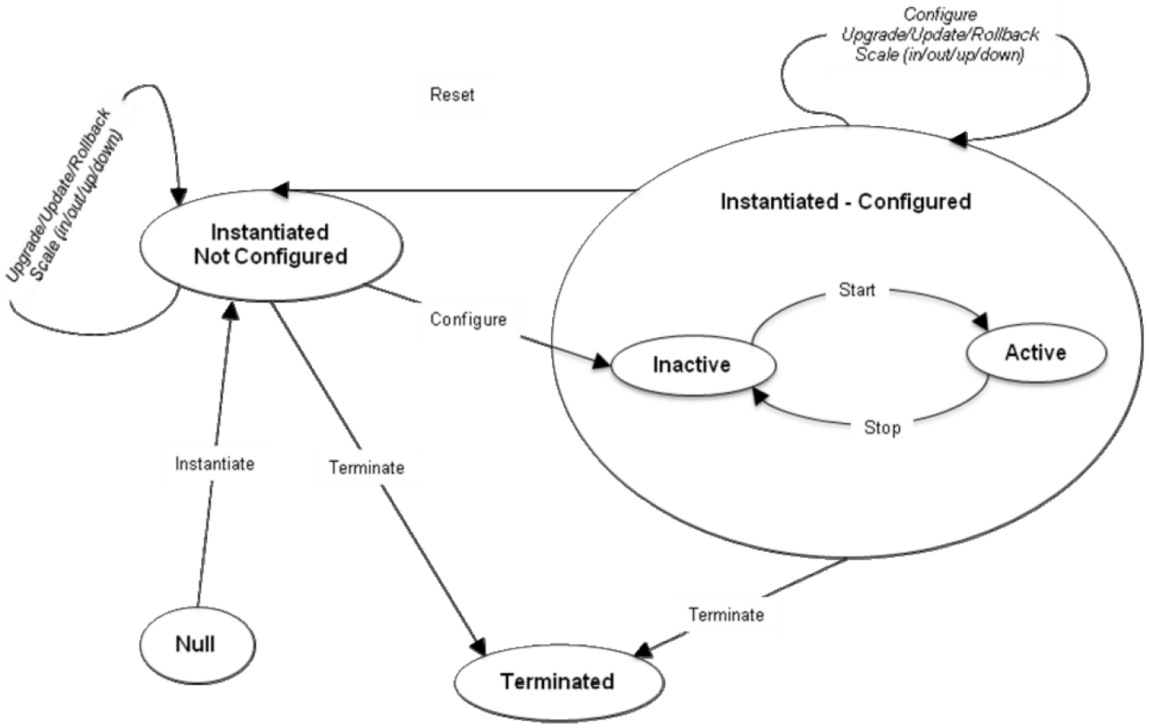

FIGURE 3.10: State diagram representing the different states of a VNF inside in the Open Baton framework. State changes happen through VNFM operations, as shown by the arches.

Handling the current state of a VNF a responsibility solely reserved to its Manager, and the NFVO must use the VNFM interface to instruct it to carry out the VNF operations it needs to do on a given function. The `Ve-Vnfm-vnf` (or `Ve-Vnfm-em`) reference point is then used by the manager to access the VNFC hosting the instance (or to contact the EMS), to execute operations on it.

As previously mentioned, Generic relies on the EMS being available (usually through a RabbitMQ message bus) to interface with the VMs containing the instances and to execute the configuration scripts specified into the VNF Descriptor for a given event.

Dependencies

A VNFD is generally stored and described as part of a larger **NSD** (Network Service Descriptor), a structure that represents a Network Service. Each NS may potentially be composed of multiple VNFs, interconnected by Virtual Links, with possible reciprocal runtime dependencies; a VNF may require the availability of another component of its package to carry out its tasks and to function correctly.

Parameter	**Meaning**
`source`	Name of the VNFR that provides parameters
`target`	Name of the VNFR that requires parameters
`parameters`	Name of the parameters required by the target

TABLE 3.3: Structure of a VNF Dependency

One of the main responsibilities of the NFVO is to correctly handle the order in which certain operations are carried on the VNFs belonging a package with dependencies. This encompasses tasks such:

1. Satisfying the **VNF Dependencies** (see Table 3.3) defined in the NSD, matching the target of each dependency with a source able to satisfy its requirements;

2. Modifying the configuration contained in the VNF, to specify the address on which the dependency source resides in the scope of the Virtual Link between them;

3. Starting the VNFs, respecting the order defined by their dependencies. For instance, a server offering a given service should not be started before the database instance it has been instructed to use;

The tasks defined by 1 and 3 do not usually require intervention by components external to the NFVO itself; the Open Baton orchestrator is capable to resolve dependencies through its internal dependency management services, and to handle the ordered issuing of events if configured to do so.

The task defined at 2 instead heavily relies on the VNFM, which needs to be able to modify the configuration of the target VNF to introduce in it the parameters resolved by the NFVO, and on the VIM, which needs to expose to the NFVO exact knowledge about the location of the source VNF.

The knowledge of the address of a VNF becomes a crucial issue when comparing the OpenStack case with the Docker case under consideration. In the former, a fully functional bare VM is instantiated by the plugin with configured network links and allocated IPs, ready to be used as a server for a VNF; this is not an approach compatible with the application container

model, which bundles the software inside of the image before of the instantiation, allocating resources only at the time of container startup.

Therefore, it is clear that the life cycles defined by Open Baton VNFs and Docker containers do not clearly map, for the following reasons:

- Dependencies are not resolved until a VNFR containing the addresses of the VNFCs is returned to the NFVO by the VNFM. It is thus necessary to let the NFVO know where the components are (or will be) located at the time an `INSTANTIATE` call to the manager returns;
- Creating a Docker container does not automatically assign IPs to the networks it is attached to, a task that is delayed until container startup. Manual allocation of IPs is therefore needed;
- The environment of Docker containers is generally immutable after their creation, and the VNFM cannot change it in during a `MODIFY` request to contain the addresses of the resolved dependencies as environment variables for the instance;
- Starting up a container at VNF instantiation to force address allocation is not a solution either, because that would also mean launching the VNF contained in it before being able to configure it correctly.
- Mapping directly a VNFC into a container is extremely likely to cause interdependency and intertwining of the implemented solution with Docker, jeopardising possible future extensions of the system to other containerisation technologies.

3.6 Solution Design

The detailed analysis of the system and its necessary requirements executed in the previous sections makes possible to define a final design for the NFV container system under construction. The following sections will give the reader an abstract overview of the various components that will be developed and implemented in the next chapter, to resolve the challenges previously delineated.

Overview

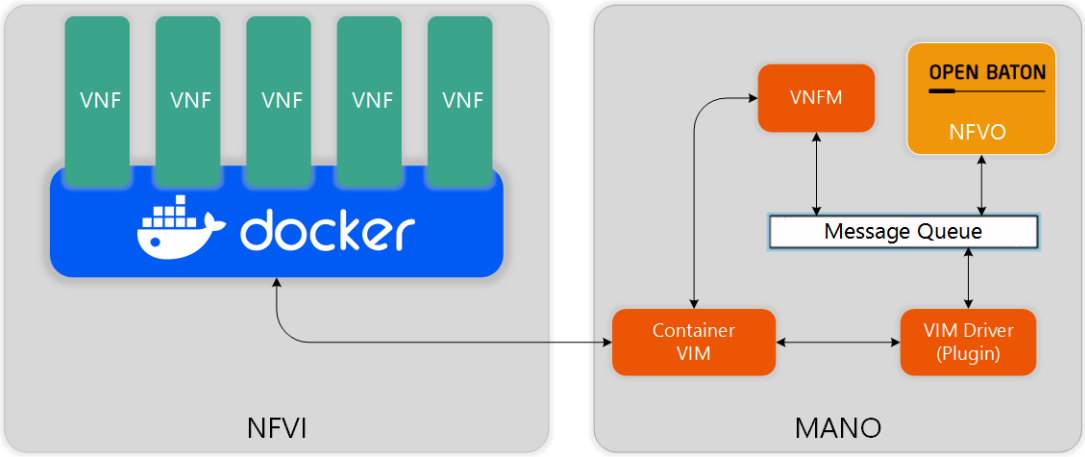

FIGURE 3.11: Abstract, high level architectural overview of the project.

The system has been abstractly designed to integrate with the existing Open Baton system, without any required modification to the existing components.

This thesis introduces three entities to the model: a **VIM** to manage containers, a **VIM Driver** plugin to expose it to the NFVO, and a **VNFM** conforming with the model described by the new driver.

An EMS system is notably assent in this design, given that the different conceptual nature of application containers tends to avoid the installation of additional software after their instantiation.

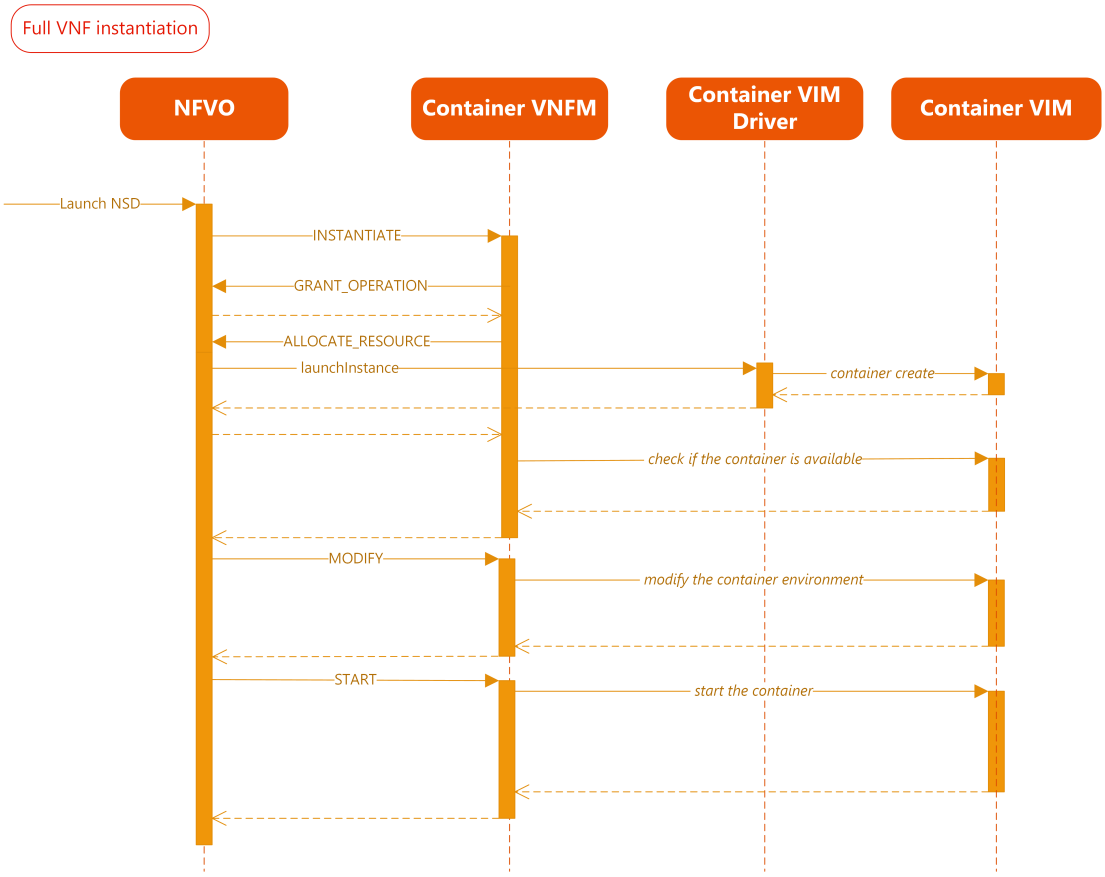

FIGURE 3.12: Abstract representation of the interactions between the NFVO and the new components.

VIM

A new VIM is needed because of the different nature of Docker containers, that makes a strong layer of astractions necessary. This component will be responsible to hold any form of state that may ever be needed by the system, to allow the other components to be completely stateless.

The MANO layer must only know and use abstractions of the Docker entities exported by the VIM, to guarantee maximum extensibility and reusability of the infrastructure with other platforms; this service and its clients will therefore only expose data structures as defined in the Open Baton catalogue of MANO types.

FIGURE 3.13: Abstraction provided by the Container VIM

The VIM will also have to implement an authentication system that fully integrates with the Open Baton model (username and password authentication) while at the same time reducing at a minimum the external exposure to the Docker daemon (because of the considerations expressed at 3.4.4) through the implementation of only those functions that will be deemed necessary to the client.

Section 4.4 describes a full implementation of a VIM compliant with the requirements previously defined, using Docker and a custom **Pop** protocol to provide a container based infrastructure through generic MANO components.

VIM Driver

The model chosen for this architecture will put the responsibility of resource allocation to the NFVO itself, instead of relying on the VNFM to carry those duties, to increase the genericness of the solution.

The new VIM Driver will have to be stateless, to allow parallel and idempotent requests from the NFVO, and to be efficient by using caches to reduce reconnection times.

A full implementation of a driver providing the necessary features has been implemented as an Open Baton plugin, and described at 4.6.1.

VNFM

The container VNFM will need to be as generic as possible, while implementing the semantics expected by the underlying infrastructure.
The VNFM will need a channel to communicate with the VIM, to execute management actions on the VNFs like `START` and `MODIFY`; this should not be done, however, at expense of the statelessness, to avoid to encumber it with the expense of critical zones that may hamper the parallelism of the request handling process. Chapter 4 will thoughtfully explain how this has been achieved, while discussing the implementation details of the VNFM and the other components.

A newly created Generic VNFM (mgmt), a manager designed to be able to interoperate with the aforementioned components using an abstract protocol, will be introduced at 4.6.4.

Chapter 4

Implementation insights

Chapter 3 has introduced the reader to the system at the core of this thesis, abstractly explaining the challenges and requirements involved with the design of an Open Baton based NFV containerisation system.

The following chapter will instead focus on the concrete implementation of the NFV architecture realised during this thesis, exhaustively explaining the different interconnected components that comprise the final infrastructure, their features, their possible uses, and the technologies used to build them.

4.1 Architectural overview

The implementation step has led to the realisation of several Go packages containing libraries and services that leverage the **Go Open Baton** libraries (see Appendix A) to interoperate with the Open Baton NFVO. These components have been designed to satisfy the necessary requirements delineated by the the analysis process made in Chapter 3 for the architecture under construction.

Here follows a list of the main elements, each one coupled with a short summary of its nature and purpose.

- **Pop**, a protocol that defines a container model with Open Baton compatible semantics. Described using Protocol Buffers files, it combines them with gRPC to offer an easily extensible and multi-language client-server solution;
- `pop/client`, a library that uses the `protoc`-generated Pop client to provide a translation layer between Pop data types and Open Baton Catalogue types;
- A **Docker Pop Server** library (`docker-pop-server`), which implements a daemon capable to manage and expose Docker containers . This server implements the routes required by the gRPC-generated stub, exposing abstract Pop entities that implement containers, networks, images and flavours dynamically associated with the resources offered by a Docker Engine instance;
- `docker-popd`, a daemon that uses the `docker-pop-server` library to provide a container-based VIM implementation;
- `pop`, a simple CLI client to directly query a `docker-pop-server` server instance;
- `mgmt`, an AMQP RPC based protocol meant to be used by a VNFM to manage le lifecyle of the VNFs contained in a VIM. This component, and how it used by the solution, will be explained thoroughly in 4.6.2;
- `mgmt-gvnfmd`, a generic VNF Manager that uses `mgmt` to manage the lifecycle of VNFs;
- `pop-plugind`, a VIM Driver that extends the Open Baton NFVO to handle Pop VIM instances. The Pop plugin also dynamically configures

`mgmt` connections for each VIM instance it becomes aware of, to allow the VNFM to send management commands to the VIM infrastructure.

Figure 4.1 shows the final architecture of the system after deployment, complete with its components and their interactions.

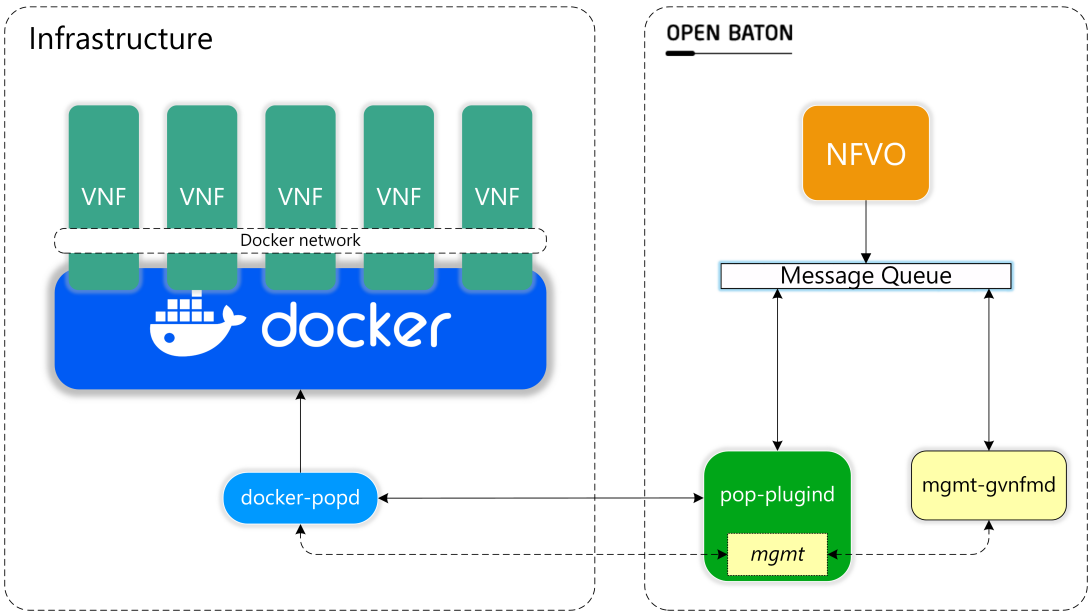

FIGURE 4.1: Final architectural overview of the deployed system. The dashed arrows represent the logical link between the VIM and the VNFM created by `mgmt`.

4.2 Pop protocol

Pop (an acronym for **Point of Presence**) is a client-server protocol that abstracts a container platform, exposing an interface suitable to be used as an **NFVI-PoP**. Described using the **Protocol Buffers** serialisation language, it provides an easily extensible framework on top of which build the design of the VIM, and uses the gRPC RPC framework by Google, which is, according to its authors [32],

> a modern open source high performance RPC framework that can run in any environment. It can efficiently connect services in and

across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

The main usage scenarios:

- Efficiently connecting polyglot services in microservices style architecture
- Connecting mobile devices, browser clients to backend services
- Generating efficient client libraries

Core Features that make it awesome:

- Idiomatic client libraries in 10 languages
- Highly efficient on wire and with a simple service definition framework
- Bi-directional streaming with HTTP/2 based transport
- Pluggable auth, tracing, load balancing and health checking

gRPC allows to create high-performance RPC protocols from Protocol Buffer definitions, automatically generating a client and a server stub in one of its many first or third party supported languages.

The protocol implements several messages that define entities (see Appendix B for a full listing) such as:

- **Flavour**: defines the amounts of resources provided by a container. In the current implementation, Flavours are provided as an interoperability measure with Open Baton, which expects a VIM to expose them;

- **Image**: represents an Image, identified by an ID and a set of names;
- **Network**: represents a network, either internal or external, identified by an ID and a name. Each network defines one or more subnets, with its own address and (if there's one) its gateway;
- **Container**: representing an instance of a given image, each container is identified by an ID, a set of names, a flavour and a map of endpoints. Each endpoint represents the networks the instance will be attached to, including its IPs (empty elements mean automatic assignment by the server); an empty map will cause the instance to be automatically attached to a private internal network.
  Additionally, each container can have **Metadata** set to it, a map of strings modifiable from the time of creation to the moment of startup that enables remote dynamic configuration of a new container instance. Using *environment variables*, the key-value string pairs contained in it will be exposed at runtime to the underlying container, providing to the software the directives it needs, like the addresses of their dependencies and other setup flags.

These abstractions, and Pop itself, have been designed with the goal to create a suitable framework to enable the creation of VIMs to access containerisation technologies in an Open Baton NFV context. A Pop server allows to overcome the lifecycle differences between VNFs and containers, keeping the state necessary to realise this away from its MANO users.

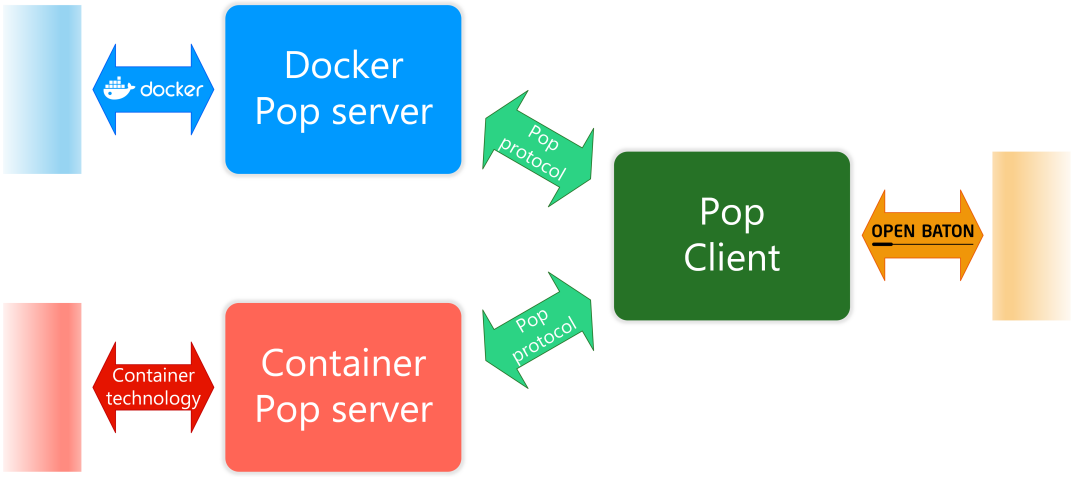

FIGURE 4.2: Pop protcol provides an abstraction that links together several technologies using a single client.

This design also allows to create a single client implementation capable to use the protocol and its semantics to interoperate with any arbitrary container-oriented platform for which a server has been written.

4.2.1 Client-server authentication

A Pop server defines at least one or multiple users, to restrict the confidentiality and the access to the resources of the administered Point of Presence. Before executing any RPC method on the remote service, the client needs to obtain a string token from the server by invoking the `Login(Credentials) returns (Token)` function with valid credentials (username and password); the returned session identifier can then be submitted along the gRPC metadata during a future request.

The validity and scope of a Pop session are completely server defined; a client must therefore handle when necessary the re-authentication of its peer with the remote server. A client can remove one or multiple sessions by invoking `Logout`, which will invalidate all of the tokens specified by the

metadata sent with the request itself.

At the moment, there is no support for ACLs and differentiated permissions; each user is an administrator with full access to the resources contained by its PoP. Supporting this may be considered for future extensions to the protocol.

4.2.2 Query operations

Pop provides several functions to query a PoP, in order to get a list of the known instances of a given entity, as specified in table 4.1. A `Filter` message can be optionally specified in each of these operations to filter only those entities having a specific ID or name.

Operation	**Takes**	**Returns**	**Description**
`Containers`	`Filter`	`ContainerList`	Returns the containers created in the PoP
`Flavours`	`Filter`	`FlavourList`	Returns the available flavours. Flavours have no purpose at the moment
`Images`	`Filter`	`ImageList`	Returns the images available in the PoP
`Networks`	`Filter`	`NetworkList`	Returns the networks managed by the PoP

TABLE 4.1: Pop query operations

4.2.3 Container operations

The main task handled by the Pop protocol is to provide an interface capable to create, start, modify and stop containers.

Container states

A container is a stateful entity that during its lifecyle transitions between several states, as defined in Figure 4.3.

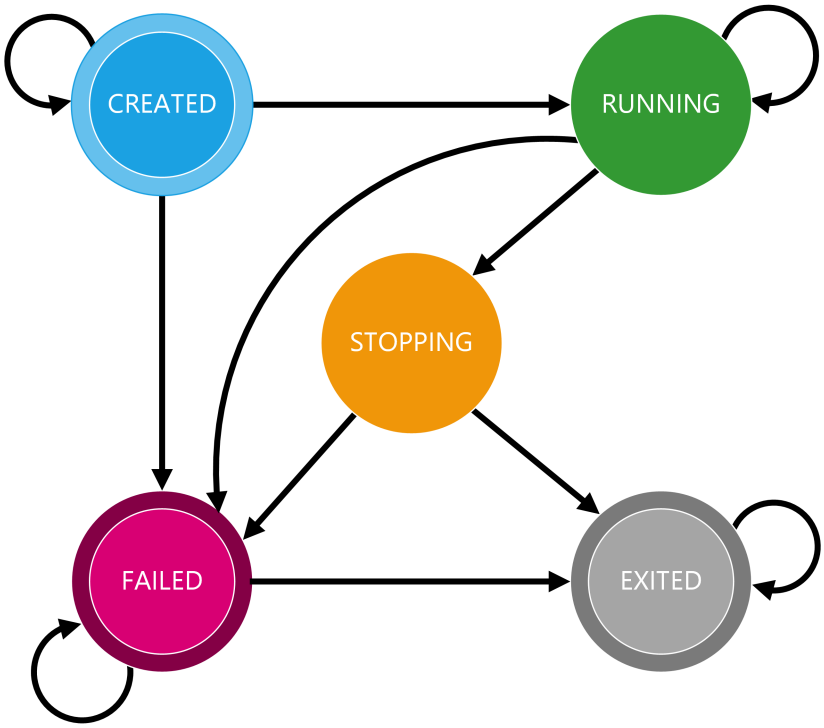

FIGURE 4.3: Possible state transitions of a Pop container (`UNAVAILABLE` is an invalid state, and it is therefore omitted).

More specifically, these are:

- **UNAVAILABLE**, an invalid state;
- **CREATED**, a state that defines a newly created, not yet started container;
- **RUNNING**, the state in which a running container resides;
- **EXITED**, a state representing a container that has been cleanly stopped by the server. Only removal is supported for stopped containers;

- **FAILED**, a state representing a container that has stopped unexpectedly or that has failed to start;
- **STOPPING**, the transient state in which a container being stopped resides.

Each of these states can only be traversed once; after reaching one of its final states (either `EXITED` or `FAILED`), the container is considered terminated and deletion is the only operation allowed on it.

Operations

The following list describes the main operations that have been defined in the protocol to operate on containers.

- `Create(ContainerConfig) returns (Container)`: creates a new container as described by the given configuration, which defines its name, its flavour, the ID of base image to be used and a map representing the Endpoints on the networks on which the newly instantiated container will be connected to. This operations does not automatically involve the creation of an actual entity inside the controlled system; a Pop server is free to use the resources underneath as it wishes within the requirements specified by the protocol. Nevertheless, the server must ensure that the returned container contains a valid, reserved address for each one of the subnets it has been connected to, either manually specified or automatically assigned;
- `Metadata(NewMetadata)`: merges the key-value string entries passed as its argument with the metadata of a given container, prioritising new values over old ones; any key paired with an empty string will cause the deletion of it from the existing map. The pairs specified by the metadata of a container are transformed in `Start` into environment variables (depending on the underlying implementation); this function is

therefore meant to be (and it is) used as a way to dynamically provide configuration to the software spawned in the underlying instances;

- `Start(Filter) returns (Container)`: starts the previously created container matched by the provided `Filter`. A successful invocation of `Start` will provide to the caller the following warranties:
  - A container will be created inside of the infrastructure, executing the software contained in the specified image;
  - The container will be reachable at the endpoints specified by its configuration, using the addresses allocated to it during `Create`;
  - The metadata set through `Metadata` requests from the moment of creation to this moment will committed and exposed to the software running in the container before its startup, as specified above;
- `Stop(Filter)`: stops a running Pop container.

  Stopping a Pop container puts it into the `STOPPING` state, and ensures the following properties:
  - The container will not be available anymore at the given endpoints;
  - The resources associated with it, such as its IPs, will be available again for allocation when a final state (either `EXITED` or `FAILED`) is reached;

  The underlying infrastructure may remove any potentially existing backing entity of the container in any moment after the invocation of `Stop`. Trying to access a `STOPPING` container is implementation defined, and may lead to unexpected results;
- `Delete(Filter)`: stops (if necessary) and deletes the container identified by the given filter. The invocation of this function only ensures

that the Pop container entity is deleted; how this reflects on the underlying infrastructure is completely left to the server implementation itself.

It is important to point out again that, as long as the requirements defined by the protocol are satisfied from the point of view of the client, the server can implement every action in any arbitrary way. This ensures the implementation complete freedom regarding how the abstraction is concretely mapped on top of the existing infrastructure.

4.3 Pop client library

The protoc-generated gRPC client stub has been used to build a client library capable of bridging together Pop servers with the Open Baton MANO framework.

FIGURE 4.4: The Pop client maps Open Baton concepts onto Pop entities.

The client realises an almost 1:1 mapping between Open Baton catalogue entities and their logical Pop equivalents, as specified in Table 4.2.

Pop (`pop`)	**Open Baton** (`catalogue`)	***Notes***
`Container`	`Server`	*Each container is mapped on an Open Baton server instance, which represents an instance available to the orchestrator for VNFC instantiation.*
`Flavour`	`DeploymentFlavour`	
`Image`	`NFVImage`	*The mapping between Pop images and the catalogue may be 1:N if a Pop image specifies multiple names. If this is the case, the client will create an NFVImage for each of the tags specified.*
`Network`	`Network`	
`Subnet`	`Subnet`	

TABLE 4.2: Pop-Open Baton entity mapping mediated by the Pop Client

The methods exposed by the `client.Client` structure closely match the operations exposed by the protocol, offering functions to query entities and to manage the container hosting a `Server`.

4.3.1 Authentication and connection pooling

`client.Client` has been designed to minimise its instantiation costs, to free the library's users from the necessity of keeping state. Making the recreation of an Pop client instance inexpensive means minimising the number of

open connections, to avoid the costs involved with re-establishing a session.

This goal is achieved through caching and sharing of the session instances between the several `client.Client`. Each one of those keeps an instance of the `creds.Credentials` structure, defined in the `creds` package, to specify and store authentication data (such as the URL of a remote Pop server instance and user credentials) to be used to either match an existing, authenticated session, or to create a new one in case none has been established yet. Before executing any request, a client must obtain a valid `session` structure from a global **session cache** instance, which handles the whole lifecycle of those objects, as shown in Figure 4.5.

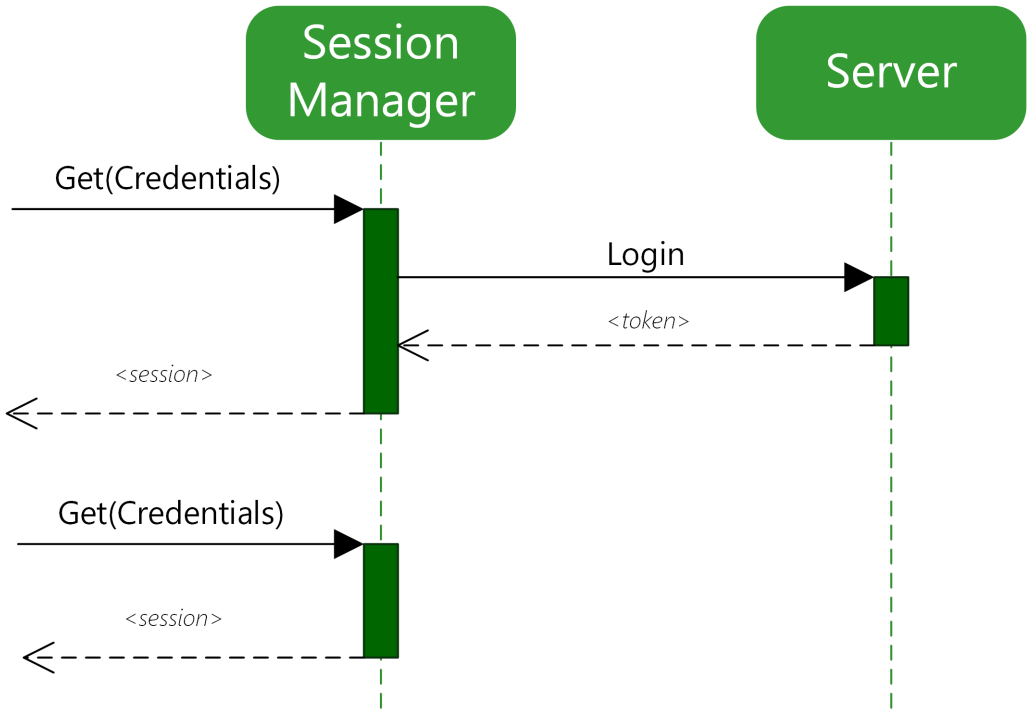

FIGURE 4.5: The session manager caches a valid connection, avoiding expensive re-connections.

A session may be terminated or destroyed in any moment by the server; to avoid the users from having to cope with this encumbrance, invalidation and

re-authentication are automatically managed by the client: a request will be automatically re-executed after a new session is successfully re-established.

4.3.2 Usage

Pop client offers a very simple, stateless interface, and can be invoked by Go code as shown in the code snippet below:

```
err := client.New(vimInstance).Delete(ctx, client.IDFilter(id))
```

This small sample represents a request to `Delete` a given container, where:

- `New` is a function that automatically extracts the credentials contained in a `catalogue.VIMInstance`, returning a `Client`;
- `ctx` is an instance of a standard Go `context.Context`. This type represents the context in which the function is executed, and can be used to specify context parameters, like deadlines;
- `client.IDFilter` returns a filter that matches a container having a given `id`.

4.3.3 CLI client

The `client` package comes with `pop`, a simple command line tool to control and administer Pop servers.

Defining commands such as `spawn` or `images`, this program makes possible to get a quick overview of the resources available on a remote point of presence, such as its Networks and Images, and administer them through operations to create and modify Servers.

```
$ export POP_AUTH="user_name:password_value"
$ pop spawn image=nginx:latest name=nginx-cont
created: Feb 21, 2017 4:28:41 PM
extId: a5aabadc-2e87-4654-a497-16f4d033cf78
extendedStatus: the container is running
flavor:
  disk: 0
  extId: docker-flavour-id
  flavour_key: docker.container
  ram: 0
  vcpus: 0
  version: 0
floatingIps: {}
hostName: ""
hypervisorHostName: ""
image:
  created: Jan 17, 2017 7:39:59 PM
  extId: sha256:
   a39777a1a4a6ec8a91c978ded905cca10e6b105ba650040e16c50b3e157272c3
  isPublic: false
  minDiskSpace: 0
  minRam: 0
  name: nginx:latest
  version: 0
instanceName: ""
ips:
  private:
  - 172.16.0.2
name: nginx-cont
status: RUNNING
version: 0
```

The listing above shows how `pop` can be used interactively to spawn a simple

Nginx server on a local Docker Pop server instance, dumping the informations about the newly instantiated `Server` as YAML for easy inspection by the operator. The `POP_AUTH` environment variable allows to specify once a username and password pair, which will be used to connect to a local Pop server instance, simplifying the user interaction with the tool when executing multiple commands.

4.4 Docker-Pop VIM implementation

Docker has been used by the prototype Pop server and its library implementation `docker-pop-server` to provide a concrete daemon implementation to the Pop protocol, using software containers to provide the necessary underlying infrastructure.

`docker-popd`, which is also written in Go, represents the Virtual Infrastructure Manager at the core of the ETSI NFV-MANO compliant container orchestration infrastructure prototype designed by this thesis. Through a complete abstraction of Docker and its peculiarities, it fully integrates with the other MANO-related Pop components, such as the Pop VIM Driver, to extend the Open Baton framework on top of this new, popular virtualisation technology.

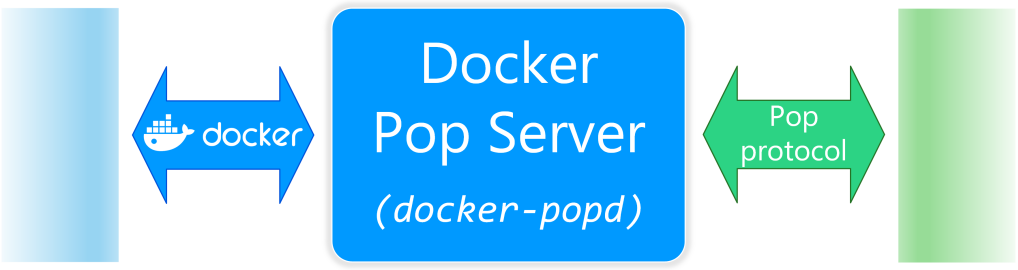

FIGURE 4.6: The Docker Pop server uses Docker to implement Pop entities.

4.4.1 Overview

The Docker Pop Server has been designed as a Go package defining a separate library, capable to be used either standalone or as an embedded component in other applications. Made of around 2000 lines of code, it can connect to either a local or a remote Docker daemon instance, using the **Docker API**, to allocate and control container instances, images and networks.
The server and the Pop protocol in general are independent from the MANO architecture, and unaware of the actual nature of the software running in the infrastructures they manage. This makes the solution suitable to be re-used and extended to fit into other non-NFV contexts, if so is desired.

4.4.2 Authentication

The Docker Pop server strives to be a valid solution to the security issues delineated in 3.4.4. By offering abstract, limited and indirect access to a Docker daemon, the target `dockerd` instance can be configured to listen to only on private, local UNIX sockets or Windows named pipes, helping to reduce the attack surface exposed by the server considerably and avoiding the risks associated with its exposure to a public TCP network.

4.4.3 Docker-Pop entity mapping

The main purpose of `docker-pop-server` is to dynamically match the entities exposed by its northbound Pop interface to satisfy the demands and requirements coming from its clients. The precise behaviour expected by these components from the Pop protocol (as described in Section 4.2) must be concretely realised by the server using suitable resources provided by the Docker infrastructure it is connected to, allowing the client to be unaware of the concrete implementation and design choices made by its remote counterpart.

The mapping between the elements is briefly introduced in the table below:

Pop concept	**Docker entity**	***Notes***
`Container`	`Container`	*This mapping is actually only valid when the Pop container is `RUNNING`, when a Docker container is spawned and associated with it.*
`Flavour`	*None*	*`docker-pop-server` ignores Flavours in the current implementation. A `docker.container` flavour with unlimited resources is provided for correctness, and every container is expected (and is automatically forced to) to use it as its Pop `Flavour`.*
`Image`	`Image`	*Images are matched 1:1 with their with Docker equivalent.*
`Network`	`Network`	*Every Pop network is matched 1:1 with a Docker equivalent.*

TABLE 4.3: Pop-Docker entity mapping mediated by the Docker Pop Server

It is important to specify that each of these mappings can be transient and of temporary nature, and the only restrictions enforced on the server are those necessary for a correct implementation of the protocol the clients expect.

4.4.4 Images

Images can be listed and filtered by the client as specified by the protocol, using the `Images` operation; these images at the moment directly match a single Docker image, manually pulled from the Docker Hub or built using a specific Dockerfile (see Section 4.5), as shown below in the output of the

`docker` and `pop` tools:

```
$ docker images alpine:latest
REPOSITORY   TAG       IMAGE ID      CREATED      SIZE
alpine       latest    88e169ea8f46  8 weeks ago  3.98 MB
$ pop images alpine:latest
created: Dec 27, 2016 7:17:25 PM
extId: sha256:88e169ea8f46<hash cut for clarity>
isPublic: false
minDiskSpace: 0
minRam: 0
name: alpine:latest
version: 0
```

Automatic pulling of images, while easy to implement, has been deemed of a lower priority than other tasks because of existing restrictions on MANO components. The Open Baton NFVO, indeed, expects a VIM image to be available and known to it before accepting the onboarding of a new NS Descriptor, making such a feature hard to integrate with the current infrastructure without deep modifications in how the orchestrator behaves.

4.4.5 Network management

All of the aspects concerning network management, such as address reservation, are internally handled by the `docker-pop-server` library. Pop `Networks` and `Subnets` are directly implemented using Docker networks configured to expose the same behaviour specified by the abstractions themselves, as shown in the next page:

```
$ docker network inspect -f "<format omitted for clarity>" private
Name=private
ID=b7c19dc2f1de131ba074f807b1975699b14f2ad0d8e8ac17ab16f2ddcafc25e5
Internal=true
Subnet=172.16.0.1/16
Gateway=172.16.0.1
$ pop networks
- extId:
   b7c19dc2f1de131ba074f807b1975699b14f2ad0d8e8ac17ab16f2ddcafc25e5
  external: false
  name: private
  shared: false
  subnets:
  - cidr: 172.16.0.0/16
    extId:
   b7c19dc2f1de131ba074f807b1975699b14f2ad0d8e8ac17ab16f2ddcafc25e5
    gatewayIp: 172.16.0.1
    name: ""
    networkId: ""
    version: 0
  version: 0
```

Each docker-pop network is limited in the current implementation to a single subnet, which can be linked to a number of containers equal to at most the amount of addresses available in the IP range associated with it.
The server is capable to automatically assign an IPv4 address to each endpoint when attached, marking the allocated IP as taken to avoid conflicts. The Docker daemon is also queried at network creation time to obtain and prereserve the addresses already associated with potentially pre-existing containers.

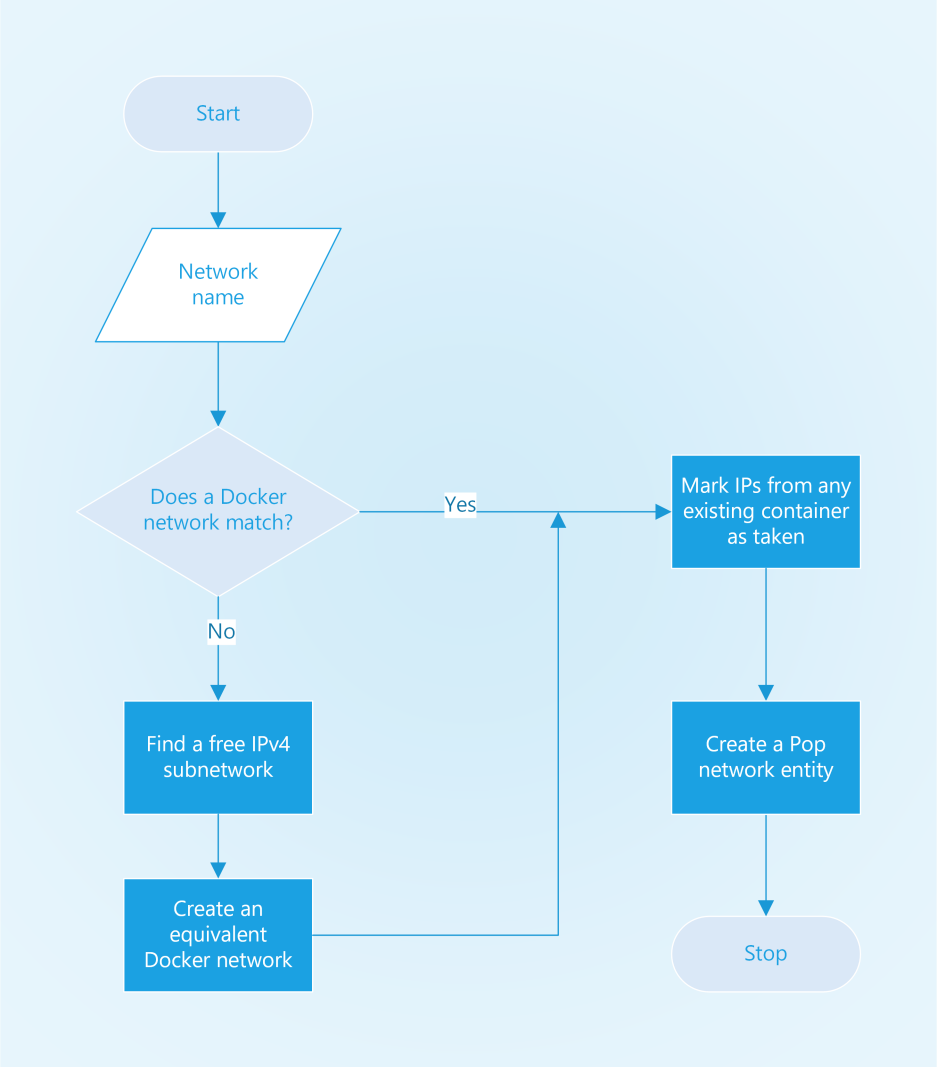

FIGURE 4.7: Flowchart describing the process behind the creation of a new private network with a dynamically associated subnet.

The current prototype supports only a single, dynamically generated private network, named `private`, which is used as the default target for every newly instantiated container. The steps operated by the server to allocate a this kind of network, as shown by the flowchart in Figure 4.7, are defined as follows:

1. The Docker daemon is queried to check if a network carrying the same name as the one under construction exists. If this test is affirmative, the instantiation steps are skipped, and the execution continues at 4;

2. A new, free subnet is sought among the available range of subnets. For sake of simplicity, the current implementation only allocates private `/16` subnets in the `172.16.0.0/12` range, and thus only allows 16 private internal subnets at any given time, without considering any pre-existing, non-Pop network that may already reside inside the Docker server. This limit is neither a design nor an architectural constraint of the server, and can be easily lifted in future releases.

3. New Docker networks and subnets are created on the current Docker instance, using the previously determined subnet and the parameters specified at the time of request;

4. The IPs of the containers already connected to the network are collected and marked as reserved;

5. A Pop `Network` instance is created, pointing at the previously defined Docker network.

The new network can then be used for either manual or automatic assignment of network addresses for newly created containers.
This Docker-independent allocation of IP addresses is fundamental when considering the dependency issues specified in 3.5.3: the server can associate a unique, valid address to a Pop container before it is started on Docker, allowing the NFVO to correctly fill the dependency parameters of newly instantiated VNFCs Instances.

4.4.6 Container management

The primary task of the Docker Pop Server is to manage and handle the mapping state of the Pop and Docker containers under its control.

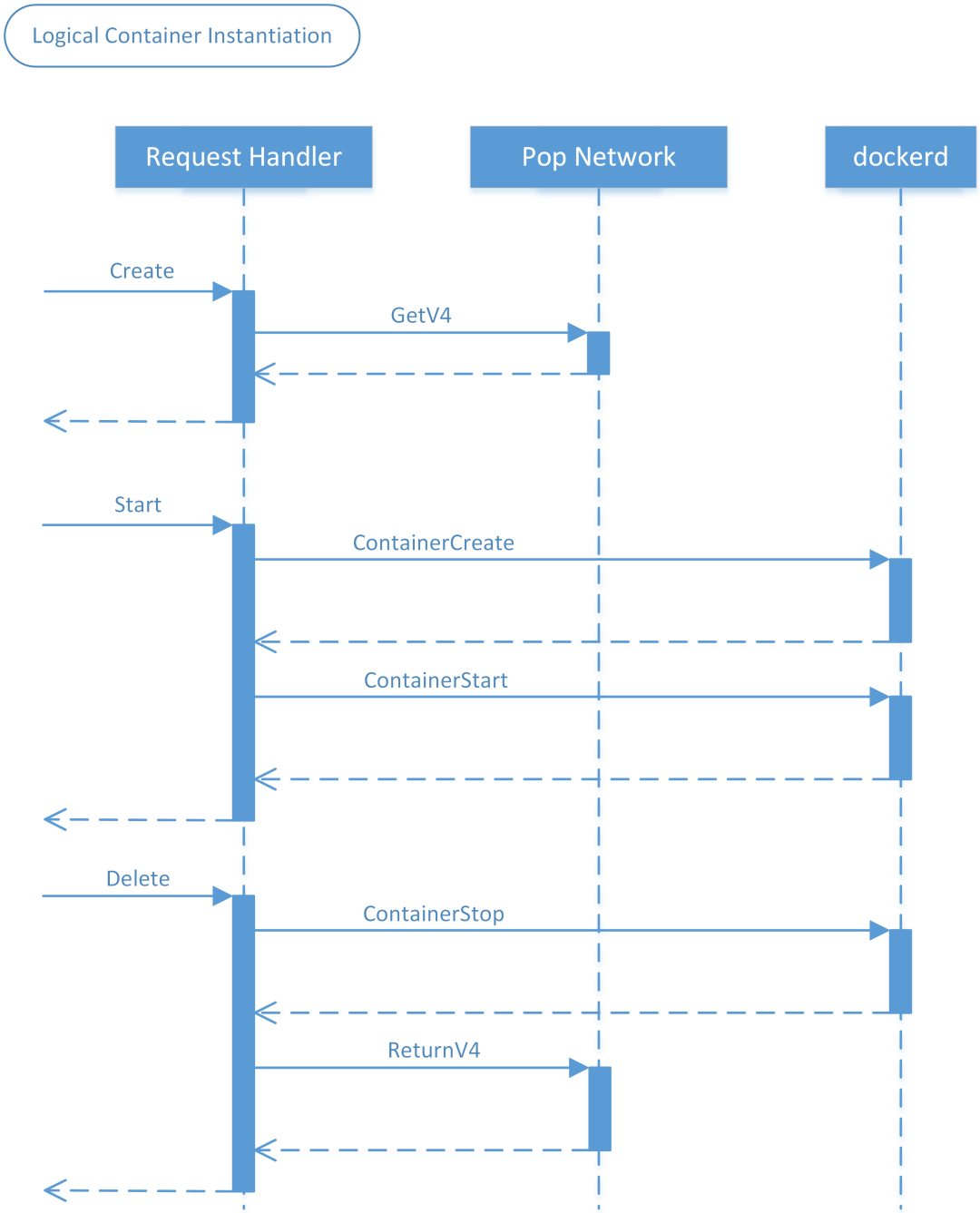

FIGURE 4.8: Full instantiation and termination sequence of a container with no pre-specified endpoints, complete with Docker API calls.

The server provides several operations to alter and modify the state of the containers, allowing a client to completely administer the virtual instances under the control of the VIM.

Container creation

After receiving a `ContainerConfig` structure at the beginning of a `Create` request and validating the parameters it specifies, the server creates a new Pop container entity inside of the its internal registry (multiple, concurrent accesses to this structure are protected by mutal exclusion locks). Any network endpoint specified by the received configuration is allocated on its corresponding network, dynamically assigning any unspecified IP. No action or operation is executed on the Docker server during or after a `Create` request.

From this point afterwards, querying the server with `Containers` requests will include the newly instantiated container, which will be in a `CREATED` state.

Metadata updates

As much metadata as necessary can be added to a container in the `CREATED` status, as specified by the Pop protocol in 4.2.3. Altering the metadata of a running container is not allowed and will cause an error.

Container start

After a `Start` request from a client is received, the server checks for the existence of the container specified by provided Filter. If this exists and its current status is `CREATED`, the following actions are executed:

1. A lock is obtained on the mutual exclusion lock associated with the Pop container;

2. A new Docker container is created, using the parameters specified by the Pop one, such as the image to use and pre-allocated IPs; the Metadata is converted into an array of shell variables and passed to the container as part of its environment. If this operations do not succeed, the container enters the `FAILED` state and an error is returned;

3. The Docker container created above is started. If this operation does not succeed, the container enters the `FAILED` state and an error is returned. Because no command line parameter has been provided to the `ContainerCreate` previously request sent to `dockerd`, the newly issued `ContainerStart` request immediately executes the `ENTRYPOINT` or `CMD` specified in the container's defining image;

4. The record entry associated with the Pop container is updated with the ID of the newly instantiated Docker container;

5. The status of the Pop container is updated to `RUNNING`.

`RUNNING` is the only state in which a concrete association between a Pop and a Docker container actually subsists.

Status checking

During its instantiation, the server takes care of spawning a background routine to periodically poll the status of the containers running in the Docker daemon it is associated with. If the Docker instance associated with a running Pop container has exited for any reason, this background task will modify the current state to reflect the unexpected state transition, changing it to either `EXITED` or `FAILED` according to what reported by the infrastructure.

Container stop

Calling `Stop` on a `RUNNING` Pop container will cause the system to begin the necessary steps to terminate the execution of its underlying associated Docker container:

1. The system determines the timeout time after which the Docker container will be forcefully stopped. This is either set according to the deadline specified by the current context, or to 5 seconds;
2. The state of the Pop container is changed to `STOPPING`;
3. A *goroutine* is spawned to stop the Docker container in the background, and a successful response is returned to the caller;
4. The container will persist in the `STOPPING` state until its complete removal. No action should be (or should be allowed to be) performed on it;
5. The IP addresses owned by the container are reverted to their respective Networks;
6. Finally, the `ContainerStop` request either succeeds or fails, and the status of the Pop container is updated to reflect this final result.

After being stopped, a Pop container has terminated its lifecycle as either an `EXITED` or `FAILED` container and can only be deleted from the system. By default, its backing Docker counterpart is deleted after stop, and the association between the two is rescinded.

Container deletion

Invoking `Delete` on a Pop container causes it to be stopped (if running) and removed by the system, independently from its current state.

The pre-deletion stopping of containers operated by `Delete` is identical to the case described above for `Stop`. After the function returns, no trace or history of the container is retained by the server.

4.4.7 Usage

The main type defined by the `docker-pop-server` library is represented by the `server.Server` structure, which defines a fully capable Pop server.

```
user, err := server.NewUser(uname, pass)
if err != nil {
  panic(err)
}

cfg = server.Config{
  PopName: "test-pop",
  Netaddr: laddr,
  Users: server.Users{
    user.Name: user,
  },
  LogLevel:   log.ErrorLevel,
  AutoRemove: true,
}

srv := &server.Server{Config: cfg, Logger: log.StandardLogger()}

go func() {
  if err := srv.Serve(); err != nil {
    log.WithError(err).Fatal("Serve failed")
  }
}()
```

The sample above is an actual snippet taken from the library's Go test files. The code, after creating a new pair of credentials using the `server.NewUser` function, instantiates and executes a new `Server`, configured with test parameters. Because no Docker host has been specified in the `server.Config`, the platform default address will be used (a local named pipe on Windows, and a local UNIX socket on UNIX).

This server will start several routines to accept incoming connections and operate background management routines, and will establish a valid connection with Docker. A default private network called `private` will also be created if not present. `server.(*Server).Serve` will block until until either the program is killed, or `server.(*Server).Close` is invoked. In the current implementation, terminating a server instance means the permanent loss of the current status of any entity launched by it.
Because of its inherent complexity, adding the support for persistent storage to the server has been deemed out of the scope of this first prototype, and has been postponed to a future development.

4.4.8 Docker-Pop Daemon

`docker-popd` is a Go daemon that uses the aforementioned library to implement a simple and easy to use standalone Pop Docker server.

`docker-popd` can either configure itself using a TOML configuration file (that can be generated using `docker-popd init`), or resort to internally defined default parameters that specify a default port on a local address. In the latter case, the `POPD_AUTH` environment variable is used to provide the server a single pair of credentials.

```
$ export POPD_AUTH="user_name:password_value"
$ docker-popd -verbose
INFO[0000] starting server pop-name=docker-popd tag="github.com/
   mcilloni/openbaton-docker/docker-pop-server.(*Server).Serve"
DEBU[0000] creating route service tag="github.com/mcilloni/
   openbaton-docker/docker-pop-server.newService"
DEBU[0000] checking Docker daemon tag="github.com/mcilloni/
   openbaton-docker/docker-pop-server.(*service).checkDocker"
DEBU[0000] creating private network if not present... tag="github.
   com/mcilloni/openbaton-docker/docker-pop-server.newService"
DEBU[0000] obtained default private network net-name=private net-
   subnet=172.16.0.0/16 tag="github.com/mcilloni/openbaton-docker/
   docker-pop-server.newService"
DEBU[0000] refresh loop spawned tag="github.com/mcilloni/openbaton-
   docker/docker-pop-server.(*service).refreshLoop"
INFO[0000] launching gRPC server pop-name=docker-popd tag="github.
   com/mcilloni/openbaton-docker/docker-pop-server.(*Server).Serve"
```

The `-verbose` parameter can be used (as in the case above) to turn on verbose logging of debug messages from the server.

4.5 Docker NFV images

The Docker-based Pop support infrastructure thoroughly introduced above would not be useful without also providing a definition of the behaviour and characteristics expected by Docker NFV images.

4.5.1 Requirements

An Open Baton Docker NFV image must be defined and designed following specific conventions, which ensure it to correctly work and integrate with the infrastructure:

- The image must be defined by a **Dockerfile**, which must either define a single `ENTRYPOINT` or a single `CMD` command;
- The image must conform to the OCI runtime workflow [13], and specifically its entrypoint should be able to be be executed without taking any command line argument;
- The image must be self contained, and must provide all of the necessary software and scripts to accomplish its primary task;
- After being published, the image creator must clearly state which configuration variables and dependencies a VNFD making use of it must specify. These will be exported to the Docker container as environment variables.

The image designer may use pre-existing and already available Docker images from a Registry as base images for a new NFV image.

4.5.2 Implementing sample SIPp client-server images

This section will illustrate a sample pair of Docker images implementing a simple client-server `SIPp` service, to better explain the required methodologies and the specifications expected from an Open Baton NFV Docker image.

Overview

As mentioned above, the provided SIPp VNF service is defined by two main components, respectively:

- A SIPp server, listening on an internal private network for incoming requests;
- One or more SIPp clients, which need to be matched with and connect to a SIPp server to accomplish their tasks.

Each one of these entities will require its own Docker image, with a unique name suitable to be inserted into an appropriate VNFD. **Alpine**, a non-GNU minimal Linux distribution based on the Musl libc and Busybox, has been chosen as the base image.

SIPp image

SIPp ships both its client and server as a single `sipp` executable, provided by Alpine in the `sipp` package. To avoid repeating package installation tasks twice, an intermediate SIPp image has been defined, as specified by the following `Dockerfile`:

```
FROM alpine:latest

RUN apk update && apk add sipp tmux && rm /var/cache/apk/*

ENTRYPOINT ["sipp"]
```

The instructions above extend the Docker-provided `alpine` image with the `sipp` and `tmux` packages, deleting the unnecessary package caches afterwards. The `tmux` command has been used as a simple way to get the output of the `sipp` command from a running session.

SIPp server

The base image generated before was then used to generate two images, representing different server and client behaviours.
The former has been defined using the `Dockerfile` shown below, which extends the previously built image with a `server_start.sh` script, to handle the startup of the SIPp server.

```
FROM mcilloni/sipp

COPY server_start.sh /opt/server_start.sh

EXPOSE 5060 5061 6000 8888

ENTRYPOINT ["sh", "/opt/server_start.sh"]
```

The given `ENTRYPOINT` will cause a container instance to automatically execute the script at startup, containing the following instructions:

```
#!/usr/bin/env sh

tmux new -d -s server-sess "sipp -sn uas -trace_msg; tmux wait-for
   -S server-end" \; wait-for server-end
```

The `sipp` server requires no parameters, and will start listening for incoming connections from other containers; an operator can attach to the spawned `tmux` session to view the output of the running instance.

SIPp client

The SIPp NFV client image is defined using a Dockerfile and a startup script, following the same fashion as the case above.

```
FROM mcilloni/sipp

COPY client_start.sh /opt/client_start.sh

ENTRYPOINT ["sh", "/opt/client_start.sh"]
```

client_start.sh:

```
#!/usr/bin/env sh


tmux new -d -s client-sess "sipp -sn uac $SERVER_PRIVATE -d
   $SIPP_LENGTH -r $SIPP_RATE -rp $SIPP_RATE_PERIOD -rate_increase
   $SIPP_RATE_INCREASE -fd $SIPP_RATE_INCREASE -rate_max
   $SIPP_RATE_MAX -rtp_echo -t $SIPP_TRANSPORT_MODE -trace_msg -
   trace_screen -trace_err -trace_rtt -trace_logs -trace_msg; tmux
   wait-for -S client-end" \; wait-for client-end
```

Unlike the server, the SIPp client instance requires several configuration parameters, both provided by the VNFD and by its server dependency, to correctly operate and detect its counterpart.
Every configuration parameter is defined by the infrastructure through the environment variables specified to the container before its instantiation.

A particular mention must be given to the SERVER_PRIVATE parameter, a configuration parameter containing the address of the remote *server* on the Virtual Link *private*. This is dynamically resolved and filled by the management services provided by the NFVO and the other MANO components, to satisfy the dependency between a client and the server.

4.6 MANO components

The second part of the implementation work has revolved around the realisation of two Open Baton MANO components, each one necessary to integrate Pop into the existing NFV orchestration framework. The final end result of this integration process has led to the realisation to a functioning and easily extensible prototype of a system capable of using containers to deploy and manage VNF instances.

Overview

The MANO components implemented by this thesis are the following:

- `mgmt-gvnfmd`, a Generic VNFM that controls its VNFs using a custom management protocol, **mgmt**, instead of an EMS;
- `pop-plugind`, a VIM Driver plugin for the Open Baton NFVO, capable of statelessly leveraging and connecting with multiple Pop instances, using the Pop client library. `pop-plugind` has also been chosen as the component tasked with offering `mgmt` server support.

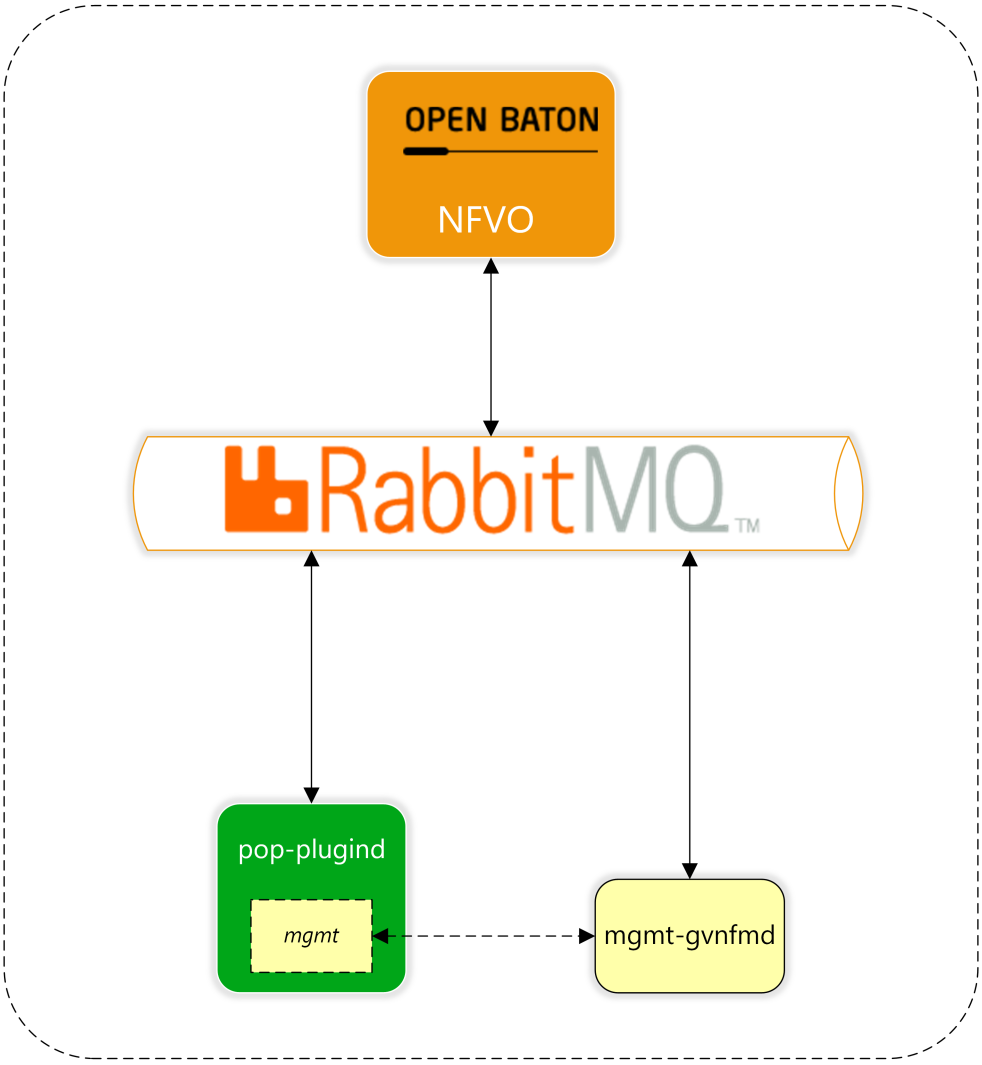

FIGURE 4.9: The new MANO components and their interconnections.

Every element described above has been designed to be reusable, stateless, and simple to further extend with additional features if necessary. Explicitly defining every interaction and protocol used by the VIM, the driver and the VNFM allows a great degree of independence between them, easing the development of new compatible software components.

4.6.1 VIM Driver

The Pop VIM Driver has been implemented as the `pop-plugind` plugin daemon, using the `plugin` framework provided by the Go Open Baton libraries. This component uses the AMQP protocol to communicate through a **RabbitMQ** message queue with the Open Baton NFV orchestrator, implementing the required plugin interface as described in 3.5.1.

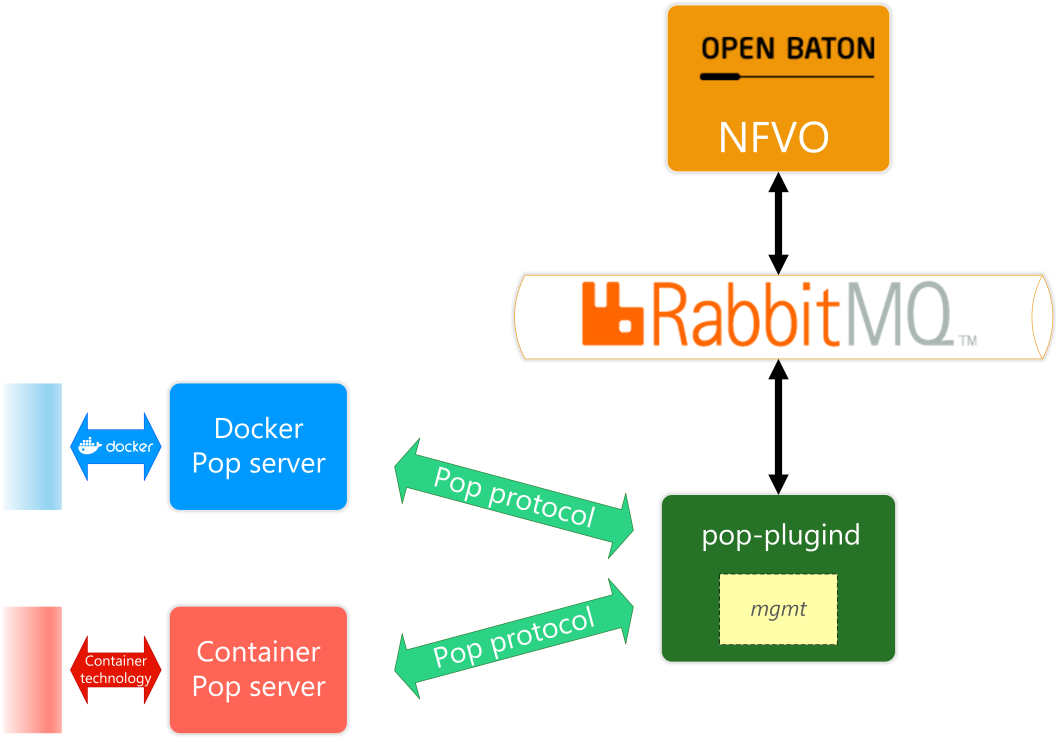

FIGURE 4.10: The Pop plugin VIM driver connects the Open Baton NFVO with Pop VIM instances.

The plugin is an almost stateless entity, that uses the **Pop client** library to efficiently connect to multiple Pop servers of any arbitrary nature. Using the credentials provided by a `catalogue.VIMInstance` structure, every request is efficiently handled by the client, offering a complete transparency regarding the connection with the server itself. The current implementation of the plugin implements the necessary methods to launch and terminate containers, plus the operations needed by the NFVO to query the entities contained by the current VIM, i.e. `Images`, `Networks`, `Flavours` and `Servers`.

While this plugin is usually used together with the `mgmt-gvnfm` Generic VNF Manager, it does not depend on it in any way, and it can be reused by other VNFMs to allocate and handle resources from Pop VIMs as they deem necessary.

4.6.2 Management Protocol

mgmt, as in **management**, is a very simple RPC protocol meant to be used to expose a small subset of the VNF management operations available on a VIM to a VNFM, using an AMQP-compliant message queue like RabbitMQ.

The protocol is composed of two main components:

- The **VIM Connector**, the client entity of the protocol, which acts as a consumer of a remote, AMQP accessible VIM Manager. The VNFM uses the Connector to operate on a remote VIM instance, executing the management operations it needs to carry out on a given VNF;
- The **VIM Manager**, which receives and executes management requests from the Connector, relaying them to the VIM it is uniquely associated with.

A Manager and its associated VIM are reachable on an AMQP queue defined by protocol as `vim-mgmt-<VIM ID>`, where <*VIM ID*> stands for the unique identifier of the VIM instance. This easily derivable queue name allows the VNFM to quickly identify the right queue using only the informations received along the VNF Record of a given Virtual Network Function.

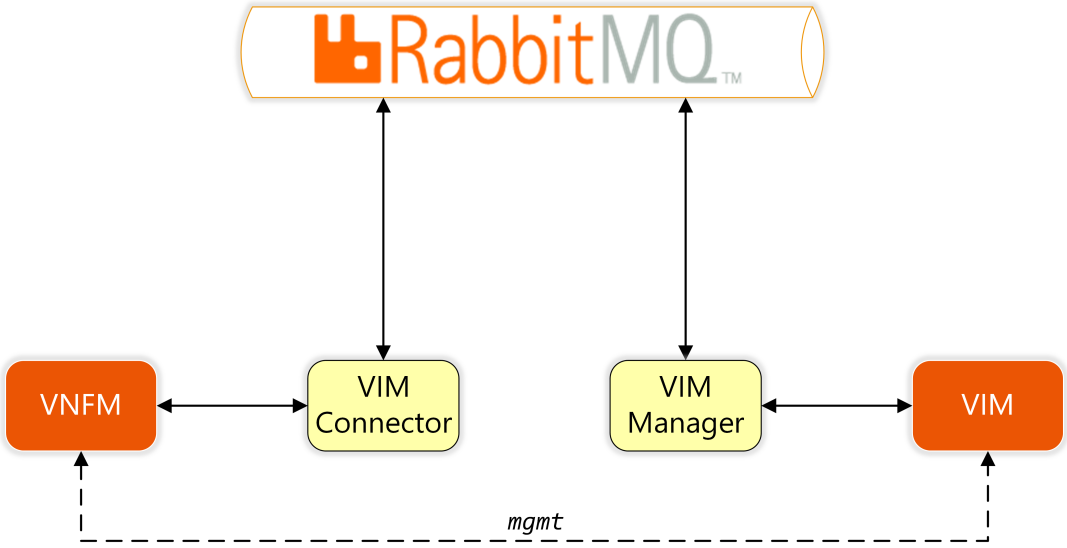

FIGURE 4.11: The Management protocol can be used to indirectly expose a VIM to a VNFM, using an AMQP queue as a bridge.

Rationale

One important goal of this thesis is to create simple, isolated components that can be reused as much as possible, using well defined protocols and abstractions. `mgmt` provides both, abstracting a VIM even further, and giving back to the VNFM the genericness lost together with the Element Management System.

Using `mgmt`, a VNFM can achieve total statelessness, eliminating any necessity to store credentials, history, or state of any sort, requiring only an AMQP queue and a remote peer; the VIM is also further protected, by exposing a very simple and restricted subset of its capabilities.

Operations

Each `mgmt Manager` allows a `Connector` to invoke a small set of operations, as described below:

- `Check(id)`: controls the availability of a VNFC instance represented by the given identifier string, returning a `Server` structure containing its properties on success;
- `Start(id)`: instructs the VIM to start a VNFC instance identified by the given identifier string;
- `AddMetadata(id, metadata)`: merges the given `metadata` with the entries already residing on the VNFC instance, overwriting any conflicting key with the newly specified values. An empty metadata key will delete its entry from the existing set.

The protocol and its implementation are designed to be easy to deploy on top of the current infrastructure, providing a simple, easy to extend and platform agnostic VIM connector.

Implementation

An implementation of `mgmt` has been developed for this thesis using the Go programming language. This library has been designed to be easy to extend, offering abstractions and genericness through the usage of interfaces when possible.

Both the `VIMManager` and `VIMConnector` provide means to piggyback on an existing AMQP Connection, providing zero-configuration setup for any MANO component already connected to the common RabbitMQ message queue.

4.6.3 Plugin management integration

The `pop-plugind` daemon provides, along with its Pop VIM Driver, a Pop-based implementation of an `mgmt Manager`.

Each time the Driver learns about a new Pop instance, a corresponding Manager is dynamically launched in the background to receive management commands from a VNFM.

This approach offers several important advantages, that help decreasing the coupling between the various architectural components:

- Only the VIM Driver keeps a direct connection with the VIM itself. This means that every request will be forced to pass through the plugin, avoiding the need of keeping sensible data (such as sessions and credentials) on multiple entities;
- A Pop VIM can completely ignore the existence of either MANO and the AMQP queue used by the Open Baton components. This avoids unnecessary coupling of the involved entities and a duplication of APIs;
- The VNFM can rely on just `mgmt` to carry out its tasks, offering generic VNF management unencumbered from the actual implementation of the VIM itself.

4.6.4 VNFM

`mgmt-gvnfmd` is a daemon that implements a Generic VNF Manager, capable to manage VNFs and VNFC instances using the `mgmt` library. Written in Go using the Go Open Baton libraries, the VNFM uses AMQP to administer a remote VIM using a VIM Manager entity connected to the message queue, without any knowledge about its concrete implementation.

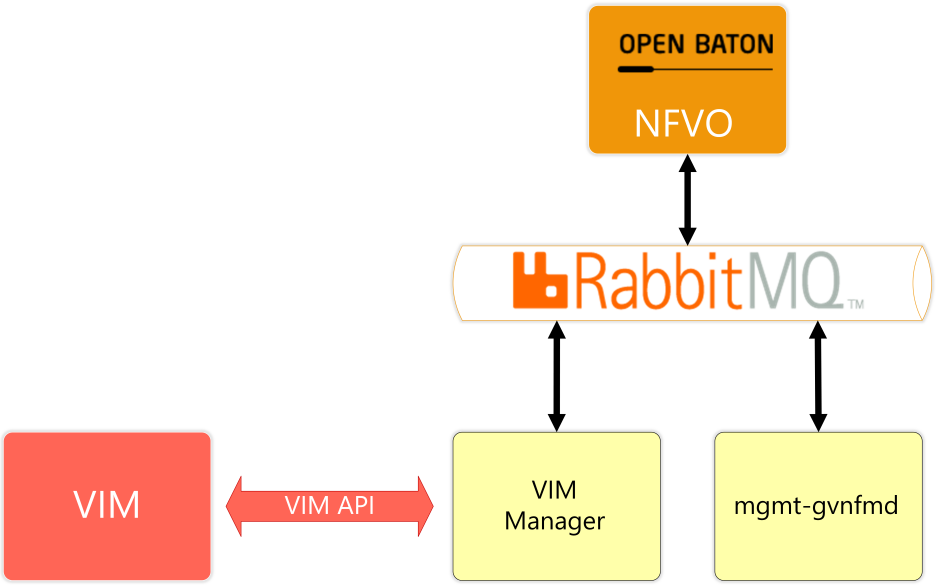

FIGURE 4.12: `mgmt-gvnfmd` manages VNFs through a `mgmt VIM Manager` connected on the RabbitMQ AMQP message bus.

The framework offered by the Go packages already implements for the most part a complete VNFM, leaving to the developer only the task to fill out the callbacks necessary to correctly handle the events issued by the NFVO.

Instantiate

The `Instantiate` callback provided by the Generic mgmt VNFM is invoked after the NFVO has finished to allocate correctly a VNF and all of its components. The VNFM uses the `Check(id)` method provided by the `mgmt Connector` to connect to the VIM and ensure that every VNFC instance belonging to the current VDUs has been correctly instantiated and is reachable through `mgmt`.

Modify

The `Modify` callback is invoked when the `MODIFY` event is received by the NFVO, with the purpose to modify and update the configuration settings

stored in a VNF.

The configuration data sources loaded by the manager during `Modify` are:

- The `configurationParameters` specified by the `provides` section of the current VNF Record;
- The `configurationParameters` specified by the `configurations` section of the current VNF Record;
- The `parameters` and the `vnfcParameters` specified by the `VNFDependency` instance sent by the NFVO along with the event. This structure represents the dependencies of the current VNF, together with the parameters the NFVO has chosen as suitable to resolve them;

The VNFM sanitises and merges all of these values into a new metadata map instance, which is then pushed to the VNFCs using the `mgmt AddMetadata` remote function.

Start

The `Start` callback is invoked when the NFVO needs the manager to start a VNF. The Generic mgmt VNF manager uses the `Start` operation provided by the `mgmt VIM Connector` to signal the Manager to start the operations necessary for the VNF to become active and operational.

Scale

The `Scale` callback is invoked when a `SCALING` event, either **out** or **in**, regarding one of the VNFs under control of the manager is issued by the NFVO.
Scaling out is the most relevant of the two possible cases, because the VNFM will be required to execute all of the instantiation, configuration and starting

operations described above to fully launch a new, functional VNFC instance in the current VDU.

Both scaling in and out of VNFs are fully supported by the `mgmt-gvnfm`.

4.7 Interaction of Open Baton components after extension

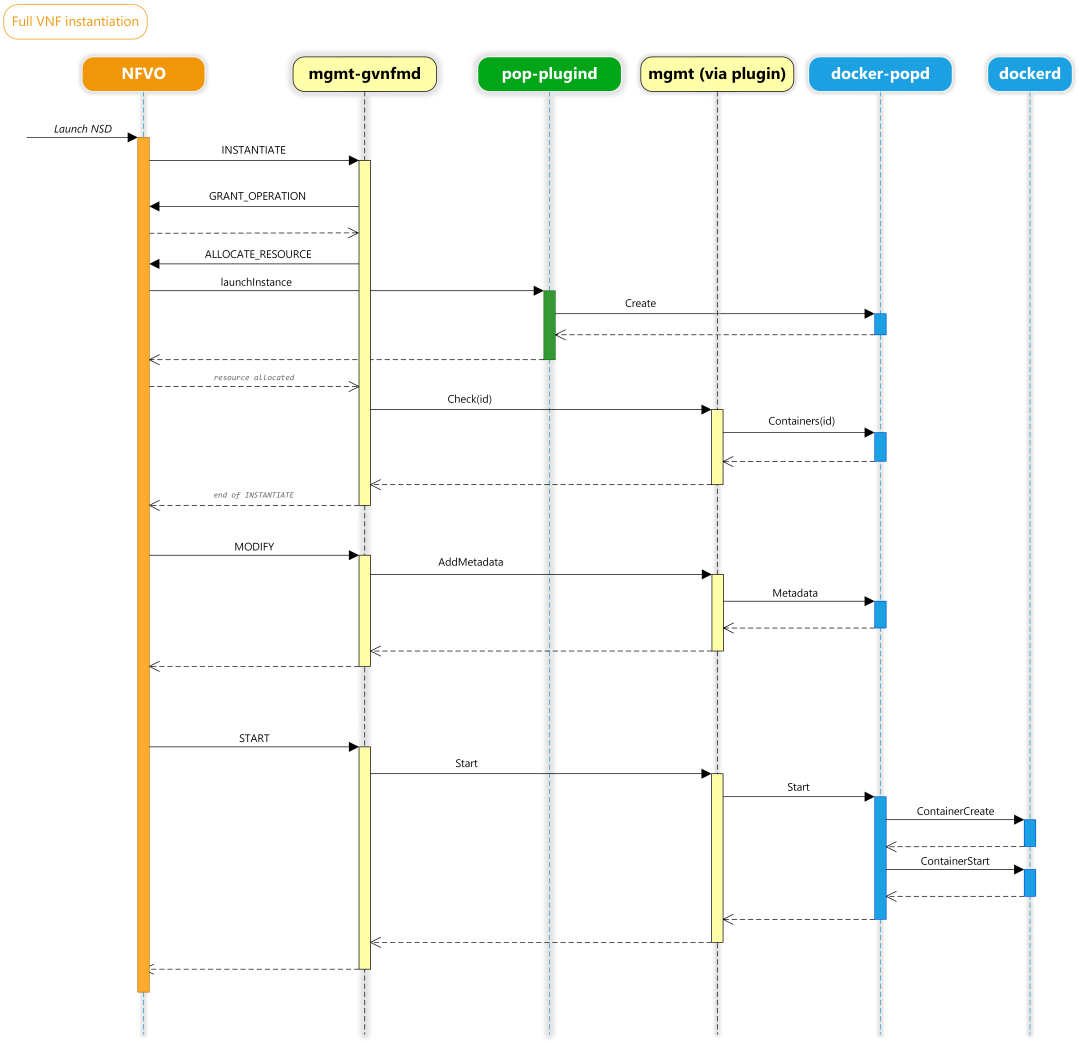

FIGURE 4.13: How the components interact to accomplish a `Launch NSD` operation.
Orange, pale yellow, green and turquoise respectively represent Open Baton, `mgmt`, Pop and Docker components.

The components implemented in this chapter can finally be deployed together, concretely realising the complete Open Baton based NFV infrastructure described by Figure 4.1.
This system implements the required prototype, capable to deploy VNFs using Docker images defined as in Section 4.5.

The sequence diagram in Figure 4.13 offers a complete overview on the system, including the interactions between the components on a deployed infrastructure, by representing the actions carried out in response of a received `Launch NSD` request:

1. The **NFVO** receives a request to launch the service described by a given NSD;

2. For each VNFD defined by the NSD, the orchestrator finds a suitable **VNFM**. In this case, the Generic mgmt VNF Manager is chosen as the endpoint to handle the VNF;

3. The `INSTANTIATE` request is received by `mgmt-gvnfmd` through the AMQP message bus;

4. The Go Open Baton VNFM framework used by the VNFM grants the request to the NFVO, sending a `GRANT_OPERATION` message;

5. The VNFM framework requires from the NFVO the allocation of the necessary VNFC instances;

6. The NFVO seeks the required VIM, and finds the appropriate VIM driver to operate on it. In this case, the `pop-plugind` plugin;

7. The NFVO sends a `launchInstance` request to the **Pop VIM Driver** plugin;

8. The plugin received a new instantiation request. If no `mgmt Manager` instance is active for the Pop instance specified by the VIM instance descriptor sent with the invocation, a new one is spawned;

9. The plugin uses the Pop client library to reach out the requested Pop server, using the informations specified by the VIM instance structure received;

10. The plugin issues a `Create` Pop request to the server. The client carries out the necessary connection and authentication steps, and then sends the request;

11. The `docker-popd` instance receives the request, and creates a container instance in its internal registry, allocating the required IP addresses and endpoints if necessary. A descriptor representing the created Pop container is then sent back to the client;

12. The Client converts the received Pop container descriptor into a `catalogue.Server` instance, that is then returned by the plugin back to the NFVO;

13. After instantiating all of the necessary `Servers` and resolving the necessary parameters, the NFVO can reply to the `ALLOCATE_RESOURCE` request previously received from the `mgmt-gvnfmd`;

The VNFM is now sure that that the required resources have been correctly allocated by the NFVO. The next logical steps involve ensuring that the VNFM can reach and manage the newly instantiated servers through its `mgmt` connection:

14. The VNFM sends a `Check(id)` request to the `Manager` spawned previously by `pop-plugind`, identified by the ID of the VIM instance specified in the VNF Record;

15. The Manager uses the Pop connection of the plugin to query the `docker-popd` instance it is associated with, sending a `Containers(id)` request;
16. The VIM returns the informations related to the requested container;
17. The VNFM receives a `catalogue.Server` instance from the Manager, and the functionality of the `mgmt` connection is validated;
18. The VNFM returns the final VNF Record to the NFVO, concluding the `INSTANTIATE` request.

The VNFs have been correctly instantiated but no configuration, including the variables needed to satisfy their runtime dependencies, has been applied to them yet. To satisfy this requirement, the NFVO may issue a `MODIFY` event:

19. The NFVO sends a `MODIFY` request to the VNF Manager, specifying the VNFR that needs to be modified with the new configuration entries;
20. The `Modify` callback of the `mgmt-gvnfmd` is invoked, receiving the VNFR of the VNF to be updated and a VNF Dependency object;
21. The VNFR fills a new metadata instance, as specified in the precedent section;
22. The VNFR uses its `mgmt` link to instruct the Manager to update the metadata associated with a given VNF, through an `AddMetadata` request;
23. The Manager issues a `Metadata` request to the Pop server, which then merges the received metadata with the existing map associated with the container;
24. The call stack is traversed backwards, concluding the operation started by the NFVO.

The VNFs are now ready to be executed; after receiving confirmation from the VNFM that `MODIFY` has succeeded, the NFVO can issue a `START` event back to the manager, in the order specified by their dependencies (by default):

25. The NFVO sends a `START` event back to `mgmt-gvnfmd`;
26. For each VNFC instance, the VNFM issues a `Start` command to the `mgmt Manager`;
27. The Manager invokes a Pop `Start` operation on the correct Pop instance (`docker-popd` in this case);
28. `docker-popd` contacts its Docker daemon (`dockerd`), issuing a new `ContainerCreate` request through the Docker API. The newly instantiated Docker container will be created with the requested environment variables and IP addresses;
29. The VIM issues a `ContainerStart` request, to start the Docker container. The `ENTRYPOINT` or `CMD` defined by the Dockerfile is executed, and the VNF is now up;
30. The Pop container enters the `RUNNING` state, and a positive response is returned to the Pop client;
31. The Manager notifies back the VNFM of the successful VNFC startup;
32. After all of the VNFC instances have been successfully started, the VNFM also notifies the NFVO of the success of the `START` request;
33. The NFVO receives an affirmative response for all of the VNFs composing the NS, and the NS Record (NSR) associated with the NSD becomes `ACTIVE`.

Chapter 5

Validation

5.1 Overview

The implemented prototype has been tested for feasibility in a sample SIPp client-server case, to ensure the conceptual validity of the final design. Further measurements have also been made to understand the impact on performances caused by the components and the deployed containers.

The tests have been carried using a standard x86-64 machine with the following characteristics:

- Intel i5-7200U CPU
- 8 GiB of DDR4 RAM at 2400Mhz
- Standard Toshiba 5400RPM Hard Drive

The operating system loaded onto the machine is Windows 10, build 14393, running the latest version of Docker (`17.03.0-ce` at the time of writing) on an Linux Hyper-V virtual machine with 2 GiB of RAM reserved.

The Go components have been compiled using Go 1.8.0 for the Windows-amd64 target, and executed directly on the host Windows system running in the Root partition.

5.2 System setup

An instance of `docker-popd` has been set-up and configured to connect to the Docker daemon running on the Linux VM mentioned above, to act as its uniquely associated VIM.

The Open Baton NFVO has been then launched inside the same Docker instance, using a Docker container provided by the project itself which also ships the RabbitMQ instance that will be used by the deployed MANO components to intercommunicate with the NFVO and each other. To allow the orchestrator to correctly interface with the Docker Pop server, an instance of the `pop-plugind` Pop VIM Driver has also been attached to the message bus, and registered with the it as a Point of Presence through the Dashboard.

Finally, the `mgmt-gvnfmd` VNF Manager has also been started and attached to the RabbitMQ instance, completing the deployment of the system.

5.3 Testing a sample SIPp NS case

The validation of the system has involved a full test of its functionality and feasibility, to ensure the correctness of the developed prototype. For the sake of this verification, the execution of a simple and complete test has been deemed necessary, using the SIPp Open Source test tool and traffic generator for the SIP protocol. [33]

The identified SIPp case solution has been chosen for several reasons, including its simplicity and completeness. Having an intrinsic dependency, it is effective to show and validate the correct functioning of the dependency resolving capabilities of the orchestration solution.

The scenario is composed by two different uses of the SIPp software: a **server**, which launches a daemon to receive and test the functionality of a SIP user agent, and a **client**, which acts as a client of the aforementioned server. The two need to be correctly configured by the orchestration system, using the configuration parameters specified by the service descriptor; the functional dependency of the client with regards to a server also needs to be satisfied, to allow SIPp to correctly find its counterpart.

Network Service Descriptor

Two SIPp client and server images, as defined at 4.5.2, have been built and imported into the running Docker Engine instance, using the `docker build` command. Afterwards, an NSD describing a service has been written to represent the images as two VNFDs, which are, respectively:

- A `sipp-server`, which uses SIPp as a daemon capable to receive requests from an arbitrary number of clients;
- A `sipp-client`, which depends on a `sipp-server` instance to correctly connect its SIPp client to a server. This dependency is expressed by the descriptor itself as a VNF dependency, which should be resolved by the NFVO and configured through `Metadata` and environment variables.

The defined NSD is also useful to test the scalability functionalities of the orchestration solution prototype, thanks to the client VNFD specifying a `scale_in_out` value of 5. This allows the NFVO to instantiate more VNFC instances to the

client VNF, up to a maximum of five. The complete listing of the used JSON Network Service Descriptor is provided for completeness in Appendix C.

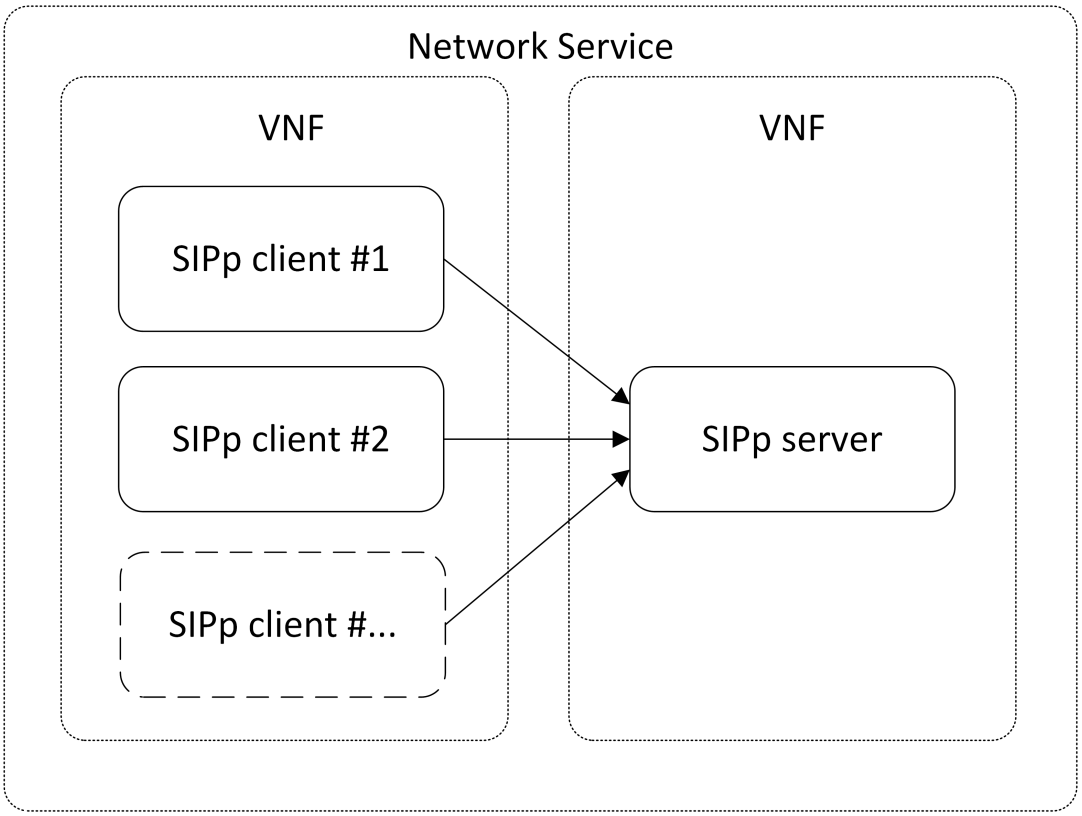

FIGURE 5.1: How the sample SIPp service is seen by the MANO components, representing an NFV point of view. The dashed box represents potentially scalable client instances.

This NSD has been directly ported from the already existing SIPp-private OpenStack NSD provided by the Open Baton project. This descriptor provides several configuration variables for the client, that will be straightly exported to the underlying container as environment variables, ready to be passed by the `client_start.sh` script to SIPp. Noteworthy is the dependency between the VNFs specified by the service descriptor, as shown in Figure 5.2.

```
"vnf_dependency":[
    {
        "source":{
            "name":"sipp-server"
        },
        "target":{
            "name":"sipp-client"
        },
        "parameters":[
            "private"
        ]
    }
]
```

FIGURE 5.2: The dependency between the `sipp-client` and `sipp-server` VNFs defined by the service descriptor, through the `private` Virtual Link.

Execution

After adding the SIPp Network Service Descriptor to the Open Baton Catalogue, the Network Service has been deployed to the set-up infrastructure through the NFVO API (using the provided Dashboard). As described in 4.7, this process involves the interaction between the components to correctly set up and initialise the two initial VNFC instances of the system, creating inside of the Docker Engine instance the configuration described in figure 5.3.

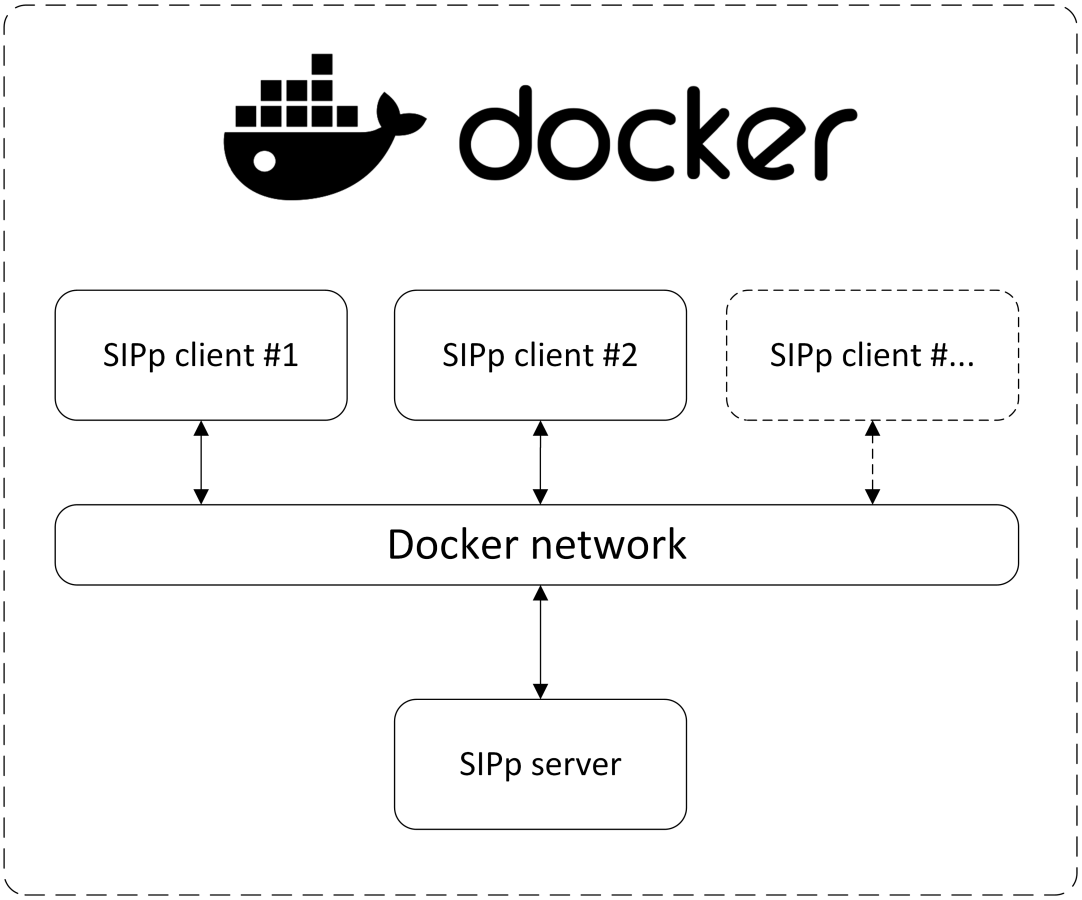

FIGURE 5.3: How the sample SIPp service is deployed inside of the target Docker Engine instance.

In order for Network Service Record associated with the deployment to reach the `ACTIVE` state, all of the MANO, Pop and mgmt component must successfully complete the steps composing the setup tasks. We can therefore expect the instances to be fully functional and configured if such a condition is reached, and their dependencies to have been successfully resolved and specified inside the environment of the containers.

To check the validity of this statement, it is possible to both use the `pop` and the `docker` tools. In particular, `pop servers` can be used to query the Pop server to ensure that the SIPp Pop containers have been correctly instantiated:

```
PS C:\> pop servers
- created: Mar 3, 2017 12:2:32 PM
  extId: 0b2fb0bc-f8a3-4d99-abe4-3cd1d2208e1c
  extendedStatus: the container is running
  flavor:
    <omitted>
  [...]
  image:
    [...]
    name: mcilloni/sipp-server:latest
    [...]
  instanceName: ""
  ips:
    private:
    - 172.16.0.2
  name: sipp-server-1880510
  status: RUNNING
  version: 0
- created: Mar 3, 2017 12:2:33 PM
  extId: d65b8acf-2996-47ae-8fc3-bd4ae3c9cc70
  extendedStatus: the container is running
  flavor:
    <omitted>
  [...]
  image:
    [...]
    name: mcilloni/sipp-client:latest
    [...]
  instanceName: ""
  ips:
    private:
    - 172.16.0.3
  name: sipp-client-672086
  status: RUNNING
  version: 0
```

`pop md get` can then be used to ensure that the VNFM has correctly pushed the metadata values to the client container:

```
PS C:\> pop md get sipp-client-672086
SERVER_PRIVATE: 172.16.0.2
SIPP_LENGTH: 0
SIPP_RATE: 10
SIPP_RATE_INCREASE: 0
SIPP_RATE_MAX: 10
SIPP_RATE_PERIOD: 1000
SIPP_RTP_ECHO: 10
SIPP_TRANSPORT_MODE: u1
```

Of particular relevance is `SERVER_PRIVATE`, an entry which contains the address of the server in the `private` network (`172.16.0.2` in this case), as resolved by the NFVO during the dependency resolution step.

Querying the `dockerd` through the `docker` CLI command shows two identically named matching containers, having the same name as their Pop counterparts. Figure 5.4 shows how the environment of the `client` reflects the variables specified in the metadata, validating the functionality of the system.

```
PS C:\> docker exec -ti sipp-client-672086 sh
/ # env
HOSTNAME=sipp-client-672086
SHLVL=1
SIPP_LENGTH=0
HOME=/root
SIPP_RATE_PERIOD=1000
SIPP_RATE=10
SERVER_PRIVATE=172.16.0.2
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SIPP_RATE_MAX=10
SIPP_RTP_ECHO=10
SIPP_RATE_INCREASE=0
PWD=/
SIPP_TRANSPORT_MODE=u1
```

FIGURE 5.4: The environment inside of the SIPp client correctly reflects the metadata configuration variables specified by the VNFM.

These parameters have been used by the `client_start.sh` script to correctly launch, configure and connect the `sipp` instance to the server running in the other VNF, over the `private` network. The output of the command, showing among other things the errors and messages generated during the exchange between the two SIPp instances, can be found in the several `/uac_*.log` files, as shown in Figure 5.5.

```
PS C:\> docker exec -ti sipp-client-672086 sh
/ # tail uac_30_messages.log
SIP/2.0 200 OK
Via: SIP/2.0/UDP 172.16.0.3:5060;branch=z9hG4bK-30-20672-7
From: sipp <sip:sipp@172.16.0.3:5060>;tag=30SIPpTag0020672
To: service <sip:service@172.16.0.2:5060>;tag=13SIPpTag0130634
Call-ID: 20672-30@172.16.0.3
CSeq: 2 BYE
Contact: <sip:172.16.0.2:5060;transport=UDP>
Content-Length: 0
```

FIGURE 5.5: A message, extracted from the message logs of the `sipp` client instance running with PID 30. Notice the usage of the correct server address, `172.16.0.3` in this sample.

Scaling out

The VNFM correctly supports scaling messages from the NFVO, providing both *scaling in* and *scaling out* functionalities to the VNF instances.
The `client` VNF previously deployed along the service can be scaled out by the NFVO, extending it with an additional VNFC instance, sending a `SCALE_OUT` request to the manager. This will cause an ulterior `sipp-client` container to be created and started inside of the pop:

```
PS C:\> pop servers
[...]
- created: Mar 3, 2017 3:47:45 PM
  extId: 6f2872d2-4d15-4db6-b999-7c96e392c6e8
  extendedStatus: the container is running
  flavor:
    <omitted>
  [...]
```

```
  image:
    [...]
    name: mcilloni/sipp-client:latest
    [...]
  instanceName: ""
  ips:
    private:
    - 172.16.0.4
  name: sipp-client-9851768
  status: RUNNING
  version: 0
PS C:\> pop md get sipp-client-9851768
SERVER_PRIVATE: 172.16.0.2
SIPP_LENGTH: 0
SIPP_RATE: 10
SIPP_RATE_INCREASE: 0
SIPP_RATE_MAX: 10
SIPP_RATE_PERIOD: 1000
SIPP_RTP_ECHO: 10
SIPP_TRANSPORT_MODE: u1
```

The listing above shows the correct configuration of the newly instantiated component.

Termination and scaling out

Finally, the system has been tested for correctness with respect to the deletion of a Network Service, which also involves the termination of the VNFs and any other component involved with the target NSR.

After receiving from an user the request to delete a NS Record, the NFVO proceeds with the invocation of `Delete`, through the Pop VIM Driver, for

each one of the VNFC instances spawned along with the service and during scaling out requests.
Both the Pop and Docker entities are correctly stopped and removed from the running system, as shown below:

```
PS C:\> pop servers
[]
PS C:\> docker ps -a -format '{{.Names}}'
openbaton
```

The `docker` CLI tool shows that only the container running Open Baton has remained in the system; each one of the instances related with the SIPp service have been permanently removed, without any trace left neither on the engine, nor on the Pop server.

Scaling in a single VNF is handled in a very similar fashion. The NFVO, after picking a VNFC instance from those contained in the VDUs associated with the VNF, proceeds with instructing the VIM to remove the container running it, leaving the rest of the service in place.

5.4 Performance measurements

After ensuring the correct functionality of the system, it is also interesting to preliminarily assess a few performance parameters of the developed prototype, while obviously keeping its experimental nature under consideration.

5.4.1 Memory usage and scalability

One of the most compelling selling points of software containers is their relatively lightweight usage of system resources, allowing to spawn several instances without incurring in high costs and overheads.

Figure 5.6 shows how the deployed service is extremely lightweight on RAM, thanks to the containers itself requiring only a negligible amount of services to support their correct functioning. A quick inspection of a running container also shows how almost 100% of the used memory has been allocated by deployed software itself, further confirming the low memory overhead of operating-system level virtualisation.

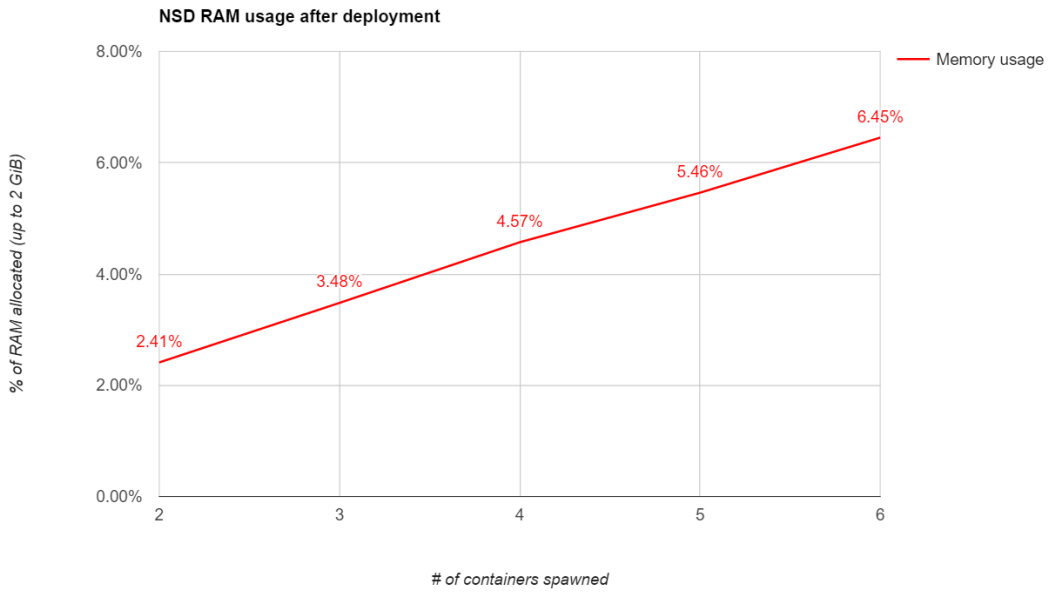

FIGURE 5.6: Even on a very memory constrained set-up, the total size in resident memory of the VNFC instances at their deployment is around 1% per container or less. Notice that the RAM consumed follows a linear trend, directly proportional with the number of instances itself.

Even more dramatic is the small consumption of storage resources achieved

by Docker, through the usage of file system overlays (such as OverlayFS) and copy-on-write techniques; each `sipp-client` instance shares the same image storage, limiting its actual size on disk of the container to just the space necessary to store the log files generated by the tasks running inside of the component.

5.4.2 Performance of new components

The prototype introduces several components, some of which are functionally similar to what already used by the Open Baton MANO framework (such as the AMQP-based VIM Driver and VNFM), and a completely new Docker and gRPC based VIM, with its relatively peculiar client library.
The remaining part of this chapter will rapidly illustrate the experimental results obtained from the tests carried out on the components during their operations, to ensure they do not introduce considerable overheads that may hinder their scaling capabilities and performances.

Memory usage related results

An important factor to keep under consideration is the memory consumed by the components itself during their execution; ideally, this amount should be either constant or at least it should be subjected to negligible variations during their usage, keeping inside of their processes only the data necessary for the correct operation of the system itself.

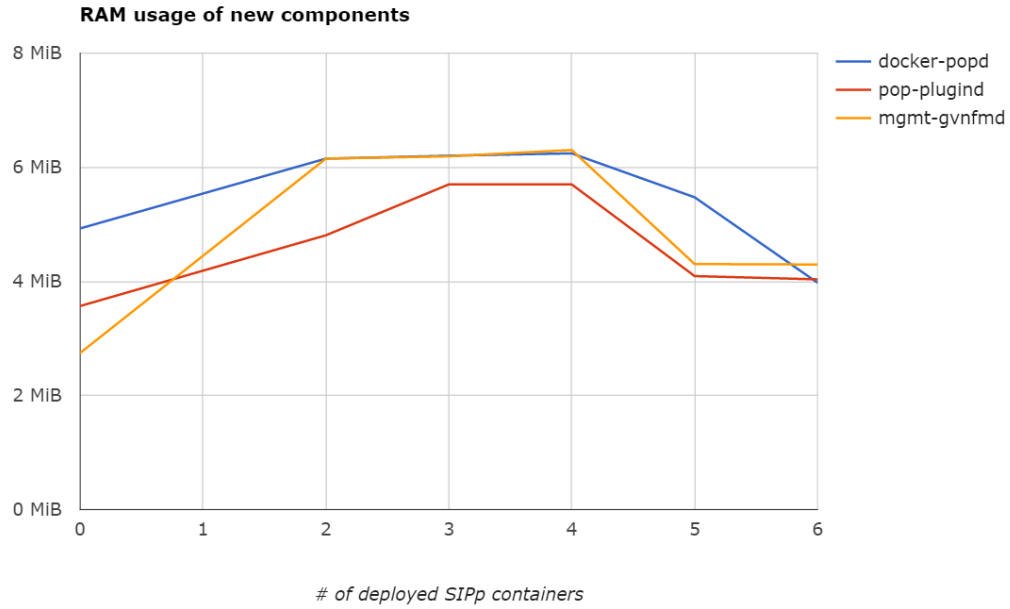

FIGURE 5.7: This graph shows total amount of resident memory consumed by the VIM Driver, the VNFM and the Pop VIM during the SIPp scenario. The RAM consumed by the system is extremely limited, with almost negligible variations.

Figure 5.7 shows very satisfying results regarding the memory usage figures of the components, with the total memory taken by system constantly staying under 20 MiB, even in face of increasing requests and more instantiated containers. The final numbers even show a decrease in memory usage, which can be traced back to the deallocation by the Go garbage collector of temporary transfer objects accumulated during the various requests.

Latency of the Pop server

The Docker Pop server and client libraries introduce several abstractions and functionalities on top of an existing Docker instance, which may negatively affect the latencies involved with container lifecycle operations.

A simple benchmark, leveraging both the Pop and the Go Docker API client libraries, has been devised to experimentally verify the amount of time required to create and fully start up a container instance, in three different circumstances:

1. Spawning a `sipp-server` container using the Docker API client;
2. Spawning a `sipp-server` Pop container, without an existing authenticated session with the server;
3. Spawning a `sipp-server` Pop container with a server for which an already authenticated session is present in the cache.

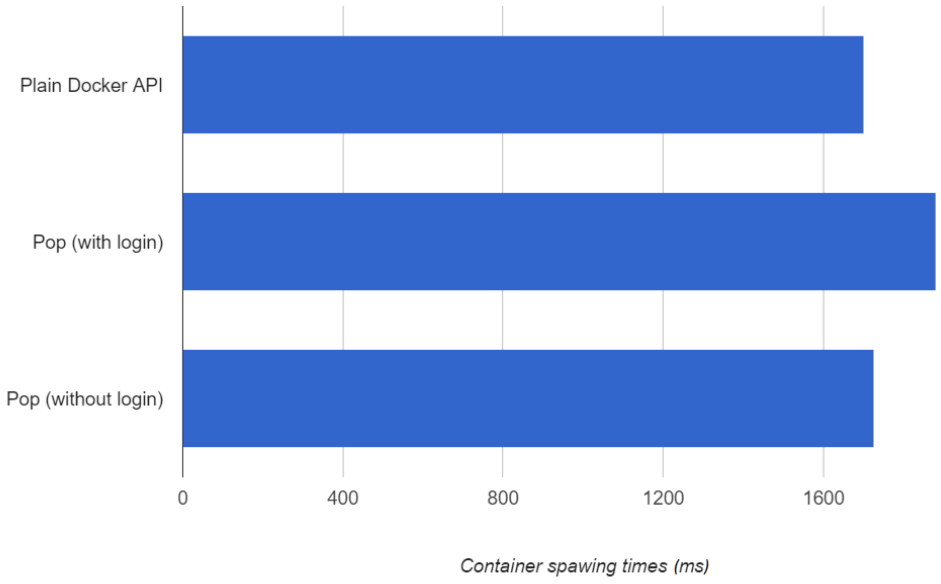

FIGURE 5.8: The latencies introduced by Pop are negligible, and generally irrelevant after a session is established.

A rapid analysis of the `docker-popd` server using the Go builtin profiling facilities has shown that `Login` is a relatively expensive operation, because of the necessary latencies introduced by the security constraints involved with

bcrypt password hashing, which may hamper the testing results. The built-in session cache provided by the client, described in 4.3.1, greatly helps to reduce the number of authentication requests needed during a normal usage of the system.
Figure 5.8 confirms this statement, showing that the session caching facilities provided by the client library efficiently mitigate the small amount of latency involved with the `Login` request, after an authenticated session is successfully established with the server.

Further tests will be carried out during future developments, to better provide performance insights regarding the realised system, given that the large overhead caused by the hardware constraints of the testing machine may have hampered the validity of some of the figures shown above. This, however, helps to reach the final conclusion that the system is perfectly viable even on constrained servers, providing good performances with low running costs.

Conclusions

This thesis addressed the challenges faced while designing an ETSI NFV-MANO compliant container orchestration infrastructure based on the Open Baton framework, analysing the current state of the art in both Network Function Virtualisation and container technologies. The requirements gathered from the analysis of the current NFV standards for Management and Orchestration led to the design and implementation of a final solution capable to provide a solid and flexible framework to extend the existing technologies to new virtualised infrastructures. Through the definition of appropriate protocols and the clear delineation of generic concepts, it has been possible to create highly abstracted components capable to correctly bridge the logical and conceptual mismatch between the Open Baton flavour of NFV and the cloud-oriented world of application container platforms.

A prototype capable of deploying Virtual Network Function component instances on top Docker of has been designed and developed, leveraging modern technologies such as the Go programming language, gRPC and Protocol Buffers to achieve a simpler design, shorter development times and an architecture inherently easy to extend and to interoperate with.

The development process has involved the implementation of a Virtual Infrastructure Manager capable of managing Docker instances and their resources, striving to comply to the NFV standard while studying its differences from the features requested from the hypervisor domain. `docker-popd` is capable to resolve effectively issues like dependencies and lifetimes of containerised VNFs, achieving the end goals established at the beginning of the design step.
The Pop protocol has been designed as an implementation agnostic protocol to add VIMs to the system without requiring new VIM drivers or changes to how the MANO components work. Both the VNFM and the VIM Driver developed together with the VIM are completely standalone components, highly reusable in contexts different than the ones delineated by this thesis.

The experimental results described by Chapter 5 have satified the goals predetermined in Chapter 1, highlighting the memory and storage efficiency of both containers and the implemented components. The memory and storage usage gains enabled by this approach are quite promising, and can result in more efficient, high performance NFV infrastructures capable to cut operating costs through the usage of less expensive hardware systems.

Although the prototype already provides a quite featureful framework implementing many of the features necessary to a container-based NFV system, it is still pretty experimental, and further work needs to be carried on both the project and the Open Baton framework itself to achieve production-ready feature completeness and efficiency.
Every component realised during the course of the project, including the Go libraries, have been or will be contributed as free and open source software to the Open Baton project. This will allow future works to leverage the amount of knowledge gathered to prosecute the process started by this thesis towards

the development of a complete, production-ready solution capable to offer a viable and feature complete alternative to Open Stack for NFV-MANO compliant systems.

Appendix A

Go Open Baton libraries

The **Go Open Baton libraries** are a set of Go packages implementing a complete SDK that simplifies the development of Open Baton MANO components (such as VNFMs and VIM Drivers) written in the **Go** programming language.

These libraries have been designed and implemented as a preliminary work of this thesis, and have been extensively used by all of the services and components mentioned in Chapter 4.

A.1 Overview

Open Baton has offered for a long time a complete set of Java packages implementing the **Catalogue** and the basic runtime environments needed by the project. While these libraries are pretty stable and have been thoroughly tested and used in components such as the NFVO, the Generic VNFM and several other critical infrastructure parts, it is still very desirable to offer the end user the widest possible range of choices when picking the language for their next VNFM or plugin.

Go, while pretty young for a programming language (its first release dates back to only 2009), is already a very capable choice, offering a very interesting mix of dynamism and static typing. Its focus on concurrency and network services, plus the strong commitment of **Google** on its development and success, makes it a very strong addition to the offer provided by the Open Baton project.

The libraries, publicly released under the **Apache 2.0** license, consist of several Go packages implementing the fundamental building blocks of various types of Open Baton components, while keeping complete compatibility with their Java counterparts
The following software solutions are provided by the project:

- A **Catalogue** of (most of) the Open Baton types, together with the serialisation facilities necessary to correctly convert them into JSON messages compatible with the rest of the infrastructure. This also encompass a set of **message** types required for NFVO-VNFM interactions;
- A complete **plugin** runtime based on AMQP, which allows an user to implement `VIM Drivers` without worrying about the protocol below;
- A complete **VNFM** runtime, built around an abstract **channel** package, which makes easy to implement a VNF Manager using one of the various protocols offered by the NFVO;
- An **AMQP channel** implementation, which implements the necessary protocol to connect a VNFM to the NFVO through an AMQP message queue.

A.2 Catalogue

Catalogue consists in two Go packages, `catalogue` and its subpackage `messages`, implementing the Go counterparts to many fundamental Open Baton types and the message types used by NFVO-VNFM communications respectively.

The Catalogue strives to describe its types using Go in an idiomatic fashion, while keeping at the same time full interoperability with the Open Baton protocols and entities. This goal is achieved through the usage of `structure tags`, which allow to mark the fields of Go structures with useful metadata (like, in this case, the name the field once serialised to JSON will have), as shown for the `VirtualLink` structure in Figure A.1.

```
type VirtualLink struct {
  ID               string   `json:"id,omitempty"`
  HbVersion        int      `json:"hb_version"`
  ExtID            string   `json:"extId"`
  RootRequirement  string   `json:"root_requirement"`
  LeafRequirement  string   `json:"leaf_requirement"`
  QoS              []string `json:"qos"`
  TestAccess       []string `json:"test_access"`
  ConnectivityType []string `json:"connectivity_type"`
  Name             string   `json:"name"`
}
```

FIGURE A.1: Definition of the `VirtualLink` structure, as defined by the `catalogue` package.

The Catalogue packages also offer several functions to create and operate on VNFR structures, and to correctly handle the marshalling from and to JSON

of both NFVO and VNFM message types.

A.3 Plugins

The Go Open Baton libraries offer the implementation of a runtime for Open Baton VIM Drivers as the **plugin** package.

This library offers a support platform which makes defining new VIM Driver plugins as simple as implementing a type adhering to the methods defined by the `plugin.Driver` interface. The runtime supports the `plugin.Plugin` instances created using the `plugin.NewVIM` method, handling tasks such as executing RPC requests from the NFVO through an AMQP message queue using parallel workers, and handling reconnection errors.

```
var driver plugin.Driver = &myDriver{}

params := &plugin.Params{ /* your configuration here */ }

plug, err := plugin.NewVIM(driver, params)
if err != nil {
    panic("error: " + err.Error())
}
```

FIGURE A.2: Simple sample of how the `plugin` package can be used to implement a VIM Driver.

The `plugin` package also implements the exchange types necessary to correctly marshal the JSON messages sent to and by the NFVO.

A.4 VNFM

The **vnfm** package provides a full runtime to support the development of VNF Managers using the Go language. This library is designed to leverage the concurrency features and primitives provided by Go to implement a parallel, worker-based infrastructure, suited to be used to develop and design VNFMs based on any available API exposed by the NFVO.

```
// import the driver
import _ "driver/package/xyz"

var handler vnfm.Handler = &myHandler{}

cfg, err := config.LoadFile("path/to/config.toml")
if err != nil {
    panic("cannot load config, " + err.Error())
}

// "xyz" is the identifier of the desired driver.
svc, err := vnfm.New("xyz", handler, cfg)
if err != nil {
    panic("error: " + err.Error())
}
```

FIGURE A.3: Simple sample of how the `vnfm` package can be used together with the AMQP channel to implement a VNF Manager.

A flexible configuration system is provided through the `vnfm/config` package, which allows a single configuration file to carry configuration parameters for both the VNFM runtime and its underlying channel.

```
[vnfm]
type = "docker"
endpoint = "docker-endpoint"
description = "Docker VNFM"
timeout = 2000

    [vnfm.logger]
    level = "DEBUG"
    use-colors = true

[amqp]
host = "localhost"
username = "admin"
password = "openbaton"

    [amqp.exchange]
    name = "openbaton-exchange"
```

FIGURE A.4: Sample TOML configuration file for a given VNFM implementation, specifying both VNFM and AMQP parameters.

A.4.1 Channel

Full independence of the runtime from the used VNFM API is achieved through the `channel.Channel` interface, an entity defining the basic set of communication primitives expected from an NFVO-VNFM connection.

The `channel` package implements a pluggable registry system, which allows implementations to register as `Channel` implementations, offering an high degree of flexibility and genericness to the infrastructure.

A.4.2 AMQP channel

The Go Open Baton libraries provide a full AMQP `channel.Channel` implementation as the `vnfm/amqp` package. Providing features such as automatic re-establishment of connections and concurrent execution of incoming requests, this package handles the full communication protocol between the VNFM and the NFVO, providing full interoperability with the rest of the architecture.

Appendix B

Pop Protocol Buffers Definition

The code listed below represents the full Protocol Buffers definition of the Pop protocol.

```
syntax = "proto3";

package vim_pop;

option go_package = "proto";

import "empty.proto";

// Service definition.
service Pop {
  // Listing functions

  // Containers returns the containers available in the PoP, either
  // created or running.
  rpc Containers(Filter) returns (ContainerList);

  // Flavours returns the available flavours.
  // This doesn't make much sense with containers, but it's here to
  // better abstract the PoP.
```

```
20  rpc Flavours(Filter) returns (FlavourList);
21
22  // Images returns the images available in the PoP.
23  rpc Images(Filter) returns (ImageList);
24
25  // Networks returns the available retworks in the PoP.
26  rpc Networks(Filter) returns (NetworkList);
27
28  // container functions
29
30  // Create creates a new container as described.
31  rpc Create(ContainerConfig) returns (Container);
32
33  // Delete stops and deletes the container identified by the given
      filter.
34  rpc Delete(Filter) returns (google.protobuf.Empty);
35
36  // Metadata adds the given metadata values to the container that
     matches with the ID.
37  // An empty value for a key means that the key will be removed
     from the metadata.
38  rpc Metadata(NewMetadata) returns (google.protobuf.Empty);
39
40  // Start starts the container identified by the given filter.
41  // Any metadata key stored in the server will be passed to the
     newly instantiated container.
42  rpc Start(Filter) returns (Container);
43
44  // Stop starts the container identified by the given filter.
45  rpc Stop(Filter) returns (google.protobuf.Empty);
46
47  // login/logout functions
48
49  // Login logs an user in and sets up a session.
50  // The returned token should be set into the metadata
```

```
51   // of the gRPC session with key "token" to authenticate your
      client.
52   rpc Login(Credentials) returns (Token);
53
54   // Logout invalids the current token.
55   rpc Logout(google.protobuf.Empty) returns (google.protobuf.Empty)
      ;
56
57   // other functions
58
59   // Info can be used to check if the Pop is alive and if your
      credentials to this service are valid.
60   // It also returns informations about this server.
61   rpc Info(google.protobuf.Empty) returns (Infos);
62 }
63
64 message Container {
65   string id = 1;
66   repeated string names = 2;
67   string image_id = 3;
68   string flavour_id = 4;
69   string command = 5;
70   int64 created = 6;
71   int64 started = 7;
72
73   enum Status {
74     UNAVAILABLE = 0;
75     CREATED  = 1;
76     RUNNING  = 2;
77     EXITED   = 3;
78     FAILED   = 4;
79     STOPPING = 5;
80   }
81
82   Status status = 8;
```

```
  string extended_status = 9;
  map<string, Endpoint> endpoints = 10;
  Metadata md = 11;
}

message ContainerConfig {
  string name = 1;
  string image_id = 2;
  string flavour_id = 3;
  map<string, Endpoint> endpoints = 4;
}

message ContainerList {
  repeated Container list = 1;
}

// Credentials represents the login credentials for a given user.
message Credentials {
  string username = 1;
  string password = 2;
}

message Endpoint {
  string net_id = 1;
  string net_name = 2;
  string endpoint_id = 3;
  Ip ipv4 = 4;
  Ip ipv6 = 5;
  string mac = 6;
}

// Filter is used to specify a filter that matches a container.
message Filter {
  oneof options {
    string id = 1;
```

```
118     string name = 2;
119   }
120 }
121
122 message Flavour {
123   string id = 1;
124   string name = 2;
125 }
126
127 message FlavourList {
128   repeated Flavour list = 1;
129 }
130
131 message Image {
132   string id = 1;
133   repeated string names = 2;
134   int64 created = 3;
135 }
136
137 message ImageList {
138   repeated Image list = 1;
139 }
140
141 message Infos {
142   string type = 1;
143   string name = 2;
144   int64 timestamp = 3;
145 }
146
147 message Ip {
148   string address = 1;
149   Subnet subnet = 2;
150 }
151
152 // Metadata contains a key-value set of metadata
```

```
153 // pairs, that will be exposed to the underlying container.
154 message Metadata {
155   map<string, string> entries = 1;
156 }
157
158 message Network {
159   string id = 1;
160   string name = 2;
161   bool external = 3;
162   repeated Subnet subnets = 4;
163 }
164
165 message NetworkList {
166   repeated Network list = 1;
167 }
168
169 message NewMetadata {
170   Filter filter = 1;
171   Metadata md = 2;
172 }
173
174 message Subnet {
175   string cidr = 1;
176   string gateway = 2;
177 }
178
179 // Token is a token generated by the server after a successful
      login.
180 // This token should be set as metadata, to authenticate every
      other
181 message Token {
182   string value = 1;
183 }
```

Appendix C

SIPp Open Baton NSD

This appendix reports the full definition of the Network Service Descriptor used in Chapter 5 to test the SIPp case.

```
{
    "name":"docker-test-NS-sipp",
    "vendor":"mcilloni",
    "version":"1.0",
    "vld":[
        {
            "name":"private"
        }
    ],
    "vnfd":[
        {
            "name":"sipp-client",
            "vendor":"mcilloni",
            "version":"1.0",
            "lifecycle_event":[
                {
                    "event":"CONFIGURE",
                    "lifecycle_events":[
                        "server_sipp_start.sh"
```

```
                    ]
                },
                {
                    "event":"INSTANTIATE",
                    "lifecycle_events":[
                        "sipp_install.sh"
                    ]
                }
            ],
            "vdu":[
                {
                    "vm_image":[
                        "mcilloni/sipp-client:latest"
                    ],
                    "scale_in_out":5,
                    "vnfc":[
                        {
                            "connection_point":[
                                {
                                    "virtual_link_reference":"
private"
                                }
                            ]
                        }
                    ],
                    "vimInstanceName":[]
                }
            ],
            "configurations": {
                "configurationParameters": [
                    {
                        "confKey": "SIPP_LENGTH",
                        "value": "0",
```

```
52                      "description": "Controls the length (in
    milliseconds) of calls. More precisely, this controls the
    duration of 'pause' instructions in the scenario, if they do not
     have a 'milliseconds' section. Default value is 0."
53                  },
54                  {
55                      "confKey": "SIPP_RATE",
56                      "value": "10",
57                      "description": "Set the call rate (in calls
     per seconds). Default is 10. If the -rp option is used, the
    call rate is calculated with the period in ms given by the user
    ."
58                  },
59                  {
60                      "confKey": "SIPP_RATE_PERIOD",
61                      "value": "1000",
62                      "description": "Specify the rate period in
    milliseconds for the call rate.  Default is 1 second.  This
    allows you to have n calls every m milliseconds (by using -r n -
    rp m). Example: -r 7 -rp 2000 ==> 7 calls every 2 seconds."
63                  },
64                  {
65                      "confKey": "SIPP_RATE_MAX",
66                      "value": "10",
67                      "description": "If -rate_increase is set,
    then quit after the rate reaches this value. Example: -
    rate_increase 10 -max_rate 100 ==> increase calls by 10 until
    100 cps is hit."
68                  },
69                  {
70                      "confKey": "SIPP_RATE_INCREASE",
71                      "value": "0",
```

```
72                      "description": "Specify the rate increase
   every -fd seconds.  This allows you to increase the load for
   each independent logging period. Example: -rate_increase 10 -fd
   10 ==> increase calls by 10 every 10 seconds."
73                  },
74                  {
75                      "confKey": "SIPP_RTP_ECHO",
76                      "value": "10",
77                      "description": "Enable RTP echo. RTP/UDP
   packets received on port defined by -mp are echoed to their
   sender. RTP/UDP packets coming on this port + 2 are also echoed
   to their sender (used for sound and video echo)."
78                  },
79                  {
80                      "confKey": "SIPP_TRANSPORT_MODE",
81                      "value": "u1",
82                      "description": "Set the transport mode: -
   u1: UDP with one socket (default), - un: UDP with one socket per
    call, - ui: UDP with one socket per IP address The IP addresses
    must be defined in the injection file. - t1: TCP with one
   socket, - tn: TCP with one socket per call, - l1: TLS with one
   socket, - ln: TLS with one socket per call, - c1: u1 +
   compression (only if compression plugin loaded), - cn: un +
   compression (only if compression plugin loaded)."
83                  }
84              ],
85              "name": "sipp-configuration"
86          },
87          "virtual_link":[
88              {
89                  "name":"private"
90              }
91          ],
92          "deployment_flavour":[
93              {
```

```
                    "flavour_key":"docker.container"
                }
            ],
            "auto_scale_policy": [
                {
                    "name":"scale-out",
                    "threshold":100,
                    "comparisonOperator":">=",
                    "period":30,
                    "cooldown":60,
                    "mode":"REACTIVE",
                    "type":"VOTED",
                    "alarms": [
                        {
                            "metric":"system.cpu.util[,idle]",
                            "statistic":"avg",
                            "comparisonOperator":"<=",
                            "threshold":40,
                            "weight":1
                        }
                    ],
                    "actions": [
                        {
                            "type":"SCALE_OUT",
                            "value":"1"
                        }
                    ]
                },
                {
                    "name":"scale-in",
                    "threshold":100,
                    "comparisonOperator":">=",
                    "period":30,
                    "cooldown":60,
                    "mode":"REACTIVE",
```

```
129                     "type":"VOTED",
130                     "alarms": [
131                         {
132                             "metric":"system.cpu.util[,idle]",
133                             "statistic":"avg",
134                             "comparisonOperator":">=",
135                             "threshold":60,
136                             "weight":1
137                         }
138                     ],
139                     "actions": [
140                         {
141                             "type":"SCALE_IN",
142                             "value":"1"
143                         }
144                     ]
145                 }
146             ],
147             "type":"client",
148             "endpoint":"docker",
149             "vnfPackageLocation":"https://github.com/openbaton/vnf-
    scripts.git"
150         },
151         {
152             "name":"sipp-server",
153             "vendor":"mcilloni",
154             "version":"1.0",
155             "lifecycle_event":[
156                 {
157                     "event":"INSTANTIATE",
158                     "lifecycle_events":[
159                         "sipp_install.sh",
160                         "sipp_server_start.sh"
161                     ]
162                 }
```

```
163             ],
164             "virtual_link":[
165                 {
166                     "name":"private"
167                 }
168             ],
169             "vdu":[
170                 {
171                     "vm_image":[
172                         "mcilloni/sipp-server:latest"
173                     ],
174                     "scale_in_out":1,
175                     "vnfc":[
176                         {
177                             "connection_point":[
178                                 {
179                                     "virtual_link_reference":"
    private"
180                                 }
181                             ]
182                         }
183                     ],
184                     "vimInstanceName":[]
185                 }
186             ],
187             "deployment_flavour":[
188                 {
189                     "flavour_key":"docker.container"
190                 }
191             ],
192             "type":"server",
193             "endpoint":"docker",
194             "vnfPackageLocation":"https://github.com/openbaton/vnf-
    scripts.git"
195         }
```

```
    ],
    "vnf_dependency":[
        {
            "source":{
                "name":"sipp-server"
            },
            "target":{
                "name":"sipp-client"
            },
            "parameters":[
                "private"
            ]
        }
    ]
}
```

Bibliography

[1] ITU-T. (Nov. 4, 2004). Definition of Next Generation Network, [Online]. Available: http://www.itu.int/ITU-T/studygroups/com13/ngn2004/working_definition.html (visited on 01/14/2017).

[2] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, *et al.*, "Network Function Virtualisation. an introduction, benefits, enablers, challenges & call for action", *SDN and OpenFlow World Congress*, pp. 22–24, 2012.

[3] SDxCentral, *What is NFV Infrastructure (NFVI)? definition*. [Online]. Available: https://www.sdxcentral.com/nfv/definitions/nfv-infrastructure-nfvi-definition/ (visited on 01/15/2017).

[4] N. E. ISG, *ETSI GS NFV-MAN 001 V1.1.1 (2014-12) Network Functions Virtualisation (NFV); Management and Orchestration*. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf (visited on 01/15/2017).

[5] E. T. S. Institution, *ETSI - European Telecommunications Standards Institute*. [Online]. Available: http://www.etsi.org (visited on 01/15/2017).

[6] SDxCentral, *What is NFV MANO?* [Online]. Available: https://www.sdxcentral.com/nfv/definitions/nfv-mano/ (visited on 01/15/2017).

[7] Docker, *Docker homepage*. [Online]. Available: http://www.docker.io (visited on 01/15/2017).

[8] Open Baton, *About Open Baton*. [Online]. Available: https://openbaton.github.io/index.html#about (visited on 01/30/2017).

[9] E. T. S. Institution, *Network Functions Virtualisation (NFV); Infrastructure; Network Domain*. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/005/01.01.01_60/gs_NFV-INF005v010101p.pdf (visited on 01/28/2017).

[10] Y. Yu, *Os-level virtualization and its applications*. ProQuest, 2007.

[11] D. Marshall, "Understanding full virtualization, paravirtualization, and hardware assist", *VMWare White Paper*, 2007.

[12] Wikipedia, *Operating-system-level virtualization (implementations)*. [Online]. Available: https://en.wikipedia.org/wiki/Operating-system-level_virtualization#Implementations (visited on 01/31/2017).

[13] Open Container Initiative, *About | Open Container Initiative*. [Online]. Available: https://www.opencontainers.org/about (visited on 02/01/2017).

[14] R. Bonafiglia, I. Cerrato, F. Ciaccia, M. Nemirovsky, and F. Risso, "Assessing the performance of virtualization technologies for NFV: A preliminary benchmarking", in *Software Defined Networks (EWSDN), 2015 Fourth European Workshop on*, IEEE, 2015, pp. 67–72.

[15] A. Ghanwani, D. Krishnaswamy, R. (Krishnan, P. Willis, natarajan.sriram@gmail.com, A. Chaudhary, and F. Huici, "An Analysis of Lightweight Virtualization Technologies for NFV", Internet Engineering Task Force, Internet-Draft draft-natarajan-nfvrg-containers-for-nfv-03, Jul. 2016, Work in Progress,

16 pp. [Online]. Available: `https://tools.ietf.org/html/draft-natarajan-nfvrg-containers-for-nfv-03`.

[16] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers", in *Computing, Networking and Communications (ICNC), 2016 International Conference on*, IEEE, 2016, pp. 1–7.

[17] K. Ye, X. Jiang, S. Chen, D. Huang, and B. Wang, "Analyzing and modeling the performance in Xen-based virtual cluster environment", in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, IEEE, 2010, pp. 273–280.

[18] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors", in *ACM SIGOPS Operating Systems Review*, ACM, vol. 41, 2007, pp. 275–287.

[19] J. Fink, "Docker: A software as a service, operating system-level virtualization framework", *Code4Lib Journal*, vol. 25, 2014.

[20] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, "Containers checkpointing and live migration", in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 85–90.

[21] Microsoft, *Hyper-V containers*. [Online]. Available: `https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container` (visited on 02/02/2017).

[22] G. A. Carella and T. Magedanz, "Open Baton: A framework for Virtual Network Function Management and Orchestration for emerging software-based 5G networks", *Newsletter*, vol. 2016, 2015.

[23] M.-P. Odini, *Nfv open source projects*. [Online]. Available: https://www.slideshare.net/mpodini/nfv-open-source-projects (visited on 03/04/2017).

[24] N. ETSI, "GS NFV 002-V1. 1.1-Network Function Virtualisation (NFV)-Architectural Framework", *publishing October*, 2013.

[25] N. ETSI, "GS NFV-INF 004 - V1.1.1-Network Functions Virtualisation (NFV); Infrastructure; Hypervisor Domain", *publishing January*, 2015.

[26] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, Xen and Docker: A performance analysis for ARM based NFVand cloud computing", in *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in*, IEEE, 2015, pp. 1–8.

[27] T. Bui, "Analysis of Docker security", *arXiv preprint arXiv:1501.02967*, 2015.

[28] C. Metz. (Jun. 25, 2014). Cloud computing could do more to save the planet than electric cars, [Online]. Available: https://www.wired.com/2014/06/containers-v-virtual-machines/ (visited on 02/14/2017).

[29] Docker, *Docker overview*. [Online]. Available: https://docs.docker.com/engine/understanding-docker/ (visited on 02/15/2017).

[30] Docker, *Understand images, containers, and storage drivers*. [Online]. Available: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/ (visited on 02/15/2017).

[31] Open Baton, *Create a VIM Driver*. [Online]. Available: https://openbaton.github.io/documentation/vim-driver-create/ (visited on 02/15/2017).

[32] gRPC, *About gRPC*. [Online]. Available: http://www.grpc.io/about/ (visited on 02/18/2017).

[33] SIPp, *Welcome to sipp*. [Online]. Available: `http://sipp.sourceforge.net/` (visited on 03/03/2017).