Symmetric redundancy of network functions and services on virtualized network infrastructures

Frederico André Dias Januário Santana

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisor: Prof. Fernando Henrique Córte-Real Mira da Silva

Examination Committee

Chairperson: Prof. Ricardo Jorge Fernandes Chaves

Supervisor: Prof. Fernando Henrique Córte-Real Mira da Silva

Member of the Committee: Prof. Rui Jorge Morais Tomaz Valadas

October 2019
Abstract

The traditional way of implementing network functions and network services does not make use of the flexibility, adaptability and interoperability offered by virtualized network infrastructures. Using these infrastructures, it is possible to scale network solutions better, because the resources are being used in the most efficient way possible by virtualizing the required resources. The technologies, tools and products associated with it can now, be a type of service and therefore automated using e.g. a data serialization language like YAML. Considering these developments, the ETSI elaborated an architectural framework, called ETSI MANO, which is used as a reference model for the creation of software that does the orchestration and management of virtualized network functions and network services. This architectural framework has three major components, the NFVO, the VNFM and the VIM that are essential for the management and operation of virtualized network functions and services systems. In this dissertation, an analysis and study of the technologies associated with the three major components of the ETSI MANO architectural framework are performed. Based on the study and analysis done, a symmetric redundant network functions and services system is built on a virtualized network infrastructure. For testing the system, it is deployed a VNF load balancer and two sites, where each site has two web servers.

Keywords

VNFs, NSs, orchestration, management. ETSI NFV framework
Resumo

O modo tradicional de implementação de funções e serviços de rede, não faz o uso devido da flexibilidade, adaptabilidade e interoperabilidade das infraestruturas em rede virtualizadas. Ao usar estas infraestruturas é possível melhorar a eficiência dos recursos usados nas soluções de rede, visto que os recursos usados são virtualizados. Os produtos, ferramentas e tecnologias associadas às funções e serviços de rede passam agora a serem tratados como serviços, podendo depois ser automatizados usando e.g. uma linguagem de serialização de dados como é a linguagem YAML. Considerando estes desenvolvimentos, o ETSI elaborou uma architectural framework denominada de ETSI MANO que é usada como modelo de referência para o desenvolvimento de software para a realização da orquestração e gestão de funções e serviços de rede virtualizados. Esta architectural framework é constituída por três blocos principais, o NFVO, o VNF e o VIM, essenciais para a gestão e operação dos sistemas de serviços e funções de rede. Nesta dissertação, é feita uma análise e estudo das tecnologias associadas com os três principais componentes da ETSI MANO architectural framework. Baseado no estudo e analise feitos, um sistema redundante de funções e serviços de rede é construído dentro duma infraestrutura de rede virtualizada. Para o teste deste sistema, é implementada uma função de rede de balanceador de carga e dois sítios, em que cada sítio é composto por dois servidores web.

Palavras-Chave

VNFs, NSs, orquestração, gestão, ETSI NFV framework
Acknowledgments

I would like to thank Professor Fernando Mira da Silva for his help, patience, encouragement and persistence to make this dissertation project possible.

I am so grateful to my mom for the support and encouragement throughout these years to keep helping me fulfil my dreams as an engineer.

To my closest friends, i would like to thank you for being what you are, a friend for the good and bad times in life.
Table of Contents

Abstract ............................................................................................................................... iv
Resumo ............................................................................................................................... vi
Acknowledgments ........................................................................................................... viii
Table of Contents ........................................................................................................... x
List of figures .................................................................................................................... xii
List of Tables ................................................................................................................... xiii
List of Acronyms .............................................................................................................. xv

1. Introduction ................................................................................................................ 18
   1.1. Motivation and Goals .......................................................................................... 18
   1.2. Document structure ............................................................................................ 19

2. State of the art ............................................................................................................. 20
   2.1. ETSI NFV architectural framework .................................................................. 20
   2.2. ETSI MANO block ............................................................................................. 23
       2.2.1. TOSCA and YAML .................................................................................. 24
   2.3. VIMs block ......................................................................................................... 27
       2.3.1. The OpenStack platform .......................................................................... 27
       2.3.2. Kubernetes ............................................................................................... 30
       2.3.3. Docker ....................................................................................................... 32
   2.4. NFVOs and VNFMs blocks .............................................................................. 33
       2.4.1. OSM ......................................................................................................... 33
       2.4.2. CORD ....................................................................................................... 34
       2.4.3. Cloudify .................................................................................................... 36
       2.4.4. Open Baton .............................................................................................. 37
       2.4.5. Tacker ....................................................................................................... 37

3. Architecture ................................................................................................................. 39
   3.1. Implementation ..................................................................................................... 40
       3.1.1. Management server deployment and configuration ..................................... 40
       3.1.2. DevStack and Tacker projects deployment and configuration ...................... 41
       3.1.3. sBackend and frontend VNFs deployment and configuration ...................... 43

4. Evaluation ..................................................................................................................... 45
List of figures

Figure 1: ETSI NFV architectural framework ................................................................. 20
Figure 2: ETSI MANO framework .................................................................................. 23
Figure 3: NSs elements and ETSI MANO records and descriptors .................................. 24
Figure 4: Conceptual architecture of the OpenStack elements ........................................ 27
Figure 5: High-level overview of the OpenStack essential projects .................................. 28
Figure 6: Kuryr architecture .......................................................................................... 29
Figure 7: Kata containers architecture .......................................................................... 30
Figure 8: Kubernetes architecture ................................................................................ 31
Figure 9: Kubernetes cluster example architecture with 1 master and 2 worker nodes .......... 32
Figure 10: Docker architecture ...................................................................................... 33
Figure 11: OSM architecture ......................................................................................... 34
Figure 12: ONOS architecture ....................................................................................... 35
Figure 13: Cloudify Manager architecture ..................................................................... 36
Figure 14: Open Baton architecture ............................................................................... 37
Figure 15: Tacker architecture and Workflow ................................................................ 38
Figure 16: Project architecture ....................................................................................... 39
Figure 17: Implementation phases ................................................................................... 40
Figure 18: Distribution of GET requests with four web servers ...................................... 45
Figure 19: Distribution of GET requests with five web servers ...................................... 46
List of Tables

Table 1: Summary table of the evaluation tests.................................................................................. 45
Table 2: Requests time with four web servers .................................................................................. 46
Table 3: Requests time with five web servers.................................................................................. 46
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETSI MANO</td>
<td>European Telecommunications Standards Institute Management and Orchestration</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>YAML</td>
<td>Ain't Markup Language</td>
</tr>
<tr>
<td>NFVO</td>
<td>Network Functions Virtualization Orchestrator</td>
</tr>
<tr>
<td>VNFM</td>
<td>Virtual Network Functions Manager</td>
</tr>
<tr>
<td>VIM</td>
<td>Virtual Infrastructure Manager</td>
</tr>
<tr>
<td>VNFs</td>
<td>Virtual Network Functions</td>
</tr>
<tr>
<td>NFVI</td>
<td>Network Functions Virtualization Infrastructure</td>
</tr>
<tr>
<td>NFV MANO</td>
<td>Network Functions Virtualization Management and Orchestration</td>
</tr>
<tr>
<td>OBSS</td>
<td>Operational and Billing Support System</td>
</tr>
<tr>
<td>ETSI NFV</td>
<td>European Telecommunications Standards Institute Network Functions virtualization</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Functions Virtualization</td>
</tr>
<tr>
<td>E2E</td>
<td>End-to-End</td>
</tr>
<tr>
<td>VNFFGs</td>
<td>Virtualized Network Functions Forwarding Graphs</td>
</tr>
<tr>
<td>NFs</td>
<td>Network Functions</td>
</tr>
<tr>
<td>PNFs</td>
<td>Physical Network Functions</td>
</tr>
<tr>
<td>NSs</td>
<td>Network Services</td>
</tr>
<tr>
<td>VLS</td>
<td>Virtual Links</td>
</tr>
<tr>
<td>TOSCA</td>
<td>Topology and Orchestration Specification for Cloud Applications</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processor Unit</td>
</tr>
<tr>
<td>VNFD</td>
<td>Virtual Network Functions Descriptor</td>
</tr>
<tr>
<td>NSD</td>
<td>Network Service Descriptor</td>
</tr>
<tr>
<td>VNFFGD</td>
<td>Virtual Network Functions Forward Graph Descriptor</td>
</tr>
<tr>
<td>VLD</td>
<td>Virtual Link Descriptor</td>
</tr>
<tr>
<td>PNFD</td>
<td>Physical Network Functions Descriptor</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>NSR</td>
<td>Network Services Record</td>
</tr>
<tr>
<td>VNFFGR</td>
<td>Virtual Network Functions Forward Graph Descriptor</td>
</tr>
<tr>
<td>VLR</td>
<td>Virtual Link Record</td>
</tr>
<tr>
<td>VNFR</td>
<td>Virtual Network Functions Record</td>
</tr>
<tr>
<td>PNFR</td>
<td>Physical Network Functions Record</td>
</tr>
<tr>
<td>VDU</td>
<td>Virtual Data Unit</td>
</tr>
<tr>
<td>CP</td>
<td>Connection Point</td>
</tr>
<tr>
<td>VL</td>
<td>Virtual Link</td>
</tr>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>APIs</td>
<td>Application Programming Interfaces</td>
</tr>
<tr>
<td>VMs</td>
<td>Virtual Machines</td>
</tr>
<tr>
<td>OSM</td>
<td>Open Source MANO</td>
</tr>
<tr>
<td>SO</td>
<td>Service Orchestrator</td>
</tr>
<tr>
<td>RO</td>
<td>Resource Orchestrator</td>
</tr>
<tr>
<td>VCA</td>
<td>VNF Configuration and Abstraction</td>
</tr>
<tr>
<td>CORD</td>
<td>Central Office Re-architected as a Datacenter</td>
</tr>
<tr>
<td>ONOS</td>
<td>Open Network Operating System</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Networking</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>GB</td>
<td>Giga Bytes</td>
</tr>
<tr>
<td>LBaaS</td>
<td>Load Balancer as a Service</td>
</tr>
<tr>
<td>XaaS</td>
<td>Everything as a Service</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
</tbody>
</table>
1. Introduction

The network infrastructures were only present in a “bare metal” from not long ago, while nowadays it is shifting to virtualized network infrastructures. The growing of these infrastructures is mainly due to their flexibility, adaptability and interoperability properties. The widespread age of “Cloud” models and virtualization contributes to be a standard when using the Internet. This project aims to study and test the technologies that exist for the orchestration and management of virtual network functions and services in a virtual network infrastructure environment using the solutions that are based on the ETSI MANO architectural framework [1]. This dissertation aims to analyse these solutions, and research simple scenarios where they will be tested in a simple network architecture solution.

1.1. Motivation and Goals

The design of traditional network infrastructures in the past was benchmarked for the minimal loss of latency, availability, throughput and the capacity to carry data, resulting in hardware and software that were developed and optimized using the criteria’s described before. With the increase of complexity and bandwidth usage on technologies such as streaming platforms, Internet of Things or by smartphones, there was a need to scale and expand the existing network infrastructure without increasing the costs too much. Unfortunately, the traditional infrastructure solution posed various bottlenecks in terms of hardware and software bringing companies and developers to find ways of removing those bottlenecks. This led to the concepts of virtualized network infrastructures and virtualized network functions. Having a VNF system fully customizable and define how the VNF system behaves, can be achieved using the ETSI MANO architectural framework. The ETSI MANO architectural framework is the reference framework that companies, developers and users adopted to develop solutions for their needs in this constant changeable world of virtualized network infrastructures. This dissertation describes and analyses the desired skills that are needed to learn more about the optimization of VNF deployment, orchestration and management on virtualized network infrastructures environments.

The main goals of this dissertation are to research the technologies, products and tools used on virtualized network infrastructures, and how the VNFs instantiation is done in the present and what is the near future for VNF management and orchestration on virtualized network infrastructures. For this, a virtualized network infrastructure is deployed to test the most documented and developed VNF placement technology that is low on resource utilization, reliable, secure and is part or fully automatable considering the available hardware at the time of implementation of the system solution. Several virtualization and high virtualization models (containers and virtual machine) are analysed, tested and compared.
1.2. Document structure

The document structure is as follows. This chapter introduces the theme of the project as well the motivation, goals and the structure of the report. Chapter two reviews the existing concepts and technologies for the virtualized network infrastructures and for the VNF instantiation, orchestration and management solutions that exist nowadays. Chapter three describes the architecture and the developed prototype for testing and validation of the architecture. Chapter four describes the tests that were performed. Chapter five provides some conclusion thoughts for the future work and the work done in the research and writing of this report. The additional sections are used for the references used in the elaboration of this dissertation report and project and to show the configuration files that were used for the implementation and evaluation of this dissertation.
2. State of the art

The limitations of network infrastructures such as flexibility, scalability, manageability and interoperability limitations makes companies and developers design new software and hardware that defines how the virtualized network infrastructures are deployed and configured to support the use of virtual network functions and services. This brings new challenges such as multivendor implementations of VNFs, managing, monitoring and configuring the life cycles and interactions of VNFs, as well the hardware resource allocations of VNFs and the interaction with the billing and operational support systems. Those challenges lead for the creation of an architectural framework also known as the ETSI NFV architectural framework, which defines a reference model for virtual network functions and services that are orchestrated and managed on a virtualized network infrastructure.

2.1. ETSI NFV architectural framework

The European telecommunications standards institute network functions virtualization architectural framework is based on a complete separation of hardware and software criteria, where network functions deployment must be automated and scalable and the control of the network functions operational parameters is done by monitoring and controlling the state of the network [2]. This framework is structured by three high level blocks such as the VNFs, the NFVI and the NFV MANO blocks. The NFVI block is the foundation for this architectural framework where it offers the hardware and software responsible for the virtualization process of the virtual instances. The VNFs block uses the resources of the NFVI block to develop software that implements virtualized network functions and services. The NFV MANO block is on its own a separate architectural framework that is responsible for the orchestration and management of the VNFs and NSs resources. The following figure illustrates the ETSI NFV architectural framework:

![ETSI NFV architectural framework](figure1.png)

Figure 1: ETSI NFV architectural framework

Resourced from [2]
The NFV MANO block is also called the ETSI MANO architectural framework where it is a reference model to developers to create software to manage and orchestrate VNFs and NSs. The next sections and subsections describe in a more detailed manner the ETSI MANO architectural framework and all the relevant technologies that are used nowadays to accomplish what the ETSI MANO architectural framework proposes.
2.2. ETSI MANO block

The ETSI MANO block decouples computation, storage and networking from the software that implements NFs by creating new entities such as, the VNFs, NFVI, PNFs, NSs, VNFFGs and VLs. The ETSI MANO architectural framework is responsible for managing the NFVI and orchestrate the resources for the NFs and VNFs. The resources that are considered are mainly CPU, memory, network components (subnets, ports, etc.) and storage. With this, it is important to define VNFs management functions based on operations such as the, instantiation, scaling, updating or upgrading and terminating of VNFs or NSs. These operations are accomplished by the creation of template files that use a template language such as TOSCA which is described in more detail in the following subsection.

The ETSI MANO architectural framework architecture is presented in the figure below:

![Figure 2: ETSI MANO framework](resourced_from_[1])

The main blocks of the ETSI MANO architecture are:

- The NFVO is responsible for the resources allocated to the VIM and the lifecycle management of the NSs.
- The VNFM is responsible for the lifecycle management of the VNF instances, where it can manage only a VNF instance or multiple VNF instances.
- The VIM is responsible for the management and control of the NFVI in terms of the network, storage and compute resources.
The Data Repositories are responsible for the storage of the templates for the VNFs and NSs. Also holds information about the resources and instances that are being used.

The ETSI MANO illustrates also that the NSs as the relation between VNFs and PNFs and defines the elements that a NS relates to, which are depicted in the following figure:

![Network Service Diagram](image)

Figure 3: NSs elements and ETSI MANO records and descriptors

As shown in figure 3, NSs can contain information about the VNFs, Physical network functions, Virtual Links and Virtual Network Function Forwarding Graphs and with this information create network forwarding paths to be used on E2E virtualized network services. The ETSI MANO architectural framework instantiation input parameters are used as descriptors which are grouped in a catalogue and therefore translated to records in a runtime context when the virtual instances are deployed. The descriptors are written in the TOSCA language and have information about the network itself, such as the topology, network path, resource requirements for the elements of the network and the physical elements. The records have not only the information given by the descriptors, but also additional runtime information such as CPU, network or disk usage.

2.2.1. **TOSCA and YAML**

TOSCA[3] or Topology and Orchestration Specification for Cloud Applications is a template language based on the data serialization and markup language YAML (Ain’t Markup Language) that describes the virtualized
network functions or services nodes and the relations between them. In fact, TOSCA is a service template language that describes virtualized network infrastructures workloads as a topology template, meaning that is basically a graph of node templates modelling the components and the relations between them. An example of a VNFD written in TOSCA is depicted on Annex A. The VNFD example is from a NFVO and VNFM software called Tacker, which describes a VNF topology with three node types, a VDU, a VL and a CP, each with different capabilities and requirements. The capabilities describe the resources that each node will be deployed with and the requirements describe the virtual networks that are associated with each VNF.

The YAML language is a data serialization and markup language where integrates and builds concepts from a lot of other languages, e.g. Python, JSON, Ruby, C and XML. YAML language indentation-based scoping is similar to Python language where the indentation facilitates the inspection of the data structures. YAML language literal style leverages this by enabling formatted text to be cleanly mixed within an indented structure without troublesome escaping. YAML also allows the use of traditional indicator-based scoping similar to the JSON language.

YAML language core type system is based on the requirements of agile languages such as Perl, Python, and Ruby. YAML directly supports both collections (mappings, sequences) and scalars. Support for these common types enables programmers to use their language’s native data structures for YAML manipulation, instead of requiring a special document object model. YAML language foremost design goals are human readability and support for serializing arbitrary native data structures [4].
2.3. VIMs block

The virtualized infrastructure managers block has the responsibility to supervise the virtual infrastructure of a network function virtualization solution. In summary the VIM is a key component of any ETSI MANO architecture, and the following subsections describe some of the most developed solutions that exist nowadays for virtualized infrastructure managers.

2.3.1. The OpenStack platform

The OpenStack platform is an open source project that aims to be a cloud operating system that manages and deploys the network, storage and computing resources of a complete virtualized infrastructure over a set of hardware resources. The following figure details the conceptual architecture of the OpenStack elements (services) [5]:

![Conceptual architecture of the OpenStack elements](image)

These elements or projects can be classified as essential projects and additional projects and are explained in more detail on the subsections below.

The growth of the Internet and hardware infrastructures network solutions implies new challenges for operators to manage, configure and launch on demand new services using the resources for them as efficient as possible. OpenStack is an excellent solution for that matter because it offers virtualization of compute, storage, networking and many other resources. Each component in OpenStack manages a different resource that can be virtualized for the end user. Separating each of the resources that can be virtualized into separate components makes the OpenStack architecture very modular. OpenStack can be divided into four groups: Control, Networking, Compute and Storage. The Control tier runs the Application Programming Interfaces (API) services, web interface, database, and message bus. The Networking tier runs network service agents for networking. The Compute tier is the virtualization hypervisor, with services and agents to handle virtual
machines. The Storage tier manages block (Volumes; partitions) and object (containers; files) storage for the Compute instances. All the components use a database and/or a message bus [7].

2.3.1.1. Essential OpenStack projects
The essential projects are the projects that an OpenStack platform cannot operate without. The essential projects to deploy an OpenStack installation are [8]:

- Nova project manages and provisions virtual machines running on hypervisor nodes [9].
- Neutron project provides network connectivity between the interfaces of OpenStack services [10].
- Glance project is a registry service that is used to store resources such as virtual machine images and volume snapshots [11].
- Keystone project is a centralized service for authentication and authorization of OpenStack services and for managing users, projects, and roles [12].

And some advisable but not mandatory projects to add to an OpenStack installation as they facilitate the usage of the OpenStack platform [8]:

- Horizon project is a web browser-based dashboard that is used to manage OpenStack services [13].
- Swift project allows users to store and retrieve files and arbitrary data [14].
- Ceilometer project provides measurements of cloud resources [15].
- Heat project is a template-based orchestration engine that supports automatic creation of resource stacks [16].
- Cinder project manages persistent block storage volumes for virtual machines [17].

A high-level overview of the described projects is depicted on the figure below:

Figure 5: High-level overview of the OpenStack essential projects

Resourced from [8]

All the services communicate with each other by APIs and the AMQP.
2.3.1.2. Relevant OpenStack additional projects

The additional OpenStack projects are software tools that are developed as a side project to add some new features / services to the OpenStack platform. The most important ones for the scope of this dissertation are:

- **Octavia and Octavia dashboard projects** that aim to be a load balancer as service project and a GUI of Octavia project that can manage a fleet of virtual machines, containers, or bare metal servers on demand [18] [19].
- **DevStack project** is a compilation of scripts to quickly bring up a complete and updated version of the OpenStack platform hosted on a bare metal or virtual machine [20].
- **Diskimage-builder project** is a tool for automatically building customized operating-system images to be used in clouds and other environments, producing cloud-images in all common formats (qcow2, vhd, raw, etc), bare metal file-system images and ram-disk images [21].
- **Kolla project** is a provider of production-ready containers and deployment tools for operating OpenStack clouds that are scalable, fast, reliable, and upgradable using community best practices [22].
- **Magnum project** makes container orchestration engines such as Docker Swarm, Kubernetes, and Apache Mesos available as first class resources in OpenStack. Magnum uses Heat OpenStack project to orchestrate an operating system image which contains Docker and Kubernetes and runs that image in either virtual or bare metal machines in a cluster configuration [23].
- **Kuryr-kubernetes and Kuryr** are OpenStack containers networking projects that enables native Neutron-based networking in Kubernetes. With Kuryr-kubernetes it is now possible to choose to run both OpenStack VMs and Kubernetes Pods on the same Neutron network [24].
- **Kata containers project** aims to “deliver standard implementation of lightweight virtual machines that feel and perform like containers but provide the workload isolation and security advantages of virtual machines” [25].

The architectures of the last two additional projects of the list above, are described in more detail by the figures below:

![Kuryr architecture](image)

*Figure 6: Kuryr architecture*

Resourced from [26]

Kuryr uses the libnetwork API to map and create Neutron objects. By doing this, the solutions that Neutron provides (security groups, NAT services and floating IP’s) for networking can be used by containers networking.
Kata containers elements include an Hypervisor to create virtual machines where containers will run, an Agent (kata-agent) for managing containers and processes in the guest machine, an open container initiative (OCI) compatible container runtime (kata-runtime) that handles all commands and launches instances, a Proxy entity that offers access to the virtual machine agent to multiple instances and runtime clients associated with the virtual machine, and a container process (kata-shim) that the container process reaper can monitor.

### 2.3.2. Kubernetes

Kubernetes is an open source solution for managing and orchestrating containers. The architecture type is client-server, where it has one or more master servers that controls and defines how the worker nodes should act and react to the master node. Kubernetes infrastructure is based in five different principles such as pods, services, volume, namespaces and deployment. Pods and Volume are the storage units of Kubernetes, where it stores all information related to the containers and the data of each Pod respectively, Services are a logical set of pods and acts as a gateway to the exterior, allowing (client) pods to send requests to the service without needing to keep track of which physical pods make up the service, Namespaces are based in the Linux namespaces, where here it is a virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users, and finally the Deployment is normally done via a deployment file in the YAML language which describes the configuration and state of pods.

The next figure presents the Kubernetes architecture:
An example of the workflow (with a single master server and two worker servers) of Kubernetes is depicted in the following figure:
Figure 9: Kubernetes cluster example architecture with 1 master and 2 worker nodes

Resourced from [29]

The master node provides Kubernetes with cluster control, making global choices for the cluster and deciding what to do when a cluster event is detected, it has a central management entity (kube-apiserver), a distributed key value storage (etcd), a scheduler that helps optimizing resource utilization (kube-scheduler) and a controller that regulates the state of the Kubernetes or manages the cloud provider (kube-controller-manager or cloud-controller-manager).

The worker node has a service daemon (kubelet) responsible for taking pod specifications and health checks, a proxy service daemon (kube-proxy) responsible for the networking in the worker node and a container runtime (Docker) that is the software that will run the containers on the Pod. Kubernetes has also some useful features (addons) like a DNS Server, a Web UI, a Container Resource Monitoring and a Cluster-level Logging that can make the life easier on the administrator of the Kubernetes platform.

Kubernetes has become in the last few years the standard container orchestration platform, mainly because of the performance gain by using containers over virtual machines and the high availability of the applications running on the containers is provided by the use of container replicas quotas and the health container checks.

2.3.3. Docker

Docker is a client-server application (like Kubernetes), that leverages the technologies of namespaces, control groups, union file formats and container formats. The Docker engine is defined as follows:
Docker is composed by a docker daemon (dockerd) that manages the Docker objects (images, containers, networks and volumes) and the communication with other Docker daemons to manage Docker images, a Docker Client that consists of a REST API and CLI (Docker command or a Docker client) being used to interact with the Docker daemon, a Docker registry for the storage of Docker images (default one being Docker Hub) and a Docker object or objects that can be images, containers or services.

Docker is a fast and consistent delivery system of applications because of the use of containers and being a light program to run offers great scaling, a fast deployment system and the amount of work uses less resources in opposition of using virtual machines.

2.4. NFVOs and VNFMs blocks

The network functions virtualization orchestrator and the virtualized network functions manager blocks are normally bundled together in a software suite but have different responsibilities that need to be attended to. The NFVO is responsible for overlooking the instantiation, scaling, updating and terminating network services. The VNFM is responsible for overlooking the instantiation, scaling, updating and terminating of virtual network functions.

The next subsections describe the most developed and documented software solutions for the implementation and configuration of NFVOs and VNFMs.

2.4.1. OSM

Open source MANO is a project that is run by the ETSI foundation, adhering to the ETSI MANO architecture framework on their management and orchestration proposal. The OSM architecture is depicted below:
As seen in figure ten OSM has three major components:

- SO that is responsible for the service orchestration, provisioning, deploying, querying and the storage of the virtual network definitions and network services catalogues.
- RO that is responsible for the resources provision of networking services over a virtualized network infrastructure.
- VCA that is responsible for the configuration of the virtual network functions using Juju charms.

The configuration files of NSDs and VNFDs are written using the TOSCA language. OSM uses these templates to instantiate network function or services and the necessary resources associated with them. OSM also supports a good variety of virtualized infrastructure managers such as OpenStack or OpenVIM [32] and the deployment of network functions virtualization services in multiple virtualized infrastructure managers.

2.4.2. CORD

In its beginning, central office re-architected as a datacentre was a simple project used by the open network operating system project which is an open source SDN controller for building next-generation SDN/NFV solutions controller. But CORD became a greater project and was separated and run by the Open Networking Foundation. The main objective of CORD is to “combine the technologies of NFV, SDN and the “elasticity of commodity clouds to bring datacentre economics and cloud agility to the Telco Central Office”[33]. CORD components are shown in the following figure:
CORD is composed by five important components:

- Kubernetes platform where CORD run all control panel elements.
- Platform that is a Kubernetes environment with an ONOS, XOS [35], Kafka [36] and a collection of logging and monitoring micro-services.
- Profile is a combination of services such as VNFs, access services or cloud services.
- Workflow being a component of a Profile where describes the business logic and state machine for one of the access technologies contained in the Profile.
- BoM that is the hardware bill of materials defined for each Kubernetes pod.

CORD is installed as a collection of Docker containers in a Kubernetes cluster where Helm[37] is used, which is a packaged manager for Kubernetes. TOSCA templates are used for the configuration and provisioning of a running system of ONOS where XOS (a model-based platform for assembling, controlling, and composing services) is used for developers to run their applications on CORD.
2.4.3. Cloudify

Cloudify is a cloud orchestration and management framework, adhering to the ETSI MANO architecture framework that facilitates how applications and services are modelled and automated in their life cycle. That includes the deployment, monitoring of applications, detecting and resolving issues that may occur while running such applications. The cloudify architecture is depicted below:

![Cloudify Manager architecture](image)

**Figure 13: Cloudify Manager architecture**

Resourced from [38]

The three major components of Cloudify are:

- Cloudify Manager being the brain of the cloudify environment to manage and deploy applications where the deployment is done using the “blueprints” that are TOSCA based templates files that can instantiate a service or application in the system.
- Cloudify Agents used to manage the running applications by using plugins.
- Cloudify Console being the command line tool that Cloudify uses to communicate with the Cloudify Manager.

The Cloudify Manager can use a CLI or GUI for communication and is composed by a Nginx web server that has the function of a proxy server and file server, the Gunicorn and Flask elements where Gunicorn and Flask provide the Cloudify REST service, a PostgreSQL database that provides the main database where it stores the application’s model and the indexing logs’ and events’ storage, a Logstash element that is used by Cloudify to pull log and event messages from RabbitMQ and index them in a PostgreSQL database, a RabbitMQ element that is used to queue deployment tasks, logs, events and metrics, a Riemann element that is an event stream processor used primarily for monitoring, a Pika element that is the communication agent programmed in Python.
of the AMQP protocol and the InfluxDB that is a database used to pull metrics from RabbitMQ and to store them.

2.4.4. Open Baton

Open Baton is an extensible and customizable framework that adhered to the ETSI MANO architecture framework. It uses TOSCA templates to deploy and configure services and applications. The main components of Open Baton are depicted in the figure below:

![Open Baton Architecture](image)

Figure 14: Open Baton architecture

Resourced from [39]

Open Baton elements are, a NFVO that is ETSI MANO compliant, a VNFM and a Generic Element Management System to manage VNFs and VNFDs, a Juju VNFM Adapter in order to deploy Juju Charms, a driver mechanism supporting different type of VIMs, a Docker VNFM and VIM driver for instantiating containers on top of Docker Engine, a powerful event engine based on a pub/sub mechanism for the dispatching of the lifecycle events execution, an autoscaling engine which can be used for automatic runtime management of the scaling operations of your VNFs, a fault management system which can be used for automatic runtime management of faults which may occur at any level, a network slicing engine which can be used to ensure a specific QoS for your NSs, a monitoring plugin integrating Zabbix as monitoring system, a marketplace useful for downloading VNFs compatible with the Open Baton NFVO and VNFM and a set of libraries (in Java, Go and Python) which could be used for building your own VNFM.

2.4.5. Tacker

Tacker is an OpenStack additional project for NSs an VNFs orchestration and management adhering to the ETSI MANO architectural framework. It has a generic NFVO and VNFM to deploy network services and virtual network functions providing E2E solutions.

The next figure shows how Tacker workflow and architecture are:
Tacker has three fundamental components:

- Network Functions Virtualization Catalog that contains the VNF, NS and VNF Forwarding Graph descriptors.
- Virtualized Network Functions Manager (VNFM) that creates, updates, deletes and monitors VNFs.
- Network Functions Virtualization Orchestrator (NFVO) that optimizes resource checks and allocation of VNFs. The NFVO can orchestrate VNFs, throughout multiple VIMs or Sites and can create a service function chain between VNFs by using a VNF Forwarding Graph Descriptor.

Tacker can be deployed and configured manually or using the additional OpenStack projects DevStack or Kolla. Tacker only supports currently as VIMs the OpenStack and Kubernetes platforms.
3. Architecture

The architecture is to deploy and test a system that encompasses one of each of the three main blocks of the ETSI MANO architectural framework presented on chapter two of this dissertation report. The proposed architecture solution will have the following entities:

- A frontend network with one VNF acting as a Load Balancer.
- A backend network with two web servers.
- The backend network will have replicated sites (one and two).
- Two virtualized infrastructure managers installed on the different sites.
- A client to connect and test the system functionalities.

The architecture is depicted below:

![Project architecture](image)

Figure 16: Project architecture

The system workflow starts by the users connecting to the system via a VNF acting as a load balancer that will redirect traffic depending on the load balancing algorithm that was selected, connecting the users to the sites on the backend network. Each site will have two web servers that will be used for testing the load balancer and the sites functionalities, e.g. handling HTTP requests.

The orchestration and management of the global system will be done via a management server. The management server is configured using the necessary software researched in chapter two of this dissertation report and some extra tools that will be described in the next subsection.

The ideal scenario will be that, the servers on site one and the VNF acting as load balancer will be hosted on virtual machines, as for the servers on site two will be hosted on containers. The process of being hosted by
virtual machines and containers is also important to test, analyse and compare how different these host virtualization techniques are in terms e.g. resource utilization.

The proposed architecture is a heavy system to deploy and the available hardware is limited, so the implementation takes all that in consideration by building the system with the minimal resources necessary not compromising the proposed architecture.

3.1. Implementation

The implementation of the architecture is defined in three different phases that are depicted below:

![Figure 17: Implementation phases](image)

The implementation starts with phase number one where a management server is deployed and configured to host the DevStack and Tacker projects used on phase number two. Phase number two main objective is to deploy and configure the DevStack and Tacker projects. Phase number three deploys and configures the backend and frontend VNFs hosted on virtual instances (containers or virtual machines). The next subsections describe in more detail how all the phases were processed.

3.1.1. Management server deployment and configuration

The main goal is to deploy a bare metal or virtual solution of a management server that consists on installing and configuring the software of the virtual network infrastructure that is going to be used to deploy the architecture solution described before. The first step is the most important one as encompasses the research of the software and hardware requirements for the system architecture and choosing the right tools to deploy the management server. The tools chosen were for a virtual solution of the management server, just because it is easier to test the hardware requirements for the architecture solution as is a better modular solution than the bare metal solution, e.g. if a virtual machine does not meet the right requirements it is faster to install and configure the hardware and operating system on the virtual machine. The tools chosen to deploy the virtual solution to the management server where, Packer [41] and Vagrant [42] that are tools respectively, to manage updated virtual images to be used on virtual machines and to deploy those virtual images onto virtualization platforms such as VirtualBox or QEMU. The problems that were encountered here were the time necessary to choose the right operating system and the minimal hardware and software requirements to deploy the management server as it depends on phase two of the implementation of the architecture solution. The operating system chosen was Ubuntu Server 16.04 with Ansible [43] and openstack-sdk packages installed with sixteen GB of RAM, five CPU cores and sixty GB of disk. Annex B depicts the contents of a Vagrantfile, used by Vagrant and written in the Ruby language where it defines the virtual machine configuration to be used for the management server.
3.1.2. **DevStack and Tacker projects deployment and configuration**

After the management server is up and running, it is time to move to the next phase where the DevStack and Tacker OpenStack additional projects are deployed and configured. These projects were chosen based on the hardware requirements of all the software researched for the ETSI MANO architectural framework blocks. For the VIM block the project used for this was DevStack where it deploys a virtualized network infrastructure platform and installs the OpenStack and Kubernetes VIMs. For the NFVO and VNFM blocks, the project used was Tacker.

The first step of using the DevStack project is meeting the following requirements:

- DevStack should be run as a non-root user with root enabled privileges.
- Having GitHub [44] and PIP [45] (package installer for Python) programs installed on the management server used to deploy DevStack.

The next step involves using the GitHub program called git to download the repository of DevStack to a folder and access the folder. When in the DevStack folder, the most important files to look for are the `stack.sh`, `openrc` and `local.conf` files. The `stack.sh` is a bash script file that uses the information of the configuration `local.conf` file to install and configure the OpenStack and Kubernetes VIMs. The `openrc` is also a bash script file used to load the OpenStack environment variables to the management server so that OpenStack can be accessed and managed. The configuration `local.conf` file needs to be created by the user of the management server is based from the `local.conf.kubernetes` file, that can be accessed on the DevsStack GitHub site with slight modifications. The relevant configurations options for the `local.conf` file are:

- The IPs and passwords for the services that will utilize that information such as, the OpenStack database and network services.
- The IP range and network interface that is going to be used to give Internet connectivity to virtual instances, e.g. virtual machines or containers.
- Enabling a log file, it is very important because if anything goes wrong, it is possible to search for errors during installation.
- Enabling the use of the Kubernetes VIM and selecting the hyperkube [47] version to be installed. The hyperkube version can be selected from google public hyperkube image repository [48].
- Enabling the necessary OpenStack services by using the `enable_plugin` command tag. The OpenStack services are then downloaded from a GitHub repository and automatically configured with the default options. The customization of the services is done via proper tags and the tag availability depends if the service has DevStack installation support.

After all the configuration is done, the file must be saved under the DevStack folder and then run the `stack.sh` script. One problem that originated running the script, was when the management server has one network interface and the login method to the management server is via SSH, it is necessary to run the script on a separate virtual terminal by using the command `screen`.

The script is going to run a series of installations and configurations on the system, so it is not recommended to run this script on a daily use operating system installation. This is one of the reasons why the management server is a virtual machine described in the phase before. The script run time depends in the amount of services it must
install and configure, but the script at a fresh install of the operating system on the management server takes about forty five minutes and the time decreases after the first run, to thirty minutes. At the end of the script the output gives useful information such as:

- The time it took to install and configure the script.
- The default IP address used by the user of the management server to access, e.g. via browser the OpenStack UI.
- The users (admin and demo) that have relevant privileges to access the OpenStack platform.
- The version of OpenStack that was installed.

In this phase the most relevant problems that were faced were, the script stops working as of a bug on the DevStack project as the association of the bridge br-ex (responsible for the routing of the external network to the internal networks of the OpenStack platform) to the network interface of the management server removes the DNS name resolution of the management server and when using a hyperkube version above version 15.0 the installation of Kubernetes fails. These problems were solved by, using a GitHub repository commit version of the DevStack project older than the one that was being used and using an older version of hyperkube.

This phase was the most time consuming because of all the configuration options that need to be learned and tweaked to fulfill the needs of the architecture solution and the limitations of hardware resources that were presented at the time of this phase implementation, e.g. the IST resources were so limited and overbooked, that the private hardware resources must be upgraded so the architecture solution implementation could continue.

After this, it is necessary to configure the management server to create and have access to the virtual instances that needed for the architecture solution implementation by loading the environment variables of the OpenStack platform onto the management server and creating a SSH key to be able to access the virtual instances. A series of configurations need to be done the OpenStack platform each time the it is deployed:

- Changing the DNS nameserver IPs on all default OpenStack networks.
- Creating a router to access the virtual machines.
- Creating ports in the router to all default OpenStack networks.
- Adding SSH keys to the OpenStack platform to be able to access the virtual instances.
- Adding or changing the rules on the default OpenStack security groups.
- Adding new operating system images to the OpenStack platform.

A problem of time consumption happens when these configurations of the OpenStack platform occur and the solution was using an automation tool such as Ansible [43], which is a Red Hat project that focuses on IT automation, using YAML has a standard to create template files, that can deploy and modify virtualized network infrastructures and their elements. The template file (also called ansible playbook) was created for configuration steps described above and is depicted on Annex D.

Now that the system is fully configured, it is time to use the Tacker project. The first thing that needs to be done is to register the OpenStack and Kubernetes VIMs onto tacker and for that it is necessary to create two YAML files that have the information needed to register both VIMS. The information is slightly different in both files as we can see in Annex E and Annex F. After the formulation and creation of the files it’s time to register the
VIMs onto Tacker via the OpenStack CLI or GUI, and depending on the commit version of Tacker, the registration of the Kubernetes VIMs cannot work due to a bug on the code, so the GitHub branch used in Tacker must be the master branch to fully pass these kind of problems, just because the branch is more often updated than the other ones.

3.1.3. Backend and frontend VNFs deployment and configuration

With the VIMs registered, it is necessary to register the backend and frontend VNFs by making one or two descriptor template files for each VIM. There are two ways to approach the configuration and deployment of the frontend VNF, one can be done manually where it is used a descriptor template file and all the configuration for the load balancer is done manually in terms of networking, failsafe protection and the software used for the load balancer or two where it is used a OpenStack additional project described on chapter two called Octavia. The Octavia project is a LBaaS where it shares the concept of XaaS[49], where anything can be called a service in a virtualization system. The configuration selected was the second one because the way that Kubernetes works with the OpenStack platform is by using the OpenStack networks via the kuryr-kubernetes project while the access to the Kubernetes Pods is done via an ingress Octavia LBaaS project controller.

The backend template files for each VIM are different. Each template file has its own configuration options as seen on Annex G and Annex H just because they use a different type of hardware and software virtualization (containers or virtual machines) that have different options for configurations, e.g. on a Kubernetes template file it is necessary to define a service type tag that can grant access from the external network to the application running on the containers.

The configuration of the VNFs can be done by:

- A bash script, where for that is necessary to use a specific image built with the OpenStack additional project diskimage-builder.
- Using a user_data tag on the template descriptor file, where configuration and installation commands can simulate like it was a bash script file.
- Using the management server by running directly the bash script file with the SSH command.

These configuration options are well documented but the safer to use would be the third option, just because the other two have problems with them. The first configuration option the diskimage-builder project has software bugs where it only builds successfully images of the latest versions of the operating systems and when using these images with Tacker the bash script does not run properly making the VNF unconfigurable. The second configuration option is a good option, but the template file organization and length can become quite unorganized and big due to inserting the bash script data onto the template descriptor file. In the third option the template descriptor file and bash script file are separated, and the script file is loaded via SSH using the management server for that matter.

The software tools used for the backend and frontend VNFs are:

- HAProxy software is used on the frontend VNFs, being the most documented, tested and versatile load balancing software.
- NGINX and PHP software are used on the backend VNFs.

The configuration script bash files for the backend VNFs can be viewed on Annex I of this dissertation report. The HAProxy software used on the frontend VNF is preinstalled with the Octavia OpenStack additional project and it is configured as a HTTP load balancer with a load balancing round-robin algorithm. The load balancer is also configured with a “health monitor” that checks the state of the backend servers. Layer seven policies that do redirection based on the path that is entered on the browser can be configured also, giving extra security to the backend servers. The backend VNFs will act as web servers that verify if the configurations on the load balancer are working as intended.

After loading the configuration files onto the VNFs, it is time to test the architecture solution implementation. In this phase the installation, deployment, connectivity and functionality are tested of the architecture solution. This phase will be described in more detail on the next section of the report.

A problem occurred in this phase were that if a test fails and it is necessary to reboot the management server the virtual network interfaces created by the DevStack OpenStack project does not persist over a system shutdown or reboot, so the solution is to remove the configurations and installations done on the management server. Gladly the DevStack developers thought of that and created two scripts called unstack.sh e clean.sh. The unstack.sh script stops all services associated with OpenStack and the clean.sh script cleans all configurations and installations done by DevStack on the management server. After that it is necessary to run the stack.sh file and to do all VNFs configuration and deployment.
4. Evaluation

The evaluation phase is the most important phase of all systems implementation, just because it validates all the work that was done. It also detects if something is not running how it should be, by testing all the elements in the system and giving out precious information to the administrator. With that information the administrator can monitor and fix all the elements that are not corresponding to the normal behaviour. The next subsections describe the tests performed on the architecture solution implementation.

4.1. Tests

The tests are divided on functional and performance tests. The tests validate if the functionality, failover and scalability of the implemented solution are working as intended. The table below describes in a short manner what tests were done on the implemented solution:

<table>
<thead>
<tr>
<th>Test number</th>
<th>Test short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>functional test: Verify VIMs and VNFs deployment and configuration</td>
</tr>
<tr>
<td>2</td>
<td>functional test: Verify frontend and backend VNFs connectivity</td>
</tr>
<tr>
<td>3</td>
<td>functional test: Verify frontend and backend VNFs functionality</td>
</tr>
<tr>
<td>4</td>
<td>functional test: Verify frontend and backend VNFs failover</td>
</tr>
<tr>
<td>5</td>
<td>performance test: Verify VNFs scalability</td>
</tr>
</tbody>
</table>

Table 1: Summary table of the evaluation tests

4.1.1. Tests 1 to 3

The tests one to three are done via a Python script that is depicted on Annex J, where issues one hundred GET requests to the VNF acting as load balancer and outputs the graph below:

Figure 18: Distribution of GET requests with four web servers
The algorithm chosen for these tests is the round robin algorithm and as it is seen on the graph above, all the one hundred requests are evenly distributed proving the web servers and load balancer functionality. The success of the web servers and load balancer functionality also validates the frontend and backend VNFs connectivity (test two) and the VIMs and VNFs deployment and configuration (test one).

4.1.2. Test 4

Test four is where the frontend failover testing is done via a terminal command (**openstack loadbalancer failover name_of_lb**) where it simulates with an interval of time a failover scenario that is performed on the load balancer. The command initiates the failover by destroying the virtual machine that hosts the load balancer, verifies that the load balancer no longer is available and performs the recoverability process of creating a new virtual machine using the load balancer configuration metadata. The backend failover is tested by creating a health monitor checker and shutting down a backend VNF virtual machine or container via the OpenStack CLI or GUI and verifying that the load balancer becomes aware and do not forward HTTP traffic to that specific virtual machine or container.

4.1.3. Test 5

Test 5 uses a scalability property on the VNFDs, where it defines how many replicas of a VM or container a system can make. To test this the script on **Annex J** was slightly modified where it calculates the GET response average time plus the computational time for each web server. The number of replicas was increased to three on site 2 and the script outputs the following graph and times:

![Figure 19: Distribution of GET requests with five web servers](image)

<table>
<thead>
<tr>
<th>site1-web1</th>
<th>site1-web2</th>
<th>site2-web1</th>
<th>site2-web2</th>
</tr>
</thead>
<tbody>
<tr>
<td>885ms</td>
<td>886ms</td>
<td>887ms</td>
<td>884ms</td>
</tr>
</tbody>
</table>

Table 2: Requests time with four web servers

<table>
<thead>
<tr>
<th>site1-web1</th>
<th>site1-web2</th>
<th>site2-web1</th>
<th>site2-web2</th>
<th>site2-web3</th>
</tr>
</thead>
<tbody>
<tr>
<td>847ms</td>
<td>847ms</td>
<td>847ms</td>
<td>846ms</td>
<td>846ms</td>
</tr>
</tbody>
</table>

Table 3: Requests time with five web servers
As it is seen on the graph above and tables, the time is reduced by forty milliseconds proving that the scalability performance is relevant and is working as intended.
5. Conclusion

All the technologies described on chapter two are still being heavily developed and complementing their core environment to using containers. Containers in comparison with virtual machines use less resources, meaning better overall performance for the workflow of a VNF solution. Nevertheless, the use of virtual machines means a better isolation for the adopted VNF solution that containers cannot deliver yet. That is why kata containers enables the merge of virtual machines and containers by adding the good features described from both virtual machines and containers.

The problems encountered while doing this dissertation were mainly, the documentation of the researched technologies that needs improvement on explaining how some of the projects work and what certain aspects of the projects do, e.g. to lost a lot of time searching for a solution when the installation script used by the DevStack project stops working. Unfortunately, due to limitations on the hardware resources it was not possible to test all the technologies described on chapter two and to scale the proposed architecture solution as it was intended. The projects used on the implementation of the proposed architecture are still being heavily developed, and still need optimization regarding merging the features of containers and virtual machines. Nevertheless, the implementation of the architecture was successful and describes well what is the expectation for the future of virtualized network functions and services.

5.1. Future work

For future work, if the resources permit, it is very important to implement and research other possible architecture scenarios. For example, creating a multi VNF network service where it deploys different VNFs with different networks graphs. With these virtual network graphs, it is possible to create virtual network paths and selecting witch VNFs are associated with each virtual network path.
References


[38] “Overview of Open Source Components in Cloudify | Cloudify Documentation Center.” [Online]. Available: https://docs.cloudify.co/4.3.0/about/manager_architecture/components/.


Annex A – VNFD example file

```yaml
1: tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0
2: description: Sample VNFD template mentioning possible values for each node.
3: metadata:
4: template_name: sample-tosca-vnfd-template-guide
5: topology_template:
6: node_templates:
7: VDU:
8:   type: tosca.nodes.nfv.VDU.Tacker
9:   capabilities:
10:     nfv_compute:
11:       properties:
12:         mem_page_size: [small, large, any, custom]
13:         cpu_allocation:
14:           cpu_affinity: [shared, dedicated]
15:           thread_allocation: [avoid, separate, isolate, prefer]
16:           socket_count: any integer
17:           core_count: any integer
18:           thread_count: any integer
19:           numa_node_count: any integer
20:       numa_nodes:
21:         node@: [id =>8, vcpus: [host CPU numbers], mem_size: => 0MB]
22:   properties:
23:     image: Image to be used in VDU
24:     flavor: Nova supported flavors
25:     availability_zone: available availability zone
26:     mem_size: in MB
27:     disk_size: in GB
28:     num_vcpus: any integer
29:     metadata:
30:     entry_schema:
31:     config_drive: [true, false]
32:     monitoring_policy:
33:       name: [noop, tcp, http-ping]
34:       parameters:
35:         monitoring_delay: delay time
36:         count: any integer
37:         interval: time to wait between monitoring
38:         timeout: monitoring timeout time
39:         actions: [failure: respawn, failure: terminate, failure: log]
40:         retry: number of retries
41:       port: specific port number if any
42:     config: Configuring the VDU as per the network function requirements
43:     mgmt_driver: [default=noop]
44:     service_type: type of network service to be done by VDU
45:     user_data: custom commands to be executed on VDU
46:     user_data_format: format of the commands
47:     key_name: user key
48:     artifacts:
49:       VNImage:
50:         type: tosca.artifacts.Deployment.Image.VM
51:         file: file to be used for image
52:       CP:
53:         type: tosca.nodes.nfv.CP.Tacker
54:         properties:
55:           management: [true, false]
56:           anti_spoofing_protection: [true, false]
57:           type: [srivn, vnic ]
58:           order: order of CP within a VDU
59:           security_groups: list of security groups
60:           requirements:
61:           - virtualLink:
62:             node: VL to link to
63:           - virtualBinding:
64:             node: VDU to bind to
65:       VL:
66:         type: tosca.nodes.nfv.VL
67:         properties:
68:           network_name: name of network to attach to
69:         vendor: Tacker
```
Annex B – Vagrant virtual machine configuration file

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  config.vm.define "osmgmt" do |vm|
    vm.box = "ubuntu-16.04-qemu"
    hostname = "osmgmt"
    provider "libvirt" do |lv|
      memory = "16384"
      cpus = 10
    end
    provision "shell", path: "copyfile.sh"
  end
end
```
Annex C – DevStack project configuration file

```yaml
# Local

# Customize the following HOST_IP based on your installation
HOST_IP=192.168.121.115
SERVICE_HOST=192.168.121.115
MYSQL_HOST=192.168.121.115
RABBIT_HOST=192.168.121.115
GLANCE_HOST_PORT=192.368.121.115:9292
ADMIN_PASSWORD=12secret34
DATABASE_PASSWORD=12secret34
SERVICE_PASSWORD=12secret34
IP_VERSION=4
SERVICE_VERSION=4

# Neutron options
#USE_SECGROUP=True
#FLOATING_RANGE="192.368.121.8/24"
#IP_ADDR_SAFE_TO_USE="10.0.1.0/24"
#Q_FLOATING_ALLOCATION_POOL=start=192.168.121.240,end=193.168.121.254
#PUBLIC_NETWORK_GATEWAY="192.168.121.1"
#PUBLIC_INTERFACE=eth0

# Logging
LOGFILE=/opt/stack/logs/stack.sh.log
VERBOSE=True
ENABLE_DEBUG_LOG_LEVEL=True
ENABLE_VERBOSE_LOG_LEVEL=True

# Neutron ML2 with OpenVSwitch
Q_PLUGIN=ml2
Q_AGENT=openvswitch

# Disable security groups
Q_USE_SECGROUP=False
LIBVIRT_FIREWALL_DRIVER=libvirt.virt.firewall.NoopFirewallDriver

# Enable heat, networking-sfc, barbican and mistral
enable_plugin heat https://opendev.org/openstack/heat
enable_plugin networking-sfc https://opendev.org/openstack/networking-sfc
enable_plugin barbican https://opendev.org/openstack/barbican
enable_plugin mistral https://opendev.org/openstack/mistral

# Ceilometer
CEILOMETER_PIPELINES_INTERVAL=300
enable_plugin ceilometer https://opendev.org/openstack/ceilometer
enable_plugin aodh https://opendev.org/openstack/aodh

# Tacker
enable_plugin tacker https://opendev.org/openstack/tacker

event_service n-novnc
event_service n-nauth
disable_service tempest
event_service

# Enable kuryr-kubernetes, docker, octavia
KUBERNETES_VIP=True
enable_plugin kuryr-kubernetes https://opendev.org/openstack/kuryr-kubernetes
enable_plugin octavia https://opendev.org/openstack/octavia
enable_plugin devstack-plugin-container https://opendev.org/openstack/devstack-plugin-container
KURYR_K8S_CLUSTER_IP_RANGE="10.10.0.0/16"
KURYR_HIPERKUBE_VERSION="v1.14.0"

[[post-config]/etc/neutron/dhcp_agent.ini]
[DEFAULT]
enable_isolated_metadata = True

[[post-config]/etc/octavia/agent.ini]
(controller-worker)
AMP_ACTIVE_RETRIES=9999
```

Annex D – OpenStack platform configuration file
Annex E – OpenStack VIM descriptor file

1 auth_url: 'http://192.168.121.115/identity'
2 username: admin
3 password: 12secret34
4 project_name: admin
5 domain_name: Default
6 user_domain_name: Default
7 project_domain_name: Default
8 cert_verify: 'False'

Annex F – Kubernetes VIM descriptor file

1 auth_url: "https://192.168.121.115:6443"
2 username: "admin"
3 password: "admin"
4 project_name: "default"
5 ssl_ca_cert: "-----BEGIN CERTIFICATE-----
6 HASH KEY
7 -----END CERTIFICATE-----"
8 type: "kubernetes"
Annex G – OpenStack backend descriptor file

```yaml
1 tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0
2 metadata:
3   template_name: site-backend-tosca-vnfd-for-vnfc
4
5 topology_template:
6   node_templates:
7     VDU1:
8       type: tosca.nodes.nfv.VDU.Tacker
9       properties:
10          name: site1-web1
11          image: ubuntu-18.04
12          flavor: ds512M
13          mgmt_driver: noop
14          key_name: ansible-key
15          config:
16            param0: key1
17            param1: key2
18
19     CP1:
20       type: tosca.nodes.nfv.CP.Tacker
21       properties:
22          ip_address: 10.10.0.12
23          management: true
24          anti_spoofing_protection: false
25       requirements:
26          - virtualLink:
27            node: VL1
28          - virtualBinding:
29            node: VDU1
30
31     VDU2:
32       type: tosca.nodes.nfv.VDU.Tacker
33       properties:
34          name: site1-web2
35          image: ubuntu-18.04
36          flavor: ds512M
37          mgmt_driver: noop
38          key_name: ansible-key
39          config:
40            param0: key1
41            param1: key2
42
43     CP2:
44       type: tosca.nodes.nfv.CP.Tacker
45       properties:
46          ip_address: 10.10.0.13
47          management: true
48          anti_spoofing_protection: false
49       requirements:
50          - virtualLink:
51            node: VL1
52          - virtualBinding:
53            node: VDU2
54
55     VL1:
56       type: tosca.nodes.nfv.VL
57       properties:
58          network_name: net0
59          vendor: Tacker
```
Annex H – Kubernetes web server descriptor file

tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: A sample containerized backend VNF

metadata:
  template_name: backendvnf-container-tosca-vnfd

topology_template:
  node_templates:
    VDU1:
      type: tosca.nodes.nfv.VDU.Tacker
      properties:
        namespace: default
        mapping_ports:
          - "80:80"
        service_type: LoadBalancer
        vnfcs:
          front_end:
            num_cpus: 0.5
            mem_size: 512 MB
            image: nginx
            ports:
              - "80"
        policies:
          - SP1:
            type: tosca.policies.tacker.Scaling
            targets: [VDU1]
            properties:
              min_instances: 1
              max_instances: 3
              target_cpu_utilization_percentage: 40
Annex I – Configuration script files for the load balancer and web servers

```
1 #!/bin/bash
2 # Load balancer configuration
3
4 # export DEBIAN_FRONTEND=noninteractive
5
6 sudo timedatectl set-timezone Europe/Lisbon
7 export up=/etc/hosts >dev/null
8
9 # sudo tee ~/etc/hosts >dev/null
10 # 127.0.0.1 localhost
11 # 192.168.1.1 site1-web
12 # 192.168.2.1 site2-web
13 # 192.168.1.1 site1-web1
14 # 192.168.1.1 site2-web1
15
16 # The following lines are desirable for IPv6 capable hosts
17 # ::1 localhost
18 # fido:0:1000:1000:1000:1000:1000:1000
19 # fido:0:1000:1000:1000:1000:1000:1000
20 # fido:0:1000:1000:1000:1000:1000:1000
21 # fido:0:1000:1000:1000:1000:1000:1000
22 # fido:0:1000:1000:1000:1000:1000:1000
23 # fido:0:1000:1000:1000:1000:1000:1000
24 # fido:0:1000:1000:1000:1000:1000:1000
25 # fido:0:1000:1000:1000:1000:1000:1000

26 # sudo tee ~/etc/haproxy/haproxy.cfg >dev/null
27 # Global
28 # log 127.0.0.1 local2
29 # chroot /var/lib/haproxy
30 # pidfile /var/run/haproxy.pid
31 # maxconn 4000
32 # user root
33 # group root
34 # daemon
35
36 # turn on stats unix socket
37 # stats socket /var/lib/haproxy/stats level admin
38
39 # defaults
40 # mode http
41 # log global
42 # option http-log
43 # option http-proxy-protocol
44 # option http-server-close
45 # option http-forwardfor
46 # option http-check-disable
47 # option tcp-check
48 # retries 3
49 # timeout http-request 10s
50 # timeout queue 1m
51 # timeout connect 1s
52 # timeout client 1m
53 # timeout server 1s
54 # timeout http-keep-alive 1s
55 # timeout check 1s
56 # maxconn 3000
57
58
59
60 # frontend http_front
61 # bind *:80
62 # stats enable
63 # stats url /stats
64 # acl db_manage_site1-web1 path -i -m beg/site1-web1.web
65 # use_backend mandos_site1-web1 if db_manage_site1-web1
66 # acl db_manage_site1-web2 path -i -m beg/site1-web2-web
67 # use_backend mandos_site1-web2 if db_manage_site1-web2
68 # acl db_manage_site2-web1 path -i -m beg/site2-web1-web
69 # use_backend mandos_site2-web1 if db_manage_site2-web1
70 # acl db_manage_site2-web2 path -i -m beg/site2-web2-web
71 # use_backend mandos_site2-web2 if db_manage_site2-web2
72 # default_backend http_back
73
74 # backend http_back
75
76 # balance roundrobin
77 # server site1-web1 192.168.1.12:80 check
78 # server site2-web1 192.168.1.13:80 check
79 # server site1-web2 192.168.1.12:80 check
80 # server site2-web2 192.168.1.13:80 check
81
82 # backend manados_site1-web1
83 # server site1-web1 192.168.1.12:80 check
84
85 # backend manados_site1-web2
86 # server site1-web1 192.168.1.13:80 check
87
88 # backend manados_site2-web1
89 # server site1-web2 192.168.1.12:80 check
90
91 # backend manados_site2-web2
92 # server site1-web2 192.168.1.13:80 check
93 # EOF
94
95 # sudo systemctl restart haproxy
```
Annex J – Python script used for tests 1 to 3

```python
import requests
import matplotlib.pyplot as plt
import timeit
from collections import defaultdict

requests_list = []
requests_time_list = []
res = defaultdict(list)

for i in range(100):
r = requests.get('http://192.168.121.254/
requests_list.append(r.content.decode('utf-8').rstrip())
requests_time_list.append((r.content.decode('utf-8').rstrip(), r.elapsed.total_seconds()))

for item, time in requests_time_list:
    res[item].append(time)

c = dict(x=requests_list.count(x)) for x in set(requests_list))
t = [{k, sum(v)) for k, v in res.items()}
print(t)

# x-coordinates of left sides of bars
left = [5, 10, 15, 20, 25]

# heights of bars
height = c.values()

# labels for bars
tick_label = [list(c.keys())[0], list(c.keys())[1], list(c.keys())[2], list(c.keys())[3],
list(c.keys())[4]]

# plotting a bar chart
plt.bar(left, height, tick_label = tick_label,
        width = 0.8, color = ['red', 'green'])

# naming the x-axis
plt.xlabel('Web Servers')

# naming the y-axis
plt.ylabel('Requests')

# plot title
plt.title('Requests distribution per web server')

# function to show the plot
plt.show()
exit
```