SUPERFLUIDITY

A SUPER-FLUID, CLOUD-NATIVE, CONVERGED EDGE SYSTEM

Research and Innovation Action GA 671566

DELIVERABLE D5.3:

PLATFORM API DESIGN AND IMPLEMENTATION

Deliverable Type:	Report
Dissemination Level:	PU
Contractual Date of Delivery to the EU:	31/01/2018
Actual Date of Delivery to the EU:	30/03/2018
Workpackage Contributing to the Deliverable:	WP5

Editor(s):	Christos Tselios (CITRIX)
Author(s):	Christos Tselios (CITRIX), Luis Tomas Bolivar (REDHAT), Jose Maria Roldan Gil (Telcaria), Carlos Martins Marques (ALB), Kevin Du (ONAPP), Stefano Salsano, Claudio Pisa, Francesco Lombardo, Luca Chiaraviglio (CNIT), Costin Raicu (UPB), Cyril Soldani (ULG), Bessem Sayadi (Nokia-BL FR)
Internal Reviewer(s)	George Tsolis (CITRIX)

Abstract: This deliverable provides an overview of the project's platform API design and implementation, along with the main principles on which it was based. In addition, we emphasize on the extension of the already available components that were enhanced under the auspices of Superfluidity project.

Keyword List: Unified Platform, Interfaces, API functions, Orchestrators, Virtualization, Containers, Networking

INDEX

List of Figures

List of Tables

Glossary

SUPERFLUIDITY DICTIONNARY	
TERM	DEFINITION
API	Application Programming Interface
CLI	Command Line Interface
CNI	Container Network Interface
CNCF	Cloud Native Computing Foundation
DNS	Domain Name System
DSCP	Differentiated Services Code Point
EC2	Elastic Compute Cloud
EPC	Enhanced Packet Core
ETSI	European Telecommunications Standards Institute
GTP	GPRS Tunning Protocol
GUI	Graphical User Interface
HOT	Heat Orchestration Template
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IPMI	Intelligent Platform Management Interface
ISG	Industry Specification Group
JSON	Javascript Object Notation
KVM	Kernel-based Virtual Machine
LBaaS	Load Balancing as-a-Service
LB	Load Balancer
LTE	Long Term Evolution
MAC	Media Access Control
MANO	Management and Orchestration
MEC	Multi-access Edge Computing
MEO	MEC Orchestrator
NAT	Network Address Translation
NEMO	NEtwork MOdeling
NIC	Network Interface Controller
NFV	Network Function Virtualization
NFVO	NFV Orchestrator
NFVI	NFV Infrastructure

NS	Network Service
NSD	Network Service Descriptor
ONOS	Open Network Operating System
OSM	Open Source MANO
OSS	Operation Support Systems
OvS	Open virtual Switch
QoE	Quality of Experience
QoS	Quality of Service
RAN	Radio Access Network
RBAC	Role-Based Access Control
RDCL	RFB Description and Composition Language
REST	Representational State Transfer
REE	RFB Execution Environment
RFB	Reusable Functional Blocks
SDN	Software Defined Networking
SEFL	Symbolic Execution Friendly Language
SGW	Serving Gateway
SNMP	Simple Network Management Protocol
SLA	Service Level Agreement
SSH	Secure Shell
TC	Traffic Control
TOF	Traffic Offloading Function
TOSCA	Topology and Orchestration Specification for Cloud Applications
URI	Uniform Resource Identifier
UM	User Management
VDU	Virtual Deployment Unit
VLAN	Virtual Local Area Network
VIM	Virtual Infrastructure Manager
VIP	Virtual IP
VNF	Virtualized Network Function
VNFD	VNF Descriptor
VNFM	VNF Manager
VM	Virtual Machine
YAQL	Yet Another Query Language
YAML	YAML Ain't Markup Language

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

1.1 Deliverable Rationale

The scope of this deliverable is to focus on the design and implementation of the platform's API. The API will be used by the architecture defined in WP3 to monitor platforms, and to instantiate, migrate and stop virtualized network services on the platform. The deliverable will further report on the integration of the Superfluidity platform's mechanisms and innovations including its optimal function allocation algorithms into a unified platform.

1.2 Executive summary

This deliverable is mainly focused on presenting the overall work related to Task 5.3, which merges all different components of Superfluidity under a coherent, uniform platform and exposes APIs according to the project's architecture developed in WP3. The work aimed to include the different contributions such as the service dynamic and allocation algorithms, the security considerations developed in Task 3.3 and the enhancements that were implemented under the auspices of Superfluidity for properly supporting containers to the overall architecture. In addition, the deliverable concentrates on the analysis that precedes the definition of the API to monitor the performance of the platform and instantiate, stop and migrate virtualized network services in order to be fully aligned with the dominant standards, as described by ETSI. Finally, the deliverable focuses on the implementation of such an API and its integration with existing frameworks such as OpenStack.

Before any development and integration activity begins, a careful analysis of the state of the art is necessary, in order to properly evaluate available components and tools that may be utilized by Superfluidity. In particular, the latest advances in relevant OpenStack projects are analysed and presented in Section 2 of this deliverable, namely OpenStack Compute and Neutron, HAProxy and Load Balancing-as-a-Service (LBaaS) [18], Heat, Mistral as well as OpenStack Telemetry. The two Platform Orchestration tools for NFV that have been integrated in the implementation of the Superfluidity architecture namely Open Source MANO, and ManageIQ are analysed in Section 3. In addition to OpenStack, an analysis of the current status of the ETSI NFV Architecture is presented in Section 4, emphasizing on the necessary network interfaces, management and configuration VNFs, the NFVI functions and the virtual resource interfaces that should be implemented towards an end-to-end solution. In the last paragraph of Section 4, the extensions to the ETSI NFV information model and the relevant descriptors are included, as proposed by the Superfluidity consortium. For properly designing and enhancing an architecture capable of being efficiently deployed on the network edge, some modules related to the Multi-access Edge Networking paradigm were also implemented, further improving the overall Superfluidity framework in terms of practicality and expandability.

In Section 5, this deliverable also presents a detailed analysis of Superfluidity mechanisms, algorithms and innovations that were integrated into the final prototype to further enhance Superfluidity's vision of an efficient 5G architecture, such as Kuryr, RDCL 3D and Symnet together with SEFL. A short subsection on the NEtwork MOdelling (NEMO) language, which has been investigated and evaluated by the project, but not selected for the integration is also reported.

2 Superfluidity Platform Building Blocks

2.1 Virtualization Platform

Virtualization is abstracting the system hardware (HW) resources to enable multiple instances to run independently on the shared hardware. The techniques of virtualization can be classified as: hypervisor, container and unikernel, which are illustrated and compared in Figure 1. The different techniques are developed and motivated by the requirements from cloud software engineering. To provide more flexible deployment model and consume less hardware resources the micro-service architecture is developed by moving from traditional virtual machine (VM) to containers. However, containers share a full host kernel and libraries, which makes the platform lost the isolation feature and open for vulnerabilities. To recover the security provided by hypervisor but also offer the advantages of micro-service architecture, unikernel is invented by removing all unnecessary code so that the runtime OS is compiled with just the libraries required to the software running on it. Unikernel can provide rapid scaling up and down based on utilization so it's a suitable solution to run Virtualized Network Functions (VNF).

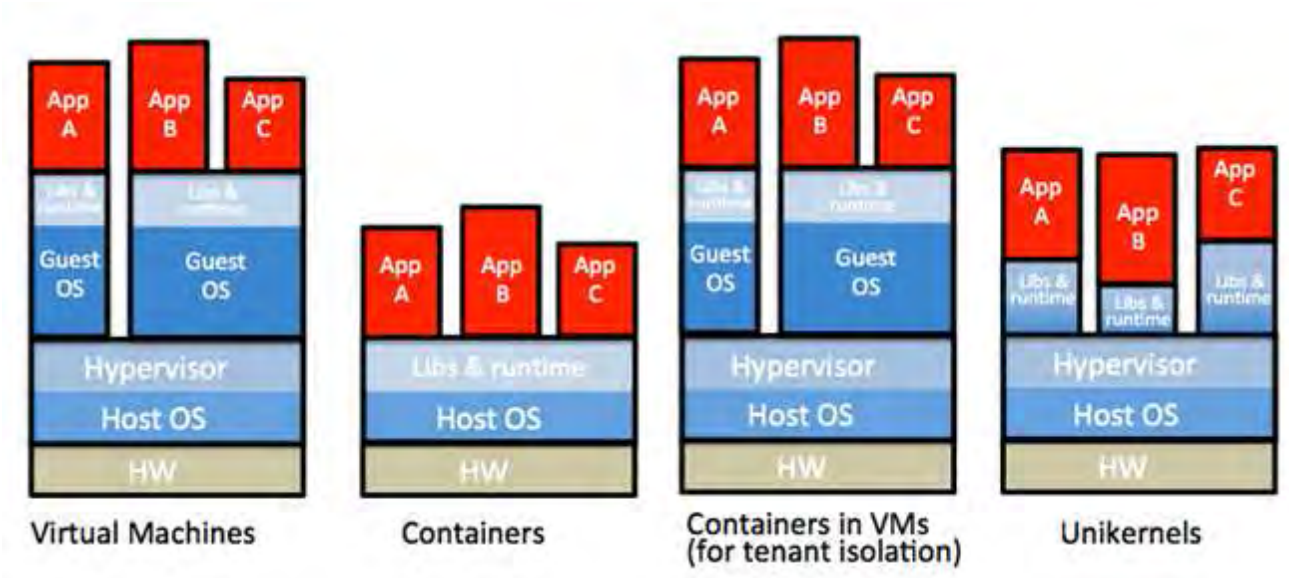

Figure 1: Techniques for Virtualization [1]

2.1.1 Hypervisor and KVM

A hypervisor is a dedicated software or firmware component that can virtualize system resources by utilizing highly efficient and sophisticated algorithms, thus allowing multiple operating systems running on different Virtual Machines (VMs) to share a single hardware host [1]. The hypervisor is actually in charge of all available resources, which are allocated accordingly, making sure that all VMs run independently without disrupting each other. A node on which any hypervisor operates creating


one or more virtual machines is often referred as host machine, while each of the virtual machines is called guest machine.

Hypervisors can be categorized as native/bare-metal (Type-1) or hosted (Type-2). Native hypervisors operate directly on the host machine's hardware to deploy and manage guest machines, while hosted hypervisors run on top of conventional operating systems, rendering the guest operating system an actual process of the host. Sometimes the distinction between the two types is not crystal clear; however, there is an apparent performance gain when using bare-metal hypervisor solutions. Specific performance requirements of the Superfluidity Virtualization Platform led towards adopting Kernel-based Virtual Machine (KVM) as the hypervisor of choice for experimenting onto, and evaluating all components and techniques currently under investigation.

Kernel-based Virtual Machine (KVM) [2] is a free, open-source virtualization solution, which enables advanced hypervisor attributes on the Linux kernel. It consists of a loadable kernel module, kvm.ko, which facilitates the core virtualization infrastructure and a processor-specific module kvm-intel.ko or kvm-am.ko for Intel and AMD processors, respectively. Upon loading the aforementioned kernel modules, KVM converts the Linux kernel into a bare metal hypervisor and leverages the advanced features of modern hardware, thus delivering unsurpassed performance levels [3].

By itself, KVM does not perform any emulation but exposes the /dev/kvm interface used by the user space host to set up the guest VM's address space and feed the corresponding simulated I/O. Besides performance, the integration with the operating system kernel provides KVM significant security and scalability benefits, especially since industry-standard x86 architecture processors and systems have grown increasingly more powerful in terms of processing, memory and I/O. Physical resources can be easily divided into sufficient pools of virtual resources that may handle larger workloads than before. Figure 2 presents the basic elements of the KVM architecture and their interaction through dedicated interfaces.

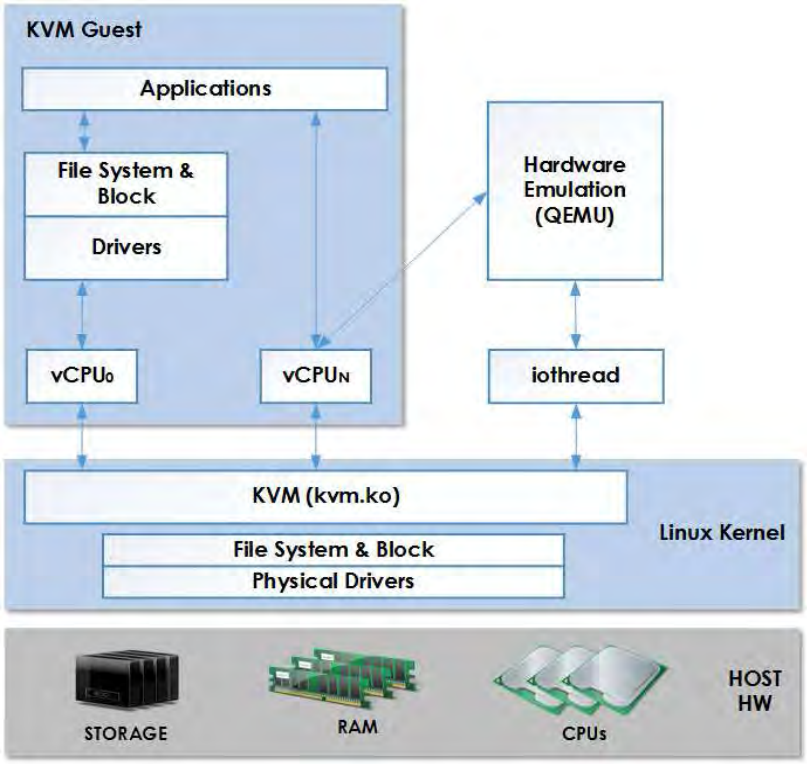

Figure 2: KVM Architecture

One of the main drivers behind the specific KVM model design was harnessing the benefits of existing Linux functionality, where the kernel was responsible for all hardware-related resource management while the KVM acted as a driver handling the newly-introduced virtualization instructions. This allowed KVM to integrate all features deployed in the Linux kernel over time, such as improvements in the CPU scheduler, efficient memory, storage and power management.

Nowadays, there are several projects that use KVM as the default hypervisor, with OpenStack being the most popular one. The use cases under investigation in the auspices of those projects involve large-scale deployments of KVM hosts, often with several VMs per deployment. As guest OSes grow larger in terms of memory and virtual CPU cores, the downtime trade-off for live migration increases. The necessity for low latency network packet processing needed by Telcos for the upcoming 5G era, in conjunction with additional security and hypervisor footprint limitation concerns, shape an ecosystem in which KVM is likely to thrive, therefore Superfluidity project partners will thoroughly inspect its capabilities.

2.1.2 Containers and Docker

Virtualisation via containers is known as containerization, which enables software applications to run on virtual operating systems. Containerization is a solution to the problem of how to run a software application smoothly when moved from one environment to another. Containers use operating systems features to package applications together with all their dependencies. A container image is

a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings [1]. Therefore, virtualisation is implemented in this case at libraries and runtime level. Containers do not need a hypervisor, they share the same operating system, libs and runtime. These shared components are read-only, while each container has its own specific access point for writing them. Containers represent an efficient method for building and running applications under micro-services architecture. Containers can be instantiated faster than virtual machines, but do not have the same level of isolation provided by the hypervisor. For this reason, to enable multitenancy, containers are usually isolated inside virtual machines, as shown in Figure 1.

Docker is the most popular and widely-used container management system [63] (Docker, n.d.). Docker introduced an entire ecosystem for managing containers, including a highly efficient and layered container image model, global and local container registries, CLI (command line interface) and clean Representational State Transfer (REST) API, and many other functions. Docker is available on multiple platforms, on cloud including Amazon Web Services and Microsoft Azure, and on premises supporting desktop and server with various operating systems. Docker uses a client-server architecture which contains the following major components: the Docker daemon, the REST API and the CLI client, as shown in Figure 3. The CLI client uses the REST API to control or interact with the Docker daemon, which creates and manages Docker objects such as images, containers, networks and volumes. The Docker registry stores Docker images. An image is a read-only template with instructions for creating a Docker container. A container is a runnable instance of an image. In Superfluidity project the containerised services are implemented using Docker.

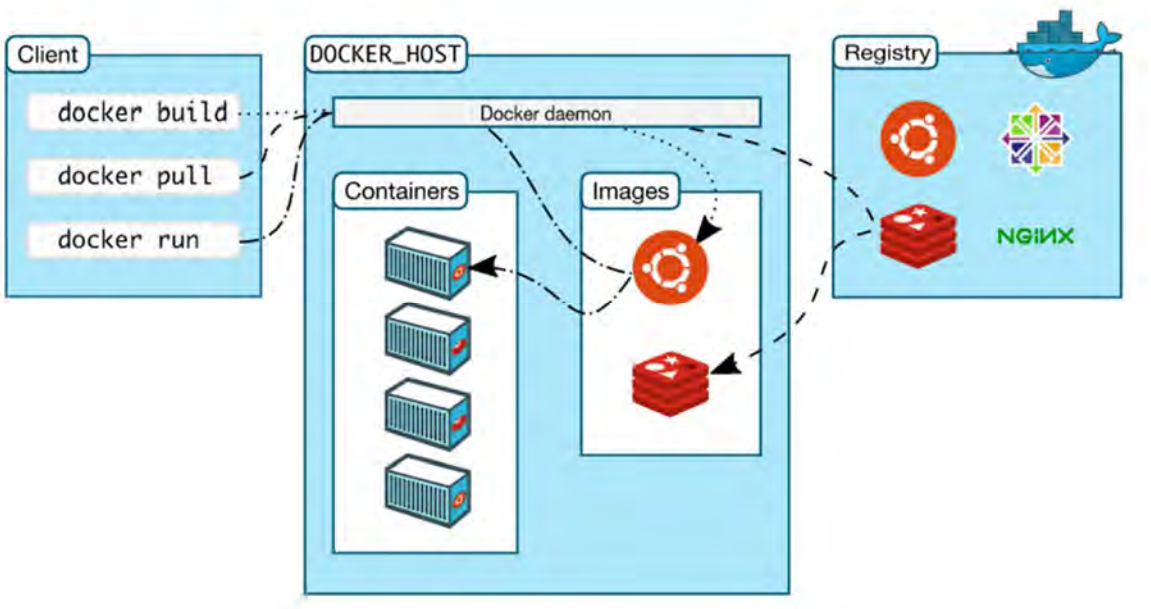

Figure 3: Docker Architecture

2.1.3 Unikernels

Virtualisation via VMs enables the software applications isolated in operating-system level but running on the shared hardware. The operating-system virtualisation is useful, but it adds another software stack which might be unnecessary, including support for old physical hardware and irrelevant optimisations on hardware. Despite these flexible layers of software support, most deployed VMs ultimately perform a single function such as web server or database. The usage of VMs is shifting toward hosting single-purpose applications. The modern hypervisor provides a resource abstraction that can scale dynamically, both vertically by adding memory and cores, and horizontally by spawning more VMs. However, many operating systems cannot support the capability provided by hypervisor and the boot time is delayed by the large number of unnecessary software stack.

Unikernel is a new workload theory of an enterprise software stack, one that promotes the qualities needed to create and radically improve the workloads in the cloud. From the perspective of the end user, the userspace traditionally comprises the code to be run, while the kernel space includes the code that needs to exist for the userspace code to work. Unikernel is combining the application itself and some libraries from the operating system (OS) into a single binary, which can run in a shared address space and execute on top of a hypervisor.

In unikernel, the application and the kernel service can run inside a unique process. There is no need for another layer of processes, threads, context switching, synchronisation primitives, etc. The whole process explicitly accesses the hypervisor on the host through a unique set of interfaces. Since in a virtualised environment, the entire process cannot be allowed to perform privileged instructions.

Since the unikernel is optimised to be run in a virtualised environment, it is built for the specific application including only the requested OS features. For instance, if the application logic uses only the network libraries from the OS, the filesystem support is not included into the unikernel binary. As a result, the global attack surface of the unikernel is much smaller that makes the applications much more secure.

2.2 Virtual Infrastructure Management

A Virtual Infrastructure Manager (VIM) can be defined as the virtualization and management layer for any cloud deployment offering primarily computational, networking and storage services. VIM coordinates physical hardware resources thus simplifying the creation of virtual server instances. It can also be used to manage a range of virtual resources across multiple physical servers and delivers centralized administration of the underlying infrastructure, including creating, storing, monitoring, backing up VMs hosting a variety of applications. For instance, a VIM may create and manage multiple instances of a hypervisor across different physical servers and occasionally facilitate virtual server migration amongst physical nodes.

VIM plays a vital role in the Infrastructure-as-a-Service (IaaS) cloud deployment model, where system administrators can provision processing, storage, network or other fundamental computing resources and seamlessly deploy and run arbitrary software, spanning from simple applications to modern operating systems. VIM software allows pooling and sharing of resources thus providing an enhanced degree of granularity in every service.

2.2.1 Openstack

OpenStack is a cloud operating system that controls large pools of compute, storage and networking resources throughout a datacenter, all managed over a dashboard that gives administrators total control while empowering all connected users to provision resources through a web interface. One may describe OpenStack as a combination of open source tools called projects that use pooled virtual resources to build and manage private clouds. OpenStack's popularity derives from its inherent support of several proprietary and open source technologies, making it ideal for heterogeneous infrastructure deployments [5]. It offers a highly modular architecture based on distinct elements for Compute, Networking and Storage service support, while each and every function can be managed using command-line tools or a highly efficient RESTful API, besides the previously mentioned web interface. Additional elements also provide Identity and Image services, while other optional projects can be bundled together to create unique deployable clouds. OpenStack modules have certain legacy codenames marking the original project they derived from. The official documentation is often referring to these modules via their codenames, therefore they are included in this section. The Superfluidity testbed is currently based on the previous OpenStack release, the architecture of which is presented in Figure 4.

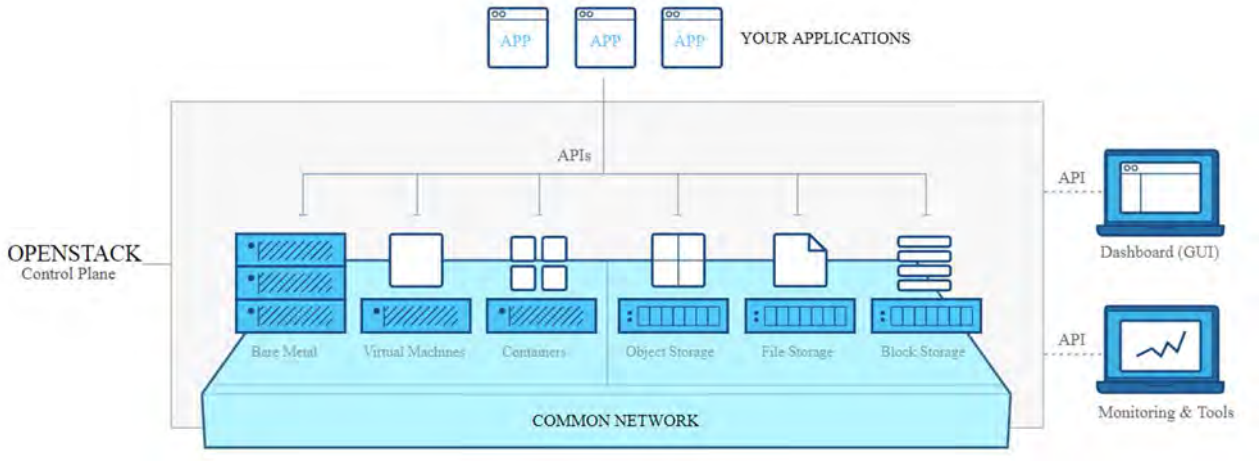

Figure 4: OpenStack High Level Architecture

2.2.2 Compute

Compute module (codename Nova) is a highly sophisticated cloud controller designed to provide massively scalable, on demand access to large pools of compute resources. Nova scales horizontally


and has the ability to work with all sorts of hardware setups, from bare-metal to high performance computing configurations, as well as the majority of the available virtualization technologies [7]. Its functionality is constantly extended through auxiliary projects and supplementary modules, such as the Ironic Project [8] that is implementing the notion of treating physical machines in the same way to on demand VMs, or the Magnum API service [10] that aims to introduce seamless container functionality thus delivering optimal quality of experience build on top of Nova.

2.2.3 Networking

2.2.3.1 Neutron

Neutron is an OpenStack Project which provides “networking as a service” between interface devices managed by interconnected OpenStack services. It implements mechanisms for pluggable, scalable, API-driven network and IP management on demand, through technology-agnostic network abstractions [11]. This ensures that networking would never become a limiting factor or bottleneck despite the highly demanding operational environment where node numbers, routing configurations and security rules may quickly escalate to over six figure numbers. In such an environment, traditional network management techniques fall short on providing a truly scalable and automated method of control, constantly supporting user’s ever-growing expectation for flexibility with quicker provisioning. Neutron supports floating IPs that enable dynamic traffic rerouting, load balancing features, software-defined-networking (SDN) technology like OpenFlow [4] and a vast extension framework of different back-ends called "plugins" offering a constantly growing variety of networking technologies, so that third-party network services can be seamlessly deployed and managed.

2.2.3.2 QoS and Role-Based Access Control (RBAC)

Quality of Service (QoS) commonly refers to the minimum service level that an application must provide to the users, for example, in a web server it could mean serving the user requests within a certain time-interval, e.g., less than 1 second. However, with the current trend of moving more and more applications to the cloud, applications are not running on dedicated server anymore, therefore making assumptions about performance more difficult as they may be affected by other co-located applications, an effect known as “noisy neighbour”. More specifically this term refers to the fact that one tenant in a cloud computing infrastructure can monopolize resources of a server, such as bandwidth, disk, I/O or CPU and negatively affect other tenants' performance when co-located in the same server.

While for CPU isolation there are already methods in place such as creating VMs with a defined size, or using virtual CPU to physical CPU pinning to avoid that some applications share cores in the same servers, there are still some other shared resources that may lead to interference between co-located VMs, such as L3 caches or the network itself. Due to the networking focus of Superfluidity, we focus


on the QoS from the network perspective. In such environment, QoS refers to the resource control system that guarantees certain network requirements such as bandwidth, latency, reliability in order to satisfy a Service Level Agreement (SLA) between an application provider and end users. One example of an application requiring this kind of assurance could be Voice over IP or video streaming, as their traffic needs to be transmitted with some minimal bandwidth constraints. For such a case, on a system without network QoS management, all traffic would be transmitted in a "best-effort" manner making it impossible to guarantee service delivery to customers as co-located applications can impact their performance if they are congesting the network.

In physical networks there were already ways of providing this – although not so flexible – by making network devices such as switches and routers to mark traffic. This enabled the option of prioritizing a flow over another to fulfil the QoS conditions agreed at the SLAs. However, again, with the raise of overlay and software defined networks at cloud platforms, a more flexible way of ensuring QoS was needed, especially at the edge network. There are already different mechanisms, such as OvS (which is presented in a following paragraph) min, max or Linux Traffic Control (TC), but there is no industry standard regarding the way of expressing bandwidth guarantees. Consequently, our goal is to enable cloud administrators to better control the network resources by allowing network tuning to specific application types and being able to provide different SLAs.

2.2.3.2.1 Features

We extended Neutron OpenStack to enable QoS related actions, such as limiting or prioritizing traffic. Note this is not yet another traffic shaping policy, but a framework where new policies can be easily included. To accomplish this, the QoS is coded as an advanced service plug-in, decoupled from the rest of the OpenStack Networking code on multiple levels, available through the ml2 extension driver.

The QoS API allows the implementation of policies, which are collections of rules to be applied on Neutron ports or networks. These policies allow the differentiation between different tenants or even applications, as they can be applied either at complete networks or per port basis, respectively. For instance, two policies can be created, one named gold and another one named bronze with different features (e.g., max bandwidth) and that will enforce different QoS needs where they are applied.

As shown in the next figure, policies are made of QoS Rules. These rules can be of different nature, known as QoS Rule Types, and as explained above, new rule types can be easily included.

So far, we have two QoS rule types are already implemented:

- QoS Bandwidth Limit: This ensures that a VM connected to a port will only be allowed to transfer at the specified maximum bandwidth, therefore limiting the impact it has in other co-located VMs, regardless of the tenant it belongs to. Note that, thanks to the design focus

on extendibility, we implemented the support for OvS and push it to the OpenStack open source community and other companies already extending the QoS bandwidth limit rule to also cover Linux bridges.

- QoS DSCP Marking: This will mark the outgoing packets from the VM connected to the port with the chosen DSCP mark, so that different network traffic prioritization can be subsequently applied.

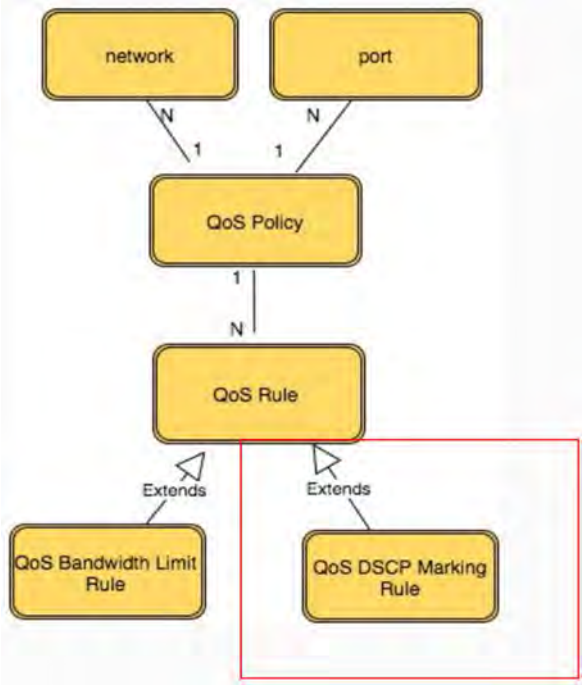

Figure 5: QoS Rules in Policies

On the other hand, besides the specific work done in the QoS API implementation, a parallel effort was made to increase the flexibility in the way the QoS policies could be apply. By default, when a policy is created, and similarly with other OpenStack functionality, such as security groups, it can be used either by the tenant who created it, or make it public, so that all tenants can use it. This does not provide the flexibility needed for applying QoS management in the 5G/Edge cloud environments. To overcome this restriction, we implemented the Role-Based Access Control (RBAC) policy framework that enables both operators and users to grant access to resources for specific projects, i.e., it is not a binary sharing (either all or none), but it can be decided with more granularity which tenants have access to what policies.

2.2.3.2.2 Usage

In order to use the QoS features, the below need to be configured:

- On the server side:
  - Enable qos service in service_plugins (neutron.conf)
  - Set the needed notification_drivers in [qos] section (message_queue is the default)

    - For ml2, add 'qos' to extension_drivers in [ml2] section
- On L2 agent side:
    - Add 'qos' to extensions in [agent] section

Moreover, to enable during QoS devstack installation, update the local.conf to include:

- enable_plugin neutron git://git.openstack.org/openstack/neutron
- enable_serivce q-qos

For more information visit: http://docs.openstack.org/mitaka/networking-guide/config-qos.html

Once it is up and running, a typical workflow is:

- Create a policy
    - neutron qos-policy-create POLICY_NAME
- Add rules to the policy. For instance, to limit max bandwidth:
    - neutron qos-bandwidth-limit-rule-create POLICY_NAME --max-kbps 3000 --max-burst-kbps 300
- Associate the policy to a network or a port
    - neutron net-update NET_NAME --qos-policy POLICY_NAME
    - neutron port-update PORT_ID --qos-policy POLICY_NAME
- [Optional] Change the policy online to immediately propagate it to the ports. This could be either detach the QoS policy from the network/port:
    - neutron net-update PORT_ID --no-qos-policy
    - neutron port-update PORT_ID --no-qos-policy

  Or modify the rules associated to the policies:
    - neutron qos-bandwidth-limit-rule-update RULE_ID POLICY_NAME --max-kbps 2000 --max-burst-kbps 200

Note for DSCP marking the process will be exactly the same, but using the DSCP marking API for creating the rules to be associated to the policies. For instance:

- neutron qos-dscp-marking-rule-create POLICY_NAME --dscp-mark 26

Finally, if the flexibility given by RBAC wants to be used, so that a defined policy may be shared with a specific tenant, the next steps are needed:

- neutron rbac-create --target-tenant TENANT_ID --actions access_as_shared --type qos-policy POLICY_ID

Where the target tenant is the project that requires access to the QoS policy, the action specifies what the project is allowed to do, and the type is the type of resource being shared, in this case the QoS policy.

2.2.3.3 OvS

Open vSwitch (OvS) is a production quality, multilayer software switch designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols [6]. Licensed under the open source Apache 2.0 license, OvS is designed to support distribution across multiple physical servers, in addition to exposing standard control and visibility interfaces to the virtual networking layer and operate both as a soft switch within the hypervisor or as the hardware's control stack. Written in platform-independent C, it is easily ported to a variety of environments offering support for advanced feature sets such as: (i) standard 802.1Q VLAN model with trunk and access ports (ii) NIC bonding (iii) Quality of Service (QoS) configuration and policing (iv) automated control through OpenFlow (v) increased visibility and monitoring using NetFlow [9] and sFlow (vi) 802.1ag connectivity fault management (vii) high performance forwarding using a dedicated Linux kernel module.

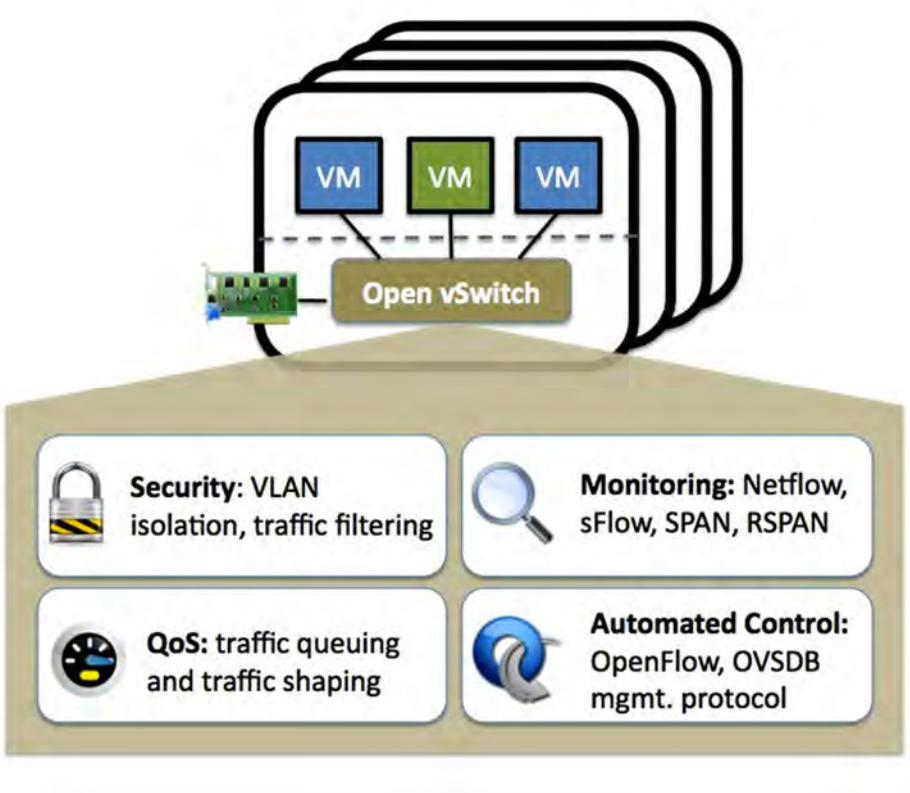

Figure 6: Open vSwitch [6]

Open vSwitch is ideal for multi-server virtualization deployments, a stack where the build-in L2 switch (the Linux Bridge) of Linux-based hypervisors fails to deliver optimal performance. These environments are often characterized by highly dynamic end-points, specific logical abstractions and finally task offloading to special purpose switching hardware. The following characteristics and design considerations help OvS to cope with the exceptional requirements of any networking setup, as described in [12].

The mobility of state

All network state associated with a network entity (i.e. a VM) should be easily transferred between different hosts. The network state may include "soft state" such as L2 learning table entries, L3 forwarding and routing policy states, monitoring configuration and QoS policies. OvS supports configuration and network state migration between instances through a real data-model that records its overall status. This approach also facilitates the development of structured automation systems.

Responding to network dynamics

Virtual environments are often characterized by high change rates. OvS supports several features that allow a network control system to adapt and respond to topology alterations. The dominant one is a network state database (OVSDB) that records topology changes through remote triggers, along with auxiliary accounting and monitoring protocols such as NetFlow [14] and sFlow [15]. Finally, OvS supports OpenFlow as a method of exporting remote access to control traffic.

Logical tag maintenance

Distributed virtual switches maintain logical context within the network through appending or manipulating tags in network packets. These tags can be used to uniquely identify interconnected nodes thus getting an overview of the actual topology. OvS supports multiple methods for specifying and maintaining tagging rules, all of which are accessible to a remote process for orchestration and stored in an optimized manner. This allows dynamic reconfiguration of all tags in case a node migrates from one datacenter to another.

Hardware integration

As stated in [12], Open vSwitch's forwarding path (the in-kernel datapath) is designed to be amenable to "offloading" packet processing to hardware chipsets, whether housed in a classic hardware switch chassis or in an end-host NIC. This allows for the OvS control path to be able to both control a pure software implementation or a hardware switch. The advantage of hardware integration is not limited to performance only. When physical switches follow the same abstractions as OvS, both bare-metal and virtualized hosting environments can be managed using identical mechanisms for automated network control.

OvS can be integrated to OpenStack Networking service using the ML2 plugin. However, since this integration involves several different entities, certain restrictions do apply, mostly related to available virtual resources and pre-deployed OpenStack components, as described in [13]. In particular, one Controller node, one Network node and two Compute nodes are needed. The Controller node requires one dedicated interface for management networking purposes, the Compute nodes need three different interfaces for Management, Tunnelling and VLAN networks respectively, while the Network node must also have an additional interface for the necessary external network.

2.2.3.4 Kuryr

As highlighted in [59], 5G deployments may need a mix of VMs and containers to fully take advantage of different features and the special needs of certain applications/middleboxes.

To provide a common infrastructure for both VMs and containers, the problem is not just how to create computational resources, be it VMs or containers, but also how to connect these computational resources among themselves and to the users, in other words, networking. This, in addition to Superfluidity project targeting quick provisioning at 5G deployments, led us to the need to further advance in the container networking and its integration in OpenStack environment.

To accomplish this, we have worked on a recent project in OpenStack named Kuryr (specially focusing on the Kubernetes section, but not solely). Kuryr tries to leverage the abstraction and all the hard work previously done in Neutron, and its plugins and services, and use that to provide production grade networking for containers use cases. In a nutshell, Kuryr aims to be the "integration bridge" between the two communities, containers and VMs networking, avoiding that each Neutron plug-in or solution needs to find and close the gaps independently.

As regards to Kuryr implementation, it mainly relies on Kubernetes and Neutron APIs to perform its courier task. It listens Kubernetes API, watching for specific events, such as containers creation, and then maps the container networking abstraction to the Neutron API. This enables the consumers to choose the vendor and keep one high quality API free of vendor lock-in, which in turn allows to bring container and VM networking together under one API.

In more details, Kuryr is composed of 2 main components:

Kuryr-controller: This is the component in charge of watching for Kubernetes events and translating them into Neutron resources. Once the Neutron resources are created, the Kubernetes API is used to communicate with the Kuryr-cni. This is performed by using Kubernetes annotations. For example, once a Neutron port is created for the container, the information that the kuryr-cni will need to later plug that container into the Neutron network is annotated into the container object by using the Kubernetes API.

Kuryr-cni: This component is triggered by the Kubelet component to handle the networking plumbing for the container. It is in charge of connecting the container to the network, based on the kuryr driver being used. This component monitors the Kubernetes API in order to discover when the pod created by the kuryr-controller is active, and therefore the complete networking process is ready.

As part of Superfluidity project, we have worked on extending kuryr to make it more suitable for 5G deployments. Among others, we have worked on:

- Enabling nested deployments where containers are running on top of OpenStack VMs by creating new kuryr-controller drivers that are able to avoiding double encapsulation with the


consequent latency reduction. This was built upon Neutron functionality for Trunk ports, and therefore also relying on Neutron API.

- Improving scalability by using ports pool at Kuryr side that minimizes the amount of calls to Neutron, reduces the load on the Neutron server, and speeds up containers boot up time. This again was relying on different functions available at Neutron API to perform certain operation in bulk request -- as well as to keeping some information on Kuryr side about the created resources to skip follow up calls to Neutron.
- Improving Kuryr efficiency by introducing CNI split capabilities that improves the containers plug-in workflow into the Neutron networks by reducing the amount of process needed.
- Improving usability by containerizing Kuryr components for easy deployment
- Improving resiliency by adding CI process that ensures previous features keep working when adding new ones

Additional details on the kuryr activities can be found in Deliverable D6.1 [59]

2.2.4 Load Balancing

2.2.4.1 Load Balancing-as-a-Service

As described in a previous paragraph, Kuryr allows to map the container networking abstraction to the Neutron API, enabling the consumers to choose the vendor and keep one high quality API free of vendor lock-in, bringing container and VM networking together under one API. To offer the aforementioned load balancing capabilities also to the container ecosystem, we have worked on Kuryr extensions to handle OpenStack Load Balancing services (both Neutron LBaaSV2 [19] and Octavia load balancers as a service) and enabling its utilization from the Kubernetes/OpenShift side.

To do that, again we have relied on Neutron APIs to handle the load balancing life cycle, i.e., to create the OpenStack load balancer, attach the corresponding listener with the right protocol and ports, create a pool of members, and add/remove the members to the pool.

This part solves the Kuryr to Neutron LBaaS interaction side. But we also had to work on the Kubernetes to Kuryr side to substitute the standard Kubernetes services (in this case cluster and load balancer types) by Neutron LBaaS resources, in this case:

- Kubernetes SVC == Neutron LBaaS
- Kubernetes SVC endpoints == Neutron LBaaS members

To accomplish this, kuryr-controller also started watching service and endpoints event from Kubernetes API, so that the needed actions are triggered, in this case:

- Calls to neutron API to create the needed load balancer, listeners, pools and add the members. Note the members are the neutron ports associated to the containers.
- Once the resources are ready, annotate the Kubernetes resources (in this case endpoints and services) with the corresponding neutron information.

Finally, note this integration is suitable for both Neutron LBaaSV2 as well as Octavia, thanks to using the OpenStack APIs. The only differences to set up one or another is the way to deploy the environment, where different configuration options may be needed -- such as applying the right Neutron security groups or deciding about the Octavia load balancing mode (i.e., L2 or L3 level).

More information about this effort can be found in Deliverable D6.1 [59].

2.2.4.2 HAProxy

HAProxy is a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based scheduler. Designed with a data forwarding orientation, its architecture is optimized to move data as fast as possible with the least possible operations by implementing a layered model of distinct bypass mechanisms. This approach ensures that data will be forwarded by the most appropriate level, rendering it suitable for TCP and HTTP-based applications where introducing additional encapsulation comes with a great computational and complexity cost. HAProxy only requires the haproxy [15] executable and the necessary configuration file to run. The configuration file is parsed before starting, then HAProxy tries to bind all listening sockets and once initiated it processes incoming connections, periodically checks the server's status and exchanges information with other HAProxy nodes.

HAProxy offers a fairly complete set of load balancing features with several supported load balancing algorithms, dynamic weight and slow-start support, and a large number of internal metrics for creating advanced load balancing strategies. Some of the algorithms include **round-robin**, (for short connections, pick each server in turn), **leastconn** (for long connections, pick the least recently used of the servers with the lowest connection count), **source** (for SSL farms or terminal server farms, the server directly depends on the client's source address), **uri** (for HTTP caches, the server directly depends on the HTTP URI), **hdr** (the server directly depends on the contents of a specific HTTP header field), **first** (for short-lived virtual machines, all connections are packed on the smallest possible subset of servers so that unused ones can be powered down) [16].

OpenStack fully supports HAProxy deployment in its controller nodes where each instance of the software configures its frontend to accept connections only to the virtual IP (VIP) address. The HAProxy backend (termination point) is a list of all the IP addresses of instances for load balances. When integrated with OpenStack, HAProxy provides a fast and reliable HTTP reverse proxy and load balancer for TCP or HTTP applications. It is particularly suited for web crawling under very high loads


while needing persistence or Layer 7 processing and can realistically support tens of thousands of connections with recent hardware [17]. An alternative LB implementation is OpenStack Octavia [55].

2.2.5 Heat

Heat is the cornerstone of the overall OpenStack orchestration project [20]. It implements an orchestration engine allowing users to launch multiple composite cloud applications through detailed deployment descriptions in specific text files called *templates* that can be treated as code. A Heat template describes the infrastructure a cloud application requires to become operational, such as servers, floating IPs, volumes, security groups or users, as well as the relationships between them. This pre-defined correlation of all application elements enables the Heat engine to interact with the virtual hardware through the OpenStack API towards creating the proper deployment environment. Heat manages the whole lifecycle of the application, therefore when infrastructure changes are necessary, users must only modify the template and use its latest version to update the existing stack. All corresponding changes are handled by Heat, which will also delete the reserved resources once the application lifespan ends. Heat primarily manages infrastructure; however, templates integrate well with software management tools such as Puppet [23], Chef [24] and Ansible [56] while also providing an autoscalling service.

Heat was born as the counterpart to the CloudFormation service in AWS. It accepts AWS templates and provides a compatible API, yet in recent OpenStack releases evolved to a new format called Heat Orchestration Template (HOT) with nicer template syntax and new features not supported by its competitor. HOT is considered reliable, supported and standardized as of OpenStack Icehouse (05/2013) release, offering extensive backwards compatibility. Since OpenStack Juno (10/2014) release, Heat supports multiple different versions of the HOT specification [21].

```
heat_template_version: 2016-10-14
description:
  # a description of the template
parameter_groups:
  # a declaration of input parameter groups and order
parameters:
  # declaration of input parameters
resources:
  # declaration of template resources
outputs:
  # declaration of output parameters
conditions:
  # declaration of conditions
```

Figure 7: Heat Orchestration Template

Table 2: Heat Orchestration Template structure

KEY NAME	DESCRIPTION
heat_template_version	This key value indicates that the YAML document is a HOT template of the specified version
description	This optional key allows for giving a description of the template, or the workload that can be deployed using the template
parameter_groups	This section allows specifying how the input parameters should be grouped and the order to provide the parameters in. This is an optional section.
parameters	This section allows for specifying input parameters that must be provided when instantiating the template. This is an optional section.
resources	This section contains the declaration of the single resources of the template. This section with at least one resource should be defined in any HOT template, or the template would not do anything when being instantiated.
outputs	This section allows for specifying output parameters available to users once the template has been instantiated. This is an optional section.
conditions	This optional section includes statements which can be used to restrict when a resource is created or when a property is defined.

2.2.6 Mistral

Mistral [25] can be defined as a simple yet scalable workflow service for cloud automation, featuring an intuitive YAML-based workflow definition language, a REST API for operating the workflows and an extensive set of actions including OpenStack operation, REST HTTP call and SSH protocol support. Mistral mostly targets administrators who are automating their operation and maintenance procedures, and integrating private clouds with their broader infrastructure. A user typically assembles a workflow using the YAML-based language and uploads the workload definition to Mistral via its REST API. Then the user can manually initiate the uploaded workflow using the same API or configure a trigger to start the workflow based on a specific event. Mistral typically provides all the necessary control points to manage task execution (i.e. suspend/resume) and observe their state, for instance, to find out whether a particular task or a sequence of tasks has successfully finished or failed.

As described in the previous paragraph, Mistral intends to add capabilities resembling to “Cloud Cron” to OpenStack, thus facilitating periodic cloud task scheduling and alleviate automated business functions consisting of multiple distributed processing steps. For efficiently tackling all issues that rise in the process, Mistral workflow service introduces the Domain Specific Language (DSL) v2 (as of 04/2015) which besides being based on YAML [22], takes advantage of additional query languages, for instance YAQL [26] and Jinja2 [27], to define expressions in workflow and action definitions.

Mistral DSL v2 introduces different workflow types and the structure of each workflow type varies according to its semantics. Basically, workflow type encapsulates workflow processing logic, a set of meta rules defining how all workflows of this type should work. Currently, Mistral provides two workflow types: Direct workflow, where each task starts based on the previous task's result, and Reverse workflow, where all relationships in the workflow task graph are dependencies. When executing reverse workflows, Mistral Engine must recursively identify all dependencies that need to be completed first. An example workflow which simply sends a command to OpenStack Compute service to start creating a VM and wait until the VM is created using a special "retry" policy is presented in

```
---
version: '2.0'

create_vm:
  description: Simple workflow example
  type: direct

  input:
    - vm_name
    - image_ref
    - flavor_ref
  output:
    vm_id: <% $.vm_id %>

  tasks:
    create_server:
      action: nova.servers_create name=<% $.vm_name %> image=<% $.image_ref %>
flavor=<% $.flavor_ref %>
      publish:
        vm_id: <% task(create_server).result.id %>
      on-success:
        - wait_for_instance

    wait_for_instance:
      action: nova.servers_find id=<% $.vm_id %> status='ACTIVE'
      retry:
        delay: 5
        count: 15
```

Figure 8: Mistral Workflow using YAML

2.2.7 Telemetry

As described in [28] the mission of OpenStack Telemetry is to reliably collect data on the utilization of both physical and virtual resources comprising deployed clouds, to persist the collected data for subsequent retrieval and analysis, as well as to trigger actions when pre-defined criteria are met. A convoluted environment such as OpenStack has vast telemetry requirements, including perplexed use cases involving metering, monitoring and alarming functions. In order to properly support dedicated components for all prerequisites, OpenStack Telemetry consists of various projects each designed to provide a discrete service in the telemetry space, and is built on an agent-based architecture [29]. Several modules combine their responsibilities to collect data, store samples in a database and provide an API service for handling incoming requests. In particular, (i) Aodh is an alarming service, (ii) Gnocchi is a time-series database and resource indexing service, (iii) Panko is an event and metadata indexing service and (iv) Ceilometer, which acts as OpenStack Telemetry's data collection service.

2.2.7.1 Ceilometer

Ceilometer is a data collection service that provides the ability to normalize and transform data across all current OpenStack core components [30]. Operating as a critical element of OpenStack's Telemetry, its data can be used to provide information that can be transformed into billable items. Ceilometer collects data using two methods:

(i) **Notification agent**, which takes messages generated on the notification bus and transforms them into Ceilometer samples or events. This is the preferred method of data collection.

(ii) **Polling agents**, will poll some API or other tool to collect information at a regular interval. The polling approach is less preferred due to the load it can impose on the API services.

The first method is supported by the ceilometer-notification agent, which monitors the message queues for notifications. Polling agents can be configured either to poll the local hypervisor or remote APIs (public REST APIs exposed by services and host-level SNMP/IPMI daemons). Ceilometer offers the ability to take data gathered by the agents, manipulate it, and publish it in various combinations via multiple pipelines. This functionality is handled by the notification agents. The overall summary of Ceilometer's logical architecture is shown in Figure 9 along with a representation of how polling and notification agents gather data from multiple sources.

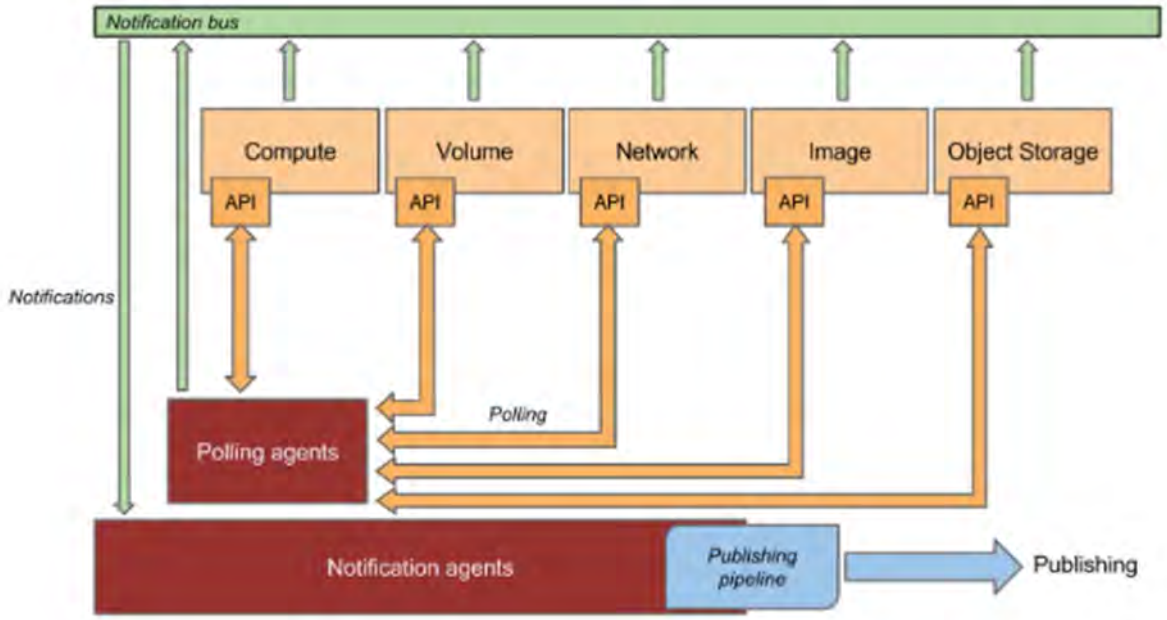

Figure 9: OpenStack Ceilometer data collection mechanisms

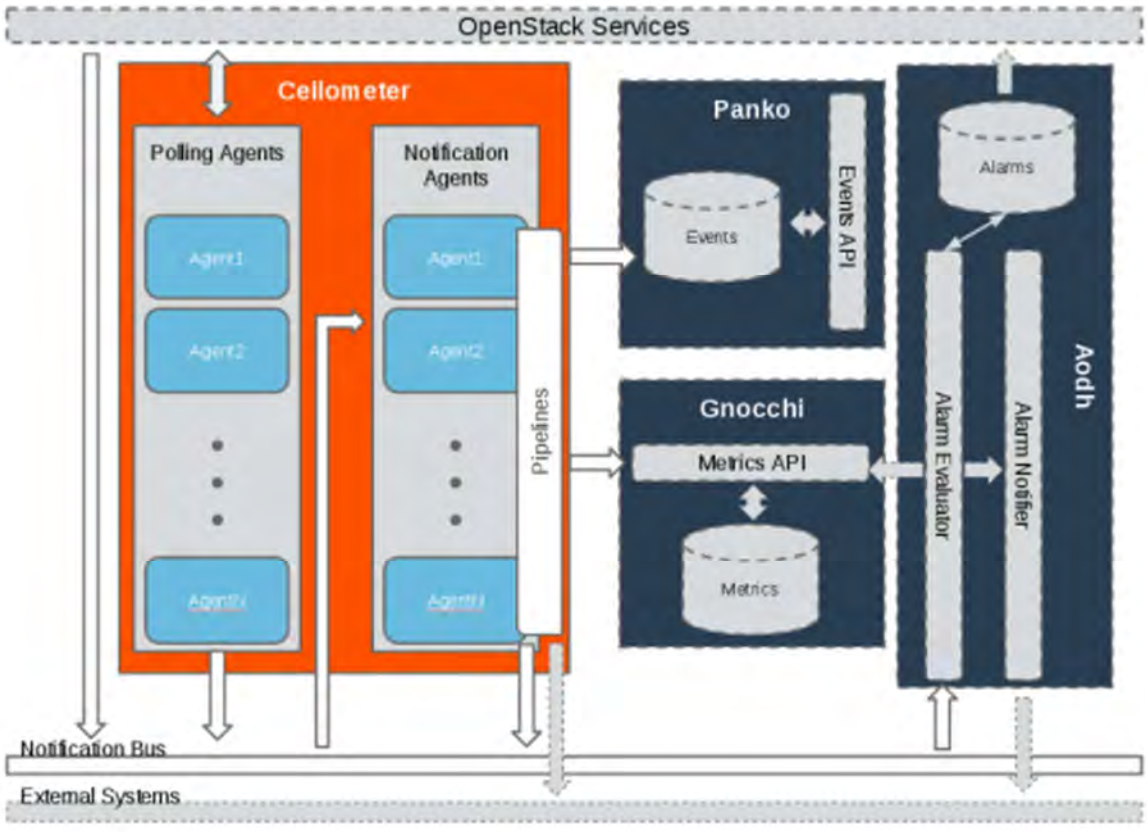

Figure 10: OpenStack Ceilometer architecture and communication interfaces


2.2.7.2 SNAP Telemetry Plugin

Snap is an open source telemetry tool which was widely used during the Superfluidity Project. The integration of snap into the project's testbed along with the necessary configuration details, metrics and result analysis is provided in Section 3.1.1 of Deliverable D4.1 [58].

2.3 Kubernetes

Kubernetes (K8s) is an open-source software that ensures an efficient orchestration of containers at scale while managing full application stacks. Such a container management platform provides high-level availability, scalability, and portability. K8s ensures greater control over the infrastructure while extending the containerization strategy based on its rich tooling. It is worth noting that Kubernetes was initially designed and implemented by Google and then donated to the Cloud Native Computing Foundation (CNCF). Today, Kubernetes is raising a lot of interest and measured as one of the most active projects on GitHub.

Kubernetes relies on a modular and scalable architecture that ensures abstraction between the applications and the underlying infrastructure. As depicted in Figure 11 like most distributed computing platform, a Kubernetes cluster consists of one master and multiple worker nodes. The master node is responsible for scheduling, provisioning, controlling and exposing application program interfaces (APIs) to the clients. Indeed, Kubernetes is highly API-centered and exposes APIs for almost every operation. All operations to perform on the cluster are passed through the APIs, which are exposed using the following two kinds of methods: User Interface (UI) and Command Line Interface (CLI).

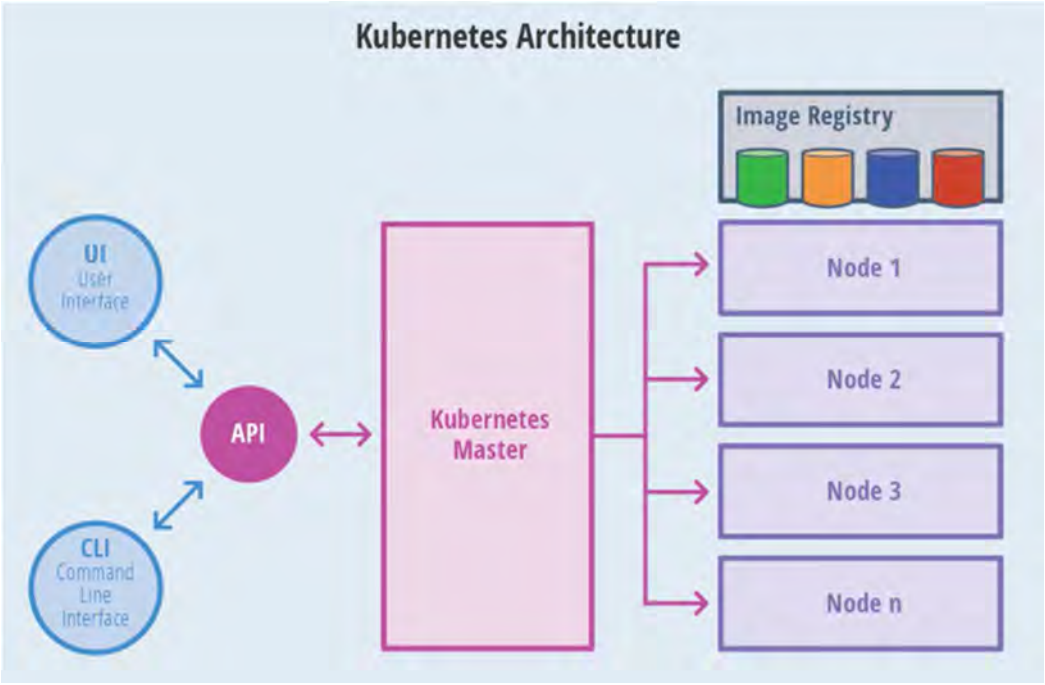

Figure 11: Kubernetes Architecture

Each Kubernetes node can have multiple pods and each pod contains different containers of an application, as illustrated in Figure 12. A service is an abstraction which defines a logical set of pods and a policy by which to access them - sometimes called a micro-service. A Service is defined using YAML or JSON. An application can be exposed using different ways, which are specified by service type and as follows.

- ClusterIP - expose the service on an internal IP in the cluster. Service is only reachable from within the cluster.
- NodePort - accessible from outside the cluster using <NodeIP>:<NodePort>.
- LoadBalancer - fixed external IP.
- ExternalName - expose the Service using an arbitrary name by returning a CNAME record with the name.

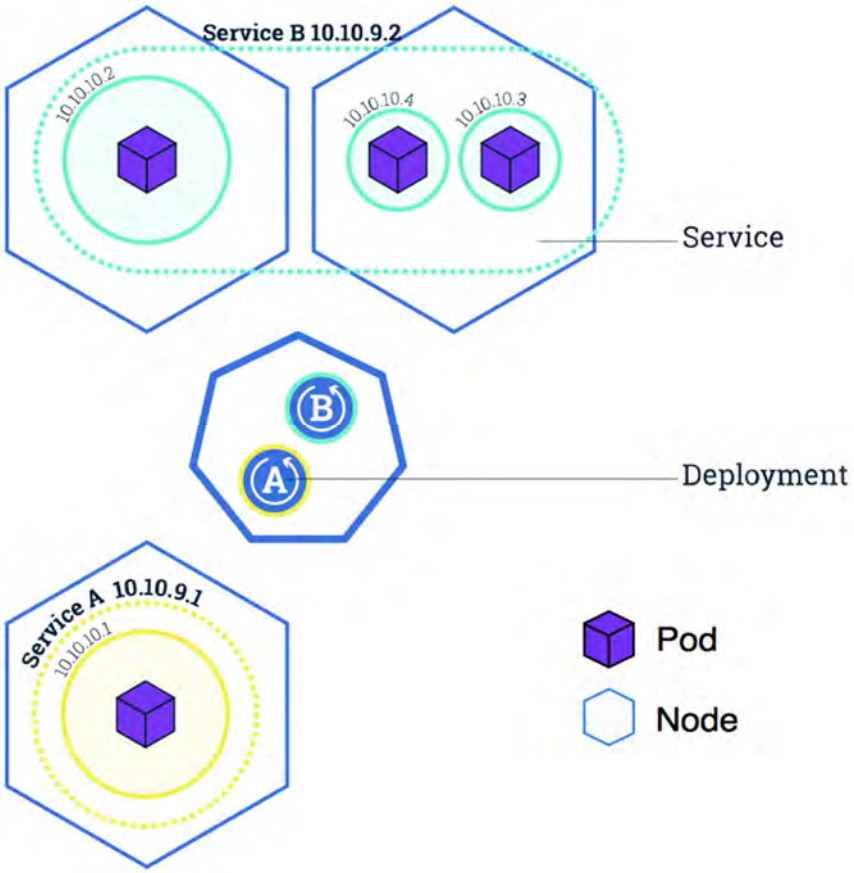

Figure 12: Kubernetes Pods and Services

2.4 Multi-Access Edge Computing

2.4.1 Introduction to MEC

Multi-access Edge Computing (MEC), formerly standing for Mobile Edge Computing, offers application developers and content providers cloud-computing capabilities and an IT service environment at the edge of the network. This environment is characterized by providing ultra-low latency and high bandwidth, as well as real-time access to local services, via suitable API, that can be leveraged by applications. Examples of these services are radio network status information or location.

MEC provides a new ecosystem and value chain. Operators can open their networks edges, e.g. the Radio Access Network (RAN), to authorized third-parties (e.g. app providers), allowing them to rapidly deploy innovative applications and services towards the subscribers, enterprises and vertical segments.

Multi-access Edge Computing will leverage new vertical business segments and services for consumers and enterprise customers. MEC use cases include, among others:

- Video analytics
- Location services
- Internet-of-Things (IoT)
- Augmented reality
- Optimized local content distribution
- Data caching.

2.4.2 Overall Architecture of MEC

MEC was the natural development in the evolution of mobile base stations and the convergence of IT and telecommunications networking.

It uniquely allows software applications to tap into local content and real-time information about local-access network conditions. By deploying various services and caching content at the network edge, core networks are alleviated of further congestion and can efficiently serve local purposes.

New MEC industry standards (coming from ETSI ISG MEC) and the deployment of MEC platforms, will act as enablers for new revenue streams to operators, vendors and third-parties. Differentiation will be enabled through the unique applications deployed in the Edge Cloud.

Figure 13 shows the MEC architecture as defined by the ETSI MEC [60].

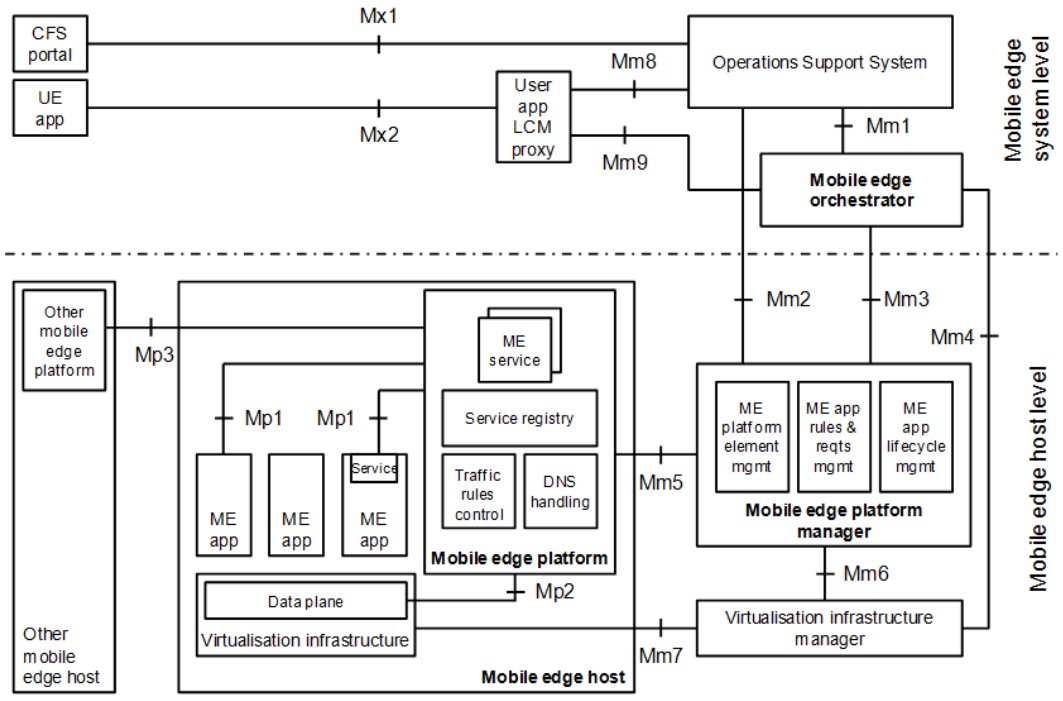

Figure 13: ETSI MEC: architecture reference model [60]

Figure 14 depicts the implementation view of the MEC architecture blocks (according to Altice Labs view).

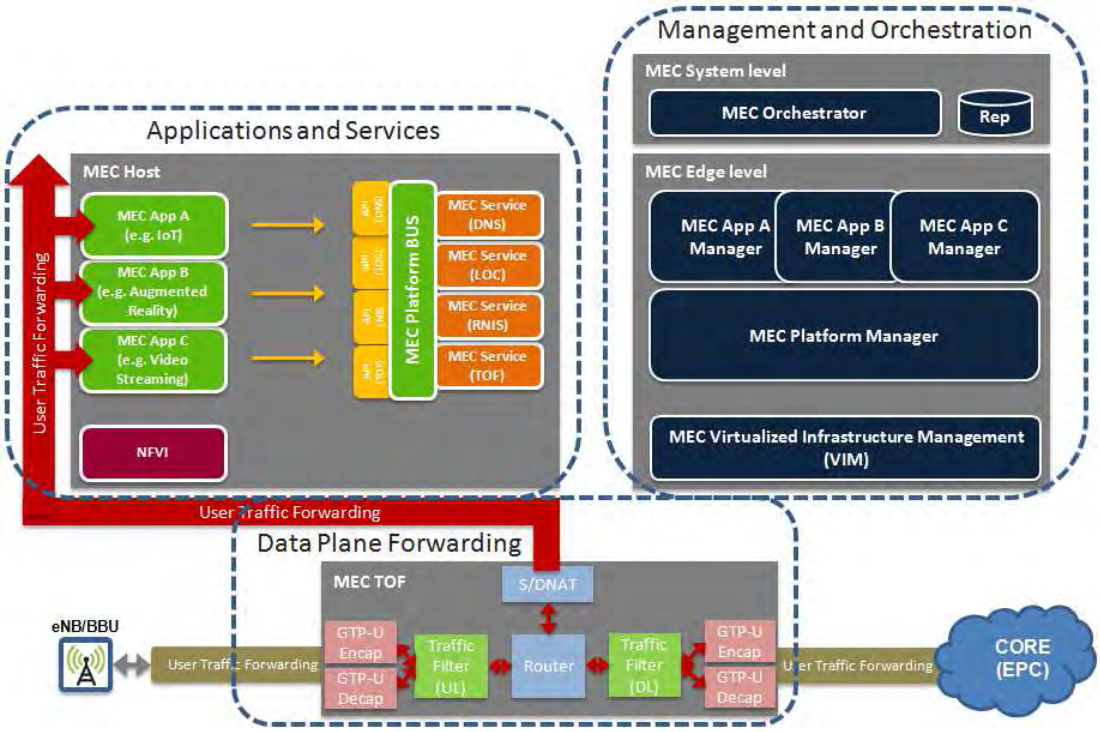

Figure 14: ETSI MEC: main architectural blocks

In this figure, 3 main building blocks can be identified: TOF (*Traffic Ofloading Function*, handling the "Dataplane Forwarding"), MEC Host (preceding the environment to run "Applications and Services") and MEC MANO (Management and Orchestration). These will be summarized in the next subsections and detailer later in dedicated sections.

2.4.2.1 MEC TOF

The high-level architecture of the MEC TOF block is depicted in Figure 15 .

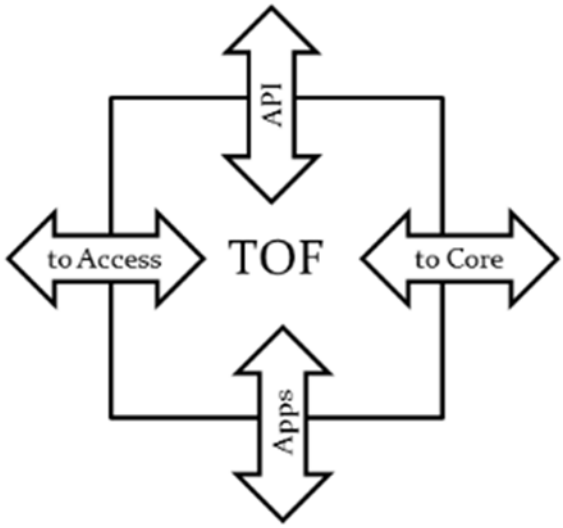

Figure 15: MEC TOF block

The TOF has three data-plane interfaces (eNodeB, SGW and Apps) and a single control interface (API). The data-plane carries user-data to/from the respective elements it connects. For instance, in an LTE environment, the eNodeB is the radio access connected to the 'toAccess' interface, the SGW is a dataplane EPC element connected to the 'toCore' interface, and Apps are running at the edge cloud and require access to the dataplane.

The data-plane interfaces should be seen as network adapters. Although there are three interfaces in the data-plane, the two that connect to the access and to the core can (with some caveats, as described later in this document) be merged into a single network adapter that connects to the S1 interface.

The API interface is a REST service and is used to provision the TOF and it works in a similar way to the Mp2 interface in the ETSI MEC architecture. This interface can be bound to a particular network adapter (management), but can also use any other interface at the host. This allows greater flexibility on how one chooses to perform the networking to reach the API. However, this also means that there is the need to externally harden the access to the API during the host's configuration (there are major security implications involved: do not overlook this step).

2.4.2.2 MEC Host

The high-level architecture of the MEC Host block is depicted in the figure below.

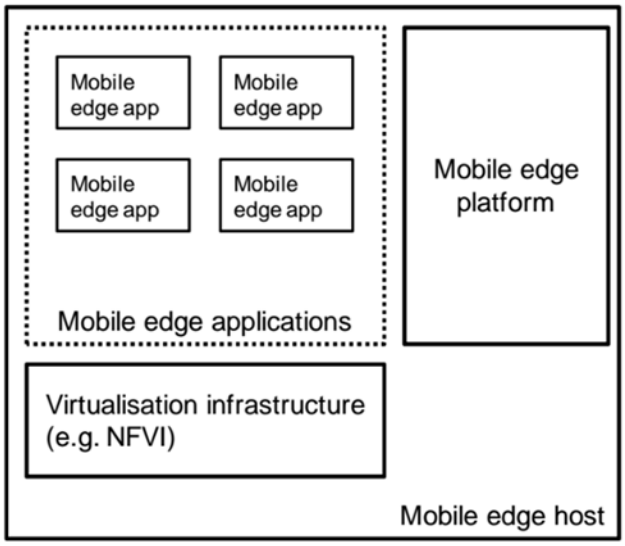

Figure 16: MEC Host Block [60]

The MEC Host is composed by three main components: virtualization infrastructure (cloud), Multi-access edge platform and Multi-access edge applications.

The virtualization infrastructure is an NFVI-like infrastructure, which provides compute, storage, and networking resources, for the purpose of running mobile edge applications (MEC Apps).

The mobile edge platform is the collection of essential functionality required to enable multi-access edge applications to consume and provide services. The multi-access edge platform also includes basic/standard services.

Multi-access edge applications are instantiated on the virtualization infrastructure of the mobile edge host based on configuration or requests validated by the mobile edge management (MEC MANO).

2.4.2.3 MEC MANO

The high level architecture of the MEC MANO block is depicted in Figure 17.

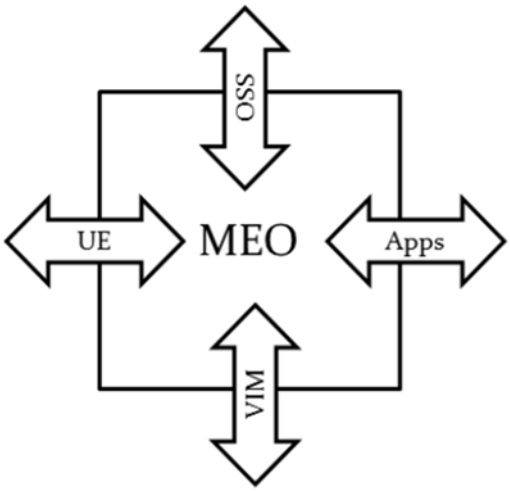

Figure 17: MEC MANO Block

The MEO (Multi-access Edge Orchestrator) has two control interfaces (UE, OSS) and two enforce interfaces (VIM, Apps). The MEO is located at the system level in order to have a global vision of the entire framework, and is co located with the OSSs.

This component needs to interact with different MEC Hosts in order to enforce the decisions. For that, the MEO needs to interface the different VIMs and the MEC applications, running in those hosting machines.

The OSS interface is an API service and is used to receive the descriptor templates from OSS for each MEC Application, and the trigger actions.

The UE interface is a complementary method to the OSSs, in order that the UEs can trigger MEC Applications execution. This interface can be used through a visual interface GUI, or a simple API, depending on the MEO implementation.

3 Superfluidity Platform Orchestration

Orchestration is the automated arrangement, coordination and management of computer systems, middleware and services. Especially in cloud computing and contemporary telecommunication infrastructure environments, virtualized networks can span a large number of networks, software elements, and hardware platforms, NFV orchestration tools must be powerful and able to work with many different standards. This leads to an overall orchestration definition, expanded to incorporate elements such as architectural composition of tools and processes, software and hardware component stitching and workflow connection and automation towards delivering a pre-defined service. Any NFV orchestrator must support functions for delivering: (i) service coordination and instantiation through software able of communicating with the underlying NFV platform, (ii) service chaining, allowing a service to be cloned and multiplied when needed, (iii) service scaling by allocating sufficient resources for service delivering and (iv) service monitoring, where the performance of both platform and resources are tracked to ensure meeting certain quality standards.

These requirements fuelled the management and organization (MANO) working group of ETSI decision to create a framework for NFV, which breaks down the management and orchestration needs of the NFV architecture, introducing an NFV Orchestrator (NFVO). The NFV Orchestrator provides management of the NFV services, which is responsible for new Network Service (NS) and VNF package onboarding, together with NS lifecycle and global resource management, validation and authorization of network functions virtualization infrastructure (NFVI) resource requests.

Superfluidity Project focused on the evaluation of two such frameworks: (i) the Open Source MANO (OSM), which delivers an open source management and orchestration stack aligned with ETSI NFV Information Models, providing an a regularly updated implementation of a production quality MANO stack that meets current and future requirements of commercial NFV networks and (ii) ManageIQ which efficiently supports both virtual machine and container deployment. Due to its limited container technology support, OSM was deployed and used as an orchestrator only to a limited part of the overall Superfluidity testbed, specialized on Multi-access Edge Computing evaluation. The main testbed orchestration functionality is provided by ManageIQ with Ansible and Kubernetes, all of which are presented in the following sections.

3.1 Open Source MANO

3.1.1 OSM Release Zero

The architectural components of Open Source MANO Release Zero are presented in Figure 18. OSM integrated OpenMANO [45], Canonical's Juju Server [46] and RiftWare [47] to operate as Resource Orchestrator, Configuration Manager and Service Orchestrator respectively, all connected through a dedicated VLAN. The minimal infrastructure requirements for installing and running OSM Release

Zero was a single server split into three different VMs. In order to efficiently divide the server into VMs a hypervisor was required, allowing optimal VM configuration and virtual resource allocation.

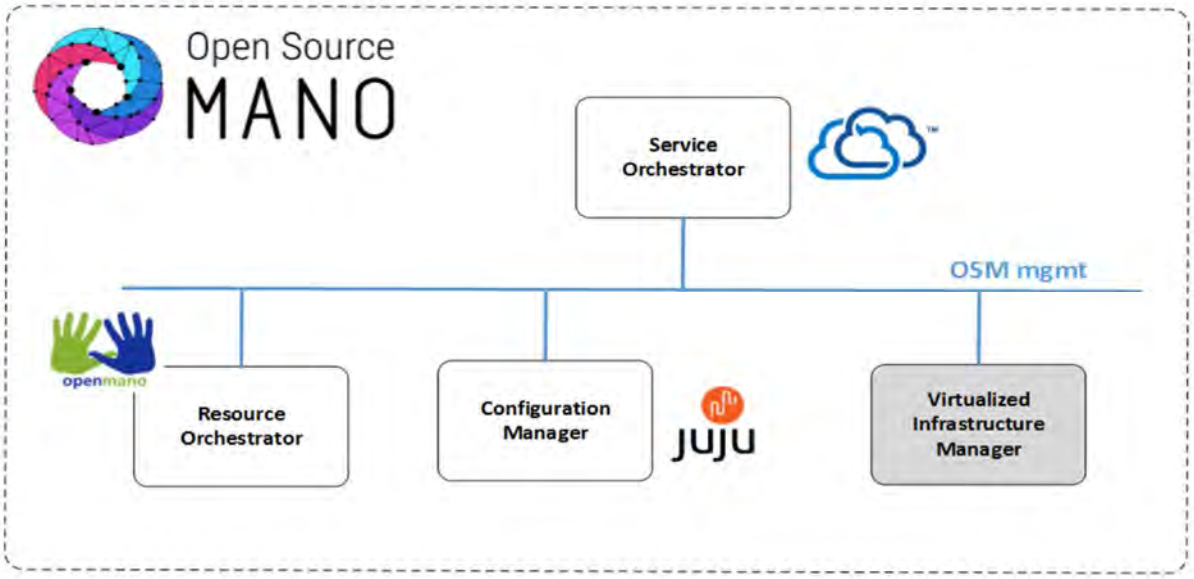

Figure 18: Open Source MANO - Release Zero

OpenMANO Installation

In order to install OpenMANO, a virtual node with minimum 1 vCPU, 2 GB of RAM, 40 GB of available disk space and 1 network interface connected to the OSM management network, is needed. The necessary base operating system namely Canonical's Ubuntu 14.04.4 LTS should be installed in advance, for the OpenMANO installation process to begin. One could install OpenMANO automatically using certain installation scripts or manually, by installing all required packages including mysql-server, configure python-argcomplete, clone the git repository and situate in the v0.0 tag, configure the previously installed MySQL by granting access privileges from localhost and add openmano client and scripts to the PATH. Once the OpenMANO installation was finished, users need to configure the openmano server, the openmano local CLI client, create an openmano tenant and attach the newly created openmano tenant to the virtual infrastructure manager datacenter, in our case an OpenStack Mitaka-based deployment.

Juju Installation

For installing a single VM with both Juju server and client, a total of 4 vCPUs, 4 GB of RAM, 40 GB of available disk space, along with the necessary OSM mgmt. network interface, are needed. Similar to OpenMANO, the base operating system is also Canonical's Ubuntu 14.04.4 LTS. Unlike OpenMANO, the only necessary package and libraries pre-requisites are Juju Server and Client which also need to be configured accordingly, as described in [48].


RiftWare Installation

The specific OSM component requires 1 vCPU, 8 GB of RAM, 40 GB of available disk space on the hypervisor to hold the disk image as well as a network interface while the base operating system is Fedora 20. In order to setup a build environment suitable for building RiftWare's major component, RIFT.ware, there are two different approaches: (i) download a pre-built image or (ii) connect to the dedicated ETSI repository and download the RIFT.ware source code. Then, initiate the build and once this process is concluded, RIFT.ware Launchpad service can be deployed. It is possible to connect to the service via a web browser, seamlessly through a self-signed certificate.

When all three virtual machines are properly instantiated, the VIM of choice can be installed and configured. Even in this preliminary OSM release, the community offers specific VNFs for testing which can be downloaded for the application's repository. Despite the fact that each OSM release is deprecated once the next release is available, all necessary documentation wikipages are still available.

3.1.2 OSM Release One

Open Source MANO Release One, greatly simplifies the overall installation process by using an all-in-one installation script. One of the major configuration changes of the specific release is that all services that were deployed in VMs in Release Zero, are now launched in containerized format. The installation script, also handles the Linux Bridge configuration, allowing the OSM node to establish connectivity with the outside world and exchange configuration information. OSM Release One now only requires a single server or VM with 8vCPUs, 16 GB of RAM, 100 GB of hard disk space and a single interface with internet access. The base OS is now Canonical's Ubuntu 16.04, configured to run LXD containers. Figure 19 summarizes the new architectural approach of OSM Release One, while Table 4 compares the amassed hardware and software requirements of Releases One and Zero.

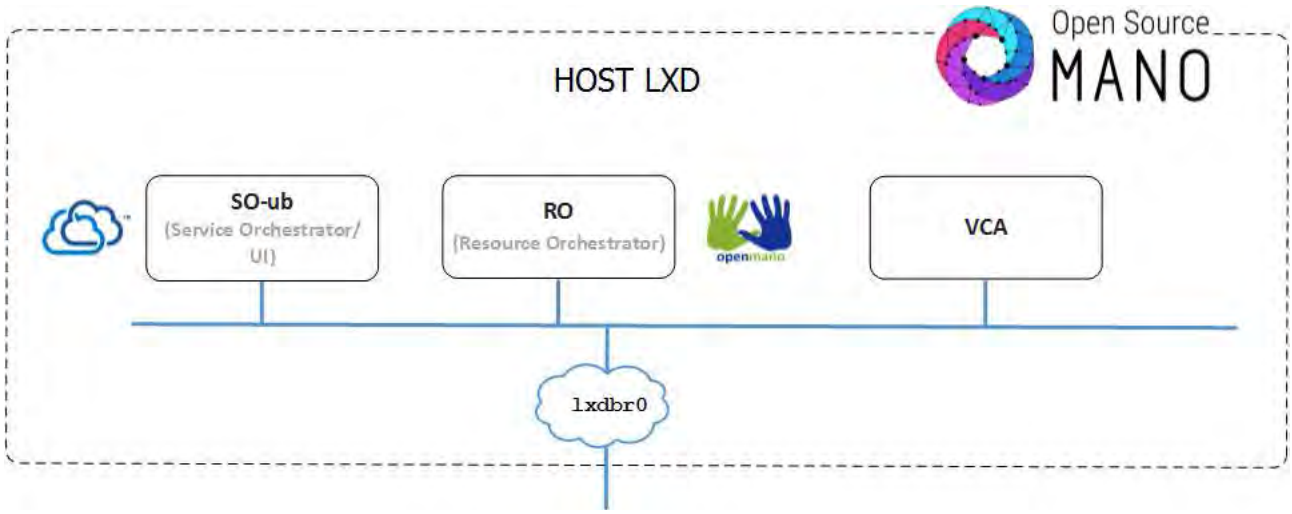

Figure 19: Open Source MANO - Release One

Table 3: OSM Release Comparison

OSM Release	vCPU (min/rec)	RAM (min/Rec)	HDD
Zero	6/24	14/40 GB	120/140 GB
One	**8**	**16 GB**	**100 GB**

Component	Linux Distro	Rec. Version	Avail. Updates
RO/CM	Ubuntu 14.04.4	0.4.38/1.25	-/2.0 (16.04 LTS)
SO	Fedora 22	4.2.0	4.2.1
HOST LXD	Ubuntu 16.04 LTS	-	-

The overall installation and evaluation process for OSM Release One can be found in the Appendix of this deliverable.

3.1.3 OSM Release Two

Continuing to meet operators' needs for predictable, high-quality Open Source MANO releases, the ETSI Open Source MANO group (ETSI OSM) announced OSM Release Two. This release brought significant improvements in terms of interoperability, performance, stability, security, and resources footprint to meet operators' requirements for trials and upcoming RFx processes.

Certain new features of OSM Release Two included:

- SDN assistance to interconnect traffic-intensive virtual network functions with on-demand underlay networks
- Support for deployments in hybrid clouds through a newly developed Amazon Web Services plugin


- OSM's plugin model for major SDN controllers has been extended with ONOS, which joins ODL and FloodLight in the list of supported controllers
- Dynamic network services to scale resources on demand
- Multiple installer options to ease OSM installation in different environments

The new SDN capabilities enabled advanced types of underlay connectivity that are often unavailable in a non-customized, virtualized infrastructure manager (VIM), thus avoiding performance degradation. This is transparent for operators, who only need to request the right type of connectivity in their virtual network functions or network service descriptors without concerns about the need for special hardware, server wiring or manual post-deployment intervention. OSM Release Two was evaluated by the Superfluidity project partners and the overall results are presented in Annex A of Superfluidity Deliverable D6.1 [59].

3.2 ManageIQ with Ansible

The main concern about the existing NFVO tools was the lack of applications life-cycle management actions for containers. This was the main reason for exploring ManageIQ as an NFVO, as it had some support for both VMs and Container providers. ManageIQ is a management project that enables managing containers, virtual machines, networks and storage from a single platform, connecting and managing different existing clouds: OpenStack, Amazon EC2, Azure, Google Compute Engine, VMware, Kubernetes or OpenShift.

The container management was being included, and there was no full support for all the application life-cycle management actions required by NFV orchestrators. Therefore, we have worked on extending Manage IQ to support the execution of Ansible playbooks on its providers.

Ansible is a software tool that automates software provisioning, configuration management, and application deployment and delivers simple IT automation that ends repetitive tasks and frees up DevOps teams for more strategic work. The reason to choose Ansible amongst others (such as puppet or chef) is its simplicity and the fact that is an agent-less configuration tool. Thus, you do not need to install anything on any of the providers being managed by ManageIQ. Any action can be triggered in any of the providers, for example, deployment of a HEAT template for the OpenStack/VMs case, or deployment of kubernetes/Openshift templates for the Kubernetes/Containers case.

The way this extension was added is by leveraging one of the most powerful capabilities of ManageIQ: the self-service. It allows administrators to create certain set of actions (following a state machine) and create a catalog with them so that users can consume them.

To support the execution of playbooks we have created a "service bundle" at the catalog of available ManageIQ services. This service bundle is a group of one or more service items that ManageIQ know how to create/handle and that are executed in a given order specified by the ManageIQ state machine. For example, in our case one service item is getting the playbooks that the user can select, and that needs to be executed before the actual playbook execution (which will be another service item).

Additionally, services typically require some inputs. In our case the playbook to be executed and the provider where it will be executed. This information is requested from the user through a ManageIQ dialog, which is created by using ManageIQ's built-in dialog editor.

Once the service bundle and the dialog are created, they need to be associated with an "entry point" in the ManageIQ workflow engine (called "Automate"). The entry point defines the process to provision the bundle. With the bundle definition, dialog, and entry point, the user just need to select the new service in the service catalog and select the desired provider and playbook action to execute and finally submit the request. This will make ManageIQ to execute the selected playbook at the chosen provider, for instance, creating a container and a load balancer at the selected Kubernetes cluster.

The integration and interaction points between ManageIQ and Ansible made through the ManageIQ self-service feature is mainly based on two points, defined as two service items belonging to the service created at the catalog:

- Playbook refresher service item: it performs 'git pull' actions to update the content of all the linked repositories where the playbooks exists. This enables easy on-boarding/removal/update of application life-cycle management actions (a.k.a. playbooks) just by interacting with git repositories in the normal way, i.e.: git commit, git push, ...
- Ansible playbook execution service item: it executes the selected playbook, parsed from the dialog, using as inventory file the information about the parsed selected provider.

As a follow-up extension we added the support to provide an inventory file (instead of automatically generating one based on the provider) to be able to specify different sites (a.k.a, providers) where the playbook could be executed. The playbook then can state where each task must be executed and specified relations between them, i.e., executing certain task in one provider before another task is executed in the same or a different provider. As an example, this is convenient when deploying the CRAN, MEC, EPC components as they can share information about each other while being created. More details can be found in Superfluidity Deliverable D6.1 [59].


4 Communication Interfaces and API Design

Application deployment in Superfluidity architecture is essential to rely on a well-defined API which operates over specific pre-defined interfaces, as those defined by the ETSI NFV Industry Specification Group (ISG) [31]. As already presented in Superfluidity Deliverable D3.1 [54] any proposition related to User-Management (UM) API development must build on the functions and reference points of the NFV Orchestrator (NFVO), augmenting them to expose the unique capabilities of the Superfluidity architecture and the newly introduced components and functional blocks. In this deliverable, our intention is to map and elaborate on specific API calls that need to be implemented on each and every main NFV reference point [31]. In the context of specifying the API to third-parties, one rather important factor to consider is the concrete textual representation of the information model elements. Practically speaking, this specifies the syntax of the files that will represent the network service templates, VNF descriptors, VNF forwarding graph descriptors, etc. For properly addressing all issues, an analysis of the actual ETSI NFV Architecture is needed.

4.1 ETSI NFV Architectures

Over the last few years, the ETSI NFV ISG [31] identified the functional blocks of a holistic NFV architecture along with the main reference points between them. Some of these blocks are already present in current deployments, while others are identified as necessary additions for enhanced virtualization-based functionality. Figure 20 depicts an overview of the particular architecture with all major functional blocks as well as the reference points in between. VNFs rely on the resources provided by the Network Functions Virtualization Infrastructure (NFVI), which is composed by the compute, networking and storage hardware resources. These resources are managed by one or more Virtualized Infrastructure Managers (VIMs), which expose northbound interfaces to the VNF Managers (VNFM) as well as the Network Functions Virtualization Orchestrator (NFVO).

The VNFM performs the lifecycle management of the VNFs through an associated VNF catalogue that stores the VNF Descriptors (VNFDs). These descriptors contain the structural properties of the VNFs, for instance the number of available ports or the internal component decomposition, along with their deployment and operational behaviour. A VNF is decomposed in one or more Virtual Deployment Units (VDUs) which can be mapped to Virtual Machines (VMs) or containers, deployed and executed over the NFV Infrastructure. The NFVO is responsible for the overall orchestration and lifecycle management of the Network Services (NS) topology, which is represented by an NS Descriptor (NSD) capturing the relations between VNFs and are stored in an NS Catalogue, parsed by the NFVO during the deployment and operational management of all services. The NFVO is also responsible for the on-boarding and validation of the descriptions of VNFs and NSs.

The main (named) reference and execution points shown by solid lines, are potential standardization targets and the communication interfaces that any API must utilize. The dotted reference points despite being present in deployments already, will need minor extensions to handle network function

virtualization, but are not the focus of NFV at the moment. As stated in [31] the architectural framework focuses on the necessary functionality for the virtualization and the consequent operation of an operator's network but does not strictly specifies which network functions should be virtualized, leaving the network owner to decide at will. In the same manner, no direct guideline is issued for communication interface functionality besides certain generic principles, thus providing an additional degree of freedom somehow suitable for the novel concepts and functional elements of Superfluidity.

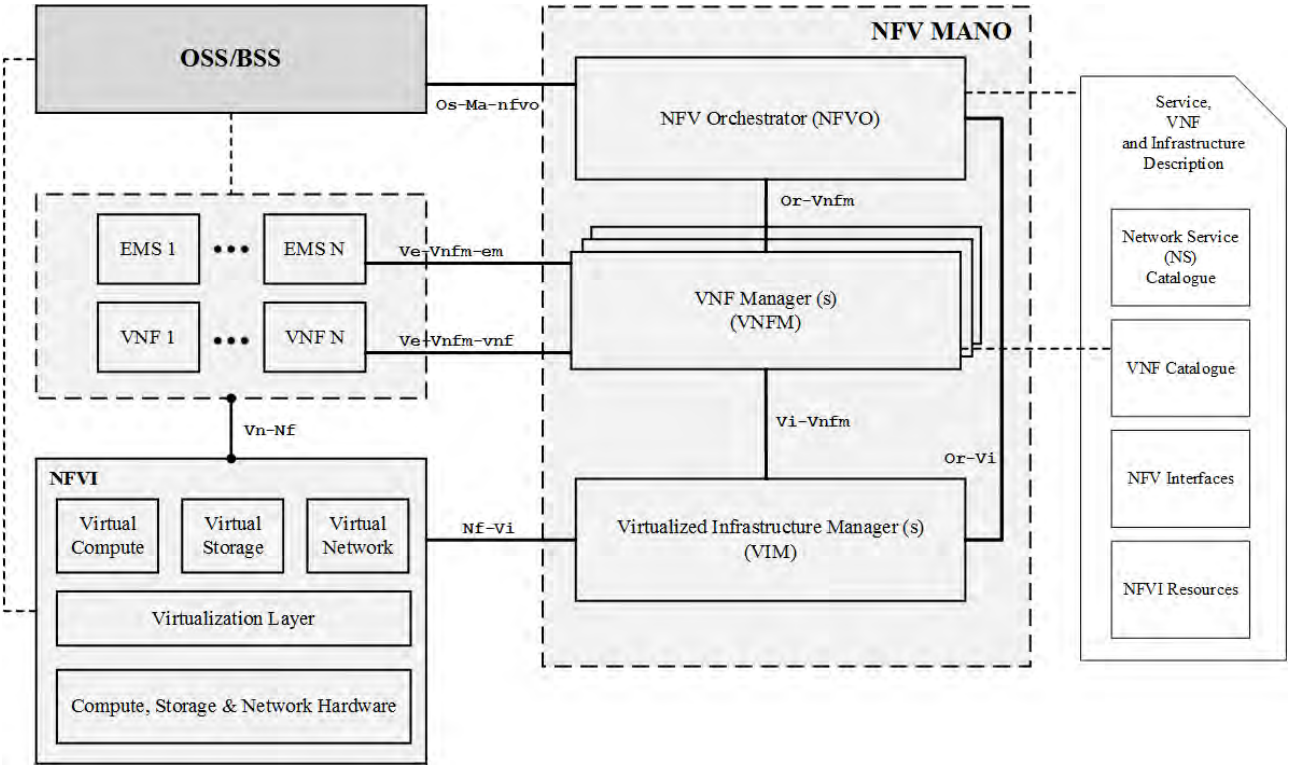

Figure 20: ETSI NFV Reference Architecture diagram

4.2 Network Service Interfaces

Network Service communications in the ETSI NFV Architecture are mostly traverse through the Os-Ma-nfvo [32], which is the sole external reference point of NFVO. This renders NFVO functionality available to OSS/BSS systems, through implementing necessary functions, allowing NFV MANO to handle:

- Network Service Lifecycle Management messages
- Network Service Lifecycle Change Notifications
- Network Service Descriptors
- Network Service Performance Management messages

- Network Service faults

In addition, the latest ETSI NFV MANO Os-Ma-Nfvo reference point specification [32] incorporates virtual network function package functionality, such as enable, disable, delete, query, subscribe/notify, fetch and abort package deletion. As also described in D3.1 [54] the top-level User-Manager API of Superfluidity must implement most of the above functions, with some of them further enhanced to leverage the project's innovations. The functions exposed by the Os-Ma-Nfvo reference point make use of a multitude of elements from the NFV Information Model [33], while NFVO must also be able to manage Network Service Templates [34], located in the NFV Service Catalogue.

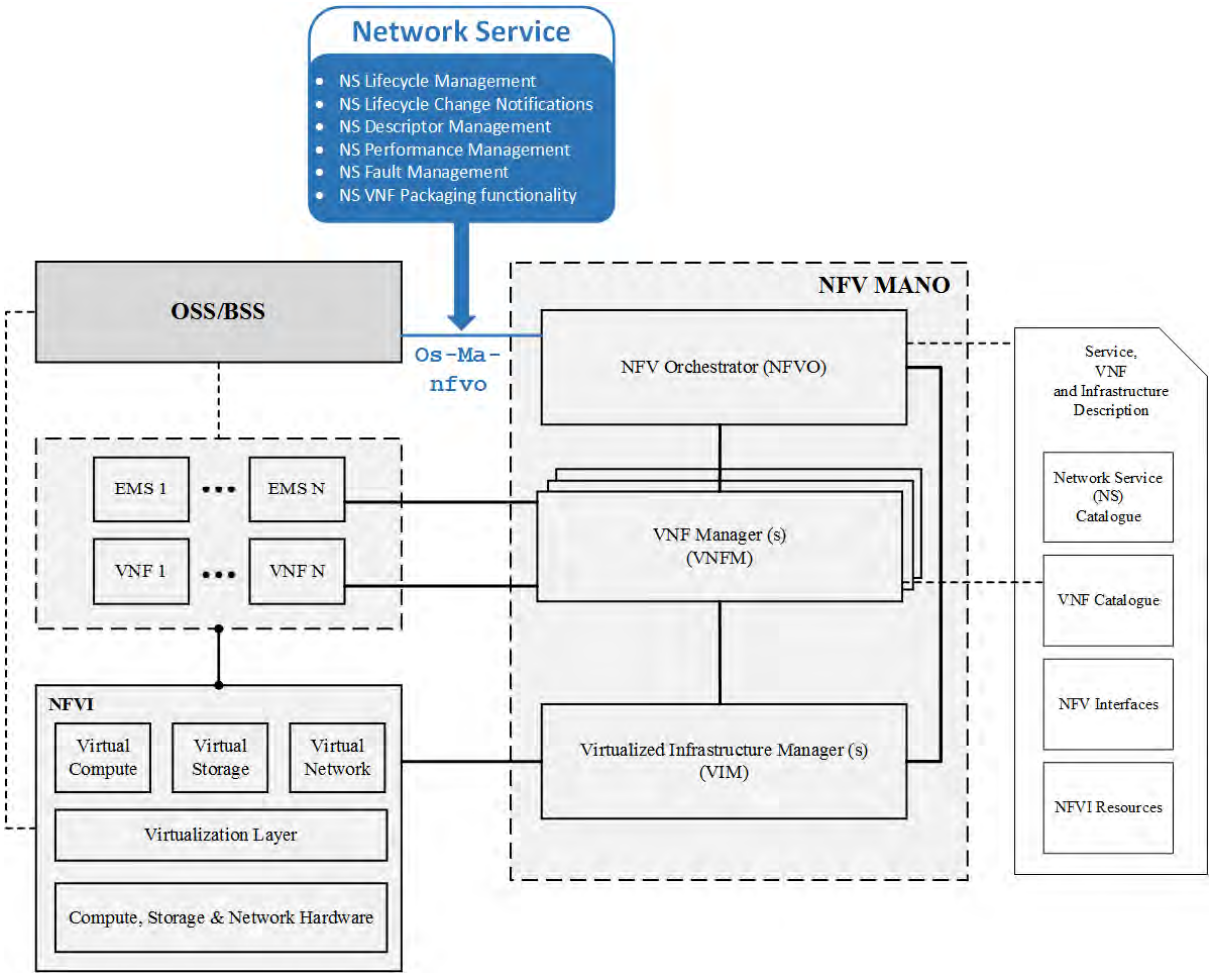

Figure 21: ETSI NFV Network Service Reference Point

4.3 Management and Configuration VNFs

ETSI NFV Architecture defines a large variety of VNFs related to management and configuration tasks. All ETSI NFV Architecture blocks connected to the VNF Manager through main NFV reference points must facilitate VNF handling, therefore support related API calls. The interconnected blocks to the VNFM are the NFV Orchestrator through the Or-Vnfm reference point, the Virtual Infrastructure

Management (VIM) through the Vi-Vnfm reference point and well as EMS/VNF blocks attached through Ve-Vnfm reference point. In addition, since NFVO also handles VNF internally, must support all corresponding functionality via the Os-Ma and Or-Vi reference points towards OSS/BSS and VIM blocks respectively. VNF actions include but are not limited to:

- VNF Package Management
- Software Image Management
- VNF Lifecycle Management and Operation Granting, along with the necessary Notifications
- VNF Configuration and Indicators
- VNF Configuration, Performance and Fault Management

Despite the inherent necessity for end-to-end functionality support, not all reference points need to integrate mechanisms for the whole set of VNF actions listed before. Figure 22 presents the specific VNFs per reference point that need to be implemented, as described in each reference point interface and information model specification document [35,36,37,38].

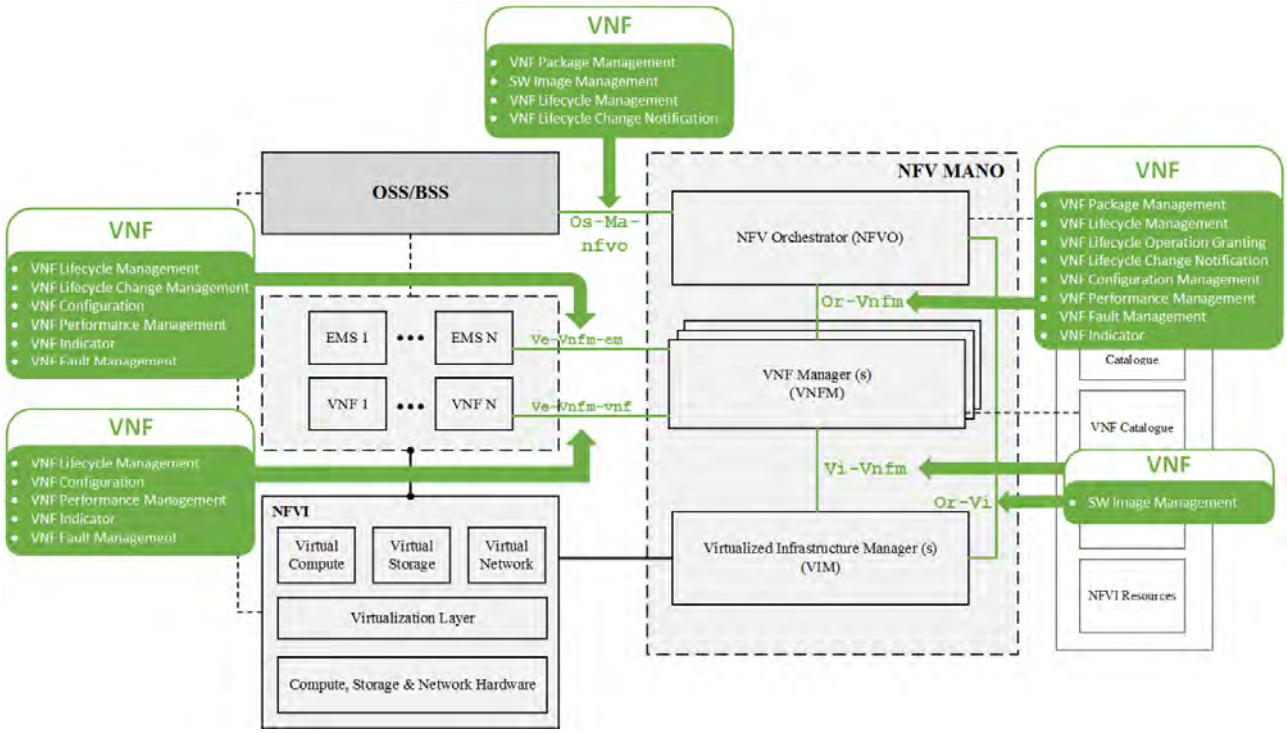

Figure 22: ETSI NFV VNF Reference Points

4.4 Virtual Resource Interfaces

The ETSI NFV Reference points related to Virtual Resource functionality are located in the NFV MANO block, consists of the NFV Orchestrator, the VNF Manager and the Virtualized Infrastructure Manager.

According to [31] the NFVO communicates (i) with the VNFM via the Or-Vnfm reference point and (ii) with the VIM via the Or-Vi reference point. In addition, the VNFM exchanges information with the VIM through the Vi-Vnfm reference point. Virtual Resource-related information, notifications, requests and responses traverse through the three reference points, it is therefore necessary to implement corresponding functions in the API, aligned with the guidelines of [35, 36, 37]. Virtual resource-related tasks are listed below:

- Virtual Resource Management
- Virtual Resource Notifications regarding available quota and possible status changes
- Virtual Resource Information, Capacity, Performance, Reservation and Fault Management
- Network Forwarding Path Management

Figure 23 presents the virtual resource information chunks per specific NFV main reference point, since like the Management and Configuration VNF case, not all functional blocks demand access to the same information subset available in the NFV MANO.

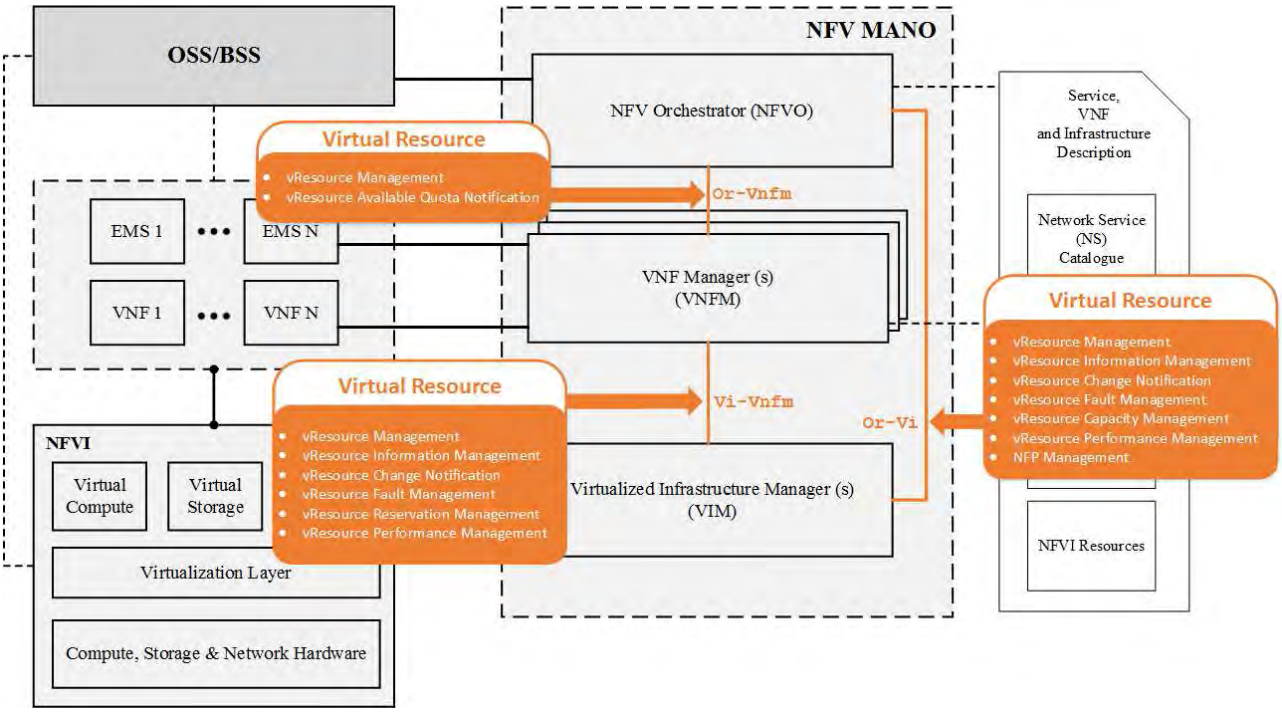

Figure 23: ETSI NFV Virtual Resource Reference Points

4.5 NFVI Functions

The ETSI NFV Architecture only defines a single reference point which facilitates communication between the NFVI and the Virtualized Infrastructure Manager, namely the Nf-Vi reference point. This

leads to a unique interface on each side for NFVI-related information exchange, in particular messaging associated to NFVI Networking, Storage and Hypervisor resources management. Figure 24 depicts the exact information elements that Nf-Vi reference point handles, are therefore need to be integrated to the Superfluidity API.

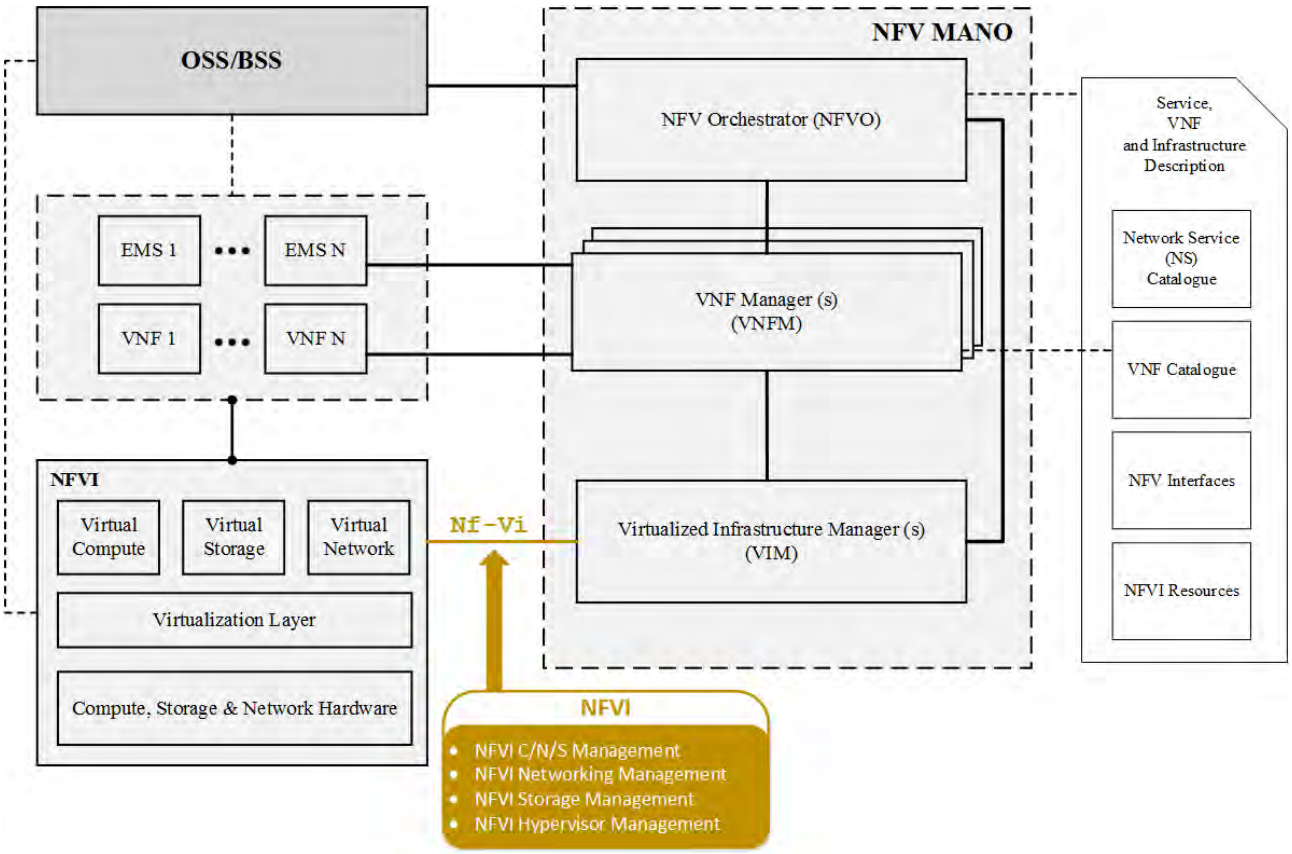

Figure 24: ETSI NFV NFVI Reference Point

4.6 External and Internal Interfaces of ETSI NFV Architecture

ETSI NFV ISG published a series of documents clarifying all specification issues derived from missing attributes of the now deprecated Release 1 of the ETSI NFV Architectural Framework. All Main NFV reference points were analyzed, in documentation containing all specification, interfaces and functions per architectural block. In April 2016, the first set of Release 2 specification was published, namely ETSI GS NFV-IFA 002 [39], ETSI GS NFV-IFA 003 [40] and ETSI GS NFV-IFA 004 [41] containing specification regarding Acceleration Technologies for VNF Interfaces, vSwitch Benchmarking and Management aspects respectively, ETSI GS NFV-IFA 005 [35] with specification regarding the Or-Vi reference point, ETSI GS NFV-IFA 006 [36] with specification regarding Vi-Vnfm reference point and ETSI GS NFV-IFA 010 v2.1.1 [42] with the necessary Functional Requirements specification. In September 2016, the remaining Release 2 requirements, interfaces, and information model specifications were completed, in particular ETSI GS NFV-IFA 007 [37] related to Or-Vnfm reference

point, ETSI GS NFV-IFA 008 [38] describing the Ve-Vnfm reference point, ETSI GS NFV-IFA 010 v2.2.1 [43] including certain updates on the previous version, ETSI GS NFV-IFA 011 [44] with the VNF Packaging Specification, ETSI GS NFV-IFA 013 [32] related to Os-Ma-Nfvo reference point, ETSI GS NFV-IFA 014 [34] containing the Network Service Templates specification and in November 2016, the ETSI GS NFV-IFA 015 [33] with the Report on the NFV Information Model. Additional specifications addressing Security, Reliability, Use Cases and other features were also available, some of which contain information related to Vn-Nf execution point and other auxiliary reference points that were not addressed recently by the ETSI NFV ISG, such as Nf-Vi. Figure 25 summarizes the corresponding documents per main reference point and functional block.

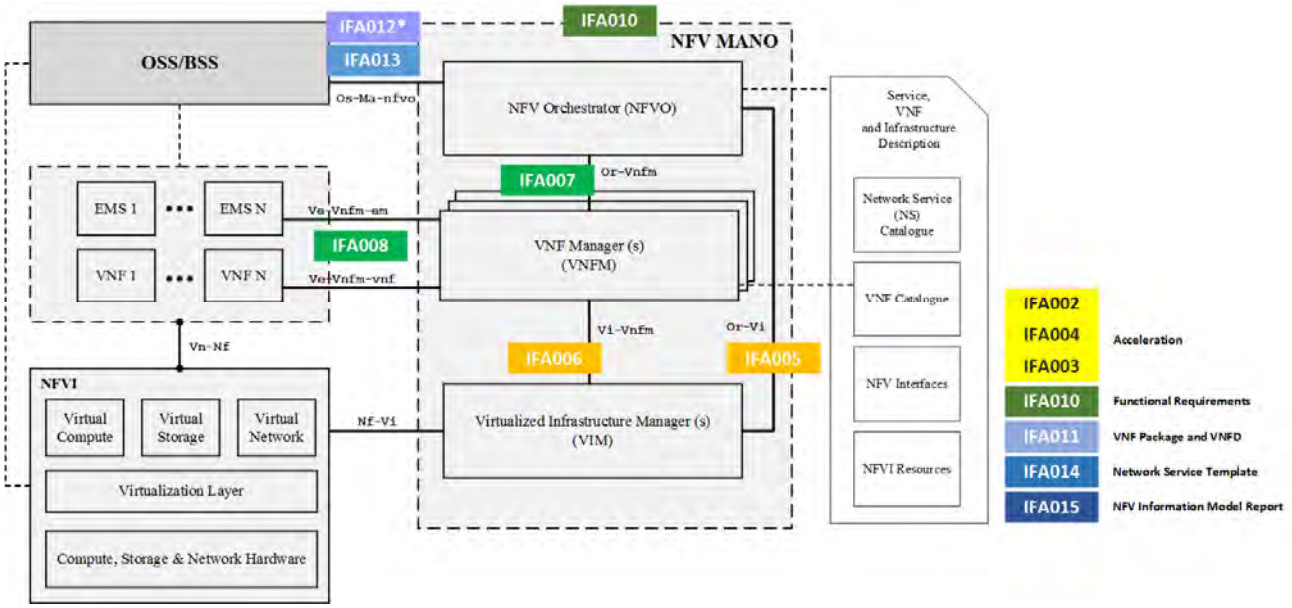

Figure 25: ETSI MANO Architecture - Specifications and Reference Point Analysis

4.7 NFV Orchestrator API/Functions

The approval of the ETSI GS NFV-IFA specifications was a major step towards enabling interoperability between Management and Orchestration functions, as well as between VNFs and Management & Orchestration functions. However, further steps are needed, related to migrating from the functional descriptions of Management and Orchestration Interfaces to Protocols/API specifications. Additionally, Network Service and VNF descriptors need to evolve from information to data models before any implementation attempt begins. An actual step towards a pragmatic solution on developing a protocol, API and data model specifications is most likely to endorse/profile externally defined solutions when most of the ETSI requirements are met. Such an attempt is made in the Open Stack APIs, which are considered as a Stage 3 solution of the VIM Northbound interfaces, currently


defined by main reference points Vi-Vnfm and Or-Vi, analyzed in ETSI GS NFV-IFA 006 [36] and ETSI GS NFV-IFA 005 [35] respectively. Furthermore, certain specification development activity inside ETSI NFV is in progress, with the pending GS NFV-SOL 002 document which will specify RESTful protocols for the Ve-Vnfm Reference Point, as well as the GS NFV-SOL 003 document which will specify RESTful protocols for the Or-Vnfm Reference point.

4.7.1 Top Level User-Manager API Implementation Challenges

As described in Superfluidity Deliverable D3.1 [54] the top-level User-Manager API of Superfluidity should implement a large variety of (i) network service descriptors, (ii) network service life-cycle calls, (iii) network service performance functions (iv) network service fault identifiers and (v) virtual network function packages, the majority of which is summarized in Table 3.

Table 4: Common API Calls

PURPOSE	FUNCTION/API CALLS
NS Descriptors	on-board, enable, disable, update, delete, query, subscribe/notify
NS Life-cycle calls	create, instantiate, update, query, terminate, delete, heal, get status, subscribe/notify
NS Performance Functions	create job, delete job, subscribe/notify, query job, create threshold, delete threshold, query threshold
NS fault identifiers	subscribe/notify, get alarm list
VNF Packagers	on-board, enable, disable, delete, query, subscribe/notify, fetch package, fetch package artifacts, abort package deletion

However, since the corresponding reference point specification was only recently published, and the protocol is still under development, there is no NFVO that implements this specification in a standards-compliant way. Upcoming RESTful protocol specification documents, will address similar cases for other reference points i.e. Or-Vnfm and Ve-Vnfm, but at the moment, all functions will be implemented by the toolchain of the NFVOs of choice. This may lead to non-standards compliant implementation that will limit the overall functionality of the platform, compared to what ETSI NFV ISG Architecture described in the first place.

4.7.2 Relation to NFV Information Model

According to [33], the NFV Information model is organized in an NFV Core Model and certain extensions which enrich the NFV Core Model functionality for specific needs. Each model, including the Core, follows a Domain-based structure with four distinct domains defined up until today namely:

- NFV Common Domain
- Virtualized Resource Domain


- VNF Domain
- NS Domain

In addition, the NFV Information Model includes three types of view: (i) Logical, which is concerned with the functionality that the system provides to end-users, (ii) Deployment, which is concerned with the functionality that is needed to deploy the provided system to the end users and (iii) Application, concerned with the functionality that the application provides to end-users.

Despite the significant level of flexibility the proposed NFV Information model supports, the Reusable Functional Block (RFB) model proposed by the Superfluidity is even more generic. The notion of recursive support of arbitrary depth of sub-classing and decomposition needs additional extensions to the NFV information model to be properly supported. Moreover, similar changes will also need to be introduced to the MANO reference points (APIs) or the corresponding NFVO toolchain. Last but not least, Superfluidity API must tackle the so called "optimal function allocation problem" consists in mapping a set of network services (decomposed into RFBs) to the underlying, available hardware block implementations in the RFB Execution Environment (REE). It aims at minimizing resource usage while meeting individual SLAs. More information regarding the optimal function allocation problem along is presented in a following section.

4.7.3 VNF Packaging Specification

NFVO is also responsible for managing VNF Packages, stored in the VNF Catalogue. Details are published in the VNF Packaging Specification [44], but at a high level they contain:

- VNF descriptor that defines metadata for package on-boarding and VNF management
- Software images needed to run the VNF
- Optional additional files to manage the VNF (e.g. scripts, vendor-specific files etc.)

Based on our detailed review of the NS template, VNF descriptor and VNF packaging specification docs, the information model entities support decomposition up to the VNF component (VNFC).

4.8 Proposed Superfludity Extensions to ETSI NFV information model

This document proposes extensions to the ETSI NFV ISG specification [44] to support nested VDUs using heterogeneous technologies.

4.8.1 Clause 7.1.6.2.2 [Vdu Information Element] Attributes

Clause 7.1.6.2.2 is modified as follows.

The following rows are added to Table 7.1.6.2.2-1.

Attribute	Qualifier	Cardinality	Content	Description
vduParent	O	0..1	Identifier (Reference to a VduId)	This is a reference to the parent VDU which contains this VDU, thus this field is needed only for Nested VDUs. The referred VduId must be defined in the current VNFD.
vduParentBareMetal	O	0..1	Boolean	Setting this field to “true” specifies that this VDU is set to be deployed on bare metal. This value overrides the value specified in vduParent (if present).
vduParentMandatory	O	0..1	Boolean	This field specifies if the parent VDU must be present or if this VDU can be deployed also without its parent VDU. This field can be present only of the vduParent field is present. The absence of this field is equivalent to setting its value to “false”.
vduNestedDesc	O	0..1	Identifier (reference to a VduNestedDesc)	This field references a VduNestedDesc element, which contains a descriptor of this VDU.

We have extended the VDU information element contained in the VNF Descriptors to reference different types of Execution Environments that can be instantiated within a VDU and described by means of some descriptor. So far we considered kubernetes VDUs and Click router [50] configurations VDUs. In particular, we have introduced four new attributes to the VDU information element: namely *vduParent*, *vduParentBareMetal*, *vduParentMandatory* and *vduNestedDesc*.

- The *vduParent* attribute is also an Identifier. In case of a nested VDU, it references the parent VDU. An example of nested VDU could be a kubernetes pod (i.e. a group of containers) *K* inside a Virtual Machine *V*. In this case the VDU associated to K would have its *vduParent* attribute set to the *vduId* of the VDU associated to V. The VDU identifier must belong to the current VNFD, as the *vduId* is unique only in a VNFD scope (see clause 7.1.6.2.2).


- The *vduParentBareMetal* attribute is a boolean which specifies if the VDU is set to be deployed directly on bare metal (e.g. a container can be deployed inside a VM or on bare metal). This field is introduced as it is not clear if the *vduParent* attribute, which is an Identifier, can have NULL or arbitrary values (e.g. the string "Bare Metal").
- The *vduParentMandatory* attribute applies to nested VDUs only and specifies if the VDU can be deployed also without its parent VDU. Referring to the above example, it specifies whether the pod K can be deployed also on bare metal (*vduParentMandatory* set to false) or if K must be deployed inside V (*vduParentMandatory* set to true).
- The *vduNestedDesc* attribute is an identifier. It provides a reference to the VduNestedDesc object which contains the descriptor that is deployed in the REE running in the VDU (in our case a Click configuration file or a Kubernetes pod/controller template). Please note that multiple *vduNestedDesc* objects can be referenced from a single VDU (to support e.g. in fastclick, click descriptors along with their associated configuration/environment files. The proposed *vduNestedDesc* attribute uses the same approach as the *swImageDesc* attribute, which provides a reference to the software image that is deployed in a "regular" VDU.

4.8.2 Clause 7.1.6.X VduNestedDesc [new element]

4.8.3 Clause 7.1.6.X.1 [VduNestedDesc] Description

This information element references the textual descriptor of a VDU.

4.8.4 Clause 7.1.6.X.2 [VduNestedDesc] Attributes

The attributes of the VduNestedDesc information element shall follow the indications provided in table 7.1.6.X.2-1.

Attribute	Qualifier	Cardinality	Content	Description
Id	M	1	Identifier	The identifier of this VduNestedDesc.
Name	M	1	String	The name of this VduNestedDesc.
Version	M	1	String	The version of the VDU descriptor.
Checksum	M	1	String	The checksum of the VDU descriptor.
vduNestedDescri	O	1	Identifier	This is a reference to the actual

ptor			(reference to a descriptor)	descriptor deployed in the VDU (e.g. a Click configuration). The reference can be relative to the root of the VNF Package or can be a URL.
vduNestedDescriptorType	O	1	Enum	Identifies the type of descriptor file referred in the vduNestedDescriptor field (Click configuration, kubernetes pod/controller template, etc.) and consequently the type of the Execution Environment running in the VDU.
envVars	O	1	Identifier (Reference to an environment file)	This is a reference to a file which contains environment variables (e.g. docker environment variables in the format accepted by the docker run --env-file option, or e.g. an ansible playbook variables file)

We have introduced above the VduNestedDesc element. Its most relevant attributes, *vduNestedDescriptorType* and *vduNestedDescriptor*, have the following functions:

- The *vduNestedDescriptor* attribute references a descriptor to be executed by the RFB Execution Environment (e.g. a click configuration file, a kubernetes pod/controller template, a dockerfile or an ansible role)
- The *vduNestedDescriptorType* attribute defines the type of RFB Execution Environment that is running in the VDU (e.g. click, kubernetes, dockerfile or ansibledocker).

4.8.5 Clause 7.1.6.4.2 [VduCpd] Attributes

Clause 7.1.6.4.2 is modified as follows.

The following rows are added to Table 7.1.6.4.2-1.

Attribute	Qualifier	Cardinality	Content	Description

InternalIfRef	O	0..1	String	Identifies the network interface of the VDU which corresponds to the VduCpd. This attribute allows to bind the VduCpd to a specific network interface of a multi-interface VDU.

We have also introduced a new attribute, namely internalIfRef, to the VduCpd information element. The VduCpd information element is referenced by the VDU information element through the intCpd attribute. We add the attribute internalIfRef to the VduCpd element to create a correspondence between a VduCpd element and the network interface of a multi-interface VDU. For example, ClickOS instances internally name their interfaces as numbers starting at "0". A ClickOS-based firewall with two interfaces would thus have an interface named "0" and an interface named "1". The firewall could expect (in its Click configuration file) traffic from an external network A on port "0" and traffic from an internal network B on port "1". The VDU corresponding to this ClickOS-based firewall would thus have two internal VDU connection points, one leading (through other connection points and virtual links) to the network A and one leading to network B. In this case the VduCpd element that would lead to network A would have its internalIfRef attribute set to "0", while the VduCpd element that would lead to network B would have its internalIfRef attribute set to "1".

4.8.6 Clause 7.1.6.X K8SServiceCpd

4.8.7 Clause 7.1.6.X.1 [K8SServiceCpd] Description

A K8ServiceCpd is an information element that describes a Kubernetes Service.

A K8ServiceCpd inherits from the Cpd Class (see clause 7.1.6.3). All attributes of the Cpd are also attributes of the K8ServiceCpd.

4.8.8 Clause 7.1.6.X.2 [K8SServiceCpd] Attributes

Attribute	Qualifier	Cardinality	Content	Description
serviceDescriptor	O	1	Identifier (reference to a Kubernetes service descriptor)	This is a reference to a kubernetes service template.

exposedPod	O	0...N	Identifier (reference to Vdu)	This is a reference to a VDU, described by a kubernetes template, to be exposed by this service.
(inherited attributes)				All attributes inherited from Cpd

The selector in the kubernetes service template must match the labels in the kubernetes pod templates associated to the VDUs referenced by the exposedPod attribute. Moreover there must be no VDU with the same label not referenced by the K8SServiceCpd.

We choose to represent a Kubernetes service with a connection point (CP) as a Kubernetes service has the function of exposing a port (and load balancing between the pods that are part of the service).

4.8.9 Clause 7.1.6.5.2 [SwImageDesc] Attributes

Clause 7.1.6.5.2 is modified as follows.

The following rows are added to Table 7.1.6.5.2-1.

Attribute	Qualifier	Cardinality	Content	Description
envVars	O	1	Identifier (Reference to an environment file)	This is a reference to a file which contains environment variables (e.g. docker environment variables in the format accepted by the docker run --env-file option)

4.8.10 Notes on Kubernetes Nesting

When specifying Kubernetes VDUs, the *vduNestedDescType* attribute is set to the value “kubernetes” while the *vduNestedDesc* attribute is set to the identifier of a Kubernetes template.

Kubernetes templates can describe a pod (along with its controller), which in general includes several containers (sharing the same IP address). Thus in case the *vduNestedDescType* is set to “kubernetes”, the VDU represents a pod (not a single container).

Application containers such as Docker do not expose to users the concept of network interface. Thus in the NFV scenario we should consider a pod as single interface VNF/VDU. This means that for kubernetes VDUs we do not need to specify the *InternalIfRef* VduCpd attribute.

Kubernetes VDUs can use the *vduParent* attribute as specified above and as exemplified below.

4.8.10.1 Example1: pod to be deployed on VM

In this example we have a kubernetes pod to be deployed inside a VM/kubernetes worker node. We assume that the VDU associated to the VM has the *vduId == "vmid"* and that the pod is described by the kubernetes template *k8stemplate*. Then the values of the attributes of the VDU associated to the pod would be:

- vduParent: vmid
- vduParentMandatory: true
- vduNestedDesc: id_vdu_nested
- id_vdu_nested.vduNestedDescriptorType: kubernetes
- id_vdu_nested.vduNestedDescriptor: k8stemplate

4.8.10.2 Example 2: pod to be deployed on bare metal

In this example we have a kubernetes pod to be deployed directly on bare metal. We assume that the pod is described by the kubernetes template *k8stemplate*. In this case the values of the VDU associated to the pod would be:

- *vduParent*: none
- *vduParentMandatory*: true
- vduNestedDesc: id_vdu_nested
- id_vdu_nested.vduNestedDescriptorType: kubernetes
- id_vdu_nested.vduNestedDescriptor: k8stemplate

4.8.10.3 Example 3: pod should be deployed on VM, but can be deployed also on bare metal

In this example we have a kubernetes pod to be deployed on a VM/kubernetes worker node, but, if needed (e.g. due to resource unavailability), the container can be deployed directly on bare metal. We assume that the VDU associated to the VM has the *vduId == "vmid"* and that the pod is described by the kubernetes template *k8stemplate*. Then the values of the VDU associated to the pod would be:

- *vduParent*: vmid

- *vduParentMandatory*: false
- vduNestedDesc: id_vdu_nested
- id_vdu_nested.vduNestedDescriptorType: kubernetes
- id_vdu_nested.vduNestedDescriptor: k8stemplate

To actually deploy a kubernetes pod in its parent VM/worker node, the *nodeSelector* field of *PodSpec* can be used: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

The appropriate *nodeSelector* value should be added during the translation/deployment phase to the kubernetes template referenced by the *vduNestedDesc* attribute.

4.8.11 Notes on Docker VDUs

Docker VDUs can be described in three ways:

1. by referencing a Docker image;
2. by referencing a Dockerfile;
3. by referencing a ManageIQ-Ansible role descriptor. **NOTE: this is not (yet) supported by ManageIQ**

In case 1, the docker container can be described by setting the following attributes of the swImageDesc information elements to the proposed values:

- *supportedVirtualisationEnvironment*: docker;
- *swImage*: name of the docker image;
- envVars: docker environment variable file as accepted by the docker run --env-file option.

In case 2, the Docker container can be described through its Dockerfile by using the vduNestedDesc element with the proposed values:

- vduNestedDescriptorType: dockerfile;
- vduNestedDescriptor: reference to the Dockerfile;
- envVars: docker environment variable file as accepted by the docker run --env-file option.

In case 3, the docker container can be can be represented by setting the vduNestedDesc attributes to the proposed values:

- vduNestedDescriptorType: ansibledocker;
- vduNestedDescriptor: ansible role descriptor;
- envVars: ansible environment variables file.


Please note that all the variables used by the Ansible role must be defined in the Ansible environment variables file.

Nested docker containers, deployed inside other containers or on VMs, shall reference the parent VM/container, described by a VDU, by using the vduParent attribute of the Vdud described above.

4.8.12 Open Issues

An open issue is how OpenStack+Openshift handles the creation of containers: does it create one kubernetes orchestrator per tenant? Per VM? Per NSD?

Another open issue is whether non-mandatory deployment in VMs should also have the meaning that the VDU can be deployed also on another VM (e.g. a different Kubernetes worker node).

Another open issue is if the parent VDU should be always set to a special vduNestedDescType (e.g. kubernetes_worker), or if the swImageDesc field should refer to an image which deploys a kubernetes worker node.

And, does the vduParentMandatory field follow the rationale/philosophy of the ETSI specifications? Or should we use different deploymentFlavors for the different deployment scenarios?

Verbatim from clause 7.1.9.2.2.2: “If the VIM supports the concept of virtual compute resource flavours, it is assumed that a flavour is selected or created based on the information in the VirtualComputeDesc information element.”. If the virtualComputeDesc object generates a Flavour in the VIM, why this information is not included in the DeploymentFlavour?

5 Superfluidity Mechanisms, Algorithms and Innovations

5.1 Integration of Exemplary SDN Controllers

5.1.1 ONOS

Open Network Operating System (ONOS) [64] provides the control plane for a software-defined network. It manages network components, such as switches or links, and runs software to provide communication services to end hosts and neighbouring networks. As a distributed SDN controller, it manages an entire network, and simplifies the development and deployment of new software, hardware and services. Customized applications can be programmed and instantiated above ONOS operative system layer.

In order to understand the integration of ONOS within Superfluidity, the different interfaces will be described then. The following picture describes the ONOS architecture.

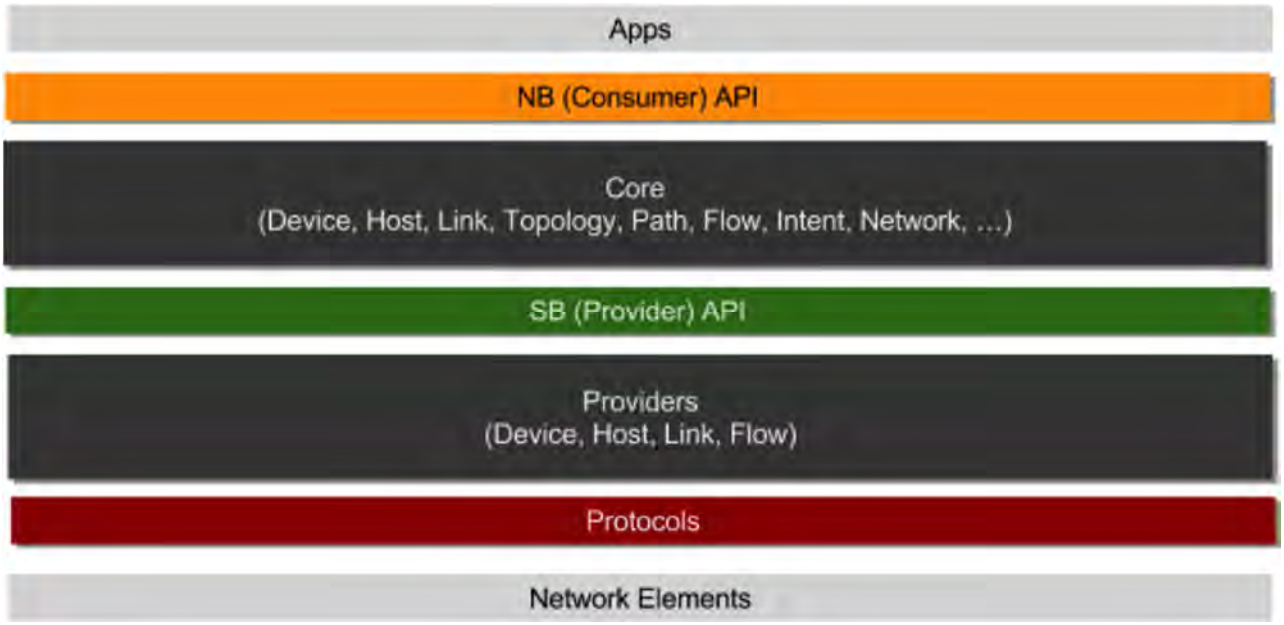

Figure 26: ONOS Architecture

Currently, two API's are identified in the ONOS stack: the northbound API and the southbound API. These two interfaces are key within the integration of ONOS with Superfluidity project. First, the southbound API enables the communication of the Core with the underlying network. ONOS considers many protocols like OpenFlow, NETCONF, SNMP, OVSDB… The communication of ONOS with the network devices will be handled by these protocols. The provider layer abstracts the configuration, control and management operations of the protocol layer. The providers will execute core requests, for instance install rules in OpenFlow, and notify the core regarding events from the devices or created by the net-cfg service. The provider implementation generalizes the logic of operating on a device, and providers can call drivers to abstract the device-specific logic. Hence,

heterogeneous devices are handled. For instance, for a device family, one can find a DeviceProvider, PacketProvider or LinkProvider, to notify the core with respect to devices, packets or links and operate.

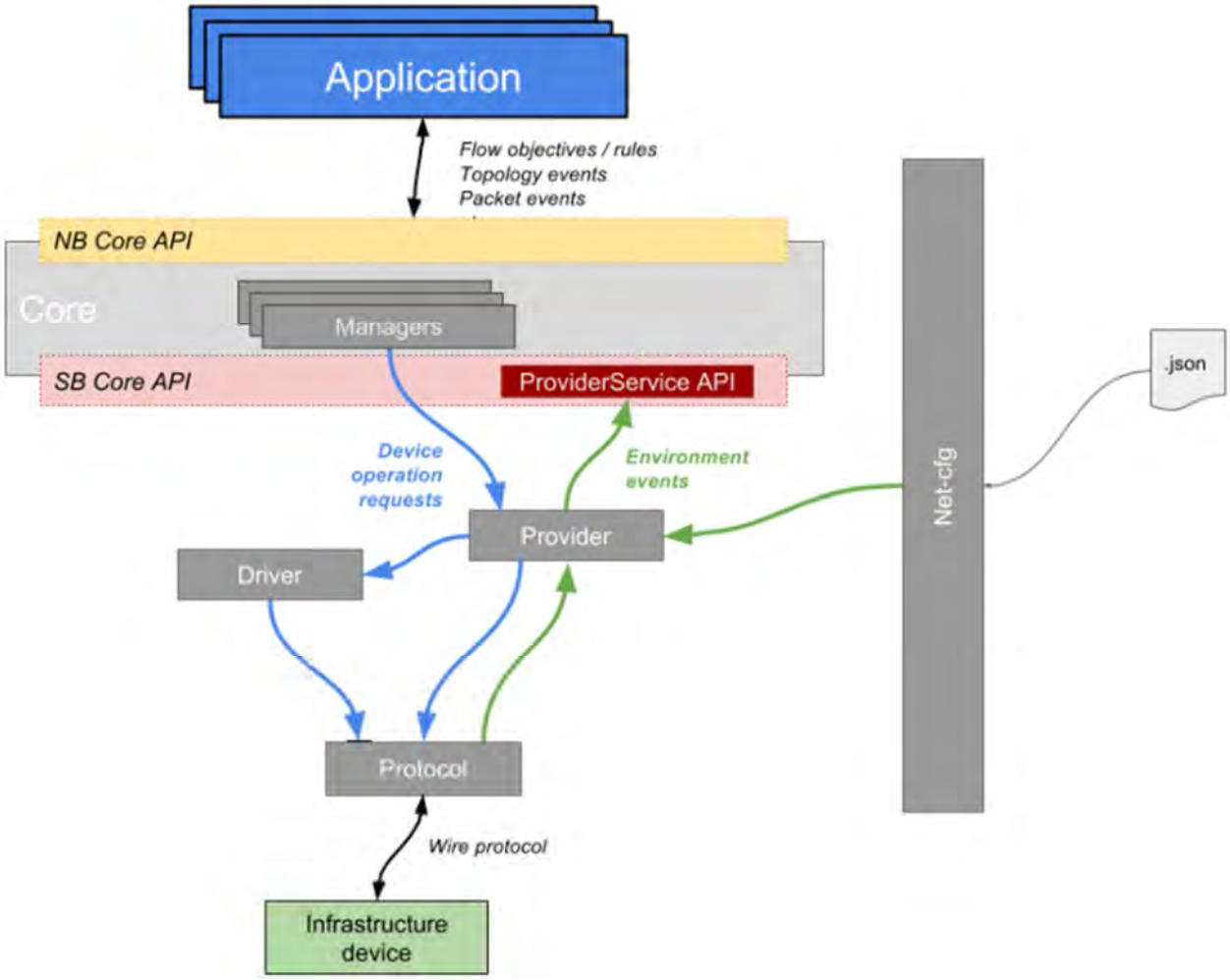

Figure 27: ONOS API

Then, the northbound API provides access to the core management. Applications use this API but it can be accessed via REST too. The core fields that can be managed are devices, links, hosts, topology, paths, flows, flows objectives, groups, meters, intents and applications. Applications itself can deploy REST APIs to consume these core elements or either other elements generated by the applications on demand (depends on the application software functionality). Thus, different applications from a functionality point of view can be developed combining the fields described and using third party software libraries. In addition, ONOS provides a GUI interface and CLI interface to access directly to its information.

Regarding the integration within superfluidity, ONOS will control the dataplane of the fronthaul network that Nokia has deployed. The southbound protocol adopted is OpenFlow, to communicate

with the network resources. Using northbound REST APIs will enable the integration of ONOS applications offering some specific functionality with the ONOS core.

5.2 Advances in Multi-Access Edge Computing Architectures

5.2.1 MEC TOF

The *Traffic Offloading Function* (TOF) will intercept the dataplane traffic, chain it through transit VNFs, then route it to an endpoint VNF or back to the dataplane where it came from. TOF is built on a modular plugin based architecture, with current implementations for GTP and Ethernet traffic interception, but being easily extensible to other dataplane protocols. A single TOF instance can handle several dataplane plugins at the same time, as long as there is no IP range overlaps on the client's side. For example, it is possible to have Ethernet and GTP plugins running at the same time. TOF allows adding a dataplane plugin without the need to rest its operation, effectively presenting a *plug and play* behavior.

TOF is built on a modular architecture, as shown in next figure:

- TOF-Core: Handles traffic classification, function chaining and executed NAT operations for endpoint VNFs. Also provides a REST north bound interface (NBI).
- TOF-Aggregator: Connects several dataplane plugins to the core, controlled by the RYU SDN controller (not shown in the figure). If only one dataplane plugin is used, this component is not mandatory.
- TOF-Dataplane plugins: Deal with the specificities of the dataplane connection; at the moment both plain Ethernet and GTP plugins are available.

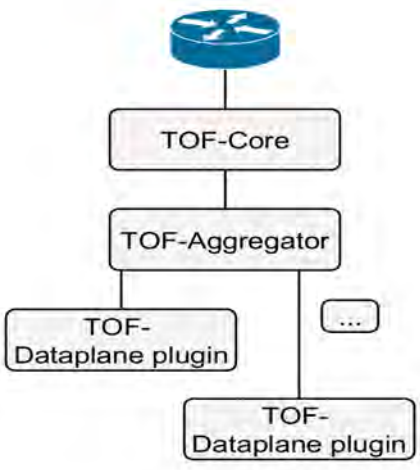

Figure 28: TOF Architecture Diagram

5.2.1.1 Dataplane Plugins

5.2.1.1.1 GTP

The *GTP* plugin intercepts traffic in the S1 interface (or any other using this encapsulation), between the RAN (Radio Access Network – eNB) and the EPC (*Enhanced Packet Core* – SGW). Ideally a dedicated adapter to the eNodeB and another for the SGW should be used. Having the possibility to have the plugin sitting in a network point where traffic between different eNB and SGW exist, if this adapter is shared, there is no possibility to auto-detect the involved eNodeBs and SGWs (therefore their IPs must be pre-provisioned in configuration files).

Capabilities Summary

The GTP key capabilities are as follows:

- Encapsulation and de-encapsulation of GTP-U users traffic (in Kernel Space);
- Auto-discovery of the TEIDs (Tunnel Endpoint IDentifier) from the GTP-U traffic;
- Learn the TEID and establish the SGW/eNodeB tunnels even when the UE traffic never reaches the EPC (use of the *Pinger* function for this purpose);
- Forwarding of the traffic to the MEC Classifier component of TOF;
- Support for multi SGWs/eNodeBs with any TEID;
- Work as regular switch for non-GTP-U traffic, in case of separated eNodeBs/SGWs interfaces;
- Auto-purge stale SGW/eNodeB tunnels after some configured idle time.

Internal Modules

This section describes the internal modules of MEC GTP component. The detailed architecture is presented in the next figure.

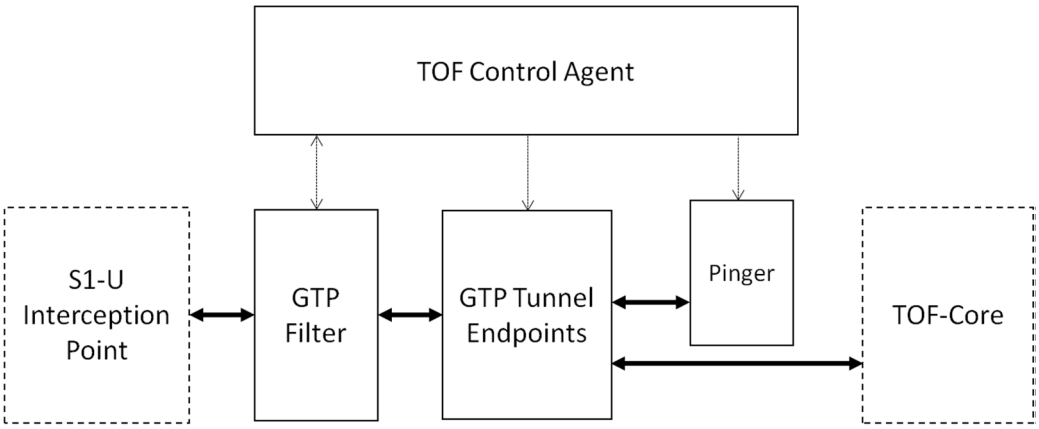

Figure 29: GTP Architecture

The GTP component comprises two large groups of components: the control-plane (the TOF Control Agent) and the data-plane (everything else). Within the data-plane, there is an additional distinction


between components, separating the ones that handle GTP-encapsulated flows and the others that handle already de-encapsulated UE-flows ('clean' user traffic).

GTP Encapsulated Data-plane

Within this type of modules, all flows are encapsulated within GTP-U. For those, the traditional flow identification/control (5-tuple or OpenFlow) considers the mobile network endpoints (eNodeB/SGW).

S1-U Interception Point

The *S1-U Interception Point* isn't actually a component of the TOF, but rather a representation of the network adapter(s) that deliver the user's traffic from the network to the TOF.

GTP Filter

The *GTP Filter* determines whether or not the GTP-U traffic must be delivered to the control-plane (happens when a new tunnel is spotted), which in turn will perform all the required *GTP Tunnel Endpoints* configuration.

A new tunnel is spotted anytime an unknown TEID arrives for a given set of IP endpoints, meaning that there is a new user session. The way this detection is performed follows a reverse logic, in which, by default, the packet goes to the control-plane unless a matching rule of TEID + endpoints matches that packet.

Because flows will only pertain to the outer tunnel endpoints, one must deeply inspect the packets to retrieve the TEID. Therefore, the GTP Filter needs to process traffic packet by packet.

All filtering process is done with iptables/netfilter rules. Packets which need to be processed in the control-plane are passed to the control application using NFQUEUES, being each queue dedicated to a single purpose (for instance, all packets that come from an eNodeB go into queue 0, while all packets that come from a SGW go into queue 1).

GTP Tunnel Endpoints

The *GTP Tunnel Endpoints* performs the GTP-U encapsulation/decapsulation of traffic that comes from / goes to the mobile network, as configured by the TOF Agent/Controller when receiving GTP-U packets from the GTP Filter.

A patched OpenvSwitch (OVS) with support for GTP-U (in kernel space) is used, along with NATing capabilities (*netfilter* for statefull translations and *tc* for stateless translations) to change the destination/source addresses as needed so that the OVS acts as the endpoint of existing S1 traffic (rules are automatically inserted by the Control Agent).

UE Decapsulated Data-plane

Since the UE traffic is already decapsulated, it is possible to control these flows using OpenFlow, forwarding to TOF-Core or to Pinger.

Pinger

The *Pinger* is a simple function which permanently generates ICMP traffic towards the outside of the mobile network.

It is used to act on behalf of the UE so that a return tunnel from the SGW can always be established, regardless whether the UE ever sends packets to the SGW (i.e. even if nothing other than the MEC app is used). The ICMP traffic causes the return of downstream traffic which allows the TEID downstream discovery.

OpenFlow is used to rewrite the source address (to match the UE's), with the addition of two virtual network adapters which are added to the port-chain (to fit in our SFF/SFC model).

Internal Interfaces

The internal interfaces between the referred components are plain shell commands, which run inside the respective element's network namespace.

The GTP Filter and the NAT components are both configured using the *iptables* command. The GTP Tunnels Endpoints, Classifier and the SFF are configured using OpenvSwitch's (OVS) OpenFlow commands. Lastly, the Pinger is not directly configured (the function is always generating ICMP requests to the target). However, OpenFlow is also used to rewrite the source address (to the one of the UE) and then connect the Pinger's output to the proper GTP Tunnel.

Flow Diagrams

A detailed description of the interaction between the internal components is described in this section in the form of flow diagrams.

TEID detection and tunnel(s) setup

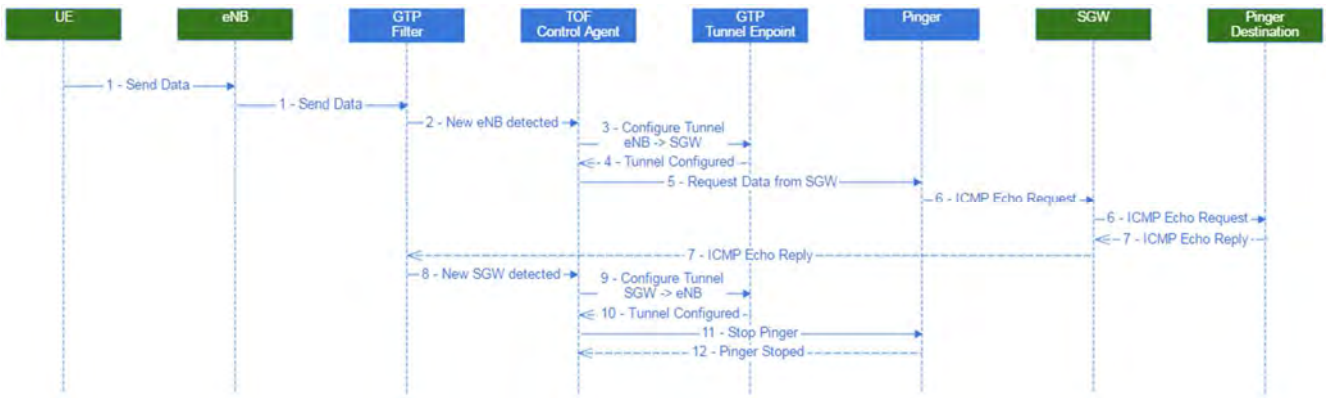

Figure 30: TEID Detection and Tunnel Setup Flow Diagram

Figure 30 depicts the TEID Detection and Tunnel Setup flow diagram. The steps are as follows:


1. The UE generates traffic to a Public Data Network (PDN) (e.g. Internet), which is forwarded to eNB. This traffic is caught at the S1 interface (between eNB and SGW) at the TOF, more precisely by the GTP Filter.
2. If the TEID from this eNB is from a new session, the TOF Control Agent is contacted.
3. The TOF Control Agent configures the GTP Tunnel between the eNB and the SGW, with the TEID present in GTP-U Header of packet.
4. The TOF Control Agent waits for the end of Tunnel Configuration.
5. To configure the tunnel from the SGW to the eNB, the TOF Control Agent configures the Pinger to generate traffic which will force a data packet from SGW.
6. The Pinger generates an ICMP Echo Request to a destination in the PDN (this traffic uses the GTP tunnel from TOF to SGW, configured in the step 3).
7. The Pinger replies and the response is caught by GTP Filter.
8. If the TEID from this SGW is a new session, the TOF Control Agent is contacted.
9. The TOF Control Agent configures the GTP Tunnel between SGW and eNB, with the TEID present in GTP-U Header of the packet.
10. The TOF Control Agent waits for the end of Tunnel Configuration.
11. The TOF Control Agent configures the Pinger to stop to generate ICMP traffic.
12. The TOF Control Agent waits for the end of Pinger configuration. The process is concluded.

5.2.1.1.2 Ethernet

The *Ethernet* plugin will intercept traffic in an Ethernet link, between an access element and the rest of the network. A dedicated adapter to the network side and another to the access side should be used. This plugin uses the Ryu SDN controller to control the OVS bridge it is running on.

Capabilities Summary

The Ethernet key capabilities are as follows:

- Forwarding of the traffic to the MEC Classifier component of TOF;
- MAC address learning (upstream) and correction (downstream) of MAC address of packets coming from the classifier.

Logical behavior

The Ryu SDN Controller is used to implement the MAC address learning and applying openflow rules to the OVS bridge.

When a packet with new MAC address appears on either the network or the access port of the bridge, the packet is sent to Ryu which learns the MAC address and implements the proper openflow flows on the bridge to redirect it from the incoming port to the MEC classifier port.


For traffic coming in from the classifier port of the bridge, if the pair “IP <-> MAC” address is known by Ryu, proper openflow flows will be implemented correcting the MAC address of said packets and forwarding them to the correct port, either the network or the access port. If unkown MAC addresses appear on the classifier port, they are ignored and the packets are discarded.

Please note that the classifier replaces the MAC address of the packets (source and destination), imposing this need to learn the pair “IP <-> MAC” address.

5.2.1.2 Aggregator

The aggregation bridge acts as a connector between the MEC classifier and the dataplane plugins, allowing more than one dataplane plugin to be running at the same time, as long as there are no overlaps on the clients’ networks (IP ranges). It uses the Ryu SDN controller to control the OVS bridge it is running on.

5.2.1.2.1 Capabilities Summary

The Aggregation key capabilities are as follows:

- IP pair learning, in order to return traffic coming from the classifier to the correct dataplane plugin;
- Dataplane *plug and play* capability, allowing dynamic addition of new dataplane plugins without restarting.

5.2.1.2.2 Logical Behavior

The Ryu SDN Controller is used to implement the IP address pair learning and applying openflow rules to the OVS bridge.

When an IP packet appears on any port of the dataplane side of the aggregation bridge, the packet is sent to Ryu which learns the IP addresses and implements the proper openflow flows on the bridge to redirect it from the incoming port to the MEC classifier port (TOF-Core).

For traffic coming in from the classifier port of the bridge, if the IP’s addresses are known by Ryu, proper openflows flows will be implemented to forward the packet to the correct dataplane port. If not, the packets are dropped, as a security mechanism.

5.2.2 TOF Core

5.2.2.1 Classifier

The Classifier determines which path packets from a particular flow must traverse (which MEC Apps), just as a SFC (Service Function Chaining) in the IETF chaining model. Because our SFC is built of port-chains, the way the Classifier reports its classification is by simply outputting to different ports, which are reserved for these criteria. The classifier supports both transit and endpoint VNFs.

For endpoint VNFs, NAT (Network Address Translation) is used to perform the replacement of the destination's IP (provisioned through the API) by the respective MEC App IP, so that traffic is delivered to that MEC App.

Port-chains are created using OpenFlow rules, as well as flows/packets are classified with OpenFlow.

Capabilities Summary

The Classifier key capabilities are as follows:

- Support for an arbitrary number of pass-through VNFs using VxLAN tunnels for VNF packet delivery;
- Support endpoint VNFs, using NAT.

Independent rules can be applied in both traffic directions (access to core network and core network to access) allowing the existence of asymmetric chains.

5.2.2.2 APIs

A detailed specification of the APIs is described in this section in the SWAGGER format [61].

5.2.2.2.1 Service API

This API configures the service application in TOF, and is referred in the previous section.

5.2.2.2.1.1 /service

GET /service

Description

List all configured services

Responses

Code	Description	Schema

200	Services configured	[
		service {
		name: string *
		rules: {
		priority: string *
		nw_dst: string
		nw_src: string *
		protocol: string
		⇄tp_dst: string
		tp_src: string
		endpoint: string
		}
		chain: {
		ue_net: [string]
		net_eu: [string]
		}
		}
		]

POST /service

Description

Configure a new service

Parameters

Name	Located in	Description	Required	Schema

service	body	Service to configure	Yes	service {
				name: string *
				rules: {
				priority: string *
				⇄nw_dst: string
				nw_src: string *
				protocol: string
				tp_dst: string

Name	Located in	Description	Required	Schema

				tp_src: string
				endpoint: string
				}
				chain: {
				ue_net: [string]
				net_eu: [string]
				}
				}

Responses

Code	Description
200	Service configured
400	Bad Request
409	Conflict

5.2.2.2.1.2 /service/{name}

DELETE /service/{name}

Description

Deletes a service based on the name

Parameters

Name	Located in	Description	Required	Schema
name	path	Name of the service to delete	Yes	⇄ string

Responses

Code	Description
200	Service deleted
404	Not found

5.2.2.2.1.3 Models

```
service {
 name:   string *
     Name of the service.
  rules: {
    Match rules as seen from the uplink. TOF will handle the downlink
    direction, reversing the rules as needed.
             priority:     string *
            In case a packet matches several entries, highest priority is
            applied. If the priority is the same for two or more services
            the behavior is undefined.
             nw_dst:   string
            The packet IP destiny on the uplink, either IP or IP/Mask.
            Required if using NAT!
             nw_src:   string *
            The packet IP source on the uplink, either IP or IP/Mask.
            Required!
             protocol:  string
            IP protocol, either 1 (icmp), 6(tcp) or 17 (UDP).
             tp_dst:    string
            The packet destination port on the uplink, use with tcp or udp
            only.
             tp_src:    string
            The packet source port on the uplink, use with tcp or udp only.
             endpoint:  string
            Destination IP on the endpoint vnf, packets IP destination will
            be NATted to this IP before delivery. Don't include to not use.
           }
  chain: {
     Uplink and downlink lists of transit vnfs.
             ue_net: [
           IPs of the transit vnfs that are to be applied to the uplink
           traffic. After all the vnfs are "travelled" the packet is either
           sent to an endpoint vnf if endpoint is defined or to the
           internet. This list may be empty.
           string
                           ]
             net_ue: [
           IPs of the transit vnfs that are to be applied to the downlink
           traffic. After all the vnfs are "travelled" the packet is either
           sent to an endpoint vnf if endpoint is defined or to the access.
           This list may be empty.
           string
                           ]
           }
}
```

5.2.3 MEC Host

The *Multi-access Edge Host* is an edge-level entity which comprises the *Multi-access Edge Platform* (MEP) and a Virtualization Infrastructure. The Virtualization Infrastructure provides compute, storage, and networking resources for the Multi-access Edge Applications (MEC Apps). The MEP is responsible for providing standard services like DNS, network information or location to be consumed

by MEC Apps, and to manage the access (authentication and authorization) of those applications to the services. MEC Apps can also produce services to be consumed by other MEC Apps.

5.2.3.1 Internal Modules

This section describes the internal modules of the MEC Host. The detailed architecture is presented in the figure below (according to ETSI ISG MEC Architecture [60]).

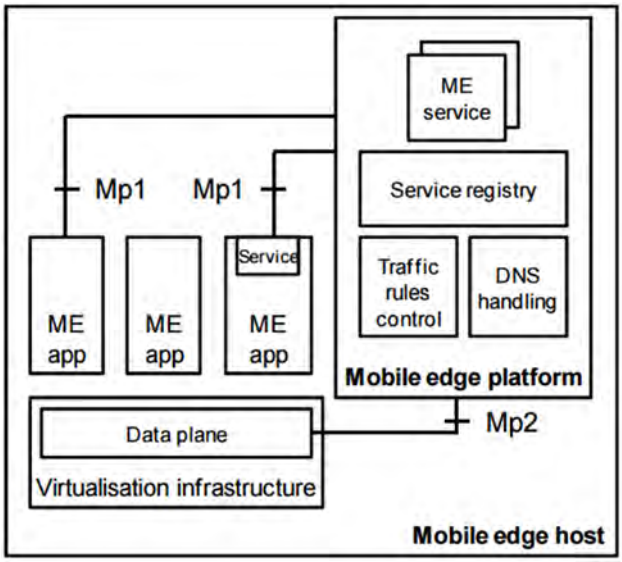

Figure 31: MEC Host Architecture

The MEC Host is made of three large groups of components: Virtualization Infrastructure, Multi-access Edge Platform and Multi-access Edge Applications.

5.2.3.1.1 Virtualization Infrastructure

The Virtualization Infrastructure of the MEC Host uses the OpenStack as the NFVI/VIM, to support all the requirements of MEC Host. Therefore, the MEC Host acquires the specifications and capabilities of this NFVI/VIM.

Capabilities Summary

This Virtualization Infrastructure is capable of:

- Deploy/dispose Multi-access Edge Applications, as VM or Containers;
- Manage the resources through Graphical User Interface (GUI);
- Routing between the MEC Apps and the dataplane;
- Isolation between MEC Applications resources;
- Isolation between traffic exchanged with different MEC applications;
- Access to services;
- Multi-tenancy

Internal Connectivity

The approach used to connect the applications to the TOF is presented in Figure 32.

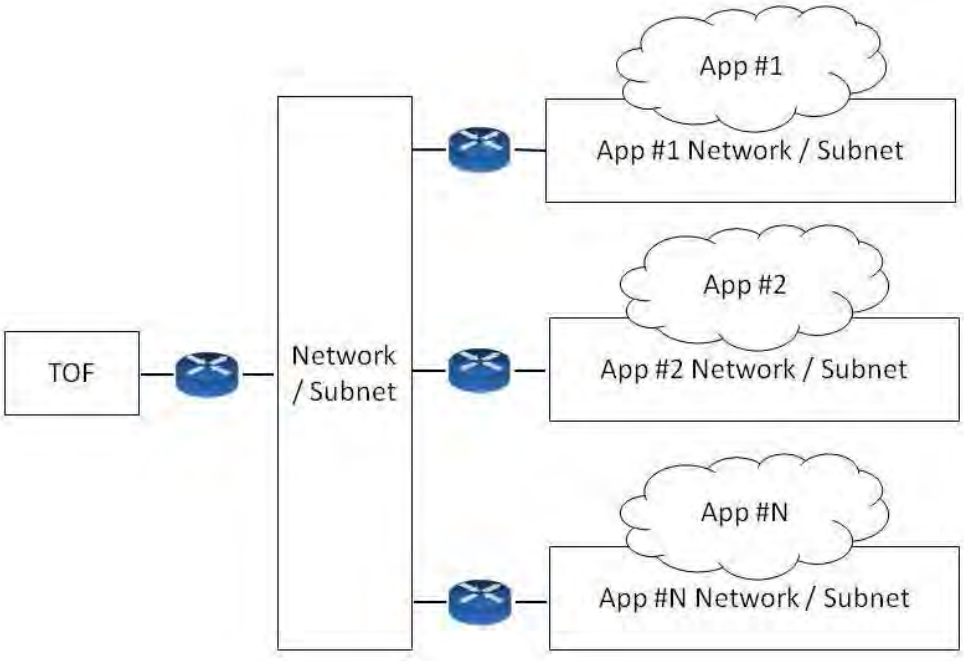

Figure 32: Connectivity between the TOF and MEC Apps

In this approach, the MEC Apps run in different tenants, as a way to isolate the resources. The same way, the network that connects MEC Apps to the data-plane (via TOF) is also isolated. This traffic is isolated trough routers and a network (public network), which can, if required, give a floating IP from the network to the MEC App's port. This way, the MEC App is accessible to the TOF using this floating IP.

The approach used to connect MEC Apps to the Service API is presented in Figure 33.

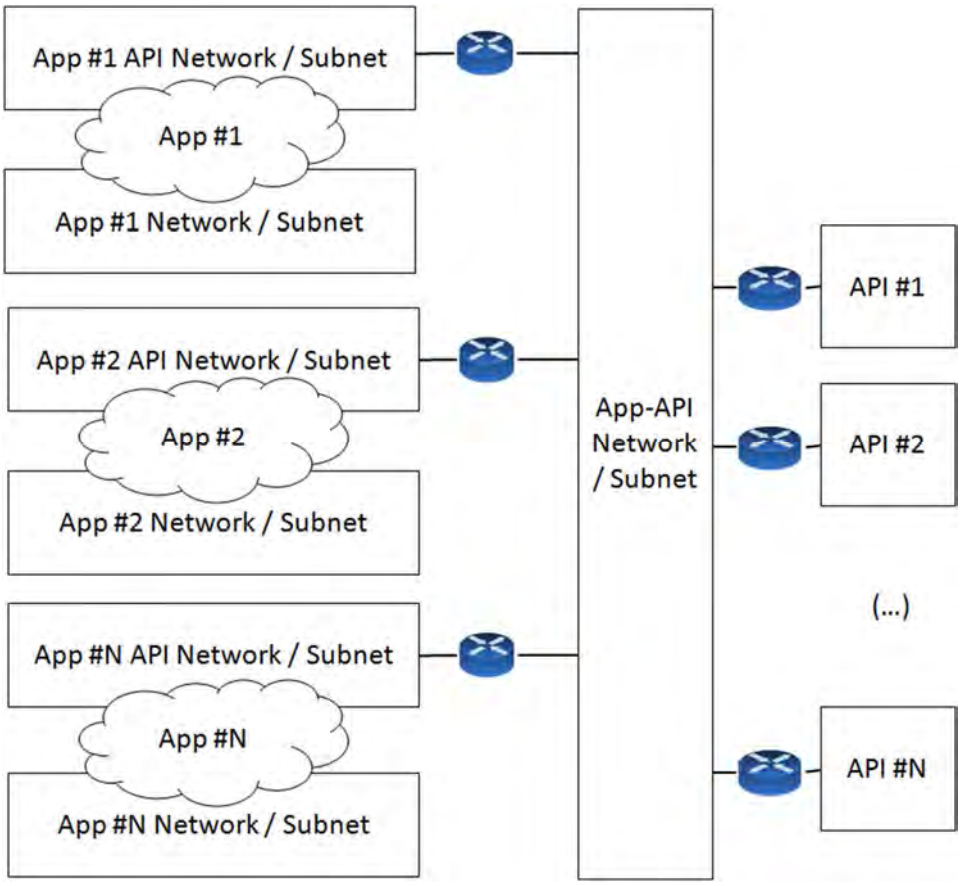

Figure 33: Connectivity between MEC Apps and Services APIs

For this approach, the strategy used was the same as the one used to connect applications to the TOF, using a network marked as public, and giving a floating IP to the applications to communicate with the MEC platform and authorized APIs.

MEC Apps are, this way, connected through two networks, one towards the TOF and another one towards the MEC Platform.

5.2.3.1.2 Multi-access Edge Platform

The Multi-access Edge Platform shall be capable of:

1. Offering an environment where the Multi-access Edge Applications can discover, register, consume and offer Multi-access Edge services;
2. Receiving DNS records from the management layer and instructing the local DNS server accordingly;
3. Hosting Multi-access edge services (e.g. RNIS and Location), in case services are virtualized;
4. Receive any service provisioning data and instructing this service accordingly;
5. Receiving traffic rules from the management layer, and instructing the dataplane accordingly.

In the scope of the Superfluidity project, to fulfill its requirements and as an SDN/NFV research activity, only the last functionally was considered relevant for implementation.

5.2.3.1.3 Multi-access Edge Applications

Multi-access edge applications run as virtual machines (VMs) on top of the virtualization infrastructure (provided by the Multi-access Edge Host), and as Kubernet containers integrated with the virtualization infrastructure trough the Kuryr OpenStack Project. This Multi-access Edge Host can access the Multi-access Edge Platform to consume and provide Multi-access Edge Services.

To interact with the Multi-access Edge Platform, MEC Applications need to implement the Mp1 interface, according to the specifications defined by the ETSI ISG MEC.

5.2.4 MEC MANO

The Multi-access Edge MANO is a core-level entity which comprises the Multi-access Edge Orchestrator (MEO). The MEO provides the information needed to deploy, in the MEC Hosts, the MEC Applications, and to manage those applications. This information is provided by OSSs or UEs in TOSCA descriptor format [49,57]. After an evaluation process of different MANO tools, OSM was selected as the basis to implement the MEO component. More details can be found in Superfluidity Deliverable D6.1 [59]

5.3 Integration of Function Allocation Algorithms

The optimal function allocation problem consists in mapping a set of network services (decomposed into RFBs) to the underlying, available hardware block implementations in the RFB execution environment (REE). It aims at minimizing resource usage while meeting individual SLAs.

Unfortunately, solving this problem ended up being harder than initially estimated, and we were not able to fully integrate it in the Superfluidity platform.

An initial abstract description of the intended API in terms of inputs and outputs is described in Section 4.3 of Superfluidity Deliverable D3.1 [54]. Additionally, how the work done could be integrated in the platform is further discussed into Superfluidity Deliverable D5.2 [62].

5.4 RDCL 3D

The vision of Superfluidity project is to orchestrate functions dynamically over and across heterogeneous environments, hence the need to interpret and operate with different RFB Description and Composition Languages (RDCLs). At least in the short-medium term, it is most likely that a variety of RDCLs will coexist and together with “meta-orchestrators” will coordinate the configuration mapping of resources over different REEs. To facilitate the transition even further, our project proposes RDCL 3D – Reusable Functional Blocks Description and Composition Language Design, Deploy and Direct, a novel “model-agnostic” web framework for descriptor design [51,52].


RDCL 3D offers a web GUI that allows visualizing and editing the descriptors of components and network services both textually and graphically. A visualized network service designer can create new descriptors or upload existing ones for visualization, editing conversion or validation. The created descriptors can be stored online, shared with other users or downloaded in textual format to be used with other tools. In particular, these descriptors can be used for the deployment and operational management of NFV services and components. A detailed analysis of the components of RDCL 3D, the methods it uses for interacting with other components of the Superfluidity testbed as well as additional links to available resources and source code repositories can be found in Superfluidity Deliverable D6.1 [59]

5.5 SEFL and Symnet

Symnet [53] is a symbolic execution tool tailored for static network analysis. The tool rapidly analyses networks by injecting symbolic packets and tracing their paths through the network, even in cases of increased network complexity where networks contain routers with hundreds of thousands of prefixes and NATs. The only limitation of symbolic execution is its inherently poor scaling ability due to the exponential complexity introduced by highly complex networks.

The Superfluidity project managed to address this limitation of symbolic execution by introducing SEFL – Symbolic Execution Friendly Language, a language specifically developed to model network processing in way that is amenable to symbolic execution. SEFL and its integration in the Superfluidity platform is described in full details in Sections 4.5 and 8 of Deliverable D3.1 [54].

5.5.1 OpenStack Neutron Configurations

We have written an OpenStack plugin that takes the router and firewall configurations and translates them into SEFL models. These could be used to check reachability before deployment, or to check that the actual deployment matches the user's intent; we are still working on integrating the results from symbolic execution back into Neutron to make it easily available to the users. A detailed analysis of the specific process can be found in Section 6 of Deliverable D6.1 [59].

5.6 NEMO

NEMO, the NEtwork MOdelling language has already been described in the scope of Superfluidity in Section 4.6 of Deliverable D3.1 [54]. In the architecture description, it was shown how NEMO could have been used for describing the composition of network resources. In order to make them more powerful, it was described how NEMO would be extended in order to cope with Virtual Network Function Descriptors (VNFDs).

Making use of this extension, we could have used the NEMO language as a description model for the previously described RDCL 3D tool, using Nodemodels in a similar way to VNFs, and Network Intents as NSDs, as can be seen in

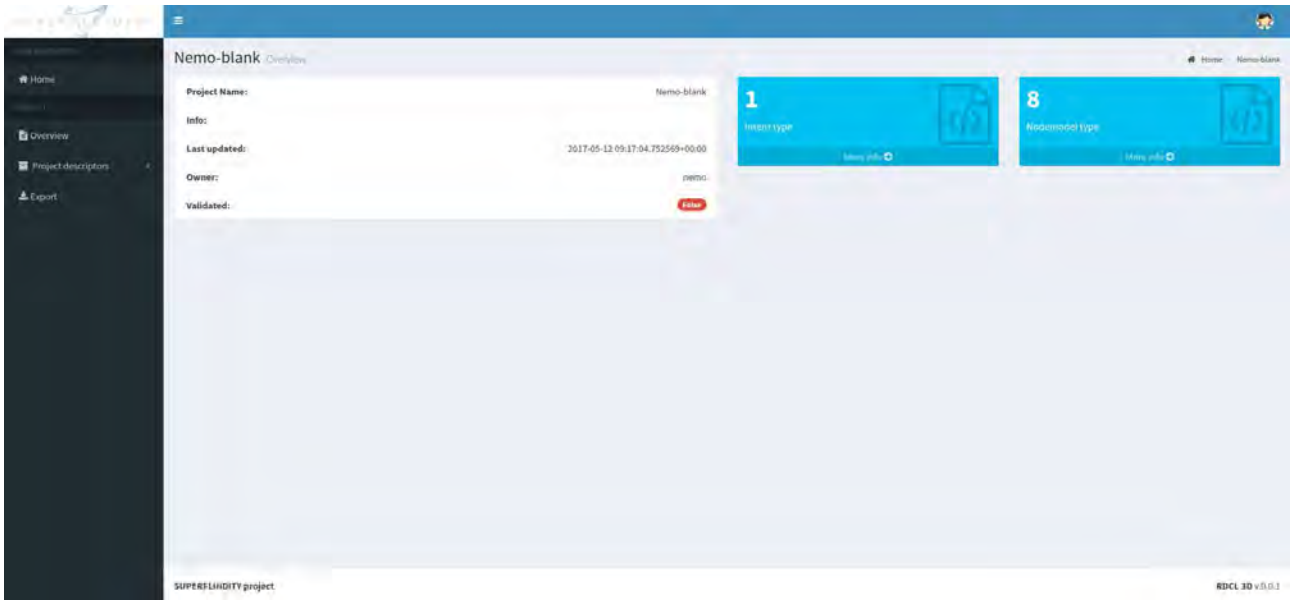

Figure 34: Overview of NEMO Project in the RDCL 3D tool

With the RDCL 3D tool, it is possible to create Nodemodels and Intents from files or direct text input, editing them later like in and, in a near future, visualize a graph of the described network.

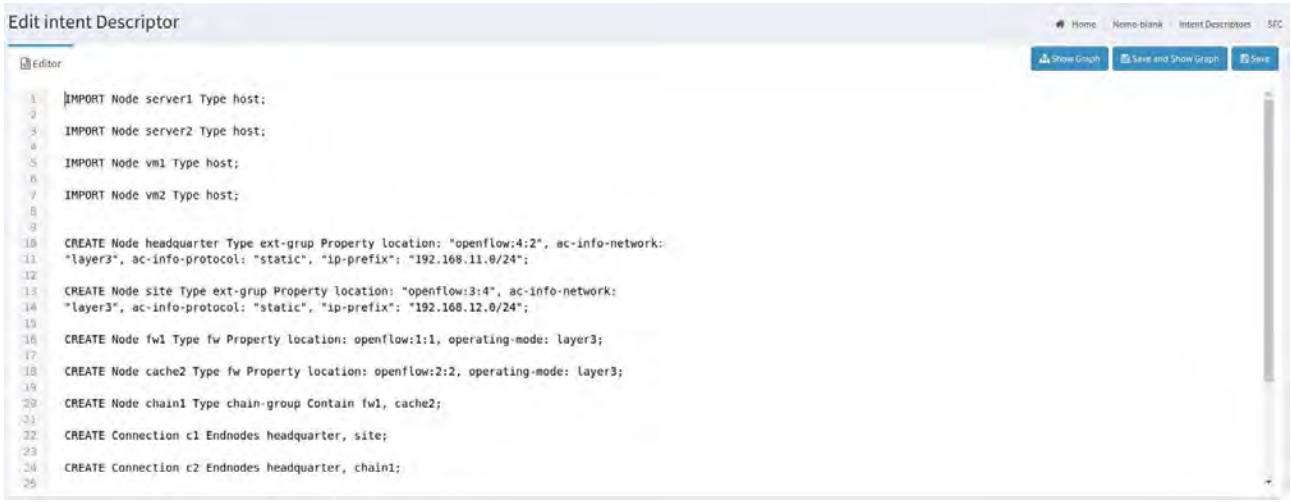

Figure 35: Editing a NEMO intent using the RDCL 3D tool

However, despite some clear advantages NEMO could provide to the Superfluidity architecture, after the overall exploration the consortium partners decided not to proceed further with its integration to the overall architecture.

6 Conclusions

Based on our detailed review of the Network Service template, VNF descriptor and VNF packaging specification docs, the information model entities support decomposition up to the VNF component (VNFC). However, as we have described previously, the RFB model proposed by Superfluidity is more general than the NFV model, since it can recursively support an arbitrary depth of sub-classing and decomposition. As a consequence, we will have to amend the NFV information model to support the RFB model of Superfluidity. Moreover, similar changes need to be introduced to the MANO reference points (APIs) as described in the corresponding section of this deliverable, or the corresponding toolchain of the NFVO.

Another important aspect for consideration was the concrete textual representation of the information model elements. Practically speaking, this specifies the syntax of the files that will represent the NS templates, VNF descriptors, VNF forwarding graph descriptors, etc. Based on the current practice of NFVOs and VNFMs, which seem to also be adopted by the ETSI standardisation efforts, the elements of the NFV Information Model are represented using a YAML syntax, and increasingly compliant with the OASIS TOSCA Simple Profile for NFV. According the original discussions, and under the assumption that the NFVOs of choice are aligned with this decision, Superfluidity decided to adopt YAML-based file formats, and, to the extent possible, TOSCA-based. Given the abovementioned requirement for a recursive composition of RFBs, the top-level RDCL was originally envisioned to utilize the NEMO language however, after the overall exploration the consortium partners decided not to proceed further with its integration to the overall architecture.

Finally, the descriptors of RFBs, for which formal analysis of correct behaviour is available, can be extended to include the relevant representation. In that context, describe specific extensions to optionally embed the domain-specific language adopted by Superfluidity, SEFL. This will facilitate automatic execution of the symbolic execution engine, SymNet, for the subset of life-cycle transitions that would require (re-)verifying the composition of RFBs.

7 References

[1] IBM, "Hypervisors, virtualization, and the cloud: Learn about hypervisors, system virtualization and how it works in a cloud environment" [Online]. Available: http://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/cl-hypervisorcompare-pdf.pdf [Accessed: 26/04/2017]

[2] Kernel Virtual Machine [Online]. Available: http://www.linux-kvm.org/page/Main_Page [Accessed: 26/04/2017]

[3] Tselios C., Tsolis G. (2016) A Survey on Software Tools and Architectures for Deploying Multimedia-Aware Cloud Applications. In: Karydis I., Sioutas S., Triantafillou P., Tsoumakos D. (eds) Algorithmic Aspects of Cloud Computing. Lecture Notes in Computer Science, vol 9511. Springer

[4] Open Networking Foundation “OpenFlow” [Online]. Available: https://www.opennetworking.org/sdn-resources/openflow [Accessed: 20/01/2018]

[5] Red Hat, “Understanding OpenStack” [Online]. Available: https://www.redhat.com/en/topics/openstack [Accessed: 20/01/2018]

[6] Linux Foundation, “Open vSwitch” [Online]. Available: http://openvswitch.org/ [Accessed: 20/01/2018]

[7] OpenStack Project “Scope of the Nova Project” [Online]. Available: https://docs.openstack.org/developer/nova/project_scope.html [Accessed 20/01/2018]

[8] OpenStack Project, “OpenStack Bare Metal Provisioning” [Online]. Available: https://wiki.openstack.org/wiki/Ironic [Accessed 20/01/2018]

[9] Cisco, “Introduction to Cisco IOS NetFlow”, White Paper [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.pdf [Accessed: 20/01/2018]

[10] OpenStack Project “Magnum” [Online]. Available: https://wiki.openstack.org/wiki/Magnum [Accessed 20/01/2018]

[11] OpenStack Project Wiki "Neutron" [Online]. Available: https://wiki.openstack.org/wiki/Neutron [Accessed 20/01/2018]

[12] Open vSwitch Documentation [Online]. Avaialble: http://docs.openvswitch.org/en/latest/intro/why-ovs/ [Accessed 20/01/2018]

[13] Open vSwitch "Classic OvS scenario" [Online]. Avaialble: https://docs.openstack.org/liberty/networking-guide/scenario-classic-ovs.html [Accessed 20/01/2018]

[14] sFlow Protocol [Online]. Available: http://www.sflow.org/ [Accessed 20/01/2018]

[15] HAProxy [Online]. Available: http://www.haproxy.org/ [Accessed 20/01/2018]

[16] HAProxy Documentation [Online]. Available: http://cbonte.github.io/haproxy-dconv/1.7/intro.html [Accessed 20/01/2018]

[17] OpenStack Documentation “HAProxy” [Online]. Available: https://docs.openstack.org/ha-guide/controller-ha-haproxy.html [Accessed 20/01/2018]

[18] OpenStack Project “Neutron/LBaaS” [Online]. Available: https://wiki.openstack.org/wiki/Neutron/LBaaS [Accessed 20/01/2018]

[19] OpenStack Project “Load Balancer as a Service (LBaaS)” [Online]. Available: https://docs.openstack.org/mitaka/networking-guide/config-lbaas.html [Accessed 20/01/2018]

[20] OpenStack Orchestration [Online]. Available: https://wiki.openstack.org/wiki/Heat [Accessed 20/01/2018]

[21] Heat Orchestration Template (HOT) specification [Online]. Available: https://docs.openstack.org/developer/heat/template_guide/hot_spec.html? [Accessed 20/01/2018]

[22] YAML Ain't Markup Language (YAML) [Online]. Available: http://yaml.org/ [Accessed 20/01/2018]

[23] Puppet [Online]. Available: https://puppet.com/ [Accessed 20/01/2018]

[24] Chef [Online]. Available: https://www.chef.io/chef/ [Accessed 20/01/2018]

[25] OpenStack Mistral Wiki [Online]. Available: https://wiki.openstack.org/wiki/Mistral [Accessed 20/01/2018]

[26] Yet Another Query Language (YAQL) [Online]. Available: https://pypi.python.org/pypi/yaql/1.0.0 [Accessed 20/01/2018]

[27] Jinja 2 : Python Templating Engine [Online]. Available: http://jinja.pocoo.org/docs/dev/ [Accessed 20/01/2018]

[28] OpenStack Telemetry Wiki [Online]. Available: https://wiki.openstack.org/wiki/Telemetry [Accessed 20/01/2018]

[29] OpenStack Telemetry Documentation [Online]. Available: https://docs.openstack.org/admin-guide/telemetry.html [Accessed 20/01/2018]

[30] OpenStack Ceilometer Developer Documentation [Online]. Available: https://docs.openstack.org/developer/ceilometer/ [Accessed 20/01/2018]

[31] ETSI Industry Specification Group (ISG) for NFV Home Page, http://www.etsi.org/technologies-clusters/technologies/nfv

[32] ETSI GS NFV-IFA 013, "NFV MANO, Os-Ma-Nfvo reference point - Interface and Information Model Specification", V (2016-10)

[33] ETSI GR NFV-IFA 015, "NFV MANO Release 2; Report on NFV Information Model" V2.1.1 (2017-01)

[34] ETSI GS NFV-IFA 014, "NFV MANO, Network Service Templates Specification", V2.1.1 (2016-10)

[35] ETSI GS NFV-IFA 005: "NFV MANO, Or-Vi reference point - Interface and Information Model Specification". V2.1.1 (2016-04)

[36] ETSI GS NFV-IFA 006: "NFV MANO, Vi-Vnfm reference point - Interface and Information Model Specification". V2.1.1 (2016-04)

[37] ETSI GS NFV-IFA 007: "NFV MANO, Or-Vnfm reference point - Interface and Information Model Specification". V2.1.1 (2016-10)

[38] ETSI GS NFV-IFA 008: "NFV MANO, Ve-Vnfm reference point - Interface and Information Model Specification". V2.1.1 (2016-10)

[39] ETSI GS NFV-IFA 002 "NFV Acceleration Technologies, VNF Interfaces Specification", V2.1.1 (2016-03)

[40] ETSI GS NFV-IFA 003 "NFV Acceleration Technologies, vSwitch Benchmarking and Acceleration Specification " V2.1.1 (2016-04)

[41] ETSI GS NFV-IFA 004 "NFV Acceleration Technologies, Management Aspects Specification" V2.1.1 (2016-04)

[42] ETSI GS NFV-IFA 010: "NFV MANO, Functional requirements specification" V2.1.1 (2016-04)

[43] ETSI GS NFV-IFA 010: "NFV MANO, Functional requirements specification" V2.2.1 (2016-10)

[44] ETSI GS NFV-IFA 011: "NFV MANO, VNF Packaging Specification". V2.1.1 (2016-10)

[45] OpenMANO [Online]. Available: https://github.com/nfvlabs/openmano [Accessed 20/01/2018]

[46] Canonical, Juju [Online]. Available: https://www.ubuntu.com/cloud/juju [Accessed 20/01/2018]

[47] Rift.io, Rift.Ware [Online]. Available: https://www.riftio.com/tag/rift-ware/ Accessed 20/01/2018]

[48] OSM, "Juju Installation in R0" [Online]. Available: https://osm.etsi.org/wikipub/index.php/Juju_installation_(release_0) [Accessed 20/01/2018]

[49] “TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0”, Committee Specification Draft 03, OASIS, 2016

[50] J E. Kohler, et al., “The Click modular router,” ACM Transactions on Computer Systems (TOCS), vol. 18, no. 3, 2000

[51] RDCL 3D Home Page,https://github.com/superfluidity/RDCL3D

[52] Salsano S, et al “RDCL 3D, a Model Agnostic Web Framework for the Design of Superfluid NFV Services and Components”, 3rd IEEE International Workshop on Orchestration for Software Defined Infrastructures, O4SDI at IEEE NFV-SDN conference, Berlin, 6-8 November 2017 (arXiv preprint: https://arxiv.org/pdf/1702.08242)

[53] Stoenescu R, et al “Symnet: scalable symbolic execution for modern networks”, Available: https://arxiv.org/pdf/1604.02847.pdf

[54] Superfluidity Project, Deliverable D3.1, “Final system architecture, programming interfaces and security framework specification”

[55] OpenStack Octavia [Online] Available: https://wiki.openstack.org/wiki/Octavia [Accessed 20/01/2018]

[56] Red Hat Ansible [Online], Available: https://www.ansible.com/[Accessed 20/01/2018]

[57] “TOSCA Simple Profile in YAML Version 1.0”, OASIS, 2016

[58] Superfluidity Project, Deliverable D4.1 “Hardware Selection, Modelling and Profiling

[59] Superfluidity Project, Deliverable D6.1 “System Orchestration and Management Design and Implementation"

[60] ETSI, “Mobile Edge Computing (MEC); Framework and Reference Architecture” ETSI GS MEC 003 v1.1.1 (2016-03), March 2016.

[61] SmartBear Software, “SWAGGER Home Page,” October 2017. [Online]. Available: https://swagger.io/.

[62] Superfluidity Project, Deliverable D5.2 “Function Allocation Algorithms, Implementation and Evaluation"

[63] Docker Inc., “Docker” [Online]. Available https://www.docker.com/ [Accessed 20/01/2018]

[64] Open Networking Foundation, “Open Network Operating System” [Online]. Available https://wiki.onosproject.org [Accessed 20/01/2018]

APPENDIX – OSM implementation evaluation

Open Source MANO Release One Deployment steps can be summarized as follows:

- Pre-requisites
  - Configure VM
  - Install Ubuntu 16.04 LTS
  - Configure Networking
  - Configure LXD container operation
- Main OSM Installation Process
  - Obtain/execute installation script
  - Access the UI using Chrome to avoid additional SSL certificate configuration issues

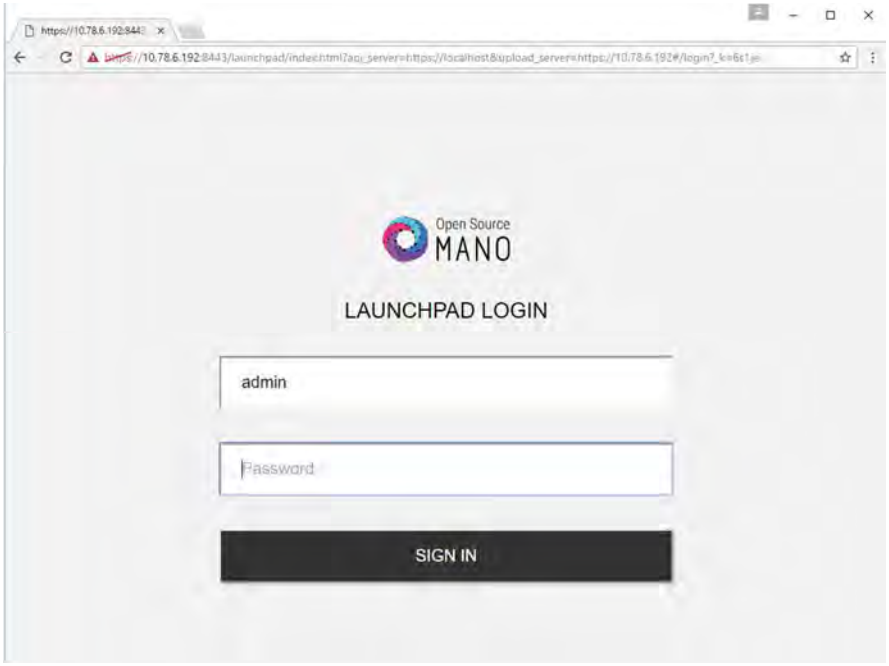

Figure 36: OSM Release One Login Screen

During the evaluation phase of OSM for the Superfluidity project, we decided to install a later OpenStack distribution than Mitaka, in order to efficiently allocate possible compatibility issues.

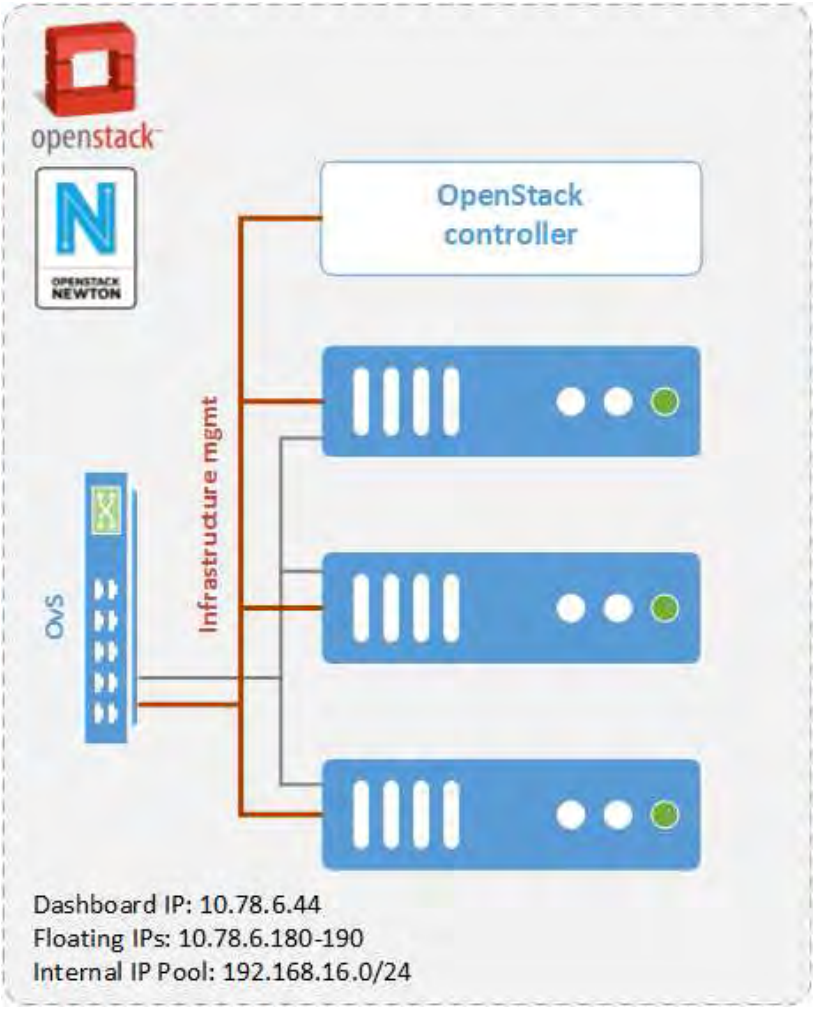

Figure 37: OpenStack Newton Deployment for OSM-VIM testing

Once the OpenStack node is deployed, additional steps are needed for configuring the node to be used by OSM, as well as attach it to the OSM itself. In particular, it is required to access the OpenStack node, create the management network along with a valid tenant/user, upload the images that will later be used by the OSM through VNFs for spawning VMs, and modify the default security group. It is advised to save all preliminary testing images under the /mnt/powervault/virtualization directory, since it is tested to be identified seamlessly by the VNFD provided by the OSM community for validation purposes.

Certain configuration steps are needed on the OSM node as well. In particular one must access the RO container, export the tenant and use CLI commands to add the OpenStack node. However, two issues are present in OSM Release One, that somehow might lead to confusion. Despite the available menu for adding OpenStack (VIM in general) nodes via the GUI, the whole process fails. At the moment, CLI commands are the only functional option for adding VIMs in the OSM node and even so, added datacenters do not appear in the GUI at all. Executing CLI commands for discovering attached datacenters reveals the available VIMs, however GUI and CLI inconsistency is present throughout the specific release. Figure 38 presents the integrated OpenStack Newton / Open Source

MANO Release One testbed in which the Network Service of Figure 39 consisting of two CirrOS-based VNFs connected by a simple VLD will be deployed. In order to proceed (i) download the required VNF and NS packages from this URL: https://osm-download.etsi.org/ftp/examples/cirros_2vnf_ns/, (ii) obtain a CirrOS 0.3.4 image (iii) upload the image into the /mnt/powervault/virtualization directory of the OpenStack node and (iv) onboard the image into OpenStack using the following command:

```
openstack image create --file="./cirros-0.3.4-x86_64-disk.img" --container-format=bare --disk-format=qcow2 cirros034
```

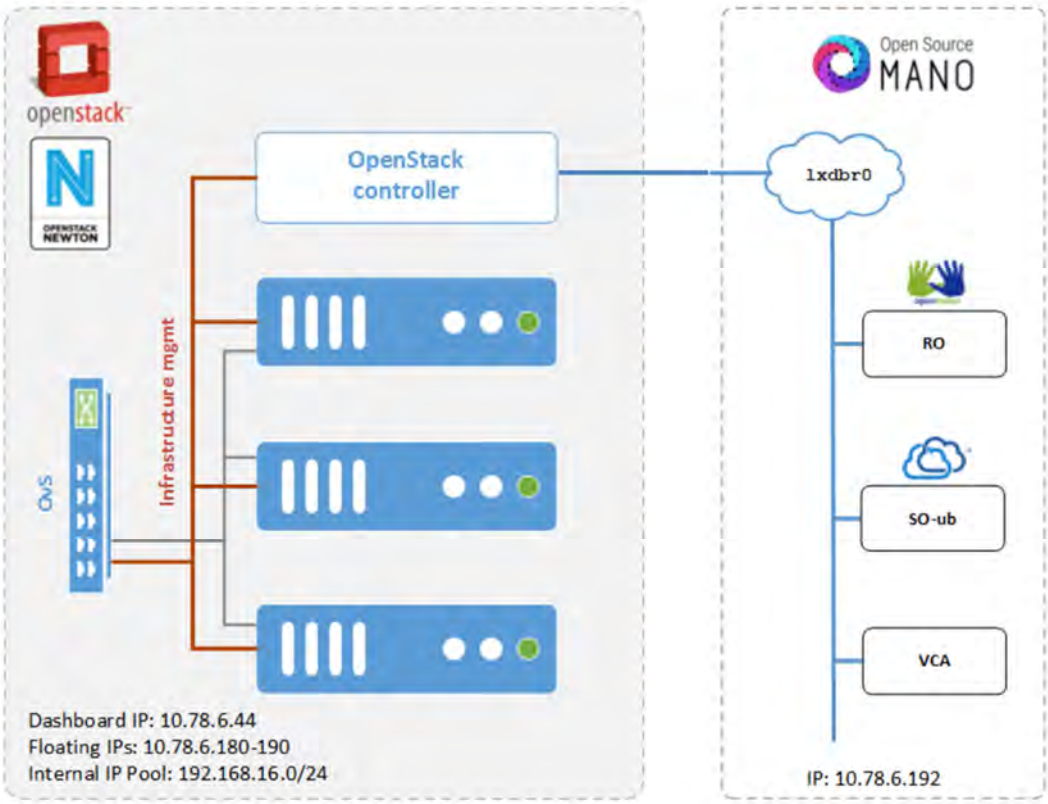

Figure 38: OpenStack Newton - OSM Release One testbed

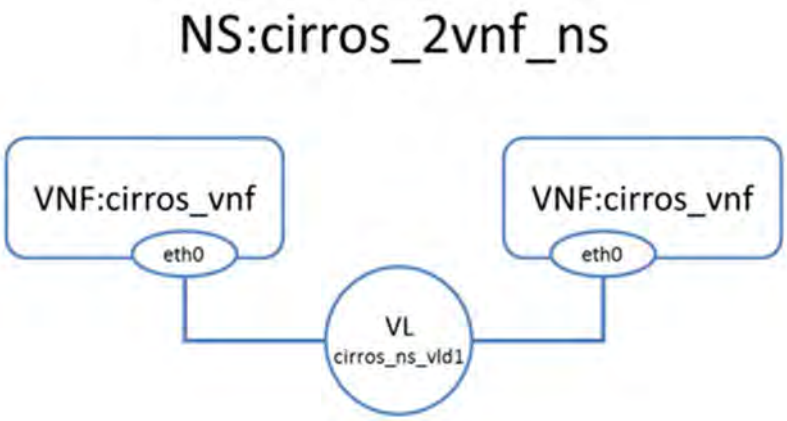

Figure 39: Validation Network Service deployment

Onboarding VNF and NS

In order to successfully onboard VNF and NS it is required to (i) access the OSM node GUI using Chrome, (ii) go to CATALOG > Import, (iii) select the element using the corresponding tab as shown in Figure 40 VNFD for VNF or NSD for NS, (iv) Drag and drop the corresponding package in the importing area, cirros_nfv.tar.gz for VNF or cirros_2vnf_ns.tar.gz for NS.

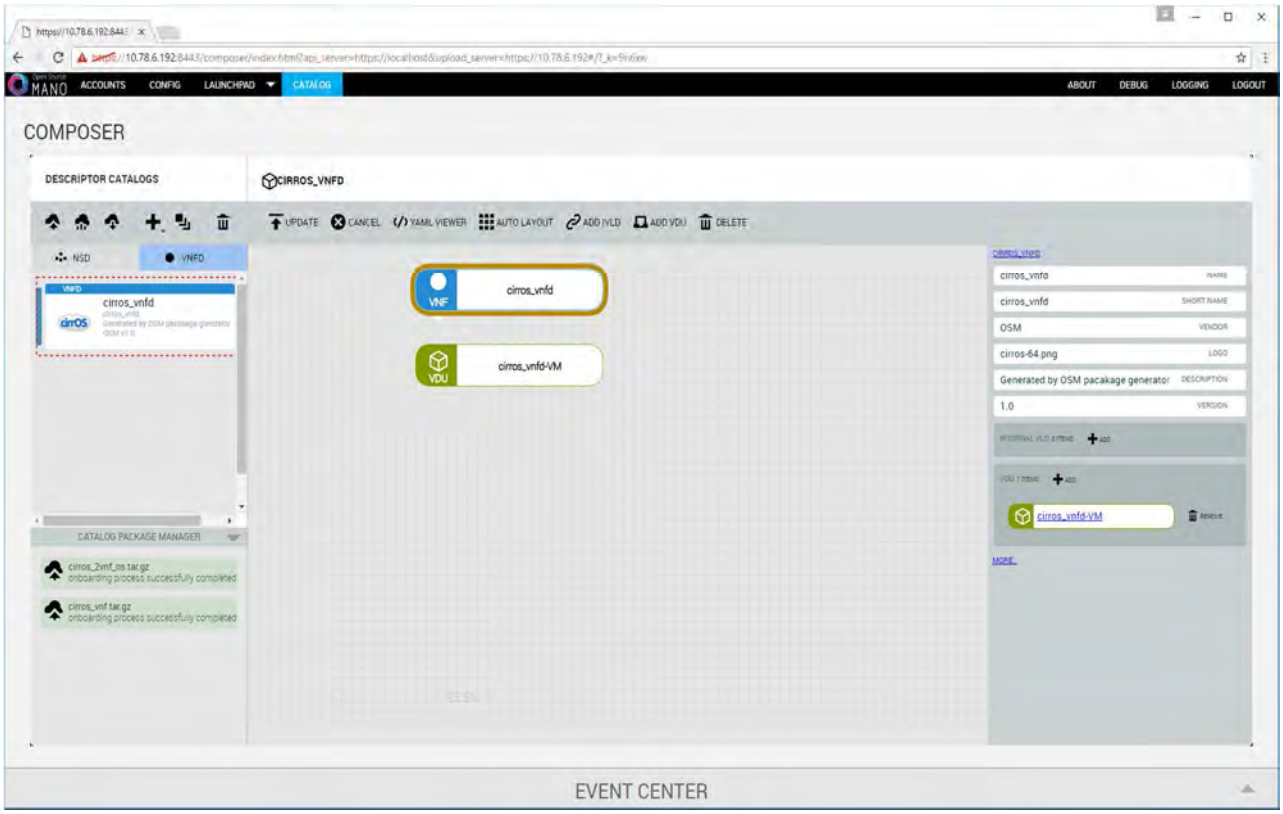

Figure 40: VNFD Onboarding

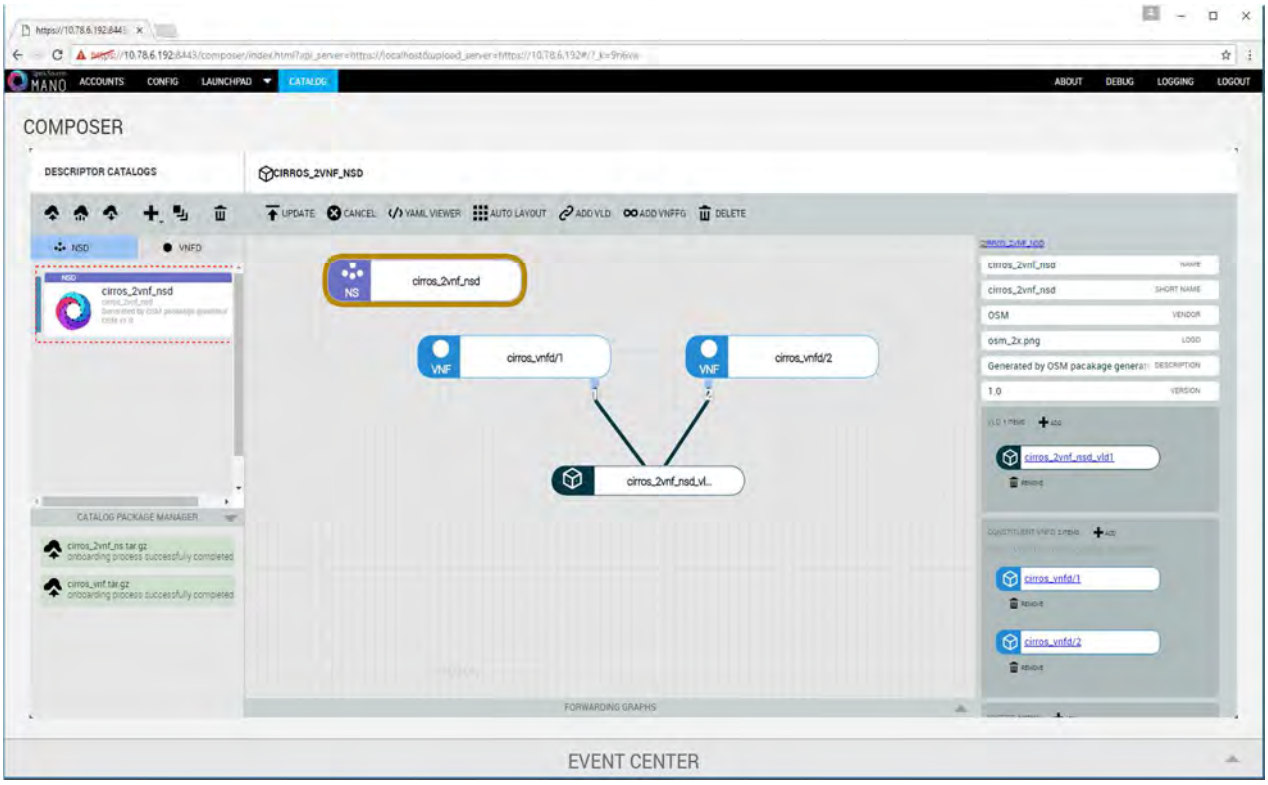

Figure 41: NSD Inspection

Instantiating the NS

For instantiating the NS (i) go to Launchpad > Instantiate, (ii) Select the NS descriptor to be instantiated and click Next (Figure 42), (iii) add a name to the NS instance and click Launch (Figure 43).

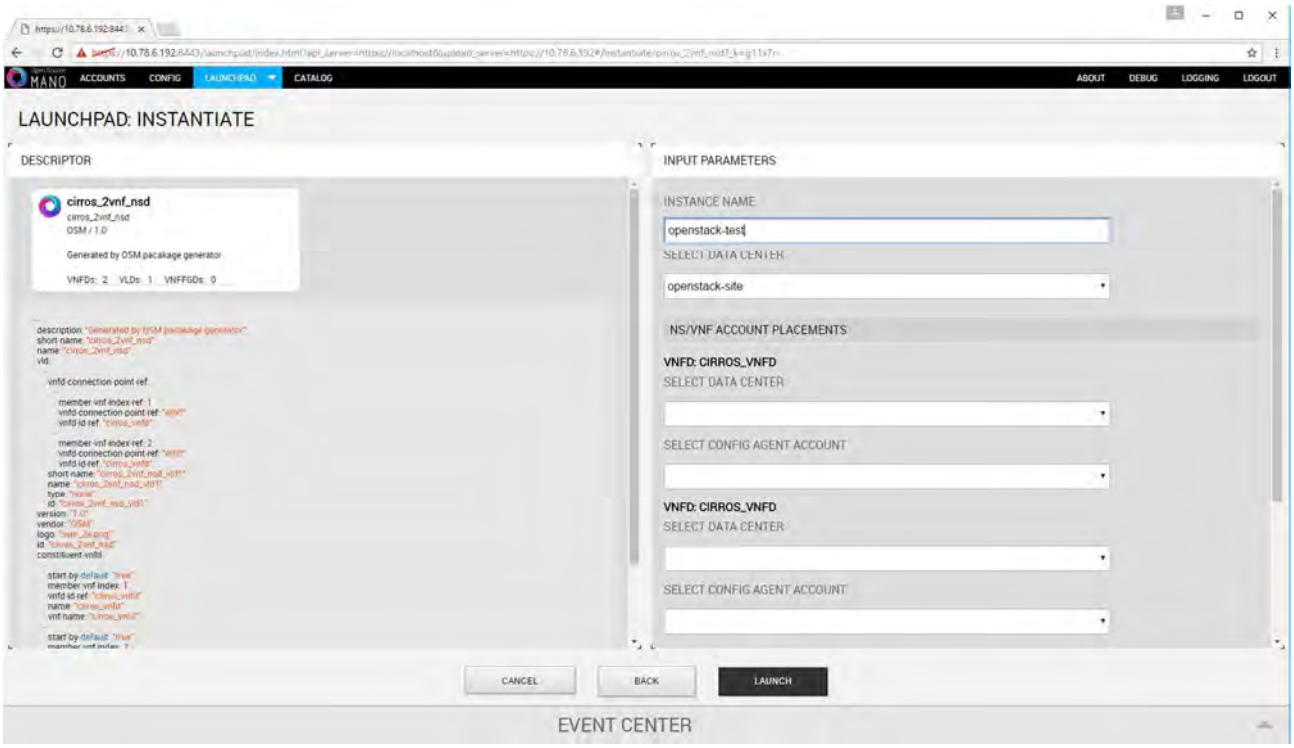

Figure 42: NS Instantiation

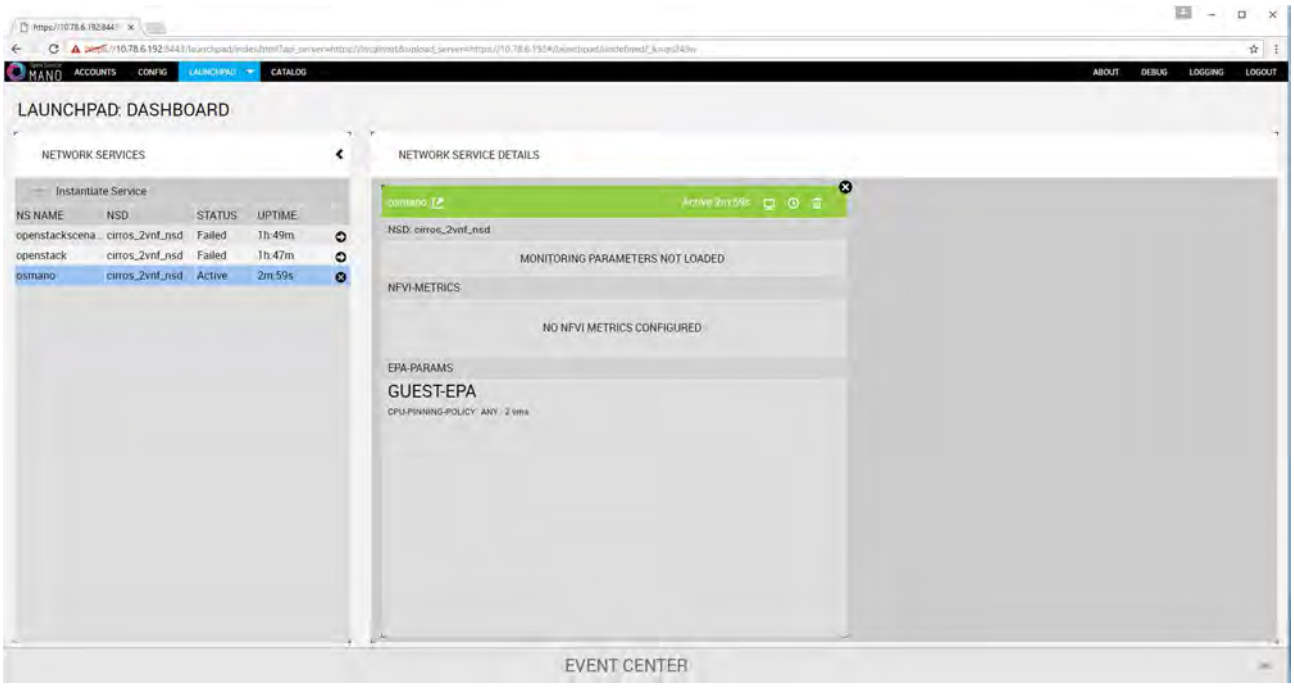

Figure 43: Successfully Instantiating the NS from the OSM Node perspective

Any successful Network Service instantiation made via the Open Source MANO Orchestrator is also visible through the Network Topology graph of the attached OpenStack VIM, as shown in Figure 44, 27 and 28 for the validation NS scenario previously deployed.

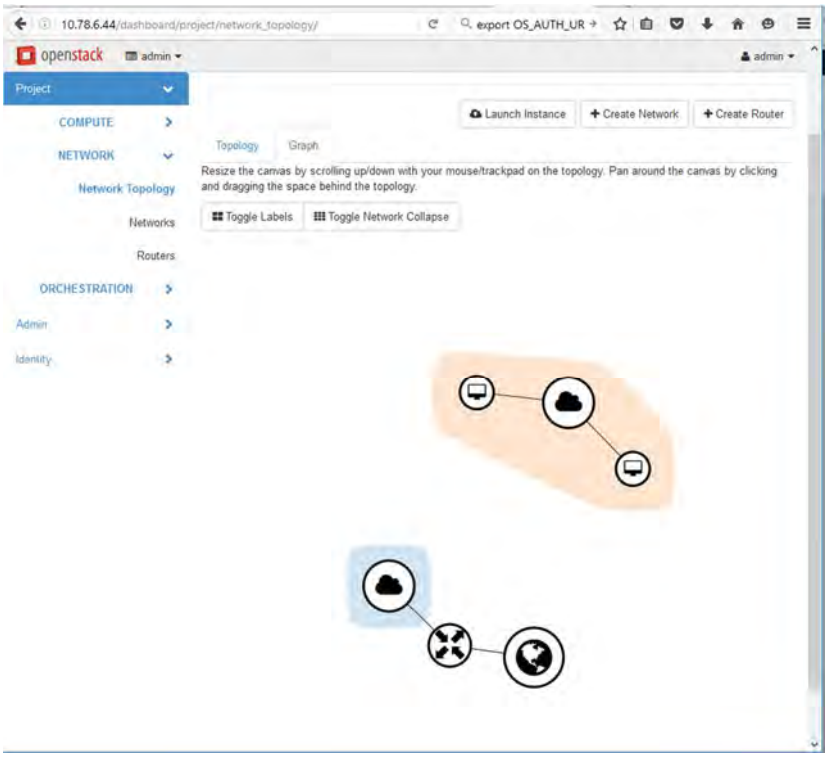

Figure 44: OpenStack Graph representing the validation NS deployment

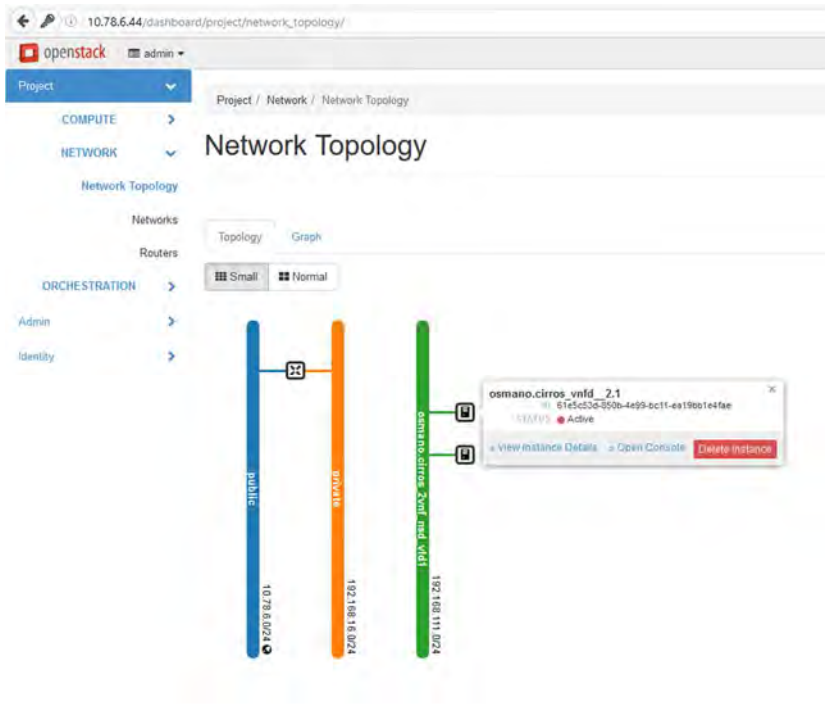

Figure 45: OpenStack Network Topology representing the validation NS deployment

Appendix– Collaboration with 5G-PPP

Superfluidity project is contributing mainly in 5G Architecture Working Group (WG) and Software Network WG, besides to coordination with 5G-PPP Steering Board and Technology Board.

This deliverable introduces a set of innovations that the project built in the NFV, SDN and MEC areas. The results have been described in the recent version of the 5G Architecture white paper produced by the 5G Architecture WG and released in the Mobile Wireless Congress 2018, in which Superfluidity is very active. Mainly, Section 5.1.2 of the white paper refers to the cloud and virtualization technologies that Superfluidity is advancing (Unikernel, container). Section 4, related to the Physical infrastructure and deployment, exploits one of the project results concerning the integration of MEC and Cloud RAN and the traffic offload implementation.

The work of Superfluidity in SDN and NFV areas, described in this document, is introduced to Software Network WG and referred by some of phase 2 projects which are now focusing on container management in 5G landscape.